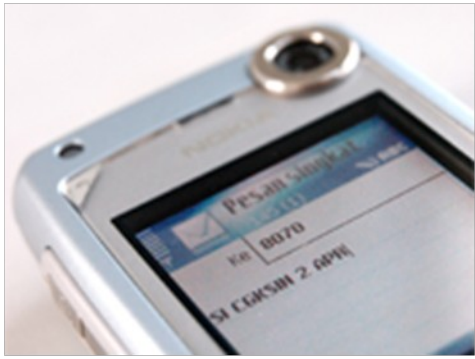


Számítógép Architektúrák

1. Információfeldolgozási modellek

Horváth Gábor
docens
BME Híradástechnikai Tanszék
ghorvath@hit.bme.hu

2012. szeptember 4.
Budapest

Adminisztratív információk

Híradástechnikai Tanszék

- Horváth Gábor – Híradástechnikai Tsz.

≠

- Horváth Gábor – Méréstechnika és Információs rendszerek Tsz.

Adminisztratív információk

- Félévközi követelmények:
  - Jelenlét gyakorlatokon
    - max. 4 hiányzás
    - Gyakorlatváltás lehetséges:
      - Szept. 14, 12 óráig
      - A két gyakvezérrel egyeztetés szükséges
  - Két nagy ZH sikeres teljesítése
    - 1. ZH: okt. 8, hétfő 8-10 (pót: okt. 24, szerda 18-20)
    - 2. ZH: nov. 12, hétfő 17-19 (pót: nov. 26, hétfő 8-10)
- + sikeres vizsga

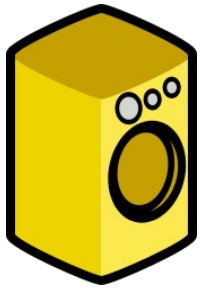
Adminisztratív információk

- Számonkérések:
  - Elméleti kérdések
  - Tesztek
  - Összekötős, kiegészítős feladatok
  - Relációanalízis
  - Számpéldák
  - Segédeszköz nem használható
  - Elégséges: 40%

Adminisztratív információk

- Segédanyagok:
  - Jegyzet: VAN! (pdf)
    - Elmélet
    - Példák
    - Irodalom
    - Kérdések & feladatok
  - Tárgy honlapja:
    - **http://www.hit.bme.hu/~ghorvath/szgarch**

Mi az a számítógép?



Mi az a számítógép architektúra?

- Computer Architecture – „Számítógép építészet”

- A megfelelő funkciójú, teljesítményű és árú számítógép felépítéséhez szükséges hardver komponensek kiválasztásának és összekötésének tudománya.

  “The science and art of selecting and interconnecting hardware components to create computers that meet functional, performance functional and cost goals.”

- A számítógépek tervezésének elmélete.

  “The theory behind the design of a computer.”

- Számítógépek elvi felépítése és alapvető működési módja.

  “The conceptual design and fundamental operational structure of a computer system.”

- A számítógépek alkatrészeinek elrendezése és kapcsolata.

  “The arrangement of computer components and their relationships.”

- *…*

Mire jó ez a tárgy?

- Most megértjük, hogy
  - ...hogy épül föl egy modern számítógép
  - ...hogy épül föl egy modern processzor
- A „profik”
  - a számítógépet nem **fekete doboz**nak tekintik
  - értik, hogy működik → hatékonyabb programot írnak
  - tisztában vannak a kompromisszumokkal (mi minek az árán lehetséges)

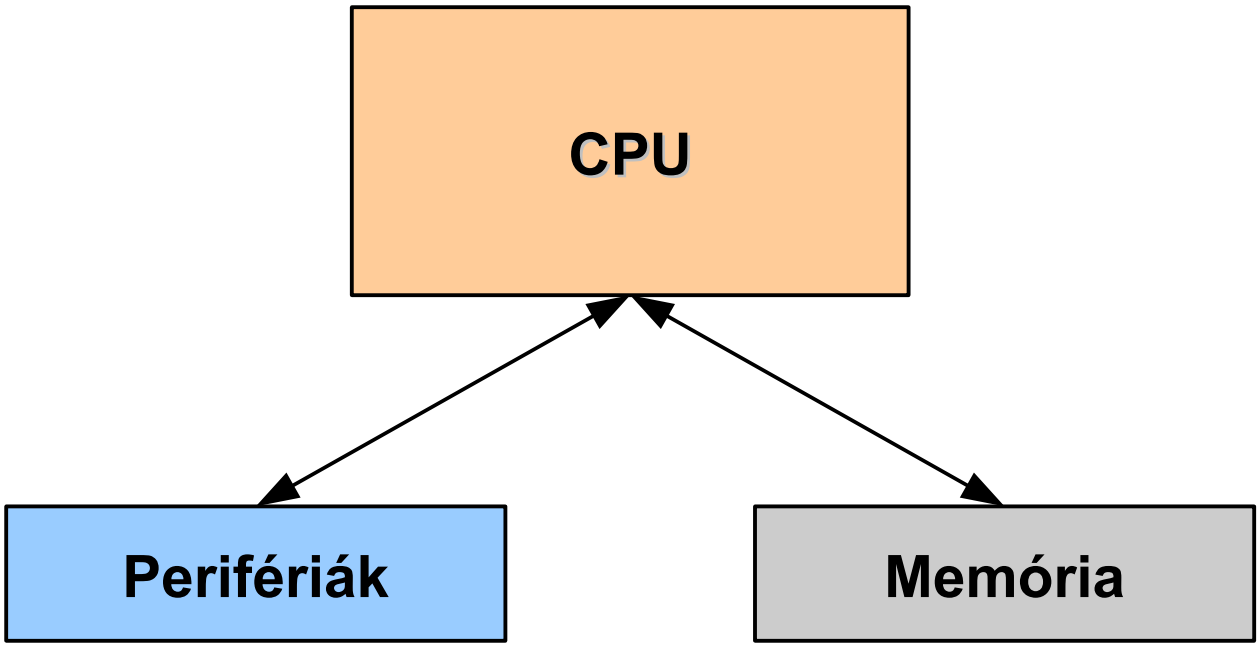
Tematika

- Információfeldolgozási modellek (1. előadás)

Tematika

- A vezérlésáramlásos modell (2. előadás)

- Neumann architektúra:



Tematika

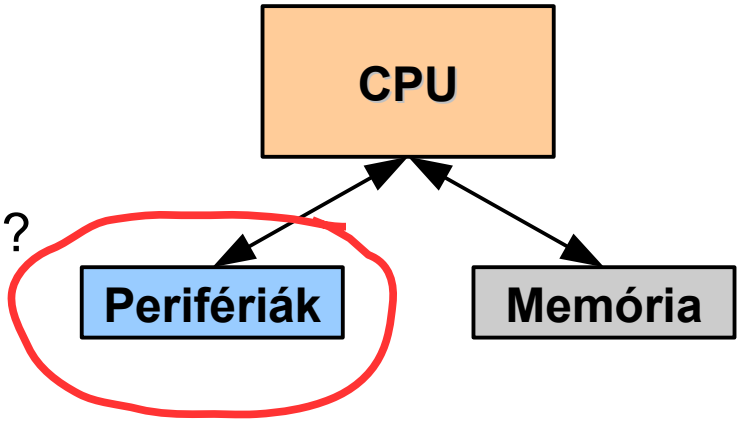
- **Perifériakezelés**
  - 3. előadás

    Hogy beszél a CPU a perifériákkal?

    És a perifériák a CPU-val?

    Hogy lehet gyorsítani?

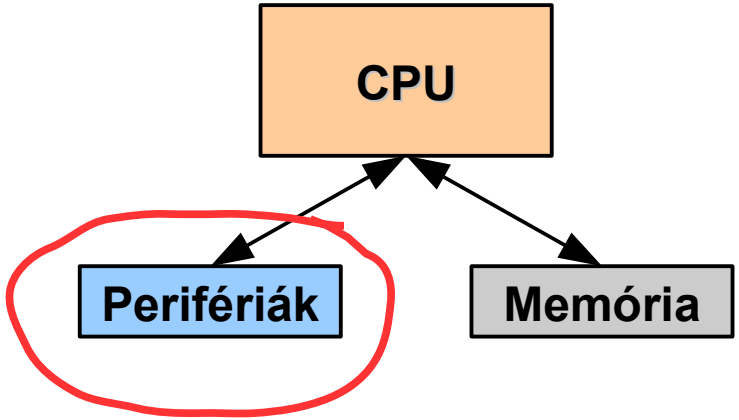
    Forgalomszabályozás, buszok, déli híd – északi híd, stb...



  - 1.-2. gyakorlat
    - Konkrét feladat: számkódos ajtónyitó – egyszerű perifériakezelés gyakorlása, táblán, papíron
  - 3. gyakorlat
    - Megcsináljuk az ajtónyitót! Arduino-val, RFID-val, 90 perc alatt, forrasztás nélkül, olcsón, C-ben írt arasznyi programmal!

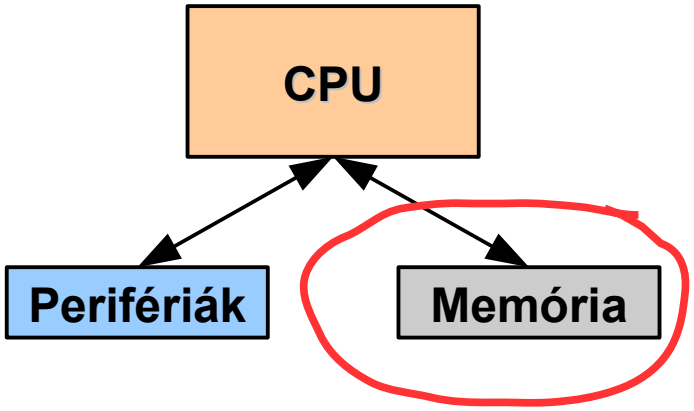
Tematika

- **Perifériakezelés**
  - Előadás:
    - A legfontosabb periféria: **a háttértár**
    - Hogy működik a HDD?
    - Hogy működik az SSD?
    - Melyik „jobb”?
    - Mi az a SATA és a SCSI?
  - Gyakorlat
    - Csatolófelületek:
      - PCI/PCI Express
      - USB
    - Számpéldák



Tematika

- **Memóriakezelés**
  - Előadás
    - Memóriatechnológiák:
      SRAM, DRAM, SDRAM, DDR, DDR2,
      DDR3, GDDR3, GDDR5, stb.



Kingston DDR3-RAM 4GB PC3-10667 HyperX Kit (KHX1333C7AD3K2/4G)

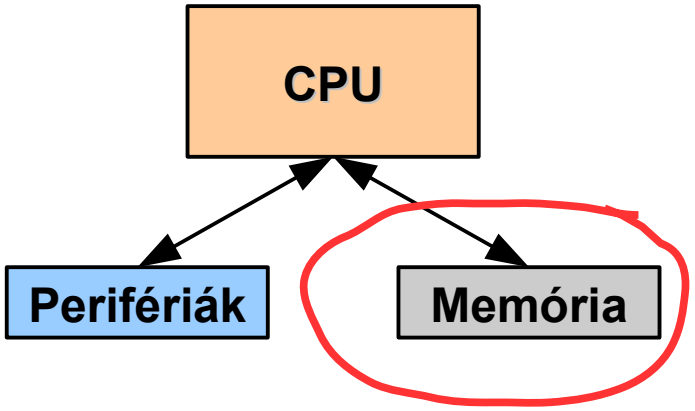
Termék-összehasonlítás

KINGSTON Memória HYPERX DDR3 4GB 1333MHz Non-ECC Low-Latency CL7 (7-7-7-20) DIMM (Kit of 2) ...

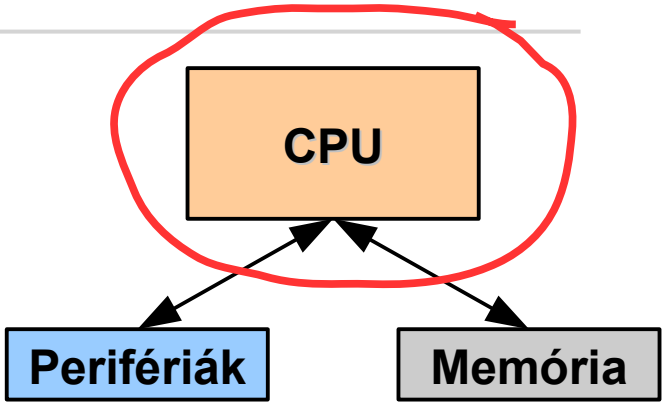
    - Hogy használjunk több memóriát, mint amennyit a processzor támogat?
    - Virtuális tárkezelés
    - Memóriavédelem (mitől kell megvédeni?)
  - Gyakorlat
    - Számpéldák

Tematika

- **Cache memória**
  - Előadás
    - Minek az?
    - Cache szervezés
    - Tartalom menedzsment
  - Gyakorlat
    - Numerikus példák



Tematika



- **CPU**
  - Előadás
    - Pipeline utasításfeldolgozás
    - Szuperskalár processzorok
      - Már egy mobilban is ilyen van!
    - Soron kívüli utasításvégrehajtás
    - VLIW/EPIC processzorok
      - Így működnek az AMD grafikus processzorai
    - Elágazásbecslés
      - Mit kell azon becsülni??
  - Gyakorlat
    - Numerikus példák:
      - egymásrahatások azonosítása
      - Utasítások manuális átrendezése, hogy gyorsabb legyen a futás
      - ciklus kifejtés, szoftver pipeline
      - ...
      - utasításütemezés

Tematika

- **SIMD feldolgozás**
  - Előadás
    - Vektorprocesszorok
    - Vektor utasításkészletek
      - SSE: PC, NEON: telefon, tablet, AltiVec: PS3, XBox
    - Tömbprocesszorok
      - Számok rendezése O(1) alatt!
  - 11. gyakorlat
    - Vektorizálás
    - SSE utasításkészlet használata

Tematika

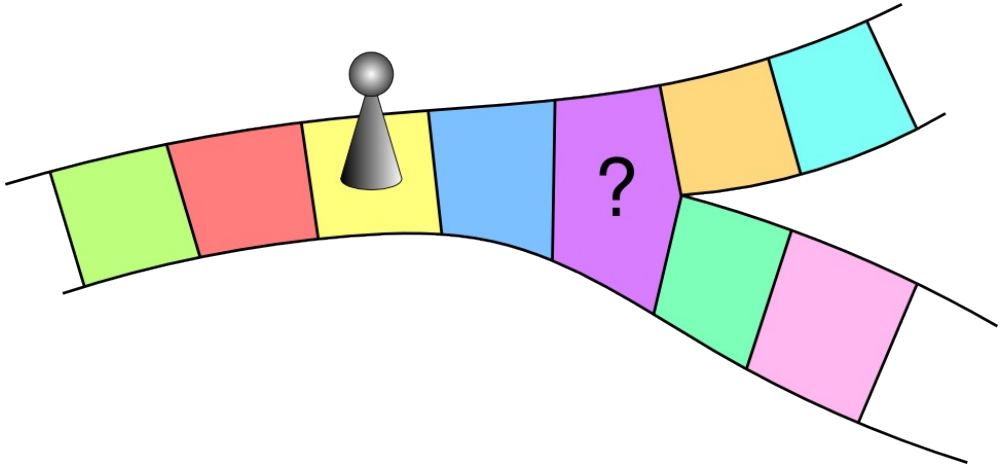
- **Mutiprocesszoros rendszerek**
  - Előadás
    - Többprocesszoros rendszerek, többmagos processzorok,
    - Többszálúságot támogató processzorok (Hyperthreading)
    - Lazán csatolt rendszerek: cluster, grid, …
    - Összeköttetések
    - Osztott memóriakezelés
  - Gyakorlat
    - Multiprocesszoros rendszerek programozása: OpenCL, OpenMP, …
- Alternatív számítógép architektúrák
  - Ha lesz rá idő
    - Kvantumszámítógépek

Információfeldolgozási modellek

- Vezérlésáramlásos
- Adatáramlásos
- Igényvezérelt

Vezérlésáramlásos modell

- Utasítások sorrendje: vezérlő token szabja meg

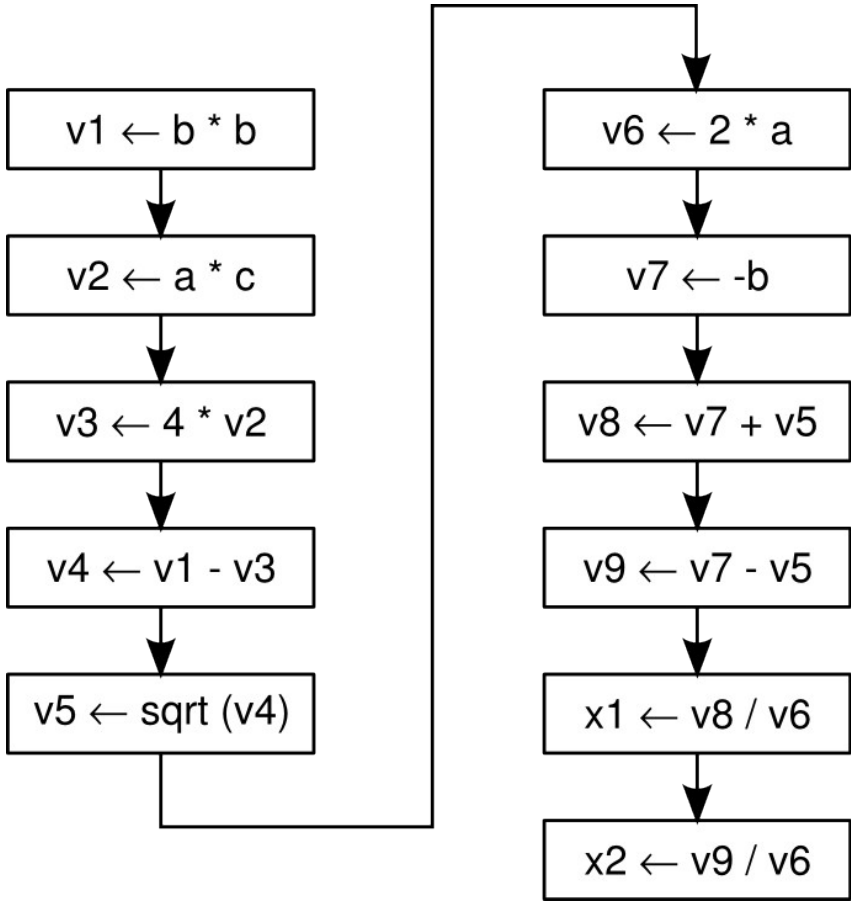


- Vezérlő token mozgása:
  - Implicit (egyet lép előre)
  - Explicit (goto, szubrutin, return, stb.)
- Utasítások közötti adatcsere: közös memória
- Program leírása: *folyamatábra*

Vezérlésáramlásos modell

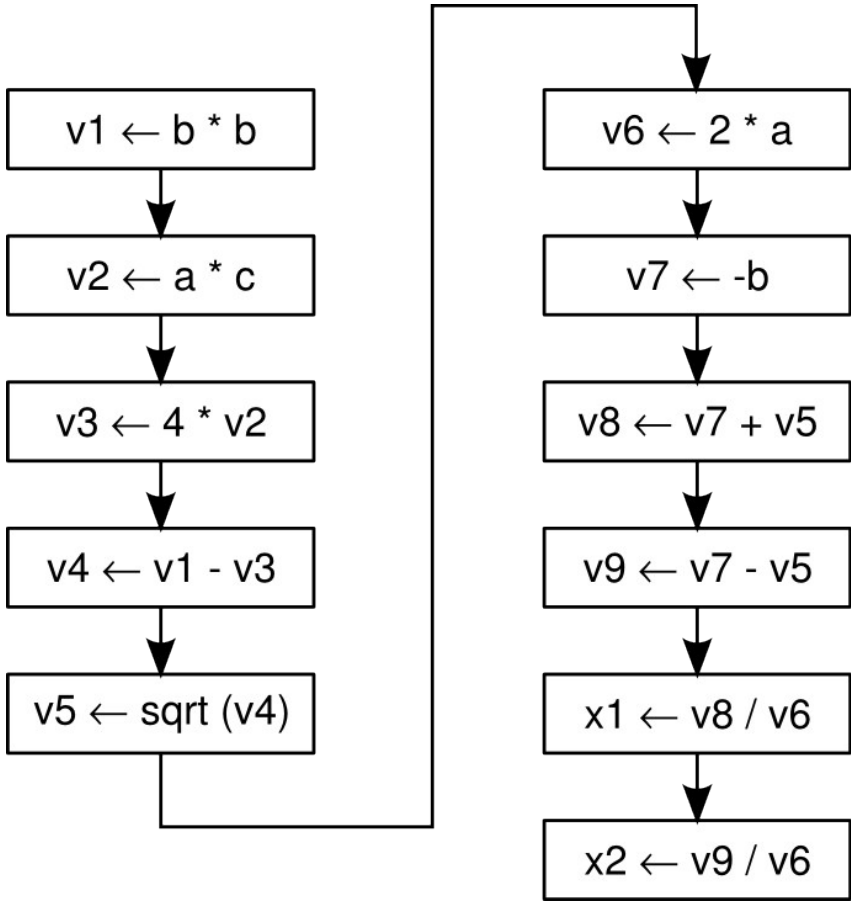
- Példa: másodfokú egyenlet gyökei

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$



Vezérlésáramlásos modell

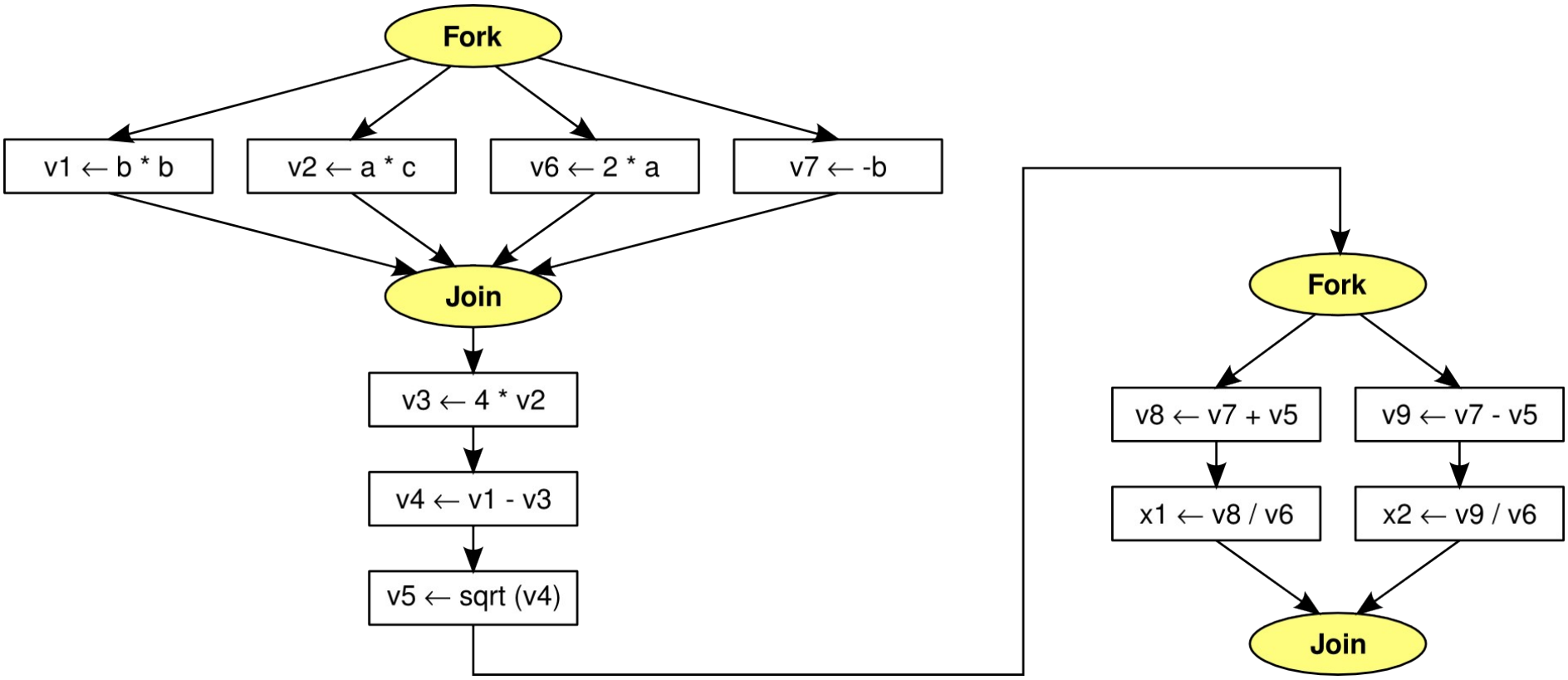
- Párhuzamosság? Hol?
  - v1, v2, v6, v7
  - v8, v9
  - x1, x2



Vezérlésáramlásos modell

- Párhuzamosság Fork/Join primitívekkel

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

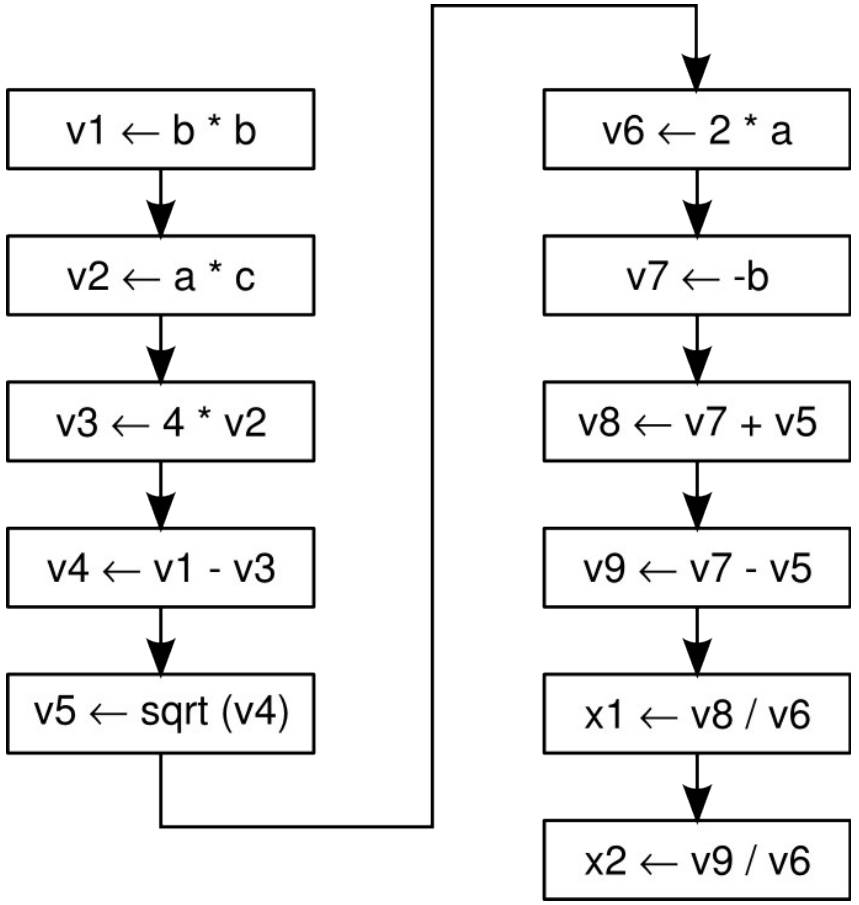


Vezérlésáramlásos modell

- Értékelés
  - Párhuzamosítási lehetőségeket
    - nem ismeri fel automatikusan,
    - nem képes kiaknázni.
  - Rendkívül elterjedt
  - Példa: Neumann architektúra
    - Vezérlő token: utasításszámláló
    - Közös memória: regiszterek + rendszermemória

Adatáramlásos modell

- Honnan tudtuk, hogy ezek párhuzamosíthatók?
  - v1, v2, v6, v7
  - v8, v9
  - x1, x2
- Függőségi analízis!

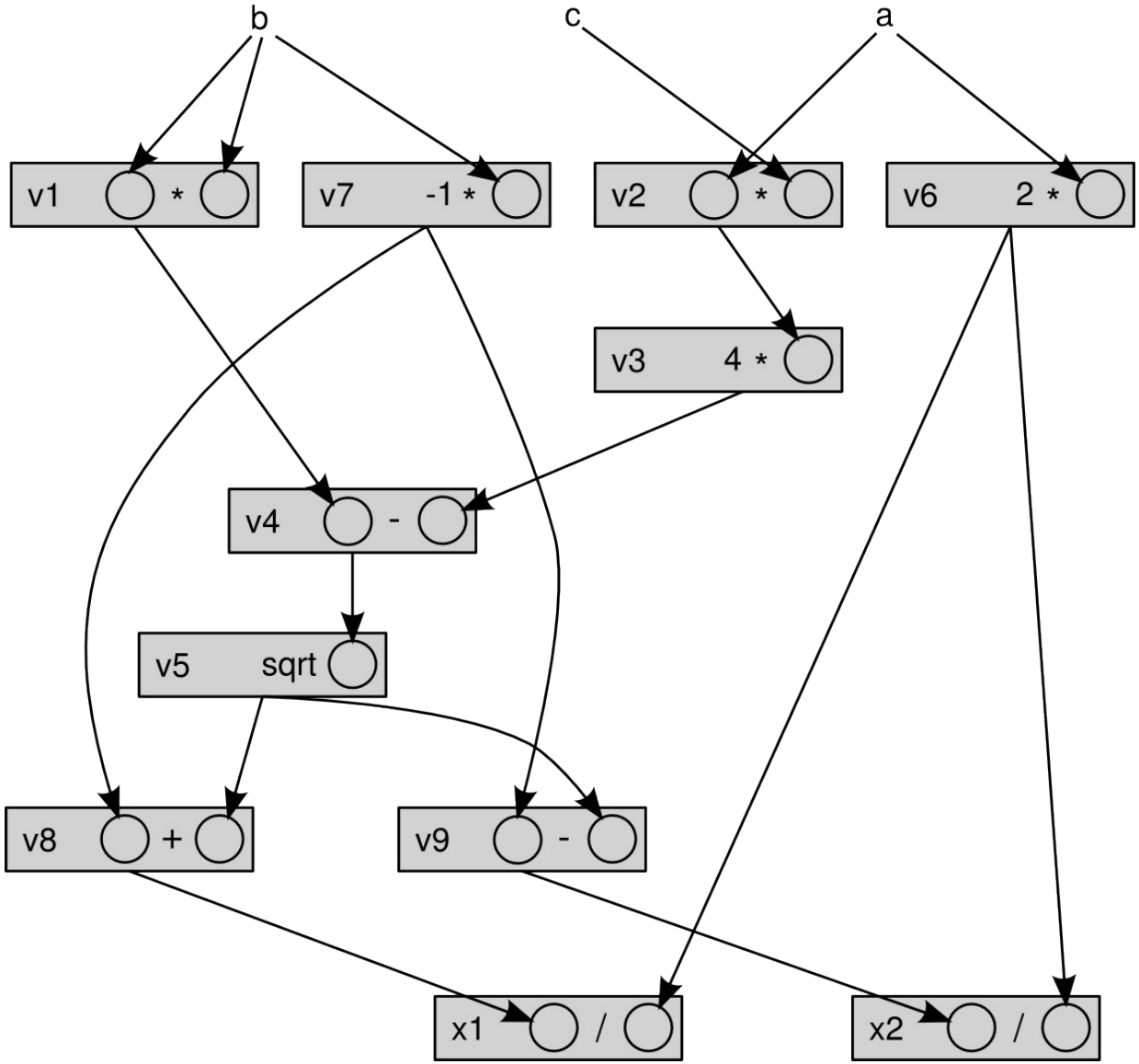


Adatáramlásos modell

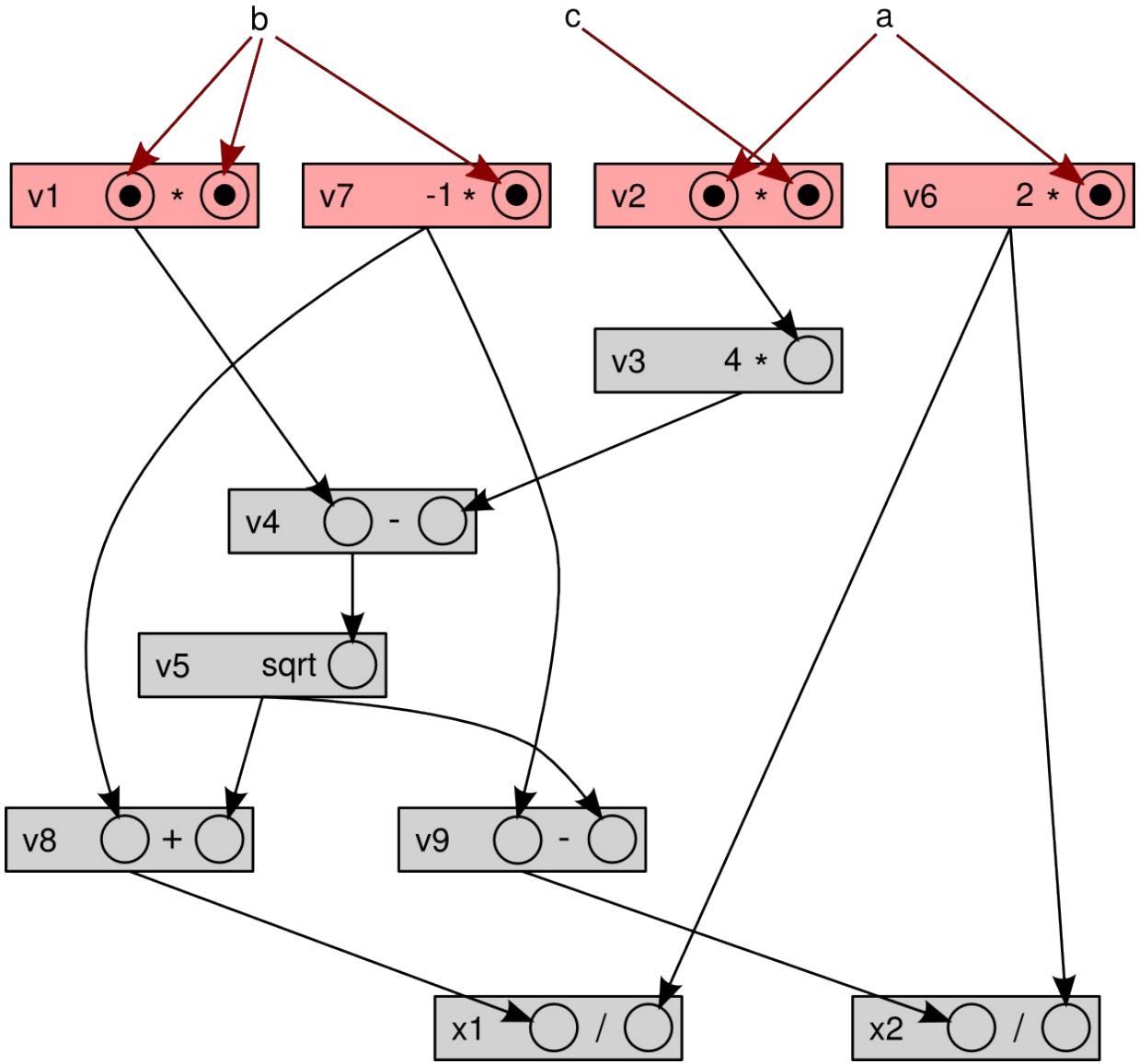
- Nincs vezérlő token
- Nincs közös memória
- Vezérlő token helyett:
  - Utasítások végrehajtása automatikus:
    - ha minden operandus megvan
- Közös memória helyett:
  - Részeredményeket közvetlenül adják át egymásnak az utasítások
- Program leírása: egy *precedenciagráf*

- Példa: másodfokú egyenlet gyökei

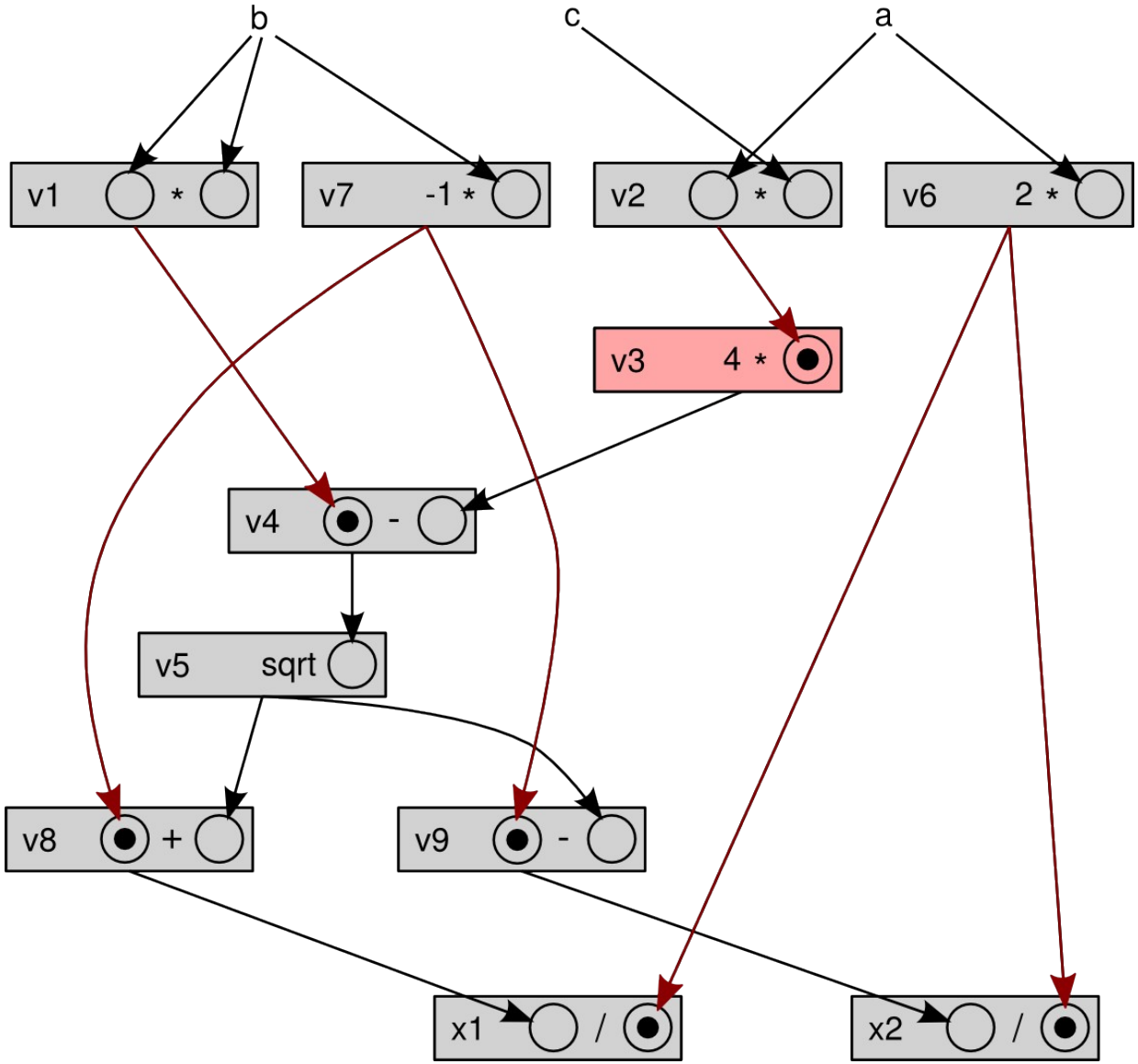
Adatáramlásos modell



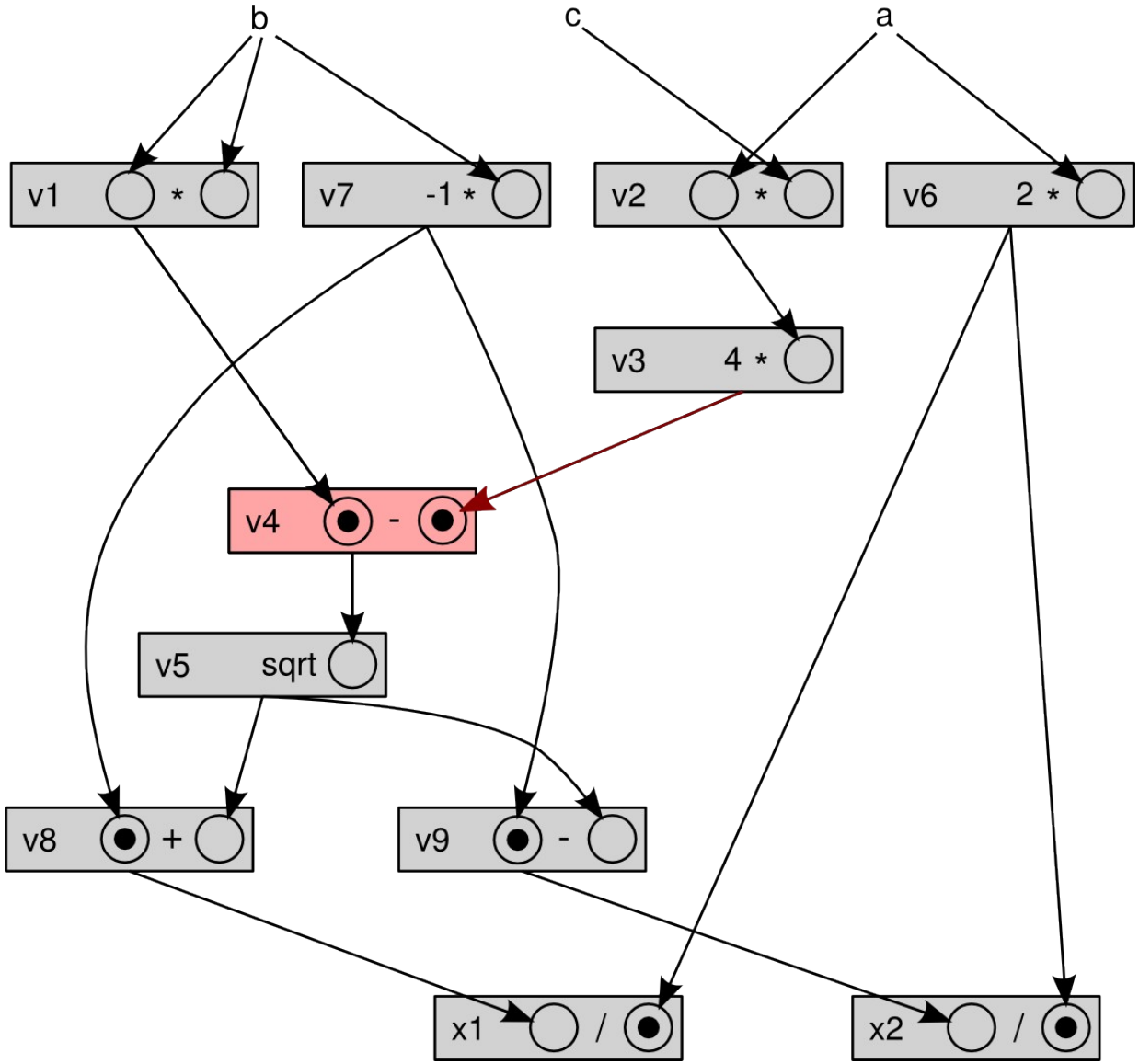
Adatáramlásos modell



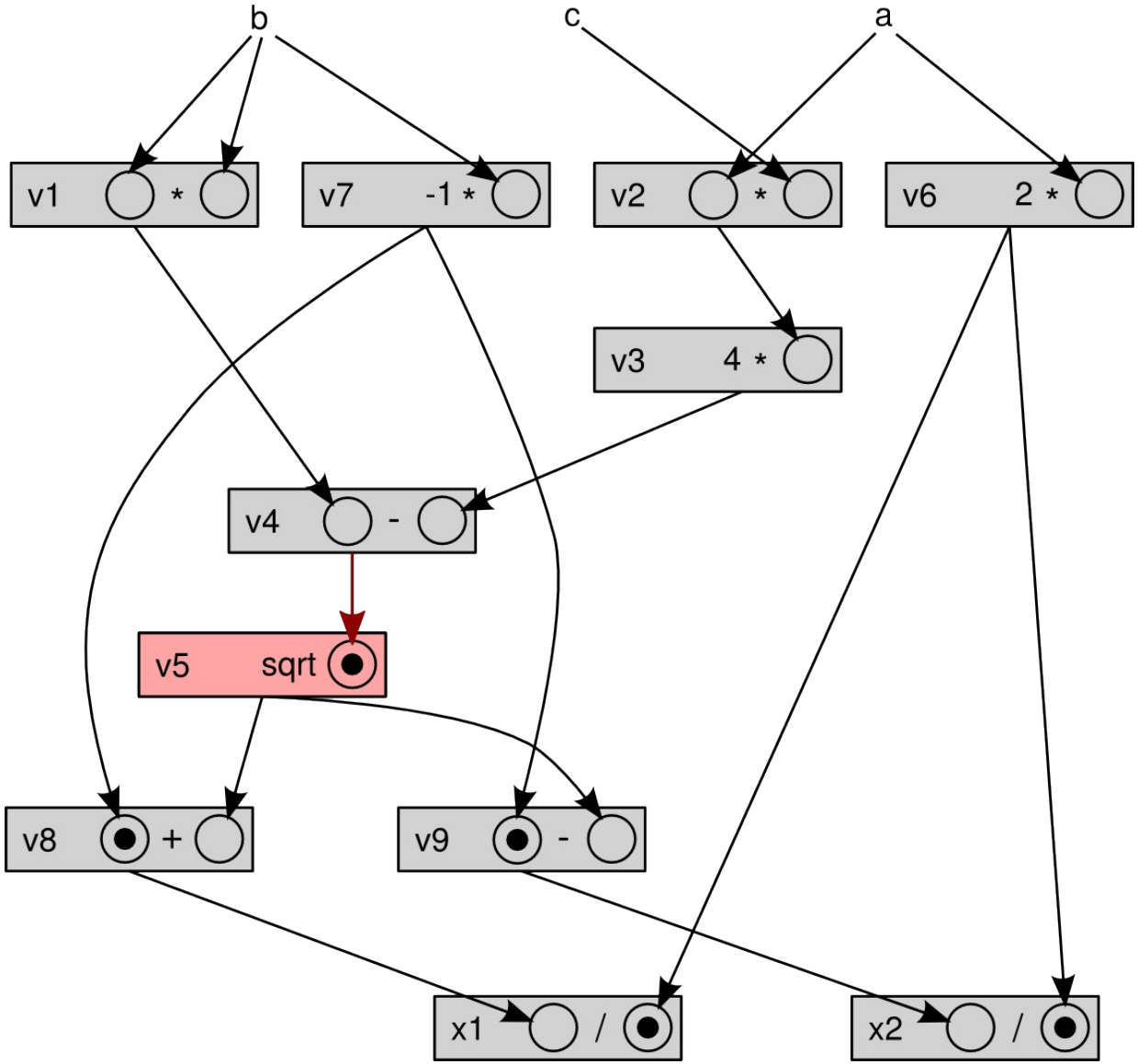
Adatáramlásos modell



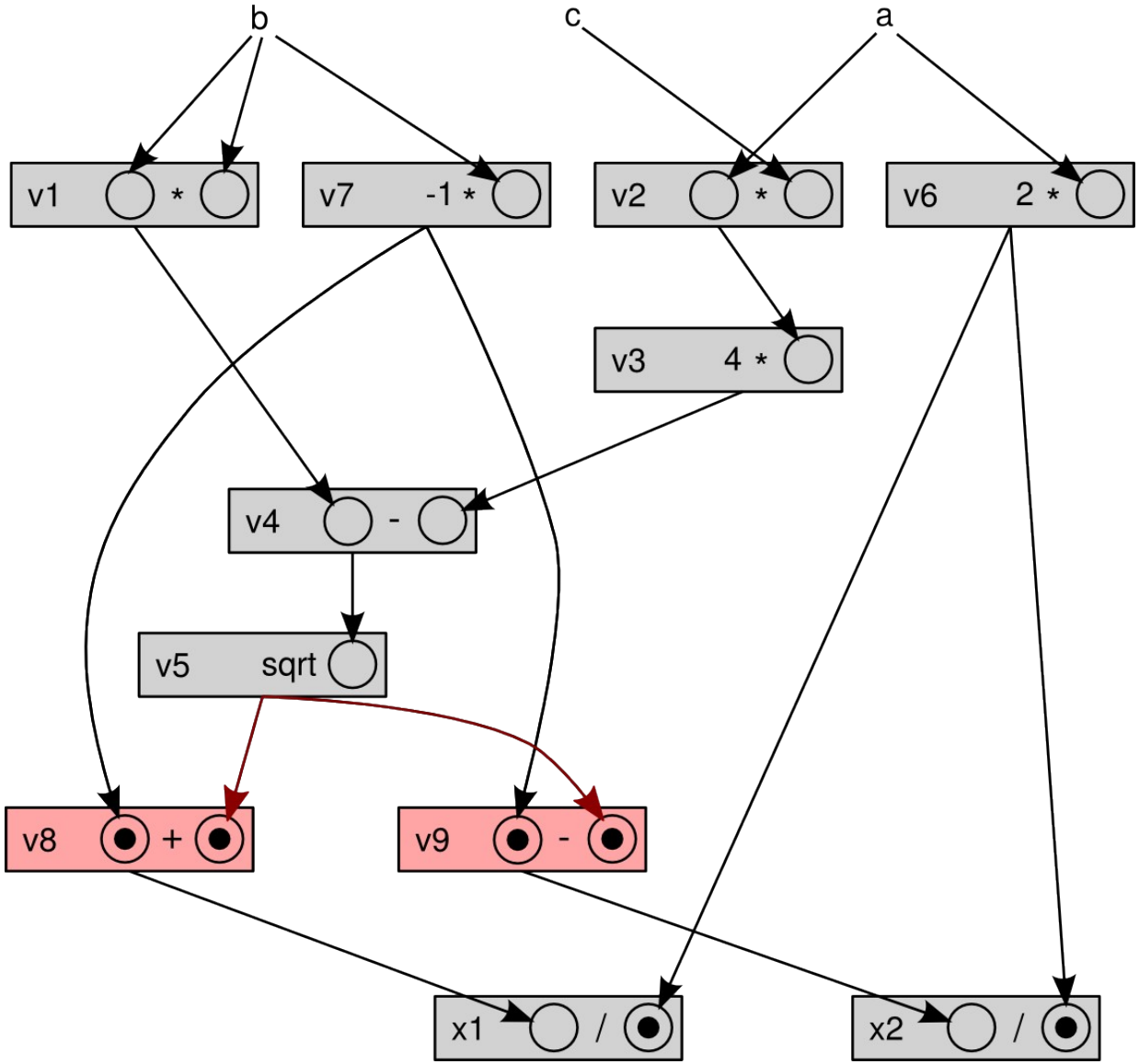
Adatáramlásos modell



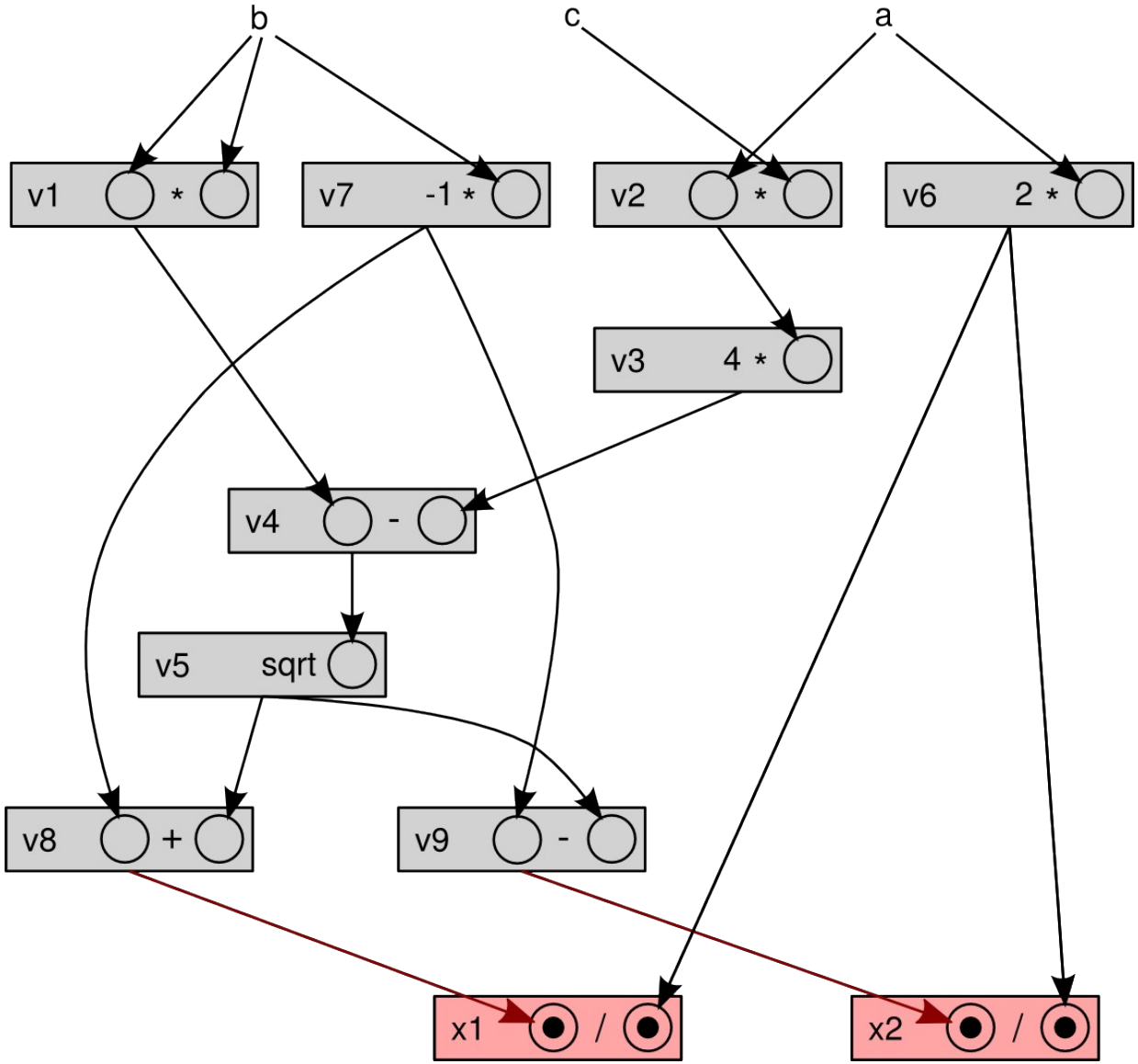
Adatáramlásos modell



Adatáramlásos modell



Adatáramlásos modell



Adatáramlásos modell

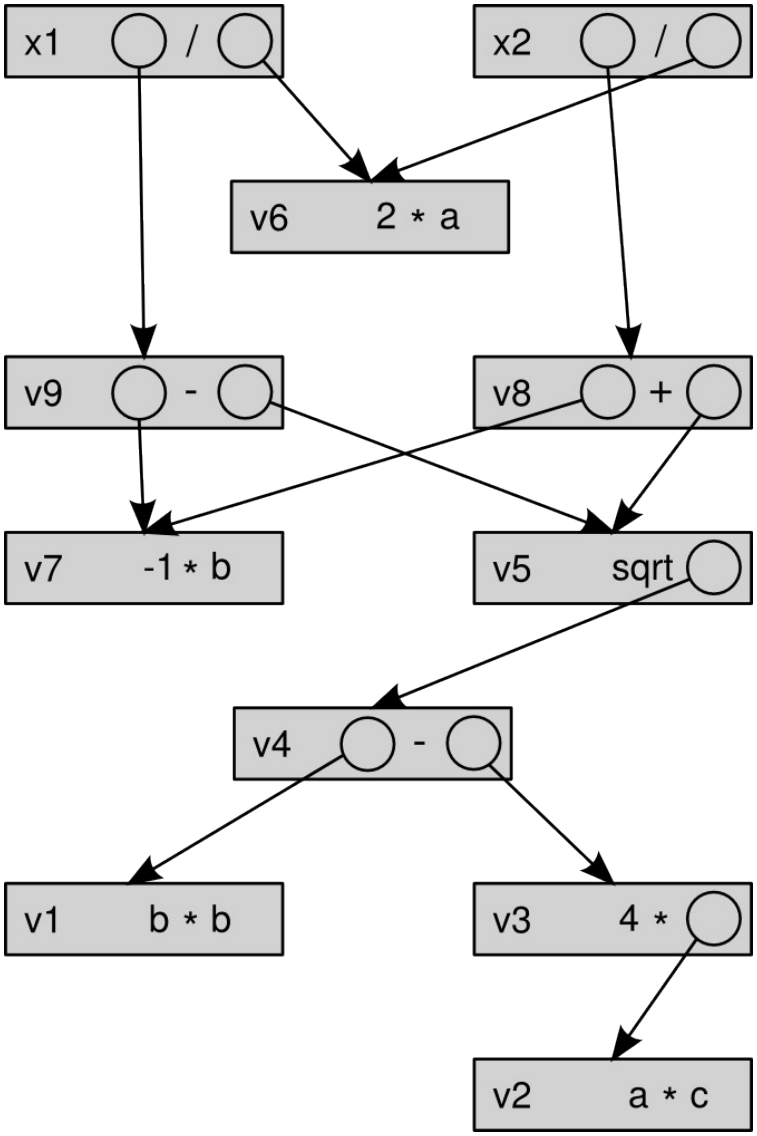
- Értékelés
  - Párhuzamosságot automatikusan felderíti
  - Tisztán adatáramlásos számítógép nem terjedt el
  - De az elv igen:
    - Táblázatkezelők
    - Out-of-order utasításvégrehajtás

Igényvezérelt modell

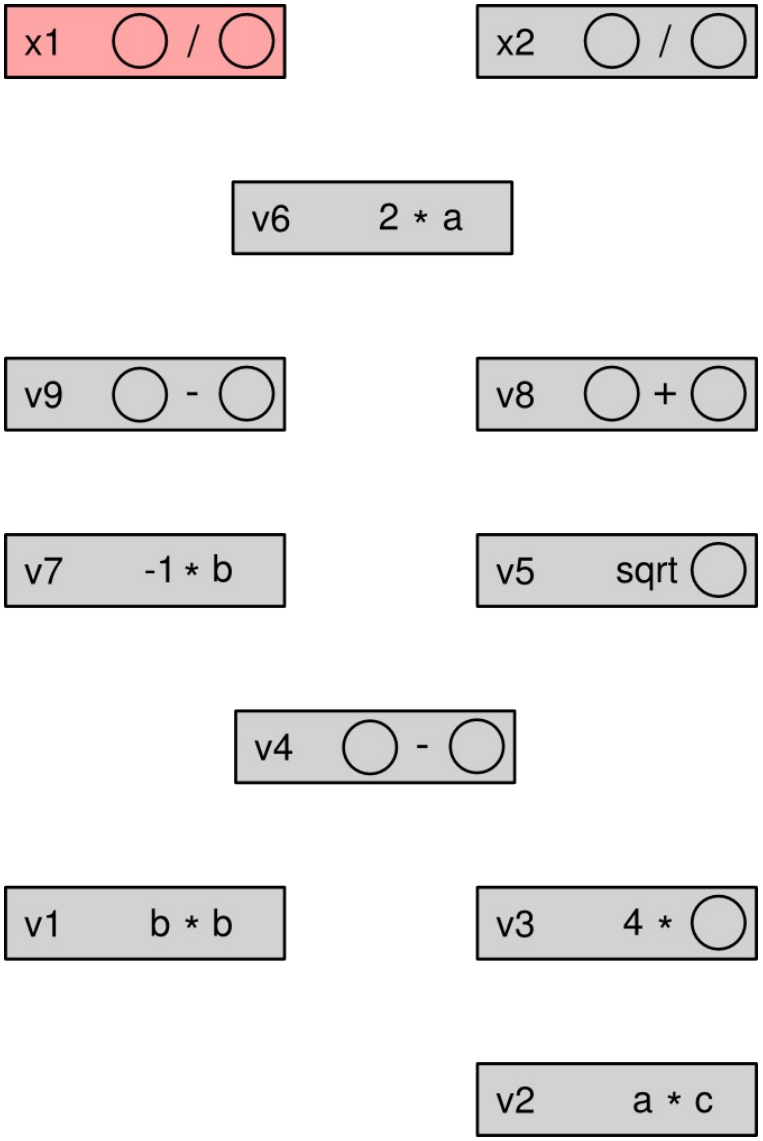
- Utasítások végrehajtása automatikus:
  - ha szükség van az eredményre
- Összehasonlítva, utasítások végrehajtási ideje:
  - Vezérlésáramlásos: ha odaér a token
  - Adatáramlásos: ha minden operandus megvan
  - Igényvezérelt: ha szükség van az eredményre

- Példa: másodfokú egyenlet gyökei

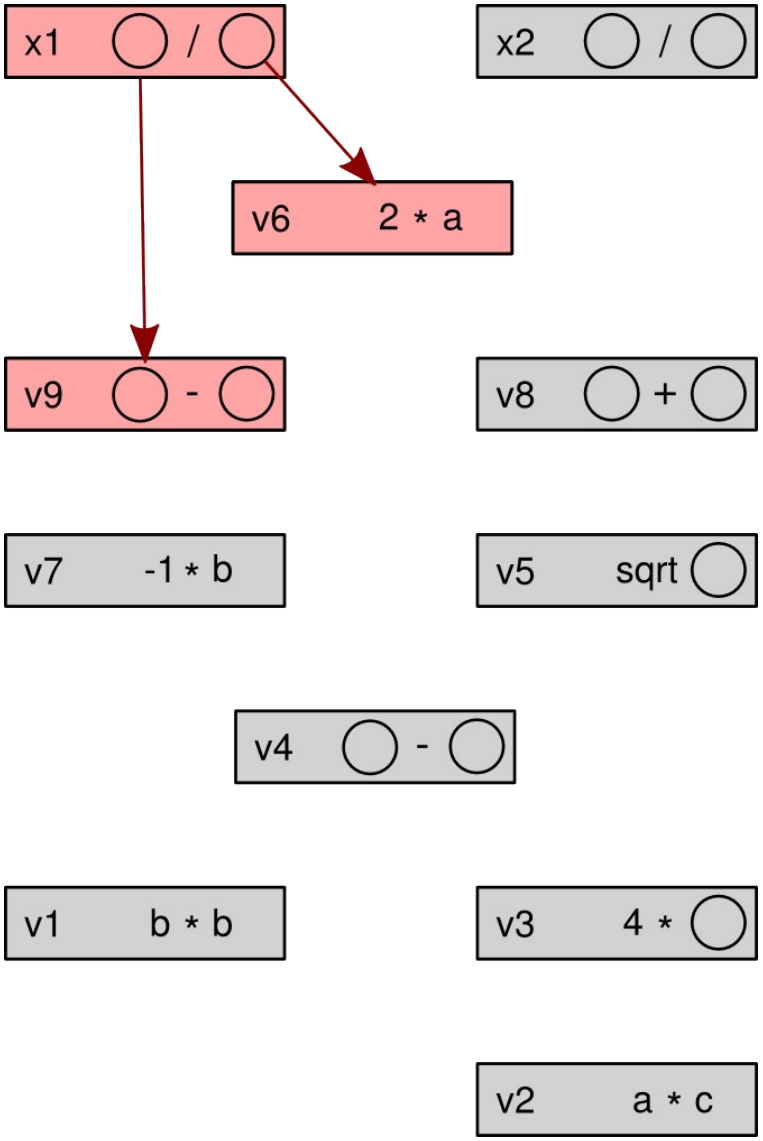
Igényvezérelt modell



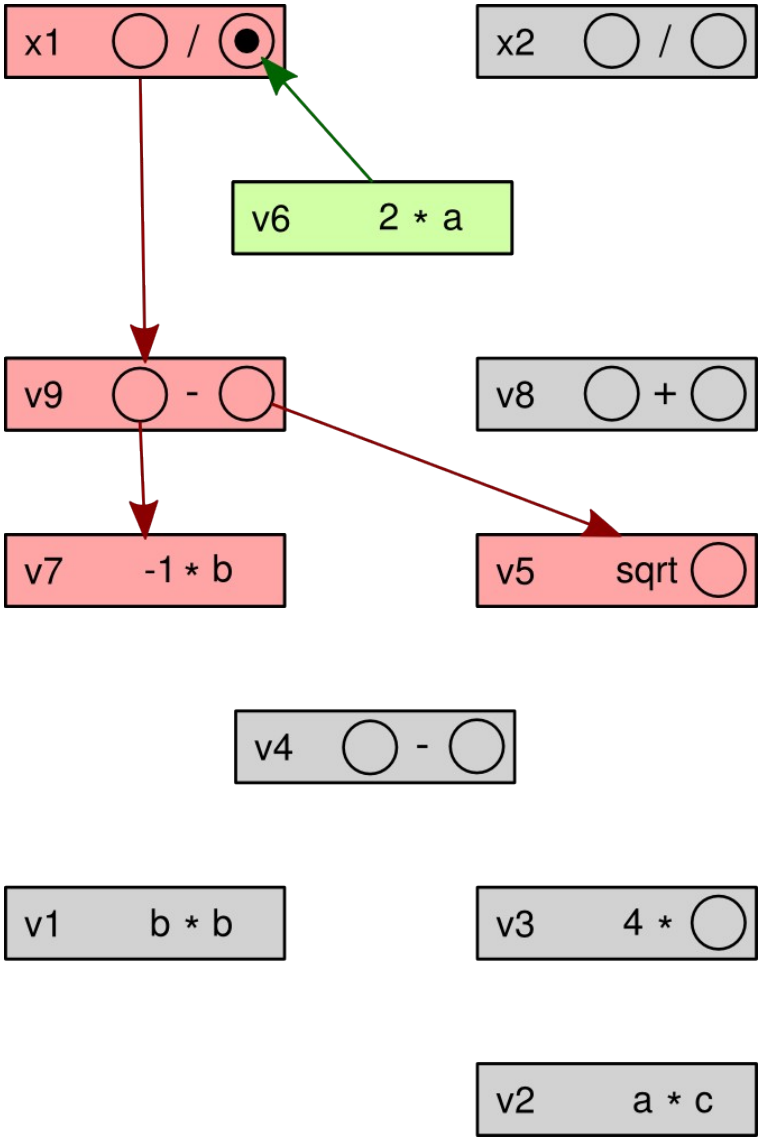
Igényvezérelt modell



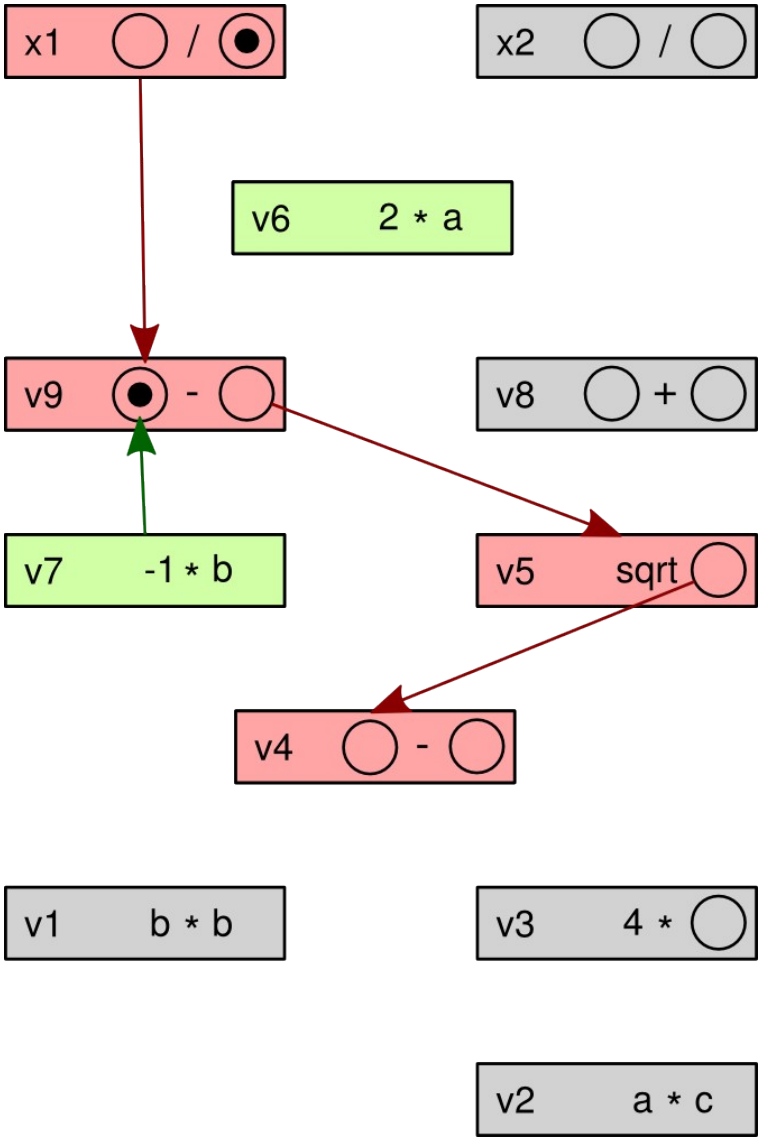
Igényvezérelt modell



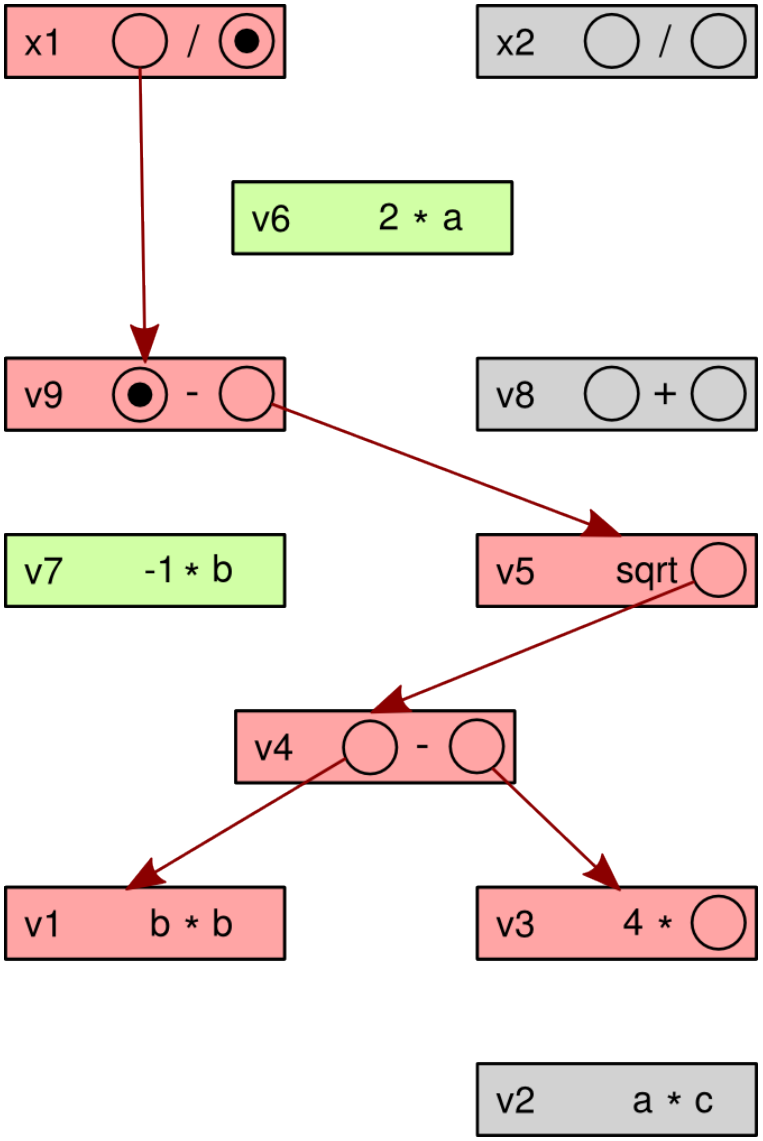
Igényvezérelt modell



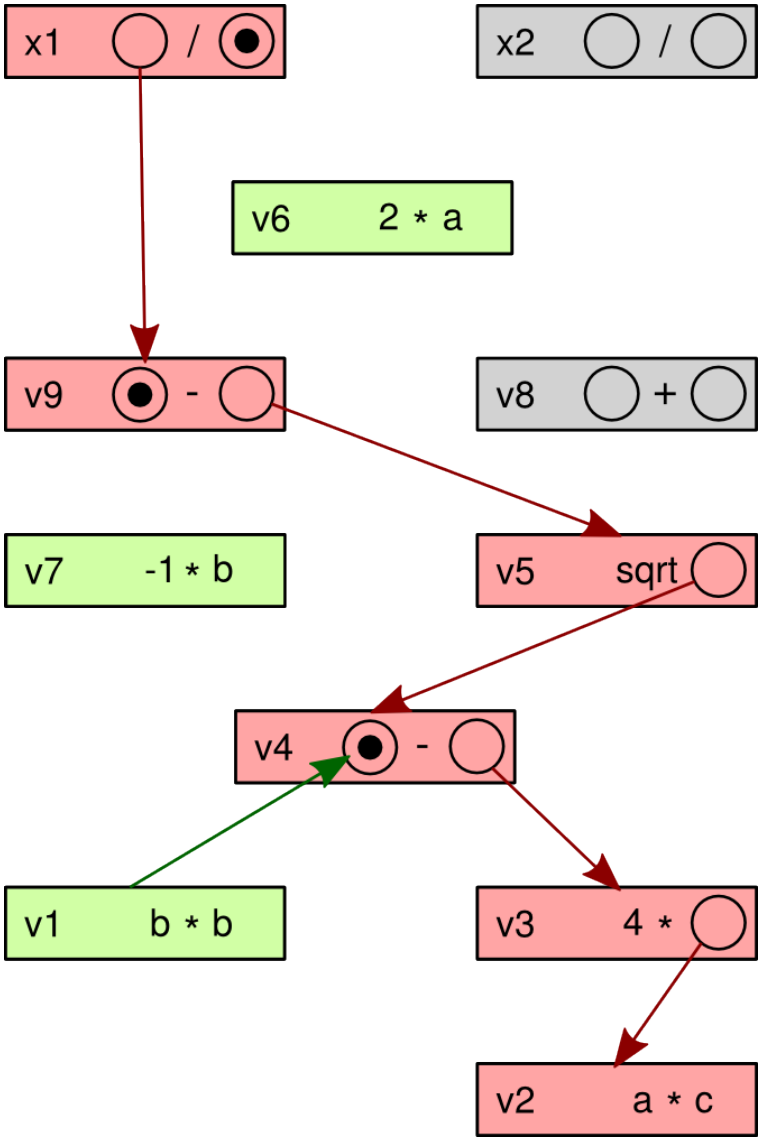
Igényvezérelt modell



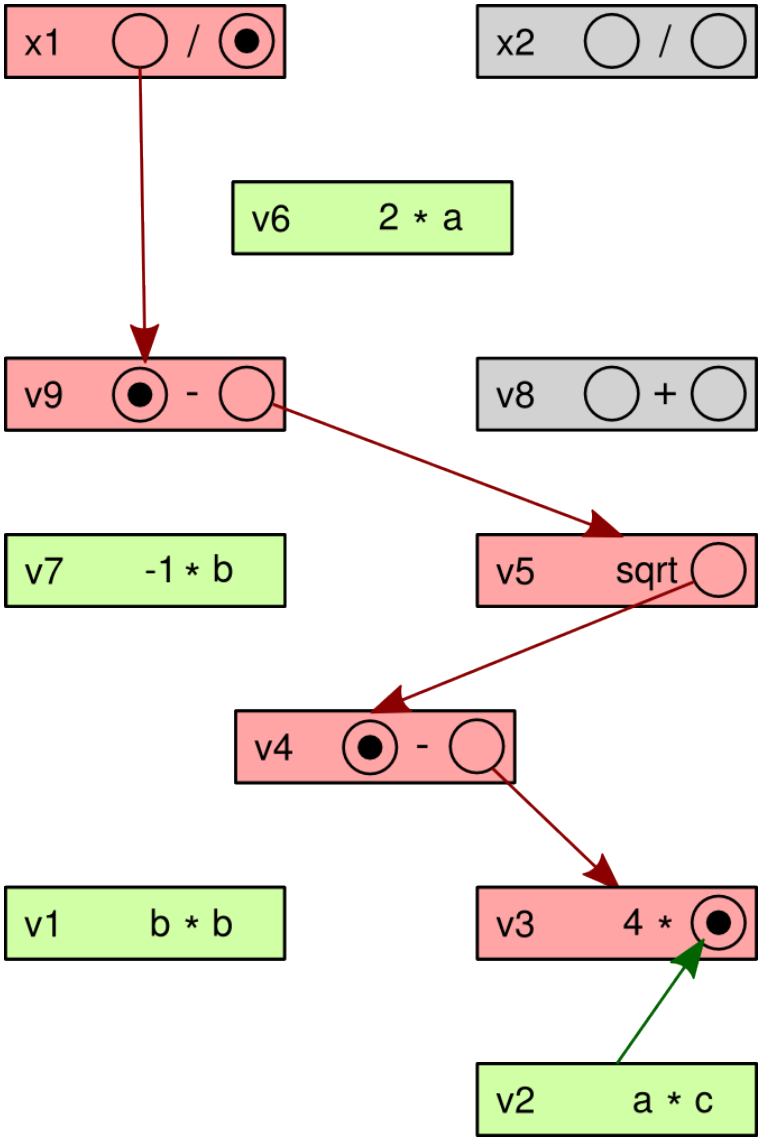
Igényvezérelt modell



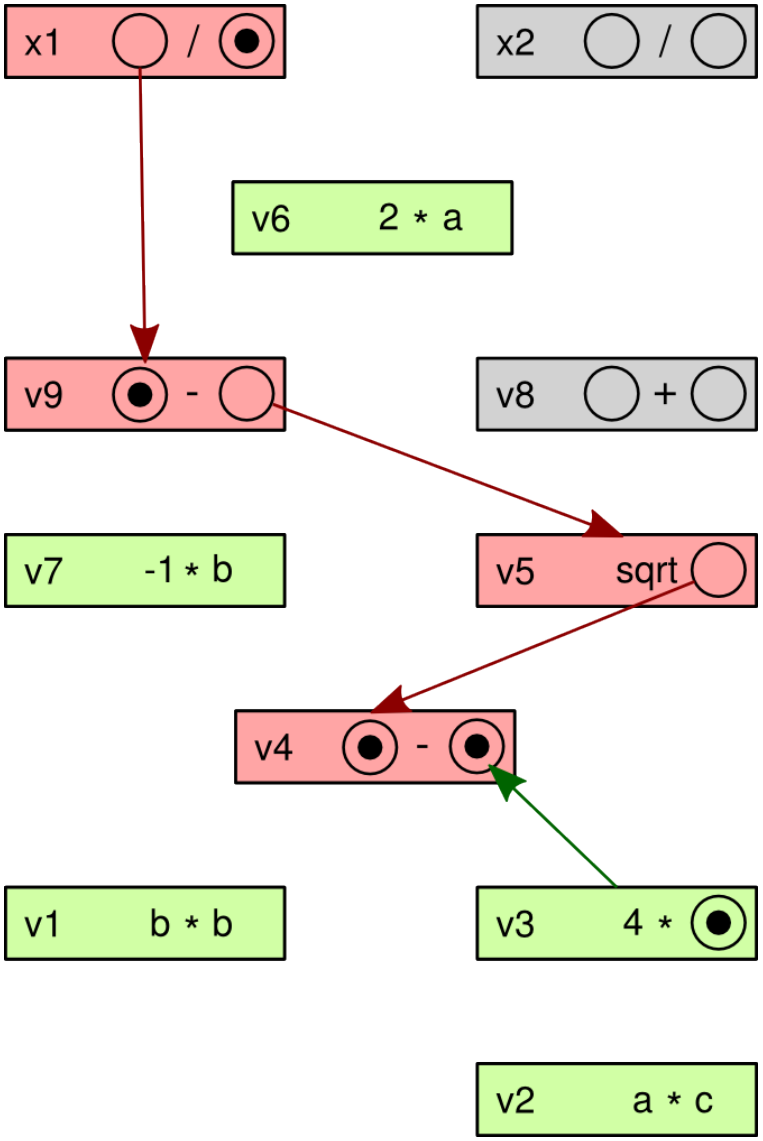
Igényvezérelt modell



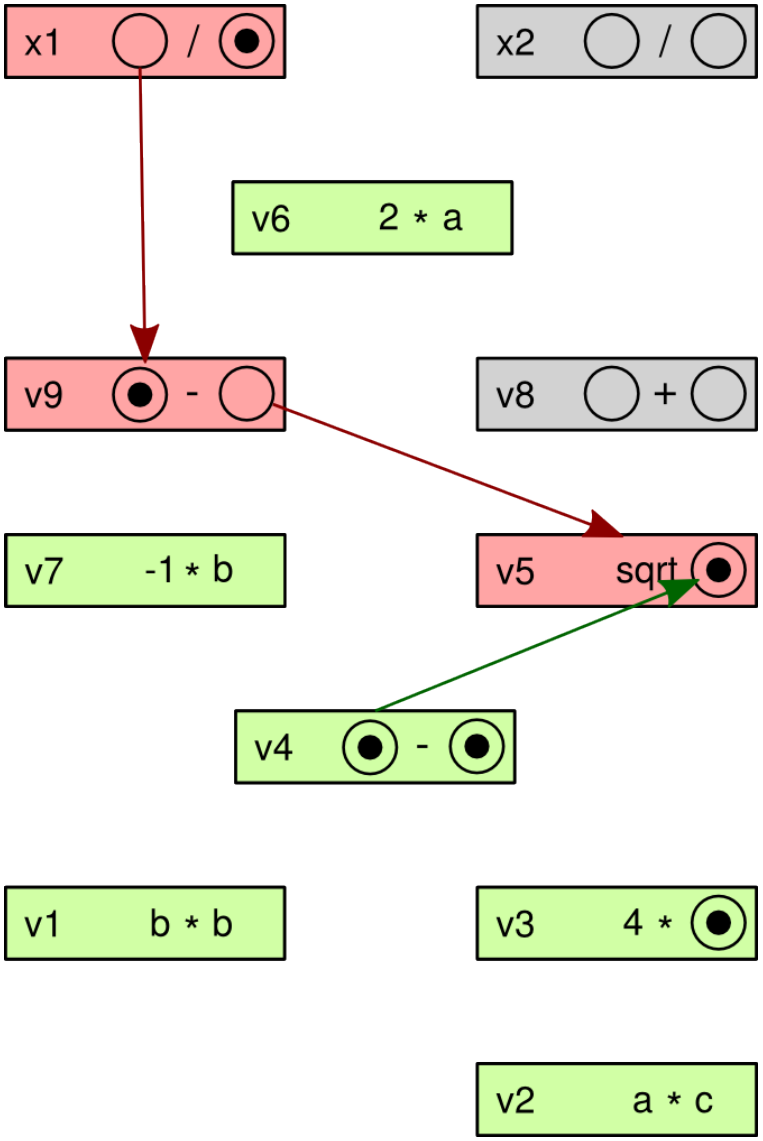
Igényvezérelt modell



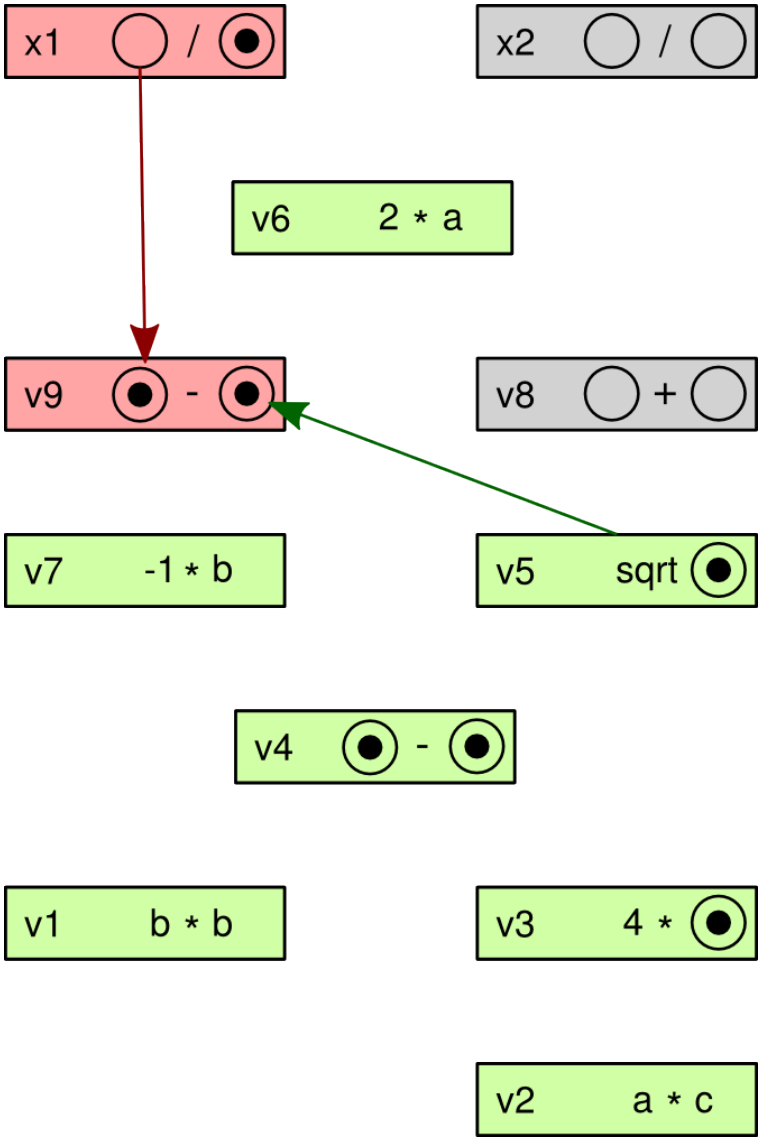
Igényvezérelt modell



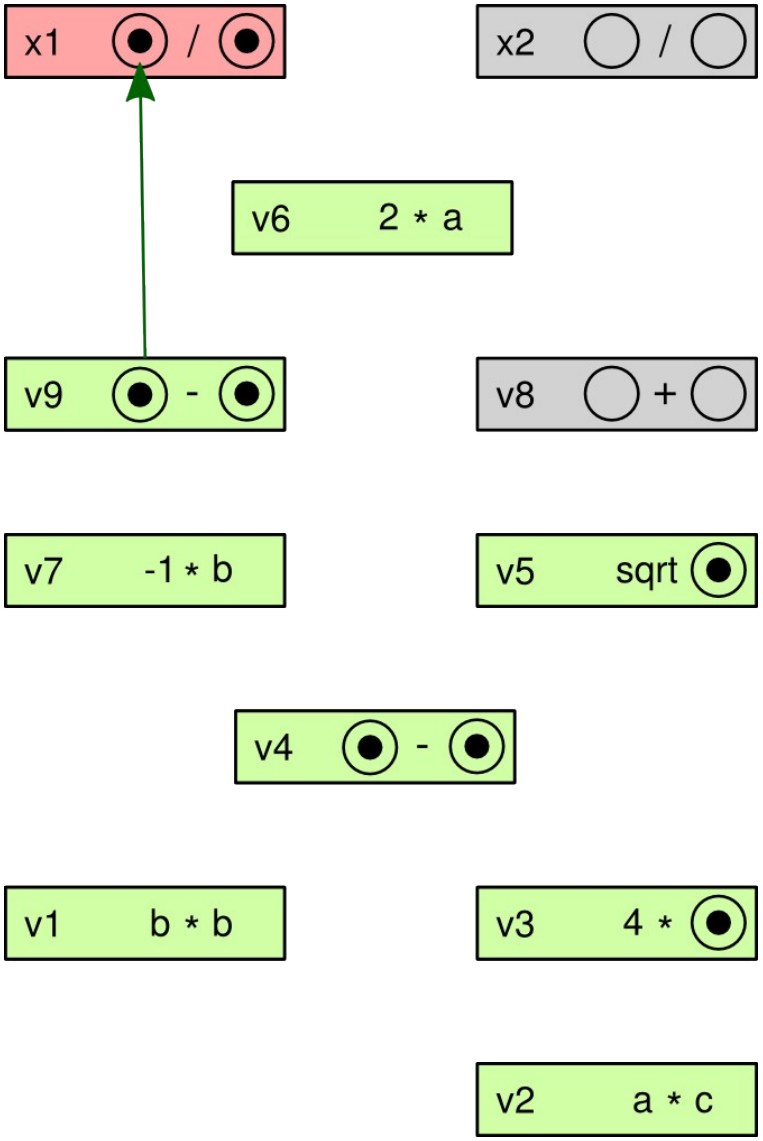
Igényvezérelt modell



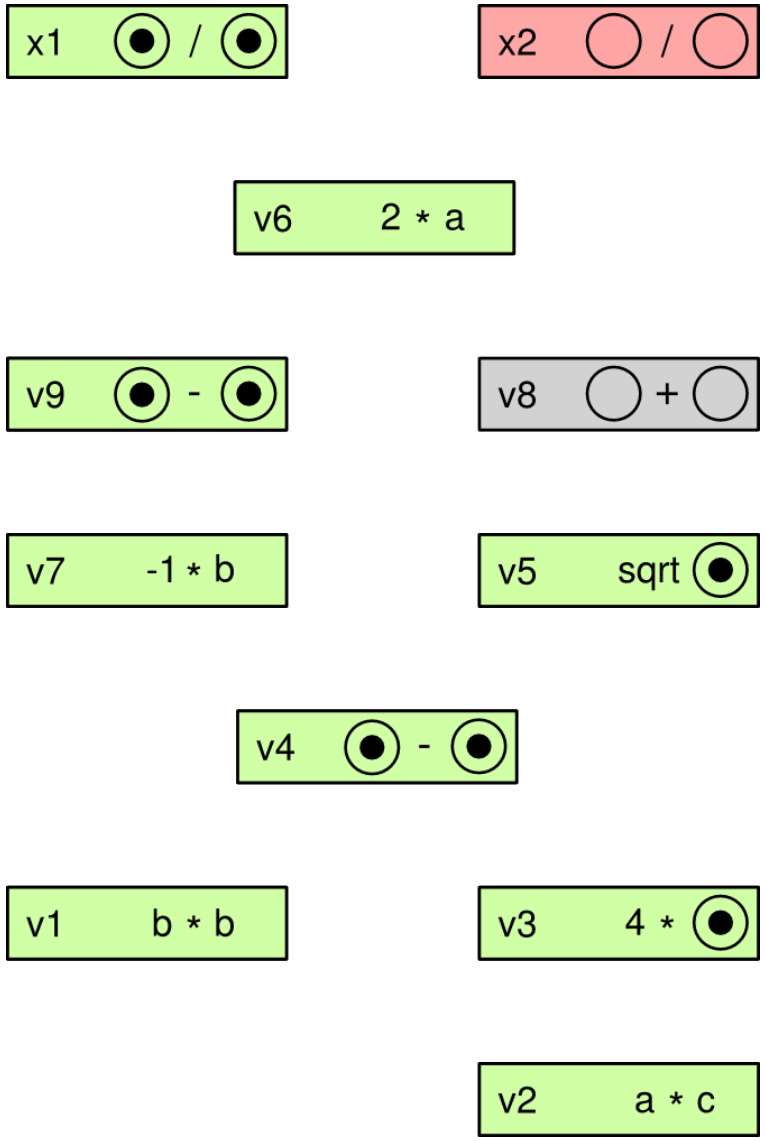
Igényvezérelt modell



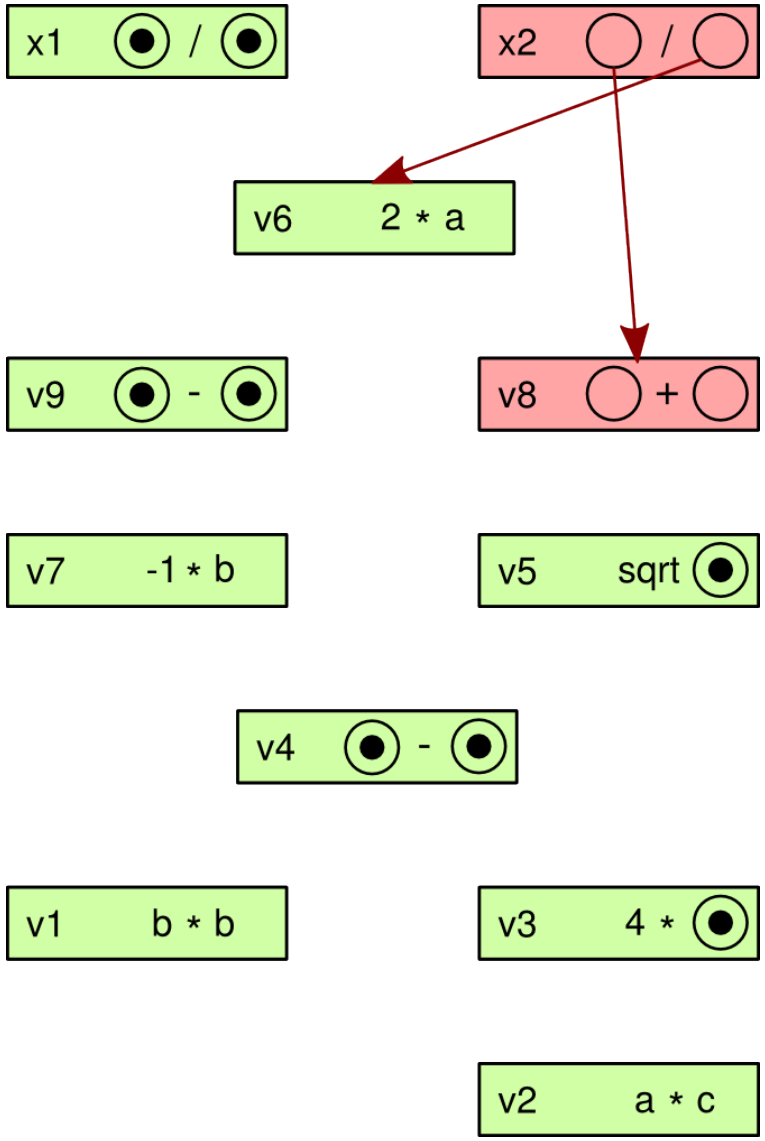
Igényvezérelt modell



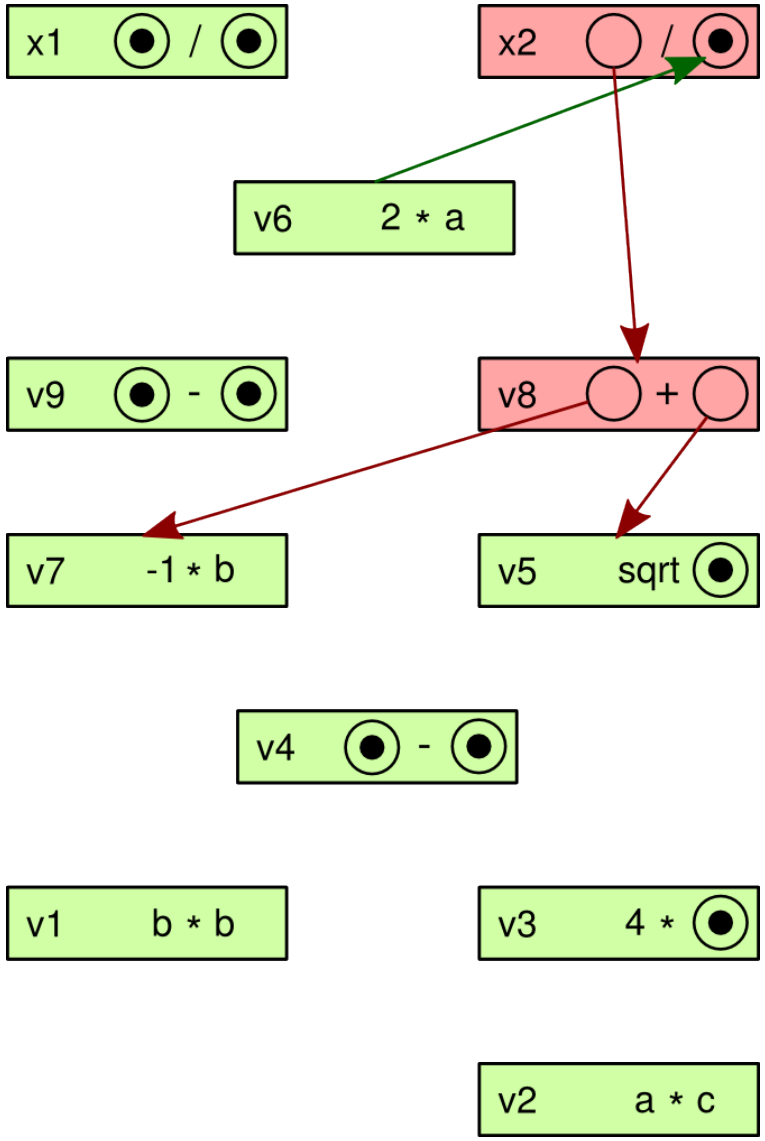
Igényvezérelt modell



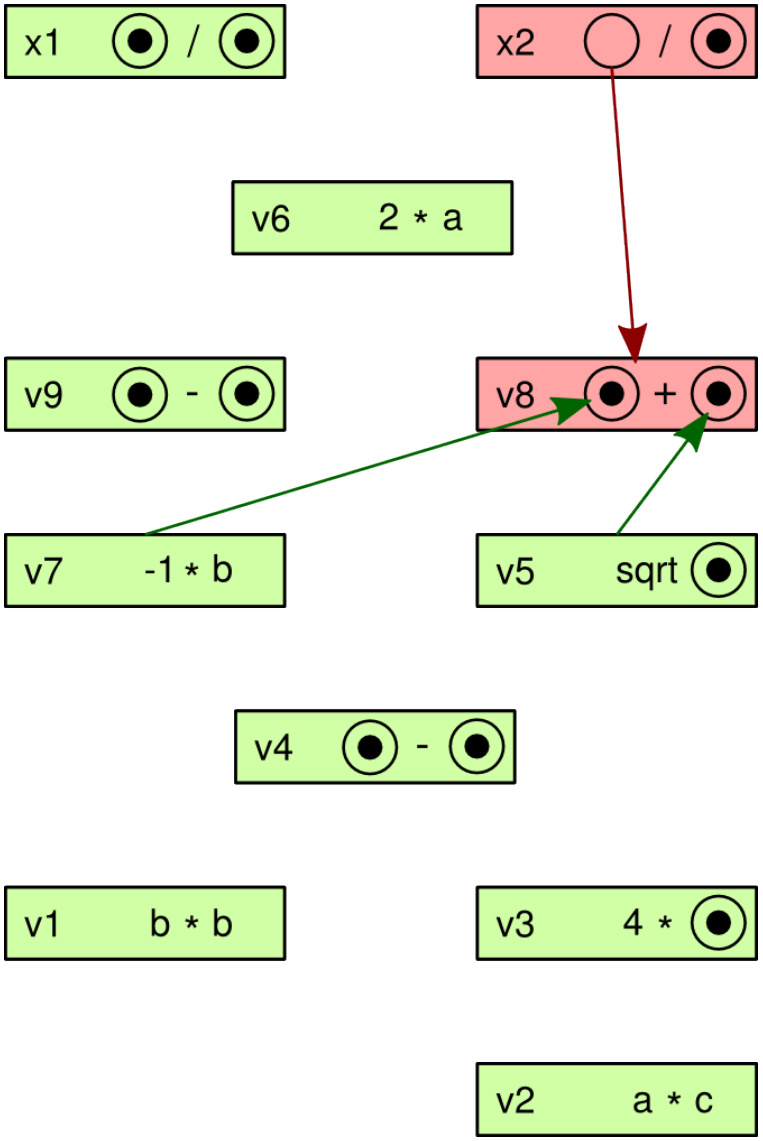
Igényvezérelt modell



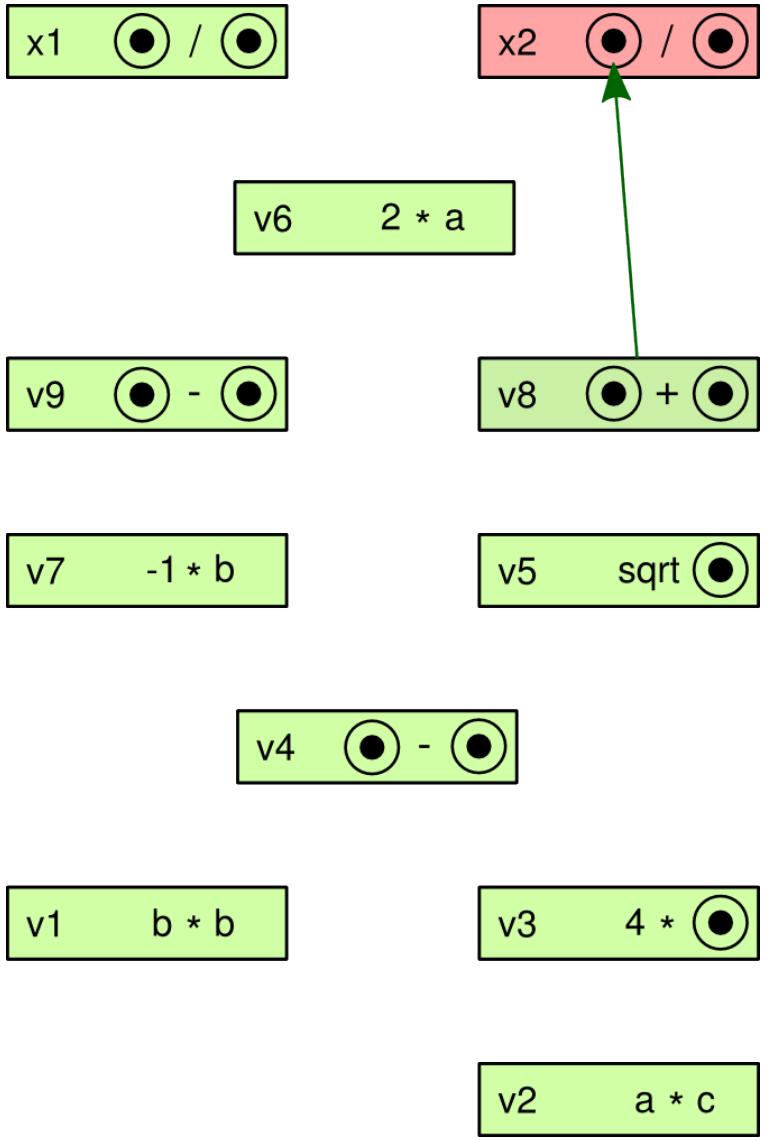
Igényvezérelt modell



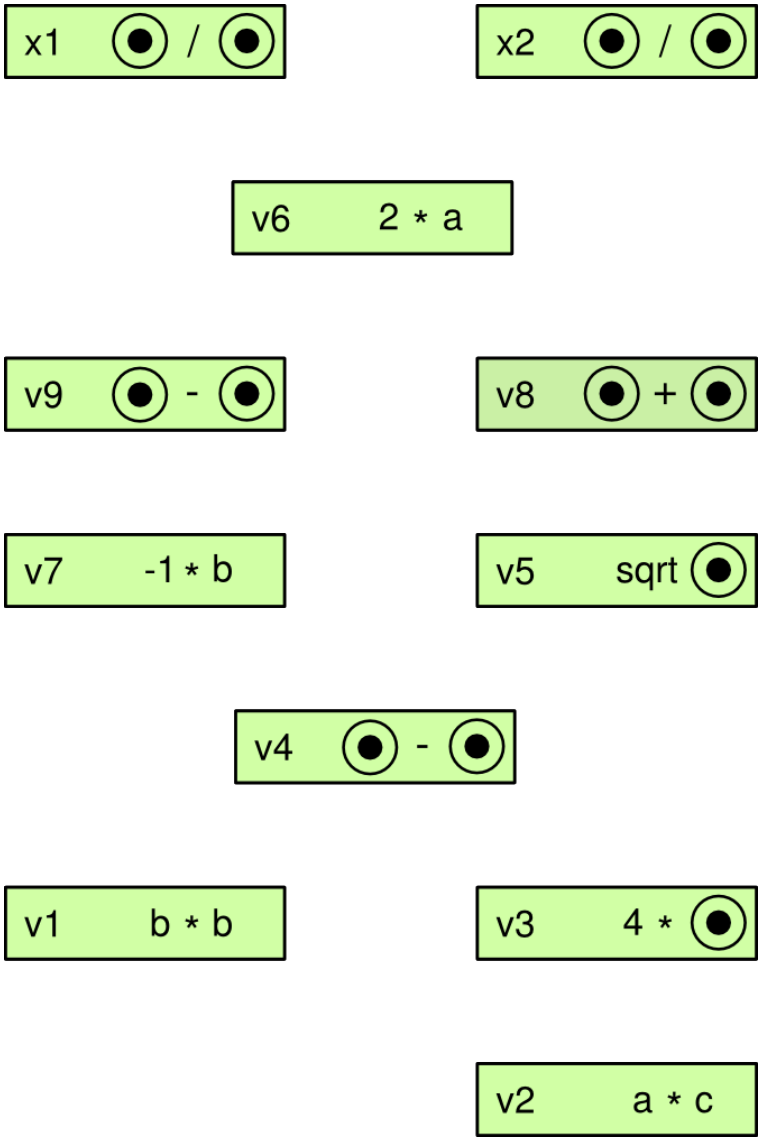
Igényvezérelt modell



Igényvezérelt modell



Igényvezérelt modell



Igényvezérelt modell

- Értékelés
  - Párhuzamosságot automatikusan felderíti
  - Tisztán igényvezérelt számítógép nem terjedt el
  - De az elv igen:
    - Funkcionális programnyelvek

