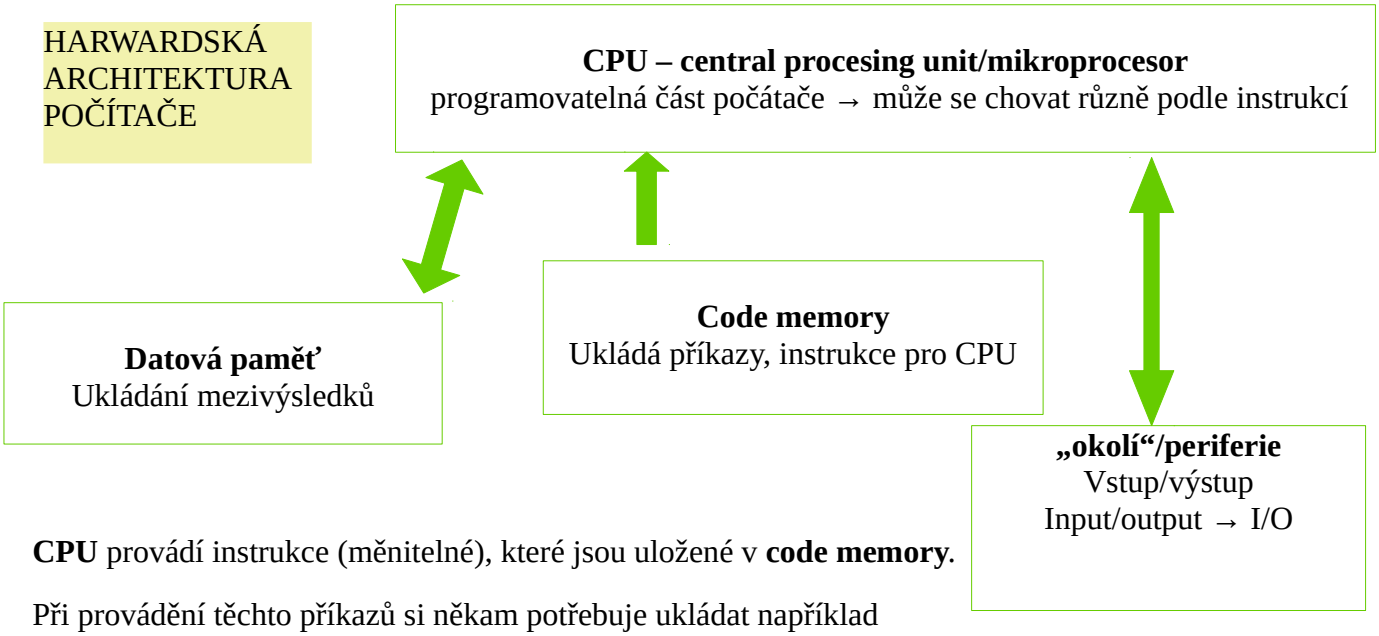
Co je to počítač ?

Počítač je stroj, který nemá pevně dané funkce → můžeme vytvářet různé programy, případně pro různá vstupní data se chová různě.



CPU provádí instrukce (měnitelné), které jsou uložené v **code memory**.

Při provádění těchto příkazů si někam potřebuje ukládat například

mezivýsledky výpočtů a poté je i číst.K tomu slouží **datová paměť**, která je jak pro zápis, tak pro čtení (zatímco z code memory pouze čte). Vstupní data, se kterými poté pracuje podle instrukcí, získává prostřednictvím periferie (monitor, klávesnice…) a výsledky tam opět odesílá → IO tedy slouží ke komunikaci a interakci s uživatelem.

CHARLES BABBAGE – Vytvořil kolem roku 1837 „analytical engine“, což byl návrh stroje, který v reálu opravdu mohl fungovat a byl programovatelný. Ke konstrukci však nikdy nedošlo, protože byla s tehdejší známou úrovní technické výroby velice nákladná a obtížná (stroj by zabíral velikost fotbalového stadionu), sestrojit by ale šel a byl by plně funkční (ovšem velice pomalý i pro běžné výpočty). Byl také „turing complete“ (počítač ve smyslu definovaném Alanem Turingem).

ADA LOVELACE - Dcera romatického básnika lorda Byrona. Splupracovala s Babbagem a prý publikovala první program na světě (spíše jen opravila Babbageův návrh). Podstatnějším přínosem bylo, že si uvědomila, že pokud umí stroj pracovat s čísly, umí pracovat i s dalšími věcmi (text, zvuk, obraz), neboť vsechno lze na akceptovatelný číselný formát převést.

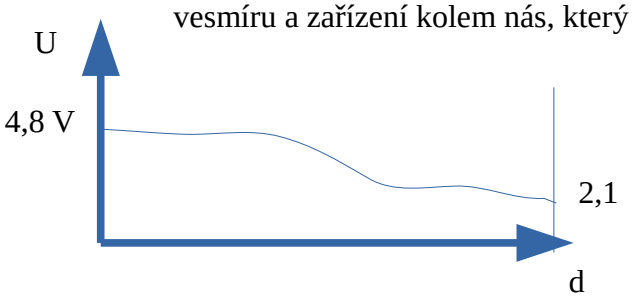
Jak přenášet data ?

ANALOGOVÝ PŘENOS

Předpokládejme, že komunikujeme pouze čísly. Nejjednoduší je spojení kabelem, po kterém přenášíme **napětí odpovídající číslu**, které chceme poslat.

ALE!

- Kde vezmeme to přesné napětí ?
- Co velká čísla ? (nutnost příliš velkého napětí nebo úmluvy například o vydělení deseti)
- ztráta dat během přenosu vlivem – teploty, vzdálenosti, šumu (statický šum pocházející z vesmíru a zařízení kolem nás, který je vidět například na nenaladěné televizi)



ŘEŠENÍ

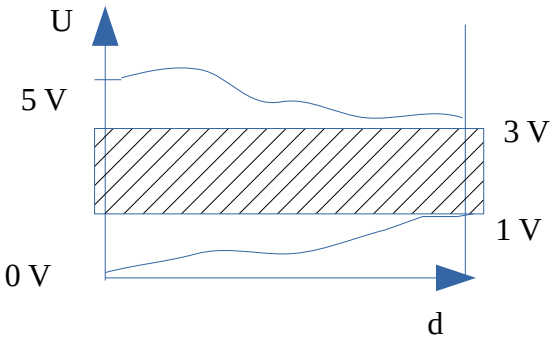
Když snížíme počet hodnot, zvýšíme přesnost přenosu → **dvojková soustava** → digitální (číslicový přenos)

1 bit (binary digit) – jedna číslice (buď 0 nebo 1)

ČÍSLICOVÝ PŘENOS

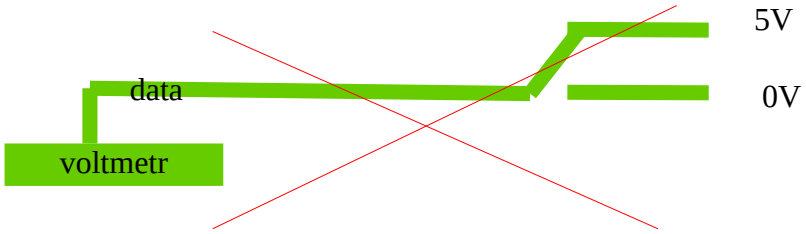
Dohodneme se na hodnotě napětí pro 1 (například 5V) a 0 (0V). Můžeme zvolit i jiné napětí (například 5V a -5V). Ačkoliv tedy původní napětí může klesnout nebo stoupnut v určitém rozpětí, stanovíme interval hodnot, pro které signál vyhodnotíme jako 1 nebo 0. Viz obrázek č.2

obrázek č. 2



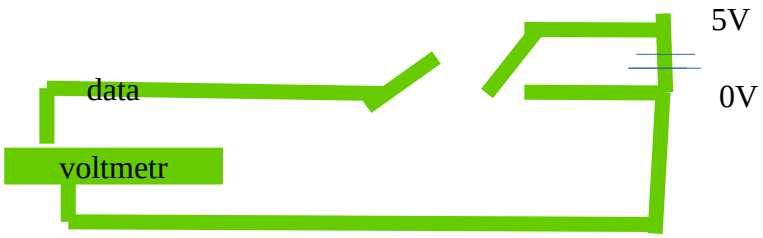
„šedá zóna“ = chyba

Pokud se hodnota trefí do šedé zóny, je to „nějaký divný“ :D



ALE! Jelikož napětí definujeme jako rozdíl potencilů dvou bodů, potřebujeme kromě voltmetru ještě tzv. referenční potenciály spojené se zemí (GND = GrouND), které budou stejné na obou koncích spojení

ŘEŠENÍ: propojíme je dvěma kabely (dráty), jeden přenáší napětí a druhý spojuje ta referenční místa.



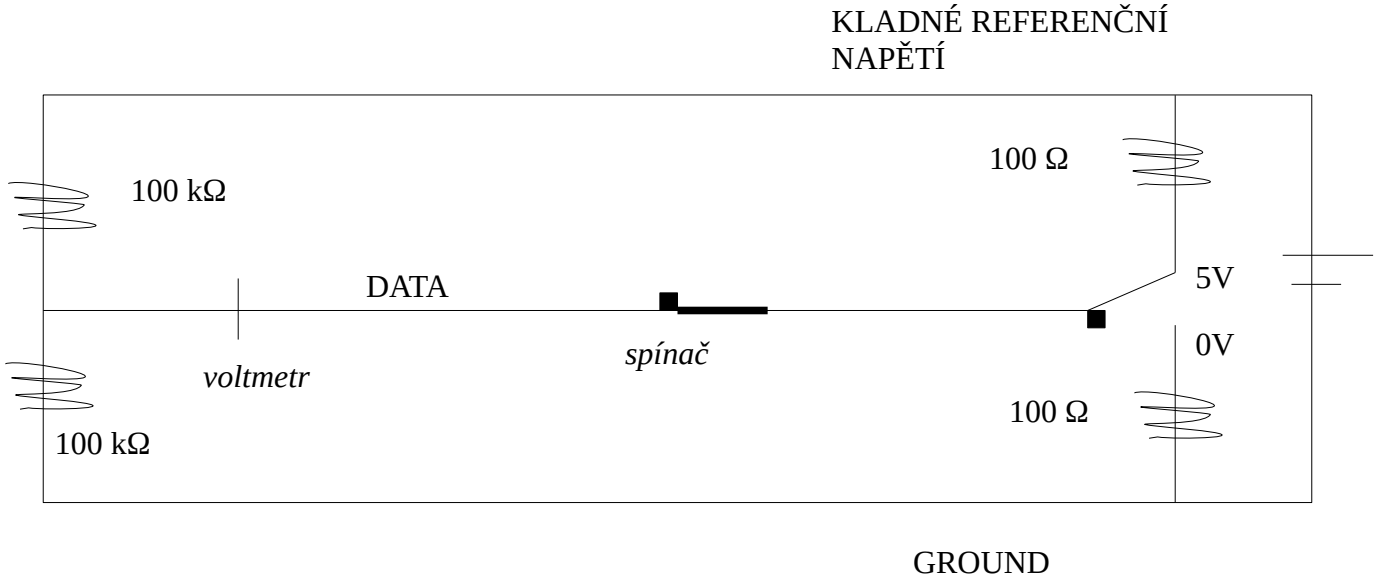
TŘÍSTAVOVÁ LOGIKA

Kromě toho se obvod může nacházet ve stavu, kde žádná informace neprobíhá (obvod je rozpojený). Tomuto stavu se říká **floating stav/stav vysoké impedance.** Potom mluvíme o třístavové logice = rozlišujeme stav 0, 1 nebo neprobíhá signál.

ALE!

Jak poznáme , že v obvodu neběží data ? Podívejme se na následující schéma (obrázek č. 3)

obrázek č. 3



Kromě referenčního kabelu (označen GROUND) si ještě přidáme propojení se zdrojem (kladné referenční napětí). Předpokládáme, že obvod má nějaký odpor, který je na obrázku rozdělen do větvě pro 0V a do větve pro 5 V (oba odpory velikosti 100 Ω. Na druhé straně přidáme dva velké odpory o síle třeba 1000 Ω.

Pokud to tedy zapojíme jako na obrázku, máme dva paralelně zapojené odpory – jeden velmi malý a druhý velmi velký, čili se celkově rovnají téměř odporu třetímu a napětí je zhruba 5V. Podobně při přepojení.

ŘEŠENÍ:

Pokud obvod rozpojíme, tak nám běží proud po drátu kladného referenčního napětí a probíhá dvěma velkými odpory. Napětí mezi dvěma sériově zapojenými odpory je poloviční, tedy asi 2,5 V a takto detekujeme floating stav. Takto můžeme také šedou zňu z obrázku č. 2 chápat jako floating stav. Toto ovšem není jediný způsob řešení

DIFERENCIÁLNÍ PŘENOS

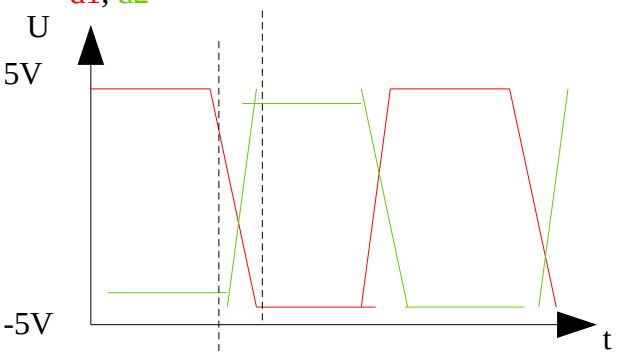
Vyplývá ze snahy ještě více eliminovat chyby a šumy v přenosu. Nepoužíváme jeden ale dva datové vodiče. Jeden se základní informací a druhý s přesně opačnou (kde je 0, dáme 1 a naopak).

Potom zjišťujeme hodnotu bitu jako **relativní rozdíl dvou napětí.**

Pokud $d_1 - d_2 > \varepsilon \rightarrow 1$

$d_1 - d_2 < \varepsilon \rightarrow 0$

obrázek č. 4 :



ALE!

1) bit-order (pořadí bitů) 1011 nebo 1101 ?

2) jak rozeznáme více stejných hodnot za sebou ?

ad 1) :

LSB (least significant bit)

MSB (most significant bit)

Dohodneme se zda přenos bude MSB-first nebo LSB-fist

RS 232
Používá sev průmyslových zařízeních, pokladnách nebo pro připojení periferií (myš)

ŘEŠENÍ ad 2)

1) rozhodneme se o rychlosti/délce bitu (baud rate = počet informací za sekundu). Například 1200bits/s → 1 bit ≐ 0,83 ms. Hodnotu sledujeme cca uprostřed toho času, kdy je nejstabilnější (viz obrázek č. 4).
    - na začátku přenosu nevíme, od kdy přesně máme počítat -přenosový kanál je standartně v nějakém **idle stavu** – přenos neprobíhá. Potřebujeme **start-conditions a stop-conditions.** Příkladem start-condition je start-bit. Pokud nechci využívat floating stavy, tak se dohodnu, že na začátku přenosu pošlu jeden start bit = 1 a od poloviny toho času začnu počítat 0,83 ms.

ČASOVÁNÍ SE OVŠEM PO ČASE ROZEJDE

Kvůli tomu se data rozsekají po konstatntních délkách (například 1 B = 1 Byte = 8 bitů) – tím navíc vyřešíme i problém s rozpoznáním přechodu do idle-stavu.

ale potom se posílá ZBYTEČNĚ MOC INFORMACÍ.

2) Dva vodiče – druhý vodič vysílá „**hodinový signál“** (CLOCK, CLK) 1 = úůatné, 0 = neplatné a my zjištujeme hodnotu jen pokud hodinový signál je 1 (respektive koukáme na hrany – když se signál mění z nuly na jedničku).

   Nyní není třeba omezovat délku přenášené informace. Navíc je možné mít různě dlouhé bity.

Posílající a přijímací se musí dohodnout na komunikačním protokolu – kolik bitů mají která čísla.

1110111000010101011001 0 - nevím, kfe to mám rozsekat

Co už umíme ?

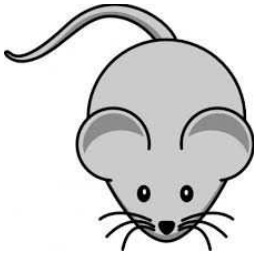
- Základní priincip stavby počítače
- přenos dat
  - posloupnost čísel, seriový přenos – sychronizace hodin (RS – 232)
  - je třeba dohodnout se na **komunikačním protokolu** – kolik a jak velkých čísel se bude posílat a co znamenají

Packet – zde ve smyslu balíčku informací, které patří k sobě (například rozdělené do více bajtů...)

Rozsah čísel

Pokud máme například 7-bitové číslo, kde jednotlivé 0 a 1 reprezentují, zda se přičítá daná mocnina dvojky, můžeme získat rozsah od 0 (0000000) do 127 (11111111), tedy obecně pro n-bitové číslo od 0 do 2^{n-1}

$\overline{\text{INF}}$ – znamená, že 0 a 1 interpretujeme obráceně (0=true, 1=false)

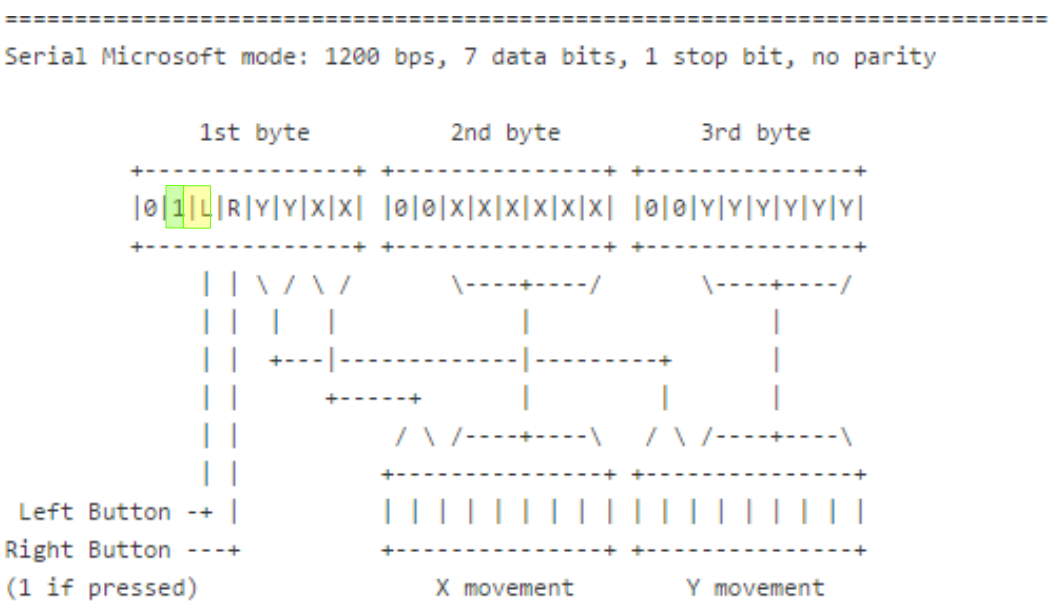


Protokol počítačové myši

RS-232 funguje právě na principu pevně dané délky informace , nikoli na základě hodinového signálu.

Typickým příkladem přenosu po lince RS-232 je například počítačová myš. Její protokol nám přesně určuje kolik bajtů se posílá a co které bity v nich reprezentují.

Kdykoliv se myš pohne, pošle nám 4 B informací :



Zkouška práce myši:
nic není stisknuté = 64
obě tlačítka = 112 = 64 + 32 + 16
levé tlačítko = 96 = 64 + 32

Jak vidíme, odpovídá to protokolu. Pokud není nic stisknuté, je šestý bit (počítáné zprava a od nuly) roven 1 ($64 = 2^6$)
Levé tlačítko – 64 opět (jednička je tam vždy) + 2^5 a pátý bit odpovídá stiknutému levému tlačítku
obě dvě – přibude ještě 2^4 což odpovídá tomu, že přibylo pravé tlačítko.

Z pozorování vidíme, že to sice opravdu funguje a myš dodržuje protokol :D

ALE! v případě větších čísel budem jen těžko rozhodovat, na místě kterých bitů je 0 či 1. Je třeba tedy zvolit jiný způsob zápisu – jednou alternativou je rovnou binární, ten ale na velká čísla zabere příliš mnoho bitů.

ŘEŠENÍM je **šesnáctková (hexadecimální) soustava.** Funguje jako jiné poziční soustavy (desítková či dvojková) :

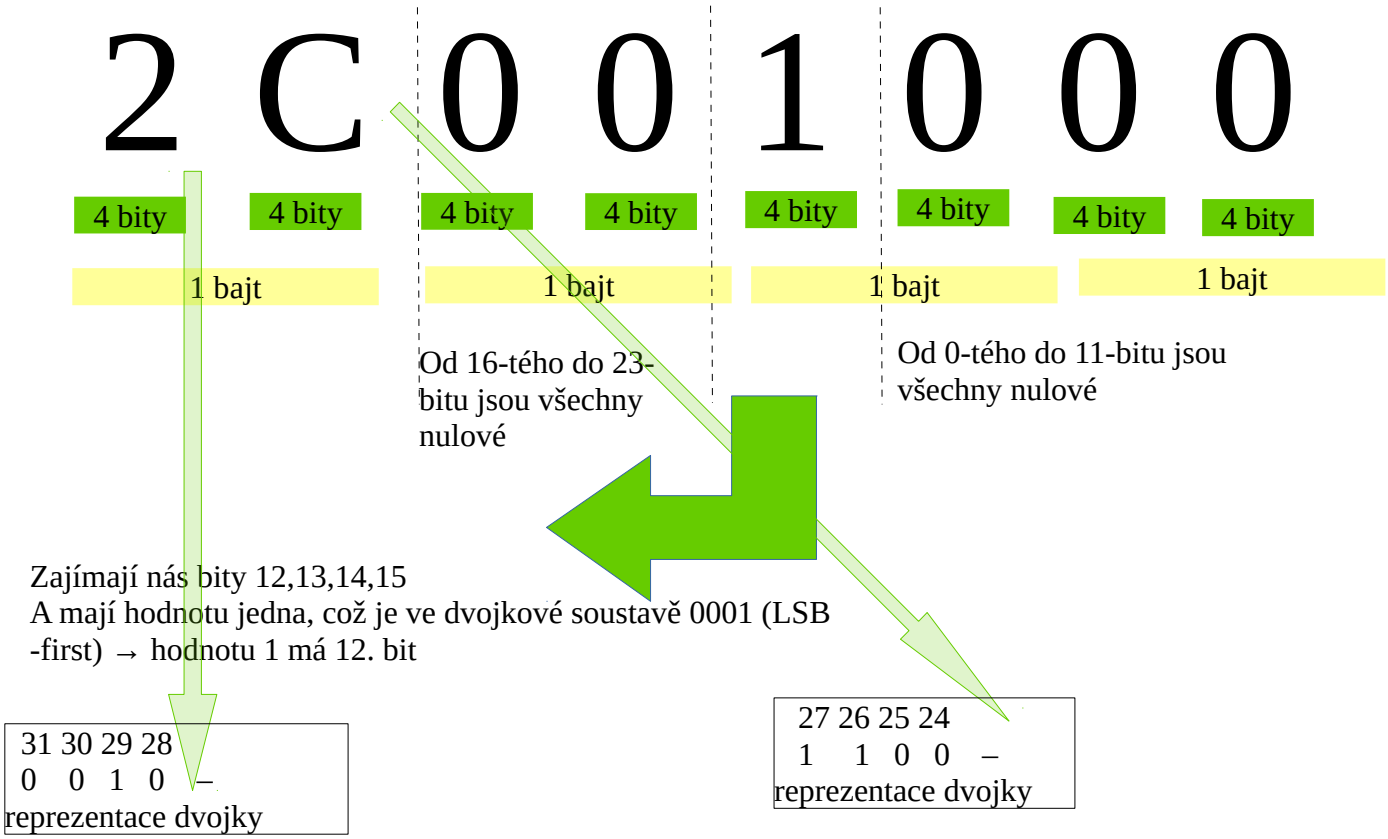
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Pomocí hodnot v prvním řádku budem zapisovat čísla a odpovídají hodnotám v druhém řádku.
Jak interpretujeme „FF“ ?
F = 15, je to tedy $15*16^0 + 15*16^1 = 255$

Jak poznám, že je to v šesnáctkové soustavě ?
$FF (v pascalu) …...přidám dolar
0xFF (C/C#)……. přidám 0x
FFh…….přidám h

Kolik mi to zabere paměti ?
Mám číslice v rozsahu 0 – 15. Horní hranice odpovídá 4 bitům (protože $2^4 = 16$ a už víme, že rozsah číslic je tedy 2^{n-1}) → na zapsání každé číslice této soustavy potřebujeme 4 bity. Číslo FF je tedy 8 bitů = 1 B a víme také že 1B nám stačí na čísla do 255, což je v souladu s naším výpočtem výše :)

Jak nám to pomůže ?
Mějme číslo 738201600 v desítkové soustavě. Z něj toho moc o hodnotě jednotlivých bitů nevidíme. Rozklad čísel 0 – 15 do dvojkové se pamatuje snáze. V šesnáctkové soustavě vypadá toto číslo takto:



BITOVÉ OPERACE

Fungují mezi dvěma operandy a fungují na jednotlivých bitech nezávisle na na sobě. (první bit prvního čísla (operace) první bit druhého čísla = první bit výsledku).

