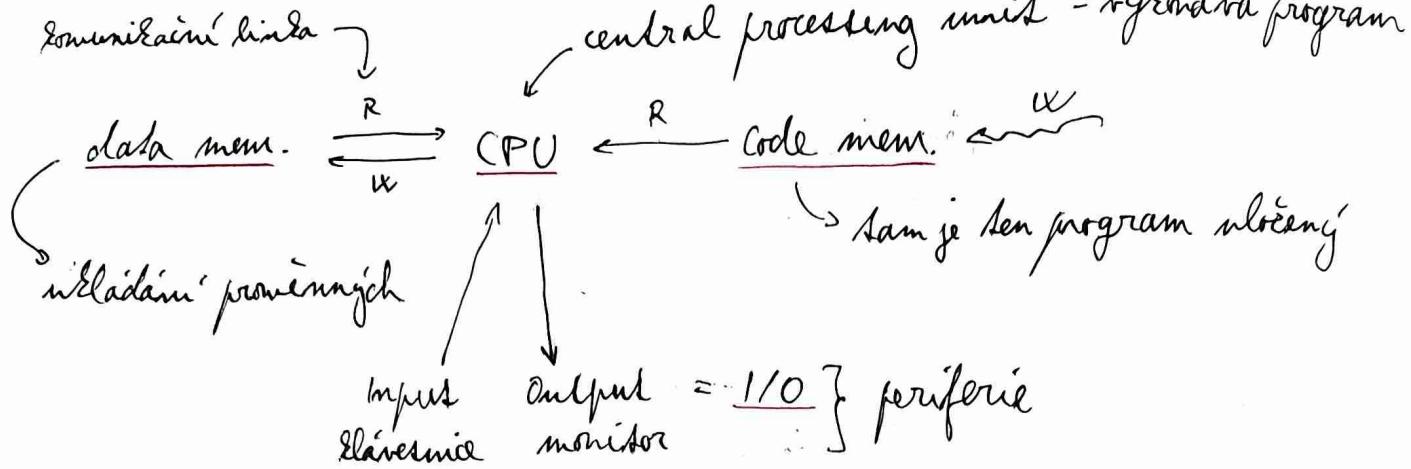


Harwickská architektura



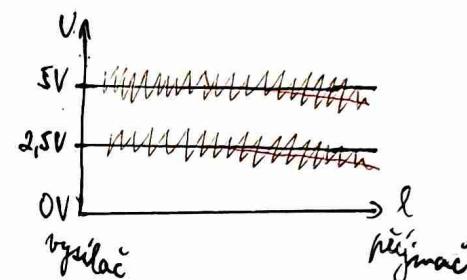
Representace nejavornejších celých čísel

Analogový přenos dat

→ checeme $0, 1, \dots 1000$

→ zvolíme měření U : $0 = 0V, 1000 = 5V$

$$\hookrightarrow 500 = 2,5V$$



→ vodiče mají odpor, ten se mimo s rezistor

→ elong. indukce: vodiči nejsou průhlední → elong. pole → na dalších vodičích se indukuje napětí → sum

→ na přijímací výměně jen přibližné hodnoty na vysílaci → není to deterministické

Digitalní přenos dat - binary digits = bits = b

- $0V = 0, 5V = 1$

- uprostřed je nějaká sedlá rovina - nad ní 1

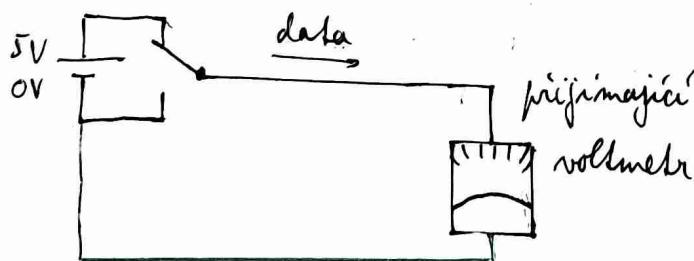
- počítáme s rozumujícím součtem | pod ní 0

Sériový digitální přenos

- posílání několika binárních čísel

- přijímací musí do napětí nějak měřit

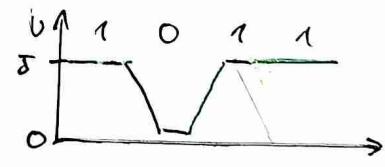
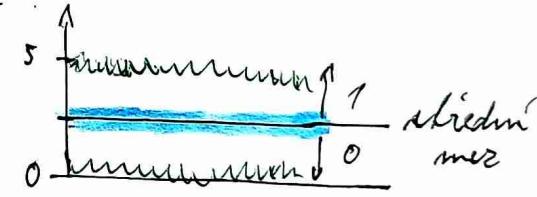
vysílač - volí $5V/0V$



Zemní vodič/GND

↳ U se měří mezi 2 body

↳ posílajeme referenční vodič



časový diagram
(timing)

Přenos dat probíhá po komunikační lince

Diferenciální píenos

diferenciální páis

→ alternativne máme dva datové rodice

- data 1 - píenos dat

- data 2 - kam se generuje spáne mapeti

→ mapeti neměníme vráci remínu rodici,

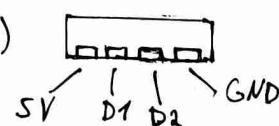
ale vráci sobě → přijimají vráci data 1 - data 2 : $> 0 \Rightarrow 1, < 0 \Rightarrow 0$

→ $0 - 5V \Rightarrow 1, -5 - 0V \Rightarrow 0 \rightarrow$ rozhla se úrovní sítma, kterou sneseme

→ na obou rodicích vzniká něméně stejný sítim, když se odecste

→ dneska nejbežnější způsob píenosu

USB (Universal Serial Bus)

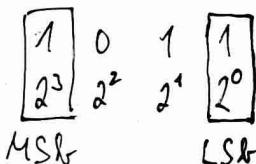


$5V + GND$ slouží k napájení
 $D1 + D2$ dif. píenos dat

Píenos binárních čísel

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

→ je třeba se dohodnout, kdyžm směrem bity čísm



← MSb - first

1101 ← Lsb - first
 $2^0 2^1 2^2 2^3$

→ je třeba dohodnout píenos dílen bita - jak dlouho trvá signal 0/1

→ definuje se to píenosovou rychlosdi $\rightarrow \Delta t$

- bity za sekundu - bps
 - symboly za sekundu - baud
- } pro uvedené případy platí $bps = baud$

→ kdy m' příjemce kontroloval mapeti?



- nejlepší když je uprostřed

→ problém: asynchronní hodiny - obě strany se musí shodnout kde je prostředek bity

→ komunikační linka může být ve třech stavech - 0, 1, idle - nic se nepřenáší

- floating star = stav s vysokou impedancí / Hi-Z - ta linka nem' rafgenu' nikam
⇒ píenosí se jen sum

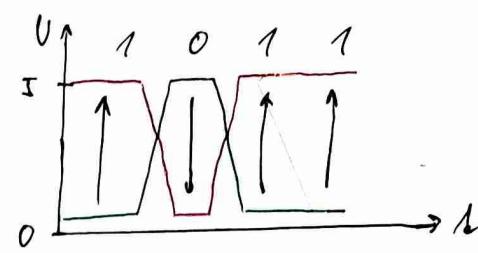
• 3-stavová logika - 0/1/1 idle definovaný jako floating stav.

• 2-stavová logika - 0/1/idle := 0 nebo 1

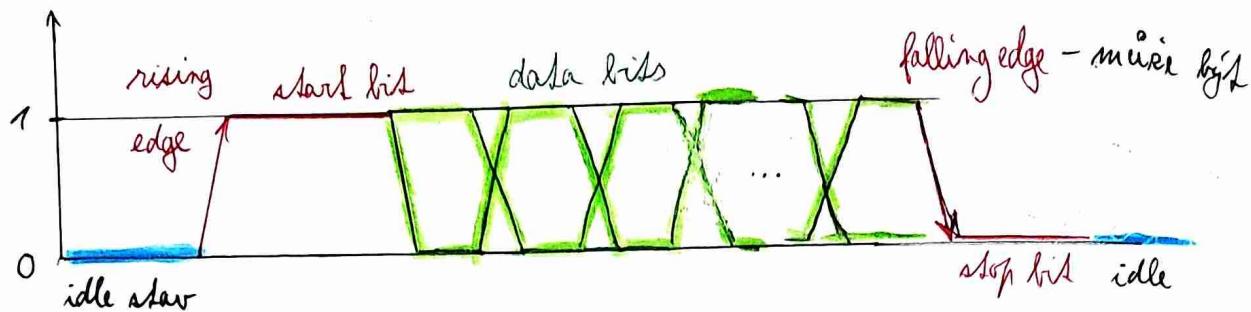
→ když se linka zapne, pak rozhla se idle (0) ⇒ příjemce n'ví, co je idle

→ začátek vysílání: start condition = rising edge (0 → 1)

→ n'ví rázne slukací píenos, pak použit start bity (1) - je třeba se domluvit kolik



+ nejdelší sedláčka



→ na start conditioň se sesynchronizují hodiny - rádiové hodiny nejsou dobrým

| | | | | | | → na nějakou dobu se korela asynchronuje

→ celom přenášíme N bliků po X blikových zásech v N/X přenosech

$$\Rightarrow \text{drehmom } X := 8 \text{ b} = 1 \text{ kgse} = 1 \text{ B}$$

\rightarrow stop condition := Edyří přeneseme právě X bitu

→ for stop condition jdeine do 0 do idle start = stop bity

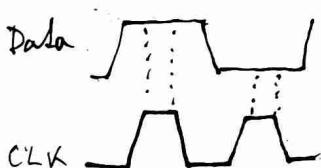
⇒ myn' mire ráčit nový pries nosť linka rukávev idle starn

$$\left. \begin{array}{l} 8 \text{ data bits} \\ 1 \text{ start bit} \\ 1 \text{ stop bit} \end{array} \right\} \underline{20\% \text{ overhead}} \rightarrow 1000 \text{ baud} = 800 \text{ bps} \quad \text{RS-232}$$

↳拙い書き方

- přenos pomocí hodinového signálu

- statický signál 010101010101...
 - fiktivně ale sponí 1 další vodič
 - hodinový signál bude může generovat využívající nebo nýkdy externí zdrojem



- Edýr je na čísle 1, tak máme ménířit
 - harnarovi je snadnější dle korál kralyc
 - rising edge \rightarrow méníříme
 - falling edge \rightarrow nemáme ménířit

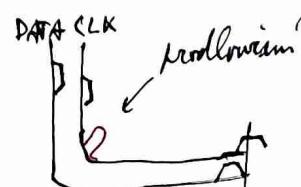
- datový signál má polohovou frekvenci - nevyužíváme plné hmity te. Technologie

⇒ Somato hyper IgM se richa SDR (Single Dara Rake)

- alternativa: libovolná brana = mřížme \Rightarrow DDR (Double Data Rate)

- ⊕ minimální overhead, neomezená délka písem

- (2) Edgerton by zrodil rizné dloně, takže nemůžou synchronizovat



- clock recovery system - neponává clock signal
 - původně: synchronizace hodin při start condition
 - návod: synchronizace hodin při bráně
 - problém: jenom hnedně kombinace 00000000 → bude tam 256 bezlých komb.
 - řešení: 8bitů - 256 komb. 10bitů - 1024

00000000	000000000 X
10110010	0101100100
:	:
11111111	1000101001

} 8b/10b encoding
mapovací tabulka

- 10bitů přidáme po 8bitových kusech - pojď 20% overhead USD
- používá se třeba 72bit/932bit ale neomezená délka přenosu

• Zde se to používá

- 1) omezená přenosová délka RS-232 linka
- 2) hodinový signál I²C linka
- 3) clock recovery USB linka

• Obojsměrný přenos

- Simplexní linka - jednosměrný přenos
 - Duplexní linka → half-duplex - výzdy → jindy ← nepraktické
full-duplex - obojsměrná linka ⇒ dvě simplexní linky
- RS-232

• RS-232 linka / digitální sériová

- full-duplexní linka se dvěma simplexními kanály, 8-bitové přenosy
- pro přenos 3 vodice: Rx (receiver) Tx (Transmitter) GND (společná zem)
- využívaly ji např. staré sériové myši
- out of band signály - další vodice
 - ↳ reprezentují nejaky stav: něco platí 1 ... přestane to platit 0 ... 1 ... 0
 - výzdy se taky definují v invertorové logice: \overline{SIG} , #SIG, /SIG, !SIG
 - ↳ power out reprezentuje 1: vše ok 0: dochází k proru
- ma RTS a DTR je výzdy 1 - blokne napájení myši: GND \Rightarrow napájení myši

Komunikační protokol / formát

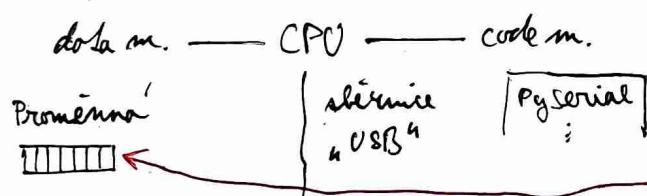
- chceme poslat aktuální datum - řádkem RTC (Real Time Clock)

MSB 0101110001 110011001100 } packet prenos - protokol ríla, jasť vypadá
den mesiac rok }
↑
RSB - first → RS-232: [100DDDDD][MMMMRRRR][RRRRRRRR] - napr

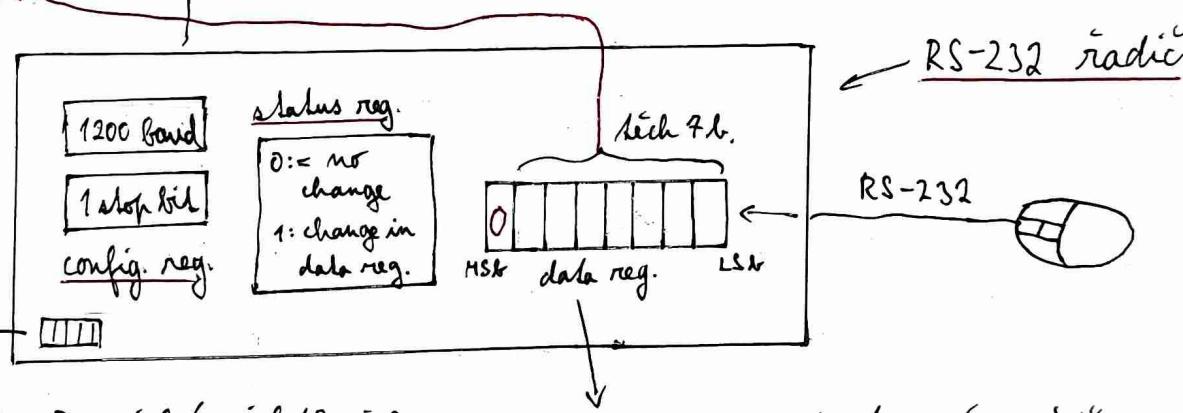
- mielime sieba najdiť poslat konštantu, ktorá ríla, jesli nasleduje čas alebo datum - nie musí byť v tom komunikáciu protokolu
- Edzbyčkom ukázali fo RS-232 preniesť 17-bitový packet: 3 byty \Rightarrow 3 prenosy

Komunikační protokol RS-232 súvisiace myši

- používá 7-bitové byty \rightarrow potreba je prekladat do 8b



\rightarrow neumiem priamo komunikať s RS-232
 \Rightarrow nuzajeme řadic / controller, ktorý
to má \rightarrow komunikácia s myšou



Python: Pyserial / serial / Serial

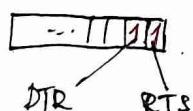
- \rightarrow do objektu Serial dáme aj config.info *
- \rightarrow keď ten RS-232 řadic nejak jednotl. identifikujeme
- \rightarrow Serial.open() do nich config. registru zapíše *
- \cdot read() \rightarrow prečíta status reg., akad tam není 1 \rightarrow keď nacierte však datoveho registru a status reg. sa zmiení na 0
- \rightarrow nastavení timeoutu: keďže je po určitej dobe všetko sivé, tak sa fa, ak .read() za chvíľu ztmie

Regist = malá slavová pamäť

Data registr - Edzí v nem sva uložená data, sterá prijima / spracováva

- ta myš má najviac 4-bitový packet = 4 RS-232 prenosy

- řadic overáda out-of-band signály RS-232 linky \Rightarrow má v sebe control register



\rightarrow jednotlivé bity odpovedají hodnotám tých OOB signálu

→ Edgy DTR a RTS nastavíme na 0, tak se myš vypne \Rightarrow bude ji resevoirat

→ inicjalizacíu' pacet - Edyť sa myš pripraviť, tak na rácielen posilať trhle

- funkce < blokující - `read(1024)` - nic nevráti dokud nepřijme 1024 B
 neblokující - `read(1024)`
 $+ \text{sincos}(0,5)$ - počet 0.5 s nic nepřijde, tak vrátí cokoli

• Hexadecimální soustava

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

$$\underline{23}_{16} = \$23 = 23h = \emptyset \times 23 = 2 \cdot 16 + 3 = 35$$

$$\begin{array}{r} \overbrace{0 \quad 0} \\ \hline 65536 \end{array}$$

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

leading zeros

- m-bit
 $0-2^m-1$
 - 4-bit \leftarrow 1 hex. císlice rávne 4 bity
 $0-15 \Rightarrow 123_{16} \sim 128$
Birod
 - $8b = 1B$
 $0-255$
 - $0 \underbrace{1 \ 2 \ 3}_{\substack{4b \\ 4b \\ 4b}} \sim 2B$
 $23_{16} = 00$

⇒ je snadno vidět, kolik paměti potřebujeme

$$738201600_{10} = \underbrace{20001000}_{\text{16}}_{\text{16}} \rightarrow 4B$$

↓
 0010 1100 0001
 ↓ ↓ ↓
 0000 0000 0000

a kde jsiou v bin jednicky

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

Bisové operace

Input: dve n-bitové čísla, Output: 1 n-bitové č.

- OR $\begin{array}{r} 1010 \\ 1100 \\ \hline 1110 \end{array}$ hodnota přírav $\begin{cases} 1: \text{zapis} 1 \\ 0: \text{obopírij hodnotu} \end{cases}$ $\Rightarrow \frac{\text{SET}}{(101)}$ → vybrane bity nastavíme na 1 a zbytek mechanice bývá

- AND
 $\begin{array}{r} 1010 \\ \times 1100 \\ \hline 1000 \end{array}$ 1: vyberu hodnotu
 & 0: rafio 0 \Rightarrow CLEAR (100) \rightarrow vybrane bity súmač

- NOT $\begin{array}{r} 1010 \\ \sim 0101 \end{array}$ $\begin{array}{r} 00001010 \\ \hline 11110101 \end{array}$ \rightarrow stáčení bitů

- $$\begin{array}{r} \text{• XOR} \\ \begin{array}{c} 1010 \\ 1100 \\ \hline 0110 \end{array} \end{array} \quad \begin{array}{l} 0: \text{despiraj} \\ 1: \text{flipni} \end{array} \quad \rightarrow \text{selektivní sláčení bitů}$$

- SHL $a \ll x = b$ $1101 \ll 2 = 0100$
 \ll $\underbrace{a}_{\text{n-bit}}$ $\underbrace{x}_{\text{n-bit}}$ \rightarrow posun x bisku & MSB.

