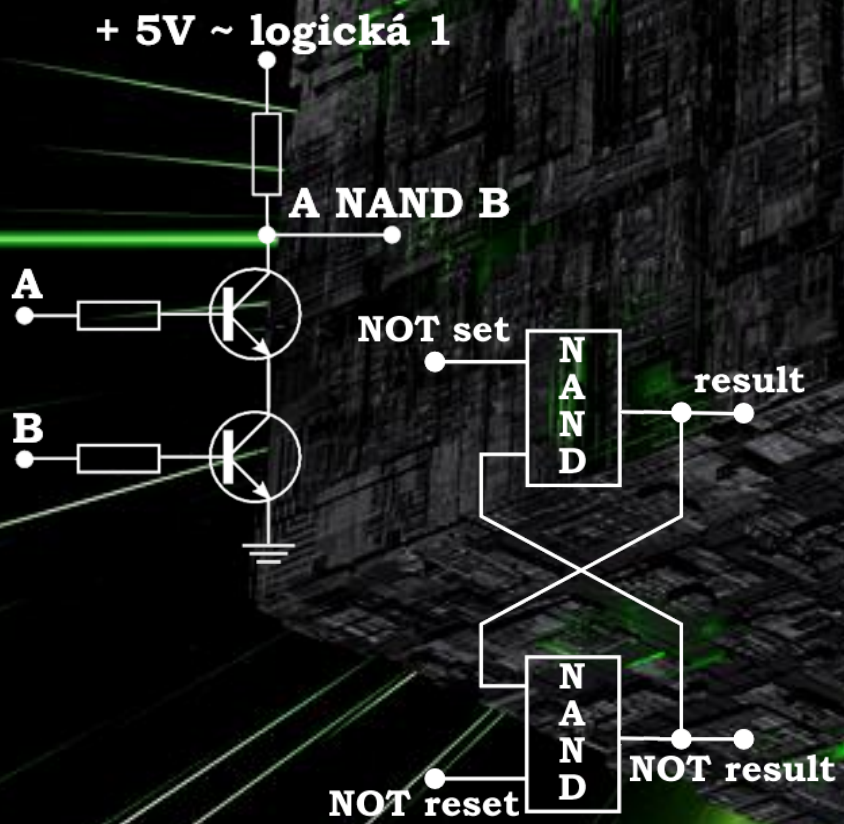
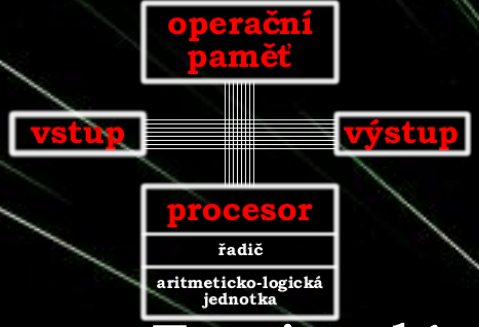
Principy počítačů

Optimalizace



Martin Urza

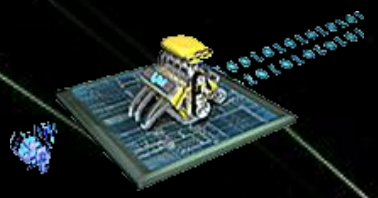


Opakování z minulé přednášky

- Z minulé přednášky by mělo být jasné, jak přibližně v hrubých obrysech funguje podle von Neumannovy architektury paměť, procesor, I/O, celý počítač….
- Dále bylo zmíněno, že von Neumannova architektura je **s určitými drobnějšími změnami** používána ve většině počítačů dodnes.
- Důvodem, proč dnešní počítače von Neumannově architektuře přesně neodpovídají, je optimalizace.
  - Von Neumannova architektura je sice velmi názorná a přehledná, ale některé problémy lze řešit efektivněji.
  - Účelem této přednášky je postupně rozebrat většinu podstatných optimalizací této architektury.

Proč optimalizovat von Neumannovu architekturu?

- Je už velmi stará, vznikla skoro před stoletím, což je ve výpočetní technice věčnost.
  - Podivuhodné je už to, že tato architektura vydržela tak dlouho (byť jen v hrubých obrysech).
- Za celou tu dobu nikdo nic jiného (rozumného) nevymyslel.
  - Respektive vymyslel, ale neujalo se to.
- Von Neumannova architektura bude pravděpodobně využívána i nadále, dá se však očekávat, že budou vymýšlena stále nová a nová zlepšení (optimalizace), které budou tuto architekturu (v rámci zvyšování výkonu) měnit.

