Principy Počítačů

poznámky

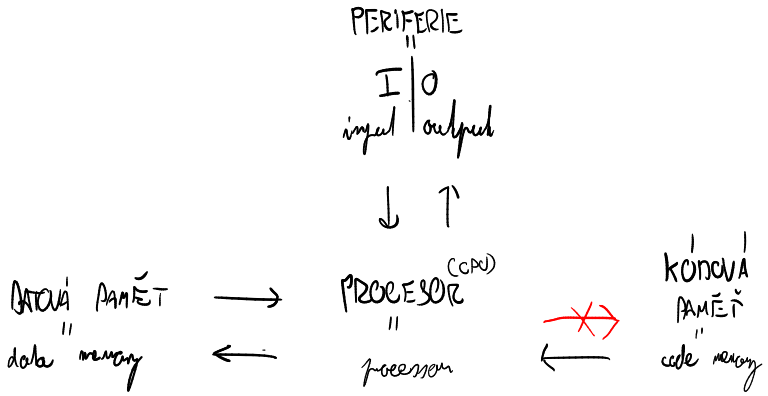
Tomáš Sláma
17. 1. 2020

Obsah

Úvodní informace

Tato stránka obsahuje moje poznámky z přednášky Pavla Ježka z akademického roku 2019/2020 | MFF (MFF UK). Pokud by byla někde chyba/nejasnost, nebo byste rádi něco přidali, tak stránku můžete upravit pull requestem (případně mi dejte vědět na mail.

Zjednodušené schéma počítače



Obrázek 1: Harvardská architektura

- vymyšlena na univerzitě v Harvardu
- **CPU** – vykonává instrukce
- **kódová paměť** – uchovává instrukce; je pouze pro čtení
- **datová paměť** – uchovává data, se kterými pracujeme

Historie

Charles Babbage (1837)

- mechanický stroj *analytical engine*
  - z dnešního pohledu plnohodnotný počítač
  - byl Turingovsky úplný!
  - nebyl nikdy fyzicky postaven
  - měl usnadnit počítání tehdy komplexního systému daní v Británii

Ada Lovelace

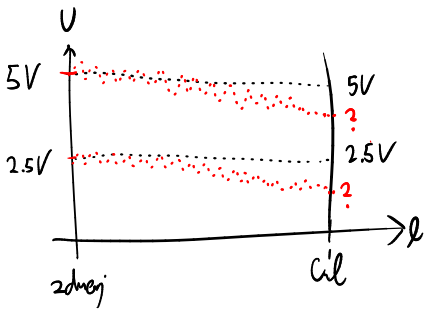
- dcera G. G. Byrona
- finančně Babbage podporovala
- příběhy o ní jako programátorce (psaní programů, ladění bugů) jsou vesměs nesmysl, ale:
  1. napsala manuál stroje
  2. napadlo ji, že by se mohlo pracovat i s **jinými informacemi než s čísly** (texty, obrazy, hudba…)

Kódování informace v počítači

- pojďme si zakódovat čísla $0, 1, 2, \ldots, 1000000$

Analogový přenos

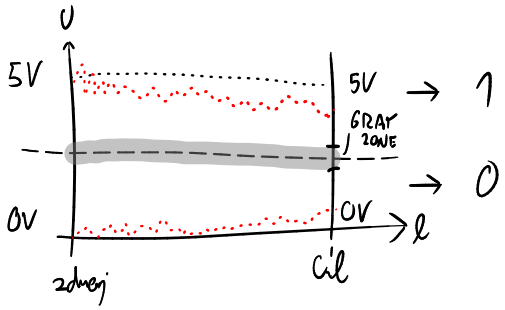
- kódování $1 = 1V$, $2 = 2V$ by nešlo, jelikož se musí čekat na bouřku
  - $0 = 0V$, $1000000 = 5V$ zní rozumněji
- v praxi zní skvěle, ale v realitě řada problémů, jelikož napětí ovlivňuje:
  - délka vodiče
  - teplota vodiče
  - elektromagnetické pole, které je vodičem jak generováno, tak přijímáno



Obrázek 2: ztrátovost analogového přenosu

Digitální přenos

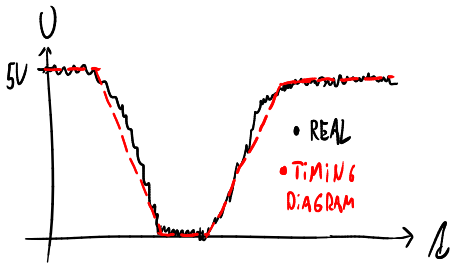
- problém je v intervalech, které se překrývají… co je odtáhnout od sebe?
- pouze *2 hodnoty* – logická 1 a logická 0
- jednotka *bit* (BInary digiT, značíme *b*)
- funguje s rozumným šumem (100procentní ale není)



Obrázek 3: digitální přenos

Sériový přenos

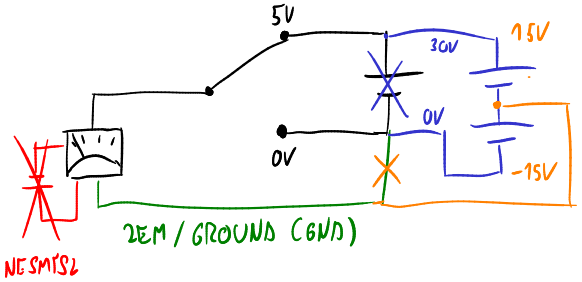
- způsob přenosu více bitů informace: pošleme je *za sebou*



Obrázek 4: přenos čísla 1011, znázornění odporu vůči změně napájení

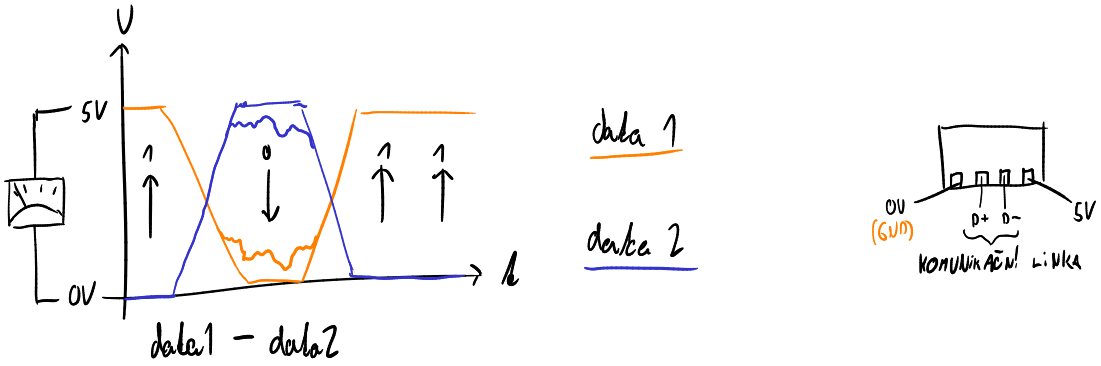
1. měření hodnoty **na měřáku**

- nezapomínat na to, že napětí je relativní



Obrázek 5: referenční přenos (barvy ukazují způsoby, jakými lze signál vést do cíle)

2. měření **rozdílu napětí** dvou vodičů
   - výhoda – elektromagnetické rušení signál neovlivňuje, jelikož rozdíl je relativní

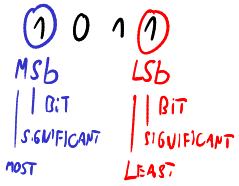


Obrázek 6: diferenciální přenos (vlevo); USB konektor (vpravo)

- délka bitů musí být jasně dána, aby přijímající dobře interpretoval informace
  – dnes bývá dána hardwarově
  – přenosová rychlost (= transfer rate)… **baud** (symbols / second)

MSb/LSb odbočka

- je potřeba se dohodnout, jak říkat různým bitům dvojkových čísel



- **LSb-first** – první v komunikaci přijde LSb, poslední MSb (MSB-first funguje analogicky)