Információfeldolgozási modellek

- Összegzés:
  - CPU-k programozói interfésze vezérlésáramlásos
  - E fölött lehet igényvezérelt elven működő funkcionális nyelvet használni
  - Ez alatt lehet adatáramlásos utasítás-végrehajtást megvalósítani

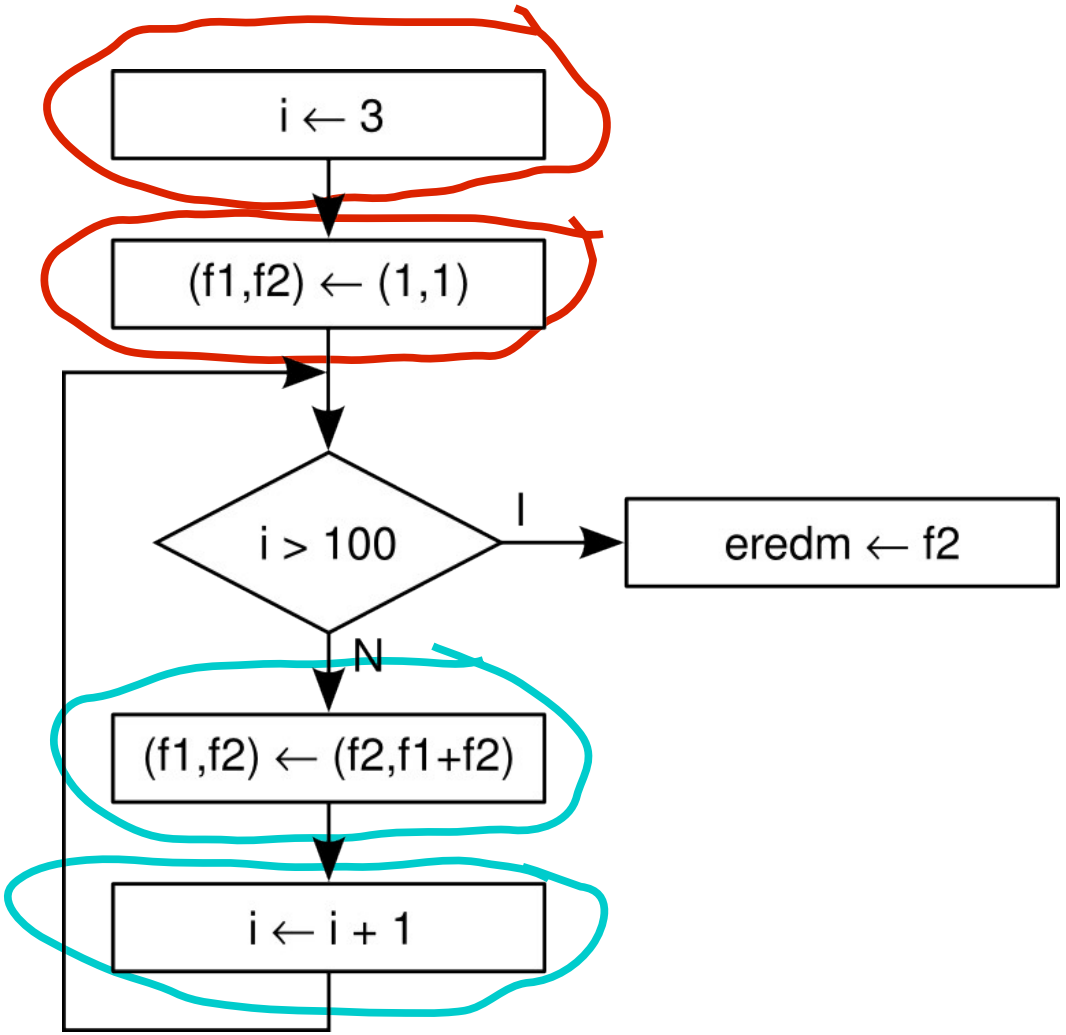
Esettanulmány

- Fibonacci sorozat 100-adik elemének kiszámolása

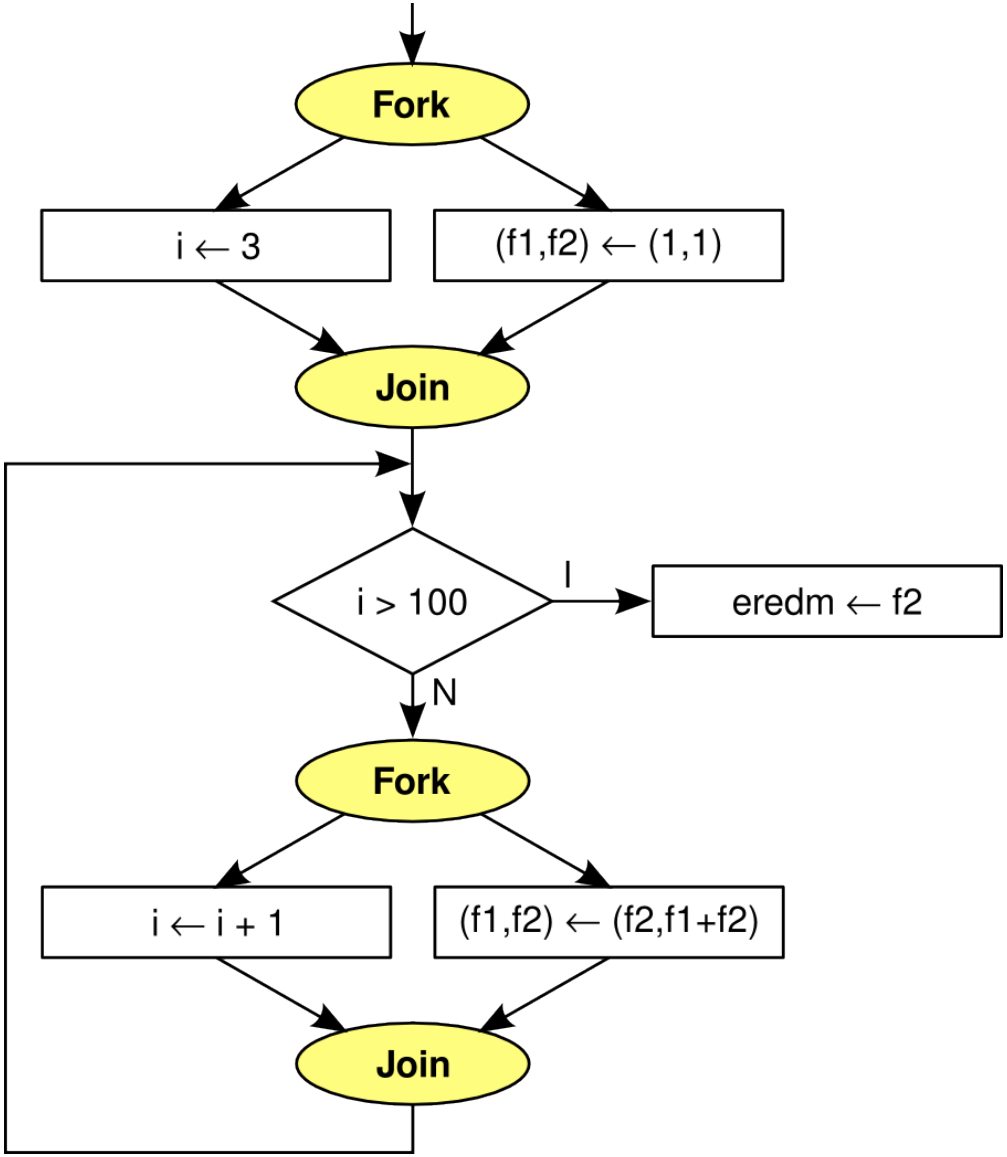
$$\phi_1 = 1, \phi_2 = 1,$$
$$\phi_n = \phi_{n-1} + \phi_{n-2},$$
$$\phi_{100} = ?$$

Vezérlésáramlásos modellel

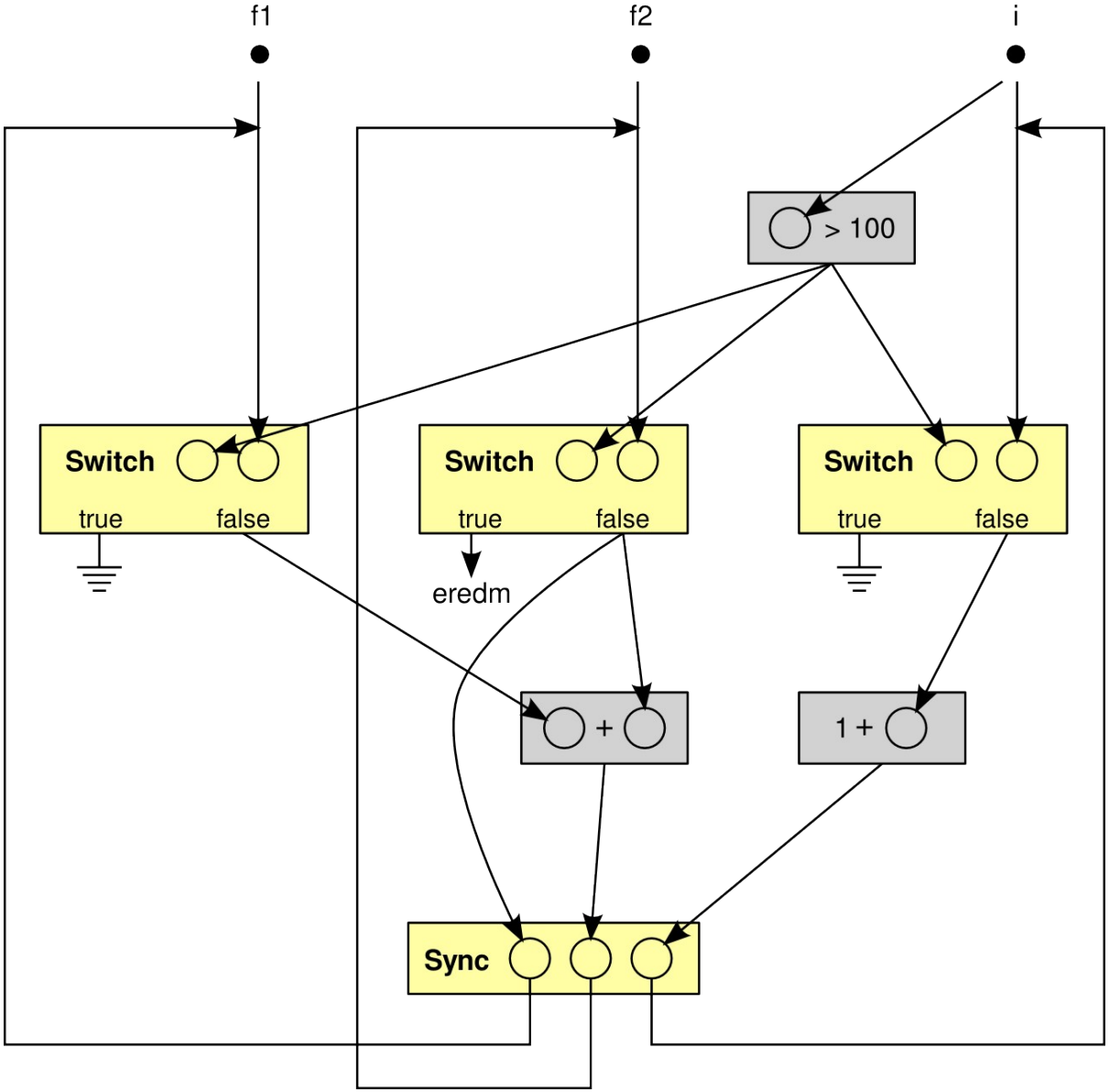
```
(f1,f2) ← (1,1);
for i=3 to 100 do
    (f1,f2) ← (f2, f1+f2);
end
eredm ← f2;
```



Vezérlésáramlásos modellel



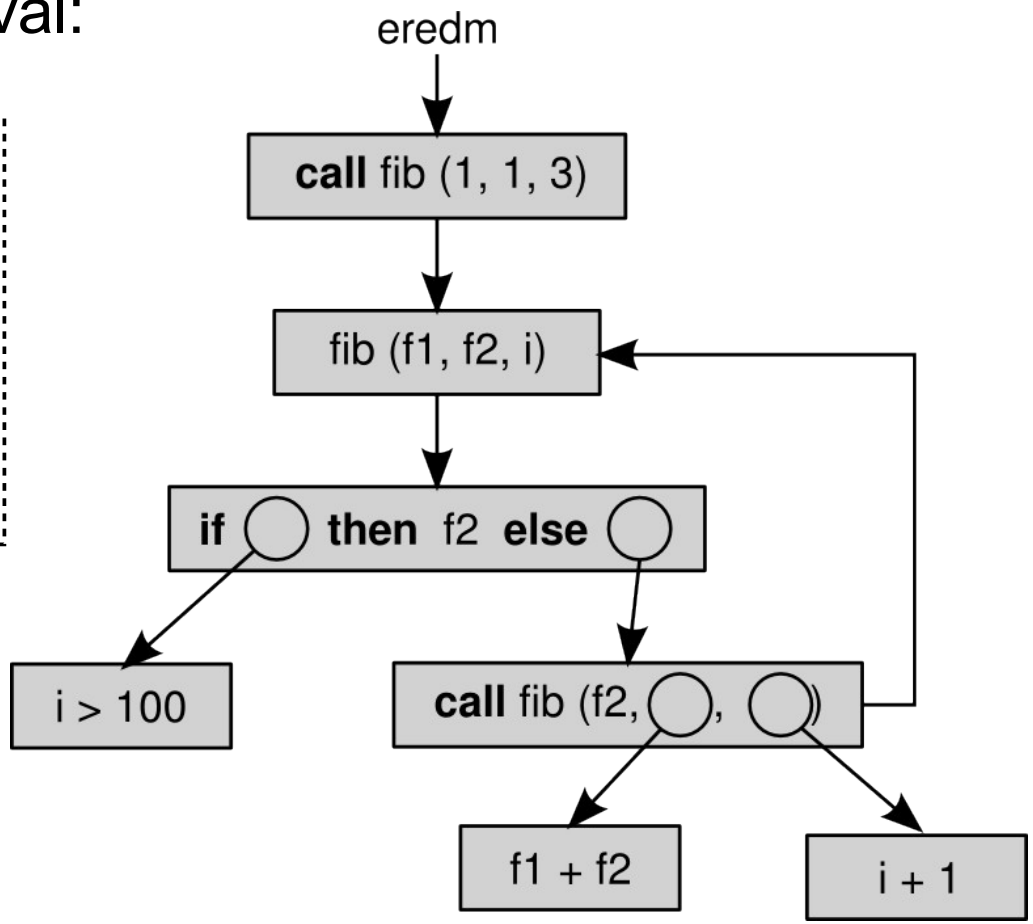
Adatáramlásos modellel

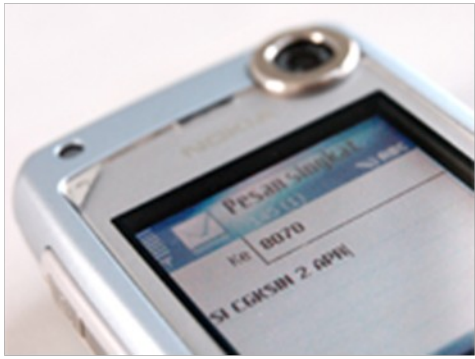


Igényvezérelt modellel

- A programot egyetlen összetett kifejezésként adjuk meg
- Legegyszerűbb rekurzióval:

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





Számítógép Architektúrák

2. A vezérlésáramlásos modell megvalósítása

Horváth Gábor
docens
BME Híradástechnikai Tanszék
ghorvath@hit.bme.hu

2012. szeptember 26.
Budapest

A Neumann architektúra

- 1945, "First Draft of a Report on the EDVAC" c. tanulmány
- Első Neumann elvű számítógép: Institute for Advanced Study, Princeton Egyetem (1945-1951)
- Igazából:
  - Turing – 1936
  - Eckert & Mauchly – 1943

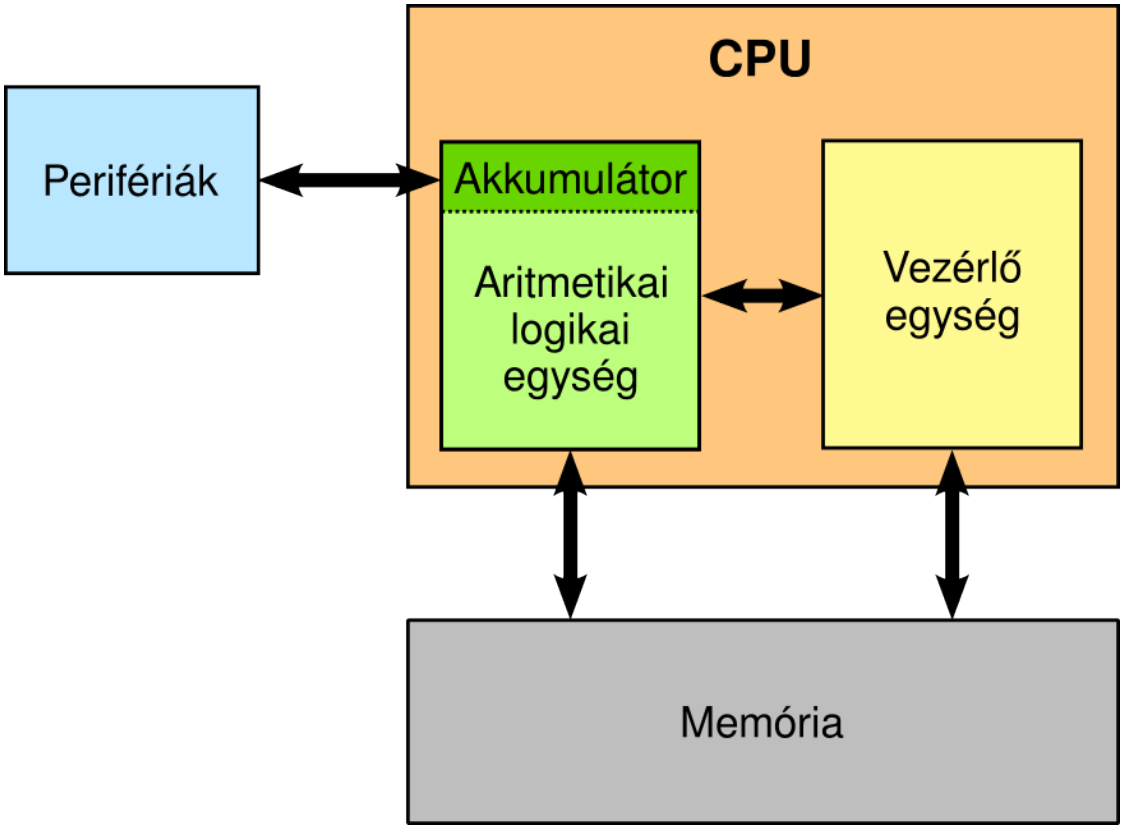
A Neumann architektúra

- Újdonság:

  Az utasítások az adatokkal együtt vannak eltárolva a memóriában

  - Korábban: programozás = kapcsolósor beállítás, újrahuzalozás
  - Motiváció:
    - A programot így könnyebb megváltoztatni
      (Új program az ENIAC-ra = 3 hét munka!!)
    - Gyorsabban lehet több programot egymás után futtatni
    - Programot generáló programot lehet írni
      – Fordítóprogramok születése

A Neumann architektúra

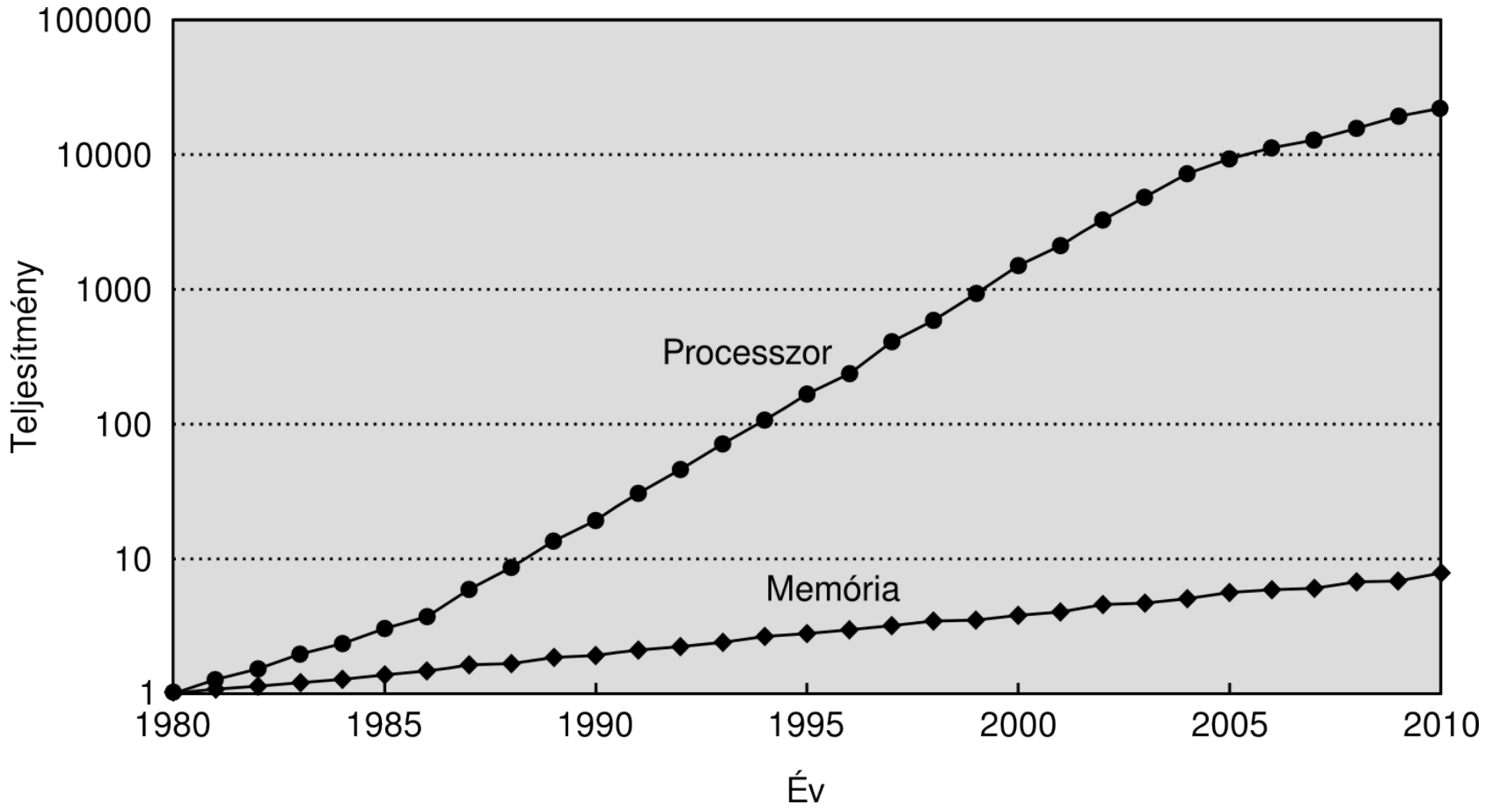


A Neumann architektúra

- Memória tartalma:
  - Fix bitszélességű adategységek
  - Lehet
    - Utasítás
    - Egész szám
    - Karakter
    - Lebegőpontos
    - …
  - Megkülönböztetés: **NINCS**!
  - Attól függ, milyen utasítás nyúl hozzá

A Neumann architektúra

- Szűk keresztmetszet:
  - **Memória-sávszélesség!**



A Neumann architektúra

- Önmódosító programok:
  - Memóriaműveletekkel átírják saját utasításaikat
  - Pár bájt spórolás vs. olvashatatlan kód
  - Soha!
- Példa: Fibonacci sorozat N. eleme

```
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:
```

Önmódosító program

Ciklusszervezés:
ciklusváltozó: R8

```
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:
```

Ciklusváltozó inicializálása

Ciklusfeltétel kiértékelése

Ciklusváltozó növelése

Következő iteráció

Önmódosító program

Eredmény gyűjtése az R2-be:
R2=fib(R8+1)

```
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:
```

Inicializálás

key-en lévő utasítás új értéke

Fibonacci sorozat köv. Elemének kiszámolása **Ez fog módosulni**

key-en lévő utasítás átírása az új értékre

Önmódosító program

R8 = 2

R10 = opcode of (R2 ← R2 + 1)

R2 = 1

```
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:
```

Önmódosító program

R8 = 2

R10 = opcode of (R2 ← R2 + 1)

R2 = 1

```
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 + 1
      MEM[key] ← R10
      JUMP loop
halt:
```

Önmódosító program

R8 = 3

R10 = opcode of (R2 ← R2 + 1)

R2 = 2

```
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 + 1
      MEM[key] ← R10
      JUMP loop
halt:
```

Önmódosító program

R8 = 3

R10 = opcode of (R2 ← R2 + 1)

R2 = 2

```
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 + 1
      MEM[key] ← R10
      JUMP loop
halt:
```

Önmódosító program

R8 = 4

R10 = opcode of (R2 ← R2 + 2)

R2 = 3

```
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 + 1
      MEM[key] ← R10
      JUMP loop
halt:
```

Önmódosító program

R8 = 4

R10 = opcode of (R2 ← R2 + 2)

R2 = 3

```
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 + 2
      MEM[key] ← R10
      JUMP loop
halt:
```

Önmódosító program

R8 = 5

R10 = opcode of (R2 ← R2 + 3)

R2 = 5

```
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 + 2
      MEM[key] ← R10
      JUMP loop
halt:
```

Önmódosító program

R8 = 5

R10 = opcode of (R2 ← R2 + 3)

R2 = 5

```
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 + 3
      MEM[key] ← R10
      JUMP loop
halt:
```

Önmódosító program

R8 = 6

R10 = opcode of (R2 ← R2 + 5)

R2 = 8

```
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 + 3
      MEM[key] ← R10
      JUMP loop
halt:
```

Önmódosító program

R8 = 6

R10 = opcode of (R2 ← R2 + 5)

R2 = 8

```
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 + 5
      MEM[key] ← R10
      JUMP loop
halt:
```

Önmódosító program

R8 = 7

R10 = opcode of (R2 ← R2 + 8)

R2 = 13

```
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 + 5
      MEM[key] ← R10
      JUMP loop
halt:
```

Önmódosító program

R8 = 7

R10 = opcode of (R2 ← R2 + 8)

R2 = 13

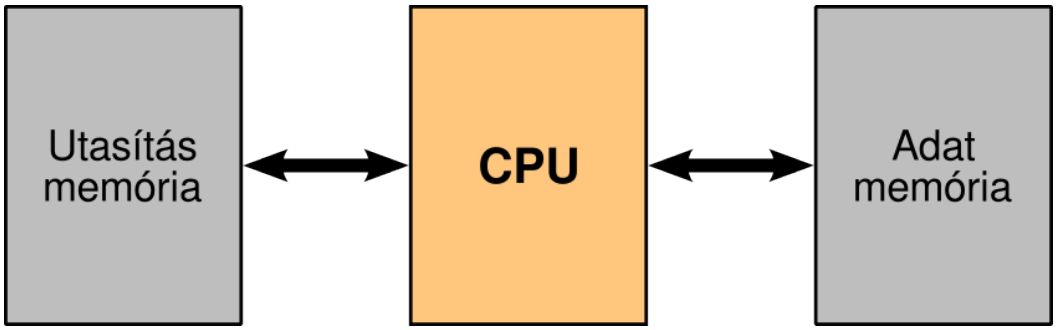
```
main: R4 ←  N
      R9 ← opcode of (R2 ← R2 + 0)
      R8 ← 1
      R2 ← 1
loop: JUMP halt IF R8 == R4
      R8 ← R8 + 1
      R10 ← R9 + R2
key:  R2 ← R2 + 8
      MEM[key] ← R10
      JUMP loop
halt:
```

stb...

A Harvard Architektúra

- Harvard Mark I:
  - Lyukkártyáról olvassa az utasításokat
  - Relékapcsolókból álló memória tárolja az adatokat
- Eltérés a Neumann architektúrához képest:

Az utasítások és az adatok tárolása fizikailag elkülönül egymástól



A Harvard Architektúra

- Klasszikus Harvard architektúra:
  - Utasítások: kizárólag az utasításmemóriában
  - Adatok: kizárólag az adatmemóriában
- A kétféle memóriaművelet átlapolható:
  - Amíg lehívja az i. utasítás operandusát
  - Azzal egy időben lehívhatja az i+1. utasítást
- Módosított Harvard architektúra:
  - Utasítás memóriából is lehet adatot olvasni
- Alkalmazás:
  - Mikrokontrollerek, jelfeldolgozó processzorok (PIC, Atmel, ...)

Utasításkészletek

- Processzorok jellemzője:

  Programozási felület / Utasításkészlet architektúra

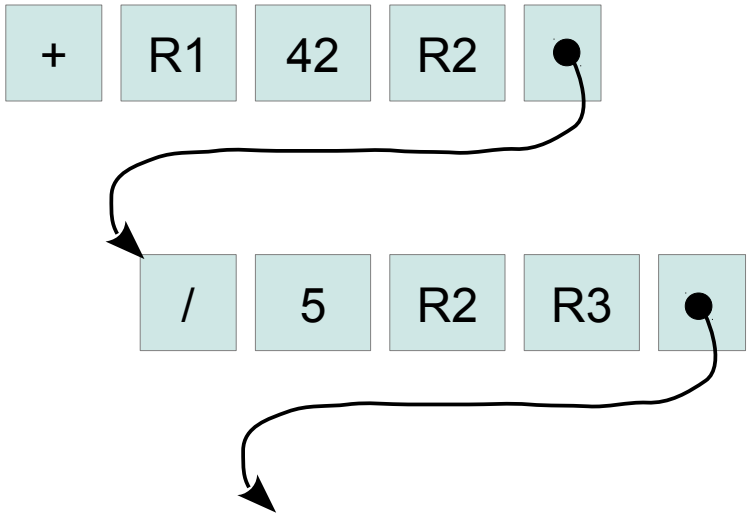
- Részei:
  - Utasítások
  - Támogatott adattípusok
  - Regiszterek
  - Címzési módok
  - Jelzőbitek
  - Perifériakezelési mód
  - Megszakítás- és kivételkezelés

Utasításkészletek

- Mit tanulunk erről?
  - Miből áll egy utasítás?
  - Milyen fajta utasítások vannak?
  - És az operandusok hol vannak?
  - Hogy kezelhető a feltételes ugrás és végrehajtás?
  - Milyen formában tároljuk az utasításokat?
  - Mi az a little-endian és big-endian?
  - RISC vs. CISC

Utasítások jellemzői

- Utasítások felépítése:
  - Utasítás kódja / művelet típusa
  - Operandusok címei / értékei
  - Eredmény címe
  - Következő utasítás címe
- Takarékosság:
  - Köv. utasítás címe elhagyható
  - Operadusok száma:
    - 3 operandusú műveletek: **R1 ← R2 + R3**
    - 2 operandusú műveletek: **R1 ← R1 + R2**
    - 1 operandusú műveletek: **ADD R1**



Utasítások jellemzői

- Utasítások fajtái:
  - Adatmozgatás

    `R1←R2, R1←MEM[100], R1←42, MEM[100]←R1, MEM[100]←42`
  - Aritmetikai és logikai

    `R1 ← R2+R3, R1 ← MEM[100]*42, MEM[100] ← R1 & R2`
  - Vezérlésátadó:

    `JUMP -42, JUMP +28 IF R1==R2, CALL proc, RETURN`
  - Veremkezelő

    `PUSH R1, PUSH 42, R2 ← POP`
  - I/O műveletek

    `OUT 42 ← R1, IN R1 ← 42`
  - Transzcendens függvények

    `R2 ← SIN R1, R2 ← SQRT 42`
  - Egyebek

Címzési módok

- Címzési mód megadja, hogy hol az operandus
- Ahol lehet:
  - Magában az utasításkódban
  - Regiszterben
  - Memóriában

Címzési mód	Példa
Regiszter	`R1 ← R2 + R3`
Közvetlen konstans	`R1 ← R2 + 42`
Direkt	`R1 ← R2 + MEM[42]`
Regiszter indirekt	`R1 ← R2 + MEM[R3]`
Eltolt indirekt	`R1 ← R2 + MEM[R3+42]`
Memória indirekt	`R1 ← R2 + MEM[MEM[R3]]`
Indexelt	`R1 ← R2 + MEM[R3+R4]`

Vezérlésátadó utasítások

- Célszerűen relatív címzés, pl. JUMP -28
- Feltételes ugrás 3 jellemző megvalósítása:
  - Feltétel kódokkal:

```
COMPARE R1, R2
JUMP label IF GREATER
```

  - Feltétel regiszterekkel:

```
R0 ← R1 > R2
JUMP label IF R0
```

  - „Összehasonlít és ugrik”:

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

Predikátumok

- Predikátumok: feltételhez kötött utasítások

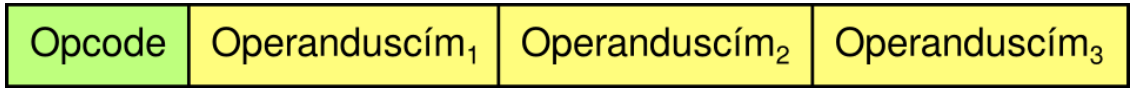
  `R1 ← R2 + 32 IF P2`

- Predikátumregiszter: végrehajtási feltételt tárol
- Állítása: összehasonlító műveletekkel

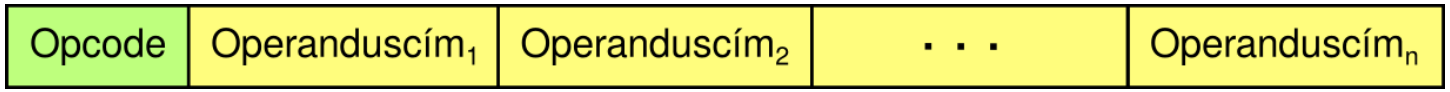
  `P2 ← R3 ≤ R5`

Utasítások jellemzői

- Utasításokat binárisan kódolva tároljuk
- Binárisan kódolt utasítások hossza szerint:
  - Fix hosszú kódolás:

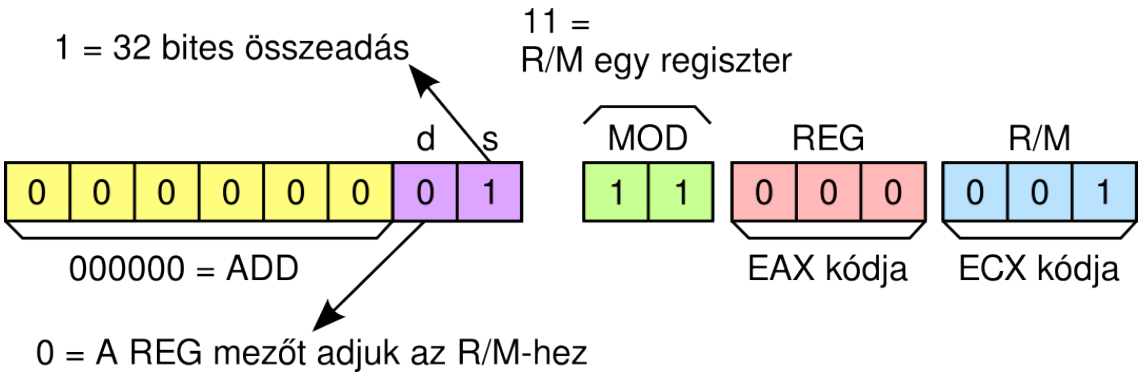


  - Változó hosszú kódolás:



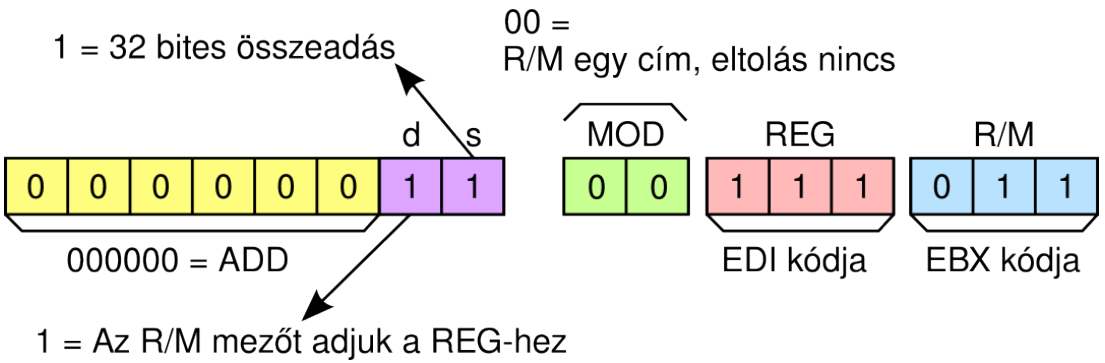
Példa utasításkódolásra (x86)

- ADD ECX, EAX



= 01 C1
(ASCII: ☺┴)

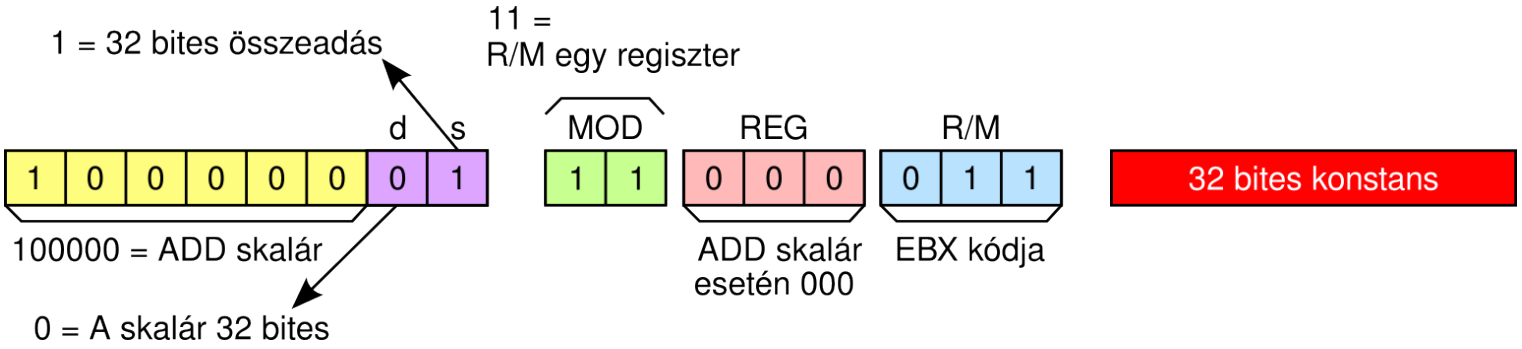
- ADD EDI, [EBX]



= 03 3B
(ASCII: ♥;)

Példa utasításkódolásra (x86)

- ADD EBX, 23423765



- = 81 C3 15 6B 65 01 (ASCII: Qü§ke☺)

Utasításkészletek további jellemzői

- Bájtsorrend:
  - Little endian: legkisebb helyiértékkel kezdi
  - Big endian: legnagyobb helyiértékkel kezdi
  - Példa: 23423765 (=1656B15)
    - Little endian: 15 6B 65 01
    - Big endian: 01 65 6B 15
- Perifériakezelő utasítások:
  - Külön I/O utasítások (IN/OUT)
  - Memóriára leképzett
- Ortogonalitás ill. közel ortogonalitás

Assembly programozás

- Alacsony szintű programozás:
  - Utasítások kézi kódolása kényelmetlen
  - Assembly-t használnak
  - Minden utasításkészlet architektúrára más és más!

RISC vs. CISC

- „Trend” a 70-es években: sok, összetett utasítás
- Motiváció:
  - Lassú memória
  - Drága memória
  - Egyszerűbb fordítóprogram
- Ez a **CISC** (Complex Instruction Set Computer)
- Jellemzői:
  - Kényelmes, összetett műveletek
  - Regiszter-memória utasítások (pl. `R1 ← R2 + MEM[42]`)
  - Redundancia
  - Sokféle címzési mód
  - Változatos utasításhossz
  - Változatos utasítás-végrehajtási idő

RISC vs. CISC

- „Trend” a 80-as/90-es években: egyszerű utasítások
- Motiváció:
  - Egyszerűbb processzor
  - Átláthatóbb mikroarchitektúra

    → hatékonyabb implementáció
- Ez a **RISC** (Reduced Instruction Set Computing)
- Jellemzői:
  - Elemi utasítások, redundancia kerülése
  - Load-Store és regiszter-regiszter műveletek

    `R1 ← R2+MEM[42]` helyett `R3 ← MEM[42]; R1 ← R2+R3.`
  - Kevés címzési mód
  - Fix utasításhossz
  - Egyforma utasítás-végrehajtási idők

RISC vs. CISC

- Összehasonlítás:
  - CISC: tömör
  - RISC: egyszerű
    - Kevesebb tervezési hiba
    - Huzalozott vezérlés (vs. Mikroprogramozás)
    - Kisebb IC
      - Alacsonyabb fogyasztás
      - Jobb gyártási kihozatal
      - Járulékos eszközökkel integrálható
  - CISC: kevés regiszter vs. RISC: több regiszter

Néhány érdekes utasításkészlet

- MISC: Minimal Istruction Set Computer
  - pl. verem alapú
- OISC: One Istruction Set Computer
  - pl. kivonás és ugrás, ha az eredmény negatív
- NISC: No Instruction Set Computer
  - Nanokódot használ: közvetlenül a programozó vezérli a CPU funkcionális egységeit

Történelemóra

x86

- Első tag: 1971, Intel 8086
- 1981: Az Intel 8088-as lesz az IBM PC processzora
- Eredetileg 16 bites volt, később 32 és 64 bites kiterjesztések
- Nagy teljesítményű szerverektől a mobil eszközök piacáig
- 40 éves kompatibilitási kényszer
- Félvezetőipari beruházások

  → hatalmas gyártástechnológiai előny

Történelemóra

ARM

- Első megvalósítás: 1987
- A legelterjedtebb utasításkészlet architektúra
- Kezdetektől fogva 32 bites
- Az ARM utasításkészlet architektúra és a processzorok licenszelhetők
- Elsődleges szempont az egyszerűség és a fogyasztás

Történelemóra

PowerPC

- 1991-ben, az IBM, Apple és Motorola összefogásában
- Felülmúlják az x86 processzorok teljesítményét
  (2010: 4.25 GHz, 8 magos, magonként 4 szál)
- A PC-kben nem terjedt el, munkaállomásokban és szerverekben igen
- Modern játékkonzolok mindegyike PowerPC-re épül

Történelemóra

SPARC

- 1987, SUN
- Kezdetektől 64 bites
- Nyílt platform
  A UltraSPARC T1 és T2 VeriLog szinten elérhető!!!
- Jelenleg a világ legerősebb számítógépe SPARC alapú (2011. 06. 20.)

Történelemóra

m68k

- 1979, Motorola 68000
- Kezdetektől 32 bites
- Az x86 legnagyobb vetélytársa volt
- Apple Macintosh, Commodore Amiga, Atari ST, SEGA MegaDrive, korai HP és SUN szerverek, lézernyomtatók,...
- Utolsó tag: 1994, Motorola 68060, kb. Pentium sebesség
- Halála: Motorola belépett a PowerPC szövetségbe

Történelemóra

Alpha

- 1992, DEC Alpha 21064
- Kezdetektől 64 bites
- Rendkívül innovatív:
  - 21164: első CPU nagy, a CPU-val egy szilíciumon elhelyezett cache-el
  - 21264: első CPU magas órajellel ÉS sorrenden kívüli végrehajtással
  - 21364: első CPU integrált memóriavezérlővel
  - 21464: első többszálúságot támogató CPU (lett volna, ha forgalomba kerül)
- Nagyon erős lebegőpontos egység

  21264 @ 833 MHz > 3x Pentium III @ 1 GHz!
- Kézi tervezés
- Halála: Compaq felvásárolja és leállítja a fejlesztést

Történelemóra

PA-RISC

- 1986, HP PA-RISC
- Eleinte 32, majd 64 bites
- A korosodó m86k szerverek kiváltására
- Nem használ másodszintű cache-t, de elsőszintűt annyit, amennyit azóta sem használ senki
- Nagyon erős lebegőpontos egység

  PA-8600 @ 552 MHz > 2x Pentium III @ 1 GHz!
- Halála: HP az Intel Itanium mellett teszi le a voksát

Történelemóra

IA-64 (Itanium)

- 1994-ben indul a HP & Intel közös fejlesztés
- Hatalmas sajtófigyelem, költséges fejlesztés
- Első tag: 2001, kiábrándító teljesítménnyel, pár ezret adtak el
- Hardveres kompatibilitás az x86-tal: a 100MHz Pentium sebességén...
- Gond: spéci compiler kell, nem gondolták, hogy ez ilyen nehéz
- Azóta is fejlesztik, 2001-2007 között 55.000 db-ot adtak el
- Egyre több cég hagy fel a platform támogatásával
  - 2008: Microsoft
  - 2011 március: Oracle

Összehasonlítás

	x86	ARM	PowerPC	SPARC
Hány bites	64	32	64	64
Megjelenés éve	1978	1983	1991	1985
Operandusok	2	3	3	3
Műveletek	Reg-mem	Reg-reg	Reg-reg	Reg-reg
CISC vs. RISC	CISC	RISC	RISC	RISC
Regiszterek sz.	8/16	16	32	32
Utasításkódolás	Vált. (1-17)	Fix (4)	Fix (4 – töm.)	Fix (4)
Felt. utasítások	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. lekép.	Mem. lekép.	Mem. lekép.
Predikátumok	Nincs	Van	Nincs	Nincs

Összehasonlítás

	m68k	Alpha	PA-RISC	Itanium
Hány bites	32	64	64	64
Megjelenés éve	1979	1992	1986	2001
Operandusok	2	3	3	3
Műveletek	Reg-mem	Reg-reg	Reg-reg	Reg-reg
CISC vs. RISC	CISC	RISC	RISC	EPIC
Regiszterek sz.	16	32	32	128
Utasításkódolás	Vált. (2-22)	Fix (4)	Fix (4)	Fix (16)
Felt. utasítások	Feltétel kód	Feltétel reg.	Összeh. & ugr.	?
Bájtsorrend	Big	Bi	Big	Bi
Címzési módok	9	1	5	?
Perifériakezelés	Mem. lekép.	Mem. lekép.	Mem. lekép.	Mem. lekép.
Predikátumok	Nincs	Nincs	Nincs	Van

Esettanulmány: Floyd-Warshall

```
#define N 100

int main () {

    int i, j, k, t;
    unsigned int path[N][N];

    // gráf éleinek hosszát véletlenszerűen beállítjuk
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            path[i][j] = rand();

    // legrövidebb út keresése
    for (k=0; k<N; k++)
        for (i=0; i<N; i++)
            for (j=0; j<N; j++) {
                t = path[i][k]+path[k][j];
                if (t < path[i][j])
                    path[i][j] = t;
            }
}
```

Esettanulmány

Megvalósítás x86-ban (=IA-32)

- 8 regiszter: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
- Címzési módok:
  - Address=Base+(Index*Scale)+Displacement

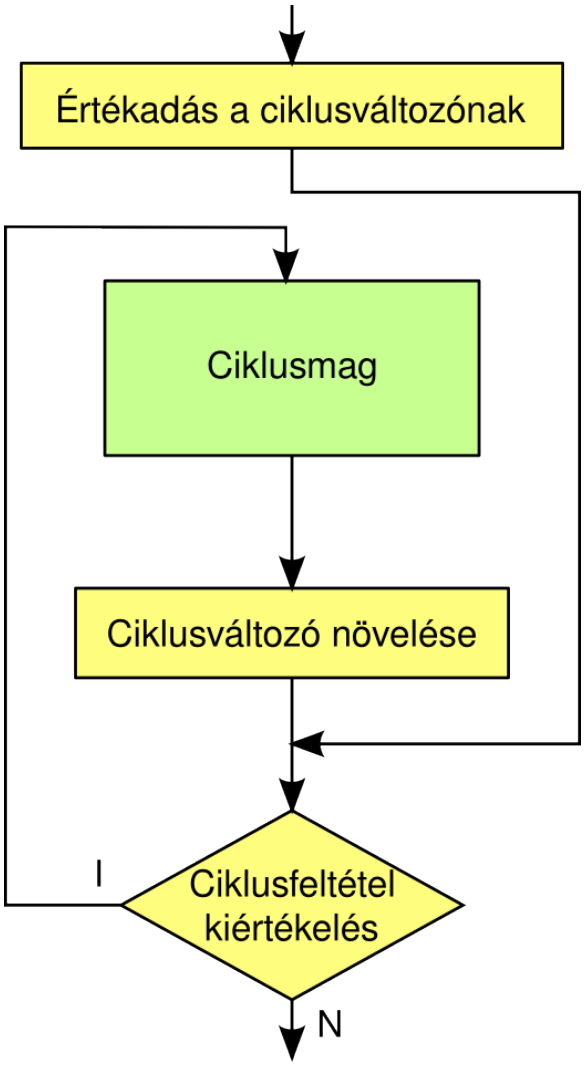
    pl. `MOV DWORD PTR [EBP+EBX*4+0004A0FCh], EAX`
- ESP mutat a stack frame-re (lokális változók kezdetére)

```
int i, j, k, t;
unsigned int path[N][N];
```

  - [ESP]: path tömb (4*100*100=40.000 bájt)
  - [esp+40000]: i változó (4 bájt)
  - [esp+40004]: j változó (4 bájt)
  - [esp+40008]: k változó (4 bájt)
  - [esp+40012]: t változó (4 bájt)

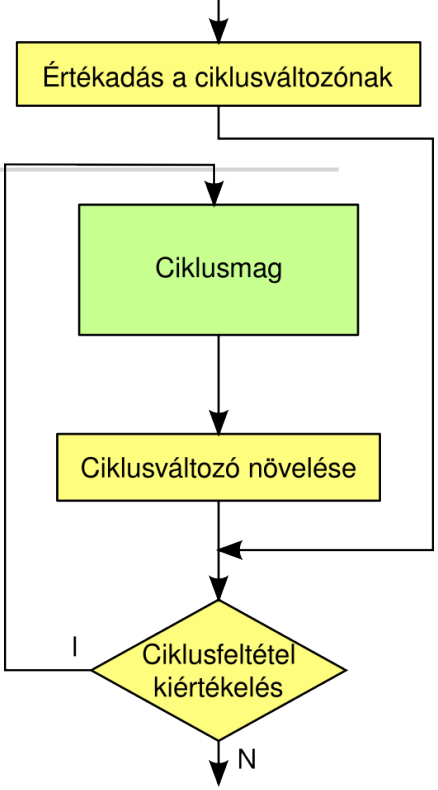
Esettanulmány

- Ciklus megvalósítása:



Esettanulmány

```
        mov     DWORD PTR [esp+40000], 0
        jmp     .L2
.L5:
        mov     DWORD PTR [esp+40004], 0
        jmp     .L3
.L4:
        mov     esi, DWORD PTR [esp+40000]
        mov     ebx, DWORD PTR [esp+40004]
        call    rand
        imul    edx, esi, 100
        add     edx, ebx
        mov     DWORD PTR [esp+edx*4], eax
        add     DWORD PTR [esp+40004], 1
.L3:
        cmp     DWORD PTR [esp+40004], 99
        jle     .L4
        add     DWORD PTR [esp+40000], 1
.L2:
        cmp     DWORD PTR [esp+40000], 99
        jle     .L5
```

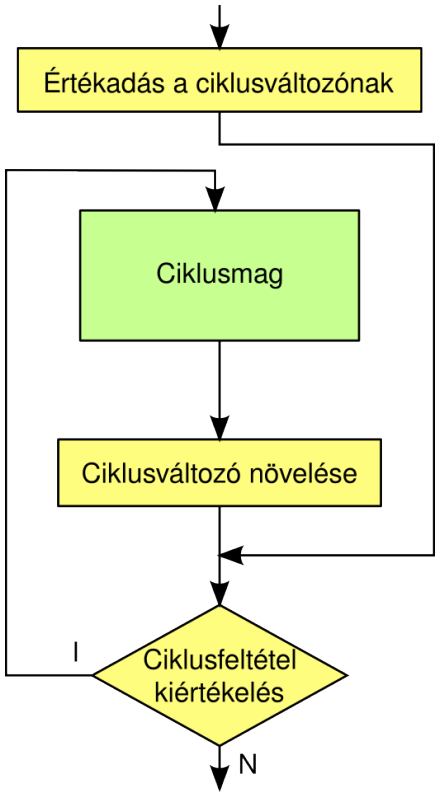



```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        path[i][j] = rand();
```

Esettanulmány

```
        mov     DWORD PTR [esp+40000], 0
        jmp     .L2
.L5:
        mov     DWORD PTR [esp+40004], 0
        jmp     .L3
.L4:
        mov     esi, DWORD PTR [esp+40000]
        mov     ebx, DWORD PTR [esp+40004]
        call    rand
        imul    edx, esi, 100
        add     edx, ebx
        mov     DWORD PTR [esp+edx*4], eax
        add     DWORD PTR [esp+40004], 1
.L3:
        cmp     DWORD PTR [esp+40004], 99
        jle     .L4
        add     DWORD PTR [esp+40000], 1
.L2:
        cmp     DWORD PTR [esp+40000], 99
        jle     .L5
```

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        path[i][j] = rand();
```



Esettanulmány

```
        mov     DWORD PTR [esp+40000], 0
        jmp     .L2
.L5:
        mov     DWORD PTR [esp+40004], 0
        jmp     .L3
.L4:
        mov     esi, DWORD PTR [esp+40000]
        mov     ebx, DWORD PTR [esp+40004]
        call    rand
        imul    edx, esi, 100
        add     edx, ebx
        mov     DWORD PTR [esp+edx*4], eax
        add     DWORD PTR [esp+40004], 1
.L3:
        cmp     DWORD PTR [esp+40004], 99
        jle     .L4
        add     DWORD PTR [esp+40000], 1
.L2:
        cmp     DWORD PTR [esp+40000], 99
        jle     .L5
```

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        path[i][j] = rand();
```

esi ← i

ebx ← j

eax ← rand()

edx ← i * 100

edx ← i * 100 + j

MEM[esp+(i*100+j)*4] ← rand()

Esettanulmány

- Fordító meg sem kísérelte regiszterbe tenni a változókat
- Minden változó meghivatkozása memóriaműveletteljár
- Kihasználja a CISC jellegzetességeket:
  - az összetett címzési módot:
    *MEM[esp+edx*4]* ← *rand()* (edx = i*100+j)
  - memóriareferens műveleteket:
    add DWORD PTR [esp+40004], 1 (j++)

Esettanulmány

Megvalósítás ARM utasításkészlettel

- 8 regiszter: r0-r15 (r13,r14,r15 speciális célú)
- Predikátumok:

```
CMP       R3, R2
MOVGE     R4, R3
MOVLT     R4, R2
```

- 3 operandusú aritmetikai és logikai műveletek (R3←R1-R2):

```
SUB       R3, R1, R2
```

- Második operandus előtt ingyen shift (R7←R5+R6*4):

```
ADD       R7, R5, R6, lsl #2
```

- Szorzás-összeadás 1 lépésben (R1←R2*R3+R4):

```
MLA         R1, R2, R3, R4
```

- Hatékony címzési módok (R1←MEM[R5+R6*4]):

```
LDR       R1, [R5, R6, #2]
```

Esettanulmány

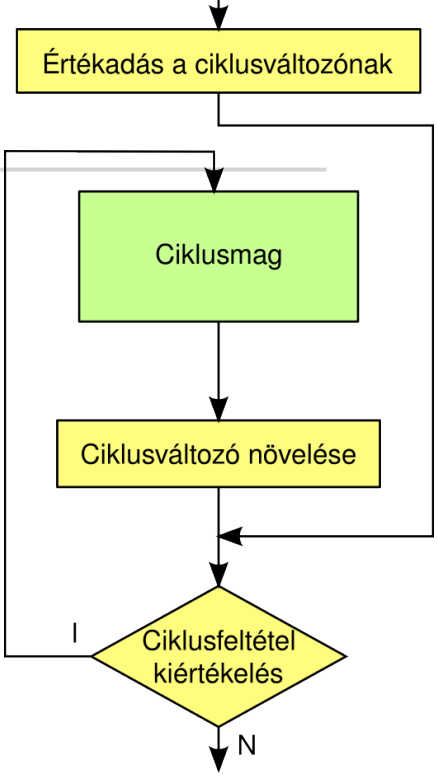
- Lokális változók:

```
int i, j, k, t;
unsigned int path[N][N];
```

  - [fp-40028]: path tömb (4*100*100=40.000 bájt)
  - [fp-28]: i változó (4 bájt)
  - [fp-24]: j változó (4 bájt)
  - [fp-20]: k változó (4 bájt)
  - [fp-16]: t változó (4 bájt)

Esettanulmány

```
        mov     r3, #0
        str     r3, [fp, #-28]
        b       .L4
.L7:
        mov     r3, #0
        str     r3, [fp, #-24]
        b       .L5
.L6:
        ...
.L5:
        ldr     r3, [fp, #-24]
        cmp     r3, #99
        movgt   r3, #0
        movle   r3, #1
        and     r3, r3, #255
        cmp     r3, #0
        bne     .L6
        ldr     r3, [fp, #-28]
        add     r3, r3, #1
        str     r3, [fp, #-28]
.L4:
        ldr     r3, [fp, #-28]
        cmp     r3, #99
        movgt   r3, #0
        movle   r3, #1
        and     r3, r3, #255
        cmp     r3, #0
        bne     .L7
```



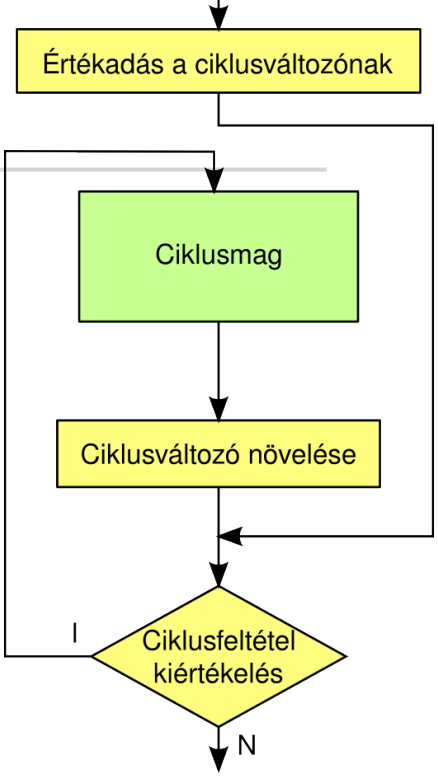

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        path[i][j] = rand();
```

A logikai kifejezés kiértékelése predikátumokkal!

Ha nem hamis, ugrik (még egy kör)

Esettanulmány

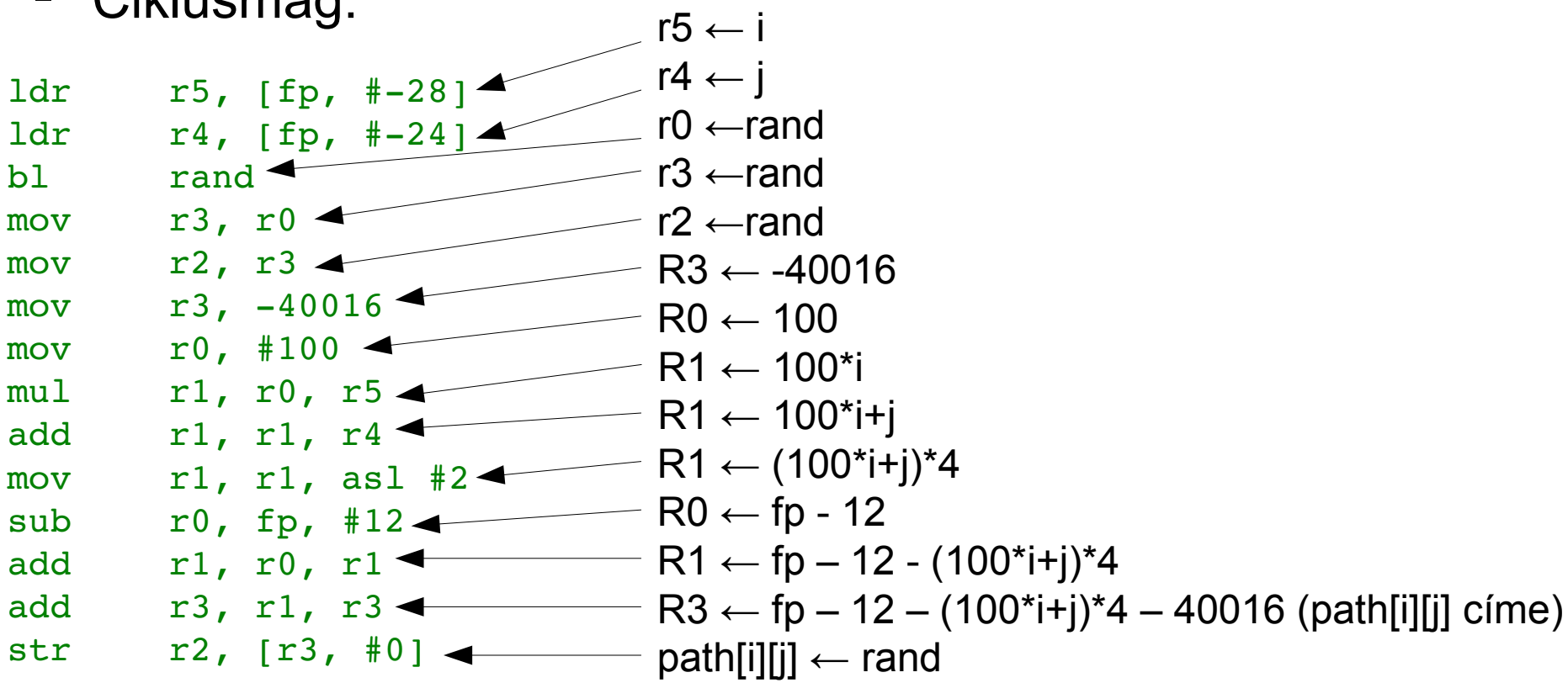
```
        mov     r3, #0
        str     r3, [fp, #-28]
        b       .L4
.L7:
        mov     r3, #0
        str     r3, [fp, #-24]
        b       .L5
.L6:
        ...
.L5:
        ldr     r3, [fp, #-24]
        cmp     r3, #99
        movgt   r3, #0
        movle   r3, #1
        and     r3, r3, #255
        cmp     r3, #0
        bne     .L6
        ldr     r3, [fp, #-28]
        add     r3, r3, #1
        str     r3, [fp, #-28]
.L4:
        ldr     r3, [fp, #-28]
        cmp     r3, #99
        movgt   r3, #0
        movle   r3, #1
        and     r3, r3, #255
        cmp     r3, #0
        bne     .L7
```




```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        path[i][j] = rand();
```

Esettanulmány

- Ciklusmag:



Esettanulmány

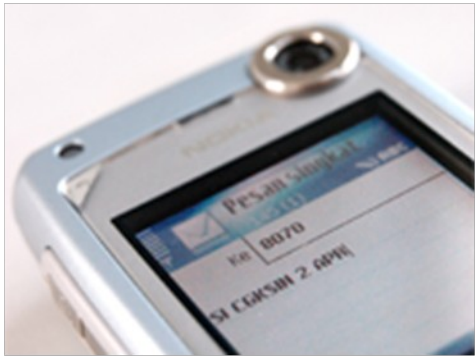
- Fordító meg sem kísérelte regiszterbe tenni a változókat
- Minden változó meghivatkozása memóriaműveletttel jár
- Láthatóak a RISC jellegzetességek:
  - Sokkal hosszabb, körülményesebb kód
    pl. ciklusváltozó növelése:

```
ldr     r3, [fp, #-28]
add     r3, r3, #1
str     r3, [fp, #-28]
```

vs.

