Rendszertechnikai megoldások a teljesítőképesség növelésére

CISC és RISC szervezés

Cél: teljesítőképesség növelése
Az utasításkészlet szempontjából vizsgálva, egy program végrehajtási ideje:
P=I⍰C⍰T, ahol:
I: a program végrehajtott utasításainak a száma
C: az utasításonkénti órajelciklusok átlagos száma
T: az órajelciklusok hossza
A végrehajtási idő csökkentésére adott technológia mellett két út is van:

CISC szervezés

Egyre összetettebb utasítások alkalmazása. Cél minél több funkciót hardverrel megvalósítani. Tipikusan mikroprogramozott vezérléssel.
Ekkor I csökken, C nő, T is nőhet (az egyszerű utasításoké is).

RISC szervezés

Kis számú, egyszerű utasítás. Load-store architektúra (műveletvégzés csak regisztereken), huzalozott vezérlés.
Ekkor I nő, de C csökken és T is csökkenhet.
Hardver helyett szoftver! Jól optimalizáló fordítóprogram szükséges. Nagy számú regiszter kell.
Lehetőségek: egyforma hosszúságú (végrehajtási idejű) utasítások: a pipeline jó lesz.
Kevesebb hely kell a lapkán, lehetőség van MMU, cache stb. integrálására!
(A korszerű processzorban mindkét megoldás – CISC és RISC – előnyeit ötvözik.)

Szuperskalár architektúra

A szóhossz növekedése miatt egy memóriaszóban több utasítás fér el. Ezek lehívása egy gépi ciklus alatt megtörténik. Ezek egyidejű végrehajtásához a processzor bizonyos funkcionális elemeit (például: fixpontos ALU, lebegőpontos ALU) többszörözik. Az utasításlehívás és dekódolás után – amennyiben a utasítások egymástól függetlenek és a szükséges erőforrások rendelkezésre állnak – a *dispatch*er egység egyidejűleg több utasítás végrehajtását is elindíthatja a megfelelő erőforrásokhoz irányítva őket. Ez *MIMD* (Multiple Instruction Multiple Data) működés!
A hardver tervezésekor el kell dönteni, hogy melyik erőforrásból mennyit építsenek be.
A fordítóprogram a program utasításait – amennyiben azok explicite és implicite függetlenek – a processzor architektúrájának ismeretében átrendezheti annak érdekében, hogy az egy utasításszóba kerülő utasítások egyidejűleg végrehajthatók legyenek.
(Az utasítás-egymásrahatások problémájával a pipeline tárgyalásánál foglalkozunk.)

VLIW: Very Long Istruction Word

A CPU-ban szuperskalár architektúrához hasonlóan a processzorban itt is többszörözik a funkcionális egységeket, de azzal ellentétben itt NEM a CPU valamely egységének a feladata, hogy eldöntse, mit lehet párhuzamosan végrehajtani, hanem ezt a feladatot a fordítóprogram végzi el. Az utasításszavak azért hosszúak, mert minden utasításban több funkcionális egységet vezérelnek.

Éppen ezért egyszerűbb hardvert és bonyolultabb fordítóprogramot igényel mint a szuperskalár architektúra.

Társprocesszor

Egy számítógép teljesítőképessége növelhető úgy, hogy az általános célú processzor mellet egy másik, speciális feladatokra (például bonyolultabb aritmetikai műveletek, I/O műveletek) tervezett processzort alkalmazunk: *társprocesszor*. (Például az Intel processzorcsaládban aritmetikai műveletekre: 80287, 80387) Ez NEM multiprocesszoros rendszer, az utasításlehívást és a tárolóhoz való összes hozzáférést (de legalábbis a címszámítást) az általános célú processzor végzi.
A társprocesszor haszna az, hogy bizonyos speciális feladatokat gyorsabban képes elvégezni. Általában egyszerre csak az egyik processzor működik, kivétel, ha a társprocesszor működése alatt az általános célú processzor további, kizárólag neki szóló utasításokat hajt végre, de ilyenkor is figyelnie kell a társprocesszor műveletének befejeződésére (például eredmény tárolása). Társprocesszor hiányában a programban a neki szóló műveleteket az általános célú processzor egy megfelelő szoftveres megszakítás kiszolgálásaként a saját utasításkészletével megvalósított algoritmussal (természetesen több lépésben) oldja meg.