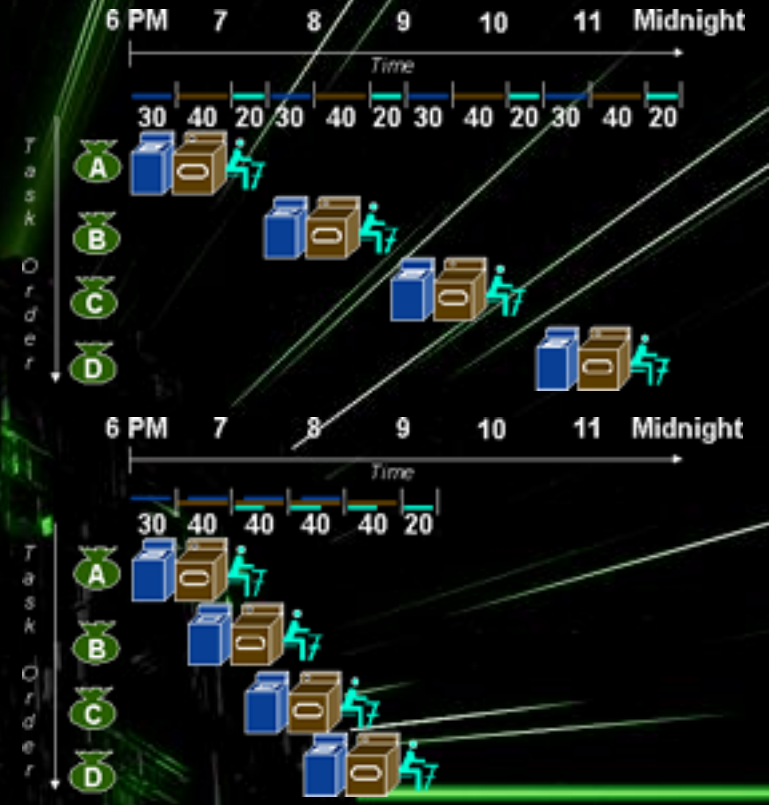


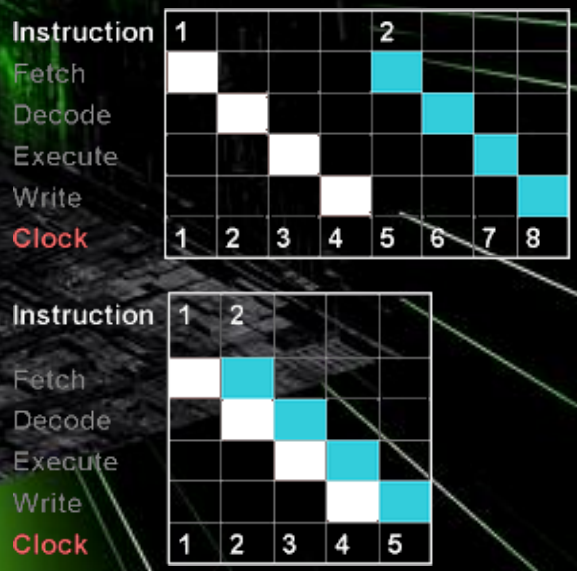
Zpracovávání instrukcí

- Podle von Neumannovy architektury provádí počítač v každý okamžik jen jednu činnost, tedy procesor vykonává vždy jedinou instrukci a až poté, co ji dokončí, začne vykonávat další (ne dřív).
- Tento předpoklad je velmi důležitý k tomu, aby bylo vůbec možné psát programy (pro korektnost kódu algoritmů je nutné, aby šly instrukce za sebou).
- Na druhou stranu je sériové zpracovávání instrukcí velmi pomalé a tím pádem zoufale neefektivní.
- Dnešní počítače jsou tedy konstruovány tak, že tuto podmínku nesplňují (kvůli efektivitě). To klade jisté nároky na programátory, ale také na překladače.

Pipelining

- Procesor provádí mnoho typů činností; každý typ je vykonáván fyzicky jinou částí procesoru.
  - V procesoru jsou fyzicky oddělená místa pro načítání instrukcí, počítání, dekódování instrukcí a tak dále.
- Toho lze využít pro zvýšení efektivity – ve chvíli, kdy je jedna instrukce vykonávána, může být zároveň jiná načítána, další dekódována a podobně.





