Számítógép Architektúrák

Információfeldolgozási modellek

1. **Vezérlésáramlásos modell**
   * vezérlő token
   * nem veszi észre a párhuzamosságot
   * Fork/Join -> programozó dolga
   * szekvenciális programok
   * Neumann architektúra
2. **Adatáramlásos modell**
   * végrehajtás akkor, ha minden operandus rendelkezésre áll
   * precedenciagráf
   * párhuzamosság automatikusan
   * pl. táblázatkezelő szoftverek
3. **Igényvezérelt modell**
   * végrehajtás akkor, ha az eredményre szükség van
   * első: közvetlenül az eredményhez vezető művelet
   * párhuzamosság automatikus
   * pl. funkcionális programozási nyelv alapja

Neumann – architektúra

* újítás: utasítások az adatokkal együtt vannak a memóriában
* gyorsabb és könnyebb programot bevinni
* 3 fő komponens: CPU, memória, perifériák
* nincs megkülönböztetve az adat az utasítástól -> proci hívásától függ
* szűk keresztmetszet: az összeköttetés, CPU és memória egymástól való eltávolítása
* busz és memória sebessége nem tud lépést tartani -> memóriarés/fal -> lemarad ☹

Harvard – architektúra

* utasítás és az adatok fizikailag elkülönülnek -> két külön interfész
* a két memória különböző technológiájú is lehet
* módosított Harvard: képes írni és olvasni az utasítás memóriát is
  + önmódosító program

Megszakításkezelés

1. **Polling**
   * minden perifériát körbekérdez -> eltart egy ideig
   * prioritás: melyiket kérdezi hamarabb
   * pl. egér
2. **Daisy chaining**
   * sorba kötött perifériák közül megkapja a sínhasználati jogot
     + ha nem ő volt -> továbbadja
     + ha ő volt: adatbuszra teszi az azonosítóját -> proci tudni fogja, ki kérte
3. **PIC** (Interrupt vezérlő)
   * több bemenete van, önmaga is egy periféria
   * igazságosabb, mint a Daisy chaining

Arbitráció a buszon

**Centralizált:**

* soros: Daisy chaining
* arbiter -> buszhozzáférés-vezérlő
* ha nem szeretné használni, továbbadja
* könnyen bővíthető
* minden igényt egyidejűleg megkap -> eldönti, ki kapja a buszhasználat jogát
* párhuzamos
* rugalmasabb priorizálás
* körbenforgó
* késleltetésérzékeny perifériák gyakrabban nyernek

**Elosztott:**

* nincs arbiter
* mindenki látja a többiek kérését
* prioritás ismerete -> visszavonhatja a saját kérését
* önkiválasztó arbitráció (pl. SCSI busz)
* ha rá akar tenni, ráteheti -> közben hallgatózik
* ha zagyvaságot hall, akkor sikertelen és megismétli
* ütközésdetektáláson alapuló
* kevésbé érzékeny a meghibásodásra, mivel nincs kulcsfontosságú szereplő

Memória

**SRAM:**

* felépítés: 6 tranzisztor -> 6T
* gyorsabb
* kisebb adatsűrűség
* cache memória
* könnyű CPU mellé integrálni, mivel olyan, mint a CPU -> csupa tranzisztor
* azonos kapacitás mellett többet fogyaszt
* a tápellátás megszűnésével tartalma elvész

**DRAM:**

* felépítés: 1 kondenzátor, 1 tranzisztor -> 1T1C
* 10 ms-onként frissíteni kell
* lassabb
* nagyobb adatsűrűség
* rendszermemória
* nehezen lehet CPU mellé integrálni -> kondenzátor miatt
* a tápellátás megszűnésével tartalma elvész
* kevesebb energiát fogyaszt