Dohoda přenosu

- problém: hodiny se časem kvůli HW rozejdou – data pak stranám nedávají smysl

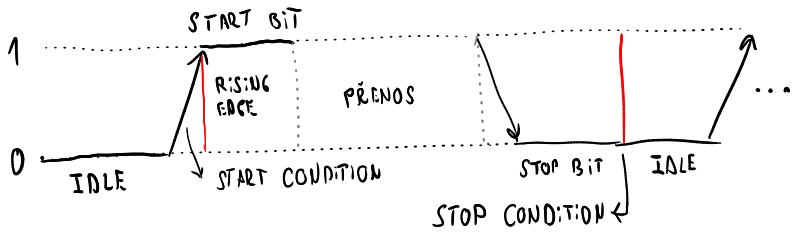
Řešení (1): nový stav

- vymyšlení *dalšího stavu* přenosu, ve kterém k přenosu nedochází

1. odpojení vodiče = **floating state** (plovoucí stav)

- lze detekovat, že je linka odpojena
- není to ideální, moc se nepoužívá

2. **idle stav**
   - na začátku je linka **idle** (tzn. přenos neprobíhá)
   - používá **RS-232**
   - start dohodou, většinou **rising edgem** (dobře se detekuje), poté držen (většinou 1) **start bit**
     - start (a end) bit je potřeba – když by to jen vystřelilo, tak by to druhá strana kvůli šumu vůbec nemusela detekovat
   - se startem (rising edge) se synchronizují hodiny
     - nejsou perfektní (mají tendenci se rozcházet) – omezení přenosu na n bitů, kde n je konstanta (v reálu $8b = 1B$)
       * pozn.: $1B$ jsou 2 šestnáctkové znaky
   - velký overhead… z $10b$ je jen $8b$ datových… $\underbrace{1000}_{\text{clock}} \cdot \underbrace{8/10}_{\text{start + end}} = 800\ bps$



Obrázek 7: přenos s idle stavem

Řešení (2): hodinový signál

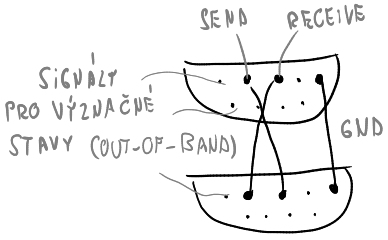
- používá I^2C
- nový (tzv. referenční) vodič s digitálním (hodinovým) signálem
- diktuje, kdy lze na datovém vodiči číst
- zdá se, že je 50% overhead, jelikož se data mohou detekovat pouze na rising edge
  - většinou ale maximální rychlost není potřeba
  - když je potřeba, tak lze obejít: detekce na rising i falling
    * značí se **DDR** (double data rate)
    * problém to hardwarově detekovat – používané jen tam, kde je rychlost nezbytná

Řešení (3): průběžná korekce

- bylo by fajn průběžné synchronizovat na rising edge… co když ale chodí samé nuly (nebo jedničky)?
- **clock recovery** (obnova hodinového signálu) – převod dat z $8b \to 10b$:
  - 4krát více možností – vyberou se jen ty „hezké“ (kde se střídají 1 a 0)
  - kódování/dekódování je prováděno pomocí tabulky
  - nemusí to nutně být 10, existují i jiné konfigurace

Typy přenosu

1. **half-duplex**: 1 datový vodič – zařízení se v přenosu střídají
   - komplikované
   - nikdy nelze posílat najednou oběma směry
2. **full duplex**: 2 nezávislé simplexí linky
   - např. RS-232 – 2 datové + 1 zem

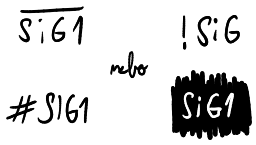


Obrázek 8: RS-232

- **význačné** (out-of-band) stavy = dochází baterka/změna módu/…
  - někdy je potřeba sdělit uprostřed přenosu
  - digitální signály (on/off)
  - lze přes to zařízení i napájet – některá to tak dělají

Značení

- většinou 1 je pravda a 0 nepravda, někdy se ale hodí prohodit:



Obrázek 9: inverzní logika

- někdy je také potřeba rozlišovat soustav, ve kterém číslo je:

$$\text{desítková} \ldots 23 = 23_d = 23_{10} = 23_{dec}$$

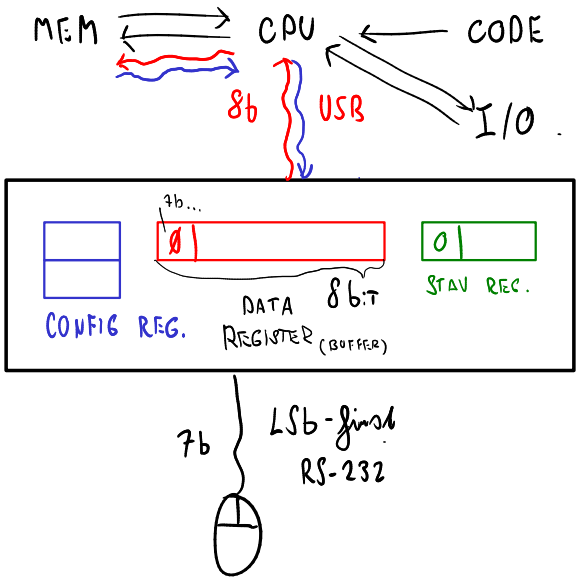
$$\text{šestnáctková} \ldots \$23 = 0x23 = 23h = 23H = 23_{16} = 23_{hex}$$

Komunikační protokol

- dohoda, jak přenos vypadá…

$$\underbrace{\text{den}}_{9b} / \underbrace{\text{měsíc}}_{5b} / \underbrace{\text{rok}}_{7b} = \underbrace{101101010/11010/0010001}_{\text{pa(c)ket}}$$

- problém: starší zařízení mají $1B = 7b$
  - řešení: řadič (controller), který přijímá data ze zařízení a zpracovává je



Obrázek 10: řadič (controller)

- config register – nastavení přenosové rychlosti, parity,…
- stav register – zda se načetl celý byte

Binární odbočka

Operace

- binární:
  - `OR`: `|` – alespoň jedno
  - `AND`: `&` – oboje
  - `XOR`: `^` – právě jedno
  - `SHL` a `SHR`: `<<` a `>>` – **posouvá k MSb, ne doleva/doprava**
    - `ROL`, `ROR` (*bit rotation*) – jako posun, ale cyklí čísla; opět k MSb
- unární:
  - `NOT`: `~` – opak

Záporná čísla

Řada způsobů, jak to dělat (špatně):

1. jako celá čísla, ale první bit je znaménko
   - docela k ničemu – žádné normální bitové operace na to nefungují (sčítání, odčítání, porovnání…)
2. **one's complement** (jedničkový doplněk) – u záporných se prohodí 1 a 0
   - funguje porovnání (kladné x kladné a záporné x záporné) a sčítání
   - problematické: máme *dvě nuly*
3. **two's complement** (dvojkový doplněk) – MSb je znaménkový bit, záporných je o **1 víc**
   - negace čísla je **flipnutí všech bitů** a **přičtení jedničky**
   - řeší to problém se dvěma nulami (negace dá jedničky a přičtení overflowne zpět na 0)
   - rozsah je -2^{n-1} až $2^{n-1}-1$ (asymetrické…)

Čísla v Pythonu

Implementace

- jako pole… ukládá si právě tolik cifer, kolik je potřeba (+ padding do bytu)
- také potřebuje vědět, kolik bytů to číslo má
  - tím pádem je max. velikost $2^{32} - 1$ bitů, což je… dost
- znamená to, že se operace chovají divně… `~255` je `-256` (`(0)1111110 -> (1)00000001`)

Převody mezi soustavami

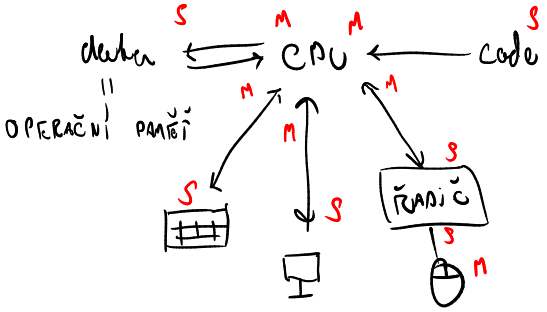
- literály: `0x123` (hex), `0o123` (oct), `0b101` (bin)
- převody: `int(str, base)`, `hex(num)`, `bin(num)`, `oct(num)`