Tyto operace můžeme brát buď jako logické operace (z matematiky) nebo jako posloupnost VSTUP → PŘÍKAZ(MASKA) → VÝSTUP

OR

SET = NASTAVENÍ

funguje stejně jako disjunkce
(1 pokud alespoň jeden z operandů je 1)

1100 VSTUP
1010 PŘÍKAZ
1110 VÝSTUP

If 0 opiš bit ze vstupu
If 1 napiš 1

AND

CLEAR

funguje stejně jako konjunkce
(1 pokud jsou oba 1)

1100
1010
1000

If 0 napiš 0
If 1 opiš vstup

NOT

FLIP = PROHOZENÍ

funguje stejně jako negace, unární operátor
(zamění hodnotů bitů)

1100
0011

XOR

SELECTION FLIP

Selektivní negace

1100
1010
0110

If 0 opiš ze vstupu
If 1 prohoď hodnotu vstupu

K čemu je to dobré ?
Sice už se díky šesnáctkové soustavě vyznáme v číslech na výstupu lépe, ale chtěli bychom rovnou vidět, které tlačítko se stisklo (vypsat pomlčku pokud není a L/R/M pokud je některé stisknuté).

Aplikujeme operaci AND a jako masku zvolíme číslo, které všechny bity kromě toho, co nás zajímá, změní v nuly ($20):

??**?**????? 
00100000

Nyní položíme výsledek roven nule:
Pokud je i náš bit roven nule, je celé číslo nulové, pokud ne, tak se nerovná nule a vypíšeme, že je stisknuté.

SHR

Shift to right
Posouvá od vyšších řádů k nižším (pokud to máme napsané v řádku MSB-first tak doprava) a doplní nuly
Analogicky funguje SHL (shift to left)

1011 SHR 2 → 0010**11**

ROR

Rotate right
Posune stejně jako SHR ale přetečené číslice nevymaže, vrátí na začátek čísla

1011 ROR 2 → 11**10**11 

K čemu je to dobré ?
Snažíme se podle protokolu myši získat x-ové a y-ové souřadnice. Potřebujeme k tomu poslední dva bity prvního čísla a posledních šest z druhého čísla. Pokud bychom měli všechny ostatí pozice bitů vynulované stačilo by aplikovat OR a měli bychom výslednou souřadnici. Použijeme tedy na první číslo SHR 6. Pak už jen aplikujeme OR a máme výsledné číslo

000000??
00??????

ALE!

Musíme vdy určit o kolik a být si jisti přesností čísla. Pokud by bylo číslo třeba šesnáctibitové, tak se nám sice posune, ale na některých mistech „vlevo“ nebudou jen nuly.

ŘEŠENÍ

Abychom tento případ ošetřili, je dobré předtím provést AND s nějakou vhodnou maskou, která nám opravdu všechny pozice vynuluje. V tomto případě $03. Druhé číslo má sice podle protokolu první dva bity nulové, ale nevíme, jestli tomu tak bude vždy a proto provedem na druhém čísle AND s číslem $3F.

REPREZENTACE ZÁPORNÝCH ČÍSEL

Zatím posíláme jen kladná čísla **– unsigned binary.** Vždycky musíme obětovat část rozsahu.
1) změníme první bit – MSB označíme jako jedna.
Pokud 5 je 0000101 a 6 je 00000110, tak -5 je 1000101 a -6 je 10000110.

ALE! To znamená, že menší číslo je v binárním zápisu vlastně větší. Navíc vznikají dvě nuly (00000000 a 10000000).

2) jednotkový doplněk – pokud je číslo záporné, znegujeme jeho absolutní hodnotu. -5 bude: 1111010 a -6: 11111001 → menší čísla jsou teď skutečně menší

ALE! pořád máme dvě nuly :/

3) dvojkový doplněk chceme z „-0“ (11111111) udělat „normální“ nulu. Jak ? Pokud přičeteme 1, vyjde nám 100000000 a protože pracujeme s osmibitovou přesností, tak se jednička odsekne. Tvořme tedy záporná čísla **NOT(abs(x)) + 1** (znegujeme absolutní hodntou a přičteme jedna)

4) s posunem (BIAS) – rozhodneme se, jakou část věnujeme kladným a jakou záporným číslům. Pokud chceme na polovinu, tak BIAS -127. **Číslo x tvoříme jako x+127**. (0 = 127, 1 = 128) → Pohybujeme se v rozsahu -127 až 128)

Při vypsání výsledků posunu myši doleva dolů (pravou a dolní část bere jako kladnou) na obrazovku nám tedy vypadávají velká čísla (např. 238). Abychom zjistili co znamenají, potřebujeme například programátorskou kalkulačku.

ALE!

23 → (odečtu 1) **236** → (zneguju) **– 237** (protože kalkulačka pracuje v 64-bitové přesnosti a doplnila tam uly, které se znegovaly na jedničky) → (převedu do hexa) **13** → (desítková) **19** → číslo je tedy **-19**

Chceme tedy pascalu vyvsvětlitl, že se na to číslo má koukat jako číslo ve dvojkovém doplňku.

ŘEŠENÍ

To, co počítač dělá, je takzvané **unsiged extension**, tedy bezznaménkové rozšíření. My chceme, aby ta čísla bral jako znaménková a provedl **znaménkové rozšíření.** To provede jen tehdy, pokud oba typy (vstup i to, do čeho konvertujeme) byly znaménkové. Proto neukládáme do rpoměnné typu Bajt, ale typu Integer.

A jak funguje bezznaménkové rozšíření ?
Místo toho, aby automaticky na volná místa doplnil nuly, doplní tam to číslo, které je na pozici prvního bitu. Pokud je tam nula, doplní nuly, pokud jedna (u záporných čísel), doplní jedničky → při konverzi se tedy změní na nuly a nám konečně vypadne hodnota, kterou chceme.

Pokud bychom chtěli konvertovat naopak z většího do menšího formátu – tak se odseknou nadbytečné bity → funguje to jenom malá čísla (která se vejdou i po konverzi do menšího typu), jinak se nám hodnoty změní a začnou vycházet blbosti.

Pokud tedy provádíme konverzi, je třeba dobře vědět, jak jednotlivé typy fungují. Navíc integer je definován někdy jako 32-bit, někde 16-bit !

každé číslici při posunu shl zvětšíme váhu --- jako bychom to číslo vynásobili dvěma tedy posun o n je to samé jako * 2^n ! To je důležité, protože bitové operace jsou nejrychlejší operace procesoru (násobení je třeba 10* pomalejší)
SHR = celočíselné dělení mocninou dvojky (běžné dělení trvá až 100* déle než b. o.)
0 1 0 1 shl 1 → 1 0 1 0
5 10

ALE! pozor – pokud máme například -5 ve dvojkovém doplňku (1011) tak posunem se nám z toho stane kladné číslo (jelikož první bit můžeme reprezentovat jako znaménkový)
chceme tedy něco takového i pro záporná čísla

ŘEŠENÍ aritmetický posun (aritmetic shift/**sar**/shift aritmetic right) – pro kladná čísla se chová uplně stejně jako normální SHL, jinak podobně jako znaménkové rozšíření : do nových bitů kopíruje původní nejvyšší bit
1 0 1 1 sar 1 → 1 1 0 1

funguje podobně jako dělení dvojkou – pro čísla dělitelná mocninou to funguje, pro nedělitelná to zaokrouhluje nahoru. Přesně to lze dořešit kombinací bitových operací

(V C : podle typu operandu se dělá aritmetický nebo normální posun, v Pascalu dělá SHR vždy logický posun a Sarlongint (číslo, okolik))

Jak je reprezentujeme v desítkové soustavě ?

2	0,	6	2	5
10^1	10^0	10^{-1}	10^{-2}	10^{-3}

REPREZENTACE REÁLNÝCH ČÍSEL

→ stejně budeme postupovat i ve dvojkové soustavě (zde 3, 625)

2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}
0	1	1	1	0	1

Fixed point – pevná pozice desetinné čárky. Např. 5.3

opět potřebujeme vědět, jak je číslo velké (například osmibitové) + kde je desetinná čárka (napevno).

Výhody ? Dobře se s tím počítá, je to rychlé

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

sečtu to jako celá bezznaménková čísla a pak vrátím na své místo desetinnou čárku

Sčítání ve dvojkové soustavě
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 a přeneseme 1 do dalšího sloupce

Sčítáme odzadu a pokud přenášíme 1 do sloupce kde už jsou dvě jedničky, sepíšeme 1 a 1 přeneseme dál

ALE! Nefunguje násobení, nelze sčítat velmi velká čísla s velmi malými
206525000000,00
0,00000000020625
s obou čísel to vezme jen nějaký rozsah a zarovnaný podle desetinné čárky a vrátí nám to například nulu

Reprezetnuejem je pomocí tzv. Vědeckého tvaru : 2,0625 x 10^{10} stejně jako v desítkové soustavě → **floating point** (plovoucí desetinná čárka)

Normalizovaný formát – před desetinou čárkou máme jen jednu nenulovou číslici

mantisa – číslo které násobíme
exponent – na kolikátou mocníme základ soustavy

například:

znaménko	exponent	exponent	exponent	mantisa	mantisa	mantisa	mantisa

musíme říct – jak velké je číslo, jak velká je mantisa/exponent, v jakém je to pořadí
mantisu obvykle nemůžeme uložit celou, jen s určitou přesností - ale i tak je to celkem rozumné zaokrouhlení
nula porušuje pravidlo normalizovaného tvaru mít před desetinnou čárkou nenulové číslo → pro všechna ostatní čísla to platí → **zápis se skrytou jedničkou** – jedničku nepíšeme, protože tam je vždy a tím získáme o bit navíc
znaménkový bit – určuje znaménko čísla exponent může být opět kladná nebo záporný :

! používá se reprezentace s bias posunem
nevýhody – ne všechny procesory mají podporu tohoto formátu, je to o dost pomalejší než práce s celými čísly → tedy pořád může být výhodné pracovat s fix-point

IEEE standart – definice standartních formátů čísel
Single – 32bit, 8 bit exponent, 23+1 bits mantisa
Double – 64 bit, 11 bit exponent,
Mantisa na spodních bitech

V pascalu i dalších jazycích jsou ijiné reálné datové typy (Real), ovšem nemusí si pak rozumět s jinými rpocesory, jazyky….

! SINGLE: Pořád nám to třeba 5 miliard + 5 sečte jako 5 mld. → nepoužívat na reálná čísla porovnávání, spíše hledat, jestli je to v nějakém okolí toho druhého čísla (kvůli zaokrouhlovacím chybám)

! ve dvojkové soustavě jsou jiná čísla periodická (například 0,1) – pak třeba cyklus přičítající 0,1 dokud hodnota != 1 bude nekonečný…

Nejjednodušší multi-drop sběrnice využívají jen dvoustatovou logiku. K vodiči se signálem je připojen tvz. **Pull-up rezistor,** který posílá 0 pro připojeno a 1 pro nepřipojeno (to se nám později hodí). Na vysílání jedničky není připojen, na vysílání nuly se připojí. Pull-up rezistor je připojený na datovou sběrnici (je větší než odpor zařízení připojených na zem, ale menší než odpor v měřiči napětí), tudíž napětí je skoro 5 V (je zapojen paralelně), když se jeden rozhodne vysílat, tak existuje kratší cesta přes seriově zapojené rezistory → odpor je velký a napětí je velmi malé → vysílá nulu.

Když vysílá více zařízení:
pokud 1 1 → 1
pokud 0 0 → 0
když jeden vysílá 1 a druhý 0 → 0 (nula přebije jedničku z toho druhého zařízení)
==> tento odpor funguje jako operace OR.

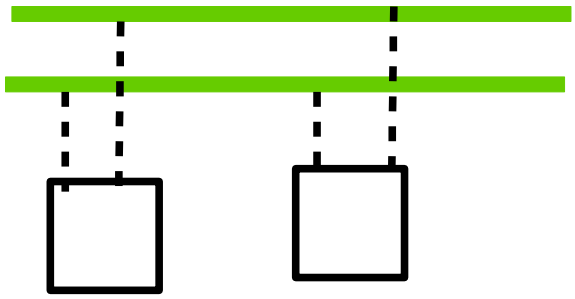
SBĚRNICE I^2C (= inter-integrated cicuits)

Relativně stará, ale stále používaná multi-drop sběrnice. Využívá rychlosti 10kHz – 100kHz – 5MHz. Rychlost se může měnit díky tomu, že využívá hodinového signálu.

Má dva vodiče:

SDA (data)

SCL (hodinový signál)



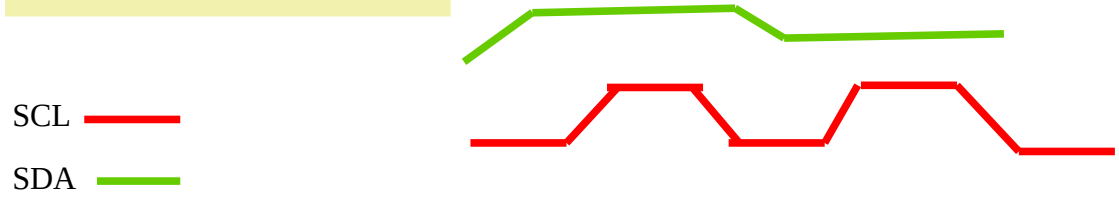
Všechna zařízení na sběrnici jsou napojena na oba vodiče, je připojena k zemi i k napájení.

Linka je také *multi-master* (multi-slave): To znamená, že i když v každou chvíli je jenom jeden master, mohou se v této pozici střídat. Některá ovšem mohou být jenom master, nebo jenom slave.

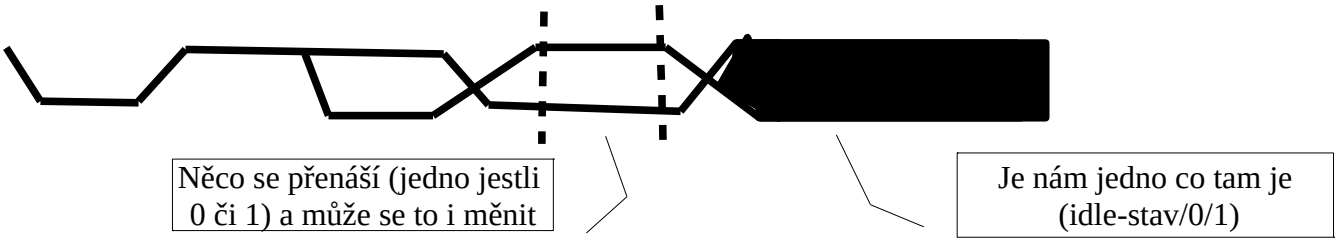
Implicitně je linka v idle-stavu → 1 zařízení se rozhodne být master a začne generovat hodinový signál → ostatní se do konce přenosu chovajíjako slave. Nejběžněji je masterem CPU.

! Master nemusí jen zapisovat, může i klást dotazy a očekávat data od slavea.

TIME DIAGRAMS

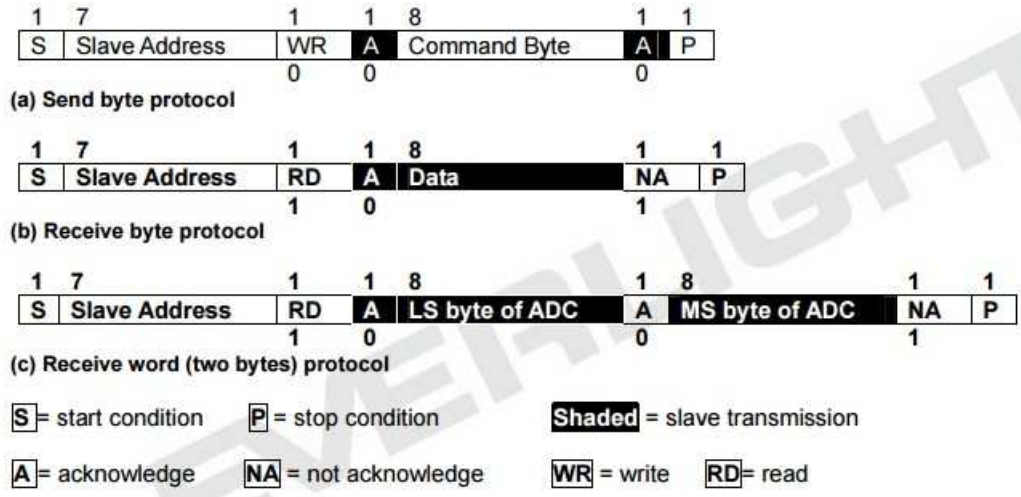


Platná hodnota se bere až když je SCL v poloze 1.



! Nemáme pevně danou délku přenosu, takže je třeba definovat start- a stop-conditions.

ŘEŠENÍ když se během 1 na hodinovém signálu změní hodnota (normálně nepovoleno), tak 0 → 1 je start a 1 → 0 je stop. (Prvně se odpojí data a až pa hodinový signál, to vytvoří tu stop-condition) Nezapomínejme, že idle-stav tu díky pull-up rezstorům vypadá tak, že je všude 1.



Jako příklad I^2C sběrnice jsme si vzali nějaký light senzor, jehož komunikační protokol můžeme vidět výše.

ACK/Acknowledge – kdykoliv někdo někoho volá (zapisuje nebo čte data), potřebuje potvrzení, že dané zařízení komunikuje – ACK bit. *NACK* je naopak „nepotvrzeno“

Klasicky 1 znamená pozitivní/potvrzení/připojeno/ano. Pokud je nějaký příkaz (WR,RD..) s nadtržítkem, znamená to, že se hodntoy interpretují naopak. U ACK by vlastně měl být pruh, protože když je zařízené vypnuté, všude běží jedničky, takže posílá nulu.

ALE! Když chce master zapisovat víckrát (například číslo ve více bajtech), je nebezpečí, že po stop-condition tam skočí jiný master.

ŘEŠENÍ místo stop-condition dá master na konec další start-condition a naváže dalším přenosem.

Detasheet – hardwarová specifikace zařízení (protokoly, zařízení atd..)

Podívejme se konkrétněji na náš light ambient senzor. Má čtyři připojení (zem, napájení, data, CLK). Načte v pravidelných intervalech nějaká data (hodnotu světla) a pak si je přečteme(podle toho se například upravuje jas na mobilech – display zhasne, když ho dáme k uchu). Některá zařízení mohou být jen read nebo write (display), tady ale můžeme(musíme) obojí.

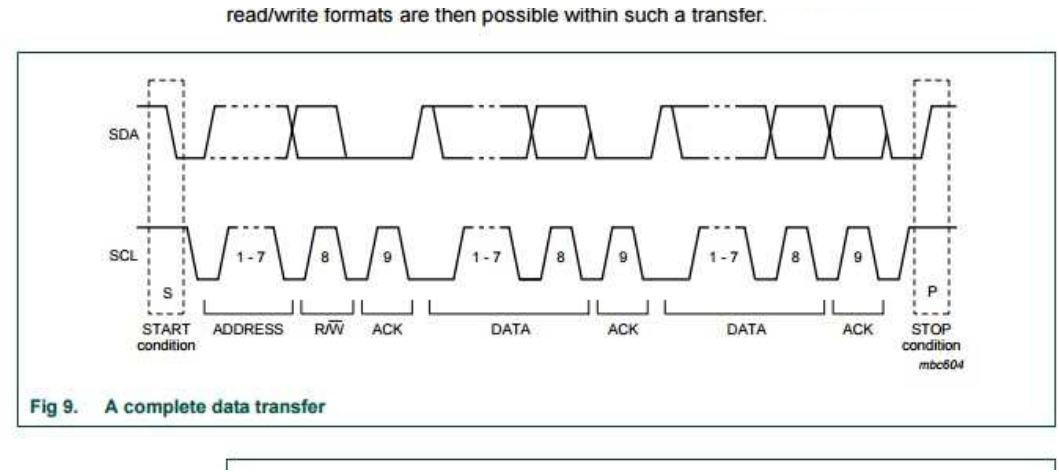


Fig 9. A complete data transfer

Přenos
1) jako master zapíšu 1 B (7 bitů adresy + 1bit informace R/W)
2) slave generuje 1 bit že slyší (ACK) (nyní vidíme, že když jsou všichni odpojeni – i master, který čeká – proudí jednička a nic se nemění, pouze když se slave aktvně rozhodne posílat 1)
3) pošlou se 2* 1 B (celé to16-bit číslo o stavu světla)

Máme definovános sběrnicí MSF-first přenos.

ALE! který bajt se posílá první ? MSB

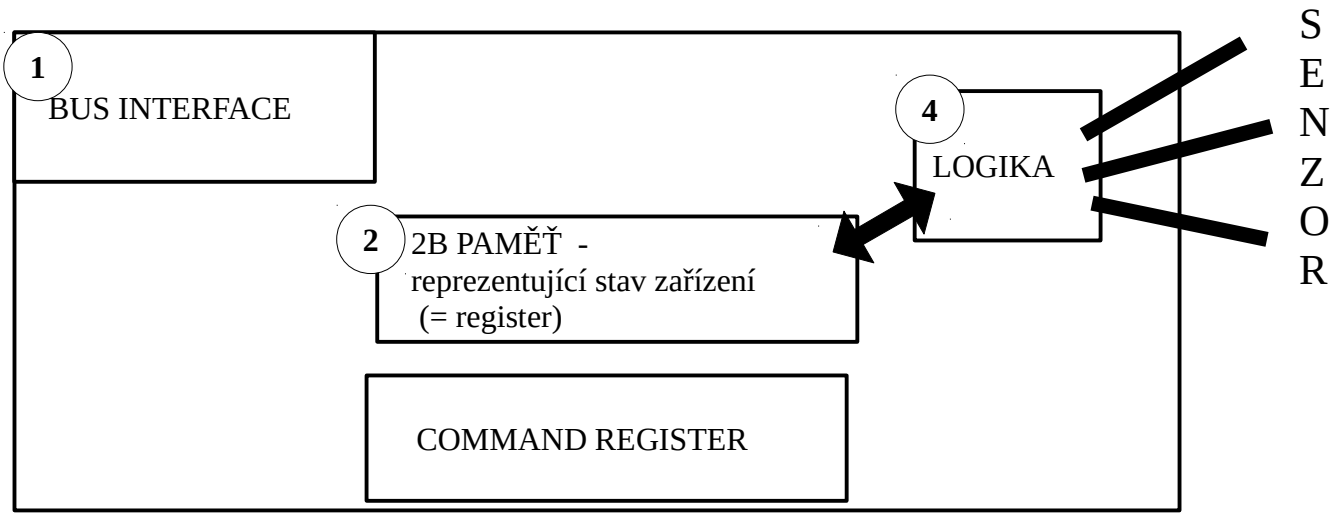
LSB

15 **8** **7** **0**

Ř: mluvíme o tzv. **Byte -order –** zajišťuje zařízení, ne sběrnice. (Opět značíme MSB x LSB-first → význam musí být jasný z kontextu).

ALE! Jak ví, jaká je jeho slave adress ? (nemá nikde možnost jí generovat atd….)

:) má jí v sobě pevně zadrátovanou (=hardwired) → nemůžeme mít na jedné I^2C sběrnici dvě stejné zařízení, protože by měly stejnou adresu.



1 Zajišťuje, že zařízení umí komunikovat přes sběrnicic (řeší pořadí bajtů..)

2 pamatuje si čísla reprezentující údaje ze senzorů (zpracovává **3**), R/O (read-only)

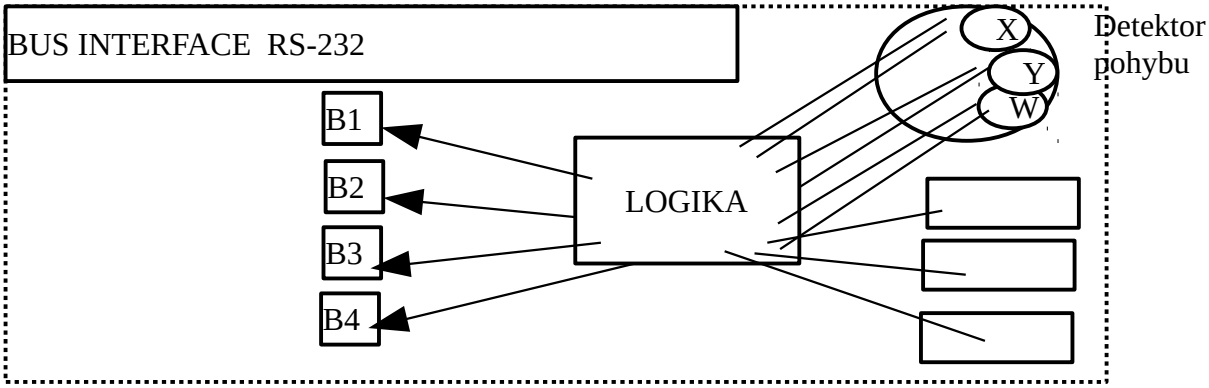
3 ukládá se sem zápis od mastera (čti/zapisuj, zapni se/ vypni se) W/O (write - only)

pozn. Registry mohou být i R/W

! Výhoda této koncepce :
pokud potřebuju komunikovat po jiné sběrnici, ***stačí vyměnit*** BUS INTERFACE a zbytek zůstane stejný.

