HÁLÓZATI RENDSZEREK
ÉS SZOLGÁLTATÁSOK
TANSZÉK

MŰEGYETEM 1782

Budapest,
2022.02.15.

SZÁMÍTÓGÉP ARCHITEKTÚRÁK

Bevezetés, Információfeldolgozási modellek

Horváth Gábor, Belső Zoltán

BME Hálózati Rendszerek és Szolgáltatások Tanszék
ghorvath@hit.bme.hu, belso@hit.bme.hu

ADMINISZTRATÍV TUDNIVALÓK

- Kommunikációs csatornák:
  - Tárgy weblap: http://www.hit.bme.hu/~ghorvath/szgarch/
  - **Tanszéki** Moodle: https://moodle.hit.bme.hu/course/view.php?id=375
  - Teams
- Oktatási segédanyagok a tárgy weblapján:
  - Előadás fóliák
  - Gyakorlat feladatlapok
  - Jegyzet
  - Videók
  - Online gyakorló app

ADMINISZTRATÍV TUDNIVALÓK

- Követelmények
  - **Számonkérések**
    - ZH: 9. hét, hétfő-kedd 18-20, HSZK + Moodle
    - Lesz próba ZH
    - Feladat típusok:
      - Igaz – hamis – passz
      - “Összekötös”
      - Rövid válaszok
      - Numerikus példák
    - Sikeresség: 40%
    - Egy szint fölött beleszámít a vizsgába
  - **Gyakorlat jelenlét**
    - Min. 70% → 9 jelenlét kötelező
- IMSc
  - plusz tananyag

MIRE JÓ EZ A TÁRGY?

- Megértjük, hogy
  - ...hogy épül föl egy modern számítógép
  - ...hogy épül föl egy modern processzor
- Mire jó ez nekünk:
  - Aki ismeri a hardvert,
    - a számítógépet nem fekete doboznak tekinti
    - hatékonyabb programot ír
    - ismeri a kompromisszumokat

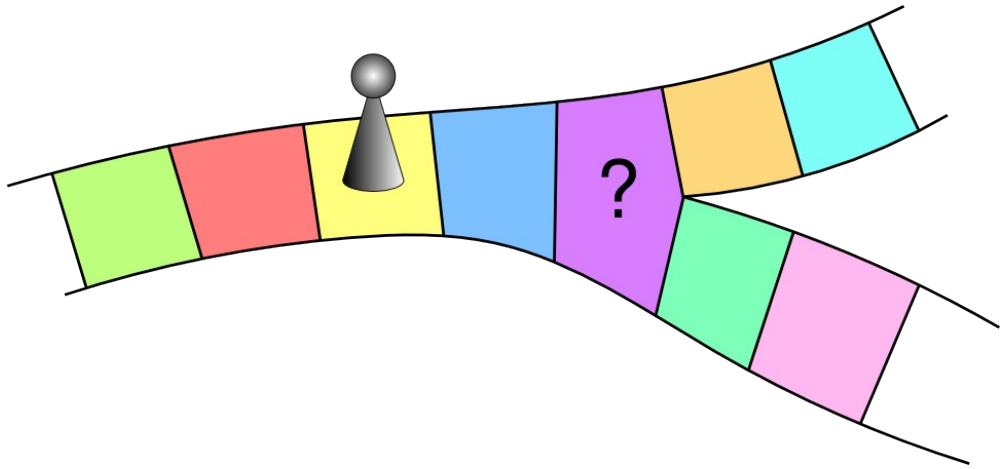
Információfeldolgozási modellek

INFORMÁCIÓFELDOLGOZÁSI MODELLEK

- Hogyan fogalmazzuk meg a problémát egy számítógépnek?
  - Vezérlésáramlásos modell
  - Adatáramlásos modell
  - Igényvezérelt modell

VEZÉRLÉSÁRAMLÁSOS MODELL

- Utasítások sorrendje: vezérlő token szabja meg

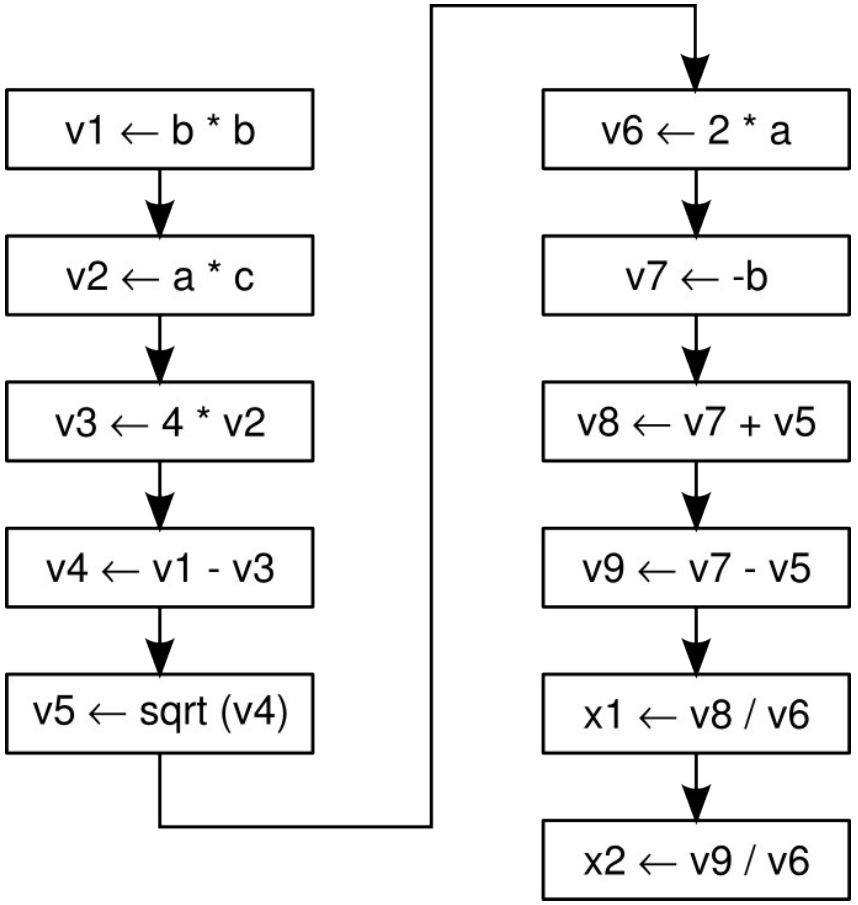


- Vezérlő token mozgása:
  - Implicit (egyet lép előre)
  - Explicit (goto, szubrutin, return, stb.)
- Utasítások közötti adatcsere: közös memória
- Program leírása: folyamatábra

VEZÉRLÉSÁRAMLÁSOS MODELL

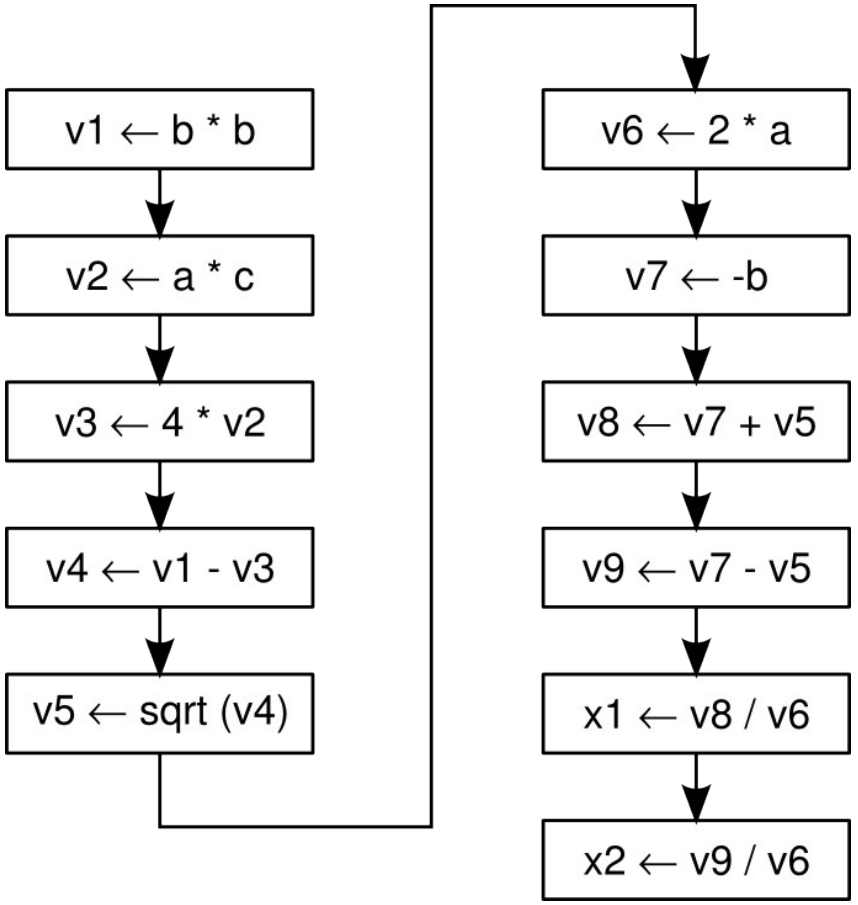
- Példa: másodfokú egyenlet gyökei

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



VEZÉRLÉSÁRAMLÁSOS MODELL

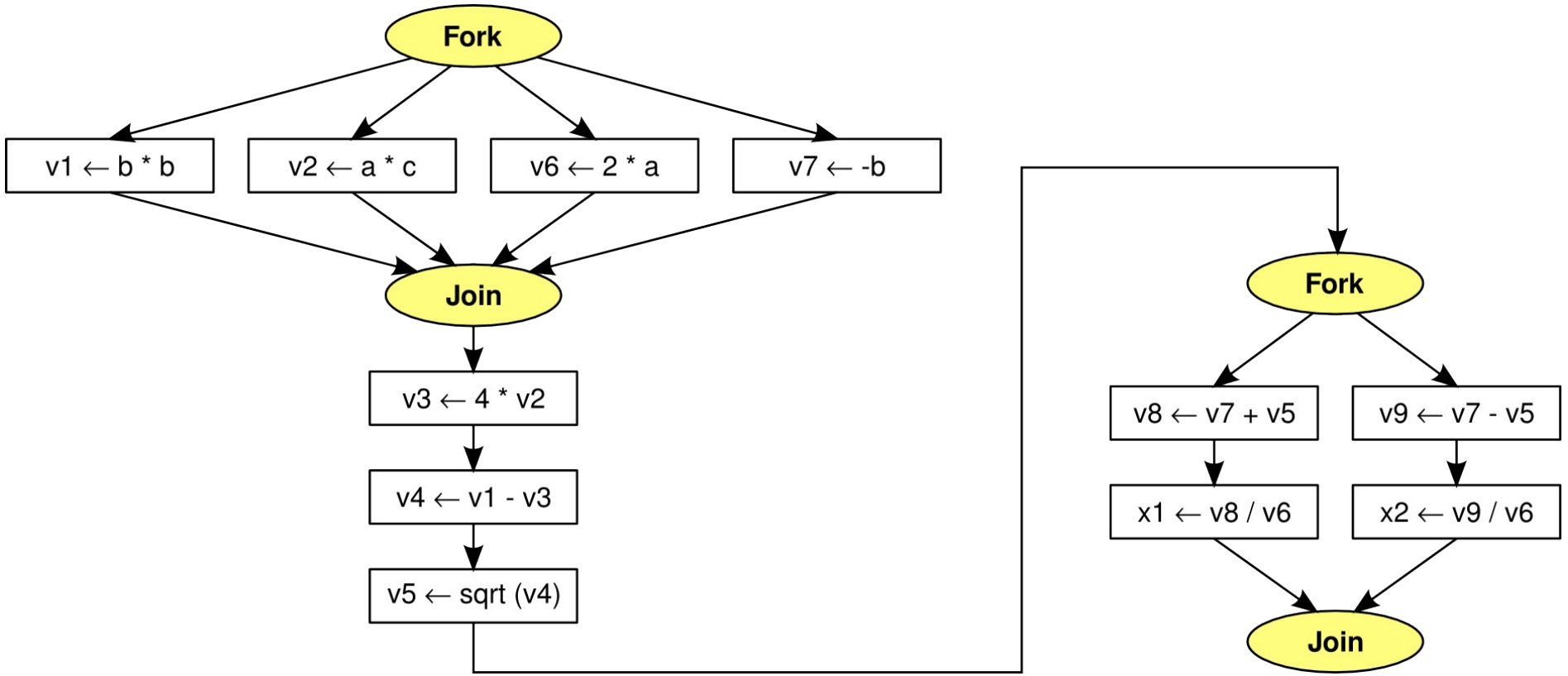
- Párhuzamosság?
  - v1, v2, v6, v7
  - v8, v9
  - x1, x2



VEZÉRLÉSÁRAMLÁSOS MODELL

- Párhuzamosság Fork/Join primitívekkel

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

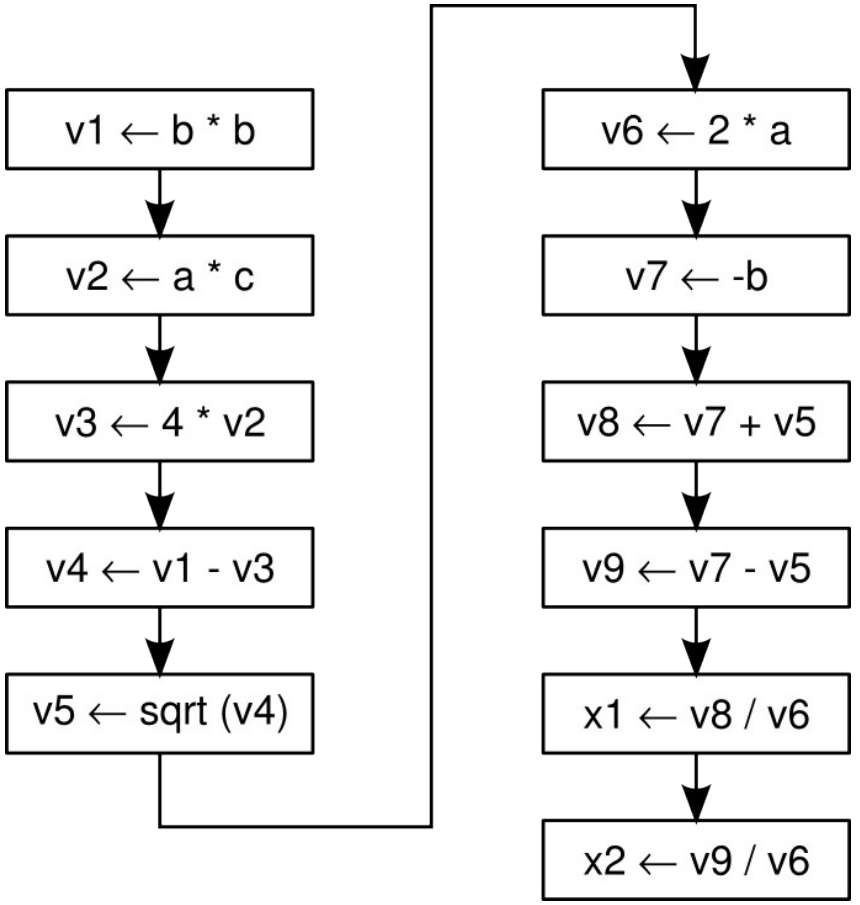


VEZÉRLÉSÁRAMLÁSOS MODELL

- 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!

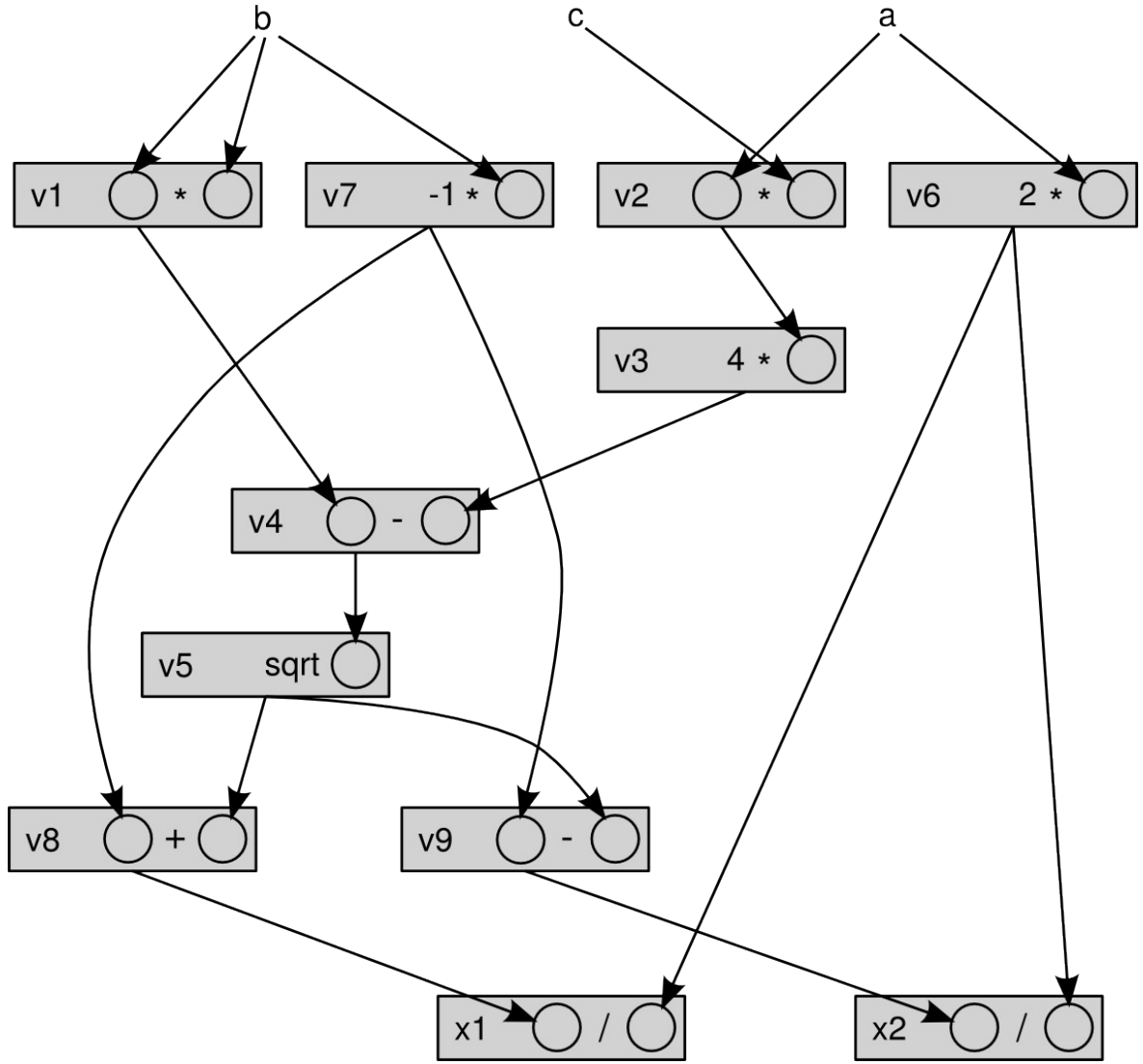


ADATÁRAMLÁSOS MODELL

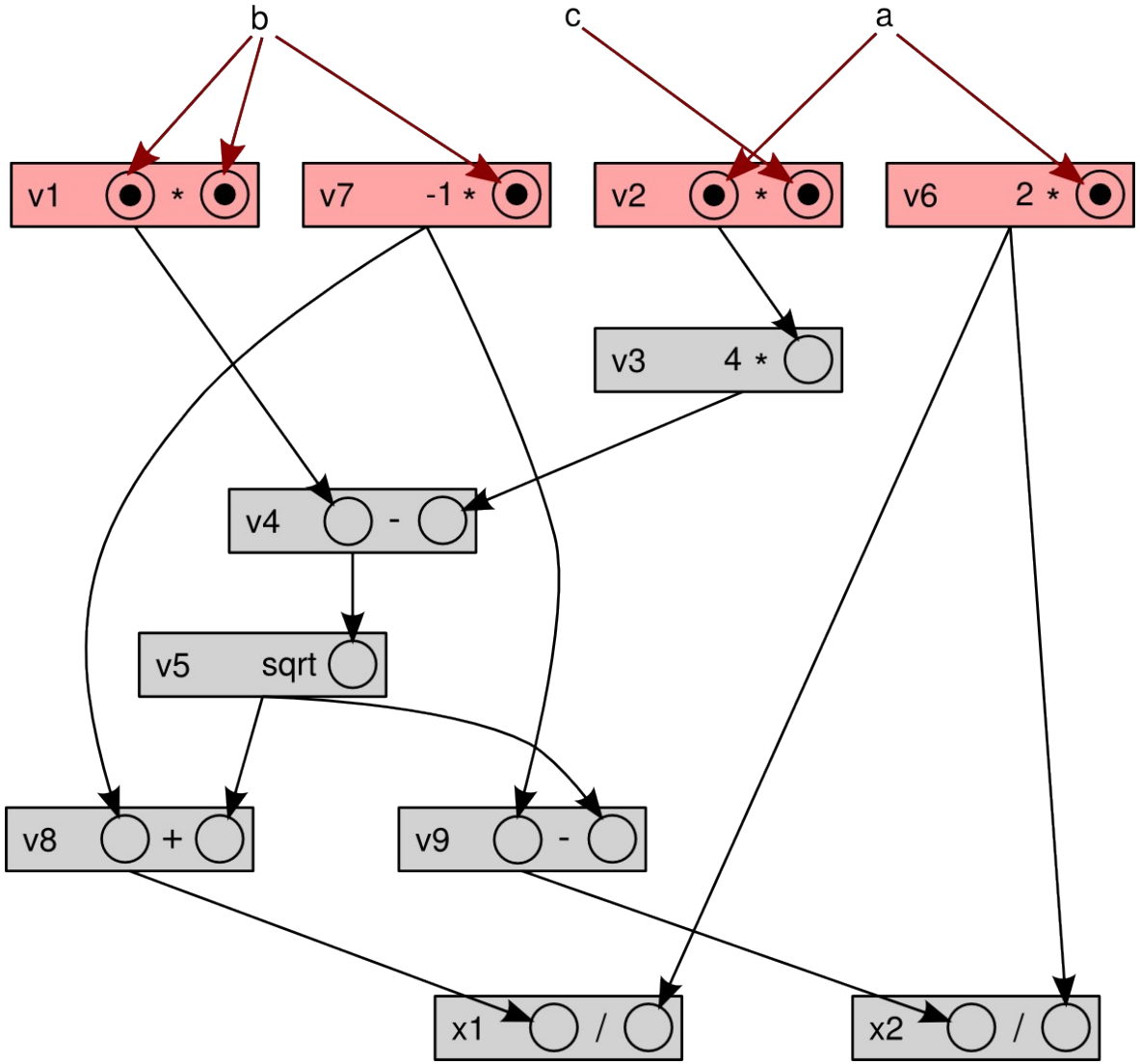
- Nincs vezérlő token
- Nincs közös memória
- Vezérlő token helyett:
  - Utasítások végrehajtása automatikus:
    - ha minden operandus megvan
- Közös memória helyett:
  - Részeredményeket közvetlenül adják át egymásnak az utasítások
- Program leírása: egy precedenciagráf

