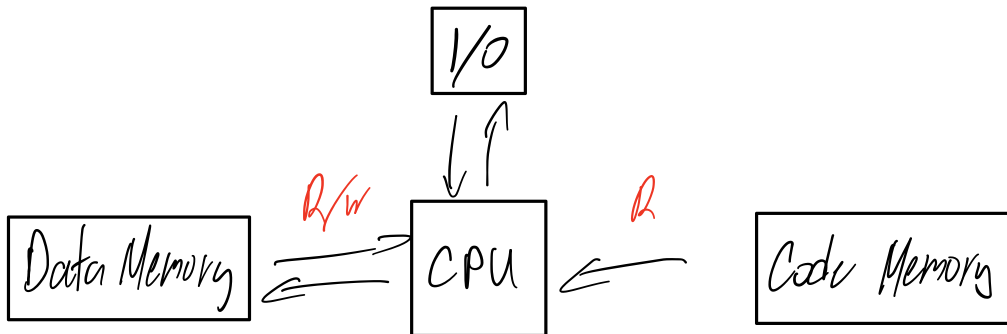
# 1. přednáška

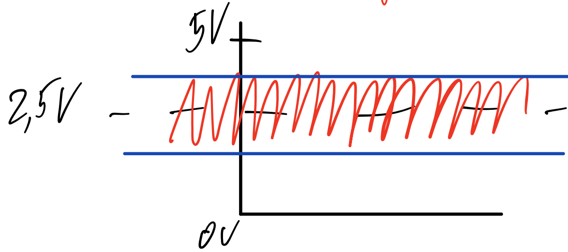
## Harvardská arch. | . Oddělení datové a kódové paměti




## Reprezentace čísel:

- nezáporné, přirozené číslo

## Princip analogového přenosu:

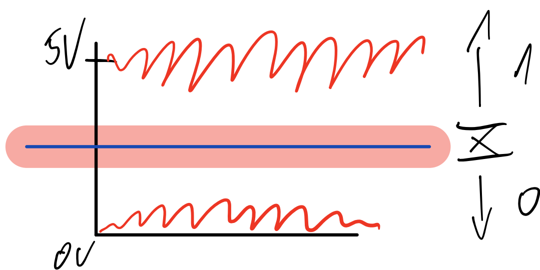



Analogový přenos

hranice, do kterého se musí vejít, abychom to vyhodnotili jako hledanou situaci.

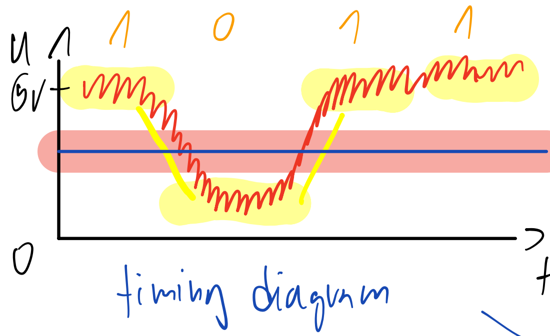
## Princip digitálního přenosu:

0V / 5V, mám jeden bit

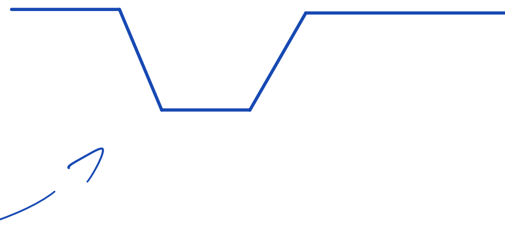



hranice mezního napětí.

## Posílání delšího čísla: (1 0 1 1)

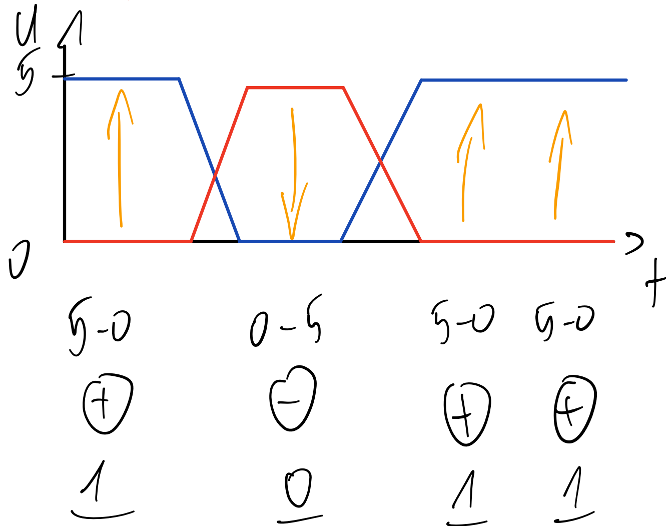



## digitální sériový přenos:



Pro měření napětí potřebuju i referenční bod... GND

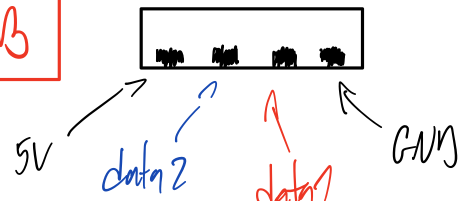
data 1/data 2




## Diferenciální sériový přenos:

- šum ovlivňuje oba vodiče téměř stejně, jejich rozdíl bude tedy téměř pořád stejný

USB




5V/GND: napájení...

## Dvojkový zápis:

| 1 | 0 | 1 | 1 |
|---|---|---|---|
| $2^3$ | $2^2$ | $2^1$ | $2^0$ |
| 8 | 4 | 2 | 1 |

$1+2+0+8=11$

LSb / MSb – first

$256 = 2^8$

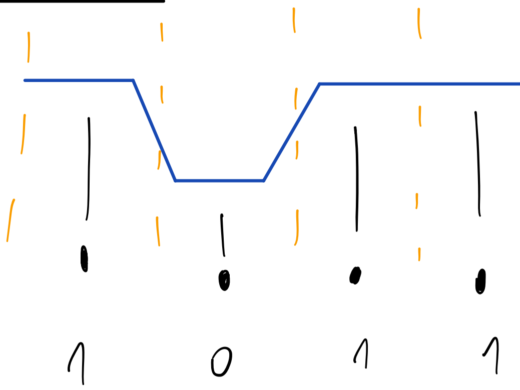
$1024 = 2^{10}$

$4096 = 2^{12}$

$65536 = 2^{16}$

Přenosová rychlost: 1 baud = (symbols per second)

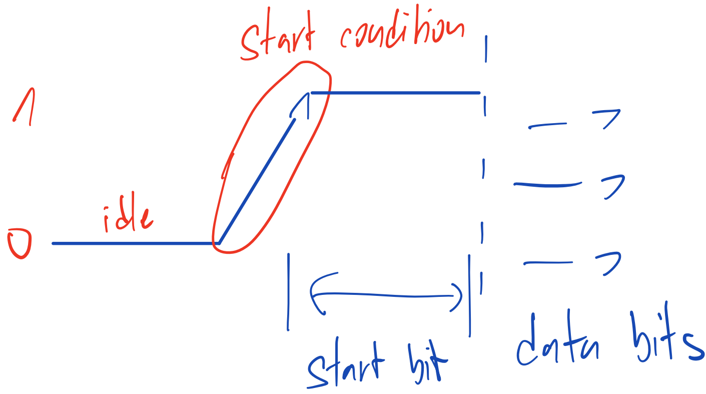
# 2. přednáška




↗ Ta správná hodnota se nabírá většinou uprostřed.

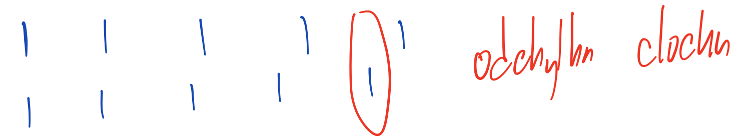
Potřebujeme ale nějaký CLOCK a IDLE stav (floating / Hi-Z)

3-state logic: 1/0/F



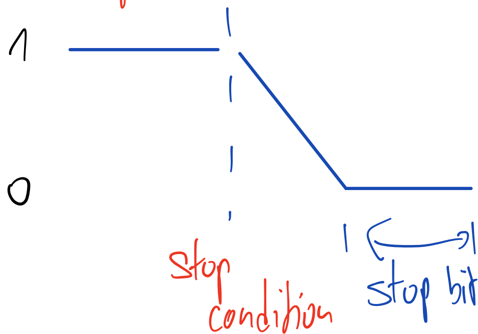

: rising edge, pak se chvíli čeká...

Pokud clock na zařízeních není stejně rychlý, brzo se to rozjede




--> const. jednotka bitů před dalším clock. sync.: 8b, 1B

Stop condition:
- napočítá x bitů, pak hned skočím a jdu do nuly.



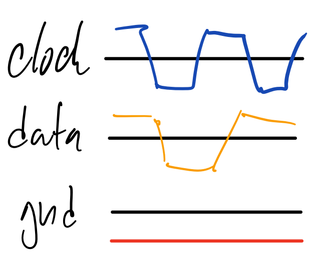

Nechť 1000 baud: (overhead?)

--> 1 stop bit
--> 1 start bit
--> 8 data bits

20% overhead

800 bits/sec

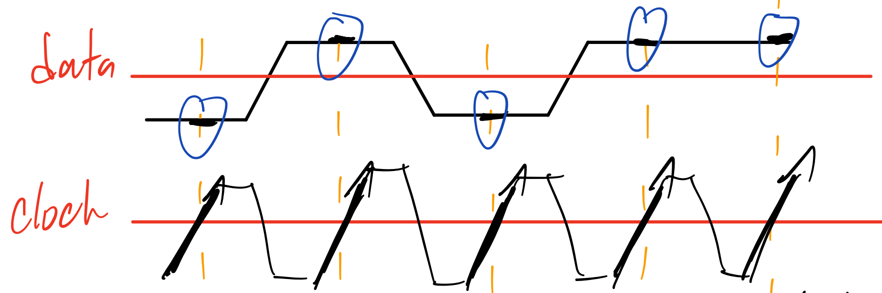
RS-232




1/0 - pravidelně

Definujeme, kdo vysílá clock, kdo poslouchá.
Rychlost je daná rychlostí clocku.

clock -> Hz
cycle/takt

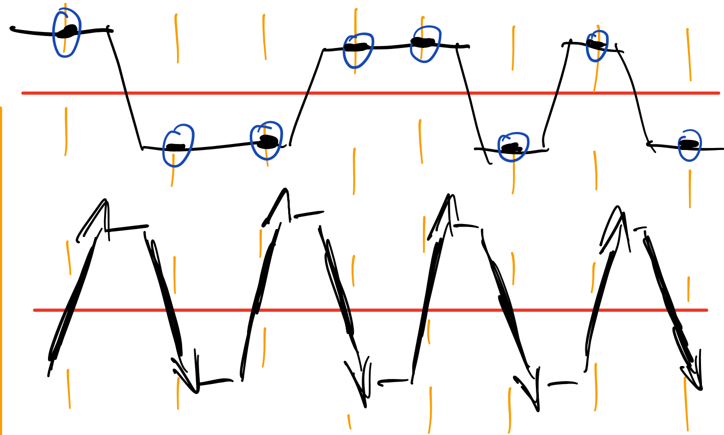



50% overhead

indikace je inkrementace napětí na clocku (falling edge)
„– teď je hrana platná"

I²C

DDR (double data rate)
- platné jsou obě falling edges



„0%" overhead

– neomezená délka přenosu

**Problém:**
odlišná délka
CLOCK a DATA
vodiče odchýlí
příjem CLOCKu a DAT.

Chceme přenést 8 bitů, pošleme 10 bitů:

00000000
11111111
$2^8$
256

0000000000
1111111111
$2^{10}$
1024

Nemůžu poslat 0000 0000 ani 1111 1111, protože tím nenajdu clock. Místo toho to namapuji na čísla z 10 bitů...

- pravidelně se střídají 1/0.
- přepisovací pravidla jsou předem dané.

$4 \cdot 256 = 1024$

CLOCK RECOVERY --> 20% OVERHEAD

USB

Simplex: ⟼ umí jen jeden

Duplex: ⟼ / ⟻ umí oba dva, vždy jen jeden

- Half: Současně vysílá jen jeden $I^2C$
- Full: Současně vysílají oba dva RS-232

RX : ↖ ↗ : RX
TX : ↗ ↘ : TX

Out of Band:

1 – True –> Můžou definovat 0/1 nějaký stavy

0 – False –> Inverzní logika: 1 – False

$\overline{SIG}$ 0 – True

/SIG

Umírám? 1 – Ne, 0 – Ano

Např.: když „dochází proud" ↗ dám tam 0. Signál je mi tři jedno.

Komunikační protokol:

definuje charakteristiku dat, jejich formát, MSb/LSb first ...

Ze samotných dat to nikdy nevyhackneme.