Všechna zařízení jsou si ***velice podobná*** (například myš…), liší se počtem senzorů, registrů a logika provádí jiné operace

MYŠ

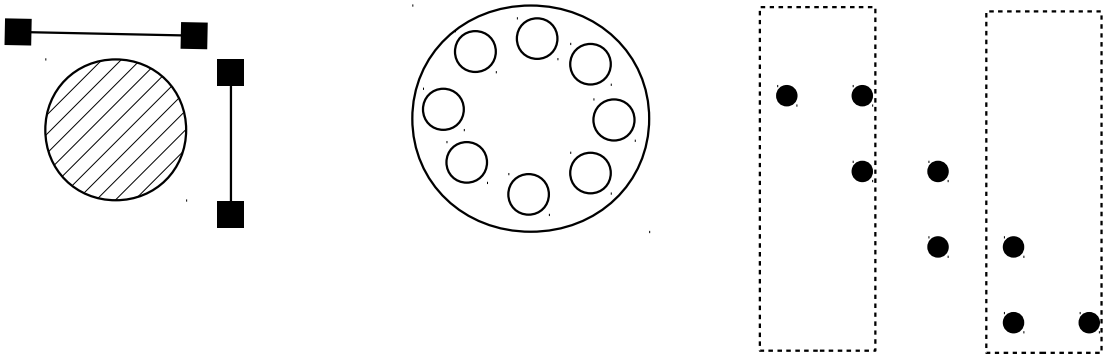


Ze senzorů pohybu a z tlačítek vede do logiky celkem devět signálů, které zpracuje, jak už víme, na 4 B číslo.

Detektor pohybu vypadá následovně:

kulička otáčí dvěma válečky

pohyb každého válečku je přenášen na clonku (kolečko s otvory) a každá clonka má dva detektory nastavené tak, aby mohly zachytit dva signály celkem čtyřmi způsoby:



Na základě tohoto se pak spočítá a kolik a kterým směrem se myš posunula.

PAMĚŤ V MULTI-DROP

Chtěli bychom mít code paměť i datovou paměť na jedné sběrnici:
Mohlo by to vypadat jako práce s registry, čili načíst celou paměť a pak s ní pracovat.

ALE! Byla by to zbytečná práce, jelikož si chceme říct jen o konkrétní data →

:) rozdělíme si jí na celky (typicky bajty) a uspořádáme ji nějak unikátně (myšlenkově po 8bitech do jednotlivých bajtů). Jedinečnému číslu bajtu říkáme adresa.

0 **1** 2 3

Lineární uspořádání. Ale v paměti jsou uložená jen data, nikoliv jejich adresy (ty jsou pevně dané – hardwired). Pokud máme adresy 0 – 65536, potřebujeme 16-bitové adresy na pokrytí všech bajtů. Ovšem současné paměti jsou velké → nevypisujeme v bajtech ale v násobcích.

! **1kB = 1024 B** ….protože 2^{10} = 1024...a rovnou z toho vidíme, kolik potřebujeme bitů adresy. Pokud by 1kB byl 1000, tak by nám v 10 bitové pamětí zbývalo 24 volných adres, nebo by nám v 9-bitové paměti nějaké chyběly.
1MB = 1024 kB …..

Snaha o rozlišení MiB = 1024 kB a MB = 1000 B, ovšem to se neujalo. Pouze prodejci udávají kapacitu ve fyzikálním významu (=1000), takže nám prodá menší kapacitu.
Příklad : Paměť 256 x 8bitů → 8 bitový adresový prostor.
Pokud chceme něco zapsat, tak sdělujeme adresu bajtu + bajt k zapsání

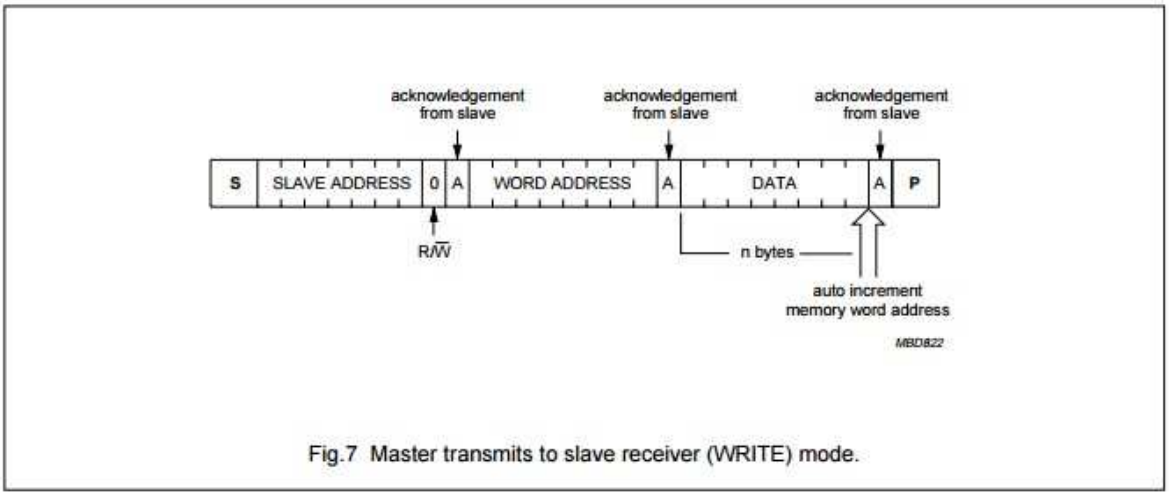


Fig.7 Master transmits to slave receiver (WRITE) mode.

Word – jednotka po které se přenáší data. V tomto případě 1 word = 1 bajt, ale mohou být zařízení, kde se na jeden přenos přenese například 16 bitů = 1 word

Zjišťujeme, že během přenosu se nám mění R/W – nejrpve proběhne přenos W – zapíšeme adresu a potom R – čteme data z bajtu adresy → potřebujeme, aby nám tam nikdo neskočil, protože pak bychom četli třeba z úplně jiného bajtu – zde využijeme toho, že místo stop-condition pošleme start-condition.

Paměť má v sobě opět jeden registr, kde si pamatuje poslední volanou adresu.

Tato konkrétní paměť má 4 bity adresy dané a 3 bity se zvolí podle toho, jak jí zapojíme ke sběrnicy (zbydou tři volné piny) → můžme mít těchto adres na sběrnici několik. (Poslední byt je R/W).

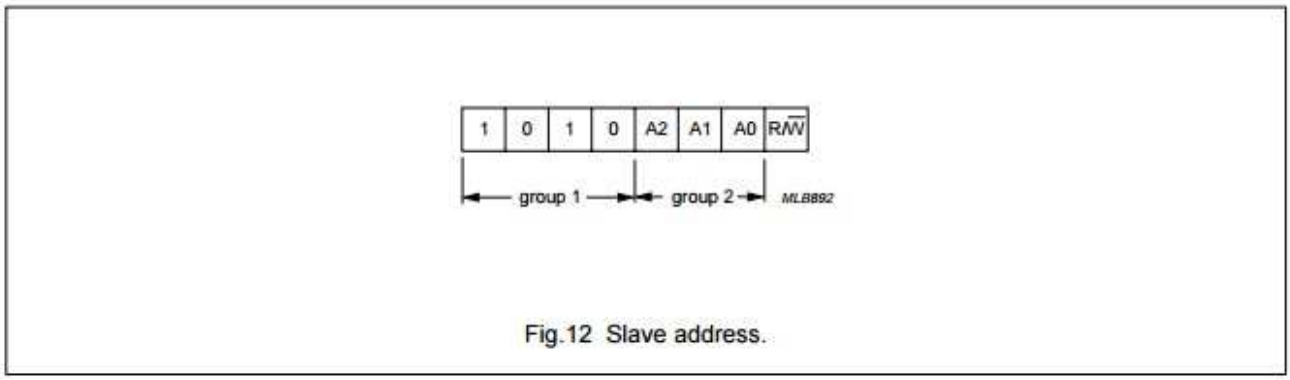


Fig.12 Slave address.

POZOR! Na jeden přenesený (přečtený) bajt potřebujeme (overhead) :

- 1 B adresy zařízení, 1B adresy bajtu - a chceme číst
- 1B adresy zařízení, 1 B data
- navíc každý bajt = 8bitů + 1 bit ACK

→ rychlost x bits/s je potom x/(4 x 9) Bytes/s

Řešení: Tato konkrétní paměť podporuje BURST TRANSFER – po přečtení jednoho bajtu zvětší adresu bajtu a když nepošleme negativní ACK tak pošle další bajt, tak dlouho dokud chceme. (podobně burst write).

Nevýhodou je, že pokud jeden master chce číst třeba dvacet minut, tak nikdo jiný nemá šanci.

Vezmeme-li si jiné zařízení, například proximity senzor, který se využívá například v mobilních telefonech. Mnohem komplikovanější než light senzor z minulé lekce. → je tam mnohem více registrů (zde konkrétně 11)

! Tady nemůžeme číst a zapisovat do všech možných registrů – je to pomalé, kolize příkazů

Řešení : každý registr může mít nějaké číslo a budeme pracovat podobně jako s pamětí. 3iformace: číslo zařízení, zápis/čtení, číslo registru

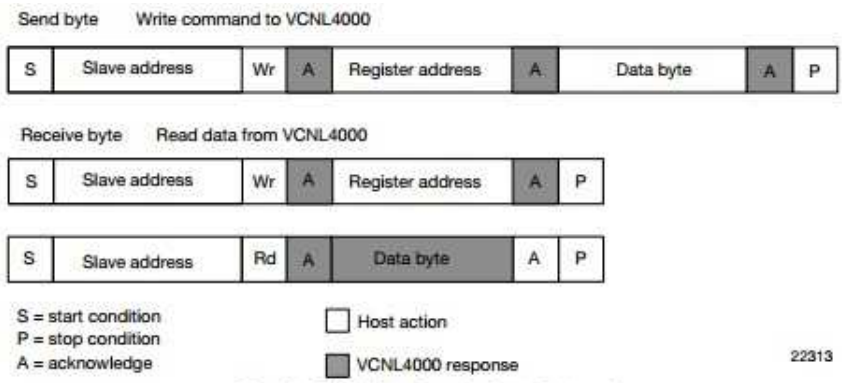


Fig. 23 - Send Byte/Receive Byte Protocol

he I²C specification from NXP for details.

8 bitů adresy → tedy mám více adres než registrů, jsou vybrány jen některé

Jak fungují paměti ?

Adresace ?

Pro příklad máme 8 B paměť → potřebujeme 3 bitový adresový prostor *(doporučuji prezentaci na webu).*

Ale nikde v paměti to ve skutečnosti **není nikde uloženo**, jakou má adresu. Ale je tam nějaká součástka, která té paměti řekne : „S tebou chci pracovat!“

Shift register – postupně přicházející bity adresy se posunují do prava a uložené hodnoty se posunují pryč.

CS (chip select singal/chip enable /CE) – když je 0, tak zařízení ignoruje veškeré ostatní signály. Když je to 1, tak začne sledovat ostatní signály a chovat se podle toho. Dříve to by přímo chip připojený na ten obvod.

Cílem je mít 1 na jednom a 0 na všech ostatních → **Demultiplexer/demux** – proloží více informací do menšího množství. Snaží se nastvait podle adresových bitů správně 0 a 1 v registrech (v CS registrů).

Je to v podstatě řada přepínačů (viz prezentace), které jsou ovládány bitovým signálem. **<u>Adresa tedy není uložena, jde o to, jak je to zadrátované.</u>**

ALE ! V průběhu nasouvaní té adresy se vytvářejí různé „náhodné“ kombinace adres a podle toho on v průběhu by měl různě měnit CS na různých registrech. Takže by mohly buňky vysílat někdy během té adresace úplně špatně. **Řešení:** Do všech registrů vede jeden vodič „chci z tebe číst“ (a jeden „chci do tebe zapisovat“). Ale všechny ostatní registry kromě toho s CS = 1 ho ignorují. Takže až když je CS = 1 a zároveň se vysílá „chci čtení“, tak ten jeden registr začne plivat data.

Ukládání dat ?

Nejmenší jednotkou je bit, ale vždycky s pamětí komunikujeme minimálně po bajtech. Můžeme si to představit jako jednotku o 8 bitech. Pokud to máme zapojené fikaně a připojíme k tomu kromě read signálu, CS a dat ještě CLK signál, tak nám provede 8 tiků, v každém se nám to posune doleva, takže to cestou „vypadne“ na datový vodič,a le pak se to zase zasune na začátek, takže po 8 ticích je obsah paměti stejný, ale stačil se nám postupně odeslat jako data.

Latch/flip-flop – nejjednodušší datová jednotka (buď „1“ nebo „0“) například tranzistory – 4 až 6 tranzistorů na jednu bitovou buňku a pak se spojují do větších celků. → SRAM (statická paměť).

Někdy se i té větší n-bitové paměti říká latch (pokud je to třeba ta mrňavá 8 bitová pamětička/registr).

RAM – random acces memory – nezávisí na pořadí čtení bitů ! **ALE POZOR !!** to bylo historicky, ale **současné RAM nejsou random acces memory.** Protože pokud podporují burst přenosy, tak je čtení následujících bajtů/bitů mnohem rychlejší než náhodné. A ještě je třeba nečíst pozpátku. Random acces memory se typicky předpokládá, když se řeší složitost algoritmů. (takže pozor, ve skutečnosti to tak úplně nefunguje. Někdy to ovlivní i asymptotickou složitost, případně zrychlí i o 10%).

Volatile – po odpojení napájení zapomene všecko.

kriteria pro porovnávání pamětí :

	transfer rate (nejlepší možná)	acces time(od chvíle kdy řeknem adresu do posílání dat)	kapacita	výhody	nevýhody
SRAM	>GB/s	1 ns	max MB	Nejrychlejší, operační paměť (aby nebrzdila procesor)	Volatile, drahá, malá kapacita (pro dynamciké proměnné)
DRAM	10GB/s	10 ns	GB	levnější	Volatile. Nutnost refreshe
EEPROM			KB - GB		Celkem drahé
flash	500 MB/s	100 ns	100GB	Větší kapacita	

DRAM – dynamic RAM, o dost levnější protože na 1 bit jen 1 tranzistor a 1 kondenzátor, takže se na stejnou plochu vecpe více bitů. Nejčastěji používaná kvůli poměru kapacita/cena. Někdy se alespoň pro část používá SRAM kvůli rychlosti. **ALE** do 10ms zapomene všecko. Všechny ty kondezátory jsou různé pospojované, takže po 10ms se všechny (vybité i nabité) dostanoou zhruba na stejnou úroveň. Závisí na okolní teplotě – tedy počítač se nebojí ukládat rozšifrovaná hesla v této paměti. Takže když vám ponoří notebook do tekutého dusíku a pak vytáhnete, tak se to dá zjistit.

Řešení: Dělá se refresh do těch 10 ms. Přečteme a znovu napíšeme celou paměť, aby se obnovily původní hodnoty.

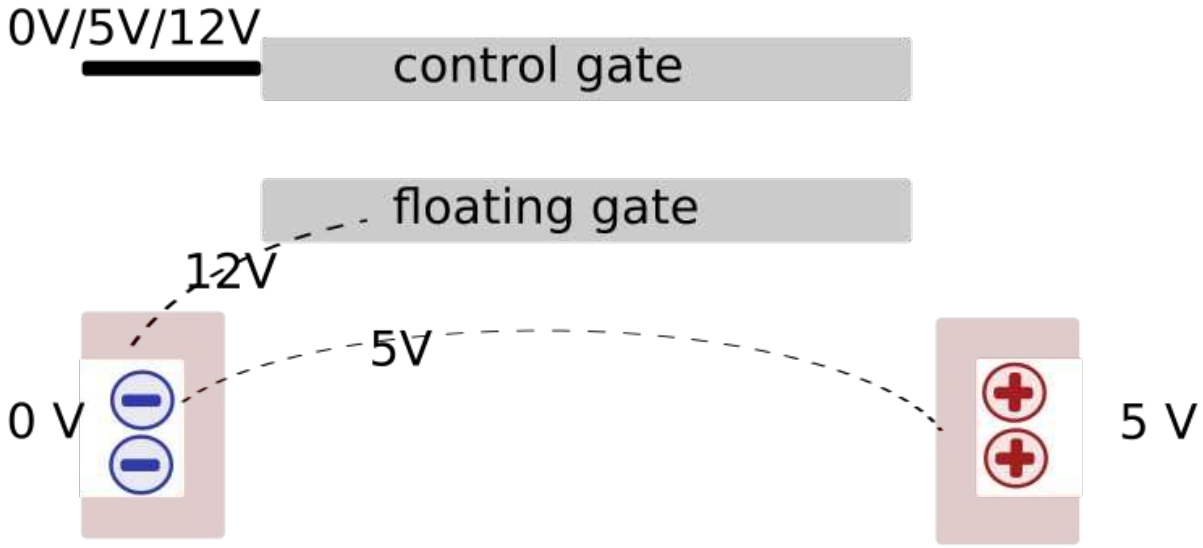
Nevýhody: potřebujeme zařízení na refresh, během refreshe se nedá s pamětí nic dělat → snižuje rychlost.

Volatile paměti jsou dost nepoužitelné pro trvalé ukládání dat. Potřebujeme non-volatile paměti.

ROM (read only memory) Je non-volatile. Potřebujeme tam alespoň jednou dostat ta data. → dostanou se tam z výroby a pak už dělají do konce života to samé.

Nevýhody : nedá se změnit. Nelze vyrobit jen jeden)vždycky seriove třeba milion)

PROM (programmable ROM) dovoluje jeden zápis, ale vyrábí se prázdná a poté se teprve do ní zapíše program (přepálíte jednotlivé diody a vícekrát to nelze změnit).



Máme dva vodiče a mezi nimi izolant/polovodiče. A vodič – control gate. Když tam není zapojeno nic, tak se nic neděje. Pokud je kladné, tak je síla tak velká, že se procpou do protějšího vodiče a začne proudit proud. A definujeme to jako „1“. Funguje to jako spínač.(nMOSFET tranzistor). Taky to funguje jako bitový AND

ALE chceme, aby si to pamatovalo nulu například když čteme (tedy připojíme napětí, ale je čtecí).

Proto máme ten druhý vodič floating gate. Když do control gate zapojíme větší napětí, taks e začnou hromadit ve floating gate. A nemají se jak dostat ven. Další elektrony se ale už nikam nedostanou, protože floating gate je narvaná elektrony a ty odpuzují ty ostatní.

Jednotlivé paměti se liší tím, jak vrátit nulu na jedničku.

EPROM Vršek čipu je vidět (třeba okýnkem). Když září UV světlo, tak se vyráží elektrony z floating gate a paměť se maže. (Když jí nechci mazat, musím zalepit okénko). Stačí cca 20min na slunci. Je to dost nešikovné : musím to vyndavat z počítače a je to pomalé.

EEPROM (electricly EPROM) mažeme elektricky. (těmto pamětem se často říká NV-RAM - non -volatile pro čtení i zápis).

FLESH MEMORY – větší kapacita, bajty složené do bloků a dá se číst jen po blocích. (→ vynucené burst přenosy). Pokud chceme celé kB, tak je o dost rychlejší než EEPROM. Ovšem čteme-li nešikovně (proti členění na bloky), může být i mnohem pomalejší.

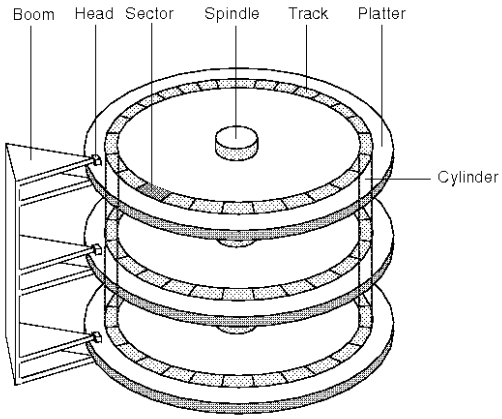
Ne všecky elektrony lze vyrazit z floating gate, takže po nějaké době je nepoužitelná. Paměti mohou mít různý počet cyklů. (100 tisíc i miliony).

Používají se pro uložení kódu (v harwardské architektuře).

3D x-point – připravovaný nový druh paměti. Podobná kapacita jako flash a nejsou o moc dražší, jsou non-volatile. Rychlost by měli mít podobnou jako DRAM. Pak by bylo možné je použít i pro operační paměti – tedy by se změnil koncept na non-volatile paměti.

Nyní potřebujeme nějakou trvalou paměť na data, která bude non-volatile. Je potřeba také veliká kapacita.

PEVNÉ DISKY



- velikost až TB

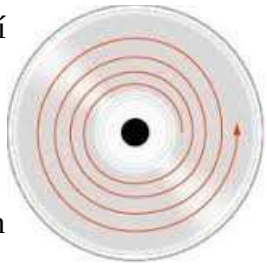
- rychlost je velice malá (nikoliv sekvenční 100MB/s, ale acces time: 1 – 10 ms)

Rychlost je pomalá kvůli architektuře disku. Hlava se přesouvá mezi jednotlivými stopami (track). Každá stopa rozdělená na sektory. Ploten může být víc. Je třeba najít plotnu, stopu a počkat na zastavení disku. Opět je tedy nejvýhodnější sekvenční čtení.

Sektor dříve velký 512 B, nyní 4096 kB.

LBA (linear block adresing) – místo čísla hlavy, stopy, sektoru si očíslujeme sektory nějak za sebou a přistupovat budeme přes jedno číslo. To také umožňuje mít různý počet sektorů na jednotlivých stopách.

V každém sektoru se opět koukáme na data jako na bajty seřazeé za sebou. Místo adresa se používá offset (vzdálenost od začátku). Tedy že „bajt je na offsetu 4“ je-li 4. (číslováno od nuly). Můžeme to aplikovat na celý disk. První sektor má bajty s adresou 0 – 511, druhý sektor 512 – 1023 … Takže máme opět lineární reprezentaci dat.



CD – čtecí mechanika vypadá podobně, ale hlava je mnohem masivnější → trvá déle jí rozpohybovat. → zásadně horší přístupová doba (100ms). Přenosová rychlost cca 10MB/s, kapacita 1 – 10GB. Výhoda je, že je výměnné. Data jsou uložena v jedné stopě, která tvoří spirálu od středu. Typicky je sektor 2kB.

SSD (solid state drive) – flash paměť s rozhraním jako pevný disk. Přirozené má flash paměť bloky, kterým můžeme říkat sektory.

Je vidět, že pro jednoduchost lze všechny tyto způsoby ukládání dat převést na lineární uspořádání dat.

Víme tedy, že všechny non-volatile paměti, sloužící jako datová úložiště lze oadresovat dle sektorů a i jednotlivé bajty, tedy převést vše na lineární adresový prostor.

Pracujeme-li s daty, chceme je nahrát z disku do paměti (operační). Disky mají velkou kapacitu a obsahují různá data. Bylo by dobré mít skupinu dat, které spolu souvisí, v jedné skupině sektorů a pojmenujeme to „soubor“.

Soubor – pojmenovaná posloupnost dat na pevném disku

ALE! data typicky nejsou přesný násobek velikosti sektorů. Nejběžnejší je, že v posledním sektoru zbyde volné místo a **nový soubor** začíná na **začátku** nového.

Metadata – délka souboru, jméno a kde začíná; ne samotná data ale „data o datech“

Souborový systém – uspořádnání dat a metadat určitým způsobem

Nyní lze přiřadit jednotlivým bajům nové adresy v adresovém prostoru souboru ! Nyní můžeme od nějaké adresy x načíst první až n-tý bajt souboru.

LITTLE ENDIAN/
BIG ENDIAN

nejběžněji když chci přístup k nějakému datu (např čtyřbajtové číslo), tak řeknu nejmenší adresu paměti, na které to leží.
Jak uložím například 32 bitové číslo ? Opět je třeba se dohodnout na byte - order → endianness

máme dvě možnosti: little-endian/LE – na nejnižší adresu LSB, big-endian/BE

Little endian/big endian (inspirace příběhem z knihy Gulliverovy cesty).
Jsou dva národy : jedni si myslí, že ten správný způsob, jak rozbíjet vejce je na menším konci (little end) a druzí, že na širším konci (big end). Pointa je, že je to jedno a místo hádek je třeba se rozhodnout a vejce spotřebovat, než se zkazí.

Little endian má **výhodu** - dost často potřebujeme přečíst jenom spodní bity čísla – například protože víme, že je to malé číslo a lze ho uložit v menším datovém bitu. Jsou-li tedy data v LE, stačí přečíst 16 bitů od zadané adresy, v BE je to komplikovanější.