Problémy s pipelinингem

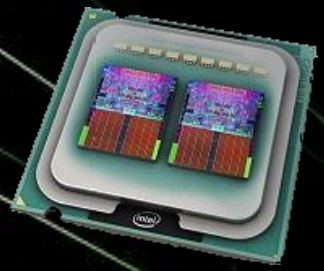
- Hloubka pipeline (tj. počet činností, na které se dá instrukce rozložit) na reálných procesorech může být až 20 (většinou méně), takže než procesor zjistí, co má instrukce dělat, je rozpracováno několik dalších.
  - To vadí skokům a zejména podmíněným, protože dokud není jasné, že se jedná o instrukci skoku (a ani není znám jeho cíl), načítají se instrukce, kterými by procesor pokračoval, kdyby ke skoku nedošlo.
- Řešení jsou různá pro různé architektury procesorů.
  - IA32 to řeší velmi složitě hardwarově, skoky detekuje předem, u podmíněných predikuje, jestli k nim dojde.
  - MIPS prostě vykoná nějaké instrukce za skokem.
    - Což musí řešit překladač (či programátor assembleru).

Hyperthreading

- Jedno jádro se navenek tváří jako více procesorů.
- Jádra s hyperthreadingem v sobě mají některé často používané části fyzicky víckrát, jiné ne.
  - Například obvody pro načítání a dekódování instrukcí budou určitě zastoupeny víckrát.
  - Aritmetické výpočetní obvody jsou v těchto jádrech zastoupeny podobně jako v jiných.
- Jádro s hyperthreadingem zpracovává dva toky instrukcí paralelně.
  - Některé části instrukcí se dějí skutečně současně, je-li to fyzicky možné (jádro na to má oddělené části).
  - Jiné části instrukcí se o zdroje střídají (je-li to možné).

Problémy s hyperthreadingem

- Operační systémy musí hyperthreading podporovat a vědět, které ze samostatně jevících se procesorů jsou ve skutečnosti jedno jádro.
  - Operační systémy totiž rozhodují, který program (resp. vlákno) poběží na kterém procesoru.
  - Příklad: Počítač má dvoujádrový procesor a mají na něm běžet dvě aplikace (resp. vlákna). Je-li zapnutý hyperthreading, systém vidí čtyři procesory. Jsou-li dvě vlákna naplánována na jedno jádro, běží pomaleji než bez hyperthreadingu (to by bylo každé vlákno naplánováno na jiné jádro).
- Využívají-li dvě běžící vlákna stejné výpočetní části jádra, musí na sebe stejně čekat, což je neefektivní.



Multiprocesory a vícejádrové procesory

- V počítači může být více procesorů, případně každý procesor může mít více jader.
  - Rozdíl mezi dvěma jednojádrovými procesory a jedním dvoujádrovým procesorem je jen v cache, což je paměť procesoru (více o tom níže), kterou mají jádra jednoho procesoru společnou, více procesorů ji má oddělenou; krom cache má každé jádro totéž (všechny části), co celý procesor.
- Počítače s více procesory (či jádry) zpracovávají více toků instrukcí současně.
- Procesory jsou rovnocenné, žádný není hlavní (jen při startu počítače chvilku pracuje pouze jeden).

Problémy s více procesory či jádry

- Běží-li na více procesorech zcela oddělené aplikace, téměř žádné problémy nejsou.
- Má-li jedna aplikace (a zejména například operační systém) využívat více procesorů efektivně, je tvorba takového programu extrémně náročná.
  - Pro problémy z podstaty paralelní to tak moc náročné není (i tak je ale nutné paralelismus zohlednit).
    - Například systémy, které využívá více uživatelů naráz a každý dělá něco jiného (třeba různé webservery).
  - Napsat efektivně paralelní aplikaci, která řeší problém, jenž je „od přírody“ sériový, je vrchol dovedností dnešních programátorů (a moc z nich to dobře neumí).
    - Například různé office programy a podobně.

Obecně o paralelismu

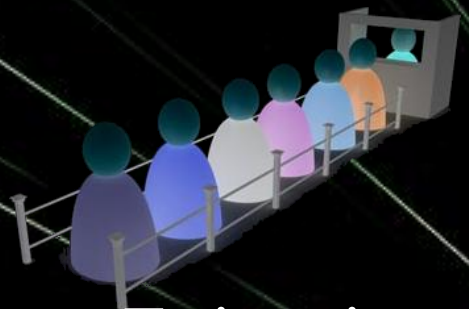
- Mnohá technologická zlepšení (například zvyšování taktovací frekvence procesorů) se projeví tak, že stejné aplikace prostě najednou běží rychleji (bez zásahu programátora).
- Paralelismus sám o sobě jednu aplikaci nezrychlí.
  - Klade vysoké požadavky na programátory.
- Program navíc není jen paralelní/neparalelní (tedy jednovláknový/vícevláknový). U paralelismu záleží navíc na jeho míře, tedy počtu vláken.
  - Dvouvláknový program poběží na čtyřech procesorech stejně jako na dvou, aplikace je tedy optimální vždy pro nějaký počet procesorů.