Virtuális tárkezelés

* Programok: Virtuális címeket használnak
* CPU lábain/buszon: Fizikai címek jelennek meg
* Virtuális → fizikai cím leképzés: Címfordítás
  + Aki csinálja: MMU
  + Amikor csinálja: minden memóriahivatkozáskor
  + Egysége: Lap (fix) vagy szegmens (változó méretű)
* Ha nem fér mindenki a fizikai memóriába → háttértárra kerül (swap-elés)
* A virtuális címtartományt fix méretű lapokra particionáljuk
* A fizikai címtartományt ugyanekkora keretekre osztjuk
* Lap ↔ keret összerendeléseket tárolja: laptábla
  + A laptábla is a fizikai memóriában van!
* TLB: translation lookaside buffer
* A gyakran használt virtuális ↔ fizikai összerendelések cache-e
* Címfordításkor először a TLB-ben keres
* TLB lefedettség: a TLB bejegyzések által lefedett címtartomány
* Cél:
  + Címfordítás legyen minél gyorsabb → virtuális cím szerinti keresés legyen gyors
  + minél kevesebbszer kelljen a memóriához nyúlni
  + Laptábla legyen a lehető legkisebb

Utasításkészletek

* Processzorok jellemzője: Programozási felület / Utasításkészlet architektúra
* Részei:
  + Utasítások
  + Támogatott adattípusok
  + Regiszterek
  + Címzési módok
  + Jelzőbitek
  + Perifériakezelési mód
  + Megszakítás- és kivételkezelés
* Utasítások felépítése:
  + Utasítás kódja / művelet típusa
  + Operandusok címei / értékei
  + Eredmény címe
  + Következő utasítás címe
* Bájtsorrend:
  + Little endian: legkisebb helyiértékkel kezdi
  + Big endian: legnagyobb helyiértékkel kezdi
  + Példa: 23423765 (=1656B15)
    - Little endian: 15 6B 65 01
    - Big endian: 01 65 6B 15
* CISC:
  + Kényelmes, összetett műveletek
  + Regiszter-memória utasítások (pl. R1 ← R2 + MEM[42])
  + Redundancia
  + Sokféle címzési mód
  + Változatos utasításhossz
  + Változatos utasítás-végrehajtási idő
* RISC:
  + Elemi utasítások, redundancia kerülése
  + Load-Store és regiszter-regiszter műveletek

R1 ← R2+MEM[42] helyett R3 ← MEM[42]; R1 ← R2+R3

* + Kevés címzési mód
  + Fix utasításhossz
  + Egyforma utasítás-végrehajtási idők

Cache memória

* Lokális elvek:
  + - Időbeliség: Ha egyszer hozzányúltunk, többször is hozzá fogunk
    - Térbeliség: Ha hozzányúltunk, a környezetéhez is hozza fogunk
    - Algoritmizálás: Speciális adatszerkezetek kiszámítható bejárása
  + Cache megvalósítások:
    - Címzési mód szerint

Transzparens

Nem transzparens

* + - Menedzsment szerint

Implicit

Explicit

Cache szervezés

Szempontok

* + - Gyors kereshetőség, egyszerű, olcsó

Tárolási egység

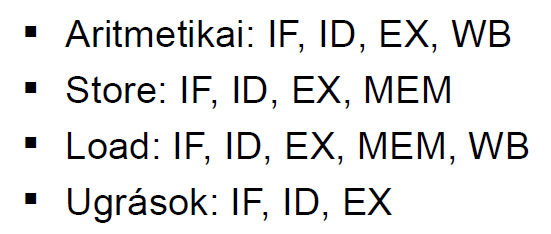
* + - 1 blokk = 2L
    - Címek alsó L bitje: blokkon belüli eltolás
    - Felső bitek: blokk sorszáma