Pascal si vybere za nás jak to ukládat. Když **program** dává pokyn procesoru :
KAŽDÝ PROCESOR UMÍ PRACOVAT JENOM S JEDNOU ENDIANITOU
(některé podporují obé, lze vybrat při startu počítače), drtivá většina jsou LE → i většina dat je uložena v LE
Lze říct, že soubor je text (f:text) případně že to je binární soubor: (f: file of byte) a funkce BlockRead umožní načíst do pole n bytů od zadané adresy.

„hele mám dvě čísla, obě mají 4 B a jedno je na adrese 64 a druhé na adrese 72“,

```
procedure BlockRead  (var F: file; var Prom; Pocet: Word)
```

Z *file* načte do *prom pocet* bytů

Nejběžnější zobrazení binárního souboru : každý byte jako 1 šesnáctkové číslo (bajt 0 je nejvíc vlevo) a po nějakých bytech to zalamuj.

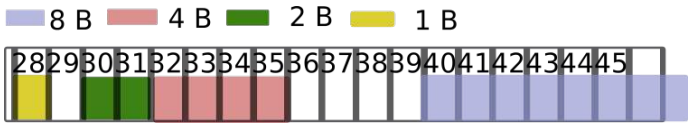
PRÁCE S DATY

je k němu třeba používat **POINTERY** (ukazatele) – číslo, které je třeba chápat jako adresu v paměti → je tedy velké jako adresy v paměti (16/**32**/64 bitů, 32 bitů nejběžnejší, tedy 4B)
Obvykle chceme navíc pointeru říci, na jaký typ proměnné ukazuje.

Alignment - proměnná leží na adrese, která je dělitelná její velikostí (protiklad je missaligned/unaligned), podle toho přiděluje adresy.

P := @abc … výsledkem je pointer s adresou
v C/C++ (@ → & , ^ → *)

Jazyk se snaží o alignment (zarovnání)



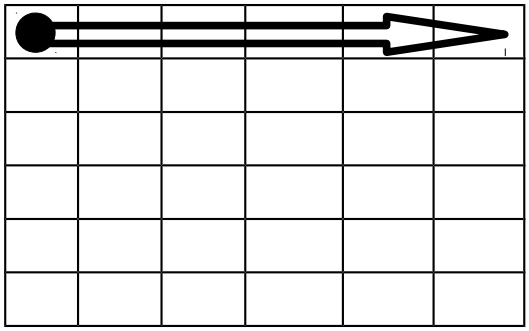
volným místům v paměti říkáme padding (vyplněny hluchými daty – *lze použít i jako sloveso (to pad)*)
nastavení zarovnání v pascalu: $codealign varmin (když = 1 → vše zarovnané na adresu 1, tedy nezarovnané)
dereference - změna hodnoty proměnné, na kterou ukazuje P (P^ := 50;)

Pointry umožňují :
přístup k jakémukoliv místu v paměti.
Stačí mít přístup k první proměnné a pokud předpokládáme, že proměnné leží za sebou (což nemusí být), tak pak stačí zvětšit pointer a pokračovat dále, čímž plynule čteme postupně obsah paměti.
Pointerová aritmetika
P + n zvětší o nkrát velikost toho typu, tedy posune ne na další adresu, ale na adresu další proměnné
ALE! Pokud P := pointer(@a) → pak se posune skutečně o zadaný počet B

POSÍLÁNÍ DAT PO SÍTI
hlavička/header – popis následujících dat před nimi (při posílání po síti)
v paketu je ethernetová hlavička (14B), IP hlavička (20B), TCP hlavička (20B) data (proměnná velikost)
někde v IP hlavičce je popsaná délka paketu
POZOR!! network order = big endian, tedy data přicházející po síti jsou BE
→ pro čtení je třeba použít metodu, která data přeskládá **JAK ?**
bitové operace
1.B posunutý o 4*8 bitů OR 2. B posunutý o 3*8bitů OR 3.B posunutý o 8bitů OR čtvrtý bajt

WireShark – zachytává síťovou komunikaci
hex editor – jednotlié bajty v hex. soustavě

Důležité je umět reprezentovat i jiná data než čísla – fotky, videa, zvuk.



OBRAZ

rozdělíme si plochu na čtverce = pixely a intenzitu světla si uložíme jako hodnotu pixelu čili opět převod na čísla

směr ukládání od nejnižší adresy (pixel v levo nahoře) a jdeme po řádcích. Souřadnice obrázku se určijí podle pixelu [0,0]. ➡ Opět to tedy chápeme lineárně *(0,0 1,0 2,0…. n-1,0 0,1 1,1 …)*

Jak vyčíslit hodnotu pixelu ?

- ***1 pixel = 1 bit***
- a zaokrouhlíme hodnotu na 0 nebo 1 → vznikne černobílý obrázek a v každém bajtu je tedy prvních 8 pixelů

ALE! Ztratíme barvu a navíc i další hodnoty intenzity (velké množství obrazové informace)

R	**G**	**B**
8 bitů	8 bitů	8 bitů
MSB		**LSB**

- rozložíme pixel na více informací (podobně jako lidské oko – čidla na intenzitu a tři čípky snímající intenzitu kolem modré, červené, zelené …) , je vidět že nepotřebujeme informace o dopadajícím ultrafialovém, infračerveném a dalším neviditelném záření.
  Pamatujeme si pro každou barvu 8 bitů …. celkem asi 16 000000 barev (což je méně než vidí lidské oko a některé barvy budou splývat)
  každý pixel = 3B

?! Jak skládáme barvy ?

225 255 255 je bílá a 0 0 0 je černá (skládání ***jako světlo***, nikoliv inkoust)

B G R ***ukládání v little-endian***
0 1 2

ALE!

datové typy jsou obvykle mocniny dvojky – 16bit je málo a 32bit je moc → **plýtvání**

bajt navíc se používá pro alfa kanál – neprůhlednost pixelu (nefunguje to tak všude) ➡
model aRGB (respektive v paměti BGRa)

ALE!

dále ***potřebujeme vědět*** (kromě té řady) :

- rozměry obrázku (šířka - abychom věděli, kdy zalomit řádek, výška – abychom nezobrazovali další náhodná data z paměti)
- jak vypadá pixel (signed/unsigned čísla, alfa kanál? , …)

Takže když to ukládám do souboru, potřebuju kromě samotných dat i ty informace (na začátku souboru hlavička)

Bitmapa – obrazová mapa → rastrová grafika, obrázek popsán mřížkou pixelů

Formáty: PNG, BMP, GIF, JPEG/JPG - data uložená stejně, akorát je vyrái třeba jiná firma a proto jsou v trochu jiném formátu

snaha o efektivitu → **komprimace** (zjednoduššíme zápis : „hele teďka bude 6 bajtů se stejnou barvou), používají se dost složité matematické algoritmy (obzvlášť JPEG).

Takže máme informace v hlavičce,

ALE! Potřebujeme vědět jaký ten soubor to je : **prvních n bajtů je signature/magic** a to nám říká, co je to za formát, abychom věděli, co ty zbylé bajty v hlavičce znamenají

Offset souboru – kolik místa zabírají data o souboru a kolik samotná data v něm

BMP má dvě hlavičky – první základní informace – 2 bajty

v druhé hlavičce je šířka a výška a počet bitů na pixel (32 s alfa kanálem)

(knihovna ZenGL – knohovna pro freepascal/lazarus, protože je problém zobrazit ta data skutečně jako barvy. ZenGL.dll – soubor s knihovnou, zglHeader.pas) v proceduře zgl.Init je nekonečný cyklus volající proceduru Update a Draw, odpovídá zobrazování her, čili zobrazujeme ten obrázek pořád dokola /neefektivn, ale stačí nám

knihovna musí vědět, jak se jmenuje ta naše procedura Update a Draw.
Někde v paměti je od nějaké adresy kod procedury Update/Draw – tuto adresu získám pointerem)

příklad :
snažíme se o vykreslení obrázku na konzoli (pascal)
číslem v šesnáctkové soustavě se předává jenom barva, další parametr je číso alfa kanál (255 max neprůhledné) → hledám 3 bajtová čísla
načtu si celé 4 bajtové číslo z adresy pixelů a vymažu poslední bajt pomocí **00FFFFFF (and)** vykreslím, ukazatel na pixel posunu a pokračuju.

v příkadu každý pixel vykreslujeme jako čtverec 5*5, aby byly dobře viditelné

ALE! Ještě zbývá jedna věc: Data jsou zarovnaná na čtyřbajtové adresy , čili pokud počet bajtů na řádek není dělitelný 4, je třeba vložit ještě nějaký padding)

ZVUK

Z fyziky víme, že zvuk je chvění vzduchu o nějaké frenkvenci
→ Zvuk můžeme reprezentovat jako **graf (výchylka)**.

Změna frekvence → změna hloubka/výška tónu. V pravidelných intervalech si zaznamenáváme amplitudu - jedntolivé sample

! ! Je třeba vědět:

- bit/sample (obvykle 16b, signed) ,
- jaká je přestávka mezi jednotlivými samply (sample rate, typicky 44 kHz) - tohle je přesně formát na CD (2kB)

typy: WAV - nejjednoduší způsob uložení samplů, FLAC, MP3 (jiné kompresní algoritmy, jinak poskládaná data, jiná sample-rate)

Saple rate – jak často zazanemenáváme výchylku → způsobuje zkreslení

TEXT

TEXT
1 znak – n bajtů
kodování - jak dlouhý je jeden znak, co které číslo znamená
pevná i proměnná délka znaků

text se čte tak, jak je to uložené – ***na nižší adrese jsou znaky, které se čtou dříve***, máme li tedy vícejazyčný text (kde x, y je text v abecedě píšící zprava doleva):

Intel xxxxxxxxxxx IBM yyyyyyyyyyy

jak se čte mezera ?

ASCII (american stadart code for info infer) – tabulka, kde každému kódu (číslu) přiřazuje znak (písmeno, znak, číslice)

ve skutečnosti(historicky) je to 7bitové kódování (0 – 127) - A-Z, a-z, 0-9, základní znaky (#&…)

víme že A-Z jsou za sebou tedy : A= x → B = x+1, C = x+2 ...ale není to jasně dané kde přesně to začíná

Code pages – tabulky s popisem, které přesně číslo patří čemu

V současné době máme 8bitové bajty → velké množství varinat pro hodnoty 128-255

! ! dokonce i pro jeden jazyk existuje více nekompatibilních code pages (například čeština)
ISO Latin2 (8859-2) – windowsy tomu říkají 28592
(DOS) Latin2 (852)
Win 1250/CP1250

byla snaha vytvořit jednouniverzální kódování pro všechny znaky všech abeced
- **UNICODE** (32-bit) - → vejde se tam tedy i klingonština, čínština, smajlík,
v současnosti není ani většina kapacity využita ovšem **0 – 127 odpovídá ASCII** od 0 - $FFFF jsou všecky „normální abecedy“

UNICODE má různá kódování :
UTF-32 - je neúsporné (každý znak 4 B)
UCS-2 - jenom do $FFFF (takže už se tam nevejdou smajlíky)
UTF-16 – proměnná délka (běžné jsou 2B, méně běžné 4B), takže se to střídá :

V unicode $D800 - $DFFF nemají přiřazená písmenka, a používají se pro různé markery (surrogate) – máme volitelných 11 bitů z toho rozsahu (pro každé číslo) - protože první bity jsou jen označení, že je to surrogate a ty „zbytky“ složíme do čísla/znaku v rozsahu $10000 - $10FFFF

!! UTF-16 i UTF-32 lze kódovat jako little endian i BE
UTF-8 (1Bx2Bx3Bx4B) znaky ASCII 0 – 127 vypadají stejně jako ASCII a vejdou se do 1B, ostatní znaky se kódují min. 2B (na rozdíl od UTF 16, kde mají všechny většinou stejnou délku)

Jak poznám, co bajty reprezentují ?
Pokud mají nejvyšší bit nastavený na **1**, tak je to sekvence. Počet nul za 1 u prvního bajtu říká, jak dlouhá ta sekvence je, u ostatních bajtů sekvence je
10xxxxxx

POZOR! Mám-li reprezentaci UNICODE, tak musím ještě rozpoznat, o jaké kódování se jedná.
FF FE → UTF 16, LE
FE FF → UTF-16, BE
EF BB BF → UTF-8

používá se často při ukládání souborů na disk, ale není to povinné
text v paměti obvykle bez markerů (předpokládá se defaultní kódování , které na počítači používáte)

ALE! co když nevím, že UNICODE ? Pak **nepoznám**, jelikož FF FE je prostě nějaké písmeno v ASCII rozšíření

Z n bytů mi vypadne kód pro znak a někde je **tabulka,** kde jsou **atributy** pro každé číslo z rozsahu například: je to písmenko ?, je to *white space* ? ($20 – mezera, tabulátor), je combinnig character ? řídící znak ?

Combining character znak, který nesmí ležet sám v textu ale doplňuje předchozí; například čínština – základní znak a doplnění tahů, písmenko a čárka nad ním)

Řídící znak například RLE (text za tím se píše right to left)
Ale pro některé běžné znaky existují zkratky bez combining character.
ALE : nemůžeme se nyní na rovnost znaků ptát jako na rovnost bytů.
Pořadí více combining characters není definované.

Modernějších jazyky (C#/ Java) UNICODE umí a pracují s nimi standartně.

KONCE ŘÁDKŮ

CR (cariage return) a LF - konce řádků (line feed)
vychází z tiskáren založených na psacích strojích. (vysvětlit)
Psací stroje mohou i za rozložení kláves. (aby nedocházelo k zaseknutí ramínek) → snaha o to,aby se psalo co nejpomaleji.

Konce řádků:
CR + LF - Windows
CR – Unix
LF -Apple
zavedené nově v UNICODE
LS - line separator
PS – paragraph separator

DÉLKA TEXTU

Textový soubor obsahuje jenom text, ale typicky máme v souboru i jiný data a je třeba rozlišit, kde text končí a začíná a jak která data interpretovat.
Existují reprezentace :
- **prvních n bytů je číslo, které říká délku textu** (typicky v bytech)
  - ***Nevýhody:*** napevno určený rozsah délky (nelze vytvořit delší řetězce) ,
    je třeba vědět, jak velké je to číslo ukazující na délku,
    endianitu čísla
- **null terminated strings** - ve všech kódováních je znak s kódem nula speciální (nemůže být v žádném řetězci), můžeme jím ukončovat (například v UTF-16 BE je to 00 00)

výhoda – mohou být libovolně dlouhé,když řetězec zkrátím od začátku (posunutí pointeru) tak se nic neděje.
nevýhoda – nevím délku dopředu. Naopak ztrácení z konce je lepší u délky.
někdy můžeme narazit na kombinaci obou variant.

(Buď musím vědět všechny informace o délce, nebo kódování, abych tu nulu poznal

Mějme pole znaku char (8bitový znak) a máme metodu, která pro pascalový string od indexu nula vyplní jednotlivá písmenka a za poslední písmenko to dá nulu → nul-terminated string a pak vytištění pomocí pointerů

Z TOHO VŠEHO PLYNE, ŽE MÁM-LI NĚKDE POSLOUPNOST BAJTŮ, TKA NIKDY NEVÍM, JAK JE REPREZENTOVAT.

Harwardská architektura se ve skutečnosti používají jen v **mikrochip (microcotroler/MC)**
periferie :
AD a DA převodník (analog/digital),
GPIO linky - jeden digitální signál (vstupní nebo výstupní) - umožňují nastavovat a posílat nuly a jedničky
co to umí ? Připojení na tlačítko, ovládat diodu/žárovku, poznat polohu točítek
typické použití : například v pračkách je docela těžké „zadrátovat“ ovládání bubnu a vody atd… → jednodušší je dát tam mikropočítač a softwerově nastavit co to má dělat
firmware (software dodaný s počítačem, který funguje po celou dobu zařízení)
na toto je harwardská architektura ideální, jelikož je jednoduchá
výhody : počítače jsou levné a programování je jednodušší než drátování
ATMEL, MICROCHIP – tradiční výrobci například ATtiney - datová paměť 256B(SRAM) a kódová 4kB(flash)
!!! ale i tak potřebuji nějaké permanentní úložiště (například stav před vypnutím) - např 256B (EEPROM – jelikož to chci přepisovat často) **!!!**

Jednočipové počítače – obsahují všechno co čekáme od počítače – umíme číst a zapisovat analogová data, obsahuje procesor i paměť,
najdeme je například v myši a dalších periferních zařízeních
obsahuje pevně zabudovaný program - *firmware*

kód v paměti je ve skutečnosti jenom binární kód, žádný vyšší jazyk, proto to není žádný jazyk založený na textovém popisu. - *strojový kód/maching code*, jeden příkaz = instrukce
vyšší jazyky se přetransformovávají do strojového kódu.

Myš
- **kole**čko a detektory(dva pro každou osu) : zjistíme, o kolik se myš posunula a navíc i kterým směrem
logickou jednotku nechceme mít napevno zadrátovanou, ale napsat program pro jednočip

připojeny senzory a tlačítka a potom to jednou za čas logika vyhodnotí a pošle po sběrnici.
Ovšem předpokládejme, že ten jednočip neumí pracovat se seriovou linkou.
mějme tedy tři 8 bitové procesory (GPIO) – pořídíme si 24 bitový registr (resp 3 registry po osmi bitech) a každý bit reprezentuje stav jedné z linek.
(2*8 vstupních a 8 výstupních : 1. registr : 0- 5 bit pro senzory, 2. registr : tlačítka 0 – 2)
máme dva senzory pro každou osu + kolečko; prostřední, pravé a levé tlačítko

A si budeme potřebovat metody metody: read(cislo registru), write(cislo registru)

musíme si dát pozor, abychom neměnili data během odesílání !!
Potřebujeme 2 algoritmy – jeden počítá(čte) změny a druhý odesílá hodnoty po sběrnici. Oba dva mají běžet neustále (nekonečné cykly) → **ALE!** musíme dělat jedno nebo druhé, ale jelikož je ten první nekonečný, tak se k druhému nedostaneme
Řešení: lépe než dva oddělené nekonečné cykly proložíme kroky obou algoritmů do jedno cyklu.
Algoritmus B : máme čtyři proměnné pro 4 bajty a potřebuju vědět kolikátý bajt a bit z něj posílám.
Vyberu a pošlu.
Potřebuju aby mělo správnou rychlost pro RS-232 linku (1/1200 s pauza)
algoritmus B funguje pro všechny typy myší pro stejnou sběrnici
algoritmus A: 1) nechci hned po detekci zapisovat, protože data se třeba ještě nestihla odeslat, je třeba to někde uložit.
2) pak se ta změna přenese. Porovnávám to s minulým paketem a pak řeším, zda to bylo +/-1 nebo beze změny.
algoritmus A upravíme třeba pro optickou myš, ale netřeba měnit B

Ale! : ve skutečnosti chceme krok algoritmu A dělat častěji , abychom i rychlé změny pohybu myši dokázali detekovat (aby se toho moc nestalo během algoritmu B) : (tedy 1* B, 4 * A….)
(dáme menší zpoždění - delay 1/1200n a provedeme to n krát) *viz zdrojový kód*

CHTĚLI BYCHOM : aby se dalo s myší komunikovat oboustranně, abychom tam mohli například poslat novou verzi firmware.

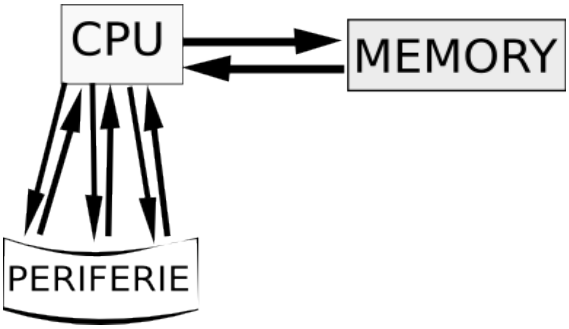
- Některé jednočipové počítače mají nějaký příkaz procesoru, který celý blok datové paměti

přepíše do kódové paměti (čili ne nějak libovolně na libovolná místa)

- Druhé řešení je spouštět kód přímo z té datové paměti a neukládat ho do kódové paměti

→ tento problém nám usnadní von Neuman architektura

Von Neumann architektura – **není rozdíl mezi datovou a kódovou pamětí**
CPU → paměť (ve které jsou data i kód(zápis i čtení), RAM)
I/O – periferie



V paměti je zase jen posloupnost bajtů a my mu řekneme, některá data chápej jako kód.
JAK ? používáme pointer (pProc = procedure; ← *definice pointeru na proceduru stejné funkce)* takže příkazy vypadají stejně, jenom ty procedury budeme číst z jiných míst paměti !
Tedy když dostane pointer na proceduru, tak se začne na ta data koukat jako na kód.

POZOR v harwardském systému by pointery nefungovaly, protože kódová a datová paměť mají oddělené adresové prostory **!!!**

POZOR jde toho také využít – například některé bajty, které se normálně reprezentují jako obrázek (divnej obrázek) ve skutečnosti jdou interpretovat jako kód a šíří se tím viry → ochrana některých částí paměti (*NX = No execute, XD – execution disable, DEP – data execution prevention* – ve Win – tam jde jen o tom, že to mohou spouštět jen vaše aplikace, cizí nejdou potom spustit, jinak nemusí jít spustit žádné)

TROCHA HISTORIE

16 bit adresovýprostor, 64kB DRAM
UNIVAC – první komerční počítač – (1951) 150tis. – 1,5 mil. dolarů, velké asi jako posluchárna, předpokládalo se, že na světě není místo pro více než 200 počítačů

AlHair - 1974 (Intel 8080) – 100 $ ale zase to k ničemu nebylo..nemělo to klávesnici, jenom tlačítka a svítící žárovičky, nepřišlo se na to, k čemu je to dobré

Apple 1 1976 666 USD (cena Steve Jobs), měl klávesnici, dal se připojit k televizi… daly se na tom hrát hry, **VisiCalc** – předchůdce excelu, to začalo dávat smysl koupit si počítač domů (například výpočty daní…),
Apple 2 1977 do roku 1993 procesor MOS 6502

Zjistilo se, že lidi kupují počítače, tak různé firmy začaly vyrábět počítče
Atari 800 – 1979 (500 USD) ….---→ taky používal procesor MOS 6502
Atari 800XE už jen cca 100$

dale populární Intel 8080 a Zilog Z80 – v současnosti se ty procesory používají do jednočipových počtačů

II směr (personal computers)
IBM 1981 PC - myšleno jako kategorie mezi levnými domácími a velkými sálovými. První stál 2000 – 5000 USD
64 kB DRAM ,ale modernější architektura Intel 8088 a dalo se tam přidávat paměť šlo to až do 640kB
Uměly také VisiCalc i hry, ale cílilo to na více profi uživatele.

V 90.letech najednou PC byly stejně levné jako domácí počítače, ale měli mnohem více možností až do současných Core i3… a je to všecko zpětně kompatibilní !!!!

Všechny následující už nejsou jednočipové.

základní deska (motherboard, mainboard) – jedna ze základních komponent počítače - *příklad Atari 800XE (zabudovaná klávesnice)*, osobní počítače jsou více konfigurovatelné

 procesory dříve asi 3000 tranzistorů..nyní asi 3 mld

CHCEME: abychom dokázali využít výkon maximálně - nejvytíženější místo je RAM a CPU komunikace - nejpomalejší je vždycky sériová sběrnice (CPU i RAM rychlejší)

řešení: využívání paralelní sběrnice - více vodičů (n) → n násobné zrychlení + extra vodiče na spešl signály jako R/W, ACK apod.

strobe (obecně)– pokud nějaký náhlý pokles něčeho signalizuje spuštění akcí)

Konkrétní příklad takové sběrnice je C**entronics /IEEE 1264/pararelní port** - pro přenos například na tiskárnu, bývala velmi populární , nemá hodinový signál, tiká asynchronně a když klesne, tak platná data, = *strobe* signál)

(šlo by to trojstavově jako floating stav, ale nejčastěji se používá pull - up rezistor, který, jak víme, vytáhne hodnotu nepřipojených vodičů na 1)

MEMORY BUS

CPU vždy master a paměť vždy slave a je to point to point propojení → nepřenáším adresu zařízení (je jasné, odkud kam to má proudit) , ale je třeba adresa bajtu, který chceme – pořídím si další adresový vodič (m adresových vodičů → 2^m adres) + CL + R + W signály

hodinový signál by měl být synchronizovaný s taktovací frekvencí procesoru, aby ten přenos fungoval.

první takt – nastavím adresu a nastavím R

další takt - před zvednutím signálu připraví paměť data a CPU je načte

při čtení je to jednodušší, neboť lze zároveň poslat adresu a po datových vodičích poslat data → zápis a čtení mohou mít různou délku, proto se to to normalizuje (obě transakce se nechávají stejně dlouho).