Aktivní a pasivní čekání

- Příklad aktivního čekání:
  Oslík: Už tam budem?
  Shrek: Ne.
  Oslík: Už tam budem?
  Shrek: Ne!
  Oslík: Už tam budem?
  Shrek: Ne!!
  Oslík: Už tam budem?
  Shrek: Jo.
  Oslík: Vážně?
  Shrek: Ne.
- Příklad pasivního čekání:
  Oslík: Řekni, až tam budem.
  Shrek (časem): Jsme tam.
- Není to moc vhodné do filmu, protože to není taková sranda.
- Na druhou stranu je to o poznání praktičtější a funkčnější přístup.
- V počítači řešíme podobný problém. Procesor potřebuje být stále informován o tom, jestli náhodou někdo třeba nestiskl klávesu, či nepohnul myší. Při použití aktivního čekání by se musel (jako osel) všech zařízení pořád ptát.

Přerušení

- Mechanismus pasivního čekání.
- Všechna zařízení mají možnost přerušit procesor v jeho činnosti, proto se tato akce nazývá **přerušení**.
- Procesor si uloží rozdělanou práci, nastane **obsluha přerušení** a pak procesor pokračuje v tom, co dělal.
  - Příklad: Klávesnice je dost málo využívané zařízení, i při velmi rychlém psaní nepřijde z klávesnice ani deset znaků za vteřinu, což je pro procesor věčnost (vteřina jsou miliardy instrukcí). Místo toho, aby procesor stále zjišťoval, jestli někdo nezmáčkl klávesu, pošle klávesnice procesoru přerušení, kdykoliv se tak stane. Procesor přeruší svou práci, uloží si ji, načte vstup z klávesnice, obnoví rozdělanou práci a v té pokračuje.



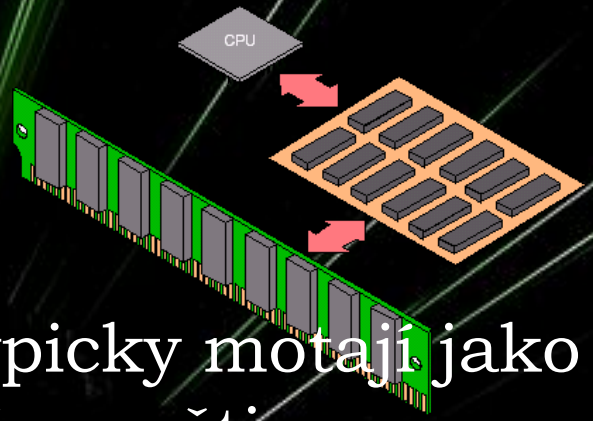
Několik detailů o přerušení

- Existuje tzv. řadič přerušení, který rozhoduje, které přerušení pošle jakému procesoru.
  - Řadič má také způsob (vektor přerušení), kterým procesoru řekne, které zařízení přerušení poslalo.
- Obsluha přerušení probíhá tak, že operační systém přiřadí každému zařízení, které je schopno vyvolat přerušení, adresu paměti, ve které je uložen kód, který má přerušení obsloužit.
  - Tento kód patří operačnímu systému, typicky ovladači daného zařízení.
- Když je přerušení obslouženo, procesor se vrátí zpět tam, kde přestal (ve vykonávání původního kódu).

Cache

- Procesory jsou výrazně rychlejší než paměti.
- Požaduje-li procesor nějaká data z paměti, té docela dlouho (desítky nanosekund) trvá, než je dodá; za tu dobu vykoná procesor až malé stovky instrukcí.
- Cache jsou paměti, které jsou výrazně rychlejší, data dokážou dodat během několik nanosekund, ale jsou výrazně (řádově) menší než operační paměť.
- Protože je cache rychlejší než hlavní paměť, bylo by nejvýhodnější mít všechna data uložena v ní. To ale nejde, protože cache je malá. Co s tím?
  - Bylo by dobré tam mít ta data, která budeme někdy v dohledné době potřebovat.