Paket:

1 datový set def. protokolem

délka < pevná / proměnná

Řadič:

- např: řadič RS-232 –> pro komunikaci mezi CPU a zařízením. Např.: RS-232 myš dává 7 bit, CPU očeká 8-bit.
- řadič má registry: PC si sáhne do registru. Např: registr poslední polohy a registr 0/1: byla/nebyla změna.
- zároveň má další registry, ve kterých jsou def. vlastnosti spojení (MSb/lenght...)

tedy: datový, stavový a konfigurační registr

# 3. přednáška

U RS-232 se napájení zařízení řešilo tvrdým nastavením True na OutOfBand vodičích. (→ +5V)

Blokující funkce:

read(n): n = 1024 bit → dokud nepřečte 1024 bitů, nepůjde dál

Neblokující alternativa:

Když je „n" 1024, přečte, jinak dělá něco jiného...

Semi blokující:

read(n) + timout (200) → bastard...

Btw: input() blokující

V Pythonu existuje lib. keyboard, kde existuje „if key is pressed"... neblokující

V protokolu mám definované, co který bit znamená. Pak čtu hodnoty, dívám se na jejich bity a vyhodnocuji. (Např.: RS-232 posílá hx 7bit. Byte) Musím zároveň rozeznat první byte celé jedné sady, abych nečetl jednotlivé sady špatně.

Reprezentace binárního čísla:

16-ková soustava: (hexadecimální / hex / šestnáctková)

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F   #16

$23_{16}$ / 0x23 / \$23 / 23h = $23_{16} = 2 \cdot 16 + 3 \cdot 1 = 32 + 3 = 35$ ($16^1$, $16^0$)

$00000123_{16} = 256 + 32 + 3 = 291$ (leading zeros; $16^2$, $16^1$, $16^0$)

# Proč hexadecimal?

Dobré praktické vlastnosti:

- 1 hex číslo zabírá 4 bity.
- 1B jsou tedy 2 hex čísla.

n bit → $0 - 2^n - 1$

3 bit → $0 - 2^3 - 1$

4 bit → $0 - 2^4 - 1$ (stačí 4 bity na uložení 1 hex číslice)

8 bit → $0 - 2^8 - 1$

8 bit = 1B → 2x hex # = 1B

4B = 32b

| 1B | | 1B | | 1B | | 1B | | |
|---|---|---|---|---|---|---|---|---|
| 2 | C | 0 | 0 | 1 | 0 | 0 | 0 | hex |
| 31–28 | 27–24 | 23–20 | 19–16 | 15–12 | 11–8 | 7–4 | 3–0 | |
| 0010 | 1110 | 0000 | 0000 | 0001 | 0000 | 0000 | 0000 | |

A B C D E F

10 11 12 13 14 15

– velmi jednoduché nahlédnutí, který bit je HIGH

- Ať už je reprezentace jakákoliv, v PC je vše zapsáno v binárce.

# Programově zjišťovat nastavené bity?

Bitové operace:

A: n

B: n

C: n

- vždy ovlivní jen ten jeden konkrétní bit.

# Bitové operace:

## | OR

| | A | 1 | 0 | 1 | 0 |
|---|---|---|---|---|---|
| hodnota | A | 1 | 0 | 1 | 0 |
| příkaz | B | 1 | 1 | 0 | 0 |
| | C | 1 | 1 | 1 | 0 |

- 1: ignoruj, napiš 1
- 0: přepiš hodnotu z A

## & AND (přepis jen konkrétních hodnot)

| | | | | | |
|---|---|---|---|---|---|
| hodnota | A | 1 | 0 | 1 | 0 |
| příkaz | B | 1 | 1 | 0 | 0 |
| | C | 1 | 0 | 0 | 0 |

- 1: přepiš hodnotu z A
- 0: ignoruj, napiš 0

## ~ NOT (bit flip)

| | | | | |
|---|---|---|---|---|
| A | 1 | 0 | 1 | 0 |
| C | 0 | 1 | 0 | 1 |

## XOR (selektivní flip)

| | | | | | |
|---|---|---|---|---|---|
| hodnota | A | 1 | 0 | 1 | 0 |
| příkaz | B | 1 | 1 | 0 | 0 |
| | C | 0 | 1 | 1 | 0 |

- 1: otoč bit
- 0: zachovej

## Hledáme „L-složku" v čísle:

$B_1$ | $B_2$

? ? L ? | ? ? ? ?

0 0 1 0 | 0 0 0 0 = 0010 0000 (0x20) — bitová maska, která mi ukáže jen konkrétní bit

0 0 L 0 | 0 0 0 0

- pokud =, L je HIGH
- pokud ≠, L je LOW

$B_1$ AND 0x20 = 0x20

# 4. přednáška

## Bitový posun (bitwise shift)

### SHL / Shift Left

a, b ∈ n-bitů

a << n ~> b

- a – vstup
- n – počet bitů posunu
- b – výstup

SHL -> posun k MSb

doplním 0, na druhé straně ořežu

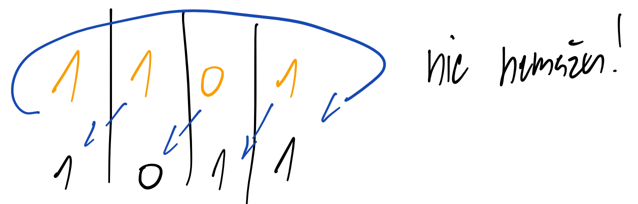
### SHR / Shift Right

a >> n ~> b

SHR -> posun k LSb

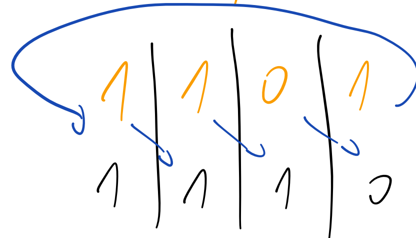
### ROL / Rotate Left

ROL -> Jako SHL, jen doplníme

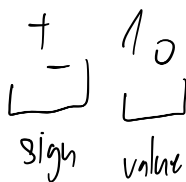



### ROR / Rotate Right

ROR -> Jako SHR, jen doplním



# Reprezentace záporných hodnot:




**Máme:**

(8-bit) Unsigned: 5 = 00000101

(8-bit) Signed: MSb = 1 => „-", MSb = 0 => „+"   ↗ rozsah: -127 _ 127

n-1 value bits
1 sign bit

-5 = 10000101 ~unsigned~> = 133
-6 = 10000110 ~> = 134

– nefungují základní operace v procesorech
-5 < -6 ⚡

(8-bit) One's compliment:   0 = -0 ⚡ – existují dvě nuly

„+" ~> unsigned(a)   5 = 00000101
„-" ~> NOT(abs(a))   -5 = 11111010   010 = 2
-6 = 11111001   001 = 1

2 > 1 => -5 > -6

0 = -0 ⚡
– zase existují dvě nuly

-5 + 1 = 11111011
00000100 = 4 ↗ repr. „-4" v One's comp.

(8-bit) Two's Compliment:

„+" ~> unsigned(a)   5 = 00000101
„-" ~> (NOT(abs(a))) + 1   -5 = 11111011
-6 = 11111010

0 je jen jedna
↗
rozsah: -128 _ 127

Všechny binární operace fungují, jenom nefunguje porovnání záporného čísla s kladným

MSb bit stále určuje +/- u One's/Two's Compliment.

Obecně se používá Two's Compliment.

Obecně je protokolem potřeba definovat, jakým způsobem se zapisuje znaménko a v jakém bitovém rozsahu.

## Reprezentace čísla v Pythonu:

| value | #platných bitů |
| --- | --- |
| | |

--> prakticky má neomezeně velké číslo

## Truncation:

5 = 0000 0101 (8 bit → 4 bit)

ostatní ztrácím ~~0000~~ 0101

21 = 0001 0101
≠ 5 0101

Truncation := $x \bmod 2^n$

Pro záporné

−2 = 1111 1110
1110 = −2

– pokud je to blízko nuly, funguje...

−128 = 1000 0000
0000 = 0

0 ≠ −128

## Zero extension:

4 bit → 8 bit

0101 = 5
0000 0101 = 5

1110 = −2
≠
0000 1110 = 14

## Sign extension:

4 bit → 8 bit

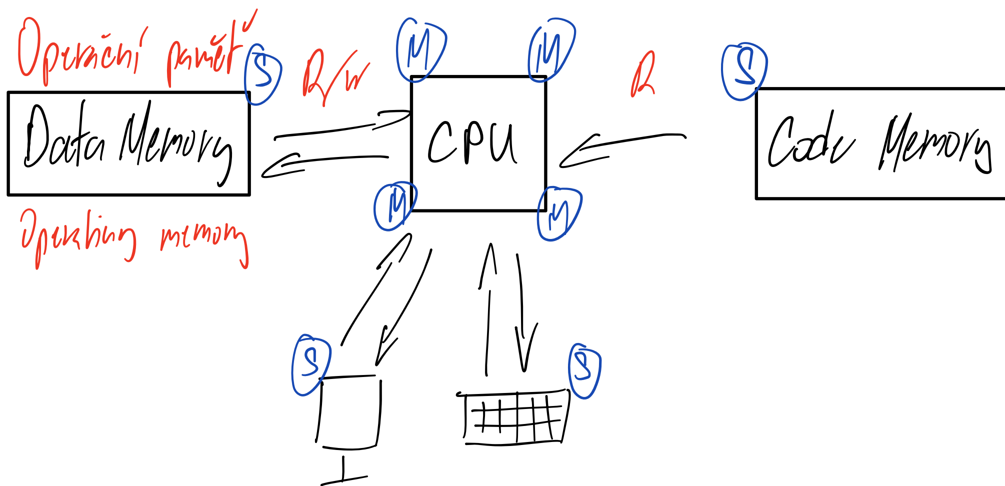
0101 = 5
‖
0000 0101 = 5

1110 = −2
‖
1111 1110 = −2

1111 = 15
≠
1111 1111 = 256

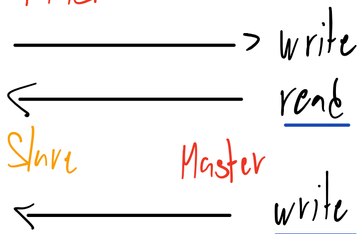
# 5. přednáška

→ typický procesor je výhradně Master

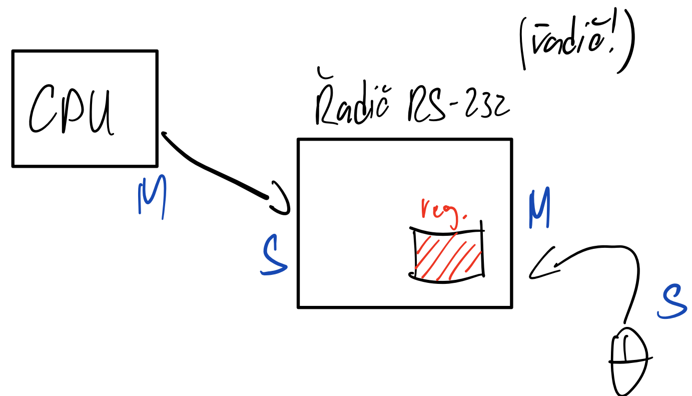



## Master vs. Slave:

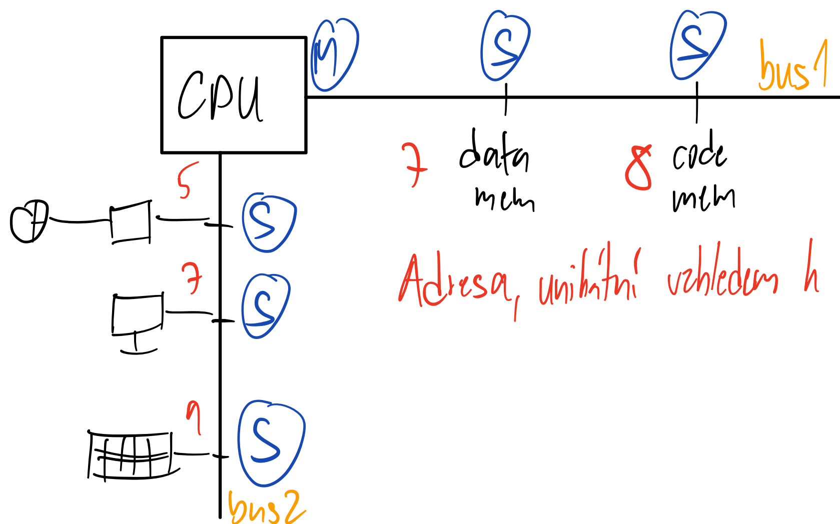
Master řídí komunikaci, Slave dává informace/reaguje




## Jak ale funguje Output only zařízení?




## Multidrop / BUS / sběrnice



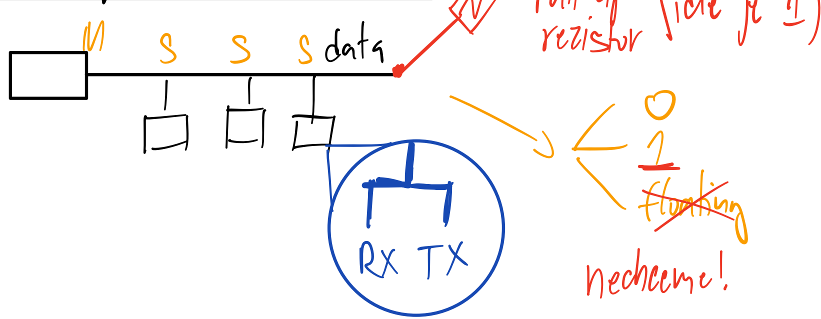

Adresa, unikátní vzhledem k sběrnici

### Musíme znát:

Rozsah adres = adresový prostor

- maximální # zařízení

Nechť je bus half-duplex:

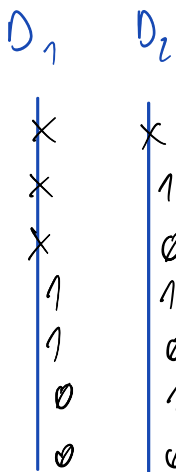



Pull-up rezistor (idle je 1) → vytáhne to do True

← 0
1
~~floating~~ nechceme!

nikdo → 1
nechci nic → — odpojeno → 1
chci 1 → — odpojeno → 1
chci 0 → připojím 0V → 0

Odpor na slave je daleko menší než na pull-upu, takže po připojení ten proud začne téct tím menším odporem a v obvodu bude napětí blízko nule.