Je třeba další vodič, který říká začátek transakce a pamatuje si adresu (**ALE** - Addres Latch enable) *(abychom poznali, jestli je to první nebo druhý takt)* – je 1 jen pro první takt s platnou adresou, jinak je nulový

potom se **sloučí** dohromady vodič/signál **read a write** (např 1 = R, 0 =W a celé dva takty se nezmění)

je rozdíl mezi cyclus x transfer

(cyclus je taktovací frekvence sběrnice a transfer je počet přenosů za nějaký čas)

half duplex - buď se všechna data čtou nebo všechna zapisují

(??? co jsou out of bend signály ??)

zvětšením počtu vodičů se zvyšuje rychlost komunikace

SNAHA O UŠETŘENÍ VODIČŮ :

multiplexování - některé jsou čistě datové/adresové a některé jsou společné

Samozřejmě nechceme procesory pro konkrétní paměť, ale univerzální. Proto mezi memory a CPU vložím *memory controller(řadič paměti)* – zajišťuje překlad komunikačních kanálu

SRAM a komunikace

mějme n datových a n adresových vodičů
všechny signály jsou opět negované.

Další signály :
OE – output enable – když je vypnutý, tak je paměť odpojená (odpovídá signálu R)
WE – write enable – data jsou platná, zapiš si je ! :)
když je vidět hrana, tak to automaticky bere jako začátek transakce a měří si (ve skutečnosti je to zadrátované) zpoždění, ve kterém si připravuje data atd...
proto musí sedět časování sběrnice a paměti, aby neprošvihla data vystavena CPU během hledání adresy - toto souvisí s *acces time* té paměti a je možné vybrat si mnoho typů paměti pro danou sběrnici a potom ještě třeba vybral *řadič*

řadič paměti : Trochu „univerzální“ překladová pomůcka. Jak uvidíme, zajišťuje, aby se CPU a paměť po sběrnici domluvily navzdory časovém spoždění, rozdílu v adresaci apod.
Zde : Na začátku zapne OU a je možné, že ř. upravuje časování (třeba pokud je paměť moc rychlá (potvora!) tak jí chvíli drží odpojenou i když už má připravená data)
v multiplex řeší překlad vodičů (???)

Paměť obvykle nepracuje na hodinový signál, ale je tam CS/CE (chip select/ chip enable) – je-li CS vypnutý, tak ignoruje naprosto vše(ode všeho se odpojí), když ne, tak se připojí a sleduje, co se děje.

Sběrnice - 16 bitů adresových, 8 bitů dat (MOS), často se to zkracuje na 8 bitová sběrnice (myslí se rychlost ne počet adres)
někdy se přenášejí data dvakrát za takt (při vzestupné a sestupné hraně platí) tzv. *DDR* (double date rate) → dvojnásobná rychlost
….....

DRAM a komunikace

tady se vyskytují další komplikace : ve skutečnosti nejsou paměťové buňky lineární,**ale v matici.**
To platí i pro SRAM, ale ta měla tolik slušnosti, že se tvářila lineárně.
CO Z TOHO PLYNE ?
Je třeba dostat číslo sloupce a číslo řádkům (nemusí být čtvercová ale je to časté - pak se bity adresy rozdělí na půl) – tento překlad řeší řadič (ale data jsou stále kompatibilní, ty vedou rovnou CPU - DRAM)
druhý problém : paměť **nezpracuje najednou číslo řádku i sloupce**, takže to má tři fáze (a tři signály) : RAS (row adress strobe) klesne a příjde číslo řádku, CAS (columm adress strobe) klesne poté a pošle se číslo sloupečku , data
opět se ušetří vodiče, protože stačí mít jeden kabel, který zvládne přenos většího z čísel m a n
Ale za 1 cyklus se musí stihnout připojit paměť a přenést a zpracovat adresu → časování je opět velice komplikované :'(

řadič tedy zpracovává i adresové signály.
i DRAM mají CE signál, kterým se dají odpojit
u DRAM je,jak víme, potřeba refresh, který ale netřeba řídit z CPU, stačí nám říkat čísla buňky

(případně řádku) a paměť si to sama obnoví – toto může dělat na pozadí řadič

JAK JÍ DÁME VĚDĚT ŽE CHCE REFRESH ?
CAS before RAS refresh- normálně to klesá naopak, tedy je takto jasné, že bude následovat refresh s číslem řádku z adresy

počty vodičů :
adresové – odpovídají velikosti adresového prostoru (n vodičů pro 2^n paměti)
datové – větší počet zrychluje přenos dat ***alfa*** (kolik je tam slov), ***beta*** (počet vodičů = velikost slova té sběrnice)

příklad : 1Kx8 = 1KB, 1Kx4 = 512 B
pozn. : *dříve existovaly sběrnice s 16b slovem, ale ve skutečnosti může být i jinak velké , pak existují DWord (double word, 32b), QWord – opět spor mezi významy, bere word, DWord atd jako konstanty*

ALE !! vzhledem k tomu, že se může lišit délka slova, tak normálně chápeme adresu jako adresu bajtu (z pohledu CPU), ale pokud se adresa slova liší od 8 bitů, tak je to pro paměť adresa slova!!!. Toto je to potřeba převést

řešení:
Příklad 1 mějme 8bitovou sběrnici, ale dva 512B moduly. Potom nám každý vrátí jedno slovo a velikost nám sedí (paměti pracují paralelně)

Příklad 2 máme 1kb paměť ale dáme tam 2 1Kb paměti a aby to bylo rychlejší, tak chceme 16bitové paměti. Šlo by použít podobný trik, například všechny liché bajty v jednom modulu, druhé v sudém modulu.
Ovšem najednou máme 2kB adres a jen 10 adresových vodičů a je třeba to správně připojit – adresu vydělíme 2 (neboli SHR – posun doprava, což je posun k nižším řádům, potom lze číst jenom dvoubajty které jsou na paměti dělitelné = dvěma (tedy které jsou zarovnané)

10. vodič s 9., 9. s 8. ….

Příklad 3 : co když to ještě zrychlíme ?
Mějme 32 datových vodičů a celkem 4 kb, máme 12 vodičů, tedy dělíme 4 (2x posun doprava a dva nám zbydou) → opět se podporují jen zarovnané adresy

JAK TO BUDE V PROGRAMU ? CO KDYŽ DÁM DO POINTERU LIBOVOLNOU ADRESU ?
řešení:
- CPU ví, jakou má sběrnici a paměť, takže vyhodí chybu (bu to bude fungovat divně nebo to vyhodí vyjímku)
- typičtější : vy můžete říct jakékoliv adresy, a on to obejde :
  čtení 4B z adresy 3 :

0 1 2 3 4 5 6 7 8 9 10 přečte z adresy 0 a z adresy 4 a potom ty přebytečné bajty zahodí a dá nám to to co chceme

i když procesor podporuje *missaligned read*, tak to bude fungovat, ale takové čtení bude trvat **dvakrát tak dlouho**

CO ZÁPIS ?
Ještě složitější. Načte 4 B a 4 B, pak si zapamatuje to co tam bylo na místech, kam jsme nic nechtěli, pak zapíše změny a to co zbylo

obvykle ale máme stejný program a chceme ho pustit na různých procesorech
2 B slovo

4 B slovo

2B na 4B zarovnání : jediný problém je, když to není v jednom slově (3 správné možnosti) , podobně při 8 bajtech --- všechny správná zarovnání dvou bajtů ze 4B jsou v pohodě i ve větších + něco navíc, tedy je dobré zarovnávat na velikost dat

Příklad 4 : 1kB modul ale 8 bitů sběrnice. Ale cokdyž máme 4kB paměti a prostě nemáme 4kB paměťový modul a nechceme 32 bit sběrnici, ale chceme 12 adresových bitů, ale do paměti jich vede jen 10

rozdělíme si to na čtvrtiny (memory bank) a pak 10 bitů adresy je v pohodě a ty horní dva určují v řadiči která čtvrtina to je, tedy kterou banku má zapnout.

a co když máme 2kB modul složený ze 2 1kB bank – bude globální CE a každá má CE signál celého modulu a potom ještě jeden signál na výběr banky
spodní dva vodiče vedou do řadiče - jeden bude CS modulu a druhý banky

číslo banky sedí s desátým adresový vodičem - lze tam připojit i ten bank select signal

Příklad 5: co když nevíme, jestli je 2kx8 rozdělený na banky – víme že tam je 11 adresových vodičů a splývá to s předchozím řešením, akorát řadič paměti má jednodušší práci, jeden kabel není nikam zapojený (pro 1kB)

kdybychom chtěli 4kB a 16bitovou sběrnici, tak dáme dva moduly jako jednu paměťovou banku a vždy dva se zapínají současně, vybírá se která banka se vybírá dá se to tedy využívat a kombinovat všelijak

Příklad 6: 8 bitů data, mějme 12 bitový prostor a 3 * 1kB modul, pokud se někdo bude ptát na 4 modul, tak se prostě nic nezapíše nebo nepřečte (datové vodiče budou ve floating stavu, když bude floating stav i na CPU tak vrátí náhodnou adresu, pokud je tam pull-up/pull-down rezistor tak se to stáhne do nuly a vrátí nulové nebo jedničkové bajty (FF))

POUČENÍ : velikost adresového prostoru CPU a paměti se nemusí rovnat.

16 adresový vodičů (64kB adres) ovšem ne každý má asi 64kB RAM, někdo má jen třeba 4kB
je možné ty další adresové vodiče ignorovat – **ALE !!** pokud spodních 12 bitů dává smysl, tak to

dál neřeší, potom máme jeden bajt z pohledu CPU na 16ti adresách (aliasing). Potom se těžko zjišťuje i kolik máme paměti.

řešení: CHCEME NA ZBYTKU MÍT FLOATING STAV – zavedeme všechny vodiče do řadiče a pokud nejsou horní bajty nulové tak se nic nezapne, ale pokud jo, tak to zapne správnou banku.

Ideální by bylo, kdyby si řadič uměl sám zapojit správnou přechodovou funkci.
Běžne se to dělá : na modul se dá EEPROM a zapíše se tam 256 bajtů informací - kapacita, časování, moduly, … přes I2C (tady se jí říká SPD – serial presents detect) , tak tam bude vést jedna I2C sběrnice, kam ty EEPROM napojíme a je třeba akorát aby každá měla jinou adresu (a víme ,že u ěchto pamětí je možné si tři bity vybrat - podle toho, do kterého slotu je to připojeno).
Je-li v zapojení do slotů díra, tak si řadič adresy přelož, aby byl adresový prostor souvislý.

Pokud bychom chtěli použít toto pro DRAM paměti, tak všechny adresové vodiče povedou
DRAM – adresové vodiče všechny do řadiče
DRAM – chceme udělat 4 banky, tak pro každou banku je třeba dva bity čísla pro každou banku

Už jsme zjistili že pro komunikaci CPU a paměti je dobré mít řadič.
Řadič nám sběrnici rozdělí na dvě části :

- local bus/FSB(front side bus) (CPU – řadič)
- memory bus (řadič – paměť)

Víme, že je stejná šířka datové sběrnice na obou stranách.
Ovšem obecně je možné mít obecně rozdílný počet adresových vodičů a překlad zajišťuje řadič paměti.

Lepší než ignorace přebytečný vodičů je, že to zpracovává (tedy nebere adresy, které mají na místě vyšších řádů než je rozsah sběrnice něco jiného než nuly.)

VELIKOST ADRESOVÉHO PROSTORU NA STRANĚ PAMĚTI VĚTŠÍ NEŽ NA STRANĚ PROCESORU ?
Jak se něco takového stane ? Máme třeba nějaký starší počítač, ale chceme zvětšit paměť při zachování procesoru.
Příklad 1 : CPU má 11 vodičů (uadresuje 2kB paměti) a deska podporuje také 2kB adresového prostoru a máme 1kB moduly (rozdělíme to na dvě banky). A chceme 4 kB paměti.

řešení:

Bylo by možné měnit přechodovou funkci řadiče tak, aby ty samé adresy vedly pokaždé na jinou část adresového prostoru na základě nějakého signálu. Stačilo by přepínat mezi horními a dolními 2kB (*bank switching*).

ALE co když leží nějaký program na dolní půlce a chce se dostat k horní půlce ? Tam se přepne a pak se najednou neví, co se děje, protože už k němu není přístup. :o

řešení:

Proto se preferuje přepínání po částech. Například by spodní 1kB mohl být napevno do nejdolnějšího modulu. A určíme základní a rozšířenou část paměti. A pak bychom mohli mít dva kontrolní signály. Jeden říká, jestli se koukáme do externí nebo základní paměti a druhým eventuálně vybereme, kterou část externí paměti používat.

Přímo pro přepínání paměťových bank se to už nepoužívá, ale přesně takhle to funguje v Atari 800 XE při přechodu na Atari 130XE

VNITŘNÍ ARCHITEKTURA CPU

CPU umí zpracovávat strojový kód *(posloupnost binárních instrukcí).*

Operační/OP kód – n bitů, které identifikují následující instrukci, za ní následují argumenty, kterých může být obecně libovolně mnoho

CPU od sebe odlišuje **instruction set/instrukční sada** – množina podporovaných instrukcí.
Typicky jsou tam proceduty pro čtení a zápis, kde jsou argumenty adresa, pak jsou instrukce pro změny interního stavu (tyto stavy podobně jako u light senzoru jsou uloženy v registrech, složitější procesory mají registrů nejvýše „malé stovky“), typicky jsou registry stejně velké. **Velikost registrů = velikost slova procesoru.** *(pak mluvíme o n-bitovém procesoru, pokud je velikost slova n).*
→ s definicí instrukční sady tedy souvisí i množina registrů, které musí mít →

tyto dvě věci dohromady se nazývají **ISA – instruction set architecture.** Pokud máme strojový kód pro určitou ISA, tak bezpečně funguje na všech CPU se stejnou architekturou.

Zdrojový kód je tedy posloupnost instrukcí, které jsou za sebou uspořádané v paměti podle toho, která instrukce se má provést dříve.
Aby procesor věděl, co má provádět, tak je třeba, aby měl ukazatel na místo, kde se zrovna provádí kód. Tuto adresu má ve spešl registru - ***IP*** *(instruction register)/****programm counter*** – obecně je to n bitový registr s adresou instrukce, která se zrovna má dělat. Po zpracování instrukce se IP zvětší o velikost instrukce, ať ta instrukce dělá cokoliv (třeba nedělala nic).

Některé procesory mají takový překladový algoritmus, který vygeneruje jinou adresu a tu teprve pošle po sběrnici. Pak mluvíme o

- fyzická adresa *physical adres* – chodí po sběrnici,
- logická adresa je ta skutečná uložená

Nyní se podíváme jak to funguje v procesoru konkrétně :
Procesory které budeme používat : 6502, 8088

6502
- 8 bitový procesor, ale uadresuje 64kB, má 16 bitové adresy.
bitové)- register se jmenuje *program counter*
- je tam právě ***výjimka,*** že counter je 16 bitový (ostatní registy , aby se tam dala zpracovat celá adresa.
- standardně má 1bajtový OP code a 1-2 bajty instrukce

8088
- zjednodušená varianta 8086 a mají stejnou instrukční sadu, 16bitové registry
- 8 - bitovou datovou sběrnici *(takže čtení je douhé na dva takty)*
- Má 20 bitový adresový prostor (až 1 MB adres). Ty adresy se tam nevejdou a výjimka tam není, všechny registry jsou stejně velké, takže tam musí být dva (16+16bitů)
1. polovina je tzv. *segment* a druhá je *offse*t (segment:offset)

Máme tedy celkem 32 bitů adresy – logická adresa je širší než fyzická. Ale to nechtěli (takže to není rozšiřitelné)
JAK TOMU ZABRÁNILI ?
Vezmeme číslo segmentu * 16 + číslo offsetu *(celkem 20 bitů)* a to se pošle jako logická adresa.
ALE!
a) mějme adresu AAAA:0000 *(protože 16 = 2^4 tak násobení 16 je posun o 4 doleva)*
potom výpočet je :
AAAA0
0 0 0 0
….....
AAAA0

b) vezmeme jinou adresu A000:AAA0
A 0 0 0 0
AAA 0
….....
AAAA0

Uups, ***stejná fyzická adresa :o*** vede to k aliasingu. Ty segmenty se zakrývají po 16 bajtech (???)

Ale programy v té době toho dovedly i využít.
Potřebujeme tedy dva registry - **CS** - *code segment* a **IP** – *instruction pointer*

Jeho instrukční sadě se říká X86, ale existují i jiné instrukční sady se stejným jménem proto se u této zdůrazňuje šesnáctibitovost: ***X86-16***

Další instrukční sady :

Intel8286 – už má 24 bitů adresový prostor (16 MB), ale adresy jsou u starých programů pořád malé. Je třeba změnit přepočet.
ALE když se změní obecně, tak nebudou fungovat starší programy.
Řešení: Proto podporuje dvě instrukční sady. Standartně bootuje jako ***reálný režim*** (zpětná kompatibilita), ale některé OP kódy nebyly použité, takže se mohly využít pro novou instrukci → zavedli jednu spešl instrukci, která je přepne do nového režimu - ***protected mode-16*** , který umí využít až 16 MB paměti.

Toto je opět interní stav procesoru, takže existuje registr ***CR0***, kde je zanesen režim procesoru.
ALE zase to bylo navržené jen pro 24 bitů **:(**

Intel 386 - zase chtěli 32 bitů i registry i adresový prostor, a chtěli aby to podporovalo zpětně → tři režimy (***protected mode 32***) a adresa je lineární 32 bitové číslo (už žádné segmenty) → register ***EIP*** – obsahuje celou 32 bitovou adresu celé prováděné instrukce
taky už měl 32 bitovou sběrnici
takže **dnes je X86** právě tato sada *(také zvanáIA32 nebo Intel 32)*

Intel pentium Pro měl 64 bitový adresový prostor, ale instrukce se zachovaly(32 bitové registry i instrkční sada) a 36 bitová paměťová sběrnice
nastalo drobné rozšíření instrukční sady, kterým se dá ovládat ***physical adress extension,*** kde zapneme *bank switching*,aby bylo možné využít celý prostor případné paměti.

Opteron – 64 bitové registry i data a 40 bitů adresa (opět horních 24 bitů je nulových)
zavedl se čtvrtý nezávislý režim X64 , ale opět startuje v reálném režimu z 80.let
RIP (register instruction point) a je 64 bitový
má to opět všechny registry které měly ty předchozí.
ALE Registry jsou nejrychlejší typ SRAM paměti(takže i ***nejdražší)***, tak by bylo blbý aby tam byly některé plonkové. **Řešení:**Takže když to je v 32 bitovém režimu, tak se to kouká jen na 32 spodních bitů (podobně pro 16bitový režim) a využívá registrů pro 64 bitový režim.
X86-64 AMD64, Intel 64, NM

existovala sada IA64, ale to není tato sada !!

existuje spoustu dalších sad, třeba: ARM (32 i 64) a spousta jejich verzí...
MIX (pevná délka instrukce, 4 bajty), 68K – CPU od Apple
Je velice těžké napsat něco, co by fungovalo na všech
u všech předchozích jsou OP i instrukce různé délky /max 15 bitů)

INSTRUKCE ASSEMBLERU

Typické operace, které se vyskytují:

6502

- no operation – nic nedělej a zvětši pointer na instrukci
- jump/branch– nastaví novou hodnotu do instruction pointu (4C _ _ *2B adresy, která se tam má přiřadit*) –; toto je tzv. ***přímý skok (direct jump)***
- někdy chceme ***nepřímý skok*** – řekneme kde v paměti najde tu pravou adresu (6C), v podstatě dereference pointeru
  (přímý i nepřímý skok je JMP ale argument se dá do ()
  čekáme že se jedná o ***absolutní skok,*** ale ve skutečnosti to znamená relativní skok. Adresu chápe jako znaménkové číslo ve dvojkovém doplňku a přičte to k EIP (záporné číslo skáče zpět), taky se přičte velikost té instrukce
  + taky existuje EB s dvoubajtovým argumentem (skáče o 127 bajtů oběma směry)
  někde je přímý skok implementován pomocí nepřímého skoku

	No operation	jump	Nepřímý skok		
6502	EA	4C *JMP* *cisloadresy*	6C		
X86	90	E9 JMP xx xx	FF25 [argument]		

!! Nyní se musíme ale taky zajímat o endianitu čísla, protož nevíme, v jaké je ta adresa. Běžně je napevno daná instrukční sadou. Všechny co tu máme jsou little-endian.

Když takhle napíšeme instrukce za sebou, tak je dost nepraktické je psát ve strojovém kódu. Proto se zavádí textová reprezentace : ***assembler.***
Ovšem CPU tyhle sprostý slova samozřejmě taky neumí zpracovat. → Potřebujeme **překladač/compiler**, překladačům assembleru se běžně říká assembler :D
ALE je třeba :

- zdrojový kód překladače dostat do paměti
- způsobit, že se to začne provádět
- že assembler najde nějaký soubor s kódem

Máme tři věci v paměti : zdrojový kód překladače, adresu souboru s kódem, vlastní strojový kód k provádění

Jenomže mi stejně nechceme psát ani v assembleru. Chceme něco univerzálního (assembler není přenosný), chceme programovat ve ***vyšším programovacím jazyce*** – Pascal, C, Java – tyto programy nás odstiňují od architektury.
Píšeme dejme tomu v pascalu. Musíme mít tedy překladač na Pascal. Takovéto kódy jsou ***Source code portable*** – že to půjde na všech typech procesorů
!! jiný překladač pro jiný procesor → jiné strojové kódy. Stejný je námi napsaný kód.

Navíc mohou být různě přeložené, různé překladače mají různou organizace kódu v paměti atd...

Kromě místa v paměti na kód je třeba místo na globální proměnné a konstanty (hned za kódem, vyhrazené místo)
Taky je možné tohle vygenerované uložit do přeloženého spustitelného ***(executable) souboru.***
OBRÁZEK Z PREZENTACE (1:20popis)
Kdyby to byl skok přímý, tak by to našlo to co chceme, ale u nepřímého skoku to na adrese instrukce přičte něco co tam je, vyskočí to, rozbije si to hlavu a nebude to fungovat :/

section souboru - data (globální proměnné atd...) , code/(*z obskurních historických důvodů se tomu říká i text)*

:)UVĚDOMME SI !
Můžeme mít překladač který běží na daném rpocesoru, ale generuje strojový kód pro jiný CPU *(cross compiler)*. Například na jednočipové počítače se nevejde překladač, strojový kód atd.., takže ho přinesu z jiného počítače :)

pro zpětný překlad (abychom ze strojového kódu viděli, co to dělá) lze použít :
Disassembler – vždy funguje (namapovat strojový kód na instrukce assembleru jde)
decompiler – založený na tom, že ví, jak to ten překladač (který přesně překladač) přeložil. Jinak je mnoho možností, jak to přeložit.

Každá instrukce na jeden řádek. Různé sady mají různé assemblery
Intel Assembler , ATIT – funguje také na Intelu.

Programovací jazyky dost často podporují datovou strukturu **zásobník**.
(Data se ukládají uspořádaně a odebírají se ta, která přibyla nejpozději). Zásobník se myšlenkově dá rozdělit na použitou a volnou část – místo, kam můžeme ještě ukládat nová data. Pak je evidentní, že odebíráme a přidáváme podle té hranice mezi dvěma částmi. Proto podpora pro zásobník obsahuje nějaký register SP(stack pointer) kam ukládám ukazatel na vršek zásobníku (*stack top)* – ukazuje buď poslední použitý, nebo první nepoužitý bajt.Tradičně mají procesory právě jeden takový register.
ZNAMENÁ TO, ŽE PODPORUJÍ JEN JEDEN ZÁSOBNÍK ?
Ne! do nějaké proměnné si uložím aktuální hodnotu stack top a do registeru si dám vršek toho druhého zásobníku. *(Tedy vždy pracuji s jedním, ale mohu je střídat)*

Se zásobníkem můžeme provádět dvě základní operace :
Push – přidat do zásobníku
Pop – odebrání ze zásobníku

x64 tam je register RSP
x86 ESP
opět používám trik se zmenším prostoru a doplnění nulami
SS SP (stack segment, stack pointer a zase přepočet)

6502 – 8bitový procesor, **pozor:** program counter má 16 bitů, ale register S je jen 8 bitový, má tedy malý rozsah. Vždy se k adrese ale přičte $100 aby to bylo výše než uplně na konci paměti. (stále jde přepínat mezi jednotlivými zásobníčky)

x86 – příkazy
PUSH *32bitovecislo* 68 + 4B hodnoty - vezme hodnotu a vloží na vršek zásobníku
ve skutečnosti napřed posunu vršek zásobníku o velikost proměnné a pak vložím
!! čekali bychom že to roste k vyšším adresám, ale ve skutečnosti přidávám na nižší adresu. !!