Harvard architektúra

Ebben az architektúrában a program utasításait és az adatokat két külön memóriában tároljuk. Ez elvi eltérés a klasszikus Neumann architektúrához képest! (A Neumann architektúra nem tesz különbséget az utasítások és az adatok között, megkülönböztetésük az algoritmusba beépített értelmezés eredménye!) Meggátolja önmódosító programok írást. A teljesítőképességet úgy képes növelni, hogy az i. utasítás végrehajtása során az adat írása vagy olvasása átlapolódik az (i+1). utasítás lehívásával.

Utasítás pipeline

Egy utasítás feldolgozásának különböző fázisai vannak, például utasításlehívás, dekódolás, végrehajtás. Egy processzor teljesítőképességét növelhetjük, ha egy utasítássorozat egymást követő utasításainak végrehajtási fázisait egymással átlapoljuk: *pipeline*. Szemléltető példa: baromfi feldolgozás futószalagon. Pipeline: *MISD* (Multiple Instruction Single Data) utasítás-szintű párhuzamosítás

Utasítás egymásrahatások

Az átlapolódó utasítások között előfordulhat *utasítás-egymásrahatás*. Ezeket 3 típusba sorolhatjuk:

Feldolgozási egymásrahatás

Feldolgozási egymásrahatás akkor fordul elő, ha két utasítás végrehajtása ugyanazt azerőforrást igényli. Ilyenkor a sorrendben második utasításnak meg kell várnia az első végrehajtását. A probléma kezelésére az erőforrásokat (funkcionális egységeket) többszörözik.

Procedurális egymásrahatás

*Procedurális egymásrahatás*ról beszélünk, ha egy feltételes ugró utasítás miatt nem tudjuk, hogy a program melyik ágon fog folytatódni. Ilyenkor elvileg megtehető az, hogy mindkét ágról elkezdünk utasításokat felhozni, de ez meglehetősen erőforrás igényes, ráadásul újabb elágazás után egyre több ággal kellene foglalkozni. A gyakorlatban ilyenkor vagy várunk vagy valamilyen módszerrel megpróbáljuk megjósolni, hogy merre folytatódik a végrehajtás, és arról az ágról folytatjuk az utasítások felhozását, de megjelöljük őket, és ha a jóslás téves volt, akkor eldobjuk az így felhozott utasításokat.

Adat egymásrahatás

Adat egymásrahatás tipikus példája, ha egy utasítás eredményét egy rá következő utasítás felhasználja. Ezt észlelni kell és bár a rákövetkező utasítás felhozása és dekódolása megtörténhet, a végrehajtáshoz meg kell várni az előző utasítás végrehajtásának befejezését. Különösen veszélyes, ha egy utasítás átírja (valamelyik) rá következő utasítást, aminek a felhozása már megtörtént. Ezt is észlelni és kezelni kell! (Természetesen az ilyen programozási stílus messziről kerülendő!)
A pipelineban a felhozott utasításokkal együtt visszük az utasításhoz tartozó programszámláló értékét, erre például a relatív ugrások kezelésénél van szükség vagy a fent említett program önmódosítás észlelésénél van szükség.

Vektorprocesszorok

A vektorprocesszorok vektor típusú adatokon dolgoznak, ilyenek elemei között végeznek azonos műveleteket. Míg az utasítás pipeline az egymást követő utasítások végrehajtásának részfázisait lapolja át, a vektorprocesszorok *adatpipeline*t használnak: itt az egymást követő vektorelemek kezelése lapolódik át. A vektorprocesszorok *SIMD* (Single Instruction Multiple Data) szervezésűek, így ráadásul két *n* elemű vektor összeadásakor az összeadást utasítást csak egyszer kell felhozni, míg hagyományos skaláris feldolgozást végző processzor esetén *n*-szer.

Tömbprocesszorok

A vektorprocesszorokhoz hasonlóan a tömbprocesszorok is SIMD szervezésűek, de a felépítésük és a működésük teljesen más.
Az utasításlehívást a tömbvezérlő egység végzi, és minden *feldolgozó és tároló egység*nek (FET) ugyanazt az utasítást küldi el, tehát a tömbprocesszor összes feldolgozó egysége ugyanazt az utasítást hajtja végre különböző adatokon. Lehetőség van feltételes utasításokra, amikben a helyileg tárolt adatok befolyásolják a feltétel kiértékelését, így lehetséges az, hogy egy utasítást némelyik FET végrehajt, másik pedig kihagy. A FET-ek a szomszédaikkal (valamilyen topológia szerint) össze vannak kötve így adatcserére képesek.

Irodalom:

- Németh Gábor, Horváth L.: Számítógép architektúrák, 2. kiadás, Akadémiai Kiadó, 1993.
- Andrew S. Tanenbaum: Számítógép-architektúrák, 2. átdolgozott, bővített kiadás, Panem Kft., 2006.