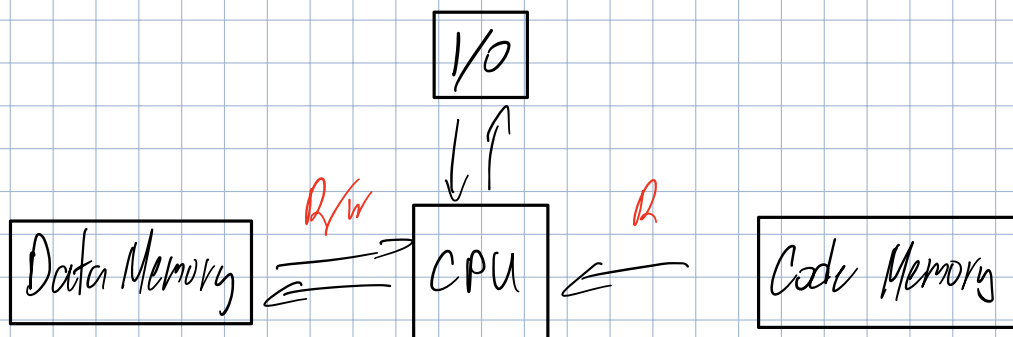


1. přednáška

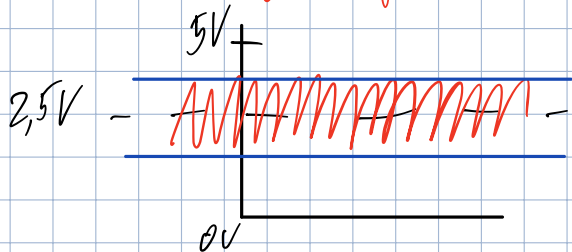
Harvardská arch. / Oddělení datové a kódové paměti



Reprezentace čísel:

- nečíslové, přirozené číslo

Princip analogového přenosu:

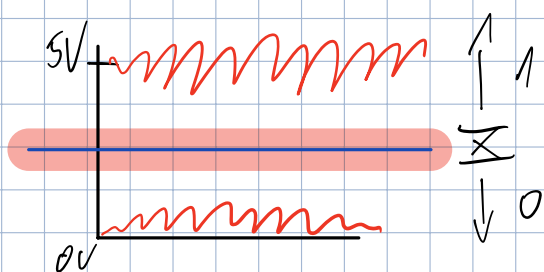


→ Analogový přenos

✓ hranice, do kterých se musí vejít, aby bylo to yhodnotili jako hledanou situaci.

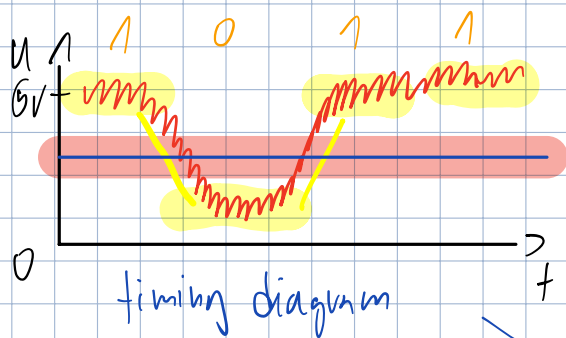
Princip digitálního přenosu:

0V/5V, má jen jeden bit



→ hranice meziho napětí.

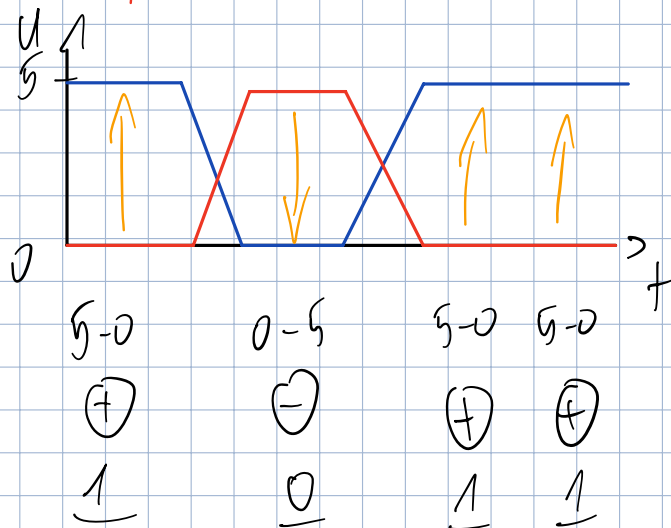
Posilání dalších čísel: (1 0 1 1)



digitální sériový přenos:

Pro měření napětí potřebuji i referenční bod... GND

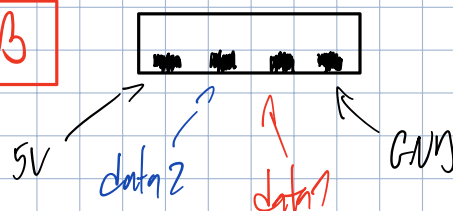
data 1 / data 2



Diferenciální sériový přenos:

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

USB



GND: napsání...

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

$$256 = 2^8$$

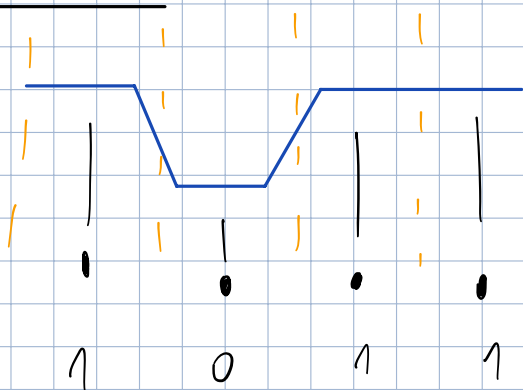
$$1024 = 2^{10}$$

$$4096 = 2^{12}$$

$$65536 = 2^{16}$$

Přenosová rychlost: 1 baud
 = (symbols per second)

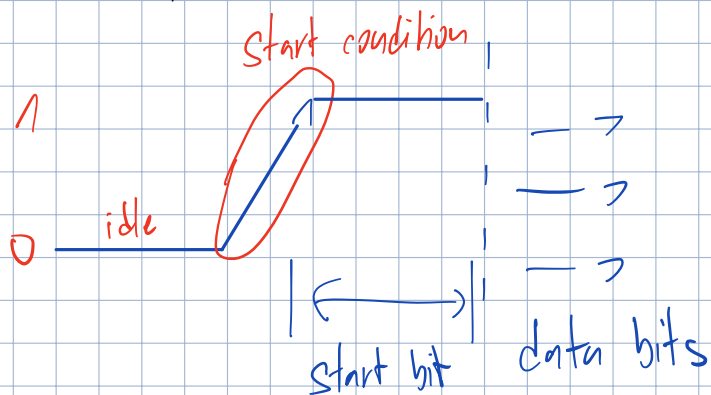
2. přednáška



Ta správná hodnota se nachází většinou uprostřed.

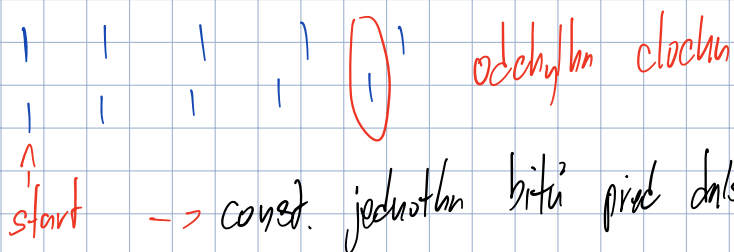
Potřebujeme ale nějaký CLOCK
a IDLE stav (floating / Hi-Z)

3-state logic: 1/0/F



: rising edge, pak se chvilí čeká...

Pokud clock na zařízeních není stejný rychle, brzo se to rozejde

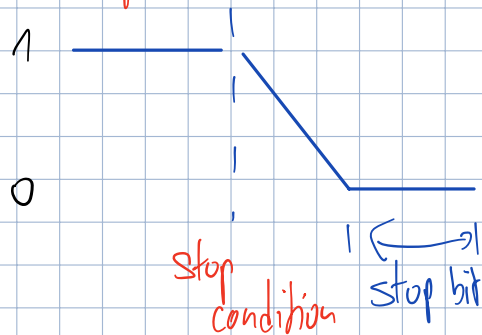


odchyln clock

start -> konst. jednotku bitů před dalším clock. sync: 8b, 1b

Stop condition:

- vysoká x bitů, pak hned skáče a jde do nuly.



Need 1000 baud: (overhead?)

RS-232

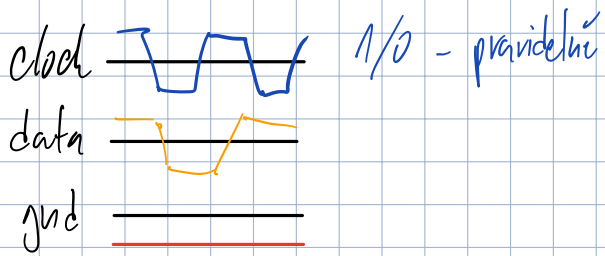
-> 1 stop bit

-> 1 start bit

-> 8 data bits

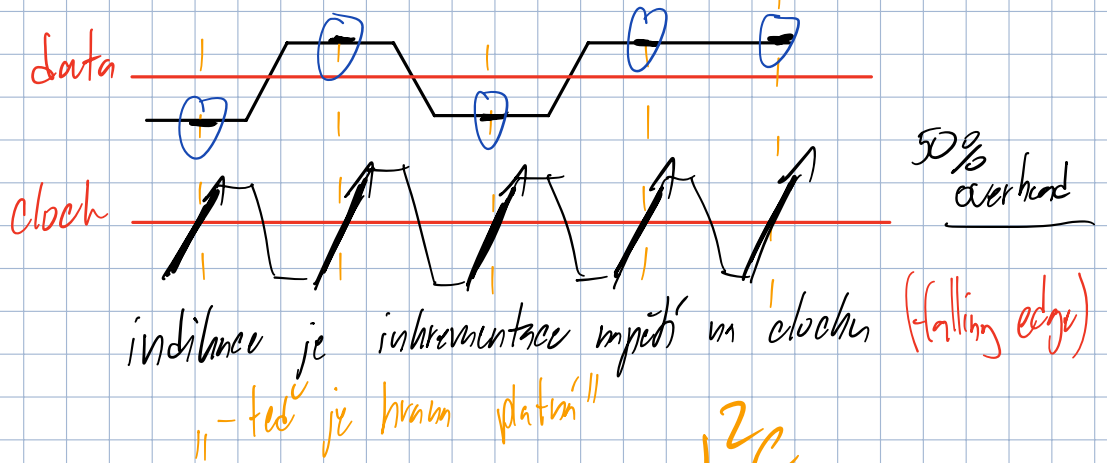
20% overhead

800 bits/sec

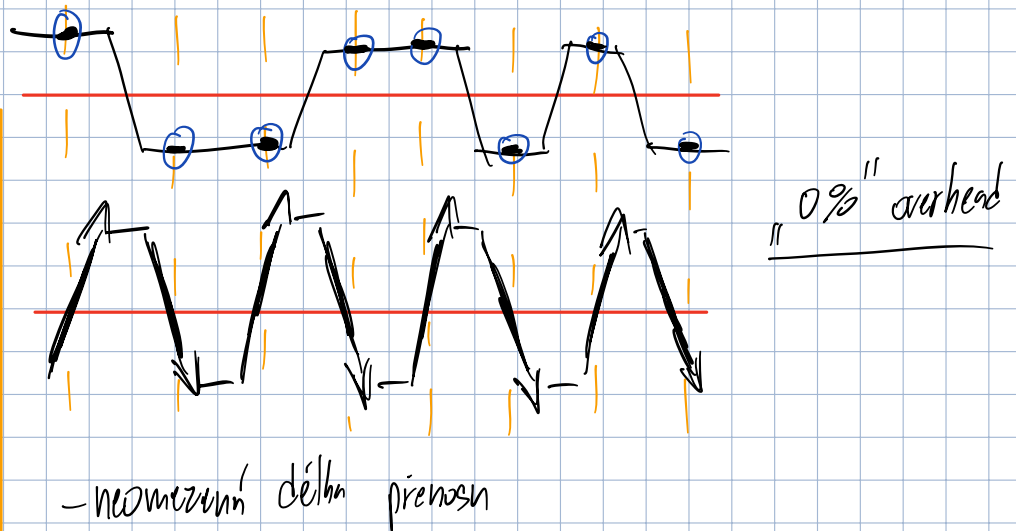


Definujeme, kdo vysílá clock, kdo poslouchá.
Rychlost je dvojnásobek rychlosti clocku.

clock $\rightarrow \frac{Hz}{\text{cycle/takt}}$



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



Problém:

odlišná délka

CLOCK a DATA

vodiče odchylí

přijem CLOCKU a DAT.