```
add     DWORD PTR [esp+40004], 1
```

  - Egyszerű memóriacímzési mód
  - Predikátumok használata



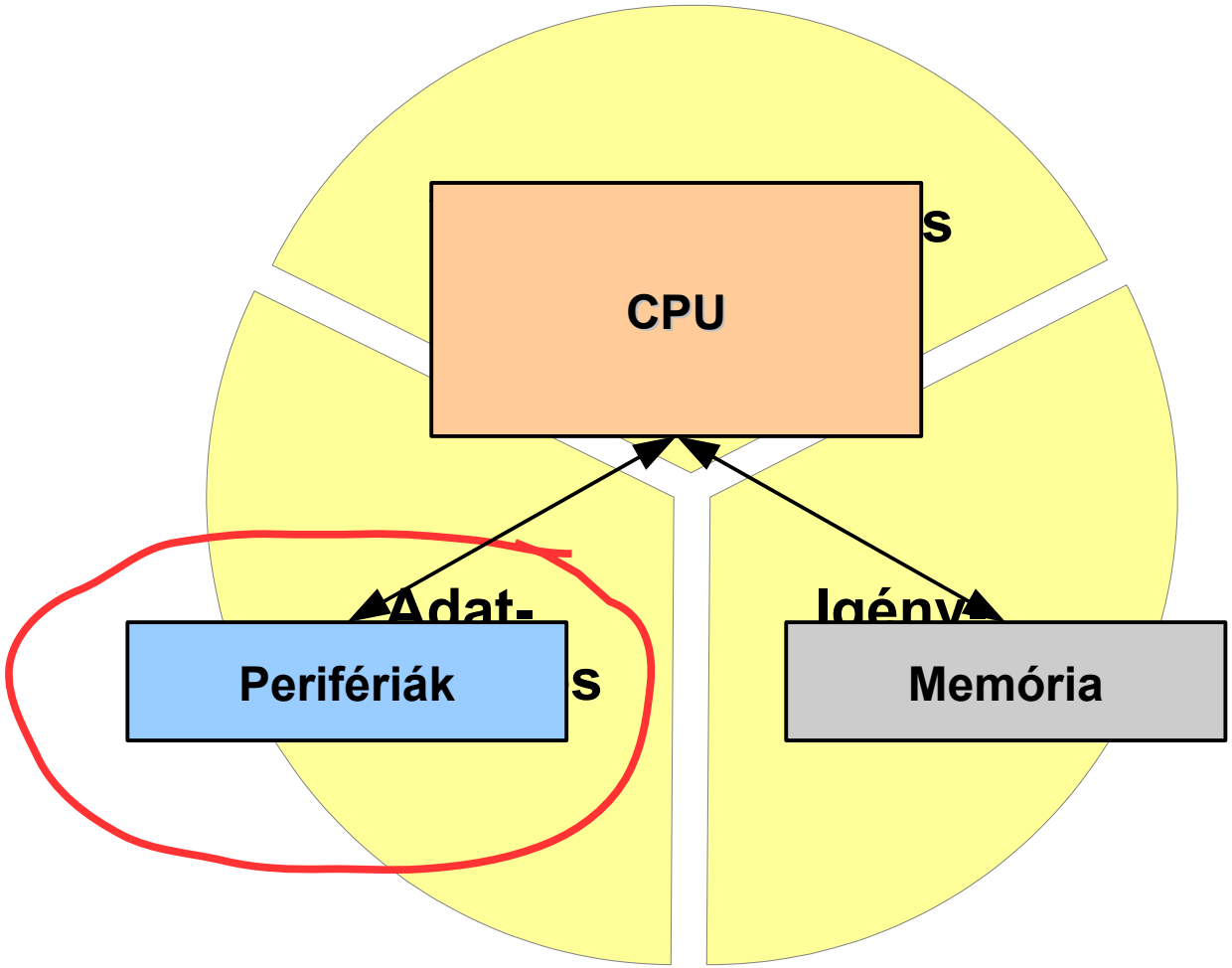
Számítógép Architektúrák

Perifériakezelés

Horváth Gábor
docens
BME Híradástechnikai Tanszék
ghorvath@hit.bme.hu

2012. szeptember 26.
Budapest

Hol tartunk?



Perifériák

- Bemeneti/Kimeneti
- Késleltetéskritikus és/vagy sávszélesség-igényes

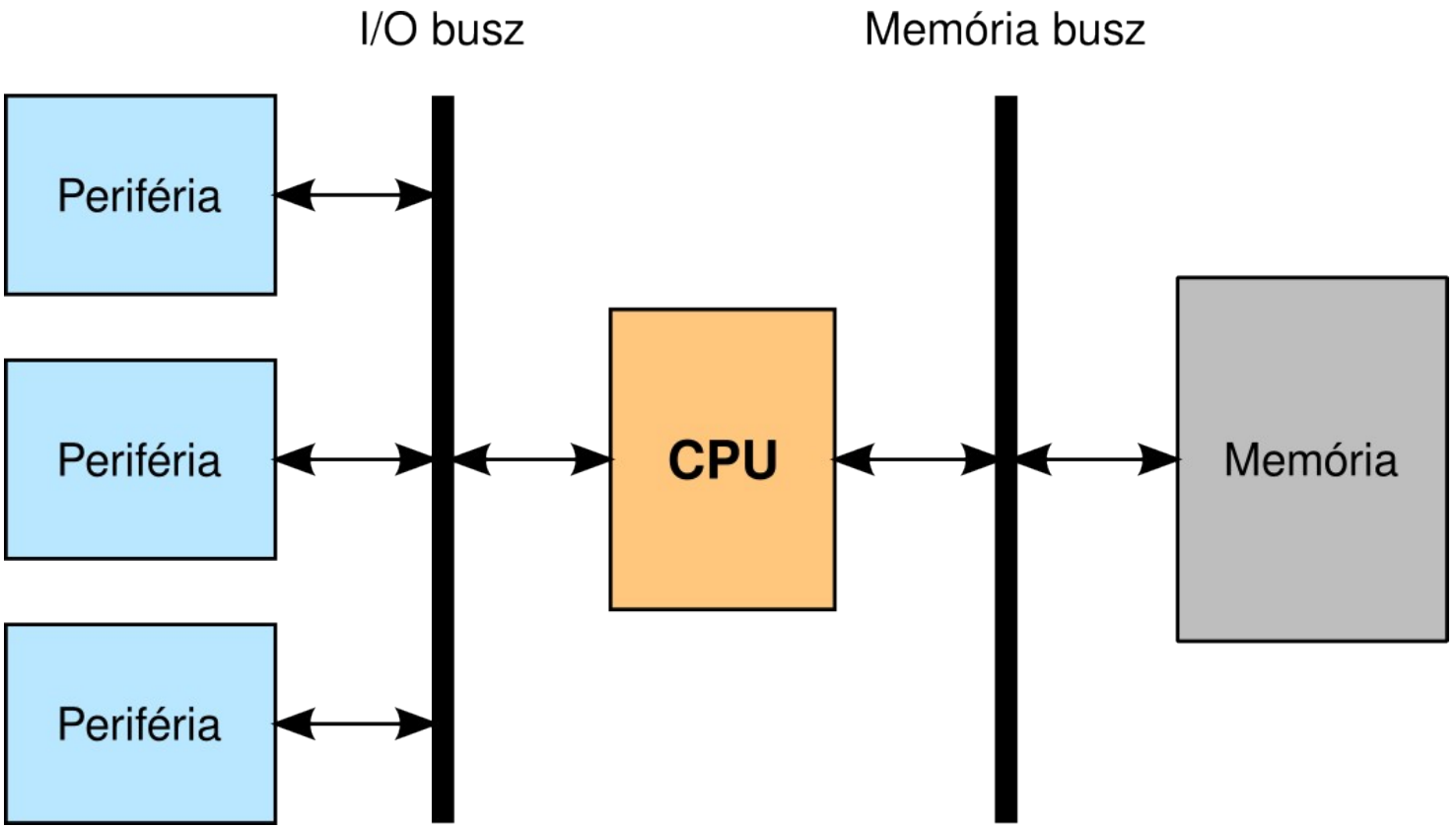
	Partner	Irány	Adatforgalom
Billentyűzet	Humán	Bemeneti	kb. 100 byte/s
Egér	Humán	Bemeneti	kb. 200 byte/s
Hangkártya	Humán	Kimeneti	kb. 96 kB/s
Printer	Humán	Kimeneti	kb. 200 kB/s
Grafikus megjelenítő	Humán	Kimeneti	kb. 500 MB/s
Modem	Gép	Ki/Be	2-8 kB/s
Ethernet hálózati interfész	Gép	Ki/Be	kb. 12.5 MB/s
Diszk (HDD)	Gép	Ki/Be	kb. 50 MB/s
GPS	Gép	Bemeneti	kb. 100 byte/s

Perifériák

- Kérdések:
  - Hogy támogatja a perifériakezelést a processzor?
  - Mi van, ha a periféria túl gyors? Vagy túl lassú?
  - Honnan tudja a processzor, hogy a perifériának közölnivalója van?
  - Hogy lehet tehermentesíteni a processzort?
  - Hogy kössük hozzá a perifériákat a processzorhoz?

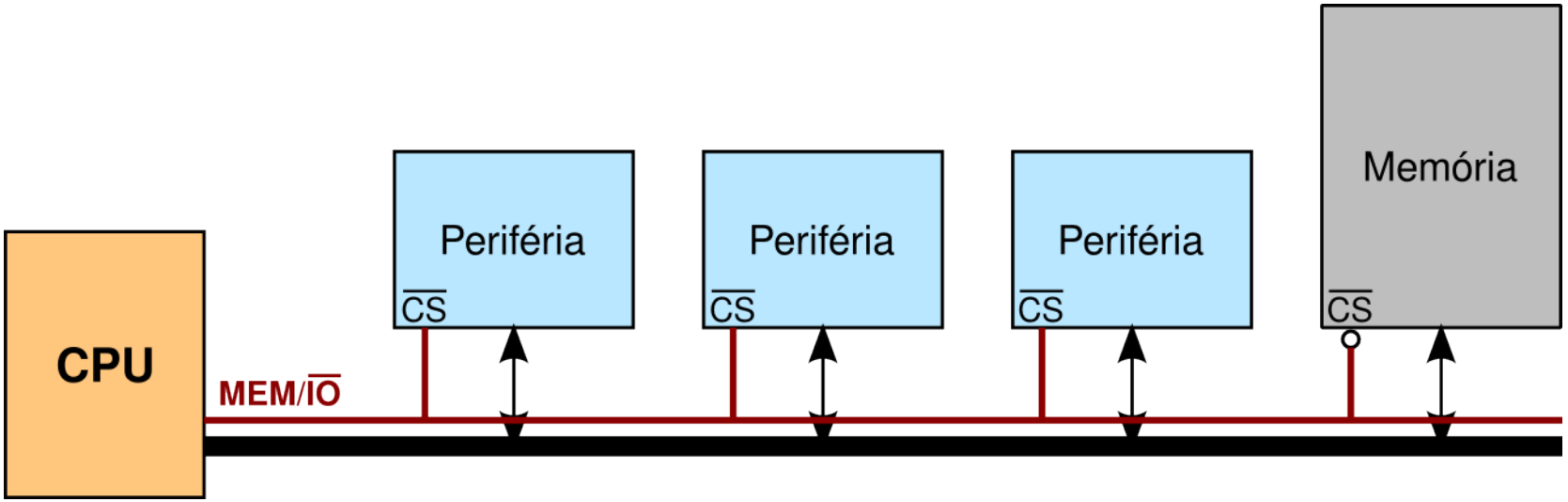
CPU perifériakezelési támogatása

1) Külön busz és külön utasítások



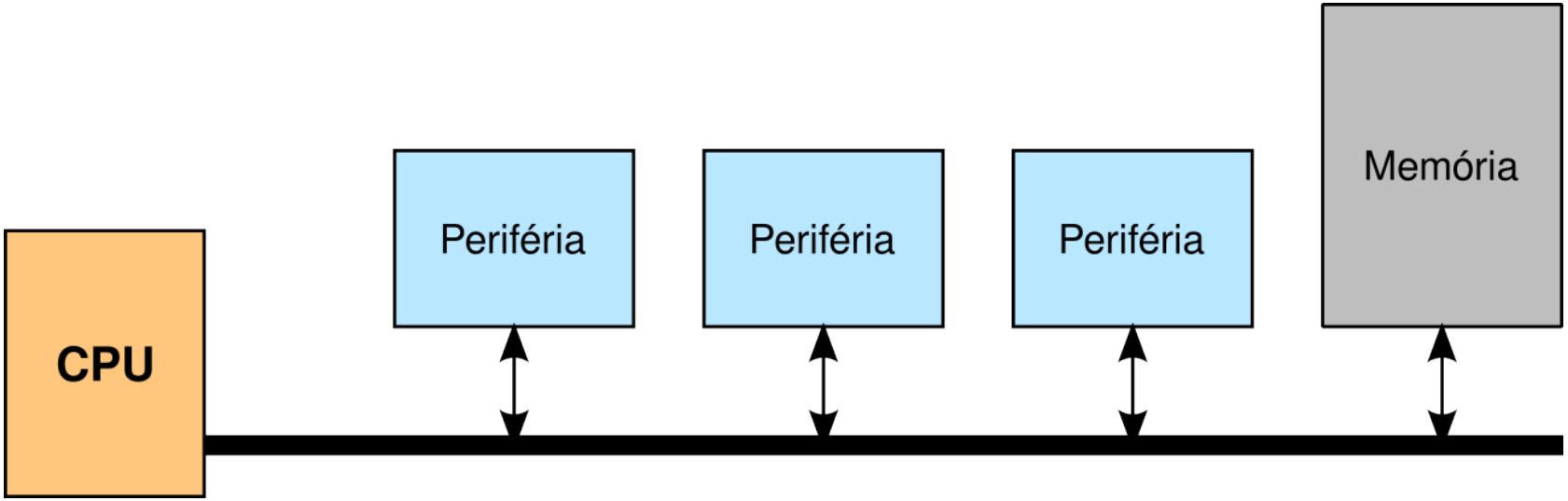
CPU perifériakezelési támogatása

2) Multiplexált busz és külön utasítások



CPU perifériakezelési támogatása

3) Közös busz, utasításokban sincs megkülönböztetés:
→ Memóriára leképzett perifériakezelés



Memóriára leképzett perifériakezelés

- Ott is terjed, ahol van külön I/O utasítás és címtér
- Előnyök:
  - Nincs szükség I/O utasításokra → Egyszerűbb processzor (RISC megoldások tipikusan ilyenek)
  - A memóriaműveletek rugalmasságával lehet perifériát kezelni (pl. változatos címzési módok)

Forgalomszabályozás

- Alapprobléma:
  - Készen áll-e mind a periféria, mind a CPU az adatátvitelre?

1) Feltétel nélküli adatátvitel

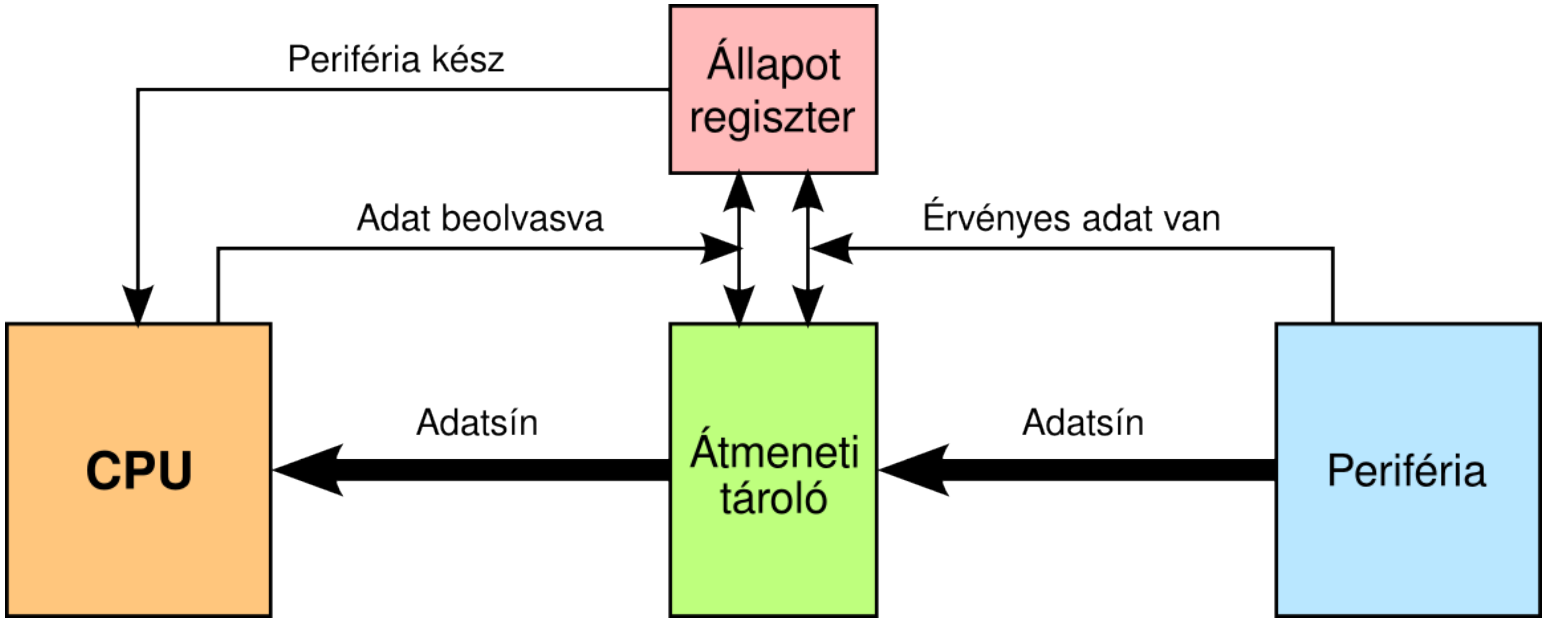
- Nincs forgalomszabályozás
- Egyik fél sem tudja közöli a másikkal, hogy kész az átvitelre
- Kétféle probléma léphet fel:
  - Adat egymásra-futás (küldő gyors, fogadó lassú)
  - Adathiány (küldő lassú, fogadó gyors)
- Példa: kapcsoló leolvasás, LED kigyújtás, ...

Forgalomszabályozás

2) Egyoldali feltételes adatátvitel

- Az egyik oldal sebessége nem befolyásolható (a másiké igen)
- Állapotregiszter: van-e érvényes adat?
- Példa: hangbemenet, hálózati kártya
- Példa: olvasás művelet

  → az állapotregiszterrel az adathiány elkerülhető!

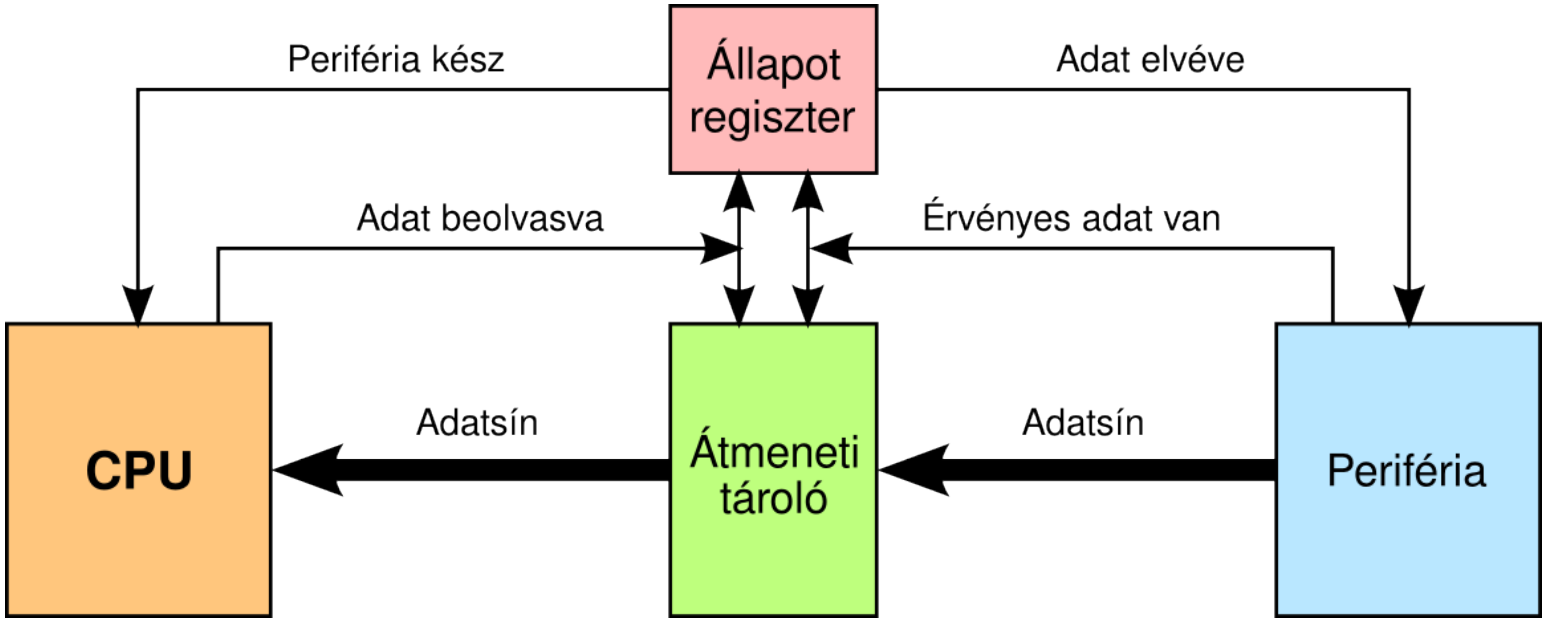


Forgalomszabályozás

3) Kétoldali feltételes adatátvitel

- Mindkét oldal sebessége befolyásolható
- Állapotregiszter: van-e érvényes adat?
- Példa: olvasás művelet

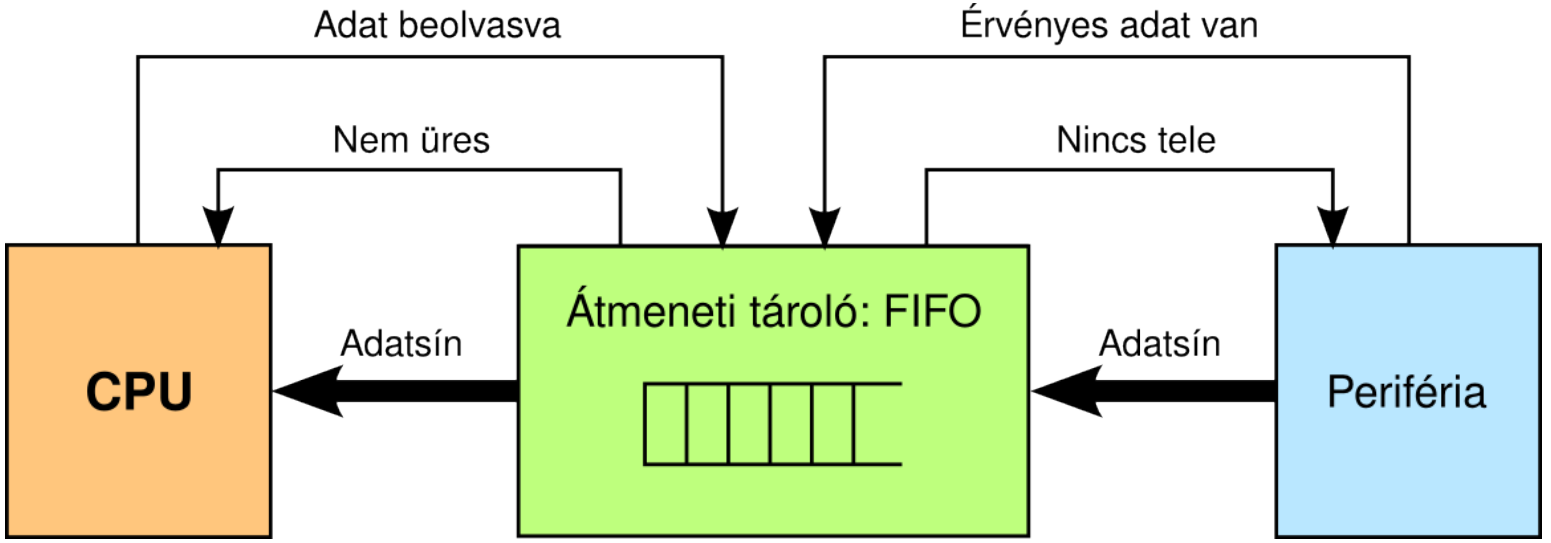
  → az állapotregiszterrel az adathiány és az adat egymásra-futás is elkerülhető!



Forgalomszabályozás

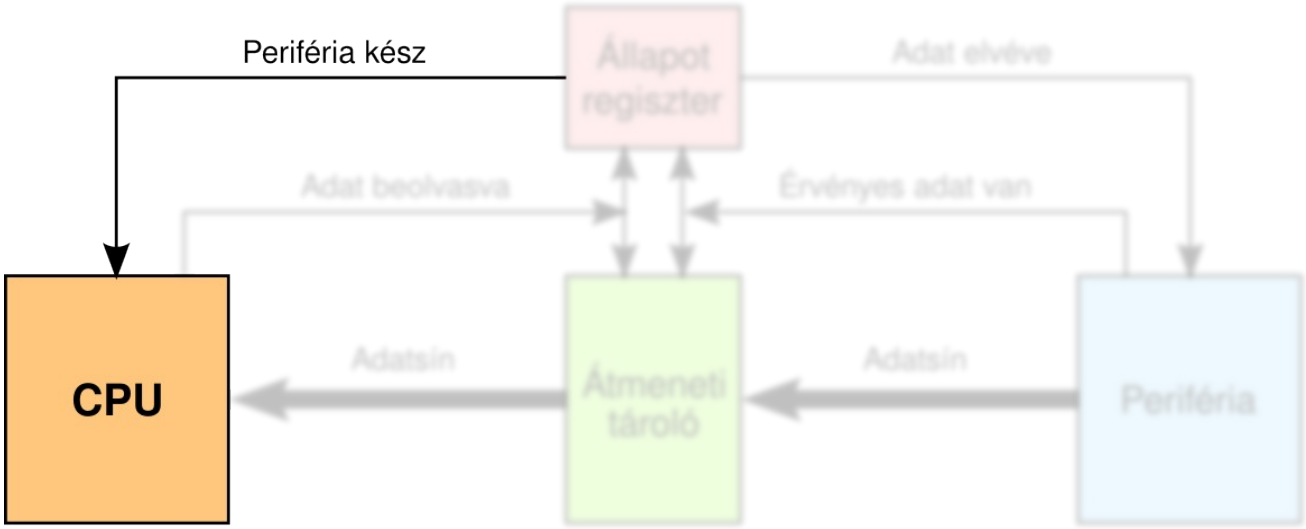
4) Feltételes adatátvitel FIFO tárolóval

- Mindkét oldal sebessége befolyásolható
- Feleknek nem kell mindig bevárniuk egymást
- Akkor jó, ha:
  - Ingadozó az adatforgalom
  - Ingadozó a felek rendelkezésre állása



Periféria jelzéseinek feldolgozása

- Honnan tudja a processzor, hogy jelzett a periféria?



- Polling: a jelzés periodikus ellenőrzése a futó programban
- Interrupt: a jelzés hatására a futó program megszakítása és a jelzés lekezelése

Polling

- Kritikus kérdés: a lekérdezések gyakorisága
  - Ha a periféria sebességéhez képest
    - Túl gyakori: CPU túl sok időt fecsérel felesleges lekérdezgetésre
    - Túl ritka: a processzor lemarad az eseményről
- Példák (CPU: 1 GHz, 1 lekérdezés 600 órajel)
  - Egér:
    - másodpercenként 30-szor kérdezzük le
    - 30 poll/s * 600 órajel/poll = 18000 órajel/s
    - CPU: 10^9 órajel/s → 18000/10^9 = 0.0018%, OK.
  - Diszk:
    - Interfész: 100 MB/s, 512 byte/blokk
    - Polling periódus: 100*10^6 byte/s / 512 byte/blokk = 195313 poll/s
    - 195313 poll/s * 600 órajel/poll = 117187500 órajel/s
    - CPU: 10^9 órajel/s → 117187500/10^9 = 11.72%
    - Megengedhetetlenül nagy: egyetlen periféria egyetlen jelzése!

Interrupt

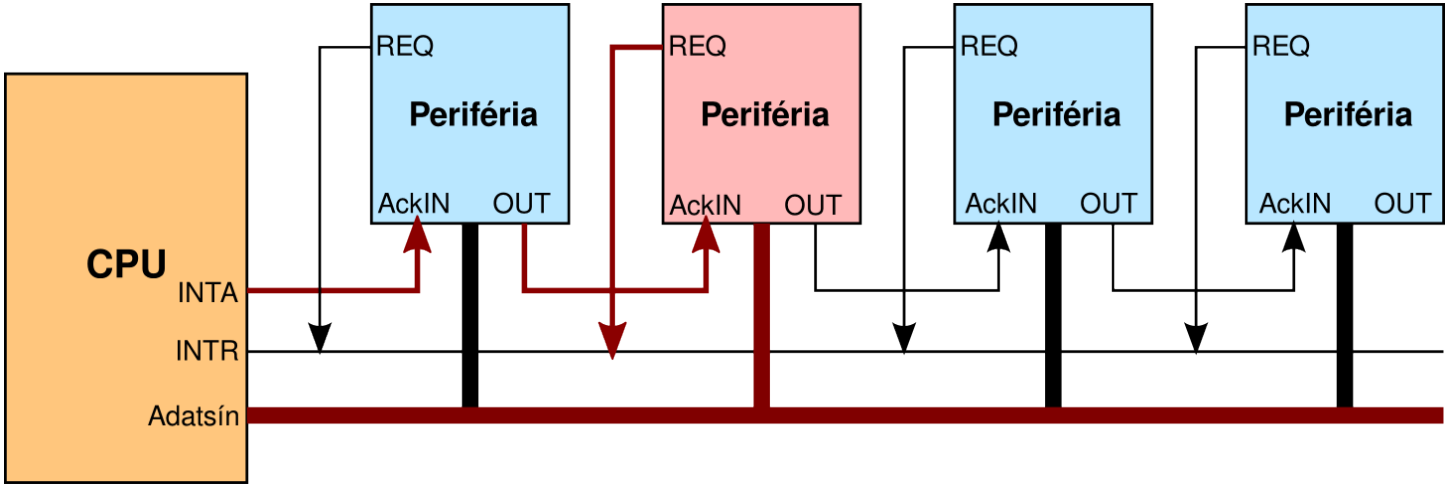
- A jelzést a CPU interrupt rendszerén keresztül kezeljük le
- A CPU-nak csak annyit kell a perifériával törődni, amennyi az adatok átvételéhez kell (ha az inaktív, semennyit)
- Példa:
  - Diszk
    - A diszk az idő 10%-ában aktív
    - Interrupt feldolgozási idő: 600 órajel
    - Tényleges adatátviteli idő: 100 órajel
    - Interrupt feldolgozásra fordított idő: 0.1*(100*10^6 byte/s / 512 byte/blokk * 600 órajel/interrupt) = 11718750 órajel/s
    - CPU: 10^9 órajel/s → 11718750/10^9 = 1.172%
    - Adatátviteli idő: 0.1*(100*10^6 byte/s / 512 byte/blokk * 100 órajel/átvitel) = 1953125 órajel/s
    - CPU: 10^9 órajel/s → 1953125/10^9 = 0.195%
    - Összesen: 1.172%+0.195% = 1.367%

Interrupt

- Gond: sokkal több a periféria, mint ahány interrupt lába a CPU-nak van
- Megoldások:
  - Polling: mindenki ugyanazon a vonalon jelez, az interrupt szubrutin mindenkit körbekérdez, hogy ki volt
  - Amikor a CPU elfogadja az interruptot, az interruptot kiváltó periféria az adatsínre teszi a számát
    - Ez a szám megmondja, hogy hányas számú interrupt szubrutinnak kell őt kiszolgálnia
    - A CPU-nak van egy táblázata: **interrupt vektor tábla**. Mutatókat tartalmaz interrupt kiszolgáló szubrutinokra.
    - Amilyen számot modott a periféria, a vektor tábla annyiadik szubrutinja fut le
- Na de mi van, ha többen is kérnek egyszerre interruptot?
  - Nem is ritka dolog
  - Miközben a CPU kiszolgál egy interruptot, több más interrupt is érkezhet
  - Valahogy sorba kell állítani a kéréseket
    - Daisy chaining-el
    - Interrupt vezérlővel

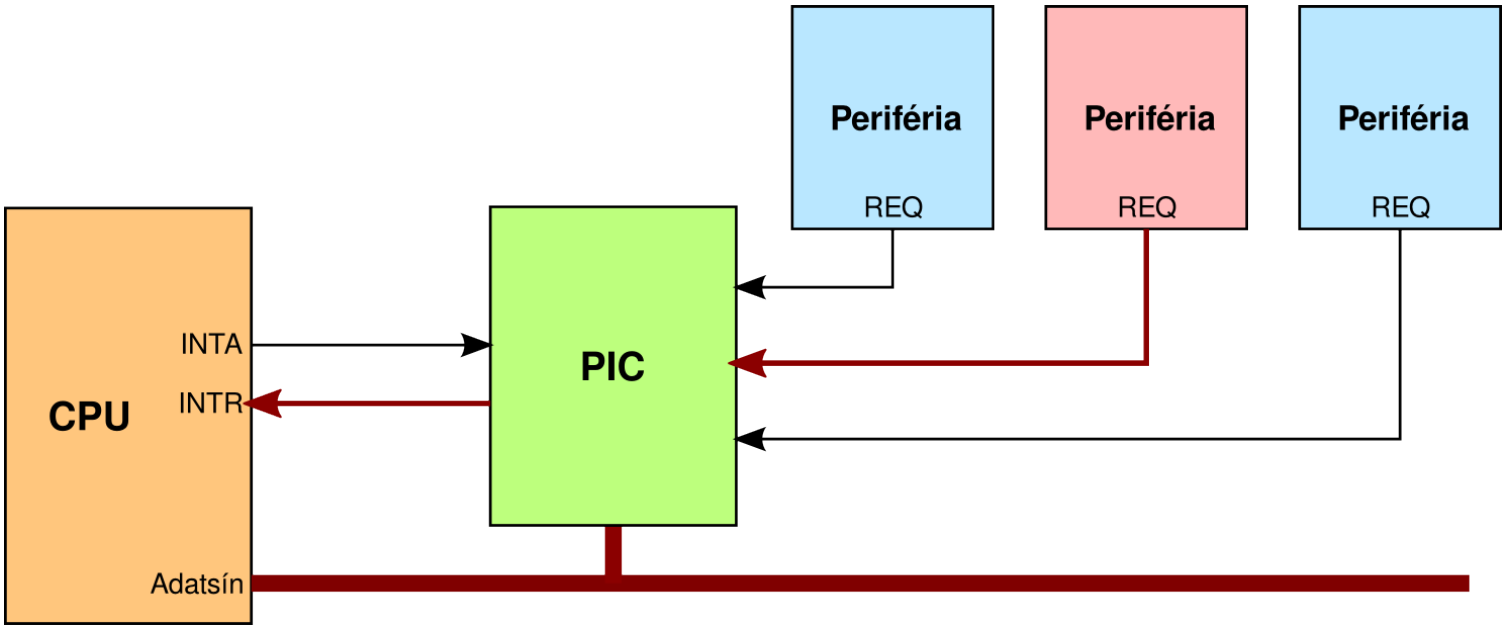
Interrupt

- Daisy chaining:
  - A CPU interrupt elfogadás jele sorban végighalad minden periférián
    - Aki nem kér megszakítást, továbbadja
    - Aki kér, megállítja a jelet, és az adatsínre teszi a számát
  - A sorrend a prioritást is meghatározza
    - A sorban hátul lévők éheznek



Interrupt

- Interrupt vezérlő (Programmable Interrupt Controller):
  - Több bemenete van
  - A PIC maga is egy periféria
  - A CPU (op. rendszer) perifériaműveletekkel beállíthatja,
    - hogy mi történjen, ha több interrupt kérés van egyszerre,
    - hogy mely kapcsolódó perifériák interruptjait engedélyezi



Interrupt

- Interrupt kezelése multiprocesszoros rendszerekben
  - Egyszerű megoldás: minden interrupt a boot processzorhoz fut be
  - Alternatív megoldás: fejlett interrupt vezérlővel (Intel: APIC, ARM: GIC, stb.)
    - Összetevők:
      - Minden processzornak van egy lokális interrupt vezérlője
      - Van egy rendszerszintű interrupt elosztó
    - Ha egy periféria interruptot jelez, a rendszerszintű elosztó továbbítja a megfelelő processzornak → **interrupt routing**
      - Az op. rendszer állítja be a rendszerszintű elosztót, hogy melyik interruptot melyik CPU lokális interrupt vezérlője kezelje
    - A lokális interrupt vezérlők egymásnak is küldhetnek interruptot
      - Ez egy lehetséges módja a processzorok egymás közötti kommunikációjának!

Interrupt

Amikor túl sok a jóból...

- Vannak perifériák, melyek túl sok interruptot generálnak
  - Pl. a gigabit sebességű hálózati interfészek
- A CPU megállás nélkül interruptot kezel, mással nem tud haladni
- Megoldás: **interrupt moderation**
  - A periféria több eseményt összevár, egy akkor jelez, egyszer
  - A CPU egy interrupt-tal lekezeli a periféria összegyűlt mondanivalóját

A processzor tehermentesítése

- Periféria → memória átvitel az eddigiek szerint:
  - Periféria → CPU, CPU → Memória
- Nem lehetne egy lépésben?
  - Gyorsabb lenne
  - CPU közben mást csinálhatna
- Megoldások:
  - DMA
  - I/O proceszor

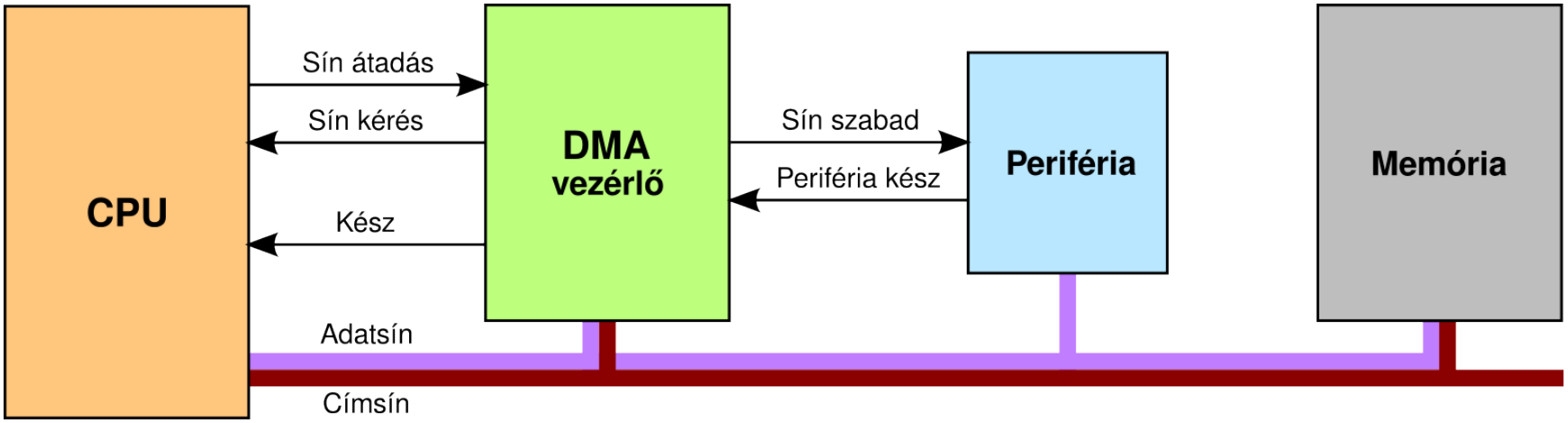
DMA

- Periféria → Memória a CPU megkerülésével

1. DMA vezérlő felprogramozása:
   - Melyik csatornán lévő perifériától,
   - melyik memória kezdőcímtől,
   - írjon-e, vagy olvasson,
   - hány adategységet kell átvinni.
2. DMA vezérlő koordinálja az adatátvitelt
   - Memóriabusz használati jogának megszerzése
   - Adatátvitel forgalomszabályozással

     → CPU szerepét játssza
3. A DMA vezérlő megszakítással jelzi, hogy végzett

DMA



- A CPU-nak csak a blokkos átvitel kezdeményezésekor és a teljes blokk átvitelének végén van dolga

DMA

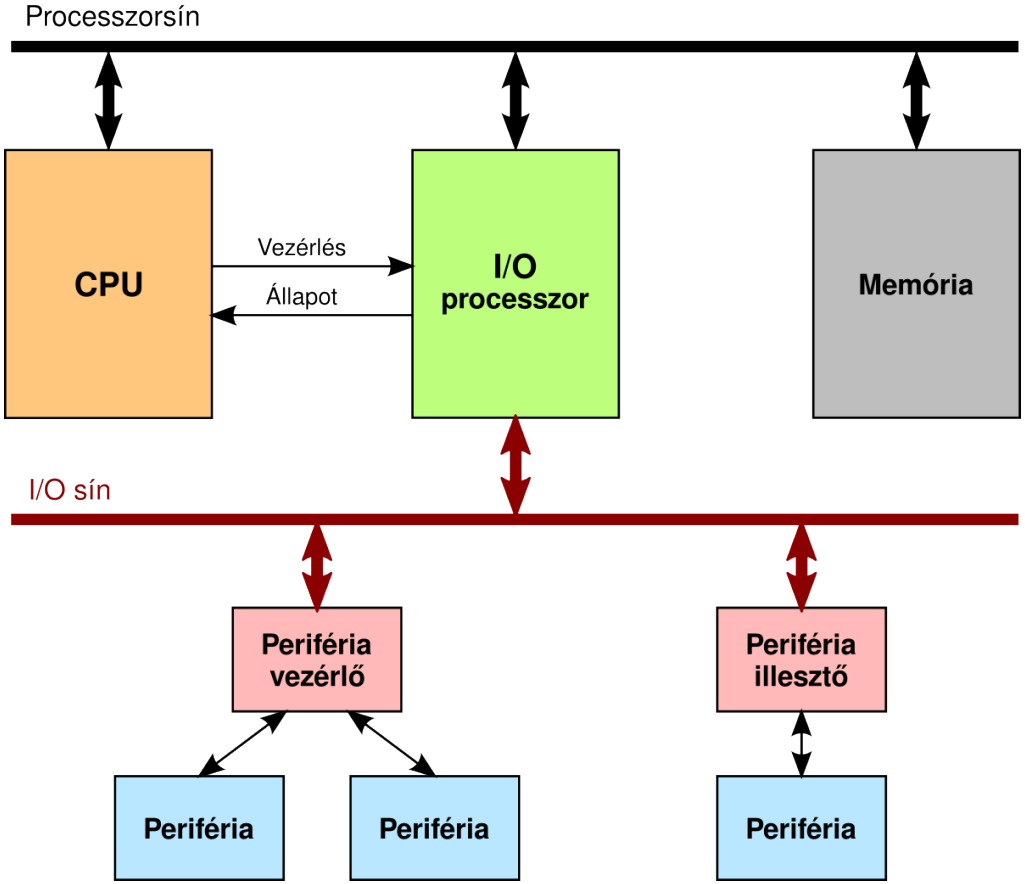
- Ez volt a rendszerszintű (*third party*) DMA vezérlő
- Ma már nem használják
- Amit ma használnak: *first party* DMA vezérlő
  - A perifériáknak saját DMA vezérlője van
  - Önállóan tudnak versenyezni a buszért
    - Egymással és a CPU-val
  - Aki nyer, az átviheti az adatait a memóriába

I/O processzor

- A rendszerszintű DMA koncepció továbbfejlesztése
- Az I/O processzor **saját utasításkészlettel** is rendelkezik
- I/O program:
  - Periféria és memóriaműveletek sorozata
  - Egyszerű adatfeldolgozási műveletek
    - CRC/paritás ellenőrzés
    - Tömörítés/kicsomagolás
    - Bájtsorrend konverzió
    - Stb.
- Végrehajtás:

  1. CPU odaadja az I/O processzornak az I/O program memóriabeli kezdőcímét
  2. Az I/O processzor kiolvassa, és sorról sorra végrehajtja
  3. A program végén megszakítással jelzi a CPU-nak, hogy kész

I/O processzor



- A CPU minden perifériaműveletet az I/O programmal ad meg
  → perifériafüggetlenség!
- A periféria illesztő/vezérlő feladata:
  Perifériaspecifikus viselkedés ↔ I/O sín protokoll fordítás

Összeköttetések

- Hogy kössük össze egymással a
  - processzort,
  - memóriát,
  - perifériákat?
- Eddig: memória & perifériák közvetlenül a CPU buszon
- Máshogy is lehet.

Busz vs. Pont-pont

- Pont-pont összeköttetések
  - Felek között dedikált csatorna
    - nincs osztozkodás, versengés, várakozás → gyorsabb
    - ahány eszköz, annyi pont-pont összeköttetés → drága
- Busz alapú összeköttetések
  - Osztott csatorna
    - osztott erőforrás → szűk keresztmetszet lehet (torlódás, egymás feltartása, stb.)
    - egy buszra kapcsolódik mindenki → olcsóbb
  - Az osztott erőforrással való gazdálkodás speciális megoldásokat igényel

Buszok szélessége

- Busz szélesség = a busz jelvezetékeinek a száma
- Széles busz:
  - Több bit vihető át egyidejűleg → gyorsabb lehet
  - Drágább
- Ellentmondás: a leggyorsabb buszok sorosak!
  - Széles busz → sok jelvezeték
  - A jelvezetékek hossza és elektromos tulajdonságai nem egyformák

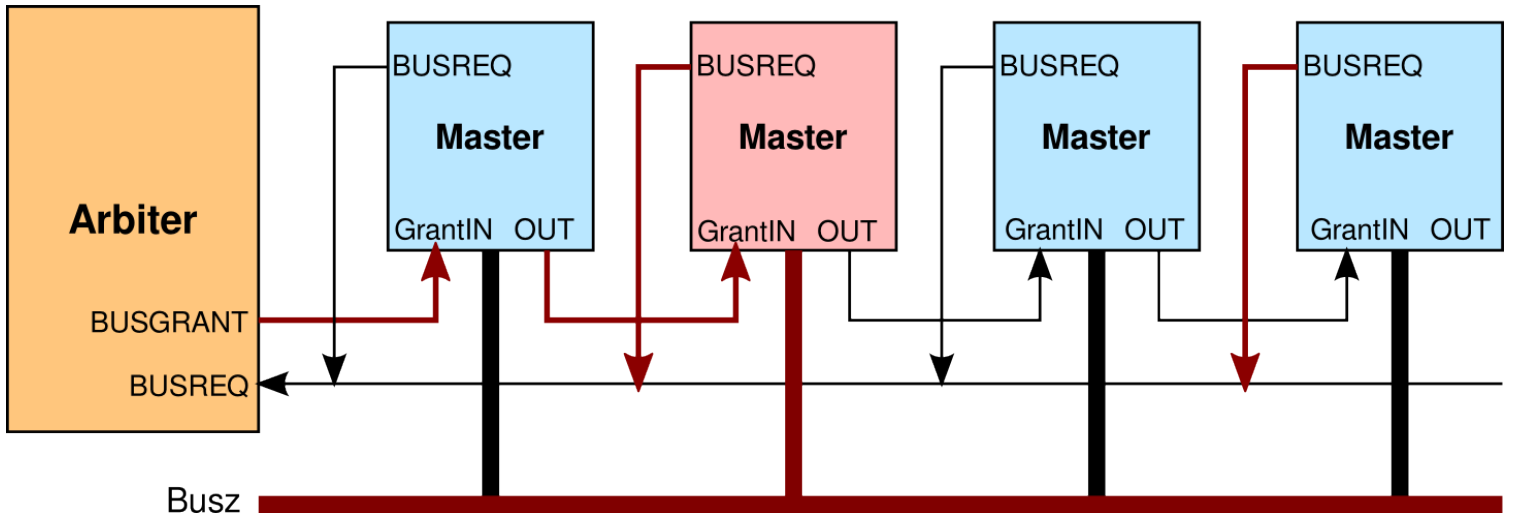
    → az egy időben küldött jelek egymáshoz képest csúszva érkeznek meg!
  - Akkor okoz gondot, ha a csúszás összemérhető az órajellel
- Trend: soros átvitelt mindenhova → nincs csúszás, ami korlátozza a sebességet

Arbitráció a buszon

- Szereplők a buszon
  - Bus Master: az az eszköz, ami képes a buszon önálló módon adatátvitel lebonyolítani
  - Bus Slave: nem képes
- A busz osztott erőforrás
  - Egyidejűleg több Master is jelenthet be igényt
  - Csak egy nyerhet
- Arbitráció
  - A buszért zajló verseny eredményének eldöntése

Centralizált arbitráció

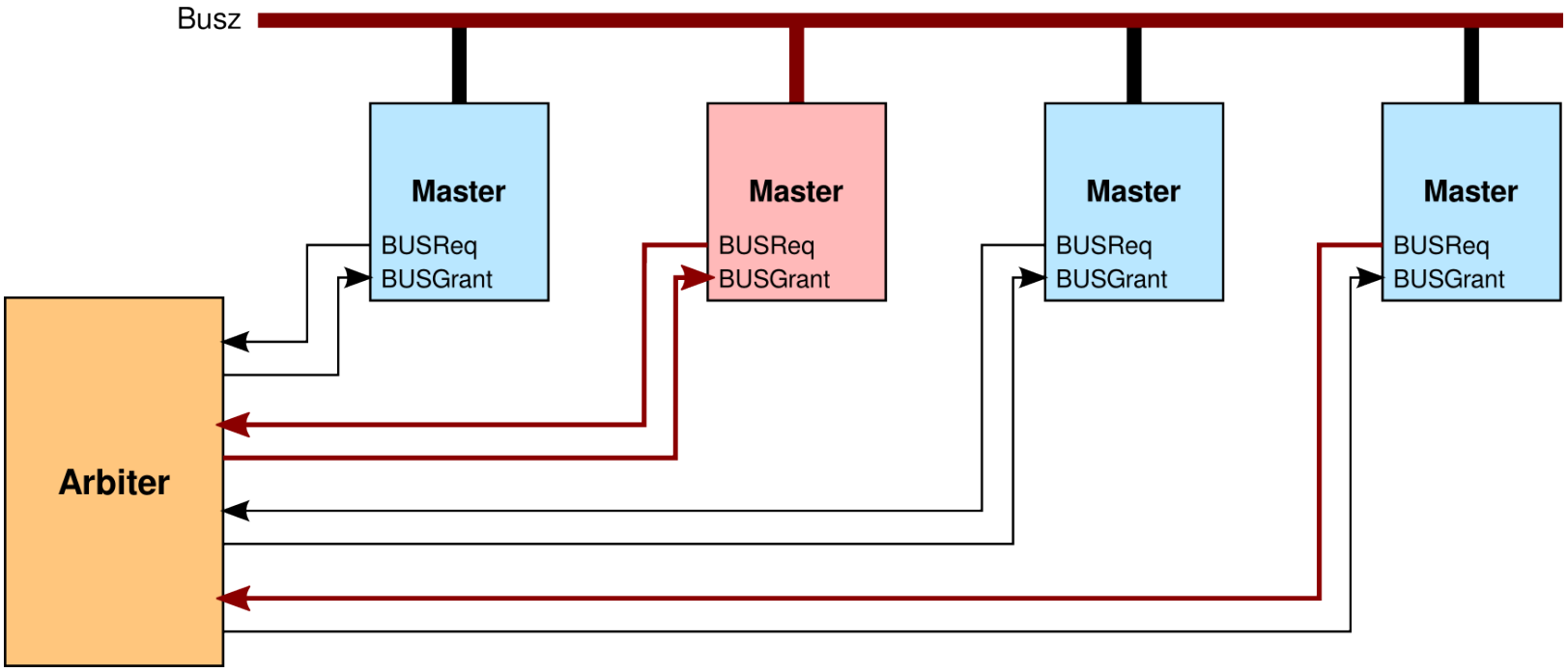
- Speciális egység: Arbiter
  - Eldönti, ki kapja a buszhasználat jogát
- Soros arbitráció: Daisy Chain



- Előny:
  - Könnyen bővíthető
- Hátrány:
  - Nem fair

Centralizált arbitráció

- Párhuzamos centralizált arbitráció
- Rugalmasabb priorizálás
  - Körbenforgó
  - Késleltetésérzékeny perifériák gyakrabban nyernek
  - Stb...

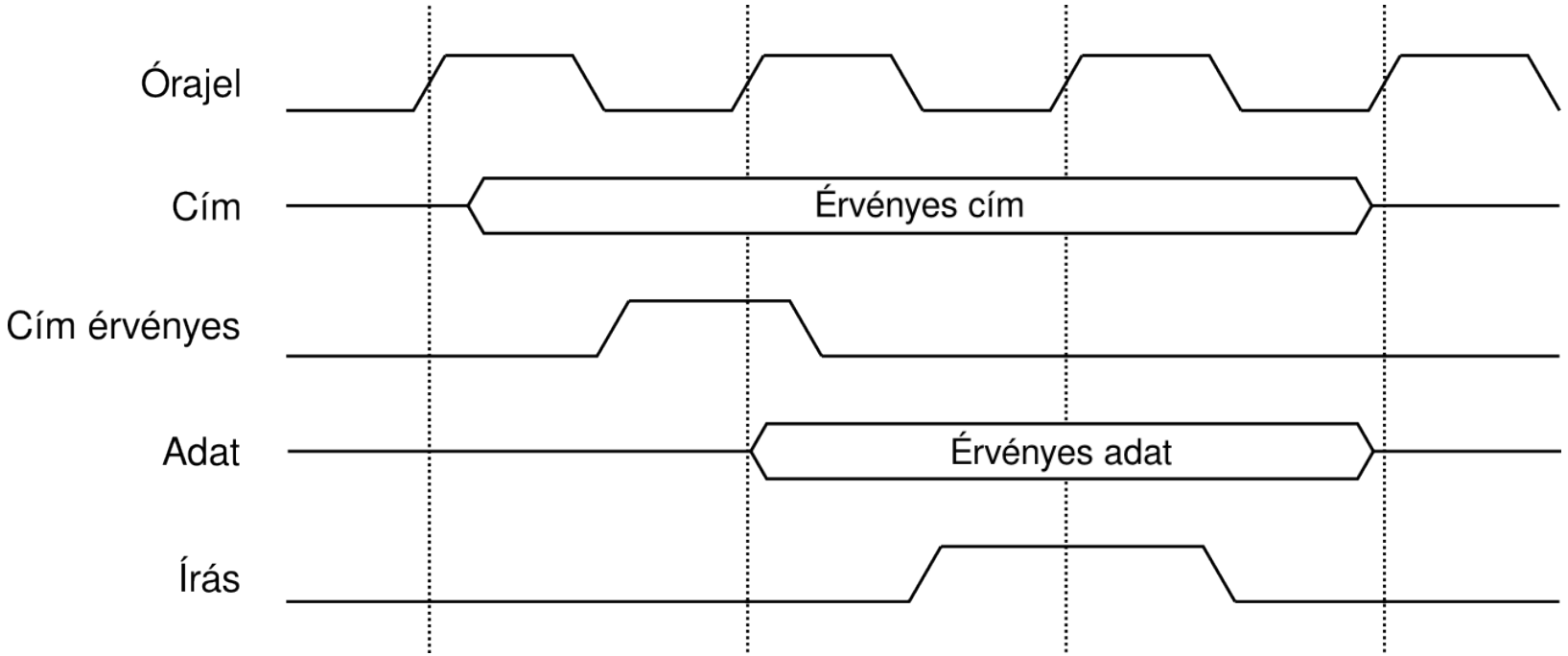


Elosztott arbitráció

- Nincs arbiter
- Önkiválasztó arbitráció (pl. SCSI):
  - Mindenki látja a többiek buszfoglalási kéréseit
  - Mindenki tisztában van a saját és a többiek prioritásával
  - A legnagyobb prioritásún kívül mindenki visszavonja igényét
- Ütközésdetektáláson alapuló busz használat
  - Nincs is arbitráció
  - Ha valaki használni akarja a buszt, rögtön neki is lát
  - Adatátvitel közben hallgatózik a buszon
  - Ha tisztán kivehető a saját „adása”, akkor jó
  - Ütközés esetén zagyvaságot hall → elhallgat, később újra próbálkozik
- Elosztott arbitráció előnye:
  - Nincs arbiter, ami, ha elromlik, megáll az élet

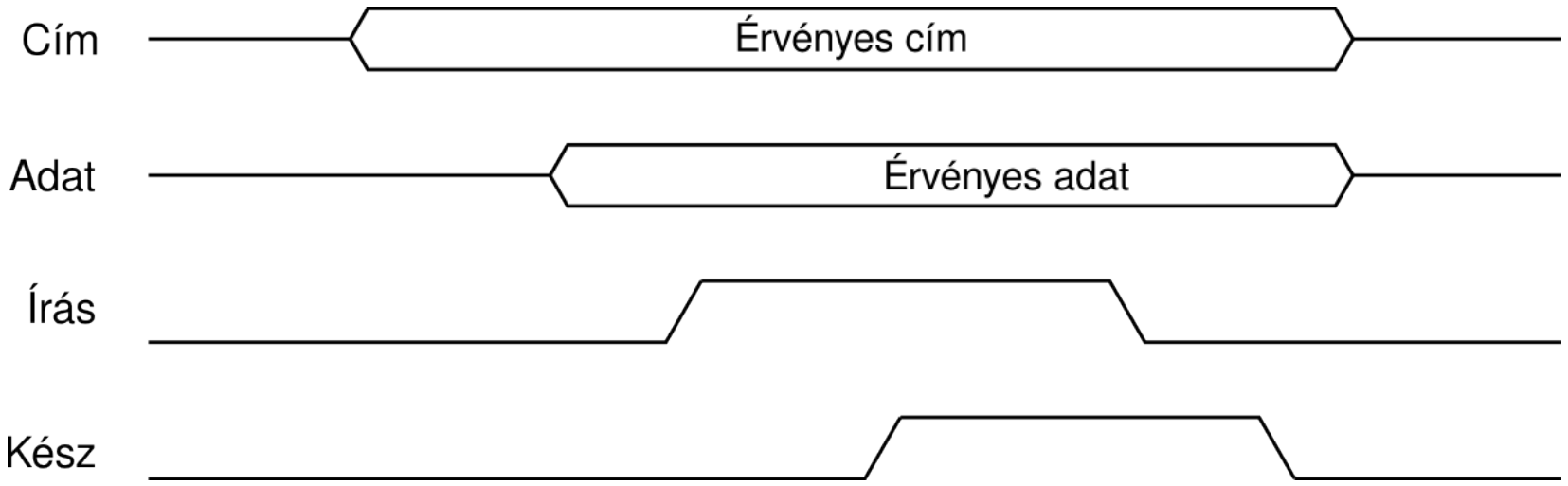
Időzítés a buszon

- Szinkron:
  - Közösen látott órajel
  - Adatok érvényességét az órajelhez kötik



Időzítés a buszon

- Aszinkron:
  - Nincs órajel
  - Adatok érvényességét a vezérlőjelek határozzák meg

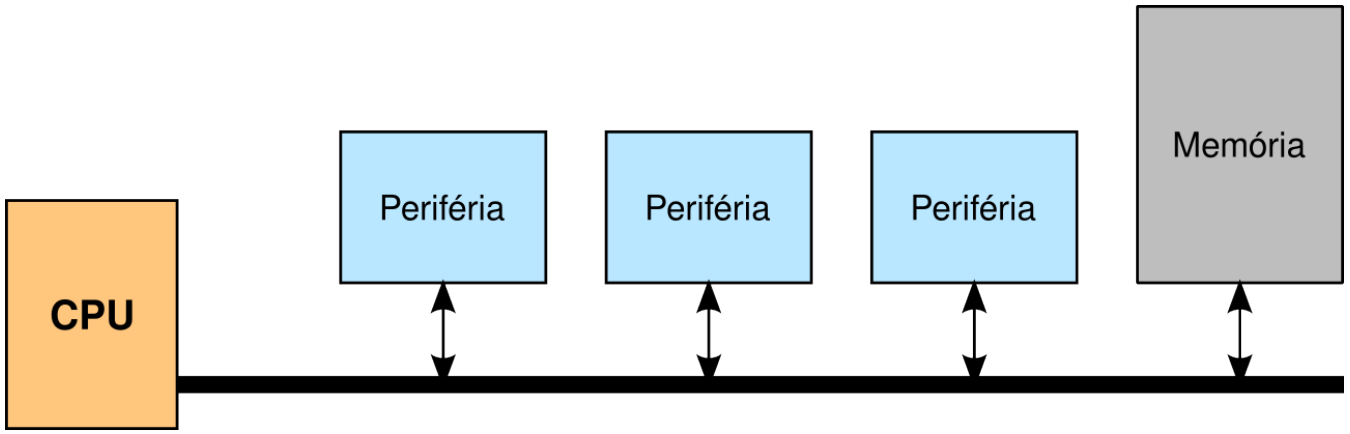


Példák

	PCI	SCSI	USB
Adategység	32/64 bit	8-32 bit	1 bit
Multiplexált?	Igen	Igen	Igen
Órajel	33/66 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	0.5 m	2.5 m	2 – 5 m

Egybuszos rendszerek

- CPU – memória – perifériák egyazon buszon

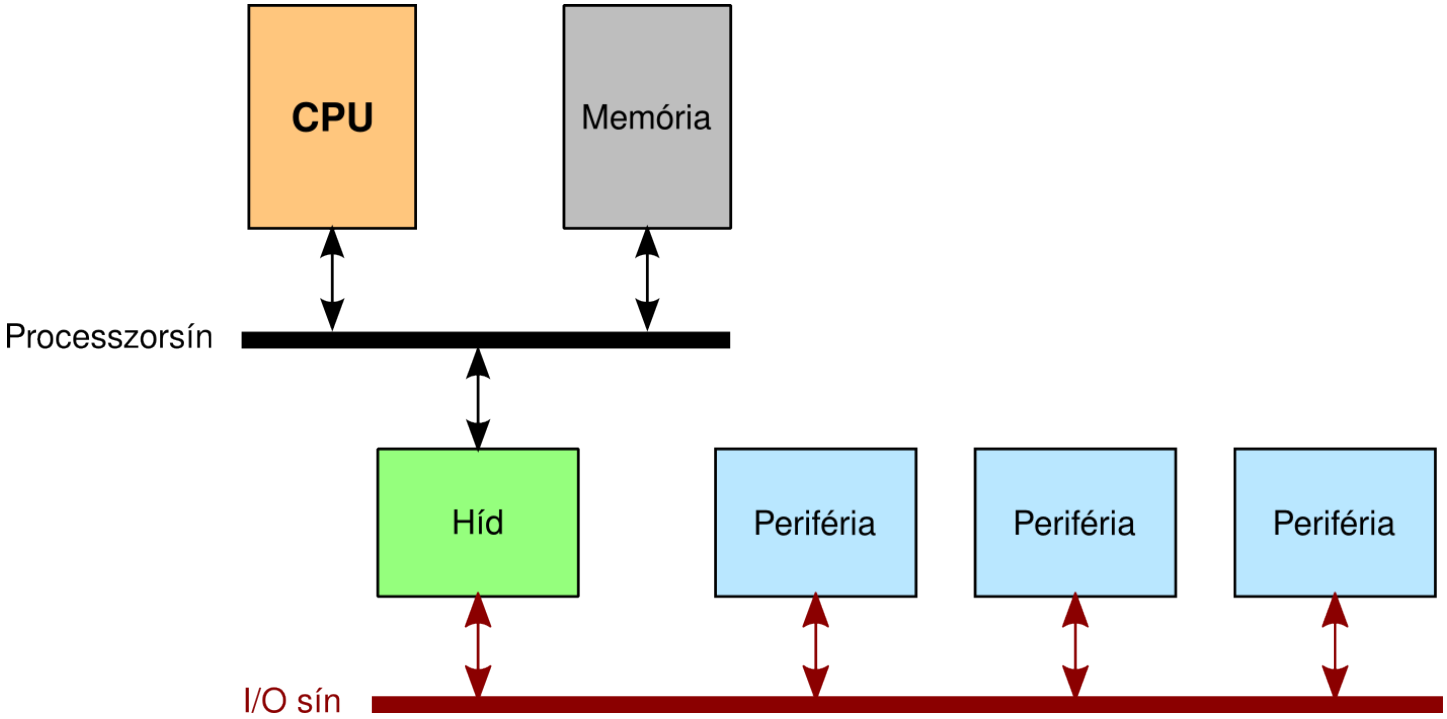


- Egyszerű megvalósítani
- Hátrány:
  - Magas CPU órajel → magas busz órajel → drágább perifériák → korlátozott fizikai távolság
  - A CPU órajele / busz órajele és a busz protokoll típustól függően változik

    → A perifériák állandó működési környezetet igényelnek
    - Nem szeretnénk kidobni a perifériákat, ha új CPU-t veszünk

Kétbuszos rendszerek

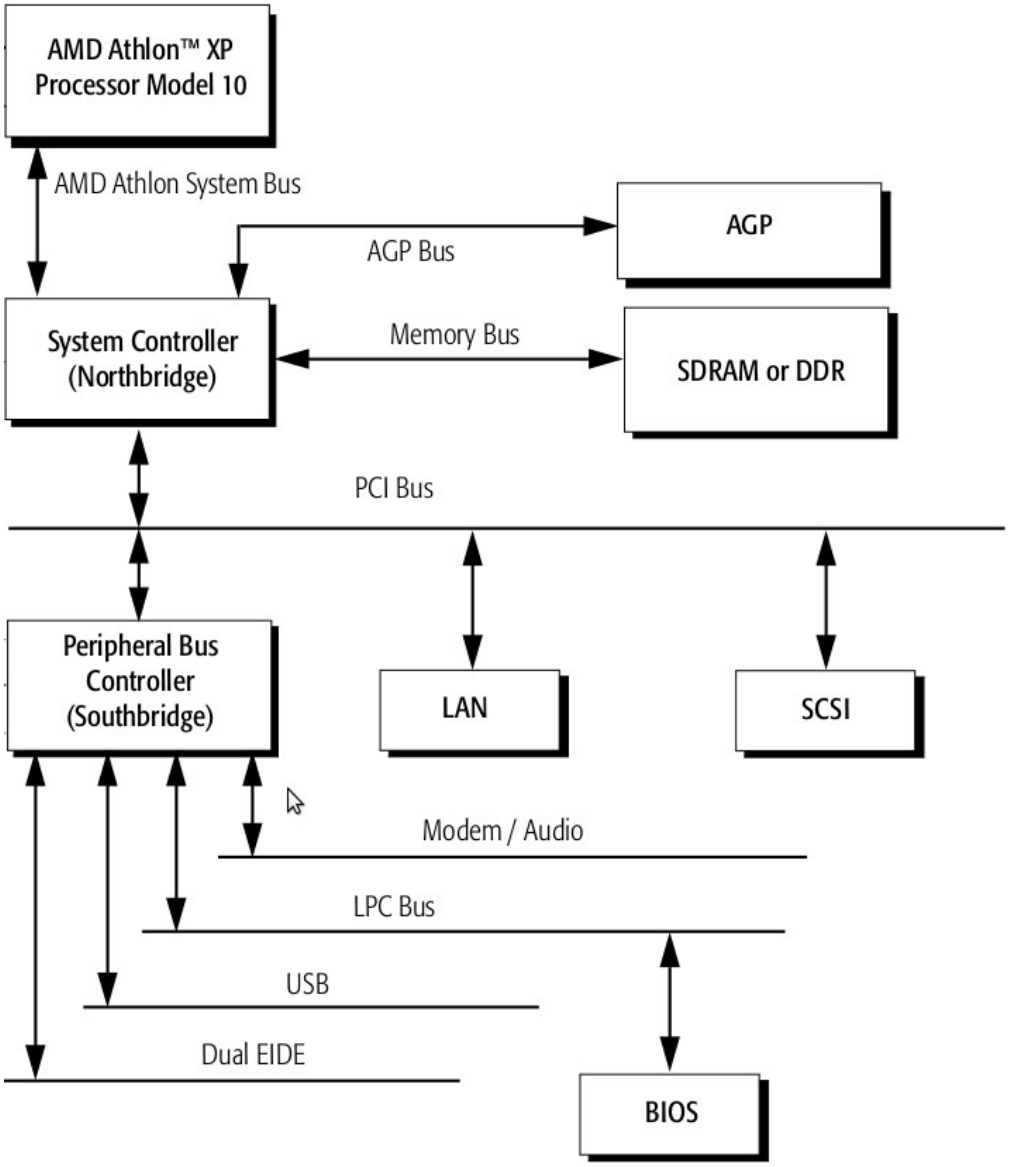
- Perifériák külön buszon, CPU-memória egy másik buszon
- Átjárás: híd



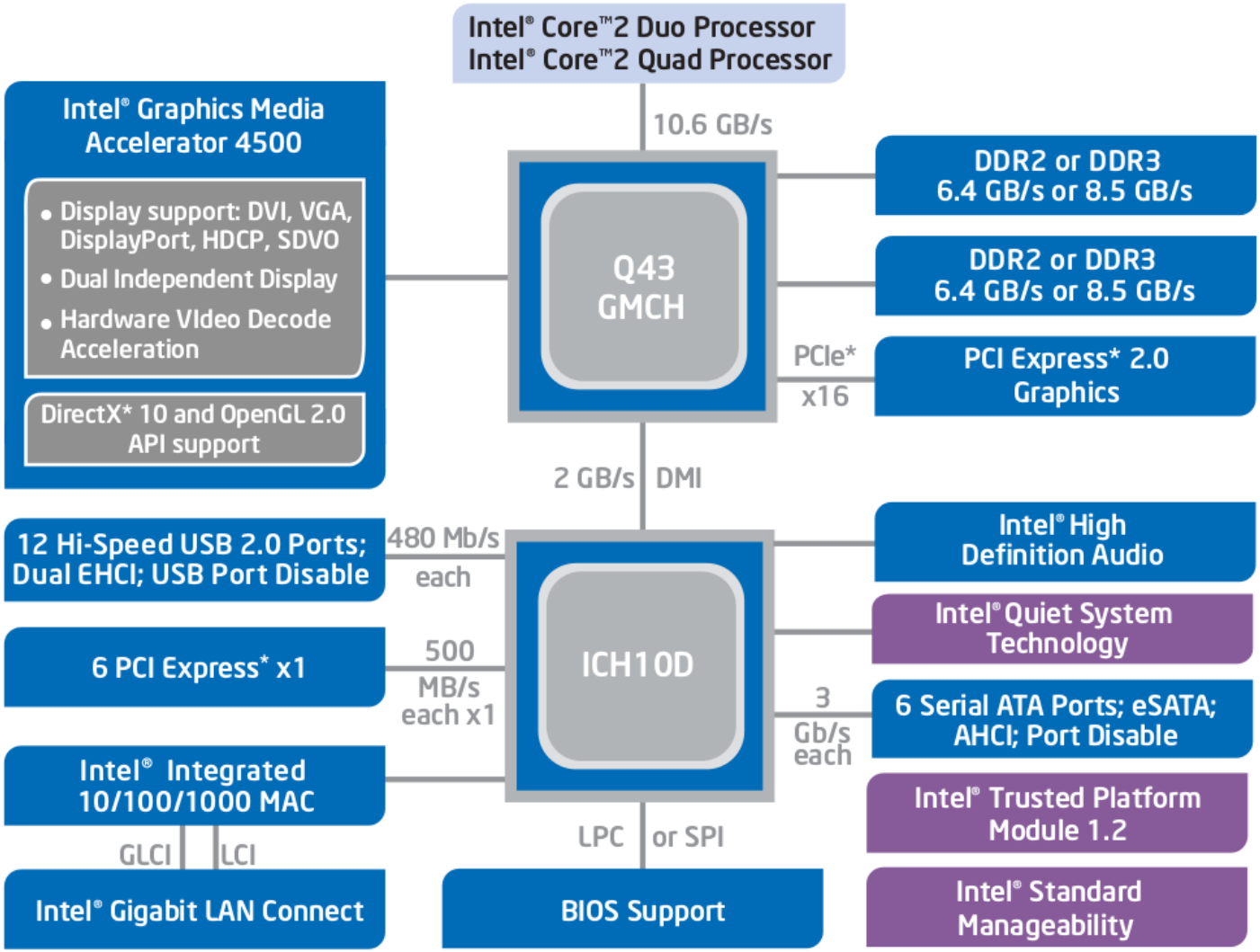
- Processzorsín: típusfüggő sebesség és protokoll
- I/O sín: állandó, szabványos sebesség és protokoll

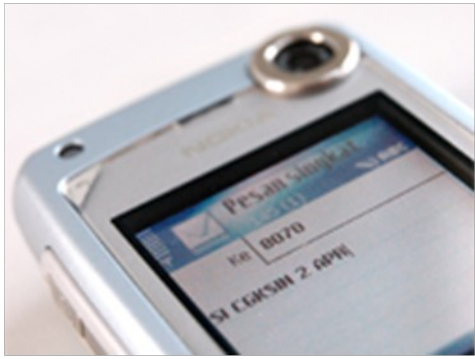
Híd alapú rendszerek

- Külön buszon:
  - CPU
  - Memória
  - Perifériák
- Processzorsín:
  - Típusfüggő
- Memóriasín:
  - Állandó, szabványos
- Perifériasín (PCI):
  - Állandó, szabványos



Híd alapú rendszerek





Számítógép Architektúrák

Háttértárak

Horváth Gábor
docens
BME Híradástechnikai Tanszék
ghorvath@hit.bme.hu

2012. október 1.
Budapest

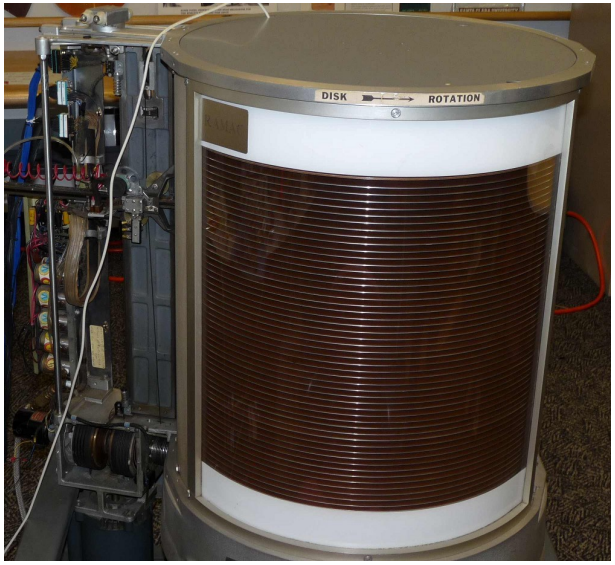
Háttértárak

- Minden számítógép részét képezik
- Teljesítményük a rendszer egészének teljesítményére kihat
- Fontos szerepet játszanak a virtuális tárkezelésben
- Amiről tanulunk:
  - HDD: merevlemez alapú háttértárak
  - SSD: félvezető alapú háttértárak
- Van még más is:
  - Optikai lemezek: sok, a HDD-nél látott megoldás visszaköszön
  - Pen-drive: alapja ugyanaz a flash memória, ami az SSD-ben van
  - Stb.

Adattárolás merevlemezen (HDD)

Merevlemezek

- Első merevlemez:
  - 1956, IBM (RAMAC 305)
  - Paraméterei:
    - Súlya: 1 tonna
    - 50 db 24 colos kétoldalas lemez
    - Két író/olvasó fej liftezik
    - 100 sáv/lemez
    - Elérési idő: 1s
    - Kapacitás: 5 millió
      7 bites karakter
- 2006: 1 col, 8 GB kapacitás

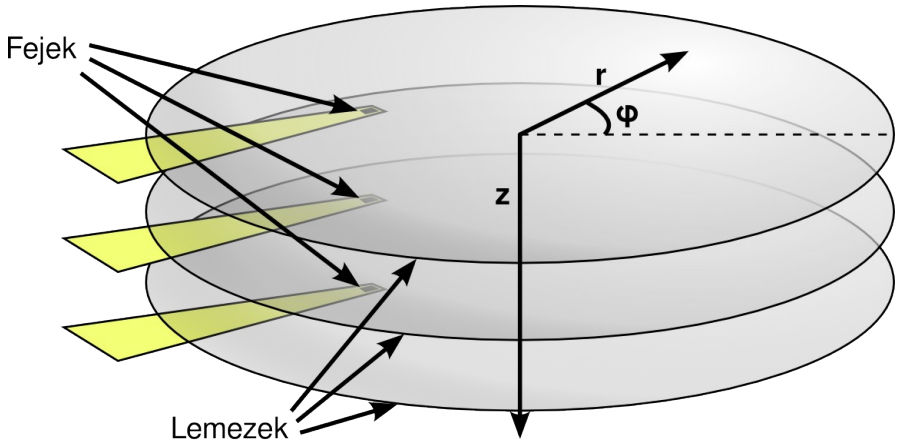


Forgólemezes adattárolás

- Első forgólemezes adattároló: *fonográf*
- A lemez sávjai mentén különféle elven tárolhatunk információt:
  - Mechanikai (fonográf)
  - Optikai (CD, DVD, stb.)
  - Ferromágneses (merevlemez)
- Forgólemezes adattároló részei:
  - **Lemez** (egy vagy két oldalas): ez az adattároló
  - **Fej**: a hordozóról leolvasott információt elektromos jellé alakítja

Forgólemezes adattárolás

- Adatok pozíciója:
  - Középponttól mért radiális távolság
  - Egy fix referenciaponthoz képesti szög
  - Az adathordozó felület (lemez, ill. a lemez melyik oldala)



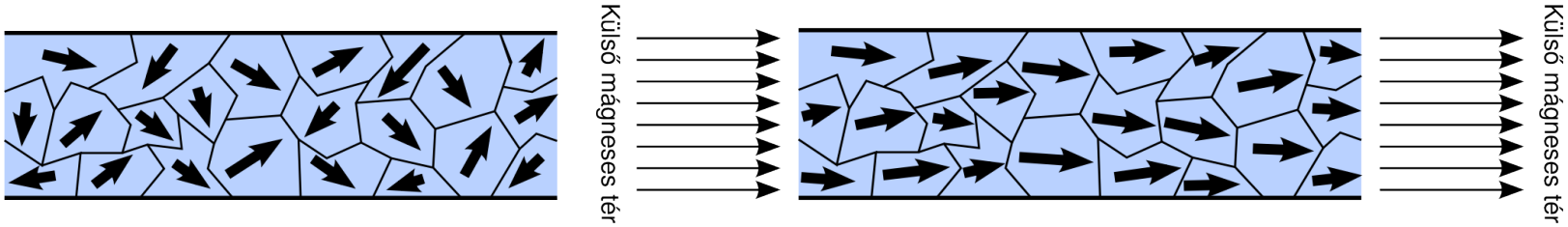
Forgólemezes adattárolás

- A fej pozícionálása:
  - Vagy a fej mozog az adat fölé
  - Vagy a lemez mozog úgy, hogy az adat a fej alá kerüljön
  - Mindkettő:
    - A fejet megfelelő radiális pozícióba viszik (*seek*)
    - A lemezt a szükséges szögnek megfelelően beforgatják
    - Hogy időt/energiát spóroljanak, a lemezt folyamatosan pörgetik

Mágneses adattárolás

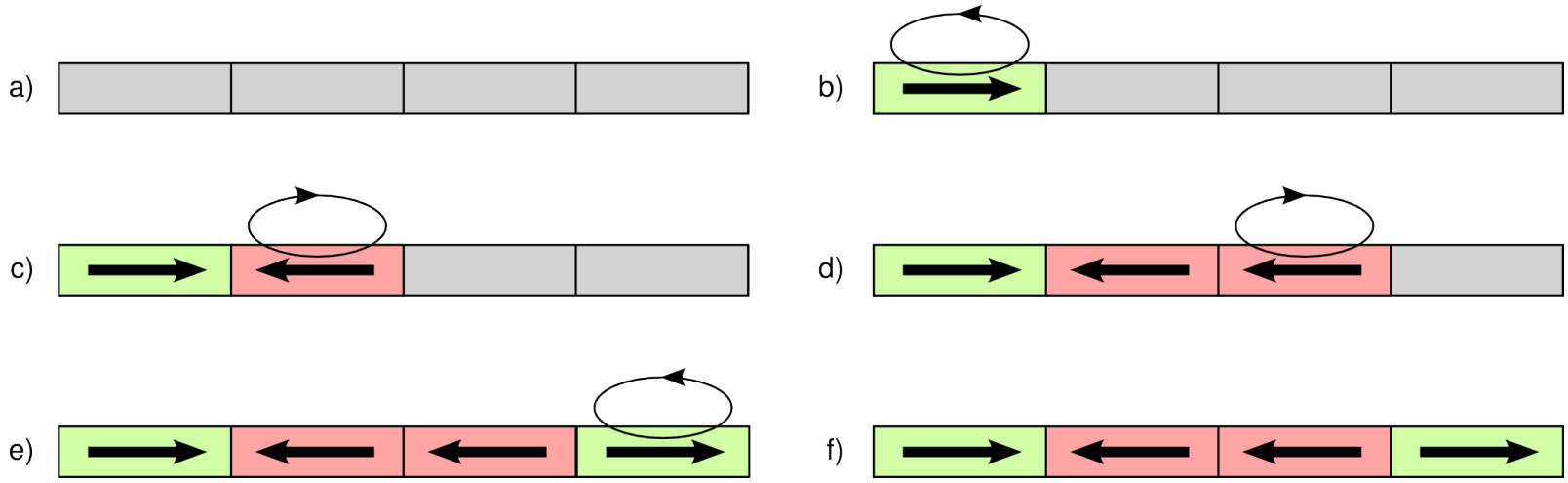
- Alapja: a ferromágneses anyagok viselkedése
  - Ilyen pl. a vas, a kobalt, a nikkel
  - Kis cellákból, *mágneses domének*ből áll
  - A mágneses doménen belül a mágneses momentum azonos

    → A doméneken belül erős mágneses tér van
  - Alapból semleges: a domének iránya véletlenszerű, összességében kioltják egymást
  - Külső mágneses tér hatására a domének a tér irányába állnak be
  - Nagyon stabil állapot!



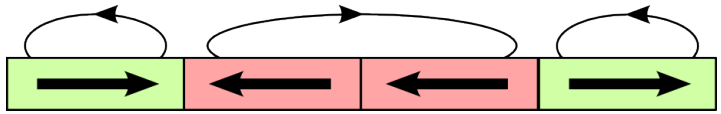
Adatok írása a mágneses felületre

- A fejet a kívánt radiális távolságba toljuk
- A lemezt a kívánt pozícióba forgatjuk
- A lemez egy kis része felett a fej egy lokális mágneses teret hoz létre
- A lemez lokálisan mágnesessé válik, és úgy is marad



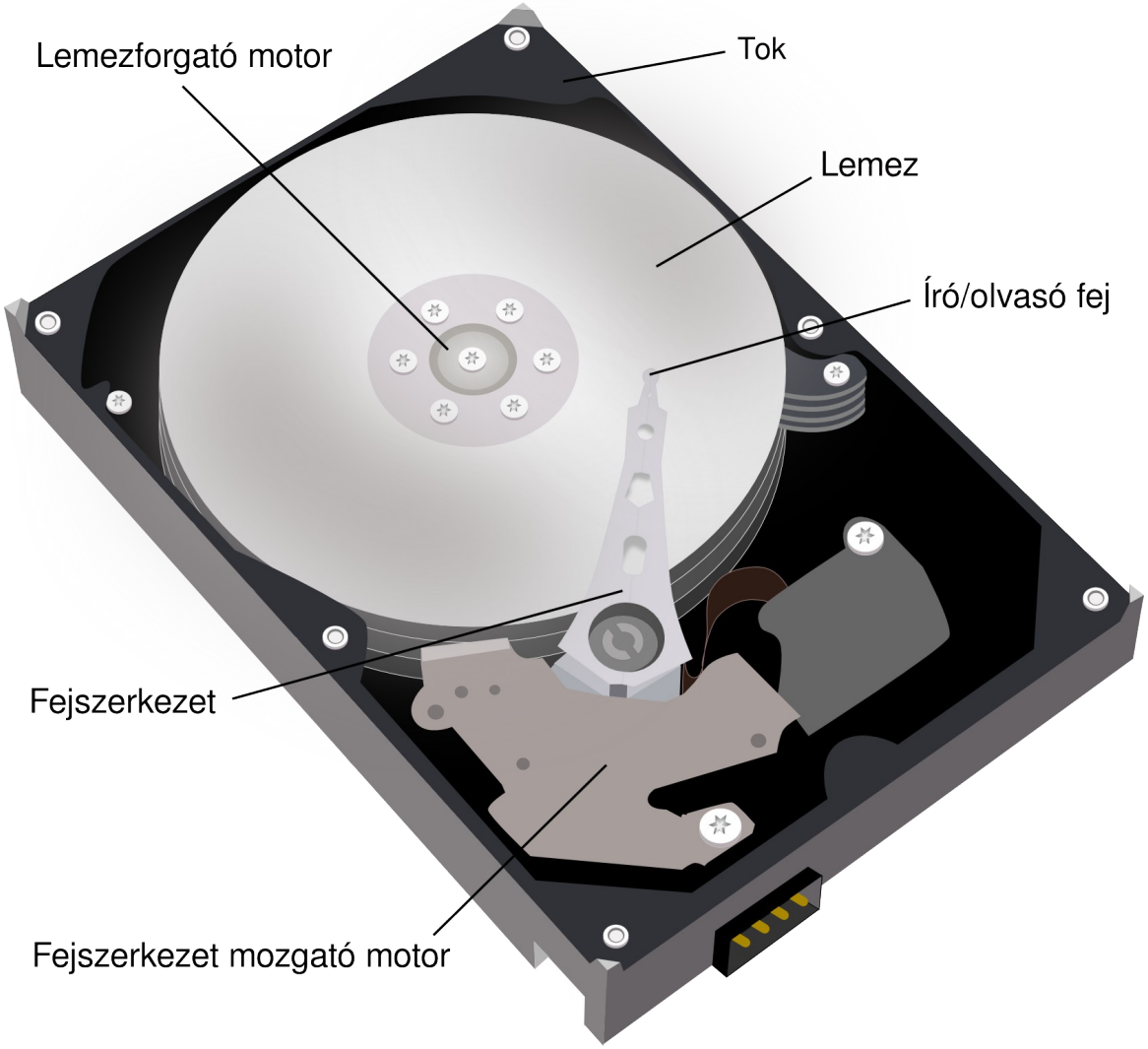
Adatok leolvasása a felületről

- A lemez által kibocsátott mágneses mezőt kell detektálni
- Lehetetlen.
- Amit lehet: a mágneses mező *megváltozását* detektálni
  - Mágneses mező változik: 1-es bit
  - Nem változik: 0-ás bit
- Példa: „101” bitsorozat:



- Következmény:
  - A biteket nem lehet egyesével módosítani!
  - Mert akkor az utána következő minden szakaszon mágneses teret kellene váltani
  - E helyett: nagyobb adategységet (szektort) vezetünk be
  - Írni/olvasni csak egy teljes szektort lehet

A merevlemezek felépítése

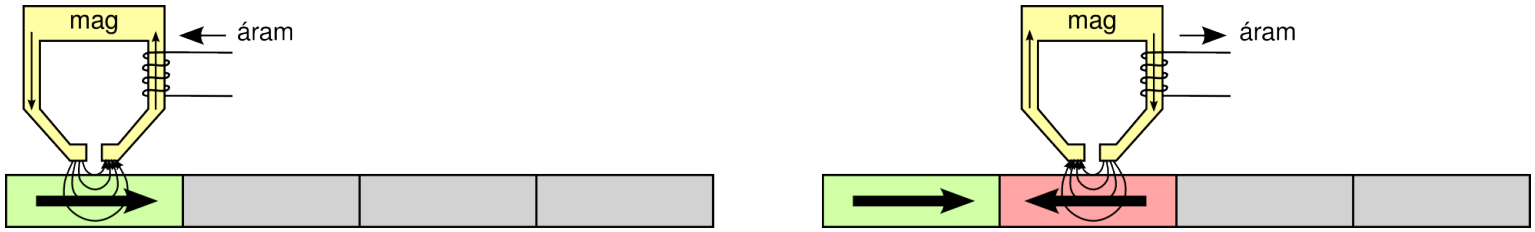


A merevlemezek felépítése

- **Lemezek**
  - Elvárások:
    - Legyen vékony (könnyű forgatás)
    - Legyen merev (ne torzuljon, ne rezonáljon)
    - Legyen sima (közel mehessen a fej)
    - Legyen az adathordozó réteg mágnesesen stabil
  - Több réteg:
    - Szubsztrát (alu, vagy újabban üveg ill. kerámia)
    - Vékony nikkel-foszfor réteg, amit nagyon simára lehet polírozni
    - kb. 25 nm vastag ferromágneses anyag (kobalt, vas-oxid, króm-dioxid, bárium-ferrit, stb.)
    - kb. 10 nm szén réteg, hogy az adathordozót védje
    - kb. 1 nm vastag kenőanyag
- **Motor**
  - Elvárások:
    - Nagy megbízhatóság (éljen túl akár 10.000 ki/be kapcsolást)
    - Minimális vibráció
    - Kis fogyasztás (a melegedés csökkenti a HDD élettartamát!)
    - Ütésállóság, kis zaj

A merevlemezek felépítése

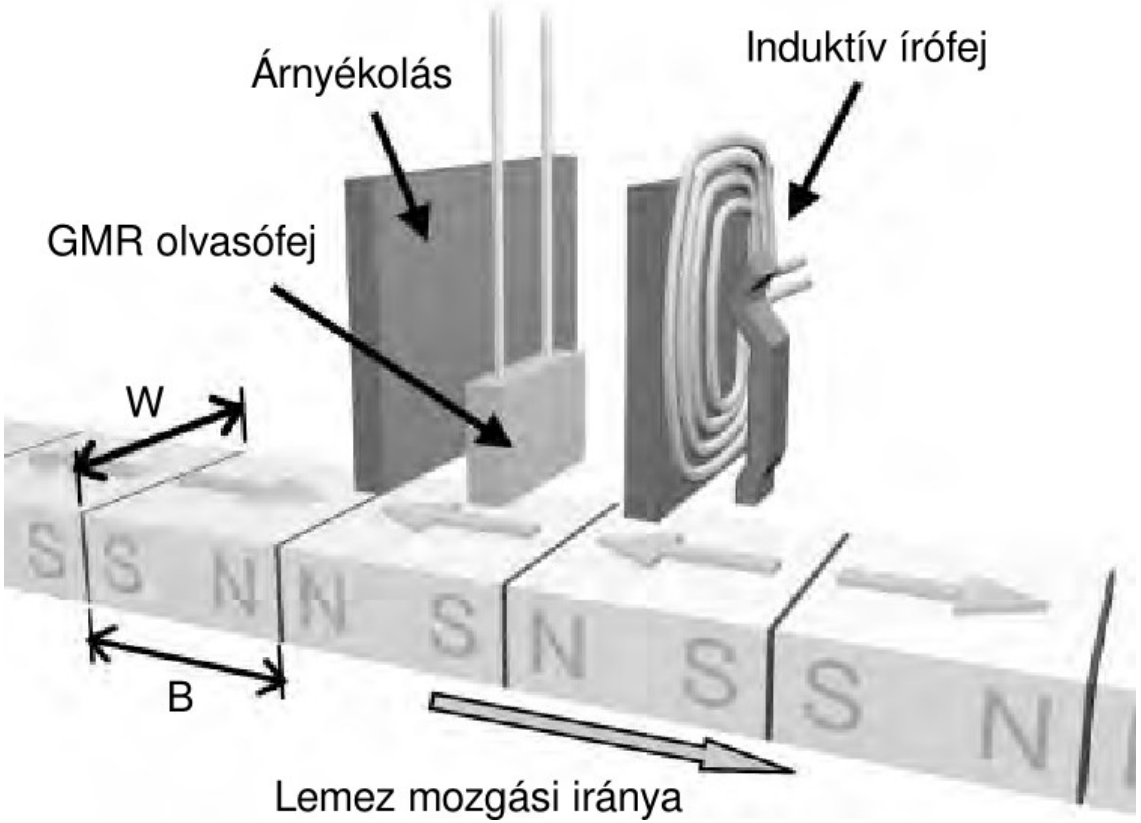
- **Író fej**
  - Induktív elven működik: gyűrű alakú mag, tekerccsel
  - A 80-as évek óta olcsón, hatékonyan gyártható



- **Olvasó fej**
  - A 90-es évekig az induktív írófejjel olvastak
    - Nem adtak rá áramot
    - A lemez mágneses mintázata indukálja az áramot
    - Az indukált áram iránya attól függ, hogy 0-1 vagy 1-0 átmenet volt-e
    - Gond: kicsi a tekercs, az indukált áram is nagyon kicsi
  - MR (magnetorezisztív) olvasófej:
    - Az MR fej ellenállása a mágneses mező változásától függ
    - Állandó áramat adunk rá, a rajta eső feszültséget mérjük
    - Továbbfejlesztése: GMR fej (1997 óta)

A merevlemezek felépítése

- GMR olvasó fej induktív író fejjel



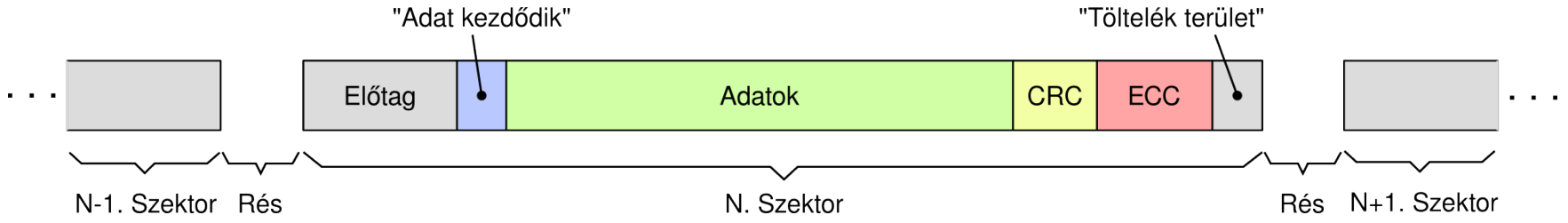
Az adatok szervezése

Adategységek

- Az olvasás jellegzetessége miatt csak komplett blokkokat írhatunk/olvashatunk
  - Fix méretű blokkok legyenek?
  - … vagy változó méretűek?
- **Count-Key-Data (CKD) rendszer**
  - Változó méretű objektumok: pl. egy egész fájl egyben
  - Egy tárolt objektumhoz tartozik:
    - Count: mérete
    - Key: kulcs – ez alapján lehet keresni
    - Data: az objektum tartalma
  - Előny: pontosan illeszkedik a tárolt adatok méretéhez
  - Hátrány: a törlések miatt kisebb-nagyobb lyukak keletkeznek
    →Időnként töredezettségmentesíteni kell (defragmentálás)

Az adatok szervezése

- **Szektor rendszer**
  - Fix adategységek – szektorok (tipikusan 512 bájt)
  - Előny: könnyebb vele bánni, a szabad hely nem töredezik
  - Hátrány: op. rendszernek kell a változó méretű fájlokat fix méretű szektorokra leképeznie
  - Egy szektor felépítése:



    - Rés: legyen idő az írófej kikapcsolására
    - Előtag: fej kalibrációja (jelerősség és adatsűrűség)
    - „Adat kezdődik” jelzi a kalibráció végét
    - „Töltelék terület” időt hagy, hogy az utolsó bájtok is elhagyják a fejet

Az adatok azonosítása

- Avagy hogy hivatkozzunk egy szektorra?
- Fizikai elhelyezkedés leírása:
  - *Sáv* (track): a radiális távolságot adja meg
  - A sávok azonosítása:
    - *Cilinder* (cylinder): egymás feletti adathordozók azonos sávjai
    - *Fej* (head): Egy cilinderen hányadik adathordozó réteg
  - Szektorok fizikai azonosítása:

    → **CHS koordinátákkal (cylinder-head-sector)**
    (melyik cilinderen, melyik fej alatt, hányadik szektor)
- Tehát a merevlemez így azonosít egy szektort.
- És a külvilág hogy azonosít egy szektort?
- Amikor az op. rendszer egy szektort beolvas, hogy mondja meg, melyikre gondol?

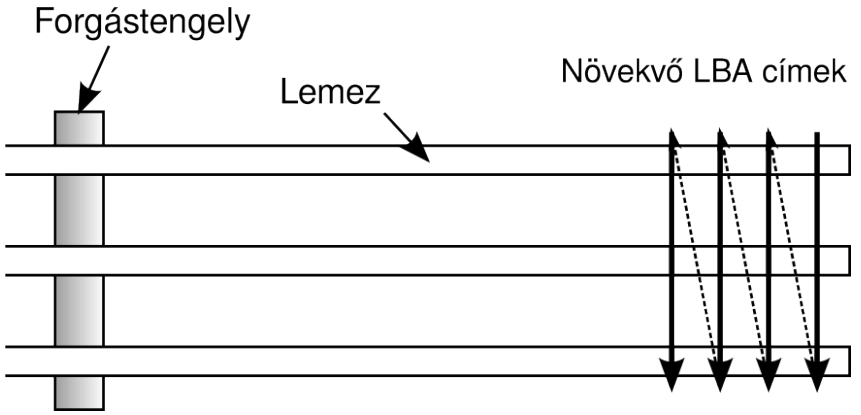
  → **Logikai címzéssel**

Az adatok azonosítása

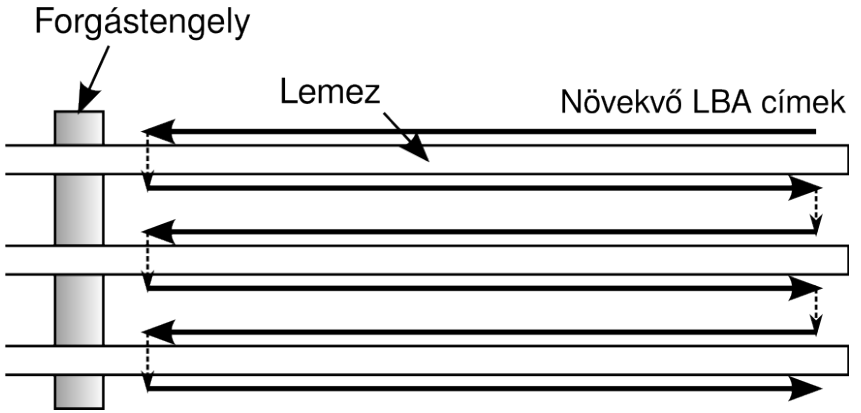
- **Logikai címzés**
  - Minek? Miért nem jó, hogy ha az op. rendszer is CHS-t használ?
    - Régen így volt. Hátrányok:
      - A HDD nem tudja a külvilág felé eltakarni a szektorok meghibásodását
      - ATA szabvány csak 8.4 GB-ot tudott CHS-el címezni
  - Ezért bevezették a logikai címzést: **Logical Block Address, LBA**
  - A szektorokat *egyetlen sorszámmal* azonosítjuk (hányadik a diszken)
  - Az op. rendszer csak egy szektorszámot mond a HDD-nek
  - A HDD ezt fizikai CHS címmé képezi le, és megkapja a szektor helyét
  - A HDD fekete doboz lett!
    - Az op. rendszernek nem kell ismernie a HDD felépítését
    - Eltitkolhatja a hibáit (a hibás szektort kiveszi a leképezésből)

Az adatok azonosítása

- Logikai címek leképzése a fizikai CHS címekre
  - Cilinder stratégia:



  - Szerpentin stratégia:



  - Stb.

Zóna rendszerű adattárolás

- A merevlemezek állandó szögsebességgel forognak
- Ha minden sávban ugyanannyi szektor van:
  - Minden szektort ugyanannyi ideig tart elolvasni
  - A külső peremen egy szektor sokkal több helyet foglal

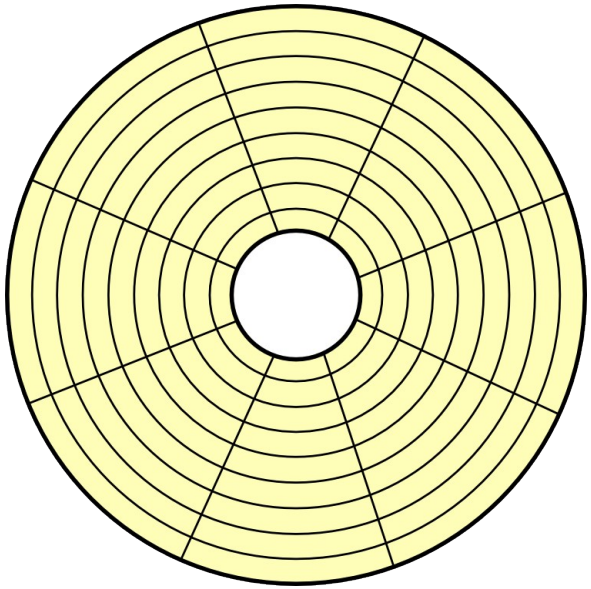
    → A belső szektorokban sokkal nagyobb az adatsűrűség
  - Régen ez volt szokásban
- Jó lenne a külső peremen is a belsőhöz hasonló adatsűrűséggel írni
- Fix adatsűrűség:
  - Kifelé haladva minden sávban egyre több szektor lesz
  - A külső sávok felé haladva egyre gyorsabban kell elolvasni a szektorokat
  - Az elektronikának minden sávban sok a dolga:

    Több 100.000 sáv adataiból kell kikeresnie, hogy épp milyen gyorsan kell olvasnia!
  - Arany középút: nem váltunk sebességet minden sávban, csak zónánként

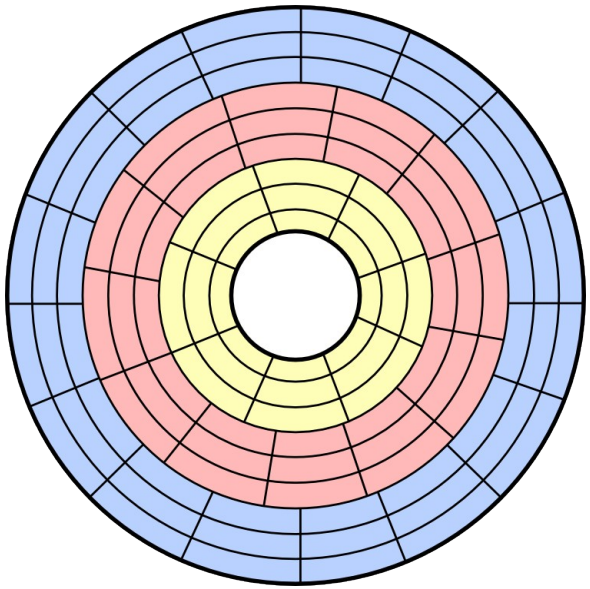
    → **zóna rendszerű adattárolás (ZBR, Zoned Bit Recording)**

Zóna rendszerű adattárolás

- A sávokat zónákba sorolják
  - Egy zónában minden sávban ugyanannyi szektor van
  - Egy zónán belül fix az adatsebesség
  - ma kb. 50 zónát használnak
  - Mellékhatás: külső zónák adatátviteli sebessége nagyobb!
    (Mert ott több a szektor, de a lemez ugyanannyi idő alatt fordul)



ZBR nélkül



ZBR-rel

Zóna rendszerű adattárolás

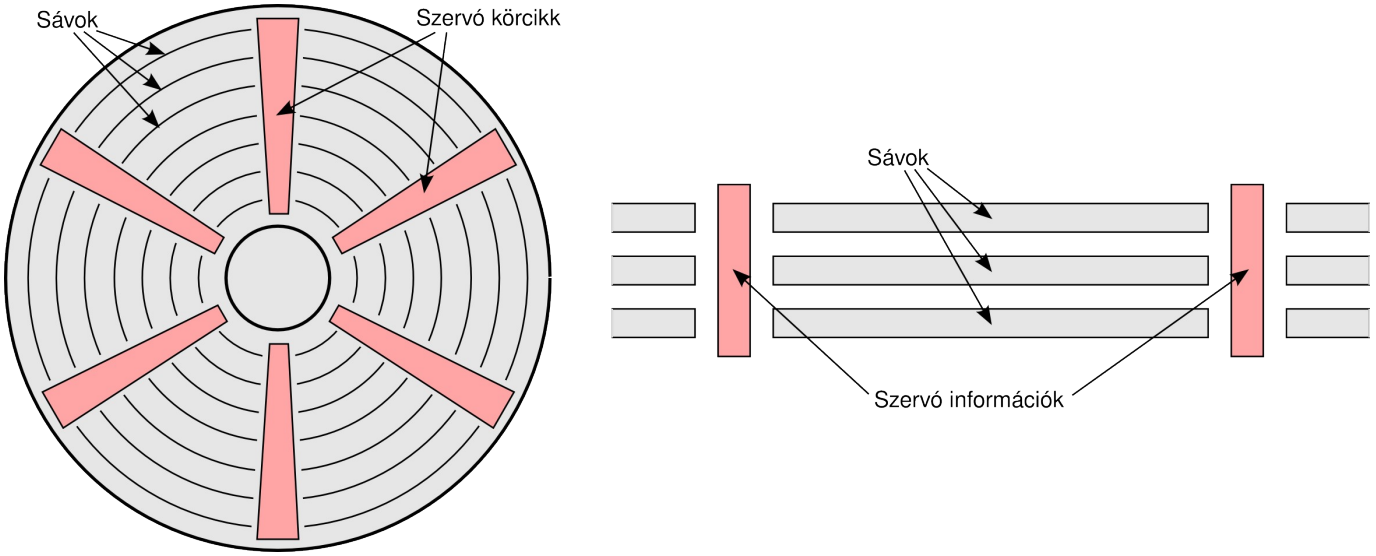
- Alternatív megoldások:
  - Állandó adatsebesség használata változó forgási sebességgel
    - CLV (constant linear velocity), optikai meghajtókban
    - A külső peremen kb. 2.5x lassabban forog a lemez, mint bent
  - (Kvázi) állandó adatsebesség zónákkal
    - Z-CLV (zoned constant linear velocity), fejlett, írásra is képes optikai meghajtókban
    - Egy zónán belül CLV-vel azonos adatsebességet használnak
    - Zónaugrásnál adatsebességet váltanak
    - Pl. egy 40x-es CD író
      - Belső zóna: 24x-es sebesség
      - Középső zóna: 32x-es sebesség
      - Külső zóna: 40x-es sebesség

Szervó vezérlés

- Nagy sebességű mozgó alkatrészek
  - A lemez kicsit vibrál
  - Maga a HDD is mozoghat
  - A fej nem tud elegendő pontossággal pozícionálni
  - Nem ritkán
    - a fej leszalad a sávról,
    - félrepozícionál,
    - vaktában repül
- Szabályozási mechanizmus kell
  - Sávváltáskor segít a fejnek célba jutni
  - Segít a sáv követésében

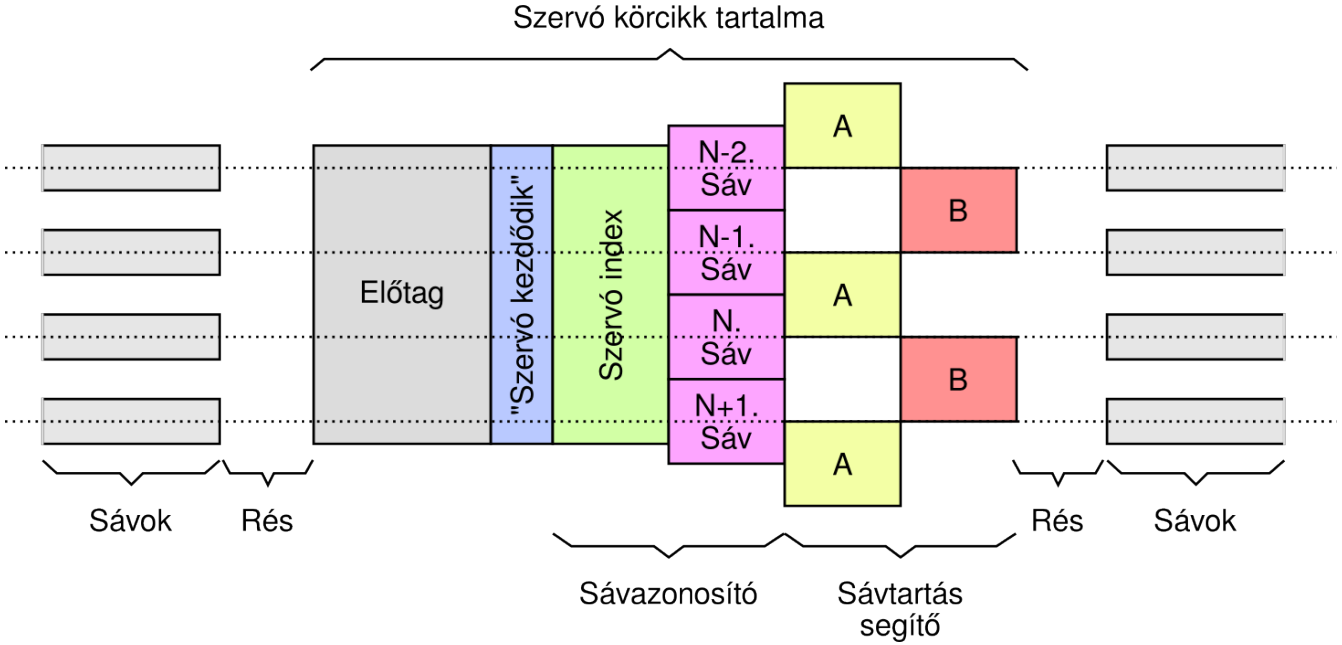
Szervó vezérlés

- Az adathordozón körcikkeket jelölnek ki (servo wedge)



- Szervó körcikk tartalma:
  - Van rajta sávazonosító – a fej rögtön tudja, merre jár
  - Sávtartást segítő sakktábla mintázat
- Gyártáskor kerül a lemezre
- Ha megsérül, a lemeznek vége

Szervó vezérlés



- Sávazonosító:
  - Szervó index: hányadik körcikk a lemezen
  - Sáv száma
- Sávtartást segítő sakktábla
  - Kétféle mágneses jel
  - Ha a fej a sáv közepén halad, mindkettőt egyforma erősen veszi
  - Ha elcsúszik, az egyik jel erősödik → a fej ellentétes irányba korrigál

Illesztőfelületek

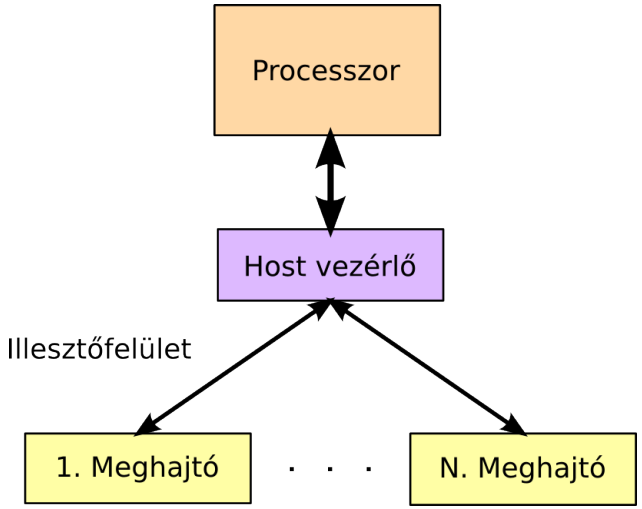
Merevlemezek illesztése

- A merevlemez szektorokra vonatkozó írás/olvasás kéréseket fogad és szolgál ki
- De hogyan juttassuk el hozzá a kéréseket? És a kérdéses adatokat hogy vigyük át?
  - Kössük a merevlemezt közvetlenül a buszra
    - Minden architektúrának más a busz protokollja
    - A busz protokoll egy architektúra különböző generációiban más és más lehet
    - Új processzor → új busz → a merevlemez nem lesz kompatibilis
    - Rossz ötlet
  - Használjunk szabványos illesztőfelületet
    - „Minden” merevlemez beszélje ugyanazt a nyelvet
    - A processzor buszára illesszünk egy olyan eszközt, ami a buszon érkező adatátviteli kéréseket lefordítja erre a szabványos nyelvre

      → Ez a **host controller**
    - Új processzor → új busz → új host controller → a merevlemez a régi!

Merevlemezek illesztése

- Az illesztőfelület részei:
  - Fizikai specifikáció: a jelátvitel részletei
    - Kábelek, csatlakozók, soros/párhuzamos, le-/felfutó él, stb.
  - Protokoll specifikáció: a párbeszéd mikéntje
    - Adatátviteli kérés paraméterei, sorrendje, stb.



Az ATA illesztőfelület

- A PC-hez kötődik
- Az ATA egyszerű, támogatása olcsó
- Fizikai tulajdonságai:
  - 40 eres kábel (később 80 lett, 40 új földvezeték belefésülésével)
  - Párhuzamos adatátvitel
  - 16 bites adategységek
  - Aszinkron átvitel, a „Read strobe” ill. „Write strobe” jelre
  - 1 kábelre 2 meghajtó tehető
- Adatátviteli módok:
  - PIO Mode:
    - A CPU felügyeli minden bájt átvitelét
    - A CPU-t teljesen leköti, közben nem tud mással foglalkozni
  - DMA mód:
    - A merevlemez beépített DMA vezérlője közvetlenül a rendszermemóriába ír/olvas
    - A CPU-nak csak el kell indítania, adatátvitel közben mást csinálhat

Az ATA illesztőfelület

- Adatátviteli módok (folyt.):
  - Ultra DMA mód:
    - Mint a DMA, de az órajel fel- és felutó élén is visz át adatot
    - 80 eres kábel kell, és CRC hibadetektálás
    - Ez a legnagyobb ATA sebesség: 133 MB/s
- Protokoll jellemzők
  - Először a paramétereket kell elküldeni (adat cím, hossz, memóriacím)
  - A parancskód elküldése indítja a műveletet
  - Az aktuális állás egy státuszregiszterből olvasgatható
  - **Amíg be nem fejeződik, nem indítható új kérés!**
- Kiegészítés
  - Optikai meghajtók új parancsokat igényelnek (bent van-e a lemez, lemez típus lekérdezés, stb.)
  - Ezek a parancsok az ATA-ban nem, az SCSI-ben megvoltak
  - ATAPI: az ATA kiegészítése SCSI parancsok átvitelével

SATA illesztőfelület

- Az ATA túl széles, árnyékolatlan szalagkábelén nem lehet gyorsabban adatot átvinni
- Megoldás: SATA
  - Protokoll specifikáció:
    - Ugyanaz maradt, mint az ATA-ban!
      (PIO, DMA, UltraDMA, parancsformátum, ATAPI, stb.)
  - Fizikai specifikáció:
    - Árnyékolt, sodort érpáron soros adatátvitel
    - Pont-pont kapcsolat a host controller-el
- SATA-1: 150 MB/s, SATA-2: 300 MB/s, SATA-3: 600 MB/s
- SATA-2 nagy újítása: **NCQ** (Native Command Queueing)
  - Nem kell megvárni a régi adatátvitel befejeződését
  - Több kérés is küldhető, a meghajtó várakozási sorba teszi
  - Ha több kérés van a sorban, a meghajtó szabadon választ egyet
    - Lehetőleg azt, amelyik a fejnek épp útjába esik

Az SCSI illesztőfelület

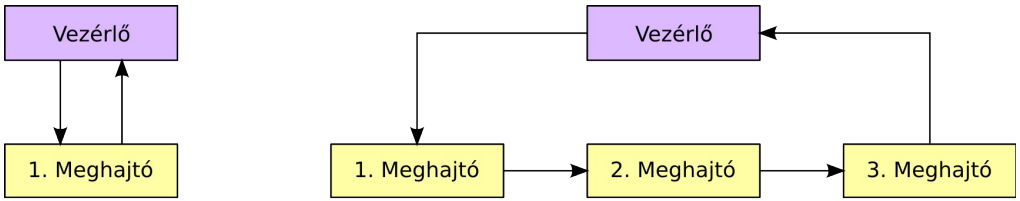
- Az ATA-nál több funkciót ismerő, összetett szabvány
- Hangsúly: megbízhatóság, hibákból való gyors felépülés
- Szerverek, munkaállomások használják
- Valójában egy busz
  - Rajta lógnak a meghajtók,
  - … és a host controller
  - Mindenkinek van egy prioritása
  - Önkiválasztó arbitráció
- Fizikai interfész:
  - Sok variáns
  - 8 vagy 16 bites adategységek
  - 50 vagy 68 eres kábel
- Protokoll specifikáció:
  - A kérések a paramétereikkel parancsleírók formájában utaznak a buszon
  - A parancsok sorbaállíthatók (Command Queueing)
- Kezdetben 5 MB/s, napjainkban 640 MB/s

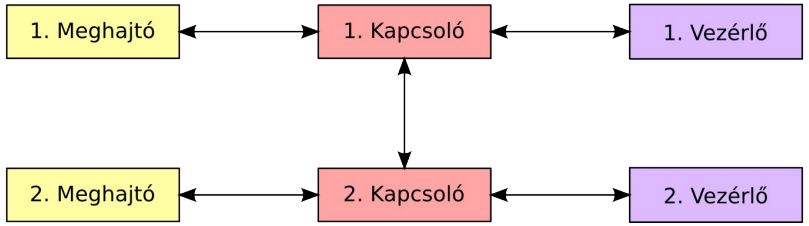
A soros SCSI illesztőfelület

- SAS
- Bevárták a SATA megjelenését
- Átvették a SATA fizikai jelétvitelét!
  (csatlakozók, kábelek, stb.)
- A SAS tehát SCSI parancsokat visz át SATA kábeleken
- A SAS tárolórendszerek SATA meghajtókat is támogatnak!
  - Fizikai akadálya nincs
  - ATA parancsok átviteléhez kiegészítették a SCSI-t: STP
- ATA ↔ SCSI kétirányú átjárhatóság:
  - ATAPI: SATA felületen SCSI parancsokat visz át
  - STP: SAS tárolórendszerben ATA parancsokat visz át
- SAS meghajtókon 2 csatlakozó van
  - Két vezérlőre / számítógépre köthető
  - Ha az egyik elromlik, a másikon elérhetőek maradnak az adatok

Az FC illesztőfelület

- Fibre Channel
  - Csak jelátviteli szabvány
  - Protokoll specifikációja nincs (tipikusan SCSI parancskészlet)
- Sebességek:
  - Optikai kábelen: 3200 MB/s, 50 km hosszú kábelek
  - Réz kábelen is lehet (lassabban, rövidebb kábellel)
- Egy csomagkapcsolt hálózat a vezérlő(k) és a meghajtók között



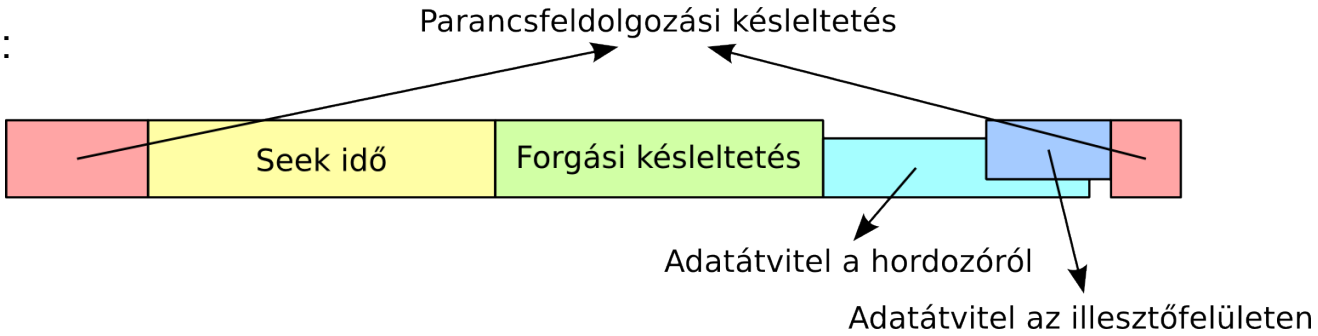


Merevlemezek teljesítménye

- Teljesítményjellemzők:
  - Válaszidő
  - Kiszolgálási idő
  - Átvitel
    - IOPS: kiszolgált ki-/beviteli kérések száma/sec
    - Adatátviteli sebesség: IOPS x Átlag blokkméret
- **Válaszidő**
  - Sorbanállási késleltetés + kiszolgálási idő
  - Nem lineárisan nő a terheléssel!
  - FIFO esetben:

Merevlemezek teljesítménye

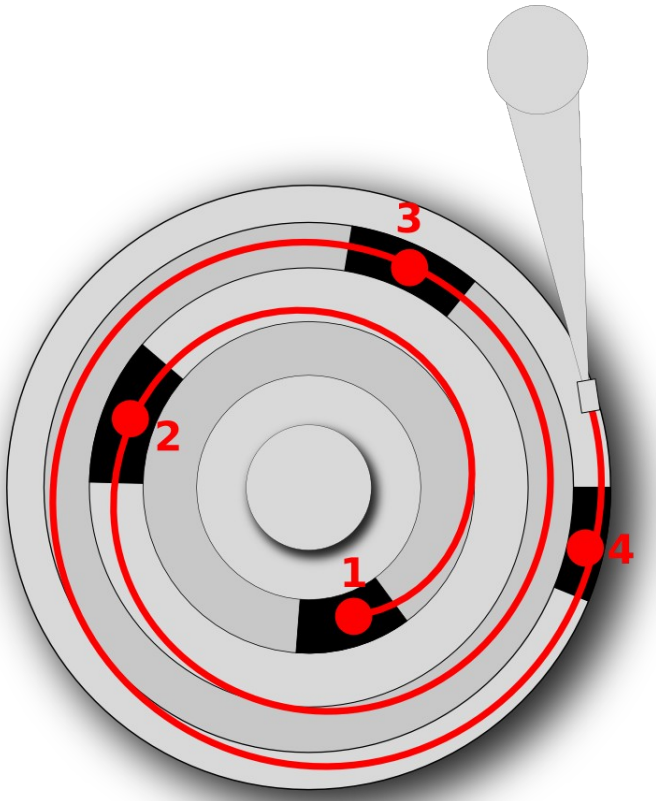
- **Kiszolgálási idő** részei:



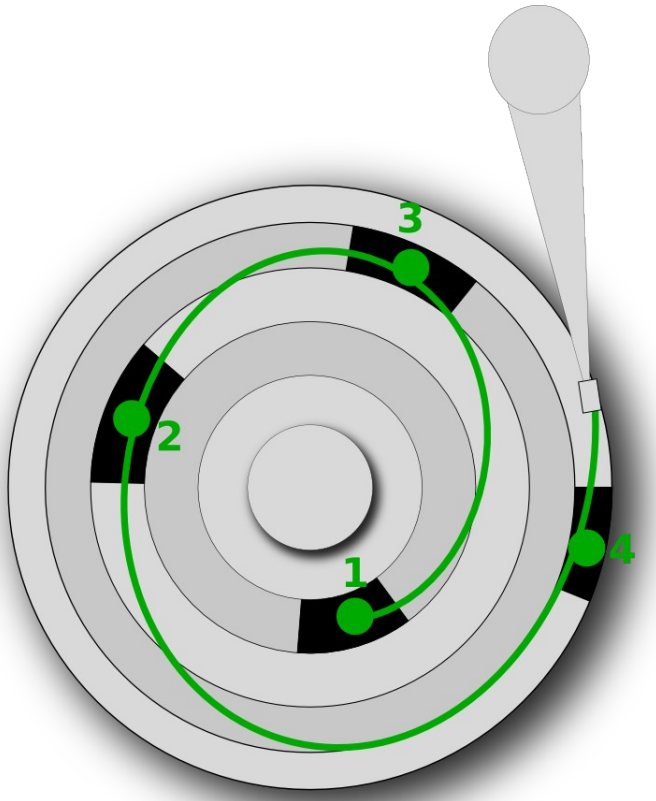
- Példa:
  - Forgási sebesség: 10.000 fordulat/perc, átlag seek idő: 4.5 ms, parancsfeldolgozási késleltetés: 0.3 ms
  - A keresett sávban legyen 600 db 512 bájtos szektor
  - Illesztőfelület sebessége: 100 MB/s
- Mennyi ideig tart egy 4 kB-os adatátvitel?
  - Forgási késleltetés: 1 fordulat: 6 ms. Véletlen időpontban átlag 3 ms-ot kell várni, hogy a keresett adat fölé kerüljön a fej
  - Adat leolvasása: 4 kB = 8 szektor. 1 szektor átvitele: 6 ms / 600 = 0.01 ms.
  - Adat átvitele az illesztőfelületen: átlapolható! Leolvasási idő + az utolsó szektor átvitele: 512 bájt / 100 MB/s = 0.005 ms
  - Összesen: 0.3 + **4.5** + **3** + 0.08 + 0.005 = **7.885 ms**
- A forgási és a seek idő dominál!
- **Érdemes sok adatot egyszerre átvinni!** Seek és forgási idő csak egyszer adódik hozzá!

Merevlemezek teljesítménye

- A SATA-2 és a SCSI is támogatja a kérések sorba állítását
- Kiszolgálási sorrend nagyon nem mindegy!



Kiszolgálás érkezési sorrendben



Kiszolgálási sorrend optimalizálása

Merevlemezek teljesítménye

- Stratégiák kiszolgálási sorrendre:
  - Csak seek időre optimalizálva
    - SSTF (Shortest Seek Time First)
      Mindig azt választja, amelyik a fej aktuális cilinderéhez legközelebb eső cilinderen van
    - LOOK (Elevator Seek)
      Elindul a fej az egyik irányba, az útbaeső igényeket kiszolgálja, majd visszafordul, ismét veszi az útba esőket, stb...
  - Teljes hozzáférési időre optimalizálva
    - SATF (Shortest Access Time First)
    - A forgási késleltetéssel is számol. Amelyik igény a legközelebb esik (seek+forgás), azt szolgálja ki
    - Sokkal hatékonyabb
    - Sokkal bonyolultabb
      – Pontos seek profile kell: meddig tart „a” cilinderből „b”-be tolni a fejet – minden cilinder párra!!!

Félvezető alapú háttértárak (SSD)

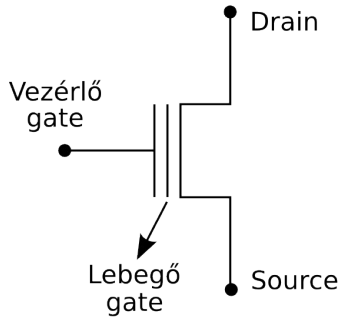
Félvezető alapú háttértárak

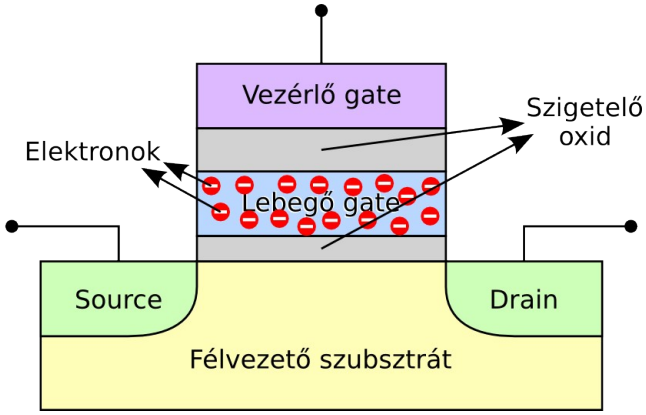
- Félvezető memória tárolja az adatokat
- Mi SSD alatt a flash memórián alapuló háttértárat értjük
- Miért jó?
  - Zajtalan
  - Hő és mechanikai behatásokkal szemben ellenáll
- Történelem:
  - 1995: M-Systems, hadászati célokra
  - 2004: Adtron, első SATA felületű SSD: 16 GB, 11.000$
  - 2007: Piaci robbanás, egyre nagyobbak, gyorsabbak és olcsóbbak
  - 2011: Thaiföldi árvíz megdrágítja a HDD-ket, SSD-k iránt rekord kereslet

Egy bit tárolása

- Alapja a lebegő gate-es tranzsitor
- **„Sima” MOSFET tranzisztor:**
  - 3 lába van: gate, source, drain
  - A source-drain áram a gate feszültséggel szabályozható
  - Kapcsolóként használva:
    - Gate-en nincs fesz.: nincs áram (kapcsoló nyitva)
    - Gate-en van fesz. (>V_T): folyhat áram (kapcsoló zárva)
- **Lebegő gate-es tranzisztor:**
  - Két gate-je van:
    - A vezérlő gate szerepe ugyanaz
    - A lebegő gate-be elektronok zárhatók
      - Benne is maradnak
      - Megváltoztatják a tranzisztor viselkedését

Egy bit tárolása



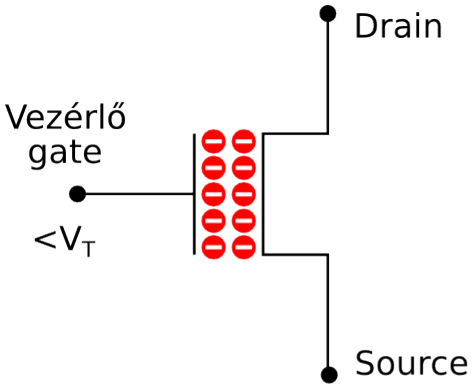


	Vannak elektronok	Nincsenek elektronok
„Normál” gate fesz.	Zár	Zár
„Kis” gate fesz.	Nyit	Zár
Gate fesz.=0	Nyit	Nyit

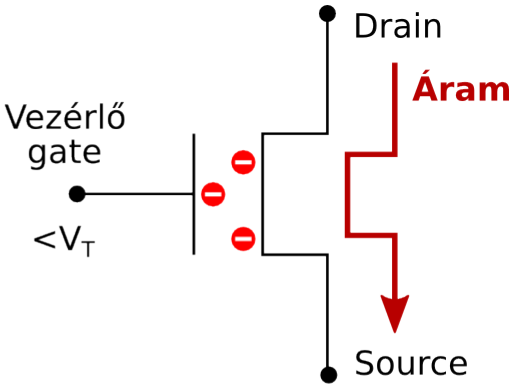
- Elektronok jelenléte: „megnehezíti” a kapcsoló zárását

Egy bit tárolása

- Adatok reprezentálása:
  - Egy tranzisztor 1 bitet tárol (egyelőre)
  - Vannak elektronok: **0**, nincsenek: **1**
- Adatok kiolvasása
  - Kis feszültséget adunk a gate-re
  - Megnézzük, folyik-e áram a source és a drain között
  - Ha igen → 1, ha nem → 0



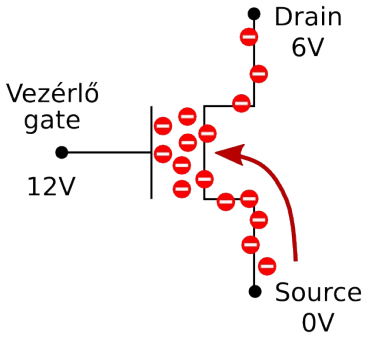
0-ás bit kiolvasása

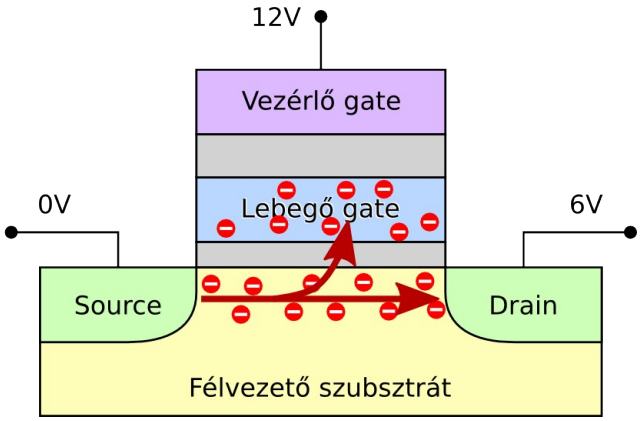


1-es bit kiolvasása

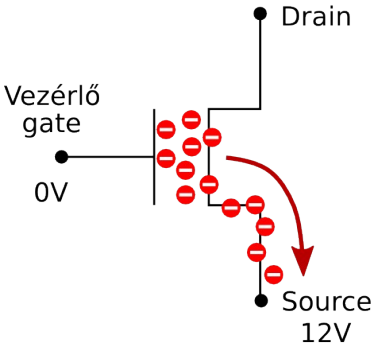
Egy bit tárolása

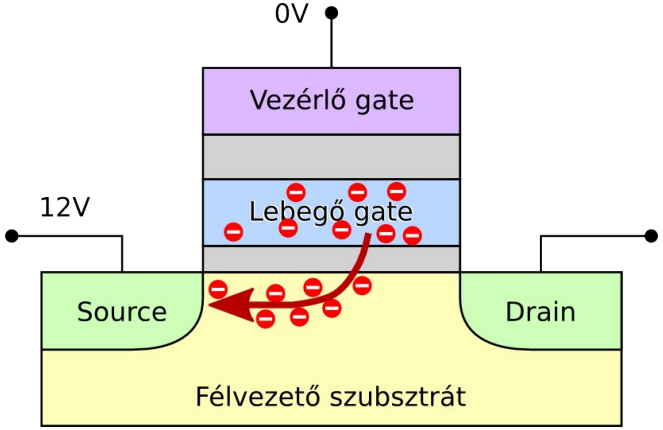
- Váltás 1-ből 0-ba: **programozás**





- Váltás 0-ból 1-be: **törlés**





Egy bit tárolása

- A programozás és törlés mellékhatása:
  - A lebegő gate-be és onnan ki mászkáló elektronok beragadhatnak a szigetelőbe (örökre)
  - A szigetelőrégteg egyre kevésbé fog szigetelni
  - **Öregedés!**
  - Adott számú programozás/törlési ciklus után használhatatlan lesz
    - SLC esetben, 20-25 nm-es gyártással kb. 100.000 ciklus

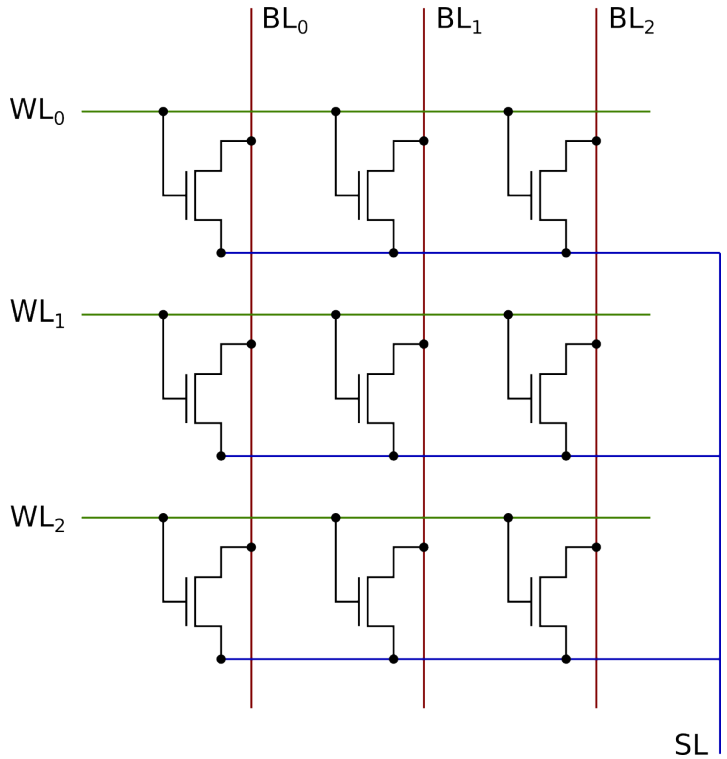
Néhány bit tárolása

- Eddig 2 töltöttségi állapot volt: nincs töltés → 1, van → 0
- Miért ne használnánk több töltöttségi szintet?
- *N* bit tárolásához 2^N töltöttségi szintet kell megkülönböztetni
- Minél több az elektron, annál „nehezebb” zárni a tranzisztort
- SLC flash: 1 tranzisztor → 1 bit
- MLC flash: 1 tranzisztor → több (tipikusan 2) bit
- Összehasonlítás:
  - Egy n bites MLC tárolási kapacitása ugyanannyi tranzisztorral n-szeres az SLC-hez képest (olcsóbb)
  - Viszont ***sokkal*** gyorsabban öregszik!
    - SLC: 100.000 ciklus
    - MLC: 5.000 ciklus

Sok bit tárolása

- A tranzisztorokat 2D rácsba rendezzük
- De hogyan?
- Kétféle elterjedt megoldás:
  - NOR flash
  - NAND flash

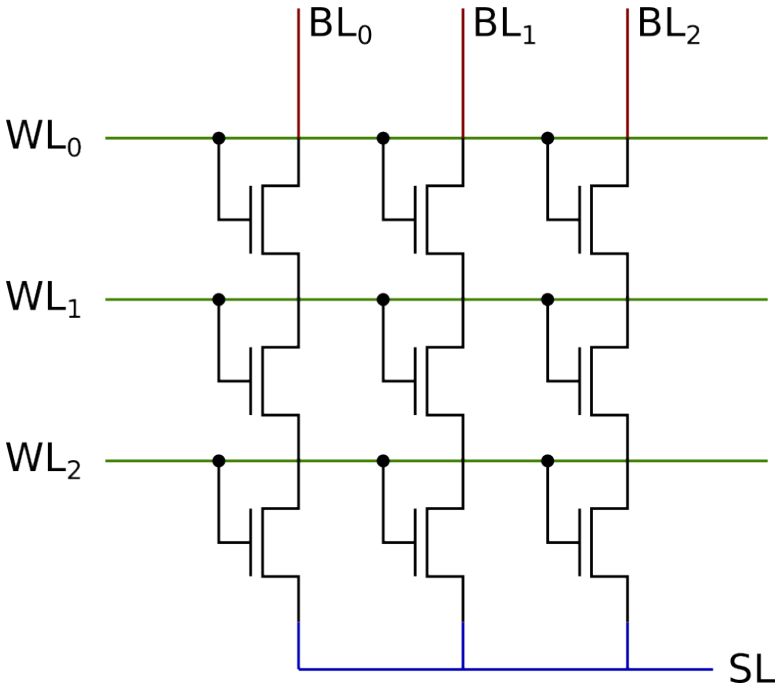
NOR flash memória



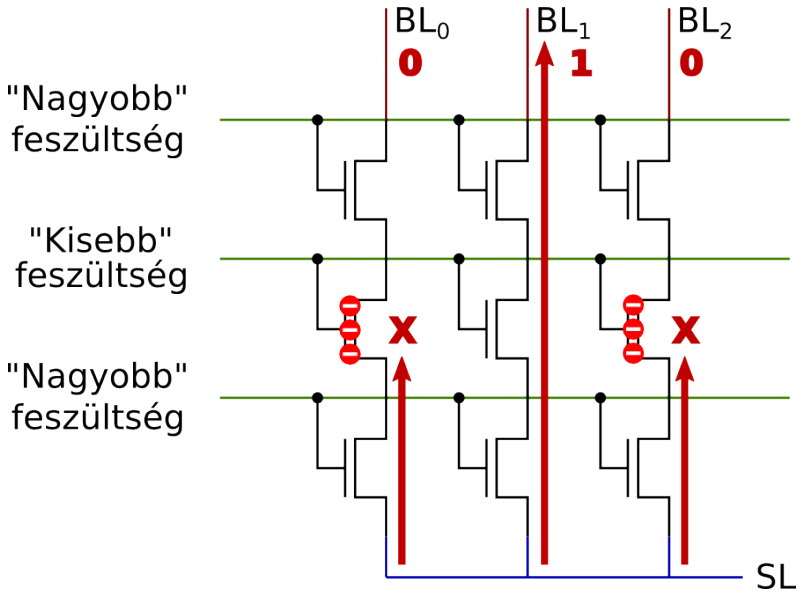
- i. sor j. oszlop kiolvasása:
  - WL_i vezetékre kis feszültség, a többire 0V
  - i. sorban az 1-et tárolók zárnak, a 0-át tárolók nyitnak
  - i,j bit értéke: folyik-e áram az SL és a BL_j között
- Bitjei egyesével programozhatók, de csak egyben törölhetők

NAND flash memória

- Toshiba, 1989
- Cél: olcsóbb gyártás (kevesebb vezeték)
- Egy oszlop source és drain elektródái össze vannak kötve
- Kevesebb vezeték → több hely marad a hasznos tranzisztoroknak



NAND flash memória



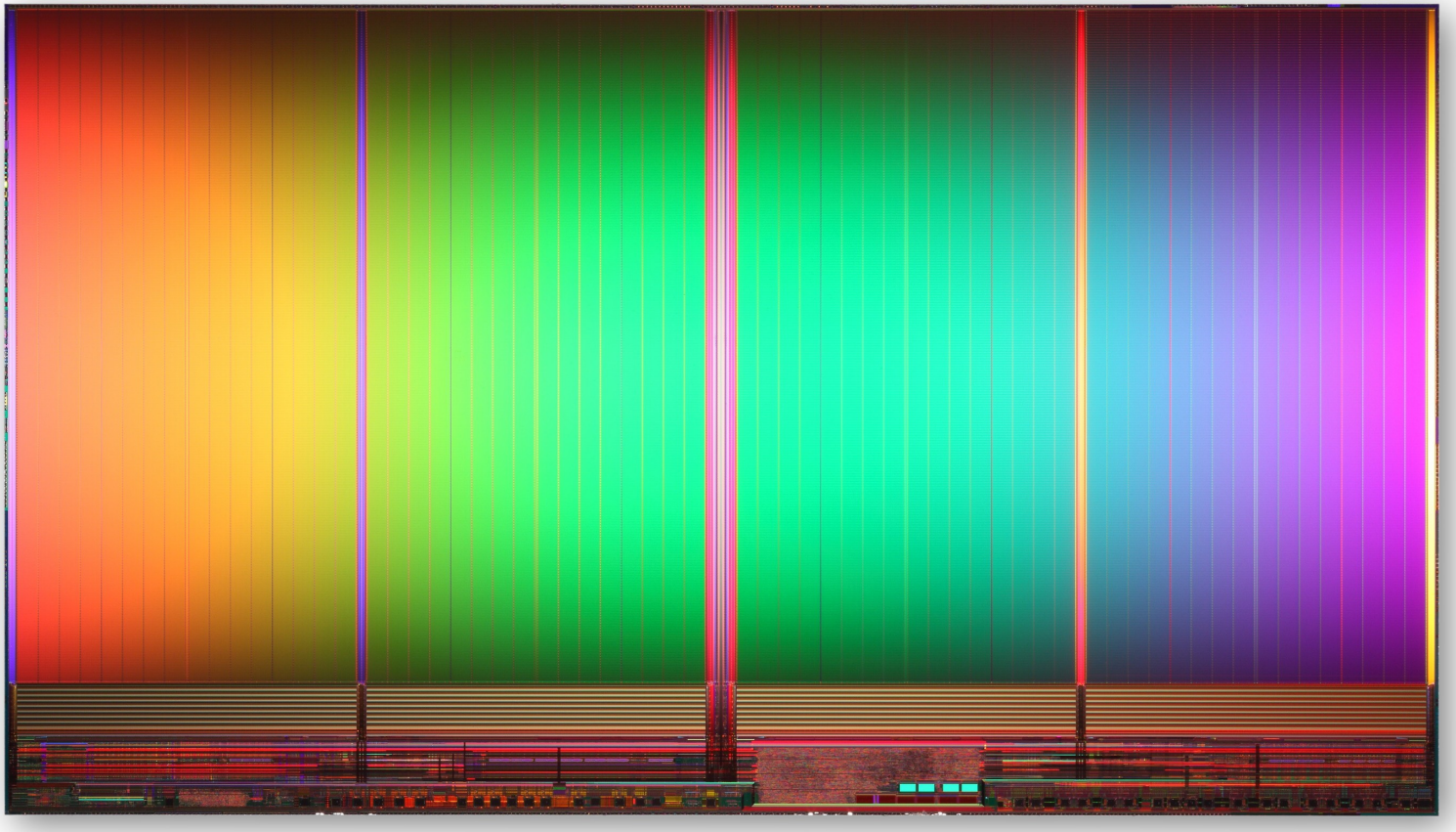
- i. sor j. oszlop kiolvasása:
  - WL_k k≠i vezetékekre „normál” feszültséget adunk (mindet kinyitjuk)
  - WL_i vezetékre kis feszültséget adunk
  - i,j bit értéke: folyik-e áram az SL és a BL_j között
- Elvileg lehet a biteket egyesével programozni, de csak egyben lehet törölni

NAND flash memória

- Kezdetektől fogva háttértárnak szánták
- Olvasás és írás egységei: **lapok**
  - Lap: a tárolómező egy sorának bitjei
- Az egész tárolómező neve: **blokk**
- Törölni csak *teljes blokkot* lehet, lapokat egyesével *nem*
- Tárolási hierarchia:
  - 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

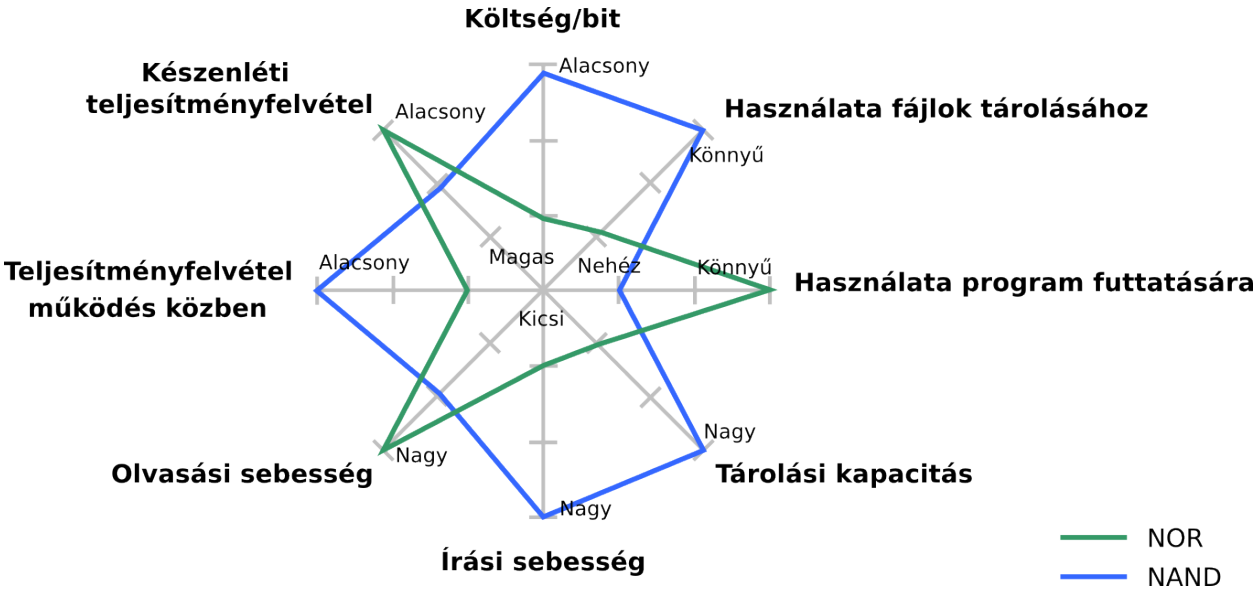
NAND flash memória

- Az Intel 8 GB-os 2-bites MLC NAND flash lapkája:



NOR vs. NAND

- Nemcsak az architektúra, a megcélzott felhasználási terület is más



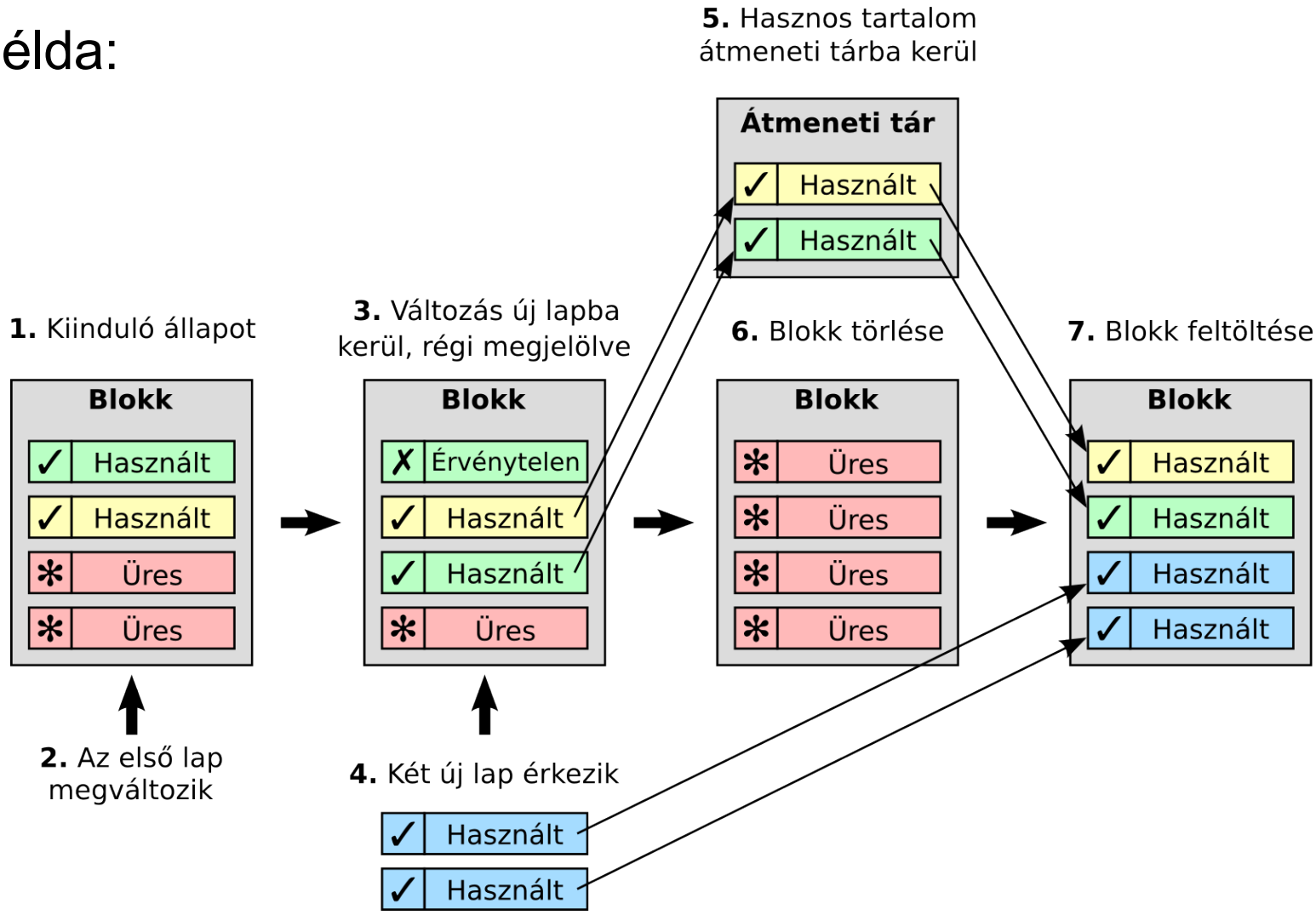
- **NOR**: nem csak háttértárként használják, hanem memóriaként, programok tárolására is. Mivel bájt szinten címezhető, a CPU közvetlenül onnan tudja lehívni az utasításokat.
  - BIOS flash, PDA, telefon, multimédiás eszközök firmware-ének tárolására, stb.
- **NAND**: a lap szervezés a szekvenciális elérésnek kedvez
  - Memóriakártyák (SD, CF, stb.), SSD-k, pendrive-ok, stb.

Az írás és törlés problémája

- HDD-hez képest alapvető változás:
  - Az SSD csak törölt lapra tud írni. Felülírni nem tud. (HDD tud)
  - Az írás és a törlés adategysége különböző
  - Írás használaton kívüli, de nem törölt lapra:
    1) Törölni kell az egész blokkot (akár 2 MB!)
    2) Visszaírni a blokk hasznos lapjait
    3) A kívánt lapra beírni a kívánt adatot
- **Minél többet használjuk, annál lassab lesz!**
  - Az új SSD tele van üres blokkokkal, sokáig nem kell törölnie
  - Akár csak 1 bájt változik egy fájlban
    - Új lapra írja a megváltozott lapot
    - A régit haszontalannak jelöli (nem törli)
  - Ha elfogynak az üres lapok, törölni kell
- Korszerű SSD-k megoldása:
  - Ráérő idejükben összetolják a hasznos lapokat
  - A felszabaduló blokkokat jó előre törlik
  - Csínján kell bánni a lapok rendezgetésével – koptatja az SSD-t!
  - Csak akkor megy, ha van ráérő idő. Ha nincs (pl. terhelt szerver), drasztikus a lassulás.

Az írás és törlés problémája

- Példa:

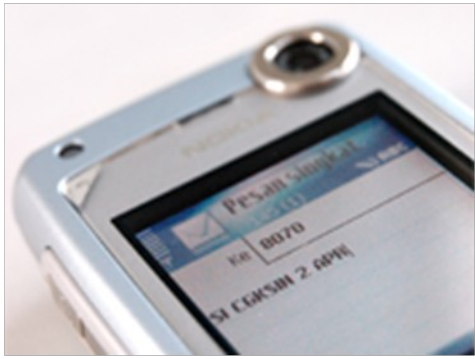


Az SSD vezérlő feladatai

- SSD-nek saját vezérlője van (tipikusan ARM processzor)
- Ez nem a host controller !!!
- Feladatai:
  - Adatátviteli kérések lebonyolítása
  - Szemétgyűjtés
    - használt lapokat egymásra tolja, felszabaduló blokkokat előre törli
  - Kopás-kiegyenlítés
    - Minden blokk egyformán kopjon (statikus és változékony egyaránt)
      - Dinamikus kopás-kiegyenlítés
        - » A beérkező írási kérést a legkevésbé kopott szabad lapot tartalmazó blokkra írjuk
      - Statikus kopás-kiegyenlítés
        - » Nem csak a szabad lapok blokkjait nézi, hanem az egészet
        - » Kiválasztja a legkevésbé kopott blokkot, és oda ír
        - » Ha ott nincs szabad lap, helyet csinál: kiszúr egyet, és áthelyezi egy öregebb blokkra
        - » Hatékonyabb

Az SSD vezérlő feladatai

- További funkciók:
  - Adattömörítés
    - Ha az adatokat tömörítjük, kevesebbet kell írni (lassul az öregedés)
    - SandForce: nem csak tömörít, hanem de-duplikál: ha egy adat már szerepel az SSD-n, nem írja rá ismételten
  - Több írási kérés bevárása
    - Hátha jön 1-2 kérés, ami ugyanarra a lapra vonatkozik
    - Ezek egyetlen írási művelettel végrehajthatók
  - Belső RAID
    - Adatokat többszörözve tárolják, hogy az öregedés okozta hibák ne jelentsenek gondot (biztonsági tartalék)
  - Túlméretezés
    - Az SSD valós kapacitása nagyobb, mint a meghirdetett
    - A különbség tartalékként működik
      - A tartalék területen mindig találunk pár üres blokkot (ha hirtelen kellene)
      - Az elöregedett blokkok kiváltására is jó



Számítógép Architektúrák

Memóriakezelés

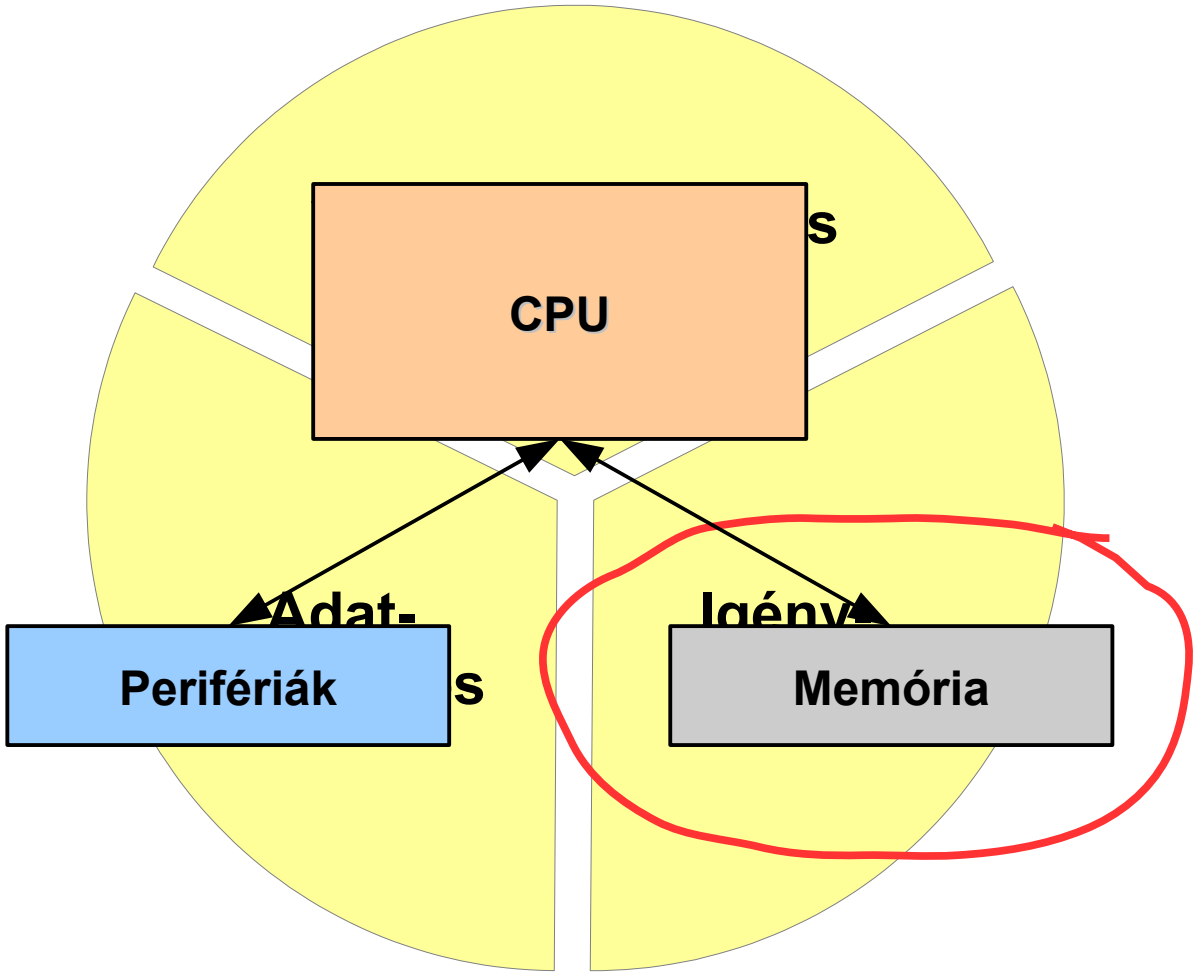
Horváth Gábor
docens
BME Híradástechnikai Tanszék
ghorvath@hit.bme.hu

2012. október 15.
Budapest

IPv6 oktatólabor

- BME-HIT: Magyarország első IPv6 oktatólaborja
- Sajtóvisszhang: 6 hazai, 1 nemzetközi napilap
- 32 bites IPv4 címek elfogytak!
- IPv6: 128 bites címzés
  - A Föld felszínének minden atomja 100 címet kaphat
  - Minden ismert csillagnak 4.8 trillió cím jut
- Mást is tud:
  - Adatbiztonság
  - Mobilitás
  - ...
- Labor célja: IPv6 eszközök és megoldások oktatása
- Részletek: **https://www.facebook.com/hit.bme.hu**

Hol tartunk?



Alapproblémák

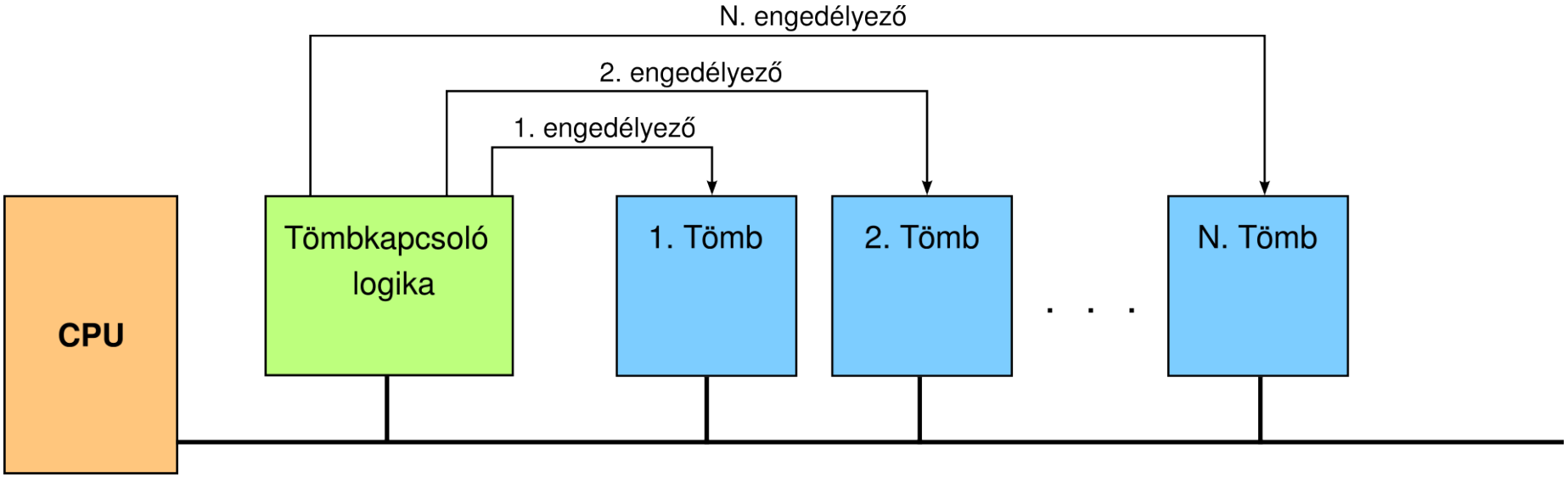
- A CPU címtartományánál több memória kellene

  → ***tárbővítés***
- Nincs annyi memóriánk amennyit a CPU címezni tud
  - De erről hallgatunk, mint a sír
  - A programok azt hiszik, van annyi

  → ***virtuális tárkezelés***

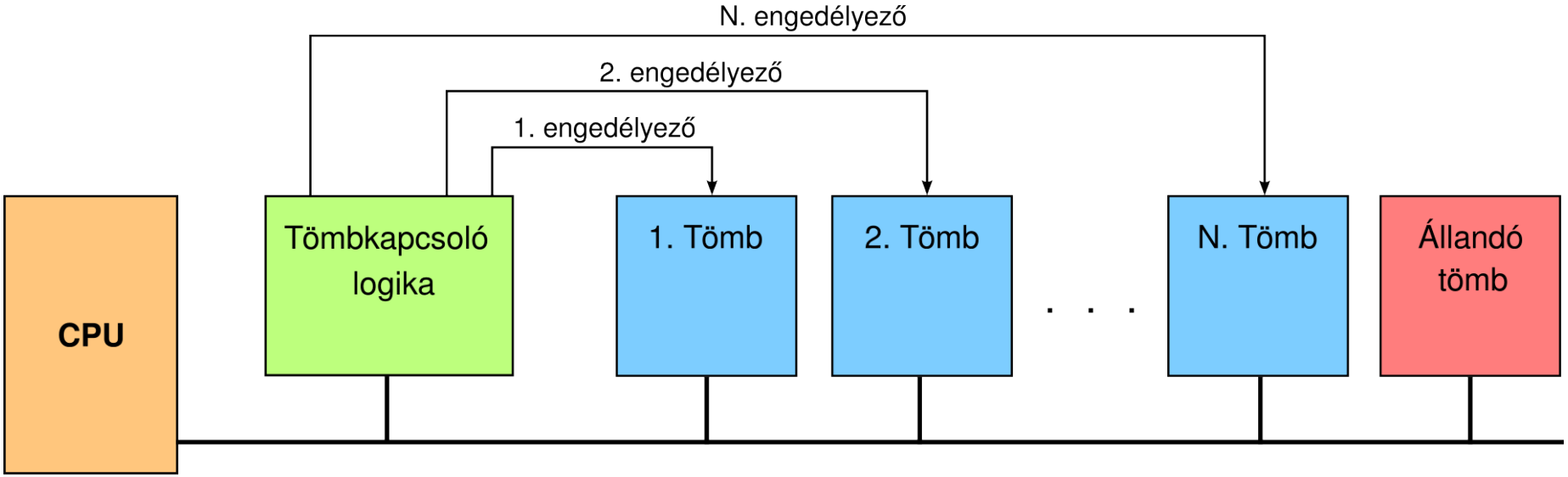
Tárbővítés tömbkapcsolással

- Tömbkapcsolás
  - Bontsuk a szükséges méretű memóriát címezhető méretű tömbökre
  - Vegyünk egy tömbkapcsoló perifériát, ami
    - Mindig csak az egyik tömböt mutatja a CPU-nak
    - Többit kikapcsolja
  - Példa: N fizikailag elkülönülő tömb:

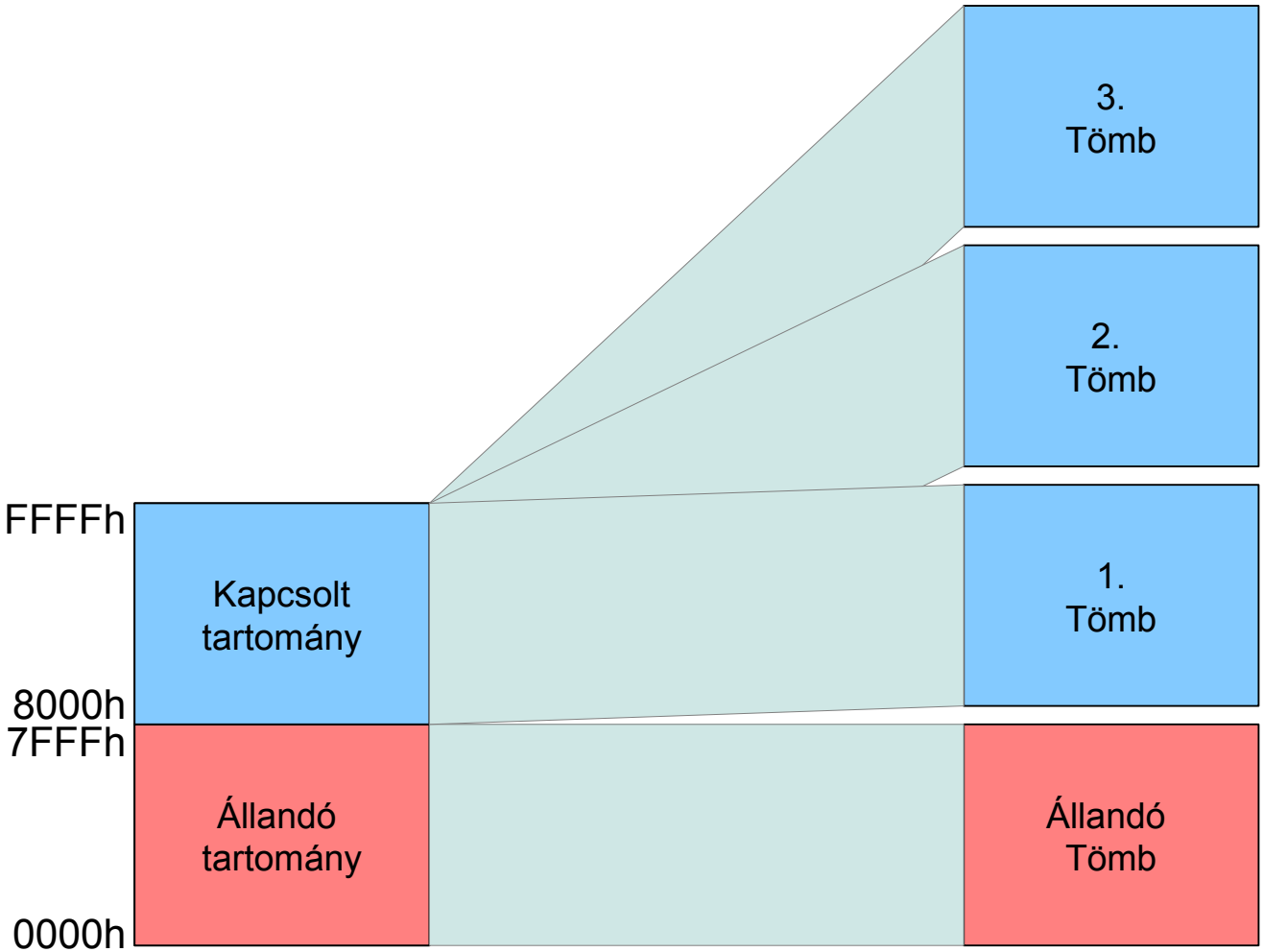


Tárbővítés tömbkapcsolással

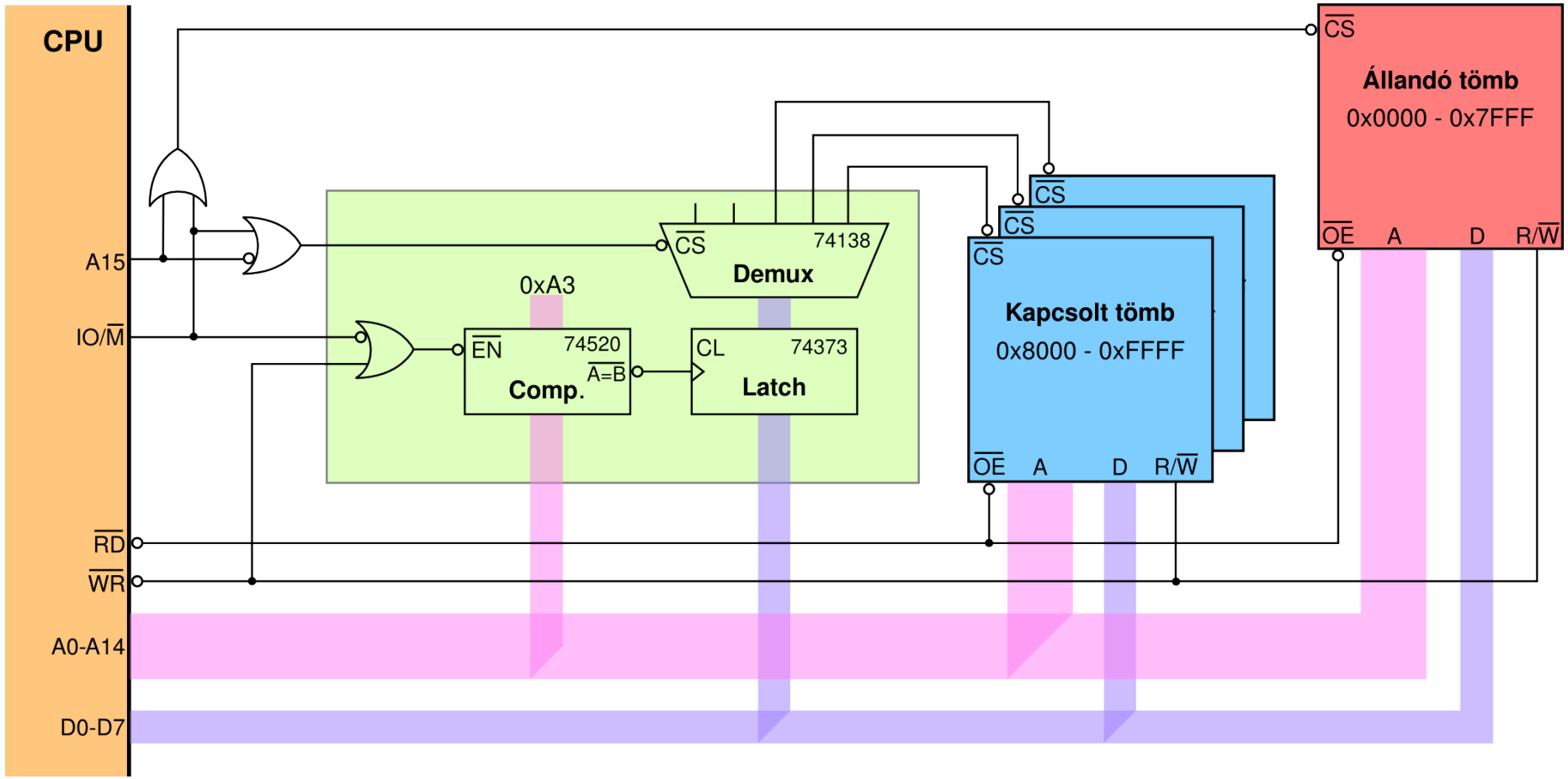
- Tömbkapcsolást a futó program kezdeményezi
  - Tömbkapcsoló perifériának elküldi az aktivizált tömb számát
- Gond:
  - egyes memóriatartalmaknak **mindig** elérhetőnek kell lennie
  - Pl. megszakításkezelő szubrutinok, stack, ...
  - Kisebb tömbök, egyet mindig bekapcsolva tartunk



Tárbővítés tömbkapcsolással

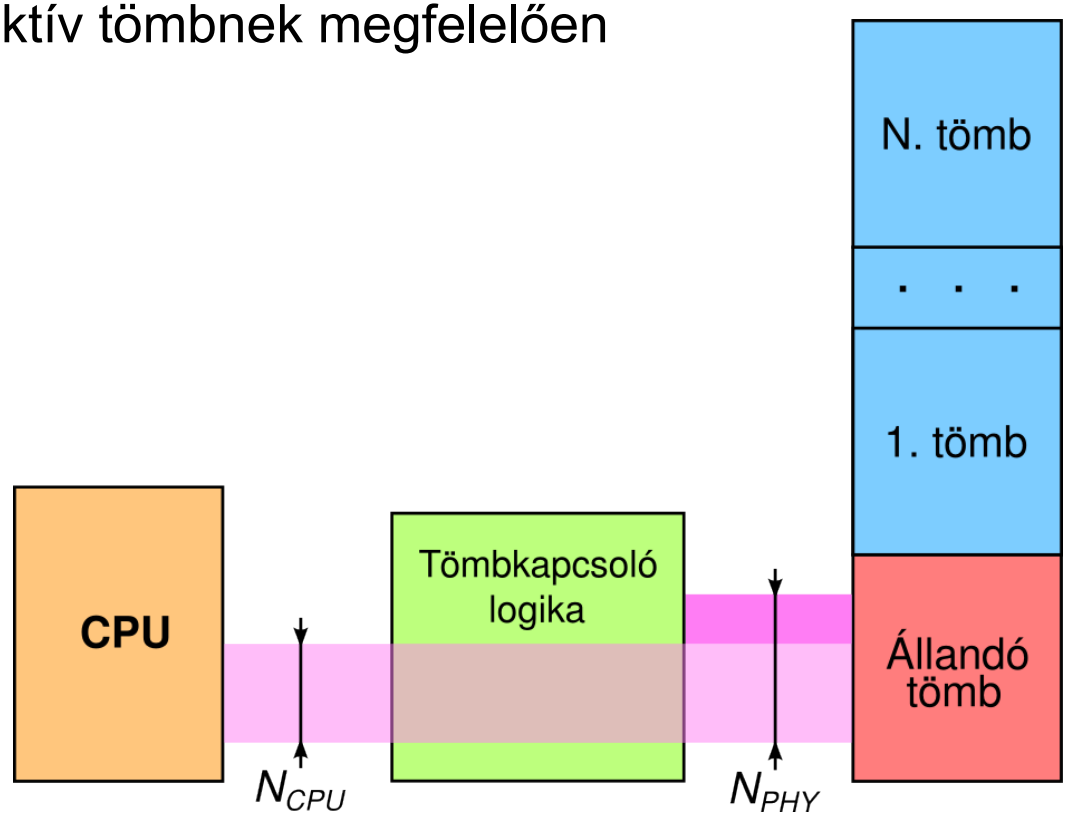


Tárbővítés tömbkapcsolással

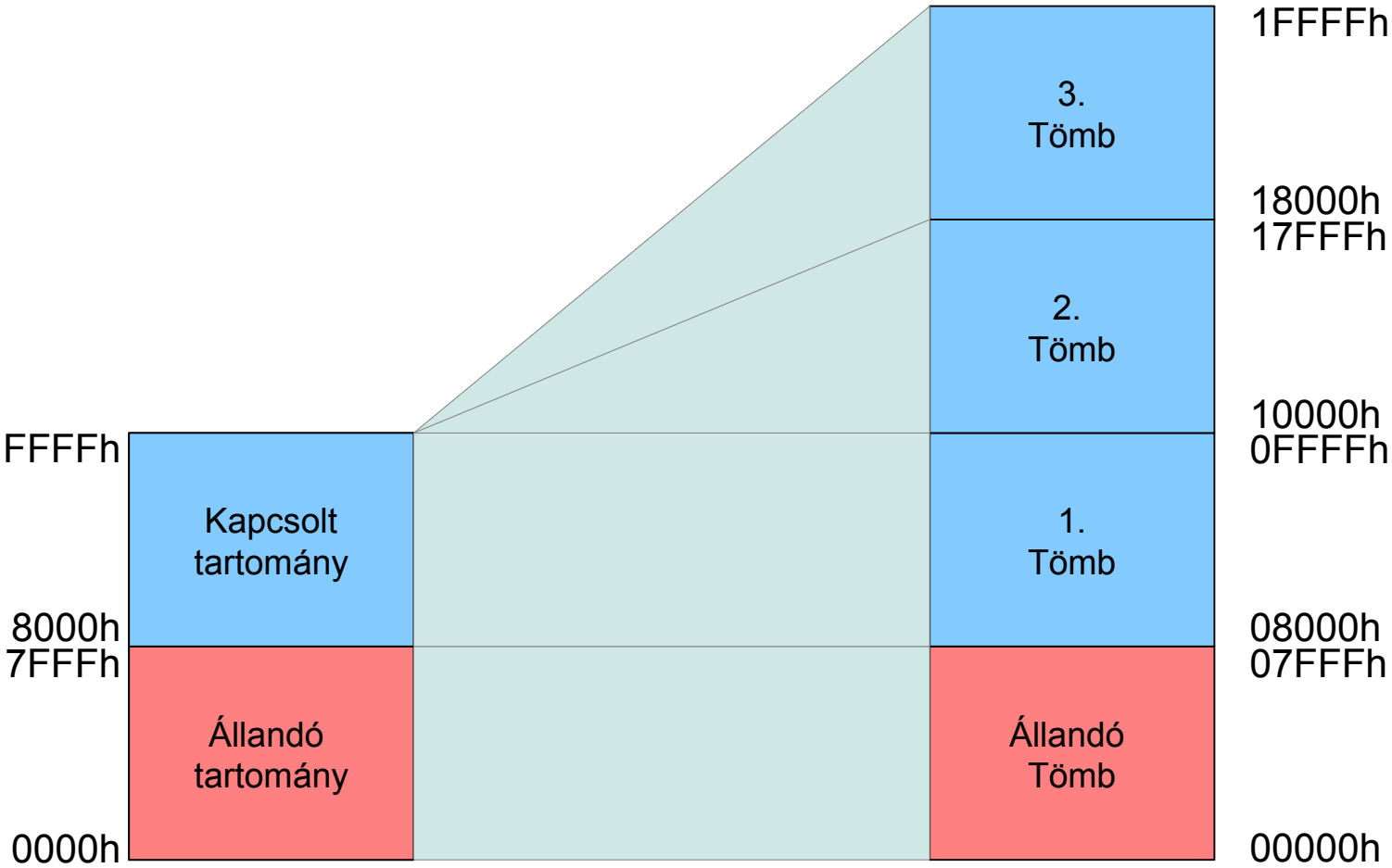


Tárbővítés tömbkapcsolással

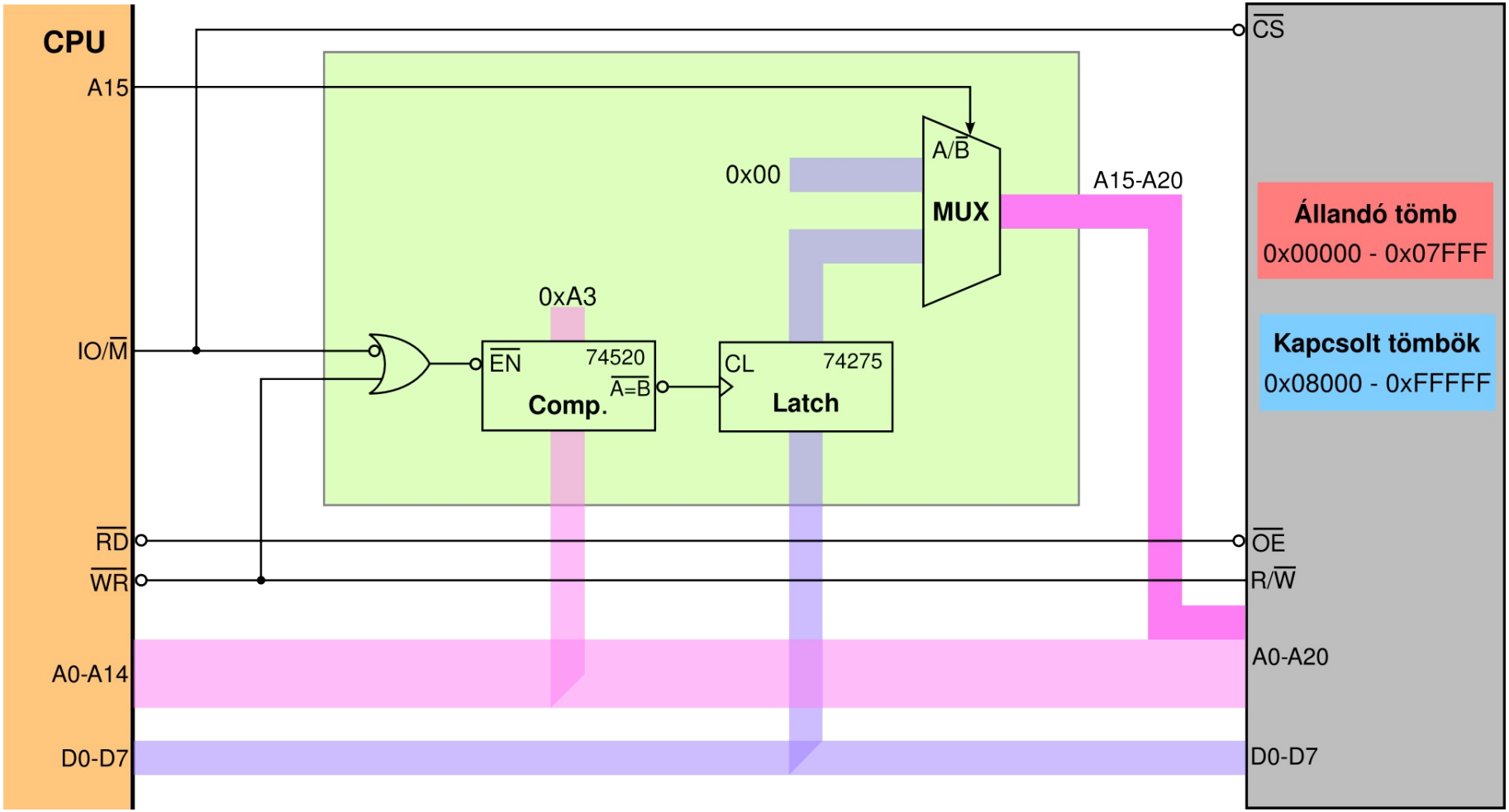
- Megvalósíthatjuk egyetlen nagy memóriamodullal is
  - A memóriacím alsó része: a CPU által kiadott cím
  - A memóriacím felső része: a tömbkapcsoló adja
    → aktív tömbnek megfelelően



Tárbővítés tömbkapcsolással



Tárbővítés tömbkapcsolással



Tárbővítés tömbkapcsolással

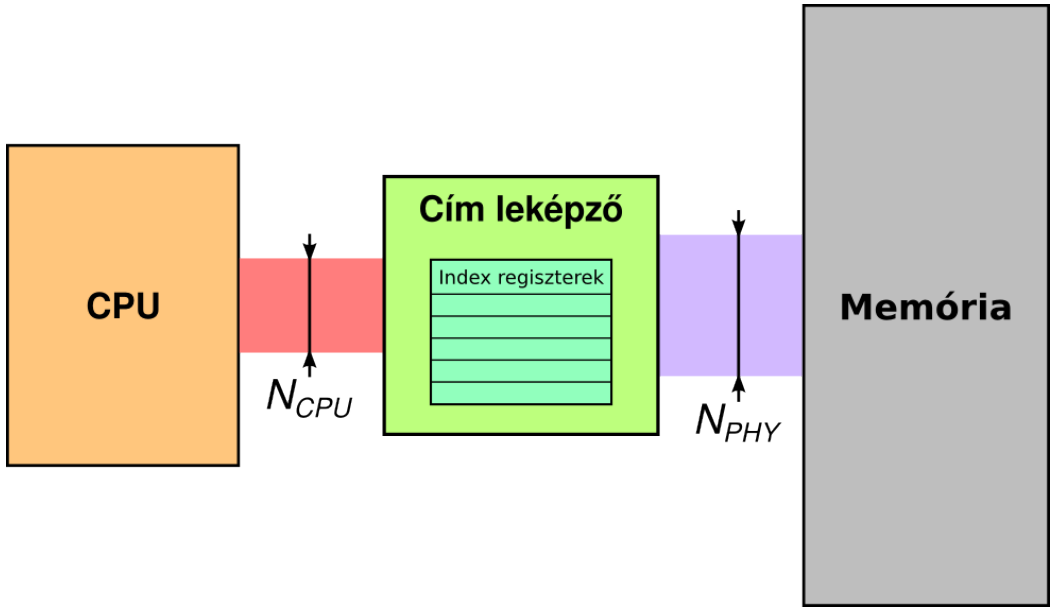
- Értékelés:
  - egyszerű megvalósítás
  - gyors működés
  - merev felépítés: a tömbök mérete azonos
    - A konkrét problémához, programhoz, adatszerkezetekhez másfajta particionálás lehet szerencsésebb
  - nem transzparens:
    - A programot a tömbök számának és méretének konkrét ismeretében kell megírni
    - Ha a tömbök méretét megváltoztatjuk, a programot is meg kell változtatni

Tárbővítés indexelt leképzéssel

- Tömbkapcsolás:
  - Teljes tömbök ki/be kapcsolása
- Indexelt leképzés:
  - Finomabb működés: ablakok mozgatása a nagy memórián
  - Ablakok méretének összege = CPU címtartománya
  - Ablakok kezdőcíme: ***indexregiszterekben***

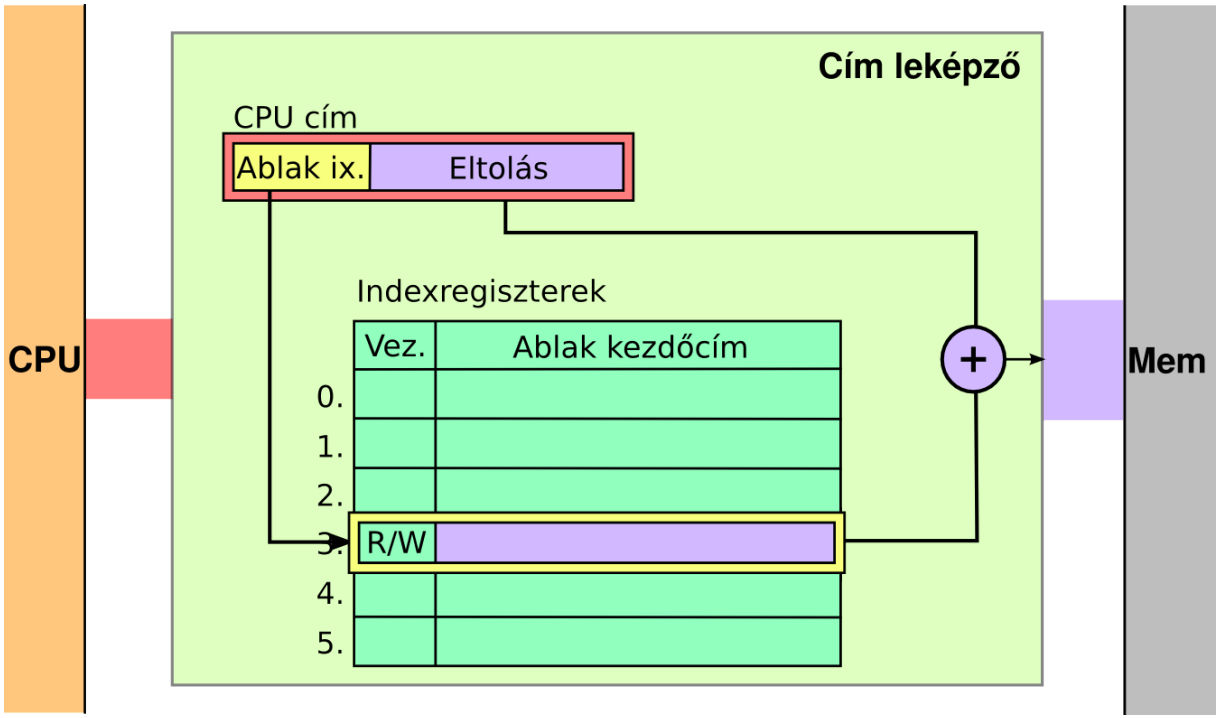
Tárbővítés indexelt leképzéssel

- Indexregiszterek:
  - Cím leképző perifériában
  - Perifériaműveletekkel írhatók/olvashatók
  - Ablak kezdőcím, jogosultságok, stb.
- Cím leképző periféria:
  - CPU cím (keskeny) ↔ Fizikai cím (széles)

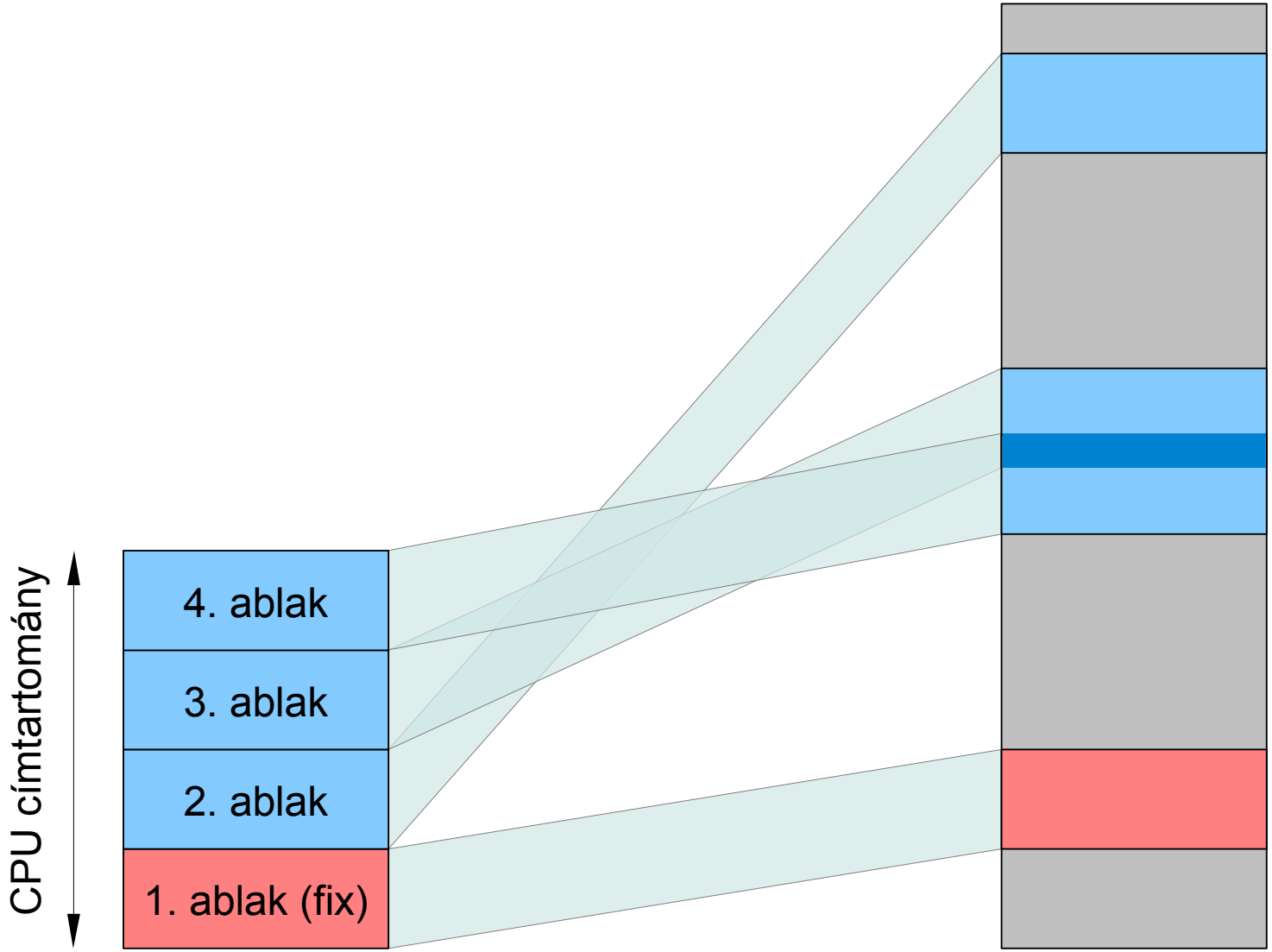


Tárbővítés indexelt leképzéssel

- Címleképző periféria:
  - A CPU címet ablak *sorszám*ra és *eltolás*ra bontja
  - Indexregiszterekből: ablak sorszám → *kezdőcím* a nagy memóriában
  - Fizikai memória címe = kezdőcím + eltolás



Tárbővítés indexelt leképzéssel



Tárbővítés indexelt leképzéssel

- Értékelés:
  - Tömbkapcsolásnál összetettebb megvalósítás
  - Egy összeadás művelet bele kell férjen a memóriaciklusba
  - Ablakok mérete azonos
  - Ez sem transzparens:
    - A programot az ablakok számának és méretének konkrét ismeretében kell megírni
    - Ha ezek változnak, a programot is meg kell változtatni

Memóriakezelés

- Volt: kicsi a CPU címtartománya
  - Tárbővítés
    - Tömbkapcsolás
    - Indexelt leképzés
- Másik alapprobléma: nagy a CPU címtartománya

Virtuális tárkezelés

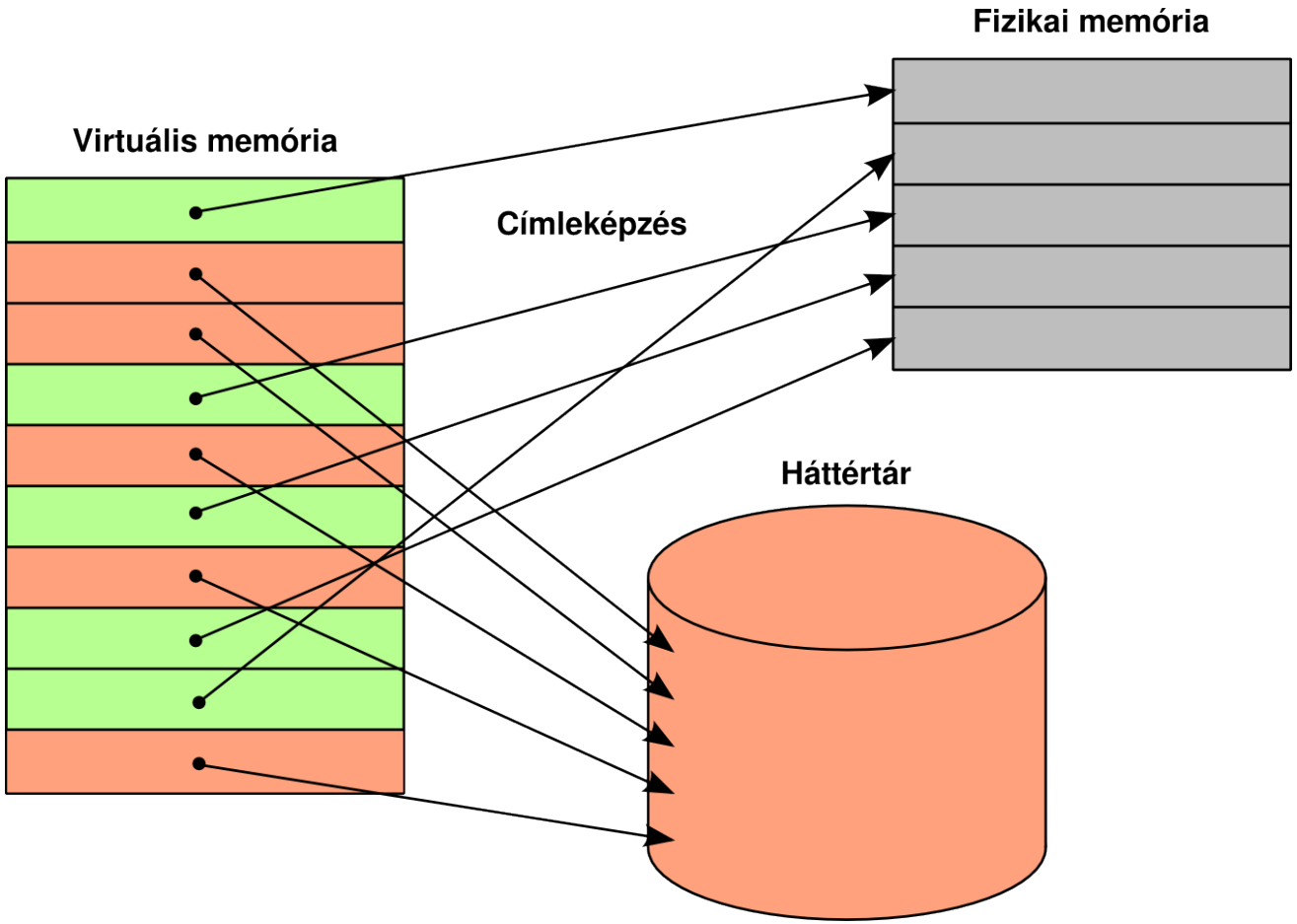
- Motiváció:
  - Olcsó 64 bites processzorok
    - Ebből pl. az AMD64 48 bitet használ: 256 TB
    - Ennyi memóriát nem tudunk hozzáilleszteni
    - Amennyi van, szűkös lehet (multitaszking)
  - Programoknak ne legyen gondja a fizikai memóriaméret
    - Hadd higgyék, hogy a teljes címtartományt lefedtük
- A CPU a programoknak a teljes címtartományt felajánlja
- Ezt a „virtuális” memóriát kell leképezni a fizikaiba

  → ***Virtuális tárkezelés***

Virtuális tárkezelés

- Programok: ***Virtuális címeket*** használnak
- CPU lábain/buszon: ***Fizikai címek*** jelennek meg
- Virtuális → fizikai cím leképzés: ***Címfordítás***
  - Aki csinálja: ***MMU***
  - Egysége: ***Lap*** (fix) vagy ***szegmens*** (változó méretű)
- Ha nem fér mindenki a fizikai memóriába
  → háttértárra kerül (swap-elés)

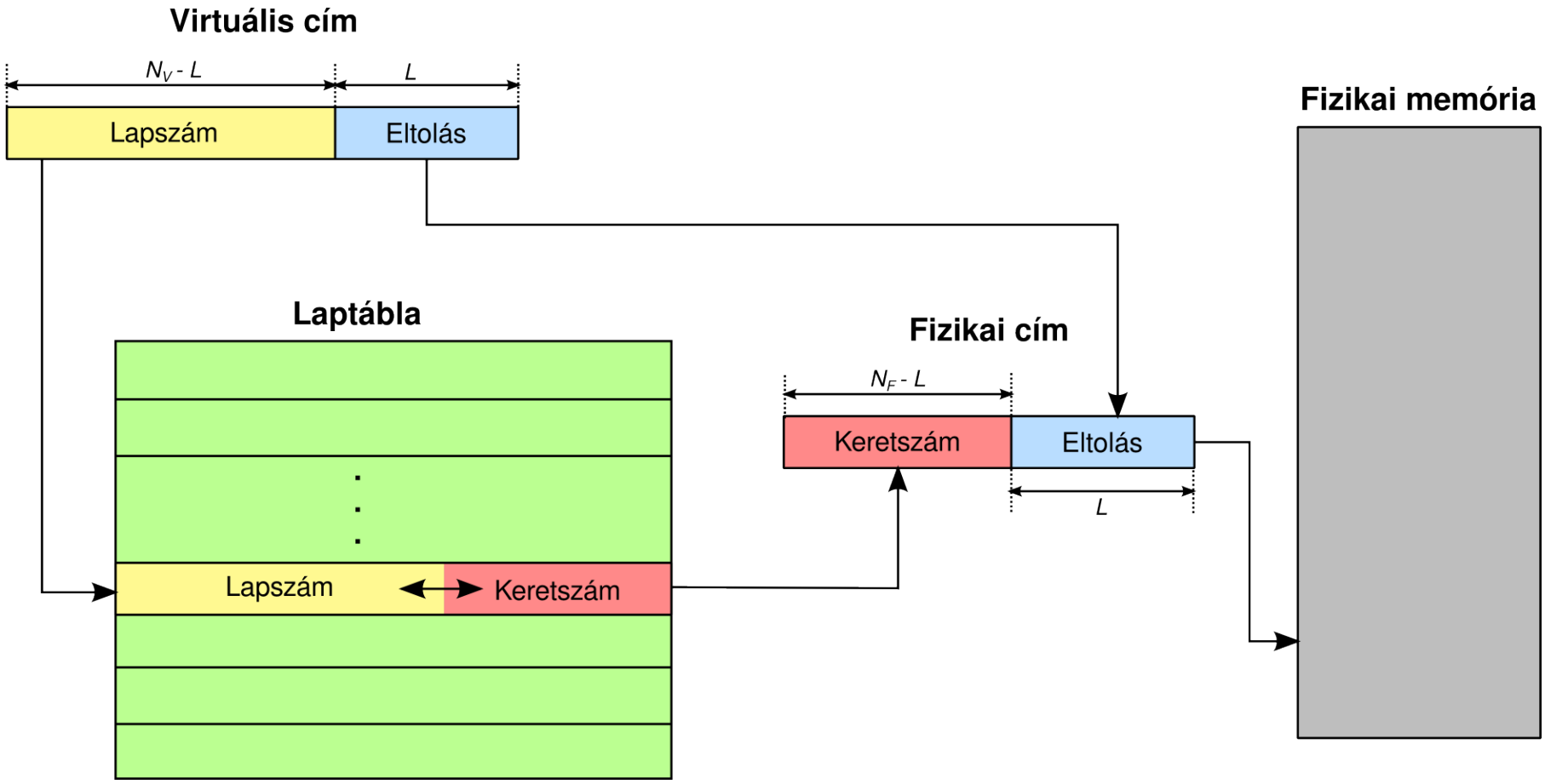
Virtuális tárkezelés



Címfordítás

- A virtuális címtartományt fix méretű lapokra particionáljuk
- A fizikai címtartományt ugyanekkora ***keret***ekre osztjuk
- Méretek:
  - Lapok mérete = 2^L
  - Címekből alsó L bit: lapon belüli eltolás
  - Felső bitek: virt. címeknél lapsorszám, fiz. címeknél keretsorszám
- Lap ↔ keret összerendeléseket tárolja: ***laptábla***
- Tartalma:
  - Lap sorszáma
  - Keret sorszáma
  - Védelmi információ (írható/olvasható)
  - Vezérlő bitek:
    - Valid
    - Dirty

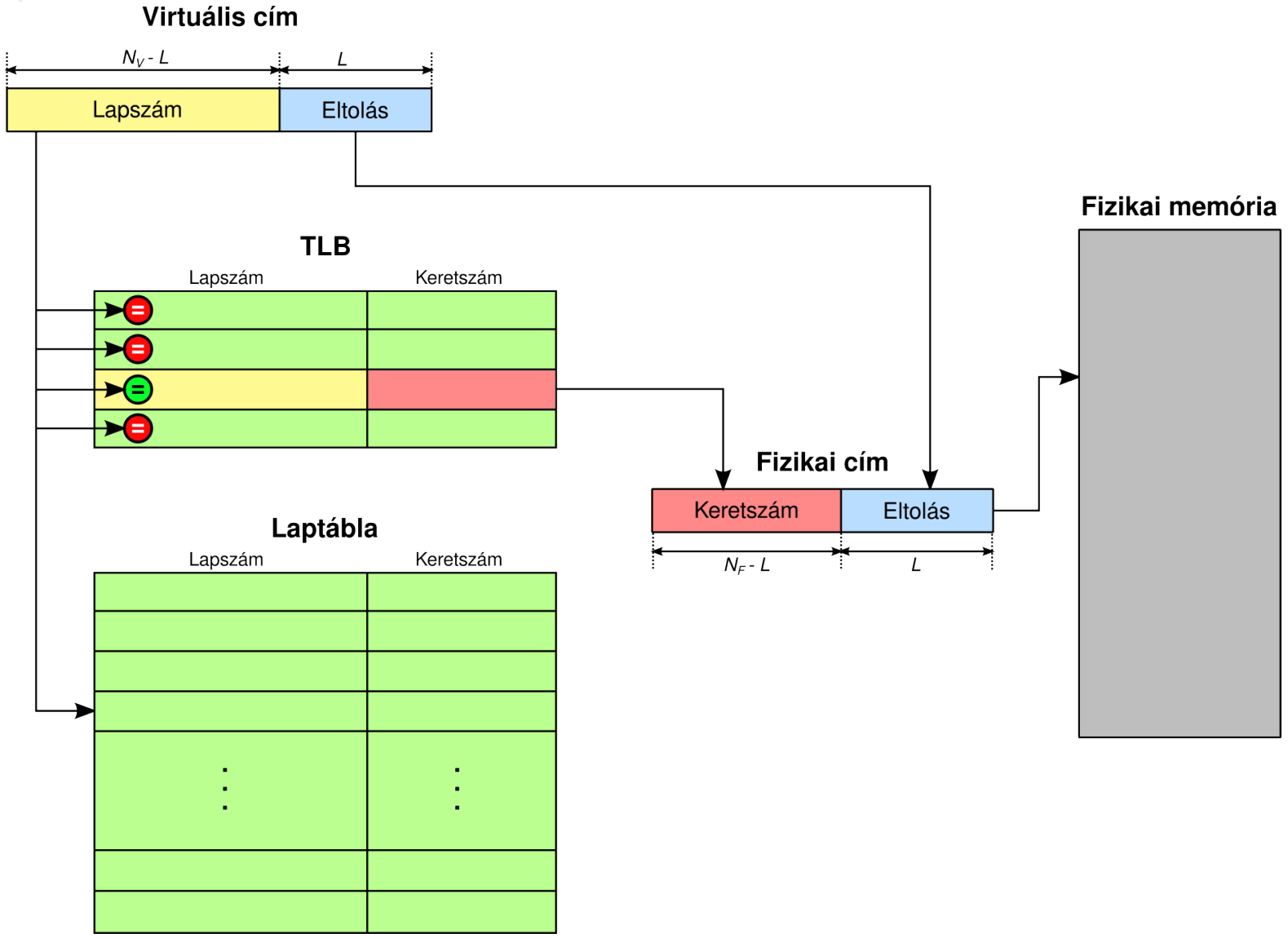
Címfordítás



TLB

- Memóriahozzáférés menete:
  - Címfordítás: Fizikai cím előállítása a virtuálisból
  - Tényleges adat olvasás/írás a fizikai címről/címre
- A laptábla is a fizikai memóriában van!
- 1 memóriaművelet a programban
  → 2 memóriaművelet a valóságban !!
- Nem elég lassú a memória e nélkül is?
- De.
- Reménység: lokalitás
- ***TLB***: translation lookaside buffer
  - A leggyakrabban használt virtuális ↔ fizikai összerendeléseket a CPU egy speciális cache-ben tárolja
  - Címfordításkor először a TLB-ben keres

Címfordítás TLB-vel



TLB jellemzői

- TLB lefedettség: a TLB bejegyzések által lefedett címtartomány
  - Minél nagyobb, annál ritkábban kell a lassú laptáblához nyúlni
- TLB megvalósítása: **tartalom szerint címezhető memória**

  → Nagy a felülete és sokat fogyaszt

  → Kicsi :(

TLB méret:	16 – 512 bejegyzés
Találat ideje:	0.5 – 1 órajel
TLB hiba esetén címfordítási idő:	10 – 100 órajel
TLB hibaarány:	0.01% – 1%

Laptábla implementációk

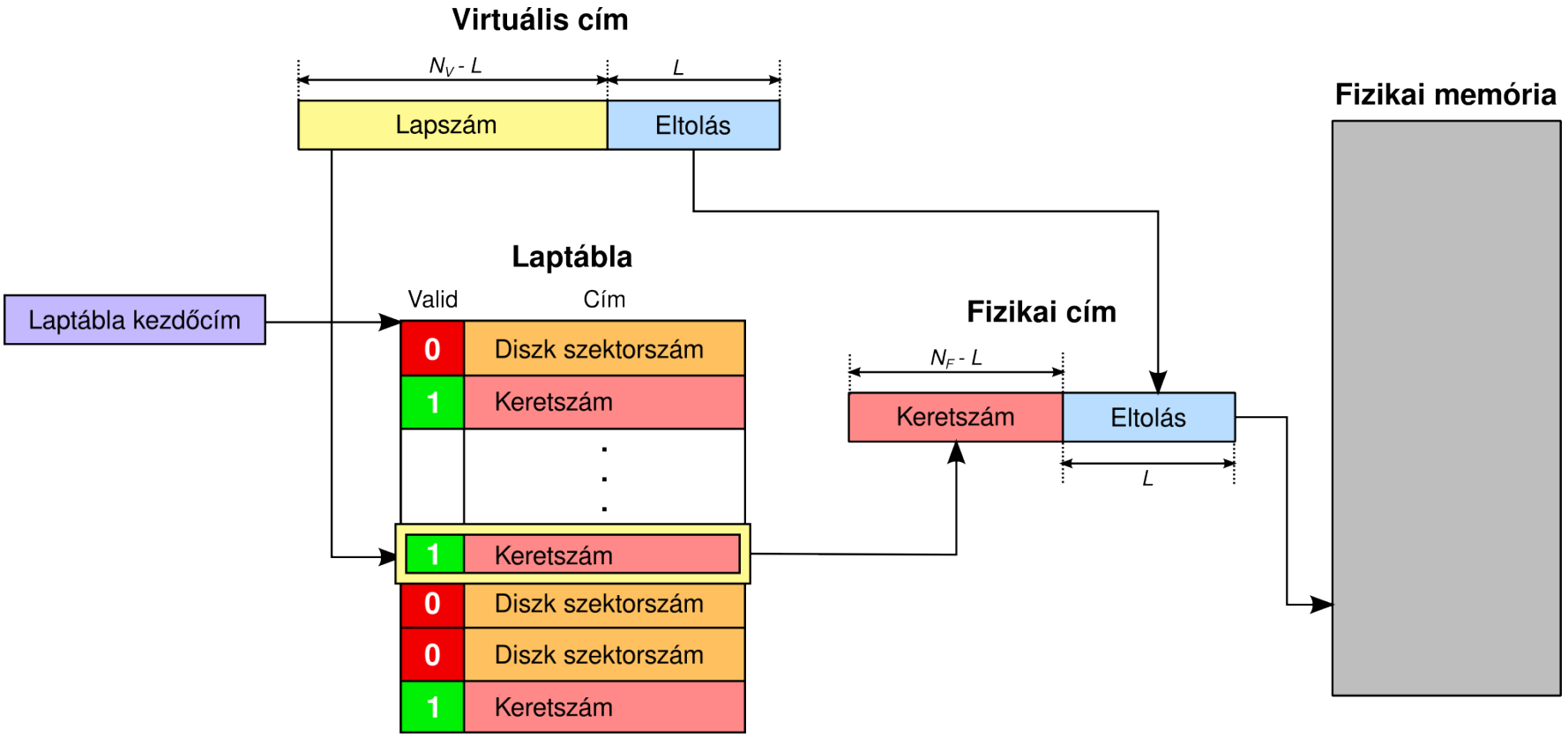
- Cél:
  - Címfordítás legyen minél gyorsabb

    → virtuális cím szerinti keresés legyen gyors

    → minél kevesebbszer kelljen a memóriához nyúlni
  - Laptábla legyen a lehető legkisebb

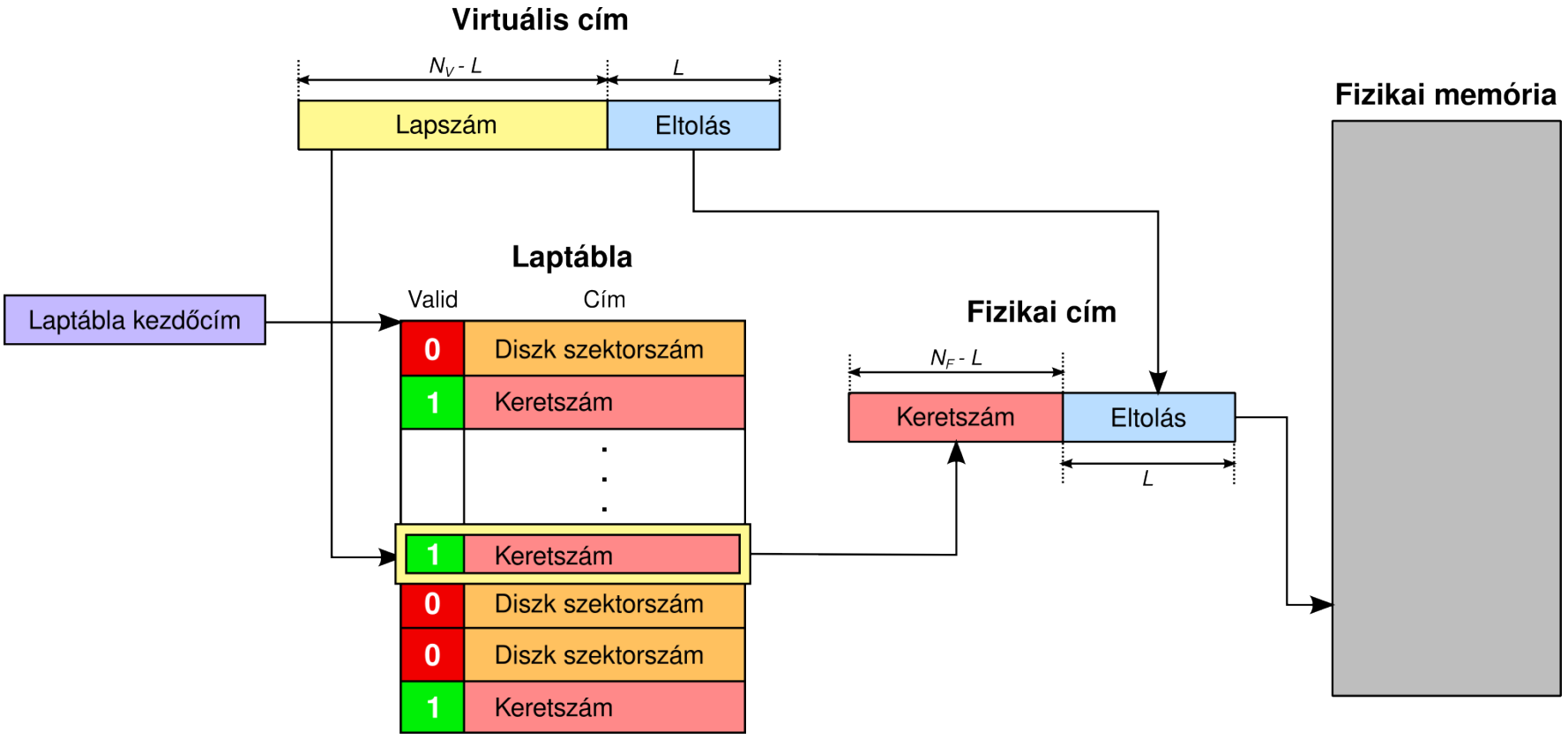
Egyszintű laptábla

- Laptáblabejegyzések tömbje
- i. bejegyzés: i. laphoz tartozó keret



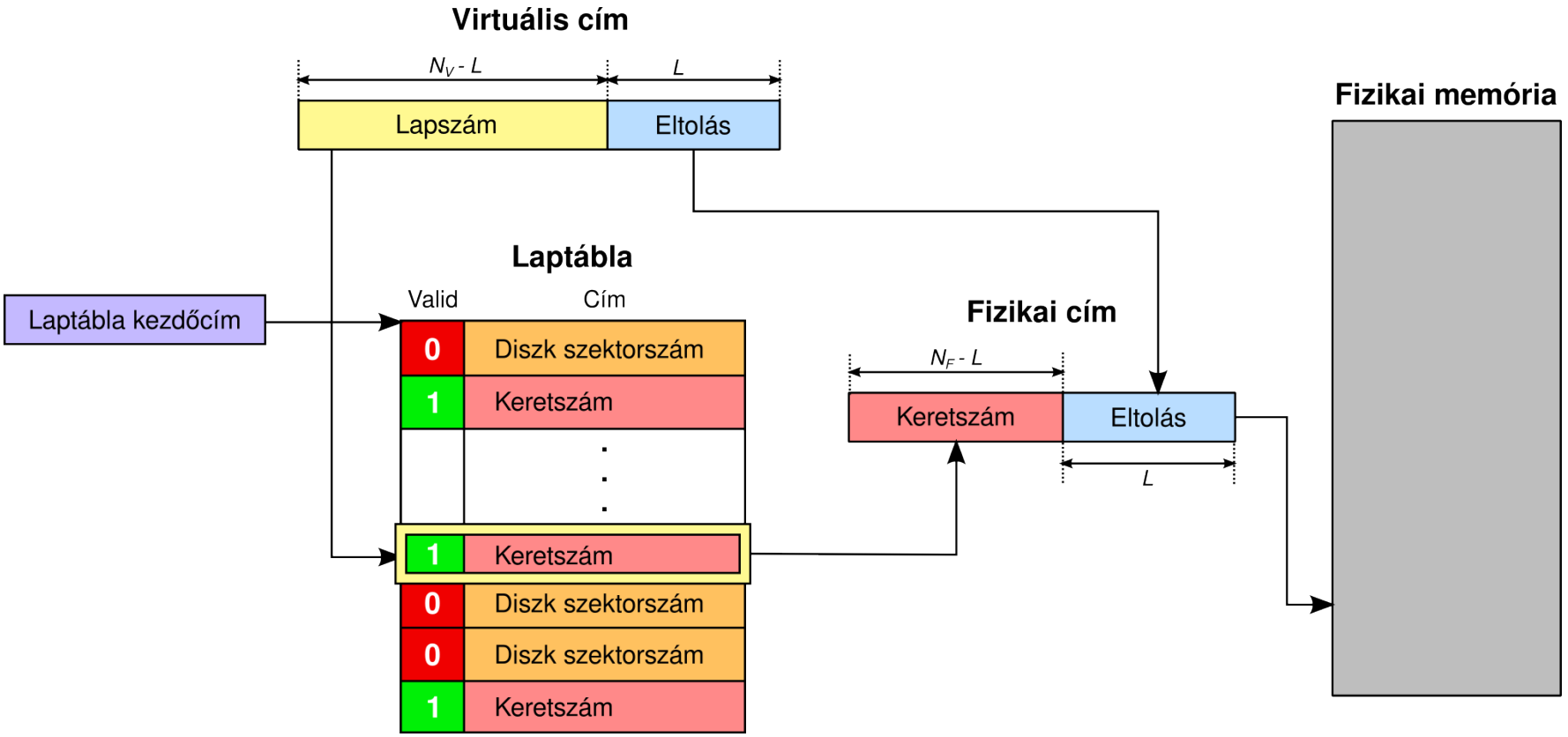
Egyszintű laptábla

- Keresés: gyors!
- Bejegyzés megtalálása: pontosan 1 memóriaművelet!



Egyszintű laptábla

- Bejegyzés mérete: kicsi
  - Nem kell tárolni a virtuális címet → az pont az index
  - Nem kell külön tárolni a háttértáron való elhelyezkedést



Egyszintű laptábla

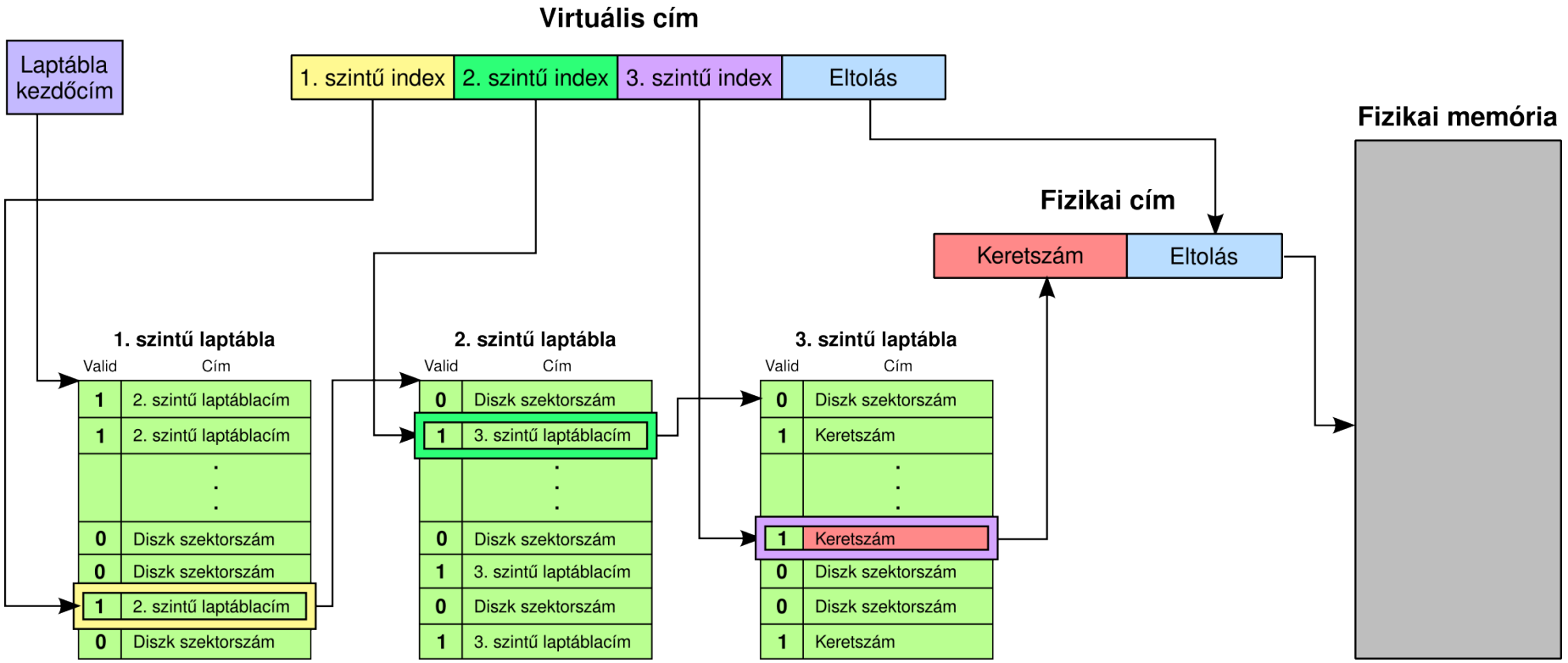
- Gyors, kevés memória-hozzáférés kell, kicsi a bejegyzés...
  ...miért nem használ mindenki ilyet?

A teljes laptáblának bent kell lennie a memóriában!

- Gyors kalkuláció:
  - 32 bites esetben, 4 kB lapokkal:
    - Lapméret: 12 bit, lapok száma: 32-12 = 20 bit, 1 mega-lap
    - 1 bejegyzés: 4 bájt elég, laptábla mérete: 4 MB
  - 64 bites esetben, 4 kB lapokkal:
    - Lapméret: 12 bit, lapok száma: 64-12 = 52 bit, 2^{52} db lap
    - 1 bejegyzés: 8 bájt, laptábla mérete: 8 * 2^{52} = 32 PB

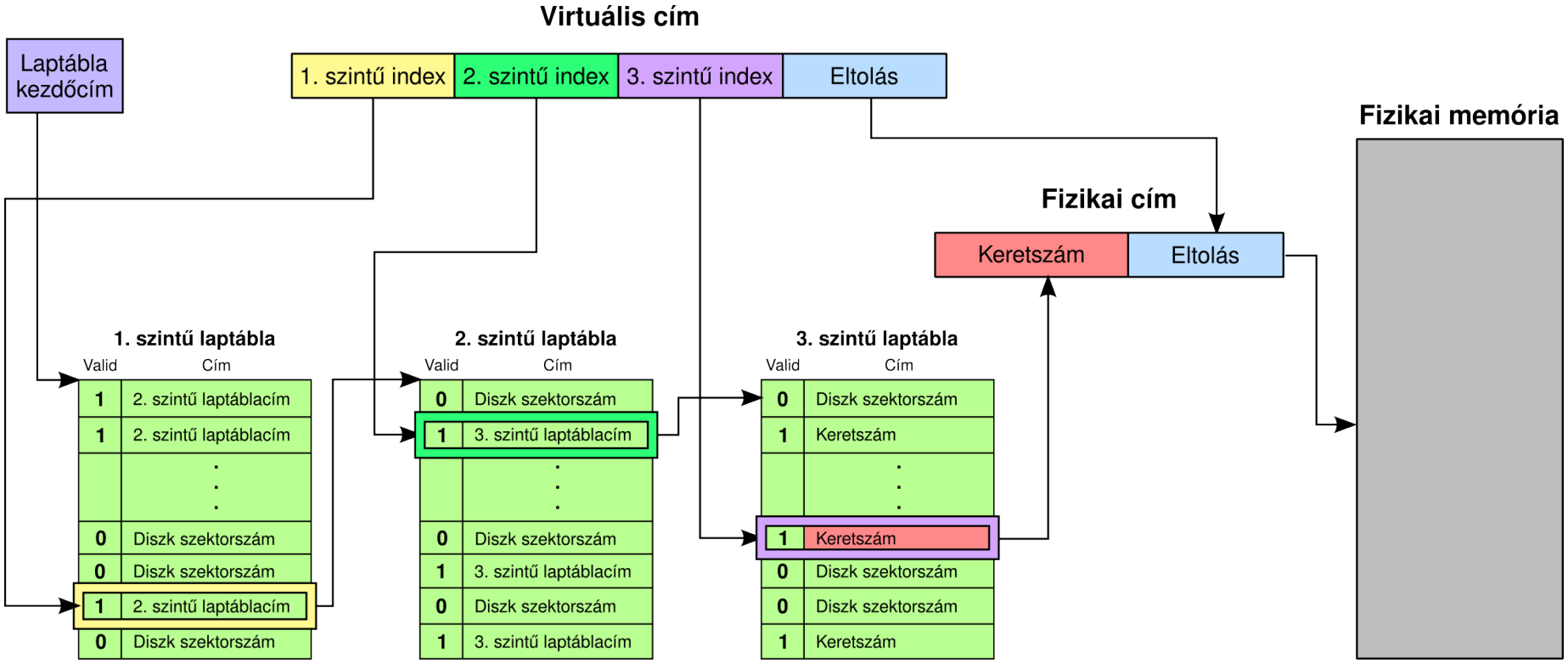
Hierarchikus laptábla

- Ötlet: magát a laptáblát is lapokra bontjuk
- A laptábla lapok elhelyezkedését egy másik laptáblában tároljuk, azokét egy harmadikban, stb.

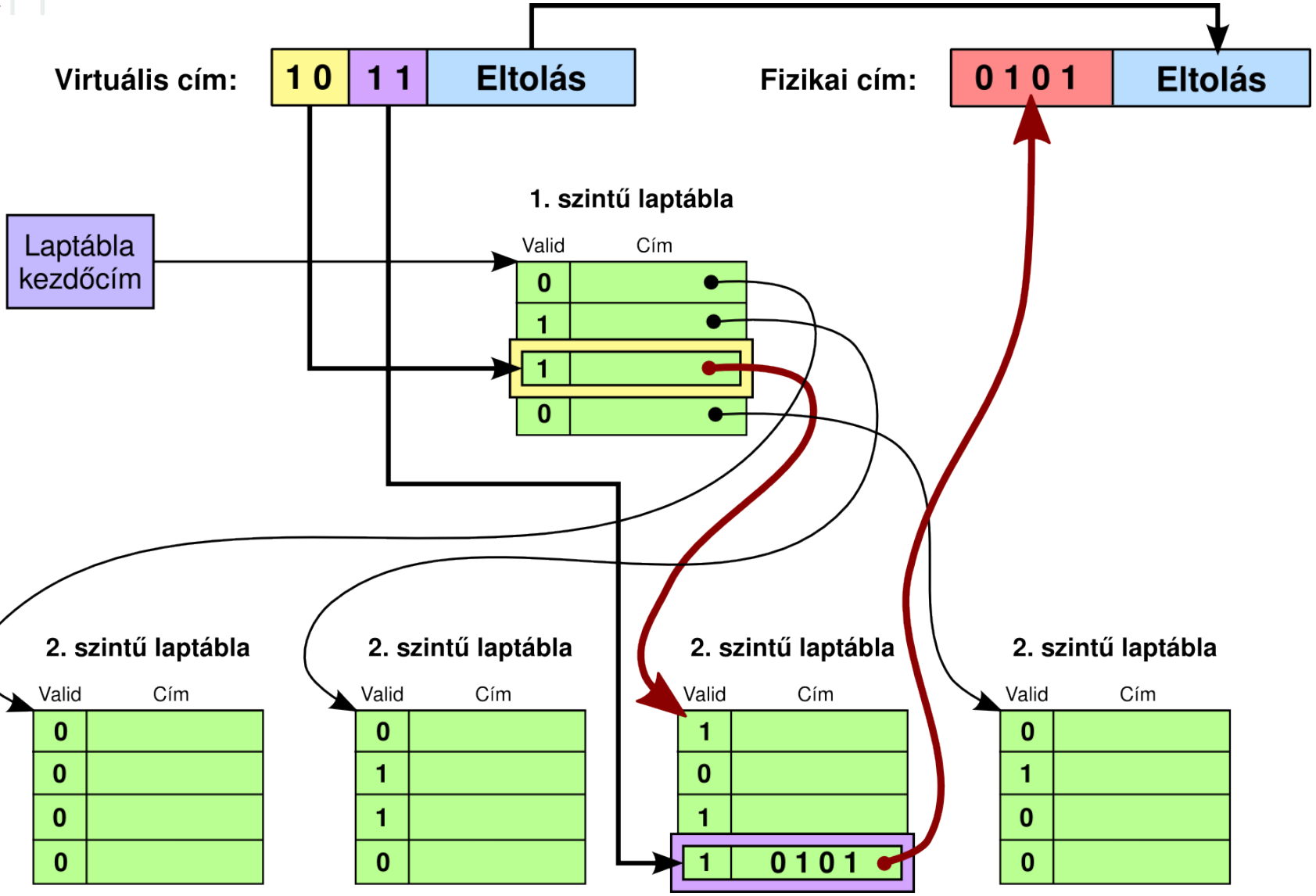


Hierarchikus laptábla

- Csak azt tartjuk a memóriában, ami tényleg kell
- Több memóriaművelet kell a címfordításhoz → lassú!



Kétszintű laptábla, példa



Hierarchikus laptábla

- Gyors kalkuláció:
  - 32 bites esetben, 4 kB lapokkal:
    - Lapméret: 12 bit, 1 bejegyzés: 4 bájt elég
    - Egy lapra 1024 bejegyzés fér → 10 bittel indexelhető
    - Kell 1024 laptábla lap, és még egy lap, ami ezekre mutat
    - 32 bites cím = 10 + 10 + 12 bites részek

      → 2 szintű laptábla kell
  - 64 bites esetben, 4 kB lapokkal:
    - Lapméret: 12 bit, 1 bejegyzés: 8 bájt
    - Egy lapra 512 bejegyzés fér → 9 bittel indexelhető
    - 64 bites cím = 7 + 9 + 9 + 9 + 9 + 9 + 12 bites részek

      → 6 szintű laptábla kell!

      6 memóriaművelet / címfordítás !!!
- X86, ARM

Virtualizált laptábla

- Olyan egyszerű és gyors volt az egyszintű laptábla...
- Túl nagy? Nem fér el? Tegye a virtuális memóriába!
- Címfordítás:
  - Számoljuk ki a ***laptábla bejegyzés virtuális címét***

    = tábla kezdőcím + bejegyzésméret * lapsorszám
  - Ha a bejegyzés címe a TLB-ből feloldható → bingó!

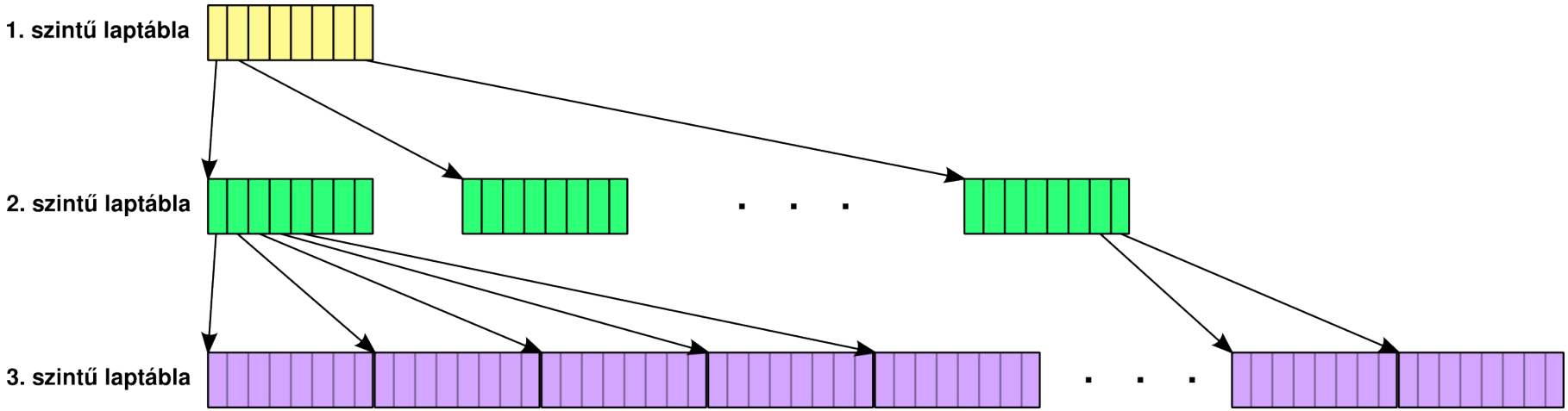
    Kiolvassuk a bejegyzést a memóriából, majd a címfordítás után jön a tényleges memória-hozzáférés

    → 1 memóriaművelet / címfordítás
  - Ha nincs TLB találat: jöhet a hierarchikus laptáblánál látott lassú megoldás

    → 4/5/6 memóriaművelet / címfordítás

Virtualizált laptábla

- Mi kell hozzá?
  - Laptábla bejegyzések egyetlen tömbben
  - Felette a hierarchia, ha nincs szerencsénk, azzal keresünk



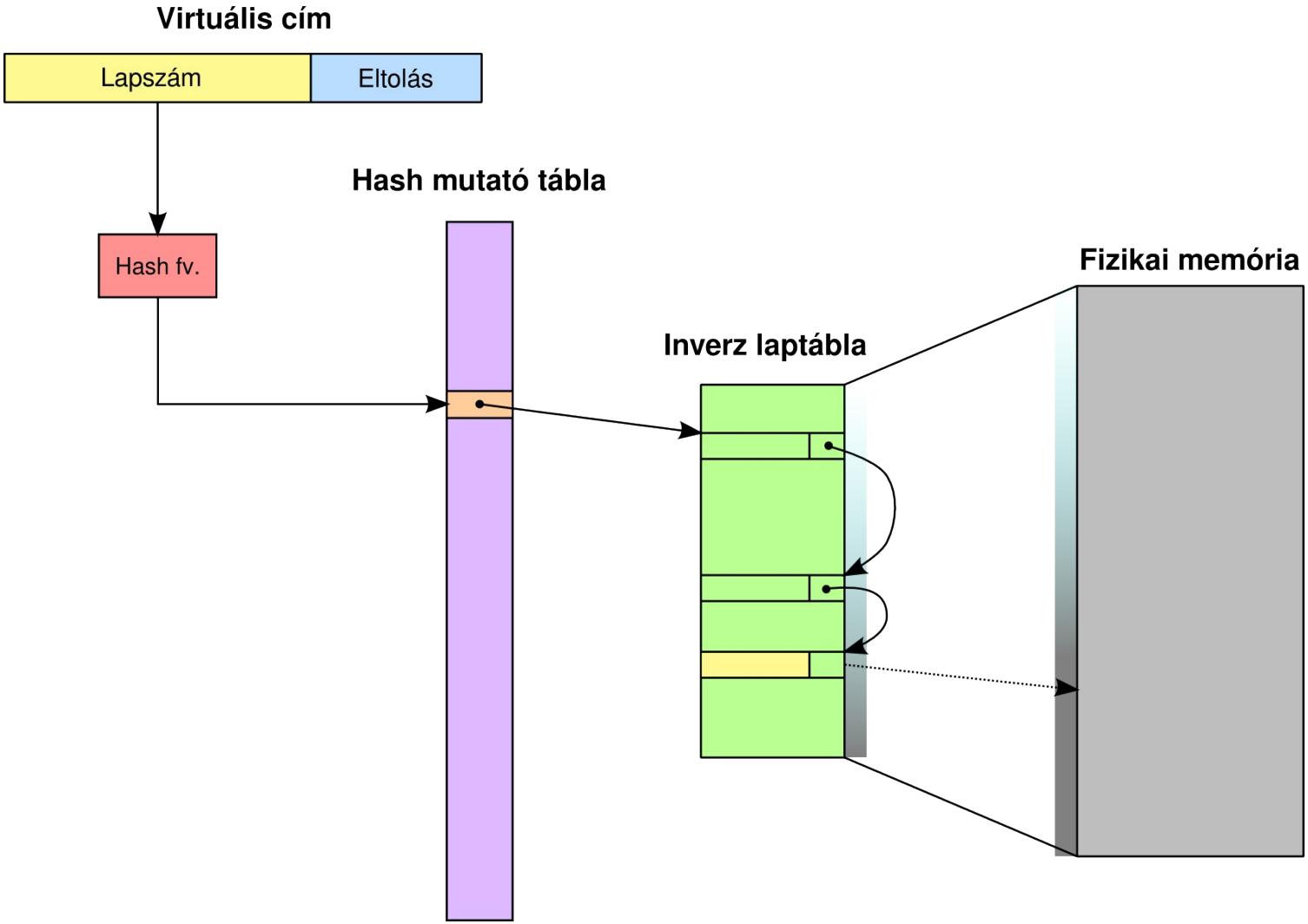
- MIPS, Alpha

Inverz laptábla

- Miért is voltak nagyok az eddigi laptáblák?
  - Összerendelés: i. bejegyzés = i. laphoz tartozó keret
  - Elemszám = lapok száma
- Fordított gondolkodás:
  - Összerendelés: i. bejegyzés = i. keretben lévő lap száma
  - Elemszám = keretek száma
  - → ***Inverz laptábla***
- Mit nyerünk?
  - Kicsi lesz a laptábla (fizikai memória méretével arányos)
- Mit vesztünk?
  - Adott lapsorszámhoz tartozó keretet keressük
  - A tömbünk indexei keretsorszámok, a bejegyzés *tartalma* a lap száma

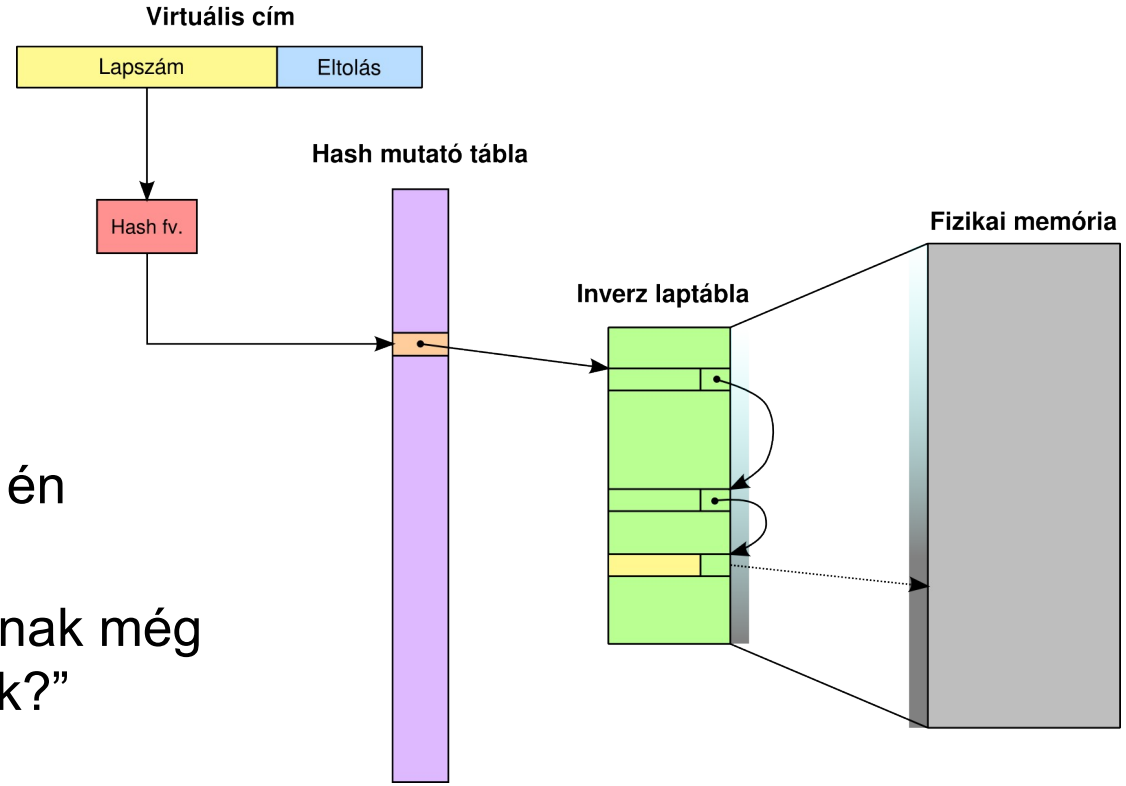
    → Tartalom szerint kell keresnünk!
- Tartalom szerinti kereséshez: hash függvény

Inverz laptábla



- Pl. jó hash függvény: Lapsorszám alsó bitjei

Inverz laptábla



- Hash mutató táblában:
  - Pointer:

    „Hol lehetnek az ilyen lapok a laptáblában?”
- Laptáblában:
  - „Ebbe a bejegyzésbe én vagyok írva?”
  - „Nem? Pech. Hol vannak még hozzám hasonló lapok?”
    „Követem a pointert”
  - „Ebbe a bejegyzésbe már én vagyok írva?”
  - „Nem. Nem baj, megyek tovább. Talán itt?”
  - „Igen? Szuper. Hányadik bejegyzésen állok? 329?”
  - „Akkor a fizikai memória 329-es keretében vagyok elhelyezve!”

Inverz laptábla

- Mitől függ a memóriaműveletek száma?

  → ahány lépést kell tenni a laptábla láncolt listájában
- És az mitől függ?

  → Attól, hogy mennyire van tele a memória
- Normál terhelés (nincs tele a memória)

  → 1 memóriaművelet / címfordítás
- Nagy terhelés (megtelt a memória)

  → >1 memóriaművelet / címfordítás

  De akkor már úgyis mindegy (swap-eléstől lassú a rendszer)
- PA-RISC, PowerPC

Méretezési kérdések

- Mekkora lapokat használjunk?
  - Nagyot, mert
    - Nagyobb a TLB lefedettség, kevesebb a laphiba
    - Diszk háttértár nagyot és kicsit kb. egyforma ideig tölt be
      - Akkor már vigyünk be minél nagyobb darabot
  - Kicsit, mert
    - A lapon csak az van, ami kell.
    - És ami nem kell, ne foglalja már a drága, kicsi memóriát
  - Gyakorlatban: 4 – 8 kB

Virtuális tárkezelés a gyakorlatban

x86

- Szegmentálás + lapozás, gyakorlatban inkább csak lapozás
- Hierarchikus laptábla. Alap: 4 kB-os lapok
- Módok:
  - 32 bit, PAE nélkül:

    12 bit eltolás, 10 bit másodszintű, 10 bit első szintű laptábla index
  - 32 bit, PAE:
    - 36 bites fizikai címek támogatása (64 GB, de csak 4 GB / program!)
    - 8 bájtos laptábla bejegyzések: 512/lap, 3 szint kell!

      12 bit eltolás, 9 bit harmad, 9 bit másod, 2 bit első szintű laptábla index
  - 64 bit:
    - 48 bites virtuális, 40 bites fizikai címek (bővíthetők 64-ig)
    - 8 bájtos laptábla bejegyzések: 512/lap, 4 szint kell!

      12 bit eltolás, 9 bit negyed, 9 bit harmad, 9 bit másod, 2 bit első szintű laptábla index
- Nagy lapok: 1 szint megspórolható

Virtuális tárkezelés a gyakorlatban

ARM

- Hierarchikus laptábla
- Vegyesen használható lapméretek:
  - 4 kB (small page)
  - 64 kB (large page)
  - 1 MB (section)
  - 16 MB (super section)
- Section / supersection: 1 szintű címfordítás
- Small page / large page: 2 szintű címfordítás

Virtuális tárkezelés a gyakorlatban

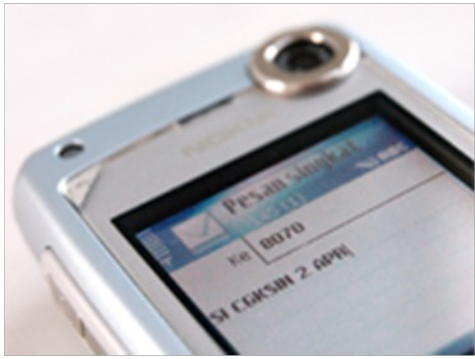
UltraSPARC

- *Szoftver menedzselt TLB*
  - Laptábla teljesen az op. rendszer kezelésében
  - Címfordítás:
    - CPU a TLB-ből próbálkozik,
    - TLB hiba: oprendszerre bízza a címfordítást!
    - A hozzárendelést az oprendszer a TLB-be írja
- Oprendszer kezeli:
  - A TLB tartalmát
  - A laptábla bejárást (tetszőleges adatszerkezet lehet)
- Solaris Unix: Inverz laptáblát használ
- SPARC Linux: Hierarchikus laptáblát használ

Virtuális tárkezelés a gyakorlatban

PowerPC

- Inverz laptábla, de láncolt lista nélkül
- 2 tömböt használ:
  - (Inverz) laptábla: lap ↔ keret összerendelések
  - Hash tábla:
    - Laphoz a hash függvény → 2 hash tábla bejegyzés
    - 1 hash tábla bejegyzés: 8 lapszám ↔ laptábla index összerendelés
    - A lapszám a 16 laptábla bejegyzés egyikében kell legyen
    - Ha nincs → laphiba (op. rendszer kezeli)



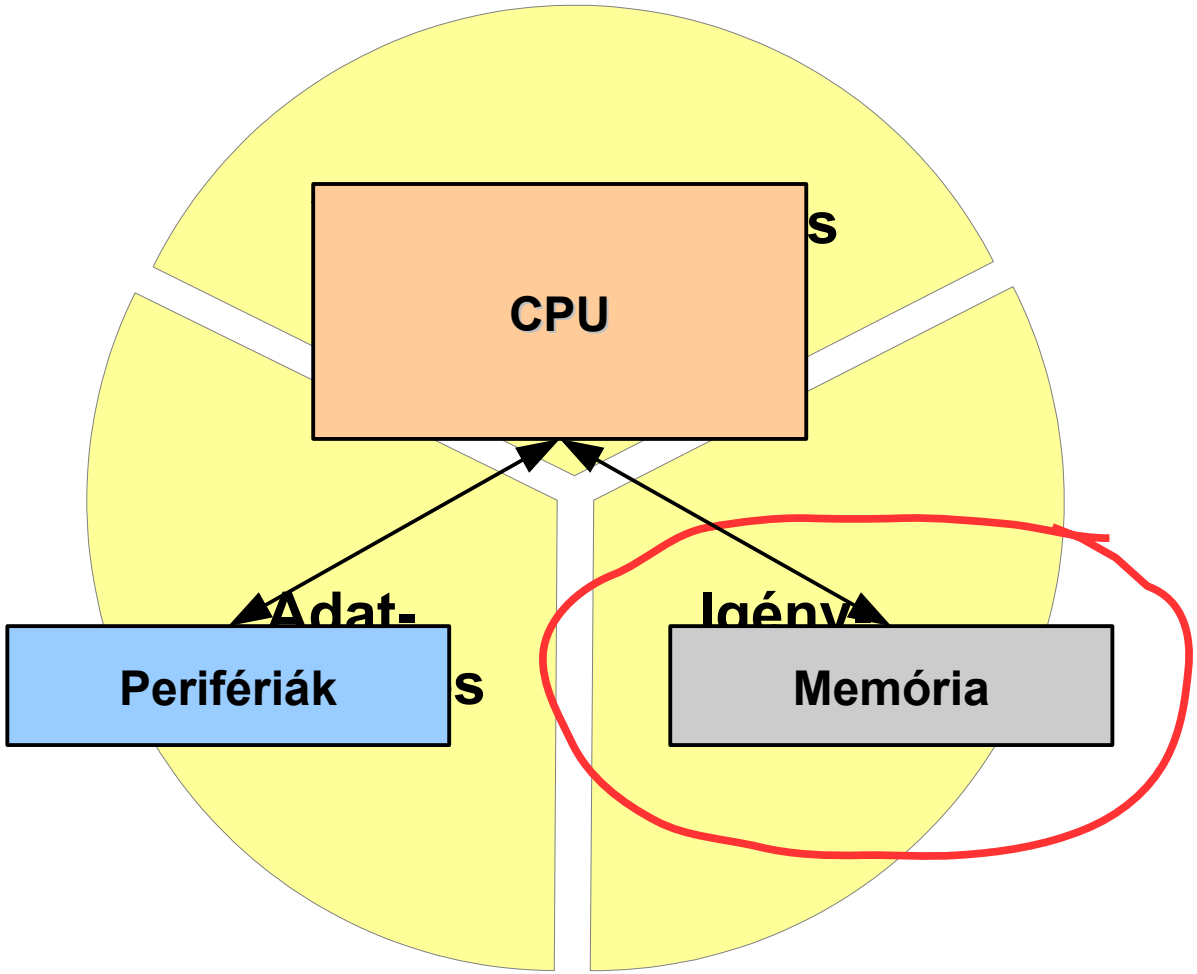
Számítógép Architektúrák

Cache memória

Horváth Gábor
docens
BME Híradástechnikai Tanszék
ghorvath@hit.bme.hu

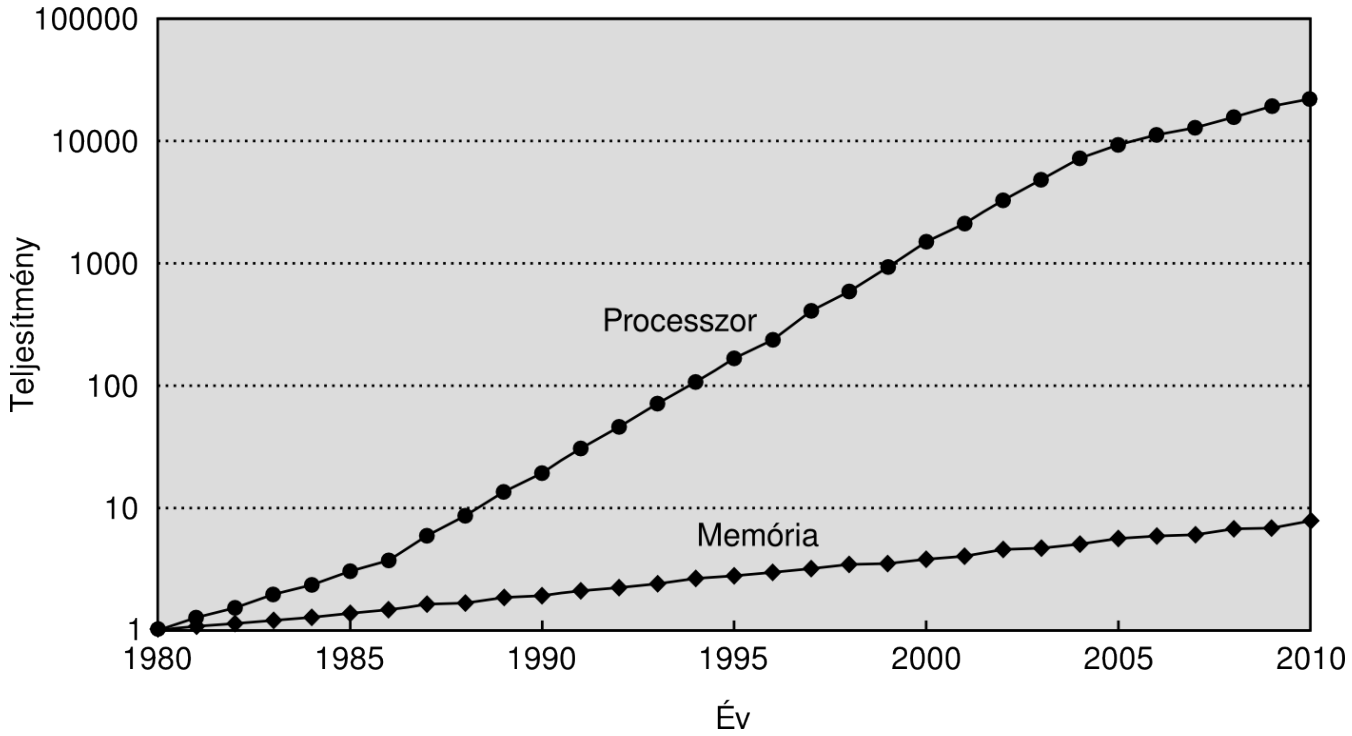
2012. november 5.
Budapest

Hol tartunk?



Már megint a memória...

- Mindenről a memória tehet.
- Mert lassú.



- A virtuális tárkezelés tetézi a bajt
  - 1 memóriabeli objektum elérése → több memóriaművelet

Reménysugár

- A programok nem véletlenszerűen nyúlkálnak a memóriába
- Sokszor speciális mintázatot követnek

  → kihasználható!
- Lokalitási elvek:
  - Időbeli: egyszer hozzányúltunk, többször is hozzá fogunk
  - Térbeli: ha hozzányúltunk, a környezetéhez is hozzá fogunk
  - Algoritmikus: speciális adatszerkezet kiszámítható bejárása
- Példa:
  - Médialejátszás:
    - Térbeli lokalitás: igen, időbeli lokalitás: nem
  - Ciklusszervezés:
    - A ciklusmag kódjára teljesül a térbeli és időbeli lokalitás is

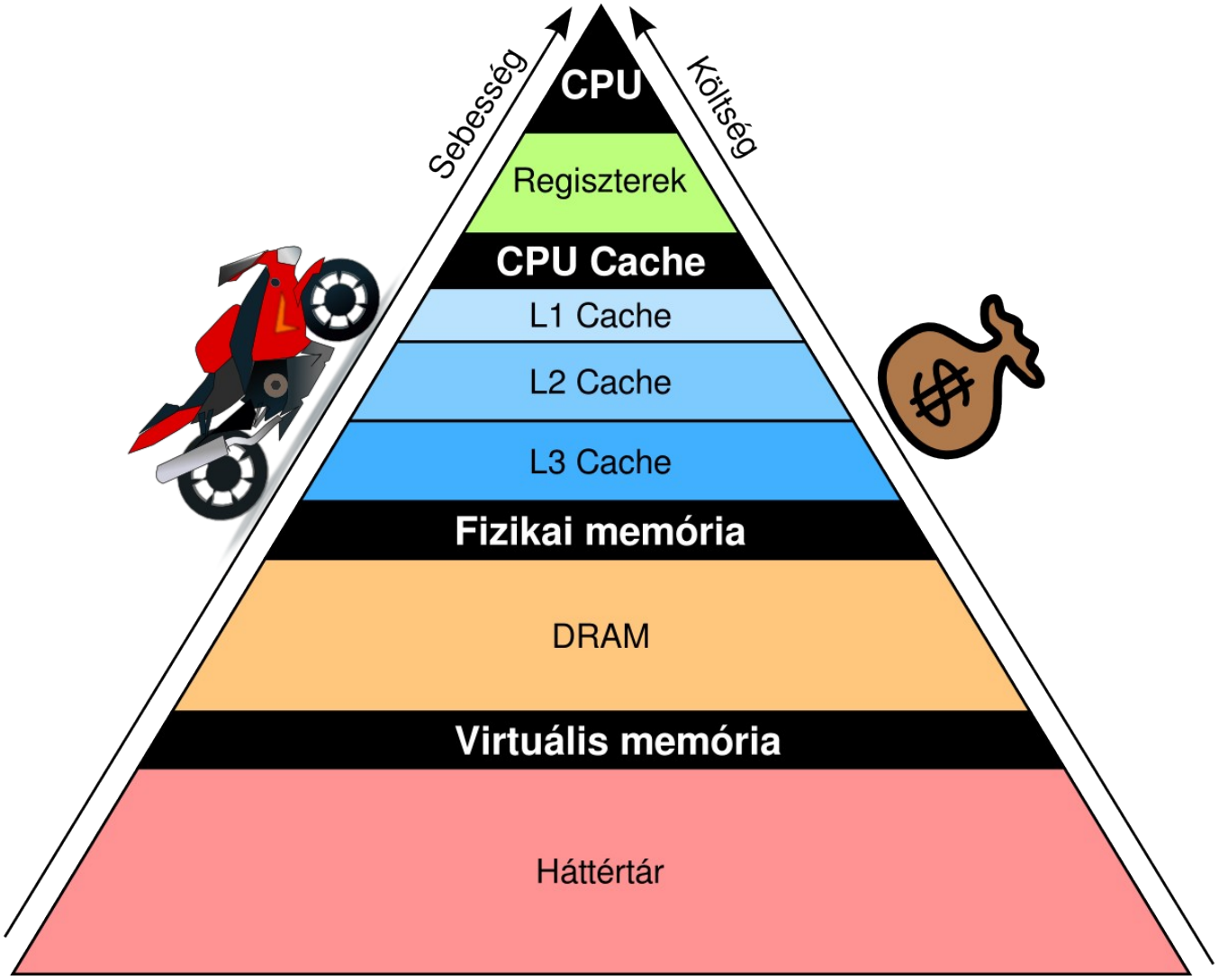
Tárhierarchia

- Ha van lokalitás:
  - vigyük a gyakran használt adatokat gyorsabb memóriába, közel a CPU-hoz
- Miért, nem minden memória egyformán lassú?
  - Van lassabb is: HDD
  - Az SRAM gyorsabb, mint a DRAM
  - Kisebb memória → nagyobb sebesség (lásd: jelterjedési idő)

Tár típusa	Elérési idő	Ár/GB
SRAM	0.5 – 2.5 ns	2000 – 5000 $
DRAM	50 – 70 ns	20 – 50 $
HDD	5 – 20 * 10^6 ns	0.2 $

(2008-as adatok)

Tárhierarchia



Cache megvalósítások

- Címzési mód szerint:
  - Transzparens:
    - A cím tartomány egy részének másolata a gyors memóriában van
  - Nem transzparens:
    - A címtartomány egy része alatt a gyors memória van
- Menedzsment szerint:
  - Implicit menedzsment:
    - A gyors memória tartalmát a hardver kezeli
  - Explicit menedzsment:
    - A gyors memória tartalmát a futó alkalmazás kezeli

Cache megvalósítások

	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: klasszikus CPU cache
- Scratch-pad memória: DSP-k, mikrokontrollerek, PlayStation 3
- Szoftver-menedzselt cache:
  - nincs cache találat → szoftver feladata a cache update
- Önszervező scratch-pad
  - Ritka megoldás

Cache megvalósítások

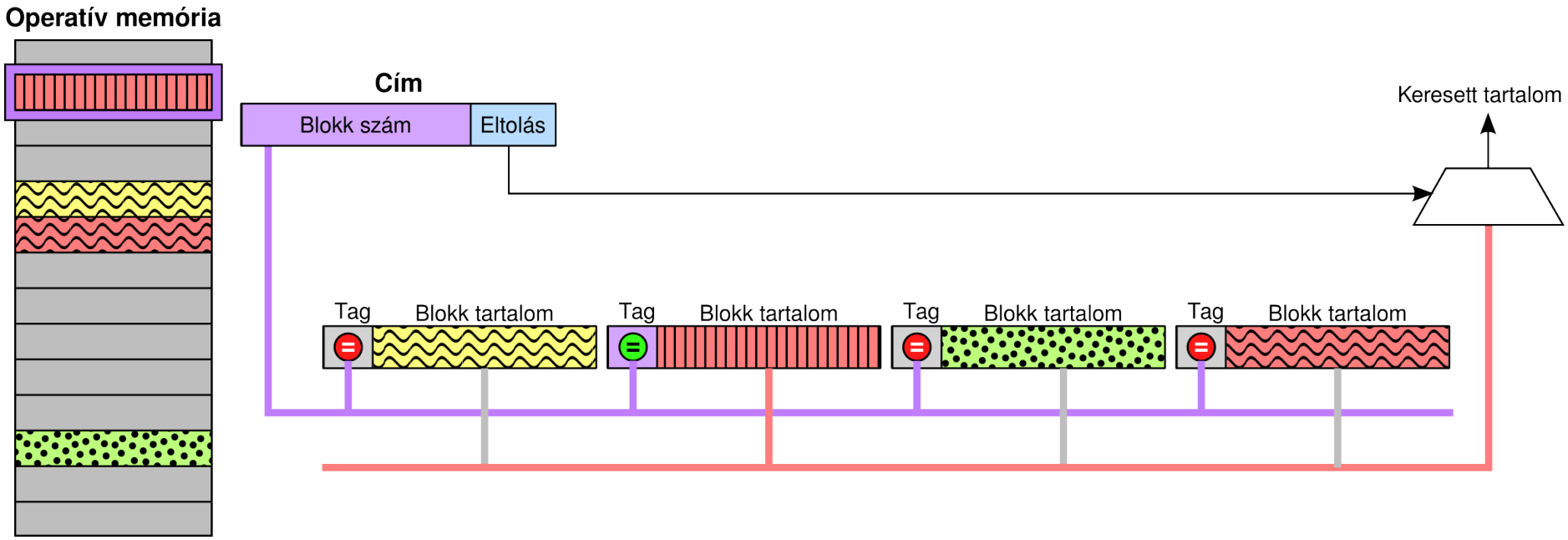
- Amiről ma tanulunk:
  - Cache szervezés:
    - Adatok hatékony tárolása a cache-ben
  - Cache tartalom menedzsment:
    - Mikor tegyünk a cache-be egy adatot
    - Ki tegyünk ki onnan, ha tele van

Transzparens cache szervezés

- Tárolási egység: ***blokk***
  - Blokkok mérete = 2^L
  - Címek alsó L bitje: blokkon belüli eltolás
  - Felső bitek: blokk sorszáma
- A cache minden blokkja mellé tárolt járulékos információ:
  - Cache ***tag*** (az op. memória hányas blokkja ez?)
  - ***Valid*** bit: ha =1, ez a cache blokk érvényes
  - ***Dirty*** bit: ha =1, erre a cache blokkra történt írás, mióta itt van
- A cache szervezés alap kérdése:
  - Hogy tároljuk a blokkokat a cache-ben
    - Hogy gyorsan kereshető legyen
    - Hogy egyszerű (gyors és olcsó) legyen

Teljesen asszociatív szervezés

- A blokkok a cache-ben bárhová elhelyezhetők
- Cache tag: ez a blokk az operatív memória hányas blokkja
- Sokat fogyaszt:
  - Keresés: cím blokk száma és az ***összes*** cache tag komparálása
  - Komparátorok szélessége: blokkszám bitszélessége



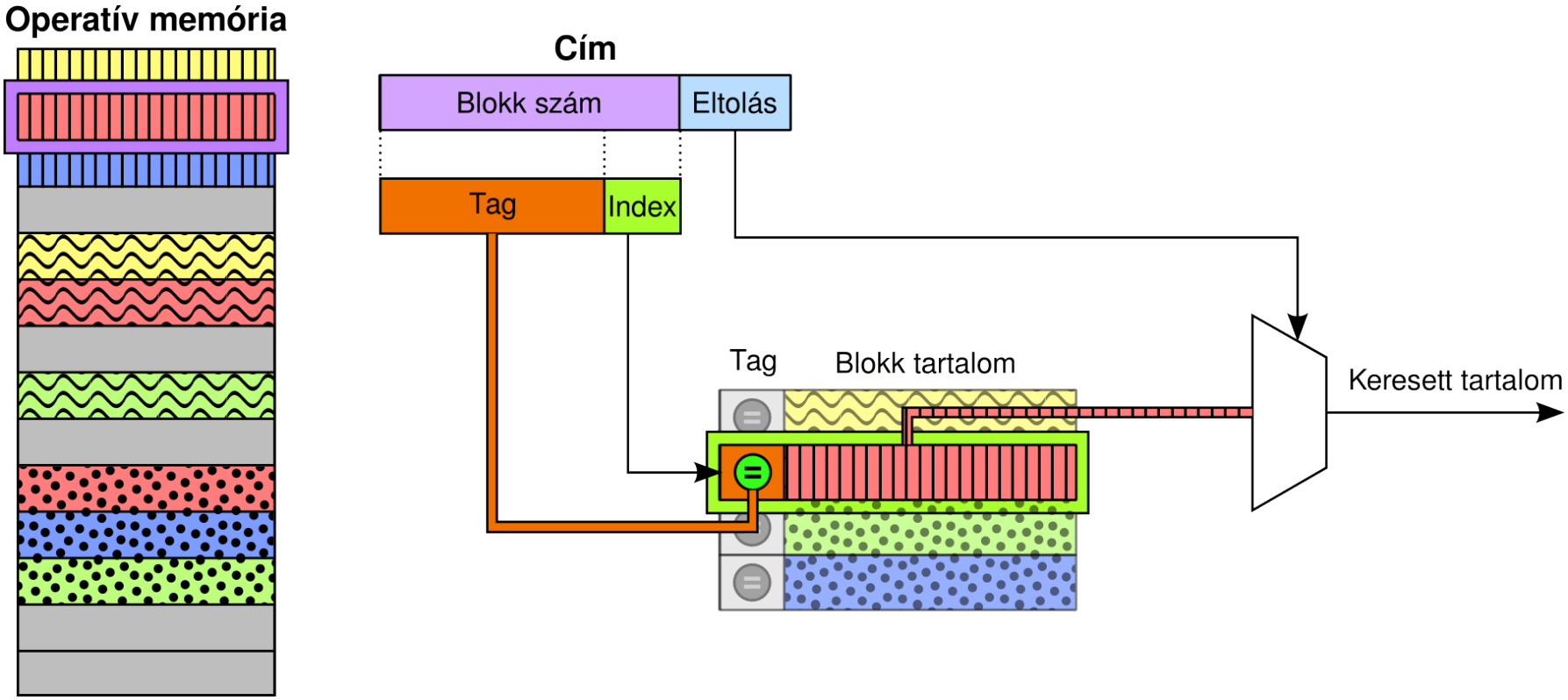
Direkt leképzés

- Minden memóriabeli blokk csak egyetlen helyre kerülhet a cache-ben
- Hogy hova, azt a memóriabeli blokkszám egyértelműen eldönti
  - Pl. a blokkszám alsó bitjei alapján

Direkt leképzés

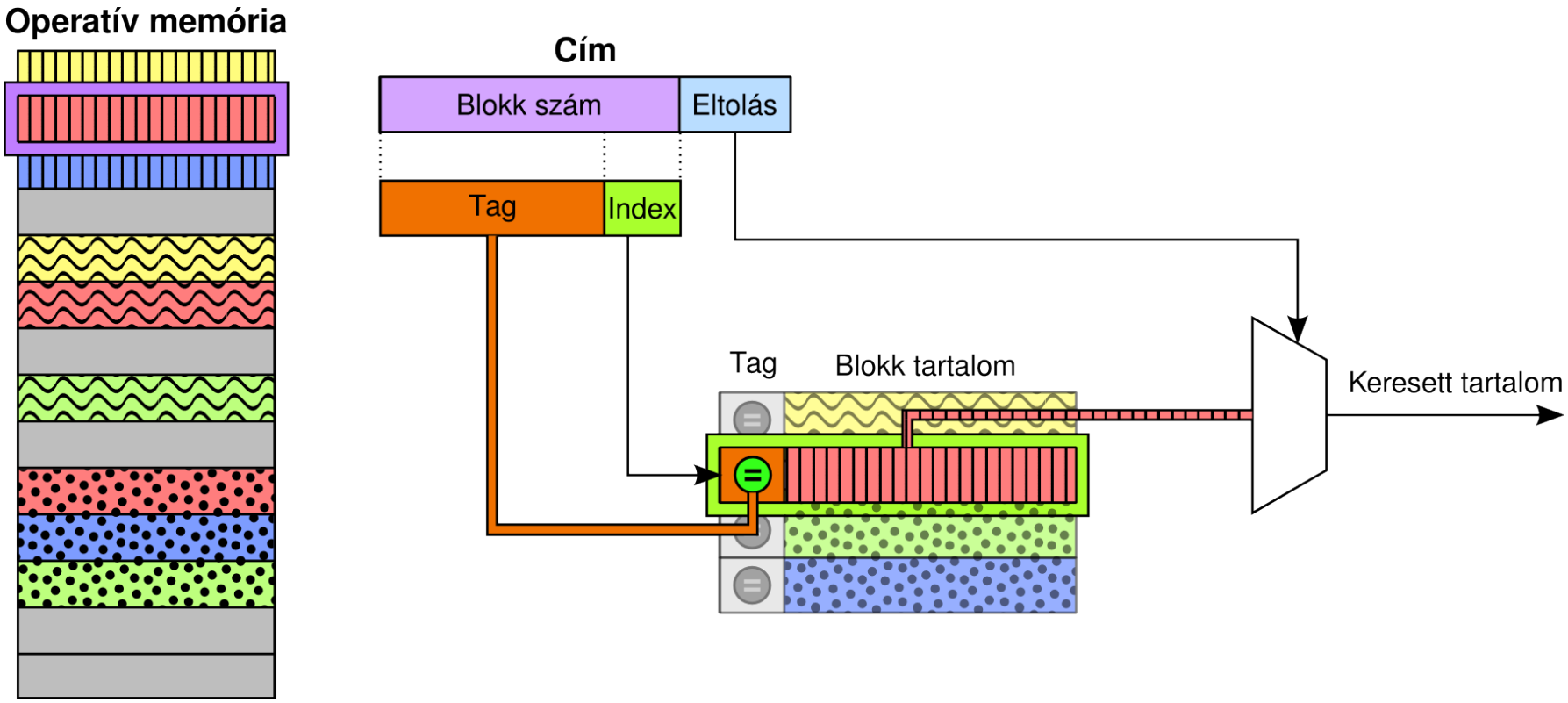
- Pl.: cache 4 blokkot tud tárolni: 1 sárgát, 1 pirosat, 1 kéket, 1 zöldet
- A memóriában a blokkok ebben a sorrendben követik egymást

  → szín = blokkszám alsó 2 bitje



Direkt leképzés

- Keresés:
  1) **Indexelés**: A cím blokkszámából a szín tudható: piros (ez az ***index***)
  2) **Komparálás**: A cache piros rekeszében ez a blokk van?
     - Egyetlen komparátor megy, és az is keskenyebb! (***Tag*** hosszú)



Értékelés

- Teljesen asszociatív:
  - Szabad blokk elhelyezés
  - Sok, széles komparátor működik benne

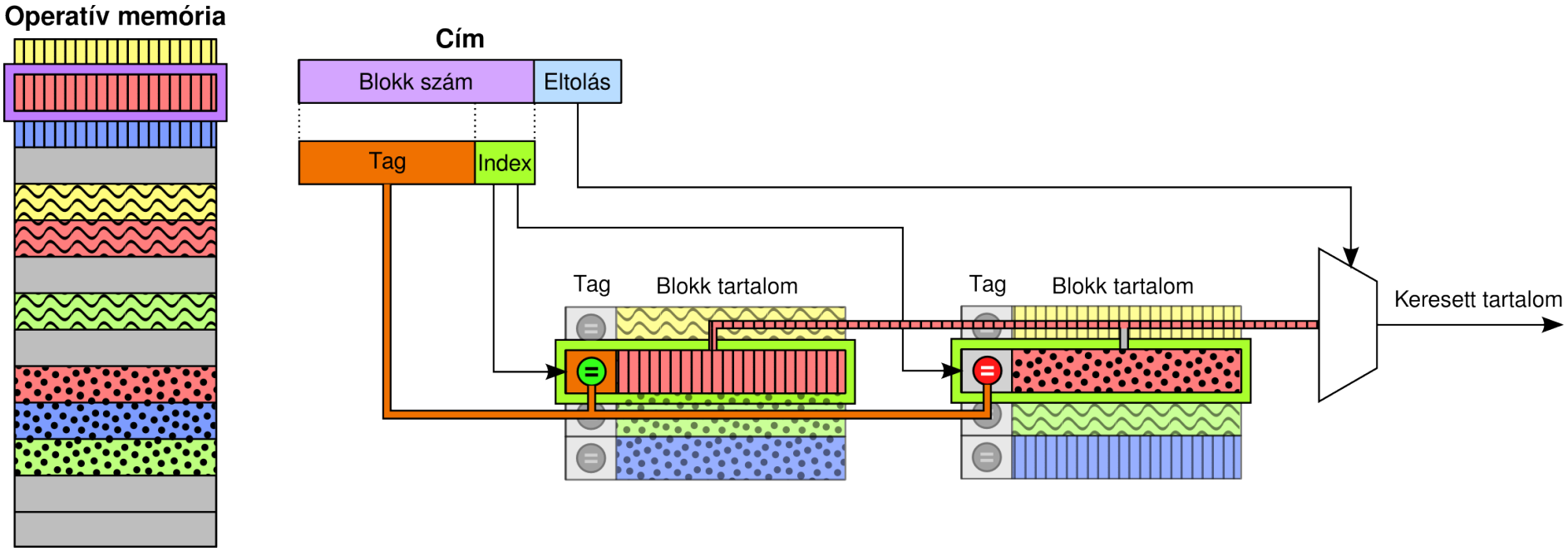
    → komplexitás, fogyasztás
- Direkt leképzés:
  - Korlátozott blokk elhelyezés:

    → Versenyhelyzet: szuboptimális működés

    Pl. a program csak piros blokkokkal dolgozik
  - Csak 1, keskenyebb komparátor dolgozik

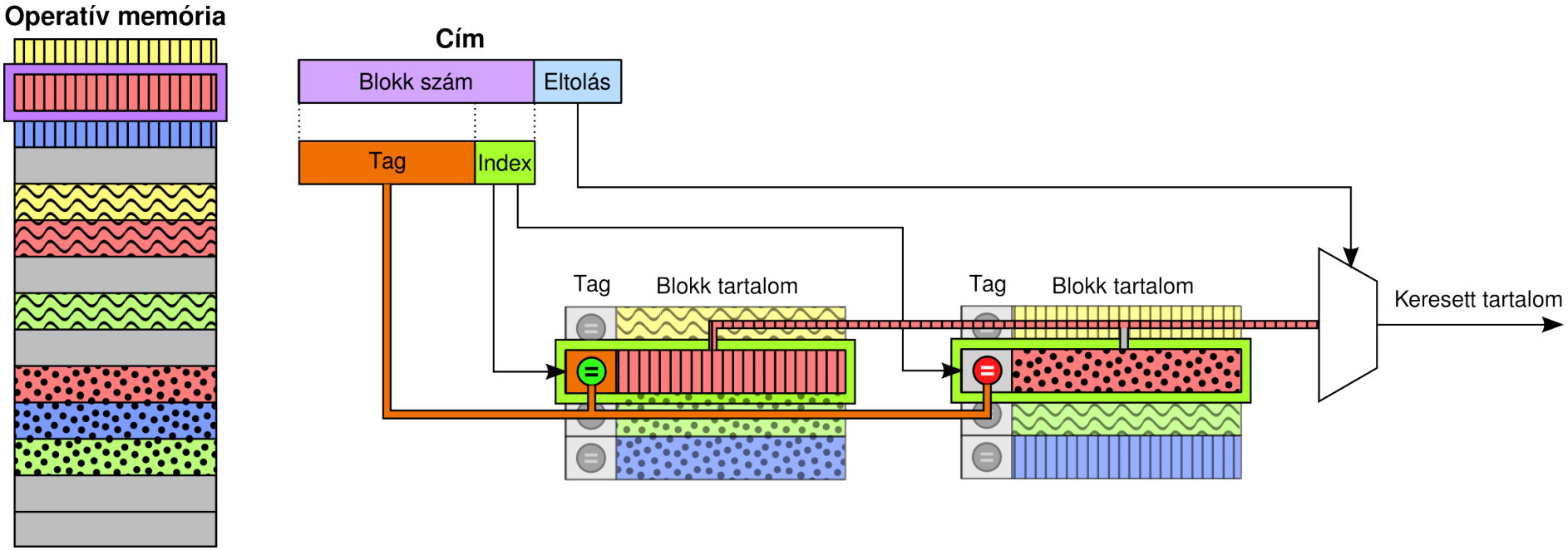
N-utas asszociatív szervezés

- Kombinálja a korábbi két megoldást
- A blokkszám alsó bitjei meghatározzák a blokk helyét
  - De nem egyértelműen!
  - Az alsó bitek kijelölnek egy ***halmazt***, ahol a blokk lehet
  - A halmaz n cache blokkból áll, ezek bármelyikében lehet a keresett blokk



N-utas asszociatív szervezés

- Keresés:
  1) **Indexelés**: A cím blokkszámából a szín tudható: piros (ez az ***index***)
  2) **Komparálás**: A cache n piros rekesze közül melyikben van ez a blokk?
     - n komparátor megy, keskenyebb szélességgel (***Tag*** hosszú)



Értékelés

- n-utas asszociatív szervezés:
  - Korlátozott blokk elhelyezés, n lehetőséggel
    → ritkábban van versenyhelyzet
  - Csak n komparátor dolgozik
    → moderált komplexitás és fogyasztás
- Tipikus n értékek:
  n=2...16

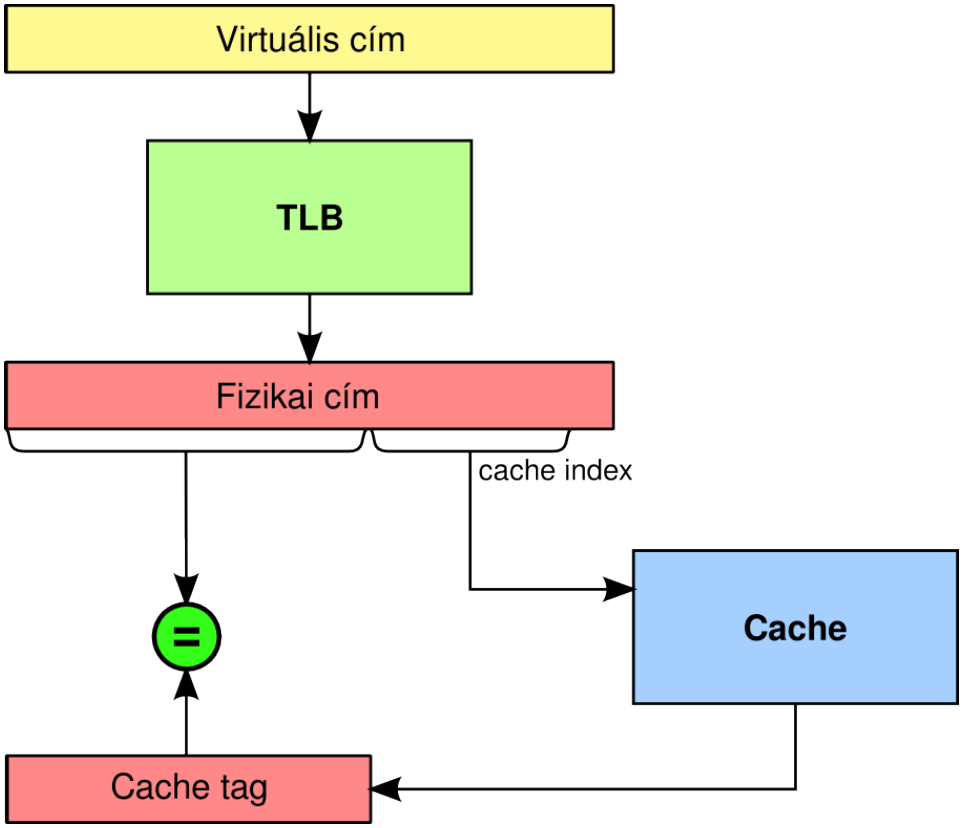
	Core i7	AMD Bulldozer	ARM Cortex A9
L1	8	2 (ut.) / 4 (adat)	4
L2	8	16	16
L3	16	64	

Cache + virtuális memória

- Mit tároljon a cache?
  - Fizikai memória blokkjait?
  - Virtuális memória blokkjait?
- Természetes választás:
  - Fizikai memória blokkjait!
    - Mert kisebb → kevesebb blokk van benne → a cache tag és a komparátorok keskenyebbek lehetnek
    - Mert a tárhierarchiában a fizikai memória van alatta

Fizikailag indexelt cache fizikai tag-el

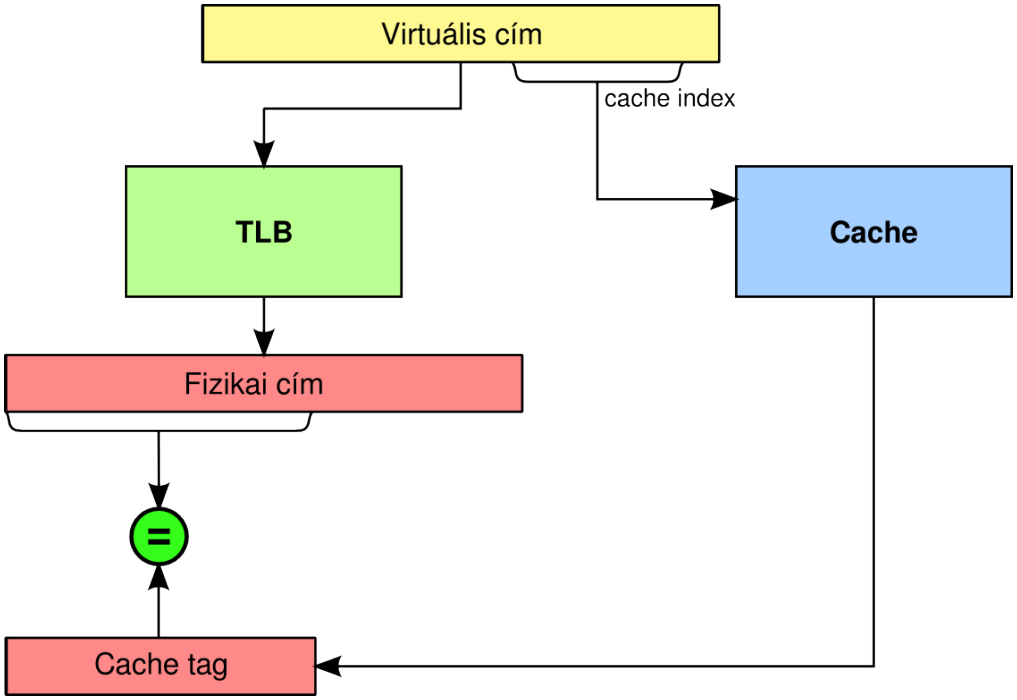
- Minden memóriaművelet:
  1) Címfordítás
  2) Cache indexelés
     (halmaz kiválasztása)
  3) Komparálás
     (halmazon belül hol lehet)



- Ez 3 lépés.
- Nem lehetne gyorsabban?

Virtuálisan indexelt cache fizikai tag-el

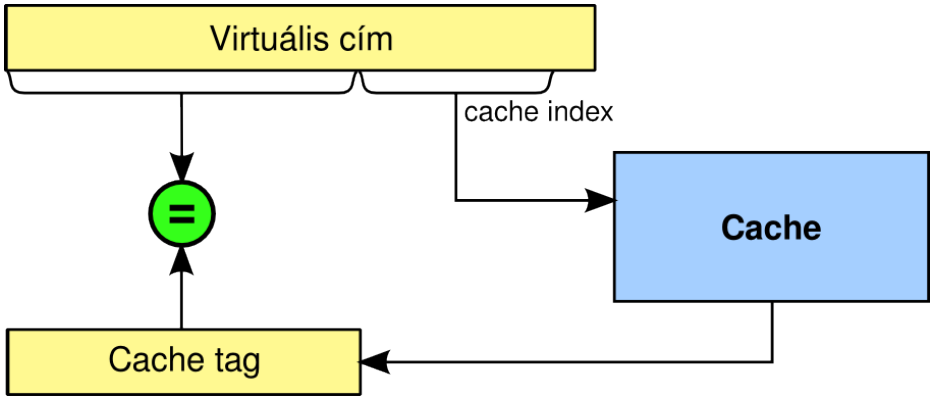
- Cache indexelés (halmaz kiválasztás) a virtuális címmel!
- Minden memóriaművelet:
  1) Egyidejűleg:
     - Címfordítás
     - Cache indexelés
       (halmaz kiválasztása)
  2) Komparálás
     (halmazon belül hol lehet)



- Már csak 2 lépés.
- Nem lehetne még egyszerűbben?

Virtuálisan indexelt cache virtuális tag-el

- Az egész cache virtuális címekre épül
- Minden memóriaművelet:
  1) Cache indexelés
     (halmaz kiválasztása)
  2) Komparálás
     (halmazon belül hol lehet)



- Címfordítás csak cache hiba esetén kell!
- Hátrány:
  - Szélesebb a tag. (Mert a virtuális cím szélesebb)
  - Ha egy keretet több lappal lehet elérni, többször lesz bent a cache-ben

Tartalom menedzsment

- Már tudjuk, milyen megoldások vannak
  - Az adatok tárolására:
    - Teljesen asszociatív: hatékony, de drága
    - Direkt leképzés: egyszerű, de nem olyan hatékony
    - N-utas asszociatív: arany középút
  - A virtuális és fizikai címek használatára:
    - Érdemes a virtuális címmel indexelni
    - Jó ötlet lehet a virtuális címből képezni a tag-et
- Jön: hogy menedzseljük a cache tartalmát?
  - Kit tegyünk bele?
  - Mikor tegyük bele?
  - Kit rakjunk ki?
- Implicit / explicit

Adatok betöltése a cache-be

- Mikor tegyünk be egy memória blokkot a cache-be?
  - Soha
  - Amikor a futó program először hivatkozik rá
  - Jó előre, mielőtt még használnánk, jól jön az még
- Soha: cache szemetelés ellen véd
  - pl. médialejátszás: kijátszik egy képet, soha többet nem kell.

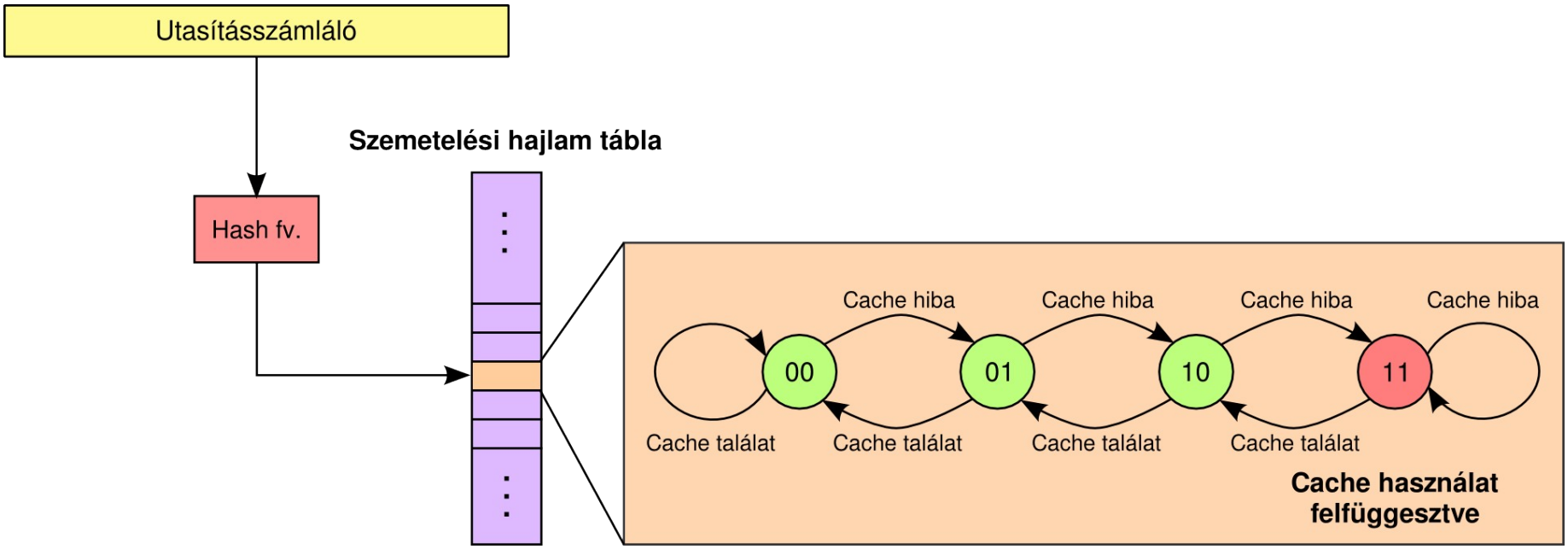
    → nem érdemes cache-be betölteni a képet
- Első hivatkozáskor: elsőre sokáig tart (memória → cache átvitel), de később gyors lesz
- Idő előtti betöltés (prefetch): spekulációt igényel
- Írás/olvasás megkülönböztetés:
  - **Write allocate**: íráskor is cache-be viszi az adatot
  - **Write no-allocate**: íráskor, ha nincs bent, nem hozza be

Cache szemetelés megelőzése

- Cache szemét: az a blokk, amire a bekerülés és a kirakás között nem volt hivatkozás
  - Kár volt behozni, kiszoríthatott hasznos blokkokat
- Explicit megoldás: spéci hardver utasítások:
  - **x86**: MOVNTI, MOVNTQ
    - SSE regiszterből memóriába, a cache kikerülésével
  - **PowerPC**: LVXL memóriából vektor regiszterbe olvas.
    - A blokk cache-be kerül, de megjelölik, ezt dobják ki elsőnek, ha kell a hely. (LVX: ugyanez, jelölés nélkül)
  - **PA-RISC**: Sok utasítás kódjában van egy bit:
    - Spatial Locality Cache Control Hint
  - **Itanium**: Adatmozgatásnál „.nt” opció
    - jelezhető, hogy az adat nem felel meg az időbeni lokalitásnak
  - **Alpha**: ECB utasítás (Evict Data Cache Block)
    - Jelezhető, hogy a blokk nem lesz mostanában hivatkozva

Cache szemetelés megelőzése

- Implicit megoldások: a CPU próbálja detektálni a szemetelő utasításokat
- Ezernyi megoldás. Példa: Rivers' algoritmus
- Ha egy utasítás sok cache hibát okoz → CPU eltiltja a cache-től



Idő előtti betöltés (prefetch)

- Kulcsfontosságú funkció
- Cél: a CPU működése ne akadjon meg cache hiba miatt
- Ezért már be kell hozni minden adatot, mielőtt még először meghivatkoznák
- ***Utasítás cache*** esetén egyszerű:
  - Az nem ugró utasítások sorban követik egymást
    → kiszámítható
  - A feltétel nélküli ugrások ugrási címe ismert
    → kiszámítható
  - A feltételes ugrások nagyon jól becsülhetők
    → egész jól kiszámítható

Idő előtti betöltés (prefetch)

- Az adatok hivatkozásának mintázata nehezen kiszámítható
- Ha a becslés:
  - Túl óvatos: nem lesz bent az adat, mikorra kell
    → sok cache hiba
  - Túl agresszív: feleslegesen hoz be szükségtelen blokkokat
    → kiszorít gyakran használtakat → sok cache hiba
- Explicit támogatás:
  - Speciális utasításokkal
- Implicit támogatás:
  - CPU spekulál

Idő előtti betöltés (prefetch)

- Explicit prefetch utasítások:
  - **x86**: PREFETCHT0/1/2, PREFETCHNTA
    - behoznak egy blokkot a cache-be
    - PREFETCHNTA: a több utas cache első blokkjába teszi a behozott adatot → elkerülhető a szemetelés is!
  - **PowerPC**: DCBT, DCBTST
    - egy blokk cache-be töltése olvasásra/írásra
  - **PA-RISC**: LDD, LDW
    - egy blokk cache-be töltése írásra/olvasásra
  - **Itanium**: lfetch
    - utasítás egy blokk cache-be töltésére
  - **Alpha**: Ha egy normál adatmozgató utasításnál célként az R31 regiszter van megjelölve, akkor azt a processzor egy idő előtti betöltés kérésnek értelmezi

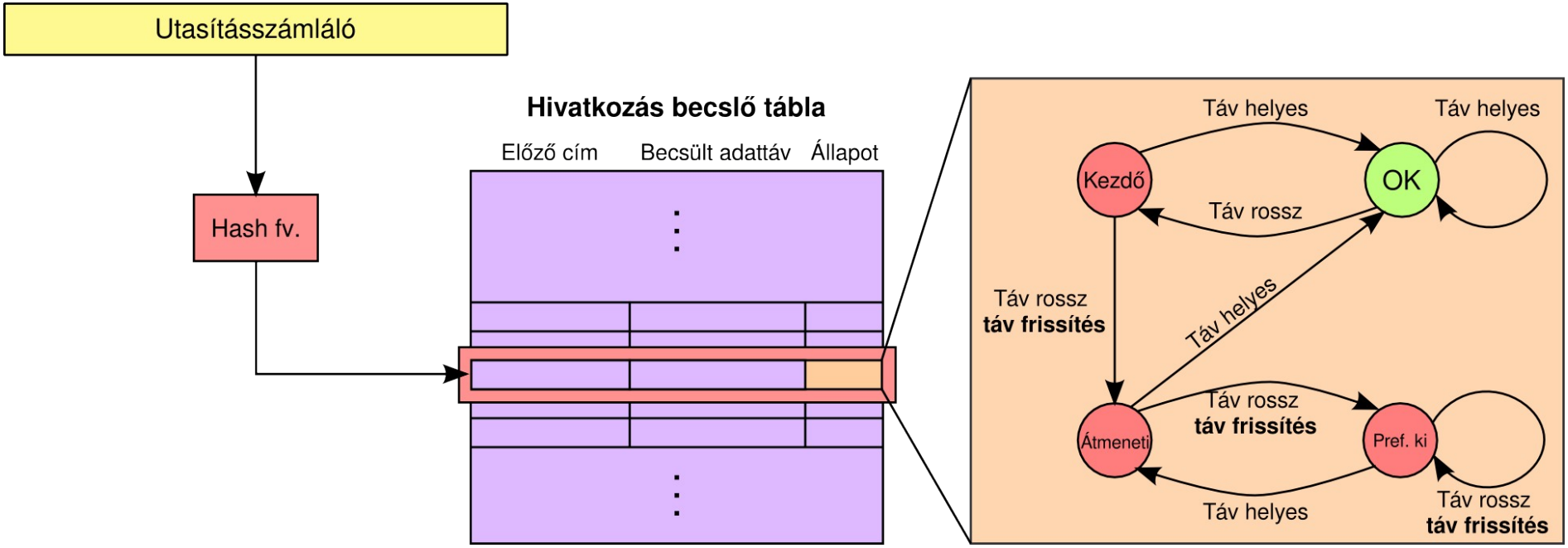
Idő előtti betöltés (prefetch)

- GCC fordító platformfüggetlen megoldása:

```
__builtin_prefetch (mutató);
```

Idő előtti betöltés (prefetch)

- Implicit algoritmus:
  - Fix távolságra lévő adatok lineáris bejárására:
    - X, X + táv, X + 2*táv, X + 3*táv, ...
  - Automatikusan detektálja az egymást követő címek távolságát
- = "Intel Smart Memory Access IP Prefetcher" a Core i7-ben



Cserestratégia

- Eldöntöttük, kit akarunk behozni, és mikor
- De hova tegyük?

  → a cache üzemi állapota az, hogy tele van
- Valakit ki kell dobni.
- Lehetséges jelöltek száma = a cache asszociativitása
  - Direkt leképzés esetén nincs választási lehetőség:

    Az új blokk csak egy helyre kerülhet. Aki ott volt, repül.
- Lehetséges algoritmusok:
  - Véletlen választás
  - Körbenforgó
  - **Legrégebben használt (LRU)**
  - Nem a legutóbb használt
  - Legritkábban hivatkozott

Cache koherencia

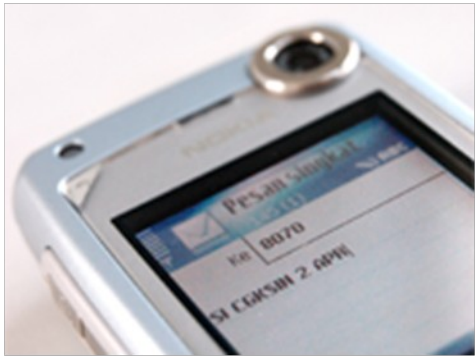
- Operatív memóriát több szereplő is eléri
- A cache tartalma összhangban kell legyen vele

A cache koherens, ha a memóriaműveletek a cache jelenlétében ugyanazt az eredményt adják, mint cache nélkül.

- Bűnös: az **írás** művelet
- Írási stratégiák:
  - Write-through

    → cache-beli írást rögtön átvezeti az operatív memóriába
  - Write-back

    → csak akkor vezeti át, amikor kikerül a cache-ből



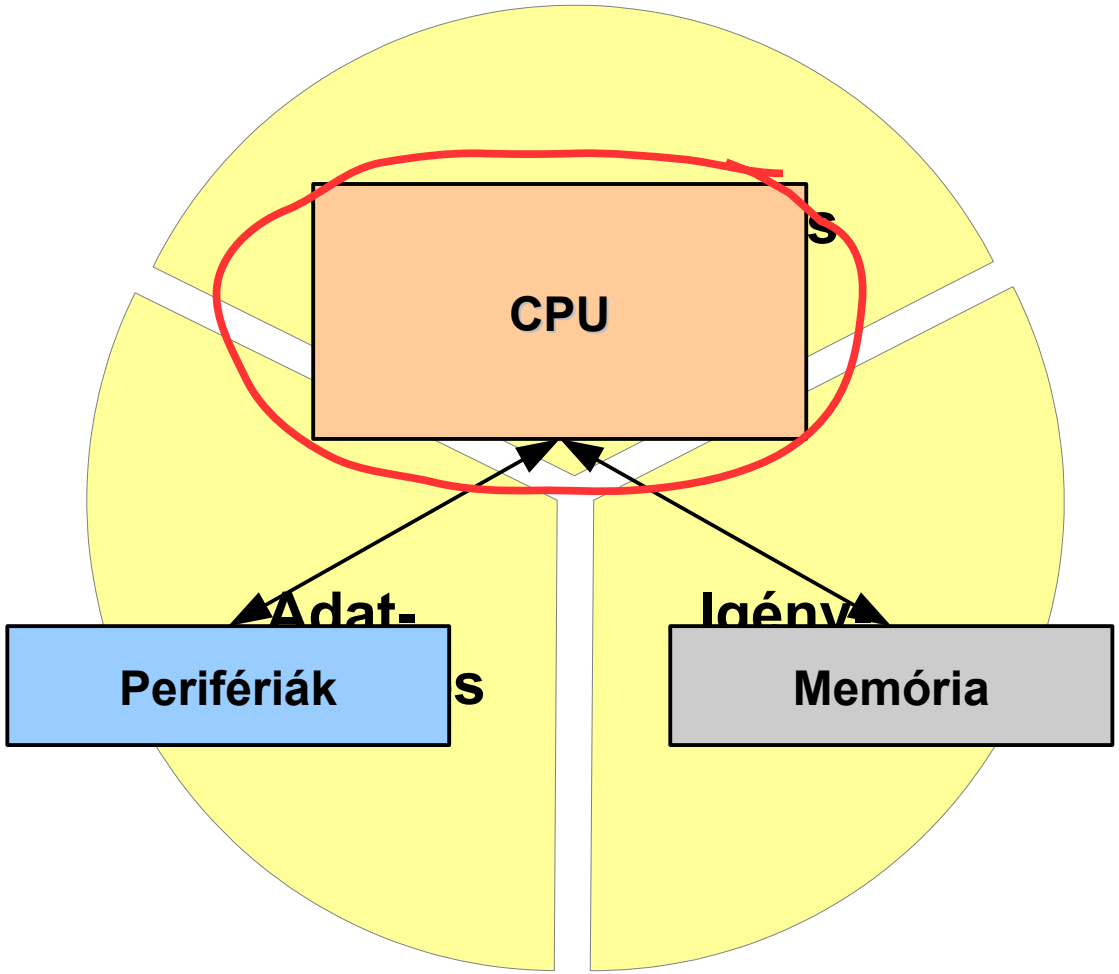
Számítógép Architektúrák

Pipeline utasításfeldolgozás

Horváth Gábor
docens
BME Híradástechnikai Tanszék
ghorvath@hit.bme.hu

2012. november 5.
Budapest

Hol tartunk?



Utasítások feldolgozása

- Most megtanuljuk, hogy lehet hatékonyan.
- Példa processzor: RISC.
- Utasításai:
  - Load/Store
    `R1 ← MEM[R0+42]` vagy `MEM[R0+42] ← R1`
  - Aritmetikai logikai
    `R1 ← R2 * R3` vagy `R1 ← 42 * R3`.
  - Vezérlésátadás:
    - Feltétel nélküli
    - Feltételes
      `JUMP -24 IF R2 == 0`
  - Load/Store kivételével más nem nyúlhat memóriához

Utasítások feldolgozása

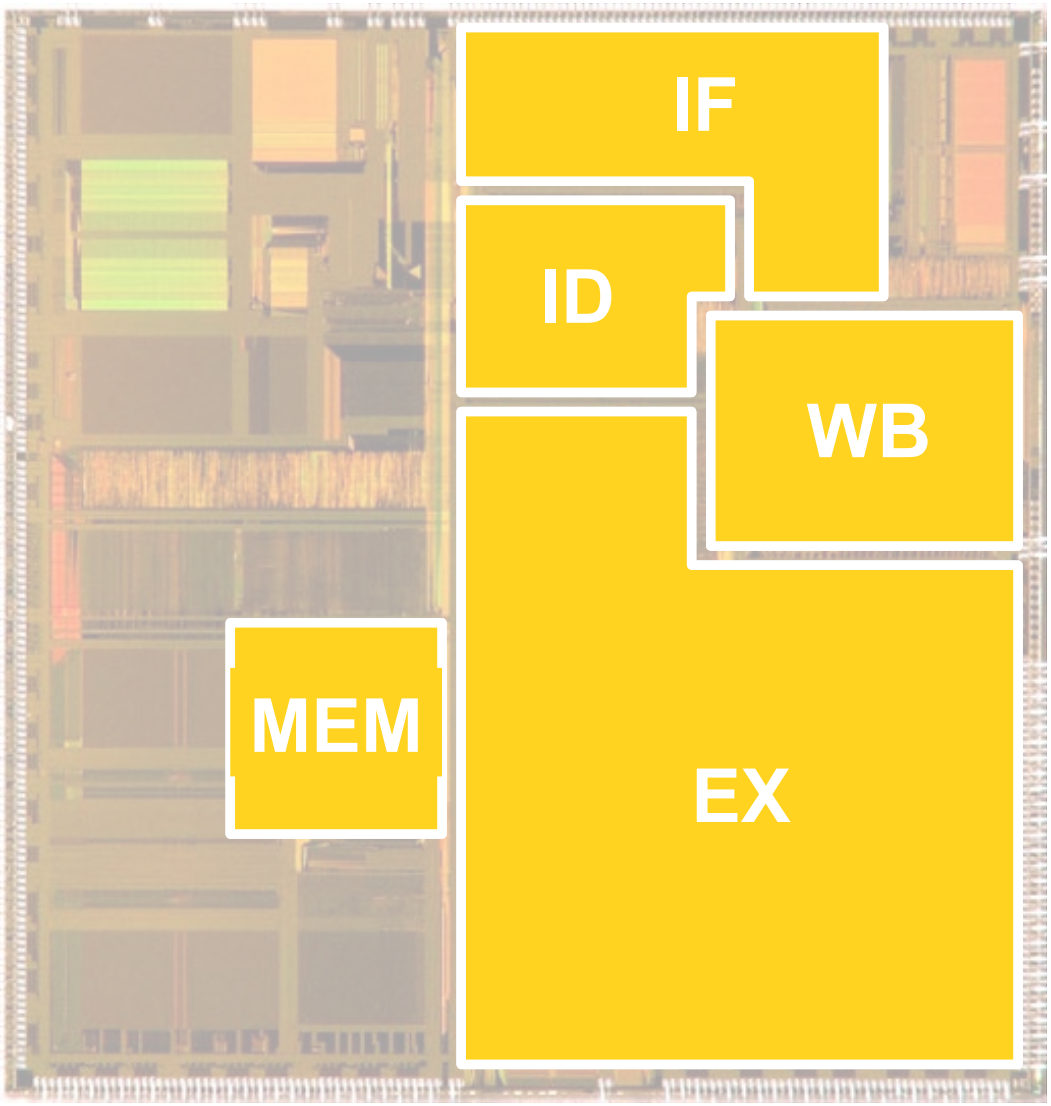
- Utasítás végrehajtásának lépései:
  1) Lehívás (Instruction Fetch, **IF**)
     + utasításszámláló növelése (ha nem ugrás)
  2) Dekódolás, regiszterek kiolvasása (Instruction decode/register fetch, **ID**)
     - Binárisan kódolt utasítás felkoncolása:
       – Típus leválasztása
         → ALU vezérlő jelek
       – Regiszter operandusok kiolvasása a regiszter tárolóból
  3) Végrehajtás (Execution, **EX**) – ALU dolgozik
     - Load/Store: címet számol (`R1 ← MEM[R0+42]`)
     - Aritmetikai/logikai: eredményt számol (`R1 ← R2*R3`)
     - Feltételes ugrás: feltételt és ugrási címet számol (`JUMP [PC]-24 IF R2==0`)

Utasítások feldolgozása

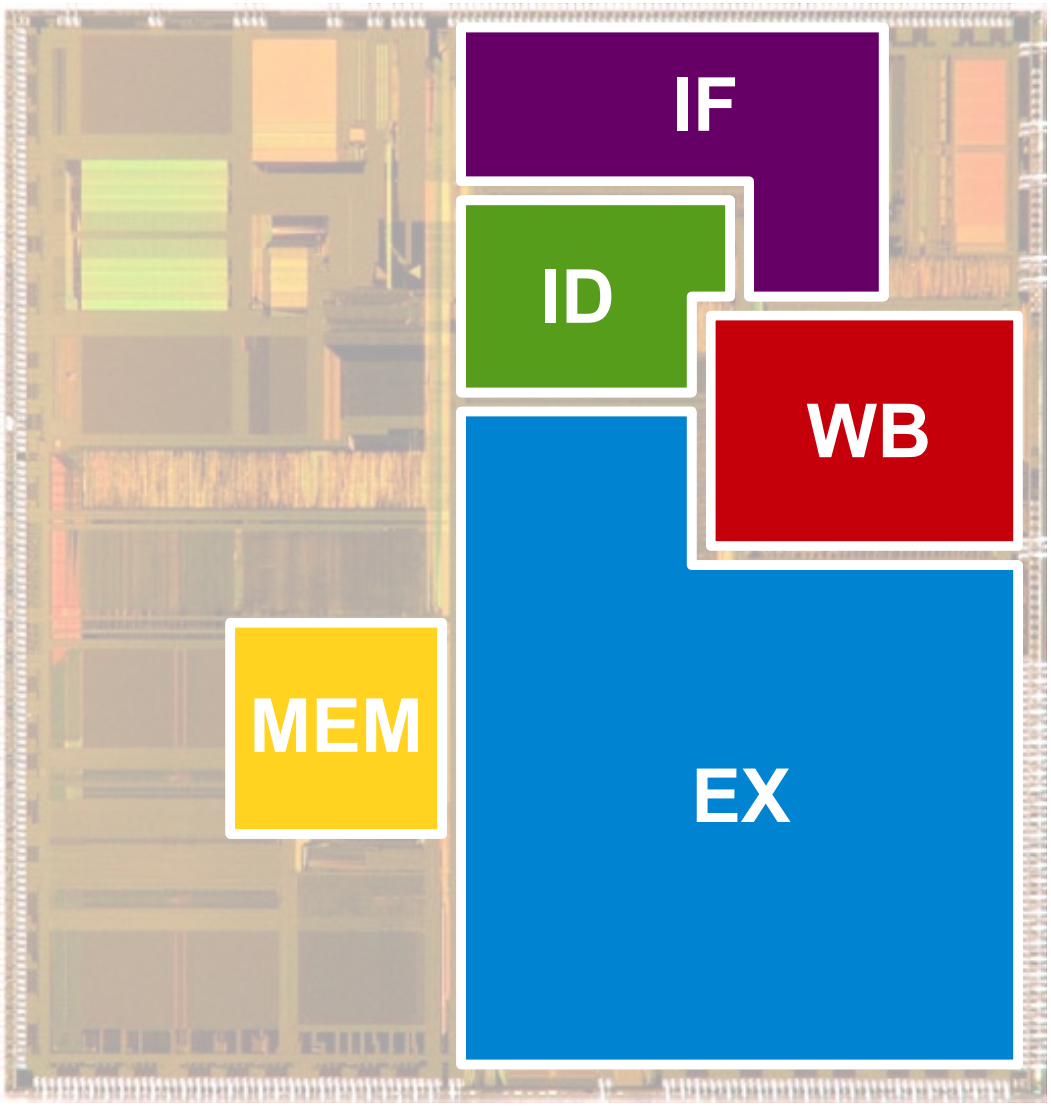
- … folytatás:
  4) Memóriaműveletek (Memory access, **MEM**)
     - Load/Store: végrehajtódik
     - Aritmetikai/logikai, ugrások: kihagyható
  5) Regiszterek frissítése (Write-back, **WB**)
     - Aritmetikai/logikai: eredményt tárol (`R1 ← R2*R3`)
     - Load: eredményt tárol (`R1 ← MEM[R0+42]`)
     - Store, ugrások: kihagyható

- Aritmetikai: IF, ID, EX, WB
- Store: IF, ID, EX, MEM
- Load: IF, ID, EX, MEM, WB
- Ugrások: IF, ID, EX

Utasítások feldolgozása

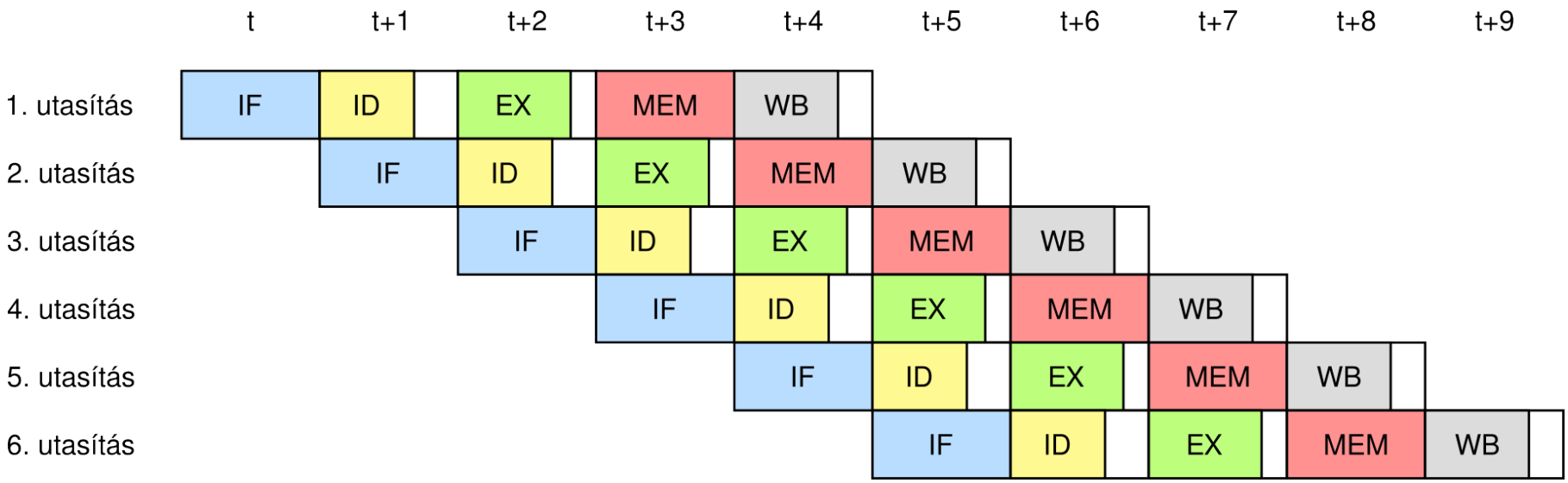


Utasítások feldolgozása



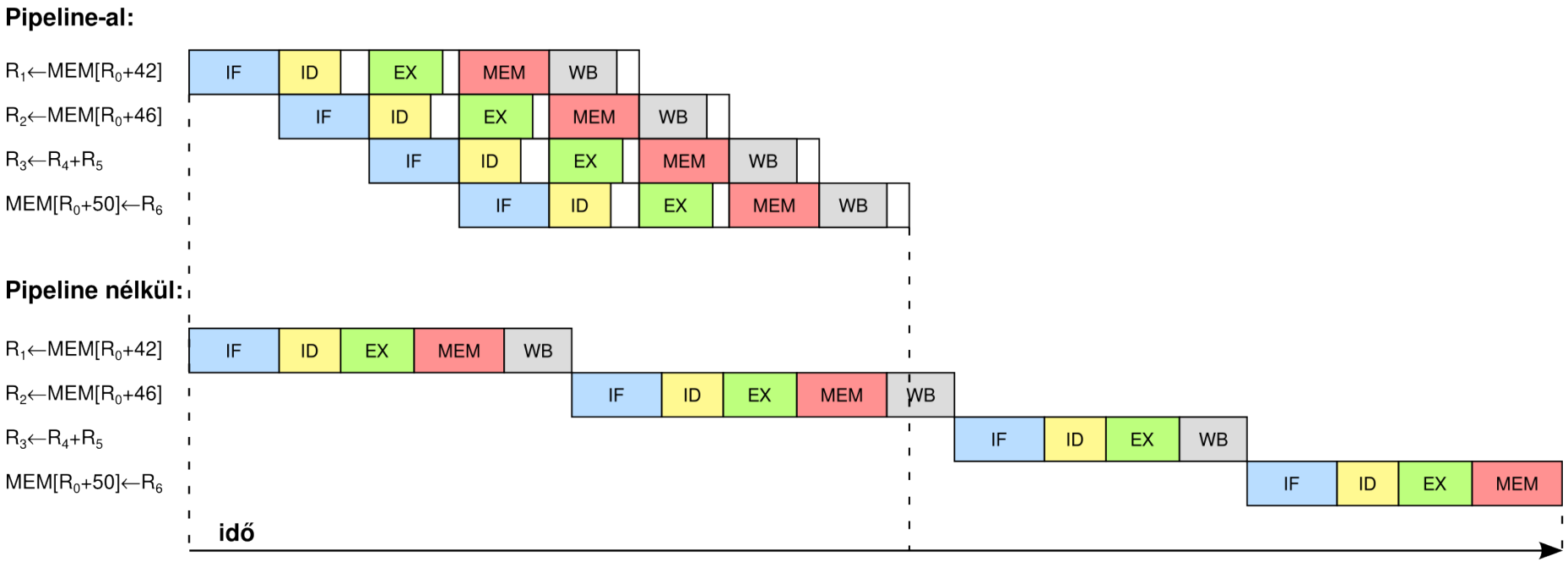
Pipeline elv

- Ez a pipeline elv
  - Lásd: chicagói vágóhidak, Ford T-model, stb.
- Átlapolt utasításfeldolgozás:
  - Minden fázisnak ugyanannyi időt kell adni: ***ciklusidő***



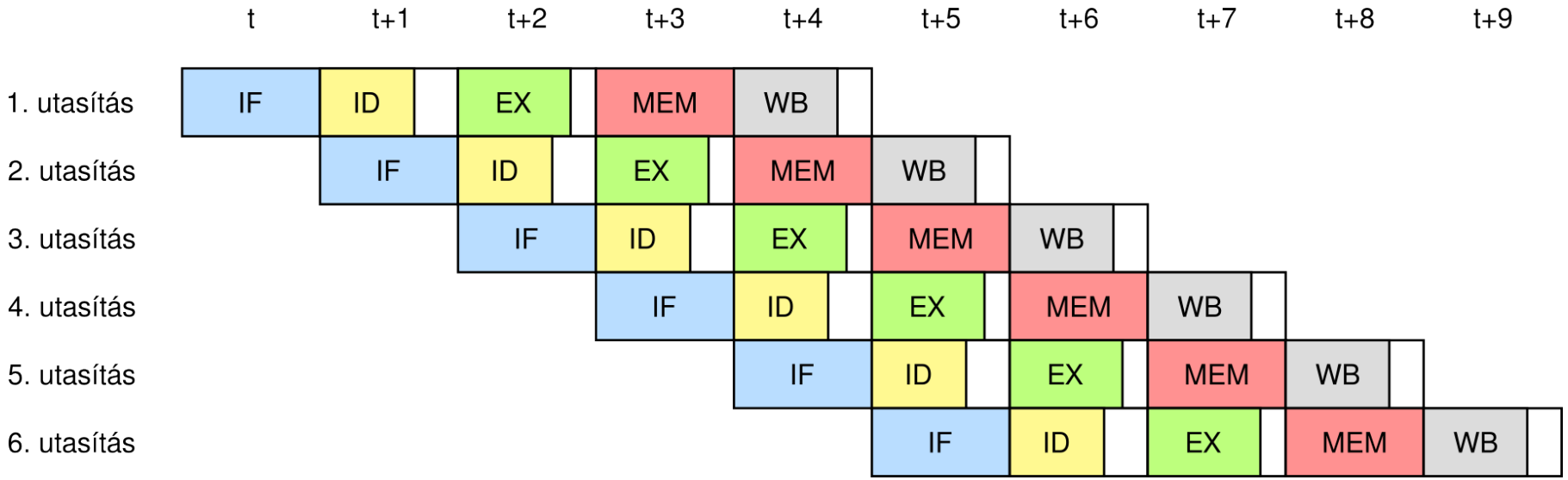
Pipeline elv

- Mennyit nyerünk?
  - Egyfelől: átlapoltunk, ez nyereség
  - Másfelől: üresjáratok! (szükségtelen fázisok, túl hosszú fázisok, ...)
- Ha függetlenek az utasítások, akkor sokat!



Pipeline elv

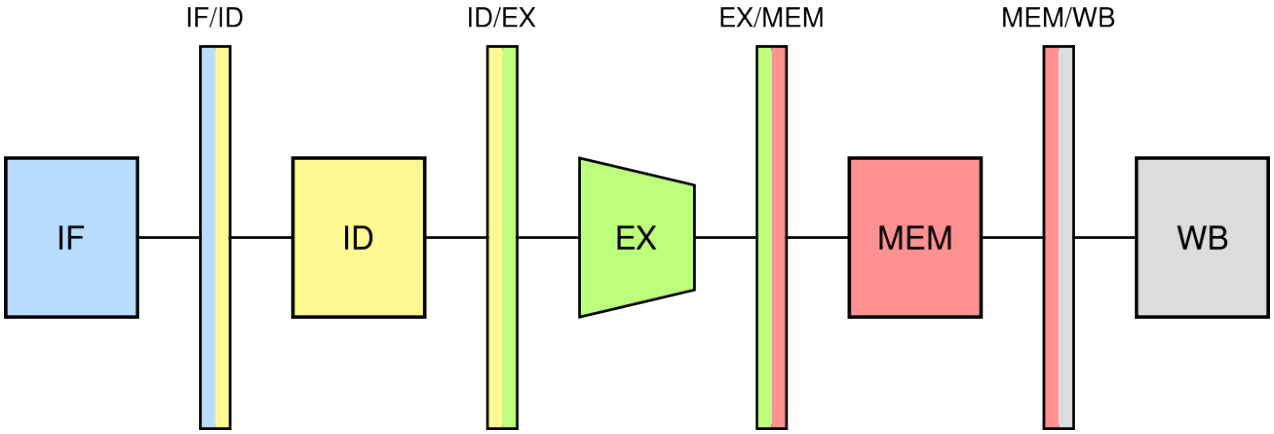
- Mennyiségek:
  - Mélység (deepness): fázisok száma
  - Késleltetés (latency): átfutási idő
  - Átviteli sebesség (throughput): kész utasítás / sec
  - Feltöltési / kiürülési idő



Pipeline regiszterek

- A fázisok egymásnak adogatják a „munkadarabbal” kapcsolatos dolgokat

  → pipeline regiszterekben tárolódnak



- IF az IF/ID-be teszi a lehívott utasítást,
- ID kiveszi, ALU vezérlőjeleket és operandusokat az ID/EX-be teszi,
- EX kiveszi, eredményt az EX/MEM-be teszi,
- Stb... (később)

Egymásrahatások

- Minden szép és jó, de:
- Az utasítások többnyire nem függetlenek!
  - Erőforrásokért versengenek

    → ***Feldolgozási egymásrahatás***
  - Egyik operandusa a másik eredménye

    → ***Adat egymásrahatás***
  - Ugrás: nem tudjuk a folytatást, amíg ki nem értékeljük a feltételt

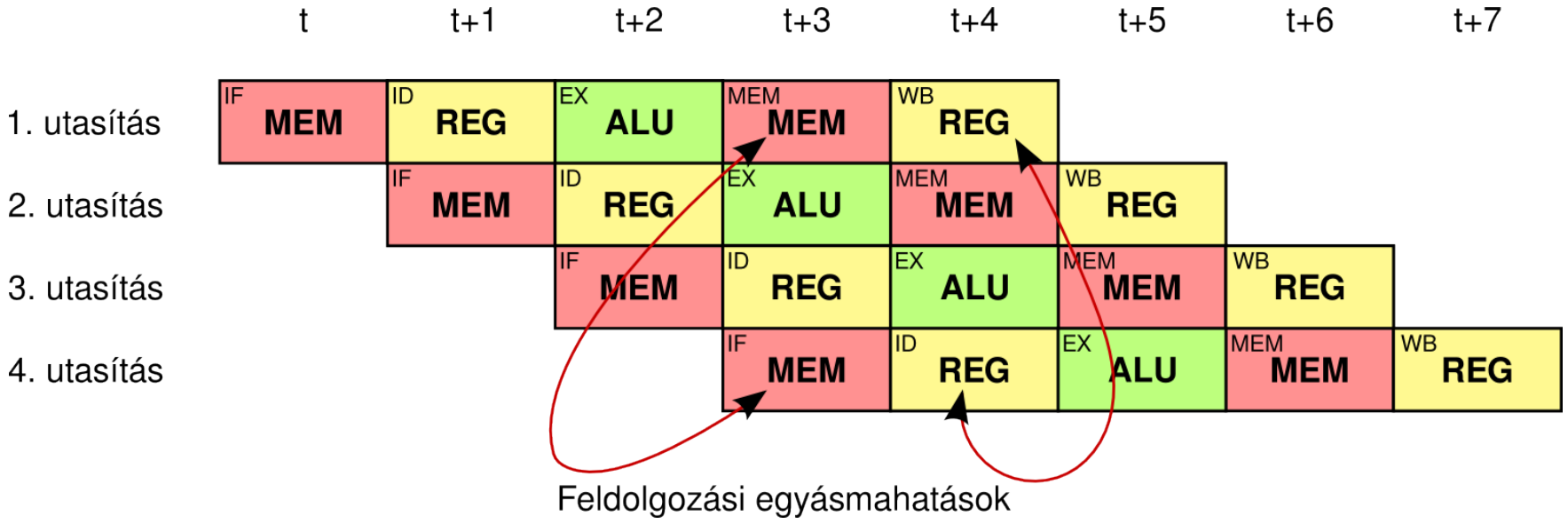
    → ***Procedurális egymásrahatás***

  Kezelés:
  - Trükkel feloldjuk
  - Ha nem megy, megállítjuk a pipeline-t, amíg meg nem oldódik

    → rombolja a hatékonyságot

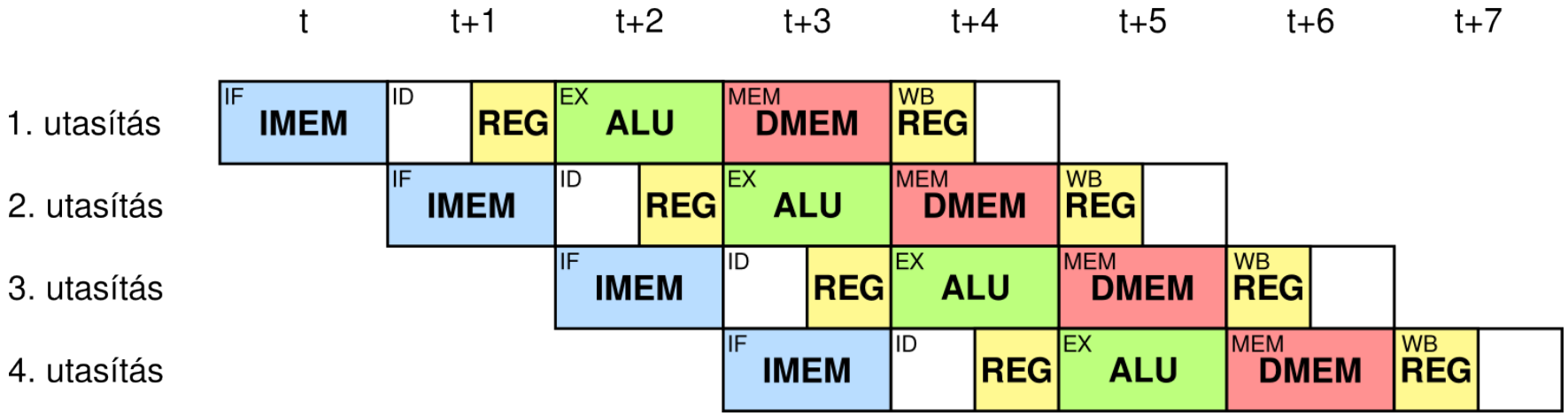
Feldolgozási egymásrahatás

- Egyes erőforrásokra több fázisban is szükség van
  - IF: Memória
  - ID: Regiszter tároló
  - EX: ALU
  - MEM: Memória
  - WB: Regiszter tároló



Feldolgozási egymásrahatás

- Feloldása:
  - Memória esetén:
    külön utasítás és adat cache
  - Regiszter tároló esetén:
    egyik az ciklus elején, másik a végén használhatja



Adat-egymásrahatás

- Forrása: adatfüggőségek
- Pl. az egyik utasítás operandusa az előző eredménye

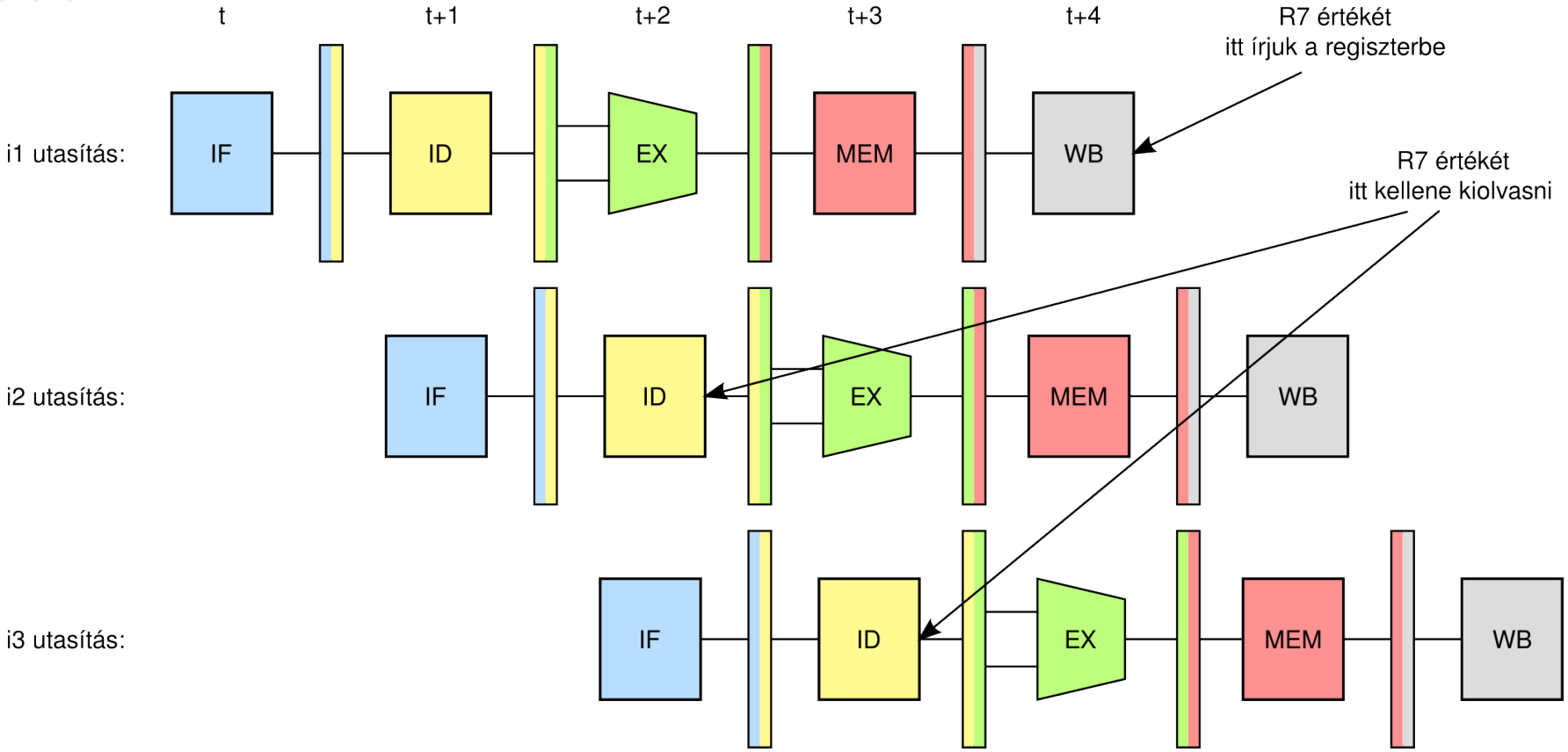
  → ***RAW függőség*** (Read After Write)