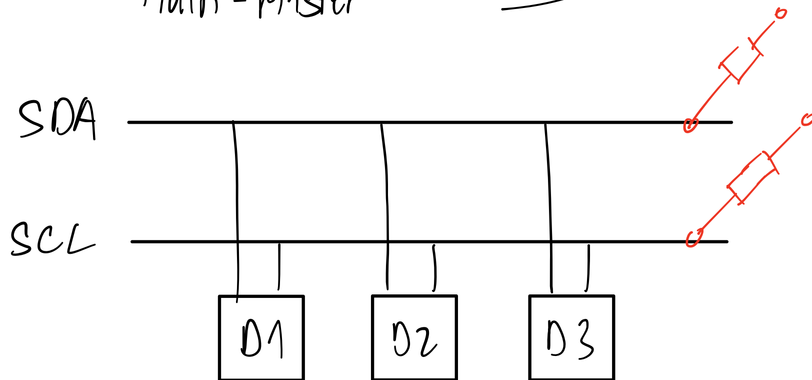


| $D_1$ | $D_2$ | Výsledek (btw. AND) |
|---|---|---|
| X | X | 1 |
| X | 1 | 1 |
| X | 0 | 0 |
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

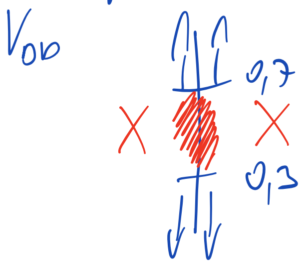
→ výsledek je deterministický (1/0)

# $I^2C$ (inter-integrated circuit)

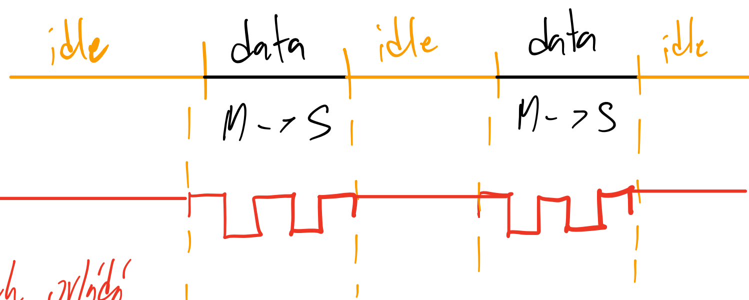
- half-duplex SDA (s pull-up)
  hodiny SCL (s pull-up)
- Multi-master → USB je single master




- pevně definované úrovně logických napěťových mezí



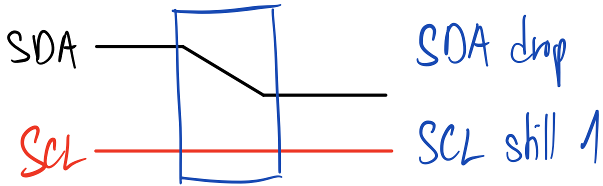

- definuje se logika SCL
  - když SCL high, data jsou platná



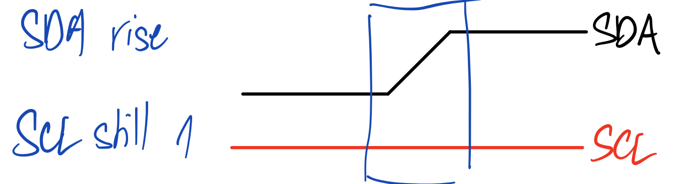

clock ovládá master.
Když není master, clock je 1.

## Start / Stop condition?

### Start condition




### Stop condition:




## Podoba prenosu:

9 bit byty = 8 bit data + 1 bit $\overline{ACK}$ (0) — NACK (1)

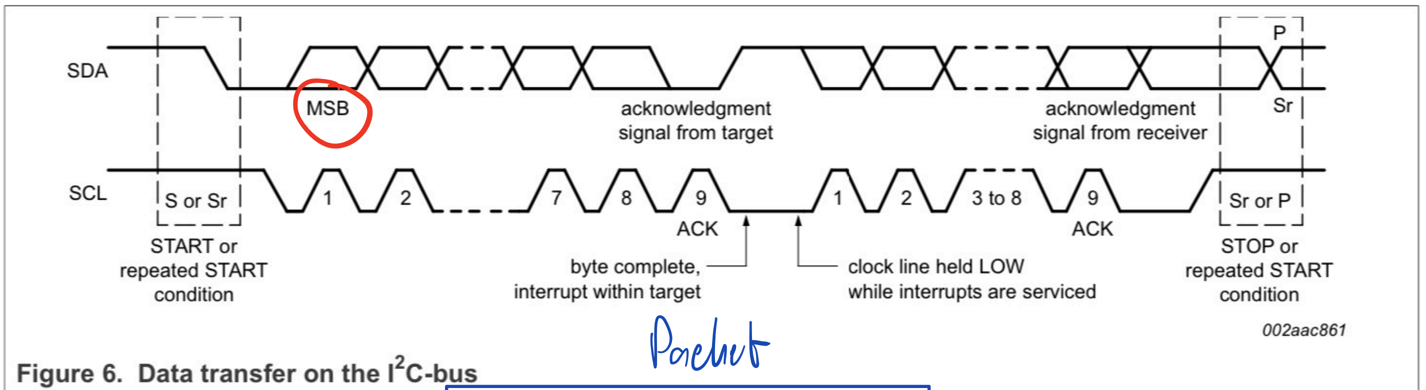
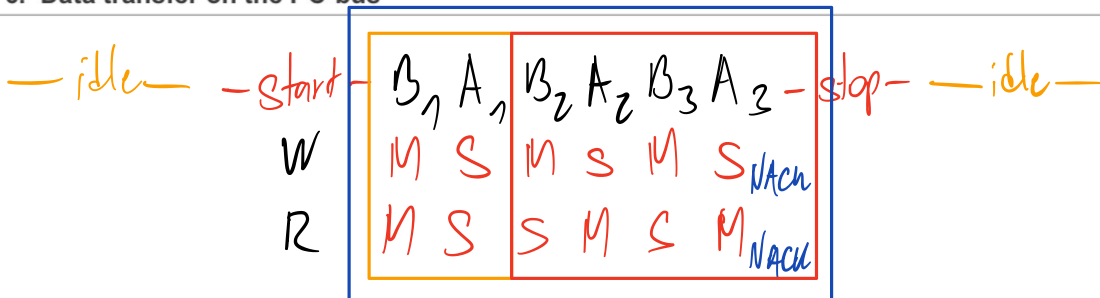


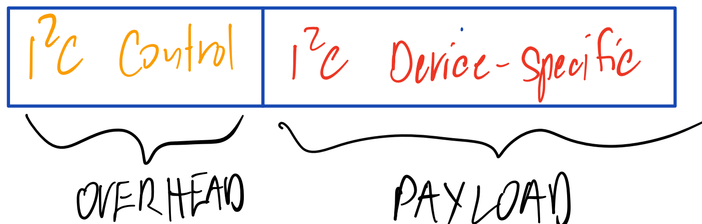
Figure 6. Data transfer on the I²C-bus

Packet




první byte má při R/W vždy stejnou formu (adresa...)

## Packet:



## I²C Control byte: || I²C Device specific:

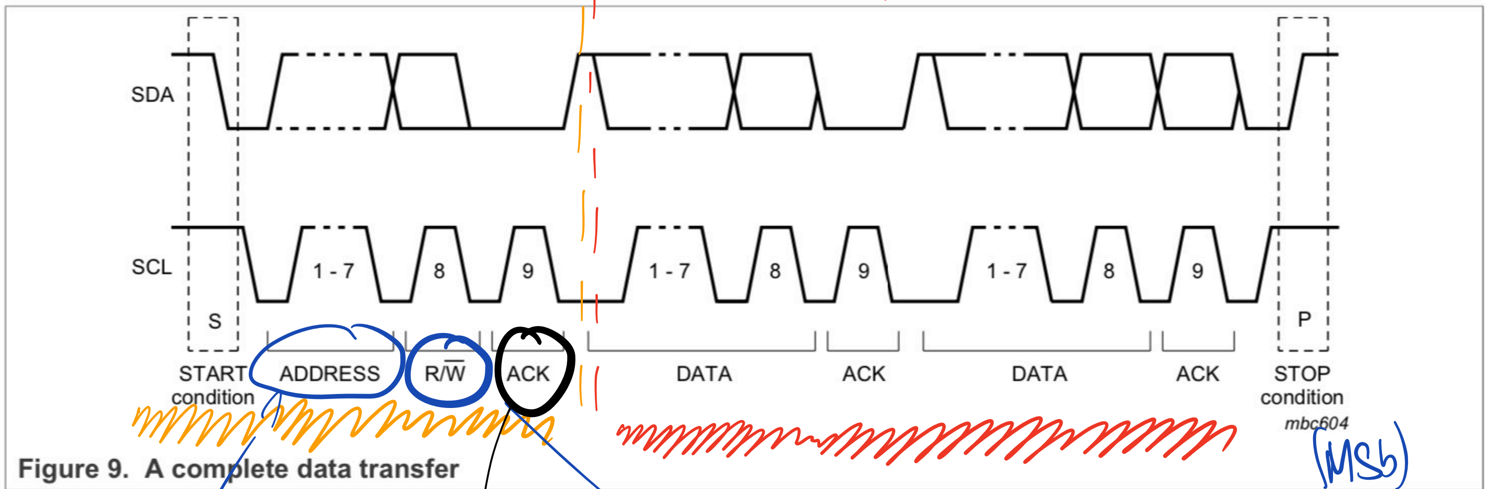


Figure 9. A complete data transfer

posílá se nejdřív adresa komu posílám příkaz

pak definuji operaci

R/W̄

R: 1 = ANO, 0 = NE

W: 1 = NE, 0 = ANO

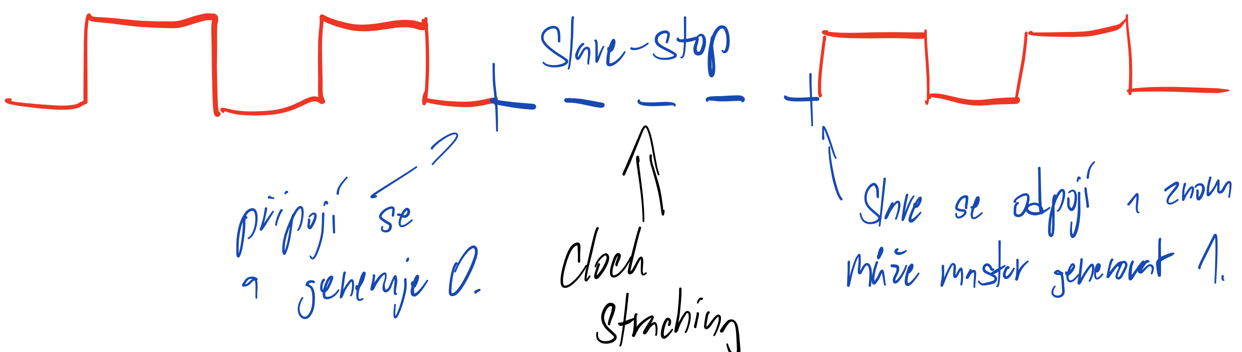
Díky pull-upu datové linky budu mít „nikdo tu není“ odpověď bez připojeného zařízení automaticky. Zároveň pokud by byl slave zaneprázdněn, pošle NACK a master pošle STOP condition.