- Példa: másodfokú egyenlet gyökei

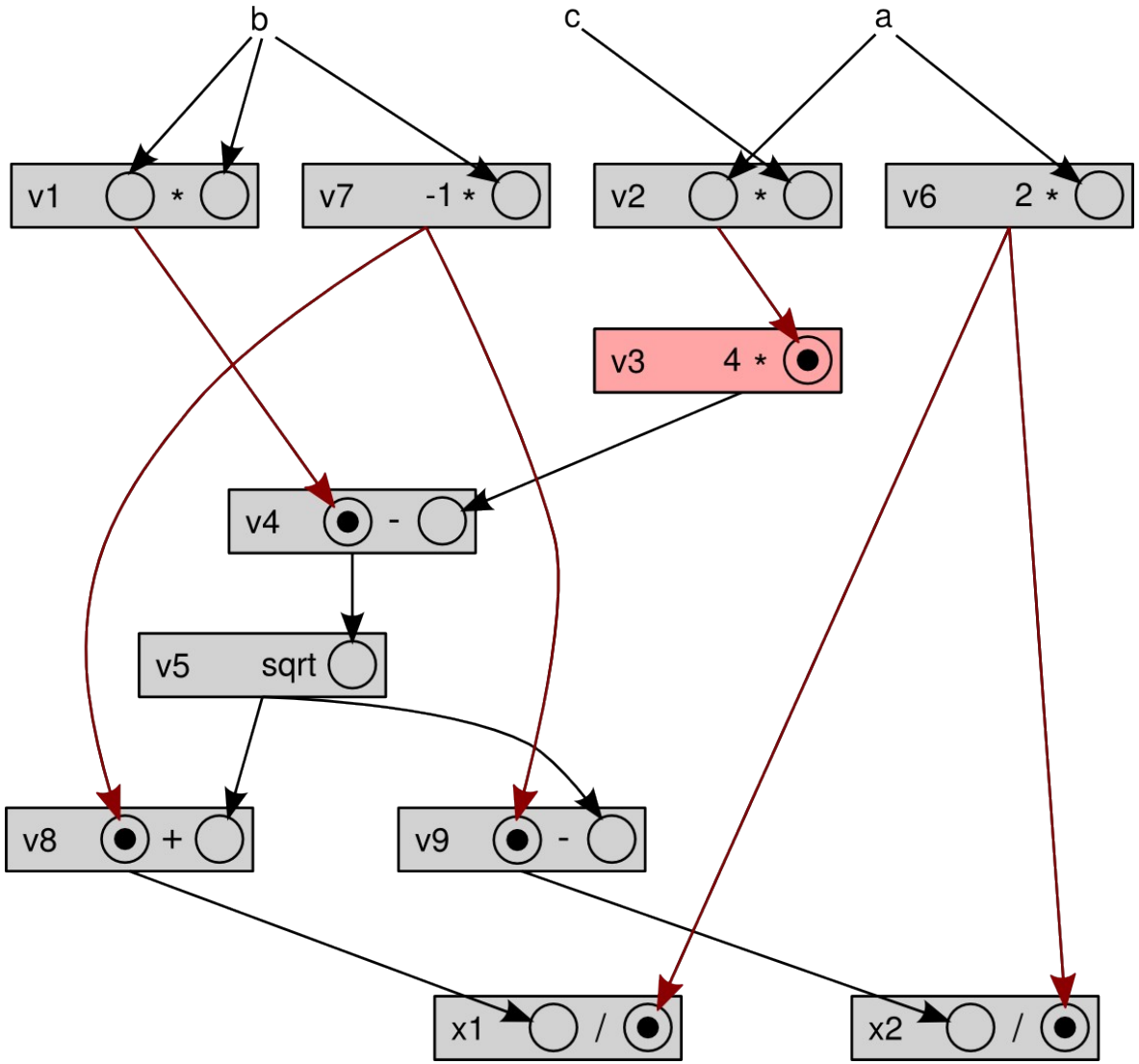
ADATÁRAMLÁSOS MODELL



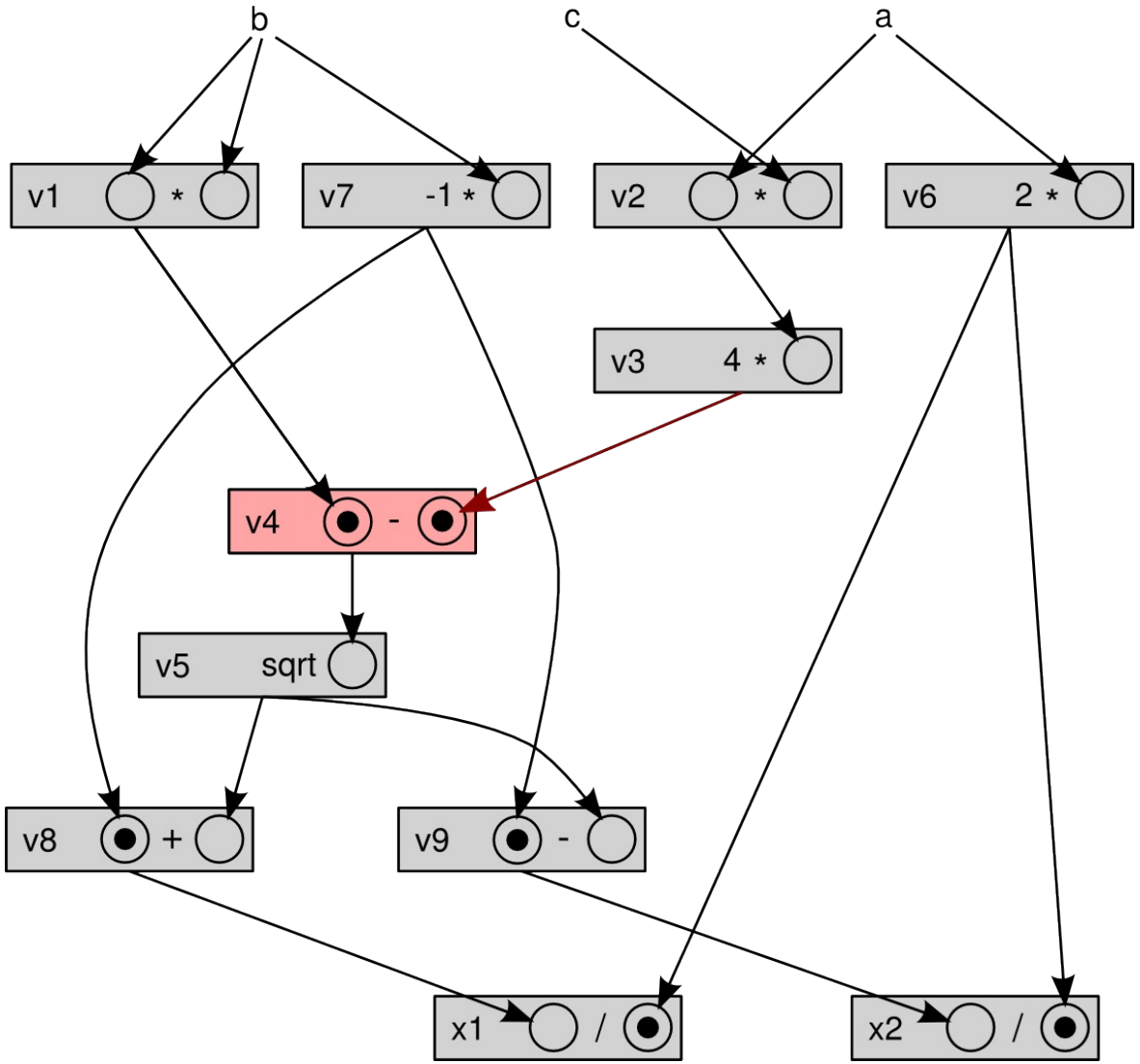
ADATÁRAMLÁSOS MODELL



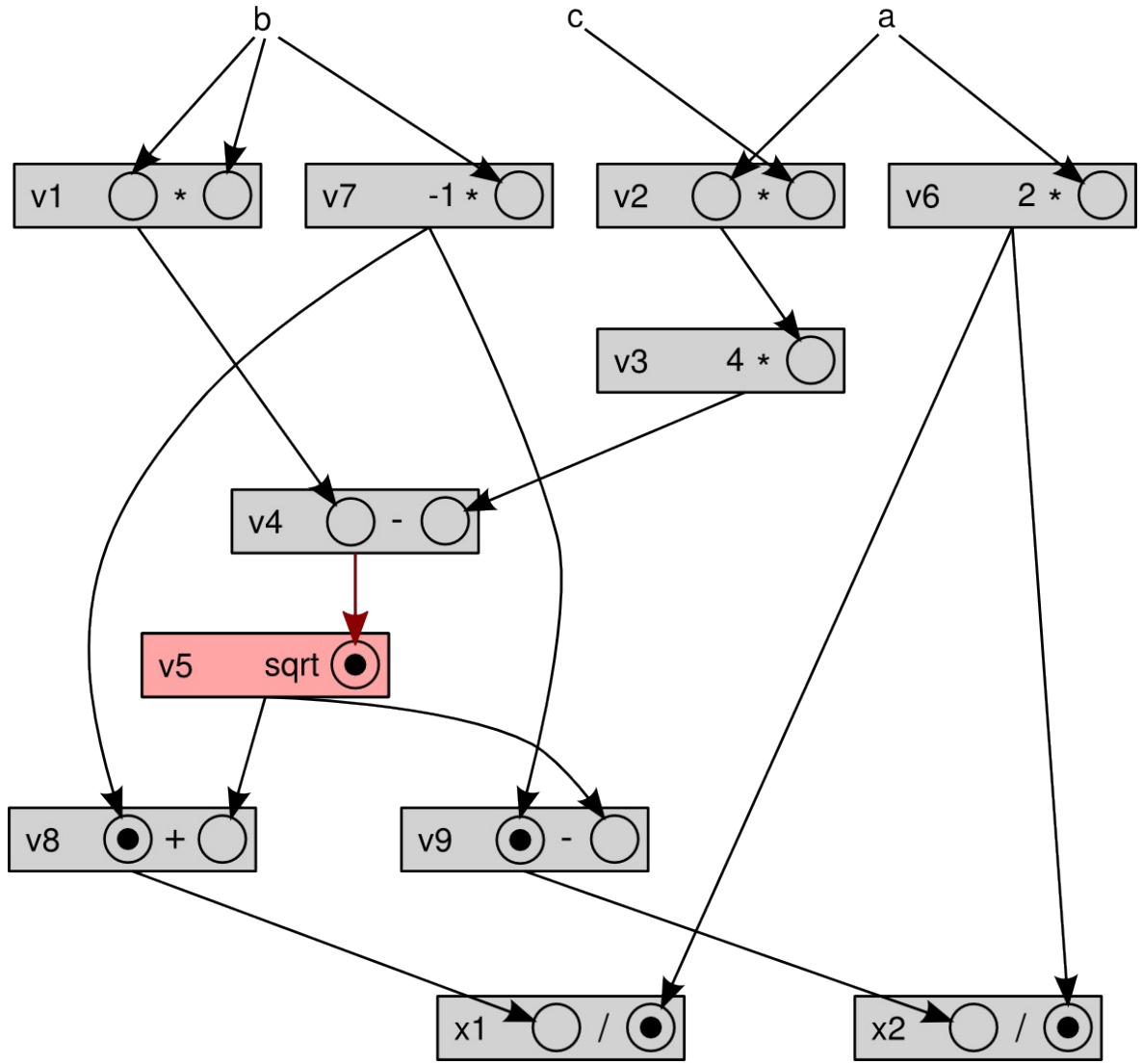
ADATÁRAMLÁSOS MODELL



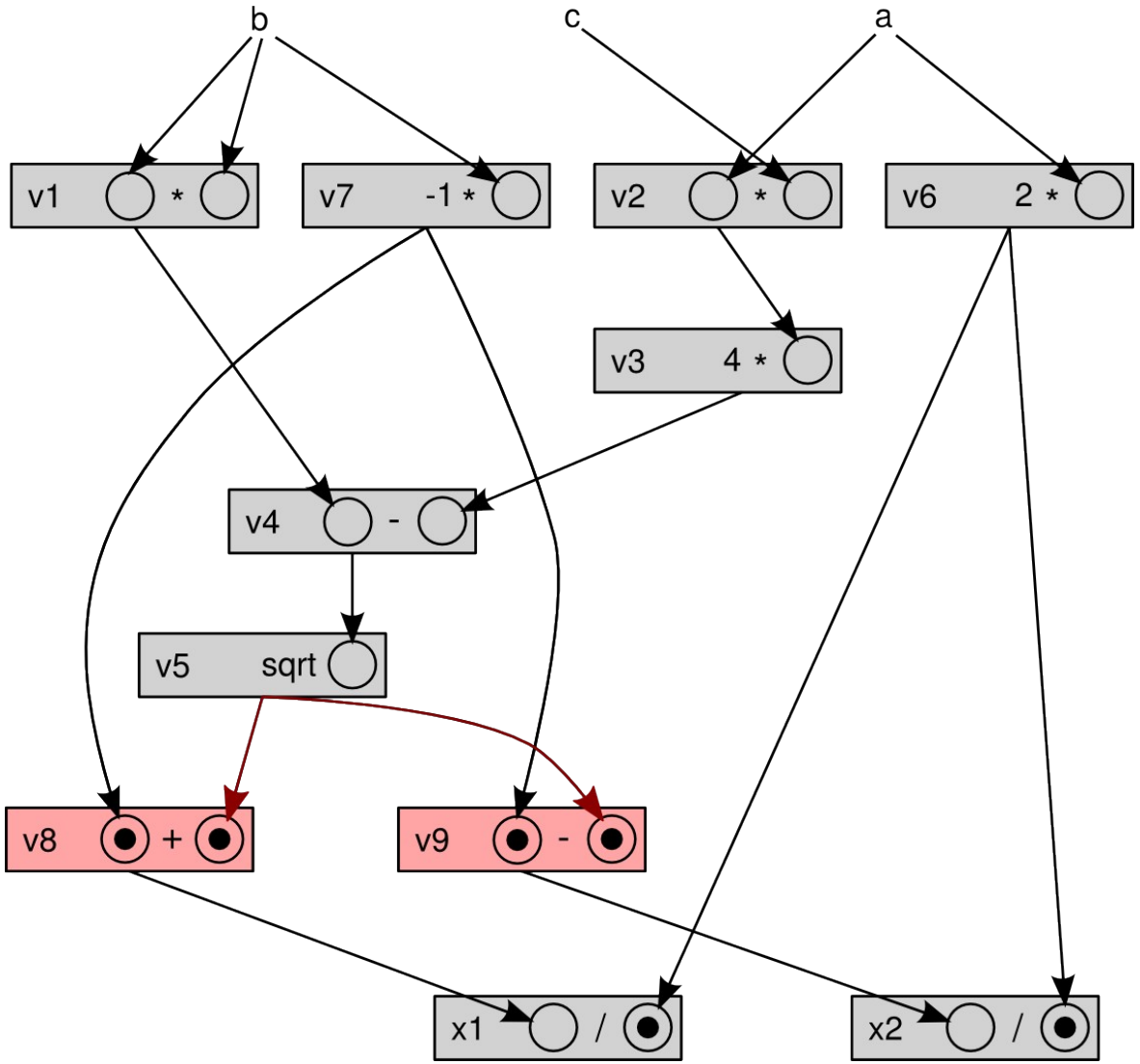
ADATÁRAMLÁSOS MODELL



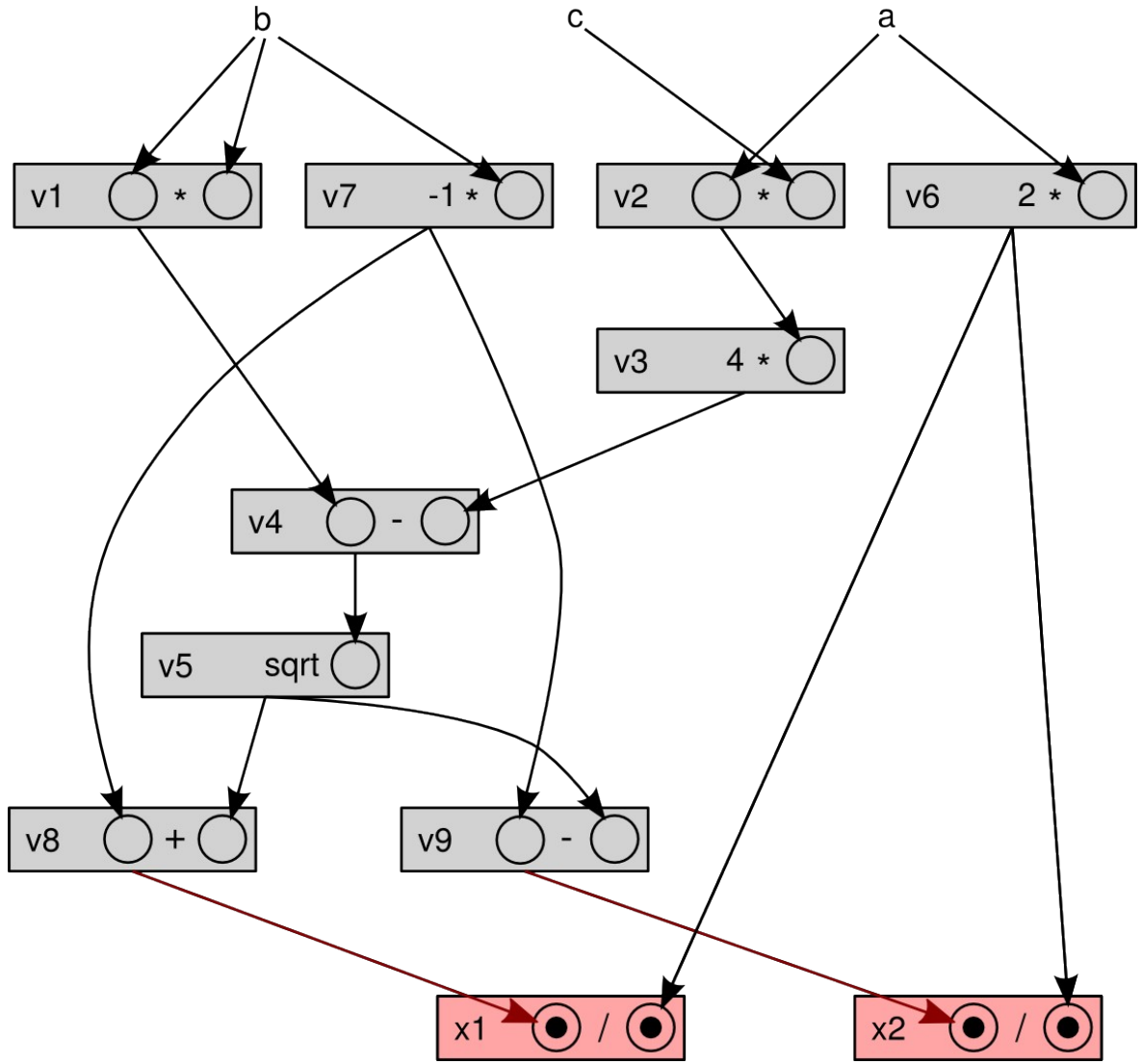
ADATÁRAMLÁSOS MODELL



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
    - Adatfeldolgozó rendszerek (pl. Apache Pig)

IGÉNYVEZÉRELT MODELL

- Utasítások végrehajtása automatikus:
  - ha szükség van az eredményre
- Összehasonlítva, utasítások végrehajtási ideje:
  - Vezérlésáramlásos: ha odaér a token

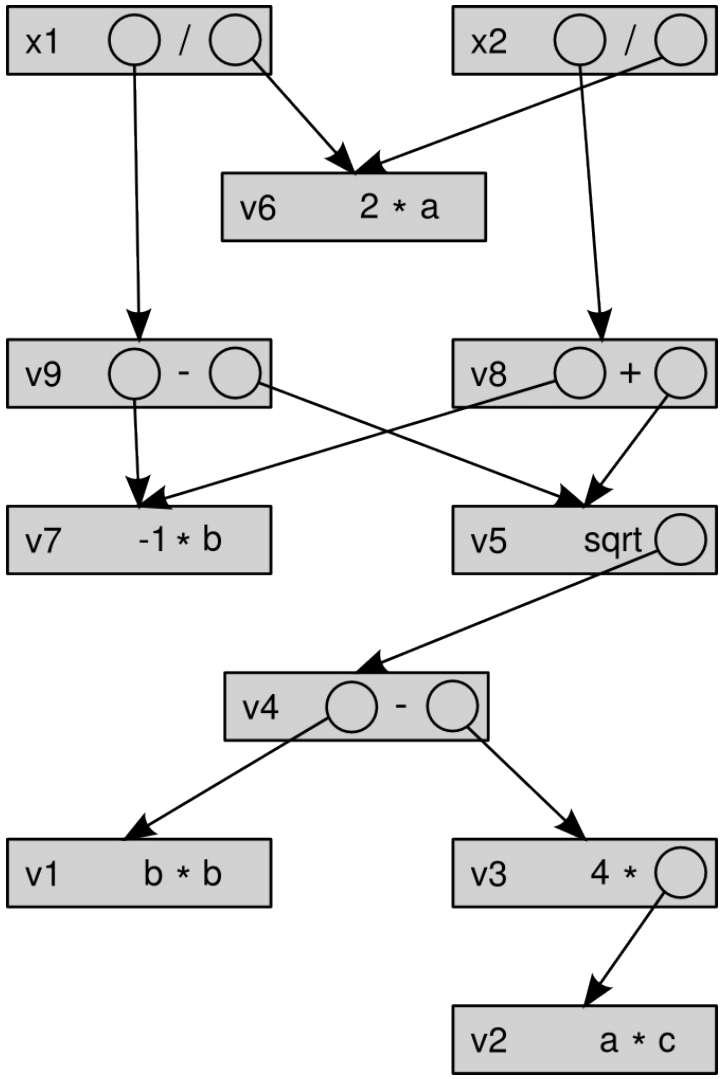
    szájbarágós
  - Adatáramlásos: ha minden operandus megvan