Cache minden blokkja mellé tárolt járulékos információ

* + - Cache tag: operatív memória hányas blokkja
    - Valid bit
    - Dirty bit

1. Teljesen asszociatív szervezés
   * Blokkok a cache-ben bárhova elhelyezhetők
   * Sokat fogyaszt
   * Keresés: Címblokk száma és az összes cache tag komparálása
   * Komparátorok szélessége = Blokkszám bitszélessége
2. Direkt leképzés
   * Minden memóriabeli blokk csak egyetlen helyre kerülhet a cache-ben
     + Ezt a memóriabeli blokkszám egyértelműen eldönti
       - Pl: blokkszám alsó bitjei alapján
   * Keresés
   * Indexelés: A cím blokkszámából tudható
   * Komparálás: A cache ezen részeiben ez a blokk van
3. N-utas asszociatív szervezés
   * Kombinálja a korábbi 2 megoldást
   * A blokkszám alsó bitjei meghatározzák a blokk helyét
     + De ez nem egyértelmű
     + Az alsó bitek kijelölnek egy halmazt, ahol a blokk lehet
     + A halmaz n cache blokkból áll, ezek bármelyikében lehet a keresett blokk
   * Keresés
   * Indexelés: A cím blokkszámából tudható
   * Komparálás: A cache n bizonyos rekesze közül melyikben van ez a blokk

Pipeline utasításfeldolgozás

* melyik fázisban mi történhet:
  + - IF: laphiba, védelmi hiba
    - ID: érvénytelen utasítás
    - EX: aritmetikai hiba
    - MEM: laphiba, védelmi hiba
    - WB: nem történhet kivétel
  + Az utasítások többnyire nem függetlenek!
  + Erőforrásokért versengenek → Feldolgozási egymásrahatás
  + Egyik operandusa a másik eredménye → Adat egymásrahatás
  + Ugrás: nem tudjuk a folytatást, amíg ki nem értékeljük a feltételt → Procedurális egymásrahatás
  + RAW
    - oka a hosszabb pipeline
    - későn áll elő az eredmény
    - ha valaki a korábbi eredménytől függ tovább kell várnia
    - több utasítás van feldolgozás alatt
  + WAW
    - 2 utasítás ugyan abba a regiszterbe menti az eredményét
    - ha megengedjük, hogy az utasítások sorrenden kívül fejeződének be, baj lehet
      * korábbi eredmény kerül bele
      * ID fázisnak detektálnia és szünetekkel kezelnie kell

Szuperskalár processzorok

* A processzor több utasítást is le tud hívni egyszerre
* A processzor végzi a
  + Párhuzamosan végrehajtható utasítások kiválogatását
  + Az utasítások végrehajtó egységhez rendelését
  + A függőségi analízist (mikor hajtható végre egy utasítás)
* Ha minden fázisban m új utasítás végrehajtása kezdődik meg → m-utas szuperskalár processzor
* Kétféle megoldás:
  + In-order: végrehajtási sorrend: program követése
  + Out-of-order: átrendezi az utasításokat, hogy gyorsabb legyen
* Szuperskalár: a CPU keresgéli a független, párhuzamosan végrehajtható utasításokat
* A fordítóprogram is megteheti! Sőt, jobban! Több utasítást lát!
* Kapjunk vérszemet: Csináljon mindent a fordító!
* Párhuzamosan végrehajtható utasítások gyűjtése, utasítások műveleti egységhez rendelése, egymásrahatások detektálása és feloldása

VLIW architektúra

* VLIW = Very Long Instruction Word
* Nem utasításokon dolgozik, hanem utasításcsoportokon
* Ezek egy egységként haladnak a pipeline-ban
* Csoporton belüli utasításoknak nincs külön címe, az egész csoportnak közös címe van
* Csoportok szerepe:
  + Független utasítások kijelölése
  + Utasítások műveleti egységhez rendelése
* Nem használt pozíciókban: NOP
* A VLIW processzorok nem foglalkoznak egymásrahatásokkal!
* Mi van, ha adat-egymásrahatás van, és szünetet kellene beiktatni?
* Processzor fütyül rá -> Vegye észre a fordító! -> Iktasson be egy csupa NOP csoportot!
* Nem sikerült elég sok párhuzamosítható utasítást találni → rossz kihasználtság

