

Architektury počítačů a paralelních systémů

Monolity

- 1) Periférie monolitických počítačů - vybrat si a popsat.
- 2) Vysvětlete PWM a kde se používá. Obrázek dobrovolný.
- 3) A/D a D/A převodníky a k čemu se používá. Nákres dobrovolný.
- 4) I2C - co, jak, kde, naskreslit.
- 5) Popiš základní konstrukci a vlastnosti mikroprocesoru.
- 6) Popiš mikropočítač, se kterým ses seznámil. Nákres.

Disky

- 7) Fyzikální popis HDD čtení, zápis a nákres. Vysvětlit podélný a kolmý zápis.
- 8) Popište a nakreslete stavbu disku. Nechtěl zápis.
- 9) Čtení CD - princip a obrázek.

Zobrazovací jednotky

- 10) Popište a nakreslete technologii LCD - výhody, nevýhody, rozdíl mezi pasivním a aktivním.
- 11) Popište a nakreslete technologii OLED - výhody, nevýhody.
- 12) Popsat E-ink - jaké má barevné rozmezí, výhody a nevýhody.
- 13) Vybrat která zobrazovací jednotka je podle tebe technicky nejzajímavější a proč (OLED, LCD, E-ink).

RISC

- 14) Popište na RISC procesoru zřetězené zpracování instrukcí, jaké má chyby a jak se řeší.
- 15) Popište na RISC procesoru zřetězené zpracování instrukcí a jak nám pomůže predikce skoku.
- 16) Jaké problémy a hazardy mohou nastat u RISC.
- 17) Popiš základní konstrukci a vlastnosti mikroprocesoru RISC.
- 18) Popiš a nakresli schéma RISC procesoru, se kterým ses seznámil.

CISC

- 19) Popište a nakreslete jakéhokoli nástupce Intel Pentium Pro, se kterým jsme se seznámili.

Paměti

- 20) Rozdělení polovodičových pamětí a jejich popis (klíčová slova a zkratky nestačí).
- 21) Jak funguje DRAM, nakresli. Napiš stručně historii.
- 22) Hierarchie paměti, popsat a zakreslit.

Architektura počítačů

- 23) Popiš základní konstrukci a vlastnosti počítače.
- 24) Jak funguje počítač a jak se vykonávají skokové instrukce.
- 25) Popište a nakreslete harvardskou architekturu, popište rozdíly, výhody a nevýhody oproti von Neumann. Na obrázku vyznačte části, které mají a nemají společné. Která architektura je podle vás lepší a proč?
- 26) Popište a nakreslete architekturu dle von Neumann. Napište jeho vlastnosti, výhody a nevýhody.

Komunikace

- 27) Komunikace se semafory a bez semaforů (indikátoru). Nakresli aspoň jedním směrem.
- 28) Přenos dat použitím V/V brány s bufferem. Nakreslit obrázek komunikace jedním směrem a jak se liší komunikace druhým směrem. V jakých periferiích se používá.
- 29) Popiš DMA blok a nakresli schéma DMA řadič v architektuře dle von Neumanna.

Assembly x86

- 30) Jak adresujeme na úrovni strojového kódu - příklad.
- 31) Podmíněné a nepodmíněné skoky v strojovém kódu.
- 32) Jak řešíme v Assembly x86 podmínky - co jím musí předcházet. Jaký je vztah mezi tím, co je předchází, a tou podminkou. Kde a proč záleží na datových typech.

CUDA

- 33) Princip programování CUDA - jak, kde, kdy se přesouvají data při výpočtu.
- 34) Jaké je C/C++ rozšíření CUDA a jak to využije programátor. Jak si programátor organizuje výpočet. K čemu je mřížka. Nákres dobrovolný.
- 35) Čemu by se měl programátor vyhnout a jak CUDA funguje.

Paralelní systémy

- 36) Vysvětlit Amdahlův zákon a jak bychom se podle něj rozhodovali.
- 37) Charakterizujte komunikační modely paralelních systémů.
- 38) Charakterizujte Flynnovu taxonomii paralelních systémů.

Monolity

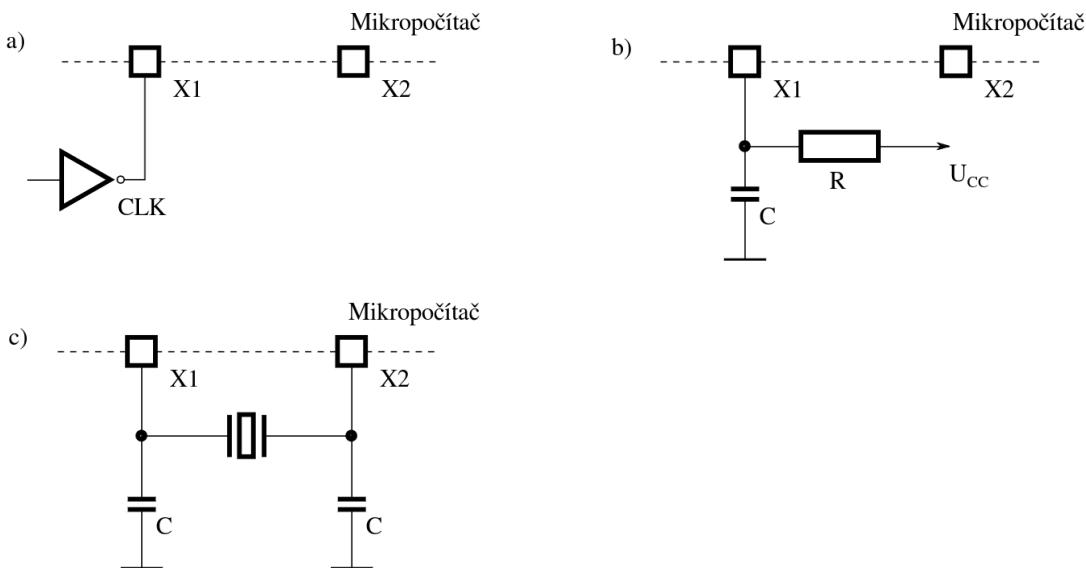
Otázky:

- 1) Popiš základní konstrukci a vlastnosti mikroprocesoru.
- 2) Periférie monolitických počítačů - vybrat si a popsat.
- 3) Vysvětlete PWM a kde se používá. Obrázek dobrovolný.
- 4) A/D a D/A převodníky a k čemu se používá. Nákres dobrovolný.
- 5) I2C - co, jak, kde, naskreslit.
- 6) Popiš mikropočítač, se kterým ses seznámil. Nákres.

1. Popiš základní konstrukci a vlastnosti mikroprocesoru (monolitu).

- mikroprocesory mohou být vyráběny pro řešení velmi specifických úloh, proto nelze jejich konstrukce a vlastnosti zcela zgeneralizovat - můžeme očekávat velké rozdíly mezi jednotlivými mikroprocesory
- převážně se používá harvardská koncepce:
 - oddělená paměť pro program a data
 - možnost použít jiné technologie (ROM, RWM) a nejmenší adresovatelné jednotky (12, 16, 32)
- procesory jsou obvykle RISC:
 - kvůli jednoduchosti, menší spotřebě energie a menší velikosti
- typy paměti mikroprocesorů / monolitických počítačů:
 - pro data se používá *RWM-SRAM* (*Read-Write Static Random-Access Memory*)
 - statické - jejich elementární paměťové buňky jsou realizovány klopnými obvody
 - pro program se používají *ROM* paměti:
 - nejčastěji *EPROM*, *EEPROM* a *Flash* paměti + také *PROM* (*OTP - One-Time Programmable*)
 - některé mikroprocesory jsou označeny jako „*ROM-less*“
 - nemají osazenou paměť pro program přímo na čipu (*On-Chip*)
 - paměť pro program se připojuje k monolitu jako externí paměť
 - např. Flash stick zapojený do *QSPI* portu na *RP2040*
- paměť je organizována na:
 - pracovní registry - obvykle jeden, dva
 - ukládají aktuálně vypracovaná data
 - jsou nejčastějšími operandy strojových instrukcí
 - „*scratch-pad*“ registry:
 - pro ukládání nejčastěji používaných dat
 - část strojových instrukcí pracuje přímo s těmito registry
 - paměť dat *RWM*:
 - pro ukládání rozsáhlějších a méně používaných dat
 - instrukční sada nedovoluje krom přesunových instrukcí s touto pamětí pracovat přímo
 - musí se neprve přesunout do pracovních registrů
- počítač obsahuje také speciální registry:
 - instrukční ukazatel (*Instruction Pointer*) - ukazuje na instrukci v paměti, která se bude vykonávat
 - instrukční registr - ukládá vykonávanou instrukci
- zásobník s návratovými adresami:
 - buď je v paměti na vyhrazeném místě nebo jako samostatná paměť typu *LIFO*
 - aby se vědělo kde je vrchol zásobníku je třeba mít *ukazatel na vrchol zásobníku* (jako registr)
- zdroje synchronizace mohou být interní a externí:
 - integrován přímo na čipu - není dobrá stabilita (i rozdílná tepota způsobí značné odchylky)

- hodí se tam, kde není potřebna vazba na reálný čas
- externí generátory - často se používají:
 - krystal (křemenný výbrus) - dobrá stabilita, dražší (postavna na piezoelektrickém jevu - krystal při deformaci generuje el. napětí)
 - keramický rezonátor - dobrá stabilita, dražší
 - RC oscilátory - může být nepřesný, levný



Obrázek 1: Externí zdroje synchronizace - a) externí zdroj, b) oscilátor s RC článkem, c) krystal

- počáteční stav *RESET*:
 - monolit je sekvenční obvod závislý nejen na instrukcích ale i na stavech a signálech
 - aby počítač spolehlivě spustil program, musí být definován přesný počáteční stav (stav *RESET*)
 - proto jsou implementovány inicializační obvody, které počítač do tohoto stavu dostanou
- ochrana proti rušení / nestabilitě / zničení obvodů:
 - mechanické vlivy - náhodné rázy, vibrace - musí být galvanicky oddělen od okolí
 - program může vlivem okolí „zabloudit“ - tento problém řeší obvod *WATCHDOG*
 - je to časovač, který je neustále inkrementován nebo dekrementován při běhu počítače
 - přetečení nebo podtečení tohoto časovače způsobí *RESET*
 - procesor tedy musí průběžně tento časovač vynulovávat
 - pokud je ale „zablouděný“, tak tuto činnost nedělá → přetečení → *RESET*
 - hlídání rozsahu napětí, ve kterém počítač pracuje:
 - např. počítač funguje jen ve stanoveném rozmezí 3-6V
 - dojde-li k tomu, že napětí napájení stoupne nad nebo klesne pod toto rozmezí → *RESET*
- má integrovaný přerušovací podsystém: (*Interrupt Subsystem*)
 - povoluje a zakazuje *interrupts* - požadavky od periferií pro procesor, aby něco bylo vykonáno
 - definuje způsob obsluhy *interruptů*
 - zjišťuje zdroj a prioritu *interruptů*
- periférie: (viz další otázka more dyk)
 - vstupně-výstupní brány (*I/O gates*)
 - sériové rozhraní (*SPI - Serial Peripheral Interface*)
 - čítače a časovače (*Counter & Timer*)
 - čítač vnějších událostí = inkrementuje se vnějším signálem
 - časovač = registr, který je inkrementován hodinovým signálem
 - A/D (*Analog to Digital*) a D/A (*Digital to Analogue*) převodníky (*ADC & DAC*)

2. Periférie monolitických počítačů - vybrat si a popsat.

Vstupní a výstupní brány (I/O gates)

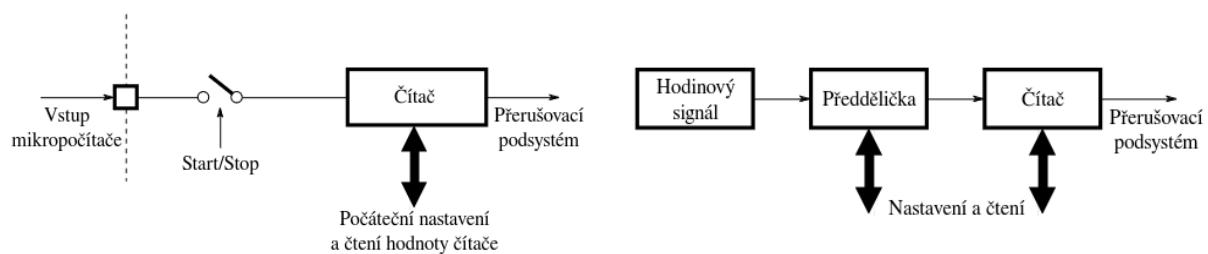
- nejčastějším a nejjednodušším rozhraním je paralelní brána neboli *port*
 - skupina jednobitových vývodů - mohou nabývat log. 0 a log. 1
 - většinou je 4-bit nebo 8-bit - předají se naráz (ne sériově)
 - lze nastavit jednotlivé vývody jako vstupní a výstupní piny (vodiče)
 - instrukční soubor s nimi pracuje buď jako s jednotlivými bity nebo celky
- tyto univerzální vývody umožňují komunikaci po sériové lince s vnějšími zařízeními
 - např. lze jimi dynamicky ovládat LCD a LED

Sériové rozhraní

- pro přenášení dat mezi periferními zařízeními a procesorem
- stačí minimální počet vodičů
- nízka přenosová rychlosť
- delší časový interval mezi přenášenými daty - třeba data zakódovat a dekódovat (např. checkword u I2C rádiové komunikace)
- základní klasifikace komunikace (standardy):
 - na větší vzdálenosti - RS232 nebo RS485 - mezi řídicím počítačem a podřízenými stanicemi
 - uvnitř elektronického zařízení - I2C (Inter-Integrated Circuit)

Čítače a časovače

- čítač - registr o N bitech
 - čítá vnější události (je inkrementován vnějším signálem dle jeho náběžné nebo sestupné hrany)
 - při jeho přetečení se předá *Interrupt Request* do *Interrupt Subsystem* mikropočítače
 - jeho počáteční hodnota se nastaví programově
 - je možné ho v libovolné chvíli od externího signálu odpojit a opět připojit
- časovač - čítač, který je inkrementován interním hodinovým signálem
 - lze jím zajistit řízení událostí a chování v reálném čase
 - při přetečení se automaticky předa *Interrupt Request*
 - krom počáteční hodnot lze nastavit i předděličku



Obrázek 2: Schéma čítače s přepínačem a časovače s předděličkou

A/D převodníky

- fyzikal. veličiny vstupují do mikropočítače v analog. formě (spojité)
- analog. signály mohou být
 - napětí (U - Voltage/Electric Tension), proud (I - Current), odpor (R - Resistance)
- převede analog. signál do digital. formy
- základní typy:
 - komparační A/D převodník
 - A/D převodník s pomocí D/A převodem

- integrační A/D převodník
- převodník s RC článkem

D/A převodníky

- převede hodnotu z digital. formy do analog. formy
- typy: PWM - Pulse Width Modulation & paralelní převodník

RTC - real time clock

- hodiny reálného času

Speciální periférie

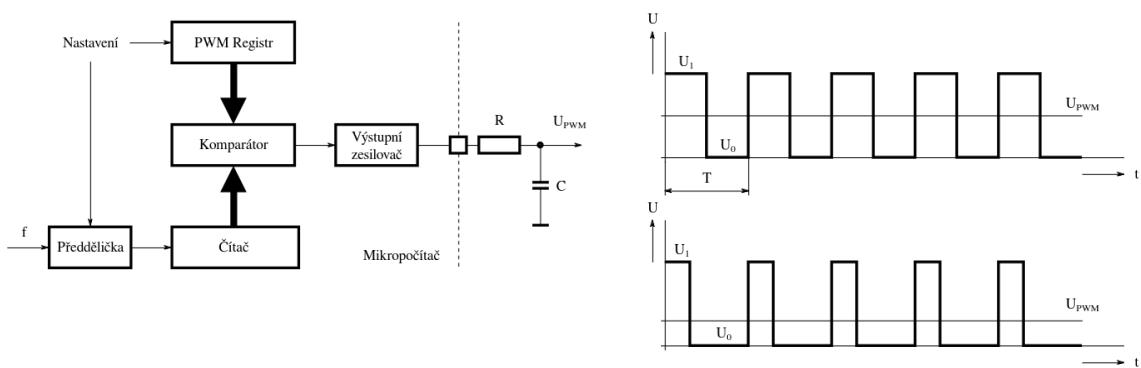
- řízení dobíjení baterií
- dvoutonový multifrekvenční generátor a přijímač
- TV přijímač
- IR vysílač a přijímač
- radiče LCD nebo LED

3. Vysvětlete PWM a kde se používá. Obrázek dobrovolný.

- realizován buď programovou implementací nebo dedikovaným obvodem
- číslicový signál na výstupu mikropočítače má obvykle 2 konstantní napěťové úrovně
 - U_0 pro logickou 0 a U_1 pro logickou 1
- poměrem časů, kdy je výstup na log. 1 a log. 0, se moduluj z digitál. hodnoty analog. signál
 - bude roven střední hodnotě napětí za dobu jedné dané periody
 - čas T_0 - U je na úrovni U_0 neboli napětí reprezentující log. 0
 - čas T_1 - U je na úrovni U_1 neboli napětí reprezentující log. 1
 - perioda - $T = T_0 + T_1$
- střední hodnota napětí, U_{PWM} , je vypočítána vztahem:

$$U_{\text{PWM}} = U_0 + (U_1 - U_0) \cdot \frac{T_1}{T_0 + T_1} \quad \text{nebo} \quad U_{\text{PWM}} = \Delta U \frac{T_1}{T} + U_0$$

- výstup se zesílí výstupním zesilovačem
- pro převod PWM pulsu na analog. veličinu se používá RC článek
 - časová konstanta RC musí být výrazně větší než T (toto způsobuje zpomalení)
- rozlišení výstup. signálu zavисí na počtu bitů komparovaných registrů (*PWM Registr* a *Čítač*)
- využití: kontrola jasu LED diod, síly fénů, větráku, LCD pixelu
- princip u LED/LCD diod: „Lidské oko nevnímá rychlé blikání jako blikání, ale jako jas.“

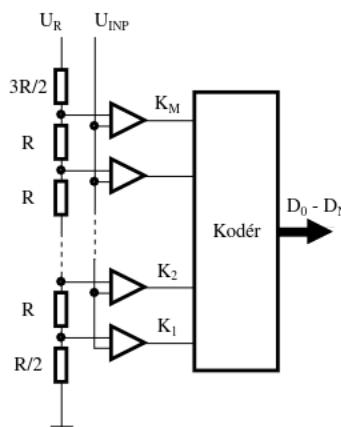


Obrázek 3: Schéma PWM obvodu a přepínání napětí v čase

4. A/D a D/A převodníky a k čemu se používají. Nákres dobrovolný.

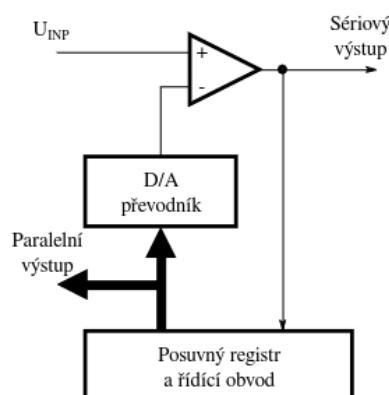
A/D typy:

- **A/D komparační** - srovnání měřené analog. veličiny s referenčními hodnotami napětí a to v určitém poměru ($1 : 2 : 4 : 8 : 16 : 32 : 64 : 128 : 256$) – realizováno odporovou děličkou
 - je to paralelní převodník
 - U_{INP} se chytne k nějakému komparátoru stejného nebo podobného napětí
 - vybraný komparátor bude mít na výstupu 1 a ostatní 0
 - kóder převede tento signál do binárního formátu
 - velmi rychlé - s více komparátory roste přesnost



Obrázek 4: Komparační A/D převodník - odporová dělička

- **A/D převodník s D/A převodem** - jeden komparátor, mění se referenční hodnota
 - podle způsobu řízení ref. hodnoty, dělíme na sledovací a approximační
 - sledovací:
 - najde měřenou hodnotu postupnou inkrementací a dekrementací ref. hodnoty o jeden krok
 - je pomalý - vhodný pro měření pomalu měnících se veličin - teplota, vlhkost
 - approximační:
 - ref. hodnota je na počátku ve středě mezi minimem a maximem měřitelného rozsahu napětí
 - podle výsledku komparátoru měřené hodnoty s ref. hodnotou se vždy posune ref. hodnota nahoru nebo dolů o polovinu zbytku intervalu
 - složitost algoritmu je $O(\log_2 n)$, kde n je počet měřitelných hodnot – jde o binární vyhledávání



Obrázek 5: A/D převodník s D/A převodem

- integrační A/D převodník:

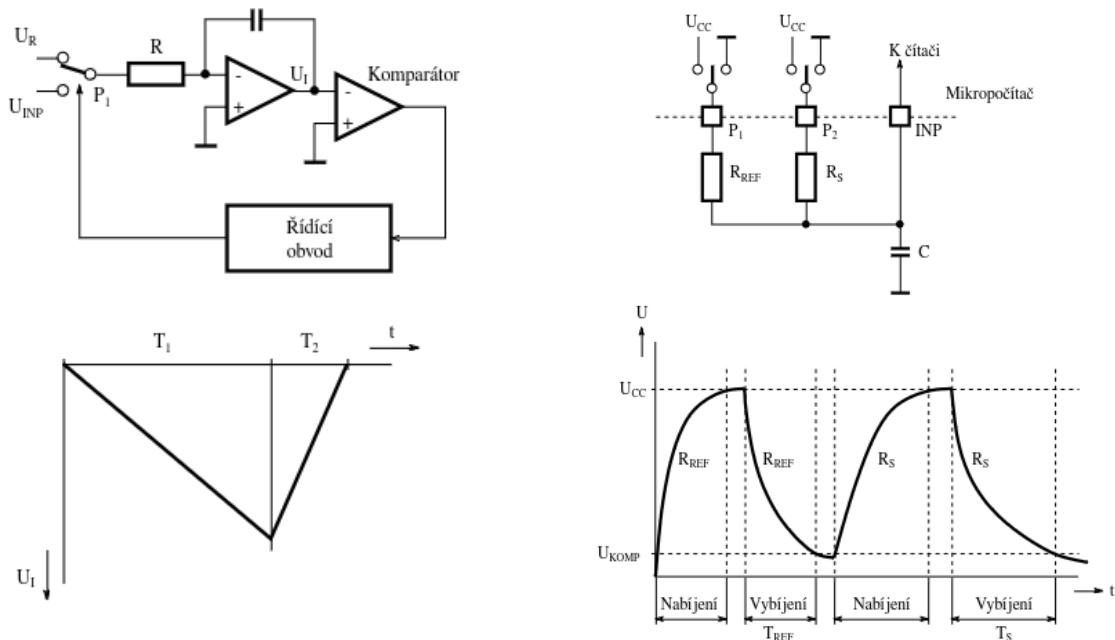
- integrátor integruje vstupní napětí U_{INP} po pevně stanovenou dobu T_1 do U_1
- po skončení T_1
 - se přepne vstup integrátoru P_1
 - integruje se dle ref. napětí U_R opačné polarity k U_{INP}
- nyní se po dobu T_2 integruje U_R dokud U_1 neklesne na $0V$
- doba T_2 je závislá na U_1 na konci T_1 - z ní lze získat hodnotu měřeného napětí:

$$U_{\text{INP}} = - \left(\frac{T_2}{T_1} \cdot U_R \right)$$

- A/D převodník s RC článkem:

- na vstupu měří odpor R_{INP} ne napětí - např. tenzometr, termistor
- princip:
 - necháme nabíjet kondenzátor přes ref. odpor R_{REF} dokud U_C v kondenzátoru nedosáhne U_{CC}
 - teď necháme konden. C vybíjet
 - přes stejný odpor dokud U v konden. neklesne na hodnotu U_{KOMP}
 - přičemž měříme čas vybíjení T_{REF}
 - to samé uděláme s měřeným odporem R_{INP} - získáme tím čas vybíjení T_{INP} (na obrázku T_s)
 - hodnotu vstupního napětí, R_{INP} , získáme vztahem:

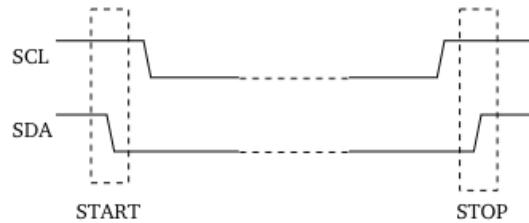
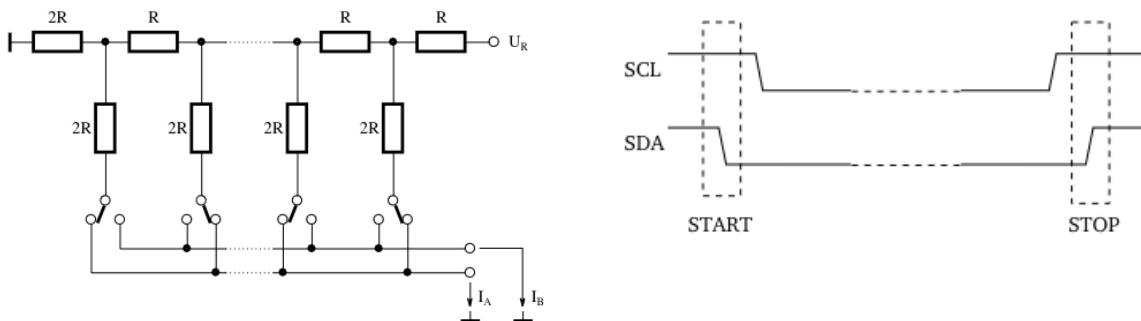
$$R_{\text{INP}} = R_{\text{REF}} \cdot \frac{T_{\text{INP}}}{T_{\text{REF}}} \Leftrightarrow \frac{R_{\text{INP}}}{R_{\text{REF}}} = \frac{T_{\text{INP}}}{T_{\text{REF}}}$$



Obrázek 6: Integrační ADC - schéma obvodu, znázornění růstu U_1 & ADC s RC článkem, znázornění napětí v kondenzátoru v čase

D/A převodníky

- PWM (viz otázka na PWM)
- **paralelní D/A převodník**
 - je rychlý
 - založeny na přímém převodu digitální hodnoty na analog. veličinu
 - základem je odporová síť, na níž se vytvářejí částečné výstupní proudy:
 - váhově řazené hodnoty - rezistory s odpory v poměrech $1 : 2 : 4 : \dots : 64 : 128$
 - R-2R - stačí rezistory s odpory R a $2R$
- digitalní hodnota přepína přepínače pod $2R$ rezistoru
- výstupem je el. proud I_A a jeho komplementární (znegovaný) proud I_B



Obrázek 7: Paralelní D/A převodník řešený pomocí R-2R
& Znázornění START a STOP řídicích signálů na SCL a SDA vodičích

5. I2C - co to je, jak funguje, kde se používá a naskreslit.

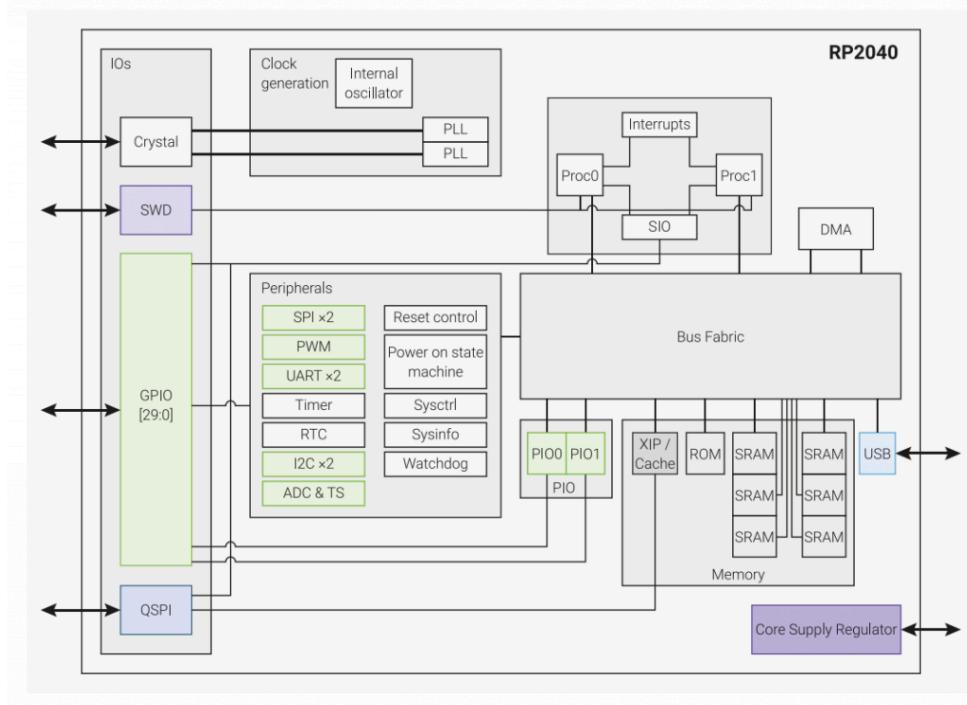
- je sériová komunikační sběrnice
 - umožňuje přenos dat mezi různými zařízeními
- byla vyvinuta firmou Phillips
 - stala se populární mezi integrovanými obvody (*IC - integrated circuit*) a perifer. zařízeními
 - pro svou jednoduchost a snadnou rozšířitelnost
- funguje na základě 2 obousměrných vodičů (ty mohou nabývat hodnot log. 0 a log. 1):
 - SDA (Serial Data Line) - pro přenos dat mezi zařízeními, data jsou zasílána sériově po bitech
 - SCL (Serial Clock Line) - pro synchronizaci přenosu
- funguje ve formě přenosu dat mezi „Master“ a „Slave“ zařízeními
 - **Master** - zodpovědný za řízení komunikace, iniciuje přenos
 - **Slave** - řízení od „Master“ přijímá a vykoná (vykoná funkci / poskytuje data)
- princip fungování:
 - v klidovém stavu obě na log. 1
 - komunikace se zahajuje řídicím signálem **START** - přivedením SDA na 0, hned po ní SCL na 0
 - ukončí se řídicím signálem **STOP** - SCL na log. 1 a hned po ní SDA na log. 1
 - musíme na začátku komunikace adresovat „Slave“ zařízení, se kterým chceme komunikovat, a zadat směr komunikace - zda chceme číst (RD) od nebo zapisovat (WR) do „Slave“ zařízení:
 - po SDA předáme adresu zařízení s řídicím bitem RD nebo WR jako 1 byte dat
 - 7 bitů slouží pro adresování zařízení a 1 bit (LSB) pro směr komunikace
 - pokud adresované zařízení zaznamená, vyšle signál ACK (log. 0) po datovém vodiči
 - zápis/write - posílame byte postupně po bitech - po každém bytu dat musí „Slave“ vyslat ACK
 - čtení/read - očekáváme data od zařízení - po každém bytu, který přijmem, vyšlem ACK

6. Popiš a nakresli schéma mikropočítače, se kterým ses seznámil.

Raspberry Pi RP2040

[specifikace přímo od Raspberry Pi]

[obrázek monolitu RP2040 přímo od Raspberry Pi]



Obrázek 8: Schéma mikropočítače / mikroprocesoru / monolitu / monolitického počítače RP2040

- dual ARM Cortex-M0+ - 2 jádra
- taktovací frekvence 133MHz
- SRAM - 264kB, 6 na sobě nezávislých bank
- až 16MB pro off-chip Flash paměť s programem - přes QSPI port
- DMA řadič
- fully connected AHB (*Advanced High-performance Bus*)
 - propojovací síť všech komponent s procesorem
- LDO (*Low-Dropout Regulator*) - pro generování core voltage supply
- PLL (*Phased-Locked Loops*) - pro generování hodinového signálu pro USB rozhraní a core clock
- GPIO (*General Purpose IO*) - piny pro obecné připojení periferií
- periférie:
 - UART (*Universal Asynchronous Receiver-Transmitter*)
 - SPI (*Serial Peripheral Interface*)
 - I2C (*Inter-Integrated Circuit*)
 - PWM (*Pulse Width Modulation*)
 - PIO (*Programmable I/O*) - pro naprogramování vlastního protokolu komunikace
 - RTC (*Real Time Clock*)
 - Watchdog
 - Reset Control
 - Timer
 - Sysinfo & Syscontrol
 - ADC (*A/D converter*)

Disky

Otázky:

- 7) Fyzikální popis HDD čtení, zápis a nákres. Vysvětlit podélný a kolmý zápis.
- 8) Popište a nakreslete stavbu disku. Nechtěl zápis.
- 9) Čtení CD - princip a obrázek.

7. Fyzikální popis HDD - čtení, zápis a nákres. Vysvětlit podélný a kolmý zápis.

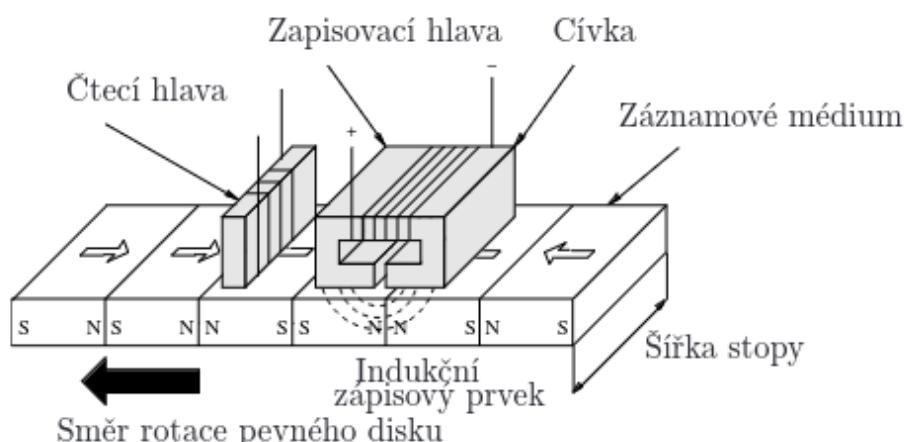
- médium HDD (*Hard Disk Drive*), na kterém se data ukládají, je feromagnetická vrstva nanesena na plotnu disku (většinou ze skla / slitiny hliníku)
- pracuje s magnetickým záznamem - tím zaznamenaná data
- feromagnetická vrstva dokáže uchovat magnetická pole
- záznamová/zapisovací hlava - jádro s úzkou štěrbinou ($1\mu\text{m}$) a navlečenou cívku
 - vrstva feromagnetická je trvale zmagnetována záznamovou hlavou
 - v bodu dotyku hlav (zapisovací a čtecí) nebo v nepatrné vzdálenosti s médiem je štěrbina

Zápis na disk

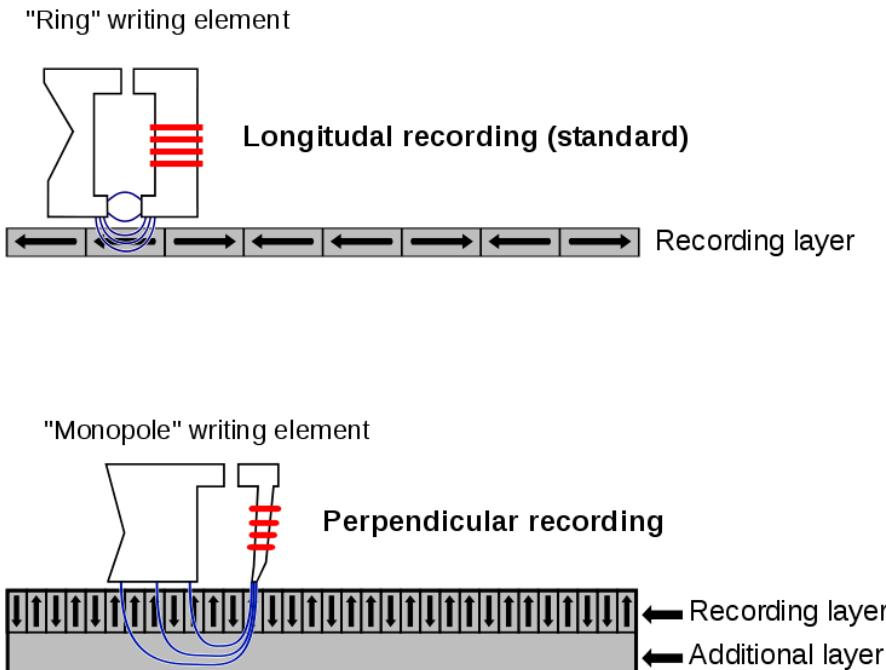
- při průchodu el. proudu cívkou proudí magnet. tok jádrem
- jádro v je části, kde je nejblíže záznamové vrstvě, přerušeno úzkou štěrbinou vyplněnou nemagnetickou látkou (nejčastěji bronz) nebo „ničím“ (vzduchem)
- v místě štěrbiny dochází k magnetickému stínění jádra a následnému vychýlení indukčních čar z jádra cívky do feromagnetické vrstvy disku
- měněním směru el. proudu v cívce se mění směr magnet. toku jádrem i štěrbinou a tím smysl magnetizace aktivní vrstvy

Čtení z disku

- při čtení se disk pohybuje stejným směrem konstantní rychlostí
- na aktivní feromagnetické vrstvě jsou místa magnetizovaná tím či oním směrem - mezi nimi jsou místa magnetického přechodu - tzv. „magnetické rezervace“
 - právě ony představují zapsanou informaci
 - změny mag. polí na feromag. vrstvě způsobují napěťové impulsy na svorkách cívky čtecí hlavy
- impulsy jsou následovně zesíleny elektrickými zesilovači



Obrázek 9: Princip magnetického zápisu na feromagnetickou vrstvu disku



Obrázek 10: Podélný a kolmý zápis aktivní vrstvy pevného disku

Podélný zápis (longitudinální zápis)

- způsob, jakým byla data tradičně zapisována na pevné disky
- magnetická pole, která reprezentují jednotlivé bity dat, jsou orientována podél povrchu disku
- data jsou zapsána v podélných stopách na disku, které jsou rozděleny na sektory

Kolmý zápis (perpendikulární zápis)

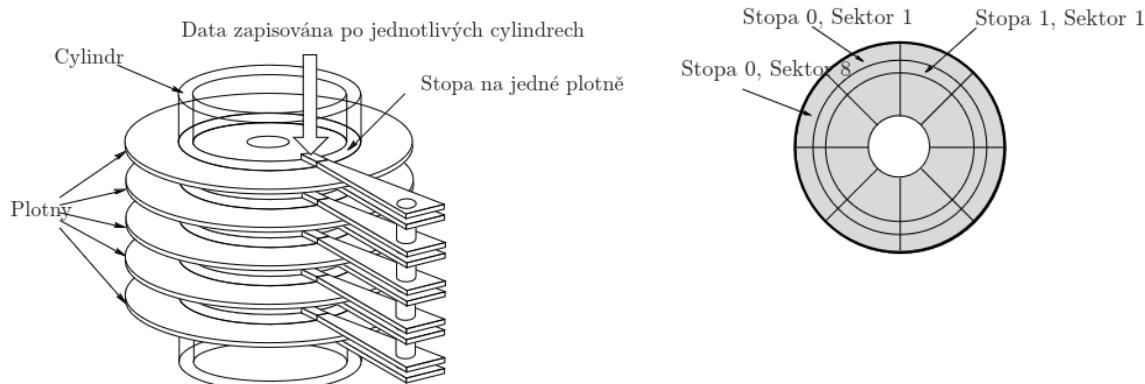
- modernější způsob zápisu na pevný disk, který umožňuje vyšší kapacitu a rychlosť zápisu
- při kolmém zápisu jsou magnetická pole orientována kolmo na povrch disku
- to umožňuje menší a hustší záznam dat na povrchu disku - zvyšuje kapacitu pevného disku
- při kolmém zápisu jsou magnetická pole stabilnější a méně náchylná k rušení

8. Popište a nakreslete stavbu pevného disku. Nechtěl podélný a kolmý zápis.

- je to uzavřená jednotka v počítači používaná pro trvalé ukládání dat (nevolutilní paměť)
- pouzdro chrání disk před nečistotami a poškozením
- obsahuje nevýjmutelné pevné plotny diskového tvaru (slitiny hliníku / sklo) - odtud *pevný disk*
- části pevného disku:
 - plotny disku
 - hlavy pro čtení a zápis
 - vzduchové filtry
 - pohon hlav
 - pohon ploten disku
 - řídící deska (deska s elektronikou)
 - kably a konektory

Geometrie disku

- uspořádání prostoru na disku - počet hlav (zapisovací a čtecí), cylindrů a stop
- data jsou na disk ukládána v bytech
 - ▶ byty jsou uspořádány do skupin po 512 bytech (nové 4KiB) zvané sektory
- sektor je nejmenší jednotka dat, kterou lze na disk zapsat nebo z disku přečíst
 - ▶ sektory jsou seskupeny do stop
 - stopy jsou uspořádány do skupin zvaných cylindry nebo válce
- předpokladem je, že jeden disk má nejméně dva povrchy (dolní a horní plocha plotny)
- systém adresuje sektory na pevném disku pomocí prostorové matice cylindrů, hlav a sektorů



Obrázek 11: Geometrie pevného disku a popis plotny

Stopy

- každá strana každé plotny je rozdělena na soustředné stopy (kružnice)
- protože povrchů i hlav je několik, je při jedné poloze hlav přístupná na každém povrchu jedna stopa - pro vybrání jedné stopy stačí elektronicky přepínat hlavy

Cylindry

- pevné disky mají více ploten (disků), umístěných nad sebou, otáčejících se stejnou rychlosí
- každá plotna má dvě strany (povrchy), na které je možno data ukládat
- diskové hlavy nemohou být vystavovány nezávisle (jsou pohybovány společným mechanismem)
- souhrn stop v jedné poloze hlav se nazývá cylindr (válec)
- počet stop na jednom povrchu je totožný s počtem cylindrů
- z tohoto důvodu většina výrobců neuvádí počet stop, ale počet cylindrů

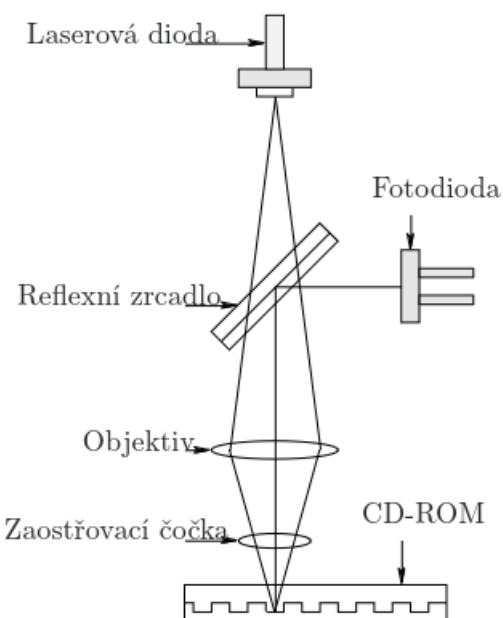
Sektory

- jedna stopa je příliš velkou jednotkou pro ukládání dat (100KiB či více bytů dat)
 - ▶ stopa se rozděluje na několik očíslovaných částí nazývané sektory
 - ▶ můžeme si je představit jako výseče na plotně
 - ▶ je to nejmenší adresovatelná jednotka na disku - na rozdíl od hlav nebo cylindrů, číslujeme od 1
 - ▶ její velikost určí řadič při formátování disku
- na začátku sektoru je hlavička identifikující začátek sektoru a obsahující jeho číslo
- konec - tzv. zakončení sektoru - pro ukládání kontrolního součtu (*ECC - Error Correcting Code*)
 - ▶ slouží ke kontrole integrity uložených dat
- jednotlivé sektory se oddělují mezisektorovými mezerami - zde není možné data uložit
- proces čtení sektoru se skládá ze dvou kroků:
 - ▶ čtecí a zápisová hlava musí přemístit nad požadovanou stopu
 - ▶ potom se čeká, až se disk natočí tak, že požadovaný sektor je pod hlavou, a pak probíhá čtení

- přemístění hlavy obvykle zabere nejvíce času
- nejrychleji se tedy čtou soubory, jejichž sektory jsou všechny na stejně stopě a stopy jsou umístěny nad sebou v jednom cylindru

9. Čtení CD - princip a obrázek.