Chceme přenést 8 bitů, pošleme 10 bitů:

00000000
11111111

2^8
256

0000000000
1111111111

2^{10}
1024

Nemůžeme poslat 0000 0000 ani 1111 1111, protože tím narušíme clock. Místo toho to namapujeme čísly z 10 bitů...

- pravidelně se střídají 1/0.

- prepisovací pravidla jsou předem daná.

CLOCK RECOVERY $\rightarrow$ 20% OVERHEAD

USB

Simplex: $\rightarrow$ umí jen jeden

Duplex: $\rightarrow / \leftarrow$ umí oba směry, vždy jen jeden

Half: Současně vysílá jen jeden 1°

Full: Současně vysílají oba směry RS-232

RX: $\rightarrow$:RX
TX: $\leftarrow$:TX

Out of Band:

1 - True $\rightarrow$

0 - False $\rightarrow$

Můžou definovat 0/1 nějaký stav

Inverzní logika: 1 - False

0 - True

1 - True

např.: když „dočkáš prvně“

dám tam 0.
Signál je
mi to jedno.

Umírání? 1 - Ne
0 - Ano

Komunikační protokol:

definuje charakteristika dat, jejich formát, MSB/LSB first...

že samotných dat to nikdy nevyhazujeme.

Paket:

délka $\leftarrow$ první proměnná

1 datový set def. protokolem

Řadič:

- např.: řadič RS-232 $\rightarrow$ pro komunikaci mezi CPU a zařízeními. Např.: RS-232 máš dva 7 bit, CPU má 8 bit.

- řadič má registry: PC si dělá do registru. Např.: registr poslední polohy a registr ON: byl/něbyl změna.

- zároveň má další registry, ve kterých jsou def. vlastnosti spojení (MSB/LSB first...)

tedy: datový, stavový a konfigurační registr

3. přednáška

U RS-232 se napájení zařízení řešilo tvrdým nastavením $T_{vcc} = +5V$ na OutOfBand vodičích.

Blukující funkce:

`read(n)`: $n = 1024$ bit $\rightarrow$ dokonce nepřeteče 1024 bitů, nepůjde dál

Neblokující alternativa:

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

Semi blokovácí:

`read(n) + timeout(200)` $\rightarrow$ bastard...

BTW: `input()` blokovácí

V Pythonu existuje lib `keyboard`, kdy existuje „if key is pressed“ ... neblokující

V protokolu máme definování, co který bit znamená. Pak čtu hodnoty, dívám se na jejich bity a vyhodnocuji. (Např. RS-232 posílá 8 bit. Byte)

Musím zároveň porozumět první byte celé jedné sady, a když něcočet jednotlivé sady špatně.

Reprezentace binárního čísla:

16-hod. soustava: (hexadecimalní / hex / šestnáctková)

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

$$23_{16} / 0 \times 23 / 023 / 23h = 23_{16} = 2 \cdot 16 + 3 \cdot 1 = 32 + 3 = 35$$

$\uparrow \quad \uparrow$
 $16^1 \quad 16^0$

$$\underbrace{0000}_{\text{leading zeros}} 123_{16} = 256 + 32 + 3 = 291$$

$\uparrow \quad \uparrow \quad \uparrow$
 $16^2 \quad 16^1 \quad 16^0$

Proč hexadecimal?

Dobré praktické vlastnosti:

- 1 hex číslo zabírá 4 bity

- 1B jsou tedy 2 hex čísla.

$$n \text{ bit} \rightarrow 0 \text{ --- } 2^n - 1$$

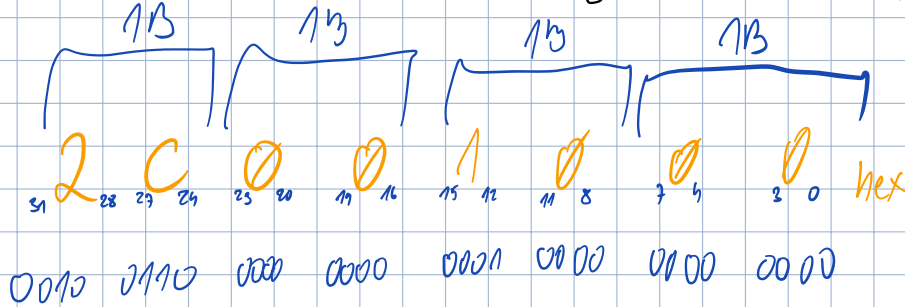
$$3 \text{ bit} \rightarrow 0 \text{ --- } 2^3 - 1$$

$$4 \text{ bit} \rightarrow 0 \text{ --- } 2^4 - 1 \quad (\text{stejně 4 bity na uložení 1 hex číslice})$$

$$8 \text{ bit} \rightarrow 0 \text{ --- } 2^8 - 1$$

$$8 \text{ bit} = 1B \rightarrow 2 \times \text{hex \#} = 1B$$

$$4B = 32b$$



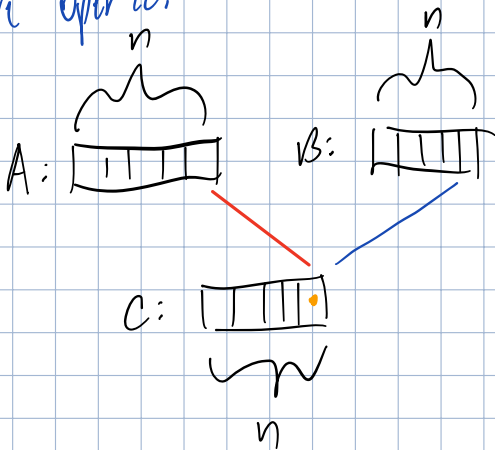
A	B	C	D	E	F
10	11	12	13	14	15

- velmi jednoduchá nahlédnutí, který bit je High

- Ať už je reprezentace jakákoliv, v PC je vše zapisováno v bináři.

Programově zjišťovat nastavení bitů?

Bitové operace:



- vždy aktivní jen ten jeden konkrétní bit.

Bitové operace:

1 OR

hodnota
příloha

A	1	0	1	0
B	1	1	0	0
C	1	1	1	0

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

2 AND (přepiš jen konkrétních hodnot)

hodnota
příloha

A	1	0	1	0
B	1	1	0	0
C	1	0	0	0

→ 1: přepiš hodnotu z 1
→ 0: ignoruj, napiš 0

~ NOT (bit flip)

A	1	0	1	0
C	0	1	0	1

XOR (selectiv flip)

hodnota
příloha

A	1	0	1	0
B	1	1	0	0
C	0	1	1	0

→ 1: otoč bit
→ 0: zachovej

Hledáme "L-složku" v čísle:

B ₇	B ₆	B ₅	B ₄	B ₃	B ₂	B ₁	B ₀
?	?	L	?	?	?	?	?
0	0	1	0	0	0	0	0
0	0	L	0	0	0	0	0

bitová maska, která mi ukáže jen konkrétní bit

= 0010 0000 (0x20)

pokud =, L je HIGH

pokud ≠, L je LOW

B₇ AND 0x20 = 0x20

4. přednáška

Bitový posun (bitwise shift)

SHL / Shift Left

$a, b \in n\text{-bitů}$

$a \ll n \rightsquigarrow b$

vstup počet bitů posunu výstup

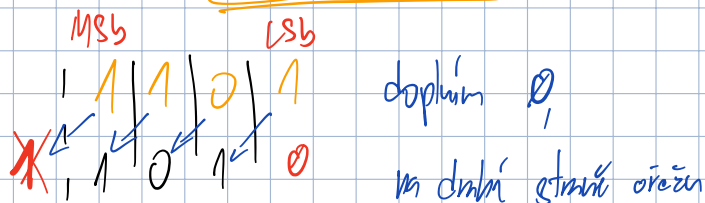
SHR / Shift Right

$a \gg n \rightsquigarrow b$

ROL / Rotate Left

ROR / Rotate Right

SHL $\rightarrow$ posun k MSb



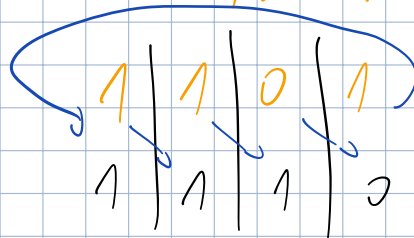
SHR $\rightarrow$ posun k LSB



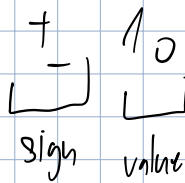
ROL $\rightarrow$ Jako SHL, jen doplňme



ROR $\rightarrow$ Jako SHR, jen doplňme



Reprezentace záporných hodnot:



Máme:

(8-bit) Unsigned:

$$5 = 0000101$$

(8-bit) Signed:

$$\text{MSb} = 1 \Rightarrow "-", \text{MSb} = 0 \Rightarrow "+"$$

n-1 value bits
1 sign bit

$$-5 = 1000101 \xrightarrow{\text{unsigned}} = 133$$

$$-6 = 1000110 \xrightarrow{\text{unsigned}} = 134$$

rozsah: -128_127

- nefungují základní operace
v procesorech

$$-5 < -6 \quad \boxed{5}$$

(8-bit) One's complement:

$$0 = -0 \quad \boxed{4} - \text{existují dvě nuly}$$

$$" + " \xrightarrow{\text{unsigned}} (a) \quad 5 = 0000101$$

$$" - " \xrightarrow{\text{NOT}(\text{abs}(a))} -5 = 11111010$$

$$-6 = 11111001$$

$$010 = 2$$

$$001 = 1$$

$$2 > 1 \Rightarrow -5 > -6$$

$$0 = -0 \quad \boxed{4}$$

- zase existují dvě nuly

$$-5 + 1 = 11111011$$

$$0000100 = 4$$

repr. "-4" v One's comp.

(8-bit) Two's Complement:

$$" + " \xrightarrow{\text{unsigned}} (a) \quad 5 = 0000101$$

$$" - " \xrightarrow{(\text{NOT}(\text{abs}(a))) + 1} -5 = 11111011$$

$$-6 = 11111010$$

0 je jen jedna

rozsah: -128_127

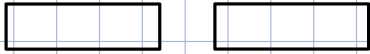
Všechny binární operace fungují, jenom nefunguje
porovnání záporného čísla s kladným

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

Obecně se používá Two's Complement.

Dobře je protokolem potřeba definován, jakým způsobem se zapisuje znaménko a v jakém bitovém rozsahu.

