

Horváth Gábor

Számítógép Architektúrák

2021. március 10.

Tartalomjegyzék

I. rész

A vezérlésáramlásos információfeldolgozási modell

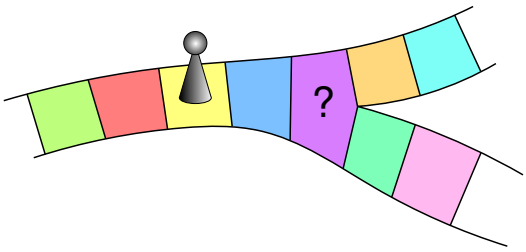
1. fejezet

Információfeldolgozási modellek

A számítógépek működését az ötvenes évektől kezdve, mintegy 30 éven át szinte kizárólag a Neumann architektúra határozta meg, amely a memóriában tárolt utasítások a programozó által megadott sorrendben történő végrehajtásán alapul. A 70-es, 80-as években több, ettől a működési modelltől lényegesen eltérő modellt definiáltak. Ezek a számítási-, más néven információfeldolgozási modellek határozzák meg, hogy egy számítógép milyen módon, milyen elven oldja meg a számítási/információfeldolgozási feladatokat.

1.1. Vezérlésáramlásos modell

Az utasítások közötti adatcserére, az operandusok, ill. a műveletek eredményeinek tárolására közös használatú memória szolgál (közös használatú abban az értelemben, hogy tartalmát minden utasítás képes írni/olvasni). Az utasítások egymásutániságát a vezérlési szál határozza meg. A vezérlési szál mentén a *vezérlő token* utasításról utasításra halad, és az érintett utasításokat sorban egymás után végrehajtja (1.1. ábra). Egy utasítás végrehajtásakor először ki kell olvasni az utasítás operandusait a közös memóriából, majd elvégezhető az utasítás által definiált művelet, végül az eredményt vissza kell írni a közös memóriába. A vezérlő token nem csak sorban egymás után tudja végrehajtani az utasításokat, bizonyos utasításokkal a vezérlő token explicit módon is mozgatható (goto, szubrutin hívás, stb.).

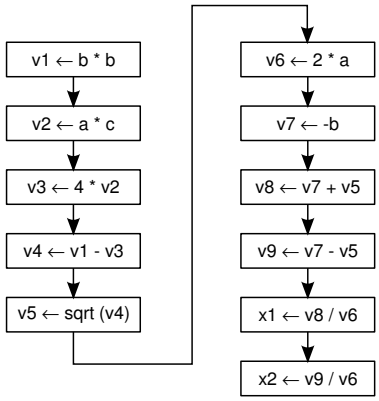


1.1. ábra. A vezérlő token

A vezérlésáramlásos modellben a program leírására a folyamatábra használatos (a magasabb szintű vezérlésáramlásos programozási nyelvekben, mint a C, C++, Java, stb. is tulajdonképpen a program egy folyamatábrának feleltethető meg). Példaként nézzük egy egyszerű, a másodfokú egyenlet gyökeinek kiszámítását végző program folyamatábráját (a valós gyökök létezésének vizsgálata nélkül, 1.2. ábra). Mint látható, ez egy elég egyszerű folyamatábra, nincs benne elágazás sem (a többi információfeldolgozási modellel való összehasonlításhoz ez éppen megfelel).

A példában a program v1...v9 változói képviselik az utasítások közötti adatcserére használatos közös memóriát. A folyamatábra pontosan előírja a program utasításainak végrehajtási sorrendjét, egy klasszikus vezérlésáramlásos számítógépnek nincs is lehetősége a sorrend megváltoztatására. Ez a vezérlésáramlásos információfeldolgozási modell legnagyobb gyengéje: *nem képes a párhuzamosság automatikus felderítésére és kiaknázására*! Hogy hol van a példában kiaknázható párhuzamosság? Például ezeken a pontokon:

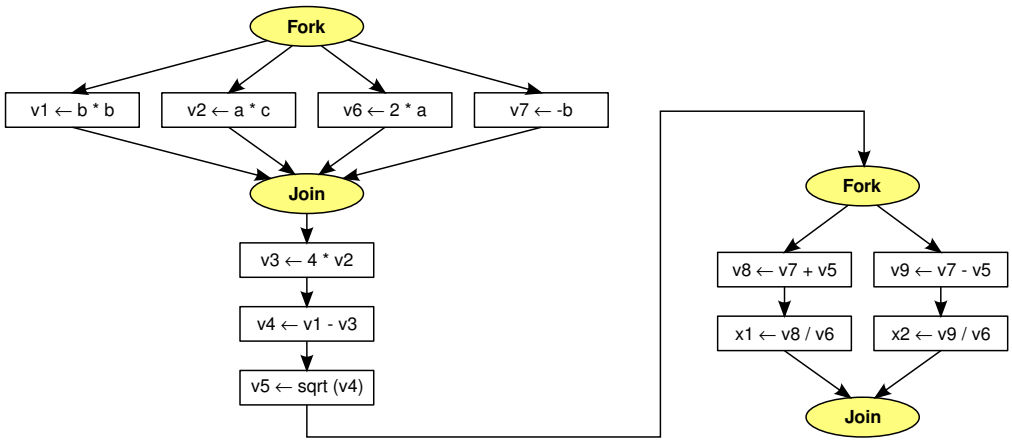
- v1, v2, v6, v7 egymással párhuzamosan számolható lenne,



1.2. ábra. Folyamatábra másodfokú egyenlet gyökeinek meghatározására

- v8 és v9 úgyszintén,
- x1 és x2 is párhuzamosan számolható.

Bizonyos kiterjesztésekkel, pl. a Fork/Join primitívek bevezetésével lehetővé tehető a párhuzamosság leírása. A Fork művelet lényegében többszörözi a vezérlő tokent, a különböző vezérlő tokenek által kijelölt utasítások pedig párhuzamosan hajthatók végre. A Join művelet összevár több vezérlő tokent, és a több vezérlő tokenből ismét egyet csinál. A Fork/Join primitívekkel le lehet ugyan írni a párhuzamosságot, de ezt a programozónak, a folyamatábra előállítójának kell megtennie. Az 1.3. ábra mutatja be a Fork/Join segítségével párhuzamossá tett másodfokú egyenlet megoldót.



1.3. ábra. Folyamatábra másodfokú egyenlet gyökeinek meghatározására, Fork/Join primitívekkel

A vezérlésáramlásos modell tehát szekvenciális programok és egyprocesszoros rendszerek esetén kiváló, ezt a modellt valósítja meg (többek között) a Neumann-architektúra is, amely a ma használatos számítógépek túlnyomó többségében a programozói interfész alapja. Ezekben a számítógépekben az utasításszámláló képviseli a vezérlő tokent, az utasítások közötti adatcserére szolgáló közös memóriát pedig a regiszterek, illetve a rendszermemória képviseli.

1.2. Adatáramlásos modell

Az adatáramlásos modellben a program utasításainak végrehajtási sorrendjét nem a programozó, hanem az utasítások operandusainak rendelkezésre állása határozza meg. Ha egy utasítás minden operandusa rendelkezésre áll, rögtön végrehajthatóvá válik. Mivel a végrehajtás eredménye esetleg egy vagy több másik utasítás operandusa,

a végrehajtás után az eredmény értékét az arra váró utasításoknak át kell adni. Tehát a vezérlésáramlásos modellhez képest a legfőbb különbségek:

- Nincs vezérlő token: egy utasítást akkor hajtunk végre, ha minden operandusa megvan.
- Nincs osztott memória: a program futása során a részeredményeket az utasítások közvetlenül adják át egymásnak.

Az adatáramlásos modellben a program az utasítások közötti függőségeket leíró precedenciagráf. A gráf csúcspontjai az utasítások, az i. és j. csúcs közötti irányított él pedig azt jelenti, hogy a j. utasítás egyik operandusa az i. utasítás eredménye.

Az 1.4. ábrán láthatjuk a másodfokú egyenletet megoldó programot, adatáramlásos modellel megvalósítva. Az utasításokat téglalappal jelöltük, az utasítások operandusait pedig a téglalapba helyezett körökkel. Az üres kör jelenti az operandus hiányát, a korongot tartalmazó kör pedig az operandus meglétét (a korong tulajdonképpen az operandus értéke). A végrehajtást az adatok (operandusok) áramlása vezérli. Az első lépésben a v1, v7, v2 és v6 utasítások rendelkeznek az összes szükséges operandussal, ezek így mind végrehajthatók, akár párhuzamosan is, ha van megfelelő számú feldolgozóegység. A v1 végrehajtása után v4-nek adja át az eredményt, v7 a v8-nak és a v9-nek, v2 a v3-nak, v6 pedig az x1-nek és az x2-nek. A továbbiakban feltételezzük, hogy minden művelet végrehajtása ugyanannyi ideig tart. A következő körben csak a v4 lesz végrehajtható, majd a v5, aztán párhuzamosan a v8 és v9, majd szintén párhuzamosan az x1 és x2.

Ahogy a példán is látható, a párhuzamosság leírása a modell természetéből fakad, nem a programozó dolga.

A 70-es évek végén, 80-as évek elején az adatáramlásos modellt a jövő ígéretes információfeldolgozási modelljének tekintették. Több neves egyetem is fejlesztett olyan számítógépet, melyet az imént látott módon, adatáramlásos elven, precedenciagráf megadásával lehetett programozni, és amely képes volt a leírt eljárásban rejlő párhuzamosítási lehetőségek kiaknázására. Ilyen próbálkozások voltak pl. az MIT Data-Flow Computer és a Manchester Data-Flow Machine is (de pl. a Texas Instruments is fejlesztett ilyen számítógépet). Végül az adatáramlásos programozási modell és a színtisztán adatáramlásos elven működő számítógépek ugyan nem terjedtek el, de az adatáramlásos információfeldolgozási elv az alábbi két területen (többek között) fontos szerepet játszik:

- A táblázatkezelő szoftverek adatáramlásos elven működnek. Ha megváltoztatunk egy adatot, akkor minden arra hivatkozó cella is frissül, majd a rájuk hivatkozó cellák, és így tovább.
- A sorrenden kívüli végrehajtást (out-of-order) támogató processzorok mind-mind adatáramlásos elven, egy precedenciagráf karbantartásával derítik fel a program (egyébként vezérlésáramlásos modellnek szánt) utasításai között a párhuzamosítási lehetőségeket. Ezek a processzorok több utasítást hívnak le, és a közöttük lévő függőségek elemzésével meg tudják határozni, hogy mely utasítások végrehajtása mikor kezdődhet el. Mivel több feldolgozóegységgel is rendelkeznek, egyidejűleg több utasítás végrehajtása is lehetséges. (Vigyázat! ennek semmi köze a többmagos processzorokhoz!)

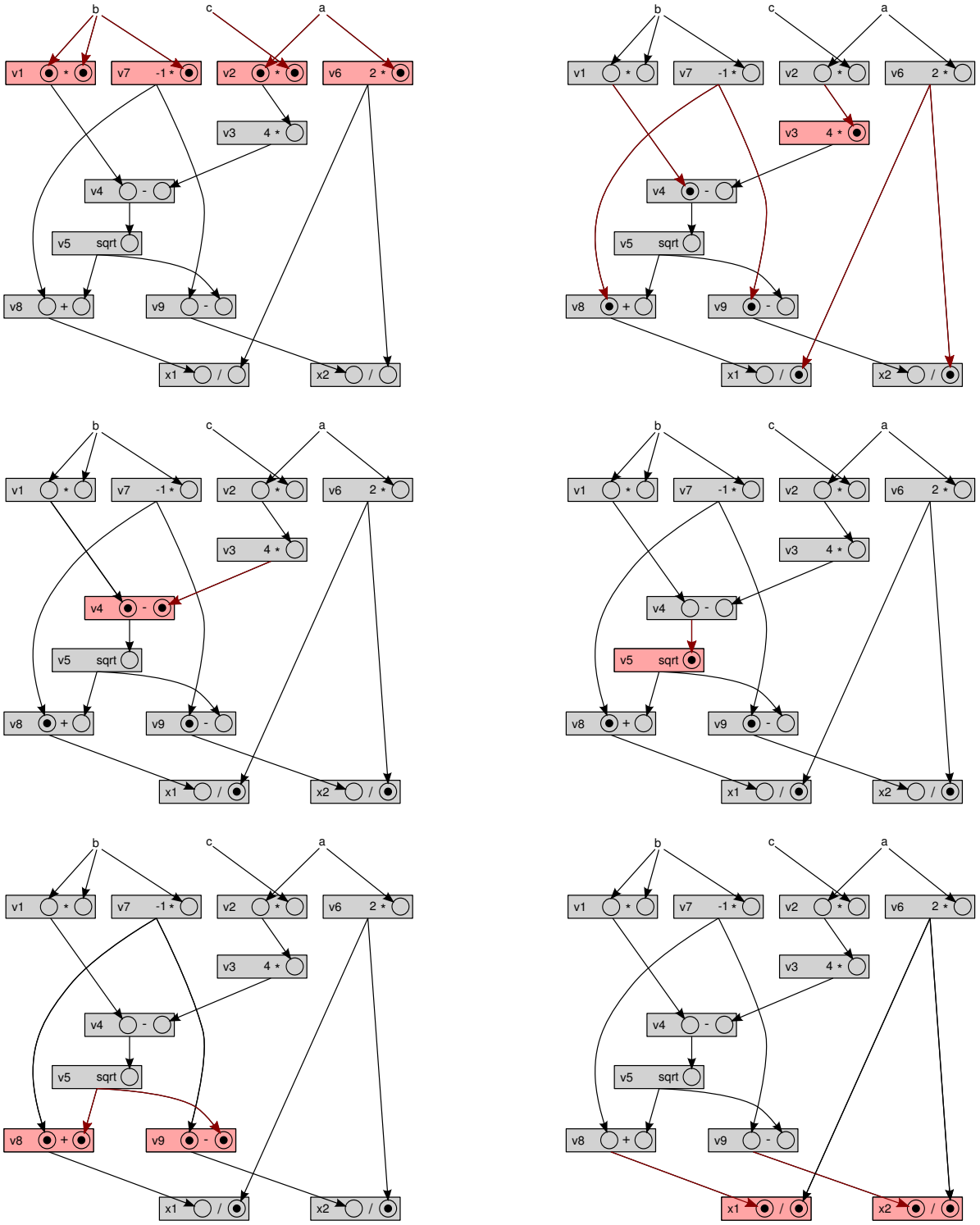
1.3. Igényvezérelt modell

Az igényvezérelt információfeldolgozási modellben egy utasítás akkor hajtódik végre, ha annak eredményére szükség (igény) van. Emlékeztetőül, az eddigi vett információfeldolgozási modellek között alapvető eltérés van az utasítások végrehajtásának idejét illetően:

- vezérlésáramlásos modellben: amikor odaér a vezérlő token (a programozó szabja meg),
- adatáramlásos modellben: amikor minden operandusa rendelkezésre áll (szorgos kiértékelés, eager evaluation),
- igényvezérelt modellben: amikor az utasítás eredményére szükség van (lusta kiértékelés, lazy evaluation).

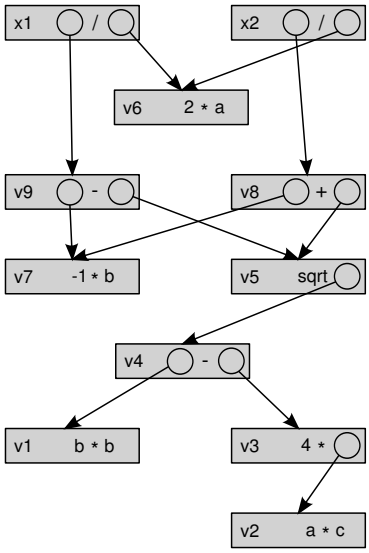
Ha az adatáramlásos modellben a program végrehajtása ”bottom-up” szervezésű (az inputtól halad az eredményig), akkor az igényvezérelt modellben ”top-down” programvégrehajtás történik. Első lépésként a közvetlenül az eredményhez vezető műveletet kell tekinteni. Ennek a műveletnek lehetnek operandusai, melyek más műveletek eredményei. Ezután azt kell megnézni, hogy az eredményhez vezető utasítás operandusait előállító műveletek végrehajtásához mire van szükség, stb.

A másodfokú egyenlet példáját az 1.5. ábrán láthatjuk, igényvezérelt megközelítésben. A számítás menetét az 1.6. ábra ábrázolja. A közvetlenül az eredményhez vezető utasítás az x1. Ennek operandusait a v9 és a v6



1.4. ábra. Másodfokú egyenlet megoldó program működése adatáramlásos modellben (balról-jobbra, felülről lefelé)

adja, tehát kezdeményezni (igényelni) kell a v9 és a v6 kiszámolását. A v6 egy elemi művelet, mely közvetlenül kiszámítható, ezzel az x1 egyik operandusa már rendelkezésre is áll. A v9 kiszámításához azonban a v7 és a v5 eredményére is szükség van, tehát kezdeményezni ("igényelni") kell ezek kiértékelését. A v7 egy elemi művelet,



1.5. ábra. Másodfokú egyenlet megoldó program, igényvezérelt modellel

de a v5-höz a v4 eredményére van szükség, stb. Az ábrán szürke szín jelöli a még ki nem értékelt utasításokat, piros jelzi, hogy az utasítás végrehajtása folyamatban van (operandusra vár), a zöld pedig a kész utasításokat jelöli. Az operandusok meglétét, akár az adatáramlásos példában, itt is fekete korong jelöli.

A működési elvből következik, hogy az igényvezérelt számítási modell is támogatja a párhuzamosítási lehetőségek automatikus felderítését és kiaknázását, hiszen egy utasítás végrehajtásához a szükséges operandusok "igénylése" az azokat előállító utasításoktól párhuzamosan történhet. A példán ez többször is megvalósul, pl. rögtön az első lépésben a v6 kiszámítása és a v9 végrehajtásának kezdeményezése egyszerre történik.

A 80-as években több kísérleti, tisztán igényvezérelt működési elvű számítógépet is építettek. Az igényvezérelt számítógép egy lehetséges megvalósítása nagy számú feldolgozó egységből és az azokat összekötő kommunikációs hálózatból áll. A feladatot (programot) tetszőleges feldolgozóegységnél be lehet adni. A feldolgozóegység a programot kisebb részfeladatokra bontja, azokat a szomszédoknak eljuttatja, és megvárja a tőlük visszaérkező részeredményeket, melyből végül az eredményt elő tudja állítani. A szomszédos csomópontok a nekik eljuttatott részfeladattal ugyanígy járnak el: még kisebb darabokra bontják, amiket az ő szomszédaiknak továbbítanak, és így tovább.

Az igényvezérelt számítógép sem hozott áttörést, maradt a vezérlésáramlásos szervezés egyeduralma. De az igényvezérelt számítási modell sem tűnt el: ez lett a funkcionális programozási nyelvek alapja. Összességében tehát ma a számítástechnikában mindhárom információfeldolgozási modell fontos szerepet játszik:

- A funkcionális programozási nyelvek alapja az igényvezérelt modell,
- a processzorok interfésze, utasításkészlete a vezérlésáramlásos modellen alapul,
- a modern processzorok belső felépítése azonban adatáramlásos modell alapján képes (a programozó számára láthatatlanul) az utasítások párhuzamos végrehajtására.

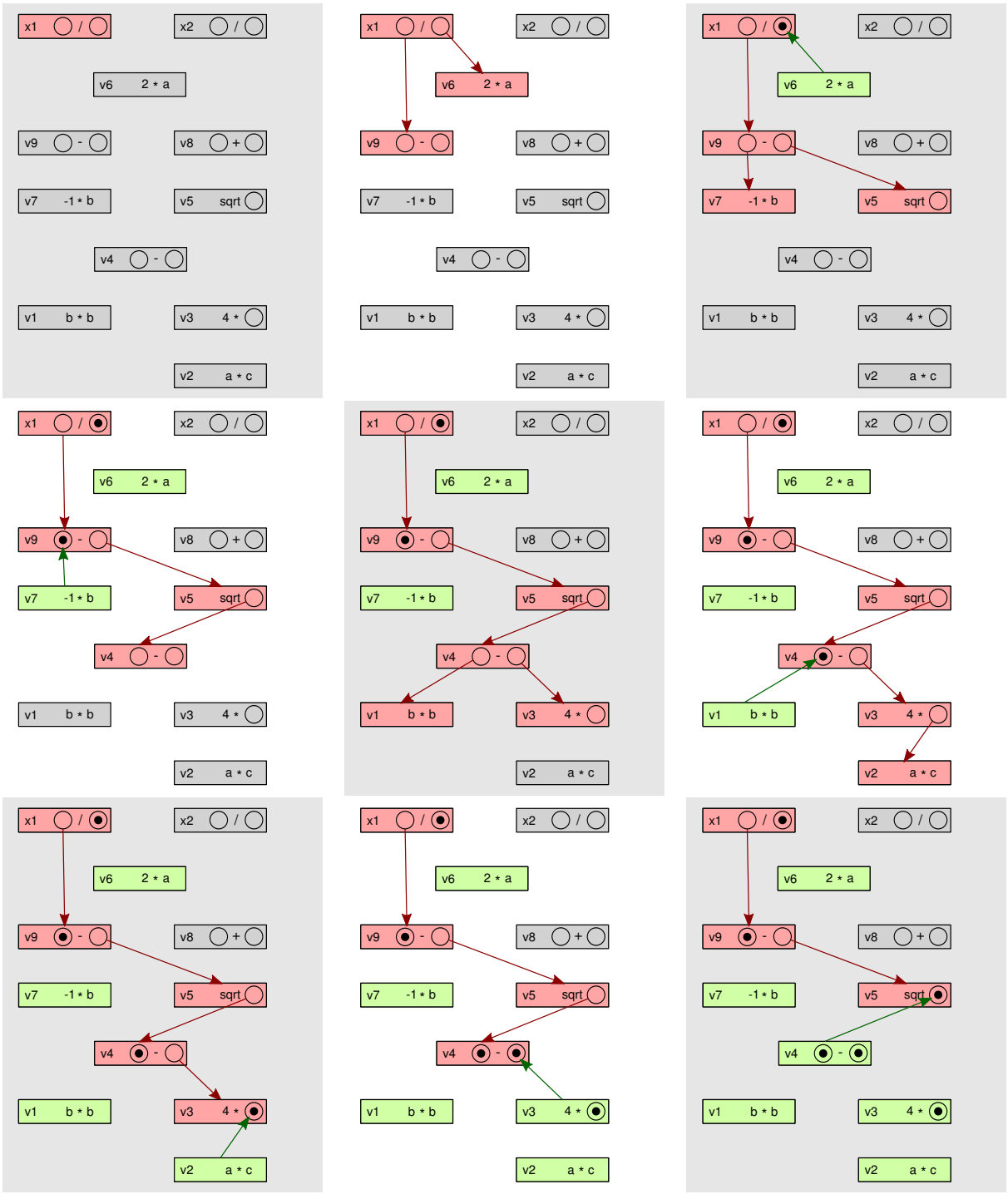
1.4. Esettanulmány

Ebben a fejezetben egy egyszerű algoritmust, a Fibonacci sorozat 100-dik elemének kiszámítását valósítjuk meg vezérlésáramlásos, adatáramlásos és igényvezérelt modellben.

1.4.1. Megoldás vezérlésáramlásos modellel

Mivel a legjobban elterjedt programozási nyelvek mind vezérlésáramlásos elven működnek, valószínűleg nem okoz nehézséget a következő, pszeudó-nyelven megfogalmazott program:

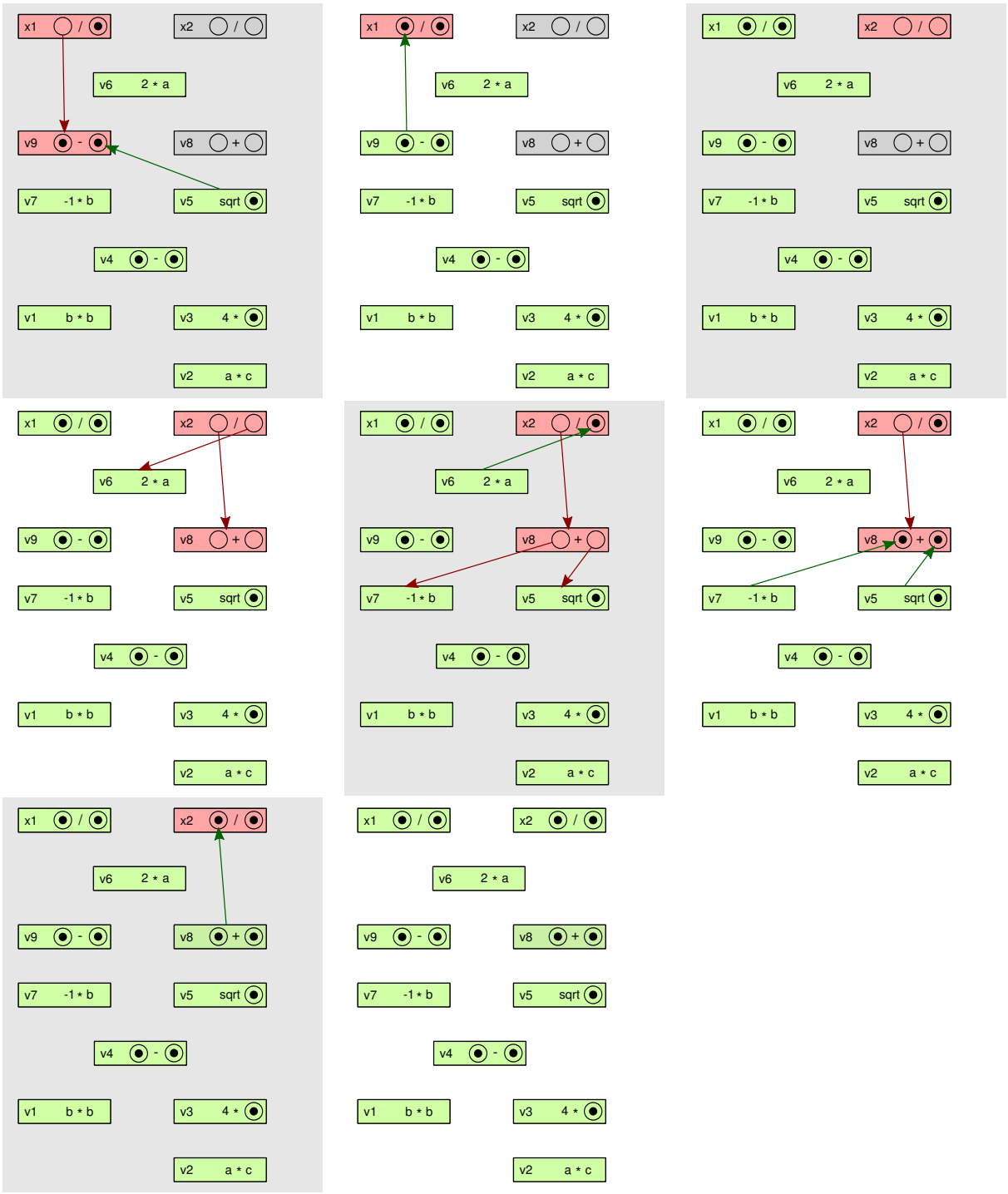
```
(f1,f2) ← (1,1);
```



1.6. ábra. Másodfokú egyenlet megoldó program működése igényvezérelt modellben (balról-jobbra, felülről lefelé), 1/2. rész

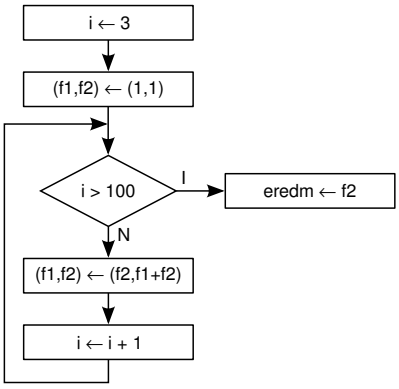
```
for i=3 to 100 do
    (f1,f2) ← (f2, f1+f2);
end
eredm ← f2;
```

Az 1.7. ábrán láthatjuk, hogy ez a program a vezérlésáramlásos modellben milyen folyamatábrának felel meg.



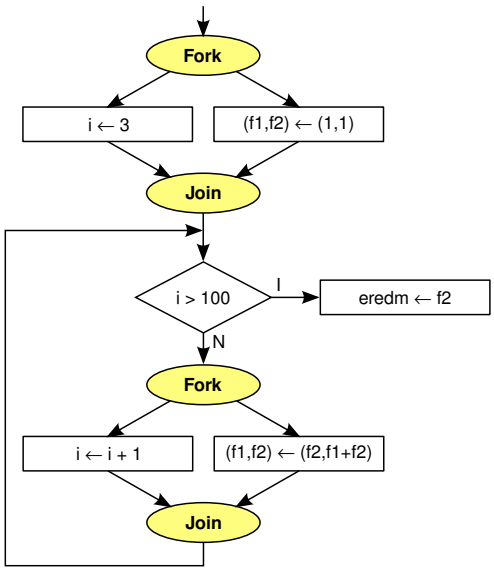
1.6. ábra. Másodfokú egyenlet megoldó program működése igényvezérelt modellben (balról-jobbra, felülről lefelé), 2/2. rész

Mint a modell leírásánál részletesebben kifejtettük, a problémát az jelenti, hogy a folyamatábra, természeténél fogva, az utasítások szekvenciális leírására szolgáló formalizmus. Ha szeretnénk az eljárásban rejlő párhuzamosítási lehetőségeket kiaknázni, akkor azt csak a vezérlő token explicit többszörözésére és összefogására bevezetett Fork és Join primitívekkel tehetjük meg. Ebben az eljárásban első pillanatra nem is látszik párhuzamosítási lehetőség, de jobban megnézve, a ciklusváltozó növelése és az (f1,f2) pár értékadása párhuzamosan végrehajtható. Az így kapott folyamatábrát az 1.8. ábrán láthatjuk. Megjegyezzük, hogy a kereskedelmi forgalomban kapható, elterjedt



1.7. ábra. Fibonacci sorozat számolásának algoritmusa, vezérlésáramlásos modellben

számítógépek (mind a személyi számítógépek, mind a szerverek, mind a munkaállomások) közül sok tartalmaz hardveres támogatást több vezérlő token használatához, de a Fork és Join műveletek annyira költségesek, hogy két értékadás párhuzamos elvégzéséhet nem éri meg alkalmazni őket (mert a Fork és a Join önmagában tovább tart mint a két értékadás együttvéve).



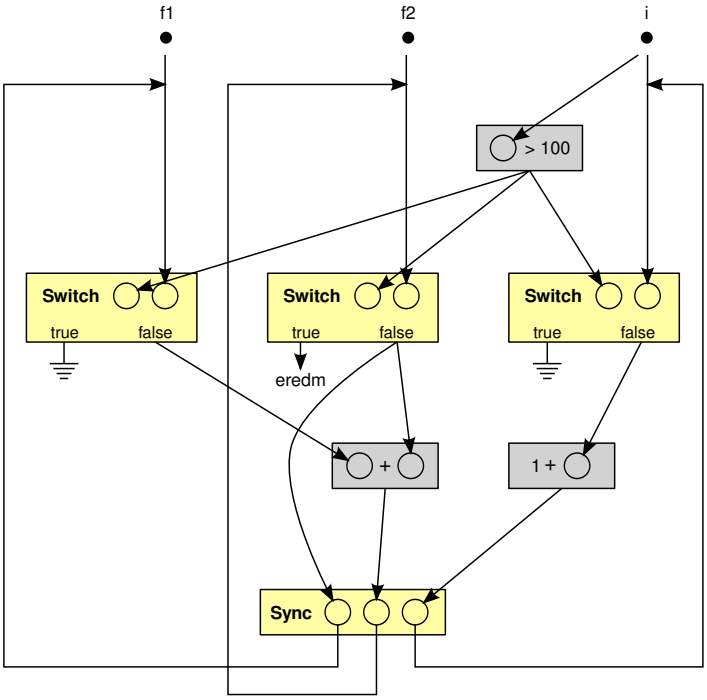
1.8. ábra. Fibonacci sorozat számolásának algoritmusa, vezérlésáramlásos modellben, párhuzamosítva

1.4.2. Megoldás adatáramlásos modellel

Az adatáramlásos modellben a program egy precedenciagráf, az utasítások közötti függőségi viszonyok leírása. Míg a vezérlésáramlásos modell esetén a folyamatábra egy jól elterjedt, egységes jelölésrendszer, addig az adatáramlásos elven történő program leíráshoz sajnos nem áll rendelkezésre ilyen egységes formalizmus.

Az 1.9. ábrán a megoldást egy lehetséges adatfolyam alapú jelölésrendszerben adjuk meg (ez csak egy lehetőség a sok közül). Ebben a jelölésrendszerben szürke téglalappal jelöltük a tényleges műveleteket végző utasításokat (mint a ciklusfeltétel kiértékelése, vagy az összeadás), sárgával pedig a leíró formalizmus részét képező elemeket. A ”switch” elem az első operandusa függvényében vagy a ”true”, vagy a ”false” irányába továbbítja a második operandusát, a ”Sync” pedig az összes operandusát változatlanul, egyidejűleg továbbítja, de csak akkor, ha mind rendelkezésre áll.

A végrehajtás kezdetén az f1-el, és az f2-vel jelölt token (zseton) értékét 1-re állítva, az i token értékét 3-ra állítva dobjuk be a rendszerbe. Ezzel a switch elemek második operandusa megvan, de a switch-ek még nem



1.9. ábra. Fibonacci sorozat számolásának algoritmusa, adatáramlásos modellben

hajthatók végre, hiszen hiányoznak az első operandusok. Így a végrehajtás kezdetekor csak a ciklusfeltételt kiértékelő művelet hajtható végre. Ez a művelet veszi az i tokent, és egy igaz/hamis értékkel ellátott tokent állít elő, melyet eljuttat mindhárom switch-nek. Ezzel mindhárom switch mindkét operandusa rendelkezésre áll, tehát a switch-ek végrehajtódnak (akár párhuzamosan). Mivel a ciklus elején vagyunk, a feltétel hamis lesz, és a switch-ek a második operandusaikat (rendre f1-et, f2-őt, i-t) továbbítják az összeadást végző műveletek felé. Ezek egyike a ciklusváltozót növeli, másika pedig az f1+f2 összeget számolja ki (akár párhuzamosan). Ezután a sync elem bevárja a műveletek eredményét, majd a tokeneket visszateszi a bemenetre, de úgy, hogy f1 helyére az f2 kerüljön, f2 helyére az f1+f2, az i helyére pedig az i+1. Ekkor indulhat a ciklus második iterációja, és így tovább, egészen addig, míg az összehasonlítás igaz nem lesz.

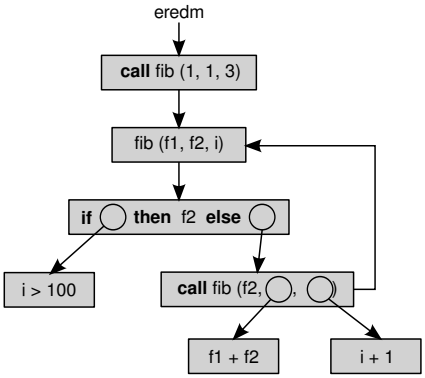
1.4.3. Megoldás igényvezérelt modellel

Az igényvezérelt modellben a programot egyetlen összetett kifejezésként kell megadni, amely végrehajtás közben, a redukciós elvet követve, fokozatosan részkifejezésekre bomlik. A Fibonacci sorozat elemeit kiszámoló program összetett kifejezésként való megadására a legegyszerűbb, ha rekurziót alkalmazunk:

```
eredm = fib (1, 1, 3);
fib (f1, f2, i) = if i > 100 then f2 else fib (f2, f1+f2, i+1);
```

A program a ”fib” függvény rekurzív definíciójából és annak meghívásából áll. Az 1.10. ábra szemlélteti, hogy végrehajtás közben hogyan bomlik részfeladatokra a probléma.

A példában az `eredm` kiszámolásához a `fib (1, 1, 3)`-et kell meghívni, vagyis kezdeményezni kell a `fib (f1, f2, i)` függvény kiértékelését az f1=1, f2=1, i=3 helyen. A függvény egy ”if” feltételes szerkezetből áll, tehát ezen a ponton kell kiértékelni az elágazási feltételt, majd az egyik ágat (ebben a sorrendben, párhuzamosítási lehetőség itt nincs). Ha az ”else” ág került kiválasztásra, akkor a fib függvény rekurzív meghívásához szükséges az argumentumok (az f1+f2 valamint az i+1) előállítása. Mivel ezek értékére ebben a pillanatban merül fel az igény, ezért ezeket most kell kiértékelni, de a sorrend mindegy - az f1+f2 és az i+1 kiszámolása ezért egymással párhuzamosan is történhet. Ha megvannak az argumentumok, akkor megtörténhet a függvény rekurzív hívása, mindaddig, amíg a 100-dik lépéshez nem érünk.



1.10. ábra. Fibonacci sorozat számolásának algoritmusa, igényvezérelt modellben

1.5. Irodalomjegyzék

[1.1] Németh Gábor, BMEVIHIA210 Számítógép Architektúrák előadásfóliák, 2010.

[1.2] Philip C. Treleaven, David R. Brownbridge, Richard P. Hopkins. *Data-Driven and Demand-Driven Computer Architecture.* Computing Surveys, Vol. 14, No 1, March 1982.

1.6. Számonkérés

1.6.1. Kiskérdések

1. Hogyan írható le egy program a vezérlésáramlásos, az adatáramlásos, és az igényvezérelt modellben?
2. Hogyan támogatja a feladatban rejlő párhuzamosítási lehetőségek automatikus felderítését és kiaknázását a vezérlésáramlásos, az adatáramlásos, és az igényvezérelt modell?
3. Mikor hajtható végre egy utasítás a vezérlésáramlásos, az adatáramlásos, és az igényvezérelt modellben?

1.6.2. Nagykérdések

1. Mutassa be és hasonlítsa össze a 3 tanult információfeldolgozási modellt!

2. fejezet

Vezérlésáramlásos architektúrák

2.1. A Neumann-architektúra

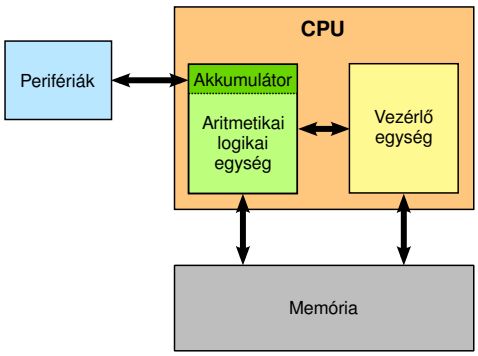
Neumann János 1945-ben, ”First Draft of a Report on the EDVAC” címmel írt tanulmányában fogalmazta meg először azokat az elveket, melyek alapján ma a ”Von-Neumann tárolt programvezérlésű számítógép”-ek működnek. (Megjegyzendő, hogy az elv kidolgozásában Neumann kollégái, Presper Eckert és John Mauchly is fontos szerepet játszottak). Az első megépített, működő Neumann-architektúrájú számítógépet a Princeton egyetemen fejlesztették ki, IAS néven, Neumann személyes részvételével, emiatt sok helyen nem Neumann, hanem Princeton architektúrának ismerik.

2.1.1. A Neumann-architektúra alapelemei

A Neumann-architektúra legnagyobb újítása elődeihez képest, hogy a programot nem mechanikus kapcsolósorok segítségével kell beállítani, hanem az az adatokkal együtt a memóriában van eltárolva. Ezt a megoldást az motiválta, hogy a mechanikus kapcsolókhoz képest így sokkal könnyebben és gyorsabban lehet a programot (az utasítássorozatot) bevinni, megváltoztatni, illetve egymás után több különböző programot lefuttatni. Emellett annak, hogy a program és az adatok is egyazon memóriában vannak elhelyezve, van még egy fontos járulékos hozadéka: lehetővé válik a ”programot generáló programok” készítése, ekkor indult el tehát a fordítóprogramok karrierje is.

Egy Neumann-architektúrájú számítógép 3 fő komponensből áll (2.1. ábra):

- CPU,
- Memória,
- Bemeneti/Kimeneti perifériák.



2.1. ábra. A Neumann-architektúra

A processzor (CPU) tartalmazza egyrészt a vezérlő egységet (control unit), másrészt az aritmetikai-logikai egységet (arithmetic-logic unit, ALU). Az ALU végzi a tényleges számításokat: összead, kivon, szoroz, oszt, logikai

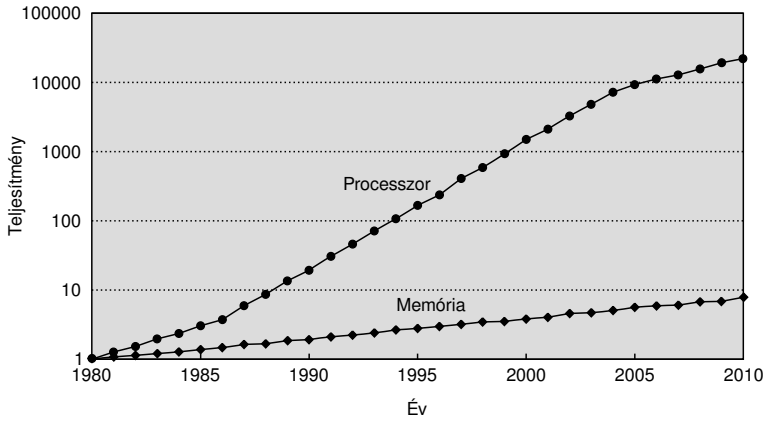
és, vagy műveleteket végez. A vezérlő egység értelmezi a memóriából lehívott utasításokat, meghatározza az utasítások végrehajtási sorrendjét, előállítja az ALU részére a vezérlőjeleket.

A bemeneti/kimeneti perifériák biztosítják a kapcsolatot a számítógép és a külvilág között. Az ALU-ban lévő akkumulátor (operandus ill. eredménytároló) tartalma egy kimeneti perifériára kiírható, vagy egy bemeneti perifériából beolvasható.

A memória adott bitszélességű adategységek tárolója, melyben minden adategység egy egyedi címmel rendelkezik. Már említettük, hogy a memóriában tároljuk az utasításokat és az adatokat is. Azonban ezek a memóriában semmilyen módon nincsenek megkülönböztetve, az, hogy egy memóriabeli objektum utasítás-e, vagy adat, attól függ, hogy a processzor utasítást hív-e le onnan, vagy adatot. Ugyanez igaz az adatokra is: a memóriában nincs letárolva az adatok típusa. Egy memóriatartalom interpretálható egészként, karakterként, vagy lebegőpontos számként egyaránt, attól függően, hogy azon egész, karakter, vagy lebegőpontos utasítások dolgoznak-e.

2.1.2. A Neumann-architektúra szűk keresztmetszete

A memória és a CPU egymástól való eltávolításának az lett a következménye, hogy a CPU és a memória közötti összeköttetés szűk keresztmetszetté vált (von Neumann bottleneck). Ahogy a processzorok mind gyorsabbá váltak, az általuk generált memóriaforgalom (mely egyaránt áll utasításokból és adatokból) is egyre nagyobbá vált. A CPU sebességének növekedési ütemével sem a memóriák sebessége, sem a buszok (összeköttetések) sebessége nem tud lépést tartani. A processzor és a memória sebessége közötti egyre nyíló ollót először memóriarésnek (memory gap) nevezték, az utóbbi években memóriafalnak (memory wall) nevezik (lásd 2.2. ábra). A memóriafal miatt, vagyis hogy a memória sebessége több nagyságrenddel is elmarad a processzor sebességétől, a processzor egyre többször kényszerül megállni, és bevárni a lassú memóriát. A probléma kezelésére ma már általánossá vált a cache memória használata.



2.2. ábra. A processzor és a memória sebességének fejlődése

2.1.3. Önmódosító programok

Mivel a programot alkotó utasítások is ugyanabban a memóriában vannak, mint az adatok, elvileg semmi akadálya annak, hogy a program módosítsa önmagát, vagyis memóriaműveletek segítségével átírja az utasításait. Azokat a programokat, amelyek ezt a technikát alkalmazzák, *önmódosító* programoknak nevezzük. Önmódosító programok révén bizonyos algoritmusokat valamivel rövidebben, kevesebb utasítással le lehet írni, ami egykor, a szűkös memóriakapacitások korában lényeges szempont volt. Mint a következő példából is látható, ez a pár bájtos spórolás a programot rendkívül nehezen olvashatóvá, karbantarthatatlanná teszi. Így ez az egykor elegánsnak számító technika ma már kifejezetten kerülendő, nem kívánatos programozói stílus. Sőt, a korszerű processzorok lehetővé teszik az operációs rendszer számára, hogy memóriavédelmet alkalmazzon, például úgy, hogy a futtatható kódot tartalmazó memóriadarabokat ”csak olvasható” jogosultsággal látja el. Ez egyben a futó programok kódjának rosszindulatú manipulációját (bizonyos vírusok tevékenységét) is lehetetlenné teszi.

Kizárólag demonstrációként (véletlenül sem követendő példaként!) bemutatunk egy önmódosító kódra épülő algoritmust a Fibonacci sorozat N. elemének kiszámítására:

```
main: R4 ← N
      R9 ← opcode of (R2 ← R2 + 0)
      R8 ← 1
      R2 ← 1
loop: JUMP halt IF R8 == R4
      R8 ← R8 + 1
      R10 ← R9 + R2
key:  R2 ← R2 + 0
      MEM[key] ← R10
      JUMP loop
halt:
```

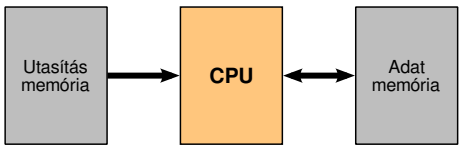
A példában az R8 a ciklusváltozó, ami 1-től N-ig pörgeti a ”loop” címke és a ”JUMP loop” közötti ciklusmagot. A ciklusmag első utasítása a ciklusváltozó növelése. Az ezt követő 3 sor tartalmazza legfontosabb részt. Az eredményt az R2 regiszterben gyűjtjük. Az R8-adik kör elején az R2 tárolja a fib(R8+1) értékét. Az önmódosító kód a ”key” címkével ellátott sorban lévő utasítás cserélgetésével valósul meg. Feltesszük, hogy az összeadó utasítás belefér egy regiszterbe, és hogy az utasítások úgy vannak kódolva, hogy az opcode(R1 ← R2 + skalár) = opcode(R1 ← R2 + 0) + skalár (itt most az opcode(”utasítás”) azt adja meg, hogy az ”utasítás” a memóriában milyen számként jelenik meg – hiszen a memóriában minden objektum egy szám). A ”key” címkével ellátott utasítás új, a ciklusmag végén beállítandó értéke az R10 regiszter tartalma lesz. Még egyszer: az R8-adik kör elején az R2 fogja tárolja a fib(R8+1) értékét. Az R8-adik körben az R10 tartalma az R2 ← R2 + fib(R8+1) utasítás opcode-ja lesz. Mivel az utasítás módosítása (R10 memóriába írása) csak a ciklusmag végén történik, a ”key” címke alatt még az előző körben odaírt utasítás szerepel, tehát az R2 ← fib(R8+1) + fib(R8), vagyis a ciklus végére R2 pont a fib(R8+2) értéke lesz. A ciklus újabb és újabb iterációi így a Fibonacci sorozat újabb és újabb elemeit számolják ki.

2.2. A Harvard architektúra

A Harvard architektúra a ”Harvard Mark I” számítógép után kapta a nevét. Ez a számítógép lyukkártyáról olvasta az utasításokat, az adatokat pedig egy relé kapcsolókból felépülő tárban tárolta. A Harvard architektúra annyiban különbözik a Neumann-architektúrától, hogy a program (az utasítások) és az adatok tárolása fizikailag elkülönül egymástól.

2.2.1. A Harvard architektúra felépítése és tulajdonságai

A Harvard architektúrájú processzorok két különálló interfésszel rendelkeznek az utasításmemória és az adatmemória felé (2.3. ábra).



2.3. ábra. A Harvard architektúra

A processzornak a két memória felé akár teljesen különböző interfésze is lehet: különbözhet az adategység (szóhossz), és az időzítések is. A két memória akár különböző technológiájú is lehet: pl. az utasításmemória lehet csak olvasható memória, pl. ROM, vagy EPROM, míg az adatmemória lehet DRAM.

A Neumann-architektúrában az utasítások és az adatok is ugyanabban a memóriában vannak, ezért adatot és utasítást egyidejűleg nem lehet olvasni. Előbb az utasítást kell lehívni a memóriából, majd ezt követően az utasítás operandusait. Ezzel szemben a Harvard architektúrának két memória interfésze van, tehát két memóriaműveletet is képes egyidejűleg végezni, egyet az utasításmemória, egyet pedig az adatmemória felé. Például, amíg a processzor lehívja az adatmemóriából az aktuális utasítás operandusait, addig, ezzel időben átlapoltan, már töltheti is le az utasításmemóriából a következő utasítást. Ezzel a kialakítással tehát némi gyorsulást lehet elérni.

Harvard architektúrát alkalmaz számos mikrokontroller, valamint jelfeldolgozó processzor (DSP).

2.2.2. Módosított Harvard architektúra

A módosított Harvard architektúra annyiban különbözik a klasszikustól, hogy a program képes írni és olvasni az utasítás memóriát is. Természetesen ettől ez még nem lett a Neumann-architektúrához hasonlatos, mert utasítást még a módosított Harvard architektúrában is csak az utasításmemóriából lehet lehívni. Az utasításmemóriához való direkt hozzáféréshez általában külön erre a célra bevezetett utasítások állnak rendelkezésre (pl. LPM (Load Program Memory) és SPM (Store Program Memory) utasítások az Atmel AVR processzorokban). Ezáltal a módosított Harvard architektúra lehetővé teszi, hogy a diszkről töltsünk be programot, és önmódosító programot is lehet futtatni.

A módosított Harvard architektúra a mikrokontrollerek közkedvelt architektúrája (pl. PIC, Atmel AVR).

3. fejezet

Utasításkészlet architektúrák

3.1. Utasításkészletek

A gyártók minden egyes processzorhoz megadnak egy *programozási felület*et (utasításkészlet architektúrát, Instruction Set Architecture, ISA). Ez a következőket foglalja magában:

- Az *utasítások*at (instructions), vagyis hogy az adott programozási felületet támogató processzorok milyen utasításokat ismernek.
- *A beépített adattípusok*at (native data types), azaz hogy az utasítások milyen adattípusokon képesek műveleteket végezni (pl. 8/16/32 bites előjeles/előjel nélküli egész, stb.)
- A programozó számára látható *regiszterek*et (registers). A processzorban a regiszterek tárolják az utasítások operandusait és eredményeit (többek között – vannak vezérlőregiszterek, státuszregiszterek, stb. is).
- A *címzési módok*at (addressing modes). A címzési módok határozzák meg, hogy az utasítások milyen módon tudnak egy használni kívánt objektumra (pl. egy művelet operandusaira) hivatkozni.
- A *jelzőbitek*et (flags). A jelzőbitek az utasítás végrehajtása során történt speciális események bekövetkeztét jelzik (pl. a zero flag jelezheti, hogy egy aritmetikai vagy logikai művelet eredménye 0 lett-e).
- A *perifériakezelés* módját (I/O handling), azaz hogy a processzor milyen lehetőséget ad a programozónak a perifériákkal való kommunikációra.
- A *megszakítás- és kivételkezelés*t (interrupt and exception handling), vagyis hogy a processzor milyen lehetőséget biztosít a programozónak a megszakításkérések és a program végrehajtása során fellépő kivételes események kezelésére.

Ahogy egy specifikációban rögzített lábkiosztású és funkcionalitású áramkör belső megvalósítása is többféle lehet, egy adott programozási felület mögött is eltérő belső mikroarchitektúrális megvalósítások állhatnak. A programozási felület állandósága biztosítja egy processzorcsalád tagjainak kompatibilitását. Példának okáért az x86 utasításkészlet architektúra 1978-ban jelent meg, és az arra fejlesztett szoftverek mind a mai napig, több, mint 30 év után is futnak a processzorcsalád legújabb tagjain, annak ellenére, hogy azok belső felépítése, mikroarchitektúrája teljesen különbözik az 1978-ban megjelent Intel 8086 processzorétól.

A továbbiakban többféle szempont szerint megvizsgáljuk, csoportosítjuk a processzorok utasításkészletét. Nem célunk valamely processzor utasításkészletének teljes bemutatása (ezt a terjedelmi és időkorlátok nem tennék lehetővé), de példaként betekintünk néhány, valamilyen szempontból érdekesnek ítélt processzor utasításkészletébe is.

3.2. Az utasítások jellemzői

3.2.1. Utasítások felépítése

A processzorok működésük során gépi utasításokat (gépi kódot) hajtanak végre. Az utasítások a memóriában binárisan kódolva vannak tárolva. Egyáltalán nem mindegy, hogy az utasítások hogyan vannak leképezve erre a bináris reprezentációra: az utasítások kódolásakor egyrészt figyelembe kell venni a helytakarékossági szempontot

(a gépi kódú programok minél kisebbek legyenek), másrészt az utasításokat úgy kell kódolni, hogy azt a processzor az utasítás letöltése után minél gyorsabban dekódolni tudja.

Általános esetben az utasítások az alábbi részekből állnak:

- Az utasítás első része az utasítás kódja (operation code, röviden: opcode), amelyből a processzor dekódolás után megtudja, hogy mit kell tennie. Az utasítás kódjából (esetleg további paraméterekből) kiderül, hogy az utasításnak még milyen paraméterei, operandusai vannak,
- az operandusok címei (vagy értékei), egyik a másik után felsorolva,
- a cím, ahová az eredményt tenni kell,
- a következő végrehajtandó utasítás címe.

A helytakarékosságot szem előtt tartva ezeket többféle módon szokták csökkenteni. Kevés kivételtől eltekintve a következő utasítás címét nem szokták utasításonként előírni, hanem a következő utasítást általában az adott utasítás után helyezik el a memóriában. Ehhez *utasításszámláló*t (Program Counter vagy más néven Instruction Pointer) használnak, aminek az értékét egy utasítás végrehajtásakor annak hosszával növelik. (Természetesen előfordulhat, hogy nem a memóriában soron következő utasítást szeretnénk végrehajtani, ezért szükség van *vezérlésátadó* utasításokra, amire bővebben az utasításfajtáknál térünk ki).

Az operandusok számának csökkentésével is lehet helyet (és komplexitást) megtakarítani:

- Számos processzor támogatja a 3 operandus címet használó műveleteket, pl: `R1 ← R2 + R3`.
- Ha az eredményt mindig az első operandus helyére tesszük, már két cím is elég, pl: `R1 ← R1 + R2`.
- Még a két cím is tovább csökkenthető: bizonyos processzorokban az első operandus és az eredmény tárolására kötelezően egy kitüntetett regisztert használnak, amit *akkumulátor*nak (accumulator) neveznek, pl. az `ADD R1` utasítás az akkumulátor értékét R1-el növeli.

(Az operandusok azonosítására használatos operandus címek jelenthetnek memóriabeli címeket (ez a *direkt memóriacímzés*), de jelenthetik valamelyik regiszter kijelölését is (*regisztercímzés*), a címzési módokkal külön, részletesen foglalkozunk.)

3.2.2. Utasítástípusok

Attól függően, hogy az egyes utasításoknak mi a feladata, különböző *utasítástípusok* (más szóval utasításfajták) léteznek, például:

- *adatmozgatás* - Címzési módtól függően sokféle lehet: a cél lehet memória vagy regiszter, a forrás lehet memória, regiszter, vagy konstans (literál).
  Példák: `R1 ← R2`, `R1 ← MEM[100]`, `R1 ← 42`, `MEM[100] ← R1`, `MEM[200] ← MEM[100]`, `MEM[100] ← 42`.
- *aritmetikai és logikai műveletek* - A négy alapműveleten és az alapvető bool algebrai műveleteken kívül ide szokták sorolni a különböző eltolásokat (shift), forgatásokat (rotate) is.
  Példák: `R1 ← R2 + R3`, `R1 ← MEM[100] * 42`, `MEM[100] ← R1 & R2`, `MEM[200] ← MEM[100] << 2`.
- *vezérlésátadó utasítások* - Feltétel nélküli és feltételes ugrások (az ugrási cím lehet abszolút illetve a PC értékéhez képest relatív is) szubrutinhívás és szubrutinból való visszatérés. (Processzortól függően az utóbbiak is lehetnek feltételesek.)
  Példák: `JUMP -42`, `JUMP +28 IF R1==R2`, `CALL proc`, `RETURN`.
- *veremkezelő utasítások* - PUSH: érték elhelyezése a verembe, POP: érték kivétele a veremből, esetleg a veremmutatót állító egyéb utasítások.
  Példák: `PUSH R1`, `PUSH 42`, `R2 ← POP`.
- *I/O műveletek* - IN: érték olvasása valamelyik (címmel kijelölt) perifériáról, OUT érték kivitele valamelyik (címmel kijelölt) perifériára.
  Példák: `IO[42] ← R1` (a 42-es perifériacímre kiadjuk az R1 értékét), `R1 ← IO[42]` (a 42-es perifériacímről beolvasunk egy adatot).
- *transzcendens függvények* számítása
  Példák: `R2 ← SIN R1`, `R2 ← SQRT 42`,
- *egyebek* - Operációs rendszer szubrutinjainak a hívása, virtuális memória kezelő utasítások, stb.

3.2.3. Címzési módok

Az operandus címzési módja határozza meg, hogy egy művelet operandusát hol találja meg a processzor. Az operandusok (a Neumann-architektúrában) alapvetően három helyen tartózkodhatnak:

- magában az utasításkódban, a gépi kódba ágyazva,
- a processzor egy regiszterében,
- vagy a memóriában.

(Módosított Harvard architektúrában az utasítás memória is szóba jöhet, mint az operandus tartózkodási helye.)

Mindenesetre azt az információt, hogy a processzor hol keresse az operandust, úgyszintén az utasítás gépi kódjában kell tárolni. Tehát amikor a processzor beolvas egy gépi kódú utasítást a memóriából, az opcode alapján (vagy az utasításszó további bitjeiből) rögtön tudja, hogy pl. az első operandusra milyen címzési mód vonatkozik. Ha "regiszter" a címzési mód, akkor az "Operanduscím$_1$" mezőből (lásd 3.1. ábra) ki tudja olvasni, hogy hányas számú regiszterben van az operandus. Ha "direkt" a címzési mód, akkor az "Operanduscím$_1$" mező egy mutatót tartalmaz az operandus memóriabeli elhelyezkedésére, stb.

Az alábbi táblázatban felsoroljuk a leggyakrabban előforduló címzési módokat:

Címzési mód neve	**jelentése (példa):**	**leírása:**
Regiszter	R1 ← R2 + R3	Az operandus egy regiszter.
Közvetlen konstans	R1 ← R2 + 42	Az operandus egy konkrét skalár konstans.
Direkt	R1 ← R2 + MEM[42]	Az operandus a memóriában van, címe: 42.
Regiszter indirekt	R1 ← R2 + MEM[R3]	Az operandus a memóriában van, R3 rá mutat.
Eltolt indirekt	R1 ← R2 + MEM[R3+42]	Az operandus a memóriában van, címe: R3+42.
Memória indirekt	R1 ← R2 + MEM[MEM[R3]]	Az operandus a memóriában van, címe: MEM[R3].
Indexelt	R1 ← R2 + MEM[R3+R4]	Az operandus a memóriában van, címe: R3+R4.

Természetesen nem minden processzor támogat ilyen sok címzési módot. Minél többféle címzési módot támogat a processzor, annál egyszerűbb assembly-ben programozni, viszont annál bonyolultabb a processzorban az összetettebb címzési módokat feldolgozó logika.

Bizonyos processzorokban egyes utasításfajták csak bizonyos címzési módokkal használhatók. Például az Intel 8085 processzornál perifériaműveletek esetén a kivitt vagy a beolvasott értéket mindig az akkumulátor hordozza ("regiszter" címzési mód), a perifériacímet pedig (8 biten) mindig direkt módon kell megadni; nem lehet például a B regiszter értékét a C regiszterben megadott sorszámú portra kivinni.

3.2.4. Elágazások kezelése

Az elágazások / vezérlésátadások (branch) során az utasítások lineáris, egyiket a másik után történő letöltése megszakad, és a feldolgozás az ugrási címen lévő utasítással folytatódik tovább. A kód áthelyezhetőségének érdekében célszerű az utasításszámlálóhoz relatív címzést alkalmazni (pl. a `JUMP -28` 28 bájtot visszaugrik a programban, attól függetlenül, hogy a program hol, melyik konkrét memóriacímen helyezkedik el). A legtöbb utasításkészlet architektúra azonban megengedi az abszolút címzést is (amikor is az ugrási cím maga a memóriacím).

A feltételes ugró utasítások esetén az ugrási feltétel megadásának módjában az egyes utasításkészlet architektúrák eltérőek lehetnek. Az alábbi három megoldás terjedt el szélesebb körben:

- *feltétel kódok* használata: Ebben az esetben az ugrás a feltételkódok állásának függvényében valósul meg. Bizonyos aritmetikai műveletek és összehasonlító utasítások beállítják ezeket a feltételkódokat, az ugró utasítások pedig a megfelelő feltételkód állásától függően hajtják végre az ugrást. Példa:

```
COMPARE R1, R2          // a háttérben beállnak az összehasonlítás
                        // eredményét tükröző feltételkódok
JUMP label IF GREATER   // az előbb beállt 'GREATER' feltételkódot
                        // vizsgálja
```

- *feltétel regiszterek* használata: Az összehasonlítás (tehát az ugrási feltétel) eredménye nem feltételkódokban, hanem egy regiszterben kerül eltárolásra, egy 0/1 érték formájában. Az ugró utasításnak ezt az eredményregisztert kell odaadni, és ha az nem 0 értéket tartalmaz, akkor megvalósul az ugrás. Példa:

```
R0 ← R1 > R2
JUMP label IF R0
```

- "*összehasonlít és ugrik*" módszer: Az ugrási feltétel kiértékelése nem különül el az ugró utasítástól. Példa:

```
JUMP label IF R1 > R2
```

Bizonyos utasításkészlet architektúrák lehetővé teszik úgynevezett *predikátumok* használatát. Ezekben az architektúrákban az utasításokat el lehet látni egy extra operandussal, a predikátumregiszterrel. A predikátumregiszterek 1 bitesek. Ha az utasításhoz kapcsolódó predikátumregiszter értéke 0, akkor az utasítást a processzor nem hajtja végre. Példa:

```
R1 ← R2 + 32 IF P2
```

Ebben az esetben a P2 predikátumregiszter értékétől függ, hogy az összeadás végrehajtódik-e, vagy sem. A predikátumregisztereket összehasonlító műveletekkel lehet beállítani:

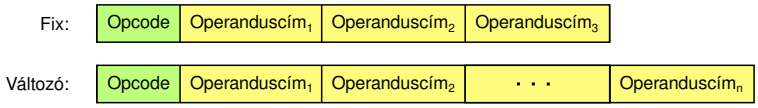
```
P2 ← R3 ≤ R5
```

Ha R3 kisebb vagy egyenlő, mint R5, P2 értéke 1 lesz, ellenkező esetben 0.

A predikátumok azért hasznosak, mert ugrás nélkül teszik lehetővé az utasítások feltételes végrehajtását. A későbbiekben látni fogjuk, mennyire fontos a hatékony végrehajtás szempontjából, hogy ne szakadjon meg az utasítások egymásutánisága.

3.2.5. Utasítások kódolása

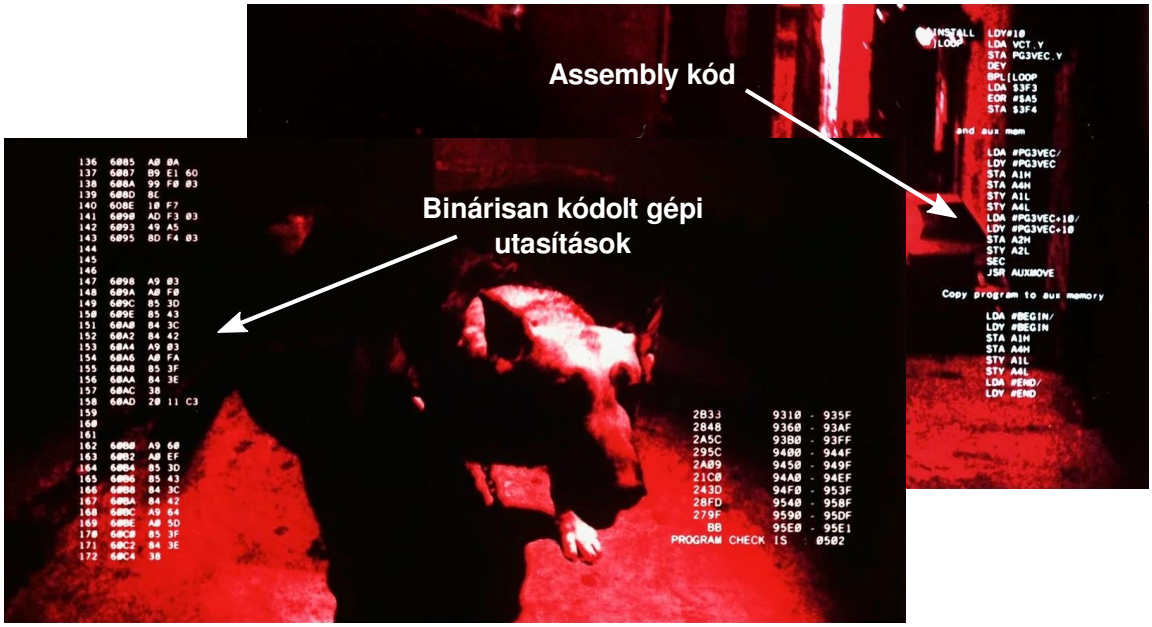
A különböző utasításkészlet architektúrák között abban is különbség van, hogy az összes utasítást azonos hosszúságú bitsorozattá kódolják-e, vagy az utasítások hossza változó is lehet. Előbbi esetben fix hosszú, utóbbi esetben változó hosszú utasításkódolásról beszélünk (3.1. ábra).



3.1. ábra. Fix és változó utasításkódolás

A változó hosszú utasításkódolás helytakarékosabb, hiszen így az utasítások tényleg csak annyi helyet foglalnak, amennyit szükséges, pl. egy 1 operandusú művelet esetén nem kell üres, nem értelmezett mezőket is az utasítás részévé tenni. Ugyanakkor az utasításlehívás és dekódolás szempontjából a fix utasításkódolás lényeges egyszerűsítést jelent.

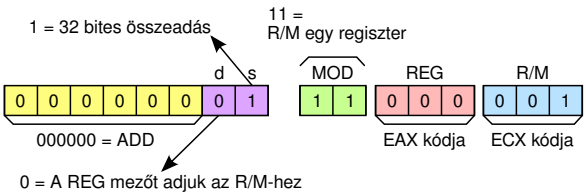
A binárisan kódolt utasításokat gépi kódnak nevezzük. Az egyes processzorokat azonban gépi kódban, tehát az egyes utasításokat műveleti kódból és címekből binárisan előállítva nem szokták programozni. Ha szükség van a lehető legalacsonyabb szintű programot írni, az adott processzor assembly nyelvét alkalmazzák. Az assembly nyelv sor orientált. Egy sorban egy gépi utasításnak megfelelő assembly nyelvű utasítás (vagy a fordítónak szóló direktíva) áll. Egy assembly nyelvű utasítást egy néhány betűs *kulcsszó*val (mnemonik) adunk meg, amelyet az utasítás operandusai követnek (ha vannak). Az assembly nyelvű programokat lefordító programot *assembler*nek nevezik. Minden egyes utasításkészlet architektúrához más és más assembly nyelv és ennek megfelelően más és más assembler tartozik. Egyszerű mikrogépes környezetben (pl. Z80 vagy 6510 alapú gépeknél) az assemblerek a fordítást tipikusan két menetben elvégezve egyből a mikrogép memóriájába, a kívánt memóriaterületre helyezték el a gépi kódra lefordított programot. A PC-s világban jelenleg is használatos megoldások lehetővé teszik, hogy a szoftverfejlesztés során az alacsony és a magas szintű nyelveket keverjük, vagyis program bizonyos részeit (a nagy számításigényű, gyors feldolgozást igénylő feladatokat) assembly-ben, más részeit egy magas szintű programozási nyelven írjunk meg.



3.2. ábra. Fix és változó utasításkódolás

A 3.2. ábrán két képet látunk a Terminátor című filmből. A hátsó (jobb oldali) az assembly programot, az első (bal oldal) pedig az ehhez tartozó binárisan kódolt gépi utasítássorozatot mutatja, mely már a processzor közvetlen bemenete lesz. A Terminátor MOS 6502-es processzort használ, melyről az ábra alapján több következtetést is levonhatunk. A gépi kód tördelése követi az assembly programot, ami segít nekünk észrevenni, hogy ez a processzor változó hosszúságú utasításkódolást alkalmaz: az első assembly sorhoz tartozó utasítás 2 bájtos, a második 3 bájtos, a negyedik egy bájtos, stb. Az assembly kódból az is jól látszik, hogy 1 operandusúak a műveletek: az `EOR` (kizáró vagy) például az egyetlen operandusa és az A (akkumulátor) regiszter között végzi el a műveletet, és az eredmény az A regiszterbe kerül vissza. A `DEY` utasításnak pedig egyetlen operandusa sincs, implicit módon az Y regisztert csökkenti eggyel. A feltételes elágazások kezelésére a `BPL` utasítás enged bepillantást: az ugrás akkor következik be, ha pozitív volt az előző művelet eredménye. Ezt a viselkedést az imént ”feltétel kódokra” alapozott elágazáskezelésnek neveztük. (Érdekességképp: ez a processzor a maga korábban teljesítményben versenyképes volt az Intel és a Motorola megoldásaival, miközben az ára azok hatodrésze volt. Sok-sok olcsó, népszerű gép épült rá, ami nagyban hozzájárult a számítástechnika elterjedéséhez. A Terminátor egyébként a Nibble magazin példaprogramját futtatja.)

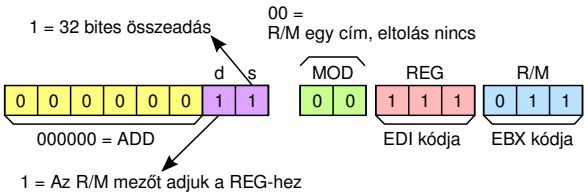
A 3.3., 3.4., 3.5. ábrán további példákat láthatunk az utasítások kódolására. A három ábra az x86 utasításkészlet összeadó utasításának (ADD) kódolását mutatja be három különböző címzési mód használatával. A 3.3. ábrán egy regisztercímzést használó összeadás, az ADD ECX, EAX, míg a 3.4. ábrán egy regiszter indirekt címzést használó alak, az ADD EDI, [EBX] kódolását láthatjuk.



3.3. ábra. Az ADD ECX,EAX utasítás kódolása

Az ábrákon jól látszik, hogy mindkét utasítás 3-3 bitet használ az operandusok forrásának leírására (az egyik operandust a ”REG”, a másikat az ”R/M” mező írja le). A 3 bit 8 regiszter kijelölését teszi lehetővé. Azt, hogy ezek a regiszterek pontosan mit is jelentenek, a címzési mód dönti el. Az x86 ADD utasításának egyik operandusa kötelezően regiszter címzéssel adandó meg, vagyis a REG mezőben lévő szám azt jelenti, hogy az annyiadik

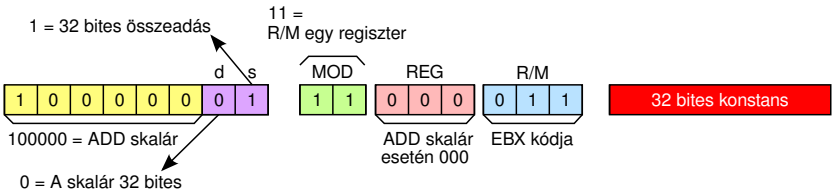
regiszter tartalma az egyik operandus. A másik operandus címzését a ”MOD” mező határozza meg. A 3.3. ábrán a MOD mező értéke 11, ami azt jelenti, hogy az oda írt 3 bites érték is egy regiszter sorszámának tekintendő, és a regiszter tartalma maga az operandus. Ezzel szemben a 3.4. ábrán a MOD értéke 00, ami a regiszter indirekt címzés kódja: az annyiadik regisztert címnek (mutatónak) tekinti, az operandus a memóriában van, ott, ahova az a regiszter mutat.



3.4. ábra. Az ADD EDI, [EBX] utasítás kódolása

Ebben a két példában tehát mindkét utasítás 2 bájtos volt, az egyik értéke 01C1h, a másiké 033Bh.

A harmadik példában (3.5. ábra) az egyik operandus egy regiszter, a másik egy közvetlen konstans, ami az utasítás kódjának része. A 32 bites konstanssal együtt ebben az esetben egy 6 bájtos utasítást kapunk (értéke 81C3156B6501h), amiből az is látszik, hogy az x86 egy változó hosszú utasításkódolást alkalmazó architektúra.



3.5. ábra. Az ADD EBX, 23423765 utasítás kódolása

3.3. Az utasításkészlet architektúrák jellemzői

3.3.1. Bájtsorrend

Ha egy egész szám ábrázolására n bájtot használunk, akkor a számhoz tartozó bájtokat a memóriában elvileg $n!$ sorrendben lehetne elhelyezni, de ezek közül leginkább csak a két végletet szokták használni: ha a legnagyobb helyiértékű byte (MSB - Most Significant Byte) van elöl (vagyis az alacsonyabb memóriacímen), és a többiek egyre csökkenő helyiérték szerinti sorrendben követik, akkor azt *felvég* (big-endian) sorrendnek nevezzük; ha pedig a legkisebb helyiértékű byte (LSB - Least Significant Byte) van elöl és a többiek egyre növekvő helyiérték szerinti sorrendben követik, akkor azt *alvég* (little-endian) sorrendnek nevezzük. (A byte-sorrend kérdését pedig angolul endianness-nek hívják. Az angol nyelvű elnevezés Jonathan Swift: Guliver utazásai című könyvéből származik, melyben két csoport azért keveredik háborúba, mert nem tudnak megállapodni abban, hogy a lágytojást a kisebbik végén (little endian), vagy a nagyobbik végén (big endian) kell-e feltörni.) Például a `0x1A2B3C4D` hexadecimális számot big-endian bájtsorrend szerint a memóriában úgy kell eltárolni, hogy az egymás utáni memóriacímekre sorban az `0x1A`, `0x2B`, `0x3C` és `0x4D` bájtok kerüljnek. Little endian esetén ezek a bájtok pont fordított sorrendben kerülnek a memóriába, vagyis a sorrend `0x4D`, `0x3C`, `0x2B` és `0x1A`.

Az egyes CPU architektúrák az egyiket vagy másikat, esetleg átkapcsolható módon mindkettőt támogatják (bi-endian). Big-endian sorrendet használó utasításkészlet architektúrák a Motorola 6800 és 68000; IBM POWER, System/360, z/Architecture; HP PA-RISC és eredetileg a SUN SPARC. Little-endian sorrendet használ az Intel x86 (beleértve x86-64 is), a MOS Technology 6502 (tehát a Terminátor is), 6510; Zilog Z80; DEC PDP-11, VAX.

Példák bi-endian sorrendet használó utasításkészlet architektúrára (némelyik szoftverből, akár működés közben is tud váltani, bizonyos rendszerekben pedig alaplapi hardver beállítás dönti el a byte-sorrendet): ARM Holdings ARM, IBM-Intel-Motorola PowerPC; DEC Alpha; SUN/Oracle SPARC v9; MIPS Technologies MIPS; Intel IA-64.

3.3.2. Perifériakezelő utasítások

Egy processzor számára kétféle lehetőség van a *perifériák* címzésére. A CPU vagy egyetlen címteret használ a memória és a perifériák elérésére, vagy két külön címteret. Az első esetben *memóriára leképzett perifériakezelés*ről (memory mapped I/O), a második esetben külön *I/O utasítások* használatáról (port I/O, port mapped I/O) beszélünk.

A külön I/O utasítások esetében a processzor utasításkészletében külön I/O kezelő utasítások állnak rendelkezésre, melyekkel egy adott című perifériáról adatot lehet beolvasni, illetve oda adatot lehet küldeni (tipikusan: IN a perifériáról való bevitelre és OUT a perifériára való kiírásra). A periféria beviteli/kiviteli utasításokban a perifériák megcímzésére tipikusan jóval kevesebb címzési mód használható, mint a memória olvasó/író utasításokban a memória megcímzésére.

Memóriára leképzett perifériakezelésnél a perifériák címei a memória címtartományának részei. Ez azt jelenti, hogy egyes memóriacímek mögött igazából perifériák állnak, az ezekre a címekre történő írás/olvasás műveletek a perifériákra vonatkoznak. Memóriára leképzett perifériakezelést megvalósító utasításkészlet architektúrákban ennélfogva nincsenek is külön perifériakezelő utasítások, a perifériákkal való kommunikációra a memória írás/olvasás műveleteket kell használni.

3.3.3. Ortogonalitás

Amennyiben egy utasításkészlet architektúrában minden (vagy legtöbb) címzést használó utasításban minden (vagy legtöbb) címzési mód használható, a processzor utasításkészletét *ortogonális*nak (vagy közel ortogonálisnak) nevezzük. (Az analógia nyilvánvaló: egy derékszögű koordináta-rendszerben egy pont koordinátáiként bármely X értékhez bármely Y érték tartozhat.) Az ortogonalitás a gépi kódban (vagy inkább assembly nyelven) dolgozó programozó számára kényelmes tulajdonság lehet, a magas szintű programozási nyelvek fordítói azonban nem mindig használják ki.

3.3.4. RISC vs. CISC

A processzorok tervezésének és gyártásának peremfeltételeit, célfüggvényeit mindig döntően befolyásolja az aktuális technológiai és gazdasági környezet. Az 1970-es évek elején a félvezető technológia lehetővé tette az egyetlen áramköri tokból álló processzorok, a *mikroprocesszor*ok gyártását. A 70-es években az újabb mikroprocesszorok egyre nagyobb számú, egyre összetettebb funkciót megvalósító utasítással rendelkeztek. Ez több szempontból is előnyös volt:

- A memóriának a CPU-hoz képest alacsony működési sebessége miatt. Ha a program összetett utasításból áll, akkor a program kevesebb utasítással írható le, tehát kevesebbszer kell a lassú memóriához fordulni a program futása során. (Egyetlen lassú memóriaművelettel lehívott utasítással a gyors processzor több munkához jut.)
- A memória (sőt a háttértárak) magas ára miatt. Ha a program összetett utasításból áll, akkor a program kevesebb utasítással írható le, tehát kevesebbet foglal a drága memóriából és háttértárból.
- A fordítók akkori technológiája és futási sebessége szempontjából előnyös volt a magas szintű programozási nyelvek és az assembly nyelv közötti szemantikus rés csökkentése az egyre bonyolultabb gépi utasítások bevezetésével. (Tehát könnyebb fordító programot készíteni, ha a processzor utasításkészlete közel van a magas szintű programnyelv utasításkészletéhez.)

Utólag ennek, az összetett utasításokat alkalmazó stratégiának az eredményeként létrejött processzorokat használó gépeket *komplex utasításkészletű számítógép*nek, azaz CISC-nek (Complex Instruction Set Computer) nevezték el. A CISC számítógépek jellemzői:

- ”Kényelmes”, összetett műveletek, tipikusan regiszter-memória utasítások (a műveletek egyik operandusa regiszter, a másik egy memóriabeli objektum lehet). Példa: `R1 ← R2 + MEM[42]` (ebben az a kényelem, hogy nem kell a MEM[42]-ből külön beolvasni az adatot, egyetlen utasítás elvégzi az adat beolvasását és az összeadást is).
- Sokféle utasítással is el lehet érni ugyanazt a hatást (redundancia).
- Sokféle címzési mód.
- Változatos utasításhossz.

- Az utasítások végrehajtási ideje változatos: vannak gyorsan és lassan végrehajtható utasítások.

Az 1980-as években született, és a 90-es években igen sikeres lett egy alternatív stratégia: a *csökkentett utasításkészletű számítógép*, azaz RISC (Reduced Instruction Set Computer). Érdekes módon a RISC processzorok legfőbb jellemzője nem is a kis számú utasítás (ahogy a neve sugallja), hanem az utasítások egyszerű mivolta. Minden utasítás egyetlen, elemi műveletet hajt végre, aminek következtében a processzorok belső felépítése sokkal egyszerűbb, átláthatóbb, ami lehetővé teszi a hatékonyabb mikróarchitektúra kialakítását. A RISC számítógépek jellemzői:

- Egyszerű, elemi utasítások, redundanciát kerülve.
- Load/Store és regiszter-regiszter típusú műveletek. Azaz, amíg a CISC esetén létezett `R1 ← R2 + MEM[42]` utasítás, addig RISC esetén ehhez két utasítás is kell `R3 ← MEM[42]; R1 ← R2 + R3`.
- Kevés címzési mód.
- Fix utasításhossz.
- Minden (vagy legalábbis a legtöbb) utasítás végrehajtása ugyanannyi ideig tart.

Mindkét tervezési stratégiának vannak előnyös és hátrányos tulajdonságai egyaránt:

- A CISC előnye a tömörség, vagyis a program kevés utasításból áll. RISC esetén a programot elemi műveletekre kell bontani, ami nagyobb kódot eredményez. Ennek ellensúlyozására például a PowerPC architektúrában hardveres programkód tömörítést alkalmaznak. A memória (és a háttértárak) árának csökkenése, sebességének növekedése és a processzor lapkáján megvalósított több szintű cache mára kiküszöbölte a program hosszának növekedésével járó problémákat.
- A RISC előnye az egyszerűség. Mivel minden utasítás elemi művelet, a processzor felépítése sokkal egyszerűbb. Az egyszerűbb processzorban kevesebb a tervezési hiba. Az egyszerűbb processzort huzalozott vezérléssel is meg lehet valósítani (nem kell mikroprogramozott vezérlés), aminek köszönhetően magasabb órajel-frekvencia érhető el (például 1995-ben a RISC szervezésű DEC Alpha 21164 EV5 már 333MHz-en működött, míg a CISC szervezésű Intel Pentium MMX P54CS még csak 133 MHz-re volt képes). Az egyszerűségnek köszönhetően kisebb lesz a processzor az IC-n, aminek további, szerteágazó következményei vannak:
  - Alacsony fogyasztás
  - Jobb gyártási kihozatal (több CPU helyezhető egy ostyára, és kisebb a selejtarány is)
  - Mivel kevés helyet foglal a CPU az IC-n, az IC-re számos járulékos eszköz integrálható (több cache, memóriavezérlő, grafikus vezérlő, I/O vezérlők, stb.)
- A CISC processzorokhoz kevesebb regiszterre van szükség. A RISC processzorokban, mivel (a Load/Store kivételével) minden művelet operandusai regiszterek, nyilván több regiszter szükséges.

Az tény, hogy az utóbbi 30 évben megjelent utasításkészlet architektúrák mind RISC szervezésűek. A ma kapható legelterjedtebb CISC architektúra, az x86 is belül már RISC kialakítású: maga az x86 utasításkészlet architektúra kétségtelenül CISC, de a processzor első dolga, hogy a betöltött komplex utasításokat egyszerű, elemi műveletek sorozatára (mikroutasításokra) bontja, olyan elemi műveletekre, melyek a RISC processzorokra jellemzőek.

A közismert processzorok közül a CISC kategóriába soroljuk a következőket:

- 8 bitesek: Intel 8080, 8085; Zilog Z80; MOS Technology 6502, 6510; Motorola 6800, 6809.
- 16 bitesek: DEC PDP-11; Intel 8086, 80286; a 8086-tal kompatibilis, de 8 bites külső adatbusszal rendelkező Intel 8088; a 32 bites regisztereket tartalmazó, de kívül 16 bites Motorola 68000 és 68010.
- 32 bitesek: DEC VAX; IBM System/360; Motorola 68020, 6830, 6840, 6860; az összes IA-32 ISA szerinti CPU, így: Intel 80386, 80486, Pentium, Pentium Pro, Pentium II, Pentium III, Pentium 4; AMD K5, K6, K7 (Athlon, Duron és a korábbi Sempron processzorok).
- 64 bitesek: IBM z/Architecture; Intel IA-64 (Itanium, Itanium 2, Itanium 9300); az összes x86-64-et implementáló CPU, így: AMD K8 (korábbi Opteron, Athlon 64, Athlon 64 FX, Athlon 64 X2, Athlon II, Athlon X2 és a későbbi Sempron processzorok), K8L (a Turion különböző fajtái kis fogyasztású célra), K10 (későbbi Opteron, Phenom, Phenom II) ; Intel CPU-k érdemben a Core mikroarchitektúrától kezdve.

A RISC kategóriába sorolt ismertebb általános célú processzorok 32-bitestől kezdődnek és gyakran a nevükben is hordozzák a RISC szót:

- 32 bitesek: IBM POWER1, POWER2 (**P**erformance **O**ptimization **W**ith **E**nhanced **R**ISC); IBM-Intel-Motorola PowerPC 615-ig (**P**erformance **O**ptimization **W**ith **E**nhanced **R**ISC – **P**erformance **C**omputing); SUN SPARC (**S**calable **P**rocessor **Arc**hitecture) 32 bites verziói ; ARM Holdings ARM (**A**dvanced **R**ISC**M**achine); MIPS Technologies MIPS32 (**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages) implementációi; HP PA-RISC (**P**recision **A**rchitecture **RISC**) eredeti 32 bites változata
- 64 bitesek: DEC Alpha; IBM POWER3-tól; IBM-Intel-Motorola PowerPC 620-tól; SUN SPARC 64 bites verziói; MIPS Technologies MIPS64 implementációi; HP PA-RISC 2.0.

3.3.5. Néhány érdekes utasításkészlet

Érdekességként megemlítjük, hogy a CISC/RISC rövidítések mintájára létezik még néhány utasításkészlet-megnevezés:

- MISC: Minimal Istruction Set Computer - Itt ténylegesen nagyon alacsony számú utasításról van szó. A CPU a műveletek operandusait tipikusan a veremből veszi, és az eredményt is oda helyezi el. Ez a működés a Forth magas szintű programozási nyelvhez illeszkedik.
- OISC: One Istruction Set Computer - Egyetlen megfelelően választott összetett utasítás (például kivonás és ugrás, ha az eredmény negatív) segítségével fel lehet építeni a szokásos utasításkészletekben szereplő alapvető utasításfajtákat.
- NISC: No Instruction Set Computer - Ennek az architektúrának nincsenek a klasszikus értelemben vett gépi szintű utasításai, hanem a fordító ún. *nanokód*ot generál (a neve azt fejezi ki, hogy mikrokódnál is alacsonyabb szintű), ami közvetlenül vezérli a processzor egyes erőforrásait.
- ZISC: Zero Instruction Set Computer - Ez az *információvezérelt* információfeldolgozási modellnek felel meg. Az alapötlete a mesterséges neurális hálózatokból származik. Nagy számú független cella egymással párhuzamosan mintaillesztést hajt végre: egy bemenő vektort hasonlít össze a saját tárolt vektorával és jelzi, hogy illeszkedett-e.

3.4. Példák utasításkészlet architektúrákra

Az alábbiakban bemutatjuk néhány fontosabb utasításkészlet architektúra rövid történetét és főbb tulajdonságait. Ezek az architektúrák a további fejezetekben rendszeresen vissza fognak térni, segítségükkel demonstráljuk, hogy a bemutatott elveket a gyakorlatban implementált, kereskedelmi forgalomban kapható (vagy régebben kapható volt) processzorok hogyan alkalmazzák. Néhány ma is élő utasításkészlet architektúra:

Az x86 architektúra. Ezt vesszük előre, mert a legtöbb ember a számítógépeket az x86 architektúrán keresztül ismeri meg. Az x86 architektúra karrierje 1978-ban indult, az Intel 8086-os processzorral. A sikert az 1981-ben megjelent IBM PC hozta meg, mely az Intel 8088-as processzorára épült. Abban az időben nem éppen ez volt a legfejlettebb, leggyorsabb, legnagyobb perspektívát kínáló választás, de a döntésben nem csak mérnöki szempontok játszottak szerepet. Az IBM PC óriási sikert aratott, ami magának az Intelnek is fejfájást okozott: a nem éppen elegáns megoldásokat felvonultató, a 8 bites mikroprocesszorok hagyatékára építő x86-ot többször megpróbálták kiváltani modernebb megoldásokkal, sikertelenül (iAPX432, Intel 960, Intel 860, Itanium - ez utóbbi a szerverek szegmensén többé-kevésbé hosszú távon is életképes lett). A hatalmas számban eladott IBM PC-nek, és az arra írt sok-sok szoftvernek köszönhetően az Intel minden, kompatibilitást megszakító próbálkozása kudarcba fulladt. Az architektúra időközben sokat fejlődött: megjelent a 32 bites, majd a 64 bites kiterjesztés, és idővel megjelentek a modern, szuperskalár, sorrenden kívüli végrehajtást (lásd későbbi fejezetek) alkalmazó megvalósítások is, nem csak az Intel, hanem több más gyártó részéről. Az x86 lassan a nagy teljesítményű munkaállomásokban és szerverekben is teret nyert, mostanában pedig az alacsony fogyasztási igényeket támasztó mobil eszközök piacát vette célba. Az 1978 óta tartó kompatibilitási kényszer ezt a fajta rugalmasságot megnehezíti, de pont az ezidő alatt keletkezett rengeteg szoftver és fejlesztői tapasztalat járul hozzá az x86 architektúra terjedéséhez. Az Intel a bevételekből hatalmas félvezető-ipari beruházásokat hajtott végre, aminek köszönhetően több, mint 1 generációs gyártástechnológiai előnye van minden versenytársával szemben. A fejlett gyártástechnológia lehetővé teszi, hogy az esetleg kevésbé innovatív termékeket is a konkurenciánál jobb műszaki paraméterekkel és kedvezőbb áron állítsa elő.

Az ARM architektúra. Talán meglepő, de nem az x86 a legelterjedtebb utasításkészlet architektúra, hanem az ARM (Advanced RISC Machines). 2010-ben 6.1 milliárd ARM processzort adtak el, ami egyes, nem hivatalos becslések szerint a 10-szerese az x86 eladásoknak (egyes források szerint 2005-ben a mobiltelefonok 98%-a, a 32 bites beágyazott rendszerek 90%-a alapult ARM processzoron). Az ARM utasításkészlet architektúrát az ARM vállalat fejlesztette ki, az első megvalósítás 1987-ben jelent meg. Az architektúra a kezdetektől fogva 32 bites (a 2011-ben bejelentett legújabb változata már 64 bites), és a kifejlesztése során fő szempont volt az egyszerűség és az alacsony fogyasztás. Maga az ARM tervez ugyan processzort, de azok gyártásával nem foglalkozik. Mind az ARM utasításkészlet, mind az ARM által tervezett processzorok tetemes összeg fejében licenszelhetők. Az ARM által tervezett processzorok sajátos, elegáns megoldásokat alkalmaznak, melyekkel a számítási teljesítmény és az alacsony fogyasztás közötti arany középút felé igyekeznek, tehát szó sincs arról, hogy minden elképzelhető erőfeszítéssel a minél nagyobb számítási teljesítményre törekednének, inkább az egyszerűség a fő szempont. Az ARM2 processzor például a maga 30.000 tranzisztorával a világ legkisebb 32 bites processzora, a jelenleg elterjedt 2 magos ARM Cortex A9 teljesítményfelvétele például teljes terhelés mellett 1-1.5 W, órajelfrekvenciától (1-1.5 GHz) függően. ARM processzor-licenesszel rendelkeznek többek között a Texas Instruments, Samsung, NVIDIA, ARM utasításkészlet-licenszet birtokol (processzorlicensz nélkül) például a Qualcomm. Az ARM elterjedésnek nagy lökést adhat, hogy a Microsoft bejelentette az architektúra támogatását a 2012-ben megjelent Windows 8 operációs rendszerében.

A PowerPC architektúra. A PowerPC utasításkészlet architektúra az IBM, a Motorola, és az Apple összefogásában született meg 1991-ben. Egy minden elemében modern, kifejezetten nagy számítási teljesítményre képes architektúra született, melyet eredetileg személyi számítógépekbe szántak. A kilencvenes évek közepére kifejezetten potens megvalósítások kerültek forgalomba, melyek felülmúlták az akkor kapható x86 processzorok teljesítményét. Az Apple Macintosh a Motorola 68000 sorozatú processzorok leváltására PowerPC platformra váltott, mely annyira erős volt, hogy valós időben emulálta a Motorola 68000 processzort, tehát a régebben írt szoftverek futtathatók maradtak. A PowerPC a személyi számítógépekben végül nem tudott versenyezni az x86-al, leginkább az arra megírt rengeteg alkalmazás (melyek PowerPC-n nem futnak), valamint az x86 nagy sorozatban való gyártásának gazdaságossága miatt. A PowerPC alapú processzorok fejlesztése azonban nem állt le, 2007-re az órajel frekvencia elérte az 5 GHz-et (!), 2010-re pedig 4.25 GHz-en üzemelő, 8 magos, magonként 4 szálat futtató változat is kereskedelmi forgalomba került. Ezek a processzorok a fejlett lebegőpontos képességeknek köszönhetően azonos órajelfrekvencián is felülmúlták az x86 számítási teljesítményét, aminek köszönhetően nagy teljesítményű szerverekben, szuperszámítógépekben alkalmazzák őket. Van egy meglepő alkalmazási terület is, ahol fontos a teljesítmény, de nem annyira fontos a 30 éves programokkal való kompatibilitás: a játékkonzolok. A játékkonzolok 2011-2012-ben mind PowerPC-t, vagy azon alapuló processzort használtak (Xbox 360, Nintendo Wii, Playstation 3).

A SPARC architektúra. A SPARC (Scalable Processor Architecture) egy szerverekbe és nagy teljesítményű munkaállomásokba szánt 64 bites processzorcsalád, melyet a SUN fejlesztett ki 1987-ben, az m68k processzorok kiváltására. A SUN, felismerve, hogy az elterjedés szempontjából kulcsfontosságú a megfelelő számú és erejű ipari partner bevonása, a SPARC fejlesztésére, promótálására, terjesztésére egy külön szervezetet hozott létre. E szervezet tevékenységének köszönhetően a SPARC egy teljesen nyitott platform lett, számos gyártó fejleszt, gyárt és forgalmaz SPARC-ra alapuló processzorokat (többek között a Texas Instruments, a Fujitsu, a Cypress is). Ez a nyitottság 2006-ban teljesedett ki, amikor is a SUN nyílttá tette az UltraSPARC T1 processzor terveit, majd 2007-ben az UltraSPARC T2 terveit is. A tervek nyitottá tételével egyidőben hatalmas mennyiségű dokumentáció vált elérhetővé a processzor felépítésétől, működésétől kezdve a gyártásra vonatkozó részletekig. Az UltraSPARC elérhető tervei sok-sok mérnök számára tették lehetővé egy valós komplexitású (nem csak tantermi) processzor működésének a megértését. A processzor képességeiről annyit, hogy a világ 500 legerősebb számítógépek listáján a 2011. június 20.-i állás szerint az első helyet foglalta el, akkora fölénnyel, hogy számítási ereje az őt követő 5 másik szuperszámítógép összegzett teljesítményét is felülmúlta. Az első helyezését napjainkban ugyan elvesztette, de a SPARC továbbra is erős szereplő a jól skálázódó szerverek világában.

Néhány további utasításkészlet architektúra, melyek ugyan már nincsenek forgalomban, de a maguk idejében fontos szerepet játszottak:

Az m68k architektúra. Az m86k utasításkészlet architektúra első képviselője, a Motorola 68000, 1979-ben jelent meg. Egy kezdetektől fogva 32 bites architektúráról van szó, mely a 80-as években és a 90-es évek elején az x86 legfőbb versenytársa volt. Erre a népszerű processzorcsaládra épült a Apple Macintosh, a Commodore

Amiga, az Atari ST, és a SEGA Megadrive is, de a korai HP és Sun szerverek és munkaállomások, valamint számos lézernyomtató is. A Motorola belépésével a PowerPC szövetségbe az m68k utasításkészlet architektúrára épülő processzorok fejlesztése is leállt. Az utolsó modell a Motorola 68060 volt, mely 1994-ben jelent meg, felépítésében és teljesítményében is az Intel Pentium processzoraira hasonlított.

Az Alpha architektúra. A 64 bites Alpha architektúrát a DEC fejlesztette saját szerverei és munkaállomásai számára. Az első modell az 1992-ben megjelent DEC Alpha 21064 volt. Az Alpha processzorok a maguk idejében a leginnovatívabb processzorok közé tartoztak: a 21164 volt az első processzor nagyméretű, a CPU-val egy szilíciumszeleten elhelyezett első szintű cache-sel, a 21264 volt az első processzor, mely a sorrenden kívül utasítás-végrehajtást magas órajellel tudta kombinálni, a 21364 volt az első processzor integrált memóriavezérlővel, a 21464 volt az első, több végrehajtási szálat támogató processzor. Sajnos a 21464 sohasem került sorozatgyártásba, mert a DEC-et ebben az időben vásárolta fel a Compaq, amely leállította az Alpha processzorok fejlesztését. Ezek fényében talán mondani sem kell, hogy az Alpha processzorok, különösen a lebegőpontos számításokban, igen erősek voltak (a 833MHz-es 21264-es lebegőpontos számítási sebessége több, mint háromszoros az 1GHz-es Pentium III-hoz képest). Érdekesség, hogy ezt a teljesítményt és ezeket az innovatív megoldásokat az akkor igen szokatlan **kézi** tervezéssel érték el (nem használtak a tervezést és a megvalósítást automatizáló szoftvereket). Pontosan az Alpha processzorok sikere volt az, ami a félvezetőipart (konkrétan a processzor tervezőket) ráébresztette arra, hogy ne támaszkodjanak kizárólag automatizált tervező eszközökre, az azokkal kapott implementációkat legalább részben igyekezzenek kézi testreszabással optimalizálni.

A PA-RISC architektúra. A kezdetben 32 bites, majd később 64 bitesre kiterjesztett PA-RISC (Precision Architecture) utasításkészlet architektúra a Hewlett-Packard fejlesztése, első implementációja 1986-ban jelent meg. A HP motivációja is ugyanaz volt, mint a SUN-é: a korosodó m68k architektúrára épülő szervereinek és munkaállomásainak frissítése saját, nagyobb teljesítményű processzorral. A PA-RISC processzorok sajátossága, hogy nem használnak másodszintű cache-t, de első szintű cache-ből olyan sokat tartalmaznak, amennyit mind a mai napig egyetlen más vetélytárs sem. Számítási teljesítményéről egy adat: az 552 MHz-es PA-8600 lebegőpontos teljesítménye kb. kétszerese az 1GHz-es Intel Pentium III-nak.

Felmerül a kérdés, hogy a 90-es évek ígéretes architektúrái, az Alpha és a PA-RISC, hová tűntek olyan hirtelen. Az ok az IA-64 architektúra színre lépése volt.

Az IA-64 (Intel Itanium) architektúra. Az IA-64 architektúra kifejlesztését a HP kezdeményezte. A 90-es években a HP megállapította, hogy a számítógépgyártóknak (mint saját magának) nem költséghatékony a saját processzor fejlesztése, ezért 1994-ben az Intel-lel szövetkezve megkezdték a munkálatokat a jövő vállalati rendszereit megalapozó utasításkészlet architektúrán, és processzoron. A fejlesztést folyamatos, intenzív sajtó jelenlét követte nyomon. Az Intel óriási mennyiségű pénzt áldozott a fejlesztésre. Mindezek eredményeképp az elemzők meg voltak győződve róla, hogy az IA-64 rövid idő múlva uralni fogja a vállalati rendszerek teljes piacát. Mivel akkor ez annyira egyértelműnek tűnt, leállították az Alpha, a PA-RISC és a MIPS (itt nem részletezett) processzorok fejlesztését. Az első IA-64 processzor 2001-ben került piacra, kiábrándító teljesítménnyel, képességeit szinte minden olyan processzorcsalád felülmúlta, amelynek kiváltására hivatott (Alpha, PA-RISC, MIPS), mindössze pár ezer darabot adtak el belőle. Az Intel elképzelése az volt, hogy a processzort hardveres emuláció segítségével x86 kompatibilissé teszik (ahogy korábban a PowerPC szoftveresen emulálta a Motorola 68000-et), de hamar kiderült, hogy ez nagyon lassúra sikerült: az emulációval a programok futtatási sebessége egy 100MHz-es Pentiumnak feleltek meg, miközben akkortájt a Pentium 1 GHz felett járt. A gondot többek között az okozta, hogy az IA-64 egy EPIC architektúra, melyben a fordítóprogram (compiler) feladata a párhuzamosan végrehajtható utasítások összeválogatása. Az ilyen fordítóprogram kifejlesztése olyan nehézségekkel járt (és jár ma is), amire sem az Intel, sem a HP nem számított, emiatt a processzor képességei mind a mai napig kiaknázatlanok. Az IA-64 processzorok fejlesztése mindezek ellenére nem állt le, hatalmas összegeket fordítanak mind a processzor, mind a fordítóprogramok továbbfejlesztésére, miközben az Itanium alapú rendszerek piaci részesedése minimális (2001 és 2007 között összesen 55.000 Itanium alapú szervert adtak el). A nem x86 alapú szerverek és munkaállomások piacán 2008-ban a PowerPC részesedése 42%, a SPARC 32%, az Itanium pedig 26%. Mindeközben ahelyett, hogy az IA-64 elfogadottsága növekedne, egyre több cég jelenti be, hogy felhagy a platform támogatásával (2008-ban a Microsoft, legutóbb 2011 márciusában az Oracle).

A bemutatott utasításkészlet architektúrák legfőbb tulajdonságait foglalja össze a következő táblázat:

	x86	**ARM**	**PowerPC**	**SPARC**
Hány bites:	64	32/64	64	64
Megjelenés éve:	1978	1983	1991	1985
Operandusok száma:	2	3	3	3
Műveletek típusa:	reg-mem	reg-reg	reg-reg	reg-reg
Tervezési stratégia:	CISC	RISC	RISC	RISC
Regiszterek száma:	8/16	16/32	32	32
Utasításkódolás:	Változó (1-17 byte)	Fix (4 byte)	Fix/változó (4 byte - töm.)	Fix (4 byte)
Felt. ugrások kezelése:	Feltétel kód	Feltétel kód	Feltétel kód	Feltétel kód
Bájtsorrend:	Little	Big	Big/Bi	Bi
Címzési módok:	5	6	4	2
Perifériakezelés:	I/O utasítások	Mem.-ra lek.	Mem.-ra lek.	Mem.-ra lek.
Predikátumok:	Nincs	Van	Nincs	Nincs
	m68k	**Alpha**	**PA-RISC**	**IA-64**
Hány bites:	32	64	64	64
Megjelenés éve:	1979	1992	1986	2001
Operandusok száma:	2	3	3	3
Műveletek típusa:	reg-mem	reg-reg	reg-reg	reg-reg
Tervezési stratégia:	CISC	RISC	RISC	EPIC
Regiszterek száma:	16	32	32	128
Utasításkódolás:	Változó (2-22 byte)	Fix (4 byte)	Fix (4 byte)	Fix (16 byte)
Felt. ugrások kezelése:	Feltétel kód	Feltétel reg.	Összehas.& ugrik	?
Bájtsorrend:	Big	Bi	Big	Bi
Címzési módok:	9	1	5	?
Perifériakezelés:	Mem.-ra lek.	Mem.-ra lek.	Mem.-ra lek.	Mem.-ra lek.
Predikátumok:	Nincs	Nincs	Nincs	Van

3.5. Esettanulmány

Ebben a fejezetben megnézzük, hogy egy egyszerű C programból milyen gépi kódú (ill. azzal egyenértékű assembly) utasítássorozat keletkezik. Ezzel az egyszerű esettanulmánnyal az a célunk, hogy egy kicsit közelebbről megismerkedjünk azokkal az utasításokkal, melyeket a processzor ténylegesen végrehajt. Látni fogjuk, hogy mennyiben különbözik egy CISC, ill. RISC utasításkészlet, valamint hogy az utasításkészletekről tanultak a gyakorlatban hogyan érvényesülnek. A későbbi fejezetekben, a processzor belső felépítésének tárgyalásakor ezekre az ismeretekre elkerülhetetlenül szükség lesz. (Nem célunk azonban a különféle utasításkészlet architektúrák részletekbe menő ismertetése.)

A vizsgálatokhoz használt C program a Fibonacci sorozat egy elemét számolja ki, az alábbi algoritmus szerint:

```
#include <stdio.h>
int main () {
  int i, n, a, b, f;
  scanf("%d", &n);
  a = 0;
  b = 1;
```

```
  for (i=0; i<n; i++) {
    f = a + b;
    a = b;
    b = f;
  }
  printf ("%d
n", f);
}
```

A programból az ”LLVM” fordítási környezet eszközeivel először egy platform- és nyelvfüggetlen bájt-kódot[1], majd a bájt-kódból második lépésben assembly programot állítunk elő[2].

3.5.1. Megvalósítás x86 architektúrán

A fenti egyszerű C programból optimalizálás nélkül az alábbi assembly kódot kapjuk (csak a ciklusra vonatkozó részt vizsgáljuk, a scanf és printf hívásokat valamint a függvény kezdetekor és a végén végzett különféle járulékos műveleteket a könnyebb áttekinthetőség érdekében eltávolítottuk):

```
    mov DWORD PTR [ESP + 36], 0        ; a ciklusváltozó 0-ba állítása
    jmp .LBB0_1                        ; rögtön ugrunk a ciklusfeltétel kiértékeléséhez
.LBB0_2:
    mov EAX, DWORD PTR [ESP + 28]      ; EAX-be olvassuk a-t
    add EAX, DWORD PTR [ESP + 24]      ; hozzáadjuk a b-t
    mov DWORD PTR [ESP + 20], EAX      ; az eredményt f-be írjuk
    mov EAX, DWORD PTR [ESP + 24]      ; EAX-be olvassuk b-t,
    mov DWORD PTR [ESP + 28], EAX      ; ... majd beírjuk a-ba
    mov EAX, DWORD PTR [ESP + 20]      ; EAX-be olvassuk f-et,
    mov DWORD PTR [ESP + 24], EAX      ; ... majd beírjuk b-be
    inc DWORD PTR [ESP + 36]           ;
.LBB0_1:                               ; a ciklusfeltétel kiértékelése
    mov EAX, DWORD PTR [ESP + 36]      ; EAX-ba tesszük a ciklusváltozót
    cmp EAX, DWORD PTR [ESP + 32]      ; összehasonlítjuk n-el
    jl .LBB0_2                        ; ha kisebb, a ciklus megy még egy kört, ugrunk
```

A kódból rövid gondolkodás után leszűrhető, hogy a függvény lokális változói hol vannak a memóriában. C nyelvről és lokális változókról lévén szó, nyilván a stack-be kerülnek, mely tetejét az x86 architektúrában az ESP regiszter mutatja. A stack tetejéhez képesti pozíciók:

Változó:	i	n	a	b	f
Memóriacím:	ESP+36	ESP+32	ESP+28	ESP+24	ESP+20

Most vegyük szemügyre a keletkezett assembly kódot. Látható, hogy az `add` utasítás két operandusú, ami a CISC architektúrákra jellemző. Egy másik CISC jellemző, hogy vannak memóriareferens utasítások: az `add` (lásd második operandus), az `inc`, és a `cmp` is ilyen. Később látni fogjuk, hogy egy RISC processzorban ez nem megengedett. Már ezen a rövidke példán is felfedezhetjük, hogy az utasításkészlet redundáns, hiszen az inkrementálás (`inc`) megvalósítható lenne az összeadással (`add`) is, de kényelmi okokból ez a művelet is kapott egy saját utasítást. Érdemes még egy pillantást vetni a feltételes ugrásra, ami két lépésben történik. Először a `cmp` elvégzi az összehasonlítást, és beállít egy (belső) feltétel kódot az eredménynek megfelelően. Magát az ugrást a `jl` utasítás végzi el, ha a (belső) feltétel kód értéke ”kisebb” (azaz az előző összehasonlításból az jött ki, hogy az első operandus kisebb a másiknál). Azaz az x86 architektúra feltétel kódra alapozott feltételes ugrást támogat.

Vegyük észre, hogy a fenti assembly kód nem éppen optimális, hiszen a ciklus belsejében szinte minden egyes művelet a memóriához fordul. A `-O3` kapcsoló megadásával engedélyezve az optimalizálást olyan kódot kapunk, melyben a függvényünk által használt pár változót a CPU regiszterei tárolják, így a ciklusban egyetlen memóriaművelet sem marad.

[1] parancssor: `clang -c fib.c -emit-llvm` (az optimalizálás engedélyezéséhez szükséges a `-O3` opció is)
[2] parancssor: `llc fib.o -march=arm` (arm mellett x86/alpha/ppc32 opciókat is használunk)

```
    xor EDX, EDX      ; A ciklusváltozó nullázása -> így gyorsabb
.LBB0_2:
    mov ESI, EDI      ; f-be tesszük az a változót
    add ESI, ECX    ; f-hez hozzáadjuk a b változót
    mov EDI, ECX      ; a-be tesszük b értékét
    mov ECX, ESI      ; b-be tesszük f értékét
    inc EDX           ; növeljük a ciklusváltozót
    cmp EDX, EAX      ; összehasonlítjuk n-el
    jl .LBB0_2       ; ha kisebb, ugrunk a ciklusmag elejére
```

A változók regiszterekhez rendelése a kód értelmezése után könnyen kitalálható, azt az alábbi táblázat tartalmazza.

Változó:	i	n	a	b	f
Regiszter:	EDX	EAX	EDI	ECX	ESI

Az optimalizált kódon is látszanak a kétoperandusú műveletek és a feltételkódra alapozott elágazás, azonban memóriareferens utasításokra most nem volt szükség.

3.5.2. Megvalósítás ARM architektúrán

Ha ARM architektúrára fordítunk, optimalizálás nélkül az alábbi assembly programot kapjuk.

```
    mov r4, #0           ; Kinullázzuk az r4-et,
    str r4, [sp, #16]    ; ... majd a ciklusváltozóba írjuk
    b   .LBB0_2          ; feltétel nélkül ugrunk a ciklusfeltétel kiértékelésére
.LBB0_1:
    ldmib sp, {r0, r1}   ; a és b változó betöltése r0-ba és r1-be, egy lépésben!
    add r0, r1, r0       ; ...elvégezzük az összeadást (a+b)
    str r0, [sp]         ; ...az eredményt f-be írjuk
    ldr r1, [sp, #4]     ; beolvassuk b-t
    str r1, [sp, #8]     ; ...beírjuk a-ba
    ldr r2, [sp]         ; beolvassuk f-et
    str r2, [sp, #4]     ; ...beírjuk b-be
    ldr r1, [sp, #16]    ; kiolvassuk i értékét
    add r0, r1, #1       ; ...hozzáadunk 1-et
    str r0, [sp, #16]    ; ...visszaírjuk a megnövelt értéket
.LBB0_2:
    ldr r0, [sp, #12]    ; betöltjük n sértékét
    ldr r1, [sp, #16]    ; betöltjük a ciklusváltozó értékét
    cmp r1, r0           ; ...összehasonlítjuk
    blt .LBB0_1          ; ...ha kisebb volt, ugrás a ciklusmag elejére
```

A stack tetejét az ARM az sp regiszterben tárolja. A változók a stack tetejéhez képest az alábbi táblázatnak megfelelően helyezkednek el a memóriában (az eltolást az ARM ”#” karakterrel jelzi).

Változó:	i	n	a	b	f
Memóriacím:	[sp #16]	[sp #12]	[sp #8]	[sp #4]	[sp]

Szemmel látható, hogy a RISC szervezésű ARM kód hosszabb, mint amit x86-ra kaptunk. Mivel utasítások csak elemi műveletekhez tartoznak, már a ciklusváltozó nullázása is két utasítást igényel (egy regiszterbe nullát írunk, majd a regiszter értékét eltároljuk a memóriába). Ugyanígy a ciklusfeltétel kiértékeléséhez is eggyel több lépésre van szükség, hiszen a `cmp` csak regiszter operandusokat fogad el. Cserébe viszont az aritmetikai műveletek három operandusúak, aminek a hozadékát az optimalizált kódon fogjuk látni. A redundanciamentesség jegyében inkrementáló utasítás nincs, az inkrementáláshoz a ciklusváltozót az `add` növeli meg eggyel. Az ARM is feltételkódokra alapozott elágazás-kezelést használ.

Az optimalizálás engedélyezésével lényegesen rövidebb kódot kapunk.

```
    mov r3, #0       ; a ciklusváltozó nullázása (i=0)
.LBB0_2:
    add r1, r0, r2   ; f = a+b
    mov r0, r2       ; a = b
    mov r2, r1       ; b = f
    add r3, r3, #1   ; i = i + 1
    cmp r3, r12      ; i és n összehasonlítása
    blt .LBB0_2      ; ...ha kisebb volt, ugrás a ciklusmag elejére
```

Ez a kód érdekes módon még az x86 kódjánál is tömörebbre sikerült, ami elsősorban a háromoperandusú összeadásnak köszönhető (az ”f=a+b” C programsorhoz x86-ban 2, ARM-ban 1 utasítás kell). A változók és regiszterek összerendelését az alábbi táblázat adja meg.

Változó:	i	n	a	b	f
Regiszter:	r3	r12	r0	r2	r1

3.5.3. Megvalósítás PowerPC architektúrán

A 32-bites PowerPC utasításkészlethez (PPC32) tartozó optimalizált assembly kód lényegét tekintve elég közel áll az ARM kódhoz.

```
    li    5, 0          ; i = 0
.LBB0_2:
    add   4, 7, 6       ; f = a + b
    mr    7, 6          ; a = b
    mr    6, 4          ; b = f
    addi  5, 5, 1       ; i = i + 1
    cmpw  0, 5, 3       ; i és n összehasonlítása, az eredményt 0-ban tároljuk
    blt   0, .LBB0_2    ; az eredménytől függően ugrik a ciklusmag elejére
```

A változókat a fordító az alábbi regiszterekhez rendelte:

Változó:	i	n	a	b	f
Regiszter:	5	3	7	6	4

Kicsit szokatlannak tűnhet, hogy a PPC-ben a regisztereket csupán egy számmal jelölik, ami nem éppen a könnyű olvashatóságot szolgálja. Ismét visszaköszönnek a RISC jellemvonások: a három operandusú műveletek és az inkrementáló utasítás hiánya. Az optimalizált kódból nem látszik, de természetesen a PPC-ben is csak regiszterek lehetnek az utasítások operandusai (a load/store utasításoktól eltekintve). Említést érdemel a kissé rendhagyó elágazás-kezelés. Az összehasonlító műveletek eredményét ugyanis a programozó egy feltétel regiszterbe elmentheti (ebből 8 áll rendelkezésre), és a feltételes elágazáskor (`blt`) meg kell adni, hogy melyik feltétel regisztert vegye figyelembe (ebben az esetben a 0-ás feltétel regisztert használtuk). Ez a megoldás mind az x86, mind az ARM megoldásánál rugalmasabb, hiszen ott csak egyetlen (az utolsó) összehasonlítás értékét tárolja a CPU. Annak ellenére, hogy több összehasonlítás eredménye is tárolható, mégsem hívjuk ”feltétel regiszterre” alapozott elágazás-kezelésnek. Ezt a titulust csak akkor érdemelné ki, ha a speciális feltétel regiszterek helyett általános célú regisztereket is használhatnánk erre a célra.

3.5.4. Megvalósítás Alpha architektúrán

Az optimalizált Alpha kód is tipikus RISC tulajdonságokkal bír.

```
    mov     $31,$2       ; ciklusváltozó nullázása
$BB0_2:
    addq    $3,$1,$4     ; f = a + b
    mov     $1,$3        ; a = b
    mov     $4,$1        ; b = f
```

```
addl    $2,1,$2      ; i = i + 1
cmplt   $2,$0,$5     ; i<n? eredmény: $5-be kerül
bne     $5,$BB0_2    ; $5 értékétől függően megy még egy kört a ciklus
```

A regiszterek és a lokális változók kapcsolata:

Változó:	i	n	a	b	f
Regiszter:	$2	$0	$3	$1	$4

A kódból az Alpha processzorok két érdekes tulajdonságát szűrhetjük le. Egyrészt furcsának tűnhet a nullázás, amihez kihasználjuk, hogy az $31-es regiszter konstans 0-át tartalmaz. Erre egy 64 bites processzornak azért van szüksége, mert a nullázás egy gyakori művelet, és egy 64 bites 0 érték utasításba kódolásánál és végrehajtásánál hatékonyabb, ha a 32 regiszter közül az utolsóra hivatkozunk.

Az elágazás kezelését illetően itt végre találkozhatunk a feltétel regiszter használatával. A `cmplt` elvégzi az összehasonlítást, és ha teljesül a "<" reláció, akkor az $5-ös általános célú regiszterbe egy 1-est ír, ha nem, akkor pedig 0-át. A `bne` utasítás végzi el az ugrást, amennyiben $5≠ 0.

3.6. Irodalomjegyzék

[3.1] Rudolf Eigenmann, David J. Lilja. *Von Neumann Computers.* Wiley Encyclopedia of Electrical and Electronics Engineering, 1999.

[3.2] David A. Patterson, John L. Hennessy. *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, 4th Edition, 2006.

[3.3] Yuan Wei Bin Huang Amit K. Naidu, *CISC vs. RISC*, `http://www.cs.virginia.edu/%7Eskadron/cs654/cs654_01/slides/amit.ppt`.

[3.4] Hannibal, *RISC vs. CISC: The Post-RISC Era (A historical approach to the debate)*, `http://staffweb.worc.ac.uk/DrC/Courses%202004-5/Comp3070/Resources/hannibal.pdf`.

[3.5] *Reduced instruction set computing*, `http://en.wikipedia.org/wiki/Reduced_instruction_set_computing`.

[3.6] *Intel 64 and IA-32 Architectures Software Developer's Manuals*, `http://www.intel.com/products/processor/manuals`.

[3.7] *Intel 80386 Reference Programmer's Manual*, `http://pdos.csail.mit.edu/6.858/2010/readings/i386/toc.htm`.

II. rész

A perifériák

4. fejezet

Perifériakezelés

4.1. A perifériák sokfélesége

A számítógépek perifériáit többféle szempont szerint csoportosíthatjuk. Attól függően, hogy az adatok a perifériából áramlanak a számítógép felé (pl. egér, billentyűzet, lapolvasó, mikrofon, stb.), vagy fordítva (pl. megjelenítő, hangszóró, stb.) megkülönböztetünk *bemeneti* illetve *kimeneti* perifériákat. Egyes perifériák esetén ez nem egyértelmű (mint pl. a printer tartalmaz nyomógombokat is, ami bemeneti, és festék szórására alkalmas eszközt is, ami kimeneti), ilyenkor az eszközt annak domináns irányultsága szerint jellemezzük. Vannak perifériák, melyek nem rendelkeznek domináns irányultsággal, ezek egyszerre be- és kimeneti eszközök (pl. diszk, hálózati kártya, stb.)

Az is lényeges szempont, hogy az adatok bevitelét ill. a kimenő adatok befogadását *ember* vagy *gép* végzi-e. Az ember, mint partner lényegesen lassabb adatbevitelre, ill. befogadásra képes, mint a gép.

Vannak perifériák, melyek *érzékenyek a késleltetésre*. Ezek nem képesek tetszőleges várakozásra, a nekik szánt adatokat az előírt határidőt betartva el kell küldeni, ill. a tőlük származó adatokat az előírt határidő előtt. A határidő lekésése után az adat aktualitása elvész. Ebbe a körbe tartoznak az olyan interaktív eszközök, mint pl. a hangkártya: a hangmintáknak meg kell érkeznie a kijátszás pillanatáig, különben zavaró minőségromlás (kattanás) lép fel.

Egyes perifériák *sávszélesség igényesek*, vagyis rövid idő alatt kifejezetten nagy mennyiségű adatot cserélnek, melyek feldogozása, vagy előkészítése kihívást jelenthet.

Az alábbi táblázat néhány elterjedten használt perifériát vesz sorra.

	Partner	**Bemeneti/kimeneti**	**Adatforgalom**
Billentyűzet:	Humán	Bemeneti	100 byte/sec
Egér:	Humán	Bemeneti	200 byte/sec
Hangkártya:	Humán	Kimeneti	kb. 96 KByte/sec
Printer:	Humán	Kimeneti	kb. 200 KByte/sec
Grafikus megjelenítő:	Humán	Kimeneti	kb. 500 MByte/sec
Ethernet hálózati interfész:	Gép	Ki/Be	kb. 12.5 MByte/sec
Diszk (HDD):	Gép	Ki/Be	kb. 50 MByte/sec
GPS:	Gép	Bemeneti	kb. 100 byte/sec

Talán ebből a hozzávetőleges táblázatból is látható, hogy nem lesz könnyű olyan egyszerű, univerzálisan használható receptúrát találni, ami minden létező periféria kiszolgálására egyformán alkalmas.

A következő fejezetekben megtárgyaljuk,

- hogy hogyan tud a processzoron futó program adatátvitelt kezdeményezni;
- hogy hogyan tud a számítógép egy perifériája adatátvitelt kezdeményezni;
- hogy hogyan kell hatékonyan levezényelni az adatátvitelt, hibamentesen és a lehető legkisebb processzor terheléssel;

- végül, hogy hogyan célszerű a perifériákat a processzorral, illetve a számítógép többi részegységével összekötni.

4.2. Program által kezdeményezett kommunikáció

A memória felé a processzor a memóriabuszon keresztül kommunikál. Ha a processzor olvasni vagy írni szeretne a memóriából, kiteszi a buszra a címet, és beolvassa a buszról a memória által erre válaszként rátett adatot (olvasás esetén). Írási művelet esetén a memóriacímmel együtt az adatot is a buszra teszi, aminek hatására a memória a kívánt címre beírja az adatot.

A perifériakezelés is ehhez hasonlóan zajlik. Ha a futó programból meg akarunk szólítani egy perifériát, akkor annak kell, hogy legyen címe, amivel hivatkozhatunk rá. A processzor megcímzi azt a perifériát, amellyel kommunikálni kívánunk, majd kiteszi, vagy elveszi az adatot attól függően, hogy írási vagy olvasási műveletről van-e szó.

Annak függvényében, hogy a memóriatartalom címzése és a perifériák címzése mennyire különül el egymástól, két esetet különböztetünk meg, ezek pedig:

- a külön I/O címtartományt használó perifériakezelés,
- a memóriára leképzett perifériakezelés.

A bővebb tárgyalás előtt fontos megjegyezni, hogy egy perifériának több címe is lehet, külön címet szokás rendelni a státusz lekérdezéséhez, a parancsok küldéséhez, az adatok kiolvasásához illetve küldéséhez. A régi, hagyományos (PS/2 csatlakozóval ellátott) PC-s billentyűzethez például 2 cím tartozik: a `0x64`-es címről olvasva megkapjuk a státuszt (pl. volt-e leütés), ugyanerre a címre írva parancsot adhatunk (pl. Caps Lock beállítása); a másik, `0x60`-as címről pedig a lenyomott billentyű kódját lehet lekérdezni.

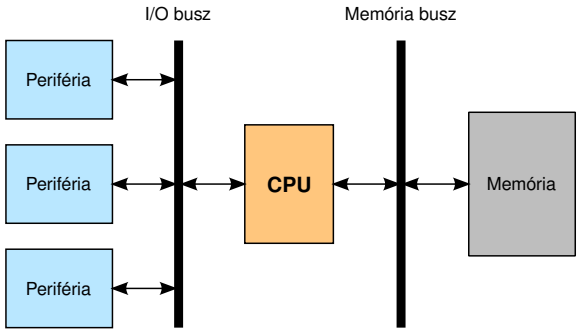
4.2.1. Külön I/O címtartományt használó perifériakezelés

Számos utasításkészlet architektúra 2 független címteret használ: egyiket a memória, a másikat pedig a perifériák címzésére. Mindkét címtér 0-val kezdődik, de nem feltétlenül ugyanott ér véget (az x86 processzorok 32 bites tagjainál a memóriacímek 32, a perifériacímek 16 bitesek). Ennek következtében például létezik `0x60`-as memóriacím és `0x60`-as perifériacím is egyszerre. Azt, hogy egy programból kiadott cím a memóriára, vagy a perifériákra vonatkozik-e, az határozza meg, hogy milyen utasítást használunk az adatátvitelre. A külön I/O címtartományt használó processzorok ugyanis külön utasításkészlettel rendelkeznek a perifériákkal történő adatcserére, és a memória elérésére.

Ha a `0x60`-as perifériacímről szeretnénk az `R0` regiszterbe olvasni (a lenyomott billentyű kódját), akkor azt az `R0 ← IO[0x60]`, utasítással tehetjük meg, míg a memória `0x60`-as címéről az `R0 ← MEM[0x60]` utasítással olvashatjuk be az adatot. Ezek külön utasítások, külön gépi kód tartozik hozzájuk.

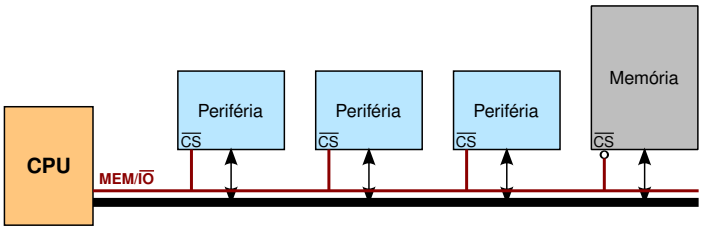
A processzor és a perifériák összekötésére két lehetőség adódik.

1. A memória és a perifériaműveletekre *szeparált busz* áll rendelkezésre. Ebben az esetben a processzor két fizikailag is különálló busszal rendelkezik, az egyiken kizárólag a memóriával, a másikon csak a perifériákkal kommunikál (4.1. ábra).



4.1. ábra. Szeparált I/O és memóriabusz

2. A memória és a perifériaműveletekre közös, *multiplexált busz* is kialakítható. Ilyenkor az a memória író/olvasó és a periféria író/olvasó műveletek ugyanazt a buszt használják. Azt, hogy az író/olvasó művelet a memóriának, vagy a perifériáknak szól-e, a buszon egy külön erre a célra fenntartott vezérlőjel határozza meg (4.2. ábra). Így működik az x86 architektúra is.



4.2. ábra. Külön I/O címtartomány használata multiplexált busszal

4.2.2. Memóriára leképzett perifériakezelés

Néhány digitális alkatrésszel (dekóderrel, komparátorral, kapuval, stb.) egyszerűen megoldható, hogy egyes kitüntetett memóriacímekre kezdeményezett adatátviteli kérésekre ne a memória, hanem egy periféria válaszoljon. Építhetünk például a processzor mellé egy egyszerű logikai hálózatot, amely a `0x60` cím megjelenésekor letiltja a memóriát, és kijelöli a billentyűzetet adatátvitelre. A program ilyen esetben a memóriakezelő utasításokkal tud a perifériával kommunikálni. C nyelven a

```
char* p = 0x60;
int x = *p;
```

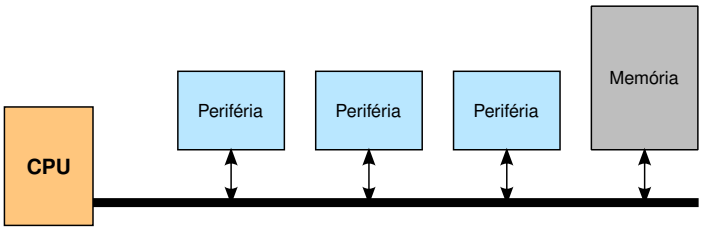
utasítások hatására a `0x60`-as memóriacímről az `x` változóba olvasunk adatot. Ha ezt a címet az imént tárgyalt hardver elemekkel a billentyűzethez rendeltük, akkor `x`-be a beolvasott bájt nem a memóriából, hanem a billentyűzettől fog érkezni.

Általánosságban elmondható, hogy memóriára leképzett perifériakezelés esetén a memória címtartomány egy része a perifériákhoz van rendelve, az arra vonatkozó memóriaműveleteket a perifériák válaszolják meg. Ennek a megoldásnak az a hátulütője, hogy a perifériák számára fenntartott címeken a memória tartalma nem elérhető, vagyis a memória teljes adattárolási kapacitása valamivel kisebb. A megoldás előnye, hogy a processzorok jellemzően sokkal több és rugalmasabb memóriakezelő utasítással rendelkeznek, mint perifériakezelő utasítással.

Memóriára leképzett perifériakezelés szempontjából a processzorokat két csoportra oszthatjuk.

- Vannak processzorok, melyek kizárólag így képesek a perifériákkal kommunikálni, mivel egyáltalán nincsenek perifériakezelő utasításaik, és ennélfogva periféria címtartományuk sem (külön busz sincs, 4.3. ábra). Ebbe a csoportba tartozik az összes RISC processor, ARM, SPARC, POWER, stb.

- Vannak olyan processzorok is, melyek rendelkeznek külön perifériakezelő utasításokkal, és periféria címtérrel is. Ez a lehetőség azonban nem zárja ki, hogy egyes perifériákat memóriára leképezve kezeljenek. Ilyen az x86 architektúra is.



4.3. ábra. Közös busz a perifériák és a memória számára

4.3. Periféria által kezdeményezett kommunikáció

Az előző fejezetben láttuk, hogy a futó program milyen egyszerűen képes egy perifériának adatot küldeni, illetve attól adatot beolvasni. Számos helyzetben azonban célszerű lehetőséget biztosítani a periféria számára is, hogy közlendőjét a lehető leghamarabb jelezhesse, ne kelljen megvárnia, amíg a futó program az állapotát lekérdezi. A billentyűzet példájánál maradva: jó lenne, ha önállóan tudná jelezni a billentyűzetleütés bekövetkeztét, és nem kellene folyamatos státusz lekérdezésekkel lassítani a program futását.

Ilyen helyzetekre való a *megszakításkezelés* (interrupt).

4.3.1. A megszakítások működése

Szinte minden processzor rendelkezik interrupt lábbal, melyre (durván leegyszerűsítve) ráköthető a periféria megfelelő jelzése. Amikor a periféria jelez, a processzor "érzékeli" a megszakításkérést, befejezi az aktuális utasítás végrehajtását, elmenti az állapotát (regisztereket, utasításszámlálót), majd egy ugrást hajt végre a megszakításkezelő szubrutinra. Ez a szubrutin elveszi a perifériától az adatot, majd a processzor állapotának visszaállítása után a program ott folytatódik, ahol megszakadt. A processzornak csak annyi figyelmet kell a perifériára fordítania, amennyit az adatok átvétele kíván, ha a periféria nem tevékeny, akkor semennyit.

A megszakításkezelő szubrutint a processzor a memória egy előre definiált címén keresi, és az operációs rendszer feladata, hogy a rendszer indulásakor a megszakítás kiszolgálásához szükséges utasítássorozatot erre a bizonyos címre helyezze.

4.3.2. Vektoros megszakításkezelés

A gondot az jelenti, hogy tipikusan sokkal több a periféria (és az általuk a processzornak szánt jelzés), mint ahány interrupt lába van a processzornak. Nem ritka, hogy egy processzornak csak egy, vagy esetleg kettő interrupt bemenete van (x86 és ARM). Ilyenkor az összes interrupt forrást a rendelkezésre álló néhány (a továbbiakban legyen egyetlenegy) interrupt lábra kell kötni.

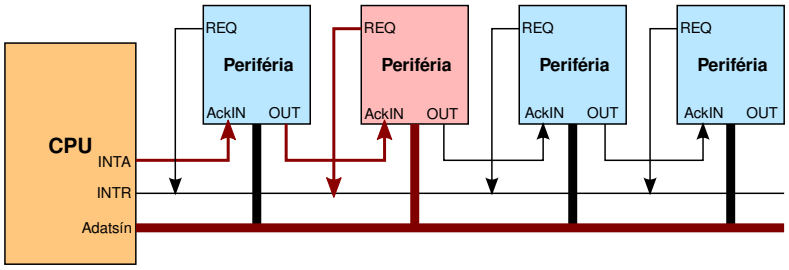
Ez a megoldás több kérdést is felvet: ha minden jelzés egyetlen pontba fut be, akkor a processzor honnan tudja, hogy melyik periféria jelzett? És vajon mi történik, ha több periféria is pont egyszerre kér megszakítást? Három elterjedt módszert mutatunk be a kérdések megválaszolására.

1. **Polling**: a processzoron futó program az megszakításkérés beérkezésekor minden egyes perifériát körbekérdez, hogy onnan származott-e az interrupt. Sok periféria esetén ez nyilván eltarthat egy ideig. Vegyük észre, hogy a körbekérdezés sorrendje egyben a megszakítások prioritását is meghatározza: ha többen kértek megszakítást egyszerre, akkor a program azt kezeli le hamarabb, amelyikről hamarabb szerzett tudomást.

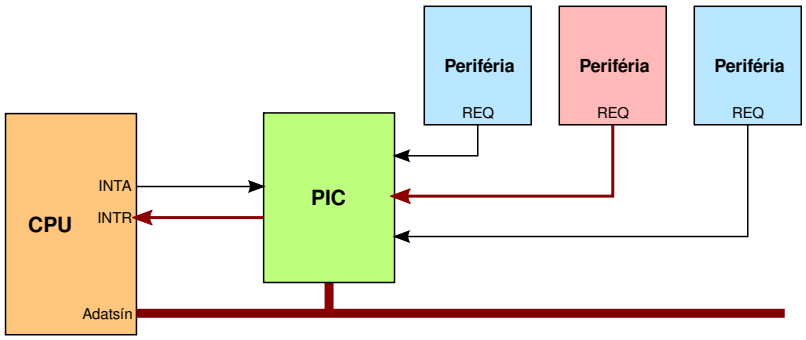
2. **Daisy chaining**: a processzor az interrupt kérés beérkezésekor ránéz az adatsínre, és az ott talált azonosítóból tudja, hogy melyik periféria jelzett. Ezt többek között a 4.4. ábrán látható módon lehet megvalósítani. A jelezni kívánó periféria az igényét a processzor INTR lábán jelenti be. A processzor ennek hatására kiadja az INTA jelet, amit a sorba kötött perifériák közül az első megkap. Ha nem az generálta a megszakítást, akkor ezt a jelet továbbadja következőnek, amely ugyanígy jár el. Amikor az INTA az interruptot generáló perifériához ér, az az adatbuszra helyezi az azonosítóját, és az INTA jelet nem is adja már tovább (4.4. ábra). A processzor az adatsínen talált azonosítóból tudni fogja, hogy melyik interrupt kezelő szubrutint kell meghívnia. Azt a táblázatot, melyben az egyes azonosítókhoz rendelt meghívandó függvény mutatók találhatók, *interrupt vektor táblának* nevezik.

   A daisy chain-ben a perifériák sorrendje a prioritást is meghatározza. Ha egyidejűleg többen is jelezni akartak a processzor felé, a sorban előbbre álló kapja meg előbb a jogot ahhoz, hogy az azonosítóját a buszra tegye, és így kiszolgálást kapjon. A daisy chain-t nagyon egyszerű megvalósítani és bővíteni, de a prioritás kötött, és ha sok a periféria, a sor végén állók ritkán jutnak kiszolgáláshoz.

3. **Interrupt vezérlő** használata (PIC: Programmable Interrupt Controller): az interrupt vezérlőhöz több periféria interrupt kéréseit is hozzá lehet kötni. Az interrupt vezérlőben el vannak tárolva az egyes bemenetekre kötött perifériák azonosítói. Amikor az egyik periféria jelez, az interrupt vezérlő szól a processzornak (INTR), majd az adatsínre helyezi a kérést indítványozó periféria azonosítóját (4.5. ábra).



4.4. ábra. Interrupt alapú perifériakezelés daisy chain-el



4.5. ábra. Interrupt alapú perifériakezelés interrupt vezérlővel

Egyidejű interrupt kérések esetén az interrupt vezérlő a daisy chain-nél kifinomultabb prioritás kezelést is alkalmazhat (pl. körbenforgó elv szerint: mindig a legutóbb kiszolgáltnak a legkisebb a prioritása). Így működik a megszakításkezelés az x86 architektúrában is.

(Az ARM architektúrában is hasonló megoldást alkalmaznak, de ott a PIC nem adja át rögtön az érintett periféria azonosítóját a processzornak, azt a megszakításkezelő szubrutin perifériakezelő utasításokkal kérdezheti le a PIC-től. Az elv ugyanaz.)

4.3.3. Megszakításkezelés többprocesszoros rendszerben

Többprocesszoros rendszerekben felmerül a kérdés, hogy az megszakítás kéréseket melyik processzor dolgozza fel. Az egyik lehetséges megoldás, hogy minden egyes megszakítás egyetlen processzorhoz fut be (jellemzően ahhoz, amelyik a rendszerindításért is felelős). Az ilyen rendszer azonban nem skálázható, hiszen hiába van több processzorunk, ha a feldolgozható megszakítások számának egyetlen processzor teherbírása szab határt.

Ezért a többprocesszoros rendszerekben a megszakítások feldolgozása két lépésben történik:

- A perifériák felől beérkező megszakítások először egy *megszakításelosztóhoz* futnak be. A megszakításelosztó feladata, hogy a megszakításokat a processzorok között szétossza (interrupt routing). Azt, hogy melyik megszakítás melyik processzorhoz kerüljön, az operációs rendszer határozza meg, amikor a rendszer indításakor felkonfigurálja a megszakításelosztót. Pl. be lehet úgy állítani, hogy a diszkek megszakításkérését az egyik, a beviteli eszközökét egy másik processzor szolgálja ki, de általában lehetőség van arra is, hogy körben forgó elven a beérkező megszakítás mindig a soron következő processzorhoz kerüljön.

- Minden processzor rendelkezik egy saját, lokális megszakítás vezérlővel. Ide futnak be az interrupt elosztótól kapott megszakítás kérések, de bizonyos perifériák az interrupt elosztó megkerülésével közvetlenül ezen jelezhetnek megszakítást. A lokális megszakítás vezérlőnek még egy fontos szerepe van: segítségével a rendszer processzorai megszakítást kérhetnek egymástól, lehetőséget teremtve a processzorokon párhuzamosan futó programok között kommunikációra.

Ilyen kétlépéses megoldást használ az Intel APIC (a PC-kben) és az ARM GIC interrupt vezérlője (a mobil eszközökben).

Interrupt forrás	Tipikus interrupt/s	Maximális interrupt/s
10 Mbps Ethernet	812	14880
100 Mbps Ethernet	8127	148809
Gigabit Ethernet	81274	1488095

4.1. táblázat. Hálózati adapterek megszakításterhelése teljes sebességű hálózati forgalommal

4.3.4. A megszakításkezelés előnyei és hátrányai

Az eddig leírtak alapján a megszakítások támogatását és használatát hasznos technikának könyvelhetjük el, hiszen csökkenti a processzor terhelését azáltal, hogy az állandó (és időtrabló) lekérdezgetések helyett a periféria maga is tudja jelezni a fontos eseményeket.

Előfordulhat azonban, hogy a megszakításkérések olyan gyorsan érkeznek egymás után, hogy azzal a processzor már nem bír. Ha a processzor az órajelciklusainak túl nagy hányadát fordítja interrupt feldolgozásra, a futó program túl sokszor szakad meg, és lassan fog futni. Vannak biztonságkritikus alkalmazások, melyekben ez nem engedhető meg. Az első holdra szállás alkalmával, 1969-ben, a radar egységből érkező adatok áradata miatt az Apollo 11 holdraszálló egységének manőverezésért felelős processzora is interrupt túlterhelésnek esett áldozatul. A normál feladatok végrehajtása (az egység irányítása) a processzoridő 85%-át vitte el, a bekapcsolva felejtett radar megszakításai 13%-ot, az űrhajósok által 5 perccel a leszállás előtt kiadott távolságlekérdezés pedig 10%-ot igényelt – volna, de ez már meghaladta a 100%-ot. Szerencsére a számítógép nagyon jól fel volt készítve ilyen esetekre, és az alapfunkció (irányítás) számára biztosította a prioritást, annak az árán is, hogy sok megszakításkérést feldolgozatlanul eldobott.

Talán a legnagyobb mennyiségű megszakítást a hálózati eszközök generálják, minden beérkezett csomagot jeleznek. A 4.1. táblázatban a maximális sebességű hálózati forgalom mellett másodpercenként kiváltott interruptok számát láthatjuk. A középső oszlop vonatkozik a ”normál” esetre, amikor hagyományos internet csomagokat (1500 bájt) szállít a hálózat. Egy meghibásodott, vagy rosszindulatú hálózati szereplő azonban teljes intenzitással küldözgetheti az Ethernet által megengedett legkisebb csomagokat (64 bájt), ezzel sokkal több megszakítást kiváltva. Gigabit Ethernet esetén ez közel 1.5MHz interrupt frekvenciát jelent. Ha csak 1000 órajelbe telik lekezelni minden megszakítást, akkor is 1.5GHz órajelet vesz el a processzortól pusztán a hálózati forgalom feldolgozása!

A megszakítások által okozott terheléssel tehát foglalkozni kell. A terhelés csökkentésére alkalmas lehetőségek:

- A megszakításkezelő szubrutinnak olyan rövidnek kell lennie, amennyire csak lehetséges.
- Az interrupt rátát egy biztonságkritikus rendszerben célszerű maximálni (a megengedett maximálison felül érkező megszakítások eldobásával).
- A hálózati eszközök gyakran interrupt moderációt alkalmaznak, ami azt jelenti, hogy az eszköz nem minden hálózati csomag érkezésekor generál megszakítást, hanem több eseményt összevár, és ezek feldolgozását egyetlen interrupt jelzéssel kéri a processzortól.
- A futó program kritikus szakaszaiban célszerű lehet a megszakításokat letiltani.

4.4. Adatátvitel

A programozó szempontjából a perifériakezelés célja végső soron az, hogy a perifériákból származó adatok a memóriába kerüljenek (pl. egy C program egy lokális változójába), vagy fordítva.

Ebben a fejezetben két alapvető témakört forgunk megvizsgálni:

- a szereplők eltérő sebességéből adódó adatátviteli hibák kiküszöbölésének módjait,
- valamint magának az adatátvitelnek a módját, vagyis hogy az adategységek milyen úton jutnak a perifériából a memóriába (vagy fordítva).

4.4.1. Forgalomszabályozás

A legritkább esetben fordul elő az a szerencsés helyzet, hogy a periféria pont akkor lesz kész az adat fogadására, amikor a processzor azt küldené, illetve hogy a periféria pont akkor küldi az adatot, amikor a processzor pont el tudja azt venni. Általában a processzor és a perifériák között lényeges sebességkülönbség van. Hogy az adatátvitel során ne fordulhasson elő adatok elvesztése, illetve egymásra várakozás, valamilyen (adat-) forgalom szabályzó technikát kell alkalmazni [31].

Feltétel nélküli adatátvitel

A feltétel nélküli adatátvitel tulajdonképpen a forgalomszabályozás hiánya. A periféria ill. a processzor semmilyen módon nem tudja közölni a másik féllel, hogy van-e új adat, illetve hogy a régi adatot már el tudta-e venni (fel tudta-e dolgozni).

A feltétel nélküli adatátvitel során kétféle probléma léphet fel:

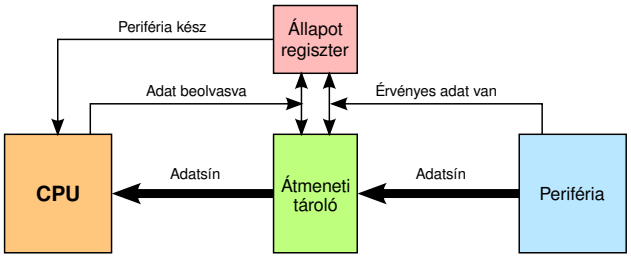
- *Adat egymásrafutás* akkor történhet, ha az adatot előállító szereplő (a processzor vagy a periféria, irányultságtól függően) gyorsabban dolgozik, mint az adatot fogadó. Az adat fogadója olyan lassú, hogy nem sikerült elvennie a neki szánt adatot, mire az előállító egy újabb adattal felülírta azt. Mivel nincsenek az adatcserét támogató, forgalomszabályozó jelzések, erről egyik fél sem értesül.
- *Adathiány* akkor léphet fel, ha az adatok fogadója gyorsabb, mint az adatok előállítója. Az adat fogadója kiolvasná az új adatokat, de azok még nem állnak rendelkezésre, de mivel erről a megfelelő jelzések hiányában nem tud, az általa kiolvasott adat érvénytelen (nem az, aminek szánták).

Feltétel nélküli adatátvitelre példa lehet pl. egy egyszerű kapcsoló, mely a perifériasínen keresztül kiolvasható. A processzor perifériaművelettel lekérdezheti a kapcsoló állását, azt azonban nem tudja megállapítani, hogy az előző lekérdezés óta hányszor nyomták meg. Ha ez nem is cél, akkor nincs szükség fejlett forgalomszabályozásra, a feltétel nélküli átvitel tökéletesen megfelel. Sok egyszerű perifériánál nincs értelme a forgalomszabályozásnak, pl. egyszerű kijelzők (LED-ek) működtetése, stb.

Feltételes adatátvitel

A feltételes adatátvitel során kiemelt szerepe van az adatcsere állapotát tároló regiszternek. Ez az 1 bites állapotregiszter ”1”-et tartalmaz, ha rendelkezésre áll az érvényes adat, és ”0”-t tartalmaz, ha nincs adat. Az adatot küldő oldal (legyen az a periféria vagy a processzor, a kommunikáció irányának függvényében) az állapotregiszter ellenőrzésével meggyőződhet róla, hogy az előző adatot elvitték-e már, így elkerülhető az adat egymásrafutás. Hasonlóan, az adatot váró oldal az állapotregiszter segítségével megállapíthatja, hogy a küldőnek van-e érvényes adata, így elkerülhető az adathiány.

Természetesen nem minden periféria sebességét lehet befolyásolni, ezért lehetséges, hogy a állapotregiszter értékét csak az egyik szereplő tudja figyelembe venni. Pl. a 4.6. ábrán egy olyan esetet láthatunk, amikor a periféria nem, de a processzor tudja figyelni az állapotregisztert. Ezt az esetet *egyoldali feltételes adatátvitel*nek nevezzük.



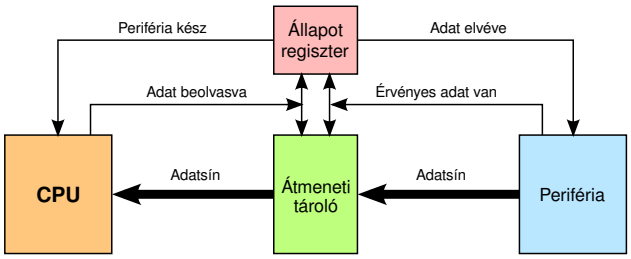
4.6. ábra. Egyoldali feltételes adatátvitel

Az ábra példájában, ha a perifériának kész adata van, azt az átmeneti tárolóban helyezi el, miközben az állapotregisztert 1-be billenti (”Érvényes adat van”). Az állapotregisztert a CPU ellenőrizni tudja (”Periféria kész”), ha 1-es állásban találja, az átmeneti tárolóból ki tudja olvasni a periféria által odatett adatot, ezzel egy időben az állapot regisztert 0-ba állítja (”Adat beolvasva”). Ebben a példában a periféria nem képes az állapotregisztert

figyelembe venni, így, ha a CPU nem elég gyors, előfordulhat, hogy a periféria egy újabb adatot ír az átmeneti tárolóba, mielőtt a CPU a régit kiolvasta volna.

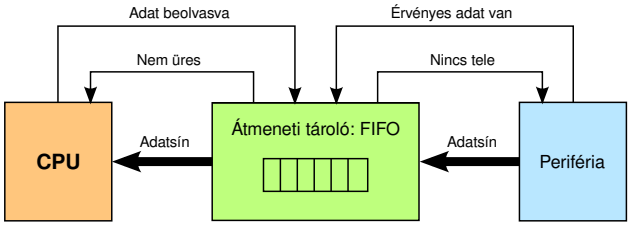
Egyoldali feltételes adatátvitelre példa lehet egy hang bemeneti eszköz (mikrofonra kötött ADC), vagy egy hálózati kártya. Egyik esetben sem lehet befolyásolni a periféria sebességét, hiszen a hang, illetve a hálózati csomagok így is, úgy is jönnek, akár ráér a CPU elvenni az adatot, akár nem.

Ha sebessége befolyásolható, a periféria is hozzáférhet az állapot regiszterhez, és mielőtt az átmeneti tárolóba beírná az új adatot, ellenőrizni tudja, hogy a CPU elvette-e a régit (4.7. ábra). Ha az állapotregisztert 1-be állítva találja, akkor várnia kell, míg ki nem olvassák. Ezt az esetet *kétoldali feltételes adatátvitel*nek nevezzük.



4.7. ábra. Kétoldali feltételes adatátvitel

Gyakran előfordul, hogy a processzor hosszabb-rövidebb ideig nem képes a periféria adatait elvenni, mert közben más, fontosabb tevékenységet is végez. Jó lenne ilyenkor, ha az átmeneti tároló nem csak egy adat tárolására lenne alkalmas, mert akkor a perifériának nem kellene megállnia és a processzort bevárnia. Az átmeneti tároló tulajdonképpen az adat küldés-fogadás mindkét résztvevőjének az ingadozó viselkedése esetén előnyös, az ingadozó rendelkezésre állást a megfelelően nagy átmeneti tároló eltakarja. A 4.8. ábrán az átmeneti tárolót egy FIFO sor valósítja meg. A processzor a FIFO ”Nem üres” jelzését figyeli, ha ez logikai igaz, akkor van a FIFO-ban adat, így van mit elvenni onnan. Az adat elvétele után az ”Adat elvéve” jelzéssel tudja a processzor a FIFO-val tudatni, hogy a sor elején álló adatot elvette, a FIFO ezt tehát törölheti. A periféria a FIFO ”Nincs tele” jelzését ellenőrzi, ha éppen küldendő adata van. Ha ez a jelzés igaz, akkor továbbítja a FIFO-nak az adatot (az ”Adat beírása” jelzés kíséretében), ellenkező esetben meg kell állnia, és bevárnia, hogy hely szabaduljon fel a FIFO-ban, azaz hogy a processzor a korábban betett adatokból legalább egyet kiolvasson.



4.8. ábra. Feltételes átvitel FIFO tárolóval

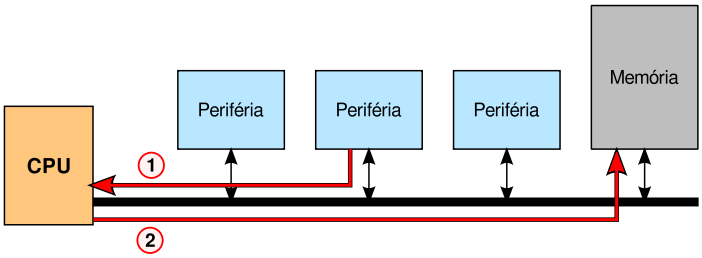
Természetesen ezek a forgalomszabályozási elvek ugyanígy érvényesek akkor is, ha az adatok a processzor felől áramlanak a periféria felé (tehát kimeneti perifériáról van szó).

Az állapot regiszter és az átmeneti tároló nem feltétlenül külön áramköri elemek, jellemzően a perifériák részeit képezik.

4.4.2. Adatátvitel a processzoron keresztül

Az adatok perifériából a memóriába juttatásának legegyszerűbb módja, ha útba ejtjük a processzort. Az első lépésben a program beolvassa az adatot a perifériából a processzor egyik regiszterébe, majd a második lépésben a regiszter tartalmát a memóriába írja (4.9. ábra).

A periféria megszólítása előtt azonban meg kell bizonyosodni afelől, hogy rendelkezik-e átvitelre kész adattal. Feltételes forgalomszabályozás mellett ez annyit jelent, hogy az adat beolvasása nem kezdhető meg, amíg az állapotregiszter értékét a periféria 1-be nem állítja. Az állapotregiszter kezelése szerint kétféle stratégiát különböztetünk meg, a *programozott adatátvitelt*, és a *megszakításra alapozott adatátvitelt*.



4.9. ábra. Adatátvitel a processzoron keresztül

Programozott adatátvitel

Programozott adatátvitel esetén a periféria állapotát periodikus lekérdezéssel, un. *polling*-al figyelik. Az adatátvitelért felelős programrész lényegében egy ciklusból áll, mely addig kérdezgeti az állapotváltozó értékét, míg 1-et nem tartalmaz, amikor is végrehajtja az adat beolvasását, majd második lépésben annak memóriába mentését.

Az alábbi példa a billentyűzettől olvas be egy adatot (`0x64`-es cím), amint az rendelkezésre áll (`IO[0x60]`-as port polling), majd az adatot a memória `0x142`-es címére teszi.

```
ciklus: R0 ← IO[0x64]
        JUMP ciklus IF R0==0
        R0 ← IO[0x60]
        MEM[0x142] ← R0
```

A polling hátránya, hogy ez a lekérdező ciklus megállítja a program futását, mely csak az adatátvitel megtörténte után tud folytatódni. Ha soká jön az adat, akkor a processzor túl sok időt fecsérel el a polling-ra, miközben valami hasznosabbra is fordíthatná az órajelciklusait. Természetesen nem kell feltétlenül folyamatosan monitorozni az állapotváltozót, elég bizonyos időnként ellenőrizni. A lekérdezések között eltelő időt *polling periódusnak* hívják, aminek a helyes megválasztása fontos kérdés, hiszen:

- ha túl gyakoriak a lekérdezések, és lassú a periféria, akkor a processzor túl sok időt fecsérel el szükségtelenül,
- ha túl ritkák a lekérdezések, két lekérdezés között esetleg több változás is bekövetkezhet, tehát a processzor adatokról marad le.

Megszakításra alapozott adatátvitel

Az állapotregiszter beköthető a processzor interrupt rendszerébe is, így minden alkalommal, amikor annak értéke 0-ból 1-be változik, egy megszakítás keletkezik. A processzor a vezérlést átadja a megfelelő megszakításkezelő szubrutinnak, melynek címét az interrupt vektor-tábla megfelelő bejegyzéséből olvassa ki.

A billentyűzet példájánál maradva tegyük fel, hogy a billentyűlenyomás, mint esemény 1-es megszakítást generál. A megszakítás hatására a processzor a vektor-tábla 1-es indexű elemét kiolvassa, függvénymutatóként értelmezi, és végrehajt egy ugrást erre a címre. Ezen a címen helyezzük el a megszakítást lekezelő programot:

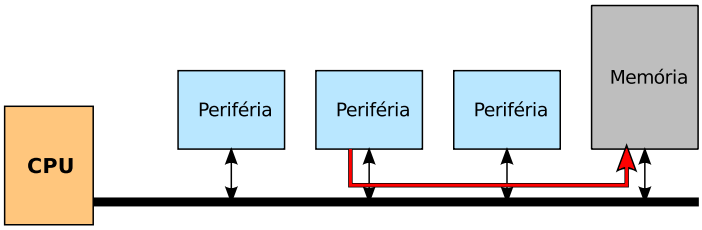
```
billkezelo: R0 mentése
            R0 ← IO[0x60]
            MEM[0x142] ← R0
            R0 visszaállítása
            RETURN
```

Vegyük észre, hogy a megszakításkezelő az R0 regisztert használja az adat ideiglenes tárolására. Annak érdekében, hogy a processzoron futó program semmit ne vegyen észre abból, hogy a billentyűzet miatt megszakították, és lefuttatták a fenti kódot, az R0 regiszter értékét el kell menteni (pl. a verembe, vagy a memóriába, stb.), majd a végén a visszatérés előtt visszaállítani.

A processzoron futó programot a billentyűzet kezelése csak annyira terheli, amennyire muszáj. Ha nem használjuk a billentyűzetet, akkor semennyire.

4.4.3. Adatátvitel a processzor megkerülésével

Az előző fejezetben úgy kerülnek a perifériából a memóriába az adatok (vagy fordítva), hogy azok a processzoron keresztülhaladnak. A program beolvasta az adatokat a perifériából egy regiszterébe, majd második lépésben a regiszterből elhelyezi azt a memóriába, ahol az éppen futó program lokális változói vannak. Ezt a fajta közvetett adatmozgatást el lehet kerülni, így a processzort tehermentesíteni, ha megengedjük, hogy a perifériából az adatok a processzor kikerülésével egyenesen a memóriába kerüljenek (4.10. ábra). Ez a technika különösen a nagy mennyiségű adat mozgatásával járó perifériaműveleteknél (pl. diszk műveletek) fontos.



4.10. ábra. Adatátvitel a processzor megkerülésével

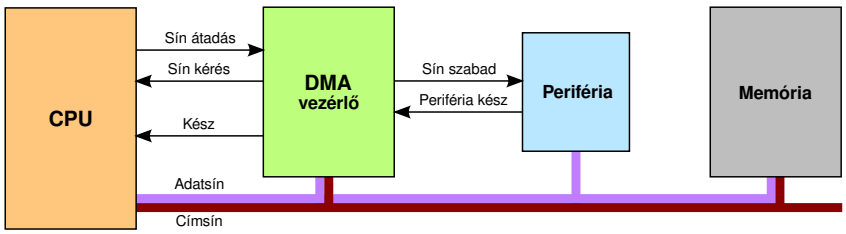
DMA

A DMA (Direct Memory Access) egy olyan mechanizmus, mely lehetővé teszi, hogy a perifériák közvetlenül, a processzor megkerülésével tudjanak adatokat a memóriába beírni, ill. onnan adatokat kiolvasni. A DMA alapú adatátvitelt a perifériához tartozó *DMA vezérlő* koordinálja. Minden perifériának lehet saját DMA vezérlője, de egy DMA vezérlő akár több perifériát is kiszolgálhat egyszerre.

Az adatátvitelt a processzor kezdeményezi azzal, hogy a DMA vezérlő számára eljuttatja az adatátvitel paramétereit (a DMA vezérlő is egy periféria, tehát vagy perifériakezelő utasításokkal, vagy memóriára leképzett perifériakezeléssel lehet vele kommunikálni):

- azt, hogy az átvitt adatokat a memóriába mely címtől kezdődően kell beírni (illetve, kimeneti periféria esetén, az adatokat mely memóriacímtől kezdve kell kiolvasni),
- azt, hogy hány adategységet kell a periféria és a memória között átvinni.

A DMA vezérlő ezután a perifériával kommunikálva sorra lekéri a kért számú adatot. A DMA alapú adatátvitel során a forgalomszabályozásért a DMA vezérlő felelős, tehát a periféria szempontjából a DMA vezérlő ilyenkor a processzor szerepét játssza. Az adatátvitel menetét a 4.11. ábrán láthatjuk.



4.11. ábra. DMA alapú adatátvitel

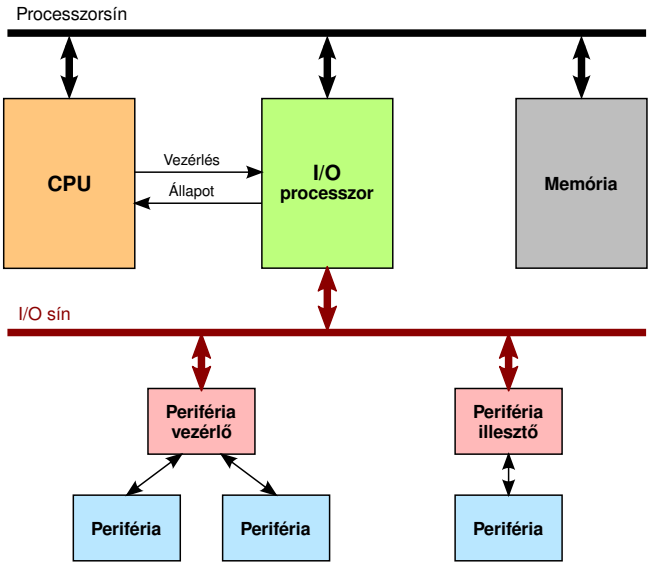
Amint a periféria előállít egy átvihető adatot, azt jelzi a DMA vezérlő felé (”Periféria kész”). A DMA vezérlő ekkor megszerzi a memóriabusz használati jogát a processzortól, majd ha ez megtörtént, a címsínre helyezi a beállított memóriacímet. Ezzel egyidőben a perifériának kiadja a ”Sín szabad” jelzést, mire az az adatsínre helyezi a átvinni kívánt adatot. Ekkor tehát egyidejűleg van a címsínen a memóriacím (a DMA vezérlőnek hála) és az adatsínen a beírandó adat (a perifériának hála), a memóriába írás tehát megtörténik. Ezután a DMA vezérlő inkrementálja a memóriacímet, dekrementálja az átviendő adatok számát reprezentáló számlálóját, és az egész kezdődik előről, amíg a számláló el nem éri a 0-t (tehát a kért számú adategység mindegyike cím-folytonosan a memóriába nem kerül). Ekkor az adatátvitel végét egy interrupt segítségével jelzi a processzor számára.

Fontos, hogy a processzornak csak a DMA átvitel kezdeményezésekor, ill. annak végeztével van az adatátvitellel kapcsolatos dolga. A processzor számára a DMA tehát annyiban jelent könnyebbséget, hogy az adatátvitellel nem adategységenként, hanem adatátviteli blokkonként kell foglalkoznia (pl. diszkek esetén egy teljes szektor beolvasható DMA-val, és csak a teljes szektor beolvasása után kell interrupt kéréssel terhelni a processzort).

Megjegyezzük, hogy vannak saját, beépített DMA vezérlővel rendelkező perifériák is, melyek nem a most megismert rendszerszintű DMA vezérlőt használják. A saját DMA vezérlő is versenybe tud szállni a busz használati jogáért, és képes az adatokat a processzor megkerülésével közvetlenül a memóriába juttatni. Az ilyen, beépített DMA vezérlőt *first-party DMA* vezérlőnek, míg a fentebb megismert rendszerszintű DMA vezérlőt *third-party* DMA vezérlőnek nevezik.

I/O processzor

Az I/O processzor a DMA koncepció továbbfejlesztése. Az I/O processzor, amellett, hogy a DMA vezérlő funkcionalitását birtokolja (tehát képes a periféria és a memória közötti adatmozgatás levezénylésére a CPU közreműködése nélkül), saját utasításkészlettel rendelkezik. Ezekből az utasításokból I/O programokat lehet összeállítani. Az I/O programokkal a perifériák és a memória közötti adatátviteli műveletek sorozatát, illetve az átvitt adatokon elvégzendő egyszerű adatfeldolgozási műveleteket lehet megadni.



4.12. ábra. I/O processzorra alapozott perifériakezelés

A CPU-nak az I/O processzor mellett nem is kell közvetlen, eszköz szintű perifériaműveleteket végeznie. A kívánt műveleteket leíró I/O utasítássorozatot a rendszermemóriába kell írni, majd az utasítássorozat memóriacímét az I/O processzornak eljuttatni. Az I/O processzor az I/O utasításokat maga hívja le a memóriából, és sorra végrehajtja azokat. A teljes I/O program végrehajtásának végét egy interrupt-tal jelzi a CPU felé. Az I/O processzor alapú perifériakezelés további előnye, hogy a CPU minden perifériával kapcsolatos adatátviteli műveletet az I/O utasításokkal tud megadni, tehát az egyes perifériák speciális viselkedését, a különféle perifériák által igényelt változatos kommunikációs eljárásokat a CPU-nak nem kell ismernie, az I/O processzor ezeket eltakarja. Sőt, a perifériáknak nem is kell közvetlenül a processzorsínhez kapcsolódniuk, elég, ha valamilyen módon összeköttetésben vannak az I/O processzorral (pl. egy külön erre a célra fenntartott I/O sínnel, vagy pont-pont kapcsolatokkal). A periféria vezérlő és illesztő feladata, hogy az I/O sín által előírt, közös kommunikációs protokollra fordítsák le a különféle perifériák nyelvét (a periféria vezérlő és illesztő közötti különbség, hogy előbbi több, utóbbi csak egy periféria kezelésére képes).