- jako materiál CDčka se používá polykarbonát, strana se záznamem je pokryta reflexní vrstvou a ochranným lakem
 - záznam je v podobě pitů (prohlubně) a polí (ostrůvky) na disku
- čtení zaznamenaných dat probíhá způsobem, kdy laser v přehrávači CD snímá z povrchu disku zaznamenaný vzor
- mechanika CD:
 - laser je umístěn rovnoběžně s povrchem disku
 - paprsek je na disk odrážen zrcadlem přes dvě čočky
 - lze velmi úzce zaostřit na malé plochy disku CD-ROM
 - fotodetektor pak měří intenzitu odraženého světla
 - laser nemůže poškodit nosič a na něm uložená data
 - při čtení se paprsek odráží od lesklého povrchu disku CD-ROM a nijak ho nepoškodí
 - mechanismus laseru je od disku asi jeden milimetr



Obrázek 12: CD mechanika - princip zápisu a čtení

- čtení dat z CD média probíhá za pomocí laserové diody
 - emituje infračervený laserový paprsek směrem k pohyblivému zrcátku
 - čtenou stopu přesune servomotor pod zrcátko na základě příkazů z mikroprocesoru
 - po dopadu paprsku na jamky a pevniny, resp. pity a pole, se světlo láme a odráží zpátky
 - dále je zaostřováno čočkou, nacházející se pod médiem
 - od čočky světlo prochází pohyblivým zrcátkem - reflexní zrcadlo
 - odražené světlo dopadá na fotocitlivý senzor - fotodioda
 - převádí světelné impulsy na elektrické
 - samotné elektrické impulsy jsou dekódovány mikroprocesorem a předány do počítače ve formě dat v binárním formátu

Zobrazovací jednotky

Otázky:

- 10) Popište a nakreslete technologii LCD - výhody, nevýhody, rozdíl mezi pasivním a aktivním.
- 11) Popište a nakreslete technologii OLED - výhody, nevýhody.
- 12) Popsat E-ink - jaké má barevné rozmezí, výhody a nevýhody.
- 13) Vybrat která zobrazovací jednotka je podle tebe technicky nejzajímavější a proč (OLED, LCD, E-ink).

10. Popište a nakreslete technologii LCD - výhody, nevýhody, rozdíl mezi pasivním a aktivním.

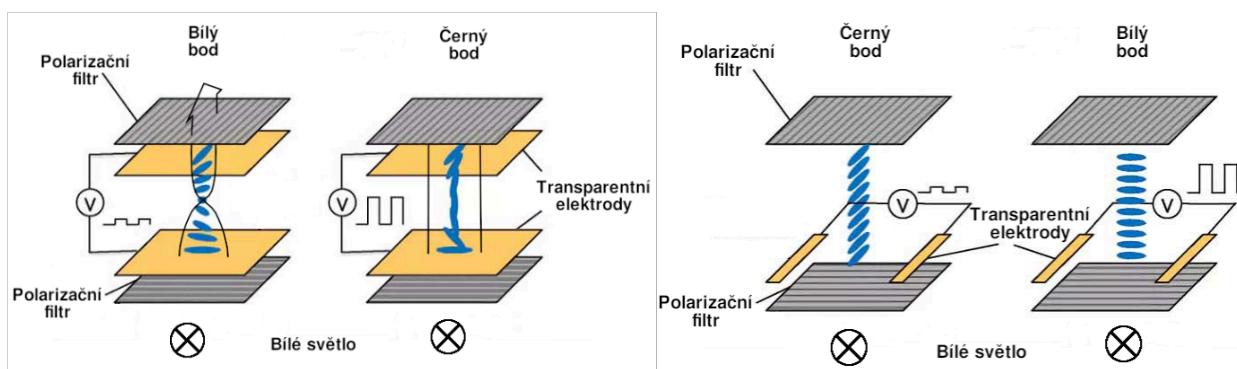
- LCD - *Liquid Crystal Display*
- používá tekuté krystaly k zobrazení jednotlivých pixelů
- v základě jsou dvojího typu:
 - TN-TFT - *Twisted-Nematic Thin-Film-Transistor*
 - IPS - *In-Place-Switching*

Princip TN-TFT LCD (*Twisted-Nematic Thin-Film-Transistor LCD*)

- 1) světlo projde polarizačním filtrem a polarizuje se
 - 2) projde vrstvami tekutých krystalů (uspořádaných do šroubovice) - světlo se otočí o 90°
 - 3) projde druhým polarizačním filtrem (které je otočené o 90° proti prvnímu)
- klidový režim (bez napětí) - propouští světlo
 - přivede-li se napětí, krystalická struktura (šroubovice) se zorientuje podle směru toku proudu
 - světlo projde prvním polarizačním filtrem, neotočí se → je definitivně zablokováno
 - střídáním proudu lze určit intenzitu propouštěného světla
 - nutno podsvítit bílým světlem (elektroluminiscenční výbojky, LED, OLED)
 - vrstva krystalů je rozdělena na malé buňky stejné velikosti, tvořící pixely

Princip IPS LCD (*In-Place-Switching LCD*)

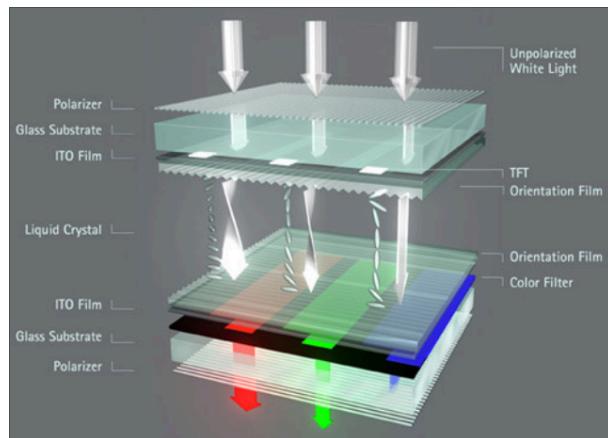
- podobné TN
- krystaly jsou uspořádány v rovině
- elektrody jsou po obou stranách buňky v jedné vrstvě
- přivede-li se napětí, krystaly se začnou otáčet ve směru elek. proudu - otočení celé roviny krystalů
 - tím otočí i světlo, které jím procházelo, a propustí se druhým polarizačním filtrem
- klidový stav - světlo neprochází přes 2. pol. filtr - „nesvítí“



Obrázek 13: Princip činnosti TN-TFT LCD a IPS LCD displeje

Barevné LCD

- každý pixel se skládá ze 3 menších bodů (subpixelů) obsahující Red, Green, Blue filtr
- propouštěním světla do barevných filtrů a složením barev dostaneme výslednou barvu pixelu



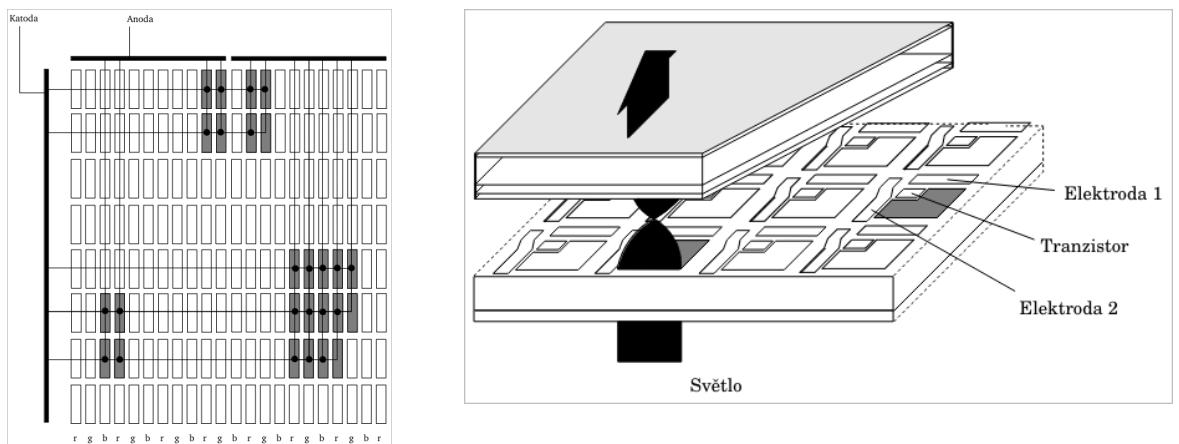
Obrázek 14: Barevný LCD - průchod bílého světla přes barevné filtry

Pasivní matice LCD

- obsahuje mřížku vodičů, body se nacházejí na průsečících mřížky
- při vyšším počtu bodů narůstá potřebné napětí → rozostřený obraz, velká doba odevzdy (3 FPS) nevhodné pro hry, filmy, televizi atd.
 - z jediného rosvíceného bodu se rozbíhají postupně slábnoucí vertikální a horizontální čáry
- používá se v zařízeních s malým displejem (hodinky)

Aktivní matice LCD

- každý průsečík v matici obsahuje svůj tranzistor nebo diodu - řeší řídicí činnost daného bodu
- pomocí tranzistoru ve spolupráci s kondenzátorem lze rychle a přesně ovládat svítivost \forall bodu
- TF (*Thin Film*) tranzistory izolují jeden bod od ostatních - eliminace „čár“ na pasivním LCD



Obrázek 15: Struktura pasivního a TFT (aktivního) displeje

Výhody:

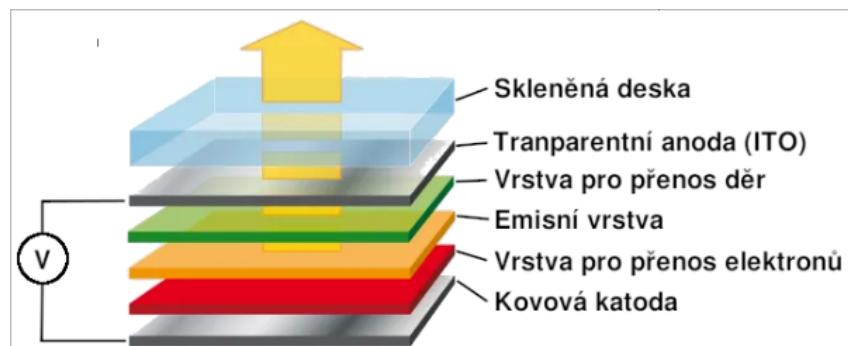
- kvalita obrazu
- životnost
- spotřeba energie
- odrazivost a oslnivost
- bez emisí

Nevýhody:

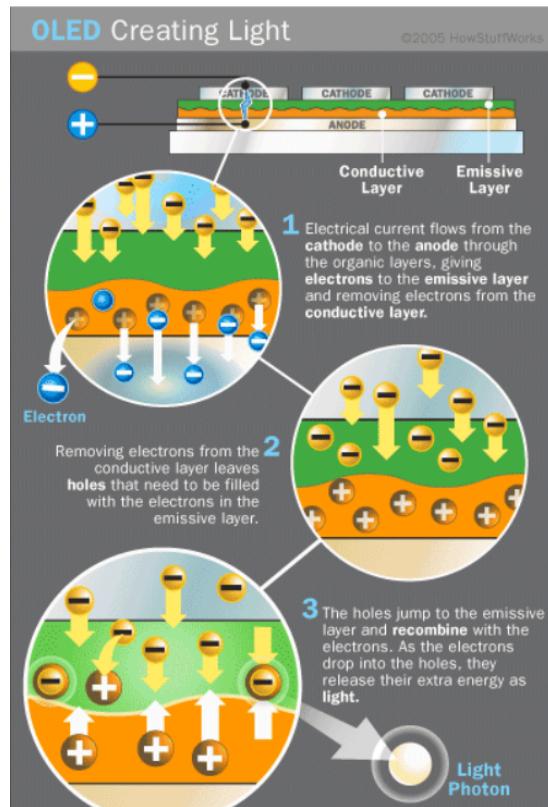
- citlivost na teplotu
- pevné rozlišení
- vadné pixely
- doba odevzdy

11. Popište a nakreslete technologii OLED - výhody, nevýhody.

- hlavním prvkem - organická dioda emitující světlo (*Organic Light Emitting Diode*)
- po přivedení napětí na obě elektrody se začnou elektrony hromadit v org. vrstvě blíže k anodě
 - anoda přítahuje elektrony
 - tím vznikají „díry“ - kladné částice / absence elektronů - ve „vrstvě pro přenos děr“
- díry představující kladné částice se hromadí na opačné straně blíže ke katodě
 - katoda odpuzuje elektrony
 - tím se ve „vrstvě pro přenos elektronů“ hromadí elektrony
- v organické vrstvě („emisní vrstvě“) začně docházet ke „srážkám“ mezi elektronami a dírami
 - elektrony zaplňí „díry“
 - to způsobí jejich vzájemnou eliminaci - **rekombinace**
 - doprovází vyzáření energie ve formě fotonu, které vnímáme jako světlo
- měněním napětí, které do diody přivádíme, způsobuje změnu jasu diody
 - čím větší napětí, tím více vzniká „děr“ → vyšší výskyt rekombinace → více vyzářených fotonů



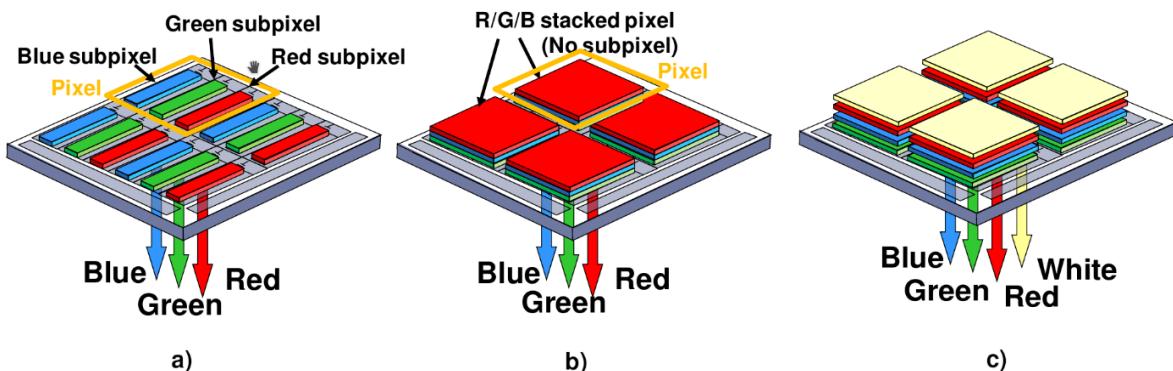
Obrázek 16: Základní struktura OLED diody



Obrázek 17: Princip činnosti organické vrstvy OLED

Barevný OLED

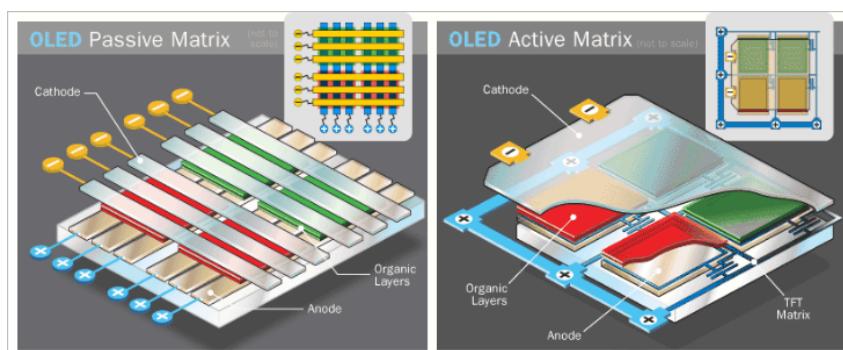
- skládání jednotlivých pixelů ze tří základních barev
- je více možností složení barev (jak je seskládat):
 - a) standardně je naskládat vedle sebe
 - b) vertikální uspořádání
 - c) vertikální uspořádání s bílou složkou



Obrázek 18: Způsoby naskládání barevných složek jednoho pixelu v OLED displejích

AMOLED vs. PMOLED (*Active / Pasivní Matrix OLED*)

- stejný princip jako aktivní / pasivní LCD
- body organizovány do pravoúhlé matice
- ▶ pasivní - každá OLED je aktitována dvěma na sebe kolmými elektrodami, procházejícími celou šírkou a výškou displeje
- ▶ aktivní - každá OLED aktitována vlastním tranzistorem (*TFT - Thin Film Transistor*)



Obrázek 19: Technologie OLED - pasivní (PMOLED) a aktivní (AMOLED)

Výhody

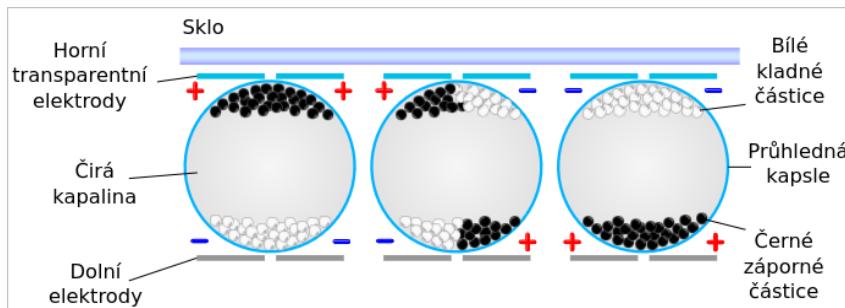
- vysoký kontrast
- velmi tenké
- plně barevné
- nízká spotřeba
- dobrý pozorovací úhel
- bez zpoždění
- možnost instalace na pružný podklad

Nevýhody

- vyšší cena

12. Popsat E-ink - jaké má barevné rozmezí, výhody a nevýhody.

- technologií E-ink používají zařízení EPD (Electronic Paper Device)
 - EPD nepotřebují elektrický proud pro statické zobrazování
- inkoust tvořen mikrokapslemi (~ desítky-stovky μm)
- částice v kapslích se přitahují k elektrodě s opačnou polaritou
- roztok - hydrokarbonový olej (díky jeho viskozitě vydrží částice na místě i po odpojení napájení)
- „stavba“ kapsle:
 - černé záporné částice jsou z uhlíku – C
 - bílé kladné částice z oxidu titaničitého – TiO_2
 - obal z oxid křemičitého – SiO_2 – a tenké vrstvy polymeru
 - jako elektroforetický roztok (elektricky separovatelný) se používá hydrokarbonový olej
- k pohybu častic je potřeba proud ~desítky nA při napětí 5-15 V
- pro barvy se používají barevné filtry (stejně jako u LCD)
 - barevná hloubka - je závislá na počtu elektrod na jeden zobrazovací bod
 - $(2^n)^c$ kde n je počet elektrod a c počet barevných složek (např. $(2^4)^3 = 16^3 = 4096$)



Obrázek 20: Technologie E-ink - pohled na kapsle ze strany

Výhody

- vysoké rozlišení
- dobrý kontrast
- čitelnost na přímém slunci
- není nutné podsvětlení
- velký pozorovací úhel
- velmi tenké
- možno používat i na pružném podkladu
- nulova spotřeba proudu přiobrazení statické informace
- minimální spotřeba při překreslení

Nevýhody

- málo odstínů šedí
- špatné barevné rozlišení
- velké zpoždění

13. Vybrat která zobrazovací jednotka je podle tebe technicky nejzajimavější a proč (OLED, LCD, E-ink).

- viz předchozí otázky

RISC

Otázky:

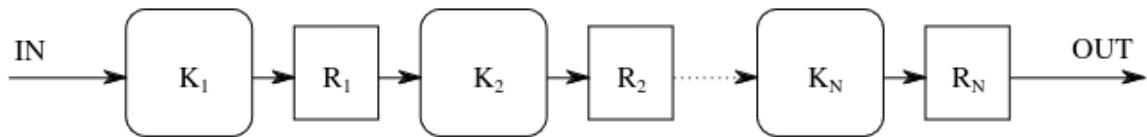
- 14) Popište na RISC procesoru zřetězené zpracování instrukcí, jaké má chyby a jak se řeší.
- 15) Popište na RISC procesoru zřetězené zpracování instrukcí a jak nám pomůže predikce skoku.
- 16) Jaké problémy a hazardy mohou nastat u RISC.
- 17) Popiš základní konstrukci a vlastnosti mikroprocesoru RISC.
- 18) Popiš a nakresli schéma RISC procesoru, se kterým ses seznámil.