## Přenosová rychlost:

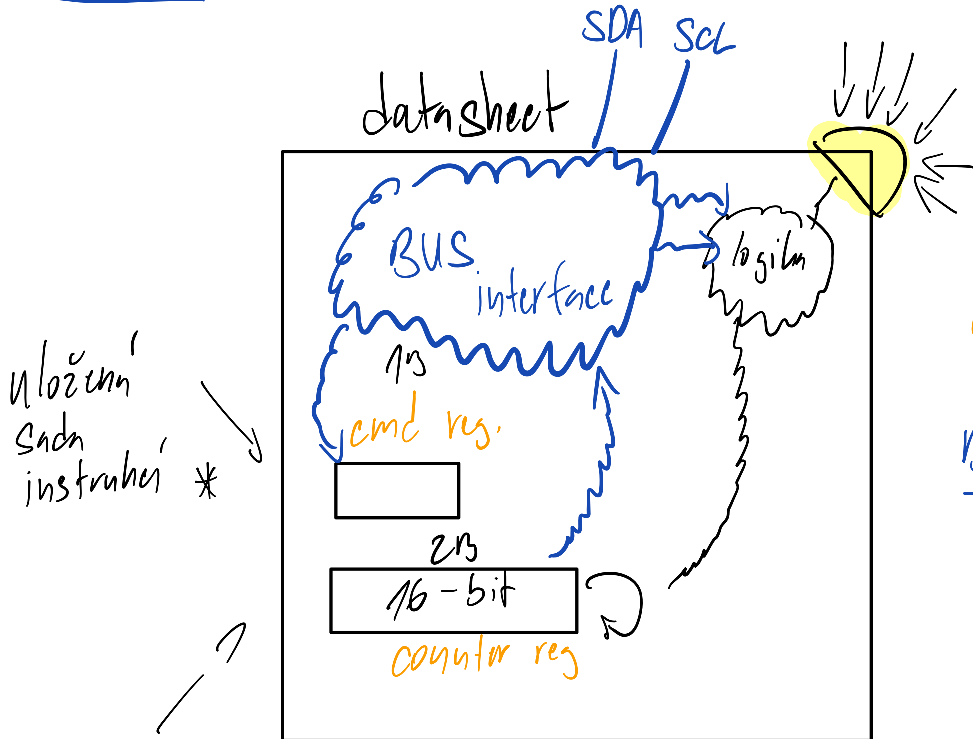
100 kHz – 5 MHz —> relativně pomalá frekvence

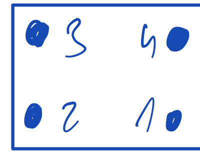
## Pokud slave nestíhá: (CLOCK HOLD LOW)

– slave může do SCL připojit svůj rezistor a držet SCL dostatečně dlouho na 0. Master SCL sleduje a když to vidí, svůj další takt pustí až slave povolí.



## Příklad: (ALS – Ambient Light Sensor)




1) $V_{DD}$
2) GND
3) SDA
4) SCL

cmd –> command register

číslo: 0 – 32 000

* logika dostane instrukci a podle cmd reg. udělá činnost

### Bus interface:

hard-wired (zadrátované):
– adresa: 0x29

pozor, to není hodnota celého bytu, protože LSb je R/W informace

v rámci jedné $I^2C$ musí být všechny adresy jednoznačně určené

## Registry:

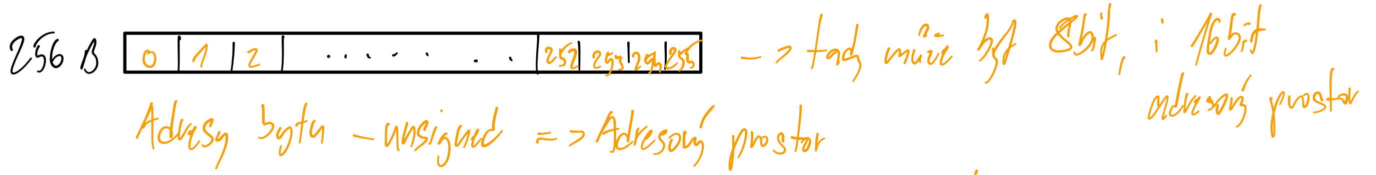
W/O – write only
R/O – read only
R/W – read write

V BUS se obecně musí definovat cíl zápisu, zdroj čtení

- Pokud zapisujeme více než bytovou hodnotu, musíme definovat byte-order

LSB/MSB first.

# 6. přednáška




256 B → tady může být 8 bit, i 16 bit adresový prostor

Adresy bytu – unsigned ⇒ Adresový prostor

200 B → opět adresový prostor musí být 8bitový, některé adresy nevyužiju.

128 B → 7bit prostor → ale i 8 bit prostor...

Je výhodné, aby byl adresový prostor rovnou velký, aby program dokázal uadresovat velké paměti.

## Zkratky velikostí:

Udává celkem dokážu uadresovat:

1 kB = 1024 bytů (≈ 1000 B) (10 bit → 1024 adres) → 16 bit prostor (16 − 10) → 64 kB

1 MB = 1024 kB (20 bit) → 24 bit prostor (24 − 20) → 16 MB

1 GB = 1024 MB (30 bit) → 32 bit prostor (32 − 30) → 4 GB

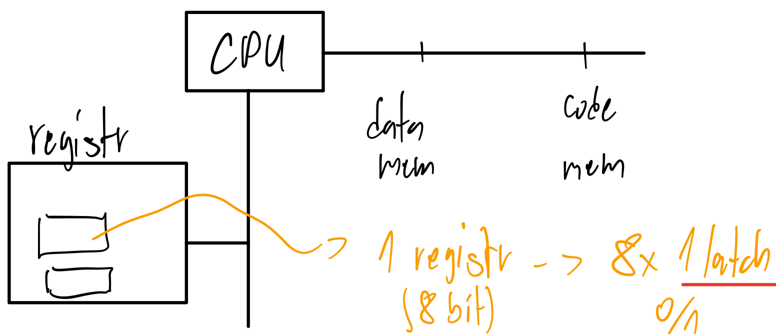
1 GiB

1 MiB – jelikož uadresování umí 1024 adres tak proč je nevyužít...

1 KiB

Prodejci disků:

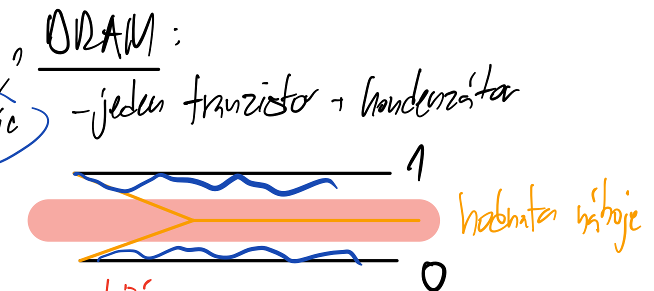
1 kB = 1000 B ☹




1 registr (8 bit) → 8x 1 latch 0/1

SRAM (static): (~~Random Access~~ Memory) dnešní RAM
- volný výběr bytu bez sekv. čtení
- uniformní rychlost přístupu

DRAM (dynamic):
- jeden tranzistor + kondenzátor




potřebuje refresh

kvůli refresh jsou pomalejší

| SRAM | | DRAM |
|---|---|---|
| schvreni ↑ | | schvreni ↑ |
| schvreni ↓ | | schvreni ↓ |
| random | | random |
| volatilní | | ultra-volatilní |
| proto není vhodná pro code mem. | | zapomene vše za tak 1ms i v napájení |
| 1B - kB - 1MB | kapacita | 1GB - 10GB |
| 10 - 100 GB/s | přenosová rychlost | 1 - 10 GB/s |
| 1ns | random access | 10ns |

# Příklad I²C SRAM paměti:

- většinou mají Analogově nastavitelnou adresu (HIGH/LOW). V datasheet definováno

## Jak vypadá zápis:

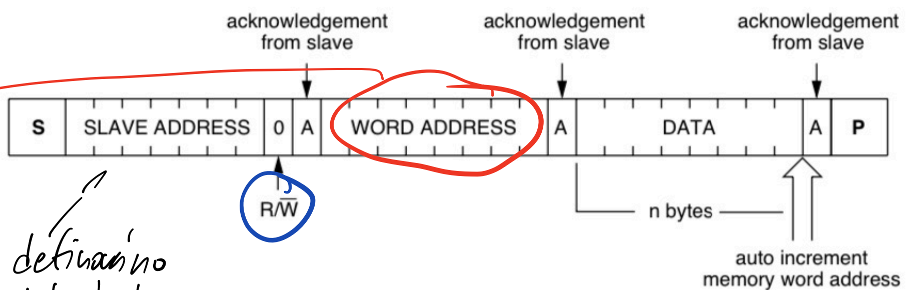



Fig.7 Master transmits to slave receiver (WRITE) mode.

Word - jednotka přenosu
- 8bit slovo -> přenese 8bitů
- s jak moc velkým číslem umím pracovat

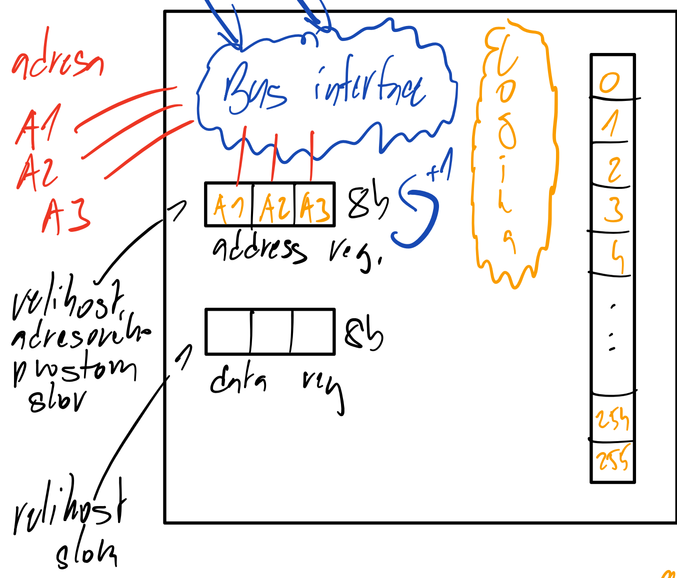
Paměť:

| pohled CPU: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| slovo 16-bit | 0 | | 1 | | 2 | | 3 | | 4 | | 5 | |
| slovo 32-bit | 0 | | | | 1 | | | | 2 | | | |

slovo 0 = byte 0 a 1

slovo 0 = byty: 0,1,2,3

SCL SDA – vždycky se vrací celé slovo – chci přečíst byte 0 v 32bit slově
→ vemu slovo 0, byty 0,1,2,3 a tam hledám...




adresa není uložena, je to jen náš popisek

$I^2C$ 100 kHz → 100 000 b/s
↓
12500 B/s

tolik bytů přeneseme

$X/(3\cdot 9)$

adresa, word adresa, data

3 708 B/s zápis

RANDOM

za nesekvenčního přístupu

## Burst přenos:

auto-increment slova → hned posílá další data dalšího slova, protože většinou chceme více dat sekvenčně?