```
i1:   R3 ← MEM[R2]
i2:   R1 ← R2 * R3
i3:   R4 ← R1 + R5
i4:   R5 ← R6 + R7
i5:   R1 ← R8 + R9
```

- RAW függőségek:

  i1 ↔ i2, i2 ↔ i3

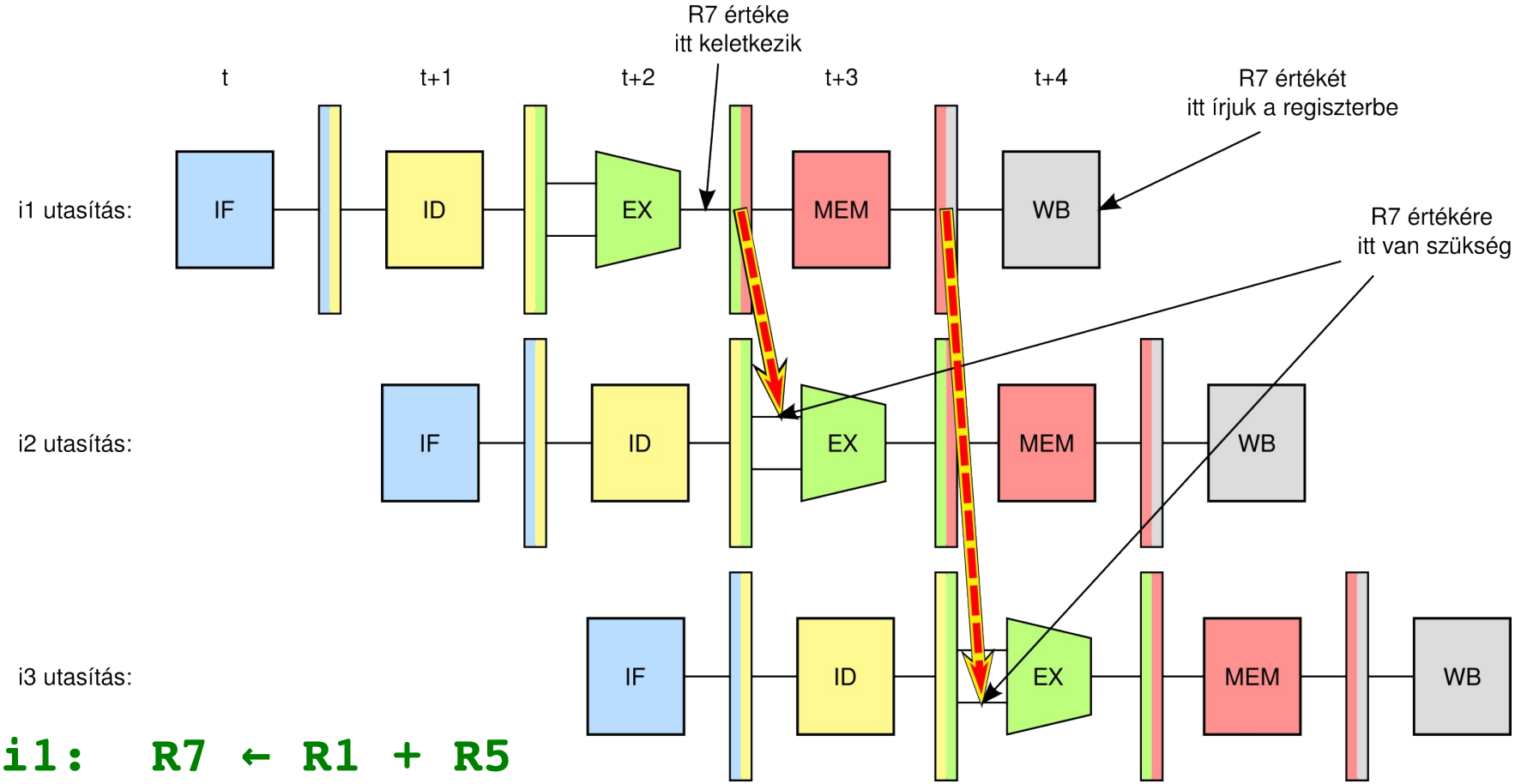
RAW egymásrahatás