Převody mezi datovými typy

- **truncation**: useknutí cifer, které se do menšího datového typu nevejdou
  - z definice nemusí dobře fungovat (máme méně čísel), může se dokonce změnit znaménko
- **extension**:
  - **zero** extension – doplnění nul na prázdná místa po zvětšení datového typu
    - nedává smysl u signed intů
  - **signed** extension – doplnění o MSb
    - dává smysl u signed intů, ale **ne u unsigned!**

Otročina

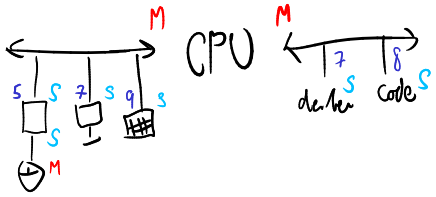
- pojmy **master** (řídící; řídí komunikaci, dává požadavky) a **slave** (řízený; vykonává požadavky)
  - v komunikaci vždy 1 master a 1 slave
  - role se mohou měnit
  - směr *master* $\mapsto$ *slave* je **write**, obrácený **read**



Obrázek 11: slave/master přenos

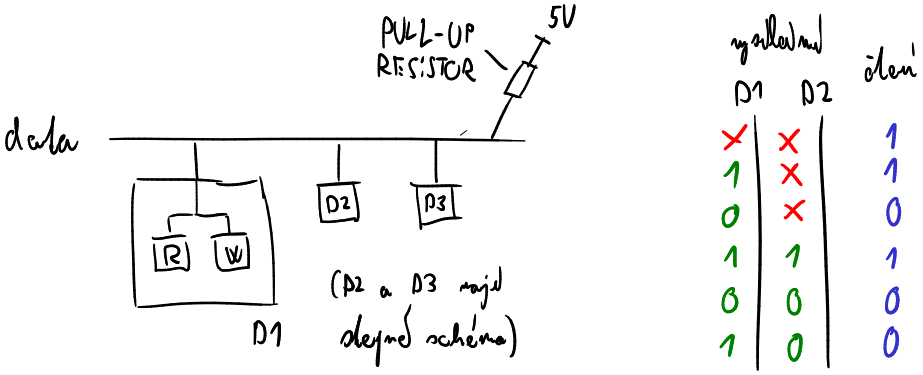
Připojení

- **point-to-point** linka:
  - z bodu do bodu – 1 master a 1 slave
  - nepraktické – reálně je procesor pořád master a museli bychom do něj vézt trilión linek
- **multidrop/bus** (sběrnice)
  - více slavů na jedné lince
  - pozor – sběrnice znamená něco jiného, ale dnes se to takhle říká



Obrázek 12: sběrnice

- rozlišují se **adresou,** která je pro každé zařízení na sběrnici *unikátní*
  - rozsahu těchto adres se říká **adresový prostor** (adress space)
- linky jsou half-duplexy bez floatingového stavu
  - vždy se mezi sebou baví 2 zařízení

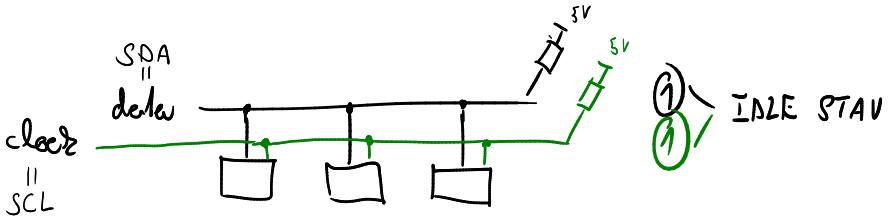


Obrázek 13: implementace sběrnice

- když nevysílá nikdo, je tam díky pull-up rezistoru 1; když někdo 0, tak 0
  - rezistor zařídí to, aby stav na lince nebyl plovoucí
- to, že se to spolu vlastně mlátí budeme řešit až na vyšší úrovni

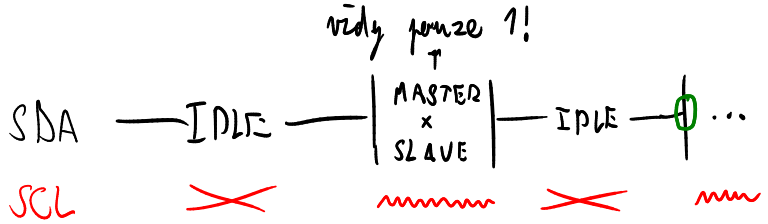
I²C (Inter Integrated Circuit)

- je to opravdová sběrnice
- podporuje **multimaster** – více zařízení mohou být master najednou
  - že musí se detekovat srážky, když chtějí 2 masterové vysílat najednou
  - rozdílné od USB (universal serial bus) – to je single master



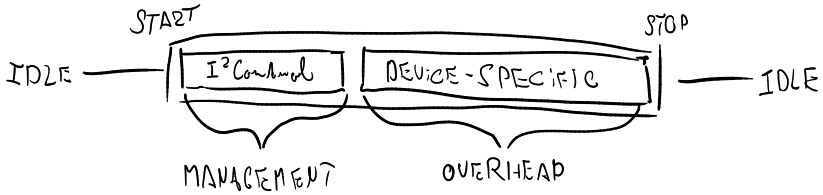
Obrázek 14: I2C sběrnice

- **SDA** = serial data; **SCL** = serial clock
- idle stav je vždy vyrušen masterem, který chce navázat komunikaci
  - rovněž *generuje hodinový signál*
  - začátek nastává, když nastane 0 na SDA a 1 na SCL (zakázaný stav!)



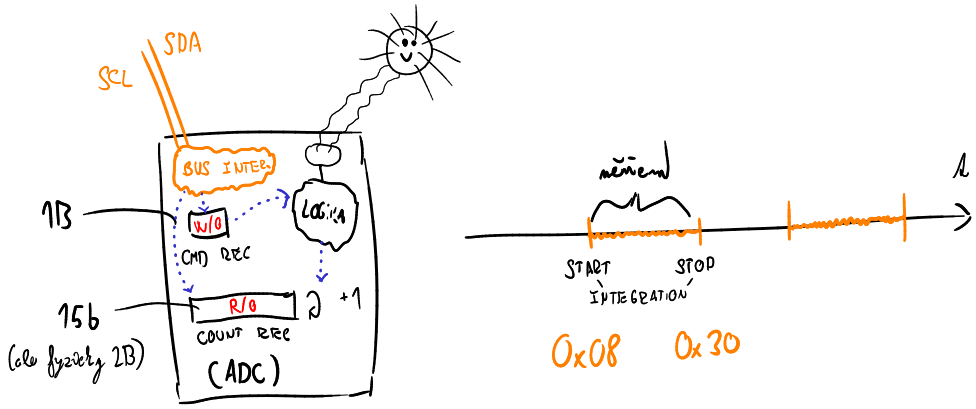
Obrázek 15: komunikace v I2C

- 9 bit na byte
  - 8 data (MSb-first)
  - 1 acknowledgement bit (ack = ano; nak = ne)
    - pro 0 je ACK, 1 je NAK, jelikož 0 stahuje… pokud tam slave není, tak tam bude 1
  - pořadí při přenosu vypadá následně:
    - write: `M/S/M/S/M/S...` (slave neustále potvrzuje že přečetl)
    - read: `M/S/S/M/S/S/M...` (slave potvrzuje že posílá a pak začne posílat)
- pro clock jsou dány standardizované rozsahy, aby to slave ustál
  - má možnost dělat **clock stretching** – pokud by nestíhal, tak může hodiny podržet na 0 (hold low)



Obrázek 16: struktura přenosu I2C

- **management** je obecný pro I^2C; zbytek je device-specific
  - 1-7 je adresa, 8 je r/w
- **payload** – to, „za co platíme“ (samotná data)



Obrázek 17: I2C zařízení (ambient light sensor)

- **ADC** = analog-digital converter – převod analogu do digitálu
  - přijímáme světlo (analogový signál)
- **bus interface** – řídí zařízení, komunikuje,…
  - pro vyrábění k jiné sběrnici stačí nahradit pouze tohle
- adresa je *pevně daná*… na sběrnici může být pouze jedna
- zařízení má 2 registry (read only, write only)
  - znamená to, že *nemusíme rozlišovat*, ze kterého registru číst a do kterého psát