14. Popište na RISC procesoru zřetězené zpracování instrukcí, jaké má chyby a jak se řeší.

- procesor je sekvenční obvod
 - vstup - instrukce a data z paměti
 - výstup - výsledky uloženy do paměti
- instrukce jsou vždy zpracovány stejným způsobem v několika fázích, např.:
 - 1) **VI** – Výběr instrukcí z paměti (Instruction Fetch)
 - 2) **DE** – Dékódování instrukce (Instruction Decoder)
 - 3) **VA** – Výpočet adresy operandů (Operand Address Calculation)
 - získá se adresa operandů, se kterou instrukce pracuje
 - 4) **PI** – Provedení instrukce (Instruction Execution)
 - 5) **UV** – Uložení výsledku zpět do paměti (Store Result)
- instrukce projde všemi těmito fázemi - pokud by trvala každá fáze 1 stroj. cyklus, tak by se 1 instrukce vykonala za 5 stroj. cyklů
- instrukce I_2 se nemůže vykonat, když procesor zpracovává inst. I_1
- osamostatněním jednot. fází vlastními obvody - je možné instr. zřetězit
 - zatímco **VI** vybírá instrukci z paměti, může **DE** dekódovat instr., kterou před jedním stroj. cyklem vybrala **VI** z paměti
 - teoreticky se tím zvýší výkon o násobek hloubky zřetězení
 - tomuto zrychlení avšak zabírají podmíněné skoky, datové a strukturální hazardy
 - podmíněné skoky:
 - neví se, kdy se skok provede a kdy ne
 - adresa IP se změní - rozpracované instrukce jsou neplatné - musí se *flushout* fronta instrukcí - **problém plnění fronty**
 - existují mechanismy jak tomu předcházet
 - predikce skoku (*Branch Prediction*)
 - jednobitová - v instrukci skoku je 1 bit vyhrazen pro *flag*, který předpovídá, jestli se bude či nebude skákat (přepíná se po jednom vykonání a nevykonání skoku)
 - dvoubitová - v instrukci jsou vyhrazeny 2 bity (přepíná se až po dvou konsekutivních vyhodnocení skoku)
 - zpozdění skoku - pokud možno, vykonají se instruce jiné ještě před instrukcí skoku (i přesto, že jim v programu instrukce skoku předchází)
 - buffer s pamětí skoků (*BTB – Branch Target Buffer*) - pamatuje si tisíce tzv. *target* adres skoků
 - strukturální hazardy:
 - pomalé sběrnice a registry mezi jednotkami jednot. fází
 - musí se koordinovat přístup ke sběrnici
 - datové hazardy:
 - instrukce potřebuje výsledky od instrukce, která ještě nebyla vykonána

	T ₁	T ₂	T ₃	T ₄	T ₅	T ₆	T ₇	T ₈	T ₉	T ₁₀	T ₁₁	T ₁₂	T ₁₃
VI	I ₁						I ₂						...
DE		I ₁						I ₂					
VA			I ₁						I ₂				
VO				I ₁						I ₂			
PI					I ₁						I ₂		
UV						I ₁						I ₂	

Obrázek 21: Sériové zpracování instrukcí - CISC



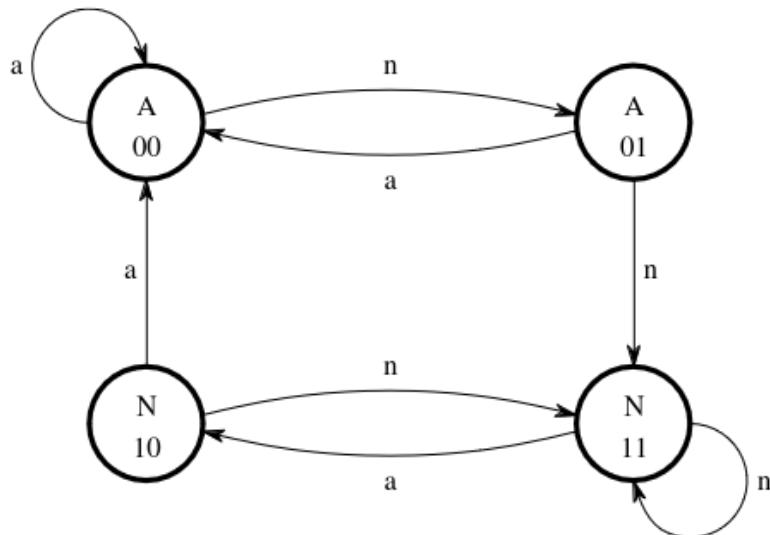
Obrázek 22: Rozdělení obvodu pro zpracování instrukcí na jednot. fázové jednotky

	T ₁	T ₂	T ₃	T ₄	T ₅	T ₆	T ₇	T ₈	T ₉	T ₁₀	T ₁₁	T ₁₂
VI	I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇	...				
DE		I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇				
VA			I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇			
VO				I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇		
PI					I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇	
UV						I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇

Obrázek 23: Zřetězené zpracování instrukcí - RISC

15. Popište na RISC procesoru zřetězené zpracování instrukcí a jak nám pomůže predikce skoku.

- viz 14. otázku - jak funguje zřetězené zpracování instrukcí
- typy řešení predikcí skoků - jednobitová, dvoubitová, BTB:
 - **jednobitová** predikce:
 - ve formátu instrukce skoku se vyhradí jeden bit pro uložení stavu *flagu* predikující, zda se skok vykoná či ne
 - buď se *flag* nastaví *staticky* programátorem/kompilátorem (*hard coded*)
 - nebo se nastavuje při běhu programu *dynamicky* dle výsledku podmínky předcházejícího skoku
 - v cyklu k selhání predikce dojde vždy 2x - první a poslední iterace
 - **dvoubitová** predikce:
 - vyhradí se dva bity - 4 možné hodnoty/stavy
 - funguje jako stavový automat se 4 stavy - NE = nebude se skákat, ANO = bude se skákat
 - 00 (ANO) - stálý stav skákání
 - 01 (ANO) - jeden neprovedený skok
 - 10 (NE) - jeden provedený skok
 - 11 (NE) - stálý stav „neskákání“
 - přechody *a* a *n* ukazují jestli se naposledy skákalo či ne
 - v cyklu se omezí počet selhání na jeden



Obrázek 24: Dvoubitová predikce - čtyřstavový automat

- **BTB (Branch Target Buffer):**
 - tabulka s uloženými adresami provedených podmíněných skoků
 - ať už jednobitová nebo dvoubitová predikce
 - většinou implementována přímo na procesorech
 - může mít až tisíce položek

16. Jaké problémy a hazardy mohou nastat u RISC.

- viz předchozí otázky - popsaný datové a strukturální hazardy, problematika zpracování podmíněných skoků

17. Popiš základní konstrukci a vlastnosti mikroprocesoru RISC.

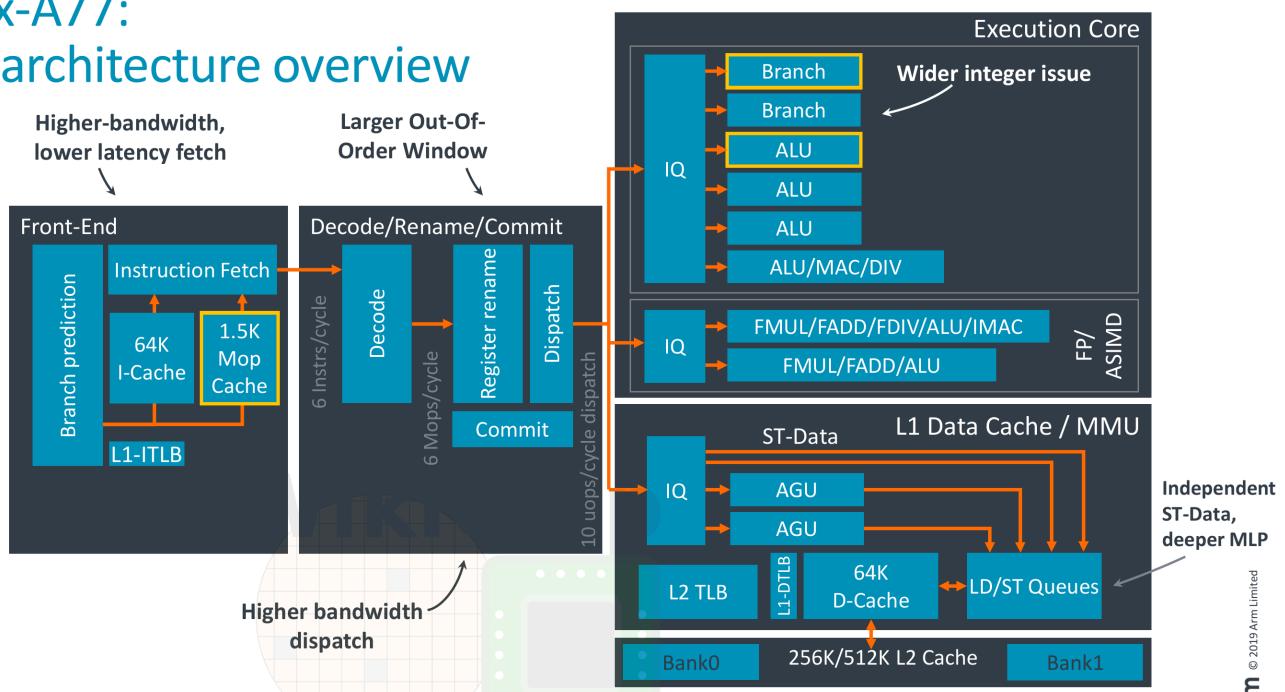
- mají malý instrukční soubor
- vývojová větev RISC vyvinula řadu zásadních kritérií, charakterizujících metodiku návrhu nejen procesoru, ale i celého počítače
- procesoru se má přenechat jenom ta činnost, která je nezbytně nutná
 - další potřebné funkce přenést do architektury počítače, programového vybavení a komplilátoru
- výsledkem návrhu jsou zejména tyto vlastnosti:
 - jedna instrukce dokončena každý strojový cyklus
 - mikroprogramový řadič (software) nahrazen rychlejším obvodovým řadičem (fyz. dráty)
 - zřetězené zpracování instrukcí
 - počet instrukcí a způsobů adresování je malý
 - data jsou z hlavní paměti vybírána a ukládána výhradně pomocí dvou instrukcí LOAD a STORE
 - instrukce mají pevnou délku a jednotný formát
 - použit vyšší počet registrů
 - složitost se z technického vybavení přesouvá částečně do optimalizujícího komplilátoru
- všechny uvedené vlastnosti tvoří dobře promyšlený a provázaný celek:
 - navýšení počtu registrů & omezení komunikace s pamětí na dvě instrukce LOAD a STORE
 - ostatní instrukce nemohou používat pam. operandy - třeba mít v procesoru více dat
 - zřetězené zpracování & formát instrukcí
 - jednotná délka instrukcí dovoluje rychlejší výběr instrukcí z paměti
 - zajišťuje lepší plnění fronty instrukcí
 - jednotný formát zjednodušuje dekódování instrukcí
 - výsledné počítače vytvořené podle těchto pravidel přinášejí výhody pro uživatele i pro výrobce
 - zkracuje se vývoj procesoru - již první realizované čipy fungují správně (většinou)
 - architektura RISC má i své nedostatky
 - nutný nárůst délky programů - tvořený omezeným počtem instrukcí jednotné délky
 - zpomalení, které by z toho mělo nutně plynout, se ale v praxi nepotvrdilo
 - procento instrukcí, které muselo být rozepsáno, je malé
 - většina výrobců CISC procesorů se uchýlila při výrobě procesorů k realizaci stále většího počtu vlastností arch. RISC

18. Popiš a nakresli schéma RISC procesoru, se kterým ses seznámil.

ARM Cortex A77

- frekvence 3 GHz
- ARM v8-A architektura (harvardká arch.)
- 64-bitová instrukční sada se SIMD rozšířením
- 13-ti úrovňové zřetězení
- 8 jader
- cache:
 - L1 - 2x64 KiB na jádrech pro instrukce a data
 - L2 - 256 nebo 512 KiB - vyrovnávací paměť mezi procesorem a hlavní pamětí (RWM SDRAM)
- out-of-order vykonávaní instrukcí - reorder buffer může mít 160 položek
- výpočetní jednotky: 4x ALU, FPU, ASIMD, 2x Branch
- prediktor větvění ~8000 položek
- macro-OP cache ~1500 položek
 - ukládá již dekódované instrukce
 - urychluje vykonávání smyček - nemusí se vždy instrukce uvnitř cyklu dekódovat znova
 - dekóder - 6 instrucí / cyklus
- využití:
 - mobilní zařízení s Android
 - SoC (System On Chip) - mikrokontroléry/monoly

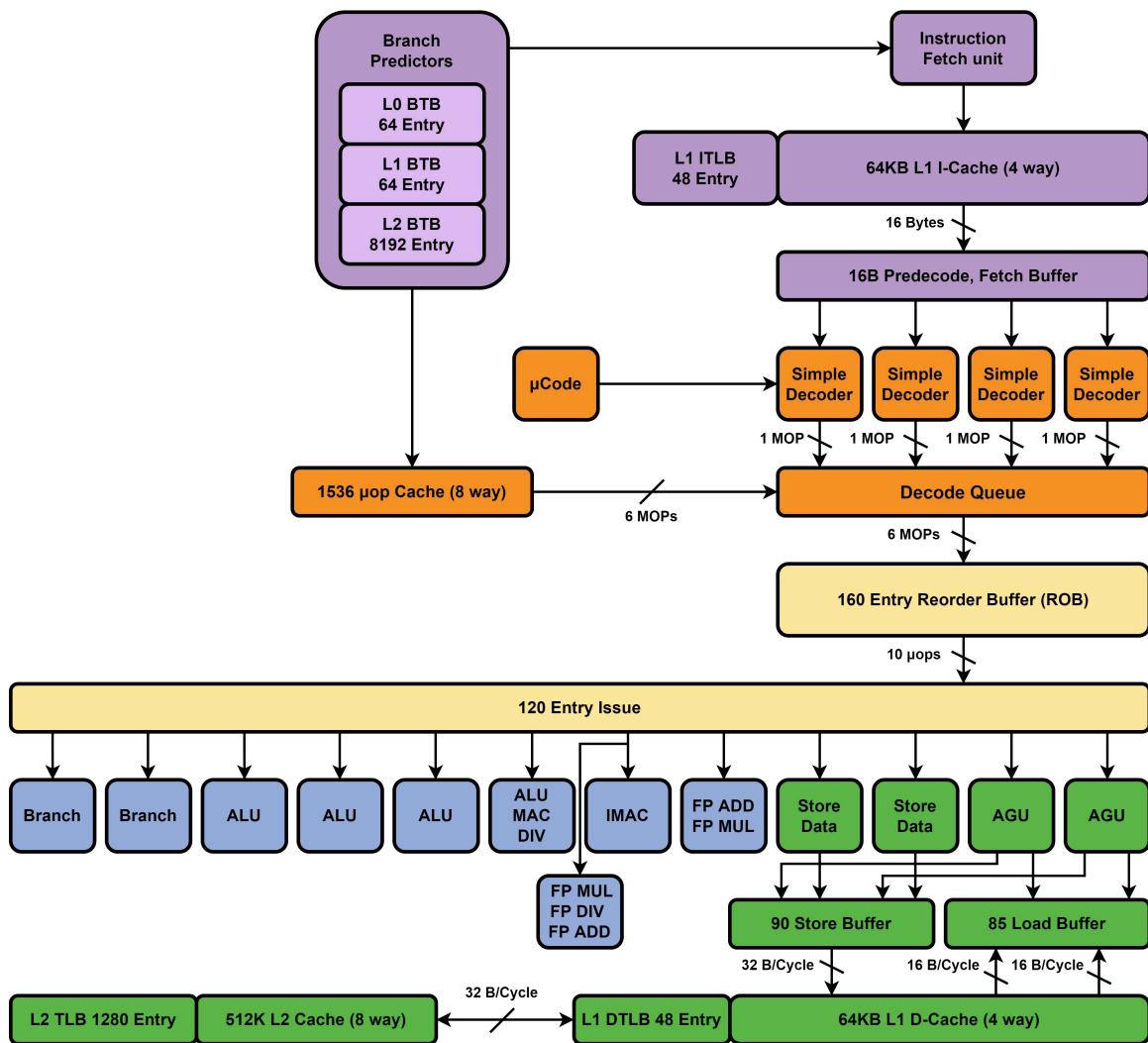
Cortex-A77: Microarchitecture overview



Obrázek 25: Architektura ARM A77

A77

Microarchitecture Block Diagram



Obrázek 26: Architektura ARM A77

CISC

Otázky:

19) Popište a nakreslete jakéhokoli nástupce Intel Pentium Pro, se kterým jsme se seznámili.

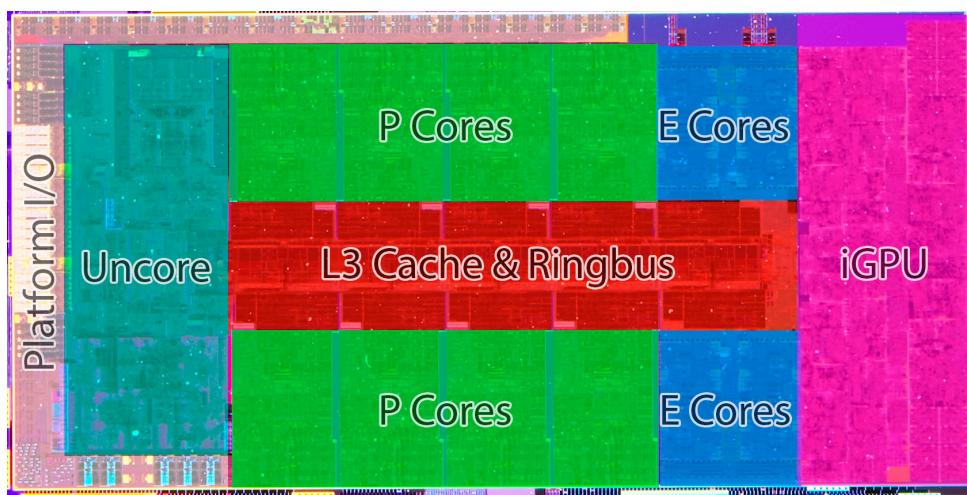
19. Popište a nakreslete jakéhokoli nástupce Intel Pentium Pro, se kterým jsme se seznámili.

Intel Core i9-12900K (2021)

- 12-tá generace - codenáme „Alder Lake“
 - hybridní arch. „Alder Lake“ - na jednom čipu dvě rozdílné arch. jader
 - nastavila nový standard pro statní výrobce procesorů
- 10 nm - Enhanced Super-Fin (*Intel 7*)
- 16 jader:
 - 8 P-cores - arch. „Golden Cove“, vhodné pro hry, videa, grafic. editory
 - 8 E-cores - arch. „Gracemont“, daemon aplikace, méně zatěžující úlohy
- taktovací frekvence - 3,2 GHz (base-mode) až 5,2 GHz (turbo-mode)
- cache:
 - L1: 80 KB per P-core (32KB I-cache + 48KB D-cache), blízko jednot. jader
 - L2: 1,25 MB per P-core, 2MB per E-core, mezi CPU a hlavní pamětí
 - L3: 30 MB společná sdílená paměť všech jader
- podpora DDR4 (*Double Data Rate*) a DDR5 čipů SDRAM
 - vysoká propustnost - rychlý přenos dat mezi hlavní pamětí a procesorem
- podpora PCIe 4.0 (*Peripheral Component Interconnect Express*) a PCIe 5.0
 - komunikace CPU s I/O zařízeními (GPU, SSD, ...)
- integrovaná grafika na čipu *Intel UHD Graphics 770*
- „out-of-order“ vykonávání instrukcí (512 položek na P-core, 256 položek na E-core)
- „Ringbus“ - název bus fabric od Intel, sběrnicová spojnice
- nový „Thread Director“ - nutný kvůli hybridnímu designu, pro rovnoměrné rozdělení zátěže úlohy procesorům

[\[Intel Core i9-12900K schémata\]](#)

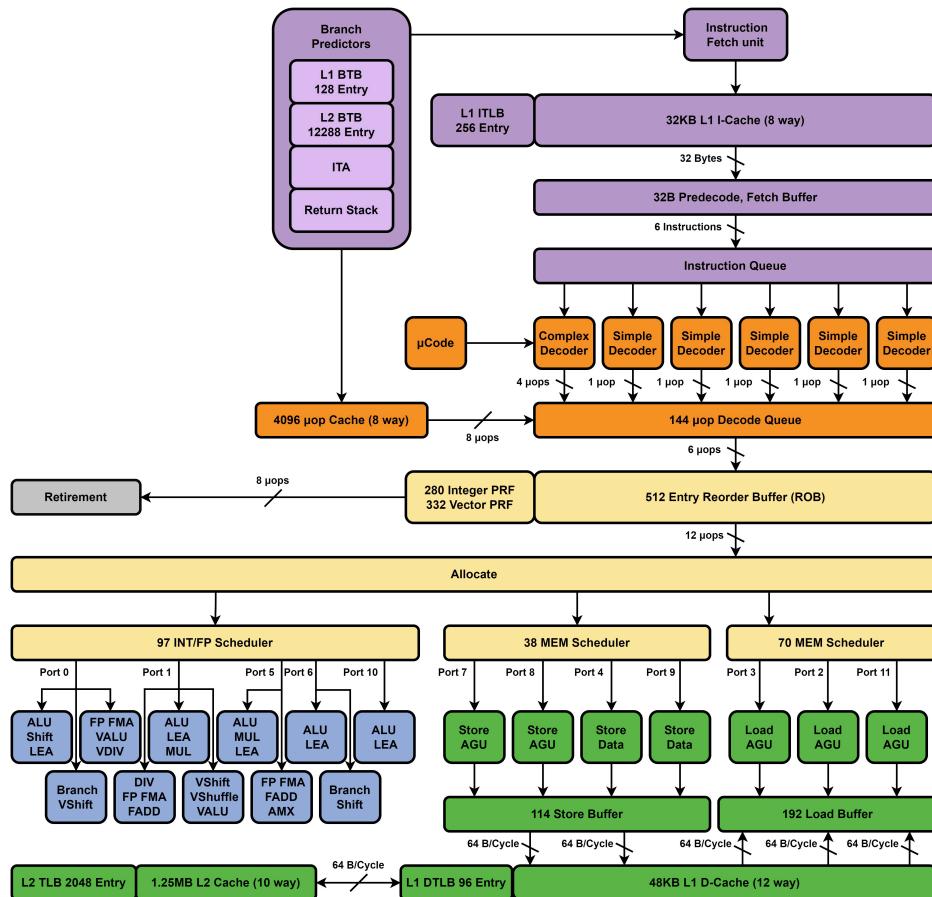
[\[Schéma jádra arch. Golden Cove\]](#)