    szorgalmas
  - Igényvezérelt: ha szükség van az eredményre

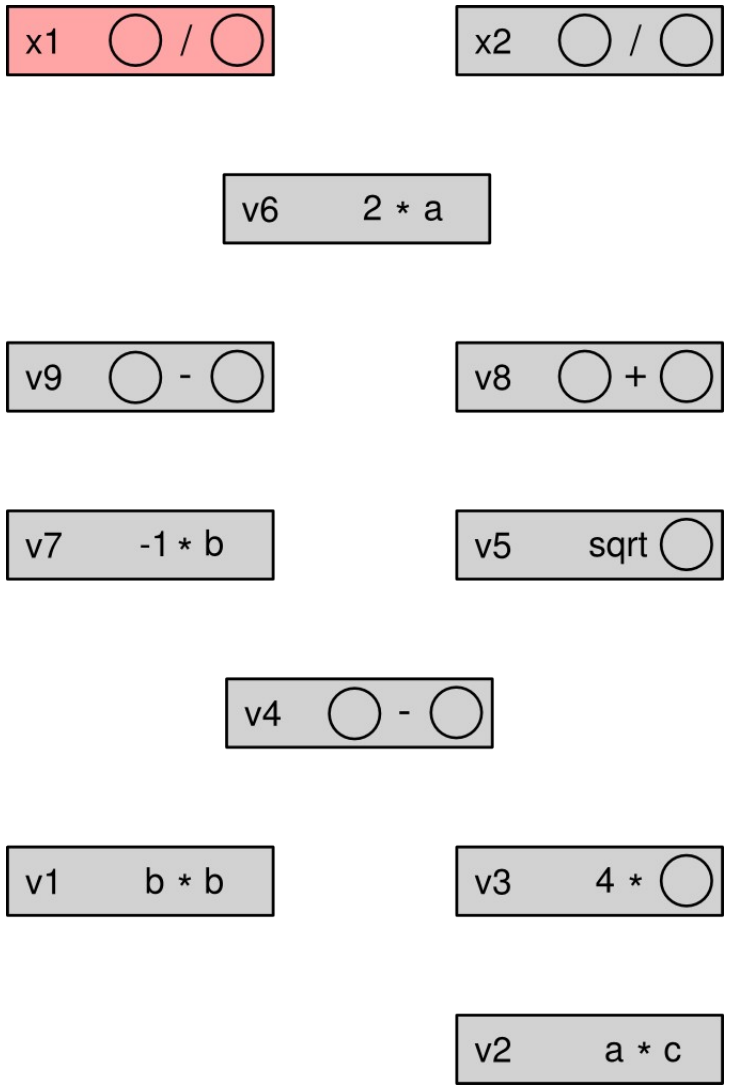
    lusta

- Példa: másodfokú egyenlet gyökei

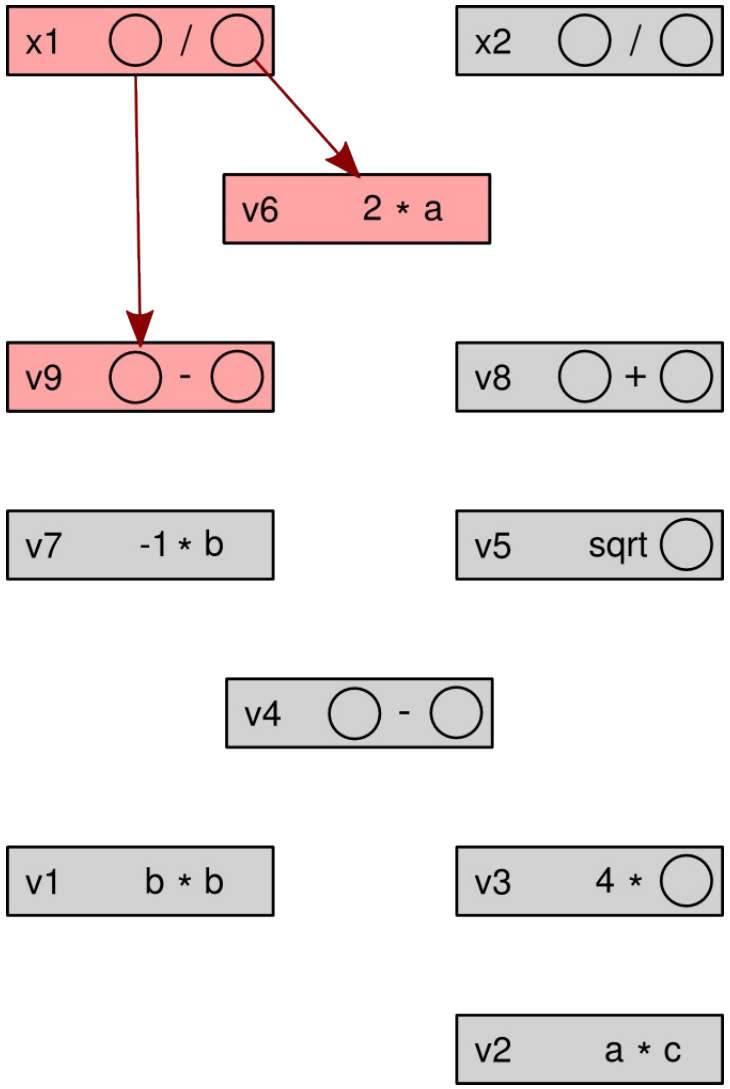
IGÉNYVEZÉRELT MODELL



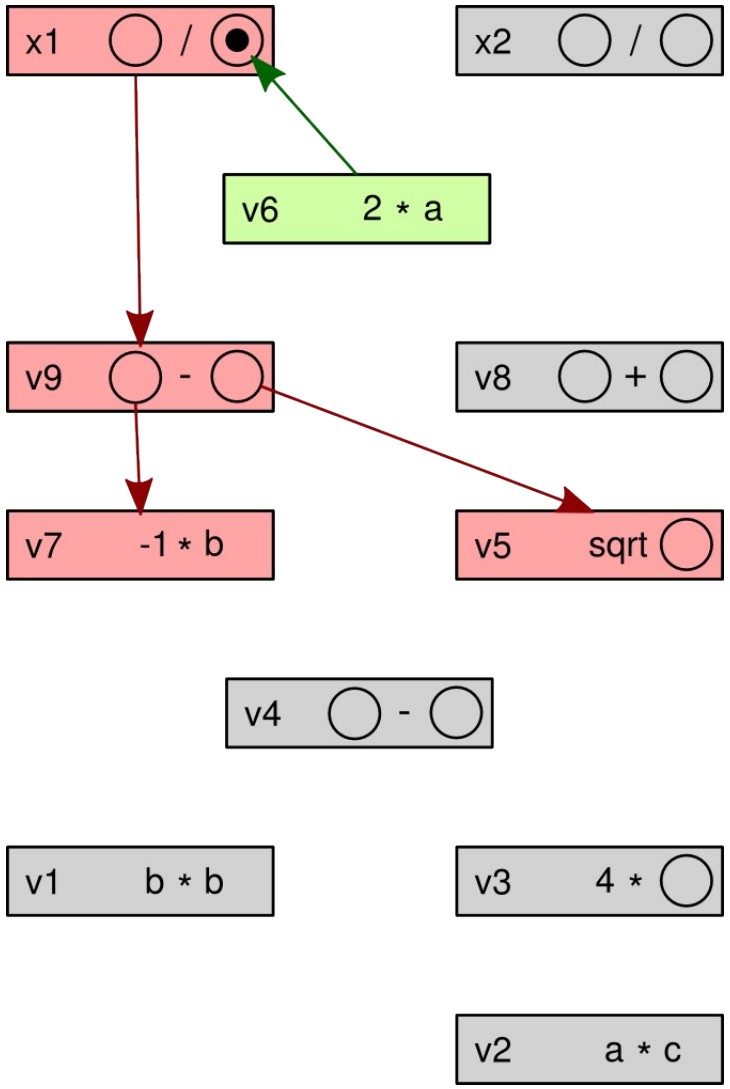
IGÉNYVEZÉRELT MODELL



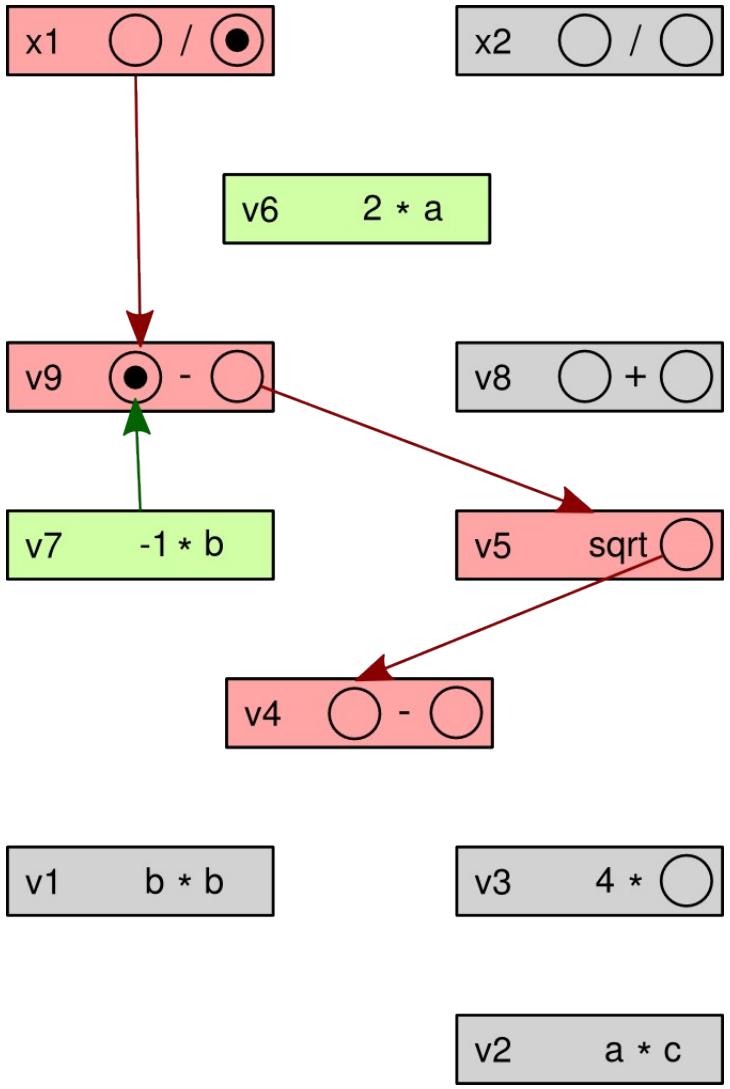
IGÉNYVEZÉRELT MODELL



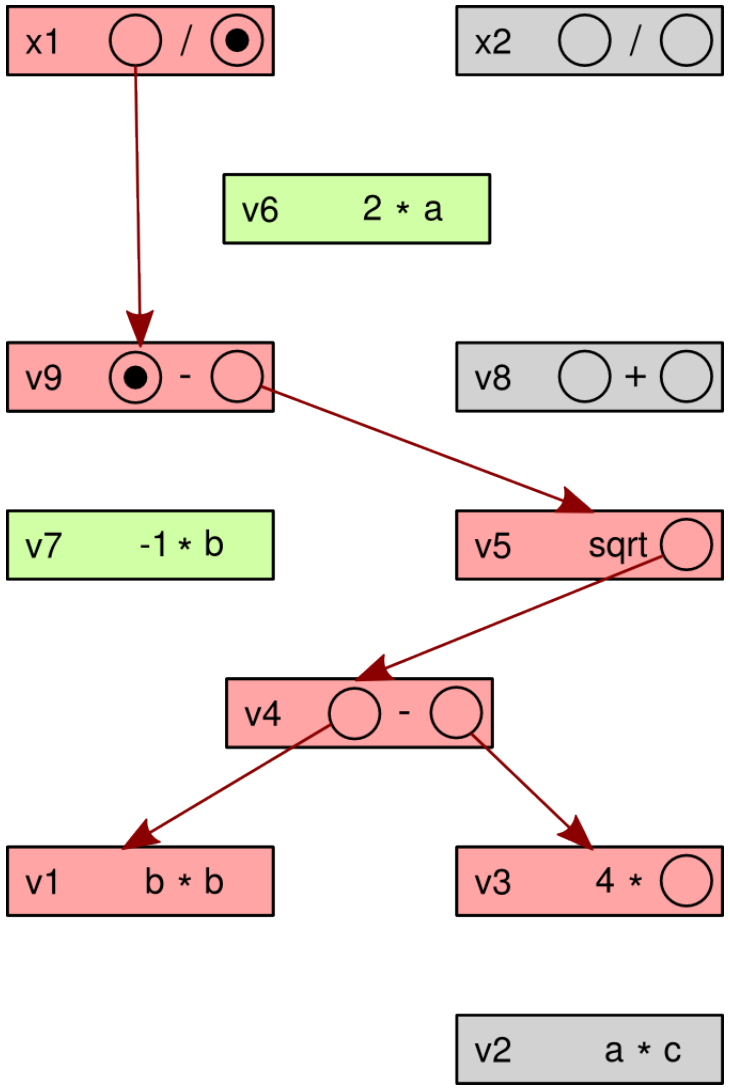
IGÉNYVEZÉRELT MODELL



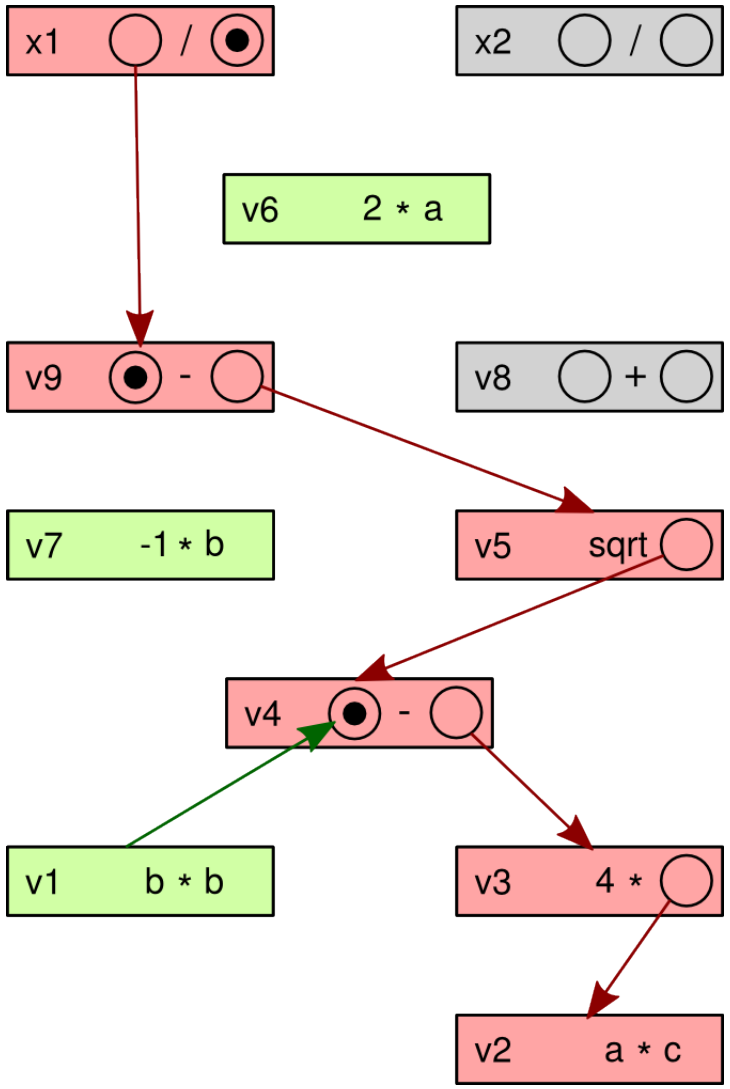
IGÉNYVEZÉRELT MODELL



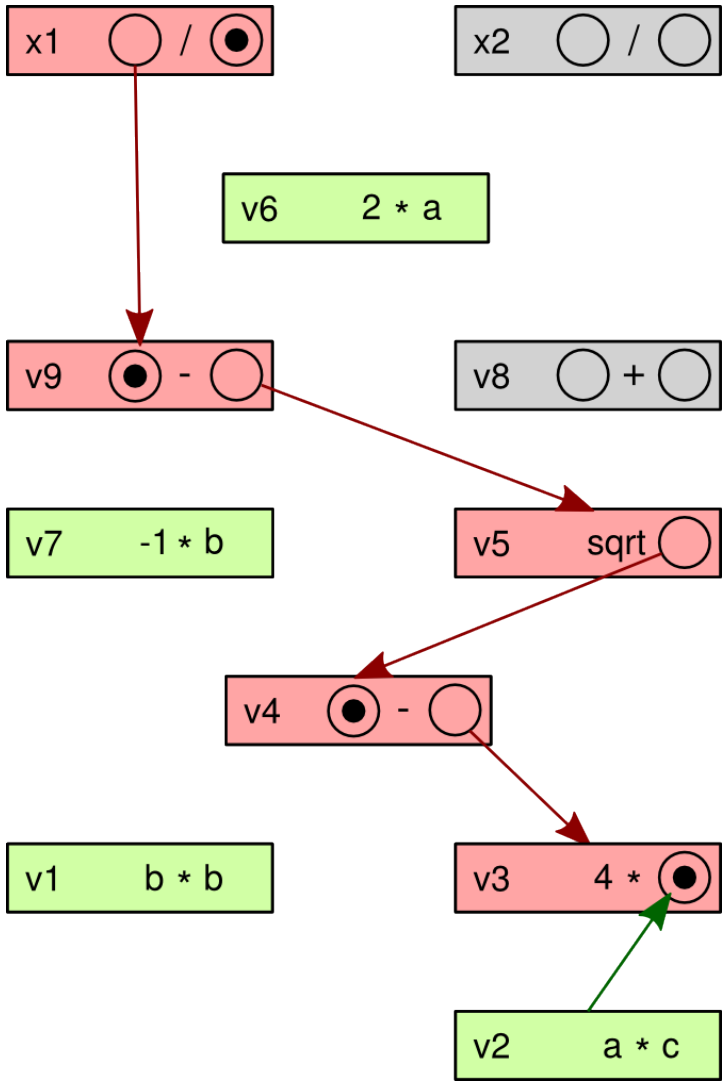
IGÉNYVEZÉRELT MODELL



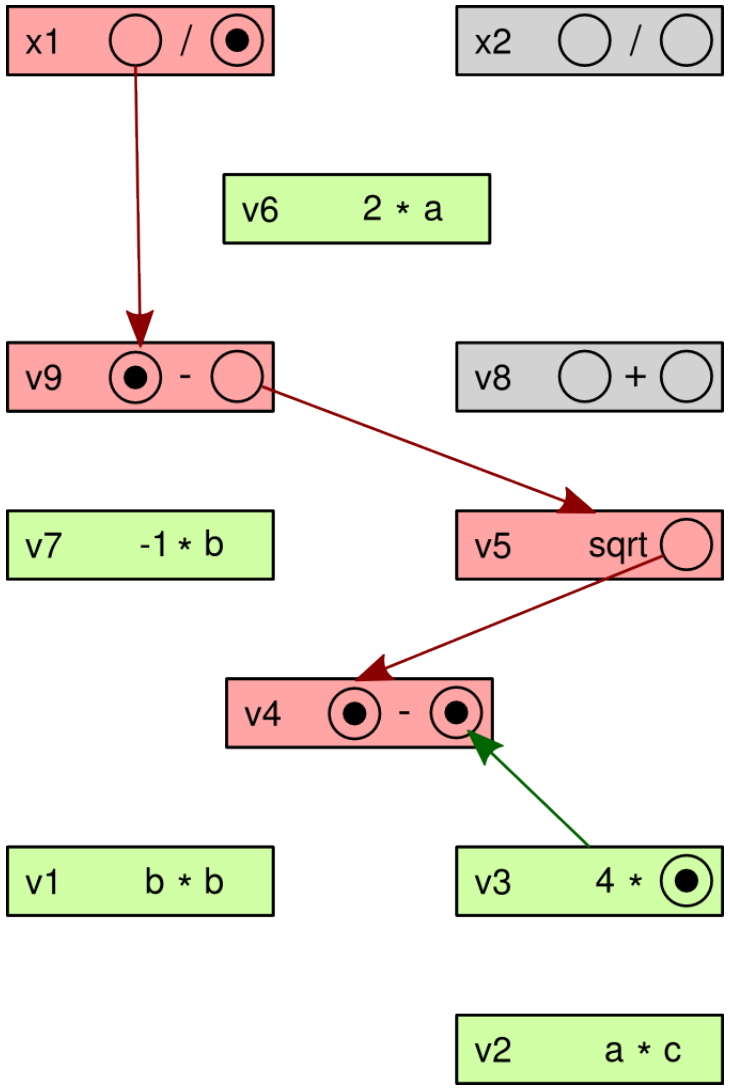
IGÉNYVEZÉRELT MODELL



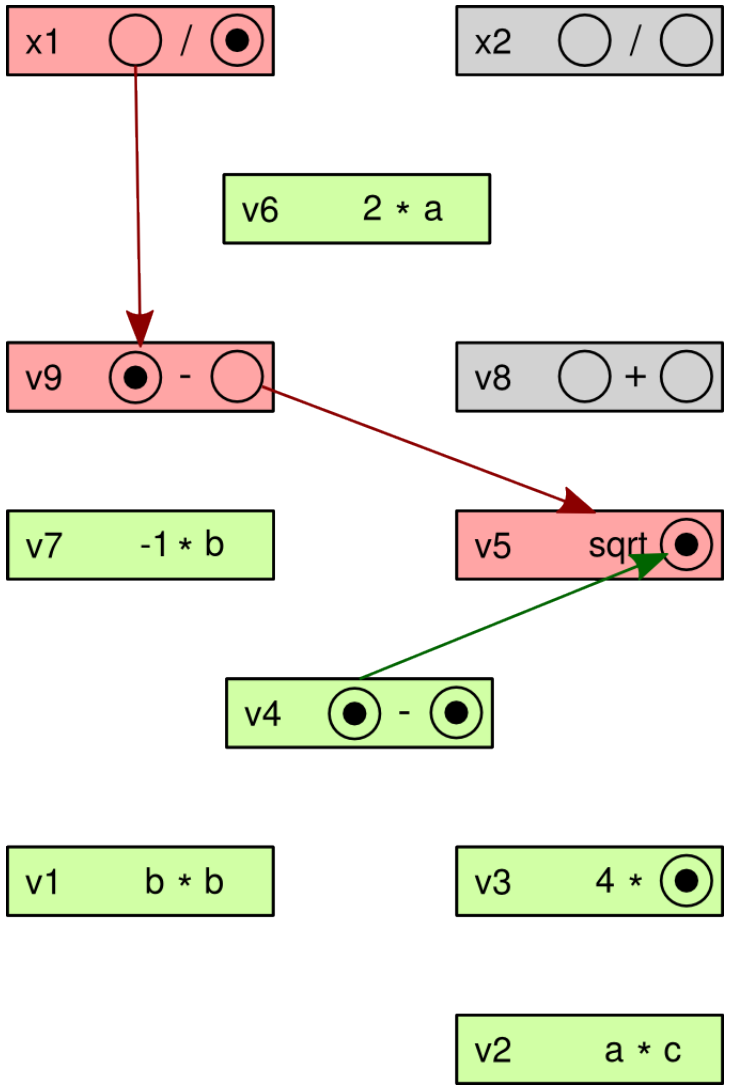
IGÉNYVEZÉRELT MODELL



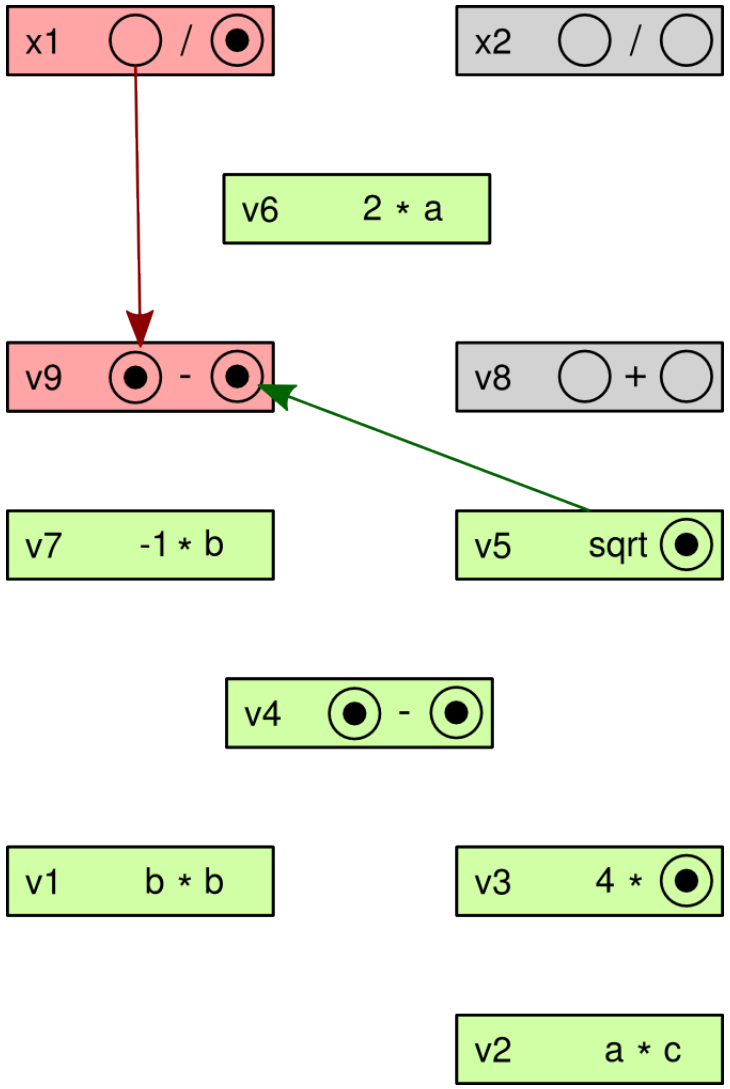
IGÉNYVEZÉRELT MODELL



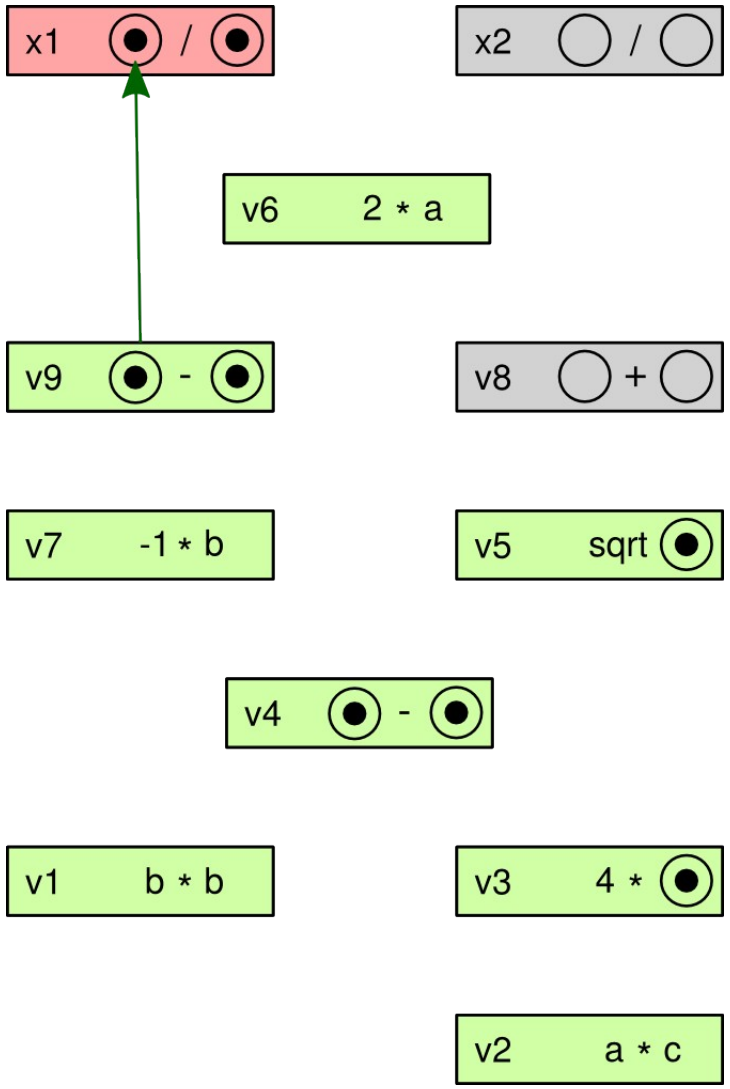
IGÉNYVEZÉRELT MODELL



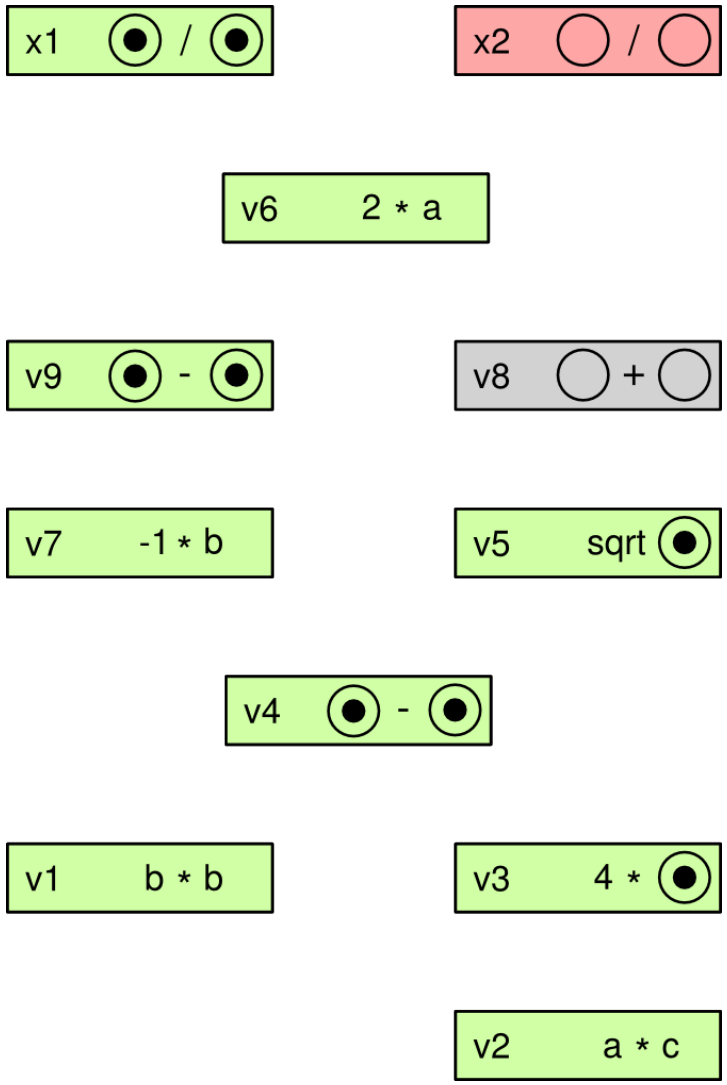
IGÉNYVEZÉRELT MODELL



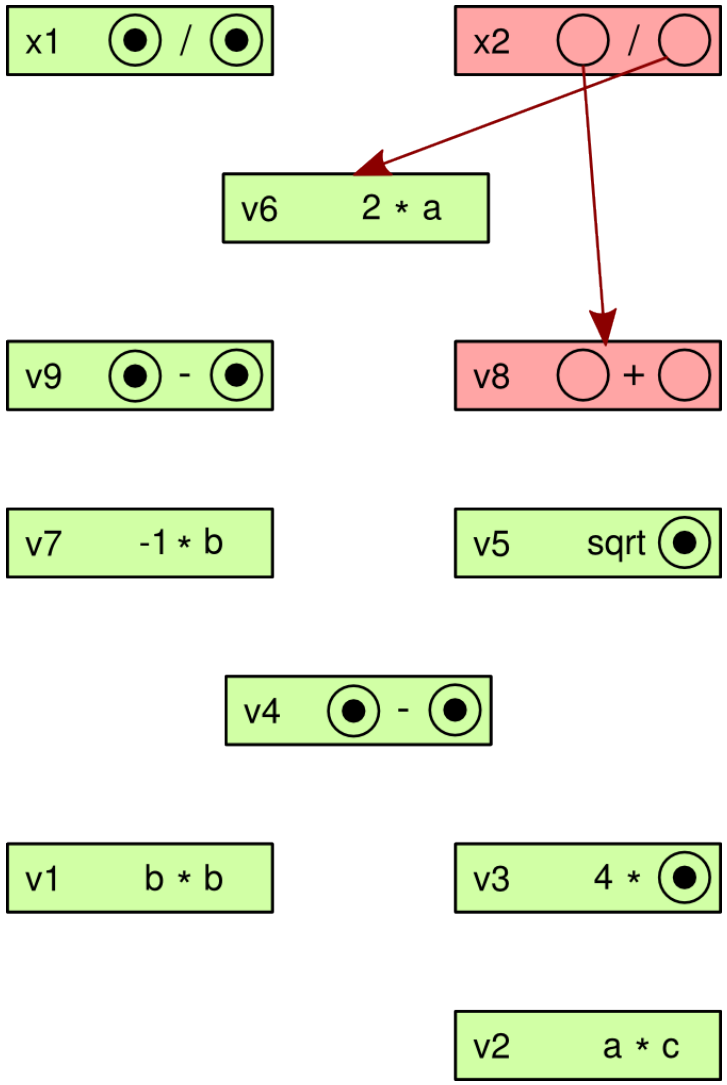
IGÉNYVEZÉRELT MODELL



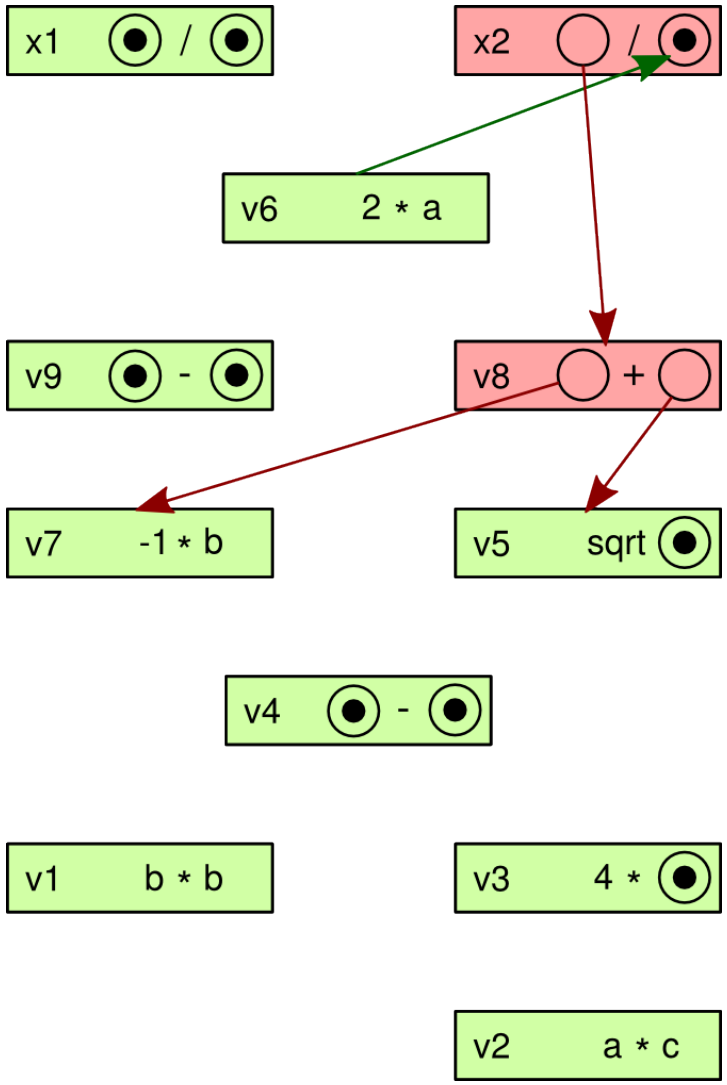
IGÉNYVEZÉRELT MODELL



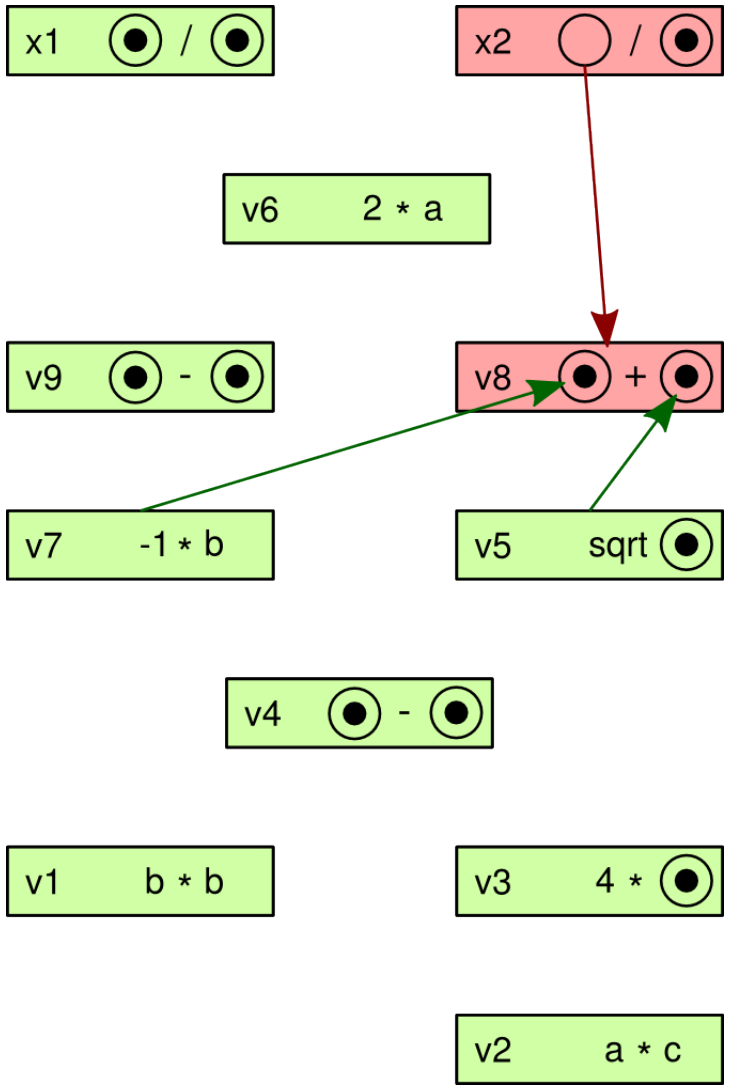
IGÉNYVEZÉRELT MODELL



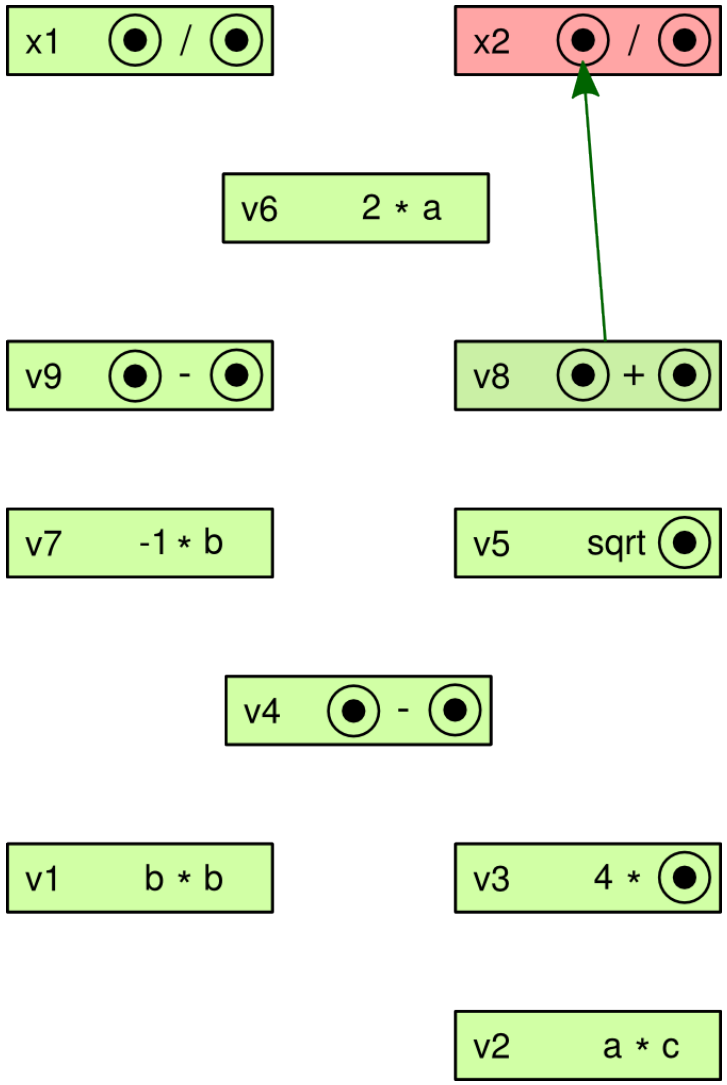
IGÉNYVEZÉRELT MODELL



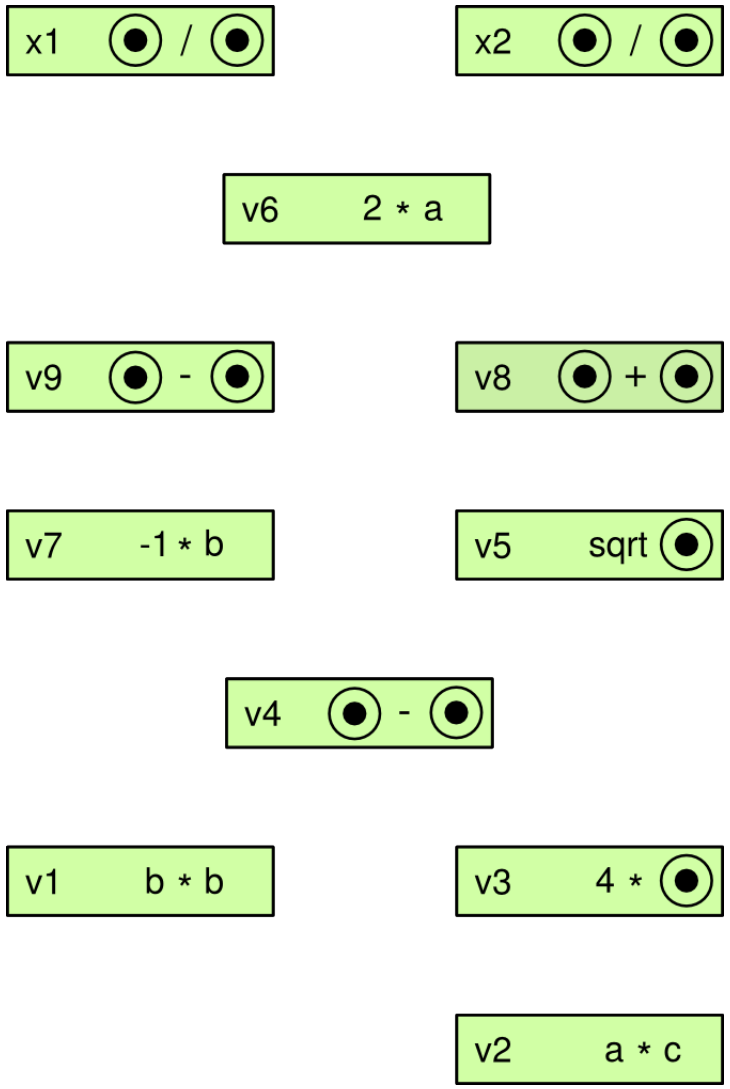
IGÉNYVEZÉRELT MODELL



IGÉNYVEZÉRELT MODELL



IGÉNYVEZÉRELT MODELL



IGÉNYVEZÉRELT MODELL

- Értékelés
  - Párhuzamosságot automatikusan felderíti
  - Tisztán igényvezérelt számítógép nem terjedt el
  - De az elv igen:
    - Funkcionális programnyelvek
    - Google map-reduce API
      - MAP művelet: részfeladatokra bontás
      - REDUCE művelet: részeredmények összegzése
      - Működés:
        - Elosztott rendszer sok számítógéppel
        - Részfeladatokra bont + részfeladatokat delegálja + eredményeket összegzi és visszaadja → mindezt rekurzívan
        - Igényvezérelt modell!