Paměť

- **paměťový adresový prostor** – rozsah adres v paměti
  - pro $200B$ bude $8b$
  - většinou i pro $128B$ bude $8b$ (se $7b$ se nepracuje skvěle)
  - pro $256B$ může být $16b$ – dobře se to scaluje (při upgradu hardwaru)

Jednotky

- $1kB$ není $1000B$ (nepraktické), ale $1024B$
  - používají všichni… až na výrobce pevných disků
  - dnes se hodí používat jasnější značení $1024B = 1KiB$ (ki-bi-bajt)

paměť	adresový prostor	v reálu
$1kB$	$10b$	$16b$ prostor ~ $64kB$ adres
$1MB$	$20b$	$24b$ prostor ~ $16MB$ adres
$1GB$	$30b$	$32b$ prostor ~ $4GB$ adres

Typy paměti RAM

- RAM = random access (memory) – můžeme přistupovat k libovolné hodnotě a bude to trvat *stejně rychle*
- zpravidla bývá r/w a **volatile** (po vypnutí ztratí data)

SRAM (static RAM)

- $1b$ = 4 až 6 tranzistorů
- kapacita je malá (řádově MB)
- všechny přístupy mají trvat stejně dlouho (staví na tom algoritmizace), ale v realitě:
  - **sekvenční** přístupy jsou rychlejší
  - **obrácené sekvenční** jsou něco mezi
  - **náhodné** přístupy jsou pomalejší
- přístup 10-100$GBps$; ~$1ns$

DRAM (dynamic random access memory)

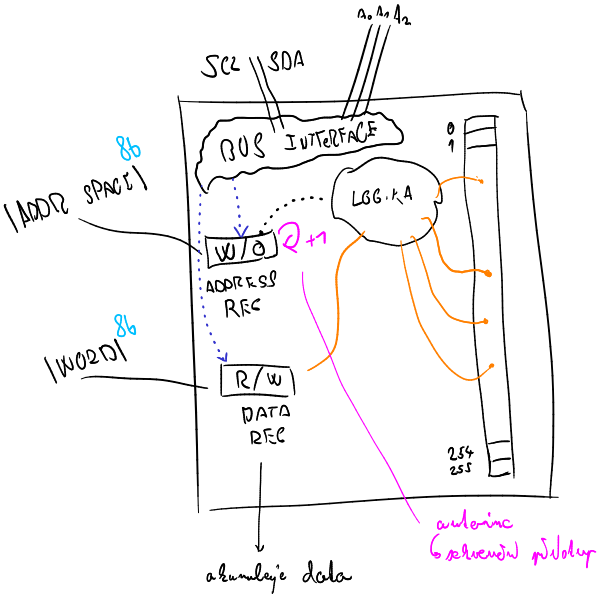
- $1b$ = 1 tranzistor a 1 kondenzátor
- levnější a větší (řádově GB), ale pomalejší (nabíjí/vybíjí se pomalu)
- v rámci přístupu pro ni platí (zhruba) to samé jako pro SRAM
- hodnoty si pamatuje *krátce* (kondenzátory…)
  - každou $1ms$ je potřeba refresh (přečteme a zapíšeme)
  - částečně může za pomalý access time (pořád se čte a zapisuje)
- přístup 1-10$GBps$; ~$10ns$

Co je to ~~jsoucno~~ slovo?

1. **správná definice** – jednotka přenosu
   - pro n-bitové zařízení je to n-bitové slovo
   - pro $8b$ paměť to např. znamená, že lze vyžádat pouze $8b$
2. **špatná definice** – $16b$ (dáno historicky)
   - *double word* (dword) = $32b$
   - *quad word* (qword) = $64b$

Rozbor PCF8570 SRAM paměti

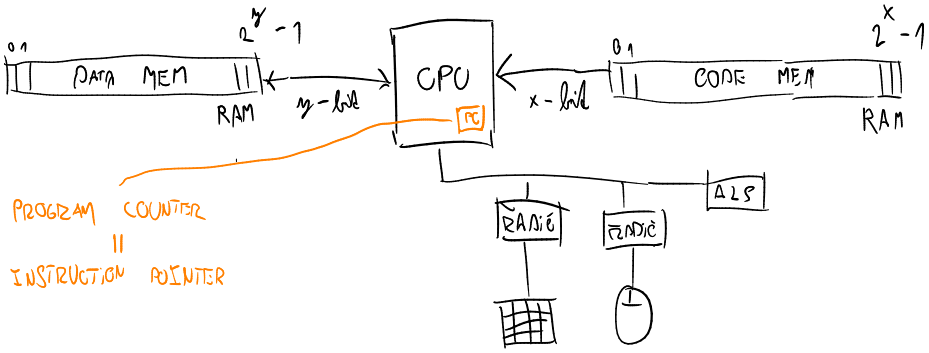
- $3b$ programovatelná adresa, `1010` vestavěná
- rychlost přenosu přes I^2C je cca. $100000Hz \sim 100000bps$
  - I^2C overhead je $x/(3 \cdot 9) \rightarrow 3708Bps$
    - na každou transakci ($8b$ + ack) přeneseme 3B, z toho jen 1 jsou data
    - jde to zlepšit – burst přenos (nezačínáme další přenosy, jen sekvenčně čteme/zapisujeme): $(x - (2 \cdot 9))/9 \rightarrow 11109Bps$
- zápis – 1 transakce
- čtení – 2 transakce (psaní do adresového registru nějakou hodnotu + zápis slova)



Obrázek 18: PCF8570 paměť

Instrukce a architektury

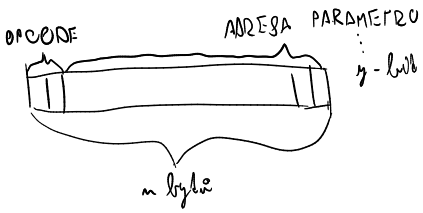
- **instrukce** – posloupnost n bytů
- **instrukční sada** (instruction set) – instrukce podporované daným CPU
- **strojový kód** – posloupnost instrukcí (machine code)
- **instruction pointer** (IP) – pozice aktuálně ukazované instrukce
  - zpravidla ukazuje na první bit (vícebitové) instrukce
  - je x-bitový (logicky stejně jako code memory)



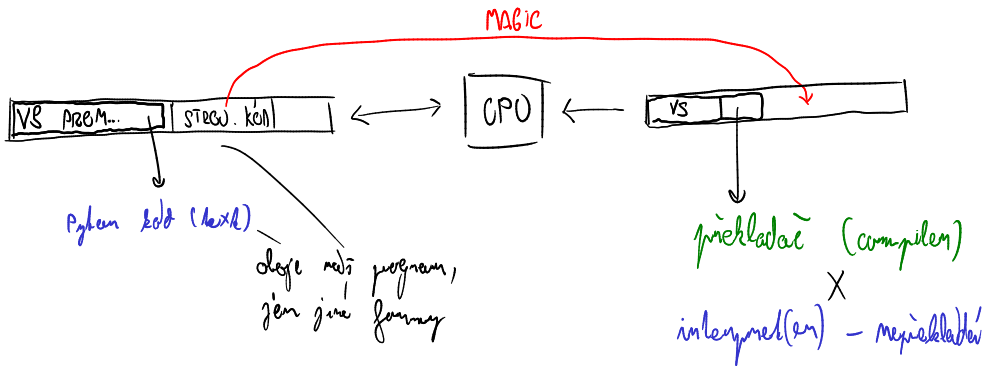
Obrázek 19: architektura s instrukcemi

- **opcode** (operation code) – typ instrukce
  - podle toho se interpretují argumenty