4.5. Összeköttetések

4.5.1. Az összeköttetések jellemzői

Osztott és dedikált összeköttetések

Az eddigi fejezetek példáiban, ábráin a perifériák egy közösen használt, osztott buszon keresztül kommunikáltak a CPU-val, illetve az I/O processzorral. Természetesen nem ez az egyetlen lehetőség a perifériák és a CPU (vagy az I/O processzor) összekötésére.

Az összeköttetések lehetnek *pont-pont alapú összeköttetések*, vagy *osztott busz alapú összeköttetések*.

A pont-pont összeköttetések esetén a kommunikáló felek (pl. a CPU és egy periféria) között egy dedikált csatorna áll rendelkezésre. Ennek megvannak az előnyei és a hátrányai is:

- A pont-pont összeköttetésekkel nagyobb adatátviteli sebességet lehet elérni, mint osztott buszok használatával, hiszen mivel az összeköttetés a felek sajátja, azon nem kell már perifériákkal osztozni, nem alakul ki versenyhelyzet.
- A pont-pont összeköttetések drágábbak, hiszen minden egyes periféria felé egy-egy külön kommunikációs csatornát kell kialakítani.

Busz alapú összeköttetés esetén a felek közötti kommunikáció egy osztott csatornán zajlik.

- A busz alapú összeköttetések olcsóbbak, hiszen csak egyetlen buszra van szükség, melyet aztán minden periféria közösen használ a kommunikációra.
- A busz, mivel osztott erőforrás, szűk keresztmetszet lehet. Minél több eszköz használja a buszt, annál nagyobb a torlódás kialakulásának esélye (annál többször akadályozzák, tartják föl egymást a kommunikációban).

Mivel mindkét megoldásnak megvan a maga vitathatatlan előnye és hátránya, mindkettőt használják, más és más célra. Pl. a PC esetén, ott, ahol fontos a nagy adatátviteli sebesség, de nem annyira fontos a nagy számú periféria támogatása, pont-pont összeköttetéseket használnak (PCI Express, AGP, HyperTransport), ahol viszont fontos az alacsony költség, valamint a nagy számú, sebességében és viselkedésében heterogén eszköz támogatása, busz alapú összeköttetéseket alkalmaznak (PCI, USB).

Az összeköttetések szélessége

Az összeköttetés *szélessége* az összeköttetést alkotó jelvezetékek száma. Minél több jelvezeték áll rendelkezésre a tényleges adatátvitelre, annál nagyobb lesz az adatátviteli sebesség, viszont a költsége is nagyobb lesz. A nagy sebességű, széles összeköttetésekkel a magas költségen kívül más gond is van: a sok-sok különböző vezetéket lehetetlen úgy elvezetni, hogy a párhuzamos vezetékekre egy időben adott jel annak minden pontján pontosan ”egymás mellett fusson”. A jelvezetékek késleltetése – az elvezetések különbözőségei miatt – nem lesz azonos, tehát a vezetékekre egyszerre ráadott jelek az összeköttetés másik oldalán egymáshoz képest el lehetnek csúszva. Osztott használatú buszok esetén ez a hatás még kifejezettebb, hiszen az egyszerre elindított jelek a különböző perifériákhoz különböző csúszással érkeznek meg. Ez a hatás korlátozza a széles, nagy sebességű buszok fizikai hosszát.

Éppen ezért a trend manapság éppen a keskenyebb összeköttetések irányába mutat (lásd: Parallel ATA - Serial ATA, Parallel port - USB port, stb.). A vezetékek számának csökkentését kétféleképpen érhetjük el:

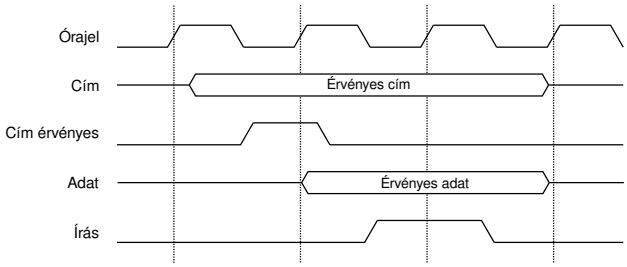
1. vagy az adategység méretének csökkentésével (pl. 16 bites adategységek helyett 8 biteseket használunk, ami kevesebb vezetéket igényel),
2. vagy multiplexált adat- és címvezetékek használatával. Ebben az esetben bizonyos jelvezetékeket adat- és címtovábbításra egyaránt használunk, időben felváltva (ezek a vezetékek egyszer az adatsín, egyszer a címsín szerepét játsszák). A vezetékek száma kétségtelenül csökken, de mivel az adatokat nem lehet tetszőleges időben átvinni (pl. amikor a multiplexált vezetékek címsín szerepet játszanak), az adatátviteli sebesség csökken.

Időzítés

A buszon az adatok időzítése szempontjából megkülönböztetünk szinkron, aszinkron, és önidőzítő átvitelt.

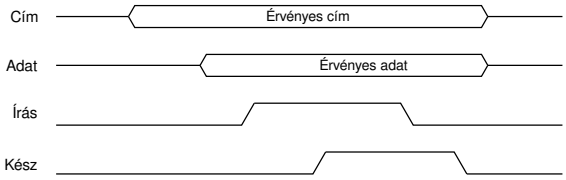
Szinkron buszok esetén a busz órajelét a buszhoz csatlakozó minden eszköz megkapja. A buszon ez az órajel határozza meg az események bekövetkezésének idejét. Ez azt jelenti, hogy a buszon lévő adat-, cím-, és vezérlés információkat a specifikáció szerint meghatározott protokollt követve, de mindig az órajel által meghatározott

ütemben kell írni ill. olvasni. A 4.13. ábrán látható példában a CPU (vagy az I/O processzor) küld egy adategységet egy master eszköznek. A processzor először a címsínre helyezi a címet, az első órajelciklus végén a ”Cím érvényes” vezérlőjel segítségével jelzi, hogy az érvényes, majd az adatsínre helyezi az adatot, és a második órajelciklus végén egy ”Írás” vezérlőjel beállításával jelzi annak érvényességét. A megcélzott eszköz a busz specifikációjában előírt protokoll ismeretében ”számolja az órajeleket”, figyeli az órajel megfelelő élei mentén az adat-, cím- és vezérlőinformációkat, és a kellő pillanatban (amikor az órajel felfutó élénél a ”Write” magas), leolvassa a buszról a neki szánt adatot.



4.13. ábra. Időzítések és forgalomszabályozás szinkron buszon

Aszinkron esetben nincs órajel. A buszon lévő adat-, cím-, és vezérlőinformációk érvényességét a vezérlőjelek határozzák meg. Az adatcsere ilyenkor szorosabb együttműködést igényel. A 4.14. ábrán például a processzor a buszra helyezi a címet és az adatot, majd az ”Írás” jel beállításával jelzi az eszköznek, hogy leolvashatja azokat. Az eszköz a ”Kész” jellel jelzi, hogy végzett az adatok leolvasásával.



4.14. ábra. Időzítések és forgalomszabályozás aszinkron buszon

Az aszinkron időzítés általában lassabb adatátvitelt enged meg (hiszen a felek közötti szorosabb együttműködés miatt több a járulékos kommunikáció: pl. adat kész - átvettem), viszont a szinkron időzítés fizikailag rövidebb buszok kialakítását teszi lehetővé, hiszen az órajelnek a busz teljes hosszát be kell járnia.

Igazán nagy sebességű adatátvitelre azonban sajnos sem a szinkron, sem az aszinkron időzítés nem alkalmas. A gond ugyanaz, mint amit a széles buszoknál láttunk: mindkettőhöz több vezeték kell, és ezért fennáll az összeköttetések szélességének tárgyalásánál leírt jelenség. Azaz, az adat és a órajel (ill. aszinkron esetben az adat érvényességét jelző vezérlőjel) a vezetékek kismértékben eltérő fizikai tulajdonságai miatt egymáshoz képest elcsúsznak, és ha ez a csúszás az adatok tartásidejével összemérhető idejű, akkor adatátviteli hiba keletkezik.

Igazán nagy adatátviteli sebesség emiatt csak soros, *önidőzítő* kommunikációval érhető el. Ebben az esetben egyetlen jelvezeték viszi át az adatot, mégpedig bitenként (ill. differenciális átvitelnél két vezeték szükséges, a jelet a két vezeték közötti feszültségkülönbség reprezentálja). Sem órajel, sem egyéb vezérlőinformáció számára nincs többlet vezeték, így csúszásról sem beszélhetünk. Az önidőzítés azt jelenti, hogy a vevő oldal magából a jelből következtet az órajel frekvenciájára és fázisára, ami a 0-1 átmenetek figyelésével, egy fáziszárt hurok segítségével megvalósítható. Természetesen csak akkor, ha van kellő számú 0-1 átmenet. Annak érdekében, hogy tetszőleges bitsorozat (még a csupa 0, ill. csupa 1 is) átvihető legyen, egy speciális kódolásra van szükség.

Erre a célra az egyik leggyakrabban alkalmazott módszer a PCI Express 1.0 és 2.0 verzióiban, a SATA interfészen, az USB 3.0-ban és a Gigabit Ethernet által is használt 8b/10b kódolás. A 8b/10b kódolás garantálja, hogy:

- nem fordul elő 5-nél hosszabb csupa 0 vagy csupa 1 sorozat,
- a 0 és 1 jelek számának különbsége egy 20 hosszú szakaszon nem lesz 2-nél nagyobb.

Ennek érdekében a 8 bites adatot egy 5 bites és egy 3 bites darabra bontja, majd ezeket egy táblázat alapján 6, illetve 4 bites jelsorozatokra cseréli. A 8b/10b kódolás hátránya, hogy az összeköttetés kapacitása számottevően

csökken, hiszen egy 10 bites szimbólum átvitelére van szükség 8 bitnyi információ közléséhez. A 10 gigabites Ethernet szabvány éppen ezért a sokkal kevésbé pazarló, de elvében ugyanúgy működő 64b/66b kódolást használja, a PCI Express 3.0, valamint az USB 3.1 pedig a még annál is jobb 128b/130b kódolást.

4.5.2. Arbitráció a buszon

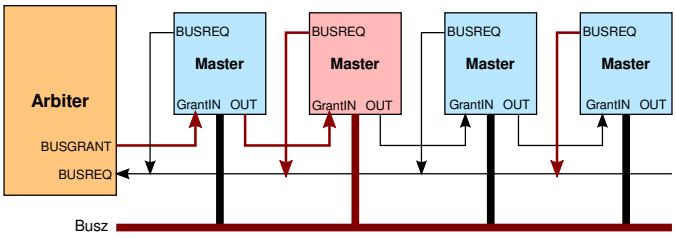
A továbbiakban a busz alapú összeköttetéseket tárgyaljuk részletesen, hiszen a busz, mint osztott közeg használata speciális eljárásokat igényel.

Nem feltétlenül minden periféria képes közvetlenül a buszon keresztül kommunikálni. Például a 4.12. ábrán egy olyan megoldás látható, ahol az egyes perifériák a periféria illesztőn, ill. periféria vezérlőn keresztül csatlakoznak az I/O buszra. A busz használata szempontjából tehát nem is a perifériák, hanem a buszra csatlakozó periféria vezérlők és illesztők a fontos szereplők, melyeket szerepük szerint két csoportra oszthatunk:

- Busz master-nek (röviden master-nek) nevezzük azt a buszra csatlakozó eszközt, mely képes a buszt aktívan használni, tranzakciókat kezdeményezni, adatot átvinni.
- Slave-nek nevezzük azt az eszközt, amely nem képes a busz önálló használatára, kezdeményezni nem tud, adatot átvinni csak akkor képes, ha megszólítják.

A buszt, mint osztott erőforrást, egyszerre csak egy master eszköz használhatja adatátvitelre. A busz használatáért tehát a master-eknek versenyezniük kell egymással. A master először bejelenti az igényét a busz használatára, a tényleges adatátvitel pedig csak akkor kezdődhet el, ha a buszért folytatott versenyt megnyerte. Azt, hogy a buszra egyidejűleg igényt tartó master-ek közül melyik kapja meg a busz használatának jogát, az *arbitráció* folyamata dönti el. Az arbitráció történhet centralizáltan, vagy elosztottan.

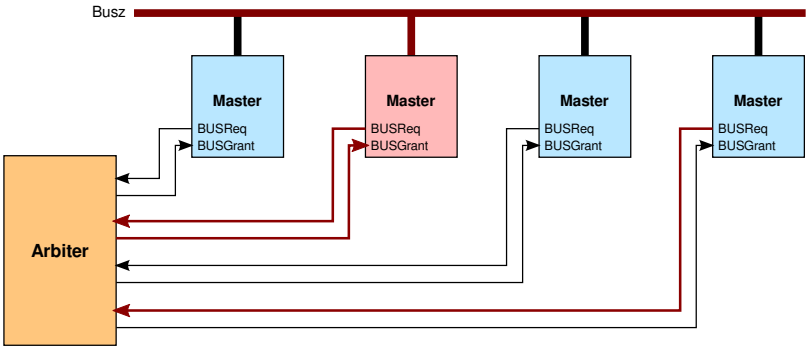
Centralizált arbitráció esetén egy erre a célra bevezetett speciális egység, az arbiter (buszhozzáférés-vezérlő) gondoskodik arról, hogy a buszt egy időben csak egy master használhassa. Centralizált arbitráció megvalósítható például a már megismert *daisy chain*ing segítségével (4.15. ábra). Ebben az esetben a busz iránti igényt mindenki egyetlen, osztott jelvezetéken, a BUSREQ vezetéken jelenti be. Az arbiter, miután ezt a jelzést megkapja, és felszabadul a busz, kiadja a BUSGRANT jelet. Ezt a jelet minden olyan master, amely *nem* szeretné használni a buszt, továbbad a szomszédjának. Ha ez a jel odaér a buszt igénylő master-hez, az számára azt jelenti, hogy megnyerte a busz használatának jogát (a jelet ekkor értelemszerűen nem adja tovább a szomszédjának). A daisy chain alapú megoldás hibája, hogy a master-ek sorrendje a láncban egyben a prioritásukat is meghatározza. Minél hátrébb van a láncban egy master, annál valószínűbb, hogy valaki lecsap a buszra, mire hozzá érne a BUSGRANT jel. A daisy chain viszont nagyon könnyen bővíthető újabb master csatlakoztatásával.



4.15. ábra. Arbitráció daisy chain alapon

Centralizált arbitrációt valósíthatunk meg úgy is, hogy az arbiter minden egyes master busz igénylését egyidejűleg megkapja (4.16. ábra). Az arbiter így pontos képet kap arról, hogy valójában mely master-ek is jelentették be az igényt a buszra, és a versenyhelyzet feloldásakor - a daisy chain-el ellentétben - több szempont alapján hozhatja meg döntését (előre beállítható, vagy dinamikus prioritásokkal, a késleltetéskritikus perifériák előnyben részesítésével, stb.). Ezt a megoldást *párhuzamos* arbitrációnak nevezzük. Így működik az arbitráció a PCI busz esetén is.

*Elosztott arbitráció*t is többféleképpen lehet megvalósítani. Egy lehetséges megoldás, hogy a buszon minden eszköznek saját vezetéke van a busz iránti igény bejelentésére. Minden eszköz látja a többiek igénybejelentéseit is. Amikor több eszköz szeretné egyidejűleg megszerezni a busz használati jogát, látják egymás igénybejelentéseit, és a saját és a többiek prioritásának ismeretében a legnagyobb prioritású eszközön kívül mindegyik visszavonja az igénylését. Ezt a megoldást *önkiválasztó* arbitrációnak hívjuk. Így működik az arbitráció a SCSI buszon is.



4.16. ábra. Arbitráció párhuzamos igénybejelentésre alapozva

Olyan megoldás is lehetséges a busz elosztott használatára, hogy az arbitrációt teljesen kihagyjuk. Ebben az esetben az eszközök a busz előzetes lefoglalása nélkül használják a buszt. Ha egy eszköz adatot kíván átvinni a buszon, rögtön megteszi. Az adatátvitel közben azonban folyamatosan hallgatózik a buszon: ha a saját adatait látja viszont, akkor sikeres volt, de ha nem, akkor valószínűleg egyidejűleg többen is megpróbáltak adatot átvinni, a buszon a több egyidejű forgalmazásból származó érvénytelen "zagyvaság" jelenik meg, tehát az adatátvitel sikertelen volt, azt később újra meg kell ismételni. Ezt a megoldást *ütközésdetektálás*on alapuló busz megosztásnak hívjuk.

Az elosztott arbitráció előnye, hogy kevésbé érzékeny meghibásodásra, hiszen nincs egy, a működés szempontjából kulcsfontosságú szereplő (az arbiter), melynek meghibásodása teljes bénulást okoz.

4.5.3. Példák buszokra

Néhány elterjedt I/O busz paramétereit foglalja össze az alábbi táblázat:

	PCI	**SCSI**	**USB**
Adategység:	32-64 bit	8-32 bit	1 bit
Multiplexált?	Igen	Igen	Igen
Órajel:	133 (266) MHz	5 (10) MHz	Aszinkron
Adatsebesség:	133 (266) MB/s	10 (20) MB/s	0.2, 1.5, 60, 625 MB/s
Arbitráció:	Párhuzamos	Önkiválasztó	Nincs, 1 master van
Masterek max. száma:	1024	7-31	1
Max. fizikai hossz:	0.5m	2.5m	2 – 5m

(Az USB busszal kapcsolatban furcsának tűnhet, hogy csak 1 master lehet a buszon. Az 5.4 fejezetben látni fogjuk, hogy az USB buszra köthető maximum 127 eszközből csakis egyetlen egy, mégpedig a vezérlő kezdeményezhet adatátvitelt.)

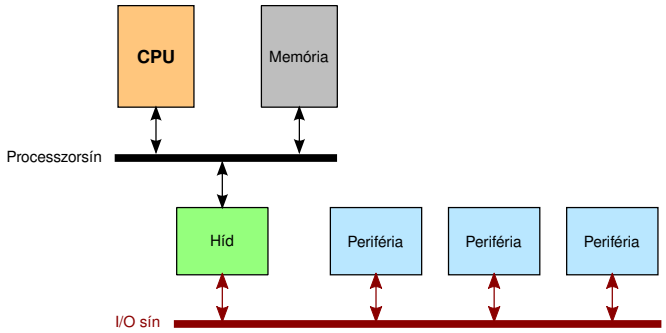
4.5.4. Egy-, több-buszos, ill. híd alapú rendszerek

A PC és az egyéb olcsó számítógépek hőskorában a processzor, a memória és a perifériák egyazon buszt használták az egymással való kommunikációra (lásd 4.2. ill. 4.3. ábra). Ez a közös busz a processzor órajelével megegyező sebességű, szinkron busz. Ezt a kialakítást rendkívül egyszerű megvalósítani, és a bővítés is könnyű: mind a memóriabővítést, mind az új perifériákat ehhez a sínhez kell csatlakoztatni.

Ahogy a processzorok órajele nőtt, egyre jobban megmutatkoztak az egybuszos architektúra hátrányai. A magasabb processzor órajel magasabb busz órajelet kíván (hogy a memóriaműveletek ne fogják vissza a processzor teljesítményét). A magas busz órajel egyrészt drágábbá teszi a perifériák illesztését (hiszen a perifériának ezzel a megnövekedett sebességgel kell feldolgoznia a processzor kéréseit), másrészt olyan fizikai követelményeket támaszt, amit a perifériák illesztése nem tesz lehetővé. Például amellett, hogy a magas órajel nagy adatátviteli

sebességet tesz elérhetővé, de a busz fizikai hosszát rendkívüli módon korlátozza. Egyes perifériák esetén nem fontos az extrém nagy adatátviteli sebesség, de az egyszerű csatlakoztathatóság annál inkább (pl. nem lenne célszerű olyan buszt bevezetni, ami a printer távolságát 50 cm-ben korlátozná). Ráadásul, míg a processzor órajele és ezzel a busz sebessége típusról típusra változhat, addig a perifériák gyártói időtálló, többféle processzor mellett is működő eszközöket szeretnének gyártani, számukra éppen az állandóság, a kiszámítható működési környezet a fontos szempont.

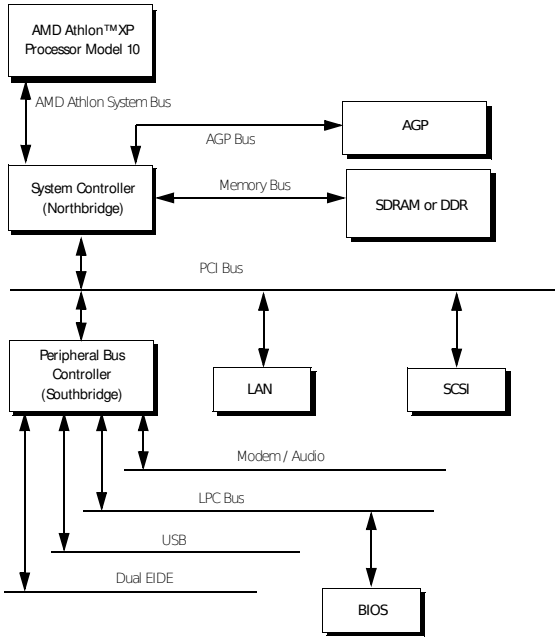
A memória és az perifériák illesztésének eltérő szempontjai vezettek oda, hogy a memória és az I/O sínt kettéválasztották (4.17. ábra, ehhez hasonló koncepciót az I/O processzorok kapcsán is láttunk, a 4.12. ábrán). A két sín közötti átjárást (a kérések tolmácsolását) a híd végzi.



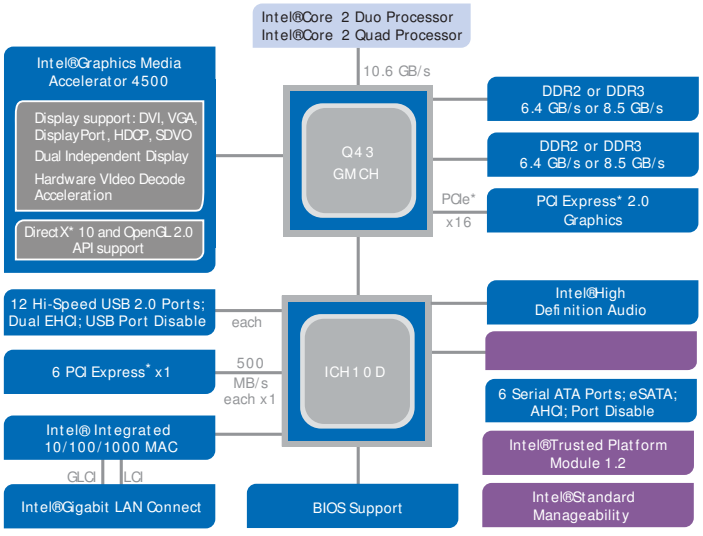
4.17. ábra. Külön processzor és I/O busz

A különválasztott processzor és I/O busz sok problémát orvosolt. Idővel azonban a memóriák is sokszínűvé váltak. A különböző memóriagyártók különféle technológiájú és változatos időzítésekkel rendelkező memóriamodulokat kezdtek gyártani, a memória ebből a szempontból egy kicsit maga is a perifériákhoz kezdett hasonlítani. Manapság ezért a 4.18. ábrán látható, híd alapú architektúrát alkalmazzák, melyben a processzor rendszersín, a memóriasín és a sokféle I/O sín mind elkülönül egymástól. A sínek közötti átjárást két híd biztosítja. Az északi híd (melyet mostanában ”Memory controller hub”-nak, MCH-nak is neveznek) a memória kezeléséért felelős. A hozzá érkező memóriakéréseket úgy szolgálja ki, hogy azt a hozzá kapcsolódó memória aktuális típusának (DDR2, DDR3, stb.) és időzítéseinek megfelelő memóriatranzakcióvá fordítja le. A memória így viszonylag szabadon, a paraméterek széles tartományában illeszthetővé válik, a processzor tudta és támogatása nélkül. Az északi hídhoz általában közvetlenül kapcsolódik a nagy memória-sávszélességet igénylő grafikus megjelenítő periféria, egy speciális nagy sebességű busszal (az ábrán AGP). A 4.18. ábrán a fő I/O busz a PCI busz. Erre kapcsolódik a déli híd (újabb neve ”I/O controller hub”, ICH), amely további I/O buszok felé biztosítja az átjárást. Az AMD Athlon mellett ilyen I/O szervezése volt az korai Intel Pentium rendszereknek is.

A korszerű PC-kben (4.19. ábra) az északi és déli híd közötti kommunikációra a PCI helyett más, nagyobb sebességű összeköttetést vezettek be, a PCI pedig a déli híd kezelésébe került, és csupán egy lett a sok támogatott csatolófelület között. A grafikus megjelenítő periféria AGP helyett PCI Express buszon kapcsolódik az északi hídhoz, valamint az északi híd szinte minden új processzorban a processzorral egyazon tokba (vagy egyazon lapkára) kerül. Ezektől a kisebb változtatásoktól eltekintve az elv ugyanaz, mint amit eddig láttunk: a különböző igényeket támasztó komponenseknek különálló, szabványos buszt építenek ki, valamint a nagyobb adatátviteli sebességet igénylő perifériákat igyekszenek minél közelebb helyezni a processzorhoz, illetve a memóriához.



4.18. ábra. Az AMD Athlon XP rendszerek blokksémája [1]



4.19. ábra. Intel Core 2 rendszerek blokksémája Q43 chipset-tel [4]

5. fejezet

Periféria csatolófelületek

Ebben a fejezetben megismerkedünk a legelterjedtebb perifériaillesztő csatolófelületekkel, a PCI-jal, a PCI Express-szel, valamint az USB-vel. Nem célunk minden részletre kiterjedő referenciát adni, de a leírtak elég részletesek ahhoz, hogy a működés alapelveit megértsük, és azonosítsuk a korábbi fejezetben tanult perifériakezelési alapelveket, pl. hogy hogyan történik a megszakítások kezelése, a forgalomszabályozás, az arbitráció, stb.

5.1. Kihívások

A PC-k elterjedésében többek között az is fontos szerepet játszott, hogy bővítőkártyák segítségével nagyon egyszerű volt a rendszert új perifériával bővíteni (külső periféria esetében is, hiszen a bővítőkártyán volt a külső periféria kábele számára a fizikai csatlakozó). Sajnálatos módon a bővítőkártyákat fogadó csatolófelületet nem éppen az időtállóság figyelembevételével alakították ki. A PC-k újabb és újabb generációinak megjelenése sokszor járt a csatolófelület változásával (ISA, MCA, EISA, VESA, stb.), és az újabb csatolófelületek inkompatibilitása miatt a régi bővítőkártyák sokszor használhatatlanná váltak.

Amellett, hogy maga a csatolófelület is változékony volt, a PC-k perifériákkal való bővítése a PCI és az USB előtti időkben néhány más problémával is küzdött.

- A PC-be illeszthető kártyák száma erősen korlátos.
- A periféria illesztéséhez ki kell kapcsolni a számítógépet.
- A kártyákkal a processzornak kommunikálnia kell, tehát kell legyen saját címtartományuk az I/O címtérből, sok kártya megszakítást is szokott generálni, és esetleg DMA átvitelben is részt vesz. Ezek az erőforrások azonban meglehetősen szűkösek voltak.
  - Szűkös I/O címtér. A PC 16 bites címteret használ a perifériák felé, ami bőven elég is lenne, ha a 90-es évek közepére ki nem alakul az a szerencsétlen gyakorlat, hogy a gyártók ennek csak az alsó 10 bitjét dekódolják. Emiatt gyakorlatilag csak 1024 I/O cím vált használhatóvá, aminek ráadásul egy jó részét a PC saját használatra lefoglalta.
  - A megszakítások korlátozott száma. A PC architektúra eredetileg 15 hardver megszakítást (IRQ) tudott megkülönböztetni, melyek túlnyomó része már foglalt, használatban volt (0: timer, 1: billentyűzet, 3-4: soros portok, 5: hang+párhuzamos port, 6: floppy, stb.). A PCI előtti időkben külön gondot jelentett, hogy az új bővítőkártya számára a megszakításkéréshez szabad vonalat találjunk.
  - A DMA csatornák korlátozott száma. Ugyanaz a helyzet, mint a megszakításokkal. Összesen 8 állt rendelkezésre az egész rendszer számára, melyek közül voltak foglaltak. A PC DMA vezérlője ráadásul kifejezetten lassú volt.
  - Az akkoriban elterjedt periféria buszokat úgy specifikálták, hogy az alaplapok gyártása a lehető legegyszerűbb és legolcsóbb legyen. Ennek érdekében több elterjedt busz gyakorlatilag kis különbséggel megegyezett a ”hozzá való” processzor rendszersínjével. Pl. az ISA busz a 8086, a VESA busz a 80486-os rendszersínjéhez idomult.

A problémák a PC-k és a hozzájuk való perifériák robbanásszerű elterjedésével a 90-es években csúcsosodtak ki, elkerülhetetlenné vált, hogy a perifériabővítés kérdését alaposan átgondolják, és a fenti problémáktól mentes, szabványos (nem csak a PC-re korlátozott) csatolófelületeket dolgozzanak ki.

5.2. A PCI csatolófelület

A PCI (Peripheral Component Interface) egy hardver eszközök számítógéphez való illesztésére szolgáló csatolófelület. Specifikációjának első változatát az Intel adta ki 1992-ben.

Főbb tulajdonságai a következők:

- Processzorfüggetlen
- Az eszközök automatikus felismerésének és konfigurálásának támogatása
- A rendszer összes PCI perifériája minimális esetben csupán egyetlen hardver megszakítást igényel (összesen)
- Memóriára leképzett perifériakezelés az I/O címtér kisebb terhelése érdekében
- Minden írási/olvasási művelet ”burst” módban is történhet: tehát a PCI nem csak a bájtok egyenkénti átvitelét támogatja, hanem egy hosszabb bájtsorozat is átvihető egyetlen PCI tranzakcióval.
- 32 vagy 64 bites adategységek átvitele
- 32 eszköz egy PCI buszon
- 256 PCI busz egy rendszerben
- Alacsony fogyasztás támogatása
- Szinkron busz 33 MHz-es órajellel (a szabvány 1995-ben megjelent 2.1-es változata már 66 MHz-es órajelet is megenged)
- Paritásellenőrzés a kiadott címekre, parancsokra és az átvitt adatokra
- Rejtett arbitráció: a buszért való versengés és a nyertes kiválasztása egy éppen zajló adatátviteli tranzakció *közben* is történhet.

A PCI eleinte lassan terjedt, de 1995-től kezdve egyre több gyártó támogatta, nem csak a PC architektúrájú számítógépek körében. Többek között 1995-től az Apple Power Macintosh, és a DEC AlphaStation, valamivel később a PA-RISC és a SUN SPARC alapú munkaállomások is rendelkeznek PCI csatolófelülettel.

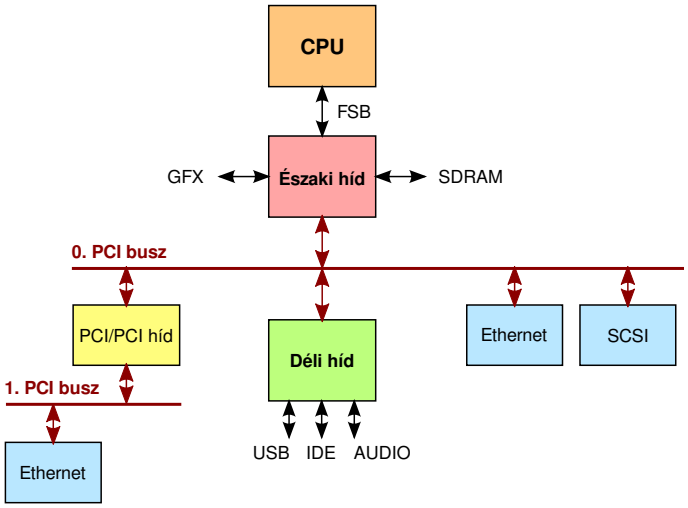
5.2.1. PCI alapú rendszerek

A PCI sín, a memória, a processzor és a további (nem PCI csatolóval rendelkező) perifériák összekapcsolására több megoldás is született. Az 5.1. ábrán az Intel Pentium Pro alapú rendszereiben alkalmazott elrendezés látható. Ebben az elrendezésben az északi híd látja el a memória tranzakciók kiszolgálását, jöjjön az a processzor, a grafikus kártya vagy a PCI sín felől. Mivel a processzor számára a memória minél gyorsabb elérése rendkívül fontos, az északi hidat és a CPU-t összekötő FSB (Front-Side Bus, rendszerbusz) nagy sebességű kapcsolatot tesz lehetővé. A grafikus kártya esetében is fontos a gyors memóriaelérés, ezért a lehető legközelebb kell kerülnie a memóriához, így az is közvetlenül az északi hídhoz kapcsolódik egy nagy sebességű kapcsolaton keresztül. A többi periféria egy PCI buszon keresztül érhető el. A PCI buszra csatlakozik a déli híd is, amely további illesztési felületeket ill. buszokat támogat a nem-PCI perifériák számára (IDE diszkek, USB busz, hang ki/bemenet, stb.).

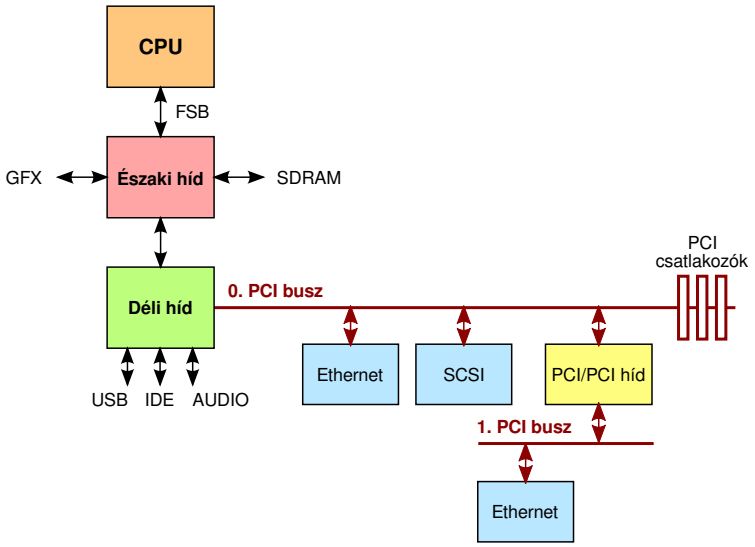
Később a PCI sávszélessége kevésnek bizonyult az északi híd és a déli híd összekötésére, ezért az Intel a Pentium 4-től kezdve az 5.2. ábrán látható módon valósítja meg a számítógép elemeinek összekötését. Ebben a megoldásban a PCI a déli híd kezelésébe kerül, az északi és a déli hidat pedig egy nem szabványos, a PCI-nál nagyobb sebességű kapcsolat köti össze (DMI vagy QPI). Ennek az elrendezésnek az az előnye is megvan, hogy nem a PCI buszt terheli a többi busz, illetve nem-PCI periféria forgalma. Mivel az északi híd és a déli híd kompetenciája teljesen elkülönül, az ilyen rendszerekben az északi hidat memóriavezérlőnek (Memory Controller Hub, MCH), a déli hídat pedig I/O vezérlőnek (I/O Controller Hub, ICH) nevezik.

Természetesen nem csak az 5.1. és az 5.2. ábra szerint lehet egy PCI buszt tartalmazó számítógépet megvalósítani, ezek csak az Intel PC-ben alkalmazott megoldásai. A két bemutatott ábrán azonosíthatjuk a PCI busz megvalósításához szükséges elemeket. Ezek a következők:

- **Host/PCI híd** (Host/PCI bridge). A Host/PCI híd végzi a ”fordítást” a processzor I/O ill. memória kérések és a PCI tranzakciók között. Az 5.1. ábra megoldásában ez a funkcionalitás az északi híd feladata, az 5.2. ábra szerinti elrendezésben pedig a déli hídé (amit ott ICH-nak hívnak).



5.1. ábra. Kezdeti PCI alapú rendszer



5.2. ábra. Újabb, PCI buszt támogató rendszer

- **PCI eszközök** (PCI device). Egy PCI sínen a specifikáció szerint 32 eszköz illesztése lehetséges. A gyakorlati tapasztalatok azonban azt mutatták, hogy különféle (elektromos) okokból 10-nél több eszköz illesztése egy sínre nem célszerű. Egy PCI eszközön (egy kártyán) azonban több, egész pontosan legfeljebb 8 logikai periféria, ún. *PCI function* helyezhető el. Ezek a logikai perifériák a PCI rendszer számára különálló egységek, melyek fizikailag közös interfészen csatlakoznak a PCI sínre.
- **PCI-PCI híd** (PCI-PCI bridge). A PCI sínen speciális eszközök, hidak is lehetnek, melyek a PCI sínre egy újabb PCI sínt kötnek. A hidak segítségével a PCI sínek hierarchikus struktúrába szervezhetők, így az illeszthető perifériák száma nagy mértékben növelhető. A PCI specifikáció 256 sín használatát teszi lehetővé, így az elméletileg illeszthető perifériák száma nagyjából: 8 function/eszköz * 32 eszköz/sín * 256 sín/rendszer = 65536.

5.2.2. Egyszerű adatátviteli tranzakció

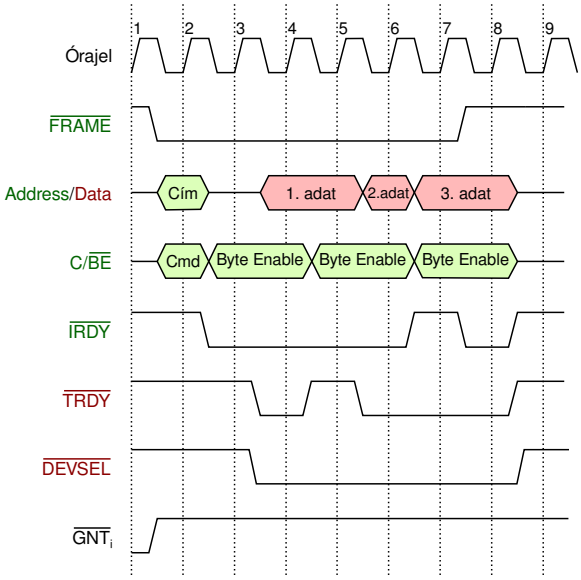
Minden PCI eszköz rendelkezik legfeljebb 6 db ”ablakkal”, amin keresztül adatokat lehet cserélni vele. Az eszköz a megfelelő konfigurációs tranzakció során (lásd később) meg tudja mondani, hogy hány és mekkora méretű

ablakra van szüksége. Egy ablak nem más, mint a memória vagy az I/O címtartomány egy folytonos darabja. Pl. ha egy eszköz 64 kB méretű memória ablakot kér, a BIOS, illetőleg az operációs rendszer az indulás során kioszt neki egy 64 kB-os darabot a teljes címtartományból. Ezután minden egyes memóriaművelet mögött, amit a processzor az ablak területére intéz, valójában PCI tranzakciókon keresztül az eszköz fog állni: az innen való olvasás a perifériáról való olvasást, az ide való írás a perifériára való írást fog jelenteni.

Az az északi híd feladata, hogy rájöjjön, hogy a processzor által kiadott cím mögött a fizikai memória, vagy egy periféria áll-e. Az északi híd nem tárolja az összes PCI periféria összes ablakát méretekkel együtt, és nem kezdi el minden egyes memóriaművelet során egyenként ellenőrizni, hogy vajon nem áll-e a cím mögött periféria, erre egyszerűen nincs idő. A PC-ben egy egészen egyszerű megoldást választottak: a 32 bites címtartomány felső 1 GB-os darabját lefoglalták a memóriára leképzett perifériaműveletek számára. Itt helyezhetők el a PCI perifériák ablakai is. Az északi híd tehát csak ránéz a címre, ha <3 GB, akkor a fizikai memóriához, ha $\geq$ 3 GB, akkor a perifériákhoz fordul. (Ez az oka, hogy alapesetben egy PC-s 32 bites operációs rendszer csak 3 GB memóriát tud kezelni akkor is, ha fizikailag több van a gépben).

A rendszer bekapcsolásakor tehát minden eszköz megkapja a kért számú és méretű ablakát a memóriából, vagy az I/O címtartományból, igény szerint. Ha a processzor I/O címre hivatkozik, vagy egy olyan memóriacímre, amely mögött PCI periféria lehet (PC-n $\geq$ 3 GB), a híd összeállít egy adatátviteli tranzakciót. A Hoszt/PCI híd nem tudja, melyik perifériának az ablakát hivatkozták meg, de ez nem baj, mert a perifériák a saját ablakaik elhelyezkedését pontosan tudják. A híd tehát elkéri a busz használatának jogát, és kiteszi a meghivatkozott címet a buszra. A buszon ülő perifériák ezt mind megnézik, és amelyik észreveszi, hogy az ablakába esik a cím, szól, hogy övé a tranzakció.

Egy példa tranzakció további lebonyolítása az 5.3. ábra szerint történik ([36]).



5.3. ábra. Példa PCI tranzakcióra

A példában a kezdeményező a Host/PCI híd, ami egy olvasási műveletet észlelt a PCI címtartományba eső memóriaterületre. A Host/PCI híd ezután versenybe kezd a buszért (hiszen lehet, hogy épp egy másik tranzakció zajlik rajta, és lehet, hogy épp többen is szeretnék egyszerre használni - az arbitráció leírását lásd később), melyet előbb-utóbb megkap, amit a számára dedikált $\overline{\text{GNT}}$ jel alacsonyba váltásából tud meg (1. órajel felfutó él). Ezután elindítja a tranzakciót: kiteszi a processzor által meghivatkozott memóriacímet a buszra az A/D (Address/Data) vezetékekre, a művelet típusát (Memory read) pedig a C/$\overline{\text{BE}}$ vezetékekre (2. órajel). Ezt követően jelzi, hogy készen áll az adatok fogadására ($\overline{\text{IRDY}}$-t (Initiator Ready) 0-ba húzza), és a C/$\overline{\text{BE}}$ vezetékekre írt "Byte Enable" 4 bites információval jelzi a címzettnek, hogy a megcímzett 32 bites adategység 4 bájtjából melyeket kéri, és milyen sorrendben (=0: legegyszerűbb eset: az egészet, úgy, ahogy van). Eközben a buszon lévő perifériák ellenőrzik a címet. Az egyik periféria hosszabb-rövidebb idő után észleli, hogy a cím az ő egyik ablakába esik, amit a $\overline{\text{DEVSEL}}$ vezeték alacsonyba húzásával jelez. Egyben közli a címzettel, hogy kész az adatok küldésére ($\overline{\text{TRDY}}$-t (Target Ready) 0-ba húzza), és ráteszi az első 32 bites adategységet az A/D vezetékekre (4. órajel). Ezt a

kezdeményező (most a Host/PCI híd) elolvassa, és várja a további adatokat (esetleg megváltoztatja a Byte Enable-t, ha akarja). Ha a megcímzett periféria nem bírja az iramot, a $\overline{\text{TRDY}}$=1-el jelezheti, hogy még nem áll készen (5. órajel). Amikor ismét készen áll, kiteszi a 2. adategységet is, melyet a kezdeményező leolvas (6. órajel). Ezután jönne a 3. adategység átvitele, de az is előfordulhat, hogy a kezdeményező az, aki nem bírja az iramot, amit az $\overline{\text{IRDY}}$=1-el jelez (7. órajel). A 8. órajelre ismét mindegyik kész az adatátvitelre, így a 3. adategység is a buszra kerül. Közben a kezdeményező jelzi, hogy vége a tranzakciónak (a 8. órajel felfutó élére $\overline{\text{FRAME}}$=1), amire a megcímzett periféria elengedi a buszt ($\overline{\text{DEVSEL}}$=1). Így olvasott ki a Host/PCI híd 3 darab 32 bites adategységet a megcímzett perifériából, melyet a processzor számára, mint a memóriaművelet eredményét, továbbít.

A példából többek között megismertük a PCI forgalomszabályozó mechanizmusát is:

- az $\overline{\text{IRDY}}$ jelzéssel jelzi a kezdeményező, hogy kész a következő adat átvitelére,
- a $\overline{\text{TRDY}}$ jelzéssel jelzi a megcímzett periféria, hogy kész a következő adat átvitelére.

Ebben a példában az adatátvitelt a processzor kezdeményezte azzal, hogy a periféria fennhatósága alá tartozó I/O vagy memóriaműveletet végzett (ezt programozott I/O-nak is nevezik). A PCI szabvány az eszközök számára ennél nagyobb önállóságot is megenged: egy PCI periféria is kezdeményezhet adatátvitelt, akár a rendszermemóriába (lényegében DMA), akár közvetlenül egy másik periféria felé, a processzor teljes megkerülésével (tehermentesítésével). A PCI tehát az alábbi tranzakciós modelleket támogatja:

- Programozott I/O. Kezdeményező: CPU, irányultság: periféria.
- DMA. Kezdeményező: periféria, irányultság: memória.
- Peer-to-peer adatátvitel. Kezdeményező: periféria, irányultság: periféria.

5.2.3. Parancsok

A legfontosabb PCI parancsokat a következő táblázat foglalja össze (nem teljes a felsorolás!):

C/$\overline{\text{BE}}$	Parancs
0000	Megszakítás nyugtázás (Interrupt Acknowledge)
0010	I/O olvasás (I/O Read)
0011	I/O írás (I/O Write)
0110	Memória olvasás (Memory Read)
0111	Memória írás (Memory Write)
1010	Konfiguráció olvasás (Configuration Read)
1011	Konfiguráció írás (Configuration Write)

A parancsok jelentése a következő:

- Megszakítás nyugtázás: A periféria által kért megszakítás nyugtázása a PCI buszon egy megszakítás nyugtázás tranzakció formájában érkezik meg a perifériához.
- I/O olvasás/írás: Egy olvasási/írási művelet a címbuszon adott I/O címre. Annak a perifériának szól, amelyiknek ez a cím az egyik I/O ablakába esik.
- Memória olvasás/írás: Egy olvasási/írási művelet a címbuszon adott memóriacímre. Annak a perifériának szól, amelyiknek ez a cím az egyik memória ablakába esik.
- Konfiguráció olvasás: Ennek segítségével lehet egy eszközről információkat lekérdezni. Ilyen információ az eszköz kódja, az eszköz gyártójának kódja, az eszköz besorolása (adattároló, hálózati eszköz, grafikus megjelenítő, multimédia eszköz, stb.), az eszköz időzítésbeli igényei, valamint az, hogy hány és mekkora ablakra van szüksége az I/O és a memória címtartományában. (A megcélzott eszköz kijelöléséről később lesz szó.)
- Konfiguráció írás: A kijelölt eszköz konfigurációs regisztereinek írása, pl. a kért ablakok kezdőcímének beállítása.

5.2.4. Arbitráció

A busz, mint osztott erőforrás használatáért való versenyre a PCI párhuzamos arbitrációt használ. A busz osztott használatát egy központi elem, az arbiter felügyeli (mely az 1. ábra szerinti elrendezésben az északi, a 2. ábra szerintiben a déli híd része lehet). Minden egyes periféria egy kizárólag számára dedikált vezetékpárral kapcsolódik az arbiterhez. Az egyik vezetéken ($\overline{\text{REQ}}$) jelentheti be a periféria, hogy szüksége lenne a buszra, a másik vezetéken ($\overline{\text{GNT}}$) jelez vissza az arbiter, ha a busz használati jogát megkapta.

A PCI ú.n. rejtett arbitrációt támogat, vagyis a busz iránti igények bejelentése és a nyertes kiválasztása az aktuálisan zajló tranzakció *alatt* történik. Amikor az arbiter kiválasztotta a nyertest, az aktuális tranzakció kezdeményezőjétől elveszi, a nyertesnek pedig megadja a $\overline{\text{GNT}}$ jelet. A nyertes azonnal elkezdheti használni a buszt, amint az aktuális tranzakció véget ért. A rejtett arbitrációnak köszönhetően tehát nem megy kárba értékes órajelciklus arbitrációs céllal, a tranzakciók szorosan követhetik egymást.

A PCI szabvány nem határozza meg, hogy az arbiter milyen algoritmussal válassza ki a nyertest, ha több eszköz is versenybe száll a buszért, csak annyit ír, hogy a döntésnek ”fair”-nek kell lennie. A döntés során figyelembe veheti az eszközök jellegzetességeit is, pl. a konfigurációs regiszterekből kiolvasott időzítési adatok alapján az eszközöket ”késleltetésre érzékeny” és ”késleltetésre nem érzékeny” csoportokba sorolhatja.

Egy lehetséges megoldás pl. a következő kétszintű körbenforgó eljárás lehet. Legyen A és B késleltetésre érzékeny, X, Y, Z pedig késleltetésre nem érzékeny eszköz. Az arbiter körbenforgó elven sorra a késleltetésre érzékeny eszközöknek adja a buszt, de minden egyes körben egy késleltetésre nem érzékeny eszköznek is odaadja a busz használati jogát. Tehát, ha minden eszköz folyamatosan jelenti be az igényét, az arbiter az alábbiak szerinti sorrendben biztosít hozzáférést a buszhoz: A, B, X, A, B, Y, A, B, Z, A, B, X, A, B, Y, A, B, Z, stb.

5.2.5. Megszakításkezelés

A PCI perifériák kétféle módon kérhetnek megszakítást a processzortól:

- A 4 megszakítás jelzésére szolgáló vezeték egyikének alacsonyra húzásával
- Üzenettel jelzett megszakítás (Message Signaled Interrupt, MSI) segítségével

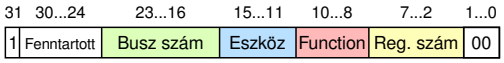
A PCI csatoló 4 interrupt vezetékkel rendelkezik. Ezeket a perifériák nem tetszőlegesen használhatják, a szabvány megköti, hogy pl. az egy function-t tartalmazó eszközök csak az A vonalat használhatják, a B, C, D-t nem. Azt azonban nem köti meg a szabvány, hogy ez a minden eszköz rendelkezésére álló 4 interrupt vezeték milyen viszonyban van a számítógép megszakításkezelésével. Olyan PCI implementáció is megengedett, melyben minden eszköz mind a 4 vezetéke egy helyre fut, így bármelyik eszköz is jelezne bármelyik vonalán, a processzor azt ugyanannak a megszakításnak érzékelné (ilyen esetben a megszakításkezelő szubrutin feladata az összes eszköz egyenkénti végigkérdezése, hogy ki is váltotta ki a megszakítást). A másik szélsőség, hogy minden eszköz mind a 4 megszakítás vezetéke mind különböző, egymástól elkülöníthető megszakításként jelentkezik a processzor felé. Ez természetesen hatékony, mert így mindig azonnal az a megszakításkezelő szubrutin hívódik meg, amelyik a megszakítást kérő eszközhöz tartozik, de sajnos ehhez a megoldáshoz nagy számú elkülöníthető hardver interruptot kellene támogatnia a számítógépnek, ami nem jellemző. A PC 15 hardver megszakítás elkülönítését teszi lehetővé, aminek egy része különféle célokra fenntartott. A szabad hardver megszakításokat a BIOS (vagy megfelelő támogatás esetén az operációs rendszer) rendeli hozzá a PCI buszhoz. Így az esetlegesen nagy számú PCI eszköz interruptjai néhány hardver megszakításra képeződnek le, vagyis a megszakításkezelőnek a PC esetén is feladata, hogy a kiváltott megszakításhoz rendelt eszközöket végigkérdezze, hogy melyik is kért megszakítást.

A megszakítás kérésének másik módja az üzenettel jelzett megszakítás. A periféria egy konfigurációs regiszterbeli bejegyzésében jelezheti, hogy támogatja az MSI-t. Ha a Host/PCI híd is támogatja, akkor lehetővé válik ennek a módnak a használata is. Ebben az esetben a perifériának kiosztanak (legalább) két számot: kap egy speciális memóriacímet, és annyi speciális azonosító számot, ahányféle megszakítást ki szeretne váltani. A periféria úgy tudja kiváltani a megszakítást, hogy erre a számára kiosztott speciális memóriacímre végrehajt egy egyszerű memória írás tranzakciót, melynek során a megszakításnak megfelelő azonosító számot írja erre a címre. Ezt a címet a Host/PCI híd ismeri fel sajátjának, így ő lesz a partner azokban a memóriatranzakciókban, melyek ezt a kiosztott speciális címet célozzák meg. Ha erre a címre akar írni valamelyik periféria, elveszi tőle az ide írt azonosító számot, és a beállításainak megfelelően kivált egy megszakítást a processzor felé. Azt, hogy a perifériák különböző megszakításait hogyan képezi le a processzor által elkülöníthető hardver megszakításokra, azt a szabvány nem rögzíti, ez megvalósításfüggő.

5.2.6. Konfigurálás

Minden PCI periféria rendelkezik 64 darab 32 bites konfigurációs regiszterrel, melyben a hozzá kötődő konfigurációs beállításokat tárolja. A konfigurációs regiszterek tartalmának kiolvasására, illetve megváltoztatására minden PCI rendszer rendelkezik 2 speciális memória vagy I/O címmel. A konfigurálást végző szoftver (BIOS ill. operációs rendszer) az egyik speciális címre beírja az elérni kívánt PCI periféria azonosítóját, és az elérni kívánt konfigurációs regiszter sorszámát, majd ezután a másik speciális címről ki lehet olvasni a kijelölt konfigurációs regiszter tartalmát, illetve az ide írt tartalom a konfigurációs regiszterbe íródik.

A PC esetén a periféria ill. a konfigurációs regiszter kijelölése a CF8h I/O címre írt 32 bites adattal lehetséges. Az 5.4. ábrán látható a 32 bit felosztása: egy része kijelöli, hogy hányas busz, egy másik része, hogy ezen a buszon hányadik eszköz, egy további része, hogy ezen az eszközön hányadik function, végül, hogy ennek hányadik konfigurációs regiszterére vagyunk kíváncsiak. Tehát ezzel a speciális címzési móddal lehet kijelölni az elérni kívánt konfigurációs regisztert, melynek tartalmát a CFCh címre írt 32 bites adattal lehet megváltoztatni, illetve erről a címről való 32 bites olvasással kiolvasni.



5.4. ábra. PCI perifériák konfigurációs regisztereinek címzése

Lássuk, mi is történik ilyenkor a háttérben (most tegyük fel, hogy csak 1 PCI busz van a rendszerben). A Host/PCI híd a 2 speciális címre történő I/O műveletekre folyamatosan figyel. Ha úgy látja, hogy valaki az előbb leírt módon kijelölt egy konfigurációs regisztert, majd azt írni vagy olvasni akarja, akkor egy konfigurációs írás ill. olvasás tranzakciót indít a megcímzett eszköz felé. Ezt az alábbi módon teszi:

- Az 5.4. ábrán látható formátumú címből látja, hogy melyik eszközről van szó. A kijelölt eszköznek beállítja az IDSEL jelzését (ez minden eszközre egyedi, tehát csak ennek az egy eszköznek lesz beállítva az IDSEL jelzés).
- A C/$\overline{\text{BE}}$ vonalakon a ”Konfiguráció olvasás” vagy ”Konfiguráció írás” tranzakció parancs kódját írja. Innentől kezdve a továbbiakban csak az IDSEL-el kijelölt eszköz figyeli a buszon zajló eseményeket, a többi megállapítja, hogy rá a tranzakció nem vonatkozik.
- Az A/D vonalakra a következő információkat teszi a Host/PCI híd:
  - a konfiguráció típusát (nem részletezzük, általában 0),
  - a function számát (tehát hogy a konfigurálás az adott eszközön hányadik logikai perifériára vonatkozik),
  - a kért konfigurációs regiszter számát.
- Ezután a konfigurációs regiszter tartalma az ”egyszerű adatátviteli tranzakció”-nál látott módon, a Byte Enable jelekkel és forgalomszabályozással fűszerezve megy át a periféria és a Host/PCI bridge között (az átvitel iránya attól függ, hogy konfiguráció írásról, vagy olvasásról van-e szó).

A minden egyes PCI periféria részére rendelkezésre álló 64 regiszterből az első 16 tartalmát a szabvány megköti, a többit a periféria gyártója szabadon használhatja. Az első 16 regiszterből számos fontos információhoz lehet jutni, pl:

- VendorID: a periféria gyártójának azonosítója
- DeviceID: a periféria azonosítója
- Revision: hányadik verzió
- Class Code: a periféria típusa: adattároló, hálózati eszköz, grafikus megjelenítő, multimédia eszköz, stb.
- Interrupt Pin: a 4 lehetséges interrupt láb melyikét használja a periféria
- Interrupt Line: A periféria megszakításkérését a processzor hányas számú megszakításként érzékeli
- Base Address Register 0...5: a periféria által használt, a memória vagy I/O ablakok kezdőcíme. Ha tartalmát csupa 1-es bitekkel töltjül fel, majd ezt követően ugyanezt a regisztert kiolvassuk, akkor a periféria megmondja, hogy mekkora ablakra van szüksége, és azt is, hogy a memóriában vagy az I/O címtartományban kéri-e.

A PCI perifériák konfigurálását a BIOS és/vagy az operációs rendszer végzi el a rendszer indulásakor. A konfiguráló szoftver tipikusan lekérdezi az összes lehetséges PCI perifériát (a 4. ábrán látható címzéssel), kiolvassa a VendorID és DeviceID mezőket, ami alapján be tudja azonosítani az eszközt, kiosztja az ablakokat, és az eszköz meghajtóprogramjára (device driver) bízza a további eszközspecifikus konfiguráció végrehajtását.

5.2.7. A PCI csatoló jelzései

Az alábbiakban felsoroljuk, hogy a PCI kártyákon, kivezetések formájában, milyen jelzések találhatók. A többségüket az eddigi leírásból már ismerjük. Maga a csatolófelület az 5.5. ábrán látható.

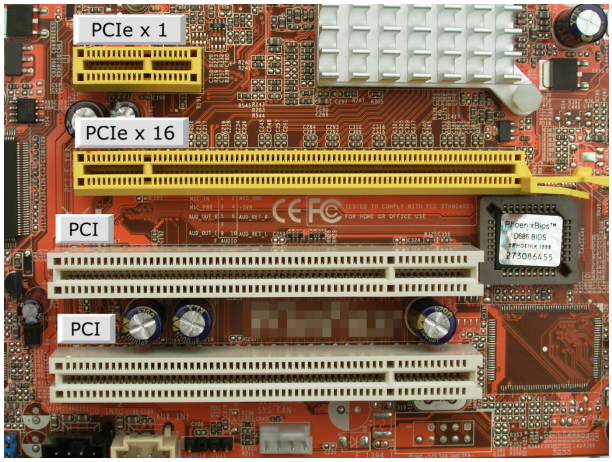
Kötelezően használandó jelek:

- Órajel (osztott)
- Reset (osztott)
- A/D[0...31]: címzésre és adatátvitelre használatos vonalak (osztott)
- C/$\overline{\text{BE}}$[0...3]: parancsok és byte enable továbbítására használatos vonalak (osztott)
- Paritás (az A/D-ra és C/$\overline{\text{BE}}$-ra számolt paritásbit, osztott)
- $\overline{\text{FRAME}}$: jelzi a tranzakció időtartamát. A kezdeményező alacsonyba húzza a tranzakció kezdetekor, és magasba, ha kész az utolsó adategység vételére (osztott)
- $\overline{\text{TRDY}}$: a megcímzett periféria ezzel jelzi, hogy kész az adatátvitelre (osztott)
- $\overline{\text{IRDY}}$: a kezdeményező ezzel jelzi, hogy kész az adatátvitelre (osztott)
- $\overline{\text{DEVSEL}}$: a megcímzett periféria ezzel jelzi, hogy dekódolta a címet, és az hozzá tartozik (osztott)
- $\overline{\text{IDSEL}}$: a konfigurálás során ez jelzi a perifériának, hogy a konfiguráló tranzakció rá vonatkozik (minden perifériának külön)
- $\overline{\text{STOP}}$: a megcímzett periféria ezzel jelezheti, hogy szeretné a tranzakciót idő előtt leállítani (osztott)
- $\overline{\text{PERR}}$: a periféria ezen keresztül jelentheti be, ha paritáshibát észlelt (osztott)
- $\overline{\text{SERR}}$: egyéb kritikus hibák jelzése (osztott)
- $\overline{\text{REQ}}$: a periféria ezzel jelentheti be igényét a busz használatára, ha tranzakciót szeretne kezdeményezni (minden perifériának külön)
- $\overline{\text{GNT}}$: az arbiter ezzel jelzi a periféria számára, hogy használhatja a buszt (minden perifériának külön)

Opcionális jelek (melyeket nem minden perifériának kötelező támogatnia):

- A/D[32...63]: címzésre és adatátvitelre használatos vonalak (osztott)
- C/$\overline{\text{BE}}$[4...7]: parancsok és byte enable továbbítására használatos vonalak (osztott)
- Paritás64 (az A/D[32...63]-ra és C/$\overline{\text{BE}}$[4...7]-re számolt paritásbit, osztott)
- $\overline{\text{INTA}}$: ha egy periféria megszakítást kér, azt ezen a 4 vonalon tudja bejelenteni (osztott, vagy mindenkinek külön)
- $\overline{\text{INTB}}$: lásd előbb
- $\overline{\text{INTC}}$: lásd előbb
- $\overline{\text{INTD}}$: lásd előbb
- $\overline{\text{CLKRUN}}$: Ha a PCI busz épp használaton kívül van, energiatakarékosság miatt mobil rendszerekben az órajelét lekapcsolják, vagy lelassítják. A perifériák erről az eseményről ezen a jelen keresztül értesülnek, illetve ezzel a jellel jelezhetik, hogy szeretnék ismét teljes sebességre kapcsolni a buszt (pl. mert tranzakciót szeretnének kezdeményezni) (osztott).
- $\overline{\text{REQ64}}$: ezzel jelezheti a kezdeményező, hogy 64 bites adatátvitelt szeretne (időzítése a $\overline{\text{FRAME}}$-mel azonos) (osztott).
- $\overline{\text{ACK64}}$: ezzel jelezheti a megcímzett periféria, hogy a 64 bites adatátvitelt képes kiszolgálni (időzítése a $\overline{\text{DEVSEL}}$-el azonos) (osztott).

(Az energia menedzsmenthez és a JTAG támogatáshoz szükséges jelzésektől eltekintünk).



5.5. ábra. A 32 bites PCI csatolófelület[1]

5.3. A PCI Express csatolófelület

A PCI Express (PCIe) felület a PCI-nál lényegesen nagyobb adatátviteli sebességet tesz lehetővé. Ennek elérése érdekében azonban a PCI-tól radikálisan eltérő megoldásokra van szükség, így a PCI és a PCI Express alapvető dolgokban különbözik egymástól.

Amiben mégis közös a PCI és a PCI Express, az a tranzakciókon alapuló adatátviteli modell, a perifériák konfigurálásának mechanizmusa, és a megszakítások kezelése. Ennek köszönhetően a programozó és az operációs rendszer szempontjából a két megoldás egymással kompatibilis: a PCI Expressben a PCI-nál megszokott módon kell az eszközöket detektálni, konfigurálni, adatátvitelt kezdeményezni, stb.

A leglátványosabb változások a PCI-hoz képest:

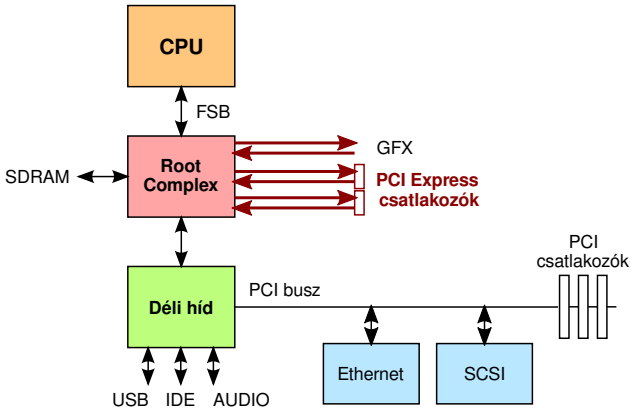
- **Soros átvitel** Első látásra azt gondolnánk, hogy a PCI-ban is látott párhuzamos átviteli mód (ahol is 32 bites adategységeket egyetlen órajel alatt át lehetett vinni) gyorsabb, mint a soros. Valójában a szinkron, párhuzamos átviteli módnak van egy komoly korlátja: nem lehet magas órajelet elérni. A párhuzamosan futó vezetékek hossza és elektromos tulajdonságai ugyanis soha nem hajszál pontosan egyformák, így a küldő oldalon egy időben ”rátett” adatok egymáshoz képest elcsúszva érnek a fogadó oldalhoz. Ez a csúszás nem okoz gondot, amíg a mértéke össze nem mérhető az órajellel. Soros adatátvitel esetén nincs ilyen elcsúszási probléma, az adatátviteli sebesség akár több nagyságrenddel is magasabb lehet, még azzal együtt is, hogy az adatokat bitenként kell átvinni.
- **Pont-pont összeköttetéseken alapuló topológia** Míg a PCI-ban a busz osztott volt (legalábbis a legtöbb vezeték), addig a PCI Express dedikált, pont-pont összeköttetéseken alapul. Ennek köszönhetően az átviteli közeg nem osztott, nem kell érte versengeni, várni rá, arbitráció sincs.

5.3.1. PCI Express alapú rendszerek

Egy egyszerű PCIe alapú rendszert mutat be az 5.6. ábra. Ebben a megvalósításban minden egyes PCIe eszköz a ”Root Complexhez” kapcsolódik, melynek funkciója Host/PCI hídnak felel meg ([8]).

A PCI Express csatolófelülete sokkal egyszerűbb, mint a PCI esetében volt. Az egyéb járulékos vezetékek mellett a tényleges kommunikáció egyetlen érpáron zajlik, soros adatátvitellel, órajel nélkül (az órajelet a két fél a vonal jeleiből egy PLL segítségével elő tudja állítani). Már egyetlen ilyen érpár is a régi PCI-nál jóval nagyobb adatátviteli sebességre képes. Két PCIe eszköz között (alapesetben) két ilyen érpár található, az egyik az oda-, a másik a vissza irányú adatforgalom számára van fenntartva, azaz a PCIe összeköttetései full duplexek, mindkét irányba lehet egyszerre adatot átvinni. Egy ilyen kétirányú soros összeköttetést a PCIe-ben *lane*-nek (pályának) neveznek. Két eszköz között több lane is kihúzható, így az adatátvitel az egymással párhuzamos (*de nem szinkron!*) pályák kihasználásával sokkal gyorsabb lehet. A PCIe szabvány x1, x4, x8, x16, x32 pályát támogat, bár a 32 pályás kialakítás nagyon-nagyon ritka (PC-ben nem létezik). A PCIe sebességeket az 5.1 táblázatban foglaltuk össze, maga a csatolófelület az 5.5. ábrán látható.

[1]Magnus Manske fotója, ”Creative Commons” licensz

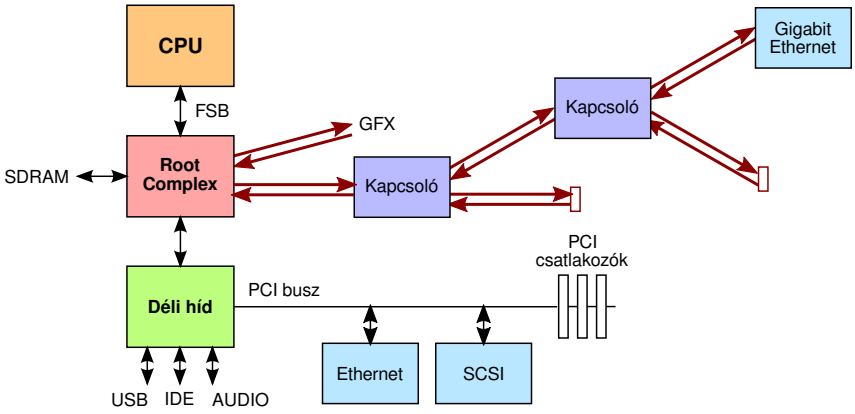


5.6. ábra. Egy egyszerű PCIe rendszer felépítése

Szabvány	PCIe x1	PCIe x16	PCIe x32
PCIe 1.0	250 MB/s	4 GB/s	8 GB/s
PCIe 2.0	500 MB/s	8 GB/s	16 GB/s
PCIe 3.0	≈ 1000 MB/s	≈16 GB/s	≈32 GB/s
PCIe 4.0	≈ 2000 MB/s	≈32 GB/s	≈64 GB/s

5.1. táblázat. PCI Express maximális adatátviteli sebességek

Igaz ugyan, hogy a PCIe pont-pont összeköttetésekre épül, de ez nem jelenti azt, hogy minden eszköznek közvetlenül a Root Complexre kell kapcsolódnia. Egy összetettebb PCIe alapú rendszert láthatunk az 5.7. ábrán, ahol egyes eszközök ú.n. kapcsolókon (switcheken) keresztül vannak összeköttetésben a Root Complexszel. Ezek a kapcsolók a tranzakciókat a megfelelő irányba továbbítva azt a látszatot keltik az eszközök számára, mintha közvetlenül a Root Complexszel kommunikálnának.



5.7. ábra. PCIe rendszer kapcsolókkal

5.3.2. PCI Express tranzakciók átvitele

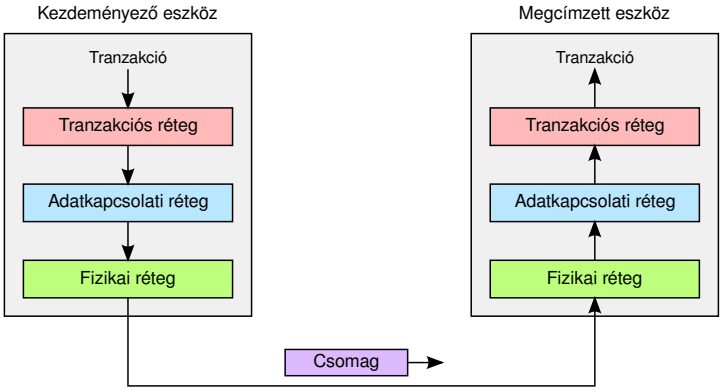
A PCI Express működése egy csomagkapcsolt hálózatra hasonlít. A tranzakciók, a hozzájuk kapcsolódó járulékos információkkal együtt egy egységként, csomagként haladnak a küldőtől a cél irányába, esetleg több PCIe kapcsoló érintésével, ”kézről kézre adva”. A tranzakciót tartalmazó csomagot a küldő bitről bitre elküldi a cél számára, az pedig összeállítja azt, majd ha az utolsó bit is megérkezett, belenéz, és végrehajtja az ott leírt tranzakcióhoz

kapcsolódó feladatokat (vagy, ha egy kapcsolóról van szó, továbbítja egy másik csomópont felé).

Amikor a periféria, vagy a Root Complex egy tranzakciót kezdeményez, az három rétegen megy keresztül, mire fizikailag a soros érpárra kerül. Ezek a rétegek más és más célból különféle járulékos információkat fűznek a tranzakció leírásához. A három réteg a következő:

1. Tranzakciós réteg
2. Adatkapcsolati réteg
3. Fizikai réteg

A tranzakció útját a rétegek és az eszközök között az 5.8. ábrán láthatjuk.



5.8. ábra. A tranzakció útja a rétegek és az eszközök között

A tranzakció egy 3-4 db 32 bites adategységből álló fejlécből, egy 0-1024 db 32 bites adategységből álló rakományból, és egy 32 bites CRC-ből (ellenőrző összeg) áll. A fejléc tartalmazza sok egyéb járulékos információ mellett a tranzakció típusát (a PCI ”C” (command) jelének felel meg: memory read/write, I/O read/write, configuration read/write, stb.), a bájtsorrendet (PCI-ban ”BE” határozta meg), a címet (PCI-ban Address). A rakományban található az átvinni kívánt adat (ami a PCI-ban az órajel szerint sorban megjelent az A/D vezetékeken). A CRC mező a teljes üzenetre (fejléc + rakomány) számolt ellenőrző összeg, amit a tranzakció célpontja ellenőriz.

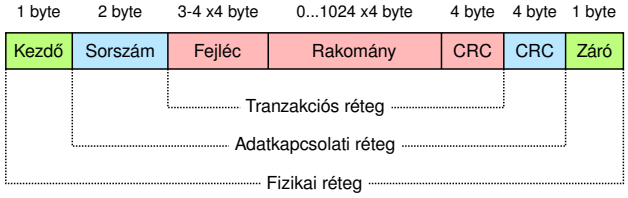
Az ily módon összecsomagolt tranzakció ezután az adatkapcsolati réteg kezelésébe kerül. Az adatkapcsolati réteg feladata, hogy ezt a tranzakciós csomagot a közvetlen szomszédba hibamentesen eljuttassa. Ennek érdekében egy újabb fejléccel egészíti ki a csomagot, amibe egy automatikusan inkrementált sorszámot ír. A csomag végére egy újabb CRC kódot helyez el, melybe már a sorszámot is belekalkulálja.

Az adatkapcsolati réteg által kiegészített csomagot ezután a fizikai réteg kapja meg. A fizikai réteg feladata a csomag átvitele a soros vonalon. A fizikai réteg a csomag elejére és végére egy-egy speciális szimbólumot helyez, hogy a vevő oldal könnyen be tudja határolni, hogy hol kezdődik és hol végződik a csomag. Ezt követően a bájtok (a teljes csomag minden bájtjára vonatkozik) bitjeit összekeveri, és a 8 bitet 10-re egészíti ki annak érdekében, hogy elegendő 0-1 váltást tartalmazzon ahhoz, hogy a másik oldal órajel nélkül is dekódolni tudja. A csomag ekkor elhagyja az eszközt.

A csomag hasznos tartalmát az átvitel során terhelő összes hozzátett fejléc és egyéb célú bájt az 5.9. ábrán látható. Ez alapján könnyen kiszámolható, hogy egyetlen 32 bites adategység (4 bájt) átviteléhez 1+2+3*4+1*4+4+4+1 = 28 bájtos forgalom terheli a soros kapcsolatot, amihez még hozzá kell számítani a 8 bitet 10 biten továbbító kódolás hatását is: 28*10/8 = 35 bájt, ami nem elhanyagolható mértékű rezsi.

A tranzakciót megcélzó eszköz a csomagot rétegek szerint alulról felfelé, vagyis az előbbivel fordított sorrendben bontja ki. Először a fizikai réteg állítja vissza a soros bitfolyamból a csomagot (ehhez meg kell találnia a csomaghatárokat, vissza kell állítani a 8-ból 10 bitbe kódolás és a bitösszekeverés hatását is), majd továbbítja az adatkapcsolati rétegnek.

Az adatkapcsolati réteg ellenőrzi a CRC-t, és ránéz a sorszámra. Ha rossz a CRC, vagy a sorszám nem pontosan 1-gyel nagyobb az előzőnél (vagyis kimaradhatott, elveszhetett pár csomag), akkor egy negatív nyugtát, ellenkező esetben pozitív nyugtát küld vissza a feladónak. A feladó, ha rossz sorszám miatt kapta a negatív nyugtát, újra tudja küldeni a korábbi, sikertelenül továbbított csomagot (ehhez persze minden, még nem nyugtázott csomagot meg kell őrizni).



5.9. ábra. PCI Express tranzakciós csomag részei

A tranzakciós réteg csak akkor kap szerepet, ha a csomag célba ért (tehát a köztes kapcsolókban nem). Ellenőrzi a CRC-t, ha rossz, negatív nyugtát küld. Ha jó volt a CRC, akkor kiolvassa a fejlécből, hogy mit is kell csinálnia a rakományban lévő adatokkal (pl. a fejlécben a cél címtartományába eső ”memory write” szerepel, akkor az adatokat a rakományból a címnek megfelelő helyre írja).

Egyes tranzakciókra válaszolni kell (non-posted transactions), másokra nem (posted transactions). Például egy memória írási műveletre nem kell válaszolni: a kezdeményező összeállítja a csomagot a címmel és a beírandó adattal, és útjára bocsátja a tranzakciót. A memória olvasási művelethez azonban szükség van válaszra: a kezdeményező egy tranzakciós csomagban elküldi a címet a memória olvasás paranccsal, amire válaszként a címzettnek egy újabb (ellentétes irányú) csomagban kell visszaküldenie a címről kiolvasott adatot.

A tranzakciós csomagok továbbítására hatással van az alábbi két tényező is:

Quality of Service támogatás A PCI Express lehetőséget biztosít a különböző elvárásokat támasztó perifériák igényeinek figyelembe vételére. A tranzakciós csomagok fejléce tartalmaz egy 3 bites ”forgalmi osztálynak” (Traffic Class, TC) nevezett mezőt, amivel a tranzakció jelezheti, hogy mennyire sürgős a dolga. Ahogy a tranzakció a csomagban halad a kapcsolók hálózatán át a periféria és a Root Complex között, ezt az információt figyelembe véve lesz, ahol megelőz másokat a fontossága miatt, máshol esetleg épp őt előzi meg egy még fontosabb csomag.

Forgalomszabályozás A PCI Express eszközök a beérkező csomagokat egy átmeneti tárolóban, bufferben tárolják, míg feldolgozásuk meg nem kezdődik. Ennek a buffernek a mérete azonban véges, és ha a küldő gyorsabban küldi a csomagokat, mint ahogy azt a fogadó fel tudja dolgozni, akkor hamar megtelik, és számos csomag tárolási hely hiányában elvész. Ennek megakadályozására a PCI Expressben egy hitelérték (kredit) alapú forgalomszabályozást vezettek be. Ennek lényege, hogy a fogadó fél rendszeresen közli a küldővel, hogy mekkora tárolási kapacitással rendelkezik (ez a kreditérték). Ha a küldő úgy ítéli meg, hogy a küldeni szándékozott csomag ennél nagyobb (nincs elég kredit), akkor a küldést egész addig késlelteti, míg a fogadónál megfelelő tárolási kapacitás nem keletkezik.

5.3.3. Megszakításkezelés

A PCI Express megszakításkezelése a processzor és a programozó szemszögéből teljesen kompatibilis a PCI-jal. Mint láttuk, a PCI-ban kétféleképpen lehetett megszakítást kérni:

- A 4 megszakítás jelzésére szolgáló vezeték egyikének alacsonyra húzásával
- Üzenettel jelzett megszakítás (Message Signaled Interrupt, MSI) segítségével

A PCI Expressben az üzenettel jelzett megszakítás (MSI) teljesen ugyanúgy működik, mint a PCI-ban. A PCI Expressben azonban nincs meg a megszakítás jelzésére szolgáló 4 vezeték. A problémát újabb üzenetek bevezetésével oldották meg: a periféria olyan üzenetet küldhet a Root Complexben működő megszakításvezérlőnek, hogy az most tekintse 0-ba húzottnak az egyik (amúgy nem létező) interrupt lábát, majd kisvártatva egy újabb üzenettel jelzi, hogy most vegye úgy, hogy ismét 1-be emelte a jelet. Az üzenetek alapján a Root Complex lejátssza a PCI-nál már látott eljárást a megszakítás kezelésére.

5.3.4. Konfiguráció

A PCIe eszközök nem 64, hanem 1024 konfigurációs regiszterrel rendelkeznek, melyből az első 64 kompatibilitási okokból megegyezik a PCI-ban rögzítettekkel. Ugyancsak kompatibilitási okokból, a PCI-nál látott címzés és regiszter kiolvasási/írási mechanizmus a PCIe esetében is működik. A 64-en felüli konfigurációs regiszterek eléréséhez azonban kicsit másféle, 64 bites címzést kell alkalmazni (nem részletezzük).

5.4. Az USB csatolófelület

Az USB (Universal Serial Bus) ma már a külső perifériák illesztésének de-facto szabványa, felhasználói szemmel valószínűleg mindenki ismeri. Ebben a fejezetben bepillantunk az USB működésének részleteibe, megnézzük, hogy a perifériakezelés korábban megismert alapelveit hogyan valósították meg a szabvány kifejlesztői. Ahogy a PCI/PCI Express esetében az 5.2. és az 5.3. fejezetekben, most is egy áttekintő képet kívánunk nyújtani, nem célunk egy minden részletre kiterjedő referenciát adni. Több, részletesebb leírás a [7], a [11] és [6] irodalmakban található.

5.4.1. Az USB története, képességei

Az USB kifejlesztését a PC-s perifériák csatlakoztatása körül kialakuló egyre nagyobb káosz motiválta. Mint már említettük, a PC-be illeszthető kártyák száma erősen korlátos, sokféle csatlakozó volt használatban, mindegyikhez külön hosszabbítót, kábelt, átalakítót, stb. kellett gyártani.

1994-ben hat vállalat, a Compaq, a DEC, az Intel, a Microsoft, a NEC és a Nortel kezdte meg a munkát egy új univerzális illesztőfelületen, később a HP, a Lucent és a Philips is csatlakozott. Az elsődleges célpont tehát a PC volt, de a szabvány megszületése után minden vezető architektúra (ARM, SPARC, Alpha, Power) USB támogatást kapott.

Az USB főbb tulajdonságai:

- Szabványos fizikai csatlakozók minden periféria számára
- Több periféria is rákötheto ugyanazon csatlakozóra (hub segítségével, lásd később)
- Összesen 127 USB eszköz is lehet egyetlen rendszerben
- Perifériák csatlakoztatása nem terheli az I/O címteret és nem kell újabb IRQ vonal sem
- A perifériákat menet közben, a számítógép kikapcsolása nélkül lehet csatlakoztatni, ill. eltávolítani
- A perifériák automatikus felismerése és konfigurálása
- Olcsó vezérlőáramkörök, csatlakozók és kábelek
- Az USB eszközök tápellátása (bizonyos határokig) az USB kábelen keresztül megoldható
- Alacsony fogyasztás támogatása

5.4.2. Az USB felépítése

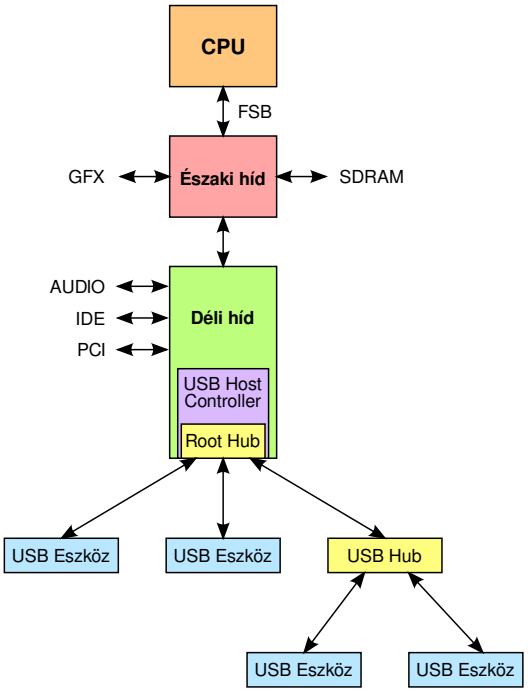
Az USB rendszerekben a perifériák fa topológiába vannak rendezve. A fa levelei maguk a perifériák, a fa csomópontjai (az elágazások) pedig a *hub*ok.

Minden USB rendszerben az USB vezérlő (*USB host controller*) végzi a busz vezérlésével kapcsolatos minden teendőt. Egy rendszerben több host controller is lehet, ilyenkor a rájuk csatlakozó eszközök független buszokat alkotnak. Az USB host controller tartalmazza az USB fa gyökerét, a *root hubot* (egy host controller pontosan egy root hubot tartalmaz). A root hubhoz, mint minden más hubhoz is, újabb hubok, vagy perifériák (*USB device*) csatlakozhatnak. Akár a PCI, az USB is megengedi, hogy egy fizikailag csatlakoztatott periféria több logikai eszközből, funkcióból (*USB function*) álljon. Ha egy periféria több funkciót tartalmaz, akkor azt az USB szabvány összetett perifériának (*compound device*) nevezi.

Az USB helyét felépítését az 5.10. ábra szemlélteti. Egy modern PC-ben a déli híd tartalmaz host controllert, de a host controller lehet egy PCI/PCI Express periféria is.

Az ábrát ismét szemügyre véve felmerül a kérdés, hogy az USB mitől is busz. Általánosságban egy buszról megszokhattuk, hogy az egy osztott közeg, melyen minden szereplő mindent hall, vannak rajta masterek, amik kezdeményezhetnek adatátvitelt, és slave-ek, amik csak válaszolhatnak a master kéréseire. Első pillanatra meglepő lehet, de az USB ezeknek a szempontoknak mind megfelel. Az USB-ben egyetlen master van: a root hub, minden más szereplő slave. Minden, amit a master forgalmaz, a fa mentén az összes slave-hez eljut (már amelyik adatsebességben kompatibilis – lásd később). Amelyik eszközre vonatkozik, az odafigyel, a többi nem. Az ellentétes irányú adatátvitel is hasonlóan zajlik: a master szól, hogy melyik perifériától vár adatot, majd elhallgat, és a megcímzett perifériától elvárja, hogy ez idő alatt tegye rá a "kábelére" a kért adatot, ami a fa mentén feljut a masterhez. Mivel a root hub az egyetlen master, egyetlen periféria sem kezdeményezhet adatmozgást, csak akkor küldhet adatot, ha megszólítják. Így arbitrációra egyáltalán nincs is szükség.

Mivel az USB ténylegesen egy busz, sújtja a buszok legnagyobb hátránya is: minél több periféria van hozzá csatlakoztatva, annál többen kénytelenek osztozni a busz sávszélességén.



5.10. ábra. Az USB rendszer helye és felépítése

5.4.3. Az USB 1.1

Az 1998-ban kiadott 1.1-es szabvány volt az USB első igazán széles körben elterjedt verziója. Mivel a szabvány későbbi verziói ugyanazokra az elvekre épülnek, az USB-vel való ismerkedést a legegyszerűbb, 1.1-es verzióval kezdjük.

Az USB 1.1 kétféle adatátviteli sebességet támogat: az 1.5 Mb/s-os kis sebességet (low speed, LS) és a 12 Mb/s-os teljes sebességet (full speed, FS). Bizonyos, tipikusan az egyszerűbb és olcsóbb eszközök csak a kis sebességű kommunikációt támogatják, míg más eszközök mindkét adatsebességen képesek működni. Előbbieket LS, utóbbiakat FS eszközöknek nevezzük.

Tranzakciók

Az USB rendszerekben a perifériákkal *tranzakciók* segítségével lehet kommunikálni. Megkülönböztetünk kis és teljes sebességű tranzakciókat, attól függően, hogy 1.5 vagy 12 Mb/s sebességgel történnek-e. A hubok az FS tranzakciókat az LS eszközök felé nem továbbítják, így minden eszköz csak olyan sebességű tranzakciókat lát, amelyeket támogat (a hub tudja, hogy egy adott portján LS vagy FS eszköz van-e).

A tranzakciókat mindig az egyetlen master, a root hub indítja. A tranzakció többek között tartalmazza a megcímzett periféria azonosítóját, azt, hogy ki- vagy bemeneti tranzakcióról van szó, valamint az átvinni kívánt adatokat. Kimeneti (a periféria felé irányuló) tranzakció esetén az átvinni kívánt adatokat a root hub, bemeneti tranzakció esetén pedig a megcímzett periféria teszi a buszra.

A tranzakciókat *keretekbe* (frame) szervezik. Egy keret pontosan 1ms ideig tart, ami FS adatsebesség esetén 1500 bájtot jelent. Egy keret természetesen több tranzakciót is szállíthat. Mivel a tranzakciók mérete, ahogy a perifériával cserélendő adatmennyiség is, változó lehet, a keretek nem feltétlenül vannak teljesen kitöltve.

Mivel a perifériák nagyon sokfélék lehetnek, eltérő igényeket támasztanak a busz felé. Van periféria, ami hibamentes adatátvitelt igényel, másnak ennél fontosabb a minél kisebb késleltetés, stb. A sokféle elvárás kiszolgálására az USB négy *adatátviteli módot* (transfer type) vezet be, ezek pedig: az *izokron* (isochronous), a *bulk*, az *interrupt* és a *control* adatátviteli mód.

Az *izokron* adatátviteli mód. A késleltetésérzékeny perifériák számára mindennél fontosabb, hogy az adatfolyam folyamatos, ingadozásmentes legyen. Ilyenek pl. az USB-re csatlakoztatható hangszóró, mikrofon, web

kamera, stb. Számukra vezették be az izokron adatátviteli módot, melynek jellemzői a következők:

- A periféria minden keretben garantáltan kap lehetőséget adatátvitelre (tranzakcióra), tehát soha nem fordulhat elő, hogy a többi periféria tranzakciói kiszorítják egy keretből.
- Az izokron adatátviteli módot folytató perifériák tranzakcióira a rendszer nem garantálja a hibamentességet. Érthető módon felesleges pl. egy USB hangszóró esetén a hibásan átvitt adat újraküldésével bíbelődni, hiszen a sikeres újraküldésre az adat már aktualitását vesztheti (a hangszórónak már korábban ki kellett volna játszania), miközben a meghibásodott adat a hangminőségben csak alig észrevehető romlást okoz.
- Az izokron tranzakciók maximálisan 1023 bájt hasznos adatot szállíthatnak, a hozzá kapcsolódó járulékos adatok (pl. fejléc) mérete 9 bájt. Izokron tranzakciót csak az FS perifériák végezhetnek.

A *bulk* adatátviteli mód. Számos periféria számára nem lényeges a kis adatátviteli késleltetés, annál inkább fontos a hibamentes átvitel. Ide tartozik pl. az USB printer, szkenner, pendrive, stb. Ezek a perifériák képesek tolerálni, hogy időnként kimaradnak 1-1 keretből, a felhasználó türelmetlenségén kívül ez semmilyen gondot nem okoz. Erre a célra vezették be a bulk adatátviteli módot. Jellemzői:

- A bulk tranzakciók késleltetésére nincs garancia.
- A bulk tranzakciók hibamentes átvitele érdekében az USB minden hibadetektáló és -javító eszközét beveti.
- A bulk tranzakciókban az átvitt adat mérete 8, 16, 32 vagy 64 bájt lehet, a hozzá kapcsolódó járulékos adatok ehhez még 13 bájtot tesznek hozzá. Bulk tranzakciót csak az FS perifériák végezhetnek.

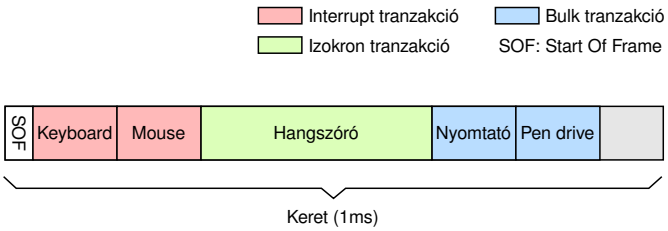
A *control* adatátviteli mód. A perifériák konfigurálását, működését befolyásoló parancsokat a control tranzakciók segítségével lehet hozzájuk eljuttatni. A control tranzakciók szabványos, vagy egyedi (gyártóspecifikus) parancsokat is szállíthatnak. Tulajdonságai:

- Ha vannak átvitelre váró control tranzakciók, akkor a rendszer a soron következő keret méretének 10%-a erejéig helyet szorít nekik.
- Garantált a hibamentes átvitel.
- A control tranzakciókat az LS és az FS eszközök is támogatják. Az adatméret LS esetben legfeljebb 8, FS esetben legfeljebb 64 bájt lehet, a hozzá kapcsolódó járulékos adatok mérete LS esetben 63, FS esetben pedig 45 bájt.

Az *interrupt* adatátviteli mód. Mivel az USB-ben a root hub az egyetlen master, a periféria nem kezdeményezhet adatátvitelt, meglepő módon még megszakításkérést sem. Az USB-ben azt, hogy a perifériának van-e közlendője, periodikus lekérdezésekkel lehet megállapítani. Erre szolgál az interrupt tranzakció. Amikor a perifériát csatlakoztatják, annak kiolvasható eszközleíróiból (device descriptor) meg lehet állapítani, hogy milyen sűrűn (hány keretenként, vagyis hány ms-onként) igényel lekérdezést. Interrupt átviteli módot használnak pl. az USB billentyűzetek és egerek. Főbb jellemzői:

- Az interrupt tranzakciók mindig beviteli irányultságúak (a periféra küld adatot a root hub felé).
- A lekérdezési periódus 1 és 255 keretidő között állítható.
- Garantált a hibamentes átvitel. Ha egy tranzakció hibásnak bizonyul (az ellenőrzőösszeg alapján), a következő keretben megtörténik az újraküldés.
- Az interrupt tranzakciókat az LS és az FS eszközök egyaránt támogatják. Az adatméret LS esetben legfeljebb 8, FS esetben legfeljebb 64 bájt lehet, a hozzá kapcsolódó járulékos adatok mérete LS esetben 19, FS esetben pedig 13 bájt.

Amikor a keretek tranzakciókkal való feltöltése zajlik (5.11. ábra), a periodikus tranzakcióknak, vagyis az izokron és az interrupt átviteli módhoz tartozó tranzakcióknak van elsőbbsége. Hogy a többi periféria is életben maradjon, a periodikus tranzakciók nem foglalhatnak többet a keret 90%-ánál. Ha egy újabb periféria csatlakozik a buszra, és periodikus átvételi módot kér olyan sávszélességgel, ami már nem elégíthető ki, akkor nem léphet be a rendszerbe. A keret fennmaradó 10%-án osztoznak a control és a bulk tranzakciók, de ahogy már fentebb említettük, a control tranzakciók a keret 10%-át garantáltan megkapják. Így legrosszabb esetben, amikor sok a periodikus és a control tranzakció, az is előfordul, hogy a keret egyetlen bulk tranzakciót sem tud elvinni.



5.11. ábra. Példa a keret kitöltésére

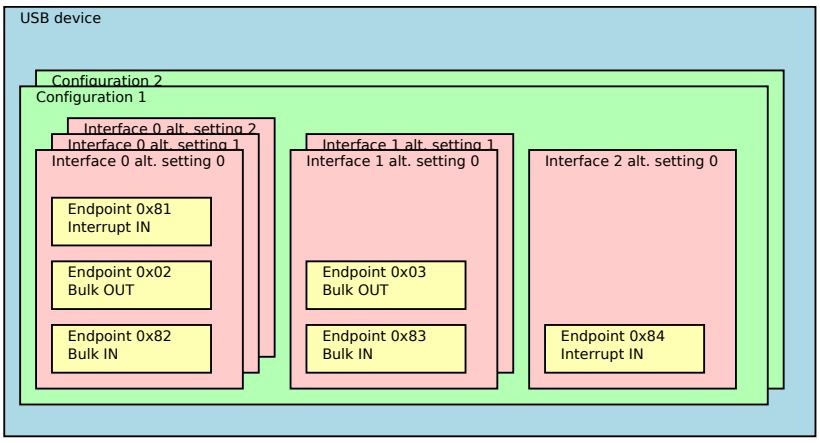
5.4.4. USB perifériák felépítése

Egy USB eszközben több logikai periféria is lehet (hasonlóan a PCI eszközöknél látott logikai funkciókhoz), amelyeket az USB-ben *interface*-eknek hívunk. Az interface az, ami a felhasználó számára egy perifériát jelent. Az interface-hez tartozik az eszköz osztály (device class) és protokoll, ami alapján az operációs rendszer megtalálhatja az eszközt kezelni képes meghajtó programot (driver-t). Több interface-t tartalmazó eszköz esetén mindegyik interface-nek saját leírója van, ezeket jellemzően külön driver kezeli. Például elterjedt az a megoldás, hogy egy USB 3G modem eszköz a modem interface mellett tartalmaz egy mass storage interface-t is, amely a modem driver-t tartalmazza. Ily módon a számítógéphez csatlakoztatva az operációs rendszer képes először felinstallálni a driver-t, mielőtt a modem-et használatba venné. A kombinált vezeték nélküli egér és billentyűzet dongle-k is két interface-t tartalmaznak, egyet az egér és egyet a billentyűzet számára.

Az egyes interface-ek tehát egymástól nagy mértékben függetlenül működnek. Egy-egy interface azonban tovább osztható adatátviteli csatornákra, *endpoint*-okra. Az endpoint jellemzői:

- Egy endpoint egy egyirányú adatátviteli csatorna.
- Az USB eszközön belül minden endpoint egy 7 bites egyedi azonosítóval (címmel) rendelkezik. Az azonosító 8. bitje (MSB) a csatorna irányát jelöli (0: kimenet, 1: bemenet).
- Az endpoint a négy adatátviteli mód valamelyikét használja.

A korábbiakban részletezett tranzakciók valójában mind egy-egy adott endpoint felé irányulnak. Egy interface tehát tipikusan több endpoint-ból állhat, amelyek vegyesen lehetnek be- és kimeneti irányultságúak, és vegyesen használhatják az egyes átviteli módokat. Például egy USB kamera esetén lehet egy izokron bemeneti végpont a video adatfolyam számára, egy másik az audio adatfolyam számára, és egy interrupt módú kimeneti végpont a konfiguráláshoz. Másik példa: az USB billentyűzet két végpontot tartalmaz, mindkettő interrupt módú, az egyik bemeneti (itt jönnek a lenyomott billentyűk kódja, a scan kódok), a másik pedig kimeneti (a LED-ek beállításához).



5.12. ábra. USB perifériák felépítése

A végpontok konfigurációját, vagyis azt, hogy hány van belőlük, és milyen a sávszélesség igényük, a periféria rendszerbe illesztésekor le kell rögzíteni. Ez a kötöttség sajnos sok periféria számára túl korlátozó lehet, pl. egy USB kamera többféle felbontást támogathat, melyekhez más és más izokron sávszélességigény tartozik. Pont

az ilyen perifériák részére vezették be az úgynevezett alternatív beállításokat, az *alternate settings*-et, melyek között a driver szabadon váltogathat. Az alternate setting megváltozása esetén (pl. a kamera felbontásának átállításakor) a rendszer kilépteti az interface-t, majd belépteti újra az új paraméterekkel, feltéve, természetesen, hogy a sávszélesség igényei továbbra is kielégíthetők.

További lehetőség az USB szabványban, hogy egy USB periféria több alternatív konfigurációt, *alternate configuration*-t tartalmazhat. Az alternatív konfigurációk önálló interface készlettel rendelkeznek, átkapcsoláskor a régi konfigurációhoz tartozó interface-ek mind kilépnek, és az újhoz tartozó interface-ek belépnek. Szinte olyan, mintha egy teljesen új USB eszközt csatlakoztattunk volna. Például a legtöbb USB Ethernet illesztő képes az USB szabványban rögzített Ethernet-over-USB üzemmódban, és a Microsoft-hoz köthető NDIS üzemmódban is működni. A két üzemmód más és más interface-eket ír elő, a váltást az alternatív konfigurációk közti átkapcsolással szokás megoldani (5.12. ábra).

Adatátvitel az USB-ben

Most pedig részleteiben áttekintjük, hogy is jutnak el az adatok a szoftvertől a perifériáig. Az adatátvitelben a következő szereplők játszanak szerepet:

- A már megismert *hardver* szereplők:
  - a host controller, a root hubbal,
  - hubok,
  - a periféria.
- A közreműködő *szoftver* szereplők:
  - a periféria eszközmeghajtója (drivere),
  - az USB driver,
  - az USB host controller driver.

A periféria eszközmeghajtója kezdeményezi az adatátvitelt. Pl. egy printer driver kap egy kinyomtatni való lapot egy applikációtól. Az ő dolga a nyomtatás "lebeszélése" a nyomtatóval, pl. hogy a kinyomtatandó lap képpontjai mellett mit kell közölni vele a felbontás, a papírméret, stb. beállításához. A periféria eszközmeghajtójának azonban nem kell foglalkoznia az USB-n való továbbítás részleteivel (ezzel is megkönnyítik az eszközmeghajtó-fejlesztők dolgát). Az USB továbbítás érdekében az egyetlen dolga, hogy összeállít egy adatátviteli kérést (I/O request packet, IRP), és ezt továbbítja az USB drivernek. Az IRP-k felépítése egyszerű: legfontosabb mezői a periféria címe, az adatátvitel iránya, és egy pointer az átvinni kívánt adatokra.

Az USB driver megkapja az adatátviteli kéréseket, és akkora méretű darabokra bontja, hogy azok egy tranzakcióban átvihetők legyenek, majd a darabokat továbbadja a host controller drivernek. (Ezen a részletességi szinten az USB driver egyéb feladataival nem foglalkozunk.)

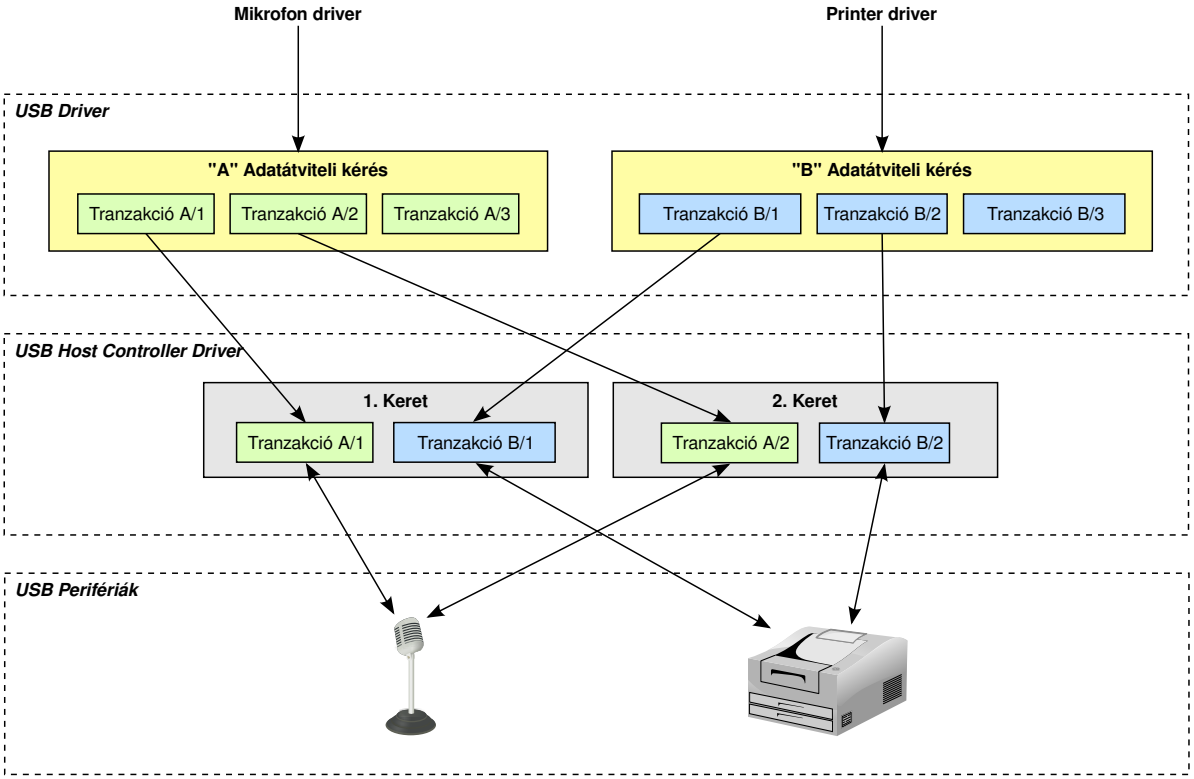
A host controller driver feladata az adatátvitel tényleges lebonyolítása. Az USB drivertől kapott átvinni szándékozott adatdarabokból tranzakciókat képez, a tranzakciókat (több adatátviteli kérésből származókat vegyesen) pedig a már megismert szabályok szerint szétosztja a keretek között. Egy láncolt listát (tulajdonképpen egy várakozási sort) tart karban, melynek minden eleme egy-egy keret tranzakcióit tartalmazza. Minden egyes keretidőben, tehát 1ms-onként leemeli a lista első elemét, és kiteszi a buszra (5.13. ábra).

Tápellátás az USB kábelen keresztül

Az USB egyik kellemes tulajdonsága, hogy a perifériák tápellátást – bizonyos határokig – az USB kábelen keresztül is meg lehet oldani, ugyanis a jelvezetékek mellett a kábel 5V tápfeszültséget is szállít.

A root hub minden egyes csatlakozójából legfeljebb 500mA áramot lehet felvenni, ezzel kell a rá csatlakozó, belőle táplálkozó eszközöknek (huboknak és perifériáknak) gazdálkodnia. Azt is figyelembe kell venni, hogy az USB szabvány megköveteli, hogy minden hub port legalább 100mA áramot biztosítson a rá csatlakozó eszköznek. Így például, ha a root hub egy portjára egy USB kábelből "táplálkozó" hubot teszünk, akkor annak a hubnak legfeljebb 4 portja lehet, mivel minden porton köteles legalább 100mA-t adni, és a hub saját fogyasztását sem szabad figyelmen kívül hagyni. Így ki is merült az 500mA-es keret. Egy USB hub azonban nem feltétlenül az USB kábelen oldja meg a tápellátását, vannak saját tápellátással rendelkező hubok is, melyek négynél akár sokkal több portot is tartalmazhatnak, akár mindegyiken a maximális 500mA áramellátást biztosítva.

Azt, hogy egy eszköz mennyi áramot igényel a normál működéséhez, a csatlakoztatáskor, a konfiguráció során ki lehet olvasni az eszközleíró tábláiból. Ha a port, amire rákötötték, ezt nem képes biztosítani, akkor az eszköz



5.13. ábra. Adatátvitel USB-n

nem lép be az USB rendszerbe. (A konfigurációs fázis során, amikor mindez kiderül, minden eszköz meg kell, hogy elégedjen a minimálisan mindig rendelkezésre álló 100mA árammal.)

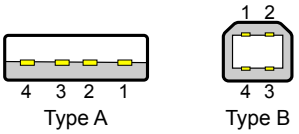
Kábelek és csatlakozók

Az USB kábel mindössze 4 vezetéket tartalmaz. Ebből kettő a periféria tápellátásáért felelős (földelés és +5V), a másik kettő pedig a jelátvitelhez szükséges. A jelátvitel fizikai miként jéről itt csak annyit jegyeznénk meg, hogy differenciálisan történik, vagyis a két jelvezeték (D+ és D-) feszültségének különbsége a mérvadó. A teljes sebességű (FS) és az alacsony sebességű (LS) eszközök kábelei nem egyeznek meg. A magasabb adatsebesség elérése érdekében az FS kábelek kétszeresen árnyékoltak, és a jelvezetékek csavart érpárt alkotnak. Az LS kábelek esetén a kábel külső árnyékolása elhagyható, és a jelvezetékeket sem kell megcsavarni, így az LS kábelek sokkal vékonyabbak, hajlékonyabbak, és nem utolsósorban olcsóbbak az FS kábeleknél.

Amikor a szabványosítás során a csatlakozókat tervezték, az alábbi szempontokat tartották szem előtt:

- A csatlakozók legyenek robusztusak – hiszen sok USB eszközt, rendeltetésükből adódóan, sokszor csatlakoztatunk a számítógéphez, ill. választunk le onnan
- Ne lehessen a csatlakozót az aljzatba fordítva bedugni
- A kábel két végén lévő eszköz összekötése értelmes legyen – ne lehessen pl. a root hub két aljzatát egy kábellel összekötni, vagy két perifériát egymással közvetlenül összekötni.

A legutóbbi szempont érdekében kétféle csatlakozótípust specifikáltak: ”A” és ”B” típusút. Az ”A” típusú (Type A) csatlakozóval csatlakozik a periféria a hubhoz, azaz az USB fában a éleknek (melyek a kábeleknek felelnek meg) a gyökér felé eső oldalán van ”A” típusú csatlakozó. A ”B” típusú (Type B) csatlakozó a kábel másik végén van, a periféria felé (ha az USB fát nézzük, a gyerekek felé eső oldalon), persze csak akkor, ha a kábelnek az a vége nincs fixen a perifériához rögzítve. A csatlakozókat és a lábkiosztást az 5.14. ábra mutatja be.



Tüske	Szerep
1	Táp (+5V)
2	Adat –
3	Adat+
4	Föld

5.14. ábra. USB csatlakozók és a tüskék kiosztása

Konfigurálás

Az USB eszközök konfigurálását az USB driver végzi a rendszer indításakor, illetve új eszköz csatlakoztatásakor. Mindkét esetben ugyanaz történik, a rendszer indítása az USB szempontjából olyan, mintha az eszközöket sorban egymás után csatlakoztatnánk. A konfigurálás során az eszköz egyedi címet kap, ami alapján a későbbiekben azonosítani lehet, lekérdezik tőle a tápellátási igényeit, az izokron sávszélesség szükségletét, stb.

Ha csatlakoztatunk egy eszközt, az alábbi események zajlanak le, mielőtt az adatforgalomban részt venne:

1. A hub érzékeli, hogy új eszköz csatlakozott rá. Ezt onnan tudja, hogy az új eszköz rögtön magasba húzza FS esetben a D+, LS esetben a D- jelvezetéket. Tehát a hub nemcsak azt látja, hogy megjelent egy periféria az egyik portján, hanem azt is rögtön érzékeli, hogy teljes sebességű átvitelre képes-e.

2. Az USB Driver időnként végigkérdezi a hubokat, hogy van-e újonnan csatlakoztatott periféria. Ha van, elkezdi annak beállítását.

3. Először is az USB driver kiad egy parancsot a hubnak, hogy küldjön reset jelet a kérdéses portra. A reset hatására az új periféria a nullás címre küldött tranzakciókra válaszolni fog.

4. A nullás címre küldött control tranzakciók segítségével az USB driver már ki tudja olvasni a periféria eszközleíró táblázatainak tartalmát. Ezekben a táblázatokban szerepelnek a periféria különböző jellemzői: gyártó, típus, az eszköz jellege, paraméterei, stb. (hasonlóan a PCI konfigurációs regisztereihez).

5. Az USB driver egy nullás címre küldött control tranzakció segítségével egyedi címet ad a perifériának. Ezért is nagyon fontos, hogy rendszerindításkor az eszközök konfigurálása egyesével történjen: garantálni kell, hogy mindig csak egyetlen periféria hallgasson a nullás címre, pont az, aminek épp a beállítása zajlik.

6. Ezután az USB driver ellenőrzi az új periféria eszközleíró táblázatainak tartalma alapján, hogy az eszköz beléphet-e az USB rendszerbe. Ehhez például szükséges, hogy tápellátási igényét a hub ki tudja elégíteni, vagy hogy sávszélesség igénye kiszolgálható legyen (pl. nem engedhető meg olyan sok izokron eszköz csatlakoztatása, hogy a keretenkénti adatfogalmi igényük összességében meghaladja egy keret kapacitását). Ha valamilyen feltétel nem teljesül, az USB driver utasítja a hubot a kérdéses port letiltására. Ha minden megfelelő volt, akkor a konfiguráció véget ér, az új eszköz részt vehet az adatforgalomban.

5.4.5. Az USB 2.0

Az USB 2.0-ás szabványt 2000-ben jelentették be. Az egyik legfontosabb előrelépés a 480 Mb/s-os ”nagy sebességű” (high speed, HS) adatátvitel megjelenése volt, ami az USB felhasználási körét nagyban kiszélesítette.

Főbb változások az USB 1.1-hez képest

A korábbi LS és FS eszközök az USB 2.0-ás rendszerekben is változatlanul működnek. Az USB 2.0 a korábbi csatlakozókkal és kábeltípusokkal kompatibilis maradt, sőt, új kábelekre sincs szükség, mert a HS eszközök az FS eszközök kábeleit használják. A mobil eszközök igényeinek kiszolgálására a szabvány utólagos kiegészítéseként bevezettek néhány új, kisebb méretű csatlakozót is.

Az adatátvitel logikája ugyanaz maradt, az USB 2.0 ugyanazokat az adatátviteli módokat támogatja, mint elődje. Az eszközök konfigurálásában, a tápellátás szabályaiban sincs számottevő változás.

Megváltozott viszont a keretidő, a korábbi 1ms-os keretek helyett az USB 2.0 125μs-os mikrokereteket használ. A 480 Mb/s-os sebességgel számolva ez keretenként 7500 bájtot jelent, szemben a régebbi verzió 1500 bájtjával.

Megnövelték a tranzakciókban szállítható adatok maximális mennyiségét, és bevezettek egy új forgalomszabályozási technikát a keretek minél hasznosabb kitöltésére is.

Nagy sebességű tranzakciók

Lássuk, miben különböznek a HS tranzakciók az LS és FS tranzakciókhoz képest.

Az izokron tranzakciók. Az izokron tranzakciókban átvihető maximális adatmennyiséget 1023-ról 1024 bájtra emelték. Ennél nagyobb változás is van: míg az USB 1.1-ben az izokron tranzakció azt jelentette, hogy minden keretben történik adatátvitel, az USB 2.0 mikrokeretenként egy helyett maximum három izokron tranzakciót enged meg egy periféria egy izokron adatkapcsolatára vonatkozóan.

A bulk tranzakciók. A bulk átvitel tranzakciónkénti adatmennyisége 512 bájt lett (fixen). E mellett a periféria felé irányuló tranzakciókhoz egy új forgalomszabályozási lehetőséget is bevezettek. Az USB 1.1 egyik problémája az volt, hogy a host nem tudott megbizonyosodni afelől, hogy a periféria készen áll-e az adat fogadására. Leküldte az adatot, és csak a negatív nyugtából tudta levonni a következtetést, hogy nem kellett volna megtennie. Ez természetesen nagyon gazdaságtalan megoldás, hiszen a feleslegesen leküldött adat feleslegesen foglalta a keret egy részét, amit esetleg hasznosabb tranzakciókkal lehetett volna kitölteni. Az USB 2.0-ban bevezették a PING tranzakciót, amivel a host megkérdezheti a perifériát, hogy mehet-e az adat. Szintén újdonság, hogy a tényleges adatok fogadása után a periféria nem csak pozitív/negatív nyugtát adhat (sikeres/sikertelen vétel), hanem azt is jelezheti, hogy sikeres volt a vétel, de most egy kis időt kér a következő adag küldéséig.

A control tranzakciók. A control tranzakciók adatmennyiségét 64 bájtban rögzítették (se több, se kevesebb nem lehet), és lehetővé tették az előbb megismert PING megoldás alkalmazását is.

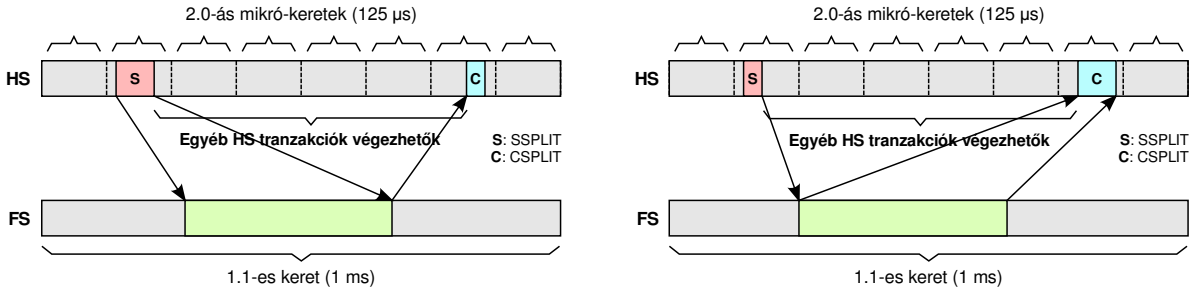
Az interrupt tranzakciók. Az interrupt tranzakciókban szállítható adatmennyiség LS esetben legfeljebb 8, FS esetben legfeljebb 64 volt, ezt a HS üzemmódban drasztikusan, legfeljebb 1024-re emelték. Ahogy az izokron tranzakcióknál is láttuk, az interrupt adatátviteli módban is 3-ra nőtt az egy mikrokeretben elhelyezhető tranzakciók száma (perifériánként és adatkapcsolatonként).

Kis- és teljes sebességű tranzakciók

Mivel megváltozott a keretidő és a bitsebesség, az LS és FS eszközök nem tudnak közvetlenül részt venni az USB 2.0 adatforgalmában. A hubok feladata, hogy a portra csatlakozó LS és FS eszközök részére a kompatibilitást biztosítsák, vagyis hogy azt a látszatot keltsék, hogy azok egy USB 1.1 rendszer részét képezik.

Nagy vonalakban ez a következőképpen zajlik: a root hub nagy sebességgel villámgyorsan leküldi az adatokat a hubnak, amire a lassú eszköz csatlakozik. Ezután a hub szép lassan LS vagy FS sebességgel lejátssza az adatcserét a perifériával, miközben a busz többi részén nagyban zajlik tovább a gyors kommunikáció. Mikor a root hub úgy gondolja, hogy a lassú adatátvitel már befejeződhetett, rákérdez a hubnál a végeredményre. Ezt hívják *osztott tranzakciónak* (SPLIT transaction). Azért osztott, mert a tranzakció nem időben folytonosan zajlik elejétől a végéig, mint ahogy eddig megszokhattuk: címzés – adatforgalom – nyugta módjára, hanem két darabban: a lassú tranzakciót meg kell indítani (ezt egy SSPLIT üzenet végzi), majd később, miután sok egyéb HS tranzakció is lezajlott a buszon, vissza kell térni rá, és le kell zárni (CSPLIT üzenet). Ha az adatforgalom a periféria felé irányul, akkor az SSPLIT végzi a címzést, és tartalmazza a leküldendő adatot is, míg a CSPLIT csak a nyugtát kéri le (sikeres/sikertelen volt-e a küldés – izokron tranzakciónál, mint tudjuk, nincs nyugta). Ha az adatot a root hub kéri a perifériától, akkor az SSPLIT csak a címzést és a kérést tartalmazza, a periféria által küldött adat a CSPLIT üzenetben kerül vissza a root hubhoz (5.15. ábra). Fontos, hogy mint minden tranzakciót, az SSPLIT és CSPLIT tranzakciót is csak a root hub kezdeményezheti. Amíg a root hub a periféria hubjától el nem kéri a CSPLIT-tel a lassú tranzakció eredményét, addig azt a hubnak a saját memóriájában tárolnia kell. Az USB 2.0-ban a hubok feladata tehát jóval összetettebb, mint az USB 1.1-ben volt.

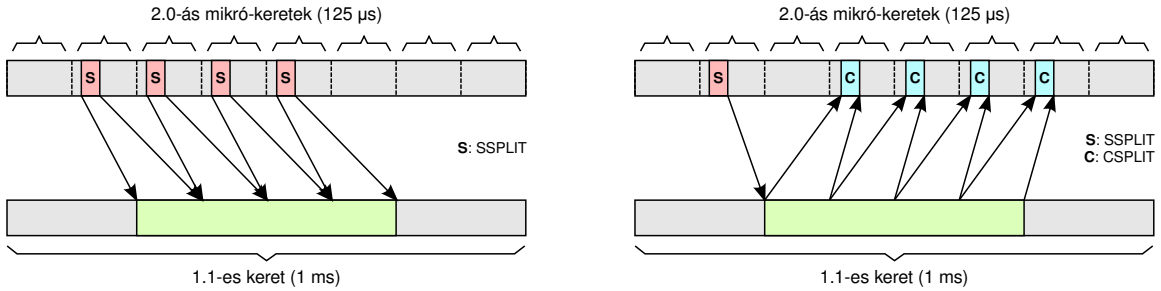
Az izokron tranzakciók esetén ennél még egy kicsit árnyaltabb a kép. Az izokron adatátviteli mód, mint láttuk, egyenletes, állandó rátájú adatátvitelt valósít meg. Ennek a filozófiának a jegyében az FS izokron tranzakciókat feldarabolva, több részletben viszi át az USB a nagy sebességű szakaszon (lásd 5.16. ábra). Ezzel a busz terhelése is egyenletesebbé válik, a HS tranzakciók elől így minden mikrokeretben a lehető legkevesebb helyet viszi el. Az



a) Periféria felé b) Periféria felől

5.15. ábra. Osztott bulk tranzakciók

ábrán az is látható, hogy a periféria felé küldött adatot nem kell nyugtázni, ennek megfelelően CSPLIT tranzakció sem szükséges.

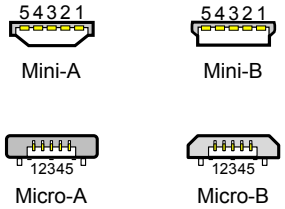


a) Periféria felé b) Periféria felől

5.16. ábra. Osztott izokron tranzakciók

Új csatlakozó típusok

A ”normál” méretű csatlakozók mellett, a vékony mobil eszközök igényeinek eleget téve, az USB 2.0-ban megjelentek a kisebb méretű, ”mini” és ”micro” csatlakozók is. A korábban olyan logikusnak tűnő ”A” és ”B” típus szerinti megkülönböztetés ezek esetében azonban nem ilyen egyszerű. Ugyanis pl. egy mobil eszköz egyszer USB perifériaként viselkedik (pl. amikor a PC-hez kötjük), másszor pedig USB hostként (pl. amikor billentyűzetet, pen drive-ot, vagy USB-s GPS-t kötünk rá). Pusztán emiatt két USB aljzat felszerelése a mobil eszközre helypazarlás lenne, ezét van olyan aljzat, amibe a Micro ”A” és a Micro ”B” csatlakozó is belefér. Ebbe az aljzatba egy ”A” típusú csatlakozót dugva a készülék hostként, ”B” típusú csatlakozót dugva pedig perifériaként viselkedik. A szerepek tisztázása miatt ehhez szükség van még egy tüskére a csatlakozóban. Az új csatlakozókat és a tüskék kiosztását az 5.17. ábrán láthatjuk.



Tüske	Szerep
1	Táp (+5V)
2	Adat-
3	Adat+
4	Szerep: leföldelve host, nyitva periféria
5	Föld

5.17. ábra. A mini és a micro USB csatlakozók és a tüskék kiosztása

5.4.6. Az USB 3.0

Az USB 3.0 szabvány (2008-ban jelent meg) legnagyobb újdonsága az USB 2.0-hoz képest a 3.8 Gb/s-os adatátviteli sebesség megjelenése, ami olyan perifériák csatlakoztatását is lehetővé tette, melyeknek az USB 2.0 sávszélessége kevés volt: nagy sebességű diszkek, SSD-k, Blu-ray írók, nagy felbontású videokamerák, monitorok.

Főbb változások az USB 2.0-hoz képest

Az új, 3.8 Gb/s-os adatátviteli sebességet szupergyorsnak (Super Speed, SS) hívják. Ilyen nagy sebesség elérése érdekében természetesen nagyon sok ponton alapvető változásokat kellett eszközölni az USB 2.0 megoldásaihoz képest (többek között annyira alapvetőeket, hogy a jelátvitel nem egy érpáron, hanem kettőn zajlik, és azon is teljesen másként). Ezek az alapvető változások sajnos a régebbi eszközökkel való kompatibilitás rovására mentek volna, ami senkinek nem lett volna jó, hiszen a felhasználók a birtokukban lévő sok-sok USB 2.0-ás perifériát továbbra is használni szeretnék.

A megoldás az lett, hogy az USB 3.0-ás tulajdonképpen két buszból áll: egy USB 2.0-ás buszból, ami az LS, FS és HS tranzakciókat szállítja, valamint az új buszból, ami az SS eszközöket szolgálja ki. A két busz teljesen elkülönül egymástól. Mint később látni fogjuk, a kábelekben is egymás mellett futnak az USB 2.0 és 3.0 vezetékei, és a csatlakozók is úgy vannak kialakítva, hogy a régi és a 3.0-ás perifériák is egyaránt csatlakozni tudjanak.

Tekintsük át először azt, hogy mit tartott meg az USB 3.0 az USB 2.0 megoldásaiból:

- A busz továbbra is fa szerkezetű, hubokból és perifériákból áll
- Az USB 1.1 óta használt átviteli módok (bulk, izokron, interrupt, control) megmaradtak
- Az eszközök konfigurálásának menete ugyanaz
- Az eszközleíró táblázatok formátuma ugyanaz (de a tartalma kibővült)
- Továbbra is kizárólag a root hub kezdeményezhet adatátvitelt
- Az LS, FS, és HS perifériák kezelése ugyanúgy történik (nem meglepő, hiszen az továbbra is a régi buszon zajlik)

Tulajdonképpen felhasználói szempontból nem sok minden változott. Az alkalmazások, ill. eszközmeghajtók továbbra is ugyanolyan adatátviteli kéréseket küldenek az USB drivernek, és továbbra is a tranzakciók képezik az adatátvitel alapegységeit. A változások csupán az adatátvitel miként jét érintik, ahogy azt a PCI és a PCI Express közötti átmenetnél láttuk.

A szupergyors (SS) adatátvitel érdekében a következő változásokat vezették be:

- Megszüntették az üzenetszórást. Míg az USB 2.0-ban a busz minden szereplője látta a tranzakciókat, az USB 3.0-ban közvetlen *forgalomirányítással* érnek célba az adatok. A hubok csak abba az egyetlen irányba továbbítják az adatokat, amerre a címzett periféria található.
- Bevezették a *store & forward* adattovábbítást: a hub a saját átmeneti tárolójába helyezi el az adatokat, míg azok teljesen meg nem érkeznek hozzá, majd csak ezután továbbítja a megfelelő kimeneti port irányába.
- Egy helyett két érpáron zajlik a kommunikáció: az egyik érpáron a periféria felé, a másikon a root hub felé tartó adatok utaznak (dual simplex adatátvitel). A korábbi egyszerű differenciális jelátvitel helyett a PCI Expressnél látotthoz nagyon hasonló megoldást vezettek be: a 8 bites adategységek bitjeit összekeverik, és 10 biten kódolják az ennél a sebességnél már nem elhanyagolható elektromágneses interferenciák hatásának minimalizálására.
- Javítottak a forgalomszabályozáson: ha egy periféria nem áll készen az adatok vételére, nem kell a root hubnak folyamatosan lekérdezgetni, maga a periféria képes szólni, ha már jöhetnek az adatok (asynchron notification). (Ez nem jelenti azt, hogy a periféria *kérhet* adatátvitelt! Azt továbbra is csak a root hub teheti meg.)
- További javítás a forgalomszabályozásban: nem kell minden tranzakciót egyenként nyugtázni, ”burst” üzemmódban a tranzakciók mehetnek szorosan egymás után, és a végén egy közös nyugta jelzi a vétel sikerét.
- Lényegesen jobb energiamenedzsment. A busz finom felbontásban szabályozni tudja a ritkán használt perifériák ”ébrenlétét”. Korábban, az USB 2.0-ban az üzenetszórás miatt minden perifériának folyamatosan nyomon kellett követnie a forgalmat, hátha neki szóló tranzakció érkezik. Az USB 3.0-ban csak a címzett periféria kapja meg az adatot, a többi addig alvó állapotba léphet.

- Az eddigi 100mA-ről 150mA-re növelték a perifériák számára rendelkezésre bocsátott áramot, a maximum pedig 500mA-ről 900mA-re nőtt. A 900mA már elegendő egy nagyobb merevlemez, vagy egy kisebb fogyasztású monitor önálló tápellátására is.

Adattovábbítás közvetlen útvonalválasztással

Ebben a fejezetben egy kicsit közelebbről is megismerkedünk az USB 3.0 szupergyors adattovábbításának egy részletével, az útvonalválasztással.

Az USB drivernek pontos képe van az USB fa mindenkori helyzetéről, hiszen rendszerindításkor és az eszközök csatlakoztatásakor ő végzi azok konfigurációját. Pontosan tudja, hogy egy adott címmel rendelkező periféria hol helyezkedik el, a neki szánt adatokat mely hubokon keresztül kell számára elküldeni. Amikor az adat egy csomagban a periféria felé útjára indul, hozzá csatol egy listát, amiben le van írva, hogy melyik útba eső hub melyik portja felé továbbítsa az adatot. A huboknak pedig nincs más dolga, mint eleget tenni ezeknek a csomaghoz csatolt útvonalirányítási információknak.

Kicsit részletesebben ez a folyamat a következőképpen zajlik. Amikor egy hub csatlakozik a buszra, az USB driver elküldi neki, hogy mi a mélysége (mélység alatt azt értjük, hogy az USB fában hányadik szinten helyezkedik el). A hub ezt az információt eltárolja. Az USB 3.0-ban az USB fa mélységét 6-ban korlátozták, vagyis az adatok legfeljebb 5 hubon haladnak át, mielőtt célba érnének. A hub mélysége tehát 1 és 5 közé kell, hogy essen. Ennek megfelelően minden csomag egy 5 elemből álló listát tartalmaz (route string), aminek az *n.* eleme azt jelenti, hogy a továbbítás *n.* lépésében a hub melyik portjára küldje tovább a csomagot. A route string minden eleme egy 4 bites szám, ugyanis egy hubnak nem lehet 15-nél több portja. Például, ha egy hub mélysége *n*, és kap egy csomagot, akkor a csomagban lévő route string *n.* eleméből fogja tudni a továbbítás irányát. Ha az *n.* elem nulla, akkor a csomag címzettje maga a hub, ellenkező esetben az *n.* elem egy kimeneti portot azonosít.

Kábelek és csatlakozók

Az USB 3.0-ban, ahogy láttuk, tulajdonképpen két busz dolgozik, az egyik az LS, FS és HS, a másik az SS tranzakciókat bonyolítja. Ennek megfelelően az ”A” és ”B” típusú csatlakozókat is két részre bontották (lásd 5.18. ábra). Az ”A” típusú csatlakozót úgy alakították ki, hogy be lehessen dugni régi, USB 2.0 aljzatba is, de tartalmaz még egy tüskesort, ami, USB 3.0 aljzatba dugva, az új szupergyors adatátvitelt szolgálja ki. Így, ha egy USB 3.0-ás eszközt 2.0-ás aljzatba helyeznek, megmarad a lehetőség, hogy ha lassabban is, de az eszköz üzemelni tudjon. A ”B” típusú csatlakozónak is két része van, a felső része felelős az SS átvitelért. Az USB 3.0 perifériákon található aljzat fogadni tudja a 2.0-ás kábeleket is, de nyilván ilyenkor a szupergyors sebességről le kell mondani.

5.18. ábra. ”A” és ”B” típusű csatlakozók USB 3.0-ban

A lábkiosztást az 5.2 táblázat tartalmazza. A tápfeszültség (+5V és GND), valamint a 2.0-ás jelvezeték (D- és D+) mellett egy új érpár felelős a periféria felé irányuló SS adatforgalom (SSRX- és SSRX+), valamint a root hub felé irányuló SS adatforgalom (SSTX- és SSTX+) szállításáért.

A ”B” típusú csatlakozók két extra lábbal rendelkeznek (DPWR és DGND), melyek speciális célokat szolgálnak. Ezeken a lábakon maga a periféria tud tápfeszültséget szolgáltatni a rá kapcsolódó eszköznek. Ennek látszólag nincs sok értelme (hiszen pont a perifériát szeretnénk táplálni), de ez teszi lehetővé a vezetéknélküli USB megvalósítását. A vezetéknélküli átvitelhez kell egy USB aljzatba dugható adó-vevő pár. Az egyiket a megfelelő hubra kell csatlakoztatni, ez a hubból vehet fel tápfeszültséget, a másikat pedig a periféria ”B” aljzatába, ez pedig erről a két lábról (DPWR és DGND) jut tápfeszültséghez. Ilyenkor a periféria tápellátását természetesen külső módon (akkumulátorról, vagy hálózati tápegységről) kell megoldani.

Végül, de nem utolsósorban az USB 3.0 csatlakozónak is van mobil eszközökhöz való ”micro” megfelelője (5.19. ábra).

Láb	Szerep
1	Táp (+5V)
2	Adat-
3	Adat+
4	Föld
5	SSRX-
6	SSRX+
7	Föld
8	SSTX-
9	SSTX+

a) ”A” típusú csatlakozók

Láb	Szerep
1	Táp (+5V)
2	Adat-
3	Adat+
4	Föld
5	SSRX-
6	SSRX+
7	Föld
8	SSTX-
9	SSTX+
10	DPWR
11	DGND

b) ”B” típusú csatlakozók

5.2. táblázat. USB 3.0 csatlakozók lábkiosztása

5.19. ábra. Micro USB 3.0 csatlakozó

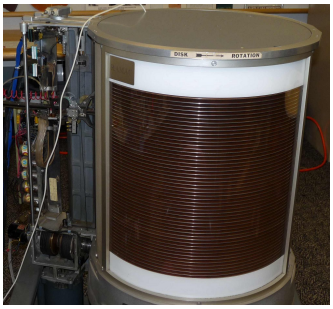
6. fejezet

Háttértárak

Vannak perifériák, melyek ma már minden számítógép részét képezik. Közülük a háttértár a legfontosabb, részben, mert teljesítménye az egész rendszer teljesítményére kihat, részben pedig azért, mert fontos szerepet játszik a virtuális tárkezelésben, mellyel hamarosan közelebbről is megismerkedünk (lásd később). Ebben a fejezetben a két legelterjedtebb háttértár technológiát, a mágneses adattároláson és forgó lemezen alapuló merevlemezeket (Hard Disk Drive, HDD), valamint a félvezető-alapú SSD-ket (Solid State Drive) mutatjuk be. Előbbiek nagy kapacitásuk és olcsó előállítási költségük, utóbbiak pedig kis méretük és nagy sebességük miatt nyertek teret a számítástechnika különböző szegmenseiben.

6.1. Adattárolás merevlemezen (HDD)

Az első, számítógépes háttértárként szolgáló forgólemezes adattárolót az IBM fejlesztette ki. Hosszas kutató-fejlesztőmunka eredményeként 1955-ben mutatták be, majd 1956-tól forgalmazták RAMAC 305 néven (6.1. ábra). Ez a szekrény méretű háttértár mágneses elven működött, és tulajdonképpen a mai merevlemezek ősének tekinthető. A mai merevlemezek azóta is ugyanazon az elven működnek, ugyanazokból a komponensekből építkeznek, csupán a dimenziók változtak az évek múlásával. A RAMAC 305 súlya 1 tonna volt, 50 darab 24 colos lemezből állt (minden lemez mindkét oldalon tárolt adatot, vagyis 100 adathordozó felülete volt), melyek között két író/olvasó fej liftezett föl-alá. Minden lemez 100 sávot tartalmazott, elérési ideje 1 s volt, és az egész rendszer összesen 5 millió 6 bites karakter tárolására volt képes.



6.1. ábra. A világ első merevlemeze

6.2. ábra. Az 1 colos Hitachi Microdrive

Az azóta eltelt évtizedekben a merevlemezek kapacitása, sebessége és megbízhatósága nőtt, előállítási költsége pedig csökkent. A miniatürizálás 1999-re elérte azt a szintet, hogy 1 col méretű, 170 MB kapacitású, Compact Flash adapterrel rendelkező mini-merevlemezek gyártása is lehetővé vált, szintén az IBM-nek köszönhetően (6.2. ábra, ezt a formátumot 8 GB kapacitással 2006-ig forgalmazzák, amikor is a félvezető alapú tárolók kiszorították a piacról).

6.1.1. A forgólemezes adattárolás elve

A forgólemezes adattárolás a fonográffal kezdődött, ahol a lemezek sávjaiban a mélyedések hordozták a (mechanikai jellegű) információt. Az optikai meghajtók és a merevlemezek ugyanezen az elven működnek, a különbség csak

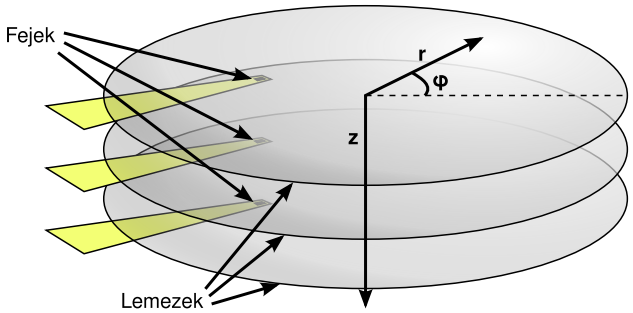
annyi, hogy előbbi esetben a felület fényvisszaverő tulajdonsága, utóbbi esetben pedig egy ferromágneses anyag mágneses mezőjének iránya tárolja az adatot.

A forgólemezes adattárolók részei:

- a kör alakú *lemez*, ami az adatokat tárolja (vagy csak az egyik, vagy mindkét oldalán),
- valamint a *fej*, ami a hordozóról leolvasott mechanikai, optikai vagy mágneses jelet elektromos jellé alakítja.

A lemezen egy adat helyét két paraméter írja le: a középponttól mért (radiális) távolsága, ami a lemezen egy kört határoz meg, valamit egy fix referenciaponthoz képesti szöge, ami az adat körön belüli pozícióját adja meg. Ahhoz, hogy egy lemezen tárolt adaton műveleteket végezzünk, a fejnek a lemez adott pontja felett kell lennie. Ezt kétféleképpen lehet elérni: a fej mozgatásával, illetve a lemez mozgatásával. A forgólemezes adattárolók mindkettőt mozgatják, a fejet a megfelelő radiális pozícióba tolják (ezt hívják *seek*-nek), majd a lemezt megforgatják, hogy a keresett adat a fej alá kerüljön. A fej csak akkor mozog, amikor szükséges, viszont a lemezt a gyakorlatban folyamatosan forgatni szokták, hiszen a lemezek súlya viszonylagosan nagy, sok időt és energiát emésztene fel a rendszeres felpörgetés.

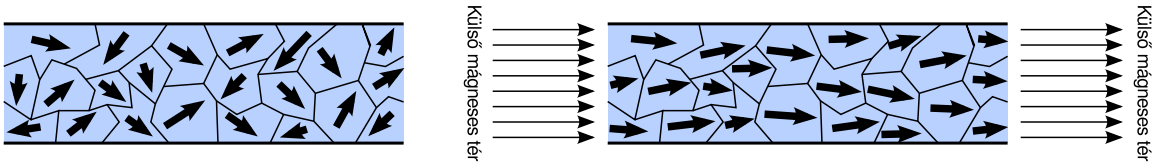
A kapacitás növelése érdekében több lemezt is egymás fölé lehet tenni, illetve a lemezek mindkét oldalát lehet adathordozóként használni. Ez az adatok helyének azonosítása szempontjából a radiális (r) és a szög (ϕ) mellett egy harmadik dimenziót (z) jelent (6.3. ábra). Cserélhető adathordozóknál (mint pl. a CD, vagy a megboldogult floppy lemez) a többlemezes megoldás nem terjedt el.



6.3. ábra. Az adatok helyének azonosítása

6.1.2. Mágneses adattárolás

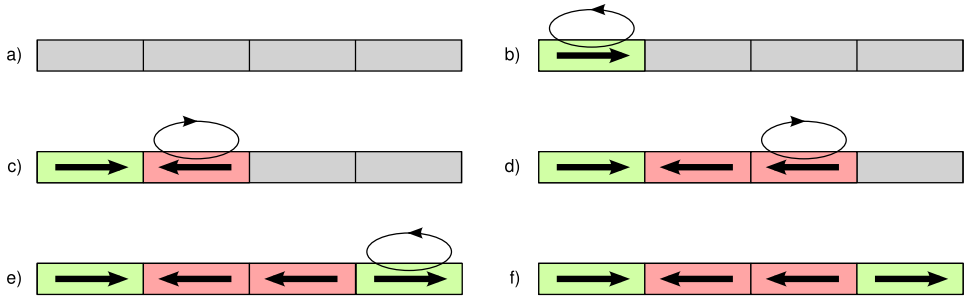
A mágneses adattárolók a ferromágneses anyagok tulajdonságait használják ki. A ferromágneses anyag jellemzője, hogy külső mágneses tér hatására mágnesessé válik, és tartósan az is marad a külső mágneses tér eltávolítása után is. Ilyen anyag például a vas, a kobalt, a nikkel. A ferromágneses anyag kis cellákból, *mágneses doménekből* áll, melyeken belül a spinek iránya azonos, vagyis az atomok mágneses momentuma azonos. A doménen belül tehát erős mágneses tér van. Ennek ellenére az anyag (pl. egy darab vas) nem feltétlenül mágneses, hiszen a domének mágnesességének iránya véletlenszerű lehet, összességében kioltva egymás hatását. Külső mágneses tér hatására azonban a domének a tér irányával párhuzamosan rendeződnek (6.4. ábra). A domének új, egyirányú elrendeződése a külső mágneses tér eltávolítása után is megmarad, ezzel az anyag mágnesessé válik. Ez az állapot az anyag számára ugyan nem minimális energiájú, de nagyon stabil.



a) Külső mágneses tér hatása előtt b) Külső mágneses tér hatására

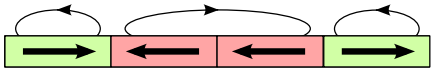
6.4. ábra. Ferromágneses anyagok domén szerkezete és viselkedése

Ezt a viselkedést úgy tudjuk adattárolás céljára kihasználni, hogy a fej segítségével a lemez egy kis része felett lokálisan mágneses teret hozunk létre, ami a fej alatt a lemezt átmágnesezi. Ezt a folyamatot a 6.5. ábra szemlélteti. A fej először az első adattároló szakasz fölé mozog (pontosabban a lemez forog be alá), és azt átmágnesezi. A következő lépésben a fej a következő szakasz fölé kerül, ahol ellentétes mágneses tér indukálásával ellentétes irányba mágnesezi a lemezt. A fej szakaszról szakaszra haladva kialakítja a kívánt mágneses mintázatot a lemezen. Ez a mintázat az író fej eltávolítása után is megmarad. (A valóságban persze nincsenek ilyen szakaszok, a lemez egy folytonos ferromágneses anyaggal van bevonva.) Megjegyezzük, hogy a nagyobb adatsűrűség érdekében kezd elterjedni a vízszintes mágneses irányok használata helyett a függőleges adattárolás (perpendicular recording), de mi az egyszerűség kedvéért a továbbiakban is a vízszintes megoldáson keresztül mutatjuk be a merevlemez működését.



6.5. ábra. Az írási folyamat

A mágneses mintázat visszaolvasásához a lemez által kibocsátott mágneses mezőt (6.6. ábra) kell valamilyen érzékelővel detektálni. Az ilyen gyenge mágneses mezők irányának meghatározása azonban a gyakorlatban kivitelezhetetlen. E helyett a merevlemez olvasófeje a mágneses mező *megváltozását* képes érzékelni. Az ábra példáján az első és a második, valamint a harmadik és negyedik szakasz között változik meg a mágneses mező iránya. Az adatok tárolásának szokásos módja, hogy a mágneses mező megváltozása jelenti az 1-es, a meg nem változása pedig a 0-ás bitet. Tehát az ábrán látható mágneses mintázathoz az ”101” bitsorozat tartozik. Ennek az adattárolási módnak van egy mélyreható következménye: a biteket nem lehet egyesével módosítani, hiszen ha egy bitet meg akarunk változtatni, akkor az utána következő összes szakaszon ellentétére kell változtatni a mágneses mező irányát, hogy a többi bit értéke ne változzon meg. Ez az oka annak, hogy a merevlemezeken minden írási/olvasási művelet alapegysége egy nagyobb blokk. Csak teljes adatblokkot lehet írni a merevlemezre, illetve leolvasni onnan.



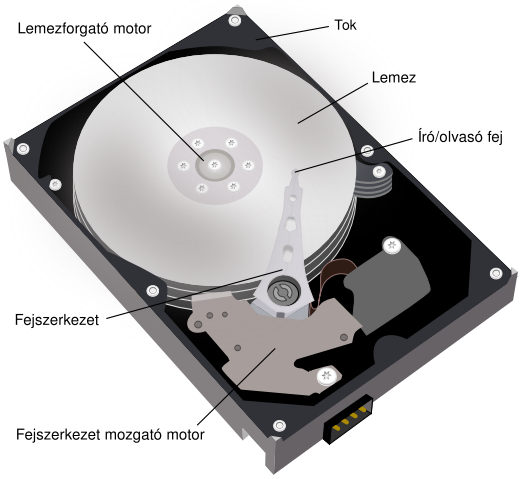
6.6. ábra. Az adathordozó által kibocsátott mágneses tér

6.1.3. A merevlemezek felépítése

A merevlemezek felépítését a 6.7. ábrán láthatjuk. A legfontosabb alkatrészek és szerepük:

- Jól záró alumínium *ház.* Mivel a merevlemezben a fej és a lemez között nagyon kicsi a távolság, a legkisebb bejutó szennyeződés is katasztrofális hibát okozhat.
- A *lemez(ek)* (disk) felszínén található a mágneses adathordozó réteg.
- A *motor* folyamatosan, megállás nélkül forgatja a lemezt.
- Az *író/olvasó fej* (head) felelős a tárolt adatoknak megfelelő mágneses mintázat létrehozásáért, illetve leolvasásáért.
- A *fejszerkezet* (attractor) a *fejszerkezet mozgató motor* segítségével a lemez (radiálisan) megfelelő pontjára mozgatja a fejet.

A továbbiakban áttekintjük ezeknek az alkatrészeknek a működési elvét, tulajdonságait.



6.7. ábra. A merevlemezek felépítése

A lemezek. A merevlemezek működése a lemezek anyagával szemben sok követelményt támaszt. A lemeznek vékonynak kell lennie, hogy ne foglaljon el túl sok helyet; könnyűnek kell lennie, hogy könnyű legyen forgatni; merevnek kell lennie, hogy ne torzuljon, és ne rezonáljon, amikor nagy sebességgel forog. Ezen kívül simának kell lennie, hogy a fej a lemez felszínén található mágneses adathordozó réteghez minél közelebb kerülhessen annak érintése nélkül. A lemez felszínére felvitt mágneses hordozónak nagy adatsűrűség mellett is stabil mágneses tulajdonságai kell, hogy legyenek (jól azonosítható legyen a mágnesezettség iránya).

A gyakorlatban a merevlemezek lemezei több rétegből állnak. A lemezek alapja, a szubsztrát, általában alumínium, amire egy vékony nikkel-foszfor réteget visznek föl, mert azt nagyon simára lehet polírozni. Napirenden van az alumínium lecserélése a hőre és mechanikai sérülésre kevésbé érzékeny üvegre, vagy kerámiára, melyek ugyan törékenyek, de a mai kis lemez átmérő mellett ez már nem olyan nagy hátrány (az üveg szubsztrát lemezek gyártását 2010-ben kezdték el, már több kereskedelmi termék is alkalmazza, főleg a 10.000 fordulat/perc sebességgel forgó nagy sebességű típusok).

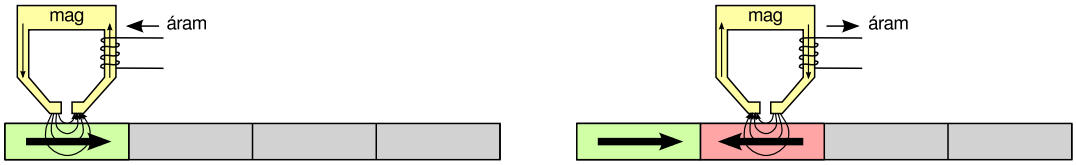
A szubsztrátra kerül a kb. 25 nm vastagságú ferromágneses adathordozó réteg, melynek anyaga kobalttal kevert vas-oxidot, króm-dioxidot, bárium-ferritet és egyéb fémeket tartalmaz. E fölé felvisznek egy 10 nm vastag szén réteget, hogy az adathordozó réteg ne legyen közvetlenül kitéve a környezeti hatásoknak, majd a lemez felszínét 1 nm vastagon kenőanyaggal vonják be.

A motor. A merevlemezben minden lemezt egy közös motor hajt. A motornak nagy megbízhatóságúnak kell lennie, hiszen éveken át tartó működést és esetleg több tízezer ki/be kapcsolást is ki kell bírnia. A fej pontos pozicionálása érdekében csak az egészen minimális vibrációval rendelkező típusok jöhetnek szóba. Fontos, hogy kicsi legyen a fogyasztása, ellenkező esetben a motor okozta melegedés csökkentené a merevlemez élettartamát. Lényeges szempontok az ütésállóság és az alacsony működési zaj is.

Az író/olvasó fej. Napjaink merevlemezeinek írófeje ugyanazon az induktív elven működik, mint a legelső merevlemezeké. Az induktív írófej egy gyűrű alakú magból (anyaga vas, vagy vas-nikkel ötvözet), és a rátekert tekercsből áll. A gyűrű a lemez felé eső felén meg van szakítva, egy egészen apró rés van rajta. Amikor a tekercsen áram halad át, mágneses teret indukál a magban, melynek iránya a tekercsben folyó áram irányától függ. A gyűrűn lévő rés két oldala két, ellentétes irányú mágneses pólust képez. A résnél a mágneses fluxus kilép a magból, és ez a mágneses fluxus mágnesezi a lemez adathordozó rétegét (6.8. ábra). Az írófejek gyártására a 80-as évektől kezdve egy nagyon hatékony, nagy pontosságú és olcsó technológiát sikerült kifejleszteni.

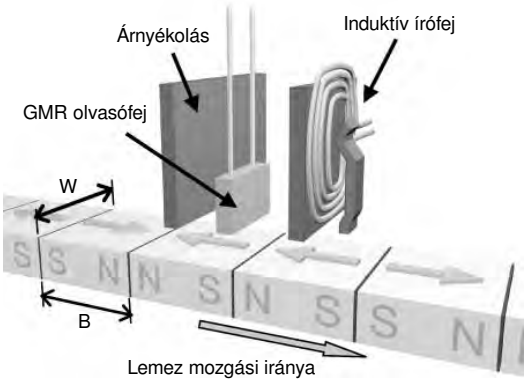
Egészen a 90-es évekig az imént megismert induktív írófejet használták a merevlemez tartalmának leolvasására is. Olvasás esetén a tekercsre nem adunk áramot. Ahogy a fej mozog az adathordozó réteg fölött, a hordozó mágneses mezőjének változása áramot indukál a tekercsben, amit az elektronika bitsorozattá alakít. Mint ahogy korábban említettük, a fej csak a változást érzékeli, tehát akkor kapunk áramot, ha két szomszédos szakasz mágnesezettségének iránya nem azonos. Változás esetén 1-es, annak hiánya esetén 0-ás bitet regisztrálunk.

Sajnos a mágneses mező változása által indukált áram nagyon kicsi. A 90-es években olyan olvasó fejeket



6.8. ábra. Induktív író fej

fejlesztettek ki, melyek a mágneses ellenállás (magnetoresistance) jelenségén alapulnak, azaz kihasználják, hogy bizonyos anyagok ellenállása függ a mágneses mező változásától. Az ezen az elven működő olvasófejeket MR fejeknek nevezik. Az MR fejre állandó áramot adnak, és a mágneses mező változását a fejen eső feszültség mérésével lehet detektálni (lásd: Ohm törvénye). Az MR fejeket később továbbfejlesztették, speciális, többrétegű felépítéssel még hatékonyabbá, érzékenyebbé tették. Ezeket a továbbfejlesztett fejeket GMR néven 1997 óta alkalmazzák. Az MR és a GMR fej csak olvasásra alkalmas, ezért ezek használata esetén egy külön induktív írófejre is szükség van (6.9. ábra).



6.9. ábra. GMR olvasó fej induktív író fejjel (rajz: Michael Xu, Hitachi)

6.1.4. Az adatok szervezése a merevlemezen

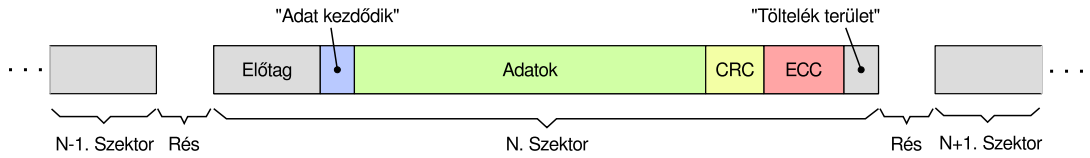
Adategységek

Ahogy már korábban beláttuk, a mágneses adattárolás tulajdonságai miatt csak nagyobb adategységeket, blokkokat lehet a merevlemezre írni, illetve leolvasni onnan. Ez rögtön felvet egy kérdést: változó blokkméretet használjunk, ami a tárolt adatok méretéhez pontosan igazodik, vagy definiáljunk fix méretű blokkokat, melyekkel könnyebb bánni, de semmi köze a tárolt adatok szerkezetéhez. Mindkettőre van példa a gyakorlatban.

Az IBM nagygépes rendszerei, a mainframe-ek az úgynevezett *Count-Key-Data* formátumot (CKD) alkalmazzák az adatok tárolására (mind a mai napig). A CKD rendszerben változó méretű adategységeket használnak (pl. egy adatbázis egy rekordja), melyeket két mezővel látnak el: a ”Count” mező a rekord azonosítóját (valamit opcionálisan a kulcs hosszát és az adat hosszát), a ”Key” pedig a hozzárendelt kulcsot adja meg (a ”Key” mező alapján is lehet adatokat keresni). A merevlemezen tehát ilyen azonosító-kulcs-adat hármasok sorakoznak egymás után – ezeket nevezzük *rekordoknak*, rövidebbek és hosszabbak vegyesen. Ennek a megoldásnak az az előnye, hogy pontosan alkalmazkodik a tárolt adatok méretéhez, hátulütője pedig (többek között), hogy a merevlemez szabad területei töredezettek lesznek. Ugyanis kisebb-nagyobb blokkok sorozatos létrehozása és törlése nyomán a szabad terület is kisebb-nagyobb lyukakból fog állni. Megeshet, hogy egy új, nagy méretű blokk már egyik lyukba sem fér bele, pedig összességében lenne elég szabad terület a merevlemezen. Hogy ez ne fordulhasson elő, időnként össze kell tolni a tárolt adatokat, hogy a szabad hely egy összefüggő területet alkosson. Ez a művelet a defragmentálás, a szabad területek töredezettség-mentesítése, ami egy időigényes művelet.

A változó adategységek alkalmazásának ma már egyeduralkodó alternatívája a fix adategységek használata. A fix hosszú blokkokat *szektor*nak hívják. A ma használatos merevlemez illesztő felületek (ATA, SCSI, FC

– lásd később) mind fix méretű szektorokkal dolgoznak. A leggyakoribb szektorméret 512 bájt, ez az ATA szabvány szektormérete, de az SCSI merevlemezek is legtöbbször ezt a méretet használják. Mint látni fogjuk, a fix méretű szektorokkal a merevlemeznek könnyebb bánni, de semmi köze nincs a tárolt adatok méretéhez, a fájlok szektorokba rendezését az operációs rendszernek kell megoldania.



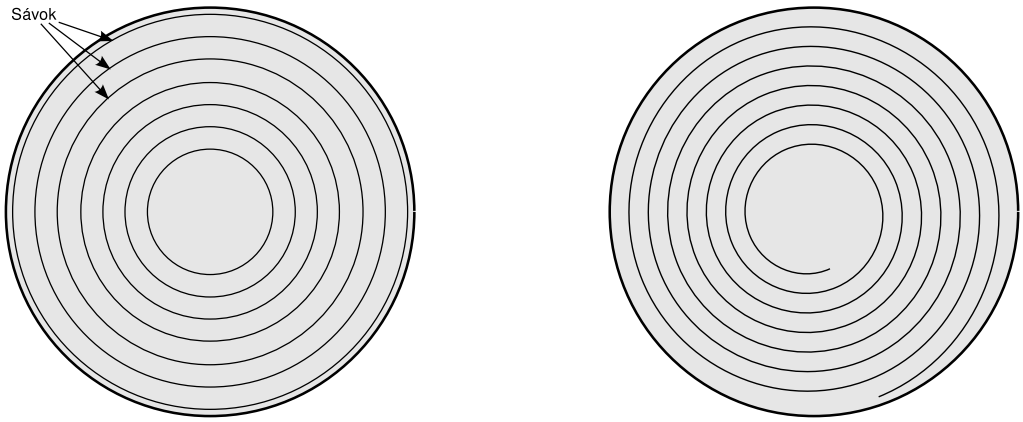
6.10. ábra. A szektorok felépítése

A 6.10. ábrán láthatjuk, hogy a merevlemez hogyan tárol egy szektort. A szektorokat *rés* (gap) választja el egymástól. Erre azért van szükség, hogy ha az író fej a kelleténél egy kicsit később kapcsolna ki, ne írjon bele a következő szektorba. Hogy miért is kapcsolna ki kicsit később az író fej? Ne felejtsük el, hogy ez egy mozgó (mégpedig igen gyorsan mozgó) alkatrészekből álló nagy adatsűrűséggel dolgozó fizikai rendszer, hajszálpontos működést egyszerűen képtelenség garantálni. Ezért, ahogy a következőkben megismerjük az adatok tárolásának és elérésének módját, jó pár ponton látunk majd ehhez a szektorközi réshez hasonló megoldásokat, melyek egy kis biztonsági tartalékot jelentenek, hogy a működésbeli bizonytalanságok ne okozzanak kárt. A rést egy *előtag* (preamble) követi. Ez egy speciális mágneses mintázat, mely alapján a fej bekalibrálhatja a hordozón lévő mágneses jel amplitúdóját és frekvenciáját, azaz megállapíthatja, hogy mennyit kell a jelen erősíteni, és hogy milyen sűrűn követik egymást az adatok az adott szektorban. Az előtag után egy *"adat kezdődik"* (address mark) jelzi a fejnek, hogy vége a kalibrálásnak, jönnek az adatok. Az *adatok*at hibadetektálásra szolgáló *ellenőrző összeg* (CRC, Cyclic Redundancy Check), és a néhány hiba kijavítását lehetővé tevő *hibajavító kód* (ECC, Error Correcting Code) követi. A szektor végén van egy kis töltelék terület (Flush Pad), ami alatt a szektorból kiolvasott utolsó bájtok is elhagyják a fejet, a dekódert és a hibajavító (ECC) áramköröket.

Az adatok azonosítása

Ahhoz, hogy szektorokat írjunk a merevlemezre, illetve olvassunk vissza onnan, tudnunk kell azonosítani őket. Már láttuk, hogy egy adat, ill. szektor helyét 3 dimenzióban adhatjuk meg: radiális távolság (r), egy referenciaponthoz képesti szög (ϕ), illetve az adathordozó réteg (z).

A szektorok a forgólemezes adattárolókban körkörösen helyezkednek el, vagy koncentrikus körökben, vagy spirálisan (6.11. ábra). A merevlemezekben a koncentrikus körök, az optikai meghajtókban pedig a spirális elhelyezés az egyeduralkodó.



a) Sávok koncentrikus körökben b) Spirális elrendezés

6.11. ábra. Szektorok elhelyezkedése az adathordozón

Most bevezetjük a merevlemezek világában használatos, szektorok azonosítására szolgáló terminológiát:

- Az azonos adathordozó rétegen fekvő, azonos radiális távolsággal rendelkező szektorok alkotják a *sáv*okat (track). Tehát az ugyanazon a sávon lévő szektorok koordinátái csak a szögben (ϕ) térnek el.
- Az egymás feletti adathordozó rétegek sávjai alkotják a *cilinder*eket (cylinder). Az egy cilinderen fekvő szektorok radiális távolsága (r) azonos, ϕ és z koordinátákban különbözhetnek. A cilindereket a bevett gyakorlat szerint kívülről befele számozzuk, a 0. cilinder az első, felhasználói adatoknak szánt cilinder. A 0. cilinder előtt (a perem felé) vannak az úgynevezett negatív cilinderek, melyeken a merevlemez a saját adatait, adminisztrációját tárolja.
- Azt, hogy egy szektor melyik adathordozó rétegen helyezkedik el, a hozzá tartozó *fej* (head) sorszámával adják meg.

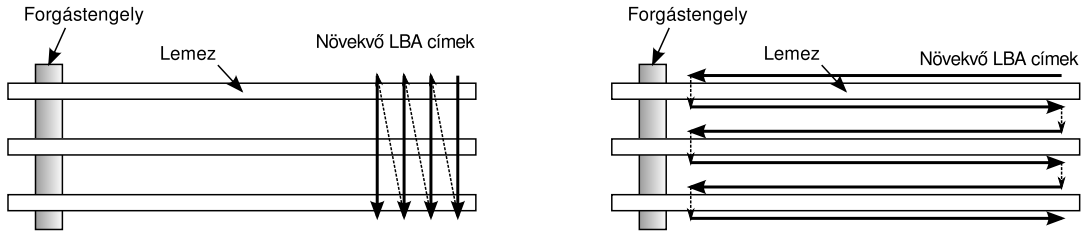
A merevlemezek tehát a szektorok helyét CHS (cylinder-head-sector) koordinátákkal azonosítják. A ”C” és ”H” koordináta megadja, hogy az adott szektor melyik cilinderen van, és a cilinderen belül melyik fejhez tartozik – így a ”C” és a ”H” együtt egy sávot azonosít. Végül az ”S” koordináta határozza meg, hogy a sávon belül hányadik szektorról van szó.

A szektorokra jellemzően másként hivatkozik a merevlemez, és a merevlemez külső felhasználója (az operációs rendszer). A merevlemez a szektorokat a fizikai elhelyezkedésüknek megfelelő CHS címzéssel azonosítja. Régebbi rendszerekben az operációs rendszer is ugyanezzel a CHS címzéssel azonosította a kiírni ill. beolvasni kívánt szektorokat a merevlemezekhez intézett kéréseiben. Ez azonban több problémát is felvetett:

- A merevlemez nem tudja a külvilág felé eltakarni a szektorok meghibásodásait. Ha egy szektor meghibásodik, arról az operációs rendszernek is tudnia kell, hogy azt a CHS címet a jövőben már ne használja többet.
- Egy praktikus probléma: az ATA szabványban a CHS címzésre rendelkezésre álló bitek száma túl kicsi. A cilinderek azonosítására 10 bit áll rendelkezésre (max. értéke 1023), a fejek száma maximum 255 lehet (fizikailag sosem használtak ilyen sok fejet), és maximum 63 szektor lehet egy sávon. Ilyen módon legfeljebb 8.4 GB-os merevlemezeket lehet megcímezni. (Egy alternatív címzési mód szerint 16383 cilinder, 16 fej és 63 szektor megengedett, a kapacitáskorlát ezekkel a számokkal is ugyanaz.)

Ez a két tényező vezetett oda, hogy a merevlemezek bevezették a logikai címzést (LBA, Logical Block Address). Az LBA nem jelent mást, mint hogy a szektorokat egyetlen számmal, a sorszámukkal azonosítják. Magyarán az operációs rendszer csak annyit mond a merevlemeznek, hogy hányadik szektort szeretné írni/olvasni, azzal nem kell foglalkoznia, hogy az a merevlemezen hol helyezkedik el. A merevlemez persze ezt a szektorsorszámot leképzi valahogy CHS koordinátákra, de hogy hogyan teszi, az az ő dolga. Ezáltal a merevlemez egy fekete doboz lett, az operációs rendszernek nem kell többé ismernie a pontos felépítését. Ha hibás szektor keletkezik, azt a merevlemez kifelé el tudja rejteni, tudni fogja, hogy oda többé nem tehet adatot, de ebből kifelé semmi sem látszik.

A logikai címek leképzése fizikai címekre többféleképpen is történhet. Két lehetséges megoldást mutat be a 6.12. ábra. A ”cilinder” stratégia szerint az egymás utáni LBA címek egyazon cilinder sávjaira kerülnek, majd ha a cilinder megtelt, akkor a merevlemez átvált a következő cilinderre. A ”szerpentin” stratégia ezzel szemben az egymás utáni LBA címeket egy adathordozó rétegen igyekszik elhelyezni.