INFORMÁCIÓFELDOLGOZÁSI MODELLEK

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

Főbb vezérlésáramlásos architektúrák

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

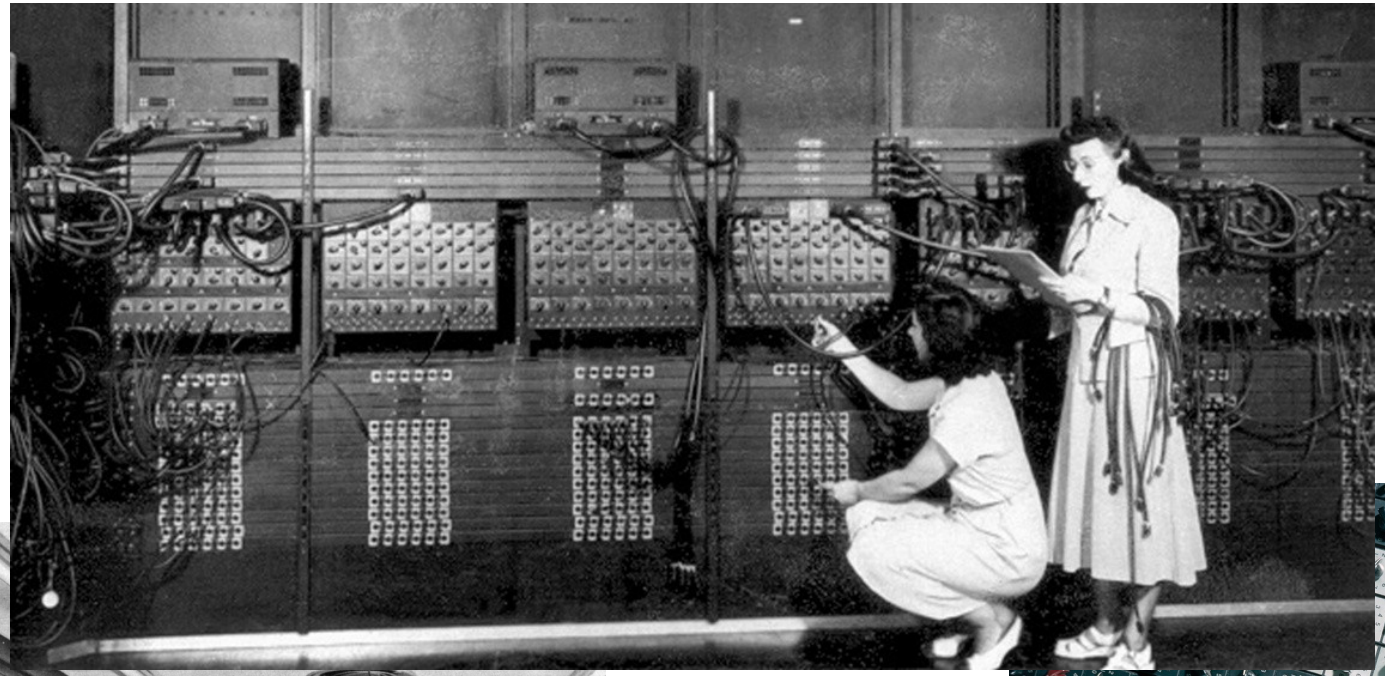
- Újdonság:

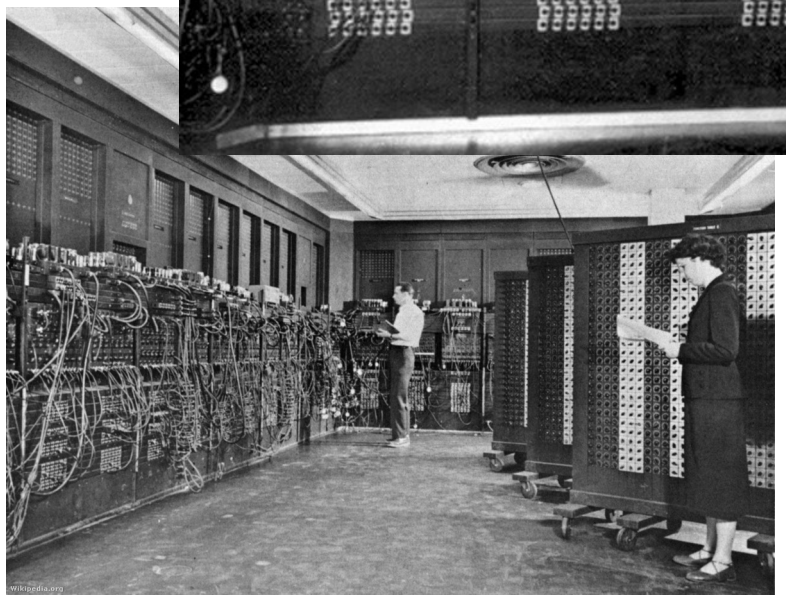
Az utasítások az adatokkal együtt vannak eltárolva a memóriában

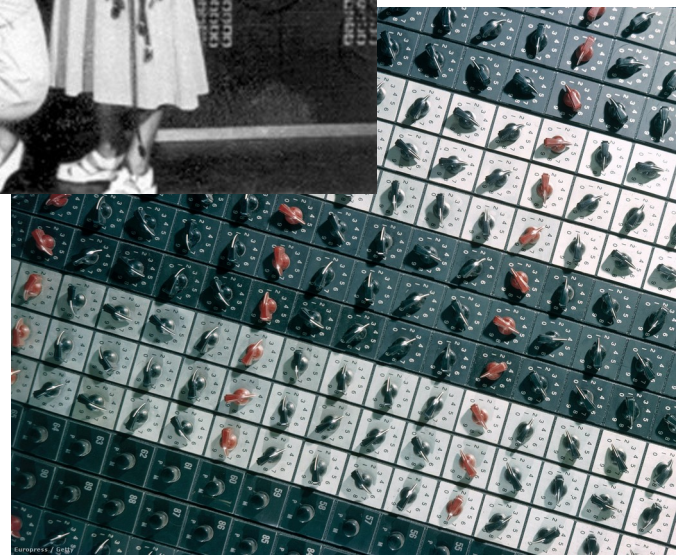
  - Korábban: programozás = kapcsolósor beállítás, újrahuzalozás
  - Motiváció:
    - A programot így könnyebb megváltoztatni

      (Új program az ENIAC-ra = 3 hét munka!!)
    - Gyorsabban lehet több programot egymás után futtatni
    - Programot generáló programot lehet írni
      - Fordítóprogramok születése

AZ ENIAC

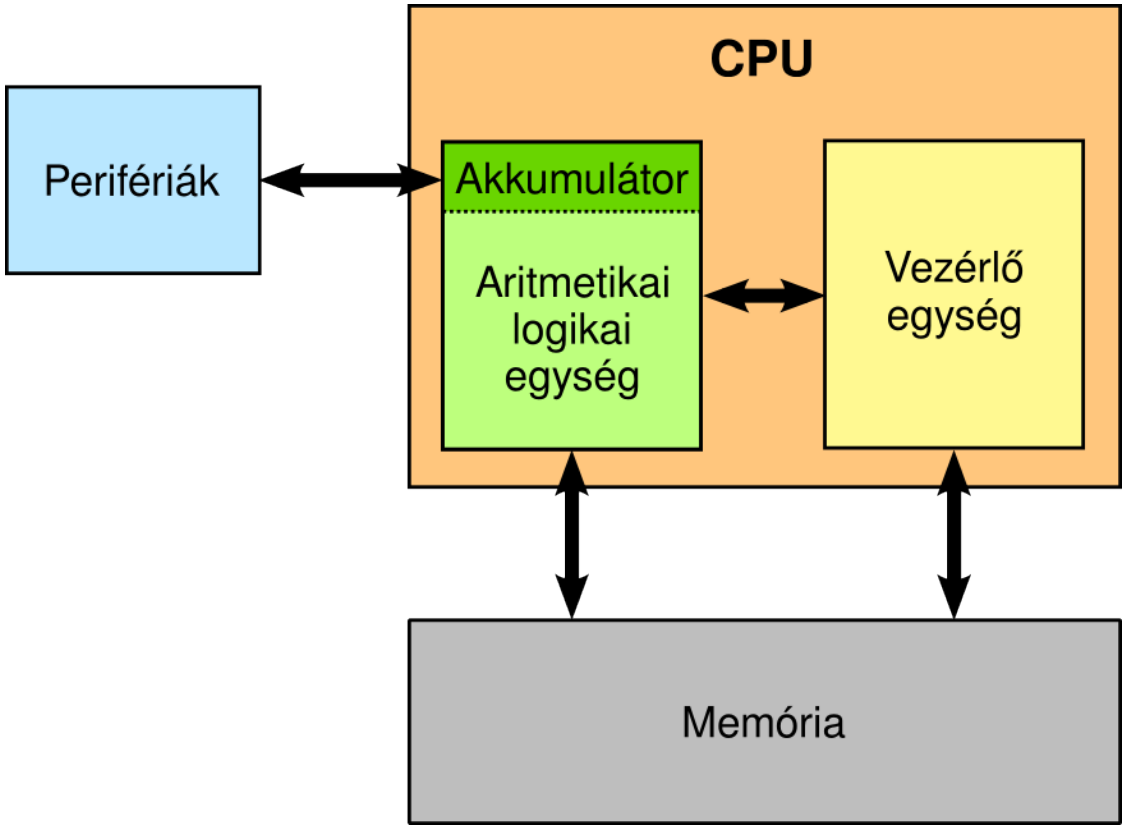






https://index.hu/tech/2016/02/14/70_eves_az_eniac/

A NEUMANN ARCHITEKTÚRA

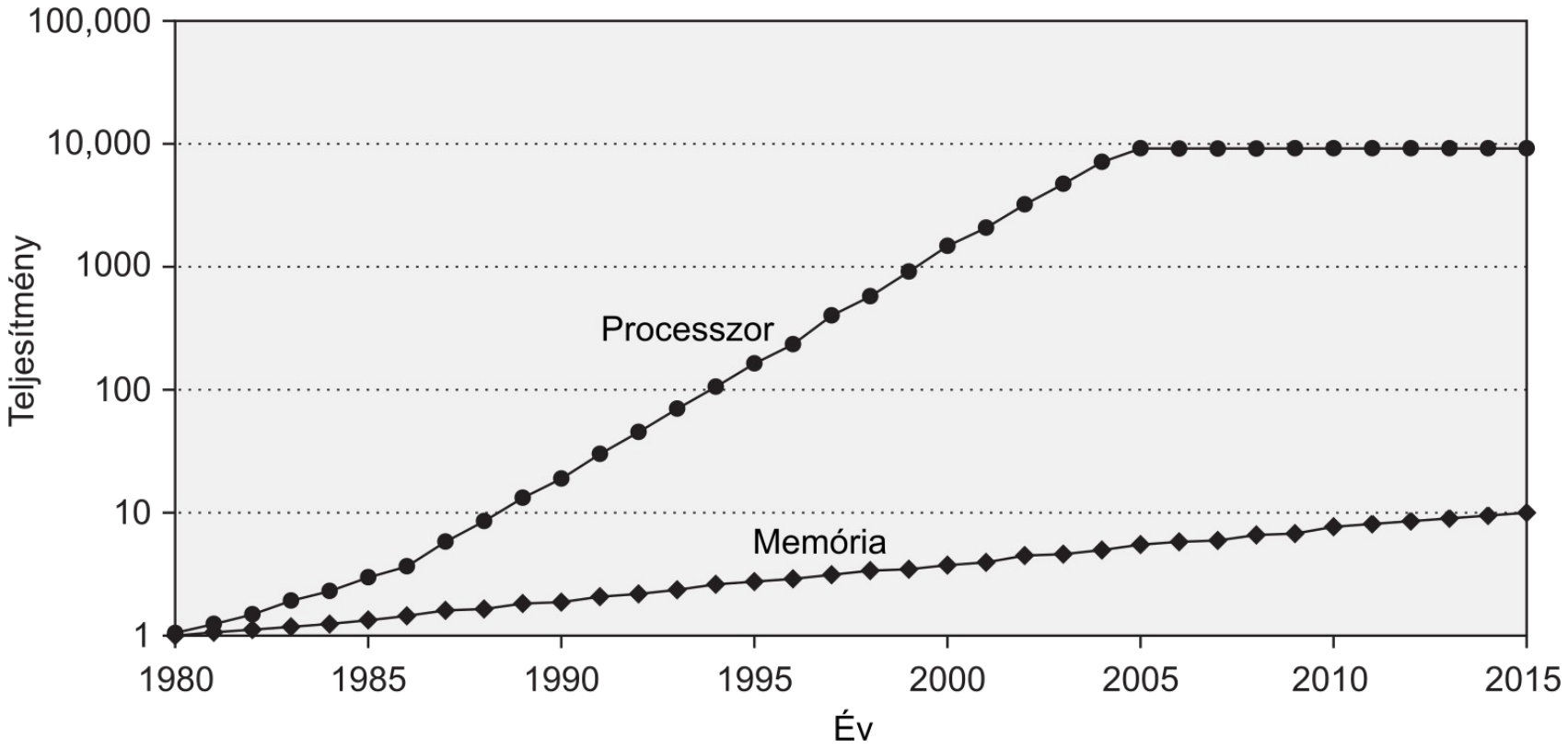


A NEUMANN ARCHITEKTÚRA

- 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!**



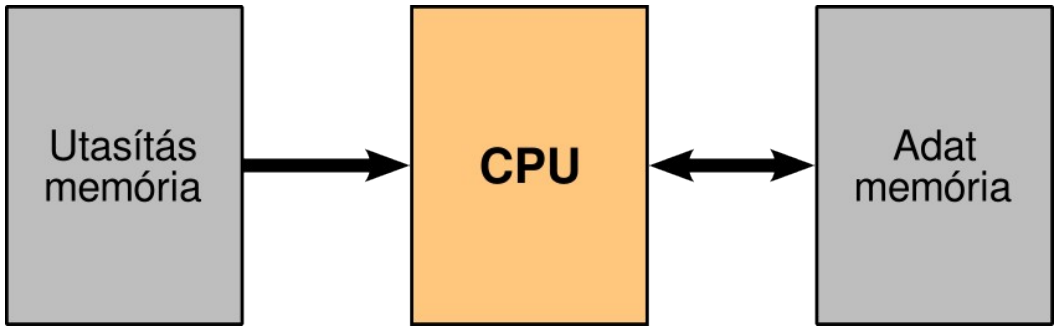
A NEUMANN ARCHITEKTÚRA

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

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, ...)

HÁLÓZATI RENDSZEREK
ÉS SZOLGÁLTATÁSOK
TANSZÉK

HÁLÓZATI RENDSZEREK
ÉS SZOLGÁLTATÁSOK
TANSZÉK

MŰEGYETEM 1782

Budapest,
2022.02.15.

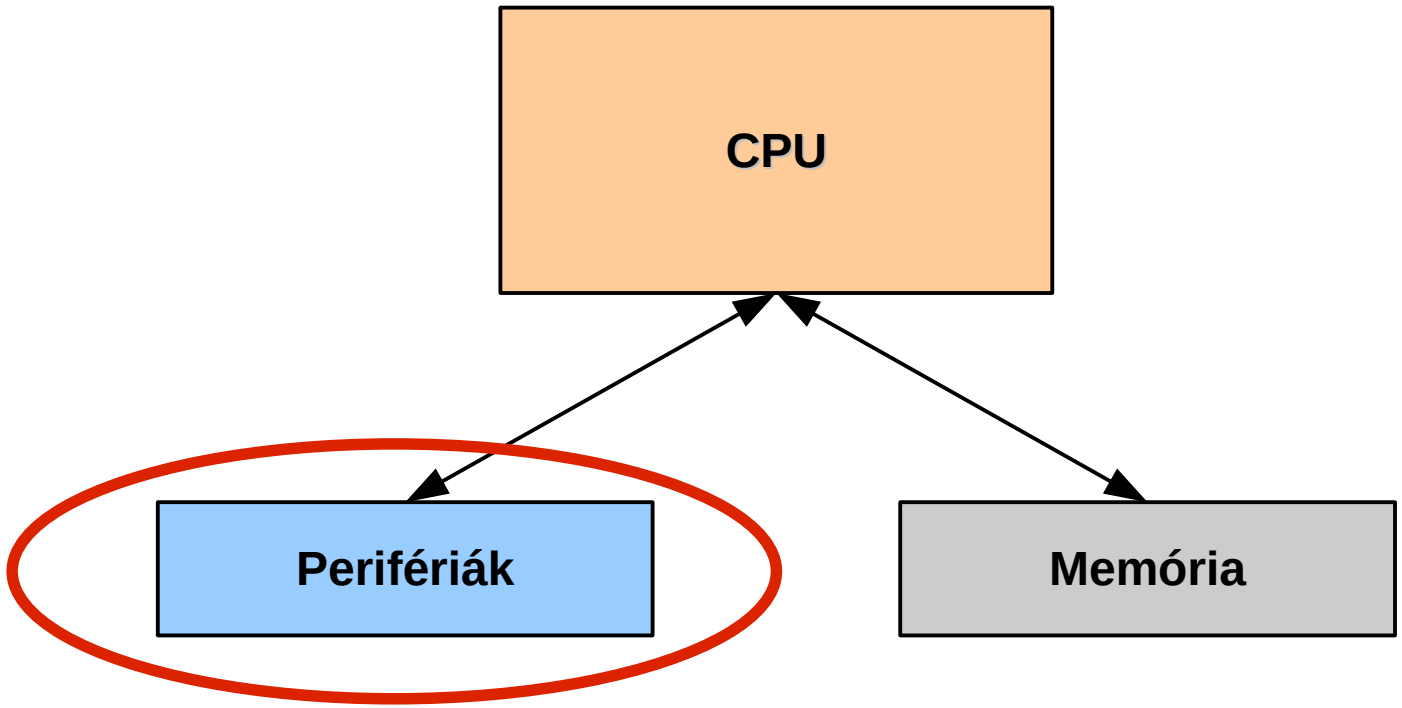
SZÁMÍTÓGÉP ARCHITEKTÚRÁK

Perifériakezelés

Horváth Gábor, Belső Zoltán

BME Hálózati Rendszerek és Szolgáltatások Tanszék
ghorvath@hit.bme.hu, belso@hit.bme.hu

HOL TARTUNK?



PERIFÉRIÁK

- Sokféle van:
  - Bemeneti / kimeneti
  - Késleltetésérzékeny / nem az
  - Sávszélesség-igényes / nem az
  - Bithibát toleráló / nem az
  - Stb.

	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:
  - Hogyan tud a processzor adatátvitelt kezdeményezni?
  - Hogyan tudnak a perifériák adatátvitelt kezdeményezni?
  - Hogyan lehet az adatátvitelt hatékonyan, hibamentesen levezényelni?
  - Hogyan/hová kössük a perifériákat a számítógéphez?

PERIFÉRIÁK

- Kérdések:
  - **Hogyan tud a processzor adatátvitelt kezdeményezni?**
  - Hogyan tudnak a perifériák adatátvitelt kezdeményezni?
  - Hogyan lehet az adatátvitelt hatékonyan, hibamentesen levezényelni?
  - Hogyan/hová kössük a perifériákat a számítógéphez?

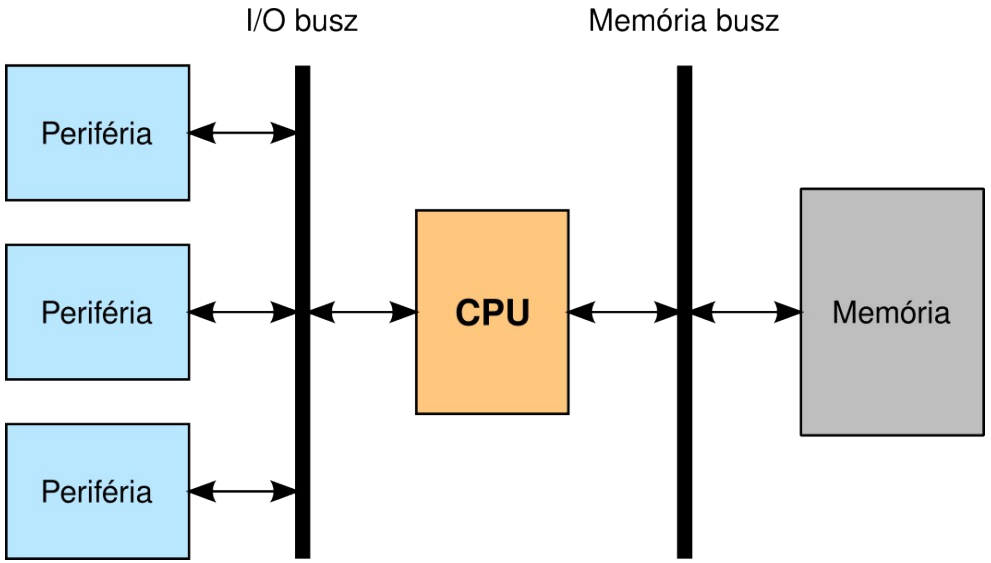
CÍMZÉS

- Memóriatartalom elérése: címzéssel
- Perifériák elérése: címzéssel

- Mennyire különülnek el a címek?
  - Lehet **külön** memória és I/O címtartomány
  - Lehet **közös** (osztott) címtartomány

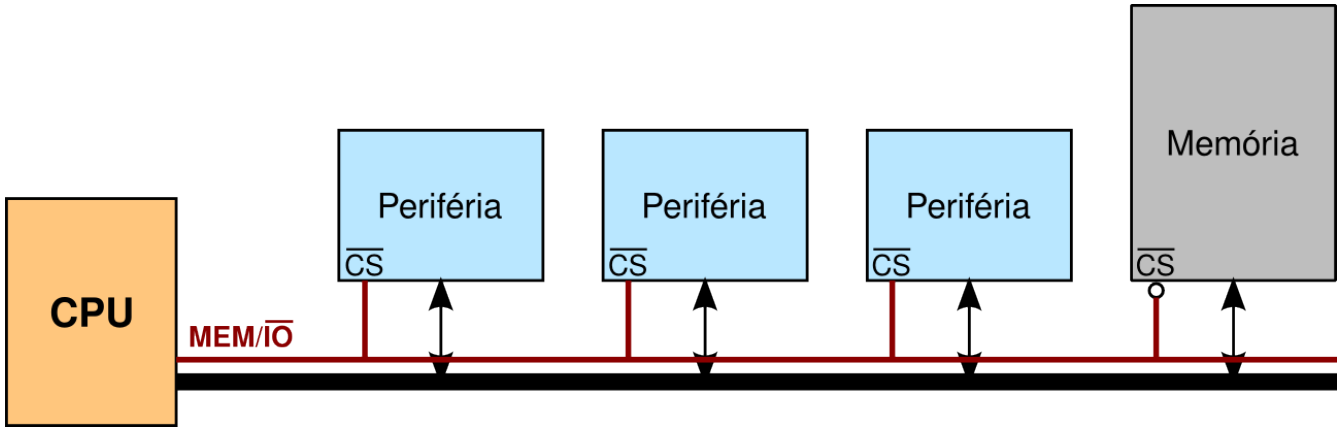
KÜLÖN MEMÓRIA ÉS I/O CÍMTÉR

- Két független címtér
  - x86: memória: 0 – 4GB, I/O: 0 – 64kB
- Külön perifériakezelő utasítások
  - `R0 ← MEM[0x60]`: memóriára vonatkozik
  - `R0 ← IO[0x60]`: perifériákra vonatkozik
- Megvalósítás:
  - Szeparált buszokkal:



KÜLÖN MEMÓRIA ÉS I/O CÍMTÉR

- Két független címtér
  - x86: memória: 0 – 4GB, I/O: 0 – 64kB
- Külön perifériakezelő utasítások
  - `R0 ← MEM[0x60]`: memóriára vonatkozik
  - `R0 ← IO[0x60]`: perifériákra vonatkozik
- Megvalósítás:
  - Multiplexált buszokkal:

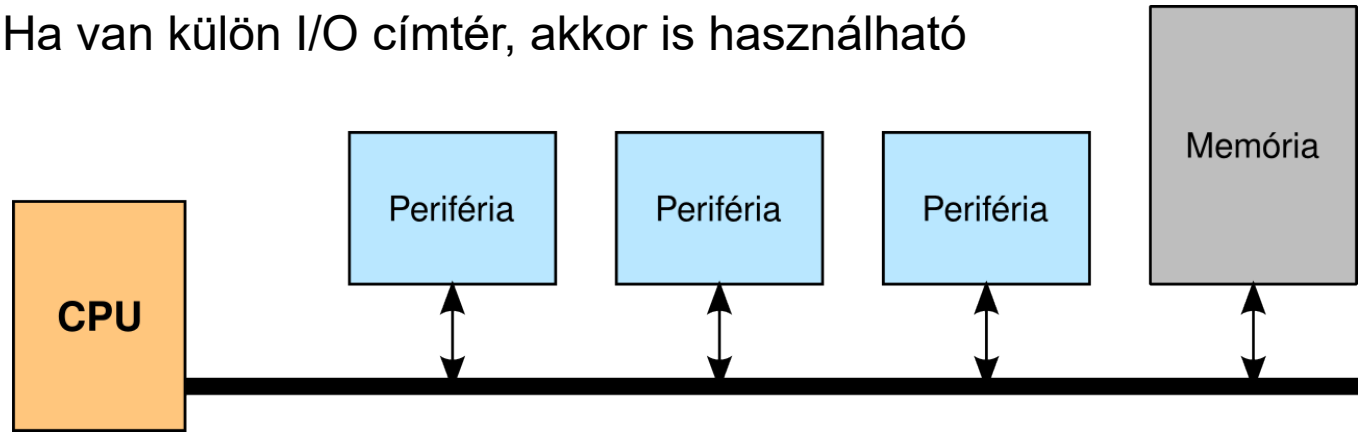


MEMÓRIÁRA LEKÉPZETT PERIFÉRIAKEZELÉS

- Egyes memóriacímekre perifériák válaszolnak
- Nagyon egyszerű a kommunikáció:

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

- **Előny**: memóriakezelő utasítások többen vannak, és kényelmesebbek, mint az I/O kezelő utasítások
- **Hátrány**: nem elérhető lyukak a memóriában
- Megvalósítás:
  - RISC CPU-k kizárólag ezt tudják
  - Ha van külön I/O címtér, akkor is használható



PERIFÉRIÁK

- Kérdések:
  - Hogyan tud a processzor adatátvitelt kezdeményezni? √
  - **Hogyan tudnak a perifériák adatátvitelt kezdeményezni?**
  - Hogyan lehet az adatátvitelt hatékonyan, hibamentesen levezényelni?
  - Hogyan/hová kössük a perifériákat a számítógéphez?

MEGSZAKÍTÁSKEZELÉS

- CPU rendszeresen kérdezgeti a perifériákat: Polling
- CPU „INT” lábán lehet megszakítást kérni
  - Gond: interrupt források száma > INT lábak száma (1-2)
  - Muszáj többeket ugyanarra az INT lábra kötni
    - Honnan tudjuk, hogy melyik kért megszakítást?
    - Mi van, ha egyszerre többen is kérnek megszakítást?

- 1. megoldás: **osztott interrupt**
  - Mindenki ugyanazon a vonalon jelez
  - Az interrupt szubrutin mindenkit körbekérdez, hogy ki volt
  - Ha többen is jeleznek egyszerre?
    - Kiszolgálási sorrend = körbekérdezési sorrend

MEGSZAKÍTÁSKEZELÉS

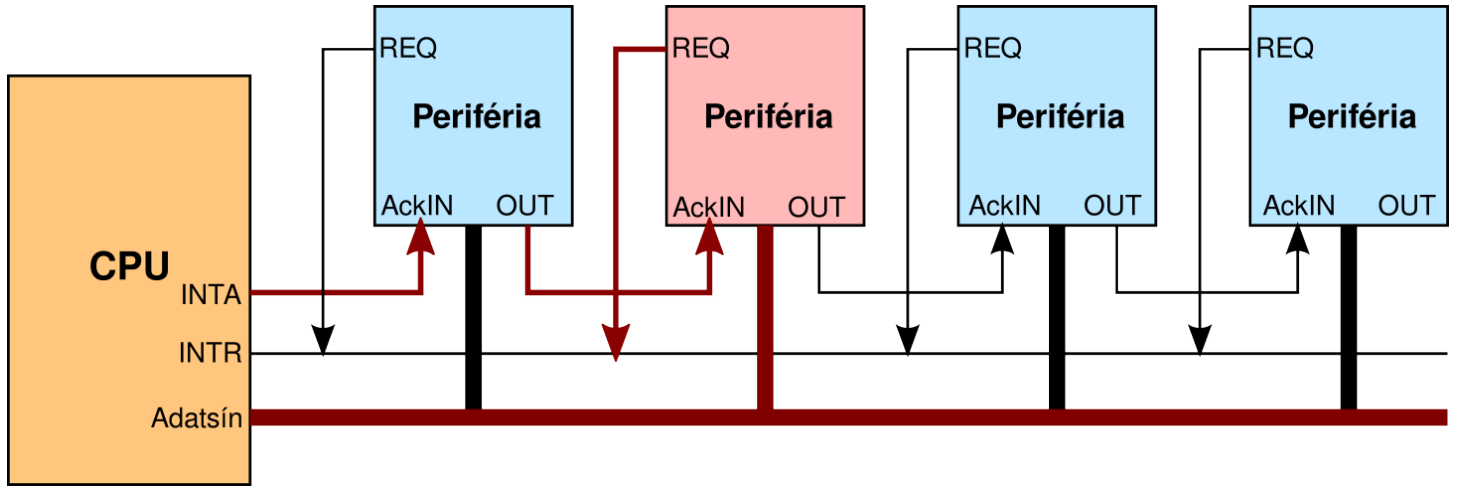
- 2. megoldás: **vektoros megszakításkezelés**
  - Megszakításkor a CPU kiad egy INTA (elfogadás) jelet
  - A jelezni kívánó periféria az adatbuszra teszi a számát
    - Ez a szám megmondja, hogy hányas számú szubrutinnak kell őt kiszolgálnia
    - A CPU-nak van egy táblázata: **interrupt vektor tábla**. Mutatókat tartalmaz interrupt kiszolgáló szubrutinokra.
    - Amilyen számot mondott a periféria, a vektor tábla annyiadik szubrutinja fut le
- A kettő kombinációja
  - Néha a vektortábla elemeinek száma sem elég
  - Az azonos vektoron levő perifériák között marad a körbekérdezés

MEGSZAKÍTÁSKEZELÉS

- 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

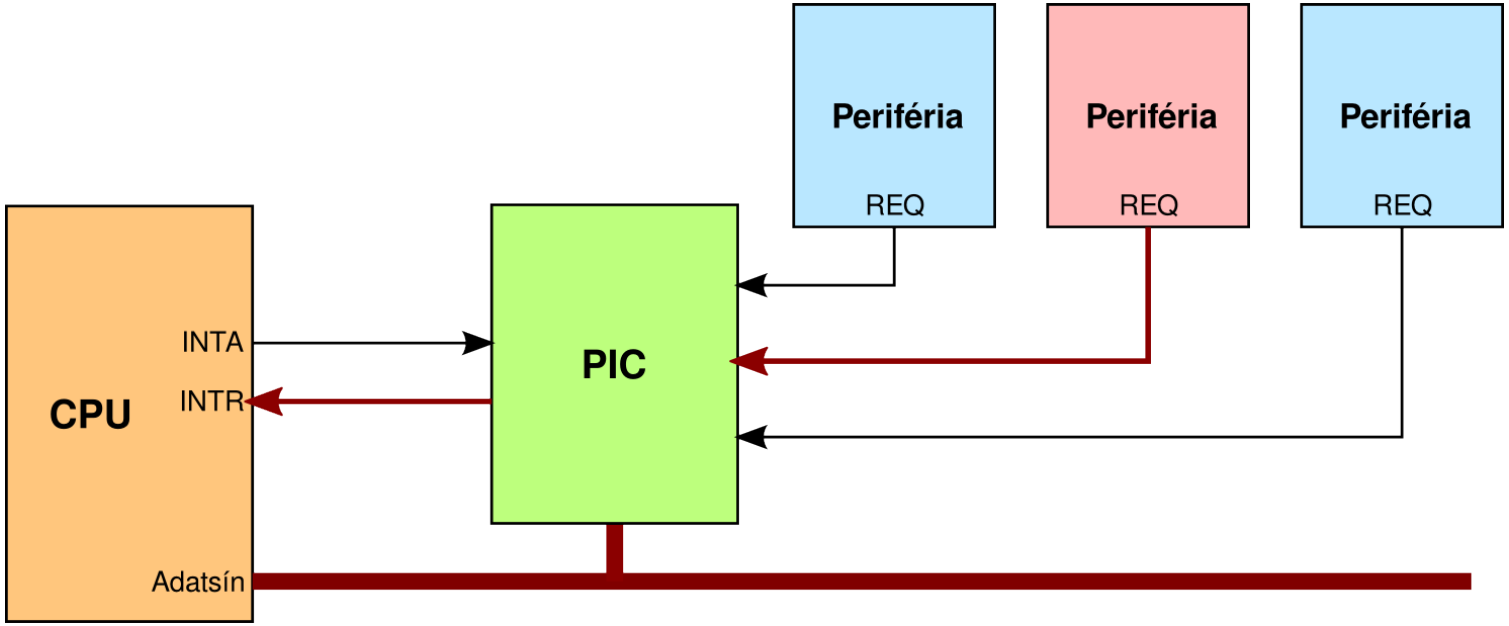
MEGSZAKÍTÁSKEZELÉS

- **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



MEGSZAKÍTÁSKEZELÉS

- **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



MEGSZAKÍTÁSKEZELÉS

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

MEGSZAKÍTÁSKEZELÉS

- Amikor túl sok a jóból...
  - Vannak perifériák, melyek túl sok interruptot generálnak
    - Pl. a gigabit sebességű hálózati interfészek

Interrupt forrás	Tipikus ráta	Maximális ráta
10 Mbps Ethernet	812	14880
100 Mbps Ethernet	8127	148809
Gigabit Ethernet	81274	1488095

- A CPU megállás nélkül megszakítást kezel, mással nem tud haladni
- Irányelvek:
  - A megszakításkezelő szubrutin legyen rövid
  - Kritikus rendszerekben az interrupt rátát maximálják
  - A futó program kritikus szakaszaiban le kell tiltani a megszakítást
  - **Interrupt moderation**
    - A periféria több eseményt összevár, és akkor jelez, egyszer
    - A CPU egy interrupt-tal lekezeli a periféria összegyűlt mondanivalóját

PERIFÉRIÁK

- Kérdések:
  - Hogyan tud a processzor adatátvitelt kezdeményezni? √
  - Hogyan tudnak a perifériák adatátvitelt kezdeményezni? √
  - **Hogyan lehet az adatátvitelt hatékonyan, hibamentesen levezényelni?**
  - Hogyan/hová kössük a perifériákat a számítógéphez?