Obrázek 27: Vnitřní uspořádání čipu Intel Core i9-12900K

Golden Cove

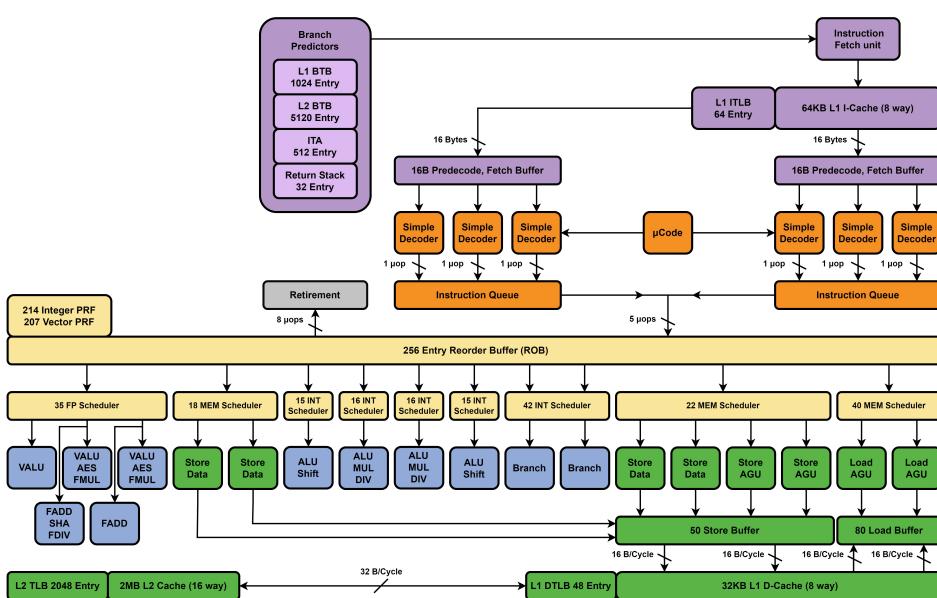
Microarchitecture Block Diagram



Obrázek 28: Architektura jádra Golden Cove - High Power

Gracemont

Microarchitecture Block Diagram



Obrázek 29: Architektura jádra Gracemont - Low Power

Golden Cove

- Branch Predictor
 - ▶ BTB ~12000 entries
- L1 I-cache 32KB
- L1 D-cache 48KB
- L2 cache 1.25MB
- 6 instructions / cycle
- 512 entry ROB (*Re-Order Buffer = Out-Of-Order Window*)
- 10 Execution Units:
 - ▶ 5x ALU
 - ▶ 3x FPU
 - ▶ 2x Branch
- supports these instruction sets:
 - ▶ FMA - extension of SSE (*Streaming SIMD Extension*)
 - ▶ AMX (*Advanced Matrix Extension*)
 - ▶ AVX-512 (*Advanced Vector Extension*)
 - ▶ VNNI (*Virtual Neural Network Instructions*)
- 2x Store Data, 2x Store AGU
- 3x Load AGU

Gracemont

- BTB - 5000 entries
- I-cache 64KB
- 6 decoded instructions/cycle
 - ▶ decodes opcode to μOps
- out-of-order window - 256 entries - dispatch to the execution units (*EU*)
- 17 Execution Units
 - ▶ integer EU
 - ▶ SIMD ALU
 - ▶ FPU
 - ▶ VALU (*Vector ALU*)
- newly supports these instruction sets:
 - ▶ VNNI (*Virtual Neural Network Instructions*)
 - ▶ AVX (*Advanced Vector Extension*)

Paměti

Otázky:

- 20) Rozdelení polovodičových pamětí a jejich popis (klíčová slova a zkratky nestačí).
- 21) Jak funguje DRAM, nakresli. Napiš stručně historii.
- 22) Hierarchie paměti, popsat a zakreslit.

20. Rozdelení polovodičových pamětí a jejich popis (klíčová slova a zkratky nestačí).

- dělení podle typu:
 - ▶ přístupu do paměti:
 - RAM (*Random Access Memory*) - náhodný přístup, kamkoli odkudkoli
 - SAM (*Serial Access Memory*) - přístup sériově, postupně (byte after byte)
 - speciální:
 - fronta - lze nahlížet na *front* a *back* ale ne do těla
 - zásobník - lze nahlížet jen na *top* zásobníku ale ne do těla
 - asociativní
 - ▶ zápis/čtení:
 - RWM (*Read Write Memory*) - lze číst i zapisovat
 - ROM (*Read Only Memory*) - lze jenom číst
 - kombinované:
 - NVRAM (*Non-Volatile RAM*) - EEPROM (*Electrically Erasable PROM*) a Flash paměti
 - WOM (*Write Only Memory*) - lze jen zapisovat (nepraktické, na Linuxu /dev/null)
 - WORM (*Write-Once Read-Many times Memory*) - např. CD-ROM (vypálí se jednou, nejsou přepsatelné, číst lze opakováně)
 - ▶ elementární buňky:
 - DRAM (*Dynamic RAM*) - náboj v kondenzátoru drží bit dat
 - postupně se vybíjí - musí se *refreshnout*, jinak se data ztratí
 - proto název dynamický
 - SRAM (*Static RAM*) - stav klopného obvodu drží bit dat
 - dokud je zapojený proud, udržuje data
 - netřeba žádný *refresh* - proto název statický
 - PROM - přepálené pojistky (*fuse blowing*)
 - přepleně a nepřepálené pojistky reprezentují log. 0 a log. 1 - jeden bit dat
 - EPROM (*Erasable Programmable ROM*), EEPROM (*Electrically EPROM*), Flash Memory
 - tranzistor s plovoucím hradlem (floating-gate transistor)
 - ▶ uchování dat po odpojení napájení:
 - *Volatile* - neuchovají data, musí se nepřetržitě napájet, bývají rychlejší
 - hlavní operační paměti DRAM („ramka“) a SRAM („cache“ paměti)
 - *Non-Volatile* - uchovají data i po odpojení, nezávislé na napájení, bývají pomalejší
 - xxROM paměti - pro bootloadery, BIOSy, Firmware atd.
 - SSD (*Solid State Drive*), HDD (*Hard Disk Drive*) - externí paměti, hlavní uložiště

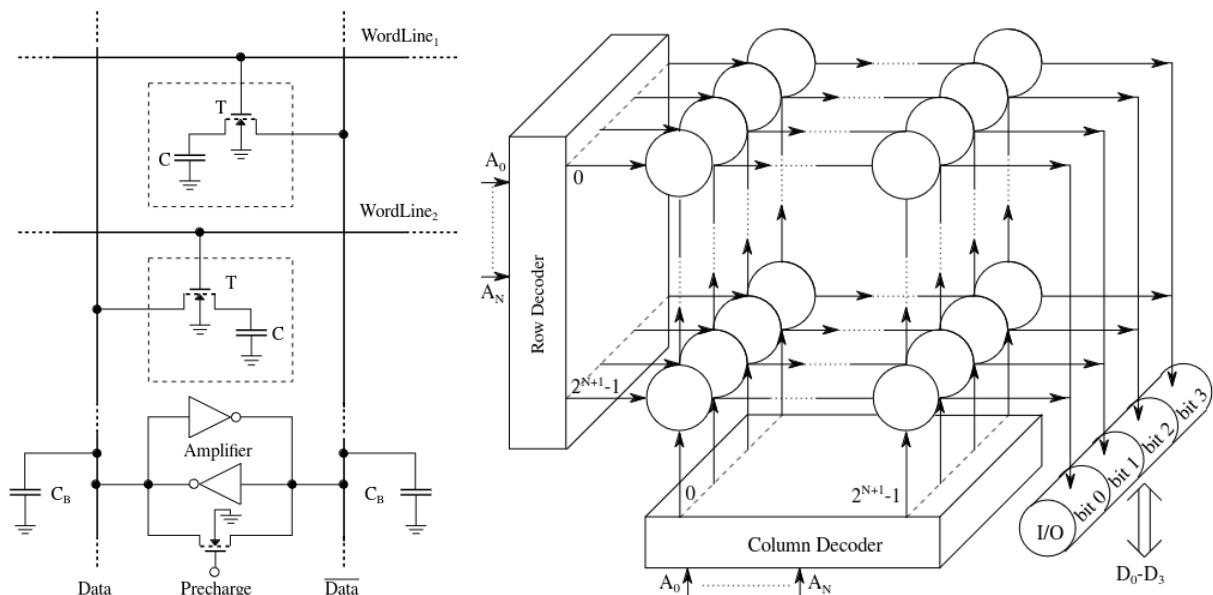
21. Jak funguje DRAM, nakresli. Napiš stručně historii.

Stručná historie

- 50. léta 20. st. - bubenové paměti, feritová jádra
- 70. léta 20. st. - bublinové paměti
- 1969 - polovodičová technologie MOS
- 1970 - Dynamický RAM
- 1971 - Statický RAM

DRAM

- informace uložena ve formě náboje v kondenzátoru
 - nabity = log. 1
 - vybitý = log. 0
- jedna „buňka“ dynamické paměti je sestavena z kondenzátoru C a tranzistoru T (jako el. switch)
- kapacita C je velmi malá (jednotky femtoFahrad) - časem ztrácí napětí (proud teče do/ven z C)
- nutný častý refresh (občerstvení), aby si uchovala svou informaci (každých ~10 ms)
 - provede se pro celý rádek při každém čtení buňky z tohoto rádku



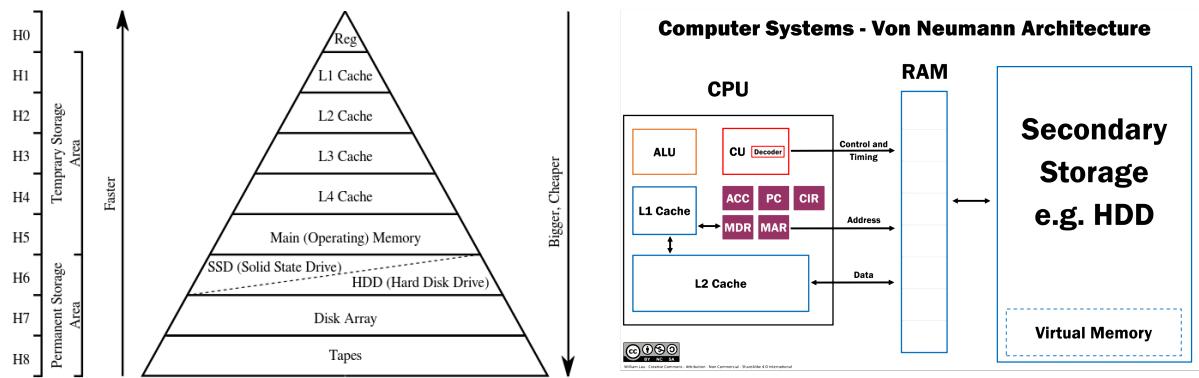
Obrázek 30: Paměťové buňky v DRAM a Organizace paměťových buněk v DRAM

- buňky jsou umístěny ve čtvercové matici (matic je vedle sebe naskládaných více)
- adresování buňky probíha ve dvou krocích:
 - 1) vybere se řádek (ROW) buňky pomocí *Row Decoder*
 - aktivuje vodič (wordline) daného řádku vedoucího z *Row Decoderu*
 - 2) vybere se sloupec (COL) buňky pomocí *Column Decoder*
 - informace uložena v této buňce (1 nebo 0) se pošle do *I/O bufferu* přes vodič (bitline) vedoucí z *Column Decoderu*
- rozdelení adresování na dva dekódery (*Row* a *Column*) znamená, že stačí poloviční počet vodičů pro adresování - např. pro 2^{20} buňek stačí 10 vodičů - $2^{10} \cdot 2^{10} = 2^{20}$
 - sdílení stejných vodičů pro přenos částečné informace se nazývá „multiplexing“
 - musíme ale zavést dva řídící signály:
 - *RAS* (Row Access Strobe) a *CAS* (Column Access Strobe)
 - proto aby se vědělo jakému dekóderu je adresa určena

- DRAM paměť je organizována tak, že se při čtení/zápisu předá více než jeden bit - to určí počet bitů v *I/O bufferu*
- princip čtení z DRAM:
 - *address buffer* příjme adresu, kterou poskytlo CPU
 - adresa je rozdělena na adresu rádku a sloupce
 - pošle se do *address buffer* jedna po druhé - adresní *multiplexing*
 - *multiplexing* je kontrolován *RAS* a *CAS* řídicími signály
 - pokud se pošle adresa rádku aktivuje se předtím *RAS* signál
 - pokud adresa sloupce tak *CAS* signál
 - *address buffer* pošle (na základě *RAS* a *CAS* signálu) částečnou *multiplexovanou* adresu bud' *Row Decoderu* nebo *Column Decoderu*
 - vyšle se *READ* řídicí signál, data adresované paměťové buňky se zesílí el. zesilovači a předá do *I/O bufferu* - výstupní data
- princip zápisu je obdobný čtení z DRAM:
 - probíha klasicky vybrání rádku a sloupce
 - namísto signálu *READ* se ale vyšle *WE (Write Enable)* řídicí signál
 - do *I/O bufferu* se zapíšou data, která se mají uložit
 - data z *I/O bufferu* se zesílí a tím přepíšou adresované buňky

22. Hierarchie paměti, popsat a zakreslit do von Neumann.

- v počítači se používají různé typy paměti
 - důvody jsou ekonomické, technické a praktické
 - kdyby existoval typ paměti, který je levný, nevolatilní, umožňuje náhodný přístup k datům a je rychlý, nepotřebovalo by se vytvářet tolik typů pamětí - taková technologie však zatím neexistuje
 - bere se v potaz kompromis mezi cenou, rychlostí a kapacitou



Obrázek 31: Pyramida hierarchie paměti a druhy paměti ve von Neumann

- paměti uspořádaný podle ceny, rychlosti a kapacity
- každá úroveň paměti tvoří vyrovnávací (*cache*) paměť té pod ní
 - data z OpMem jsou částečně v LCache
 - data z SSD jsou částečně v OpMem atd.
- dělí se na volatilní (*Temporary Storage Area*) a nevolatilní (*Permanent Storage Area*)
- dělí se na čistě polovodičové a mechanické
- podle „H“ úrovně:
 - H0 - registry procesorů, odpovídá rychlosti CPU
 - H1 - Cache L1, SRAM - rychlosť odpovídá vnitř. sběrnicím CPU, kapacita 10s až 100s kB / jádro
 - H2 - Cache L2, SRAM - 100s kB až 1s MB / jádro
 - H3 - Cache L3, SRAM - 10s MB, sdílena mezi jádry, mezi CPU a hlavní pamětí
 - H4 - Cache L4, SRAM - 10s až 100s MB
 - H5 - hlavní operační paměť, DRAM - 1s až 100s GB
 - H6 - SSD (Solid State Drive), HDD (Hard Disk Drive), Flash - 100s GB až 1s TB
 - H7 - disková pole (Disk Array) - „naskládané“ pevné disky, 10s TB až 1s PB
 - H8 - pásové jednotky (Tapes) - velmi pomalé, mrtvě kapacity

Architektura počítačů

Otázky:

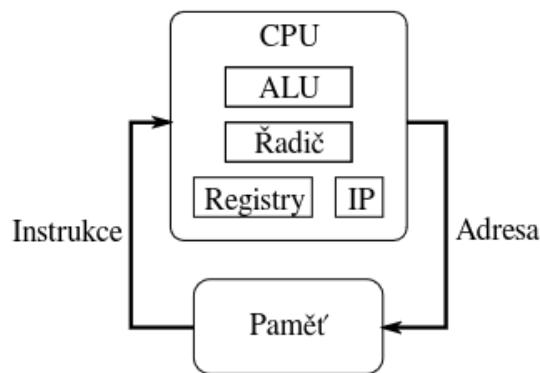
- 23) Popiš základní konstrukci a vlastnosti počítače.
- 24) Jak funguje počítač a jak se vykonávají skokové instrukce.
- 25) Popište a nakreslete architekturu dle von Neumanna. Napište jeho vlastnosti, výhody a nevýhody.
- 26) Popište a nakreslete harvardskou architekturu, popište rozdíly, výhody a nevýhody oproti von Neumannovi. Na obrázku vyznačte části, které mají a nemají společné. Která architektura je podle vás lepší a proč?

23. Popiš základní konstrukci a vlastnosti počítače.

- všechny počítače pochází z von Neumannovy základní koncepce - počítač je řízen obsahem paměti
 - představen v roce 1945 - EDVAC (*Electronic Discrete Variable Automatic Computer*)
- von Neumann stanovil kritéria a principy, které musí počítač splňovat:
 - 1) generický design - struktura počítače je nezávislá na typu řešené úlohy
 - 2) počítač se programuje obsahem paměti
 - 3) strojové instrukce a data jsou uloženy v téže paměti - stejný přístup do paměti
 - rozdílné je to u harvardské koncepce - samostatná paměť pro program (*instrukce*) a data
 - 4) paměť je rozdělena do buňek stejné velikosti - jejich pořadová čísla jako adresy
 - 5) následující krok je závislý na tom předchozím
 - 6) program je sekvence instrukcí, které se sekvenčně vykonají
 - 7) změna pořadí provádění instrukcí se provádí za pomocí skoků ((ne)podmíněný)
 - 8) pro rezprezenaci instrukcí, dat, čísel, znaků se používá dvojková soustava
 - 9) počítač se skládá z řídicí a aritmeticko-logické jednotky (dnes jako celek *CPU* - centrální procesní jednotka), paměti a *I/O* jednotek

24 Jak funguje počítač a jak se vykonávají skokové instrukce.

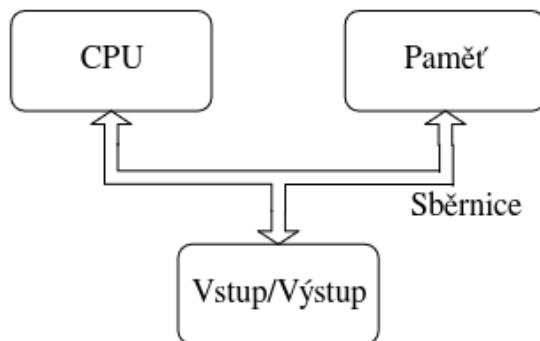
- počítač je programován obsahem paměti
- instrukce se vykonávají sekvenčně
- každy krok závisí a tom předchozím
- procesor (*CPU*) je sekvenční obvod - vstupem jsou strojové instrukce z paměti
 - má svou vlastní rychlou paměť pro ukládání výsledku instrukcí - *registry*
 - instrukční ukazatel (*IP* - *Instruction Pointer*) - někdy jako *PC* (*Program Counter*)
 - ukazuje na instrukci v paměti, která má být vykonána
 - pomocí něj „*fetchne*“ insrukci z paměti a vykoná ji



Obrázek 32: Princip fungování počítače

- následně se inkrementuje o délku právě provedené instrukce
- pokud instrukce potřebuje data z paměti, vyžádá si je *CPU* stejným způsobem jako instrukce
- výsledek se uloží do paměti přes sběrnici (nebo ještě zůstává v registrech)
- skokové instrukce jsou dvojího typu - podmíněné a nepodmíněné
 - když *CPU* zpracovává nepodmíněný skok, nastaví svůj *IP* na místo kam se skáče
 - následné instrukce se *fetchují* od této adresy
 - u podmíněného skoku, *CPU* vyhodnotí, jestli se bude skákat
 - na základě stavu *flags* registru, který uchovává aktuální stav procesoru
 - až potom se nastaví *IP* na cílovou (*target*) adresu (skočilo se) nebo
 - se pouze inkrementuje o délku právě vykonalé instrukce (neskočilo se)

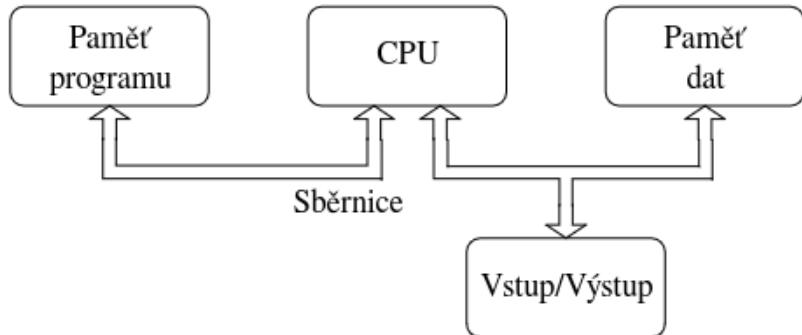
25. Popište a nakreslete architekturu dle von Neumann. Napište jeho vlastnosti, výhody a nevýhody.