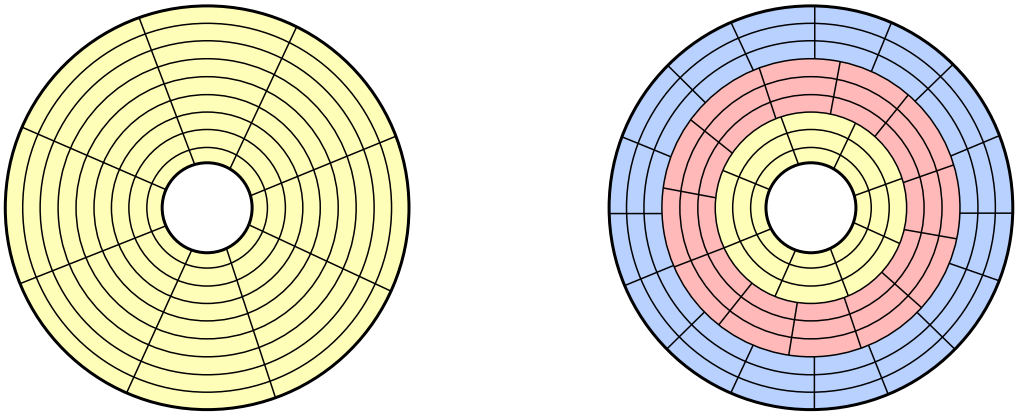
a) ”Cilinder” stratégia b) ”Szerpentin” stratégia

6.12. ábra. Logikai címek leképzése fizikai címekre

Zóna rendszerű adattárolás

A merevlemezek lemezei állandó szögsebességgel forognak. Régebben, az egyszerűség jegyében, minden sáv ugyanannyi szektort tartalmazott, hiszen ekkor sávtól függetlenül ugyanannyi ideig tart minden szektor írása/olvasása. Ha a 6.13. ábra a) részét nézzük, láthatjuk is ennek a megoldásnak a hátrányát: ha minden szektorra

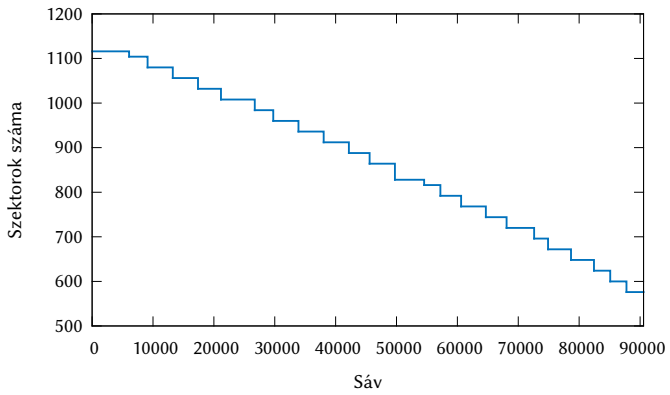
ugyanannyi időt szánunk, akkor a lemez külső felén lévő szektorok sokkal nagyobb helyet foglalnak, mert a kerületi sebesség a lemez peremén sokkal nagyobb, mint a forgástengely közelében. Így az adatsűrűség a lemez belsejében sokkal nagyobb lesz, mint a peremén. A nagyobb tárolási kapacitás érdekében jó lenne a lemez peremén is ugyanolyan sűrűn tárolni az adatokat, mint a belsejében.



a) ZBR nélkül b) ZBR-rel

6.13. ábra. Szektorok a lemezen

Ha az adatsűrűséget rögzítjük (hogy egyforma ”hosszúak” legyenek a szektorok az adathordozón), akkor a perem felé haladva minden sávban egyre több szektor lesz. Ahogy a perem felé haladunk, egyre gyorsabban kell a fejnek leolvasnia az adatokat, hiszen a sáv körsebessége egyre nő, a lemez egyre gyorsabban forog ki a szektorral a hátán a fej alól. Ez önmagában még nem lenne gond, de az már annál inkább, hogy az elektronikának minden egyes sávváltásnál a több százezer sáv adataiból kell kikeresnie, hogy az adott sávban épp milyen gyorsan kell leolvasni az adatokat. Az arany középút a *zóna rendszerű adattárolás* (ZBR, Zoned-Bit Recording, a 6.13. ábra b) része). A lemezen a sávokat zónákhoz rendelik (egy modern merevlemez kb. 50 zónát használ). Egy zónán belül minden sáv azonos számú szektort tartalmaz, tehát állandó a szektorok olvasási/írási ideje, így csak zónaváltáskor kell adatsebességet váltani. Persze van némi pazarlás a tárolási kapacitást illetően, hiszen a zónák belső felében nagyobb az adatsűrűség, mint a peremén, de megfelelően sok zónával ez a kárba veszett kapacitás elfogadható mértékűre szorítható le.



6.14. ábra. A sávonkénti szektorszám a Hitachi Travelstar 5K160 HDD-ben

A Hitachi Travelstar 5K160 (160 GB-os) merevlemez például 24 zónára osztja fel 90575 sávját úgy, hogy a sávonként szektorszám 576 és 1116 között alakul (a 6.14. ábra, [2]).

Mivel a perem felé eső zónák írási/olvasási sebessége nagyobb, a ZBR-rel a merevlemez átlagos adatátviteli sebessége is javul, annál is inkább, mert a perem felé eső – gyors – zónákban több a szektor, mint a lemez belsejébe eső – lassú – zónákban.

Megjegyezzük, hogy a változó körsebesség problémáját az optikai meghajtók másként oldották meg. Az olcsóbb optikai meghajtók (általában azok, amik nem tudnak írni) a másik utat választották: az adatsebességet a teljes lemez területén állandóra rögzítik, és a lemez forgási sebességével játszanak. Ezt a technikát CLV-nek (Constant Linear Velocity) hívják. Az állandó körsebesség eléréséhez a külső peremen 2.5× lassabban forog a lemez, mint a belsejében. Az optikai meghajtók eredeti céljának ismeretében ez érthető választás, hiszen az állandó bitrátával kódolt zene és a mozgókép lejátszása állandó adatsebességű adathordozót igényel. A fejlettebb, írásra is képes optikai meghajtók a ZBR-hez hasonló zónás megoldást alkalmaznak, a Z-CLV-t. A zónán belül CLV-vel biztosítják az állandó adatsebességet, de ez az adatsebesség zónáról zónára változik. Például egy 40×-es CD író a legbelső zónájában 24×-os, a második zónában 32×-es, és csak a külső zónában éri el a 40×-es sebességet.

Szervó vezérlés

A merevlemez egy mozgó alkatrészeket tartalmazó berendezés, a lemez is, a fej is mozog. A nagy adatsűrűség miatt ráadásul a fejet nagyon pontosan kell pozicionálni. Egyrészt ennek a precíziós fejmozgatásnak is megvannak a maga határai (tetszőleges pozicionálási pontosságot nem lehet elérni), másrészt külső hatások miatt is felléphetnek apró elmozdulások, vibrációk, stb. ami mind bizonytalanság forrását képezi. Mindezek következtében működés közben nem is nagyon ritkán fordul elő, hogy a fej leszalad a sávról, vagy seek közben félrepozicionál, esetleg teljesen elveszti a fonalat és vaktában repül.

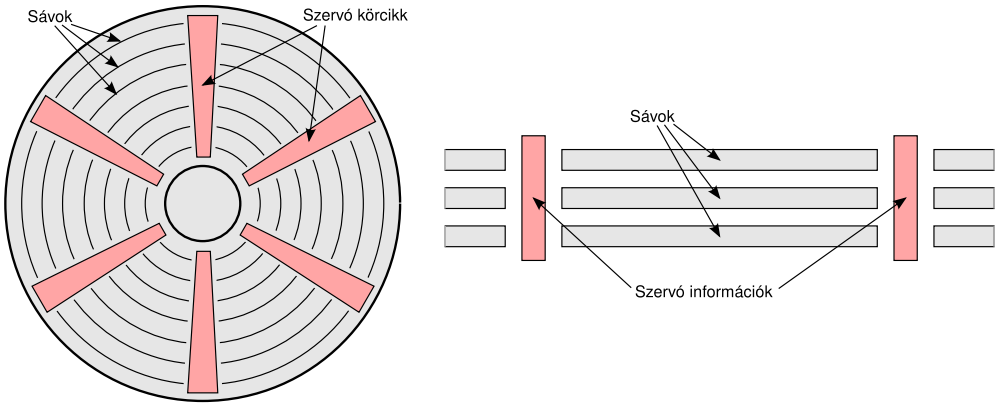
Azt a szabályozási mechanizmust, ami a fej pontos pozicionálásáért felelős, *szervó vezérlésnek* hívjuk. Feladatai:

- Sáv váltáskor (seek) biztosítja, hogy a fej pontosan a kívánt sávot találja meg.
- Segít a sáv követésében: folyamatos apró korrigáló mozdulatokkal biztosítja, hogy a fej mindvégig a sávon haladjon, ne fusson le róla.

Régebben a szervó vezérlést szolgáló adatokat és speciális mágneses mintákat egy saját, dedikált adathordozó felületen helyezték el, ami felhasználói adatokat nem is tartalmazott. Ma már olyan kevés lemez van egy merevlemezben (1, 2, esetleg 3), hogy ez nagy pazarlás lenne. Ezért ezeket a kritikus fontosságú szervó adatokat a felhasználói adatokkal együtt tárolják. Erre a célra minden adathordozó felületen egyenletes távolságonként körcikkeket jelölnek ki (servo wedge, 6.15. ábra), melyeken kétféle információt helyeznek el (6.16):

- sávazonosítót, melyet elolvasva a fej rögtön tisztába kerül a pontos pozíciójával;
- sávtartást segítő sakktábla szerű mintázatot.

Ezeket az információkat a gyártás során írják a merevlemezre, és később nem is módosulnak, integritásuk (sérülésmentességük) kritikus fontosságú a merevlemez számára.

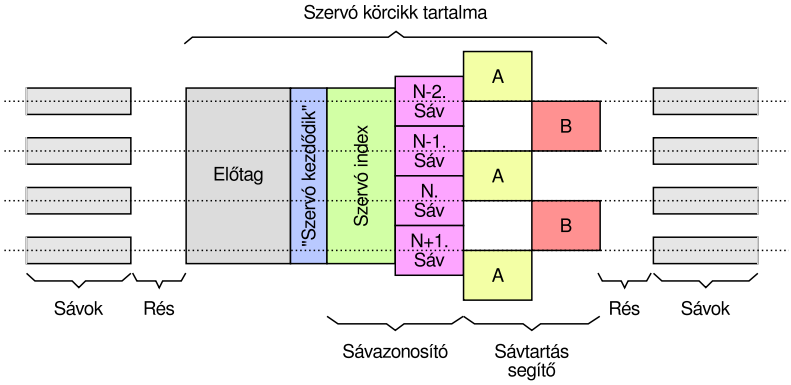


6.15. ábra. A szervó információk helye a lemezen

A sávazonosító két részből áll: az első része a körcikk egészén végighúzódik *szervó index*, megadja, hogy a lemezen ez éppen hányas számú szervó terület, a második rész pedig a *sáv számát* tartalmazza. A fej az előbbi segítségével a szög (ϕ), az utóbbival a radiális (r) pozícióját tudja beazonosítani, és szükség esetén ennek megfelelően tud korrigálni.

A sávtartást kétféle mágneses mintázat (”A” és ”B”) sakktáblaszerű elhelyezésével segítik. A két mintázat közötti határ pont a sávok középvonalára esik. Optimális esetben, ha a fej épp a sáv középvonalán halad, akkor egy szervó terület keresztezésekor egyforma erősen érzékeli az ”A” és ”B” mintázatot. Ha az egyiket erősebben érzékeli, akkor a fej a sáv egyik feléhez közelebb került, ezért pozíciója korrekcióra szorul. Annak ismeretében, hogy a fej páros vagy páratlan sávon halad-e, a szükséges korrekció iránya meghatározható abból, melyik mintázatot érzékelte erősebben.

A szervó területen található információk képét a 6.16. ábra szemlélteti. Ismét szükség van résre, hogy egy túl későn kikapcsolt írófej ne tudja a szervó információt károsítani. Az előtag és a ”szervó kezdődik” mezők szerepe megegyezik a szektorok felépítésénél látott előtag és ”adat kezdődik” mezőkével.



6.16. ábra. A szervó információk felépítése

6.1.5. A merevlemezek teljesítményjellemzői

A merevlemezek teljesítményét az alábbi mennyiségekkel szokás jellemezni:

- *Válaszidő* (Response Time): a parancs merevlemezbe érkezésétől az adatátvitel befejeződéséig eltelt idő
- *Kiszolgálási idő* (Service Time): a parancs feldolgozásának kezdetétől az adatátvitel befejeződéséig eltelt idő
- *Átviteli sebesség* (Throughput): átviteli sebesség alatt kétféle dolgot érthetünk. A kiszolgált adatátviteli kérések számát másodpercenként (I/O operations per second, IOPS), valamint az átvitt adatok mennyiségét másodpercenként (pl. MB/s-ban mérve). A két mennyiség nem független egymástól: átvitt adatok mennyisége másodpercenként = IOPS × kérésenkénti adatmennyiség.

Egyes illesztőfelületek, mint például az ATA és a SATA-1 (6.3. fejezet) előírják, hogy a következő kérést csak akkor lehet elküldeni a merevlemeznek, ha az már befejezte az előző feldolgozását. Ebben az esetben válaszidő megegyezik a kiszolgálási idővel.

Más illesztőfelületek, mint a SATA-2 vagy a SCSI megengedik, hogy a merevlemez egy várakozási sorban raktározza a hozzá érkezett, de még fel nem dolgozott adatátviteli kéréseket (a SATA 32, a SCSI 256 igény várakoztatását engedi meg). Ebben az esetben a válaszidő a sorban állás várakozási idejének és a kiszolgálási időnek az összege lesz.

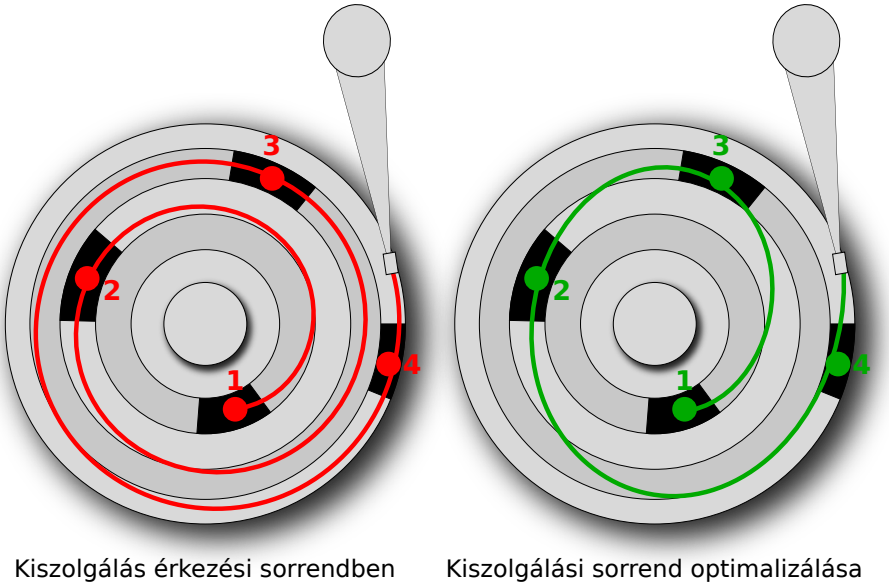
A várakozási sor kezelése

Ha több adatátviteli kérés is várakozik a sorban, a merevlemez maga dönt a kiszolgálás sorrendje felől, lehetőleg úgy, hogy összességében a lehető legkisebb fejmozgással és forgással végre tudja hajtani mindet.

A várakozási sor alkalmazása és a kiszolgálás sorrendjének optimalizálása a merevlemez teljesítményét nagy mértékben javítani tudja. A 6.17. ábrán látható példában az optimalizált sorrend szerinti kiszolgálás fele annyi lemezfordulat alatt kiszolgálta a 4 adatátviteli kérést az optimalizálás nélküli esethez képest.

Néhány gyakran használt stratégia a várakozási sorból a következő parancs kiválasztására (tehát a parancsütemezésre):

- *Kizárólag a seek időre optimalizáló stratégiák.* Ebbe a körbe tartozik az SSTF (Shortest Seek Time First) és a LOOK (Elevator Seek). Az SSTF mindig azt az igényt választja ki a sorból, amelyik a fej aktuális pozíciójához



6.17. ábra. Hozzáférési idők a kiszolgálási sorrend optimalizálásával és anélkül

legközelebbi cilinderre vonatkozik. Sajnos az SSTF ütemező nem fair, mert a lemez közepére vonatkozó kéréseket favorizálja. Az SSTF-nél igazságosabb a LOOK ütemezés, melynek lényege, hogy a fej elindul a lemez belseje felől a külső perem felé, és az útba eső kéréseket sorban kiszolgálja. Ha elérte a külső peremet, visszafordul, és a lemez belseje felé haladva megint kiszolgál minden útjába eső igényt.

- *Teljes hozzáférési időre optimalizáló stratégiák.* Az előbb látott ütemezők nem számolnak a forgási késleltetéssel. Előfordulhat, hogy időben jobban megéri kicsit messzebb eső cilinderre váltani, ha a lemez hamarabb odaforog a keresett szektorral. Az SATF (Shortest Access Time First) ütemező azt az igényt választja ki a sorból, amelyik a leghamarabb érhető el a merevlemez számára, mind a seek, mind a forgási késleltetés figyelembe vételével. Az SATF lényegesen hatékonyabb, mint az SSTF vagy a LOOK, különösen, ha hosszúak a várakozási sorok, bőven van igény, amiből lehet válogatni. Az SATF legnagyobb gondja, hogy pontos *seek profile*-t igényel, azaz pontos időadatokra van szükség, hogy az ”a” cilinderből milyen sokáig tart seek-elni a ”b” cilinderre (amihez a cilinderek több tízezres száma mellett óriási táblázat, vagy valami jó közelítés kell).

A 6.17. ábra jól mutatja, mennyivel jobb a teljes hozzáférési időre optimalizáló stratégia (jobb oldal), mint a pusztán seek időre optimalizáló (bal oldal).

A kiszolgálási idő összetevői

Miután a merevlemez parancsütemezője kiválasztott egy parancsot a várakozási sorából, végre kell hajtania, ki kell szolgálnia azt. Egy adatátviteli igény kiszolgálása több összetevőből áll. Egy olvasási igény összetevői a következők (6.18. ábra):

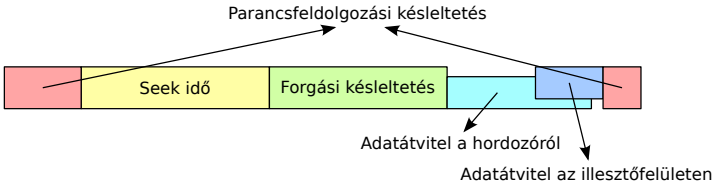
- Parancsfeldolgozási késleltetés (Command Overhead). A parancsfeldolgozási késleltetés az az idő, amit a merevlemez még/már nem a parancs tényleges végrehajtására fordít, hanem annak elő-, ill. utófeldolgozására. A merevlemezben a parancsok végrehajtásának koordinálását egy mikroprocesszor végzi. Az igény kiszolgálásának kezdetekor ez a mikroprocesszor értelmezi a parancsot, valamint az igény kiszolgálásának lezárásaként a mikroprocesszor jelzi a vezérlő felé a művelet végét, tehát parancsfeldolgozási késleltetés a végrehajtás elején és végén is felmerül.
- Seek idő. A seek idő az az idő, amíg a fej az aktuális tartózkodási helyéről elindulva eléri a parancs által kijelölt cilindert. Elsősorban a lemez átmérőjétől, és a fejszerkezet súlyától függ.
- Forgási késleltetés. Ha a fej elérte a kívánt cilindert, meg kell várni, hogy a kijelölt szektor beforogjon a fej alá. A forgási késleltetés fordítottan arányos a lemez forgási sebességével.

Parancsfeldolgozási késleltetés:	1 ms
Átlagos seekidő olvasáskor:	11 ms
Átlagos seekidő íráskor:	13 ms
Forgási sebesség:	5400 RPM (fordulat/perc)
Szektorok mérete:	512 bájt
Maximális adatátviteli sebesség:	100 MB/s

6.1. táblázat. A Hitachi Travelstar 5K160 160 GB-os merevlemez paraméterei

- Leolvasás a hordozóról. Miután a fej a lemez megfelelő pozíciójába kerül, indulhat az adatok leolvasása a mágneses hordozóról. Ennek a sebességét az határozza meg, hogy milyen gyorsan forog a lemez, és hogy milyen sűrűn helyezkednek el az adott sávban a szektorok.
- Adatátviteli idő. A leolvasott adatok továbbítása a meghajtó illesztőfelületén.

Ha elég gyors az illesztőfelület, akkor a leolvasási és az adatátviteli időket át lehet lapolni: egy leolvasott szektor már átvihető az illesztőfelületen, miközben a fej a következő szektort olvassa.



6.18. ábra. Egy olvasási igény kiszolgálási idejének összetevői

A nagyságrendek érzékeltetése kedvéért vegyük egy gyakorlati példát. A Hitachi Travelstar 5K160 (160 GB, SATA) merevlemez gyári adatlapján ([2]) a 6.1. táblázatban összefoglalt adatokat találjuk[1].

Először számoljuk ki egy egyetlen szektor olvasására vonatkozó adatátviteli kérés kiszolgálási idejét, feltéve, hogy a szektor a 8000-es sávban van! Az adatlap szerint ebben a sávban 1104 szektor található.

- Parancsfeldolgozási késleltetés: ez adott, 1 ms.
- Seek idő: szintén adott, 11 ms.
- Forgási késleltetés. Először kiszámoljuk egy teljes körülfordulás idejét a forgási sebesség ismeretében:

$$T_{\text{fordulat}} = \frac{60 \text{ sec/perc} \cdot 1000 \text{ ms/sec}}{5400 \text{ fordulat/perc}} = 11.111 \text{ ms/fordulat}.$$

  Ha az adatátviteli kérésünk egy véletlen időpontban érkezik, a lemez bárhol lehet, a forgási késleltetés 0 és 11.11111 ms között egyenletes eloszlású, várható értéke 5.55555 ms.
- Adat leolvasása a hordozóról: Miután a fej a megfelelő szektorra pozicionált, a leolvasás $11.11111/1104 = 0.01006$ ms ideig tart, hiszen ennyi idő alatt forog a lemez egy szektornyit.
- Adat átvitele az illesztőfelületen: Ki kell számolnunk, hogy a 100 MB/s sebességű interfészen mennyi ideig tart 512 bájt átvitele. Ebből $512/(100 \cdot 1024 \cdot 1024) = 0.00488$ ms adódik.

A kérés teljes kiszolgálási ideje: $1 + 11 + 5.55555 + 0.01006 + 0.00488 = 17.57$ ms, melyben látványosan dominál a két mechanikai jellegű késleltetés, a seek és a forgási idő. Ez a fejezet egyik súlyponti mondanivalója.

Most nézzük meg, hogy alakul a kiszolgálási idő, ha nem csak 1, hanem 100 egymás utáni szektort szeretnénk leolvasni. A parancsfeldolgozási idő, a seek és a forgási késleltetés nyilván ugyanannyi marad, hiszen ezekre a műveletekre parancsonként csak egyszer van szükség.

[1] Azért egy ilyen "elavult" típust vettünk példaként, mert a gyári adatlapja mintaszerű: minden fontos, és kevésbé fontos, de érdekes részletet tartalmaz, ami nem mondható el a modernebb merevlemezek adatlapjairól.

- Adat leolvasása a hordozóról: most 100 szektort kell leolvasni 1 helyett, tehát a leolvasási idő az előbbi 100-szorosa, 1.006 ms.
- Adat átvitele az illesztőfelületen: az adatátvitel már az adatok leolvasása közben megkezdődhet, és akkor fejeződik be, amikor az utolsó szektor leolvasása után az is továbbításra kerül az illesztőfelületen (lásd az átlapolást a 6.18. ábrán). Így az átviteli idő nem átlapolható része továbbra is egyenlő az 1 szektorra számolt átviteli idővel, vagyis 0.00488 ms.

A 100 szektorra vonatkozó kérés teljes kiszolgálási ideje: $1 + 11 + 5.55555 + 1.006 + 0.00488 = 18.57$ ms, azaz 100-szoros adatmennyiség leolvasásához mindössze 5.7%-kal több időre volt szükség! Ez a fejezet másik súlyponti mondanivalója.

Most már nyilvánvaló, hogy sebesség szempontjából a lehető legrosszabb, amit egy merevlemezzel meg lehet tenni, az a kis méretű fájlok véletlen elérése. Ahogy egyre nagyobb szekvenciális szektorsorozatra vonatkozik a művelet, úgy amortizálódik le a seek és forgási idő a teljes kiszolgálási időhöz képest.

Átviteli sebesség

Ha az átviteli sebességet a diszken véletlenszerűen elhelyezkedő, kis mennyiségű adatok segítségével mérik, akkor *véletlen átviteli sebességről* (random throughput), ha nagy, folytonosan elhelyezkedő adatmennyiséggel mérik, akkor *folytonos átviteli sebességről* (sequential throughput) beszélünk. Mindkét teljesítményjellemző fontos, hiszen míg a véletlen átviteli sebességben jelen van, sőt, nagyon is dominál a seek idő és a forgási késleltetés, addig a folytonos átviteli sebességben ezek az összetevők minimális szerepet játszanak.

Most számítsuk ki a fenti példán a 4 kB-os blokkméretre vonatkozó adatátviteli sebességet! A 4 kB kis blokkméretnek számít, így ez véletlen átviteli sebesség lesz. Egy 4 kB-os blokk 4096/512 = 8 szektorból áll, melynek kiszolgálási ideje a fenti részeredmények felhasználásával $1 + 11 + 5.55555 + 8 \cdot 0.01006 + 0.00488 = 17.641 ms$. Nézzük meg, hány 4 kB-os átvitel fér egy másodpercbe: 1000/17.641 = 56.67, tehát a merevlemez véletlen adatátviteli sebessége 56.67 IOPS, adatmennyiséggel együtt kifejezve pedig $56.67 \cdot 4096 = 232120$, azaz 0.22 MB/s. Akármilyen alacsonynak tűnik, ez bizony nem egy kirívóan lassú merevlemez.

Ha a folytonos adatátviteli sebességre vagyunk kíváncsiak, nagy mennyiségű, pl 16 MB méretű, folytonosan elhelyezkedő adat átviteli idejéből kell kiindulnunk. A 16 MB összesen 32768 szektorból áll, így a kiszolgálási idő $1 + 11 + 5.55555 + 32768 \cdot 0.01006 + 0.00488 = 347.2 ms$. Ilyen kiszolgálási időkből 1000/347.2 = 2.88 fér egy másodpercbe, tehát a folytonos átviteli sebesség 2.88 IOPS, az adatmennyiséget is figyelembe véve pedig $2.88 \cdot 16 = 46$ MB/s.

6.2. Félvezető alapú háttértárak (SSD)

Az SSD (Solid-State Drive) háttértárakban az adatokat félvezető alapú memória tárolja. Mivel a HDD-kkel szemben az SSD-k nem tartalmaznak mozgó alkatrészt, a mechanikai behatásokra, ütésekre, vibrációra ellenállóbak, és működésük nem jár zajjal.

Különféle memória technológiákon alapuló SSD-ket már az 1970-es évek óta használnak adatok tárolására. Mi a továbbiakban SSD alatt a ”modern”, flash memória alapú SSD-ket értjük. A flash memória jellegzetessége, hogy az adatokat tápellátás nélkül is megőrzi. Az első flash alapú SSD-t az M-Systems fejlesztette ki 1995-ben. Ezek a kezdeti modellek nem voltak meggyőzően gyorsabban a HDD-knél, az előállításuk is nagyon költséges volt, de a jó ütésállóság és az extrém hőmérsékletek tolerálása miatt hadászati célokra ideálissá váltak.

Ahogy teltek az évek, rengeteg pénzt és energiát fektettek a félvezető alapú adattárolás fejlesztésébe. Kezdetben drága, prémium kategóriás terméknek számítottak, majd 2004-ben megjelent az első SATA illesztőfelülettel ellátott SSD. Ezt az SSD-t az Adtron gyártotta, kapacitása 16 GB, ára pedig 11.000$ volt. Ettől kezdve az ár zuhanórepülésbe, a kapacitás pedig meredek emelkedésbe kezdett. 2007-ben következett be a robbanás, az árak és a tárolási kapacitás ekkor érte el széleskörű elterjedéshez szükséges kritikus szintet. A 2011-es, nagy merevlemezgyártókat sújtó thaiföldi árvíz pedig megadta a végső lökést az SSD szektornak: ma már a HDD-vel összemérhető áron (igaz, kisebb kapacitás mellett) be lehet szerezni a HDD-knél időközben lényegesen gyorsabbá vált SSD-ket.

A korábbiakban láttuk, hogy a HDD-nek vannak hátrányos jellemzői, most majd látni fogjuk, hogy bizony az SSD-nek is vannak. Nem lehet általánosságban kijelenteni, hogy az SSD ”jobb”, mint a HDD, még csak az sem igaz, hogy mindig gyorsabb (van az adatátviteli kéréseknek olyan sorozata, amit a HDD gyorsabban szolgál ki). Ezért is fontos megérteni mindkét technológia előnyeit és hátrányait, hogy a konkrét felhasználás ismeretében, az előnyök és hátrányok mérlegelésével a lehető legjobb döntést tudjuk hozni.

	Vannak elektronok	Nincsenek elektronok
”Normál” gate fesz.	Zár	Zár
”Kisebb” gate fesz.	Nyit	Zár
Gate fesz. = 0V	Nyit	Nyit

6.2. táblázat. A lebegő gate-es tranzisztor működése

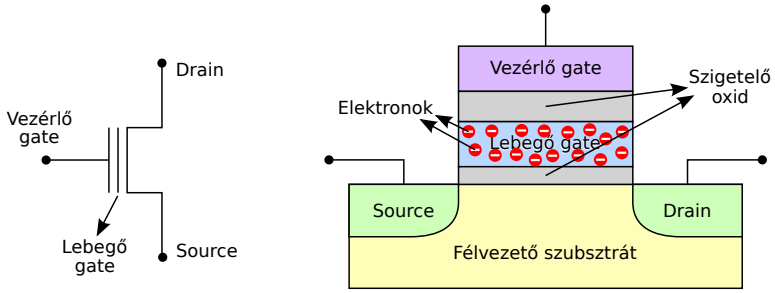
6.2.1. Az adattárolás elve a flash memóriában

A ma kapható SSD tárolók túlnyomó többségében az elemi adategységeket, a biteket úgynevezett lebegő gate-es tranzisztorok (floating gate transistor) tárolják. Nem célunk a lebegő gate-es tranzisztorok minden részletre kiterjedő ismertetése, és nem feltételezzük a hagyományos tranzisztorok ismeretét sem, de egy áttekintő kép mindenképp szükséges az SSD-k sajátosságainak megértéséhez.

Egy hagyományos MOSFET tranzisztornak (Metal–Oxide–Semiconductor Field Effect Transistor) 3 elektródája van: a gate, a source, és a drain. A gate-re adott feszültséggel lehet szabályozni a source és a drain között folyó áram erősségét. Ha kapcsolóként működtetjük a tranzisztort, akkor ez a következőt jelenti:

- Ha nem adunk feszültséget a gate-re, akkor a source és a drain között nem folyik áram, a tranzisztor által reprezentált kapcsoló *nyitott* állásban van.
- Ha a Gate-re megfelelően nagy feszültséget adunk (voltage threshold, V_T, általában 0.5V), akkor a Source és a Drain között áram tud folyni, a tranzisztor által reprezentált kapcsoló *zárt* állásban van.

A lebegő gate-es tranzisztor ehhez képest annyiban tér el, hogy két gate-et tartalmaz (6.19. ábra). A vezérlő gate szerepe ugyanaz, mint a FET-ben, vagyis ezzel lehet szabályozni a source-drain áramot. A lebegő gate (ami nincs kivezetve) viszont olyan, mint egy ketrec, amibe elektronokat lehet zárni. A tranzisztor másként viselkedik, ha a lebegő gate-jében van elektron, mint ha nincs. Az elektronokat különféle úton-módon lehet a lebegő gate-be kényszeríteni, illetve onnan eltávolítani (lásd később), a lényeg, hogy ha a lebegő gate-be elektronokat zártunk, akkor azok mindenféle tápellátás nélkül, hosszú távon (évtizedekig) ott is maradnak.



6.19. ábra. Lebegő gate-es tranzisztorok áramköri jele és felépítése

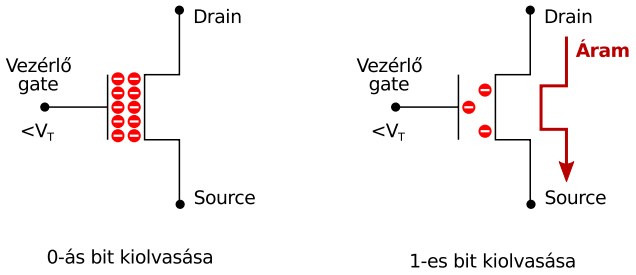
A lebegő gate-es tranzisztorban a lebegő gate-en lévő elektronok jelenléte a tranzisztorral megvalósított kapcsoló záráshoz (a source - drain áram megindulásához) V_T feszültséget befolyásolja, a következő módon (lásd még 6.2. táblázat):

- Ha vannak többlet elektronok a lebegő gate-ben, akkor a lebegő gate-es tranzisztor nagyobb küszöb feszültségű lesz. (A lebegő Gate többlet elektronjai mintegy „árnyékolják” a vezérlő gate hatását.)
- Ha nincsenek többlet elektronok a lebegő gate-ben, akkor a tranzisztort - az előzőhöz képest - sokkal kisebb gate – source feszültséggel is zárni lehet (V_T kisebb lesz).

Ezek ismeretében a lebegő gate-es tranzisztort a következőképpen használhatjuk adattárolásra:

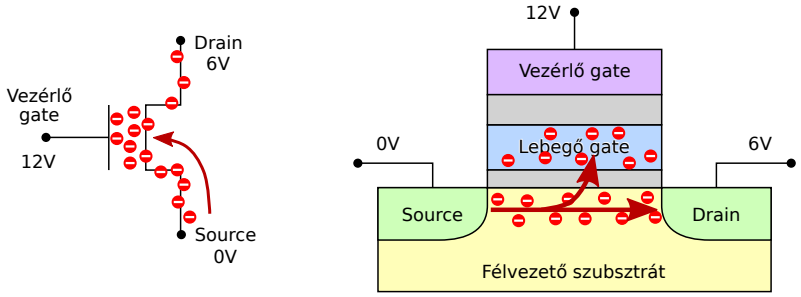
Adatok reprezentálása. Egy tranzisztor egy bitet tárol (SLC esetben – lásd később). Ha a lebegő gate fel van töltve elektronokkal, az 0-ás bitet, ha nincs, az 1-es bitet jelent.

Adatok kiolvasása. A 6.2. táblázat alapján úgy olvashatjuk ki a tárolt bit értékét, hogy kis feszültséget adunk a vezérlő gate-re, és megnézzük, hogy folyik-e áram a source és a drain között. Ha igen, akkor 1-es bitet, ha nem folyik áram, 0-ás bitet tárol a tranzisztor (6.20. ábra).



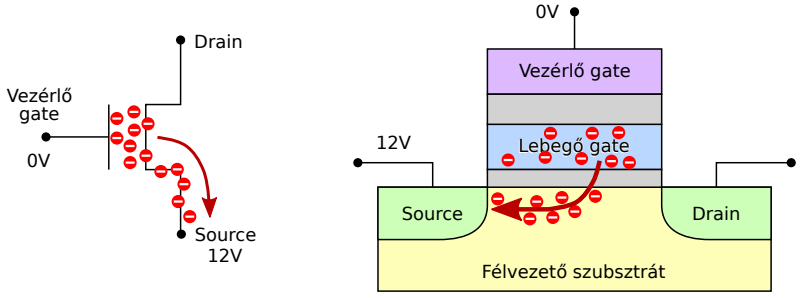
6.20. ábra. A tárolt bit kiolvasása a lebegő gate-es tranzisztorból

Váltás 1-ből 0-ba. Ennek a műveletnek *programozás* a neve. Ehhez a váltáshoz a lebegő gate-et elektronokkal kell feltöltenünk. Ezt úgy érhetjük el, hogy a source-ot leföldeljük, a vezérlő gate-re és a drain-re nagy feszültséget adunk. Ilyenkor a nagy energiájú elektronok átjutnak a lebegő gate és a szubsztrát közötti szigetelő rétegen, és a lebegő gate-ben is maradnak (6.21. ábra, ez a „hot electron insertion method” – a programozásnak más módja is létezik).



6.21. ábra. Programozás: váltás 1-ből 0-ba

Váltás 0-ból 1-be. Ezt a műveletet *törlés*nek hívják. A vezérlő gate-et le kell földelni, a drain-t szabadon lebegve hagyni, és a source elektródára nagy feszültséget kell adni. Ekkor az alagúteffektusnak köszönhetően az elektronok a szigetelőrétegen átlépve elhagyják a lebegő gate-et (6.22. ábra).



6.22. ábra. Törlés: váltás 0-ból 1-be

A programozásnak és a törlésnek azonban van egy igen kellemetlen mellékhatása: a lebegő gate-be kényszerített, illetve az azt elhagyó elektronok néha beleragadnak a szigetelőrétegbe, és ott is maradnak. Ettől a szigetelőrétegnek megváltoznak az elektromos tulajdonságai, a tranzisztor zárásához szükséges V_T feszültség

pedig emiatt eltolódik. Minél többször töröljük a tranzisztort, annál több elektron szorul a szigetelőbe, ami egyre kevésbé fog szigetelni. A lebegő gate-es tranzisztor tehát *öregszik* (kopik). Egyszer aztán eljön az a pont, amikor már nem képes adatot tárolni. Az, hogy ez hány törlési ciklus után következik be, a gyártástechnológiától függ: minél kisebb a tranzisztor, annál hamarabb. Ma, a 20-25 nm-es technológia mellett kb. 100 ezer törlést bírnak ki a tranzisztorok (SLC adattárolás mellett).

6.2.2. SLC és MLC adattárolás

Ahogy láttuk, a lebegő gate-en lévő töltés jelenléte határozza meg, hogy a tranzisztor 0-ás, vagy 1-es bitet tárol-e. Ha vannak töltések, akkor 0-ás, ha nincsenek, 1-es bitet feleltetünk meg neki.

Egy egyszerű ötlettel azonban meg lehet oldani, hogy egy lebegő gate-es tranzisztor több bitet is tárolni tudjon. A módszer lényege, hogy nem csak a ”van töltés” – ”nincs töltés” állapotokat különböztetjük meg, hanem több töltési szintet is használunk. Így a legegyszerűbb esetben négy töltési szinttel 2 bitet is tárolni lehet: ”11” bitek felelnek meg annak az esetnek, amikor nincs töltés a lebegő gate-en, ”10” bitet tárol a tranzisztor, ha kis, ”01”-át, ha több, és ”00”-át, ha sok töltés van rajta. A töltés mennyisége határozza meg a tranzisztor zárásához szükséges feszültséget, minél több a töltés, annál nagyobb feszültség zárja a tranzisztort.

Ha ezt a módszert használva több bitet tárolunk egy tranzisztorral, akkor MLC (Multi-Level Cell), ha egy tranzisztor csak egy bitet tárol, akkor SLC (Single-Level Cell) flash memóriáról beszélünk. Egy n bites MLC flash memória tranzisztorai 2^n féle töltöttségi szint megkülönböztetésével tárolják az adatokat.

Az MLC flash memóriák ugyan sokkal olcsóbbak, de cserébe gyorsabban öregszenek, több, mint egy nagyságrenddel kevesebb programozási/törlési ciklust bírnak ki, mint az SLC társaik. Ennek megfelelően az MLC az árérzékeny piacot, az SLC pedig a nagy megbízhatóságot és nagyobb terhelhetőséget igénylő vállalati célokat szolgálja ki.

Az SSD gyártók az utóbbi években az ”MLC” megjelölést kissé eltérő értelemben kezdik használni. Ha ma MLC SSD-t veszünk, akkor az két bitet tárol tranzisztoronként, míg az újonnan bevezetett TLC (Triple-Level Cell) megnevezéssel illetik azokat az újabb típusokat, melyek három bit tárolására képesek tranzisztoronként.

6.2.3. NOR és NAND flash architektúra

Most, hogy megismertük, hogyan lehet biteket tárolni, írni és olvasni, megnézzük, hogy hogyan lehet nagy mennyiségű adat tárolását a lehető leghatékonyabban megoldani. A szilíciumostya két dimenziós, tehát ezt a nagy mennyiségű tranzisztort két dimenzióban, rácsszerűen kell elrendeznünk.

A két legelterjedtebb megoldás a rácsba rendezett tranzisztorok összekötésére a NOR és a NAND struktúra.

A NOR flash memória

A NOR flash memóriában a tranzisztorok a 6.23. ábra szerint vannak összekötve. Az ugyanazon sorhoz tartozó tranzisztorok vezérlő gate-jeit egy közös vezeték,a szóvezeték (word line, WL) köti össze, a bitvezeték (bit line, BL) pedig az egyazon oszlophoz tartozó tranzisztorok drain elektródáit köti össze. A source elektródák az egész tárolómezőre közös source vezetékre (source line, SL) kapcsolódnak.

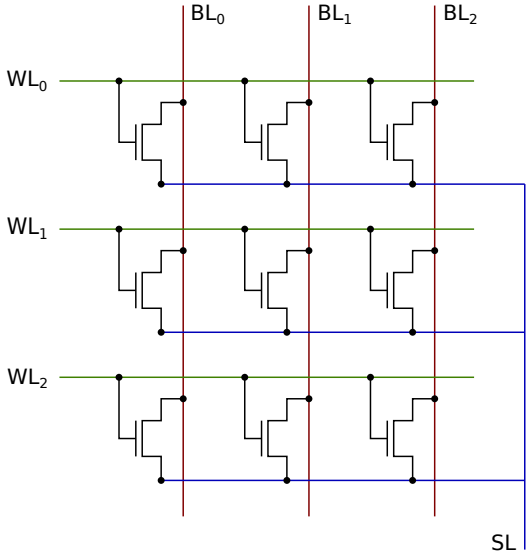
Ha az i. sor j. oszlopában tárolt bitre vagyunk kíváncsiak, akkor a $WL_k,\ k \neq i$ vezetékekre $0V$, míg a WL_i vezetékre kis feszültséget adunk. Ennek hatására az i. sorban lévő tranzisztorok közül a logikai 1-et tárolók zárni, a logikai 0-át tárolók pedig nyitni fognak (a többi sor tranzisztorai mind nyitva vannak). Tehát a tárolt bit értékét úgy tudjuk megállapítani, hogy megnézzük, folyik-e áram a BL_j és az SL között. Az i. sor j. oszlopában tárolt bit 1-es, ha folyik áram, és 0-ás, ha nem.

Annak semmi akadálya, hogy a NOR flash bitjeit egyesével programozzuk (vagyis 1-ből 0-ba váltsuk), de törölni csak az egész tárolómezőt egyben lehet (egyéb fizikai okok mellett azért is, mert az SL közös).

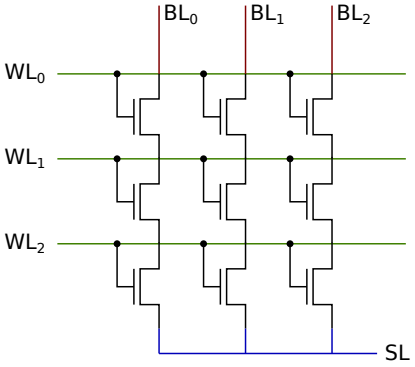
A NAND flash memória

A NAND flash architektúrát a Toshiba vezette be 1989-ben. A NAND flash-ben úgy rendezik el a tranzisztorokat, hogy egy oszlopban az egymás ”alatti” tranzisztorok source és drain elektródái egymásba érnek, össze vannak kötve (6.24). Így jóval kevesebb a tranzisztorok közötti összeköttetés, több hely marad a tényleges adattárolást végző tranzisztoroknak, ezért sokkal nagyobb adatsűrűséget lehet így elérni.

Az i. sor j. oszlopában tárolt bit kiolvasásához *minden* $WL_k,\ k \neq i$ vezetékre ”normál” feszültséget kell adni, a WL_i-re pedig a 0 és az 1 elkülönítését lehetővé tevő ”kisebb” feszültséget. Ennek következtében a $\neq i$ sorok



6.23. ábra. NOR flash architektúra



6.24. ábra. NAND flash architektúra

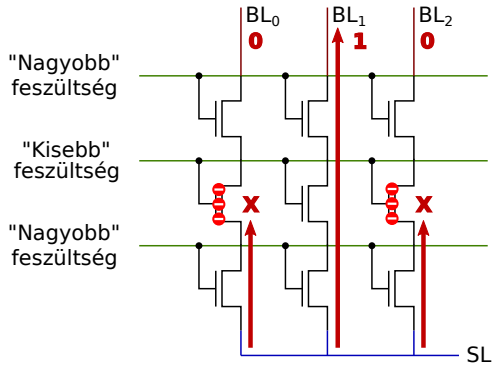
tranzisztorai mind zárnak, ”átengedik” az áramot. Így az *i*. sor *j*. tranzisztorának töltése (vagyis a tárolt bit értéke) dönti el, hogy a BL_j és az *SL* között folyik-e áram, vagy sem (6.25. ábra).

A NAND flash kifejlesztésekor egy háttértárként funkcionáló tárolót kívántak létrehozni, ennek megfelelően a programozási és olvasási műveletek adategységei a HDD-hez hasonlóan nem bitek vagy bájtok, hanem nagyobb méretű *lapok* (page). A lap nem más, mint a tárolómező egy sorába tartozó bitek összessége. Annak ellenére, hogy nincs fizikai akadálya annak, hogy a biteket egyesével programozzuk és olvassuk, a NAND flash minden programozási és olvasási művelete lapokra vonatkozik. Sajnos a NOR flash-hez hasonlóan a NAND flash-ben is csak az egész tárolómezőt egyszerre lehet törölni. Pontosítunk: a tárolómezőt, amit egyszerre lehet törölni, a flash terminológia *blokknak* (block) hívja. A törlés problémáját tehát a NAND flash úgy oldja meg, hogy nem integrálja a teljes tárolókapacitást egyetlen tárolómezőbe, hanem kisebb, egymástól szeparált törlési egységeket – blokkokat alakít ki.

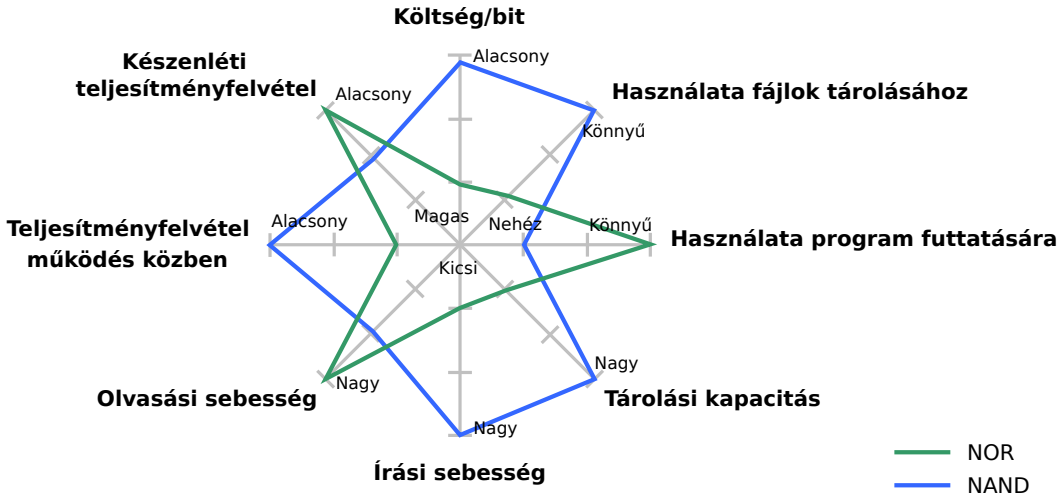
Összehasonlítás

Mint láttuk, a NAND és a NOR flash architektúrák nemcsak abban különböznek, hogy máshogy vannak összekötve a tranzisztorok, hanem a megcélzott felhasználási terület is eltér, ezért az összehasonlítás nem lesz teljesen igazságos (6.26. ábra, [5]).

A NOR flash-t gyorsabban lehet olvasni, mert az elemi adattároló egységek párhuzamosan vannak bekötve, míg a NAND-ban az olvasás lassabb, mert a jelnek az oszlop sok-sok tranzisztorán mind át kell haladnia a kiolvasásig. A NAND flash-ben gyorsabb a programozás, többek között azért, mert egy egész lap egyszerre,



6.25. ábra. NAND flash architektúra



6.26. ábra. A NOR és a NAND flash összehasonlítása

egyetlen művelettel beírható, míg a NOR egyenként, tranzisztorról tranzisztorra végzi a programozást. A NAND a törlésben is gyorsabb, mivel más fizikai effektust használ (ennek részleteibe nem mentünk bele).

Tárolási kapacitást tekintve a NAND flash architektúrával közel kétszer nagyobb tranzisztor sűrűséget lehet elérni.

Összességében, mindkét megoldásnak megvan a létjogosultsága. A NOR flash nagy előnye, hogy nem laponként, hanem bájtonként közvetlenül címezhető, így kitűnően alkalmas például beágyazott rendszerekben (PDA, hordozható zenelejátszó, telefon, fénymásoló, PC BIOS, stb.) a ritkán változó programkód tárolására. A NAND flash háttértárként funkcionál, a lap alapú szervezés az adatok szekvenciális elérésének kedvez. NAND flash tárolóra épülnek a memóriakártyák, SSD-k, pendrive-ok. Újabban a NAND flash kezd teret nyerni a NOR-ral szemben a beágyazott rendszerek háttértáraiban is.

Skálázhatósági problémák

A gyártástechnológia töretlen fejlődése lehetővé teszi, hogy egységnyi területen egyre több tranzisztort lehessen elhelyezni, aminek köszönhetően a flash memória alapú adattárolók tárolási kapacitása folyamatosan nő. Ez a növekedés azonban nem tart örökké, sőt, a fizikai korlátokat már napjainkban is feszegetik.

Az egyik nehézség, amivel szembe kell nézni az adatsűrűség növelésekor, hogy a túl közel helyezett tranzisztorok között áthallás alakul ki. A közelség miatt a tranzisztorok nem tudnak teljesen függetlenül működni, az egyiken végzett műveletek, ha kis mértékben is, de hatással lesznek a szomszédos tranzisztorokra is.

A másik, sokkal drasztikusabb korlátozó tényező, hogy a tranzisztorok méretének csökkenésével azok élettartama is csökken. Kisebb méretben a törlés/programozás során a szigetelőrétegbe beszoruló elektron hatása is kifejezettebb. Az SSD meghajtók hozzávetőleges élettartama a 6.3. táblázatban látható ([41]). Nem csak a méret

	≈50 nm	≈34 nm	≈25 nm
SLC	100.000	100.000	100.000
MLC	10.000	5.000	3.000
TLC	2.500	1.250	750

6.3. táblázat. Az SSD meghajtók által elviselt törlési/programozási ciklusok száma a gyártástechnológia és az egy tranzisztor által tárolt bitek függvényében

Hierarchia szint	Tipikus méret
1 lebegő gate-es tranzisztor:	1-3 bitet tárol
1 lap:	512 byte – 8 kB
1 blokk:	128 – 256 lap
1 tárolósík:	1024 blokk
1 szilíciumlapka:	1 – 4 tárolósík
1 tok:	1 – 4 szilíciumlapka

6.4. táblázat. Tipikus méretek a NAND flash tároló hierarchiájának egyes szintjein (2012-es adatok)

csökkenése ront az élettartamon, hanem az is, hogy hány bitet tárol egy tranzisztor. A táblázat alapján láthatjuk, hogy egy TLC meghajtón nagyon változékony adatot nem érdemes tárolni. A kritikus, vállalati környezetben is használt SLC meghajtók élettartamát csak egyre erőteljesebb és nagyobb hibajavító kódok használatával tudják fenntartani (ami persze a hasznos adatok elől viszi el a helyet).

A fejlődés iránya: 3D-NAND töltésfogó flash tranzisztorokkal

A skálázhatóságot megnehezítő két probléma mindegyikére van megoldás, melyek éppen napjainkban vannak a bolti megjelenés küszöbén.

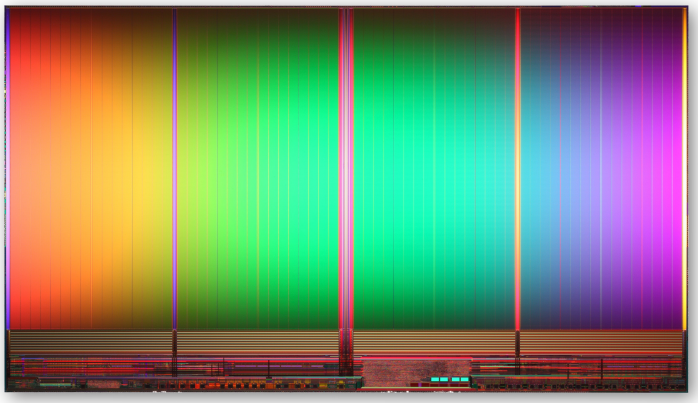
A tranzisztorsűrűség növelésére a 3D-NAND technológia, ill. a Samsung elnevezésében a V-NAND újítása, hogy a tranzisztorokat nem csak egy kétdimenziós rácsba rendezik, hanem egymás tetejére is rétegezik.

Az élettartam növelésének érdekében pedig a lebegő gate-es tranzisztorokat töltésfogó tranzisztorokkal (CTF, charge trap flash) váltják le. A lebegő gate-es tranzisztorokban a töltést a vezető tulajdonságú lebegő gate tárolja. Ezt úgy is felfoghatjuk, mintha az elektronokkal teli lebegő gate-ben egy vödör víz lenne, mely a sok törlési/programozási ciklus miatt öregedő szigetelő réteg egyre szaporodó lyukain el tud szivárogni, így a tárolt információ elvész. A töltésfogó tranzisztorban azonban az elektronokat egy szigetelő rétegben kialakított lyukak tárolják (gondojunk egy vizet tároló szivacsra), ami a töltéseket sokkal jobban a helyükön tartja.

Az első új generációs SSD meghajtót a Samsung dobta piacra 2014-ben (Samsung SSD 850 Pro), mely 32 réteget tartalmaz, és már CTF tranzisztorokra épül. A Samsung állítása szerint élettartama 10-szerese a hagyományos MLC-s társaiknál.

6.2.4. Az adatok szervezése

A NAND flash-ben a tranzisztorok elhelyezésének hierarchiája nem áll meg a lapok és a blokkok szintjénél. Kezdjük az elejétől: a biteket a lebegő gate-es tranzisztorok tárolják (MLC esetben egy tranzisztor több bitet is). Egy sornyi tranzisztor alkot egy lapot. A lap a programozás és az olvasás alapegysége. Több lap alkot egy blokkot. A blokk a törlés alapegysége. Több blokk alkot egy tárolósíkot (plane). Egy szilíciumlapkán több tárolósíkot is el szoktak helyezni. Végül, egy tokba gyakran több szilíciumlapkát is tesznek. A tipikus méreteket a 6.4. táblázat foglalja össze.



6.27. ábra. Az Intel 8GB-os NAND flash lapkája (2 bites MLC, 8kB-os lapok, 256 lap/blokk, 1024 blokk/tárolósík, 4 tárolósík/lapka, www.intel.com/pressroom)

6.2.5. Szektorszintű műveletek az SSD-n

Az SSD-hez befutó olvasási műveletek logikai címekkel, LBA-val hivatkoznak a szektorokra, melyeket az SSD vezérlője fizikai címekre képez le. A leképzés során kikeresi a nyilvántartásában, hogy az adott szektor melyik blokk hányas lapján van, kiolvassa a megfelelő lapot, és a megtalált szektor visszaadásával fejezi be a kérés végrehajtását.

Az írási kérésekkel korántsem ilyen egyszerű a helyzet. Ha a felhasználó egy fájlon dolgozik, és menet közben többször is elmenti a megváltozott tartalmat, azt a HDD nagyon egyszerűen érvényesíti: a régi tartalmat (mágneses jeleket) felülírja az új tartalommal. *Erre az egyszerű műveletre az SSD azonban nem képes.*

A technológiai bevezetőben már láttuk, hogy az SSD csak törölt lapokra tud írni. Az SSD tehát nem ismeri a felülírás fogalmát. Ha megváltozik egy lap tartalma, akkor az SSD keres egy még érintetlen (törölt) lapot, és odaírja a megváltozott tartalmat, miközben a régi lapot érvénytelennek jelöli. A régi lapra pedig, hiába került használaton kívül, nem lehet írni egészen addig, amíg a lapot tartalmazó teljes blokk (ami akár 2 MB is lehet!) törlésre nem kerül.

Az SSD vezérlő tehát minden laphoz egy állapotot rendel, mely lehet

- ”Használt”: a lap érvényes adatokat tartalmaz;
- ”Érvénytelen”: lap tartalma megváltozott, máshová került, de attól még ilyen lapra írni nem lehet, mivel nincs törölve;
- ”Törölt”: a lapra a blokkjának törlése óta nem írtak, azaz ez egy írásra kész lap.

Lapírás menedzsment

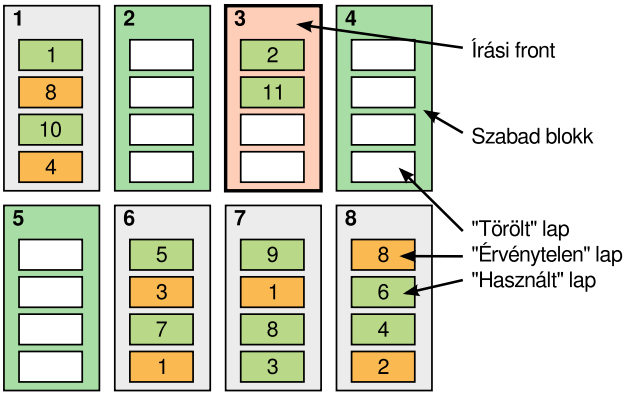
A gyártók ugyan nem publikálják, hogy az SSD vezérlőjük pontosan hogyan is kezelik az írási műveleteket, de a tudományos szakirodalomban megjelent cikkek és az SSD szimulátorok mind-mind ugyanazt a (vélhetően a gyakorlatban is alkalmazott) eljárást követik, így ezzel érdemes megismerkedni.

Ebben az eljárásban a vezérlő az SSD szabad blokkjai közül mindig kiválaszt egyet speciális célra, ez az *írási front* (write frontier, [17]), mely a kiválasztás pillanatban csupa ”törölt”, azaz írható lapot tartalmaz. Az írási műveletek a következőképpen zajlanak:

- Minden írási művelet az írási frontra vonatkozik. Ha írási kérés érkezik, az SSD mindig az írási front soron következő lapjára helyezi az adatot. Ha felülírásról van szó, akkor ezzel egy időben érvényteleníti a régi tartalmat tároló lapot (mely valószínűleg egy másik blokkon található).
- Ha az írási front minden lapja megtelik, az SSD vezérlő a törölt blokkok listájáról kiválaszt egyet, és az lesz az új írási front.

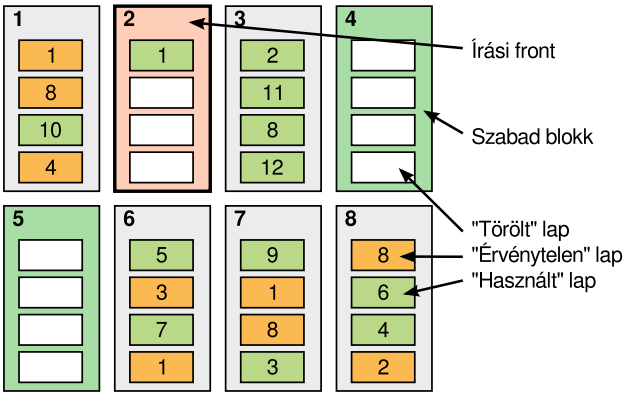
A 6.28. ábrán látható példán az SSD 8 blokkból áll, melyből pillanatnyilag 3 törölt (a 2-es, 4-es és 5-ös), és az aktuális írási front a 3-mas blokk. Minden blokk 4 lapból áll, a zöld a használt, a narancs az érvénytelen, a fehér a törölt lapokat jelöli. Az adatokat tartalmazó lapokon a tárolt LBA címek szerepelnek (most nagyvonalúan

tekintsünk el attól a nehézségtől, hogy a szektorméret és a lapméret nem egyenlő). Vannak LBA címek, melyek többször szerepelnek, ami módosításra utal. Pl. a 8-as LBA cím eredetileg az 1-es blokkon volt eltárolva, de a tartalmának módosítása után ezt az SSD érvénytelenítette, és az új tartalmat a 8-as blokkra helyezte. Egy újabb módosítás után a 8-as LBA cím tartalma a 7-es blokkra került, és a zöld színből látható, hogy ez az aktuális változat.



6.28. ábra. Példa az írási műveletek végrehajtására

Az ábrát kiindulási pontnak tekintve tegyük fel, hogy az SSD-hez írási kérés érkezik a 8-as, a 12-es és az 1-es LBA címekre. A 8-as esetében érvényteleníteni kell a régi változatot (7-es blokk 3-mas lapja), az új változat pedig az írási front soron következő, vagyis a 3. lapjára kerül. A 12-es LBA cím eddig nem szerepelt az SSD-n, ezért ezt egyszerűen csak az írási front végére lehet írni. Ekkor érkezne az 1-es címre az írási kérés, csakhogy az írási front megtelt. Az SSD új írási frontot választhat, a 2-es, a 4-es és az 5-ös blokkok jöhetnek szóba, hiszen ezek szabadok (töröltek). Most válasszuk a 2-est írási frontnak, erre kerül az 1-es LBA cím tárolásra szánt tartalma, miközben a korábbi, felülírásra szánt tartalmat érvényteleníteni kell (1-es blokk 1-es lapja). A végállapotot a 6.29. ábrán láthatjuk.



6.29. ábra. A a 6.28. ábra példája három új írási művelet után

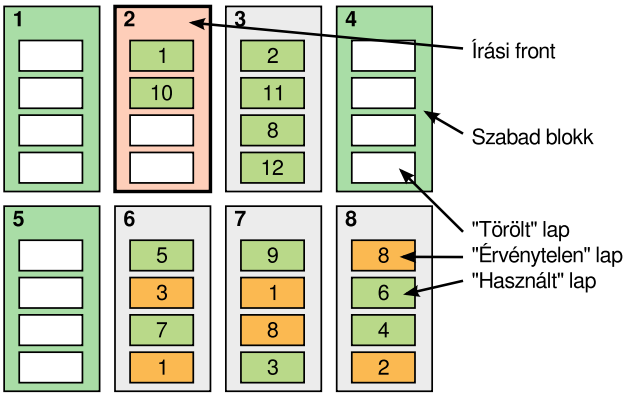
Ahogy jönnek az írási kérések sorra egymás után, úgy fogynak a szabad, törölt blokkok az SSD-n. Vegyük észre, hogy a törölt blokkok nem csak akkor fogyhatnak el, ha megtelik a meghajtó. Az SSD tele lehet érvénytelenített lapokkal, melyek nem hordoznak hasznos információt, viszont írni sem lehet rájuk. Ha a törölt blokkok listája fogyni kezd, működésbe lép a *szemétgyűjtő algoritmus* (garbage collection), amelynek az a feladata, hogy egy kis átrendezéssel új törölt blokkokat hozzon létre.

A szemétgyűjtő algoritmus működése:

- Kinéz egy blokkot a meghajtón (pl. azt, amelyiken a legtöbb ”érvénytelen” lap van),
- ennek a blokknak a ”használt” állapotú lapjait az írási frontra írja,
- majd törli az egész blokkot, és hozzáfűzi a törölt blokkok listájához.

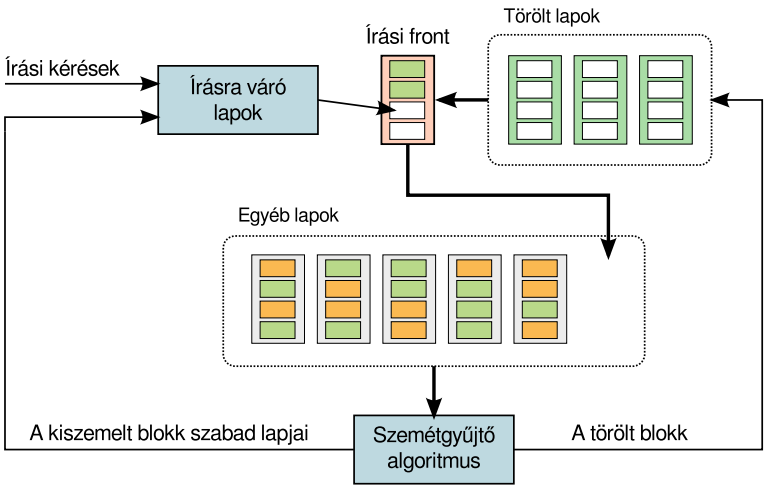
Ezeknek a lépéseknek a kellő számú ismétlésével gyarapítható a törölt blokkok listája.

Ha a bekapcsol a szemétgyűjtő algoritmus a 6.29. ábrán látható helyzetben, akkor valószínűleg az 1-es blokkot fogja rendbetételre kiválasztani. A rajta lévő egyetlen hasznos lapot (10-es) az írási frontra másolja, majd törli a blokkot (törölheti, hiszen hasznos adat már nincs rajta). Ezzel a a 6.30. ábrán látható helyzet alakul ki, eggyel több törölt blokkal.



6.30. ábra. A szemétgyűjtés eredménye a 6.29. ábra példájából kiindulva

Az írás és a szemétgyűjtés folyamatát a 6.31. ábrán foglaljuk össze ([22]). A megismert működésnek van egy fontos következménye, *az SSD egyre lassabb lesz, ahogy egyre többet használjuk.* Egy új SSD tele van törölt blokkokkal, a szemétgyűjtő algoritmusra jó ideig nincs szükség. Használat közben azonban egyre fogynak a törölt blokkok, és előbb-utóbb a szemétgyűjtés is lassítani fogja az írási műveleteket. A korai SSD-kben ez a hatás az idő előrehaladtával drasztikus lassulást okozott (hiszen megabyte nagyságrendű adatmozgatást igényelt a legkisebb írási kérés is). Ma már a korszerű vezérlők "ráérő" idejükben is szemetet gyűjtenek. Erre persze csak akkor képesek, ha van ráérő idejük. Ha olyan intenzitással érkeznek az írási/olvasási kérések (pl. szerver környezetben), hogy nem jut idő a lapok rendezgetésére, az SSD drasztikusan lelassul, akár még a HDD-nél is lassabb lehet.



6.31. ábra. Az írás és a szemétgyűjtés folyamata

6.2.6. Az SSD vezérlők jellemzői és feladatai

Az SSD-ben a *vezérlőnek* (controller) nagyon fontos szerepe van, sokkal több a feladata, mind egy HDD esetén (az SSD vezérlője nem keverendő össze az illesztőfelület Host Controller-ével). Célja, hogy a NAND flash jellegzetességeiből adódó nehézségeket, kényelmetlenségeket elrejtse, és egy egyenletes teljesítményű, nagy sebességű, megbízható háttértárat mutasson a külvilág felé.

Ennek érdekében minden SSD vezérlőnek áldozatokat kell hoznia. Ezek közül a legfontosabbak az alábbiak.

- *Többletírás.* Az SSD tranzisztorait nem csak a felhasználó által ráírt adatok koptatják. Maga az SSD is végez írás műveleteket, melyeket nem a felhasználó kezdeményez, pl. a szemétgyűjtés közben is írások sorozata történik. Ez a jelenség a többletírás (Write amplification). Annál jobb egy SSD vezérlő, minél kisebb többletírással képes megoldani a működését, hiszen ekkor a leghosszabb a meghajtó élettartama.
- *Túlméretezés.* A túlméretezés (Over-provisioning) azt jelenti, hogy az SSD fizikai tárolókapacitása nagyobb, mint a felhasználók számára felkínált tárolókapacitás. A kettő különbsége tartalékként működik. Több szempontból is hasznos lehet, hogy viszonylag nagy mennyiségű törölt blokk mindig rendelkezésre áll: gyorsítja a szemétgyűjtést, kiváltja az elöregedett blokkokat, stb. Másfelől viszont mindez a hasznos (felhasználó által látható) tárkapacitás rovására megy. Tipikus mértéke 6.25%, ami azt jelenti, hogy egy 128 GB-os meghajtóból 120 látszik, egy 512-esből 480, stb. Egyes gyártók (pl. Samsung) lehetővé teszik a túlméretezés szoftveres beállítását, vagy akár teljes kikapcsolását.

Végül megemlítünk néhány eljárást, melynek célja a teljesítmény további növelése. Ezek már nem létfontosságú funkciók, nélkülük is működőképes az SSD, de egy modern vezérlő ezeket mind alkalmazza.

- *Kopás-kiegyenlítés* (Wear Leveling) Egy háttértár ritkán és gyakran változó tartalmat egyaránt tárol. Gondoljunk csak arra, hogy az operációs rendszer részei vagy a futtatható állományok az SSD-re írás után sokáig nem fognak változni, míg például a böngésző cache egy nagyon változékony terület, böngészés közben folyamatosan új tartalom kerül bele, régi tartalom törlődik ki. Ennek az a következménye, hogy az SSD-n a ritkán változó tartalmat tároló blokkok nagyon lassan, a változékony tartalmat tároló blokkok pedig nagyon gyorsan öregszenek, kopnak. Sokkal szerencsésebb lenne, ha a kopás a teljes tárolási kapacitásra egyenletes lenne, ekkor ugyanis a teljes kapacitás a lehető legtovább maradna használható, majd egyszerre menne tönkre. Az SSD vezérlő feladata, hogy a kopást egyenletessé tegye. Ennek érdekében egy táblázatot tart nyilván, amiben tárolja, hogy melyik blokk hány törlésen esett már át. A kiegyenlített kopás elérése érdekében az írási frontot célszerű úgy kiválasztani, hogy az minél kevésbé kopott legyen (a nagyon kopottakat kímélendő). A szemétgyűjtő algoritmusnak is célszerű néha a kevésbé változékony blokkok törlése, még akkor is, ha csupa használt lapot tartalmaznak, hogy a kopás minél egyenletesebb legyen.
- *Adattömörítés.* Az SSD-re kerülő adatok tömörítésével (compression), kevesebb lap írására lehet szükség. Ha egy fájl kevesebb lapot foglal, az mind sebesség, mind élettartam szempontjából kedvező. Van olyan vezérlő (pl. a SandForce), ami nemcsak hogy tömörít, de a duplikációt is elkerüli. Egy rendkívül hatékony (és persze titkos) algoritmussal meg tudja állapítani, hogy egy bizonyos tartalom szerepel-e már az SSD-n, vagy sem. Ha szerepel, akkor nem is kell még egyszer eltárolni, ami megint csak gyorsabb válaszidőt és hosszabb élettartamot eredményez. Az SSD vezérlő által végzett tömörítés ugyanakkor a felhasználó számára láthatatlan, a meghirdetett tárolási kapacitásnál a hatékony tömörítés ellenére sem lehet több adatot tárolni a meghajtón!
- *Több írási kérés bevárása.* (Write coalescence) Az írási kéréseket sok SSD nem hajtja végre azonnal, hanem kicsit vár (egy belső memóriában tárolva az adatokat), hátha érkezik még 1-2 írási kérés, ami ugyanarra a lapra vonatkozik. Szerencsés esetben a több, azonos lapra vonatkozó írási kérést egyetlen flash programozási lépéssel tudja tárolni, ami az öregedés szempontjából kedvező.
- *Belső RAID szervezés.* Mivel minden SSD-nek szembe kell néznie a blokkok elöregedésével, a hibadetektálás és a hibák megfelelő kezelése megkerülhetetlen funkció. Az adatokat gyakran többszörözve tárolják (ha az egyik blokk elöregedett, legyen biztonsági tartalék). Az SSD-k a külvilág felé eltakarva ugyan, de belül sokszor a HDD-k RAID szervezéséhez hasonló redundáns szervezést használnak a biztonságos adattárolás érdekében.

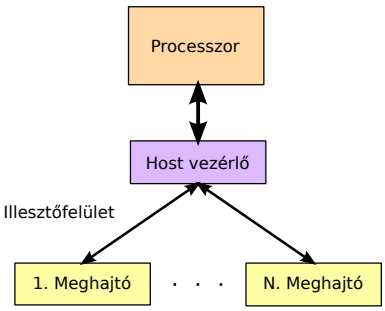
6.3. Illesztőfelületek

Az illesztőfelület az a kommunikációs csatorna, melyen keresztül adatátviteli kéréseket lehet intézni a meghajtó (pl. merevlemez, SSD) felé, és melyen keresztül a meghajtó válaszol azokra. Mint minden illesztőfelület, az adattárolók illesztőfelületei is két szinten specifikálják a kommunikáció miként jét:

- A fizikai specifikáció adja meg a jelátvitel részleteit: milyen legyen a kábel, milyenek legyenek a csatlakozók, hogyan kell egy bitet átvinni, felfutó vagy lefutó élre történjenek-e az események, stb.

- A protokoll specifikáció pedig leírja, hogy milyen párbeszéd szükséges az adatátvitel lebonyolításához: milyen parancsok vannak, milyen információkat tartalmazzon egy adatátviteli kérés, milyen sorrendben kell ezeket az információkat eljuttatni a merevlemezhez, hogyan kell elvenni a merevlemeztől a kérésre adott választ, stb.

A kommunikáció résztvevői a host vezérlő (host controller), valamint a meghajtók (6.32. ábra). A processzor a host vezérlővel áll kapcsolatban, neki küldi el az adatátviteli kérést, amit a host vezérlő az illesztőfelület nyelvére lefordítva továbbít az érintett meghajtó felé. A host vezérlő például lehet a déli híd integrált része, vagy egy PCI/PCI Express periféria is.



6.32. ábra. Az adatátvitel résztvevői

Manapság a legelterjedtebb illesztőfelületek az ATA, annak soros átvitelt alkalmazó változata, a SATA, a SCSI, a soros SCSI és a Fibre Channel (FC). Ezek mindegyikének van valami jellegzetes tulajdonsága, ami a számítástechnika egy-egy szegmensében egyeduralkodóvá teszi.

ATA illesztőfelület

Az ATA illesztőfelület egyértelműen a PC architektúrához kötődik. A felsorolt illesztőfelületek közül egyszerűsége miatt az ATA illesztőfelület támogatása a legolcsóbb a merevlemez és a PC gyártók számára is.

Az ATA 16 bites adategységeket definiál, melyeket egy 40 eres szalagkábelen párhuzamosan visz át. Az adatátvitel nem szinkron, az adategységek kiírása ill. beolvasása a ”Write Strobe” és ”Read Strobe” vezetékek alacsony szintje mellett történik. A 40 eres kábelt a nagy sebességű UDMA mód megjelenésével 80 eresre kellett cserélni 40 extra föld kábel beiktatásával (ez a nagy adatsebesség mellett már nem elhanyagolható kábelek közötti áthallás csökkentésére szolgált). Egy szalagkábelre két meghajtót lehetett csatlakoztatni, melyeket master-nek és slave-nek hívnak, de ez valójában hibás elnevezés, mert semmi köze a buszok szereplőinek master-slave szerepéhez (a szabvány későbbi változatai a ”Drive-0” és ”Drive-1” elnevezést használják).

Az ATA az alábbi adatátviteli módokat támogatja:

- PIO Mode: Ebben a módban a teljes adatátvitelt a processzor felügyeli, kiadja a meghajtónak a parancsot, majd minden egyes bájt átviteléről egyesével intézkedik (pl. írás esetén a processzor bájtonként olvassa ki a memóriából az adatokat és adogatja a meghajtónak). Mivel az adatátvitel ideje alatt a processzor nem tud mással foglalkozni (pl. más taszk végrehajtására kapcsolni, míg az adatátvitel le nem zajlik), multi-taszking környezetben nem a PIO mód a legszerencsésebb választás.
- DMA Mode: DMA üzemmódban a merevlemez DMA vezérlője képes az adatokat közvetlenül a rendszermemóriából kiolvasni, illetve közvetlenül oda írni. A processzornak csak kiviteli/beviteli kérést kell a meghajtóhoz eljuttatnia, az adatátvitel már a közreműködése nélkül történhet, például kihasználhatja az adatátvitel lezárásáig rendelkezésre álló időt arra, hogy közben más taszkok végrehajtásával is haladjon.
- Ultra DMA Mode: Az UDMA mód ugyanúgy működik, mint a DMA mód, de az órajel fel- és lefutó élén is történik adatátvitel, tehát az adatátviteli sebesség megduplázódott. Ennél a sebességnél már szükség van a 80 eres szalagkábelre. Ez a mód biztosítja a leggyorsabb adatátvitelt, 133 MB/s-ot.

A kiviteli/beviteli kérések indításához először a kérés paramétereit kell a meghajtóhoz eljuttatni (átvinni kívánt adat hossza, memóriacím, stb.), majd a parancskód elküldése indítja el az adatátvitelt. Az adatátvitel aktuális állását egy státuszregiszter rendszeres olvasgatásával lehet nyomon követni. Az ATA szabvány egyik korlátozása, hogy addig nem lehet új adatátviteli kérést indítani, amíg az előző le nem záródott. Látni fogjuk, hogy ennek

komoly teljesítménybeli következményei lesznek, de az ATA ezt a kompromisszumot hozta az egyszerűségért és az alacsony költségért cserébe.

Amikor az optikai meghajtók megjelentek, az ATA szabványból számos olyan parancs hiányzott, melyek az optikai meghajtók kezeléséhez feltétlenül kellenek (pl. tálca kinyitása, médium típusának lekérdezése, van-e médium a meghajtóban, stb.). Ezek a parancsok az SCSI illesztőfelületben megvoltak. Érdekes módon az ATA szabványt nem az optikai meghajtók kezeléséhez szükséges parancsok beemelésével bővítették ki, hanem lehetővé tették, hogy az ATA kábelen és csatlakozókon SCSI parancsokat lehessen eljuttatni a meghajtókhoz. Ez az ATAPI (ATA Packet Interface), mely lehetővé tette, hogy a SCSI interfészt ”beszélő” optikai meghajtókat a PC-kben elterjedt ATA felületen keresztül lehessen csatlakoztatni.

SATA illesztőfelület

Ahogy láttuk, az ATA szabványban 80 eres, extra földkábelekkel teletűzdelt szalagkábel kellett a nagy sebességű UDMA mód használatához. Nyilvánvalóvá vált, hogy a túl széles, árnyékolatlan kábelen zajló párhuzamos adatátvitel a még nagyobb sebesség elérésének fő gátja. Erre a kihívásra adott válasz a soros ATA (Serial ATA, SATA) megjelenése (ezzel egyidejűleg a régi ATA-t átkeresztelték PATA-ra, párhuzamos ATA-ra).

A SATA illesztőfelület ugyanazt az ATA protokollt használja, mint a régi PATA: a parancsok, azok paraméterei, az adatátviteli módok mind ugyanazok. Csak a jelátvitel fizikai mikéntje változott meg: a SATA sodort, árnyékolt érpáron soros átvitellel dolgozik. Az eszközök pont-pont összeköttetésben állnak a SATA vezérlővel.

A nagyobb sebességű adatátvitelhez a már meglévő adatátviteli módokat újakkal egészítették ki. A SATA első változata 150 MB/s, a SATA-2 már 300 MB/s, a SATA-3 pedig 600 MB/s maximális adatátviteli sebességet biztosít.

A SATA-2 a nagyobb sebesség mellett egy másik fontos újdonságot is hozott: a meghajtók a parancsokat várakozási sorokban tudják várakoztatni (NCQ, Native Command Queueing). Ennek köszönhetően nem kell már megvárni a következő kiviteli/beviteli kéréssel az előző befejezését. A meghajtó rögtön fogadni tudja a következő parancsot, akkor is, ha épp el van foglalva, ilyenkor beteszi a várakozási sorába. Ráadásul, ha a meghajtónak több kiviteli/beviteli kérés van a sorában, azokat szabadon, tetszőleges sorrendben kiszolgálhatja, például úgy, hogy a fejnek éppen útba eső kéréseket veszi előre.

SCSI illesztőfelület

Az SCSI az ATA-nál sokkal fejlettebb, összetettebb szabvány. Egyrészt nagyobb funkcionalitást nyújt (kb. 60 parancsot ismer), másrészt nagyobb hangsúlyt fektet a megbízhatóságra, és a hibákból való felépülésre, mint az ATA. Ennek köszönhetően SCSI interfésszel drágább rendszerekben, szerverekben találkozhatunk (pl. Amiga, régebbi Apple Macintosh és SUN számítógépekben).

A SCSI valójában egy busz, melyre több meghajtó, valamint a SCSI vezérlő csatlakozik. Akár a PATA, a SCSI is párhuzamos adatátvitelt használ, 8 bites vagy 16 bites adategységekkel. Előbbi esetben 50 eres, utóbbi esetben 68 eres kábelre van szükség. A busz minden szereplőjéhez tartozik egy prioritás (a vezérlőé a legnagyobb), és önkiválasztó arbitrációval dől el, hogy éppen melyik kapja a busz vezérlésének a jogát. A kiviteli/beviteli kérések parancsleírók (Command Descriptor Block, CDB) formájában utaznak a buszon. A CDB tartalmazza a parancsot, és annak minden paraméterét is.

Az SCSI adatátviteli sebessége a kezdeti 5 MB/s-ról egészen 320 MB/s-ig nőtt, ennél többet a párhuzamos adatátvitelből nem hoztak ki. A SCSI interfészben a SATA-2-nél jóval korábban jelent meg a parancsok sorba állításának lehetősége (Command Queueing).

Soros SCSI illesztőfelület

Az SCSI-ben is eljött az a pont, amikor a párhuzamos interfész megakadályozta az adatátviteli sebesség további növelését. Ekkor a SCSI is soros átvitelre váltott. Ez azonban a SATA bevezetése után történt meg, ami számos előnnyel járt. Látva a SATA sikerét, a soros SCSI (Serial Attached SCSI, SAS) gyakorlatilag átvette a SATA soros jelátviteli interfészét, kábelekkel, csatlakozókkal együtt. A SAS tehát SCSI parancsokat visz át SATA fizikai interfészen. A fizikai kompatibilitás lehetővé tette, hogy a SAS tároló rendszerekhez SATA meghajtókat is lehessen csatlakoztatni. Ennek fizikai akadálya nincs, hiszen ugyanazok a csatlakozók, viszont a SAS-t egy olyan megoldással is kiegészítették, amivel ATA parancsokat lehet a SAS protokoll segítségével továbbítani (SATA Tunneling Protocol, STP).

Ezzel megvalósult az ATA és a SCSI parancsok mindkét irányú átjárhatósága:

- Az ATAPI szabvány lehetővé teszi, hogy SATA illesztőfelületen SCSI parancsokat küldjünk egy meghajtónak,

- az STP szabvány pedig lehetővé teszi, hogy ATA parancsokat küldjünk egy SAS tárolórendszerbe kötött SATA meghajtónak.

A SAS meghajtókat onnan ismerhetjük fel, hogy nem egy, hanem két csatlakozóval rendelkeznek, aminek köszönhetően két külön vezérlőre, esetleg két külön számítógépre lehet őket csatlakoztatni egyszerre. A kettős bekötés a hibatűrést javítja, hiszen valamelyik fél kiesése esetén a másik fél továbbra is eléri a meghajtó adatait.

Adatátviteli sebességben a SAS a 600 MB/s-mal jelenleg ugyanott tart, mint a SATA.

FC illesztőfelület

Az FC (Fibre Channel) egy optikai jelátvitelen alapuló, nagy sebességű adatátviteli technológia, gazdag funkcionalitással. Az előbbiekkel ellentétben az FC ”csak” egy háttértárak részére kifejlesztett jelátviteli szabvány, magas szintű logikai protokollt, vagyis kiviteli/beviteli parancskészletet nem specifikál. A gyakorlatban szinte kizárólag az SCSI parancskészletét használják az FC-n illesztett meghajtók.

Az optikai kábelnek köszönhetően óriási adatátviteli sebességet lehet elérni (2011-ben 3200 MB/s), és nagy távolságot lehet áthidalni (akár 50 km-t is). A költségek lefaragása érdekében később réz vezetékek használata is lehetővé vált, persze kisebb adatátviteli sebesség és távolság mellett.

Az FC tárolórendszer tulajdonképpen nem más, mint egy csomagkapcsolt hálózat, melynek csomópontjai a vezérlő (Controller), a meghajtók (Drive), és a köztes kapcsolók (FC Switch, feladata a SCSI kérések továbbítása a megfelelő irányba). Az SCSI kérést egy csomag formájában adogatják egymásnak a csomópontok, míg célba nem ér.

7. fejezet

Számpéldák, feladatok a perifériakezelés témakörében

7.1. Feladatok az általános perifériakezelés témakörében

1. Feladat

Processzorunk órajel frekvenciája 100MHz. A számítógéphez egy billentyűzetet kötünk, melyen átlagosan 10 karaktert ütnek le másodpercenként, de két leütés akár 50 ms-onként is követheti egymást. A billentyűzet állapotának lekérdezése (mely tartalmazza a lenyomott gomb kódját is) 500 órajelet igényel. A megszakítás-feldolgozási idő ezen felül még 100 órajel.

(a) Hányszor kell másodpercenként lekérdezni a billentyűzetet, hogy ne maradjunk le semmiről?

(b) Mekkora terhelést jelent a processzor számára, ha a billentyűzet kezelésére polling-ot használunk?

(c) Mekkora terhelést jelent a processzor számára, ha a billentyűzet kezelésére megszakításkezelést használunk?

Megoldás

(a) Ha nem akarunk lemaradni egyetlen gombnyomásról sem, 50 ms-onként kell lekérdeznünk a billentyűzetet. Vagyis a kérdésre a válasz:

$$\frac{1 \text{ lekérdezés}}{50ms} \cdot \frac{1000ms}{1s} = 20 \text{ lekérdezés}/s$$

(b) Először számoljuk ki, hány órajel megy el másodpercenként a billentyűzet kérdezgetésére:

$$20 \text{ lekérdezés}/s \cdot 500 \text{ órajel/lekérdezés} = 10^4 \text{ órajel}/s$$

Mivel a processzor órajel frekvenciája 100MHz (10^8 órajel/s), a terhelés:

$$\frac{10^4 \text{ órajel}/s}{10^8 \text{ órajel}/s} = 0.0001 = 0.01\%.$$

(c) Interrupt használatával csak akkor kell a billentyűzethez fordulni, ha tényleges leütés történt (10-szer másodpercenként). Ekkor azonban mind az megszakítás-feldolgozási idő, mind a gomb kódjának kiolvasása terheli a processzort. A terhelés tehát:

$$\frac{10 \text{ leütés}/s \cdot (100 \text{ órajel/leütés} + 500 \text{ órajel/leütés})}{10^8 \text{ órajel}/s} = 6 \cdot 10^{-5} = 0.006\%.$$

2. Feladat

Processzorunk órajel frekvenciája 1GHz. A számítógéphez egy 100 Mbit/s sebességű hálózati interfészt kötünk, melyen 1500 bájtos (=12000 bit) csomagok közlekednek. A periféria állapotának lekérdezése (mely nem tartalmazza a hálózati csomag tartalmát) 600 órajelet igényel. Az interrupt feldolgozási idő 400 órajel. Ha jött csomag, annak átvitele 5000 órajelet vesz igénybe. Jelenleg épp egy 24 Mbit/s sebességű letöltés van folyamatban a hálózaton.

(a) Hámy ms-onként érkeznek a csomagok 100 Mbit/s sebesség mellett? És 24 Mbit/s sebesség mellett?

(b) Hány ms-onként kell lekérdezni a perifériát, hogy biztosan ne maradjunk le egy csomag érkezéséről?

(c) Mekkora terhelést jelent a periféria kezelése a processzor számára, ha a csomagok érkezését polling-al figyeljük?

(d) Mekkora terhelést jelent a processzor számára, ha interrupt-ot használunk a csomag érkezések jelzésére?

Megoldás

(a) Azt kell kiszámolnunk, hogy a 100 Mbit/s sebességű hálózati interfészen mennyi idő szükséges egy csomag (12000 bit) átviteléhez:

$$\frac{12000 \text{ bit/csomag}}{100 \cdot 10^6 \text{ bit}/s} = 0.12 \cdot 10^{-3} s/\text{csomag} = 0.12 ms/\text{csomag}.$$

Ugyanez 24 Mbit/s sebességgel:

$$\frac{12000 \text{ bit/csomag}}{24 \cdot 10^6 \text{ bit}/s} = 0.5 \cdot 10^{-3} s/\text{csomag} = 0.5 ms/\text{csomag}.$$

(b) Pont annyi időnként kell lekérdezni, amennyi időnként elméletileg jöhet csomag, azaz $0.12 ms$-onként.

(c) A processzor terhelése ebben a példában két részből áll: az állandó kérdezgetésből, hogy volt-e csomag, valamint ha volt, akkor a csomag tartalmának átvételéből. A lekérdezésre fordított idő kiszámolható az alábbi módon.

$$\frac{1}{0.12 \cdot 10^{-3}} \text{ lekérdezés}/s \cdot 600 \text{ órajel/lekérdezés} = 5 \cdot 10^6 \text{ órajel}/s.$$

Ha érkezett csomag, az át is kell venni. Mivel $0.5 ms$-onként érkezik ténylegesen csomag, az átvételükre fordított órajelek száma másodpercenként

$$\frac{5000 \text{ órajel/csomag}}{0.5 \cdot 10^{-3} s/\text{csomag}} = 10 \cdot 10^6 \text{ órajel}/s.$$

A processzor órajel frekvenciája 1GHz, másodpercenként 10^9 órajellel tud gazdálkodni, tehát a terhelés:

$$\frac{5 \cdot 10^6 \text{ órajel}/s + 10 \cdot 10^6 \text{ órajel}/s}{10^9 \text{ órajel}/s} = 0.015 = 1.5\%.$$

(d) A 24 Mbit/s letöltési sebesség mellett $1/0.5 \cdot 10^{-3} s/\text{csomag} = 2000$ csomag/s sebességgel jönnek a csomagok. Minden csomag érkezésekor interrupt keletkezik, mely az interrupt feldolgozási idővel és a csomag átviteli idővel terheli a processzort. A processzor terhelése tehát

$$\frac{2000 \text{ csomag}/s \cdot (400 \text{ órajel/csomag} + 5000 \text{ órajel/csomag})}{10^9 \text{ órajel}/s} = 0.0108 = 1.08\%.$$

3. Feladat

Egy speciális számítógép egy 100MHz órajel frekvenciával hajtott processzort, egy melegedésre hajlamos, de ideiglenesen kikapcsolható perifériát, valamint egy hőmérséklet érzékelőt tartalmaz. A hőmérséklet érzékelő bármikor leolvasható (egy leolvasás 500 órajelet vesz igénybe), de be lehet állítani úgy is, hogy egy hőmérsékleti küszöb túllépésekor kérjen megszakítást (a megszakítás kiszolgálása 120 órajelet vesz igénybe). Túlmelegedés átlagosan percenként egyszer következik be, amikor is a processzor kikapcsolja a túlhevült perifériát, majd nem sokkal ezután újra bekapcsolja azt.

(a) Ha nem használunk megszakítást, hányszor kell másodpercenként lekérdezni a hőmérőt, ha a rendszerünk mindössze $1ms$ ideig tolerálja a túlhevülést?

(b) Mekkora átlagos késéssel értesülünk a túlhevülésről, ha nem használunk megszakítást? (ms-ban kifejezve)

(c) Ha nem használunk megszakítást, mekkora terhelést jelent a processzornak a hőmérő folyamatos lekérdezése? (%-ban kifejezve)

(d) Mekkora terhelést jelent a processzornak (%-ban), ha a túlhevülés figyelését megszakításkezeléssel oldjuk meg?

Megoldás

(a) Ha $1ms$-ig tolerálja a rendszer a túlhevülést, akkor $1ms$-onként kell lekérdezni a hőmérőt, vagyis másodpercenként 1000-szer.

(b) Lehet, hogy a lekérdezéssel épp elkaptuk a túlhevülés pillanatát, ekkor $0ms$ a késleltetés, de az is lehet, hogy épp a lekérdezés után következett be, ekkor $1ms$ lesz a késleltetés. A kérdésre a válasz a kettő átlaga, vagyis $0.5ms$.

(c) Másodpercenként 1000 lekérdezéssel és 500 órajeles lekérdezési idővel számolva a terhelés

$$\frac{1000 \text{ lekérdezés}/s \cdot 500 \text{ órajel/lekérdezés}}{100 \cdot 10^6 \text{ órajel}/s} = 0.005 = 0.5\%,$$

hiszen a processzor sebessége 100MHz= $100 \cdot 10^6$ órajel/s.

(d) Megszakítás esetén csak a 120 órajeles interrupt feldolgozási idővel kell számolnunk, hiszen az interrupt-ból már eleve értesülünk az eseményről, nem kell a hőmérőt is leolvasnunk. Mivel átlagosan percenként egyszer kapunk interrupt-ot, a terhelés

$$\frac{\frac{1}{60} \text{ interrupt}/s \cdot 120 \text{ órajel/interrupt}}{100 \cdot 10^6 \text{ órajel}/s} = 2 \cdot 10^{-8} = 2 \cdot 10^{-6}\%.$$

7.2. Feladatok a merevlemezek témakörében

4. Feladat

Egy merevlemez 3 db kétoldalas lemezt tartalmaz, melyek mindegyikén 50000 sáv található, minden sávban 1000 szektorral. A szektorok mérete 500 bájt. ZBR nincs, az adatátviteli interfész sebessége pedig $500 \cdot 10^6$ bájt/s. A parancsfeldolgozási késleltetés olyan kicsi, hogy nullának tekintjük. Az átlagos seek idő 8 ms, a lemez forgási sebessége 10000 fordulat/perc.

(a) Adjuk meg CHS koordináta rendszerben a lemez kapacitását!

(b) Mekkora a merevlemez kapacitása bájtokban mérve?

(c) Menyi a lemezek teljes körülfordulási ideje?

(d) Mennyivel lassabb egy 100 egymásutáni szektorra vonatkozó olvasási kérés teljes kiszolgálási ideje, mint egy 1 szektorra vonatkozó kérésé?

(e) Mekkora seek idővel érhetjük el, hogy egy szektor teljes kiszolgálási ideje ugyanakkora maradjon, ha a lemezt fele olyan gyorsan forgatjuk? (ms-ban kifejezve)

Megoldás

(a) A cilinderek száma megegyezik a sávok számával, vagyis $C = 50000$. Mivel három lemezünk van, mindegyiken két adathordozó réteggel, a fejek száma $H = 6$. A sávonkénti szektorok száma pedig $S = 1000$.

(b) A merevlemez kapacitása:

$$\begin{aligned} &3 \text{ lemez/merevlemez} \cdot 2 \text{ hordozóréteg/lemez} \cdot 50000 \text{ sáv/hordozóréteg} \\ &\cdot 1000 \text{ szektor/sáv} \cdot 500 \text{ bájt/szektor} = 150 \cdot 10^9 \text{bájt/merevlemez} \end{aligned}$$

(c) A körülfordulási idő a forgási sebességből számítható,

$$\frac{1}{10000} \text{ perc/fordulat} \cdot 60000 \ ms/\text{perc} = 6ms,$$

hiszen 1 perc 60000 ms-ból áll.

(d) A feladat megválaszolásához célszerű kiszámolni a kiszolgálási idő komponenseit:

$$\begin{aligned} \text{parancsfeldolgozási idő} &= 0 \\ \text{átlagos seek idő} &= 8ms \\ \text{átlagos forgási késleltetés} &= \text{teljes fordulat ideje } /2 = 3ms \\ \text{egy szektor leolvasási ideje} &= 1/1000\text{-ed fordulat ideje} = 0.006ms \\ \text{adatátviteli idő} &= \frac{500 \text{ bájt/szektor}}{500 \cdot 10^6 \text{ bájt}/s} = 0.001ms \end{aligned}$$

Ezek alapján az egyetlen szektor olvasására vonatkozó kérés teljes kiszolgálási ideje:

$$8ms + 3ms + 0.006ms + 0.001ms = 11.007ms.$$

Ha 100 egymás utáni szektort kell beolvasni, akkor csak egyszer kell seek-elni és a forgási késleltetést megvárni. Továbbá, a szektor leolvasási idő és az adatátviteli idő átlapolhatók, vagyis az első szektor átvitele közben már lehet a második szektort leolvasni a lemezről. Az átlapolás miatt a leolvasás + átvitel együttesen $100 \cdot 0.006ms + 0.001ms$ ideig tart (a második tag az utolsó szektor átviteli ideje, azt nem lehet átlapolni). A teljes kiszolgálási idő tehát

$$8ms + 3ms + 100 \cdot 0.006ms + 0.001ms = 11.601ms.$$

(Vegyük észre, hogy alig több, mint az 1 szektorra vonatkozó esetben!)

(e) Ha a lemez fele olyan gyorsan forog, akkor a forgási késleltetés $6ms$-ra nő, a szektor leolvasási idő pedig $0.012ms$-ra. A kérdésre a választ az alábbi egyenlet megoldásából kapjuk:

$$x + 6ms + 0.012ms + 0.001ms = 11.007ms,$$

amiből a kívánatos seek időre $x = 4.994$ adódik.

5. Feladat

Egy merevlemez 3 db kétoldalas lemezt tartalmaz, melyek mindegyikén 20000 sáv található, minden sávban 1000 szektorral. A szektorok mérete 500 bájt. ZBR nincs. Az átlagos seek idő 4 ms. Mérésekkel megállapítottuk, hogy az egy véletlen szektorra vonatkozó olvasási kérések átlagos kiszolgálási ideje 10 ms.

(a) Ha a parancsfeldolgozási késleltetéstől és az interfészen való átviteli időtől eltekintünk, milyen gyorsan forog a lemez? (fordulat/perc-ben megadva)

(b) Meddig tart egy szektor leolvasása az adathordozóról, ha már ott van a fej?

(c) Ha a parancsfeldolgozási idő 0.1 ms, az adatátviteli interfész sebessége pedig $50 \cdot 10^6$ bájt/s, akkor mennyi a 2000 bájtos blokkokra vonatkozó (véletlen) adatátviteli sebesség? Mennyi az $50 \cdot 10^6$ bájtos blokkokra vonatkozó (folytonos) adatátviteli sebesség?

Megoldás

(a) A feladat megoldásához tudni kell, hogy a forgási késleltetés a lemez körülfordulási idejének (x) a fele ($x/2$), a szektorok leolvasási idejét pedig úgy kaphatjuk meg, hogy a körülfordulási időt elosztjuk az egy sávban található szektorok számával ($x/1000$). Így x-re az alábbi egyenletet kapjuk:

$$4 + x/2 + x/1000 = 10,$$

amiből $x = \frac{6000}{501}\ ms$ adódik. Ezt átváltva fordulat/perc-re a kérdésre a válasz

$$\frac{60000\ ms/\text{perc}}{\frac{6000}{501}\ ms/\text{fordulat}} = 5010\ \text{fordulat/perc}.$$

(b) Ha már ott van a fej, a leolvasási időt úgy kapjuk meg, hogy a körülfordulási időt elosztjuk az egy sávon lévő szektorok számával:

$$\frac{\frac{6000}{501}\ ms}{1000} = \frac{6}{501}\ ms = 0.011976\ ms$$

(c) A szektorok átviteli idejét eddig még nem számoltuk ki. Az $50 \cdot 10^6$ bájt/s-os interfészen egy 500 bájtos szektor átviteli ideje:

$$\frac{500\ \text{bájt/szektor}}{50 \cdot 10^6\ \text{bájt}/s} = 10^{-5} s = 0.01\ ms.$$

Mivel ez az érték kisebb, mint a szektor leolvasási idő, az átlapolás kihasználásával a kiszolgálási idő:

$$0.1 + 4 + \frac{6000}{501}\frac{1}{2} + 4 \cdot 0.011976 + 0.01 = 10.146 ms,$$

hiszen 2000 bájt 4 szektor átvitelét jelenti. Másodpercenként így $1000/10.146 = 98.561$ kérést tud a merevlemez feldolgozni (IOPS), tehát a véletlen adatátviteli sebesség $98.561 \cdot 2000 = 197122$ bájt/s.

A folytonos adatátviteli idő kiszámolásához $50 \cdot 10^6/500 = 100000$ szektor átviteli idejét kell kiszámolnunk, ami

$$0.1 + 4 + \frac{6000}{501}\frac{1}{2} + 100000 \cdot 0.011976 + 0.01 = 1207.7 ms.$$

Ebből másodpercenként $1000/1207.7 = 0.828$ szolgálható ki (IOPS), a folytonos adatátviteli idő pedig $0.828 \cdot 50 \cdot 10^6 = 41.4 \cdot 10^6$ bájt/s.

6. Feladat

Egy merevlemez 3 db kétoldalas lemezt tartalmaz, melyek mindegyikén 100000 sáv található. A sávok két zónára oszlanak (ZBR), 1-től 50000-ig sávonként 2000, 50001-től 100000-ig sávonként 1000 szektorral. A szektorok mérete 500 bájt. Az adatátviteli interfész sebessége pedig $250 \cdot 10^6$ bájt/s. A parancsfeldolgozási késleltetés 1 ms. Az átlagos seek idő 5 ms, a lemez forgási sebessége 6000 fordulat/perc.

(a) Mekkora a merevlemez kapacitása bájtokban mérve?

(b) Mennyi a lemezek teljes körülfordulási ideje?

(c) Mennyi egy egyetlen szektorra vonatkozó olvasási kérés átlagos kiszolgálási ideje, ha a szektor a 25000-es sávba esik?

(d) És ha a szektor a 75000-es sávba esik?

Megoldás

(a) A merevlemez kapacitása:

$$\begin{aligned}&3 \text{ lemez/merevlemez} \cdot 2 \text{ hordozóréteg/lemez}\\ &\cdot(50000 \text{ sáv/hordozóréteg} \cdot 1000 \text{ szektor/sáv} + 50000 \text{ sáv/hordozóréteg} \cdot 2000 \text{ szektor/sáv})\\ &\cdot 500 \text{ bájt/szektor} = 450 \cdot 10^9 \text{bájt/merevlemez}\end{aligned}$$

(b) A körülfordulási idő a forgási sebességből számítható,

$$\frac{1}{6000} \text{ perc/fordulat} \cdot 60000 \ ms/\text{perc} = 10 ms,$$

mivel 1 perc 60000 ms-ból áll.

(c) Egy szektor adathordozóról való leolvasási ideje az első zónában:

$$\frac{10 ms/\text{fordulat}}{2000 \text{szektor/fordulat}} = 0.005 ms/\text{szektor}.$$

A megadott sebességű interfészen egy 500 bájtos szektor átviteli ideje:

$$\frac{500 \text{ bájt/szektor}}{250 \cdot 10^6 \text{ bájt}/s} = 2 \cdot 10^{-6} s = 0.002 \ ms.$$

A teljes kiszolgálási időhöz össze kell adni a parancsfeldolgozási időt (1 ms), az átlagos seek időt (5 ms), az átlagos forgási késleltetést (10 ms /2 = 5 ms), a szektor leolvasási idejét (0.005 ms), valamint az átviteli idejét (0.002 ms):

$$1 + 5 + 5 + 0.005 + 0.002 = 11.007 ms.$$

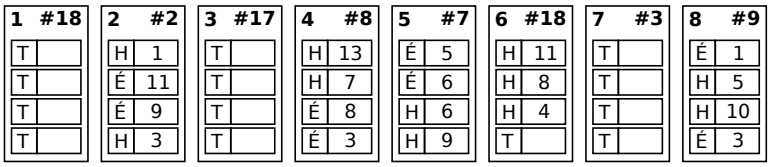
(d) A másik zónában csak annyi a különbség, hogy a szektorok leolvasása kétszer annyi ideig tart (hiszen fele annyi van belőlük sávonként), így a válasz

$$1 + 5 + 5 + 0.01 + 0.002 = 11.012 ms.$$

7.3. Feladatok az SSD témakörében

7. Feladat

Egy 8 blokkból álló SSD pillanatnyi állapota az alábbi ábrán látható.



A blokkok bal felső sarkában a blokk sorszáma, a jobb felsőben pedig az eddigi törlések száma látható. Minden blokk 4 lapot tárol, melyekhez nyilvántartjuk az állapotukat (”H”=használatban, ”É”=érvénytelen, ”T”=törölt), valamint, ha volt már rájuk írás, akkor az, hogy melyik LBA cím vonatkozik rájuk (most tekintsünk el attól, hogy a szektorméret és a lapméret nem egyenlő).

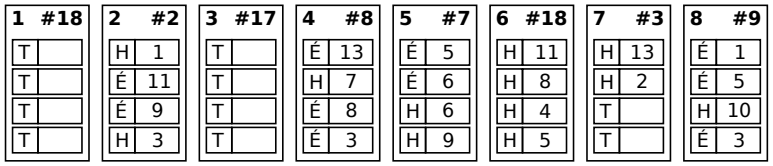
A kiinduló állapotban az 1-es, 3-mas és 7-es blokkok törölt állapotban vannak, az írási front pedig a 6-os blokk.

(a) Hogyan változik az SSD állapota, ha sorban egymás után az 5-ös, a 13-mas és a 2-es LBA címekre érkezik írási kérés? Ha új írási frontra van szükség, az SSD válassza azt, amelyik a lehető legegyenletesebb kopáshoz vezet!

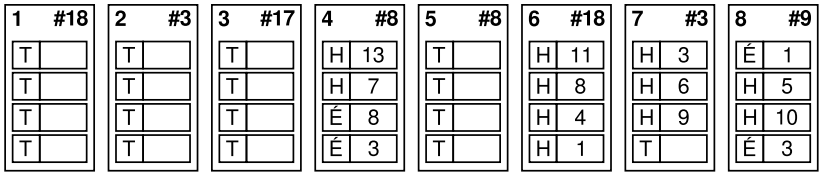
(b) Hogyan változik az SSD állapota, ha a kiindulási állapotban bekapcsol a szemétgyűjtő algoritmus, és meg sem áll, amíg a törölt blokkok listája eggyel nem nő (vagyis négy törölt blokk nem lesz)? (A szemétgyűjtő válassza mindig a legtöbb érvénytelen lapot tartalmazó blokkot, ha több ilyen is van, akkor azon közül a legkevésbé kopottat.)

Megoldás

(a) Az 5-ös LBA cím új tartalmát az írási front végére kell írni (6-os blokk, 4-es lap), és a régi változatát érvényteleníteni kell (8-as blokk, 2-es lap). Ezután az írási front betelik, újat kell keresni. Annak érdekében, hogy az SSD kopása a lehető legegyenletesebb legyen, a szabad blokkok közül (1-es, 3-mas, 7-es) azt választjuk, amelyik a legkevésbé kopott, vagyis a legkisebb a törlésszámlálója. Így a 7-es blokk lesz az új írási front. A 13-mas LBA cím tartalma már ide kerül (régi helyén, a 4-es blokk 1-es lapján érvénytelenítjük), és végül ide kerül a 2-es LBA címre szánt adat is. A végállapot tehát:



(b) A szemétgyűjtő algoritmus négy olyan blokkot is talál, melyen két érvénytelen lap van, ezek közül a 2-es a legkevésbé kopott, ezért ezt teszi rendbe. A 2-es blokk 1-es lapját az írási frontra másolja, új írási frontra vált (ez lesz a 7-es blokk), és oda másolja a 4-es lapot. Ezzel a 2-es blokk csupa haszontalan adatot tartalmaz, törölhető. (Ne felejtsük növelni a törlések számát a jobb felső sarokban!) Sajnos még mindig csak 3 törölt lap van, nem lett több, mint volt, tehát a szemétgyűjtés folytatódik. A következő rendrakásra kijelölt blokk az 5-ös. Két használatban lévő blokkját az írási frontra, vagyis a 7-es blokkra másoljuk, aztán ezt a blokkot is töröljük. Most már eggyel több törölt blokkunk van, a szemétgyűjtés befejeződik, az SSD állapota pedig a következő:



III. rész

A memória

8. fejezet

Memória technológiák

A 2.1.2. fejezetben megállapítottuk, hogy a memória a Neumann-architektúra szűk keresztmetszete, mivel a sebessége jóval kisebb ütemben nő, mint a processzoré. A memórián tehát a számítógép teljesítményét tekintve nagyon sok múlik. Ebben a fejezetben áttekintjük, hogy egy modern számítógép rendszermemóriája hogyan épül fel; hogy milyen megoldásokat és trükköket alkalmaznak annak érdekében, hogy a memóriakérések kiszolgálása minél gyorsabb, az átviteli sebessége pedig minél nagyobb legyen.

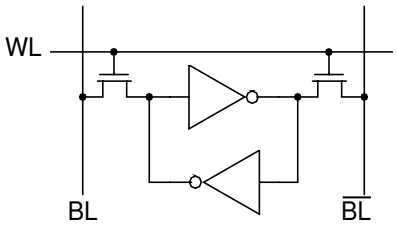
8.1. Adattárolás SRAM és DRAM cellákkal

A memória egyes bitjeinek tárolására két klasszikus megoldás terjedt el: a statikus RAM (SRAM), valamint a dinamikus RAM (DRAM). Látni fogjuk, hogy ezek nem versengő technológiák, sokkal inkább kiegészítik egymást, így mindkettővel érdemes megismerkedni.

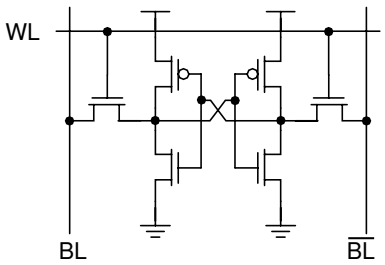
8.1.1. Az SRAM tárolócella

Az egyetlen bit tárolására képes SRAM tárolócella (8.1. ábra) nagyon emlékeztet egy egyszerű flip-flop-ra: a keresztbe kötött inverterek egy bi-stabil multivibrátort valósítanak meg. Ez az áramkör mindaddig képes megőrizni a bit értékét, amíg ellátjuk tápfeszültséggel. Az ábrán az inverterek mellett két tranzisztor is látható, melyek szerepe, hogy az inverterek kimenetét a bitvezetékre (BL, bit line) vezessék, ha a szóvezetékkel (WL, word line) a tárolócellát olvasás vagy írás műveletre jelöljük ki. Mivel egy invertert két tranzisztorral meg lehet valósítani, az SRAM memóriában egyetlen cella tárolásához 6 tranzisztor szükséges (lásd 8.1. ábra, jobb oldal), amit a szakirodalomban ”6T” cellának neveznek. (Léteznek 4T és 10T SRAM cella struktúrák is, de ezekkel most nem foglalkozunk, a 6T messze a legelterjedtebb).

Az olvasás a következőképpen zajlik. Először az előfeszítő áramkörök a ponált és a negált bitvezetékeket egyaránt logikai 1 szintre húzzák fel, majd a bitvezetékekről lekapcsolódnak. A bitvezetékek kapacitása miatt a bitvezetékek feszültségszintje megmarad. (Ez az előkészítő lépés azért hasznos, mert a celláknak könnyebb/gyorsabb egy előre feltöltött bitvezetéket 0-ra lehúzni, mint 1-re felhúzni.) A bit értékének kiolvasásához a WL szóvezetékre logikai 1 értéket adunk. A hozzáférési tranzisztorokon át a bitek értéke megjelenik a bitvezetékeken: ha a bit

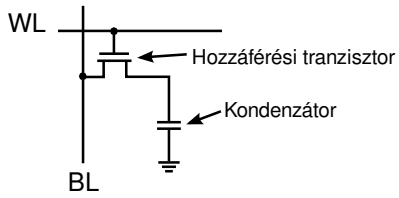


Inverterekkel



Tranzisztorokra lebontva

8.1. ábra. SRAM tárolócella



8.2. ábra. DRAM tárolócella

értéke 1, akkor a BL 1 marad, $\overline{\text{BL}}$-t pedig 0-ra húzza le a bit inverze; a 0 értékű bitnél pedig a helyzet éppen fordított. A BL és a $\overline{\text{BL}}$ különbségét az érzékelő erősítők érzékelik, és a kimenetükön rendelkezésünkre áll a bit értéke. Minél érzékenyebbek az érzékelő erősítők, annál gyorsabb a kiolvasás. A bitek felismerését az is segíti (és gyorsítja), hogy nem abszolút jelszinteket, hanem csak a BL és $\overline{\text{BL}}$ különbségét kell felismerni.

Az olvasás az SRAM állapotát (a tárolt bit értékét) nem változtatja meg (hamarosan látni fogjuk, hogy ez egyáltalán nem természetes).

Az írás hasonlóképpen zajlik. A ponált és negált bitvezetékekre ráadjuk a beírandó bitek logikai értékének megfelelő feszültséget: ha a bit értékét 1-be szeretnénk állítani, akkor $\text{BL} = 1, \overline{\text{BL}} = 0$; ha 0-ba, akkor $\text{BL} = 0, \overline{\text{BL}} = 1$. Ezt követően a WL szóvezetékre logikai 1 értéket adunk. A cella által tárolt bit értéke a hozzáférési tranzisztorain keresztül felveszi a bitvezetékek által rákényszerített értéket, mivel a bitvezetékek meghajtó tranzisztorai erősebbek, mint a cellák tranzisztorai.

8.1.2. Az DRAM tárolócella

Dinamikus RAM esetén a bitek tárolása teljesen másképp történik: a flip-flop helyett egy kondenzátor töltöttsége hordozza az információt (lásd a 8.2. ábra). A kondenzátor melletti hozzáférési tranzisztor szerepe az, hogy a kondenzátort összekösse a bitvezetékkel, írás vagy olvasás céljából. Mivel egyetlen bit tárolásához 1 tranzisztor és 1 kondenzátor szükséges, a DRAM cellák felépítését ”1T1C” névvel is illetik.

Ha ki szeretnénk olvasni a tárolt bit tartalmát, először is elő kell feszíteni a bitvezetéket, mégpedig logikai 0 és 1 szint közé ”félútra”. Ezután a szóvezetékre logikai 1-et adunk, aminek a hatására a hozzáférési tranzisztor zár, és a kondenzátor összekapcsolódik a bitvezetékkel, töltése (vagy a töltés hiánya) pedig módosítja a bitvezeték feszültségszintjét. A bitvezeték végén elhelyezkedő érzékelő erősítők érzékelik ezt a (kondenzátor apró mérete miatt egyébként igen kis) változást, és előállítják a megfelelő logikai 0 vagy 1 szintet. A DRAM tárolócellák nagyon fontos tulajdonsága, hogy *a kiolvasás destruktív*, hiszen például egy 1-es bit kiolvasáskor a kondenzátor töltése a bitvezetéken át távozik, ezért a kiolvasás után egy külön lépésben rögtön vissza is kell állítani számára az elveszett töltését.

A tárolócella írása hasonlóan történik. A szóvezeték ”kijelöli” a cellát (a hozzáférési tranzisztor a kondenzátort a bitvezetékre kapcsolja), majd a tárolni kívánt logikai szintnek megfelelő töltést a bitvezetéken keresztül a kondenzátorba kényszerítjük.

Sajnos a DRAM tárolócellák a bitek kiolvasása nélkül is elveszítik tartalmukat, mert a kondenzátor töltése képes magától elszivárogni. Emiatt időnként (néhányszor 10 ms-onként, jellemzően 64 ms-onként) a tárolt információ frissítésre szorul, ami abból áll, hogy ki kell olvasni és vissza kell írni a tárolt bitet. Ezt a fajta megoldást pont azért hívják *dinamikus* RAM-nak, mert periodikus frissítésre van szüksége.

8.1.3. Összevetés

Mint láttuk, az SRAM cellák 6T, a DRAM cellák 1T1C felépítésűek, ráadásul az SRAM-nál bitenként két bitvezeték van, ezért a kevesebb alkatrész miatt a DRAM technológiával adott méretű lapkán több (6-8x annyi) bit tárolható, következésképpen olcsóbb.

Működési sebességet tekintve azonban az SRAM előnyösebb, ennek oka szintén a belső felépítésében keresendő. Egy SRAM cellában a bi-stabil multivibrátor bitvezetékét aktív eszköz (inverter) hajtja meg, így gyorsabban beáll a bit értékének megfelelő jelszint (néhány ns), ezzel szemben a DRAM kondenzátorának csekélyke töltése csak lassabban képes a bit értékének megfelelő jelszint-változást okozni a bitvezetéken (néhányszor 10ns). A DRAM-ot ráadásul időnként frissíteni is kell.

Az SRAM további előnye, hogy gyártástechnológiai szempontból pont ugyanolyan tranzisztorokból áll, mint a processzor, így könnyedén ugyanazon félvezető lapkára lehet integrálni őket.

Mindkét technológia megtalálta a maga helyét a számítógépekben. Míg a nagy adatsűrűség és az alacsony ár miatt a számítógépek rendszermemóriája DRAM-ra alapul, a nagy sebességű adattárolást igénylő cache memóriák SRAM-ot használnak, és a processzor regisztereit is SRAM tárolja.

Az utóbbi években a DRAM egy változata, az eDRAM (embedded dynamic random access memory) is terjedni látszik. Az eDRAM cellák felépítésben és viselkedésben is hasonlítanak a DRAM cellákra, azonban a processzorral egy lapkára helyezhetők. Mivel előállításuk olyan gyártástechnológiai lépéseket is igényel, melyekre egy processzor gyártása esetén (eDRAM nélkül) nem lenne szükség, az eDRAM használata növeli a gyártási költségeket. A nagyobb költségeket azonban sok esetben ellensúlyozza a processzorhoz ”közel” fekvő, azzal széles, nagy sebességű belső buszon kommunikáló, az SRAM-nál lényegesen nagyobb tárolási kapacitással bíró memória jelentette teljesítménytöbblet. Az eDRAM-ot használó processzorok körébe tartozik az IBM POWER7 (2009, 32MB eDRAM alapú harmadszintű cache), az Intel Haswell processzorok GT3e grafikus vezérlőt tartalmazó változata (2013, 128MB eDRAM alapú negyedszintű cache), és számos játékkonzol processzora (a Sony Playstation 2 processzora 4MB eDRAM, a Playstation Portable MIPS alapú processzora 32MB eDRAM-mal rendelkezik, a Nintendo Wii U grafikus processzora 32MB eDRAM-ot tartalmaz).

8.2. DRAM alapú rendszermemóriák

Most, hogy megismertük, hogy egy DRAM cella hogyan képes tárolni egy bitet, megnézzük, hogy ezekből az 1 bites alapelemekből hogyan épül fel a számítógép rendszermemóriája.

8.2.1. Áttekintő kép

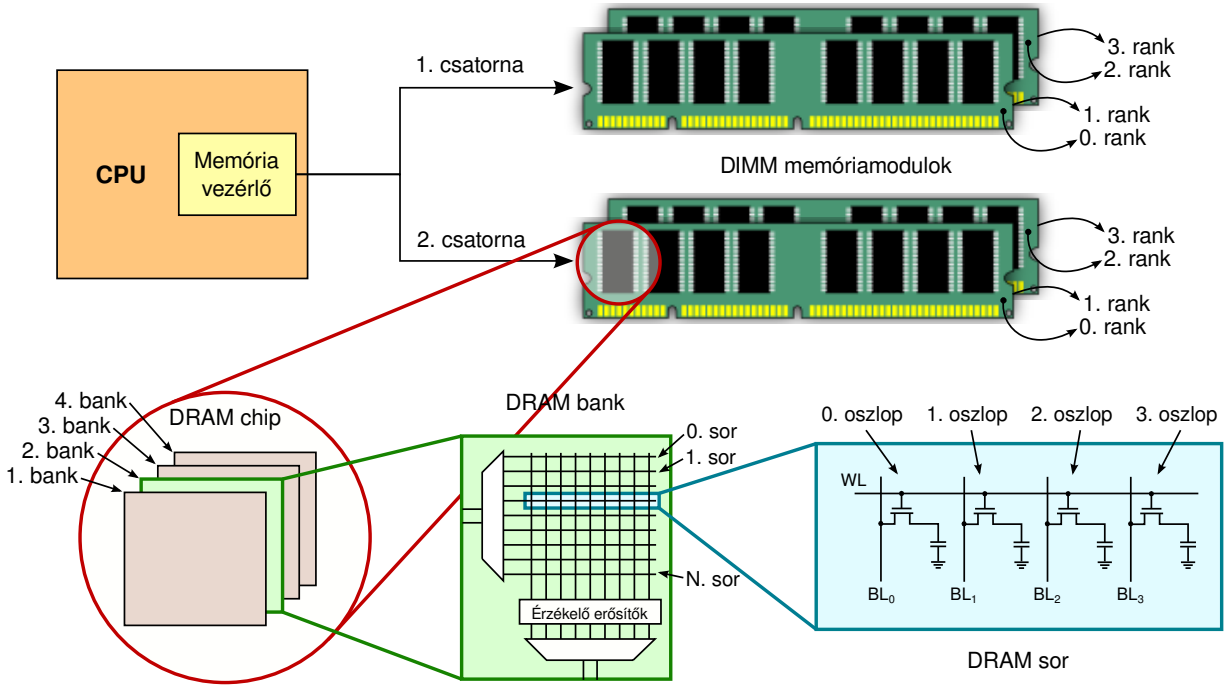
Egy modern DRAM alapú rendszermemória áttekintő képét mutatja be a 8.3. ábra. Mint látható, egy meglehetően összetett hierarchikus felépítésről van szó. A mai számítógépekben a processzor memória írási és olvasási kéréseit nem közvetlenül a memóriamodulokhoz, hanem a memóriavezérlőhöz intézi (ez sokszor a processzorral egy lapkán található, de ettől még egy különálló funkcionális egységnek tekintjük). Az ábra példájában egy kétcsatornás memóriavezérlőt látunk, mindegyik csatornán 2-2 DIMM modullal. A DIMM (dual in-line memory module) két oldala két független eszköznek minősül, melyek osztoznak a cím-, és adatbuszokon. Ezeket a független memória eszközöket ”rank”-nek hívja a szakirodalom. Minden rank több, általában azonos típusú DRAM chip-et tartalmaz. A DRAM chip-ek belső felépítése sem egyszerű. Egy DRAM chip több független tárolómezőből, úgynevezett ”bank”-ból áll. A DRAM bank nem más, mint a 8.1.2. fejezetben megismert DRAM tárolócellák mátrixba rendezett halmaza.

8.2.2. Egy DRAM bank felépítése

Egy DRAM bank felépítését a 8.4 ábra szemlélteti. Egy bankon belül az azonos sorban elhelyezett DRAM cellák szóvezetékei (WL), illetve az azonos oszlopba elhelyezett cellák bitvezetékei (BL) össze vannak kötve. A mátrixba elrendezett elemi tárolócellák szóvezetékei egy dekóder, a *sordekóder* (row decoder) kimenetei. A bitvezetékek pedig egy *érzékelő erősítők*be, majd egy multiplexerbe, az úgynevezett *oszlop-multiplexerbe* futnak be.

Két-fázisú hozzáférési protokoll

Ahhoz, hogy egy kívánt adatot kiolvassunk egy DRAM bankból, először a megfelelő sor címét kell a sordekóderre tenni. A sordekóder ennek (és csak ennek) a sornak a szóvezetékére logikai 1-es szintet ad, mire a sor összes cellája engedélyezett lesz, a sor összes kondenzátora összeköttetésbe kerül a bitvezetékekkel. Az érzékelő erősítők a bitvezetékek alapján megállapítják a cellákban tárolt bitek értékeit, és ideiglenesen tárolják is azt. Ezen a ponton tehát az érzékelő erősítőkben rendelkezésre áll a *teljes sor* tartalma, melyből az oszlopcím alapján az oszlop-multiplexer választja ki a megfelelőt (az érzékelő erősítőre szoktak sorbufferként, row buffer-ként is hivatkozni). Tehát a DRAM bank címzése két fázisban történt: először a sorcímet kellett rátenni a sordekóderre, majd (kisvártatva) az oszlopcímet az oszlop-multiplexerbe. Ezt az eljárást ezért két-fázisú hozzáférési protokollnak nevezik.



8.3. ábra. Modern DRAM alapú memóriarendszer

A lábak számának csökkentése végett a sor- és oszlopcímek átviteléhez ugyanazokat a címvezetékeket, lábakat használják. Az első fázisban ezekre a lábakra a sorcímet, majd ezután a második fázisban az oszlopcímet kell kitenni.

DRAM parancsok

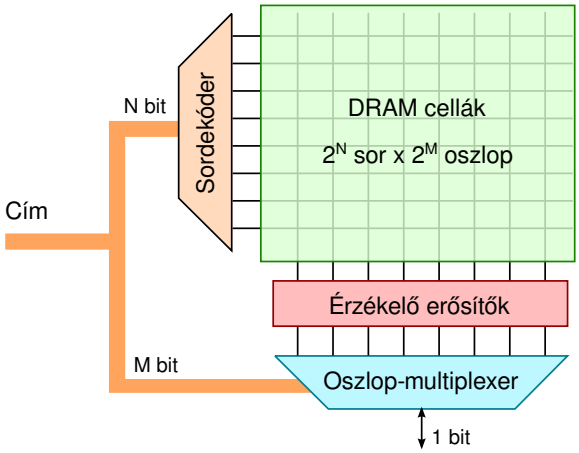
Egy DRAM bankkal *parancsok* segítségével lehet kommunikálni. A legfontosabb DRAM parancsok a következők:

- ACTIVATE: ezzel a paranccsal lehet megnyitni egy adott sort. A sor tartalma az érzékelő erősítőkbe kerül átmeneti tárolásra, a DRAM olvasás destruktív volta miatt pedig a cellák elvesztik tartalmukat. A cellák tartalmának regenerálása a sor megnyitása után azonnal (külön parancs nélkül) megindul.
- READ: ezzel a paranccsal lehet az éppen megnyitott sorból kiolvasni egy oszlopot. Valójában az érzékelő erősítőkben átmenetileg tárolt sor bitjei közül olvassa ki a megfelelőt.
- WRITE: ez a parancs az épp nyitott sor egy oszlopát megváltoztatja. A változást először a sort átmenetileg tároló érzékelő erősítőkben érvényesítik, majd a megváltozott DRAM cellák frissítése is megtörténik.
- PRECHARGE: az aktuális, nyitott sor bezárására szolgál. Az érzékelő erősítők alaphelyzetbe állnak, egyúttal feltöltik (precharge) az összes bitvezetéket is, hogy a következő sormegnyitás művelet, ha majd szükség lesz rá, gyorsan megtörténhessen.
- REFRESH: megnyit egy sort, majd rögtön le is zárja azt. Ezzel a sor minden bitje kiolvasásra, majd visszaírásra kerül. Erre hozzávetőleg 64 ms-onként szükség van, mert a cellák kondenzátoraiból elszivárgó töltések miatt e nélkül elveszne a tárolt információ. A frissítés felfrissíti a kondenzátorok töltését. A REFRESH parancsnak nincs paramétere, auto-inkremens módon mindig a soron következő sort fogja frissíteni.

Például, ha az adat olvasási kérések sorban egymás után az alábbi módon követik egymást:

```
(3.sor, 8. oszlop),
(3.sor, 14. oszlop),
(1.sor, 3. oszlop),
(1.sor, 4. oszlop),
```

akkor a következő parancsokra van szükségünk, feltéve, hogy a bank előfeszített (precharged) állapotban van:



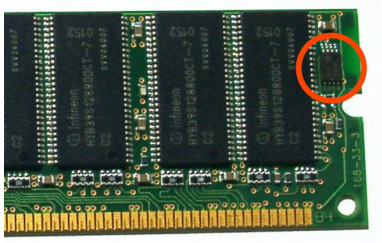
8.4. ábra. Egy DRAM bank felépítése

```
ACTIVATE 3
READ 8
READ 14
PRECHARGE
ACTIVATE 1
READ 3
READ 4
```

Egy nyitott sor eléréséhet tehát csak READ/WRITE parancsok szükségesek. Egy meg nem nyitott sor eléréséhez viszont szükség lehet a PRECHARGE-re, amennyiben az előzőleg használt sort még nem zárták be, majd ACTIVATE-el meg kell nyitni az elérni kívánt sort, a READ/WRITE parancsok csak ezután következhetnek.

Időzítési paraméterek

A DRAM-nak a parancsok végrehajtásához idő kell, melyet a memóriavezérlőnek figyelembe kell vennie, miközben memóriaműveleteket végez. Egy DRAM memória modul a DRAM chip-eken felül tartalmaz még egy IC-t (Serial presence detect, SPD, 8.5. ábra), mely ezeket az időzítési információkat tárolja, a rendszer indulásakor a memóriavezérlő innen szerez tudomást a parancsok késleltetéséről.



8.5. ábra. Az időzítési információkat tároló SPD

A sok-sok különféle időzítési érték (timing) közül érdemes kiragadni az alábbi négyet, mert egyrészt ezek a legfontosabbak, másrészt ezekkel szokták jellemezni a DRAM chip-ek késleltetését:

- T_{RCD} (Row-to-Column command Delay): Ennyi ideig tart egy sor megnyitása, vagyis ennyi idő telik el a megnyitás parancstól számítva addig, amíg a sor tartalma megjelenik az érzékelő erősítőkben.
- CL, vagy T_{CAS} (Column Access Strobe time): Egy nyitott sor egy oszlopának a kiolvasásához szükséges idő. Az olvasás parancs után ennyi idő múlva jelenik meg az (első) adat a modul adatbuszán.
- T_{RP} (Row Precharge): A PRECHARGE parancs végrehajtásához szükséges idő.

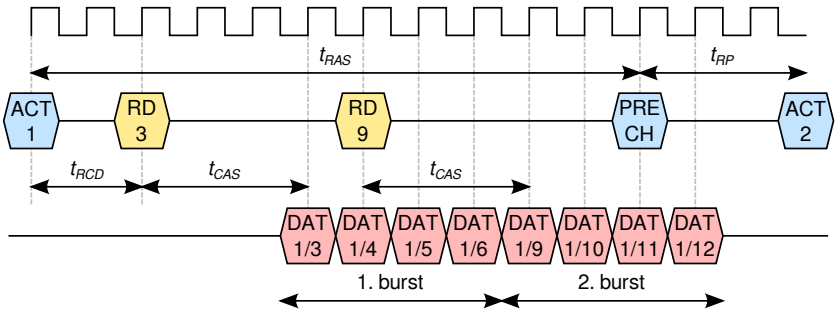
- T_{RAS} (Row Active Time): Az a minimális idő, amíg egy sor nyitva lehet (ennyi idő kell a kondenzátorok töltésének regenerálásához). Ha figyelembe vesszük, hogy egy új sor megnyitása előtt elő-feltöltés is szükséges, megkapjuk, hogy a sor megnyitási parancsok nem követhetik gyakrabban egymást, mint $T_{RC} = T_{RAS} + T_{RP}$.

Ezeket az időzítési értékeket aszinkron memóriák esetén időben (ns), szinkron DRAM esetén pedig órajelben kell érteni, és a számértékeket egymástól kötőjellel elválasztva $T_{CAS} - T_{RCD} - T_{RP} - T_{RAS}$ sorrendben szokták megadni. A 4 érték helyett sokszor egyszerűsítve, a T_{CAS} feltüntetésével adják meg a DRAM késleltetését (pl. egy CL7-es memóriamodul esetén $T_{CAS} = 7$).

Burst adatátviteli mód és átlapolt parancsfeldolgozás

Az imént tárgyalt időzítések ismeretében megállapíthatjuk, hogy ha nagyobb mennyiségű (sor-folytonos) adatot szeretnénk kiolvasni a memóriából, akkor ezt nem éppen gazdaságos oszloponként megtenni, hiszen minden egyes oszlop átvitelét egy T_{CAS} késleltetés előzi meg. A hatékonyság növelése érdekében két technikát vezettek be:

- A T_{CAS} késleltetés amortizálása érdekében számos DRAM technológia csoportos, ún. *"burst" átvitelt* használ, ami azt jelenti, hogy egy READ/WRITE parancsra nemcsak egy, hanem több egymás utáni oszlop átvitele is megtörténik. Az egyszerre átvitt oszlopok számát, azaz a burst hosszát egy konfigurációs parancs segítségével lehet beállítani. Minimális értékét az adott technológia előírja, tipikus értéke pedig megegyezik a CPU cache blokkok méretével. A minimálist burst hossz pl. DDR memóriák esetén 2, DDR2 esetén 4, DDR3 és DDR4 esetén pedig 8 oszlop.
- *A parancskiadás és az adatátvitel átlapolható,* vagyis új parancsot ki lehet adni még mielőtt a korábbira megérkezne a válasz.



8.6. ábra. DRAM adatátvitel, időzítésekkel

A 8.6 ábrán látható példán először megnyitjuk az 1-es sort, majd ezután $T_{RCD} = 2$ késleltetéssel adhatjuk ki az első olvasási kérést a 3. oszlopra. Az adatok $T_{CAS} = 3$ idő múlva jelennek meg a DRAM adatbuszán. Ha a burst hossz 4 oszlopnyira van beállítva, akkor nem csak a 3. oszlop, hanem a 4., 5. és 6. oszlop is az adatbuszra kerül. A következő olvasási kérés a 9. oszlopra vonatkozik. Figyeljük meg, hogy parancs már akkor kiadható, amikor az előző olvasási kérés még be sem fejeződött (ezt jelenti az átlapolás). Megfelelő időzítéssel el lehet érni, hogy az előző burst-öt rögtön kövesse a következő, így az adatbusz sávszélességének kihasználtsága növelhető. A két burst átvitele után a példa lezárja a nyitott sort egy PRECHARGE parancs segítségével. Ennek kiadásakor figyelembe kell venni, hogy a megnyitás és a lezárás között legalább T_{RAS} időnek el kell telnie. (A sort érintő utolsó READ parancs és a PRECHARGE közt eltelő időre is van időzítési előírás, de azzal most nem foglalkozunk.)

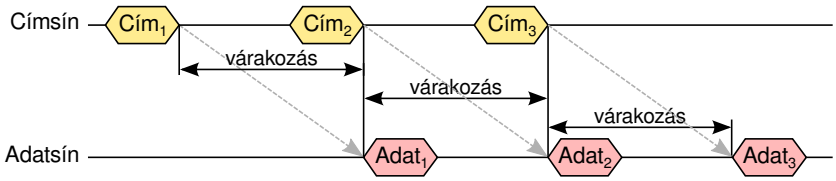
A parancsok ütemezése, a sok-sok időzítési kényszer figyelembevétele a memóriavezérlő feladata.

8.2.3. DRAM memóriamodulok

Egy DRAM memóriamodul több DRAM chip-et, valamint egy SPD IC-t (8.5. ábra) tartalmaz. Először megvizsgáljuk egy DRAM chip belső felépítését, majd azt, hogy az azonos memória modulon található DRAM chip-ek milyen viszonyban vannak egymással.

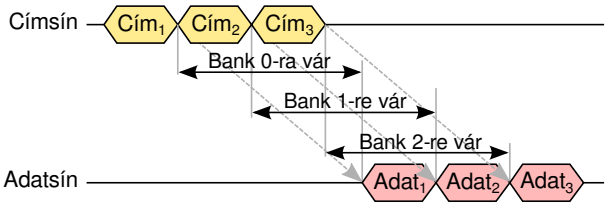
A DRAM chipek felépítése

Már a 8.3. ábrán is utaltunk rá, hogy egy DRAM chip több memória bankot tartalmaz. Ezek a bankok egymástól független működésre képes DRAM cella-mátrixok. Minden bank saját sordekóderrel, oszlopmultiplexerrel és érzékelő erősítőkkel rendelkezik. Mivel a bankok egymástól függetlenek, mindegyik tartalmazhat 1-1 nyitott sort. Összességében a DRAM chip így több sort is nyitva tarthat, ami kisebb memóriakésleltetést tesz lehetővé, hiszen több adathoz lehet a sormegnyitás/zárás többletkésleltetése nélkül hozzáférni.



8.7. ábra. Adatátvitel különböző sorokból, átlapolás nélkül

A 8.7. ábrán egy olyan esetet látunk, melyben az egymás után elérni kívánt memóriacímek $\texttt{Cím}_1$, $\texttt{Cím}_2$, ...más és más sorokra vonatkoznak, ezért minden esetben meg kell várni a hosszadalmas sormegnyitási és sorlezárási késleltetéseket. Ha azonban ezek a címek különböző bankokra vonatkoznak, akkor ezek a késleltetések átlapolhatók, így összességében a teljes adatátvitel hamarabb befejeződik (8.8).



8.8. ábra. Átlapolt adatátvitel több bank esetén

A DRAM chip-ekben lévő bankok számát az adott szabvány rögzíti. Az első generációs DDR-SDRAM memóriák, valamint a DDR2 memóriák 4 bankkal rendelkeztek, melyet a DDR3 szabvány 8-ra, a DDR4 pedig 16-ra növelt.

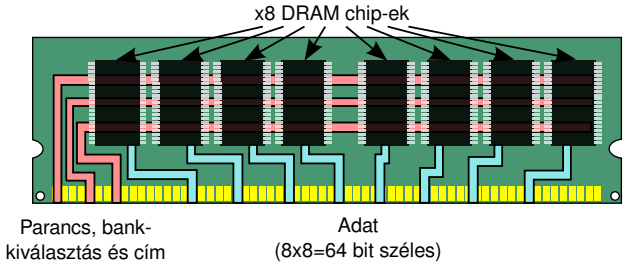
A DRAM chip-ek többnyire nem bit szervezésűek. A memória írási/olvasási műveletek több bitre vonatkoznak, vagyis a bankok a nyitott sor egy oszlopának kiválasztásakor nem 1, hanem több (4, 8, esetleg 16) bitet adnak vissza. Az oszlopok bitszélességének megfelelően megkülönböztetünk x4, x8 és x16 memóriamodulokat.

Az ismereteinket összegezve tehát a DRAM chip-ek az alábbi interfésszel rendelkeznek:

- Parancs vezetékek: Ezen keresztül lehet jelezni a chip-nek a kívánt művelet típusát (ACTIVATE, READ, WRITE, REFRESH, stb.)
- Bank kiválasztó vezetékek: Ezen keresztül jelezzük, hogy a művelet melyik bankot érinti.
- Címvezetékek: Itt kell ráadni az adott parancshoz tartozó címet. ACTIVATE esetén sorcímet, READ és WRITE esetén oszlopcímet vár a DRAM chip.
- Adatvezetékek: A READ/WRITE műveletek esetén ide kerül a kiolvasott adat, illetve innen veszi el a beírandó adatot a DRAM chip. Szélessége 4, 8 vagy 16 bites, attól függően, hogy x4, x8 vagy x16-os chip-ünk van-e.

A memóriamodulok felépítése

Mint láttuk, a DRAM chip-ek meglehetősen szűk adatszélességgel rendelkeznek (x4, x8 vagy x16-osak). Az adatátviteli sebesség növelése érdekében egy memóriamodul több DRAM chip-et tartalmaz, melyek teljesen szinkronizáltan működnek. Osztoznak a parancs, a bank kiválasztó és a cím vezetékeken, az adatvezetékeik összefogásával azonban a memóriamodul adategysége sokkal szélesebbé tehető. A 8.9. ábrán például 8 darab egyenként x8-as chip-et tartalmazó, összességében tehát 64 bites adategységeket használó memóriamodult kapunk. Minden parancsot minden chip megkap, tehát a modulon lévő chip-ek megfelelő bankjaiban ugyanazok a sorok lesznek nyitva.

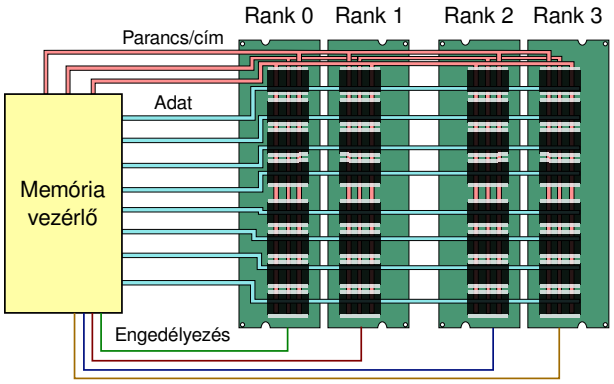


8.9. ábra. Egy DRAM memóriamodul felépítése

Ezzel a megoldással az *adatátviteli sebesség* 8-szorosára nőtt (hiszen minden kiadott memóriaművelet 8 bit helyett 64 bit adat beírásával/kiolvasásával jár), azt azonban nagyon fontos megjegyezni, hogy a *késleltetés*, vagyis a cím ráadásától az adat megjelenéséig eltelt idő nem változik.

8.2.4. Több DRAM rank használata

A tárolási kapacitás növelése érdekében több memória *rank*-et is alkalmazhatunk. A memória rank-ek egymástól független egységek, melyek nemcsak a parancs, a bank kiválasztó és a címvezetékeken, de az adatvezetékeken is osztoznak. Ezt úgy kell elképzelni, mintha több memóriamodult helyeznénk el egy sínre (pl. az alaplapon lévő foglalatokon keresztül), de mielőtt a rank és a memóriamodul fogalma teljesen összekeveredne, gyorsan megjegyezzük, hogy a kétoldalas DIMM memóriamodulok (Dual In-line memory module) két különálló rank-ként viselkednek. Tehát ha a számítógépünkben 2 db DIMM memóriamodult helyezünk (egyazon csatornára – lásd később), akkor 4 memória rank-ről beszélünk, melyek mindegyike 64 bites adategységeket használ. Mivel egyazon buszon helyezkednek el, és minden vezetéken osztoznak, gondoskodni kell róla, hogy mindig csak az egyik legyen bekapcsolva. Erre a célra a minden rank-hez külön, dedikáltan kihúzott engedélyező (chip select) vezetékek szolgálnak (8.10. ábra). Tehát ha a számítógép memóriavezérlője a 3. rank-ből szeretne adatot olvasni (ami lehet például a második DIMM modul első oldala), akkor azt az egyet be-, a többit pedig ki kell kapcsolnia.

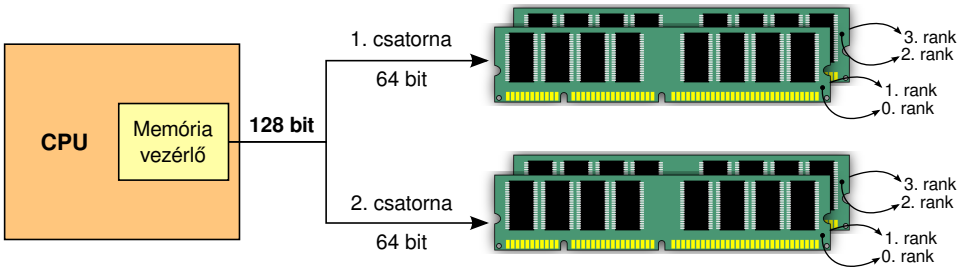


8.10. ábra. DRAM rank-ek egyazon csatornán

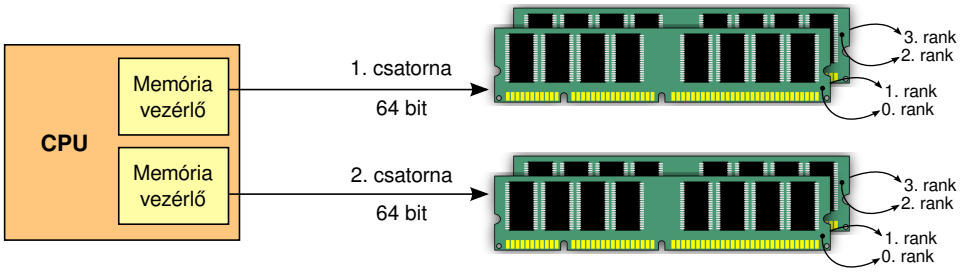
A memória rank-ek számának növelése (vagyis DIMM modulok, vagy több modul használata) nem növeli a memória sávszélességét, hiszen a rank-ek ugyanazon a (többnyire 64 bites) adatbuszon osztoznak. Több rank esetén azonban több sort tud a memóriavezérlő nyitva tartani, hiszen a rendszerben lévő bank-ok száma annyi szorosára nő, ahány rank-ünk van, így végső soron a több rank használata a tárolási kapacitás növelése mellett némi késleltetésbeli előnnyel is jár.

8.2.5. Több memóriacsatorna használata

Többcsatornás memóriavezérlőket több, mint 50 éve használnak a memória sávszélességének növelésére (pl. a 60-as években a CDC6600, illetve az IBM System/360 Model 91). A memóriacsatornák egymástól független buszokat



8.11. ábra. Többcsatornás memóriaelérés szinkronizált csatornákkal



8.12. ábra. Többcsatornás memóriaelérés független csatornákkal

használnak, így a külön csatornákra helyezett memóriamodulok, illetve rank-ek egymással párhuzamosan képesek működni.

A csatornák kezelése szempontjából két esetet különböztetünk meg:

- *Szinkronizált csatornák* használata (8.11. ábra). Ebben az esetben az egyes memóriacsatornák tökéletes szinkronban működnek, ami csak akkor kivitelezhető, ha a memóriamodulok teljesen egyforma mérettel és időzítésekkel rendelkeznek. A memóriarendszer ekkor úgy működik, mintha egyetlen csatornánk lenne, de szélesebb adategységekkel (például két 64 bites csatorna egyenértékű egy egycsatornás, 128 bites adategységekkel dolgozó memóriával).
- *Független csatornák* használata esetén (8.12. ábra) minden csatornához külön memóriavezérlő tartozik, melyek a hozzájuk rendelt rank-eket egymástól függetlenül kezelik. Ilyen esetben a DIMM moduloknak nem kell feltétlenül egyformáknak lenniük.

A PC-k világában manapság független csatornákat használó 2 csatornás (Core i3/i5), illetve 3 csatornás (Core i7) megoldások vannak jelen, de Alpha processzorokhoz a 1995-ben készült 8 csatornás memóriaelérést támogató alaplap is.

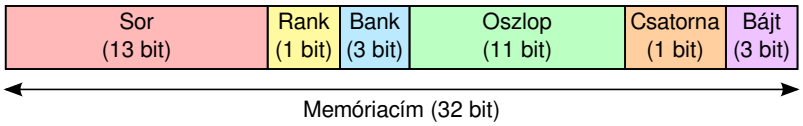
8.2.6. A memóriavezérlő feladatai

A memóriavezérlő feladata, hogy a processzor és a perifériák által kiadott memóriaműveleteket végrehajtsa, vagyis leképezze a több bank-ból és több rank-ből álló rendszermemóriára, mely esetleg még többcsatornás hozzáférést is támogat.

A processzor által kiadott címek leképzése a memóriaeszközökre

Ha egy memóriaírási/olvasási kérés érkezik a memóriavezérlőhöz, annak első dolga, hogy a vonatkozó memóriacím alapján megállapítsa, hogy a kérés melyik memóriacsatornára, azon belül melyik rank-re, melyik bank-ra, és annak melyik sorára vonatkozik. Ezt az eljárást címleképzésnek hívják.

A címleképzést úgy érdemes megvalósítani, hogy a számítógép működése során a memóriavezérlőhöz sorban befutó memóriakérés-sorozat kiszolgálása során minél ritkábban kelljen sort váltani, hiszen a sorváltás költséges művelet, egy ACTIVATE parancsot, és az ezzel járó T_{RCD} késleltetést igényel. Figyelembe véve, hogy a futó programok jellemzően egymáshoz közeli memóriacímeket adnak ki (a lokalitási elveket hamarosan, a 10.1.



8.13. ábra. Címleképzés

fejezetben tárgyaljuk), a leképezés során minél nagyobb összefüggő, sorváltást nem igénylő címtartományt célszerű kialakítani.

A címleképzés egy lehetséges módját a 8.13. ábrán, egy példán keresztül mutatjuk be. Ebben a példában 32 bites memóriacímeket használunk, és az egyszerűség kedvéért legyen 2x2GB DIMM memóriamodulunk 2 rank-kel, kétcsatornás eléréssel. A DRAM chip-ek tartalmazzanak 8 bankot, és minden bank álljon 2^{13} sorból és 2^{11} oszlopból. Az adategység legyen 64 bit (8 bájt). Az ábrán látható leképzés szerint a cím felső 13 bitje jelölik ki a sort, így 2^{19} bájt, vagyis fél megabájt összefüggő memóriaterület tartozik minden egyes sorhoz. A következő bit a rank-et határozza meg, ez esetben 1 bites, hisz minden csatornán egyetlen DIMM modul van, mely 2 rank-et tartalmaz. Az utolsó 3 bit azt határozza meg, hogy a 64 bites (=8 bájtos) adategységen belül a memóriacím melyik bájtra vonatkozik. Természetesen ettől eltérő címleképzés is elképzelhető.

Kérések átrendezése

A 8.2.2. fejezetben megismertük a DRAM bankok elérési protokollját. Láttuk, hogy ha az egymás utáni memóriakérések egy sorra vonatkoznak, akkor gyorsan kiszolgálhatók, míg ha különböző sorokba esnek, akkor hosszadalmas sorbezárás és sormegnyitás parancsokat is be kell iktatni.

A legegyszerűbb memóriavezérlők valóban érkezési sorrendben (FCFS, First Come, First Served) hajtják végre a memóriakéréseket, azonban a fejlettebbek képesek a memóriakérések átrendezésére úgy, hogy összességében a teljes memóriakérés-sorozat teljes kiszolgálási ideje a lehető legrövidebb legyen. Ennek egy módja az FR-FCS (First Ready, First Come First Served), ami előreveszi azokat a kéréseket, melyeket az aktuálisnál gyorsabban ki lehet szolgálni. Az alábbi példa bemutatja az egyébként rendkívül egyszerű ötletet:

FCFS ütemezés

Kérések:

```
(3.sor, 8. oszlop)
(1.sor, 3. oszlop)
(3.sor, 14. oszlop)
```

Parancsok:

```
ACTIVATE 3
READ 8
PRECHARGE
ACTIVATE 1
READ 3
PRECHARGE
ACTIVATE 3
READ 14
```

FR-FCFS ütemezés

Kérések:

```
(3.sor, 8. oszlop)
(3.sor, 14. oszlop)
(1.sor, 3. oszlop)
```

Parancsok:

```
ACTIVATE 3
READ 8
READ 14
PRECHARGE
ACTIVATE 1
READ 3
```

Mint látható, a második és harmadik kérés felcserélésével a memóriavezérlő meg tudott spórolni egy sorlezárás és sormegnyitás műveletet.

Gazdálkodás a nyitott sorokkal

A sormegnyitás műveletek mellett a sorlezárásnak (PRECHARGE) is számottevő késleltetése van (T_{RP}). A memóriavezérlő a sormegnyitás és lezárás műveleteket a késések átrendezésével tudja minimalizálni, de mit tegyen, ha épp elfogynak a memóriakérések (mert mindet kiszolgálta). Kétféleképpen cselekedhet:

- Nyitva hagyja az aktuális sort. Ez a taktika akkor előnyös, ha a következő befutó memóriakérés ugyanarra a sorra vonatkozik. Mivel a vezérlő nyitva hagyta, ACTIVATE nélkül, kis késleltetéssel ki lehet szolgálni az

új igényt. Ha azonban a következő kérés egy másik sorra vonatkozik, akkor annak megnyitása előtt még a nyitva tartott sor lezárására (PRECHARGE) is időt kell vesztegetni, ami növeli a késleltetést.

- Lezárja az aktuális sort. Ha a következő kérés ugyanerre a sorra vonatkozik, akkor időt veszít, mert a feleslegesen lezárt sort újra meg kell nyitni (ACTIVATE). Amennyiben a következő kérés másik sorra hivatkozik, akkor a taktika bevált, hiszen csak egy sormegnyitás parancsot kell kivárni, sorlezárást nem.
- Adaptív taktikát választ (APM, Active Page Management). A memóriavezérlő a memóriaelérések múltbeli mintázata alapján megpróbálja megtippelni, hogy a következő kérés ugyanarra a sorra fog-e vonatkozni, és az aktív sor lezárása felől spekulatív módon dönt. Az újabb generációs Intel Core i7 processzorok beépített memóriavezérlője ezt az eljárást használja, engedélyezését, finomhangolását bizonyos alaplapok BIOS-a lehetővé is teszi (”Adaptive Page Closing” opció).

Frissítések ütemezése

A DRAM chip-ek frissítéséről is a memóriavezérlőnek kell gondoskodnia. A frissítéshez rendszeres időközönként ki kell adnia a REFRESH parancsot, melynek hatására a kiválasztott rank DRAM chip-jeiben a soron következő sorok cellái kiolvasásra, majd visszaírásra kerülnek. A frissítés megkönnyítésére a DRAM chip-ek tartalmaznak egy számlálót, ami a következő frissítésre kijelölt sorra mutat, és amely minden REFRESH parancsra automatikusan egyel nő. A frissítés idejére a DRAM chip természetesen nem tud hasznos adat olvasási/írási műveletet végezni, tehát ez kieső idő. Annak érdekében, hogy ez a kieső idő ne legyen túl hosszú, a memóriavezérlő nem egyszerre, REFRESH parancsok özönével frissíti az összes sort, hanem időben elosztva, egyenletesen: végrehajt pár parancsot, frissít egy sort, megint végrehajt pár parancsot, frissíti a következő sort, és így tovább.

8.2.7. A DRAM technológiák evolúciója

A DRAM chip-ek nagyjából 15 évvel ezelőttig aszinkron interfésszel rendelkeztek. Ez azt jelentette, hogy nem volt órajel, a DRAM parancsok mindegyikéhez tartozott egy láb, melyen egy felfutó éllel kellett a kívánt műveletet kezdeményezni. Ezután áttértek a szinkron, órajel alapú működésre (SDRAM, Synchronous DRAM), amely időtállónak bizonyult, napjainkban is SDRAM-ot használunk. Ezen a ponton, az órajel megjelenésénél kapcsolódunk be a DRAM technológiák evolúciójának áttekintésébe.

SDR SDRAM

A szinkron DRAM-ok első generációjának képviselőjét SDR SDRAM (Single Data Rate SDRAM) névvel illették. Az SDR SDRAM már a kezdetekkor is több (2 vagy 4) bankot használt. A burst adatátvitelt is támogatta, az egyszerre átvinni kívánt adatok számát (a burst hosszát) 1 és 8 között lehetett beállítani (mivel 64 bites adategységeket használt, ez legfeljebb 64 bájt egymás utáni elküldését jelenti).

DDR SDRAM

A DDR (Double Data Rate) SDRAM azonos órajel esetén az SDR SDRAM-hoz képest kétszer akkora átviteli sebességre képes. Ezt úgy érték el, hogy az órajelnek mind a felfutó, mind a lefutó élénél végez adatátvitelt. Míg az SDR SDRAM minimális burst hossza 1, a DDR SDRAM-é 2.

A szabványos elnevezés már a órajel frekvenciájának a dupláját használja, így a DDR-200 SDRAM valójában 100 MHz-en működik. (A DDR-200 jelölés tulajdonképpen arra utal, hogy a memória olyan gyors, mintha 200 MHz-es SDR SDRAM lenne.) Például egy 200 MHz-en működő 64 bit széles DDR-400-at PC-3200 néven lehet megvásárolni, és maximum 3200 MB/s átviteli sebességre képes (hiszen a 64 bit=8 bájt, és a 400 MHz-es adatátvitel miatt 400 millió adategységet tud másodpercenként átvinni).

DDR2 SDRAM

A DDR2 SDRAM-tól kezdve kétféle órajelet különböztetünk meg: a DRAM chip a *belső órajelet* használja a parancsok átvételére és a belső működés vezérlésére, az adatok átvitelét azonban az ennél kétszer nagyobb frekvenciájú *külső órajelen* végzi. A DDR2 SDRAM így külső órajelciklusonként visz át 2 adategységet (1-et a felfutó, 1-et a lefutó élen – végtére is DDR megoldásról van szó), tehát a belső órajel minden periódusa alatt 4-et. Ennek megfelelően míg a DDR-nél 2 volt a minimális burst hossz, itt már 4 lett. A bankok száma is megduplázódott, egy DDR2 SDRAM chip 4 vagy 8 bankot tartalmazhat.

Sajnos a szabványos jelölés értelmezése nem egyszerű. Egy DDR2-800 névvel ellátott SDRAM-nak a belső órajele 200 MHz, a külső 400 MHz, a ”800” úgy jön ki, hogy a külső órajel fel- és lefutó élénél is van adatátvitel, így pont olyan gyors, mint egy 800 MHz-es SDR SDRAM. A 64 bites adatszélesség miatt pedig PC2-6400 modulként adják el, és maximum 6400 MB/s átviteli sebességre képes (hiszen a 64 bit=8 bájt, és a 800 MHz-es adatátvitel miatt 800 millió adategységet tud másodpercenként átvinni).

DDR3 SDRAM

A DDR3 SDRAM külső órajele már négyszer akkora, mint a belső, és mivel DDR, minden külső órajelciklusban két adatot visz át (fel- és lefutó élre), tehát a belső órajel minden periódusa alatt 8-at. A DDR3 memóriák 8 bankot használnak, és az átviteli burst hosszát 8 adategységben rögzítették.

Egy DDR3-1600 névvel jellemzett DDR3 SDRAM-nak a belső órajele 200MHz, a külső 800MHz, az ”1600” úgy jön ki, hogy a külső órajel fel- és lefutó élénél is van adatátvitel. A 64 bites szélesség miatt pedig PC3-12800 modulként adják el, mivel legfeljebb 12800 MB/s átviteli sebességre képes.

DDR4 SDRAM

A DDR3 utódja a sorozat legfiatalabb tagja, a DDR4 SDRAM. A DDR4 esetében felhagytak a külső órajel növelésével, a DDR3-hoz hasonlóan 8 adategységet képes átvinni a belső órajel egyetlen ciklusa alatt. Ennek megfelelően a burst hossz is maradt a rögzített 8 adategység. A teljesítményt úgy fokozták, hogy a bankok száma 16-ra nőtt, és az órajel frekvenciája is nagyobb, mint a DDR3 esetében volt. A nagyobb órajel miatt paritásbittel védik a parancs- és címvezetékeket, és ellenőrző összeggel (CRC) az adatvezetékeket.

	SDR	**DDR**	**DDR2**	**DDR3**	**DDR4**
Belső órajel	66-133 MHz	133-200 MHz	100-200 MHz	100-200 MHz	200-533 MHz
Adatok/belső órajel	1	2	4	8	8
Adatátviteli seb. (MB/s)	528-1064	2128-3200	3200-6400	6400-12800	12800-34112
Burst hossz	1-8	2-8	4-8	8	8
Bankok száma	2-4	2-4	4-8	8	16
Feszültség	3.3V	2.5V	1.8V	1.5V	1.05-0.2V

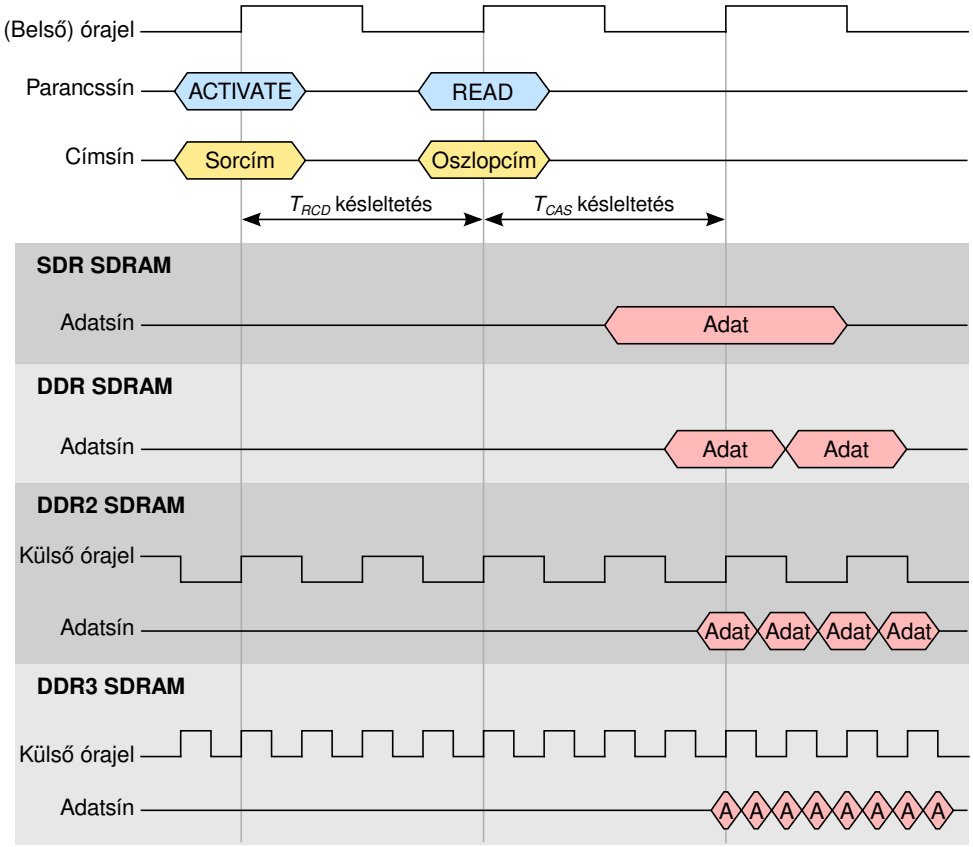
8.1. táblázat. SDRAM technológiák összehasonlítása

GDDR5 SDRAM

A GDDR5 SDRAM technológiát kifejezetten nagy sávszélesség igényt támasztó célokra fejlesztették ki, jellemzően magasabb árfekvésű grafikus kártyákban használják. Egy GDDR5 alapú rendszer a megismert szervezésnél sokkal egyszerűbb: nincsenek benne rank-ek, és modulok sem. Minden egyes DRAM chip közvetlenül a vezérlőhöz van kötve, grafikus kártya esetén a grafikus processzorhoz. Az x4, x8, ill. x16 adatszélességet használó DDR3 memória chip-ekkel ellentétben a GDDR5 chip-ek 32 bitesek, így 8 chip segítségével máris 256 bites adatszélességet kapunk. A bankok felépítése is némileg eltérő az eddig megszokottól: 16 bank van, mindegyikben egyszerre két sor lehet nyitva, és egy írás valamint egy olvasás művelet egyidejű kiszolgálására is képes.

Míg a DDR2-től kezdve külön külső és belső órajelről beszéltünk, a GDDR5 esetén már háromféle órajelet kell megkülönböztetni. Az első a DRAM cellák órajele, legyen például 375 MHz (nagyjából a belső órajel megfelelője). A GDDR5 chip-eknek azonban ezen órajel négyszeresével lehet parancsokat adni, tehát a parancs órajel a példánkban 1500 MHz. Az adatok átvitele (a külső órajel) még ennek is a négyszerese, vagyis 6 GHz (!), ami 256 bites adatszélesség mellett 192 GB/s adatátviteli sebességet jelent.

Az impresszív átviteli sebességnek természetesen ára van: az előállítási költség 4-5-szörös, a fogyasztás nagyobb, és a chip-ek fizikai távolsága a memóriavezérlőtől igencsak korlátos. Jelenleg a legnagyobb GDDR5 alapú memóriát használó számítógép, a Sony Playstation 4, 8 GB memóriát tartalmaz, ezt megelőzően ennyit ebből a fajta memóriából még nem használtak.



8.14. ábra. SDRAM technológiák

Összegzés

A tárgyalt memóriatechnológiák legfőbb tulajdonságait a 8.1 táblázat foglalja össze. Mint látható, a belső órajel a vizsgálatunkban felölelt 5 generáció alatt nem változott számottevően. A belső órajellel van összefüggésben a memóriaműveletek *késleltetése* (a cím ráadásától a megfelelő adat megjelenéséig eltelt idő) is, mely évtizedek óta néhányszor 10 ns-os nagyságrendbe esik. A többcsatornás memóriavezérlők alkalmazása, a DDR technológia és a külső órajel növelése azonban az *adatátviteli sebesség* jelentős növekedését tette lehetővé. Ugyanezek a következtetések vonhatók le a 8.14 ábráról is, amely abból a szempontból még idealizált is, hogy helytakarékossági okokból mind a T_{RCD}, mind a T_{CAS} értéke 1 (pontosabban csak SDR és DDR esetben 1, DDR2 RAM-oknál 2, DDR3-nál pedig 4 – mivel ezeket a késleltetéseket nem belső, hanem külső órajelben mérik).