Pseudocode :
SP := SP – sizeof(x);
SP^ := x;

Immediate - vložená konstanta – tedy nikoli proměnná

Push and Pop Variants on x86 (IA-32)

Machine code	Intel assembler	Comment
68 xx xx xx xx	**PUSH** xxxxxxxx**h** (or **PUSH DWORD PTR** xxxxxxxx**h**)	push 32-bits of x (x = *immediate*)
66 68 xx xx	**PUSH** xxxx**h** (or **PUSH WORD PTR** xxxx**h**)	push 16-bits of x
FF 35 xx xx xx xx	**PUSH [**xxxxxxxx**h]** (or **PUSH DWORD PTR [**xxxxxxxx**h])**	push 32-bits from address x (x = *absolute address*)
66 FF 35 xx xx xx xx	**PUSH WORD PTR** xxxxxxxx**h**	push 16-bits from address x
8F 05 xx xx xx xx	**POP [**xxxxxxxx**h]** (or **POP DWORD PTR [**xxxxxxxx**h])**	pop 32-bits and save them to address x (x = *absolute address*)
66 8F 05 xx xx xx xx	**POP WORD PTR [**xxxxxxxx**h]**	pop 16-bits and save them to address x

NECHCI-LI VKLÁDAT 4B HODNOTY ?
existují různé varianty instrukcí (viz předchozí slajd)
68 xx xx xx xx
66 68 xx xx PUSH WORD PTR - *pro vložení dvoubajtů*
někdy nevkládám konstantu, ale mám proměnnou na nějaké adrese a chci přidat jenom její adresu (odkaz) :
FF35 [xx xx] (32 bitů adresy, tedy 4B; dám mu absolutní adresu)
SP:= SP – sizeof(P^) !! *vezme to velikost cíle ne adresy !!*
66 FF 35 podobně ale 16 bitová adresa (***PUSH WORD PTR)***

POP má jenom variantu s absolutní adresou
8F 05 odeber 4Bajtovou hodnotu a dej jí na tu adresu
66 8F 05 odeber 2B hodnotu a dej jí na tuhle adresu
(22:52 POPIS PSEUDOKÓDU ???)

NA CO SI DÁT POZOR
jediná informace o zásobníku je vršek → když budeme moc pushovat tak přepíše další části paměti – ***stack owerflow*** *(přetečení zásobníku)*

když vícekrát zavolám pop než tam je tak ***stack underflow*** – nepřepisuje paměť, jenom čte nesmysly

!!POZOR v zásobníku nejsou ani hranice mezi zadanými hodnotami – když nevím, jak velké ty věci tam jsou a když pak čtu větší proměnné než tam jsou, tak to zase bude dělat blbosti **!!!!**

K ČEMU JE TO DOBRÉ ?
Mám-li proceduru G a proceduru F, která v rámci sebe volá G, tak je tam nějaký „jump back“ (viz předchozí kód)

Co je to přesně „jump back“ ?
Šlo by vyhradit si globální proměnnou na adresu odkud přicházíme.
ALE když je těch volání víc, tak je třeba mít více proměnných. Ale pokud se volají rekurzivně, tak nestačí ani jedna proměnná pro každou proceduru !! (takže je nemohu ukládat do registru)
řešení: Ideální je nová proměnná pro každou proceduru, kterou volám a důležité je také pořadí procedur (a jejich adres) → můžeme použít zásobník :) kterému říkáme ***call stack (****zásobník volání)*

poté si udělám POP z vršku zásobníku a
RET (RTS) – v assembleru
ALE !! Není to tak jednoduché. Zacyklíme se. Chceme dát push IP. Aleprotože to vede na začátek zásobníku, tak tam je PUSH → takže se chceme posunout o instrukci PUSH, to se ale vrátíme na jump a to taky nechceme (ještě ostřejší cyklus), tak se to musí posunout ještě o JUMP ! **:)**

na to je ***CALL (JSR)*** - volání podprogramu (*subrutine)* – push na vršek zásobníku dá adresu instrukce za call (aby bylo kam se vrátit a pokračovat) a pak jumpne na adresu která je v argumentu (???)
call má také varianty (přímé nepřímé volání atd..)
(36:41)
(37:51) + prezentace

Stack frame/zásobníkový rámec – jedna „položka“ zásobníku

!! POP jen posune vršek zásobníku, ale nesmaže to co tam bylo **!!**

co kdybychom měli nějakou složitější proceduru s parametry s dvěma argumenty(2B a 4B) a předpokládejme, že se rekurzivně volá :
Argumenty existují vždycky když se zavolají (je třeba alokovat místo), dealokace s vynořením z volání.
Předpokládejme, že to budeme ukládat do registrů procesoru a pamatovat si, kde to je uložené → **co když nám registry dojnou ? !!**
Registry mají určitou velikost a pak dochází k plýtvání, ale zase jsou rychlé (řádově rychlejší),ale je to komplikované
řešení: mohli bychom to ukládat do zásobníku - je možné v jiném místě paměti mít myšlenkový ne hardwarový zásobník (pamatovat si pointer na vršek atd..).
Nejběžnější je ukládat to na ten volací zásobník, protože je mohu zahrnout do volacího rámce procedury

V JAKÉM POŘADÍ JE TAM BUDU UKLÁDAT ?
Zleva doprava či naopak, na přeskáčku ? Je to jedno, ale musí se dohodnout volající a volaný → calling convention – zahrnuje obecný popis jak se procedura volá(kam se ukládají procedury a argumenty, zarovnání argumentů) *protože tohle není vidět z dat v zásobníku atd...to ví překladač nebo autor kódu*
nejčastější je zprava doleva

#cotodělákurňatenkód
Co vlastně dělá takový kód ?
zapíše na zásobník metodu (adresu), argumenty, posune zásobník...
když dojde na konec rekurze a začne se vynořovat, tak přečte horní 4 B ze zásobníku a posune se dolů, zase přečte čtyři bajty, jenže ouha...**ALE !!** tam byly argumenty metody a ne adresa odkud někam přišel.

Řešení: Všimneme si , že argumenty mají stejnou životnost jako celá volaná metoda/podprogram. Tedy po skončení to někdo musí mazat. Mohl by to dělat volaný, ale to by musel přepsat argumenty adresou a posunout, pak by to fungovalo.
nejběžněji to odebírá volající (za call) se odeberou argumenty ze zásobníku (tedy jen posunu vršek zásobníku i o jejich velikost)

ALE !! co kdybych chtěl s těmi argumenty pracovat ? ==> To jsou pevné offsety od začátku zásobníku.
Proto se to ukládá zprava doleva, aby ten první byl vždy na stejné pozici a pak teprve následovaly ty další argumenty, kterých někdy může být libovolné množství (třeba u funkce write)

kromě argumentů je tam ještě druhý typ dat : ***lokální proměnné*** – stejná životnost jako argumenty
Místo v zásobníku tady ale musí dělat volaný, protože kolik jich je, to není vidět zvenku té procedury.
→
Vidíme tedy, že v každém **stack frame je**

- argumenty
- návratová adresa
- lokální proměnné
- občas si potřejue překladač někam něco uložit a potom to na konci procedury zruší

strojový kód se zkládá z

- kódu na přípravu procedury (alokování proměnných) – tzv. prolog
- samotný kód procedury
- návrat vršek zásobníku zpět (dealokace lok proměnných)
- RET
- uzavření a návrat z procedury – epilog

KDYŽ JE TO FUNKCE ? Lokální proměnná musí přežít zánik procedury, ale stačí jen chvilička (kde se ukládá návratová hodnota je součást *calling convection*)
typicky se to ukládá do nějakého registru, ale je možné že by mohla být větší než registry, tak je možné to předat na zásobník, takže se musí předem říct, že po argumentech nebo před argumenty bude místo na tu návratovou hodnotu.

pro funkci f1 (a,b : longword, var c : longword)
a někde globální x,y : longword
x na adrese 1000 a y na 1004 :

zavoláme a předáme 5, x,y
push 5
push [x]
push 1004
call (na adresu první instrukce)

ALE !! stále jsme ve von Neumann architektuře, a máme paměť DRAM (volatile), takže co bude dělat procesor po spuštění ? Nějaký blbosti ? Mohli bychom tam dodat ROM která je non-volatile, ale tam by nešly proměnné (resp jen pro čtení)
je třeba to zkombinovat: nahradíme část adresového prostoru pamětí RAM a část ROM a jednu část bude mít s kódem a jinou RAM (zapisovatelnou).
Řadič vždycky zařadí ROM na vršek adresového prostoru (on ví, která to je) aby bylo kam případně rozšiřovat prostor RAM paměti.

Ale jak ví, která paměť je ROM a kde má začít ? Procesor to má pevně dané (u 8688 je to FFF0) a my se tomu musíme přizpůsobit.
(IA32 - FFFFFF0) vždycky je to tak, aby se tam vešlo aspoň instrukce jump a ta už může skočit kam chce (u 6502 je na FFFC ale rovnou skočí na FFFFD a pak teprve skončí) (???) někdy se tomu ukazateli na začátek startování počítače říká ***reset vector***

Pro jednočipové je to fajn, ale větší počítače mají více programů, které se střídají. Stále tam máme nějaký firmware, který provede kontrolu hardware a uvede ty zařízení do nějakého rozumného stavu a poté provede nabootování. To znamená, že chce lokalizovat další užitečný kus software. Ten by mohl být na nějaký další optional ROM (tam kdyžtak udělá jump nebo call) Aby věděl, jestli tam taková věc je, tak je například někde nějaké ***magic number,*** které když je v určité hodnotě tak vím, že mám hledat optional ROM.
Atari 800XE - tam je standardně nainstalovaná optional ROM a dá se tam programovat v jazyku Basic. Dají se tam zapojit i další ROMka (cartrige) kde může být něco jiného, a pokud jí to najde tak to ani nepustí ROMku (třeba hry)

TYPY A PRŮBĚH BOOTOVÁNÍ

Cold/hard start – nabootuje a má prázdnou paměť (během vypnutí ztráta napájení volatile pamětí)
warm/soft – FFF....0 udělám jump na ten firmware a tam se nesmazala paměť od posledního zápisu. Firmware si totiž dohodne místo v paměti, kam po úspěšném warm startu uloží dlouhá kostanta a když tam je, tak přeskočí některé fáze bootování a ví se, že proběhl teplý start. Pamatuje si to potom například zdroják v Basicu z minula.
Občas se prý panu profesorovi stane, že ty paměti se rozbijí a zapomenou zapomínat.

Zkusily jsme program v Basicu na výpis čísel od 1 do 100 ale je dost pomalé (oproti aktuálnímu pascalu).
PROČ JE TO TAK POMALÉ ?
Jak se tam ve skutečnosti provádí kód Basicu ? Čekali bychom, že tam je překladač Basicu – ale to by nebylo rychlé spuštění a pomalý výpis (bylo by to naopak)
Ve skutečnosti tam celou dobu běží ***interpret(interpreter)*** – program ve strojovém kódu, který bere jako vstupní data zdroják jiného kódu a provádí to, co příkaz v Basicu říká.

Píšeme v v pascalu překladač na basic :
lexikální analýza kódu – rozsekání na jednotlivá slova
Uděláme si vícerozměrné pole, kde v řádcích jsou řádky kódu a ve sloupcích příkazy
a pole pro všechny proměnné a ještě proměnnou, která mi ukazuje, který řádek probíhá
a proměnná která říká, zda mám skončit a spousta IF, abychom rozlišili, co po nás ten příkaz chce.

INTERPRET – VÝHODY A NEVÝHODY
nevýhody : musí být v paměti s programem a poběží to daleko pomaleji (10-100x pomalejší)
výhody : naprogramovat interpret je jednoduché, ale napsat překladač do strojového kódu je těžké
Atari má 64 kB adresového prostoru – tam by se nejspíš nevešel ani ten překladače :/
zase rychlé je třeba sčítání - to se provádí rovnou a je to rychlejší než kdyby se to překládalo (???)

Jak to funguje ?
1 část ROM je editor kódu a druhá část je interpret
- chci-li nový jazyk a jeho překladač pro všechny platformy tak je to plno práce, ale když napíšu interpret v pascalu (a ten umím přeložit), přeložím ho (to umím všude), tak to funguje.

Obecně by takový počítač mohl mít na vstupu i assembler nebo strojový kód → tak bychom vytvořili emulátor procesoru (něco co se chová jako procesor)
virtuální komponenty (fiktivní komponenty CPU v emulátoru)

Naštěstí už to někdo udělal za nás.
(???) NOP NOP JMP FC06
máme i dvě procedury simulující sběrnici

WOW, najednou umí procesor provádět instrukce ze sady 6502
6502 byl tak populární, že na to je veliké množství emulátorů.
Abychom nemuseli pořád psát adresy, tak je dobré vědět, že ty instrukce jsou za sebou → ale museli bychom vědět, jak je ta předchozí instrukce dlouhá – to nekdy není lehké spočítat, takže v assembleru jsou tzv. labely (slovo:) a pak se odkazuji na ten label

VIRTUALIZACE

Rozdíl mezi simulátorem a emulátorem ?
emulátor se z vnějšku chová jako procesor ale vevnitř je implementován jinak
simulátor simuluje vnitřní chování – časování, chování v okrajových situacích atd...
(ale někdy se tato dvě slova zaměňují)
mohli bychom tam simulovat i všechna zařízení, která v Atari jsou a simulovat tak celé chování počítače – a taky už to někdo udělal za nás :)
Ale i kdyby tam byly všechny součásti, tak to stejně nebude fungovat, je třeba tam dát ještě *memory image* paměti Atari a načíst do pomyslné ROMky (taky už to někdo udělal :D :D)

F5 funguje jako klávesa reset, která tu už není a shift + F5 studený restart , některé další znaky jsou také přemapované
výhoda je, že v emulátoru mohu kontrolovat co se tam vlastně děje (vytvořím tzv, sandbox – pískoviště), takže mi třeba nedovolí zformátovat celý disk nebo podobnou blbost.
Emulátorům počítačů se říká ***virtual machine*** - některé emulují ten samý procesor na kterém běží

Kde se to používá ?
používá se to například na serverech, kde každá aplikace může běžet ve vlastním emulátoru a mám to zabezpečené, protože když se jeden proces zblázní, tak mu prostě zruším celé pískoviště.
(VMware, Qemu– příklady virtual machine)
Dají se udělat emulátory, které jsou částečně překádané ***(just-in-time překlad, JIT překladač)*** - vždycky kus přeloží do strojového kódu - ale vždycky to kontroluje : volám z normálního překladu a občas odskočím do virtual machine a pak vracím zpět
zvládne to rychlost až 1:1, někde běží instrukce dokonce rychleji, protože softwarově zde některé zadrátované instrukce lze zapsat efektivněji (v rychlejších instrukcích)

Taky nemusím překládat strojový kód reálného procesoru, ale překládat do strojového kódu fiktivního procesoru, tzv. ***intermediate lenguage/ IL*** –poižívá to například C# nebo Java
Běží to vlastně ve virtual machine (.NET kde běží JIP – pro C#), takže stačí aby byl přeložený jednou do intermediate kódu a je to přenositelné na binární úrovni, stačí že tam poběží .NET , navíc je tam sandboxing, tedy určitá ochrana.
Java to má podobné a běží tam JVM
Také jdou programy přeložit z jakéhokoliv jazyka do Dalvyk bytecode které běží na androidu (překlad mezi dvěma binárními kódy) – třeba i program v C#

Typicky chci ale bootovat z pevného disku, nikoliv z ROMky - umí to i Atari :
Když se nenajde ani jedna ROMka, tak se provede self-test, kde se testuje, co se děje.
V Atari počítači mohla nebýt ROMka a rovnou to skočilo na self-kontrol (jelikož je možné tam určit, že to vždycky přeskočí tu ROMku a skočí to na self-test)

V téhle době se ukládalo na magnetofonové pásky a typičtěji diskety (floppy disk)což je výměnné (v diskové mechanice) a kapacita je cca 1,2 MB . Vevnitř je kotouček rozdělený na sektory a stopy podobně jako pevný disk
na Atari běžné 130kB (a sektory 120 B velké)

Když zrovna disketa nefunguje, můžeme zase udělat obraz diskety (disk image)
Běžné je že nultá sektor je ***boot sektor*** , kde může být strojový kód.
Konvence je, že existuje nějaká značka , která značí, jestli je tam rozumný kód.
Běžně je tam spešl kód ***(boot loader),*** který ví, kde je něco rozumného a zavolá to (jump).

ALE celý adresový pamět okupuje RAM ovšem ještě je tam ROM a pokud je to na stejných adresách, tak to dá přednost ROM (vznikne nám ***shadow RAM*** – nejde se k ní dostat), proto když se při startu odignoruje (odmapuje) RAM, tak se to přepne do ROM a naopak.

CHTĚLI BYCHOM MÍT NA DISKU VÍCE APLIKACÍ

0. tý sektor je stále boot sektor a zbytek disku se rozdělí na oddíly ***(partition),*** kde každý obsahuje nezávislý kus software a můžeme si pro každý zavést nějaké logické sektory jinak číslované → pak se na partišny můžeme koukat jako na oddělené l***ogické disky*** a každá partition má svůj vlasntí boot sektor.

Ten první skutečný boot sektor je ***MBR (master boot record, master boot sector)*** a je tam kód zvaný ***boot manager.***
Pokud jsou ty disky oddělené, tak nám zobrazí možnosti a pak sám nahraje vybraný boot sektor vybraného programu. *V podstatě tam může být boot loader i boot manager.*
Když udělám warm reset, tak už je nahraná v paměti nějaká apliace a skočí to rovnou na ní.

Toto všecko lze vyzkoušet na emulátoru Atari. Navíc je tam velmi často možnost přerušit provoz (FF1) a přepnout do monitoru virtual machine, kde je vidět co se tam děje, a nastavovat různé vlastnosti. Lze tam připojit i virtuální obrazy disket.

Ale hry jsou pak trochu méně plynulé než na normálním Atari (ačkoliv běží na mnohem modernějším procesoru, neboť se simuluje i celá grafická karta).

Ať se bootuje do hry nebo do aplikaci (editor basic, VisiCalc), tak všechny aplikace musí mít nějaké **funkce pro komunikaci s hardware** (třeba čtení z klávesnice, zobrazení textu na obrazovce), a to samé potřebuje boot loader a boot manager. Čtení sektoru, zjištění pamětí - potřebují bootloadery atd.. obzvlášť když je to složité a nepřehledné jako v Atari, tak je potřeba ***memory map*** *(popisuje,kde je jaká paměť)*

ALE je hloupé aby to každá aplikace dělala znova a na každý počítač zvlášť. Tohle by měl asi přirozeně dělat firmware toho počítače **!!** poskytuje tedy základní funkce pro práci s hardware.

Sám je třeba ani nevolá (i když může i něco vypisovat, například error hlášky)

ALE Abychom to mohli používat, je třeba se domluvit, jaké funkce, jaké názvy, jaké argumenty atd... → ***API*** (application programming interface) - informace o funkcích, které jeden program poskytuje jinému),
!!! na úrovni API se neřeší calling convention, taky se neví jak se ukládají čísla (dvojkový doplněk atd..) → to řeší až ***ABI*** (aplication, bianary interface)
když se mluví o API tak se předpokládá, že je tam implicitně nějaké ABI konvencí známé
v různých typech Atari jsou různé druhy hardware, → různé druhy firmware, ale pokud bude API stejné, tak budu volat jen API a neřeším, jak to implementuje. :)

IBM PC mají taky API v sobě, ale ty funkce jsou jiné, jiná architektura (liší se třeba i argumenty, podle typu adresace atd..) .
Původní firmware v IBM PC se jmenoval ***BIOS (***basic input output interface)
(!! BIOS není obecně API ani obecně firmware !!)

Ve všech dalších verzích Atari se rozšiřoval firmware, ale zachovávalo se API → zpětná kompatibilita (takže třeba také omezení, že to lze uložit při bootování jen do spodního 1kB)
Nyní se začíná prosazovat **UEFI** (*unified extended firmware interface)*. V UEFI už nefungují staré bootloadery.
Protože v současnosti je přechodové období, tak často počítače podporují obojí.

JAK PŘESNĚ VOLÁM FUNKCE Z API ? Mějme dva programy, které se chtějí navzájem volat.
ALE! pokud to překládáme jako celek, tak najednou voláme funkci o které překladač neví, protože je z jiného programu.
Řešení: říct napevno adresy všech funkcí, které API poskytuje a definujeme v API, že tahle funkce je na téhle adrese. **ALE!** když přilinkuju ke zdrojáku ještě RTL knihovnu, tam pak může být procedura která něco dělá například pro dělení nulou, ale je potřeba aby se před začátkem programu nastavila obsluha těchto přerušení (nevolalo to kernel). (???)
Řešení: Ve skutečnosti je náš program vnořený do té knihovny.
z našeho programu generuje metodu **_Main,** kde je tělo našeho programu a v RTL je generované volání na _Main → tak se zajistí, že Main je na volání našeho hlavního programu.

CO JE DÁL V KNHOVNÁCH ?
Další věci, které nás odstiňují od architektury prcesoru (například CPU v Atari neumí dělit) →

ALE! Neustále se rozšiřují a mění implementace firmware, tak je dost těžké, aby to stále vycházelo na stejné adresy.

Řešení: Vyrobit tabulku, která by říkala seznam adres základních funkcí firmware. →
Pak je napevno jenom adresa tabulky a adresy (pointery) v ní uložené se mění a provádíme nepřímá volání (někdy tam nejsou pointery ale rovnou instrukce JUMP) a máme-li potom lepší inplementaci některé funkce, tak potom změním, kam to povede (*hook* – zahákuju nějakou funkci, tedy vytáhnu si jí odjinud)

používá s to často na PC, třeba když je někde nějaká optional ROM a nahookuje se v ní nějaká funkce a pak se udělá RET, takže si všichni myslí, že to nebylo v optional ROM a pak to pokračuje dál.

ALE! stále máme problém, že je někde napevno daná nějaká adresa.
Řešení: procesory mají hardware podporu pro spešl instrukci zavolání tabulky a není tam adresa té tabulky (***použije se takzvané softwarové přerušení*** - ***software interrupt*** – to znamená, že program volá funkci někoho jiného)

Softwarové přerušení : máme někde ***IVT (interrupt vector table)*** plná vektorů přerušení a já nevolám adresu té fukce ale říkám, že chci n tou funkci z tabulky a procesor ví, kde ta tabulka je a najde požadovanou instrukci (bázová adresa je první adresa instrukce)

Ukázalo se totiž že mít tabulku na konci adresového prostoru je dost špatné. Protože adresa nula se občas definuje jako neplatná (nill), tak existuje register, kde je uložená bázová adresa té tabulky a intrukce je furt stejná, akorát se k tomu x4 připočte adresa v registru. Takže pak můžeme registrem přesunout tabulku a pořád to bude fungovat :)

Dost často se procesor dostane do nějaké divné situace (dělení chybou, neznámý OP kód) – vyhodí fault/trop/exception (exception v C# s tím nesouvisí) v takovémto případě bych taky chtěl zavolat nějakou obsluhu vyjímky. Pro každý chybový stav je definovaný vektor přerušení v tabulce (a potom se ukazuje na nějaký vektor přerušení - ***interrupt handler,*** funkce, která to nějak vyřeší)
Když udělám v programu int 0 tak se nepozná, jestli je to fakt chyba nebo se to volalo
Součástí int je call a součástí call je jump a push a pak se posune counter, ale já bych chtěla typicky vědět, která instrukce způsobila chybu → pokud nastane výjimka, tak se na zásobník pushne adresa instrukce která způsobuje závadu.

Podobně se řeší třeba nezarovnané čtení a zápis (je možné nastavit, aby to házelo chybu při čtení/zápisu nezarovnaných adres)

ALE! když pracuji ve vyšším programovacím jazyku je třeba aby za nás ten program propojil správné vyjímky a jejich obsluhu.
Jak přesně se to dělá ?
Překlad totiž probíhá komplikovaněji, než jak myslíme. Celý zdroják nemusí být v jednom souboru. A pak se překládá postupně soubor po souboru.
Potom **ALE!** co když volám proceduru z druhého souboru ? Pak bude problém.

Řešení:

vytvořím object file, přípona .obj (win) .o (unix) (*ale nesouvisí to s objektovým rpogramováním*) a pak je v souboru tabulka, kde jsou opsané nevyřešené závislosti ***(unresolved symbols/references)*** poznamenám tam problematické výrazy a všechna která se na tu neznámou adresu odkazují.

Pak tam bude druhá tabulka, kde budou napsané symboly, které tam jsou definované a lze je použít jinou částí programu.
Aby to celé fungovalo, přijde na scénu **linker** a ten to slinkuje (static linking) → vytvoří spustitelný soubor, který už bude fungovat (tj najde k hledanám symbolů jednoho souboru adresu toho symbolu v jiném souboru, který homá definovaný).

Vznikne nám **library (knihovna)** *.lib / .a* - vytvořený soubor z toho linkování (static library) a jde to zase zlinkovat
→ udělám si soubor se společnými funkcemi pro všecko a pak to vždycky jenom slinkuju.
(1:13 API knihovny ?)
Podobně mohu udělat tzv. ***runtime library (RTL)*** - knihovnu standardních funkcí jednoho jazyka.

z toho je vidět že RTL budou mít různé verze pro různé procesory které pak taky nemusí umět reálná čísla ve floating point a podobně.

dále typicky nechceme zpracovávat paměť ale jen si říkat o dost velkou paměť a zase jí vracet. Tedy chceme to zpracovávat jako haldu (rozdělení do bloků a poznamenání, které bloky jsou volné a které nejsou)
→ takže celkově nám jazyk vyrábí běhové prostředí (runtime)
a pak ještě potřebujeme abstrakce nad hardware - obrazovka atd...