Obrázek 33: Počítač dle *von Neumann*

- pro vlastnosti viz otázku 21
- CPU (*Central Processing Unit*) - sdružené řídicí a výpočetní jednotky
- paměť - společná pro program i data
- vstup/výstup (*Input/Output*) - pro externí zařízení
- sběrnice - sdružené datové a řídicí signály, propojuje *CPU*, paměť a *I/O*
- výhody:
 - rozdělení paměti pro program a data si rozhodne sám programátor
 - řídicí jednotka přistupuje do paměti pro data i instrukce jednotným způsobem
 - jedná sběrnice znamená jednodušší design a výrobu
- nevýhody:
 - společné uložení dat a programu může vést při chybě k přepsání valstního programu
 - jediná sběrnice je *bottleneck* (úzké místo) - nižší propustnost dat

26. Popište a nakreslete harvardskou architekturu, popište rozdíly, výhody a nevýhody oproti von Neumann. Na obrázku vyznačte části, které mají a nemají společné. Která architektura je podle vás lepší a proč?



Obrázek 34: Počítač dle harvardské architektury

- vznikla pár let potom, co von Neumann představil svou koncepci
- vymyslel jej odborný tým z Harvardsé univerzity
- od von Neumann se moc neliší - snažila se vyřešit její nedostatky
 - pouze oddělení paměti na dvě samostatné - pro program a pro data
- pro vlastnosti viz 21. otázku - jsou až na rozdělení paměti stejné
- výhody:
 - oddělení paměti pro data a program
 - program nemůže přepsat sám sebe
 - paměti mohou být vytvořeny odlišnými technologiemi
 - jiná velikost nejmenší adresovací jednotky
 - např. ROM pro program a RWM pro data
 - dvě sběrnice zvyšují propustnost - současně přistupování pro data i instrukce
- nevýhody:
 - dvě sběrnice - vyšší nároky při vývoji řídicí jednotky + zvyšuje náklady pro výrobce
 - nevyužitelná část paměti nelze využít - program nemůže být v paměti pro data a naopak

Komunikace

Otázky:

- 27) Komunikace se semafory a bez semaforů (indikátorů). Nakresli aspoň jedním směrem.
- 28) Přenos dat použitím V/V brány s bufferem. Nakreslit obrázek komunikace jedním směrem a jak se liší komunikace druhým směrem. V jakých periferiích se používá.
- 29) Popiš DMA blok a nakresli schéma DMA řadič v architektuře dle von Neumanna.

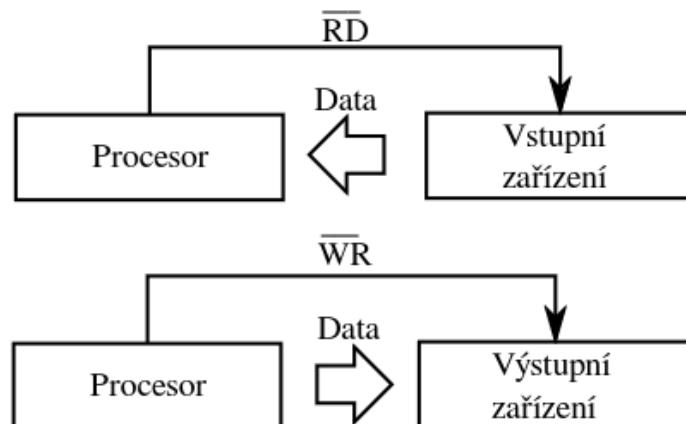
Technika I/O brán

- *I/O gate* (vstupně-výstupní brány)
 - obvod zprostředkovávající předávání dat mezi sběrnicí počítače a perifériemi
 - základem je *latch register* (záhytný registr) s *tří-stavovým výstupem* (three-state, tří-stavový budič sběrnice)
 - 1) *Inactive* stav - stav vysoké impedance, „Do not disturb“
 - 2) *Input* stav - periférie data přijímá
 - 3) *Output* stav - periférie vysíla data
 - má ho každý tzv. budič sběrnice (každé zařízení/periférie, které je napojené na sběrnici)
 - jeho stav dává vědět ostatním zařízením/periferiím, jestli na sběrnici data zapisuje (*Output*), čte (*Input*) nebo s ní momentálně nekomunikuje (*Inactive*)
 - možnost použít brány s pamětí (*buffer*)
 - ten je potřebný při obostranném (úplného) korespondenčním režimu
 - v následujících otázkách jsou popsány některé druhy technik I/O brán

27. Komunikace se semafory a bez semaforů (indikátorů).

Nakresli aspoň jedním směrem.

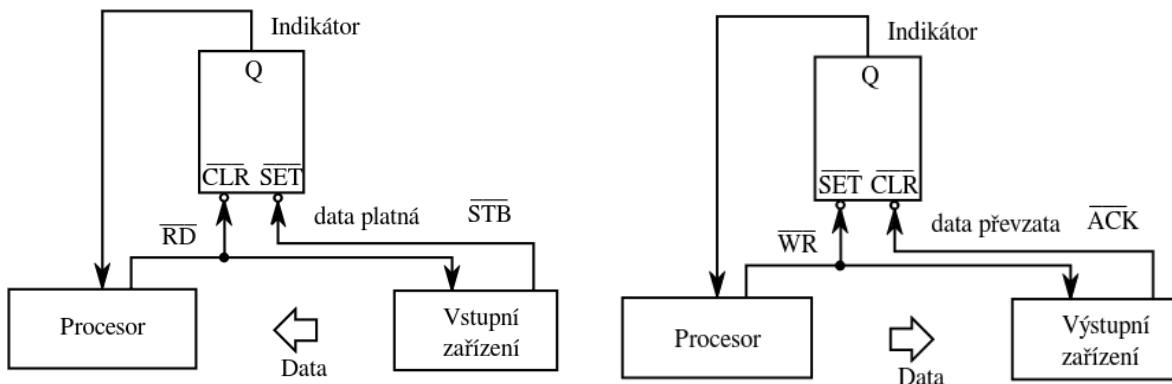
Technika nepodmíněného vstupu a výstupu (*bez semaforu/indikátoru*)



Obrázek 35: Technika nepodméněného vstupu a výstupu *bufferu*

- vstup (input) - procesor vyšle signál *RD* (read)
 - přikáže tím vstupnímu zařízení předat data do procesoru
 - nijak se nekontroluje jestli je periférie připravená (očekává se, že je vždy připravená)
- výstup (output) - procesor vyšle signál *WR* (write)
 - výstupní zařízení data z procesoru převeze
 - nijak se nekontroluje, jestli data periferní zařízení opravdu převzalo
- tento způsob je velmi jednoduchý
- předpokládá neustálou připravenost periferního zařízení

Technika podmíněného vstupu a výstupu (se semaphorem/indikátorem)



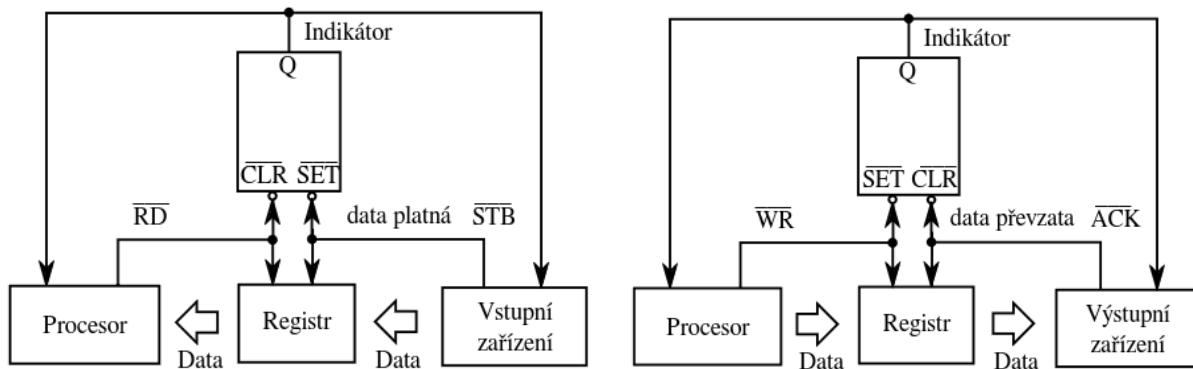
Obrázek 36: Technika podmíněného vstupu a výstupu bez *bufferu*

- vstup (input)
 - ▶ periferní zařízení připraví data na vyslání - pokud jsou data platná vyšle signál *STB (Strobe)*
 - ▶ ten nastaví indikátor (*flag*) Q na 1 - $Q = 1$
 - ▶ pokud je $Q = 1$ jsou data připravena pro předání do procesoru
 - ▶ procesor si průběžně stav Q kontroluje:
 - vidí že $Q = 1 \rightarrow$ data se přečtem procesorem vysláním signálu *RD (read)* z procesoru
 - ten zároveň nastaví $Q = 0$
- výstup (output)
 - ▶ procesor vyšle signál *WR (write)* pro zápis dat do výstupního zařízení a data pošle
 - zároveň tím nastaví $Q = 1$
 - ▶ periferní zařízení data z procesoru převeze a vyšle signál *ACK* - data převzata
 - nastaví tím $Q = 0$
 - dá tím procesoru najevo, že data skutečně převzalo
 - procesor může vyslat další data - periférie je připravena přijímat další data
- obrázky popisují jednosměrný korespondenční režim (neúplný) - není zde *buffer* uprostřed komunikace
 - ▶ vysílač dat (ať už procesor nebo periférie) je povinen si data udržovat při celém průběhu komunikace - nemá *buffer* (na obrázku jako registr), kam by je průběžně mohl zapisovat

28. Přenos dat použitím V/V brány s bufferem. Nakreslit obrázek komunikace jedním směrem a jak se liší komunikace druhým směrem. V jakých periferiích se používá.

- funguje na principu *input/output* v technice bez *bufferu* (minulá otázka) ale tentokrát ten *buffer* má
- využívá *buffer* (na obrázku registr) jako vyrovnávací paměť a klopný obvod (*flip-flop*) jako semafor/indikátor
- jde o obousměrný korespondenční režim (úplný) komunikace
 - ▶ možnost vzájemného blokování (*interlock*) - vysílač dat a příjemce dat testují stav indikátoru Q
- vstup (input)
 - ▶ Q informuje procesoru připravenost dat ve vyrovnávací paměti (*bufferu*)
 - ▶ pro periférii informuje Q
 - zda procesor data již přečetl a je možno do *bufferu* zaslat další data
 - nebo data jěště přečtena nebyla a nemůže zaslat další data do *bufferu*
- výstup (output) - význam indikátoru Q je pro procesor a periferii opačný než v *input*

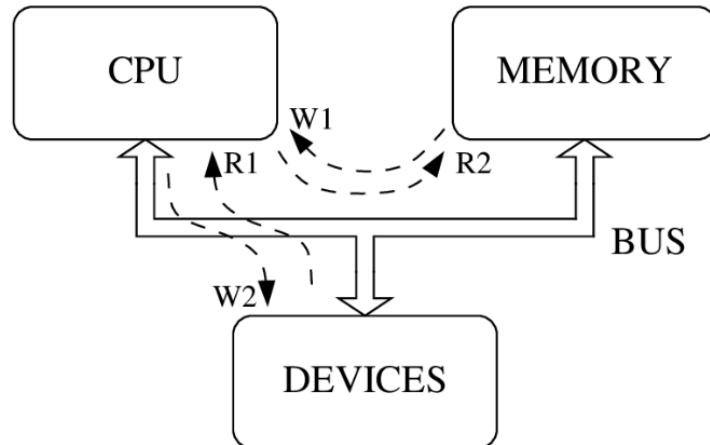
- využití: sériové komunikační porty (*SPI, UART, I2C* apod.), paměťové karty, audio a video zařízení, síťové karty, tiskárny



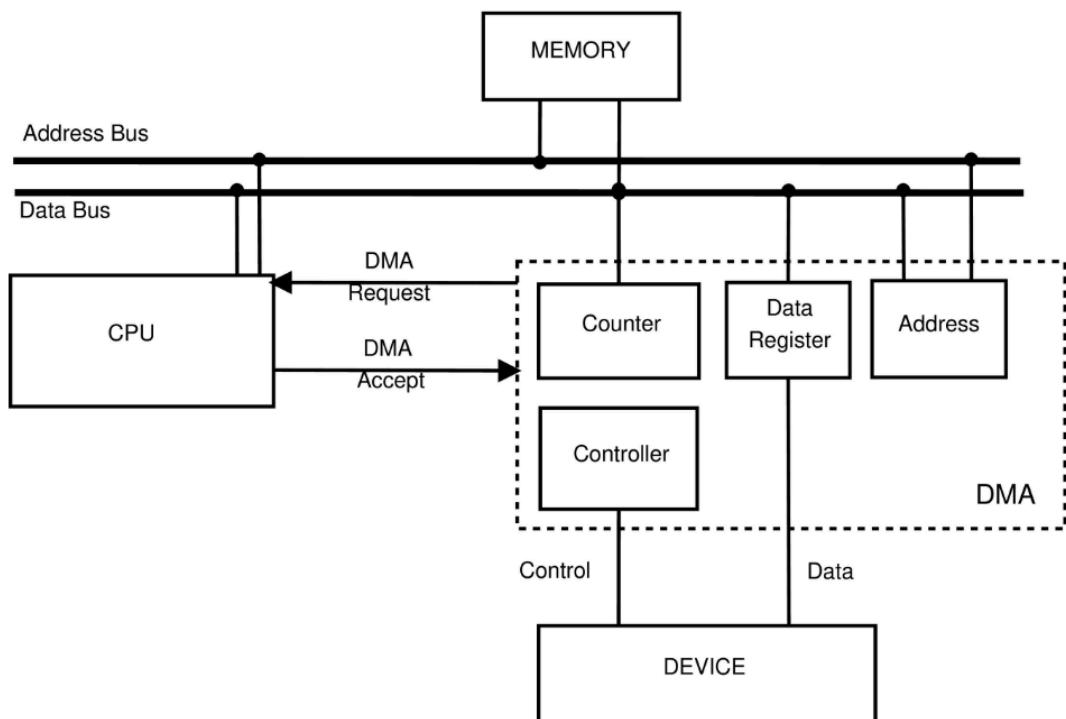
Obrázek 37: Technika korespondenčně obousměrná s *bufferem*

29. Popiš DMA blok a nakresli schéma DMA řadič v architektuře dle von Neumanna.

- DMA (*Direct Memory Access*) blok / kontrolér / řadič
 - umožňuje periferním zařízením vstupovat do hlavní paměti přímo
 - přímý přesun dat mezi hlavní pamětí a periférií s minimální účasti procesoru
 - bez DMA bloku musí každý byte dat z periférie projít procesorem a až potom procesor může přistoupit do paměti
 - procesor se samotného přesunu dat neučastní - pouze nastaví/naprogramuje *DMA blok*
 - sběrnice musí být při přesunu uvolněna - může být maximálně jeden *budič* sběrnice
 - procesor přepne všechny budíče sběrnice do stavu vysoké impedance
 - DMA zajistí přesun - generuje sám adresy v paměti, kam se bude zapisovat/číst
 - v DMA bloku jsou tři registry sloužící pro styk se sběrnici:
 - data register* - obsahuje slovo, které má být přesunuto z periferie do paměti nebo naopak
 - address register* - pro uchování adresy v hlavní paměti, na kterou bude slovo zapsáno nebo čteno
 - counter* - počet slov, které mají být ještě přesunuty
 - operace přístupu do paměti (z pohledu zápisu do paměti):
 - procesor naprogramuje blok DMA - nastaví se registry *counter* a *address*
 - blok DMA spustí periferní zařízení a čeká až bude připraveno
 - periférie oznámí DMA bloku, že je připravena - DMA blok vyšle *DMA Request*
 - procesor dokončí strojovou instrukci a začne se věnovat *DMA Requestu*
 - přímý přístup do paměti se vykonává synchronně při normální činnosti procesoru
 - synchronně se přepíná přístup ke sběrnici a paměti mezi DMA blokem a procesorem
 - DMA blok pracuje s pamětí ve fázi Φ_1 a procesor ve fázi Φ_2 interního hodin. signálu
 - DMA blok vyšle na adresovou sběrnici obsah svého *address registru* a na datovou sběrnici obsah svého *data registru*
 - počka jeden paměťový cyklus → inkrementuje *address register*, dekrementuje *counter* (počet slov, které mají být ještě přesunuty)
 - testuje zda *counter* = 0
 - pokud ano - ukončí DMA komunikaci a vrátí kontrolu nad sběrnici procesoru
 - pokud ne - proces se opakuje od 5. kroku
 - čtení funguje obdobně, jen opačným směrem



Obrázek 38: Přenos dat *bez DMA bloku* (*ve von Neumann*) vyžadující neustálý zásah procesoru



Obrázek 39: Přenos dat *s DMA blokem* (*ve von Neumann*) nevyžaduje zásah procesoru (minimální)

Assembly x86

Otázky:

- 30) Jak adresujeme na úrovni strojového kódu - příklad.
- 31) Podmíněné a nepodmíněné skoky v strojovém kódu.
- 32) Jak řešíme v Assembly x86 podmínky - co jím musí předcházet. Jaký je vztah mezi tím, co je předchází, a tou podminkou. Kde a proč záleží na datových typech.

30. Jak adresujeme na úrovni strojového kódu - příklad.

- adresování dělíme na *přímé* a *nepřímé*
- přímé adresování:
 - uvádíme konkrétní adresu v paměti

```
int a = 12;                                // v jazyce C
int* a_ptr = &a;
printf("%p\n", a_ptr);                     // kontrátní adresa v paměti
$> 0x7ffc58e5f904                         ; v jazyce Assembly
```

- např. adresy globálních proměnných
- není běžné aby programátor adresoval data v paměti přímo

- nepřímé adresování:

- adresujeme přes registry

```
mov rdx, 0x7ffc58e5f904          ; v registru se uloží adresa v paměti
mov eax, dword [rdx]             ; do registru eax se uloží hodnota 12
```

- v 64-bit systémech s x86 instrukční sadou adresujeme pomocí:
 - registrů (bázová adresa a index v poli, max. 2 registry)
 - měřítka (počet bytů - char = 1, short = 2, int = 4, long = 8, a více),
 - konstanty - většinou není potřebná

```
[bázový_registr + index_registr * měřítka + konstanta]
mov eax, dword [rdx + rcx * 4 + 0]

– rdx registr je zde bázovým registrem
– rcx registr představuje indexový registr
– číslo 4 je měřítkem - např. adresujeme pole int array[N] = {0}, protože sizeof(int) = 4
– vevnitř [...] můžeme použít jenom operace sčítání (+) a násobení (*)
```

31. Podmíněné a nepodmíněné skoky v strojovém kódu.

- skokové instrukce:

```

; tyto jsou nepodmíněné
jmp <target> ; nepodmíněný skok na <target>
call <target> ; zavolání podprogramu, skočí se <target>
ret ; skočí se zpět do nadprogramu
      ; jeho adresa je na stacku

; další jsou už podmíněné
; tyto testují registr rcx a jsou využívány pro cykly
loop <target> ; if(--ecx) goto <target>
loope <target> ; if(--ecx && zf) goto <target>
loopz <target> ; stejně jako loope
loopne <target> ; if(--ecx && !zf) goto <target>
loopnz <target> ; stejně jako loopne
jcxz <target> ; if(!rcx) goto <target>

; tyto rozhodují o vykonání skoku na základě jednoho bitu ve flags registru
; tím rozhodnou o vykonání skoku
; (testují jednotlivé bity ve stavovém (flags) registru procesoru)
jz / je <target> ; if equal = 1
jnz / jne <target> ; if equal = 0
js <target> ; if sign = 1
jns <target> ; if sign = 0
jc <target> ; if carry = 1
jnc <target> ; if carry = 0
jo <target> ; if overflow = 1
jno <target> ; if overflow = 0

; tyto řeší porovnávání čísel, také hledí na bity ve flags registru
; (testují více flagů (bitů) ve flags registru)
ja / jnbe <target> ; if below = 0 && equal = 0
jb / jnae / jc <target> ; if below = 1 && equal = 0
jae / jnb / jnc <target> ; if below = 0 && equal = 1
jbe / jna <target> ; if below = 1 && equal = 1
jg / jnle <target> ; if less = 0 && equal = 0
jl / jnge <target> ; if less = 1 && equal = 0
jge / jnl <target> ; if less = 0 && equal = 1
jle / jng <target> ; if less = 1 && equal = 1

```

- skoky nepodmíněné, **jmp**, **call**, **ret**, se vykonají vždy - IP skočí na cílovou (*target*) adresu
- skoky podmíněné se vykonají pouze, tehdy když jsou nastaveny správné flagy
 - podmíněným skokovým instrukcím vždy předchází operace, které *setnou* nebo *clearnou* flagy ve *flags* registru, který uchovává aktuální stav procesoru
 - jakékoli aritmetické nebo logické operace **sub**, **add**, **mul**, **and**, **or**, **xor** ...
 - nejjednodušejí instrukcí **cmp** <cokoli>, <cokoli>
 - stejná jako instrukce **sub**, ale neuloží výsledek
 - pouze nastaví příznakové bity v registru **flags**
 - instrukce **test** <cokoliv>, <cokoliv> je stejná jako instrukce **and**, ale neuloží výsledek
- viz další otázka pro podrobnější vysvětlení podmínek v Assembly

32. Jak řešíme v Assembly x86 podmínky - co jím musí předcházet. Jaký je vztah mezi tím, co je předchází, a tou podmínkou. Kde a proč záleží na datových typech.