Napjainkban tehát a rendszermemória lényegesen több memóriakérést tud kiszolgálni, mint akár csak 10 évvel ezelőtt, de a kérések kiszolgálásának a késleltetése nagyságrendileg ugyanaz maradt. Ebben a kérdésben fizikai okokból várhatóan a DRAM cellákra épülő jövőbeli memóriatechnológiák sem hoznak majd változást.

9. fejezet

A virtuális memória

A programozó számára a memória elérése egyszerű feladat. A processzor utasításkészlete memóriakezelő utasításokat kínál számára, melyek segítségével a memória adott címére tud hivatkozni, onnan adatokat tud beolvasni, illetve oda adatokat tud írni. Alapesetben, ha nincs virtuális memória, a programozó által kiadott cím egyenesen a rendszermemória felé továbbítódik, az `R0 ← MEM[42]` utasítás a rendszermemória 42-es címén lévő adatot fogja visszaadni.

Ha virtuális memóriát használ a processzor, akkor nem ez a helyzet. A programozó egy képzeletbeli, virtuális memórián végez műveletet, nem pedig a valós, fizikai memórián. Ez a nyilvánvaló bonyolítás, mely, mint látni fogjuk, még lassítja is a memóriaeléréseket, számos előnnyel jár. A virtuális memória nem létszükséglet, mégis olyan alapvető fontosságú, hogy minden mai, modern, multi-tasking képességekkel rendelkező operációs rendszer megköveteli a processzortól ezt a képességet.

9.1. Alapelvek

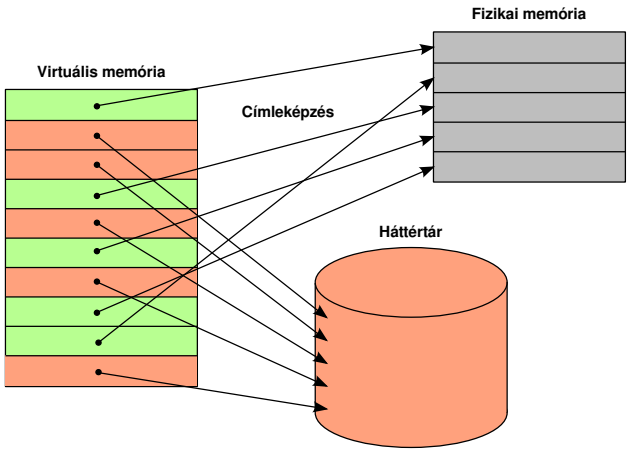
Multi-tasking környezetben egy processzor több programot is futtathat egyidejűleg, melyek memóriaigénye dinamikusan változik, a programok által használatban lévő összes memória mérete hosszabb-rövidebb időre akár túl is lépheti a fizikailag rendelkezésre állót. A szoftverfejlesztés során nagy könnyebbség lenne, ha a programoknak nem kellene tudomást vennie a fizikai memória végességéről, illetve arról, hogy a rendelkezésre álló szabad memória dinamikusan változik a többi, párhuzamosan futó program memóriaigényeinek függvényében. Előnyös lenne, ha minden program számára ”virtuálisan” a processzor teljes címtartománya rendelkezésre állna, az pedig legyen a processzor és az operációs rendszer ügye, hogy a futó programok ”virtuális” memóriáját hogy kell a szűkös fizikai memóriába bezsúfolni, és hogy mit kell csinálni ha az éppen nem fér be. Ezt a memóriakezelési elvet nevezik *virtuális tárkezelés*nek, mely 1962-ben bukkant fel először, és a 70-es évekre érte el azt a formáját, melyet a mai virtuális memória-kezelésre képes processzorok is birtokolnak.

A processzoron futó programok tehát *virtuális címek*et használnak (nem is használhatnak mást), amivel a processzor teljes címzési képességét kihasználhatják. Például egy 32 bites rendszerben a programok 2^{32} bájt, azaz 4 GB memóriát használhatnak akkor is, ha ennél kevesebb van fizikailag a számítógépben, vagy a többi futó program mellett ennél kevesebb jut. A processzor lábain és a buszon azonban már *fizikai cím* jelenik meg. Tehát a processzornak a program futása közben kiadott minden egyes (virtuális) memóriacímet át kell alakítania fizikai címmé, hiszen a fizikai memóriát csak az alapján szólíthatja meg. Ezt a folyamatot *címfordítás*nak (address translation) nevezzük, amit a processzorban található hardver, az MMU (memory management unit) végez.

Az nem lenne gazdaságos, ha az MMU minden egyes virtuális címre, vagyis bájtonként nyilvántartaná, hogy az a fizikai memóriában éppen hol van. Ezért a virtuális és a fizikai memóriát nagyobb darabokra, adategységekre osztjuk, és a közöttük lévő kapcsolatot ezekre a nagyobb adategységekre tartjuk nyilván. Ezeket az adategységeket *lap*oknak hívjuk, ha méretük fix, ebben a fejezetben pedig kizárólag a lapszervezésű virtuális memóriakezeléssel foglalkozunk, mivel az utóbbi évtizedekben használatos processzorok szinte kizárólag ezt alkalmazzák.

A nagyságrendek érzékeltetése kedvéért nézzük meg egy x86-64 architektúrájú, 64 bites processzor logikai és fizikai címtartományát. Az x86-64 jelenlegi implementációi 48 bites virtuális címeket használnak, ami 2^{48} bájt = 256 terabyte memória címzését engedi meg, de az architektúra későbbi bővítést is lehetővé tesz egészen 2^{64} bájt = 16 millió terabyte-ig. Ha a futó programoknak ezt a teljes tartományt a rendelkezésére bocsátjuk, felmerül a kérdés, hogy hol tároljuk ennek a hatalmas címtartománynak a fizikai memóriából kilógó részét? A válasz: ez az operációs rendszer dolga, ami jellemzően a fizikai memóriánál lényegesen nagyobb (és sokkal lassabb) háttértáron

fogja elhelyezni a fizikai memóriába be nem férő adatokat (9.1. ábra).



9.1. ábra. Virtuális tárkezelés elve

9.2. Virtuális és fizikai címek, a címfordítás folyamata

Lapszervezésű virtuális memória-kezelés esetén az N_V bites virtuális címtartományt fix, egyforma méretű lapokra particionáljuk, az N_F bites fizikai memóriát pedig ugyanekkora méretű *keretekre* (frame) osztjuk. Ha a lapok mérete 2 hatványa (L bites), akkor mind a virtuális, mind a fizikai címeket 2 részre bonthatjuk: az alsó L bit lesz a lapon belüli bájtpozíció, a fennmaradó felső bitek (a virtuális címek esetén $N_V - L$, fizikai címeknél $N_F - L$ bit) pedig a lapot, illetve a keretet azonosítja.

Például egy 32 bites rendszerben, mely 1 GB ($= 2^{30}$ bájt) méretű fizikai memóriát tartalmaz és melynek virtuális memória-kezelése 4 kB-os ($= 2^{12}$ bájt) lapokat használ, ezek a paraméterek a következők:

- A program által használt memóriacímek $N_V = 32$ bitesek, melynek alsó $L = 12$ bitje a lapon belüli bájtpozíció, a maradék $N_V - L = 20$ bit pedig a lapok azonosítására szolgál. Ebből meg is kapjuk, hogy a virtuális memória $2^{20} = 1048576$ lapból áll.
- A fizikai memória címzéséhez $N_F = 30$ bit szükséges. Ennek alsó $L = 12$ bitje a lapon belüli pozíció, a fennmaradó $N_F - L = 18$ bit pedig a keretek sorszáma. Ebben a példában tehát a fizikai memória $2^{18} = 262144$ keretből áll, ennyi lapot tud tárolni.

Az összerendelést, ami megadja, hogy melyik lap melyik keretre lett elhelyezve (illetve, hogy melyik keret melyik lapot tartalmazza), a *laptábla* tartja nyilván. A laptáblából lehet csupán egyetlen, globális, de lehet minden folyamatnak (futó programnak, processznek) külön-külön is laptáblája, ez megvalósítás függő.

A laptábla laptáblabejegyzésekből áll, minden bejegyzés egy lap-keret összerendelést ír le, melyből többek között a következő információknak kell kiderülnie:

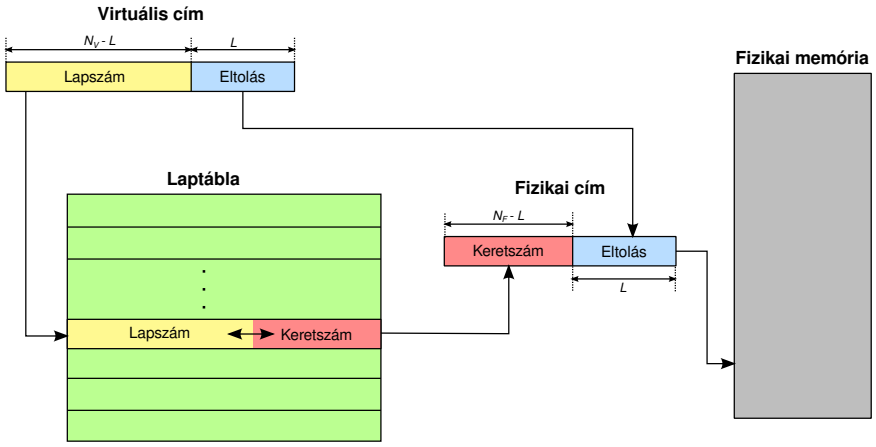
- A lap sorszáma
- A keret sorszáma
- A lapot birtokló folyamat azonosítója
- A lapon végezhető megengedett műveletek (írható/ olvasható/ végrehajtható)
- Vezérlő bitek:
  - ”Valid” bit: ha =1, akkor ez egy érvényes összerendelés. Ha =0, akkor a lap nincs bent a fizikai memóriában, ilyenkor a laptábla bejegyzésben benne kell lennie a lapot tároló diszk szektor számának, vagy egyéb, a lap háttértáron való helyére vonatkozó információnak.
  - ”Dirty” bit: ha =1, akkor erre a lapra történt írási művelet, mióta a fizikai memóriába került. Ezt a bitet a CPU tartja karban (automatikusan), ezzel az operációs rendszer munkáját segítve. Dirty=0 lapokat nem kell a diszkre kiírni, ha kiszorulnak a fizikai memóriából (hiszen a tartalmuk a fizikai memóriába helyezés óta változatlan, a diszk-en tárolt változat tehát továbbra is aktuális).

– ”Accessed” bit: ha =1, akkor volt ”mostanában” hivatkozás erre a lapra. Ezt is a CPU tartja karban. Ez a bit segít az operációs rendszernek, amikor el kell döntenie, hogy melyik lapot dobja ki a fizikai memóriából, ha egy új lapnak nincs helye. Az accessed=0 lapokból fog válogatni.

A laptábla – megvalósítástól függően – vagy a lapok, vagy a keretek számával arányos, de mindenesetre elég nagy ahhoz, hogy magát a laptáblát is a rendszermemóriában kelljen tárolni.

A címfordítás menete a következő. Memória hozzáférés esetén a processzor a hivatkozott címet virtuális címnek tekinti. Leválasztja a címről a lapszámot, majd megkeresi a laphoz tartozó laptábla bejegyzést. Ha a bejegyzés szerint a lap a fizikai memóriában van (valid=1), akkor a keret számát a bejegyzésből kiolvasva, azt a cím eltolás mezőjével kibővítve megkapja a fizikai címet. Valid=0 esetén meghívja az operációs rendszer erre a célra beregisztrált szubrutinját, ami a laptábla bejegyzésből kiolvassa, hogy a kérdéses lap hol található a háttértáron, azt beolvassa, elhelyezi a fizikai memóriában, majd aktualizálja a laptábla bejegyzést. Előfordulhat, hogy az operációs rendszernek nincs hova betölteni a lapot, mert nincs szabad keret. Ilyenkor kiválaszt egyet, ha a lap „dirty”, kiírja a háttértárra (közben frissítve a vonatkozó laptábla bejegyzést), majd az így felszabaduló keretbe kerül a hivatkozott lap (9.2. ábra). Célszerű accessed=0, azaz rég használt lapot választani.

A laphiba lekezelése után a processzor a laphibát okozó, félbeszakadt utasítás végrehajtását folytatja, így a futó program semmit nem vesz észre a címfordításból, a felhasználó pedig legfeljebb a diszkműveletet és a rövid fennakadást észleli, amivel a lapok diszkről memóriába mozgatása jár.



9.2. ábra. Címfordítás, ha a keresett lap a fizikai memóriában van

9.3. A TLB (Translation Lookaside Buffer)

A címfordítás folyamatából talán már ki is olvasható a virtuális tárkezelés legnagyobb problémája. Nevezetesen, ha egy program memóriaműveletet szeretne végezni, *a processzornak két körben kell a rendszermemóriához fordulnia*:

1. először a hivatkozott laphoz tartozó laptábla bejegyzést kell kiolvasnia (hiszen a laptábla is ott van),

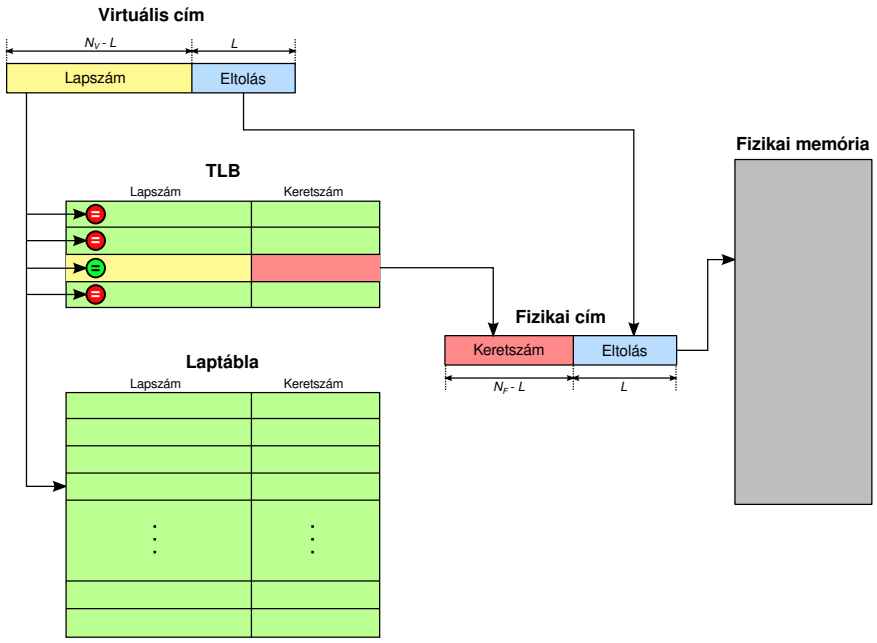
2. majd ez alapján tudja a fizikai címet kiszámolni és annak ismeretében a kívánt memóriaműveletet elvégezni.

Sajnos bizonyos (később részletezett) okokból akár több memóriához fordulás is szükséges lehet a megfelelő laptábla bejegyzés megtalálásához, ami tovább lassítja a folyamatot.

Szerencsére egy egyszerű megoldással ki lehet használni, hogy a programok memóriahozzáférési mintázata többnyire szabályos. Nem jellemző, hogy egy program véletlenszerűen címezget össze-vissza a teljes címtartományban. Sokkal inkább jellemző a lokális viselkedés, vagyis pár lapon sokat dolgozik (pl. ahol épp az aktuálisan használt adatszerkezetek vannak), majd ezután pár másik lapot kezd el intenzíven használni, stb.

A gyakorlat is azt mutatja, hogy ha a gyakran használt lapokhoz tartozó laptábla bejegyzéseket a processzorban, egy erre a célra kialakított speciális, gyors elérésű cache memóriában tároljuk, akkor a címfordítások nagy részét a processzor ”házon belül” meg tudja oldani, a lassú rendszermemóriában lévő laptábla kiolvasása nélkül. Ezt a speciális, gyors elérésű tárolót TLB-nek (translation lookaside buffer) nevezzük.

A TLB tehát a gyakran használt lapok lapszám ↔ keretszám összerendeléseit tárolja, A TLB-vel segített címfordítási folyamat esetén a processzor először a TLB-ben keresi a hivatkozott laphoz tartozó keretet, és ha nincs találat (ezt a helyzetet TLB hibának hívjuk), akkor kénytelen azt a memóriából kiolvasni (9.3. ábra). A TLB hatékony működése, azaz az alacsony TLB hibaarány kulcsfontosságú a virtuális tárkezelés teljesítménye szempontjából. Minél nagyobb a TLB-ben tárolt (cache-elt) laptábla bejegyzések által lefedett címtartomány - a TLB lefedettség - annál ritkábban kell a memóriában lévő laptáblához fordulni.



9.3. ábra. Címfordítás, ha a keresett lap a TLB-ben van

A TLB-t tartalom szerint címezhető memóriával valósítják meg, vagyis a keresés a TLB bejegyzései között egyetlen lépésben megtörténik. A tartalom szerint címezhető memóriáknak azonban vannak hátrányos tulajdonságai: nagy felületet igényel a félvezetőn, és nagy a fogyasztása. Ez is az oka annak, hogy a TLB lefedettség nem tud lépést tartani a használatos memóriák méretének növekedésével. 1990-ben a MIPS R2000-nek 64 bejegyzéses TLB-je volt, a 2006-ban megjelent Intel Core processzorban pedig 256 bejegyzéses, az arány jóval kisebb az operatív memóriák méretének növekedési rátájánál. Ebből adódóan a relatíve kis TLB-vel jól kell gazdálkodni, tartalmának hatékony menedzselésén sok múlik. Ezt a menedzsmentet (vagyis hogy mely bejegyzések legyenek a TLB-ben, mikor hozzunk be újat, és ezzel melyik régi bejegyzést írjunk felül, stb.) végezheti kizárólagosan a hardver vagy a szoftver is (architektúrafüggő). A TLB tipikus paramétereit a 9.1. táblázat foglalja össze ([32]).

TLB méret:	16-512 bejegyzés
Találat ideje:	0.5 - 1 órajel
TLB hiba esetén a címfordítás ideje:	10 - 100 órajel
TLB hibaarány:	0.01% – 1%

9.1. táblázat. Tipikus TLB paraméterek 2011-ben

9.4. Laptábla implementációk

A virtuális memóriakezelés hatékony megvalósításához olyan adatszerkezetekre és algoritmusokra van szükség, melyek lehetővé teszik a gyors címfordítást. A címfordítás fentebb megismert működése alapján a következő 3

kritériumot támasztjuk a laptáblával szemben:

- Legyen gyors a virtuális cím (lapszám) szerinti keresés
- A laptáblához a lehető legkevesebbszer kelljen fordulni egy címfordítás alkalmával (hiszen az minden alkalommal egy memóriaműveletet jelent, ami órajelciklusokban mérve igen költséges)
- A laptábla legyen a lehető legkisebb (hiszen a nagy laptábla jelentősen csökkentheti a fizikai memória felhasználói programok számára fennmaradó részét)

Ezen követelmények alapján, a valós termékekben való megjelenésük sorrendjében áttekintünk 4 megoldást.

9.4.1. Egyszintű laptábla

Az egyszintű laptábla egy fizikai memóriában folytonosan tárolt tömb, melynek i. eleme az i. laphoz tartozó laptábla bejegyzés. Lássuk, hogy az imént megfogalmazott követelmények szempontjából hogyan teljesít ez a megoldás.

A keresés (1. kritérium) *gyors*, hiszen a processzor a keresett laptábla bejegyzés memóriabeli helyét könnyedén kiszámolja:

$$i\text{. laphoz tartozó bejegyzés címe} = \text{laptábla kezdőcím} + i \cdot \text{egy bejegyzés mérete}$$

Következésképp a bejegyzés kiolvasása csupán egyetlen memóriaműveletbe kerül, hiszen a CPU pontosan tudja, hogy mely címen van a bejegyzés, vagyis a memóriaműveletek számában (2. kritérium) is jó ez az adatszerkezet.

A laptábla mérete (3. kritérium) két tényezőből áll: mekkora egy laptábla bejegyzés, ill. hogy hány bejegyzésből áll a laptábla. Az egyszintű laptábla esetén a laptábla bejegyzés nagyon kicsi is lehet, hiszen:

- a lapszám $\leftrightarrow$ keretszám összerendelés megadásához nem kell tárolni a lapszámot, hiszen az az index, így az i. bejegyzésbe elég beírni az i. lapot tároló keretet;
- valid=0 esetén (ha a lap nincs bent a fizikai memóriában), a keretszám tárolására szolgáló mezővel az operációs rendszer gazdálkodhat. Valid=0 esetén egy lap vagy a háttértáron lehet, vagy sehol (használaton kívül van), előbbi esetben a lap merevlemezen való elhelyezésével kapcsolatos információ is kerülhet a keretszám mezőbe.

Összességében egy laptábla bejegyzés csak a vezérlő biteket és egyetlen cím jellegű információt tartalmaz (keretszám vagy háttértárra vonatkozó cím). Ez 32 bites rendszereken jellemzően 4 bájtba, 64 bites rendszerekben 8 bájtba belefér.

Az egyszintű laptáblák hátulütője, hogy a teljes laptáblának benne kell lennie a memóriában. A nagyságrendek érzékeltetése kedvéért számoljuk ki a laptábla méretét 32 bites címek, 4 bájtos bejegyzések és 4 kB méretű lapok mellett! A 4 kB ($= 2^{12}$ bájt) méretű lapokból megkapjuk, hogy $L = 12$. Tehát $N_V - L = 20$ bit marad a lapok számozására, vagyis $2^{20} = 1048576$ lapunk van. Mindegyikhez 4 bájtos bejegyzés tartozik, vagyis a laptábla 4 MB helyfoglalást jelent, ami bizonyos esetben akár túl nagy is lehet. 64 bites címekkel, 8 bájtos bejegyzésekkel és továbbra is 4 kB-os lapokkal pedig a laptábla $8 \cdot 2^{52}$ bájt (32 petabájt) méretű, ekkora memória 2011-ben nemhogy kereskedelmi forgalomban, de a földkerekségen sincs. Az egyszintű laptábla ennélfogva inkább elméleti, mint gyakorlati lehetőség, modern, 64 bites rendszerekben nem jöhet szóba a használata.

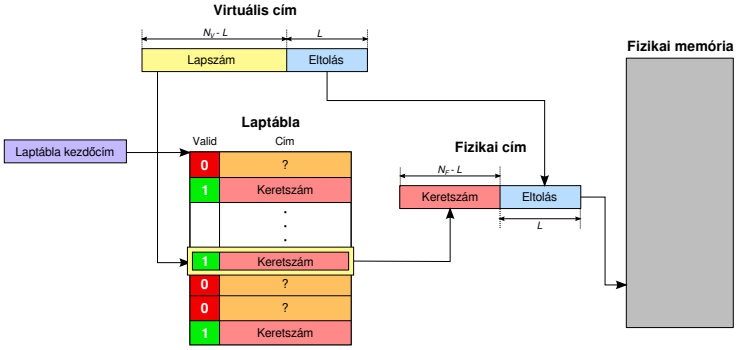
A címfordításhoz szükséges bejegyzés megtalálása és a fizikai cím összeállítása a 9.4. ábrán látható módon történik. Az i. lap helyét a laptábla i. bejegyzés adja meg, melyhez az eltolás hozzáfűzése után megkapjuk a fizikai címet. A laptábla kezdőcímét sok processzor egy speciális regiszterben tárolja, ahová az operációs rendszer a rendszerindításkor beírja, hogy hová képzeli a laptábla elhelyezését.

9.4.2. Hierarchikus laptábla

Az egyszintű laptábla problémája, hogy túl nagy, és ráadásul a teljes laptáblának a memóriában kell lennie.

Ezen a nehézségen úgy lehet felülkerekedni, hogy magát a laptáblát is a virtuális tárkezelés alá vonjuk, a ritkán használt részeit a diszkre lehet menteni, a soha nem használt részeit pedig el sem kell tárolni.

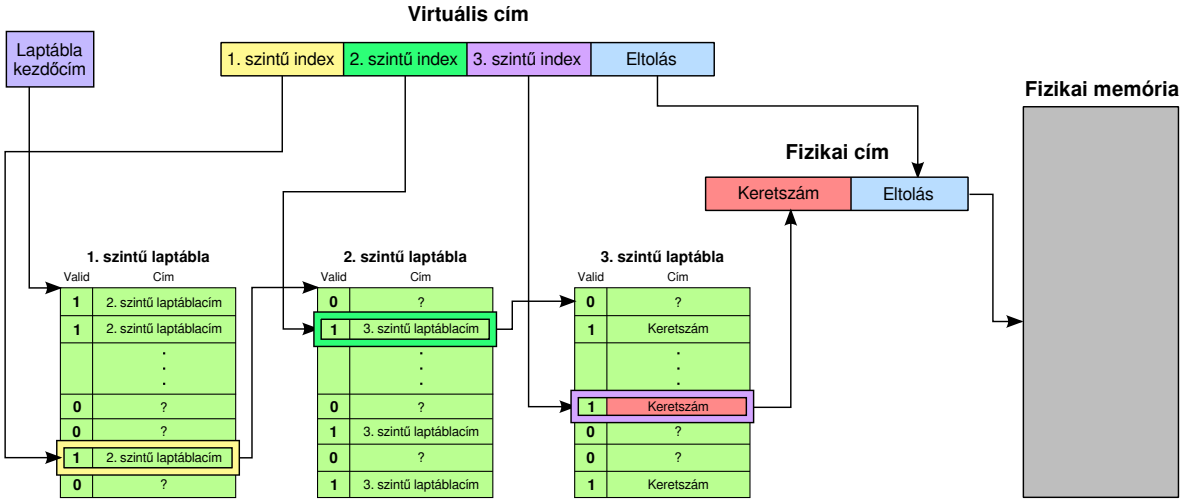
Ennek érdekében a laptáblát akkora méretű darabokra kell bontani, hogy ezek a darabok elférjenek egy lapon. Így lehetővé válik, hogy a laptábla nem használt, lapnyi méretű darabjait ugyanúgy a háttértáron tartsuk, mint bármelyik más felhasználói lapot. Mivel megszűnik a laptábla folytonossága, további lapokra van szükség, melyek mutatókat tartalmaznak a laptábla címfordítást leíró lapjaira. Ha ezek a mutatók is olyan sokan vannak,



9.4. ábra. Címfordítás egyszintű laptáblával

hogy nem férnek el egy lapon, újabb lap(ok)ra van szükség, melyek mutatókat tartalmaznak ezekre a mutatókat tartalmazó lapokra, és így tovább. Egy fa adatszerkezetet kapunk, melyben minden csomópont egy lapnyi bejegyzés tartalmaz. A levél csomópontokban a bejegyzések címfordítási információt, a köztes csomópontok pedig mutatókat tartalmaznak a gyerek csomópontokra.

A címfordításhoz szükséges bejegyzést a fa top-down (gyökértől gyerekig) bejárásával kapjuk meg, a 9.5. ábrának megfelelően.

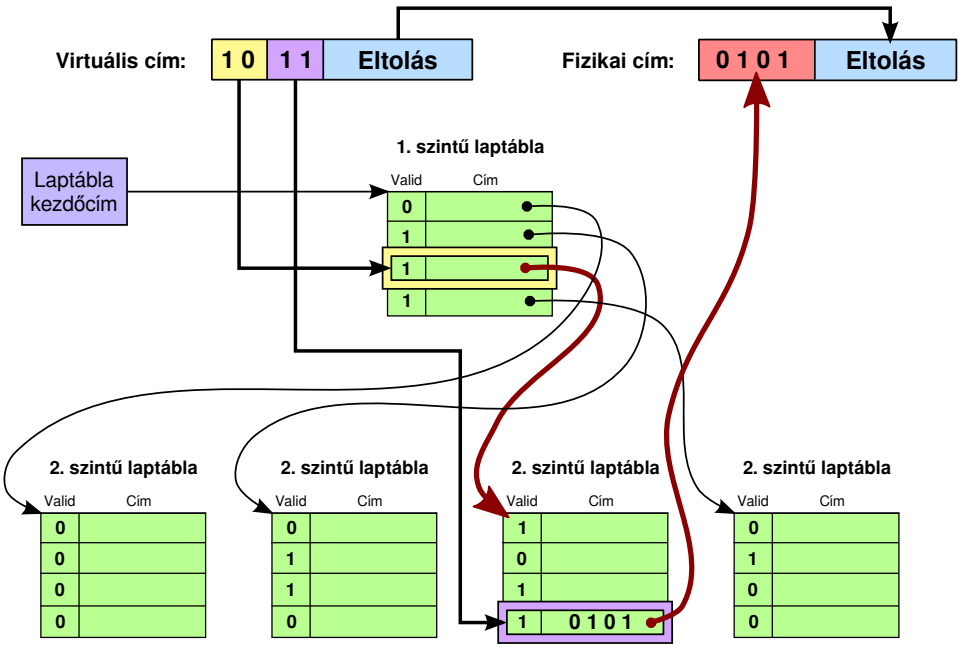


9.5. ábra. Címfordítás, háromszintű laptáblával

A hierarchikus laptábla lehetővé teszi, hogy ne kelljen a teljes laptáblát a memóriában tartani, csak annak gyakran használt részeit. Ezért az előnyért azonban nagy árat fizetünk: a címfordításhoz szükséges bejegyzés megtalálásához be kell járni a fát, annyi lépés és így annyi ”lassú” memóriaművelet szükséges, amilyen mély a laptábla hierarchia.

Kétszintű laptáblán alapuló címfordításra mutat egy konkrét példát a 9.6. ábra. A virtuális cím lapszám és eltolás mezőkre tagolódik, a példában a lapszám éppen ”1011”. A 4 bites lapszám első két bitje indexeli az első szintű laptáblát. Az első szintű laptábla kezdetét egy speciális regiszter tárolja. Az első két bit ”10”, vagyis a táblázat 2-es bejegyzését kell kiolvasni. A bejegyzés tartalmaz egy mutatót, ami megmutatja, hogy az ”10”-val kezdődő címek fordításához melyik másodszintű laptáblát kell használni. A bejegyzésben ”Valid=1” olvasható, ami jelen esetben azt jelenti, hogy ez a másodszintű laptábla a memóriában van, és nem a háttértáron. Ezután ebből a másodszintű laptáblából kiolvassuk a lapszám alsó 2 bitje (=”11”) által adott bejegyzést, jelen esetben a harmadikat. Ebben a bejegyzésben Valid=1-et találunk (a lap a fizikai memóriában van), valamint a laphoz tartozó keret számát (=”0101”). A keretszám és az eltolás egymás után illesztve megadja a virtuális címhez tartozó fizikai címet. A hierarchikus laptáblák előnye, hogy nem kell a teljes laptáblát a fizikai memóriában tárolni. Jelen

eseteben a ”00”-val és az ”11”-gyel kezdődő lapokhoz tartozó másodszintű laptáblák a háttértáron vannak (a hozzájuk tartozó első szintű laptáblabeli bejegyzésben Valid=0). A ”00” kezdetű lapok esetén ez érthető, hiszen a másodszintű laptábla alapján csupa nem használt, háttértáron lévő lapokról van szó, de az ”11” kezdetű lapok többségével is ugyanez a helyzet.



9.6. ábra. Példa kétszintű laptáblán alapuló címfordításra

Vegyük elő ismét az egyszintű laptáblánál bevezetett példát, tehát 32 bites címeink, 4 bájtos bejegyzések és 4 kB méretű lapjaink vannak. Ekkor a virtuális cím alsó 12 bitje lesz az eltolás, a fennmaradó 20 bit pedig a lap sorszáma. Egy 4 kB méretű lapba 1024 ($= 2^{10}$) darab laptábla bejegyzés fér el, vagyis a 2^{20} darab lap leírója 2^{10} lapot foglal. Ezek elhelyezkedését (a laptábla lapok fizikai memóriabeli illetve a háttértárbeli címét) pedig 2^{10}, egy újabb lapon elhelyezett bejegyzés írja le, ami pont meg is tölti a 4 kB méretű lapot. Ezt a felsőbb szintű lapot tehát mindig a fizikai memóriában tartjuk, az alacsonyabb szintű lapokat pedig, melyek a tényleges címfordítási információkat tartalmazzák, igény szerint tárolhatjuk vagy a fizikai memóriában, vagy a háttértáron.

Annak érdekében, hogy a hierarchikus laptáblák problémáját megértsük, számoljuk ki a 64 bites rendszerekben szükséges laptábla hierarchia szintjeinek a számát. A 64 bites címből legyen megint 12 bit az eltolás, a fennmaradó 52 bit pedig 2^{52} lap használatát teszi lehetővé. 8 bájtos bejegyzésekből 512 ($=2^9$) fér el egy lapon, tehát 52/9 felső egész része, vagyis 6 lesz a hierarchia szintek száma. Vagyis minden egyes memóriaművelet során a processzornak 6 további memóriaműveletet kell elvégeznie, hogy a virtuális címből a fizikai címet ki tudja számolni (feltéve, hogy a vonatkozó laptábla bejegyzés nincs bent a TLB-ben), ami nagy mértékben le tudja lassítani a program futását.

A hierarchikus laptábla messze a legnépszerűbb megoldás a 32 bites processzorok körében, ilyen megoldást használ az x86 és az ARM is. Az x86 és az ARM architektúrák 64 bites kiterjesztése is megtartotta ezt az adatszerkezetet, annak ellenére, hogy a hierarchikus laptábla 64 bites címzés mellett túl méllyé válik. TLB hiba esetén 4-5-6 memória-hozzáférés szükséges a címfordításhoz, ami, tekintve, hogy a memóriaműveletek időigénye akár 2-3 nagyságrenddel is nagyobb lehet az órajelciklusnál, lényegesen rontja a rendszer teljesítményét.

9.4.3. Virtualizált laptábla

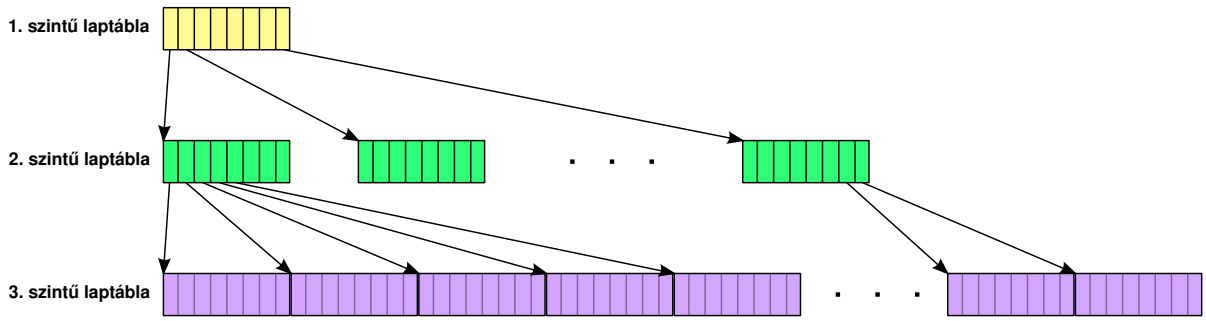
Egy ügyes trükkel, amit a MIPS és a 64 bites Alpha processzorok is alkalmaznak (OSF/1 és OpenVMS operációs rendszerek alatt), szerencsés esetben egyetlen memóriaműveletre lehet korlátozni a címfordításhoz szükséges laptábla bejegyzés betöltését.

Emlékezzünk vissza milyen egyszerű volt a bejegyzés megtalálása az egyszintű laptáblák esetén:

$$i\text{. laphoz tartozó bejegyzés címe} = \text{laptábla kezdőcím} + i \cdot \text{egy bejegyzés mérete}$$

Ez az egyszerűség annak köszönhető, hogy a fizikai memóriában folytonosan (egy ”tömbben”) helyezkednek el a bejegyzések. Az világos, hogy pl. 64 bites címek esetén ez nem járható út, hiszen a laptábla nem fér be folytonosan a fizikai memóriába. De a virtuális címtartományba igen!

A trükk lényege, hogy a tényleges címfordítási információt tartalmazó laptábla bejegyzéseket a virtuális címtartományban folytonosan helyezzük el, egy tömbben, ahogy azt az egyszintű laptábla tette. Ennek ellenére továbbra is hierarchikus laptáblánk van, melynek gyökere és a köztes mutatókat tartalmazó laptáblái tetszőlegesen elhelyezhetők a memóriában, de a hierarchia legalsó szintjéhez (a 9.7. ábrán a 3. szinthez) tartozó lapok legyenek virtuálisan egy folytonos címtartományban (9.7. ábra). Virtualizált laptáblákat használó processzorokban az 1. szintű laptábla fizikai kezdőcímét és a tényleges címfordítási információkat tartalmazó, legalsó szintű (az ábrán a 3. szintű) laptábla virtuális kezdőcímét egy speciális regiszter tárolja.



9.7. ábra. Virtualizált laptábla elhelyezkedése

A címfordítás menete a következő lesz:

1. A futó program által hivatkozott virtuális címről leválasztjuk a lapszámot
2. Tegyük fel, hogy a laphoz tartozó laptábla bejegyzés nincs a TLB-ben (ellenkező esetben abból közvetlenül, számottevő időveszteség nélkül végre lehet hajtani a címfordítást), vagyis TLB hiba van, a bejegyzést be kell olvasni a fizikai memóriából
3. Kiszámítjuk a keresett laptábla bejegyzés virtuális címét (=laptábla hierarchia alsó szintjének (az ábrán a 3. szintjének) virtuális kezdőcíme + keresett lap sorszáma szorozva egy laptáblabejegyzés méretével)
4. Megnézzük, hogy van-e olyan TLB bejegyzés, ami a laptábla bejegyzés virtuális címét leképzi fizikai címre
5. Ha van ilyen TLB bejegyzés, akkor nyert ügyünk van: a címfordítást elvégezve közvetlenül kiolvassuk a fizikai memóriából a keresett laptábla bejegyzést, végrehajtjuk a program által hivatkozott cím címfordítását és elvégezzük a program által kiadott memóriaműveletet (tehát 2-szer kellett a memóriához fordulni).
6. Ha nincs ilyen TLB bejegyzés, akkor a klasszikus módon, a hierarchikus laptábla top-down bejárásával jutunk el a keresett laptábla bejegyzésig, és csak ezután végezhetjük el a program által kiadott memóriaműveletet (tehát pl. 64 bites rendszerben 4/5/6 + 1-szer kellett a memóriához fordulni).

A TLB hatékony megvalósítása a virtualizált laptáblák esetén még fontosabb, hiszen az nemcsak a program által kiadott címekhez kell, hogy biztosítson gyors címfordítást, de a TLB hiba esetén magát a laptábla bejegyzést is a TLB-re támaszkodva keressük a fizikai memóriában. Ez a kettős nyomás még nagyobb TLB-k használatát teszi szükségessé, aminek fogyasztási/melegedési vonzatai vannak.

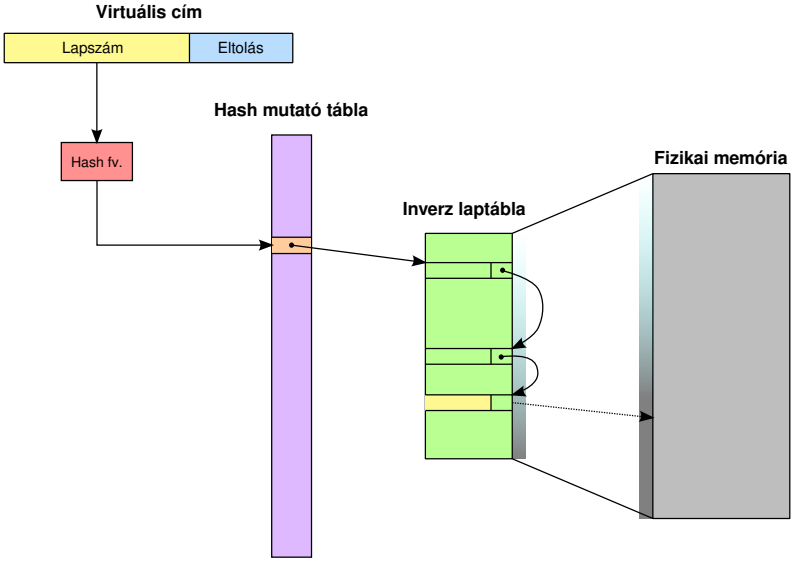
9.4.4. Inverz laptábla

A ”hagyományos” laptáblák minden laphoz egy bejegyzést tartalmaznak, melyből kiolvasható hogy az a lap melyik keretben, vagy ha háttértáron van, annak melyik szektorában található.

Az inverz laptábla pont fordítva működik. Minden kerethez tartalmaz egy bejegyzést, melyből kiolvasható, hogy abban a keretben melyik lap található. Ennek megfelelően az inverz laptáblák mérete nem a lapok számával, hanem a keretek számával arányos, vagyis kizárólag a fizikai memória méretétől függ. Ha kevés a fizikai memória,

akkor kevés a keret, a laptábla is kicsi. Ha nagy a fizikai memória, akkor nagyobb a laptábla. Ez a tulajdonság a manapság tipikusan használt memóriaméretek és a 64 bites címeknek köszönhető óriási virtuális címtartomány mellett egy nagyon lényeges előny.

Az adatszerkezet azért *inverz*, mert a laptáblát nem a lapsorszámmal, hanem a keret sorszámmal indexeljük. Örvendetes, hogy kicsi a laptábla, de van egy probléma: a laptáblát a processzor úgy szeretné használni, hogy a lapsorszám alapján keresi a keretet. Tehát az inverz laptáblában tartalom szerint kellene keresni, és a megtalált bejegyzés laptáblabeli indexe lesz a keresett keret. Bármennyire is tűnik rossz ötletnek ez a megoldás, vannak nagyon hatékony implementációk (PA-RISC és POWER), melyek 64 bites virtuális címeket használnak, és az inverz laptáblának köszönhetik, hogy TLB hiba esetén kevés memóriaműveletre van szükségük a címfordításhoz.



9.8. ábra. Címfordítás inverz laptáblával

A hatékony implementáció fontos eleme a hash tábla (9.8. ábra). A lapsorszámból egy hash függvény segítségével képezünk egy indexet, majd kiolvassuk egy ú.n. hash mutató tábla (hash anchor table) ennyiedik bejegyzését. Ebben a bejegyzésben egy mutató található az első olyan laptábla bejegyzésre, amely olyan lapot tartalmaz, aminek ez a hash értéke. Megnézzük ezt a laptábla bejegyzést, hogy a benne lévő lapsorszám egyezik-e azzal, amit keresünk. Előfordulhat, hogy nem, hiszen a hash függvények tulajdonsága, hogy több lapsorszámra is adhatják ugyanazt az értéket. Ebben az esetben a laptábla bejegyzésben található egy újabb mutató, ami a következő olyan laptábla bejegyzésre mutat, aminek ugyanez a hash értéke. Tulajdonképpen az azonos hash értéket adó lapokat tartalmazó bejegyzések láncolt listába vannak fűzve, melynek első elemét a hash mutató tábla megfelelő bejegyzése tartalmazza. Ezen a láncolt listán kell végigmenni, amíg az általunk keresett lapot tartalmazó bejegyzést meg nem találjuk. Ekkor ránézünk, hogy a laptábla hányadik bejegyzésén állunk: ez lesz a keret száma, ahol a keresett lap található.

Ha a hash tábla értékkészlete elég nagy, akkor ezek a láncok rövidek, jellemzően csak 1-2 elemből állnak, így a keresett keret 1-2 lépésben megtalálható. A nagy értékkészletű hash függvény alkalmazásának hátránya is van: a nagy méretű hash mutató tábla, ami a laptábla mellett szintén a fizikai memóriában van tárolva.

9.4.5. Szoftver menedzselt TLB

Az eddigiekben feltételeztük, hogy a TLB hiba feloldásakor a hardver, vagyis a processzor járja be a laptáblát a keresett laphoz tartozó laptábla bejegyzés betöltése érdekében. Ez azonban nem az egyetlen lehetőség. Számos architektúra alkalmaz *szoftver menedzselt TLB-t*, ami annyit jelent, hogy a processzor nem foglalkozik a laptáblával. A TLB-t természetesen ekkor is ő kezeli, de TLB hiba esetén értesíti az operációs rendszert, hogy keresse meg az adott laphoz tartozó laptábla bejegyzést. Az operációs rendszer olyan adatszerkezetet használ a bejegyzések tárolására, amilyet csak akar (p. a SPARC esetén a SUN Solaris operációs rendszer inverz laptáblát, a Linux pedig hierarchikus laptáblát alkalmaz). A megtalált bejegyzést ezután odaadja a processzornak, az pedig a TLB-be

helyezi. Van olyan architektúra, melyben még arra is van mód, hogy az operációs rendszer határozza meg, hogy az újonnan TLB-be kerülő bejegyzés melyik régit írja felül.

A megoldás nagy előnye, hogy semmi nincs hardverhez kötve, vagyis az operációs rendszer frissítésével könnyűszerrel át lehet állni egy újabb, jobb, továbbfejlesztett laptábla adatszerkezetre. Ugyancsak előny, hogy jóval bonyolultabb eljárások is alkalmazhatók, hiszen a hardver megoldással ellentétben nem kell a fogyasztási és a tranzisztorok számára vonatkozó korlátokkal számolni.

A szoftver menedzselt TLB hátránya, hogy a címfordítás lényegesen lassabb lehet, hiszen maga a címfordítást végző operációs rendszer szubrutin utasításai is a lassú memóriában vannak, melyeket be kell tölteni, és végre kell hajtani.

Hardver menedzselt laptáblát használ az x86, az ARM, a PowerPC, szoftver menedzseltet a MIPS, az Alpha és a SPARC. A PA-RISC architektúrában elmosódnak a határok. A hardver inverz laptáblát implementál, de nem követi az azonos hash függvénnyel rendelkező lapok láncolt listáját. Ha elsőre nem találja meg a lapot, szól az operációs rendszernek, hogy járja be a láncolt listát ő.

9.4.6. Méretezési kérdések

Eddig még nem foglalkoztunk azzal a kérdéssel, hogy a virtuális tárkezelés során a lapméretet milyen nagyra érdemes megválasztani.

Érvek nagyobb lapok mellett:

- kevesebb a TLB hiba, mivel nagyobb a TLB lefedettség
- ha a háttértár diszk alapú, akkor a lapok memóriába töltési idejét a forgatási és fejpozicionálási idő dominálja, vagyis nagy lapok betöltése szinte ugyanannyi ideig tart, mint kis lapoké. Nagy lapok esetén a forgatási és fejpozicionálási idő amortizálódik.

Érvek kisebb lapok mellett:

- Ha kisebb a lap, akkor kisebb az esélye, hogy a szükséges adatokon kívül ugyanazon a lapon lévő, de esetleg soha nem használt adatokat is a ”drága”, gyors elérésű memóriában kelljen tárolni.

Mindezen szempontok figyelembe vételével 4-8 kB méretű lapok használata a legelterjedtebb.

9.5. Címtér-elkülönítés a multi-tasking operációs rendszerek támogatásához

A bevezetőben is említettük, milyen fontos a virtuális memória támogatása a multi-tasking operációs rendszerek számára. Ezek az operációs rendszerek több taszkot is tudnak futtatni egyidejűleg, melyek a fizikai memóriát közösen használják. Ha kevesebb a processzor, mint a futó taszk, akkor az operációs rendszer akkor is képes az egyidejű futtatás látszatát kelteni, mégpedig időosztással: egy ideig az egyik taszkot hagyja futni, majd elveszi tőle a processzort, elmenti az állapotát, és a következő taszkra vált, stb.

Egy modern operációs rendszer minden taszk számára egy saját, összefüggő, 0-val kezdődő címtartományt kínál fel, attól függetlenül, hogy éppen hány taszk fut, és mennyi memória áll rendelkezésre fizikailag. Számos architektúrán minden taszk (egyidejűleg) megkaphatja elméletileg akár a 0. címtől a virtuális címtartomány felső határáig tartó teljes memóriát is. A taszkok így egy állandó futási környezetet érzékelnek, ami a szoftverfejlesztést nagy mértékben megkönnyíti, hiszen fordítás során tudható, hogy a program belépési pontja, az egyes függvények, globális változók, stb. milyen címre kerülnek. A címtér-elkülönítés azonban nem csak kényelmi célokat szolgál. Azáltal, hogy egy taszk saját virtuális címtartománnyal rendelkezik, lehetetlenné válik, hogy más taszkokhoz tartozó memóriaterületeket érjen el, esetleg hibás működés vagy rosszindulat következtében módosítson. Így a címtér-elkülönítés a 16.4.3. fejezetben tárgyalt memóriavédelem egyik fő eszköze lesz.

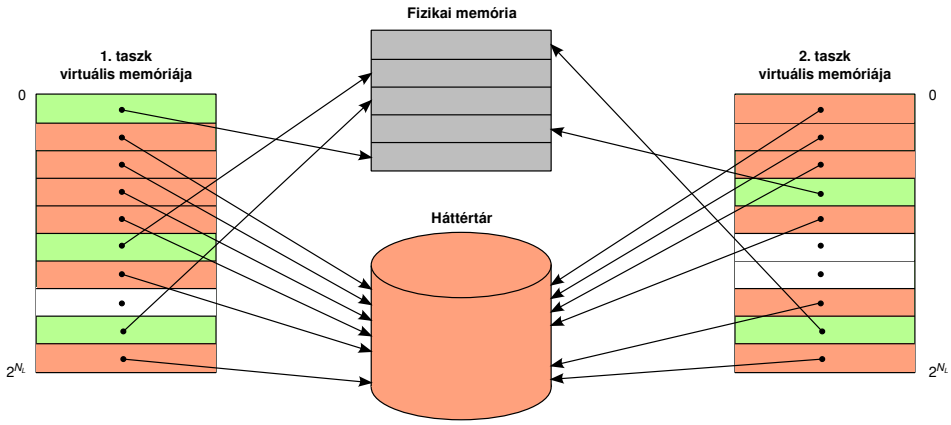
A taszkok címterének elkülönítésére pár architektúra hardver támogatással is rendelkezik, de ennek hiányában, megfelelő laptábla-menedzsmenttel, az operációs rendszer is megvalósíthatja azt.

9.5.1. Címtér-elkülönítés kizárólag a laptáblára alapozva

Ha minden taszk külön, saját laptáblát kap, akkor könnyen megoldható, hogy mindegyikük 0-val kezdődő, saját címtartománnyal rendelkezzen. A taszkváltást ilyenkor laptábla váltás is kíséri, ami kimerül annyiban, hogy az

operációs rendszer lecseréli a processzor speciális, a laptábla kezdetére mutató regiszterének tartalmát. Ezzel tulajdonképpen átvált az egyik taszk virtuális memóriájáról a másik taszkéra.

Az operációs rendszer menet közben igyekszik a fizikai memóriában rendelkezésre álló keretekkel gazdálkodni, azokat úgy megosztani a futó taszkok között, hogy az egyrészt igazságos legyen, másrészt, amennyire lehetséges, mindegyiknek elég legyen. Minden taszkhoz rendel bizonyos számú keretet, melyeken a taszk a gyakran használt lapjait tárolhatja (9.9. ábra), a többit pedig a korábban megismertek szerint a háttértárra helyezi. Mindezt minden taszk saját laptábláján, külön-külön könyveli.



9.9. ábra. Több taszk laptáblája egy multi-tasking operációs rendszerben

A virtuális tárkezelés segítségével az is megoldható, hogy a taszkok virtuális memóriájában legyenek olyan címtartományok, melyek minden taszkra közösek, ezek mögött a fizikai memória ugyanazon keretei állnak. Ezeknek az osztott címtartományoknak fontos szerepük van, többek között ezeken keresztül lehet kommunikálni az operációs rendszerrel.

Látható tehát, hogy a multi-taszking megvalósítása, a taszkok címterének elkülönítése már alapszintű virtuális memóriakezeléssel is megoldható. A laptáblák közötti váltogatásnak azonban van egy olyan vonzata, mely ronthatja a rendszer teljesítményét. Laptáblaváltáskor ugyanis a TLB tartalma is kiürítésre (érvénytelenítésre, flush) szorul, hiszen megváltoznak a lap↔keret összerendelések, az egyik taszk egy lapja más kereten található, mint a másik taszk ugyanazon lapja. A kiürítés után, amíg a TLB újra meg nem telik, minden laphivatkozás lassú laptábla bejárással jár.

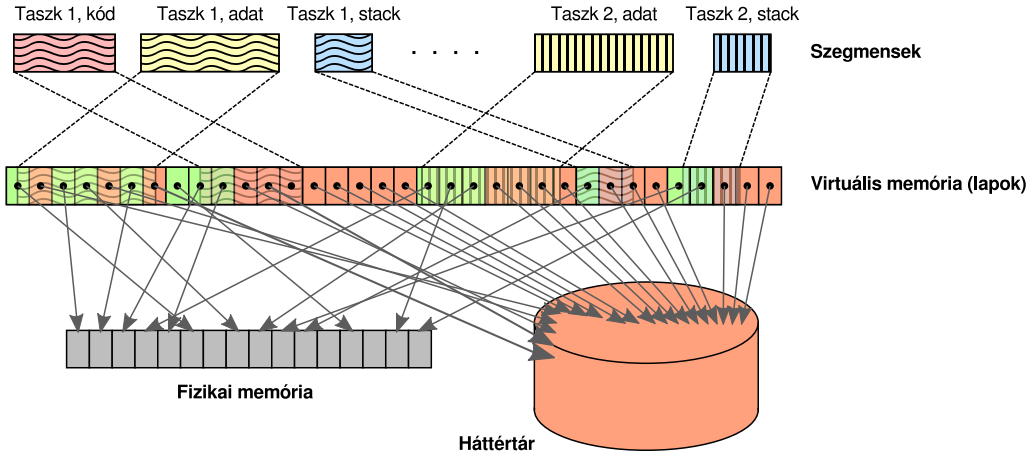
Ennek a problémának a kiküszöbölésére számos architektúra bevezette a *címtér azonosítót* (address space identifier, ASID), amely szerepét tekintve nem más, mint egy taszk azonosító. A TLB bejegyzések pedig, az eddigi ”lapszám” és ”keretszám” mellett egy újabb mezőt kapnak, a címtér azonosítót, amely meghatározza, hogy az adott TLB bejegyzés mely taszkhoz tartozik. A TLB így most már több taszk lap↔keret összerendeléseit is tárolni tudja egyszerre. Az operációs rendszer taszkváltáskor egy speciális regiszterbe írja az épp folytatni kívánt taszk azonosítóját, és onnantól kezdve az MMU minden memóriahivatkozáskor csak az ahhoz tartozó TLB bejegyzéseket fogja figyelembe venni. A TLB kiürítése és az újbóli feltöltése okozta időveszteség így kiküszöbölhető. ASID támogatással rendelkezik többek között az ARM, a SPARC, az Alpha és a MIPS architektúra.

9.5.2. Szegmentálásra alapozott címtér-elkülönítés

A taszkok címterének elkülönítésére használható a számos architektúra által támogatott kétlépcsős címfordítás (two-stage translation), más néven lapozott szegmentálás (paged segmentation) is.

A szegmentálás valójában ennél sokkal többre képes. Bármilyen, logikailag egybetartozó objektumot (pl. a taszk kódját, adatait, stack-ét, vagy egy nagyobb adatszerkezetet) külön *szegmensbe* lehet helyezni. A szegmensek változó hosszúságúak (sőt, futás közben is növelhetők, ill. csökkenthetők), a virtuális memóriában bárhonnét kezdve, folytonosan helyezkednek el, és belül saját címtérrel rendelkeznek.

Szegmentált lapozás esetén a memória tartalma a 9.10. ábrán látható módon szerveződik. A változó méretű memóriabeli objektumokat a szegmensek képviselik, melyeket a szegmensleíró tábla a virtuális memóriára képez le, majd a virtuális memória fix méretű lapjait a laptábla alapján vagy a fizikai memória, vagy a merevlemez tárolja.



9.10. ábra. Lapozott szegmentálás

A programból kiadott memóriacímek így egy sokkal hosszabb, összetettebb folyamaton mennek keresztül, mire fizikai cím válik belőlük. A címleképzés két lépése:

1. Első szint. A szegmensek leképzése a virtuális memóriára (pontosabban a program által kiadott, ún. effektív cím (az x86 terminológia szerint logikai cím) leképzése virtuális címre)
   - Adategység: a változó méretű szegmensek, melyek a programok, ill. az operációs rendszer igényeihez, kényelméhez igazodnak
   - A leképzéshez használt adatszerkezet: a szegmensleíró tábla, melynek minden sora egy-egy szegmens elhelyezkedését adja meg. Mezői: a szegmens kezdete a virtuális memóriában, hossza, védelmi információk, valid bit (ugyanis komplett szegmenseket is lehet a háttértáron tárolni, de erre most nem térünk ki).

2. Második szint: A virtuális memória lapjainak leképzése a fizikai memóriára (azaz a virtuális cím leképzése fizikai címre)
   - Adategység: a fix méretű lapok, melyek a hardver által könnyen kezelhetők
   - A leképzéshez használt adatszerkezet: a laptábla, mely a lap↔keret összerendeléseket tárolja

Az x86 architektúra 32 bites üzemmódjában is van lehetőség szegmenskezelés használatára, sőt, ezt a funkciót ki sem lehet kapcsolni. Ebben az architektúrában minden egyes taszkhoz 3 szegmenst feltétlenül hozzá kell rendelni:

- A kódszegmenst, innen hívja le a processzor a gépi kódú utasításokat. Az utasításszámláló is a kódszegmensen belülre vonatkozik.
- Az adatszegmenst, amire az adatelérő/adatmozgató utasítások vonatkoznak. Az adatokra mutató pointerek mind az adatszegmensre vonatkoznak, annak kezdetéhez relatívak.
- A stackszegmenst, amiben a stack-kezelő (push, pop) utasítások dolgoznak.

Minden függvényhívást és adatelérést, amely az adott taszk kód-, illetve adatszegmensére vonatkozik, *közeli ugrásnak*, illetve *közeli címzésnek* neveznek, míg a szegmenseken átnyúló ugrást és adatelérést *távolinak*.

A modern operációs rendszerek jellemzően nem élnek a szegmentálás adta lehetőséggel, mert a szegmens-szervezés okozta többletadminisztráció nem térül meg. Mind a Linux, mind a Microsoft Windows újabb verziói kizárólag a laptáblára alapozott megoldást használják (9.5.1. fejezet). Az x86 architektúrában, 32 bites üzemmódban az ún. ”flat” memóriamodellt követik, vagyis egyetlen szegmenst hoznak létre (a 0-ás kezdőcímtől 4 GB-ig), és mindent ezen keresztül valósítanak meg. A 64 bites x86 architektúrában a szegmentálás már nem is használható (speciális esetektől eltekintve), ez a funkció fokozatosan eltűnik a palettáról.

Az x86-on kívül a PowerPC is támogat szegmentálást, de egy kissé kötöttebben, rugalmatlanabbul: a szegmensek hossza fix (256MB), és a szegmens azonosítója a program által kiadott címek felső bitjeiből származik.

9.6. Fizikai címkiterjesztés (PAE)

Amikor a 32 bites processzorok megjelentek, elképzelhetetlennek tűnt, hogy egy személyes, házi használatra szánt számítógépben valaha is 4 GB memória legyen. Ez az idő azonban mégiscsak eljött, nemcsak az x86 alapú PC-k, hanem az ARM alapú mobil eszközök világában is. Természetesen a memóriakorlát átlépése érdekében be lehet vezetni új, 64 bites utasításkészlet architektúrát, de ez sajnos a kompatibilitás feladásával jár. Meg lehet úgy tervezni egy processzort, hogy a régi, 32 bites utasításkészletet is támogassa a 64 bites mellett, de az óriási számban piacon lévő 32 bites programok akkor sem fognak tudni profitálni a többlet memóriából, hiszen azok továbbra is csak 32 bites memóriacímeket használhatnak, amivel továbbra is csak 4 GB memóriát tudnak elérni.

Mégis van egy olyan technika, ami lehetővé teszi, hogy 32 bites processzorok 4 GB-nál több memóriát tudjanak címezni, és azt az operációs rendszer ki is tudja használni. Ezt a technikát fizikai címkiterjesztésnek (physical address extension, PAE) hívják.

A PAE ötlete nagyon egyszerű. A 32 bites programok semmiképp sem tudnak túllépni a 4GB-os korláton. De ha van egy multi-tasking operációs rendszerünk, mely több taszkot (programot) futtat, akkor az egyenként 4 GB-os memóriafelhasználás összességében túllépheti a 4 GB-t!

A 9.9. ábrára ismét ránézve látható, hogy semmi akadálya egy nagyobb fizikai memória használatának. Az egyetlen nehézség, hogy a laptábla bejegyzések ”keret” mezője szélesebb lesz, ami magát a laptábla bejegyzést is szélesebbé teszi. Emiatt a PAE használatához szükség van az operációs rendszer támogatására is, hogy az új formátumú laptábla bejegyzéseket is kezelni tudja.

Összefoglalva tehát, a PAE segítségével a programok továbbra sem tudnak 4 GB-nál több memóriát használni, de több program együttes memóriaigénye meghaladhatja a 4 GB-os korlátot. A 32 bites x86 architektúra a Pentium Pro óta 36 bites fizikai címeket támogat, így a PAE segítségével 64 GB-ot tud kezelni. Az ARM saját tervezésű 32 bites processzorai közül a Cortex-A15-től kezdődően lehet 40 bites fizikai címeket használni, amely 1 TB memória illesztését teszi lehetővé.

9.7. Virtuális tárkezelés a gyakorlatban

Ebben a fejezetben röviden áttekintjük, hogy néhány elterjedtebb processzorcsalád milyen módon valósította meg a virtuális tárkezelést.

Az x86 virtuális tárkezelése

Az x86 32 bites üzemmódban lapozott szegmentálásra, 64 bitesben pedig lapozásra alapozott virtuális tárkezelést alkalmaz.

Alapesetben az x86 4 kB-os lapokat használ, és hierarchikus laptáblában tárolja a laptáblabejegyzéseket. A hierarchiaszintek száma, a virtuális címek tagozódása és a laptáblabejegyzések mérete azonban sok mindentől függ.

32 bites üzemmód, PAE nélkül. Ez a 32 bites üzemmód alapesete. A laptábla a klasszikus kétszintű hierarchikus laptábla felépítését követi. A virtuális cím egy 12 bites eltolásból, egy 10 bites másodszintű és egy 10 bites első szintű laptábla indexből áll. A bejegyzések 4 bájtosak, így mind az első, mind a másodszintű laptáblák pontosan egy lapnyi memóriát foglalnak (2^{10} * 4 byte = 4 kB). A processzor által támogatott címtartomány 4 GB.

32 bites üzemmód, PAE mellett. A PAE engedélyezése után nagyobbak, 4 helyett 8 bájtosak lesznek a laptábla bejegyzések, hogy a nagyobb fizikai memóriának megfelelő több keretet tudjanak hivatkozni. A 8 bájtos bejegyzésekből azonban csak 512 fér egy lapra, ami = 2^9, emiatt sajnos a korábbinál eggyel magasabb, három szintű laptáblák használata vált szükségessé. A virtuális címek megváltozott tagozódása tehát: 12 bit eltolás, 9 bites harmadszintű laptábla index, 9 bites másodszintű laptábla index, majd a fennmaradó 2 bit egy mindössze 4 bejegyzésből álló első szintű laptáblát indexel. A nagyobb laptáblabejegyzésnek köszönhetően a 32 bites virtuális címek hosszabb, 36 bites fizikai címekre képezhetők le, ezzel 64 GB-ra nő a támogatott memória mérete.

64 bites üzemmód. A 64 bites üzemmódban a virtuális címek jelenleg 48 bitesek, a fizikai címek pedig processzortól függően 40-48 bitesek (a jövőben mind a virtuális, mind a fizikai címek könnyűszerrel bővíthetők lesznek, egészen 64 bitig). A laptáblabejegyzések 8 bájtosak, így egy lapra 512 bejegyzés fér el. A virtuális címek

tagozódása: 12 bit eltolás, 9 bit negyedszintű laptábla index, 9 bit harmadszintű laptábla index, 9 bit másodszintű laptábla index, 9 bit első szintű laptábla index, vagyis minden egyes címfordításhoz négy laptáblát kell elolvasni, ami lassúvá teheti a memóriaműveleteket.

Az x86 nem csak 4 kB, hanem nagyobb lapok használatát is lehetővé teszi, ekkor a laptábla hierarchia eggyel kevesebb szintből fog állni. Ekkor az alapesetben 12 bites eltolás kiegészül a legalsóbb szintű laptábla indexszel. Tehát a 32 bites, PAE nélküli üzemmódban 12+10 bites lesz az eltolás (ami 2^{22} = 4 MB lapokat jelent); a 32 bites, PAE-val kiegészített, illetve a 64 bites üzemmódban pedig 12+9 bit lesz az eltolás (ekkor a lapok mérete 2^{21} = 2 MB).

ARM

Az ARM architektúra is hierarchikus laptáblát alkalmaz. Többféle lapméretet támogat, melyeket az x86-tal ellentétben vegyesen is lehet használni (a lap mérete a vonatkozó laptábla bejegyzésben szerepel). A rendelkezésre álló lapméretek:

- 4 kB (lap),
- 64 kB (nagy lap),
- 1 MB (section),
- 16 MB (super section),

melyek közül a nagy lapok és a super section-ök kezelése elég körülményes.

Akárcsak az x86-nál, az ARM architektúrában is meg lehet spórolni egy hierarchiaszintet a címfordítás során a nagyobb lapozási egységek (section, super section) használatával: a lapok és a nagy lapok 2, a section-ök és super section-ök 1 szintű címfordítást igényelnek.

PowerPC

A POWER architektúra inverz laptáblát alkalmaz, de nem a klasszikus módon: az azonos hash-hez tartozó lapok láncolt listáját nem valósították meg. A POWER virtuális tárkezelése két tömböt használ:

- Az egyik az inverz laptábla, melynek bejegyzéseiben mind a lap, mind a keret száma szerepel
- A másik egy hash tábla. A hash függvény minden lapszámhoz 2 hash tábla bejegyzést rendel. Minden hash tábla bejegyzés 8 db lapszám – laptábla index összerendelést tárol. Tehát a lapszámhoz hash-elt 16 összerendelést a processzor mind végig nézi, hogy megállapítsa, hogy a laptábla hányadik bejegyzése tartozik a keresett lapszámhoz. Ha a 16 összerendelés egyike sem tartozik a keresett lapszámhoz, a processzor laphibát jelez az operációs rendszernek.

A megoldás előnye, hogy a processzornak nem kell láncolt listát kezelnie. A megoldás hátránya, hogy nagyon kicsi valószínűséggel ugyan, de előfordulhat, hogy több mint 16 olyan lapot használ a programunk, melyekhez mind ugyanaz a hash érték tartozik. Ilyenkor abban az esetben is laphibát kapunk, ha a lap a fizikai memóriában van – ezt az esetet az operációs rendszernek kell lekezelnie (pl. ilyen esetekre maga is karbantarthat egy laptáblát, és abból próbálja a keresett keretet megtalálni).

10. fejezet

Cache memória

10.1. Lokalitási elvek

A modern processzorok belső működési sebessége (órajele) jóval, akár több nagyságrenddel is meghaladja a busz és az operatív memória sebességét. A helyzet tovább romlik, ha virtuális tárkezelést alkalmazunk, hiszen ilyenkor egy memóriatartalom kiolvasásához több lépésre is szükség lehet (laptábla kiolvasás és a tényleges adat kiolvasása), ha pedig a keresett tartalom nincs a memóriában, akkor az elérési időhöz hozzájön még a lap háttértárról való behozásának ideje is.

A memória elérés tehát teljesítmény szempontjából egy szűk keresztmetszet: ha a végrehajtandó utasításhoz nincsenek kéznél a kívánt adatok, a processzor várakozásra kényszerül, nem tud teljes kapacitásával dolgozni, tehát rossz lesz a kihasználtsága.

Nagy szerencse, hogy a számítógépen futó programok nem véletlenszerűen nyúlkálnak a memóriába. A memóriaelérések a legtöbb esetben bizonyos speciális mintázatot követnek időben és (cím)térben egyaránt, és ezt ki lehet használni a teljesítmény növelése érdekében. Ez a speciális mintázat persze az alkalmazástól függ, de az általános célú programok túlnyomó részében a memóriaműveletek *lokalitási elv*eket követnek. Többféle lokalitási elv is létezik:

- *Időbeli lokatitás* (Temporal locality): Ha egy memóriában tárolt adaton műveletet (írás/olvasás) végzünk, akkor valószínűleg hamarosan újra műveletet végzünk rajta.
- *Térbeli lokatitás* (Spatial locality): Ha egy memóriában tárolt adaton műveletet (írás/olvasás) végzünk, akkor valószínűleg a közelében lévő adatokon is műveletet fogunk végezni.
- *Algoritmikus lokalitás* (Algorithmic locality): Sok program nagy dinamikus adatszerkezetekkel dolgozik (láncolt listákkal, fákkal), melyeket újra és újra bejár, különböző műveleteket végezve rajtuk. Ez a fajta elérési mintázat sem nem időbeli, sem nem térbeli lokalitás (hiszen az adatszerkezet nem feltétlenül folytonos a memóriában), viszont a szabályos viselkedést esetleg ki lehet használni a gyorsabb végrehajtás érdekében.

Az adott alkalmazástól függ, hogy a lokalitási elvek milyen mértékben érvényesülnek. A térbeli lokalitás kitűnően érvényesül médiatartalmak lejátszásánál (hiszen a lejátszás szekvenciálisan történik), akárcsak szövegszerkesztéskor (egy oldal megjelenítése után nagy eséllyel a következő, vagy előző oldal tartalmát kell elérni). Ugyanakkor az időbeli lokalitás elvének éppen a médialejátszás nem felel meg: a már lejátszott tartalomra a közeljövőben valószínűleg nem lesz szükség. Az időbeli lokalitás gyakran érvényesül az iteratív algoritmusokban, pl. egy ciklusban, ahol a ciklus utasításait sokszor ismételve kell végrehajtani.

10.2. A tárhierarchia

A lokalitási elvek tehát tálcán kínálják a lehetőséget: a gyakran használt adatokat, és azok környezetét vigyük a processzorhoz a lehető legközelebb, egy olyan memóriába, ami olyan gyors, hogy sebessége a processzort a lehető legkevésbé korlátozza. Ez a cache memória. A cache memória kialakítására általában SRAM-ot használnak, szemben az operatív memóra céljára használt DRAM-mal (lásd 8.1.1. és 8.1.2. fejezetek). Az SRAM jóval rövidebb elérési idejű a DRAM-nál, de sajnos van pár hátránya: kialakítása sok tranzisztort igényel, sokat fogyaszt és melegszik, így a cache mérete lényegesen kisebb az operatív memóriánál. Az eddigiek alapján tehát

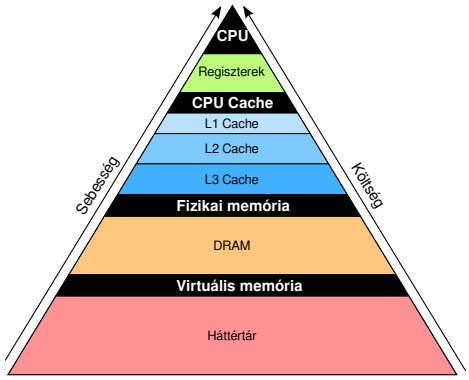
már háromszintű a tárhierarchiánk: háromféle technológiával előállított, különböző sebességű és árú "helyen" tároljuk az adatainkat (a nagyságrendek érzékeltetése kedvéért a 10.1. ábra számszerűen is összefoglalja ezeket a paramétereket [32]).

	Elérési idő	Ár/GB
SRAM	0.5-2.5 ns	2000-5000$
DRAM	50-70 ns	20-50$
HDD	$5 - 20 \cdot 10^6$ ns	0.2$

10.1. ábra. Tár technológiák hozzáférési ideje és ára 2008-ban

A leglassabb tárolón (a háttértáron) tartjuk a legkevésbé használt adatokat (ezt a virtuális tárkezelés eleve támogatja), a gyakran használt adatokat az operatív memóriában tároljuk, végül az éppen futó algoritmus aktuálisan használt adatait - pl. a program aktuális függvényének lokális változóit - pedig még közelebb visszük a processzorhoz, és egy nagy sebességű, de kis méretű SRAM-ba, a cache-be tesszük. Ha a lokalitási elveket ügyesen kihasználva jól menedzseljük a három szint közötti adatmozgatást, akkor elérhetjük, hogy a processzor számára mindig kéznél legyenek az éppen szükséges adatok, így a memória lassúsága jelentette szűk keresztmetszet lényegesen enyhíthető.

A gyakorlatban nem csak 3, hanem akár 4-5-6 szintű tárhierarchiákat is alkalmaznak úgy, hogy a cache funkcionalitását több szinten valósítják meg (10.2. ábra).



10.2. ábra. Tárhierarchia

10.3. Cache megvalósítási lehetőségek, alapfogalmak

A cache megvalósításokat kétféle szempontból csoportosítjuk: címzési mód szerint és menedzsment mód szerint.

A címzési mód szerint:

- *Transzparens:* Transzparens címzési mód esetén a cache tartalma az operatív memória különböző részeinek másolata. A cache a fizikai vagy virtuális memóriabeli címekkel címezhető, vagyis a futó alkalmazások számára teljesen transzparens, azok esetleg nincsenek is tudatában a cache jelenlétének. Az alkalmazások által kiadott címet először a cache-ben kezdjük keresni, és csak ha ott nincs meg, akkor fordulunk a lassú operatív memóriához.
- *Nem transzparens*: A nem transzparens címzési mód azt jelenti, hogy a cache és az operatív memória címzés szempontjából is elkülönül. A címtartomány egy része nagy sebességű SRAM-mal, a fennmaradó része pedig a lassú de olcsó DRAM-mal van megvalósítva. Az alkalmazásoknak tudnia kell a cache létezéséről, annak címtartománybeli pozíciójáról, méretéről, hiszen a gyors elérésű adatok ott helyezkednek el.

Menedzsment szerint:

- *Implicit*: Implicit módon kezelt cache esetén a cache tartalmát maga a cache (ill. a cache vezérlő) menedzseli, az operációs rendszertől és az alkalmazástól függetlenül dönt a cache és a memória közötti adatmozgatásról, vagyis arról, hogy a memória mely részei kerüljenek be a gyors elérésű SRAM-ba ill. ha a cache tele van mely részek kerüljenek ki belőle.
- *Explicit*: Az explicit menedzsment azt jelenti, hogy a futó alkalmazás vagy az operációs rendszer felügyeli a cache tartalmát, explicit ”load” és ”store” műveletekkel.

A kétféle szempont szerinti csoportosítás alapján az alábbi 4 esetet különböztetjük meg ([26]):

	Címzési mód	**Menedzsment**
Transzparens cache	Transzparens	Implicit
Szoftver-menedzselt cache	Transzparens	Explicit
Önszervező scratch-pad	Nem transzparens	Implicit
Scratch-pad memória	Nem transzparens	Explicit

Transzparens cache esetén az alkalmazás nincs tudatában a cache létezésének. A cache a ”háttérben” tevékenykedik, próbál a futó alkalmazás és a processzor keze alá dolgozni, megbecsüli, hogy mely adatok kellhetnek az alkalmazásnak gyakran a közeljövőben, és azokat a cache memóriába mozgatja. Szinte az összes általános célú processzorban ilyen cache található.

Scratch-pad memória esetén mindent az alkalmazás vezérel. Tudja, hogy a memóriának mely címtartománya van gyors SRAM-mal megvalósítva, és ezt mint egy jegyzetfüzetet használja, a gyakran használt adatait odaírja, ha megtelt, ő dönti el, hogy mit írjon felül. Az alkalmazás tudtán kívül a cache-be semmi nem kerül be. Speciális célokra írt alkalmazások számára előnyös lehet, mert teljesen kiszámítható a viselkedése, hiszen nem a hardver - esetleg szerencsétlen - spekulatív döntései határozzák meg a cache tartalmát és ezzel a memóriaműveletek sebességét. DSP-k és mikrokontrollerek előszeretettel használnak scratch-pad memóriát.

A *szoftver-menedzselt cache*-ek nagyon hasonlítanak a transzparens cache-ekre abból a szempontból, hogy a címzés transzparens, tehát címtartományban nem különül el az operatív memóriától. A cache a háttérben figyeli az alkalmazás által hivatkozott címeket, és ha van találat, akkor azt ő gyorsan fel is oldja, elkerülendő az operatív memória lassú elérését. A transzparens cache-hez képest a különbség abból adódik, hogy ha nincs találat (cache hiba történik, vagyis a hivatkozott adatok nincsenek a cache-ben), akkor nem a cache vezérlő gondoskodik a cache frissítéséről, hanem az elhelyezési döntéseket, a spekulációt az alkalmazásra bízza: meghívja az alkalmazás erre a célra beregisztrált szubrutinját.

Az *önszervező scratch-pad* ritka megoldás. A hardver támogatja a tartalom menedzselését (mikor, mi kerüljön be a cache-be), de a gyors elérésű SRAM memória az operatív memória címtartományának a része.

A következő fejezetek az alábbiak szerint épülnek fel:

- A 10.4. fejezet a cache szervezéssel foglalkozik, vagyis azzal, hogy hogyan kell a cache-be került adatokat elhelyezni és abban hatékonyan keresni transzparens és nem transzparens címzés esetén.
- A 10.5. fejezet foglalkozik a tartalom menedzsmenttel, vagyis hogy mikor hozzunk be a cache-be egy új adatot, és ha nincs hely, kit vegyünk ki a cache-ből, implicit vagy explicit menedzsment esetén.

10.4. Cache szervezés

10.4.1. Transzparens címzésű cache szervezés

A cache memória az adatokat nem bájtonként, hanem nagyobb darabokban kezeli, a tárolás alapegysége a *blokk* (a szerepe a virtuális memóriakezelésnél látott *lap*pal azonos). A blokkokat a blokkszámmal azonosítjuk. Ha a blokkok mérete 2 hatványa, akkor a teljes cím felső bitjei a blokkszámnak, alsó bitjei pedig a blokkon belüli eltolásnak felelnek meg. Pl. ha 32 bites címeink vannak, és a cache blokkok mérete 64 bájt, akkor a blokkon belüli eltolás 6 bitet vesz igénybe (mert $2^6 = 64$), tehát 26 bit marad a blokkok azonosítására.

A cache-ben lévő minden egyes tárolt blokk mellett a blokk operatív memóriabeli elhelyezkedésével kapcsolatos és egyéb, tartalommenedzsmentet segítő információkat is el kell helyezni (a virtuális memóriánál látott laptábla bejegyzéshez hasonlóan). Ezek a járulékos információk a következők:

- Cache *tag*: információ arra nézve, hogy az adott cache blokk az operatív memória mely blokkját tartalmazza (ennek mérete és értelmezése a különféle cache szervezési megoldások esetén eltér, ezeket később, még ebben a fejezetben tárgyaljuk)
- *Valid bit*: =1, ha az adott cache blokk érvényes adatot tartalmaz
- *Dirty bit*: =1, ha az adott cache blokk tartalma módosult, mióta a cache-be került. Ilyen esetben ugyanis a blokkot, ha már nincs rá szükség a cache-ben, nem lehet felülírni egy másikkal, hanem előtte a megváltozott tartalmat vissza kell írni az operatív memóriába

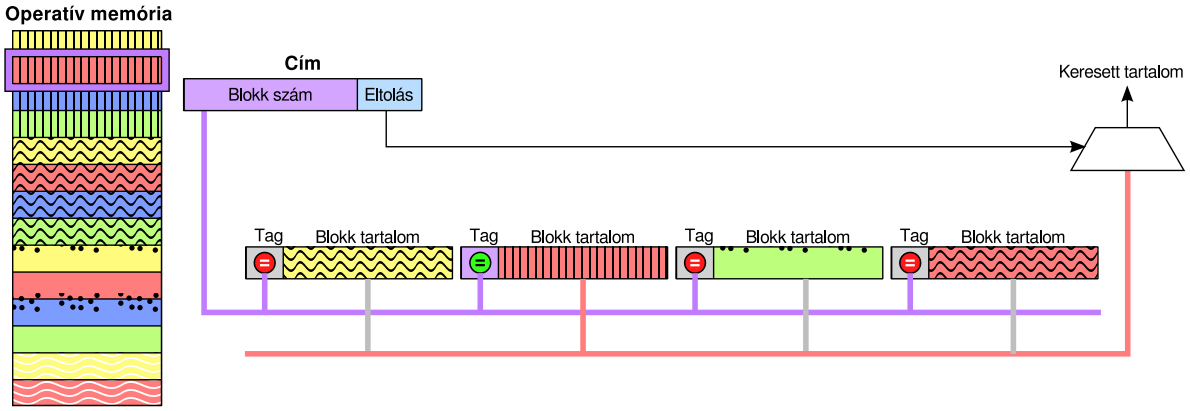
A ”cache szervezés” fogalma arról szól, hogy az operatív memóriából behozott blokkokat hova helyezzük el a cache-en belül, milyen és mennyi járulékos információt tárolunk, ill. egy memóriaművelet esetén hogy találjuk meg a vonatkozó adatokat a cache-ben. Az alábbiakban sorra veszünk néhány elterjedt megoldást, melyekre mind megvizsgáljuk, hogy a következő, fontos szempontok szerint hogyan teljesítenek:

- Keresés sebessége: egy címről gyorsan el kell tudni dönteni, hogy benne van-e a cache-ben, és ha benne van, gyorsan ki kell tudni olvasni a belőle a cím által hivatkozott adatot.
- Egyszerűség és alacsony fogyasztás: mivel a cache általában a processzorral egy félvezető lapkán van kialakítva, fontos szempont, hogy kevés helyet foglaljon (hogy több hely maradjon a processzor számára), valamint hogy a fogyasztási keretből minél kisebbet vegyen el.

Teljesen asszociatív szervezés

A teljesen asszociatív cache szervezés (fully associative cache organization) azt jelenti, hogy az operatív memória bármely blokkját a cache bármely blokkjába tehetjük. A cache tag ebben az esetben maga a blokk operatív memóriabeli sorszáma. A keresés úgy történik, hogy a hivatkozott címről leválasztjuk a blokkszámot, és ezt az összes cache bejegyzés tag mezőjével összehasonlítjuk. Ezt természetesen nem úgy kell elképzelni, hogy a blokkokat sorban, elejétől a végéig bejárjuk és a tag mezőket összehasonlítjuk. A cache memóriában a tag-eket tartalom szerint címezhető memória tárolja, amit (kissé egyszerűsítve) úgy lehet elképzelni, mintha egy-egy komparátor tartozna hozzájuk (melynek bitszélessége a blokkszám bitszélessége), és így a keresett blokkszámot az összes tag-gel egyszerre, egy lépésben ”komparáljuk”. Amelyik cache blokk tag-je találatot jelez, ott van eltárolva a hivatkozott adat (10.3. ábra).

Ennek a működésnek az a hátránya, hogy minden memória hivatkozásnál az összes tag-et össze kell hasonlítani a keresett blokkszámmal. Igaz, hogy ez egy lépésben történik, de közben a teljes tartalom szerint címezhető tag memória (az összes ”komparátor”) működik, ami fogyasztás szempontjából nagyon kedvezőtlen.



10.3. ábra. Teljesen asszociatív cache szervezés

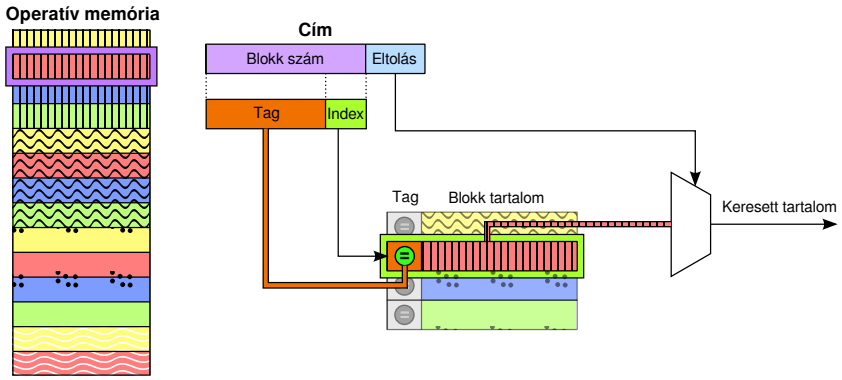
Direkt leképzés

Direkt leképzés (direct-mapped organization) esetén egy memóriabeli blokk száma alapján egyértelműen (direkt módon) eldől, hogy az a cache melyik blokkjában tárolható. Ennek egy lehetséges (és a könnyű megvalósíthatóság miatt elterjedt) megoldása: ha a cache n blokk tárolására alkalmas, akkor a memória $m.$ blokkja a cache-ben

az *m modulo n*-dik blokkba helyezhető el. Természetesen, mivel a cache kisebb, mit az operatív memória, több memóriabeli blokk is ugyanarra a helyre képeződik le a cache-ben, ekkor versenyhelyzetről, konfliktusról beszélünk.

A 10.4. ábrán látható példában 4 blokknyi hely van a cache-ben, az első csak a sárga, a második csak a piros, a harmadik csak a kék, a negyedik pedig csak a zölddel jelölt memóriabeli blokkokat tudja tárolni. Ha n kettő hatványa, akkor a blokkszám alsó bitjei közvetlenül megadják a cache-beli elhelyezkedést (színét). A példában *n=4*, vagyis a blokkszám alsó 2 bitje határozza meg, hogy a blokk hol lehet a cache-ben. Tehát az operatív memória piros blokkjai csak a cache piros színű blokkjába helyezhető, egy piros blokk vagy ott van, vagy nincs bent a cache-ben. Ahhoz, hogy megállapítsuk, hogy az a bizonyos piros blokk van a cache-ben, amit keresünk, össze kell hasonlítani a tag-eket. A cache tag ebben a szervezésben a blokkszámnak az a része, ami nem a cache-beli elhelyezkedés leírására használatos (lásd a 10.4. ábra), vagyis elegendő azt megnézni, hogy hányadik piros blokkot keressük, és hogy a piros tárolóban ugyanaz a piros blokk van-e. Ennek eldöntésében csak a piros blokkokhoz tartozó cache tag-et kell összehasonlítani a cím megfelelő részével, tehát csak egyetlen összehasonlításra van szükség, ami lényeges fogyasztási és költség szempont. Továbbá az összehasonlítandó adatok (azaz a tag-ek) bitszélessége is kisebb a teljesen asszociatív esethez képest, ugyanis a tag ebben az esetben rövidebb, mint a blokkszám.

Tehát a keresés két részből áll: 1. a blokkszám alsó bitjei *indexelik* (kiválasztják) a megfelelő cache blokkot, 2. az ahhoz tartozó tag-et a blokkszám tag részével *összehasonlít*va állapítjuk meg, hogy megtalálható-e a blokk a cache-ben.



10.4. ábra. Direkt leképzés

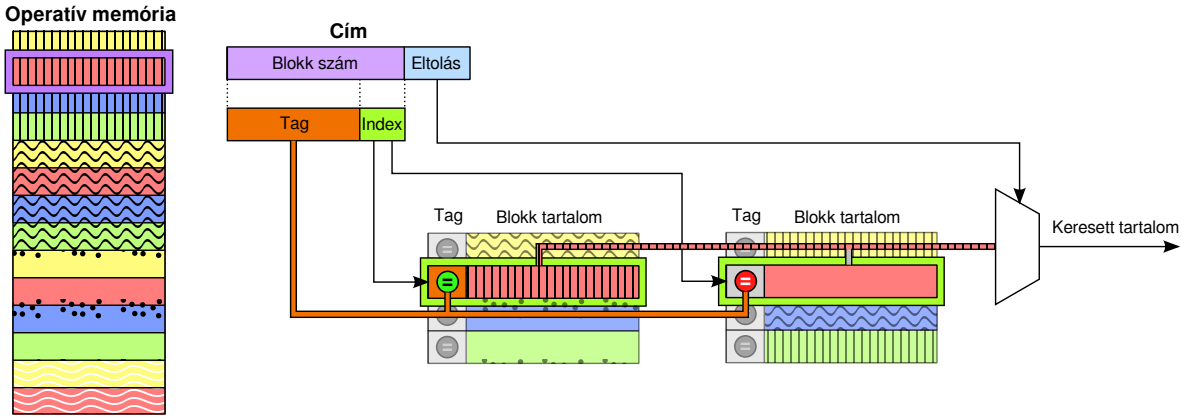
A megoldás hátránya, hogy a cache bizonyos blokkjaira versenyhelyzet alakulhat ki. Ha például az éppen futó algoritmus két pirossal színezett memória blokkra is gyakran hivatkozik, de mivel a szervezés sajátossága miatt a cache-ben csak az egyik helyezhető el, gyakran kell a piros cache blokk tartalmát cserélni (akkor is, ha egyébként máshol még lenne hely a cache-ben), ami rontja a hatékonyságot.

n-utas asszociatív szervezés

A direkt leképzésnek nagyon előnyös tulajdonsága volt a gyors keresés, az alacsony fogyasztás és a kis bitszélességű tag-ek (”komparátorok”) használata. A teljesen asszociatív leképzés előnye volt, hogy egy blokk a cache-be bárhova elhelyezhető, így a cache tényleg a leggyakrabban hivatkozott blokkokat (és csak azokat) tartalmazza. Az n-utas (vagy halmaz-) asszociatív leképzés (n-way vagy set associative organization) a kettő előnyeit próbálja egyesíteni.

A direkt leképzéshez hasonlóan az n-utas asszociatív leképzésben is a blokkszám alsó bitjei határolják be, hogy egy blokk hol lehet a cache-en belül. Azonban a direkt leképzéssel ellentétben a blokkok cache-beli elhelyezését a blokkszám nem határozza meg egyértelműen, hanem n lehetőséget biztosít. Vagyis a memóriabeli blokkszám alsó bitjei kijelölnek egy n cache blokkból álló halmazt, ahol a hivatkozott tartalom előfordulhat. Az, hogy az n blokk közül melyik tartalmazza a keresett adatot, a blokkokhoz tartozó tag-ek összehasonlításával (”komparálásával”) dönthető el.

A keresés tehát két részből áll: 1. a blokkszám alsó bitjei *indexelik* (kiválasztják) a megfelelő cache blokk halmazt (n blokkot), 2. az ezekhez a blokkokhoz tartozó tag-eket a blokkszám tag részével *összehasonlít*va állapítjuk meg, hogy megtalálható-e a blokk a cache-ben (10.5.ábra).



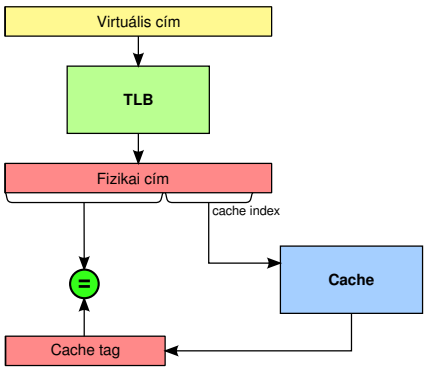
10.5. ábra. 2-utas asszociatív szervezés

Ez a kialakítás tehát gyors és hatékony is, hiszen az indexelés lényegesen leszűkíti a keresési teret és így az egyidejűleg működő komparátorok számát, de a direkt leképzésnél lényegesen rugalmasabb, ritkábban fordul elő az azonos halmazba képezett blokkok között a versenyhelyzet.

10.4.2. Transzparens cache szervezés és a virtuális memóriakezelés viszonya

Egész eddig kerülgettük a forró kását: hol lassú memóriát, hol operatív memóriát említettünk a cache-elés tárgyaként. Ennek oka, hogy egy virtuális tárkezelést támogató processzor esetén egyáltalán nem egyértelmű, hogy a cache szervezés során a cache indexelésére és a tag-ek összehasonlítására a fizikai, vagy a virtuális címet használjuk-e. A tárhierarchia tárgyalásakor láttuk, hogy a cache ”alatt” a fizikai memória áll, annak elérését gyorsítja, ezért kézenfekvőnek látszik, hogy a cache a keresés és a tárolás során fizikai címeket használjon. Mint ki fog derülni, a fizikai helyett a virtuális címek használatának is vannak előnyei. Az alábbiakban végigvesszük az összes lehetséges kombinációt, és mérlegeljük, melyik milyen előnnyel, ill. hátránnyal jár.

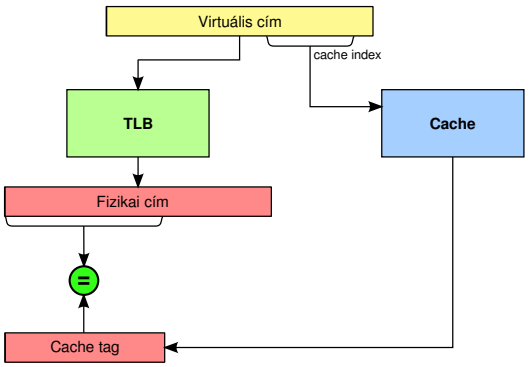
A 10.6. ábra azt az esetet mutatja, amikor a keresés mindkét lépése, azaz a cache indexelése és a tag-ek összehasonlítása során is a fizikai címeket használjuk. Mivel a futó program virtuális címeket használ, minden cache keresést meg kell, hogy előzzön a címfordítás, tehát a cache elérési időhöz hozzáadódik a címfordítási idő is. A címfordítást szerencsés esetben (tulajdonképpen jól méretezett rendszerek esetén az esetek többségében) a TLB oldja fel (de az sem nulla idő alatt), ha nincs TLB találat, a fizikai memóriában lévő laptáblához kell fordulni, hogy a virtuális címhez tartozó fizikai címet megkeressük. Látni fogjuk, hogy sebesség szempontjából nem ez lesz a legjobb választás.



10.6. ábra. Fizikailag indexelt cache fizikai tag-gel

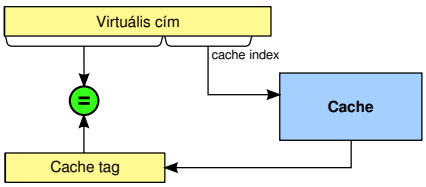
A teljesítmény javítható, ha el tudjuk érni, hogy a TLB és a cache legalább részben párhuzamosan tudjon működni. Ennek egy módja, hogy a cache indexelését a virtuális címmel végezzük, így ugyanis nem kell megvárni a címfordítást (10.7. ábra). Az indexelés és a címfordítás tehát egymással párhuzamosan zajlik. Az index által

kiválasztott cache blokk(ok) tag-einek összehasonlítása pedig a fizikai címek alapján történhet, remélve, hogy mire erre sor kerül, a TLB már végez a címfordítással.



10.7. ábra. Virtuálisan indexelt cache fizikai tag-gel

Ezt a gondolatot tovább is lehet vinni, és a teljes cache működést a virtuális címekre is lehet alapozni. Ilyenkor mind az index, mind a tag a virtuális címből képződik (10.8. ábra). Vegyük észre, hogy cache találat esetén a TLB-re nincs is szükség! Viszont a virtuális címek szélesebbek, mint a fizikai címek (hacsak nincs fizikai címkiterjesztés, 9.6. fejezet), így a cache tag is szélesebb lesz, tehát egyrészt a szélesebb tag-et kell tárolni, másrészt a szélesebb tag-et szélesebb komparátorral kell összehasonlítani a keresett cím megfelelő részével.



10.8. ábra. Virtuálisan indexelt cache virtuális tag-gel

A negyedik kombinációnak, vagyis a fizikailag indexelt és virtuális tag-eket használó cache-nek nincs túl sok értelme, hiszen a cache keresés során először kell indexelni és csak utána tag-et összehasonlítani. A virtuális tag hiába van meg azonnal, ha az indexeléshez mindenképp meg kell várni a címfordítást.

Megjegyezzük, hogy a cache szervezésben a virtuális címek felhasználásának nem csak előnyei vannak, mint ahogy a fenti leírás sugallja. A virtuális memória-kezelés lehetővé teszi, hogy a fizikai memória egy bizonyos keretét a virtuális memória több lapján keresztül is elérhessük egyidejűleg. Ezt az (általunk nem tárgyalt) esetet megengedve előfordulhat, hogy a virtuális címekre alapozott cache-ek a fizikai memória ugyanazon blokkját többször is tartalmazzák, ami nem kívánatos, hiszen feleslegesen pazarolja a helyet az amúgy is szűkös cache-ben. Másrészt, multi-tasking környezetben a taszk váltás mindig laptábla váltással is jár. Ha a cache szervezés a virtuális címekre alapul, akkor minden egyes taszk váltáskor érvényteleníteni kell a cache tartalmát.

10.4.3. Nem-transzparens cache szervezés

Nem transzparens cache szervezés esetén a címtér egy része SRAM-mal van megvalósítva, az ide helyezett adatok tehát lényegesen gyorsabban lesznek elérhetők. A transzparens címzésű cache-ekkel ellentétben ilyenkor a cache nem a lassabb memória egy részének másolatát tartalmazza, hanem önálló, saját címmel rendelkező objektumokat, melyek tipikusan nincsenek is a lassabb memóriában tárolva. Ez a szervezés nem igényel ”keresés” műveletet, hiszen minden cache-ben tárolt adatnak saját címe van, a futó applikáció egyszerűen lekéri a cache n. címének tartalmát. Így tehát nem kellenek tag-ek, vezérlő bitek, és a fogyasztást növelő összehasonlító logikára sincs szükség.

Ezt a megoldást, egyszerűsége és kiszámíthatósága miatt, széles körben alkalmazzák mikrokontrollerekben, de az IBM Cell processzor (a Sony PlayStation 3 processzora) is tartalmaz scratch-pad memóriát, amely, mint láttuk, nem transzparens szervezésű cache.

10.5. Tartalom menedzsment

Az előző fejezetben leírtuk, hogy milyen megoldások vannak az adatok szervezésére, tárolására a cache-ben. Azzal, hogy mikor, mely adatokat helyezzünk el a cache-ben, és ha tele van, mely adatokat távolítsunk el belőle, a tartalom menedzsment foglalkozik. Implicit esetben ezeket a döntéseket maga a cache (ill. a cache vezérlő) végzi, explicit esetben pedig az alkalmazás, illetve az operációs rendszer.

10.5.1. Adatok betöltése a cache-be

A tartalom menedzsment szempontjából talán a legfontosabb döntés, hogy az operatív memória egy blokkját mikor töltsük a cache-be. Ezzel kapcsolatban az alábbi három lehetőségünk van:

- soha,
- amikor a futó program hivatkozik rá,
- jó előre betöltjük, hátha hamarosan szükség lesz rá.

Látszólag a "*soha*" nem tűnik értelmes megoldásnak (hiszen miért is van cache, ha nem használjuk?), de nagyon is az. Vannak ugyanis olyan alkalmazások, programok, melyek memóriahivatkozásaira nem teljesül az időbeli lokalitás elve. Ilyen alkalmazás például a médiatartalmak lejátszása: folyamatos lejátszáskor tipikusan egyszer játszanak ki egy képet vagy hangsort, utána azokra az adatokra soha nem is történik további hivatkozás. Ebben az esetben nem jó döntés minden hivatkozott blokkot betölteni a cache-be, hiszen többször úgysem lesz rájuk szükség, viszont a cache-ből kiszorítják a tényleg gyakran hivatkozott blokkokat. Ezt úgy nevezik, hogy az alkalmazás tele szemeteli a cache-t (cache pollution). Jó lenne tehát az ilyen alkalmazásokat detektálni, és az általuk generált blokkokat a cache-be nem betölteni.

A legegyszerűbb megoldás az, hogy a blokkot akkor töltjük be a cache-be, amikor a futó programnak először van rá szüksége. Ekkor azonban a program futása némi késleltetést szenved, amíg az adatmozgatás meg nem történik (igaz, hogy a további, jövőbeli hivatkozások már a gyors cache-ből oldhatók fel). Ennek a késleltetésnek az elkerülésére szokás *idő előtti betöltés*t (prefetch) alkalmazni. Ekkor a tartalom menedzsmentet végző szoftver vagy hardver elem megpróbálja felderíteni az adatelérések mintázatát, és megtippelni, hogy a közeljövőben mely blokkokra fog a program hivatkozni. Ha a tipp jó, az adatok már a cache-ben lesznek akkor is, amikor az első alkalommal van rájuk szükség.

10.5.2. Cache szemetelés megelőzése

Cache szemétnek minősül az a blokk, ami a cache-be kerülése és a cache-ből való kirakása között egyszer sem lett meghivatkozva (kivéve esetleg azt a hivatkozást, ami a cache-be töltéshez vezetett). Ilyen esetben a blokk nyilvánvalóan feleslegesen került a gyors elérésű cache-be, hiszen a gyors elérés nincs kihasználva, a jelenléte (helyfoglalása) miatt pedig kiszorulhatnak hasznos blokkok a cache-ből.

Speciális hardver utasítások

Szinte minden modern processzor utasításkészlet tartalmaz olyan utasításokat, amellyel befolyásolni lehet az egyébként hardveres implicit cache tartalom menedzsmentet. Példa a cache szemetelés elkerülésére bevezetett utasításokra:

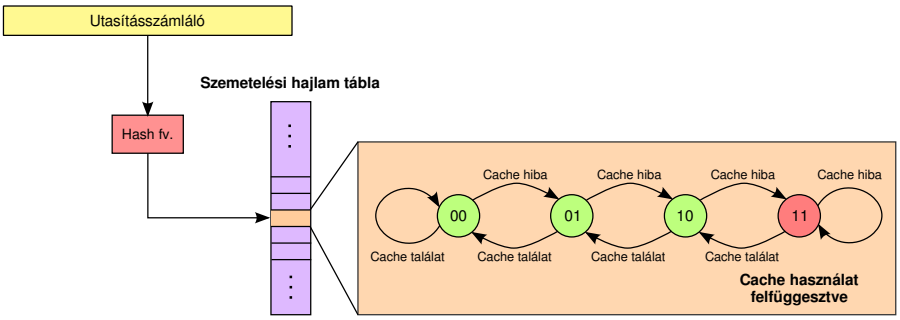
- x86: MOVNTI, MOVNTQ: adat mozgatása SSE regiszterből az operatív memóriába, a cache kikerülésével.
- PowerPC ([35]): LVXL: olvasás az operatív memóriából egy vektor regiszterbe. Az érintett blokk ugyan a cache-be kerül, de egy speciális jelölést kap, aminek hatására ez a blokk kerül ki elsőnek a cache-ből, ha helyszűke adódik. Az utasítás párja az LVX, aminek ugyanaz az adatmozgatás a feladata, de nem jelöli meg a cache blokkot. A programozó tehát jelezheti, hogy a kívánt memóriatartalom megfelel-e az időbeli lokalitásnak, vagy sem.
- PA-RISC ([27]): néhány memóriaműveletet végző utasítás rendelkezik olyan opcióval, amivel lehet jelezni, hogy a hivatkozott adat a cache-be bekerüljön-e, vagy sem. Ennek az opciónak (Spatial Locality Cache Control Hint) a betartása azonban a specifikáció szerint nem kötelező a processzor számára.
- Itanium ([3]): az adatmozgató utasításoknál egy .NT opció megadásával lehet jelezni a processzornak, hogy ez az adat időben nem lokális (non-temporal).

- Alpha ([15]): az ECB (Evict Data Cache Block) utasítással lehet utalást tenni a processzornak hogy a megadott cache blokk a közeljövőben nem lesz újra hivatkozva.

Időben nem lokális viselkedés kiszűrése

Ha a szoftver (explicit módon) nem használja ki az utasításkészlet által biztosított lehetőségeket a cache szemetelés elkerülésére, más, automatikus (implicit) módot kell keresni. A szakirodalomban a 90-es évek második felében szép számban publikáltak leírásokat különféle heurisztikákról, melyek a cache szemetelés problémáját kezelik. Ezek közül most egy olyan megoldást tárgyalunk, amely nem igényel módosítást a cache szervezésében (tehát nem kívánja meg újabb vezérlő bitek karbantartását magában a cache-ben, illetve a keresés és tárolás mechanizmusa is változatlan marad).

Ez az eljárás (Rivers' non-temporal streaming cache) megpróbálja nyomon követni a futó program memória-író/olvasó utasításai nyomán fellépő cache találatok és cache hibák alakulását. Ha egy utasítás (pl. egy ciklusban) nagy számú cache hibát generál, állandóan blokkokat kell behozni a kedvéért, de cache találatot egyáltalán nem produkál, akkor az egy "cache szemetelő" utasításnak minősül, és az általa hivatkozott blokkokat nem hozzuk be a cache-be.



10.9. ábra. Cache szemetelés elkerülése Rivers' algoritmusával

Az eljárás sémája a 10.9. ábrán látható. Ennek központi eleme egy táblázat, melyet az utasításszámláló indexel, és melynek egy eleme az adott címen lévő utasítás "szemetelési hajlamát" tartalmazza. Ebben a példában erre a célra 2 bitet tartunk fenn, amihez egy 4 állapotú véges automata tartozik. Minden, az utasítás által előidézett cache találat csökkenti a "szemetelési hajlamot", és minden cache hiba növeli azt. Ha a "szemetelés hajlam" elérte a maximumot, letiltjuk az ezzel az utasítással hivatkozott blokkok cache-be töltését. A táblázat persze nem lehet olyan nagy, hogy az utasításszámláló minden értékéhez külön bejegyzés tartozzon, ezért a gyakorlatban a táblázatot az utasításszámláló egy hash függvényével indexelik: ha a táblázat mérete kettő hatványa, akkor az utasításszámláló alsó bitjei erre a célra megfelelnek. Nem baj, ha hash ütközés miatt több utasítás is a táblázat ugyanahhoz az eleméhez van rendelve, ez "csak egy heurisztika", a rossz döntés nem okoz hibás működést, legfeljebb kicsit romló teljesítményt.

10.5.3. Idő előtti betöltés

Az idő előtti betöltés (prefetch) a modern cache menedzsment egy rendkívül fontos funkciója, amely, hatékony megvalósítást feltételezve, a processzor kihasználtságát növelni tudja azáltal, hogy a lehívott memóriaobjektumok már akkor a gyors elérésű cache-ben lesznek, amikor először szükség van rájuk.

Eddig nem volt róla szó, de fontos körülmény, hogy a legtöbb modern processzor külön cache-t tartalmaz az utasításoknak és az adatoknak. Ezek tartalom menedzsmentje eltérő, hiszen az utasítások és az adatok más és más lokalitási elveket követhetnek. Idő előtti betöltésre mindkét esetben szükség van. Az utasítás cache esetén ez egyszerűbb feladat, hiszen az utasítások sokszor lineárisan egymás után következnek, nehezen kiszámítható viselkedés csak a feltételes ugró utasításokból ered. Az ilyen feltételes ugró utasítások viselkedését, az elágazás becslését máshol, a pipeline feldolgozással foglalkozó fejezetben tárgyaljuk.

Az adatok idő előtti betöltése nagyobb kihívást jelent, hiszen az adatokhoz való hozzáférés nem mindig követ szabályos, kiszámítható mintázatot. A legtöbb processzor erre a célra tartalmaz implicit támogatást, de szinte mindegyik ad lehetőséget explicit menedzsmentre is.

Explicit menedzsment alatt azt értjük, hogy rendelkezésre állnak olyan utasítások, melyekkel az alkalmazás kérheti/javasolhatja egy blokk idő előtti cache-be töltését.

Implicit menedzsment alatt azt értjük, amikor a processzor cache prefetch egysége spekulatív módon próbálja kitalálni, hogy az alkalmazásnak milyen adatokra lesz a jövőben szüksége. Egyszerű heurisztikák arra, hogy mikor és hogyan töltsünk be jó előre blokkokat a cache-be:

- Ha hivatkozás történik az *i.* blokkra (akár bent van a cache-ben, akár nincs), akkor betöltjük a cache-be az *i+1.* blokkot is (feltéve, ha még nincs bent), mert feltételezzük, hogy az alkalmazás a térbeli lokalitásnak megfelelően előbb utóbb ahhoz is hozzá akar majd férni.
- Ha hivatkozás történik az *i.* blokkra és az nincs bent a cache-ben, akkor az *i.* mellett betöltjük a cache-be az *i+1.* blokkot is (feltéve, ha még nincs bent), ismét a térbeli lokalitással indokolva a döntést

Ezek az egyszerű heurisztikák nem túl hatékonyak, érezhetően túl nagy adatforgalmat bonyolítanak, és a blokkok túl agresszív idő előtti betöltése kiszorít a cache-ből olyan blokkokat, melyeket tényleg célszerű lenne bent tartani. Jó lenne tehát ezeknél kifinomultabb módszereket kitalálni, melyek tényleg csak a szükséges blokkokat töltik be idő előtt.

Speciális hardver utasítások

Példaként felsoroljuk, hogy néhány architektúra milyen utasításokat kínál adatok idő előtti betöltésére, vagyis explicit prefetch menedzsmentre:

- x86: PREFETCHT0, PREFETCHT1, PREFETCHT2, PREFETCHNTA: behoznak egy blokkot a cache-be. Az első három utasítás között a P4 óta nincs különbség, a PREFETCHNTA pedig úgy hozza be a kívánt blokkot a cache-be, hogy azt mindig a több utas asszociatív cache a blokk által kijelölt halmazának első blokkjába helyezi el. Ezzel elkerülhető a cache szemetelés is, hiszen ez az utasítás a halmazok többi blokkjából nem képes hasznos adatot kiszorítani.
- PowerPC ([24]): dcbt (Data Cache Block Touch), dcbtst (Data Cache Block Touch for Store): egy blokk cache-be töltése olvasásra/írásra
- PA-RISC ([27]): LDD, LDW: egy blokk cache-be töltése írásra/olvasásra
- Itanium ([3]): lfetch: utasítás egy blokk cache-be töltésére
- Alpha ([15]): Ha egy normál adatmozgató utasításnál célként az R31 regiszter van megjelölve, akkor azt a processzor egy idő előtti betöltés kérésnek értelmezi

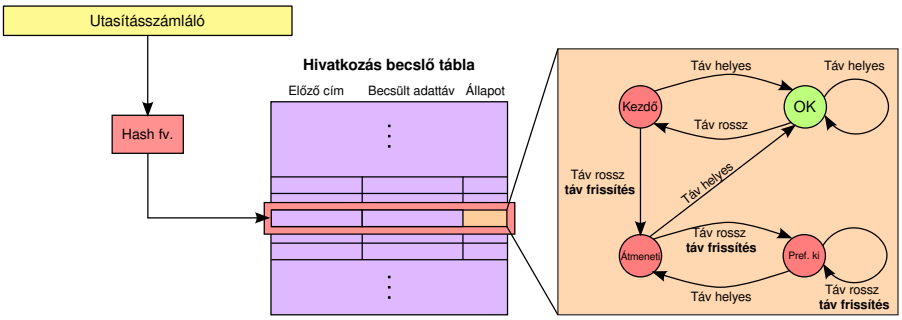
A GNU C fordító (gcc) fordító tartalmaz egy beépített függvényt: "`__builtin_prefetch`", amely a fordító által támogatott platformokon platformfüggetlen lehetőséget kínál adatok idő előtti cache-be töltésére.

Explicit menedzsmenttel (a felsorolt utasításokkal) akkor érdemes élni, ha az alkalmazás az adateléréseit olyan minta szerint végzi, amit a processzor nem tud kiszámítani (ehhez tehát elkerülhetetlen megismerni a processzor implicit menedzsment megoldását is).

Hivatkozásbecslő tábla használata

A számos implicit prefetch eljárás közül bemutatunk egyet, amihez hasonló talán a jelenleg elérhető processzorokban a legelterjedtebben használatos. Ez az eljárás ekvidisztáns (egymástól azonos távolságra lévő) adatok ciklikus feldolgozását képes idő előtti betöltéssel gyorsítani. Arról van tehát szó, hogy egy utasítás a memória *X.* bájtjának hivatkozása után az *X + táv*, majd az *X + 2 táv*, *X + 3 táv*, stb. bájtjait hivatkozza sorban egymás után (pl. egy struktúrából álló tömbön haladva végig minden elem bizonyos mezőjén műveletet végez). A bemutatott eljárás képes az adattávolságot (*táv*) megtanulni, és ezt felhasználva "előre dolgozni" a cache töltésével. A cache szemetelés elkerülésénél bemutatott eljáráshoz hasonlóan most is egy táblázatot tartunk karban (hivatkozásbecslő tábla), melyben nyomon követjük az egymás utáni memóriahivatkozások távolságát. Ha a távolság legalább egy rövid időre állandó, alkalmazzuk az idő előtti betöltést.

A működést a 10.10. ábrán láthatjuk ([14]). A hivatkozásbecslő táblában az utasításszámláló egy adott értékéhez egy hash függvény segítségével rendelünk hozzá egy bejegyzést. Ebben a bejegyzésben tároljuk az adott utasítás által legutóbb hivatkozott címet, az eddigi viselkedés alapján tapasztalt adattávolságot, valamint állapotinformációs biteket (jelen esetben kettőt). Az állapotinformációs bitek egy véges automata szerint változnak. Ez a véges automata követi nyomon, hogy az utasítás egymás utáni hivatkozásai között azonos-e a távolság.



10.10. ábra. Idő előtti betöltés hivatkozásbecslő táblával

Ha nem, akkor nem tudjuk megbecsülni a következő hivatkozott címet, tehát nem végzünk idő előtti betöltést (nem generálunk feleslegesen memóriaforgalmat, sem cache szemetet). Ha már néhány hivatkozás óta azonos adattávolságot tapasztalunk, vesszük a bátorságot és a következő elemet (elemeket) idő előtt behozzuk.

Pontosan ilyen elven működik az Intel Core i7 processzor prefetch egysége is (hangzatos neve: ”Intel Smart Memory Access IP Prefetcher”).

10.5.4. Cserestratégia

A cache üzemi állapota az, hogy tele van. Ebből következik, hogy egy új blokk betöltése egyben egy bent lévő blokk kirakásával jár (csere történik). Kérdés, hogy a bent lévő blokkok közül melyiket célszerű feláldozni, hogy a döntés a rendszer teljesítménye szempontjából a lehető legkedvezőbb legyen.

A lehetséges áldozatok száma (amely blokkok közül választani lehet) a cache asszociativitásával egyezik. Direkt leképzéses szervezés esetén nincs döntési helyzet, a behozott blokk sorszáma meghatározza, hogy hova kell kerülnie a cache-ben, a már ott lévő blokk pedig kirakásra kerül. Teljesen asszociatív szervezés esetén a behozott blokk bárhova kerülhet, vagyis a teljes cache-ből lehet áldozatot választani. Egy 4 utas asszociatív cache-ben pedig a behozott blokk sorszáma által meghatározott (indexelt) 4 lehetőség van az elhelyezésre.

A potenciális áldozatok közül többféle szempont szerint lehet kiválasztani azt a blokkot, aminek a helyére végül a behozott blokk kerül. Az optimális választáshoz sajnos a jövőbe kellene látni, hiszen azt a blokkot lenne a legjobb kirakni, amelyre a jövőben a legkésőbb (vagy sohasem) lesz hivatkozás. Ez nyilván nem implementálható stratégia. Tipikus, a gyakorlatban is használt cserealgoritmusok:

- Random: véletlenszerű választás,
- Round robin: körben forgó elv: először az 1-es potenciális áldozatot választjuk, a következő alkalommal a 2-est, majd a 3-ast, és így tovább,
- **Legrégebben használt** (Least recently used, LRU): azt a potenciális áldozatot rakjuk ki, amelyre a legrégebben történt hivatkozás (az időbeli lokalitási elv teljesülését feltételezve erre a blokkra lesz a legkisebb valószínűséggel szükség a jövőben),
- Nem a legutóbb használt (Not most recently used, NMRU): a legutóbb hivatkozott kivételével véletlenszerűen választunk áldozatot,
- Legritkábban hivatkozott (Least frequently used, LFU): a behozása óta legkevesebbszer hivatkozott blokkot választjuk áldozatnak.

A legnépszerűbb cserestratégia az LRU, de mivel azt elég nehéz pontosan implementálni (számolni kell a hivatkozásokat, a csere alkalmával pedig meg kell keresni a legkisebbet) ezért ennek különféle közelítéseit szokták használni (az NMRU is ilyen: könnyen megvalósítható, és a régebben hivatkozott blokkok közül választ).

10.5.5. Írási műveletek

Egyes korai cache megoldások ill. speciális rendszerek különbséget tesznek az írási és olvasási műveletek között úgy, hogy ha a cache-ben nem meglévő blokkra írási művelet történik, nem hozzák be a cache-be. Ezt a viselkedést

write-no-allocate stratégiának hívják. A legtöbb cache megoldás nem alkalmaz ilyen megkülönböztetést, írás esetén is behozzák a blokkot a cache-be, vagyis *write-allocate* stratégiát követnek.

Transzparens cache esetén, amikor a cache nem önálló objektumokat, hanem az operatív memória blokkjainak *másolatait* tartalmazza, a memóriára vonatkozó írás műveletekkel óvatosan kell bánni. A számítógépben nem feltétlenül a processzor az egyetlen szereplő, melynek hozzáférése van a memóriához. Ugyanazt a memóriát osztottan használhatja több processzor is, valamint a perifériák is (gondoljunk a DMA-ra). Fontos, hogy a memória és a cache koherens maradjon, vagyis mindig a legfrissebb, naprakész állapotot tartalmazzák.

Memóriába írás esetén az operatív memória egy blokkja és annak cache-beli másolata eltérő lesz. Egyfelől célszerű minden, a cache-be történő módosítást (írási műveletet) minél hamarabb átvezetni az operatív memóriába, másfelől jó lenne az ehhez szükséges lassú memóriaműveletek számát minimalizálni. Kétféle cache írási stratégiát különböztethetünk meg:

- *Write-through* stratégia: ha egy cache-ben lévő blokkra írási művelet történik, azt rögtön átvezetjük az operatív memóriába
- *Write-back* stratégia: az írási műveletek hatására nem kerül azonnal frissítésre az operatív memória, csak akkor, amikor a blokk kikerül a cache-ből

A write-through stratégia igyekszik a cache és az operatív memória tartalmát minél gyorsabban összhangba hozni, míg a write-back arra játszik, hogy az írási műveletek időben és térben lokálisak, tehát érdemes lehet bevárni több írási műveletet is mielőtt a módosításokat átvezetnénk az operatív memóriába. Egyes architektúrák az egyik, másik a másik stratégiát követik, mindkettőre van példa.

Mivel a cache jóval gyorsabb, mint az operatív memória, ezért az írási műveletek "átvezetése" nem úgy történik, hogy a processzor megáll, és kivárja, míg a lassú operatív memória végez a cache blokk beírásával. E helyett egy úgynevezett *írási buffert* (write buffer) alkalmaznak, és a frissítendő blokkokat ebbe a bufferbe (mely lényegében egy FIFO sor) helyezik. Az írási buffer az operatív memória komótos tempójában, a háttérben végzi a módosított blokkok frissítését, a processzor feltartása nélkül. A write-through stratégia tehát minden írási műveletkor, a write-back pedig a cache-ből való távozáskor helyezi a write-bufferbe a módosult blokkot (ha még nem volt ott). A write buffer mindig az operatív memória legfrissebb, de még nem aktualizált tartalmát tárolja, ezért memória olvasási műveletek során először a write buffer-ben kell körülnézni, hátha a keresett adat épp ott vár kiírásra.

10.5.6. Többszintű cache memóriák

A memóriaműveletek sebességének növelése érdekében gyakran alkalmaznak többszintű cache memóriát. Ha egy adat nincs meg az elsőszintű cache-ben, akkor a másodszintű, majd a harmadszintű, stb. cache-ben kell tovább keresni. A memóriaműveletet csak akkor szükséges az operatív memória felé továbbítani, ha a hivatkozott cím egyik cache-ben sem volt benne.

A különböző szintű cache memóriák méretben, szervezésben, és blokkméretben is eltérőek, hiszen más célt kell kiszolgálniuk.

- Az elsőszintű cache elsődleges célja a villámgyors adatelérés, azaz hogy a processzor a megcímzett adatot a lehető legkisebb késleltetéssel megkapja, így az utasítások végrehajtásában ne legyen fennakadás.
- A másodszintű cache célja a cache hiba arány minimalizálása. Ha már nem volt találat az elsőszintű cache-ben, elvárjuk, hogy a másodszintű cache, ha kissé nagyobb késleltetéssel is, de az esetek többségében találatot érjen el.
- A további cache szintek késleltetése egyre nagyobb, és ezzel együtt egyre nagyobb találati arányt várunk el tőlük.

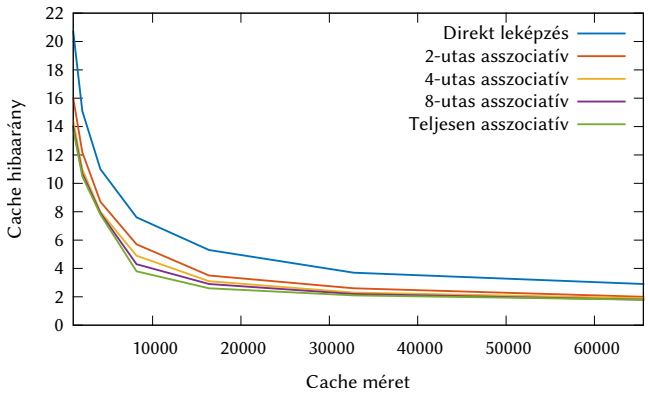
Ezen elvárások következtében az első szintű cache mérete a legkisebb, hiszen már csak jelterjedési időkre gondolva is beláthatjuk, hogy kisebb méret mellett kisebb késleltetést lehet elérni. A cache hierarchiában egyre lejjebb lépve nő a cache-ek tárolási kapacitása, és a találati arány növelése érdekében nő az asszociativitása.

10.6. Cache memóriák a gyakorlatban

10.6.1. Cache szervezések összehasonlítása

A `valgrind`[1] programcsomag `cachegrind` nevű eszköze lehetővé teszi éles (erre a célra előzetesen nem előkészített) programok cache viselkedésének szimulációját. Ezzel az eszközzel egy egyszerű alkalmazás cache viselkedését fogjuk megvizsgálni különféle cache méretek és szervezések mellett.

Ez az alkalmazás a GNU C fordító lesz[2], mellyel egy "hello world!" programot fordítunk le. Az eredményeket[3] a 10.11. ábra mutatja be. A cache méret növelésével a hiba arány gyors ütemben csökken. Szintén várható volt, hogy a direkt leképzés mellett kapjuk a legrosszabb, és teljesen asszociatív szervezés mellett pedig a legjobb eredményeket. Ami némiképp meglepő, hogy már a 2-utas eset is jelentős javulást hoz, a 8-utas pedig alig rosszabb a teljesen asszociatívnál. A 10% alatti cache hibaarányhoz a teljesen asszociatív és a 8-utas asszociatív szervezés mellett 2 kB méret is elegendő, míg a kevésbé asszociatív szervezések ehhez 4 kB cache memóriát igényelnek.



10.11. ábra. A GCC cache hiba-aránya a cache méret függvényében

A mérési eredmények alátámasztják azt az elterjedt gyakorlatot, hogy az első szintű cache memória méretét 32 kB körülire választják. A 64 kB-os méret a mi esetünkben is alig hozott javulást a 32 kB-oshoz képest, míg a 16 kB már számottevően rosszabbul teljesített, mint a 32 kB.

10.6.2. Cache szervezés és menedzsment néhány processzorcsaládban

Ebben a fejezetben áttekintjük néhány konkrét processzor cache szervezését, illetve az elsőszintű cache-ben alkalmazott menedzsment stratégiákat. Az összehasonlításban 2 x86 architektúrájú, PC-kben alkalmazott processzort, és 2 ARM architektúrájú, mobil eszközökben használatos SoC-t vetünk össze. Az x86 processzorok közül a Pentium 4 2002-ben, a Core i7-2600 pedig 2011-ben jelent meg. Az egyik ARM Soc egy meglehetősen régi, a Raspberry Pi-ben is ketyegő ARM1176JZF-S processzor 2002-ből, míg a Rockchip RK3188-as rendszerchipben dolgozó ARM Cortex-A9 magot 2009-ben jelentették be.

A 10.1. táblázat a cache szervezéseket mutatja be (most csak az adat cache-el foglalkozunk). Minden cache szinten 3 érték jellemzi a szervezést: az első a cache mérete, a második az asszociativitása (vagyis hogy hány utas asszociatív), a harmadik pedig a cache blokkok mérete. Jól látható a korábban előrevetített tendencia, a processzortól távolodva nő a méret és az asszociativitás. Az összehasonlításból kiugrik a Pentium 4 kicsi elsőszintű gyorsítótára, de a következő fejezet méréseiből látni fogjuk, hogy miért döntöttek így az Intel mérnökei: ennek a processzornak a legkisebb az L1 cache késleltetése. A Core i7-2600 értékein pedig tükröződik a gyártástechnológia fejlődése, ennek a legnagyobb az L1 asszociativitása, és a CPU mellé egyazon lapkára befért egy harmadszintű gyorsítótár is.

Az L1 (adat) cache menedzsment eljárásokat mutatja be a 10.2. táblázat. A "címek" oszlop arra vonatkozik, hogy a cache virtuális, vagy fizikai címekkel dolgozik-e. FF jelenti a fizikai indexek és fizikai tag-ek, VF pedig a virtuális indexek és fizikai tag-ek használatát. Látható, hogy a VF kombináció a legnépszerűbb megoldás. Az "írás" oszlopban a write-back (WB) és a write-through (WT) stratégiák szerepelhetnek. Az összehasonlításban szereplő legtöbb processzor mindkettőt támogatja, szabadon választható, hogy melyik legyen használatban. A

[1] `http://valgrind.org/`
[2] `gcc.gnu.org`
[3] Parancssor: `valgrind --tool=cachegrind --D1=<méret>,<asszociativitás>,64 gcc -o hellow hellow.c`

Processzor/SoC	L1 cache	L2 cache	L3 cache
Raspberry Pi	(16kB; 4; 32)	(128kB; 4; 32)	-
RK3188	(32kB; 4; 32)	(512kB; ?; 32)	-
Pentium 4 Northwood	(8kB; 4; 64)	(512kB; 8; 128)	-
Core i7-2600	(32kB; 8; 64)	(256kB; 8; 64)	(8MB; 16; 64)

10.1. táblázat. Néhány architektúra cache szervezésének paraméterei

lapcsere vonatkozásában feltűnő, hogy az egyszerűséget és kis fogyasztást előtérbe helyező ARM architektúrájú processzorok a véletlen, illetve körben forgó eljárásokat implementálták, míg az Intel processzorai az LRU-t (illetve annak közelítő változatát) alkalmazzák.

Processzor/SoC	Címek	Írás	Blokkcsere
Raspberry Pi	VF	WB/WT	Random/Round robin
RK3188	Ut.: VF, Adat: FF	WB/WT	Random/Round robin
Pentium 4 Northwood	VF	WB/WT	pszeudo-LRU
Core i7-2600	VF	WB	pszeudo-LRU

10.2. táblázat. Néhány architektúra elsőszintű cache menedzsment stratégiái

11. fejezet

Lokalitástudatos programozás

Az előző fejezetekben láttuk, hogy a memóriatartalom gyors elérésében kulcsfontosságú szerepet játszik a memóriahivatkozások lokális viselkedése. A lokalitást a tárhierarchia minden szintje kihasználja:

- A DRAM alapú rendszermemóriában a szomszédos címekhez tartozó nyitott sorok elérése gyors. Ha egy program össze-vissza címez a memóriában, akkor a sorlezárás és megnyitás műveletek miatt nő a memóriaműveletek késleltetése.
- A virtuális memória-kezelés során a TLB-ben szereplő lapok címfordítása gyors. Ha egy program össze-vissza címez a memóriában, akkor sok lesz a TLB hiba, amikor is a processzor lassú laptáblabejárásra kényszerül.
- A cache memóriában szereplő blokkok elérése gyors. Ha egy program össze-vissza címez a memóriában, akkor sok lesz a cache hiba, amikor is a processzor a lassú rendszermemóriához kénytelen fordulni.

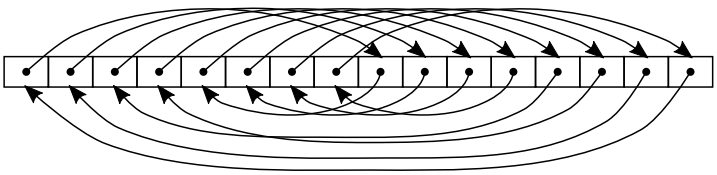
A processzor kihasználása érdekében tehát a programozónak a tárhierarchia keze alá kell dolgoznia, és a programot, amennyire ez lehetséges, úgy kell megírni, hogy a memóriahivatkozásokra minél inkább teljesüljenek a lokalitási elvek.

11.1. A tárhierarchia teljesítményének mérése

Elsőként megvizsgáljuk, hogy megéri-e egyáltalán lokalitás-tudatosan programozni. Megmérjük a tárhierarchia különböző szintjeinek sebességét, hogy megállapítsuk, mennyit nyerhetünk egy cache-barát programozási stílus elsajátításával.

11.1.1. Az időbeli lokalitás hatása

A memóriaműveletek késleltetésének méréséhez veszünk egy N méretű tömböt, mely a tömb különböző elemeire mutató mutatókat tartalmaz, lehetőleg véletlenszerű módon. A mutatók által egy olyan lánchoz jutunk, amely a tömb minden elemét ciklikusan bejárja, de a bejárásban semmi szabályosság nincsen (nehogy a cache prefetch algoritmusa rátanuljon). A 11.1. ábra egy ilyen láncot ábrázol, ami ugyan szabályosnak tűnik, de közelebbről megnézve nem az.



11.1. ábra. A tömb véletlen bejárásához használt mutatólánc

A mérést végző program nagyon egyszerű: egy ciklusban ki kell olvasni a tömb aktuális elemét, majd az ott talált címre kell lépni.

```
for (int i=0; i<iterations; i++)
    p = *p;
```

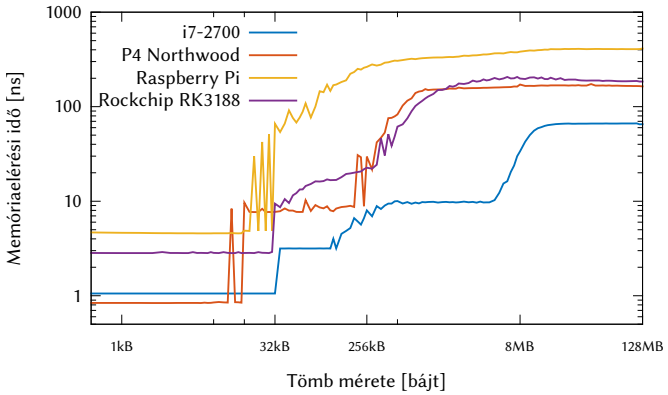
A ciklust elég hosszan futtatva, a futási időt az iterációk számával elosztva megkapjuk a memóriaelérések késleltetési idejét.

Annak érdekében, hogy a ciklusváltozó növelése, a ciklusfeltétel kiértékelése és az ugrás végrehajtása a lehető legkisebb mértékben befolyásolja a mérés eredményét, a ciklust kifejtjük, és minden körben sok, pl. 100 memóriaműveletet végzünk.

```
for (int i=0; i<iterations/100; i++) {
    p = *p;
    p = *p;
    ...
    p = *p;
}
```

Ha kis méretű mutatótömböt használunk a méréshez, akkor egy olyan program viselkedését utánozzuk, mely a memóriában egy szűk, jól behatárolható helyen végez műveleteket (pl. a lokális változóival dolgozik). A nagy tömbök rossz időbeli lokalitással jellemezhető, viszonylag nagy memóriatartománnyal operáló programok viselkedését modellezik.

Arra számítunk, hogy kis méretű tömb esetén, amikor a teljes tömb befér a cache memóriába, kedvezőbb késleltetést kapunk, mint nagy tömbök esetén, amikor a kesze-kusza pointer-lánc miatt minden egyes memória-hozzáférés cache hibát (és valószínűleg TLB hibát is) okoz. Az eredmények a 11.2 ábrán láthatók[1] (a görbék nem teljesen simák, mivel a mérés teljes zavartalansága, az összes zavaró körülmény kiiktatása nem valósítható meg egykönnyen).



11.2. ábra. A memóriakésleltetések a tömb méretének függvényében

A görbék jellege minden vizsgált architektúrára hasonló, és nagyon sokat elmond a tárhierarchiától. Amíg kis tömbökön mérünk, a memóriakésleltetés is alacsony, hiszen a teljes tömb befér a cache memóriába. A tömb méret növelésével egyszer csak átlépjük az elsőszintű gyorsítótár méretét, és egyre több cache hibát kapunk, amikor a másodszintű cache késleltetése lesz meghatározó. Még nagyobb tömbök már ezen is túllógnak, és a harmadszintű cache (ha van), a rendszermemória késleltetése, illetve a szaporodó TLB hibák határozzák meg a memóriaműveletek késleltetését.

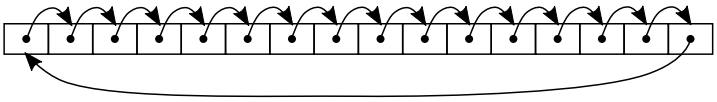
A görbéről, a közel vízszintes szakaszok behatárolásával, le lehet olvasni a tárhierarchia egyes szintjeinek méretét. Jól látható, hogy az i7 processzor és az rk3188 rendszercsip 32 kB, a raspberry pi processzora 16 kB, a p4 pedig 8 kB L1 cache memóriával rendelkezik. Az is tisztán látható, hogy az Intel miért döntött ilyen szokatlanul szűkös első szintű gyorsítótár mellett: a p4 L1 cache-ének késleltetése a nála 10 évvel fiatalabb modellénél is kisebb késleltetéssel rendelkezik. Az i7 processzor görbéjén jól azonosítható a 256 kB-os másodszintű, és a 8 MB-os harmadszintű gyorsítótár mérete is.

A mérési eredmények főbb üzenete azonban az, hogy az L1 cache késleltetése több nagyságrenddel alacsonyabb, mint a rendszermemóriáé. A p4 esetén a különbség 200-szoros!

[1] A méréshez az `lmbench` programot használtuk, a `-t` opcióval

11.1.2. A térbeli lokalitás hatása

Az iménti kesze-kusza tömb-bejárásról nem mondható el, hogy a térbeli lokalitás teljesülne rá. A térbeli lokalitás hatásának kimutatásához a tömb szabályos bejárása szükséges, a tömb mutatóiból ezért a 11.3. ábrán látható láncot alakítunk ki, azaz a mérés során a tömb elemein sorban, egymás után lépkedünk végig.

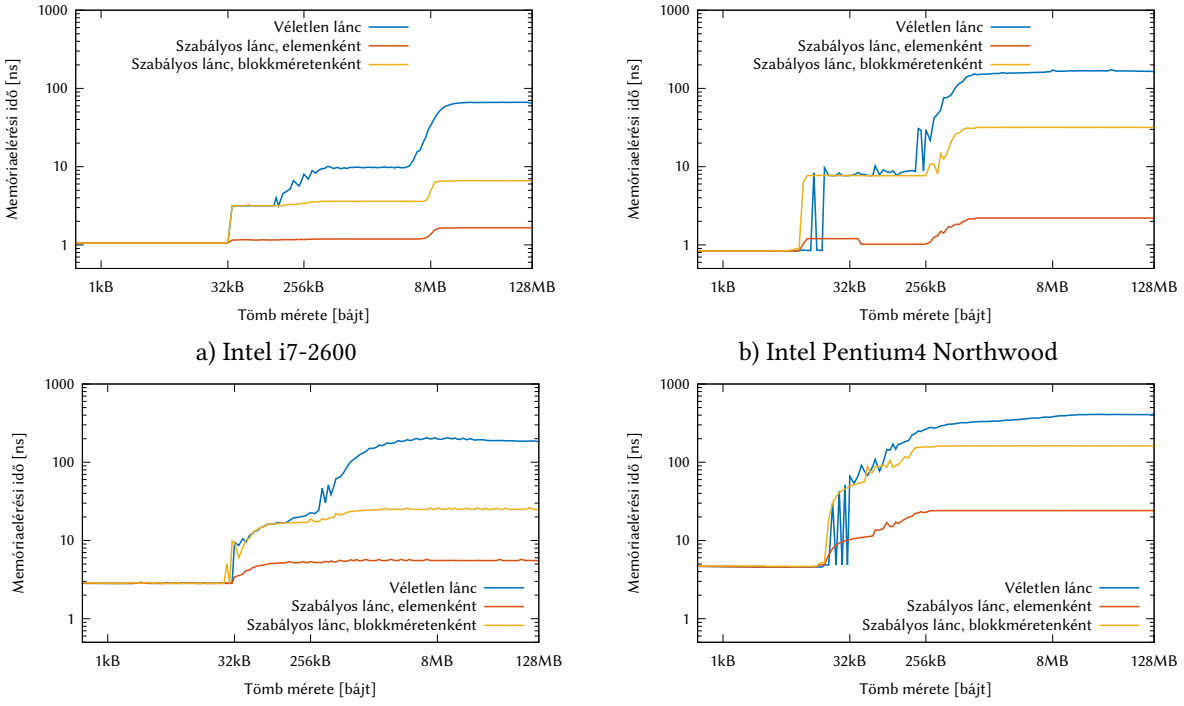


11.3. ábra. A prefetch hatékonyságának méréséhez használt pointer tömb

Várakozásaink szerint így alacsonyabb késleltetéseket kapunk, több okból is.

- Alacsony lesz a cache hiba-arány, hiszen ha a bejárás során egy blokk a cache-be kerül, a tömb soron következő elemei, melyekre a szabályos bejárás miatt úgyis szükség van, már a cache-ben lesznek.
- Alacsony lesz a TLB hiba-arány is. Amíg a tömb egy lapméretnyi tartományán végigsétálunk, addig ugyanazt a lap ↔ keret összerendelést kell használni.
- Ha a cache alkalmaz prefetch algoritmust, akkor az rátanul a szabályos bejárásra, és a blokkok idő előtti betöltésével még tovább csökkenti a memóriaműveletek késleltetését.

Azt az esetet is érdemes megvizsgálni, melyben a tömbön nem elemenként, hanem cache blokk méretű ugrásokkal haladunk végig. Így a bejárás során minden egyes hivatkozás újabb és újabb blokk fog esni, azaz a fenti 3 hatásból az elsőt ki lehet iktatni, és csupán az alacsony TLB hiba-arány, valamint a prefetch jótékony hatásának megfigyelésére is mód nyílik. A mérési eredményeket a 11.4. ábrán láthatjuk.



a) Intel i7-2600

b) Intel Pentium4 Northwood

c) Rockchip RK3188

d) Raspberry Pi

11.4. ábra. Memóriakésleltetések véletlen és soros bejárás mellett

A grafikonok alapján megállapíthatjuk, hogy minden architektúrában rendkívül kifizetődő a tömb szabályos bejárása, még akkor is, ha a tömb lényegesen nagyobb a cache memória méreténél. Nagy tömbméret mellett 40-80-szoros a gyorsulás a véletlen bejáráshoz képest. A nehezített esetben, amikor blokkonként lépünk, érthető módon kicsit rosszabbak az eredmények, de még mindig jelentős a javulás a tömb véletlen bejárásához képest, különösen az i7 processzor esetében (10-szeres), amely kifinomult prefetch logikát tartalmaz.

11.2. Lokalitásbarát ciklusszervezés

A programozói gyakorlatban számos alkalommal van szükség egy-, vagy többdimenziós tömbök bejárására, azokon különféle műveletek végzésére. A tömbök bejárásához ciklusok szükségesek, melyek a tömb elemeit valamilyen szisztéma szerint sorra veszik. Az előző fejezet mérési eredményeiből látható, hogy távolról sem mindegy, hogy ez a bejárás milyen szisztéma szerint történik. Ebben a fejezetben megismerkedünk néhány lokalitásbarát ciklusszervezési technikával, melyek segítségével a gyakorlatban is profitálhatunk eddigi, tárhierarchiával kapcsolatos ismereteinkből.

11.2.1. Ciklusegyesítés

A 11.5. ábra két C nyelvű programrészletet mutat be. Mindkettőnek ugyanaz az eredménye, de a b) esetben az összes műveletet egyetlen ciklus végzi el (loop fusion). Számoljuk ki mindkét megoldásra a cache hiba-arányt, ha N (a tömb mérete) nagy, feltéve, hogy a cache blokk méret 64 bájt, és a tömbelemek `double` típusúak (melynek mérete 8 bájt). Tegyük fel továbbá, hogy `i`, `c`, `x`, és `sum` elérése nem jár memória-hozzáféréssel, azokat a processzor regiszterei tárolják.

Az a) algoritmus első ciklusa végigsétál az `a` és `b` tömbökön. Ha cache hiba történik, 8 tömbelem kerül a cache-be, hiszen ennyi `double` fér el egy blokkba. Ez a ciklus tehát 8 lépésenként fog cache hibát okozni, mind az `a`, mind a `b` tömbök elérése során. A ciklusban összesen $2N$ memóriahivatkozás történt, $2N/8$ cache hibával. A következő ciklus ismét végigmegy a `b` tömbön. Ha N nagy, akkor a tömb első elemei már nem lesznek a cache-ben, hiszen azokat az előző ciklus során a tömb hátsó elemei kiszorították onnan. Ismét N memóriahivatkozást, és $N/8$ cache-hibát kapunk. Hasonlóan számolhatunk a harmadik ciklusban is, $3N$ memóriahivatkozást és $3N/8$ cache hibát kapunk. Összességében az a) algoritmus $6N$ memóriahivatkozást végez, amiből $6N/8$ jár cache hibával, vagyis a cache hiba-arány 1/8, ami 12.5%.

```
for (i=0; i<N; i++)
    b[i] = c * a[i] + x;
sum = 0;
for (i=0; i<N; i++)
    sum += b[i];
for (i=0; i<N; i++)
    d[i] = a[i] + b[i];
```

a) Eredeti kód

```
sum = 0;
for (i=0; i<N; i++) {
    b[i] = c * a[i] + x;
    sum += b[i];
    d[i] = a[i] + b[i];
}
```

b) Ciklusegyesítéssel

11.5. ábra. Példa a ciklusegyesítésre

Most vizsgáljuk meg a b) algoritmust. A ciklusmag első sora továbbra is 2 memóriahivatkozást tartalmaz, és továbbra is minden 8. lépésben vált ki cache hibát. A második sorban is van egy memóriahivatkozás, ez azonban soha nem okoz cache hibát, hiszen a `b[i]` értéke az első sor révén ekkorra már a cache-ben lesz. A harmadik sorban 3 memóriahivatkozás van, melyből `a[i]` és `b[i]` soha nem jár cache hibával, `d[i]` pedig 8 lépésenként vált ki cache hibát. A ciklus 1 körében tehát a 6 memóriahivatkozásából $3 \cdot 1/8$ cache hiba lesz. Összességében, az egész ciklus $6N$ memóriahivatkozásából $3N/8$ okoz cache-hibát, a cache hiba arány 6.25%, ami fele az a) algoritmusénak.

A 11.1. táblázatban látható, hogy ez az elméleti okfejtés hogy jelenik meg a gyakorlatban. Vannak olyan architektúrák, melyekben tényleg közel kétszeres a javulás, míg más architektúrákban sokkal kisebb a különbség. Ne felejtsük el, hogy a futási idő nem csak memóriahivatkozásokból áll, a lebegőpontos műveletek, és a ciklussal járó feltételes ugrások is számottevő végrehajtási időt igényelnek. Az tény, hogy a ciklusegyesítés mindenhol gyorsabb futási időt eredményezett.

Az első ökölszabály tehát, hogy kerüljük a tömbök ismételt bejárását. Amit lehet, egyetlen ciklusban, egyetlen érdemes megvalósítani.

11.2.2. Ciklusok sorrendjének optimalizálása

A 11.6. ábrán két algoritmus látható, mindegyik egy kétdimenziós tömb (`double` típusú) elemeit adja össze. A különbség csupán a ciklusok sorrendje. Az a) megoldásban a külső ciklus sorról sorra halad, és a második ciklus

	i7-2600	P4 Northwood	Raspberry Pi	RK3188
a) algoritmus	16.533 ms	109.974 ms	698.450 ms	115.354 ms
b) algoritmus	8.469 ms	84.917 ms	203.755 ms	97.126 ms

11.1. táblázat. A ciklusegyesítés hatása $N = 2^{22}$ esetén

adja össze az adott sor oszlopait, míg a b) megoldás pont fordítva működik, oszlopról oszlopra halad, és a belső ciklusával adja össze az aktuális oszlop elemeit.

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        sum += a[i][j];
```

a) Bejárás sor-folytonosan

```
for (j=0; j<N; j++)
    for (i=0; i<N; i++)
        sum += a[i][j];
```

b) Bejárás oszlop-folytonosan

11.6. ábra. Példa kétdimenziós tömb bejárására

A két megoldás tehát ugyanazt a tevékenységet végzi, de megfelelően nagy tömbméret (N) esetén futási idejük lényegesen különböző. A cache hiba-analízishez tudni kell, hogy a C programozási nyelvben a kétdimenziós tömbök a memóriában sor-folytonosan helyezkednek el. Így az a) algoritmus lényegében memória-folytonosan végigsétál a tömb összes ($N \times N$) elemén, az elejétől a végéig. A 11.1.2 fejezet megállapításai alapján ez a lehető legszerencsésebb bejárás, 64 bájtos blokkméret és 8 bájtos elemméret mellett ez 1/8-os cache-hibaarányhoz (12.5%) vezet. Ha a cache prefetch algoritmust használ, akkor még ennél is kevesebb lesz a cache hiba.

A b) algoritmus viszont a memóriában $N \cdot 8$ hosszú lépéseket tesz, hiszen `a[i][j]` és `a[i+1][j]` ilyen távol vannak egymástól. Ha $N > 8$, akkor a belső ciklus minden lépése cache hibát eredményez. Ha $N \cdot 8$ nagyobb, mint a cache mérete, akkor a helyzet még rosszabb, mert amikor a külső ciklus lép egyet a `j+1`-es oszlopra, az `a[0][j+1]` elem már nem lesz a cache-ben (addigra kiszorul onnan). Ebben az esetben tehát a két egymásba ágyazott ciklus minden egyes memóriahivatkozását cache hiba kíséri. A b) algoritmus cache hiba-aránya 100%.

	i7-2600	P4 Northwood	Raspberry Pi	RK3188
a) algoritmus	6.312 ms	8.973 ms	605.757 ms	14.879 ms
b) algoritmus	6.926 ms	160.78 ms	4363.129 ms	60.96 ms

11.2. táblázat. A ciklussorrend optimalizálásának hatása $N = 2048$ esetén

A valós mérési adatokat a 11.2. táblázat foglalja össze. Az i7-2600 processzor prefetch algoritmusa a b) esetben is képes volt megtanulni a bejárást (hiszen az is szabályos, csak nagyobbakat lép), ennek köszönhetően a ciklusok sorrendje nem volt olyan jelentős hatással a futási időkre, mint a többi processzor esetén. A p4 esetén az eltérés 18-szoros!

A fejezet tanulsága az, hogy a ciklusok sorrendjét az adatszerkezet memóriabeli elhelyezkedéséhez igazodva célszerű meghatározni, ha erre van lehetőség.

11.2.3. Blokkos ciklusszervezés

A 11.7. a) ábrán látható algoritmus az `a[N][N]` mátrix transzponáltját (átlójára való tükrözését) állítja elő a `b[N][N]` mátrixban. A kettős ciklus magjában mind az `a[i][j]`, mind az `b[j][i]` elemre hivatkozás történik. Az `a[i][j]` elemekre történő hivatkozások a tömb elemeit sor-folytonosan éri el. Ha egy cache blokkba a tömb 8 eleme fér el (64 bájtos blokk mérettel és 8 bájtos `double` mérettel számolva), akkor a sor-folytonos elérés minden 8. hivatkozáskor okoz cache hibát (lásd 11.2.2. fejezet). A `b[j][i]` hivatkozások azonban oszlop-folytonosan haladnak végig a tömbön, melyek, ha N elég nagy, mindig cache hibával járnak. Összességében a $2 \cdot N^2$ memóriahivatkozásból $N^2/8 + N^2$ cache hiba adódik, tehát a cache hiba-arány 56.25%.

A futási időn ezúttal úgy javítunk, hogy a ciklusok által megvalósított mátrixműveletet blokkonként végezzük el (a 11.7. b) ábra), vagyis a teljes mátrixban blokkról blokkra haladva hajtjuk végre a transzponálás műveletét

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        b[j][i] = a[i][j];
```

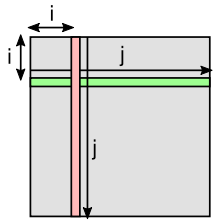
a) Eredeti kód

```
for (bi=0; bi<=N-BLK; bi+=BLK)
    for (bj=0; bj<=N-BLK; bj+=BLK)
        for (i=bi; i<bi+BLK; i++)
            for (j=bj; j<bj+BLK; j++)
                b[j][i] = a[i][j];
```

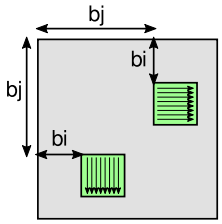
b) Blokkos ciklusszervezéssel

11.7. ábra. Példa a blokkos ciklusszervezésre

(loop tiling). Ha a blokkok méretét megfelelően sikerül megválasztani, akkor mind az `a[i][j]`-vel, mind pedig az `b[j][i]`-vel elért memóriatartalmak teljes egészében a cache memóriába férnek, és a blokk feldolgozásának végéig nem is szorulnak ki onnan (a blokkos és a blokkmentes megoldások memóriaelérési mintázatát a 11.8 ábra hasonlítja össze). Ezzel a megoldással a $2 \cdot N^2$ memóriahivatkozásból csupán $N^2/8 + N^2/8$ cache hiba keletkezik, tehát a cache hiba-arány most csak 12.5%.



a) Egyszerű ciklusszervezéssel



b) Blokkos ciklusszervezéssel

11.8. ábra. A mátrixtranszponálás memóriahivatkozásai egyszerű és blokkos ciklusszervezés esetén

Sajnos a blokk méret megfelelő megválasztása koránt sem egyszerű feladat; akár túl kicsire, akár túl nagyra választva visszakapjuk a blokkok nélküli algoritmus teljesítményét. Az optimális blokkméret architektúrafüggő, többek között függ a cache és a cache blokkok méretétől is. A 11.9. ábrán látható, hogy mekkora a blokkos ciklussal elérhető nyereség a vizsgált architektúrákon. Az i7 esetén a 8-as blokkméret az ideális, ennél az értéknél a futási idő csupán a nyolcada a blokk nélküli, nem lokalitástudatos megoldásnak. A Pentium 4 a legjobb futási időt 32-es blokkméretnél érte el, amikor is kevesebb, mint ötöde idő alatt végzett a transzponálással, mint blokkok nélkül.

11.2.4. Esettanulmány: lokalitásbarát mátrixszorzás

A lokalitásbarát szemlélet további elmélyítése érdekében megnézzük, hogy hogyan lehet egy, a mérnöki életben gyakran felbukkanó művelet, a mátrixszorzás algoritmusát a lehető leghatékonyabban megvalósítani.

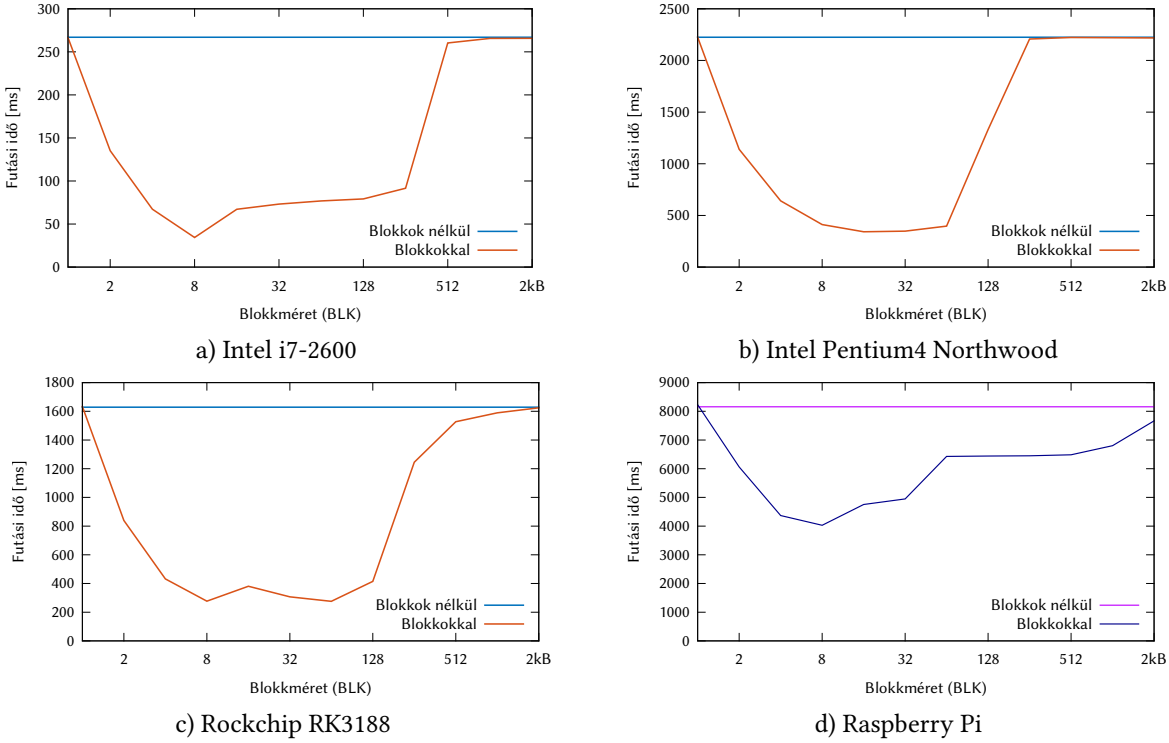
Két $N \times N$ méretű mátrix, $A = \{a_{ij}, i, j = 1, \ldots, N\}$ és $B = \{b_{ij}, i, j = 1, \ldots, N\}$ szorzata egy olyan C mátrix, melynek elemei

$$c_{ij} = \sum_{k=1}^{N} a_{ik} \cdot b_{kj}, \quad i, j = 1, \ldots, N.$$

A képletet alapján szinte minden programozó a 11.10. ábrán látható kóddal valósítja meg az algoritmust.

A ciklussorrend optimalizálása

Vizsgáljuk meg a naiv implementáció cache-beli viselkedését! A három egymásba ágyazott ciklus összesen N^3-ször hajtja végre a ciklusmag egyetlen sorát. A ciklusmag 4 memóriaműveletet tartalmaz: a `c[i][j]`, az `a[i][j]` és a `b[i][j]` kiolvasását, valamint a `c[i][j]` írását. A `c[i][j]`-t a két külső ciklus sor-folytonosan járja be, tehát a 11.2.2. fejezet eredményei alapján $N^2/8$ cache hiba keletkezik (64 bájtos cache blokkok és 8 bájtos elemméret esetén). Vegyük észre, hogy a legbelső ciklus `c` mindig ugyanazon elemét hivatkozza, az ismételt hivatkozások pedig csupa cache találatot jelentenek. Az `a[i][k]` kiolvasása $N^3/8$ cache hibával jár, hiszen ennek a mátrixnak a sorait is sor-folytonosan járjuk be (k szerint). A `b[k][j]` elérései azonban nem túl szerencsések: k egymást követő értékei a `b` egymást követő soraira vonatkoznak, melyek, ha a sorok elég hosszúak, mindig cache hibát



a) Intel i7-2600

b) Intel Pentium4 Northwood

c) Rockchip RK3188

d) Raspberry Pi

11.9. ábra. A blokkos ciklusszervezés jótékony hatása különböző architektúrák esetén, N=4096

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        for (k=0; k<N; k++)
            c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

11.10. ábra. A mátrixszorzás naiv megvalósítása

fognak jelenteni. A `b[k][j]` elérései összesen N^3 cache hibát váltanak ki. Az eredmény `c[i][j]`-be írása viszont sosem okoz cache hibát, hiszen ez az elem a korábbi kiolvasás miatt mindig a cache-ben lesz. Összességében tehát a $4N^3$ memóriaműveletből $N^2/8 + N^3/8 + N^3$ jár cache hibával, a cache hibaarány $(N^2/8 + N^3/8 + N^3)/(4N^3)$, mely $N \to \infty$ esetén $9/32 = 28.125\%$.

A memóriaműveletek lokalitásának javítása érdekében a 11.2.2. fejezet tanácsait megfogadva érdemes lehet optimalizálni a ciklusok sorrendjét. Mivel ez az algoritmus 3 egymásba ágyazott ciklusból áll, összesen 6-féle ciklussorrend létezik. A naiv implementációnak megfelelő (ijk) mellett megnézzük a többi sorrend viselkedését is.

Elsőnek lássuk az (ikj) ciklussorrendet. A `c[i][j]` elemek kiolvasása $N^3/8$ cache hibát okoz, hiszen sor-folytonos elérésről van szó, és az előbbi (ijk) sorrenddel ellentétben a legbelső ciklus sajnos nem ugyanazt az elemet hivatkozza. Az `a[i][k]` kiolvasása most is sor-folytonos, de mivel a legbelső ciklus mindig ugyanazt az elemet hivatkozza, csak $N^2/8$ hibával kell számolni. A `b[k][j]` elérései úgyszintén sor-folytonosak, $N^3/8$ cache hibával járnak. A $4N^3$ memóriahivatkozásból $N^3/8 + N^2/8 + N^3/8$ cache hibát kapunk, amiből az $N \to \infty$ határátmenettel $1/16 = 6.25\%$ adódik. Ez az érték kevesebb, mint negyede a kiindulási, naiv algoritmus cache hiba-arányának!

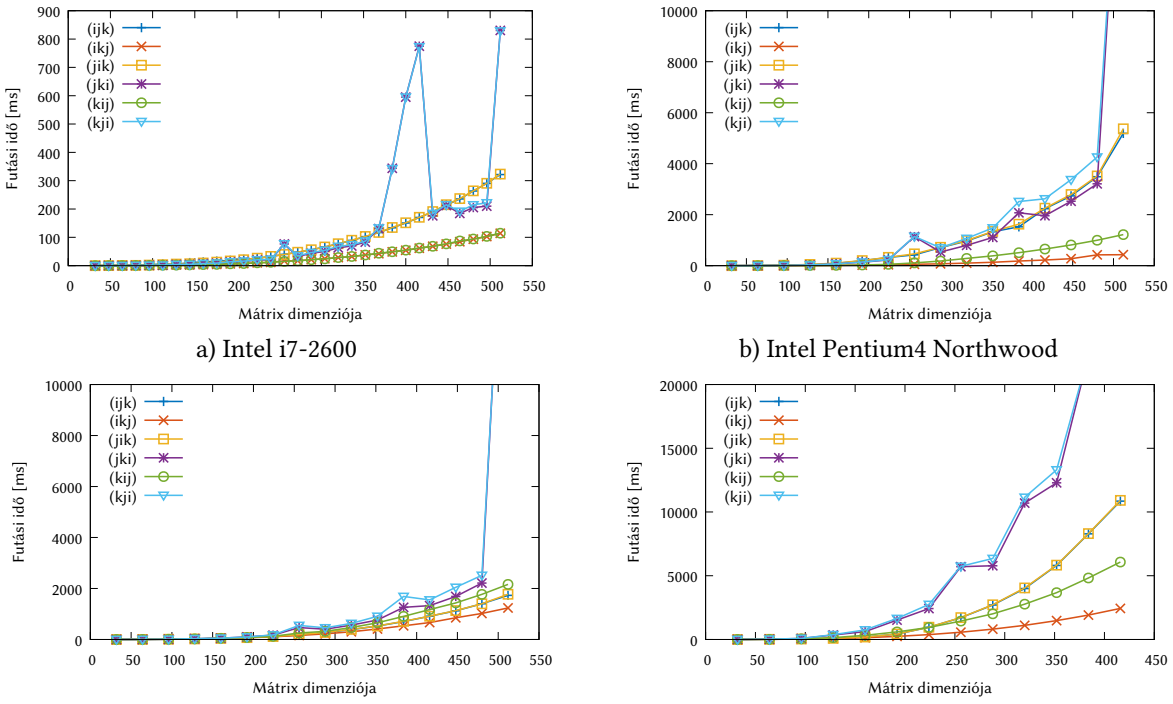
A többi ciklussorrendre is az eddigiekkel hasonló gondolatmenettel számíthatjuk ki a cache hiba-arányt, az eredményeket a 11.3. táblázat tartalmazza. Látható, hogy a naiv implementáció a lokalitás szempontjából nem a legjobb, de nem is legrosszabb. Mind aszimptotikusan, mind véges N értékekre az (ikj) sorrend a legjobb választás.

Az elméleti eredményeket gyakorlati mérésekkel is alátámasztjuk, a már megismert négy architektúrán keresztül. A 11.11. ábra grafikonjai a futási időket ábrázolják a mátrixméret függvényében. A grafikonokon látható, hogy vannak architektúrák, melyekben kisebb a lokalitás szerepe, például a kevésbé hatékony cache

Ciklussorrend	`c[i][j]` olv.	`a[i][k]`	`b[k][j]`	`c[i][j]` ír.	Cache hiba-arány
(ijk)	$N^2/8$	$N^3/8$	N^3	0	28.125%
(ikj)	$N^3/8$	$N^2/8$	$N^3/8$	0	6.25%
(jik)	N^2	$N^3/8$	N^3	0	28.125%
(jki)	N^3	N^3	N^2	0	50%
(kij)	$N^3/8$	N^2	$N^3/8$	0	6.25%
(kji)	N^3	N^3	$N^2/8$	0	50%

11.3. táblázat. A mátrixszorzó algoritmus cache hiba-analízise különböző ciklussorrendek esetén

menedzsment miatt (a két vizsgált ARM rendszerben nincs LRU blokkcsere, és prefetch sem!), vagy azért, mert a memóriaműveletek mellett a lebegőpontos számítások, illetve a ciklusok jelenléte is számottevő terhelést okoz. Az azonban egyértelműen látható, hogy minden architektúrában az (ikj) ciklussorrend bizonyult a leggyorsabbnak. (A néhány görbén tapasztalható ugrás nem mérési hiba, reprodukálható, de az okára nem sikerült rájönni.)



a) Intel i7-2600 b) Intel Pentium4 Northwood

c) Rockchip RK3188 d) Raspberry Pi

11.11. ábra. A mátrixszorzás futási ideje különböző ciklussorrendek esetén

Blokkos ciklusszervezés

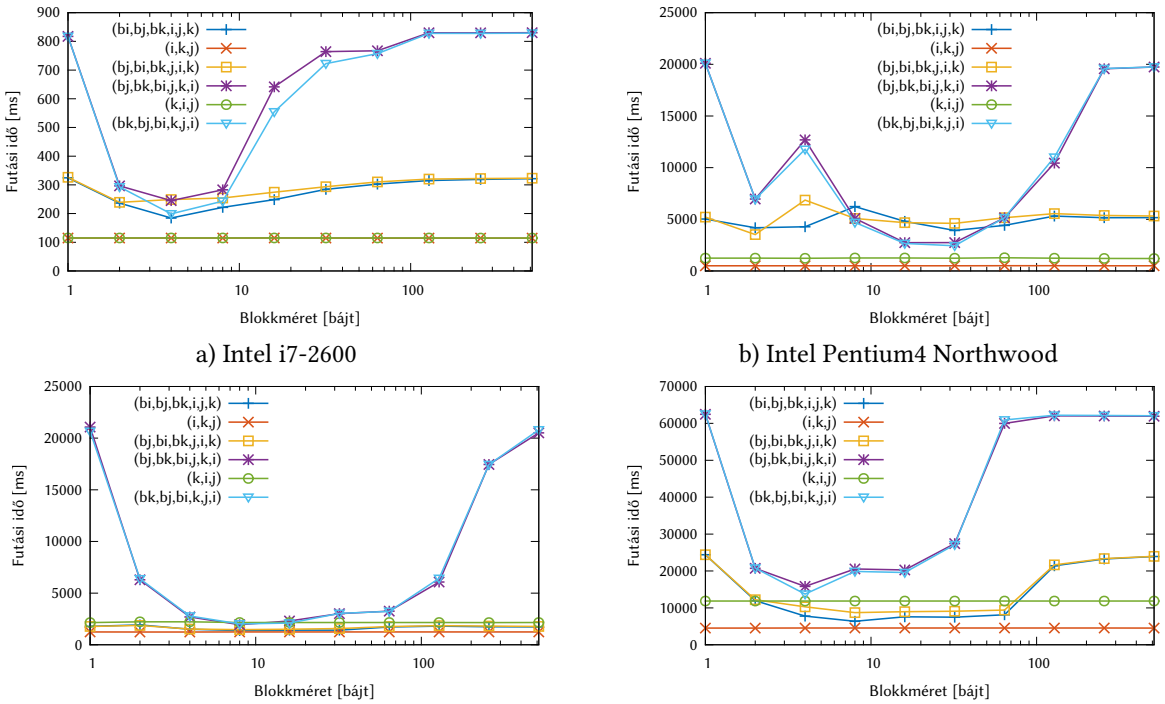
A három egymásba ágyazott ciklus blokkos szervezésére többféle lehetőség is adódik. A blokkos szervezés alkalmazható mind a három ciklusra egyaránt, valamint csupán két-két ciklusra korlátozva is. Az összes lehetőség kipróbálása és összehasonlítása arra vezetett, hogy a legjobb eredményt mindhárom ciklus blokkossá tétele adja. Például az (ijk) ciklussorrend blokkos változata a 11.12. ábrán látható.