$(X-(2\cdot 9))/9$

11 109 B/s zápis

BURST

## Zápis:

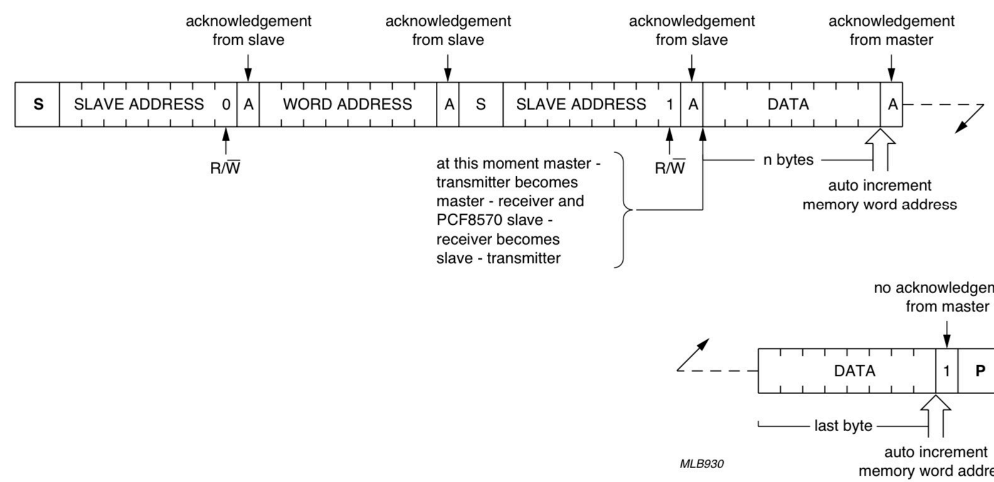
- pošli data do buňky z registru 1 $I^2C$ word 2B

speed(write) > speed(read)

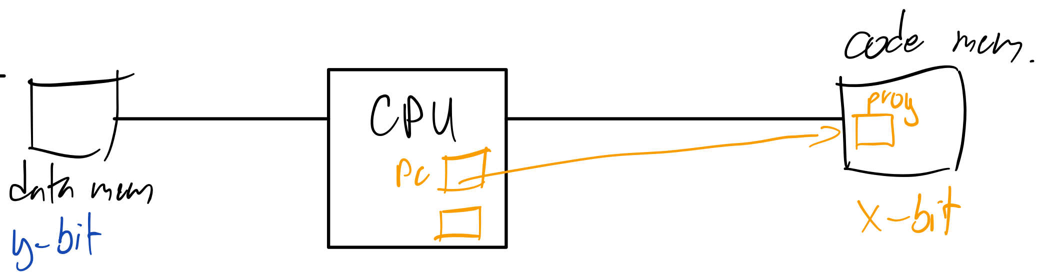
## Čtení:

- 1 write address reg. → 1 $I^2C$ word 2B } 4B
- 1 read data reg → 1 $I^2C$ word 2B

pokud mám více registrů, mám reg. adr. prostor, takže i definuji, do kterého registru chci psát/číst z.

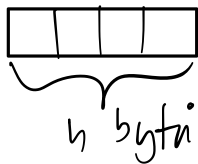


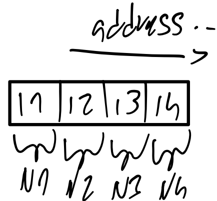
# 7. přednáška




## Instruction (Instruction set)

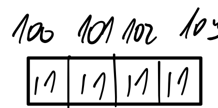
- instrukce / funkce procesoru




$IP = IP + N_1$ → posun na další instrukci

- jestli mám instrukci na více adresách, udává se nejnižší adresa (bázová)



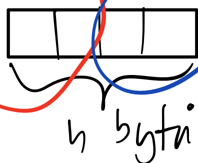

base add = 100

offset od bázové adresy

Instruction pointer

OP code

Arg. / implicitní / explicitní




jednoznačný identifikátor

→ to tahám z code memory

Op code je ve velikosti slova code memory

arg. je ve velikosti slova data memory

→ to tahám z data memory

## Registr procesoru:

- program counter (stav na jaké instrukci jsem)

## Machine code:

- série instrukcí

## Compiler:

→ chodí se přímo do adres paměti

- překlad textového zápisu do strojového kódu

## Interpret:

→ je pomalý: na jednu instrukci kódu se provede spoustu instrukcí interpretu.

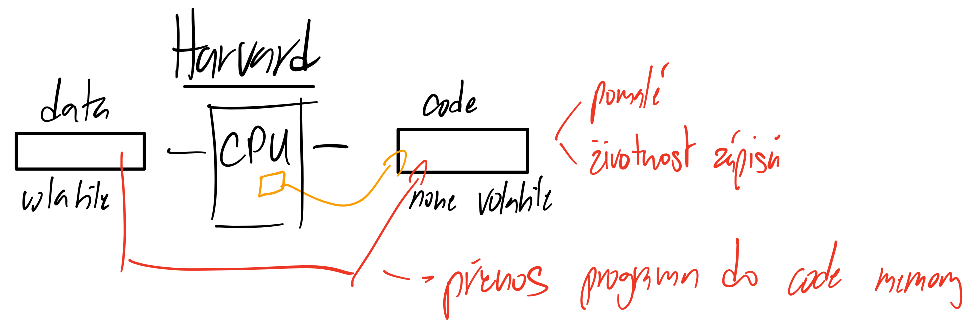
- rovnou z místa přečte a spustí

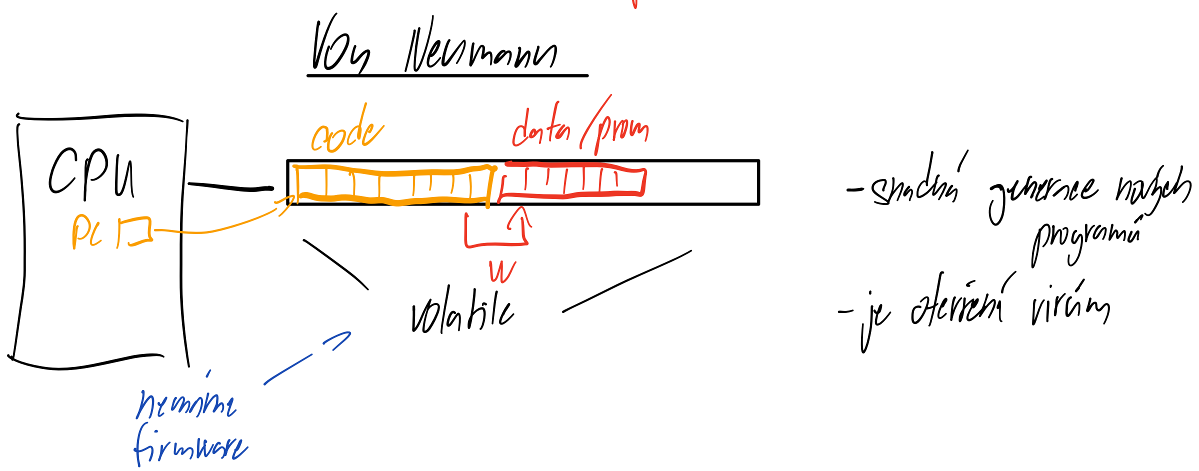
# Endianita dat: Little/Big

Little: LSb first —> 78 56 34 12
Big: MSb first —> 12 34 56 78

0x12345678

- pořadí zápisu jednotlivých bytů vícebytové hodnoty
- potřeba dohodnout předem (dáno procesorem, Intel —> Little End)




# Procesor MOS 6502

- 8 bit - 8 bit word - umí sčítat dvě 8 bit čísla
- 16 bit adresový prostor

## Assembler:

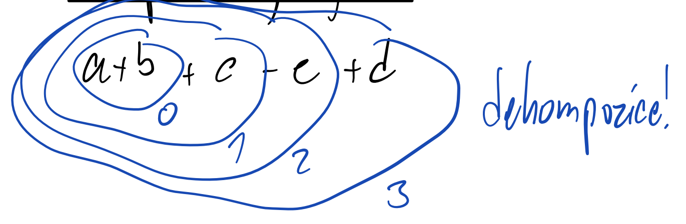
- jednodušší textový zápis instrukcí procesoru pro snadnější správu

### Registry:

- speciální / Program counter
- obecné = mezivýsledky

```
a = b+c    LOAD   addr(b) ~> R1
           ADD    R1 + addr(c) ~> R2
           STORE  R2 ~> addr(a)
```

## Komplikovaný výraz:




## 6502/x86 neumí operaci

s dvěma argumenty, vždy s jedním registrem a jedním argumentem.

# Basic operations:

## Do nothing

| 6502 | | Intel x86 |
|---|---|---|
| Ø | offset from base | Ø |
| $EA | machine code | $90 |
| PC := PC + 1 | program counter | EIP := EIP + 1 |
| NOP | Assembler | NOP |

## Jump

2 byte argument (6502: bytes 1–2) · 4 byte argument (Intel x86: bytes 1–4)

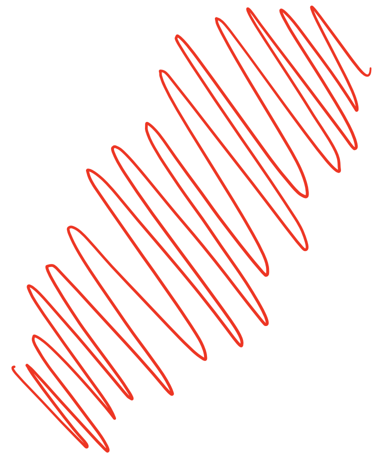
| 6502 | | | | Intel x86 |
|---|---|---|---|---|
| Ø | 1 | 2 | offset from base | Ø 1 2 3 4 |
| $4C | $xx_0$ | $xx_1$ | machine code | $E9 $xx_0$ $xx_1$ $xx_2$ $xx_3$ |
| PC := $xx_1$ $xx_0$ | | | program counter | EIP := $xx_3$ $xx_2$ $xx_1$ $xx_0$ |
| JMP $ $xx_1xx_0$ | | | Assembler | JMP $xx_3$ $xx_2$ $xx_1$ $xx_0$h |

Little endianita!

## (immediate) Load (instruction)

| 6502 | | |
|---|---|---|
| Ø | 1 | offset from base |
| $A9 | $xx_0$ | machine code |
| PC := PC + 2 | | program counter |
| LDA #$ $xx_0$ | | Assembler |

A := $ $xx_0$



# (immediate) Load (absolute address)

| 0 | 1 | 2 |
|---|---|---|
| $AD | $XX_0$ | $XX_1$ |

PC := PC + 3

LDA $ $XX_1XX_0$

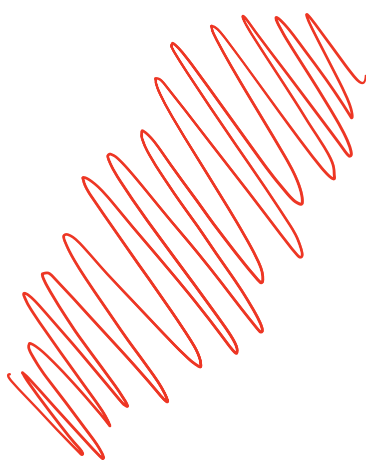
A := MemReadByte($ $XX_1XX_0$)

offset from base

machine code

program counter

Assembler



Existují tři registry v 6205:

| | A | X | Y |
|---|---|---|---|
| | LDA | LDX | LDY |
| load from argument | $A9 | $A2 | ... |
| load from address: | $AD | $AE | ... |

# Store

STA $XXXX

STX $XXXX

STY $XXXX

| | |
|---|---|
| A | 8bit |
| X | 8bit |
| Y | 8-bit |
| PC | 16bit |

# Copy (Transfer)

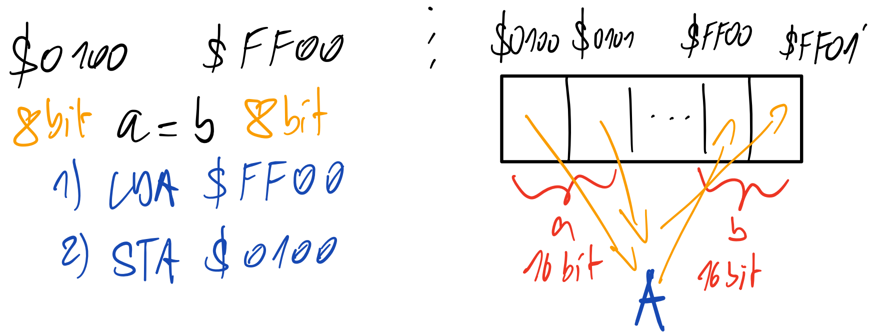
TAX Transfer from A to X

TXA

$0100 $FF00

8bit a = b 8bit

1) LDA $FF00
2) STA $0100




1) LDA $FF00
2) STA $0100
3) LDA $FF01
4) STA $0101

v 8bit slow

2x více!

## Příznakový registr: (flags register) (reg. P)

- flag (1-bit info)

N byte ... | flag | ...

- zero --> byl poslední výsledek nula? 0/1
- sign/negative sign --> byl poslední výsledek záporný? 0/1
- carry (přenos) --> má různé funkce

často pro conditional jumps

CLC (clear carry) P.carry := 0
SEC (set carry) P.carry := 1

### 6502

- akumulátorová architektura
- reg. A je akumulátor
  - většina operací pracuje jen s A.