Reprezentace čísla v Pythonu:

value # platných bitů
 → pythonu má neomezeně velký 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
 -128 = 1000 0000
 0000 = 0

- pokud je to blízko nuly, funguje...
 0 ≠ -128

Zero extension:

4 bit 0101 = 5
 ↓
 8 bit 0000 0101 = 5

4 bit 1110 = -2
 ↓
 8 bit 0000 1110 = 14

Sign extension:

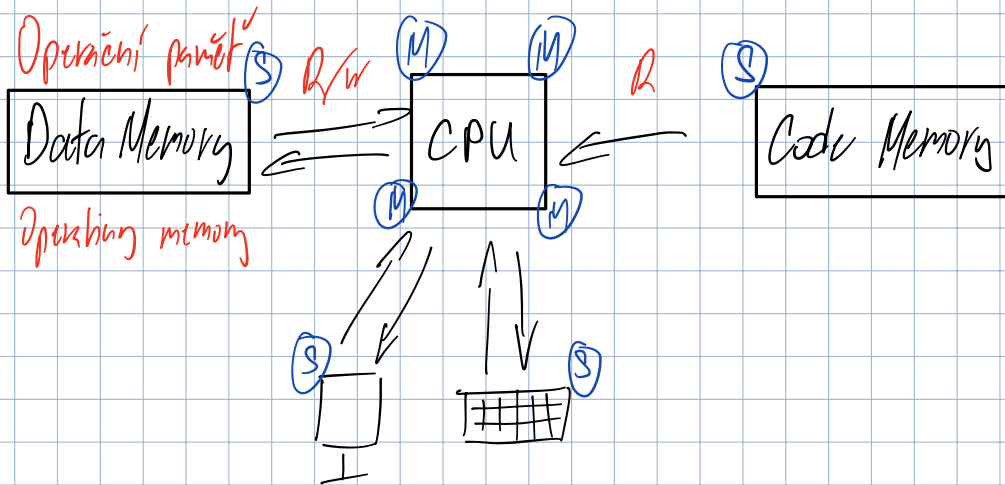
0101 = 5
 0000 0101 = 5

4 bit 1110 = -2
 ↓
 8 bit 1111 1110 = -2

4 bit 1111 = 15
 ↓
 8 bit 1111 1111 = 255

5. přednáška

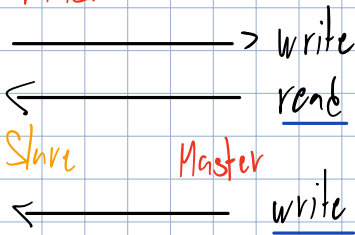
typický procesor je výhradně Master



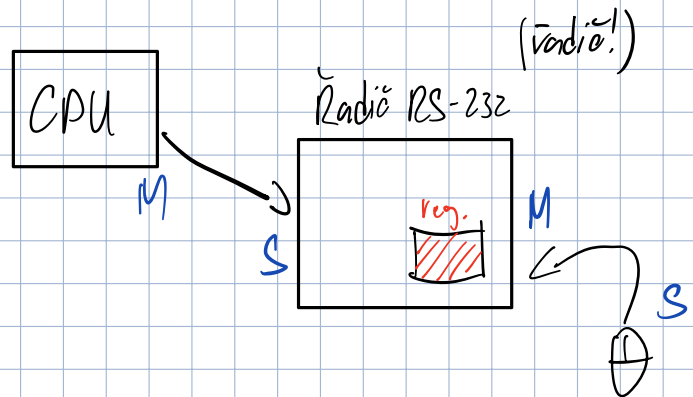
Master vs. Slave:

Master řídí komunikaci, Slave dává informace / reaguje

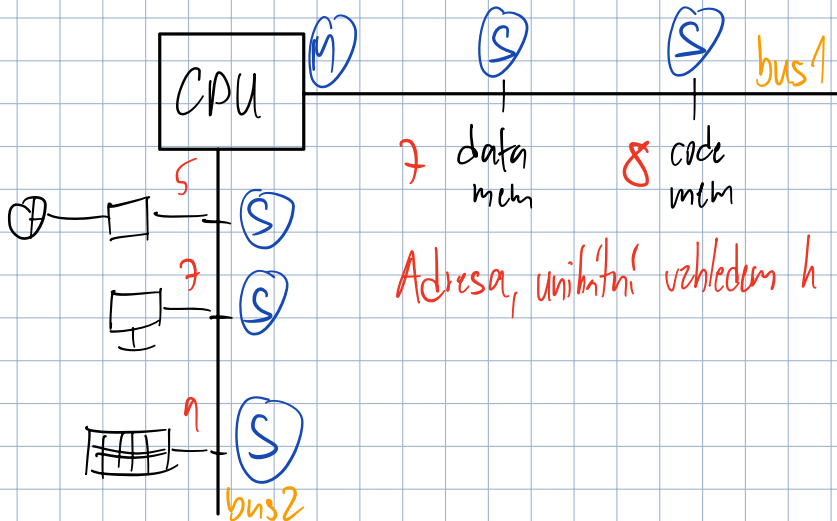
Master x Slave



Sak ale funguje Output only zařízení?



Multidrop / BUS / sběrnice

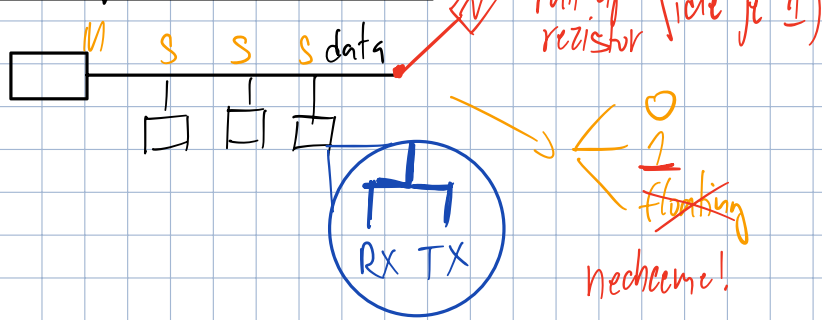


Adresa, unikátní vzhledem k sběrnici

Musíme znát:

Rozsah adres = adresový prostor
- maximální # zařízení

Nechť je bus half-duplex:



nicdo $\rightarrow 1$
 nechci nic $\rightarrow$ 1 odpojena $\rightarrow 1$
 chci 1 $\rightarrow$ 1 odpojena $\rightarrow 1$
 chci 0 $\rightarrow$ připojím 0V $\rightarrow 0$

Odpor na staru je delší menší než na pull-upu, takže po připojení ten proud začne téct tím menším odporem a v obvodu více napětí blíže nule.

D ₁	D ₂
x	x
x	1
x	0
1	1
1	0
0	1
0	0

Výsledek (bit, AND)

1
1
0
1
0
0
0

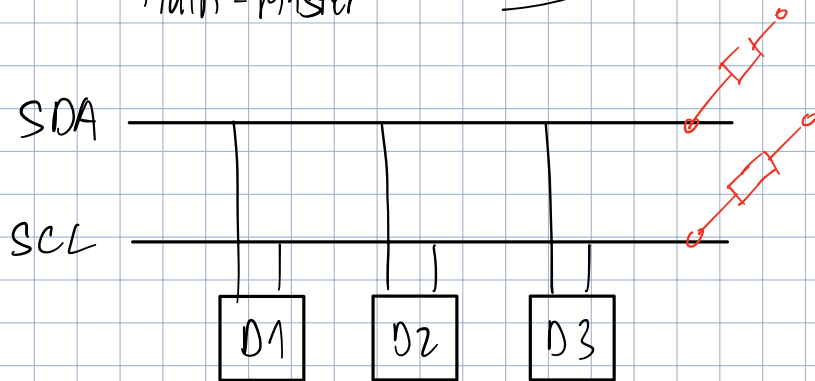
výsledek je deterministický (1/0)

I²C (inter-integrated circuit)

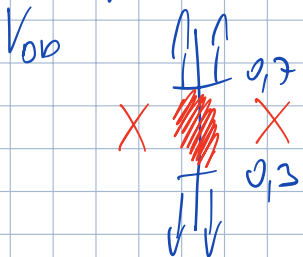
- half-duplex SDA (s pull-up)
 hodiny SCL (s pull-up)

USB je single master

- Multi-master

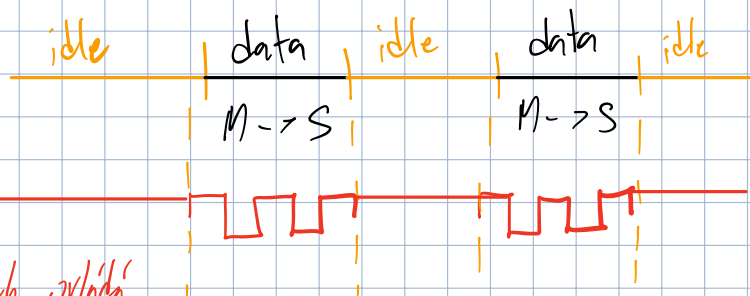


- první definování úrovně logických napětí mezi



- definuje se logika SCL

- když SCL high, data jsou platná



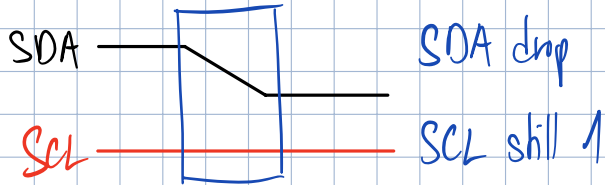
SCL

clock ovládá master.

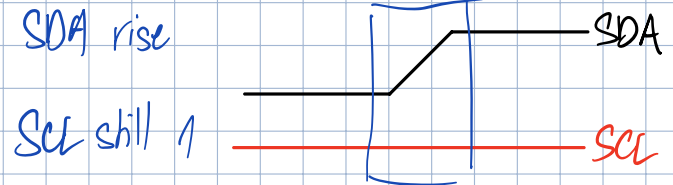
Když není master, clock je 1.

Start / Stop condition?

Start condition

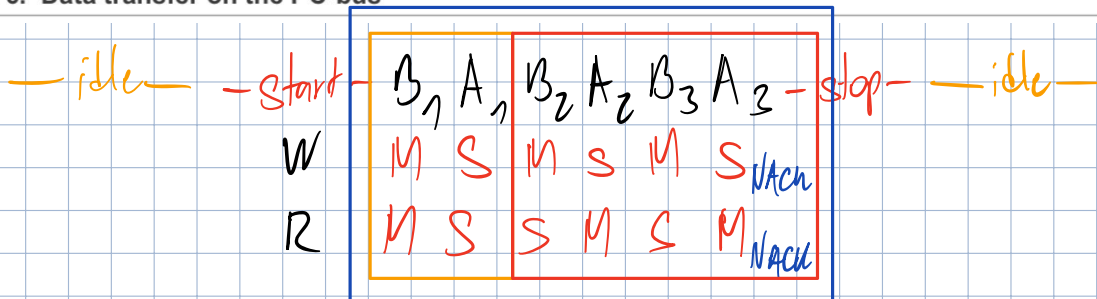
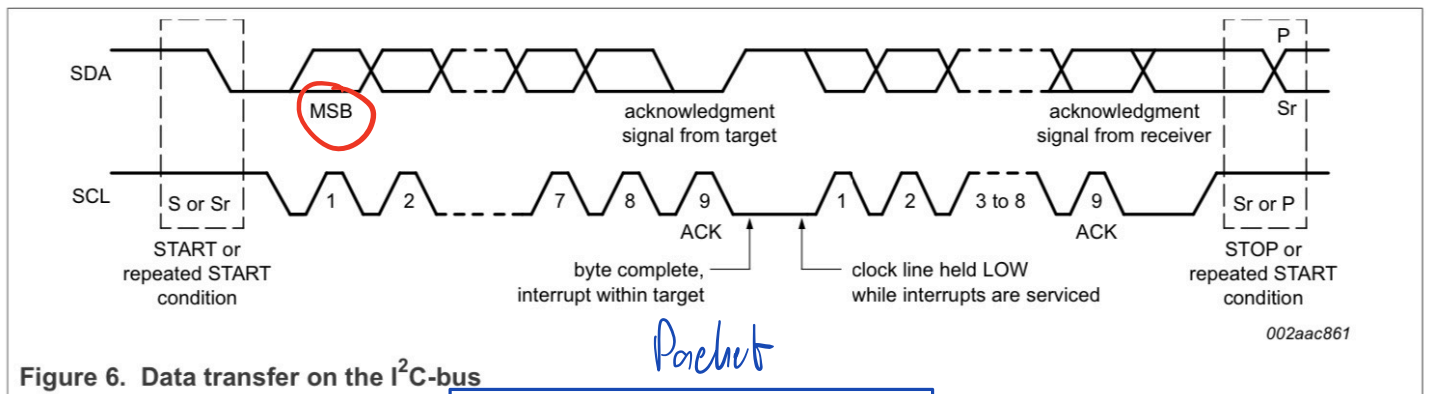


Stop condition:



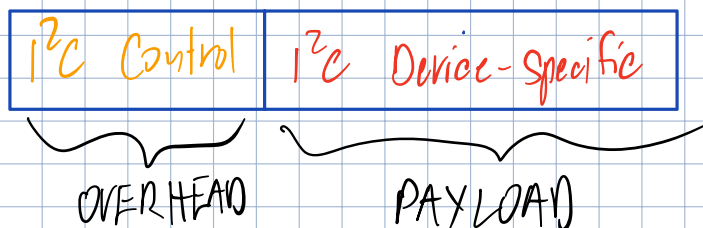
Podoba prenosu:

9 bit byty = 8 bit data + 1 bit ACK (1)



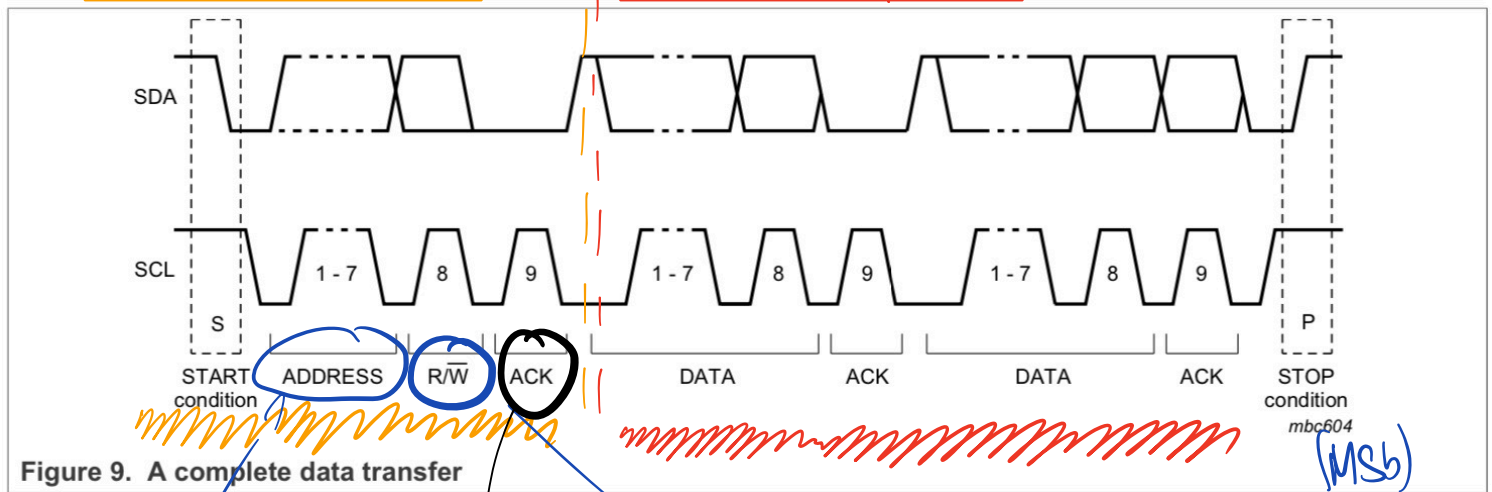
prvý byte má pri R/W vždy stejnou formu (adresu...)

Packet:



I²C Control byte:

I²C Device specific:



posílá se nejprve adresa
kde posílám pítinu

Díky pull-up datové linky
bude mít „nikdo tu není“
odpověď bez připojeného zařízení
automaticky. Zařízení poleže by byl slave
zapřetrhnut, posle NACK a master posle STOP condition.

R/W

R

1 = RW

0 = NE

W

1 = NE

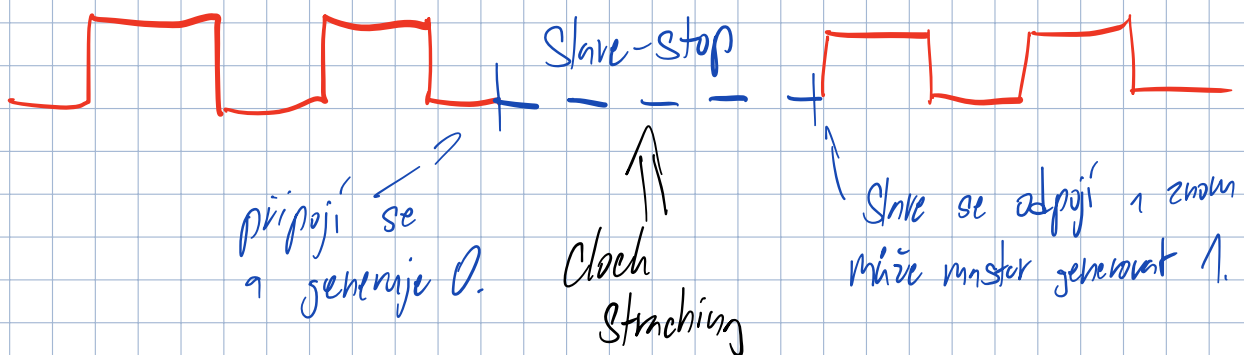
0 = AW0

Přenosová rychlost:

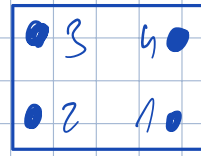
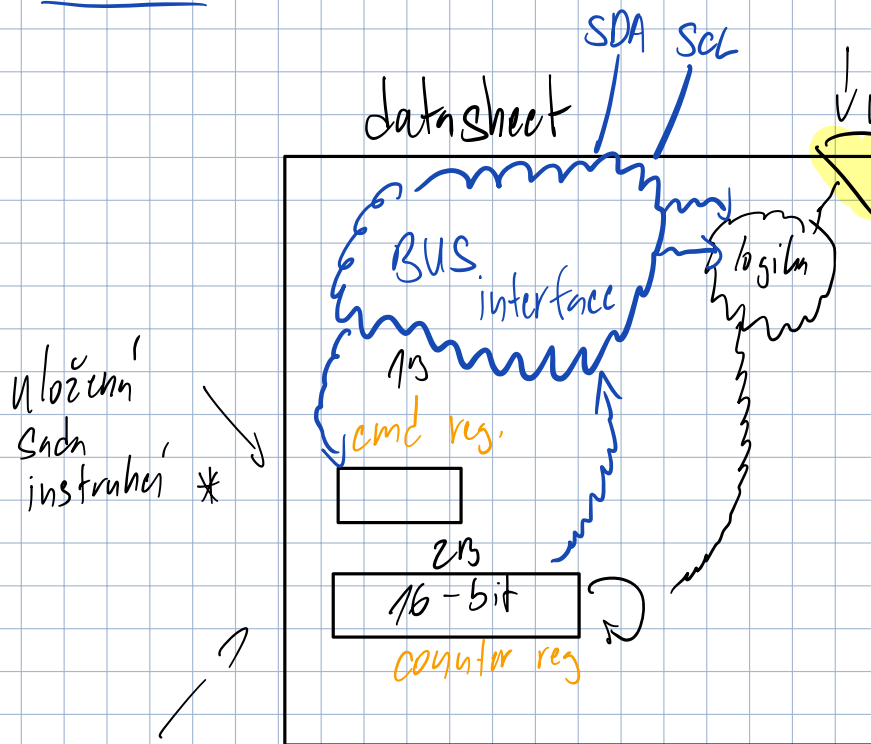
100 kHz – 5 MHz → relativně pomalá frekvence

Poleh slave nestihá: (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 tahat posle až slave povolí.



Příklad: (ALS - Ambient Light Sensor)



- 1) Vcc
- 2) GND
- 3) SDA
- 4) SCL

cmd → command register

Bus interface:

hard-wired (zadrátování):

- address: 0x29

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

V rámci jedné IC musí být všechny adresy jednoznačně určeny

číslo: 0 - 32000

* logika dostane instrukci a podle cmd reg. udělá činnost

Registry:

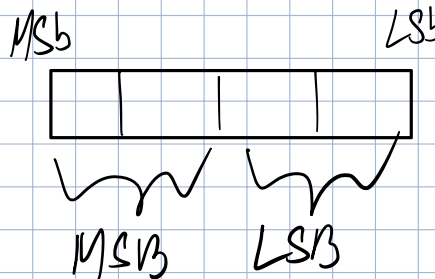
W/O - write only

R/O - read only

R/W - read write

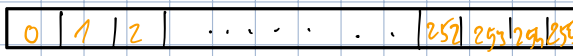
V BUS se obecně musí definovat cíl zápisu, zdroj čtení

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



LSB/MSB first.

6. přednáška

256 B  -> tedy může být 8bit, i 16bit adresový prostor
Adresy bytu - unsigned => Adresový prostor

200 B -> opět adresový prostor musí být 8bitový, náhodou adresy nevyžijí.

128 B -> 7bit. prostor -> ale i 8bit. prostor...

Je výhodnější, aby byl adresový prostor rovnou velký, aby program dokázal adresovat velké paměti.

Zkratky velikosti:

1 kB = 1024 bytů

"1000 B (10 bit -> 1024 adres) -> 16 bit. prostor -> 64 kB

1 MB = 1024 kB

(20 bit) -> 24 bit. prostor -> 16 MB

1 GB = 1024 MB

(30 bit) -> 32 bit. prostor -> 4 GB

1 GiB

1 MiB

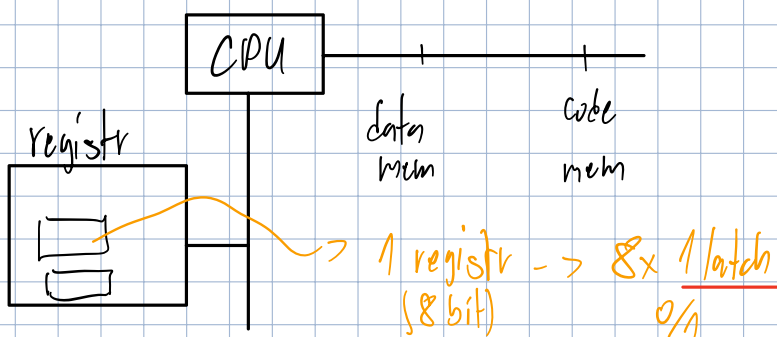
1 KiB

- jelikož adresování má 1024 adres
takto pro je nevhodné...

Udělá celkem dobrý adresant:

Prodejci disků:

1 kB = 1000 B



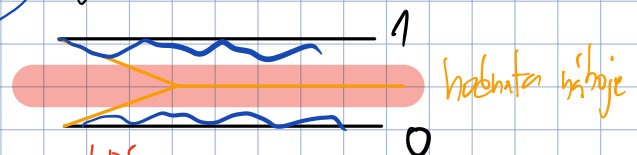
static

~~Starší RAM~~
SRAM = (Random Access Memory)

- volný výběr bytu bez sekv. čtení
- uniformní rychlost přístupu

DRAM:

dynamic - jeden tranzistor + kondenzátor



kvůli refresh jsou pomalejší

SRAM

DRAM

schvácení ↗

schvácení ↘

random

schvácení ↗

schvácení ↘

random

volatilní

proto není vhodná pro kód mčm.

ultra-volatilní

zapomene vše za tak 1ms i v napájení

1B - kB - 1MB kapacita 1GB - 10GB

10 - 100 GB/s přenosová rychlost 1 - 10 GB/s

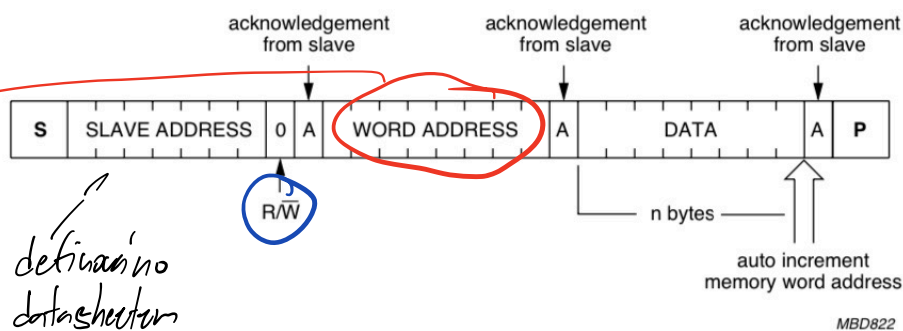
1ns random access

10ns

Příklad I²C SRAM paměti:

- většina má Analogově nastavitelnou adresu (HIGH/LOW). v datasheet definováno

Jak vypadá zápis:



Word - jednotka přenosu

- 8bit slovo -> přenos 8bitů

- s jak moc velkým číslem umím psát

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

Paměť:

pohled CPU: 0 1 2 3 4 5 6 7 8 9 10 11

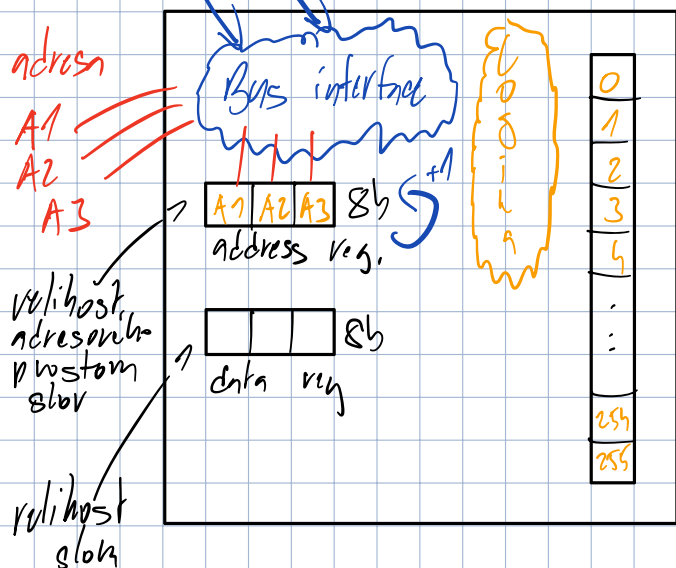
slovo 16-bit 0 1 2 3 4 5

slovo 32-bit 0 1 2

slovo 0 = byte 0 a 1

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

SCL SDA - vždychy se vrací celý slovo - chci přičíst byte 0 v 32bit slově



adresa není uložena, je to jen náš popis

remín slovo 0
byte 0, 1, 2, 3
a tam hledám...

f_c 100 kHz $\rightarrow$ 100 000 b/s

tolik byte přemísíme

$$X / (3 \cdot 4)$$

adresa, word adresa, data

12500 B/s

RANDOM

3708 B/s

2xpis

za nesynchronního přístupu

Burst přenos:

auto-increment slova $\rightarrow$ hned posílá

další data dalšího slova, protože
řešíme chceme více dat synchronně?

$$(X - (2 \cdot 4)) / 4$$

11 109 B/s

2xpis

BURST

2xpis:

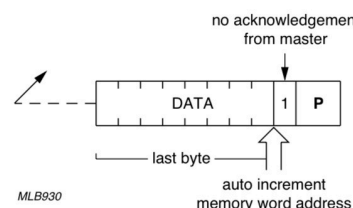
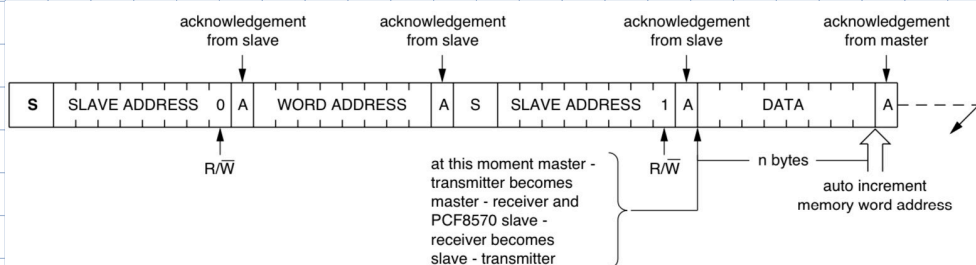
- posílá data do buňky 2 registru 1 f_c word 2x3

speed(write) > speed(read)

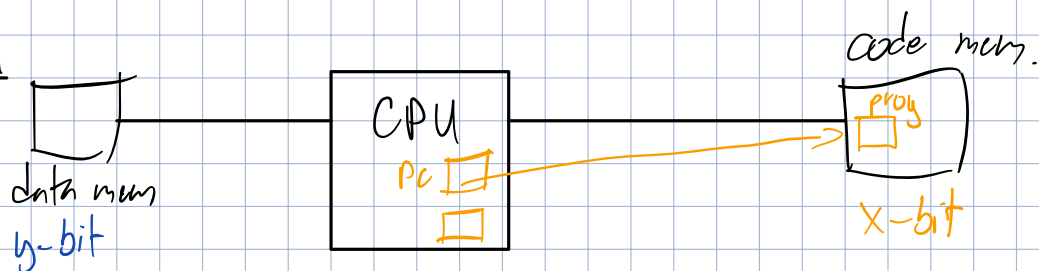
Čtení:

- 1 write address reg. $\rightarrow$ 1 f_c word 2x3
 - 1 read data reg. $\rightarrow$ 1 f_c word 2x3
- } 4B

pohyb mám více registru,
mám reg. adr. prostor,
takže i definuji, do kterého
registru chci psát/číst 2.

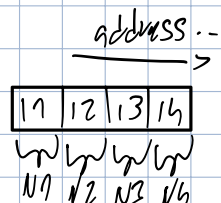
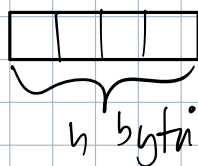


7. přednáška



Instruction (Instruction set)

- instrukce / funkce procesoru

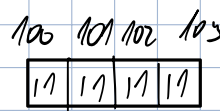


Registr procesoru:

- program counter (stav na jaké instrukci jsem)

posun na další instrukci

- jestli má instrukce více adresách, udává se nejvyšší adresa (báze)



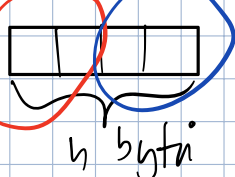
base add = 100

offset = báze adresy

OP code

Arg.

implicitní
explicitní



jednoznačný identifikátor

Instruction pointer

to takhle z code memory

Op code je ve velikosti slova code memory

arg. je ve velikosti slova data memory

to takhle z data memory

Machine code:

- série instrukcí

Compiler: $\rightarrow$ chodí se přímo do adres paměti

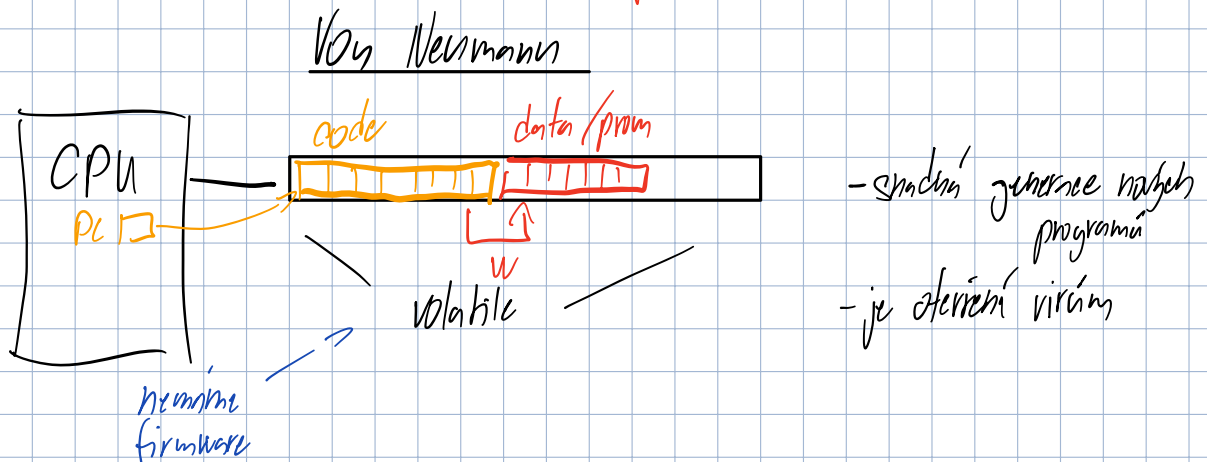
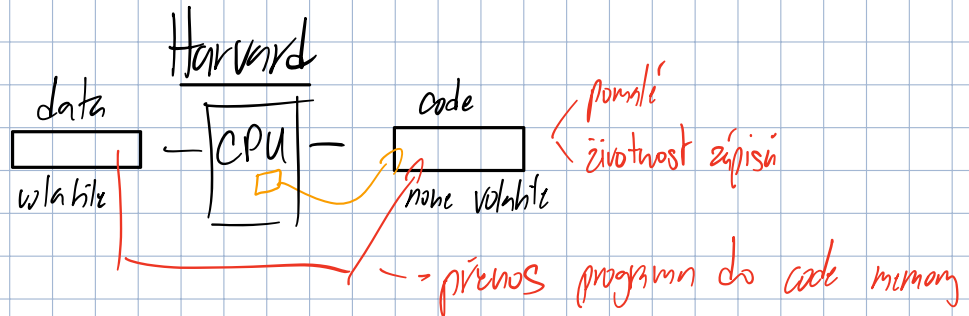
- překládá textového zápisu do strojového kódu

Interpreter: $\rightarrow$ je pomalý: na jednu instrukci kódu se provede spousta instrukcí interpretu.

- rovnou z místa překládá a spouští

Endianness dat: Little/Big Little: LSB first $\rightarrow$ 78 56 34 12
Big: MSB first $\rightarrow$ 12 34 56 78

- pořadí zápisu jednotlivých bytů určuje hodnoty 0x12345678
- potřeba dohodnout problém (dává procesorem, Intel $\rightarrow$ Little End)



Processor MOS 6502

- 8 bit - 8 bit word - umí počítat do 8 bit čísla
- 16 bit adresový prostor

Assembler:

- jednodušší textový zápis instrukcí procesoru pro snadnější správu

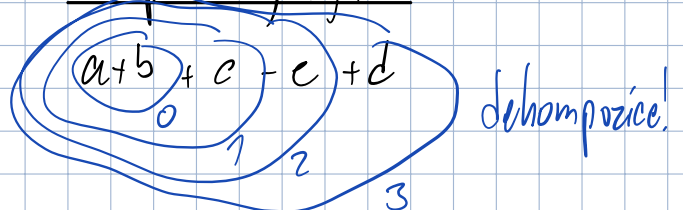
Registry:

- Speciální Program counter
- obecné = mezinárodní

$a = b + c$

LOAD	addr (b) $\sim \rightarrow$ R1
ADD	R1 + addr (c) $\sim \rightarrow$ R2
STORE	R2 $\sim \rightarrow$ addr (a)

Komplikovaný výraz:



6502/x86 namí operací

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

6502

0
\$EA
 $PC := PC + 1$
NOP

Basic operations:

Do nothing
offset from base
machine code
Program counter
Assembler

Intel x86

0
\$90
 $EIP := EIP + 1$
NOP

2 byte argument

0 1 2
\$hC xx_0 xx_1
 $PC := xx_1 xx_0$

JMP $\$xx_1 xx_0$

Jump

offset from base
machine code
Program counter

Assembler

4 byte argument

0 1 2 3 4
\$F9 xx_0 xx_1 xx_2 xx_3
 $EIP := xx_3 xx_2 xx_1 xx_0$

JMP $xx_3 xx_2 xx_1 xx_0 h$

Little
endianity!