A két legjobb ciklussorrend esetén (ezek az (ikj), illetve a (kij)) a ciklusok blokkossá tétele egyáltalán nem hozott javulást, a többi sorrend esetén azonban a blokkméret függvényében kisebb-nagyobb mértékben gyorsabbá vált a program. Az eredményeket a 11.13. ábra mutatja be. Látható, hogy a korábban legjobbnak talált két

```
for (bi=0; bi<N; bi+=BLK)
    for (bj=0; bj<N; bj+=BLK)
        for (bk=0; bk<N; bk+=BLK)
            for (i=bi; i<bi+BLK; i++)
                for (j=bj; j<bj+BLK; j++)
                    for (k=bk; k<bk+BLK; k++)
                        c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

11.12. ábra. A mátrixszorzás megvalósítása blokkos ciklusszervezéssel

ciklussorrend blokkszervezés nélkül is gyorsabb, mint az összes többi blokkszervezéssel.



a) Intel i7-2600

b) Intel Pentium4 Northwood

c) Rockchip RK3188

d) Raspberry Pi

11.13. ábra. A mátrixszorzás futási ideje blokkos ciklusszervezés esetén

Konklúzióként elmondható, hogy bizonyos algoritmusoknál mind a ciklussorrend optimalizálása, mind a blokkos szervezés komoly mértékben képes csökkenteni a futási időt. Az azonban sajnos architektúrafüggő, hogy épp melyik technikát célszerű alkalmazni, és mekkora a nyereség.

12. fejezet

Számpéldák, feladatok a memóriakezelés témakörében

12.1. Feladatok a memóriatechnológiák témakörében

1. feladat

Egy számítógép rendszermemóriája egycsatornás, 64 bites adategységekkel rendelkező DDR1-DRAM-ra épül, melyben a burst méret 8 adategységnyire van beállítva.

A több bank-os és több rank-os felépítéstől most tekintsünk el.

A memória késleltetések legyenek az alábbiak (órajelek számában megadva):

- T_{RP} = 4 (a PRECHARGE parancs végrehajtásához szükséges idő)
- T_{RCD} = 5 (ennyi ideig tart egy sor megnyitása)
- T_{CAS} = 9 (egy nyitott sor egy oszlopának a kiolvasásához szükséges idő. Az olvasás parancs után ennyi idő múlva jelenik meg az *első* adat a modul adatbuszán)

A memóriavezérlőhöz beérkező 64 bájtos (tehát 1 burst-nyi) olvasási kérések az alábbi sor, oszlop koordinátákra vonatkoznak:

- (3. sor, 8. oszlop), (3. sor, 2. oszlop), (7. sor, 9. oszlop)

Kezdetben a 7. sor van nyitott állapotban. Az utolsó parancs után a memóriavezérlő nem zárja le a nyitott sort.

(a) Adja meg a fenti kérésekhez tartozó, memóriavezérlő által kiadott DRAM parancsokat (sorrendhelyesen), FCFS, valamint FR-FCFS ütemezés mellett! (Feltesszük, hogy a 64 bájtos kérések nem lógnak túl az adott soron.)

(b) Az FCFS ütemezést alapul véve hányadik órajelben jelenik meg a (3. sor, 8. oszlop) kérésre érkező első adat a memóriamodul adatbuszán? És a (3. sor, 2. oszlop) kérésre érkező adatátvitel mely órajelben záródik le?

Megoldás

(a) FCFS ütemezés esetén a 3. sor megnyitása előtt le kell zárni az épp nyitott 7. sort. FR-FCFS esetén azonban épp kapóra jön, hogy a 7. sor nyitva van, hiszen így a harmadik kérés előrehozható, és ezzel egy nyitás-zárás parancspár megspórolható. A parancsok sorrendje az alábbi:

FCFS ütemezés szerint:

```
PRECHARGE
ACTIVATE 3
READ 8
READ 2
PRECHARGE
ACTIVATE 7
READ 9
```

FR-FCFS ütemezés szerint:

```
READ 9
PRECHARGE
ACTIVATE 3
READ 8
READ 2
```

(b) Mielőtt a (3. sor, 8. oszlop) kérésre megérkezne az első adat, ki kell várni egy PRECHARGE késleltetést ($T_{RP} = 4$), egy ACTIVATE késleltetést ($T_{RCD} = 5$), valamint a READ parancs késleltetését ($T_{CAS} = 9$). Összesen tehát $4 + 5 + 9 = 18$ órajel múlva érkezik meg az első adat az adatbuszon. Ezután 4 órajel szükséges a burst átviteléhez (hiszen a DDR1 az órajel fel- és lefutó élén is képes adatot átvinni, a burst hossza pedig 8), és további 4 órajel a következő, (3. sor, 2. oszlop)-ra vonatkozó kérés utolsó adategységének átviteléhez. A (3. sor, 2. oszlop)-ra vonatkozó kérés utolsó adategysége tehát $4 + 5 + 9 + 4 + 4 = 26$ órajel múlva jelenik meg az adatbuszon. Vegyük észre, hogy az átlapolt feldolgozás miatt nem kellett még egy READ késleltetést figyelembe venni (lásd: 8.6. ábra).

12.2. Feladatok a virtuális tárkezelés témakörében

2. Feladat

Legyenek a virtuális címek 16 bitesek, a fizikai címek 15 bitesek, a lapméret legyen 2^{12}bájt = 4 kB méretű, a laptábla pedig legyen egyszerű egyszintes laptábla 8 bites bejegyzésekkel.

(a) A virtuális címekben hány bit tartozik a lapok azonosításához és hány a lapon belüli eltoláshoz? Rajzolja fel a virtuális címek tagozódását!

(b) Mekkora a teljes laptábla mérete?

(c) Hány lap fér a fizikai memóriába?

(d) Legyen a laptábla aktuális állapota a következő:

Laptábla:

	Valid:	Keret:
0:	1	5
1:	1	7
2:	0	?
3:	1	2
4:	0	?
5:	0	?
6:	1	1
7:	0	?
8:	1	6
9:	0	?
10:	0	?
11:	0	?
12:	1	0
13:	1	3
14:	0	?
15:	1	4

Hol található a 3-mas, az 6-os és a 11-es lap?

(e) Módosítsa a laptábla tartalmát a következők szerint:

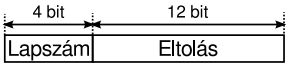
- A 6-os lap a háttértáron van
- A 11-es lap a fizikai memóriában az 1-es keretben található
- A 8-as lap a háttértáron van
- A 2-es lap a fizikai memóriában a 6-os keretben található

(f) Feltéve, hogy nincs laphiba, hány memóriaművelet kell a címfordításhoz

- TLB találat esetén?
- TLB hiba esetén?

Megoldás

(a) Mivel a lapok mérete 2^{12}, az eltoláshoz 12 bit tartozik. A virtuális címek fennmaradó $16 - 12 = 4$ bitje a lapot azonosítja. A kért rajz:



(b) Mivel 4 bit azonosítja a lapokat, $2^4 = 16$ lap van, mindegyikhez 8 bites (=1 bájtos) bejegyzés tartozik, vagyis a laptábla mérete 16 bájt.

(c) A lapok 12 bitesek, a fizikai memória 15 bites, tehát a keretek azonosítására 3 bit marad, vagyis $2^3 = 8$ lap fér a fizikai memóriába.

(d) Az egyszintes laptáblát nagyon könnyű kiolvasni. A 3. lap helyét a 3. bejegyzés adja meg, mely szerint ez a lap a fizikai memóriában van (hiszen valid=1), mégpedig a 2-es kereten. Hasonlóképpen a 6. bejegyzésből kiolvashatjuk, hogy a 6. lap az 1. kereten található. A 11-es laphoz tartozó bejegyzésben a valid=0 jelzi, hogy ez a lap a háttértárra került.

(e) A feladatban kért változtatásokat ugyanebben a szellemben kell elvégezni: a 6-os laphoz valid=0-t írunk, a 11-eshez valid=1-et, és 1-es keretet, stb. Végül az alábbi állapotot kapjuk:

Laptábla:

	Valid:	Keret:
0:	1	5
1:	1	7
2:	1	6
3:	1	2
4:	0	?
5:	0	?
6:	0	?
7:	0	?
8:	0	?
9:	0	?
10:	0	?
11:	1	1
12:	1	0
13:	1	3
14:	0	?
15:	1	4

(f) TLB találat esetén egy memóriaművelet sem kell (hiszen a TLB a processzorban van), TLB hiba esetén egyetlen memóriaművelettel kiolvasható a memóriából a címfordításhoz szükséges bejegyzés.

3. Feladat

Legyenek a virtuális címek 16 bitesek, a fizikai címek 15 bitesek, a lapméret legyen $2^{12} = 4$ kB méretű. Használjunk 2 szintű hierarchikus laptáblát, minden szintre egyforma széles indexekkel, 16 bites bejegyzésekkel.

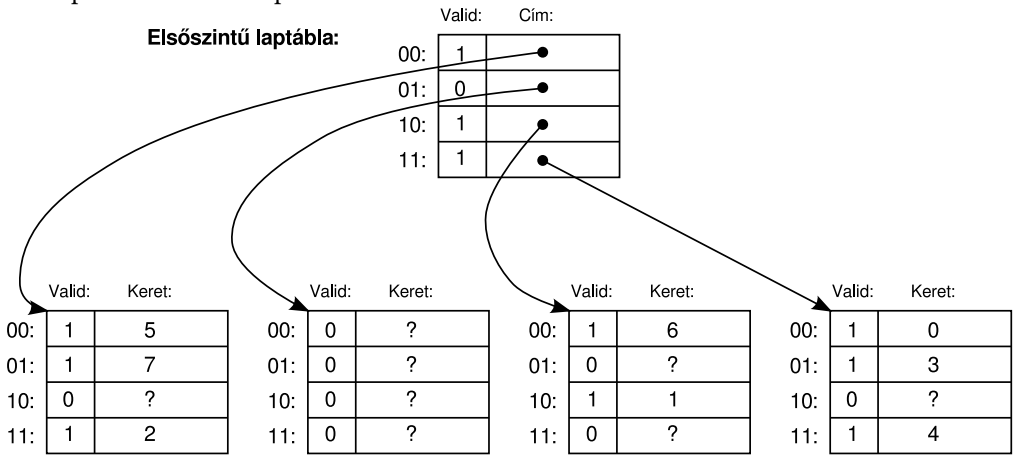
(a) A virtuális címekben hány bit tartozik a lapon belüli eltoláshoz és hány a különböző szintű laptáblák indexeléséhez? Rajzolja fel a virtuális címek tagozódását!

(b) Mekkora az összes laptábla összegzett mérete?

(c) Optimális körülmények között mennyi a laptáblák által elfoglalt memória minimális mérete? Mennyivel kisebb ez, mintha egyszintű laptáblát használnánk?

(d) Hány lap fér a fizikai memóriába?

(e) Legyen a laptábla aktuális állapota a következő:



Hol található a 3-mas (=0011), az 10-es (=1010) és a 11-es (=1011) lap?

(f) Módosítsa a laptábla tartalmát a következők szerint:

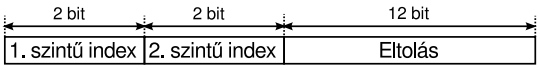
- A 10-es (=1010) lap a háttértáron van
- A 11-es (=1011) lap a fizikai memóriában az 1-es keretben található
- A 8-as (=1000) lap a háttértáron van
- A 2-es (=0010) lap a fizikai memóriában a 6-os keretben található

(g) Feltéve, hogy nincs laphiba, hány memóriaművelet kell a címfordításhoz

- TLB találat esetén?
- TLB hiba esetén?

Megoldás

(a) Mivel a lapok mérete 2^{12}, az eltoláshoz 12 bit tartozik. Mivel a laptáblák mérete minden szinten egyforma, a virtuális címek fennmaradó $16 - 12 = 4$ bitje kétfelé oszlik, a kétszintű laptábla mindkét szintjén 2 bitet használunk indexelésre.



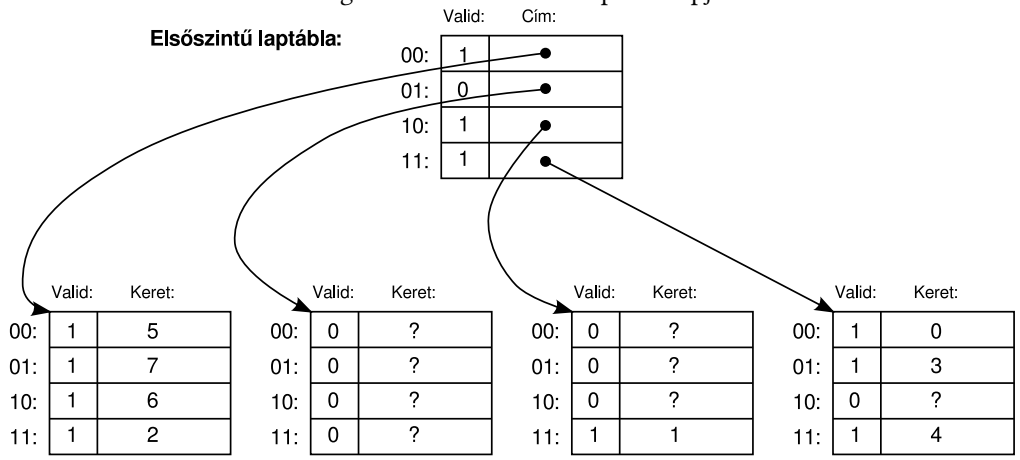
(b) Mivel 2 bites laptábla indexeink vannak, egy laptábla mérete = 2^2*bejegyzés mérete = 8 bájt. Első szintű laptáblából 1 db van, másodszintűből 4 db, egyenként 8 byte-osak, összesen tehát $5 * 8 = 40$ bájtot foglal az összes laptábla együttvéve.

(c) A minimumot úgy kapjuk meg, ha minden hierarchia szintből csak egyetlen egy van a fizikai memóriában: $2 * 8 = 16$ bájt. Egyszintű esetben az egyetlen laptáblának mindig a memóriában kell lennie. Ha egyszintű lenne a laptábla, $2^4 = 16$ bejegyzés kellene, tehát a laptábla mérete mindig $16 * 2 = 32$ bájt lenne. Ez ugyan kisebb, mint a kétszintű maximális mérete (ami 40 bájt), de jóval nagyobb, mint a kétszintű minimális mérete.

(d) Az eltolás mező 12 bites, a fizikai memória 15 bites, tehát a keretek azonosítására 3 bit marad, vagyis $2^3 = 8$ lap fér a memóriába.

(e) A kétszintű laptábla kiolvasása egy kétszintű fa bejárásához hasonlít. A 3-mas lap elhelyezkedésének kiderítéséhez először le kell választani az első- és a másodszintű laptábla indexeket. Mivel a 3 binárisan 0011, az elsőszintű laptáblaindex 00, vagyis 0, a másodszintű pedig 11, vagyis 3. A keresett laptábla bejegyzés így a 0. elsőszintű laptábla bejegyzés által mutatott másodszintű laptábla 3. bejegyzése. Itt valid=1 található, így a lap a memóriában van, mégpedig a 2-es kereten. Ugyanígy járunk el a 10-es bejegyzéssel is. Mivel a 10 binárisan 1010, mind az első, mind a másodszintű index 10, azaz 2. Az első szintű laptábla 2-es bejegyzése által mutatott másodszintű laptábla 2-es bejegyzésében valid=1-et, és 1-es keretet találunk. Végül, a 11-es laphoz a 10-es lap utáni bejegyzés tartozik, ahol a valid=0 jelzi, hogy ez a lap éppen a háttértáron van.

(f) A feladatban kért változtatások elvégzése után az alábbi állapotot kapjuk:



(g) TLB találat esetén egy memóriaművelet sem kell (hiszen a TLB a processzorban van), TLB hiba esetén viszont két memóriaműveletre van szükség, mire elérjük a másodszintű laptáblát, amiben a címfordításhoz szükséges keret száma megtalálható.

4. Feladat

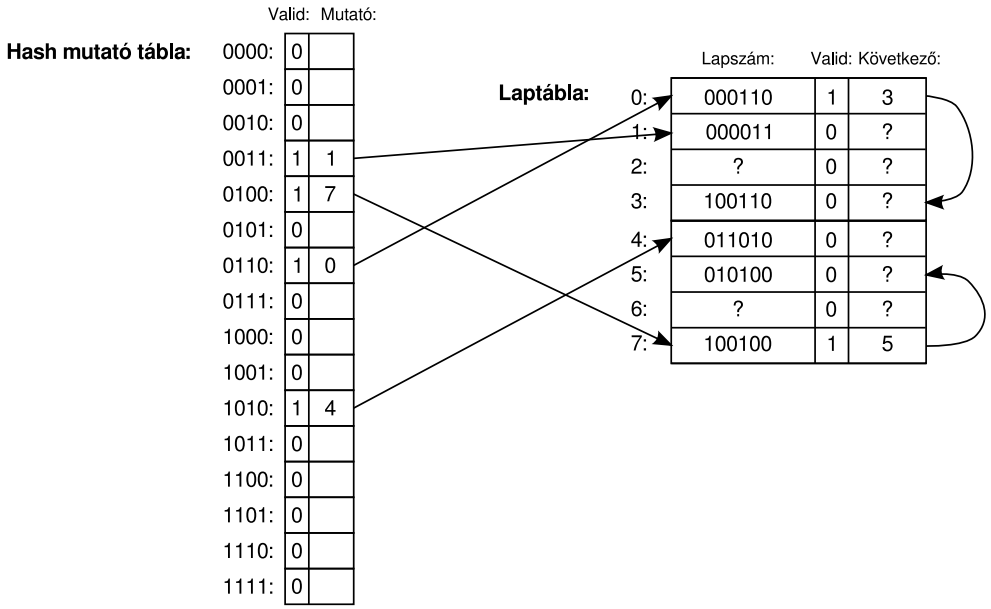
Legyenek a virtuális címek 16 bitesek, a fizikai címek 13 bitesek, a lapméret legyen $2^{10} = 1$ kB méretű. Használjunk inverz laptáblát, melyben a hash függvény értéke a virtuális cím 10.-13. bitje által meghatározott szám. A laptáblabejegyzések mérete 16 bit.

(a) A virtuális címekben hány bit tartozik a lapon belüli eltoláshoz és hány a lapok azonosításához? Rajzolja fel a virtuális címek tagozódását!

(b) Hány lap fér a fizikai memóriába?

(c) Mekkora a teljes laptábla mérete? Mennyivel kisebb ez, mintha egyszintű laptáblát használnánk? Mekkora a hash mutató tábla mérete?

(d) Legyen a laptábla aktuális állapota a következő:



Hol található a 000011-es, az 101011-es, a 010100-ás és a 001010-ás lap?

(e) Módosítsa a laptábla tartalmát a következők szerint:

- Az 101000 lap a fizikai memóriában az 6-os keretben található
- Az 100100 lap a háttértáron van
- Az 110110 lap a fizikai memóriában a 2-es keretben található

(f) Feltéve, hogy nincs laphiba, hány memóriaművelet kell a legjobb és a legrosszabb esetben a címfordításhoz

- TLB találat esetén?
- TLB hiba esetén?

Mikor áll elő a legrosszabb eset?

Megoldás

(a) Mivel a lapok mérete 2^{10}, az eltoláshoz 10 bit tartozik. A virtuális címek fennmaradó $16 - 10 = 6$ bitje szolgál a lapok azonosítására.

(b) Az eltolás mező 10 bites, a fizikai memória 13 bites, tehát a keretek azonosítására 3 bit marad, vagyis $2^3 = 8$ lap fér a memóriába.

(c) Az inverz laptábla jellegzetessége, hogy mérete a keretek számával azonos. Jelen esetben 8 keret van, tehát a laptábla mindössze 8 bejegyzésből áll, és mivel minden bejegyzés 2 bájtos, a laptábla mérete: $8 * 2 = 16$ bájt. Ha egyszintű lenne a laptábla, $2^6 = 64$ bejegyzés kellene, tehát a laptábla mérete mindig $64 * 2 = 128$ bájt lenne. A hash függvény a cím megadott 4 bitjéhez rendel hozzá egy laptábla indexet. Mivel 4 bitet használ, értékkészlete $2^4 = 16$, vagyis 16 bejegyzés szükséges. Minden bejegyzés egy laptábla index, és mivel csak 8 elemű a laptábla, a laptábla indexet 3 bittel le lehet írni. További 1 bitre van szükség, amivel jelezzük, hogy érvényes-e a mutató (nem NULL-e). Tehát a hash mutató tábla mérete: $16 * 4 = 64$ bit, $= 8$ bájt.

(d) A bejegyzések megkereséséhez először ki kell számítani a lapszámra a hash függvény értékét, amely jelen esetben a virtuális cím 10-13. bitje, vagyis a lapszám utolsó 4 bitje. A 000011-es lapra ez 0011 (decimálisan 3). A hash mutató tábla ezen bejegyzése érvényes elemet tartalmaz, mégpedig az 1-es laptábla bejegyzést. Nézzük tehát a laptábla 1-es bejegyzését, melynek a "lapszám" mezőjében pont megtaláltuk a keresett lapunkat, vagyis a 000011-et. A laphoz tartozó keret az 1-es, mivel a laptábla 1-es keretében találtuk meg ezt a bejegyzést.

Most lássuk a következő lapot, az 101011-át. Ennek hash értéke 1011 (decimálisan 11), melyre nincs érvényes hash tábla bejegyzés, vagyis ez a lap a háttértáron található.

A következő lap, a 010100-ás hash értéke 0100. A hash tábla a 7-es értéket tartalmazza, vagyis a laptábla 7-es bejegyzésében kell folytatnunk a keresést. A 7-es bejegyzésből látjuk, hogy az ott lévő lap nem az, amit keresünk (bár ugyanaz a hash értéke). Viszont a valid=1 mezőből megtudjuk, hogy a "következő" mező érvényes elemet tartalmaz, aminek mentén folytathatjuk a keresést. Jelen esetben az 5-ös bejegyzéssel. Az 5-ös bejegyzés "lapszám" mezője pedig megegyezik az általunk keresettel, tehát a 010100-ás lap az 5-ös kereten található.

Végül, a 001010-ás lap hash értéke 1010, mely a hash táblában érvényes, a 4-es bejegyzésre mutat. A laptábla 4-es bejegyzése nem a keresett lap, és a valid=0 miatt a keresést nem folytathatjuk a "következő" mező mentén, arra a konklúzióra jutunk, hogy ez a lap a háttértáron található.

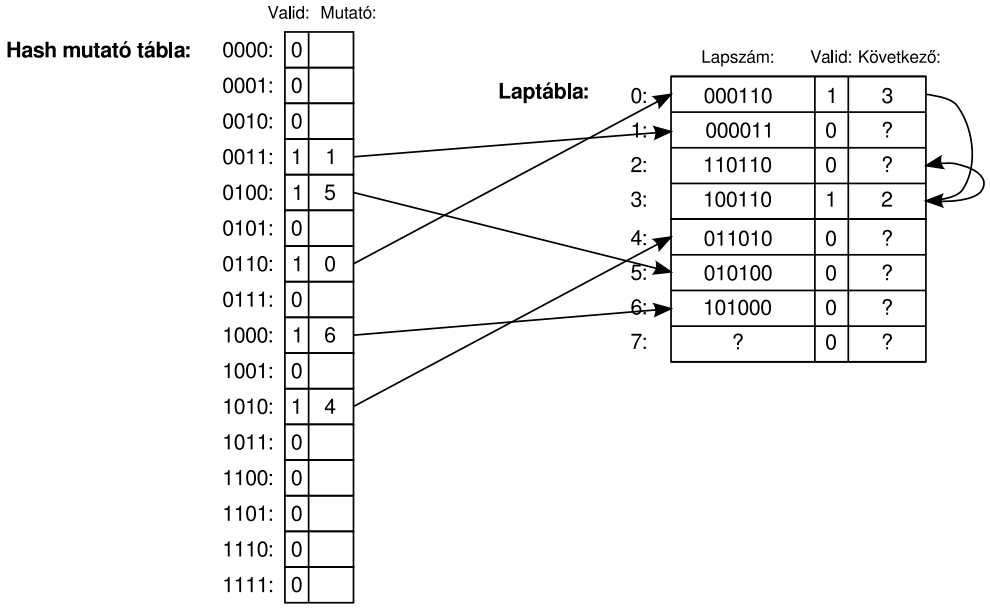
Általánosságban, a processzor címfordításkor ugyanezt az utat járja be: kiszámolja a hash értékét, és bejárja a láncolt listát, míg meg nem találja azt a keretet, amiben pont a keresett lap van.

(e) Az 101000 lappal könnyű a dolgunk, mert a hash tábla megfelelő eleme (1000) még nem tartalmaz érvényes bejegyzést. Mivel a lap a 6-os keretre kerül, a laptábla 6-os bejegyzésbe beírjuk a lap számát, és a hash tábla 1000-ás bejegyzésébe pedig 6 kerül.

Most az 100100 diszkre kerülését kell a laptáblában leadminisztrálnunk. A processzor a 0100 hash értékből kiindulva rögtön megtalálja a 7-es kereten, és ebből a bejegyzésből törli. Ebben a bejegyzésben azonban a valid=1 jelzi, hogy van még úgyszintén 0100 hash értékkel rendelkező lap, az 5-ös kereten. Így a hash táblát is frissíteni kell, hiszen ezentúl a 0100 hash értékkel kezdődő lapokat nem a 7-es, hanem az 5-ös bejegyzéstől kezdve kell keresni.

Végül be kell jegyeznünk, hogy a 110110 lap a 2-es kereten található. A laptábla 2-es keretébe beírjuk ezt a lapszámot. Ezután a processzor észleli, hogy a hash tábla 0110 bejegyzése érvényes, és végigjárja a láncot, kezdve a 0-ás bejegyzéssel, és folytatva a 3-mas bejegyzéssel. Ezt a láncot kellene kiegészíteni az új lapunkkal, azaz a 3-mas bejegyzésben, ahol a lánc végződik, a valid bitet 1-be állítjuk, és a "következő" mezőt a 2-es keretre irányítjuk. Így a processzor, amikor 0110 végű lapot keres, a mi 110110-ás lapunkhoz is eljut (igaz, hogy már három lépésben).

A három módosítás elvégzése után a laptábla tartalma:



(f) TLB találat esetén egy memóriaművelet sem kell (hiszen a TLB a processzorban van). TLB hiba esetén a legjobb esetben 2 memóriaművelet kell: 1 a hash mutató tábla kiolvasásához, 1 az általa kijelölt laptábla bejegyzés kiolvasásához (ahol a legjobb esetben pont a keresett lapszámot látjuk viszont). A legrosszabb helyzet abban a valószínűtlen esetben áll fenn, ha sok, a fizikai memóriában tárolt lap hash-e azonos. Mivel a lapszámnak csak 2 bitje nem vesz részt a hash képzésében, összesen 4 laphoz tartozhat azonos hash. Ekkor a

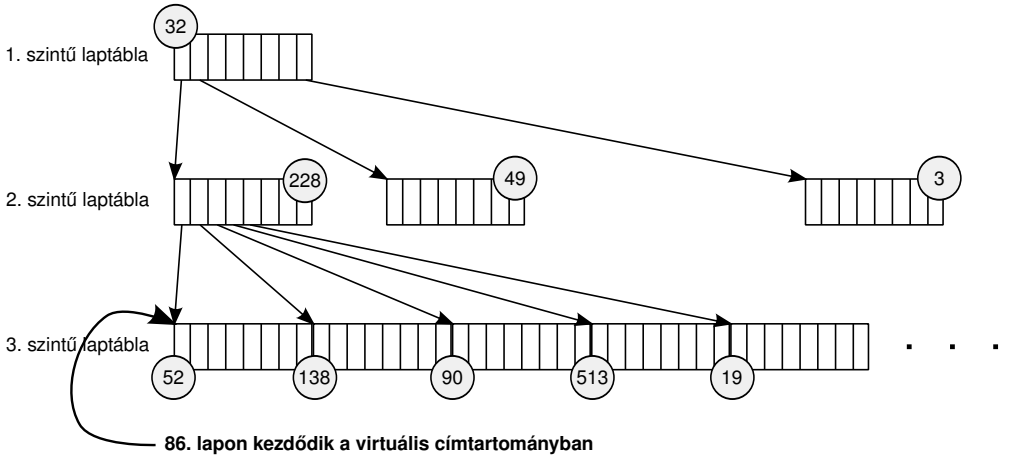
hash mutató tábla az első elhelyezett laptábla bejegyzésre mutat, ahonnét kezdve a láncolt lista segítségével be kell járni mind a 4 bejegyzést, mire végre a lánc végén meg nem találjuk a keresett lapot. Ahogy megyünk a láncban, minden érintett bejegyzésnél ellenőrizni kell, hogy abban a keretben épp a keresett lap van-e. A hash mutató tábla kiolvasásával együtt ez összesen 5 memóriaművelet.

5. Feladat

Egy processzor használjon 4 bejegyzéses TLB-t a címfordítás gyorsítására. A TLB egy adott pillanatban a következő két érvényes összerendelést tartalmazza:

Valid	Lapszám	Keretszám
1	88	90
1	1399	843
0	?	?
0	?	?

A laptábla legyen virtualizált, az alsó szint a 86. lapon kezdődik, egy szelete az alábbi ábrán látható (körökbe írt számok jelzik, hogy a laptáblák melyik keretben vannak):



Legyenek a virtuális címek 42 bitesek, az eltolás mező legyen 12 bites, a laptáblák minden szinten 1024 bejegyzést tartalmaznak, minden bejegyzés mérete 4 byte. A lapok mérete 4 kB. A fizikai címek 32 bitesek.

Hányszor kell a fizikai memóriához nyúlni, amikor az alábbi lapok címfordítását végzi a processzor? Melyik keret hányadik byte-ját és milyen sorrendben kell kiolvasni a fizikai memóriából a címfordítás kedvéért? Tételezzük fel, hogy minden érintett lap a fizikai memóriában van!

(a) 1399

(b) 2064

(c) 4101

Megoldás

(a) Az ehhez a laphoz tartozó összerendelés megtalálható a TLB-ben, tehát egyszer sem kell a fizikai memóriához nyúlni. Az 1399-es lap a 843-as kereten található.

(b) Először is kiszámoljuk a laphoz tartozó laptábla bejegyzés virtuális címét. A laptábla hierarchia legalsó szintje, ami a lap – keret összerendeléseket tartalmazza, a 86. lapon kezdődik. Mivel egy laptábla 1024 bejegyzést tartalmaz, a 2064 összerendelését tartalmazó bejegyzés a 88. lapon található (annak 2064-2048 = 16. bejegyzése). Most megnézzük a TLB-t, és örömmel látjuk, hogy a szükséges bejegyzés fizikai címét a TLB alapján ki tudjuk találni: a bejegyzés címe a 90-es keret 16*4-edik bájtja, vagyis 368704. Tehát a fizikai

memóriából kiolvassuk a 368704-ről a szükséges laptábla bejegyzést, amiből ki tudjuk olvasni a 2064-es laphoz tartozó keretet. Csak egyetlen egyszer kellett a fizikai memóriához fordulni!

(c) Akár az előbb, kiszámoljuk a 4101-as laphoz tartozó laptábla bejegyzés virtuális címét: a 90-es lap (86+4101/1024 egész része) 5. bejegyzése, vagyis a 90-es lap 20. bájtja. Sajnos a 90-es laphoz tartozó keretet a TLB nem tartalmazza. Nincs más választásunk: a hierarchikus laptábláknál megszokott módon végig kell sétálni az összes szinten, hogy megtaláljuk a bejegyzést: 3 szintünk van, 3 memóriaolvasásra lesz szükség. A 90-es lapot keressük: először az első szintű laptábla 0. elemét kell kiolvasni a memóriából (32-es keret legeleje: 32*4096 = 131072), majd ez tartalmaz egy mutatót az első második szintű laptáblára (228-as keret), ahol a 4-edik bejegyzés tartalmazza a mutatót arra a laptáblára, amely a 4096 és a 5120 közötti lapok összerendeléseit tartalmazza. Ez a 4. bejegyzés a 16. bájtnál kezdődik (228*4096+16 = 933904). Innen kiolvastuk a vonatkozó legalsó szintű laptáblához tatozó keret címét (19) és ennek az 5. bejegyzése lesz a címfordításhoz szükséges összerendelés (19*4096+5*4 = 77844). Ezt kiolvasva már el tudjuk végezni a címfordítást.

Tehát a fizikai memóriából az alábbi címekről kellett olvasni, ebben a sorrendben:

(a) Első szintű laptábla bejegyzés: 131072

(b) Második szintű laptábla bejegyzés: 933904

(c) Harmadszintű laptábla bejegyzés: 77844

12.3. Feladatok a TLB-vel segített virtuális tárkezelés témakörében

6. Feladat

Egy virtuális tárkezelésre képes processzor 16 bites virtuális, és 15 bites fizikai címeket támogat. A lapméret 4096 (= 2^{12}) bájt. A címfordításhoz egyszintű laptáblát használ, valamint egy 4 bejegyzéses, LRU algoritmussal menedzselt teljesen asszociatív TLB-t.

A futó program sorban egymás után az alábbi lapokra hivatkozik:

- 13, 12, 7, 2, 13.

A laptábla és a TLB tartalma kezdetben a következő:

Laptábla:

	Valid:	Keret:
0:	1	5
1:	1	7
2:	1	4
3:	1	2
4:	0	?
5:	1	1
6:	0	?
7:	0	?
8:	1	6
9:	0	?
10:	0	?
11:	0	?
12:	1	0
13:	1	3
14:	0	?
15:	0	?

TLB:

Valid:	Lap:	Keret:	Kor:
1	8	6	3
1	12	0	2
0	?	?	4
1	1	7	1

(a) Adja meg a laptábla és a TLB tartalmát a fenti laphivatkozások után! Ha az operációs rendszer egy újabb lapot akar a fizikai memóriában elhelyezni, akkor egy bent lévő lapot ki kell vennie onnan. A soron következő kiszemelt áldozatok legyenek az 1-es, majd az 5-ös lapok. Lapcsere esetén, ha a memóriából kikerült laphoz tartozik TLB bejegyzés, akkor az operációs rendszer azt az egy TLB bejegyzést érvényteleníti.

(b) A megadott hivatkozásokból hány esetén volt TLB találat?

(c) A megadott hivatkozásokból hány esetén volt laphiba?

Megoldás

(a) Először a 13-as lapot hivatkozzuk meg. Nincs bent a TLB-ben (TLB hiba), ezért a laptáblából kell kiolvasni (13-as bejegyzés), a 3-mas keret tartozik hozzá. Ezt a lap ↔ keret összerendelést a TLB-be is fel kell jegyezni, hátha a közeljövőben ismét lesz rá hivatkozás. A TLB-ben van egy érvénytelen bejegyzés (ahol valid=0), oda fogjuk beírni, miközben a ”kor” mezőket is frissítjük: az új bejegyzés kora 1 lesz, hiszen épp most volt rá hivatkozás, a többi bejegyzést pedig öregbítjük.

A következő, 12-es laphivatkozás TLB-ből feloldható, a TLB második sorából kiolvassuk, hogy a 0-ás keretben van. A kor mezőket frissítjük, az épp most hivatkozott sor lesz a legfrissebb (kor=1).

A 7-es lap nincs a TLB-ben, ezért a processzor a laptáblához fordul. A laptáblában a valid=0 bejegyzés jelzi, hogy a lap a háttértáron van, tehát meg kell hívni az operációs rendszert, hogy helyezze el a fizikai memóriában, hiszen szükség van rá. Az operációs rendszer a saját nyilvántartása alapján úgy dönt, hogy az 1-es lapot áldozza fel (a feladat megadta), azt kiírja a merevlemezre (laptáblába valid=0-t ír, a TLB-ben pedig érvényteleníti a hozzá tartozó sort), az így felszabaduló 7-es keretet pedig megkapja a 7-es lap. A 7-es lap bejegyzését is frissíti az operációs rendszer (valid=1, keret=7), végül visszaadja a vezérlést a processzornak, amely ezt az összerendelést a TLB-be is feljegyzi, a korábban az 1-eshez tartozó, érvénytelenített sorba.

A 2-es lap TLB hibát okoz, de a laptáblából kiolvassuk, hogy a 4-es keretben van, és ezt a TLB legöregebb bejegyzését felülírva tároljuk.

A 13-mas lap hivatkozása TLB találat lesz, a laptábla változatlan, a TLB-ben csak a ”kor” mezők változnak.

A laptábla végső állapota és a TLB bejegyzések az egyes hivatkozások után a következőképpen alakulnak:

Laptábla végállapot:

	Valid:	Keret:
0:	1	5
1:	**0**	**?**
2:	1	4
3:	1	2
4:	0	?
5:	1	1
6:	0	?
7:	**1**	**7**
8:	1	6
9:	0	?
10:	0	?
11:	0	?
12:	1	0
13:	1	3
14:	0	?
15:	0	?

13-mas után:

Valid:	Lap:	Keret:	Kor:
1	8	6	4
1	12	0	3
1	13	3	1
1	1	7	2

12-es után:

Valid:	Lap:	Keret:	Kor:
1	8	6	4
1	12	0	1
1	13	3	2
1	1	7	3

7-es után:

Valid:	Lap:	Keret:	Kor:
1	8	6	4
1	12	0	2
1	13	3	3
1	7	7	1

2-es után:

Valid:	Lap:	Keret:	Kor:
1	2	4	1
1	12	0	3
1	13	3	4
1	7	7	2

13-mas után:

Valid:	Lap:	Keret:	Kor:
1	2	4	2
1	12	0	4
1	13	3	1
1	7	7	3

(b) Két TLB találat volt, a második hivatkozás (a 12-es lapra), és az ötödik hivatkozás (a 13-mas lapra).

(c) Egy laphiba volt, a harmadik hivatkozás (a 7-es lapra).

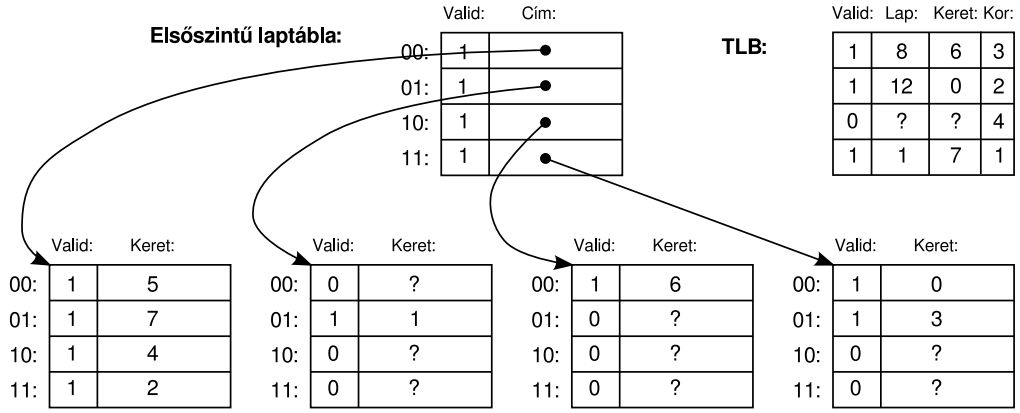
7. Feladat

Egy virtuális tárkezelésre képes processzor 14 bites virtuális és 13 bites fizikai címeket támogat. A lapméret 1024 bájt (=2^{10}). A címfordításhoz kétszintű laptáblát használ, valamint egy 4 bejegyzéses, LRU algoritmussal menedzselt teljesen asszociatív TLB-t.

A futó program sorban egymás után az alábbi lapokra hivatkozik:

- 13, 12, 7, 2, 13.

A laptábla és a TLB tartalma kezdetben a következő:



(a) Adja meg a laptábla és a TLB tartalmát a fenti laphivatkozások után! Ha az operációs rendszer egy újabb lapot akar a fizikai memóriában elhelyezni, akkor egy bent lévő lapot ki kell vennie onnan. A soron következő kiszemelt áldozatok legyenek az 1-es, majd az 5-ös lapok. Lapcsere esetén az operációs rendszer a TLB teljes tartalmát érvényteleníti.

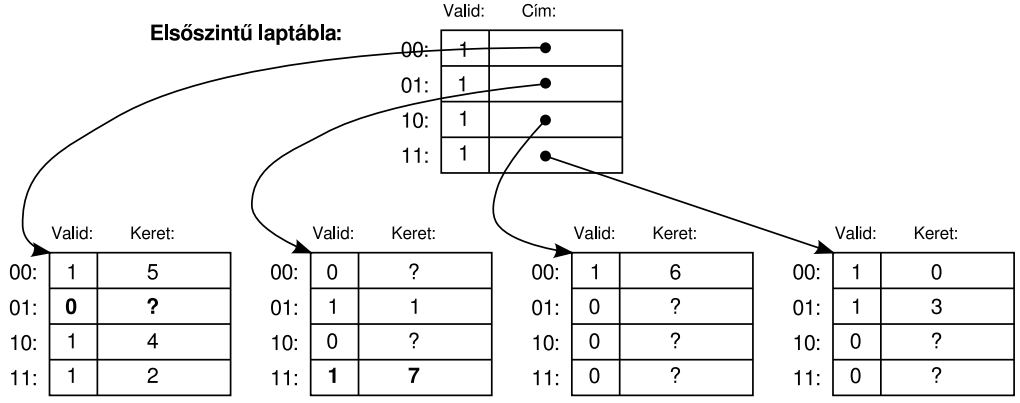
(b) A megadott hivatkozásokból hány esetén volt TLB találat?

(c) A megadott hivatkozásokból hány esetén volt laphiba?

Megoldás

(a) A megoldás hasonló az előző feladat megoldásához, hiszen a lap ↔ keret összerendelések, valamint a TLB kezdeti tartalma ugyanaz. Az egyik eltérés a laptábla adatszerkezet, melynek használatát a korábbiakban már megismertük, bejárása a lapszámok bináris alakjának ismeretében egyszerű (13=1101, 12=1100, 7=0111, 2=0010). A másik eltérés, hogy laphiba esetén most a teljes TLB-t érvényteleníteni kell.

A laptábla végső állapota és a TLB bejegyzések az egyes hivatkozások után a következőképpen alakulnak:



TLB:

13-mas után:

Valid:	Lap:	Keret:	Kor:
1	8	6	4
1	12	0	3
1	13	3	1
1	1	7	2

12-es után:

Valid:	Lap:	Keret:	Kor:
1	8	6	4
1	12	0	1
1	13	3	2
1	1	7	3

7-es után:

Valid:	Lap:	Keret:	Kor:
1	7	7	1
0	?	?	2
0	?	?	3
0	?	?	4

2-es után:

Valid:	Lap:	Keret:	Kor:
1	7	7	2
1	2	4	1
0	?	?	3
0	?	?	4

13-mas után:

Valid:	Lap:	Keret:	Kor:
1	7	7	3
1	2	4	2
1	13	3	1
0	?	?	4

(b) Egy TLB találat volt, a második hivatkozás (a 12-es lapra).

(c) Egy laphiba volt, a harmadik hivatkozás (a 7-es lapra).

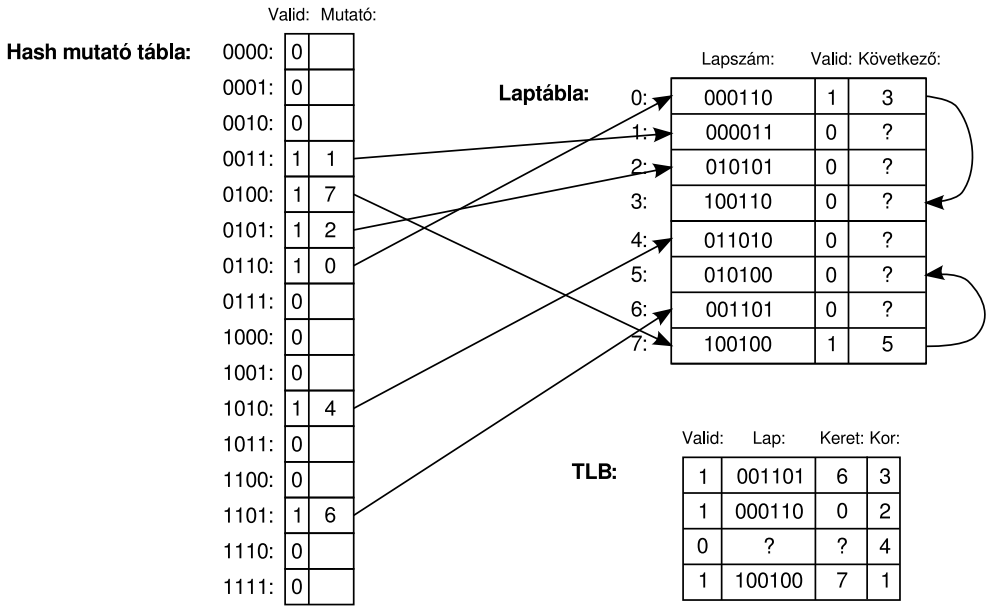
8. Feladat

Egy virtuális tárkezelésre képes processzor 16 bites virtuális és 13 bites fizikai címeket támogat. A lapméret 1024 bájt (=2^{10}). A címfordításhoz inverz laptáblát használ, melyben a hash függvény értéke a virtuális cím 10.-13. bitje által meghatározott szám. A TLB 4 bejegyzéses, teljesen asszociatív, LRU algoritmussal menedzselt.

A futó program sorban egymás után az alábbi lapokra hivatkozik:

- 010100, 000110, 110100, 000011, 010100.

A laptábla és a TLB tartalma kezdetben a következő:



(a) Adja meg a hash mutató tábla, a laptábla és a TLB tartalmát a fenti laphivatkozások után! Ha az operációs rendszer egy újabb lapot akar a fizikai memóriában elhelyezni, akkor egy bent lévő lapot ki kell vennie onnan. A soron következő kiszemelt áldozatok legyenek az 100110-ás, majd az 100100-ás lapok. Lapcsere esetén, ha a memóriából kikerült laphoz tartozik TLB bejegyzés, akkor az operációs rendszer azt az egy TLB bejegyzést érvényteleníti.

(b) A megadott hivatkozásokból hány esetén volt TLB találat?

(c) A megadott hivatkozásokból hány esetén volt laphiba?

Megoldás

(a) Az első lap, a 010100 nincs a TLB-ben, tehát meg kell keresni a laptáblában. A hash érték 0100, melyre a hash mutató tábla 7-et tartalmaz. A laptábla 7-es bejegyzésétől kezdve a láncolt listán haladva az 5. bejegyzésben találjuk meg a lapot. A TLB-be tehát, a harmadik (érvénytelen) sorba beírjuk a 010100 ↔ 5 összerendelést, és frissítjük a ”kor” mezőket.

A második lap, a 000110 benne van a TLB-ben, tehát csak a ”kor” mezőket kell frissíteni, hiszen a 000110 lesz a legutóbb hivatkozott, a többi pedig hozzá képest öregszik.

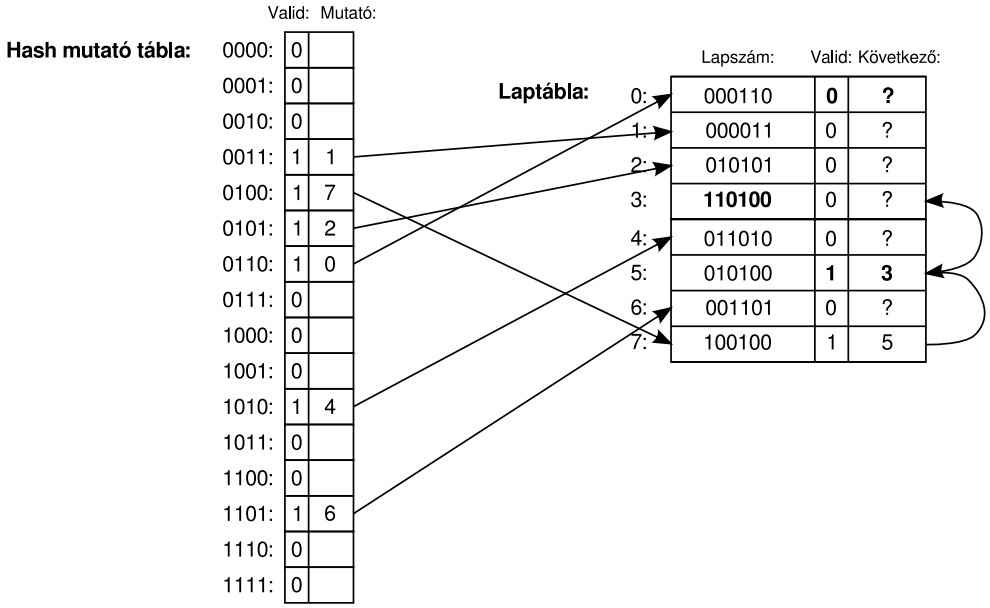
Az 110100 lap megkereséséhez a hash tábla által mutatott 7-es bejegyzéstől indulunk, de a láncolt lista bejárása után arra jutunk, hogy nincs bent a fizikai memóriában, laphiba történt. Mivel nincs szabad keret, áldozatot kell választani, és annak a helyére kell betenni. Az áldozat a feladat szerint az 100110 lap, melyet először el kell távolítani. Ez a lap szerepel a 0110 hash értékkel rendelkező lapok láncában, ahonnan most ki kell fűzni (a 0. laptábla bejegyzésbe valid=0 és ”következő”=? kerül). A helyére beírható az 110100, amit viszont be kell

fűzni a 0100 hash értékkel rendelkezők listájába (az 5. bejegyzésbe valid=1 és ”következő”=3 kerül). A TLB-t is frissíteni kell, a legrégebbi elem helyére eltároljuk az 110100 ↔ 3 összerendelést, és frissítjük a ”kor” mezőket.

A 000011 nincs a TLB-ben, de a 0011 hash érték mentén elindulva azonnal megtaláljuk a laptábla 1-es bejegyzésében. A TLB legrégebbi elemének helyére beírjuk, hogy 000011 ↔ 1, és frissítjük a ”kor” mezőket.

Végül az 010100 lap benne van a TLB-ben, csak a kor mezők frissítése szükséges.

A laptábla és a hash mutató tábla végső állapota:



A TLB bejegyzések az egyes hivatkozások után a következőképpen alakulnak:

010100 után:

Valid:	Lap:	Keret:	Kor:
1	001101	6	4
1	000110	0	3
1	010100	5	1
1	100100	7	2

000110 után:

Valid:	Lap:	Keret:	Kor:
1	001101	6	4
1	000110	0	1
1	010100	5	2
1	100100	7	3

110100 után:

Valid:	Lap:	Keret:	Kor:
1	110100	3	1
1	000110	0	2
1	010100	5	3
1	100100	7	4

000011 után:

Valid:	Lap:	Keret:	Kor:
1	110100	3	2
1	000110	0	3
1	010100	5	4
1	000011	1	1

010100 után:

Valid:	Lap:	Keret:	Kor:
1	110100	3	3
1	000110	0	4
1	010100	5	1
1	000011	1	2

(b) Két TLB találat volt, a második hivatkozás (a 000110), és az ötödik (010100).

(c) Egy laphiba volt, a harmadik hivatkozás (az 110100-ás lapra).

12.4. Feladatok a cache memória témakörében

9. Feladat

Vegyünk egy 256 byte méretű cache-t 64 byte-os blokkmérettel. A cache kezdetben csupa érvénytelen blokkot tartalmaz.

Egy program az alábbi memóriablokkokról olvas (ebben a sorrendben):

- 1, 3, 8, 4, 3, 6, 8, 1

Adja meg a cache hibák számát és a cache végső tartalmát LRU algoritmus mellett

(a) direkt leképzés esetén,

(b) teljesen asszociatív szervezés esetén,

(c) két utas asszociatív szervezés esetén.

Megoldás

(a) Direkt leképzés esetén az LRU-nak nincs szerepe. Azt, hogy egy blokk hova kerül a cache-be, a blokk számának 4-el való osztási maradéka határozza meg. Az egységes jelölés kedvéért az azonos indexű cache blokkokat egymás alá írjuk, így most a cache pillanatnyi állapota 4 egymás alá rajzolt blokkból áll. A cache tartalma az alábbiak szerint alakul.

Valid	Blokk
0	?
1	1
0	?
0	?

Valid	Blokk
0	?
1	1
0	?
1	3

Valid	Blokk
1	8
1	1
0	?
1	3

Valid	Blokk
1	8 4
1	1
0	?
1	3

Valid	Blokk
1	8 4
1	1
0	?
1	3

Valid	Blokk
1	8 4
1	1
1	6
1	3

Valid	Blokk
1	8 4 8
1	1
1	6
1	3

Valid	Blokk
1	8 4 8
1	1
1	6
1	3

A 8 memóriaműveletből 6 cache hiba, és 2 cache találat volt.

(b) A teljesen asszociatív esetben már az LRU algoritmusnak is van szerepe, azt is nyomon kell követni, hogy melyik blokkot mikor hivatkoztuk utoljára. A ”tag”, vagyis a tárolt blokk száma mellett ezért a blokkok korát is tárolni és folyamatosan frissíteni kell. Új blokk érkezésekor, ha nincs szabad hely, a legöregebb helyére tesszük. Mivel a teljesen asszociatív szervezés esetén minden blokknak ugyanaz az indexe (nincs indexelés), a cache pillanatnyi állapotát egyetlen sorba, egymás mellé rajzolt 4 blokkal jelöljük. A memóriahivatkozások hatására a cache állapota a következőképpen alakul.

	Valid	Blokk	Kor	Valid	Blokk	Kor	Valid	Blokk	Kor	Valid	Blokk	Kor
1-es után:	1	1	1	0	?	2	0	?	3	0	?	4
3-mas után:	1	1	2	1	3	1	0	?	3	0	?	4
8-as után:	1	1	3	1	3	2	1	8	1	0	?	4
4-es után:	1	1	4	1	3	3	1	8	2	1	4	1
3-mas után:	1	1	4	1	3	1	1	8	3	1	4	2
6-os után:	1	6	1	1	3	2	1	8	4	1	4	3
8-as után:	1	6	2	1	3	3	1	8	1	1	4	4
1-es után:	1	6	3	1	3	4	1	8	2	1	1	1

Megint csak 2 cache találat volt.

(c) A két utas asszociatív esetben az első sor (0-ás index) 2 blokkja páros, a második sor (1-es index) 2 blokkja páratlan blokkokat tud tárolni. Először ez alapján kell kiválasztani a megfelelő sort, az LRU algoritmust csak a soron belüli két blokk közül való választáskor kell alkalmazni.

1-es után:

Valid	Blokk	Kor	Valid	Blokk	Kor
0	?	1	0	?	2
1	1	1	0	?	2

3-mas után:

Valid	Blokk	Kor	Valid	Blokk	Kor
0	?	1	0	?	2
1	1	2	1	3	1

8-as után:

Valid	Blokk	Kor	Valid	Blokk	Kor
1	8	1	0	?	2
1	1	2	1	3	1

4-es után:

Valid	Blokk	Kor	Valid	Blokk	Kor
1	8	2	1	4	1
1	1	2	1	3	1

3-mas után:

Valid	Blokk	Kor	Valid	Blokk	Kor
1	8	2	1	4	1
1	1	2	1	3	1

6-os után:

Valid	Blokk	Kor	Valid	Blokk	Kor
1	6	1	1	4	2
1	1	2	1	3	1

8-as után:

Valid	Blokk	Kor	Valid	Blokk	Kor
1	6	2	1	8	1
1	1	2	1	3	1

1-es után:

Valid	Blokk	Kor	Valid	Blokk	Kor
1	6	2	1	8	1
1	1	1	1	3	2

Ismét csak 2 cache találat történt.

10. Feladat

Vegyünk egy 32 kB méretű cache-t 64 byte-os blokkmérettel. A CPU rendelkezzen 32 bites fizikai, 48 bites virtuális címekkel.

4-utas asszociatív szervezés mellett hány darab és hány bites összehasonlítást kell végezni egy keresés során, ha

(a) Fizikailag indexelt cache-t használunk, fizikai tag-ekkel

(b) Virtuálisan indexelt cache-t használunk, virtuális tag-ekkel

Megoldás

Mindkét esetben a cím alsó 6 bitje az eltolás ($2^6 = 64$). A cache-be 32 kB / 64 byte = 512 blokk fér el. Ha 4-utas a szervezés, akkor az index egy adott értékéhez 4 blokk tartozik, vagyis 512/4 = 128 féle indexünk lehet, ami 7 bittel írható le.

(a) Ha a cache kizárólag fizikai címekre alapozott, akkor a tag-ek mérete 32-7-6 = 19 bit. Minden egyes kereséskor a 4-utas cache-ben az index által kijelölt 4 blokk 19 bites tag-eit kell összehasonlítanunk a cím megfelelő bitjeivel. Vagyis egyidejűleg 4 darab 19 bites összehasonlítást kell elvégezni.

(b) Ha virtuális tag-eket használunk, akkor a tag-ek mérete 48-7-6 = 35 bit. Minden egyes kereséskor 4 db 35 bites összehasonlításra van szükség.

11. Feladat

Vegyünk egy 512 byte méretű cache-t 64 byte-os blokkmérettel. A cache fizikai címeket használ mind az indexeléshez, mind a tag-ekhez. A fizikai címek 16 bit szélesek.

Egy program az alábbi memóriacímekről olvas (ebben a sorrendben):

- 13, 136, 490, 541, 670, 74, 581, 980

(a) Adja meg a fenti címekhez tartozó ”tag”, ”index” és ”eltolás” mezőket

- teljesen asszociatív szervezés
- direkt leképzés
- két utas asszociatív szervezés

esetén.

(b) Adja meg a cache végső tartalmát mind a 3 esetben, LRU algoritmus mellett! (A cache kezdetben csupa érvénytelen blokkot tartalmaz)

Megoldás

(a) Először is, az 512 byte-os méret 8 blokk tárolását teszi lehetővé. Mivel a blokkok mérete 64 byte, mindhárom esetben 6 bites lesz az eltolás mező. A további mezők mérete a cache szervezéstől függ:

- Teljesen asszociatív szervezés esetén nincs indexelés, minden blokk bárhova kerülhet. Index: 0 bit, tag: $16 - 6 = 10$ bit.
- Direkt leképzés esetén a cache-nek mind a 8 blokkja ”direktben” az index mezővel kerül kiválasztásra, tehát az index mező 3 bites. A maradék $16 - 3 - 6 = 7$ bit lesz a cache tag-ek mérete.
- Két utas asszociatív szervezés esetén minden blokk két helyre kerülhet a cache-ben. Tehát a cím indexe alapján a 4 darab 2 utas tároló egyikét kell kijelölni. A 4 féle indexhez 2 bit szükséges, a tag-ek tehát $16 - 6 - 2 = 8$ bit szélesek lesznek.

A memóriacímek felbontása ennek megfelelően történik. A blokk szám és az eltolás szétválasztásához a címet el kell osztani 64-el, az eredmény egész része a blokk száma, a maradék a blokkon belüli eltolás. A blokk számból direkt leképzés esetén 8-al, a 2 utas asszociatív esetben pedig 4-el való osztás maradékaként kapjuk az indexet, egész részeként pedig a cache tag-et. Ennek megfelelően:

	13	**136**	**490**	**541**	**670**	**74**	**581**	**980**
Eltolás:	13	8	42	29	30	10	5	20
Tag teljesen asszoc. esetben:	0	2	7	8	10	1	9	15
Tag direkt leképzéskor:	0	0	0	1	1	0	1	1
Index direkt leképzéskor:	0	2	7	0	2	1	1	7
Tag 2-utas asszoc. esetben:	0	0	1	2	2	0	2	3
Index 2-utas asszoc. esetben:	0	2	3	0	2	1	1	3

(b) Vegyük észre, hogy 8 blokknyi hely van a cache-ben, és mi pont 8 adatot akarunk elhelyezni, a teljesen asszociatív szervezés pedig semmilyen megkötést nem ír elő a blokkok elhelyezésére. Vagyis a helyes megoldás: teljesen asszociatív szervezés mellett a cache a végső állapotában az összes blokkot tartalmazza, tetszőleges sorrendben.

Direkt leképzés esetén a blokkok elhelyezése egyértelmű, az index meghatározza. Összesen 4-szer fordul elő, hogy korábban már szereplő blokkot felül kell írni, tehát csak 4 blokk lesz a végén a cache-ben.

2-utas asszociatív esetben 2 azonos indexű blokkot is tárolni tudunk. Ezzel együtt ebben a konkrét példában egyszer sem fordul elő, hogy felül kell írni egy korábbi blokkot, tehát a végén 8 blokk lesz a cache-ben.

A megoldáshoz tartozó ábra:

Direkt leképzés:

Valid	Tag	Blokk tartalom
1	1	
1	1	
1	1	
0	?	
0	?	
0	?	
0	?	
1	1	

2-utas asszociatív szervezés:

	Valid	Tag	Blokk tartalom
Index = 0	1	2	
Index = 1	1	2	
Index = 2	1	2	
Index = 3	1	3	

Valid	Tag	Blokk tartalom
1	0	
1	0	
1	0	
1	1	

IV. rész

A processzor

13. fejezet

Pipeline utasításfeldolgozás statikus ütemezéssel

13.1. Az utasításfeldolgozás fázisai

A hatékony utasítás-feldolgozás alapjainak bemutatásához egy egyszerű (hipotetikus) RISC utasításkészletet vezetünk be, melyet a következő fejezetekben demonstrációs célokra használunk. Az utasításkészlet tulajdonságai a következők:

- Minden utasítás hossza egyforma, 4 bájt.
- Az utasítások típusai:
  - Load/Store műveletek: a memóriából egy regiszterbe, ill. egy regiszterből a memóriába mozgatnak adatot. A címeket 2 paraméterrel kell megadni (ezek a Load/Store utasítások operandusai), egy bázisregiszterrel és egy eltolással, ez utóbbi egy skalár konstans. A kettő összege adja meg a címet a memória olvasáshoz, illetve íráshoz. Pl. $R_i \leftarrow MEM[R_k + 42]$ vagy $MEM[R_k + 42] \leftarrow R_i$.
  - Aritmetikai és bitműveletek: első operandusuk kötelezően egy regiszter, a második operandus lehet regiszter, vagy skalár konstans (immediate) is. Az eredményt regiszterben tárolják. Utasításkészletünk 3 operandusú műveleteket támogat, vagyis a forrásoperandusok mellett az eredményoperandust is explicit módon meg kell adni. Pl. $R_i \leftarrow R_j * R_k$, vagy $R_i \leftarrow R_k * 42$.
  - Vezérlésátadó műveletek:
    - Feltétel nélküli ugrás: az operandusként megadott, az utasításszámlálóhoz képest relatív címre ugrik. Az ugrási cím regiszter, vagy skalár konstans is lehet. Pl. JUMP -24, vagy JUMP R_i.
    - Feltételes ugrás: az ugrási cím mellett megjelenik egy második operandus is, amely egy regiszter. Az ugrás feltétele, hogy ez a regiszter nulla legyen. Pl. JUMP -24 IF $R_j == 0$.
- A Load és a Store kivételével nincsenek memóriareferens utasítások.

Mindezek ismeretében megadunk egy lehetséges módot az utasítások végrehajtásának részfeladatokra, fázisokra bontására. Ez nem feltétlenül egy minden szempontból optimális megoldás lesz, viszont a későbbiekben ebből kiindulva könnyedén fel tudjuk majd építeni az utasítást végrehajtó pipeline-t. Tehát egy utasítás végrehajtásának fázisai:

1. Az első lépés az **utasítás lehívása** (instruction fetch, *IF*) a memóriából. Ezután az utasításszámlálót megnöveljük az utasítás méretével, hogy az a következő utasításra mutasson.
2. A második lépés az **utasítás dekódolása** és a regiszterek kiolvasása (instruction decode/register fetch, *ID*). Ennek során az utasításszóból leválasztjuk az utasítás típusára és operandusaira vonatkozó részeket. Az operandusként hivatkozott regisztereket beolvassuk a regiszter tárolóból. Előkészítjük az ALU számára a vezérlő jeleket egyrészt a művelet típusára vonatkozólag (összeadást, kivonást, szorzást, osztást, összehasonlítást kell-e majd csinálni) másrészt az operandusok forrására vonatkozólag (azaz hogy honnan vegye az operandusokat: az imént beolvasott regiszterek értékét használja-e, vagy az utasításba beágyazott skalár konstanst, vagy ugró utasítás esetén az utasításszámlálót, stb).

3. A harmadik lépésben történik az ALU számára beállított **műveletek végrehajtása** (execution, *EX*). Ez az utasítás típusától függően mást és mást jelent:
   - Load/Store utasításnál az ALU összeadóját használva ki kell számítani a betöltés/írás memóriacímét (össze kell adni a bázisregiszter tartalmát az eltolással)
   - Aritmetikai utasításnál az ALU elvégzi a kívánt aritmetikai műveletet
   - Ugró utasításnál az ALU egyrészt kiértékeli az ugrási feltételt (ha feltételes ugrásról van szó), valamint kiszámítja az utasításszámláló új (ugrás utáni) címét

4. A negyedik fázisban végrehajtjuk az utasításhoz **kapcsolódó memóriaműveletek**et (memory access, *MEM*). A Load/Store utasítások esetén itt történik az adatok tényleges beolvasása/kiírása a memóriából/memóriába. Egyéb utasítások esetén ez a fázis kihagyható.

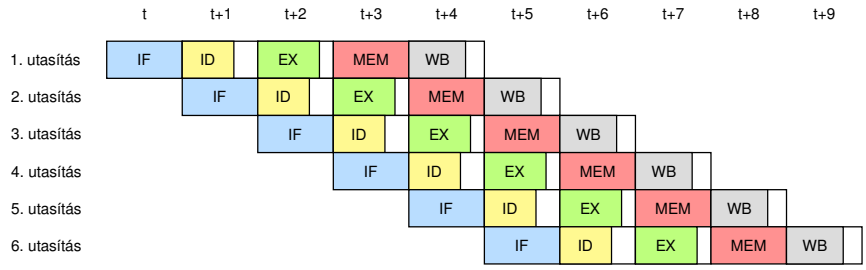
5. Az ötödik lépésben a megváltozott **regiszterek frissítésé**t (write-back cycle, *WB*) végezzük el. Ha az utasítás végrehajtása során van olyan regiszter, aminek az értéke megváltozott, akkor azt itt írjuk vissza a regiszter tárolóba, máskülönben ez a fázis kihagyható. Ilyen utasítások az aritmetikai utasítások (az ID fázisban olvassuk ki az operandusokat a regiszter tárolóból, az EX-ben elvégezzük a műveletet, a WB fázisban pedig beírjuk az eredményt a megfelelő regiszterbe), valamint ilyen a Load utasítás is (a MEM fázisban olvastuk be a megfelelő adatot, amelyet a WB fázisban írunk be a megfelelő regiszterbe).

A leírtakból látható, hogy egy utasítás végrehajtásához nem kell azt mindig mind az 5 fázison keresztültolni: egy aritmetikai utasítás az IF, ID, EX, WB fázisokból áll, a Store-hoz az IF, ID, EX, MEM fázisok szükségesek (az aritmetikai utasításokhoz hasonlóan 4 fázis), a Load mind az 5 fázist igényli, az ugró utasítások pedig csak az első 3-mat.

13.2. Pipeline utasításfeldolgozás

Az előző fejezetben sorra vett fázisok egymás után rendezésével és futószalag-szerű üzemeltetésével az időegységenként feldolgozott utasítások száma jelentősen növelhető. Az ötlet a 20. század eleji Chicago-i vágóhidak és Henry Ford autógyára óta ugyanaz: amikor egy munkadarab (utasítás) feldolgozása a következő fázisba lép, az előző fázis erőforrásaival a következő munkadarabon (utasításon) is el lehet kezdeni a munkát.

Vagyis ahelyett, hogy az utasításokat sorban egymás után hajtanánk végre (a következőt csak akkor elkezdve, ha az előző végrehajtása teljesen befejeződött), az egymás utáni utasítások végrehajtását átlapolhatjuk. Amikor befejeződik az IF fázis és az utasítás az ID fázisba lép, a következő utasítás végrehajtásának az IF fázisa ezzel (az előző ID fázisával) átlapolva megkezdődhet. A következő ciklusban az első utasítás az EX fázisba lép, a második az ID-be, és egy újabb utasítás pedig az IF-be (már 3 utasítás végrehajtása zajlik átlapolva), és így tovább. Ezzel elérhetjük, hogy az utasítások végrehajtásához szükséges erőforrások a lehető legjobban ki legyenek használva. Ahhoz, hogy a dolog működjön, a futószalagot állandó sebességgel kell hajtani, vagyis minden fázis számára ugyanannyi idő (egy *ciklusidő*) áll majd rendelkezésre a hozzá rendelt feladat elvégzésére akkor is, ha ezt az időt nem használja ki teljesen, mert hamarabb végez (lásd 13.1. ábra).

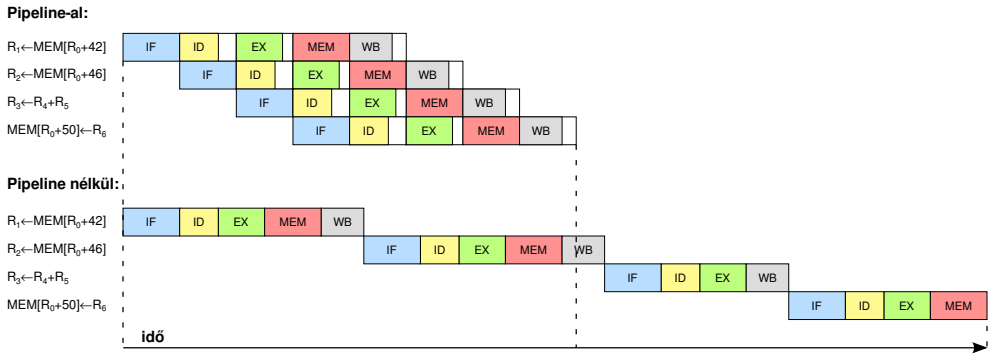


13.1. ábra. Pipeline fázisok

13.2.1. A pipeline késleltetése és átviteli sebessége

Lássuk, mennyit nyerünk a pipeline feldolgozással. Pipeline nélkül egy aritmetikai utasítás végrehajtása $T_{IF} + T_{ID} + T_{EX} + T_{WB}$ ideig, a Store $T_{IF} + T_{ID} + T_{EX} + T_{MEM}$ ideig, a Load pedig $T_{IF} + T_{ID} + T_{EX} + T_{MEM} + T_{WB}$

ideig tart. Pipeline feldolgozás esetén minden utasítást minden fázison át kell tolni (legfeljebb az aritmetikai utasításokkal a MEM, a Store-okkal a WB fázis nem csinál semmit), és a pipeline ciklusideje a legidőigényesebb fázis idejével egyezik meg, vagyis minden utasítás feldolgozása $T = 5 \cdot \max\{T_{IF}, T_{ID}, T_{EX}, T_{MEM}, T_{WB}\}$ ideig tart, ami nyilvánvalóan hosszabb a pipeline nélküli végrehajtási időnél. Paradox módon a pipeline feldolgozás mégis hatékonyabb, ugyanis az átlapolt utasítás-végrehajtás révén - ideális esetben - minden ciklusidőben befejeződik egy utasítás (lásd 13.2. ábra).



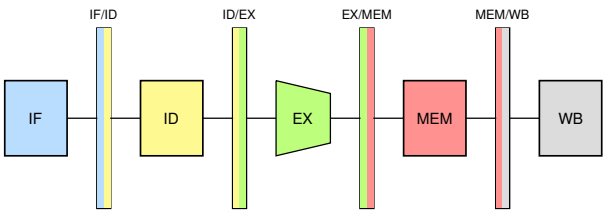
13.2. ábra. Utasítások feldolgozási ideje pipeline-nal és anélkül

A pipeline-nal kapcsolatos méret-, idő- és sebesség-jellegű mennyiségek:

- A pipeline *mélysége* (deepness) a feldolgozási fázisok száma (jelen példában ez 5).
- A pipeline *késleltetése* (latency) az utasítások végrehajtásának elkezdése és befejezése között eltelő idő. Értéke: a pipeline mélysége $*$ $\max\{T_i\}$.
- A pipeline *átviteli sebessége* (throughput): az időegységenként befejezett utasítások száma. Értéke (ideális esetben): 1/ciklusidő.
- *Feltöltődési idő*: Az első utasítás betöltésétől a pipeline megteléséig eltelő idő. A pipeline akkor telik meg, amikor a pipeline mélységével megegyező számú utasítás végrehajtása van folyamatban. Az üres pipeline egy feltöltődési idő múlva fejezi be az első utasítás végrehajtását (és ezután ciklusidőnként befejez egyet). A feltöltési idő értéke megegyezik a pipeline késleltetésével.
- *Kiürülési idő*: Az utasítások lehívásának megállításától számítva az az idő, amíg az utolsó, a pipeline-ban feldolgozás alatt álló utasítás el nem hagyja az utolsó fázist.

13.2.2. Pipeline regiszterek

Ahogy egy utasítás belép a pipeline-ba, és végighalad annak minden egyes fázisán balról jobbra, magával visz a végrehajtásához szükséges számos információt is. Ezeket az információkat az úgynevezett pipeline regiszterek tárolják (az első kivételével minden fázis előtt található egy ilyen). A fázisok a pipeline regiszterekből nyerik a tevékenységükhöz szükséges információkat, majd ezek egy részét a ciklusidő végén a jobb oldali szomszédos fázis felé továbbítják, annak pipeline regiszterébe írják. A pipeline regisztereket a szomszédos fázisok neveivel azonosítjuk (13.3. ábra).



13.3. ábra. Pipeline regiszterek

A pipeline működése - az utasítás-végrehajtás 1. fejezetben leírt módja ismeretében - tehát az alábbi módon történik:

1. Az **IF** fázis lehívja az utasításszámláló által mutatott utasítást, és megnöveli az utasításszámlálót úgy, hogy az a következő utasításra mutasson. Mind magát az utasítást, mind az utasításszámláló megnövelt értékét elhelyezi az IF/ID regiszterben.

2. Az **ID** fázis az utasítást dekódolja, és előállítja a további fázisok számára a vezérlő jeleket (pl. hogy az ALU-nak, ha majd odaér az utasítás, összeadást, kivonást, szorzást, osztást, összehasonlítást kell-e majd végeznie, stb...), és ezeket elhelyezi az ID/EX regiszterben. Ezután (ezzel párhuzamosan) az utasítás által kijelölt művelet operandusait is az ID/EX regiszterbe helyezi: ha az operandus egy regiszter, akkor kiolvassa annak értékét és azt, ha skalár konstans, akkor azt írja be.

3. Az **EX** fázis az ID/EX-ben előírt műveletet elvégzi a szintén az ID/EX-ben található operandusokkal, az eredményt pedig az EX/MEM pipeline regiszterbe írja.

4. A **MEM** fázis belenéz az EX/MEM regiszterbe. Ha az ID fázis egy ”Store” bitet beállított, akkor az EX/MEM-ben lévő adatot az úgyszintén ott lévő címre írja. Ha az ID fázis a ”Load” bitet állította be, akkor EX/MEM-ben lévő címről beolvas egy adatot, és azt a MEM/WB regiszterbe teszi. Egyéb esetekben a MEM fázisnak nincs dolga, továbbítja az EX/MEM tartalmát MEM/WB-be.

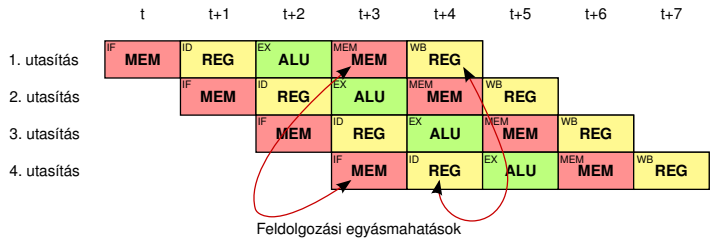
5. A **WB** fázis a MEM/WB-ben lévő adatot beírja az ott megjelölt regiszterbe (ezzel véglegesíti az eredményt).

13.3. Egymásrahatások a pipeline-ban

Egy utasítássorozat végrehajtása során számos körülmény felléphet, ami miatt a pipeline-t, a futószalagot meg kell állítani. Ezek a körülmények abból adódnak, hogy az utasítások általában nem egy független, lineáris folyamot alkotnak, és gyakran az egyes végrehajtási fázisok egymással is versengenek bizonyos erőforrásokért. Az ilyen helyzeteket egymásrahatásnak nevezzük, és mivel rontják a teljesítményt, lassítják az utasítássorozat végrehajtását, különféle megoldásokat kell kitalálni a kezelésükre.

13.3.1. Feldolgozási egymásrahatás

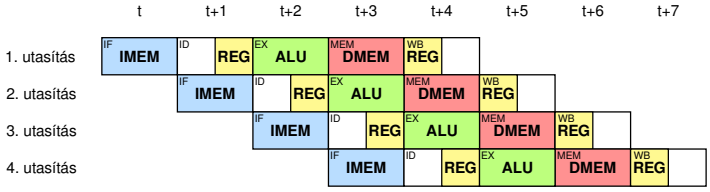
Feldolgozási egymásrahatás akkor lép fel, ha a processzor bizonyos erőforrásaira a pipeline több fázisának is egyidejűleg szüksége van. Ebben az esetben a pipeline megáll, hiszen az erőforrásért versengő fázisok nem tudnak egyszerre dolgozni, az egyik kénytelen egy ciklusidőnyi szünetet tartani, amíg a szükséges erőforrás felszabadul. A 13.4. ábrán láthatjuk az egyes fázisok által igényelt erőforrásokat.



13.4. ábra. Pipeline fázisok erőforráshasználata

A 13.4. ábrán két feldolgozási egymásrahatást is láthatunk: a memória elérésére szükség van az IF és a MEM fázisban is (t+3-ban), a regiszter tárolót pedig az ID és a WB fázis egyaránt használja (a t+4-ben). Mindkét problémára van megoldás. A memóriáért folytatott versenyt úgy tudjuk feloldani, hogy az utasításmemóriát és az adatmemóriát szétválasztjuk. Ezt vagy Harvard architektúra alkalmazásával érhetjük el, vagy külön utasítás és adat cache használatával (melyet egyébként Harvard cache-nek is neveznek). A regiszterekért folytatott versenyt pedig úgy tudjuk megszüntetni, hogy gyorsabb regiszter tárolót veszünk, amit az ID ill. a WB fázisok közül az egyik a ciklusidő első felében, a másik pedig a második felében használ. Célszerű azt a sorrendet követni, hogy a WB a ciklusidő első felében írja, az ID pedig a ciklusidő második felében olvassa a regiszter tárolót, hiszen így az

ID fázis egy aktuálisabb képet lát a regiszterekről. A 13.5. ábrán láthatjuk a fázisok erőforrás használatát a leírt megoldások alkalmazásával. A feldolgozási egymásrahatásokat sikerült kiküszöbölni.



13.5. ábra. Pipeline fázisok erőforráshasználata, feldolgozási egymásrahatás nélkül

13.3.2. Adat-egymásrahatás

Egy tipikus programban tipikus szituáció, hogy egy utasítás egyik vagy mindegyik operandusa az előtte lévő utasítások eredménye. (Ez teljesen természetes dolog, hiszen egy kicsit is összetettebb számítást már részműveletekre kell bontani, és a részműveletek eredményeiből egy későbbi művelet állítja elő a számítás eredményét.) Ilyenkor azt mondjuk, hogy ezen utasítások között *adat-függőség* van.

Egy példa a j. és az i. utasítás közötti adatfüggőségre, ha az i. utasítás eredménye a j. utasítás bemenő operandusa. Ezt a típusú függőséget *read-after-write*, röviden **RAW** függőségnek hívjuk. Tekintsük az alábbi példát:

```
i1:  R3 ← MEM [R0]
i2:  R1 ← R2 * R3
i3:  R4 ← R1 + R5
i4:  R5 ← R6 + R7
i5:  R1 ← R8 + R9
```

Ebben a példában 2 RAW függőség is van:

- Az i2 utasítás RAW függőségben van az i1-gyel, mert egyik bemeneti operandusa, nevezetesen az R3 regiszter az i1 utasítás eredménye.
- Az i3 utasítás RAW függőségben van az i2-vel (ugyanezen okokból).

A teljesség kedvéért itt jegyeznénk meg, hogy van két másfajta függőség is: az egyik a *write-after-write* - ***WAW*** - függőség, ami akkor lép fel, ha két utasítás ugyanabba a regiszterbe írja az eredményét (példánkban i2 és i5 áll WAW függőségben egymással, mert mindkettő R1-be írja az eredményét), valamint a *write-after-read*, **WAR** függőség, amikor az egyik utasítás olyan regiszterbe írja az eredményét, ami egy korábbi utasítás bemenő operandusa (példánkban i4 áll ilyen viszonyban i3-mal).

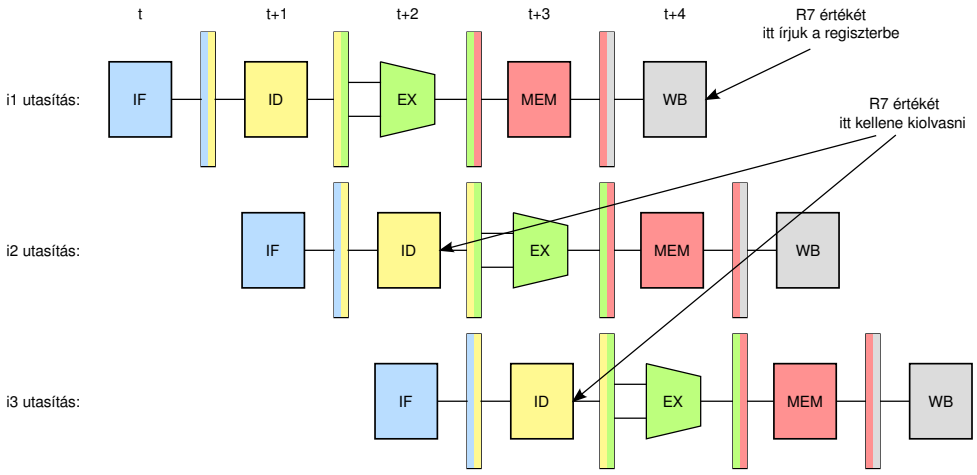
A jelen fejezetben bemutatott egyszerű, 5 fokozatú pipeline-ban a WAW és a WAR függőségek nem okoznak problémát, hiszen az egymás utáni utasítások feldolgozása során mind a regiszterek kiolvasása az ID fázisban, mind a regiszterek kiírása a WB fázisban az utasítások sorrendjének megfelelően, sorrendhelyesen történik.

A RAW függőségek okozta probléma megértéséhez vegyük az alábbi, egyszerűbb példát:

```
i1:  R7 ← R1 + R5
i2:  R8 ← R7 - R2
i3:  R5 ← R8 + R7
```

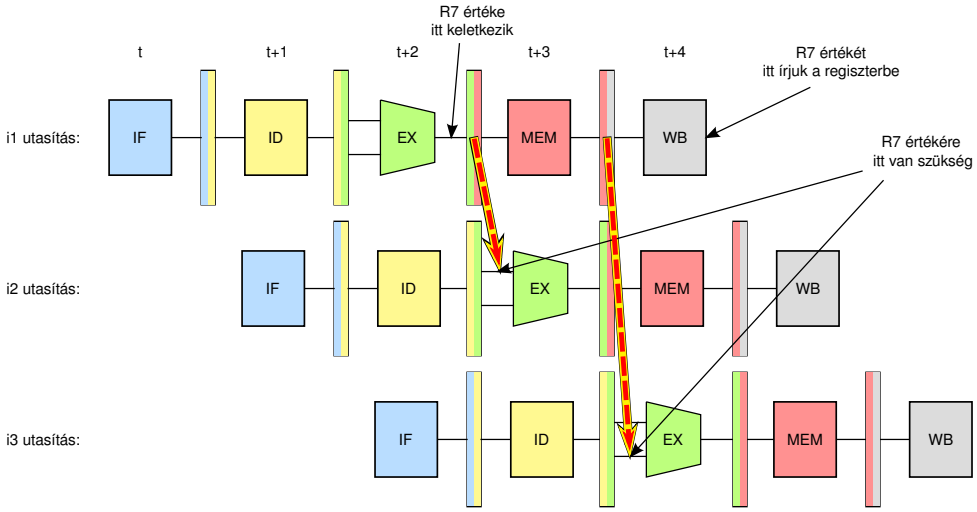
Ebben a példában 3 RAW függőséget is találhatunk: i1 és i2 között, i2 és i3 között, valamint i1 és i3 között. A 13.6. ábrán látható a probléma: az i2 utasítás végrehajtásakor az R7 regiszter értékét az ID fázisban, a t+2. lépésben kellene kiolvasni, de az R7 helyes értéke csak a t+4. lépésben kerül be a regiszter tárolóba, amikor az i1 utasítás a WB fázisban van. Ugyanez a helyzet az i3 végrehajtása során is: annak az R7-re t+3-ban, az ID fázisban lenne szüksége, de az R7 csak t+4-ben kerül beírásra a regiszter tárolóba.

A megoldást az úgynevezett *forwarding* technika jelenti. Ez a következőképpen működik. Vegyük észre, hogy az R7 regiszter értékére az i2 végrehajtása során valójában csak t+3-ban van szükség (hiszen ekkor lép az EX fázisba), és azt is vegyük észre, hogy az R7 új, majdan beírásra szánt értéke ekkorra már készen van,



13.6. ábra. RAW függőségek a pipeline-ban

egészen pontosan az EX/MEM pipeline regiszterben. A RAW függőséget tehát az i2 végrehajtásakor az ID fázisban detektálni kell, és olyan vezérlő jelet kell az ALU részére előállítani, hogy az az EX fázisban a problémás operandust ne az ID/EX pipeline regiszterből vegye (ahova az ID fázis során normális esetben kerülne), hanem az EX/MEM pipeline regiszterből, ahová az ALU az eredményt szokta írni. Hasonlóan kell eljárni az i3 végrehajtása során is. Az i3-nak R7 értékére legkésőbb t+4-ben kellene az eredmény, de ekkor, a ciklus elején az még nincs bent a regiszter tárolóban, viszont megtalálható a MEM/WB pipeline regszterben. Az ID fázisban ezt a helyzetet is detektálni kell, és az ALU-nak olyan vezérlő jelet kell adni, hogy az i3 végrehajtásakor ezt az operandust az ID/EX pipeline regiszter helyett a MEM/WB regiszterből vegye. A forwarding segítségével feloldott RAW függőségeket a 13.7. ábra szemlélteti. Ebben az 5 fokozatú pipeline-ban, amivel ebben a fejezetben foglalkozunk, a 2-nél nagyobb távolságra lévő utasítások közötti RAW függőség nem jelent problémát, hiszen mire odaér a vezérlés, az érintett regiszter már a helyes értéket fogja tartalmazni, amikor az ID fázis ki akarja azt olvasni.

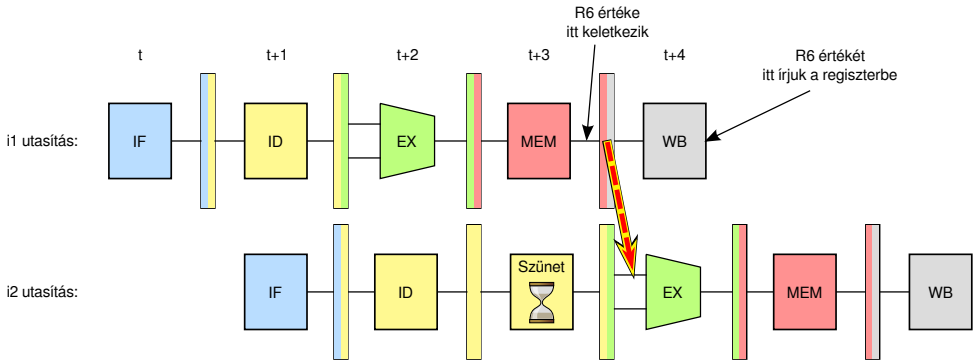


13.7. ábra. RAW függőségek feloldása forwarding-gal

Sajnos a RAW függőségek nem mindig oldhatók fel ilyen egyszerűen. Lássuk például a következő esetet, melyben egy Load és egy aritmetikai művelet között áll fent a RAW függőség:

```
i1:  R6 ← MEM[R2]
i2:  R7 ← R6 + R4
```

Ebben az esetben a korábban látott forwarding technika nem hoz teljes megoldást, hiszen az R6 majdani tartalma az i1 MEM fázisának végén jelenik meg (a MEM/WB pipeline regiszterben) a t+3. ciklus végére, viszont az R6 értékére az i2-nek már az EX fázisban, a t+3. ciklus elején szüksége van. Ilyenkor nincs más megoldás, meg kell állítani a pipeline-t. Az i2 és az utána a pipeline-ban lévő utasítások feldolgozása 1 ciklusidőre szünetel, és új utasítás sem hozható be. Az 1 ciklusidő szünet után az i2 utasítás az EX fázisba léphet, és az ALU az R6 értékét forwarding segítségével a MEM/WB regiszterből kiveheti (13.8. ábra). A szünet szükségességét az ID fázisban detektálni kell, és a későbbi fázisok megfelelő vezérlő jeleit ennek megfelelően kell előállítani, valamint a szünet idejére az új utasítások lehívását (tehát az IF fázist) is le kell tiltani.



13.8. ábra. Szünet beiktatása Load utasítás okozta RAW függőség esetén

13.3.3. Procedurális egymásrahatás

A pipeline működés szempontjából nehézséget jelentenek a feltételes ugrások, ugyanis az ugrási feltétel kiértékelése és az ugrás címének a kiszámítása csak az EX fázisban történik. Kérdés, hogy az IF egység honnan töltse le a következő utasításokat, amíg ki nem derül az elágazás kimenetele és címe? A legegyszerűbb megoldás persze az, hogy feltételes ugró utasítások esetén, amíg azok túl nem esnek az EX fázison, a további utasítások lehívása szünetel (pl. az Intel 80386 így működött). A sok szünet azonban rontja a pipeline teljesítményét (átviteli sebességét), különösen ha a pipeline nagyon hosszú, és az utasításletöltés valamint a feltételes ugrások kiértékelése a pipeline-ban távol vannak egymástól. Ezért szinte minden processzor alkalmaz valamilyen becslést az elágazás kimenetelére nézve, ezt elágazásbecslésnek nevezzük.

Statikus elágazásbecslés

Statikus elágazásbecslés esetén a processzor a program korábbi ”előéletének” figyelembe vétele nélkül, minden tárolt információ híján próbálja megtippelni az elágazás kimenetelét. A legegyszerűbb heurisztikák:

- Mindig az ugrás meg nem története mellett tesszük le a voksot. Könnyen látható, hogy ez a stratégia ”ingyen van”, hiszen amikor kiderül, hogy tévedtünk, a pipeline-ba (tévesen) lévő, végrehajtás alatt lévő utasítások még nem juthattak el sem a MEM, sem a WB fázisba, nem változtathatták meg a program állapotát, vagyis könnyen érvényteleníthetjük őket (pl. bebillentünk egy bitet a megfelelő pipeline regiszterekbe, hogy a MEM és a WB fázis hagyja őket figyelmen kívül). Pl. az Intel 80486 így működött. Sajnos a feltételes ugrások bekövetkezése gyakoribb, mint a be nem következése (lásd: ciklusok).
- Vannak processzorok, melyek az ugrás bekövetkezését feltételezik (de ez a stratégia csak akkor hasznos, ha a feltétel kiértékelése később történik, mint az ugrási cím kiszámítása).
- Az is egy stratégia lehet, hogy a visszafelé mutató ugrási cím esetén az ugrást, előre mutató cím esetén a be nem következést feltételezzük.

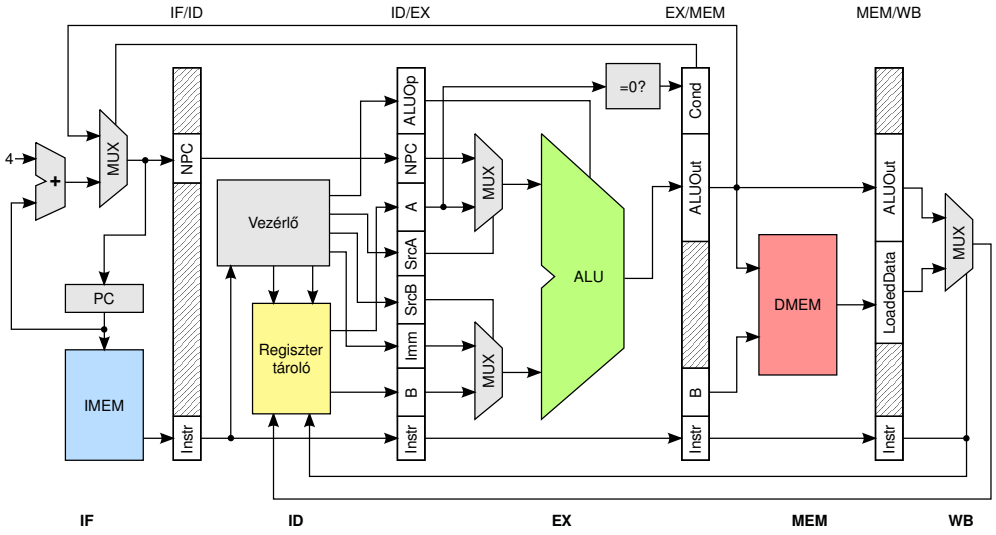
Dinamikus elágazásbecslés

A modern processzorok mély pipeline-nal rendelkeznek, ezért nagyon fontos a jó elágazásbecslés, hiszen a rossz döntés sok szükségtelen utasításra elpazarolt ciklusidővel jár. Emiatt az elágazásbecslés szerepe egyre fontosabb.

Az előbbi statikus megoldások helyett dinamikus becsléssel sokkal jobb becslési pontosságot lehet elérni. A dinamikus elágazásbecslés lényege, hogy az elágazásbecslő logika folyamatosan követi, monitorozza a program feltételes ugró utasításait (természetesen ez mindenféle adatok folyamatos könyvelésével jár) és a múltbéli viselkedés valamint a környező más feltételes ugró utasítások együttes viselkedése alapján becsüli meg az ugrás kimenetelét és címét. Egyes processzorokban neurális hálózatokkal, vagy a mesterséges intelligencia egyéb eszközeivel is megtámogatják a döntést. A dinamikus elágazásbecslésről egy későbbi fejezetben sokkal részletesebben lesz szó.

13.4. Pipeline implementáció

Ebben a fejezetben megadjuk a felvázolt 5 fokozatú pipeline egy lehetséges implementációját. A blokksémát a 13.9. ábrán láthatjuk. Az egyszerűség kedvéért (hogy kiférjen az ábrára) az első körben nem veszünk tudomást az egymásrahatásokról (ezt a hiányosságot később pótoljuk).



13.9. ábra. Az 5 fokozatú pipeline egy lehetséges implementációja

Most ismét sorra vesszük az ábra alapján az egyes fázisokat, és egy pszeudo-nyelv segítségével egészen pontosan megfogalmazzuk, hogy mi is történik:

1. **IF** fázis: A ”PC” regiszter tárolja az utasításszámlálót (program counter). Az IF fázis a ciklus elején kiolvassa az utasításmemóriából (IMEM) az utasításszámláló által mutatott utasítást, és beteszi az IF/ID pipeline regiszter ”Instr” mezőjébe. Ezután egy összeadó egység a PC értékéhez hozzáad 4-et (most ilyen hosszúak az utasításaink). Az IF fázisban látható multiplexer vagy ezt a megnövelt értéket, vagy ha később, az EX fázisban egy feltételes ugró utasítás ugrási feltétele teljesül, az ugrási címet írja vissza a PC regiszterbe, hogy a következő ciklusban innen hívhassuk le a soron következő utasítást. Az utasításszámláló új értékét mindeközben az IF/ID regiszter ”NPC” mezőjébe is beírjuk.
   Formálisan megfogalmazva:

```
IF/ID.Instr ← IMEM[PC]
PC ← SELECT (EX/MEM.Instr.Opcode == branch && EX/MEM.Cond; ...
                          ... true: EX.MEM.ALUOut, false: PC+4)
IF/ID.NPC ← PC
```

2. **ID** fázis: A vezérlő fogja az IF/ID ”Instr” mezőjében lévő utasítást, és dekódolja azt: Ha aritmetikai műveletet lát, beállítja az ALU számára a megfelelő vezérlőjelet az ID/EX pipeline regszter ”ALUOp” mezőjébe. Ha Load, Store vagy ugrás műveletet lát, akkor összeadás műveletet kell beállítania, hiszen ezekben az esetekben az ALU-t címszámításra használjuk (a Load és a Store esetén a bázisregiszterhez kell majd hozzáadni az eltolást,

az ugrás esetén pedig az utasításszámlálóhoz). Az utasítás regiszter operandusainak az értékét kiolvassa a regiszter tárolóból, és az ID/EX ”A” és ”B” mezőjébe írja. Ha a második operandus nem regiszter, hanem skalár konstans, akkor azt leválasztja az utasításszóról, és az ”Imm” mezőbe írja (immediate operandus). Beállítja az ALU számra a vezérlő jeleket, hogy a műveletek operandusait honnan vegye (”SrcA” és ”SrcB”): ugrás esetén az első operandust az ”NPC”, más esetben az ”A” mezőből kell venni, ha a második operandus skalár konstans, akkor azt az ”Imm”, ha nem, akkor pedig a ”B” mezőből kell vennie. Az ID fázis az ”NPC” és az ”Instr” mezőket továbbítja az EX felé. Tehát:

```
ID/EX.NPC ← IF/ID.NPC
ID/EX.Instr ← IF/ID.Instr
ID/EX.Imm ← IF/ID.Instr [imm]
ID/EX.A ← Reg [IF/ID.Instr [ra]]
ID/EX.B ← Reg [IF/ID.Instr [rb]]
ID/EX.SrcA ← SELECT (IF/ID.Instr.Opcode == branch; ...
                          ... true: pc, false: reg)
ID/EX.SrcB ← SELECT (IF/ID.Instr.HasImm; ...
                          ... true: imm, false: reg)
ID/EX.ALUOp ← SELECT (IF/ID.Instr.Opcode == arithm; ...
                          ... true: IF/ID.Instr.Func, false: "+")
```

3. **EX** fázis: Az EX muliplexerei az ID/EX ”SrcA” és ”SrcB” jelei alapján kiválasztják a megfelelő adatforrást, ezek lesznek az elvégzendő művelet operandusai. Az ALU elvégzi az ”ALUOp” mezőben jelzett műveletet, és az eredményt az EX/MEM ”ALUOut” mezőjébe írja. Ha feltételes utasításról van szó, elvégzi a ugrási feltétel ellenőrzését is (az első regiszter operandus 0-val egyezőségét vizsgálja), és a kimenetelét a ”Cond” mezőbe teszi. Az utasítás mellett a második operandust (”B”) is tovább kell adni a MEM fázisnak, hiszen egy Store művelet esetén ebben van a memóriába írandó szó.

```
EX/MEM.Instr ← ID/EX.Instr
ALU.A ← SELECT (ID/EX.SrcA; reg: ID/EX.A, pc: ID/EX.NPC)
ALU.B ← SELECT (ID/EX.SrcB; reg: ID/EX.B, imm: ID/EX.Imm)
ALU.Op ← ID/EX.ALUOp
EX/MEM.ALUOut ← ALU.Out
EX/MEM.Cond ← ID/EX.A == 0
EX/MEM.B ← ID/EX.B
```

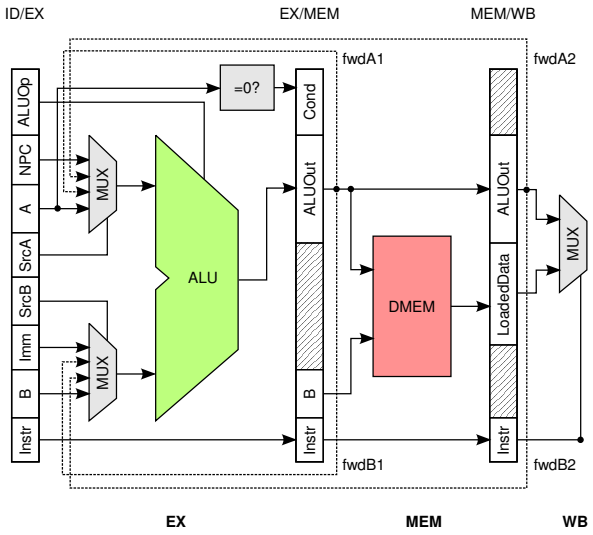
4. **MEM** fázis: Load és Store utasítás esetén itt kell elvégezni a memóriaműveletet. Load esetén az EX/MEM pipeline regiszter ”ALUOut” mezője tartalmazza a memóriacímet, a beolvasott adat pedig a MEM/WB ”LoadedData” mezőjébe kerül. Store utasítás esetén szintén az ”ALUOut” tartalmazza a címet, a memóriába írandó adatot pedig az EX/MEM regiszter ”B” mezője.

```
MEM/WB.Instr ← EX/MEM.Instr
if (ID/EX.Instr.Opcode == Load): MEM/WB.LoadedData ← MEM [EX/MEM.ALUOut]
if (ID/EX.Instr.Opcode == Store): MEM [EX/MEM.ALUOut] ← EX/MEM.B
if (ID/EX.Instr.Opcode == arithm): MEM/WB.ALUOut ← EX/MEM.ALUOut
```

5. **WB** fázis: Aritmetikai és Load műveleteknél ebben a fázisban történik meg az eredmények visszaírása a regiszter tárolóba. A multiplexer, attól függően hogy aritmetikai vagy Load művelet történt-e, vagy az ALUOut, vagy a LoadedData mező tartalmát írja a regiszter tárolóba. Az ”Instr” mezőben lévő utasítás alapján az is eldönthető, hogy az utasítás eredményét melyik regiszterben kell elhelyezni (rd).

```
Reg [MEM/WB.Instr [rd]] ← SELECT (MEM/WB.Instr.Opcode; ...
                        ... arithm: MEM/WB.ALUOut, Load: MEM/WB.LoadedData
```

Az egymásrahatások figyelembe vételéhez nézzük először a legegyszerűbb esetet, az aritmetikai utasítások RAW függőségeit. Mint láttuk, ez a fajta függőség a forwarding technika alkalmazásával a pipeline megállítása nélkül teljes mértékben kiküszöbölhető. A 13.9. ábrához képest az lesz a különbség, hogy az ALU operandusainak kiválasztó multiplexerei nem csak az ID/EX pipeline regiszterből vehetik az operandusokat, hanem a forwarding utakon keresztül is, az EX/MEM és a MEM/WB "ALUOut" mezőjéből (lásd 13.10. ábra, szaggatott vonalak).



13.10. ábra. Forwarding utak az ALU operanduskiválasztó multiplexerében

Természetesen ekkor az ID fázisban ezeknek a multiplexerek a vezérlő jeleit is az egymásrahatások tudatában kell előállítani. Ha az ID fázis úgy látja, hogy a hozzá érkezett utasítás egyik operandusa megegyezik a pipeline-ban eggyel vagy kettővel előbbre járó utasítás eredmény regiszterével, akkor a megfelelő forwarding utat kell kiválasztania az ALU multiplexerének, amikor az EX fázisba lép az utasítás végrehajtása:

```
ID/EX.SrcA ← SELECT (IF/ID.Instr.Opcode == branch; true: pc, ...
    ... false: SELECT (IF/ID.Instr.Opcode == arithm &&
        && IF/ID.Instr[ra] == ID/EX.Instr[rd]; true: fwdA1, ...
            ... false: SELECT (IF/ID.Instr.Opcode == arithm &&
                && IF/ID.Instr[ra] == EX/MEM.Instr[rd]; true: fwdA2, ...
                    ... false: reg))
ID/EX.SrcB ← SELECT (IF/ID.Instr.Opcode == branch; true: pc, ...
    ... false: SELECT (IF/ID.Instr.Opcode == arithm &&
        && IF/ID.Instr[rb] == ID/EX.Instr[rd]; true: fwdB1, ...
            ... false: SELECT (IF/ID.Instr.Opcode == arithm &&
                && IF/ID.Instr[rb] == EX/MEM.Instr[rd]; true: fwdB2, ...
                    ... false: reg))
```

Amint korábban láttuk, a Load műveletek és az aritmetikai műveletek között előfordulhat olyan RAW függőség, ami szükségessé teszi a pipeline megállítását. Ezt az esetet az ID fázis a fentiekhez nagyon hasonló módon detektálni tudja, és a megállításhoz szükséges logika aktiválásával kezelni tudja (ez a logika nincs az ábrán). Vegyük észre, hogy a szünet után ismét szükség van forwarding-ra, ekkor a MEM/WB pipeline regiszter "LoadedData" mezőjét kell az ALU multiplexerére vezetni és operandusnak kiválasztani. Tehát az ALU operanduskiválasztó multiplexereken már 5 bemenet lesz: a 9. ábrán látott kettő, és három forwarding út: az EX/MEM.ALUOut, a MEM/WB.ALUOut valamint a MEM/WB.LoadedData.

A procedurális egymásrahatás kezelése attól függ, milyen elágazásbecslő módszert választunk. A legegyszerűbb megoldás, ha az IF egység a feltételes ugró utasítás detektálása után két ciklusidőnyi szünetet tart (szüneteltei az utasításlehívást), ennyi idő kell ugyanis ahhoz, hogy az ugró utasítás az EX fázisba érjen, és az ugrási feltétel valamint az ugrási cím rendelkezésre álljon. A szüneteltetéshez szükséges logika az adat-egymásrahatások kezeléséhez amúgy is szükséges, tehát ez az eljárás kis ráfordítással megvalósítható. Hasonlóan egyszerű megoldás,

hogy az ugrás meg nem történését feltételezve folytatjuk az utasítások lehívását. Ha az ugrási feltétel mégis teljesül, az időközben tévesen lehívott, IF és ID fázisokban tartózkodó utasításokat hatástalanítani kell valahogy. Ezt például úgy is megtehetjük, hogy bebillentünk egy bitet az IF/ID és ID/EX regiszterekben (amit az utasítások később magukkal cipelnek az EX/MEM és MEM/WB regiszterekbe is), és ez a bit megtiltja a MEM és a WB fázisban a memória ill. a regiszter tároló írását.

13.5. Kivételek, megszakítások kezelése a pipeline-ban

Az utasítások végrehajtása közben számos olyan váratlan esemény történhet, ami az utasítások feldolgozásának ”normál” menetét megszakítja. A vezérlés ilyenkor az esemény lekezelését végző hardvernek vagy szoftvernek adódik át. Az esemény lekezelése után - az esetek többségében - a megszakított utasítássorozat végrehajtása folytatódhat. Ezeket az eseményeket a különböző gyártók különbözőképpen nevezik: *kivétel*nek (exception, Motorola), vagy *megszakítás*nak (interrupt, IBM, Intel).

Ezeket a kivételes eseményeket többféle szempont szerint is csoportosíthatjuk:

- Szinkron vagy aszinkron: ha az esemény minden egyes alkalommal ugyanott és ugyanakkor keletkezik, ha egy utasítássorozatot ugyanazon formában a processzor és a memória ugyanazon állapotában többször lefuttatunk, akkor szinkron eseményről beszélünk. Nem szinkron, vagyis aszinkron események például a processzortól és a memóriától függetlenül működő perifériák kérései.
- Maszkolható vagy nem maszkolható: A maszkolható események kezelését a futó program a maszk megfelelő beállításával letilthatja. A nem maszkolható események kezelése nem letiltható.
- Utasítások között vagy közben kezelendő események: Vannak események, melyek kezelése nem tűrhet halasztást, még az aktuálisan végrehajtás alatt álló utasítás sem fejezhető be. Ezeket az eseményeket utasítások közben feldolgozandó eseményeknek nevezzük. Ilyen pl. virtuális memóriakezelés esetén a laphiba: egy memóriaművelet során előfordulhat, hogy a hivatkozott lap a háttértáron van. Ilyenkor félbeszakítjuk az utasítás végrehajtását, majd az esemény lekezelése (a lap behozása) után onnan folytatjuk, ahol abbahagytuk. Más események kezelése ráér addig, amíg az aktuálisan végrehajtás alatt álló utasítás végrehajtása befejeződik.
- Folytatható vagy terminális: Ha egy esemény terminális, akkor a bekövetkezése esetén a futó program végrehajtása végleg megszakad. Folytatható események esetén az esemény kezelése után a program végrehajtása folytatható.

Az alábbi táblázatban összegyűjtöttünk néhány jellegzetes példát az utasításfeldolgozás közben fellépő eseményekre:

Esemény típusa	**Szinkr./Aszinkr.**	**Maszkolható**	**Kezelendő**	**Folytatható**
Periféria kérés	Aszinkron	Igen	Utasítások között	Igen
Integer aritm. túlcsord.	Szinkron	Igen	Utasítások közben	Igen
Laphiba	Szinkron	Nem	Utasítások közben	Igen
Védelmi hiba	Szinkron	Nem	Utasítások közben	Igen
Érvénytelen utasítás	Szinkron	Nem	Utasítások közben	Igen/Nem
Hardver meghibásodás	Aszinkron	Nem	Utasítások közben	Nem

Pipeline végrehajtás során a kivételek kezelését jó néhány körülmény megnehezíti. A kivételkezelés szempontjából az lenne a kívánatos viselkedés, ha a kivétel kezelését megelőzően minden, a kivételt okozó (i.) utasítás előtti ($\leq$i.) utasítás végrehajtása befejeződne, az ezt követő ($\geq$i.) utasítások pedig a kivételkezelés után gond nélkül újraindíthatók lennének. Azaz a kivételt kezelő szoftver vagy hardver olyan állapotot lásson, hogy a kivétel előtti utasítások teljesen lefutottak, az azután következők pedig egyáltalán nem. Ezt a viselkedést *pontos kivételkezelés*nek nevezik.

A pontos kivételkezelést azonban nem könnyű implementálni. Mivel több utasítás végrehajtása zajlik egyidejűleg, a kivétel bekövetkeztekor jó lenne megakadályozni, hogy a kivételt okozó utasítás után a pipeline-ban lévő további, elkezdett utasítások akár a memóriába, akár a regiszter tárolóba írjanak. A memóriába és a regiszter

tárolóba írás letiltása ráadásul gyakran nem elegendő: lehet, hogy a kivétel keletkezése pillanatában már meg is történt a baj, vagyis egy későbbi utasítás már végzett is az írással. Ilyenkor a kivételkezelő hívásakor már nem lehet visszaállítani a kivétel keletkezésekor fennálló állapotot. A helyzet megoldására meglehetősen bonyolult logika szükséges, amit az egyszerűbb processzorok nem is tartalmaznak, ilyenkor a gyártó deklarálja, hogy a processzor nem képes pontos kivételkezelésre.

A pontos kivételkezeléshez az is hozzátartozik, hogy a kivételek bekövetkezése és kezelése az utasítások sorrendjében történjen. Pipeline végrehajtás esetén ez nem megy magától. Először is nézzük meg, hogy az egyes pipeline fázisokban milyen kivételek történhetnek:

- IF fázis: Laphiba, védelmi hiba
- ID fázis: Érvénytelen utasítás
- EX fázis: Aritmetikai hiba (pl. integer túlcsordulás)
- MEM fázis: Laphiba, védelmi hiba
- WB fázis: Itt nem történhet kivétel

Vegyünk egy egyszerű példát, amiben egy aritmetikai és egy Load utasítás követi egymást:

$R_k \leftarrow r_m + R_n$	IF	ID	EX	MEM	WB	
$R_i \leftarrow MEM[R_j]$		IF	ID	EX	MEM	WB

Ha az aritmetikai utasítás az EX fázisban vált ki kivételt (pl. túlcsordulást), a Load pedig az IF fázisban (pl. laphibát), és a kivételeket a kiváltásuk pillanatában kezelnék, akkor felborulna a sorrend (és sérülne a pontos kivételkezelés elve), hiszen a későbbi utasítás kivételét hamarabb kezelnénk, mint a korábbiét. Ezt a problémát például úgy lehet feloldani, hogy a kivételeket nem azonnal, a keletkezésük pillanatában kezeljük. Minden utasítás egy speciális regiszterben, a kivétel státusz vektorban gyűjti a vele kapcsolatban felmerült kivételeket. Amint keletkezett kivétel, a memória és regiszter tároló írását azonnal letiltjuk, de a kivétel tényleges kezelését csak a WB fázisban (vagy az után) kezdjük el, ahol rá kell nézni a kivétel státusz vektorra, és az időben legkorábban bekövetkezett kivételt kell lekezelni. Mivel a kivételek kezelése minden utasítás esetén a feldolgozás ugyanazon fázisában (az utolsóban) történik, a sorrend garantáltan helyes marad.

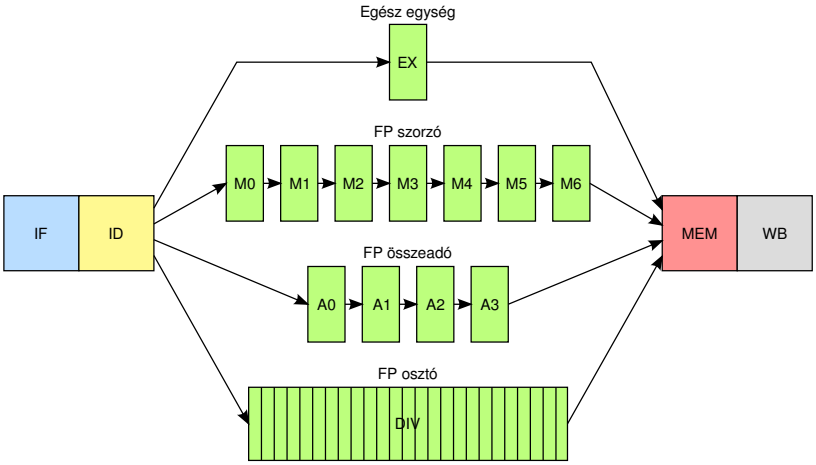
13.6. Eltérő késleltetésű aritmetikai műveletek kezelése

Az eddigiekben feltételeztük, hogy az ALU minden aritmetikai műveletet egyetlen ciklusidő alatt el tud végezni. Amíg kizárólag egész (integer) műveleteket végzünk, ez reális feltételezés. A lebegőpontos műveletek (floating point, FP) elvégzése azonban tipikusan sokkal több időt vesz igénybe, mint az egész műveleteké. Kár lenne viszont a pipeline-t úgy lelassítani, hogy a ciklusidőbe a leghosszabb FP művelet is beleférjen.

Az egyik lehetséges megoldás, hogy magát az EX fázist is pipeline-nal valósítjuk meg: megfelelő kialakítással el lehet érni, hogy ha a műveletek több ciklusidőt vesznek is igénybe, de legalább minden ciklusban új műveletet indíthatunk. Ebben a fejezetben ennél is tovább megyünk: olyan pipeline aritmetikai egységet veszünk alapul, melyben az egyes FP aritmetikai műveletek *különböző mélységű* pipeline-nal vannak megvalósítva. Ehhez egy olyan példát fogunk összeállítani, melyben az ALU egy integer egység mellett egy FP szorzót, egy FP összeadót valamint egy FP osztó egységet tartalmaz. Az FP szorzó és összeadó maga is pipeline felépítésű, tehát minden egyes ciklusidőben egy új műveletet képes elkezdeni (tehát iterációs ideje = 1), az osztó viszont erre nem képes (tehát csak akkor képes új műveletet indítani, ha az előzőt befejezte: iterációs ideje megegyezik a késleltetésével). A 3 FP egység késleltetése különböző, az alábbi táblázat szerint alakul (az értékek ciklusidőben vannak megadva):

Aritmetikai egység	**Késleltetés**	**Iterációs idő**
Integer ALU	1	1
FP összeadó.	4	1
FP szorzó	7	1
FP osztó	25	25

A pipeline szerkezetét a 13.11. ábra szemlélteti. Több aritmetikai egységünk van, de vigyázzunk: az ID egységet minden ciklusban legfeljebb egy utasítás hagyja el, amely a 4 egység valamelyikébe lép be. Szó sincs tehát (egyelőre) arról, hogy a több egység rendelkezésre állását úgy használjuk ki, hogy ugyanabban a ciklusban több egység felé is továbbítunk utasítást.



13.11. ábra. Pipeline eltérő késleltetésű aritmetikai egységekkel

A kialakítás egyik érdekessége, hogy az utasítások, bár végrehajtásuk sorban egymás után kezdődik, soron kívül fejeződhetnek be, hiszen az egyik utasítás kevesebb, a másik több ciklusidőt tölt a pipeline-ban. Lássunk erre egy példát (a lebegőpontos regisztereket D-vel jelöljük):

$D4 \leftarrow D1 * D5$	IF	ID	M0	M1	M2	M3	M4	M5	M6	MEM	WB
$D2 \leftarrow D1 + D3$		IF	ID	A0	A1	A2	A3	MEM	WB		
$D0 \leftarrow MEM[R0 + 4]$			IF	ID	EX	MEM	WB				
$MEM[R0 + 8] \leftarrow D5$				IF	ID	EX	MEM	WB			

Látható, hogy először a 3., majd a 4., a 2., és végül az 1. utasítás fejeződik be. Ez nem feltétlenül jelent problémát, ebben a példában sem volt belőle gond, hiszen az utasítások egymástól függetlenek, semmilyen egymásrahatás nem történt. Természetesen általános esetben gondoskodnunk kell róla, hogy a program szemantikája az utasítások soron kívüli befejeződése miatt ne változzon. Ezt a feladatot az ID egység fogja ellátni. Nézzük, milyen eddig nem látott, extra feladatai lesznek az ID fázisnak az eltérő késleltetésű aritmetikai egységek miatt:

1. **Feldolgozási egymásrahatások kezelése**
   A korábbi 5 fokozatú pipeline-unkban a feldolgozási egymásrahatásokat sikerült tervezési úton kiküszöbölni. Most ez sajnos nem lehetséges. Az alábbi példában pédául a 11. ciklusban egyszerre 3 utasítás is a WB fázisba ér, és mindhárman szeretnék elérni a regiszter tárolót (sőt, a 10. ciklusban a MEM fázist is egyszerre érik el):

Utasítás	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
$D4 \leftarrow D1 * D5$	IF	ID	M0	M1	M2	M3	M4	M5	M6	MEM	WB
…		IF	ID	EX	MEM	WB					
…			IF	ID	EX	MEM	WB				
$D2 \leftarrow D1 + D3$				IF	ID	A0	A1	A2	A3	MEM	WB
…					IF	ID	EX	MEM	WB		
…						IF	ID	EX	MEM	WB	
$D0 \leftarrow MEM[R0 + 4]$							IF	ID	EX	MEM	WB

Az ebből adódó feldolgozási egymásrahatásokat úgy lehet kiküszöbölni, hogy az ID fázis nyilvántartja, hogy a jövőben melyik erőforrás mikor lesz foglalt, és a nála tartózkodó utasítást csak akkor engedi továbblépni a megfelelő aritmetikai egységbe, ha az az utasítás feldolgozásának végéig a további fázisok egyikében sem fog olyan erőforrást igényelni, amit más is használ. Ezt könnyedén előre ki tudja számítani, hiszen az ID fázis tudja, hogy az utasításnak melyik aritmetikai egységre lesz szüksége, így pontosan lehet tudni, hogy mikor ér a MEM ill. a WB fázisba. Amíg az utasítást az ID fázis nem engedi továbblépni, a további utasítások lehívása szünetel, tehát a pipeline teljesítménye romlik.

Egy másfajta feldolgozási egymásrahatás, ha két FP osztás követi egymást. Mivel az FP osztó nem pipeline szervezésű, a második osztás utasításnak 24 extra ciklust kell az ID egységben töltenie, mire az FP osztó felszabadul. Erre az időre a további utasítások lehívása szintén szünetel.

2. **A RAW egymásrahatások gyakrabban fordulnak elő és gyakrabban állítják meg a pipeline-t**
Korábban, lebegőpontos műveletek nélkül csak a Load és az aritmetikai utasítások RAW függősége igényelt egy ciklusidőnyi szünetet az utasítások lehívásában. Most, hogy az aritmetikai műveletek eredménye a pipeline sokkal későbbi fázisában áll elő, ezek a teljesítményt rontó szünetek sokkal gyakoribbá és hosszabbá válnak. Másrészt, az eddigieknél távolabbi utasítások között is gondot jelent (detektálást és forwarding-ot, vagy/és szünetet igényelve) a RAW függőség. Példa:

Utasítás	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	
i1: $D4 \leftarrow MEM[R0+4]$	IF	ID	EX	MEM	WB				…
i2: $D0 \leftarrow D4 * D6$		IF	ID	szün	M0	M1	M2	M3	…
i3: $D2 \leftarrow D0 + D8$			IF	szün	ID	szün	szün	szün	…
i4: $MEM[R0+4] \leftarrow D2$					IF	szün	szün	szün	…
	9	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
i1:									
i2:	M4	M5	M6	MEM	WB				
i3:	szün	szün	szün	A0	A1	A2	A3	MEM	WB
i4:	szün	szün	szün	ID	EX	szün	szün	szün	MEM

A példában a szorzás csak akkor kezdhető el, ha a Load eredménye már megvan (a MEM fázisban kerül beolvasásra). Az összeadásnak meg kell várnai a szorzás eredményét (12. fázisban kezdhető el a végrehajtása). A Store utasítás címszámítása időben megtörténhet (13. fázis), de maga a memóriaművelet csak akkor mehet végbe, amikor az összeadás elkészül, vagyis a 16. ciklusban. Ekkor azonban épp az összeadó utasítás van a MEM fázisban, tehát a Store-nak feldolgozási egymásrahatás miatt még egy ciklust várakoznia kell.

3. **A WAW függőségek detektálása és kezelése**
Mivel az 5 fokozatú pipeline-ban az utasítások a megfelelő sorrendben fejeződtek be, a WAW függőségek nem jelentettek gondot. Most nem ez a helyzet. Könnyen előfordulhat, hogy egy későbbi utasítás hamarabb fejeződik be egy korábbinál, és ha mindkét utasítás ugyanabba a regiszterbe írja az eredményét (tehát WAW függőség áll fent), akkor a végén nem a későbbi, hanem a korábbi utasítás eredménye lesz a regiszterben. Az alábbi példában is ez történik, a program szemantikája szerint azt várnánk, hogy végül a Load eredménye lesz a D2-ben, de nem ezt fogjuk tapasztalni, mert a korábbi összeadás később fejeződik be, és felülírja a D2 regiszter tartalmát.

Utasítás	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
$D2 \leftarrow D1 + D3$	IF	ID	A0	A1	A2	A3	MEM	WB
…		IF	ID	EX	MEM	WB		
$D2 \leftarrow MEM[R0+4]$			IF	ID	EX	MEM	WB	

A WAW függőségek detektálása ismét az ID fázis feladata, és mint az összes többi egymásrahatás esetén, itt is a futószalag megállítása és a további utasítások lehívásának szüneteltetése a legegyszerűbb megoldás. Ebben a példában 2 ciklusidőnyi szünetet kell beiktatni, hogy a WB fázisok sorrendje megfelelő legyen.

Némi meggondolás után beláthatjuk, hogy a WAR függőségek nem okoznak gondot, mert az utasítások végrehajtása sorban kezdődik el, így azok a regisztereket is a helyes sorrendben olvassák ki.

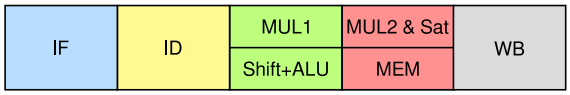
13.7. Alternatív pipeline struktúrák

A tárgyalt 5 fokozatú pipeline természetesen csak egy a sok lehetséges struktúra közül. Minden processzor gyártó meghozza a maga döntéseit és kompromisszumait a megfelelő pipeline tervezése során. A fokozatok száma nem feltétlenül 5, lehet kevesebb, vagy több is. A feldolgozási fázisok sorrendje is döntés kérdése, bizonyos keretek között. Az alábbiakban, gyakorlati példaként két ARM processzor pipeline-ját mutatjuk be.

Az ARM926EJ-S processzor pipeline-ja

Az ARM926EJ-S processzor 5 fokozatú pipeline-al rendelkezik, melynek felépítése nagyban hasonlít a megismert demonstrációs célú pipeline-hoz. A fázisok az alábbiak (lásd 13.12. ábra):

1. IF: Utasításlehívás
2. ID: Dekódolás, majd az órajelciklus második felében az operandusok kiolvasása a regisztertárolóból
3. EX: Az aritmetikai és bitműveletek, valamint egy opcionális bitenkénti shift-elés történik ebben a fázisban, amennyiben nem szorzás műveletről van szó. A szorzás ugyanis két órajelciklust igényel, melyből az első az EX fázisra esik.
4. MEM: Itt történnek a Load/Store utasításokhoz kapcsolódó memóriaműveletek. Ez a fázis azonban az aritmetikai műveletek esetén (melyek nem igénylik a memóriához fordulást) sem telik tétlenül. Egyrészt a szorzás műveletek a MEM fázis alatt fejeződnek be. Másrészt az ARM képes telítődő aritmetikával, azaz túlcsordulás nélkül is számolni (pl. -2^{31} csökkentése esetén nem pörög át az eredmény $2^{31}-1$-be, hanem marad -2^{31}), és a telítődés ellenőrzése és kezelése szintén a MEM fázisban történik meg.
5. WB: Az eredmények visszaírása a regisztertárolóba, mint a korábbi 5 fokozatú minta pipeline-unknál.



13.12. ábra. Az ARM926EJ-S processzor pipeline-ja

Az ARM1176JZF-S processzor pipeline-ja

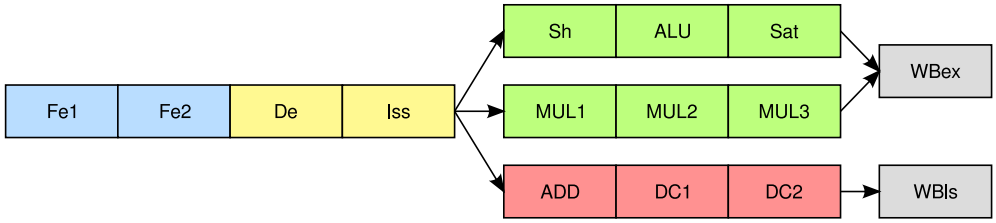
Az ARM1176JZF-S processzor, dolgozik többek között a Raspberry Pi miniszámítógépben is, 8 mélységű pipeline-al rendelkezik (13.13. ábra), de a tanult 5 fázis ebben a pipeline-ban is ugyanúgy tetten érhető.

1. Fe1: Az utasításlehívás első fázisa, a cím kiadása a memóriának (ill. az utasítás cache-nek), és a válaszként érkező utasításszó átvétele
2. Fe2: Az utasításlehívás második fázisa, többek között az elágazásbecslés ekkor történik meg
3. De: Dekódolás
4. Iss: Operandusok kiolvasása a regisztertárolóból, valamint az utasítás továbbirányítása a megfelelő feldolgozóegység felé

Az utasítások feldolgozására három műveleti egység áll rendelkezésre:

- Aritmetikai műveletek esetén az utasítás végrehajtása négy további fázist igényel:
  1. Sh: egy bitenkénti shift művelet végrehajtását (amit az ARM utasításkészlet minden aritmetikai művelethez csatolhatóan megenged),
  2. ALU: magának az aritmetikai/logikai művelet végrehajtását,
  3. Sat: az eredmény potenciális telítődésének kezelését,
  4. WBex: az eredmény visszaírását a regiszter tárolóba.
- Szorzás esetén szintén négy fázisra van szükség:

  1.-3. MUL1, MUL2, MUL3: a szorzás elvégzése,
  4. WBex: az eredmény visszaírása a regiszter tárolóba.
- A memóriaműveletek végrehajtása szintén négy fázisban történik:
  1. ADD: címszámítás,
  2. DC1: adat cache kiolvasása vagy írása, első fázis,
  3. DC2: adat cache kiolvasása vagy írása, második fázis,
  4. WBls: az eredmény visszaírása a regiszter tárolóba (load művelet esetén).



13.13. ábra. Az ARM1176JZF-S processzor pipeline-ja

Az aritmetikai műveletek és a szorzó pipeline nem képes párhuzamos működésre, míg a load/store pipeline igen.

14. fejezet

Pipeline utasításfeldolgozás dinamikus ütemezéssel

14.1. Motiváció: utasítássorrend-optimalizálás

Egy program utasításainak megfelelő átrendezésével sok esetben csökkenteni lehet a futási időt, hiszen egy szerencsés utasítássorrend végrehajtása közben esetleg kevesebb pipeline szünet beiktatására lehet szükség, mint egy kedvezőtlenebb sorrend esetén. Az átrendezéssel természetesen megfelelő körültekintéssel kell bánni, hogy a program végeredménye ne változzon, továbbra is a programozó szándékának megfelelő tevékenységet végezze. Az utasítássorrend-optimalizálás demonstrálására lássuk egy példát. Alább egy C nyelven megírt ciklust láthatunk:

```
for (i=0; i<N; i++)
    Z[i]=A*X[i];
```

a ciklusmag alacsony szintű nyelvre fordított változata pedig a következő:

```
i1: D2 ← MEM[R1]
i2: D3 ← D2 * D0
i3: MEM[R2] ← D3
i4: R1 ← R1 + 8
i5: R2 ← R2 + 8
```

Tegyük fel, hogy ezt az utasítássorozatot egy olyan utasítás pipeline segítségével hajtjuk végre, melyben az egész- és a memóriaműveletek 5 fázist igényelnek (IF, ID, EX, MEM, WB), a lebegőpontos szorzás esetén azonban 9 fázis szükséges (IF, ID, M0, … , M4, MEM, WB). Az utasítások ütemezése a 14.1. ábrán látható. Feltételezzük, hogy az `A` változót a `D0` regiszter, az `X[i]`-re, valamint a `Z[i]`-re pedig az `R1`, illetve `R2` regiszter mutat.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
`i1:`	IF	ID	EX	MEM	WB									
`i2:`		IF	ID	*A**	M0	M1	M2	M3	M4	MEM	WB			
`i3:`			IF	*F**	ID	*A**	*A**	*A**	*A**	EX	MEM	WB		
`i4:`					IF	*F**	*F**	*F**	*F**	ID	EX	MEM	WB	
`i5:`										IF	ID	EX	MEM	WB

14.1. ábra. A ciklusmag utasításainak ütemezése

Az utasítássorozat végrehajtásához 14 órajelciklusra volt szükség. A futási időt két tényező növelte: az első két utasítás közötti RAW függőség (1 ciklusnyi szünet), valamint a második és harmadik utasítás közötti RAW

függőség (4 egységnyi szünet). Mindkét esetben az jelenti a problémát, hogy az utasításoknak várakozniuk kell, amíg az operandusaik rendelkezésre nem állnak. Adódik tehát a lehetőség, hogy az adatfüggésben lévő sorok közé tőlük független sorokat mozgassunk, hogy addig is hasznos munkával teljen az idő, amíg az operandusra várakozó utasításra nem kerül a sor. Példánkban a legtöbb, amit tehetünk, hogy harmadik és a negyedik sort felcseréljük, amivel az alábbi utasítássorozatot kapjuk:

```
i1': D2 ← MEM[R1]
i2': D3 ← D2 * D0
i3': R1 ← R1 + 8
i4': MEM[R2] ← D3
i5': R2 ← R2 + 8
```

Az átrendezett utasítássorozat végrehajtása 14 helyett csak 13 ciklus vesz igénybe (14.2. ábra). Hosszabb utasítássorozatok esetén természetesen ennél nagyobb mértékű gyorsulás várható, hiszen több a lehetőség az adatfüggésben lévő utasítások eltávolítására.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
i1':	IF	ID	EX	MEM	WB									
i2':		IF	ID	A^*	M0	M1	M2	M3	M4	MEM	WB			
i3':			IF	F^*	ID	EX	MEM	WB						
i4':					IF	ID	A^*	A^*	A^*	EX	MEM	WB		
i5':						IF	F^*	F^*	F^*	ID	EX	MEM	WB	

14.2. ábra. Az átrendezett utasítássorozat ütemezése

Az adatfüggőségek felderítésével és az adatfüggőségben lévő utasítások eltávolításával tehát optimalizálható a programok futási ideje. Ezzel a megközelítéssel azonban van két probléma: egyrészt a lehetőségek kihasználása a programozón (illetve, magas szintű nyelvek esetén a fordítóprogram képességein) múlik, másrészt pedig különböző pipeline struktúrák esetén különböző lesz az optimális utasítássorrend. Az lenne a legjobb megoldás, ha az utasítások átrendezését maga a processzor végezné, automatikusan, a program futása közben. Hiszen az ilyen processzorra nagyon kényelmes lenne programot írni, nem kellene állandóan szem előtt tartani az utasítássorrend fontosságát. Ha ezzel a képességgel rendelkezik a processzor, akkor *soron kívüli utasítás-végrehajtásról* beszélünk.

Amilyen mágikusnak tűnik is, a soron kívüli utasítás-végrehajtás nem egy nagyon bonyolult eljárás, és nem is napjaink vívmánya. Az első erre képes számítógép 1964-ben megjelent, Seymour Cray által tervezett CDC 6600-as volt. Azonban az Intel legújabb Core i7 processzorai is az 1967-ben publikált, egy későbbi fejezetben részletesen tárgyalt Tomasulo algoritmus szerint működnek.

14.2. A soron kívüli végrehajtás alapelemei

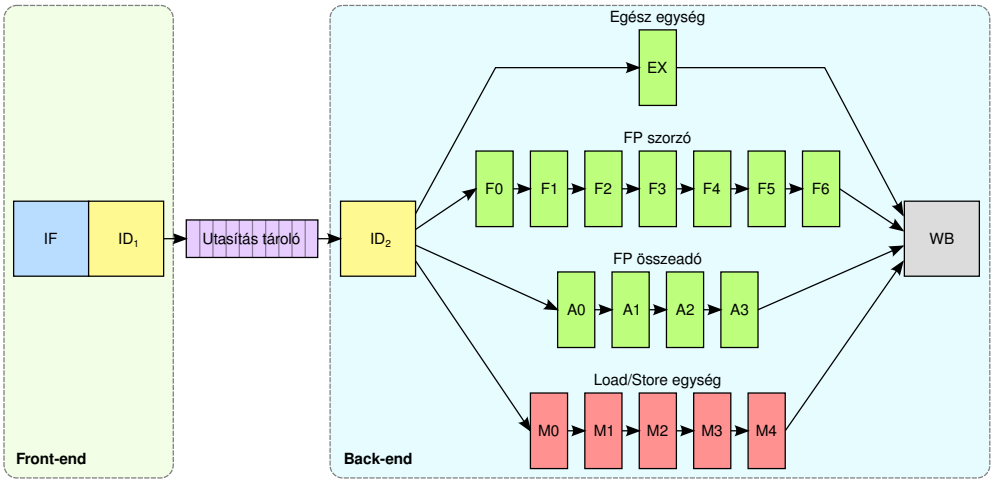
A sorrenden kívüli végrehajtás a sorrendi végrehajtáshoz képest lényegesen eltérő megoldásokat igényel. Az utasításfeldolgozó futószalag nem futószalag többé, hiszen a későbbi utasítások - ha végrehajtásuknak semmi akadálya nincs - megelőzhetik a korábbiakat. Az utasítások sorrenden kívüli végrehajtása során figyelembe kell venni azt is, hogy a program szemantikája nem változhat. Az ehhez szükséges logika: nyilvántartások az utasítások végrehajtási állapotáról és a műveleti egységek foglaltságáról, valamint az utasításokat a műveleti egységekhez hozzárendelő logika a processzort bonyolulttá teszi.

14.2.1. Az utasítástároló

Ahhoz, hogy a processzor az utasításokat a program szerintitől eltérő sorrendben hajtsa végre, egyszerre több utasításnak is a látóterében kell lennie. Ezeket az utasításokat az *utasítástároló* (reservation station, instruction pool, instruction window) tárolja. Az utasítástároló nem lehet máshol, mint az ID fázisban, hiszen az utasítások lehívása és dekódolása még program szerinti sorrendben történik, míg a végrehajtás (EX) már soron kívül. Az

utasítástároló az ID fázist két részre bontja: az utasítás dekódolásáért és az utasítástárolóba helyezésért felelős részt *Dispatch*-nek (DS), az operandusok összeszedését és a megfelelő végrehajtó egység felé továbbítást végző részt pedig *Issue*-nak (IS) hívják.

Az utasítástároló nemcsak a dekódolási fázist, hanem az egész pipeline-t is két részre bontja. Az utasítások letöltéséért és előkészítéséért felelős pipeline fázisok (IF és DS) alkotják a *front-end*-et, a tényleges végrehajtást végző fázisok pedig a *back-end*-et. A front-end sorrendi (in-order), a back-end soron kívüli (out-of-order) feldolgozást követ. Az ily módon módosított pipeline a 14.3. ábrán látható, melyen egy további változás is megfigyelhető: out-of-order processzorokban gyakori megoldás, hogy a Load/Store utasítások külön műveleti egységet kapnak saját címszámolóval, így az aritmetikai műveleteknek nem kell feleslegesen áthaladnia a MEM fázison.



14.3. ábra. Out-of-order pipeline

Az ábrán látható utasítástároló *centralizált*, mivel egy darab van belőle a processzorban, és ez az egy minden utasítást tárol. Egyes processzorok *elosztott* utasítástárolót alkalmaznak, melyben minden műveleti egységhez tartozik egy saját, csak az oda tartó utasítások számára fenntartott tároló. A gyakorlatban használt kialakítást, illetve méreteket a 14.4. ábra foglalja össze.

Processzor	Utasítástároló típusa	Utasítástároló mérete
IBM Power4	Elosztott	31
Intel Pentium III	Centralizált	20
Intel Pentium 4	Hibrid (1 mem. műv., 1 többi)	126 (72, 54)
Intel Haswell	Centralizált	56
AMD K6	Centralizált	72
AMD Opteron	Elosztott	60

14.4. ábra. Néhány processzor utasítástárolójának paraméterei

14.2.2. A dinamikus ütemező

A *dinamikus ütemező* feladata, hogy az utasítástárolóból egy utasítást végrehajtásra kiválasszon. Azért dinamikus, mert nem a program szerinti sorrendet követi, hanem a pillanatnyi helyzetnek legmegfelelőbb döntést hozza. A dinamikus ütemező adatáramlásos elven működik. Azon utasítások közül válogat, melyek készek a végrehajtásra, vagyis minden operandusuk rendelkezésre áll, és van fogadóképes szabad műveleti egység.

A dinamikus ütemező (leegyszerűsítve) egy precedenciagráfot tart karban. A precedenciagráf csomópontjai az utasítástárolóban lévő utasítások, a csomópontból induló irányított élek pedig megmutatják, hogy mely

más utasításokat kell bevárni, mielőtt a végrehajtás elkezdődik. Az élek tulajdonképpen az adatfüggőségeket reprezentálják. Egy utasítás még nem hajtható végre, ha

- egyik operandusa még nem áll rendelkezésre (RAW függőség),
- vagy ugyanabba a regiszterbe írná az eredményt, mint egy korábbi, még nem végrehajtott utasítás (WAW függőség),
- vagy olyan regiszterbe írná az eredményt, ahonnan egy korábbi utasítás még nem olvasta ki a régi értéket (WAR függőség).

A precedenciagráf minden egyes új belépő utasítás dekódolásakor bővül. A dinamikus ütemező olyan utasítást (precedenciagráf csomópontot) választ végrehajtásra, amelyik minden kimenő éle már befejeződött utasításra mutat. Egy adott ciklusban az is előfordulhat (sőt, ez a tipikus helyzet), hogy egy adott műveleti egységre váró több utasítás is kész a végrehajtásra. Ilyenkor azt mondjuk, hogy *versengés* van a műveleti egységért, és az ütemező az implementációk többségében a legöregebb utasítást választja, de vannak kifinomultabb algoritmusok, melyek azt az utasítást választják, melynek szerepe kritikus az utasítássorozat végrehajtása szempontjából (pl. sokakat feltart, jobb hamar túlesni rajta).

A dinamikus ütemezés szemléltetéséhez térjünk vissza a korábbi ciklushoz, és kövessük végig két iteráció futását:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
i5:  R2 ← R2 + 8
i6:  D2 ← MEM[R1]
i7:  D3 ← D2 * D0
i8:  MEM[R2] ← D3
i9:  R1 ← R1 + 8
i10: R2 ← R2 + 8
```

A precedenciagráf alakulása a 14.5. ábrán látható. Az egyszerűség kedvéért most minden utasítás csak 3 állapotban lehet: betöltve az utasítástárolóban (kék), futó állapotban (piros), és kész állapotban (zöld). Az egész és a memóriaműveletek végrehajtása tartson 1, a lebegőpontos szorzásé pedig 5 órajelig. A pipeline front-endje minden ciklusban egy új utasítást hív le és helyez el az utasítástárolóban, tehát minden körben egy új kék csomópont jelenik meg a precedenciagráfban, a függőségek által meghatározott élekkel együtt. Ha van olyan csomópont, aminek minden függősége feloldódott, akkor az végrehajtható. A második lépésben például `i1` végrehajtása zajlik, ami a harmadik lépésre be is fejeződik, ezzel lehetővé válik `i2` elkezdése. Az `i2` sokáig tart, és több utasítás is függ tőle, ezért az utasítástárolóban elkezdenek felgyűlni a várakozó utasítások. Amikor az `i4` megjelenik, kiderül, hogy rögtön végrehajtható, így az megelőzi az `i3` végrehajtását. Itt tehát máris tetten érhető az out-of-order végrehajtás. Akárcsak a nyolcadik lépésben, ahol `i4` már befejeződött, de `i5` még el sem kezdődött.

Fontos megjegyezni, hogy a dinamikus ütemezés alapvetően tér el a 13.6. fejezetben látott pipeline-tól, ahol program szerinti sorrendben kezdődött az utasítások végrehajtása, de soron kívül fejeződött be. A dinamikus végrehajtás mellett az utasítások elkezdési sorrendje is out-of-order lehet.

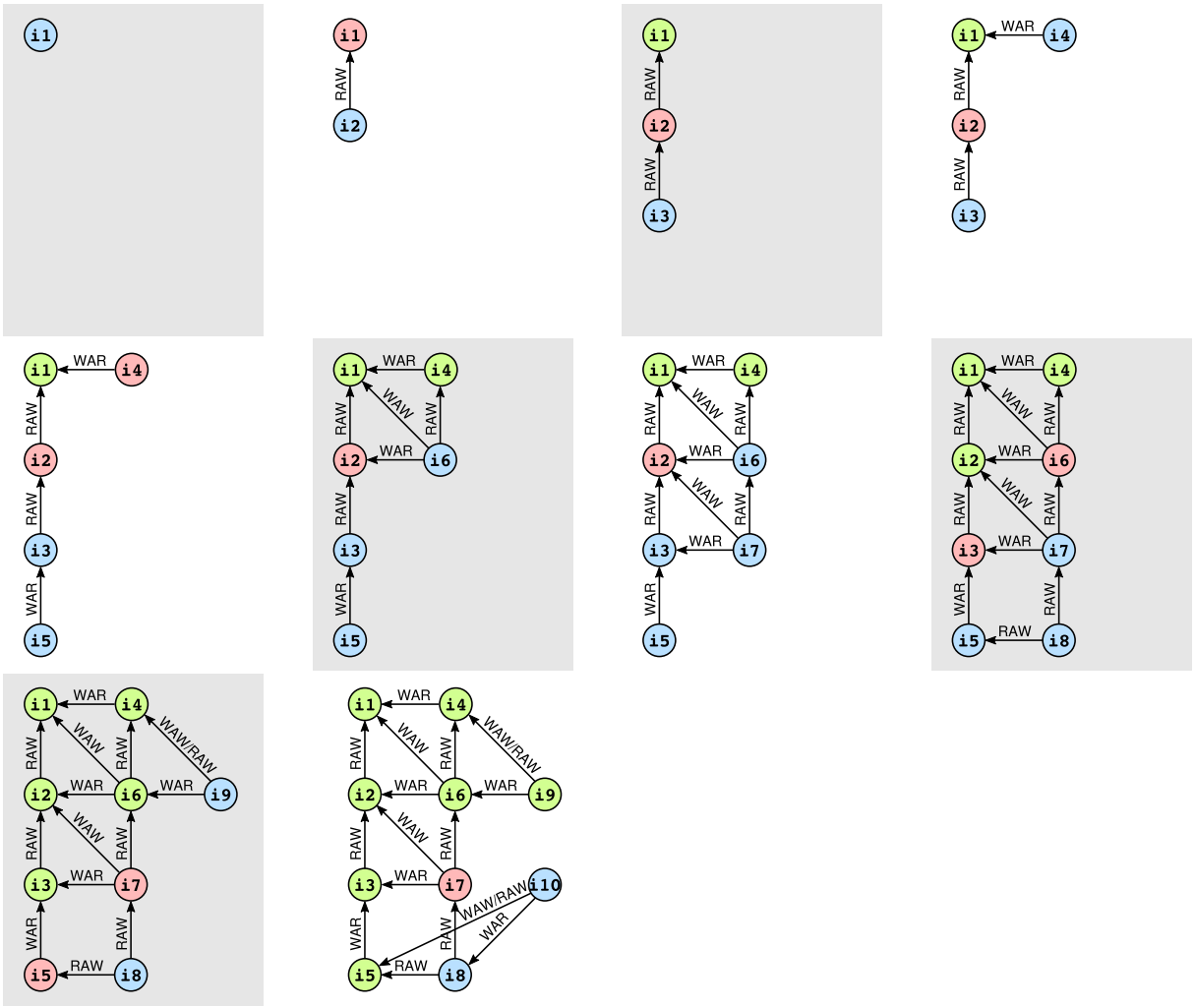
14.2.3. Regiszterátnevezés

Az out-of-order utasítás-végrehajtás hatékonyságát nagyban befolyásolja, hogy az utasítások precedenciagráfja mennyire sűrű. Ha sok a függőség, akkor az utasítások egymásra várakoznak, ha azonban kevés az él, akkor gyakran talál az ütemező végrehajtható utasítást.

A függőségi gráf megritkításához a program bizonyos függőségeit meg kell szüntetni. Tekintsük át ismét a függőségek típusait, és azonosítsuk, hogy miért is alakulnak ki.

RAW függőség esetén egy utasítás felhasználja egy korábbi utasítás eredményét:

```
D3 ← D2 * D0
...
MEM[R2] ← D3
```



14.5. ábra. A precedenciagráf karbantartása és a dinamikus ütemezés

A RAW függőségek a probléma természetéből fakadnak, azt megszüntetni a program szemantikájának megsértése nélkül nem lehet. Ezeket a függőségeket nem lehet kiiktatni, ezért *valódi függőségnek* nevezik őket.

A WAR függőség azt jelenti, hogy egy utasítás olyan regiszterbe írja az eredményét, amelyből egy korábbi utasítás olvas:

```
D3 ← D2 * D0
...
D2 ← MEM[R1]
```

illetve, WAW függőséget akkor kapunk, ha két utasítás ugyanabba a regiszterbe kívánja írni az eredményét:

```
D3 ← D2 * D0
...
D3 ← MEM[R1]
```

A WAR és WAW függőségek csakis azért állnak fenn, mert a programozó újra felhasznál egy korábban már használt regisztert. Ha mindig új regiszterbe tenné az utasítások eredményét, akkor sem WAR, sem WAW függőség nem lenne a programban. Természetesen a regiszterek száma erősen korlátozott (pl. x86 architektúrában 8, ARM-ban 16 van belőlük), ezért újrafelhasználásuk nem úszható meg. A WAR és WAW függőségeket regiszterátnevezés segítségével meg lehet szüntetni, így ezeket *álfüggőségeknek* (anti-dependency) hívják.

Az álfüggőségek megoldásának legkézenfekvőbb módja az lenne, ha a gyártók a processzoraik újabb és újabb generációiba egyre több regiszter használatát tennék lehetővé. Erre azonban nincs mód, hiszen a regiszterek számát az utasításkészlet architektúra rögzíti, és a kompatibilitás fenntartása végett ettől eltérni nem lehet.

A regiszterátnevezés lényege, hogy a processzorban a programozó számára láthatatlan módon nagy számú *fizikai regisztert* alakítanak ki. A processzor az eredetileg *architekturális regisztereket* használó utasításokat rögtön a lehívás után átírja úgy, hogy azok a fizikai regisztereket használják, és hogy a program ténylegesen profitáljon azok nagy számából.

A regiszterátnevezéshez két adatszerkezet szükséges: egy *regiszterleképző tábla*, amely az architekturális regiszter ↔ fizikai regiszter leképezéseket tartalmazza, valamint a szabad fizikai regiszterek listájára. Az algoritmus formálisan megfogalmazva a következőképpen működik:

```
Az  Ri ← Rj műv. Rk utasítás regiszter átnevezés után
 Ra ← Rb műv. Rc utasítássá válik, ahol:
    Rb = RegiszterLeképzőTábla[Rj],
    Rc = RegiszterLeképzőTábla[Rk],
    Ra = A következő szabad fizikai regiszter,
majd frissítés: RegiszterLeképzőTábla[Ri]= Ra.
```

A 14.7. ábrán láthatjuk a példaként használt utasássorozat eredeti és regiszterátnevezés utáni változatát. A regiszterleképző tábla alakulását a 14.6. ábra követi, melynek első oszlopa (”Kezdeti”) a kiindulási állapotot tükrözi. Most jelölje az egész típusú fizikai regisztereket `T` kezdőbetű, a lebegőpontos típusúakat pedig `U` kezdőbetű. A regiszterátnevezés minden egyes utasításra teljesen szisztematikus módon történik. Az első utasítás lehívása után a processzor a táblázat alapján átnevezi a forrásregisztert (`R1`→`T3`), majd az eredményt egy használaton kívüli, ”tiszta” regiszterbe írja (`U25`). Ezután frissíteni kell a regiszterleképző táblát, hogy a jövőben a `D2`-t használó utasítások az `U25`-ből olvassák ki a tényleges értéket (a táblázat ”1”-es oszlopa). A második utasítással is ugyanígy járunk el. Először az utasítás jobb oldalán, a forrásregisztereket kell átnevezni a táblázat alapján, majd az eredmény ismét tiszta regiszterbe kerül, végül a táblázatot is frissíteni kell. A harmadik (store) utasítás annyiban különleges, hogy általa nem változnak a regiszterek, így a harmadik lépésben a táblázat frissítésére nincs szükség.

Arch.	Kezdeti	1	2	3	4	5	6	7	8	9	10
R0	T21										
R1	T3				T47					T49	
R2	T46					T48					T50
R3	T8										
D0	U9										
D1	U24										
D2	U17	U25					U27				
D3	U4		U26					U27			

14.6. ábra. A regiszterleképző tábla alakulása a regiszterátnevezés során

A processzor természetesen nem a 14.6. táblázatot tárolja, hanem csak annak az éppen aktuális oszlopát.

A regiszterátnevezésnek köszönhetően eltűntek a WAR és WAW függőségek, hiszen soha nem írunk olyan regiszterben, amit korábban már használtunk. (Természetesen egy valós rendszerben nem áll rendelkezésre végtelen számú fizikai regiszter, de a már nem használt regisztereket ismét fel lehet szabadítani.)

Csak a RAW függőségek maradtak, aminek köszönhetően a precedenciagráf lényegesen ritkábbá vált (lásd 14.8. ábra), és ez a futási időre is jótékony hatással van. Az utasítások állapotának alakulását a 14.9. ábrán követhetjük végig. Az első öt lépésben nincs különbség, de a hatodikban már van: a ciklus második iterációja is elkezdhetővé vált, hiszen az elsőtől különböző regisztereket használ. A nyolcadik lépésre az átnevezett kód már 5 utasítást befejez, és 1 várakozik, míg az eredeti esetben ugyanekkor csak 3 utasítás van kész, és 3 várakozik. Természetesen egy éles rendszerben, ahol több utasítás is a processzor látóterében van, a regiszterátnevezés haszna még szembetűnőbb.

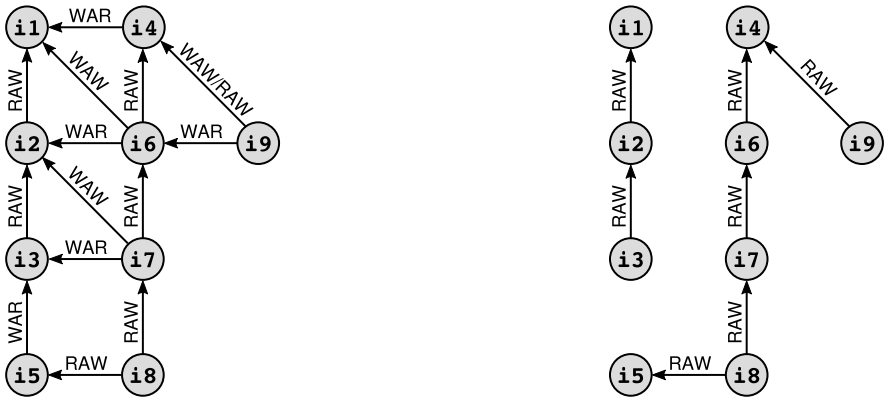
Eredeti kód:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
i5:  R2 ← R2 + 8
i6:  D2 ← MEM[R1]
i7:  D3 ← D2 * D0
i8:  MEM[R2] ← D3
i9:  R1 ← R1 + 8
i10: R2 ← R2 + 8
```

Regiszterátnevezés után:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
i4:  T47 ← T3 + 8
i5:  T48 ← T46 + 8
i6:  U27 ← MEM[T47]
i7:  U28 ← U27 * U9
i8:  MEM[T48] ← U28
i9:  T49 ← T47 + 8
i10: T50 ← T48 + 8
```

14.7. ábra. A példaprogram regiszterátnevezés előtt és után



a) Regiszterátnevezés előtt b) Regiszterátnevezés után

14.8. ábra. Precedenciagráf a regiszterátnevezés előtt és után

14.2.4. A sorrend-visszaállító buffer

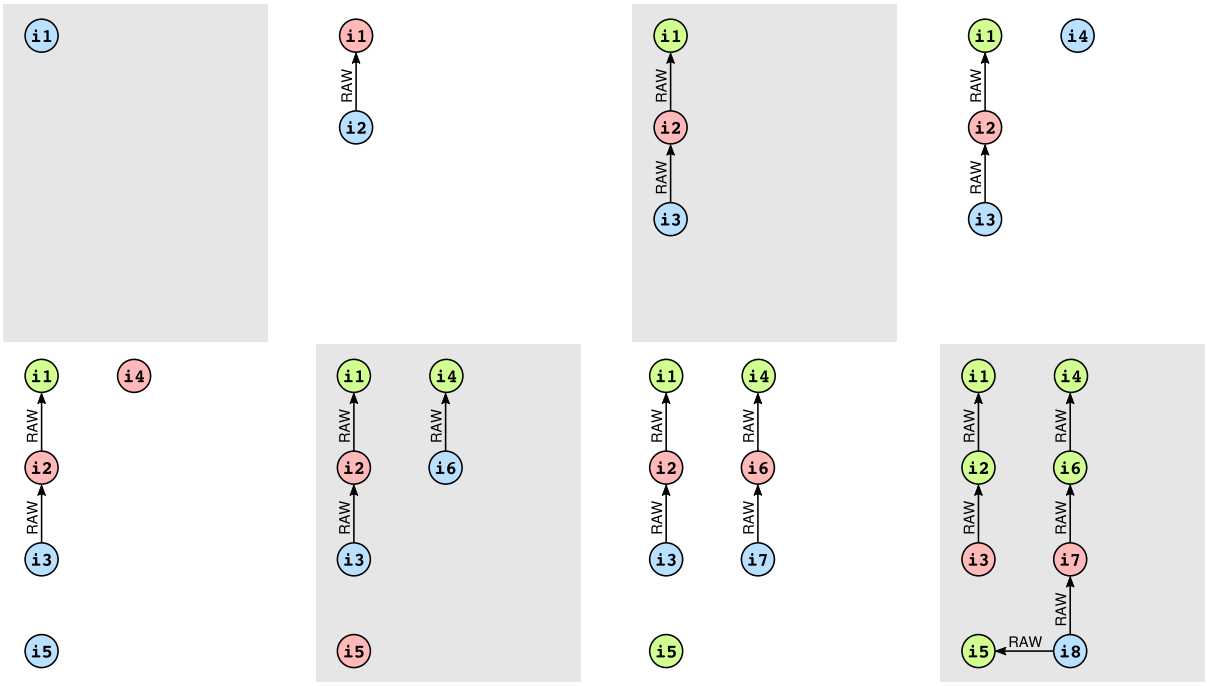
A soron kívüli végrehajtásnak van egy kellemetlen mellékhatása: a program futása során a memória-, illetve regisztertartalom nem a program által előírt sorrendben változik. A végén a végeredmény persze jó lesz, ha a dinamikus ütemező tiszteletben tartja a precedenciagráfot, de a részeredmények nem feltétlenül a program alapján várt sorrend szerint gyűlnek a memóriában. Vannak esetek, amikor ez semmilyen kellemetlenséget nem okoz. Ha azonban a regiszterkészletre, illetve a memóriatartalomra a számítógép más komponensének is van rálátása, akkor óvatosabbnak kell lenni, hiszen nem biztos, hogy fel van készítve erre a helyzetre. Márpedig egy modern számítógépben a memória egy osztott erőforrás:

- Többmagos processzorok, illetve multiprocesszoros rendszerek esetén a memóriatartalmat mindegyik mag, illetve processzor nyomon tudja követni, és a sorrenden kívüli végrehajtást tetten tudja érni.
- A memóriához a perifériák is közvetlenül hozzáférnek, DMA segítségével.

Ugyanezzel a problémával szembesülünk egy megszakításkérés esetén. A megszakításkezelő szubrutin nem feltétlenül van felkészítve arra az esetre, hogy a számítógépen olyan állapotban találja, hogy pár későbbi utasítás már lefutott, míg pár korábbi még el sem kezdődött.

A legbiztosabb megoldás, ha a processzor mindent megtesz annak érdekében, hogy a sorrenden kívüli működését a külvilág elől elrejtse, és sorrendi végrehajtás látszatát keltse. Ennek eszköze a *sorrend-visszaállító buffer* (reorder buffer, ROB).

A ROB egy FIFO adatszerkezet. Amikor egy utasítás a DS fázis során bekerül az utasítás tárolóba, egyben egy bejegyzést is kap a ROB-ban. Mivel a DS fázist az utasítások a program szerinti sorrendben hagyják el, a ROB-ban a hozzájuk tartozó bejegyzések is program szerinti sorrendben jelennek majd meg. Egy utasításhoz tartozó ROB bejegyzés az alábbi információkat tartalmazza:



14.9. ábra. Dinamikus ütemezés a regiszterátnevezés után

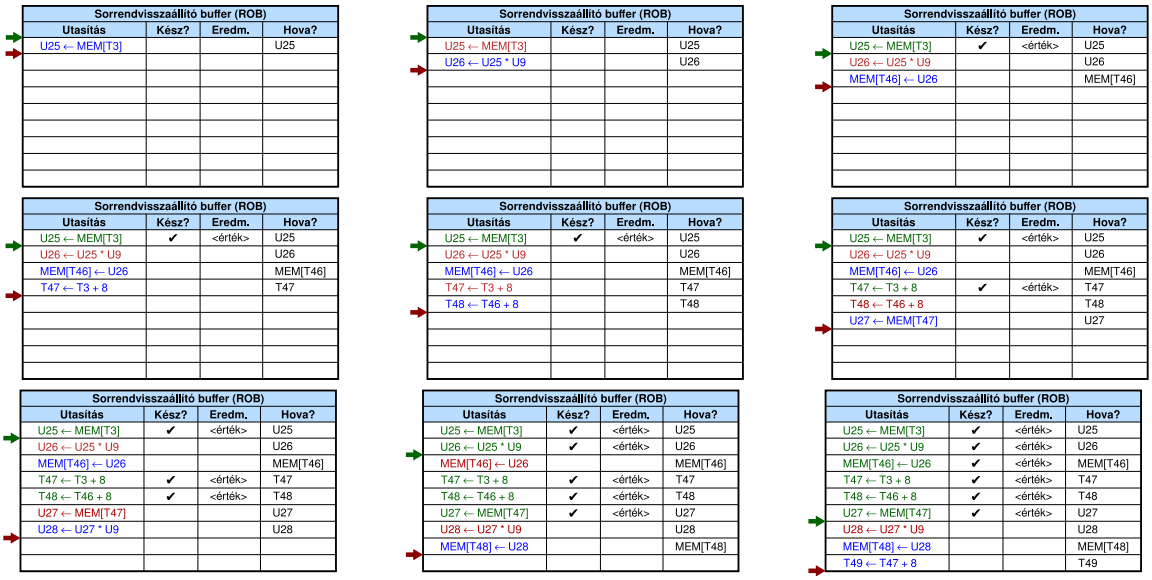
- ”Ready” bit: az utasítás végrehajtása kész-e. A további 3 mezőnek csak akkor van értelmes értéke, ha Ready=1.
- Az utasítás eredménye (maga az érték)
- Aritmetikai vagy Load utasítás esetén: a regiszter (logikai) neve, ahová az eredményt kellene írni
- Store utasításnál: a memóriacím, ahová a megadott értéket be kell írni

Amikor a processzor végrehajtott egy utasítást, annak eredményét nem vezeti át rögtön az eredményregiszterbe, vagy – store utasítás esetén – a memóriába, hanem ideiglenesen a sorrend-visszaállító bufferben tárolja.

Mivel sorrenden kívüli végrehajtásról van szó, az eredmények a sorrend-visszaállító bufferbe is sorrenden kívül kerülnek, össze-vissza jelennek meg a Ready=1 bittel rendelkező bejegyzések. A sorrend-visszaállító bufferben ideiglenesen tárolt eredményeket csak akkor fogja a processzor a tényleges rendeltetési helyére átvezetni, ha minden (program szerint) korábbi utasításra már megtette azt. Tehát miközben zajlik az utasítások végrehajtása, a ROB szépen sorban, a FIFO elvnek megfelelően érvényesíti az elkészült utasítások által előírt regiszter vagy memória írásokat. Ha olyan bejegyzéshez ér, amelyben Ready=0, akkor ott megáll,és megvárja, amíg elkészül. Ezzel az egyszerű megoldással garantálható, hogy minden egyes részeredmény a program szerinti sorrendben fog feltűnni a memóriában.

A gyakorlatban a sorrend-visszaállító buffer miatt a WB fázist kétfelé szokták bontani: a végrehajtás kész (complete, CM) fázisban történik az utasítás eredményének beírása a ROB-ba, a kilépési fázisban (retire, RT) pedig az utasítás ROB bejegyzésének az érvényesítése.