### Bitové operace (na 6205 8bitové)

- všechny pracují jen s akumulátorem

| | | | |
|---|---|---|---|
| ORA | A | Bitwise/or | imm/addr |
| AND | A | Bitwise/and | imm/addr |
| EOR | A | Bitwise/xor | imm/addr |
| ASL A | A | shift left | |
| LSR A | A | shift right | |
| ROL A | A | role left | |
| ROR A | A | role right | |
| ? NOT | | EOR #$FF | |

$a = a \mid b$

$a, b \in$ uint_8
a = $A000
b = $B000

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

6502 Little End

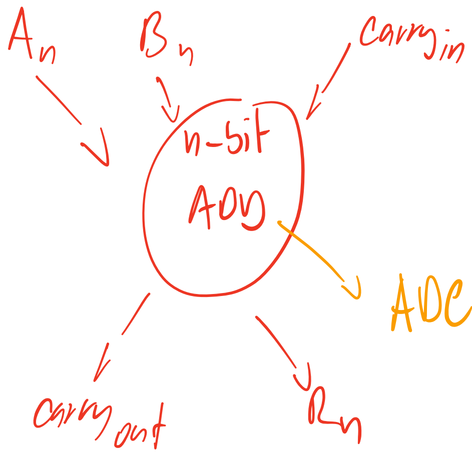
$a, b \in$ uint_16

$A000, $A001 $B000, $B001

```
LDA $A000    LDA $A001
ORA $B000    ORA $B001
STA $C000    STA $C001
```

## Sčítání:

- normalizace (zero/sign extension)



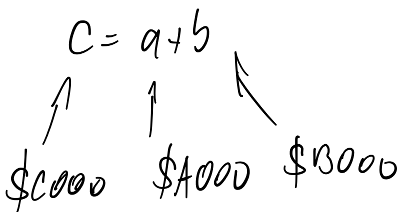

Součet můžu reprezentovat pomocí 2-bitů:

0_3 → A 0 . . . 1
B 0 . . . 1
carry = C 0 . . . 1
všechny možné varianty

- takto mám sčítací jednotku pro n-bitů, kterou můžu řetězit

ADC imm/addr

result := A + imm/addr + P.carry (tohle je 1 bit)
P.carry := result.8 (poslední bit jde do registru)
A := result.7 . . . result.0 (zbylých 8 bitů je můj výsledek)




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

→ musím si pro jistotu carry před ADC vynulovat.

$c = a + b$, $a, b \in$ uint_16 —> jelikož je Little End, musím nejdřív počítat LSB a jít postupně k MSB.

$C000, $C001 ← c; $A000, $A001 ← a; $B000, $B001 ← b

```
LDA $A000
CLC
ADC $B000
STA $C000
LDA $A001
ADC $B001
STA $C001
```

(u LDA $A001) —> teď si carry NEMAŽU!

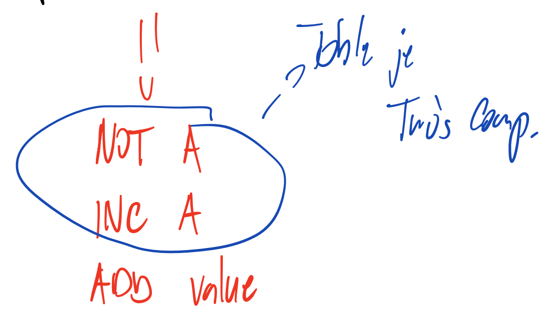
—> truncation (zahazování posledního carry)

## Sčítání znaménkových čísel:

- funguje úplně stejně jako unsigned, pokud jsou čísla ve Two's compliment formátu.
- Všechny binární operace fungují, takže provedu součet a pak jen jinak reprezentuji výsledek.

## Odčítání: ?

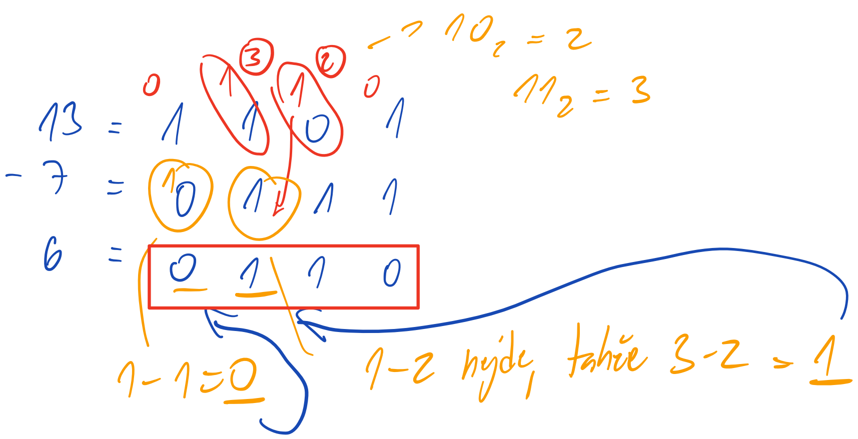
A = value − A —> A = value + (−A) —> A = −A + value




v 6205:

```
EOR #$FF
CLC
ADC #1
CLC
ADC #value
```

### Subtract with borrow (SBB) (x86)




ukládá se borrow do „carry" bitu.

# Subtract with carry (SCC): 6205

- Stejní jako SBB, jen

| carry = 1 => Borrow = 0 |
|---|
| carry = 0 => Borrow = 1 |

Negace!

$A - X - B$

↓

$A - X - B + 256$ → $256_{10} = \cancel{1}0000\ 0000_2$ (truncation → no effect)

$A - X - (1 - c) + 256$ → $B = NOT(c) = 1 - C$ (1-bit)

$1 - 1 = 0$

$0 - 1 = 1$

$A - x - 1 + C + 256$

$A - X + C + 255$

$A + (255 - X) + C$ → $255 - X = \alpha$

$255 = 1111\ 1111$
$-\ něco = 1010\ 0010$
$= 0101\ 1101$ NOT!

$A + NOT(X) + C$

| $\alpha = NOT(X)$ |
|---|
| ADC $\alpha$ |

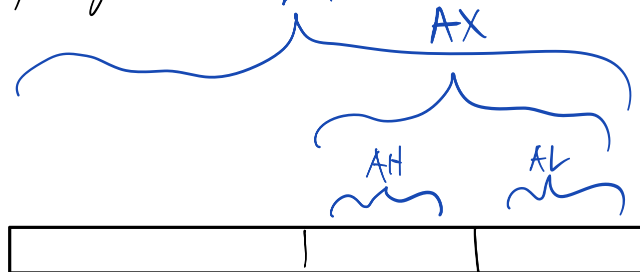
Pozor!

chci borrow 0

| CLC | STC |
|---|---|
| SBB | SCC |

# Rozdíl mezi 6205 a x86

- více registrů, segmentování:

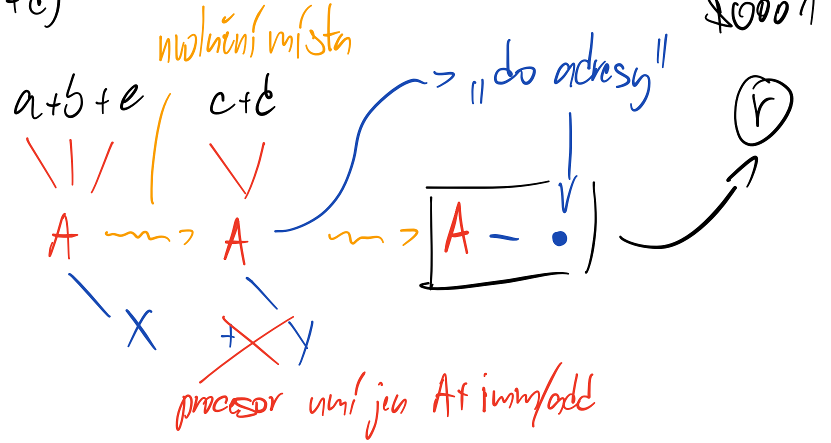



→ i když budu pracovat s „virtuální" AL, bude se dělat ; truncation.

## 6205

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

$Temp_1 = a+b+e$
$temp_2 = c+d$
$r = temp_1 - temp_2$




```
LDA $A000
CLC
ADC $B000
CLC
ADC $E000
TAX

LDA $C000
CLC
ADC $D000
STA $AAAA

TXA
SEC
SBC $AAAA
STA $0001
```

$AAAA je dočasné místo v paměti

Dokud to jde, nezapisovat do paměti, jelikož je to daleko pomalejší...

### Rychlost operací:

Mějme takt procesoru x GHz, jednotka 1 takt

Nejrychlejší: AND, OR ..., SHL, SHR
ADC $r_1, r_2$ / SBB ~1 takt

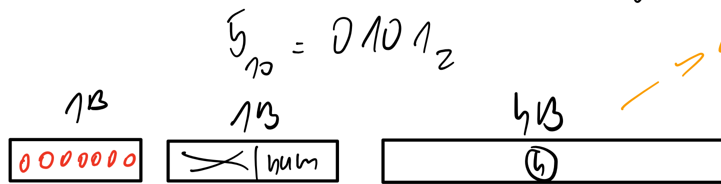
Pomalé: LOAD, STORE ~10 taktů
ADC $r_1$, [r LOAD] ~10 taktů

- zároveň pokud mám 8 bit procesor a sčítám 16 bit, tak násobím čas $\frac{16}{8}$ krát.

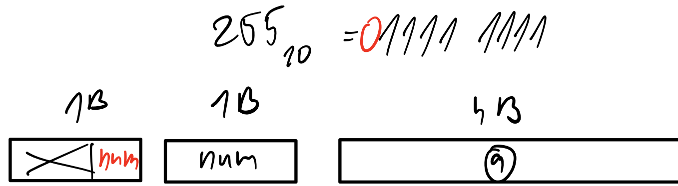
### Další pomalé:

| | | x86/x64 | ARM | Microcontroller | 6502 |
|---|---|---|---|---|---|
| MULT | ~10 takt | ✓ | ✓ | ✓/✗ | ✗ |
| DIV | ~100 takt | ✓ | ✓/✗ (SW) | ✗ (SW) | ✗ (SW) |

# 1. přednáška: Čísla v Pythonu




kolik je platných cifer

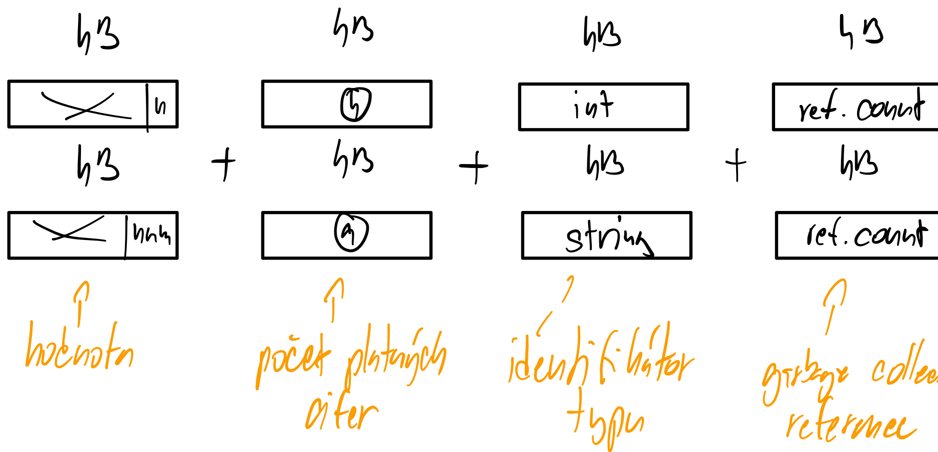



Tohle v reálu ukládají do 4B

V C/C#/Java/C++
- deklarují se proměnné

Pointer v Pythonu stojí 8B

nenormalizuje se tak často




80% overhead

Celkem = 24B

## Garbage Collector:

- reference counting, počítá, kolikrát je proměnná potřebná.
- když je count 0, tak to smaže...

Python 10x pomalejší jak C++.
Interpretace dalších 10x
Celkem 100x pomalejší.

Zároveň v Pythonu jsou všechny objekty immutable.
Když udělám: $x = x + 1$, vytvoří se nový objekt $x+1$.

## Proč je Python takový?

V C#: uint_8 → $x = 255_{10} = 1111\ 1111_2$

$x + 1 = \cancel{1}0000\ 0000_2 = 0_{10}$

-5 _ 256
nejčastější hodnoty, které se rovnou cachují předem

V takových jazycích to přetíká/podtíká. To se v pythonu nestane.

Vstupy/Výstupy umím správně zvolit velikostně, ale uprostřed výpočtu to může vystřelit až moc.

# Reprezentace „necelých" čísel:

leading zeros — trailing zeros

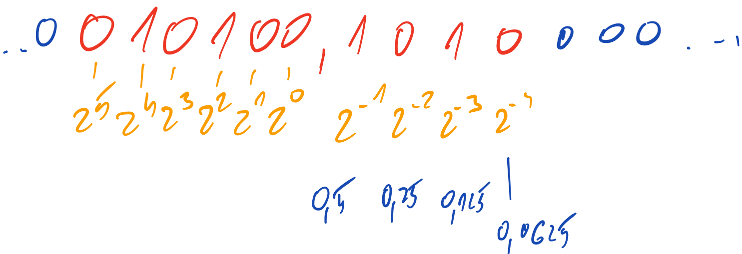
000 20,625 000

– v 10kové soustavě

## V 2-kové soustavě:

20,625

↗ tohle je fixed point




## Fixed point:

- celkový počet bitů
- rozdělení před + za bitů čárkou
  - např.: 5+3
- když přeteče, ztrácíme informaci

1,25 + 1,875 (8bit) (5+3)

00001,010 + 00001,111

jako unsigned:

00001010 -> 10
00001111 -> 15
00011001 -> 25

$00011{,}001_2$

$3{,}125_{10}$

Umí sčítat pomocí klasických operací procesoru.