  - bývá $1B$ až $2B$
- **argumenty** – s čím instrukce pracuje – hodnota/adresa/…



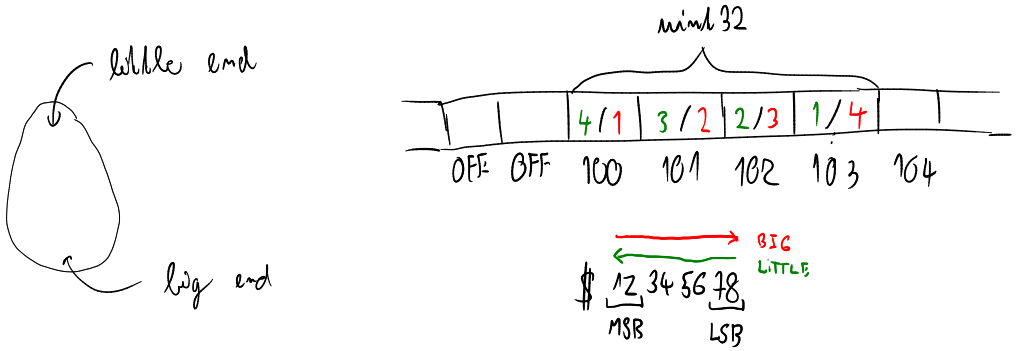
Obrázek 20: struktura instrukcí



Obrázek 21: jak to funguje doopravdy

Endianita

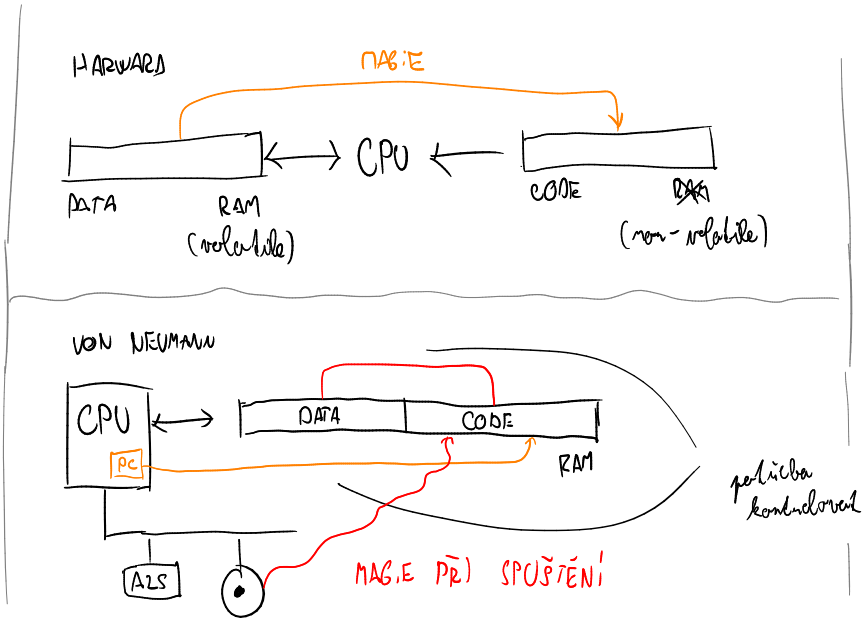
- pochází z Gulliverových cest podle skupin lidí, kteří jedli vejce z různých konců – little end(iáni), big end(iáni)
  - je to ve výsledku vlastně úplně jedno, jen je potřeba si něco zvolit
- většina procesorů je **little endian** (ten divný, obrácený způsob)
- pozor – endianita *bitů* a *bytů* může být rozdílná!



Obrázek 22: endianita

Harvard × von Neumann

	Harvard	von Neumann
paměť	data + kód jsou na *rozdílných sběrnicích*	data + kód jsou na *stejné sběrnici*
využití	mikročipy	počítače



Historie CPU architektur

- 6502 (Apple I – Wozniak)
  - 8-bit CPU
  - von Neumann
  - 16-bit addr space (64 kB operační paměti)
  - little endian
- Intel 8088
  - 16-bit CPU
  - 20-bit addr space (1 MB operační paměti)
  - little endian
  - převládlo – 20-bit address space byl velká výhoda

Assembler

- jelikož jsou instrukce pouze posloupnost hex cifer, tak se špatně čtou – řeší to **assembler**
  - skok je např. `JMP`, load je `LDA`, store je `STA`

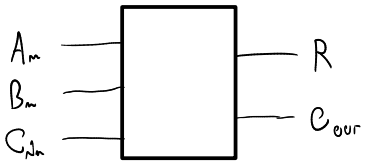
Příznakový registr CPU

- 1 příznak (flag) = 1 bit informace
  - **zero** – jestli poslední výsledek byla 0
  - **sign** – záporný výsledek operace
  - **carry** (přenos) – z nějakých operací (např. sčítání)
    - operace to definují různě (podle toho, co potřebují)
- `SET` a `CLR` instrukce umožňují explicitně upravovat hodnoty registrů

6502 – akumulátorová architektura

- `A` – registr, na kterém se provádějí všechny aritmetické operace
- podpora `AND`, `OR`, `XOR`, `SHR`, `SHL`, `ROL`, `ROR`
  - bacha… `NOT` nemá (jde ale přes `XOR` s jedničkami)
- pro vícebitové proměnné je potřeba rozdělit na části a řešit zvlášť
- aritmetika je trochu problematická – x86 ani 6502 neumějí ukládat na třetí adresu

Sčítání a odčítání



Obrázek 23: sčítací (odčítací) blackbox

- je potřeba explicitně nastavit `carry` příznak na 0 (`CLC` instrukce)
  - řetězení: `LDA CLC ADC STA | LDA ADC STA | ...`
- implementace odčítání pomocí sčítání
  1. přepsání na sčítání: `A - B -> A + (-B) -> A + NOT(B) + 1`
  2. subtract w/ borrow (**SBB**) – carry je borrow
     - 1 znamená, že si něco potřebuji půjčit z vyššího řádu
     - principiálně stejné jako normální sčítání
     - x86 to takhle dělá
     - problematické: v HW to není lehké implementovat
  3. subtract w/ carry (**SBC**) – carry je *not borrow*
     - pozor – nezapomenout na začátku pomocí `SEC` nastavit carry na 1!
     - snadněji se to HW implementuje
     - funguje na to hezký matematický trik:

```
A - X - B              // odečítání X a borrow od A
A - X - B + 256        // 256 je obecně n-bit
A - X - (1 - C) + 256  // použití carry flagu
A + (255 - X) + C
A + NOT(X) + C
```

x86

- více registrů = více svobody!
- 32*b* slovo + 32*b* address space
- obecné registry (7) – můžeme s nimi dělat, co chceme
  - v rámci zpětné podpory dovolují pracovat s prvními 8*b*, 16*b*, 9-16*b*…
  - lehčeji se řeší komplexní výrazy – `a + b + e - (c + d)`
    - u 6502 bychom museli pořád ukládat do akumulátoru (a dokonce ještě někam do paměti)
    - u x86 můžeme použít více registrů

Příklady instrukcí

- obecný tvar (assembleru): `OP target, source`
- `MOV EAX, [EXP]` – přesun z adresy na pozici hodnoty `EXP` do registru `EAX`
- `MOV [EAX], EXP` – přesun z `EXP` do paměti na adrese v `EAX`
- `ADD`/`ADC` – s/bez carry

Rychlost operací

- **GHz** – jednotka frekvence, ve které se měří rychlosti (moderních) procesorů
  - 1 takt… $1/x$
- dnes už lze za 1 takt zvládnout základní instrukce – logické, aritmetické…
  - bavíme se čistě o operacích *na registrech* – přístup do paměti je významně pomalejší
- také záleží na tom, zda nesčítáme $64b$ čísla na $32b$ architektuře
  - když ale sčítáme $16b$ na $32b$, tak je to také 1 takt

Čísla v Pythonu, vol. 2

- bloky paměti jsou vlastně $32b$ (ne $8b$), jelikož se očekává, že Python bude běžet (alespoň) na $32b$ architektuře
- kolik zabírá `x = 5` v Pythonu:
  - $+4B/8B$ (podle architektury), jelikož vše v Pythonu je objekt
  - $+4B$ samotného zápisu čísla
  - $+4B$ určující, kolik bitů je pro čísla využito
  - $+4B$ určující typ proměnné (číslo)
  - $+4B$ na **reference counting** – kolik proměnných ukazuje na hodnotu
    - je využíván garbage collectorem, aby vyhazoval z paměti to, co není používáno
  - musí se s tím provádět hrozně šaškáren, je to proto vážně pomalé (klidně až 100x pomalejší než jazyky jako C#)
- přičítání *vytvoří nový objekt:*
  - jelikož jsou `int`y immutable, tak `a = 300; a += 1` vytvoří úplně nový objekt
    - lze otestovat tím, že na `a` voláme `id(a)` před a po přičtení
  - trochu záchrana – čísla od -5 do 256 jsou optimalizována (uložena někde v tabulce), jelikož se s nimi dost často počítá
- má to výhodu – nemusíme řešit **arithmetic overflow** (přetečení) nebo **underflow** (podtečení)
- ještě pozor: reálně je v každém bloku uloženo jen $30b$ hodnot, jelikož poslední bit je chápán jako carry příznak (Python nevidí do procesoru)