A sorrend-visszaállító buffer működését a 14.10. ábra mutatja be, az eddig követett példán keresztül. Minden ciklusban egy újabb utasítás lép be a processzorba, és kap új helyet a ROB-ban. Az ábrán a ROB-ban lévő első, ill. utolsó érvényes utasítást a zöld, ill. a piros nyíl jelöli. Az utasítás belépésekor a ROB-ba kerül az eredmény rendeltetési helye is. A második lépésben elindul az első utasítás végrehajtása, és bejegyzésre kerül a második. A harmadik lépésben az első utasítás befejeződik, az eredménye a ROB-ba kerül, és mivel minden korábbi utasítás már érvényre jutott (mivel ez az első), ezért ez az eredmény beíródik a rendeltetési helyére, majd az utasítás elhagyja a ROB-ot. A második utasítás lassú, végrehajtása 5 ciklust igényel. A már megismert sorrenden kívüli végrehajtás elve szerint közben megelőzi őt a negyedik és az ötödik utasítás is, ezek eredménye is a ROB-ba kerül. Ezeket az eredményeket még nem lehet a rendeltetési helyre írni, hiszen akkor hamarabb jutnának érvényre, mint a még be nem fejezett második és harmadik utasítás – és pont ezt szeretnénk elkerülni. Ez a két eredmény csak a kilencedik lépésre jut érvényre, amikor már minden korábbi utasítás eredménye érvényre jutott.



14.10. ábra. Példa a sorrend-visszaállító buffer alakulására

A példán még egy fontos észrevételt lehet tenni. A hetedik lépésben indul a `U27 ← MEM[T47]` utasítás végrehajtása, de ekkor T47 aktuális értéke még nem a regisztertárolóban, hanem a ROB negyedik sorában található, tehát onnan kell kiolvasni azt.

14.2.5. Spekulatív végrehajtás

Alapesetben az utasítások átrendezésének gátat szabnak a feltételes ugró utasítások. Amíg az ugrási feltétel nem kerül kiértékelésre, a processzor nem tudja, hogy merre folytatódik a program, így nem tud újabb utasításokat lehívni és bevonni a dinamikus ütemezésbe. Alapesetben tehát az utasítások átrendezése a feltételes ugró utasítások által határolt programrészekre, az úgynevezett *basic block*-okra korlátozódik.

Sajnos a valós programokban jellemzően nagyon sok a feltételes ugrás, ami rontaná a sorrenden kívüli pipeline hatékonyságát. A megoldást a *spekulatív végrehajtás* jelentheti. Ebben az esetben a processzor elágazásbecslés segítségével megtippeli, hogy a feltételes ugrás merre viszi tovább a programot, és ezt a becslést készpénznek is veszi: elkezdi lehívogatni a megtippelt ágról az utasításokat, és bevonja azokat a dinamikus ütemezésbe. Természetesen időnként előfordul, hogy a becslés rossz volt, amikor ténylegesen kiértékelődik az ugrási feltétel. Ekkor módot kell teremteni a tévedésből lehívott és végrehajtott utasítások hatásának semmissé tételére.

Ha van sorrend-visszaállító buffer, akkor a spekulatív végrehajtás viszonylag egyszerűen megvalósítható. A processzor bátran végrehajthatja a megtippelt irányból származó utasításokat, de a ROB-ba azt is be kell jegyeznie, hogy azok melyik feltételes ugrás melyik irányából származnak. Az utasítások eredménye, ahogy korábban láttuk, először a ROB-ba kerül, és nem az utasítás szerinti rendeltetési helyére. Csak annyival kell kiegészíteni a működést, hogy a spekulatív utasításokat nem engedjük érvényre jutni (az eredményüket a ROB-ból a rendeltetési helyére írni), amíg be nem bizonyosodik, hogy a tipp, aminek a révén a processzorba jutottak, helyes volt.

14.2.6. Memóriaműveletek adat-egymásrahatása

Az eddigiekben az egyszerűség kedvéért nem vettük figyelembe, hogy adat-egymásrahatás nemcsak a regisztereken dolgozó aritmetikai utasítások között, hanem a memóriaműveletek között is lehet. Nézzük a következő utasítássorozatot:

```
i1:  R2 ← R1 + 4
i2:  D0 ← D1 / D2
i3:  MEM[R1+8] ← D0
i4:  D3 ← MEM[R2+4]
```

Ebben a példában az `i3` és `i4` memóriaműveletek között RAW egymásrahatás van, hiszen az `i4`-ben az olvasás ugyanarról a memóriacímről történik, ahova az `i3` ír. Sorrenden kívüli végrehajtás esetén, hacsak ez ellen nem teszünk intézkedéseket, az `i4` hamarabb futhat le, mint az `i3` (mert az `i3`-at feltartja az `i2`, ami egy lassú lebegőpontos osztás), ezzel a `D3` regiszter értéke nem az lesz, mint amit a program sorrendi végrehajtásával kapnánk.

A regiszterek közötti egymásrahatások a dekódolási fázisban *észrevehetők* és várakoztatással kezelhetők voltak. Memóriaműveletek esetén nem ilyen egyszerű a helyzet. Ha a processzor Load utasítással találkozik, további megfontolásokat kell tennie, hogy vajon sorrenden kívül végrehajthatja-e, esetleg egy memóriára vonatkozó adat-egymásrahatást kockáztatva. A processzornak a következőket kell ellenőriznie:

1. Ha a ROB-ban nincs a kérdéses Load előtt egyetlen Store sem, akkor a Load végrehajtható, legalábbis ez a fajta memóriára vonatkozó adat-egymásrahatás nem fordulhat elő.

2. Ha a ROB-ban a korábbi utasítások között egy olyan befejezett Store-t talál, amely ugyanarra a címre vonatkozik, mint a Load, akkor a Load-nak nem kell a memóriához fordulnia, még csak a cache-hez sem, mert a keresett adat megvan a ROB-ban, onnan kiolvasható, a Load tehát késlekedés nélkül befejezhető.

3. Ha a ROB-ban a korábbi utasítások között egy olyan befejezetlen Store-t talál, amely ugyan befejezetlen, de a cím már ki van számítva és a ROB-ban rendelkezésre áll, és ez a cím pont megegyezik a Load címével, akkor a Load végrehajtásával várni kell. Addig kell várni, amíg a Store ROB bejegyzésében meg nem jelenik a tárolandó adat, ezt követően a Load ezt az értéket közvetlenül onnan átveheti.

4. Ha a ROB-ban a korábbi utasítások között egy olyan befejezetlen Store-t talál, amelynek még a címe sincs kiszámolva, akkor van igazán gond. Mit csináljunk a Load-dal? Biztos ami biztos alapon várjuk meg vele a Store-t, hátha pont ugyanarra a címre akar írni? De akkor ezzel az összes Load-tól függő utasítást is feltartjuk, többnyire feleslegesen. Vagy legyünk bátrak, és hajtsuk végre a Load-ot rögtön, a Store előtt, mondván hogy olyan nagy az a memória, kicsi az esély rá, hogy két memóriaművelet pont ugyanarra a címre vonatkozik? De mit tegyünk, ha mégis ugyanaz az írási és az olvasási cím, tehát valódi RAW egymásrahatás alakul ki? Sajnos ez a lokalitási elvek miatt nem is olyan ritka esemény. Ezekkel a kérdésekkel foglalkozik a *memória egyértelműsítés* (memory disambiguation) folyamata.

Memória egyértelműsítés

A memória egyértelműsítés (memory disambiguation) tehát annak az eldöntése, hogy egy Load és egy korábbi, még be nem fejezett (és címszámítás előtt álló) Store között van-e RAW egymásrahatás, vagy nincs. Ez a processzor teljesítménye szempontjából egy lényeges kérdés, hiszen RAW egymásrahatás esetén a Load-ot várakoztatni kell, ellenkező esetben végrehajtható (ha nincs más egymásrahatás ami a végrehajtását akadályozná). Ráadásul ez a döntési helyzet a tipikus működés során elég gyakran fennáll: a statisztikák szerint a memóriaműveletek teszik ki a programok utasításainak közel harmadát.

Lehetőségek memória egyértelműsítésre:

- *Konzervatív stratégia*: minden Load köteles megvárni, hogy a ROB-ban előtte lévő Store-ok befejeződjenek, hátha épp ugyanazt a címet fogják írni, amit a Load olvasni akar. Ez biztos megoldás, de teljesítmény szempontjából nem éppen optimális.

- *Optimista stratégia*: ne vegyünk tudomást a potenciális RAW egymásrahatásról, a Load bátran megelőzheti a Store-okat a sorrenden kívüli végrehajtás jegyében. Ez persze gyorsabb feldolgozást eredményez, de ha rossz volt a döntés, vagyis ha az írási és olvasási cím mégis megegyezik, akkor vissza kell pörgetni az eseményeket: a Load és az utána következő utasítások eredményeit érvényteleníteni kell a ROB-ban, majd ezeket újra végre kell hajtani.

- *Spekulatív stratégia*: optimistán kezd, de ha a programban ugyanarra a Load utasításra az optimista spekuláció egyszer (vagy - implementációtól függően - többször) nem vált be, akkor azt a processzor egy olyan Load-nak minősíti, amely szeret a korábbi Store-ok címéről olvasni, utasításszámlálóját egy erre a célra fenntartott bufferben elhelyezi, és ha legközelebb belefut, akkor vele a konzervatív stratégia szerint jár el.

14.3. Soron kívüli végrehajtás a gyakorlatban

14.3.1. A Scoreboard algoritmjus

A CDC6600-as számítógép volt az első, amely sorrenden kívüli utasítás-végrehajtást valósított meg a ma már kezdetlegesnek számító ”scoreboard” (irányítótábla) eljárással. A scoreboard eljárás az out-of-order végrehajtást centralizált irányítás segítségével oldja meg, regiszterátnevezést, sorrend-visszaállító buffert nem alkalmaz. Az alapja egy irányítótábla, ami nyilvántartja, hogy mely utasítások végrehajtása kezdődött el, melyik utasítás épp melyik fázisban tart, és hogy alakul a műveleti egységek foglaltsága és a regiszterek tartalma. Ezen információk alapján pontosan tudható, hogy mikor és milyen, kezelést igénylő egymásrahatások léphetnek fel utasítások között.

Az eredeti scoreboard eljárás 5 fázisra bontja az utasítások végrehajtását:

1. Utasításlehívás (instruction fetch, **IF**): Szerepe és működése megegyezik a korábban látottakkal
2. Hozzárendelés (dispatch, **DS**): Ebben a lépésben történik az utasítás dekódolása, és a megfelelő műveleti egységhez rendelése. Ha nincs az utasítás típusának megfelelő szabad műveleti egység, vagy más egyéb feldolgozási egymásrahatás történik, akkor az utasítás várakozni kényszerül (de az ezen a ponton blokkolt utasítást más nem előzheti be, tehát az új utasítások lehívása is szünetelni fog). Továbbá, ha WAW függőség áll fent valamely végrehajtás alatt álló utasítással, akkor szintén kényszerű várakozás következik (hiszen félő, hogy ha elindítjuk a végrehajtását, az új utasítás hamarabb végez, mint a régi, ezzel megsértve az eredmény regiszterek program által előírt írási sorrendjét). Ha ezek egyike sem áll fent, akkor az utasítást felvisszük az irányítótáblára, a további életének nyomon követésére.
3. Végrehajtásra kijelölés (issue, **IS**): Ebben a fázisban kell kiolvasni forrásoperandusokat a regisztertárolóból, ezzel az utasítás készen áll a végrehajtásra. Nem léphet az utasítás ebbe a fázisba, ha RAW egymásrahatás áll fenn valamely, már végrehajtás alatt álló utasítással, mivel ekkor a regisztereket még nem szabad kiolvasni (mert a forrásoperandust előállító utasítás még nem írta be a regiszter friss értékét a regisztertárolóba), ilyenkor az utasítás várakozni kényszerül.
4. Végrehajtás (execute, **EX**): Az utasítás által előírt művelet végrehajtása ebben a fázisban történik. Attól függően, hogy milyen műveletet kell elvégezni, ez egy vagy több órajelciklust is igénybe vehet.
5. Regiszterírás (writeback, **WB**): Ebben a lépésben kerül be a művelet eredménye a regisztertárolóba. Az utasítás a beírással várakozni kényszerül, ha WAR egymásrahatás áll fenn egy másik utasítással. Azaz, ha az utasítás a sorrenden kívüli végrehajtás során megelőzött egy korábbit, de a korábbi utasítás egyik forrás operandusa pont megegyezik a WB-re váró eredmény regiszterével. Ilyenkor nem írhatjuk felül az eredményregiszter értékét, mert még vannak, akik a régi értékre kíváncsiak, de még nem olvasták el.

A tanult egyszerű 5 fokozatú pipeline-hoz képest annyi a különbség, hogy az ID fázis ketté bomlik (DS és IS fázisokra), valamint nincs megkülönböztetett MEM fázis, a memóriaműveletek végrehajtása az EX fázisban történik.

A fentiek szerint scoreboard eljárásban tehát az egymásrahatások kezelése minden esetben szünetek beiktatásával, a problémás utasítás várakoztatásával történik, az alábbiak szerint:

- *Feldolgozási egymásrahatás* esetén az utasítást nem engedjük belépni a DS fázisba,
- *WAW egymásrahatás* esetén az utasítást nem engedjük belépni a DS fázisba,
- *RAW egymásrahatás* esetén az utasítást nem engedjük belépni a IS fázisba,
- *WAR egymásrahatás* esetén az utasítást nem engedjük belépni a WB fázisba.

Az algoritmus legfontosabb szereplője, az irányítótábla, három részből áll:

1. Az **utasítás állapot tábla**: ez a táblázat tartja nyilván, hogy a lehívott, de még be nem fejezett utasítások végrehajtása mely fázisban tart.
2. **Regiszter állapot tábla**: minden regiszterhez nyilvántartja, hogy abba mely műveleti egység eredménye fog kerülni. Ha üres egy bejegyzés, akkor nincs olyan folyamatban lévő utasítás, melynek az lenne az eredményregisztere.

3. **Műveleti egység tábla**: minden műveleti egységre az alábbi információkat tároljuk:
   - ”Foglalt” bit
   - Művelet kódja (összeadás, kivonás, stb.)
   - R: a művelet eredményét mely regiszterbe kell írni
   - RA, RB: a két forrás operandus értéke (üres(ek), ha még nem áll(nak) rendelkezésre)
   - EA, EB: ha valamelyik forrásoperandus nem áll rendelkezésre (vagy épp egyik sem), ezekben a bejegyzésben szerepel, hogy azok mely műveleti egység eredményeként fognak előállni

A 14.11., 14.12. és a 14.13. ábrákon látható példa egy 7 utasításból álló program végrehajtását demonstrálja a scoreboard eljárás segítségével. Ebben a példában 5 műveleti egység van: egy egész számokat kezelő aritmetikai egység, egy Load egység, egy Store egység, és két lebegőpontos egység. A lebegőpontos egységek belül pipeline szervezésűek (bár ennek ebben a példában nincs szerepe), késleltetésük 3 ciklusidő.

A példában végigkövethető az algoritmus működése. Mielőtt az utasítás a DS fázisba lépne, két feltételt kell ellenőrizni:

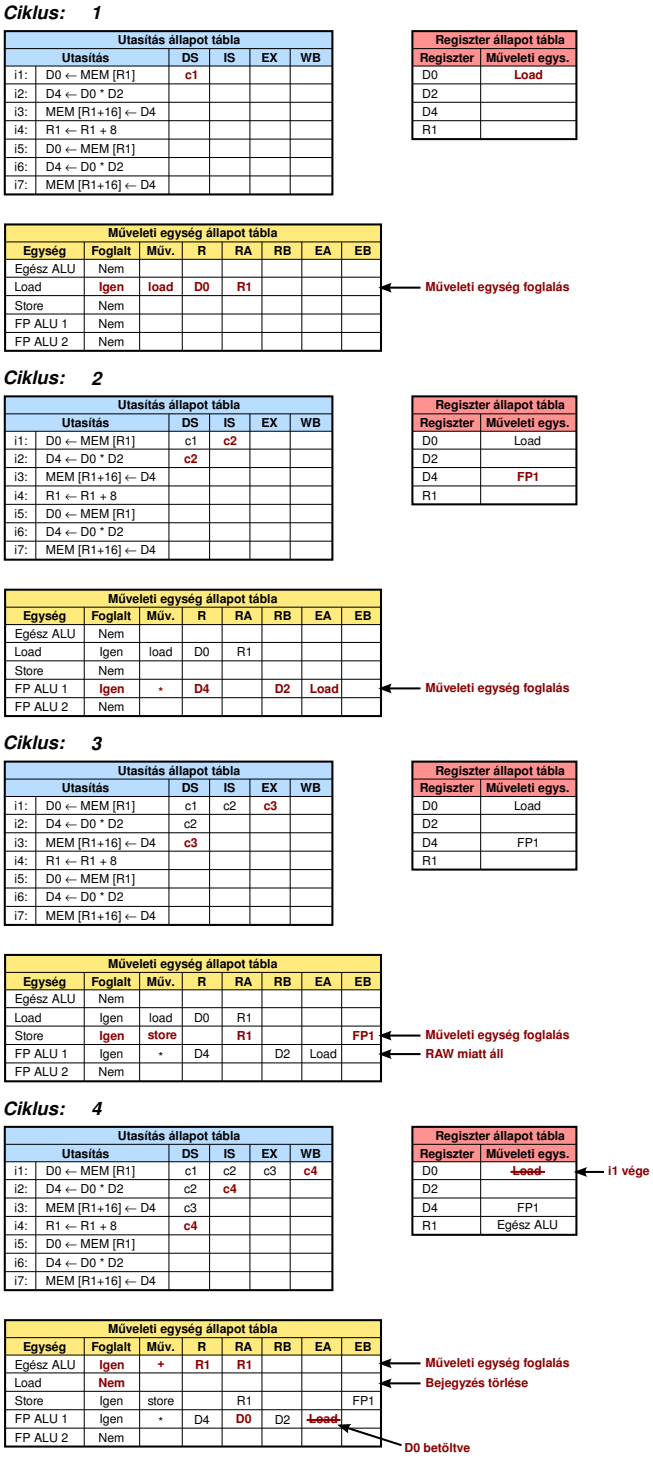
- Van-e szabad műveleti egység az utasítás számára. Ez a műveleti egység állapot táblából megállapítható. Ha nincs, akkor feldolgozási egymásrahatás történt, az utasítást nem engedjük a DS fázisba lépni.
- Van-e végrehajtás alatt olyan utasítás, amelyik épp ugyanabba a regiszterbe ír, mint a vizsgált utasítás. Ez a regiszter állapot táblából kiderül: ha az eredményregiszternek megfelelő sor nem üres, akkor az egyik futó utasítás oda fog írni. Ez WAW egymásrahatást jelent, az utasítást nem engedjük a DS fázisba lépni.

Ha sem feldolgozási, sem WAW egymásrahatás nincs, akkor a lehívott utasítás a DS fázisba léphet. A *DS fázisba lépéskor* a következő adminisztratív feladatokat kell elvégezni:

- Bejegyezzük a regiszter állapot táblába, az utasítás eredményregiszterének megfelelő sorba annak a műveleti egységnek az azonosítóját, amelyik az utasítást végre fogja hajtani. Vegyük észre, hogy a regiszter állapot táblában ez a sor mindenképp szabad, ha nem lenne szabad, az utasítás nem léphetett volna be a DS fázisba.
- Ezzel egyidejűleg az utasítást hozzárendeljük egy műveleti egységhez, a műveleti egység állapot tábla egy sorának kitöltésével. Az utasítás végrehajtásához szükséges műveleti egységnek megfelelő sorba a ”foglalt” bitet 1-be billentjük, és az utasításnak megfelelő műveletet bejegyezzük. A forrásoperandusokra vonatkozó bejegyzések (RA, RB) kitöltésénél két lehetőség van: ha a regiszter ”szabad”, tehát nincs olyan végrehajtás alatt álló utasítás, ami írni akarná, akkor beírjuk a regiszter azonosítóját. Azt, hogy egy regiszter ”szabad”, onnan vesszük észre, hogy a regiszter állapot tábla vonatkozó bejegyzése üres. Ha a regiszter nem szabad, akkor annak értéke egyelőre nem áll rendelkezésre, ilyenkor az EA vagy EB bejegyzésbe (attól függően, hogy az egyik, vagy a másik forrásoperandusról van-e szó) beírjuk annak a műveleti egységnek az azonosítóját, amelyikre várunk, amelyik az operandust elő fogja állítani. Ennek a kérdéses, megvárandó műveleti egységnek az azonosítóját szintén a regiszter állapot táblából lehet kiolvasni. Ebben az esetben (tehát ha legalább az egyik forrás operandus regiszter nem ”szabad”), RAW függőség alakult ki: az utasítás bennragad a DS fázisban, meg kell várnia, míg a forrásoperandusai előállnak.

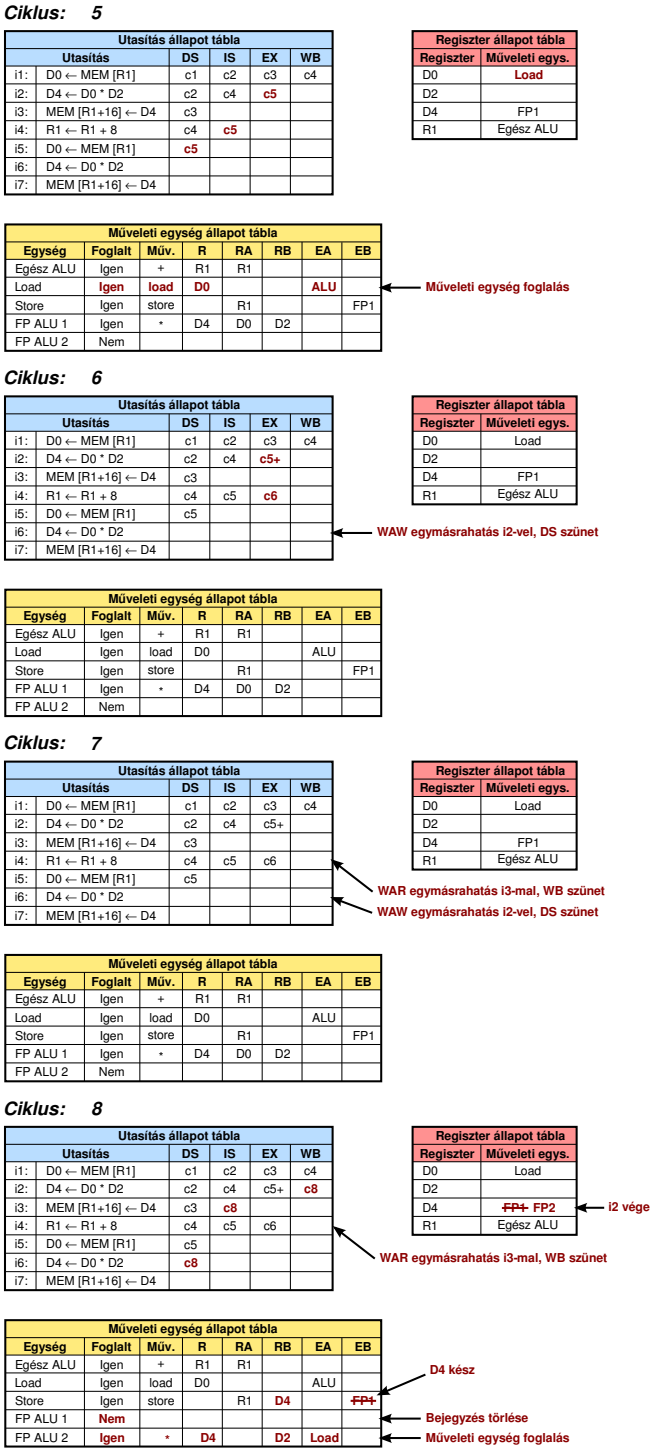
Egy utasítás csak akkor tud az IS fázisba lépni, ha a végrehajtásához szükséges minden operandus rendelkezésre áll. Ebben a fázisba történik a forrásoperandus-regiszterek kiolvasása, majd ezután az EX fázisban a végrehajtása. Ha egy műveleti egység végez a feladatával (ezzel befejezve a hozzá tartozó utasítás EX fázisát), akkor erről értesítést küld a scoreboard-nak. Az eredmények beírása a regisztertárolóba (tehát a WB fázisba lépés) csak akkor történhet meg, ha nincs WAR egymásrahatás. Ezt a scoreboard ellenőrizni tudja: körülnéz a műveleti egység és az utasítás állapot táblában, hogy van-e olyan korábban érkezett utasítás a DS vagy IS fázisban, amelynek a forrásoperandusa pont az a regiszter, amit írni szeretnénk. Ezek a korábbi utasítások még nem olvasták ki a forrásoperandus-regisztereiket, amíg ezt meg nem teszik, a későbbi utasítások nem írhatják felül ezeket, vagyis addig nem léphetnek a WB fázisba. A WB fázis egyben az utasítás végrehajtásának befejezését jelenti. Ekkor, az eredmény regisztertárolóba írásán kívül a táblázatokat is frissíteni kell:

- A regiszter állapot táblában az eredményregiszterhez tartozó bejegyzést törölni kell
- Azok az utasítások, akik RAW egymásrahatás miatt erre az eredményre vártak (tehát a műveleti egység táblában az EA vagy EB mezőben a most végzett műveleti egység azonosítója szerepelt), most folytathatók lesznek (EA vagy EB mezőjüket törölni kell, az RA és RB mezőjüket pedig kitölteni)
- A műveleti egység táblában a használt műveleti egység ”foglalt” bitjét 0-ba állítjuk



14.11. ábra. Példa a scoreboard eljárás működésére, 1/3. rész

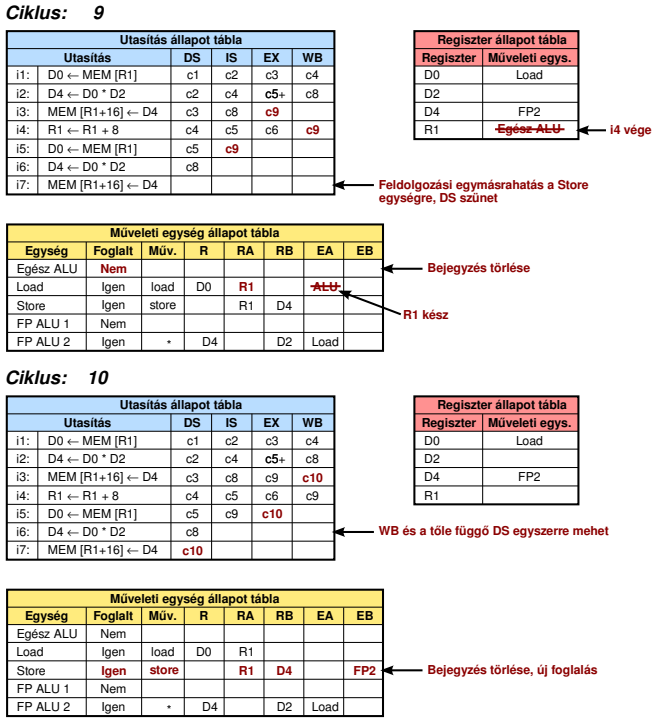
A példában minden elképzelhető egymásrahatás szerepel: feldolgozási, RAW, WAW, WAR egyaránt. Az utolsó ciklushoz pörgetve látható, hogy sorrenden kívüli végrehajtás is történt, az i4 utasítás megelőzte az i3-t, nem csak a befejeződését, hanem a végrehajtás elkezdését tekintve is. Kevésbé ”állatorvosi ló” jellegű példában több sorrenden kívül végrehajtott utasítást láthatnánk.



14.12. ábra. Példa a scoreboard eljárás működésére, 2/3. rész

14.3.2. A Tomasulo algoritmus

Az IBM 1967-ben, a CDC6600 után 3 évvel, arra mintegy válaszlépésként jelentette be az IBM 360/91-es processzorát. A processzor a konkurencia ”scoreboard” eljárásához képest egy egészen új, annál sokkal hatékonyabb algoritmust használt az utasítások sorrenden kívüli végrehajtására. Ez az algoritmus Tomasulo algoritmus néven vált ismerté, és idővel rendkívül sikeres lett: szinte minden modern out-of-order processzor a Tomasulo algoritmus ötletén



14.13. ábra. Példa a scoreboard eljárás működésére, 3/3. rész

alapul. A különböző gyártók persze itt-ott továbbfejlesztették, néhány ponton kicsit módosították, de a működési elv 1967 óta ugyanaz.

A Tomasulo algoritmus már használja a regiszterátnevezést (ott jelent meg először a gyakorlatban), így a WAR és a WAW függőségek nem állítják meg az utasításfeldolgozást, ellenben a scoreboard algoritmussal. A Tomasulo algoritmus eredeti, publikált változatában nem volt sorrend-visszaállító buffer, de a ROB-bal való kiegészítésre később több megoldás is született. Ebben a fejezetben az algoritmus Intel P6 architektúrában (Pentium Pro - Intel Core i7) alkalmazott változatát mutatjuk be, amely szintén tartalmaz ROB-ot.

Az alkalmazott adatszerkezetek:

1. **Sorrend-visszaállító buffer (ROB):** A ROB bejegyzéseket egy táblázatban tároljuk, melyben két mutatót tartunk karban, az egyik az első, a másik az utolsó bejegyzésre mutat. Amikor egy új bejegyzés kerül a ROB-ba, az utolsóra mutató mutatót megnöveljük, és a mutatott helyre kerül az új utasítás bejegyzése. A ROB bejegyzések feldolgozása, érvényre juttatása a sor elejéről történik (amikor a kapcsolódó utasítás az RT, kilépési fázisba került), ekkor egyel megnöveljük az elsőre elemre mutató mutatót. Ezzel a technikával tulajdonképpen egy cirkuláris buffer segítségével valósítjuk meg a ROB FIFO adatszerkezetét. Egy bejegyzéshez tartozó mezők:
   - V: az utasítás eredménye (egy érték)
   - R: az eredményregiszter neve, ebbe a regiszterbe kell írni az utasítás értékét (hacsak nem Store-ról van szó)
   - Addr: erre a memóriacímre kell írni az utasítás értékét (ha Store-ról van szó)
   - IS/EX/CM: jelöli, hogy az utasítás feldolgozása melyik fázisban tart. Nincs szerepe az algoritmus működésében, csak illusztrációs célokat szolgál
   - ”Ready” bit: =1, ha a bejegyzéshez tartozó utasítás végrehajtása befejeződött
2. **Utasítástároló (RS):** Az utasítások a DS fázisban az utasítástárolóban várnak. Centralizált RS-t használunk, a következő mezőkkel:
   - Egység: az utasítás által igényelt műveleti egység neve
   - Művelet kódja (összeadás, kivonás, stb.)
   - T (Target): az utasítás ROB bejegyzésének a száma, oda kell majd végrehajtás után írni az eredményt

- RA, RB: a két forrás operandus értéke (üres(ek), ha még nem áll(nak) rendelkezésre)
- EA, EB: ha valamelyik forrásoperandus nem áll rendelkezésre (vagy épp egyik sem), ezekben a bejegyzésekben szerepel, hogy azok mely ROB bejegyzésben fognak megjelenni

3. **Regiszter állapot tábla:** Minden logikai regiszterhez tartozik egy bejegyzés. Ha épp nincs olyan utasítás, ami írni akarná, akkor itt szerepel a regiszter értéke, ellenkező esetben annak a ROB bejegyzésnek a száma, melyben a regiszter értéke megtalálható, vagy a jövőben megtalálható lesz.

4. **CDB:** Common Data Bus, egy üzenetszórásos csatorna. Ha egy utasítás befejeződött, ezen a csatornán kürtöli világgá a hírt, a ROB bejegyzésének a számával (T) és az utasítás eredményével (V) együtt. Mivel ezt minden várakozó utasítás hallja, amelyik épp erre az eredményre várt, el tudja tárolni a kihirdetett értéket.

A 14.14., 14.15. és a 14.16. ábrán látható példa egy 7 utasításból álló program végrehajtását demonstrálja a Tomasulo algoritmus P6-os változata szerint. Ebben a példában 5 műveleti egység van: egy egész számokat kezelő aritmetikai egység, egy Load egység, egy Store egység, és két lebegőpontos egység. A lebegőpontos egységek belül pipeline szervezésűek (bár ennek ebben a példában nincs szerepe), késleltetésük 3 ciklusidő. Az utasítástároló centralizált, 3 bejegyzéses. Az egyszerűség kedvéért az ábrán szereplő ROB-ban nem tüntetjük fel a ”Ready” bitet, azt, hogy az utasítás végrehajtása kész, onnan fogjuk látni, hogy az utasítás a CM fázisban van.

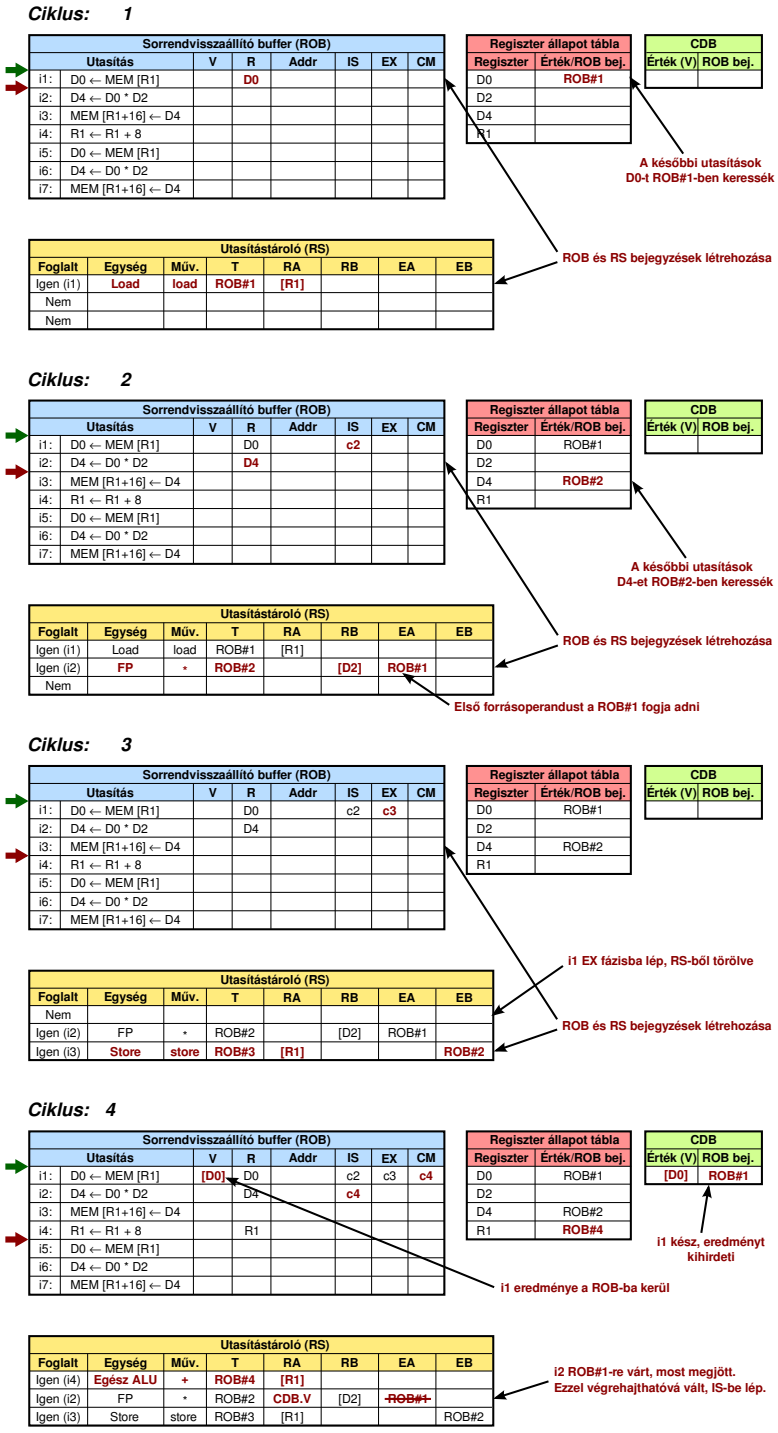
Kövessük végig egy utasítás útját a végrehajtás során. Az utasítás lehívása után (IF fázis) a ”dispatch” (DS) fázis feladata, hogy az utasítás számára bejegyzést foglaljon a ROB-ba és az RS-be. A ROB-ba szigorúan az utolsó érvényes sor után (az ábrán piros nyíllal jelölve) kell bejegyzést allokálni az új utasításnak, míg az utasítástárolóban egy tetszőleges szabad bejegyzést el lehet foglalni. Ha vagy a ROB-ban, vagy az RS-ben nincs több hely, akkor az utasítás várakozni kényszerül (ez feldolgozási egymásrahatást jelent). A feldolgozási egymásrahatás feloldásáig a további utasítások lehívása is szünetel. A ROB bejegyzés kitöltendő mezői:

- A ”V” mezőt, amely az utasítás eredményét tárolja, ilyenkor még nem kell kitölteni.
- Ha aritmetikai vagy Load utasításról van szó, az eredmény regiszter logikai nevét kell az ”R” mezőbe írni. Végül ide fog majd a ”V”-ben tárolt eredmény értéke kerülni. Ha Store utasításról van szó, és a vonatkozó memóriacím ezen a ponton már ismert, akkor az az ”Addr” mezőbe kerül. Ha címszámításra van szükség, akkor az ”Addr” a Store művelet végrehajtása során, a cím kiszámítása után kerül a kitöltésre.
- Az IS/EX/CM egyike sincs megjelölve, hiszen az utasítás egyelőre a DS fázisban tart.

Az utasítástároló (RS) bejegyzésének kitöltése:

- A ”Foglalt” bit 1-be állítódik, jelezve, hogy ez a bejegyzés már használatban van.
- Az ”Egység” mezőbe kerül az utasítás által igényelt műveleti egység neve (Load/Store/Egész ALU/FP ALU).
- A ”Művelet” mező az utasítás alapján kitölthető.
- A ”T” mezőbe kell írni az utasításhoz tartozó ROB bejegyzés számát. Ide kerül az eredmény, ha majd az utasítás végrehajtása befejeződik.
- Az utasítás megpróbálja összeszedni a forrás operandusait. Ránéz a regiszter állapot táblára. Ha a forrás regiszternek megfelelő helyen a regiszter értékét találja, bemásolja magának az RA ill. RB mező(k)be. Ha a regiszter állapot táblában egy ROB bejegyzés számot talál, az azt jelenti, hogy a regiszter nem ”szabad”, mert van olyan utasítás, melynek ez lesz az eredmény regisztere. Ez gyakorlatilag egy RAW egymásrahatást jelent. Ilyenkor az utasítástároló EA ill. EB mezőjébe azt a ROB bejegyzés számot kell beírni, ahol majd a regiszter értékét szolgáltató utasítás eredménye meg fog jelenni.

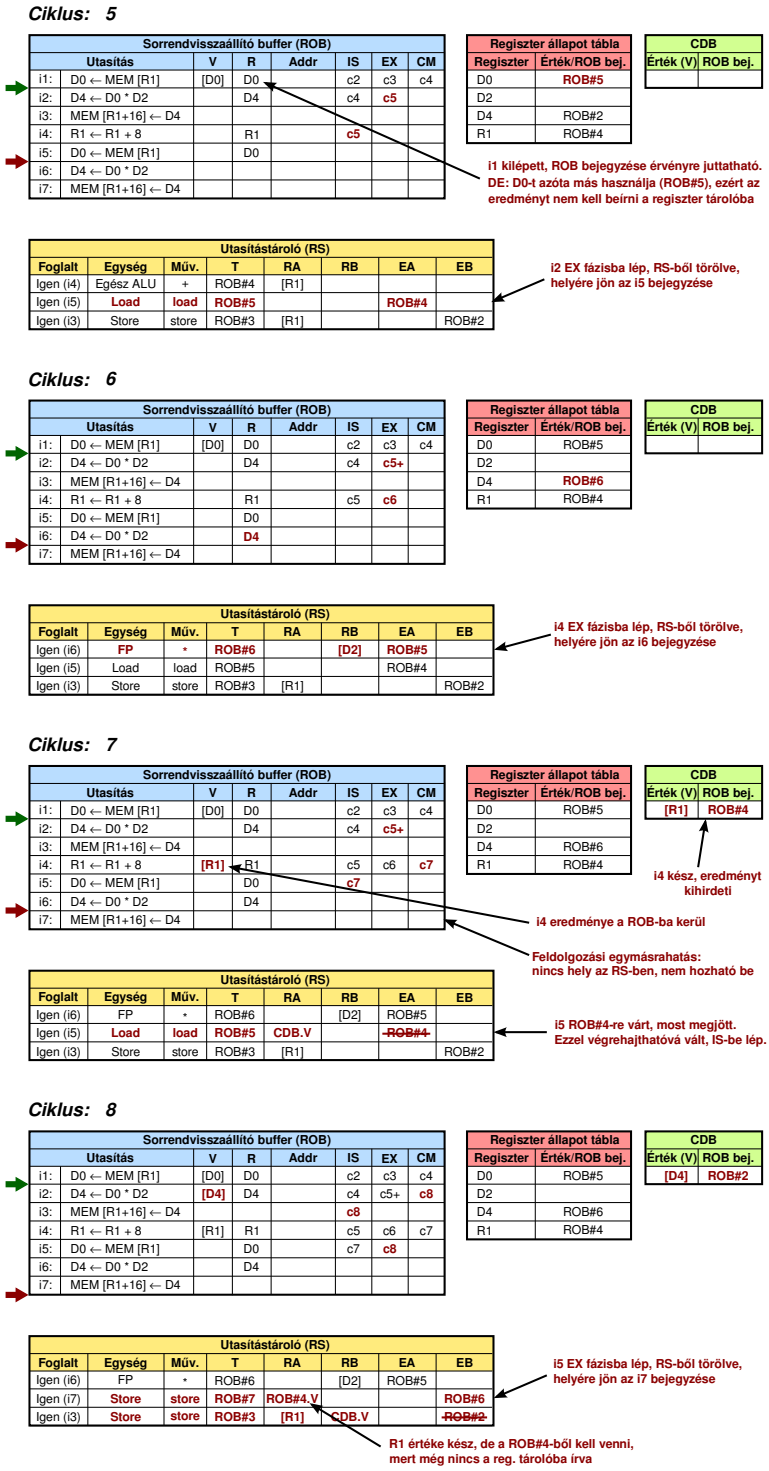
Egy utasítás akkor mehet tovább az ”issue” (IS) fázisba, ha a végrehajtásához szükséges forrásoperandusok értéke rendelkezésre áll, tehát ha az utasítástárolóban az EA és EB mezője is üres, és ha van a végrehajtásához szükséges szabad műveleti egység. Az IS fázis után megkezdődhet az utasítás végrehajtása (EX), amely az utasítástól függően több ciklusig is eltarthat. Az IS során az utasítás a feldolgozó egységbe kerül, így az EX kezdetére a hozzá tartozó RS bejegyzés felszabadul. A végrehajtás befejeződése után az eredmény a ”complete” (CM) fázisban kerül meghirdetésre a CDB nevű üzenetszórásos csatornán keresztül. A meghirdetett üzenet tartalmazza a befejeződött utasítás ROB számát, és az eredmény értékét. Az üzenetszórásnak köszönhetően ez az eredmény az RS-ben várakozó minden utasításhoz eljut. Amelyik utasítás pont erre vár (tehát akinek a forrás operandusa pont a befejeződött utasítás eredmény regisztere), az a buszból kiolvassa az értéket, és ezáltal a RAW egymásrahatás megszűnik.



14.14. ábra. Példa a P6-ban megvalósított Tomasulo algoritmus működésére, 1/3.rész

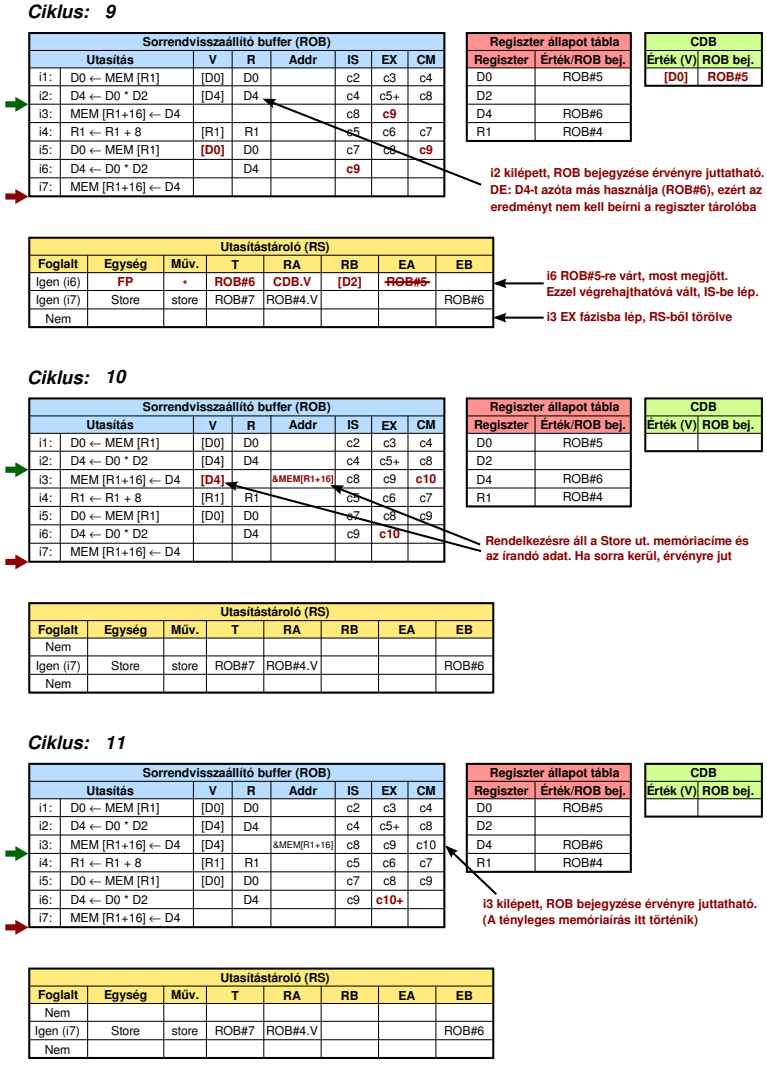
Az utasítás végrehajtás ”retire” (RT) fázisában történik meg az utasítás ROB bejegyzésének érvényre juttatása, vagyis a tárolt eredmény regiszter tárolóba vagy a memóriába írása. Egy utasítás csak akkor léphet RT fázisba, ha ő van a ROB ”tetején”, vagyis rá mutat az első érvényes ROB bejegyzésre mutató mutató. Ezt a mutatót az RT fázis után léptetjük.

Ebben a példában is látunk RAW és feldolgozási egymásrahatást, de a WAR és a WAW az algoritmus természetéből adódóan nem okozott gondot. Láttuk, hogy a WAR és a WAW egymásrahatást regiszter átnevezéssel



14.15. ábra. Példa a P6-ban megvalósított Tomasulo algoritmus működésére, 2/3. rész

lehet elkerülni. Hol történt a példában a regiszter átnevezés? A Tomasulo algoritmusban a regiszter átnevezés bújtatott formában van jelen: a fizikai regiszterek tulajdonképpen a ROB bejegyzésnek felelnek meg (melynek "V" mezője hordozza a regiszter értékét). Minden pont úgy történt, ahogy a regiszter átnevezés bevezetésénél láttuk: az újonnan érkező utasítások eredménye számára új fizikai regisztert foglalunk (ez lesz az ő ROB bejegyzése), a regiszter leképző tábla szerepét pedig a regiszter állapot tábla játssza.



14.16. ábra. Példa a P6-ban megvalósított Tomasulo algoritmus működésére, 3/3. rész

15. fejezet

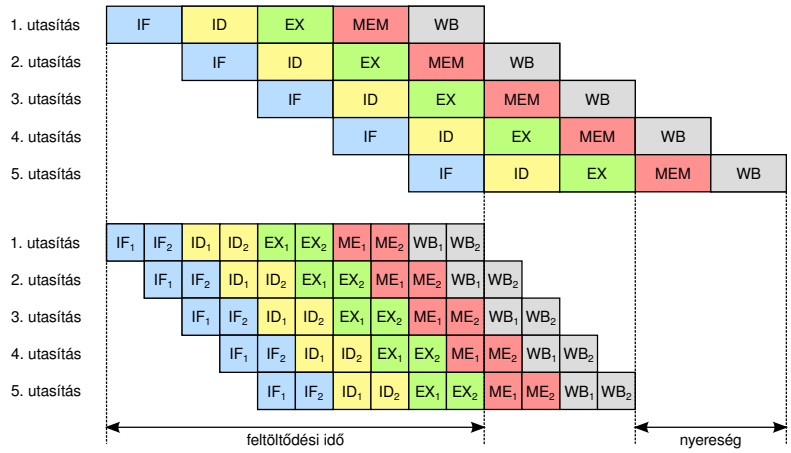
Széles utasítás pipeline-ok

15.1. A pipeline utasításfeldolgozás átviteli sebességének növelése

Az utasításfeldolgozás átviteli sebességét, vagyis az időegységenként végrehajtott utasítások számát kétféleképpen növelhetjük: vagy a pipeline *mélységét*, vagy a *szélességét* növeljük.

A pipeline mélysége úgy növelhető, hogy az utasításfeldolgozás minden fázisát további részfázisokra bontjuk. Mivel a részfázisok késleltetése kisebb lesz, mint az eredeti fázisok késleltetése volt, a ciklusidő (mely a leglassabb fázis késleltetése) kisebbre vehető (vagy, ami ugyanezt jelenti, az órajel frekvencia növelhető), ami időegységenként több utasítás feldolgozását teszi lehetővé.

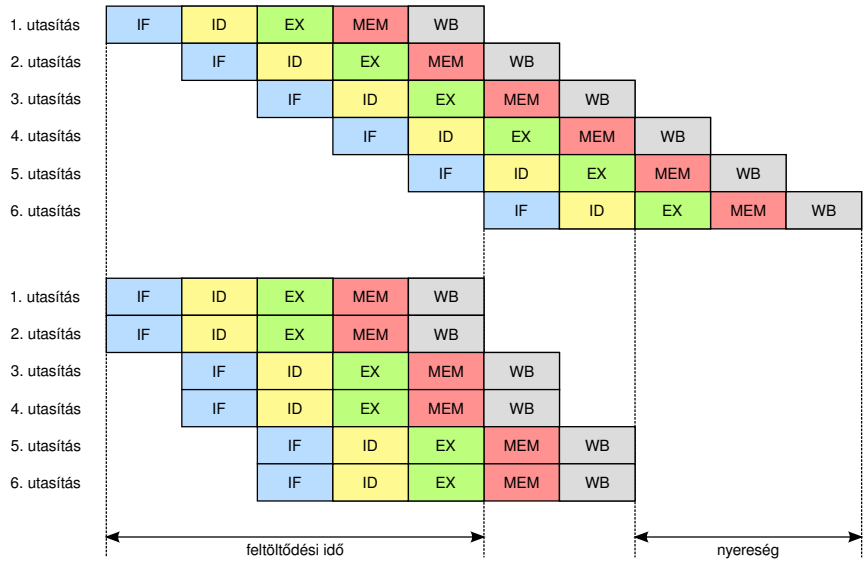
Vegyük alapul a már tanult 5 fokozatú pipeline-t, és vágjunk szét minden fázist két részfázisra. Feltéve, hogy ez a szétvágás kiegyensúlyozott volt, vagyis a keletkezett két részfázis késleltetése közel feleakkora, mint az eredeti fázisé volt, a ciklusidőt felére csökkenthetjük, ezzel (ideális körülmények között) megduplázva az időegységenként feldolgozható utasítások számát (lásd 15.1. ábra).



15.1. ábra. Átviteli sebesség növelése mélyebb pipeline-nal

A pipeline teljesítménye a futószalag szélesítésével is növelhető. Ez azt jelenti, hogy egyszerre több utasítás feldolgozása is megkezdődhet egyidejűleg, melyek együtt haladnak végig a feldolgozás fázisain, mintha több pipeline lenne a processzorban. Ehhez az kell, hogy az IF fázis egyszerre több utasítást is le tudjon hívni, az ID egy ciklusidő alatt több utasítást is képes legyen dekódolnia, stb., és ezzel együtt természetesen a pipeline regiszterek számát is többszörözni kell. Ha minden ciklusban m új utasítás végrehajtása kezdődik meg, m széles pipeline-ról és m-utas végrehajtásról beszélünk. Szélesebb pipeline-al a ciklusidő rövidítése nélkül (tehát azonos órajel frekvencián) is növelhető az átvitel (lásd 15.2. ábra).

Intuitíve úgy lehet elképzelni a két lehetőséget, hogy az egyik esetben dupla sebességűre kapcsoljuk a futószalagot, a feldolgozó egységek számát pedig megkétszerezzük és feladatuk mennyiségét felére vesszük. A másik esetben pedig minden feldolgozó egységet duplikálunk és a széles futószalagra egymás ”mögé” egyszerre



15.2. ábra. Átviteli sebesség növelése 2-utas szuperskalár pipeline-nal

két munkadarabot helyezünk.

Elméletileg, ha k-szorosára növeljük a pipeline mélységét és m-szeresére a szélességét, akkor $m \cdot k$-szoros teljesítményjavulást kapunk. Ez egy elméleti érték, amit számos viselkedésbeli, tervezési ill. technológiai tényező korlátoz:

- nagyon mély és/vagy széles pipeline-ok esetén a sok átlapoltan/párhuzamosan futó utasítás között az egymásrahatások előfordulásának gyakorisága az átlapoltan futó utasítások számával meredeken nő (elég valószínűtlen, hogy az utasítássorozat csupa egymástól független műveletből áll),
- nagyon széles pipeline-hoz sok műveleti egységre van szükség, melyek között a RAW egymásrahatások kezelésére szolgáló forwarding utak száma m-mel négyzetesen nő,
- hiába mély a pipeline, a ciklusidő nem lehet tetszőlegesen rövid, részben fogyasztási okokból, részben pedig azért, mert a pipeline regiszterek írása/kiolvasása fizikai okokból nem lehet tetszőlegesen kicsi, és az bele kell, hogy férjen egy ciklusidőbe,
- a mély pipeline-nak megvan az a hátránya, hogy a spekulatív döntések (pl. elágazásbecslés) kimenetele sok fázissal később derül ki, eközben sok, potenciálisan felesleges utasítás végrehajtása kezdődhet el.

15.2. Ütemezés többutas pipeline-ban

Ahhoz, hogy az utasítás pipeline szélesítése ténylegesen megtérüljön, vagyis a lehető legtöbb utasítás végrehajtása tudjon minden ciklusban elkezdődni, fontos, hogy az utasításfolyamban mindig legyen megfelelő számú független, végrehajtásra kész utasítás. A független utasítások összeválogatására több megoldás is létezik: végezheti futási időben maga a processzor, vagy fordítási időben a fordítóprogram is, előbbi esetben dinamikus, utóbbiban statikus ütemezésről beszélünk.

A dinamikus ütemezés valós időben, dinamikusan változó környezetben történik (hiszen nem lehet előre tudni, hogy melyik utasítás mennyi ideig fog futni, mert az függ pl. attól, hogy történt-e cache-hiba, laphiba, egymásrahatás, stb.). A processzor komoly erőfeszítéseket tesz annak érdekében, hogy a látóterében lévő utasítások közül kiválogassa azokat, amelyek végrehajtása egymással párhuzamosan elkezdhető, és ezzel a műveleti egységek kihasználtságát a lehető legmagasabban tartsa.

Statikus ütemezés esetén a fordítóprogram sok terhet levesz a processzor ütemezőjének válláról. Hiszen azt, hogy a program utasításai közül melyek azok, amik párhuzamosan végrehajthatók, a fordítóprogram is eldöntheti. Ráadásul sokkal hatékonyabban, hiszen egyrészt a fordítási idő nem kritikus, tehát fordítás közben bőven van idő jobb heurisztikákat alkalmazni az utasítások átsorrendezésére és független utasításokból álló csoportok képzésére, másrészt a fordító az utasítások sokkal nagyobb környezetével tud dolgozni. A processzor csak a már letöltött

	Párhuzamosan végrehajtható csoportok kiválasztása	Hozzárendelés a műveleti egységekhez	Végrehajtási idő meghatározása
Szuperskalár	Hardver	Hardver	Hardver
EPIC	Fordító	Hardver	Hardver
Dinamikus VLIW	Fordító	Fordító	Hardver
VLIW	Fordító	Fordító	Fordító

15.3. ábra. A hardver és a fordítóprogram szerepe a széles pipeline-t alkalmazó architektúrákban ([39])

utasításokat tudja rendezgetni, de a fordító játéktere sokkal nagyobb, mivel egyszerre sokkal több utasításra van rálátása.

Természetesen a tisztán statikus és tisztán dinamikus ütemezés mellett vannak középutak, többféle esetet különböztethetünk meg annak megfelelően, hogy az utasítások ütemezéséből mekkora részt vállal a processzor, illetve a fordító. A 15.3. ábra négy, a további fejezetekben részletesen tárgyalt, széles pipeline-t használó architektúrát sorol fel. A leginkább dinamikus architektúrát a szuperskalár processzorok képviselik. Szuperskalár esetben sem a fordító, sem a programozó nincs tudatában annak, hogy a processzor több utasítás párhuzamos elkezdésére képes. A hardver kénytelen felderíteni a párhuzamosan végrehajtható utasításokat, azt, hogy melyiket melyik műveleti egység hajtsa végre, és azt is, hogy melyik utasítás végrehajtása mikor kezdhető meg (tehát az egymásrahatások követése és kezelése is a hardver feladata). A másik véglet a VLIW (Very Long Instruction Word) architektúra. A VLIW architektúrában mindent a fordító végez, beleértve az utasítások műveleti egységekhez rendelését és az egymásrahatások feloldását is. A középutat az EPIC (Explicitly Parallel Instruction Computing) architektúra jelenti, mely szerint a fordító feladata az utasítások párhuzamosan végrehajtható csoportokba rendezése, minden más a hardveren múlik.

15.3. Szuperskalár processzorok

A szuperskalár processzorok olyan széles pipeline-al rendelkező processzorok, melyek képesek egyidejűleg több utasítás végrehajtását elkezdeni, és saját maguk, hardver szinten képesen az egyidejű végrehajtásra alkalmas utasítások kijelölésére. Ha a processzor egyidejűleg m utasítás végrehajtását képes elkezdeni, akkor m-utas szuperskalár architektúráról beszélünk.

Két megoldás létezik:

- Sorrendi (in-order) szuperskalár processzorokban az utasítások végrehajtása szigorúan a programban megadott sorrendben kezdődhet meg (pl. DEC Alpha 21164, ARM Cortex A8).
- Sorrenden kívüli (out-of-order) superskalár processzorok viszont az utasításokat dinamikusan átrendezve (de a program szemantikáját megtartva) próbálják elérni a széles pipeline minél jobb kihasználását (pl. Intel Pentium III, ARM Cortex A9).

Az in-order és az out-of-order szuperskalár processzorok közötti különbség jobb megértéséhez vegyük az alábbi utasítássorozatot:

```
i1:  R1 ← R2 + R3
i2:  R4 ← R1 - R5
i3:  R7 ← R8 - R9
i4:  R0 ← R2 + R3
```

Két-utas szuperskalár pipeline-t feltételezünk. In-order esetben az utasítások sorrendje nem változhat, tehát az i1 és i2 közötti RAW függőség miatt az első két utasítás nem hajtható végre párhuzamosan. Először az i1 hajtódik végre, majd jó néhány ciklussal később, amint az `R1` értéke előáll, a végrehajtási fázisba léphet mind az i2, mind az i3 egyszerre (mivel ezek függetlenek), végül a következő ciklusban az i4 is.

Órajelciklus	Utasítások
1	i1
2	i2, i3
3	i4

Out-of order esetben a processzor átrendezheti az utasítások sorrendjét. Mivel az i2 nem hajtható végre, amíg az i1 meg nem adja az R1 értékét, ezért a processzor az i1 mellé az i3-mat párosíthatja párhuzamos végrehajtásra (ezek között ugyanis nincs adatfüggőség). A következő ciklusban a végrehajtási fázisba léphet az i4 is, végül, ha az R1 értéke előállt, az i2 is.

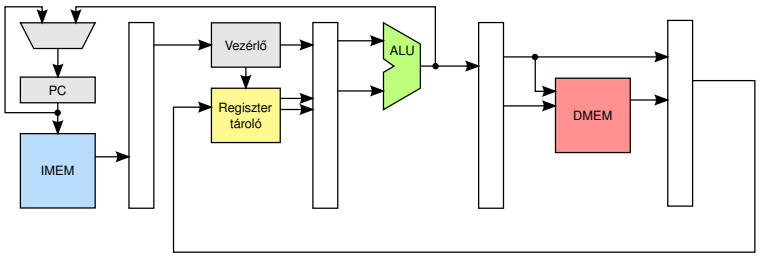
Órajelciklus	Utasítások
1	i1, i3
2	i2, i4

Látható, hogy az out-of-order feldolgozás sokkal nagyobb játékteret enged a műveletei egységek jobb kihasználásához, cserébe jóval bonyolultabb a megvalósítása.

Megjegyezzük, hogy a szakirodalomban az in-order szuperskalár pipeline-t is statikusan ütemezettnek tekintik, hiszen a fordító által generált utasítássorrend sok mindent előre meghatároz.

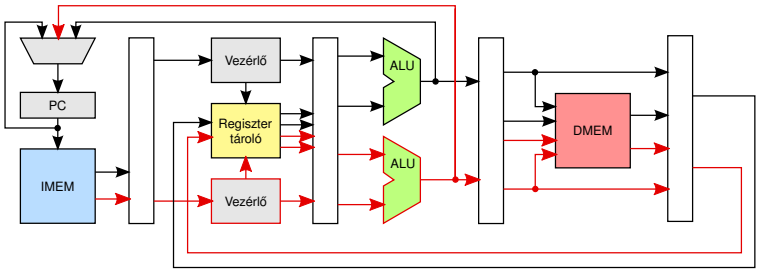
15.3.1. In-order szuperskalár pipeline

A 15.4. ábra a 12.4. fejezetben megismert 5 fokozatú pipeline egyszerűsített implementációját ábrázolja, szuperskalár viselkedés nélkül.



15.4. ábra. Klasszikus 5 fokozatú pipeline - egyszerűsített felépítés

Ha a pipeline szélességét 2-re akarjuk növelni, a 15.5. ábrán látható felépítéshez jutunk.



15.5. ábra. Klasszikus 5 fokozatú pipeline – 2-utas in-order szuperskalár feldolgozással

Az alábbiakban megnézzük, hogy a szuperskalár feldolgozás megvalósítása milyen változtatásokat kíván a pipeline különböző fázisaiban.

IF - Utasításlehívás

Ha az m lehívandó utasítás egymást szekvenciálisan követi (nem tartalmaz ugrást), akkor a dolgunk nem nehezebb, mint az 1-utas esetben: a PC által hivatkozott címről le kell hívni mind az m utasítást. A memória szószélesség növelésével ez egyetlen memóriatranzakcióval megoldható. Ha az m utasítás tartalmaz ugrást is, akkor nehézségekbe ütközünk, mert nincs idő arra, hogy cache memóriából egynél több címéről is adatokat hívjunk le (nem fér bele egy ciklusidőbe). A legegyszerűbb megoldás, hogy ilyenkor az adott ciklusban csak az elágazásig hívjuk le az utasításokat. Így ugyan nem hívunk le annyi utasítást, amennyit lehetne, de "néhányat" igen, tehát emiatt nem marad feldolgozandó utasítás híján egyetlen fázis sem.

ID - Dekódolás

m utasítás párhuzamos dekódolása során bizonyos feladatok elvégzését lehet zavartalanul párhuzamosan végezni (pl. ALU műveleti kód kinyerése az utasításból, esetlegesen tartalmazott skalár konstans kinyerése, stb.), de más feladatok összetettebbé válnak.

Nehézségek:

- Továbbra is csak egy regisztertárolónk van, és ebből az egy tárolóból kell kiolvasni mind az m utasítás legrosszabb esetben $2 \cdot m$ operandusának értékét, egyetlen ciklusidő alatt. A $2 \cdot m$ kiolvasás szekvenciális elvégzésére nincs idő, célszerű tehát többportos regisztertárolót alkalmazni, amely a $2 \cdot m$ kiolvasást a $2 \cdot m$ porton keresztül egyidejűleg el tudja végezni. Ez természetesen lényegesen növeli a regisztertároló bonyolultságát.
- Az adatfüggőségek detektálása és a forwarding utak kiválasztása a pipeline szélességével négyzetesen több ellenőrzést (összehasonlítást) igényel, hiszen minden fázisban m utasítás van, és az adatfüggőségek felderítéséhez mindegyik utasítás forrásoperandusait mindegyik későbbi utasítás eredményregiszterével össze kell hasonlítani.
- A fel nem oldható adatfüggőség esetén, mint korábban láttuk, "szünetet" kellett beiktatni, tehát nem engedtük továbblépni az utasítást a következő fázisba. Egy utas feldolgozás esetén ez egyértelmű, de mit tegyünk, ha m utasítás van egyidejűleg az ID fázisban, és ezek egy része megállni kényszerül, másik része pedig tovább mehetne? Az in-order viselkedésből adódóan, ha az m utasítás közül a program szerinti korábbi utasítás kényszerül várakozásra, akkor minden nála későbbi utasítás is várakozni fog, hiszen későbbi utasítás nem előzhet be egy korábbit. Erre az esetre vonatkozik a következő példa (sárga szín jelöli az adatfüggőség miatt várakozni kényszerülő utasítást):

Utasítás	**1**	**2**	**3**	**4**	**5**	**6**	**7**
i1: …	IF_1	ID_1	EX_1	MEM_1	WB_1		
i2: $R1 \leftarrow MEM[R0 + 4]$	IF_2	ID_2	EX_2	MEM_2	WB_2		
i3: $R2 \leftarrow R1 + 32$		IF_1	ID_1	szün	EX_1	MEM_1	WB_1
i4: $R3 \leftarrow R0 * R4$		IF_2	ID_2	szün	EX_2	MEM_2	WB_2

Ha az m utasítás közül egy későbbi kényszerül várakozásra, a program szerint őt megelőzőket természetesen továbbengedjük a back-end felé. A kérdés most már csak az, hogy az így felszabaduló helyekre beengedjünk-e utasításokat az IF fázisból, vagy ne. Természetesen hatékonyabb megoldás, ha a felszabaduló helyekre újakat hozunk, melyek dekódolása elkezdődhet, és az IF fázisban, ahonnan jöttek, helyükre új utasítások hívhatók le. Ez történik az alábbi példában, melyben az i5 (ha nincs a végrehajtását akadályozó egymásrahatás) beléphet az ID fázisba, az ott várakozó i4 mellé (az in-order elv ezzel nem sérül):

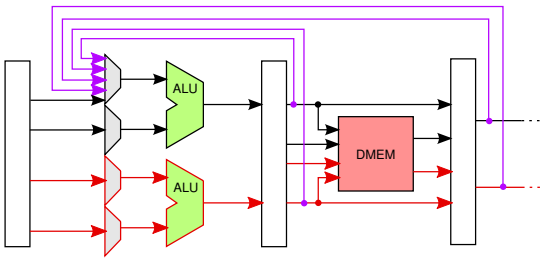
Utasítás	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
i1: …	IF_1	ID_1	EX_1	MEM_1	WB_1			
i2: $R1 \leftarrow MEM[R0 + 4]$	IF_2	ID_2	EX_2	MEM_2	WB_2			
i3: $R3 \leftarrow R0 * R4$		IF_1	ID_1	EX_1	MEM_1	WB_1		
i4: $R2 \leftarrow R1 + 32$		IF_2	ID_2	szün	EX_2	MEM_2	WB_2	
i5: …			IF_1	ID_1	EX_1	MEM_1	WB_1	
i6: …			IF_2	szün	ID_2	EX_2	MEM_2	WB_2

Ennél lényegesen egyszerűbb az implementáció, ha nem töltjük ki a felszabaduló helyeket, és az IF fázisban lévő m utasítást csak egyszerre (mindent vagy semmit alapon) engedjük az ID fázisba lépni:

Utasítás	1	2	3	4	5	6	7	8
i1: …	IF_1	ID_1	EX_1	MEM_1	WB_1			
i2: $R1 \leftarrow MEM[R0+4]$	IF_2	ID_2	EX_2	MEM_2	WB_2			
i3: $R3 \leftarrow R0 * R4$		IF_1	ID_1	EX_1	MEM_1	WB_1		
i4: $R2 \leftarrow R1 + 32$		IF_2	ID_2	szün	EX_2	MEM_2	WB_2	
i5: …			IF_1	szün	ID_1	EX_1	MEM_1	WB_1
i6: …			IF_2	szün	ID_2	EX_2	MEM_2	WB_2

EX - Végrehajtás

A műveleti egységek többszörözése nem jelent problémát. A forwarding utak nagy száma azonban igen! Mivel minden egyes fázisban potenciálisan m utasítás tartózkodik, és bármelyik utasítás operandusa megegyezhet bármelyik korábbi utasítás eredmény regiszterével, a forwarding utak száma m-mel négyzetesen arányos. A forwarding utakon 32 vagy 64 bites adatokat kell továbbítani, tehát a nagy számú forwarding busz elvezetése, fogyasztása, késleltetése gyorsan határt szab a pipeline szélességének. A sok forwarding úttal együtt az ALU operanduskiválasztó multiplexerei is szélesek lesznek. Azt, hogy milyen sok forwarding útról van szó, a 15.6. ábra szemlélteti: és ez csak a legegyszerűbb eset, és az ábra csak az egyik ALU egyik operandusának forwarding útjait tartalmazza.



15.6. ábra. Forwarding utak 2-utas in-order pipeline egyik ALU-jának egyik operandusára

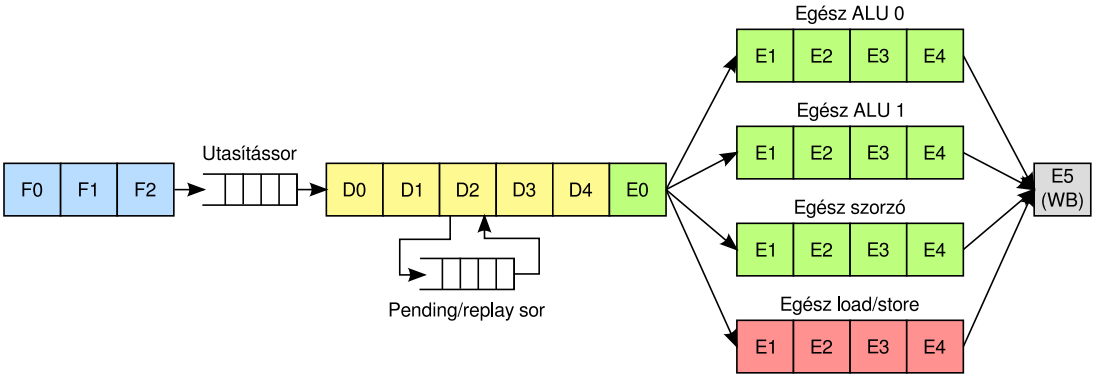
A problémát a műveleti egységek csoportosításával lehet enyhíteni. A műveleti egységeket *cluster*ekbe csoportosítjuk, és a forwarding utakat csak a clustereken belül építjük ki. Ha egy utasítás RAW függőségben áll egy másik clusterbe tartozó utasítással, akkor a szükséges operandust nem tudja forwarding segítségével megkapni, több ciklusnyi szünetet kell tartania, mire a várva várt operandus egy regiszterbe beírásra kerül, ahonnan aztán a következő lépésben ki tudja olvasni és így a végrehajtása elkezdhetővé válik.

MEM - Memóriaműveletek

A MEM fázisban tartózkodó Load és Store utasításoknak a memóriát (illetőleg az adat cache-t) egyidejűleg kellene elérnie, egy lépésben akár m különböző címről/címre kell adatokat olvasni/írni. Ezt vagy többportos memóriával lehet megvalósítani (de ekkor kezelni kell az egyező vagy átfedő címre hivatkozásból eredő konfliktusokat is), vagy pedig továbbra is csak egy memóriaműveletet engedünk végrehajtani ciklusonként. Ez utóbbi lehetőség nem rontja a pipeline hatásfokát olyan drasztikusan, mint amennyire első pillantásra látszik: a memóriaműveletek a statisztikák szerint program utasításainak csak a harmadát teszik ki. Tehát pl. egy kétutas in-order szuperskalár pipeline-ban jó eséllyel a MEM fázisban lévő két utasítás közül legfeljebb az egyik fog a memóriához fordulni. Ha véletlenül mégis mindkettő memóriaművelet lenne, akkor feldolgozási egymásrahatás alakul ki, és a későbbi utasításnak meg kell várnia, míg a korábbi végez a memóriaműveletekkel.

WB - Regiszter-írás

Ahhoz, hogy több utasítás is tudja egyidejűleg írni a regisztereket, több-portos regiszter tárolót kell alkalmazni.



15.7. ábra. Az ARM Cortex-A8 processzor 2-utas in-order szuperskalár futószalagja

Összességében elmondható, hogy az egyes fázisokban a fentebb felmerülő problémákat megoldva az in-order szuperskalár pipeline viszonylag egyszerűen megvalósítható, a működés lényegében megegyezik azzal, amit a klasszikus 5 fokozatú pipeline kapcsán megismertünk. A tárgyalt egyszerű in-order szuperskalár pipeline-t éppúgy ki lehet terjeszteni különböző késleltetésű műveleti egységek kezelésére, mint ahogy azt az egy utas megfelelőjével tettük.

Az ARM Cortex-A8 processzor pipeline-ja

Ebben a fejezetben megismerkedünk az ARM saját tervezésű, 2005-ben bejelentett és 2008 körül elterjedt processzorának, a Cortex-A8-nak a felépítésével[1].

Az ARM Cortex-A8 egy két-utas in-order szuperskalár futószalaggal rendelkezik (15.7. ábra), amely, mindamellett, hogy a tárgyalt elveket szépen demonstrálja, tartogat néhány egyedi megoldást, különlegességet is.

Utasításlehívás Az utasításlehívó egység ebben a processzorban külön életet él. Folyamatosan hívogatja le az utasításokat, spekulatív módon, órajelenként akár 4-et is. A lehívott utasítások az utasítássorba (instruction queue) kerülnek, *melynek funkcióját tekintve semmi köze nincs a sorrenden kívüli végrehajtásnál látott utasítástárolóhoz.* Az utasításlehívás három fázisból áll:

- F0: Ez a fázis generálja a utasításszámláló következő értékét reprezentáló virtuális címet. Ha az előző utasítás ugrás volt, akkor ezt a címet az elágazásbecslő szolgáltatja, ellenkező esetben a korábbi utasításszámlálót inkrementálja.
- F1: A lehívó egység ebben a fázisban adja ki a címet az utasítás cache-nek. Ugró utasítás esetén az utasításszámláló értéke alapján kiválasztja a dinamikus elágazásbecsléshez használt adatokat.
- F2: Ebben a fázisban érkezik meg az utasítás cache-től az utasításszó, melyet az utasítássorba kell helyezni. Ha ugrásról van szó, akkor az ugrási cím becslését az F2 fázisban küldik vissza F0-nak, hogy onnan folytathassa az utasítások lehívását (ha az F1-ben lévő utasítás rossz ágról érkezett, akkor azt eldobja).

Az utasítássor 12 ARM utasítás tárolására képes, fő célja, hogy kiegyenlítse az utasításlehívás és a végrehajtás sebességének ingadozását. Ha az utasítás végrehajtása során bármi fennakadás van (stall, például egymásrahatás miatt), akkor a lehívó egység nem áll meg, az utasítássor kapacitásának erejéig folytathatja a tevékenységét. Ha pedig a lehívó egység akad meg (pl. elágazáskor, cache hiba esetén, stb.), akkor az utasítássorban még mindig van pár utasítás, ami a probléma megoldásáig részben vagy teljesen el tudja látni a végrehajtóegységeket feladattal.

Dekódolás Feladata az gépi kódú utasítás értelmezése és futtatásra előkészítése, 5 ciklust is igénybe vesz. A dekódolás során különleges szerepe lesz még egy várakozási sornak, a ”pending/replay” sornak is.

- D0 és D1: Meghatározásra kerül az utasítás típusa, a forrás- és eredményoperandusok, és a végrehajtáshoz szükséges műveleti egység típusa.

[1] Azért esett a választás erre a viszonylag régi processzorra, mert nagyon jól ismert a struktúrája, nem utolsósorban [42]-nak köszönhetően. A processzortervezők általában ennél jóval kevesebb információt tesznek publikusan elérhetővé.

- D2: Az utasítást a ”pending/replay sorba” teszi, illetve korábbit kivesz onnan.
- D3: A statikus ütemezés megvalósítása. Itt kezdődik a szuperskalár viselkedés. Az ütemező ránéz a program szerint következő két utasításra (hiszen ez egy in-order szuperskalár processzor), és ellenőrzi, hogy lesz-e egymásrahatás a végrehajtásuk során, illetve hogy végrehajthatók-e együtt egyáltalán. A D3 fázist 0, 1 vagy 2 utasítás hagyhatja el, és miután elhagyta, ezek az utasítások már nem állhatnak meg többé a végrehajtásuk végéig.
- D4: Előállítja az ALU és a load/store egységek számára a megfelelő vezérlőjeleket.

A ”pending/replay” sor valójában két várakozó sort jelent. A ”pending” sorban vannak a félig dekódolt utasítások, várják, hogy D3-ba léphessenek. Előfordul, hogy az ütemező csak 1 utasítást enged tovább innen, ekkor az ott ragadt utasítás ebben a sorban várja meg az új párját, amivel majd együtt haladhat végig a 2-utas pipeline hátralévő részén.

A ”replay” sorban a végrehajtás alatt álló, de még be nem fejezett utasítások állnak, ennek a szerepe sokkal érdekesebb. Ez a processzor ugyanis különleges módon kezeli az egymásrahatásokat. Ha az ütemező egymásrahatást észlel, nem várja ki, amíg elmúlik, hanem megbecsüli, hogy kb. mikor fog megoldódni, és úgy indítja el a következő utasítást, hogy mire a kritikus részhez ér, már becslése szerint rendeződjön a probléma. Például ha egy olyan utasítás várja a D3-ban az indítását, amelyik egy korábbi, cache-hibát okozó load művelet eredményére vár, akkor a klasszikus esetben meg kellene várni, míg a cache-blokk megérkezik, és befejeződik a load. A Cortex-A8 processzor ütemezője azonban megbecsüli, hogy ez meddig fog tartani, és úgy indítja rá a várakozó utasítást, hogy a neki kellő adat pont akkora jelenjen meg, mire szüksége lesz rá. Ez persze időnkét nem sikerül, mert hamarabb ér oda az utasítás, mint az adat. Mivel a D3 utáni fázisokban várakozni ebben a pipeline-ban nem lehet, ezért a peches utasítást törölni kell, majd a ”replay” sorból újra kell indítani a végrehajtását.

Végrehajtás Ez a processzor (az opcionális kiegészítések nélkül) háromféle műveleti egységgel rendelkezik:

- Egész ALU: összeadás, kivonás, bitműveletek. Kettő van belőle, így egyszerre két ALU-t használó utasítás is végrehajtható párhuzamosan.
- Egész szorzó: csak egyetlenegy van belőle, és annak is kötött a használata, ugyanis a pipeline-ban utazó utasításpárok közül csak az öregebbik használhatja. Ha a fiatalabbnak is szüksége van szorzásra, akkor az ütemező csak a következő ciklusban engedi ki a D3-ból, így abban a körben már az lesz az öregebbik.
- Load/store egység: ebből is csak egy van, de azt az egyet az utasításpár bármelyik tagja igénybe veheti.

Az elágazások kiértékelését az Egész ALU műveleti egységek végzik, de az együtt utazó utasításpárok közül mindig legfeljebb csak egy lehet ugró utasítás. Ha nem ez a helyzet, akkor az ütemező az öregebbet engedi tovább a D3-ból, egyedül.

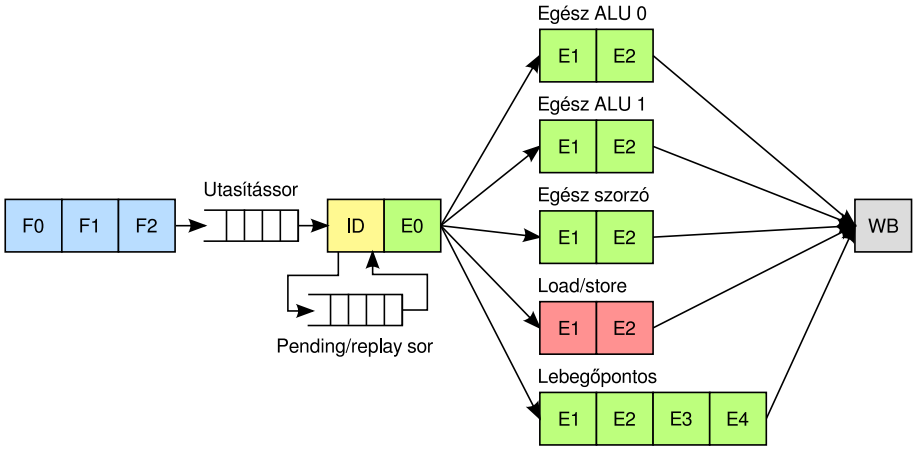
A végrehajtás minden esetben 6 fázisból áll. Az első fázisban (E0) történik a regiszterek értékének kiolvasása a regisztertárolóból. A további 4 lépés (E1-E4) magának a műveletnek a végrehajtása, az utolsó lépés (E5) pedig a write-back, az eredmény visszaírása a regisztertárolóba.

- Egész ALU műveletek esetén: E1: opcionális shiftelés (az ARM utasításkészletben minden aritmetikai művelet kiegészíthető egy bitenkénti eltolással), E2: ALU művelet, E3: az eredmény potenciális telítődésének kezelése, E4: elágazásbecslő táblázatainak frissítése.
- Egész szorzó: E1-E3: szoroz, E4: a szorzás eredményét hozzá tudja adni az eredményregiszterhez.
- Load/store egység: E1: címszámítás, E2-E4: adat cache kiolvasása, ill. írása.

Ha összeadjuk a fázisok számát, megkapjuk, hogy a pipeline mélysége 14.

Az ARM Cortex-A53 processzor pipeline-ja

Az ARM Cortex-A53 az imént tárgyalt Cortex-A8 modernebb változata, 2012-ben jelentették be, az első implementációk 2014 végén kerültek a piacra. Ez a processzor is 2-utas in-order szuperskalár képességekkel rendelkezik, mind a pipeline szerkezete, mind az alapmegoldások terén nagyban hasonlít a Cortex-A8-ra (15.8), de a tervezők ezúttal mindent bevetettek, hogy ebből a felépítésből kihozzák a maximumot. Az utasításlehívás továbbra is 3 fázisból áll, melyek szerepe megegyezik a Cortex-A8-nál látottal. A dekódolás azonban már egyetlen órajel alatt megtörténik. A műveleti egységek késleltetése is csökkent, a lebegőpontos műveletek 4, minden más pedig 2 lépés alatt befejeződik. Összességében a pipeline mélysége 8.



15.8. ábra. Az ARM Cortex-A53 processzor 2-utas in-order szuperskalár futószalagja

Míg a Cortex-A8 két utasítást csak korlátozott esetekben tudott ténylegesen egyszerre végrehajtani, a Cortex-A53 sokkal megengedőbb. Megduplázták a load/store, a lebegőpontos és a vektor feldolgozó egységek számát, így most már gyakorlatilag tetszőleges két egymást követő utasítás végrehajtása elkezdhető egyszerre, ha azok függetlenek egymástól. A forwarding utak számát megnövelték, így a pipeline ritkábban áll meg adat-egymásrahatás miatt. A memóriakezelés is javult, a TLB nagyobb, és asszociatívabb, az idő előtti betöltést végző algoritmusok (prefetch) fejlettebbek lettek, és az elágazásbecslés is lényegesen hatékonyabbá vált [20].

15.3.2. Out-of-order szuperskalár pipeline

A 13.7. fejezetben tárgyalt, out-of-order végrehajtásra képes processzorokat nagyon könnyű szuperskalárrá tenni. Egyszerűen több utasítást kell lehívni és dekódolni egyszerre, majd az utasítástárolóba és a sorrend-visszaállító bufferbe helyezni.

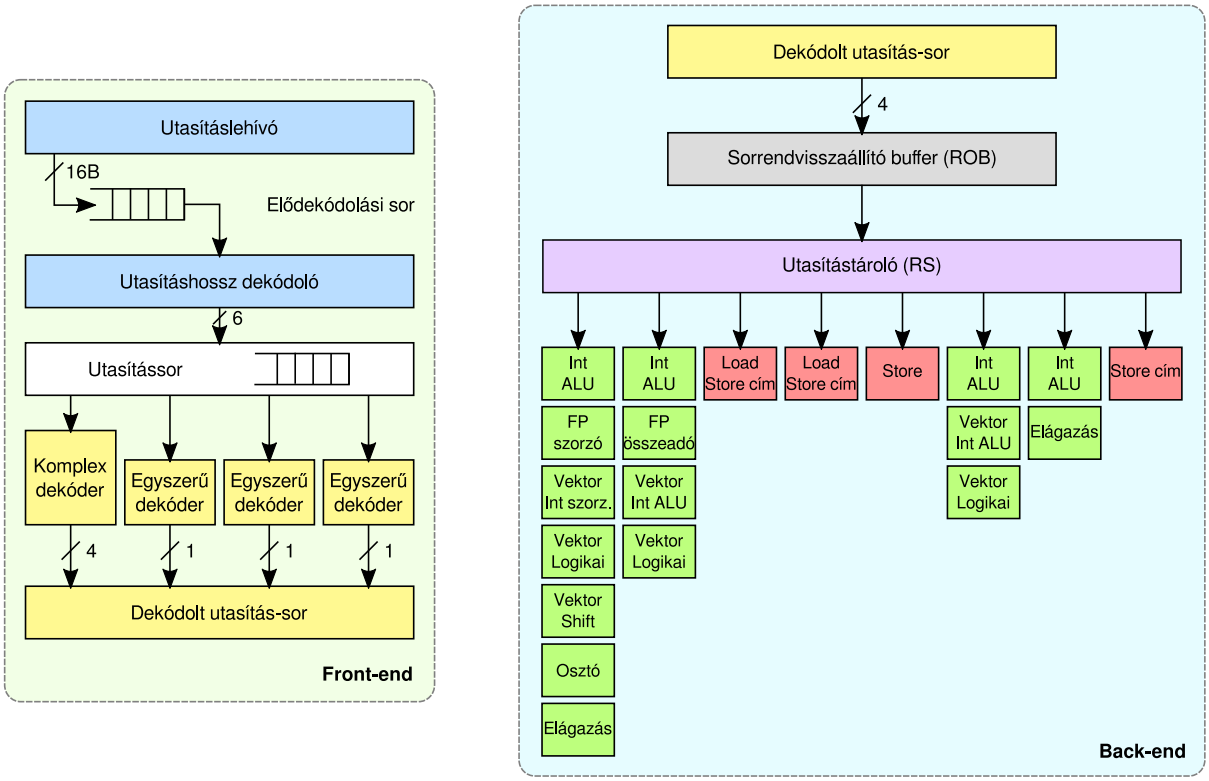
A dinamikus ütemezőt csak annyiban kell megváltoztatni, hogy ha a precedenciagráf alapján több utasítás is végrehajtható állapotban van egyszerre, akkor a műveleti egységek kapacitásának erejéig egyazon ciklusban többet is végrehajtásra ütemezhet.

Az Intel Haswell processor pipeline-ja

Az Intel 2013-ban jelentette be a Haswell mikroarchitektúrát, ami a Core i7-4xxx sorozatú processzorok alapja. Futószalagja kifejezetten széles, sorrenden kívüli szuperskalár felépítésű, nagy számú feldolgozó egységgel.

A pipeline szerkezete a 15.9. ábrán látható ([28]). Mivel az x86/x64 architektúra CISC filozófiát követ, a front-end nem olyan egyszerű, mint az eddig látott RISC példákban volt. Az első probléma, hogy az x86/x64 változó hosszúságú utasításkódolást követ, az utasítások mérete 1 és 15 bájt közé esik. Az utasításlehívó ennek megfelelően nem is utasításokat, hanem bájtokat hív le az utasítás cache-ből, minden ciklusban 16-ot, és az elődekódolási sorba helyezi azokat. Az utasításhossz dekódoló feladata az utasításhatárok megállapítása. Az összetartozó bájtokat utasításokká ragasztja össze, és az utasítássorba teszi, ciklusonként legfeljebb 6-ot. Az utasítássor a dekódolásra váró utasításokat tartalmazza, szerepe nem keverendő össze az utasítástárolóval. A dekódolás során a Haswell az x86/x64 összetett műveleteket leíró CISC utasításaiból fix méretű, RISC-szerű mikró utasítások sorozatát állítja elő. Minden ciklusban legfeljebb 4 utasítást képes feldolgozni, de csak akkor, ha a 4 utasításból csak az egyik igényel komplex dekódert. Komplex dekóder kell minden olyan utasítás dekódolásához, amelyből egynél több mikró utasítás keletkezik. Az így előállított mikró utasítások egy újabb sorba, a dekódolt utasítás-sorba kerülnek további feldolgozásra várva.

A dekódolt mikró utasítások ezután kapnak egy bejegyzést a ROB-ban, melynek kapacitása 192. A Haswell az regiszterátnevezés céljára külön fizikai regisztertárolót használ, nem pedig a ROB-ot, ahogy azt a 14.3.2. fejezetben láttuk. A fizikai regiszterkészlet 168 egész és 168 vektor, ill. lebegőpontos regisztert tartalmaz, a load és store utasítások beolvasott, illetve kiírandó adatait (melyet a 14.3.2.fejezetben szintén a ROB tárolt) egy 72 bejegyzéses Load Buffer és egy 42 bejegyzéses Store buffer tárolja. Egy ciklusban legfeljebb 4 új mikró utasítás léphet a ROB-ba, illetve a ROB legfeljebb 4 elkészült utasítást tud érvényre juttatni (RT - retire fázis), és törölni a ROB-ból.



15.9. ábra. Az Intel Haswell processzor front-endje és back-endje

A ROB-ba lépés és a regiszterátnevezés után a mikró utasítás a 60 bejegyzéses, centralizált utasítástárolóban foglal helyet, amíg a dinamikus ütemező végrehajtásra ki nem jelöli. A dinamikus ütemező minden ciklusban legfeljebb 8 utasítás végrehajtását kezdheti meg, mindegyiket különböző ”port” felé indítja útnak. Minden port más és más feldolgozóegység készlettel rendelkezik, ezért hiába van 8 végrehajtásra kész utasítás, nem feltétlenül lehet azokat egyszerre elindítani. Pl. az ábrán is látható, hogy egész ALU-val csak 4 port rendelkezik, ezért 4-nél több egész művelet nem indulhat el egyszerre.

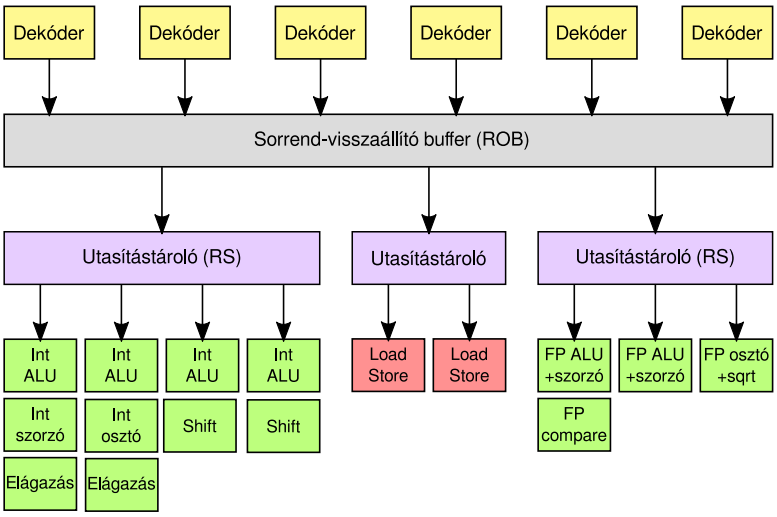
Az Apple Cyclone processzor pipeline-ja

A Cyclone az Apple saját fejlesztésű, 64 bites ARM utasításkészlettel rendelkező, out-of-order szuperskalár processzormagja, melyet 2013-ban jelentett be. Első implementációja az Apple A7-ben, az iPhone 5s processzorában mutatkozott be, finomított, továbbfejlesztett változatát pedig az Apple A8 tartalmazza (erre épül az iPhone 6).

A pipeline felépítéséről sokáig semmilyen információ nem állt rendelkezésre, egészen addig, amíg az Apple ki nem egészítette az LLVM fordítóprogram-csomagot egy pipeline modellel a Cyclone-specifikus optimalizálás érdekében. Ebből a 15.10. ábrán látható futószalag körvonalazódott ([37]).

Az ARM utasítások dekódolását egymással párhuzamosan 6 dekóder végzi, és ez a 6 dekódolt utasítás egyszerre léphet be a sorrend-visszaállító bufferbe, melynek kapacitása 192 (akár a Haswellé). A sorrend-visszaállító buffer mellet az utasításokat az utasítástárolóban is el kell helyezni. A Haswell-el ellentétben a Cyclone nem centralizált, hanem részben elosztott utasítástárolót alkalmaz, egyet az egész műveleteknek (kapacitása 48), egyet a memóriaműveleteknek (28 bejegyzéses), valamint egyet a lebegőpontos és vektorműveleteknek (kapacitása 48). A dinamikus ütemező nem kevesebb, mint 9 porttal rendelkezik (ez több mint a amennyi a Haswell-nek van!), tehát – szerencsés esetben – 9 utasítás végrehajtását indíthatja el, ha ez a 9 olyan végrehajtó egységeket igényel, melyek különböző portokon érhetők el.

A Cyclone ezt a mobil eszközökben szokatlanul erős felépítést az energiatakarékosság javítására fordítja. Az erős processzor ugyanis pillanatok alatt elvégzi az aktuális feladatát, majd mehet vissza aludni egy alacsony fogyasztású állapotba.



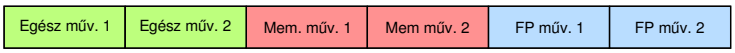
15.10. ábra. Az Apple Cyclone processzor pipeline-ja

15.4. A VLIW architektúra

A szuperskalár processzorokkal szemben a VLIW architektúra statikus ütemezést használ, vagyis az egyidejűleg végrehajtható utasításokat a fordítóprogram válogatja össze fordítási időben.

A VLIW processzorok 1 utas pipeline-nal rendelkeznek, de nem utasításokon, hanem független utasításokból álló utasításcsoportokon (VLIW) dolgoznak. Ezek az utasításcsoportok egy darabban, egy egységként haladnak a pipeline-ban lépésről lépésre. A fordítóprogram garantálja, hogy az egy csoportba tartozó utasítások egymástól függetlenek, így párhuzamosan végrehajthatók. Ennek megfelelően a processzor nem is ellenőrzi a csoporton belüli függőségeket. Az utasításcsoporton belül az egyes utasításoknak nincs külön címe, a csoport minden utasítása ugyanahhoz a címhez (utasításszámlálóhoz) tartozik.

Az utasításcsoportok (15.11. ábra) rögzített számú utasítást tartalmaznak. A csoporton belül minden pozícióhoz egy műveleti egység is tartozik, tehát a csoportok szerepe nem csak a független utasítások kijelölése, hanem a műveleti egységekhez való hozzárendelése is. A nem használt pozíciókat NOP (No Operation) utasítások töltik ki. Az utasításcsoport annyi idő alatt halad át a pipeline-on, amennyi ideig a benne lévő, leghosszabb feldolgozást igénylő utasítás végrehajtása tart.



15.11. ábra. Utasításcsoport a VLIW architektúrában

A klasszikus VLIW processzorok nem ellenőriznek semmilyen egymásrahatást, így nem tudják az utasításokat várakoztatni sem. Ha adat-egymásrahatás miatt egy utasításcsoportnak várakoznia kellene (megvárnia, hogy az előző utasításcsoport befejezze az ő egyik utasításának a forrásoperandusának a kiszámolását), akkor a fordító csupa NOP utasításból álló csoportot szúr be, pont annyit, amennyi ahhoz kell, hogy az egymásrahatás ne okozzon gondot.

Nézzünk egy konkrét példát az utasítások csoportosítására és ütemezésére, melyben az egész műveletek 1, a memóriaműveletek 3, a lebegőpontos műveletek pedig 4 ciklusideig tartanak, az utasításcsoport összetétele pedig a 15.11. ábra szerinti:

A program:

```
i1:  R3 ← MEM[R1+0]
i2:  R4 ← MEM[R1+4]
i3:  D1 ← MEM[R1+8]
i4:  R5 ← R3 + R4
i5:  R6 ← R3 - R4
i6:  D2 ← D1 * D1
i7:  MEM[R2+0] ← R5
i8:  MEM[R2+4] ← R6
i9:  MEM[R2+8] ← D2
```

Csoportokba rendezve:

	Egész 1	**Egész 2**	**Mem 1**	**Mem 2**	**FP 1**	**FP 2**
1:	NOP	NOP	i1	i2	NOP	NOP
2:	NOP	NOP	i3	NOP	NOP	NOP
3:	NOP	NOP	NOP	NOP	NOP	NOP
4:	i4	i5	NOP	NOP	NOP	NOP
5:	NOP	NOP	i7	i8	i6	NOP
6:	NOP	NOP	NOP	NOP	NOP	NOP
7:	NOP	NOP	NOP	NOP	NOP	NOP
8:	NOP	NOP	NOP	NOP	NOP	NOP
9:	NOP	NOP	i9	NOP	NOP	NOP

A sok-sok NOP rögtön azt sugallja, hogy nem sikerült elég sok párhuzamosan végrehajtható utasítást találni ahhoz, hogy az utasításcsoportok jobban kitöltöttek legyenek. Ráadásul 4 csupa NOP csoportot is be kellett iktatni, hogy az adat egymásrahatások ne okozzanak gondot.

Az előbbiek alapján tehát a fordítóprogramnak az alábbi feladatai vannak:

- az utasítások utasításcsoportokba rendezése úgy, hogy az utasításcsoportok a lehető legjobb kitöltöttségűek legyenek (azaz a műveleti egységek jól ki legyenek használva),
- az egy csoportba kerülő utasítások párhuzamos végrehajthatóságának biztosítása,
- az utasításcsoportok ütemezése a megfelelő számú csupa NOP csoporttal úgy, hogy az egymásrahatások ne okozzanak gondot.

A kereskedelmi forgalomban lévő implementációk tipikusan 3-4, extrém esetben akár 28 utasításból álló csoportokat használnak. Mivel az utasításfeldolgozás legnagyobb komplexitást igénylő feladatait (mint pl. az egymásrahatások kezelését) nem kell a processzornak elvégeznie, a processzor nagyon egyszerű lesz: a VLIW architektúrát többnyire olcsó, kis fogyasztású beágyazott rendszerekben alkalmazzák. Az egyszerűség akkor is szempont, ha VLIW feldolgozóegységekből sokat kell egy lapkára elhelyezni (pl. az AMD grafikus processzorai az R300 óta mind VLIW architektúrára épülnek - 3 vagy 4 utasítás/csoport), vagy ha a processzorral szemben elvárás a kiszámítható, spekulációtól és predikciótól mentes működés (mint a DSP-k, pl. TMS320C6x - 8 utasítás/csoport).

A klasszikus VLIW processzorok hátrányai:

- A lefordított program kizárólag azon a hardveren fut, amire lefordították, hiszen az egy csoportba tehető utasítások száma, a műveleti egységek darabszáma és késleltetése az utasításcsoportok kialakításánál ki lett használva. Nem lehet tehát a jövőben a processzorcsalád új tagjaként 1-2 extra műveleti egységet hozzáadni a processzorhoz, hiszen akkor növelni kellene a csoportba tehető utasítások számát, ami a régi szoftverek újrafordítását teszi szükségessé. Szintén újrafordítás szükséges, ha az új modellben a műveleti egységek késleltetése csökken, hiszen akkor kevesebb NOP-ot kell beszúrni adategymásrahatás esetén.
- Az egyik legnagyobb gond, hogy VLIW architektúrában a transzparens cache bevezetése gondot okoz! A transzparens cache alkalmazása esetén ugyanis a memóriaműveletek sebessége nem lesz többé állandó (hiszen cache találat és cache hiba esetén más-más lesz a késleltetés), sőt, nem lesz előre, fordítási időben kiszámítható. A fordító nem fogja tudni eldönteni, hogy adat-egymásrahatás esetén mennyi időt kell várni (hány csupa NOP csoportot kell beiktatni) a következő utasítás elkezdéséig. Ezért a klasszikus VLIW processzorok nem használnak cache-t.
- A programok mérete nagyon nagy is lehet a sok NOP miatt.

A klasszikus VLIW processzoroknál némileg rugalmasabbak a ***dinamikus VLIW processzorok***. Ahogy a fenti táblázatból kitűnik, ennél a megoldásnál az utasításcsoportok végrehajtási idejét a hardver (a processzor) határozza meg, tehát a dinamikus VLIW processzorok képesek az egymásrahatások kezelésére, így az utasításcsoportok szükség szerinti késleltetésére is. Ennek köszönhetően lehetővé válik pl. a cache kezelése is, hiszen cache hiba esetén a hosszabbra nyúlt memóriaművelet idejére a processzor képes a pipeline-t megállítani. Dinamikus VLIW processzorok esetén a fordítóprogram által generált kód is valamivel szélesebb körben használható: nem kell újrafordítani a programot, ha kijön egy újabb processzor a korábbiaktól eltérő (kisebb) késleltetésű műveleti egységekkel, a régi kód futni fog, új, a megváltozott késleltetéseknek megfelelő ütemezéssel (gyorsabban).

15.5. Az EPIC architektúra

Az EPIC architektúra a HP és az Intel közös kutatási programjának eredményeképpen jött létre 1994-ben, az első kereskedelmi forgalomba került megvalósítása a 2002-ben megjelent Intel Itanium processzor volt. Az volt a cél, hogy az utasításszintű párhuzamosítási lehetőségek felderítését a fordítóprogramok végezzék, de a VLIW korlátaitól minél inkább sikerüljön megszabadulni. Az teljesen racionális, hogy a párhuzamosság felderítését a fordítóprogramra érdemes bízni, hiszen a fordító által egyszerre áttekinthető nagy számú utasítás alapján jobb döntést lehet hozni, és az emiatt esetlegesen megnövekedett fordítási idő általában tolerálható. A VLIW hátrányain úgy sikerült túllépni, hogy az utasításcsoportokat az EPIC architektúrában kicsit másképp kell értelmezni. A VLIW rendszerekben az utasításcsoportok mérete megegyezett a processzor műveleti egységeinek számával, és a csoporton belüli pozíció még azt a bizonyos műveleti egységet is kijelölte, aminek az oda helyezett utasítást végre kell hajtania. Az EPIC rendszerben ezeket a kötöttségeket megszüntették: az utasításcsoportok egyedüli szerepe, hogy az egymással párhuzamosan végrehajtható utasításokat egységbe zárja.

15.12. ábra. Utasításcsoport a EPIC architektúrában

Egy utasításcsoport az EPIC architektúrában a 15.12. ábrán látható módon néz ki. Az ábra szerint 3 utasítás helyezhető egy csoportba. Azt, hogy a csoportban lévő utasításoknak milyen az összetétele, a fordító a ”template típus” mező kitöltésével jelzi. Pl. ha ”M”-mel jelöljük a memóriaműveleteket, ”I”-vel az egész, ”F”-fel a lebegőpontos műveleteket, ”B”-vel pedig az elágazásokat, akkor a 3 utasítást tartalmazó csoport template típusa az utasítások típusának egymás után írása. Pl. az ”MFI” típusú csoportban az első utasítás egy memóriaművelet, a második lebegőpontos, a harmadik pedig egész típusú aritmetikai művelet. További template típusok lehetnek: MFI, MMI, MII, MIB, MMF, MFB, stb., de nem feltétlenül minden lehetséges kombináció megengedett. Az EPIC processzorok gyártói a támogatott template típusokat előre rögzítik, és a processzorcsalád további generációinak megjelenésekor ezt legfeljebb bővíthetik (így megmarad a kompatibilitás).

Az EPIC processzorok jellemzően in-order pipeline-nal rendelkeznek, melyben az utasításcsoportok egyszerre, egy feldolgozási egységként haladnak fázisról fázisra. Az egymásrahatások észlelését és kezelését a hardver, tipikusan a pipeline ID fázisa végzi, a tanult módszerekkel (forwarding, szünetek beiktatása, stb.). Sőt, a pipeline lehet in-order szuperskalár felépítésű is (itt a szuperskalár viselkedés nem egyes utasításokra, hanem utasításcsoportokra vonatkozik). Fontos, hogy az utasításcsoport méretének semmi köze nincs a processzorban lévő műveleti egységek számához. Attól, hogy csak 3 utasítás helyezhető egy csoportba, a processzorban még lehet ennél sokkal több egység, ekkor az egymástól független csoportokat az in-order szuperskalár pipeline-nál látott módon párhuzamosan végre lehet hajtani. Az egymástól független csoportok felderítését a fordító is támogathatja: a template típus mező egyik bitje jelezheti, hogy a soron következő csoport az aktuális csoporttól függetlenül végrehajtható. Ezek a megoldások lehetővé teszik, hogy egy processzorra lefordított program egy újabb, gyorsabb és több műveleti egységgel rendelkező típuson is változtatás (újrafordítás) nélkül fusson.

15.6. Hatékonyságot javító technikák statikusan ütemezett többutas pipeline-hoz

A VLIW és EPIC architektúrák hatékonyságának kulcsa, hogy a rendelkezésre álló párhuzamos végrehajtási lehetőségeket minél jobban kihasználjuk, azaz az utasításcsoportok legyenek minél jobban kitöltve párhuzamosan végrehajtható utasításokkal. A párhuzamosan végrehajtható utasítások összeválogatása gyakran nem egyszerű feladat. Ebben a fejezetben néhány, a VLIW és EPIC processzorok és a hozzájuk kifejlesztett fordítóprogramok által alkalmazott technikát mutatunk be az utasításcsoportok kitöltöttségének növelésére.

15.6.1. Ciklusok optimalizálása

Vegyük az alábbi, teljesen közönséges példát:

```
for (i=0; i<N; i++)
    B[i] = A[i] + C;
```

Nézzük meg, milyen elemi műveletek történnek a ciklus belsejében:

```
for (i=0; i<N; i++) {
    Load A[i]
    B[i] = A[i] + C;
    Store B[i]
}
```

Ebből egy naív fordítóprogram a következő kódot generálja:

```
i1:  loop: D1 ← MEM[R1+0]
i2:        R1 ← R1 + 8
i3:        D2 ← D0 + D1
i4:        MEM[R2+0] ← D2
i5:        R2 ← R2 + 8
i6:        JUMP loop IF R1 != R3
```

A kódban R1 mutat az ”A” tömb aktuális elemére, R2 a ”B” tömb aktuális elemére, R3-ban az ”A” tömb végét tároljuk ($R3 = R1 + N \cdot 8$, ha R1 eléri R3-mat, a ciklus megáll), D0-ban pedig ”C”-t. Feltesszük továbbá, hogy a lebegőpontos számok 8 bájtot foglalnak.

Egy klasszikus VLIW processzor számára ezeket az utasításokat a következőképpen lehet csoportosítani (2 egész és 2 lebegőpontos aritmetikai egységet, valamint 2 memória műveleti egységet feltételezve, lásd 1. ábra, az egész műveletek 1, a memóriaműveletek 3, a lebegőpontos műveletek pedig 4 ciklusideig tartanak):

	Egész 1	**Egész 2**	**Mem 1**	**Mem 2**	**FP 1**	**FP 2**
1:	i2		i1			
2:						
3:						
4:					i3	
5:						
6:						
7:						
8:	i5	i6		i4		

A példában a ciklusmag utasításai között (a Load és az összeadás, ill. az összeadás és a Store között) RAW egymásrahatás van, ezért nem tehetők egy utasításcsoportba. Először is, az adatfüggőségek miatt i3-nak meg kell várnia, míg i1 lefut (3 órajelciklus), i4-nek pedig meg kell várnia, míg i3 lefut (4 órajelciklus). Az adatfüggőségek kezelésére a VLIW processzorok csupa NOP csoportok beszúrását teszik szükségessé (az üresen hagyott cellák NOP-okat jelentenek). Ha dinamikus VLIW vagy EPIC architektúrát alkalmazunk, akkor a csupa NOP csoportokat nem kell ugyan a fordítóprogramnak beszúrnia, mert beszúrja a hardver automatikusan (ezek a szünetek az adatfüggőség feloldásához mindenképp szükségesek), de a futási időn ez nem segít. Ezzel a megoldással tehát egy iteráció (a ciklusmag egyszeri lefutása) 8 órajelciklust vesz igénybe, tehát 0.125 iteráció/órajelciklus sebességet kapunk.

Cikluskifejtés

A processzor kihasználtsága szempontjából jó lenne a függőségek miatti csupa NOP csoportok (vagy a hardveresen beszúrt szünetek) helyett valami hasznos műveletet végezni. Mivel tölthetnénk ki a sok üresjáratot, ha egyszer a ciklus műveletei (load - összeadás - store) függenek egymástól? A következő iterációk műveleteivel! Az ötlet a következő: mivel a ”for” ciklus egymás utáni iterációi egymástól függetlenek, írjuk át a ciklust úgy, hogy 4-esével lépjen, és a ciklus minden körében egyszerre négy műveletet végezzen:

```
for (i=0; i<N; i+=4) {
    B[i] = A[i] + C;
    B[i+1] = A[i+1] + C;
    B[i+2] = A[i+2] + C;
    B[i+3] = A[i+3] + C;
}
```

(Feltételeztük, hogy N osztható 4-gyel. Ellenkező esetben a maradék műveleteket a cikluson kívülre kell mozgatni.) Lássuk a generált kódot:

```
i1:  loop: D1 ← MEM[R1+0]
i2:        D5 ← D0 + D1
i3:        MEM[R2+0] ← D5
i4:        D2 ← MEM[R1+8]
i5:        D6 ← D0 + D2
i6:        MEM[R2+8] ← D6
i7:        D3 ← MEM[R1+16]
i8:        D7 ← D0 + D3
i9:        MEM[R2+16] ← D7
i10:       D4 ← MEM[R1+24]
i11:       D8 ← D0 + D4
i12:       MEM[R2+24] ← D8
i13:       R1 ← R1 + 32
i14:       R2 ← R2 + 32
i15:       JUMP loop IF R1 != R3
```

Ez a négyesével léptetett for ciklus akkor nyer értelmet, ha az egymással adat-egymásrahatásban álló utasításokat a lehető legjobban eltávolítjuk egymástól. Ha az egymástól függő utasítások olyan távol vannak egymástól, hogy mire a későbbinek szüksége van a korábbi eredményére, az már rendelkezésre fog állni, nem kell megállítani a pipeline-t és megvárni a korábbi utasítás befejeződését. Az átrendezett kód:

```
i1:  loop: D1 ← MEM[R1+0]
i2:        D2 ← MEM[R1+8]
i3:        D3 ← MEM[R1+16]
i4:        D4 ← MEM[R1+24]
i5:        R1 ← R1 + 32
i6:        D5 ← D0 + D1
i7:        D6 ← D0 + D2
i8:        D7 ← D0 + D3
i9:        D8 ← D0 + D4
i10:       MEM[R2+0] ← D5
i11:       MEM[R2+8] ← D6
i12:       MEM[R2+16] ← D7
i13:       MEM[R2+24] ← D8
i14:       R2 ← R2 + 32
i15:       JUMP loop IF R1 != R3
```

Az átrendezett kódhoz tartozó csoportosítás:

	Egész 1	Egész 2	Mem 1	Mem 2	FP 1	FP 2
1:			i1			
2:			i2			
3:			i3			
4:	i5		i4		i6	
5:					i7	
6:					i8	
7:					i9	
8:				i10		
9:				i11		
10:				i12		
11:	i14	i15		i13		

Ezzel a csoportosítással elértük, hogy ne kelljen csupa NOP csoportot beszúrni (vagy, dinamikus VLIW vagy EPIC processzorokban a pipeline-t az egymásrahatás miatt megállítani). A ciklus 4 iterációja (a ciklusmag 4 lefutása) 11 órajelciklust vett igénybe, tehát a végrehajtási sebesség 0.36 iteráció/órajelciklus. Ezt a technikát **ciklus kifejtés**nek (loop unrolling) nevezzük. Vegyük észre, hogy ebben a példában lenne értelme a ciklust akár 8-asával is léptetni, mert a VLIW processzorunkban 2 memória és 2 lebegőpontos egység is van. Ha ezt kihasználjuk, akkor 0.72 művelet/ciklus-ra növelhetjük a sebességet.

Ez a technika nem VLIW/EPIC specifikus, a cikluskifejtés egy általánosan használt ciklus optimalizálási technika.

Szoftver pipeline

Nem a ciklus kifejtés az egyetlen ciklus átszervezési technika, melynek segítségével teljesítménynövekedést lehet elérni. Most egy kicsit más oldalról közelítjük meg a problémát. Íme az előbbi példa:

```
for (i=0; i<N; i++)
    B[i] = A[i] + C;
```

A ciklus belsejében lévő 3 művelet nyilván összefügg, nem lehet őket egyszerre végrehajtani. Alakítsuk át a ciklust a következőképpen:

```
...
for (i=2; i<N-2; i++) {
     Load A[i]
      B[i-1] = A[i-1] + C;
     Store B[i-2]
}
...
```

Az átalakított ciklus ugyanazt csinálja, mint az eredeti, de ahhoz képest egy fontos előnye van: a ciklus belsejében lévő 3 művelet egymástól független, így azokat egy utasításcsoportba helyezve egy VLIW vagy EPIC processzor egy lépésben képes végrehajtani. Tehát amíg az ”A” i. elemét betöltjük, azzal egyidőben kiszámolhatjuk a ”B” i-1-edik elemét és a memóriába írhatjuk a ”B” i-2-ik elemét. Ezt a típusú ciklusszervezést **szoftver pipeline**-nak nevezik. Az analógia kedvéért: egy 3 fázisú hardver (utasítás-) pipeline estén egy adott ciklusban pillanatfelvételt készítve azt látjuk, hogy amíg az i. utasítás a 3. fázisban van, azzal egyidőben az i+1. utasítás a 2., az i+2. utasítás pedig az 1. fázisban van. A szoftver pipeline-ban a ”pipeline ciklus”-nak a programban lévő ciklus egy iterációja felel meg, az utasításoknak a ciklusmag (eredeti, átrendezés előtti formája), a fázisoknak pedig a ciklusmag utasításai. A szoftver pipeline-ná átalakított ciklus előtt és után intézkedni kell a ”pipeline feltöltéséről és kiürítéséről”, vagyis a ciklus elé ki kell venni a 0. és 1. elem betöltését, valamint a 0. elem kiszámítását, illetve a ciklus után az N-1. elem kiszámítását és az N-2. és N-1. elem memóriába írását.

A ciklusmag utasításait így egy csoportba tehetjük ugyan, de a ciklus egymás utáni iterációi között adatfüggőség van, ami miatt az utasításcsoportok közé csupa NOP csoportokat kell elhelyezni. Az üdvözítő megoldás ötvözi a ciklus kifejtést a szoftver pipeline-nal, így a következő megoldásra jutunk:

```
i1:  loop: D1 ← MEM[R1+0]
i2:        D2 ← MEM[R1+8]
i3:        D3 ← MEM[R1+16]
i4:        D4 ← MEM[R1+24]
i5:        R1 ← R1 + 32
i6:        D5 ← D0 + D1
i7:        D6 ← D0 + D2
i8:        D7 ← D0 + D3
i9:        D8 ← D0 + D4
i10:       MEM[R2+0] ← D5
i11:       MEM[R2+8] ← D6
i12:       MEM[R2+16] ← D7
i13:       R2 ← R2 + 32
i14:       MEM[R2-8] ← D8
i15:       JUMP loop IF R1 != R3
```

	Egész 1	Egész 2	Mem 1	Mem 2	FP 1	FP 2
1:			i1			
2:			i2			
3:			i3			
4:	i5		i4			
5:			i1		i6	
6:			i2		i7	
7:			i3		i8	
8:	i5		i4		i9	
9:			i1	i10	i6	
10:			i2	i11	i7	
11:	i13		i3	i12	i8	
12:	i5	i15	i4	i14	i9	
13:				i10	i6	
14:				i11	i7	
15:				i12	i8	
16:				i14	i9	
17:				i10		
18:				i11		
19:				i12		
20:				i14		

Feltűnő a hasonlóság a cikluskifejtéssel kapott kóddal, van azonban egy lényeges különbség, ami itt nem is látszik: a cikluskifejtés példájához képest itt R2 $2 \cdot 4 \cdot 8 = 64$-gyel hátrébb jár. Nézzük, mi is történik egy kiszemelt iterációban, egy utasításcsoport, mondjuk az i1-i6-i10 csoport végrehajtásakor. Amikor egy tömbelemet betöltjük a D1 regiszterbe (i1), azzal egyidőben az előző órajelciklusban betöltött tömbelemhez (szintén D1-ben

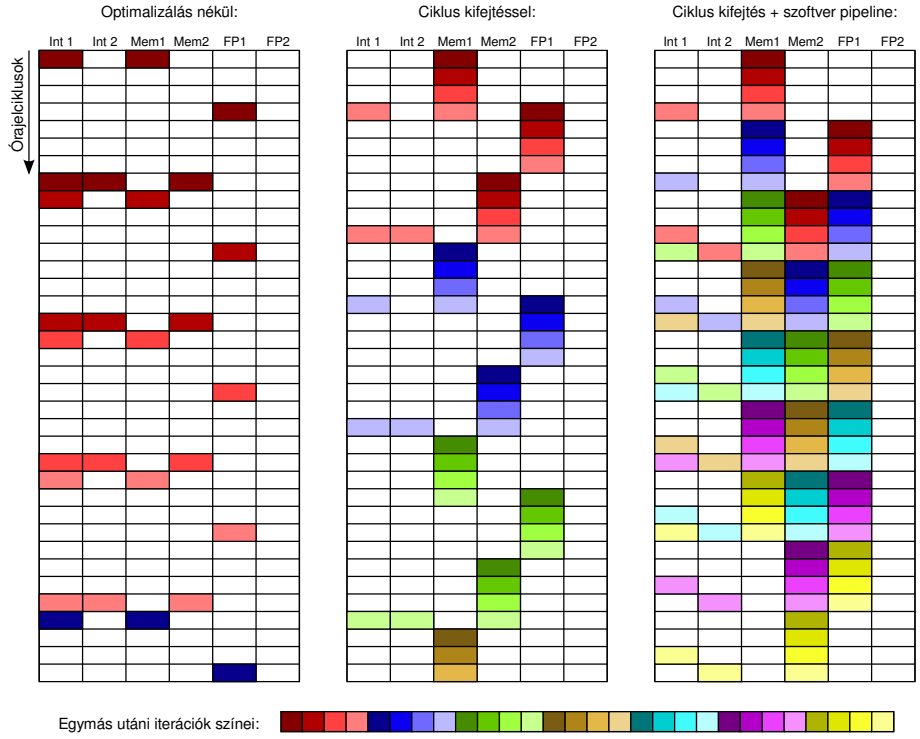
van) hozzáadhatjuk D0-t (i6), és mindezekkel egyidőben az előző kör összeadásának eredményét (D5-ben van, az előző körben számoltuk ki, a két körrel korábban betöltött tömbelemből) kiírhatjuk a memóriába (i10).

A táblázat az N=12 esetet ábrázolja (vagyis a cikluskifejtés után kapott for ciklus csak egy kört megy). Az 1.-8. sorok a szoftver pipeline feltöltését, a 13.-17. sorok a kiürítését végzik. Általános esetben a sárgával jelölt 9.-12. sorok annyiszor hajtódnak végre, ahány lépést a cikluskifejtés után kapott for ciklus megtesz. Látható, hogy a sárga részben minden egyes cilkusban történik egy lebegőpontos művelet, vagyis a két technika ötvözésével elérhetjük az ideális 1 iteráció/órajelciklus sebességet, feltéve persze hogy a ciklus elég sok lépést megtesz ahhoz, hogy a feltöltéshez és kiürüléshez szükséges idő amortizálódjon.

(Furcsának tűnhet a programkódban az i13. és i14. utasítások sorrendje: ezeket megcserélve kapnánk logikus programot. Azonban ezeket megcserélve a csoportosítást ábrázoló táblázat 12. sorában 3, az egész egységet igénylő utasítást is el kellene végezni egyszerre - az R1 növelését, az R2 növelését valamint a feltételes ugrást - amihez nincs elég erőforrás, hiszen a példában 2 egész egységet feltételeztünk. Ezért az R2 növelését egy órajelciklussal hamarabb végezzük el, és a fennmaradó Store művelethez tartozó címet ennek megfelelően számítjuk ki.)

Összegzés

A 15.13. ábra hatásosan szemlélteti, hogy milyen komoly teljesítményjavulás érhető el a cikluskifejtés és a szoftver pipeline alkalmazásával.



15.13. ábra. Ciklusoptimalizálás hatása a program futási sebességére

A 15.13. ábrán jól látható, hogy a ciklus kifejtéssel megszüntethető a csupa NOP-okat tartalmazó ciklusok száma. Ha a szoftver pipeline technikát is használjuk, akkor a kifejtett ciklus különböző színekkel jelzett iterációiban végzett műveletek mintegy egymásra tolódnak, ami a műveleti egységek jobb kihasználásához vezet.

Az out-of-order szuperskalár pipeline-nal rendelkező processzorok terjedésével ezeknek az optimalizációs technikáknak a jelentősége valamelyest csökkent, hiszen a processzor pl. a Tomasulo algoritmus segítségével ugyanezt a fajta utasításátrendezést hardveresen elvégzi. A hardverhez képest a ”kézi” (vagy a fordítóprogram által elvégzett) ciklusoptimalizálás előnye, hogy nagyobb ciklusok esetén is alkalmazható, a processzor ugyanis csak a (nem túl nagy számú) lehívott utasítások körében tud az utasítások sorrendjével játszani. A kódgenerálás során alkalmazott cikluskifejtés jótékony hatással van a procedurális egymásrahatások számára is, hiszen a kifejtett ciklus kevesebb iterációt tesz meg, kevesebb lesz a feltételes ugrás, az utasítások folyamatos lehívása kevesebbszer

akad meg. A modern fordítóprogramok ezeket az eljárásokat támogatják (gyakran manuálisan kell bekapcsolni, mint pl. a GNU C complier esetén a -funroll-loops opció megadásával).

15.6.2. Predikátumok használata

A procedurális egymásrahatás, vagyis a feltételes ugró utasítások a pipeline utasításfeldolgozás hatékonyságát rontják, hiszen az ugrási feltétel kiértékeléséig nem lehetünk biztosak benne, hogy honnan kell a további utasításokat letölteni. A probléma kezelésére általában elágazásbecslést (branch prediction) végeznek, a processzor megpróbálja megtippelni az ugrási feltétel kimenetelét és az ugrási címet. Az elágazásbecslő lehet, hogy téved, ekkor érvényteleníteni kell azokat az utasításokat, melyek végrehajtását tévedésből elkezdte, tehát az ezekre fecsérelt idő kárba veszett. Ha az elágazásbecslő el is találja az ugrás kimenetelét és címét, a becslési döntés meghozása általában 1-2 órajelciklust igényel (ezalatt a pipeline várakoztatja az utasítást és a mögötte jövőket), tehát a feltételes ugrás mindenképp egy teljesítményt rontó tényező.

A VLIW és EPIC processzorokban egy érdekes megoldás révén lehetőség van a feltételes ugró utasítások számának csökkentésére. Ezekben az architektúrákban az utasításokat el lehet látni egy extra operandussal, a predikátumregiszterrel. A predikátumregiszterek 1 bitesek. Ha az utasításhoz kapcsolódó predikátumregiszter értéke 0, akkor az utasítást a processzor nem hajtja végre. Példa:

```
R1 ← R2 + 32 IF P2
```

Ebben az esetben a P2 predikátumregiszter értékétől függ, hogy az összeadás végrehajtódik-e, vagy sem. A predikátumregisztereket összehasonlító műveletekkel lehet beállítani:

```
P2 ← R3 ≤ R5
```

Ha R3 kisebb vagy egyenlő, mint R5, P2 értéke 1 lesz, ellenkező esetben 0.

Elágazások feltételes ugrás nélkül

A predikátumok segítségével sokszor el tudjuk kerülni a feltételes ugrásokat. Nézzük az alábbi példát, amely ”c”-be írja ”a” és ”b” minimumát:

C kód:

```
if (a ≤ b)
    c = a;
else
    c = b;
```

Predikátumok nélkül:

```
        JUMP label IF R1>R2
        R3 ← R1
        JUMP end
label:  R3 ← R2
end:
```

Predikátumokkal:

```
P1,P2 ← R1 ≤ R2
R3 ← R1 IF P1
R3 ← R2 IF P2
```

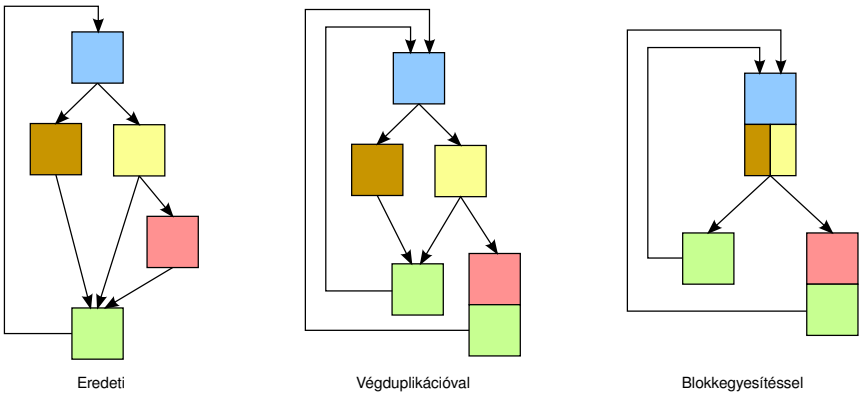
Predikátumokkal láthatóan nem volt szükség a JUMP utasításra. A példában, akárcsak a legtöbb VLIW/EPIC processzorban, a predikátumokkal operáló kódban az összehasonlítás két predikátumot fog beállítani: ha az összehasonlítás igazat ad, az elsőt 1-be, a másodikat 0-ba állítja, ellenkező esetben pedig fordítva. Tehát P1 a reláció teljesülését, P2 a nem teljesülését fogja jelenteni (P2 a P1 komplemense).

A predikátumokra alapozott megoldás nemcsak a feltételes ugrás kiküszöbölése miatt szerencsés. Vegyük észre, hogy a második és harmadik sor egymástól függetlenek, tehát egy utasításcsoportba tehetők. Ez általában is igaz: egy if-then-else C nyelvi szerkezet blokkjai független utasításokat tartalmaznak, tehát a predikátumok segítségével a teljesülő és a nem teljesülő ág utasításait egy csoportba lehet fogni. Ilyenkor a processzor lehívja az utasításcsoportot, amely mindkét ág utasításait tartalmazza, de a csoportból csak azokat hajtja végre, amelyek a ciklusfeltétel teljesülésének megfelelő predikátummal vannak ellátva.

Blokkegyesítés

Egy VLIW/EPIC fordítóprogram a processzor erőforrásainak minél jobb kihasználása érdekében megváltoztathatja az utasítások sorrendjét. A független utasítások csoportokba rendezése során azonban a fordító mozgásterét nagy mértékben korlátozzák a feltételes ugró utasítások, hiszen fordítási időben a végrehajtási történet, így az

elágazási feltétel is ismeretlen. Olyan utasításokat nem lehet felcserélni, illetve nem kerülhetnek egy csoportba, melyek között az eredeti programban feltételes ugró utasítás van, hiszen az elágazás kimenetele futás közben derül ki. A fordítóprogram tulajdonképpen az ugró utasítások által határolt utasítássorozatokon belül (*egyszerű blokkokban*, basic block) válogathat független utasításokat keresve. Az egyazon egyszerű blokkba tartozó utasítások feldolgozását nem zavarja procedurális egymásrahatás sem. Az előző fejezetben látott predikátumokra alapozott technikát felhasználhatjuk arra, hogy a programot a feltételes ugró utasítások számát csökkentve kevesebb, de nagyobb méretű egyszerű blokkra bontsuk, ami a pipeline feldolgozás számára előnyösebb.



15.14. ábra. Ugrások számának csökkentése blokkegyesítéssel

A 15.14. ábra példájában a színes négyzetek az egyszerű blokkok, a nyilak pedig a vezérlésátadást reprezentálják. Az eredeti programban először a zöld színnel jelzett egyszerű blokkot duplikáljuk, az egyik példány csak a piros, a másik pedig a sárga és barna blokkokból érhető el. Ezután jön a blokkegyesítés: a kék blokk után a vezérlés vagy a sárga, vagy a barna blokkra kerül - ezt az elágazást a már látott módon, predikátumokra támaszkodva megszüntetjük, a sárga és barna blokkokból így egy egyesített blokkot kapunk. Az egyesített blokkban a sárga és a barna blokkhoz tartozó utasítások függetlenek, egy utasításcsoportba helyezhetők. A kapott eredmény több szempontból is előnyös: egyrészt kevesebb az ugrás, így kevesebb a procedurális egymásrahatás is. Másrészt a blokkegyesítés révén az eredetinél nagyobb blokkokat kapunk, így a fordítóprogramnak a több utasítással nagyobb játéktere lesz az optimális utasítás sorrend és az optimális utasítás csoportosítás kialakításához (nagyobb "merítésből" könnyebben talál függetlenül végrehajtható utasításokat).

15.6.3. Spekulatív betöltés

A fordítóprogramok által alkalmazott gyakori technika (nem csak VLIW/EPIC környezetben), hogy az egymással RAW egymásrahatásban álló utasításokat a fordító egymástól a lehető legjobban eltávolítja, hogy a korábbi eredménye már készen álljon, mire a tőle függő későbbi utasítás végrehajtása megkezdődik. Az utasítások eltávolításának gátat szabnak a feltételes ugrások. Egy egyszerű blokkon belül bizonyos fokig lehet ugyan tologatni az utasításokat, de az egyszerű blokkot határoló feltételes ugrások elé és mögé nem tolhatunk utasítást, hiszen fordítási időben nem tudjuk az ugrás kimenetelét előre megjósolni. Tehát egyszerű blokkon belül célszerű lehet, egyszerű blokkok között pedig nem tűnik ésszerűnek az utasítások mozgatása.

spekulatív betöltés nélkül:

```
        JUMP label IF R8==0
        R1 ← MEM [R0]
        R2 ← R1 * R1
        JUMP end
label:  ...
end:
```

spekulatív betöltéssel:

```
        ...
        R1 ← MEM [R0] / spekulatív
        ...
        JUMP label IF R8==0
        spekulatív Load véglegesítés
        R2 ← R1 * R1
        JUMP end
label:  ...
end:
```

A példában az eredeti, bal oldali kóddal az a probléma, hogy a Load és az aritmetikai művelet RAW egymásra-hatásban állnak egymással, a szorzás nem kezdhető el a Load eredménye nélkül, a pipeline egy vagy több ciklust haszontalan várakozással kénytelen tölteni. A fordító nem tudja a Load-ot az egyszerű blokkon belül feljebb tolni, hiszen pont az az egyszerű blokk első utasítása, ezért azt a feltételes ugrás elé mozgatja. Ekkor a Load spekulatívvá válik, az eredményét nem kell rögtön az R1-be írni, csak akkor, ha az elágazás tényleg úgy alakul, hogy a JUMP feltétele hamis. Ekkor az utasítás eredeti helyén véglegesíthető az utasítás eredménye (R1-be íródik a beolvasott memóriatartalom). Mivel a Load jó néhány utasítással megelőzi a szorzást, a szorzás haladéktalanul elkezdhető lesz, mire odaér a végrehajtás. Ha a feltételes ugrás bekövetkezik (a fordító spekulációja nem jön be), akkor a Load eredménye eldobható.

16. fejezet

Elágazásbecslés

16.1. Az elágazásbecslés szerepe

Ha nem lennének ugró utasítások, az utasításlehívó egység lenne a processzor egyik legegyszerűbb alkotóeleme, hiszen mindig csak az utasításszámláló növelése és a soron következő utasítás beolvasása lenne a feladata.

Azonban minden utasításkészlet architektúrában vannak ugró utasítások, melyek az utasításfolyam szekvenciális viselkedését megtörik, és ezek feldolgozása az utasításlehívó egység számára problémát jelent. Az utasításlehívónak ugyanis azonnal el kell döntenie, hogy a program merre folytatódik, máskülönben nem tudja végrehajtandó utasításokkal ellátni a pipeline további fázisait. Sokszor ezt a döntést az időkorlátok miatt olyan gyorsan kell meghoznia, hogy csak egy becslésre tud hagyatkozni. Ha rosszul tippel, az időközben tévedésből behozott utasítások végrehajtása feleslegesnek bizonyult, a processzor szükségtelen utasításokra pazarolta erőforrásait.

Mielőtt rátérnénk, hogy hogyan lehet nagy találati pontossággal megbecsülni a program folytatásának irányát, áttekintjük azokat az utasításokat, melyek az utasítások egymásutániságát megszakíthatják.

- A *feltétel nélküli ugró utasítások* a magas szintű nyelvek ”goto” utasításának felelnek meg. Egyetlen bemenőoperandusuk az ugrási cím, azaz hogy merre folytatódik tovább a program, mi lesz a következő utasítás. A feltétel nélküli ugrás lehet *direkt*, ha az ugrási cím egy konstans, pl. `JUMP -28`, illetve lehet *indirekt*, ha az ugrási cím egy regiszter értéke, pl. `JUMP R1`.

  Bizonyos architektúrákban az ugrási címet az utasításszámlálóhoz képest értik (relatív), másokban ez egy abszolút memóriacím. Az x86 mindkét fajta viselkedésre tartalmaz utasítást.

- A *feltételes ugró utasítások* az ugrási cím mellett egy ugrási feltétellel is rendelkeznek. Az ugrás csak akkor következik be, ha az ugrási feltétel teljesül. A feltétel megadása architektúrafüggő, lehet feltétel kóddal vagy feltétel regiszterrel megadott, illetve lehet az utasítás része is (”összehasonlít és ugrik”, lásd 3.2.4. fejezet). Ebben a fejezetben az utóbbi formát használjuk. Ebben az esetben is beszélhetünk direkt, pl. `JUMP -28 IF R2>0`, illetve indirekt esetekről, pl. `JUMP R1 IF R2>0`.

  A programban feltételes ugrással járnak az if-then-else szerkezetek, illetve a ciklusokban a ciklusfeltétel kiértékelése is.

- A *szubrutin hívások*, illetve a *szubrutinból való visszatérés* is a szekvenciális utasítás-végrehajtás megszakadásával jár. A szubrutin hívás is lehet direkt, pl. `CALL -28` vagy *indirekt*, pl. `CALL R1`.

  A szubrutinhívás formája szemmel láthatóan nagyban hasonlít a feltétel nélküli ugrásokhoz, a különbség csak annyi, hogy szubrutinhíváskor a visszatérési cím a verembe kerül. Vannak utasításkészlet architektúrák, melyekben nincs is külön szubrutin hívás, vagy éppen nem stack-el oldják meg a visszatérési cím átadását. Ebben a fejezetben stack alapú függvényhívásokat veszünk alapul.

  A szubrutinból való visszatérés alakja: `RET`, amely szintén feltétel nélküli ugrás, de az ugrási cím ez esetben a veremből származik.

A program folytatásának kiderítése érdekében ezekkel az utasításokkal a lehívás után a következő műveleteket kell elvégezni:

- *Az ugrási cím kiszámolása.* Direkt ugrás esetén ez csak annyit jelent, hogy az ugrási címet, ami egy konstans, hozzá kell adni az utasításszámlálóhoz. Indirekt ugrás esetén rosszabb a helyzet, hiszen az ugrási címet egy regiszter tartalmazza. Ennek értékét vagy a regisztertárolóból kell kiolvasni, vagy, ha éppen van olyan végrehajtás alatt álló utasítás, amelynek ez az eredményoperandusa, akkor annak végrehajtását meg kell várni. Szubrutinból való visszatérés esetén az ugrási cím ráadásul egy lassú memóriaművelet eredménye (hiszen a stack a memóriában van).

- *Az ugrási feltétel kiértékelése.* Akár az indirekt ugrásoknál, itt is gondot jelent, hogy az ugrási feltételben szerepet játszó regisztert vagy a regisztertárolóból kell venni, vagy meg kell várni az azt előállító utasítás befejeződését.

Annak érdekében, hogy a processzor futószalagja lehetőleg egyetlen ciklusnyi ideig se álljon meg, az ugrási címet és az ugrási feltételt nem számolják, hanem becslik. Természetesen a pipeline feldolgozás egy későbbi fázisában sor kerül ezek pontos kiszámolására is. Ekkor az utasításlehívó ellenőrzi, hogy jól tippelt-e, és ha kiderül, hogy nem, akkor az időközben elkezdett, rossz ágról származó utasításokat érvényteleníti.

Ennek a becslésnek a pontossága egy mély és széles pipeline esetén még kritikusabb, hiszen egy ilyen processzorban rossz döntés esetén sokkal több szükségtelen utasítás végrehajtása kezdődhet el, mire a tévedés kiderül. A nagyságrendek szemléltetése kedvéért számoljuk ki, hogy alakul egy átlagos program futása egy ma átlagosnak mondható processzoron tökéletes és kevésbé tökéletes elágazásbecslést feltételezve:

- Vegyünk egy 4-utas szuperskalár processzort, melyben az elágazásbecslés sikeressége a feltételes ugró utasítás pipeline-ba lépésétől számítva 17 ciklus múlva derül ki, azaz rossz döntés esetén 17 ciklus munkája vész kárba (ezek az Intel Core i7 Nehalem valós paraméterei).

- Tegyük fel, hogy a programban minden negyedik utasítás feltételes ugrás (ez általános programok esetén reális szám).

- Tegyük fel, hogy az elágazásbecslő 67%-os találati aránnyal dolgozik.

- Tegyük fel, hogy az utasítások végrehajtása során az elágazások miatti procedurális egymásrahatáson túl más egymásrahatás nem lép fel.

Számoljuk ki, átlagosan hány ciklus szükséges egy utasítás végrehajtásához! Mivel 4-utas szuperskalár a processzor, ciklusonként 4 utasítás fejeződik be, vagyis átlagosan 0.25 ciklusonként távozik egy utasítás. Az utasítások negyede ugrás, melyek 0.33 valószínűséggel vonnak maguk után egy 17 ciklusidőnyi extra késleltetést, azaz: $0.25 + 0.25 \cdot 0.33 \cdot 17 = 1.65$ ciklus/utasítás. Tökéletes elágazásbecslés esetén az utasítások távozási ideje: 0.25 ciklus/utasítás, vagyis nem kevesebb, mint $1.65/0.25 = 6.6$-szor jobb futási időket kapnánk, ha tökéletes lenne a processzor elágazásbecslője! Az elágazásbecslésen tehát nagyon is sok múlik.

A rossz döntés okozta kiesett ciklusok számát foglalja össze az alábbi táblázat néhány, a maga idejében ill. ma elterjedt processzor esetében:

	Kiesett ciklusok száma:
Intel Pentium I MMX::	4-5
Intel Pentium 4:	átlagosan 45!
Intel Core2:	15
Intel Core i7 Nehalem:	17
Intel Core i7 Sandy Bridge:	15
Intel Atom:	13
AMD K8 és K10:	12
Via Nano:	16
ARM Cortex A8:	13
ARM Cortex A9:	8

16.2. Ugrási feltétel kimenetelének becslése

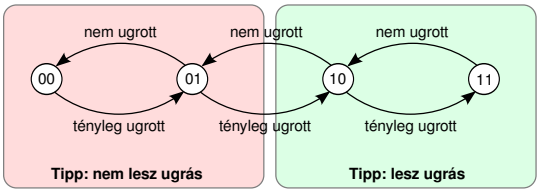
A feltételes ugrások kimenetelét akkor lehet nagy találati pontossággal becsülni, ha azok szabályosan viselkednek. Szerencsére a statisztikák szerint az általános célú programokban az ugró utasítások túlnyomó része valóban szabályosan, kiszámíthatóan viselkedik. Nézzük az alábbi kimenetel-sorozatokat (minden sor egy feltételes ugró utasítás egymás utáni kimeneteleit reprezentálja, az 1 jelöli a megvalósult, a 0 a nem megvalósult ugrásokat):

- 1111111?
- 11111101101111011111?
- 11001100110011?
- 11111111111000000000000?

Az emberi szem könnyen kitalálja a legvalószínűbbnek tűnő folytatást, de a döntést algoritmizálni, ráadásul az egyszerűségre és gyorsaságra vonatkozó rendkívül szoros követelmények szem előtt tartásával, közel sem egyszerű.

16.2.1. Egyszerű állapotgép alapú megoldások

A legegyszerűbb megoldás, ha a processzor minden feltételes ugró utasítás viselkedését egy állapotgép segítségével követi nyomon. Az állapottér egy része az ”ugrik”, a másik része a ”nem ugrik” döntésnek felel meg. Az egyes állapotok azt jelentik, hogy a processzor mennyire biztos benne, hogy ugrás lesz. Menet közben folyamatosan követi az ugrások *tényleges* kimenetelét, és ez alapján frissíti az utasítás állapotot, ha a tényleges kimenetel egyezik azzal, amit tippelt, akkor a bizonyosabb döntéshez tartozó, ellenkező esetben a bizonytalanabb döntéshez tartozó irányba.



16.1. ábra. Állapotgép elágazásbecsléshez

A 16.1. ábrán látható állapotgép tulajdonképpen egy 2 bites *telítődő* számlálónak felel meg. Minden egyes feltételes ugró utasításhoz tartozik egy ilyen számláló. Ha az ugrás ténylegesen bekövetkezik, eggyel növeljük, ha ténylegesen nem következik be, egyel csökkentjük a számláló értékét. Ha az utasítás megjelenik a pipeline-ban, és predikcióra van szükség, csak rá kell nézni a hozzá tartozó számlálóra: ha az értéke 0 vagy 1, akkor a jóslás az lesz, hogy az ugrás nem következik be, 2 vagy 3 esetén pedig a bekövetkezésre tippelünk (bináris reprezentáció esetén a számláló felső bitje lesz a predikció).

Természetesen nem csak 2 bites számlálót lehet használni elágazásbecslésre. Az 1 bites számláló esetén a predikció mindig az ugró utasítás előző kimenetele lesz. Az elágazásbecslők pontosságát szimulációs vizsgálatokkal szokták kiértékelni: vesznek egy sor gyakran használt algoritmust, lefuttatják egy processzor szimulátoron, és a szimuláció során könyvelik a prediktor találatainak és az ugrási utasítások számának arányát. Egy konkrét tanulmány (1993-ból, [12]) az 1 bites állapotgépre 89%-os, a 2 bitesre pedig 93%-os pontosságot mért. Azt is kimutatták, hogy 3 bites számlálóval csak egészen minimálisan javul a predikció pontossága, míg 3-nál több bites számláló gyakorlatilag nem hoz további javulást. A 2 bites számlálók használata ezek alapján jó választásnak tűnik.

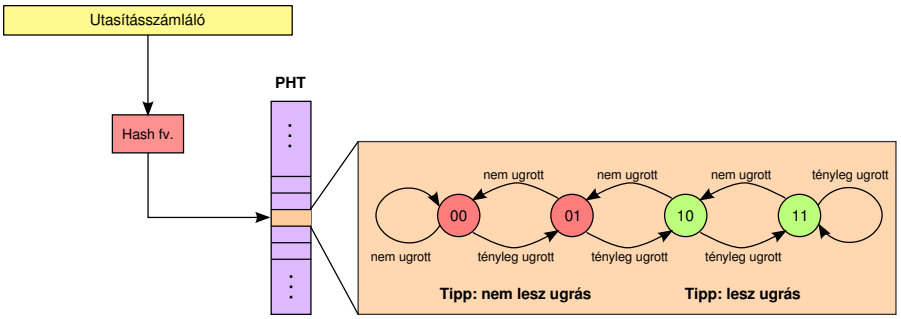
Nézzünk egy konkrét példát az állapotgép alapú elágazásbecslés hatékonyságára:

```
for (i=0; i<m; i++) {
    for (j=0; j<n; j++) {
        ...
        ...
    }
}
```

A for ciklusról tudjuk, hogy minden új iteráció előtt ellenőrzi a ciklusfeltételt, és ha az nem teljesül, elhagyja a ciklust. Egy n kört futó for ciklusban így összesen $n+1$-szer kell kiértékelni ezt a feltételes ugrást, amely n-szer teljesül, végül 1-szer, a végén, amikor elhagyjuk a ciklust, nem teljesül. A példában a külső ciklus m-szer hajtja végre a belsőt. Nézzük meg, hogy a belső ciklus ciklusfeltételéhez tartozó feltételes ugró utasítás predikciója mennyire hatékony. A belső ciklus teljes lefuttatása az 1 bites számlálóval 2 téves predikciót jelent: egyet a ciklus végén, amikor az ugrások sorozatos teljesülése után a végén egyszer nem történik ugrás (mert a ciklusból való kilépés miatt a futás a következő, ciklus utáni utasítással folytatódik). A másik téves predikció pedig akkor történik meg, amikor a belső ciklust a külső újra lefuttatja: a ciklusfeltétel első kiértékelésekor az 1 bites séma az előző kimenetelt adja, az pedig pont a ciklus elhagyása volt. Nagyságrendileg tehát az összes $\approx n \cdot m$ feltételes ugrásból $\approx 2m$ predikciója téves. Ha 2 bites számlálót alkalmazunk, jobb eredményt kapunk. Ha n elég nagy, a kérdéses feltételes ugró utasításhoz tartozó számláló hamar 11-be telítődik, a prediktor megtanulja, hogy ez az ugrás meg szokott valósulni. A ciklus végén ugyanúgy téved, mint az 1 bites, de ez az egy tévedés nem változtatja meg homlokegyenest a jövőbeli predikciókat: a prediktor ugyan egy kicsit elbizonytalanodik, a számláló értéke lecsökken 11-ről 10-ra, de még mindig az ugrás bekövetkezésére fog tippelni, amikor a külső ciklus a következő körben a belsőt újra lefuttatja. Nagyságrendileg tehát az összes $\approx n \cdot m$ feltételes ugrásból most csak $\approx m$ predikciója téves.

Ennek az elágazásbecslési módszernek nagy előnye, hogy kicsi az adminisztrációs igénye: csak 2 bitet kell ugró utasításonként nyilvántartani, amit vagy növelni, vagy csökkenteni kell az ugrási feltétel kiértékelése után, és melynek legfelső bitje egyben az ugrás becsült kimenetele is. Az azonban kérdés, hogy ezt a 2 bitet hol tároljuk, és hogy honnan tudjuk, hogy a pipeline-ba belépő feltételes ugró utasításhoz melyik 2 bit tartozik. Erre vonatkozólag több lehetőség is van:

- Alkalmazhatjuk a cache-nél megismert trükköket. Az ugró utasítások címe a cache tag, a számláló 2 bitje pedig az adat. A belépő ugró utasítás címe alapján tartalom szerinti címzéssel keressük meg a cache-ben a hozzá rendelt számlálót. Ezt a cache-t a szokásos cache-nél megismert eszközökkel lehet szervezni és menedzselni: több utas szervezés, LRU cserestratégia, stb. Egy gond van ezzel a megoldással: nem éri meg! 2 hasznos bit kedvéért 32 bites tag-eket tárolni és 32 bites tartalom szerinti címzést implementálni nem ésszerű.
- Az utasításcache-ben a cache blokkok-ban fent lehet tartani egy kis területet az adott blokkban lévő ugró utasítások számlálóinak. Ezt a megoldást előszeretettel alkalmazza az AMD.
- Tárolhatjuk a 2 bites számlálókat egy külön táblázatban. Ezt a táblázatot *minta előzmény tábl*ának hívjuk (PHT, pattern history table). Mivel ilyen kicsi hely kell egy számlálónak, a PHT-nak nagyon sok eleme lehet. Nem ritka a több ezres méret sem: ennyi ugró utasítás adatait tudja egyszerre nyomon követni a processzor. Azt, hogy a pipeline-ba belépő ugró utasításhoz melyik számláló tartozik, az utasítás címének alsó bitjei, vagy a cím egy hash függvénye dönti el. Természetesen lehet ütközés (több utasításhoz is ugyanazt a számlálót rendeljük), de ennek valószínűsége a PHT nagy mérete miatt kicsi, és ha mégis bekövetkezik, akkor sem történik hiba, legfeljebb pár rosszul sikerült predikció (lásd 16.2. ábra, nem véletlen a hasonlóság a cache memóriánál megismert Rivers' algoritmussal, 10.9. ábra!).



16.2. ábra. Elágazásbecslés ugrási előzmény táblával

Ilyen megoldást használ pl. az Intel Pentium 1 processzor is, bár a 2 bites számlálókat nem PHT-ban tárolja, hanem a becsült ugrási címekkel együtt (lásd később). Érdekesség, hogy ebben a processzorban a számláló logikát

hibásan megvalósították meg, mert az ugrás tényleges bekövetkezése esetén a 00 állapotból nem a 01-be, hanem az 11-be ugrott az állapotgép.

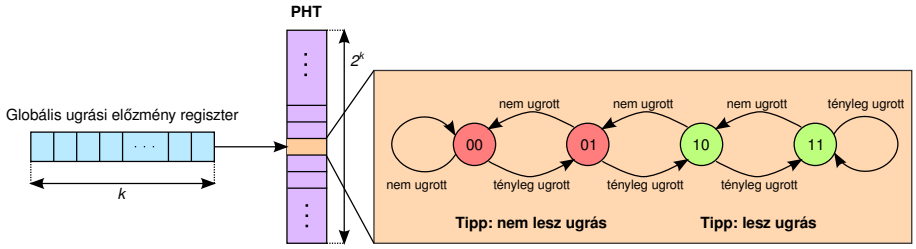
16.2.2. Korrelácó figyelembe vétele

Az állapotgép alapú megoldások sok esetben hatékonyak, de nem veszik figyelembe, hogy az ugró utasítások kimenetele gyakran függ a korábbi ugró utasítások kimenetelétől. Például ragadjunk ki egy részletet az *eqntott* nevű, bool kifejezésekből PLA programozáshoz igazságtáblát építő programból:

```
if (a==2)
    a = 0;
if (b==2)
    b = 0;
if (a!=b) {
    ...
    ...
}
```

Ebben a programrészletben a harmadik if ugrási feltételének értékét egyes esetekben az első két if ugrási feltétele meghatározza. Ha az első két if feltétele igaz volt, akkor a harmadik if feltétele nem lesz igaz. Az egymás utáni feltételes ugró utasítások ezt a fajta összefüggőségét (korrelációját) jó lenne az elágazásbecslés során kihasználni.

A korreláció kihasználásának eszköze a *globális ugrási előzmény regiszter* (global branch history register). Ez lényegében egy shift regiszter, ha a processzor egy ugró utasítást kiértékel, annak kimenetelét (0-át vagy 1-et) ebbe a regiszterbe shifteli. Az addigi legmagasabb helyiértékű bit természetesen ekkor elvész. Egy k bites globális ugrási előzmény regiszter az előző k ugrás kimenetelét tárolja. A predikció most is a PHT alapján történik, de a PHT most nem az egyes ugró utasítások kimenetelének becslését támogatja, mint eddig, hanem az egyes ugrási előzményminták esetén valószínű kimenetelek becslését segíti, így a PHT-t nem az ugró utasítás címe, hanem a globális ugrási előzmény regiszter fogja indexelni (16.3. ábra).



16.3. ábra. PHT globális ugrási előzmény regiszterrel

Pl. ha a globális ugrási előzményregiszter értéke épp ”1100110011”, és belép a pipeline-ba egy ugró utasítás, akkor a PHT 1100110011-edik (decimálisan a 819-edik) bejegyzésében ülő 2 bites telítődő számláló tartalma dönti el, hogy a belépő ugró utasítás kimenetelét hogyan becsüljük. A 819-edik bejegyzés segít eldönteni, hogy a ”1100110011” előzmények után egy ugrás meg szokott-e valósulni, vagy sem. Valószínűleg a 819-es bejegyzésben a ”00” értéket látjuk majd, ami azt jelenti, hogy nagyon valószínű, hogy a ”1100110011” után egy 0 következik. Ha ilyen előzményekkel az ugrás mégis megvalósul, akkor a 819-es bejegyzést 00-ról 01-re módosítjuk, tehát elkönyveljük, hogy ilyen előzményekkel még mindig 0-ra tippelünk, de már nagyobb bizonytalansággal.

Ha k bites globális ugrási előzmény regisztert használunk, akkor 2^k méretű PHT esetén a regiszter közvetlenül indexelheti a PHT-t, hash függvényre nincs szükség.

16.2.3. Kétszintű prediktorok

Az egyszerű állapotgép alapú megoldások kizárólag az adott ugró utasításra lokális döntéseket tudnak hozni. A korrelációt is figyelembe vevő, globális ugrási előzmény regiszterre alapozott megoldás pedig csak a globális előzmények alapján dönt, nem veszi figyelembe az egyes ugró utasítások esetleg eltérő, a globális előzmények

alapján esetleg nem jól megjósolható viselkedését. A figyelembe vett lokális/globális információk szerint ez a két eljárás a lehetőségek két szélsőséges esete. Több lehetőség is van a lokális és globális ugrási információk kombinálására az elágazásbecslés találati pontosságának növelése érdekében.

A kétszintű prediktorok az alábbi két fajta információt használják a döntéshozásban:

- ugrási előzmény regisztert, melyben a sorakozó bitek az ugró utasítások egymás utáni kimeneteleit jelentik,
- minta előzmény táblát, melynek egy eleme (pl. egy telítődő számláló, lásd fentebb) megadja, hogy az eddigi viselkedés alapján várható-e ugrás.

Mindkét féle információ lehet lokális ill. globális:

- *Globális ugrási előzmény regiszter* esetén a processzor egy darab ugrási előzmény regisztert tart karban, melybe minden egyes ugró utasítás kimenetelét beshifteli.
- *Lokális ugrási előzmény regiszter* esetén annyi ugrási előzmény regiszter van, ahány ugró utasítás. Minden egyes ugró utasítás múltbéli kimeneteleit külön, saját regiszterben tároljuk. Azt a táblázatot, melyben az ugró utasítások előzményeit tároljuk, *ugrási előzmény tábl*ának hívjuk (BHT, branch history table). A BHT minden bejegyzése egy ugró utasítás előzményregisztere, melynek bitjei az ugró utasítás korábbi kimenetelei.
- *Globális PHT* esetén a rendszerben egyetlen táblázat áll rendelkezésre, amely az ugrások várható kimenetelét tartalmazza
- *Lokális PHT* esetén minden utasításnak külön PHT-je van, amiben az ugrás várható kimenetelei vannak különféle körülmények között.

Az esetleg nem egyértelmű megfogalmazás (különösen a globális/lokális PHT-vel kapcsolatban) értelmet nyer, ha végigvesszük az összes lehetséges kombinációt:

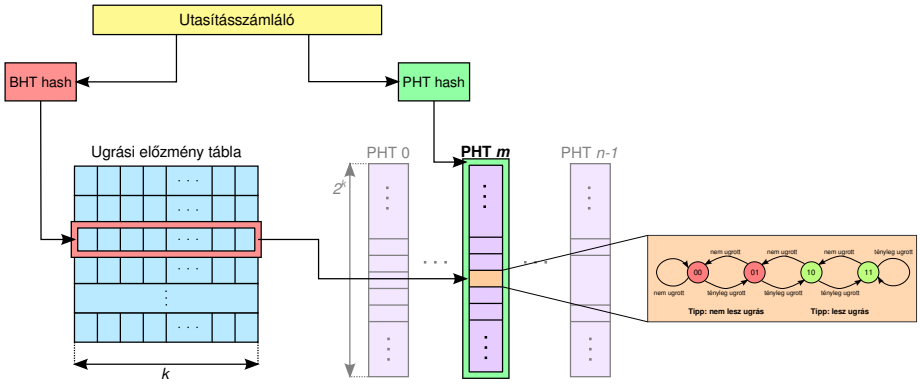
1. **Globális ugrási előzmény regiszter globális PHT-vel**: ez az eset felel meg a 16.2.2 fejezetnek, ill. a 16.3. ábrának. Egy darab ugrási előzmény regiszter van, mely megindexeli az egy darab globális PHT-t, hogy kiderüljön, hogy az adott múltbeli ugrási kimenetelek mellett várható-e ugrás, vagy sem.

2. **Lokális ugrási előzmény regiszter lokális PHT-vel**: minden ugró utasításhoz külön tartjuk nyilván a múltbeli kimeneteleit egy shift regiszter segítségével. Minden ugró utasításnak saját PHT-je van, melyben tároljuk, hogy az adott előzmények után várható-e ugrás.
   Példa: legyen egy ugró utasítás lokális előzmény regisztere ”0101”. (Ez azt jelenti, hogy az előző és a hárommal előző ugrás megvalósult, a kettővel és néggyel előző nem.) Tegyük fel, hogy ez az ugró utasítás megjelenik a pipeline-ban, és el kell végeznünk az elágazásbecslést. Ránézünk a lokális előzmény regiszterére (melyet a BHT tárol - hogy hogyan, azt később tárgyaljuk), ami megadja, hogy az utasítás saját külön bejáratú PHT-jének a 0101-edik, vagyis decimálisan az 5-ödik bejegyzését kell nézni. Az 5-ödik bejegyzésben szerepel, hogy ez az ugró utasítás ”0101” előzményekkel szokott-e ugrani, vagy sem. Ha a PHT 2 bites telítődő számlálókat tartalmaz, és itt az szerepel, hogy ”00” vagy ”01”, akkor arra tippelünk, hogy nem lesz ugrás, ”10” vagy ”11” esetén pedig fordítva. Később pedig, amikor megtörténik az ugrási feltétel tényleges kiértékelése, a PHT megfelelő elemét frissíteni kell. (Lásd 16.4. ábra. Minden ugró utasításnak természetesen nincs PHT-je és BHT bejegyzése, az ugró utasítást azonosító utasításszámlálót hash függvények rendelik hozzá a korlátozott számú PHT-hez ill. BHT bejegyzéshez.)

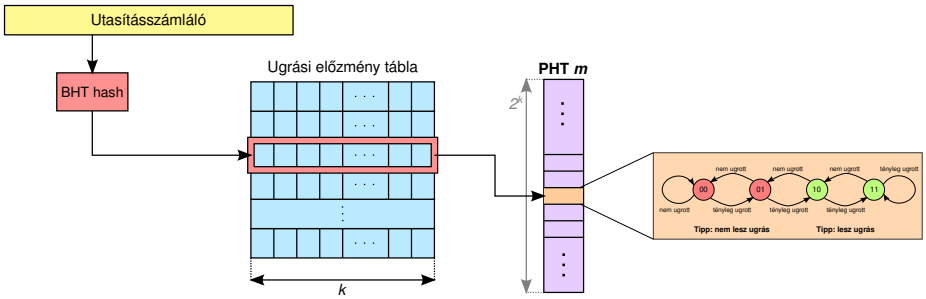
3. **Lokális ugrási előzmény regiszter globális PHT-vel**: minden ugró utasításhoz külön tartjuk nyilván a múltbeli kimeneteleit, de csak egy, mindenkire nézve közös PHT áll rendelkezésre (16.5. ábra).
   Példa: Mint az előbb, tegyük fel, hogy egy ugró utasítás lokális előzmény regisztere ”0101”. Ekkor a közös PHT-ben kell megnézni, hogy ”úgy általában”, a ”0101” mintát követő előzmények után teljesülni szoktak-e az ugró utasítások.

4. **Globális ugrási előzmény tábla lokális PHT-vel**: Egyetlen közös előzmény tábla van, de PHT-ja minden ugró utasításnak saját van. Az utasítás specifikus PHT-ben kell megnézni, hogy az adott globális előzmények után lesz-e ugrás, vagy sem (16.6. ábra).

A figyelembe vehető előzmények számát (a múltbéli ugrások tárolt kimeneteleinek számát) meghatározza, hogy az előzmény regiszter és a PHT lokális-e, vagy globális. Ha mindkettő globális, akkor sok múltbeli kimenetelt, tehát hosszú idejű korrelációt is figyelembe lehet venni, $k = 14$ hosszú globális előzmény regiszter mellett 2^{14}



16.4. ábra. Lokális ugrási előzmény regiszter lokális PHT-vel



16.5. ábra. Lokális ugrási előzmény regiszter globális PHT-vel

méretű PHT-ra van szükség, ami 2 bites számlálókat véve alapul még csak 4 KB helyet foglal ($2^{14} \cdot 2/8 = 4096$ byte). Ha mindkettő lokális, és egyszerre 1024 ugró utasítás becsléséhez akarunk adatokat tárolni (tipikus szám), akkor ugyanez a helyfoglalás csak $k = 4$ hosszú előzmény regiszterrel jön ki ($1024 \cdot 2^4 \cdot 2/8$ byte, de tárolni kell a lokális előzmény regisztereket is, ami ezen felül még $1024 \cdot 4/8$ byte).

A ”gshare” prediktor

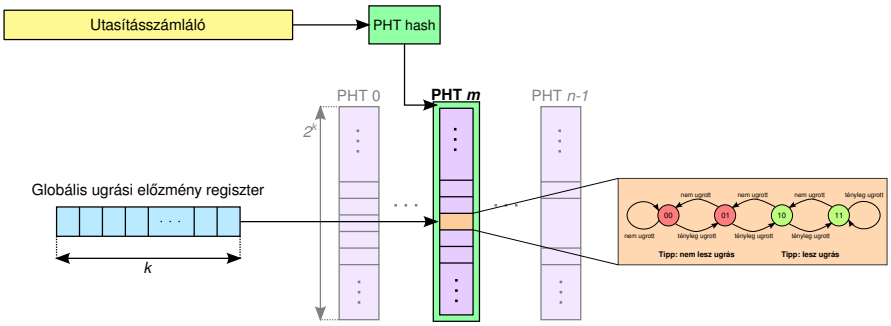
Az imént sorra vetteken kívül más módja is van a lokális és globális információ együttes figyelembevételének. A 16.2.1 fejezetben megismert, egyszerű állapotgép alapú eljárás (ami egy korrelációt figyelembe nem vevő lokális eljárás) az utasításszámlálóval indexelte a PHT-t. A 16.2.2 fejezetben megismert, globális ugrási előzményt figyelembe vevő eljárás a globális ugrási előzmény regiszterrel indexelte a PHT-t. Ahhoz, hogy az ugró utasításra lokális és a környezetből eredő globális információkat is figyelembe tudjunk venni egyidejűleg, az utasításszámláló és a globális ugrási előzmény regiszter egy függvényével kell megindexelni a PHT-t. A *gshare prediktor* a kettő XOR függvényét alkalmazza (16.7. ábra).

A gshare prediktor, a rendkívül egyszerű megvalósíthatósága és a kitűnő becslési pontossága miatt nagy népszerűségnek örvend. Ezt az eljárást alkalmazza a Sun UltraSPARC III, IBM PowerPC 4, XBox 360 ”Xenon” CPU, AMD Athlon K6.