HATÉKONY ADATÁTVITEL

- Végső cél:
  - Adatok átvitele a perifériából a memóriába
- Kérdések:
  - Mi van, ha a küldő és a fogadó sebessége eltérő?
    → **Forgalomszabályozás**
  - Milyen útvonalon juttassuk az adatokat a memóriába?

FORGALOMSZABÁLYOZÁS

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

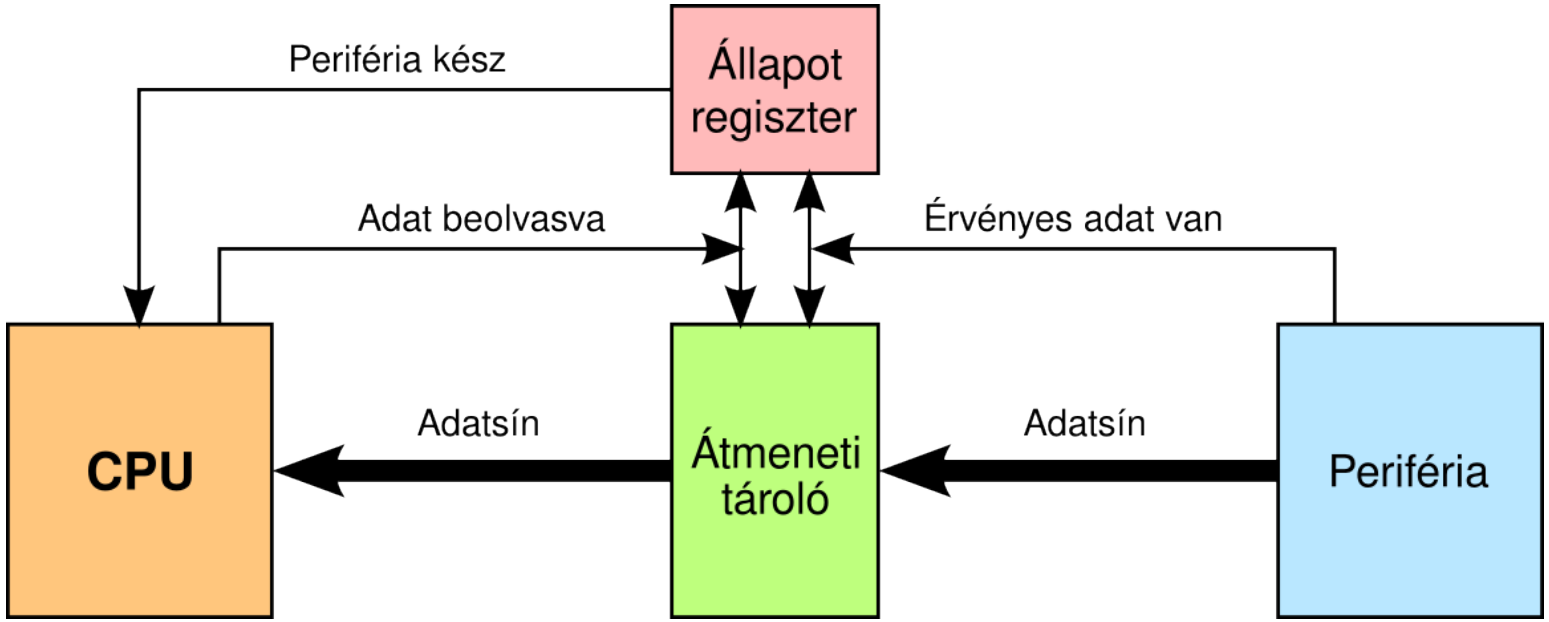
1) Feltétel nélküli adatátvitel

- 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, ...

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

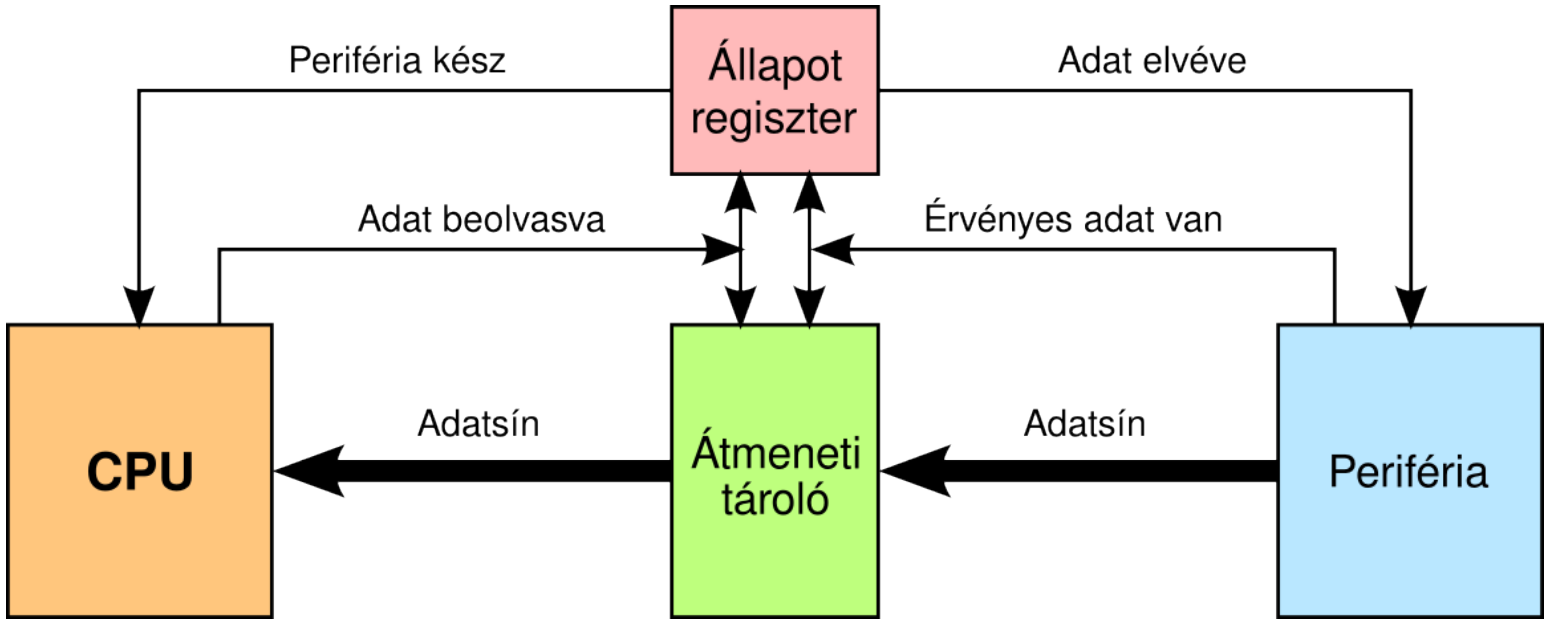
  → az állapotregiszterrel az adathiány elkerülhető!



3) Kétoldali feltételes adatátvitel

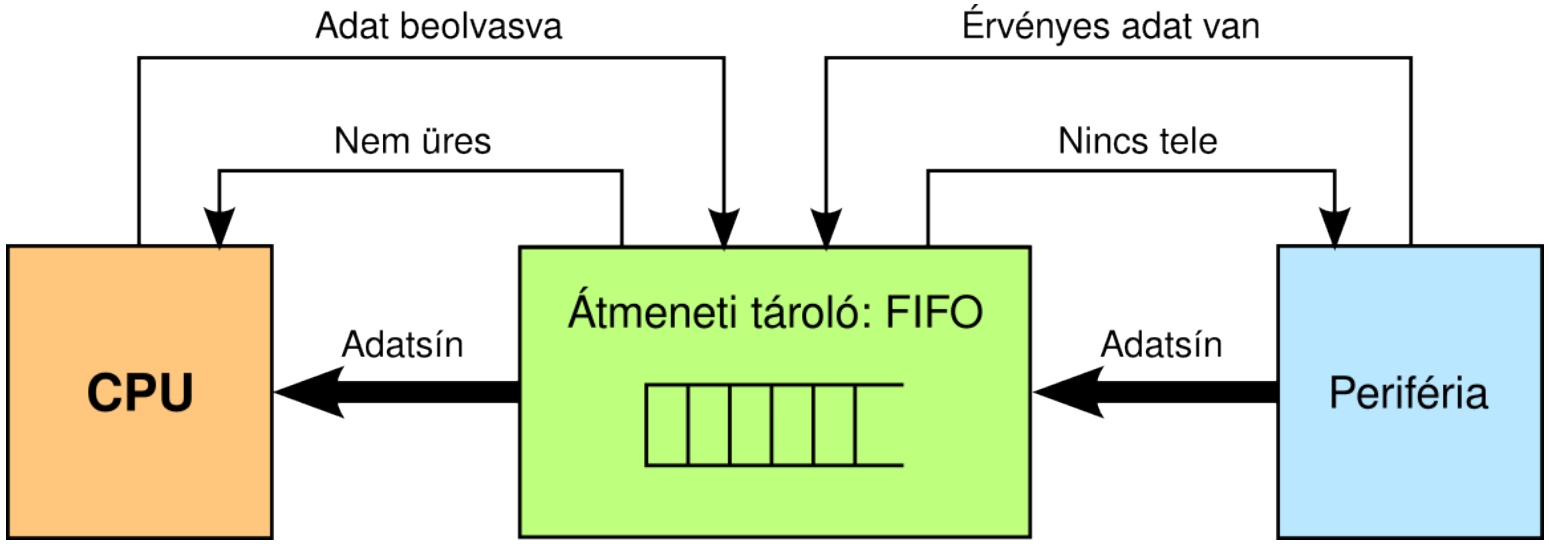
- Mindkét oldal sebessége befolyásolható
- Állapotregiszter: van-e érvényes adat?
- Példa: olvasás művelet

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



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

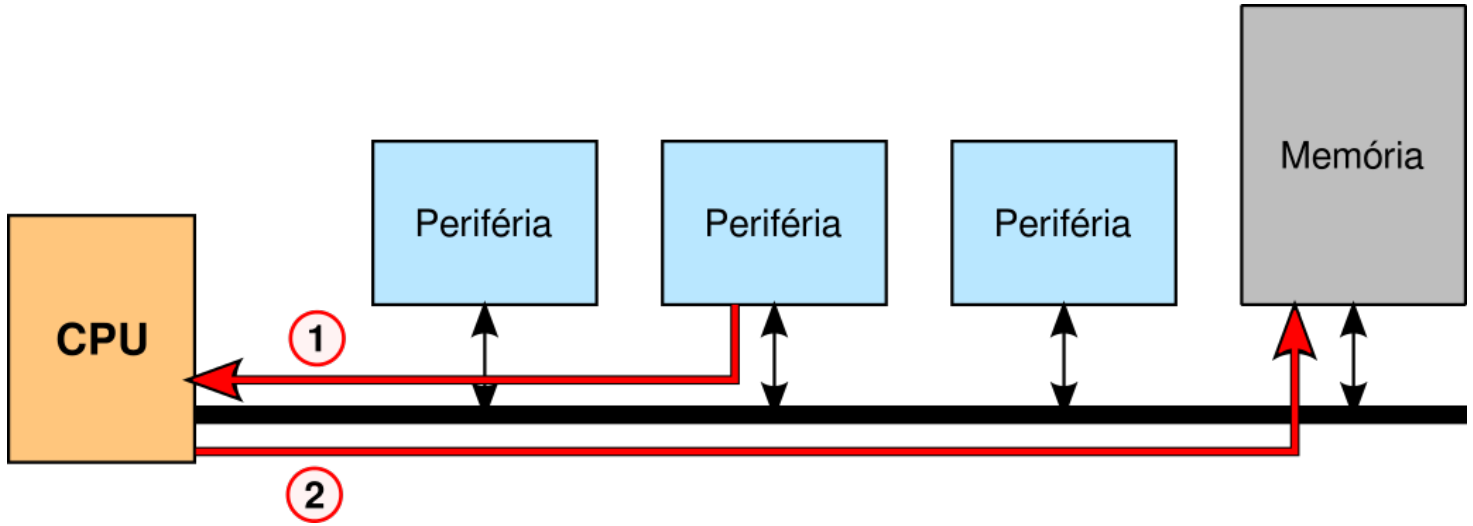


HATÉKONY ADATÁTVITEL

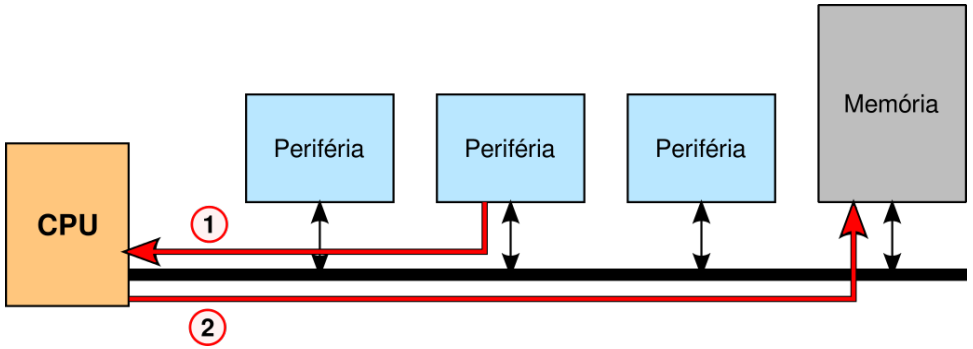
- Végső cél:
  - Adatok átvitele a perifériából a memóriába
- Kérdések:
  - Mi van, ha a küldő és a fogadó sebessége eltérő?
    → Forgalomszabályozás √
  - **Milyen útvonalon juttassuk az adatokat a memóriába?**

HATÉKONY ADATÁTVITEL

- Végső cél:
  - Adatok átvitele a perifériából a memóriába
- Milyen útvonalon juttassuk az adatokat a memóriába?
- 1. megoldás: **a processzoron keresztül**



HATÉKONY ADATÁTVITEL

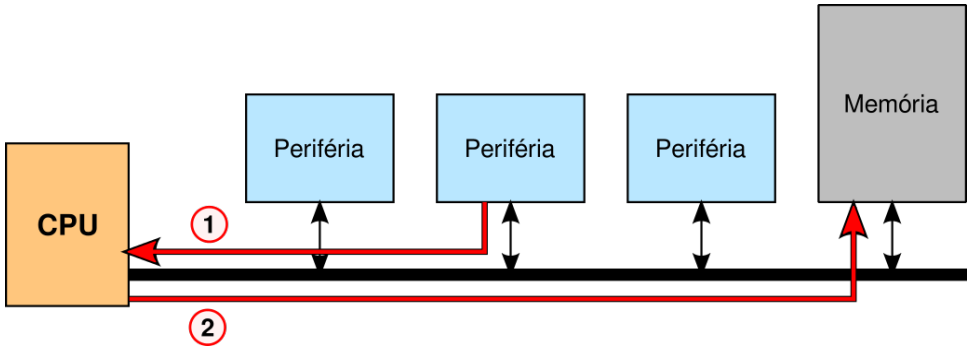


- **Polling:**
  - A periféria állapotát folyamatosan figyeljük:

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

  - Kritikus kérdés: polling periódus
    - Túl gyakori → nagy CPU terhelés
    - Túl ritka → adatvesztés

HATÉKONY ADATÁTVITEL



- **Megszakításra alapozott adatátvitel:**
  - A periféria készenlétét megszakítással detektáljuk
  - Megszakításkezelő szubrutin:

```
billkezelo: PUSH R0
            R0 ← IO[0x60]
            MEM[0x142] ← R0
            POP R0
            RET
```

  - Csak annyira terheli a CPU-t, amennyire muszáj

POLLING VS. INTERRUPT

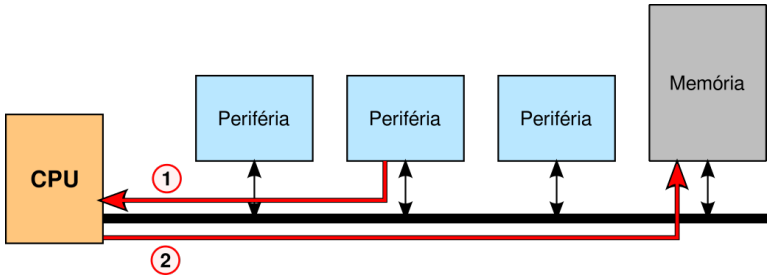
- Példák pollingra (CPU: 1 GHz, 1 lekérdezés 600 órajel)
  - Egér:
    - másodpercenként 30-szor kérdezzük le
    - 30 poll/s * 600 órajel/poll = 18000 órajel/s
    - CPU: 10^9 órajel/s → 18000 / 10^9 = 0.0018%, OK.
  - Diszk:
    - Interfész: $100 \cdot 10^6$ byte/s, 500 byte/blokk
    - Polling periódus: $100 \cdot 10^6$ byte/s / 500 byte/blokk = 200 000 poll/s
    - 200 000 poll/s * 600 órajel/poll = 120 000 000
    - CPU: 10^9 órajel/s → $120 \cdot 10^6$ / 10^9 = 12%
    - Megengedhetetlenül nagy: egyetlen periféria egyetlen jelzése!

POLLING VS. INTERRUPT

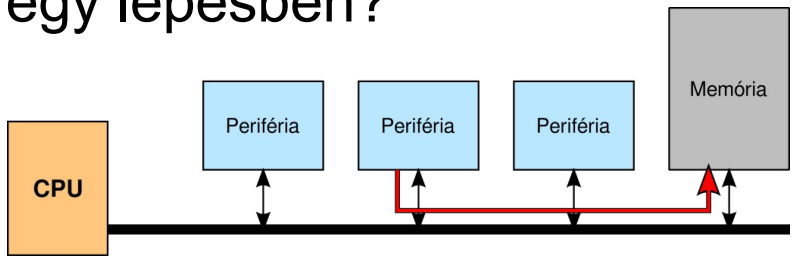
- Példa interrupt-ra alapozott adatátvitelre:
  - Diszk
    - A diszk az idő 10%-ában aktív
    - Interrupt feldolgozási idő: 600 órajel
    - Tényleges adatátviteli idő: 100 órajel
    - Interrupt feldolgozásra fordított idő:
    - $0.1 \cdot (100 \cdot 10^6$ byte/s / 500 byte/blokk · 600 órajel/interrupt)
      = 12 000 000 órajel/s
    - CPU: 10^9 órajel/s → $12 \cdot 10^6 / 10^9 = 1.2\%$
    - Adatátviteli idő:
    - $0.1 \cdot (100 \cdot 10^6$ byte/s / 500 byte/blokk · 100 órajel/átvitel)
      = 2 000 000 órajel/s
    - CPU: 10^9 órajel/s → $2 \cdot 10^6 / 10^9 = 0.2\%$
    - Összesen: 1.2 % + 0.2% = 1.4%

A PROCESSZOR TEHERMENTESÍTÉSE

- Periféria → memória átvitel az eddigiek szerint:



- Nem lehetne egy lépésben?

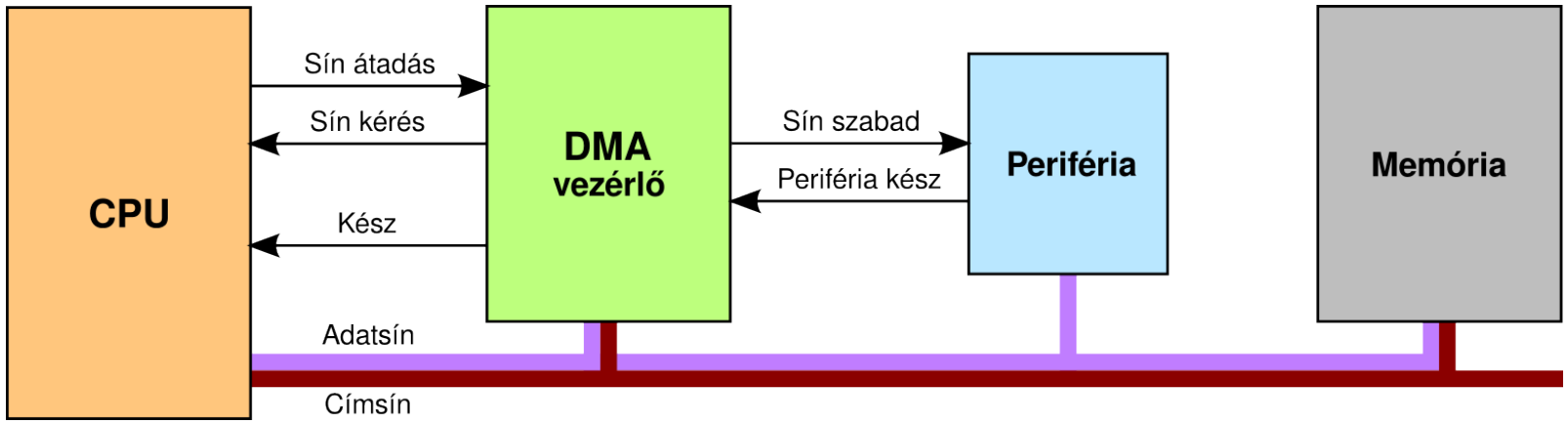


  - Gyorsabb lenne
  - CPU közben mást csinálhatna
  - Megoldások:
    - DMA
    - I/O processzor

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



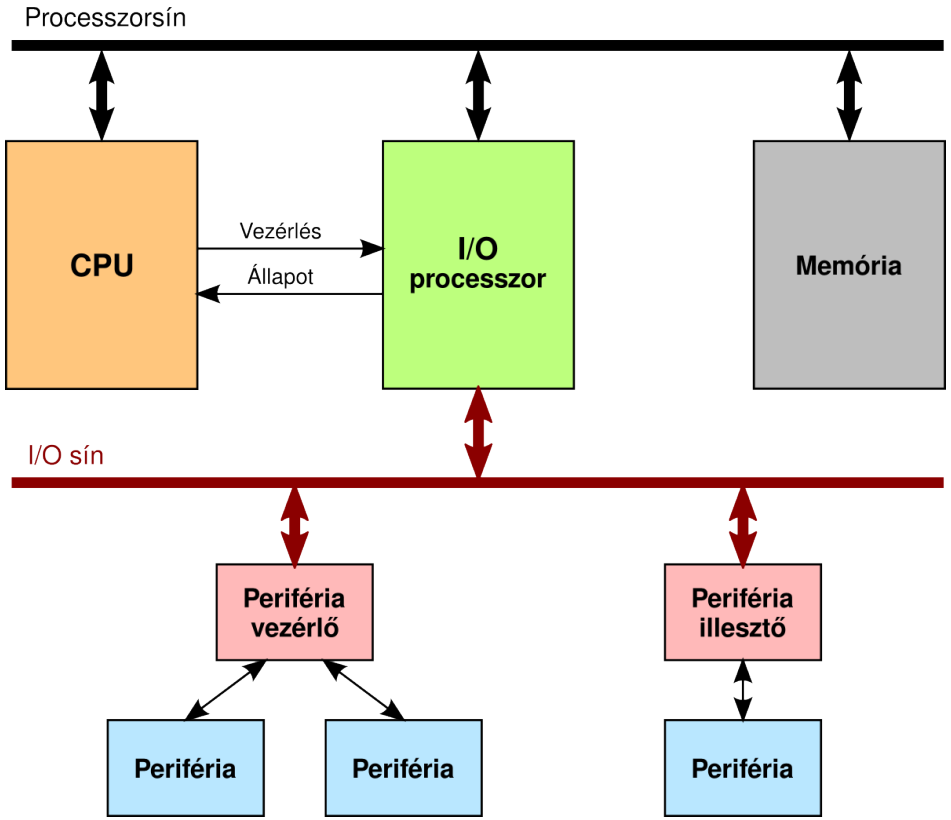
- **A CPU-nak csak a blokkos átvitel kezdeményezésekor és a teljes blokk átvitelének végén van dolga**
- Ez a rendszerszintű (third party) DMA vezérlő
- Ma már nem használják
- Amit ma használnak: first party DMA vezérlő
  - A perifériáknak saját DMA vezérlője van
  - Önállóan tudnak versenyezni a buszért
    - 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 utasításkészlett**el 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

PERIFÉRIÁK

- Kérdések:
  - Hogyan tud a processzor adatátvitelt kezdeményezni? √
  - Hogyan tudnak a perifériák adatátvitelt kezdeményezni? √
  - Hogyan lehet az adatátvitelt hatékonyan, hibamentesen levezényelni? √
  - **Hogyan/hová kössük a perifériákat a számítógéphez?**

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

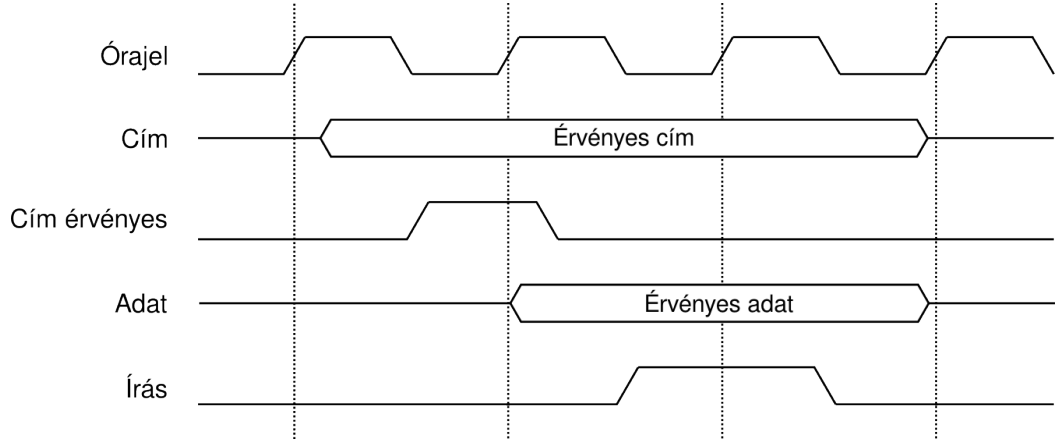
ÖSSZEKÖTTETÉSEK SZÉLESSÉGE

- Szélesség = a jelvezetékek a száma
- Széles összeköttetés:
  - 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

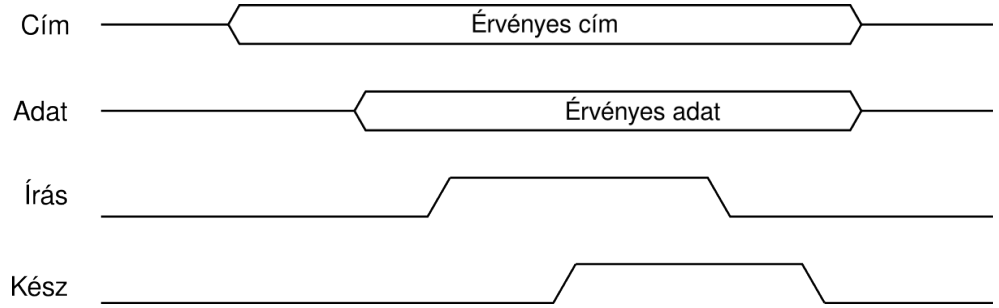
    → az egy időben küldött jelek egymáshoz képest csúszva érkeznek meg!
  - Akkor okoz gondot, ha a csúszás összemérhető az órajellel
- Trend: soros átvitelt mindenhova → nincs csúszás, ami korlátozza a sebességet

IDŐZÍTÉS

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



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



- Szinkron – aszinkron: **melyik jobb?**

IDŐZÍTÉS

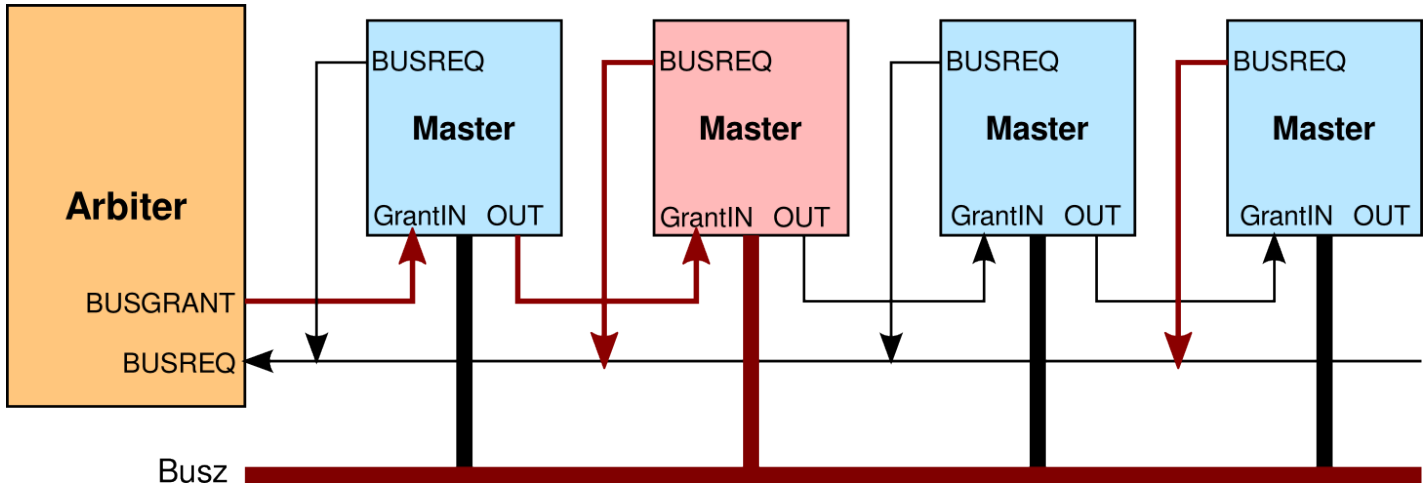
- Egyik sem:
  - **Adat vezeték – órajel vezeték között csúszás léphet fel!**
- Megoldás:
  - Ne legyen órajel
  - Ne legyenek adat érvényességére vonatkozó vezérlőjelek
  - Csak 1 szál drót az adatnak (soros átvitel)
- Honnan tudja a periféria, hogy hol vannak a bithatárok?
  - Jöjjön rá a 0-1 bitek váltakozásából
    → önidőzitő adatátvitel
  - Csak akkor megy, ha van 0-1 váltakozás
- **8b/10b kódolás**:
  - Csinál 0 – 1 átmenetet (csupa 0 – csupa 1 sorozat max. hossza: 5)
  - 20 hosszan max. 2 lehet a 0-k és 1-k száma közt a különbség
  - Gigabit Ethernet, PCI Express 1.0 és 2.0, USB 3.0, SATA, HDMI, DVI, stb.
  - Hátrány: overhead
  - Továbbfejlesztés:
    - 64b/66b: 10 GB Ethernet, 100 GB Ethernet
    - 128b/130b: PCI Express 3.0/4.0, USB 3.1, SATA 3.2, DisplayPort 2.0
    - 256b/257b: Fibre Channel