Násobení a dělení

- trochu těžší instrukce – bývají pomalejší (je dobré s tím v algoritmech počítat):
  - násobení – 10 taktů
  - dělení – 10/100 taktů

operace	x86/x64	ARM (mobily)	μC	6502
*	ano	ano	možná	ne
//	ano	možná	ne	lmao

- je potřeba to na architekturách, které to nemají, softwarově implementovat
  - `SHL` je násobení 2 je `SHL`; dělení 2 je `SHR`
    - násobení a dělení jinými čísly lze rozdělit na posuny a součty (pokročilé, nebrali jsme; je na to ale úžasná knížka *Computer Systems: A Programmer's Perspective*)
  - pozor na signed čísla – `SHL` funguje jen pro menší čísla a `SHR` nefunguje vůbec (vytváří nuly)
    - je potřeba `SAR` (kopíruje MSb), ale pořád to není ono (`-5 // 2 = -3`)

Tomášovo odbočka (příklady instrukcí)

- v rámci přípravy na zkoušku je naprosto super si zkusit generovat z Cčkových zdrojáku assembler:
  - na Linuxu `gcc -g -c soubor.c; objdump -S soubor.o;` dělá přesně tohle
  - pozn.: není to Intel syntax – pro ten je třeba k `objdump` přidat `-d` a `--disassembler-options=intel`

```
#       int a = 5;
movl    $0x5,-0x14(%rbp)

#       int b = a + 5;
mov     -0x14(%rbp),%eax
add     $0x5,%eax
mov     %eax,-0x10(%rbp)

#       int c = ~a;
mov     -0x14(%rbp),%eax
not     %eax
mov     %eax,-0xc(%rbp)

#       int d = a * b;
mov     -0x14(%rbp),%eax
imul    -0x10(%rbp),%eax
mov     %eax,-0x8(%rbp)

#       int e = a - (b + c) - d;
mov     -0x10(%rbp),%edx
mov     -0xc(%rbp),%eax
add     %eax,%edx
mov     -0x14(%rbp),%eax
sub     %edx,%eax
sub     -0x8(%rbp),%eax
mov     %eax,-0x4(%rbp)
mov     $0x0,%eax

#       int f = a | (b & c) ^ d & e;
mov     -0x18(%rbp),%eax
and     -0x14(%rbp),%eax
mov     %eax,%edx
mov     -0x10(%rbp),%eax
and     -0xc(%rbp),%eax
xor     %edx,%eax
or      -0x1c(%rbp),%eax
mov     %eax,-0x8(%rbp)

#       int g = e << 10 + f >> 10;
mov     -0x8(%rbp),%eax
add     $0xa,%eax
mov     -0xc(%rbp),%edx
mov     %eax,%ecx
shl     %cl,%edx
mov     %edx,%eax
sar     $0xa,%eax
mov     %eax,-0x4(%rbp)
mov     $0x0,%eax

#       short g = f;
mov     -0x8(%rbp),%eax
mov     %ax,-0x1e(%rbp)
```

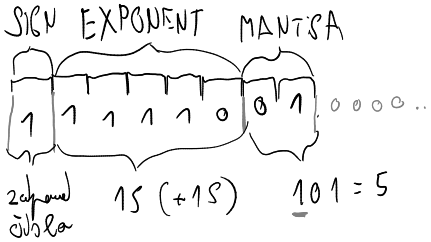
Reálná čísla

Fixed-point

- pevný počet bitů pro části před a za desetinou čárkou
- hezky na tom funguje aritmetika – můžeme normálně sčítat, odčítat, porovnávat…
  - výrazně rychlejší než floating-point (viz. dále)
- problém na operacích s hodně velkými a hodně malými čísly – přesnost…

Floating-point

- čísla v normalizovaném formátu: $\text{sign} \cdot 1.0110101 \cdot 2^{10001001}$



Obrázek 24: struktura floating-point čísla

- v mantise první 1 ignorujeme (je tam totiž v normalizovaném tvaru vždy)
- exponent je v *bias* reprezentaci: mapování $(-n, n) \mapsto (0, 2n + 1)$
  - převody jsou přičítání/odčítání biasu
  - hodí se (později uvidíme proč)
- SW implementace by byla pomalá – bývá to podporováno v CPU
  - floating-point registry
  - stejně pomalejší než celá čísla

x86/x64	ARM (mobily)	μC	6502
HW	HW/SW	SW	SW

IEEE 754

- standarta definující $32b$ a $64b$ floating point čísla
- pro $32b$ (float) je $1b/8b/23b$
- pro $64b$ (double) je $1b/11b/52b$
  - takhle je to ukládané Pythonu

Ošklivá čísla

- 0.1 nelze reprezentovat jako floating-point číslo (nekonečný dvojkový zápis)
  - programovací jazyky zaokrouhlují – *nikdy floating point čísla neporovnávat,* používat $abs(a - b) < \varepsilon$

Speciální hodnoty

- pro nulový exponent je číslo v denormalizovaném tvaru (pro reprezentaci fakt hodně malých čísel)
- 0 jsou samé nuly (až na znaménko)
  - proto jsme volili reprezentaci s biasem
  - znaménko znamená, že máme 2 nuly, standard definuje oboje jako to samé
- samé 1 v exponentu nabývá podle znaménka a mantisy speciálních hodnot (∞, nan…):
  - $\infty/2 = \infty$; $\infty + 2 = \infty$
  - $\infty/\infty =$ nan (not a number)
    - cokoliv + nan je nan
    - existuje více typů nan (podle vzniku)
  - $1.0/0$ bývá v normálních jazycích ∞, Python ale kontroluje dělení nulou a hodí chybu

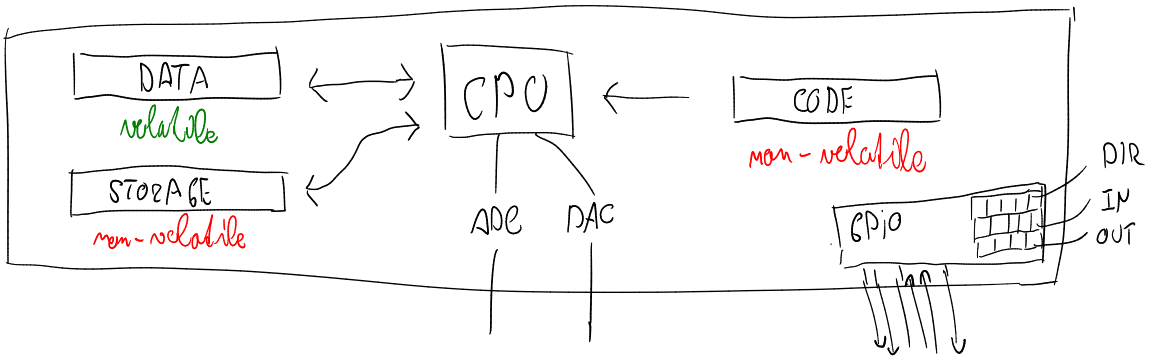
Paměti ROM

- jsou *non-volatile* – hodnoty přežijí vypnutí počítače

typ	write	read
ROM (Read Only Memory)	1 (výrobce)	∞

typ	write	read
PROM (Programable ROM)	1 (vypálení)	∞
EPROM (Erasable PROM)	„∞“	∞
EEPROM (Electrically EPROM)	„∞“	∞

- EPROM – problém s mazáním (dělá se to s UV zářením… nepraktické)
- EEPROM – všechno tím nahradit nechceme, je pomalejší než RAM
  - také jim lze říkat NVRAM (non-volatile RAM)
  - omezený počet writů kvůli tomu, že fyzikálně se elektrony připojí do obalu atomů a nejdou pak moc dobře vytlačit
  - adresovatelné po individuálních bytech
  - **flash** – jiná výrobní technologie
    - dokáže zapsat/vracet velké bloky – $1/10kB$
    - rychlejší na přístup k většímu počtu dat, pomalejší na random přístup