- viz předchozí otázka - řeší se tam skokové instrukce a komparační instrukce `cmp`, `test`
 - v Assembly řešíme podmínky instrukcemi, které nastaví příznakové bity v registru `flags`, a skokovými instrukcemi
 - ▶ příznakové bity v 8-bit `flags` registru:
 - ZF (*zero flag*) - nastaví se pokud výsledek operace je nula
 - SF (*sign flag*) - nastaví se pokud výsledek operace je záporné číslo
 - OF (*overflow flag*) - nastaví pokud dojde ke znamékovému přetečení (*signed overflow*)
 - CF (*carry flag*) - nastaví se pokud dojde k neznamékovému přetečení (*unsigned overflow*)
 - PF (*parity flag*) - nastaví se pokud výsledek má sudý počet jedniček (např. 0110 1001)
 - AF (*auxiliary carry flag*) - nastaví se pokud dojde ke `carry out` ve spodním nibble
 - DF (*direction flag*) - řídí směr `string` operací
 - IF (*interrupt flag*) - řídí povolení a zakázání přerušení
 - ▶ instrukce, které nastaví příznakové bity:
 - logické operace - `and <d>, <s>` `or <d>, <s>` `xor <d>, <s>` `not <r>`
 - aritmetické operace - `add <d>, <s>` `sub <d>, <s>` `inc <r>` `dec <r>` `mul <r>` `div <r>`
 - komparační a testovací operace - `cmp <d>, <s>` `test <d>, <s>`
 - řídící operace - `clc` ; clear carry, `stc` ; set carry - obdobně pro všechny příznak. bity
 - řetězcové operace - `movs`, `cmps`, `lod`, `stos`
 - ▶ skokové instrukce: (viz minulá otázka bruv)
 - na datových typech záleží v Assembly při všem - programátor je zodpovědný za vše a to i za správné určení datových typů
 - ▶ Assembly neřeší co nějaká sekvence bitů v paměti znamená/reprezentuje - `int`, `float`, `long`, `struct {int, float}, char`
 - ▶ je mu to jedno - jsou to jenom bity v paměti bez žádného významu
 - ▶ programátor jím dává význam - sami o sobě jen jsou jen „jedničky“ a „nuly“
 - syntakticky hledí Assembly na datové typy zadané programátorem u instrukcí s `d` a `s` parametry: `INSTRUCTION_NAME <d>, <s>` ; `d` = destination, `s` = source
 - ▶ myšleno jako destination a source parametry - jsou jimi např.:
 - přesuny - `mov dest src` ; velikost dest a src musí být stejná
 - násobení a dělení - `div reg` ; registry musí být stejně velikosti (rax / rcx)
 - komparace - `cmp dest src` ; musí být stejně velikosti
 - datové typy specifikují pouze velikost vyhrazené paměti
 - ▶ žádné info o tom zda je to `int`, `char`, `char[]`, `float`
 - `db / byte` ; 1 byte, 8 bitů
 - `dw / word` ; 2 byty, 16 bitů
 - `dd / dword` ; 4 byty, 32 bitů
 - `dq / qword` ; 8 bytů, 64 bitů
 - ▶ zkrácené se používají při deklaraci proměnných v `section .data`
 - ▶ plné názvy se používají pro určení velikosti operandu v `section .text`
- ```

section .data
some_var: db "Hello, World!", 0x00 ; some_var = "Hello, World!" + "\0"

section .text
 mov eax, dword [rdx + rcx * 4] ; do eax se vloží dword toho co je na adrese

```

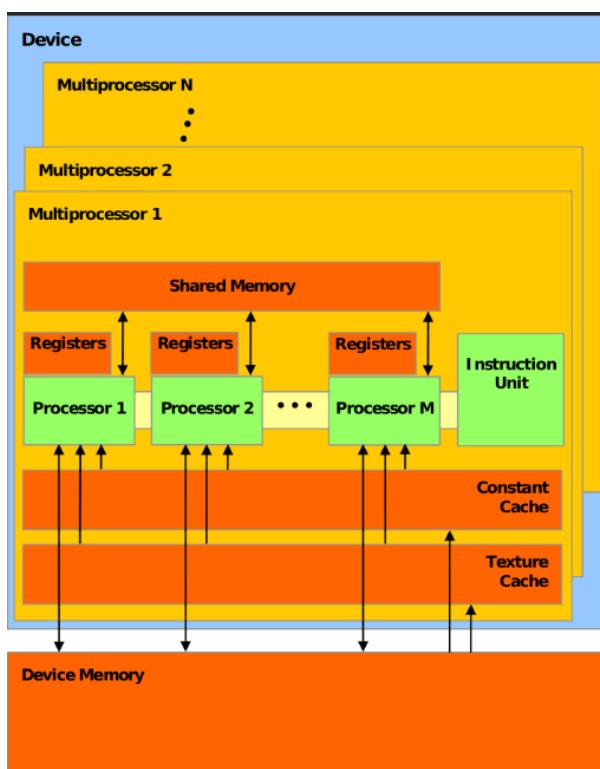
# CUDA

## Otázky:

- 33) Princip programování CUDA - jak, kde se přesouvají data při výpočtu.
- 34) Jaké je C/C++ rozšíření CUDA a jak to využije programátor. Jak si programátor organizuje výpočet. K čemu je mřížka. Nákres dobrovolný.
- 35) Čemu by se měl programátor vyhnout a jak CUDA funguje.

## Grafické karty (Nvidia) a CUDA

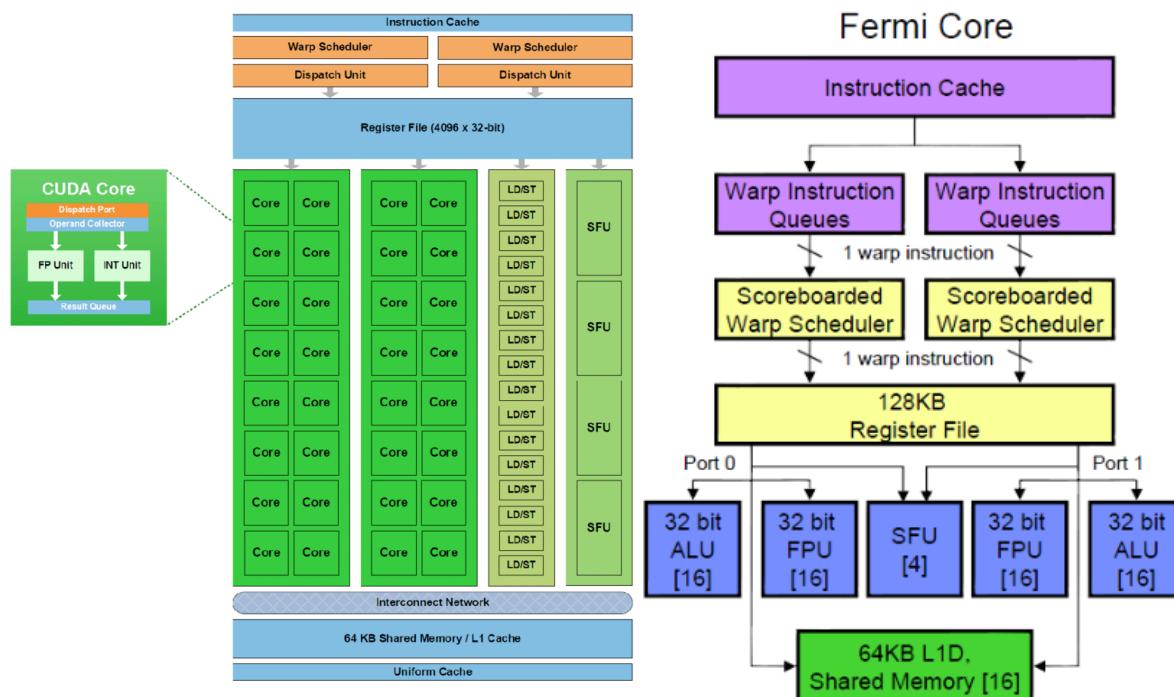
- CUDA je málé rozšíření jazyka C/C++ (také Python, Fortran)
  - programové rozhraní umožňující využití GPU vypočetní síly
- umožňuje využití výpočetní síly grafické karty:
  - masivní paralelismus - stovky tisíc vláken vykonávají stejný kód
  - všechna vlákna musí být na sobě nezávislé
    - GPU nezaručuje synchronní pořadí vykonání instrukcí
  - grafické karty jsou navrženy tak, aby maximalizovali výpočetní výkon
    - nejlépe by se mělo cyklům a podmíněným skokům zcela vyhnout
  - nepodporuje „*out-of-order*“ vykonávání instrukcí jako CPU-čka od Intelu
  - je optimalizováno pro sekvenční přístup do globální (hlavní / „*Device*“) paměti grafické karty
    - přenosová rychlosť až stovky GB/sekunda
  - většina tranzistorů na kartě je výpočetních (je zde minimum řídicích a pomocných obvodů)



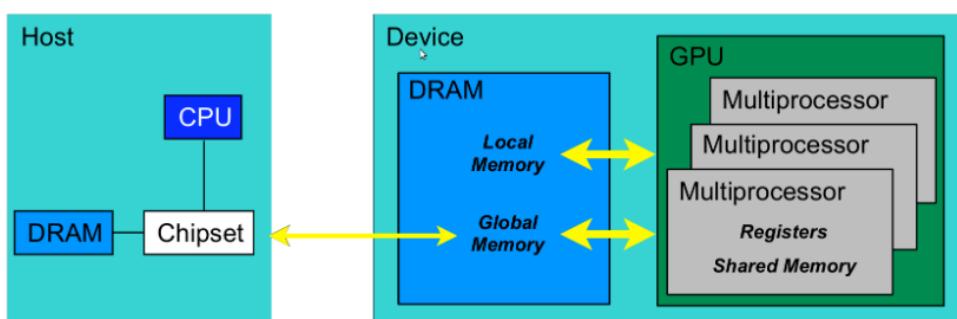
Obrázek 40: Pohled na GPU architekturu z „high level“

- na obrázku:
  - grafické karty jsou rozdělené do *multiprocesorů*
    - je to „pole“ procesorů, které pracují spolu - vykonávají stejné instrukce na jiných datech (*SIMD*)
    - každý *multiprocesor* může vykonávat jiný kód (*MIMD*)

- každá grafická karta obsahuje různý počet *multiprocesorů*
- *Device Memory* je paměť sdílená všemi *multiprocesory*
  - skládá se z *Global Memory*, *Texture Memory* a *Constant Memory*
- každý multiprocesor obsahuje svůj *Shared Memory*, jednotlivé procesory a vyrovnávací paměti
  - *Shared Memory* je sdílená paměť mezi všemi procesory v jednom multiprocesoru
  - každý procesor má navíc své vlastní registry
  - každý multiprocesor má vyrovnávací paměti (*Cache Memory*) pro rychlejší přístup k datům v globální *Device* paměti
    - *Constant Cache* je pro rychlý přístup k datům v *Constant Memory*
    - *Texture Cache* je pro rychlý přístup k datům v *Texture Memory*

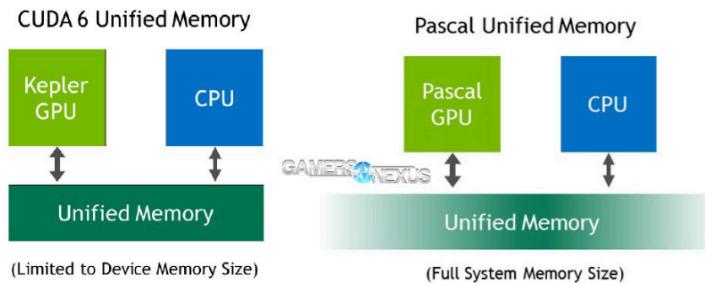


Obrázek 41: Fermi architektura schéma v GPU a jeho zjednodušené schéma



Obrázek 42: Organizace paměti - grafická karta („*Device*“) v počítači

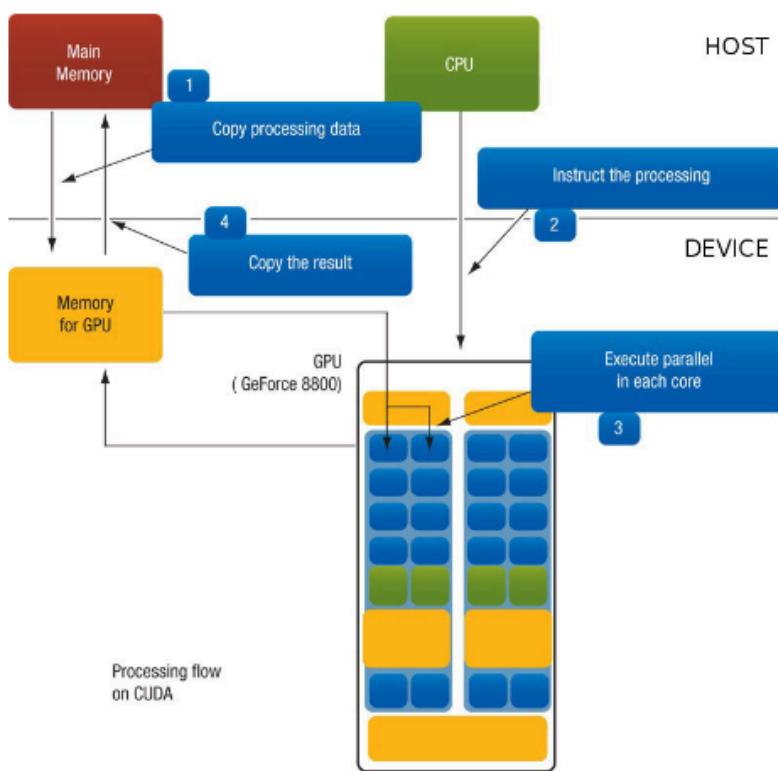
- „*Device*“ je grafické karty a skládá se z:
  - DRAM paměti (*Globální paměť grafické karty* nebo „*Device Memory*“)
  - GPU multiprocesorů
- „*Host*“ je jakýkoli počítač s nainstalovanou grafickou kartou
- paměti od „*Host*“ a „*Device*“ jsou propojeny pouze sběrnicí - nejsou sdílené



Obrázek 43: Unifikované paměť u moderních GPU architektur

- některé počítače mají společnou sdílenou paměť mezi *CPU* a *GPU*

### 33. Princip programování CUDA - jak, kde, kdy se přesouvají data při výpočtu.



Obrázek 44: Výpočetní proces na grafické kartě

- výpočetní proces s CUDA se skládá z několika kroků:
  - 1) data se z hlavní operační paměti počítače („Host Memory“) přesunou do paměti grafické karty („Device Memory“)
    - ▶ nakopírují se data z „ramky“ do globální paměti grafické karty
    - ▶ data jsou přenášena přes sběrnici počítače - dnes obvykle přes PCIe (Peripheral Component Interconnect Express - standard pro vysokorychlostní seriové sběrnice)
  - 2) procesor CPU dá pokyn ke zpracování
    - ▶ naorganizuje vlákna (konfiguruje mřížku) a poskytne kód (instrukce), který se má spustit
  - 3) vlákna (procesory) v GPU multiproceserech vykonají poskytnutý kód (instrukce)
    - ▶ paralelně se spustí všechna nakonfigurovaná vlákna, která vykonají stejný kód (instrukce) na jiných datech - SIMD (Single Instruction Multiple Data)
  - 4) přesunutí výsledných zpracovaných dat z globální paměti grafické karty („Device Memory“) zpět do hlavní paměti („Host Memory“)

## 34. Jaké je C/C++ rozšíření CUDA a jak to využije programátor. Jak si programátor organizuje výpočet. K čemu je mřížka.

### Nákres dobrovolný.

- CUDA přináší rozšíření jazyka C/C++, které umožňuje využít výpočetní výkon grafických karet vyráběných společností Nvidia
- jsou jimi *kernely*, modifikátory funkcí a proměnných, datové typy a struktury, předdefinované globální proměnné, API funkce s předponou *cuda*
  - ▶ *kernel* je funkce napsaná pro spuštění v jednotlivých jádrech grafické karty:
    - klasicky napíšeme funkci v C++ a označíme ji modifikátorem *\_global\_* (před/nad signaturu funkce)
    - voláme je z *CPU*-čka („*Host*“) následující syntakcí:

```
functionName<<<gridDimensions, blockDim>>>(parameters...)
```

- do trojtých „*chevrons*“ <<< >>> zadáváme konfiguraci mřížky a to:
  - velikost (dimenze) mřížky - jejím datovým typ je *dim3*
  - velikost (dimenze) bloků v mřížce - jejím datovým typ je také *dim3*
  - mřížka je od toho aby se *GPU*-čku při spuštění *kernelu* nařídilo, jaký májí data „*tvar*“
    - ▶ *RGBA* obrázek o velikosti 1200x800 nakonfigurujeme do mřížky 1200x800x4, kde čísla jsou odpovídají počtem vláken na danou dimenzi
      - při spuštění se ještě musí hledět na to, jak tyto vlákna přerozdělíme do bloků stejných rozměrů, tak aby se vešel každý pixel obrázku do finální mřížky
      - ▶ mřížka může dále představovat tvar tenzorů, matic, vektorů apod.

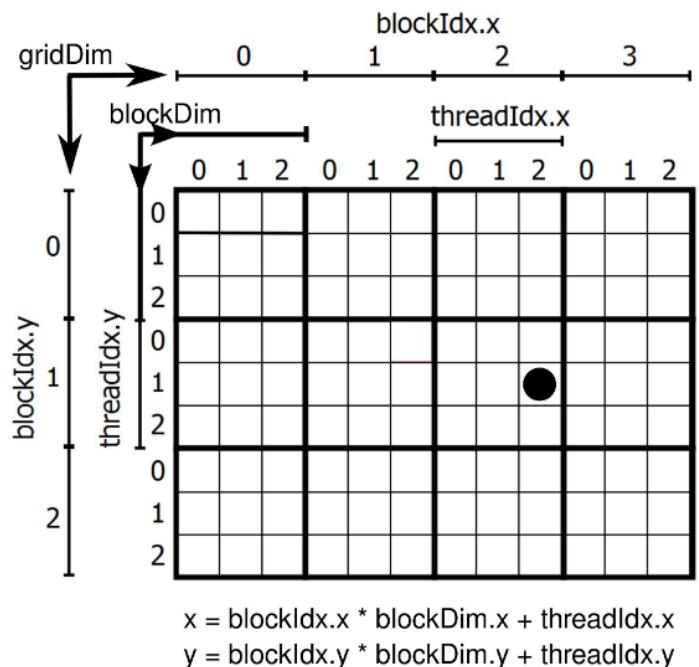
- ▶ modifikátory funkcí - píšeme před signaturou funkcí:
  - *\_host\_* - značí, že tuto funkci může spustit/zavolat pouze *CPU*-čko („*Host*“)
  - *\_device\_* - značí, že tuto funkci může spustit/zavolat pouze *GPU*-čko („*Device*“)
  - *\_global\_* - modifikátor pro tzv. *kernely* - funkce spouštějící se na *GPU* multiprocesorech („*Device*“) a jsou zavolány *CPU*-čkem („*Host*“)
- ▶ modifikátory proměnných - píšeme před datovým typem proměnné:
  - *\_device\_* - deklaruje proměnnou, která bude sídlit v „*Device*“ paměti po celou dobu běhu programu
  - *\_constant\_* - stejný jako *\_device\_* ale není možné ji přepsat - je „*read-only*“
  - *\_shared\_* - proměnná sídlí v „*Shared Memory*“ jednoho bloku - je přístupná pouze pro vlákna v tomto bloku, nikým jiným
  - *\_managed\_* - je přístupná jak z „*Host*“ tak i z „*Device*“ a je spravována tzv. „*Unified Memory*“ systémem - CUDA spravuje přesun dat automaticky
- ▶ datové typy a struktury:
  - všechny běžné datové typy v C/C++:

```
char, uchar // typedef unsigned char uchar
int, uint // typedef unsigned int uint
short, ushort // typedef unsigned short ushort
long, ulong // typedef unsigned long ulong
float, double
```

- používají se struktury se stejnými jmény jako datové typy s číselnou příponou (1 až 4)
  - značí kolik je tohoto datového typu vně struktury - přistupuje se k nim pomocí přístupových operátoru (. ->) a jmén položek .x, .y, .z, .w (klasická C syntaxe)
  - *int3*, *uchar3*, *float3*, *uint3*, ... , *dim3* = *uint3*

```
int3 some_variable(12, 34, 56); // int3 some_variable = { .x=12, .y=34, .z=56 };
some_variable.x = 61; // int3 some_variable = { .x=61, .y=34, .z=56 };
```

- `dim3` je datovým typem mřížky a bloku zadávané do trojitého „chevrons“ při volání *kernelu*
- `cudaError_t` je datový typ pro udržení *error code* (návratové hodnoty CUDA API funkcí)
  - je důležitý při ladění kódu – lze ho předat do `cudaGetErrorString(...)` funkce, která vrátí zprávu o tom, kde nastala chyba (co se stalo)
- ▶ předdefinované globální proměnné:
  - slouží k zjištění přesného *ID* (polohy, adresy) každého vlákna ve spuštěném *kernelu*
  - `dim3 gridDim` - udává dimenze trojrozměrné mřížky
  - `dim3 blockDim` - udává jak velký je blok v mřížce, opět je trojrozměrný
  - `uint3 blockIdx` - udává pozici bloku v mřížce, ve kterém se vlákno zrovna nachází
  - `uint3 threadIdx` - udává třídimenzionální pozici vlákna v bloku, ve kterém se nachází
  - `int warpSize` - záleží na architektuře grafické karty, udává kolik je vláken na jednom „Warp“
    - není pro výpočet důležitý - je užitečný pro optimalizaci výpočtu pro určité architektrury grafických karet
  - různé flagy/makra zadávané do `cuda` funkcí jako: `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, `cudaMemcpyDeviceToDevice`, ...