```
i1:   R7 ← R1 + R5
i2:   R8 ← R7 + R2
i3:   R5 ← R8 + R7
```

RAW egymásrahatás

- Kész van az az eredmény, csak rossz helyen → ***forwarding***



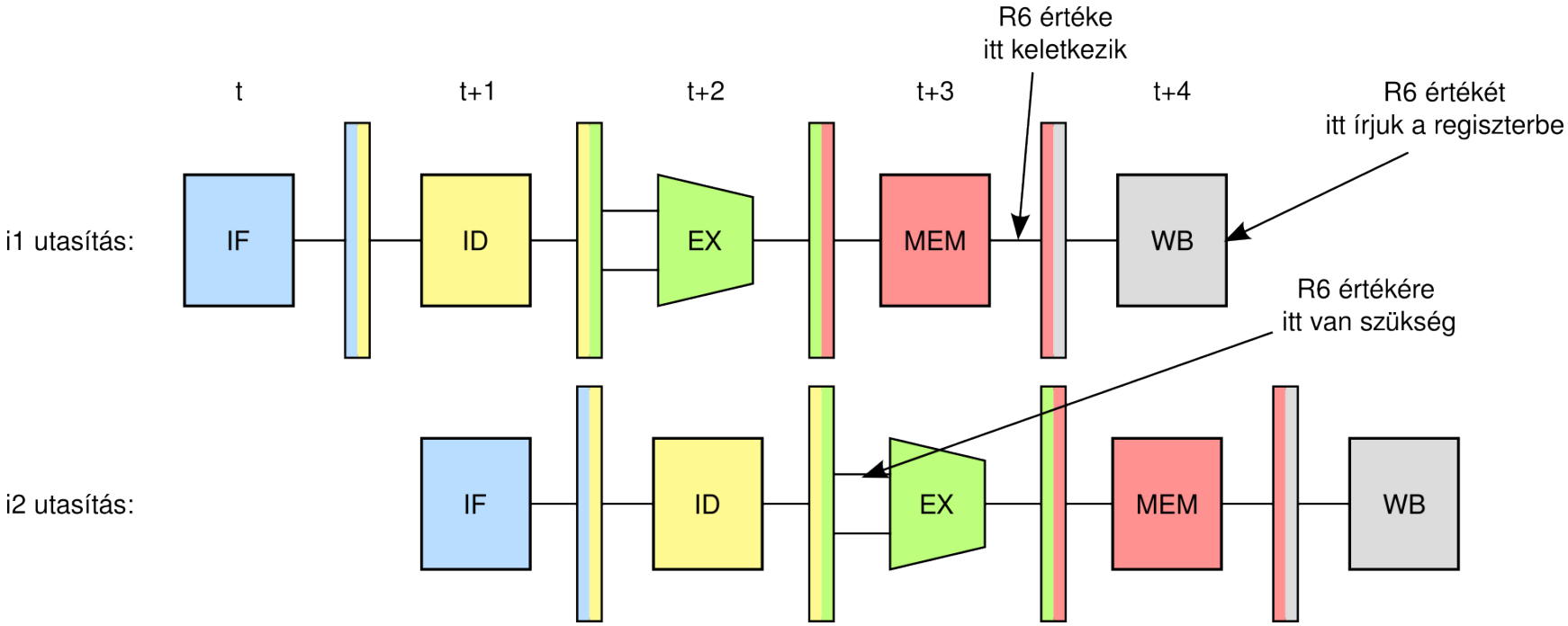

```
i1:   R7 ← R1 + R5
i2:   R8 ← R7 + R2
i3:   R5 ← R8 + R7
```

RAW egymásrahatás

- Forwarding: van, ami ellen nem véd:

```
i1:  R6 ← MEM[R2]
i2:  R7 ← R6 + R4
```

- R6: értéke a MEM végén áll elő
- i2-nek az EX elején kellene

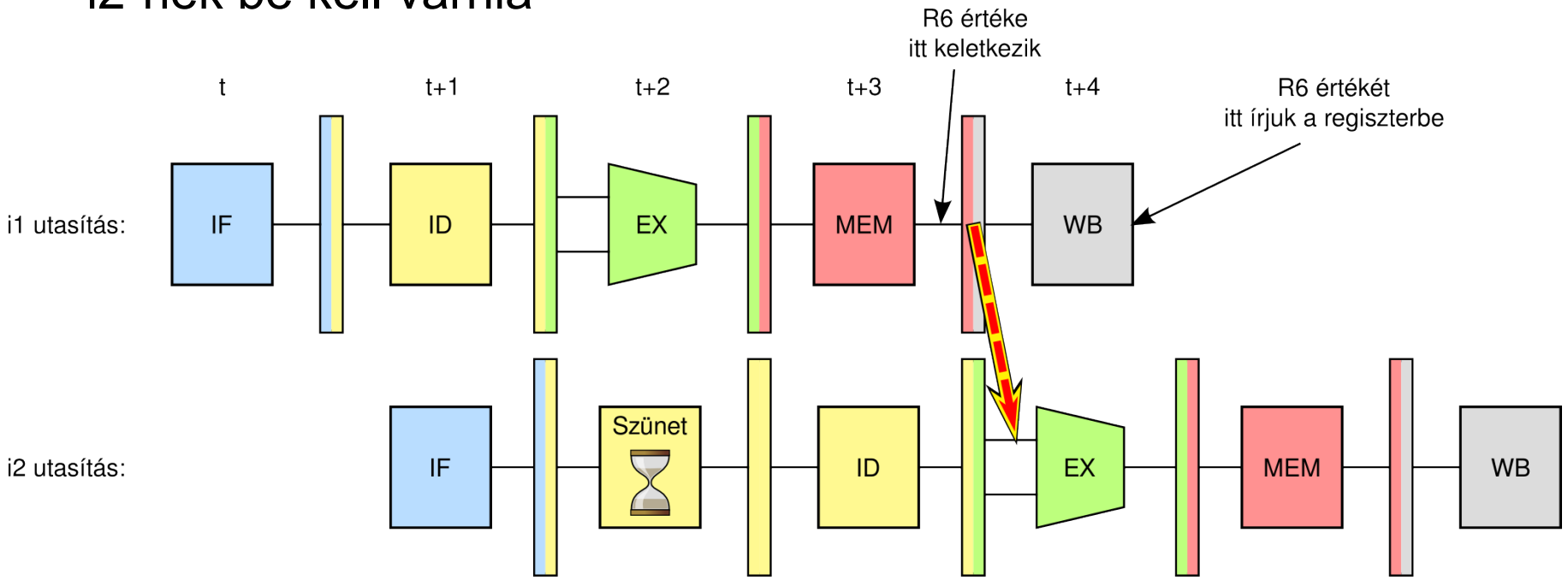


RAW egymásrahatás

- Forwarding: van, ami ellen nem véd:

```
i1:  R6 ← MEM[R2]
i2:  R7 ← R6 + R4
```

- R6: értéke a MEM végén áll elő
- i2-nek az EX elején kellene
- i2-nek be kell várnia



Adat-egymásrahatás

- Nem csak RAW függőség van

```
i1:   R3 ← MEM[R2]
i2:   R1 ← R2 * R3
i3:   R4 ← R1 + R5
i4:   R5 ← R6 + R7
i5:   R1 ← R8 + R9
```

- WAW: i2 ↔ i5
- WAR: i3 ↔ i4

- Most az egyszerű pipeline-unkban csak a RAW gond

Procedurális egymásrahatás

- Feltételes ugró utasításokkal baj van
  - IF betölti őket
  - Ciklusfeltétel & ugrási cím csak az EX-ben derül ki
  - Addig honnan vegyük a további utasításokat?
- Megoldások:
  - Álljunk meg és várjuk meg, míg az EX-be kerül → Intel 80386
  - Tippeljük meg a kimenetelt és az ugrási címet

    → ***elágazásbecslés***

Procedurális egymásrahatás

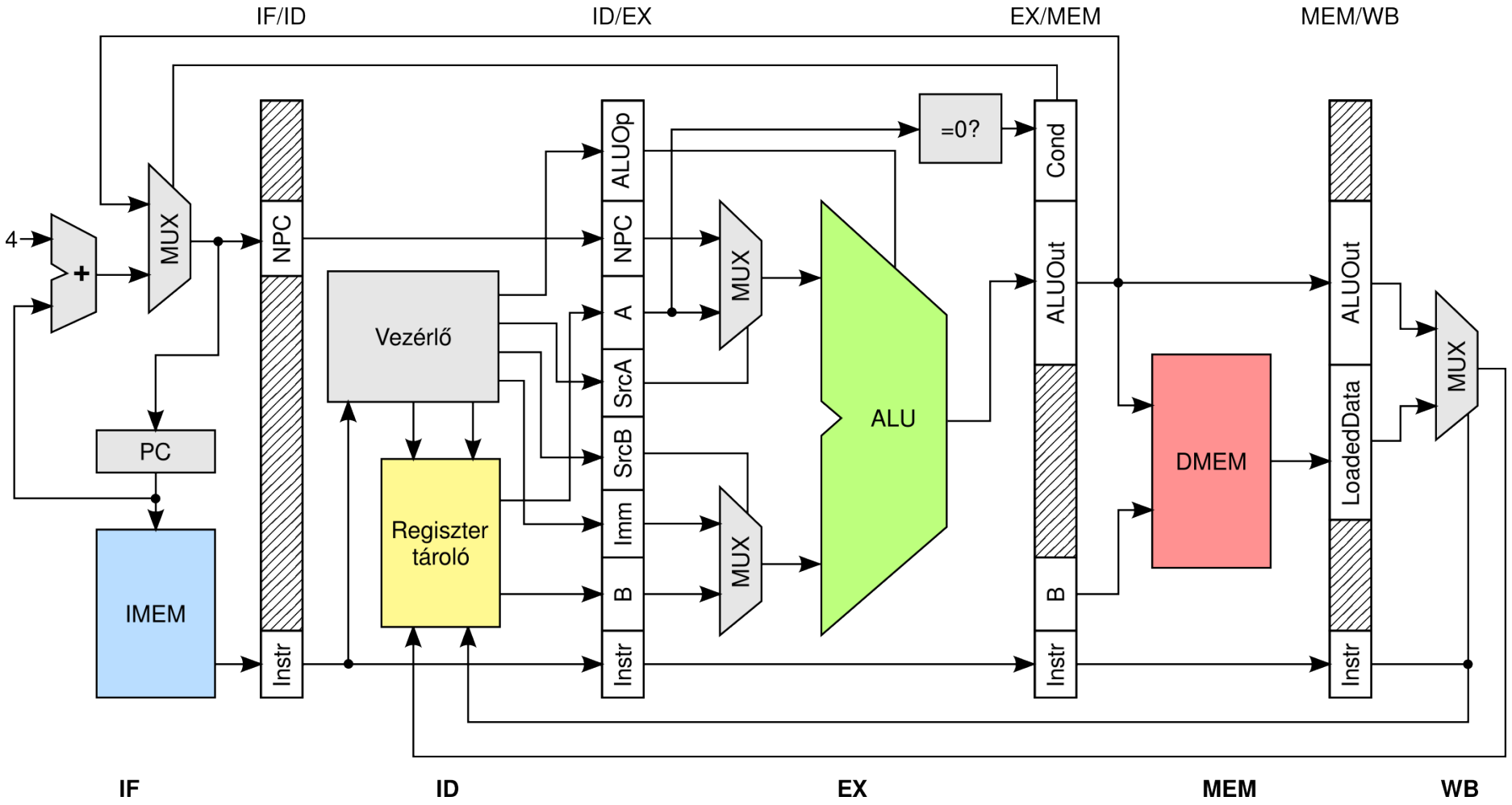
- **Statikus elágazásbecslés** → nem adaptív
  - Ugrás meghiúsulására voksol
    - Ingyen van.

      Ha rossz a tipp, érvénytelenítjük a tévesen behozottakat
  - Ugrás bekövetkezésére voksol
    - De akkor honnan veszi a címet?
    - Akkor jó, ha a cím hamarabb előáll, mint a feltétel
  - Visszafele ugrás: megvalósulást, előre: meghiúsulást tippel
    - Motiváció: ciklusok visszafele ugranak
- **Dinamikus elágazásbecslés**
  - Rátanul minden ugrás viselkedésére
  - Drasztikusan jobb, mint a statikus
  - Nem utópia! Sőt, nem is túl bonyolult!

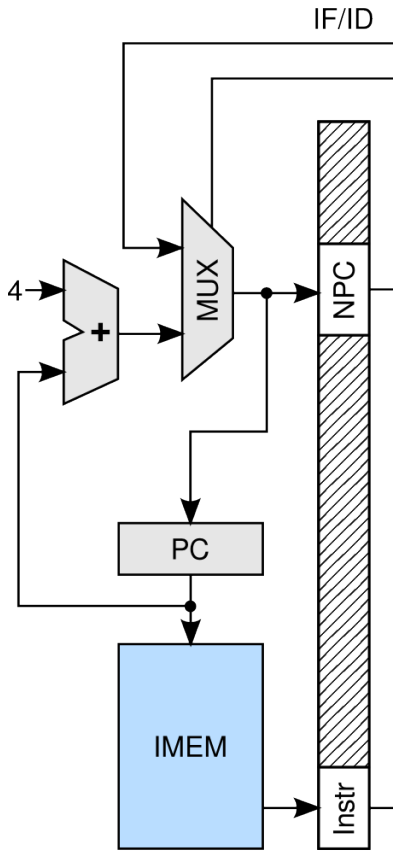
Részösszefoglaló

- Mit tanultunk eddig?
  - Pipeline elvet
  - Fázisok pipeline regisztereken keresztül kommunikálnak
  - Hogy hogyan kell kezelni az egymásrahatásokat
- Csináljunk saját pipeline-os processzort!

Pipeline implementáció

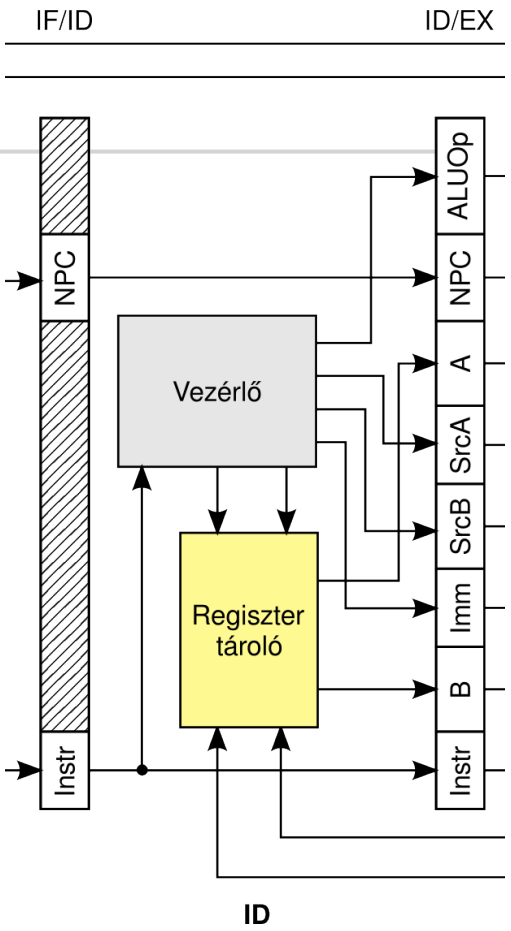


IF fázis



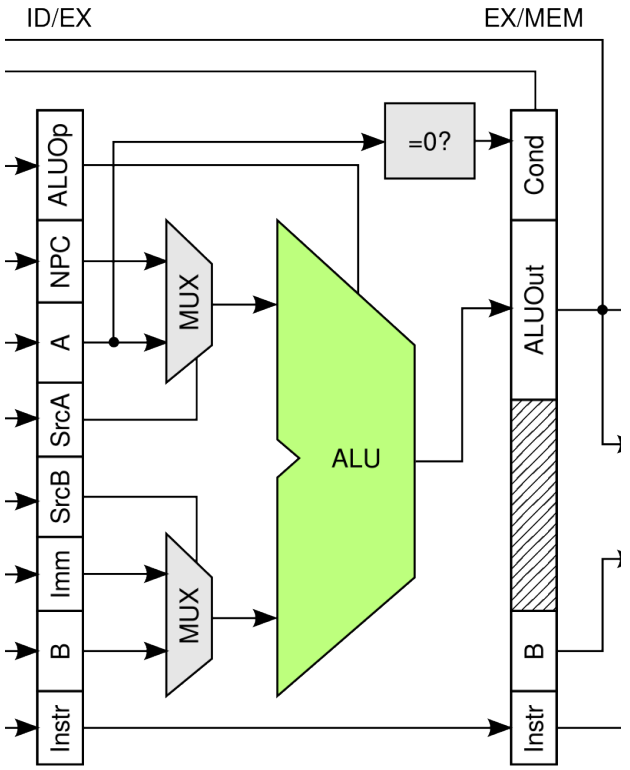

```
IF/ID.Instr ← IMEM[PC]
PC ← SELECT (EX/MEM.Instr.Opcode == branch &&
          EX/MEM.Cond; true: EX.MEM.ALUOut, false: PC+4)
IF/ID.NPC ← PC
```

ID fázis



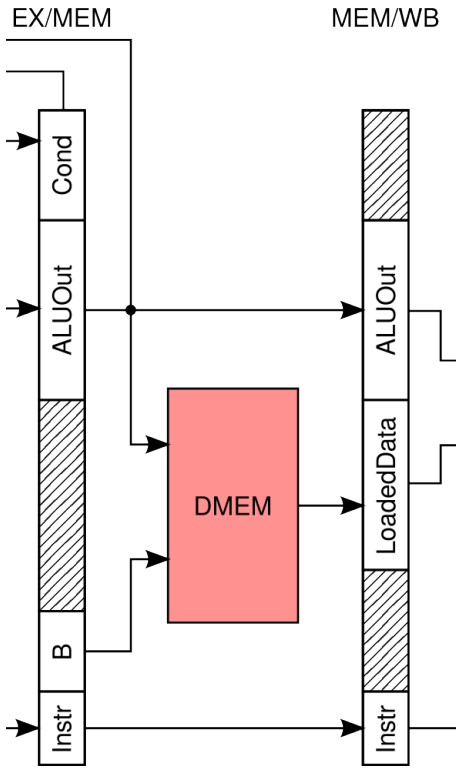

```
ID/EX.NPC    ← IF/ID.NPC
ID/EX.Instr  ← IF/ID.Instr
ID/EX.Imm    ← IF/ID.Instr [imm]
ID/EX.A      ← Reg [IF/ID.Instr [ra]]
ID/EX.B      ← Reg [IF/ID.Instr [rb]]
ID/EX.SrcA   ← SELECT (IF/ID.Instr.Opcode == branch;
                  true: pc, false: reg)
ID/EX.SrcB   ← SELECT (IF/ID.Instr.HasImm;
                  true: imm, false: reg)
ID/EX.ALUOp  ← SELECT (IF/ID.Instr.Opcode == arithm;
                  true: IF/ID.Instr.Func, false: "+")
```

EX fázis



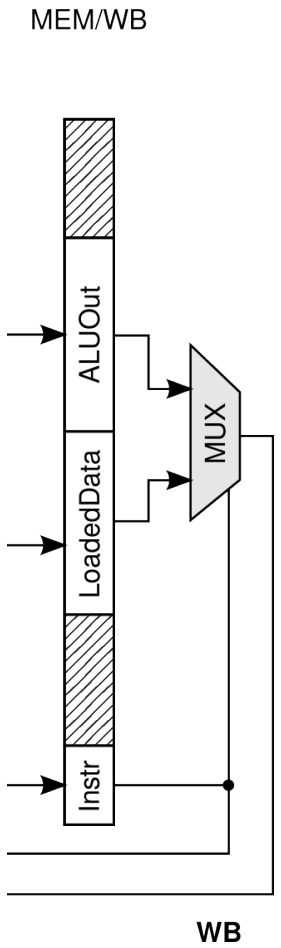

```
EX/MEM.Instr ← ID/EX.Instr
ALU.A ← SELECT (ID/EX.SrcA; reg: ID/EX.A, pc: ID/EX.NPC)
ALU.B ← SELECT (ID/EX.SrcB; reg: ID/EX.B, imm: ID/EX.Imm)
ALU.Op ← ID/EX.ALUOp
EX/MEM.ALUOut ← ALU.Out
EX/MEM.Cond ← ID/EX.A == 0
EX/MEM.B ← ID/EX.B
```

MEM fázis



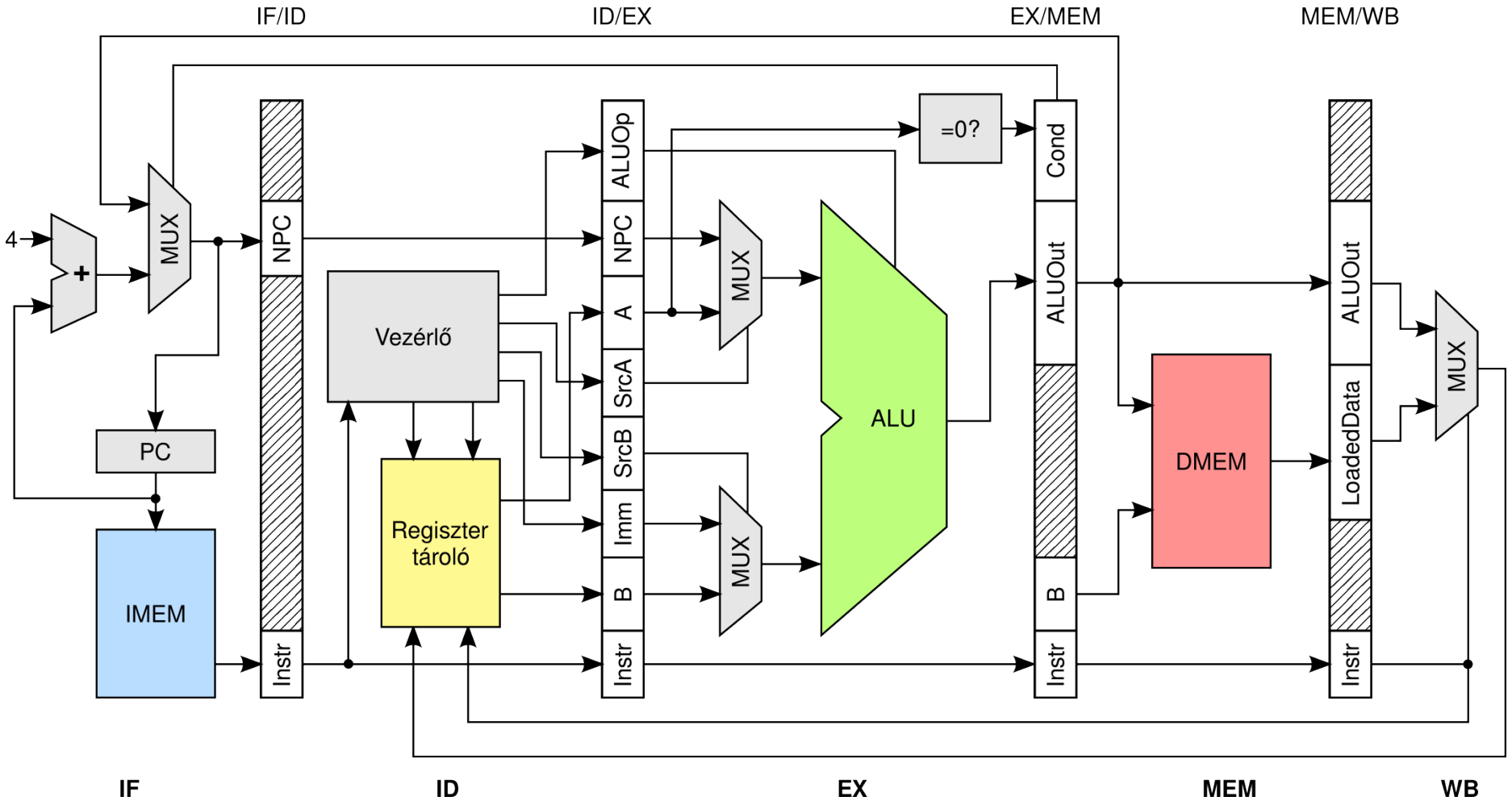

```
MEM/WB.Instr ← EX/MEM.Instr
if (ID/EX.Instr.Opcode == Load):
    MEM/WB.LoadedData ← MEM [EX/MEM.ALUOut]
if (ID/EX.Instr.Opcode == Store):
    MEM [EX/MEM.ALUOut] ← EX/MEM.B
if (ID/EX.Instr.Opcode == arithm):
    MEM/WB.ALUOut ← EX/MEM.ALUOut
```

WB fázis



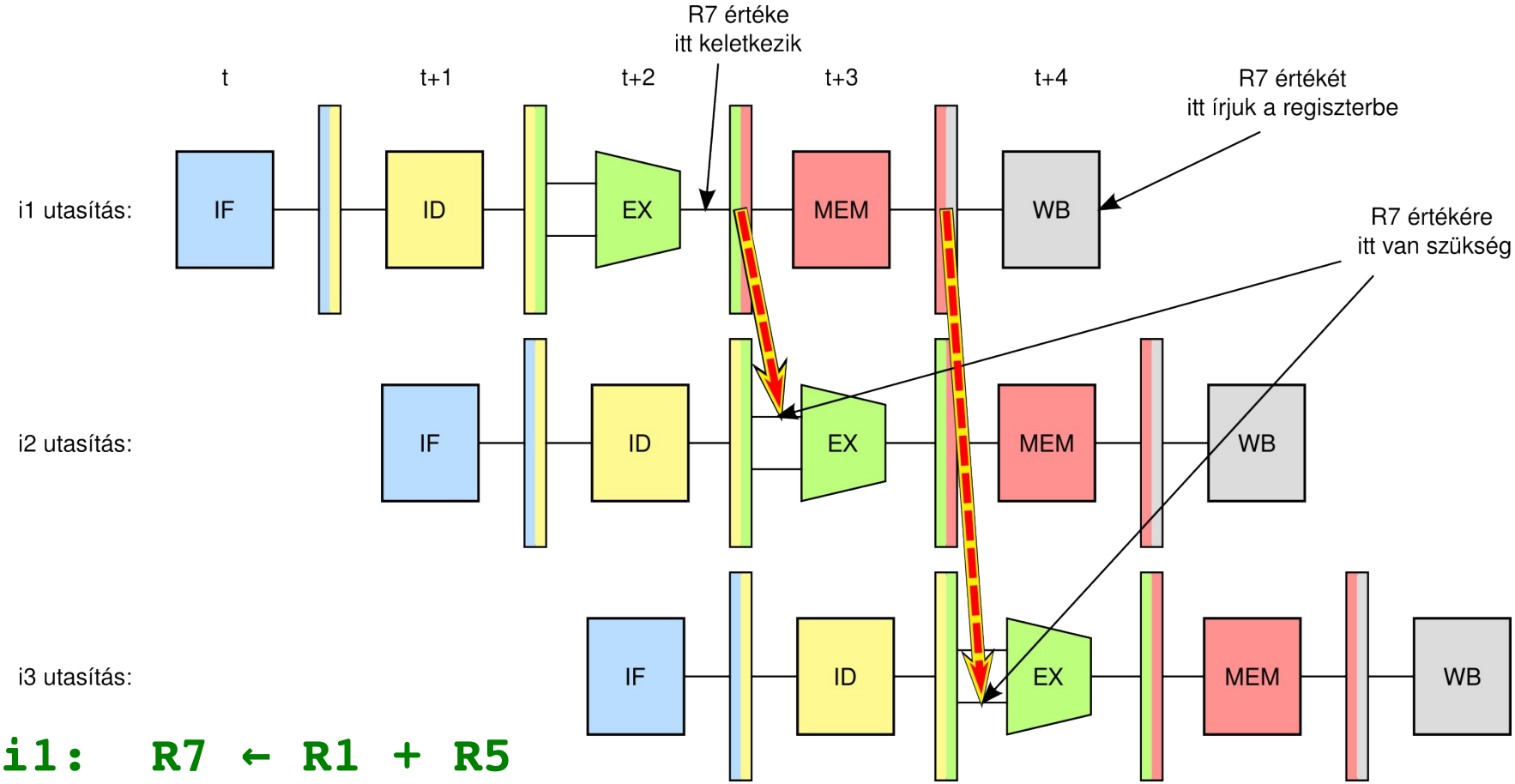

```
Reg [MEM/WB.Instr [rd]] ← SELECT (MEM/WB.Instr.Opcode;
    arithm: MEM/WB.ALUOut,
    Load: MEM/WB.LoadedData)
```

Pipeline implementáció



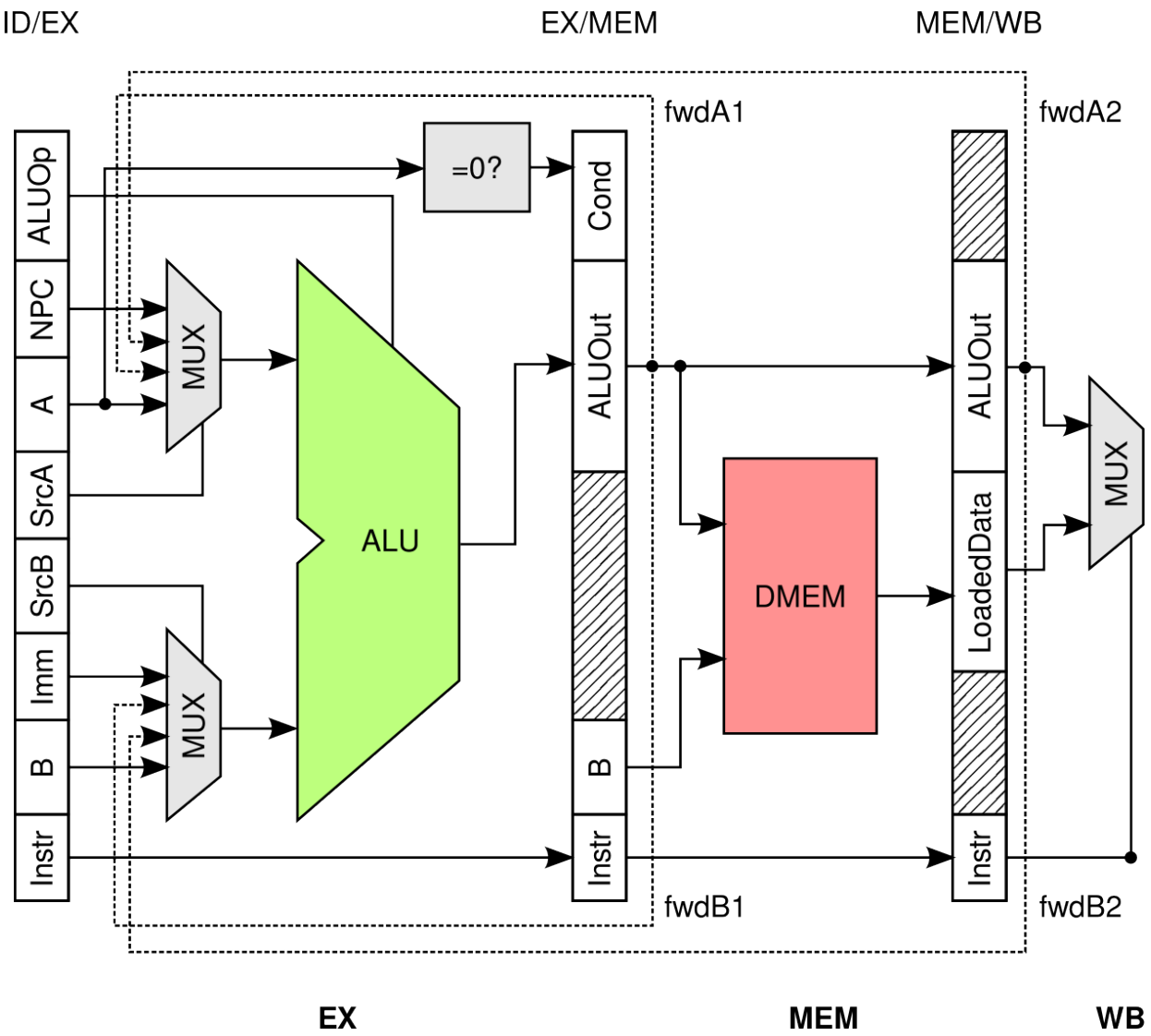
RAW egymásrahatás

- Emlékeztető: forwarding




```
i1:   R7 ← R1 + R5
i2:   R8 ← R7 + R2
i3:   R5 ← R8 + R7
```

RAW egymásrahatás



RAW egymásrahatás

- Honnan tudja az ID, hogy mik lesznek az ALU operandusai?

- ```
  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.HasImm; true: imm,
          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))
  ```

Egyéb egymásrahatások

- Ha szünetet igényel a RAW egymásrahatás:
  - ID az előbbi módon detektálja
  - Szól az IF-nek, hogy most ne hívjon le új utasítást
- Procedurális egymásrahatás:
  - Megáll & vár taktika: 2 szünet beiktatása, mint előbb
  - „Csak nem következik be” taktika:
    - Folytatja a következő utasítások lehívását
    - Ha mégis ugrás történt, a két félig kész utasítást hatástalanítani kell:
      - Utasítások egy „Valid” flag-et cipelnek magukkal
      - Ha kiderül, hogy félbe kell hagyni őket, Valid=0
      - Valid=0-ás utasítást a MEM és a WB fázis semmibe vesz

Részösszefoglaló

- Mit tanultunk eddig?
  - Pipeline elvet
  - Fázisok pipeline regisztereken keresztül kommunikálnak
  - Hogy hogyan kell kezelni az egymásrahatásokat
  - Csináltunk saját pipeline-os processzort
  - A processzorunk már az egymásrahatásokat is kezeli
- Addig jó, amíg valami rendkívüli nem történik
- Mi történhet?
  - Periféria kérés,
  - Laphiba,
  - Védelmi hiba,
  - Érvénytelen utasítás,
  - Stb.

Kivételek, megszakítások

- Szinkron vs aszinkron
  - Szinkron, ha a cpu és a memória ugyanazon állapotában mindig ugyanúgy bekövetkezik
- Maszkolható vs. nem maszkolható
- Utasítások között vs. közben kezelendő
- Folytatható vs. terminál

Esemény	Szinkron/Aszinkron	Maszkolható	Kezelendő	Folytatható
Periféria kérés	Aszinkron	Igen	Ut. között	Igen
Int túlcsordulás	Szinkron	Igen	Ut. közben	Igen
Laphiba	Szinkron	Nem	Ut. közben	Igen
Védelmi hiba	Szinkron	Nem	Ut. közben	Igen
Érvénytelen ut.	Szinkron	Nem	Ut. közben	Igen/Nem
HW hiba	Aszinkron	Nem	Ut. közben	Nem

Pontos kivételkezelés

- Kívánatos viselkedés:
  - Ha az i. utasítás közben keletkezik kivétel
  - Tűnjön úgy, mintha <i utasítások befejeződtek volna
  - És a >i utasítások egyáltalán el sem kezdődtek volna

  → ***pontos kivételkezelés***
- Pipeline feldolgozás esetén nem triviális
- A kivételkor jó lenne megakadályozni, hogy a későbbi utasításoknak hatása legyen
- … de lehet, hogy már meg is történt a baj

Pontos kivételkezelés

- Melyik fázisban mi történhet:
  - 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

	1	2	3	4	5	6
Rk ← Rm + Rn	IF	ID	EX	MEM	WB	
Ri ← MEM [Rj]		IF	ID	EX	MEM	WB

- 1. utasítás: EX fázisban túlcsordulás
- 2. utasítás: IF fázisban laphiba

  → Felborul a sorrend! Egy későbbi utasítás hibája jön előbb!!

Pontos kivételkezelés

- Egy lehetséges megoldás:
  - A kivételt nem kezeljük azonnal
  - Csak feljegyezzük, hogy volt (utasítás görgeti maga előtt)
  - Tényleges kezelés: WB fázisban, vagy utána

    → sorrend garantáltan helyes marad!

	1	2	3	4	5	6
Rk ← Rm + Rn	IF	ID	EX	MEM	WB	
Ri ← MEM [Rj]		IF	ID	EX	MEM	WB

Eltérő idejű aritmetikai műveletek

- Áttekintettük, mi a helyzet a kivételekkel
- Újabb csavar: mi van, ha az EX fázis nem mindenkinek egyforma hosszú?

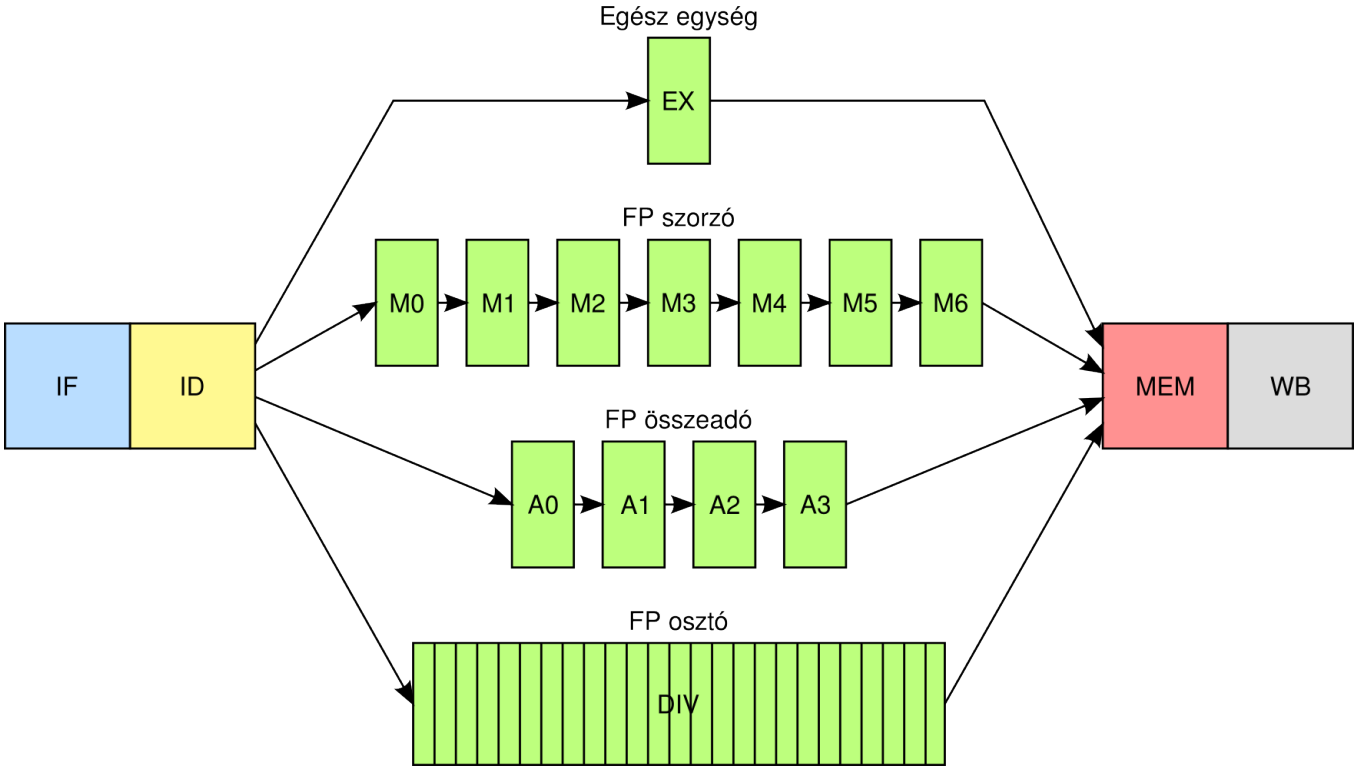
Eltérő idejű aritmetikai műveletek

- Aritmetikai műveletek ***késleltetése***:
  - Lebegőpontos műveletek tovább tartanak
  - Szorzás tovább tart, mint az összeadás
  - Osztás még tovább tart
- A műveleti egységek lehetnek maguk is pipeline szervezésűek
  - Pl. az összeadás 3 ciklus késleltetésű,
  - ...de minden ciklusban új összeadást tud kezdeni

  → ***Iterációs idő*** = 1

Eltérő idejű aritmetikai műveletek

Egység	Késleltetés	Iterációs idő
Integer ALU	1	1
FP összeadó	4	1
FP szorzó	7	1
FP osztó	25	25



Eltérő idejű aritmetikai műveletek

- Érdekesség:
  - Végrehajtás kezdete: ***in-order***, vége: ***out-of-order*** is lehet!

D4 ← D1 * D5	IF	ID	M0	M1	M2	M3	M4	M5	M6	MEM	WB
D2 ← D1 + D3		IF	ID	A0	A1	A2	A3	MEM	WB		
D0 ← MEM [R0+4]			IF	ID	EX	MEM	WB				
MEM [R0+8] ← D5				IF	ID	EX	MEM	WB			

- Nem volt gond, utasítások függetlenek
- Nem lesz mindig így
- ID egység új feladatai:
  - a program szemantikájának megőrzése
  - újfajta egymásrahatások kezelése

Újfajta feldolgozási egymásrahatások

Utasítás	1	2	3	4	5	6	7	8	9	10	11
D4 ← D1 * D5	IF	ID	M0	M1	M2	M3	M4	M5	M6	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
D2 ← D1 + D3				IF	ID	A0	A1	A2	A3	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
D0 ← MEM [R0+4]							IF	ID	EX	MEM	WB

- Gond:
  - 10.: egyszerre hárman MEM-ben
  - 11.: egyszerre hárman WB-ben
- Megoldás:
  - ID tudja, mi mennyi ideig tart → előre látja ezt a helyzetet
  - Ha valaki feldolgozási egymásrahatást okozna → parkoltatja

Több RAW egymásrahatás

- Adatfüggőségek még nagyobb gondot okoznak
- Oka: hosszabb pipeline:
  - Későn áll elő az eredmény

    → ha valaki a korábbi eredményétől függ, többet kell várnia
  - Több utasítás van feldolgozás alatt

    → nagyobb a sansz a RAW függőségre

Utasítás	1	2	3	4	5	6	7	8	9	10	11	12	14	14	15	16	17
D4 ← MEM [R0+4]	IF	ID	EX	ME	WB												
D0 ← D4 * D6		IF	ID	**SZ**	M0	M1	M2	M3	M4	M5	M6	ME	WB				
D2 ← D0 + D8			IF	**SZ**	ID	**SZ**	**SZ**	**SZ**	**SZ**	**SZ**	**SZ**	A0	A1	A2	A3	ME	WB
MEM [R0+4] ← D2					IF	**SZ**	**SZ**	**SZ**	**SZ**	**SZ**	**SZ**	ID	EX	**SZ**	**SZ**	**SZ**	ME

WAW egymásrahatások

- WAW függőség:
  - Két utasítás ugyanabba a regiszterbe menti az eredményét
- Ha megengedjük, hogy az utasítások sorrenden kívül fejeződjenek be → baj lehet!
  - Mert lehet, hogy a végén a korábbi eredménye kerül bele!
  - Az ID fázisnak detektálnia, és szünetekkel kezelnie kell

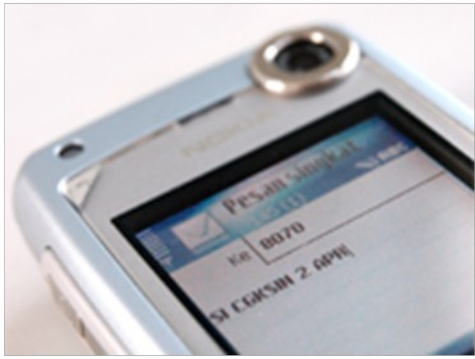
Utasítás	1	2	3	4	5	6	7	8
D2 ← D1 + D3	IF	ID	A0	A1	A2	A3	MEM	WB
...		IF	ID	EX	MEM	WB		
D2 ← MEM [R0+4]			IF	ID	EX	MEM	WB	

- Láthatóan rossz.
  - Megoldás: ID 2 szünetet iktat be a 3. utasításnál

Konklúzió

- Szép, elegáns alapelv
- Nehézségek:
  - Egymásrahatások / függőségek
  - Kivételek
  - Különböző feldolgozási idők

  → Minél hosszabb a pipeline, annál nagyobb a fejfájás



Számítógép Architektúrák

Szuperskalár processzorok

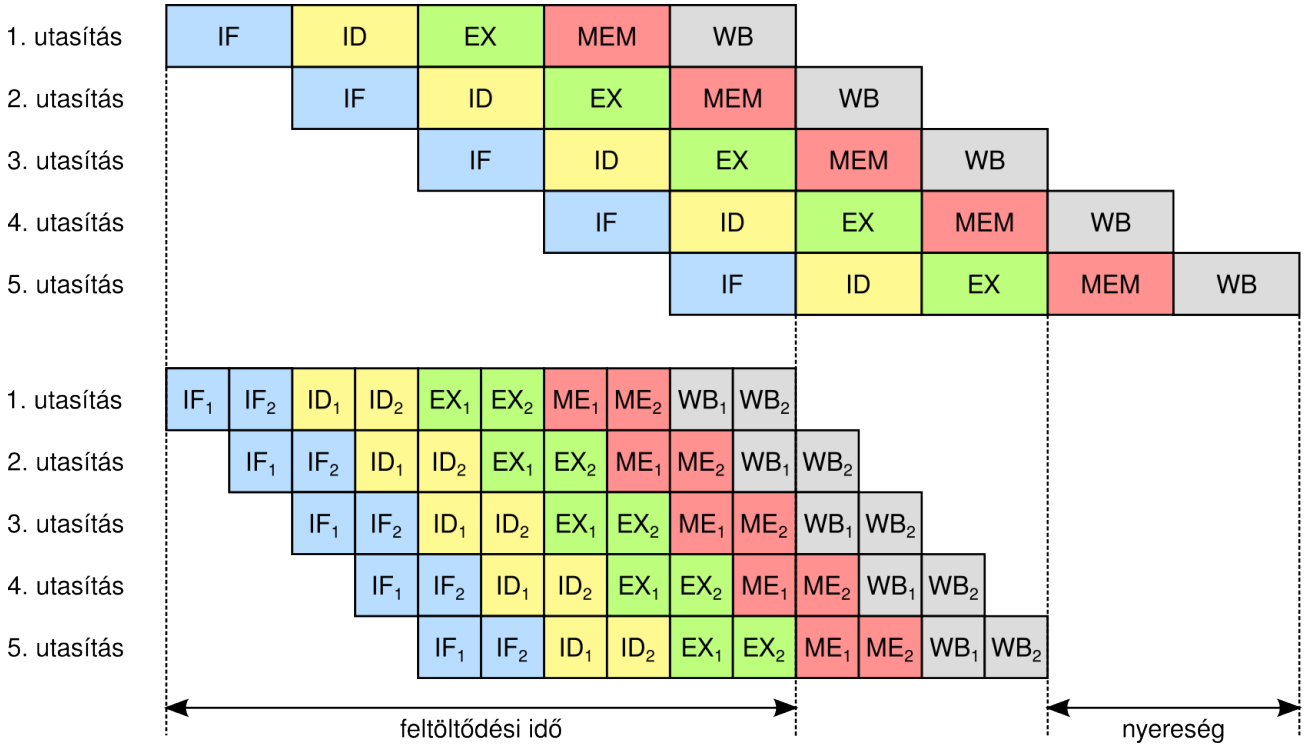
Horváth Gábor
docens
BME Híradástechnikai Tanszék
ghorvath@hit.bme.hu

2012. november 23.
Budapest

Utasításfeldolgozás gyorsítása

1. Lehetőség: Mélyebb pipeline

- Ciklusidőt felére csökkentjük (órajel 2x)
- Ehhez minden fázist két részre kell bontanunk
- Megdupláztuk az átviteli sebességet!



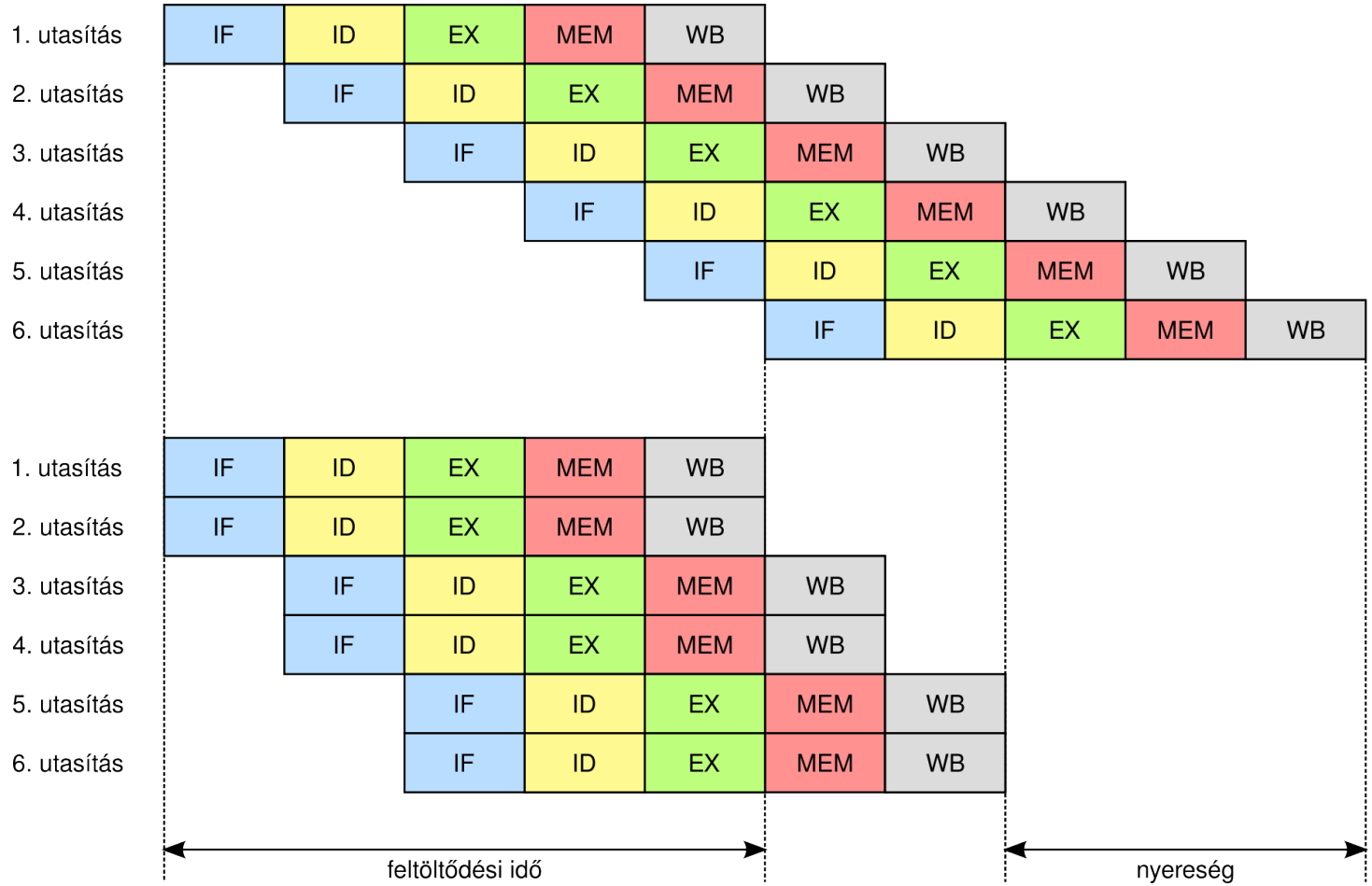
Utasításfeldolgozás gyorsítása

2. Lehetőség: Szélesebb pipeline

- Minden fázisban több utasítást dolgozunk fel
- Ha minden fázisban m új utasítás végrehajtása kezdődik meg

  → ***m-utas szuperskalár processzor***

Szélesebb pipeline

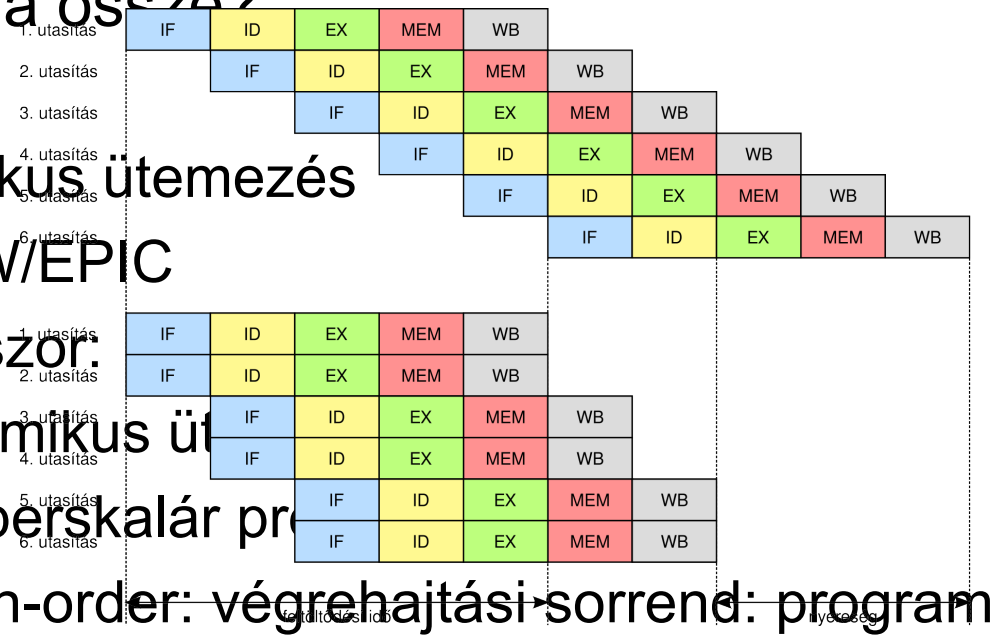


Értékelés

- Mélység: k-szoros, szélesség: m-szeres
  → Elméletileg m*k-szoros gyorsulás
- Gyakorlatilag több korlát:
  - m*k nagy: sor utasítás fut egyszerre → sok az egymásrahatás → sokszor kell megállni a feloldáshoz → romló hatékonyság
  - széles pipeline → m-el négyzetesen növő forwarding utak
  - k hiába nagy → órajel nem lehet tetszőlegesen nagy
    - pl. pipeline regiszter írás/olvasás bele kell férjen
  - k nagy: rossz spekulatív döntések drasztikus hatása
    - pl. rossz elágazásbecsléskor sor tévesen elkezdett utasítás
- Tipikus értékek:
  - Mélység: 5-30
  - Szélesség: 1-6

A pipeline szélesítése

- Hatékonyság kulcsa:
  - Elég sok független utasítás
- Ki válogatja össze?
  - Fordító:
    - Statikus ütemezés
    - VLIW/EPIC
  - Processzor:
    - Dinamikus üt
    - Szuperskalár pr
      - In-order: végrehajtási sorrend: program követése
      - Out-of-order:
        - » átrendezi az utasításokat
        - » program szemantikáját megtartja



In-order vs. Out-of-order

- Példa:

```
i1: R1 ← R2 + R3
i2: R4 ← R1 – R5
i3: R7 ← R8 – R9
i4: R0 ← R2 – R3
```

- 2-utas esetben:

In-order:

Órajel	Utasítások
1:	i1
2:	i2, i3
3:	i4

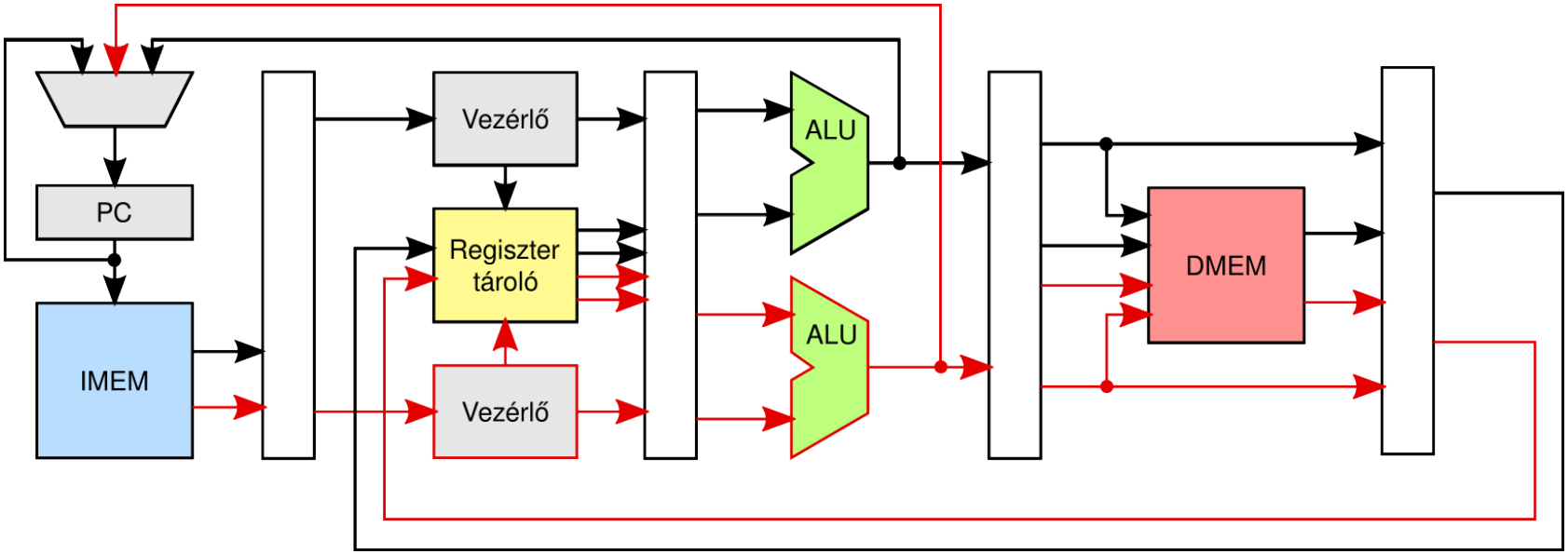
Out-of-order:

Órajel	Utasítások
1:	i1, i3
2:	i2, i4

In-order szuperskalár processzorok

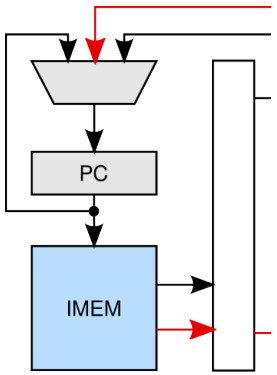
In-order szuperskalár pipeline

Hagyományos egy-utas nem szuperskalár processzor



Két-utas in-order szuperskalár processzor

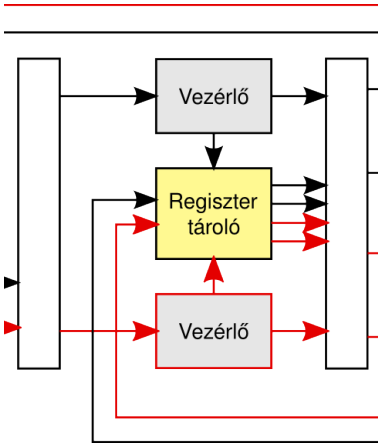
IF – Utasítás lehívás



- Most *m* utasítást kell lehívnia
- Ha az *m* utasítás sorban jön:
  - Semmi gond:
    - Memória szószélesség növelése
      (eddiginél több bájt lehívása a cache-ből)
- Ha ugrást tartalmaz:
  - Ez már nehezebb
    - Kiolvas → elágazást becsül → kiolvas: nincs rá idő!
    - Az elágazásig hívjuk le az utasításokat
    - Romlik a hatékonyság

ID – Dekódolás

- Most *m* utasítást kell dekódolnia.
- Mik is a dekódolás feladatai?
  - ALU vezérlőjelek előállítása
    - Nem nehezebb *m*-re mint 1-re
  - Operandusok kiolvasása
    - Csak 1 regiszter tárolónk van!
    - 2*m kiolvasásra nincs idő
    - Több portos regisztertároló kell
      - Bonyolult
  - Adatfüggőségek detektálása
    - ...folyt...



ID – Dekódolás

- Adatfüggőségek detektálása
  - m-el négyzetesen bonyolódik!
  - Az 1-es úton lévő utasítás:
    - 1-es operandusreg. == 1-es úton az előbbi utasítás eredm. reg.?
    - 2-es operandusreg. == 1-es úton az előbbi utasítás eredm. reg.?
    - 1-es operandusreg. == 1-es úton a kettővel előbbi utasítás eredm. reg.?
    - 2-es operandusreg. == 1-es úton a kettővel előbbi utasítás eredm. reg.?
    - 1-es operandusreg. == 2-es úton az előbbi utasítás eredm. reg.?
    - 2-es operandusreg. == 2-es úton az előbbi utasítás eredm. reg.?
    - 1-es operandusreg. == 2-es úton a kettővel előbbi utasítás eredm. reg.?
    - 2-es operandusreg. == 2-es úton a kettővel előbbi utasítás eredm. reg.?
  - A 2-es úton lévő utasítás:
    - … ugyanez ...

ID – Dekódolás

- Szünetek kezelése

  1. ha a korábbi utasítás kényszerül szünetre

     → az újabb nem előzheti meg (in-order elv)

Utasítás	1	2	3	4	5	6	7
i1: ...	IF1	ID1	EX1	MEM1	WB1		
i2: R1 ← MEM [R0+4]	IF2	ID2	EX2	MEM2	WB2		
i3: R2 ← R1 + 32		IF1	ID1	szün	EX1	MEM1	WB1
i4: R3 ← R0 * R4		IF2	ID2	szün	EX2	MEM2	WB2

ID – Dekódolás

- Szünetek kezelése

  2. ha az újabb utasítás kényszerül szünetre
     - a korábbi folytatódhat
     - bejöhet valaki a helyére?

- Igen:

Utasítás	1	2	3	4	5	6	7	8
i1: ...	IF1	ID1	EX1	MEM1	WB1			
i2: R1 ← MEM [R0+4]	IF2	ID2	EX2	MEM2	WB2			
i3: R3 ← R0 * R4		IF1	ID1	EX1	MEM1	WB1		
i4: R2 ← R1 + 32		IF2	ID2	szün	EX2	MEM2	WB2	
i5: ...			IF1	ID1	EX1	MEM1	WB1	
i6: ...			IF2	szün	ID2	EX2	MEM2	WB2

ID – Dekódolás

- Szünetek kezelése

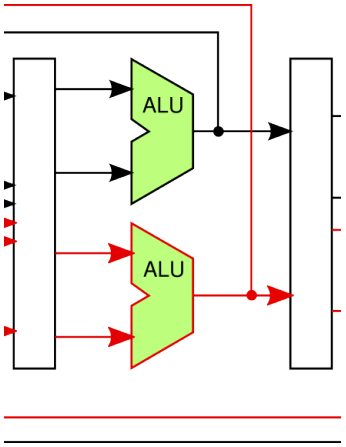
  2. ha az újabb utasítás kényszerül szünetre
     - a korábbi folytatódhat
     - bejöhet valaki a helyére?

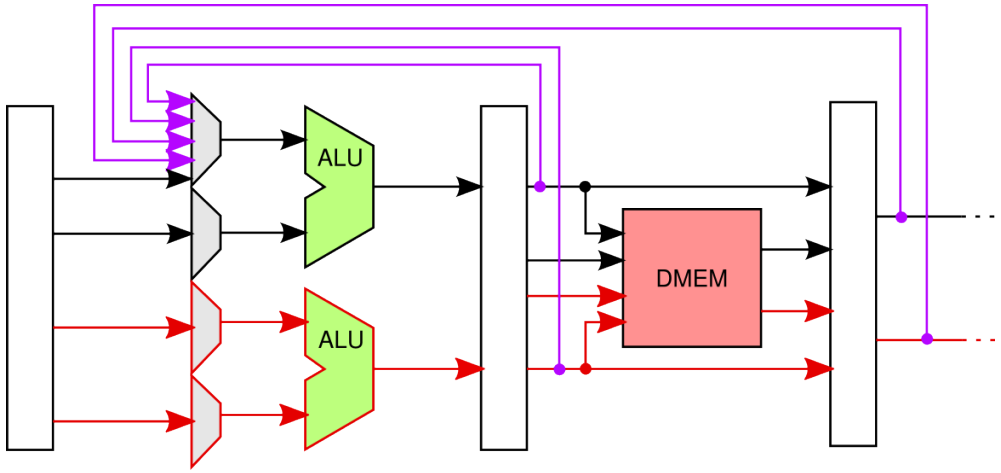
- Nem:

Utasítás	1	2	3	4	5	6	7	8
i1: ...	IF1	ID1	EX1	MEM1	WB1			
i2: R1 ← MEM [R0+4]	IF2	ID2	EX2	MEM2	WB2			
i3: R3 ← R0 * R4		IF1	ID1	EX1	MEM1	WB1		
i4: R2 ← R1 + 32		IF2	ID2	szün	EX2	MEM2	WB2	
i5: ...			IF1	szün	ID1	EX1	MEM1	WB1
i6: ...			IF2	szün	ID2	EX2	MEM2	WB2

EX – Végrehajtás

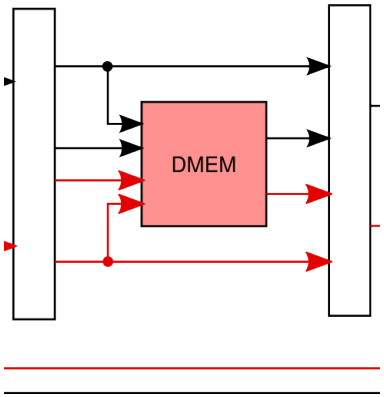
- Most *m* utasítást kell végrehajtania.
- *m* db ALU
  - OK, nem probléma.
- Forwarding utak
  - m^2 van belőle
  - 32 – 64 bites sínek!
  - Nem OK.
  - Komoly probléma!





MEM – Memóriaműveletek

- Most *m* utasítás akarja egyszerre.
- Két lehetőség:
  - 2 portos adat cache
  - Csak 1 memóriaműveletet engedünk egyszerre
    - Ha az m-ből több is igényel MEM fázist, a többi vár (feldolgozási egymásrahatás)



WB – Eredmény regiszter tárolás

- Több portos regiszter tároló kell

Out-of-order szuperskalár processzorok

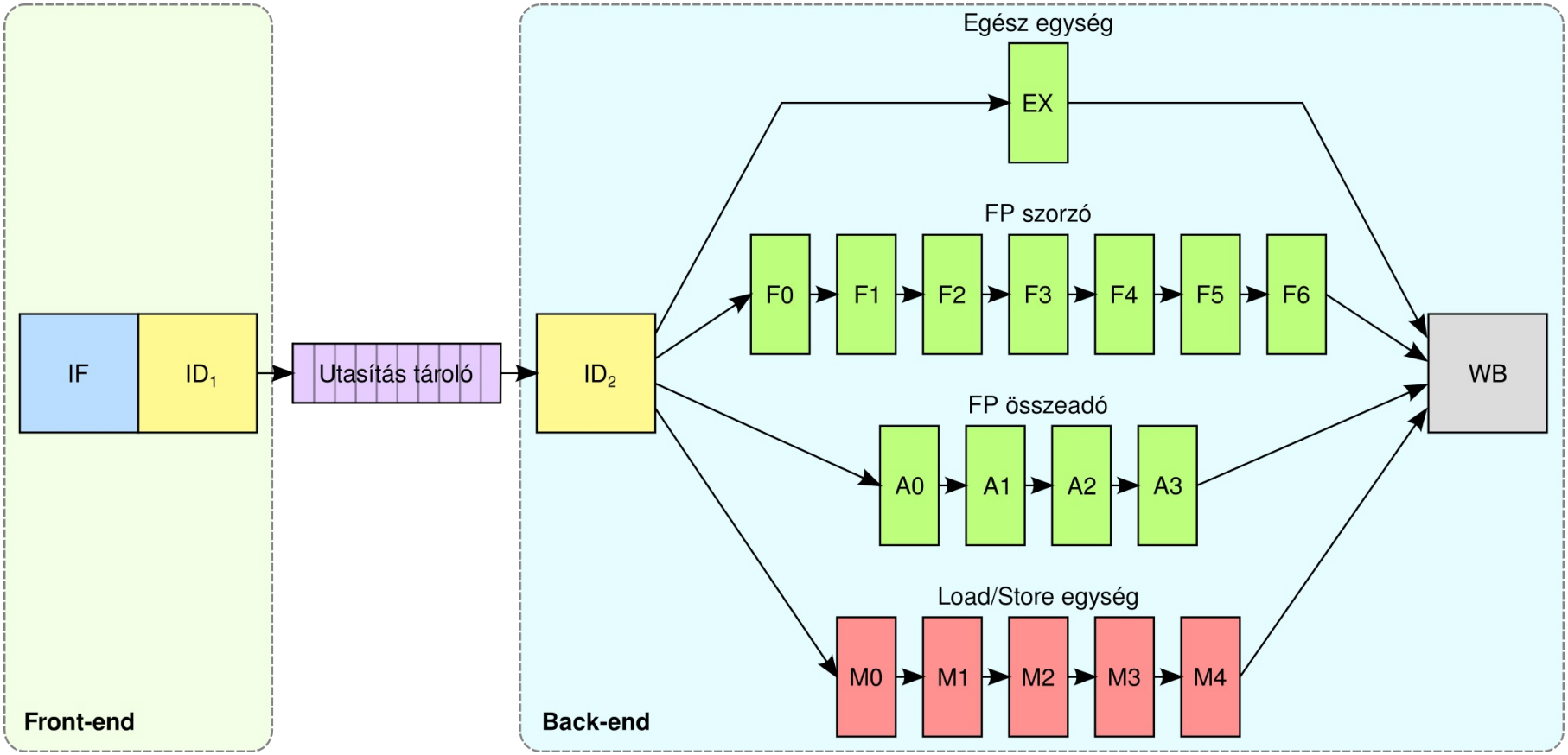
Alapelvek

- Az utasítások ***megelőzhetik*** egymást
- Ez már nem futószalag többé
- Program szemantikája nem változhat
  - Nyilvántartásokat kell vezetni:
    - Az utasítások végrehajtási állapotáról
    - Műveleti egységek foglaltságáról
  - Döntéseket kell hozni
    - Utasítások műveleti egységekhez rendelése

→ Bonyolúlt processzor

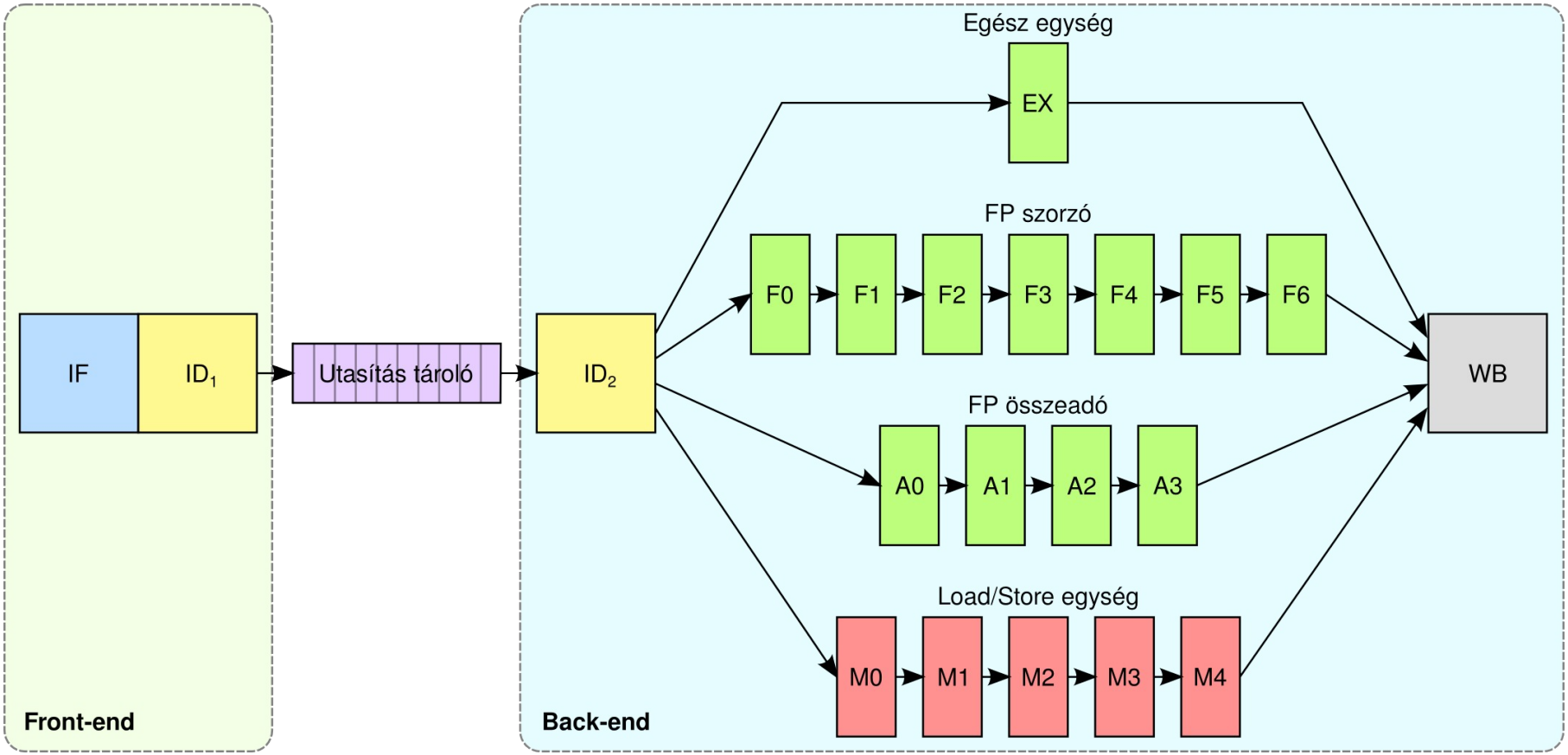
Out-of-order pipeline

- Lehívás (IF), dekódolás (ID1):
  - sorrendben, mint in-order esetben



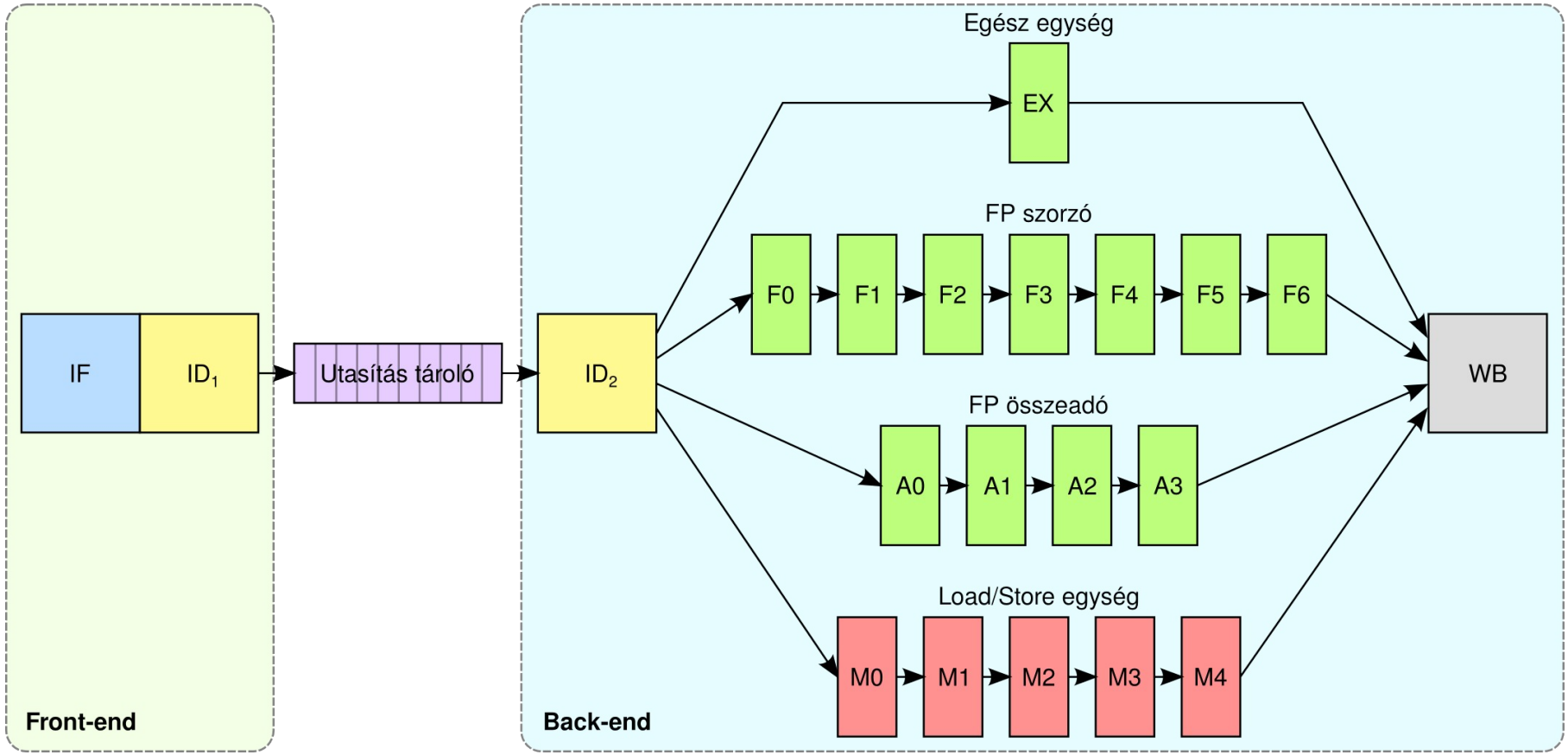
Out-of-order pipeline

- Utasítások átmenetileg egy tárolóba kerülnek (sorban)
- Innen nem feltétlenül sorrendben indul a végrehajtásuk



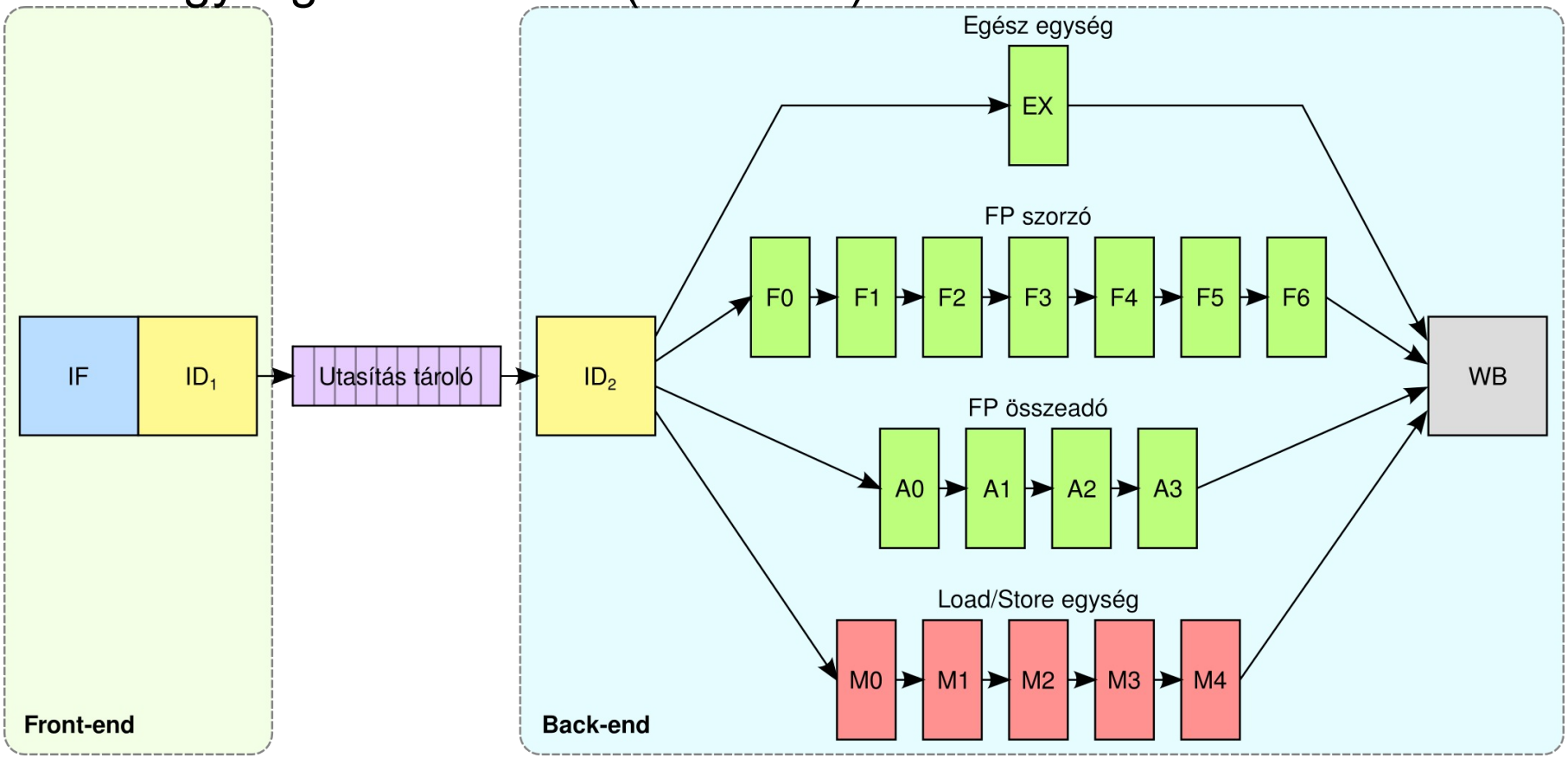
Out-of-order pipeline

- Back-end-be lépés:
  - Sorrenden kívül
  - Szuperskalár esetben egyszerre több végrehajtása is indulhat



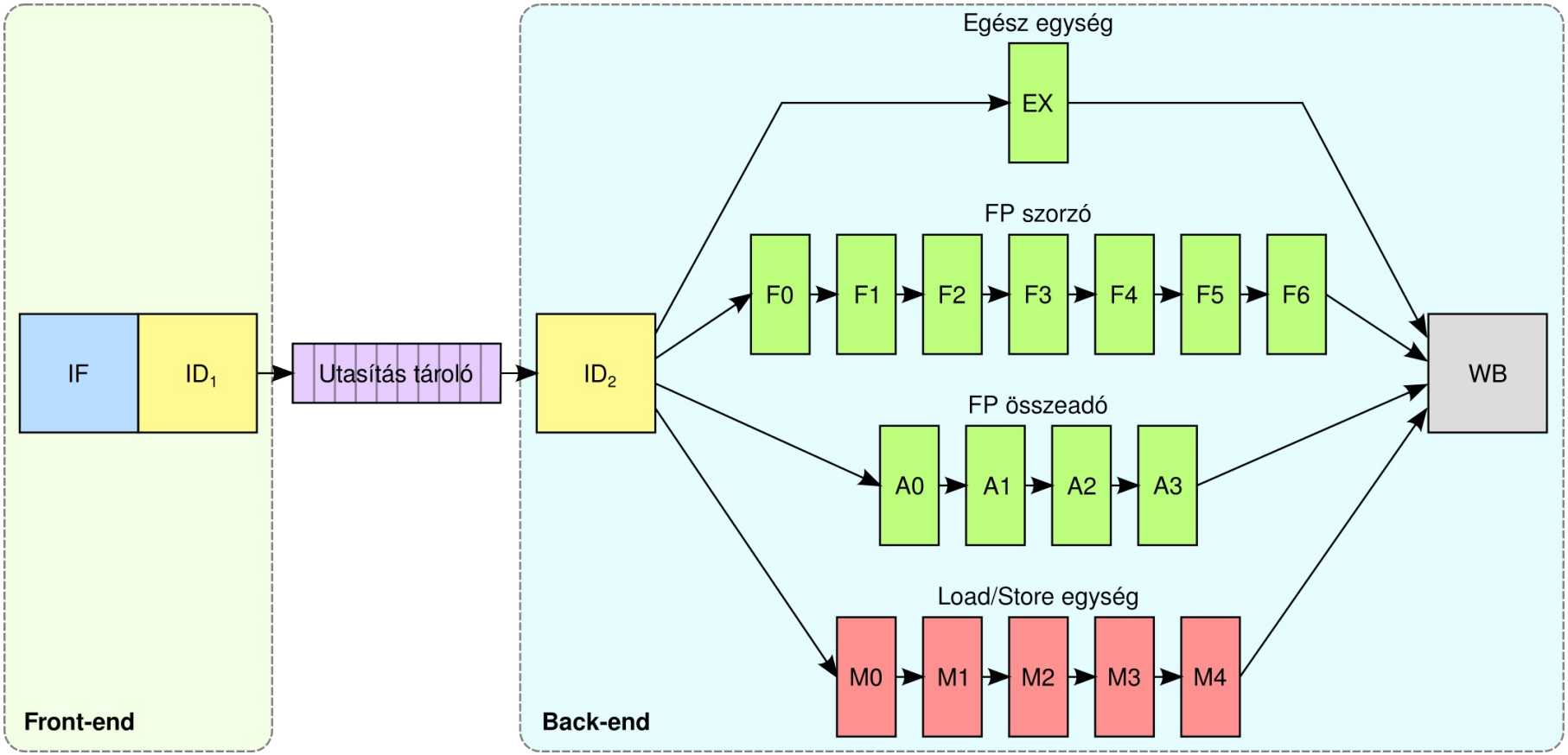
Out-of-order pipeline

- ID ketté bomlik (nem egységes elnevezések!):
  - Dispatch (DS): dekódolás, tárolóba tétel (ábra: ID1)
  - Issue (IS): operandusok összevadászása, műveleti egységhez rendelés (ábra: ID2)



Out-of-order pipeline

- A Load/Store egység csak egy a műveleti egységek közül
- Saját összeadóval maga számol címet



Out-of-order pipeline

- Új problémák:
  - WAW és WAR adat-egymásrahatások is gondot okoznak!