Musí ale mít stejný formát!!! např.: (5+3)

## Reálné násobení / dělení:

Pokud násobím mocninou dvojky, platí zase SHL/SHR

– při dělení ale ztrácíme přesnost za desetinnou čárkou

0,5 (5+3) x 0,5 (5+3) = (10+6), musím to shiftnout doprava

Při dělení se ale ztrácí desetinná přesnost

| | x86/x64 | ARM | Microcontroller | 6502 |
|---|---|---|---|---|
| MULT → ~ 10 takt | ✓ | ✓ | ✓/✗ | ✗ |
| DIV → ~ 100 takt | ✓ | ✓/✗ (SW) | ✗ (SW) | ✗ (SW) |

## Násobení bez „násobení":

SHL 1
0 0 1 0 1 1 = 11
0 1 0 1 1 0 = 22
$2^1 = 2$

SHL 3
0 0 0 1 0 1 = 5
1 0 1 0 0 0 = 40
$2^3 = 8$

SHR 1
0 0 1 0 1 1 = 11
0 0 0 1 0 1 = 5
11//2 = 5

**Unsigned:**

$$x \text{ SHL } n = x \cdot 2^n$$

$$x \text{ SHR } n = x \text{ DIV } 2^n$$

**Signed:**

SHL 1
1 1 1 0 1 1 = −5
1 1 0 1 1 0 = −10

SHL 3
1 1 1 0 1 1 = −5
0 1 1 0 0 0 = 24 ✗

if small enough

$$x \text{ SHL } n = x \cdot 2^n$$

SHR 1
1 1 1 0 1 1 = −5
0 1 1 1 0 1 = 29 ✗

NO NO

$$x \text{ SHR } n \neq x \text{ DIV } 2^n$$

SAR – aritmetický posun doprava

SAR 1
1 1 1 0 1 1 = −5
1 1 1 1 0 1 = −3
ALMOST DIV!

SAR 1
1 1 1 0 1 0 = −6
1 1 1 1 0 1 = −3
DIV!

| Python | Java |
|---|---|
| << SHL | << SHL |
| >> SAR | >> SAR |
| | >>> SHR |

10. přednáška

# Fixed vs Float:

A: −101 00 101 00000 0000, 0 0 0 . . . . . . = 1,0100101 * $2^{15}$

B: + . . . . . 0 000 0, 0 000 0000 10100101 = 1,0100 101 * $2^{-9}$

takle bych spotřeboval hrozně bitů...

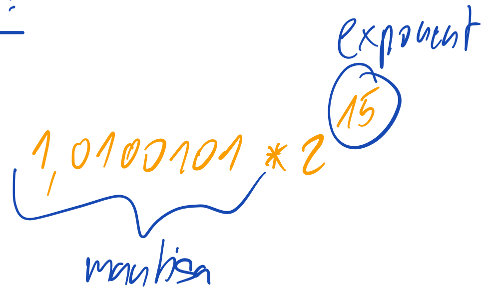
## Normalizovaný zápis:

- no leading zero
- před čárkou jen jedna cifra

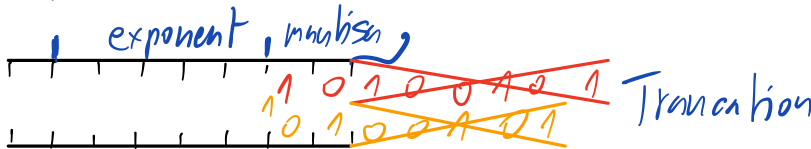
1, . . . .

## Floating point:

- exponent
- mantisa




Definujeme formát: 8 bit → 5b exp, 2b mantisa

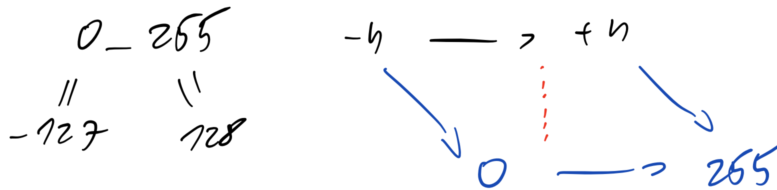



Truncation → ztrácíme jen „menší" hodnoty → není to přesné, ale řádově správné

Se skrytou mantisou → víme, že první cifra mantisy je 1. Můžu posunout a přesnost si jen pamatuju.

- exponent celé +/- číslo → nezmění se Two's Compliment, ale reprezentací posunem:

## Bias:

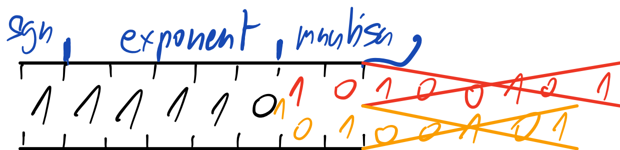



Definujeme, o kolik posouváme

Bias (−127) (o kolik posunu hodnotu vady)

Sgn = 0 = +, 1 = −

A:



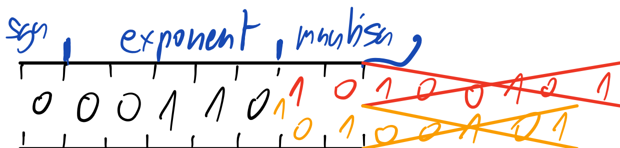

Máme 5 bit exponent, tedy 0 _ 31

Bias (−15)

$2^{15}$ 15 + 15 = $30_{10}$ = $11110_2$

$2^{-9}$ −9 + 15 = $6_{10}$ = $00110_2$

B:




- Nefunguje klasická hardware aritmetika.

Aritmetické operace se řeší SW --> je to ale ultra pomalé.

Některé procesory umí HW sčítání floatů. --> to je o něco rychlejší

x86 — HW; ARM — HW/SW; MicroController — SW; 6502 — SW — ne všechny procesory to mají

Potřeba mít správný formát float čísel (IEEE 754)
- single - 32bit (23+8+1) (mantisa, exponent)
- double - 64bit (52+11+1)

**Python:**
float = 64 bit

**Numpy:**
float32
float64

**C#:**
float = 32bit
double = 64 bit

Aritmetické operace FLOATů: (32bit)

A + B:

A = 1,01001010000000000000000000 — trailing zero do 23 bitů

B = 0,00000000000000000000000000|10100101 — Truncation (denormalizace)

23 bit Mantisa

Velké číslo + něco malého = Jen něco velkého. Čím větší přesnost, tím menší chyb. (u Pythonu to neřešíme)

Floaty se neporovnávají! (==), ale koukáme na $(a-b) > \varepsilon$ — jejich rozdíl je větší než...

- často je lepší převádět na jiné celé jednotky.
- existují tak malá čísla, která nejdou reprezentovat.
  <=> pak se některými hodnotami exponentu reprezentují ty záporné.

# IEEE 754

mantisa = 0 → $+\infty$ / $-\infty$

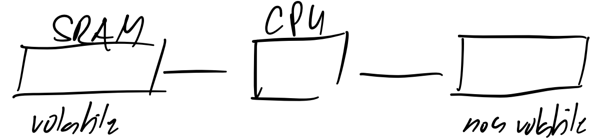
$x/0 \rightsquigarrow \pm\infty$

$\infty/\infty \rightsquigarrow$ NAN - Not A Number

*ROM -> non-volatile

**ROM:** (Read Only Memory) [R]
- read only memory
- jednorázově zapsáno, pak se jen čte

MC -> většinou jsou Harvardi




**PROM:** (Programmable ROM) [R/(W)]
- user programmable device (once)
- konkrétní diody se „spálí"

**EPROM:** (Erasable PROM) [R/W]
- „nekonečně" mnoho zápisů
- mazání UV zářením (vystavit na slunce)

**EEPROM:** (Electrically EPROM)
- „nekonečně" mnoho zápisů
  - po nějaké době už není zápis možný

(100 000 – 1 000 000) zápisů

- umí zapisovat na jednotlivé bity

**Flash**

(10 000 – 100 000) zápisů

- dlouhodobě data prohnijí
- za několik let se data ztrácí!!
- čtou/zapisují se jednotlivé bloky

| SRAM | DRAM | EEPROM/flash |
|---|---|---|
| 10 GB/s | 1–10 GB/s | 100 – 1000 MB/s |
| ~1 ns | ~ 10 ns | ~ 100 ns |
| 1B – 1kB – MB | 1 GB | 100 GB – 1 TB |

Flash reflektuje novější programy, které čtou většinou sekvenčně a nepřepisuje se tak často.

| | EEPROM | FLASH |
|---|---|---|
| sekvenční? | ✓ | ✓ |
| random read | ✓ | ✗ |
| random write | ✓ | ✗ |

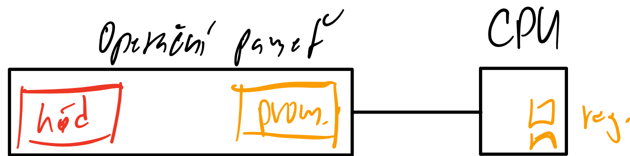
důsledek blokového přenosu (sekvenční, random read)

random write ✗: zapisuji vždy celý blok

## NVRAM – NonVolatileReadWriteMemory

- to jsou flashe

## Mějme PC: (Von Neumann)




## ADC:

- analog/digital convertor

## DAC:

- digital/analog convertor

## GPIO: (general purpose input/output)

- má direction register (jestli jsou linky R/W)
- input registr (stavy, které linky jsou vstupní)
- output regist (stavy, které linky jsou výstupní)

## Mass Storage Device:

- non volatile pro ukládání dat (zejména uživatele)
- zaznamenání stavů, které chci znát po restartu

## Buffer pro stav:

- většinou EEPROM kde se ukládají mezistavy programu (nastavení... atd...)

# Pevný disk / hard disk drive (HDD): (5400 – 7200 – 10 000 – 15 000)

stopa

1 2 3 4 5 6

jednotlivý sektor (512 B) (4096 B) advanced format

– vnější sektor je menší

– disk se roztočí a točí dokola

vnější sektory jsou daleko rychlejší

head (pohyb je „seek")

N S / S N

dipóly po 0/1

cylinder

stopa má obě strany

každá hlava má svoje id

ID := CHS = CylinderHeadSector

- sekvenční přístup ↗ ✓
- sekvenční přístup ↘ X
- náhodný přístup ✓/X
- random stopa **X** (musí se bradit)

**Výhoda:**

- kapacita (~10 TB)
- cena
- archivace (obsah se neztrácí)
- opravdu neomezený počet zápisů!!!!

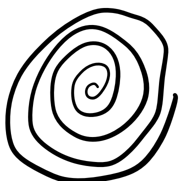
**Nevýhoda:**

- rychlost
- náchylnost na hlum

# CD / DVD / Blu-Ray (optický)

Buď se světlo laseru odrazí zpět nebo ne. (1) (0)
Proto se „vypalují" CD čka. – takové vypálení se má.