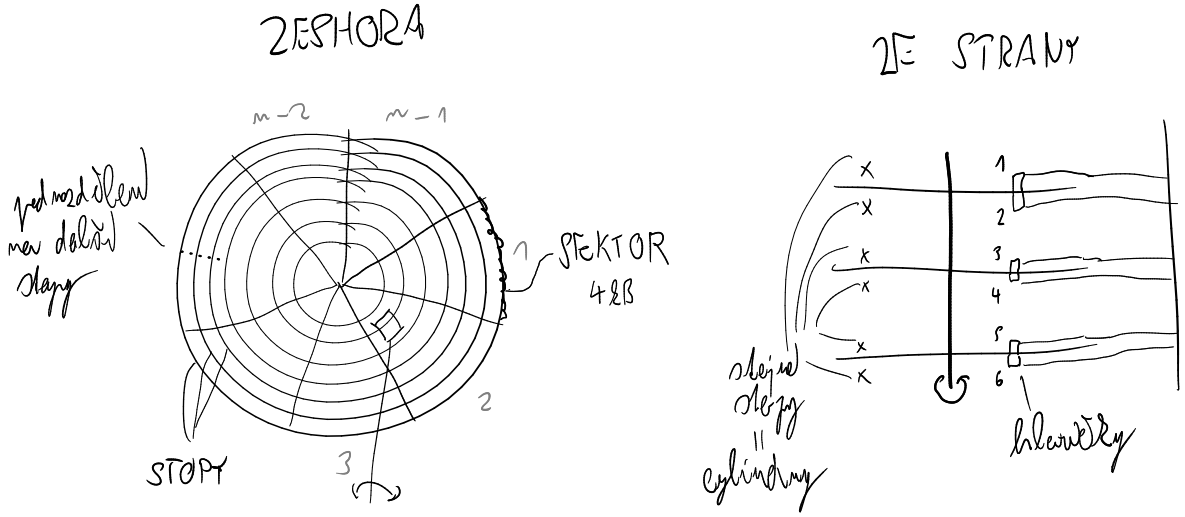
Obrázek 25: Harvard počítač (s GPIO)

- **GPIO** = General Purpose Input and Output
  - slouží jak pro vstup, tak pro výstup
  - `DIR` registry určuje směr pinů, `IN` a `OUT` jsou hodnoty na vstupu/výstupu

Permanentní datové úložiště

HDD

- jsou **magnetické**
- **sektor** – dnes $4kB$ (dříve $512B$)
- **hlavičky** – pohybují se všechny najednou
- disk se otáčí, hlavičky se hýbají do strany – podle toho přístup k bytům
- adresa hodnoty je trojice **CHS** (cylinder, head, sector)
- přístup vně je lepší – rychlost sektorů dále od středu je větší
  - zaplňují se od vnějších po vnitřní



Obrázek 26: HDD

Výhody

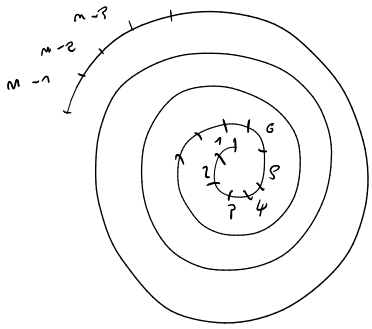
- levnější než alternativy
- velké množství dat

Nevýhody

- náchylné na poškození
- sekvenční přístup je fajn (disk se otáčí), obráceny je příšerný
- docela pomalé… $10ms$ sekvenční / $0.5MBs$ obrácený sekvenční

CD / DVD / BLURAY

- jsou **optické** – pokud se světlo odrazí tak 1; jinak 0
- nejsou optimální pro archivační účely – vrací se do svého původního stavu
- oproti pevným diskům jsou data ukládána do **spirály** (stejně jako gramofonová deska)



Obrázek 27: CD

- používají **LBA** – linear block addressing (není to už trojice – lehčí na programování)
- přenosová rychlost je menší ($10MBps$), přístupová také ($100ms$)

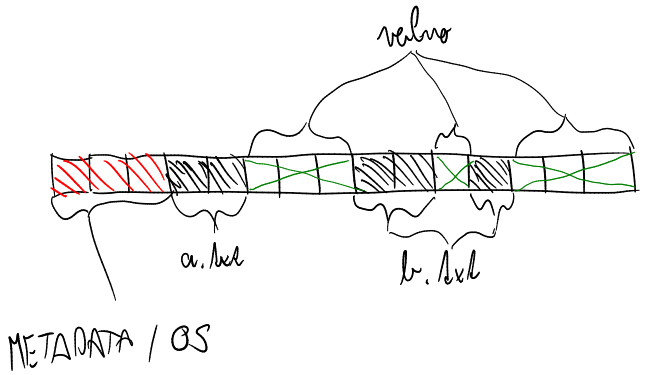
Řadiče pro úložiště

- registry:
  - adresový

  - příkazový
  - **buffer** (pro postupné ukládání dat pro čtení/zápis)
  - info (počet sektorů, velikost sektoru…)
- bylo by nepraktické používat pro každé zařízení jiný protokol – pro všechny se tedy používá **LBA**, jen tam jsou vždy pro příslušná zařízení převody:
  - pro HDD dochází k mapování LBA $\iff$ CHS
  - takhle připojený flash disk je vlastně SSD (Solid State Drive)

Adresování, soubory

- **offset** (v B) – posun od začátku paměti
- **base address** – odkud začínáme (čteme/píšeme/…)



Obrázek 28: struktura paměti

- **metadata** – *data o datech*; ukládá:
  - čísla sektorů, kde se soubor nachází
    - princip *fragmentace* – rozdělení souborů do více sektorů; nechtěné (přístup je pomalejší)
  - jméno souboru
  - velikost
  - obecně: volné sektory
- **OS** – abstrakce nad disky
  - stejné API pro čtení, psaní, práce s metadaty…
  - používají všechny programy – `open()` volá (*C*-čkovou funkci, která volá) systémovou funkci

V Pythonu

Soubory

- otevření: `with open("<path>", "<mode>", encoding="<encoding>") as f:`
  - `path` je cesta k souboru
  - `mode` je podle toho, co chceme dělat (v tomhle kurzu ale většinou chceme `rb`):
    - `r` – čtení
    - `w` – psaní
    - `b` – je to binární soubor
  - když nespecifikujeme `encoding`, tak použije kódování typické pro daný OS
    - Windows – Windows-1250 (`encoding="windows-1250"`)
    - Linux – (asi) UTF-8 (`encoding="utf-8"`)
    - funguje také např. ASCII (`encoding="ascii"`)
- `f.readline()` – čtení jednoho řádku
- `f.read(<bytes>)` – čtení daného počtu bytů; volání bez parametru přečte vše
  - může vrátit méně, když toho tam tolik není
  - pozor! u `b` módu vrací `bytes`, jinak `str`

- `f.seek(pos)` – nastavení offsetu od začátku souboru (v bytech)
  - čtení ho mění
  - `f.seek(pos, mode)` definuje způsob posunu
    * 0: posun od začátku souboru (default)
    * 1: posun od aktuální pozice
    * 2: posun od konce souboru

Byty

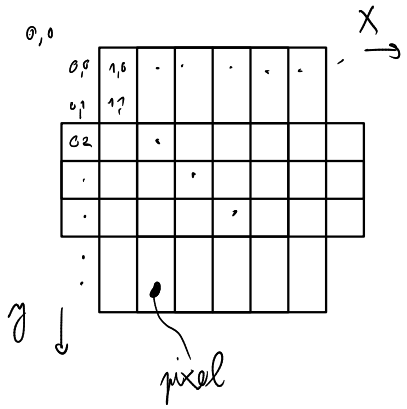
- `<str>.encode("<encoding>")` – `str -> bytes`
- `<bytes>.decode("<encoding>")` – `bytes -> str`
- bytes je **immutable** (nelze měnit) – je potřeba použít bytearray (`bytearray(<size>)`)
  - lze převádět z `bytes` vcelku přímočaře: `bytearray(<bytes object>)`

Šestnáctkový výpis

- hex view(er) – vypsání souborů jako šestnáctkové byty
  - na Linuxu:
    * `xxd <soubor>`; obrácený převod `xxd -r <soubor>`
    * `hexdump <soubor> -C`
- pozor – $1B = 2$ znaky
- 3 sloupečky (tradičně po 16 znacích – dobře se počítá pozice):
  1. hex offset (pozice v souboru)
  2. hex zápis (`AF 1F 3C 14 ...`)
  3. pokus o interpretaci dat jako text (nezná encoding…)