- $$\begin{array}{l} \text{• } \frac{\text{SHR}}{\gg} \quad \underbrace{a \gg x}_{\text{m-bit}} = \underbrace{b}_{\text{m-bit}} \quad 1101 \gg 2 = 0011 \\ \qquad \qquad \qquad \rightarrow \text{push } \& \text{ LSh.} \end{array}$$

$$\underline{1 < n = 2^m}$$

→ bitore' posung (bitwise shifts)

$$\alpha_{SHRm} = \alpha_{SHLm} = 0$$

• využití binárních operací

→ v kom. f. sériové myši je bylo $B = 01LRYYXX$

→ chceme zjistit hodnotu L \Rightarrow použijeme binárovou masku pro AND

$$\begin{array}{r} \text{AND } ??L??\dots? \\ \hline \begin{array}{l} 00100000 \\ \hline 00L00000 \end{array} \end{array}$$

$$L=1 \Leftrightarrow (B \& \text{0x20}) != \emptyset$$

kontáme jen slevu 1

$$\rightarrow B_1 = \dots XX$$

$$\begin{array}{r} 00000011 \\ \hline 000000XX \end{array} \quad \underbrace{B_1 \& \text{0x03}}_{\text{SHL } 6}$$

$$B_2 = \dots XXXXXX$$

$$\begin{array}{r} 00111111 \\ \hline 00XXXXXX \end{array} \quad \underbrace{B_2 \& \text{0x3F}}$$

$$\hookrightarrow XX000000 \mid 00XXXXXX = XXXXXXXX$$

\Rightarrow OR lze použít na kombinování dvou řetězců

\rightarrow příklady

$0x7F02 \mid 0x8E18$	$0111 1111 0000 0010$	$= 1111 1111 0001 1010$
$1000 1110 0001 1000$	$F F 1 A$	
$256 \mid 0x00FF$	$0000 0001 0000 0000$	$= 0x01FF$
$1111 1111 1111 1111$		
$0x1234 \& 0x0200$	$0001 0000$	$= 0x0200$
	$0101 0100$	
$0xC9815093 \& 0x00004000$	$0001 1110$	$= 0x00004000$
	$1110 1110$	
$0xC9815093 \& 0xFFFFEFFF$	$1100 0011$	$= 0xC9805093$
	$0011 0010$	
$0xC9815093 \wedge 0xFF000000$	$1001 0010$	$= 0x36815093$
	$0010 0010$	

• binárové rotace

$$\bullet \underline{\text{ROL}} \rightarrow \text{rotace \& MSb}$$

$\underbrace{1101}_{1101} \text{ ROL } 3 = 1110$ $\text{ROL } m = \text{identita}$

$$\bullet \underline{\text{ROR}} \rightarrow \text{rotace \& LSB}$$

$\underbrace{1101}_{1101} \text{ ROR } 2 = 0111$ $0: 00000000$
 $-0: 10000000$

• Znaménková čísla

$$\bullet 8\text{-bit unsigned}$$

5	00000101
6	00000110

$$-127 - 127$$

znaménkový bit je MSb

$$\bullet 8\text{-bit signed}$$

5	00000101
-6	100000110

$$-a = a \text{ XOR } 10000000$$

$\hookrightarrow 1 \text{ znaménkový bit} + 7\text{-bit unsigned} = \text{repräsentace s explicitním znam. bitem}$

$+ := 0 \rightarrow \text{pro nezáporná čísla}$ $= \text{signed magnitude}$

$- := 1 \rightarrow \text{je signed i unsigned repr. stejná}$

\rightarrow procesor má pouze 1 bit, ráčí a odečítá unsigned č. \rightarrow bude to fungovat i pro signed?

$$-5: 10000101 \rightsquigarrow 128 + 5 = 133$$

$$-6: 10000110 \rightsquigarrow 128 + 6 = 134$$

$$133 < 134 \Rightarrow -5 < -6!$$

$$133 + 1 - 134 \Rightarrow -5 + 1 = -6!$$

\rightarrow operace pro unsigned nefungují pro signed \Rightarrow procesor by musel mít ty příslušný abstract

• jednícíký doplněk n-bitová přesnost

(ones' complement)

+ → unsigned

- → NOT(abs(a))

$\bar{S}: 00000101$

$-\bar{S}: 11111010$

$-6: 11111001$

$-\bar{S} > -6 \checkmark$

$$-\bar{S} + 1 = 11111011 = -4$$

NOT: 00000100

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

$$-127 - 127$$

→ příklad máme obecný malý: $-0 = 11111111$ edge $-a = \text{NOT}(A)$
 $0 = 00000000$

↳ edge $-a = \text{NOT}(a) + 1$: $-0 = 11111111 = 1 \underbrace{00000000}_{8\text{bit}} + 1 = 00000000$ pro 8-bit přesnost

• dvojkódy doplněk

(two's complement)

+ → unsigned

- → NOT(abs(a)) + 1

$$\left. \begin{array}{l} -a = \text{NOT}(a) + 1 \\ \end{array} \right\}$$

$S: 00000101$

$-\bar{S}: 11111011 \Rightarrow -\bar{S} = -256 + 128 + 64 + 32 + 16 + 8 + 2 + 1$

$-6: 11111010 -128 = -256 + 128$

→ unív. reprezentace $\frac{-128 \text{ až } 127}{-2^{n-1} \text{ až } 2^{n-1}}$

$-128: 10000000$

$$-2^{n-1} \text{ až } 2^{n-1} - 1$$

→ posloupnost: $\oplus > \otimes \checkmark \ominus > \oslash \checkmark$

$$+ > - \times$$

$-5 > 5 \left. \begin{array}{l} \text{potřebujeme novou operaci pro} \\ \text{processor} \end{array} \right\}$

→ sčítání, odčítání funguje

$\left. \begin{array}{l} \text{signed} \\ \text{unsigned} \end{array} \right\}$ posloupnost

↳ MSB rychláří jako znaménkový bit

príklad: $11111111 \rightarrow \text{NOT} + 1 \rightarrow 00000001 = 1 \rightarrow -1$

9b.

python: $a = 254 \quad \underbrace{00000000}_{10} \quad \underbrace{11111110}_{11} \quad \# \text{platných b.} \quad \left. \begin{array}{l} \text{unsigned 32-bit č.} \\ \rightarrow \text{max. velikost č. je } 2^{32} \end{array} \right\}$

↳ python automaticky dělá znaménková č.

→ čísla mohou být v max. oboru bytů

→ python mě nechápe na 10 254 konkr. jde int8 $\Rightarrow -2$

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

• unív. přesnosti = truncation

8-bit S 00000101

$$X = 01001101$$

$$Y = X \bmod 2^n$$

4-bit S 0101

↳ m-bit $\underbrace{01101}_{Y}$

$$0101 = 00000101 \bmod 2^4$$

$$m=5$$

$$a = \text{int16}(12)$$

$$-2 = 11111110$$

$$-128 = 10000000$$

$$1110 = -2 \checkmark$$

$$0000 \neq -128$$

$$a = \text{int8}(a)$$

- rozšíření frekvence
- beznaménkové rozšíření
(zero extension)

- naménkové rozšíření
(sign ext.) → do nových bitů malopíše MSB

4bit	0101	5	→ for - by 2r refungoval
8bit	00000101	5	
	0101	5	
	00000101	5	
	1110	-2	
	11111110	-2	

↳ pro unsigned se nefunguje : 1111 15
⇒ 1111 1111 255

→ numpy: uint → uint zero ext.

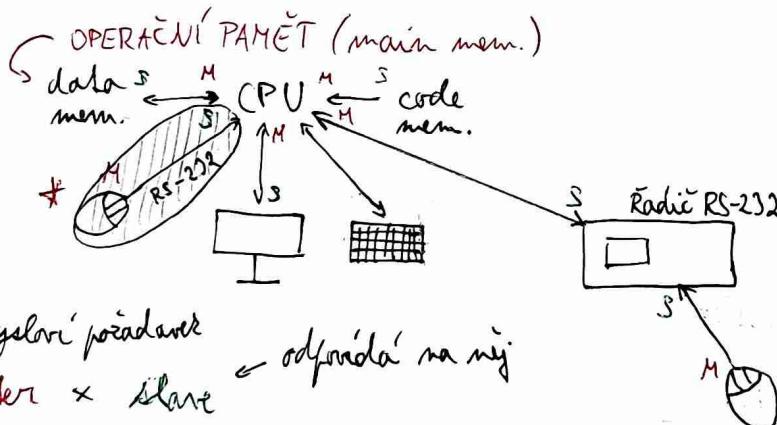
uint → int	}	sign ext.
int → uint		
int → int		

→ python bistrová negace 254 ~ 254 01111110 100000001

normalní jazyk : 0x101

python : signed 9-bit
⇒ -255 = -0xFF

• Master × slave



✓ systémový pořádek
master × slave

→ zápis dat do slave (write)

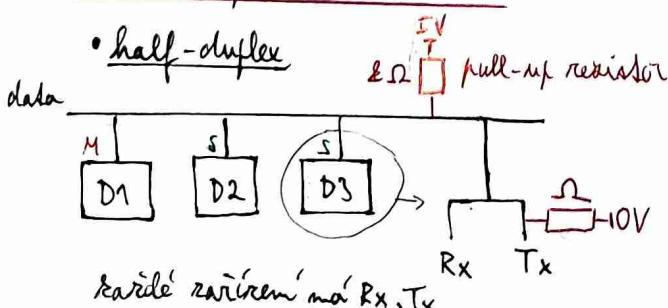
← čtení dat ze slave (read)

Na rozdíl si mohou prohodit role

* Procesor se nemůže chovat jako slave ⇒ potřebujeme ten rádiový

• point-to-point komunikací linka vede mezi 2 zařízeními ⇒ hodičí linky × CPU

• multidrop / bus / sběrnice - na 1 k. l. je připojeno více zařízení



• 0 zařízení → pull-up r. 5V ⇒ 1

• 1 zařízení

- nechce vysílat ⇒ odpoji Tx ⇒ 1

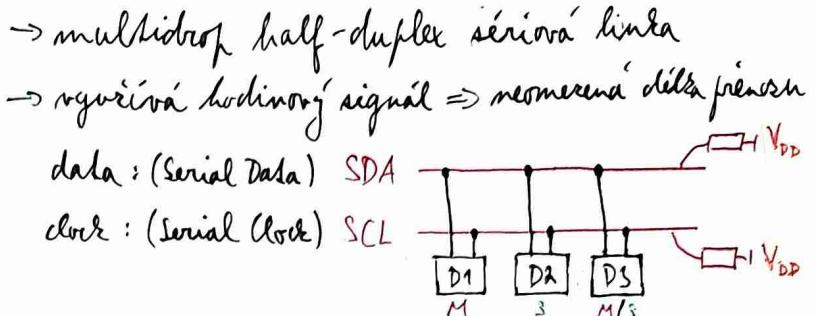
- 1 ⇒ odpoji Tx = 1

- 0 ⇒ připoji Tx ⇒ déliv nafili ⇒ stvor 0

• 2 rářírem'

	D1	D2	BUS	IDLE
AND	x	x	1	
	x	1	1	linka se
	x	0	0	chorá déle-
	{ 1	1	1	místnicky
	{ 0	0	0	
	{ 0	1	0	
	{ 1	0	0	

• I²C (Inter Integrated Circuit)



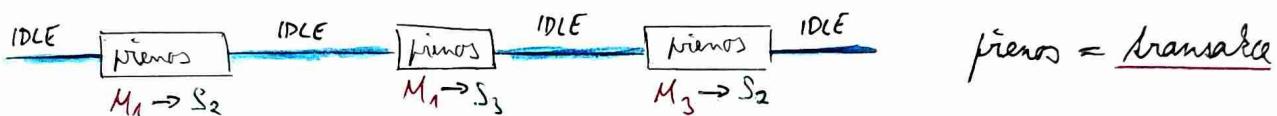
→ Napájecí napětí V_{DD} / V_{CC} - může být různé

↳ merní napětí: 0V \rightarrow 0.3V_{DD} \rightsquigarrow 0
 0.7V_{DD} \rightarrow V_{DD} \rightsquigarrow 1

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

→ I²C je multimaster

→ může být více masterů
 a rářírem' mohou mít různou



prenos = transakce

→ je to nějak vyrovnáno, aby dvě rářírem' nevyšílala rářoven'

→ SCL signál vždy generuje master asynchronní komunikace

→ IDLE: vše je odpojené \Rightarrow 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

I²C používá

bit-order

MSB-first

→ I²C používá 9-bitové byty: 8 data bits + 1 control bit - povolení ACK

0 = ACK = acknowledged

1 = NACK / NAK = negative ACK

PACKET: START B1 a₁ B2 a₂ B3 a₃ STOP

I²C write M S

— NAK: slave ně nemá kam odpovídat

I²C read M S

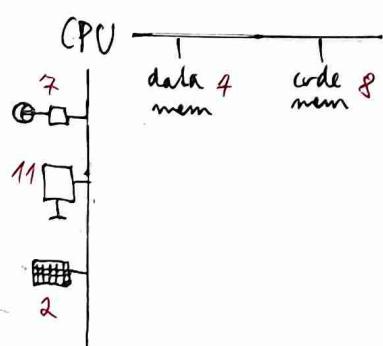
— NAK: master ně něčeho čist dál

I²C specific control
device specific

ACK: pokud je vše pořád správ. tak i poslední je ACK

* adres slava
předávání
overhead payload \rightarrow to někdejší v packetu - rářivý na rářírem'

• adresový prostor (address space)



→ rářírem' na abstrakčních mají adresy \leftarrow slavové mají

→ když má abstrakce n-bitový adresový prostor, tak rářírem' má místní adresy $0 - 2^n - 1$

* I²C: 7-bitový adresový prostor \Rightarrow 0 - 127 max 128 slavů

R/W bit 1: read

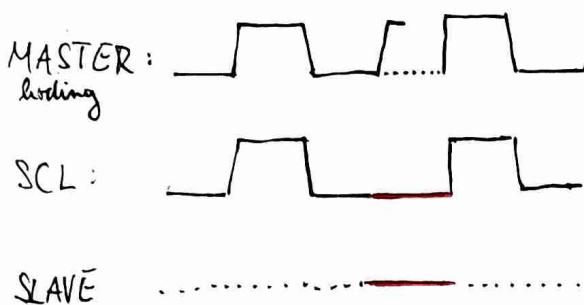
0: write

I²C nemají

masivní adresy

• Clock stretching

- na I²C sběrnici by měl hodinový signál mít frekvenci 100 kHz - 5 MHz
- některá levná zařízení umí číst jen hodně malé frekvence



Slave počkeší bit až, než ho může spracovat než přijde další bit.

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

→ master se snáší vysílat 1, ale vidí, že

na SCL je 0 ⇒ master se odpojí ⇒ slave spracuje 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 I²C zařízení

- měří intenzitu světla → připravovaný jas obrazek

- ALS má counter register

↳ když senzor detectuje foton(y),

tal se incrementuje

→ před začátkem měření se signáluje

— měření ... —

count = 0 stop integration

start integration ←

- command register - formátuje si poslední příkaz

↳ posléze write s nějakým kódem

- ALS komunikuje s I²C pomocí sběrnicového rozhraní (bus interface)

- má hardwarově danou slave address → je hardwired (zadržovaná)

→ na jedné sběrnici nemohou být dva

→ do (MD reg.) pouze zapisovat ⇒ write-only W/O

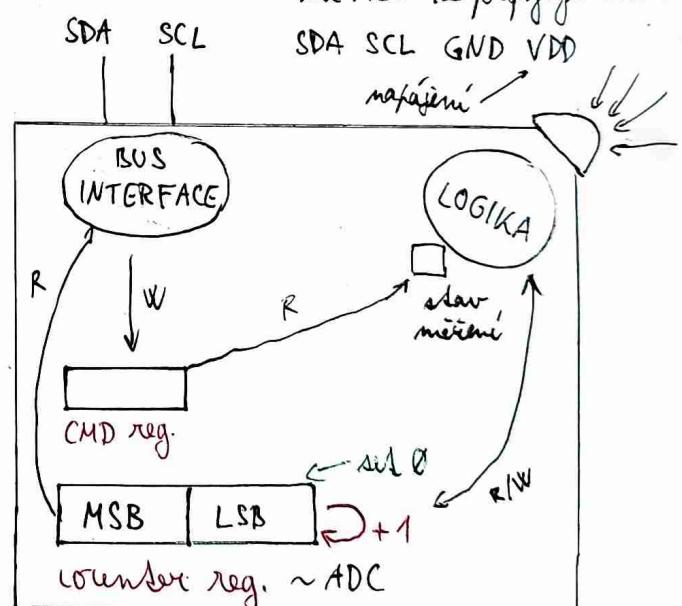
→ R count. reg pouze číst ⇒ read-only R/O

→ ALS ho nemá, ale jsou i read-write R/W

→ na ALS se připojuje 4 vodiče

SDA SCL GND VDD

nafází



} logika málo mělo dělat,
ale to jde mimo ráz

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

→ myšlenka:  ← 32 b. packet pro 4 byty, MSB-first

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

→ byte order { MSB-first } je něčta ho řeší, když posíláme více bytů
LSD-first

→ ALS posílá obsah counter reg. LSB-first, ale I²C má MSB-first, tedy
bity v obou bytech jsou MSB-first

• paměť počítáče

→ paměťový adresový prostor

256 B 

Adresa bytu: n-bit unsigned

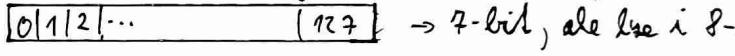
celkově $2^n B \Rightarrow$ 8-bitový adresový prostor

⇒ pro 256 B paměti potřebujeme
alespoň 8-bit adresový p.,

200 B 

ale funguje i libovolný
řád \Rightarrow 16-bit 0-65535

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

128 B 

→ 7-bit, ale byt i 8-bit ...

256 B 

... [65536]

⇒ pro 8-bit a.p. bychom museli používat min 8 → kdybychom vyměnili paměť za
řešení s 16-bit a.p. tak bychom museli psát program

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

• jednotky

65536

• výrobci paměťových jednotek

1KB = 1024 B \rightarrow 10-bit \rightarrow 1KiB \rightarrow 16-bit \rightarrow 64 kB

1KB = 1000 B

1MB = 1024 kB \rightarrow 20-bit \rightarrow 1MiB \rightarrow 24-bit \rightarrow 16 MB

1TB = 1000 MiB

1GB = 1024 MB \rightarrow 30-bit \rightarrow 1GiB \rightarrow 32-bit \rightarrow 4 GB

① ②

1TB | ① Jak mohou promítanou potřebujeme na aloaci adresy?
1PB | ② Jak velký adresový prostor lze pomocí ní nadresovat?

• registr řadice

1 registr \rightarrow 8 bit = 8. (1bit) \rightarrow implementace použí 1 latch (4-6 tranzistorů)

hamatuje si 1 nebo 0

• paměť SRAM S = Static

→ implementace stejnou technologií jako registry $256 B = 256 \cdot 8 \cdot 1 \text{bit}$

RAM = Random Access Memory

Random Access Memory

- 1) Může se využívat, ke kterému byly přistupují - dneska skoro všechny paměti
- 2, Uniform speed - přístup k libovolném bytu v libovolném řádku stejně stojí
 ↳ tohle pro paměti RAM většinou neplatí \Rightarrow nejsou random access

přístup	SRAM	DRAM	SRAM	DRAM
sekvencí \uparrow	✓	✓	• kapacita	13-23 MB
sekvencí \downarrow	✗	✗	• přenosová rychlosť	10-100 GB/s
random	✗	✗	• přístupová doba (access t.)	$\sim 1\text{ ms}$
				$\sim 10\text{ ms}$

\rightarrow charakteristika vlastnost RAM

- R/W \rightarrow dá se použít pro data mem. i code mem. - ale to jenom dneska tří paměti
- volatile \rightarrow když ji odpojíme od proudu, tak zanemene svůj obsah
 \Rightarrow code mem. musí být non-volatile \Rightarrow code mem. nemůže být RAM

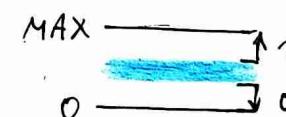
\rightarrow 1 bit \sim 4-6 tranzistorů \Rightarrow tří paměti nemohou být moc velké

\Rightarrow chceme, aby operační paměti (data mem.) byla velká \Rightarrow SRAM se nehodí

- DRAM - nemá random-access, je R/W a volatile

\rightarrow levnější než SRAM, 1 bit \sim 1 tranzistor + 1 kondenzátor, větší kapacita

\rightarrow kondenzátor \leftarrow naplněním elektronu \rightarrow 1
 neplnění \Rightarrow 0



\rightarrow Dynamic RAM - data se zanemenuje každých 1ms - i když to je zanemene'

\hookrightarrow protože kondenzátor 1 se vybije do kondenzátoru s 0

\Rightarrow neč se to stanoví, tak obnovíme původní náboj = refresh

\hookrightarrow když se děláme pravidelně, tak se nezanejme

\rightarrow to dělá buď CPU nebo nějaká speciální součástka

\rightarrow Problem: když se provádí refresh DRAM, tak nemá možné číst ani zapisovat

\Rightarrow DRAM je asi 10x pomalejší než SRAM

\rightarrow registry : SRAM

Operacní paměť: SRAM / DRAM

\rightarrow je volatile \Rightarrow když máme nejake proměnné heslo \rightarrow je významná paměť se samoz smaze ✓

I²C 256B SRAM

I²C

nařízení

Slave address

- má 8 vodících : SDA, SCL, VDD, VSS(GND), A0, A1, A2, TEST

- má programovatelnou adresu - má A0, A1, A2 můžeme připojit VDD nebo GND
⇒ adresa : 1010 A2 A1 A0

- kom. protokol:

Slave a.	0
----------	---

^w Word a.] Data] → co chci rafat na tu adresu
↳ adresa slova *

→ slovo (word) = jednotka přenosu / spracování

→ definováno pro kardálé zářízení

→ 8-bit slovo ⇒ zářízení pošle 1B v kardálé transakci

→ n-bitové slovo ⇒ má ho n-bitové zářízení

⇒ n-bitová paměť má n-bitové slovo ve n-bitovém adresovém prostoru !