Většina aplikací ovšem také potřebuje nějaké soubory ukládat, to už firmware nedovede - potřebujeme mít nějaký souborový systém.

OPERAČNÍ SYSTÉM

Potřebovala bych udělat mezivrstvu mezi firmware a aplikací (případně i hardwarem) → **operační systém /OS**
skládá se ve skutečnosti z více software)

Nejdůležitější částí je ***kernel (jádro)*** – ve kterém je zase řada funkcí pro aplikace (poskytuje API) takže aplikace nevolají firmware a nekomunikují s hardware ale jenom s OS.

Funkce operačního systému

Systémové volání/sys call
Volání aplikací a OS

- práce se vstupem a výstupem (základně s textovým - klávesnice a obrazovka, ale některé umí třeba i myš, práci se zvukem atd…)
- abstrakce nad diskem , takže v aplikacích tam bude funkce jako readfile a writefile(nikoliv čtení sektorů disku)
- nahrávání aplikací (programloader)
- správa paměti

→ tyto operační systémy s těmito funkcemi se nazývají ***DOS*** *(disk operating system)*
současné operační systémy mají plno funkcí navíc, ale třeba MS DOS má přesně tyto čtyři funkce Atari DOS podobně.

FILE SYSTEM

Víme už že kromě samotných souborů jsou na disku metadata, které něco vypovídají o těch souborech

Naformátování disku – i když je prázdný, jsou tam uložena metadata ve vhodném formátu daného souborového systému.

Svazek - partišna naformátovaná nějakým souborovým systémem

Nuže, máme soubor a chceme aby měl name(jméno) :

- Mohla by tam být jedna souvislá tabulka jmen a hledalo se v ní, ale to je nepraktické
- Mějme jména v hierarchické struktuře (například pomocí lomítek a adresářů, do kterých třídíme) → potom máme jméno adresar1/adresar2/soubor (úplná cesta) (lomítko na win je \)

→ to by nevadilo tomu, aby tam stále byla jen jedna tabulka, a občas se to používá.

Mimochodem můžeme vyrobit obraz disku a také máme formát .zip. To je vlastně jeden soubor formátovaný jako souborový systém (ale tam se čeká že zapíše a rozbalí najednou).
Podobně třeba .docx je ve skutečnosti taky z více dokumentů ale tam to taky není zapsané hierarchicky ale s těmi plnými cestami a jsou za sebou

ALE běžně přistupuji k náhodným datům a je třeba v tom efektivně vyhledávat →
řešení: Opravdu hierarchický systém – jméno souboru je skutečně jen „soubor.txt“ a jinde je soubor „adresar2“ kde je seznam souborů, které obsahuje.
→ Potom jsou na disku uložené ještě ty adresáře a v nich si chci pamatovat nějaká data
Nejjednodušeji je to jen další soubor, kde jsou zapsány potřebné informace.

POZOR! Normálně nechci, aby nějaká aplikace četla a zapisovala adresářový soubor jako normální soubor

řešení: normálně je to v OS zakázané → ale bylo by dobré tedy ten soubor odlišit od ostatních (například booleanovská hodnota v metadatech apod.), dále tam je třeba úvést oprávnění, datum vytvoření atd…

ALE sice vím, ve kterém adresáři jaké soubory jsou, ale stále nevím, kde na tom disku to je. :/ →

==> Takže je třeba ještě mít seznam sektorů. Kdyby to bylo skutečně za sebou, tak stačí first sector + počet sektorů

ALE soubory nejsou stejně velké jako sektory, takže je lepší velikost souboru,

ALE soubory nemusí být souvislé, protože když soubory zvětšuju, tak bych musel moc často předělávat celý disk a přesouvat megabajty dat, aby to stále bylo zarovnané

řešení: proto je lepší podpora fragmentace (odlišuje to logický sektor souboru – což je 1….n-tý sektor souboru, ale ty mohou být uložené na různých fyzických sektorech)
Atari dos nemá více adresářů, ale jenom jeden, ale v zásadě to funguje podobně.

TYPY SOUBOROVÝCH SYSTÉMŮ :

- ext3/ext4 – vychází z původního linuxového systému
- na win běží NTFS – koplikovanější ale zase podobné koncepty
- FAT

JAK PŘESNĚ TO FUNGUJE ?
Základní myšlenka je taková, že logický disk je rozdělený na dvě části – soubory a metadata a stále je na 0 boot sektor a pak někde musí být popis souborového systému (někdy i v části boot sektoru, tzv ***superblock***
důležitá informace je o kořenovém adresáři – buď je pevně dané, kde je kořenový adresář nebo je tam na něj odkaz

ALE 1 sektor ve skutečnosti není úplně vhodná jednotka pro souborový systém (pro malé soubory to bude plýtvání místem) → chceme aby byla co nejmenší
pro velké soubory když budou malé sektory, tak tam bude strašně moc záznamů o sektorech → snaha aby byl co největší

řešení: kompromis → určují si jinou jednotku a říkají jí ***block/cluster*** a a zase je na začátku poznamenáno jak je to velké (např 1.kB), tak je takže se to čísluje až opravdu od té datové části disku
v obsahu adresářového souboru – jméno a odkaz (číslo, kolikátý záznam v tabulce to je) do tabulky v té první části tzv. ***tabulka inodů*** → stejný soubor může mít více jmen

v každém **inode** jsou veškeré informace – délka a další atributy + seznam sektorů - například pole s čísly případně u delších souborů tam jsou odkazy na další záznamy s dalšími údaji
MFT (master file table) - tabulka ve NTFS – velikost tabulky může růst, protože je ona sama jako normální soubor v datové části

JAK VYTVOŘIT NOVÝ SOUBOR ? vybereme volný cluster → takže někde musí být popis toho, kde je volné místo - říká se tomu ***bitmapa (binární mapa).*** Když chceme připsat data, tak najdeme první volný cluster a přidáme jeho adresu do seznamu v tabulce

FAT - jednodušší, nejstandartizovanější souborový systém (víceméně všechny OS s ním umí pracovat), proto ho používají např SD karty. aby to všichni podporovali (zvládne ho i jednočip !)

- adresářové položky obsahují všechno (kompletní záznam o existenci souboru – kde, jméno atd…) tyto položky mají pevně velké 32B (viz dokumentace) – je tam jméno (8 + přípona 3B) , navíc tam je seznam sektorů ve formě jednostranného vázaného seznamu (v záznamu adresáře je odkaz na první fyzický cluster který je zabraný) pak je tam FAT tabulka (file alocation table) a pro každý cluster je tam, který cluster je další

- popis souborového systému je opět v části bootovacího sektoru, tabulky tam jsou pro jistotu dvě (kdyby se v jedné vytvořila chyba)
- kořenový adresář je tam napevno na nějaké adrese

Když soubor zabírá 2,3,4 tak v adresáři je 2 a v tabulce je pak 3 a u trojky je čtyři a pak musí být poznat kde to končí
12,16,32 – varianty FAT podle délky bitů záznamu (konec se značí jako odpovídající počet F)

Chceme-li vytvořit nový soubor, tak připíšeme položku do adresáře, ale potom je potřeba najít první prázdný cluster a když to mažeme, tak zase změnit informace o tom, kde je něco volno

ALE používání funkce **readfile** (a/b/c.txt, offset, kolik bajtů načíst, buffer kam to načtu) - na to se musí lokalizovat kořenový adresář, v tabulce najít jméno a, potom najít kde je a a pak kde je b → tak už jenom hledání souboru je spousta čtení v disku. A když chci číst z konce toho souboru, tak to znamená třeba přečíst milion záznamů → takže je to velice nepraktické

řešení: místo toho je tam tabulka předpřipravených záznamů ***(file descriptor table***, pak ***file descriptor block*** – jeden záznam) se kterými bude chtít aplikace pracovat. Takže když dám Open tak to v tabulce najde a celé to hledání provede jen jednou, takže pak dám open(handle) – kde handle je jednoznačné identifikační číslo v tabulce, tak už to ví.
Bylo by šikovné kdyby se pamatovalo z posledních čtení, že třeba logický cluster 5 znamená fyzický cluster 500 → třeba čtení ve FAT pozpátku je velice dlouhé ale sekvenčně je to rychlé

Když přidá nový cluster souboru, tak se změní FAT tabulka a to se dělá dost často, **ALE** pokud mám rotační disk tak skákat z jednoho konce disku na druhý je strašně pomalé,
řešení: je tam někde na FCB uloženo co se má změnit a když použiju close, tak zapíší data změn, která si někde pamatoval

ve WIN 32 API - otevření souboru je create file
POISX – norma pro unix-like OS, linux API má jinájména funkcí atd...
→ pokud v aplikaci použiju jedno api, tak potom to na jiné operačním systému to fungovat nebude
Ovšem v rámci WIN 32 jsou různá ABI ale API je stejné (??)
!! na linuxu je to jinak, jednotlié systémy mohou mít různé API → tedy není přenosná na binární úrovni

řešení:

- potřebuju překladač pro konkrétní liuxový operační systém

- nebo dám mezi aplikaci vrstvu software tak, aby to bylo nezávislé na ABI → zase si pořídím knihovnu, která nás odstíní od operačního systému , také tam bude třeba implementace haldy atd.. a překládám jen knihovnu pro různé varianty ABI takže pokud používám jen knihovny, tak to přeložím do obj file (nikoliv do executable souboru) a stačí vždycky slinkovat to s jinou knihovnou a pak to bude fungovat všude (občas je nutné přeložit znova celé ty zdrojáky, protože knihovny mají různý typ obj) →

software extraction layer - knihovna volá funkce nejakého další systému, kteý teprve volá kernel (například zavaděč Wine - pro spouštění windozích aplikací na unixu
Cyglin - používá win 32 api a umožnuje spustit unixové aplikace na win
například andorid má Android API , které je jiné než API běžných operačních systému a pod androidem je linuxový kernel a andorid akorát předkládá všecko pro ten linux
axistuje windroid -
Windows NT (od XP) navržené tak, že kernel má nativní API (to není win 32 api) , ale jsou tam růtné subsystémy, třeba win 32 api, a to teprve volá nativní api →
potom existují subsystémy třeba pro MS DOS a jde na tom pouštět historické aplikace a pro 64-bitová Wokna je také subsystém win64
aby i běžné aplikace mohly být pro win 32 na win 64
→ api a abi WAU (windows anytime upgrade) 64 - překládá 32 do 64 a pak to teprve posílá dál

Minule jsme řešili, že RTL knihovna funguje jako nadstavba nad OS aby to bylo přenosné na úrovni binárního kódu, další odstínění aby to bylo přenositelné mezi systémy

ALE je možné že v RTL nejsou nějaké funkce, které OS podporují – například grafika a zvuk

řešení:

- buď ty funkce nevyužívám
- nebo je možné rozdělit kód na dva zdrojáky – grafika a zbytek, v té grafice obejdu RTL knihovnu a budu komunikovat s OS a pro přenositelnost přepíšu pak jen tu část , která mluví se systémem

- taky to by šlo si udělat knihovnu pro práci s grafikou pro každý OS a zase všecky aplikace by zase byly přenosné *(například ZenGL/QT/GTK)*

- ale ani API OS nemusí být dostatečné – například v Atari , tam není práce se zvukem a grafikou, pak se obchází i OS a komunikuje se s hardware → daleko více verzí aplikace

Víme,že různé OS mají různé ABI jak třeba vypadá ?

Linux : je na to vyhrazené jedno softwarové přerušení $80, MS DOS 21, native api windows 2E, win 32
→ desítky fukncí, ale jenom jedno přerušení, jeden hendler pro vstup do kernelu a pak mám jako parametr ID té procedury, kterou volám

proč se na to používá sw přerušení ? Nezávislé na umístění obsluhy

odbočka 1 (všeobecně pro všechny procesory)
když se podívame na běžné procesory , tak známe dva registry – stack pointer a programm couter, ale obvykle se v CPU pamatují nějaké informace o stavu - tzv. příznaky (typicky jednobitová boolean hodnota) a mají na to příznakový register *(u X86 se jmenuje flex a má 16 bitů, u X32 je rozšířený, 6502 – 8 bitový register (procesor status/stavový register))*
Ale jsou standartizované příznaky, které tam určitě jsou. S technických důvodů je tady omezen přístup, ale existuje tam pushff a popf (uložit a zapsat stav registeru)
u 8-bitového php/plp

odbočka 2 (např pro X86, ARM, ale ne pro všechny)
některé mají instrukční sadu rozdělenou
privilegované instrukce – potenciálně nebezpečné a spíše s nimi pracuje kernel
zbytek není privilegovaný
CPU má dva režimy práce – privilegovaný/privilage/režim jádra/kernel mode/supervizor mode/ring 0 - v něm je povoleno úplně všecko (všechny instrukce)
uživatelský režim/user mode, ring 3 - zákaz privilegovaných instrukcí → vede k vyvolání přerušení pokud jí někdo volá *(výjimka 13 - general protection fault*)

Co jsme se dozvěděli z odboček ? Jeden z příznaků v registru je třeba režim supervisor.
Všechny CPU startují v privilegovaném režimu, takže se to chová jako že jiný režim nemá a než kernel začne volat nějakou aplikaci, tak to změní na uživatelský režim.

ALE Někdy ale aplikace chce volat zpět kernel (API funkci) a pak potřebuje, aby se povolil znovu prvilegovaný režim. Zapnutí privilegovanéhorežimu je ovšem privilegovaná instrukce

řešení: Softwarové přerušení - nepřímé volání ale před tím se CPU přenese do supervizorského režimu, takže všechny přerušení obsluhuje kernel,proto to umožní použít supervizorský režim .

Vypadá to, že když zavolá API tak se nastaví supervizorský režim a pak se to zase vrátí do uživatelského režimu, ale my vlastně chceme, aby se to vrátilo do toho režimu před voláním, proto máme jeden registr, kam si to uložíme.

ALE nejprve chceme RET a pak popf. To nejde, už není co popovat.
→ spešl intrukce návrat z přerušení IRET, RTI

STRÁNKOVÁNÍ

Tak fajn, kernel si připraví tabulku přerušení atd ale ta aplikace by mohla třeba přepsat adresy v tabulce a zas by to bylo děravý.
proto je třeba ještě nějaký **koncept ochrany paměti** (kdo kde co může dělat)

JAK ? šlo by to tabulkou ale běžne v relitě je logický adresový prostor rozdělen na ***stránky (page)*** a jejich velikost je mocnina dvojky *(abych poznal z adresy kolikátá stránka to je , vršek je číslo stránky a spodek je offset od začátku stránky (spodních 12 bitů))*
Stránky běžne velké cca 4 kB ale lze to i změnit. Tabulka (page table) popisující toto leží standartně v souborovém prostoru a pro každou stránku tam bude záznam (všechny záznamy stejně velké)
v CPU je register ukazující na bázovou adresu stránkovací tabulky (např na intelu je to register C3) potom kdykoliv dělá CPU nějaké čtení paměti, tak si zkontroluje kolikátá stránka to je, z registru koukne do tabulky a kouká, jestli jet o čtení povolené. T*aková tabulka je ale velice velká, tak je to ve skutečnosti složitější, třeba strom a pod, protože některé velké kusy paměti mají stejná oprávnění.*

První příznak por každou stránku je příznak present (jestli tam ta stránka je nebo není. Třeba když kus toho prostoru tam reálně není), ten se kontroluje vždy, zbytek jen v uživatelském režimu
Další záznmy – jestli je to přístupné jen v kernelovém (1) nebo i v uživatelském (0) → zase ochrrana zničení dat, ktreré potřebujeme – části kernelu, tabulka vektorů přerušení, stránkovací tabulka... , Execution disable je další příznak

Potom se při volání API kontroluje jestli někdo nechce zaisovat do read-only paměti apod.
Zrovna ta stránkovací tabulka může být read-only, aby byla ochráněná.

U většiny procesorů nevzniká obecná výjimka, ale pokud je to jen přístup k nedovolené stránce tak ***výpadek stránky(page fault)*** *např na intelu 14*
PROČ ? Abychom to jednoduše poznali a druhak se chce vědět, kam se ten kód snažil přistupovat, takže procesor si uloží ještě adresu na kterou se snažil někdo lést → je možné že se dá i na zásobník ale třeba na X86 je spešl regitr CR2 a pak ještě se uloží co přesně se dělalo (zapisovat do read-only, provádět kód do executable disable)
Tato chyba se dá opravit, a dá se změnit záznam tabulky a pak na zásobníku je adresa poslední insrukce takže po opravě udělám RET tak je možné že je to opravené a bude to fungovat.

POZOR! ošem těch může být v jedné instrukci více (třeba push 5)(??)
vyvolá první page fault , když to opravíme, tak se to pokusí znova dělat to samé ale najednou je tam třeba druhá nepřístupná stránka a zase se to dá možná povolit, znova to restartuje a pak může vést okaz zase někam jinak, takže jedna instrukce se může řešit fakt docela dlouho

POZOR! win a linux musí běžet jen na procesorech které podporují stránkování a dva režimy

jak se kernel dostane do paměti ?
Zapne se procesor a běží firmware, nahraje se bootmanager nebo bootloader – právě OS si někam dá svůj boot sektor s bootloaderem a pak už běží kernel (v privilegovaném režimu), sám se nastaví, vyplní tabulky nastaví registry atd... pak musí spustit nějakou aplikace, takže slouží jako programloader

Atarai DOS se nahrál a pak donahrál Visicalc takže je tam nahráno obojí, ale nyní apliakce využívá API funkce OS , **ALE** já bych chtěla, aby se tam dalo pusitit více aplikací než jen jednu danou → takže součástí je také **shell ,** který dává možnost vybrat si, jakou další aplikaci chceme spustit.
exe(jmenoprogramu) a kernel load(porogram.exe)
shell může vypadat různě, například jako příkazová řádka, shell je teda normální aplikace a v nějakém cyklu čte klávexninci, zavolá API kernelu, ta to vypíše na obrazovku, když zmáčknu enter, tak to zavolá API kernelu, aby to spustilo tu aplikaci

Jak vypadá spustitelný soubor ? Obraz paměti po překladu ?
Obvykle začíná hlavičkou, která o něm něco říká. Zase obsahuje magic značku, jestli to je fakt spustitelný soubor a ty jsou různé podle OS.
Atari - 1. dva bajty mají hodnotu $FF, protože na něm omylem pouštěli textové soubroy a $FF v textu není.
MS DOS začíná mz (podle autora)
formát PE *(portabel executable)* – že je to přenosné mezi různými verzemi, má to tam hlavičku i MS DOS kvůli zpětné kompatibilitě , ostatní systémy přeskočí mz hlavičku a běží to, v DOS to vypíše něco jako „toto není program pro dos“ a skončí to
v unixu : el (executable and linkable)

Za hlavičkou může být přesně obraz paměti, **ALE** kam to má uložit ? Často tam budou **absolutní hodnoty a volání** a když to uložím jinam, než kde to bylo tak to nebude fungovat.
řešení: Buď se to načte tam kde to bylo je tam uložená bázová adresa) a pak to běží (Atari dos).
Jenže když mám v paměti více programů (třeba jen shell a další aplikaci) ? je možné že se to bude překrývat . V Atari se vědělo kde je shell a prgramátoři by si to museli ošetřit, že to bude začínat někde jinde

Moderních systémech to tak nejde !! →
aplikace si nemohou jen tak někam ukládat data, takže to musí řeši kernel (správa paměti), aplikace si říká o velikost paměti a kernel ví, kde je jaké místo. Knihovna si musí paměť zabrat po kouscích nemůže si všecko zabrat pro haldu!! :/ Typicky po kernelu chce stránku a nebo násobek stránky a pamatuji si to pro celé stránky a runtime pak řeší třeba že se mu něco vrátilo v dispoze atd.. na té přidělení paměti
kernel, když spouští aplikaci, tak se volá sám a alokuje si pro ní místo.

Taky pod kernelem roste zásobník a je třeba paměť pro něj rezervvovat
stále je možné, že zásobník rost až moc →
poznačím si stránku za zásobníkem jako non present (gaurd page) a když se tam najednou dostanu, tak si změním záznam a použiju jí, restart volání bude v pohodě a zase si vytvořím další guard page , takžepak runtime nemůže růst haldou do zásobníku a naopak to taky může volat běhovou chybu.

stránky mezi stack a dalšíma věcma můžou být jako non present, aby ta aplikace nealokovala sama pamět ale opravdu žádala kernel.

Jak to řeší náš problém s adresami ?
Musím provést relokaci : projít všechna místa, kde je adresa kterou volá program sám do sebe a opravit to
přičtu „real base – image base“
a jak poznám co je adresa ? Má jenom množinu bajtů a nevím co je adresa a co intrukce → to mumusí někdo sdělit – ví to programátor nebo překladač. Takže součástí spustitelného souboru je ***relocation table*** což je seznam adres - offset od začátku souboru, kde je adresa *(přesně první bajt adresy)*

a co call ? Musím tam zadát adresu ale kde je tam to místo kde mám v programu začínat ? Šlo by napevno říst první bajt, ale typicky je v hlavičcke ***entry poin***t – adresa začátku toho všeho.
(A ta se taky musí relokovat !!)

!!! Máme-li dva soubory (a.obj b.obj), oba generuje překladač a linker to slinkuje ale i ty odkazy pak vedou jinam, takže už v těch object filech to muís být a už první relokaci provede linker a pak vygeneruje sjednocení ěch dvou relokačních tabulek. **!!!**

když už loader dělá takové věci, takby mohl udělat ještě něco šikovného …. ? (viz příští přendáška)

Známe už dvě vyjímky - pokus o nepovolenou privilegovanou instrukci, výpadek stránky

Běh aplikací – nabootoje se, incializuje se kernel a má funkci exec, která spustí něco jako shell, obraz toho souboru se nahraje do paměti a zavolá se entry point .

Viděli jsme že nahrání toho spustitelného souboru není tak jednoduché, provádí se například relokace
A když už ten loader dělá plno věcí, tak by to mohlo vyřešit ještě něco :
!! v nainstalovaném OS máme stovky i tisíce spustitelných souborů a součástí každého souboru bude přilinkovaná runtime knihovna toho jazyka → máme-li tam tisíce programů v jednom či dvou jazycích, tak je tam mockrát kód jedné té knihovny !!

Chceme : v exe jenom kód dané aplikace a runtime knihovna by byla distribuovaná nezávisle a linkování by dělal za běhu loader ne my, programátoři . Byla by tam tedy jen tolikrát, koli běží apliakcí v paměti a na disku by byla jen jednou. →

DYNAMICKÉ LINKOVÁNÍ

Měli bychom rozlišovat exe soubor a ***dll (dynimic linker library)***
základ je opět hlavička, data aplikace, tabulky
Pak tam budou další dvě tabulky - exports, imports - exporty knihovny se berou jako API a překladač ví, které tam zahrnout.
v tabulce nevyřešených symbolů bude jméno a výskyty, u knihovny bude ještě ve které knihovně ho hledáme.
tedy provede se relokace, pak se projde tabulka importů, které knihovny to chce a ty se taky nahrajou do paměti a provede se linking

Ještě ta dynamická knihovna může záviset na další knihovně takže to loader nahrává, dokud ten graf závislosti není souvislý. I u knihoven je poznačen entry point, to by se mohlo ignorovat, ale využívá se to, že ten entry point třeba spustí nějaký runtime nebo nastaví nějaké konstanty někam atd..
zavolají se všechny entry ponty knihoven a pak entry point programu .
Programy knihoven chceme velice krátké a pak to hned přejde ke kódu. Knihovna i executable soubor mohl vzniknout před tím statickým linkováním .
Výhoda dynamických knihoven je, že pokud je tam chyba, tak se opraví všechny aplikace. Navíc je to v jednom souboru, takže se nikde nic neztratí.
ale zase pokud tam během opravy rozbijeme něco jiného tak to nefunguje všude.

V pascalu takový věc začíná „library“ a ne „program“
chci-li zahrnout proceduru do tabulky exportů, tak tam napíšu „export“.
V souboru naopak napíšu název a za to „external jmenoknihovny name jmenoporgramu“
pak to obojí přeložím a vypadne mi to. We win se dá stáhnout program na prohlížení tabulek.

Využití ? : například Windowsy mají native api s přerušením $2E a je tam win32 subsystém, který poskytuje win 32 API. Cože je jenom sada linkovaných knihoven. A to co tam vidíme, tak to jsou ty dynamicky linkované knihovny. Chtěli li bychom pro win v assembleru, tak bychom využívali api funkce win 32.
db (definuj byte).
Někdy se k souboru přidá statická knihovna s názvy využívaných funkcí a v jejich těle je jediná instrukce – skok na tu funkci ve správné knihovně. a tam mám poznamenané importy do x.dll (v te co mám je jen @x.dll)

assemblerem přeložim na obj, a pak pustím linker a ten to slinkuje s knihovnami a vypadne mi exe soubor kde to bude všecko fungovat
jde tam také zakázat, aby se tam linkovalo něco dalšího zbytečného.