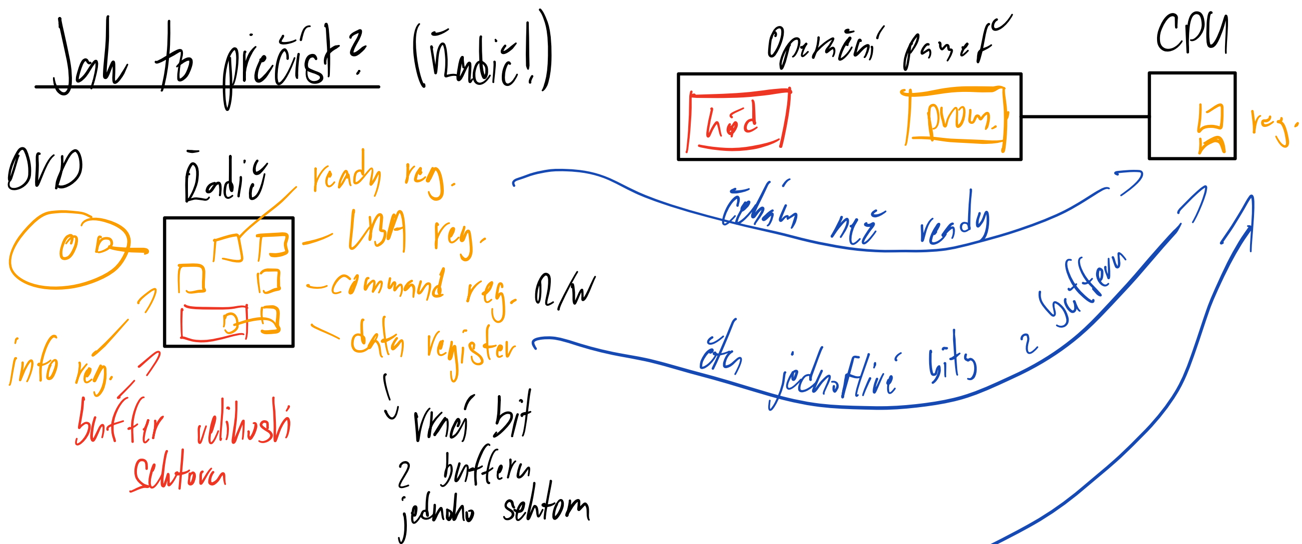
- Zase sektory, jen jedna (spirální stopa)

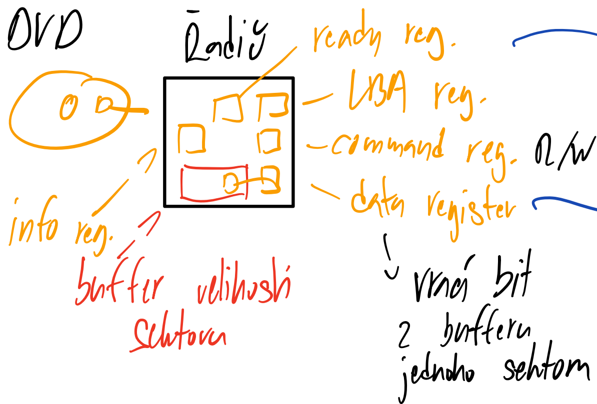


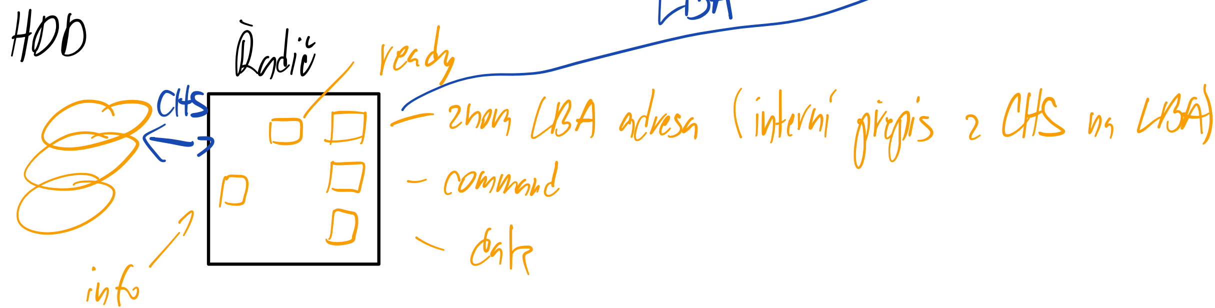
2048 B

- rozdělení na sektory (LBA – Linear Block Addressing)
- ultrapomalé random čtení, proto jen pro sekvence
-

# Jak to přečíst? (řadič!)



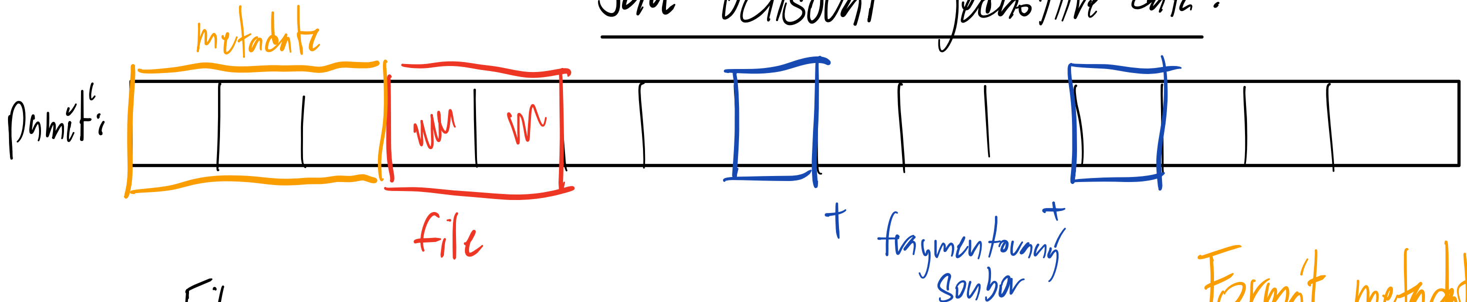




## SSD

- Flash s řadičem od HDD
  - přepisování na LBA adresy zejména
- sektor má bity: data jsou „v sektoru xy, na offsetu +2"

## Jak odlišovat jednotlivé data?




File
- metadata
  - seznam sektorů
  - délka souboru v B
  - jméno (cesta) /path/to/file.txt
  - seznam volných sektorů

adresář file

## Formát metadat:

Filesystém:
- potřeba naformátovat

(OS) OPEN/READ...

proč f.open()? f.close()? Metadata načítám jen jednou.

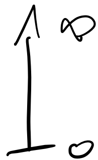
12. přednáška

# Jak uchovat fotku?

Budu mít nějaký rastr, do kterého budu uchovat intenzitu v tom místě.

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0,0 | 1,0 | 2,0 | .. | .. | .. |
| 1 | 0,1 | 1,1 | 2,1 | .. | .. | .. |
| 2 | | | | | | |
| 3 | | | | | | |
| 4 | | | | | | |
| 5 | 1 | | | | | |

- nejčastěji uchováváme řádky zleva doprava, pak jdeme do nižšího řádku.

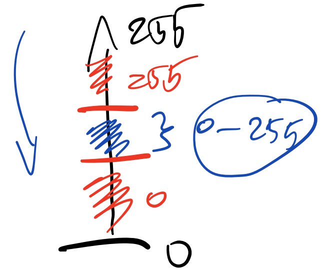



pixel

bits per pixel (BPP) / Bitdepth

definujeme tím hodnotu světla

rozsah se posouvá

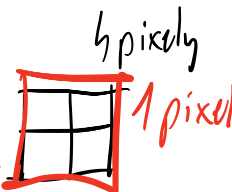



## 1bit depth (Monochrome)

- jeden pixel = bílá/černá

## Dithering

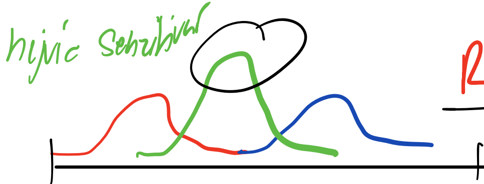
- Spojím několik pixelů dohromady, pak ztrácím rozlišení ale mám $2^4$ odstínů...




## 8bit depth:

- většinou je HDR omezený, záleží jak moc
- větší rozsah, vypadá lépe.

Chceme barvu – musíme rozlišit frekvenci těch fotonů:




RGB – stabilně uchováváme tyto tři hodnoty kanálů.

## 3bit RGB

- u každé složky 0/1 → na 4-bit umím zapsat intenzitu kanálu.

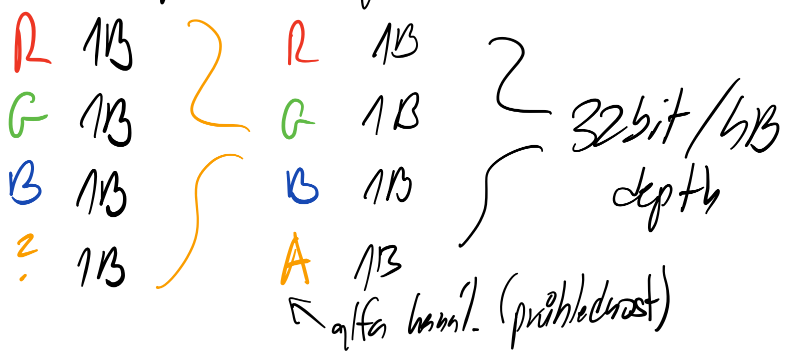
## 2B RGB (15bit depth)

RGB – 15 bitů (5+5+5)

## 16 bit: (5+6+5):

- přidali jsme zelenou protože jsme na ni citliví

## 24bit depth (8bit per channel) (TrueColor):



|  | R | G | B |
|---|---|---|---|
| White: | FF | FF | FF |
| Black: | 00 | 00 | 00 |

Jeden pixel

| A | R | G | B |
|---|---|---|---|
| B | B | B | B |

--> potřeba definovat MSB/LSB

## Jak to uhládat?

**Metadata**
- počet kanálů
- bitová hloubka
- výška/šířka v pixelech

**Data**

--> Tohle je soubor, co má i vlastní systémový metadata

## Textová data: (String)

- posloupnost znaků (písmena, číslice, spec. znaky, white-space)
  <--> posloupnost kódů
- dělám to pro zpětnou rasterizaci
- v datech není dáno kódování --> musí se definovat předem!

### Kódování:
- jednoznačné, oboustranné kódování
- mapa mezi kódy, délka kódu (proměnná / pevná)

### ASCII - 7bit:

0 _ 127 --> 0 ≠ 0

A, B, _ Z, a, b _ z, 0, 1 _ 9
X X+1 X X+1 Z Z+1

7 bitů uhládáme do LSb bitů

128 _ 255 neobsazeno

↘ Code Page (lokální rozšíření)

CS --> ISO 8859-2 (ISO) Latin2

MSDOS 852 (DOS) Latin2

WIN 1250

### UNICODE:
- všechny možné znaky
- UTF-32 - uhládám 32bit nosný číslo
  little/big endian
- vždy se uhládá v pořadí, jak se slovo čte. Takže pokud kombinuji, střídám to.

13. přednáška

## Memory controller:

- zajišťuje refresh DRAMu
- můžeme zapojit více paměťových modulů do jedné datové sběrnice
  - → Modul$_1$ bude mít adresu ∅
  - Modul$_2$ bude mít adresu X
  - } pak rozdělí do jednotlivých modulů a procesor nic neví.
- zároveň můžu na procesor napojit novější moduly s jiným protokolem, tedy to memory controller přeloží.

CPU vždy volá „chci byte z adresy $xx$_0$xx$_1$ ..."

## System BUS: (PCIe)

- dokážu zapojit vše (operační paměť, zařízení...) na jeden BUS
- multidrop
- full-duplex
  1) Adresa zařízení
  2) Paměťové packety
     - memory write (MWr, STORE)
     - memory read (MRd, LOAD)
     - } packety na konkrétní byty, mířící pro Memory controller
- multi master (Memory controller si sám říká, kdy pošle data)

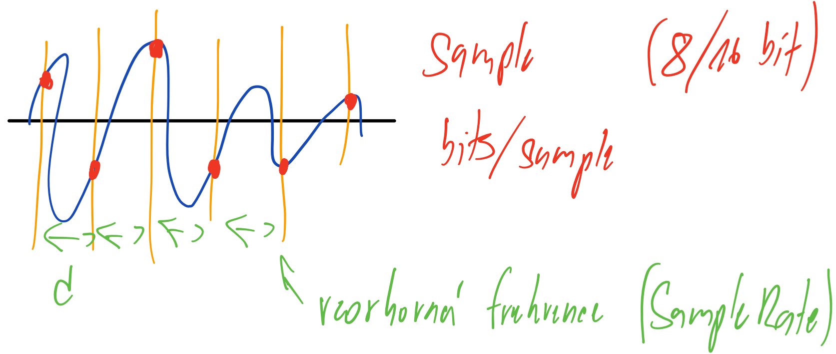
## Memory Map IO:

- registry zařízení jsou namapované do adresového prostoru operační paměti
- výhoda je, že komunikace po celém BUSu je tak unifikovaná, ať už jde o jakékoliv zařízení

## HCI: (HostControllerInterface)

- většinou to je vůbec procesor a definuje, jak mluvit s ním

## Zvuková karta:




## Ideální DAC

- u zvuku je potřeba přesnost!!
- potřebuju ale buffer (Soundbuffer)
  takže bude precizní časování
  a ve správném čase čtení/přehrání

## HW akcelerace (CPU offloading):

- odložení nějakých operací do HW místo SW, aby se SW odlehčilo.
- většinou ve zvukovkách mají DSP (digital signal processor)
  - tam se provádí HW accel.

## GPU:

- má nějakou bitmapu, která ukazuje, co se má zobrazit a ta karta to pak ve frekvenci (50 Hz) posílá do monitoru.

## Polling

- blokování CPU time „čekáním na"

## Bootloader:

- pamatuje si ve kterých sektorech jsou užitečné „SW"
- nahrává kernel OS do adresního prostoru
  - a pak OS dělá tu abstrakci nad HW

## CPU startup:

1) ID -> x86: $FFFFFFF0
   - zbývá 16 bitů, kde uložím JUMP na reálný začátek firmwaru

2) Nalezení užitečného „SW"
   - existuje i OptionROM, kde je ten zajímavý SW

občas třeba nastaví zařízení a pak pošle průběh zpět do firmwaru