EPIC architektúra

* Utasításcsoport: párhuzamosan végrehajtható utasítások halmaza
* Csoport összetétele: sablon meghatározza
* Pl. M – memória, I – egész, F – lebegőpontos, B – ugrás
* MFI: egy memória, egy lebegőpontos és egy egész van benne
* MFI, MMI, MII, MIB, MMF, MFB, stb.
* Specifikáció rögzíti a megengedett kombinációkat
* Processzorcsalád bővítésekor ez csak bővülhet
* Egymásrahatásokat a processzor kezeli
* Utasításcsoport méretének semmi köze a műveleti egységek számához!

Elágazásbecslés

* Procedurális egymásrahatás:
  + Ugrások okozzák
  + Megtörik az utasításfolyam szekvenciális viselkedését
  + Mi a gond? Pl. feltételes ugrás esetén tudni kell:
    - Az elágazás kimenetelét (ugrik-e, vagy sem)
    - Ugrás esetén az ugrási címet (hová ugrik)
* IF dolga: utasítások betöltése
* Nincs ideje feltételkiértékelésre és címszámításra!
* Egy tipphez van ideje – Ha bejön, annak örül – Ha nem:
  + A tévedésből elkezdett utasításokat érvényteleníteni kell » Legközelebb tanul a hibájából
* Minél később derül ki a rossz döntés, annál több időt fecsérel a processzor tévedésből betöltött utasításokra
* Minél hosszabb a pipeline, annál nagyobb a veszteség!
* Minden ugró utasításhoz egy állapotgépet rendelünk
  + Ez tárolja, mennyire szeret ugrani
  + Ha a közepesnél jobban szeret, akkor ugrásra tippelünk
  + Táblázat frissítése: a tényleges bekövetkezés ismeretében
* Tárolás: táblázatban: PHT (Pattern History Table)
* A trükk: tároljuk az egymást követő utasítások kimeneteleit egy shift regiszterben → globális előzmény regiszter
* Honnan kell betölteni a következő utasítást?
  + Ezt az IF-nek sürgősen tudni kell!
  + Feltételes és feltétel nélküli ugrásoknál egyaránt kérdés.
  + Kiolvassuk egy speciális "ugrási cím buffer"-ból (BTB: Branch Target Buffer)
  + Könnyebb/gyorsabb innen kiolvasni, mint kiszámolni

Vektorprocesszorok

* Hogyan tettük eddig hatékonnyá az utasítás-végrehajtást?
* Utasítás szintű párhuzamosítással
* Egyszerű pipeline: átlapoltuk több utasítás végrehajtását
* Szuperskalár: a CPU több, általa párhuzamosíthatónak talált utasítást hajt végre egyszerre
* VLIW/EPIC: a CPU több, a fordító / programozó által párhuzamosíthatónak talált utasítást hajt végre egyszerre
* Megvannak a korlátai:
  + Minél több utasítás áll feldolgozás alatt egyszerre, annál nagyobb a sansz az egymásrahatásra
  + Sok műveleti egység → sok forwarding út kell
  + Ciklusidő nem lehet tetszőlegesen rövid
  + Időnként becslés / spekuláció szükséges → ha rosszul sikerül, kárba ment egy csomó idő
* Klasszikus skalár adattípus és műveletek mellett / helyett
  + Vektor adattípus
  + Vektorkezelő utasítások
* Minden modern szuperszámítógép ilyen
* Miért is jobb vektorprocesszorral?
  + Rövidebb, tömörebb kód
  + Nincs szükség ciklusra!
* A vektorműveletek implicit feltételezik, hogy a vektorelemek függetlenek
* Mostantól csak regiszter-regiszter
* Miért szerencsések a vektor memóriaműveletek?
  + Vektor beolvasás: fix (nagy) számú memóriaművelet
  + Kiadom az egyik elem címét
  + Következő ciklusban még nem jön meg, de attól még kiadhatom egy másik elem címét → pipeline-szerű megoldás
* Vektorműveletek gyorsítása: Műveleti egységek többszörözésével
* Mély adat-pipeline alkalmazásával: Ez egy olyan pipeline, melyben nincs egymásrahatás!