Teoreticky by se ty knihovny nemusely spouštět s exe souborem, ale až za běhu bychom je mohli rozhodnout , že chceme nějakou další knihovnu. Příkaz Looplibrary - do paměti nahraje knihovnu za běhu a ta provede to linkování tak, že řeknu co teda hledám a dostanu opravdu pointer na funkci

lazy/demand loading
typicky se někdy při celém běhu nepoužije celý kód ale chcmee aby se nahrál jen ten kod který potřebujeme aby bylo loadovánírychejší, takže si na to jen vyhradíme paměť ale celý ho nenahrajeme.

A až když se to začne volat, tak se koukneme co tam mělo být, nahrajeme to, restartujeme intrukci a bude to fungovat

chceme vědět že progam končí aby chom se vrátli do exec a pak se vrátil do kernelu.
Každá apliakce by někdy měla zavolat unkci exit a zabít se.

Aby to fungovalo tak posuneme stackpointer o velikost funkce exit. protože na vršku je odkaz na program, my ho posuneme na exec, takže když se dá exit tak ret načte exec a to už funguje.
taky ale potřebujeme uvolnit zdroje – paměť (stránky)
každ aplikaci budeme říkat proces a pak tam budeme mít v kernelu pro každý proces záznam v tabulke a každý proces má jednoznačné proces ID a někde v systému si budeme pamatovat v kontex switchi číslo (curr PID) právě běžícíhi procesu.

a pak si ke každé stránce poznamenáme PID procesu a pak se projdou všechny stránky a ty které mají číslo danoho procesu se určí jako prázdné.

VLASTNÍ OŠETŘENÍ VÝJIMEK ?
Některé runtimy si chtějí implementovat vlastní ošetření vyjímek , ale když spustíme dvě aplikace, tak každá aplikace má svojí obsluhu a takže to uděláme tak, že všechny procesy budou volat kernelí vyjímky a ten si pak v rámci běhu toho procesu (aktuálního) zjistí, čí vyjímky má zavolat , případně vyhodí svojí (neboli – má přece tabulku pro každý proces, takže se do ní koukne, jestli tahle výjimka není ošetřená a poud ano, tak to provede).

zase je i tabulka pro aktuální rpoces s otevřenými soubory - takže když se to zabije, tak se všecky najednou mohou zavřít.
Součástí procesu je i aktuální adresář, takže je tam podpora relativních cest. A ví se také, které všechny knihovny se mohou vymazat z paměti.

ALE !! Posun kvůili exitu nebude fungovat, máme-li tam nějaké argumenty. Navíc je možná, že exit voláme z nějakého subrutine, takže se zase vůbec nevrátíme zpět. Stačilo by tedy pamatovat si adresu před entry pointem někam dál(to si uložíme než se ějak rekurzivně zanořímeú a tu obnovíme a připočteme že tam bude entery point, aby se to vrátilo opravduz exit do exec.

!! představem si že máme x stránek alokovaných pro haldu a pak si zavoláme další aplikaci, ta si zase alokuje haldu. Máme ochráněný kernel, ale pořád si aplikace navzájem mohou škodit.
Řešení: Vytvoříme si stránkovací tabulky pro každý proces a označíme si přístupé v té tabulce jen stránky toho procesu, kterému tabulka patří. Takže z každého procesu to pujde na jinou tabulku

máme strandartně neplatný uakzatel na příklad nula (reprezentuje nil) takže tu stránku udělá nepřístupnou a když se budeme snažit dereferencovat nil, tak to vyhodí chybu.

DEBUGGING ?
Někde v paměti je kód aplikace a někde je kód debugeru → když uděláme krok <=> uděláme breakpoint na jednu instrukci a pak na jinou. Ten debuger si přepíše a zkopíruje ten kód a já můžu dělat všecko možné kolem a když jdu na další instrukci, tak to najde v původním programu další intrukci, ta se provede,zapíšou se proměnné a pak to vrátí zase do toho debuggeru
má na to přerušení CD03 a obkod CC (zkratka za to první) - často to práznou pamět vyplní na CC brt 00

chceme aby se debuger zastavil na první instrukci příkazu. Každý soubor má tabulku, kde pro každý řádek je napsáno, kde je první instrukce. A taky je tam seznam kde je jaká proměnná.

ZÁKLADNÍ OPERACE
a+b+c se dělá po kouskách, většinou to musí být jen operatory binární. Někde ani nejde zvolit si, kam to chci uložit, ale musím to uložit do jedné z těch dvou operandů.

load – načtení z paměti do registeru
store - načte z registru do paměti
!! protože někde to umí provídět jenom z registrů nikoliv z paměti (tzv loadstore architekruta)

procesory proto potřebují obecné registery, které PCU sám od sebe nijak neinterpretuje.
X86 – obecná registrová architektura
Někdy je jeden register který není ekvivalentí ostatním a plno funkcí pracuje natvrdo s tím jedním říká se mu ***akumulátor*** → akumulátorová architektur → zjednodušuje to procesor ale komplikuje programování

základní intrukce je load – načtení do registru (typicky tedy akumulator)
procesor je osmibitový → osmibitové hodnoty
zase je tam víc typů – 16bitová adresa do paměti kde je to číslo nebo rovnou to číslo, ale často mají funkce různé adresové režimy → indexová adresa , například řekneme konstantu a ono to přičte k adrese a tam to najde a nebo máme registry a můžeme dělat jen: jeden z těch registrů + konstanta jako adresa

můžeme říct adresu - konstantu, kde je pointer, kde je 16 bitové číslo a to se vezme jako číslo na adresu (protože kam bychom uložili 16 bitů adresy ?!)

store
nkědy cheme taky přeášet mzi register ynavzájem - TAX (z registru A do X atd…) a nejsou tam úplně všechny kombinace → někdy je potřeba více insstrukcí na jeden požadovaný přenos.
TAX TXA TAY TYA TSX TXS

!!push x a push konstanty nelze ale lze jen push ac umulator.

příznak zero - když je tam 1 tak výsledekm předchozí operace byl nula
příznak znaménko - byl li předchozí operace záporná
kromě operace tedy některé instrukce nastavují nějaké příznaky

taky jsou spešl instrukce pro práci s příznaky , ale nejde to se všemi, někde třeba jen carry

bitové operace to typicky umí vždy - a tady třeba zrovna to umí jen s akumulátorem a taky je tam třeba jen shl 1 shr 1 a není tam negace pač se to dá udělat orováním
a všechny tyto operace zase zapisují do registrů.

!! Ale všechny zase pracují s osmibitovou přesností a tak když je to bit po bitu, tak to mít (16 bitové číso) na dva kousky

složitější je ADC – advice carry - přičítá to to carry příznak a navíc to ještě poslední číslici z výsledku to dá do carry. A ten muísme teda vynulovat. Kdybychom chtěli sčítat 16 bitová čísla, tak to přesně udělá carry (→ je to v 17 bitové přesnoti a řeší to tam právě přetečení)

násobení a dělení pk už dokážeme naprogramovat. Odečítání je vlastně sčítání opačné hodoty (ve dvojkovém doplnku zvětšíme o 1 a znegujeme)

6502 - akumulátorová architektura , nyní už umíme zapsat uplně cokoliv (teoreticky :D :D)
rozepíšeme si to, aby to byly jen binární operac a nebo připravené pro volání funkce a potřebujeme spoustu proměnných na mezivýsledky. Chcem to také zjednodušit, aby to byla 1 operace nebo 1 funkce.
A ještě tam přepíšeme aby se to nahrálo kam má, když výsledek se vrací do jednoho z operandů.
Zprava doleva ukládáme argumenty , běžně se vrací návratová hodnota v akumulátoru a omezíme se na 8bitové proměnné !!ještě neumíme násbit → to musí volat proceduru v runtime toho jazyka.
A protože zase umíme pushovat jen akumulátor, tak to musíme podle toho upravit a přepočítání jakbysmet.

!!problém je když voláme mezivýpočty a všecko je v akumulátorech, tak to musíme nějak přesouvat a dočasně ukládat.
taky potřebujeme alokovat místo na zbývající proměnné - typicky na zásobníku a zase to musíme udělat přes akumulátor.
!! a taky musíme odstranit argumenty z volacího zásobníku
RET se vrací na zásobník ne na akmulátor , ukazujeme na první nepoužité místo a pak někam musíme dát návratovou hodnotu
(jak se přistupuje k lokálním proměnným ?)

X86 mají obecnou registrovou architekturu, máme 32 bitovou architekturu a chceme třeba pracovat s 8 bitovými hodnotami → koukáme se na jeden registr jako na víc menších registrů.

MOV cíl adresa (!! nejde to z paměti do paměti, vždy je jedno registr a jedno paměť)
podporuje zase všechny adresovací režimy ale není tam nikde napevno registr třeba na indexování
zase jsou ta všechny bitové operace, práce s příznaky atd.. taky je tam modifikace pro sčítání bez přičítání carry

→ nyní přepis výrazu do assembleru je mnohem jednodušší, taky má obecné násobení a dělení ;)
a máme obecný pop, takže stačí že to pushneme na zásobník ale pak to muísme ve zprávném pořadí posunout zpět.

!!Někdy ale očekáváme že během volání funkce je nemění obsah registrů → součástí calling convention musí být řečeno, které registry se zachovávají (typicky je to třeba jen jeden který se zachovává)

CHCEME BÝT TURING COMPLETE
Koncept podmíněných intrukcí – když není splněná podmínka, tak to udělá další intrukci, jinak se to provede
ale většina má podmíněnou instrukci skoku - když něco neplatí, tak skočíme jinam apod.
A pak ale musíme ještě udělat skok přes tu else větve pokud se if větev provádí a obvykle se rozhodují jenom na základě příznaků z registru.
pomocí toho jde napsat i and a or
?? co porovnání na rovnost a nerovnost ? Stačí odečíst ty dvě hodnoty a když se rozdíl rovná nule,t ak to platí a jinak to neplatí (a na nulu máme register)
občas se hodí někam uložit i ten mezivýsledek → non-destructive substraction – změní přáznak ale neničí register

nerovnost A-B a carry = 1 tak A < B apod… často ma assemblery i instrukce přímo proto když se něco rovná atd… = <=,>= tak to napíšeme pomocí OR

cykly ?
Na podmíněné skoky se udělá while – když to platí tak se to provádí a zpět na podmínku, když neplatí tak to skočí dál
for cyklus je z while cyklu a iterujeme

ještě jedn architektura: zásobníková architectura (stack machine) zas nejsou tam registry, stack pointer atd… ale s pamětí pracuje jen jako load a druhý argument je implicitní a ty registry jsou v zásobníku a my si nemůžeme vybrat, kam se to uloží

virtual machiny používají právě tohle, protože je tam pak pěkně vidět co se tam děje, zase tady není variabilita → nedá se optimalizovat využití zásobníků → je ot pomalé.

Inline assembler, lze psát kus v assembleru a kus v jazyku a pak zase naopak
v Atari je Kyan pascal (překladač)
#A kod mezi tim je v assembleru#
!!nejde tam přípona pas musí tam být přípona P.
!! je to velice pomalé (program s třemi funkcemi by se překládal třeba 2 minuty)

i FPC podporuje inline assembler (asm assembler end) a v inline assembleru lze používat fukce a proměnné ze zbytku programu.
$ASMMODE INTEL – aby to používao tu sysntaxi, kterou známe. Během debuggování lze přepnout na disassembler .

POZOR Když tam napíšu něco v assembleru, tak už to pak není přenosné
příklad napsat virtuální kernel a použít tam třeba funkci exec a podobně, ukládat si vršky zásobníku a podobně, tak to v Pascalu neudělám, ale pak je lepší použít ten assembler co nejméně a dát do nějaké extra knihovmy.

ALE víme že na DRAM mají pamět tak 1GB ale jsou velice pomalé, a my to potřebujeme pro všechny operace načíst do registrů procesoru, a procesor je velice rychlý, takže se to všecko velice zdržuje.

Řešení: V krátkém časovém okamžiku potřebují procesory data tak v řádu megabajtů.
vytvoříme cash paměť, která je taky celkem rychlá a ta to tam zrychluje. Je organizovaná do řádků a načte vždy celý řádek (protože ho určitě budu někdy potřebovat)
má-li program potžebu více paměti než cash, tak je to najednou strašně moc moc pomalé
pokud chceme nap(například porovnávání třídících algoritmů pak může dopadat překvapivě -u programování je tedy třeba přemýšlet **!!)**

JAK SE KOMUNIKUJE SE ZAŘÍZENÍMI ?
K CPU je připojená pamět RAM a ROM tak je tam nějaká netriviální část adresovaného prostoru nevyužitá
Co vlastě chceme ? akorát číst a zapisovat do malých zařízení (registrů toho zřízení). Když jich bude málo, tak stačí abychom si nějaké volné adresy namapovali na ty registry - zařizuje řadič paměti.
Ve skutečnosti totiž na jednu parlelní sběrnici připojíme nejen paměti, ale i ostatní zařízení → ***systém bus***
a všechna připojená zařízení se nazývají ***controller/řadič***

Povedou tam signály do bus interface + datové vodiče ale musí tam vést adresové vodiče, pač na 4 registry stačí 2 bity, ale kdyby tam vedly jen 2 vodiče tak by to vedlo k aliasingu → vede tam CE signál z řadiče paměti a vybírá, které zařízenímchceme. Nebo tam vede adress decode a tam vedou zbývající adresové vodiče.

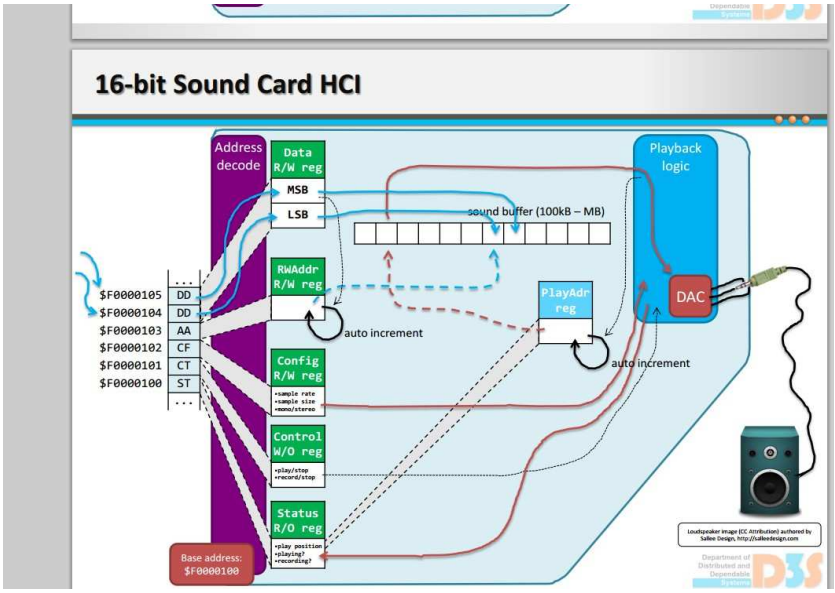
Víme : V počítačích běžně vede jedna sběrnice (systémová) a na ní je řadič paměti, paměti a jednotlivá zařízení a adresy mapované v jednom adresovém prostoru

řadič klávecnice (KBC - keyboard kontroller) : když kmunikujeme s nějaký zařízením, je třea věět kde jsou ajké registry ---> HCI (host controller interface)
ideálně registr kde e poslední stisknutá klávesa a když to přečtu tak se to zárovň vynuluje

ZVUKOVÁ KARTA

zvuková karta – 8 bitová, vevnitř buffer vzorků(samplů) a pak logika na převod do analogového signálu.
Moderní zvukové karty jsou 616 bitové a 16 bitů velký mají i datový registr. → namapována na va 8 bitové registry

!!! je důležité, aby se zvuk reprodukova (četlo se z bufferu) ve správné chvíli, tedy ne když je tam půlka star a půlka nové adresy → přidáme nějaký kontrolní signál.
Realné zvukové kartmají buffer v řádech stovek MB takže zabírá v adresovém prostoru více.
Ve skutečnosti by mělo být řečeno jestli je to mono/stereo a taky aby to přehrávalo v jiné rychlosti.
!!! Také chceme ticho – například samé nuly, ale to by byl nějaký šum → kontrolní registr na odpojení
To by se pak dal připojit mikrofon a naávat něco do registru a vyčíst a uložit →
stavový registr – přehrává se, nahrává, neděje se nic
taky bychom chtěli vidět, jak je na tom logika a jak rychle máme dodávat data
a to už máme třeba 6 B v adresovém prostoru .



GRAFICKÁ KARTA

dříve ***VGA rozhraní*** – monitory fungovaly jako televize (elektronové dělo střílící elektrony a jezdící po řádku) a ta karta má tři signály (rgb) a tam se posílá přímo intenzita toho paprsku + ještě zalomení řádku
→ grafická karta musí mít někde informaci, jak to má vypadat, tedy bitmapa (jednoho frame) – celé obrazovky
mohl by tam být buffer na data, ale občas stačí změnit jenom některou část bitmapy a zbytek nechat.
→ celá grafická pamět je namapovaná do adresového prostoru a zapisujeme přiímo jednotlivé pixely
ALE! moderní grafické karty mají paměť obrovskou, takže by to zabralo strašně moc → okno /window na nějakou část adresového rpostoru (třeba 100MB) a potom registr který mění pozici okna ← *bank switching*
VRAM (normální DRAM pro ukládání dat grafické karty)

KOMUNIKACE PO SBĚRNICI

řadič paměti má v sobě registr (tedy je také namapovaný v prostoru a CPU s ním komunikuje pomocí registrů).
vždycky to bylo tak že CPU je master a pamět je slave
ale hodilo by se aby zařízení byla někdy master → **DMA (direct memory acces**))třeba grafická karta by někdy chtěla číst z paměti)
je třeba dohodnout ***arbitraci sběrnice*** – kdy je kdo master a slave (aby nedošlo ke kolizím)
JAK ?
Máme li dva mastery a oba chtěji komunikovat se slavem
pull up rezistory (odpojení je 1) → víme že 0 přebije jedničku
když oba masteři generují stejnou hodnotu, tak je to v pohodě dokud se shodnou, ale jakmile se začnou lišit tak najednou vidí ten co zapisuje 1 že někdo jiný zapisuje nulu, takže se odpojí, transakce je neprovedená a musí to zkusit znova.

ALE!

pro složitější sběrnice se to nehodí, prototže pořád může být jeden master předbíhán
proč více masterů ? Třeba grafická karta – nemá detekovanou video paměť, ale využívá část operační paměti RAM , ale v 32bitovém prostoru by to mohl být problém

řešení:

- signál halt – když je 1, tak se zastaví procesor,nic nedělá a odpojí se od sběrnice

ale co když jsou tam dva masteři ? A navíc strašně dlouho trvá čtení těch dat a procesor nic nedělá, → ale pokud je tam pamět cash, tak může CPU chvíli běžet z catche

- signál Hold – vzdá se sběrnice, holdA – potvrdí že se odpojil a pak teprve druhý master začne něco dělat

ale mohla by tam být RS232 a tam mít připojenou myš → ***HBA*** – hosl bus adapter - popojení dvou sběrnic
byl by tam jeden registr, kam dáme bajt a on ho pak pošle po bitech a pak registr na nastavení frekvence atd...

pak bychom tam ale ještě mohli mít I2C sběrnici, pro kterou nemám řadič → ***Bridge*** – propojuje dvě sběrnice, z nichž ani jedna není systémová .
Vede tam vstupní a výstupní linka, a říkáme co má poslat po I2C sběrnici na konkrétní adresu :

- pošle začátek,
- R/W,
- kolik n bajtů budem posílat,
- pošleme těch n bajtů
- ukončovací znak
- bridge sám sleduje idle stav,posílá ACK atd..... pak se to zabalí do dat RS232 sběrnice a až ten řadič si to nějak přebere. Tedy ten bridge je tady dost inteligentní :)

:(taky bychom mohi mít řadič připojený přímo na systémovou sběrnici a na I2C , který by byl jednodušší

- controlní registr – informace o stavu/změna konfigurace
- stavový regstr
- datový registr

JAK ?

- zapíšeme do controlního registru někam 1 → aby se vysílala start condition
- čekáme, až se mu podaří procpat se na sběrnici a až se jinde tam objeví 1 tak vidíme, že se procpal na sběrnici (čteme to v cyklu)
- pak do datového zapsat, ať neposílá start condition ale co a jak
- a pak zase čteme tak dlouho, jestli se to odeslalo
- pak číst ze stavového registru jestli neztratil arbitraci, jestli to došlo atd...

ALE!

- kdyby šlo všecko dobře, tak uděláme 11 čtení a zápisů na sběrnici
- pro rychlé procesory bude tato sběrnice strašně moc moc pomalá, pač čekání na další zápis a tak je dlouhé

polling - opakované dotazovování na stav než sezmění

řešení: z řadiče vede ***interrupt request****(hardwarové, asynchronní přerušení)* a když se to změní,tak se něco stalo a vede to do procesoru (ale to by to musel kontrolovat, proto tam vede ta žádost o přerušení) →
když přijde žádost o řerušení, tak mezi těmi instrukcemi, co provádí vloží to přerušení a vloží se tam provádění celého handleru z přerušení (jen na hranice mezi instrukcemi)
je dvou typů

- edge triggeret – všimne si hrany a až má čas, tak to tam vloží
- level interrupt – kdykoliv je mezi instrukcemi a je signál nastaven na 1, tak přerušení vznikne

ALE! s asynchornním přerušením program nepočítá → když v náhodném místě přerušíme kod, tak se mohou změnit registry a pak by nešlo pokračovat → se hned někam(na zásobník typicky) uloží obsah všech registrů

:) Řadič rozseká algoritmus na části, které se provádějí mezi tím, co má procesor čas.

ALE!

- je tam více zařízení, která chtějí generovat přerušení ale je tam jen jeden signál !

obsahy registrů pak nazčátku musíme se zeptat zařízení, kdo to vyvolal

řešení: ***Interrupt controller*** : máme -li sdílení přerušení, tak chceme řadič, který kontroluje ta přerušení a on si bude řídit žádosti od zařízení. Pak pošle CPU to přerušení - ten to udělá, pošle zpět řadiči ACK a čeká, že řadič tomu přiřadí nějaké číslo přerušení → taky užvíme, které zařízení toto přerušení vyvolalo
int – vyvolání přerušení

ALE! Může se nám stát, že řadič bude rychlejší, takže vznikne ***race condition (časově závislá chyba)***

řadič i zařízení drží 1, ale někdy chceme říst, aby negeneroval přerušení a nebo aby řadič přestal generovat signál.

ALE! Jelikož kykoliv je hrana, tak se volá přerušení, takže když ho vyvolám jednou, tak se mezi obsluhou přerušneí může vyvolat další → tam nebezpečí rekurzivních volání přerušení, přetečení zásobníku

interrupt able/enable -příznak (CLI, STL) – povolení přerušení (hardwarové) – takže to na začítku provádění obsluhy zakážeme

ALE! ale když to bude vznikat tak rychle za sebou, tak to stejně nestihneme změnit :o → ***Atomic*** - ***atomické akce*** – nepřerušitelné akce.

NMI (nemaskované přerušení) – jen na hranicích dvou instrukcí a ignoruje se maskování přerušení (například když chceme zmačknout reset, nebo je li špatně něco v registrech, řadiči atd...)

KONKRÉTNÍ SBĚRNICE

sběrnice ISA
IBM PC
(4,77 MH, 8MH) - 8 bit data + 20 bit adresy nebo 16 bit data + 24 adresy
fekvence CPU je typcky rychlejší než sběrnice → vložíme tam bridge pevně se 3 takty nic neděje aby měla čas najít data v zařízení
má ještě jeden adresový prostor **(IO prostor)** - aby tam nevznikala shadow RAM
a zařízení se rozhodují kam se namapují
→ a pak se musíme vždy rozhodnout, kterého adresového protoru se instrukce týká (IOR/IOW)

Navíc má dva signály pro optimalizaci
NOWS – když už je připravena, tak se nemusí čekat
CHRDY – pokud naopak potřebuj více času

ČTENÍ Z PEVNÉHO DISKU
řadič pevných disků a zase bude buffer a pak si přečteme ta data a registry na adresu
nyní už máme řadiče integrované v pevných discích (IDE/ATA)
z disku chceme velká množství dat, ale typicky vezmeme dvě data, provedem instrukci a vrátíme a to jsou nejméně 4 instrukce (načtení OP kódu, načtení registru, OP kód druhé instrukce, zápis zpět), v reálu vlastně 8 nstrukcí

PIO - ***programmed IO*** - přenášení dat programem → ale to e velice neefektivní

DMA přenos - donutit zařízení, aby samo zapsalo a nešlo to přes CPU → zavedeme tam DMA řadit a ten bude korigovat to všecko (ve vnitř bude registr s cílovou adresou – fyzická !!! - někdy tedy musíme ručně přepočítat, co se děje v programu)
CPU se odpojí (hold)
DMA řadič vystaví adresu a dá signál že zapisuje
zařízení vystaví data a pak se přenáší data dokud je signál v 1

sběrnice PCI
pararelní, multiplex, 33 MHz , (taky jsou 66MHz a 64 bit)
libovolné zařízení může fungovat jako master
DMA musí mít zařízení v sobě ***(DMA bus mastering)***
vždycky je tam nějaký bridge sloužící jako **arbiter**