→ procesory mají velkou bitost ⇒ jeho registry mají velikost toho slova,
operace provádějí s čísly o délce toho slova ...

→ 16-bitová paměť myslí ne v bytech, ale ve slovech - ta jsou číslorana'
ale CPU (program) pracuje s adresami bytek

8-bit mem v pořadí CPU

0	1	2	3	4	5	6	7	8	9	10	11	12	...
---	---	---	---	---	---	---	---	---	---	----	----	----	-----

16-bit mem.

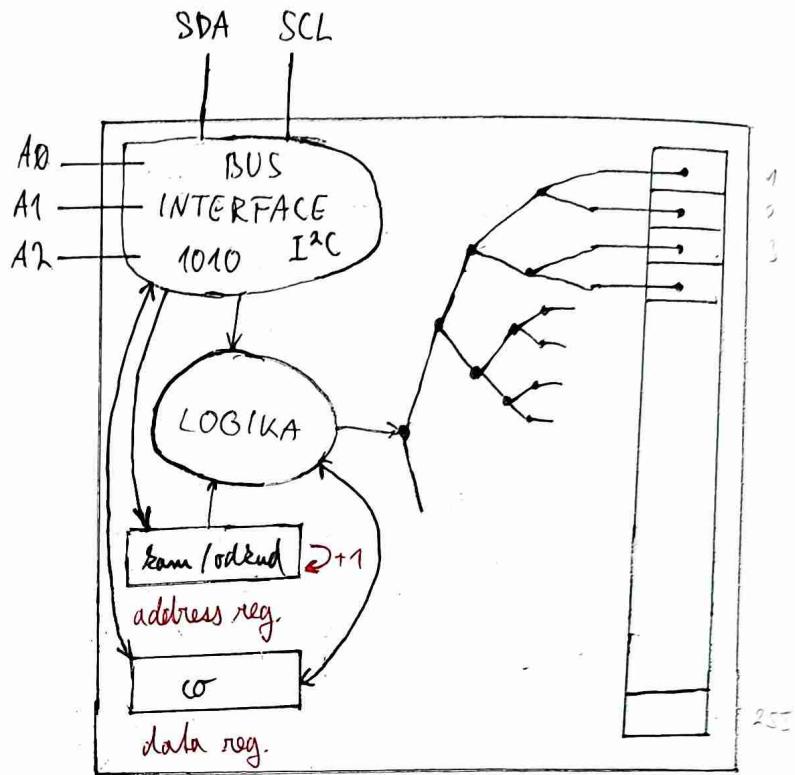
0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

32-bit mem.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300	301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325	326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375	376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400	401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425	426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450	451	452	453	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475	476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500	501	502	503	504	505	506	507	508	509	510	511	512	513	514	515	516	517	518	519	520	521	522	523	524	525	526	527	528	529	530	531	532	533	534	535	536	537	538	539	540	541	542	543	544	545	546	547	548	549	550	551	552	553	554	555	556	557	558	559	560	561	562	563	564	565	566	567	568	569	570	571	572	573	574	575	576	577	578	579	580	581	582	583	584	585	586	587	588	589	590	591	592	593	594	595	596	597	598	599	600	601	602	603	604	605	606	607	608	609	610	611	612	613	614	615	616	617	618	619	620	621	622	623	624	625	626	627	628	629	630	631	632	633	634	635	636	637	638	639	640	641	642	643	644	645	646	647	648	649	650	651	652	653	654	655	656	657	658	659	660	661	662	663	664	665	666	667	668	669	670	671	672	673	674	675	676	677	678	679	680	681	682	683	684	685	686	687	688	689	690	691	692	693	694	695	696	697	698	699	700	701	702	703	704	705	706	707	708	709	710	711	712	713	714	715	716	717	718	719	720	721	722	723	724	725	726	727	728	729	730	731	732	733	734	735	736	737	738	739	740	741	742	743	744	745	746	747	748	749	750	751	752	753	754	755	756	757	758	759	760	761	762	763	764	765	766	767	768	769	770	771	772	773	774	775	776	777	778	779	780	781	782	783	784	785	786	787	788	789	790	791	792	793	794	795	796	797	798	799	800	801	802	803	804	805	806	807	808	809	810	811	812	813	814	815	816	817	818	819	820	821	822	823	824	825	826	827	828	829	830	831	832	833	834	835	836	837	838	839	840	841	842	843	844	845	846	847	848	849	850	851	852	853	854	855	856	857	858	859	860	861	862	863	864	865	866	867	868	869	870	871	872	873	874	875	876	877	878	879	880	881	882	883	884	885	886	887	888	889	890	891	892	893	894	895	896	897	898	899	900	901	902	903	904	905	906	907	908	909	910	911	912	913	914	915	916	917	918	919	920	921	922	923	924	925	926	927	928	929	930	931	932	933	934	935	936	937	938	939	940	941	942	943	944	945	946	947	948	949	950	951	952	953	954	955	956	957	958	959	960	961	962	963	964	965	966	967	968	969	970	971	972	973	974	975	976	977	978	979	980	981	982	983	984	985	986	987	988	989	990	991	992	993	994	995	996	997	998	999	1000
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	------

- konkrétní 256B SRAM má 8-bit slovo, takže adresa slova = adresa byte *
- dílčí většina familií 16-bit → slovo je často myšleno 16 bitů
- doubleword (DWORD / DW) = dvojslovo - dvojnásobek slova - často 32 bitů
- quadword (QWORD / QW) = čtyřislovo - čtyřnásobek slova - často 64 bitů

- overhead komunikačního protokolu
 - na 1 datový byt je potřebovat 3·9 bitů $\Rightarrow \frac{19}{27} \approx 70\%$ overhead
 - pro 16-bitové slovo: na 2B 4·9 bitů $\Rightarrow \frac{36-16}{36} \approx 55\%$ overhead
- v paměti mají adresy uvedené, ale logiku se pouze "výhodou" může správná cesta ke konkrétní adrese - O a 1 indikuje kam zahrát
- má address register, kde je uložena adresa slova co chceme
 - ↳ jeho velikost = velikost a.p. slov.
- v data registeru je slovo, které chceme přečíst nebo zapsat
 - ↳ jeho velikost = velikost slova
- burst přenos: když máme rápidně něco přečíst z paměti, tak se hodnota v adresovém reg. incrementuje \Rightarrow když rápidně 1 čtu více slov sekvencí, tak ho mohu udělat v 1 transakci \Rightarrow malý overhead
- pro mezonečný přenos: 2·9 bitů + 1bit pro každý datový byt $\Rightarrow \frac{18+1}{18+9} = \frac{1}{9} = 11\%$



→ write x read transakce

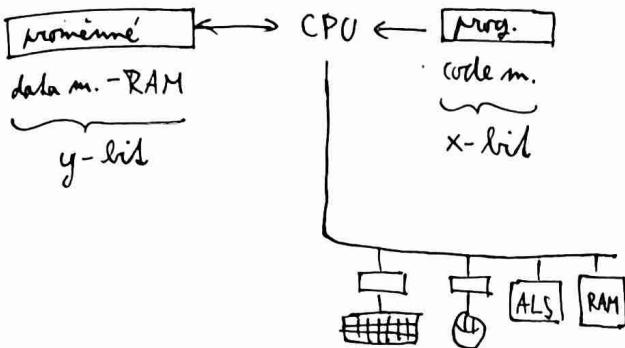
- write = 1 write
- read = 1 write (Slave address + word address)
- read (Slave address) \rightarrow myní slave posílá sekvenci slova od čtemi je formulejší než zápis - to je omezení I²C ne RAM
 - \rightarrow DRAM má rychlejší čtemi než zápis
- Registrový adresový prostor
 - když má nějaké rozdílné více 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. \leftarrow write
 \rightarrow Slave a. Data \leftarrow posílá slave

↳ adresy mají v podobě $0, 1, \dots, n$, ale mají různou techniku přijetí

Harvardská architektura



→ procesor podporuje nějaké konkr. prototypy
↳ y-bit a x-bit adresovými prostory
⇒ musíme vybrat data m. a code m., které ten procesor podporuje

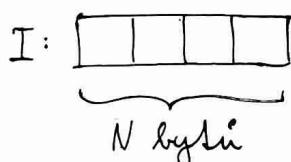
→ instrukce procesoru = písáry, které ten procesor umí vykonávat

↳ instrukční sada (Instruction set) je souběžná řada instrukcí

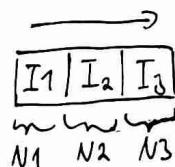
↳ různé procesory mají různé instrukční sady

→ instrukce jsou uložené v code mem. jako posloupnosti bytů

→ některé procesory mají různé instrukce stejně dlouhé (homogené) jiné je možné heterogenní



⇒ posloupnost instrukcí

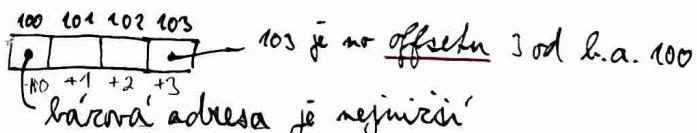


procesor je vykonává směrem k rostoucím adresám

→ procesor v sobě má registry, které reprezentují jeho stav

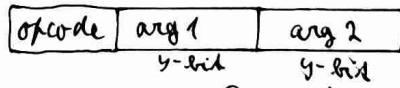
• Program Counter (PC) → je v něm uložena adresa instrukce,
= Instruction Pointer (IP) která se právě vykonává

→ když je instrukce vícebytná tak odkazuje na táckou adresu



→ až aktuální instrukce sloni, tak IP incrementuje o její velikost (+N1)

→ instrukce se skládá ze 2 částí



1) opcode - identifikátor té instrukce

$$105 := \oplus$$

*

2) argumenty - co si tam - mohou být implicitní (incrementace o 1)

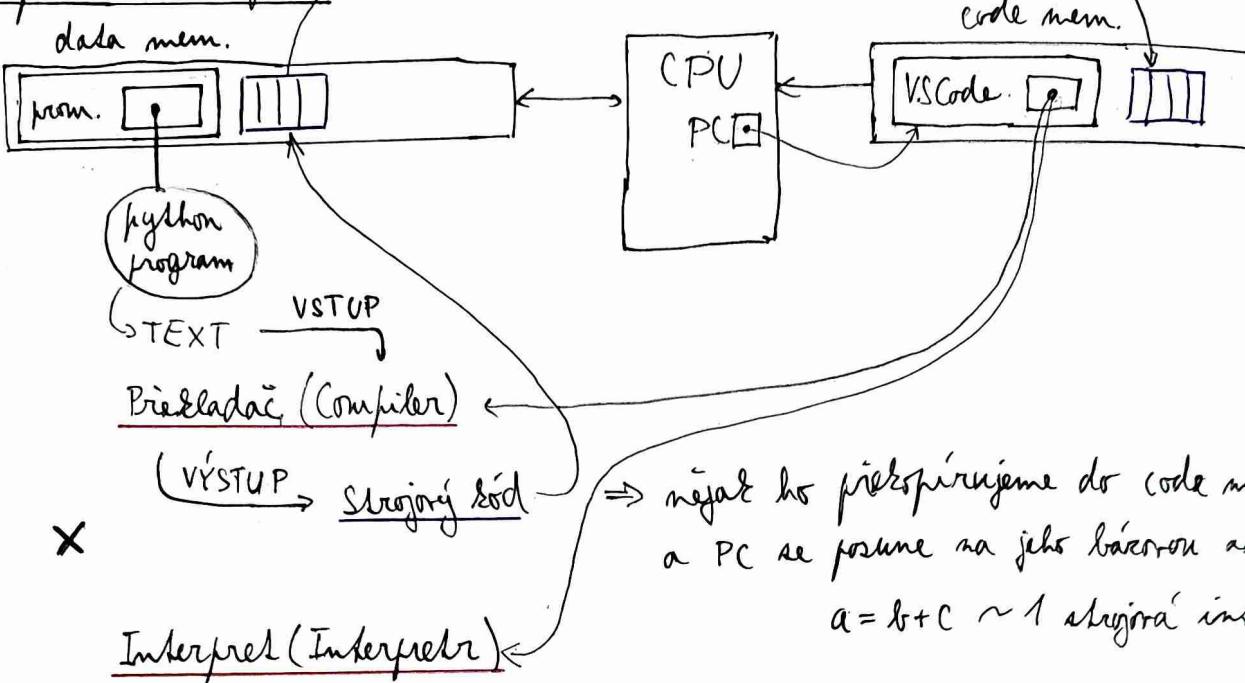
→ velikost program counter je x bitů pro code mem. a x-bitovým adresovým p.

→ argumenty jsou proměnné, reprezentované jejich adresami

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

→ strojový kód = program psaný pomocí instrukcí procesoru

• Compiler x Interpreter



⇒ nějak ho přespiřujeme do code mem.
a PC se posune na jeho barvou adresu
 $a = b + c \sim 1$ strojová instrukce

```
if znak == "+": } spousta instrukcí naráží a ten nás program se
    x = a           můžete nejdřív do strojového kódu
    y = b           z = x + y add
    if ...
    if ...
```

⇒ interpret je mnohem fornalejsí než prekompilátor,
ale je mnohem snazší ho napsat
⇒ python je interpretován, C, C++, Java, ... komplikovaně

PYTHON

proměnná = ID jméinem

STROJOVÝ KÓD

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

⇒ prekompilátor musí mít freekled, kde je volné místo v
fameli, aby se proměnné nacházely na volných adresách

• Endianista (Endianness)

→ chceme uložit nějakou nicabystovou proměnnou někam do fameli

	48	56	34	12
	12	34	56	48

\$FF \$100 \$101 \$102 \$103

MSB LSB
\$12 34 56 48

bárová
adresa
proměnné

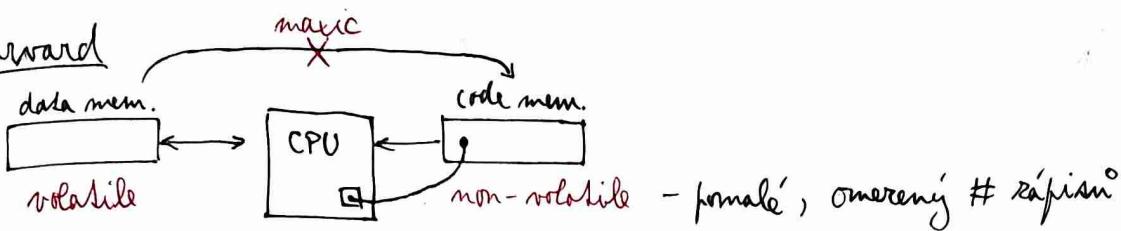
uint 32

- Little Endian (LE) - LSB ukládám na nejménší adresu
- Big Endian (BE) - LSB ukládám na největší adresu
- pravidlo LLL: LE je pro LSB na Lowest adrese

→ endianista je dán procesorem - většina dnešních procesorů je LE

→ problém: Z BE počítáce uložím data na flashen → Edýtor ji zapojí do LE počítáce, tak by si do paměti mohlo uložit BE data - musí se to řešit

• Harvard

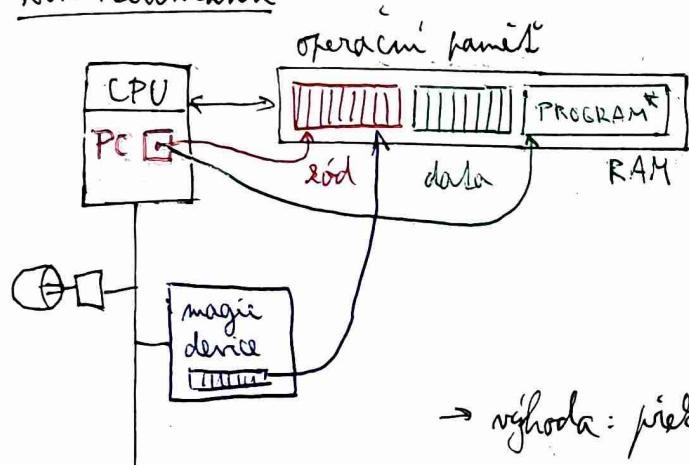


→ mohli bychom všechno interpretovat ⇒ formale

→ překladač pořebuje rozdílný kód z data m. do code mem.

⇒ Asynchronní procesory → Harvardskou architekturou totéž nemají magic

• von Neumann



- PC odkazuje jen do té části paměti, kde je kód
 - Edýtor je argumentem nějaké instrukce proměnné, takže se uloží do té části paměti, kde jsou data
- hardware je to složitější

→ výhoda: překladač vygeneruje data reprezentující stejný kód toho programu a registru PC na myž zájmu uložit
⇒ ten program se zájme provádět ani bychom něco kopírovali
→ nebezpečí: hacker může do dat uložit šodlivé instrukce

→ operací paměť je RAM (rychlá) ⇒ volatile ⇒ může se nám ten počítací program

⇒ magical device, které při zapnutí počítací ještě předtím, než procesor začne pracovat napíší do operací paměti ten počítací program z mezipaměti non-volatile paměti

→ historicky: Apple II - VisiCalc (Excel) ⇒ piersalna aplikace

→ procesor začíná umí pracovat s jediným x-bitovým adresováním prostorem

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

- 6502: 8-bit procesor, 16-bit adresový prostor → adresuje 64 kB
- Intel 8088: 16-bit procesor, 20-bit adresový prostor → 1 MB
($\times 86\text{-}16$)

↳ IBM PC - na tu dobu hodně dobré

↳ měl také jen 64 kB paměti, aby kvůli 20-bit a.f. kompatibilní

⇒ by 8-bit 16-bit family - pro výkonejší procesor bylo třeba psát programy ⇒ do IBM PC stačilo dát menší paměť

- Intel x86/IA 32: 32-bit procesory, 32-bit adresový p. → 4 GB

↳ Trik: 2 instrukční sady: stávající x86-16 + novou ⇒ back-compatibility

- Intel 64/x64: 64-bit procesory, 64-bit adresový p. $\sim \infty$

↳ 3 instrukční sady ⇒ back-compatibility

Základní instrukce

↙ 32-bit a.f.