Strip-mining

* Támogatott vektor méret hardveresen kötött: MVL (Maximum Vector Length)
* Ritkán kell pont ekkora vektor
* Ha kisebbel számolunk:
  + VLR beállítása (Vector Length Register) Kisebb VLR → kisebb futási idő
* Ha nagyobbal számolunk: Vektorunk MVL méretű darabokra szeletelése
  + Minden szeletre a művelet végrehajtása → ez a Strip-mining

Tömbprocesszorok

* Elemei:
  + 1 db vezérlőegység
  + Sok-sok feldolgozóegység
  + Lokális memóriával
  + Többiek felé összeköttetéssel
* Működése: Vezérlőegység minden ciklusban kiad egy utasítást
  + Üzenetszórással, megkapja minden feldolgozóegység
  + Minden feldolgozóegység végrehajtja a saját adatain (maszkregiszter van, aki nincs engedélyezve, az nem csinál semmit)
* Hatékonysági problémák, ha sok a feldolgozóegység:
  + Üzenetszórással mennek ki az utasítások: Sokáig tart, mire mindenkihez elér, hosszú vezetékek: fogyasztás, melegedés

Szisztolikus tömbprocesszorok

* Olyan speciális tömbprocesszorok, melyekben:
  + Minden feldolgozóegység csak a közvetlen szomszédokkal van összekötve
  + Feldolgozóegységek közötti kommunikációt közös órajel hajtja
  + Nincs központi vezérlőegység: minden ciklusban mindenki ugyanazt a műveletet hajtja végre
  + A tömb csak a széleken kapcsolódik a külvilághoz
* Működés:
  + Minden órajelciklusban mindenki
  + Elveszi a szomszédoktól a bemenetét
  + Végrehajtja a műveletet (mindenki ugyanazt)
  + Továbbítja a szomszédainak az eredményt
* Kedvező tulajdonságok:
  + Rendkívül rövid jelutak
  + Kis fogyasztás
  + Magas órajel
  + Gyártástechnológiai előnyök
  + Ismétlődő, szabályos mintázatot hatékonyan lehet gyártani
  + Könnyen bővíthető: csak a széleihez kell tenni pár új feldolgozóegységet
* Kedvezőtlen tulajdonságok:
  + Globális órajelhálózat
  + Nagy rendszerekben egyesekhez hamarabb ér az órajel, mint másokhoz → kavarodást okozhat

Hullámfront tömbprocesszorok:

* Mint a szisztolikus, de:
  + Teljesen aszinkron
  + Nincs órajel
  + Adatfolyam alapú: Egy feldolgozóegység akkor végzi el a műveletet, ha minden input megjött
  + Szomszédok közötti adatcsere összetettebb protokollt igényel
  + Cserébe megússzuk az órajel-hálózatot → még nagyobb sebesség érhető el

Multiprocesszoros rendszerek

* Multi-tasking
  + Több „program” végrehajtása „egyszerre”
  + Már 50 éve is támogatták a profi op. rendszerek
  + Egy processzoron is megy: ekkor a párhuzamos végrehajtásnak csak a látszatát keltik → időosztással
  + Hozzávalók: – 1 db időzítő, ami fix időközönként interruptot generál – 1 db megszakításkezelő rutin, ami taszkot vált

1) Megszakítja az aktuális taszk végrehajtását, állapotát elmenti (regiszterek, verem mutató, stb.)