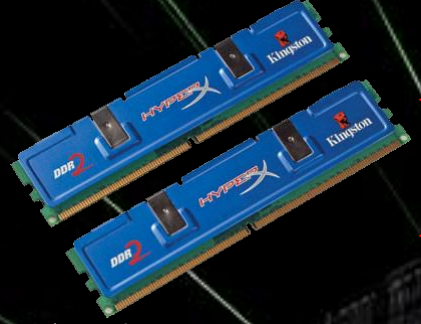
Správa cache



- Využíváme toho, že programy se typicky motají jako vítr v bedně stále na stejném místě paměti.
  - To je dáno tím, jak procesor provádí instrukce: buď se vrací tam, kde už byl (cykly), nebo jde pomalu vpřed.
- Načte-li procesor data z paměti, uloží se kopie těchto dat (plus nějaká další data z adres kolem nich) do cache. Není-li v cache místo, uloží se tato data přes nejstarší z těch, co tam jsou (u nejstarších dat je nejnižší pravděpodobnost, že budou znovu potřeba).
- Chce-li procesor načíst data z paměti, požadavek vyřizuje nejdřív cache, která se podívá, jestli je náhodou nemá; pokud ano, pošle je procesoru.

Různé významy slova cache

- Českým ekvivalentem slova cache je vyrovnávací paměť a může označovat více různých věcí (vždy se jedná o paměť).
  - Procesorová cache.
    - L1, L2 (L3).
    - TLB.
  - Cache pevného disku.
    - Softwarová v OS.
    - Hardwarová přímo v disku.
  - Databázová cache.
- Účel i princip fungování je vždy prakticky stejný, rozdíly jsou jen v detailech.



DMA

- DMA (Direct Memory Access) je způsob kopírování dat (což je pomalá operace) z jednoho zařízení do druhého, aniž by tato data šla přes procesor.
- Procesor DMA inicializuje, pak už DMA běží sám.
  - Příklad: Zařízení A posílá data zařízení B.
    - Řadič řekne zařízení A, že od něj chce procesor data.
    - Řadič řekne zařízení B, že mu procesor pošle data.
    - Protože procesor i obě zařízení jsou na stejné sběrnici, mají obě zařízení pocit, že komunikují s procesorem, ač tomu tak ve skutečnosti není.
    - Procesor může v mezičase vykonávat nějaké instrukce, ke kterým nepotřebuje sběrnici.
      - Sběrnice je využita zařízeními A a B.

CISC vs RISC

- Dříve bylo zvykem dělat procesory s obrovskými instrukčními sadami (Complex instruction set computer – CISC). Takové procesory měly pro hodně složitějších operací speciální instrukce.
  - Zjistilo se, že 98% času bylo vykonáváno jen 15% instrukcí, tedy 85% instrukčních sad bylo téměř nevyužito, většina programů nepoužívala více než polovinu instrukční sady. Proto nastupuje RISC....
- RISC (reduced instruction set computer) používá jen málo instrukcí, složitější úkony musí programátor či překladač tvořit pomocí více jednoduchých instrukcí.
  - Tyto jednoduché instrukce jsou optimalizovány na co nejvyšší výkon.

Proč je CISC pomalejší než RISC?

- CISCové procesory realizují složité instrukce pomocí tzv. mikrokódu, kterým jsou složitější instrukce programovány, tedy skládány z jednodušších.
  - To znamená, že takto složená instrukce je samozřejmě pomalejší, protože její vykonání trvá více taktů (tiků hodin).
- Obecně platí, že hardwarové řešení čehokoliv bývá rychlejší než softwarové.
  - Mikrokód jsou vlastně softwarově naprogramované instrukce.
    - Toto naprogramování nedělají aplikační programátoři, nýbrž návrháři procesorů, nelze měnit, je pevně zadrátované uvnitř procesoru.

Post-RISC

- Nakonec se ukázalo, že oba přístupy mají něco do sebe.
  - RISCové procesory jsou efektivnější, složitější příkazy složené z více jednoduchých instrukcí jsou v součtu rychlejší než složitá instrukce, která trvá dlouho.
  - CISCové procesory zase ulehčovaly programování, respektive později tvorbu překladačů.
- Některé procesory tedy kombinují oba přístupy, z každého využívají lepší vlastnosti.
  - Takové procesory mají širokou instrukční sadu, což je typické pro CISCovou architekturu.
  - Nejčastěji používané instrukce jsou optimalizované.

Rekapitulace

- Celá tato přednáška (až na slajdy o RISCových a CISCových instrukčních sadách) je o optimalizaci von Neumannovy architektury prvky, které do ní původně nepatří.
- Seznam optimalizací je téměř vyčerpávající, alespoň tedy v podstatných věcech. Existuje sice mnoho dalších detailů, ale to podstatné bylo řečeno v této přednášce.
- Pochopíte-li von Neumannovu architekturu a její základní fungování a poté ještě navíc optimalizace, o kterých mluví tato přednáška, získáte povšechnou představu o fungování většiny dnešních PC.