6502 (LE)

0

\$EA

Intel x86 (LE)

0

\$90

0 1 2

\$4C xx₀ xx₁

PC := \$xx₁ xx₀

0 1 2 3 4

\$E9 xx₀ xx₁ xx₂ xx₃

PC := \$xx₃ xx₂ xx₁ xx₀

← offset od base adresy | nic neděláj

← machine code | PC := PC + 1

instrukce skoku (jump)

↳ PC sbětí na adresu X | unconditional jump

→ 3 podmínený skok ⇒ pro if-va sbětí na nej

→ programovat v machine kódu by bylo zhruba složitější

Assembler

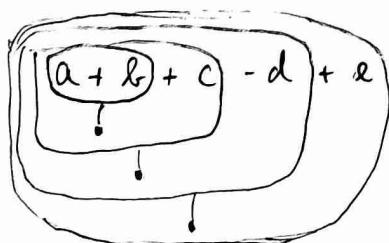
→ pro kódový procesor jazyk

- programovací jazyk, který je textovým zápisem toho stejněho kódu

6502: nic nedělej: NOP

skok : JMP \$xx₁ xx₀

} pak se musí přeložit do machine kódu



MIPS: a := b op c

x86, 6502: a := a op b

↳ umí pouze primitivní, primární operace, ...

Assembler

$$a = b + c$$

A hand-drawn diagram consisting of three blue lines. All three lines originate from a single point located at the bottom left of the frame. Each line extends upwards and to the right, with slight variations in slope and curvature.

adresy \Rightarrow 32-bit a. p. by 16 instrukcji byla mrc allraha'

→ casto: pri operaci může být jen 1 proměnná, rbusk je uložený
6502 x86 jeho maximálně v některých obecných registrech toho procesoru
↳ PC / IP je speciální reg.

LOAD addr(b) → R1

→ Load-Store arch.

ADD R1 + addr(c) → R2

→ všechny argumenty musí být reg.

STORE R2 \rightarrow addr(a)

→ nafri. MIPS

\rightarrow x-bitový procesor má x-bitové všechny reg.

→ kontakta /immediate hochwota = argument, který nemá adresu pro německého čtenáře

1, LDA # \$XX, ← load constant \$XX₁₆ to register A 8-bit processor

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

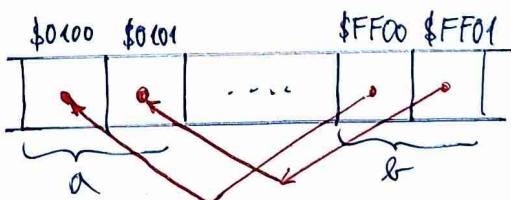
↳ na úrovni strojového kódu lze jsou dvě odlišné instrukce

→ Copy-on-write registry (Transfer): TAB → B := A
EDR0J ← ↳ CIC

mid8: $a = b$ $\text{addr}(a) = \emptyset \times 0100$ $\text{addr}(b) = \emptyset \times FFOO$

LDA \$FF00

STA \$0100



visit 16:

LDA \$FF00
STA \$0100

LDA \$FF01
STA \$0101

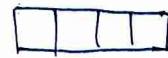
$$a = b$$

→ příklad 6502 LDA #\$XX₀ ≡ \$A9 XX₀
 LDA \$XX₁XX₀ ≡ \$AD XX₀XX₁
 STA \$XX₁XX₀ ≡ \$8D XX₀XX₁

32-bit program. one ma \$A409
 32-bit program. two ma \$A410

→ zapiš dané příkazy do strojového kódu a Assembleru. Strojové kódy mohou být:
 a) \$1400 b) \$1500 c) \$15FC

a) two = one



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

LDA \$A404	AD 04 A4	1400: AD 04 A4 8D 10 A4 AD 05 A4 8D 11 A4 AD 06 A4 8D
STA \$A410	8D 10 A4	1410: 12 A4 AD 07 A4 8D 13 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	

b) one = 1277 = \$000004FD

LE	LDA #\$FD	0 1 2 3 4 5 6 7 8 9 A B C D E F
	STA \$A404	1500: A9 FD 8D 04 A4 A9 04 8D 05 A4 A9 00 8D 06 A4 8D
	LDA #\$09	1510: 07 A4
	STA \$A405	
	LDA #\$00	
	STA \$A406	
	STA \$A407	

c) two = -2 = \$FFFF FFFE

LDA #\$FE	0 1 2 3 4 5 6 7 8 9 A B C D E F
STA \$A410	15FC: A9 FE 8D 10 A4 A9 FF 8D 11 A4 8D 12 A4 8D 13 A4
LDA #\$FF	
STA \$A411	
STA \$A412	JMP \$XX ₁ XX ₀ ≡ \$4C XX ₀ XX ₁
STA \$A413	NOP ≡ \$EA

→ příklad: Buňkové program máte. Zapiš koncovou hodnotu všech bytů, které program zahrnuje.

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

A = 03

• (\$A90C)¹ = 03.

• (\$A901)¹ = 03

A = (2009)¹ = AB

• (\$A902)¹ = AB

→ příslad: Před během programu jsou na adresách 8000-800F maly. Napiš hexdump pro program.

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

8000: 12 12 00 00 00 00 00 00 00 00 00 00 00 00 00 00

$$A = FF \rightarrow A = 00 \rightarrow (8000)^1 = 00 \rightarrow A = 12 \rightarrow (2011)^1 = 12$$

$$A = EA \rightarrow (2014)^1 = EA \rightarrow (2010)^1 = EA \rightarrow (201C)^1 = EA$$

$$A = 12 \rightarrow (8000)^1 = 12 \rightarrow A = 12 \rightarrow (2011)^1 = 12 \rightarrow (8001)^1 = 12$$

• Příznakový registr procesoru (flags register)

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

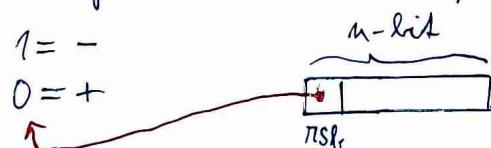
→ ty flagy spolu resourví - u mysi: L R M ← flagy

• 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



pro znaménková čísla dává smysl
pro bezznaménková čísla to je právě MSb

• carry - pomocný příznak, když nějaká instrukce potřebuje bit navíc

→ nějaký výsledek se nepojde do obecného registru, tak se ten bit navíc uloží sem

→ některé instrukce mají side efekty → nastavování příznaku

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

→ typicky to dělají všechny aritmetické instrukce

→ člení flagy: procesory standardně nemají instrukci na člení flagu

→ conditional jump/branch

if flag: JMP ← jaro podmínka if-n dáme ten flag

else: NOP

→ nastavování příznaku

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

×86: CLC }
STC } carry
CLZ }
STZ } zero

↳ 6502 umí nastavovat jen carry flag

• obecná reg. architektura (x86)

→ všechny obecné r. jsou ekvivalentní
⇒ více instrukcí, složitější výroba

• akumulačná arch. (6502) reg A

→ 1 obecný reg. je akumulátor
→ většina aritmetických operací
umí pracovat jenom s akumulátorem
→ některé procesory mají více akumulátorů

→ akumulačná arch. 6502

→ AND, OR (ORA), XOR / EOR, NOT

↳ 6502 nemá NOT, NOT := EOR #\$FF

$A := A \text{ op imm/addr}$

↳ umí jen při-ovrávat, ...

→ SHL, SHR, ROL, ROR

↳ 6502 umí shiftovat / rolovat jen o 1: $A := A \text{ op } 1$

→ Python: $a = a \mid b$

$a, b = \text{uint8}$ ↑ ↑

adresy → \$A000 \$B000

LDA \$A000

ORA \$B000

STA \$A000

→ $a, b = \text{uint16}$: je jedno r. jde o pořadí 16 r-registers

\$A000:

LSB	MSB
a ₀	a ₁
LSB	MSB

 a = a₁ | a₀
\$B000:

LSB	MSB
b ₀	b ₁
LSB	MSB

 LE

LDA \$A000

ORA \$B000

STA \$A000

LDA \$A001

ORA \$D001

STA \$A001

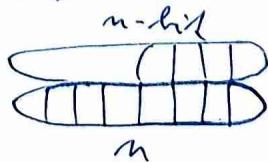
→ sčítání

01011	input A
01110	input B
01110	carry
11001	result

⇒ stále máme ten carry flag

⇒ vždy pod sebou sčítáme m-bitová čísla

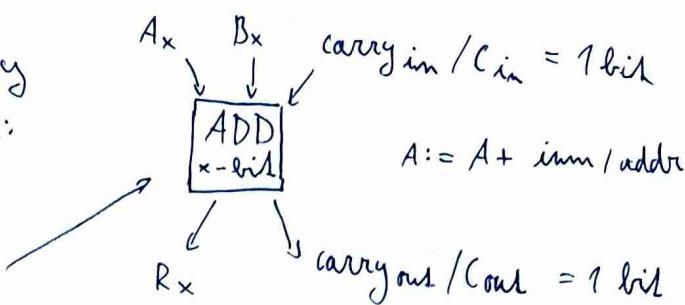
↳ tedyby 8bit ADD 16bit ⇒ sign/zero extenzion



- 6502 pracuje s 8-bit slovy

\Rightarrow sčítání 16-bit proměnných:

sčítání s přenosem
(add with carry) ADC



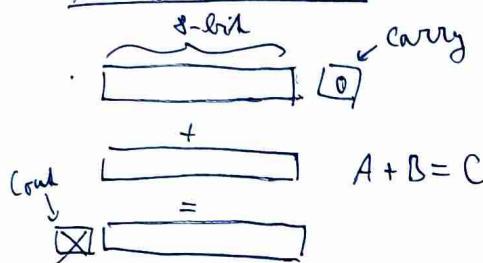
$$A := A + imm / \text{addr}$$

Cout	Result
1 bit	8-bit

$\Rightarrow 9 \text{ bit}$

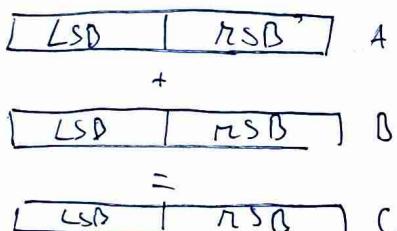
\Rightarrow u posledního ADC uděláme Truncation tím, že zapomeneme Cout

\rightarrow sčítání 8-bit čísel

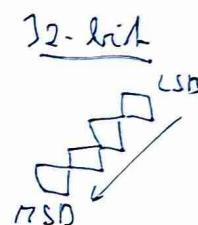
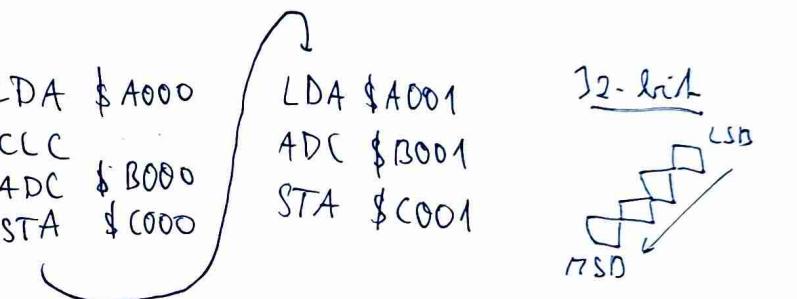


LDA \$A000
CLC
ADC \$B000
STA \$C000

\rightarrow sčítání 16-bit čísel



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



\rightarrow sčítání znaménkových čísel - se dvojicemi doplnků

\rightarrow stačí se na to znaménková čísla konkat jaro na bezznaménkové, sčítání je a pot se na výsledek konkat jaro na znaménkové čísla

\rightarrow increment / decrement

6502 nemá pos reg. A

\rightarrow INX $x := x + 1$
DEX $x := x - 1$

side effects:

P. NEGATIVE = X.7
If $X = 0$: P. Zero := 1
else: P. Zero := 0

\rightarrow odčítání unsigned

uděláme $A := \text{value} - A \rightarrow$ pot zmíne i $A := A - \text{value}$

\rightarrow vyvážíme dvojicový doplnek

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

$$\left. \begin{array}{l} \text{temp1} = A \\ A = \text{value} \end{array} \right\} A = \text{temp1} - A = A - \text{value}$$

\rightarrow NOT A \rightarrow INC A \rightarrow ADD value

6502:
EOR #\$FF \leftarrow NOT
CLC
ADC #\$01 \leftarrow INC
CLC
ADC value

$\rightarrow A, \text{value}$ jsou 8-bit unsigned

$$\Rightarrow (-A) \quad \boxed{1} \quad \boxed{7F}$$

\rightarrow pořebujeme, aby to byla 7bit čísla
tak: $\boxed{01} \quad \boxed{7F}$

$\Rightarrow \ominus$ jde udělat pomocí \oplus , ale normální procesory mají instrukci odčítání

→ Subtract with borrow 6502 ji' nema' × 86 ans

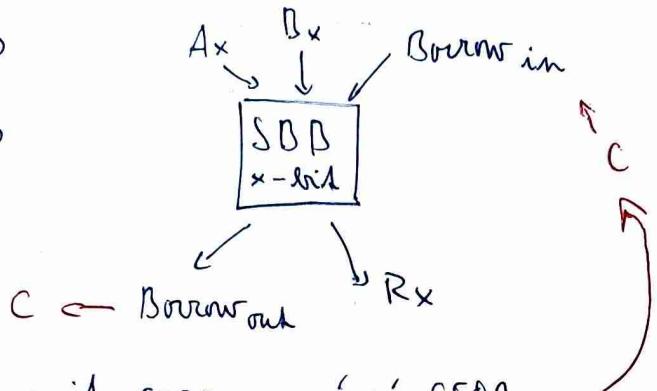
SBD a, b

$$a := a - (b + \text{Borrow})$$

carry := borrow from last bit

$$C = B$$

$$\begin{matrix} 1 & 1 \\ 0 & 0 \end{matrix}$$



$$\begin{array}{r} 11101 \\ - 0111 \\ \hline 0110 \end{array} \quad \begin{array}{l} 13 \\ - 7 \\ \hline 6 \end{array}$$

x-biting SBD

CLC } 8-bit odčítání
SBD }

→ Subtract with carry má' ji' 6502

SBC

$$C = \text{NOT}(B)$$

$$\begin{matrix} 1 & 0 \\ 0 & 1 \end{matrix}$$

$$\text{result} := A - \text{imm/addr} - \text{NOT}(P.\text{carry})$$

8-bit přesná

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

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

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

$$= A + \text{NOT}(X) + C$$

$$\underline{SBC X := ADC \text{ NOT}(X)}$$

SEC } $C=1 \Rightarrow D=0$
SBC } 8-bit odčítání

hardware je snadné k implementaci

→ 8-bit odčítání

$$A := \text{value} - A \equiv \begin{array}{l} \text{Temp} 1 := A \\ A := \text{value} \end{array}$$

$$\left. \begin{array}{l} \text{SEC } (C=1) \\ \text{SBC } \text{Temp} 1 \end{array} \right\} =$$

$$\begin{array}{l} \text{SEC} \\ \text{NOT } \text{Temp} 1 \\ \text{ADC } \text{Temp} 1 \end{array} \equiv \begin{array}{l} \text{NOT Temp} 1 \\ \text{INC Temp} 1 \\ \text{ADD Temp} 1 \end{array}$$

*

ADD + $\underbrace{1 \times \text{carry}}_{\text{INC}}$

* Postup je mimořádně strany funguje i pro scítání 8b. čísel

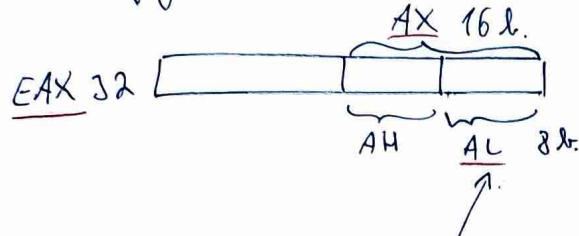
Obecná registrární arch. ×86

instrukční pointer : EIP 32 bit

příznačný register : EFlags 32 bit

7 obecných registrů 32 bit

→ co když bychom chtili sčítat 16-bit proměnné? ⇒ další 16-bit instrukce



můžeme ještě ovlivnit na
1B. AL / AH

↳ 16-bit pokud do
spodní části toho reg.

⇒ chová se tak jako 16-bit reg.,
ale chtili jsme 16-bit 32 bit

↳ probíhá se truncation
16 b, 17 b, ... při npr. sčítání

→ arch ×64 → 64 bit reg. + 32 bit operace ⇒ všechny 64 bit reg + dletem na 12, ...

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

MOV r, imm → LOAD

MOV imm, r → STORE

MOV r₂, r₁ → transfer

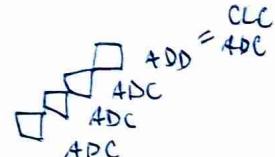
OP target, source → target := target OP source

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

→ operace: ADD, ADC, SUB, SBD, IMUL, IDIV

Operational
mohou být
mezi registry

OR, AND, XOR, NOT, SHR, SAR



→ operace mohou sideeffects, MOV nemá

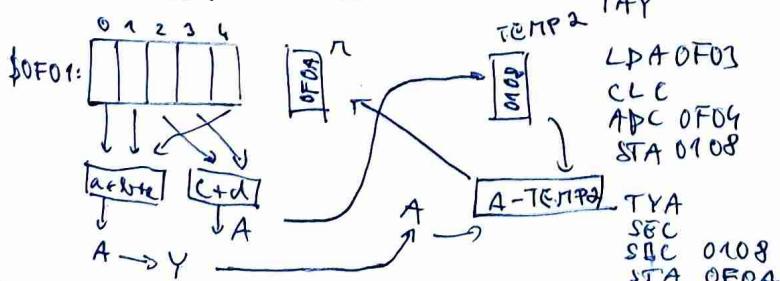
→ příklad: r = a + b + c - (c + d)

6502, 8-bit proměnné

$$\text{TEMP1} = a + b + c$$

$$\text{TEMP2} = c + d$$

$$r = \text{TEMP1} - \text{TEMP2}$$



LDA 0F01

CLC

ADC 0F02

CLC

ADC 0F03

TAY

LDA 0F03

CLC

ADC 0F04

STA 0108

TYA

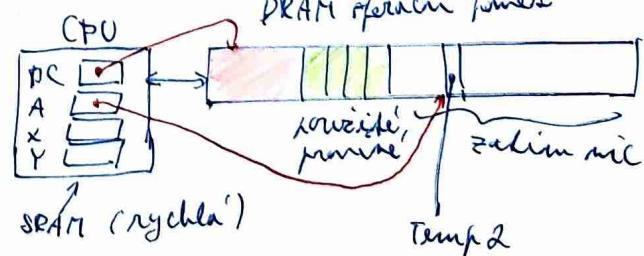
SBC

SBC

STA

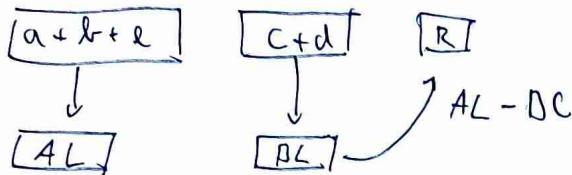
0108

0F04



↳ druhou r je, když si chceme něco uladit do registru

$$\rightarrow x86 \quad r = a + b + c - (d + e)$$



MOV AL, [00000F01 h]

ADD AL, [00000F02 h]

ADD AL, [00000F05 h]

MOV BL, [00000F03 h]

ADD BL, [00000F04 h]

SUB AL, DL

MOV [00000F04 h], AL

\rightarrow otočené hodiny byly 32-bit → na x86 se pouze změní adresy + opisy inst.