ARBITRÁCIÓ A BUSZON

- Busz: osztott erőforrás
  - Meg kell oldani, hogy egyszerre csak egyvalaki használja
- 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

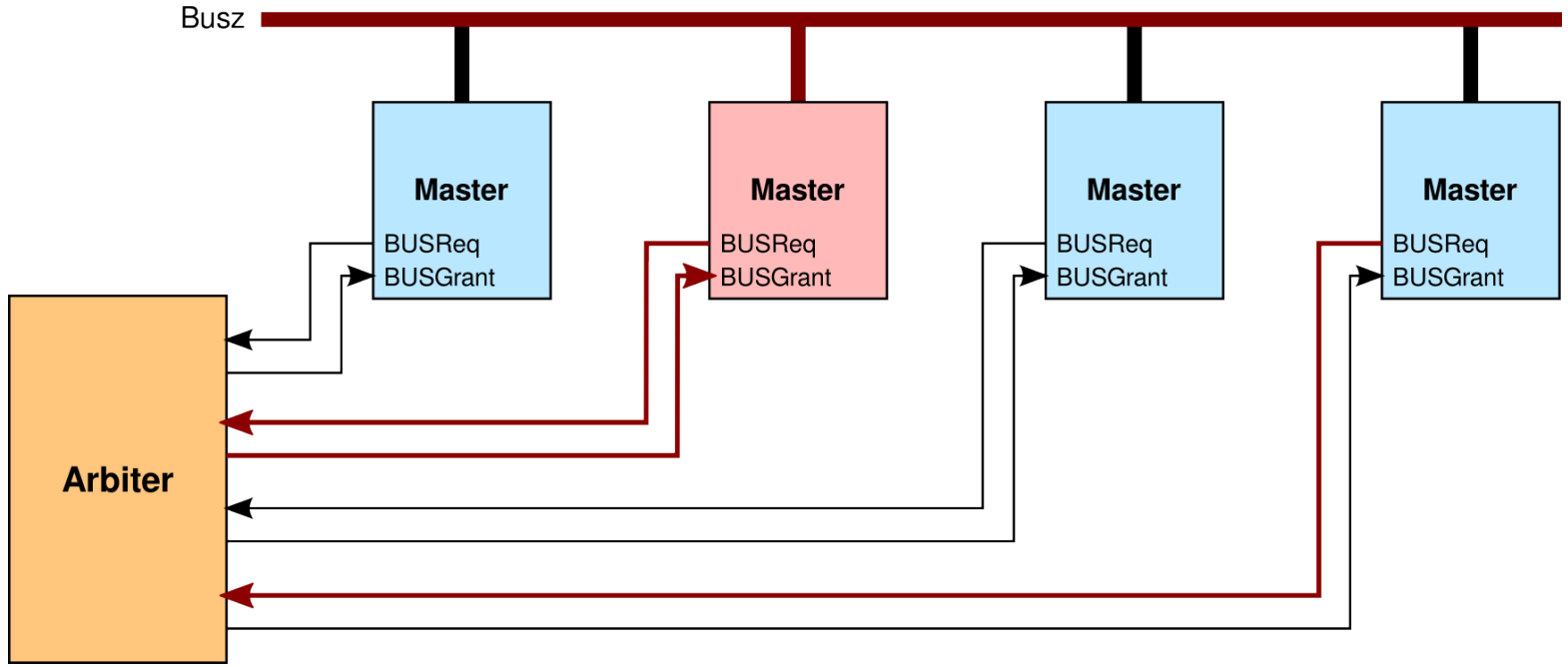
- Speciális egység: **Arbiter**
  - Eldönti, ki kapja a buszhasználat jogát
- Soros arbitráció: **Daisy Chain**



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

CENTRALIZÁLT ARBITRÁCIÓ

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

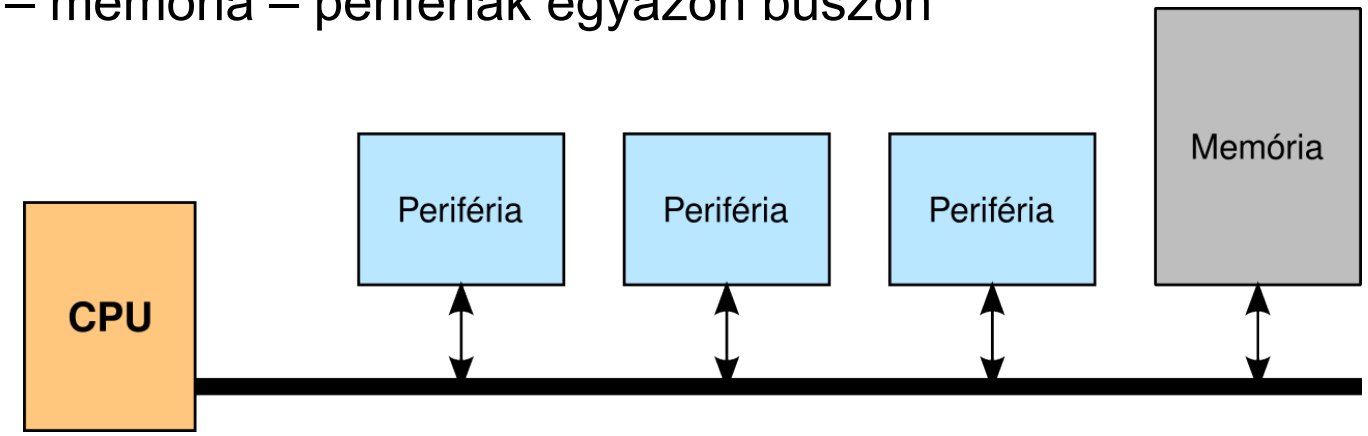


ELOSZTOTT ARBITRÁCIÓ

- 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

EGYBUSZOS RENDSZEREK

- CPU – memória – perifériák egyazon buszon

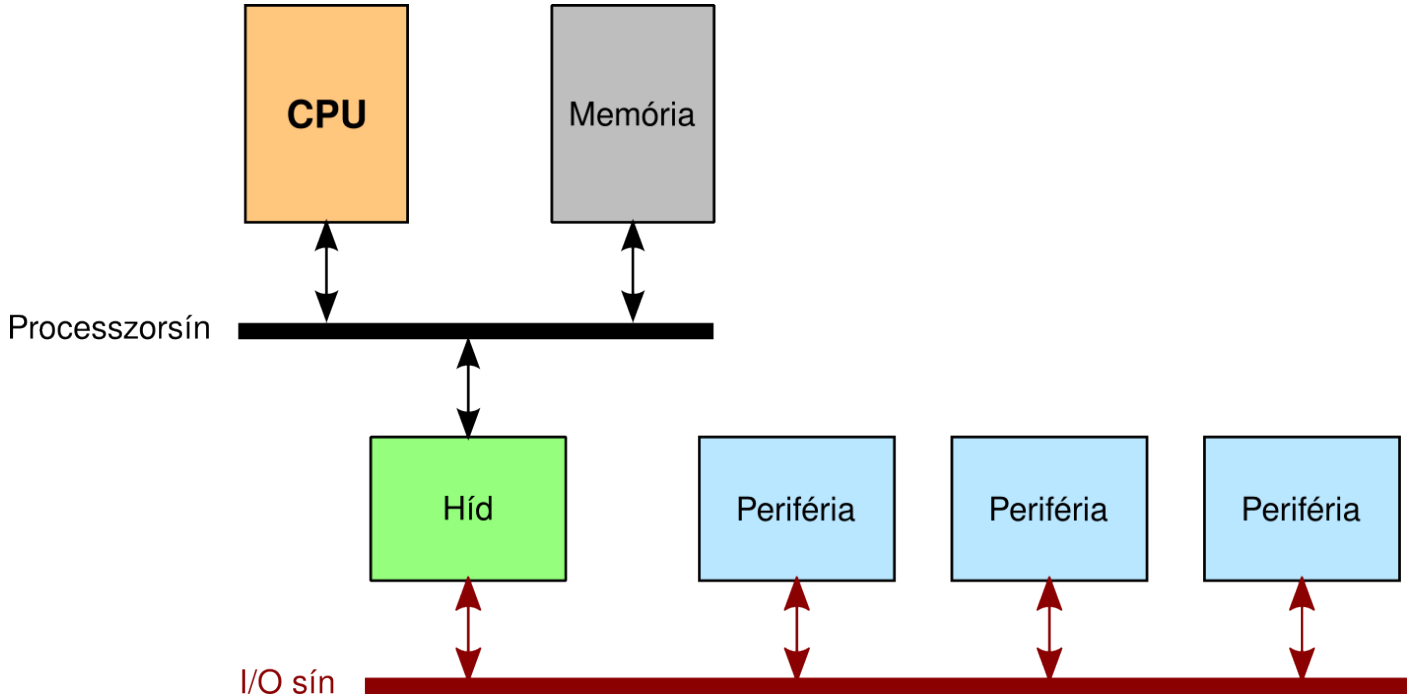


- Egyszerű megvalósítani
- Hátrány:
  - Magas CPU órajel → magas busz órajel → drágább perifériák → korlátozott fizikai távolság
  - A CPU órajele / busz órajele és a busz protokoll típustól függően változik

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

KÉTBUSZOS RENDSZEREK

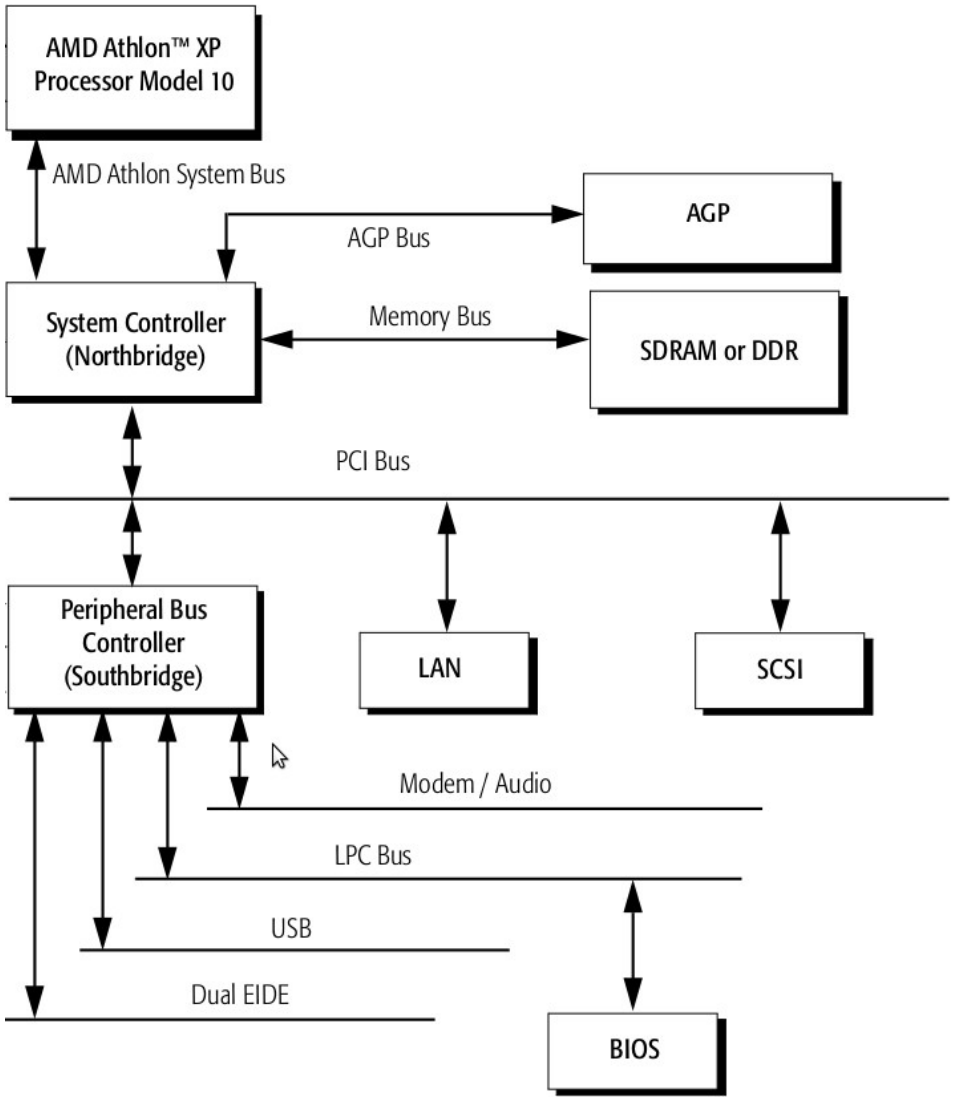
- Perifériák külön buszon, CPU-memória egy másik buszon
- Átjárás: **híd**



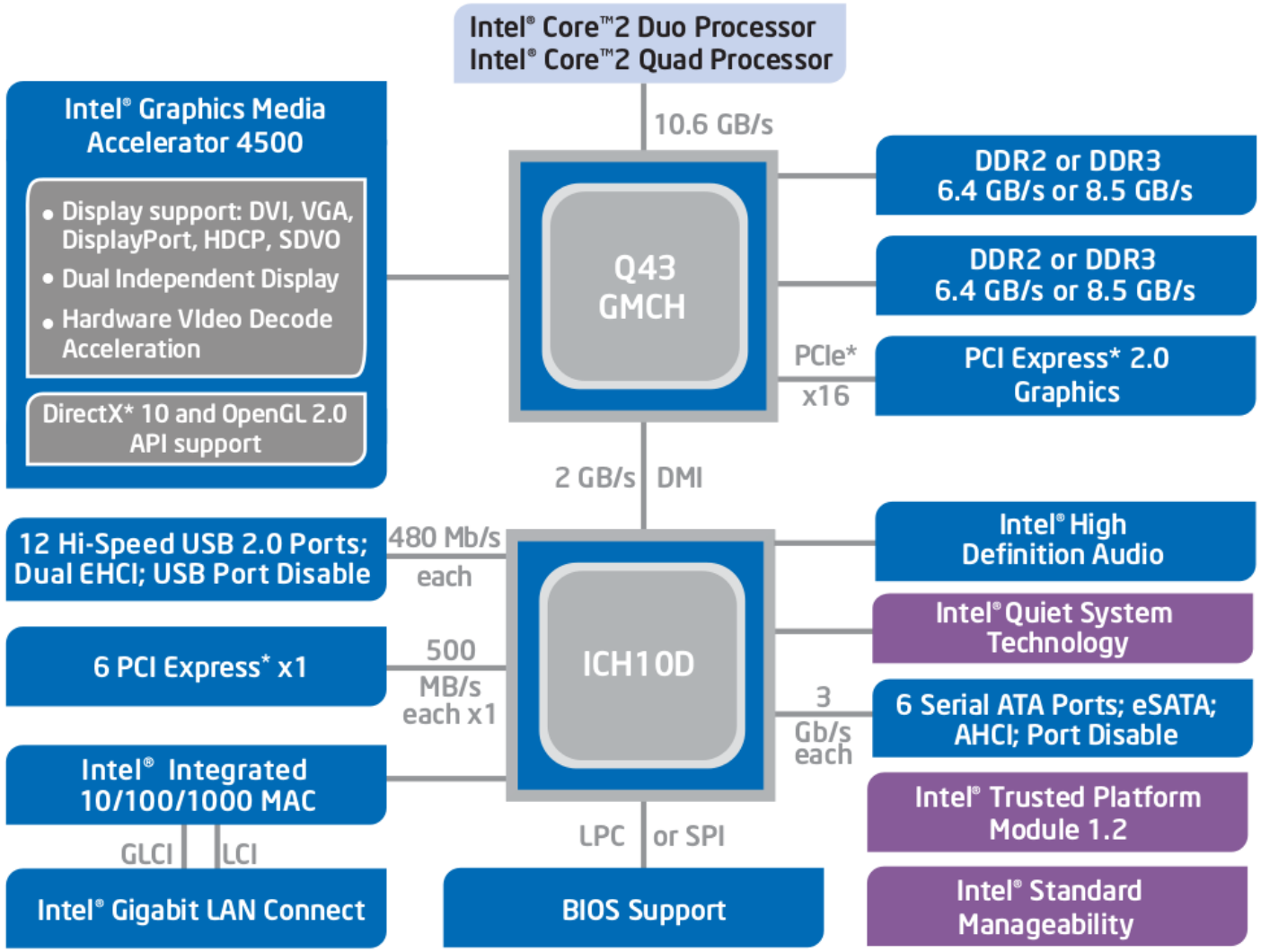
- Processzorsín: típusfüggő sebesség és protokoll
- I/O sín: állandó, szabványos sebesség és protokoll

HÍD ALAPÚ RENDSZEREK

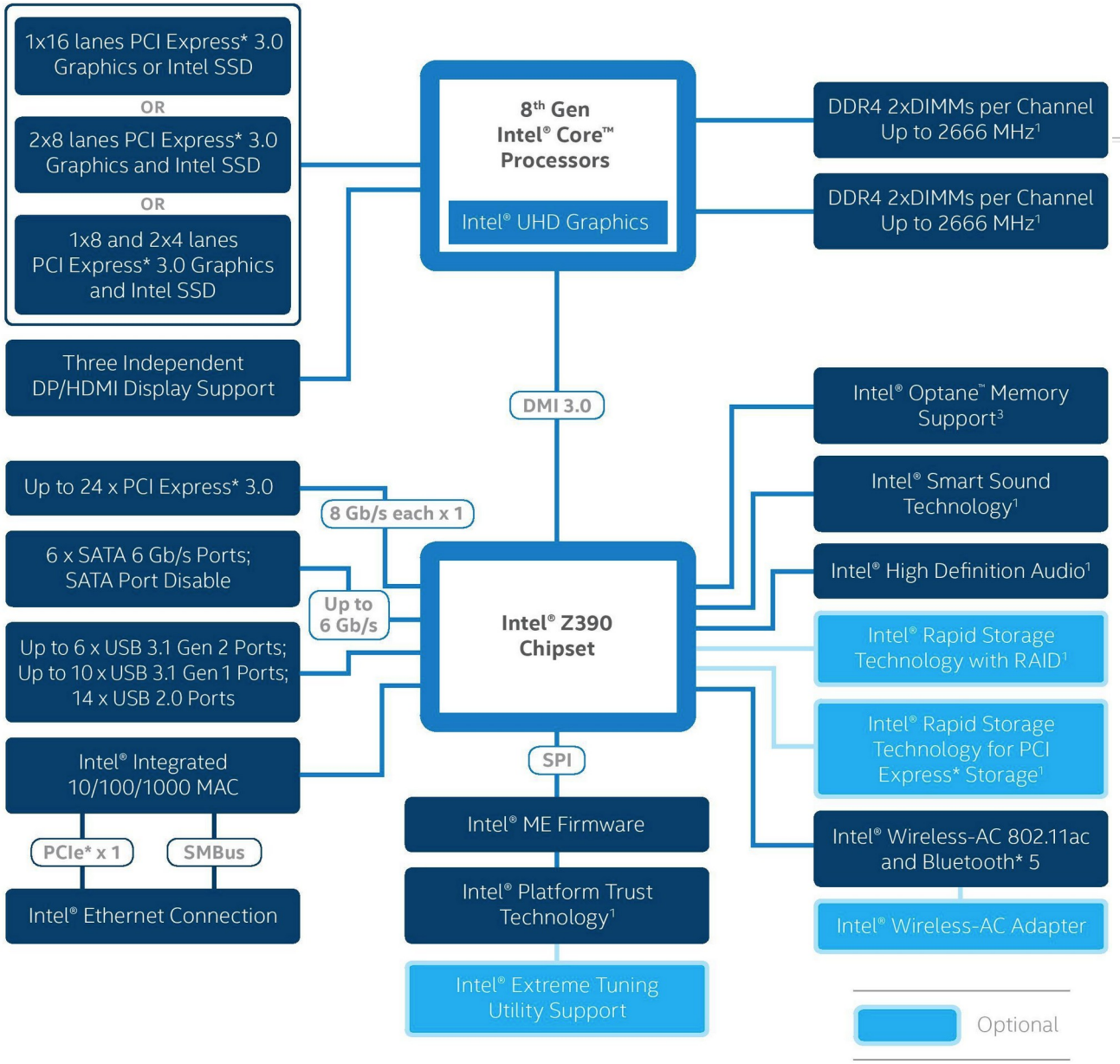
- Külön buszon:
  - CPU
  - Memória
  - Perifériák
- Processzorsín:
  - Típusfüggő
- Memóriasín:
  - Állandó, szabványos
- Perifériasín (PCI):
  - Állandó, szabványos



HÍD ALAPÚ RENDSZEREK



HÍD ALAPÚ RENDSZEREK



HÁLÓZATI RENDSZEREK
ÉS SZOLGÁLTATÁSOK
TANSZÉK

HÁLÓZATI RENDSZEREK ÉS SZOLGÁLTATÁSOK TANSZÉK

SZÁMÍTÓGÉP ARCHITEKTÚRÁK

Csatolófelületek

Horváth Gábor, Belső Zoltán

BME Hálózati Rendszerek és Szolgáltatások Tanszék
ghorvath@hit.bme.hu, belso@hit.bme.hu

Budapest,
2022.03.02.

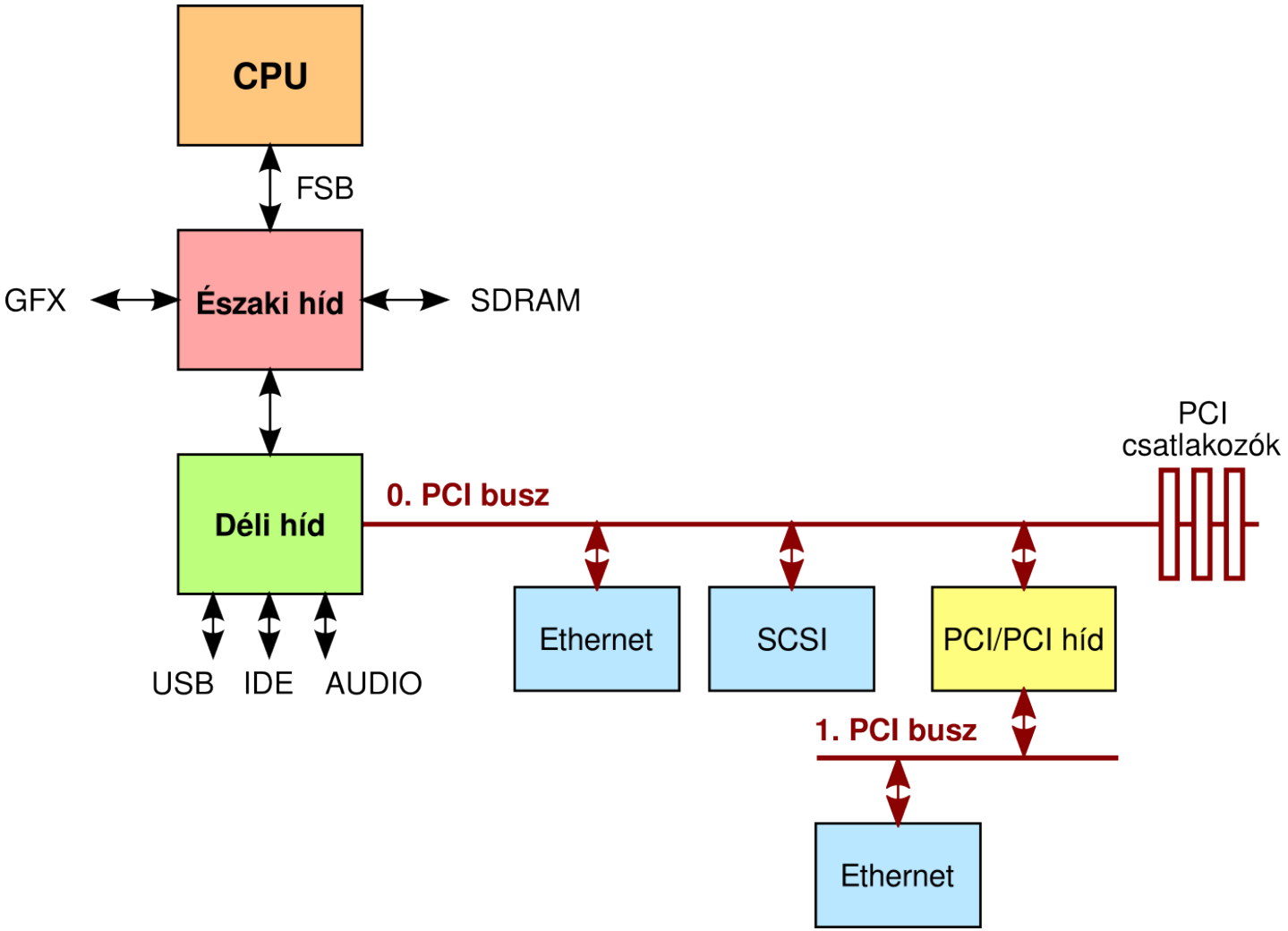
MŰEGYETEM 1782

MOTIVÁCIÓ

- 90-es évek: a PC-s perifériák csatlakozása körül káosz
  - Ezernyi gyártóspecifikus csatlakozó
  - Ha nem volt az alaplapon, illesztőkártya kellett
  - A PC-be tehető illesztőkártyák száma korlátos
  - Az ezernyi csatlakozóhoz különböző kábelek, hosszabbítók kellenek
- Megoldás:
  - PCI (= Peripheral Component Interfész, 1992, Intel)
  - USB (= Universal Serial Bus, 1994, Compaq, DEC, Intel, Microsoft, …)
- Közös tulajdonságok:
  - Processzorfüggetlenek
  - Automatikus periféria felismerés és konfiguráció támogatása
  - A számítógép kikapcsolása nélkül illeszthető/eltávolítható perifériák

PCI

PCI HELYE A SZÁMÍTÓGÉPBEN



A PCI BUSZ ELEMEI

- A PCI busz elemei:
  - **Host/PCI híd**
    - Processzor I/O és memória kérések ↔ PCI tranzakciók
  - **PCI eszközök**
    - Max 32 / busz, elektromos okokból inkább max. 10
    - Egy eszközön 1-8 function (logikai periféria)
  - **PCI/PCI híd**
    - Egy PCI sínre egy másik sínt illeszt
  - Illeszthető perifériák max száma elméletileg: 65536

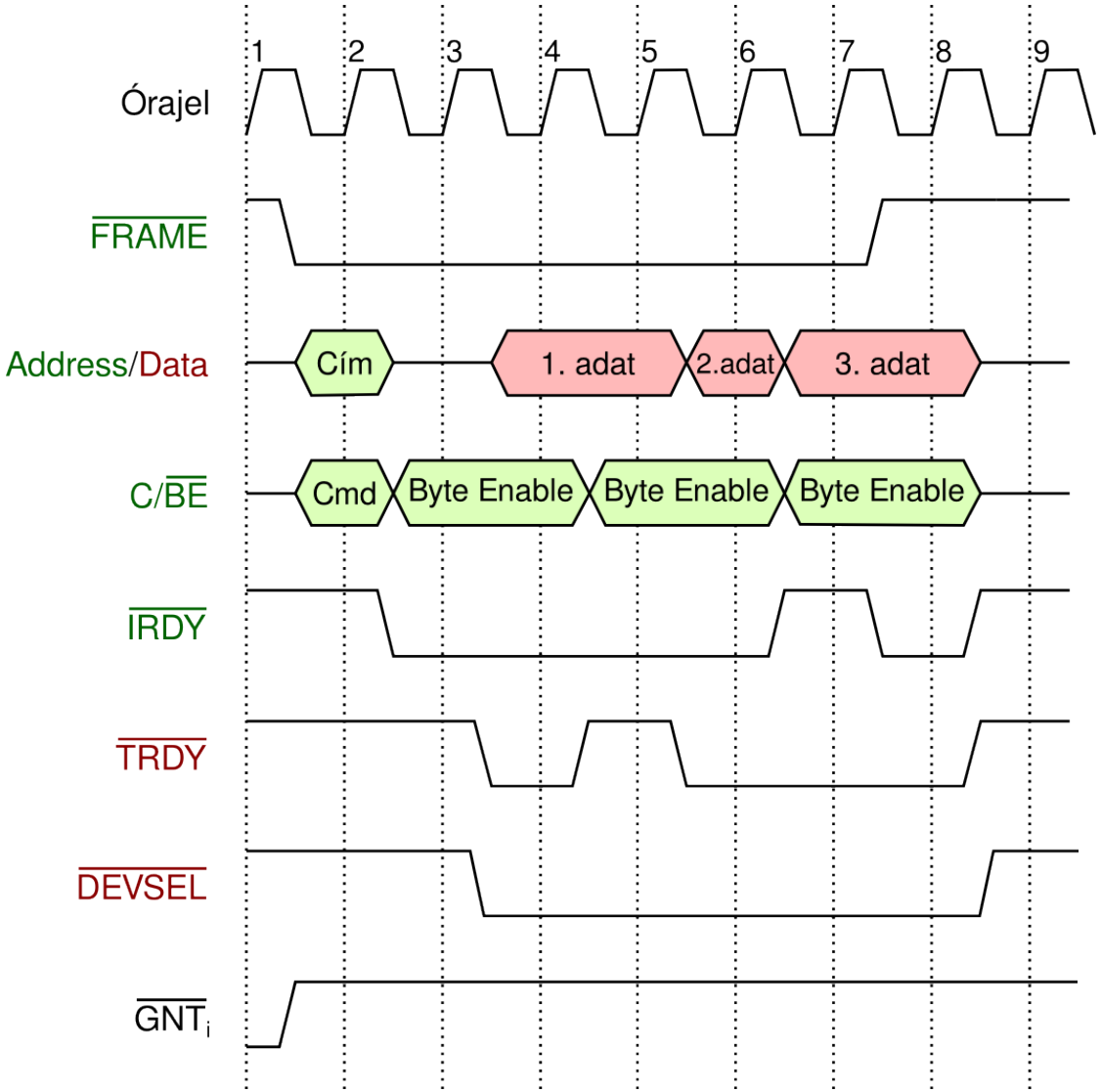
AZ ADATÁTVITEL ELVE

- Minden PCI perifériának (funkciónak) max. 6 **ablak**
  - Ablak lehet:
    - A memória címtartomány egy része
    - Az I/O címtartomány egy része
  - Az eszköz figyeli az ablakait célzó műveleteket, ha ilyet lát, válaszol