Reprezentace obrazu

- **bitmapy** – mapy bitů
- rozdělíme na **pixely** – na každém bude informace o tom, „jaké je tam světlo“



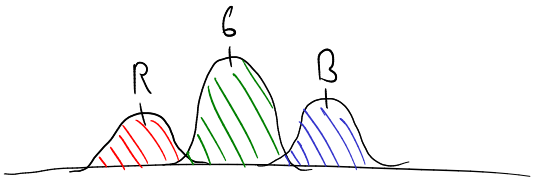
Obrázek 29: obrázek v počítači

- indexováno (x, y)
  - pozor – y jdoucí dolů *stoupá*, neklesá
- v paměti bývá uloženo po řádcích

Pixel

- co sem uložit? fotonů máme od 0 do ∞
  - je třeba (dobře) stanovit meze (analog $\rightarrow$ digital) – mapování intenzit

- **bit depth** (bpp = bitová hloubka) – kolik dat máme na pixel:
  - $1b$ – černobílá
  - $8b$ – odstíny šedi
  - floating pointy – větší rozsah (HDR)
    - problémy: lidské oko to neumí dobře zpracovávat a foťáky to neumí dobře fotit
- není foton jako foton: frekvence určuje barvu
  - vnímáme malé spektrum (viditelně světlo)
  - tyčinky (rozsah) x čípky (frekvence – 3 barvy)



Obrázek 30: čípky v oku

- barevná bit depth:
  - $3b$ na pixel – dost neúsporné (jen jestli je červelá/zelená/modrá) a blbě se rozděluje
  - $4b$ na pixel – poslední je intenzita (1 násobí vše 2x)
  - $2B$ na pixel – 5R/5G/5B/1 nic
    - někdy ten 1 bývá v zelené složce (oko je na ni citlivější)
  - $3B$ na pixel (true color; 65535 barev)
    - je to ošklivě nesoudělné; ukládá se většinou do $32b$… co se zbylými $8b$?
      - plýtvat
      - vytvořit na alpha kanál určující (ne)průhlednost pixelu (255 je neprůhledné, 0 transparentní)

Ukládání do paměti

- nestačí je jen uložit za sebe – potřebujeme metadata (obrázku, ne souboru), jako např.:
  1. bitová hloubka
  2. výška, šířka
  3. pořadí barev (endianita) – `RGB(A)` / `(A)BGR` (populárnější)
  4. offset, na kterém začínají data
- obecně metadata (hlavička) bývají na začátku souboru, abychom ji přečetli první

BMP

- původně Windows formát
- relativně jednoduchý
- data jsou little endian (jak čísla, tak barvy)
- první $2B$ jsou **magic** (signature) znaky
  - měly by být unikátní pro typy souborů
  - dány autorem
  - zajímavost – exe má `5a 4d`, což je v ASCII `MZ` – iniciály Marka Zbikowskiho (autor)

Reprezentace textu

- **string** – posloupnost znaků
  - písmena (abcdegh)
  - číslice (123456789)
  - symboly (@#$%^&*)
  - whitespace (mezera, tabulátor)
- **grafém** – nejmenší jednotka psaného jazyka
- **kódování** (většinou pro celý text) zavádí:
  1. 1 znak $\iff$ 1 kód (číslo)

2. kód $\iff$ binární reprezentace
   - zda bude 1 kód = $1B$, $2B$, proměnlivý…
- ukládaní do paměti tak, jak by se text četl (latinka levo-pravo, arabština pravo-levo…)
- historicky *nemívá metadata* – problematické (určovaní kódovaní)

ASCII

- *American Standard Code for Information Interchange*
- standardizování v 80. letech
- $7b$ kódování (0-127)
- číslice i písmena jsou v kódování blízko sebe – lze je dobře vyčíst, převádět…
- **extended** – rozšíření
  - každá část Evropy (W/M/E) si to rozšířila jinak
    - ISO8859-2 (Latin 2) – snaha o standardizaci, ale trochu pozdě
    - Win1250 – Windowsové kódování

Unicode

- standardizace – všechny jazyky, všechny symboly, ~~žádné problémy~~
- 0-127 – odpovídají kódům z ASCII
- 128-$\$FFFF$ – běžně znaky
- problém – neurčili binární reprezentaci, takže vznikly různé:
  - **UTF-32** – každý znak je $4B$
    - 2 verze UTF32 LE a UTF32 BE… guláš
    - paměťově ne moc příjemné
  - **UCS-2** – jednoduché (a debilní)
    - podporuje pevné $2B$… dokážeme reprezentovat pouze znaky v téhle mezi
  - **UTF-16** - proměnlivá délka znaku ($2B/4B$)
    - 4B… **surrogates** (náhradníci): pro určité hodnoty prvních $2B$ musí být přečteny druhé $2B$
      - výsledek se dohromady skládá magií
    - nelze přesně říct, kolik znaků je v souboru s tímhle kódováním uloženo
    - také varianty LE a BE
  - **UTF-8** – $1B$, $2B$, $3B$, $4B$ znaky
    - $1B$ – první bit je 0
    - nB – prvních n bitů je 1, ten za tím 0 (pro $n = 2$ je to 110)
      - každý další byt začíná 10 – lehce lze zjistit, že jsou součástí nějakého znaku
    - neřeší se endianita – jsou to prostě velká čísla
    - populární na internetu

Rasterizace

- string $\mapsto$ bitmapa
- potřebujeme rozeznávat, kdy skočit na další řádek… každý systém to řeší jinak:
  - původně `CR` (carriage return) + `LF` (line feed) (z psacích strojů)
    - zachovalo se pro Windows
  - Unix si zvolil `LF`
  - Mac OS si zvolil `CR` (moderní ale už používají `LF`)
  - Unicode – 2 nové znaky… `LS` (line separator) a `PS` (paragraph separator)
    - naprosto vůbec se to nechytlo, je v tom ještě větší guláš

Dokončení rozdělané magie

- **bridge** – řadič, který hostu zpřístupňuje jinou sběrnici
- **memory controller** – middle man mezi pamětí a CPU

  - řeší refresh u DRAM
  - maskuje to, že máme 1GB a 512MB paměť – linearizuje to pro CPU

Systémová sběrnice

- **PCIe**
  - sériová
  - jsou na tom všechna zařízení; na packet reagují jen ta, pro která je určený
  - 2 dedikované druhy packetů (memory **write**, memory **read**)
    - jen memory controller reaguje na tenhle packet
    - není tam adresa zařízení, ale paměti
  - příklad:
    1. CPU pošle **MRd** (Memory Read packet)
       - cílová adresa je adresa paměti
       - musí tam být uložena i adresa procesoru, aby mohl přijít packet zpět
    2. memory controller vykoná požadavek, pošle **CpID** (Completion with data)
       - cílová adresa je adresa procesoru (aby to došlo správnému procesoru)

Memory/mapped I/O [wiki]

- princip používání stejného adresového prostoru jak pro paměť, tak pro I/O
  - pro I/O jsou mapovány nějaké části paměti, které jsou volné
  - je potřeba, aby adresy byly unikátní

Co dělat po startu?

- **CPU startup vector** – odtud procesor začíná vykonávat instrukce
  - hardkódované v CPU (např. $0xFFFFFFF0$)
    - bývá tam v paměti skok někam, kde se instrukcí vejde více
- **firmware ROM** – paměť (non-volatile), kde je uložený program, který po startu dělá:
  1. test a konfigurace HW
     - mapování (nekonfliktních) adres pro zařízení
  2. hledání užitečného softwaru (**bootování**)
     - další rom – **option ROM**
       - mívaly starší systémy
       - bootuje instantně (je to ROM…)
       - princip cartrigových her – instantní spuštění
       - bývá např. u grafických karet – počáteční nastavení
     - **pevný disk** – mívá boot sector, který je uzpůsobený k načtení do paměti a spuštění
       - je tu tzv. **bootloader** – menší prográmek, který hledá na disku zbylá data
         - dnes většinou načítá **kernel** – převezme základní funkce FS, načte zbytek OS…
  3. implementuje funkce pro bootování – načti sektor, znak z klávesnice, vykresli něco…