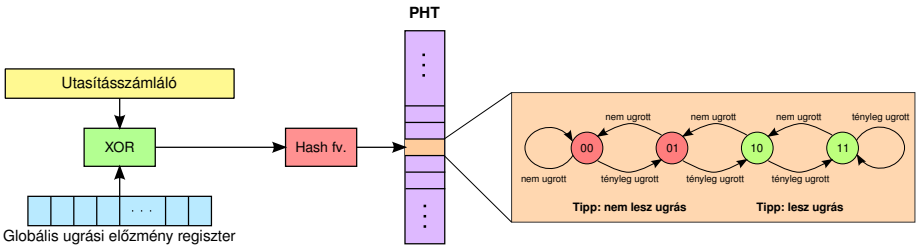
Hasonló elágazásbecslés található az ARM Cortex A8 processzorokban is, 10 bites globális előzmény regiszterrel, amely kiegészítve az ugró utasítás címének utolsó 4 bitjével egy 4096 elemű 2 bites telítődő számlálókból álló PHT-t indexel.

16.2.4. Kombinált megoldások

Ahogy a korábbi fejezetek rámutattak, vannak feltételes ugró utasítások, melyek viselkedésének előrejelzésére a lokális, mások viselkedésének előrejelzésére a globális információra alapozott elágazásbecslés sikeresebb. Célszerűnek tűnhet több elágazásbecslési eljárást is beépíteni a processzorba, és egy kiválasztó logikára bízni annak eldöntését, hogy egy adott ugró utasítás elágazásbecsléséhez mikor melyiket alkalmazzuk.



16.6. ábra. Globális ugrási előzmény regiszter lokális PHT-vel



16.7. ábra. A gshare prediktor

Az ”agree” prediktor

Az agree prediktor arra a felismerésre alapul, hogy az ugró utasítások általában valamilyen irányba ”húznak”: vagy többnyire ugranak, vagy többnyire nem ugranak. Az agree prediktorban két elágazásbecslő eljárást kombinálnak:

- Egy teljesen lokális, egyszerű állapotgépre alapozott algoritmus becsüli meg, hogy az ugró utasítás melyik irányba ”húz” (bias),
- és egy globális ugrási előzmény regisztert alkalmaznak globális PHT-vel annak megbecslésére, hogy az elágazás vajon a tipikus irányba fog-e megvalósulni (amerre általában ”húz”), vagy épp ellenkezőleg (agree or disagree with the bias).

A végső döntést a két eljárás eredményének kombinációja adja, az alábbi igazságtáblának megfelelően:

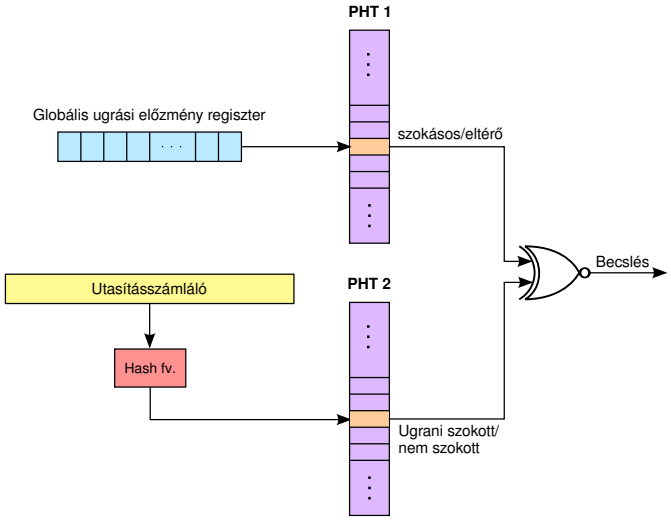
Lokális becslő:	Globális becslő:	Becslés végeredménye:
1	1	1
0	0	1
1	0	0
0	1	0

Vagyis ha a globális becslő a tipikus viselkedést jósolja (”agree”), akkor a becslés eredménye a lokális eljárás kimenete lesz, ha a globális azt jósolja, hogy most a szokásostól eltérő viselkedés várható (”disagree”), akkor a becslés eredménye a lokális eljárás kimenetével ellentétes lesz. Amit kaptunk, az nem más, mint egy negált XOR függvény. Az ”agree” prediktor blokksémája a 16.8. ábrán látható.

Az Intel Pentium 4 bizonyos változatai az agree prediktor egy változatát használták (a különbség csak annyi, hogy a teljesen globális eljárás szerepét egy gshare prediktor játssza, 16 bites globális előzmény regiszterrel).

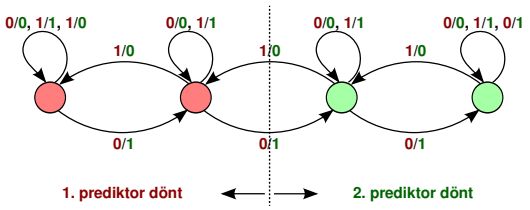
”Tournament” prediktorok

A tournament prediktorok még az agree prediktornál is egyszerűbbek. Ismét két prediktor működik egyszerre, egy lokális és egy globális információkra alapozott. Egy konkrét ugró utasítás elágazásbecslését ezek közül mindig



16.8. ábra. Az agree prediktor

csak az egyik végzi, mégpedig az, amelyik mostanában pontosabban becsült. Azt, hogy mostanában melyik becsült pontosabban, egy egyszerű állapotgéppel követhetjük nyomon. Például a 16.9. ábrán 4 állapotot használunk, ha az állapotgép az első két állapot valamelyikében van, akkor az 1., ellenkező esetben a 2. prediktor fogja becsülni az ugrás kimenetelét. Ha az egyik eljárás jól, a másik rosszul tippelt, akkor a jó irányába lépünk egyet. Mivel 4 állapotunk van, 1 hiba még belefér, 1 rossz becslés esetén még nem vesszük el az aktuálisan döntési helyzetben lévő prediktortól a döntési jogot. (Az ábrában az élek feliratában az első érték azt jelzi, hogy az 1., a második azt, hogy a 2. eljárás eltalálta-e a becslést).



16.9. ábra. Eljáráskiválasztó állapotgép a tournament prediktorban

A tournament prediktoroknál a felhasznált két (vagy több) becslési eljárás tárigényén kívül számolni kell azzal is, hogy az eljáráskiválasztó állapotgép állapotát is tárolni kell. Ez a 4 állapotú megoldás esetén minden ugró utasításra 2 bitet jelent.

Az Alpha 21264 processzor pontosan ilyen tournament prediktort használt, 2 bites eljáráskiválasztó állapotgéppel. A két becslő eljárás egyike lokális ugrási előzmény regisztert használt lokális PHT-vel, a másika pedig globális ugrási előzmény regisztert globális PHT-vel.

16.3. Ugrási cím becslése

Ha a pipeline-ba belép egy ugró utasítás (legyen az feltételes vagy feltétel nélküli), azon nyomban tudni kell, hogy hol fog folytatódni a program végrehajtása. Feltételes ugró utasítás esetén ez azt jelenti, hogy meg kell tippelni, hogy az ugrás be fog-e következni, vagy nem, de minden (nem csak feltételes) ugró utasítás esetén tudni kell az ugrási címet is. Nincs idő címszámítással bíbelődni, mert az ugrási címre a következő ciklus elejéig (tehát még az ugró utasítás dekódolása előtt!) szükség van: onnan kell lehívni a program soron következő utasítását.

16.3.1. Az ugrási cím buffer

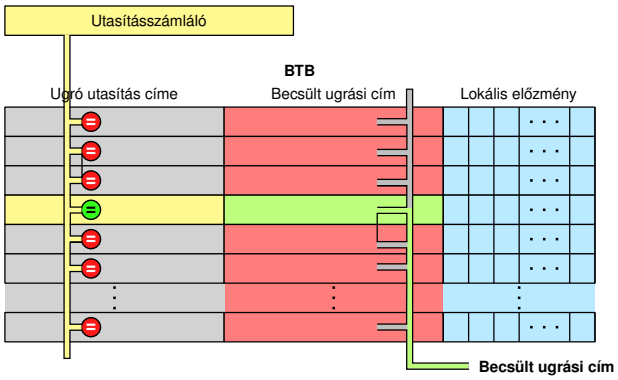
Ami a szűkös időkeretbe belefér, az nem más, mint az ugrási cím kiolvasása egy külön erre a célra fenntartott speciális cache-ből, az ugrási cím buffer-ből (branch target buffer, BTB). Ebben a cache-ben találhatók az összerendelések az ugró utasítások címe (utasításszámlálója) mint cache tag, és a hozzájuk tartozó várható ugrási címek között. Mint minden cache, ez is egy tartalom szerint címezhető memória, mely a tag mezők egyidejű "komparálása" révén egy lépésben vagy találatot ad az ugrási címmel együtt, vagy BTB hibát jelez (vagyis nincs találat).

A BTB szervezése bármilyen, a cache memóriánál megismert szervezést követhet, általában több utas (2 vagy 4 utas) asszociatív szervezés használatos. A tartalom menedzsment célszerűen a legrégebbi ugró utasítást választja a felülíráshoz, amikor egy új, BTB-ben még nem szereplő ugró utasítás felbukkan (LRU). Ha az ugrási cím becslése tévesnek bizonyul (az ugró utasítás elért a címszámítási fázisba, és a BTB tartalmától különböző ugrási cím jött ki), akkor a régi értéket a BTB-ben felülírják.

Ha feltételes ugrásról van szó, a BTB az ugró utasítás címén (utasításszámlálóján, vagy az abból képzett cache tag-en) és a becsült ugrási címen kívül számos egyéb, az adott ugró utasításhoz tartozó, az elágazásbecslést támogató járulékos információt is tartalmazhat:

- egyszerű állapotgép alapú elágazásbecslés esetén a telítődő számláló értékét (általában 2 bit),
- lokális ugrási előzményt használó elágazásbecslés esetén az adott ugró utasítás ugrási előzményét,
- lokális PHT használata esetén az adott ugró utasításhoz tartozó PHT-t.

A 16.10. ábra egy teljesen asszociatív szervezésű BTB-t ábrázol, melyben az ugrási címmel együtt a lokális előzmények is a bejegyzés részeit képezik.



16.10. ábra. Ugrási cím buffer teljesen asszociatív szervezéssel, lokális előzményekkel

Néhány processzor BTB-jének paramétereit a 16.1. táblázat foglalja össze.

A legnagyobb kitolás, ami egy BTB-vel történhet, az az objektumorientált nyelvekben használatos virtuális függvényhívás. Ebben az esetben ugyanis ugyanaz az ugró utasítás nem csak 1 címre ugorhat, hanem többre, attól függően, hogy mi az objektum tényleges típusa. Mivel a BTB a klasszikus esetben csak egy ugrási címet tud tárolni, egy heterogén kollekció bejárása az ugrási címek sorozatos téves becslésével járhat (mivel a BTB mindig a legutóbbi ugrási címet tartalmazza, de virtuális hívás esetén nem biztos, hogy a következő ugrás is ugyanarra a címre történik majd).

A megoldás az lehet, hogy a BTB több ugrási címet is tárol, és az előzmények alapján próbálja meg megtippelni, hogy éppen melyikre fog majd a program ugrani.

16.3.2. Szubrutinhívások visszatérési címének becslése

A szubrutinból való visszatérés (`RET`) egy speciális ugró utasítás, amelynél az ugrási címet a stack tetejéről leolvasott adat határozza meg. (Már amennyiben a processzor a szubrutinhívások visszatérési címeit stack-ben tárolja, mint pl. az x86/x64. Ebben a fejezetben ezt feltételezzük.)

Mivel a stack a rendszermemóriában helyezkedik el, kezelése meglehetősen lassú, ezért a szubrutinhívások visszatérési címét is célszerű becsülni. Erre a célra azonban a BTB-nél hatékonyabb adatszerkezetet is lehet

	BTB bejegyzések	BTB szervezés	Return stack méret
Intel Pentium 4:	4096	valósz. 8 utas asszoc.	nincs
Intel Core2:	ciklusokhoz: 128 indirekt ugr.: 8192 egyéb ugr.: 2048	2-utas asszoc. 4 utas asszoc. 4 utas asszoc.	16
Intel Haswell:	ismeretlen	ismeretlen	16
Intel Atom:	128	4 utas asszoc.	8
AMD Steamroller:	L1: 512 L2: 10240	4-utas asszoc. 5-utas asszoc.	24
Via Nano:	4096	4 utas asszoc.	Igen mély
ARM Cortex A8/A9:	512	2 utas asszoc.	8

16.1. táblázat. Néhány processzor BTB paramétere és return stack mérete

használni: a return stack-et. A return stack nem más, mint egy processzorban elhelyezett, nagy sebességű, de kis kapacitású stack. Szubrutinhíváskor a visszatérési cím nemcsak a rendszermemóriában található lassú stack tetejére kerül, hanem a return stack tetejére is. A szubrutin végén, a `RET` utasításkor az utasításlehívó egység a return stack tetejéről azonnal, még a ciklusidőn belül le tudja olvasni a visszatérési címet, nem kell megvárni a lassú rendszermemóriát.

A return stack kis mérete miatt ez a becslés csak akkor hatékony, ha az egymásba ágyazott szubrutinhívások száma a return stack kapacitása alatt marad, ellenkező esetben a return stack telítődik, és csak az utolsó néhány, a méretének megfelelő számú visszatérési címet tudja nagy sebességgel szolgáltatni.

A gyakorlatban használt return stack méreteket a 16.1. táblázatban láthatjuk.

16.4. Elágazásbecslés-tudatos programozás

A 10.6.2. fejezetben megnéztük, hogy a memóriahierarchia működésének ismeretében milyen programozási szokásokkal, technikákkal lehet egy memória-intenzív algoritmus futási idejét a lehető legkisebbre szorítani.

Az elágazásbecslés témakörében is adódik hardvertudatos optimalizálási lehetőség. Vannak olyan programozási praktikák, ökölszabályok, melyekkel az elágazásbecslő keze alá dolgozva esetenként lényegesen gyorsabb futást lehet elérni, mint egy hardvert figyelmen kívül hagyó implementációval.

16.4.1. Az elágazások számának csökkentése

Az elágazásbecslés nem véletlenül kapott egy teljes fejezetet, minden modern processzorban kiemelt fontosságú funkció. Az elágazások nagymértékben ronthatják a pipeline kihasználtságát, különösen ha nem szabályosan viselkednek. A legkézenfekvőbb dolog, amit ez ügyben egy programozó tehet, hogy kevesebb elágazást használ.

Bizonyos if-then-else szerkezeteket ki lehet váltani egy jól irányzott logikai kifejezéssel. Az alábbi példa egy egész értékeket tartalmazó tömb 500-nál nagyobb elemeit adja össze:

```
for (int i=0; i<N; i++)
    if (data[i] > 500)
        sum += data[i];
```

(alg. 1a.)

Ebben a példában két ponton is elágazásbecslésre van szükség, egyszer a ciklusfeltétel kiértékelésekor, majd az `if` utasítás feltételének kiértékelésekor. Előbbitől most tekintsünk el, már láttuk, hogy az elágazásbecslők a ciklusokat nagyon pontosan kezelik, kétbites állapotgéppel legfeljebb 1 hibás becsléssel számolhatunk a ciklus teljes futása során. Ha a `data` tömb elemei véletlenszerűek, pl. 0 és 1000 között egyenletes eloszlással, akkor az `if` kimenetele kiszámíthatatlan, átlagosan az esetek felében rossz becslést kapunk, és ez teljesítménycsökkenéssel jár.

Ez a feltétel azonban kiváltható egy logikai kifejezéssel. Azt kell észrevenni, hogy nekünk azokat az elemeket kell összeadnunk, melyekre `(data[i]-501)`≥ 0. Shift-eljük el `(data[i]-501)`-et 31-el jobbra! A C nyelv aritmetikai shift-et alkalmaz, tehát a jobb-shift operátora úgy működik, hogy negatív számokra 1-es bitet hoz be balról, egyébként pedig 0-át. Tehát, ha `(data[i]-501)`≥ 0 teljesült, akkor a shift-elés csupa 0, ha nem teljesült, csupa 1-es bitet eredményez. Ennek ismeretében az elágazásmentes változat a következő:

```
for (int i=0; i<N; i++) {
    int t = (data[i]-501) >> 31;
    sum += ~t & data[i];
}
```
(alg. 1b.)

A következő példában a tömb `min` és `max` közé eső elemeit adjuk össze:

```
for (int i=0; i<N; i++)
    if (min<=data[i] && data[i]<=max)
        sum += data[i];
```
(alg. 2a.)

Az `if` itt most két feltételt is tartalmaz. Először a két feltételt egyre csökkentjük. A fő észrevétel ez esetben az, hogy `min<=data[i] && data[i]>=max` logikai kifejezés és a `(unsigned)(data[i]-min)<=max-min` logikai kifejezés ugyanaz. Azt könnyű látni, hogy `data[i]>max` esetén mindkét kifejezés hamis. Ha azonban `data[i]<min`, akkor `data[i]-min` negatív, ami előjel nélküli típusra alakítva egy igen nagy szám lesz (lásd: kettes komplemens számábrázolás), nagyobb, mint a legnagyobb pozitív előjeles egész, és amiatt az "új", egyszerűbb kifejezésünk úgyszintén hamis lesz, pont, ahogy szeretnénk. A hatékonyabb algoritmus tehát:

```
for (int i=0; i<N; i++)
    if ((unsigned)(data[i]-min) <= max-min)
        sum += data[i];
```
(alg. 2b.)

Most már csak egy összehasonlítás van az if feltételében, úgyhogy alkalmazhatjuk az első példa trükkjét, így egyetlen feltételes elágazás sem marad:

```
for (int i=0; i<N; i++) {
    int t = (max-min-(unsigned)(data[i]-min)) >> 31;
    sum += ~t & data[i];
}
```
(alg. 2c.)

Sokszor az if-then-else szerkezetek kiválthatók tömb, ún. lookup table használatával. A következő példa egy `N` hosszú karaktertömb minden elemét ellenőrzi, és a betűk kivételével mindent szóközre cserél. Az egyszerű, nem elágazástudatos megoldás:

```
for (int i=0; i<N; i++)
    if (!((data[i]>='a' && data[i]<='z') || (data[i]>='A' && data[i]<='Z')))
        data[i] = ' ';
```
(alg. 3a.)

Ebben az algoritmusban a feltételkiértékelések száma a tömb méretével arányos. Ha a szöveg véletlenszerű betűket tartalmaz, akkor az elágazásbecslő sokat hibázik, ami lassítja a programot. Az elágazás ebben az esetben is kiváltható. Első lépésben egy segédtömböt (lookup-table-t) készítünk el, amely minden betűre tartalmazza azt a karaktert, amire azt ki kell cserélni. Ennek a felépítése tartalmaz ugyan if-then-else szerkezetet, de csupán 256-ot, `N` értékétől függetlenül. A második lépésben haladunk végig a nagy `data` tömbön, de akkor már a segédtömbből olvassuk a cserekaraktereket, nem pedig az if-then-else segítségével. A végeredmény:

```
for (int j=0; j<256; j++)
    if (!((j>='a' && j<='z') || (j>='A' && j<='Z')))
        LUT[j] = ' ';
    else
        LUT[j] = j;
for (int i=0; i<N; i++)
    data[i] = LUT[data[i]];
```
(alg. 3b.)

A konkrét számítógépeken kapott mérési eredményeket a 16.2. táblázat veti össze. Az elágazások eliminálása

lényegesen javítja a futási időket. A legnagyobb mértékű gyorsulás az elágazás sikerességére leginkább rászoruló, mély pipeline-al rendelkező i7 és P4 architektúrák esetében mutatkozott, az egyszerűbb felépítésű processzoroknál kisebb a nyereség.

	i7-2600	P4 Northwood	Raspberry Pi	RK3188
1. példa, eredeti	7.583 ms	14.122 ms	59.202 ms	11.296 ms
1. példa, elágazásmentes	1.297 ms	4.251 ms	58.410 ms	8.628 ms
2. példa, eredeti	8.211 ms	19.6 ms	73.267 ms	21.496 ms
2. példa, 1 elágazással	7.942 ms	14.295 ms	61.578 ms	13.347 ms
2. példa, elágazásmentes	1.203 ms	4.252 ms	58.328 ms	8.268 ms
3. példa, eredeti	6.533 ms	10.377 ms	48.532 ms	21.397 ms
3. példa, segédtömbbel	1.151 ms	3.641 ms	37.896 ms	18.240 ms

16.2. táblázat. Az elágazások csökkentésének hatása különböző architektúrákon, N=2048*1024

16.4.2. Az elágazások kiszámíthatóbbá tétele

Az előző példában azért kaptunk kedvezőtlen futási időket az if-then-else elágazást tartalmazó algoritmusokra, mert a tömb elemei véletlenszerűek voltak, az elágazási feltétel kimenetelében az elágazásbecslő semmilyen szabályosságot nem tudott felfedezni. Ebben a fejezetben mérésekkel számszerűsítjük, hogy mennyit számít a kiszámíthatóság a program futási sebessége szempontjából.

Az első példában az ugrások kimenetelének a kiszámíthatóságát vizsgáljuk. Visszatérünk egy korábbi programrészlethez, amely egy tömb 500-nál nagyobb elemeit összegzi:

```
for (int i=0; i<N; i++)
    if (data[i] > 500)
        sum += data[i];
```

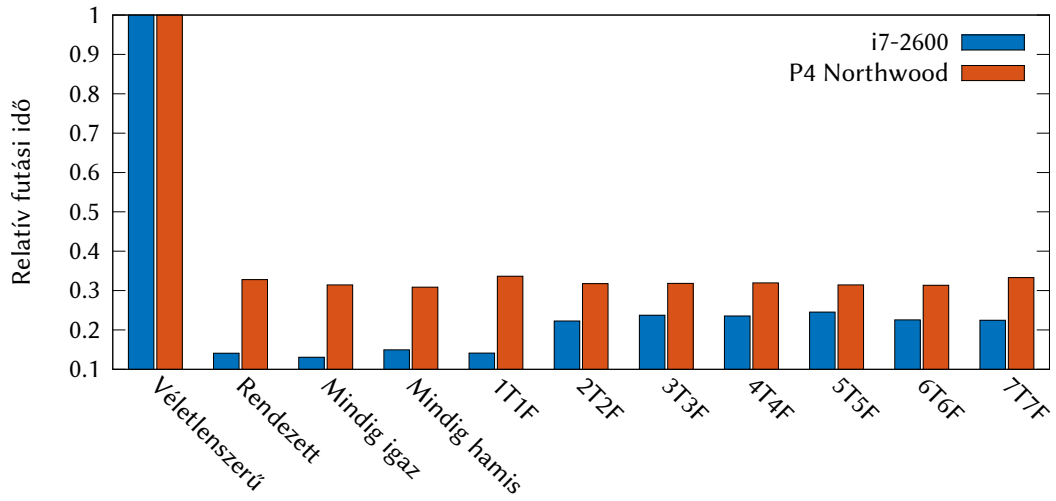
A futási időket az alábbi esetekre vizsgáljuk:

- a tömb véletlenszerű elemeket tartalmaz,
- rendezett tömbre,
- olyan tömbre, melynek minden elemére teljesül a feltétel,
- olyan tömbre, melynek egyik elemére sem teljesül a feltétel,
- végül pedig a feltétel teljesülésére egy szabályos mintát írunk elő. Ebben az esetben a szabályosságot NTNF alakban jellemezzük, melyben periodikusan N teljesülés N meghiúsulást követ.

Az eredmények a 16.11. ábrán láthatók, melyen az oszlopmagasságok a véletlenszerű esetre kapott futási időkhöz relatívak. A tömb feldolgozás előtti rendezése drasztikus gyorsulást eredményez, legfőképp a Core i7-2600 architektúrán, amely a legfejlettebb elágazásbecslővel rendelkezik. A Core i7 szemmel láthatólag még az 1T1F (TFTFTFTF...) mintázatot is tökéletesen felismeri, de a 2T2F (TTFFTTFF...) már egy kicsit lassítja a futást. A Pentium 4 esetén kevésbé szembetűnő a szabályos esetek közötti különbség, de a szabályosságot ez a processzor is meghálálja.

16.4.3. Az ugrási cím becsülhetőségének javítása

Most azt vizsgáljuk meg, hogy az ugrási cím kiszámíthatóságának mekkora a jelentősége. A méréshez az objektumorientált programozásban nap mint nap használt eszközt, egy heterogén kollekciót hozunk létre. Egy olyan tömböt, mely egy `A` osztály 16 féle leszármazottját tudja tárolni. Az `A` osztály és egy tetszőleges leszármazott `B1` definíciója a következő:



16.11. ábra. Az ugrási feltétel kiszámíthatóságának a hatása

```
class A {
public:
    virtual int value ()=0;
    virtual int type () const =0;
    virtual ~A() {}
};
```

```
class B1 : public A {
    int v;
public:
    B1 () : v(rand()) {}
    int value () { return ++v; }
    int type () const { return 1; }
    ~B1 () {}
};
```

A `B1`...`B16` osztályokból vegyesen feltöltünk egy nagy méretű tömböt, majd az így kapott heterogén kollekcióra végrehajtjuk az alábbi algoritmust:

```
sum = 0;
for (i=0; i<siz; i++)
    sum += data[i]->value();
```

Ez a `for` ciklus a heterogén kollekció minden elemére végrehajt egy virtuális tagfüggvény-hívást, amely egy összetett művelet. Az objektum memóriaterületén (mely a `data[i]` címen kezdődik) található egy adatszerkezet, melyet *virtuális függvény-táblának* hívnak, és amely mutatókat tartalmaz az aktuális objektum példány virtuális függvényeire. Ha `data[0]` típusa `B1`, `data[1]` típusa pedig `B2`, akkor a `value()` függvényre vonatkozó bejegyzésben a `data[0]` virtuális függvény-táblája a `B1::value()`-ra, a `data[1]`-é pedig a `B2::value()`-ra mutató függvénypointert tartalmaz.

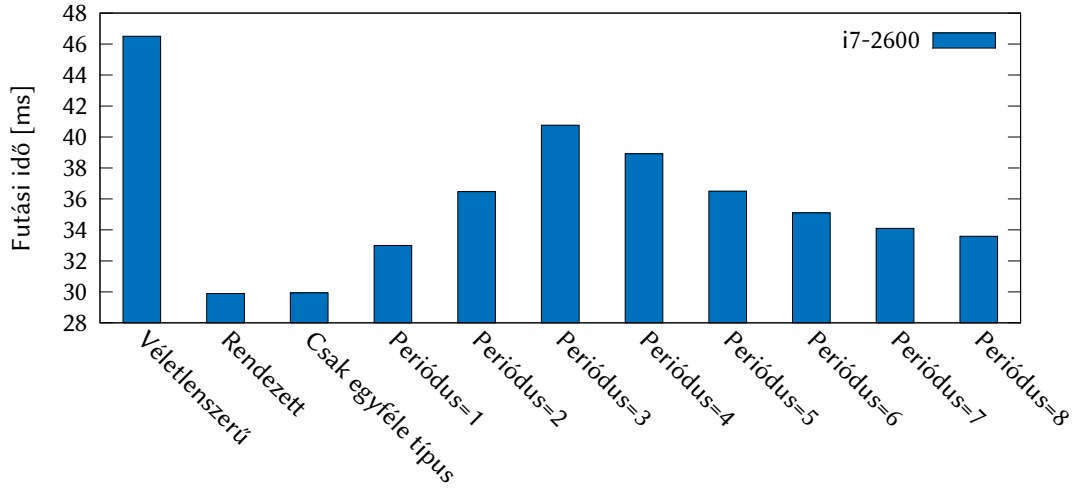
A virtuális függvényhívás tehát *indirekt* függvényhívás, ahol az objektum típusának megfelelő `value()` függvényre kerül a vezérlés. Ha a heterogén kollekcióban keverten, véletlen sorrendben vannak a különféle típusú elemek, akkor a processzorok ugrási cím becslő eljárása nagyon rossz hatásfokkal működik. Célszerű tehát a heterogén kollekciót az objektumok típusa szerint rendezni, hiszen ekkor a tömbben egyforma típusú elemek kerülnek egymás mellé, azonos ugrási címmel, melyre a processzor hamar rátanul.

A rendezettség, illetve szabályosság hatását az alábbi esetekre vizsgáltuk:

- A heterogén kollekció elemei véletlenszerűen tartoznak a 16 leszármazotthoz,
- a heterogén kollekciót a tárolt objektumok típusa szerint rendezzük (a `type()` függvény segítségével),
- a heterogén kollekció minden eleme egyazon osztály példánya,
- végül, az elemek típusára szabályos mintázatot írunk elő. Egy ”periódus” paraméter határozza meg, hogy hány azonos típusú objektum van egymás mellett a tömbben.

A mérési eredményeket a 16.12. ábra foglalja össze. Csak az i7-2600 processzor számait közöljük, mivel a többi architektúra olyan fejletlen ugrási cím becsléssel rendelkezik, hogy minden oszlop egyforma magasnak adódott. Az igazsághoz hozzátartozik, hogy a virtuális függvényhívás mögött *indirekt ugrás* áll, ami a legnehezebben becsülhető ugrástípus, számos processzor meg sem próbálkozik a becslésével.

Az i7-2600 processzor esetén azonban a heterogén kollekció rendezése számottevő gyorsuláshoz vezetett (a különbség 50% körüli). A processzor még az egyszerűbb, 1 periódusú szabályosságot is ki tudja használni (B1, B2, ..., B16, B1, B2, ..., B16, ...), de a kettő periódusút már nem. Hosszabb periódusok esetén azért javul az eredmény, mert a kézenfekvő, előző ugrási címre tippelő becslés egyre kevesebb hibával jár.



16.12. ábra. Az ugrási cím kiszámíthatóságának a hatása

17. fejezet

Védelem

A védelem (protection) célja a számítógép bizonyos szoftver és hardver komponenseinek korlátozása, a rendszer integritásának és működőképességének fenntartása érdekében. Hatóköre nemcsak a processzorra korlátozódik, hanem a memóriára és a perifériákra is, de mivel a védelmi szabályok érvényesítése a processzor dolga, itt, a processzorokkal foglalkozó részben tárgyaljuk.

A védelem feladata, hogy a rosszindulatú vagy hibás taszkokat megakadályozza abban, hogy *másokban* kárt tegyenek, azaz

- más taszk vagy az operációs rendszer privát adatait, kódját elérjék, megváltoztassák;
- más taszk vagy az operációs rendszer privát felhasználásra szánt függvényeit meghívják;
- az operációs rendszer megkerülésével és tudta nélkül kommunikáljanak perifériákkal.

Tehát a védelem a többi taszkot és az operációs rendszert védi, hogy a hibás taszkok mellett is maradéktalanul el tudják látni feladatukat.

A rendszer integritására, működőképességére nem csak a hibásan megírt szoftver jelent veszélyt, hanem a perifériák is. Ezért egy jól átgondolt védelmi szabályrendszer a perifériákra is korlátozást vezet be, hogy azok egy rosszul irányzott DMA művelet segítségével, vagy bármi más módon ne tudjanak kárt okozni a taszkoknak, illetve az operációs rendszernek.

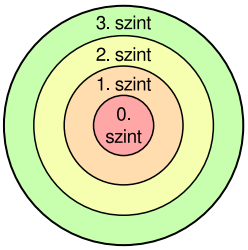
17.1. A privilégiumi szintek fogalma

Ahhoz, hogy védelmi szabályokat lehessen megfogalmazni, a taszkokhoz privilégiumi szinteket rendelnek a szerint, hogy milyen jogosultságokkal rendelkeznek. A legprivilegizáltabb, vagyis a legtöbb jogosultsággal rendelkező taszkok a 0-ás szinthez tartoznak, és a szintek növekedése a jogosultságok fokozatos korlátozását követi. A privilégiumi szinteket gyűrűk formájában is szokás megjeleníteni, és védelmi gyűrűknek nevezni (17.1. ábra).

A privilégiumi szintek száma architektúrafüggő. Egy védett operációs rendszer lehetővé tételéhez 2 szint már elegendő (a 0. szintre kerül az operációs rendszer, a legkülsőre pedig a felhasználói programok), de több szint használatának is lehet értelme a finomabb jogosultságkezelés érdekében. Az 1973-ban megjelent Honeywell 6180-as számítógép 8 védelmi szintet különböztet meg, az x86 architektúra 4-et, míg számos további (múltbéli és élő) architektúra, mint az ARM, MIPS, Alpha, SPARC, Motorola 68k csupán 2-őt. Azt pedig az operációs rendszere válogatja, hogy a rendelkezésre állók közül hány szintet használ valójában. Az x86 4 védelmi szintjéből mind a Windows, mind a Linux operációs rendszerek újabb változatai csupán kettőt használnak: a 0-ást az operációs rendszernek, a 3-ast pedig a felhasználói programoknak.

A taszkokhoz rendelt privilégiumi szintek az alább képességeket befolyásolják:

- **Azt, hogy egy taszk milyen utasításokat használhat.** Minden védelmi támogatással rendelkező utasításkészlet architektúra rendelkezik *privilegizált utasításokkal*, melyet csak a magas privilégiumi szinttel rendelkező taszkok (pl. az operációs rendszer) adhatnak ki, a felhasználói taszkok nem. Ide tartozik pl. a laptáblák kezelése is, hiszen a laptábla módosításával egy tetszőleges taszk a teljes fizikai memóriát írni és olvasni tudná.
- **Azt, hogy egy taszk milyen memóriaterületeket érhet el.** Nyilvánvalóan nem lenne szerencsés, ha egy felhasználói taszk betekintést kaphatna az operációs rendszer privát adatszerkezeteibe, ha ezeket írni is



17.1. ábra. Védelmi gyűrűk (privilégiumi szintek)

képes lenne, az a rendszer stabilitását is fenyegetné. Az ellentétes irány viszont megengedett, sőt, egyenesen elkerülhetetlen ahhoz, hogy egy operációs rendszer be tudjon tölteni egy alacsonyabban privilegizált taszkot a memóriába.

- **Hogy egy taszk milyen függvényeket hívhat meg.** Az operációs rendszer belső használatra szánt függvényeinek meghívását célszerű elérhetetlenné tenni egy alacsonyabban privilegizált taszk számára.
- **Hogy egy taszk mely perifériákat használhatja.** Szintén célszerű megakadályozni, hogy egy taszk a kritikus fontosságú perifériákat közvetlenül érje el, hiszen ellenkező esetben lehetősége nyílna rá, hogy tévedésből, vagy szándékosan úgy kezeljék azokat, hogy érvénytelen állapotba kerülve megbénítsák a rendszert.

17.2. Memóriavédelem

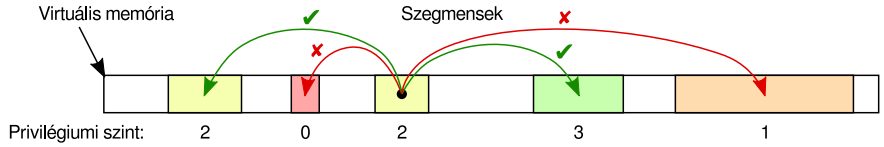
A memóriavédelem alapeszköze a címtér-elkülönítés (9.5), hiszen ha a taszkok külön címtérben dolgoznak, akkor nem képesek egymás adataihoz hozzáférni.

17.2.1. Memóriavédelem szegmensszervezésű memóriakezelés esetén

Ha a hardver támogatja a szegmensszervezést (ill. lapozott szegmentálást, lásd 9.5.2. fejezet), akkor a memóriavédelem nagyon egyszerűen és rugalmasan megvalósítható: minden egyes taszkot és magát az operációs rendszert is külön-külön szegmensbe kell helyezni. A szegmensleíró tábla a szegmens kezdetén és hosszán kívül ugyanis védelmi biteket is tartalmaz, melyek megadják, hogy a tartalmuk milyen privilégiumi szinten van. Amíg egy taszk a saját szegmenseivel dolgozik, addig másnak nem eshet bántódása. Azonban, ha egy taszk egy másik (vagyis *távoli*) szegmenshez kíván hozzáférni, akkor a processzor védelmi rendszere összehasonlítja a futó taszk és a megcélzott szegmens privilégiumi szintjét, és ellenőrzi, hogy megengedett-e a hozzáférés.

Az x86 architektúrában a szabályok a következők (17.2. ábra):

- Egy taszk írhatja/olvashatja a vele megegyező, vagy nála magasabb szintű (tehát kevésbé privilegizált) szegmensek adatait, távoli címzés segítségével.
- Egy taszk *nem* férhet hozzá a nála jobban privilegizált (tehát alacsonyabb szinten lévő) szegmensekhez.



17.2. ábra. Megengedett szegmensközi memóriaműveletek

Ha a memóriaművelet nem megengedett, akkor egy kivétel keletkezik, és az operációs rendszer erre a célra beregisztrált szubrutinja kapja meg a vezérlést. Az operációs rendszer jellemzően terminálja a taszkot, miközben a felhasználó felé hibát jelez (segmentation fault, protection fault, stb.)

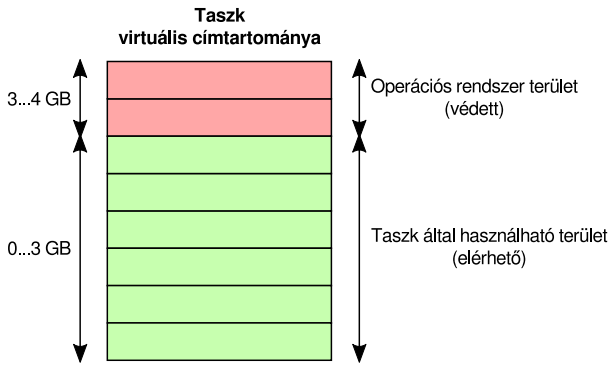
17.2.2. Memóriavédelem tisztán a laptáblára alapozva

Szegmentálás hiányában a memóriavédelem megvalósítása egy kicsit más megközelítést igényel. Az itt leírt módszer messze a legnépszerűbb a gyakorlatban, hiszen számos architektúrában nincs szegmentálás, ráadásul manapság az operációs rendszerek akkor sem használják ki a szegmentálás okozta lehetőségeket, ha az rendelkezésre áll (pl. Windows és Linux x86-on).

A 9.5. fejezetben már láttuk, hogy az operációs rendszerek képesek minden taszk számára külön címteret biztosítani, ha taszkváltáskor laptáblát is váltanak. Ebben az esetben a taszkok által kiadott virtuális címek garantáltan nem tudnak hozzáférni egy másik taszk memóriaterületéhez, hiszen az ahhoz tartozó keretek nincsenek benne a laptáblájában. Ez a teljes elkülönítés meg is oldaná a tökéletes memóriavédelmet. A probléma azonban az, hogy az operációs rendszernek is mindig elérhetőnek kell lennie a virtuális memóriában, több okból is:

- Egy periféria bármikor megszakítást generálhat, a megszakítást lekezelő szubrutinnak mindig rendelkezésre kell állnia a virtuális memóriában. Ennek speciális esete, ha a hardveres időzítő generál megszakítást, hogy ideje taszkot váltani: a taszkváltást végző szubrutinnak, amely nem az épp futó taszk, hanem az operációs rendszer része, elérhetőnek kell lennie, hogy rá kerülhessen a vezérlés.
- A taszkok időnként hívják az operációs rendszer szubrutinjait (ún. rendszerhívásokat), pl. ha meg szeretnének nyitni egy fájlt, kérik a lenyomott billentyű kódját, stb. Ha ezek a szubrutinok nem érhető el a virtuális címtérben, akkor nem lehet meghívni őket, hiszen nincs cím, ahova ugrani kellene.

A mai modern operációs rendszerek ezért minden taszk virtuális címterét két részre bontják: az alsó fele (*user space*) a taszk használatában van, a felső felére vonatkozó laptábla bejegyzéseket pedig úgy töltik ki, hogy a mögött mindig az operációs rendszer álljon (*kernel space*). Ekkor azonban tenni kell valamit, hogy a taszk ne tudjon hozzáférni az immár számára is láthatóvá (címezhetővé) vált operációs rendszer területhez. A megoldást a laptábla bejegyzések egy speciális bitje jelenti, amivel az adott laphoz való hozzáférést korlátozni lehet. Az x86 laptábla bejegyzéseiben ezt ”User/Supervisor” bitnek hívják, ha egy lapra be van állítva, akkor azt csak a legmagasabb privilégiumi szintről lehet elérni, az illetéktelen elérési kísérlet pedig védelmi hibát eredményez, amit az operációs rendszer lekezel. Ezek a lehetőség szinte minden architektúra laptábla bejegyzéseiben rendelkezésre áll (pl. az ARM ”Domain Access Control” biteknek hívja). Az x86-on, 32 bites módban a Windows 2 GB-ot ad a címtartományból a taszkoknak (3 GB-ra növelhető), 64 bites módban pedig 8 TB-ot. A Linuxnál ez paraméter, a kernel fordításakor beállítható.



17.3. ábra. A virtuális címtartomány megosztása a taszk és az operációs rendszer között

Összegzésképp, a tisztán lapozásra alapozott memóriavédelem elemei:

- Minden taszk virtuális címtartománya fel van osztva a taszk és operációs rendszer között.
- Az operációs rendszer taszkváltáskor laptáblát is vált, úgy, hogy az operációs rendszerre vonatkozó laptábla bejegyzések ugyanazok maradjanak.
- Az operációs rendszer területére vonatkozó laptábla bejegyzésekben be kell állítani a ”User/Supervisor” bitet.

Ezzel megoldódott a taszkok szeparálása úgy, hogy az operációs rendszer is mindig elérhető, de mégsem tud a taszk ártani neki. Van azonban egy probléma: ha az operációs rendszert elérhetetlenné tettük a taszkok számára, hogyan tudja a taszk meghívni annak rendszerhívásait? Ezzel foglalkozik a következő fejezet.

17.3. A vezérlésátadások ellenőrzése

Egy alacsony privilégiumi szintű taszknak időnként szüksége lehet egy magasabb privilégiumi szintű taszk részére a vezérlést átadni, például amikor egy felhasználói program az operációs rendszer függvényeit (rendszerhívásait) hívja, pl. ha ki akar írni valamit a képernyőre, meg akar nyitni egy fájlt, be akarja olvasni a legutóbb leütött billentyűzet kódját, stb.

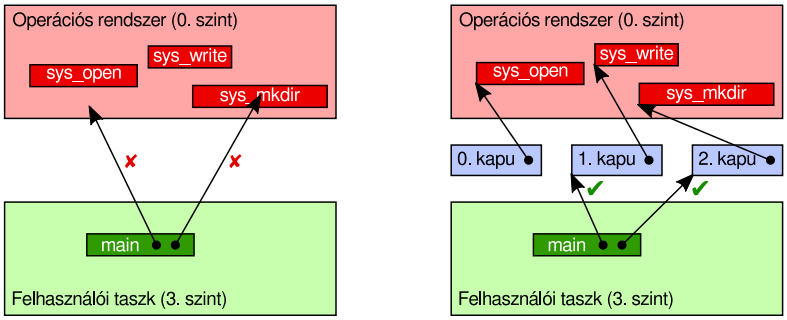
Ennek a vezérlésátadásnak azonban kontrolláltnak kell lennie, nem lehet a védett, magasan privilegizált kódba bárhol belépni, vagy annak bármilyen függvényét lefuttatni.

17.3.1. Kapukra alapozott távoli vezérlésátadás az x86-ban

Szegmensszervezés mellett a távoli vezérlésátadás (pl. operációs rendszer hívása) kézenfekvő módja lenne, ha csak meg kellene adni a cél szegmenst és azon belül a cél bájtpozíciót, és a processzor onnan folytatná az utasítások végrehajtását.

Ez azonban nem megengedett. Magasabban privilegizált szegmens felé nem engedhető meg, hogy *bárhová* át lehessen adni a vezérlést, hiszen az operációs rendszer kizárólag a taszkok számára exponált rendszerhívásait szeretné láthatóvá tenni. Másrészt alacsonyabban privilegizált szegmensre sem lehet ugrani, hiszen az megbízhatatlanabb kódot tartalmaz, nem garantált a sikeres visszatérés.

Az x86-ban a szegmensközi vezérlésátadásra úgynevezett kapukat (gate-eket) használnak (több fajtája is van, call gate, task gate, trap gate, interrupt gate, a köztük lévő különbséget nem tárgyaljuk). A távoli vezérlésátadásra szolgáló utasításnak egyetlen paramétere van: hányas kapura szeretne ugrani. A kapukat a rendszermemóriában tárolt leíró táblázat sorolja fel, főbb mezői: a célszegmens, az azon belüli bájtpozíció (ezek alapján végrehajtható az ugrás), valamit védelmi információk: legalább hányas szint szükséges a meghíváshoz. Ennek a megközelítésnek az a fő ötlete, hogy a kapukat egy felhasználói taszk nem változtathatja meg. Az operációs rendszer hozza létre, minden rendszerhíváshoz egyet-egyet, és a felhasználói taszk csak annyit közölhet a processzorral, hogy hányas kaput szeretné meghívni (17.4. ábra).



17.4. ábra. Szegmenseken átnyúló távoli vezérlésátadás kapukkal

(Zárójelben jegyezzük meg, hogy ez azért nem ilyen egyszerű. Többek között egy érdekes probléma a stack kezelése. Rendszerhívás esetén, ami tulajdonképpen egy függvény, a hívási paraméterek és a lokális változók a stack-be kerülnek. De kinek a stack-jébe? Ha a hívó stack-jébe kerülnének, akkor visszatérés után a felhasználói taszkok látnák a rendszer stack-jükben hagyott lokális változóit, amit esetleg rosszindulatúan "hasznosíthatnak". Ha az op. rendszer a saját stack-jét használja a rendszerhíváskor, akkor viszont nem tudja átvenni a függvény paramétereit, amit a hívó a saját stack-jén keresztül szeretne átadni. A megoldás: külön stack-et használ mindkét fél, és távoli híváskor a hívási paramétereket a processzor a hívó stack-jéből a hívott stack-jébe másolja.)

17.3.2. Szoftver megszakításra alapozott rendszerhívások

A szegmentálás és a kapuk használata egy jól átgondolt, rugalmas megoldást adott a különböző szegmensekben, különböző privilégiumi szinteken futó taszkok közötti kommunikációra.

A gyakorlatban sokszor nincs szükség ekkora flexibilitásra. A modern operációs rendszerek nem engedik meg a különböző felhasználói taszkok közötti vezérlésátadást, így az egész problémakör arra a kérdésre korlátozódik, hogy hogyan lehet egy felhasználói taszkból az operációs rendszer rendszerhívásait biztonságosan meghívni.

Erre a célra a legtöbb architektúrán a legtöbb operációs rendszer szoftver megszakításokat (vagy annak valamely változatát) használja, x86-on is, ahol egyébként lenne mód a kapuk alkalmazására.

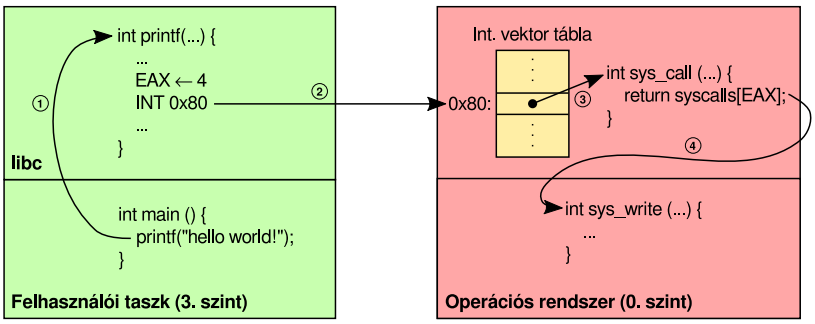
A 4.3. fejezetben megismerkedtünk a perifériák által kiváltott megszakításokkal, annak kezelési módjával. Bevezettük az interrupt vektor táblát, amely a különböző interruptok kiszolgálását végző szubrutinok kezdőcímét tartalmazza. Arról azonban ott még nem volt szó, hogy megszakítást nem csak periféria, hanem maga a futó program is kiválthat, az erre szolgáló utasítás által, ezeket a megszakításokat hívják *szoftver megszakításnak*. Az x86 architektúra 256 elemű, az ARM 8 elemű vektortáblával rendelkezik, ennyi megszakításkezelő szubrutint tudnak kezelni. Célszerű ezeket diszjunkt módon felosztani, egy részére hardvert kötni, a fennmaradó vektorokat pedig szoftveres megszakításként használni. Az x86-ban az `INT` utasítással lehet ilyen megszakítást kiváltani, pl. az `INT 0x80` a vektortábla 128-as bejegyzésében található függvénypointert hívja meg. Az ARM architektúra csak egyetlen szoftveresen meghívható megszakítást támogat, ezt az `SWI` utasítással lehet kiváltani, és ekkor a vektor tábla (fixen) 2-es bejegyzésében található függvényre ugrik.

A szoftveres megszakítás tehát olyan, mint egy függvényhívás. A védelem szempontjából van azonban egy fontos jellegzetessége:

- x86 esetén a vektortábla nem csak a kiszolgáló függvényre mutató pointert tartalmazza, hanem azt is, hogy milyen privilégiumi szinten történik a lekezelése,
- az ARM (és szinte minden más RISC) architektúrában pedig megszakítás esetén (legyen az szoftveres vagy hardveres), a legnagyobb privilégiumi szintre kapcsol a processzor.

A szoftver interruptok tehát alkalmasak arra, hogy alacsony privilégiumi szintű kódból magasat hívjunk meg. (A vektortáblát természetesen csak az magasan privilegizált módban lehet megváltoztatni, pl. az operációs rendszer indításkor feltölti).

Könnyen belátható, hogy elegendő, ha az operációs rendszer egyetlen belépési pontot, egyetlen szoftver megszakítást kínál fel a taszkok számára. Azt, hogy a taszk a sok közül pontosan melyik rendszerhívást szeretné meghívni, át lehet adni regiszterben. A 17.5. ábrán egy példát láthatunk, melyben a felhasználói taszk a ”hello world!” üzenetet kívánja kiírni a képernyőre Linux operációs rendszer alatt. A szabványos c könyvtár (libc) printf függvénye ennek érdekében egy ponton az operációs rendszerhez fordul, hogy karaktereket írjon a standard kimenetre. Ehhez a `sys_write` rendszerhívás szükséges: az `EAX` regiszterbe 4-et ír (ez az `sys_write` sorszáma), és az `INT 0x80` kiadásával a Linux rendszerhívásokra fenntartott szoftver megszakítását meghívja. Ezen a ponton a processzor privilégiumi szintet emel, és a linux `system_call()` függvénye kerül meghívásra, mely az `EAX=4` alapján a `sys_write()` függvényre adja a vezérlést.



17.5. ábra. Rendszerhívás szoftveres megszakításra alapozva

ARM architektúrában is hasonló a rendszerhívások megvalósítása: az `SWI 4` hatására, a privilégiumi szint megemelése után meghívódik a `system_call()`, amely az utasításszóból kiolvassa, hogy a 4-es rendszerhívásra, vagyis a `sys_write()` függvényre van szükség.

Az x86 architektúrában a Pentium II processzorral megjelent két új utasítás, a `sysenter` és a `sysexit` (az AMD esetében `syscall` és a `sysret`), melyeket kifejezetten a rendszerhívások meghívására és az azokból való visszatérésre vezettek be. Ugyanúgy működnek, mint az imént leírtak, de lényegesen gyorsabbak, mint a szoftver megszakítások lekezelése. Mielőtt megjelent volna ez a gyors megoldás, a Windows a `0x2e`-s, a Linux pedig a `0x80`-as megszakítást foglalta le rendszerhívások számára.

17.4. Perifériák védelme, ill. védelem a perifériáktól

Emlékezzünk rá, hogy a PCI/PCI Express perifériák gyakorlatilag tetszőleges memóriacímre kezdeményezhetnek adatátvitelt. Elég egy hibás eszközmeghajtó, vagy egy rosszindulatú periféria, és egy DMA művelettel tetszőleges taszk, vagy magának az operációs rendszernek a területe is módosíthatóvá válik.

Védelmi szabályokra van szükség annak érdekében, hogy

- meg lehessen tiltani egy programnak, hogy az operációs rendszer megkerülésével kommunikálhasson egy perifériával,
- meg lehessen tiltani egy perifériának, hogy egy védett memóriaterületet elérjen, netalántán összefirkáljon.

17.4.1. A perifériákhoz való hozzáférés szabályozása

Memóriára leképzett perifériakezelés esetén egyszerű a megoldás: az operációs rendszernek elérhetetlenné kell tennie a perifériákhoz rendelt memóriatartományt a taszkok számára, például úgy, hogy a laptábla bejegyzésekben beállítja a ”Supervisor” bitet.

Szeparált I/O utasítások esetén két lehetőség áll rendelkezésre:

a) Az I/O utasítások privilegizálttá tételéve megakadályozható, hogy egy felhasználói taszk bárminemű perifériakezelést végezzen.

b) Minden taszkhoz meghatározzuk, hogy mely I/O címekhez kaphat hozzáférést. Ez egy lényegesen engedékenyebb politika, mely lehetővé teszi, hogy egyes, a rendszer szempontjából kevésbé kritikus perifériákat egy felhasználói program közvetlenül is el tudjon érni.

Az Intel x86 architektúra mindkét megoldást támogatja. Az aktuálisan futó taszk jellemzőit tartalmazó adatszerkezetben (melyet az operációs rendszer állít össze, és amely a rendszermemóriában található), szerepel egy ún. ”I/O bitmap” mező, mely egy bitekből álló tömb. Az *i*. pozíciójába írt 1-es az *i*. I/O cím engedélyezését jelenti az aktuális taszk számára.

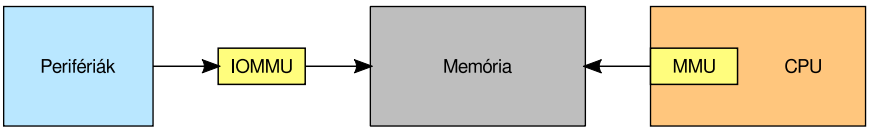
Ugyanebben az adatszerkezetben, egy másik mező segítségével pedig a taszk eltiltható minden perifériakezelő utasítástól.

17.4.2. A perifériák memóriaműveleteinek korlátozása

A védelem szemszögéből a DMA egy veszélyes művelet, hiszen ennek során a periféria fizikailag címzi és éri el a memóriát, a processzor minden memóriavédelmi eszközének megkerülésével.

A processzoron futó taszkok memória-hozzáféréseit a processzorban található MMU felügyeli, mely a virtuális címekből a laptábla segítségével állítja elő a fizikai címet, miközben a jogosultság ellenőrzésére is képes.

A perifériák memória-hozzáféréseit is lehet ehhez hasonló módon felügyelni, ehhez azonban egy új elem, az IOMMU (I/O memory management unit) szükséges. Korábban az IOMMU a processzort kiszolgáló északi híd része volt, ma már azonban a processzor és az északi híd egy eszközben, integrált formában van jelen. Az IOMMU nem egy új fogalom, a SPARC, a MIPS és az Intel Itanium architektúrák régóta használják, de a PC-k világába az AMD hozta be 2007-ben, az Intel 2008-ban követte (a VT-d részeként).



17.6. ábra. DMA-val szembeni memóriavédelem IOMMU-val

Az IOMMU működésében nagyon hasonlít az MMU-hoz. A perifériák virtuális címekkel dolgoznak, melyeket az IOMMU fordít fizikai címekké egy saját laptábla segítségével. Ezt a laptáblát is az operációs rendszer kezeli. A laptábla bejegyzésekben található védelmi bitek lehetővé teszik a fizikai memória egyes részeinek levédését a DMA műveletekkel szemben. A memóriavédelem mellett az IOMMU más egyéb szempontból is hasznos:

- A 32 bites buszt használó (PCI/PCI Express) eszközök számára lehetővé válik a 4 GB feletti címtartomány használata
- A DMA átvitelhez nincs szükség fizikailag folytonos memóriatartományra

18. fejezet

Számpéldák, feladatok a processzorok témakörében

18.1. Feladatok a pipeline utasításütemezés témakörében

18.1.1. Feladat

Legyen adott az alábbi utasítás sorozat:

```
i1:  R0 ← MEM [R1+8]
i2:  R2 ← R0 * 3
i3:  R3 ← MEM [R1+12]
i4:  R4 ← R3 * 5
i5:  R0 ← R2 + R4
```

Az utasítássorozatot a 12.4. fejezetben megismert egyszerű 5 fokozatú pipeline-t használó processzor hajtja végre.

(a) Azonosítsa az összes előforduló adat-egymásrahatást!

(b) Adja meg az utasítássorozat ütemezését (melyik utasítás mikor melyik fázisban van)! Ha szünetet kell beiktatni, jelezze, hogy mi az oka! Használja az alábbi jelöléseket:

- A^*: a szünet oka adategymásrahatás
- F^*: a szünet oka feldolgozási egymásrahatás
- P^*: a szünet oka procedurális egymásrahatás

(c) Minden egyes utasításnál jelölje, hogy amikor az EX fázisba ér, szükség van-e forwarding-ra! Ha igen, adja meg, hogy melyik pipeline regiszterből kell a szükséges értéket visszacsatolni!

(d) Ha egységnyi ideig tart minden fázis, mennyivel gyorsabb a pipeline-al való végrehajtás, mint anélkül?

(e) Rendezze át az utasítássorozatot úgy, hogy az a lehető leggyorsabban fusson le, gyorsabban, mint eredetileg!

Megoldás

(a) Az alábbi adat-egymásrahatások fordulnak elő a kódban:

- RAW egymásrahatások: i1 ↔ i2, i3 ↔ i4, i2 ↔ i5, i4 ↔ i5
- WAW egymásrahatás: i1 ↔ i5
- WAR egymásrahatás: i2 ↔ i5

(b) Az ütemezés a következők szerint alakul:

Utasítás	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.
i1	IF	ID	EX	MEM	WB						
i2		IF	ID	A^*	EX	MEM	WB				
i3			IF	F^*	ID	EX	MEM	WB			
i4					IF	ID	A^*	EX	MEM	WB	
i5						IF	F^*	ID	EX	MEM	WB

Az i2-nek szüksége lenne az i1 eredményére, de az sajnos csak a MEM fázis végén, az 5. fázisban fog rendelkezésre állni. Ezért az i2 végrehajtását 1 ciklussal késleltetni kell. A késleltetés után, amikor az i2 végrehajtása az EX fázisba lép, az i1 eredménye még nem lesz az R0 regiszterbe visszaírva, ezért azt forwardig segítségével a MEM/WB pipeline regiszterből kell az EX fázisnak kiolvasnia. Az i3-mat is késleltetni kell, hiszen a 4. ütemben az ID egység foglalt (az i2 foglalja), így az i3 az IF fázisban 1 ciklust várakozni kényszerül. Mivel a 4. ütemben foglalt az IF, az i4 végrhajtásának elkezdése is késlekedést szenved, csak az 5. ütemben kerülhet az IF fázisba. Sajnos az i4-nek szüksége van az i3 eredményére, ami csak a 8. ütemre lesz kész, ezért ismét szünetet kell beiktatni.

(c) Forwarding használata:

- i2: az EX során a MEM/WB regiszterből veszi az R0 leendő értékét
- i4: az EX során a MEM/WB regiszterből veszi az R3 leendő értékét
- i5: az EX során az EX/MEM regiszterből veszi az R4 leendő értékét

(d) Az ütemezésből látható, hogy pipeline-al 11 ciklus kellett a végrehajtáshoz (feltöltéssel és kiürüléssel együtt). Pipeline nélkül:

- i1: IF+ID+EX+MEM+WB = 5 ciklus
- i2: IF+ID+EX+WB = 4 ciklus
- i3: IF+ID+EX+MEM+WB = 5 ciklus
- i4: IF+ID+EX+WB = 4 ciklus
- i5: IF+ID+EX+WB = 4 ciklus

összesen: 22 ciklus.

(e) A pipeline-t úgy lehet felgyorsítani, hogy az utasításokat átrendezve elérjük, hogy ne kelljen szüneteket beiktatni. Pl. így:

```
i1:  R0 ← MEM [R1+8]
i2:  R3 ← MEM [R1+12]
i3:  R2 ← R0 * 3
i4:  R4 ← R3 * 5
i5:  R0 ← R2 + R4
```

Ezzel a cserével eltávolítottuk egymástól a feloldhatatlan RAW függésben lévő utasításokat. A feladat nem kérte, de a cserékkel együtt az ütemezés a következő:

Utasítás	1.	2.	3.	4.	5.	6.	7.	8.	9.
i1	IF	ID	EX	MEM	WB				
i2		IF	ID	EX	MEM	WB			
i3			IF	ID	EX	MEM	WB		
i4				IF	ID	EX	MEM	WB	
i5					IF	ID	EX	MEM	WB

18.1.2. Feladat

Legyen adott az alábbi utasítás sorozat:

```
i1:  R5 ← MEM [R3+24]
i2:  R6 ← MEM [R4+16]
i3:  R7 ← R6 + R5
i4:  R8 ← R6 - R5
i5:  R5 ← R7 * R8
i6:  R4 ← R4 + 4
```

Az utasítássorozatot egy 6 fokozatú utasítás pipeline-t használó processzor hajtja végre. A pipeline minden utasítás végrehajtását 5 részműveletre bontja: betölti (IF), dekódolja (ID), végrehajtja a vonatkozó aritmetikai műveletet (EX), majd a vonatkozó memóriaműveletet (MEM), végül a regisztertárolóba írja az eredményregiszter értékét (WB). Az ID, EX, MEM és WB fázisok késleltetése 1 órajel, az IF fázis késleltetése 2 órajel, de iterációs ideje 1 órajel (vagyis 2 pipeline fokozatként jelenik meg: IF0 és IF1). Minden utasítás végrehajtása mind az 5 részműveleten átesik, függetlenül attól, hogy szüksége van-e rá. Minden forwarding út használata megengedett. Ha bármilyen egymásrahatás az utasítás megállítását igényli, az utasítás mindig a legutolsó olyan fázisban áll meg, ameddig egymásrahatás nélkül eljut.

(a) Azonosítsa az összes előforduló adat-egymásrahatást!

(b) Adja meg az utasítássorozat ütemezését (melyik utasítás mikor melyik fázisban van)! Ha szünetet kell beiktatni, jelezze, hogy mi az oka! Használja az alábbi jelöléseket:

- A^*: a szünet oka adategymásrahatás
- F^*: a szünet oka feldolgozási egymásrahatás
- P^*: a szünet oka procedurális egymásrahatás

(c) Minden egyes utasításnál jelölje, hogy amikor az EX fázisba ér, szükség van-e forwarding-ra! Ha igen, adja meg, hogy melyik pipeline regiszterből kell a szükséges értéket visszacsatolni!

(d) Ha egységnyi ideig tart minden fázis, mennyivel gyorsabb a pipeline-al való végrehajtás, mint anélkül?

(e) Rendezze át az utasítássorozatot úgy, hogy az a lehető leggyorsabban fusson le, gyorsabban, mint eredetileg!

Megoldás

(a) Az alábbi adat-egymásrahatások fordulnak elő a kódban:

- RAW egymásrahatások: i1 ↔ i3, i2 ↔ i3, i1 ↔ i4, i2 ↔ i4, i3 ↔ i5, i4 ↔ i5
- WAW egymásrahatás: i1 ↔ i5
- WAR egymásrahatás: i3 ↔ i5, i4 ↔ i5, i2 ↔ i6

(b) Az ütemezés az alábbiak szerint alakul:

Utasítás	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.
i1	IF0	IF1	ID	EX	MEM	WB						
i2		IF0	IF1	ID	EX	MEM	WB					
i3			IF0	IF1	ID	A^*	EX	MEM	WB			
i4				IF0	IF1	F^*	ID	EX	MEM	WB		
i5					IF0	F^*	IF1	ID	EX	MEM	WB	
i6							IF0	IF1	ID	EX	MEM	WB

A szünet beiktatása azért szükséges, mert i3-nak kellene az R6 értéke, de azt az i2 csak a MEM fázis végén, a 6. ütemben állítja elő. Ekkor kerül be a MEM/WB regiszterbe, ahonnét az EX fázis ki tudja venni, amikor i3-at hajtja végre. Tehát a szünet miatt i3 az 5. és a 6. ütemben is az ID fázisban tartózkodik, ami miatt az i4 nem tud továbblépni az IF1-ből, vagyis a szünet őt is akadályozza. Mivel az 6. ütemben foglalt az IF1, az i5 is blokkolódik, az i6 lehívása pedig nem történhet meg egészen a 7. ütemig.

(c) Forwarding használata:

- i3: az EX során a MEM/WB regiszterből veszi az R6 leendő értékét
- i5: az EX során az EX/MEM regiszterből veszi az R8 leendő értékét, és a MEM/WB-ből az R7 leendő értékét

(d) Az ütemezésből látható, hogy pipeline-al 12 ciklus kellett a végrehajtáshoz (feltöltéssel és kiürüléssel együtt). Pipeline nélkül:

- i1: IF0+IF1+ID+EX+MEM+WB = 6 ciklus
- i2: IF0+IF1+ID+EX+MEM+WB = 6 ciklus
- i3: IF0+IF1+ID+EX+WB = 5 ciklus
- i4: IF0+IF1+ID+EX+WB = 5 ciklus
- i5: IF0+IF1+ID+EX+WB = 5 ciklus
- i6: IF0+IF1+ID+EX+WB = 5 ciklus

összesen: 32 ciklus.

(e) A pipeline-t úgy lehet felgyorsítani, hogy az utasításokat átrendezve elérjük, hogy ne kelljen szüneteket beiktatni. Pl. az i6-ot, ami i2 után mindegy, hogy mikor hajtódik végre, beszúrhatjuk az i2 mögé, így az i3-nak egy ciklussal később lesz szüksége az i2 eredményére, és addigra az pont meg is lesz:

```
i1:  R5 ← MEM [R3+24]
i2:  R6 ← MEM [R4+16]
i3:  R4 ← R4 + 4
i4:  R7 ← R6 + R5
i5:  R8 ← R6 - R5
i6:  R5 ← R7 * R8
```

Ezzel a cserével eltávolítottuk egymástól a feloldhatatlan RAW függésben lévő utasításokat. A feladat nem kérte, de a cserékkel együtt az ütemezés a következő:

Utasítás	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.
i1	IF0	IF1	ID	EX	MEM	WB					
i2		IF0	IF1	ID	EX	MEM	WB				
i3			IF0	IF1	ID	EX	MEM	WB			
i4				IF0	IF1	ID	EX	MEM	WB		
i5					IF0	IF1	ID	EX	MEM	WB	
i6						IF0	IF1	ID	EX	MEM	WB

Ebben az optimalizált kódban a forwarding a következő pontokon lesz szükséges:

- i4: az EX során a MEM/WB regiszterből veszi az R6 leendő értékét
- i6: az EX során az EX/MEM regiszterből veszi az R8 leendő értékét, és a MEM/WB-ből az R7 leendő értékét

18.1.3. Feladat

Legyen adott az alábbi utasítás sorozat:

```
i1:  D0 ← D1 * D2
i2:  D3 ← D0 + D5
i3:  MEM [R0+4] ← D3
i4:  MEM [R0+12] ← D0
```

A D0-D5 regiszterek lebegőpontos számokat, az R0 egész számot tárol.

A pipeline minden utasítás végrehajtását 5 részműveletre bontja: betölti (IF), dekódolja (ID), végrehajtja a vonatkozó aritmetikai műveleteket, majd a vonatkozó memóriaműveleteket (MEM), végül a regisztertárolóba írja az eredményregiszter értékét (WB). Minden utasítás mind az 5 részműveleten átesik, függetlenül attól, hogy szüksége van-e rá. Az IF, ID, MEM és WB fázisok késleltetése 1 órajel. Az aritmetikai művelet lehet egész művelet, mely késleltetése 1 órajel (EX, a címszámítást is ez az egység végzi), lebegőpontos összeadás, melynek késleltetése 4, iterációs ideje 1 (A0, A1, A2, A3), valamint lebegőpontos szorzás, melynek késleltetése 7, iterációs ideje pedig 1 (M0, M1, M2, M3, M4, M5, M6). A három aritmetikai egység képes párhuzamos működésre, és az utasítások soron kívüli befejezésére, ha ennek semmilyen szemantikai vagy egymásrahatásbeli akadálya nincs. Minden forwarding út használata megengedett. Ha bármilyen egymásrahatás az utasítás megállítását igényli, az utasítás mindig a legutolsó olyan fázisban áll meg, ameddig egymásrahatás nélkül eljut.

(a) Azonosítsa az összes előforduló adat-egymásrahatást!

(b) Adja meg az utasítássorozat ütemezését (melyik utasítás mikor melyik fázisban van)! Ha szünetet kell beiktatni, jelezze, hogy mi az oka! Használja az alábbi jelöléseket:

- A^*: a szünet oka adategymásrahatás
- F^*: a szünet oka feldolgozási egymásrahatás
- P^*: a szünet oka procedurális egymásrahatás

(c) Rendezze át az utasítássorozatot úgy, hogy az a lehető leggyorsabban fusson le, gyorsabban, mint eredetileg!

(d) Adja meg az átrendezett utasítássorozat ütemezését is, a szünetek okának megjelölésével!

Megoldás

(a) Az alábbi adat-egymásrahatások fordulnak elő a kódban:

- RAW egymásrahatások: i1 ↔ i2, i2 ↔ i3, i1 ↔ i4
- WAW egymásrahatás és WAR egymásrahatás nincs

(b) Az ütemezés az alábbiak szerint alakul:

Utasítás	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	
i1: $D0 \leftarrow D1 * D2$	IF	ID	M0	M1	M2	M3	M4	M5	…
i2: $D3 \leftarrow D0 + D5$		IF	ID	A^*	A^*	A^*	A^*	A^*	…
i3: $MEM[R0 + 4] \leftarrow D3$			IF	F^*	F^*	F^*	F^*	F^*	…
i4: $MEM[R0 + 12] \leftarrow D0$									

	9	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
i1:	M6	MEM	WB						
i2:	A^*	A0	A1	A2	A3	MEM	WB		
i3:	F^*	ID	EX	A^*	A^*	F^*	MEM	WB	
i4:		IF	ID	F^*	F^*	F^*	EX	MEM	WB

Az is helyes megoldás, ha az adatfüggőségben lévő Store műveletek nem az EX-ben, hanem az ID-ben várakoznak!

(c) Az utolsó két sor megcserélésével gyorsabb futást várunk, hiszen így távolabb kerülnek egymástól az adatfüggőségben lévő utasítások

(d) Az átrendezés után az ütemezés az alábbiak szerint alakul:

Utasítás	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	
i1: $D0 \leftarrow D1 * D2$	IF	ID	M0	M1	M2	M3	M4	M5	…
i2: $D3 \leftarrow D0 + D5$		IF	ID	A^*	A^*	A^*	A^*	A^*	…
i3: $MEM[R0 + 12] \leftarrow D0$			IF	F^*	F^*	F^*	F^*	F^*	…
i4: $MEM[R0 + 4] \leftarrow D3$									

	9	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
i1:	M6	MEM	WB						
i2:	A^*	A0	A1	A2	A3	MEM	WB		
i3:	F^*	ID	EX	MEM	WB				
i4:		IF	ID	EX	A^*	F^*	MEM	WB	

18.1.4. Feladat

Legyen adott az alábbi utasítás sorozat:

```
i1:  D2 ← D0 * D1
i2:  MEM [R0+0] ← D2
i3:  D2 ← D0 + D1
i4:  MEM [R0+8] ← D2
```

A D0-D2 regiszterek lebegőpontos számokat, az R0 egész számot tárol.

A pipeline minden utasítás végrehajtását 5 részműveletre bontja: betölti (IF), dekódolja (ID), végrehajtja a vonatkozó aritmetikai műveleteket, majd a vonatkozó memóriaműveleteket (MEM), végül a regisztertárolóba írja az eredményregiszter értékét (WB). Minden utasítás mind az 5 részműveleten átesik, függetlenül attól, hogy szüksége van-e rá. Az IF, ID, MEM és WB fázisok késleltetése 1 órajel. Az aritmetikai művelet lehet egész művelet, mely késleltetése 1 órajel (EX, a címszámítást is ez az egység végzi), lebegőpontos összeadás, melynek késleltetése 4, iterációs ideje 1 (A0, A1, A2, A3), valamint lebegőpontos szorzás, melynek késleltetése 7, iterációs ideje pedig 1 (M0, M1, M2, M3, M4, M5, M6). A három aritmetikai egység képes párhuzamos működésre, és az utasítások soron kívüli befejezésére, ha ennek semmilyen szemantikai vagy egymásrahatásbeli akadálya nincs. Minden forwarding út használata megengedett. Ha bármilyen egymásrahatás az utasítás megállítását igényli, az utasítás mindig a legutolsó olyan fázisban áll meg, ameddig egymásrahatás nélkül eljut.

(a) Azonosítsa az összes előforduló adat-egymásrahatást!

(b) Adja meg az utasítássorozat ütemezését (melyik utasítás mikor melyik fázisban van)! Ha szünetet kell beiktatni, jelezze, hogy mi az oka! Használja az alábbi jelöléseket:

- A^*: a szünet oka adategymásrahatás
- F^*: a szünet oka feldolgozási egymásrahatás
- P^*: a szünet oka procedurális egymásrahatás

Megoldás

(a) Az alábbi adat-egymásrahatások fordulnak elő a kódban:

- RAW egymásrahatások: i1 ↔ i2, i3 ↔ i4

- WAW egymásrahatás: i1 ↔ i3
- WAR egymásrahatás: i2 ↔ i3

(b) Az egyetlen nehézség, hogy a WAW elkerülése érdekében az i3 ID fázisában annyi szünetet kell beiktatni, hogy végül az i3 WB-je ne előzze meg az i1 WB-jét, máskülönben rossz érték kerülne a D2 regiszterbe. Az ütemezés tehát az alábbiak szerint alakul:

Utasítás	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	
i1: $D2 \leftarrow D0 * D1$	IF	ID	M0	M1	M2	M3	M4	M5	…
i2: $MEM[R0 + 0] \leftarrow D2$		IF	ID	EX	A^*	A^*	A^*	A^*	…
i3: $D2 \leftarrow D0 + D1$			IF	ID	A0	A1	A2	A3	…
i4: $MEM[R0 + 8] \leftarrow D2$				IF	ID	F^*	F^*	F^*	…

	9	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
i1:	M6	MEM	WB						
i2:	A^*	F^*	MEM	WB					
i3:	A^*/F^*	A^*/F^*	A^*/F^*	MEM	WB				
i4:	F^*	F^*	EX	F^*	MEM	WB			

18.2. Feladatok a regiszter átnevezés témakörében

18.2.1. Feladat

Legyen adott az alábbi utasítás sorozat:

```
i1:  D1 ← MEM [R0+0]
i2:  D1 ← D0 * D1
i3:  D4 ← D2 / D1
i4:  D2 ← MEM [R1+0]
i5:  D3 ← D0 + D2
i6:  D4 ← D4 - D3
i7:  MEM [R1+16] ← D4
```

A ”D” kezdetű regiszterek lebegőpontos, az ”R” kezdetűen egész értékeket tárolnak.

(a) Rajzolja fel az utasítások percedenciagráfját! Az éleket cimkézze fel az egymásrahatások típusának megfelelően (RAW/WAR/WAW)! Ha két utasítás között több egymásrahatás is van, tüntesse fel mindet! Feltéve, hogy minden utasítás az utasítástárolóban van, hány ciklus szükséges az utasítássorozat végrehajtásához egy ideális out-of-order szuperskalár processzoron, melynek végtelen sok feldolgozóegysége van, és minden utasítást egyetlen lépésben végrehajt?

(b) Szüntesse meg a WAW és WAR adat-egymásrahatásokat regiszter átnevezés segítségével! A regiszter átnevezést végezze el minden utasításra szisztematikusan! A lebegőpontos fizikai regisztereket jelölje `T0, T1, T2, ...`, az egész típusú fizikai regisztereket `U0, U1, U2, ...`. A regiszterleképző tábla tartalma kezdetben:

Logikai	Fizikai regiszter
R0:	U7
R1:	U3
D0:	T6
D1:	T2
D2:	T8
D3:	T1
D4:	T11

Ha új, szabad fizikai regiszterre van szüksége, válassza a táblázatban szereplő legnagyobb sorszámú után következőt!

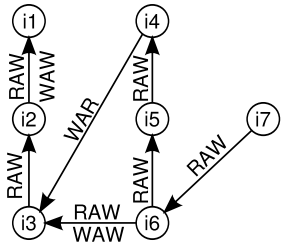
(c) Rajzolja fel az utasítások percedenciagráfját a regiszter átnevezés után! Hány ciklus szükséges az utasítássorozat végrehajtásához az ideális out-of-order szuperskalár processzoron?

Megoldás

(a) Az alábbi WAW és WAR adat-egymásrahatások fordulnak elő a kódban:

- WAR:
  - i4: i3-mal
- WAW:
  - i2: i1-mal
  - i6: i3-mal

Az egymásrahatások alapján felrajzolhatjuk a precedenciagráfot:



A sok függőség miatt ezt az utasítássorozatot egy ideális szuperskalár processzor sem képes 7 ciklusnál gyorsabban végrehajtani. Az első lépésben csak az i1 hajtható végre, a másodikban csak az i2, stb.

(b) A regiszter átnevezést a 14.2.3. fejezetben megismert eljárás szerint, sorról sorra haladva végezzük el. Először az első sort vesszük (i1), és a nyíl jobb oldalán szereplő forrás operandusokat (esetünkben R0-t) a regiszter leképző tábla alapján átnevezzük. Ezután az utasítás eredményét egy eddig még nem használt, tiszta regiszterbe tesszük (T12), majd frissítjük a regiszter leképző táblát, hogy azok az utasítások, melyek az eredménnyel (D1 regiszterrel) dolgoznak tovább, tudják, hogy a T12-t kell használniuk. Ennek megfelelően a D1-hez tartozó bejegyzésbe T12 kerül. Ezután vesszük a következő sort (i2), a forrásoperandusokat (D0,D1) átnevezzük a táblázat segítségével, az eredményt vadonat új regiszterbe tesszük (T13), és a táblázatban rögzítjük, hogy az eredeti eredmény regiszter (D1) értékét már a T13 tartalmazza. Ugyanígy járunk el a többi sorral is. Egyedül az utolsó sornál jöhetünk zavarba, hiszen az i7 eredménye nem regiszterbe, hanem a memóriába kerül. Ilyenkor természetesen nem allokálunk új eredmény regisztert i7 számára, és ennek megfelelően a regiszter leképző tábla sem változik az i7 utasítás átnevezése után.

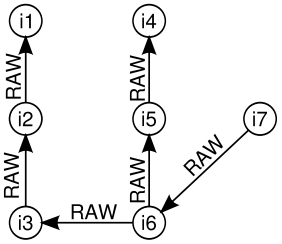
Az utasítás sorozat az átnevezés után tehát a következőképpen néz ki:

```
i1:  T12 ← MEM [U7+0]
i2:  T13 ← T6 * T12
i3:  T14 ← T8 / T13
i4:  T15 ← MEM [U3+0]
i5:  T16 ← T6 + T15
i6:  T17 ← T14 - T16
i7:  MEM [U3+16] ← T17
```

A regiszter leképző tábla alakulását az egyes utasítások regiszterátnevezése után az alábbi táblázatban követhetjük nyomon:

Logikai regiszter:	Fizikai regiszter: Kezdő:	i1:	i2:	i3:	i4:	i5:	i6:	i7:
R0:	U7							
R1:	U3							
D0:	T6							
D1:	T2	T12	T13					
D2:	T8				T15			
D3:	T1					T16		
D4:	T11			T14			T17	

(c) A regiszterátnevezés eredményeképp a WAW és WAR egymásrahatások láthatóan megszűntek. A precedenciagráf ennek megfelelően ritkább lesz:



Az ideális szuperskalár processzor az első ciklusban i1-et és i4-et egyszerre képes végrehajtani, majd a második ciklusban i2-őt és i5-öt egyszerre, ezután pedig i3-mat, i6-ot, i7-et sorban egymás után. Ez 5 lépés, 2 lépéssel gyorsabb, mint regiszterátnevezés előtt.

18.2.2. Feladat

Legyen adott az alábbi utasítás sorozat:

```
i1:  D1 ← D2 - D3
i2:  D4 ← D1 + D2
i3:  D5 ← D4 + D1
i4:  D0 ← D2 / D5
i5:  D1 ← MEM [R0+0]
i6:  D4 ← D1 + D2
i7:  D5 ← D4 + D1
```

A ”D” kezdetű regiszterek lebegőpontos, az ”R” kezdetűen egész értékeket tárolnak.

(a) Rajzolja fel az utasítások percedenciagráfját! Az éleket cimkézze fel az egymásrahatások típusának megfelelően (RAW/WAR/WAW)! Ha két utasítás között több egymásrahatás is van, tüntesse fel mindet! Feltéve, hogy minden utasítás az utasítástárolóban van, hány ciklus szükséges az utasítássorozat végrehajtásához egy ideális out-of-order szuperskalár processzoron, melynek végtelen sok feldolgozóegysége van, és minden utasítást egyetlen lépésben végrehajt?

(b) Szüntesse meg a WAW és WAR adat-egymásrahatásokat regiszter átnevezés segítségével! A regiszter átnevezést végezze el minden utasításra szisztematikusan! A lebegőpontos fizikai regisztereket jelölje `T0, T1, T2, ...`, az egész típusú fizikai regisztereket `U0, U1, U2, ...`. A regiszterleképző tábla tartalma kezdetben:

Logikai	Fizikai regiszter
R0:	U4
D0:	T6
D1:	T2
D2:	T8
D3:	T1
D4:	T9
D5:	T10

Ha új, szabad fizikai regiszterre van szüksége, válassza a táblázatban szereplő legnagyobb sorszámú után következőt!

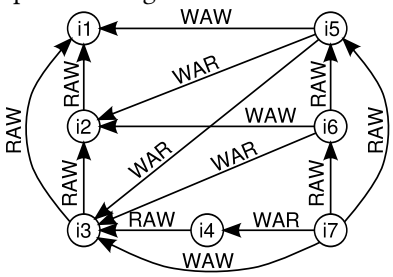
(c) Rajzolja fel az utasítások percedenciagráfját a regiszter átnevezés után! Hány ciklus szükséges az utasítássorozat végrehajtásához az ideális out-of-order szuperskalár processzoron?

Megoldás

(a) Az alábbi WAW és WAR adat-egymásrahatások fordulnak elő a kódban:

- WAR:
  - i5: i2-vel és i3-mal
  - i6: i3-mal
  - i7: i4-gyel
- WAW:
  - i5: i1-gyel
  - i6: i2-vel
  - i7: i3-mal

Az egymásrahatások alapján felrajzolt precedenciagráf:



A függőségek miatt ezt az utasítássorozatot egy ideális szuperskalár processzor 6 ciklus alatt hajtja végre: először az i1-et, majd i2-őt, majd az i3-at, ezután i4-et és i5-öt egyszerre, végül i6-ot és i7-et egymás után.

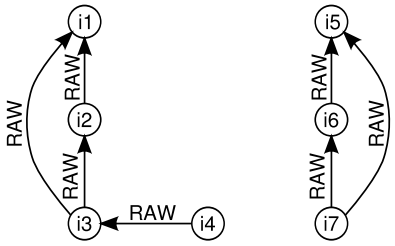
(b) Az előző példánál követett megoldási menet az alábbi utasítássorozatot eredményezi:

```
i1:  T11 ← T8 - T1
i2:  T12 ← T11 + T8
i3:  T13 ← T12 + T11
i4:  T14 ← T8 / T13
i5:  T15 ← MEM [U4+0]
i6:  T16 ← T15 + T8
i7:  T17 ← T16 + T15
```

A regiszter leképző tábla tartalma az egyes utasítások regiszterátnevezése során az alábbi táblázat szerint alakul:

Logikai regiszter:	Fizikai regiszter: Kezdő:	i1:	i2:	i3:	i4:	i5:	i6:	i7:
R0:	U4							
D0:	T6				T14			
D1:	T2	T11				T15		
D2:	T8							
D3:	T1							
D4:	T9		T12				T16	
D5:	T10			T13				T17

(c) A regiszterátnevezésnek hála a WAW és WAR egymásrahatások mind megszűntek, a precedenciagráf pedig a következő:



Az ideális szuperskalár processzor az első ciklusban i1-et és i5-öt egyszerre képes végrehajtani, majd a második ciklusban i2-őt és i6-ot egyszerre, ezután i3-mat és i7-et egyszerre, végül i4-et egyedül. Ez 4 lépés, 2 lépéssel gyorsabb, mint regiszterátnevezés előtt.

18.3. Feladatok a VLIW architektúrák témakörében

18.3.1. Feladat

Adott az alábbi C kód:

```
if (a < 0) {
    b = 1;
    c = 2;
}
else {
    d = 3;
    e = 4;
}
```

(a) Írja át a fenti C kódot a tárgyban használt pszeudo-assembly nyelvre! Hány utasításra van szükség?

(b) Írja át a fenti C kódot a tárgyban használt pszeudo-assembly nyelvre, predikátumok alkalmazásával! Hány utasításra van szükség?

(c) Hány utasítást kell végrehajtani a predikátumokkal és az anélkül megvalósított kódban, ha

- az $a < 0$ az esetek 20%-ában teljesül?
- az $a < 0$ az esetek 80%-ában teljesül?

(d) Ideális esetben hány lépésben hajtható végre a fenti kód 4 végrehajtó egységgel rendelkező VLIW processzorral predikátumokkal és anélkül?

Megoldás

(a) A C kód az alábbi 6 utasítással írható le (a, b, c, d és e változók rendre az R0, R1, R2, R3, R4 regiszterekben vannak):

```
        JUMP label IF R0≥0
        R1 ← 1
        R2 ← 2
        JUMP end
label:  R3 ← 3
        R4 ← 4
end:
```

(b) Predikátumok használatával 5 utasítás is elég (de nem ez a predikátumok fő előnye!):

```
P1,P2 ← R0 < 0
R1 ← 1 IF P1
R2 ← 2 IF P1
R3 ← 3 IF P2
R4 ← 4 IF P2
```

(c) Predikátumokkal a fenti pszeudo-assembly kód a feltétel teljesülési gyakoriságától függetlenül mindig 5 lépés alatt fut le. Predikátumok nélkül: ha $a < 0$ teljesül, 4 (az első négy), ha nem teljesül, 3 lépés (ez első és a label címke utáni kettő) szükséges. Tehát:

- ha az $a < 0$ az esetek 20%-ában teljesül: átlagosan $0.2 \cdot 4 + 0.8 \cdot 3 = 3.2$ ciklus
- ha az $a < 0$ az esetek 80%-ában teljesül: átlagosan $0.8 \cdot 4 + 0.2 \cdot 3 = 3.8$ ciklus

Tehát ebből a szempontból a predikátumok kifejezetten rossz választásnak tűnhetnének, de vegyük észre, hogy predikátumokkal nincs szükség elágazásbecslésre! A teljes képhez az is kell, hogy ha a predikátumok nélküli kódban rosszul dönt az elágazásbecslő, hozzá kell venni a járulékos kiesett ciklusokat is.

(d) VLIW architektúra esetén a predikátumos kód ideális esetben 2 órajelciklus alatt lefut, hiszen a predikátumok kiértékelése 1 órajelciklus (ideális esetben), az azt követő 4 egymástól független utasítás pedig 1 újabb órajelciklus alatt egyszerre végrehajtható. Predikátumok nélkül, ha az eredeti C kód feltétele teljesül, 2 órajelciklus kell (első csoport: első JUMP, utána a két értékadás, valamint a második JUMP mehet egyszerre a második csoportban), ha nem teljesül, akkor is 2 órajelciklus kell (első JUMP, a label utáni két értékadás egyszerre). Tehát a predikátumok ugyanakkora futási idő mellett, elágazásbecslés nélkül alkalmasak ennek az egyszerű feltételes szerkezetnek a leírására. Megjegyezzük, hogy más példákban lehet, hogy a predikátumos, megint másokban lehet, hogy a predikátumok nélküli eredményez hatékonyabb kódot, az, hogy melyiket érdemes használni, mindig mérlegelés kérdése.

18.3.2. Feladat

Legyen adott az alábbi utasítássorozat:

```
i1:  R2 ← MEM [R0+0]
i2:  R3 ← R0 * R2
i3:  R8 ← R4 / R3
i4:  R5 ← MEM [R1+8]
i5:  R6 ← R2 + R5
i6:  R9 ← R5 / R6
i7:  R10 ← R6 * R9
```

Az utasítássorozatot lefordítjuk egy olyan VLIW processzorra, melyben az alábbi utasítások helyezhetők el egy utasításcsoportban:

- 2 db memóriaművelet (végrehajtási idő: 3 órajel, iterációs idő: 1)
- 2 db egész aritmetikai művelet, vagy ugró utasítás (végrehajtási idő: 1 órajel)

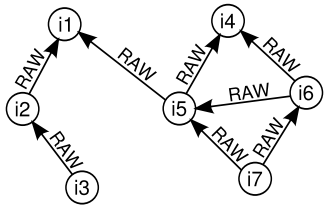
(a) Ütemezze a megadott utasítássorozatot a megadott VLIW processzoron! Határozza meg az utasításcsoportok tartalmát, és a csoportok végrehajtásának idejét!

(b) Mennyivel gyorsabb az így kapott program, mintha hagyományos 1-utas processzoron futtatnánk (pipeline nélkül)?

(c) Hány utasításcsoportot állít elő a fordító klasszikus, és hányat dinamikus VLIW architektúra esetén?

Megoldás

(a) A párhuzamosan végrehajtható utasítások összeválogatásához érdemes először a függőségi viszonyokat felderíteni. Az alábbi ábrán látható függőségi gráfban a csomópontok az utasítások, a csomópontból induló irányított élek pedig azt jelzik, hogy az utasítás melyik másik utasítás eredményétől függ:



A függőségi gráfból könnyen kiolvasható, hogy az első órajelben a két memóriautasítás (i1 és i4) hajtható végre egyszerre, hiszen azok nem függenek senkitől. Ezek eredménye csak 3 órajelciklus múlva áll majd rendelkezésre, amikor is a csak az i1-re és i4-re váró i2 és i5 végrehajthatóvá válik, egyszerre. A következő körben a csak a már elkészültektől függő utasítások, i3 és i6 hajtható végre párhuzamosan, majd az i7. Az eredmény tehát (a ki nem töltött helyeken NOP értendő):

	Mem 1	**Mem 2**	**Egész 1**	**Egész 2**
1:	i1	i4		
2:				
3:				
4:			i2	i5
5:			i3	i6
6:			i7	

(b) Ha egyszerűen összeadjuk minden utasítás végrehajtási idejét, 11 jön ki, szemben az előző pontban, a VLIW megoldásra kiszámolt 6 órajellel.

(c) A klasszikus VLIW architektúra nem képes az egymásrahatások kezelésére (sem a detektálására, sem a megfelelő számú szünet beiktatására), ilyenkor a fordítóprogramnak kell a kellő számú szünetet a programba elhelyeznie. A fordító az előbbi táblázat minden sorát a programba fordítja (a csupa NOP sorokat is), tehát a program 6 utasításcsoportból fog állni. Dinamikus VLIW esetben az egymásrahatásokat a processzor kezeli, ilyenkor a 2 csupa NOP sort nem kell a programnak tartalmaznia, mert a processzor észre fogja venni, hogy a memóriaműveletek miatt 2 szünetre lesz szüksége. Ekkor tehát 4 utasításcsoportból áll a program.

18.3.3. Feladat

Legyen adott az alábbi utasítássorozat:

```
i1:  R1 ← R2 - R3
i2:  R4 ← MEM [R1]
i3:  R6 ← R4 + R1
i4:  R0 ← R2 / R6
i5:  R7 ← MEM [R9]
i6:  R8 ← R7 + R2
i7:  R7 ← R7 + R6
```

Az utasítássorozatot lefordítjuk egy olyan VLIW processzorra, melyben az alábbi utasítások helyezhetők el egy utasításcsoportban:

- 2 db memóriaművelet (végrehajtási idő: 3 órajel, iterációs idő: 1 órajel)
- 2 db egész aritmetikai művelet, vagy ugró utasítás (végrehajtási idő: 1 órajel)

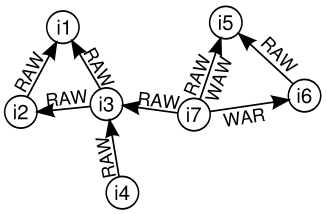
(a) Ütemezze a megadott utasítássorozatot a megadott VLIW processzoron! Határozza meg az utasításcsoportok tartalmát, és a csoportok végrehajtásának idejét!

(b) Mennyivel gyorsabb az így kapott program, mintha hagyományos 1-utas processzoron futtatnánk (pipeline nélkül)?

(c) Hány utasításcsoportot állít elő a fordító klasszikus, és hányat dinamikus VLIW architektúra esetén?

Megoldás

(a) A párhuzamosan végrehajtható utasítások összeválogatásához érdemes először a függőségi viszonyokat felderíteni. Az alábbi ábrán látható függőségi gráfban a csomópontok az utasítások, a csomópontból induló irányított élek pedig azt jelzik, hogy az utasítás melyik másik utasítással áll adat-egymásrahatásban:



Ebben a példában már találunk WAW és WAR egymásrahatásokat is, természetesen azokat is figyelembe kell venni az ütemezés során.

A függőségi gráfból könnyen kiolvasható, hogy az első órajelben az i1 és az i5 hajtható végre egyszerre, hiszen azok nem függenek senkitől. i5 eredménye csak 3 órajelciklus múlva áll majd rendelkezésre, így a csak tőle függő i6 csak a 4. ütemben válik végrehajthatóvá. Az i1 által kiszámolt címről a memóriaművelet a 2. ütemben indul, a tőle függő i3 így csak az 5. ütemben hajtható végre, az i3-tól függő i4 és i7 pedig csak a 6.-ban (de akkor egyszerre). Az eredmény tehát (a ki nem töltött helyeken NOP értendő):

	Mem 1	**Mem 2**	**Egész 1**	**Egész 2**
1:	i5		i1	
2:		i2		
3:				
4:			i6	
5:			i3	
6:			i4	i7

(b) Ha egyszerűen összeadjuk minden utasítás végrehajtási idejét, 11 jön ki, szemben az előző pontban, a VLIW megoldásra kiszámolt 6 órajellel.

(c) A klasszikus VLIW architektúra nem képes az egymásrahatások kezelésére (sem a detektálására, sem a megfelelő számú szünet beiktatására), ilyenkor a fordítóprogramnak kell a kellő számú szünetet a programba elhelyeznie. A fordító az előbbi táblázat minden sorát a programba fordítja (a csupa NOP sort is), tehát a program 6 utasításcsoportból fog állni. Dinamikus VLIW esetben az egymásrahatásokat a processzor kezeli, ilyenkor a csupa NOP sort nem kell a programnak tartalmaznia, mert a processzor észre fogja venni, hogy a memóriaműveletek miatt szünetre lesz szüksége. Ekkor tehát 5 utasításcsoportból áll a program.

18.4. Feladatok az elágazásbecslés témakörében

18.4.1. Feladat

A processzor a program végrehajtása közben az alábbi sorrendben, az alábbi címeken lévő feltételes ugró utasításokat értékeli ki (a zárójelben ”T” jelzi, ha az ugrás ténylegesen bekövetkezett, és ”N”, ha nem):

- 464 (T), 543 (N), 777 (N), 543 (N), 777 (N), 464 (T), 777 (N), 464 (T), 543 (T)

A processzor egyszerű, 2 bites állapotgépre alapozott dinamikus elágazásbecslést használ, és 4 ugró utasítást tud követni.

A feladat a következők:

(a) Hány bitet foglal a PHT?

(b) Írja fel a PHT tartalmát minden egyes feltételes ugrás végrehajtása után! Jelezze, hogy a becslő eltalálta-e az ugrás kimenetelét! Kezdetben minden állapotgép értéke 1.

Megoldás

(a) Mivel 4 ugró utasítást tud a processzor követni, 4 bejegyzéses lesz a PHT. Minden egyes bejegyzéshez egy 2 bites állapotgép tartozik (értéke: 0: nagy eséllyel nem fog ugrani - 3: nagy eséllyel ugrani fog), vagyis a PHT mérete 8 bit.

(b) Azt, hogy az adott címeken lévő ugró utasításokhoz melyik PHT bejegyzés tartozik, az utasítások címeinek 4-el való osztási maradéka határozza meg, azaz:

- 0, 3, 1, 3, 1, 0, 1, 0, 3

Ezután egyszerű a dolgunk: az állapotgép először becsül (ha 0 vagy 1 az értéke, az ugrási feltétel meghiúsulását (N), ha 2 vagy 3 az értéke, a feltétel teljesülését (T) becsüli), majd a tényleges bekövetkezés függvényében frissíti az állapotgépet (a számlálót csökkenti, ha nem volt ugrás, és növeli, ha volt – persze egy 2 bites számlálót nem lehet 0 alá és 3 fölé vinni). Ezek ismeretében az események hatására a PHT a következőképpen alakul (**S** a sikeres, **H** a helytelen becslést jelöli):

Bejegyzés	Kezdetben	1.	2.	3.	4.	5.	6.	7.	8.	9.
0	1	**2 H**	2	2	2	2	**3 S**	3	**3 S**	3
1	1	1	1	**0 S**	0	**0 S**	0	**0 S**	0	0
2	1	1	1	1	1	1	1	1	1	1
3	1	1	**0 S**	0	**0 S**	0	0	0	0	**1 H**

18.4.2. Feladat

A processzor a program végrehajtása közben az alábbi sorrendben, az alábbi címeken lévő feltételes ugró utasításokat értékeli ki (a zárójelben ”T” jelzi, ha az ugrás ténylegesen bekövetkezett, és ”N”, ha nem):

- 464 (T), 543 (N), 777 (N), 543 (N), 777 (N), 464 (T), 777 (N), 464 (T), 543 (T)

A processzor korrelált elágazásbecslést alkalmaz globális, 2 bites állapotgépeket tartalmazó PHT-vel, úgy, hogy az utolsó 2 feltételes ugrás kimenetelét tudja nyomon követni.

A feladat a következők:

(a) Hány bitet foglal a PHT?

(b) Írja fel a PHT tartalmát minden egyes feltételes ugrás végrehajtása után! Jelezze, hogy a becslő eltalálta-e az ugrás kimenetelét! Kezdetben minden állapotgép értéke 1, a globális ugrási előzmény regiszter értéke binárisan 11.

Megoldás

(a) A globális ugrási előzmény regiszter 2 bites, hiszen az az utolsó két feltételes ugrás kimenetelét tárolja. A globális ugrási előzmény regiszter minden értékéhez tartozik egy 2 bites állapotgép, mely annak a becslésére használatos, hogy a globális ugrási előzmény regiszter adott tartalma (vagyis az előző két ugrás eredménye) mellett mennyire valószínű a következő feltételes ugrás ugrási feltételének teljesülése. Azaz a PHT mérete 8 bit.

(b) Azt, hogy az adott címeken lévő ugró utasításokhoz a PHT melyik bejegyzését használjuk a becsléshez, a globális ugrási előzmény regiszter határozza meg. A becslés után a globális ugrási előzmény regisztert balra kell shiftelni (a legszélső bit elvész), és az ugrás tényleges kimenetelét kell a másik oldalon behozni. Tehát, ha a globális ugrási előzmény regiszter 11-ből indul, a felsorolt ugrások során a becsléshez az alábbi értékeket fogja tartalmazni (zárójelben decimálisan):

- 11 (3), 11 (3), 10 (2), 00 (0), 00 (0), 00 (0), 01 (1), 10 (2), 01 (1)

Ezután egyszerű a dolgunk: az globális ugrási előzmény regiszter meghatározza, hogy a PHT hányadik bejegyzésével kell dolgoznunk. A PHT megfelelő bejegyzésében lévő állapotgép először becsül (ha 0 vagy 1 az értéke, az ugrási feltétel meghiúsulását (N), ha 2 vagy 3 az értéke, a feltétel teljesülését (T) becsüli), majd a tényleges bekövetkezés függvényében frissíti az értékét (a számlálót csökkenti, ha nem volt ugrás, és növeli, ha volt – persze egy 2 bites számlálót nem lehet 0 alá és 3 fölé vinni). Ezek ismeretében az események hatására a PHT a következőképpen alakul (**S** a sikeres, **H** a helytelen becslést jelöli):

Bejegyzés	Kezdetben	1.	2.	3.	4.	5.	6.	7.	8.	9.
0	1	1	1	1	**0 S**	**0 S**	**1 H**	1	1	1
1	1	1	1	1	1	1	1	**0 S**	0	**1 H**
2	1	1	1	**0 S**	0	0	0	0	**1 H**	1
3	1	**2 H**	**1 H**	1	1	1	1	1	1	1

18.4.3. Feladat

Egy processzorban az ugrási címek becslésére BTB-t használnak. A BTB találati arány (vagyis hogy egy ugrási utasításra egyáltalán tartalmaz becslést) 90%. BTB találat esetén a becslés pontossága 90%. Ha az ugró utasítás nincs a BTB-ben, 4 ciklus, ha benne van, de rossz a becslés, 3 ciklusnyi késleltetést szenved a futó program végrehajtása. A programban a feltételes elágazások aránya 15%. Most feltesszük, hogy semmi más (semmilyen más egymásrahatás) nem lassítja a program futását, és ha nincs ugrás, a pipeline minden ciklusban végezni tud egy utasítás végrehajtásával.

(a) Átlagosan hány ciklusonként végez a pipeline egy utasítás végrehajtásával?

(b) Hogy viszonyul az eredmény a BTB nélküli esethez, amikor minden ugró utasítás egységesen 2 szünet beiktatását vonja maga után?

Megoldás

(a) Az utasítások 85%-a nem ugrás, ezekből minden ciklusban végez 1. Az utasítások 15%-a ugrás, ezen belül 10% nincs a BTB-ben (1+4 késleltetés a végrehajtási idő), 90% benne van, de 10% eséllyel rosszul becsül (1+3 késleltetés a végrehajtási idő). A BTB-ben lévő, jól becsült ugrási című ugró utasítások végrehajtási ideje 1. Összegezve:

$$0.85 \cdot 1 + 0.15 \cdot (0.1 \cdot 5 + 0.9 \cdot 0.1 \cdot 4 + 0.9 \cdot 0.9 \cdot 1) = 1.1005 \text{ ciklus/utasítás}$$

(b) Ebben az esetben az eredmény:

$$0.85 \cdot 1 + 0.15 \cdot 3 = 1.3 \text{ ciklus/utasítás},$$

vagyis lassabb, mint BTB-vel.

18.4.4. Feladat

Az alábbi C kód kiszámolja egy lebegőpontos számokat tartalmazó tömb elemeinek átlagát:

```
avg = 0;
for (int i = 0; i < N; i++)
    avg += x[i];
avg /= N;
```

(a) Írja át a fenti C kód ciklusát a könyvben használt pszeudo-assembly nyelvre!

(b) A naív implementációhoz hány utasításcsoport szükséges a ciklus végrehajtásához egy olyan VLIW processzoron, melyben az alábbi utasítások helyezhetők el egy utasításcsoportban:

- 2 db memóriaművelet,
- 2 db lebegőpontos művelet,
- 2 db egész aritmetikai művelet, vagy ugró utasítás?

(Most feltételezzük, hogy minden művelet 1 órajel alatt végrehajtható)

Adja meg az utasítások ütemezését is!

(c) Alakítsa át a ciklust úgy, hogy a műveletek szoftver pipeline szerint végrehajthatók legyenek! Adja meg az utasítások ütemezését a szoftver pipeline kihasználásával! Hány utasításcsoportra van szükség?

Megoldás

(a) A C kód ciklusa az alábbi utasítássorozattal írható le (R0 kezdetben az x tömb elejére, R1 a tömb végére mutat, D0-ba olvassuk be a memóriából a következő elemet, D1 tárolja az avg változó értékét):

```
i1: label: D0 ← MEM [R0]
i2:        D1 ← D1 + D0
i3:        R0 ← R0 + 8
i4:       JUMP label IF R0≠R1
```

(b) Az előbbi naív implementáció ütemezése az alábbiak szerint alakul:

	Egész 1	Egész 2	Mem 1	Mem 2	FP 1	FP 2
1:	i3		i1			
2:	i4				i2	

Tehát 2 utasításcsoportra van szükség, mely annyiszor hajtódik végre, ahány kört megy a ciklus, így jelen esetben a teljes ciklus $2 \cdot N$ ütem alatt fut le.

(c) Szoftver pipeline-hoz azt az ötletet kell alkalmaznunk, hogy amíg az i. elem betöltését végezzük, azzal együtt végezhetjük az i-1. iteráció műveletét (avg növelése), hiszen addigra az i-1. elem már be lesz töltve. Tehát az eredeti ciklus az alábbi elemi műveletekből építhető fel:

```
tmp_0 ← Load x[0];
for (int i = 1; i < N; i++) {
    tmp_i ← Load x[i];
    avg += tmp_{i-1};
}
avg += tmp_{N-1};
```

Figyeljük meg, hogy tmp_i helyett elegendő egyetlen változót használni (*tmp*), hiszen a ciklusmag mindkét utasítása egyszerre, egymással párhuzamosan hajtódik végre, így a *tmp* régi értékét használja az összeadás, és a *tmp* új értéke a tömbből beolvasott új elem lesz. Most D0 legyen a memóriából beolvasott elem (tehát a *tmp*), és D1 az eredmény értéke.

Vegyük észre továbbá, hogy a tömb aktuális elemére mutató pointer (R0) növelése, és az ugrási feltétel kiértékelése is párhuzamosan történhet, de ehhez az kell, hogy az ugrás ne ”R0≠tömb vége” esetén teljesüljön, hiszen ennek kiértékelésével párhuzamosan nő R0 értéke, azaz még az R0 régi értékével számol a feltétel. A helyes ugrási feltétel tehát: ”R0≠tömb vége-8” lesz. A ”tömb vége - 8”-at most R1 fogja tárolni. Egy további művelet megspórolása érdekében pedig R0-t kezdetben nem a nulladik, hanem az első elemre állítjuk.

Az utasítássorozat a következő lesz ({} zárójelek az egy csoportba szánt utasításokat jelölik) .

```
i1:     D0 ← MEM [R0-8]
label: {i2: D0 ← MEM [R0], i3: D1 ← D1 + D0,
         i4: R0 ← R0 + 8, i5: JUMP label IF R0≠R1}
i6:     D1 ← D1 + D0
```

Az i1 az előbbi, elemi műveleteket tartalmazó kód for ciklus előtti utasításához, i6 a for ciklus utáni utasításához tartozik. Lényegében ezek a szoftver pipeline feltöltését és kiürítését szolgálják.

Az ütemezés:

	Egész 1	Egész 2	Mem 1	Mem 2	FP 1	FP 2
1:			i1			
2:	i4	i5	i2		i3	
3:					i6	

Tehát csak 3 csoportra volt szükség. Ebből a 3 csoportból csak a 2. a ciklus magja, a feltétel kiértékelése és maga az ugrás csupán egyetlen utasításcsoportot igényel, tehát a ciklus egy köre egy órajel idő alatt lefut, a teljes ciklus pedig ($N-1$ kör, +1 utasításcsoport előtte, +1 utána) $N+1$ órajel alatt fut le.

18.4.5. Feladat

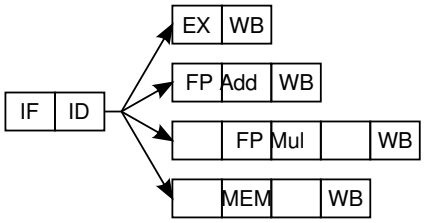
Az alábbi C kód kiszámolja egy diszkrét valószínűségi változó várható értékét:

```
ex = 0;
for (int k = 0; k < N; k++)
    ex += k * p[k];
```

A műveletek végrehajtási ideje a következő:

- Egész aritmetikai műveletek: 1 órajel
- Load/Store: 3 órajel
- Lebegőpontos szorzás: 4 órajel
- Lebegőpontos összeadás: 2 órajel

A pipeline felépítését az alábbi ábra mutatja:



Az utasítások sorrenden kívüli befejezése megengedett, amennyiben a program értelmén nem változtat. Több WB is végbemehet egyszerre, feltéve, hogy nem ugyanazt a regisztert írják.

(a) Írja át a ciklus magját a tárgyban használt pszeudo-assembly nyelvre!

(b) Hány órajel szükséges a ciklusmag egyszeri végrehajtásához?

(c) Optimalizálja a ciklust ciklus-kifejtéssel! Hányszoros ciklus-kifejtésre van szükség? Adja meg a kifejtett ciklus magját pszeudo-assembly nyelven! Hány órajel szükséges az *eredeti* ciklusmag egyszeri végrehajtásához?

Megoldás

(a) Feltéve, hogy R0 tartalmazza a mutatót a p tömb aktuális elemére, D0 tárolja ex-et, és D1 a ciklusváltozó értékét lebegőpontosan, a ciklusmag a következőképpen alakul:

```
i1:  D2 ← MEM [R0]
i2:  D3 ← D2 * D1
i3:  D0 ← D0 + D3
i4:  D1 ← D1 + 1.0
i5:  R0 ← R0 + 8
```

(b) Írjuk fel az ütemezést a feladatban adott pipeline-al:

Utasítás	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.
i1	IF	ID	MEM0	MEM1	MEM2	WB							
i2		IF	ID	A^*	A^*	M0	M1	M2	M3	WB			
i3			IF	F^*	F^*	ID	A^*	A^*	A^*	A0	A1	WB	
i4						IF	F^*	F^*	F^*	ID	A0	A1	WB
i5										IF	ID	EX	WB

Tehát 13 órajelre van szükség. Mivel a 11. órajelben már a ciklus következő körének az utasítása el is kezdhető, ezért most inkább vegyük úgy, hogy 10 órajelre van szükség a ciklus egy körének végrehajtásához.

(c) Háromszoros cikluskifejtéssel a ciklus három körét számoljuk ki egyszerre. Mivel több utasításunk van, nagyobb a mozgástér a csere-berére, nagyobb eséllyel tudjuk a RAW függő utasításokat eltávolítani egymástól a szünetek elkerülése érdekében:

```
i1:   D2 ← MEM [R0]
i2:   D4 ← MEM [R1]
i3:   D6 ← MEM [R2]
i4:   D3 ← D2 * D1
i5:   D1 ← D1 + 1.0
i6:   R0 ← R0 + 8
i7:   D5 ← D4 * D1
i8:   D1 ← D1 + 1.0
i9:   R1 ← R1 + 8
i10:  D7 ← D6 * D1
i11:  D1 ← D1 + 1.0
i12:  R2 ← R2 + 8
i13:  D0 ← D0 + D3
i14:  D0 ← D0 + D5
i15:  D0 ← D0 + D7
```

Felírhatnánk az utasítások ütemezését, de felesleges. Az utasításokat sikerült úgy átrendezni, hogy a RAW függő utasítások távolsága nagyobb, mint a várt operandus előállításához szükséges idő (memóriaműveleteknél 3, FP szorzásnál 4, FP összeadásnál 2, int műveletnél 1 órajel). Ennek köszönhetően az utasítássorozat végrehajtása alatt egyetlen szünet beiktatására sem lesz szükség. 15 utasításunk van, tehát a for ciklus 3 köre 15 órajelciklus alatt végrehajtódott (vagy legalábbis 15 órajel alatt minden utasítás elkezdődött – majd befejeződnek, miközben a for ciklus következő 3 körének zajlik a végrehajtása). Ezzel elértük az 5 órajel / ciklus egy köre sebességet.

Az utasítások ügyes rendezgetésével kiderül, hogy kétszeres cikluskifejtés is elegendő ugyanehhez a sebességhez:

```
i1:   D2 ← MEM [R0]
i2:   R0 ← R0 + 8
i3:   D4 ← MEM [R0]
i5:   D3 ← D2 * D1
i6:   D1 ← D1 + 1.0
i4:   R0 ← R0 + 8
i7:   D5 ← D4 * D1
i8:   D1 ← D1 + 1.0
i9:   D0 ← D0 + D3
i10:  D0 ← D0 + D5
```

V. rész

Párhuzamos feldolgozás

19. fejezet

SIMD feldolgozás

19.1. Flynn taxonómia és az SIMD feldolgozás

A ma elterjedt működési elv szerint üzemelő, a jelenlegi technológiával gyártott processzorok utasításfeldolgozó sebességét nem lehet tetszőlegesen növelni (lásd: ”A szuperskalár pipeline” rész 1. fejezete):

- Nagyon mély és/vagy széles pipeline-ok esetén a sok átlapoltan/párhuzamosan futó utasítás között az egymásrahatások előfordulásának gyakorisága az átlapoltan futó utasítások számával meredeken nő (elég valószínűtlen, hogy az utasítássorozat csupa egymástól független műveletből áll).
- Nagyon széles pipeline-hoz sok műveleti egységre van szükség, melyek között a RAW egymásrahatások kezelésére szolgáló forwarding utak száma a pipeline mélységével négyzetesen nő.
- Hiába mély a pipeline, a ciklusidő nem lehet tetszőlegesen rövid, részben fogyasztási okokból, részben pedig azért, mert a pipeline regiszterek írása/kiolvasás bele kell, hogy férjen egy ciklusidőbe.
- A mély pipeline-nak megvan az a hátránya, hogy a spekulatív döntések (pl. elágazásbecslés) kimenetele sok fázissal később derül ki, eközben sok, potenciálisan felesleges utasítás végrehajtása kezdődhet el.

A számítási teljesítmény további növelésének tehát nem az utasítás-pipeline növelése/mélyítése a legkényelmesebb módja. Egy lehetséges alternatíva, ha az utasítások nem egy, hanem több adaton végeznek műveleteket egyszerre. Ezt a feldolgozási elvet Single Instruction Multiple Data, röviden SIMD feldolgozásnak nevezzük. Az SIMD jelölést Flynn 1966-ban vezette be, amikor a számítógép architektúrákat klasszifikálta a szerint, hogy az utasítások és adatok viszonya hogyan alakul. Flynn 4 kategóriát definiált:

- **SISD** (Single Instruction Single Data): ebben az architektúrában egyetlen utasítássorozat végrehajtása zajlik, az utasítások pedig skalárokon végeznek műveleteket. Ide sorolhatók pl. az egy processzoros PC-k.
- **SIMD** (Single Instruction Multiple Data): továbbra is egyetlen utasítássorozat végrehajtása zajlik, de az egyes utasítások nem egy, hanem sok adaton végzik el a hozzájuk tartozó műveleteket. Ide tartoznak a vektorprocesszorok és a tömbprocesszorok is, és bizonyos szempontból a grafikus kártyákon található GPU-k is.
- **MIMD** (Multiple Instruction Multiple Data): több utasítássorozat végrehajtása zajlik egyidejűleg, melyben a különböző utasítások különböző adatokon végeznek műveleteket (tehát egyidejűleg több utasítás operál egyidejűleg több adaton). Ide tartoznak a multiprocesszoros rendszerek.
- **MISD** (Multiple Instruction Single Data): több utasítássorozat végrehajtása zajlik egyidejűleg, de a különböző utasítások ugyanazokon az adatokon végeznek műveleteket. Leginkább hibatűrő rendszerekben használatosak (pl. az űrhajózásban). Az adatokat több, különböző rendszer is feldolgozza, majd ezek eredményeit összevetik, és ha nem ugyanazt kapják, akkor hiba történt.

19.2. Vektorprocesszorok

A vektorprocesszorok az SIMD feldolgozás klasszikus megtestesítői. A vektorprocesszorok a skalár adattípus mellett bevezetik a több skalárt egybefogó *vektor* adattípust, és az ezek feldolgozására szolgáló *vektorkezelő utasítások*at. Manapság szinte minden szuperszámítógép vektorprocesszorokból épül fel.

19.2.1. A vektor adattípus és használatának előnyei

A vektorprocesszorokban a vektorműveletek a vektor minden elemére vonatkoznak, vagyis egyenértékűek egy teljes ciklussal. Hasonlítsuk össze egy egyszerű algoritmust megvalósítását klasszikus skalár, illetve vektorprocesszor esetén.

C program:

```
for (i=0; i<64; i++)
    C[i] = A[i] + B[i];
```

Klasszikus (skalár) megvalósítás:

```
       R4← 64
loop:  D1 ← MEM[R1]
       D2 ← MEM[R2]
       D3 ← D1 + D2
       MEM[R3] ← D3
       R1 ← R1 + 8
       R2 ← R2 + 8
       R3 ← R3 + 8
       R4 ← R4 - 1
       JUMP loop IF R4!=0
```

Vektorprocesszorral:

```
VLR ← 64
V1 ← MEM[R1]
V2 ←MEM[R2]
V3 ← V1 + V2
MEM[R3] ← V3
```

(A példában a VLR a vektor hossz regisztert jelöli, ami meghatározza, hogy az utána következő vektor utasítások hány elemű vektorokon dolgozzanak. A vektorregisztereket V-vel jelöljük.)

A példa segít azonosítani a vektorprocesszorok előnyeit:

- A program kódja sokkal tömörebb.
- Nincs szükség a ciklusra. Elsőre talán nem is látszik, hogy ez mekkora nyereség:
  - Skalár megközelítés esetén, ahogy pörög a végrehajtás a ciklusban (a loop címke és a JUMP között), minden egyes körben újra és újra le kell hívni és dekódolni a ciklus magjában lévő utasításokat (összesen 64-szer!). Vektorprocesszor esetén nincs ciklus, a vektorokon operáló utasításokat csak egyszer kell lehívni és dekódolni.
  - Skalár megközelítés esetén a ciklus minden egyes körében fellép a procedurális egymásrahatás, tehát a processzornak elágazásbecsléssel kell megtippelnie az ugrás kimenetelét és címét, összesen 64-szer. A vektorprocesszoros változatban nincs is elágazás.
- A vektorműveletek implicit feltételezik, hogy a vektor elemei függetlenek, tehát a vektor elemei között nincs adat-egymásrahatás, így
  - a hardvernek csak a vektorműveletek közötti egymásrahatást kell ellenőriznie, az elemek közöttit nem,
  - a műveleti egységek többszörözésével vagy igen mély pipeline alkalmazásával nagy teljesítmény érhető el (lásd később).

19.2.2. Vektorprocesszorok felépítése

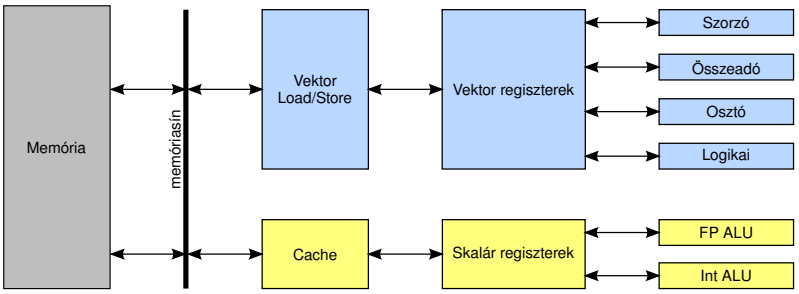
A vektorprocesszorok túlnyomó többsége két kategóriába sorolható:

- A regiszter-regiszter vektorprocesszorokban a vektorkezelő utasítások operandusai (a Load-tól és a Store-tól eltekintve) vektor regiszterek. Tehát ahhoz, hogy a memóriában lévő vektorok között műveleteket tudjunk végezni, előbb az adatokat a memóriából be kell tölteni a vektorregiszterekbe.

- A memória-memória vektorprocesszorok közvetlenül a memóriában lévő adatokkal tudnak műveleteket végezni.

A regiszter-regiszter kialakítás előnye, hogy a számítások operandusai és részeredményei a műveleti egységekhez közel, a processzorban helyezkednek el, nem kell azokat minden egyes alkalommal a memóriából beolvasni ill. oda visszaírni. A továbbiakban csak a regiszter-regiszter vektorprocesszorokkal foglalkozunk, mivel a 80-as évek óta minden kereskedelmi forgalomban kapható vektor processzor ilyen felépítésű.

A vektorprocesszorok felépítését és főbb összetevőit a 19.1. ábra ábrázolja.



19.1. ábra. Vektorprocesszorok felépítése

Az ábrán a skalár és a vektor számítások részére külön regiszterek és műveleti egységek állnak rendelkezésre. Vannak olyan vektorprocesszorok is, melyben a skalár műveleteket is a vektor műveleti egységek végzik el. A vektorregiszterek mérete fix, tipikusan 64 és 4096 közé esik. A memóriában tárolt skalár és vektor adatok gyors elérése különböző módon biztosított:

- A skalár adatok számára egy cache memória áll rendelkezésre.

- Vektorokat nem tárolnak a cache memóriában, de ez nem jelenti azt, hogy elérésük lassú lenne, egyszerűen csak más technikával, a vektorok memóriabeli szabályos elhelyezkedésének kihasználásával, és a memória speciális kialakításával teszik gyorssá a memóriaműveleteket.

Memóriakezelés

A vektorok memóriából regiszterbe töltése során a memóriaműveletek olyan speciális mintázatot követnek, amit ki lehet használni a teljesítmény érdekében.

Tipikusnak mondható, hogy a memóriaműveletek több órajelet vesznek igénybe, a cím megérkezése és a kapcsolódó adat buszon való megjelenése között több órajel telik el. A vektorok betöltése azonban sok memóriaműveletet, a vektor összes elemének a betöltését jelenti. Ha a rendszermemóriát (a teljes címtartományt) több fizikai memóriamodul (ú.n. memóriabank) segítségével valósítjuk meg, akkor a processzor sok időt nyerhet azzal, hogy amíg egy vektorelem betöltése kapcsán az azt tároló memóriabank válaszára vár, addig egy másik vektorelem betöltését is elkezdheti, értelemszerűen egy olyan elemét, amely egy másik memóriabankban található. A legcélszerűbb a memóriabankokat *fésű*sen hozzárendelni a címtérhez (interleaving). Ez azt jelenti, hogy n memóriabank esetén az első a 0, n-1, 2n-1, ... szavakat, a második az 1, n, 2n, ... szavakat, a harmadik a 2, n+1, 2n+1, ... szavakat tárolja, stb.

Nézzük a következő példát. Tartson minden memóriaművelet 5 órajelig, és a címtartomány legyen lefedve 6 memóriabankkal, fésűs címzést alkalmazva. A CPU 1 órajel alatt 1 címet tud kiadni, és 1 adatot tud fogadni. A betöltendő vektor 64 elemű, az elemek a memóriában a 15636. bájttól kezdve sor-folytonosan helyezkednek el, egy elem 4 bájtot foglal.

	Bank					
Órajel	**0**	**1**	**2**	**3**	**4**	**5**
0		15636				
1		fogl.	15640			
2		fogl.	fogl.	15644		
3		fogl.	fogl.	fogl.	15648	
4		Adat[0]	fogl.	fogl.	fogl.	15652
5	15656		Adat[1]	fogl.	fogl.	fogl.
6	fogl.	15660		Adat[2]	fogl.	fogl.
7	fogl.	fogl.	15664		Adat[3]	fogl.
8	fogl.	fogl.	fogl.	15668		Adat[4]
9	Adat[5]	fogl.	fogl.	fogl.	15672	
10		Adat[6]	fogl.	fogl.	fogl.	15676
11	15680		Adat[7]	fogl.	fogl.	fogl.
12	fogl.	15684		Adat[8]	fogl.	fogl.
13	fogl.	fogl.	15688		Adat[9]	fogl.

A fésűs címzésnek köszönhetően a 4. órajeltől kezdve órajelenként érkeznek az adatok a processzorba, tehát egy kezdeti rövid szakasztól eltekintve sikerült kiküszöbölni a memória késleltetését (minél nagyobbak a vektorok, annál jobban leamortizálódik a kezdeti késleltetési idő a vektor teljes beolvasási idejéhez képest).

Műveleti egységek

A vektorműveletek végrehajtása kétféleképpen gyorsítható:

1. **A műveleti egységek többszörözésével**: Mivel a vektor elemei függetlenek, a vektorműveletek a vektorok minden elemére egymástól függetlenül elvégezhetők. Ha lenne elegendő számú műveleti egységünk, akár a vektor minden elemét külön műveleti egység számolhatná, így a vektorművelet nem tartana tovább egy skalár műveletnél. Ilyen nagy számú műveleti egység (64-4096 db) processzorba építése sajnos általában nem kivitelezhető. Minél több műveleti egység van (tipikusan 4-16), annál gyorsabban hajthatók végre a vektorműveletek.

2. **Mély adat-pipeline alkalmazásával**: Ha nincs is annyi műveleti egység, mint amekkorák a vektorok, akkor is lehet a végrehajtást hatékonyan végezni. Mivel a vektor elemei függetlenek, nincs közöttük adat-egymásrahatás, így akár nagyon mély pipeline alkalmazásával a vektor egymás utáni elemein átlapoltan végezhetjük el a kívánt műveletet.

Ahhoz, hogy az adat-pipeline-t részleteiben megismerjük, nem árt megismerni a lebegőpontos számok formátumát és az aritmetikai műveletek végrehajtásának módját. A lebegőpontos számok reprezentációján az IEEE 754 szabvány rögzíti, az alábbi módon:
Szám = $(-1)^s \cdot c \cdot b^q$, ahol: s: előjel, c: mantissza, b: bázis, q: karakterisztika. A bázis többnyire megegyezik a tároláshoz használt számrendszerrel, vagyis 2-vel. Most nézzük meg, hogy a lebegőpontos összeadás hogyan történik (nagyon nagy vonalakban):

- Először ellenőrizzük, hogy valamelyik operandus nulla-e,
- a két operandus *illesztjük*, vagyis azonos karakterisztikára hozzuk,
- az összeadást elvégezzük,
- az eredményt a szabvány szerinti normálalakra hozzuk és a tárolási pontosság ismeretében csonkoljuk.

Hasonlóan, a lebegőpontos szorzás is több fázisra osztható, például:

- Ellenőrizzük, hogy valamelyik operandus nulla-e,
- a karakterisztikákat összeadjuk,
- a mantisszákat összeszorozzuk,
- az eredmény előjelbitjét előállítjuk,
- az eredményt a szabvány szerinti normálalakra hozzuk és a tárolási pontosság ismeretében csonkoljuk.

Természetesen ennél több fázisra is bonthatjuk a műveleteket, ha ezek közül valamelyik lépést tovább bontjuk. A vektor műveleti egységek a hozzájuk rendelt vektorelemek feldolgozását átlapoltan végezhetik: például a szorzó, amíg az egyik elemre már az utolsó fázist végzi, a következő elemre ezzel egy időben előállítja az előjelbitet, miközben egy harmadik elemre éppen a mantisszákat szorozza, stb. A következő táblázat a V2=V0+V1 művelet végrehajtását mutatja be, 4 fázisú összeadó pipeline-t és 8 elemű vektorokat feltételezve:

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
V2[0] ← V0[0] + V1[0]	A0	A1	A2	A3							
V2[1] ← V0[1] + V1[1]		A0	A1	A2	A3						
V2[2] ← V0[2] + V1[2]			A0	A1	A2	A3					
V2[3] ← V0[3] + V1[3]				A0	A1	A2	A3				
V2[4] ← V0[4] + V1[4]					A0	A1	A2	A3			
V2[5] ← V0[5] + V1[5]						A0	A1	A2	A3		
V2[6] ← V0[6] + V1[6]							A0	A1	A2	A3	
V2[7] ← V0[7] + V1[7]								A0	A1	A2	A3

Ha több műveleti egységünk van, és belül mindegyik pipeline szervezésű, akkor a feldolgozás a következő ütemezés szerint történik (8 elemű vektorokat, 2 műveleti egységet és 4 fázisú összeadó pipeline-t feltételeztünk):

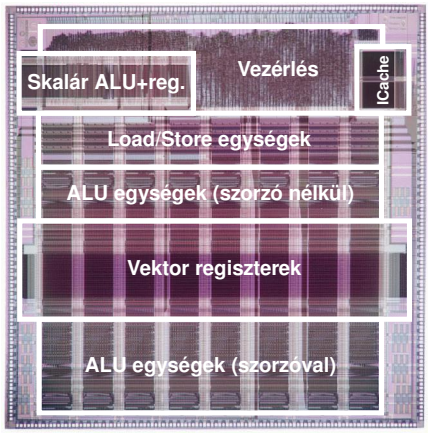
	1	**2**	**3**	**4**	**5**	**6**	**7**
V2[0] ← V0[0] + V1[0]	A0	A1	A2	A3			
V2[1] ← V0[1] + V1[1]	A0	A1	A2	A3			
V2[2] ← V0[2] + V1[2]		A0	A1	A2	A3		
V2[3] ← V0[3] + V1[3]		A0	A1	A2	A3		
V2[4] ← V0[4] + V1[4]			A0	A1	A2	A3	
V2[5] ← V0[5] + V1[5]			A0	A1	A2	A3	
V2[6] ← V0[6] + V1[6]				A0	A1	A2	A3
V2[7] ← V0[7] + V1[7]				A0	A1	A2	A3

A két műveleti egységgel tehát 11 helyett csak 7 ciklusra volt szükség, a különbség a vektor méret növelésével tart a kétszereshez.

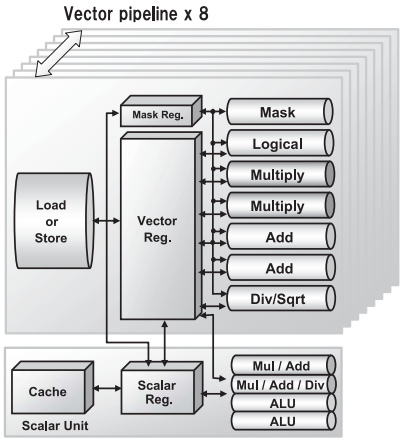
Nem lehet eléggé hangsúlyozni, hogy ennél a fajta adatpipeline-nál nincs egymásrahatásokat vizsgáló és kezelő logika, mivel a vektorok elemei függetlenek egymástól. Emiatt a pipeline szinte tetszőleges mély lehet, a mélység növelése minden esetben növeli a számítási teljesítményt. Az egyetlen korlátozó tényező az, hogy hány elemi fázisra vagyunk képesek felbontani egy aritmetikai műveletet.

Példaként a 19.2. ábra a Berkely egyetem T0 (Torrent-0, 1995) mikroprocesszorának szilíciumszeletét mutatja be, melyen jól elkülöníthető 8 függőleges sáv, melyek mindegyike a 32 hosszú vektor regiszterekből 4 elem kezeléséért felelős. Minden sávban 2 ALU található, melyek közül csak az egyik tartalmaz szorzó egységet. Így, ha nem szorzásról van szó, kétszeres számítási teljesítmény érhető el. A processzorban külön regisztertároló és ALU található a skalár műveletek számára.

Egy másik példa a NEC SX-9 szuperszámítógép vektorprocesszora (19.3. ábra). A NEC SX-9 2008-ban a kapható leggyorsabb szuperszámítógép volt, pl. a német nemzeti meteorológiai központnak van belőle 2 darab az időjárási modellek számításainak támogatására (2011-ben mindkettő 224 processzort és 7 TB memóriát tartalmaz). A NEC SX-9 vektorprocesszora 256 hosszú vektorokkal dolgozik, és 8 pipeline szervezésű feldolgozóegysége van. A vektoraritmetikai egységek 3.2 GHz-en, a skalár egység (4 utas out-of-order) 1.6 GHz-en üzemel.



19.2. ábra. A T0 vektorprocesszor felépítése



19.3. ábra. A NEC SX-9 vektorprocesszor felépítése

19.2.3. Vektorprocesszorok jellegzetes megoldásai

Strip-mining

A vektorprocesszorok vektorregisztereinek mérete hardveresen kötött, ez a Maximum Vector Length (**MVL**). A gyakorlati problémák megoldása során elvétve fordul elő, hogy ez a kötött méret pont megegyezik azzal a vektormérettel, amire éppen a számítás során szükség van. Ha az MVL-nél kisebb vektorokkal szeretnénk dolgozni, akkor be kell állítani a vektorméret regisztert (Vector Length Register, **VLR**) a kívánt értékre. A VLR meghatározza, hogy a további műveleteket a vektorregiszter hány elemére kell elvégezni. Ha a VLR sokkal kisebb, mint az MVL, rövidebb futási időkre számíthatunk.

A másik eset, ha az MVL-nél nagyobb vektorokkal szeretnénk műveleteket végezni. Ebben az esetben a nagy vektorainkat MVL méretűekre kell szeletelni, és ezeken egyenként elvégezni a kívánt műveleteket. Ezt az eljárást hívják *strip-mining*-nak.

Az alábbi példában N méretű vektorokat adunk össze, ahol N futási időben derül ki (feltesszük, hogy értékét az R0 regiszter tartalmazza). Ekkor a fordító az alábbi C kódból a következő vektorizált kódot állítja elő, strip-mining alkalmazásával:

C program:

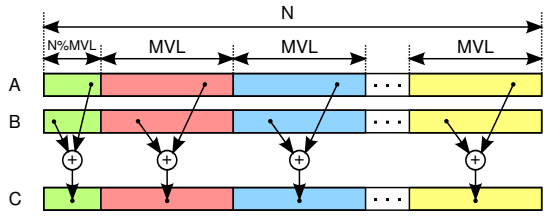
```
for (i=0; i<N; i++)
    C[i] = A[i] + B[i];
```

Vektorizálva:

```
    VLR ← R0 % MVL
```

```
loop:  V1 ← MEM[R1]
       V2 ← MEM[R2]
       V3 ← V1 + V2
       MEM[R3] ← V3
       R4 ← VLR * 8
       R1 ← R1 + R4
       R2 ← R2 + R4
       R3 ← R3 + R4
       R0 ← R0 - VLR
       VLR ← MVL
       JUMP loop IF R0!=0
```

Az N hosszú vektort a ciklus több körben adja össze. Az első utáni körök MVL méretű darabokat, az első pedig a maradékot (N%MVL) dolgozza fel (19.4. ábra). Az első lépésben a VLR a maradék méretével lesz egyenlő, a többi körben pedig MVL-lel. Minden lépés után VLR*8 bájttal léptetjük a 3 tömb pointerét (feltesszük, hogy a vektorelemek 8 bájt hosszúak), valamint csökkentjük az N értékét reprezentáló R0 regisztert a feldolgozott elemek számával, ami éppen VLR. A ciklus akkor lép ki, ha minden elemet feldolgoztunk.



19.4. ábra. Strip-mining vektorösszeadás

Vektorműveletek feltételes végrehajtása

A legtöbb vektorprocesszor rendelkezik egy maszk regiszterrel, amely logikai (1 bites) elemekből áll, és meghatározza, hogy a vektorműveleteket a vektorregiszterek mely elemeire kell elvégezni. A maszk regiszter közvetlenül, vagy vektor összehasonlító utasításokkal állítható be. Példa (kezdetben az A vektor mutatója R0-ban, B vektoré R1-ben, C vektoré R2-ben van):

C program:

```
for (i=0; i<N; i++)
    if (B[i]>0)
        C[i] = A[i] / B[i];
```

Vektorprocesszorral:

```
V1 ← MEM[R1]
MASK ← V1>0
V0 ← MEM[R0]
V2 ← V0 / V1
MEM[R2] ← V2
```

Beállított maszk regiszter esetén a vektorműveletek a processzor fejlettségének függvényében kétféleképpen valósíthatók meg:

1. Egyszerű implementáció: A vektor ALU a műveleteket a vektor minden elemére végrehajtja, de a maszk által letiltott elemeket nem írja vissza az eredményt tároló vektorregiszterbe.

2. Hatékony implementáció: A vektor Load/Store egység és a vektor ALU a maszk által letiltott elemeket kihagyja a memória- ill. az aritmetikai műveletek végrehajtása során. A fenti példában a B vektor beolvasása teljes hosszában megtörténik (a V1 regiszterbe), ez alapján a vektor összehasonlító utasítás előállítja a

maszkot, majd ezután az összes többi művelet már csak a maszk által engedélyezett elemekre fog végrehajtódni. Tehát a 3. sorban a letiltott elemeket már eleve be sem tölti a memóriából, az osztásnál ezeket eleve kihagyja, és a Store is csak az engedélyezett elemeket írja vissza a memóriába. Ez a megoldás nyilván jobban kihasználja az erőforrásokat, és gyorsabb végrehajtási időt eredményez.

Művelet láncolás

A vektorutasítások sorozatát a vektorprocesszorok ugyanolyan utasítás pipeline segítségével hajtják végre, mint a skalár processzorok. A skalár processzorok tárgyalásánál láttuk, hogy a pipeline feldolgozás hatékonyságát a RAW egymásrahatások rontják a legjobban. RAW egymásrahatások természetesen a vektorprocesszorokban is vannak:

```
V1 ← MEM[R1]
V3 ← V1 + V2
V5 ← V3 * V4
```

Hasonló, de skalár műveleteket tartalmazó utasítássorozat esetén az összeadás addig nem kezdhető el, amíg a Load be nem fejeződött, és a addig nem kezdhető el, amíg az összeadás be nem fejeződött. Skalár processzorok esetén a várakozással elvesztegetett idő csökkentésére alkalmas volt a forwarding technika. Forwarding segítségével pl. a szorzásnak nem kellett megvárnia, amíg az összeadás eredménye bekerül a V3 regiszterbe, a V3 értékét közvetlenül az ALU kimenetéről leszedve a szorzás több ciklussal hamarabb volt elkezdhető.

A forwarding vektorprocesszoros megfelelője a művelet láncolás (vector chaining). A vector chaining lényege, hogy a következő (függő) utasításnak nem kell megvárnia az előző számolás végét, ahogy szépen sorban készülnek el az eredmény vektor elemei, azokat máris fel lehet használni a következő művelet elkezdésére. Vagyis amint a Load végzett a V1 első elemének betöltésével, máris indulhat az összeadás első elemének kiszámítása. Hasonlóan, amint van kész eredmény a V3-ban, azokkal rögtön elkezdhető a szorzás, nem kell megvárni, hogy minden elem össze legyen adva V3-ban.

Például 2 Load/Store és 2 aritmetikai műveleti egység esetén, 2 ciklusideig tartó memóriaelérést, 3 ciklusideig tartó pipeline szervezésű lebegőpontos összeadást és 4 ciklusideig tartó lebegőpontos szorzást feltételezve, művelet láncolás alkalmazásával a fenti 3 utasítás végrehajtása a 19.1. táblázatban látható módon ütemezhető.

19.3. SIMD utasításkészlet kiegészítések

A vektor utasítások nem csak professzionális célokra, szuperszámítógépeken, hanem otthoni, multimédiás célokra is hasznosak lehetnek. Ennek megfelelően az utóbbi két évtizedben az általános célú processzorokat is felruházták vektor feldolgozó képességekkel, vektor regiszterek és az ezeken operáló vektor utasítások bevezetésével. Ezek a képességek persze eltörpülnek a vektorprocesszorok képességei mellett:

- nagyon kicsi a vektorméret (a legnagyobb, általános célú processzorban használatos vektor regiszter 256 bites, 2-16 elemmel),
- nincs VLR,
- nincs maszk regiszter,
- a művelet láncolást nem támogatják,
- nincs adatpipeline, a feldolgozás a műveleti egységek többszörözésével történik (annyi műveleti egység kell, amekkora a vektorméret).

Mindezek ellenére ezek a kis méretű vektorok és a hozzájuk kapcsolódó utasítások is jó szolgálatot tesznek, jelentősen gyorsítva néhány speciális, de népszerű alkalmazást:

- Képfeldolgozási feladatokban a képpontok általában 4 komponensre bonthatók (piros, zöld, kék, alfa), így 4 elemű vektorokban tárolhatók, rajtuk vektorutasításokkal műveletek végezhetők. (Példa: fényesség, kontraszt, vagy az operációs rendszer grafikus interfészének megjelenítésére, compositing, stb.)
- 3D grafikai alkalmazásokban és számítógépes játékokban a 3 dimenziós vektorokat homogén koordinátarendszerben 4 koordináta írja le, így a koordináta transzformációkat ezekkel az egyszerű vektorutasításokkal hatékonyan el lehet végezni.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
V1[0] ← MEM[R1+0]	T0	T1								
V1[1] ← MEM[R1+8]	T0	T1								
V1[2] ← MEM[R1+16]		T0	T1							
V1[3] ← MEM[R1+24]		T0	T1							
V1[4] ← MEM[R1+32]			T0	T1						
V1[5] ← MEM[R1+40]			T0	T1						
V3[0] ← V1[0] + V2[0]			A0	A1	A2					
V3[1] ← V1[1] + V2[1]			A0	A1	A2					
V1[6] ← MEM[R1+48]				T0	T1					
V1[7] ← MEM[R1+56]				T0	T1					
V3[2] ← V1[2] + V2[2]				A0	A1	A2				
V3[3] ← V1[3] + V2[3]				A0	A1	A2				
V1[8] ← MEM[R1+64]					T0	T1				
V1[9] ← MEM[R1+72]					T0	T1				
V3[4] ← V1[4] + V2[4]					A0	A1	A2			
V3[5] ← V1[5] + V2[5]					A0	A1	A2			
V1[10] ← MEM[R1+80]						T0	T1			
V1[11] ← MEM[R1+88]						T0	T1			
V3[6] ← V1[6] + V2[6]						A0	A1	A2		
V3[7] ← V1[7] + V2[7]						A0	A1	A2		
V5[0] ← V3[0] * V4[0]						M0	M1	M2	M3	
V5[1] ← V3[1] * V4[1]						M0	M1	M2	M3	
V1[12] ← MEM[R1+96]							T0	T1		
V1[13] ← MEM[R1+104]							T0	T1		
V3[8] ← V1[8] + V2[8]							A0	A1	A2	
V3[9] ← V1[9] + V2[9]							A0	A1	A2	
V5[2] ← V3[2] * V4[2]							M0	M1	M2	M3
V5[3] ← V3[3] * V4[3]							M0	M1	M2	M3

19.1. táblázat. Példa művelet láncolásra

- Számos egyszerű tudományos alapfeladat jól vektorizálható (pl. lineáris egyenletek megoldása, stb.).

A vektorkezelő utasítások az alábbi kategóriákba sorolhatók:

- Vektor-vektor műveletek:
  - Inter-vektor műveletek: vektor operandusokon végeznek elemenként aritmetikai-, bitenkénti-, logikai-, összehasonlító- műveleteket. Pl. elemenkénti összeadás, szorzás, bitenkénti ÉS, min/max, sqrt, stb. Az inter-vektor műveleteket minden vektor utasításkészlet kiegészítés támogatja.
  - Intra-vektor műveletek: egyetlen vektor elemein végez az elemek között értelmezett műveletet. Pl. a vektor elemeinek összeadása, összeszorzása, stb.
  - Vektorok elemek átrendezése (shuffling)
- Skalár-vektor műveletek: egy vektor minden elemével és egy skalárral végzett műveletek. Pl. egy vektor minden elemének megszorzása egy skalárral, stb.
- Vektor load/store műveletek: a memória és a vektorregiszterek közötti adatmozgatásra szolgálnak.

A legnépszerűbb, manapság használatos vektor utasításkészlet kiegészítések tulajdonságait foglalja össze az alábbi táblázat:

Vektor kieg.	Utasításkészlet	#Vektorreg.	Vektor-hossz	Elemek típusa
MMX	x86	8	64 bit	8x8, 4x16, 2x32 bit integer
3DNow	x86	8	64 bit	2x32 bit float
SSE	x86/x64	8	128 bit	4x32 bit float
SSE2-4	x86/x64	8	128 bit	16x8, 8x16, 2x32 bit integer, 4x32, 2x64 bit float
AVX	x86/x64	16	256 bit	8x32, 4x64 bit float
AltiVec	Power	32	128 bit	16x8, 8x16, 4x32 bit integer, 4x32 bit float
NEON	ARM	32/16	64/128 bit	8x8, 4x16, 2x32 bit integer, 2x32 bit float

A vektor adattípusok és a vektorműveletek – talán meglepő módon – magas szinten, C/C++ nyelven is elérhetők, kihasználhatók, úgynevezett *intrinsic* utasítások segítségével. Kis szépséghiba, hogy ez a magas szintű támogatás platformfüggő, tehát minden SIMD utasításkészletre más és más. A megfelelő (utasításkészlet függő) fejállomány betöltése után rendelkezésre állnak a vektor adattípusok (pl. SSE esetén `__m128`, NEON esetén `float32x4_t`, stb.), és minden egyes támogatott vektorművelethez egy-egy C függvény.

Ezek használatra adunk egy példát az alábbiakban. Az eredeti függvény egy HD felbontású kép színtelítettségét növeli meg 50%-al. Az egyszerűség kedvéért a kép piros, zöld és kék komponenseit külön-külön `float` típusú tömb tárolja (melyek kezdetét a `srcR`, `srcG` és `srcB` mutatja), a keletkező, telítettebb kép pedig új helyre kerül (`dstR`, `dstG` és `dstB`). Arra is figyelmet kell fordítani, hogy az egyes intenzitásértékek a 0 és 255 közötti tartományban maradjanak. Az eredeti, SIMD utasításokat nem használó kód:

```
void saturate () {

    float r, g, b, p, val;

    for (int i=0; i<height*width; i++) {
        r = *srcR;
        g = *srcG;
        b = *srcB;

        p = sqrt (r*r + g*g + b*b);
```

```
            val = p + (r - p) * 1.5f;
            *dstR = val>255.0 ? 255.0 : val<0 ? 0 : val;
            val = p + (g - p) * change;
            *dstG = val>255.0 ? 255.0 : val<0 ? 0 : val;
            val = p + (b - p) * change;
            *dstB = val>255.0 ? 255.0 : val<0 ? 0 : val;

            srcR++; srcG++; srcB++;
            dstR++; dstG++; dstB++;
        }
}
```

Az SSE2 és NEON intrinsic-ek használatával átírt algoritmus a 19.5. ábrán látható. Mindkét utasításkészlet kiegészítés támogatja a 4 `float` típusból áll vektorokon végzett műveleteket. A `for` ciklus négyesével lépked a bemenetet szolgáltató és a kimenetet reprezentáló tömbökön. A következő 4 elem vektor változóba olvasását az SSE2 esetén a `_mm_load_p`, a NEON esetén a `vld1q_f32` végzi. Ezen különféle összeadás, kivonás és szorzás műveletet végzünk, valamint egy gyökvonást. Az SSE2-ben van valódi gyökvonás, míg a NEON a négyzetgyököt csak közelíteni tudja, annak is csak a reciprokát. A kapott eredményeket vektorműveletekkel a 0-255 tartományban kényszerítjük (`_mm_min_ps`, `_mm_max_ps`, valamint `vminq_f32`, `vmaxq_f32`), és a végén a memóriába írjuk (`_mm_store_ps`, valamint `vst1q_f32`).

A konkrét architektúrákon végzett mérési eredményeket a 19.2. táblázat foglalja össze. A 4 elemű vektorokkal dolgozó változatok (SSE2, ill. NEON) hozzávetőleg negyedannyi idő alatt végeztek a számítással. A Core-i7 esetén született egy AVX változat is, mely 8 elemű vektorral dolgozik, de ez csak minimális mértékben volt képes javítani az SSE2-vel kapott futási időn. Ennek oka, hogy az algoritmus memóriakorlátos lett, a cache nem tudja akkora sebességgel szolgáltatni az adatokat, amekkora sebességgel az AVX képes lenne feldolgozni azokat.

Intel Core i7-2600	SIMD nélkül	15.166 ms
	SSE2	3.829 ms
	AVX	3.698 ms
Intel Pentium-4	SIMD nélkül	139.758 ms
	SSE2	36.355 ms
RK3188	SIMD nélkül	155.012 ms
	NEON	44.026 ms

19.2. táblázat. Az SIMD utasításokkal elérhető gyorsulás

19.4. Tömbprocesszorok

19.4.1. Klasszikus tömbprocesszorok

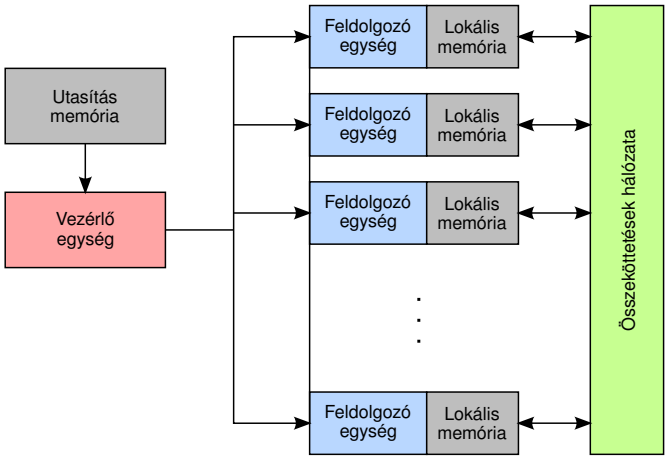
A klasszikus tömbprocesszorok (array processor) egy vezérlőegységből (control unit), és számos feldolgozóegységből (processing element) állnak. A feldolgozóegységek teljesen egyformák, és a műveleti egységek mellett lokális memóriával is rendelkeznek (melyek tartalma természetesen nem egyforma). A feldolgozóegységek adatot tudnak cserélni a velük összeköttetésben álló más feldolgozóegységekkel (19.6. ábra).

A programok végrehajtását a vezérlőegység koordinálja. A vezérlőegység minden ciklusban kiad egy utasítást, melyet üzenetszórással minden feldolgozóegység megkap. Ezt az utasítást minden feldolgozóegység végrehajtja a saját - lokális memóriájában tárolt - adatain. A tömbprocesszor tehát tényleg egy SIMD architektúra, hiszen egyazon utasítássorozat végrehajtása zajlik sok, a különböző feldolgozóegységekben tárolt adaton. Ha egy műveletet csak bizonyos adatokon kell végrehajtani, természetesen lehet alkalmazni a vektorprocesszoroknál látott maszkolásos technikát: minden feldolgozóegységben el kell helyezni egy feltételregisztert. Az egyes feldolgozóegységekben az utasítás tényleges végrehajtása vagy kihagyása a feltételregiszter állásának függvénye. A feltételregiszter tartalmát az előzőleg végrehajtott utasítások eredménye módosítja.

```
#include <xmmintrin.h>
void saturateSSE2 () {

    float p, val;
    __m128 r0, r1, r2, r3, r4;
    const __m128 r5 = 1.5f, 1.5f, 1.5f, 1.5f;
    const __m128 r6 = 0.0f, 0.0f, 0.0f, 0.0f;
    const __m128 r7 = 255f, 255f, 255f, 255f;

    for (int i=0; i<height*width; i+=4) {
        r1 = _mm_load_ps (srcR);
        r0 = r1;
        r0 = _mm_mul_ps (r0, r1);

        r2 = _mm_load_ps (srcG);
        r4 = r2;
        r4 = _mm_mul_ps (r4, r2);
        r0 = _mm_add_ps (r0, r4);

        r3 = _mm_load_ps (srcB);
        r4 = r3;
        r4 = _mm_mul_ps (r4, r3);
        r0 = _mm_add_ps (r0, r4);

        r0 = _mm_sqrt_ps (r0);

        r1 = _mm_sub_ps (r1, r0);
        r1 = _mm_mul_ps (r1, r5);
        r1 = _mm_add_ps (r1, r0);
        r1 = _mm_min_ps (r1, r7);
        r1 = _mm_max_ps (r1, r6);

        _mm_store_ps (dstR, r1);

        r2 = _mm_sub_ps (r2, r0);
        r2 = _mm_mul_ps (r2, r5);
        r2 = _mm_add_ps (r2, r0);
        r2 = _mm_min_ps (r2, r7);
        r2 = _mm_max_ps (r2, r6);

        _mm_store_ps (dstG, r2);

        r3 = _mm_sub_ps (r3, r0);
        r3 = _mm_mul_ps (r3, r5);
        r3 = _mm_add_ps (r3, r0);
        r3 = _mm_min_ps (r3, r7);
        r3 = _mm_max_ps (r3, r6);

        _mm_store_ps (dstB, r3);

        srcR+=4; srcG+=4; srcB+=4;
        dstR+=4; dstG+=4; dstB+=4;
    }
}
```

```
#include <arm_neon.h>
void saturateNEON () {

    float p, val;
    float32x4_t r0, r1, r2, r3, r4;
    const float32x4_t r5 = vdupq_n_f32 (1.5f);
    const float32x4_t r6 = vdupq_n_f32 (0.0f);
    const float32x4_t r7 = vdupq_n_f32 (255.0f);

    for (i=0; i<height*width; i+=4) {
        r1 = vld1q_f32 (srcR);
        r0 = vmulq_f32 (r1, r1);

        r2 = vld1q_f32 (srcG);
        r4 = vmulq_f32 (r2, r2);
        r0 = vaddq_f32 (r0, r4);

        r3 = vld1q_f32 (srcB);
        r4 = vmulq_f32 (r3, r3);
        r0 = vaddq_f32 (r0, r4);

        r0 = vrecpeq_f32 (vrsqrteq_f32 (r0));

        r1 = vsubq_f32 (r1, r0);
        r1 = vmulq_f32 (r1, r5);
        r1 = vaddq_f32 (r1, r0);
        r1 = vminq_f32 (r1, r7);
        r1 = vmaxq_f32 (r1, r6);

        vst1q_f32 (dstR, r1);

        r2 = vsubq_f32 (r2, r0);
        r2 = vmulq_f32 (r2, r5);
        r2 = vaddq_f32 (r2, r0);
        r2 = vminq_f32 (r2, r7);
        r2 = vmaxq_f32 (r2, r6);

        vst1q_f32 (dstG, r2);

        r3 = vsubq_f32 (r3, r0);
        r3 = vmulq_f32 (r3, r5);
        r3 = vaddq_f32 (r3, r0);
        r3 = vminq_f32 (r3, r7);
        r3 = vmaxq_f32 (r3, r6);

        vst1q_f32 (dstB, r3);

        srcR+=4; srcG+=4; srcB+=4;
        dstR+=4; dstG+=4; dstB+=4;
    }
}
```

19.5. ábra. Színtelítettség növelése SIMD utasításkészlettel



19.6. ábra. Tömbprocesszor blokksémája

Vegyük észre, hogy ha a feldolgozóegységek közvetlen összekapcsolását nem engedjük meg, egy egyszerű vektorprocesszort kapunk. Később látni fogjuk, hogy a hatékonyság szempontjából pont az lesz kulcsfontosságú, hogy a feldolgozóegységek megfelelő struktúra szerint össze vannak kapcsolva, és egymással adatot tudnak cserélni.

Más és más feladatok a feldolgozóegységek más és más összekötöttségét igénylik, ezért a struktúra rugalmasabbá tétele érdekében az összeköttetéseket célszerű átkonfigurálhatóvá tenni.

19.4.2. Szisztolikus és hullámfront tömbprocesszorok

A tömbprocesszorok klasszikus változatának a hatékony megvalósítását több tényező is gátolja, különösen ha a feldolgozóegységet száma nagyon nagy. Az egyik problémát az utasítások üzenetszóráson alapuló terjesztése okozza. Az elérhető működési sebességet korlátozza, hogy az utasításoknak egy ciklusidő alatt a tömbprocesszor egy pontjából, a vezérlőegységből, minden egyes feldolgozóegységbe el kell jutnia. A távoli feldolgozóegységekkel való adatcsere pedig nemcsak sokáig tart, de fogyasztási és termikus vonzata is van. A másik, hatékony megvalósítást gátló tényező, ha a feldolgozóegységek összekötöttsége bonyolult, illetve ha egymástól távoli feldolgozóegységek is össze vannak kötve egymással. Ebben az esetben is a hosszú jelutak jelentik a gondot (késleltetés, fogyasztás, melegedés).

A felmerült problémák megoldására több, speciális tömbprocesszor struktúrát is bevezettek.

Szisztolikus tömbprocesszorok

A szisztolikus tömbprocesszorok olyan speciális tömbprocesszorok, melyek a következő tulajdonságokkal bírnak:

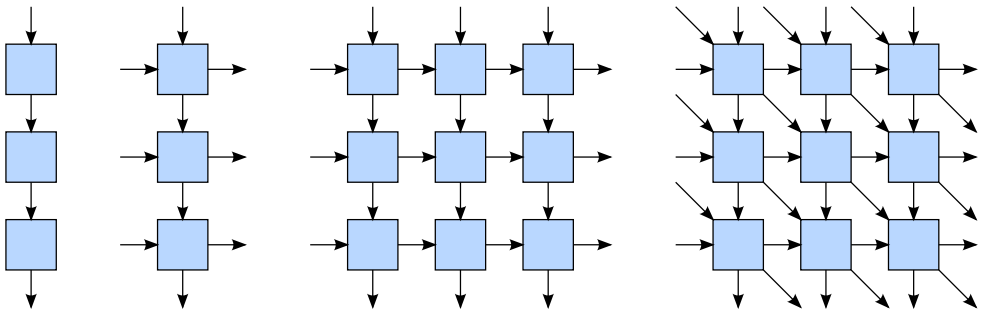
- a feldolgozóegységek kizárólag a legközelebbi szomszédaikkal vannak összekötve,
- a feldolgozóegységek közötti kommunikáció teljesen szinkronizált (közös órajel hajtja),
- a tömb csak a szélein kapcsolódik a külvilághoz, azaz csak a széleken lehet inputot beadni, ill. a széleken jelenik meg az output,
- nincs központi vezérlőegység: minden feldolgozóegység minden órajelciklusban ugyanazt a műveletet végzi.

A szisztolikus tömbprocesszorok feldolgozóegységei minden egyes órajelciklusban veszik a szomszédaiktól érkező bemeneteket, elvégeznek rajta valamilyen műveletet, és az eredményt továbbítják más szomszédaik felé. Ennek a kialakításnak a következő előnyös tulajdonságai vannak:

- Mivel minden feldolgozóegység csak a legközelebbi szomszédaival van összekötve, rendkívül rövidek lesznek a jelutak. Emiatt:
  - kedvezőbb a fogyasztás,
  - és a kisebb jelátviteli késleltetések miatt magasabb órajel frekvencia érhető el.

- Az integrált áramkörök gyártástechnológiájának egyik sajátossága, hogy a kis méretű ismétlődő mintázatokat tartalmazó szilíciumszeletek előállítása nagyon hatékony (hamar bevezethető a gyártás és kicsi a selejt-arány).
- Ez a fajta struktúra könnyen bővíthető, hiszen új feldolgozóegységek hozzáadásakor azokat csak a tömb széléhez kell "csatlakoztatni", bővítéskor a tömb belseje nem változik.

Néhány tipikus (gyakorlatban hasznos algoritmusokhoz szükséges) összekötöttségi struktúrát mutat be a 19.7. ábra.



19.7. ábra. Tipikus szisztolikus tömbprocesszor struktúrák

Hullámfront tömbprocesszorok

A szisztolikus tömbprocesszorok egyetlen hátránya a globális órajelhálózat. Minél nagyobb a feldolgozóegységek száma, annál hosszabb vezetéken kell az órajelet áthajtani, hogy az minden feldogozóegységhez odaérjen. Nagy számú feldolgozóegység vagy nagy órajel frekvencia esetén az sem elhanyagolható tényező, hogy egyes feldolgozóegységekhez hamarabb ér el az órajel, mint másokhoz, ami kavarodáshoz, helytelen működéshez vezethet (clock skew).

A globális órajelhálózat elhagyásával megkapjuk a teljesen aszinkron működésű hullámfront tömbprocesszorokat. A hullámfront tömbprocesszorok működése tulajdonképpen adatfolyam alapú: egy feldolgozóegység akkor végzi el a hozzá rendelt műveletet, ha minden bemenete megérkezett, és a kimenetéről már elvették az előző művelet eredményét. A szomszédos feldolgozóegységek közötti adatcsere tehát egy összetettebb protokollt igényel (átadás/átvétel visszaigazolással), ami ugyan némi hardveres komplexitás-növekedést jelent, cserébe a globális órajelhálózat hiányának köszönhetően még olcsóbb az előállítás, még kisebb a fogyasztás és még nagyobb órajel frekvencia érhető el.

Mivel a feldolgozóegységet egyformák, sebességük is egyforma, a kialakítás természetéből adódóan az adatok hullámszerűen terjednek végig a tömbön, innen kapta a nevét is.

19.4.3. Szisztolikus tömbprocesszorok tervezése

Az erre alkalmas algoritmusok szisztolikus tömbprocesszoron való megvalósítására egy többé-kevésbé mechanikus eljárást dolgoztak ki a 80-as években. Ebben a fejezetben egy példán keresztül mutatjuk be ezt az eljárást (Megjegyezzük, hogy az itt és a továbbiakban leírtak nem csak a szisztolikus tömbprocesszorokra érvényesek, hanem a hullámfront tömbprocesszorokra is.)

A példa algoritmus a vektor-mátrix szorzás, melynek során egy NxM-es mátrixot szorzunk meg egy N hosszú vektorral, jobbról. Az eredmény egy M hosszú vektor lesz:

```
for (int i=0; i<M; i++) {
    C[i] = 0;
    for (int j=0; j<N; j++)
        C[i] = C[i] + A[i][j] * B[j];
}
```

Egy hagyományos, szekvenciális processzoron ez az algoritmus $N \cdot M$ lépést vesz igénybe.

A teljes függőségi gráf

A függőségi gráf az algoritmus műveleteinek egymástól való függését írja le. A szisztolikus tömbprocesszor párhuzamos feldolgozást végez (a feldolgozóegység egymással párhuzamosan működnek), ezért fontos felderíteni, hogy az algoritmus mely lépései függetlenek egymástól, vagyis mely műveleteket lehet egymással párhuzamosan elvégezni.

Érezzük, hogy a példaként felhozott mátrix-vektor szorzás bizonyos műveletei párhuzamosíthatók (pl. az eredmény vektor minden elemét egymással párhuzamosan számolhatjuk), de mivel szisztematikus (gép által végezhető) eljárást keresünk a párhuzamosíthatóság felfedésére, erre a következtetésre a programkód alapján kell jutnunk. Sajnos a példában van egy nyelvi megoldás, ami nehezíti ezt a munkát: a `C[i] = C[i] + ...` sorról van szó. A C[i] változónak pusztán memóriatakarékossági okokból adunk többször értéket. Ez bizonyos célokra teljesen racionális, de az ilyen *többszörös hozzárendelés* megnehezíti a párhuzamosíthatóság felderítését, elrejti az algoritmus természetét. Ezért a programot átfogalmazzuk úgy, hogy csak *egyszeres hozzárendelést* engedünk meg:

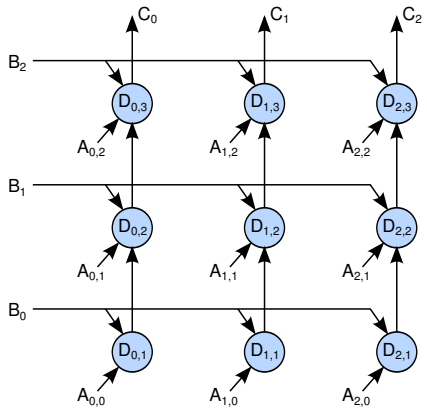
```
for ( int i=0; i<M; i++) {
    D[i][0] = 0;
     for ( int j=0; j<N; j++)
         D[i][j+1] = D[i][j] + A[i][j] * B[j];
    C[i] = D[i][N];
}
```

(Megjegyezzük, hogy a párhuzamosságot automatikusan támogató programozási nyelvek mind egyszeres hozzárendelésűek, azaz egy változónak csak egyetlenegyszer lehet értéket adni. Ilyenek például a funkcionális nyelvek: F#, ML, stb.)

Ez az átfogalmazott programkód tulajdonképpen az alábbi rekurzív eljárást valósítja meg:

$$D_{i,j+1} = D_{i,j} + A_{i,j} \cdot B_j, \quad \text{ahol } D_{i,0} = 0, \text{ és } C_i = D_{i,N}.$$

A teljes függőségi gráf a 19.8. ábrán látható. A gráf csomópontjai a rekurzívan számolt mennyiségnek, most a $D_{i,j}$-nek felelnek meg. Az irányított élek jelzik a függőségi viszonyokat. Ahogy az ábrán is látszik, a példa algoritmus 2 dimenziós ($N \times M$-es) indextérben mozog.