- Ablakok kiosztása:
  - Az eszköz megmondja, hogy neki hány és mekkora kell
  - A BIOS ill. az op. rendszer osztja ki az ablakokat...
  - … majd a perifériában beállítják az ablakok kezdetét

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

ADATÁTVITEL



Forgalomszabályozás:

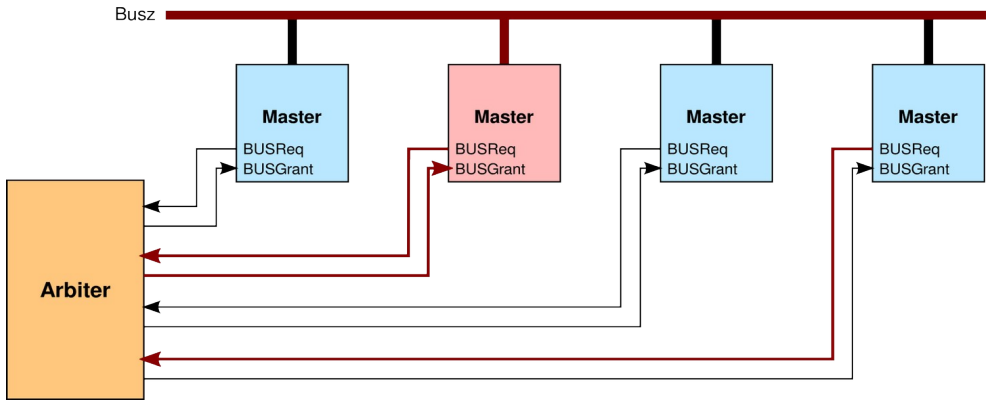
- $\overline{\text{IRDY}}$: kezdeményező kész a következő adat átvitelére
- $\overline{\text{TRDY}}$: a megcímzett periféria kész a következő adat átvitelére

Fázisok:

- Címzés + parancs
- Címzett jelenetkezik
- Frame átvitel adategységenként forgalomszabályozással
- Frame vége → tranzakció vége

A PCI ADATÁTVITEL ELEMEI

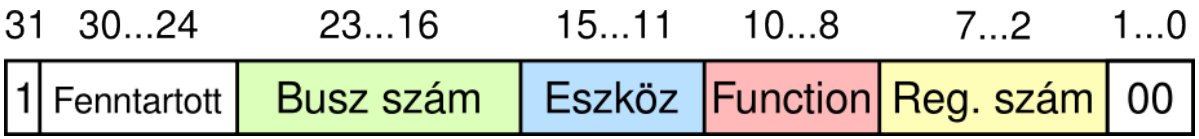
- **Arbitráció**
  - Párhuzamos arbitráció:



  - Rejtett arbitráció:
    - Versengés és eredményhirdetés az aktuális tranzakció ***közben***
  - Fair: figyelembe veheti a periféria késleltetés-érzékenységét
- **Interrupt kezelés**
  - Kétféleképpen:
    - Az interrupt vezeték egyikével (4 van)
      - Több eszköz is osztozhat ugyanazon a megszakításon
    - Üzenettel jelzett megszakítással (MSI)
      - A periféria egy speciális adatot ír egy speciális címre
      - Host/PCI híd figyeli a címet, az írást CPU interrupt-ra képezi le

KONFIGURÁCIÓ

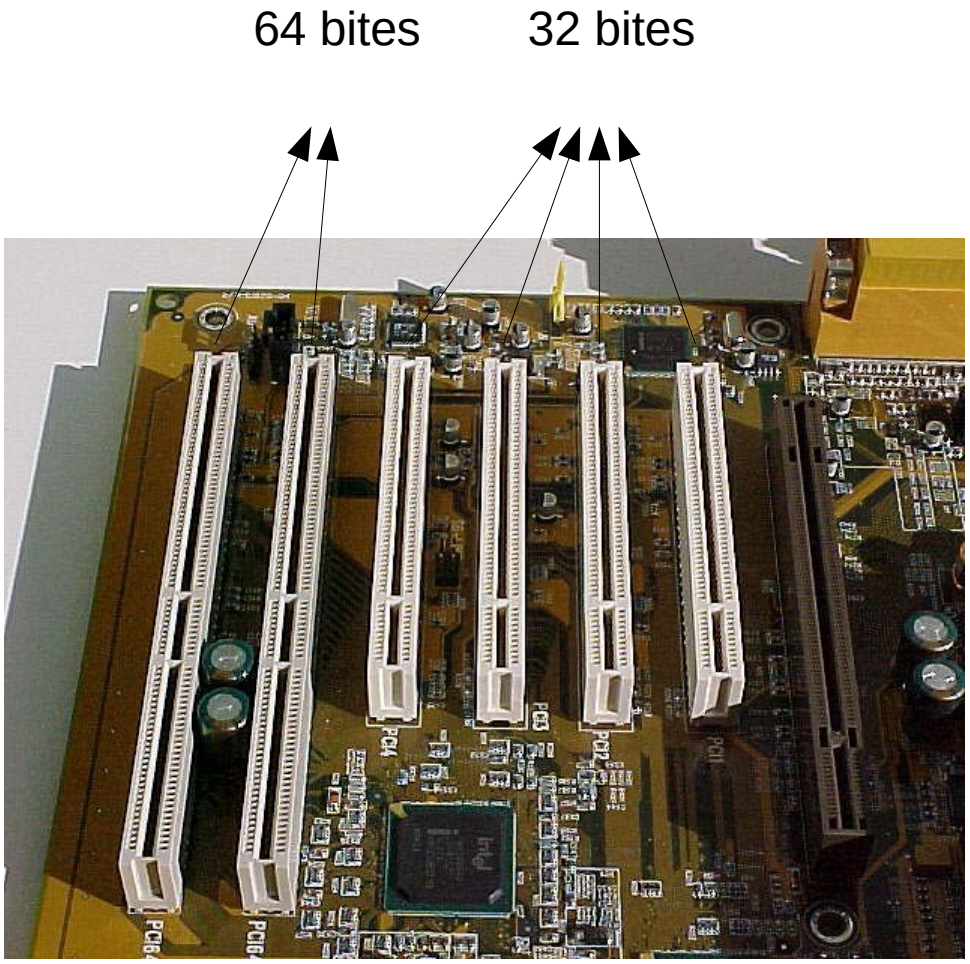
- Perifériánként 64 db 32 bites **konfigurációs regiszter**
  - 16 kötött: VendorID, DeviceID, Revision, Base Address Register 0...5
- Konfigurációs regiszterek címzése:
  - PCI host contrelleren keresztül
  - Hivatkozás egy regiszterre:



  - Host/PCI híd kijelöli a kívánt eszközt konfigurációra **dedikált vezetékkel**
  - Írás/olvasás tranzakció adja meg / állítja be a kívánt regisztert
- Rendszerindítás:
  - Eszközök szisztematikus lekérdezése
  - Ablakok kiosztása
  - Device driver betöltése
  - További eszközfüggő konfiguráció
  - ...

A PCI CSATLAKOZÓ

- Kötelezően használandó:
  - Órajel, Reset
  - A/D[0...31]
  - C/$\overline{\text{BE}}$[0...3]
  - $\overline{\text{FRAME}}$
  - $\overline{\text{IRDY}}$, $\overline{\text{TRDY}}$, $\overline{\text{STOP}}$
  - $\overline{\text{DEVSEL}}$, $\overline{\text{IDSEL}}$
  - Paritás, $\overline{\text{PERR}}$, $\overline{\text{SERR}}$
  - $\overline{\text{REQ}}$, $\overline{\text{GNT}}$
- Opcionális:
  - A/D[32...63]
  - C/$\overline{\text{BE}}$[4...7]
  - $\overline{\text{INTA}}$, $\overline{\text{INTB}}$, $\overline{\text{INTC}}$, $\overline{\text{INTD}}$
  - $\overline{\text{REQ64}}$, $\overline{\text{ACK64}}$
  - $\overline{\text{CLKRUN}}$

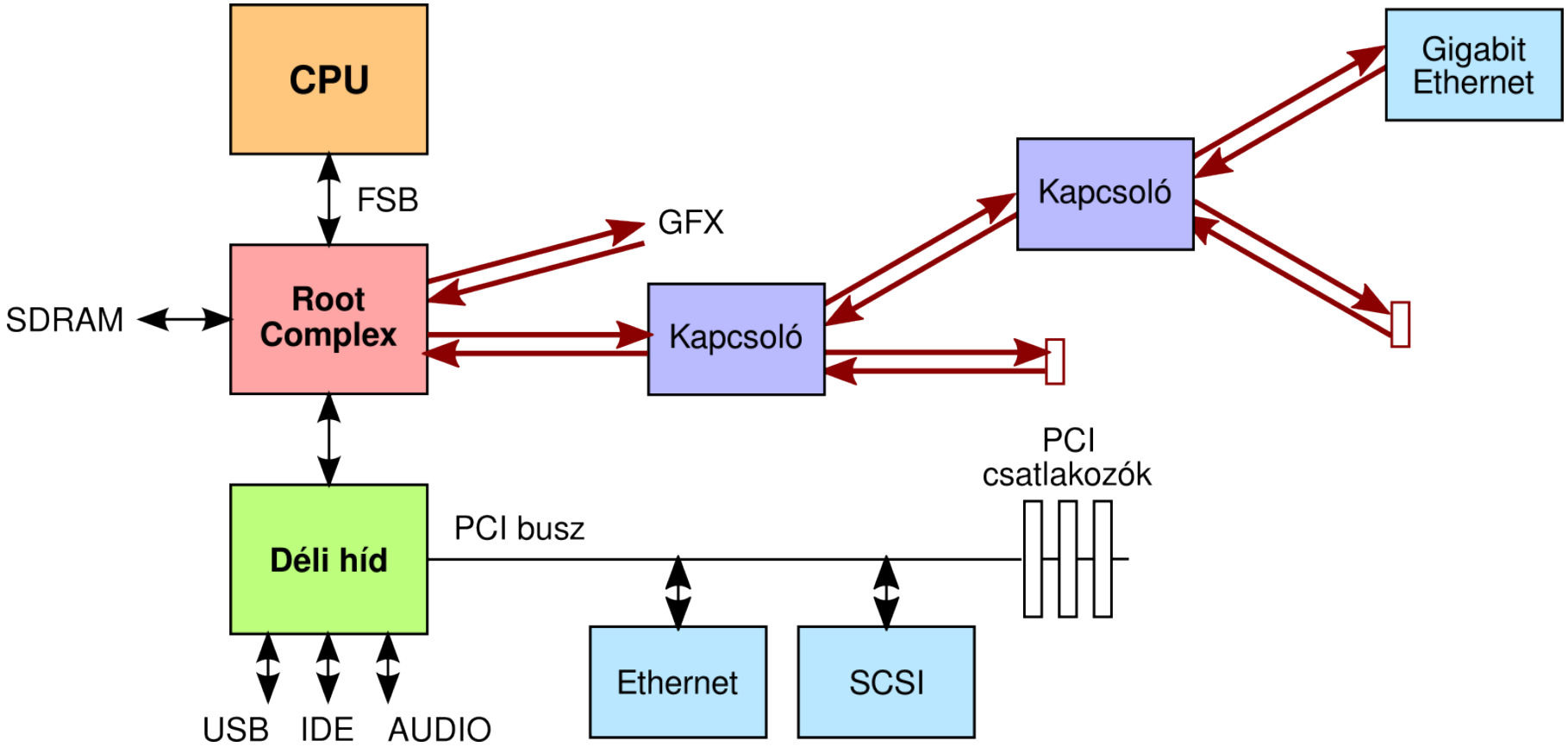


PCI Express

ÚJDONSÁGOK

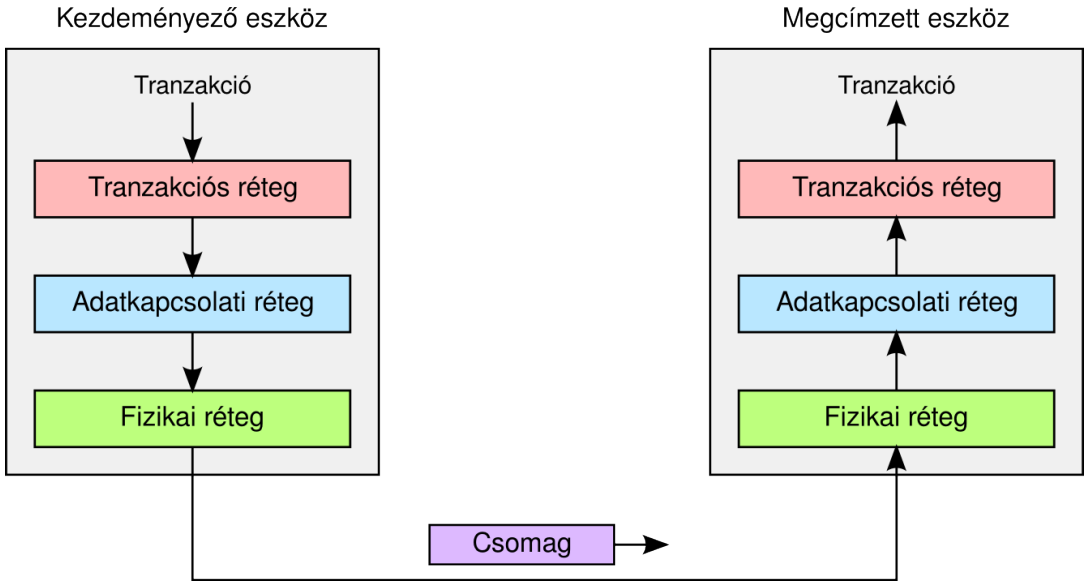
- Célok:
  - PCI-nál nagyobb adatátviteli sebesség
  - **Lehető legteljesebb szoftveres PCI kompatibilitás**
- Leglátványosabb eltérések:
  - **Pont-pont összeköttetések**
    - Nem osztott a közeg → nincs versengés, várakozás, éhezés, arbitráció
  - **Soros átvitel**
    - Órajel nincs (önidőzítő átvitel)
    - Egy érpár: 32 Gbit/s (PCIe 5.0)
    - Két eszköz között full duplex (kétirányú) kapcsolat:
      → 2 soros érpár
    - Két eszköz több full duplex soros kapcsolattal is össze lehet kötve
      - 1 full duplex soros kapcsolat neve: ***lane*** (pálya)
      - Szabvány: 1x, 4x, 8x, 16x, 32x lane
      - Több lane esetén a bitek párhuzamosan vihetők át
        - → **De nem szinkronban !!!**
        - 32x pálya esetén 32x32 Gbit/s = 128 GB/s

PCI EXPRESSZ HELYE A SZÁMÍTÓGÉPBEN



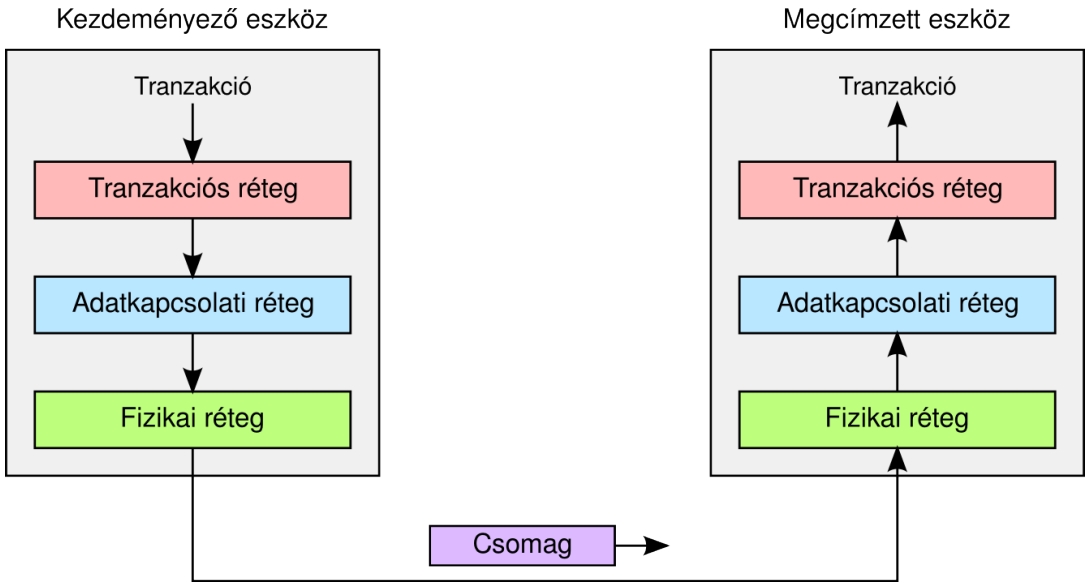
PCI EXPRESS MŰKÖDÉSE

- Csomagkapcsolt hálózat:
  - Eszközök kapcsolókon keresztül kötődnek a Root Complexhez
  - Tranzakciók csomagként utaznak

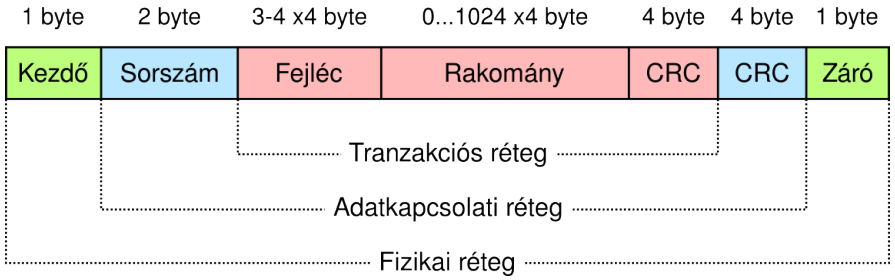


  - Tranzakciós réteg: tranzakció becsomagolás, címzés
  - Adatkapcsolati réteg: sorszámozás / újraküldés
  - Fizikai réteg: csomaghatárok, 8b/10b (128b/130b)

OVERHEAD



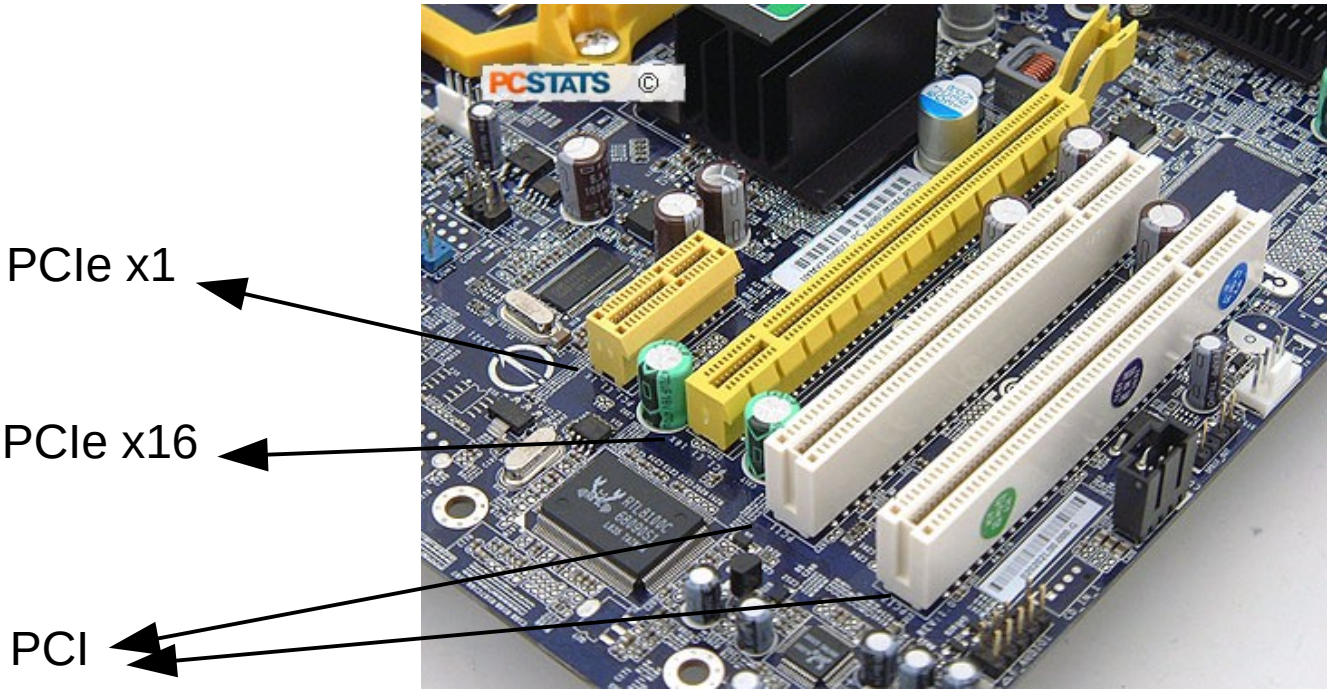
- Csomag formátuma:



- 1 db 32 bites adat átvitele: 1+2+3*4+1*4+4+4+1 = 28 bájt
  + 8b/10b overhead, = 35 byte !!!

INTERRUPT ÉS KONFIGURÁLÁS

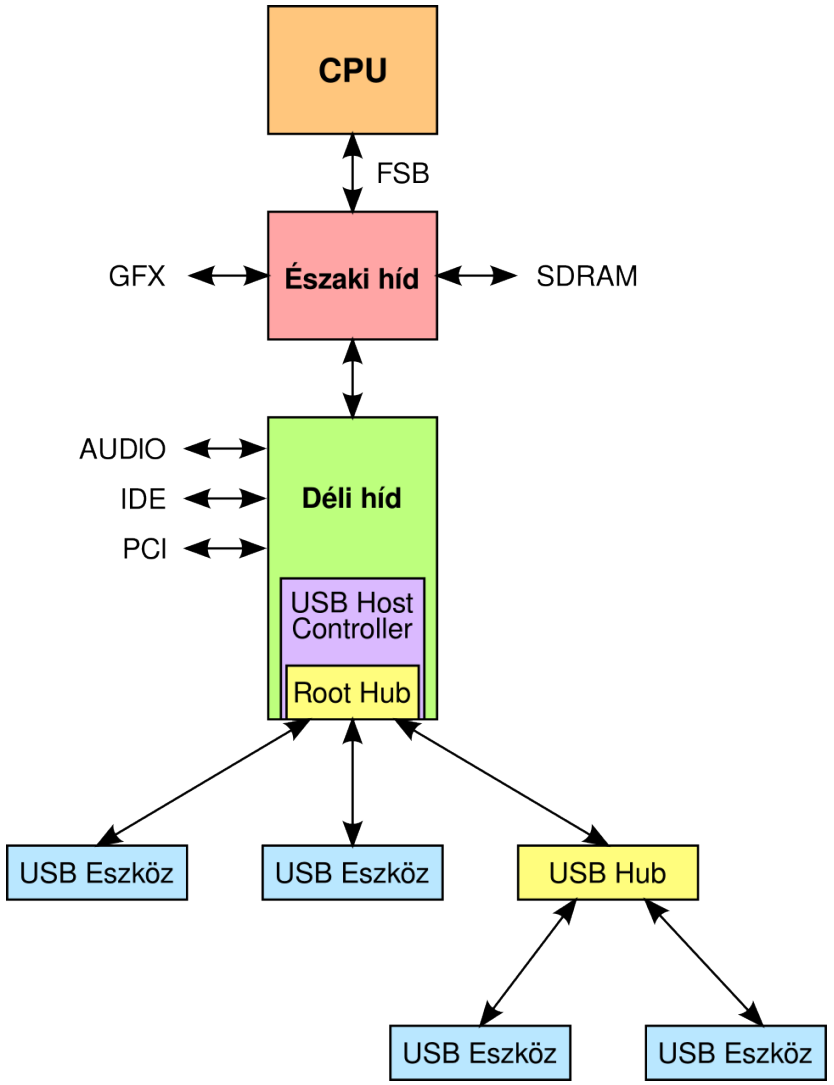
- Megszakításkezelés
  - Üzenet alapú (MSI) → mint a PCI-ban
  - Jelvezeték alapú: emulált
- Konfiguráció:
  - Több konfigurációs regiszter (1024)
  - Első 64 írása/olvasása: mint PCI-ban (teljes kompatibilitás)



USB

AZ USB FELÉPÍTÉSE

- Fa topológia
  - Levelek: az **USB perifériák**
  - Csomópontok: **USB hub-ok**
  - Gyökér: **root hub**
    (a host controller része)
- Periféria lehet:
  - Egyszerű eszköz
  - Összetett eszköz:
    - Fizikailag egy eszköz
    - Logikai több eszköz



MITŐL BUSZ

- Megszoktuk, hogy a busz osztott közeg
  - Minden szereplő mindent hall
- De **ez egy fa**!
- **Mégis busz**
  - Egyetlen master van, a root hub
  - Amit a master beszél, azt mindenki hallja (terjed a fában)
  - Akire vonatkozik, az odafigyel, a többi nem
  - Amit a periféria mond, az a fában felfelé haladva éri el a mastert
- Nem kell arbitráció!
  - Mivel csak 1 master van
  - Senki nem szólalhat meg, csak ha a master erre utasítja
- Ha busz, akkor megvan a hátránya is:
  - Minél többen csatlakoznak rá, annál lassabb lesz

TRANZAKCIÓK ÉS KERETEK

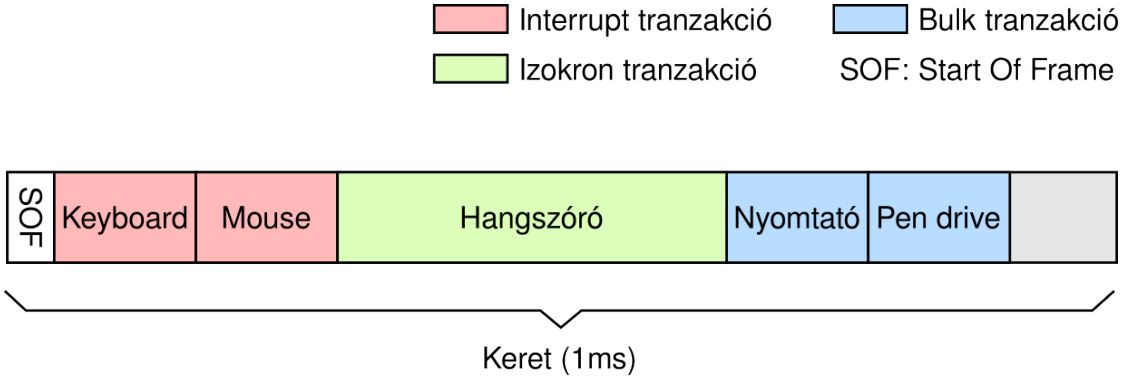
- A perifériákkal **tranzakciók** segítségével kommunikálunk
- Mindig a master, a root hub indítja
- Tartalma:
  - Megcélzott periféria azonosítója
  - Ki- vagy bemeneti művelet
  - Átvinni kívánt adatok
    - Kimeneti irány esetén a root hub teszi a buszra
    - Bemeneti irány esetén a megcélzott periféria
- A tranzakciók **keretekben** utaznak
  - Minden keret adott ideig tart
    - USB 1.1 full speed: 1 ms / keret, 1500 bájt rakomány
    - USB 2.0 high speed: 125 µs / keret, 7500 bájt rakomány
  - Minden keret több tranzakciót is szállíthat
  - Csak *teljes* tranzakciókat szállíthat (lehet, hogy nem lesz kitöltve)

ADATÁTVITELI MÓDOK

- **Adatátviteli módok**: hogyan vigyük át az adatot
  - **Bulk**
    - Nagy tömegű adat hibamentes átvitele
    - Késleltetés-garancia nincs
    - Pl.: külső háttértár, nyomtató, stb.
  - **Izokron**
    - Késleltetésérzékeny perifériáknak valós idejű átvitel
    - Bithibamentesség nem garantált
    - Pl.: USB mikrofon, web kamera
  - **Interrupt**
    - USB-ben nincs interrupt → pollinggal szimuláljuk!
    - Hibamentesség + késleltetés garancia
    - Pl.: beviteli eszközök
  - **Control**
    - Hibamentesség + késleltetés garancia
    - Szabványban rögzített üzenet formátum és struktúra

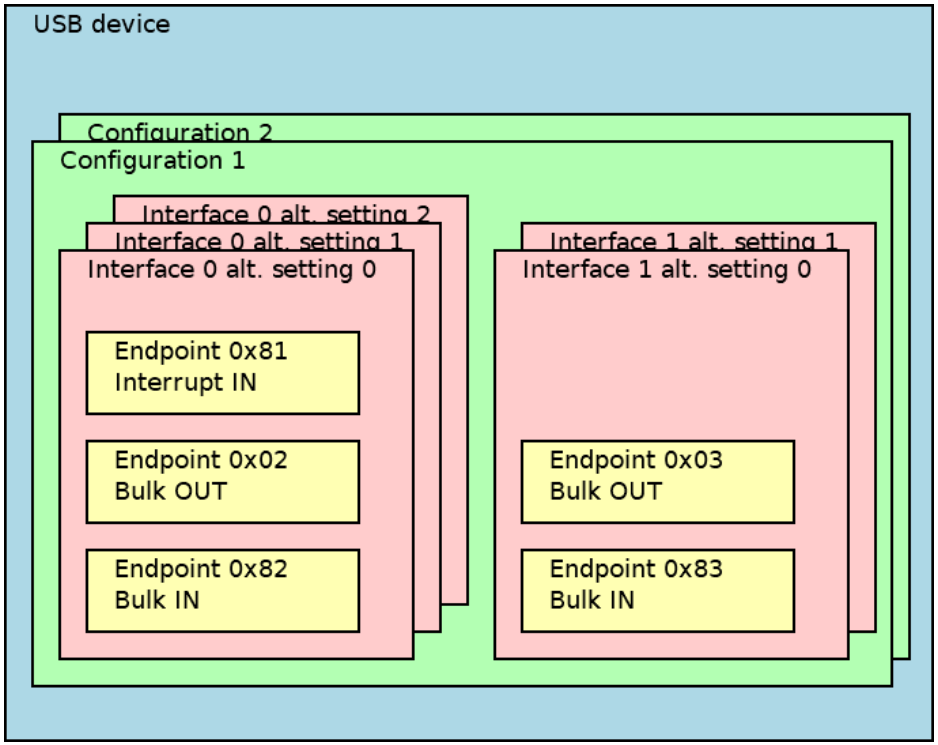
A KERETEK KITÖLTÉSE

- Gazdálkodás a sávszélességgel
  - Az izokron és interrupt tranzakcióknak elsőbbsége van
    - De nem foglalhatnak többet a keret 90%-ánál
    - Ha csatlakozik egy új periféria, amivel több lenne 90%-nál
      → nem engedik belépni
  - A fennmaradó 10%-ban elsőbbsége van a control tranzakcióknak
  - Maradék: bulk. Ha marad.