```
i1:   R3 ← MEM[R2]
i2:   R1 ← R2 * R3
i3:   R4 ← R1 + R5
i4:   R5 ← R6 + R7
i5:   R1 ← R8 + R9
```

  - WAW: i2 ↔ i5: nem cserélhetők fel!
  - WAR: i3 ↔ i4: nem cserélhetők fel!

Out-of-order pipeline

- Na jó, de hogy találjuk ki a legjobb sorrendet?
- Néhány régi, jól ismert eljárás dominál
- Kettőt tanulunk:
  - Scoreboard – 1964
    - Mert ezzel kezdődött minden
    - Mert egyszerű
  - Tomasulo – 1967
    - Változatait mind a mai napig használják
    - pl. Intel Core i7

A kezdetek...

- Első out-of-order processzor:
  - 1964: CDC 6600
- Tervező: Seymour Cray
- Tulajdonságok:
  - 16 műveleti egység
  - 5 tonna
  - 10 MHz órajel
  - >400.000 tranzisztor
  - Huzalozott vezérlés
- 5 éven át a leggyorsabb a világon

CDC-6600

Freon alapú hűtés
minden szekrényben:

- Duál vektorgrafikus megjelenítő:

Cray vs. IBM

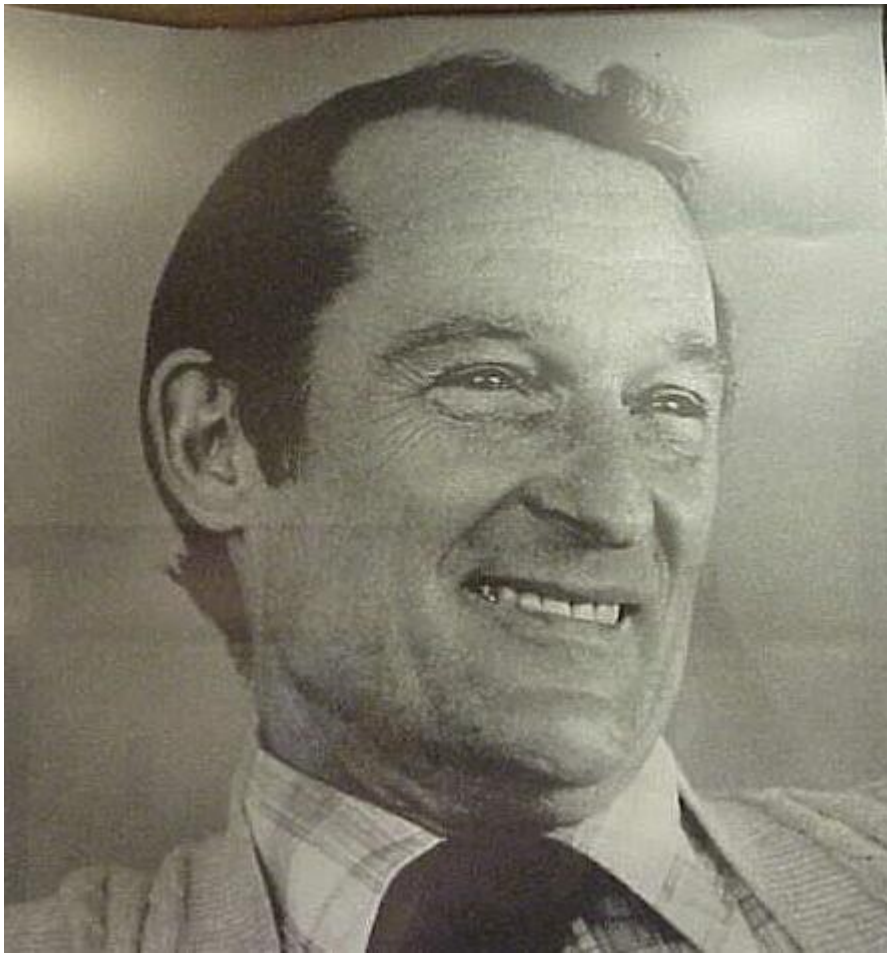
- **Thomas Watson (IBM):**

  „Múlt héten megjelent a CDC 6600. Azt értem, hogy annál a cégnél összesen 34-en dolgoznak a portással együtt. Közülük csak 14 mérnök és csak 4 programozó. Azt már nem értem, hogy a mi hatalmas fejlesztési tevékenységünk mellett hogy veszthettük el a vezető pozíciónkat, hogy hagyhattuk, hogy más építse meg a világ leggyorsabb számítógépét.”

Cray vs. IBM

- **Seymour Cray (CD)**

  „Úgy tűnik, Mr. Watson már meg is válaszolta a kérdését”



Scoreboard

- Utasítások végrehajtása a CDC 6600-ban 5 fázisból áll:
  - IF: lehívás
  - DS: dekódolás & műveleti egységhez rendelés
  - IS: forrásoperandusok kiolvasása
  - EX: végrehajtás
  - WB: regiszter visszaírás
- Egymásrahatások kezelése:
  - WAW: nem engedjük belépni a DS-be

```
D1 ← D2 / D3;   …   ; D1 ← MEM [R0]
```

  - RAW: nem engedjük belépni az IS-be

```
D1 ← D2 / D3;   …   ; D5 ← D1 + D0
```

  - WAR: nem endgedjük belépni a WB-be

```
D5 ← D1 / D3;   …   ; D1 ← MEM [R0]
```

Scoreboard

- A CDC 6600 Scoreboard-ot használ
  - Az utasítások végrehajtási állapotának
  - A műveleti egységek foglaltsági állapotának

  ...nyilvántartására.
- Táblázatok:
  - **Utasítás állapot tábla**: az utasítás melyik fázisban tart
  - **Regiszter állapot tábla**: a regisztert melyik utasítás szándékozik megváltoztatni
  - **Műveleti egység tábla**: minden műveleti egységre tárolja:
    - Foglalt?
    - Mit kell csinálni?
    - Mi a két forrás operandus értéke?
    - Ha valamelyik még nincs kész, melyik művelet eredménye lesz?
    - Hova kell az eredményt írni?

Scoreboard

- Példa:
  - 5 műveleti egység:
    - 1 ALU
    - 1 Load
    - 1 Store
    - 2 Lebegőpontos aritmetikai
      - Késleltetés: 3 ciklus, iterációs idő: 3 ciklus
  - **Nem** szuperskalár
    - Egyesével léptet be utasításokat
    - De legalább out-of-order
    - Triviális lesz szuperskalárrá tenni

Scoreboard példa

Ciklus:

Utasítás állapot tábla					
Utasítás		DS	IS	EX	WB
i1:	D0 ← MEM [R1]				
i2:	D4 ← D0 * D2				
i3:	MEM [R1+16] ← D4				
i4:	R1 ← R1 + 8				
i5:	D0 ← MEM [R1]				
i6:	D4 ← D0 * D2				
i7:	MEM [R1+16] ← D4				

Regiszter állapot tábla	
Regiszter	Műveleti egys.
D0	
D2	
D4	
R1	

Műveleti egység állapot tábla							
Egység	Foglalt	Műv.	R	RA	RB	EA	EB
Egész ALU	Nem						
Load	Nem						
Store	Nem						
FP ALU 1	Nem						
FP ALU 2	Nem						

Scoreboard példa

Ciklus: 1

Utasítás állapot tábla					
Utasítás		DS	IS	EX	WB
i1:	D0 ← MEM [R1]	c1			
i2:	D4 ← D0 * D2				
i3:	MEM [R1+16] ← D4				
i4:	R1 ← R1 + 8				
i5:	D0 ← MEM [R1]				
i6:	D4 ← D0 * D2				
i7:	MEM [R1+16] ← D4				

Regiszter állapot tábla	
Regiszter	Műveleti egys.
D0	Load
D2	
D4	
R1	

Műveleti egység állapot tábla							
Egység	Foglalt	Műv.	R	RA	RB	EA	EB
Egész ALU	Nem						
Load	Igen	load	D0	R1			
Store	Nem						
FP ALU 1	Nem						
FP ALU 2	Nem						

← Műveleti egység foglalás

Scoreboard példa

Ciklus: 2

Utasítás állapot tábla					
Utasítás		**DS**	**IS**	**EX**	**WB**
i1:	D0 ← MEM [R1]	c1	**c2**		
i2:	D4 ← D0 * D2	**c2**			
i3:	MEM [R1+16] ← D4				
i4:	R1 ← R1 + 8				
i5:	D0 ← MEM [R1]				
i6:	D4 ← D0 * D2				
i7:	MEM [R1+16] ← D4				

Regiszter állapot tábla	
Regiszter	**Műveleti egys.**
D0	Load
D2	
D4	**FP1**
R1	

Műveleti egység állapot tábla							
Egység	**Foglalt**	**Műv.**	**R**	**RA**	**RB**	**EA**	**EB**
Egész ALU	Nem						
Load	Igen	load	D0	R1			
Store	Nem						
FP ALU 1	**Igen**	*	**D4**		**D2**	**Load**	
FP ALU 2	Nem						

← **Műveleti egység foglalás**

Scoreboard példa

Ciklus: 3

Utasítás állapot tábla					
Utasítás		**DS**	**IS**	**EX**	**WB**
i1:	D0 ← MEM [R1]	c1	c2	**c3**	
i2:	D4 ← D0 * D2	c2			
i3:	MEM [R1+16] ← D4	**c3**			
i4:	R1 ← R1 + 8				
i5:	D0 ← MEM [R1]				
i6:	D4 ← D0 * D2				
i7:	MEM [R1+16] ← D4				

Regiszter állapot tábla	
Regiszter	**Műveleti egys.**
D0	Load
D2	
D4	FP1
R1	

Műveleti egység állapot tábla							
Egység	**Foglalt**	**Műv.**	**R**	**RA**	**RB**	**EA**	**EB**
Egész ALU	Nem						
Load	Igen	load	D0	R1			
Store	**Igen**	**store**		**R1**			**FP1**
FP ALU 1	Igen	*	D4		D2	Load	
FP ALU 2	Nem						

Műveleti egység foglalás

RAW miatt áll

Scoreboard példa

Ciklus: 4

Utasítás állapot tábla					
Utasítás		DS	IS	EX	WB
i1:	D0 ← MEM [R1]	c1	c2	c3	**c4**
i2:	D4 ← D0 * D2	c2	**c4**		
i3:	MEM [R1+16] ← D4	c3			
i4:	R1 ← R1 + 8	**c4**			
i5:	D0 ← MEM [R1]				
i6:	D4 ← D0 * D2				
i7:	MEM [R1+16] ← D4				

Regiszter állapot tábla	
Regiszter	Műveleti egys.
D0	~~Load~~
D2	
D4	FP1
R1	Egész ALU

← i1 vége

Műveleti egység állapot tábla							
Egység	Foglalt	Műv.	R	RA	RB	EA	EB
Egész ALU	**Igen**	**+**	**R1**	**R1**			
Load	**Nem**						
Store	Igen	store		R1			FP1
FP ALU 1	Igen	*	D4	**D0**	D2	~~Load~~	
FP ALU 2	Nem						

← Műveleti egység foglalás

← Bejegyzés törlése

← D0 betöltve

Scoreboard példa

Ciklus: 5

Utasítás állapot tábla					
Utasítás		**DS**	**IS**	**EX**	**WB**
i1:	D0 ← MEM [R1]	c1	c2	c3	c4
i2:	D4 ← D0 * D2	c2	c4	**c5**	
i3:	MEM [R1+16] ← D4	c3			
i4:	R1 ← R1 + 8	c4	**c5**		
i5:	D0 ← MEM [R1]	**c5**			
i6:	D4 ← D0 * D2				
i7:	MEM [R1+16] ← D4				

Regiszter állapot tábla	
Regiszter	**Műveleti egys.**
D0	**Load**
D2	
D4	FP1
R1	Egész ALU

Műveleti egység állapot tábla							
Egység	**Foglalt**	**Műv.**	**R**	**RA**	**RB**	**EA**	**EB**
Egész ALU	Igen	+	R1	R1			
Load	**Igen**	**load**	**D0**			**ALU**	
Store	Igen	store		R1			FP1
FP ALU 1	Igen	*	D4	D0	D2		
FP ALU 2	Nem						

← **Műveleti egység foglalás**

Scoreboard példa

Ciklus: 6

Utasítás állapot tábla					
Utasítás		DS	IS	EX	WB
i1:	D0 ← MEM [R1]	c1	c2	c3	c4
i2:	D4 ← D0 * D2	c2	c4	**c5+**	
i3:	MEM [R1+16] ← D4	c3			
i4:	R1 ← R1 + 8	c4	c5	**c6**	
i5:	D0 ← MEM [R1]	c5			
i6:	D4 ← D0 * D2				
i7:	MEM [R1+16] ← D4				

Regiszter állapot tábla	
Regiszter	Műveleti egys.
D0	Load
D2	
D4	FP1
R1	Egész ALU

WAW egymásrahatás i2-vel, DS szünet

Műveleti egység állapot tábla							
Egység	Foglalt	Műv.	R	RA	RB	EA	EB
Egész ALU	Igen	+	R1	R1			
Load	Igen	load	D0			ALU	
Store	Igen	store		R1			FP1
FP ALU 1	Igen	*	D4	D0	D2		
FP ALU 2	Nem						

Scoreboard példa

Ciklus: ***7***

Utasítás állapot tábla					
Utasítás		DS	IS	EX	WB
i1:	D0 ← MEM [R1]	c1	c2	c3	c4
i2:	D4 ← D0 * D2	c2	c4	c5+	
i3:	MEM [R1+16] ← D4	c3			
i4:	R1 ← R1 + 8	c4	c5	c6	
i5:	D0 ← MEM [R1]	c5			
i6:	D4 ← D0 * D2				
i7:	MEM [R1+16] ← D4				

Regiszter állapot tábla	
Regiszter	Műveleti egys.
D0	Load
D2	
D4	FP1
R1	Egész ALU

WAR egymásrahatás i3-mal, WB szünet

WAW egymásrahatás i2-vel, DS szünet

Műveleti egység állapot tábla							
Egység	Foglalt	Műv.	R	RA	RB	EA	EB
Egész ALU	Igen	+	R1	R1			
Load	Igen	load	D0			ALU	
Store	Igen	store		R1			FP1
FP ALU 1	Igen	*	D4	D0	D2		
FP ALU 2	Nem						

Scoreboard példa

Ciklus: 8

Utasítás állapot tábla					
Utasítás		DS	IS	EX	WB
i1:	D0 ← MEM [R1]	c1	c2	c3	c4
i2:	D4 ← D0 * D2	c2	c4	c5+	**c8**
i3:	MEM [R1+16] ← D4	c3	**c8**		
i4:	R1 ← R1 + 8	c4	c5	c6	
i5:	D0 ← MEM [R1]	c5			
i6:	D4 ← D0 * D2	**c8**			
i7:	MEM [R1+16] ← D4				

WAR egymásrahatás i3-mal, WB szünet

Regiszter állapot tábla	
Regiszter	Műveleti egys.
D0	Load
D2	
D4	~~FP1~~ FP2
R1	Egész ALU

i2 vége

Műveleti egység állapot tábla							
Egység	Foglalt	Műv.	R	RA	RB	EA	EB
Egész ALU	Igen	+	R1	R1			
Load	Igen	load	D0			ALU	
Store	Igen	store		R1	**D4**		~~FP1~~
FP ALU 1	**Nem**						
FP ALU 2	**Igen**	*	**D4**		**D2**	**Load**	

D4 kész

Bejegyzés törlése

Műveleti egység foglalás

Scoreboard példa

Ciklus: 9

Utasítás állapot tábla					
Utasítás		DS	IS	EX	WB
i1:	D0 ← MEM [R1]	c1	c2	c3	c4
i2:	D4 ← D0 * D2	c2	c4	c5+	c8
i3:	MEM [R1+16] ← D4	c3	c8	**c9**	
i4:	R1 ← R1 + 8	c4	c5	c6	**c9**
i5:	D0 ← MEM [R1]	c5	**c9**		
i6:	D4 ← D0 * D2	c8			
i7:	MEM [R1+16] ← D4				

Feldolgozási egymásrahatás a Store egységre, DS szünet

Regiszter állapot tábla	
Regiszter	Műveleti egys.
D0	Load
D2	
D4	FP2
R1	~~Egész ALU~~

i4 vége

Műveleti egység állapot tábla							
Egység	Foglalt	Műv.	R	RA	RB	EA	EB
Egész ALU	**Nem**						
Load	Igen	load	D0	**R1**		~~ALU~~	
Store	Igen	store		R1	D4		
FP ALU 1	Nem						
FP ALU 2	Igen	*	D4		D2	Load	

Bejegyzés törlése

R1 kész

Scoreboard példa

Ciklus: 10

Utasítás állapot tábla					
Utasítás		DS	IS	EX	WB
i1:	D0 ← MEM [R1]	c1	c2	c3	c4
i2:	D4 ← D0 * D2	c2	c4	c5+	c8
i3:	MEM [R1+16] ← D4	c3	c8	c9	**c10**
i4:	R1 ← R1 + 8	c4	c5	c6	c9
i5:	D0 ← MEM [R1]	c5	c9	**c10**	
i6:	D4 ← D0 * D2	c8			
i7:	MEM [R1+16] ← D4	**c10**			

Regiszter állapot tábla	
Regiszter	Műveleti egys.
D0	Load
D2	
D4	FP2
R1	

WB és a tőle függő DS egyszerre mehet

Műveleti egység állapot tábla							
Egység	Foglalt	Műv.	R	RA	RB	EA	EB
Egész ALU	Nem						
Load	Igen	load	D0	R1			
Store	**Igen**	**store**		**R1**	**D4**		**FP2**
FP ALU 1	Nem						
FP ALU 2	Igen	*	D4		D2	Load	

Bejegyzés törlése, új foglalás

Scoreboard értékelés

- Utasításokat akkor hajtottuk végre, amikor minden operandus rendelkezésre állt

  → Lásd: ***adatáramlásos információfeldolgozási modell***!
- Egyszerű
- Viszonylag hatékony
  - Magas szintű programok (FORTRAN): 1.7x gyorsabbak
  - Alacsony szintű, optimalizált programok: 2.5x gyorsabbak
- Egymásrahatások kezelése:
  - WAW: nem engedjük belépni a DS-be
  - RAW: nem engedjük belépni az IS-be
  - WAR: nem engedjük belépni a WB-be

A Tomasulo algoritmus

A kihívó

- 1967: IBM 360/91-es processzor
- Out-of-order algoritmust Robert Tomasulo tervezte
- A P6 architektúra ma is ennek egy változatát használja (pl. Intel Core i7)
- Miben jobb?
  - Egymásrahatások kezelésében
    - WAW, WAR feloldása: ***regiszter átnevezés***sel
    - Feldolgozási egymásrahatások kiküszöbölése várakozó sorral: ***utasítás tároló***
  - Könnyen kiegészíthető, hogy a külvilág sorrendi végrehajtást lásson: ***sorrendvisszaállító buffer***rel

Egymásrahatások jobb kezelése

- Példa:

```
i1: R1 ← R2 / R3
i2: R4 ← R1 + R5
i3: R5 ← R6 + R7
i4: R1 ← R8 + R9
```

- i1 ↔ i4: WAW, i2 ↔ i3: WAR egymásrahatás
- i4 nem előzheti meg i1-et
  - Pedig de jó is lenne: i1 egy osztás, sokáig tart
- Scoreboard megoldása:
  - WAW: i4-et megakasztja a DS fázisban, bevárja i1-et
  - WAR: i3-mat megakasztja a WB fázisban, bevárja i2-őt
- Tomasulo jobbat talált ki

Egymásrahatások jobb kezelése

- Példa:

```
i1: R1 ← R2 / R3
i2: R4 ← R1 + R5
i3: R5 ← R6 + R7
i4: R1 ← R8 + R9
```

- Miért van WAW? Miért pont R1-be ír i4 is?

  → **Mert kevés a regiszter!**
  Újrafelhasználjuk, ami már nem kell!

- Tegyünk a processzorba új regisztereket
  - Utasításkészlet regiszterei: ***logikai/architekturális regiszterek***
  - Sok-sok privát regiszter (kintről nem látszik): ***fizikai regiszterek***
- ...és ilyen esetben a CPU írja át i4-et úgy, hogy az egy épp nem használt fizikai regiszterbe írjon

Regiszter átnevezés

- ...ez az elv a **regiszter átnevezés**.
- Ez volt:

```
i1: R1 ← R2 / R3
i2: R4 ← R1 + R5
i3: R5 ← R6 + R7
i4: R1 ← R8 + R9
```

- Ez lett:

```
i1: R32 ← R2 / R3
i2: R33 ← R32 + R5
i3: R34 ← R6 + R7
i4: R35 ← R8 + R9
```

(Most R0-R31: architekturális, >R32: fizikai regiszterek)

- i1 ↔ i4: WAW eltűnt!
- i2 ↔ i3: WAR eltűnt!
- Szabad a pálya az utasítás csere-bere előtt!

Regiszter átnevezés

- Hogy csináljuk szisztematikusan?
- Kell egy ***Regiszter leképző tábla***
  - Megmondja, hogy egy architekturális regiszter melyik fizikaihoz van rendelve
- Algoritmus:
  - `Ri ← Rj műv. Rk` átalakítása `Ra ← Rb műv. Rc`-re:
    - Rb = regiszterleképzőtábla[Rj]
    - Rc = regiszterleképzőtábla[Rk]
    - Ra = következő szabad fizikai regiszter
    - regiszterleképzőtábla[Ri] = Ra

Utasítás tároló

- Regisztereket átneveztük, WAW, WAR elfelejtve
- Utasítások az utasítástárolóban várják a végrehajtást
  - Reservation Station, Instruction Window, stb.
  - Lehet *elosztott* (műveleti egységenként külön)
  - Lehet *centralizált* (minden műveleti egységre közös)

CPU	Utasítás tároló típusa	mérete
IBM PowerPC4	Elosztott	31
Intel Pentium III	Centralizált	20
Intel Pentium 4	Hibrid (1 mem. Műv, 1 többi)	126 (72/54)
Intel Core	Centralizált	32
AMD K6	Centralizált	72
AMD Opteron	Elosztott	60

Sorrendvisszaállító buffer

- Regiszterek átnevezve, utasítások sorban állnak
- Végrehajtás soron kívül
  - A program ugyan szemantikailag helyes, de
  - nemkívánatos mellékhatás:
    - Regiszterek/memória nem a program sorrendjében változnak
    - Külső szemlélő v. interrupt nem biztos, hogy erre fel van készülve
- Megoldás: sorrendvisszaállító buffer (ROB)
  - Utasítás belépésekor egy bejegyzést kap
  - Ha a végrehajtás kész, beíródik, hogy
    - Mi az eredménye
    - Melyik regiszterbe / memóriacímre szánja az eredményt
  - Akkor érvényesülnek a változások, ha minden korábbi utasításé már érvényesült!

Tomasulo algoritmus a P6-ban

- Reorder Buffer mezők:
  - V: utasítás eredménye (érték)
  - R: eredményregiszter neve, ide kell majd írni az eredményt
  - Addr: Store esetén erre a címre kell majd írni az eredményt
  - „Ready” bit: kész-e az utasítás végrehajtása
- Reservation Station mezők:
  - Egység: az utasítás által igényelt egység azonosítója
  - Művelet kódja (+, –, *, /, Load, Store, stb.)
  - T: ebbe a ROB bejegyzésbe kell majd írni az eredményt
  - RA, RB: a 2 forrásoperandus
  - EA, EB: ha nincs meg minden forrásoperandus, ebből a ROB bejegyzésből kell majd venni, ha kész lesz

Tomasulo algoritmus a P6-ban

- Regiszter állapot tábla:
  - Minden architekturális regiszterhez egy bejegyzés
    - Vagy a regiszter értéke
    - Vagy a ROB bejegyzés száma, ahol
      - Vagy már meg is van az értéke
      - Vagy még nincs meg az értéke, de majd ott jelenik meg
- CDB: Common data bus
  - Üzenetszórásos busz
  - Ha egy utasítás kész, ezen keresztül kürtöli világgá
  - Aki pont rá várt, így tudja meg, hogy már nem kell várnia

Tomasulo P6

- Példa:
  - 5 műveleti egység:
    - 1 ALU
    - 1 Load
    - 1 Store
    - 2 Lebegőpontos aritmetikai
      - Késleltetés: 3 ciklus, iterációs idő: 3 ciklus
  - Centralizált, 3 bejegyzéses utasítástároló
  - **Nem** szuperskalár
    - Egyesével léptet be utasításokat
    - De legalább out-of-order
    - Triviális lesz szuperskalárrá tenni

Tomasulo P6 példa

Ciklus:

Sorrendvisszaállító buffer (ROB)							
Utasítás		V	R	Addr	IS	EX	CM
i1:	D0 ← MEM [R1]						
i2:	D4 ← D0 * D2						
i3:	MEM [R1+16] ← D4						
i4:	R1 ← R1 + 8						
i5:	D0 ← MEM [R1]						
i6:	D4 ← D0 * D2						
i7:	MEM [R1+16] ← D4						

Regiszter állapot tábla	
Regiszter	Érték/ROB bej.
D0	
D2	
D4	
R1	

CDB	
Érték (V)	ROB bej.

Utasítástároló (RS)							
Foglalt	Egység	Műv.	T	RA	RB	EA	EB
Nem							
Nem							
Nem							

Tomasulo P6 példa

Ciklus: ***1***

Sorrendvisszaállító buffer (ROB)							
Utasítás		V	R	Addr	IS	EX	CM
i1:	D0 ← MEM [R1]		D0				
i2:	D4 ← D0 * D2						
i3:	MEM [R1+16] ← D4						
i4:	R1 ← R1 + 8						
i5:	D0 ← MEM [R1]						
i6:	D4 ← D0 * D2						
i7:	MEM [R1+16] ← D4						

Regiszter állapot tábla	
Regiszter	Érték/ROB bej.
D0	ROB#1
D2	
D4	
R1	

CDB	
Érték (V)	ROB bej.

A későbbi utasítások
D0-t ROB#1-ben keressék

Utasítástároló (RS)							
Foglalt	Egység	Műv.	T	RA	RB	EA	EB
Igen (i1)	Load	load	ROB#1	[R1]			
Nem							
Nem							

ROB és RS bejegyzések létrehozása

Tomasulo P6 példa

Ciklus: ***2***

Sorrendvisszaállító buffer (ROB)							
Utasítás		V	R	Addr	IS	EX	CM
i1:	D0 ← MEM [R1]		D0		c2		
i2:	D4 ← D0 * D2		D4				
i3:	MEM [R1+16] ← D4						
i4:	R1 ← R1 + 8						
i5:	D0 ← MEM [R1]						
i6:	D4 ← D0 * D2						
i7:	MEM [R1+16] ← D4						

Regiszter állapot tábla	
Regiszter	Érték/ROB bej.
D0	ROB#1
D2	
D4	ROB#2
R1	

CDB	
Érték (V)	ROB bej.

A későbbi utasítások
D4-et ROB#2-ben keressék

Utasítástároló (RS)							
Foglalt	Egység	Műv.	T	RA	RB	EA	EB
Igen (i1)	Load	load	ROB#1	[R1]			
Igen (i2)	FP	*	ROB#2		[D2]	ROB#1	
Nem							

ROB és RS bejegyzések létrehozása

Első forrásoperandust a ROB#1 fogja adni

Tomasulo P6 példa

Ciklus: ***3***

Sorrendvisszaállító buffer (ROB)							
Utasítás		V	R	Addr	IS	EX	CM
i1:	D0 ← MEM [R1]		D0		c2	**c3**	
i2:	D4 ← D0 * D2		D4				
i3:	MEM [R1+16] ← D4						
i4:	R1 ← R1 + 8						
i5:	D0 ← MEM [R1]						
i6:	D4 ← D0 * D2						
i7:	MEM [R1+16] ← D4						

Regiszter állapot tábla	
Regiszter	Érték/ROB bej.
D0	ROB#1
D2	
D4	ROB#2
R1	

CDB	
Érték (V)	ROB bej.

i1 EX fázisba lép, RS-ből törölve

ROB és RS bejegyzések létrehozása

Utasítástároló (RS)							
Foglalt	Egység	Műv.	T	RA	RB	EA	EB
Nem							
Igen (i2)	FP	*	ROB#2		[D2]	ROB#1	
Igen (i3)	**Store**	**store**	**ROB#3**	**[R1]**			**ROB#2**

Tomasulo P6 példa

Ciklus: 4

Sorrendvisszaállító buffer (ROB)							
Utasítás		V	R	Addr	IS	EX	CM
i1:	D0 ← MEM [R1]	[D0]	D0		c2	c3	c4
i2:	D4 ← D0 * D2		D4		c4		
i3:	MEM [R1+16] ← D4						
i4:	R1 ← R1 + 8		R1				
i5:	D0 ← MEM [R1]						
i6:	D4 ← D0 * D2						
i7:	MEM [R1+16] ← D4						

i1 eredménye a ROB-ba kerül

Regiszter állapot tábla	
Regiszter	Érték/ROB bej.
D0	ROB#1
D2	
D4	ROB#2
R1	ROB#4

CDB	
Érték (V)	ROB bej.
[D0]	ROB#1

i1 kész, eredményt kihirdeti

Utasítástároló (RS)							
Foglalt	Egység	Műv.	T	RA	RB	EA	EB
Igen (i4)	Egész ALU	+	ROB#4	[R1]			
Igen (i2)	FP	*	ROB#2	CDB.V	[D2]	~~ROB#1~~	
Igen (i3)	Store	store	ROB#3	[R1]			ROB#2

i2 ROB#1-re várt, most megjött. Ezzel végrehajthatóvá vált, IS-be lép.

Tomasulo P6 példa

Ciklus: 5

Sorrendvisszaállító buffer (ROB)							
Utasítás		V	R	Addr	IS	EX	CM
i1:	D0 ← MEM [R1]	[D0]	D0		c2	c3	c4
i2:	D4 ← D0 * D2		D4		c4	**c5**	
i3:	MEM [R1+16] ← D4						
i4:	R1 ← R1 + 8		R1		**c5**		
i5:	D0 ← MEM [R1]		D0				
i6:	D4 ← D0 * D2						
i7:	MEM [R1+16] ← D4						

Regiszter állapot tábla	
Regiszter	Érték/ROB bej.
D0	**ROB#5**
D2	
D4	ROB#2
R1	ROB#4

CDB	
Érték (V)	ROB bej.

i1 kilépett, ROB bejegyzése érvényre juttatható. DE: D0-t azóta más használja (ROB#5), ezért az eredményt nem kell beírni a regiszter tárolóba

Utasítástároló (RS)							
Foglalt	Egység	Műv.	T	RA	RB	EA	EB
Igen (i4)	Egész ALU	+	ROB#4	[R1]			
Igen (i5)	**Load**	**load**	**ROB#5**			**ROB#4**	
Igen (i3)	Store	store	ROB#3	[R1]			ROB#2

i2 EX fázisba lép, RS-ből törölve, helyére jön az i5 bejegyzése

Tomasulo P6 példa

Ciklus: 6

Sorrendvisszaállító buffer (ROB)							
Utasítás		V	R	Addr	IS	EX	CM
i1:	D0 ← MEM [R1]	[D0]	D0		c2	c3	c4
i2:	D4 ← D0 * D2		D4		c4	**c5+**	
i3:	MEM [R1+16] ← D4						
i4:	R1 ← R1 + 8		R1		c5	**c6**	
i5:	D0 ← MEM [R1]		D0				
i6:	D4 ← D0 * D2		**D4**				
i7:	MEM [R1+16] ← D4						

Regiszter állapot tábla	
Regiszter	Érték/ROB bej.
D0	ROB#5
D2	
D4	**ROB#6**
R1	ROB#4

CDB	
Érték (V)	ROB bej.

Utasítástároló (RS)							
Foglalt	Egység	Műv.	T	RA	RB	EA	EB
Igen (i6)	**FP**	*	**ROB#6**		**[D2]**	**ROB#5**	
Igen (i5)	Load	load	ROB#5			ROB#4	
Igen (i3)	Store	store	ROB#3	[R1]			ROB#2

i4 EX fázisba lép, RS-ből törölve, helyére jön az i6 bejegyzése

Tomasulo P6 példa

Ciklus: 7

Sorrendvisszaállító buffer (ROB)							
Utasítás		V	R	Addr	IS	EX	CM
i1:	D0 ← MEM [R1]	[D0]	D0		c2	c3	c4
i2:	D4 ← D0 * D2		D4		c4	**c5+**	
i3:	MEM [R1+16] ← D4						
i4:	R1 ← R1 + 8	**[R1]**	R1		c5	c6	**c7**
i5:	D0 ← MEM [R1]		D0		**c7**		
i6:	D4 ← D0 * D2		D4				
i7:	MEM [R1+16] ← D4						

Regiszter állapot tábla	
Regiszter	Érték/ROB bej.
D0	ROB#5
D2	
D4	ROB#6
R1	ROB#4

CDB	
Érték (V)	ROB bej.
[R1]	**ROB#4**

i4 kész, eredményt kihirdeti

i4 eredménye a ROB-ba kerül

Feldolgozási egymásrahatás:
nincs hely az RS-ben, nem hozható be

Utasítástároló (RS)							
Foglalt	Egység	Műv.	T	RA	RB	EA	EB
Igen (i6)	FP	*	ROB#6		[D2]	ROB#5	
Igen (i5)	**Load**	**load**	**ROB#5**	**CDB.V**		~~**ROB#4**~~	
Igen (i3)	Store	store	ROB#3	[R1]			ROB#2

i5 ROB#4-re várt, most megjött.
Ezzel végrehajthatóvá vált, IS-be lép.

Tomasulo P6 példa

Ciklus: 8

Sorrendvisszaállító buffer (ROB)							
Utasítás		V	R	Addr	IS	EX	CM
i1:	D0 ← MEM [R1]	[D0]	D0		c2	c3	c4
i2:	D4 ← D0 * D2	**[D4]**	D4		c4	c5+	**c8**
i3:	MEM [R1+16] ← D4				**c8**		
i4:	R1 ← R1 + 8	[R1]	R1		c5	c6	c7
i5:	D0 ← MEM [R1]		D0		c7	**c8**	
i6:	D4 ← D0 * D2		D4				
i7:	MEM [R1+16] ← D4						

Regiszter állapot tábla	
Regiszter	Érték/ROB bej.
D0	ROB#5
D2	
D4	ROB#6
R1	ROB#4

CDB	
Érték (V)	ROB bej.
[D4]	**ROB#2**

Utasítástároló (RS)							
Foglalt	Egység	Műv.	T	RA	RB	EA	EB
Igen (i6)	FP	*	ROB#6		[D2]	ROB#5	
Igen (i7)	**Store**	**store**	**ROB#7**	**ROB#4.V**			**ROB#6**
Igen (i3)	**Store**	**store**	**ROB#3**	**[R1]**	**CDB.V**		~~**ROB#2**~~

i5 EX fázisba lép, RS-ből törölve, helyére jön az i7 bejegyzése

R1 értéke kész, de a ROB#4-ből kell venni, mert még nincs a reg. tárolóba írva

Tomasulo P6 példa

Ciklus: 9

Sorrendvisszaállító buffer (ROB)							
Utasítás		V	R	Addr	IS	EX	CM
i1:	D0 ← MEM [R1]	[D0]	D0		c2	c3	c4
i2:	D4 ← D0 * D2	[D4]	D4		c4	c5+	c8
i3:	MEM [R1+16] ← D4				c8	**c9**	
i4:	R1 ← R1 + 8	[R1]	R1		c5	c6	c7
i5:	D0 ← MEM [R1]	**[D0]**	D0		c7	c8	**c9**
i6:	D4 ← D0 * D2		D4		**c9**		
i7:	MEM [R1+16] ← D4						

Regiszter állapot tábla	
Regiszter	Érték/ROB bej.
D0	ROB#5
D2	
D4	ROB#6
R1	ROB#4

CDB	
Érték (V)	ROB bej.
[D0]	**ROB#5**

i2 kilépett, ROB bejegyzése érvényre juttatható. DE: D4-t azóta más használja (ROB#6), ezért az eredményt nem kell beírni a regiszter tárolóba

Utasítástároló (RS)							
Foglalt	Egység	Műv.	T	RA	RB	EA	EB
Igen (i6)	**FP**	*	**ROB#6**	**CDB.V**	**[D2]**	~~**ROB#5**~~	
Igen (i7)	Store	store	ROB#7	ROB#4.V			ROB#6
Nem							

i6 ROB#5-re várt, most megjött. Ezzel végrehajthatóvá vált, IS-be lép.

i3 EX fázisba lép, RS-ből törölve

Tomasulo P6 példa

Ciklus: 10

Sorrendvisszaállító buffer (ROB)							
Utasítás		V	R	Addr	IS	EX	CM
i1:	D0 ← MEM [R1]	[D0]	D0		c2	c3	c4
i2:	D4 ← D0 * D2	[D4]	D4		c4	c5+	c8
i3:	MEM [R1+16] ← D4	**[D4]**		**&MEM[R1+16]**	c8	c9	**c10**
i4:	R1 ← R1 + 8	[R1]	R1		c5	c6	c7
i5:	D0 ← MEM [R1]	[D0]	D0		c7	c8	c9
i6:	D4 ← D0 * D2		D4		c9	**c10**	
i7:	MEM [R1+16] ← D4						

Regiszter állapot tábla	
Regiszter	Érték/ROB bej.
D0	ROB#5
D2	
D4	ROB#6
R1	ROB#4

CDB	
Érték (V)	ROB bej.

Rendelkezésre áll a Store ut. memóriacíme és az írandó adat. Ha sorra kerül, érvényre jut

Utasítástároló (RS)							
Foglalt	Egység	Műv.	T	RA	RB	EA	EB
Nem							
Igen (i7)	Store	store	ROB#7	ROB#4.V			ROB#6
Nem							

Tomasulo P6 példa

Ciklus: 11

Sorrendvisszaállító buffer (ROB)							
Utasítás		V	R	Addr	IS	EX	CM
i1:	D0 ← MEM [R1]	[D0]	D0		c2	c3	c4
i2:	D4 ← D0 * D2	[D4]	D4		c4	c5+	c8
i3:	MEM [R1+16] ← D4	[D4]		&MEM[R1+16]	c8	c9	c10
i4:	R1 ← R1 + 8	[R1]	R1		c5	c6	c7
i5:	D0 ← MEM [R1]	[D0]	D0		c7	c8	c9
i6:	D4 ← D0 * D2		D4		c9	**c10+**	
i7:	MEM [R1+16] ← D4						

Regiszter állapot tábla	
Regiszter	Érték/ROB bej.
D0	ROB#5
D2	
D4	ROB#6
R1	ROB#4

CDB	
Érték (V)	ROB bej.

i3 kilépett, ROB bejegyzése érvényre juttatható.
(A tényleges memóriaírás itt történik)

Utasítástároló (RS)							
Foglalt	Egység	Műv.	T	RA	RB	EA	EB
Nem							
Igen (i7)	Store	store	ROB#7	ROB#4.V			ROB#6
Nem							

Tomasulo P6 értékelés

- Utasításokat akkor hajtottuk végre, amikor minden operandus rendelkezésre állt

  → Lásd: ***adatáramlásos információfeldolgozási modell***!
- Egyszerű, de sokkal hatékonyabb a Scoreboard-nál
- Egy pillanat: hol volt itt a regiszterátnevezés?
  - Bújtatott formában
  - Fizikai regiszterek: ROB bejegyzések
    - Ide kerülnek az eredmények
    - Az ettől függő utasítások innen veszik a forrásoperandust
  - Regiszter leképző tábla: regiszter állapot tábla
    - Itt az összerendelés:

      Melyik architekturális regiszterhez melyik fizikai (ROB bejegyzés) tartozik

Egy eddig elfeledett rém: A memóriaműveletek RAW egymásrahatása

Memória RAW egymásrahatás

- Milyen is a „sima” RAW egymásrahatás?

```
i1: R1 ← R2 / R3
i2: R4 ← R1 + R5
```

- Nincs mit tenni, i2 meg kell, hogy várja i1-et
- Scoreboard és Tomasulo sem tudott jobbat
- Könnyen detektálható, kezelhető

- Milyen is a memória RAW egymásrahatás?

```
i1: R2 ← R1 + 4
i2: D0 ← D1 / D2
i3: MEM[R1+8] ← D0
i4: D3 ← MEM[R2+4]
```

- i3 ↔ i4 ilyen!

Memória RAW egymásrahatás

- Milyen is a memória RAW egymásrahatás?

```
i1: R2 ← R1 + 4
i2: D0 ← D1 / D2
i3: MEM[R1+8] ← D0
i4: D3 ← MEM[R2+4]
```

- Nem lehet előre detektálni!
- Egészen i3 és i4 cím számításáig nem derül ki, hogy RAW áll fent
- Miért számít?
  - RAW egymásrahatás nélkül i3 és i4 felcserélhető
    - Jó is lenne felcserélni, mert D0 csak sokára lesz kész
    - Addig is haladnánk, ha közben i4-et feldolgoznánk
  - RAW egymásrahatás esetén nem cserélhetők fel!

Memória RAW egymásrahatás

- Mit tehetünk, ha Load-ba futunk? Végrehajtsuk?
  - ROB-ban nincs előtte Store → mehet
  - ROB-ban van előtte befejezett Store
    - más címre → mehet
    - ugyanarra a címre → mehet, de nem a memóriából olvasunk, hanem a ROB-ból
  - ROB-ban van előtte befejezetlen Store, cím kiszámolva
    - más címre → mehet
    - ugyanarra a címre → meg kell várni az érték megjelenését
  - ROB-ban van előtte befejezetlen Store, cím sincs kiszámolva
    - Bátrak vagy gyávák legyünk?
    - Várjunk, attól félve, hátha RAW lesz?
    - Haladjunk, arra számítva, hogy úgysem lesz RAW?

Memória egyértelműsítés

- Belefutunk egy Load-ba
- Feldolgozás alatt van a ROB-ban előtte egy Store
- Végre merjük-e hajtani a Load-ot?

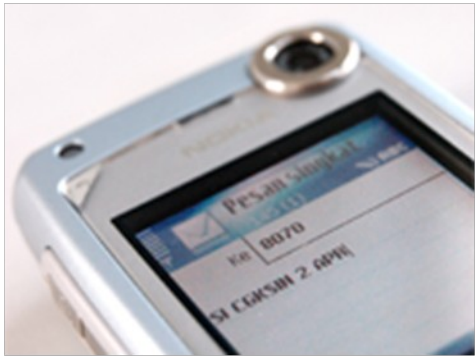
  → ***Memória egyértelműsítés (memory disambiguation)***
- Nagyon gyakori!
  - Utasítások közel harmada memóriaművelet
  - Lokalitási elvek miatt gyakran vonatkoznak egyazon címre

Memória egyértelműsítés

- Lehetséges stratégiák:
  - **Konzervatív** (gyáva):
    - Load-ok kötelesek bevárni minden előttük lévő Store-t
  - **Optimista** (vakmerő)
    - Ne vegyünk tudomást a problémáról
    - Gyorsabb, de kiderülhet, hogy rossz volt a döntés (írási és olvasási cím megegyezik)
    - Ilyenkor vissza kell pörgetni az eseményeket:
      - Load utáni utasítások invalidálása és újrakezdése a ROB-ban
  - **Spekulatív** (okostojás)
    - Optimistán kezd
    - Ha egy utasítással egyszer pórul jár, megjegyzi: vele legközelebb konzervatív lesz

Szuperskalár összefoglaló

- In-order: egyszerű
- Out-of-order: bonyolúlt
  - Adatáramlásos elven történő utasításvégrehajtás
  - Táblázatok tartják nyilván a műveleti egységek, utasítások és regiszterek állapotát
- Miért szereti a programozó az out-of-order processzort?
  - Mert nem kell kézzel optimalizálni az utasítás sorrendet (lásd: gyakorlat)
  - A CPU automatikusan összeszedi és végrehajtja, amit végre lehet



Számítógép Architektúrák

VLIW / EPIC architektúrák

Horváth Gábor
docens
BME Híradástechnikai Tanszék
ghorvath@hit.bme.hu

2012. november 23.
Budapest

Szuperskalár processzorok

- A szuperskalár processzorban több műveleti egység van
- Több utasítást is végre tud hajtani párhuzamosan
- Hogy válogatja össze a párhuzamos végrehajtásra alkalmasakat?
  - Adatáramlásos modell szerint
  - Scoreboard
  - Tomasulo
- Ez volt a ***dinamikus ütemezés:***
- Valós időben, a CPU döntött arról,
  - Hogy melyik utasítás mikor kezdhető el
  - Hogy melyik utasítás melyik műveleti egységbe kerüljön

Dinamikus ütemezés

- Mi a baj a dinamikus ütemezéssel?
  - Csak a CPU látóterében lévő utasításokból válogat
  - Komplexitás!
    - Sebességet és költséget is befolyásolja

Statikus ütemezés

- Nem tudná a fordítóprogram összeválogatni a párhuzamosan végrehajtható utasításokat?
- De.

  → ***Statikus ütemezés***
- Miért jobb, ha a fordító csinálja?
  - Nem baj, ha lassú

    → jobb heurisztikákat használhat
  - Az egész programot átlátja

    → az utasítások nagyobb környezetéből válogat

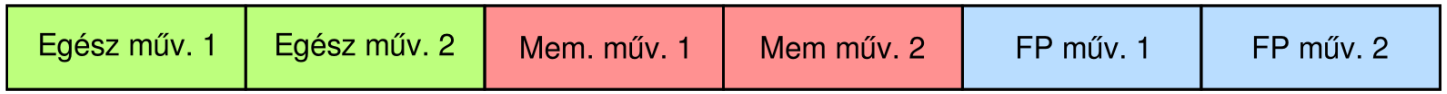
Utasítások ütemezése

- A tisztán dinamikus és tisztán statikus megoldás között vannak köztes lehetőségek is

	Párhuzamosan végrehajtható csoportok kiválasztása	Hozzárendelés a műveleti egységekhez	Végrehajtás idejének meghatározása
Szuperskalár	Hardver	Hardver	Hardver
EPIC	Fordító	Hardver	Hardver
Dinamikus VLIW	Fordító	Fordító	Hardver
VLIW	Fordító	Fordító	Fordító

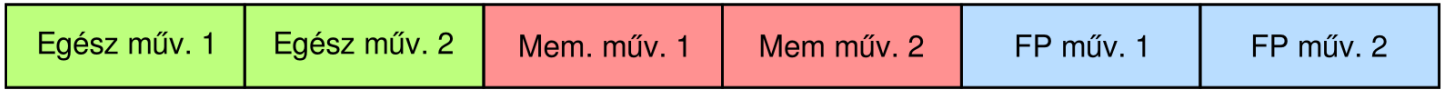
A VLIW architektúra

- VLIW = Very Long Instruction Word
- 1 pipeline, de
- Nem utasításokon dolgozik, hanem ***utasításcsoport***okon



- Ezek egy egységként haladnak a pipeline-ban
- Csoporton belüli utasításoknak nincs külön címe, az egész csoportnak közös címe van
- Csoportok szerepe:
  - Független utasítások kijelölése
  - Utasítások műveleti egységhez rendelése

A VLIW architektúra



- Nem használt pozíciókban: NOP
- Meddig tart a végrehajtása?
  - Amíg a benne lévő leglassabbé
- Döbbenet:

  A VLIW processzorok nem foglalkoznak egymásrahatásokkal!
- Mi van, ha adat-egymásrahatás van, és szünetet kellene beiktatni?
- Processzor fütyül rá
  - Vegye észre a fordító!
  - Iktasson be egy csupa NOP csoportot

A VLIW architektúra

- Példa:
- Késleltetések: egész – 1, memória – 3, lebegőpontos – 4

```
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
```

	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 VLIW architektúra

- Nem sikerült elég sok párhuzamosítható utasítást találni → rossz kihasználtság

- Tehát a fordítóprogram feladatai:
  - Utasítások csoportokba rendezése
    - A lehető legnagyobb kitöltöttségre törekedve
    - Csoport utasításai párhuzamosan végrehajthatók legyenek
  - Utasítások ütemezése
    - Egymásrahatások felismerése
    - Szükséges csupa NOP szünetek beiktatása

A VLIW architektúra

- Tipikus utasításcsoport-méretek:
  - 3-4, extrém esetben akár 28 utasítás
- Minden nehezet a fordító csinál

  → processzornak alig marad dolga!
- VLIW processzorok tipikus alkalmazása:
  - Olcsó, kis fogyasztású beágyazott rendszerek
  - Ha fontos a kiszámítható, spekuláció és predikciómentes működés: DSP (pl. TMS320C6x – 8 utasítás/csoport)
  - Grafikus processzorok: sok egyszerű feldolgozóegység kell pl. ATI a Radeon HD óta napjainkig: VLIW3 ill. VLIW4

A VLIW architektúra

- Hátrányok:
  - A program csak azon a processzoron fut, amire lefordították
    → Nem lehet a processzorcsalád új tagjába
    - Több műveleti egységet tenni
    - Gyorsabb műveleti egységeket tenni
  - Nagy probléma a transzparens cache megvalósítása!
    - Memóriaműveletek sebessége nem lesz állandó
    - Fordítóprogram nem tudja, hogyan ütemezzen
      → VLIW processzorokban nincs cache
  - Programok mérete igen nagy a sok NOP miatt

A dinamikus VLIW architektúra

- A fordító csak csoportosít
- A processzor ütemez
  - Észleli az egymásrahatásokat
  - Szüneteket tud beiktatni
- Előnyök:
  - Lehet cache-t csinálni!
  - Lehet gyorsabb műveleti egységeket bevezetni
    - Nem kell újrafordítani a programot

Az EPIC architektúra

- HP & Intel közös kutatási projektje
- Kezdés: 1994, első implementáció: 2002 (Intel Itanium)
- Cél:
  - Fordítóra nagyobb feladatot bízni a párhuzamosság felderítésében
  - A VLIW korlátai nélkül

Az EPIC architektúra

- VLIW-ben az utasítás csoportbeli helye a műveleti egységet is kijelölte:

- EPIC-ben nem. Utasításcsoport: párhuzamosan végrehajtható utasítások halmaza

Az EPIC architektúra

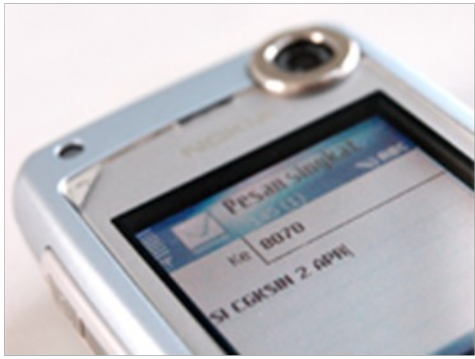
- Csoport összetétele: sablon meghatározza
- Pl. M – memória, I – egész, F – lebegőpontos, B – ugrás
  - MFI: egy memória, ehy lebegőpontos és egy egész van benne
  - MFI, MMI, MII, MIB, MMF, MFB, stb.
  - Specifikáció rögzíti a megengedett kombinációkat
  - Processzorcsalád bővítésekor ez csak bővülhet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

Az EPIC architektúra

- Lehet EPIC szuperskalár processzort készíteni:
  - Több műveleti egységgel
  - A CPU több csoport egyidejű végrehajtását végzi
- Lehet csoportokat láncolni
  - Így jelezheti a fordító, hogy nem 3, hanem 6, 9, stb. független utasítást talált
- → nem kell újrafordítani a programot, ha bővül és/vagy gyorsul a processzor

Összegzés

- Nem csak dinamikus ütemezés van a világon
  (→ out-of-order végrehajtás)
- Statikus ütemezés:
  - VLIW: fordítóprogram / programozó ütemez
  - EPIC: fordítóprogram / programozó tippeket ad
  - Egyszerűbb hardver
  - Nehéz (nagyon!) hatékony fordítóprogramot írni
  - Nehéz alacsony szinten programozni
- VLIW/EPIC processzorok akkor gyorsak, ha ki tudjuk tölteni az utasításcsoportokat



Számítógép Architektúrák

Elágazásbecslés

Horváth Gábor
docens
BME Híradástechnikai Tanszék
ghorvath@hit.bme.hu

2012. november 23.
Budapest

Procedurális egymásrahatások

- Procedurális egymásrahatás:
  - Feltételes ugrások okozzák
  - Megtörik az utasításfolyam szekvenciális viselkedését
- Mi a gond?
  - Feltételes ugrás esetén tudni kell:
    - Az elágazás ***kimenetelét*** (ugrik-e, vagy sem)
    - Ugrás esetén az ***ugrási címet*** (hová ugrik)
  - IF dolga: utasítások betöltése
    - Nincs ideje feltételkiértékelésre **és** címszámításra!
    - Egy tipphez van ideje
      - Ha bejön, annak örül
      - Ha nem:
        - » A tévedésből elkezdett utasításokat érvényteleníteni kell
        - » Legközelebb tanul a hibájából

Feltételes ugrás

- A feltételes ugró utasítások alakja:

  `JUMP R1+8 IF R2>0`

  - **`R1+8`**: ugrási cím
  - **`R2>0`**: ugrási feltétel
- Az elágazásbecslés feladata:
  - Kimenetel becslése
  - Ugrási cím becslése

Az elágazásbecslés jelentőssége

- Mit nyerünk, ha bejön a becslés?
  - Fennakadás nélkül zajlik az utasítások betöltése
- Mit vesztünk, ha rossz a becslés?
  - Minél később derül ki a rossz döntés, annál több időt fecsérel a processzor tévedésből betöltött utasításokra
  - Minél hosszabb a pipeline, annál nagyobb a veszteség!
  - Példa (Intel i7 Nehalem):
    - 4 utas szuperskalár processzor
    - A belépéstől számítva 17 ciklus múlva számolja ki a tényleges ugrási címet és értékeli ki az ugrási feltételt
    - Tfh. minden negyedik utasítás feltételes ugrás
    - Tfh. az elágazásbecslő pontossága 67%
    - Tfh. más egymásrahatás nincs

Az elágazásbecslés jelentőssége

- A példa folytatása:
  - 4 utas szuperskalár: órajelciklusonként 4 utasítás az átvitel
    → Ideális eset: átlag **0.25 órajelciklus/utasítás**
  - Utasítások negyede ugrás, minden ugrás 0.33 valószínűséggel 17 ciklusnyi extra késleltetést (kidobott időt) jelent
  - Átvitel: 0.25 + 0.25*0.33*17 = **1.65 ciklus/utasítás**
  - **6.6x** gyorsabb lenne a processzor, ha tökéletes lenne az elágazásbecslője!!!

Az elágazásbecslés jelentőssége

- A rossz döntés okozta kiesett ciklusok száma:

CPU	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

Ugrási feltétel kimenetelének becslése

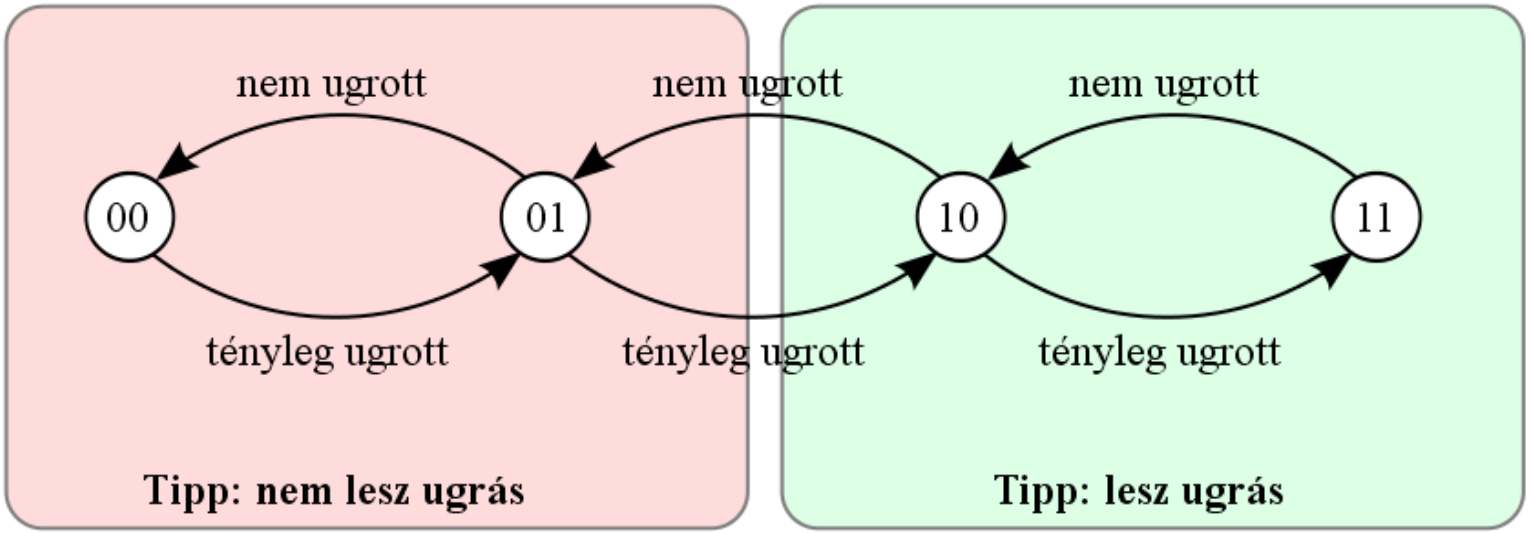
Mi is a feladat?

- Ismert (lehet): az ugró utasítás kimenetelei a múltban
- Feladat: mi lesz a következő kimenetel?
  - **1111111?**
  - **1111110110111110111111?**
  - **110011001100 11?**
  - **111111111110000000000000?**

- Elvárások:
  - Legyen a becslő *gyors*
  - Legyen a becslő *egyszerű*

Egyszerű állapotgép alapú módszer

- Minden ugró utasításhoz egy állapotgépet rendelünk
- Ez tárolja, mennyire szeret ugrani
- Ha a közepesnél jobban szeret, akkor ugrásra tippelünk
- Táblázat frissítése: a **tényleges** bekövetkezés ismeretében



Egyszerű állapotgép alapú módszer

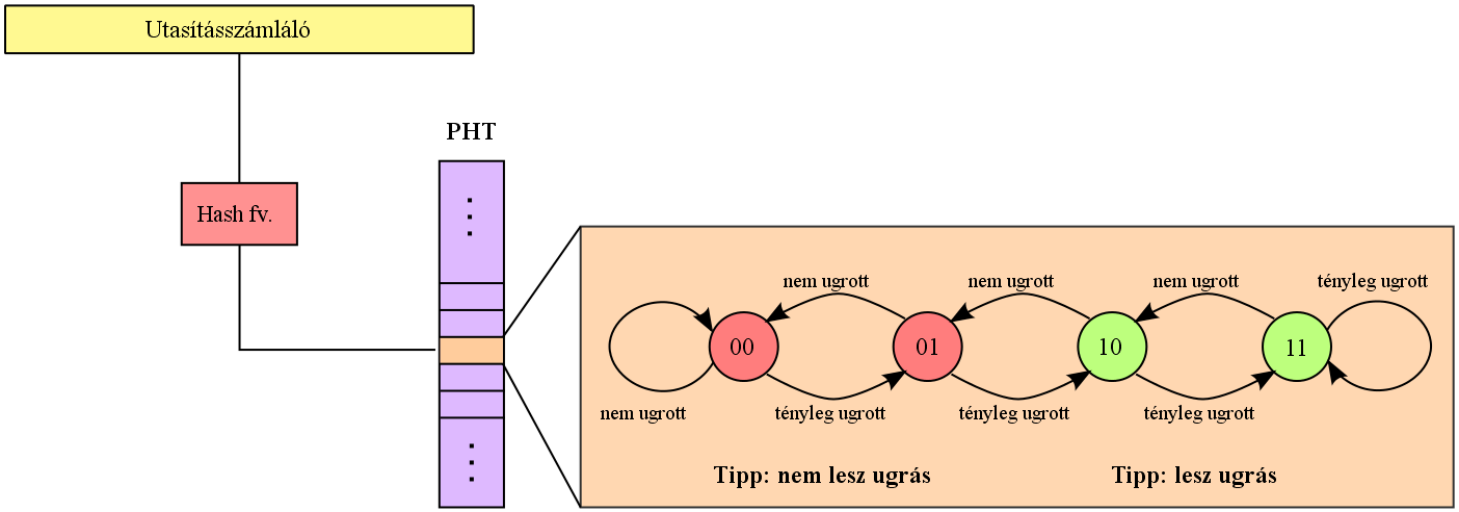
- Hatékonyságvizsgálat:

```
for (i=0; i<m; i++) {
   for (j=0; j<n; j++) {
         ...
  }
}
```

- Belső ciklus becslését vizsgáljuk: n*m döntés
- 1 bites számlálóval: 2*m rossz becslés
- 2 bites számlálóval: m rossz becslés

Egyszerű állapotgép alapú módszer

- Hol tároljuk az állapotot?
  - Valamilyen cache szerű szervezéssel
    - 32 bit-es tag, 2 bites adat → nem éri meg!
  - Utasítás cache blokkjaiban az utasítások mellett → AMD
  - Külön táblázatban: **PHT** (Pattern History Table)
    - PI: Pentium 1



Korreláció figyelembe vétele

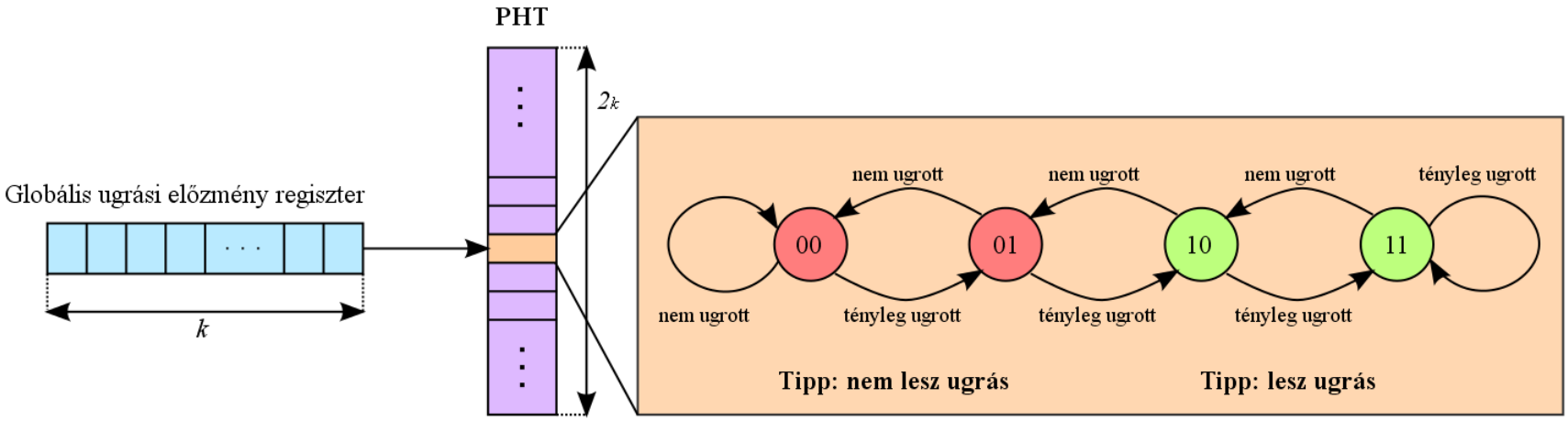
- Az ugró utasítások kimenetele gyakran függ más ugró utasítások kimenetelétől
- Példa:

```
if (a==2)
    a = 0;
if (b==2)
    b = 0;
if (a!=b) {
    ...
}
```

- Ha az első kettő igaz volt, a harmadik nem lesz igaz!
- Jó lenne kihasználni

Korreláció figyelembe vétele

- A trükk: tároljuk az egymást követő utasítások kimeneteleit egy shift regiszterben → **globális előzmény regiszter** (Global Branch History Register, GBHR)
  - Ha feltételes ugrás történik, a *tényleges* kimenetele jobbról lép be (0 vagy 1)
  - Egy k bites GBHR az utolsó k ugrás kimenetelét tárolja
  - A PHT-t ezzel indexeljük

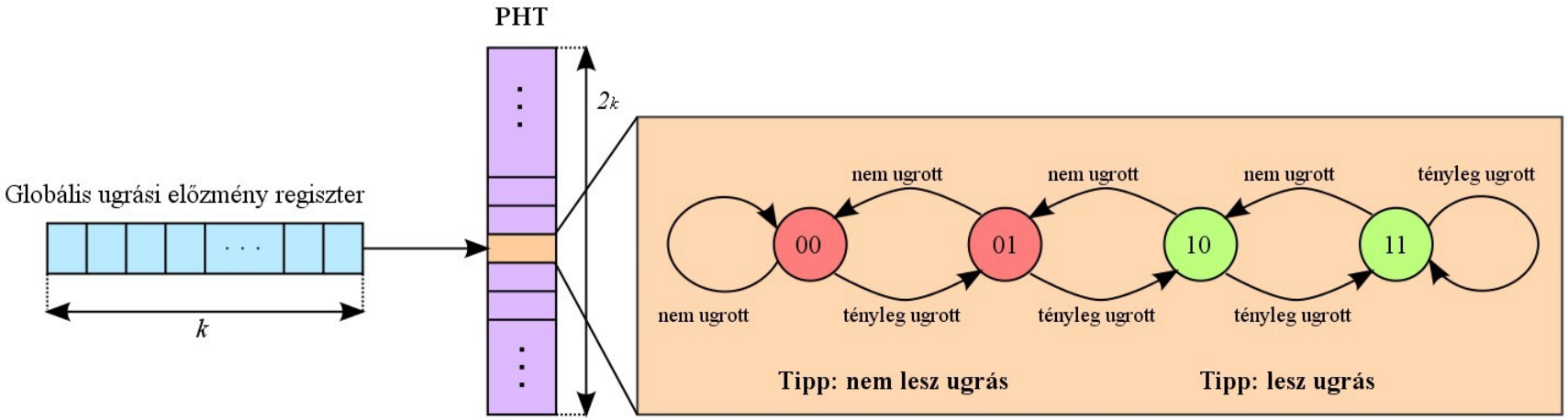


Kétszintű prediktorok

- Tanultunk egyszerű állapotgép alapú becslést
  - Utasításra lokális döntést hoznak
- Tanultunk korrelációt figyelembe vevő módszert
  - Kizárólag a globális előzmények alapján dönt
- Miért ne kombináljuk a kettőt?
- Használunk:
  - Előzmény regisztert
  - PHT-t (ugrási hajlandóságot)
- Mindkettő lehet lokális és globális is!
  - Globális ugrási előzmény: ahogy tanultuk
  - Lokális ugrási előzmény: minden utasításra külön-külön
  - Globális PHT: egyetlen rendszerszintű PHT minden utasításra
  - Lokális PHT: minden ugró utasításra külön, tárolja a várható kimeneteleket különféle körülmények között

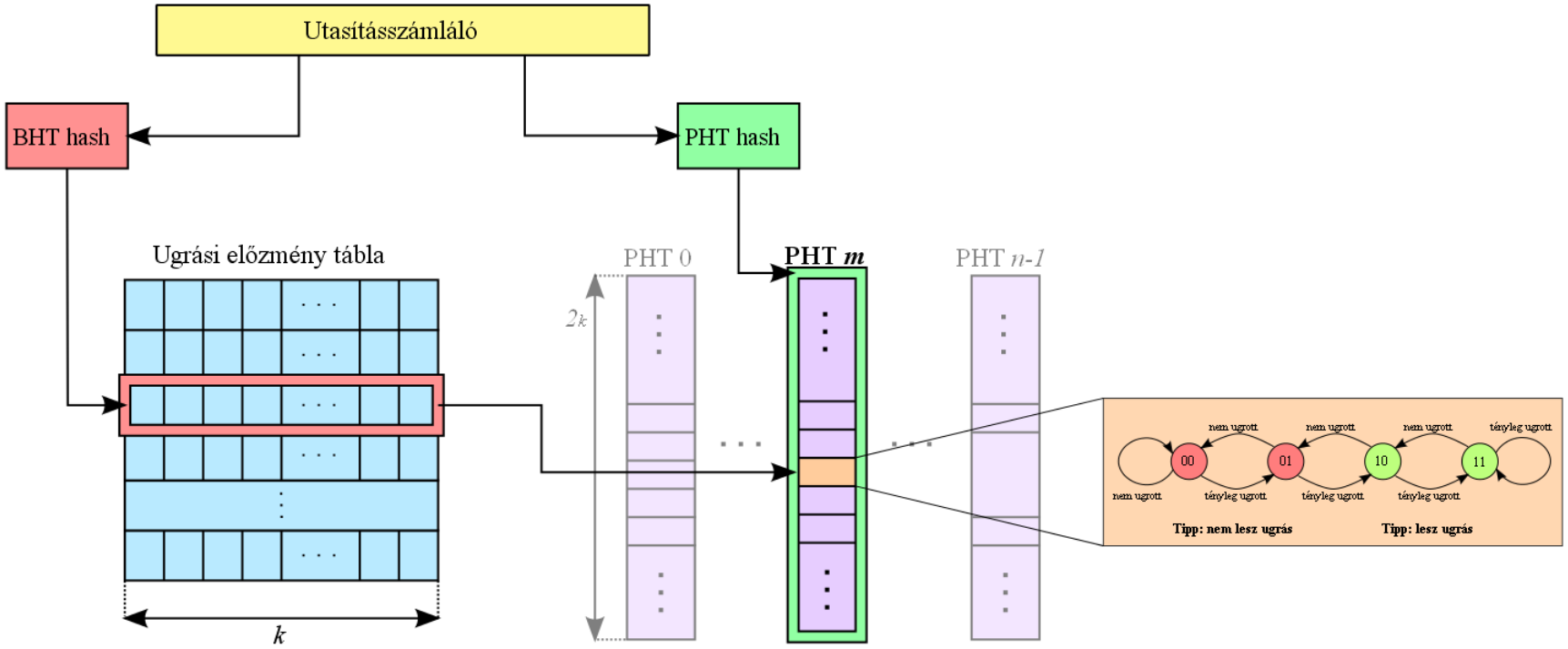
Globális BHR + globális PHT

- Egy db GBHR van, minden ugró utasítás beleshifteli a kimenetelét
- Egy db PHT van, mely tárolja, hogy az adott GBHR érték (kimenetel kombinációk) mellett mennyire esélyes az ugrás
- Már ismerjük (→korreláció figyelembe vétele)



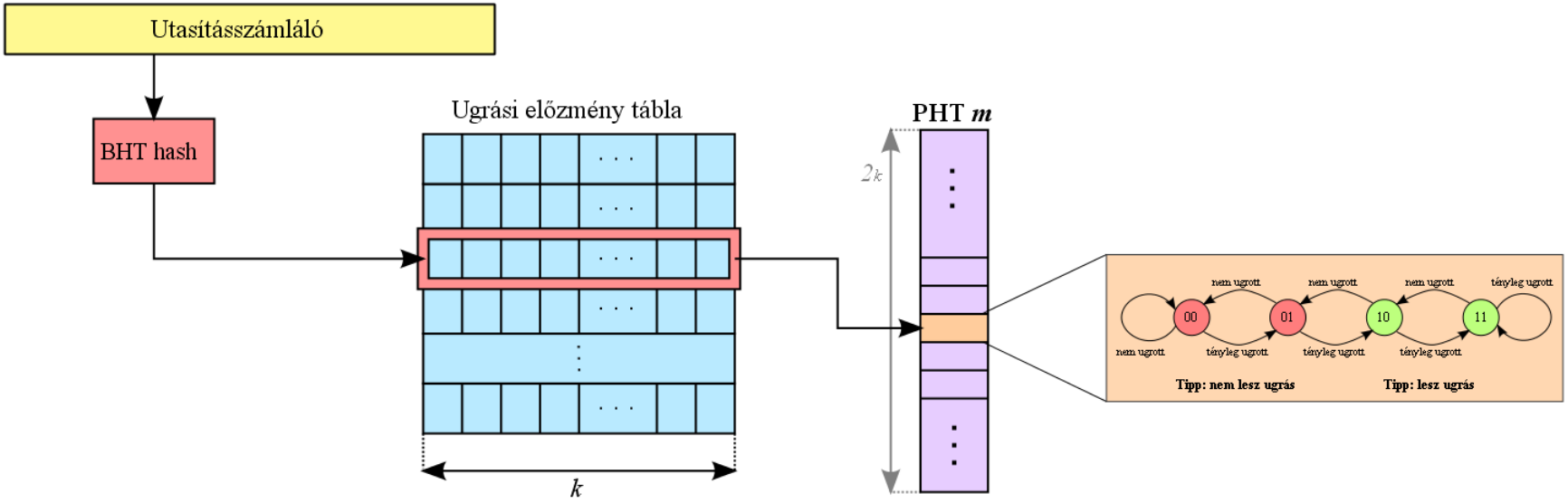
Lokális BHR (azaz BHT) + lokális PHT

- Minden utasításra külön nyilvántartjuk a múltbéli kimeneteleit
- Minden utasításra külön van egy saját PHT
  - A múltbéli kimenetel-kombinációkhoz tárolja, hogy a következő ugrás mennyire esélyes



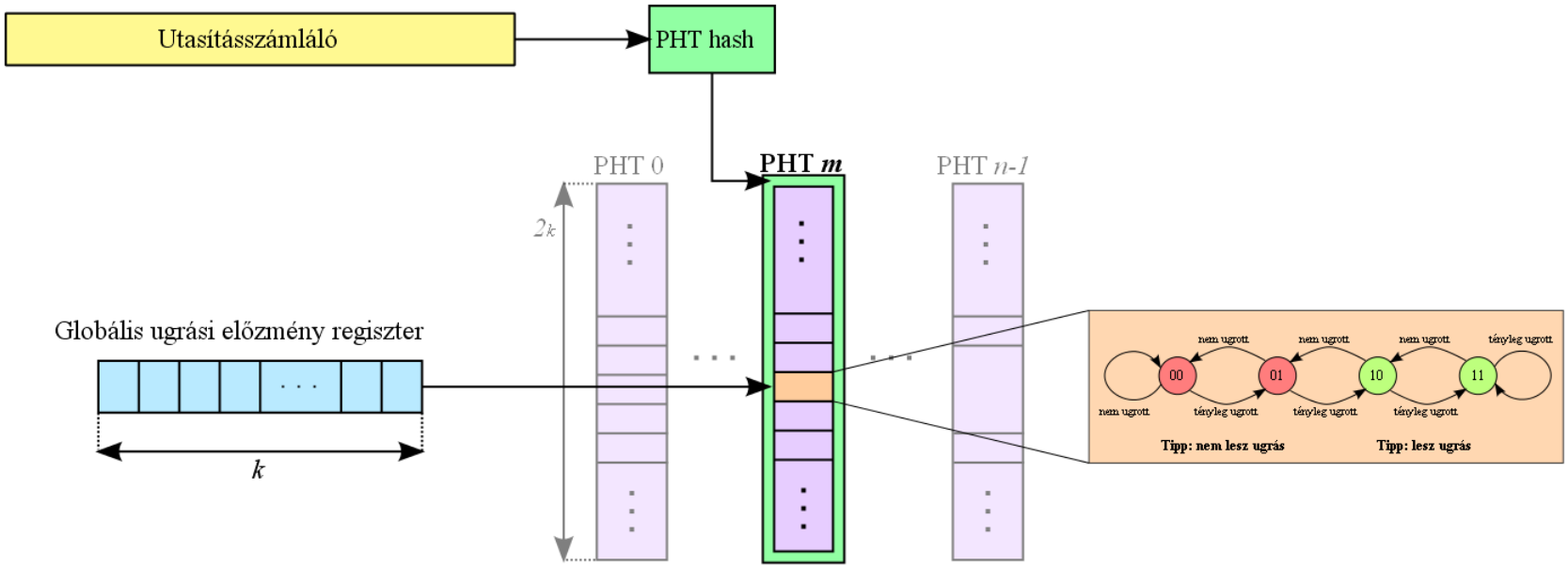
Lokális BHR (azaz BHT) + globális PHT

- Minden utasításra külön nyilvántartjuk a múltbéli kimeneteleit
- De csak egyetlen PHT van! Megadja, hogy egy adott múltbéli kimenetel-kombináció mellett milyen esélyes az ugrás



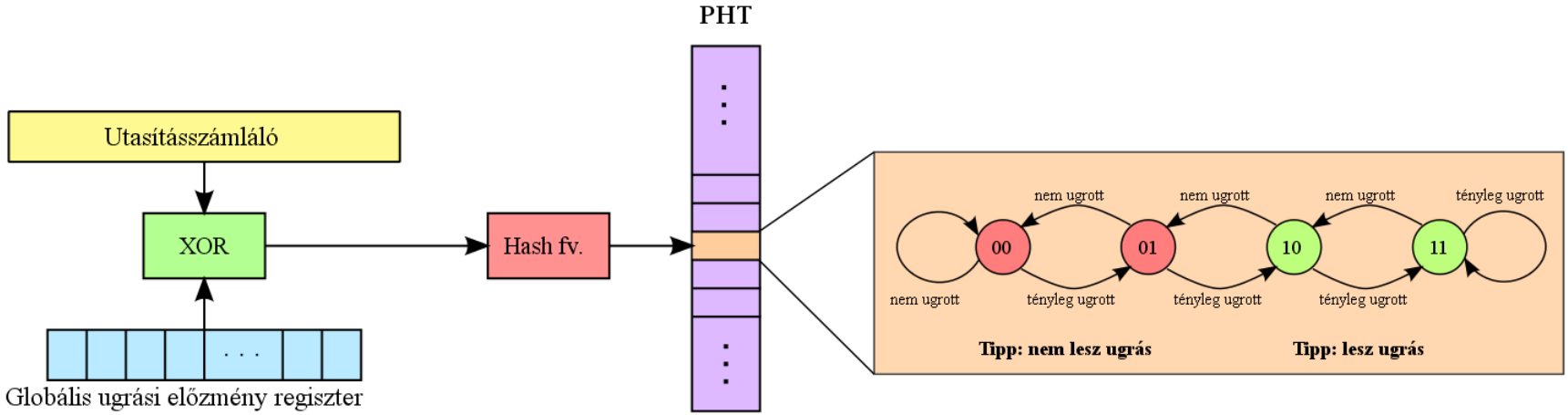
Globális BHR + lokális PHT

- Egyetlen közös előzmény regiszter van
- Minden ugró utasításnak saját PHT-ja van.
  - Az utasítás specifikus PHT-ben kell megnézni, hogy az adott globális előzmények után lesz-e ugrás, vagy sem.



A gshare prediktor

- Szintén kombinálja a lokális és globális információkat
  - Hogy döntünk lokálisan: megnézzük, hogy az adott utasításhoz mit jósol a PHT
  - Hogy döntünk globálisan: megnézzük, hogy az adott ugrási előzmények mellett mit jósol a PHT
  - Kombináljuk a kettőt, de ne bonyolítsuk, mint az előbb



- Nagyon egyszerű, és meglepően pontos!
- SPARC, POWER4, XBox 360, AMD Athlon,
  egy kicsit módosított változata az ARM Cortex A8-ban

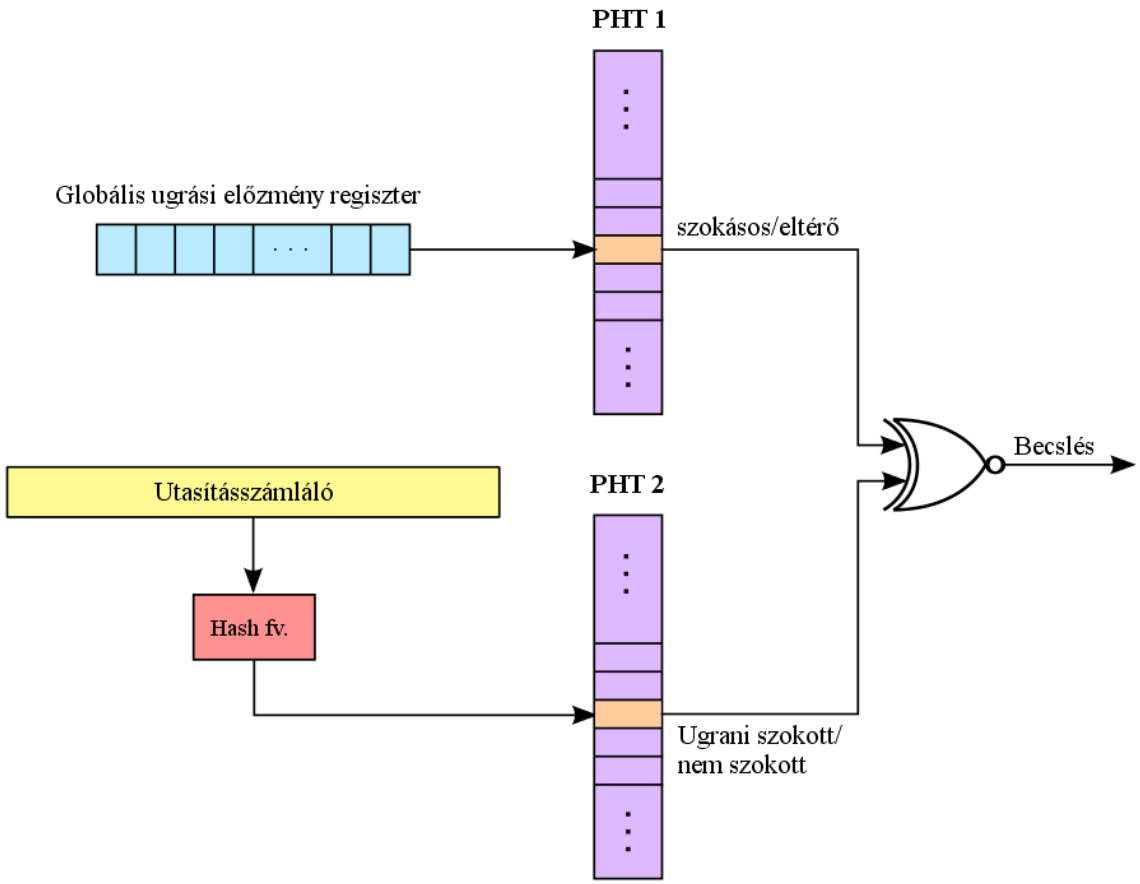
Kombinált megoldások

- Egyszer a lokális, egyszer a globális jobb
- Az előbb mutattunk módszert, ami lokális és globális információ alapján is becsül
- Újabb lehetőség:
  - Működtessünk egy prediktort tisztán lokális információra alapozva
  - Egy másokat tisztán globálisra alapozva
  - Higgyünk egyszer az egyiknek, egyszer a másiknak

    → kombinált prediktor
  - Azt, hogy melyiknek higgyünk, egy kiválasztó logika dönti el

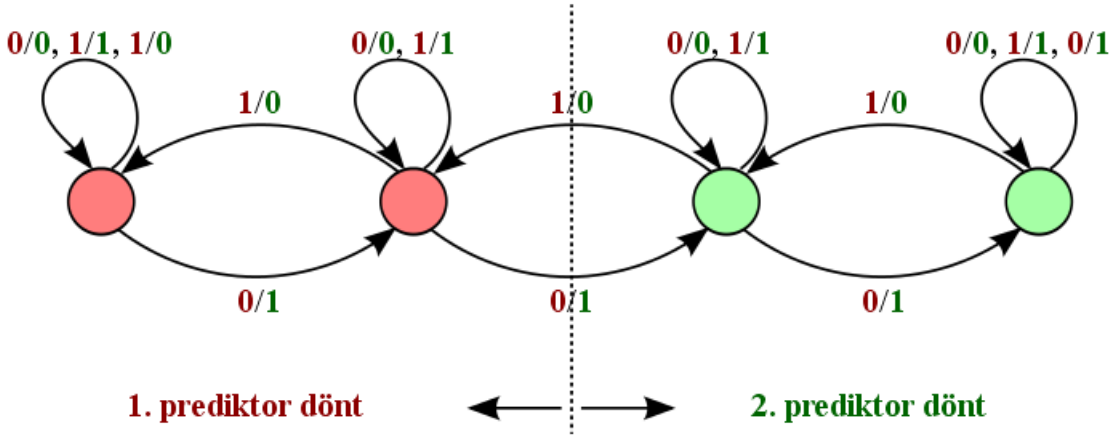
Az agree prediktor

- Lokális eljárás: merre „húz” az adott ugró utasítás
- Globális eljárás: az ugrás a tipikus irányba fog-e megvalósulni



"Tournament" prediktorok

- Két prediktor működik egyszerre, egy lokális és egy globális
- Mindig csak az egyik dönt: az, amelyik mostanában pontosabban becsült
- Ezt egy állapotgép követi, hogy mostanában melyik vált be jobban



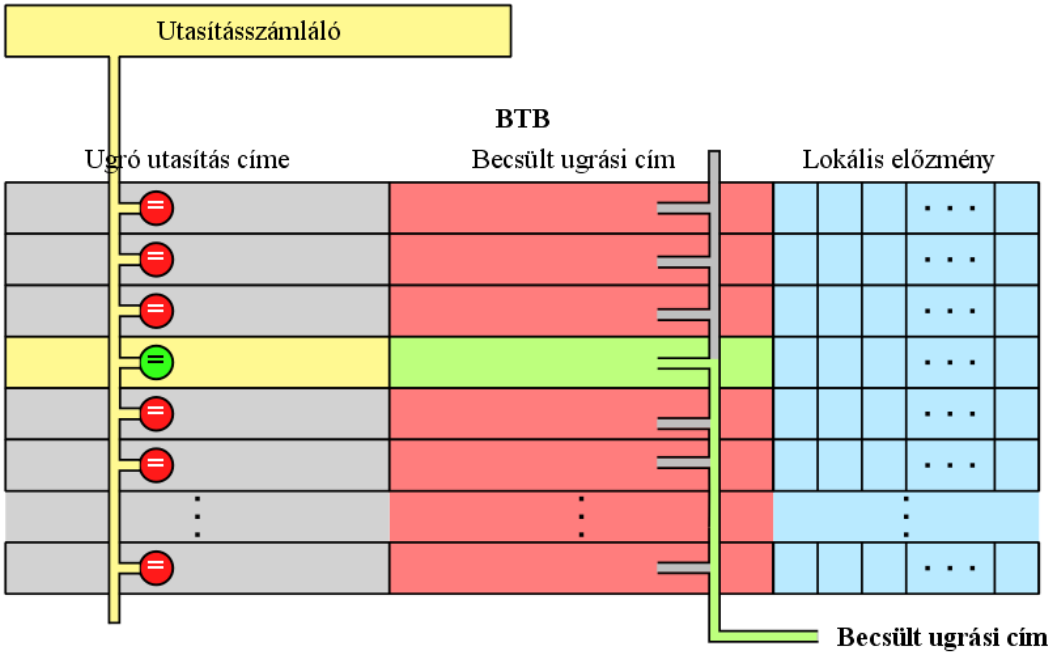
Ugrási cím becslése

Ugrási cím becslése

- Honnan kell betölteni a következő utasítást?
  - Ezt az IF-nek sürgősen tudni kell!
  - Feltételes és feltétel nélküli ugrásoknál egyaránt kérdés.
- Kiolvassuk egy speciális "***ugrási cím buffer***"-ból (BTB: Branch Target Buffer)
- Könnyebb/gyorsabb innen kiolvasni, mint kiszámolni
- A BTB mezői:
  - tag: az ugró utasítás címe
  - várható ugrási célcím
- Szervezése:
  - Valamilyen cache szervezés, pl. 4 utas asszociatív
  - Tartalommenedzsment: LRU
- Téves becsléskor korrigálják a régi értéket!

Az ugrási cím buffer

- Ha már egyszer minden egyes ugró utasításhoz van bejegyzése, mást is tárolhatunk benne:
  - Az ugrási hajlandóságot
  - Az utasítás korábbi kimeneteleit (előzmény)
  - Lokális PHT-t
- Például:

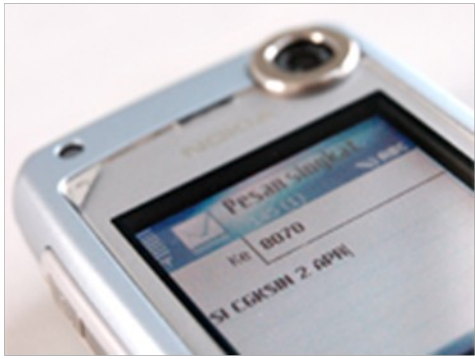


Tipikus ugrási cím buffer paraméterek

Processzor	BTB bej.-ek száma	BTB szervezése
Intel Pentium I MMX	256	4 utas asszoc.
Intel Pentium 4	4096	val. 8 utas asszoc.
Intel Core2 (ciklusokhoz)	128	2 utas asszoc.
Intel Core2 (indir. ugrásokhoz)	8192	4 utas asszoc.
Intel Core2 (egyéb ugrásokhoz)	2048	4 utas asszoc.
Intel Atom	128	4 utas asszoc.
AMD K8 és K10	utasítás cache része	
Via Nano	4096	4 utas asszoc.
ARM Cortex A8/A9	512	2 utas asszoc.

Nehézségek

- Hogy lehet kibabrálni az ugrási cím becslővel?
- Ugrabugráljunk kiszámíthatatlanul!
  - Pl. heterogén kollekció bejárása és virtuális függvények hívása c++-ban
  - Függvénypointereket tartalmazó tömb
- Lehetséges megoldás:
  - A gond az, hogy az ugrási cím buffer csak egyetlen ugrási címet tud tárolni
  - Tároljon többet! A globális előzmények függvényében dönt a becslő, hogy melyik ugrási címet dobja be tippként
  - Így működik az ARM Cortex A15



Számítógép Architektúrák

SIMD feldolgozás

Horváth Gábor
docens
BME Híradástechnikai Tanszék
ghorvath@hit.bme.hu

2012. december 3.
Budapest

Hatékony utasításfeldolgozás

- Hogyan tettük eddig hatékonnyá az utasítás-végrehajtást?
  - ***Utasítás szintű párhuzamosítással***
    - Egyszerű pipeline: átlapoltuk több utasítás végrehajtását
    - Szuperskalár: a CPU több, általa párhuzamosíthatónak talált utasítást hajt végre egyszerre
    - VLIW/EPIC: a CPU több, a fordító / programozó által párhuzamosíthatónak talált utasítást hajt végre egyszerre
- Megvannak a korlátai:
  - Minél több utasítás áll feldolgozás alatt egyszerre, annál nagyobb a sansz az egymásrahatásra
  - Sok műveleti egység → sok forwarding út kell
  - Ciklusidő nem lehet tetszőlegesen rövid
  - Időnként becslés / spekuláció szükséges → ha rosszul sikerül, kárba ment egy csomó idő

Hatékony utasításfeldolgozás

- Ennyi korlát mellett miért szeretjük az utasításszintű párhuzamosítást?
- Mert egyszerű programozni!
  - A programozó
    - Fejében folyamatábra
    - Szekvenciális algoritmust ír
    - Skalár változókon
  - A párhuzamosítási lehetőségeket a CPU és/vagy a fordító fedi fel
    - Módja: függőségi analízis, adatáramlásos elv
    - El van rejtve a programozó elől

Flynn taxonómia

- Flynn felosztása az utasítások és adatok viszonya szerint:
- SISD (single instruction, single data):
  - Egy utasítássorozat végrehajtása skalár adatokon
  - Ezt tanultuk eddig
- SIMD (single instruction, multiple data)
  - Egy utasítássorozat több adaton végez műveletet egyszerre
  - Vektorprocesszorok/tömbprocesszorok, stb.
  - Ezt tanuljuk meg ma
- MIMD (multiple instruction, multiple data)
  - Több utasítássorozat több adaton dolgozik
  - Multiprocesszoros rendszerek
  - Ezt tanuljuk meg hamarosan
- MISD (multiple instruction, single data)
  - Hibatűrő rendszerekben

Vektorprocesszorok

Vektorprocesszorok

- Klasszikus skalár adattípus és műveletek mellett / helyett
  - Vektor adattípus
  - Vektorkezelő utasítások
- Minden modern szuperszámítógép ilyen

Példa

- C program:

```
for (i=0; i<64; i++)
    C[i] = A[i] + B[i];
```

- Klasszikus (skalár) megoldás:

```
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
```

Vektoros megoldás:

```
VLR ← 64
V1 ← MEM[R1]
V2 ← MEM[R2]
V3 ← V1 + V2
MEM[R3] ← V3
```

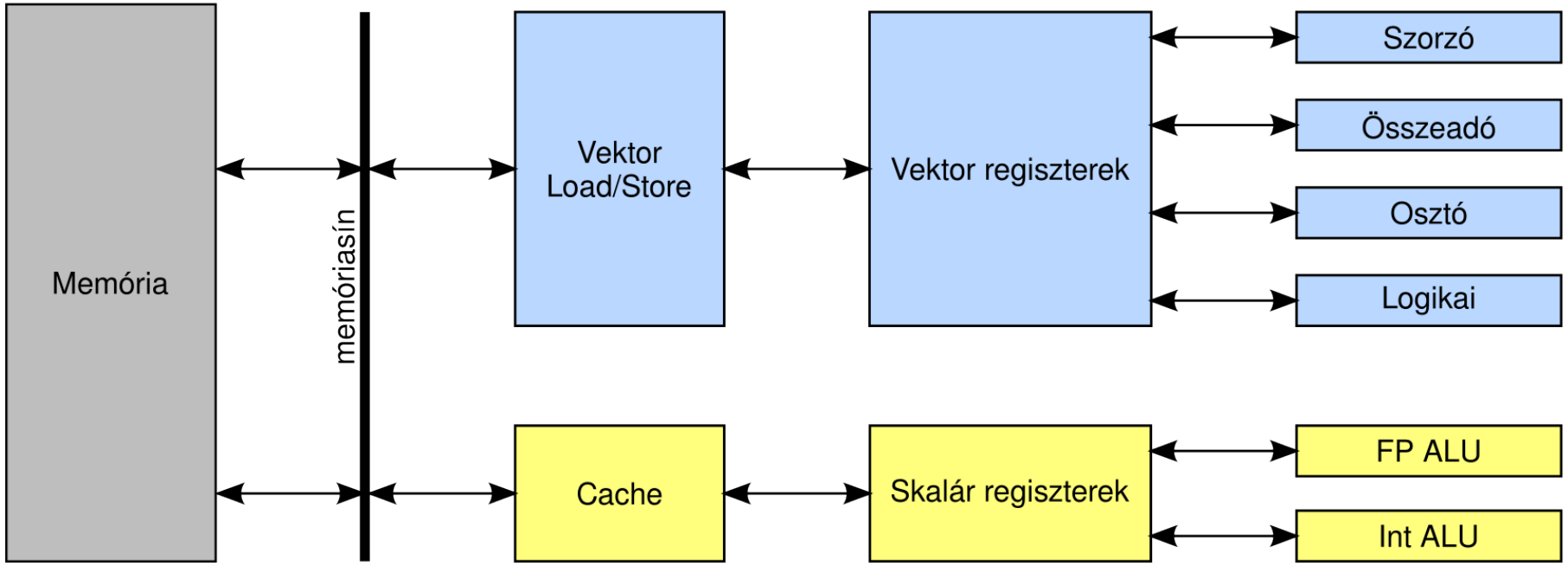
Példa

- Miért is jobb vektorprocesszorral?
  - Rövidebb, tömörebb kód
  - ***Nincs szükség ciklusra!***
    - Mi a baj a ciklussal?
      - Minden körben újra és újra
        - » Le kell hívni
        - » Dekódolni
        - » Végrehajtani

        a ciklusmag utasításait.
      - Minden körben procedurális egymásrahatás
        - » 64-szer kell elágazásbecslést végezni!
  - ***A vektorműveletek implicit feltételezik, hogy a vektorelemek függetlenek***
    - Sok műveleti egységgel vagy/és igen mély pipeline-al nagy teljesítmény érhető el

Vektorprocesszorok felépítése

- Típusok:
  - Regiszter-regiszter
  - Memória-memória
- Mostantól csak regiszter-regiszter:



Vektorprocesszorok működése

- Miért csak a skalár operandusoknak jár cache?
- Vektorok-műveleteken máshogy gyorsítunk:

  → Spéci memóriakezeléssel
- Miért lassú egy skalár memóriaolvasás?
  - Cím kiadás és adat megérkezése között több órajelciklus telik el
- Miért szerencsések a vektor memóriaműveletek?
  - Vektor beolvasás: fix (nagy) számú memóriaművelet
  - Kiadom az egyik elem címét
  - Következő ciklusban még nem jön meg, de attól még kiadhatom egy másik elem címét
  - Stb. → pipeline-szerű megoldás
  - Ehhez kell, hogy

    az egymás utáni elemek külön memória-bankban legyenek

Fésüs címzés

Órajel	Bank					
	0	1	2	3	4	5
1		15636				
2		Fogl.	15640			
3		Fogl.	Fogl.	15644		
4		Fogl.	Fogl.	Fogl.	15648	
5		adat[0]	Fogl.	Fogl.	Fogl.	15652
6	15656		adat[1]	Fogl.	Fogl.	Fogl.
7	Fogl.	15660		adat[2]	Fogl.	Fogl.
8	Fogl.	Fogl.	15664		adat[2]	Fogl.
9	Fogl.	Fogl.	Fogl.	15668		adat[3]
10	adat[4]	Fogl.	Fogl.	Fogl.	15672	

Műveleti egységek

- Vektorműveletek gyorsítása:
  - ***Műveleti egységek többszörözésével***
    - Vektorelemek függetlenek
    - pl. 4 műveleti egység → egyszerre 4 elem számolása
  - ***Mély adat-pipeline alkalmazásával***
- Adatpipeline? Az mi?
- Lebegőpontos számok: Szám = $(-1)^s * c * b^q$
- Példa: lebegőpontos összeadás (4 fázis):
  - Ellenőrizzük, valamely operandus nulla-e
  - A 2 operandus azonos karakterisztikára hozása
  - Összeadás elvégzése
  - Eredmény normál alakra hozása

Adat-pipeline

- Példa: lebegőpontos szorzás (5 fázis)
  - Ellenőrizzük, valamely operandus nulla-e
  - Karakterisztikákat összeadjuk
  - Mantisszákat összeszorozzuk
  - Eredmény előjel bitjét előállítjuk
  - Eredményt normál alakra hozzuk
- Ha egy vektorelemre az első fázis kész, rögtön kezdi a következő elemre is az első fázist → futószalag

Adat-pipeline

	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

Adat-pipeline

- 2 műveleti egységgel:

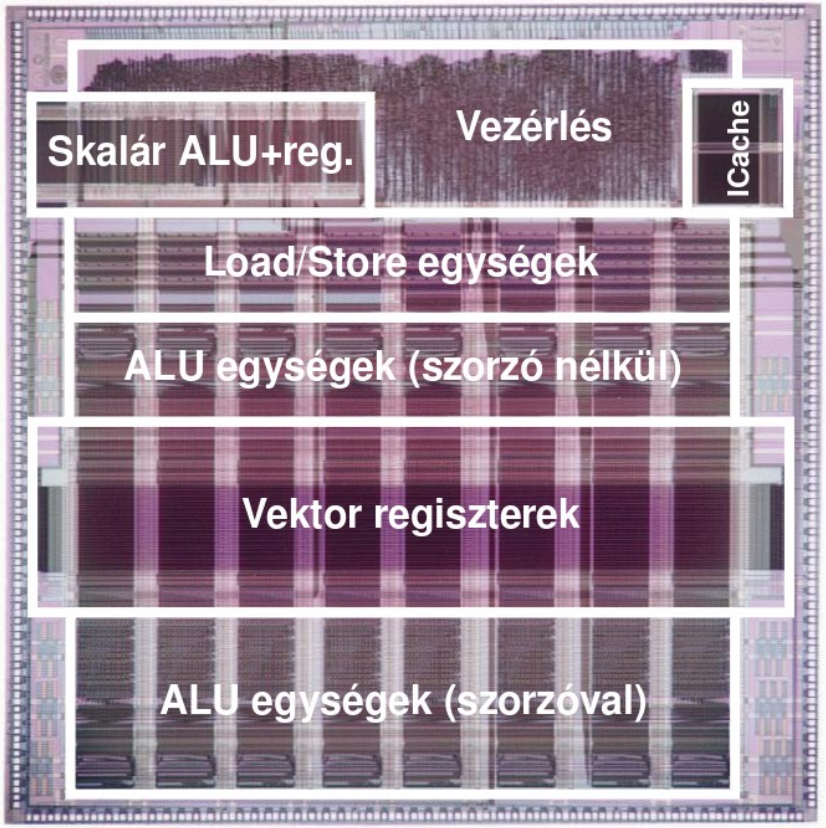
	1	2	3	4	5	6	7	8	9	10	11
V2[0] ← V0[0]+V1[0]	A0	A1	A2	A3							
V2[1] ← V0[1]+V1[1]	A0	A1	A2	A3							
V2[2] ← V0[2]+V1[2]		A0	A1	A2	A3						
V2[3] ← V0[3]+V1[3]		A0	A1	A2	A3						
V2[4] ← V0[4]+V1[4]			A0	A1	A2	A3					
V2[5] ← V0[5]+V1[5]			A0	A1	A2	A3					
V2[6] ← V0[6]+V1[6]				A0	A1	A2	A3				
V2[7] ← V0[7]+V1[7]				A0	A1	A2	A3				

Adat-pipeline

- Nagyon fontos:
  - Ez egy olyan pipeline, melyben
    - Nincs egymásrahatás!
    - Nem kell egymásrahatást vizsgáló és kezelő logika
    - Emiatt tetszőleges mély lehet
    - Emiatt tetszőleges széles lehet
  - Egyetlen korlátozó tényező:
    - Hány részfázisra tudunk bontani egy aritmetikai műveletet

Példa 1.

- Berkely T0 (Torrent-0, 1995)
- 32 hosszú regiszterek
- 1 sáv: 4 elem kezelése (ábrán 8 függőleges sáv)
- Minden sávban 2 ALU
- … de csak az egyik tud szorozni

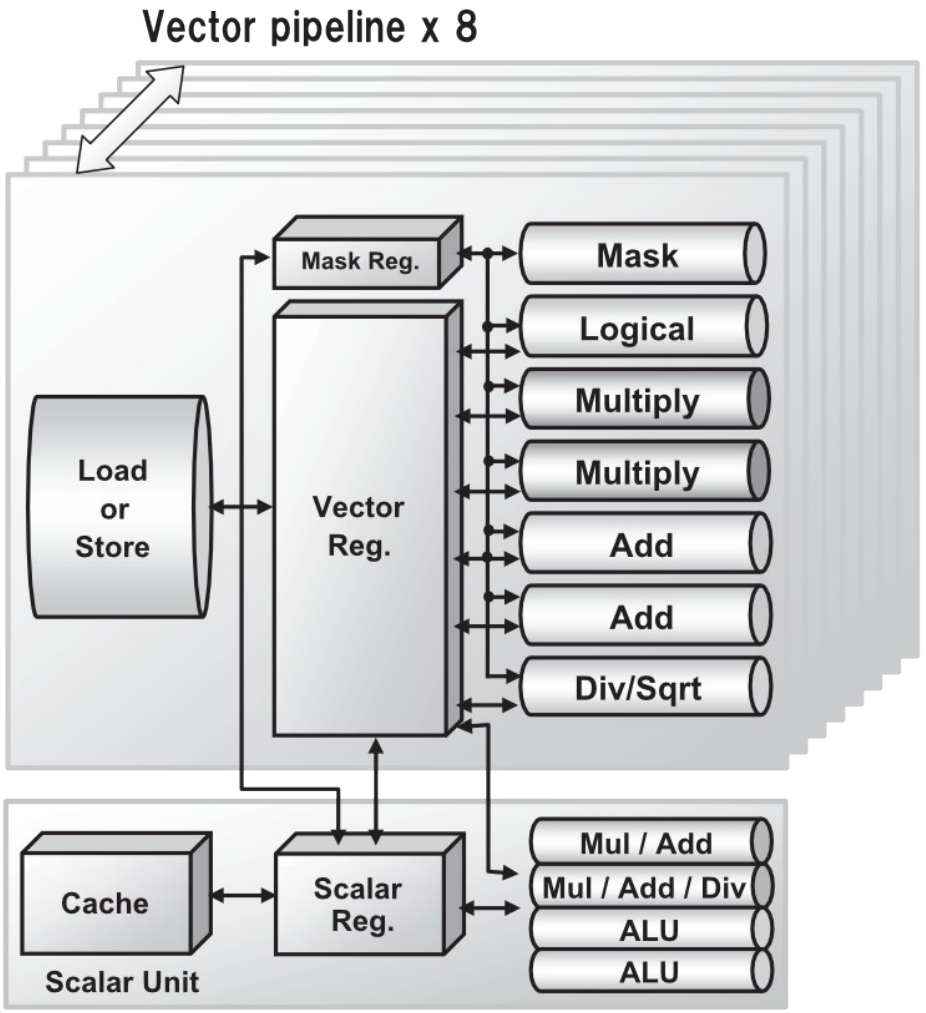


Példa 2.

- NEC SX-9 szuperszámítógép processzorra
- 2008-as év leggyorsabb szuperszámítógépe
- Német meteorológiai központ ezzel dolgozik (2 db)
- 2011-ben: 224 db vektorprocesszor + 7 TB memória
- 1 processzor:
  - 256 hosszú vektorokat kezel
  - 8 pipeline szervezésű műveleti egysége van
- Vektor ALU: 3.2 GHz
- Skalár egység (4 utas out-of-order): 1.6 GHz

Példa 2.

- A NEC SX-9 felépítése:



Vektorprocesszorok jellegzetes megoldásai

Strip-mining

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

  → ez a ***Strip-mining***

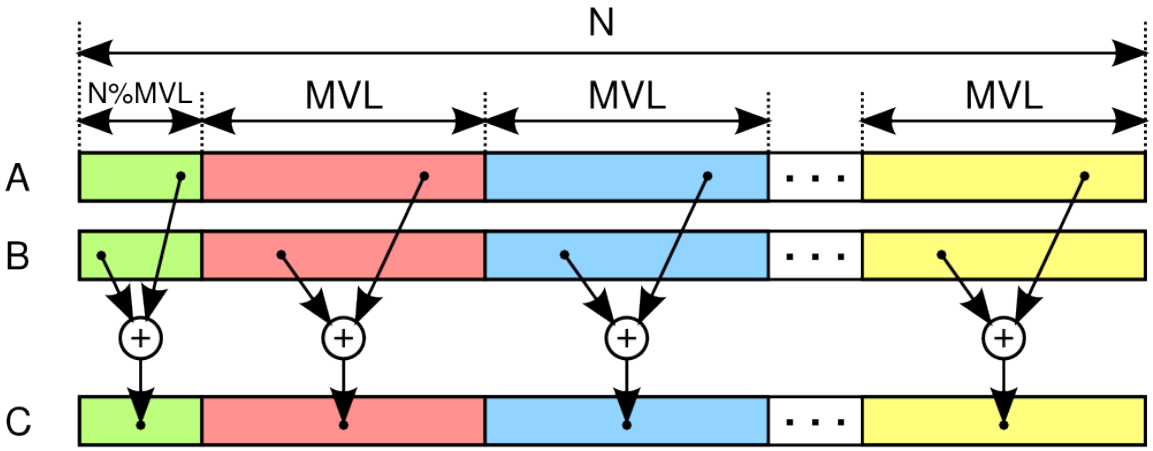
Strip-mining példa

- C kód:

```
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
```



Feltételes vektorműveletek

- Maszk regiszter:
  - Ahol =0, arra az elemre a köv. vektorművelet nem vonatkozik
- 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
```

- Kétféleképpen valósítható meg:
  - Naív: Vektor ALU mindig minden elemre számol, de a maszkolt elemek eredményét nem tárolja
  - Hatékony: A Load/Store és az ALU a maszkolt elemeket kihagyja → gyorsabb végrehajtás

Művelet láncolás

- RAW egymásrahatások vektorprocesszorban is vannak:
- ```
  V1 ← MEM[R1]
  V3 ← V1 + V2
  V5 ← V3 * V4
  ```