19.8. ábra. A teljes függőségi gráf

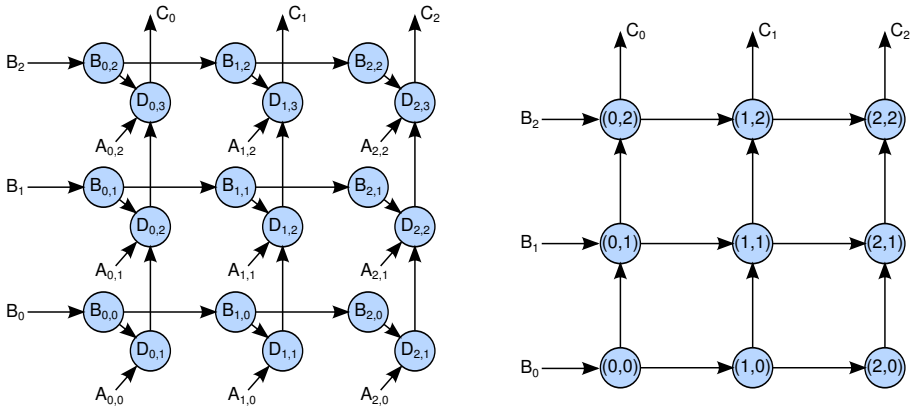
A lokalizált függőségi gráf

A teljes függőségi gráf tartalmazhat olyan megoldásokat, melyek a tömbprocesszoros megvalósítást megnehezítik. Ebben a példában a "B_j" bemenet okoz gondot, mert ettől a bemenettől a függőségi gráfban egyidejűleg több csomópont is függ. Az algoritmusok tömbprocesszorra való leképzéséhez ú.n.*lokalizált függőségi gráf*ra van szükség, melyben függőségek csak a szomszédos csomópontok között vannak. Ehhez a már eddig is rekurzív algoritmusunkat lokálisan rekurzívvá kell alakítani:

```
for ( int i=0; i<M; i++) {
    D[i][0] = 0;
     for ( int j=0; j<N; j++) {
        B[i+1][j] = B[i][j];
        D[i][j+1] = D[i][j] + A[i][j] * B[i][j];
    }
    C[i] = D[i][N];
}
```

(ahol kezdetben B[0][j] = B[j]).

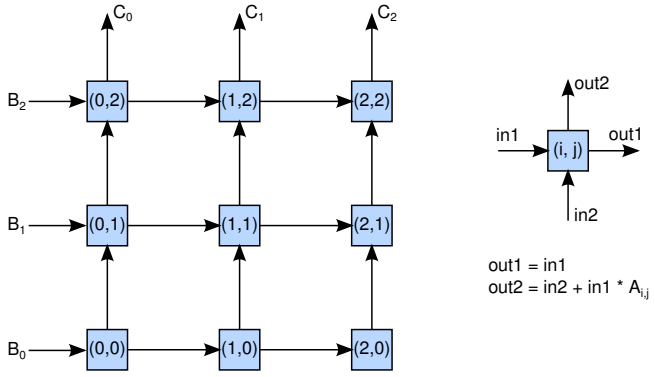
A lokalizált algoritmussal a 19.9. ábrán látható lokális függőségi gráfot kapjuk. Ebben a függőségi gráfban tényleg csak az egymással szomszédos csomópontok vannak összekötve. Ezt a lokális függőségi gráfot szokás egyszerűsítve ábrázolni úgy, hogy amit csak lehet, összevonunk. Az egyszerűsített ábrán most pl. a (1,2)-mal jelzett csomópontot a $B_{1,2}$, a $D_{1,3}$, és az $A_{1,2}$ csomópontok összevonásával kaptuk meg.



19.9. ábra. A lokális függőségi gráf és az egyszerűsített formája

Közvetlen leképzés szisztolikus tömbprocesszorra

A lokális függőségi gráf akár közvetlenül is leképezhető szisztolikus tömbprocesszorra úgy, hogy annak minden csomópontját egy-egy külön feldolgozóegységnek feleltetünk meg. Mivel a lokális függőségi gráfban minden csomópont csak a szomszédjával áll függőségi kapcsolatban, a kapott feldolgozóegység hálózat eleget tesz a szisztolikus tömbprocesszorok szükséges kritériumainak. Az így kapott struktúra és az egyes feldolgozóegységek feladata a 19.10. ábrán látható (a mátrix $A_{i,j}$ együtthatói a csomópontokban vannak eltárolva).



19.10. ábra. Közvetlen leképzés szisztolikus tömbprocesszorra és a feldolgozóegységek feladata

Egy lehetséges, a feladatot megvalósító struktúra tehát már megvan, már csak az hiányzik, hogy milyen időzítésekkel (milyen ütemben) kell a tömbprocesszor szélein a bemeneteket megjelentetni, hogy a kívánt kimenetet (a

mátrix-vektor szorzás eredményét) kapjuk. Ezt az időzítési feladatot *ütemezés*nek hívjuk.

A közvetlen leképzés esetén az ütemezést adatáramlásos elven határozzuk meg. Minden ciklusban azok a feldolgozóegységek végeznek hasznos feladatot, melyek működéséhez minden operandus rendelkezésre áll (természetesen minden ciklusban minden feldolgozóegység működik, hiszen *valami* mindig van a bemenetén, de csak ezek adnak értelmes eredményt). A bemeneteket úgy kell ütemezni, hogy minden ütemben a hasznos munkát végző feldolgozóegységeket ellássuk a feladatukhoz szükséges inputtal. A példában az első ütemben csak a (0,0) feldolgozóegység tud hasznos munkát végezni, tehát az első ütemben csak a B_0-át kell az (0,0) bemenetére adni. A második ütemben a (0,1) és az (1,0) is megkapja (0,0) kimenetét, tehát a (0,1)-nek a második ütemben van szüksége B_1-re, és így tovább. Azt, hogy az egyes bemeneteket melyik ütem elején kell megjelentetni, mely feldolgozóegységek aktívak és a kimenetek mely ütem végén jelennek meg, a következő táblázat foglalja össze:

Ütem	Bemenet	Aktivitás	Kimenet
1	B[0]	(0,0)	-
2	B[1]	(0,1) (1,0)	-
3	B[2]	(0,2) (1,1) (2,0)	C[0]
4	-	(1,2) (2,1)	C[1]
5	-	(2,2)	C[2]

Némi meggondolással belátható, hogy ez a szisztolikus tömb $N + M - 1$ lépésben végez azzal a mátrix-vektor szorzással, ami egy klasszikus, szekvenciális számítógépnek $N \cdot M$ lépésig tartott.

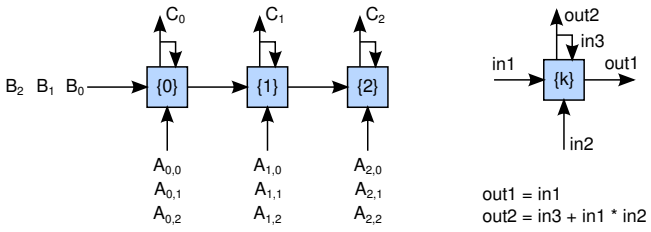
Vegyük észre, hogy a feldolgozóegységek aktivitása a végrehajtás során úgy alakul, mint egy hullámfront: a (0,0)-ből indulva, átlósan, észak-keleti irányba mutató normálvektorral a (2,2) felé.

Projekció

Látható, hogy a közvetlen leképzéssel kapott szisztolikus tömbprocesszor feldolgozóegységeinek nagyon rossz a kihasználtsága. Az előbbi táblázat alapján minden feldolgozóegység csupán egyetlen ütemben dolgozott az 5 ütemig tartó művelet során. A kihasználtságot lehet javítani, ha kevesebb feldolgozóegységet veszünk, melyek mindegyike a lokalizált függőségi gráf több csomópontjának szerepét is el tudja játszani. Pl. a példában a (0,0), a (0,1) és a (0,2) a függőségi viszonyok miatt nem hajtható végre soha egyszerre, ezért ezeket ugyanarra a feldolgozóegységre képezhetjük, mely először a (0,0), majd a (0,1), utána pedig a (0,2) szerinti műveleteket fogja elvégezni.

Projekció a vízszintes tengelyre

A vízszintes tengelyre való projekció azt jelenti, hogy a két dimenziós függőségi gráfot úgy tesszük egy dimenzióssá, hogy azt levetítjük a vízszintes tengelyre. Azaz, a (0,0), (0,1), (0,2) műveletet képezzük ugyanazon feldolgozóegységre, az (1,0), (1,1), (1,2)-t egy másikra, és a (2,0), (2,1), (2,2)-t pedig egy harmadikra, a 19.11. ábra szerint.



19.11. ábra. Leképzés vízszintes projekcióval

Az eredeti 9 helyett tehát 3 feldolgozóegység fogja végrehajtani a feladatot, az alábbi táblázat szerint:

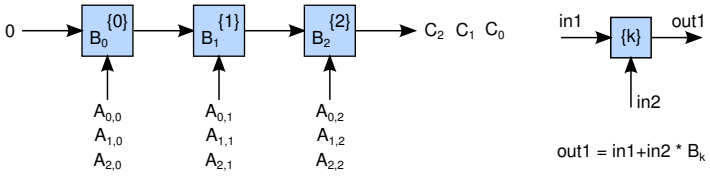
Ütem	Bemenet	Aktivitás	Kimenet
1	B[0] A[0][0]	{0}	-
2	B[1] A[0][1] A[1][0]	{0} {1}	-
3	B[2] A[0][2] A[1][1] A[2][0]	{0} {1} {2}	C[0]
4	A[1][2] A[2][1]	{1} {2}	C[1]
5	A[2][2]	{2}	C[2]

A táblázatból látható, hogy minden egyes feldolgozóegység 3 ütemben is dolgozott, tehát tényleg javult a kihasználtság. Például a {0} feldolgozóegység az 1. ütemben a (0,0), a 2. ütemben a (0,1), a harmadik ütemben pedig a (0,2) szerepét tölti be. Mivel a feldolgozóegységet egyformák, ez semmi extra komplikációt nem jelent.

Lényegében minden feldolgozóegység az eredményvektor egy elemét állítja elő. A feldolgozóegységek out2 kimenetének visszacsatolása egyfajta memóriát képvisel, ami az eredmény C[1] = A[1][0] * B[0] + A[1][1] * B[1] + A[1][2] * B[2] összeg tagjait akkumulálja (a második feldolgozóegység esetén).

Projekció a függőleges tengelyre

A vízszintes helyett elvégezhetjük a projekciót a függőleges tengelyre is, azaz a (0,0), (1,0), (2,0) műveletet képezzük ugyanazon feldolgozóegységre, az (0,1), (1,1), (2,1)-t egy másikra, és a (0,2), (1,2), (2,2)-t pedig egy harmadikra, a 19.12. ábrának megfelelően.



19.12. ábra. Leképzés függőleges projekcióval

Ismét 3 feldolgozóegységet kaptunk. Az alábbi táblázat megadja a végrehajtás menetét és az ütemezést:

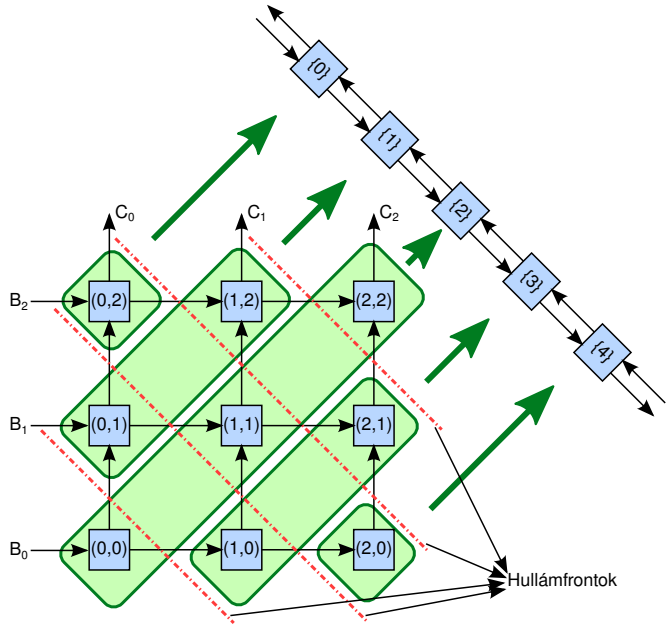
Ütem	Bemenet	Aktivitás	Kimenet
1	A[0][0]	{0}	-
2	A[0][1] A[1][0]	{0} {1}	-
3	A[0][2] A[1][1] A[2][0]	{0} {1} {2}	C[0]
4	A[1][2] A[2][1]	{1} {2}	C[1]
5	A[2][2]	{2}	C[2]

A táblázatból látható, hogy minden egyes feldolgozóegység 3 ütemben is dolgozott, tehát tényleg javult a kihasználtság. Például a {0} feldolgozóegység az 1. ütemben a (0,0), a 2. ütemben a (1,0), a harmadik ütemben pedig a (2,0) szerepét tölti be.

Az így kapott tömbprocesszorban az egyes feldolgozóegységek az eredmény C[1] = A[1][0] * B[0] + A[1][1] * B[1] + A[1][2] * B[2] összeg tagjait számolják, tehát az első feldolgozóegység mindig az eredményvektor minden elemére az összeg első tagot számolja ki, a második ehhez hozzáadja a másodikat, a harmadik pedig a harmadik tagot is hozzávéve előállítja az eredményt.

Projekció diagonál irányban

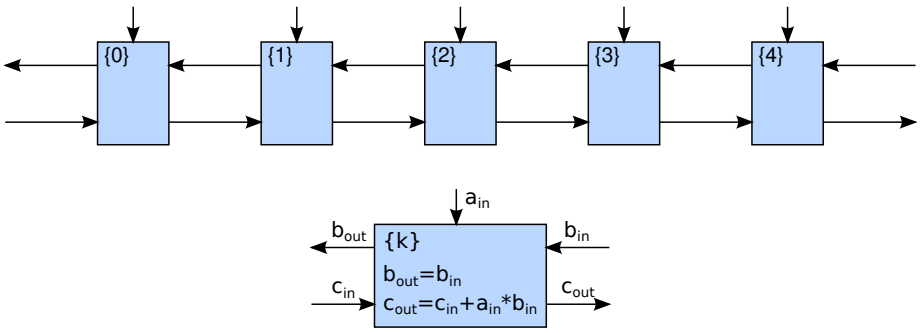
Nemcsak horizontális és vertikális irányokban, hanem diagonálisan, a hullámfrontok mentén is végezhetünk projekciót. Talán ez a legtermészetesebb megoldás, hiszen a hullámfrontok mentén egymástól független, tehát párhuzamosan végrehajtható műveletek vannak, így az azonos hullámfronthoz tartozó műveletekhez célszerű külön-külön feldolgozóegységeket rendelni (19.13. ábra).



19.13. ábra. Projekció a hullámfrontok mentén

A háromféle projekció közül ez lesz a legszerencsésebb megoldás, mivel a feldolgozóegységekben sem visszacsatolással kialakított memóriára, sem a B vektor elemeinek az eltárolására nincs szükség. Az esetek túlnyomó többségében a hullámfront mentén végzett projekciót érdemes alkalmazni.

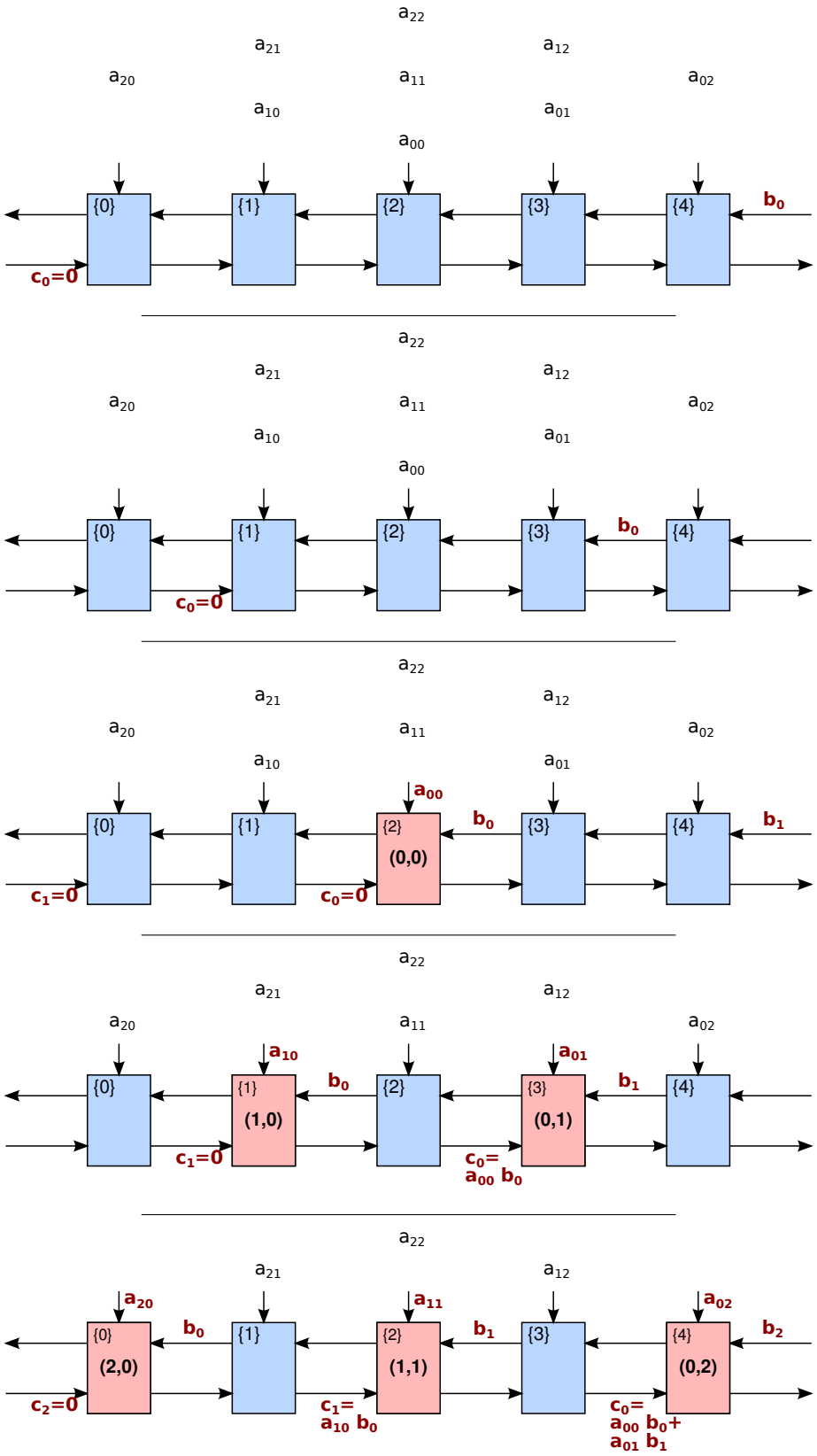
A hullámfront alapú projekció eredményeként kapott tömbprocesszort a 19.14. ábra, a megfelelő ütemezést a 19.15. ábra mutatja be.



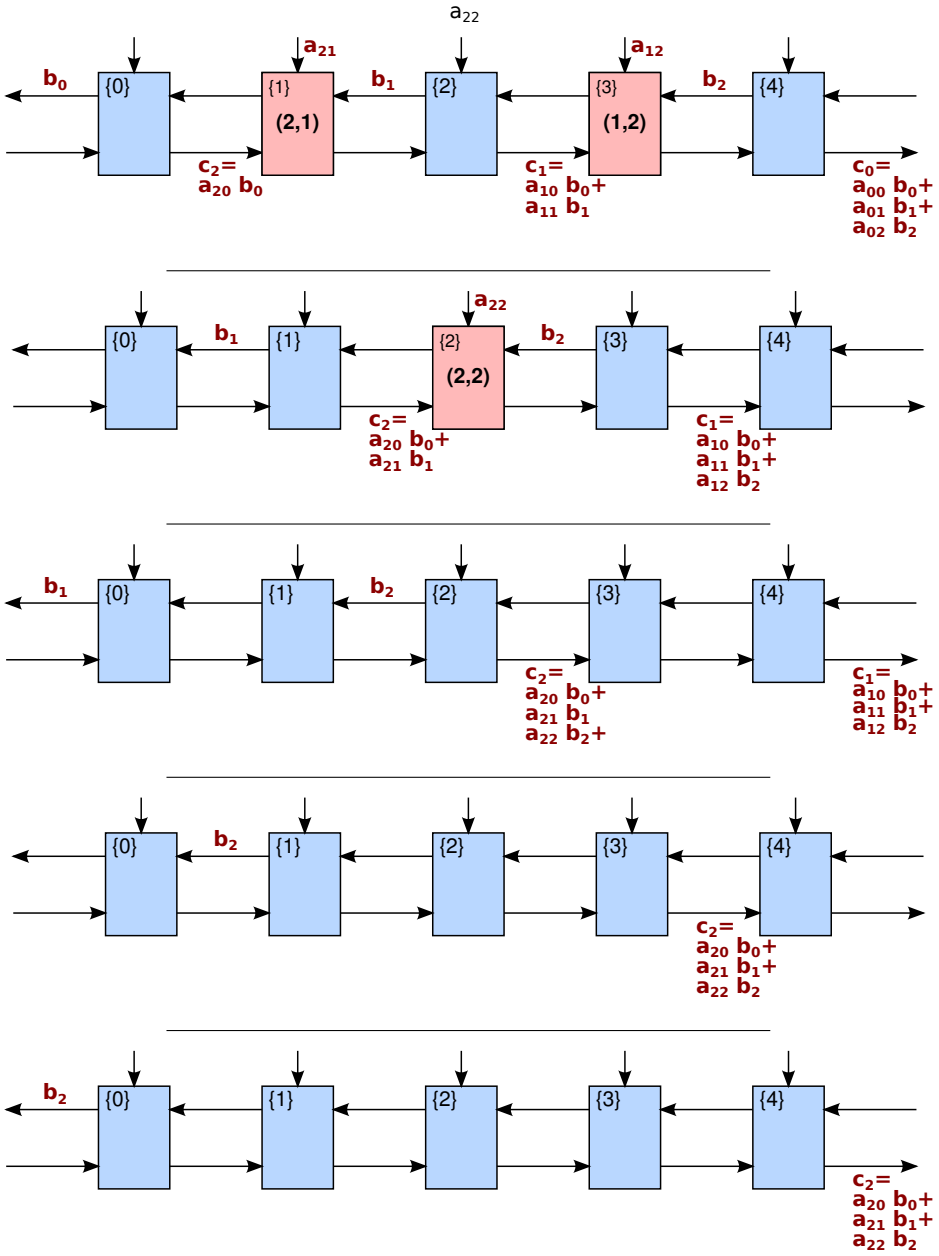
19.14. ábra. Tömbprocesszor hullámfrontok mentén végzett projekcióval

Pipeline viselkedés a projekció elhagyásával

Ha a projekciót nem végezzük el, a közvetlen leképzéssel kapott szisztolikus tömb pipeline adatfeldolgozásra is képes lesz. Láttuk, hogy a tényleges műveleteket végző feldolgozóegységek aktivitása hullámfrontként vonul (0,0)-ból (2,2)-be. Vegyük észre, hogy ha a mátrixot egy másik vektorral is meg akarnánk szorozni, nem kell megvárni az előző művelet végét: az első hullámfront mögött rögtön egy újabbat indíthatunk, az után egy újabbat, és így tovább. A következő táblázatban különböző színű hátterekkel azonosítjuk a megszorzandó vektorokat, az azokon dolgozó feldolgozóegységeket és a hozzájuk tartozó eredményeket:



19.15. ábra. Ütemezés hullámfrontok mentén végzett projekció esetén , 1/2.rész



19.15. ábra. Ütemezés hullámfrontok mentén végzett projekció esetén, 2/2. rész

Ütem	Bemenet	Aktivitás	Kimenet
1	B[0]	(0,0)	-
2	B[0] B[1]	(0,0) (0,1) (1,0)	-
3	B[0] B[1] B[2]	(0,0) (0,1) (1,0) (0,2) (1,1) (2,0)	C[0]
4	B[0] B[1] B[2]	(0,0) (0,1) (1,0) (0,2) (1,1) (2,0) (1,2) (2,1)	C[0] C[1]
5	B[0] B[1] B[2]	(0,0) (0,1) (1,0) (0,2) (1,1) (2,0) (1,2) (2,1) (2,2)	C[0] C[1] C[2]
6	B[0] B[1] B[2]	(0,0) (0,1) (1,0) (0,2) (1,1) (2,0) (1,2) (2,1) (2,2)	C[0] C[1] C[2]
7	B[0] B[1] B[2]	(0,0) (0,1) (1,0) (0,2) (1,1) (2,0) (1,2) (2,1) (2,2)	C[0] C[1] C[2]

Ha tehát ugyanazt a mátrixot kell más-más vektorokkal szorozgatni, akkor a pipeline viselkedésnek köszönhe-

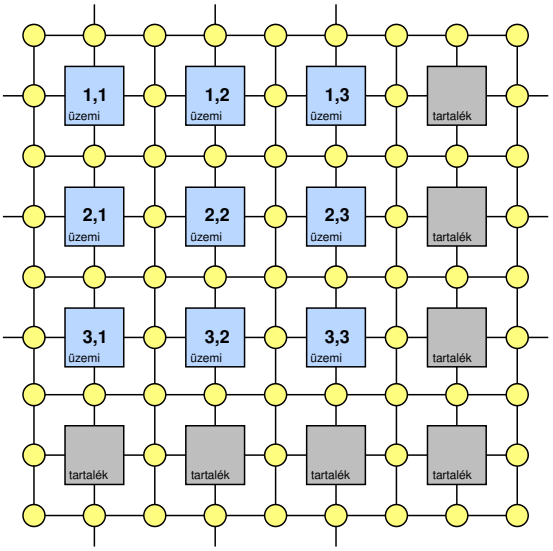
tően aszimptotikusan 1 mátrix-vektor szorzás egyetlenegy ütemet vesz igénybe!

19.4.4. Tömbprocesszorok hibatűrése

Van olyan működési környezet, ahol követelmény a hosszú ideig tartó hibamentes működés (pl. műholdak, vagy egyéb nehezen megközelíthető helyek esetén). A szisztolikus tömbprocesszorok naiv implementációja (feldolgozóegységek egymás mellé ültetése és a szomszédok összehuzalozása) hibatűrés szempontjából nem szerencsés, hiszen egyetlen feldolgozóegység meghibásodása esetén a tömbprocesszor outputja hibás lesz, így a teljes tömb használhatatlanná válik.

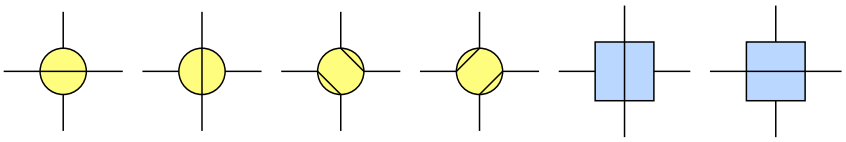
A hibatűrés javítása érdekében a tömbprocesszorban üzemi állapotban nem működő, "tartalék" feldolgozóegységeket is célszerű elhelyezni, melyeket az összeköttetések átkonfigurálhatóvá tételével a meghibásodott feldolgozóegységek helyett szükség esetén üzembe lehet állítani. A kulcsfontosságú kérdés az, hogy hogyan lehet a feldolgozóegységek közötti összeköttetéseket úgy kialakítani, hogy bonyolultak se legyenek (hiszek akkor azok is meghibásodhatnának), de képesek legyenek esetleg több hiba esetén is a hibás feldolgozóegységeket kikerülve a tartalék feldolgozóegységeket bekötni a szabályos struktúrába.

Egy lehetséges megoldás, ha a feldolgozóegységek közé egyszerű kapcsoló csomópontokat helyezünk el (a 19.16. ábrán sárgával jelölve).



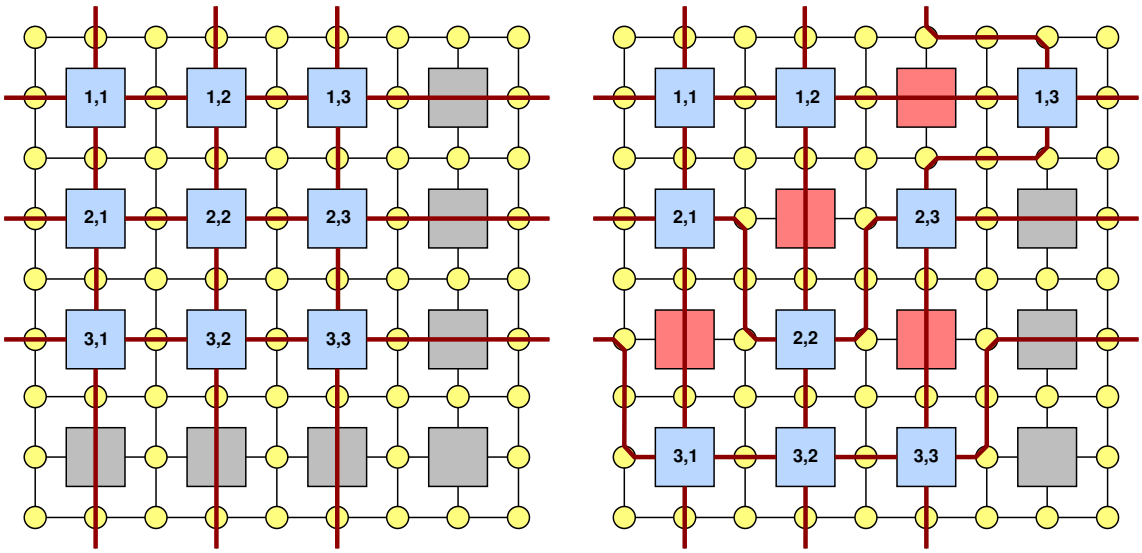
19.16. ábra. Szisztolikus tömbprocesszor tartalék egységekkel és átkonfigurálható összeköttetésekkel

A kapcsoló csomópontok 4 féle állást tudnak felvenni, a 4 irányból jövő vezetékeket 4 féleképpen tudják összekötni (19.17. ábra). Továbbá, a feldolgozóegységeket ki kell egészíteni a rövidre zárás képességével, hogy hiba esetén a rövidzárat bekapcsolva a jelet a rossz feldolgozóegységen keresztül az egyik irányba változatlan formában továbbítani lehessen (19.17. ábra).



19.17. ábra. A kapcsoló csomópontok és a feldolgozóegységek rövidzárainak állásai

Talán nem is látszik elsőre, hogy ez a megoldás milyen rugalmas. A 19.18. ábrán látható példában a tömb 9 üzemi és 6 tartalék feldolgozóegységből áll, és akár 3 meghibásodást is képes átvészelni (a hibás feldolgozóegységeket piros szín jelöli).



19.18. ábra. Szisztolikus tömbprocesszor hibamentes és 4 hibás, átkonfigurált állapotban

20. fejezet

Multiprocesszoros rendszerek

20.1. Explicit párhuzamosság

A párhuzamosság formái:

1. Bit szintű párhuzamosság: A processzorokban, az aritmetikai műveletek során használatos szavak, adategységek hossza fokozatosan emelkedett, 4 bitről indulva a nyolcvanas évek közepéig elérte a 32 bitet, majd a 90-es évek elején a 64 bitet. Ezen a ponton a trend egy kicsit lassult, a 64 bites szóhossz már a legtöbb célra megfelelő lebegőpontos számábrázolási pontosságot tesz lehetővé, és a 64 bit a jelenlegi ütemben növekvő kapacitású memóriák címzésére is még sokáig elegendő lesz.

2. Implicit párhuzamosság: Az implicit párhuzamosság azt jelenti, hogy a programban rejlő párhuzamosítási lehetőségek felderítése és kiaknázása a programozó tudta és kifejezett támogatása nélkül történik. Fontos, hogy a programozó szekvenciális programot ír, nem is kell foglalkoznia azzal, hogy a program bizonyos részei egymással párhuzamosan fognak futni, ezért nem kell a programot a hardverhez igazítani: ha bővül a feldolgozó egységek száma, azt a rendszer automatikusan ki tudja majd használni. Az implicit párhuzamosság leggyakoribb megjelenési formái:

   (a) Utasítás pipeline alapú párhuzamosság: Az utasítás pipeline az utasítások végrehajtásának átlapolásán alapul. Az utasítás pipeline-ban az utasítások végrehajtása lényegében párhuzamosan történik, még ha azok a végrehajtás különböző fázisaiban vannak is. A pipeline fázisainak száma meghatározza a párhuzamosság fokát. A párhuzamosság foka növelhető ugyan a fázisok számának növelésével, de ennek gátat szabnak az utasítások közötti egymásra hatások. Minél több utasítás végrehajtása zajlik egyszerre, annál nagyobb az esélye, hogy ezek az utasítások valamilyen függőségi viszonyban vannak egymással (pl. az egyik eredménye a másik bemenete), így mégsem lehet tetszőlegesen átlapolni a végrehajtásukat.

   (b) Műveleti egységek többszörözésén alapuló párhuzamosság: A szuperskalár és a VLIW/EPIC architektúrák alapötlete, hogy többszörözzük meg a processzorban a műveleti egységeket, derítsük fel a független utasításokat, és a több műveleti egységgel hajtsuk végre őket párhuzamosan. A szuperskalár processzorokban a hardver, a VLIW processzorokban a szoftver (a fordítóprogram) feladata a párhuzamosan végrehajtható utasítások összeválogatása. A párhuzamosan végrehajtható utasítások számának ismét gátat szabnak az utasítások közötti adatfüggőségek: ha egy utasítás végrehajtásához szükség van egy korábbi eredményére, akkor nyilván nem lehet őket párhuzamosan végrehajtani, akárhány műveleti egység is van a processzorban. A gyakorlatban, valós programok futtatása alapján az a tapasztalat, hogy a 2-utas rendszerek (melyek egyszerre két műveletet képesek elvégezni párhuzamosan) jelentősen gyorsabbak, mint az 1-utasok, a 4-utasok még szignifikánsan gyorsabbak a 2-utasoknál, de a 6-utas rendszerek teljesítménybeli hozadéka alig mérhető.

3. Explicit párhuzamosság: Mint láttuk, az implicit párhuzamosságnak gátat szabnak az eredetileg szekvenciálisnak szánt program utasításai közötti egymásrahatások (leginkább az adatfüggőségek). Ezen úgy lehet segíteni, hogy a programozó explicit módon megadja a program párhuzamosan végrehajtható részeit, vagyis a programot eleve a párhuzamos végrehajtás szándékával írja. Amint azt az információfeldolgozási

modelleknél láttuk, ennek módja lehet többek között a vezérlő token explicit többszörözése (FORK), majd a tokenek későbbi összefogása (JOIN). A több vezérlő token több, egymástól független logikai vagy fizikai processzor kezelésében lehet. A program több, a logikai vagy fizikai processzorokon párhuzamosan futó ága persze cserélhet adatot egymással, használhat közös memóriát, akár még közös cache-t is, a párhuzamosan futó ágak időnként megvárhatják egymást, stb. Ebben a fejezetben az explicit párhuzamosság fajtáit, lehetőségeit, alapproblémáit járjuk körbe.

Az explicit párhuzamosságot támogató legelterjedtebb hardver megvalósítások a *többszálú processzorok* (20.2. fejezet) és a *multiprocesszoros rendszerek* (20.3. fejezet). A többszálú processzorok olyan (tipikusan szuperskalár) processzorok, melyek több utasításszámlálóval rendelkeznek, és hol az egyik, hol a másik utasításszámláló által kijelölt utasításokat hajtják végre a processzor erőforrásain. A multiprocesszoros rendszerek ezzel szemben több teljes értékű processzorból állnak, melyek mindegyike a saját utasításszámlálóján keresztül más-más utasítássorozatot hajt végre.

20.2. Több végrehajtási szálat támogató egyprocesszoros rendszerek

Ebben a fejezetben látni fogjuk, hogy több végrehajtási szálat érdemes lehet hardveresen támogatni egyprocesszoros rendszerekben is. Ebben az esetben ugyanis a processzor a több utasításfolyamból könnyebben össze tud válogatni annyi egymástól független, végrehajtható állapotban lévő utasítást, hogy a műveleti egységek a lehető legjobban ki legyenek használva. Ha az egyik szál megakadna is pl. egy cache hiba vagy egy adatfüggőség miatt, a processzor a másik szál felé tud fordulni, és annak utasításait kezdi el végrehajtani. Ha a fennakadás oka megszűnik, a végrehajtás visszatérhet az első szálhoz.

20.2.1. Többszálú végrehajtás hardver támogatás nélkül

A hardveres többszálú processzorok megismerése előtt célszerű tisztázni, hogy mi is az a többszálú végrehajtás.

Először nyúljunk vissza egy régebbi fogalomhoz: az időosztásos multitaszk operációs rendszerekhez. A professzionális célra szánt operációs rendszerek már kb. 50 évvel ezelőtt is támogatták több ”program” (taszk) egyprocesszoros rendszeren való egyidejű végrehajtását. Ha csak egy processzor, és emiatt egy utasításszámláló áll rendelkezésre, akkor a több taszk párhuzamos futásának látszatát időosztással lehet kelteni. Ennek érdekében a számítógépben egy időzítőt helyeznek el, ami meghatározott időközönként megszakítást generál. Ekkor az operációs rendszer időzítőt kiszolgáló szubrutinja hívódik meg, ami a következőket teszi:

- A futó taszk végrehajtását megszakítja: állapotát a memóriában menti (többek között a regiszterek értékét, az utasításszámlálót, a verem mutatót).
- Kiválaszt egy másik taszkot, aminek a futása korábban ugyanígy félbe lett szakítva.
- Ennek a kiválasztott taszknak előveszi a memóriába letárolt állapotát: visszaállítja a regiszterek és a verem mutató megszakításkori értékét.
- Visszaállítja a kiválasztott taszk elmentett utasításszámlálóját: ettől kezdve tehát a processzor innen fogja folytatni az utasítások végrehajtását.

Ez a taszkváltás elnagyolt vázlata. Egy modern processzorban a taszkváltáskor ennél azért több minden történik, pl. a regiszterek, a veremmutató és az utasításszámláló értékének cseréjén kívül mást is kell csinálni:

- A modern operációs rendszerek minden taszkra külön laptáblát tartanak karban, hogy azok azt higgyék, hogy a teljes virtuális címtartomány csak hozzájuk tartozik. Ennek érdekében taszkváltáskor át kell állítani a laptáblára mutató hardver regiszter értékét, hogy az ne a régi, hanem az új taszk laptáblájára mutasson. Ekkor azonban a TLB teljes tartalmát is érvényteleníteni kell, tehát a taszkváltás után jó darabig minden memóriahivatkozás TLB hibával fog járni, míg a TLB meg nem telik az új taszk gyakran használt laptábla bejegyzéseivel.
- Sok processzor taszkváltáskor üríti az utasítás cache-t, tehát a taszk végrehajtása jó pár cache-hibával fog indulni.

Összességében elmondható, hogy a taszkok közötti váltás nagyon lassú: az Intel Pentium 4 esetében például akár 2-3 ezer órajelciklust is igénybe vehet.

Nem ritka, hogy a taszkok (programok) végrehajtása is több, párhuzamos végrehajtási szálra bomlik (a programozó az explicit párhuzamosság jegyében megmondta, hogy a program mely részei hajthatók végre párhuzamosan). Az egy taszkhoz tartozó végrehajtási szálak közösen használják a taszk rendelkezésére álló memóriát (pl. látják egymás lokális változóit is, stb.), tehát az egy taszkhoz tartozó szálak közötti kapcsoláskor nem indokolt sem a laptáblamutató állítása, sem a TLB kiürítése, sem az utasítás cache kiürítése. A taszkokkal ellentétben tehát az egy taszkhoz tartozó szálak közötti váltás nagyon gyors: csak a regisztereket és az utasításszámlálót kell átállítani.

A továbbiakhoz a következő tanulságokat kell levonni:

- Több program (taszk) párhuzamos futásának látszatát egy processzor esetén időosztással lehet kelteni.
- A taszkok közötti váltás nagyon lassú
- Egy taszkon belül a szálak közötti váltás nagyon gyors

Most, hogy tudjuk a taszkok és a szálak közötti különbséget, fontos leszögezni: ez az operációs rendszerek és a programozás elnevezési rendszere, és sajnálatos módon nem egyezik meg a számítógép-architektúrák témakörében használatos elnevezési rendszerrel. A több végrehajtási szálat támogató (multi-threading) processzorok ugyanis valójában több *taszkot* támogatnak, tehát ezeket helyesen több taszkot támogató (multi-tasking) processzoroknak kellene nevezni.

A továbbiakban az elterjedt (ellentmondásos) elnevezéshez tartjuk magunkat, tehát többszálú processzorokról és szálakról beszélünk, de közben nem fejeltjük el, hogy nem szálakról, hanem taszkokról van szó. Sajnos ez az elnevezési inkonzisztencia gyakran okoz zavart ezen a szakterületen, nemcsak magyar, hanem angol nyelven is.

20.2.2. Az időosztásos többszálú végrehajtás hardver támogatása

Ahogy azt az egyszerű utasítás pipeline esetében láttuk, egy program végrehajtása során számos olyan esemény történhet, ami az utasítások folyamatos végrehajtását megakasztja. Ilyen események lehetnek többek között:

- a TLB hiba,
- a cache hiba,
- adatfüggőségek, stb.

Jó lenne, ha a végrehajtásában megakadt utasításfolyamot ilyenkor a helyzet megoldásáig félre lehetne tenni, és egy másik utasításfolyamot elővenni, és azzal haladni tovább. Pontosan ezt teszik a többszálú végrehajtást támogató (multithreading) processzorok.

A többszálú végrehajtást támogató processzorok számos erőforrást többszöröznek: annyi utasításszámláló és annyi külön regiszter tároló van a processzorban, ahány szálat támogatnak. A szálak osztoznak (többek között) a TLB-ben, az utasítás-cache-ben, az elágazásbecslő adatszerkezetekben is: ezeket az erőforrásokat azonban nem szokták többszöröznі, hanem inkább a bejegyzésekben azt is nyilvántartják, hogy azok melyik szálhoz tartoznak. Pl. a TLB bejegyzések is kiegészülnek egy újabb mezővel, a bejegyzést használó szál azonosítójával. Mivel ezeken az erőforrásokon több szál osztozik, számolni kell azzal a kellemetlen lehetőséggel, hogy egy ”rosszul” viselkedő szál kiszorítja pl. a TLB-ből, vagy az utasítás-cache-ből a másik szál bejegyzéseit.

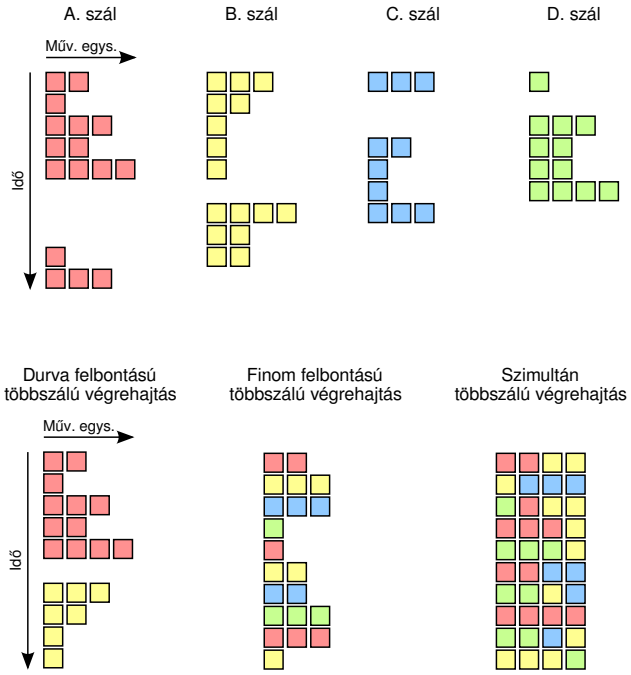
A szálak közötti váltás így semennyi időt sem vesz igénybe. Minden utasításhoz a lehívás után hozzárendelnek egy szálazonosítót, ami biztosítja, hogy az utasítás a saját szálának erőforrásain dolgozzon. A pipeline-ban így vegyesen előfordulnak majd különböző szálakhoz tartozó utasítások, melyek között a processzor csak annyiban tesz különbséget, hogy a szálnak megfelelő regiszterkészlettel, cache és TLB bejegyzésekkel, stb. dolgozik.

A processzor műveleti egységeinek szálakhoz rendelésére kétféle stratégia terjedt el:

- *Finom felbontású többszálú végrehajtás* (fine-grained multithreading) esetén a processzor minden órajelciklusban más szál utasításait hajtja végre. A 20.1. ábrán, középen található egy példa. Az ábrán függőlegesen lefelé telik az idő, vízszintesen pedig az egy órajelciklus alatt egyszerre ütemezhető (tehát független) utasítások száma van feltüntetve. A finom felbontásnak megfelelően először a piros, aztán a sárga, majd a kék és a zöld szál kap egy-egy órajelet, melynek során minden szál az egyszerre ütemezhető utasításait végrehajtja. Az ábrán jól látszik a többszálú végrehajtás előnye: pl. többszálúság nélkül a kék szálban az első három és az azt követő két utasítás között két szünet telik el, pl. adat-egymásrahatás miatt, ez idő alatt tehát a processzor kihasználatlan. Finom felbontású többszálúság esetén azonban a 4 szál miatt csak minden negyedik lépésben kapja meg a kék szál a processzort, tehát bármi is okozta a két szünetet, bőven volt ideje megoldódni, mire újra a kékre jut a sor.

- *Durva felbontású többszálú végrehajtás* (coarse-grained multithreading) esetén csak akkor vált szálat a processzor, ha az aktuális szál nagyon megakad (pl. cache hiba esetén, vagy akár csak másodszintű cache hiba esetén), 1-2 órajelnyi szünetre nem történik szál váltás. A szálak közötti váltás a processzort váratlanul éri (nem úgy, mint a finom felbontású esetben), tehát kell egy kis idő, mire a pipeline megtelik az új szál utasításaival, így a szál váltás néhány órajelnyi szünettel jár (lásd: 20.1. ábra bal szélső esete).

(A hardver támogatással nem segített többszálú végrehajtás a ”durva” esethez hasonlít, de akkor a szálak közötti váltás nem 1-2, hanem sokkal több órajelet vesz igénybe a regiszterek mentése és visszatöltése miatt.)



20.1. ábra. Lehetőségek a többszálú végrehajtás hardveres támogatására

20.2.3. Szimultán többszálú végrehajtás

A finom és durva felbontású többszálas megközelítéssel szemben a szimultán többszálú végrehajtás csak a szuperskalár processzorokhoz köthető. A szuperskalár processzorokban több műveleti egység áll rendelkezésre, melyek segítségével az utasítássorozatból egyszerre több, egymástól független utasítás hajtható végre. Ezek a műveleti egységek gyakran kihasználatlanok, ha a program felépítése olyan, hogy az utasítások közötti egymásrahatások az utasítások együttes végrehajtását gátolják.

A szimultán többszálú végrehajtás ezeket a szabad erőforrásokat hasznosítja: ha egy utasítássorozat utasításai nem képesek kihasználni az összes műveleti egységet, akkor vegyünk egy másik utasítássorozatot (ugyanazon taszk másik szálát), és próbáljuk meg annak utasításait végrehajtani a fennmaradó műveleti egységekkel. Azaz egy órajelciklusban több szál utasításai is (szimultán) végrehajthatók a rendelkezésre álló műveleti egységek erejéig. Minél több végrehajtási szálunk van, annál jobban ki tudjuk tölteni a műveleti egységeket.

A 20.1. ábra jobb oldalán láthatunk példát szimultán többszálúságra. A processzor a 4 műveleti egységét sorra tölti ki a 4 szál utasításaival, körbenforgó elv szerint. Persze vigyázni kell, hogy az azonos szálakhoz tartozó utasításcsoportok közötti szünetek száma megegyezzen az egyszálú végrehajtáskor szükségessel, hiszen az egymásrahatások feloldásához, cache, vagy TLB hiba, stb. feloldásához továbbra is ugyanannyi idő kell, csak most a kieső ”lyukakat” más szálak utasításaival tölthetjük ki.

A tapasztalatok szerint a többszálúság hardveres támogatása mindössze 5-10%-os komplexitás növekedéssel jár, így szinte minden nagyobb gyártó a többszálúság támogatásának valamely formája mellett tette le a voksát:

Processzor	Megjelenés éve	Többszálúság formája	Támogatott szálak száma
Intel Pentium 4	2002	Szimultán	2
Intel Itanium 2	2006	Durva felbontású	2
IBM POWER5	2004	Szimultán	2
IBM POWER7	2010	Szimultán	4
UltraSPARC T1	2005	Finom felbontású	4
UltraSPARC T2	2007	Finom felbontású	8

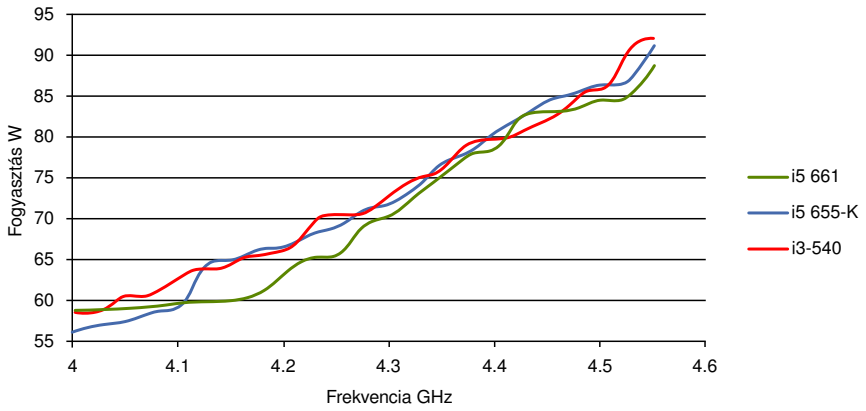
20.3. Multiprocesszoros rendszerek

A multiprocesszoros rendszerek teljes értékű processzorokból állnak.

20.3.1. Motiváció és nehézségek

A multiprocesszoros rendszerek használatát az alábbi szempontok motiválják:

- Az utasításszintű párhuzamosság korlátozott lehetőségei. Fentebb láttuk, hogy a tipikus programokban lakozó utasításszintű párhuzamosság gyakran már ahhoz sem elegendő, hogy 4-6 párhuzamosan működni képes műveleti egységet jól kihasználjon. A helyzetet rontja, hogy nincs 100% pontosságú elágazásbecslés, időnként cache hibák lassítják a memóriaműveleteket, stb. Ha több utasítást szeretnénk párhuzamosan végrehajtani, a párhuzamosan futtatható részeket a programozónak explicit módon jeleznie kell.
- Energiahatékonyság. A processzorok energia felhasználása nem egyenes arányban, hanem annál gyorsabban nő a számítási teljesítménnyel. Ezt támasztja alá a 20.2. ábra is (forrás: Anandtech), mely szerint ezen a bizonyos processzoron 12.5%-os órajel frekvencia (tehát számítási teljesítmény) növelés 50%-os fogyasztás-emelkedéssel járt. Ebből kiindulva, a fogyasztás szempontjából hatékonyabb lehet N darab processzort használni egységnyi órajellel, mint 1 processzort N-szeres órajellel.



20.2. ábra. Fogyasztás az órajel-frekvencia függvényében

- Költséghatékonyság. A processzorok ára sem lineárisan, hanem egy konvex függvényt követve (általában exponenciálisan) nő a számítási teljesítménnyel. Jól párhuzamosítható feladatok esetén kisebb költséggel lehet sok lassabb, de olcsó processzorból multiprocesszoros rendszert építeni, mint kevesebb (vagy esetleg egyetlen) gyors processzorból.
- Egyszerű bővíthetőség. Egy multiprocesszoros rendszert (ha erre megfelelően fel van készítve) könnyű bővíteni újabb processzorok hozzáadásával.
- Hibatűrés. Ha a sor processzorból egy meghibásodik, nem áll meg az élet, a többi átveszi a feladatait.

Ennyi előny számbavétele után felmerül a kérdés, mik az explicit párhuzamos rendszerek korlátai. A nehézségek a *szoftver* számlájára írhatók. Ahogy korábban már láttuk, a vezérlésáramlásos rendszerben a párhuzamosság felderítését szoftveresen kell megoldani. Ezt két módon tehetjük meg:

- Egyrészt fejleszthetünk olyan fordítóprogramot, amely a magas szintű nyelvben leírt programban automatikusan felderíti a párhuzamosítható részeket, és ennek megfelelő kódot generál (tehát a FORK és JOIN műveleteket a fordító helyezi el a programban). A programozónak könnyű a dolga, a szokásos szekvenciális gondolkodással írja a programot, a munka oroszlánrésze a fordítóprogramra hárul. Sajnos ilyen "intelligens" fordító, legalábbis számottevő hatékonysággal működő, nem áll rendelkezésre. (A párhuzamosító fordító a compiler fejlesztők "Szent Grál"-ja).
- Másrészt a párhuzamosítást a programozóra is lehet bízni: helyezze el ő a FORK és JOIN primitíveket, ahol azt jónak látja. Jobb híján ez ma a jellemző szemlélet. Sajnos a párhuzamosság "kézi" felderítése és megvalósítása nem egyszerű feladat, rengeteg nehezen felderíthető potenciális hibalehetőséggel.

20.3.2. Amdahl törvénye

Azt, hogy egy N processzorból álló multiprocesszoros rendszer mekkora teljesítménynövekményt képes hozni az egyprocesszoros rendszerekhez képest, az határozza meg, hogy a feladat (a futó program) milyen mértékben párhuzamosítható. Ha a feladat nem párhuzamosítható, akárhány processzort is vennénk, a szekvenciális futási igény miatt az mindig csak az egyik processzoron fut majd, és pontosan annyi idő alatt végez, mint egy egyprocesszoros rendszeren. A másik véglet az, ha a program teljes mértékben párhuzamosítható, ebben az esetben az N processzor N-szeres sebességgel hajtja végre azt (most tekintsünk el a program párhuzamos részei között zajló kommunikáció költségétől).

Egy valós program tartalmaz mind párhuzamosítható, mind szekvenciálisan végrehajtandó részeket is. A több processzor hadba fogásával elérhető teljesítménynövekedésről *Amdahl törvénye* segítségével kaphatunk képet. Az *Amdahl törvénye* egy rendkívül egyszerű, de nagyon is megdöbbentő eredményt adó összefüggés, melyet most magunk levezetünk.

Legyen a programunk P része párhuzamosítható, és a fennmaradó $1-P$ része szekvenciálisan végrehajtandó. Legyen a teljes program futási ideje egy egyprocesszoros rendszeren 1. A program $1-P$ részét adó szekvenciális részek tehát $(1-P)\cdot 1$ ideig futnak. A program párhuzamosítható P része pedig N processzor használatával $1/N$ ideig fog futni. Tehát N processzorral a teljes futási idő: $(1-P)+P/N$. Az N processzoros végrehajtás által elért teljesítménynövekedés megkapható az 1 processzoros végrehajtási idő (1) és az imént számolt N processzoros végrehajtási idő hányadosaként. Amdahl törvénye tehát:

$$S_P(N) = \text{Teljesítménynövekmény N processzorral} = \frac{1}{(1-P)+P/N}$$

Mi ezen a megdöbbentő? Lássunk egy numerikus példát: van 100 processzorunk, és ezen szeretnénk a programunkat legalább 80-szor gyorsabban lefuttatni, mint egy egyprocesszoros rendszeren. Ekkor $N=100$, $S_P(N)=80$, a fenti összefüggésből kijön, hogy

$$P = \frac{S_P(N)-1}{S_P(N)}\cdot\frac{N}{N-1} \simeq 0.9975,$$

vagyis ekkor a programnak mindössze 0.25%-a lehet szekvenciális! Ha a programunk 5%-a szekvenciális ($P=0.95$), akkor az $N=100$ processzorral elérhető teljesítménynövekmény:

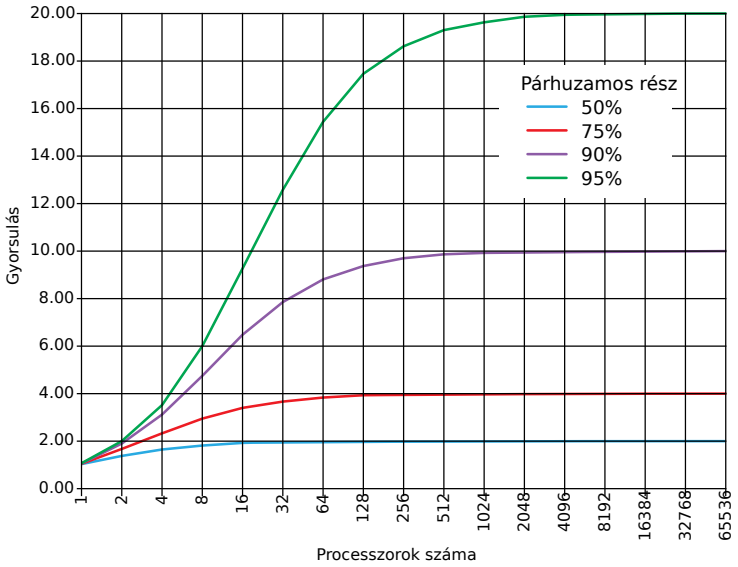
$$S_P(100) = \frac{1}{0.05+0.0095} \simeq 16.8,$$

tehát mindössze 16.8-szoros!

Próbáljuk meg kicsit jobban megérteni a dolog természetét. Vegyünk egy másik példát (a könnyebb számolás kedvéért). Egy program egy egyprocesszoros rendszeren 20 óra alatt fut le, és ebből 1 órányi futási idő esik a nem párhuzamosítható részekre. A 19 órányi párhuzamosítható rész a processzorok számának növelésével tetszőlegesen csökkenthető, végtelen sok processzorral aszimptotikusan 0 óra lesz a párhuzamos rész futási ideje. A szekvenciális rész viszont továbbra is 1 óráig fog futni, akárhány processzor is van. Tehát a maximális elérhető gyorsulás (végtelen sok processzort feltételezve): $S_P(\infty) = 20/1 = 20$-szoros. Ez az Amdahl törvényéből is kiolvasható, hiszen

$$S_P(\infty) = \lim_{N\to\infty} S_P(N) = \frac{1}{1-P}.$$

A 20.3. ábra bemutatja az elérhető gyorsulást a processzorok számának és a párhuzamosíthatóság mértékének függvényében. Szemmel látható, hogy minden görbe aszimptotikusan vízszintessé válik, és az előbbi $S_P(\infty)$ értéket veszi föl.



20.3. ábra. Elérhető gyorsulás a processzorok száma és a párhuzamosítás függvényében

Az Amdahl törvénye tehát rádöbbent minket arra, hogy a sok processzor minél jobb kihasználása érdekében kritikus fontosságú a program minél nagyobb részének párhuzamosíthatóvá tétele (P növelése). Sajnos a vezérlésáramlásos elven működő programozási nyelvekben ez nem megy magától: mind a fordítóprogramnak, mind a programozónak nehézséget okoz.

20.3.3. Rendszerezés

A multiprocesszoros rendszereket két szempont szerint rendszerezzük.

Az első szempont, hogy a különböző processzorokon futó taszkok hogyan tudnak egymással kommunikálni:

- Ha van olyan, minden processzor számára látható közös memória, melyen keresztül a taszkok egymással kommunikálni tudnak, akkor *osztott memóriával rendelkező multiprocesszoros rendszer*ről beszélünk.
- Ha ilyen közös memória nincs, a taszkok üzenetek segítségével tudnak egymással kommunikálni. Ezeket a rendszereket *üzenetküldésen alapuló multiprocesszoros rendszer*eknek nevezzük.

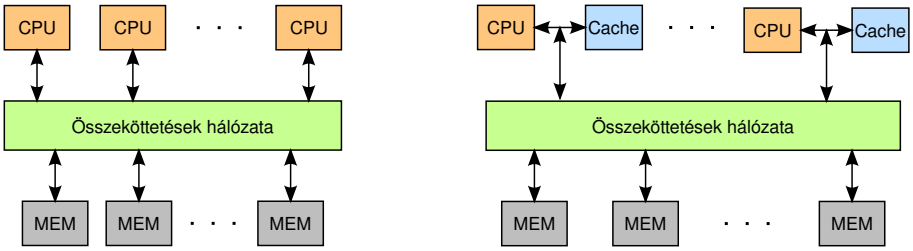
A másik szempont, hogy a multiprocesszoros rendszerben a memóriaműveletek ideje azonos-e:

- Ha igen, akkor UMA (Uniform Memory Access) multiprocesszoros rendszerünk van. UMA rendszerekben nincs jelentősége annak, hogy egy adatot hova írunk a memóriába: akárhova is írjuk, ugyanannyi ideig fog tartani mind az írás, mind a visszaolvasás minden egyes taszk számára egyformán.
- Ha nem, akkor NUMA (Non-Uniform Memory Access) multiprocesszoros rendszerünk van. NUMA rendszerekben nagyon is fontos, hogy egy adat hova kerül. Fontos, hogy a memóriába írt adat mind az íróhoz, mind az azt elolvasó másik processzorhoz ”közel” legyen.

Multiprocesszoros rendszerek osztott memóriával

Az osztott memóriával rendelkező multiprocesszoros rendszerekben (shared-memory multiprocessors) tehát van osztott memória, melyhez az összes processzor hozzáfér. Ezt a közös memóriát az összes processzor tudja címezni, írni/olvasni. A osztott memóriával rendelkező multiprocesszoros rendszerben futó taszkok rendkívül egyszerűen tudnak egymással kommunikálni: ha az egyik taszk adatot kíván küldeni egy másiknak, beírja az osztott memóriába, a másik pedig egyszerűen kiolvassa onnan.

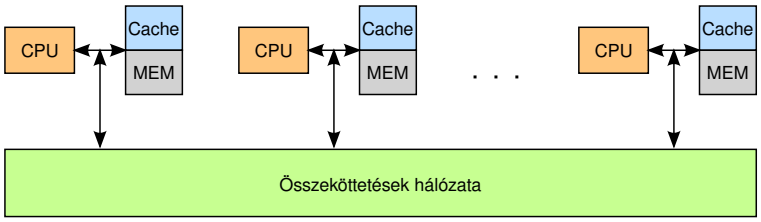
A 20.4. ábra a. pontjában egy UMA rendszer látható: a processzorok egy összeköttetés-hálózaton keresztül érik el a memória modulokat. A b. pontban ehhez képest annyi a különbség, hogy a processzorok saját cache-el rendelkeznek. Az igaz, hogy a memória-hozzáférések ideje ekkor már nem lesz egyforma, hiszen a cache gyorsabban elérhető, mint a közös memória, mégis ezt a megoldást is UMA-nak vehetjük (a cache-től eltekintve ez is uniform memória-elérési idejű). Ez a b. pont felel meg a napjainkban elterjedt több-magos processzoroknak is (a több-magos processzorokban a processzorok, a cache-ek és az összeköttetés-hálózat a tokban található).



20.4. ábra. Osztott memóriára alapozott UMA multiprocesszoros rendszerek

Sajnos az osztott memóriára alapozott UMA rendszerek (vagy ahogy másképp nevezik: SMP (Symmetric multiprocessor)) nem jól skálázódnak a processzorok számával. Ha egy új processzort illesztünk a régiek mellé, annak teljes memóriaforgalma terheli az összeköttetés-hálózatot, akár sokat kommunikál a többi processzorral, akár nem, ami könnyen szűk keresztmetszet lehet. A gyakorlatban ezért ilyen kialakítással legfeljebb 100-as nagyságrendben kötnek össze processzorokat.

Valódi skálázható megoldást a NUMA rendszerek jelentenek (20.5. ábra). Ebben az esetben a multiprocesszoros rendszer processzort, cache-t és memóriát tartalmazó csomópontokból áll. A címtér továbbra is közös a processzorok számára, de a címtér fel van osztva és a címtér egyes tartományai a különböző csomópontokhoz vannak hozzárendelve (pl. 32 bites címtér és 4 csomópont esetén az első csomóponthoz rendeljük a 0-1 GB tartományt, a másodikhoz az 1-2 GB címtartományt, stb.). Minden egyes processzor nagy sebességgel hozzá tud férni a címtartomány azon részéhez, amely a saját csomópontjában található memóriával van lefedve, a címtartomány más csomóponthoz rendelt részét pedig az összeköttetés-hálózat segítségével tudja elérni, lényegesen nagyobb hozzáférési idővel. Ennek a megoldásnak az a nagy előnye, hogy a csomópontok az általuk futtatott programok kódját és a csak helyben szükséges adatokat a lokális memóriában tudják tárolni, a lassabb, összeköttetés-hálózatot terhelő memória-hozzáférések csak akkor szükségesek, amikor a taszkok a kommunikáció jegyében egymás memóriájába írnak / memóriájából olvasnak.



20.5. ábra. Osztott memóriára alapozott NUMA multiprocesszoros rendszer

Üzenetküldéssel kommunikáló multiprocesszoros rendszerek

Az üzenetalapú (message passing) multiprocesszoros rendszerekben a csomópontok teljes számítógépek: processzor(ok)ból, memóriából, saját perifériákból állnak. A csomópontokban lévő memória a csomópont sajátja, kívülről közvetlenül nem elérhető. A csomópontok egy összeköttetés-hálózattal vannak összekötve. A különböző csomópontokon futó taszkok üzenetek segítségével kommunikálnak egymással. Ehhez az architektúrához is a 20.5. ábrán látható blokkséma tartozik, a különbség csak annyi, hogy itt a taszkok közötti kommunikáció nem memóriaműveletekkel, hanem üzenetekkel történik.

Az üzenetalapú kommunikáció két alapművelete:

- üzenet küldése (send)

- üzenet fogadása (bevárása, receive)

A kommunikációban résztvevő felek azonosítására minden taszkot saját, egyedi azonosítóval, ID-vel látnak el. Az üzenet – a mondanivaló mellett – tartalmazza a küldő és a címzett egyedi azonosítóját, valamint egy üzenet-azonosítót (tag) is. Az üzenet elküldése (send) után az összeköttetés-hálózat a címzett azonosítója alapján tudja az üzenetet kézbesíteni. A címzett a receive művelettel jelzi, hogy üzenetet vár, valamint, hogy pontosan milyen üzenetet is vár (megadható a címzettek és üzenet-azonosítók egy tartománya).

A kommunikáció lefolyása lehet *szinkron*, vagy *aszinkron*. Az üzenet küldése mindkét esetben ugyanúgy kezdődik: a küldő taszk a memóriában összekészíti az elküldeni kívánt üzenetet minden szükséges járulékos információval együtt, majd kiadja a ”send” parancsot. Ezután az üzenetek továbbításáért felelős hardver egység DMA művelettel közvetlenül kiolvassa a memóriából az üzenetet, és továbbítja a címzett felé. A küldő taszk futása *szinkron* esetben mindaddig fel van függesztve, míg a címzett az üzenetet el nem vette. *Aszinkron* esetben a küldő taszk az üzenet elküldése után azonnal folytatódik, attól függetlenül, hogy az üzenet a másik félhez még nem érkezett meg. A címzett taszk a ”receive” parancs segítségével megjelöli, hogy kitől vár üzenetet (küldő és üzenet azonosító alapján), és megad egy memóriacímet, ahova az érkező üzenet elhelyezését kéri. Ezután a címzett taszk futása felfüggesztésre kerül mindaddig, míg a várt üzenet meg nem érkezik. Az üzenet megérkezése után a címzett taszk futása folytatódik, miközben a megérkezett üzenet az általa megjelölt memóriacímen rendelkezésre áll.

20.4. Összeköttetések

Akár osztott memóriára alapozott (legyen az NUMA vagy UMA), akár üzenetküldésre alapozott multiprocesszoros rendszerünk is van, a csomópontokat mindenképpen össze kell kötni egymással:

- osztott memóriás, UMA rendszerek esetén minden processzort minden memóriamodullal össze kell kötni (de a processzorokat egymás között és a memóriamodulokat egymás között nem!),
- egyéb esetekben a processzorból, és memóriából álló csomópontokat kell összekötni egymással.

Az összeköttetések segítségével

- osztott memóriás multiprocesszoros rendszerekben memóriakérések,
- üzenetalapú multiprocesszoros rendszerekben üzenetek

továbbítása zajlik.

20.4.1. Tulajdonságok

Az összeköttetés-hálózatot egy $G = (V, E)$ gráffal írjuk le, ahol a V az összekötendő elemek (csúcsok) halmaza, az E pedig a csúcsok közötti élek halmaza. Ha $u \in V$ és $v \in V$ csomópontok egymással közvetlenül össze vannak kötve, akkor $(u, v) \in E$. A multiprocesszoros rendszerek természetéből fakadóan az összeköttetések kétirányúak, vagyis minden összeköttetés mentén egyidejűleg mindkét irányban történhet kommunikáció, vagyis G egy irányítatlan gráf.

Az összeköttetés-hálózatok tulajdonságai:

- Átmérő (diameter): $\delta(G)$ a csomópontok közötti leghosszabb távolság. Két csomópont közötti távolság alatt a két csomópont közötti *legrövidebb* út hosszát értjük. Legfeljebb ekkora utat tesznek meg a memóriakérések ill. üzenetek, míg eljutnak a rendeltetési helyükre. Minél kisebb az átmérő, annál gyorsabban jutnak el az üzenetek a küldőtől a címzettig.
- Felezési keresztmetszet (bisection width): $B(G)$ azon élek minimális száma, melyek elhagyásával a gráf két egyenlő (egymással össze nem kötött) részre bomlik. (Ha a csúcsok száma páratlan, az két rész csúcsszámának különbsége legfeljebb 1 lehet.) Ha N csomópont használja az összeköttetés-hálózatot, a felező vágatot átlagosan az üzenetek fele keresztezi, a sikeres kommunikációhoz $B(G) \geq N/2$ szükséges.
- Késleltetés (latency): az az idő, ami az üzenetek / memóriakérések átviteléhez szükséges a küldő és a cél csomópont között. Minél kisebb, annál jobb.
- Átvitel (throughput): az a sebesség, amivel az összeköttetés hálózaton új üzeneteket / memóriakéréseket lehet indítani. Minél nagyobb, annál jobb.

- Élösszefüggőség (arc connectivity): a gráf összefüggőségének megszüntetéséhez eltávolítandó élek minimális száma. Minél nagyobb az élösszefüggőség, annál több út kötheti össze a csomópontokat, így annál ritkábban fordul elő, hogy egy élre versenyhelyzet alakul ki.
- Költség (cost): egy összeköttetés-hálózat költségét sokféle szempont figyelembe vételével lehet meghatározni, például a szükséges élek és egyéb hálózati elemek számával.
- Skálázhatóság (scalability): a késleltetés, az átvitel és a költség hogy változik a processzorok számának függvényében.

Az összeköttetés hálózatok *topológiája* határozza meg, hogy a hálózat elemei geometriailag milyen módon vannak elrendezve.

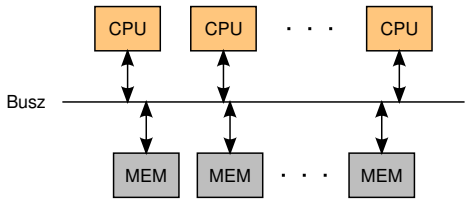
Az összeköttetés hálózatokat kétféle csoportra oszthatjuk: direkt és indirekt hálózatokra.

- Direkt (vagy statikus) hálózatok esetén a multiprocesszoros rendszer csomópontjai (processzorok ill. memória modulok) közvetlenül, pont-pont összeköttetésekkel kapcsolódnak egymáshoz. Az egymással szomszédságban nem álló csomópontok üzenetei más processzorokon, ill. memória modulokon kell, hogy áthaladjanak. A multiprocesszoros rendszer csomópontjai tehát aktív szerepet játszanak nem csak a saját, de mások üzeneteinek továbbításában is. Ez a megoldás az üzenetalapú multiprocesszoros rendszerekre jellemző.
- Indirekt (vagy dinamikus) hálózatok esetén a csomópontok nincsenek egymással közvetlen kapcsolatban. A csomópontok nem vesznek részt az üzenetek továbbításában: az üzenetet ”bedobják” az összeköttetés hálózat határán, és a hálózat felelőssége, hogy az üzenet a címzetthez jusson. Egy indirekt hálózatban egy csomópont nem találkozik olyan üzenettel, ami őt nem érinti. Indirekt hálózat például egy busz alapú hálózat, vagy egy speciális, kizárólag az üzenetek továbbítására szolgáló eszközökből álló hálózat. Ez az osztott memóriás multiprocesszoros rendszerek domináns felépítése.

20.4.2. Indirekt összeköttetés-hálózatok

Busz alapú összeköttetés-hálózatok

A busz alapú hálózatok talán a legegyszerűbb összeköttetés-hálózatok. A csomópontok (processzorok, ill. memória modulok) egy osztott, közös használatú buszon keresztül kommunikálnak egymással (20.6. ábra). A küldő fél a buszra teszi az üzenetet (pl. memória kérést), a buszra kapcsolódó többi csomópont pedig folyamatosan fülel, hogy neki szóló üzenet van-e éppen a buszon. A buszok nagy előnye, hogy nem csak pont-pont kommunikációt, hanem üzenetszórást is lehetővé tesz, tehát egy csomópont küldhet egy mindenkinek szóló üzenetet is. Természetesen a buszt egyidejűleg csak egyetlen csomópont használhatja, vagyis a busz használati jogáért versenyezni kell, valamilyen arbitrációs eljárást kell alkalmazni.



20.6. ábra. Busz alapú hálózat

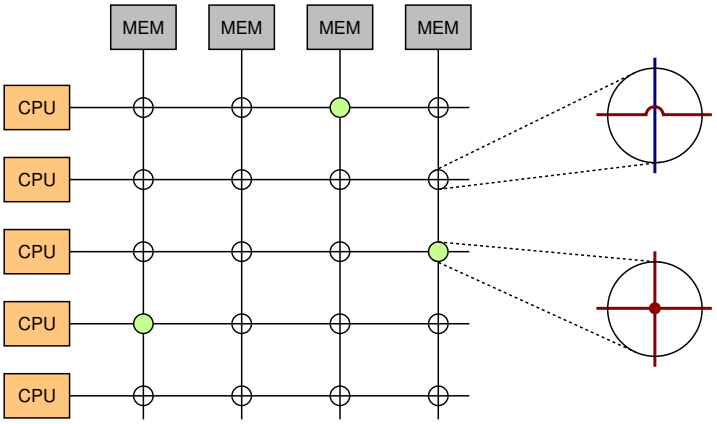
A busz alapú hálózatokban további tulajdonságai:

- Minden csomópont azonos távolságra van a többitől.
- A költség jól skálázódik: újabb csomópontok hozzáadásával nem nő (továbbra is egy osztott busz van használatban).
- Az átvitel azonban rosszul skálázódik: a busz átviteli képessége adott, és ez annál többfelé oszlik, minél több szereplő használja a buszt. Emiatt a busz alapú összeköttetés-hálózatokat csak kis számú (néhány tucat) processzor ill. memória összekötésére használják.

Busz alapú összeköttetést használtak a Sun Enterprise szerverek (1996-2001) és az Intel Pentium alapú multiprocesszoros rendszerek.

Crossbar hálózatok

P processzor és M memória modul összekötésének egy másik egyszerű módja a crossbar használata. A crossbar $P \cdot M$ db kétállású kapcsolót tartalmaz (20.7. ábra). Ha az i. sor j. kapcsolóját bekapcsoljuk, az i. processzor összeköttetésbe kerül a j. memóriamodullal. A crossbar egy blokkolásmentes hálózat, ami azt jelenti, hogy amikor egy processzor egy memóriamodulhoz kapcsolódik, azzal egyidejűleg bármely más processzor bármely más memóriamodulhoz kapcsolódhat.



20.7. ábra. Crossbar hálózat

A crossbar hálózatok előnye, hogy egyidejűleg több processzor is teljes sebességgel kommunikálhat a memória modulokkal (mindaddig, amíg különböző memóriamodulokkal kommunikálnak), hátránya, hogy az ára rosszul skálázódik a processzorok és a memóriák számával. Egy újabb processzor csatlakoztatásához 1 újabb összeköttetés, valamint M db új kapcsoló bekötése szükséges.

Crossbar hálózat köti össze például az IBM Power4, Power5, valamint a Sun Niagara processzormagjait az L2 cache memóriabankjaival.

Többfokozatú crossbar hálózatok

Láttuk, hogy a buszok nagyon jól skálázódnak költség szempontjából, de rosszul átvitel szempontjából. A crossbar pont fordítva: jól skálázódik átvitel szempontjából, de rosszul költség szempontjából. A rossz skálázhatósági tulajdonságaik mindkettő használatát csak kis méretű multiprocesszoros rendszerekben teszik lehetővé.

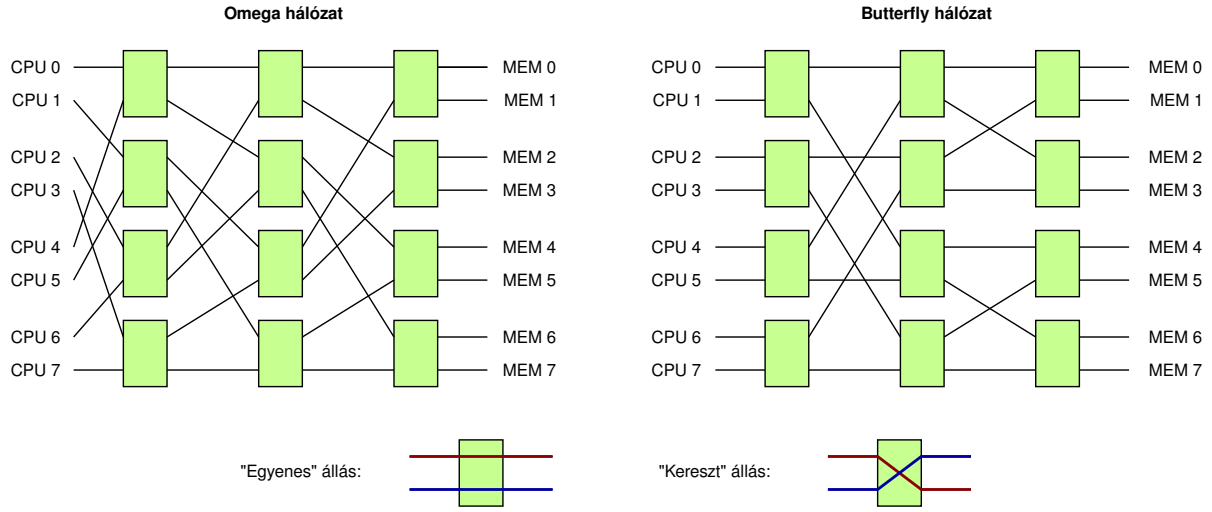
A többfokozatú crossbar (multistage crossbar) az arany középút: átvitel szempontjából jobban skálázódik a busznál, és költség szempontjából jobban skálázódik a crossbar-nál.

A többfokozatú crossbar ötlete, hogy vegyünk kis méretű crossbar kapcsoló hálózatokat, és ezekből, mint építőelemekből rakjunk össze egy nagy méretű összeköttetés-hálózatot. E köré az egyszerű ötlet köré lassan egy egész elmélet született, és alapjául szolgált mind a régi telefonközpontok, mind a modern számítógép hálózatok berendezéseinek működéséhez.

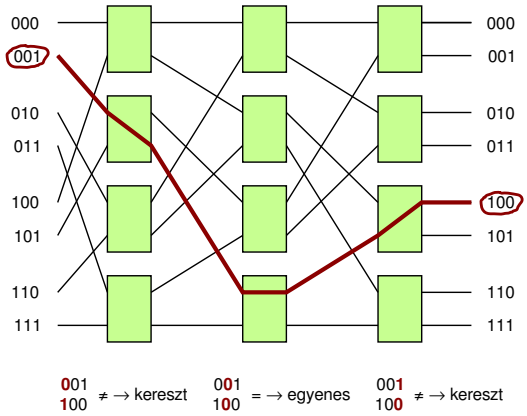
Most egy szándékosan leegyszerűsített képen keresztül mutatunk be a többfokozatú crossbar működését: 2x2-es elemi crossbar kapcsolókból fogunk építkezni. Arra, hogy ezeket az elemi kapcsolókat hogyan kell összekötni, hogy végül nagy számú csomópont összekötésére legyenek alkalmasak, többféle megoldás is létezik. A 20.8. ábra 8 processzor és 8 memória modul összekapcsolását mutatja be omega és butterfly elrendezésben. Mindkét esetben 3 fokozatra, és fokozatonként 4 elemi kapcsolóra volt szükség.

Az omega hálózatokban a kapcsolók működése rendkívül egyszerű (lásd: 20.9. ábra). A forrás és a cél csomópontot bináris számokkal azonosítjuk. A kapcsolóhálózat i. fokozatában a forrás és a cél csomópont azonosítójának i. bitjét kell összehasonlítani: ha a 2 bit megegyezik, ”egyenes” állásba, ha különbözik, ”kereszt” állásba kell az elemi kapcsolót állítani, így végül pont a kívánt forrás és cél csomópont lesz összekötve.

Sajnos a többfokozatú crossbar hálózatok nem blokkolás-mentesek: előfordulhat, hogy két csomópont-pár között olyan utak vezetnek, melyeket nem lehet egyidejűleg felépíteni. A 20.10. ábrán például a kék és a piros útvonal az első és a második fokozatban is ugyanazon kapcsolókon halad át, de a két összeköttetés e két kapcsoló különböző állását igényeli. Mivel a kapcsolók nyilván nem lehetnek egyszerre ”egyenes” és ”kereszt” állásban is, az egyik csomópont-pár várakozni kényszerül.



20.8. ábra. Omega és Butterfly hálózatok



20.9. ábra. Kapcsolók állítása az Omega hálózatban

Az Omega hálózatok P processzor és memória esetén $\log_2 P$ fokozatból állnak, minden fokozatban $P/2$ elemi kapcsolóval, tehát összesen $P/2 \cdot \log_2 P$ elemi kapcsolóra van szükség. Ilyen összeköttetés-hálózatot használt a $P = 512$ processzorból álló IBM RP3.

20.4.3. Direkt összeköttetés-hálózatok

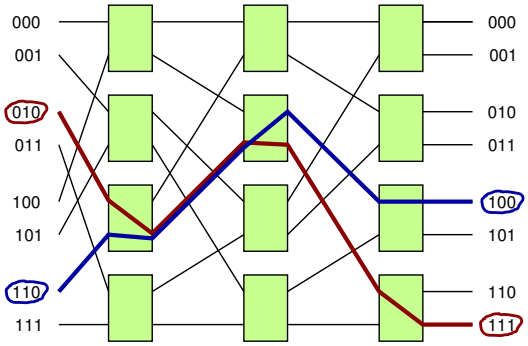
Teljesen összekötött hálózat

Egy teljesen összekötött hálózatban minden csomópont minden más csomóponttal össze van kötve. A 20.11. ábra egy 8 csomópontból álló teljesen összekötött hálózatot ábrázol. Ez a megoldás ideális abból a szempontból, hogy minden üzenet egyetlen lépésben eléri a címzettet. Blokkolás, versenyhelyzet nincs.

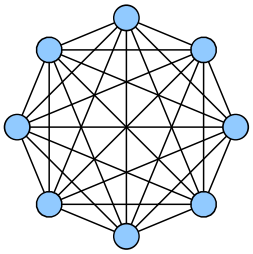
Egy N csomópontból álló teljesen összekötött gráf átmérője 1, felezési keresztmetszete $N^2/4$ (hiszen ha két $N/2$ csomópontból álló részgráfot összekötünk egy teljesen összekötött gráffá, akkor $N/2 \cdot N/2$ élre van szükség). Az élösszefüggőség $N - 1$, ennyi élet kell eltávolítani, hogy egy csomópont leszakadjon. A teljesen összekötött gráf átvitelben jól, költségben rosszul skálázódik, hiszen egy $N + 1$. csomópont hozzáadása nem lassítja a már meglévők kommunikációját, viszont N új él hozzáadását igényli.

Lineáris és gyűrű topológia

A teljesen összekötött hálózatnak ugyan előnyös tulajdonságai vannak, de költség és skálázhatóság szempontjából nagy számú csomópont összekötésére nem alkalmas. Az összeköttetések számának tekintetében a másik végletet



20.10. ábra. Blokkolás az Omega hálózatban



20.11. ábra. Teljesen összekötött hálózat

a lineáris és a gyűrű topológiák jelentik. Ebben az esetben a belső csomópontok csupán két szomszéddal vannak összekötve, a két szélső csomópontnak a lineáris hálózatok esetében csak egy szomszédja van, míg gyűrű esetén a két szélső csomópontot egymással összekötve azok fokszáma is kettő lesz (20.12. ábra).

20.12. ábra. Lineáris és gyűrű topológiák

N csomópontból álló lineáris topológia esetén a hálózat ármérője $N-1$, hiszen ez az első és az utolsó csomópont között vezető út hossza. A felezési keresztmetszet és az élösszefüggőség is 1, hiszen 1 él elhagyásával a gráf kétfelé szakad.

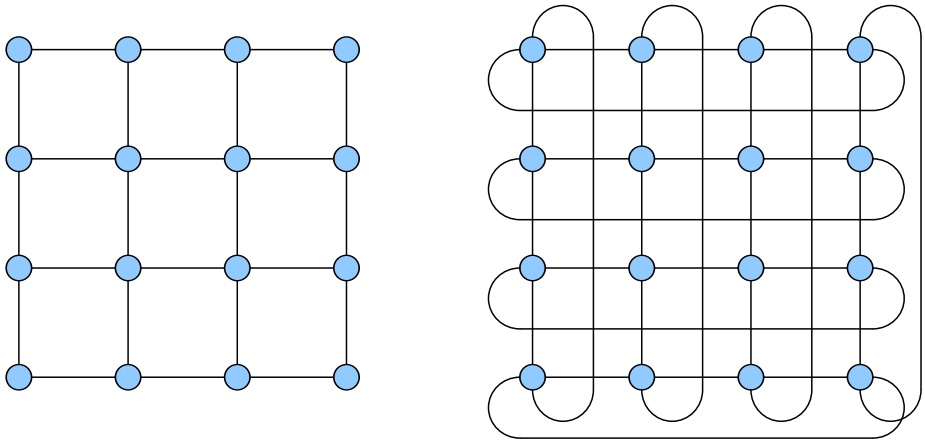
A gyűrű jellemzői valamivel jobbak: átmérője $N/2$, hiszen ennyi lépés alatt a közelebbi irányba elindulva minden csomópont elérhető. Mind a felezési keresztmetszet, mind az élösszefüggőség 2, ennyi élet kell elhagyni a gráf széteséséhez.

A lineáris és a gyűrű topológiák tehát költség szempontjából jól skálázhatók: egy új csomópont hozzáadása mindössze egyetlen új él hozzáadásával jár. Sajnos az átvitel szempontjából ezek a megoldások rosszul skálázhatók, hiszen kevés az él, az újabb és újabb csomópont hozzáadásával járó kommunikációs többlet ezt a kevés élet terheli.

Kétdimenziós háló és tórusz

A lineáris és gyűrű topológiák legfőbb hátrányán, az átvitel rossz skálázhatóságán segít, ha a csomópontokat kétdimenziós elrendezésben kötjük össze. A háló topológia esetén egy négyzetrácsot kapunk. A kétdimenziós tórusz a hálótól csak annyiban tér el, hogy a szélső csomópontok össze vannak kötve a másik oldali megfelelőjükkel (20.13. ábra).

Feltéve, hogy mindkét dimenzióban $\sqrt{N}$ csomópont van, a kétdimenziós háló átmérője $2 \cdot \sqrt{N}$, hiszen ennyi lépés kell, mire egy üzenet az egyik sarokból az átellenes sarokba eljut. A felezési keresztmetszet $\sqrt{N}$, az élösszefüggőség pedig 2 (a sarokban lévő csomópont 2 él eltávolításával leválasztható).



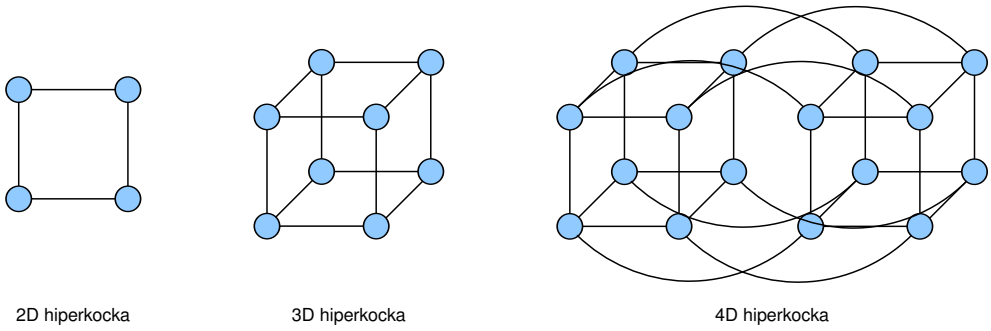
20.13. ábra. Kétdimenziós háló és tórusz topológiák

A kétdimenziós tórusz esetén az átmérő $2 \cdot \sqrt{N}/2$, a felezési keresztmetszet $2 \cdot \sqrt{N}$, az élösszefüggőség pedig 4.

A kétdimenziós tórusz egy gyakorlatban elég gyakran használt topológia, az Alpha 21364 processzora köré épülő szerverekben például ezt alkalmazták. Mind a háló, mind a tórusz elve kiterjeszthető több dimenzióra is, pl. 3D háló topológia a meteorológia modellezésére különösen hatékony, ilyet használt a Cray T3E szuperszámítógép is.

Hiperkocka

Egy érdekes és számos hasznos tulajdonságot mutató topológia a hiperkocka. Ha N csomópontunk van, a hiperkocka $\log_2 N$ dimenziós, és minden dimenzió mentén 2 csomópontot tartalmaz. A 20.14. ábrán egy 2, 3 és 4 dimenziós hiperkocka látható 4, 8 és 16 csomóponttal.



20.14. ábra. Hiperkockák

Egyáltalán nem könnyű belátni, de az ábrán jól követhető, hogy egy k dimenziós hiperkocka átmérője k, élösszefüggősége is k, felezési keresztmetszete pedig $N/2$.

20.4.4. Útvonalválasztás

Az útvonalválasztás határozza meg, hogy egy multiprocesszoros rendszerben az üzenetek, ill. memóriakérések hogyan jutnak el a küldőtől a címzettig. Hacsak nem szomszédos a küldő és a címzett, ezek a kérések több csomópontot is érintenek. Direkt összeköttetés-hálózat esetén ezek a csomópontok processzorok, ill. memória modulok, indirekt esetben pedig az összeköttetés-hálózat üzenetek továbbítására szolgáló eszközei. Bármelyik esetről is legyen szó, az üzenetek továbbítását csomópontról csomópontra *kapcsolásnak* nevezzük, az abban résztvevő szereplőket pedig *kapcsolóknak*. Tehát a kapcsoló feladata, hogy a hozzá befutó üzenetet továbbadja

a megfelelő szomszédos kapcsolónak, hogy az minél hamarabb eljusson a címzetthez (direkt összeköttetés-hálózatokban tehát a kapcsoló funkcionalitás magában a processzorban, vagy a processzorból és memóriából álló csomópontokban van megvalósítva, indirekt esetben pedig a hálózat ilyen kapcsolókból áll).

Az útvonalválasztó algoritmusokat az alábbi két csoportba soroljuk:

- Statikus útvonalválasztásról beszélünk, ha az üzenetek útvonala kizárólag a küldőtől és a címzettől függ. Tehát minden üzenet, melynek ugyanaz a küldője és a címzettje, minden körülmények között ugyanazon az útvonalon fog haladni.
- Dinamikus útvonalválasztás esetén a hálózat pillanatnyi állapota is szerepet játszik az üzenettovábbítási döntésekben. Dinamikus útvonalválasztással lehetővé válik az összeköttetés-hálózat forgalmas részeinek elkerülése (tehát a jobb terhelés-kiegyenlítés), és a meghibásodott összeköttetések kikerülése is.

Az útvonalválasztó algoritmusoknak a további elvárásoknak kell eleget tenniük:

- gyors működés: különösen a kapcsolókat nem célszerű hosszú, időigényes feladatokkal terhelni, hiszen ezzel más, ugyanazt a csomópontot keresztező üzeneteket is fenntartunk
- holtpont-mentesség: ne fordulhasson elő, hogy üzenetek egy halmaza a továbbítás során olyan szituációba kerül, hogy egyes erőforrások használatáért (pl. összeköttetések, tárolók, stb.) kölcsönösen egymásra várakoznak.

Statikus útvonalválasztás

Az alábbiakban három teljesen eltérő megközelítést tekintünk át a statikus útvonalválasztásra.

Útvonalválasztás dimenzió-sorrendben (dimension-order routing): Ez az eljárás akkor alkalmazható, ha a csomópontok valamilyen szabályos, többdimenziós geometriai alakzat mentén vannak összekapcsolva. Egy ilyen rendszerben a csomópontok az alakzatban elfoglalt helyük szerint vannak azonosítva. Pl. egy kétdimenziós háló (20.13. ábra) esetén a csomópontokat a függőleges és vízszintes index alapján egy (x,y) párossal azonosíthatjuk. Ha egy kapcsoló kap egy üzenetet, akkor ránéz a címzett azonosítójára: ha annak x koordinátája kisebb, mint az övé, akkor a baloldali, ha nagyobb, a jobboldali szomszédja felé továbbítja. Ha a címzett x koordinátája megegyezik a kapcsolóéval, akkor ránéz a következő koordinátára, az y-ra. Ha nagyobb, mint az övé, felfele, ha kisebb, lefele küldi tovább. Háromdimenziós háló esetén ugyanez a helyzet, de ott az azonosító 3 tagú: (x,y,z). A kapcsolók először az x szerint határozzák meg a továbbküldési irányt, majd egyezés esetén az y, végül a z koordináta alapján döntenek a továbbküldés irányáról.

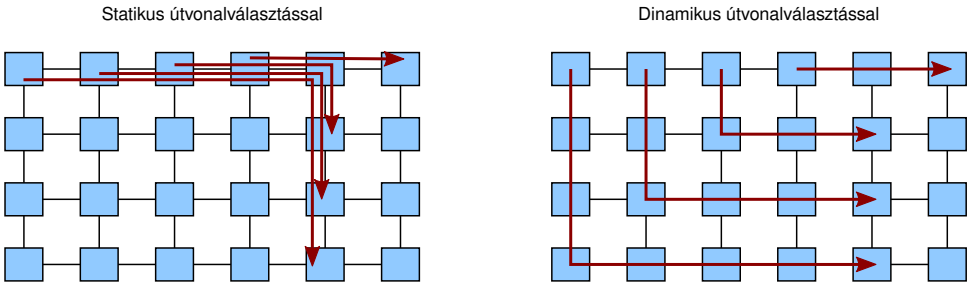
Küldő által meghatározott útvonalválasztás (source-based routing): Ebben az esetben maga a küldő találja ki az útvonalat a címzett felé, és az általa kigondolt útvonalat, vagyis a keresztezni kívánt csomópontok listáját hozzáfűzi magához az üzenethez. A kapcsolók dolga rendkívül egyszerű: fogni kell az üzenetet, csak le kell olvasni a következő csomópont azonosítóját, és továbbítani felé. Ezzel minden intelligencia a küldőnél van, tehát minden küldő csomópont számára elengedhetetlen az összeköttetés-hálózat topológiájának és a hibás összeköttetéseknek a naprakész ismerete. Az eljárás egy hátrányos tulajdonsága, hogy az üzenetek méretét a csatolt érintendő csomópont-lista megnöveli.

Táblázat alapú útvonalválasztás (table-driven routing): A kapcsolók ebben az esetben egy nagy táblázatot tárolnak, mely minden potenciális címzetthez tartalmaz egy bejegyzést, mégpedig azt, hogy a neki címzett üzeneteket melyik szomszédos csomópont felé kell továbbítani. Ha a kapcsoló kap egy üzenetet, megnézi a címzett mezőjét, felcsapja a táblázatát, és a címzetthez tartozó bejegyzésnek megfelelő szomszédjának továbbítja az üzenetet. Ha az összeköttetés-hálózat nagy számú csomópontból áll, a táblázat mérete túl nagy, a benne lévő keresés pedig túl időigényes lehet.

Dinamikus útvonalválasztás

A dinamikus eljárások adaptívak, alkalmazkodnak a hálózat aktuális forgalmi viszonyaihoz, és a meghibásodott összeköttetéseket is képesek megkerülni. Ennek érdekében azonban sokkal nehezebb kiszámítani az üzenetek továbbítási idejét (egyszer rövidebb, máskor hosszabb ideig tart), nehezebb biztosítani a holtpont-mentességet, valamint intézkedni kell róla, hogy az üzenetek véges idő alatt kézbesítve legyenek, vagyis hogy ne keringhessenek a hálózatban az örökkévalóságig.

A 20.15. ábrán egy olyan szituációt látunk, ahol 4 küldő-címzett páros között zajlik kommunikáció egyidejűleg. A statikus eljárás szerint ez közös összeköttetéseket tartalmazó útvonalakon valósul meg. Közös részeket tartalmazó útvonalak használata egy nagy hálózatban szinte elkerülhetetlen – ha ezeket az útvonalakat véletlenül éppen egyszerre kell használni, akkor az átviteli késleltetés megnő, hiszen a közös összeköttetéseken az üzeneteket csak egymás után lehet elküldeni. A dinamikus útvonalválasztással ez a probléma kiküszöbölhető, a példában a dinamikus útvonalválasztó például négy független útat talált.



20.15. ábra. Dinamikus útvonalválasztás

21. fejezet

Multiprocesszoros rendszerek osztott memóriával

21.1. Az osztott memóriakezelés alapproblémái

A 20.3.3. fejezetben láttuk, hogy a multiprocesszoros rendszerek megvalósításának egyik lehetséges módja, hogy a processzorok közös címtartománnyal rendelkeznek, és a különböző processzorokon futó programok az osztott memórián keresztül képesek egymással adatot cserélni, kommunikálni.

Ez a megoldás rendkívül népszerű, hiszen napjainkban a számítástechnika minden ágában, még a legolcsóbb szegmensben is széles körben elterjedtek a többmagos processzorok, melyek valójában osztott memóriát (közös fizikai címteret) használó multiprocesszoros rendszerek. A ”többmagos processzor” elnevezés tulajdonképpen félrevezető, hiszen a ”magok” minden szempontból teljes értékű processzorok, mindegyik saját utasításszámlálóval, cache memóriával, műveleti egységekkel, stb. rendelkezik (ellentétben a szuperskalár processzorokkal, melyek csak egyetlen utasításszámláló mentén futtatják a programot, és ellentétben a többszálú párhuzamosságot támogató processzorokkal, melyek csak a műveleti egységek jobb kihasználása érdekében tartanak kézben több utasításszámlálót). Ennek megfelelően az osztott memóriakezelés működésének megismerése a szoftverfejlesztők számára is fontos, hiszen csak a hardver ismeretében érthető meg, hogy egy többmagos környezetben futó program teljesítményét mi határozza meg, mik a lehetőségek az optimalizálásra.

21.1.1. A cache koherencia-probléma

Egyprocesszoros rendszerekben a cache memória a tárhierarchia kulcsfontosságú eleme, hiszen önmagában a viszonylag lassú rendszermemória nem lenne képes a jóval sebesebb processzort folyamatosan utasításokkal és adatokkal ellátni.

A multiprocesszoros rendszerekben a cache memória szerepe még fontosabb. Hiszen nem elég, hogy a DRAM-ra épülő rendszermemória technológiai okokból lassú, de a memóriához intézett kéréseknek és a memória válaszának még egy összeköttetés hálózaton is át kell haladnia, amely egyrészt további késleltetés forrása, másrészt egy közös kommunikációs csatorna, melyen több processzor memóriaforgalma osztozik.

Következésképp multiprocesszoros rendszerekben is célszerű cache memóriát használni. Csakhogy a közös memória cache-elése újabb problémákat vet fel. Transzparens cache esetén a cache memória a rendszermemória blokkjainak *másolatát* tartalmazza. Ha több processzor is jelen van, akkor a rendszermemória egy blokkja esetleg több processzor cache-ében is jelen lehet. Ha az egyikük ír erre a blokkra, akkor gondoskodni kell róla, hogy minden szereplő értesüljön a változásról. Ezt azonban nem is olyan egyszerű (hatékonyan) elérni.

A probléma szemléltetésére vegyünk egy példát. Két processzor, $\mathscr{P}_1$ és $\mathscr{P}_2$ hivatkozik a rendszermemória $\mathscr{M}$-edik blokkjára, melynek értéke kezdetben x. Ezután $\mathscr{P}_1$ megváltoztatja a blokk tartalmát x'-re. Mivel $\mathscr{P}_1$ tudatában van, hogy a memória osztott erőforrás, a *write-through* politikát követve (10.5.5. fejezet) hamar át is vezeti a változásokat a rendszermemóriába, így oda is az x' kerül. A rendszer szereplői azonban az igyekezet ellenére sem látják ugyanazt a képet a memória tartalmáról, hiszen a $\mathscr{P}_2$ processzor cache-ében továbbra is a régi tartalom, x található (21.1. ábra). Ilyenkor azt mondjuk, hogy a memóriarendszer nem *koherens*.

A formális definíció szerint a rendszermemória akkor koherens, ha az alábbi három feltétel teljesül:

Lépés	Esemény	$\mathscr{P}_1$ cache-e	$\mathscr{P}_2$ cache-e	Memóriatartalom
1				x
2	$\mathscr{P}_1$ kiolvas	x		x
3	$\mathscr{P}_2$ kiolvas	x	x	x
4	$\mathscr{P}_1$ módosít	x'	x	x'

21.1. ábra. A cache koherencia-probléma szemléltetése

1. Ha a $\mathscr{P}_1$ processzor a memória $\mathscr{M}$. címére ír, majd ezután ugyanonnan olvas, akkor az általa odaírt értéket kell visszakapnia, ha időközben más processzor nem írt ugyanerre a címre. (Természetes feltétel, egyprocesszoros rendszerekre is.)

2. Ha a $\mathscr{P}_1$ processzor a memória $\mathscr{M}$. címére ír, majd ezután a $\mathscr{P}_2$ processzor ugyanonnan olvas, akkor $\mathscr{P}_2$-nek egy bizonyos idő elteltével a $\mathscr{P}_1$ által odaírt értéket kell visszakapnia, ha időközben más processzor nem írt ugyanerre a címre. Tehát a memóriatartalom megváltozásának minden szereplő számára láthatóvá kell válnia. (Ezt fogja megsérteni a 21.1. ábra példája, mivel a $\mathscr{P}_2$ processzor nem értesül a változásról.)

3. Az ugyanazon memóriacímre irányuló írás műveletek sorban egymás után történnek meg. Tehát ha két processzor két különböző értéket ír ugyanazon címre, akkor a két írás sorrendjét a rendszer minden résztvevője ugyanannak látja.

A koherencia tehát valójában nem a cache-re, hanem a teljes memóriarendszerre vonatkozik, de mivel a problémák forrása a cache, a szakirodalom cache koherencia-problémának (cache coherence problem) nevezi. Megjegyezzük továbbá, hogy cache koherencia-probléma egyprocesszoros esetben is felmerül, amikor egy periféria DMA segítségével olyan memóriaterületet módosít, melynek van másolata a cache-ben. Ez a helyzet azonban könnyen kezelhető, hiszen az operációs rendszer pontosan tudja, hogy a periféria hová fog írni, azt is tudja, mikor végez, és kérheti a processzortól a cache ürítését. Többprocesszoros rendszerben azonban a koherenciaprobléma gyakrabban lép fel, és sokkal inkább teljesítménykritikus.

21.1.2. A memória konzisztencia-probléma

21.2. Cache koherencia-protokollok busz összeköttetés esetén

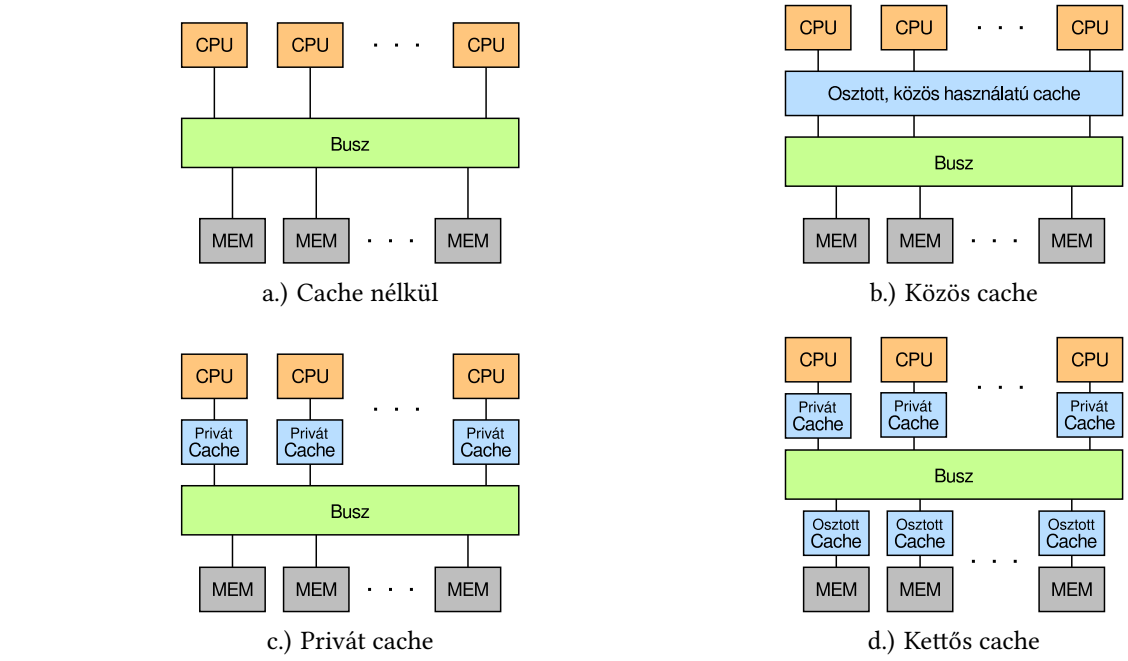
Ez a fejezet olyan multiprocesszoros rendszerek koherenciaproblémájával foglalkozik, melyben a memóriaműveletek ideje minden processzor számára hozzávetőleg azonos (UMA architektúra), valamint a processzorok és a memória (ill. memóriák) egy közös buszon, vagy bármilyen üzenetszórást támogató összeköttetés-hálózaton keresztül kommunikálnak.

21.2.1. A cache-ek elhelyezkedése és menedzsmentje

A cache elhelyezésére több lehetőség is adódik. A 21.2. ábra a.) részében egyáltalán nincs cache. Ez a teljesség kedvéért számba vett elméleti lehetőség, hiszen multiprocesszoros rendszerekben a teljesítmény szempontjából a cache megléte fontos. A b.) ábrán egyetlen, minden résztvevő számára osztott cache-t találunk. A koherenciaprobléma ilyenkor könnyen megoldható, viszont a processzorokat az első szintű cache-el összekötő összeköttetés-hálózat túlságosan növeli a memóriaműveletek késleltetését. A c.) és d.) ábra a gyakorlatban releváns esetek. Minden processzor saját, nagy sebességű cache-el rendelkezik, emellett a memóriaegységek is tartalmazhatnak további cache-t az elérés gyorsítása érdekében.

A koherenciaprobléma oka, hogy a rendszermemória egy adott blokkjának másolatát több processzor is tárolja a cache-ében. Megoldására kétféle mód kínálkozik. Nem lép fel koherenciaprobléma, ha

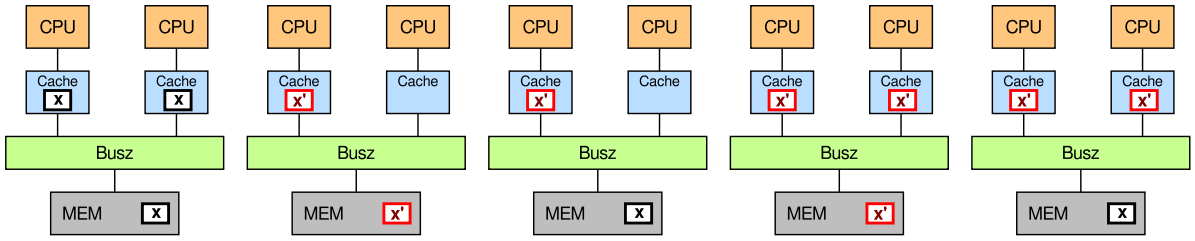
- a blokkot csak egyetlen processzor cache-eli,
- vagy ha gondoskodunk róla, hogy az összes cache-elt másolat tartalma ugyanaz legyen.



21.2. ábra. Cache-ek elhelyezkedése busz alapú multiprocesszoros rendszerekben

Mindkét megoldás életképes, és mindkettőt használják valós rendszerekben. Az első az *érvénytelenítésre alapozott* (write-invalidate), a második pedig a *frissítésre alapozott* (write-update) algoritmusok alapgondolata.

Annak függvényében, hogy a cache menedzsment write-through vagy write-back típusú-e, illetve hogy write-invalidate vagy write-update megoldást használ-e, négy különböző lehetőség különíthető el, melyet a 21.3. ábra mutat be.



21.3. ábra. Koherenciaprotokollok osztályozása

A write-invalidate alapú megoldások esetén a cache tartalmának módosításakor a többi processzor egy olyan üzenetet kap, melyben felszólítják az érintett blokk érvénytelenítésére. Mivel a busz, mint összeköttetés, támogatja az üzenetszórást, elegendő kiszólni a buszra, hogy az adott számú blokkot kéretik érvényteleníteni. Write-through esetben ezután a memória tartalma is frissül, write-back esetben csak akkor, ha a blokk elhagyja a cache-t.

A Write-update ehhez képest annyiban tér el, hogy az érvénytelenítő üzenet helyett egy frissítő üzenet kerül a buszra, a blokk új tartalmával együtt. Minden szereplő, aki megtalálja a cache-ében az érintett blokkot, leveszi a buszról a friss tartalmat és a cache-ében lévő idejétmúlt változatot erre cseréli.

A cache hibák számát tekintve a write-update természetesen előnyösebb, viszont sokkal nagyobb busz forgalmat generál, mint a write-invalidate. Minden egyes processzor minden egyes írás műveletnél a buszra teszi az érintett blokk tartalmát, ami a busz átbocsátó képességét hamar megtöltheti, így ez a rosszabbul skálázható megoldás.

21.2.2. Érvénytelenítésre alapozott protokollok

Mivel a rendszer szereplői egy közös buszhoz kapcsolódnak, képesek ”kihallgatni” a többiek memóriaelérést célzó üzeneteit (snooping), és ezekből a kihallgatott üzenetekből képesek megállapítani, hogy mi lehet a cache-ükben tárolt blokkok állapota, mikor jött el az ideje az érvénytelenítésnek. Sőt, a hatékonyabb együttműködés érdekében a koherencia fenntartását segítő üzeneteket is hirdetnek ezen a közös buszon.

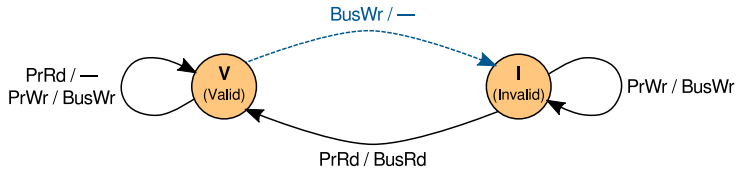
Egy egyszerű kétállapotú protokoll

Write-through, write-invalidation és write-no-allocate cache esetén a koherencia különösen egyszerűen megvalósítható. Elegendő, ha a processzorok egy cache-ükben lévő blokk módosulásakor értesítik a többi processzort a buszon kihirdetett ”BusWr” üzenet segítségével. A többi processzor ennek hatására érvényteleníti az érintett blokkot a saját cache memóriájában.

A cache-ben a blokkok kétféle állapotot vehetnek fel:

- *Valid* (érvényes) állapotban a blokk a processzor cache memóriájában van. Olvasáskor semmi tennivaló nincs, hiszen az olvasás művelet nem befolyásolja a koherenciát. Íráskor viszont a write-through politika szerint egy ”BusWr” busz tranzakció indul, ami a memóriába írja a megváltozott tartalmat. Ezt a ”BusWr” üzenetet azonban a busz sajátosságai miatt nem csak a memória hallja, hanem a többi, hallgatózó processzor, melyek ennek hatására érvénytelenítik a cache-ükben az érintett a blokkot, amennyiben az benne volt.
- *Invalid* (érvénytelen) állapotban a blokk nincs a cache-ben. Olvasáskor egy ”BusRd” üzenet segítségével kell beszerezni az érintett blokk kópiáját a cache-be, amely így valid (érvényes) állapotba kerül. Íráskor a write-no-allocate elvnek megfelelően a blokk nem kerül a cache-be, marad az érvénytelen állapot, de a ”BusWr” üzenetre szükség van, hogy a memória tartalma frissüljön, és hogy a többi processzor cache-éből ez a blokk a kikerüljön.

A viselkedést a 21.4. ábrán látható állapotgép írja le. A folytonos vonal a helyi (processzor által kezdeményezett) műveletek által kiváltott, a szaggatott vonal pedig a buszról lehallgatott üzenetek által kiváltott átmeneteket jelzi. Az átmenetek címkéin az első tag az átmenetet kiváltó esemény, a második tag pedig az átmenet hatására a buszra kerülő üzenet.



21.4. ábra. A cache állapota egyszerű, kétállapotú protokoll esetén

Sajnos ennek az egyszerű megoldásnak súlyos hátrányai vannak. Mind a write-through, mind a write-no-allocate viselkedés hátrányosan hat a hatékonyságra. Mivel minden egyes processzor minden egyes írás műveletét egy ”BusWr” busz üzenet kísér, a busz terhelése óriási.

Az MSI protokoll

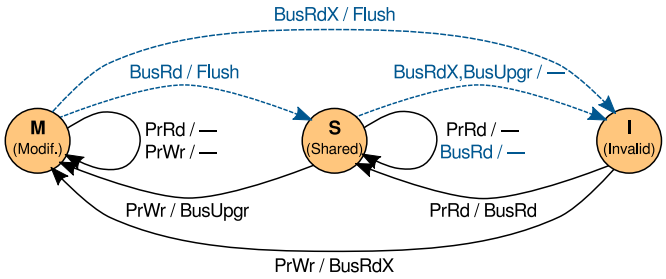
Ha a memória frissítése write-through helyett write-back módon történne, az nagyban csökkentené a busz terhelését, hiszen ekkor a cache-ben tárolt adatok olvasása és írása is helyben, a busz kihagyásával zajlana. Ehhez a cacke blokkok három különböző állapotát kell megkülönböztetni:

M *Modified* (módosult) állapotban van egy blokk, ha írás történt rá, de a módosítás még nem került át a rendszermemóriába (ezt a write-back politika megengedi). A blokkról pillanatnyilag ez az egyetlen létező cache-elt kópia a rendszerben, és az egyetlen hely, amely az adott blokk naprakész állapotát tartalmazza.

S *Shared* (oszott) állapotban vannak azok a blokkok, melyek több processzor cache-ében is jelen vannak. A blokk rendszermemóriabeli képe, és az összes cache-ben szereplő másolat mind-mind megegyeznek egymással.

I *Invalid* (érvénytelen) egy blokk, ha nincs benne a cache-ben.

Az állapotok kezdőbetűiből származik az *MSI protokoll* elnevezés. A három állapot a cache memóriában a ”valid” és ”dirty” bitek segítségével tökéletesen leírható. A ”valid=1” és ”dirty=1” kombináció az **M** állapotnak felel meg, a ”valid=1” és ”dirty=0” az **S** állapotnak, míg a ”valid=0” nem más, mint az **I** állapot. Az állapotok közötti átmeneteket és az átmenetek során keletkező busz üzeneteket a 21.5. ábra mutatja be.



21.5. ábra. Cache blokk állapotátmenetek az MSI protokollban

Ha **M** állapotú blokkhoz nyúl a processzor, akkor nem kell a buszon üzengetnie sem az írás (”PrWr”), sem az olvasás (”PrRd”) művelet során, hiszen garantált, hogy ő birtokolja az egyetlen létező, aktuális tartalommal bíró példányt az egész rendszerben. Amint egy másik processzor is hozzá szeretne férni a blokkhoz, ez a kivételezett helyzet megszűnik, és a módosult tartalmat a rendszermemóriába is át kell vezetni. Kívülről jövő olvasási igény (”BusRd”) esetén a blokk **S** állapotúvá válik, mivel az így már több cache memóriában is jelen lesz. Az olvasási igényre válaszként a processzor a buszra teszi a blokk tartalmát (”Flush” üzenet), amit mind a ”BusRd” kérője, mind a rendszermemória hall, a ”BusRd” kérője ezt a saját cache-ébe teszi **S** állapottal, a rendszermemória pedig frissíti a tartalmát. Ha egy processzor kizárólagos használati igényt jelent be erre a blokkra (”BusRdX”), azért, mert írni szeretne rá, akkor a blokk kikerül a cache-ből, de előtte egy ”Flush” üzenettel a buszra kerül a tartalma. A buszról a blokk tartalmát leolvassa a ”BusRdX” kezdeményezője, és **M** állapottal a saját cache-ébe teszi azt, de a blokk tartalmát leolvassa a rendszermemória is, így ott is frissül a tartalom.

Az **S** állapotú blokkokat minden processzor szabadon olvashatja. Írás esetén gondoskodni kell róla, hogy a többi processzor törölje a cache-éből, hogy az állapota végül **M** lehessen. A törlést (érvénytelenítést) a ”BusUpgr” üzenettel lehet kérni, melyre a hasonló célú ”BusRdX” üzenettel ellentétben nem kell válaszolni.

Ha cache hiba történik (**I** állapotú blokkra hivatkozáskor), olvasás esetén a processzor egy ”BusRd” üzenettel, írás esetén pedig egy ”BusRdX” üzenettel szerzi be a blokkot, amely az előbbi esetben **S**, utóbbiban **M** állapottal kerül a cache-be.

A későbbiek során fontos lesz, hogy a klasszikus MSI protokollban azokat a cache hibákat, melyek osztott blokkra vonatkoznak (vagyis legalább egy cache-ben **S** állapottal megtalálhatók), a lassú rendszermemória szolgálja ki. Ez nem tűnik racionálisnak, hiszen célszerűbb lenne az adatokat egy olyan cache-ből átkérni, ahol már jelen van, hiszen a cache→cache átvitel sokkal gyorsabb, mint a memória→cache átvitel. Csakhogy az MSI protokoll a blokk potenciálisan sok birtokosa közül nem képes kiválasztani azt az egyet, amelyik a blokk átküldésével kisegíti a cache hibába futó processzort. Ez az MSI protokoll egyik fogyatékossága.

Rövidítés	**Jelentése**
PrRd	A helyi processzor olvassa a blokkot
PrWr	A helyi processzor írja (módosítja) a blokkot
BusRd	Egy távoli processzor olvasni szeretné a blokkot
BusWr	Egy távoli processzor módosította ezt a blokkot
BusRdX	Egy távoli processzor kizárólagos használatra kéri a blokkot
BusUpgr	Egy távoli processzor a cache blokk érvénytelenítését kéri
Flush	Az egyik processzor a buszra teszi egy cache blokk tartalmát

21.1. táblázat. Egy cache blokk állapotát érintő események

Érdemes megvizsgálni, hogy az MSI protokoll teljesíti-e a 21.1.1. fejezetben, a koherencia kritériumaként

felsorolt három feltételt. A második feltétel teljesül, mivel a megváltozott memóriatartalomról minden szereplő értesül, amikor a ”BusRd” üzenettel kezdeményezett olvasás műveletet hatására a legfrissebb tartalmat birtokló cache az **M** állapotot elhagyva a ”Flush” üzenettel a buszra teszi a blokkot. Az első és a harmadik kritérium teljesülése mélyebb megfontolást igényel. A buszon megjelenő olvasás (”BusRd”) és írást kezdeményező (”BusRdX”) üzenetek sorrendjét a busz természetéből fakadóan minden szereplő ugyanannak látja. Azonban vannak olyan rejtett írás műveletek (az **M** állapotban), melyek nem látszanak a buszon. A blokkot birtokló processzor nyilván abban a sorrendben észleli ezeket az írás műveleteket, amilyen sorrendben kérte azokat. A többi processzor viszont csak akkor értesül a változásról, amikor egy ”BusRd” vagy ”BusRdX” hatására megszűnik az **M** állapot, ekkorra azonban az összes ”rejtett” írás művelet lezajlik, a rendszer minden szereplője ismét a buszon keresztül, egyformán értesül a módosult tartalomról.

Esemény	$\mathscr{P}_1$ cache	$\mathscr{P}_2$ cache	$\mathscr{P}_3$ cache	Busz tranzakciók	Adatforrás	Rendszermemória
Kezdetben	**I**	**I**	**I**	–	–	Friss
$\mathscr{P}_1$ olvas	**S**	**I**	**I**	BusRd	Memória	Friss
$\mathscr{P}_1$ ír	**M**	**I**	**I**	BusUpgr	–	Nem friss
$\mathscr{P}_2$ ír	**I**	**M**	**I**	BusRdX, Flush	$\mathscr{P}_1$	Nem friss
$\mathscr{P}_3$ olvas	**I**	**S**	**S**	BusRd, Flush	$\mathscr{P}_2$	Friss
$\mathscr{P}_1$ olvas	**S**	**S**	**S**	BusRd	Memória	Friss

21.6. ábra. Példa az MSI protokoll működésére

Az MSI protokoll működését egy konkrét példán szemlélteti a 21.6. ábra. A példában 3 processzor, $\mathscr{P}_1$, $\mathscr{P}_2$ és $\mathscr{P}_3$ játszik szerepet, melyek ugyanazon a cache blokkon végeznek műveleteket. A táblázat követi a blokk állapotát a processzorokban, a műveleteket kísérő busz tranzakciókat, valamint azt, hogy az adatot melyik szereplő szolgáltatja. Az első lépésben a $\mathscr{P}_1$ olvasási kérését, melyet egy ”BusRd” üzenettel jelzett, a rendszermemória szolgálta ki. A második lépésben ugyanerre a blokkra ír, ehhez előbb egy ”BusUpgr” üzenetet ad ki, amely jelzi a többi processzornak, hogy aki rendelkezik másolattal, az érvénytelenítse azt (jelen esetben nem volt ilyen processzor). Erre a ”BusUpgr” üzenetre nem kell adattal válaszolni, ez csak egy jelzés a többiek irányába. A harmadik lépésben $\mathscr{P}_2$ szeretné írni a blokkot, egy ”BusRdX” üzenetet ad ki, amely kizárólagos használatra kéri azt. A blokk tartalmát $\mathscr{P}_1$-től kapja meg, egy ”Flush” üzenet által, melynek hatására a rendszermemória is frissül. A negyedik lépésben $\mathscr{P}_3$ olvasná a blokkot, ”BusRd”-et ad ki. A $\mathscr{P}_2$, a blokk legfrissebb változatának tulajdonosa ennek hatására egy ”Flush” tranzakcióval válaszol, melyet mind a $\mathscr{P}_3$, mind a rendszermemória észlel. Végül az ötödik lépésben $\mathscr{P}_1$ is olvasni szeretné a blokkot. Mivel már van **S** állapotú kópia a rendszerben, az MSI pedig nem tud ezzel a helyzettel mit kezdeni ($\mathscr{P}_2$ vagy $\mathscr{P}_3$ adja át?), a rendszermemóriából jön a válasz a ”BusRd”-re.

MESI protokoll

Az MSI protokoll egészen hatékonyan működik, hiszen az **M** állapotú blokkra vonatkozó memóriaműveletek egyáltalán nem terhelik a buszt. Némi finomítással azonban tovább lehet fejleszteni az eljárást.

Gyakori szituáció, hogy egy program olvas valamit a memóriából (pl. egy lokális változót), majd módosítja annak értékét, vagy módosítja egy közeli (azonos blokkban lévő) adat értékét. Az ilyen olvasás – írás pár az MSI protokollban két busz üzenettel járt: az olvasás ”BusRd” az **I** állapotból **S**-be, az ezt követő írás pedig a ”BusUpgr” üzenet kíséretében **S** állapotból **M**-be vitte az állapotgépet, és erre a két üzenetre akkor is szükség volt, ha a blokk valójában nem is osztott.

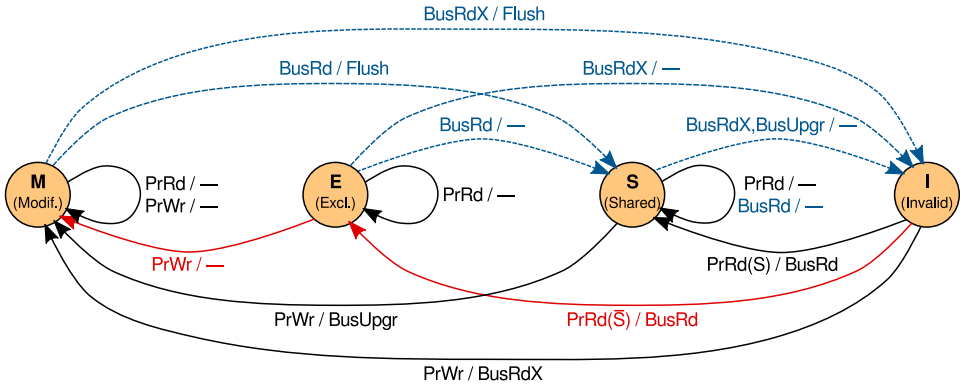
Az üzenetek száma csökkenthető, ha még egy állapotot vezetünk be:

E *Exclusive* (kizárólagos) állapotú egy blokk, ha ez az egyetlen másolat létezik belőle az egész rendszerben, és ez a másolat tiszta, tartalma megegyezik a rendszermemória tartalmával.

Emlékeztetőül: az **M** állapot azt jelentette, hogy csak egy másolat van, és az módosított, az **S** pedig azt, hogy több másolat is van, de tiszták, így az **E** pont a kettő között áll.

Az immár 4 állapotú állapotgép átmenetei a 21.7. ábrán láthatók. Két lényeges eltérés van az MSI protokollhoz képest. Egyrészt, az **E**-ből **M** átmenet nem jár busz üzenettel, tehát az olvasás – írás párosok (az ábrán pirossal jelölve) takarékosabbak a busszal. Másrészt, cache hibát okozó olvasás (**I** állapotban ”PrRd”) esetén nem egyértelmű,

hogy **S**-be, vagy **E**-be kell-e menni, attól függ, hogy az érintett blokk bent van-e valamelyik processzor cache-ében. Ezt a kérdést úgy lehet eldönteni, hogy miközben a processzor a ”BusRd” üzenet válaszára, tehát a blokk tartalmára vár, a többi processzor körülnéz a cache-ében, és ha azt tapasztalja, hogy neki megvan a blokk, akkor egy közösen használt, ”S”-nek nevezett (shared, osztott) vonalra logikai igaz értéket hajt. Ha a blokk megérkezik a buszra, a processzor ránéz az ”S” vonalra, és ha hamis értéket lát ($\overline{\text{S}}$, senkinek nincs meg a blokk), akkor **E** állapotba, ha igaz értéket lát (S, legalább egyvalakinek megvan), akkor **S** állapotba lép.



21.7. ábra. Cache blokk állapotátmenetek a MESI protokollban

A 21.8. ábra a MESI protokollt alkalmazza a korábbi példára. Az első két lépés pont egy olvasás – írás pár, amin az MSI-hez képest egy üzenet meg is takarítható. A kevesebb üzenetnek köszönhetően alacsonyabb a memóriaműveletek késleltetése, és kisebb a busz terheltsége is, ami így több processzor kiszolgálását is el tudja látni.

Esemény	$\mathscr{P}_1$ **cache**	$\mathscr{P}_2$ **cache**	$\mathscr{P}_3$ **cache**	**Busz tranzakciók**	**Adatforrás**	**Rendszermemória**
Kezdetben	**I**	**I**	**I**	–	–	Friss
$\mathscr{P}_1$ olvas	**E**	**I**	**I**	BusRd($\overline{\text{S}}$)	Memória	Friss
$\mathscr{P}_1$ ír	**M**	**I**	**I**	–	–	Nem friss
$\mathscr{P}_2$ ír	**I**	**M**	**I**	BusRdX, Flush	$\mathscr{P}_1$	Nem friss
$\mathscr{P}_3$ olvas	**I**	**S**	**S**	BusRd(S), Flush	$\mathscr{P}_2$	Friss
$\mathscr{P}_1$ olvas	**S**	**S**	**S**	BusRd(S)	Memória	Friss

21.8. ábra. Példa az MESI protokoll működésére

A MESIF protokoll

Ha a MESI protokollban egy processzornak olyan blokkra van szüksége, ami **S** állapotban jelen van már másoknál, akkor a kérést a lassú rendszermemóriának kell kiszolgálnia, hiszen nincs olyan mechanizmus, ami eldönti, hogy a kérésre a blokk birtokosai közül melyik válaszoljon a blokk átadásával. A *MESIF protokoll* éppen ezt a problémát orvosolja, mégpedig úgy, hogy egy újabb (immár ötödik) állapotot vezet be, amely kijelöli, hogy ebben a szituációban melyik blokkbirtokosnak kell válaszolnia. Az új, **F** (Forward, továbbító) nevű állapot ettől eltekintve minden egyéb szempontból megegyezik az **S** szerepével. Így, ha egy processzornak épp arra a blokkra van szüksége, ami máshol már osztott állapotban rendelkezésre áll, akkor a birtokosok közül az **F** állapotú cache fogja a kérést megválaszolni a rendszermemória helyett. A válaszként küldött cache blokkal együtt pedig a kérő az **F** állapot is megörökli, miközben a küldő visszavált **S**-be. Ez garantálja, hogy a blokk cache kópiái közül legfeljebb egy lehet **F** állapotban (ha épp egy sincs, az sem gond, ekkor a rendszermemória válaszolja meg a blokk olvasási kérést). A példa MESIF protokollos változata a 21.9. ábrán látható.

Mivel több helyzetben történik cache→cache átvitel, mint a MESI protokollban, a memóriaműveletek átlagos válaszideje rövidebb.

Esemény	$\mathscr{P}_1$ cache	$\mathscr{P}_2$ cache	$\mathscr{P}_3$ cache	Busz tranzakciók	Adatforrás	Rendszermemória
Kezdetben	**I**	**I**	**I**	–	–	Friss
$\mathscr{P}_1$ olvas	**E**	**I**	**I**	BusRd($\overline{\text{S}}$)	Memória	Friss
$\mathscr{P}_1$ ír	**M**	**I**	**I**	–	–	Nem friss
$\mathscr{P}_2$ ír	**I**	**M**	**I**	BusRdX, Flush	$\mathscr{P}_1$	Nem friss
$\mathscr{P}_3$ olvas	**I**	**S**	**F**	BusRd(S), Flush	$\mathscr{P}_2$	Friss
$\mathscr{P}_1$ olvas	**F**	**S**	**S**	BusRd(S), Flush	$\mathscr{P}_3$	Friss

21.9. ábra. Példa az MESIF protokoll működésére

A MOESI protokoll

A MESI protokoll másik népszerű kiterjesztése a *MOESI protokoll*, mely szintén egy új állapottal bővíti az állapotgépet. Ez a protokoll úgy spórol a busz és a memória sávszélességével, hogy a módosított blokkok megosztása esetén nem frissíti a rendszermemóriát (szemben a MESI-vel, ami **M→S** átmenet esetén mindig frissített), hanem egy újabb, **O** állapotba lép.

Az **O**, *owner* (tulajdonos) állapot azt jelenti, hogy a blokk módosítva lett, de a módosítás után több, csak olvasható (**S** állapotú) kópia is keletkezett belőle más processzoroknál, és a rendszermemória frissítése még nem történt meg. Az **O** állapotú cache felelős a többi processzor felől érkező olvasási kérések megválaszolásáért. Ha a blokk az **O** állapotot elhagyja, pl. azért, mert valaki más írni szeretne rá, akkor a módosítást át kell vezetni a rendszermemóriába is. A 21.10. ábra mutatja be a példa viselkedését a MOESI protokollal.

Esemény	$\mathscr{P}_1$ cache	$\mathscr{P}_2$ cache	$\mathscr{P}_3$ cache	Busz tranzakciók	Adatforrás	Rendszermemória
Kezdetben	**I**	**I**	**I**	–	–	Friss
$\mathscr{P}_1$ olvas	**E**	**I**	**I**	BusRd($\overline{\text{S}}$)	Memória	Friss
$\mathscr{P}_1$ ír	**M**	**I**	**I**	–	–	Nem friss
$\mathscr{P}_2$ ír	**I**	**M**	**I**	BusRdX, Flush	$\mathscr{P}_1$	Nem friss
$\mathscr{P}_3$ olvas	**I**	**O**	**S**	BusRd(S), Flush	$\mathscr{P}_2$	Nem friss
$\mathscr{P}_1$ olvas	**S**	**O**	**S**	BusRd(S), Flush	$\mathscr{P}_2$	Nem friss

21.10. ábra. Példa az MOESI protokoll működésére

21.2.3. Frissítésre alapozott protokollok

Az érvénytelenítésre alapozott protokollok hátránya, hogy akárhányszor egy blokk módosul, a rendszerben lévő többi kópiát érvényteleníteni kell. Az érvénytelenítés miatt a blokkot használó processzorok ezután újra kénytelenek betölteni a blokkokat a cache memóriájukba, ami egyrészt busz sávszélességet fogyaszt, másrészt növeli a memóriaműveletek átlagos késleltetését.

A frissítésre alapozott (write-update) protokollok ezzel szemben nem érvénytelenítenek, hanem a változásokat terjesztik a blokkot birtokló cache memóriák körében. A legnevezetesebb frissítésre alapozott koherenciaprotokoll a Dragon protokoll, melyet a Xerox Palo Alto-i kutatóközpontja (PARC) dolgozott ki a 80-as években.

Ez a megoldás minden processzor cache memóriájában a blokkok alábbi 4 állapotot különbözteti meg:

E *Exclusive-clean* (egyedi és tiszta) állapotú egy blokk, ha csak ebben az egy cache memóriában szerepel, és betöltése óta még nem módosult.

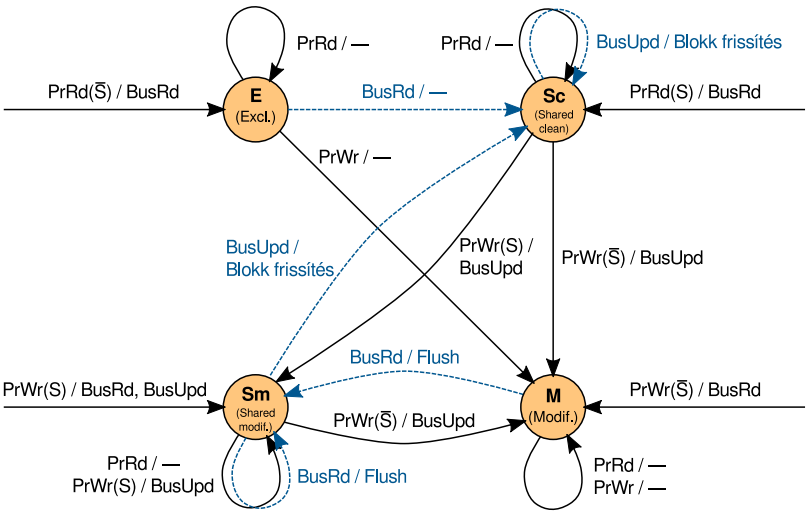
Sc *Shared-clean* (megosztott és tiszta) állapotú egy blokk, ha több cache memóriában is szerepel, és vagy betöltése óta változatlan, vagy nem ez a processzor végezte az utolsó módosítást.

Sm *Shared-modified* (megosztott és módosult) állapotú a blokk, ha több cache memóriában is szerepel, betöltése óta módosult, és ez a processzor végezte az utolsó módosítást.

M *Modified* (módosult) állapotú a blokk, ha csak ebben az egy cache memóriában szerepel, és betöltése óta módosult.

Az ”érvénytelen” állapot hiányának oka, hogy érvénytelenítés ebben a protokollban nincs, egy blokk csak akkor lehet érvénytelen, ha a processzor még sosem hivatkozott rá, vagy kapacitáshiány miatt korábban kikerült a cache-ből.

Az állapotátmeneteket a 21.11. ábrán láthatjuk. Az ”S”-el jelölt (shared) jelzésnek most is fontos szerepe van, hiszen ezáltal szerez tudomást a processzor arról, hogy a blokk más cache-ben is megtalálható-e. Ha egy busz tranzakció közben egy másik processzor igazba állította (jelezvén, hogy az érintett blokk osztott), akkor csak az **Sc** és **Sm**, ha az értéke hamis (más processzornál nincs kópia), akkor csak az **E** és **M** állapotot veheti fel egy blokk.



21.11. ábra. Cache blokk állapotátmenetek a Dragon protokollban

A Dragon protokollban minden memóriaírás műveletet – ha megosztott blokkra vonatkozik – egy frissítő (”BusUpd”) busz tranzakció követ, melynek során a módosult blokk megjelenik a buszon, lehetőséget adva a blokkot birtokló cache memóriáknak a tartalmuk frissítésére. Amelyik processzor az utolsó módosítást végezte, arra száll az **Sm** állapot. Vegyük észre, hogy ezáltal minden módosított blokk, bármennyi processzor cache-ében legyen is jelen, pontosan egy helyen lesz vagy **Sm**, vagy **M** állapotú. Ez az egy, kitüntetett cache lesz a felelős a rendszermemória frissítéséért, ha kapacitáshiány miatt a blokk a cache-ből végül távozni kényszerül.

Bár első látásra a frissítésre alapozott koherenciaprotokollok hatékonyabbnak tűnnek az érvénytelenítésre alapozottaknál, számos esetben éppen fordított a helyzet. Például, ha a memória-hozzáférések sorozatában sok az írás művelet, akkor jobb döntésnek tűnik az érintett blokkok kizárólagossá tétele (érvénytelenítéssel) és írássorozat helyi, buszkímélő végrehajtása, mint a busz ”BusUpd” üzenetekkel való elárasztása. Az egymást követő írás műveletek átlagos számából (write run) valamint az egyes busz üzenetek sávszélességigényéből ki lehet számolni, hogy melyik eljárás a takarékosabb, de ez természetesen a futó szoftvertől függ, a processzor tervezésekor a várható használat ismeretében kell a döntést meghozni.

21.3. Skálázható cache koherencia-protokollok

Ahogy azt a 20.3.3. fejezetben is láthattuk, igazán jól skálázható multiprocesszoros rendszereket NUMA felépítéssel lehet megvalósítani (20.5. ábra). NUMA rendszerekben az összeköttetés-hálózatot csak a távoli memóriaműveletek során kell igénybe venni. Ha jó a feladatok és adatszerkezetek csomópontokhoz rendelése, és a futó programok memóriahivatkozásai időben és térben lokálisak, akkor csak a különböző processzorokon futó programok közötti kommunikáció terheli az összeköttetés-hálózatot, és a memóriaműveletek átlagos késleltetése is alacsony, hiszen a legtöbb hivatkozás a csomópontok helyi, lokális memóriáját érinti.

Ha a NUMA architektúra csomópontjai cache memóriával is rendelkeznek, és a cache koherenciája az egész rendszerre nézve biztosított, akkor cc-NUMA (cache coherent non-uniform memory access) architektúráról beszélünk.

Sajnos a cache koherencia biztosítására a 21.2. fejezetben megismert protokollok nem alkalmazhatók közvetlenül. Az egyik probléma, hogy a hallgatózás (snooping) lehetővé tételéhez üzenetszórásos összeköttetés-hálózatot igényelnek, márpedig egy NUMA architektúrában jellemzően nem az van. (Hiszen a NUMA lényege a skálázhatóság, és a busz nem éppen a jó skálázhatóságáról ismert). A másik probléma a működésükből adódik: minden eddig megismert koherenciaprotokoll lényege az volt, hogy a memóriaműveletek által kiváltott *minden üzenetet* a rendszer *minden szereplő* megkapja, ami szintén a skálázhatóság ellen hat.

Szerencsére nincs szükség alapjaiban új megoldásokat bevezetni a skálázhatóság érdekében. A korábban megismert, a cache blokkok helyzetét nyomon követő állapotgépeket továbbra is megtartjuk, és a buszüzenetek is hasonlóak lesznek, csak az üzenetszórást kell feltétlenül kiváltani valami mással.

21.3.1. Könyvtárt használó koherencia protokollok

Az üzenetszórást a legegyszerűbben úgy lehet kiváltani, ha a koherencia fenntartása érdekében küldözgetett üzenetek *célzottan* jutnak el az érdekelt csomópontokhoz. Ehhez azonban pontosan tudni kell, hogy a rendszermemória egyes blokkjai mely csomópontok cache-ében találhatók meg. Ezt az információt a *könyvtár* (directory) tárolja, és mivel a cél a skálázhatóság, a könyvtár is elosztottan kerül megvalósításra.

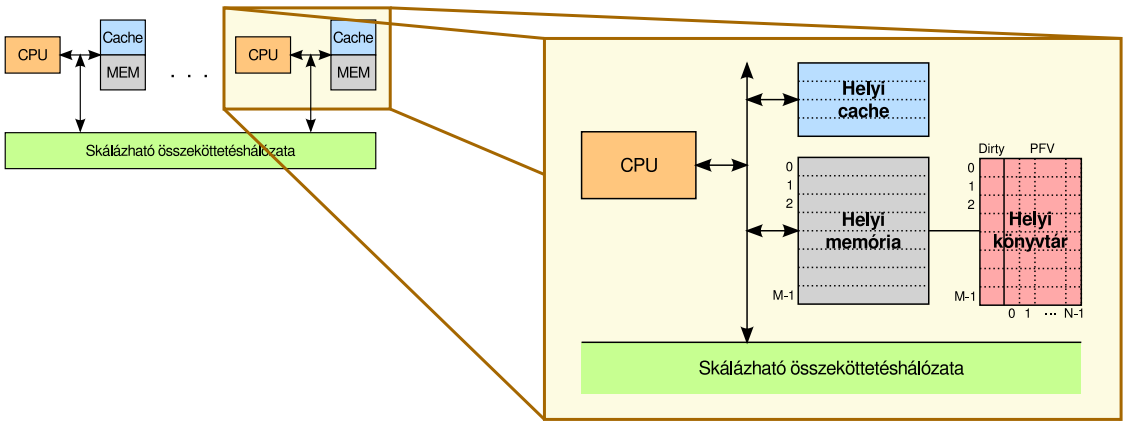
A könyvtár elhelyezésére két lehetőség adódik:

1. A könyvtár tekinthető az egyes csomópontok lokális memóriájának részeként, mely annak minden blokkjáról számon tartja, hogy a rendszer mely csomópontjai rendelkeznek belőle másolattal.

2. A könyvtár tekinthető az egyes csomópontok lokális cache-ének részeként, mely annak minden blokkjáról számon tartja, hogy mely egyéb csomópontok rendelkeznek még belőle másolattal.

A fejezet további része az első megoldást részletezi, vagyis úgy tekintjük, hogy a könyvtár a lokális memória része (21.12). Ha a lokális memória M blokkból áll, akkor ez a könyvtár is M bejegyzéses (az ábrákon M sorból áll). Minden egyes bejegyzés két fontos mezővel rendelkezik:

- A ”dirty” bit, ami 0, ha a lokális memóriában a blokk aktuális, legfrissebb tartalma található, és 1, ha a lokális memória tartalma (a blokkra nézve) idejétmúlt, a legfrissebb tartalom valamelyik csomópont cache memóriájában van (vélhetően **M** állapotban).
- A ”presence-flag vector” (PFV, jelenléti bitek vektora), melynek i. bitje 1, ha az i. csomópont cache-ében ez a blokk jelen van. N processzor esetén a PFV szélessége N.

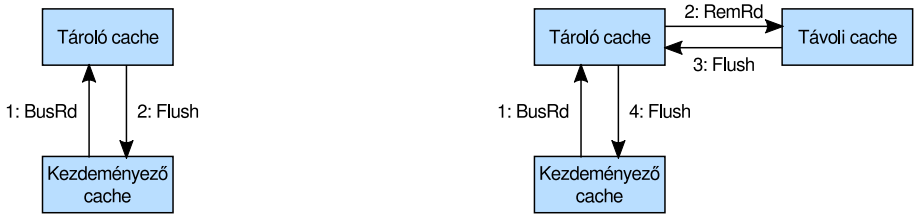


21.12. ábra. Könyvtár a cc-NUMA rendszerek csomópontjában

Az egyszerűség kedvéért most az érvénytelenítésen alapuló MSI protokoll könyvtárat használó változatát tekintjük át, de a többi fent tárgyalt koherenciaprotokoll értelemszerűen ugyanígy skálázhatóvá tehető.

Először tegyük fel, hogy az egyik processzor (mostantól ”kezdeményező”) egy olyan blokkra ad ki olvasási kérést, ami sem a saját cache-ében, sem a lokális memóriájában nem található. A cím alapján azonban egyértelműen eldönthető, hogy melyik csomóponthoz tartozik (pl. $N = 2^K$ csomópont esetén a fizikai címek felső K bitje megadja), és a ”BusRd” kérést célzottan ennek a csomópontnak (mostantól ”tároló”) küldi el. Két eset lehetséges.

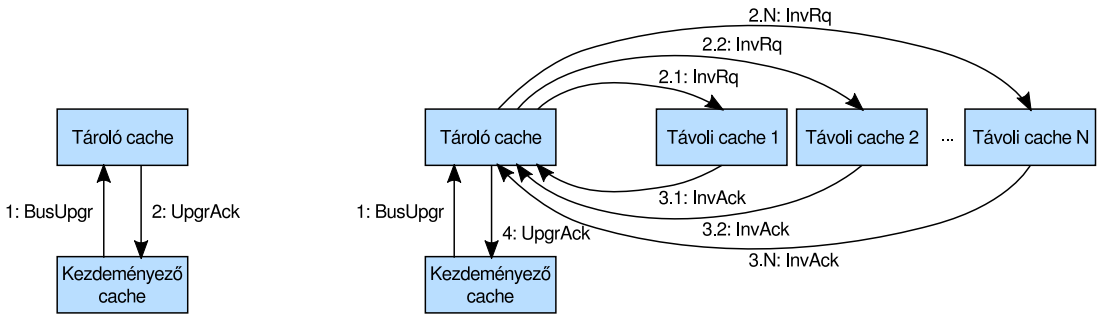
Ha a tároló csomópont könyvtárában a kérdéses blokkra dirty=0 szerepel (21.13. a.) ábra), akkor a blokkot postafordultával el is küldi a kezdeményezőnek egy ”Flush” üzenet formájában (ismét célzottan, üzenetszórás nincs). Ezzel egy időben a PFV megfelelő bitjét 1-be billentve feljegyzi, hogy már a kezdeményező csomópontnál is van egy példány a blokkból. Ha azonban a dirty bit 1 volt (21.13. b.) ábra), akkor a tároló csomópont a ”BusRd” kérést nem tudja azonnal megválaszolni. A PFV bitekből kideríti, hogy mely távoli csomópontnál van a módosult blokk, és küld számára egy távoli olvasás, ”RemRd” üzenetet, mire az visszaküldi a blokkot egy ”Flush” üzenetben, célzottan a tároló csomópontnak (közben a távoli csomópont végrehajtja a megfelelő lépést az állapotgépében, most pl. **M→S**). A blokk megérkezésével a tároló csomópont frissíti a memóriát, dirty=0-t állít be, egy ”Flush”-al megválaszolja a kezdeményező csomópont várakozó kérését, és frissíti a PFV megfelelő bitjét.



a.) Dirty=0 eset b.) Dirty=1 eset

21.13. ábra. A ”BusRd” tranzakció lebonyolítása könyvtár alapú koherencia protokollokban

Az írási kérések kezelése is hasonlóan egyszerű. Egy cache-beli, **S** állapotú blokk módosítása érdekében a kezdeményezőnek egy ”BusUpgr” üzenetet kell küldenie a tároló csomópont számára. Ha a tároló csomópont a PFV alapján úgy találja, hogy csak a kezdeményezőnek van a blokkról másolata, akkor rögtön egy ”UpgrAck” üzenettel nyugtázza a kérést, mire a kezdeményező **M** állapotba léphet, és már írhat is a cache-beli blokkjára (21.14. a.) ábra). Ha azonban a blokk osztott, más csomópont is rendelkezik kópiával, akkor a tároló csomópont nekik érvénytelenítő üzenetet küld (”InvRq”, ha az összeköttetés hálózat megengedi, akkor akár párhuzamosan), melyre azok nyugtával (”InvAck”) válaszolnak. Ha minden nyugta megjött, akkor mehet az ”UpgrAck” nyugta a kezdeményező kérésére (21.14. b.) ábra). Ezt követően a tároló csomópont 1-be billenti a dirty bitet (hiszen a memória tartalma nem friss), és a PFV bitjeit is frissíti, hiszen most már csak a kezdeményező csomópont cache-ében található meg a blokk.



a.) A blokk nincs megosztva b.) A blokk meg van osztva

21.14. ábra. A ”BusUpgr” tranzakció lebonyolítása könyvtár alapú koherencia protokollokban

A ”BusRdX” üzenet kezelése megegyezik a ”BusUpgr” üzenet kezelésével, az egyetlen eltérés, hogy a tároló csomópont egy ”Flush” üzenet kíséretében a blokkot is elküldi a kezdeményező csomópontnak.

A leírt viselkedés természetesen nem az egyetlen elképzelhető megoldás, a könyvtár alapú protokolloknak nagyon sok változata van, szinte minden gyártó bevet néhány trükköt az üzenetek számának csökkentésére.

21.3.2. A könyvtár memóriaigényének csökkentése

A 21.3.1. fejezetben leírt megoldás legnagyobb hátránya a könyvtár potenciálisan nagy mérete. Ha a memória M blokkból áll, és a csomópontok száma N, akkor minden egyes lokális memóriához egy $M \times N$ méretű könyvtár

tartozik (a dirty bitektől eltekintve), így a rendszerben az elosztott könyvtár teljes mérete $M \times N^2$, ami a csomópontok méretével négyzetesen arányos. Ez a paraméterek függvényében lehet elhanyagolhatóan kicsi, vagy megengedhetetlenül nagy is. Ha a cache blokk méretet B jelöli (bitben kifejezve), akkor a könyvtár relatív mérete a rendszer teljes memóriakapacitásához képest az alábbi módon számolható ki:

$$\frac{\text{könyvtárméret}}{\text{memóriaméret} + \text{könyvtárméret}} = \frac{M \times N^2}{M \times N \times B + M \times N^2} = \frac{N}{B + N}.$$

Feltéve, hogy a blokkméret 64 bájt ($B = 512$), a képlet alapján egy 16 csomópontos rendszerben a memória 3%-a fordítódik a könyvtár tárolására, de 512 csomópont esetén már 50%, 4096 csomópont esetén pedig már 89%, ami pazarlóan nagy szám. A fejezet további része a könyvtár méretének kordában tartására mutat be néhány elterjedt megoldást.

PFV helyett mutatóvektor

A klasszikus, PFV-re alapozott megoldás azért fogyaszt sok memóriát, mert a PFV *minden csomópontra* feljegyzi egy bittel, hogy rendelkezik-e a blokkal. Az azonban nagyon ritka szituáció, hogy egy adott blokk minden csomópont cache-ében egyidejűleg benne lenne. A gyakorlatban a megosztott blokkok jóval kevesebb cache-ben szerepelnek (legfeljebb néhányszor tíz, bár ez természetesen alkalmazásfüggő). Az ötlet tehát az, hogy a bitmező helyett minden blokkra tároljuk azon csomópontok azonosítóját, melyek épp birtokolnak egy kópiát belőle. Például $2^{12} = 4096$ csomópont esetén a PFV hossza 4096 bit, de ha ehelyett a blokk 16 potenciális birtokosának azonosítóját tároljuk, akkor csak $16 \cdot 12 = 192$ bitre van szükség.

Sajnos ez a megoldás sem tökéletes, hiszen fel kell készülni arra az esetre is, ha a blokkot túl sokan osztják meg, például 16-nál többen, ha minden blokkra csak 16 csomópont azonosítót tárolunk. Ekkor az alábbi lehetőségek közül választhatunk:

- Átváltás üzenetszórás módba. Ha túl sok csomópont cache-ében szerepel egy blokk, akkor az azzal kapcsolatos koherencia üzeneteket a rendszer minden csomóponthoz eljuttatja. Ez természetesen nagyon sávszélesség-igényes megoldás, de ha kellőképpen ritkán következik be, akkor elfogadható kompromisszum lehet.

- Nem engedjük meg, hogy egy adott számú csomópontnál többen osszák meg a blokkot. Ha újabb csomópont szeretne hozzáférni egy blokkhoz, akkor egy régebbi, a blokkot használó csomópont cache-ének a könyvtár küld egy érvénytelenítő üzenetet.

Cache-hez kapcsolódó könyvtárak használata megoldás

Nem nehéz belátni, hogy egyszerre egy időben a memóriabeli blokkoknak csak egy egészen kis hányada szerepel a csomópontok cache-ében. Ez természetes, hiszen a cache sokkal kisebb, mint a rendszermemória. Felesleges tehát a rendszermemória minden egyes blokkjának állapotáról nyilvántartást vezetni, hiszen a könyvtár bejegyzésinek túlnyomó többségében úgyis csupa 0 bitekből áll a PFV. Sokkal célszerűbb a könyvtárat a cache memóriához csatolni (21.3.1 fejezet eleje, 2. lehetőség), és csak a cache-ben ténylegesen szereplő blokkok állapotát és megosztottságát nyomon követni.

Ezt az elvet alkalmazza az SCI (Scalable Coherent Interface) is. Az SCI-ben a cache blokkokat megosztó csomópontok egy (kétszeresen) láncolt listába vannak fűzve. A tároló csomópont minden egyes blokkra tárolja az első csomópont azonosítóját, ahol a blokk megtalálható. Ha van további birtokos, akkor ebben a csomópontban is található (a blokk tartalma mellett) egy mutató a következő csomópontra, és így tovább. A memóriaigény alacsony: blokkonként egy mutató kell a tároló csomópontba, valamint minden cache minden blokkja mellé kell egy mutató, amely a következő megosztó csomópontra mutat (illetve kétszeres láncolás esetén kettő). Új megosztás esetén az új csomópont a lánc elejére kerül, így ez a művelet gyors. Az érvénytelenítő üzeneteket pedig a megosztó csomópontok adogatják egymásnak a lánc mentén.

A megoldásnak két hátránya van. Egyrészt a megvalósítás bonyolultabb, mint a PFV alapú eljárásé, másrészt nagyobb az átlagos késleltetése, mivel a több csomópontot érintő üzeneteket csak sorban egymás után, a lánc követésével lehet eljuttatni az érintetteknek (PFV esetén, megfelelő összeköttetés-hálózattal egyszerre is lehetett, lásd 21.14. b.) ábra).

21.3.3. Hierarchikus koherencia-protokollok

Nagyobb rendszerekben a csomópontok összekötöttsége nem feltétlenül olyan egyszerű, mint ahogy azt a 20.5. ábra mutatja. A csomópontokat gyakran csoportokba (clusterekbe) szervezik. Minden csoport tartalmaz egy összeköttetés-hálózatot a csoport csomópontjainak (intra-cluster) kommunikációjához, de rendelkezésre áll egy csoportokat összekötő (inter-cluster) összeköttetés-hálózat is, az eltérő csoportba tartozó csomópontok kommunikációjához.

A gyakorlatban ez egy népszerű megoldás, elég csak a többmagos processzorok térnyerésére gondolni. A többmagos processzorok önmagukban is multiprocesszoros rendszerek, saját összeköttetésekkel és koherencia-protokollal. Ezeket újabb összeköttetés-hálózattal összekapcsolva még nagyobb rendszereket lehet építeni.

A csoportokon belüli, és a csoportokat összekötő összeköttetés-hálózat eltérő koherencia protokollt is használhat, mindkettő lehet üzenetszórásra alapozott, vagy könyvtárat használó is.

Üzenetszórásra alapozott csoporton belüli és csoportközi koherencia protokoll

Ebben az esetben mindkét szinten üzenetszórásra alapozott protokoll biztosítja a koherenciát. A hierarchikus felépítés számos esetben a memóriaműveletek kisebb késleltetéshez, és busz sávszélesség megtakarításhoz vezet. Például egy blokk beszerzéséért indított ”BusRd” tranzakció először a csoporton belül kerül meghirdetésre. Ha van olyan csomópont, amelyik rendelkezik a blokkal, akkor válaszol, és a tranzakció sikeresen lezárul anélkül, hogy az egész rendszert terhelte volna, a tranzakció tehát csak a csoporton belül okozott terhelést. Ha a csoportból senki nem rendelkezik a blokkal, akkor a ”BusRd” a csoportokat összekötő buszra kerül, és a rendszer összes csomópontjához eljut.

Üzenetszórásra alapozott csoporton belüli és könyvtáral használó csoportközi koherencia protokoll

Ilyen esetben a könyvtár alapú koherenciaprotokoll a csoportokat tekinti csomópontnak, tárolja, hogy a memória egyes blokkjai mely csoportokban van megosztva. A működés alapvetően a könyvtár alapú algoritmust követi. Egy ”BusRd” üzenet először a tároló csomópontba jut el, mely, ha nem tud közvetlenül válaszolni, egy olyan csoportba továbbítja azt, ahol a nyilvántartása szerint a blokk megtalálható. Ha egy csoport kap egy koherenciaprotokoll-üzenetet, akkor azt a belső, üzenetszórásos buszra teszi, a csomópontok pedig a 21.2. fejezetben megismert módon reagálnak.

Könyvtárat használó csoporton belüli és csoportközi koherencia protokoll

Ez a megoldás többmagos processzorokból álló multiprocesszoros rendszerekben igen népszerű. Minden csomópont (processzormag) rendelkezik saját cache memóriával, és a csoport (többmagos processzor) is rendelkezik egy nagyobb, a csoporton belül osztott cache memóriával. Egy könyvtár alapú protokoll biztosítja a csoporton belüli csomópontok közötti cache koherenciát, és egy másik könyvtár alapú protokoll a csoport szintű osztott memória koherenciáját.

21.4. Memória konzisztencia-modellek

21.5. Osztott memóriakezelés a gyakorlatban

21.5.1. Néhány architektúra gyakorlati megoldásai

21.5.2. Osztott memóriával rendelkező multiprocesszoros rendszerek programozása

21.5.3. Akaratlan blokkmegosztás, avagy teljesítménycsökkenés a koherencia protokoll működéséből fakadóan

Irodalomjegyzék

[1] AMD Athlon XP Processor Model 10 Data Sheet. http://support.amd.com/us/Processor_TechDocs/26237.PDF. 2012.09.30.

[2] Hitachi Travelstar 5K160 Hard Disk Drive Specification . http://www.hgst.com/tech/techlib.nsf/techdocs/8037f4603fe5b763862571840068043c/$file/5k160_sata_spv1.2.pdf. 2014.08.29.

[3] Intel itanium architecture software developer's manual vol. 3 rev. 2.2: Instruction set reference. Intel. 2006.

[4] Intel Q45 and Q43 Express Chipsets Product Brief. http://www.intel.com/Assets/PDF/prodbrief/320665.pdf. 2012.09.30.

[5] NAND vs NOR flash memory technology overview. Toshiba. 2006.

[6] USB Specification. http://http://www.usb.org/developers/docs/. 2014.08.29.

[7] Don Anderson. *Universal Serial Bus System Architecture.* Addison-Wesley, 2001.

[8] Don Anderson, Tom Shanley, and Ravi Budruk. *PCI express system architecture.* Addison-Wesley, 2004.

[9] Krste Asanovic. Lecture Notes for CS 252 Graduate Computer Architecture. http://inst.eecs.berkeley.edu/~cs252/fa07/lectures/L06-VLIW.pdf. 2007.

[10] Krste Asanovic. Lecture Notes for CS 252 Graduate Computer Architecture. http://inst.eecs.berkeley.edu/~cs152/sp11/lectures/L15-Vector.pdf. 2011.

[11] Jan Axelson. *USB complete: the developer's guide.* Lakeview Research, 2009.

[12] Jean-Loup Baer. *Microprocessor architecture: from simple pipelines to chip multiprocessors.* Cambridge University Press, 2009.

[13] Randal Bryant and O'Hallaron David Richard. *Computer systems: a programmer's perspective.* Prentice Hall, 2003.

[14] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. *Computers, IEEE Transactions on*, 44(5):609–623, 1995.

[15] Compaq Computer Corporation. *Alpha Architecture Handbook.* Compaq, 1998.

[16] David E Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel computer architecture: a hardware/software approach.* Gulf Professional Publishing, 1999.

[17] Peter Desnoyers. Analytic modeling of SSD write performance. In *Proceedings of the 5th Annual International Systems and Storage Conference*, page 12. ACM, 2012.

[18] Michel Dubois, Murali Annavaram, and Per Stenström. *Parallel computer organization and design.* Cambridge University Press, 2012.

[19] Agner Fog. The microarchitecture of intel, amd and via cpus. http://www.agner.org/optimize/microarchitecture.pdf, 2014.

[20] Andrei Frumusanu and Ryan Smith. ARM A53/A57/T760 investigated. http://www.anandtech.com/show/8718/the-samsung-galaxy-note-4-exynos-review/3. 2015.02.10.

[21] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[22] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 10:1–10:9, New York, NY, USA, 2009. ACM.

[23] Lee Hutchinson. Solid-state revolution: in-depth on how SSDs really work. http://arstechnica.com/information-technology/2012/06/inside-the-ssd-revolution-how-solid-state-disks-really-work. 2012.06.04.

[24] IBM. *PowerPC Microprocessor Family: The Programming Environments Manual for 64-bit Microprocessors*. IBM, 2005.

[25] Bruce Jacob and Trevor Mudge. Virtual memory in contemporary microprocessors. *Micro, IEEE*, 18(4):60–75, 1998.

[26] Bruce Jacob, Spencer Ng, and David Wang. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[27] Gerry Kane. *PA-RiSC 2.0 Architecture*. Prentice Hall PTR, 1996.

[28] David Kanter. Intel's Haswell CPU Microarchitecture. http://www.realworldtech.com/haswell-cpu/. 2012.11.13.

[29] Sun Yuan Kung. Vlsi array processors. *Englewood Cliffs, NJ, Prentice Hall, 1988, 685 p. Research supported by the Semiconductor Research Corp., SDIO, NSF, and US Navy.*, 1, 1988.

[30] Alvin Lebeck. Lecture Notes for Compsci 220 / ECE 252 Advanced Computer Architecture I. http://www.cs.duke.edu/courses/fall06/cps220/. 2006.

[31] Gábor Németh and László Horváth. *Számítógép Architektúrák*. Akadémiai kiadó, 1993.

[32] David A Patterson and John L Hennessy. *Computer organization and design: the hardware/software interface*. Newnes, 2013.

[33] Thomas Rauber and Gudula Rünger. *Parallel programming: For multicore and cluster systems*. Springer Science & Business Media, 2013.

[34] Nakazato Satoshi, Tagaya Satoru, Nakagomi Norihito, Watai Takayuki, and Sawamura Akihiro. Hardware technology of the sx-9 (1) – main system. *NEC Technical Journal*, 3(4):15–18, 2008.

[35] Freescale semiconductor. *AltiVec Technology Programming Interface Manual*. Freescale, 1999.

[36] Tom Shanley and Don Anderson. *PCI system architecture*. Addison-Wesley, 1995.

[37] Anand Lal Shimpi. Apple's Cyclone Microarchitecture Detailed. http://www.anandtech.com/show/7910. 2014.03.31.

[38] Anand Lal Shimpi. The SSD Anthology: Understanding SSDs and New Drives from OCZ. http://www.anandtech.com/show/2738. 2009.03.18.

[39] Mark Smotherman. Understanding epic architectures and implementations. In *40th Annual ACM Southeast Conference*, pages 71–78, April 2002.

[40] William Stallings. *Computer organization and architecture: designing for performance*. Pearson, 2013.

[41] Kristian Vättö. Understanding TLC NAND. http://www.anandtech.com/show/5067/understanding-tlc-nand. 2012.02.23.

[42] David Williamson. Arm cortex-a8: A high-performance processor for low-power applications. *Unique Chips and Systems*, pages 79–106, 2007.