2) Kiválaszt egy korábban megszakított taszkot

3) Visszatölti annak állapotát

4) Visszatölti az utasításszámlálóját → mostantól őt hajtja végre a processzor

* Igazából több dolgot is kell csinálni a taszkváltáskor:
  + Laptáblát váltani (minden taszknak saját laptáblája lehet)
  + TLB-t üríteni
  + Esetleg utasítás cache-t üríteni
  + Ez is eltart egy darabig, de utána jön még a sok TLB és cache hiba, mire mindkettő újra feltöltődik → Taszkváltás nagyon lassú!
* Multi-thread:
  + Gyakran egy program (taszk) belül maga is párhuzamos részekre, szálakra bomlik
  + Mert a programozó így akarta
  + Thread-ek jellemzői:
    - Osztoznak a taszk memóriáján (lokális változóin is)
    - A szálak külön utasításszámlálóval és regiszterkészlettel rendelkeznek, de ez minden, amiben különböznek
    - Egy taszkon belül a szálak váltása (időosztásos rendszerben): Nem kell laptáblát váltani → nincs TLB és cache ürítés, csak a regisztereket kell váltani → A szálváltás rendkívül gyors!
* Több taszk párhuzamos futásának látszatát egy processzoron időosztással lehet kelteni
* A taszkok közötti váltás nagyon lassú
* Egy taszkon belül a szálak közötti váltás nagyon gyors
* Időosztásos többszálú végrehajtás

1. Finom felbontású multi-threading-et támogat:
   * Minden órajelben szálat vált
   * Mire egy szálra újra sor kerül, jó eséllyel megoldódik a problémája (nem kell külön szünet)
2. Durva felbontású multi-threading-et támogat:
   * Csak akkor vált szálat a processzor, ha az aktuális megakad

* Szimultán többszálú végrehajtás
  + Az időosztásos alternatívája
  + Csak szuperskalár architektúrával működik
  + Szuperskalár esetben a sok műveleti egység gyakran kihasználatlan
  + Mert nincs elég párhuzamosítható utasítás a programban
  + A nem használt műveleti egységeken hajtsuk végre egy másik szál utasításait!
* Amdahl törvénye - teljesítménynövekmény az 1 processzoros rendszerhez képest:
  + S P (N )= 1 (1−P)+ P/N
* Osztályozás:
  + kommunikáció szerint:
    - Osztott memórián alapuló Kommunikáció: egyik taszk beírja, másik kiolvassa
    - Üzenetküldésen alapuló Kommunikáció: a taszkok üzenetet írnak egymásnak, és azt küldözgetik
  + A memóriaműveletek költsége (futási ideje) szerint:
    - UMA (Uniform Memory Access)
      * Minden memóriaművelet azonos ideig tart
      * Mindegy, hogy egy adat hol van: minden processzornak ugyanannyi ideig tart hozzányúlni
    - NUMA (Non-Uniform Memory Access)
      * Fontos, hogy egy adat hol van • Célszerű egy processzor által gyakran használt dolgokat hozzá közel elhelyezni
* Összeköttetés-hálózat:
  + Akár osztott memóriás, akár üzenetküldéses, akár UMA, akár NUMA: a szereplőket össze kell kötni
  + Kit kivel is kell összekötni?
    - Osztott memóriás UMA esetben: Processzorokat a memóriákkal
    - Osztott memóriás NUMA esetben: CPU-ból, cache-ből, memóriából álló modulokat egymással
    - Üzenetküldéses esetben: A résztvevő komplett számítógépeket
  + Mi is zajlik az összeköttetéseken?
    - Osztott memóriás esetben: memóriakérések / válaszok
    - Üzenetalapú esetben: üzenetek
* Direkt összeköttetés-hálózatok:
  + A hálózat csomópontjai a multiprocesszoros rendszerek csomópontjai, maguk alkotják a hálózatot
* Indirekt összeköttetés-hálózatok:
  + A multiprocesszoros rendszer csomópontjai nincsenek egymással közvetlen kapcsolatban, csak a hálózattal