- A forwarding vektorprocesszoros megfelelője:
  - ***Művelet láncolás*** (vector chaining)
  - Függő utasításnak nem kell megvárnia az előző utasítás végét
  - Ahogy V1 első eleme megvan, V3 első eleme el is kezdhető
  - Amint V3 első eleme megvan, V5 első eleme el is kezdhető
  ```

Műveletláncolás példa

Utasítás	1	2	3	4	5	6	7	8	9
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			

...

Műveletláncolás példa

Utasítás	1	2	3	4	5	6	7	8	9
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

...

SIMD utasításkészlet-kiegészítések

SIMD utasításkészlet kiegészítések

- Vektor utasítások otthoni használatra is hasznosak
  - Képfeldolgozási célokra
  - 3D grafikai alkalmazásokban / játékokban
  - Egyszerű tudományos alapfeladatok is jól vektorizálhatók
- Számos processzor támogat vektorműveleteket
- De attól még nem lesznek vektorprocesszorok!
  - Nagyon kicsi a vektorméret (legjobb: 256 bites)
  - Nincs VLR
  - Nincs maszk regiszter
  - Nincs művelet láncolás
  - Nincs adat-pipeline
    - Annyi műveleti egység van, amekkora a vektorméret

SIMD utasításkészlet kiegészítések

- Utasítások fajtái:
  - Vektor-vektor műveletek:
    - Inter-vektor: 2 vektor között. Eredmény: vektor
      - pl. 2 vektor összeadása
    - Intra-vektor: 1 vektor elemein. Eredmény: skalár
      - pl. vektor elemeinek összeadása
    - Vektorelemek átrendezése (shuffling)
  - Skalár-vektor műveletek:
    - pl. minden elem megszorzása egy skalárral

SIMD utasításkészlet kiegészítések

Vektor kieg.	Utasításkészlet	Vekt.reg. száma	...hossza	Elemek típusa
MMX	x86	8	64 bit	Int: 8x8, 4x16, 2x32 bit
3DNow	x86	8	64 bit	Float: 2x32 bit
SSE	x86/x64	8	128 bit	Float: 4x32 bit
SSE2-4	x86/x64	8/16	128 bit	Int: 16x8, 8x16, 4x32 bit. Float: 4x32, 2x64 bit
AVX	x86/x64	16	256 bit	Float: 8x32, 4x64 bit
AltiVec	Power	32	128 bit	Int: 16x8, 8x16, 4x32 bit Float: 4x32 bit
NEON	ARM	32	64 bit	Int: 8x8, 4x16, 2x32 bit Float: 2x32 bit

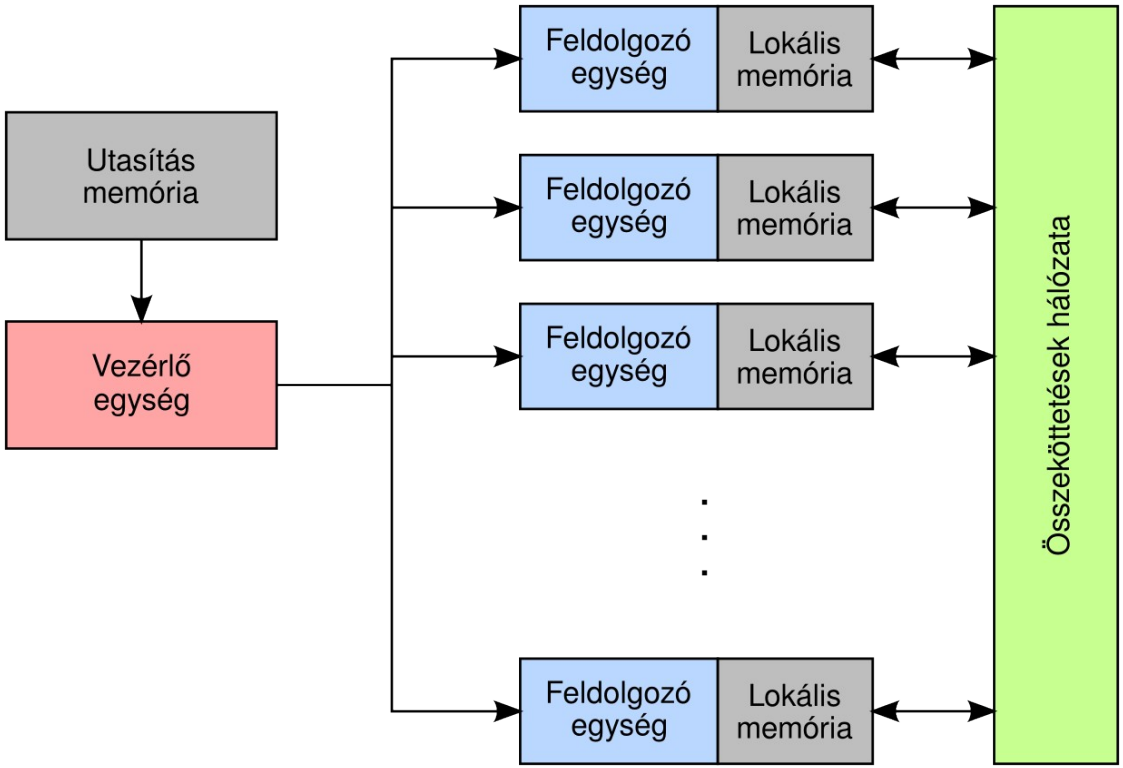
Tömbprocesszorok

Klasszikus tömbprocesszorok

- SIMD egy másik formája
- Elemei:
  - 1 db vezérlőegység
  - Sok-sok feldolgozóegység
    - Lokális memóriával
    - Többiek felé összeköttetéssel
- Működése:
  - Vezérlőegység minden ciklusban kiad egy utasítást
  - Üzenetszórással, megkapja minden feldolgozóegység
  - Minden feldolgozóegység végrehajtja a saját adatain

    (maszkregiszter van, aki nincs engedélyezve, az nem csinál semmit)

Klasszikus tömbprocesszorok



- Más és más feladat más és más összeköttetéseket igényel
  → Célszerű konfigurálhatóvá tenni

Klasszikus tömbprocesszorok

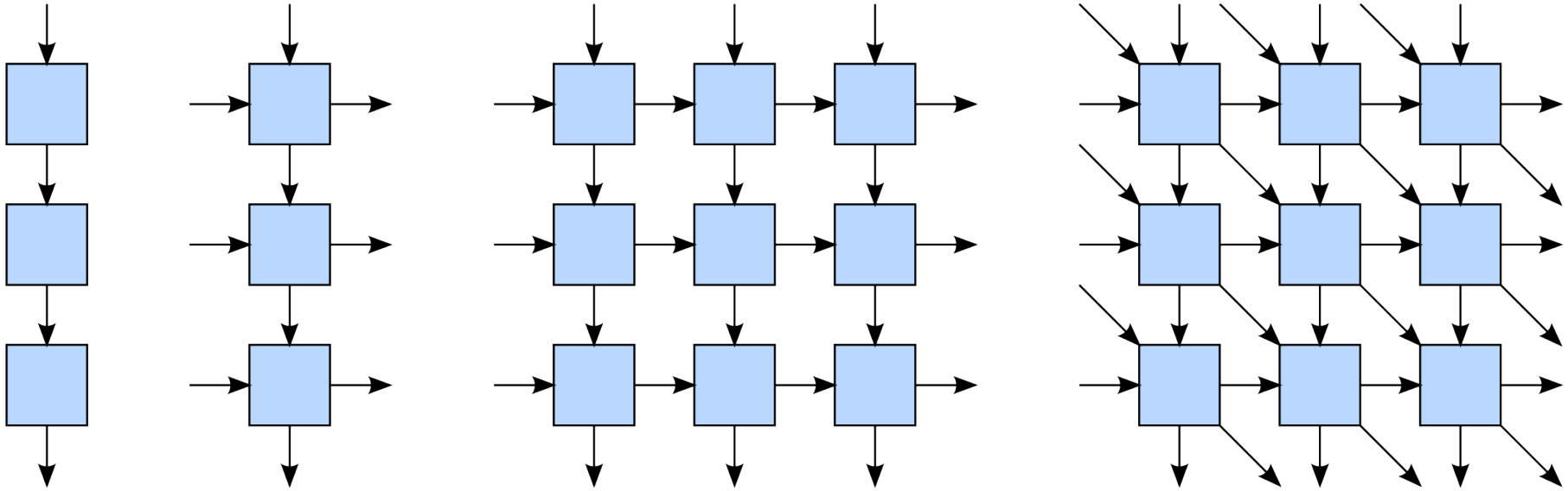
- Hatékonysági problémák, ha sok a feldolgozóegység:
  - Üzenetszórással mennek ki az utasítások:
    - Sokáig tart, mire mindenkihez elér
    - Hosszú vezetékek: fogyasztás, melegedés
  - Távoli feldolgozóegységek adatcseréje:
    - Sokáig tart, mire célba ér
    - Hosszú vezetékek: fogyasztás, melegedés
- Nagy teljesítményű, sok feldolgozóegységgel rendelkező tömbprocesszorhoz ***speciális struktúra*** kell!

Szisztolikus tömbprocesszorok

- Olyan speciális tömbprocesszorok, melyekben
  - Minden feldolgozóegység csak a közvetlen szomszédokkal van összekötve
  - Feldolgozóegységek közötti kommunikációt közös órajel hajtja
  - Nincs központi vezérlőegység: minden ciklusban mindenki ugyanazt a műveletet hajtja végre
  - A tömb csak a széleken kapcsolódik a külvilághoz
- Működés:
  - Minden órajelciklusban mindenki
    - Elveszi a szomszédoktól a bemenetét
    - Végrehajtja a műveletet (mindenki ugyanazt)
    - Továbbítja a szomszédainak az eredményt

Szisztolikus tömbprocesszorok

- Tipikus elrendezések:



Szisztolikus tömbprocesszorok

- Kedvező tulajdonságok:
  - Rendkívül rövid jelutak
    - Kis fogyasztás
    - Magas órajel
  - Gyártástechnológiai előnyök
    - Ismétlődő, szabályos mintázatot hatékonyan lehet gyártani
  - Könnyen bővíthető: csak a széleihez kell tenni pár új feldolgozóegységet
- Kedvezőtlen tulajdonságok:
  - Globális órajelhálózat
  - Nagy rendszerekben egyesekhez hamarabb ér az órajel, mint másokhoz → kavarodást okozhat

Hullámfront tömbprocesszorok

- Hullámfront tömbprocesszorok:
  Mint a szisztolikus, de:
  - Teljesen aszinkron
  - Nincs órajel
  - Adatfolyam alapú:
    - Egy feldolgozóegység akkor végzi el a műveletet, ha minden input megjött
  - Szomszédok közötti adatcsere összetettebb protokollt igényel
  - Cserébe megússzuk az órajel-hálózatot

    → még nagyobb sebesség érhető el

Tömbprocesszorok tervezése

Tömbprocesszorok tervezése

- Többé-kevésbé mechanikus eljárás
- Példa: vektor-mátrix szorzás

$$c = A \cdot b \rightarrow c_i = \sum_{j=1}^{N} A_{i,j} b_j$$

- C kód:

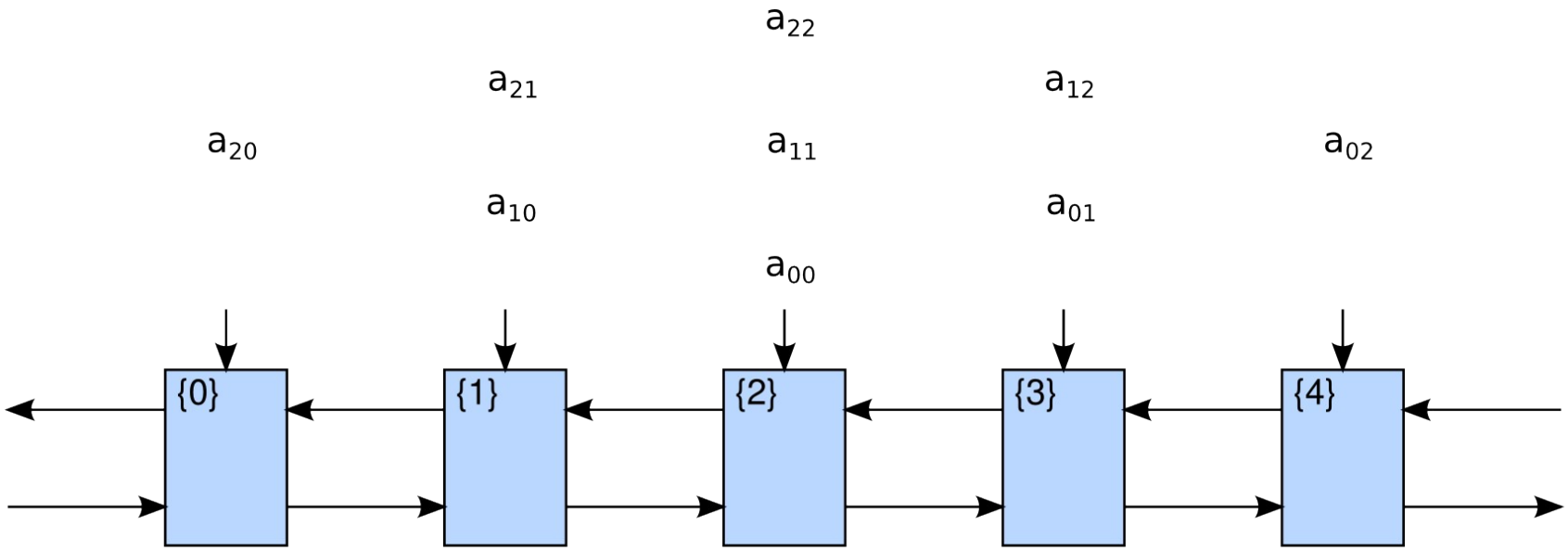
```
for (int i=0; i<M; i++) {
    C[i] = 0;
    for (int j=0; j<N; j++)
        C[i] = C[i] + A[i][j] * B[j];
}
```

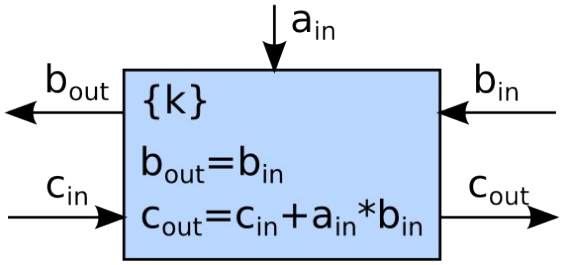
- „Hagyományos” processzoron N*M lépés

Tömbprocesszorok tervezése

- Tervezési lépések:
  1) Teljes függőségi gráf felrajzolása
  2) Lokalizált függőségi gráf felrajzolása
  3) Projekció szisztolikus tömbprocesszorra
  4) Ütemezés megadása

Projekció diagonál irányban, ütemezés

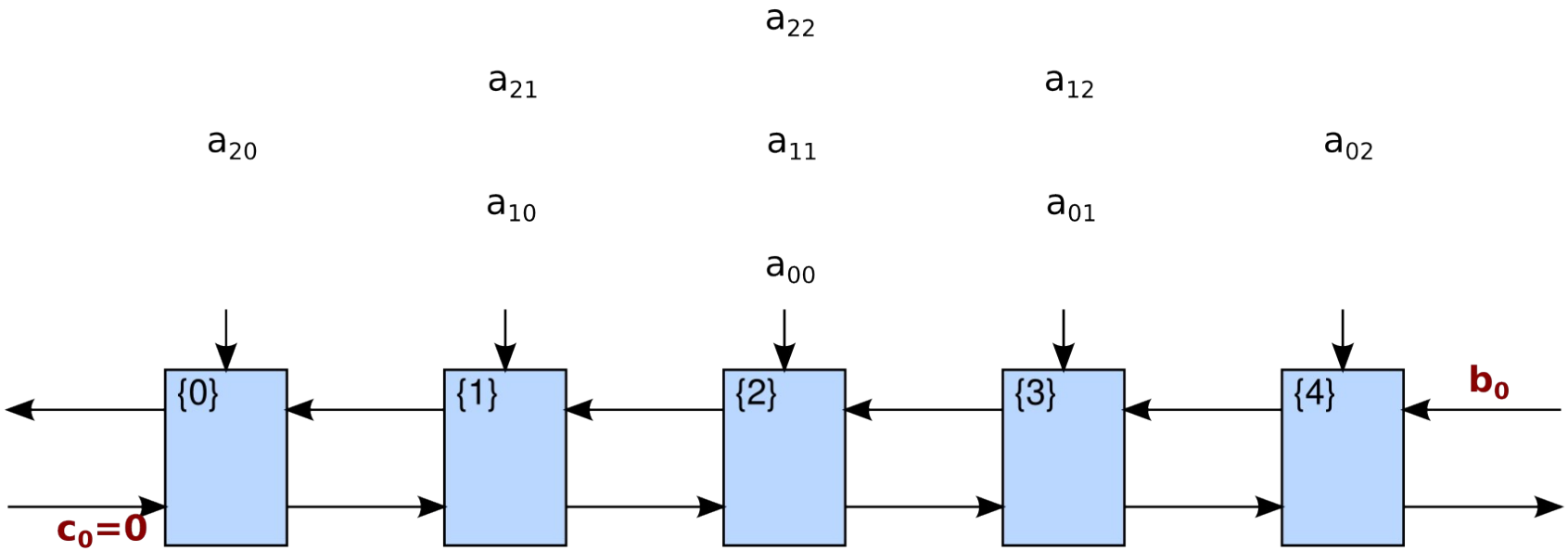


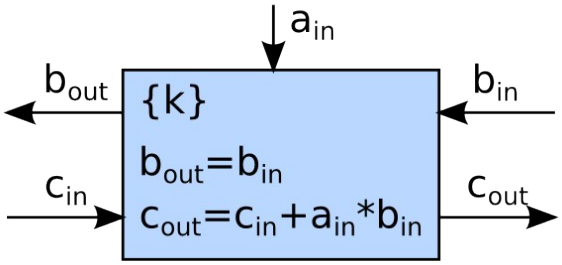


$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix}$$

Ciklus:

Projekció diagonál irányban, ütemezés

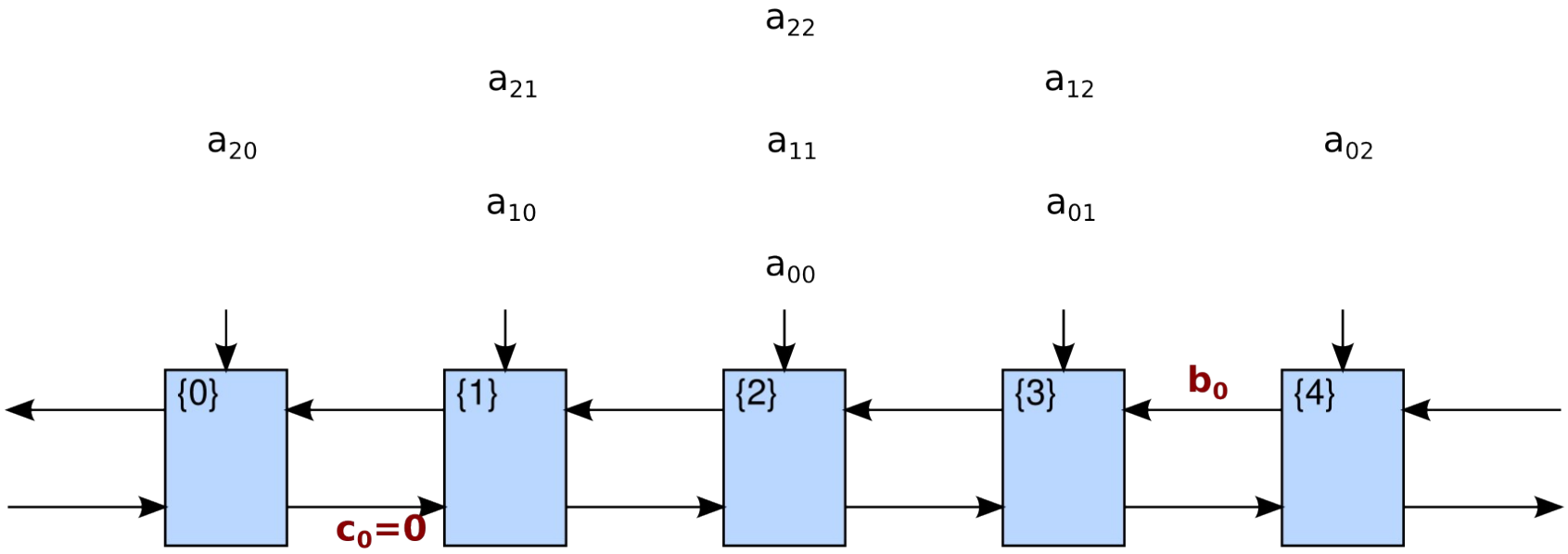


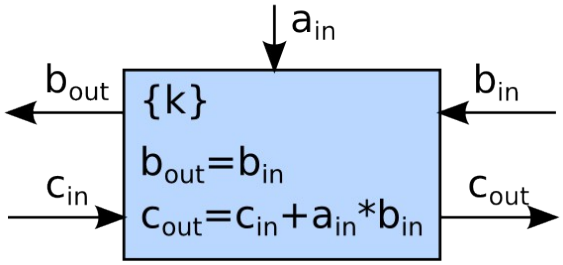


$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix}$$

Ciklus: 1

Projekció diagonál irányban, ütemezés

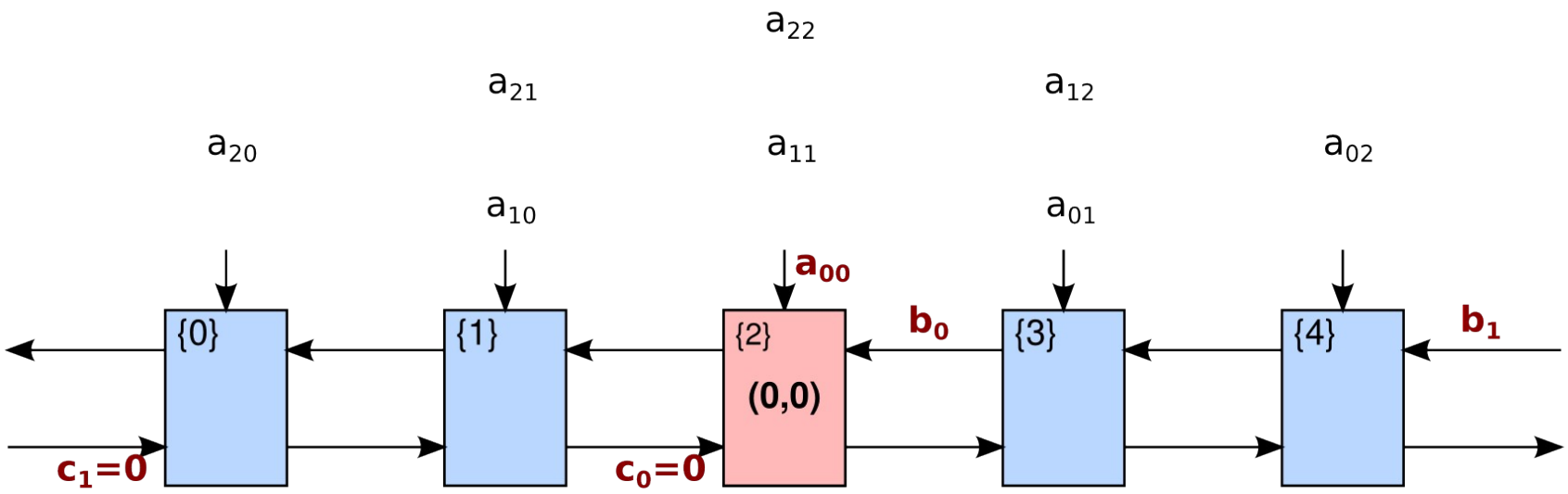


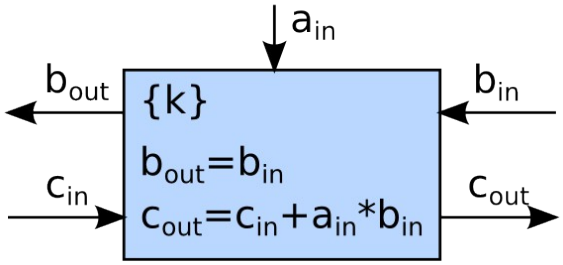


$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix}$$

Ciklus: 2

Projekció diagonál irányban, ütemezés

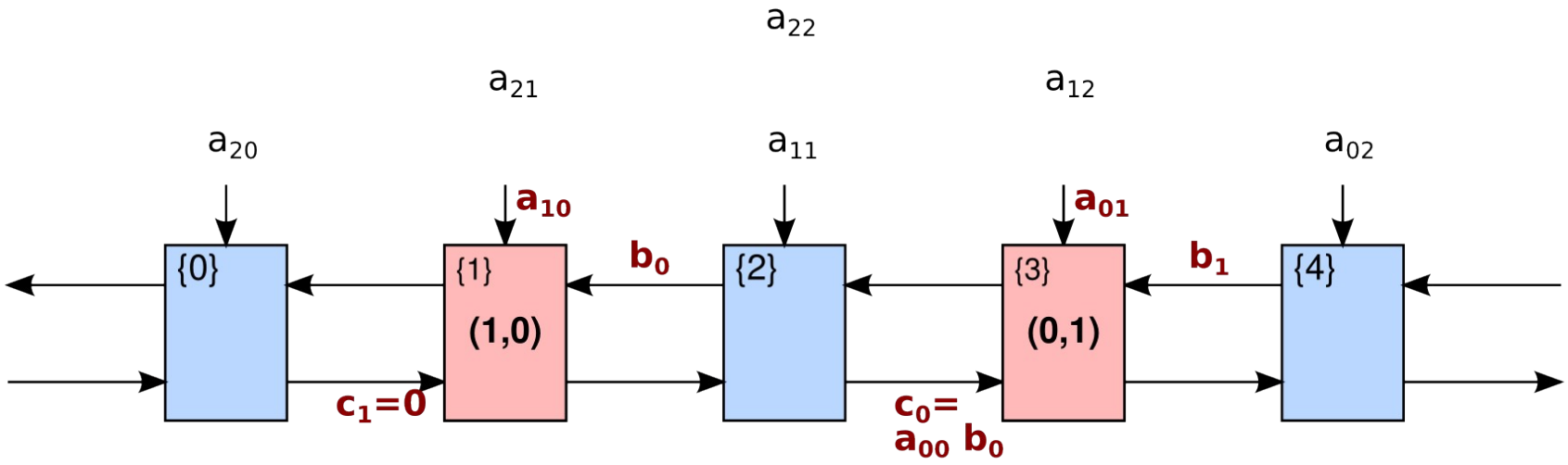


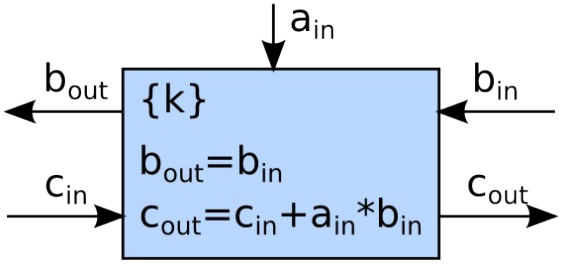


$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix}$$

Ciklus: 3

Projekció diagonál irányban, ütemezés

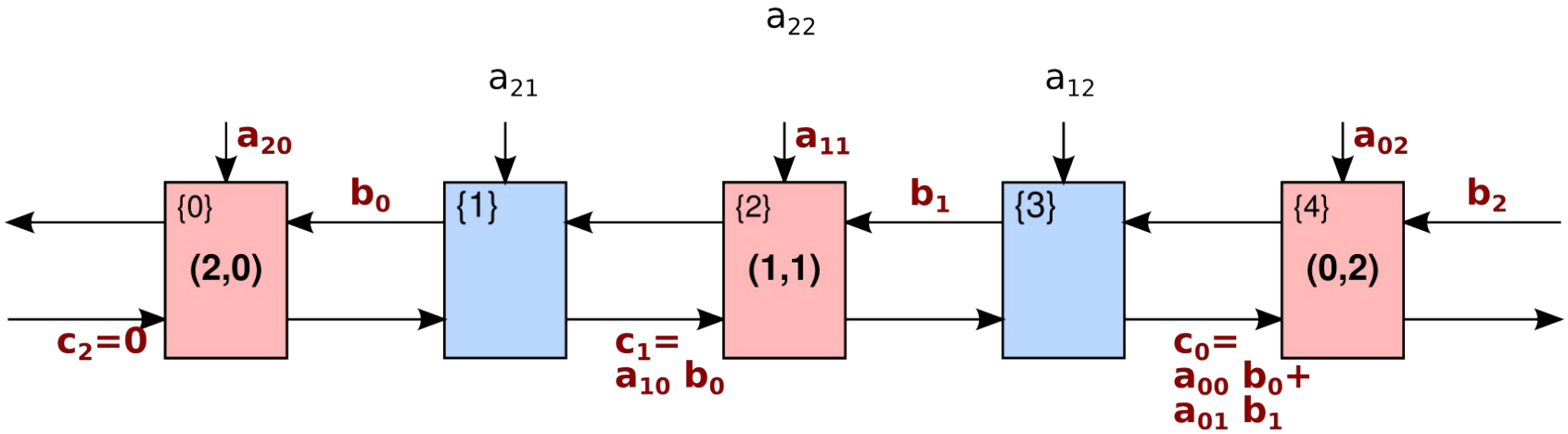


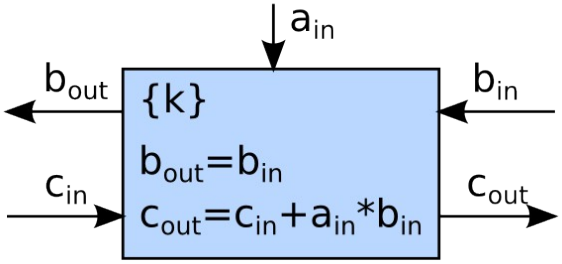


$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix}$$

Ciklus: 4

Projekció diagonál irányban, ütemezés

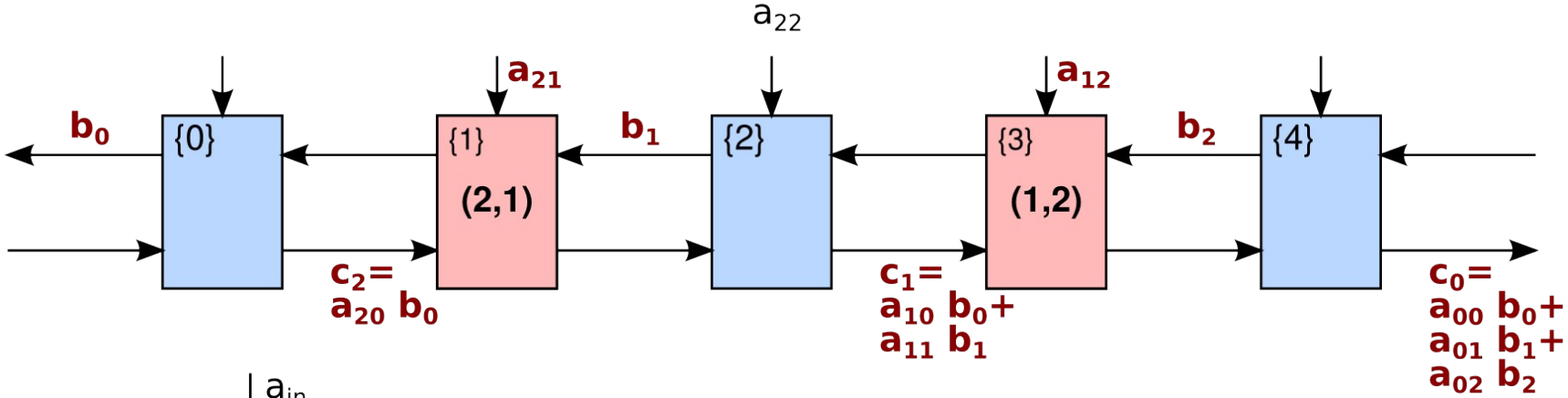


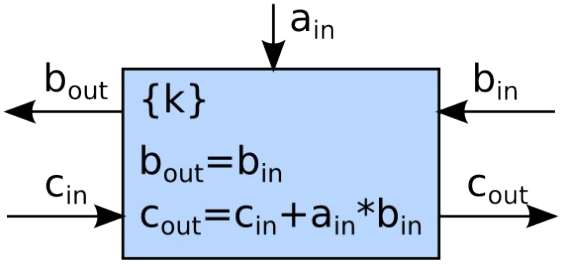


$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix}$$

Ciklus: 5

Projekció diagonál irányban, ütemezés

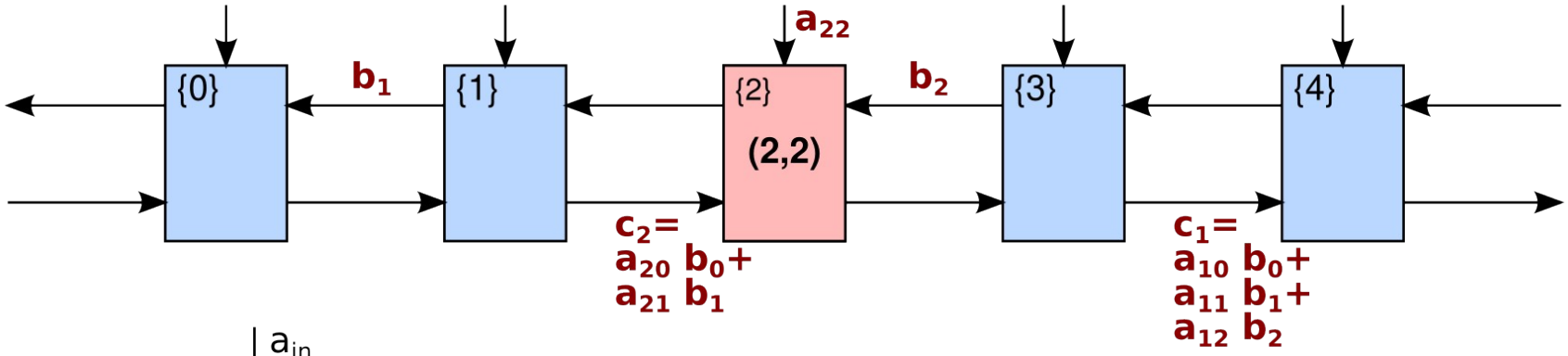


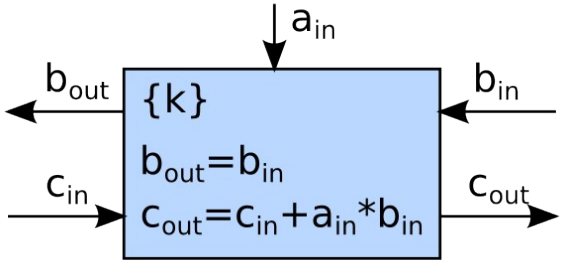


$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix}$$

Ciklus: 6

Projekció diagonál irányban, ütemezés



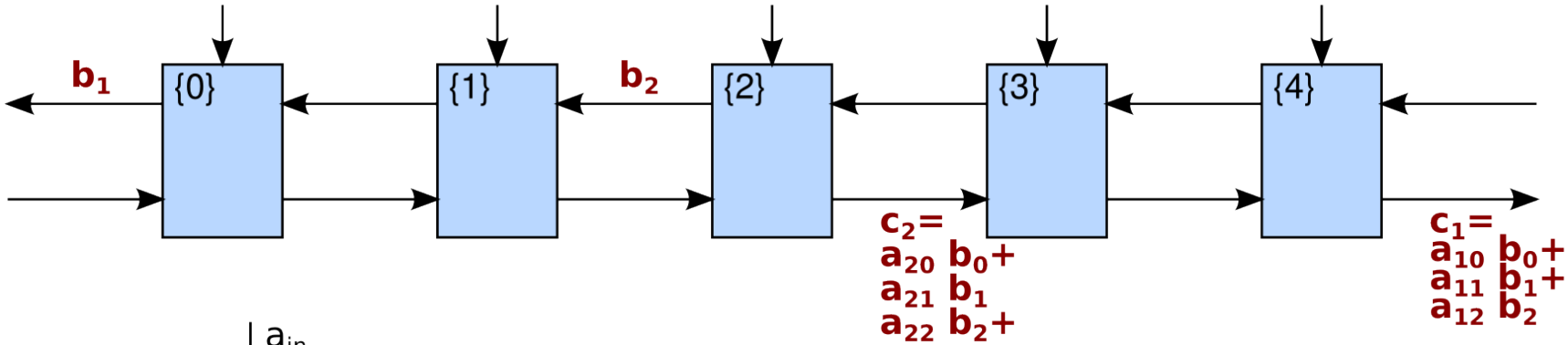


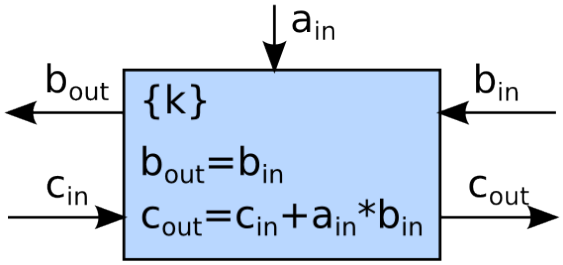
$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix}$$

$$c_0 = a_{00}\, b_0 + a_{01}\, b_1 + a_{02}\, b_2$$

Ciklus: 7

Projekció diagonál irányban, ütemezés



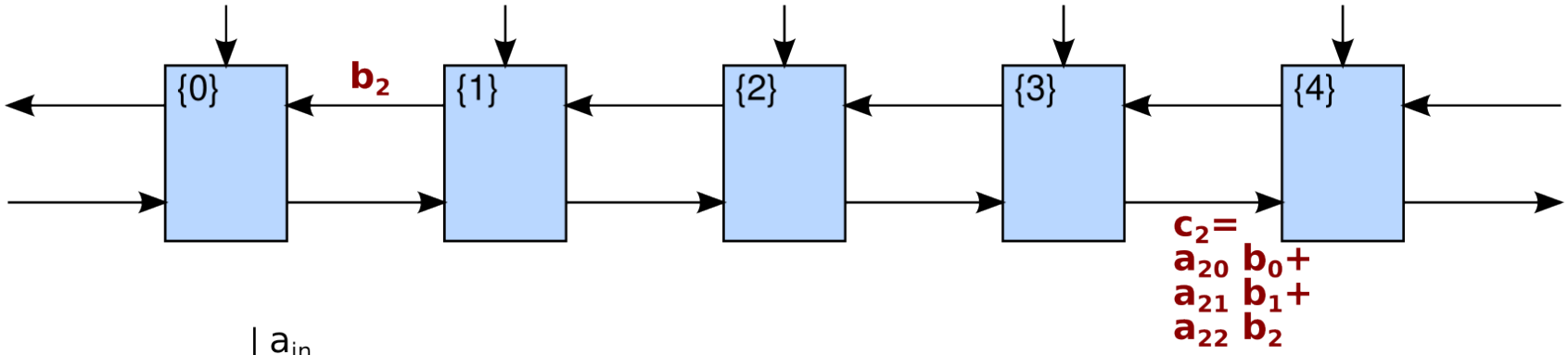


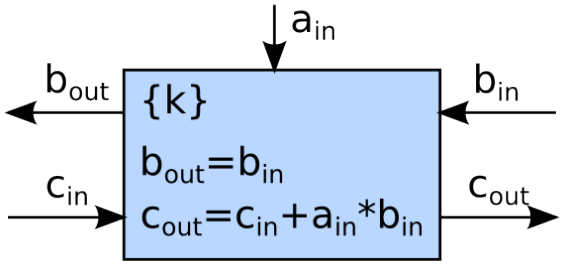
$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix}$$

$$c_0 = a_{00}\, b_0 + a_{01}\, b_1 + a_{02}\, b_2$$

Ciklus: 8

Projekció diagonál irányban, ütemezés



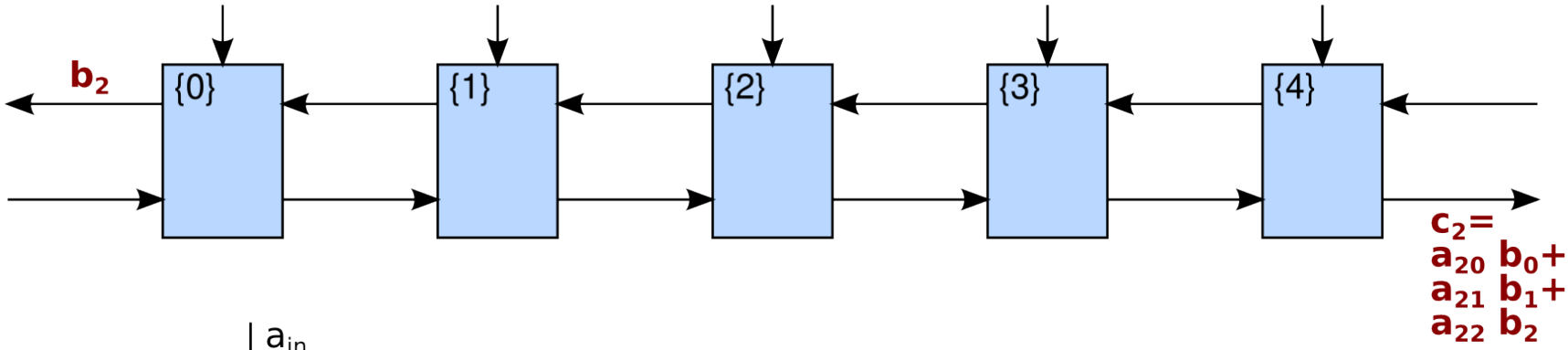


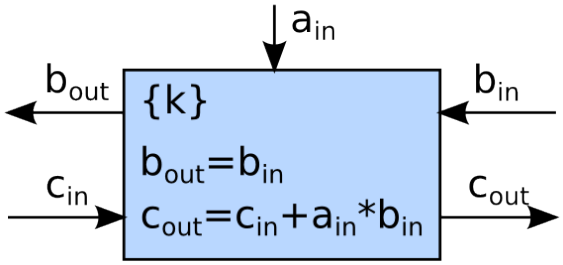
$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix}$$

$$c_0 = a_{00}\, b_0 + a_{01}\, b_1 + a_{02}\, b_2$$
$$c_1 = a_{10}\, b_0 + a_{11}\, b_1 + a_{12}\, b_2$$

Ciklus: 9

Projekció diagonál irányban, ütemezés



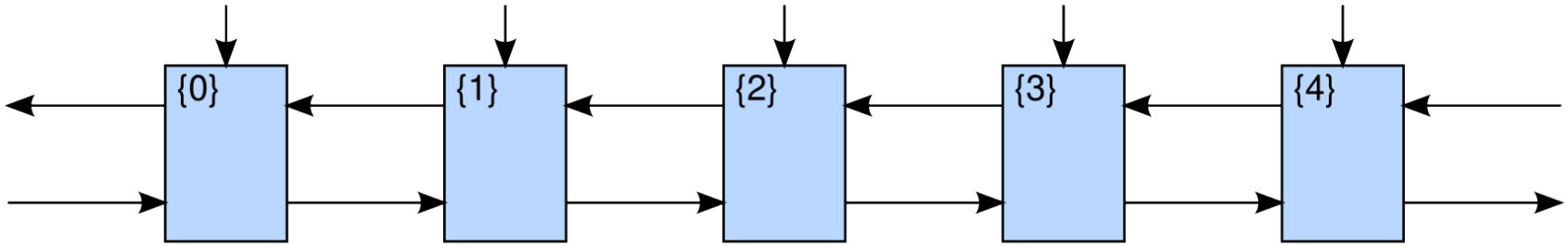


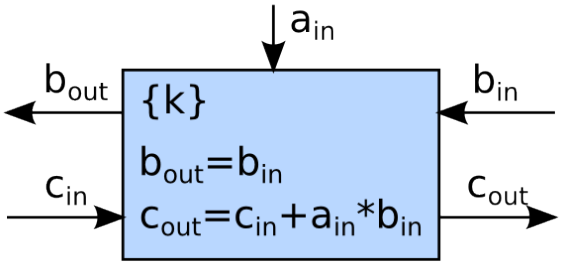
$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix}$$

$$c_0 = a_{00}\, b_0 + a_{01}\, b_1 + a_{02}\, b_2$$
$$c_1 = a_{10}\, b_0 + a_{11}\, b_1 + a_{12}\, b_2$$

Ciklus: 10

Projekció diagonál irányban, ütemezés





$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix}$$

$$c_0 = a_{00}\, b_0 + a_{01}\, b_1 + a_{02}\, b_2$$
$$c_1 = a_{10}\, b_0 + a_{11}\, b_1 + a_{12}\, b_2$$
$$c_2 = a_{20}\, b_0 + a_{21}\, b_1 + a_{22}\, b_2$$

Ciklus: 10

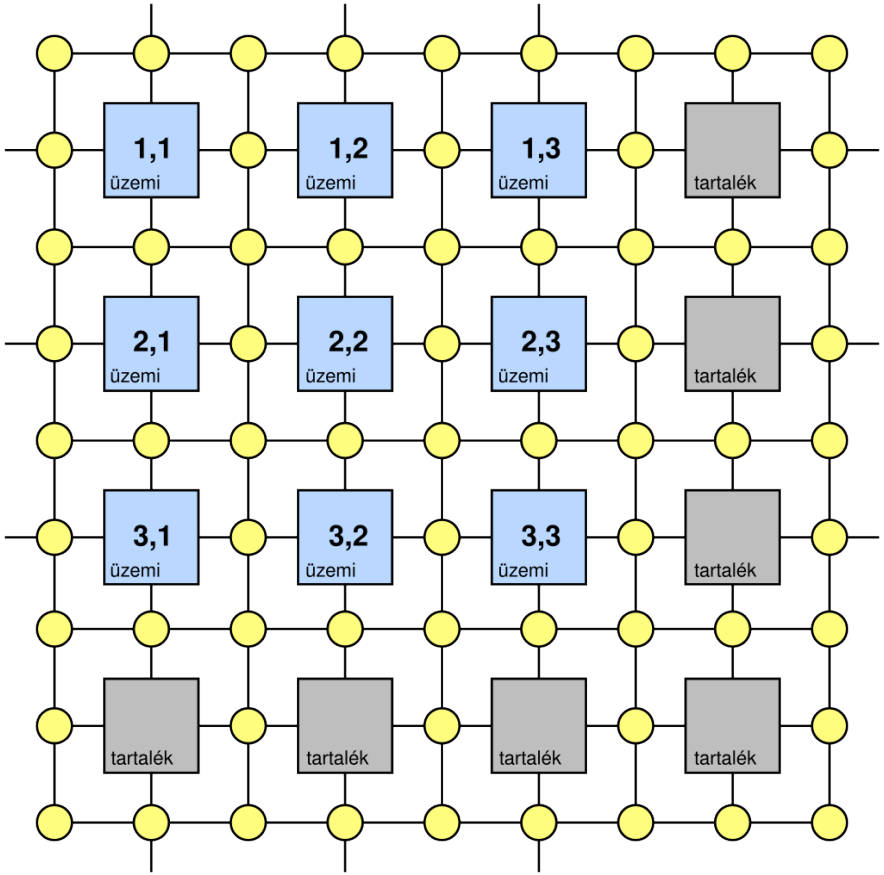
Tömbprocesszorok hibatűrése

Tömbprocesszorok hibatűrése

- Tömbprocesszorok hibatűrése:
  - Rossz.
  - Ha valaki elromlik, rossz az eredmény → kuka
- Van, ahol fontos a hosszú ideig hibamentes működés
  (műholdak, nehezen megközelíthető helyek, stb.)
- Jó lenne tartalék feldolgozóegységeket betenni
- De akkor átkonfigurálhatóvá kell tenni az összeköttetéseket

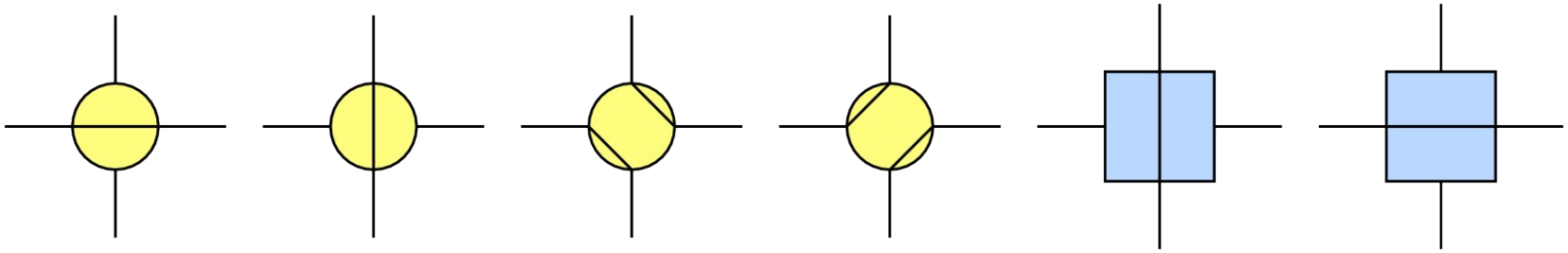
Tömbprocesszorok hibatűrése

- Egy lehetséges megoldás:
  - Üzemi és tartalék feldolgozóegységek
  - Kapcsoló csomópontok

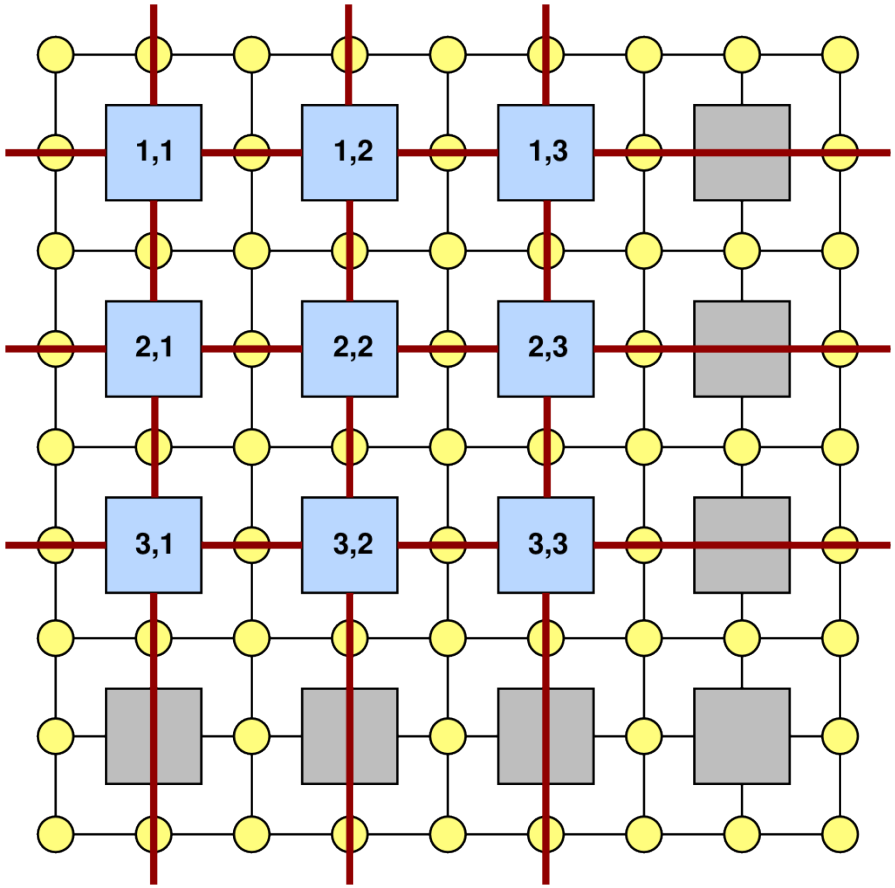


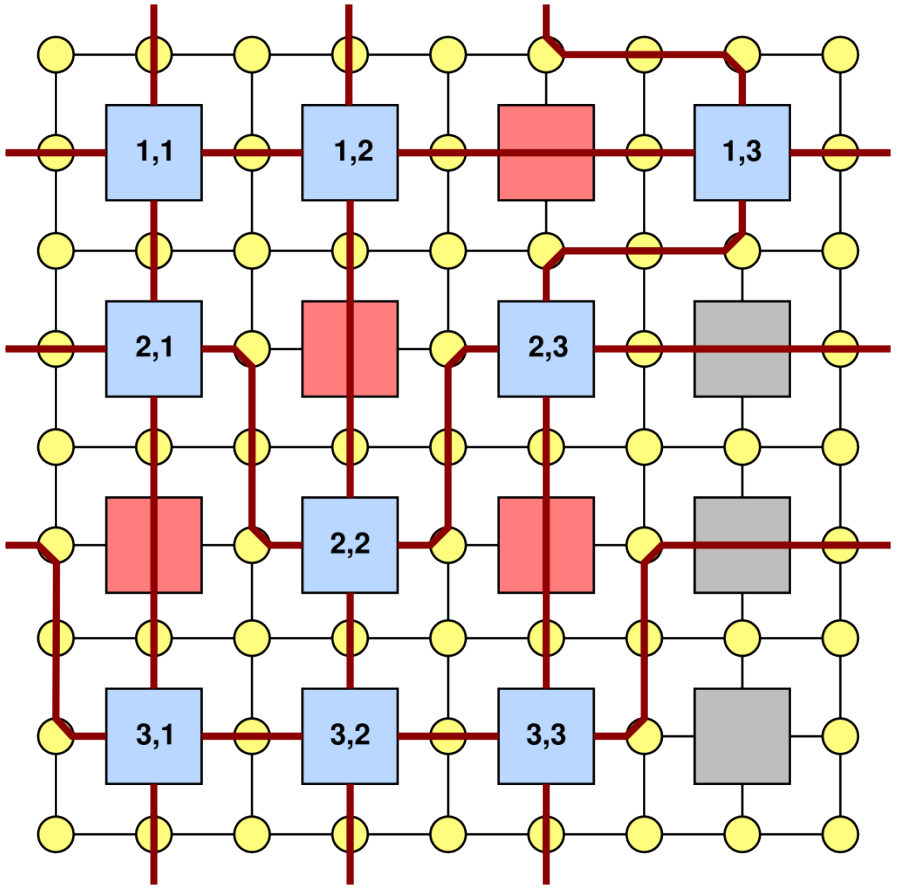
Tömbprocesszorok hibatűrése

- Átkonfigurálás:
  - Kapcsolók 4-féle lehetséges állásba állíthatók
  - Rossz feldolgozóegységek rövidzárhatók
    - Input rögtön továbbmegy a kimenetre



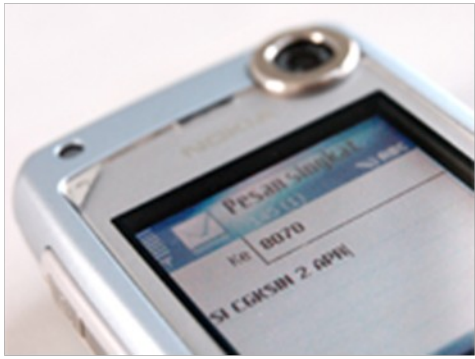
Példa hibaállapotra





SIMD konklúzió

- Nagy számítási teljesítmény egy módja:
  - Több adaton is elvégezzük ugyanazt az utasítást
- Két SIMD megoldást néztünk
  - Vektorprocesszort
    - Vektorregiszterek + vektorműveletek
    - Speciális, vektorprocesszorra jellemző technikák
  - Tömbprocesszort
    - A szisztolikus és a hullámfront fajtája a nyerő
    - Nem könnyű ráhúzni egy algoritmust
    - De ha sikerül, nincs az a CPU, amire ne verne pár nagyságrendet



Számítógép Architektúrák

Multiprocesszoros rendszerek

Horváth Gábor
docens
BME Híradástechnikai Tanszék
ghorvath@hit.bme.hu

2012. december 3.
Budapest

Párhuzamosság formái

- A párhuzamosság milyen formáit ismerjük?
  - Bit szintű párhuzamosság
    - Aritmetikai műveletekben használt adategységek mérete
    - 4-8-16-32 bit relatíve gyors volt, 64 bit később jött, és jó darabig elég is lesz
  - Implicit párhuzamosság
    - Programban lévő párhuzamosságot a programozó tudta nélkül használjuk ki
    - Programozó szekvenciális programot ír
    - Nem kell a programot hardverhez igazítani
    - Tanult megvalósításai:
      - Utasítás átlapolás → utasítás pipeline
      - Műveleti egység többszörözés → szuperskalár, VLIW

Párhuzamosság formái

- A párhuzamosság milyen formáit ismerjük? Folyt...
  - Explicit párhuzamosság
    - Implicit párhuzamosság gátja:
      - Adatfüggőségek
      - Miért rontják a hatékonyságot?
      - Mert a programozó nem is tudja, hogy lehetősége van az utasítások párhuzamos végrehajtására (vagy a programnyelv nem engedi leírni)
    - Megoldás:
      - Explicite mondja meg a programozó, hogy a programjának mely részeit lehet párhuzamosan végrehajtani!
      - Vezérlőtoken többszörözés / összevárás (FORK/JOIN)

Párhuzamosság formái

- Explicit párhuzamosság hardver támogatása:
  - Több végrehajtási szálat kezelő processzorok
  - Multiprocesszoros rendszerek

Több végrehajtási szálat kezelő processzorok

Több szálat kezelő processzorok

- Mi az utasítás-pipeline rákfenéje?
  - Cache hiba
  - TLB hiba
  - Adatfüggőségek
- Ilyenkor mi történik?
  - Semmi

    → Szünet történik
- Megoldás:
  - Kezeljünk több utasításszámlálót!
  - Ha fennakadás van, a megoldásig tegyük félre az utasítássorozatot, és vegyük elő a másikat

    → Sokkal kevesebb szünet
  - Ezt csinálják a több szálat kezelő processzorok

Több szálat kezelő processzorok

- De mi az a szál?
- Szál = thread
- Ezek multi-thread processzorok
- … és nem multi-task processzorok
- Mi a különbség??

Multi-thread vs. multi-task

- ***Multi-tasking***
  - Több „program” végrehajtása „egyszerre”
  - Már 50 éve is támogatták a profi op. rendszerek
  - Egy processzoron is megy: ekkor a párhuzamos végrehajtásnak csak a látszatát keltik → ***időosztással***
    - Hozzávalók:
      - 1 db időzítő, ami fix időközönként interruptot generál
      - 1 db megszakításkezelő rutin, ami ***taszkot vált***
        1) Megszakítja az aktuális taszk végrehajtását, állapotát elmenti (regiszterek, verem mutató, stb.)
        2) Kiválaszt egy korábban megszakított taszkot
        3) Visszatölti annak állapotát
        4) Visszatölti az utasításszámlálóját → mostantól őt hajtja végre a processzor

Multi-thread vs. multi-task

- Igazából több dolgot is kell csinálni a taszkváltáskor:
  - Laptáblát váltani (minden taszknak saját laptáblája lehet)
  - TLB-t üríteni
  - Esetleg utasítás cache-t üríteni
- Ez is eltart egy darabig, de utána jön még a sok TLB és cache hiba, mire mindkettő újra feltöltődik

  → ***Taszkváltás nagyon lassú!***
- Akár 2 - 3 ezer órajel is lehet (Intel Pentium4)

Multi-thread vs. multi-task

- ***Multi-thread:***
  - Gyakran egy program (taszk) belül maga is párhuzamos részekre, ***szálak***ra bomlik
  - Mert a programozó így akarta
  - Thread-ek jellemzői:
    - Osztoznak a taszk memóriáján (lokális változóin is)
    - A szálak külön utasításszámlálóval és regiszterkészlettel rendelkeznek, *de ez minden*, amiben különböznek
    - Egy taszkon belül a szálak váltása (időosztásos rendszerben):
      - Nem kell laptáblát váltani → nincs TLB és cache ürítés
      - Csak a regisztereket kell váltani

      → ***A szálváltás rendkívül gyors!***

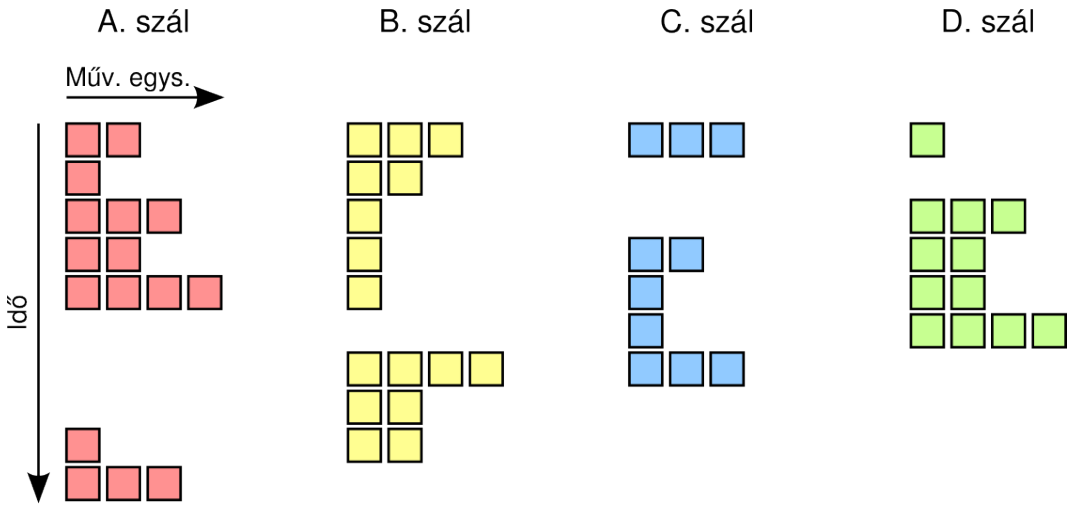
Multi-thread vs. multi-task

- Tanulság:
  - Több taszk párhuzamos futásának látszatát egy processzoron időosztással lehet kelteni
  - A taszkok közötti váltás nagyon lassú
  - Egy taszkon belül a szálak közötti váltás nagyon gyors
- A több szálat kezelő (multi-threading) processzorok
  - Meglepetés: valójában ***több taszkot*** támogató processzorok!
  - Zavaró elnevezési inkonzisztencia
  - A továbbiakban a számítógép-architektúrák területén elterjedt elnevezést használjuk: többszálúság, szálak (valójában multi-taszk, taszkok)

Időosztásos többszálú végrehajtás

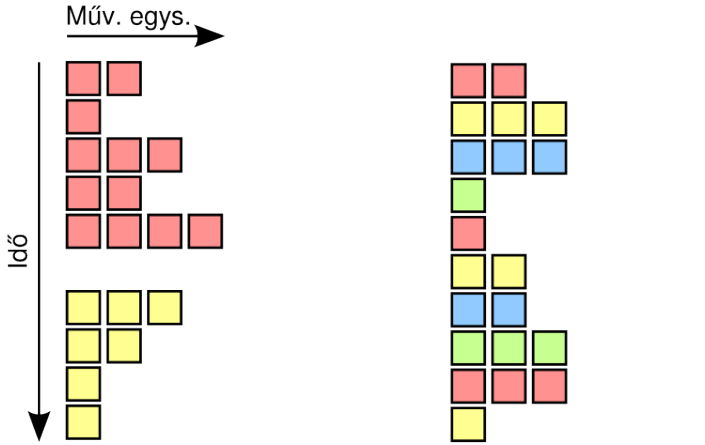
- A többszálú végrehajtást támogató processzorok egy része
  - ***Finom felbontású multi-threading***-et támogat:
    - Minden órajelben szálat vált
    - Mire egy szálra újra sor kerül, jó eséllyel megoldódik a problémája (nem kell külön szünet)
  - ***Durva felbontású multi-threading***-et támogat:
    - Csak akkor vált szálat a processzor, ha az aktuális megakad
- Hardveres megvalósítás:
  - Szálváltási idő: 0 vagy 1 órajelciklus (finom, ill. durva esetben)
  - Utasítások belépéskor egy szálazonosítót kapnak
  - A processzorban több utasításszámláló és regiszter-tároló van
  - TLB, cache, elágazásbecslő adatszerkezetek, stb. osztottak
    - Bejegyzésekben új mező: melyik szálhoz tartozik
  - Minden utasítás a szálazonosítójának megfelelő regiszterkészlettel, TLB és cache bejegyzéssel, stb. dolgozik

Időosztásos többszálú végrehajtás



Durva felbontású többszálú végrehajtás

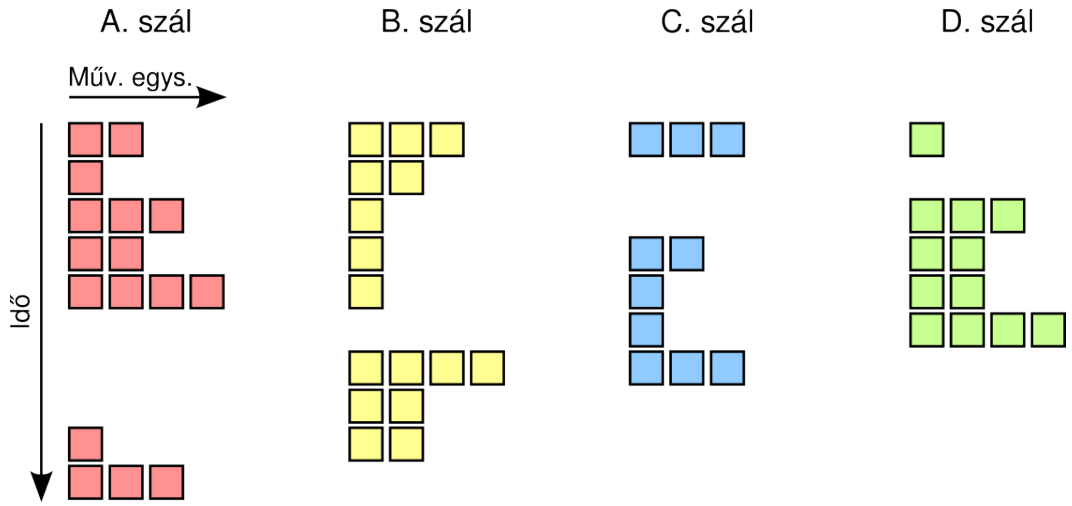
Finom felbontású többszálú végrehajtás



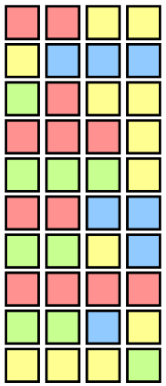
Szimultán többszálú végrehajtás

- Az időosztásos alternatívája
- Csak szuperskalár architektúrával működik
- Szuperskalár esetben a sok műveleti egység gyakran kihasználatlan
  - Mert nincs elég párhuzamosítható utasítás a programban
- A nem használt műveleti egységeken hajtsuk végre egy másik szál utasításait!

Szimultán többszálú végrehajtás



Szimultán
többszálú végrehajtás



Több szálat kezelő processzorok

- Minden formáját könnyű hardveresen megvalósítani

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 POWER6	2010	Szimultán	4
UltraSPARC T1	2005	Finom felbontású	4
UltraSPARC T2	2007	Finom felbontású	8

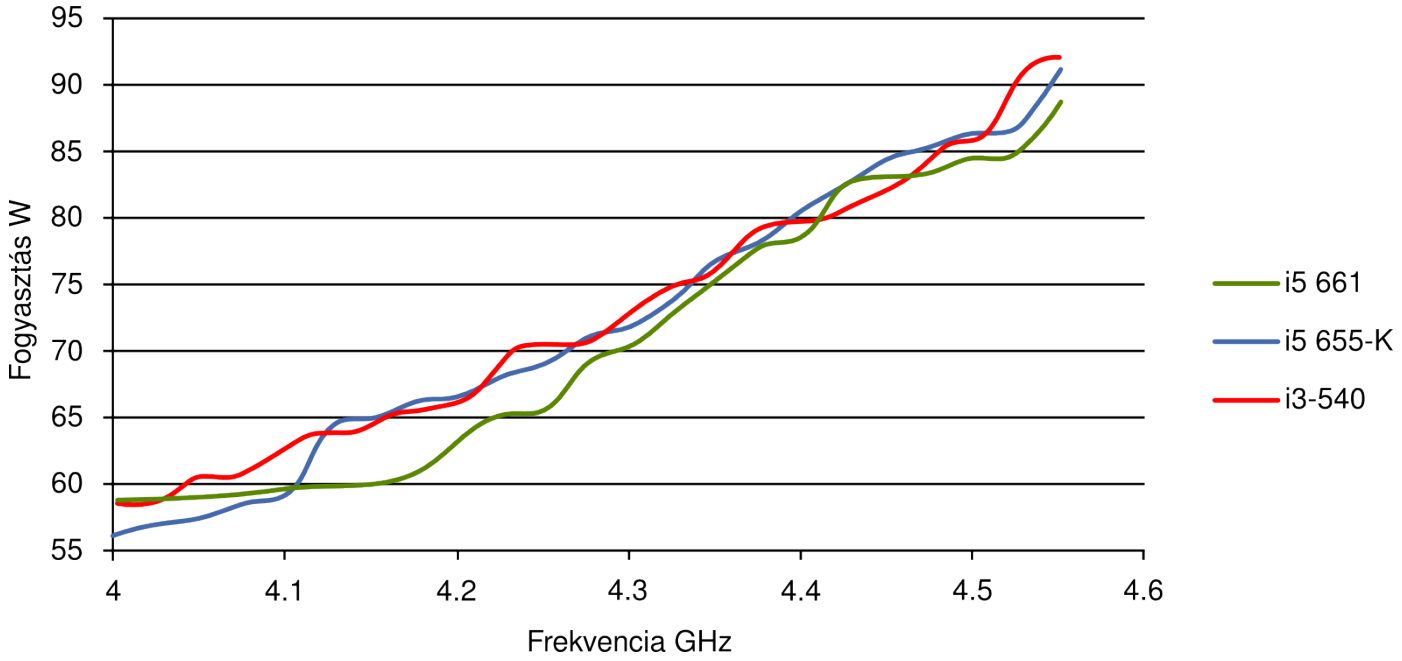
Multiprocesszoros rendszerek

Multiprocesszoros rendszerek

- Teljes értékű processzorokból álló rendszerek
- Tartalomjegyzék:
  - Miért is jó ez nekünk?
  - Mennyit nyerünk rajta?
  - Hogy is kell összekötni azokat a processzorokat?

Motiváció

- Az utasításszintű párhuzamosság lehetőségei korlátozottak. Az explicit párhuzamossággal nagyobb léptékű párhuzamosítás érhető el.
- Energiahatékonyság: fogyasztás meredeken nő CPU számítási teljesítménnyel (azonos technológiával):



Motiváció

- Költséghatékonyság: a processzorok ára exponenciálisan nő a számítási teljesítménnyel
  - Olcsóbb lassúból többet venni, mint gyorsból egyet
- Egyszerű bővíthetőség:
  - Nagyobb számítási teljesítményhez csak hozzá kell adni még pár processzort
- Hibatűrés:
  - Ha egy elromlik, a többi átveheti a feladatait

Korlátok

- A multiprocesszoros rendszerek világában minden szép és jó?
- Nem. A bajok forrása: ***a szoftver***
- Ki párhuzamosítja a programokat?
  - ***A fordító***:
    - A programozó szekvenciális programot ír
    - A fordító felfedezi benne a párhuzamosítható részeket
    - Az elterjedt programozási nyelvekhez ilyen intelligens fordító nincs (egyelőre)
  - ***A programozó***:
    - Nehéz
    - Rengeteg a potenciális hibalehetőség
    - Ez a ma uralkodó szemlélet

A teljesítménynövekmény korlátai

- Avagy: Ha N processzort veszek a gépembe N-szer gyorsabban fut a programom?

- Nem. Csak ha a probléma teljes mértékben párhuzamosítható.
- Általános esetben az Amdahl törvény adja meg a teljesítménynövekményt

Amdahl törvénye

- Legyen a programunk
  - P része tetszőlegesen párhuzamosítható
  - 1-P része szekvenciálisan végrehajtandó
- Legyen a futási idő 1 processzoros rendszerben: *1*
- Kérdés: mennyi a futási idő *N* processzor esetén?
  - Ha az egész szekvenciális lenne: 1
  - Ha az egész párhuzamosítható lenne: 1/N
  - Ha P része párhuzamosítható: (1-P)*1 + P/N
- ***Amdahl törvénye: teljesítménynövekmény az 1 processzoros rendszerhez képest:***

$$S_P(N) = \frac{1}{(1-P) + P/N}$$

Amdahl törvénye

- Példa: van 100 processzorunk. 80-szoros gyorsulást szeretnénk. A program mekkora része lehet nem párhuzamosítható?

$$P = \frac{S_P(N) - 1}{S_P(N)} \frac{N}{N-1} \simeq 0.9975$$

- Mindössze 0.25%-a lehet szekvenciális!
- Ha a kód 5%-a szekvenciális (*P=0.95*):

$$S_P(100) = \frac{1}{0.05 + 0.0095} = 16.8$$

- Mindössze 16.8-szoros a sebesség! 100 processzorral!

Amdahl törvénye

- Nézzük meg a formulát még egyszer:

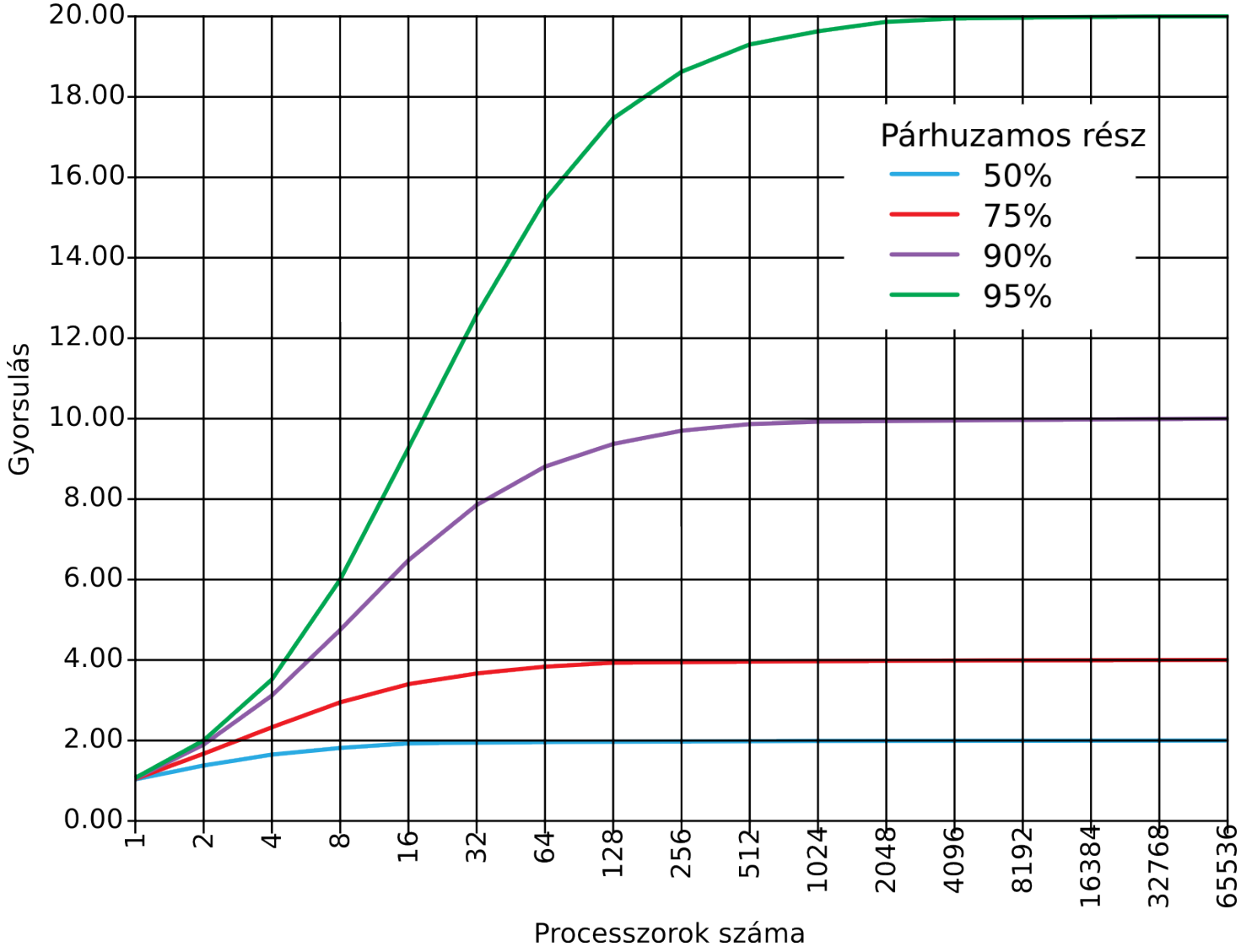
$$S_P(N) = \frac{1}{(1-P) + P/N}$$

- Végtelen sok processzorral:

$$S_P(\infty) = \lim_{N \to \infty} S_P(N) = \frac{1}{1-P}$$

- Akárhány processzorom is van, ennél nagyobb gyorsulás nem érhető el
  - A szekvenciális rész futási ideje korlátozza

Amdahl törvénye



Multiprocesszoros rendszerek osztályozása

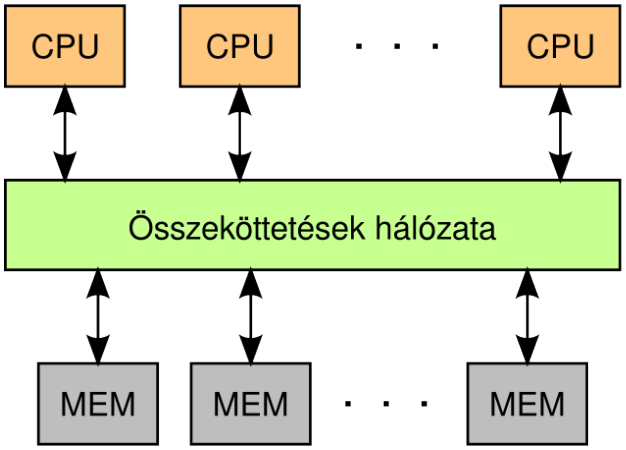
Osztályozás

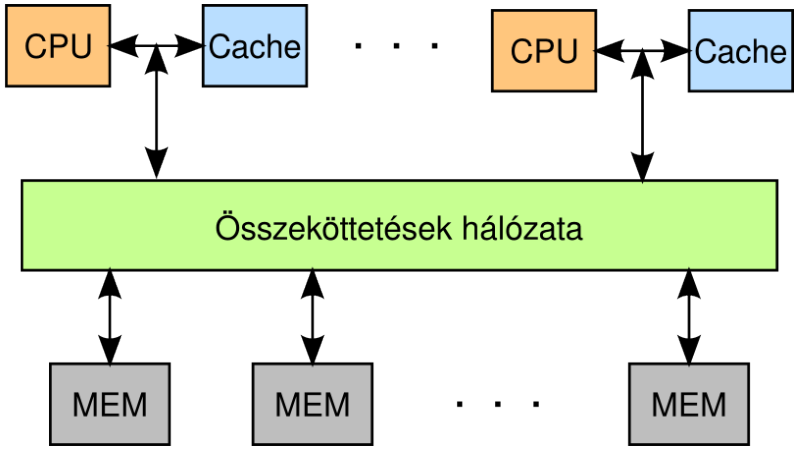
- A különböző processzorokon futó taszkok közötti ***kommunikáció szerint***:
  - Osztott memórián alapuló
    - Kommunikáció: egyik taszk beírja, másik kiolvassa
  - Üzenetküldésen alapuló
    - Kommunikáció: a taszkok üzenetet írnak egymásnak, és azt küldözgetik
- A ***memóriaműveletek költsége*** (futási ideje) ***szerint***:
  - UMA (Uniform Memory Access)
    - Minden memóriaművelet azonos ideig tart
    - Mindegy, hogy egy adat hol van: minden processzornak ugyanannyi ideig tart hozzányúlni
  - NUMA (Non-Uniform Memory Access)
    - Fontos, hogy egy adat hol van
    - Célszerű egy processzor által gyakran használt dolgokat hozzá közel elhelyezni

Multiproc. osztott memóriával

- A taszkok kommunikációja nagyon egyszerű

  UMA eset:



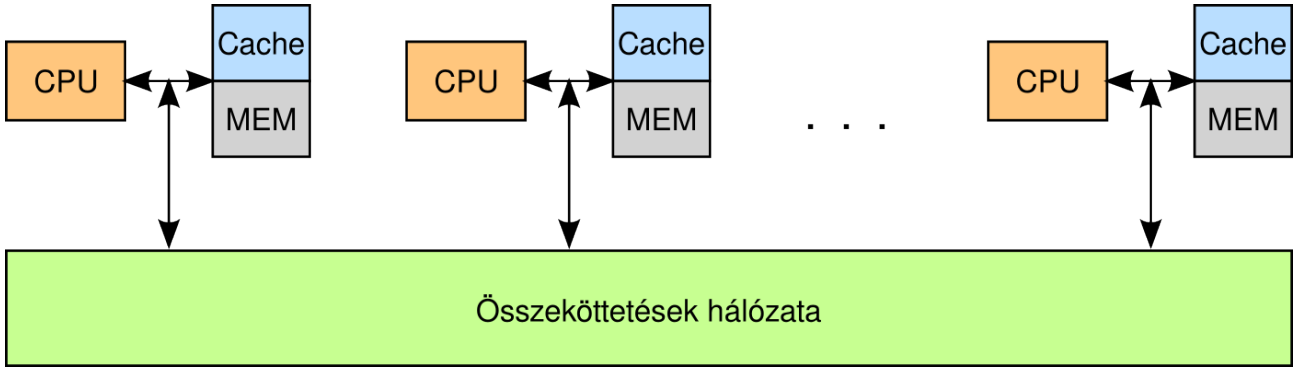


- Egyszerű megvalósítás
- Rossz skálázhatóság:
  - +1 új CPU berakásakor annak teljes memóriaforgalma terheli
- Max. 100 processzorig

Multiproc. osztott memóriával

NUMA eset:

- A rendszer csomópontjai: CPU, Cache, Memória egyben



- Címtér közös
  - Mindenki gyorsan hozzáfér a saját címtér darabkájához
  - Ha máséhoz akar hozzáférni, az a hálózaton valósul meg
  - Hálózati terhelés: csak a taszkok közötti kommunikáció esetén → skálázható!

Multiproc. üzenetküldéssel

- Csomópontok: teljes értékű számítógépek, külön címtartománnyal
- Kommunikáció: üzenetekkel
  - Explicit hívások:
    - Üzenet küldése (send)
    - Üzenet fogadása (bevárása, receive)
  - Minden taszknak saját egyedi azonosítója van
  - Üzenet tartalma:
    - Rakomány (payload)
    - Küldő azonosítója
    - Címzett azonosítója
  - Összeköttetés hálózat kézbesíti
- Üzenetküldés módjai:
  - Szinkron: A felek bevárják egymást
  - Aszinkron: A küldő nem várja meg a csomag kézbesítését, fut tovább

Összeköttetés-hálózatok

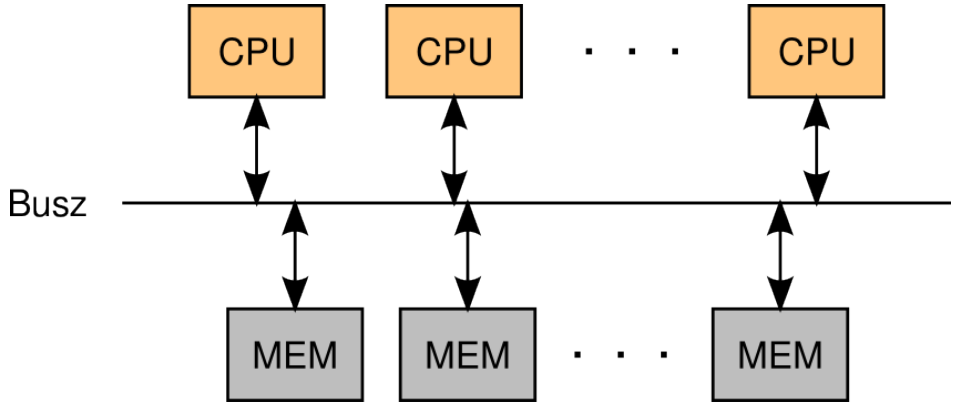
Összeköttetés-hálózatok

- Akár osztott memóriás, akár üzenetküldéses, akár UMA, akár NUMA: ***a szereplőket össze kell kötni***
- Kit kivel is kell összekötni?
  - Osztott memóriás UMA esetben:
    - Processzorokat a memóriákkal
  - Osztott memóriás NUMA esetben:
    - CPU-ból, cache-ből, memóriából álló modulokat egymással
  - Üzenetküldéses esetben:
    - A résztvevő komplett számítógépeket
- Mi is zajlik az összeköttetéseken?
  - Osztott memóriás esetben: memóriakérések / válaszok
  - Üzenetalapú esetben: üzenetek

Tulajdonságok

- Direkt összeköttetés-hálózatok:
  - A hálózat csomópontjai a multiprocesszoros rendszerek csomópontjai, maguk alkotják a hálózatot
- Indirekt összeköttetés-hálózatok:
  - A multiprocesszoros rendszer csomópontjai nincsenek egymással közvetlen kapcsolatban, csak a hálózattal

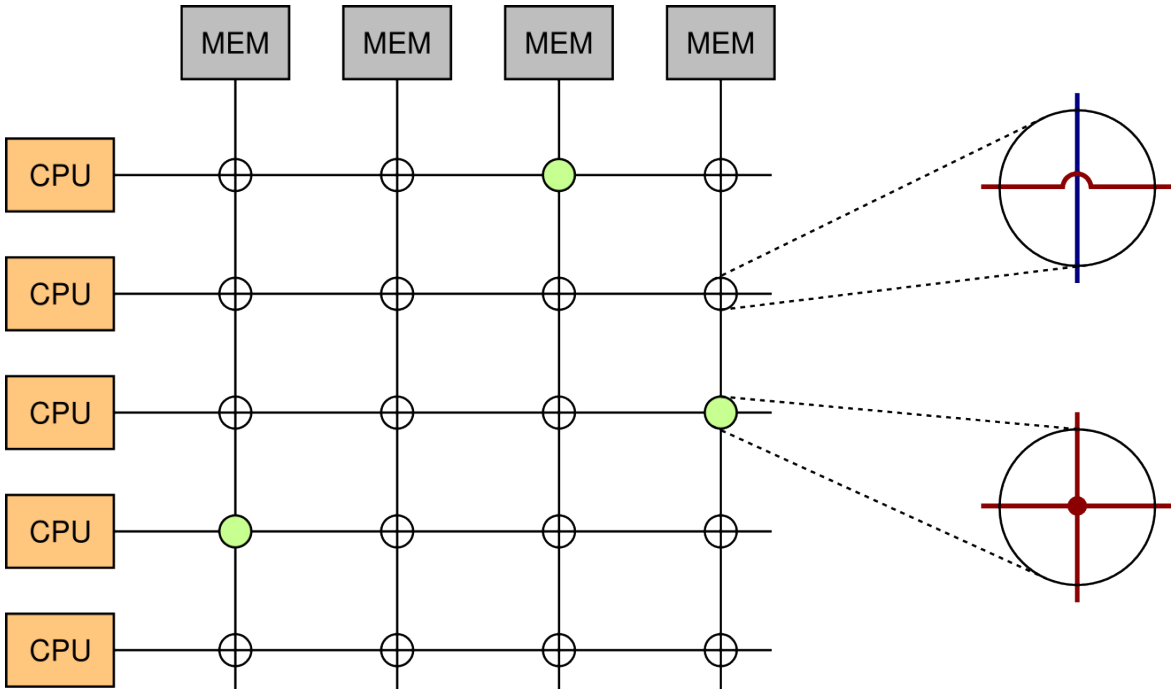
Busz



- Indirekt
- Előnyök:
  - Nemcsak pont-pont, hanem üzenetszórásos kommunikáció is lehet
  - Minden csomópont azonos távolságra van a többitől (késleltetésben)
  - A költség jól skálázódik: új csomópont hozzáadásával nem nő
- Hátrányok:
  - Az átvitel rosszul skálázódik: fix átviteli képességen osztoznak a csomópontok
- Sun Enterprise szerverek (1996-2001), Intel Pentium alapú rendszerek

Crossbar

- Indirekt
- P processzor, M memória összekötése P*M kapcsolóval

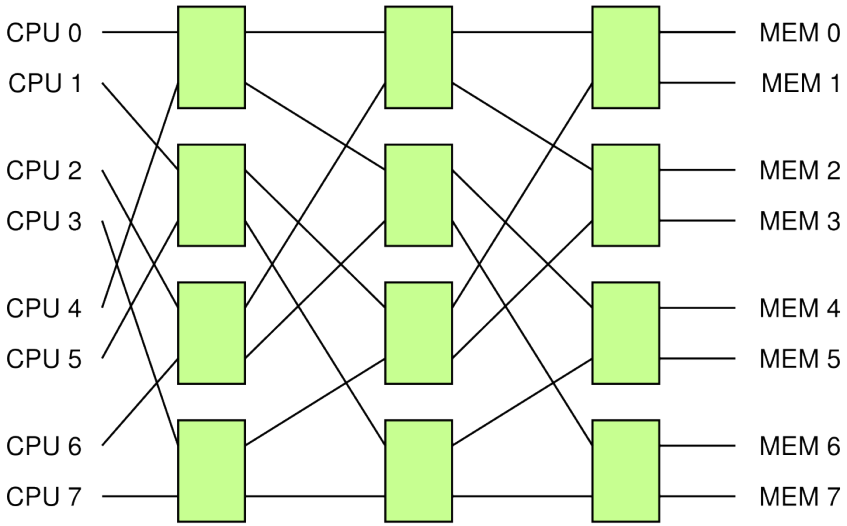


- Egyidejűleg több teljes sebességű kommunikáció is mehet
- Új CPU: 1 új összeköttetés és M új kapcsoló kell (ami drága)
- Példa: IBM POWER4, POWER5, Sun Niagara

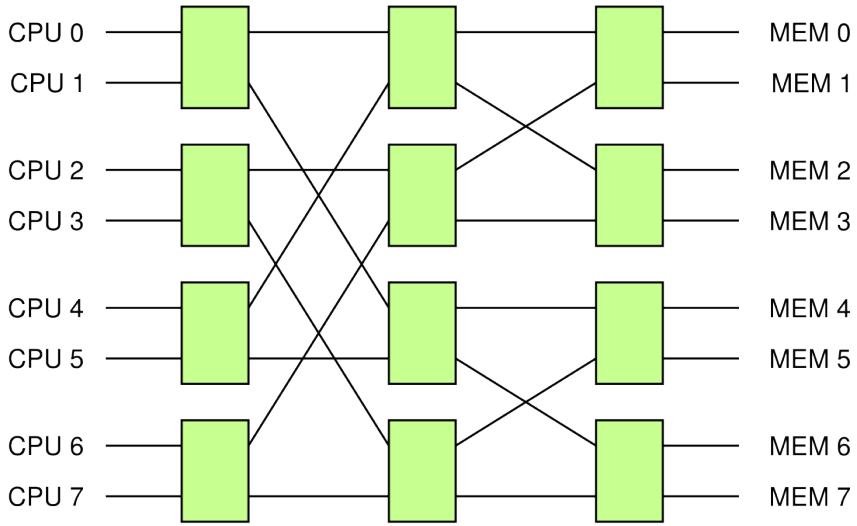
Többfokozatú crossbar

- Indirekt
- Kicsi (ált. 2x2-es) crossbar-okból építkezik
- Jól skálázódó, elmélettel jól alátámasztott lehetőség
- Több megoldás is létezik az összekötöttségi struktúrára

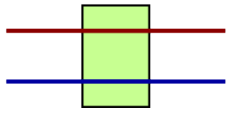
Omega hálózat



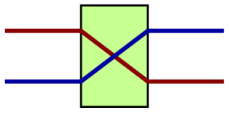
Butterfly hálózat



"Egyenes" állás:

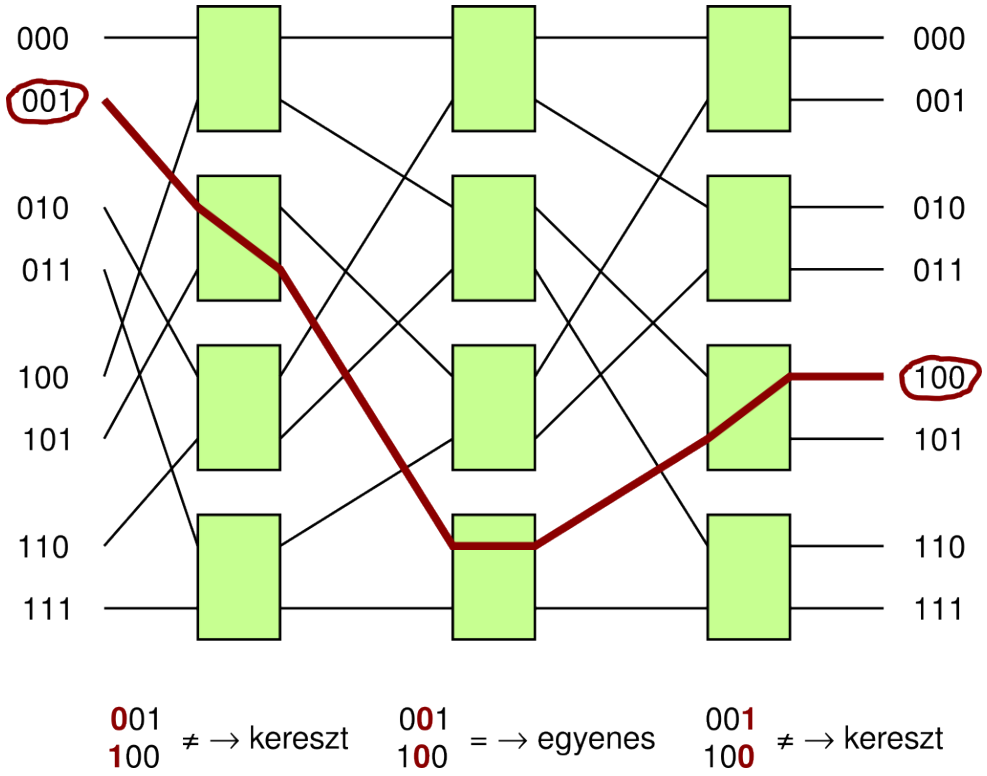


"Kereszt" állás:



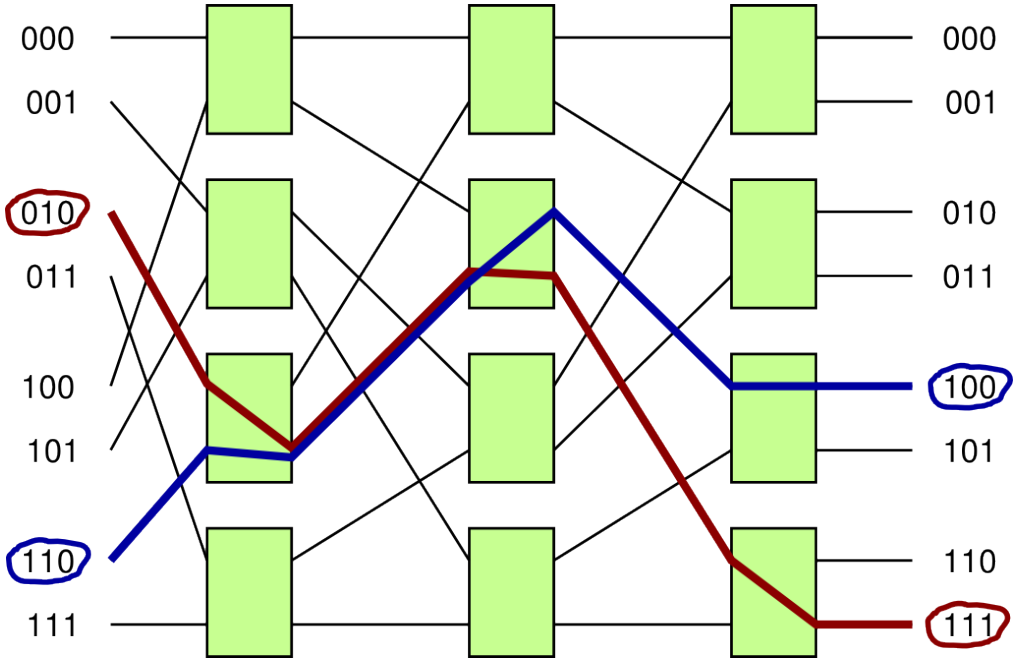
Többfokozatú crossbar

- Az omega hálózatban a kapcsolók működése nagyon egyszerű:
  - Az i. fokozatban a küldő és a címzett azonosítójának i. bitjét kell összehasonlítani



Többfokozatú crossbar

- Sajnos ez sem blokkolás-mentes:

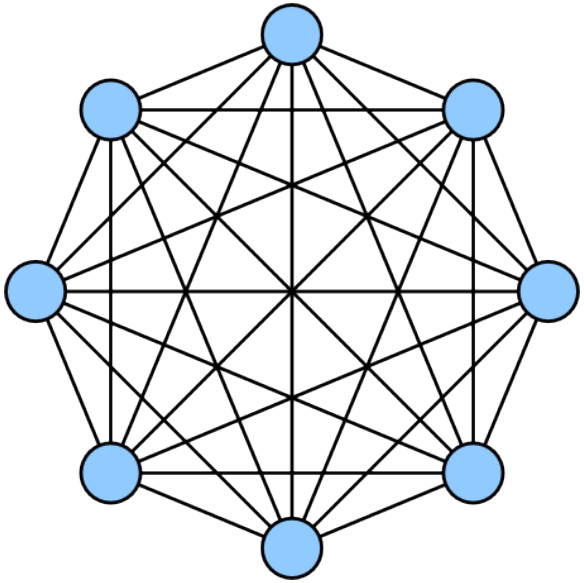


Többfokozatú crossbar

- *P* processzor esetén a szükséges kapcsolók száma
  - Többfokozatú crossbar-ral: P*log2(P)
  - Egyszerű crossbar-ral: P*P
- Példa: IBM RP3

Teljesen összekötött hálózat

- Direkt összeköttetés



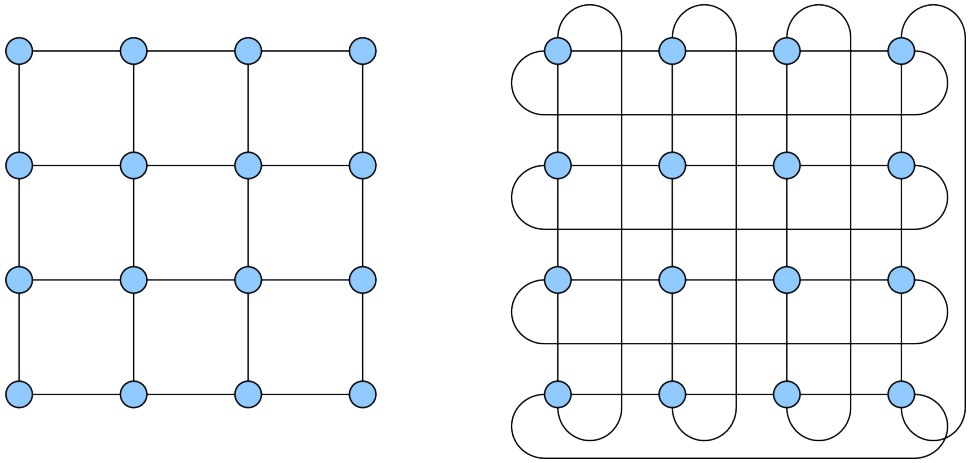
- Tulajdonságok *N* csomóponttal:
  - Átmérő: 1
  - Felezési keresztmetszet: N/2 * N/2
  - Élösszefüggőség: N-1
- Költség szempontjából rosszul skálázódnak:
  1 új csomópont → mindenki máshoz új összeköttetés kell
- Átvitel szempontjából jó skálázódnak:
  az új csomópont nem terheli a már meglévő összeköttetéseket

Lineáris és gyűrű topológiák

- Direkt összeköttetés

- Tulajdonságok *N* csomóponttal (lineáris és gyűrű):
  - Átmérő: N / N/2
  - Felezési keresztmetszet: 1 / 2
  - Élösszefüggőség: 1 / 2
- Költség szempontjából jól skálázódnak:
  1 új csomópont → csak 1 új összeköttetés
- Átvitel szempontjából rosszul skálázódnak:
  az új csomópont kommunikációs többlete is ezt a kevés élet terheli

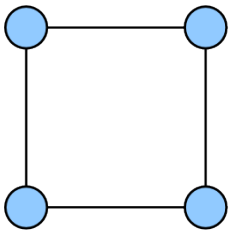
Kétdimenziós háló és tórusz



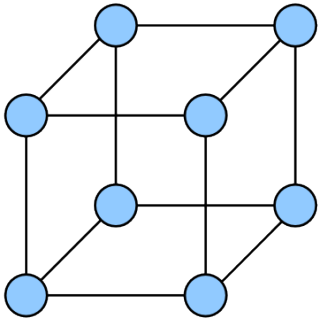
- Tulajdonságok *N* csomóponttal (háló és tórusz):
  - Átmérő: $2\sqrt{N}$ / $\sqrt{N}$
  - Felezési keresztmetszet: $\sqrt{N}$ / $2\sqrt{N}$
  - Élösszefüggőség: 2 / 4
- Költség és átvitel szempontjából való skálázhatóság szempontjából ez a gyakori kompromisszum
- Példa:
  - Cray T3E (3D háló, meteorológiai célokra)
  - IBM Blue Gene/L

Hiperkocka

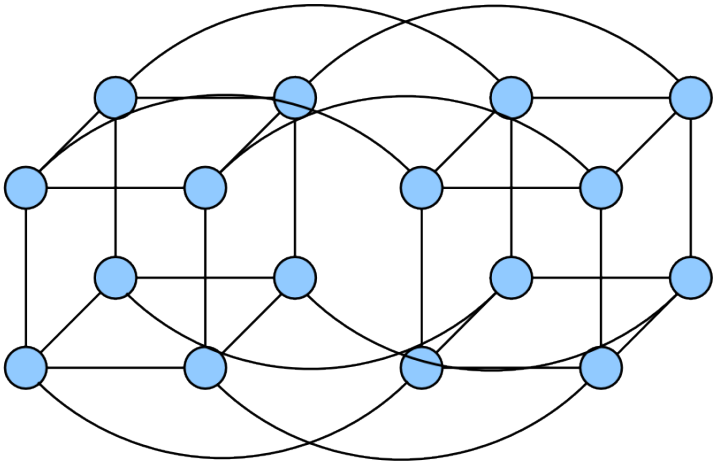
- Direkt összeköttetés



2D hiperkocka



3D hiperkocka



4D hiperkocka

- N csomópont → log2(N) dimenzió
- Tulajdonságok (N csomópont, k=log2(N) dimenzió esetén):
  - Átmérő: k
  - Felezési keresztmetszet: N/2
  - Élösszefüggőség: k
- Nagyon elterjedt, sokféle topológiát belé lehet ágyazni

Útvonalválasztás

Útvonalválasztás

- Feladata: milyen útvonalon jut el az üzenet a küldőtől a címzettig
- Döntési pontok:
  - A multiprocesszoros rendszerek csomópontjai
    - Direkt összeköttetéseknél:
      - CPU-ból, memóriából és cache-ből álló rendszerek,
      - Vagy processzorok ill. memória modulok
    - Indirekt összeköttetéseknél:
      - A hálózat csomagtovábbító eszközei
- Két kategória:
  - Statikus: az útvonal csak a küldőtől és a címzettől függ
  - Dinamikus: A hálózat pillanatnyi állapota is szerepet játszik

Statikus útvonalválasztás

- Három elterjedt megoldás:
  1) **Útvonalválasztás dimenzió-sorrendben:**
     - Szabályos geometriai struktúra esetén
     - Csomópontokat elhelyezkedésük alapján azonosítjuk
     - Példa 2D háló esetén: címzett: (x,y)
       - Kapcsoló ránéz: x balra van: balra, x jobbra van: jobbra küldi
       - Ha x koord. stimmel, az y-ra néz: ha feljebb van, arra, ha lejjebb, arra küldi
  2) **Küldő által meghatározott útvonalválasztás**
     - Küldő kitalálja az útvonalat, és beleírja az üzenetbe
     - Intelligencia a küldőnél: pontosan ismernie kell a hálózatot
     - A csatolt útvonal növeli az üzenetek méretét

Statikus útvonalválasztás

- ...folytatás:

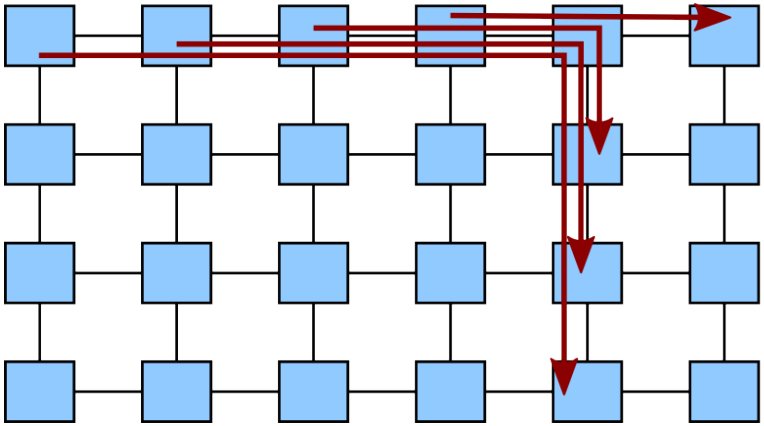
3) Táblázat alapú útvonalválasztás

- Kapcsolók táblázatot tartanak karban:
  - Potenciális címzetthez továbbküldési irányt tartalmaz
- Ha nagy a hálózat, nagy a táblázat:
  - Sok memóriát igényel
  - Sokáig tart benne a keresés

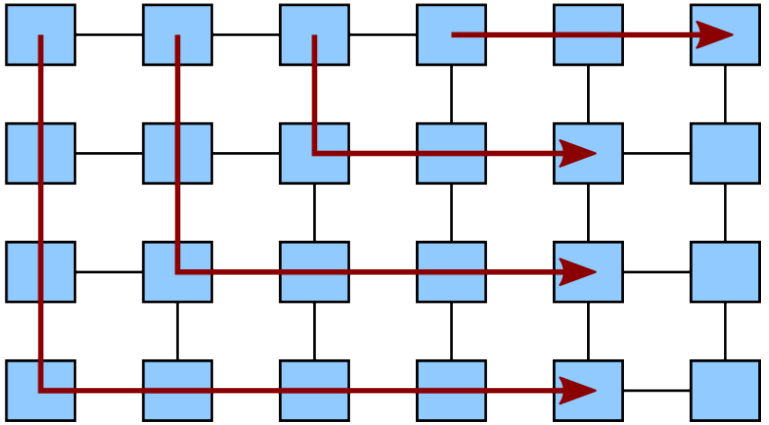
Dinamikus útvonalválasztás

- Alkalmazkodik a hálózat aktuális viszonyaihoz:
  - Elkerüli a terhelt részeket
  - Kikerüli a meghibásodott összeköttetéseket
- Kiszámíthatatlanabb lesz az üzenetküldések ideje
- Jó megvalósítás esetén javulhat az átviteli sebesség és a késleltetés

Statikus útvonalválasztással



Dinamikus útvonalválasztással



Konklúzió

- Az explicit párhuzamosság formái:
  - Több szálat hardveresen támogató processzorok
  - Multiprocesszoros rendszerek
- Tanultuk:
  - Több szál hardveres támogatásának módjai:
    - Időosztásos: finom ↔ durva felbontású
    - Szimultán
  - Hatékonysági kérdések: Amdahl törvénye
  - Multiprocesszoros rendszerek osztályozása
    - Osztott memóriás ↔ üzenetküldéses
  - Összeköttetések
  - Megoldások útvonalválasztásra