+ je to skoře rychlejší

na 6502 by to bylo mnohem komplikovanější

24-bit - x86 [8bit 16bit]

6502 [8bit 8bit 8bit]

\rightarrow finished

$$\begin{array}{r} 11101010 \\ 11011111 \\ \hline 11111110 \text{ Carry} \\ 111001001 \end{array}$$

$$\begin{array}{r} 11101011 \\ 101 \\ \hline 00001111 C \\ 11110000 \end{array}$$

- one word 32 0xA400
- two word 32 0xA404
- three word 16 0xA408

0xA400 0 1 2 3 4 5 6 7 8 9

LG

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 + \$09

{ 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

$$\cdot \text{two} = \text{one} + \text{two}$$

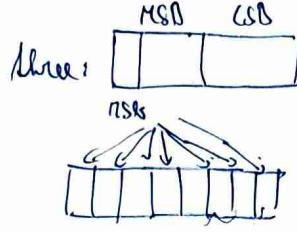
→ stejné

$$\cdot \text{one} = \text{two} + 500$$

→ stejné

$$\cdot \text{two} = \text{one} + \text{three} \rightarrow \text{potřebujeme sign ext. three}$$

↳ například MSB.



→ horní 2 byly 00 nebo FF

→ formální řešení: \$D500 - \$D600

$$\text{SHL} = \text{ASL} \quad \text{SHR} = \text{LSR}, \text{shift } \pm 1$$

$$\text{LDA } \$A400 \quad \leftarrow \text{MSB}$$

$$\text{AND } \#\$80 \quad \leftarrow [RSB/0000..]$$

$$\text{STA } \$A400$$

SHR →

$$\begin{array}{l} \text{LSR} \\ \text{ORA } \$D500 \\ \text{STA } \$D500 \end{array}$$

7-dílk.

← 00 v FF

CLC

$$\text{LDA } \$A400$$

$$\text{ADC } \$A408$$

$$\text{STA } \$A409$$

$$\text{LDA } \$A401$$

$$\text{ADC } \$A409$$

$$\text{STA } \$A403$$

$$\text{LDA } \$A402$$

$$\text{ADC } \$D500$$

$$\text{STA } \$A406$$

$$\text{LDA } \$A403$$

$$\text{ADC } \$D500$$

$$\text{STA } \$A407$$

→ příklad

$$\begin{array}{r} 11101010 \\ - 11011111 \\ \hline 11111 \text{ borrow} \\ 00001011 \end{array}$$

$$\begin{array}{r} 11101011 \\ - 00000101 \\ \hline 0100 \text{ b} \dots 111110000 \text{ b} \\ 11100110 \dots 111110001 \end{array}$$

$$00001101 - 13$$

$$- 00011100 - 28$$

8-bit truncation

→ unsigned form. na stejných adresách

$$\cdot \text{two} = \text{one} - \text{two}$$

$$\cdot \text{one} = \text{two} - 500$$

$$\cdot \text{two} = \text{one} - \text{three}$$

→ stejné, pouze menší převod rozdílu

SEC

{ LDA \$A400

 SBC \$A404

 STA \$A404

SEC

{ LDA \$A404

 SBC #\\$FD

 STA \$A400

:

* x86 one uint32 0000A400h

two uint64 0000A404h

three uint64 0000A40Ch

$$\cdot \text{three} = \text{two} + \text{three}$$

MOV EAX, [0000A404h]

ADD EAX, [0000A40Ch]

MOV [0000A40Ch], EAX

MOV EAX, [0000A408h]

ADC EAX, [0000A410h]

MOV [0000A410h], EAX

$$\cdot \text{two} = \text{one} - \text{three} \rightarrow \text{zero ext.}$$

MOV EAX, [0000A400h]

SUB EAX, [0000A40Ch]

MOV [0000A40Ch], EAX

MOV EAX, #\\$00000000

SBB EAX, [0000A410h]

MOV [0000A410h], EAX

→ příklad 6502

A	word8	\$C12A
B	word8	\$C12B
C	word8	\$C12C
D	word16	\$C200
E	word16	\$C202

rovná: \$D500 - \$D600

$$\rightarrow 15 = \$000F$$

$$• A = A + B + C$$

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

$$• E = E + D + 15 + (B - A)$$

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 ×86

$$• A = A + B + C$$

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 několik jednotek, které si mezi sebou předávají data
 - ↳ taktovací f. různým jednotkám, když mají přijímat / posílat data $\sim \text{CLK}$
 - ⇒ ten hodinový signál časuje vydávání jednotlivých kroků procesoru
- 1 tick = 1 cycle → když jednotek je hodně + doba jež pracoval paralelně
 - ↳ moderní procesory dokáží za 1 tick vykonat více instrukcí
- rychlosť instrukcí

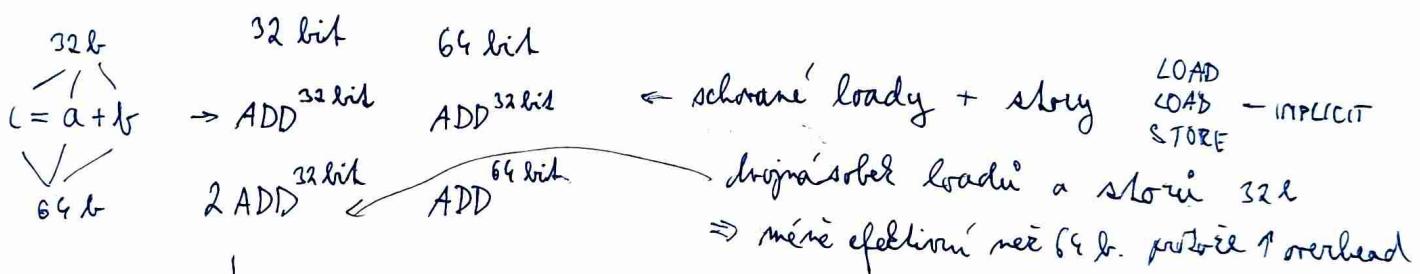
fast • 1 tick - bináře operace (AND, ... SHL, ...), ADC, SBB/SBC

slow • LOAD, STORE - pracují s operační pamětí DRAM - formát

! ADC $r_1 \quad r_2$

ADC $r_1 [addr_1] \leftarrow$ implicitní load

řetíz slavě lepší



načítá + uloží se 16 B do paměti \rightarrow 32 bit méně fiktivní než 64 bit.

ale obvykle je formát \Rightarrow 32 bit nebude ohrazený krokem veče 64

\rightarrow 8 bit 6502 \rightarrow 64 bit. proměnné 8ADD^{exit}

6502 80386^{x86} dnesni'

1. f. ~1 MHz ~33 MHz ~ GHz

→ rychlosť instrukce ovlivňuje kroky

• clock rate

• počet kroků, které instrukce trvá

↳ rozsahuje do DRAM? \Rightarrow formát

rychlosť

1. SHL EAX, 1 2. MOV EAX, [addr]

ADD ECX, EDX

MOV [addr], EAX

kroků
rychlejší
TRANSFER \leftarrow MOV EDX, EAX

ADD ECX, 01234567h

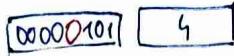
↳ dlouhá instrukce \Rightarrow kroků formátování

hodinový
signál

• Python

int 5 = 0101

int 255 = 01111111

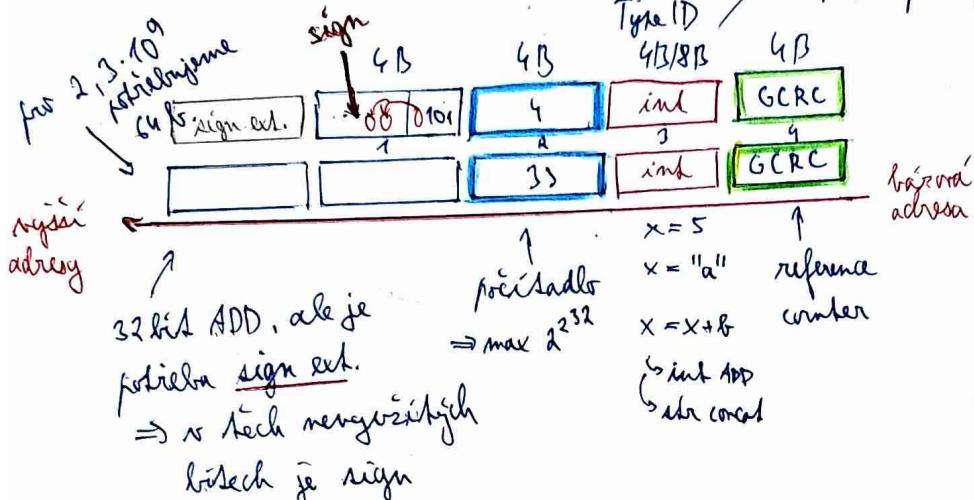


→ funkce placující listy ~ funkce adder

→ operace ADD → museli bychom hrát s bity → hrozí neefektivita

→ Python 1.5 zatímto nedělá → tych schůzivcům nelze dát 8b. ale 32b.

32b./64b. platforma (procesor)



* Python je rádové
formulejší své C/C#

20B overhead

→ Python x = 5

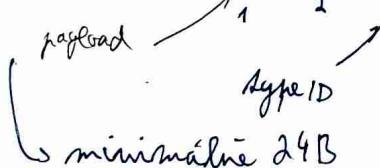
adresa → 32b.f. → 4B

4B.f. → 8B

⇒ referel (pointer) = forměná, ne která je adresa k paměti
reference (Python) nemůžeme registrovat k adresám

⇒ tedy můžeme $x = \text{vahy}$ → změní se ta referenční adresa

⇒ zabírá $2B + 4B + 4B + 4B + 4B = 24B$ → za původní 5 pořád zabírá formět



minimálně 24B

⇒ Garbage collection - nové objekty, ne které
nic neobsazuje a oznáčí je za prázdnou paměť

• C/C#

uint 32 a = 5

4B

→ formy typu

→ formy velikostí

↳ objektu sice má
rozdílnou formět

⇒ LOAD LOAD STORE

⇒ Reference counting

$$\begin{array}{ll} x = 5 & 5^{+1} \\ y = x & 5^{+1} \\ x = "a" & "a^{+1} 5^{-1} \\ y = 4 & 5^{-1} \end{array}$$

ediny dosáhnuti mohu, když to
je odpaže ⇒ ta pamět se může
novou použít

⇒ cykly a grafy ref? → jehož za čas prochází graf
referencí a hledá izolované komponenty

⇒ v Pythonu můžeme načíst adresu - určí že je hr na objekt, rovněž
platných cífer → normalizace ⇒ pak se to sečte ⇒ generace nového výsledku

⇒ v Pythonu jsou int, str, ... immutable - nová hodnota → nový objekt ⇒ placující
nový objekt ⇒ mají identickou = hashy + GC ⇒ store TypeID, ... ⇒ renormalizace

⇒ finální store ⇒ 1 ADD ⇒ Python → desítky instrukcí LOAD, STORE, ... *

Násobení + Dělení //

→ hardware využívá různé metody

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

✗ → možné je dělat SW

→ jde je to rychlé a tažké

	HW	SW
mul	1-10	ještě komplikující
div	10-100	

↑ racionálně implicitně lze dělat a složit

Násobení dělení množinami 2

unsigned

$$\begin{array}{l} \text{Shift left} \\ \text{Shift left} \end{array} \quad \begin{array}{l} \cancel{x \text{ SHL } m = (x * 2^m) \bmod 2^n} \\ \cancel{x \text{ SHR } m = x // 2^m} \end{array} \Rightarrow \underline{\text{SHL} = \text{Logical shift}}$$

signed

• násobení - pokud nevyužíváme sign延展 bit $\cancel{x \text{ SHL } m = x * 2^m}$ pro dosahování mala ✗

• dělení - pro sítadla čísla to bude fungovat

pro - posíláme doleva přidáváme 1 $\Rightarrow \underline{\text{SAR} = \text{Arithmetic shift}}$

\Rightarrow slovo to funguje

nečítá sign ext. ✓

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

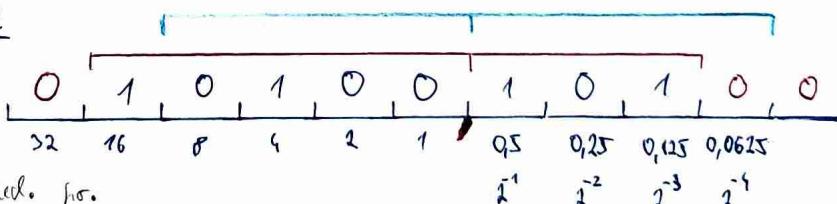
$$-6 \text{ SAR } 1 = -3$$

→ rozšiřuje se do nahoru \Rightarrow $\begin{cases} - \Rightarrow +1 \\ + \Rightarrow +0 \end{cases}$

$$\cancel{(x + \text{MSR}(x)) \text{ SAR } m = x // 2^m} \quad \leftarrow$$

Reprezentace reálných čísel

0020,62500 ^{trailing zeros}



trailing zeros

fixed. fr.

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

\rightarrow sčítání - posíláme stejnou fixed-point repr.

$$1.25 + 1.875 = 3.125$$

\Rightarrow můžeme to sčítat jako bezznaménková algoritma i odcítat

$$\begin{array}{r} 00001.010 \\ + 00001.111 \\ \hline 11.001 \end{array}$$

\Rightarrow sčítání & fixed-point čísla jsou rychle

$$5.3: 1.250 \cdot 2^3 + 1.875 \cdot 2^3 = C \cdot 2^3$$

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

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

Záporná čísla ve fixed-point

↳ chceme $1.25 \approx 5.3 \Rightarrow 1.25 \cdot 2^3 = a = 10 \Rightarrow 1.25 = 10 >> 3$

$$10: 01010.000 \Rightarrow 1.25 : 00001.010$$

↳ $-1.875 \approx 5.3 \Rightarrow -1.875 \cdot 2^3 = -15$

$$-15: 10001.000 \Rightarrow -1.875 : \underbrace{11110.001}_{\text{či je? *}}$$

$$\begin{aligned} * 11110.001 &\Rightarrow \text{NOT}(11110001) + 1 = 00001111 \\ &\quad 00001.111 = 1.875 \end{aligned} \quad \left. \begin{array}{l} 11110.001 = -1.875 \end{array} \right\}$$

Vztažení čísel ve fixed-point

→ normálne podľa endienističky

→ na \$0010F300 až \$0010F31F sú same malomé čísla

⇒ následne

- 18.375 ako 8.24 na \$0010F300 → 12.60 00 00
- 18.375 ako 12.20 na \$0010F308 → 01 2.6 00 00
- 1040.5 ako 8.8 na \$0010F30C → 0E 10.80 ← remain fixed
- 1.5 ako 4.4 na \$0010F310 → 1.8
- 65535 ako 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:	<u>00</u>	<u>00</u>	<u>60</u>	<u>12</u>				<u>00</u>	<u>00</u>	<u>26</u>	<u>01</u>	<u>20</u>	<u>10</u>			
\$0010F310:	<u>18</u>				<u>00</u>	<u>00</u>	<u>FF</u>	<u>FF</u>	<u>00</u>							

Floating point repr.

$$20,625 = 10100.101$$

ředěcká notace

$$A = -1010\ 0101\ 0000\ 0000.0$$

B =

$$0.0000\ 0000\ 1010\ 0101 = \underbrace{1,0100101}_{\text{mantisa}} \times \underbrace{2^{-9}}_{\text{exponent}}$$

normalizovaný zápis

- zádne leading zeros
- 1 cifra pred :
- 0 nemá reprezentaci!

$$\rightarrow \text{exp 5b} \rightarrow 0 \dots 31$$

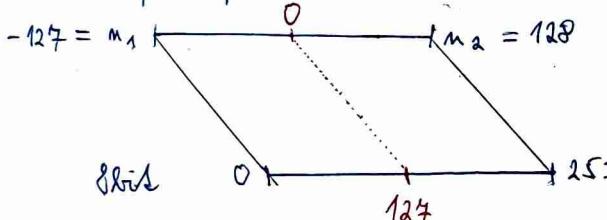
$$\Rightarrow \text{exp bias} + 15 \Rightarrow -15 \dots 16$$

$$15 \leftrightarrow 30$$

$$-9 \leftrightarrow 6$$

bias repr. celých č.

↳ repr. s posunem



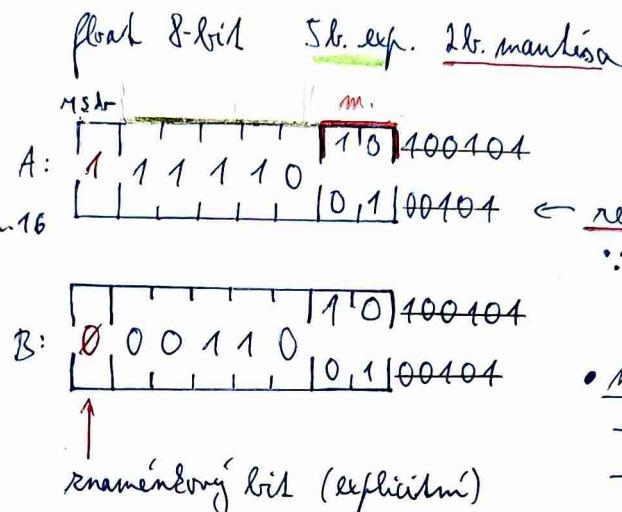
$$\Rightarrow \text{posun} + 127 = \text{bias} + 127$$

$$-127 + 127 = 0$$

$$-126 + 127 = 1$$

$$0 + 127 = 127$$

$$128 + 127 = 255$$



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

repr. se sčítají 1
↳ všechna čísla jsou $1 \dots 2^e$

musíme řešit

→ float 8b

→ 5b exp. s bias + 15

→ 2b mantisa se sčítají 1

→ nejdřív m, exp, sign

float aritmetika

→ 10-100 radek

→ musíme ji softwarově implementovat

→ mnohem pomalejší než fixed-point

→ měkké procesory mají hardwarenou podporu

→ redleady obecných registrů pro celočíselnou aritmetiku mají další sadu pro práci

→ float čísl. → další instrukční sada

FADD, FSUB, FLOAD, FSTORE, ...

Standard IEEE 754

single 32b. → 8b. exp + 23b. mantisa

-127 - 128 resp. -126 - 127 - 0,255 mají

double 64b. → 11b. exp + 52b. mantisa

→ 0,255 mají jedenam

-1023 - 1024

→ sčítá 1, bias upravuje

	Python	C/C#
single	float	float
double	float	double

x86/x64 ARM µC 6502
HW HW/SW SW SW

HW footp. pro single + double

\rightarrow síťové floating-point čísel 23 b. mantisa \leftarrow single

$$A + B = A$$

regional se lo sé 23%

denormalizace menšího čísla \rightarrow může se r koho stát nula!

→ single: 23b. na mantise ⇒ Edyří rozdíl exponencií je něčí něč 23, když se k základu

$$\text{decimal} \Rightarrow \begin{array}{r} 10\ 000 \\ + 0.001 \\ \hline \end{array} \sim \left. \begin{array}{l} 2^{13} \\ 2^{-10} \end{array} \right\} \text{je 10 na branici}$$

$$2^{2^3} = 8 \cdot 10^6$$

$\Rightarrow 10^9 + 16 = 10^9$

\downarrow \downarrow

2^{30} 2^4

abstrakte
informelle

! 0.1 má reálné dvojité součátky
 nerovný desetinný rozvoj $b = a$ \times
 $-E < (b-a) < E$ ✓

→ representation only

→ minimalist exponent

$$\text{minimální exponent} \quad \underbrace{0,00\ldots 001101}_{127} = 1.101 * 2^{-127} = 0.\overline{1101} * 2^{-126}$$

$$\begin{array}{r} m \\ \overbrace{\quad\quad\quad} \\ -127 \boxed{110100\dots} \\ = 0.m * \frac{1}{2}^{126} \end{array}$$

→ IEEE: edgy min.. exponent \Rightarrow mantissa denormalizierung

$$0.00\dots 00\underbrace{0}_{127}1101 = 0.1101 * 2^{-127}$$

$$= \underbrace{0.01101}_{m} * 2^{-126}$$

$000\dots 00$
"

$$\begin{array}{r} -127 \\ \hline 01101000 \end{array} = 0, M * 2^{-126}$$

$0 := [0/1] - 127 \quad 00000000 \rightarrow$ minimální element a nula má max. hodnotu

↳ realizirjene -0 a +0 pričemž $-0 = +0$

(s minimum) exp. from no bias repr. same' naly \rightarrow 0 := same' naly

→ max. eff.