Obrázek 45: Způsob adresování jednotlivých vláken ve dvou-dimenzionální mřížce

```
uint3 pos = {
 blockIdx.x * blockDim.x + threadIdx.x,
 blockIdx.y * blockDim.y + threadIdx.y,
 blockIdx.z * blockDim.z + threadIdx.z
}; // uint3 pos ... představuje přesnou pozici jednoho vlákna v celé mřížce
```

- ▶ API funkce s předponou `cuda`: (je jich stovky, tady jen pár z nich)

```
cudaError_t cudaDeviceReset(void);
cudaError_t cudaDeviceSynchronize(void);
cudaError_t cudaGetLastError(void);
const char* cudaGetErrorString(cudaError_t error);
cudaError_t cudaMalloc(void** devPtr, size_t size);
cudaError_t cudaMallocManaged(void** devPtr, size_t size, unsigned int flags);
cudaError_t cudaFree(void* devPtr);
cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, cudaMemcpyKind k);
```

### 35. Čemu by se měl programátor vyhnout a jak CUDA funguje.

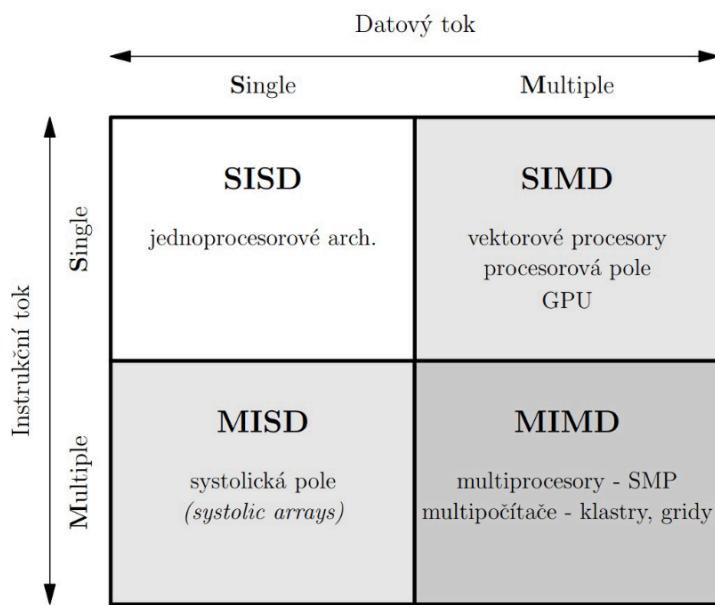
- fungování CUDA je popsáno v předchozích otázkách
- měl by se minimalizovat přesun dat mezi „*Host*“ a „*Device*“ paměťmi
  - přes API funkci `cudaMemcpy( ... )`, ideálně přesouvat data jen dvakrát - před výpočtem a po dokončení výpočtu
- používat *GPU* pouze pro výpočetně náročné úlohy:
  - image-processing (grafické editory)
  - signal-processing (DAW - Digital Audio Workspace)
  - v odvětví Machine-Learning - maticové a tenzorové operace
  - ale ne všechny úlohy jsou vhodné počítat přes grafickou kartu
- pro intensivní přesuny mezi paměťmi využít „*pipelining*“ a „*prefetching*“
  - přes API volání funkce `cudaMemPrefetchAsync( ... )` - asynchronně začne přesun dat za současného běhu hlavního programu
- data by se měla zorganizovat tak, aby se k nim mohlo přistupovat sekvenčně/sériově
  - *GPU* multiprocesory jsou optimalizovány na sekvenční přístup do svých „*Shared Memory*“
  - např. při násobení matic  $A \cdot B = C$  bychom měli matici  $B$  před výpočtem transponovat
- zredukovat co nejvíce počet „*divergentních*“ vláken
  - nejlépe, aby všechny vlákna vykonávali stejný sekvenční kód
- optimálně nakonfigurovat velikost mřížky (Grid) a bloků (Block) v mřížce
  - pomáhá znalost architektury pracující grafické karty

# Paralelní systémy

## Otázky:

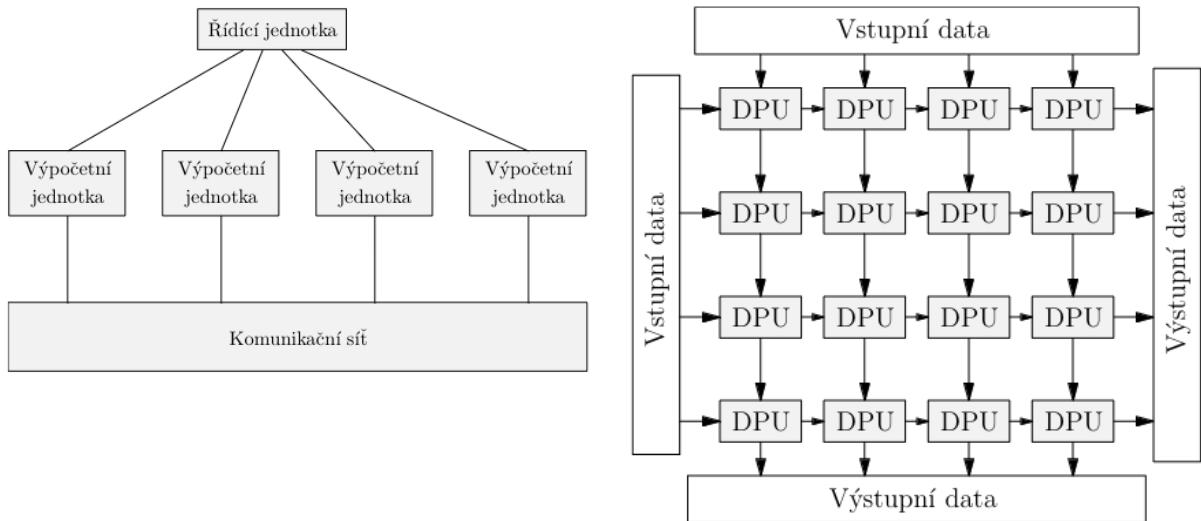
- 36) Charakterizujte Flynnovu taxonomii paralelních systémů.
- 37) Charakterizujte komunikační modely paralelních systémů.
- 38) Vysvětlit Amdahlův zákon a jak bychom se podle něj rozhodovali.

## 36. Charakterizujte Flynnovu taxonomii paralelních systémů.



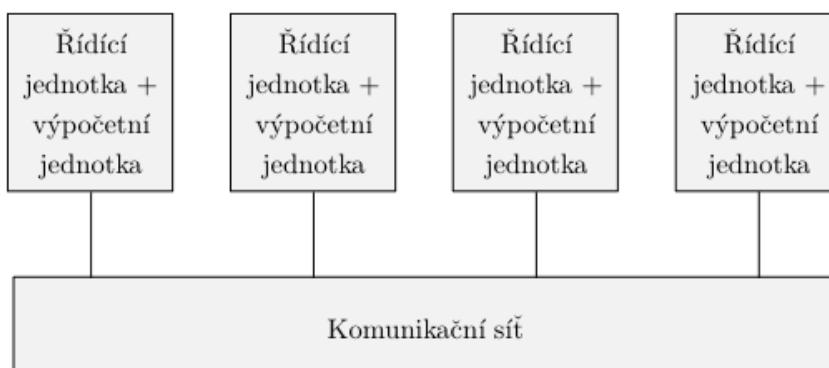
Obrázek 46: Flynnova taxonomie paralelních systémů

- **SISD architektury (Single-Instruction Single-Data)**
  - na jednu instrukci připadají jedny data
  - jde o typické jednoprocesorové architektury
    - mohou sice zpracovávat více instrukcí zároveň pomocí „pipelining“
    - to však neznamená, že provádí kód paralelně
    - jen zpracovávají několik instrukcí zároveň pomocí sekvenčního obvodu
- **SIMD architektury (Single-Instruction Multiple-Data)**
  - na jednu instrukci připadá více dat
  - příkladem mohou být různé vektorové a maticové operace
    - násobení vektorů a matic, sčítání vektorů a matic, konvoluce matic, ...
  - tyto architektury mají jednu řídicí jednotku a několik výpočetních jednotek
    - výpočetní jednotky v jednu danou chvíli provádí stejnou instrukci na jiných datech
  - jde tedy hlavně o vektorové procesory (specializované pro operace na vektorech) a procesorová pole (matice *elementárních* procesorů - PE - Processing Element)
  - jsou jimi např.:
    - procesory s rozšířenou instrukční sadou
      - MMX - *Multi-Media Extension*
      - SSE - *Streaming SIMD Extension*
      - 3DNow!
      - AVX - *Advanced Vector Extension*
      - AMX - *Advanced Matrix Extension*
      - VNNI - *Virtual Neural Network Instructions*
    - jednotlivé multiprocesory v grafických kartách



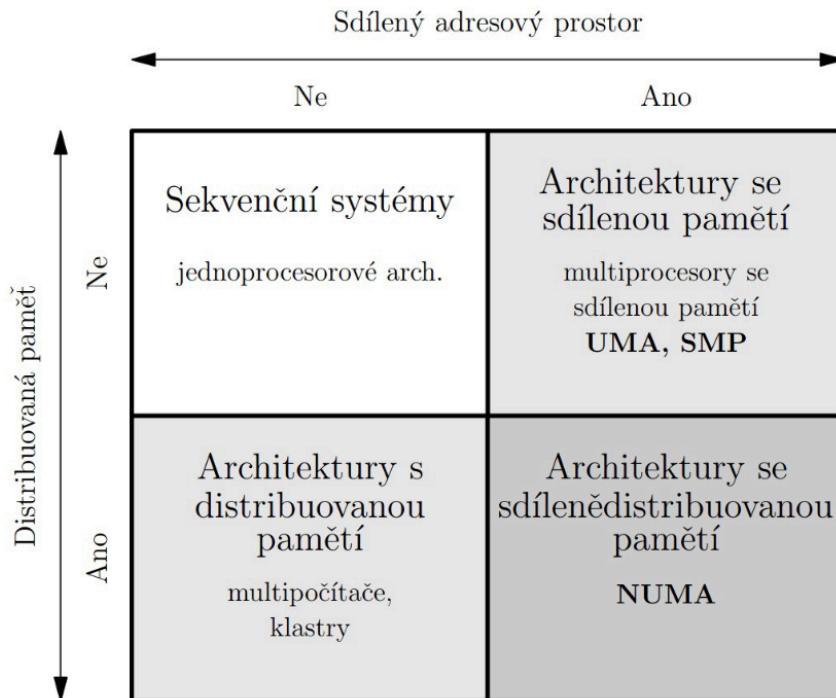
Obrázek 47: SIMD architektura a MISD architektura

- **MISD architektury (Multiple-Instruction Single-Data)**
  - jedny data jsou postupně zpracovány více instrukcemi
  - typickým zástupcem jsou systolická pole
    - je to homogenní síť úzce spojených DPU-ček („Data Processing Units“) neboli uzlů
    - z latinského *systola* - kontrakce srdce - krev jako data, kontrakce jako více instrukcí
  - použití:
    - obvody implementující třídicí algoritmy, Hornerovo schéma pro vyčíslení polynomu, násobení matic Cannonovym algoritmem apod.
- **MIMD architektury (Multiple-Instruction Multiple-Data)**
  - systémy schopné provádět různé instrukce nad různými daty
  - typickými zástupci jsou multiprocesory (na GPU) a multipočítáče
    - samostatné jednotky propojené komunikační sítí
  - v praxi se nepíše kód pro každý procesor zvlášť
    - všude běží stejný kód - podle ID procesu se zpracovávají různé větve kódu
    - mluví se spíš o SPMD (Single-Program Multiple-Data)
  - všechny významné paralelní architektury dnes spadají do MIMD kategorie



Obrázek 48: MIMD architektura

### 37. Charakterizujte komunikační modely paralelních systémů.



Obrázek 49: Klasifikace komunikačních modelů paralelních systémů

#### Architektury se sdílenou pamětí (ANO Sdílený adresový prostor, NE Distribuovaná paměť)

- obsahují fyzický sdílenou paměť (společná paměť i virtualní adresový prostor)
  - všechny procesory, výpočetní jednotky, do ní mají stejně rychlý přístup
  - jde o UMA (*Unified Memory Access*) architektury
    - stejná adresa na různých procesorech odkazuje na stejnou paměťovou buňku ve fyzické paměti
  - je prostředkem komunikace mezi procesory
    - tvoří úzké místo (*bottleneck*) - omezuje se počet procesorů (do 100)
- jednotlivé procesory mají také svou lokální paměť *Cache*
  - mnohem rychlejší - není přístupná jiným procesorům - to už není UMA ale NUMA (*Non-UMA*)
- typický příklad je SMP (*Symmetric Multiprocessing*)
  - grafické karty mívají několik multiprocesorů, každý se svou „Shared Memory“ (viz CUDA)
- standardem pro vývoj je OpenMP (*Open Multi-Processing*) API pro C, C++ a Fortran

#### Architektury s distribuovanou pamětí (NE Sdílený adresový prostor, ANO Distribuovaná paměť)

- nemají společnou paměť ani virtuální adresový prostor
  - adresa do paměti v procesorech neodkazuje na stejnou fyzickou paměťovou buňku
  - procesory komunikují spolu zasíláním zpráv přes komunikační síť
    - náročnější z pohledu programátora
- odstranění společné paměti umožňuje vytvářet systémy s tisíci procesory
- standardem pro vývoj je MPI (*Message Passing Interface*) API

#### Architektury se sdíleně-distribuovanou pamětí (ANO Sdílený adresový prostor, ANO Distribuovaná paměť)

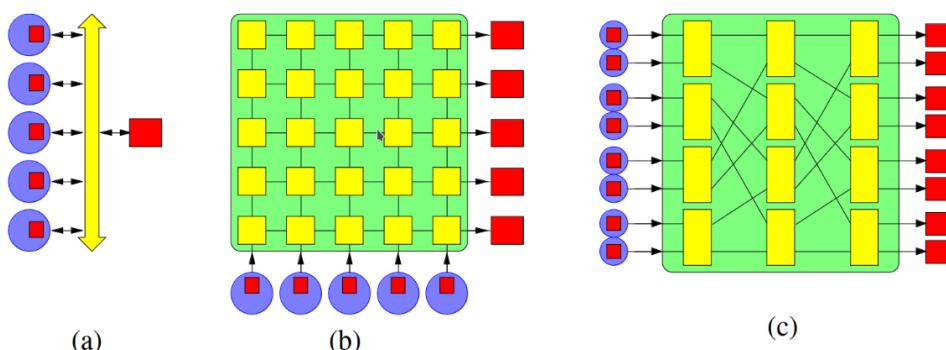
- jde o architektury s distribuovanou pamětí s podporou sdíleného adresového prostoru
- jde o NUMA (*Non-Unified Memory Access*)

### Porovnání systému se sdíleným a nesdíleným adresovým prostorem

- programování postavené na principu zasílání zpráv je náročnější
- posílání zpráv lze emulovat na systémech se sdílenou pamětí
  - programy napsané se standardem *MPI* (Arch. s dist. pam.) běží na SMP (Arch. se sdíl. pam.)
  - programy napsané standardem *OpenMP* (Arch. se sdíl. pam.) neběží na systémech s dist. pamětí

### SMP (Symmetric Multiprocessing) - multiprocesory se sdílenou pamětí

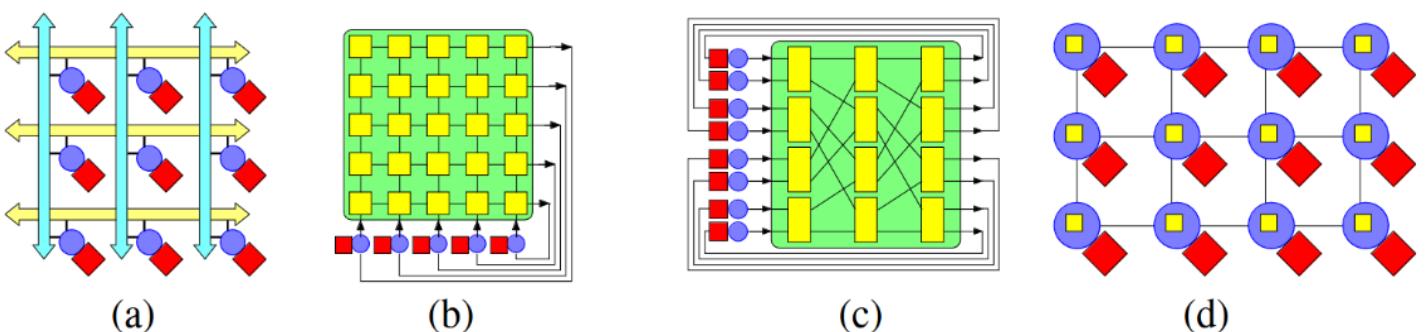
- stovky procesoru se skrytou pamětí (Cache paměti)
- mají jednu centrální sdílenou paměť - může mít několik bank
  - je nutná synchronizace přístupu do této paměti
  - škálovatelnost je limitována propustností paměťového rozhraní (Memory Interface)
- propojovací síť:
  - a) centrální sběrnice (*bus based*) - desítky okolo jedné sběrnice
  - b) křížový (*crossbar*) přepínač - přepínání procesorů mezi více bank paměti
  - c) nepřímá vícestupňová síť (*multistage interconnection network*) - levnější a chudší *crossbar*



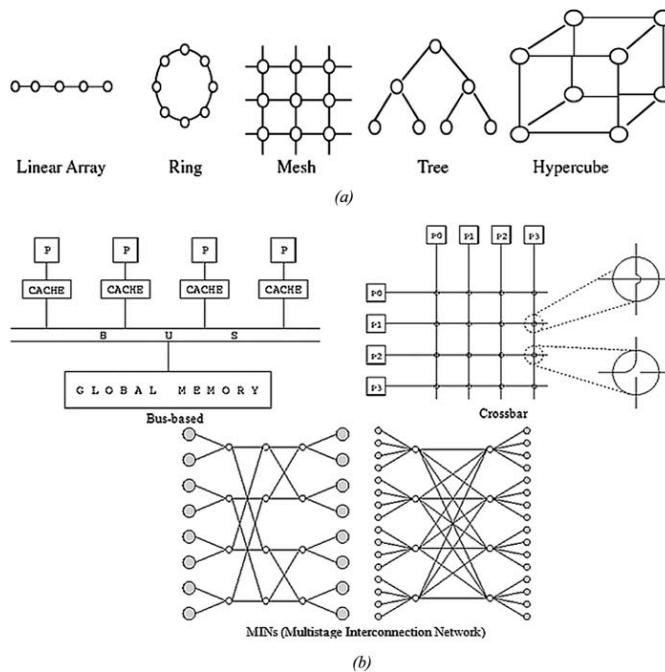
Obrázek 50: Projovací sítě u SMP architektur

### DMP (Distributed Multiprocessing) - multiprocesory s distribuovanou pamětí

- výkonné samostatné počítače s lokálními paměti
  - mají-li sdílenou paměť → mohou být také SMP
- všechny procesory si mohou současně přistupovat navzájem do svých lokálních pamětí
- škálovatelnost je mnohem vyšší než u SMP
- projovací síť:
  - a) 2D mřížka (*matrix based*)
  - b) křížový přepínač (*crossbar*)
  - c) nepřímá vícestupňová síť (*multistage interconnection network*)
  - d) přímá síť (*mesh*)



Obrázek 51: Propojovací sítě u DMP architektur



Obrázek 52: Různé typy projovacích sítí

### 38. Vysvětlit Amdahlův zákon a jak bychom se podle něj rozhodovali.

- u paralelizace je problém růstu výkonnosti jako celku
  - navýšením výpočetní síly, by měl (v ideálním světě) růst výkon o stejný faktor
  - objevují se zde však ztráty výkonu:
    - při komunikaci - pomalé sběrnice, nutná koordinace (jeden po druhém) apod.
    - kvůli nerovnoměrné či neoptimální vytížení procesorů - některé procesory pracují naplno, jiné jen se jen částečně zapojí do zpracování úlohy
    - kvůli neznalosti vhodných algoritmů - vyberem neoptimální řešení pro řešení problému
- zrychlení jsou trojího typu:
  1. zpomalení - velmi často se systém zpomalí
  2. lineární - ideální růst výkonu
  3. superlineární - nad-lineární růst výkonu (vzácně)
- čistě paralelních úloh je velmi málo:
  - ve většině případů je kombinováno se sériovým zpracováním - program není paralelně zpracován po celou dobu jeho běhu, většina segmentů je zpracována sériově
  - paralelizace je proto jenom částečná
- zrychlení výkonu lze odvodit z Amdahlova zákona:

$$S_z = \frac{1}{(1 - f_p) + \frac{f_p}{N}} \text{ nebo } S_z = \frac{t}{tf_s + t\frac{f_p}{N}} \text{ kde platí } f_s + f_p = 1$$

$S_z$  součinitel zrychlení - poměr doby výpočtu na jednom procesoru k době při částečné paralelizaci

$f_s$  je podílem sériové délky při výpočtu (běhu programu)

$f_p$  je podílem paralelní délky při výpočtu (běhu programu)

$t$  je celková doba sériového výpočtu (běhu programu)

$N$  je počet použitých procesorů při paralelním výpočtu (běhu programu)