(immediate) Load (instruction)

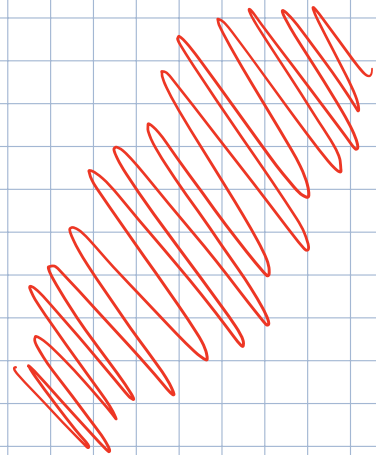
0 1
\$A9 xx_0
 $PC := PC + 2$

LDA # $\$xx_0$

$A := \$xx_0$

offset from base
machine code
Program counter

Assembler



(immediate) Load (absolute address)

0 1 2
\$AD XX₀ XX₁

PC := PC + 3

LDA \$XX₁XX₀

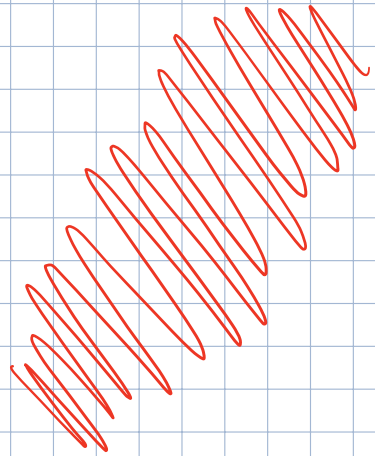
A := MemRead Byte(\$XX₁XX₀)

offset from base

machine code

Program counter

Assembler



Existuju tri registri u 6205:

A X Y

LDA LOX LOY

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

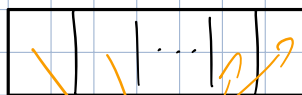
v 8bit slova

\$0100 \$FF00 ; \$0100 \$0101 \$FF00 \$FF01

8bit a = b 8bit

1) LDA \$FF00

2) STA \$0100



16 bit 16 bit

A

1) LDA \$FF00

2) STA \$0100

3) LDA \$FF01

4) STA \$0101

2x vice!

Priznamový registr: (flags register) (reg. P)

- flag (1-bit info)

N byte ... flag ...

- zero $\rightarrow$ byl poslední výsledek nula? 0/1

- sign/negative sign $\rightarrow$ byl poslední výsledek záporný? 0/1

- carry (přenos) $\rightarrow$ má různé funkce

často pro conditional jumps

6502

CLC (clear carry) P.carry := 0

SEC (set carry) P.carry := 1

- akumulátorová architektura

- reg. A je akumulátor

- většina operací pracuje jen s A.

Bitové operace (m 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 rlc left

ROR A A rlc right

? NOT EOR #\$FF

$$a = a \mid b$$

$a, b \in \text{uint}_8$

$a = \$A000$

$b = \$B000$

LDA \$A000

ORA \$B000

STA \$A000

6502 Little End

$a, b \in \text{uint}_{16}$

$\swarrow \quad \nearrow$
 $\$A000, \$A001 \quad \$B000, \$B001$

LDA \$A000

ORA \$B000

STA \$C000

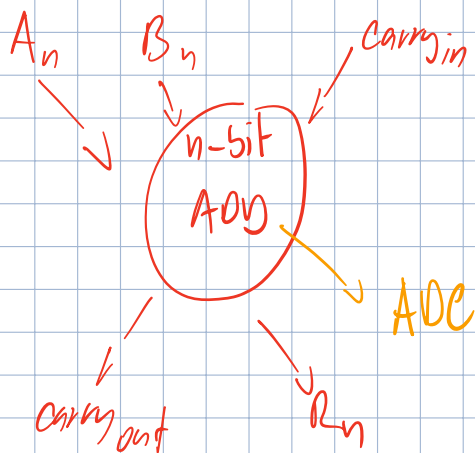
LDA \$A001

ORA \$B001

STA \$C001

Sečítání:

- normalize (zero/sign extension)



Sečet můžeme reprezentovat pomocí 2-bitů:

0-3 $\swarrow$

A	0	...	1
B	0	...	1
C	0	...	1

carry = všechny možné varianty

- takže můžeme sečítací jednotku pro n-bitů,
 kterým můžeme řešit

ADC imm/addr

result := $A + \text{imm/addr} + P.\text{carry}$ (tohle je 7 bitů)

$P.\text{carry} := \text{result}.8$ (poslední bit jde do registru)

$A := \text{result}.7 \dots \text{result}.0$ (zbylých 8 bitů je můj výsledek)

$C = a + b$

$\swarrow \quad \uparrow \quad \nwarrow$
 $\$C000 \quad \$A000 \quad \$B000$

LDA \$A000

CLC

ADC \$B000

STA \$C000

$\rightarrow$ musíme si pro jistotu carry před ADC uvoľnit.

$C = a + b$, $a, b \in \text{uint}_8$ — — jelikož je Little End, musím nejdřív počítat LSB a jít postupně k MSB.

↑
\$A000, \$A001 \$B000, \$B001
\$C000, \$C001

LDA \$A000

CLC

ADC \$B000

STA \$C000

LDA \$A001

ADC \$B001

STA \$C001

→ teď si carry NEKLAŽU!

→ truncation (zakroužení posledního carry)

Sčítání zmíněných čísel:

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

Odečítání: ?

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

Subtract with borrow (SBB) (x86)

$13 = 00001101$
 $- 7 = 11111011$
 $6 = 00000110$

↑ 10₁₀ = 2
 ↑ 11₁₀ = 3

1-1=0 1-2 nýde, takže 3-2 = 1

uhlédá se borrow do „carry“ bitu.

!!
 NOT A
 INC A
 ADD value

→ tohle je Two's Comp.

in 6205

EOR \$FF

CLC

ADC #1

CLC

ADC #value

Subtract with carry (SBC) - 6205

- Stejně jako SBB, jen

Carry = 1 $\Rightarrow$ Borrow = 0
Carry = 0 $\Rightarrow$ Borrow = 1

Negace!

$$A - X - B$$

$$A - X - B + 256$$

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

$$A - X - 1 + C + 256$$

$$A - X + C + 255$$

$$A + (255 - X) + C$$

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

$$X = \text{NOT}(X)$$

$$\text{ADC } X$$

$$256_{10} = \cancel{1}000\ 000_2$$

truncation

no effect

1-bit

$$B = \text{NOT}(C) = 1 - C$$

$$1 - 1 = 0$$

$$0 - 1 = 1$$

$$255 = 1111\ 1111$$

$$- \text{ničco} = 1010\ 1010$$

$$= 0101\ 1101 \quad \text{NOT!}$$

$$255 - X = X$$

Pozor!

CLC

SBB

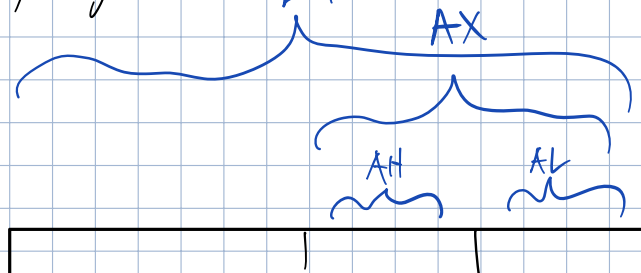
chci borrow 0

STC

SEC

Rozdí/ mezi 6205 a x86

- více registrů, segmentování: EAX



i když bude
pouzat s „virtuální“
AL, bude se dělat i
truncation.

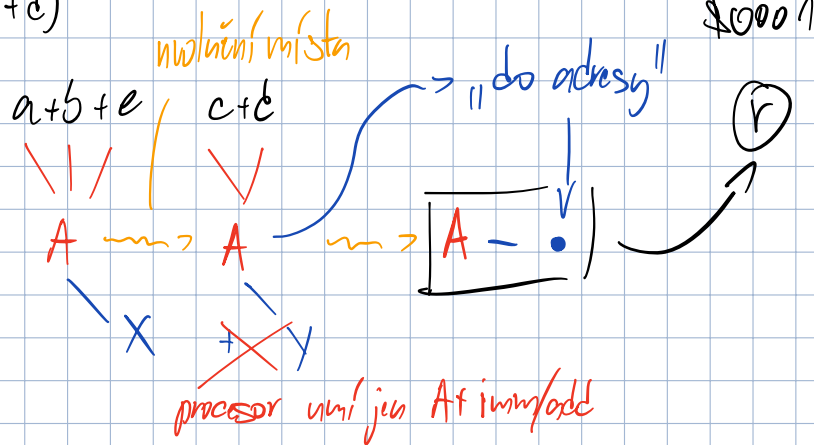
6205

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

$$\text{Temp1} = a + b + c$$

$$\text{temp2} = c + d$$

$$r = \text{temp1} - \text{temp2}$$



Dobud to je, nezapisovat do pameti, jelikož je to dleho pomalejší...

LDA \$A000
CLC
ADC \$B000
CLC
ADC \$F000
TAX

LDA \$C000
CLC
ADC \$D000
STA \$AAAA

TXA
SEC
SBC \$AAAA
STA \$0001

\$AAAA je dočasní místo
✓ pameti

Rychlost operací:

Mějme tablet procesor x GHz, jednotka 1 takt

Najrychlejší: AND, OR ..., SHL, SHR

ADC r₁, r₂ / SBB ~ 1 takt

Pomalé: LOAD, STORE

~ 10 taktů
ADC r₁, [r₂] ~ 10 taktů

- zpracování jedné min 8 bit procesor
a sčítání 16 bit, tak násobím čas $\frac{16}{8}$ krát.

Další pomalé:

MULT → ~ 10 takt

DIV → ~ 100 takt

x86/x64

✓

✓

ARM

✓

✓/x

SW

Microcontroller

✓/x

x

SW

6502

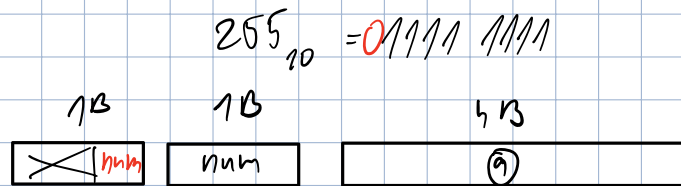
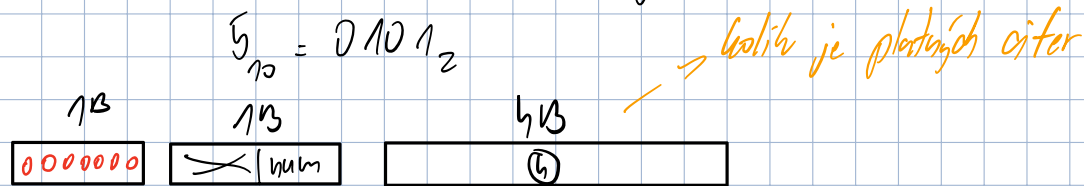
x

x

SW

9. přednáška:

Číslo v Pythonu



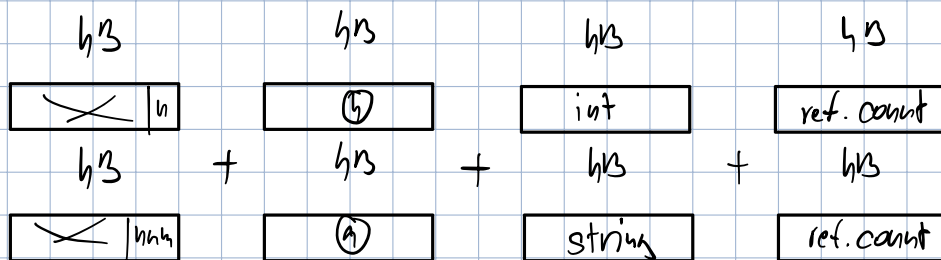
V C/C#/Java/C++

- deklarují se proměnné

Tohle v reálu ukládají do 4B

Pointer v Pythonu stojí 8B

normalizuje
se tak
části



↑
hodnota

↑
počet platných
cifer

↑
identifikátor
typu

↑
globální kolekce
reference

80%
overhead

$$\text{Cellum} = \underline{\underline{2 \times 4B}}$$

Garbage Collector:

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

Python 10x pomalejší
jako C++.

Interpreter dalších 10x

Cellum 100x pomalejší.

Proč je Python takový?

zároveň v Pythonu
jsou všechny objekty imutabilní.
když udělám: $x = x + 1$, vytvoří se nový objekt $x + 1$.

V C#: `uint_8` → $x = 255_{10} = 11111111_2$
 $x + 1 = 100000000_2 = 0_{10}$

-5 - 256

nejmenší hodnoty, které se
rovnou číselní předělu

V takových jazycích to přetáhne / pootáhne! To se v Pythonu nestane.