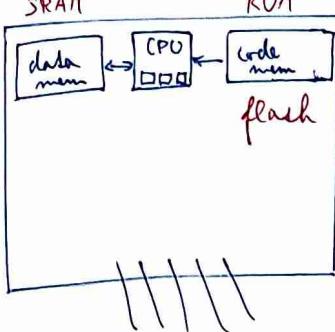
- multorai mantissa  froble sign $\infty + \infty = \infty$, $5/0 = \infty$
 - nenulorai mantissa \rightarrow Not a Number NaN ∞/∞

↳ přideme r nejvýší exponent, ale získáme NaN

→ nice drunk NaN \Rightarrow wegle Astrolat $A == NaN$

⇒ müssen sie gewusst haben, dass man sie nicht mit der Wahrheit bestimmt.

• Ježdříví / µC / MCU



→ Harvardská architektura

- data mem. SRAM 256B - 1KB - 10KB
- code mem. non-volatile - ROM

→ firmware - ten µC voládá pravidla ⇒ firmware je ten program co jí řídí - staci mi ten program mít když jsem jednou

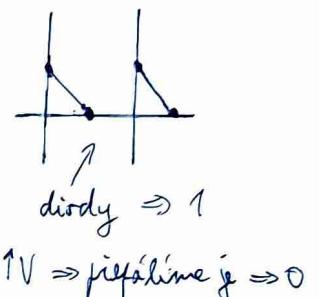
- ROM (Read Only Memory): ∞ read 1 write ← návážné



- PROM (Programmable ROM): ∞ read 1 write ← uznatelský write

- EPROM (Erasable PROM): ∞ read ∞ write ← nejbezpečnejší ROM

⇒ char. vlastnost paměti ROM: **non-volatile**



→ překlad: IEEE, například paměti mezi \$0010F300 a \$0010F32F, původně 00, 1E

$$\cdot \$0010F300 : \begin{array}{c} 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{array} \text{ 8bit} \quad \boxed{1100\ 00\ 11} \quad 11.1 = 1.11 * 2^1 \rightarrow 16$$

$$\cdot \$0010F302: x=118, 33984375 = 1110\ 110.01010111 = 1.11011001010111 * 2^6 \rightarrow 21$$

16 b. 10m, 5e $\boxed{0\ 10101\ 1101100101}$ $E = \frac{6+12+7}{133}$ $6+1023 = 1024 + 5$

$$\cdot \$0010F304: -x 32b. 23m, 8e \boxed{1\ 1000010\ 1101100\ 10101110\ 0000\ 0000}$$

$$\cdot \$0010F310: -x 64b. 52m, 11e \boxed{1\ 1000000\ 0101\ 1101\ 1001\ 0101\ 1100\ 0000}$$

$8 + \$0$

$$\$0010F300: \begin{array}{cccccccccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & A & B & C & D & E & F \\ \underline{C} & \underline{1} & \underline{65} & \underline{59} & \underline{00} & \underline{4E} & \underline{EC} & \underline{C2} & \underline{00} & \underline{00} & \underline{80} & \underline{FF} & \underline{71} & \underline{00} & \underline{00} & \underline{80} \end{array}$$

$$\$0010F310: \begin{array}{cccccccccccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \underline{00} & \underline{00} & \underline{00} & \underline{00} & \underline{C0} & \underline{95} & \underline{5D} & \underline{C0} & \underline{00} \end{array}$$

$$\$0010F320: \begin{array}{cccccccccccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \underline{00} & \underline{00} \end{array}$$

$$0 = 128 \rightarrow 255$$

$$\cdot \$0010F308: +∞ 32bit \quad \boxed{0\ 1111111\ 000\ 0000\ 4*\$0}$$

$$\cdot \$0010F30C: 0,00000678 = 0.0000000000000000011100011011$$

+15 16b. 10m 5e $= 0.0001110001 * 2^{-14}$ $-15 + 15 = 0$

$$\boxed{0\ 00000\ 0001110001} \quad 0071$$

$$\cdot \$0010F30E: -0,0 16b. \boxed{1\ 0---0} \quad 8000$$

$$\cdot \$0010F320: -3.25 80b. 64b. mantisa bez signifikace 1) 15b. exp bias + 16 383$$

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

$$\boxed{1\ 1000000000000000\ 1101\ 0000}$$

C 0 0 0 0 P 0 7D 0

• EPROM - erase pomocí UV ~ 20 min. na slunci

• E²PROM (Electrically EPROM) - ∞ R, "∞" W \times flash

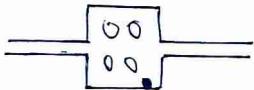
	SRAM	DRAM	<u>E²PROM</u> x flash	HDD	(CD/DVD)
rychlosť	> 10 GB/s	1-10 GB/s	100 MB/s - 1 GB/s - 5 GB/s	100 MB/s	10 MB/s
písanie	< 1 ms	~10 ms	~100 ms	~ms	~100 ms
kapacita	8B-MB	MB-GB	MB-GB — 1TB	TB-20TB	
serialný člení	✓	✓	✓	✓	
random 1 byte	✗	✗	✗	✗	
random zápis 1 byte			✗	✗	
random 1 blok			✓	✓	

• EEPROM

→ bytov adresatelná

→ serielyn počet rápisu do 1 byte

~ 100 000 - 1000 000 W



Celstrying se dříví než nějaké

romise 1 := hodně E^-

0 := řidné E^-

→ nějaké E^- tam zůstanou natrvalo

→ natronec se ten bytov zapisuje

• NVRAM - non-volatile RAM → read-write

→ debilní název

→ využívá EEPROM a flash - nejbezpečnější non-volatile paměti

→ po dlouhé době (měsíce roky) se ten elektron vymazuje prý

⇒ natronec to vtrati svou informaci

⇒ špatně pro archivaci

• flash

→ rozdělena do bloků 1 blok = 1 kB - 16 kB

→ v 1 transakci se musí přenést celý blok

→ jde by měla obří slovo

★ → protože velká písmenná rychlosť

→ velmi rychlé serialní člení

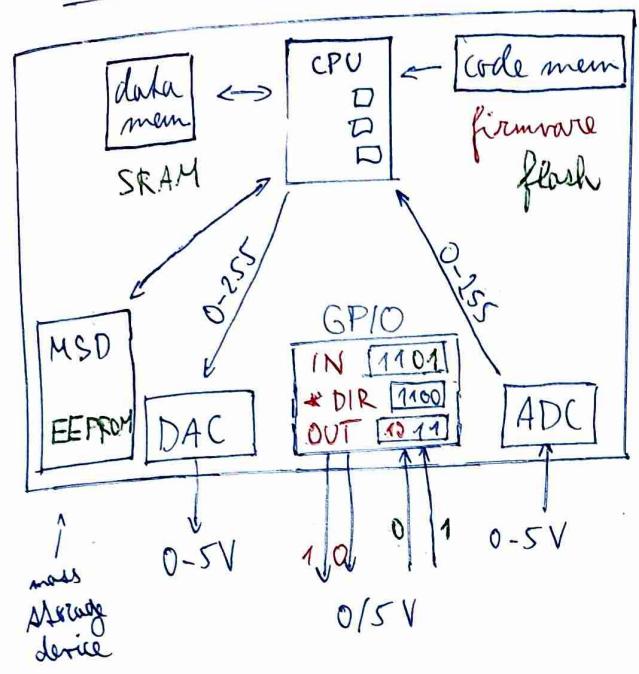
→ velmi pomalé random 1 byte

→ firmware se často nahrazuje na flash

→ 10 000 - 100 000 W dr 1 blok

→ když se rozbije 1 byte, pak je špatný celý blok

• microcontroller



- ADC - Analog Digital Converter

↳ dostane napětí a vrací nájemce všechno hodnoty
→ na praci se stáváme anglickem → V → užel
např. → ALS: interistikní skála → hodnota

- DAC - Digital Analog Converter

→ hradná ťa → na feši'

$b_{m-bit} \Rightarrow m-bit$ log' DAC (ADC)

- GP10 - General Purpose Input / Output

↳ digitální vstup a výstup

→ out: výkázaní segmentovaného displeje

→ in: *Slacidae* napi. rafmata / roymata

GP10: Direction register: říká, jestli je konkrétní vodič in/out

↳ m-für GPIO má n podícn

Output register: ready bit reprezentuje jidlova su linku

\rightarrow funkce je reakcí na vstupní signál, tak $0 \rightarrow 0V$, $1 \rightarrow 5V$ — logické 0/1

Input register: noslouží akéhokoli druhu, co vidi na input linkách

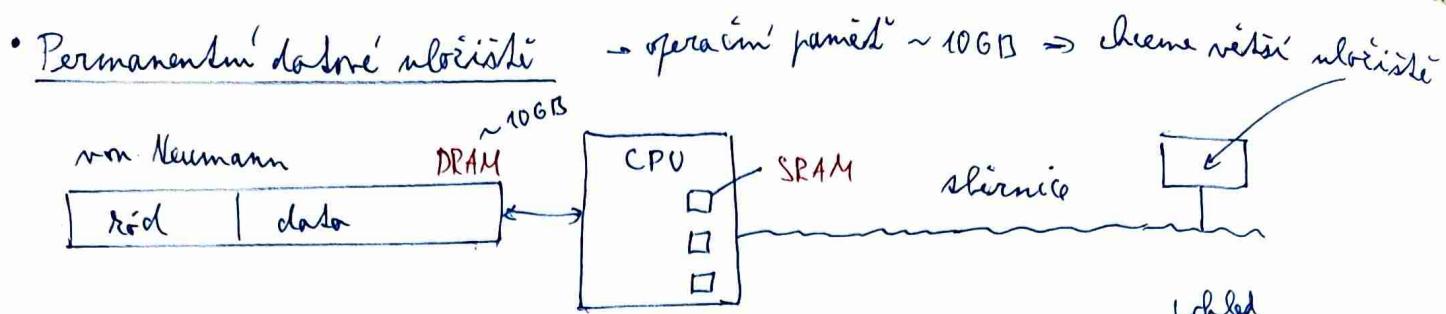
- Permanentni 'dolore' ulociste (mass storage device)

SRAM volatile

6. číslo ne kam nložit načárem' vizitáku, aby se rozmnožilo pro výskytu všechny

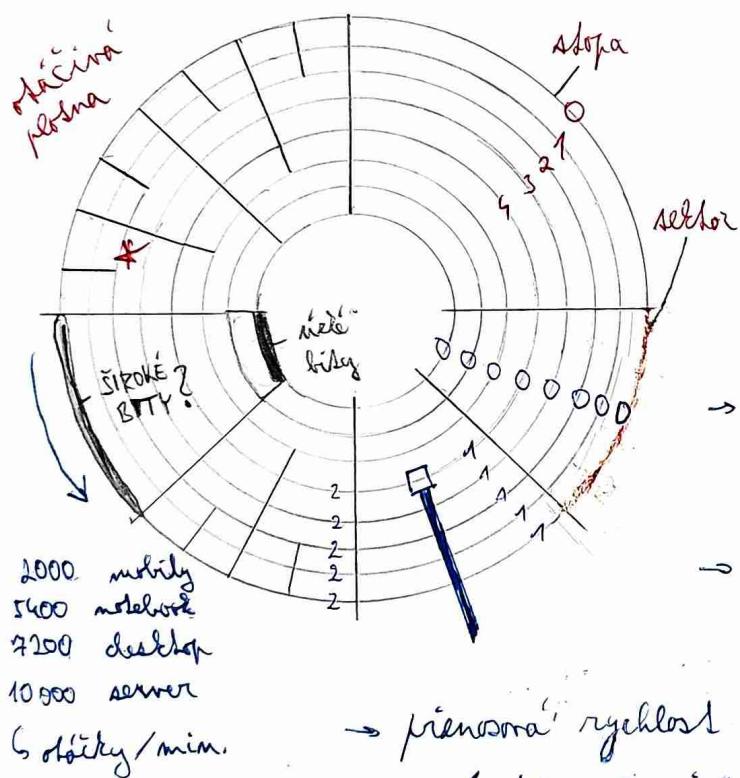
→ učením si ba pri káde' zmene

\Rightarrow EEPROM je cesta, protože má hodně write + je adresovatelná pro bytych



• Pevný disk / Hard disk drive (HDD)

- non-rotating + úložiště magneticky
- 1 bit = skupina difólií, které mají stejnou orientaci
↳ skupina, aby se m. pole bylo dost silné
- mezi nimi musí být mezera, aby se ty difóly nestřílely magneticky
↳ mezera - stejný materiál jako difóly, ale ty difóly tam nejsou orientovány
↳ pravděpodobně m. pole k té mezery je malové
- koncentrické sloopy



→ bity jsou na koncích = složek trasy
↳ číslové označení dovnitř

→ sloopy jsou rozdělené do výsečí = sekční sektory
1 sektor ~ 512 B - dnes
↳ advanced format 4 kB = 4096 B - dnes

→ bity jsou čtecí hlavou
↳ čím rychleji se disk stáčí, tím rychlejší čtení
→ když chceme číst k jiné sloopě, musí se hlava pohnout

↳ písací hlava

→ hledání sloopy = met

⇒ seek-time - formule

= přeskočit doba ~ 10 ms

→ přenosová rychlosť ~ 100 MB/s

• 1 sloopa sekvencně ✓

• 1 sloopa sekvencně ✗

• 1 sloopa random ✗

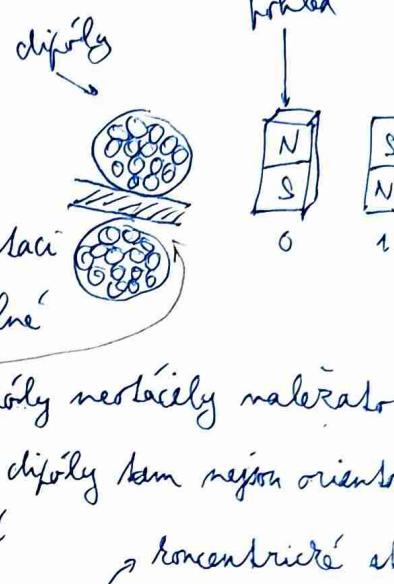
• náhodný přístup ✗

• výhody

→ kapacita 1 TB - 20 TB → hodně dobrý formát cena / výkon

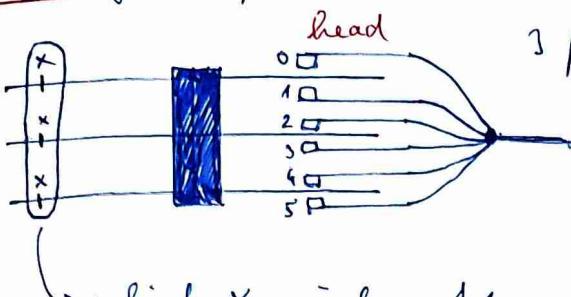
→ při zápisu se ten materiál nijak nemění ⇒ read as write

→ dobré pro archivaci - rázem užívání bez změny 10-100 let



→ sektory se rozdělují na podsektory - když by byly souběžné sirké

→ platen je více pod sebou



→ plateny → 6 porohů → 6 hlav

↳ data uložíme nahoru i dolu

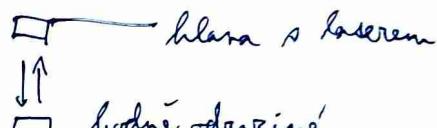
→ všechny hlavy se pohybují majednou

→ cylinder X = všechny sloopy se stejným číslem X

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

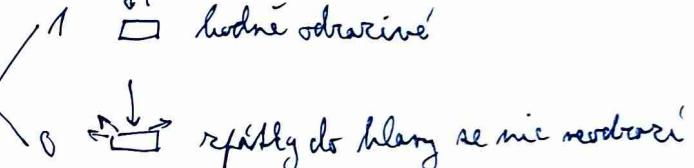
→ disk se ročí w ⇒ priesová rychlosť menších slopp } rafinované
je mnohem menší, než tich menších } ravenie

→ HDD majú rychlosť - když platenka myzadlím



• CD / DVD / Blue Ray - rychlosť

→ optický rozdíl - 1 bit = kus materiálu



↳ zápis: laserem se změní struktura koho

materiál a následně doba si to pamatuje → po 1-10 letech se to mění opět

→ jede 10 m/s lisované → fyzicky žáruje je diva

→ maximální hlaška - laser + musí poznat odražené ⇒ dloně se roztřípne ⇒ přístupevná

→ pomalu se to skáče ⇒ priesová rychlosť ~10 MB - 100 MB/s

doba
~100 ms

→ je to dělené na sekvencí priesky

→ sloopy mají kružnice ⇒ 1 spirální slope - rozdělená do sektorů

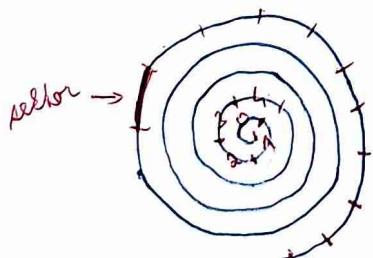
↳ stejně velké na délku i šířku

$$1 \text{ sektor} = 280 = 2048 \text{ B}$$

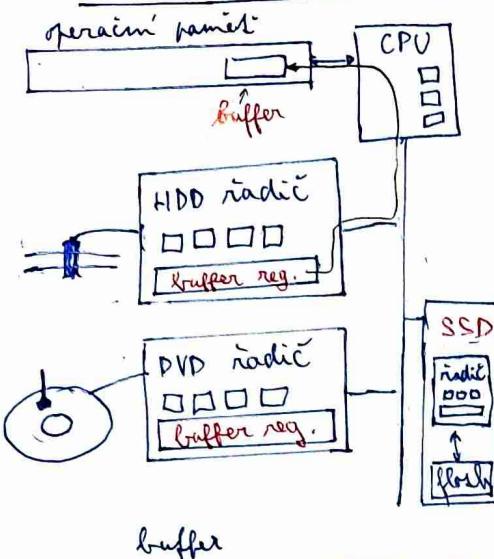
→ LBA (Linear Block Addressing) adresa sektoru

↳ sloučí nám 1 číslo → indexujeme konkrétně

→ může být malá CD disk, protože člene rozděluje



→ CD/DVD + HDD



• DVD mechanika ← v ní DVD rádič

- address reg. → LBA funkcii má i následnou
- cmd. reg → primary read, write
- status reg → skončilo ně počin / čtení? bit ready
- buffer → rádič umí přečíst / zapsat celý sektor
 - ↳ je tam uložený celý sektor
 - nemá možné k nimu přistupovat
- data reg. → data v něm se kopírují do bufferu
- info reg. → kapacita toho média, co je k rádiči připojené

$\boxed{0} \rightarrow +1 \cdot \text{addr. reg.}$ - říká kam do bufferu se má zapsat obsah data reg.

⇒ až načerpejme celý buffer, tak dojde k aritmatickemu přesunu → řetez 0

Write: pomocí data reg. napřímo data do bufferu
poté zapsat LBA adresu a poslat write příkaz

• HDD rádič / HDC (Hard Disk Controller)

→ ten komunikační protokol máte nyní už plně stejný, kromě adresy C/H/S

⇒ my rádiči poslouží LBA adresu a on ji přenese na C/H/S

↳ sektory jsou nejdříve lineárně uspořádány ⇒ nemusíme řešit, jestli komunikujeme s HDD nebo DVD

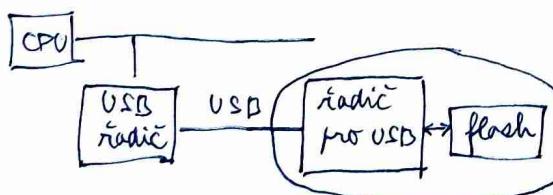
• SSD (Solid State Drive) = flash + rádič

→ flash paměť komunikuje v blozech

→ když ji přes rádič přijme k operaci → komunikace stejným protokolem jako HDD, DVD

→ nevýhody: načítací se do opětovně - sektory písmem fungují, menší kapacita, využívají

• flashka



flashka - formátovaná než SSD - USB je formátován

↳ má jen 1 cíp flash

SSD - libovolný druh má flash čipy, které pracují

současně - ale celí ještě přijíma 1 rádič

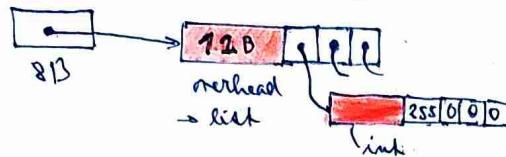
→ read: do operací paměti se mi přenese celý sektor & buffer

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

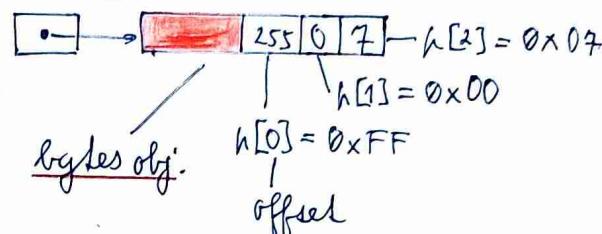
→ my ten sektor uložíme do paměti na baázovou adresu x

⇒ offset od začátku sektoru = offset od baázové adresy

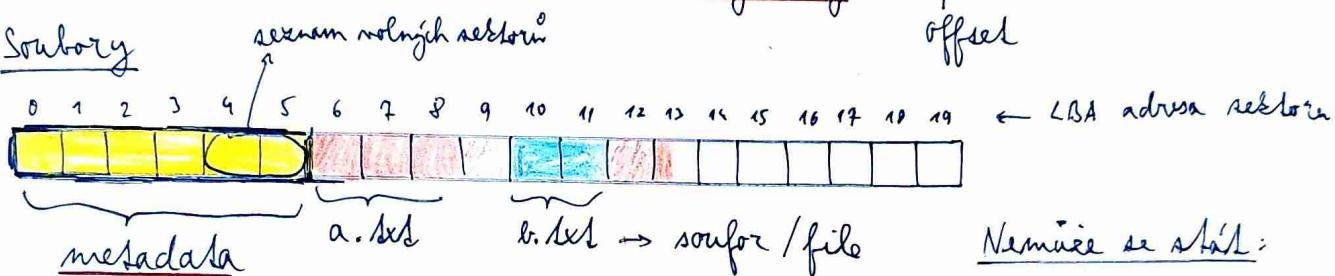
Python: `l: list = [255, 0, 7]`



`p = bytes([255, 0, 7])`



→ Soubory



Nemůže se stát:

- file - musíme si formátovat data -> sektorech

| metadata = data popisující finální data

• seznam čísel sektorů, na kterých je leží

↳ sektory nemusí jít za sebou → a by mělo smíšit o 2 sektory

⇒ fragmentace sektorů

• délka souboru v bytech - zjistíme, jak moc je zaplněný posledním sektorem

• identifikátor souboru (x/y/a. sek.) - x, y jsou adresáře / složky

• datum a čas...

cesta k souboru

directory / folder

- file system - specifikuje formát metadat

↳ část sektoru vyhradíme na určitámu metadata

- metadata si formátují i seznam volných sektorů

- nový disk je třeba naformátovat - zapsat do sektoru těch metadat takovou sladkou, aby to odpovídalo pravidelným diskům

- operacní systém - uložení v operací paměti

- má funkce na práci se souborovým systémem

Python: `f = open("a. sek", "r")` → OPEN, rafamaduje si ID

`f.readline()` → READ(ID)

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

→ přečte si metadata

`OPEN(path)` → ID souboru

`READ(ID, buffer, offset)`

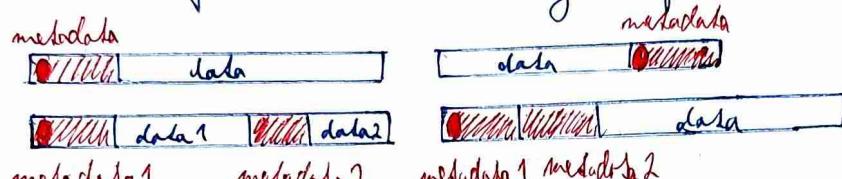
`CLOSE(ID)`

↳ offset od začátku souboru

↳ kam uložit ta data

↳ co se přičteno

Eda chce me cílit

- text file - byly toho souborů jen nějaké hodnoty znaků → text
- binary file - byly toho souborů množí nějaký něčemu popsany
→ jeho file formatu → text i čísla
- file format: říkáme, že soubor .mid má mít nějakou strukturu
→ v souboru je hlavička / hlavičky s informacemi o něm - například obecná


header - metadata
- na začátku 1. hlavičky je magic number / signature
↳ definice toho souboru dáná postupnost byteů - posuvnáček určujítoho formátu

- python: `f = open(file, rb)` → pohled na soubor jako postupnost byteů
↳ typ objektu f je jiný než normální read
- `f.read(8)` vrátí objekt typu bytes 8 byteů
↳ zavola OS-READ(10, buffer, offset, count)
↳ OS v paměti vybírá buffer, kam nabírá sektor obsahující některé data
a pak je předpijuje do toho objektu bytes
- python si pamatuje aktuální offset od začátku souboru

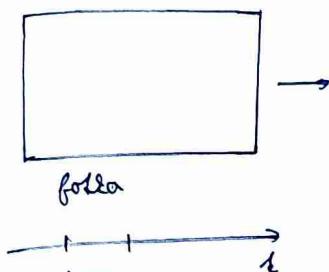
$$\begin{array}{l} f.read(5) \rightarrow +5 \\ f.seek(11) \rightarrow =11 \\ f.tell() \rightarrow 11 \end{array}$$

$$\begin{array}{l} seek(11, 0) \rightarrow =11 \\ 1 \rightarrow +=11 \\ 2 \rightarrow +=11 \text{ od konce souboru} \end{array}$$

konc - offset

- Operaciní systém
 - má nějakou tabulku s referencemi souborů
 - když stiskne soubor, tak si pamatuje nějaký rozdíl mezi ID a
nachází si tam jeho metadata
 - `f.read()` zavola funkci OS, která se konáne do té tabulky podle ID
- Obráz disku (disk image)
→ když kompletně celý obsah disku uložíme na nějakém jiném disku jako bin. file

• Reprezentace obrázků



Lzeba expozice

→ uložení obrázků - pro řádcích

"řetěz": $\begin{matrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 0,0 & 1,0 & 2,0 & 0,1 & 1,1 & 2,1 \end{matrix}$

• Grayscale - monochromatické obr.

→ 1 bpp $\xrightarrow{\text{11}}$ nejjednodušší reprezentace

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

↳ skutečně nápojné, větší ztráta, ale může se to hodit pro uC

→ dithering - iluze více barev na úkor rozmístění

1	1	0	0
1	1	0	0
1	0	1	0
0	1	0	1

černá = tmavé sedá = bílá = světlé sedá =

sedá =

→ 8 bpp $\xleftarrow{0} \xrightarrow{255} \infty$

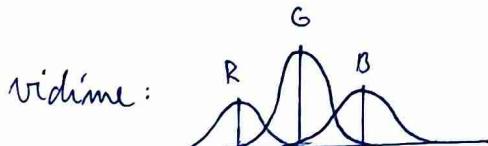
↳ musíme dobře zvolit ty hranice podle intenzity světla
→ neexistuje žádnej ideální řešení pro všechny situace

→ HDR (High Definition Range) - myslíme si řešení pro řadu fotografií

⇒ 1 exponent - sloučit \downarrow tempo

→ problém: lidé si dají totyž nerovný hrdání jasné světlo + když nemůžeme zobrazit

• Barevné - barevná frekvence



- uložení do souboru / paměti

• soubor: offsety $0, 1, 2, \dots$

• paměti: offsety $0, 1, 2, \dots$

↳ od barevné adresy

⇒ pohled na to je stejný

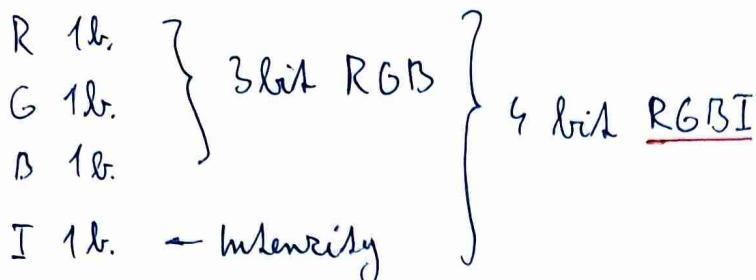
• paměť: nejdříve offset $0, 1, 2, \dots$

pixel ~ intervalu světla co má řešit plošná objedna

$\uparrow \infty$ - ta intervala reálně nemívají mezi sebou omezená
 $\downarrow 0$ - 1 pixel \rightarrow n-bit \rightarrow bpp = bits per pixel

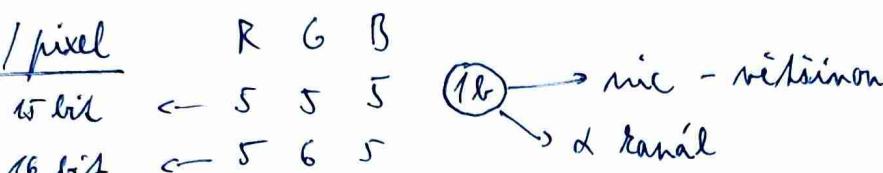
↳ barevná hloubka (bit depth)

• 4 bit



- rase můžeme dítět dithering

• 2B/pixel



High-color

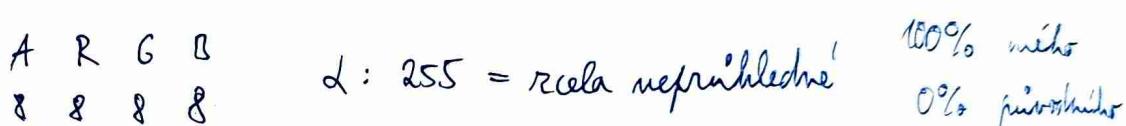
↳ ten bit navič dáme G: do vidí nejvíce odstínů G

• 24 bitová kloubka



True-color

• 32 bitová kloubka



100% něho
0% původního

↳ černe pracovat

↳ 32 bit slony

↓
chci dát nějaký svij obrázek přes původní

0 = rada průhlednosti
200% původního

✓
zářejí průzir

→ metadata obrázků

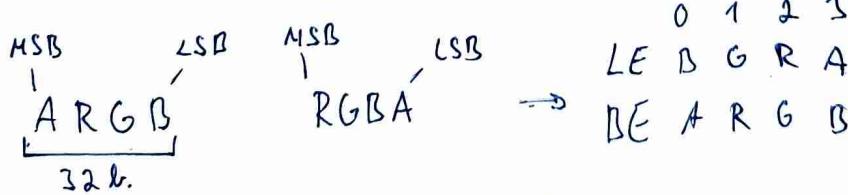
• počet kanálů - 1 monochromatický, 3RGB, 4 RGB +

• bitová kloubka - bpp + bits per kanál

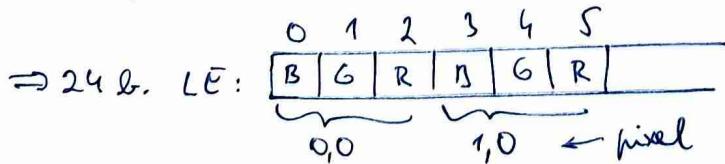
• šířka v pixelech

• výška v pixelech

→ EXIF: detaily o tom, jak se to nafotilo



hlavice souboru



⇒ Rasterní obraz / bitmapa

Reprezentace textu

↳ řetězec / string = posloupnost znaků

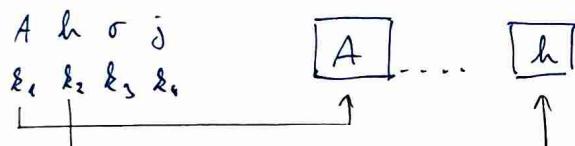
A+a obrázek \leftarrow grafem
↗

- ↳ písmeno, číslice, speciální znaky #, \$
- ↳ bílý znak / white space ↳, tab, ...
- ↳ různé znaky

→ zavedeme kódování znaku

- mapování 1 znak = 1 kód
- mapování kód \rightarrow binární reprezentace
 - a, fiksná délka - všechny kódy jsou 1B / 2B ...
 - b, proměnná délka

→ rasterizace textu - převodení kódování na obrázky



/ zde nejsou všechny kódování, jen když je obracujeme špatně

→ je důležité se dohodnout na kódování

• <u>ASCII</u>	7-bit 0-127	A, B, ... Z	a b ... z	0 1 ... 9
	$\hookrightarrow = 0x20$	$x \quad x+1 \quad x+26$ 65 66 91 $0x41$	$y \quad y+1 \quad y+26$ 97 98 126 $0x61$	$z \quad z+1 \quad z+9$ 48 49 57 $0x30$

→ 1r 7bit kódování mělásme do 8bit bydě

⇒ kódy 128-255 věnujeme nějakým neanglickým znakům

= 8bitové rozšíření ASCII

\hookrightarrow codepage

→ do těch 128 znaků se neresta ani Evropa

3 kódování: Západní, Střední, Východní Evropa

⇒ ke každému znaku kódování máme různé kódování → blbosti

→ kódování čísly ISO 8859-2 - americké ISO Latin-2 } rámce se říká
linux 852 - MS DOS DOS Latin-2 } Latin-2

windows WIN 1250 - nové od MS

• Unicode - ASCII 0-127 = Unicode 0-127

$\hookrightarrow \$D800 - \$DFFF$ nic neobsahuje
→ pro diakritické znaky

→ všechny znaky, rozsah 0- \$10FFFF - nejběžnejší znaky 128- \$FFFF

UTF-32 → form LE i BE varianty!

• Vládání - v tom pořadí, v jakém se to čte: Ahoj: A h o j
fajzur se čte: ~1~2~3: ? ~ 1 ~

• UCS-2 - Unicode 0- $\$FFFF$ jsou běžné znaky $\Rightarrow 2B$ - reprezentuje se

• UTF-16 - podporuje všechny Unicode znaky

-> nemá jenom' dílen znaku

- běžné $\rightarrow 2B$
- neběžné $\rightarrow 4B$

UTF-16 LE

UTF-16 BE

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

-> když známe velikost souboru, tak neznáme # znaků

jak víme, že je mezi 2 běžné znaky?

\rightarrow UTF-16 má často jaro dve 2B čísla, když jsou v rozsahu $\$D800 - \$DFFF$, tak užíváme třetí běžné znaky

Surrogate \Rightarrow 2 surrogates kodují 1 znak $\approx \$10000 - \$10FFFF$

• UTF-8 1, 2, 3, 4 byte' sekvence

1B = ASCII 0-127 \rightarrow ASCII kodování je stejné jako UTF-8 - pro číslice ASCII

\rightarrow česká písmena jsou 2B \Rightarrow UTF-8 je vhodné i pro češtinu

\rightarrow 1B racíma' na 0, a pot vždykdy je ASCII

\rightarrow 2B / 3B / 4B - všechny byly racinají na 1 } ostatní byly 10
no 1110 11110 ← první byly