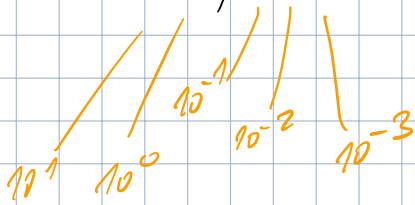
Vstupy/Výstupy umím správně zvolit velikostní, ale uprostřed
výpočtu to může vystrčit až moc.

Reprezentace "neokladných" čísel:

leading zeros

trailing zeros

000 20,625 000



- v 10kové soustavě

V 2-kové soustavě:

20,625

tohle je fixed point

0010100,1010000...

2⁻⁵ 2⁻⁴ 2⁻³ 2⁻² 2⁻¹ 2⁰ 2⁻¹ 2⁻² 2⁻³ 2⁻⁴

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

Fixed point:

- celkový počet bitů
- rozdělení integer + fraction bitů čísel
- např: 5+3
- když přeteče, ztratíme informaci

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

00001,010 + 00001,111

jako unsigned:

00001010 → 10

00001111 → 15

00011001 → 25

0011,001₂
3,125₁₀

Umí počítat pomocí
klasických operačních procesů.

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

Reálné násobení/dělení:

Pohyb násobím mocninou dváhek, platí zase SHL/SHR

- při dělení ale ztratíme přesnost za desčinnou částí

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

Při dělení se ale ztrácí desčinná přesnost

MULT $\rightarrow$ ~ 10 fahit
DIV $\rightarrow$ ~ 100 fahit

x86/x64

ARM

Microcontroller

6502

✓

✓

V/x

X

✓

V/x

X

X

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: if small enough

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

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

X

NO NO

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

SHR 1
1 1 1 0 1 1 = -5
0 1 1 1 0 1 = 29

X

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: $-1010010100000000,000\dots = 1,0100101 \times 2^{15}$
 B: $+ \dots\dots 0000,0600000010100101 = 1,0100101 \times 2^{-9}$
tabulka bych spotřebovala více bitů...

Normalizovaný zápis:

- no leading zero
 - před čárkou jen jedna cifra
- 1, ...

Floating point:

- exponent
- mantisa

$1,0100101 \times 2^{15}$
 exponent
 mantisa

Definujeme formát: 32 bit $\rightarrow$ 5b exp, 26b mantisa

není to přesné, ale řádově správné

exponent, mantisa

~~1 0 1 0 0 1 0 1~~
~~0 1 0 0 0 0 1 0 1~~

Truncation

$\rightarrow$ ztrácíme jen „malé“ hodnoty

- exponent celí $\pm$ číslo

Se skrzten mantisou

$\rightarrow$ víme, že první cifra mantisy je 1. Můžeme posunout a přesnost si jen posunujeme.

neznáme se Two's Complement, ale reprezentaci posunem:

Bias:

0 - 255
 -127 128

$-n \rightarrow +n$
 0 $\rightarrow$ 255

Definujeme, o kolik posunujeme

Bias (-127) (o kolik posunujeme hodnotu vzdy)

sgn = $\begin{cases} 0 = + \\ 1 = - \end{cases}$

sgn, exponent, mantisa

A: ~~1 1 1 1 1 0 1 0 1 0 0 1 0 1~~
~~0 1 0 0 0 0 1 0 1~~

Máme 5 bit exponent, tedy 0-31

Bias (-15)

$2^{15} = 15 + 15 = 30_{10} = 11110_2$
 $2^{-9} = -9 + 15 = 6_{10} = 00110_2$

sgn, exponent, mantisa

B: ~~0 0 0 1 1 0 1 0 1 0 0 1 0 1~~
~~0 1 0 0 0 0 1 0 1~~

- Nefunguje klasická hardware aritmetika.

Aritmetické operace se řeší SW $\rightarrow$ je to ale ultra pomalé.

Některé procesory umí HW softwarí floatů. $\rightarrow$ to je o něco pomalejší

x86	ARM	MicroController	6502	- ne všechny procesory to mají
HW	HW/SW	SW	SW	manipulace exponent

Potřebu mít správný formát float čísel (IEEE 754) $\left\{ \begin{array}{l} \text{single} - 32 \text{ bit } (23+8+1) \\ \text{double} - 64 \text{ bit } (52+11+1) \end{array} \right.$

Python:

float = 64 bit

Assembly:

float 32

float 64

C#:

float = 32 bit

double = 64 bit

Aritmetické operace FLOATŮ: (32 bit)

trailing zero dle 23 bitů

A+B:

A=

1,010010100000000000000000, Truncation

B=

0,100000000000000000000000, 10100101

denormalizace

23 bit Manipulace

Velké číslo + něco malého = Jan něco velkého. Čím větší přesnost, tím méně chyb.
(U Pythonu to neřešíme)

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

- často je lepší převést na jinou celou jednotku.

- existují tak malá čísla, která už nikdy reprezentovat.

$< >$ podle se některými hlubokými exponenty reprezentují ty záporné.

IEEE 754

manipulace = 0 $< >$ $\pm \infty$

x/0 $\leadsto \pm \infty$

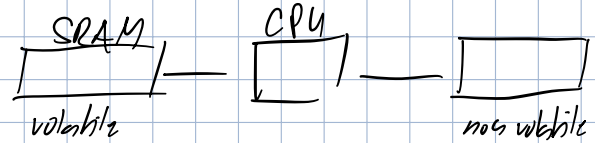
$\infty/\infty \leadsto \text{NaN} - \text{Not A Number}$

*ROM -> non-volatile

ROM: (Read Only Memory) [R]

- read only memory
- jednorázově zapisáno, pak se jen čte

MC -> většina jsou Harvardi



PROM: (Programmable ROM) [R/W]

- user programmable device (once)
- konkrétní diody se "spálí"

EPROM: (Erasable PROM) [R/W]

- "nehodněně" mnoho zápisů
- mazání UV zářením (vystavit na slunce)

EEPROM: (Electrically EPROM)

- "nehodněně" 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 protahují
- za několik let se data ztrácejí!!
- čten/zapisují se jednotlivé bloky

SRAM

10 GB/s

~ 1 ns

KB - 1GB - MB

DRAM

1-10 GB/s

~ 10 ns

1 GB

EEPROM/flash

100 - 1000 MB/s

~ 100 ns

100 GB - 1 TB

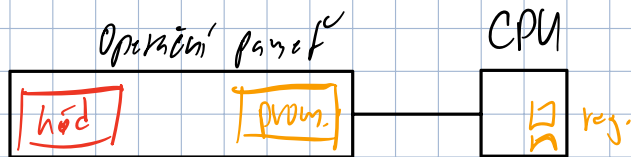
	EEPROM	FLASH
sekvencně	✓	✓
random read	✓	✗
random write	✓	✗

2 důsledků blokového přechodu
zapisují větší celý blok

Flash reflektuje novější programy, které čtou většina sekvencně a nepřepisují se tak často.

NVRAM - NonVolatileReadWriteMemory
- to jsou flashe

Nějme PC: (Von Neumann)



A/D:

- analog/digital converter

DAC:

- digital/analog converter

GPIO: (general purpose input/output)

- má direction register (jestli jsou linky r/w)
- input register (stav, které linky jsou vstupní)
- output register (stav, které linky jsou výstupní)

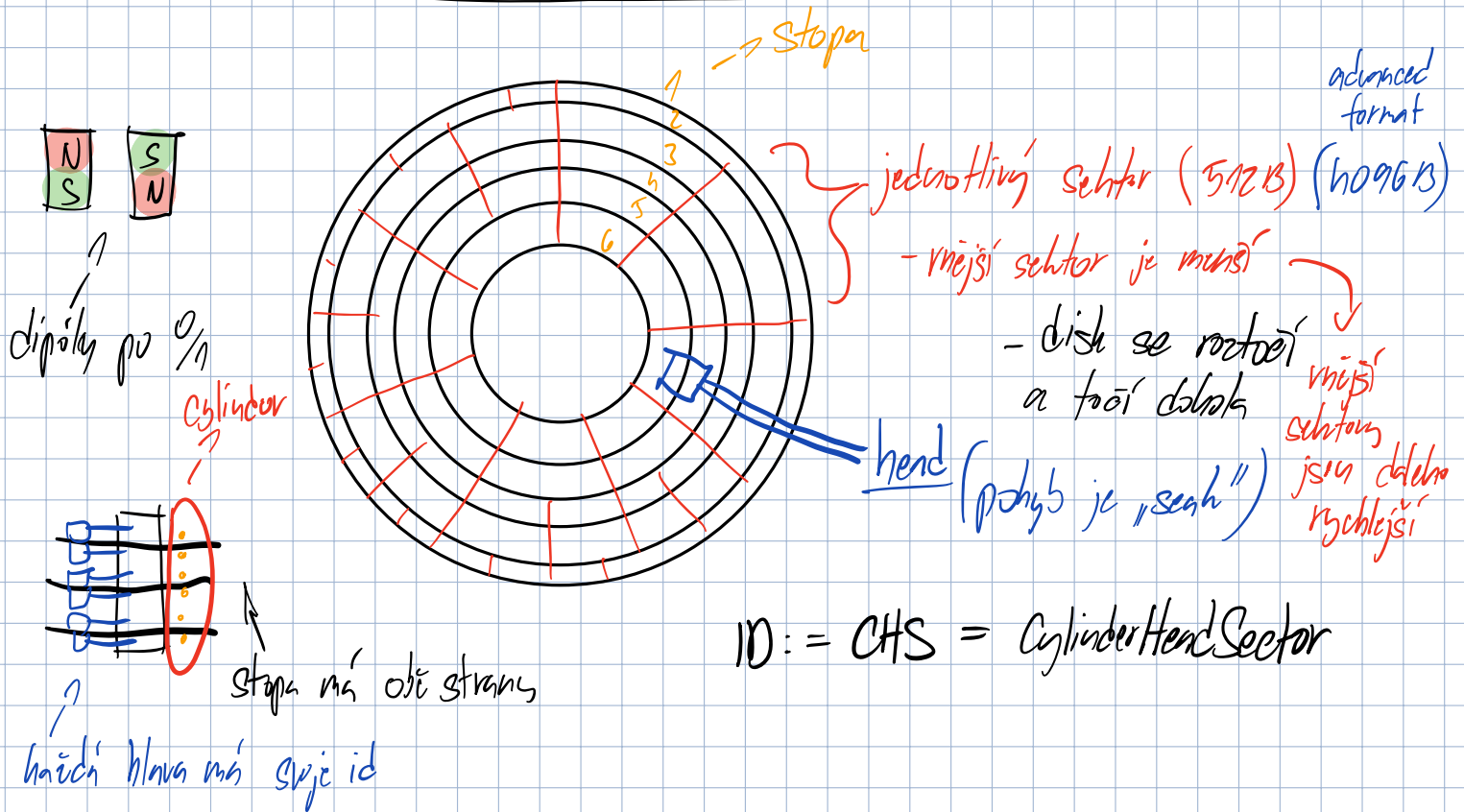
Mass Storage Device:

- non volatile pro ukládání dat (zejména uživatelé)
- zaznamenávají stav, který mají dát po restartu

Buffer pro stav:

- většinou EEPROM, kde se ukládají meziprogramy (instalace... atd...)

První disk / hard disk drive (HDD) : (5400 - 7200 - 10 000 - 15 000)



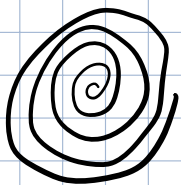
- schvácení přístup ☒ ✓
- schvácení přístup ☐ X
- náhodný přístup ☒ ✓/X
- random stopa ☒ X (musí se brát)

Výhody:

- kapacita (~10TB)
- cena
- archivace (lze se natáčet)
- opravdu nemožný počet zápisů!!!!