\Rightarrow nebereme když máme všechny hodnoty, ale jaro posloupnost bylo

\Rightarrow nemusíme řešit endienitu \Rightarrow UTF-8 je jen jedno

Windows - UTF-16 LE

→ Unix, Internet - UTF-8

• New line

~~~ nl

~~~ CR nl

~~~ LF

nl  $\leftarrow$  prázdný

nl = rasterizací engine rámeček na nový rámeček

ASCII

$\Rightarrow$  CR = Carriage return  $\leftarrow \$0D$

LF = Line feed  $\leftarrow \$0A$

nl díln' prodej OS

✓

DOS  $\rightarrow$  Windows : new line = CR + LF

print()

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
- `f.readline() → str ←`
- `OS → bytes`
- `encoding` → převede to z nějakého původního kódování do Unicode (UTF-8)

 WIN → WIN 1250

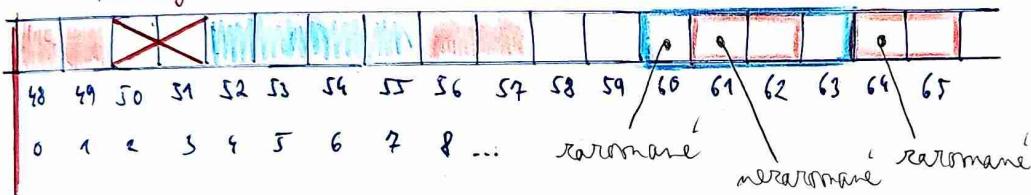
• `f = open(file, "rb")`

`f.read() ←`

`OS → bytes`

### Zarovnání dat (data alignment)

→ chceme, aby rozkladní data - čísla - ležela na plných adresách  
padding



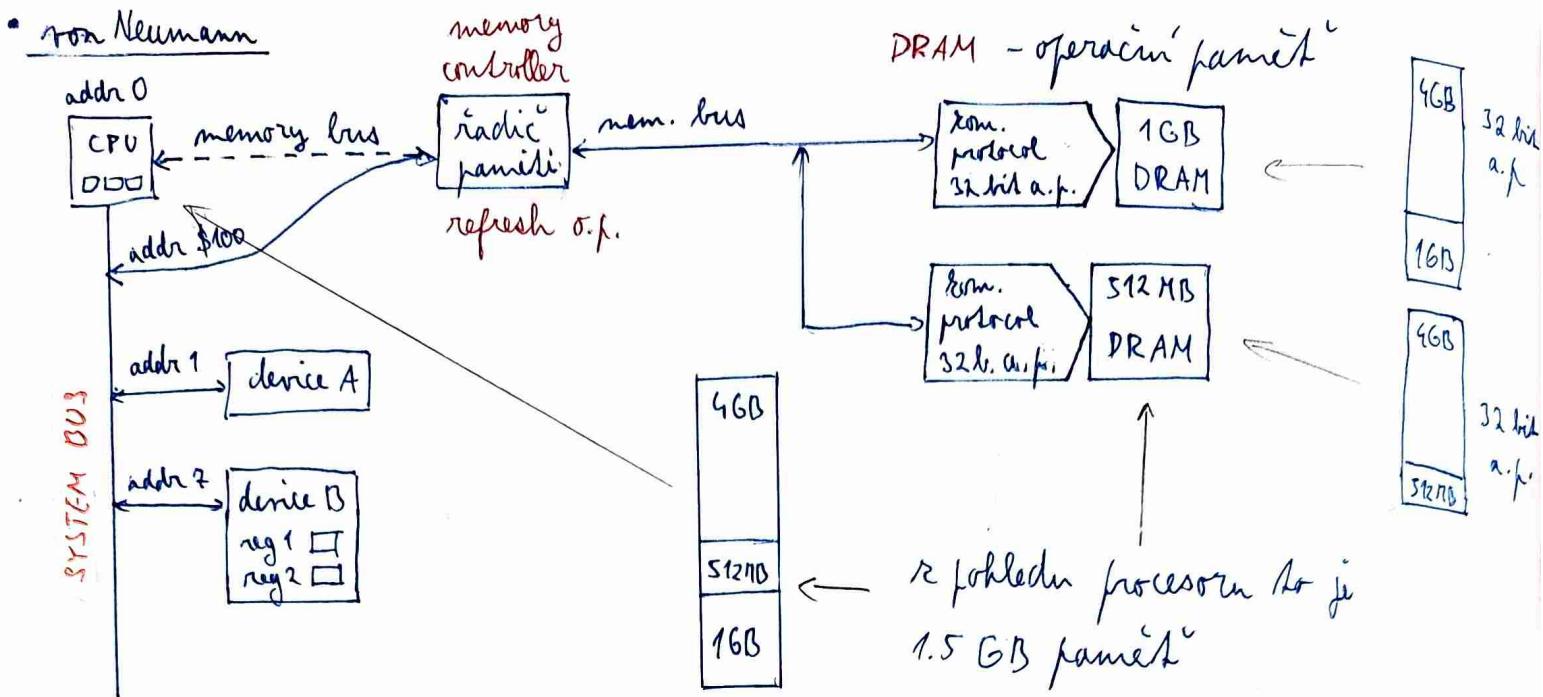
→ když máme nějaký soubor dat, tak by to celé mělo být zarovnáno včetně největší věci - min 2<sup>64</sup> ⇒ 8B → pak ně řešíme jen zarovnání na offsety

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

↳ nevyužité mezery = padding

nebo vice

→ chceme to zarovnávat, abychom na data mohli číst nor load pro libovolné blokovou paměť ⇒ 4B slova - bylo by to 2x pomalejší, kdyby to bylo misaligned



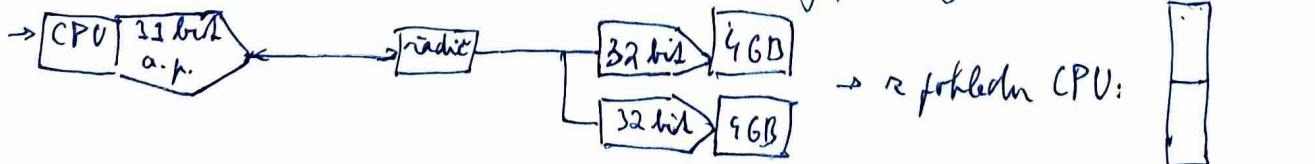
- procesor si myslí, že komunikuje s pamětí, ale ten radic paměti přijme ten paket a podle kom. protokolu vytváří jiný pro tu operací paměti
- při refreshi nemůže číst instrukce
  - ↳ moderní procesory mají cache, když se ukládají nejčastěji používaná místa v paměti

- paměťové moduly - operací paměť musíme posíládat z více modulů  $\left. \begin{matrix} 0.5 \text{ GB} \\ 1 \text{ GB} \end{matrix} \right\} 1.5 \text{ GB}$ 
  - ↳ radic paměti provede mapováním těch jednotlivých modulů na nějaké barvy adresy v paměťovém adresovém prostoru toho procesoru
  - modul 0 → base addr. 0, exp. X $B$
  - modul 1 → base addr. X

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

→ předávání pro modul 1: odečít od té adresy barvy adresu (X), se kterou paket a poslat do modulu 1

→ když má procesor X bitů a.p., tak k nim měl mít přijít paměť s a.p. Y a ten radic paměti bude posíládat ty pakety



• Systémová sběrnice - je na ní přijem řadič paměti a mechanika zařízení

↳ bylo by složitě, kdyby z procesoru mohly být 2 sběrnice

→ musí být multidrop

⇒ používá se PCI Express (PCIe) - síťová, multidrop, full-duplex

→ má 2 druhé porty

1, adresace adresou zařízení

LOAD addr  
STORE addr

2, adresace adresou v paměťovém adresovém prostoru ↘

⇒ memory write packet MWr

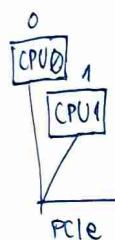
⇒ memory read packet MRd

→ procesor pošle read packet na rozhraní LOAD

→ řadič paměti posílá data a pak procesoru Completion Data (CplD) packet

⇒ pro daný odpojek je řadič master a procesor slave

⇒ procesory i řadiče paměti mají adresu - ID



⇒ CPU0 chce od řadiče číst ⇒ řadič pak pošle na adresu 0 a řadič říká že je to X

⇒ ostatní procesory/zařízení to ignorují

+ musí se identifikovat - procesor si bude očekávat něco odpojek

• memory mapped I/O (MMIO) → LOAD, STORE je něco

→ jak komunikovat s registry zařízení? → starší procesory měly speciální instrukce nařízení

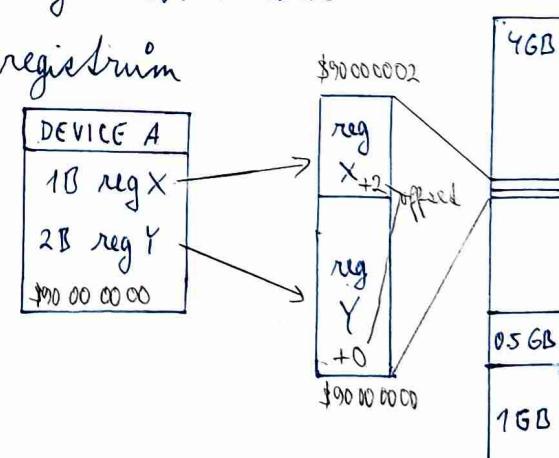
→ v paměťovém a. prostoru bude asi hodně nevyužitého místa

⇒ přiřadíme nějaké adresy z paměti k nim registru

- když chce procesor něco ze paměti, pak pošle adresu → řadič to spracuje

- když chce něco ze registru zařízení, pak pošle adresu → řadič říká, že není pro něj → ignoruje ho → spracuje ho sám registr, pro který je to vícenásobné

⇒ každé zařízení má své vlastní adresu, kde jeho registry začínají



• Host-Controller Interface (HCl) - Jen komunikací portů je vlastně dáný kon

systémovou sběrnicí → některé zařízení říšíme jenom co znamenají ty registry

⇒ jak například seber

paměti do bufferu

→ data reg = \$90000002

• buffer - bytový array (4096)

• for i in range(4096):

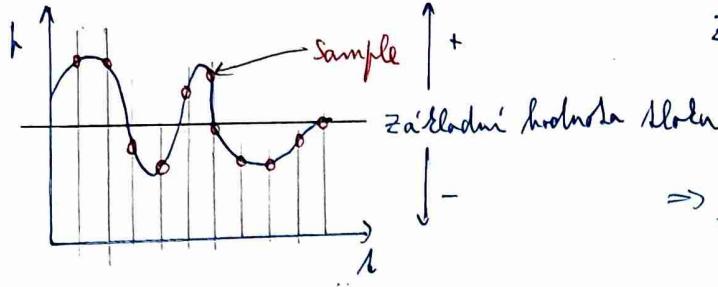
→ mov AL, [90000002, i]

→ mov [buffer+i], AL

spalování čísel a datového řádku paměti

[buffer] je šířka adresy bufferu

## • Zvuková karta



Zvuk: zahrnující několik v rozích intervalů

$\Rightarrow$  sample jsou signed čísla

8 bit  $\rightarrow$  16 bit  $\rightarrow$  24 bit

normální kvalita zvuku



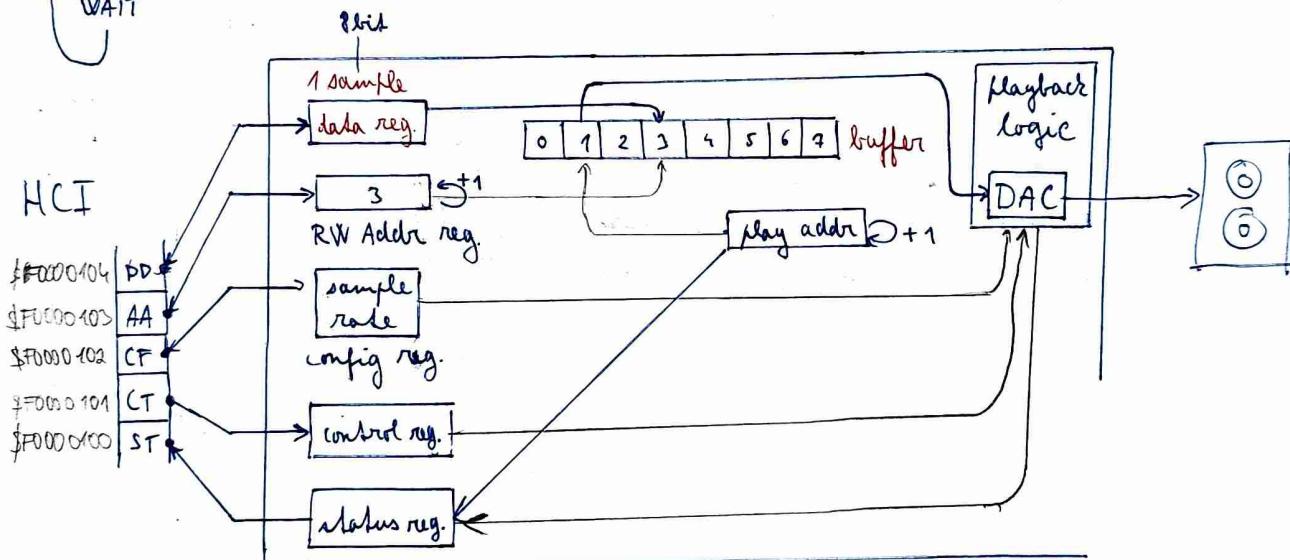
normální

$\rightarrow$  reproduktorem můžeme posílat adekvátní napětí  $\Rightarrow$  zvuková karta je DAC

$\rightarrow$  můžeme ji v pravidelných intervalech posílat k monitoru

$\Rightarrow$  vzorovací frekvence / sampling rate  $\sim 22\text{kHz} - 44,1\text{kHz} - 96\text{kHz}$

STORE  
WAIT  
 $\rightarrow$  je potřeba to dělat velmi rychle, ab procesor nemusí počítat a nežedit



$\rightarrow$  do bufferu si zvukové karty nahajíme sample třeba na 1ms dopředně

$\Rightarrow$  ta karta je pak v pravidelných intervalech posílá reproduktoru

$\rightarrow$  v adresovém reg. ji můžeme říct kam zapisovat nové sample

$\rightarrow$  do config. reg. zapisujeme sample rate, režim vzorku, mono / stereo

Podle počtu mikrofonů • mono zvuk - 1 mikrofon

mikrofonů • stereo zvuk - vícekanálový zvuk  $\rightarrow$  ukládání je možné dvojkřídlá

0 0 1 1 2 2

$\rightarrow$  control reg: stop, play, record - typická karta bude mít ADC a umí zahrávat

$\rightarrow$  status reg: playing? recording? play position

$\Rightarrow$  pro některé sample musíme řídit o endianitě toho data reg

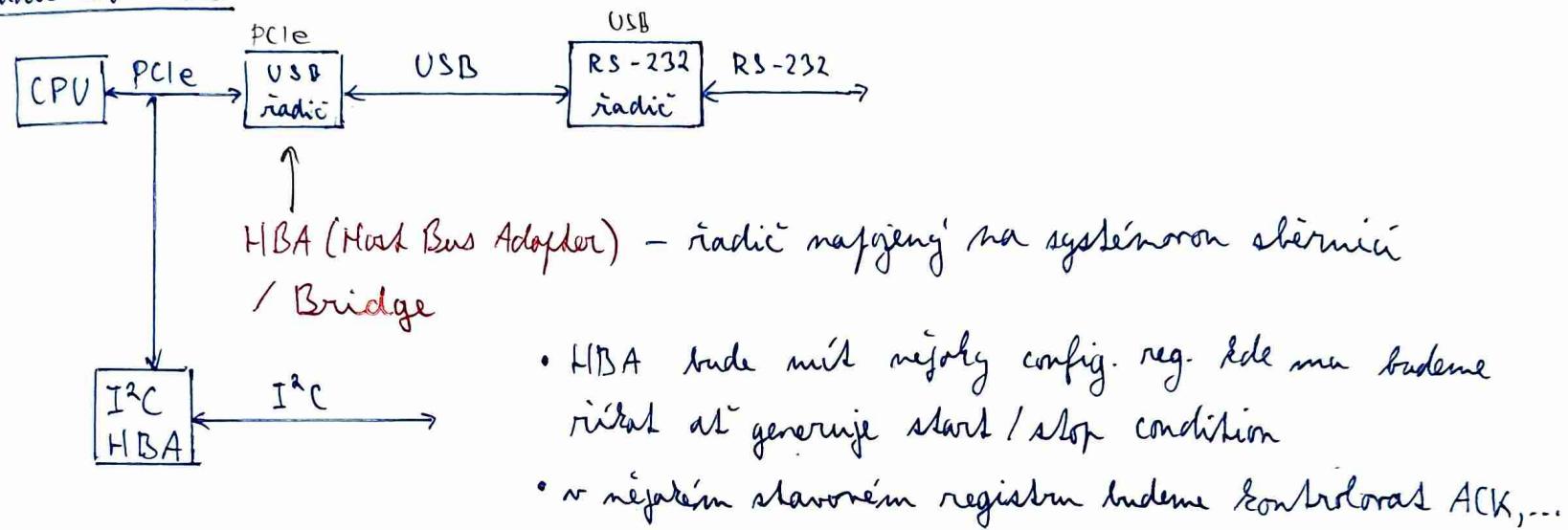
$\rightarrow$  musíme to samy zapisovat ve správné endianitě



## mixární rámci

- máme 2 posloupnosti rozveru a chceme je pletat priebežne → musíme je synchronizovať
  - mohlo by to dělat procesor, ale jde k tomu hardwarem
- ⇒ HW akcelerace (CPU offloading) — kartka umí do mixární hardwarem
- v dnešních kartač bývá nějaký jednoduchý procesor DSP (Digital Signaling Processor)

## řadič sběrnice

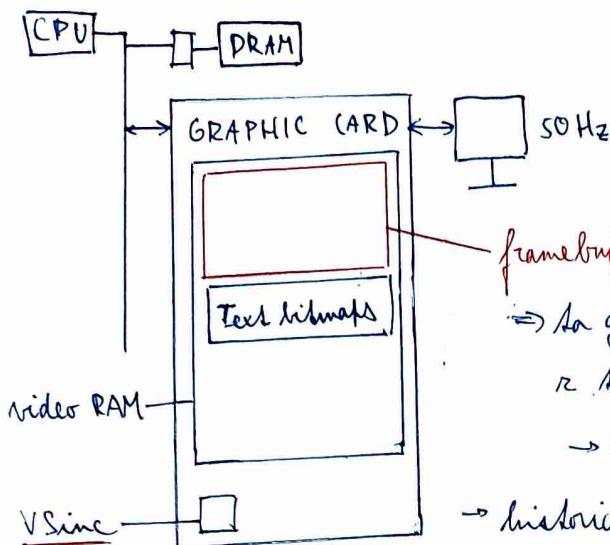


⇒ aby chom provedli 1 transakci na té sběrnici za tím (I<sup>2</sup>C), než musíme provést něco (11) transakcí na té sběrnici před tím (PCIe)

⇒ máme nějaký while cyklus kde porád loadujeme ten stavovg reg. pro ACK

⇒ Device polling — během pollingu procesor plýtvá čas

## • Grafická karta



→ Grafická karta má v sobě méně výkonů paměť DRAM, která se říká video RAM

rasterový obrázek

framebuffer - je v něm uložená bitmapa aktuálního stavu obrazovky  
 ⇒ Na grafické kartě pravidelně posílá info o pixelech  
 ⇒ V každém framebufforu jsou monitory  
 $\rightarrow 50 \text{ Hz} \Rightarrow 50 \text{ kŕát } 1/\text{s} \text{ do fosky}$

→ historicky nejedlý laser střílel elektrony na sklokoře ⇒ obrázek

## • VGA - analogová linka mezi g. kartou a monitorem

(↳ anologová hodnota U říká kolik e<sup>-</sup> střílel ⇒ intenzita obrazu  
 ↳ možné pro 3 RGB kanály)

→ dnesní CRT monitor by to musel převést pomocí ADC

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

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

→ Edgě chce měnit jeho obsah, tak rovnou měříme x/y position do libovolného pixelu

→ problem: grafická karta rytíkreslí na obrazovku → rytíkreslí všechny obrázky

a TĚD přepíše frame-buffer ⇒ dolní polovina se zobrazí jinak

→ 1 frame je divizný

vertical sync.

→ historicky když sem laser digital, tak se musel po obrazovce vrátit

⇒ je samozřejmě frame - nejedlý VSync register nám říká, jestli je to okno → frame buffer budeme měnit v tom okně

→ nebo máme 2 frame-buffery

## → HW akcelerace

1, HW rasterizace textur - ve video RAM má malé bitmapy těch textur

↳ neposíláme ji hodnoty px, ale čísla znaku → tiskárna je v té tabulce

2, Rasterizace 2D grafiky - nabíráme tam 2D vektorovou grafiku

3, Rasterizace 3D grafiky - měříme do video paměti nabíráme informace

o těch triangle meshes + textury a tím 3D rasterizační engine

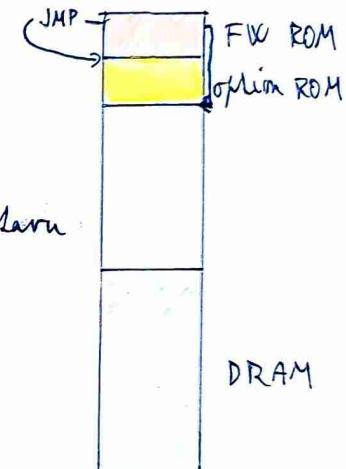
generuje obsah frame-buffern, aby to vypadalo, že se na ten model koukáme z různého úhlu

- moderní grafické karty mají programovatelný procesor, abychom mohli využívat 3D rasterizaci → do té video paměti nahráváme program ve skriptovém kódu, který ten procesor požaduje
- ⇒ tento programem se říká shadery

## • Firmware počítací

- von Neumann: pf. paměť je volatilé → kde se nachází počítační program?
- ⇒ k rádici paměti přiřazíme nejprve non-volatile ROM paměť
  - která bude nahrazovat firmware toho počítací
- ⇒ Na FW ROM nasmajíme nějak do f.a.f. toho procesoru
- ⇒ procesor má hardwired start-up vector ← počítační adresa IP po zapnutí
- ⇒ výrobce toho počítací samy může nahrať JMP instrukci na ten firmware - může si ho užít jak potřebuje

dneska  
FW ROM ~ 1MB



## BOOTOVÁNÍ POČÍTAČE

- 1, Test & Config. HW - rádiové rádice provede do počítačního stavu
  - nastaví bárové adresy registrů závislosti
  - při domluvání těch adres komunikuje se závislostmi tomu jejich adres
  - ⇒ plug & play - něco tam shrábíme a ono se to "samo" na konfiguruje aby se ty adresy někdy
- 2, naložení vnitřecného SW

→ na mother boardu toho počítací použijete další paměti se SW ← option ROM by

→ jsem rase někde nasmajované a ten FW ví edle \*

⇒ avšak dobehne, tak prodele JMP na rádius té option ROM by

\* ví, ede mužem mít - na rádiu mají nějaké ID magic number

→ programy na nich ROM kód si ponejmou někdy od operacní paměti

→ mass storage devices - 0 / metadata data

↳ boot sector - někde v něm je magic / jestli je reálně není bootovatelný

↳ jejich 0. sektor může obsahovat startující kód

↳ když obsahuje, tak si ho naopírává od operacní paměti a spadne

→ option romky dneska již existují - treba grafická karta aby se na konfigurovala

→ v boot sektoru je boot loader → ten zde v jeho sektořech je

něco vnitřecného a naopírává je do operacní paměti ⇒ další SW

3) abstrakce nad HW - do boot sektoru by se měsíel rádnyj delší program

→ ve FW jsou nějaké rádkachní funkce

- read sector - přečtení sektoru

- read key - přečtení vstupu z klávesnice

- print char - zobrazení znaku na monitor



když bootloader potřebuje  
násled sektor, tak se dělá  
hromadí kódu FW

→ historicky byl ten finální program na nejakej disketu a ten  
bootloader prostě kopíruje celou tu disketu do operační paměti

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

⇒ bootloader někam nabíráje 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

- ↳ funkční rádky, grafický shell: lišta, plocha

↓  
linux

windows