CD/DVD/Blu-Ray (optický)

- ① Bud' se světlo laserem odrazí zpět nebo ne.
 Proto se "vyplývají" CDčka. - takové vyplnění se má.
- zase sektory, jen jeden (spirální stopa)



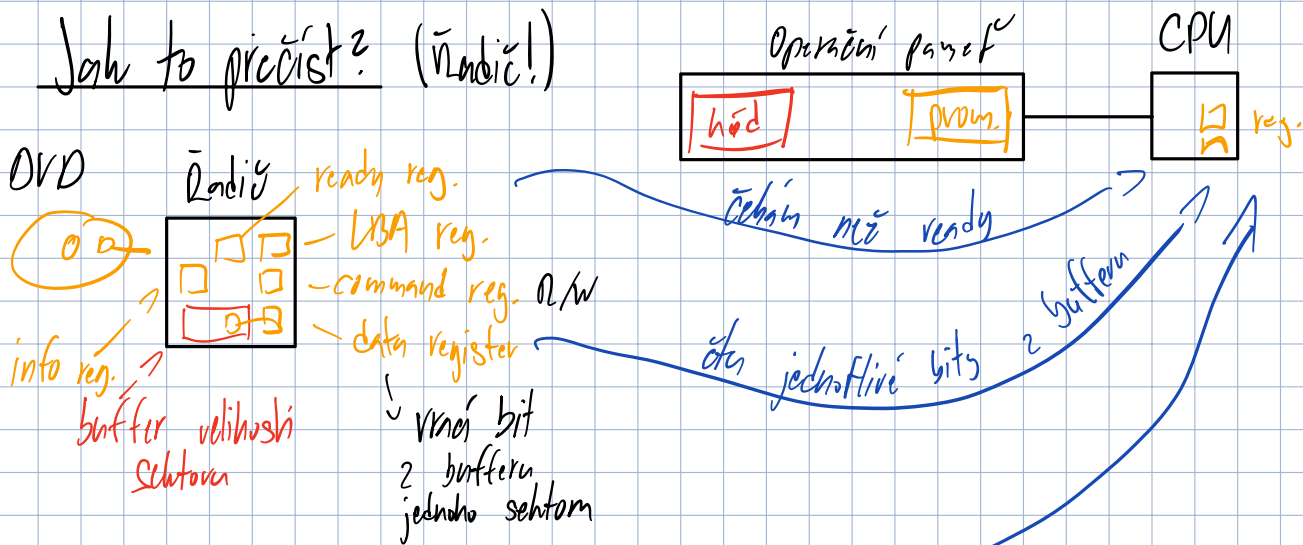
- rozdělení na sektory (LBA - Linear Block Addressing)
- ultrapomali random čtení, proto jen pro sekvence

Nevýhody:

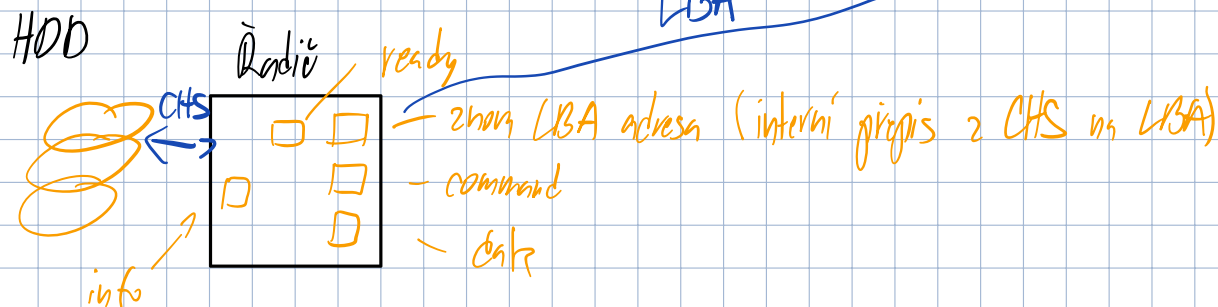
- rychlost
- nákladnost na hlavu

2048 B

Jak to přechází? (řadič!)



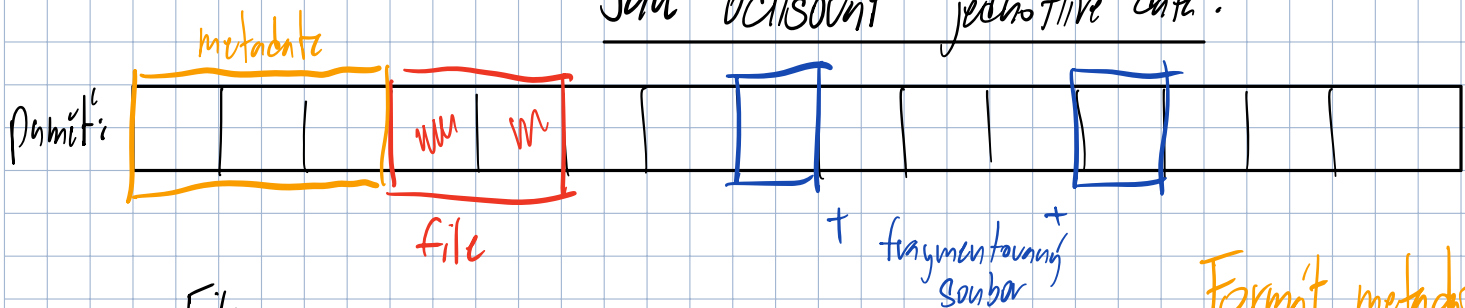
HDD



SSD

- Flash s indexem od HDD
- přepisování na LBA adresy zjmén
- 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 metadata:

Filesystem:

- potřebná informace

(OS) OPEN/READ...

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

f. close()? Metadata načítám jen jednou.

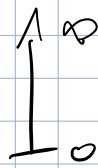
12. přednáška

Jak uhlédít fotku?

Budu mít nějaký metr, do kterého budu uhlédít intuitivně na tuhle metu.

	0	1	2	3	4	5
0	00	00	20	..	..	..
1	01	01	21	..	..	..
2						
3						
4						
5						

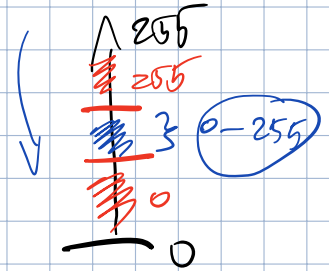
- nejčastěji uhlédáme větší zkus doprava,
pak jdeme do místa vzhledu.



pixel
bits per pixel (bpp) / Bitdepth

definujeme
tím hodnotu
světla

rozsaah a
posuvu



15bit depth (Monochrome)

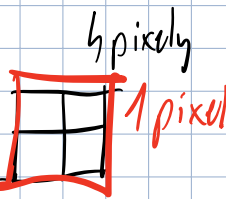
- jeden pixel = bílá/černá

8bit depth:

- všechny je HDR omezený, takže je má
- větší rozsah, vypadá lépe.

Differing

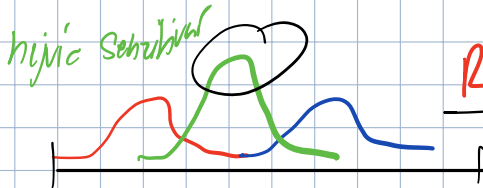
- spojíme několik pixelů dohromady,
pak ztrácíme rozlišení, ale máme 2ⁿ odstínů...



3bit RGB

Chceme barvu - musíme rozlišit frekvenci těch fotek:

- a každé slovo 0/1



RGB

- stabilně uhlédáme
tyto tři hodnoty
barvy.

na 4-bit máme zapsat
intenzitu barvy.

2B RGB (15bit depth)

RGB - 15 bitů (5+5+5)

16bit: (5+6+5):

- přidali jsme zelení
protože jsme na ní
ořezali

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

R 1B

G 1B

B 1B

2 1B

R 1B

G 1B

B 1B

A 1B

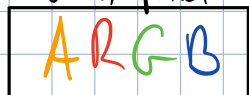
alpha barva. (průhlednost)

32bit/4B
depth

White: ^RFF ^GFF ^BFF

Black: 00 00 00

Jeden pixel



B B B B

-> potřeba definovat
MSB/LSB

Jak to uložit?

Metadata

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

Data

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

Textová data: (String)

- posloupnost znaků (písmena, číslice, spec. znaky, white-space)

<-> posloupnost kódů

- děláme to pro zpětnou rasterizaci

- v datech není dříve kódování -> musí se definovat předem!

Kódování:

- jednosměrné, obousměrné kódování

- mapa mezi kódy, délkou kódu (proměnná / pevná)

ASCII - 7 bit:

0 - 127

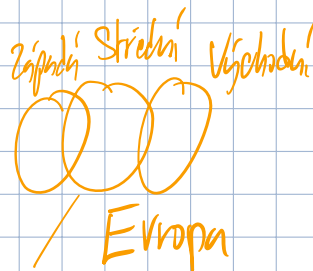
0 ≠ 0

A, B, ..., Z, a, b, ..., z, 0, 1, ..., 9

X X+1

X X+1

2 2+1



7 bitů uložíme do LSB bitů

128 - 255 neobsazeno

Code Page (lokální rozšíření)

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

MSDOS 852 (OAS) Latin 2
WIN 1250

UNICODE:

- všechny možné znaky

- UTF-32 - uložíme 32 bitový náhodný číslo

little/big endian

- vždy se ukládá v párech, jak se slovo čte. Takže pokud kombinují, střídám to.

13. přednáška

Memory controller:

- zajišťuje refresh DRAM
 - můžeme zapojit více paměťových modulů do jedné datové sběrnice
 - Modul₁ bude mít adresu 0
 - Modul₂ bude mít adresu x
- } pak možná do jednotlivých modulů a procesor nic není.
- zároveň můžeme na procesor napojit novější moduly s jiným protokolem, když to memory controller převede.

CPU vždy volá „chci byte z adresy $x x_0 x_1 \dots$ “

System BUS: (PCIe)

- doháním zapojit vše (operační paměť, zařízení...) na jeden BUS
- multi drop
- full-duplex

1) Adresa zařízení

2) Paměťové pakety

MWR STORE

memory write

memory read

MRC LOAD

} pakety na krátkou dobu, určité pro Memory control.

- multi master (Memory controller si sám říká, kdy posle data)

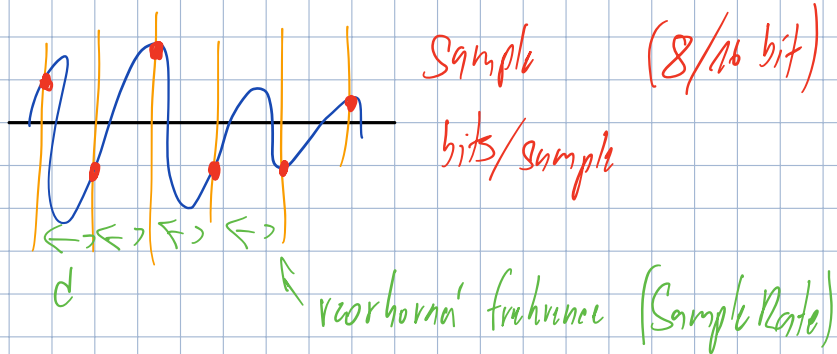
Memory Map IO:

- registry zařízení jsou nainstalované do adresního prostoru operační paměti
- výhodou je, že komunikace po celém BUSu je tak unifikovaná, ať už jde o jakýkoliv zařízení

HCI: (Host Controller Interface)

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

Zvuková karta:



Ideální DAC

- u zvuku je potřeba přesnost!!
- potřebuji ale buffer (Sound buffer)
takhle bude přesná časování
a ve správném čase čtení/převod

HW akcelerace (CPU offloading):

- odložení nejdůležitějších operací do HW místo SW, aby se SW odlehčilo.
- většinou ve zvukových kartách mají DSP (digital signal processor)
 - tak se provádí HW accel.

GPU:

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

Polling

- blokování CPU time "čekáním na"

Bootloader:

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

CPU startup:

- 1) IO -> x86: \$FFFFFFFF0
 - zbývá 16 bitů, kde uložíme JUMP na reálný začátek firmwaru

2) Nahrání uživatelského "SW"

- existuje i Option ROM, kde je ten zajímavý SW

→
občas třeba nastavit zařízení a pak poslat příkaz spustit firmwaru