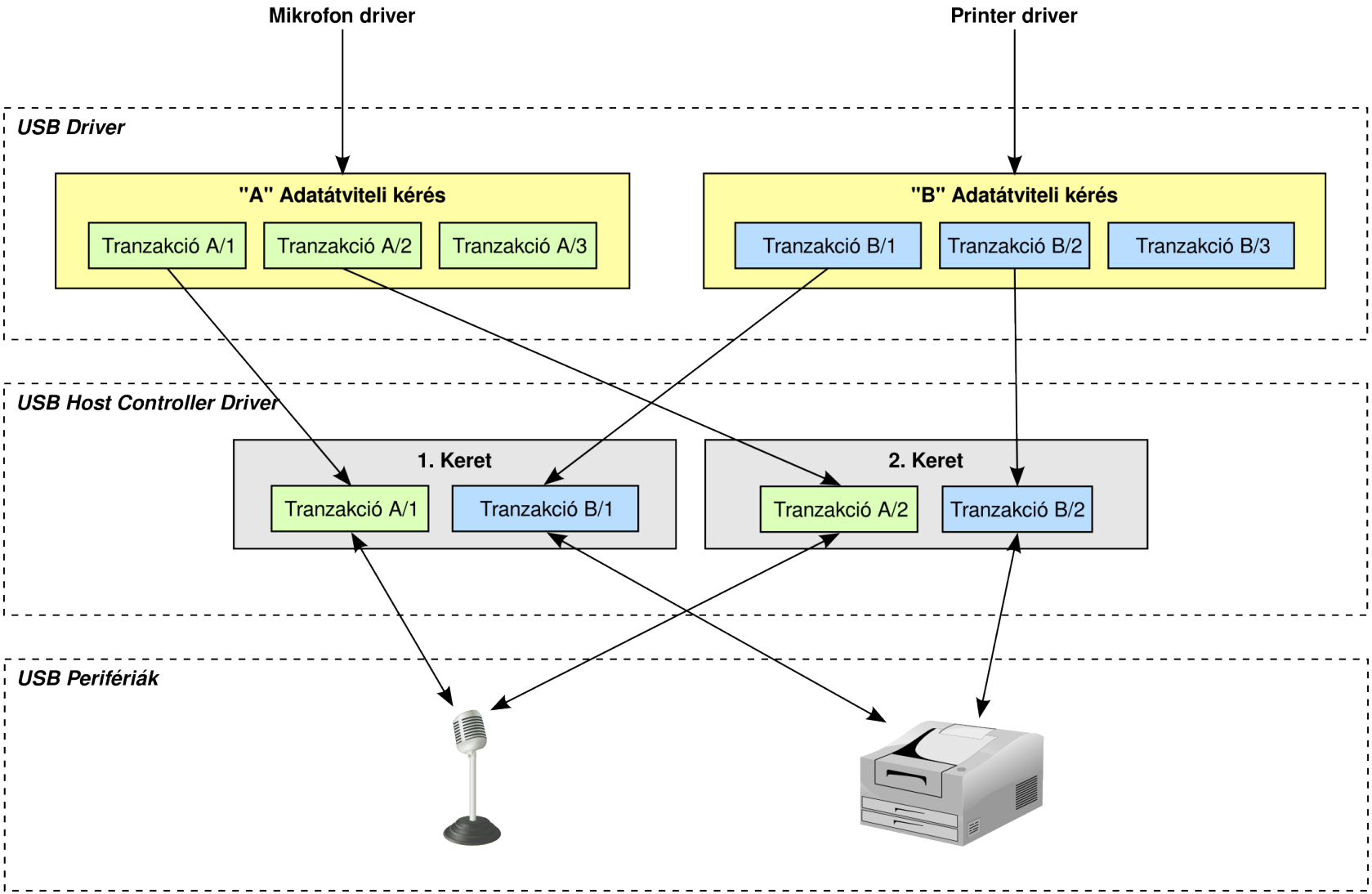


USB ESZKÖZÖK FELÉPÍTÉSE

- 1 USB eszközben több **interface** lehet
  - logikai periféria, ≈ PCI function
  - pl.: USB modem + pendrive a driverrel
- 1 interfésznek több adatátviteli csatornája (**endpoint**)
  - Egyirányú és saját címe van (7 bit cím + 1 bit irány)
  - A 4 átviteli mód egyikét használja
- Interfészenként lehetnek alternatív beállításai (**alt setting**)
  - Pl. web kamera különböző felbontásokban, különböző izokron sávszélességet használva
- Az egész USB eszköznek lehet több konfigurációja (**configuration**)
  - Pl. hálózati adapter Ethernet-over-USB vagy Microsoft NDIS üzemmódot használ-e



AZ ADATÁTVITEL FOLYAMATA



KONFIGURÁLÁS

- Rendszerindításkor, új eszköz csatlakozásakor
- A hub érzékeli, hogy új eszköz csatlakozott
- Az USB driver időnként végigkérdezi a hub-okat, jött-e új periféria az egyik portra. Ha jött, konfigurálja őket:
  1) Szól a hub-nak, hogy reset-elje a portot
     Reset hatására a periféria a 0-s címre hallgat
  2) Kiolvassa a periféria eszközleíró táblázatait
     (0-s címre küldött control tranzakciókkal)
  3) Benne vannak a gyártó, típus, eszköz jellege, paraméterei, stb.
  4) Egyedi címet ad a perifériának
     (0-s címre küldött control tranzakcióval)
  5) Ellenőrzi, beléphet-e:
     - Izokron és interrupt adatátviteli igénye kielégíthető-e
     - Kielégíthető-e a tápellátási igénye
  6) Ha minden rendben, az új eszköz beléphet az adatforgalomba

VERZIÓK

- USB 1.1:
  - **LS**: low speed, 1.5 Mbit/s
  - **FS**: full speed, 12 Mbit/s
- USB 2.0:
  - **HS**: high speed, 480 Mbit/s
  - LS, FS megmaradt (kompatibilitás nem triviális)
  - Kisebb különbségek
    - A forgalomszabályozásban
    - A csomagméretekben
- USB 3.0:
  - **SS**: super speed, 5 Gbit/s
  - Teljesen más struktúra!
- USB 3.1:
  - **SS+**: super speed+, 10 Gbit/s
- USB 3.2:
  - Két lane segítségével még magasabb sebesség (20 Gbit/s)
- USB 4:
  - Adatátvitel a PCIe fizikai rétegével, USB Type-C csatlakozóval (40 Gbit/s)
  - USB 2.0/3.0/3.1/3.2 kompatibilitás

USB 3.0 SUPERSPEED TRANZAKCIÓK

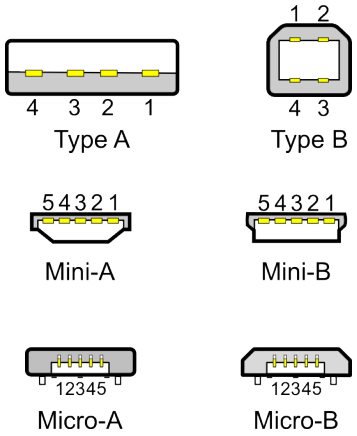
- Ami nem változott:
  - Fa topológia, hub-okkal a csomópontokban
  - A 4 adatátviteli mód megmaradt
  - Ugyanúgy kell az eszközöket konfigurálni
  - Továbbra is a root hub az egyetlen master
    → Szoftver szinten nincs változás
- Változások:
  - Fizikai szinten szinte minden. Teljesen új rendszer.
  - Kompatibilitás: az USB 3.0 tulajdonképpen két külön busz egymás mellett
    - Egy USB 2.0-ás busz az LS, FS és HS tranzakcióknak
    - Egy külön busz az SS tranzakcióknak
  - **Megszűnt az üzenetszórás**
    - Forgalomirányítással érnek célba az adatok
    - A hub-ok csak arra továbbítják a tranzakciót, amerre a címzett van
  - **Store and forward csomagtovábbítás**
    - A hub megvárja, amíg teljesen megjön hozzá a tranzakció, csak ezután küldi tovább
  - **Egy helyett két érpáron zajlik a kommunikáció**
    - Oda-vissza irányba párhuzamosan
    - A PCI Express-nél látott módon (bitkeverés, 8-ból 10-be kódolás, stb.)

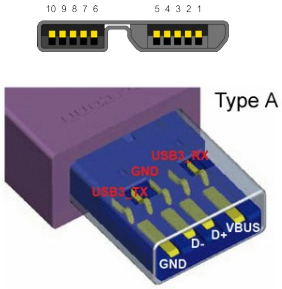
TÁPELLÁTÁS

- A jelvezetékek mellett „föld” és „+5V” vezeték is van
  → Perifériák ebből is megoldhatják a tápellátásukat!
- Áramfelvétel:
  - Maximum:
    - USB 1.0 – 2.0: max. 500 mA,
    - USB 3.0: max. 900 mA
  - A konfigurációs fázisban max. 100mA-t vehetnek fel
  - A konfigurációs fázisban közlik, mennyi áramot igényelnek
  - Ha nincs ennyi, nem léphetnek be a rendszerbe

KÁBELEK, CSATLAKOZÓK

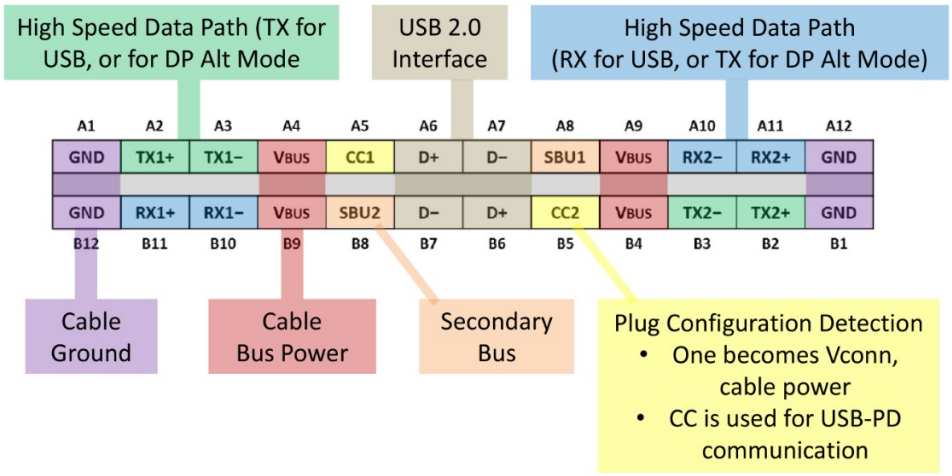
- Kábel: 4 vezeték
  - Táp: 0V, 5V
  - Jelátvitel: D+, D–, differenciális, soros átvitel
- USB 1.1 csatlakozók
  - Type A: periféria a hubhoz
  - Type B: hub a perifériához (ha külön van)
- USB 2.0 csatlakozók
  - Új méretek: mini, mikró
  - Van aljzat, amibe az „A” és „B” csatlakozó is belemegy
    - „A”-t beledugva host-ként viselkedik
    - „B”-t beledugva egyszerű periféria
  - 5. láb: „szerep”: földelve host, nyitva periféria
- USB 3.0/3.1:
  - Kábel: benne fut az USB 2.0 csavart érpárja,
    és az USB 3.0 két csavart érpárja is → jóval vastagabb (drágább)
  - 2 részre bontották: USB 2.0 és USB 3.0 részre
- USB 3.2/4.0:
  - Type-C csatlakozó





TYPE-C KÁBEL

- Type-C kábel:
  - Tetszőleges irányból csatlakoztatható
  - 24 láb (2x12 a kétirányú kialakítás miatt)
  - Ugyanolyan a csatlakozó a master és a slave számára
    - A CC-n (configuration channelen) lebeszélik a szerepeket
  - Vezetékek:
    - USB 2.0 vonalak
    - USB 3 SuperSpeed/SuperSpeed+ vonalak (duplex!), 2x
    - „SideBand” vezetékek jövőbeli célokra
      - pl. analóg audio
    - Alternatív mód: az SS+ és a SideBand más adatátviteli technológiák támogatására is használható
      - Példa: DisplayPort, Thunderbolt, HDMI
  - A kábel aktív szereplő lehet
    - Részt vesz a szerepek kiosztásában, és a támogatott (nem USB) átvitel lebonyolításában
    - Kábel tápja: CC-n keresztül
- USB tápellátás Type-C kábellel
  - Áram 5A-ig, feszültség 20V-ig (ez 100W!!!)
  - A kábel aktívan részt vehet a feszültség és az áramigény egyeztetésében



HÁLÓZATI RENDSZEREK
ÉS SZOLGÁLTATÁSOK
TANSZÉK

HÁLÓZATI RENDSZEREK ÉS SZOLGÁLTATÁSOK TANSZÉK

MŰEGYETEM 1782

Budapest,
2022.03.08.

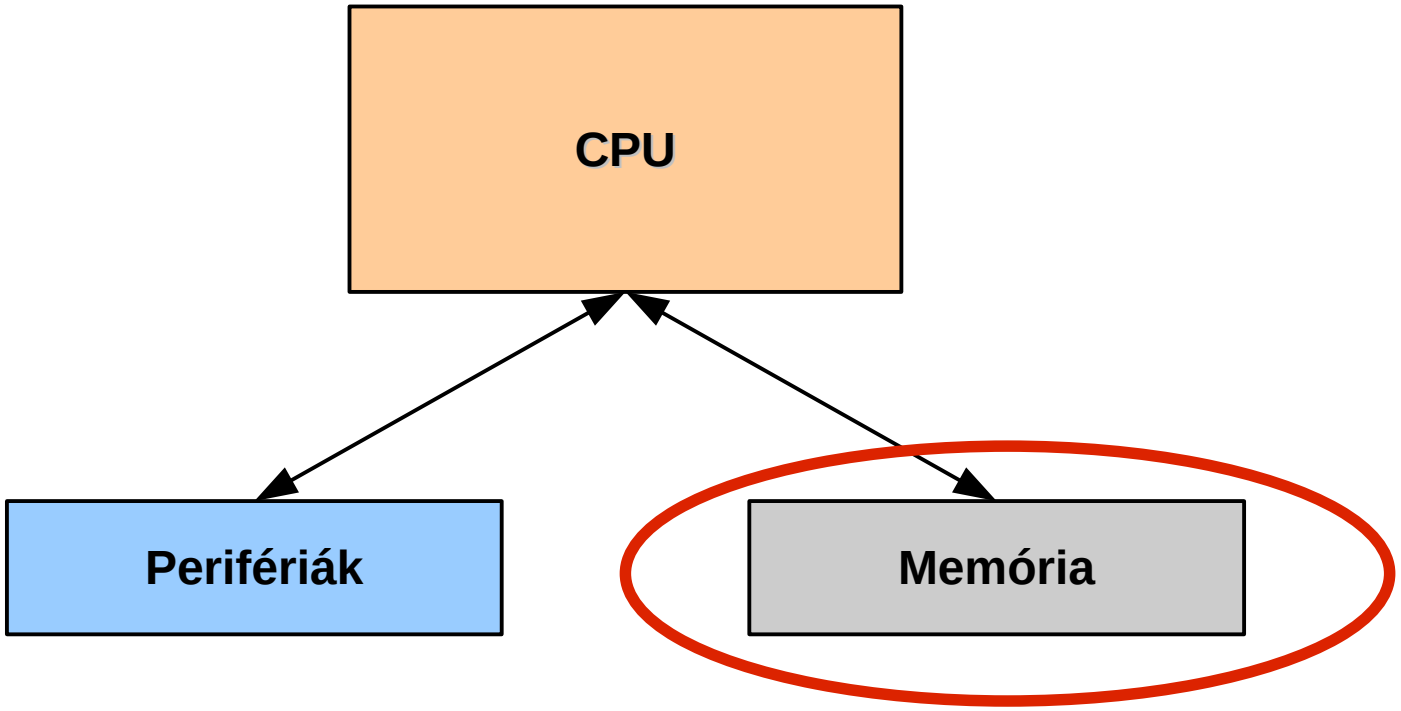
SZÁMÍTÓGÉP ARCHITEKTÚRÁK

Memória technológiák

Horváth Gábor, Belső Zoltán

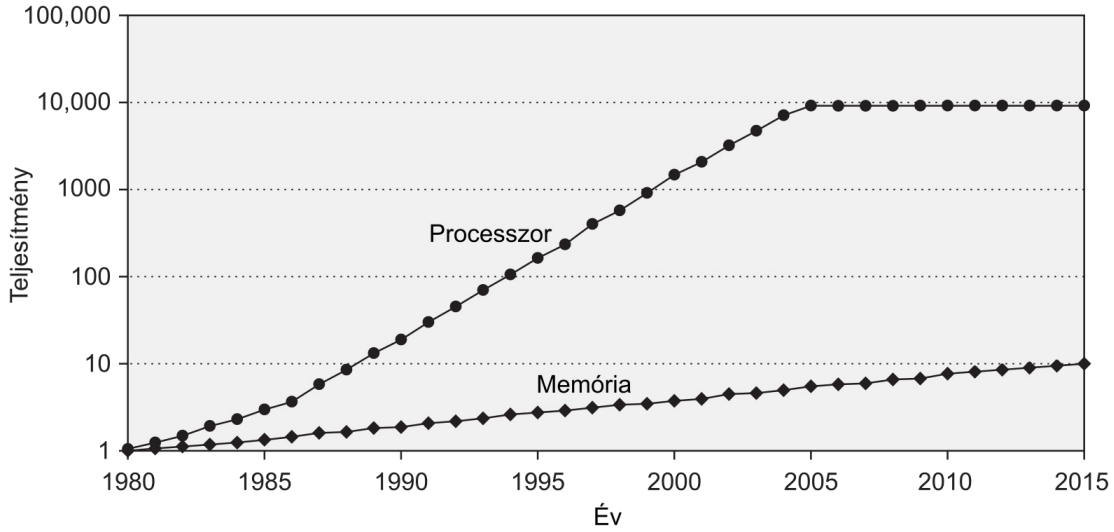
BME Hálózati Rendszerek és Szolgáltatások Tanszék
ghorvath@hit.bme.hu, belso@hit.bme.hu

HOL TARTUNK?



MIT TANULUNK A MEMÓRIÁKRÓL?

- Motiváció: Neumann-architektúrában szűk keresztmetszet



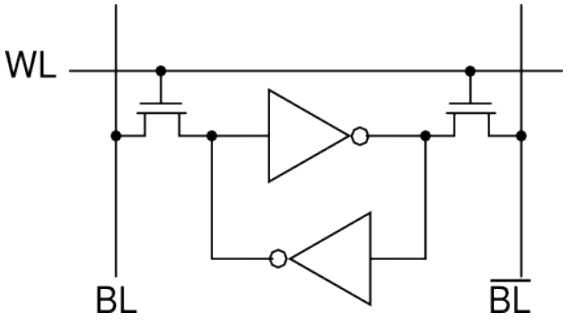
- Témakörök:
  - Memóriatechnológiák
    - DRAM, SRAM, memóriamodulok, többcsatornás memória-vezérlők, időzítések, DDR1-2-3-4-5, GDDR, stb.
  - Virtuális tárkezelés
    - Ha a valós memória mennyiségét a programok elől el akarjuk takarni
  - Gyorsítótár (cache memória)
    - A gyakran használt tartalmak elhelyezése egy gyors, de kicsi tárban

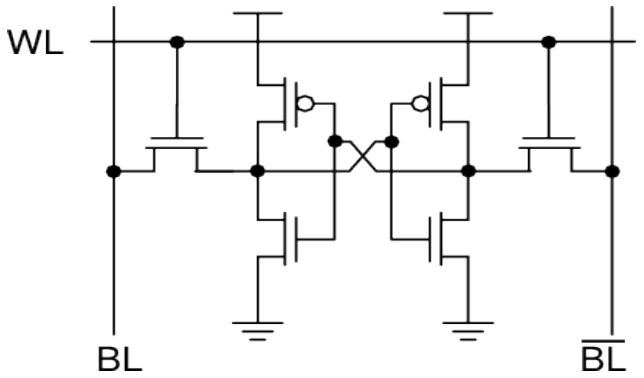
MEMÓRIA TECHNOLÓGIÁK

- Egyetlen bit tárolása:
  - SRAM-mal
  - DRAM-mal

STATIKUS RAM (SRAM)

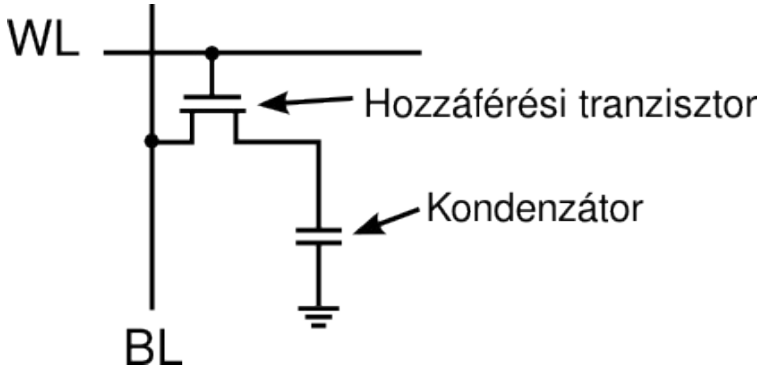
- 1 bit tárolása: két szembekötött inverter
  - Olyan, mint egy flip-flop!
  - **1 bit → 6 tranzisztor**: 6T
    - 2 hozzáférési: BL-ra és $\overline{BL}$-ra köti a flip-flopot
    - 4 a flip-flop megvalósítása
- **Olvasás**:
  - WL-re logikai 1
  - Bit értéke: BL-re, negáltja: $\overline{BL}$-re
  - Érzékelő erősítők: BL és $\overline{BL}$ különbségét figyelik
- **Írás**:
  - BL-t és $\overline{BL}$-t a tárolni kívánt bit értékre kényszerítjük
  - WL-re logikai 1
  - A kényszerítés erősebb, mint az inverterek





DINAMIKUS RAM (DRAM)

- 1 bit tárolása: kondenzátorral
  - Töltött: bit=1, üres: bit=0
  - + kell 1 hozzáférési tranzisztor
  - **1 bit → 1 tranzisztor + 1 kondenzátor**: 1T1C
- **Olvasás**:
  - BL-t 0 és 1 közé félútra töltjük (precharge, előfeszítés)
  - WL-re logikai 1
  - Érzékelő erősítők: Merre változik a BL szintje?
    - Nő: bit=1
    - Csökken: bit=0
  - Kifolyt a töltés a kondenzátorból!
    → az olvasás destruktív
    → a végén vissza kell írni bele a kiolvasott bitet
- **Írás**:
  - WL-re logikai 1
  - A BL-en keresztül feltöltjük a kondenzátort, vagy kiengedjük a töltését
- **Frissítés**:
  - Magától is elszivárog a kondenzátor töltése!
  - Frissíteni kell (néhány 10 ms-onként) → kiolvasás + visszaírás



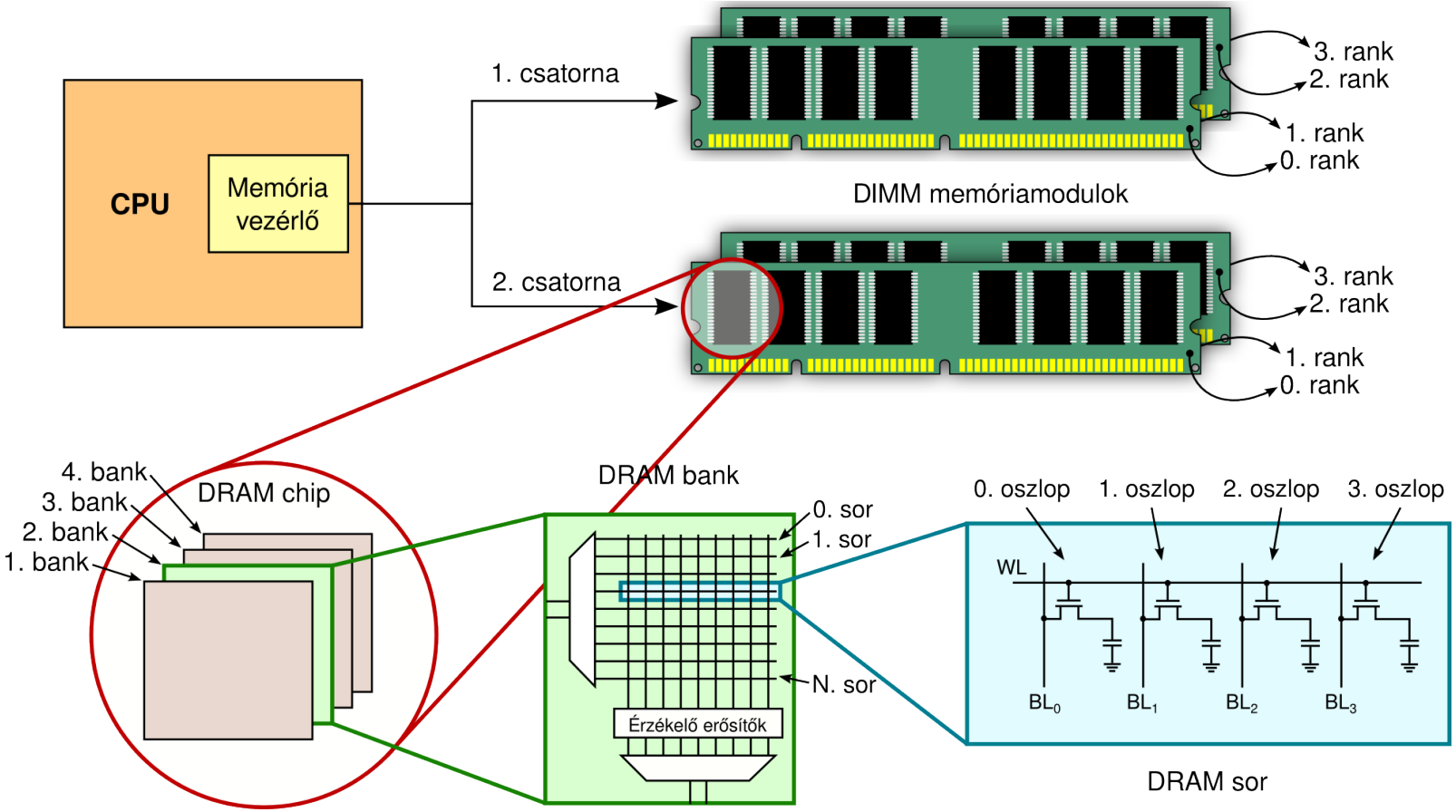
SRAM VS. DRAM

- Melyik a *gyorsabb*?
  - SRAM: tranzisztor hajtja a bitvezetéket
  - DRAM: a kondenzátorból kitévelygő elektronok állítják a bitvezetéket
  - … és a DRAM-ot még frissítgetni is kell!
  - **Gyorsabb: SRAM**
- Melyiknek nagyobb az *adatsűrűsége*?
  - SRAM: 6T
  - **DRAM:** 1T1C → **nagyobb adatsűrűség**
- Mire használják?
  - SRAM: cache
  - DRAM: rendszermemória
- Melyiket lehet a CPU mellé integrálni?
  - SRAM: könnyen. Csupa tranzisztor, mint a CPU.
  - DRAM: nehezen. Kondenzátort kell csinálni (CPU-hoz nem)
  - eDRAM: CPU mellé tett DRAM
    - 2009, POWER7, 32MB L3 cache eDRAM-ból
    - 2013, Intel Haswell GT3e, 128MB L4 cache eDRAM-ból
    - Konzolok: PlayStation 2, PlayStation Portable, Nintendo Wii U, stb.

DRAM alapú rendszermemóriák

ÁTTEKINTŐ KÉP

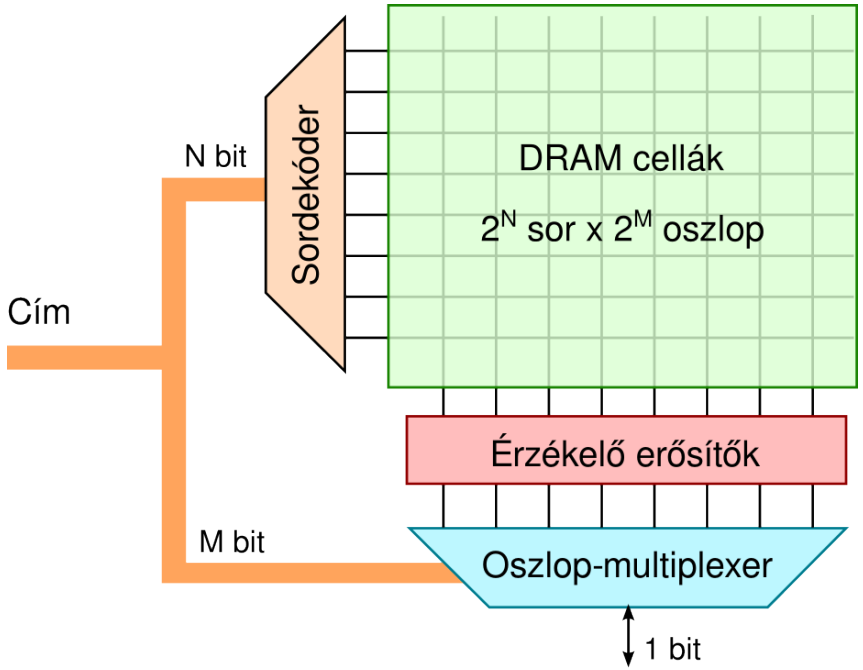
- Hogy csináljunk DRAM cellából memóriát?
  - Úgy, hogy nagy, olcsó legyen, kicsi legyen a késleltetése, nagy legyen az átviteli sebessége



DRAM BANK

Cellák mátrixban

- Egy sor: szóvezetékek összekötve
- Egy oszlop: bitvezetékek összekötve



- Olvasás:
  - A **sordekóder** kijelöl egy sort
  - A sor összes bitjét kiolvassák az **érzékelő erősítők**
  - A sorból a kívánt oszlopot kiválasztja az **oszlop-multiplexer**

- Két-fázisú műveletek:
  - Spórolunk a címbusz szélességével
  - Címbusz: sorcím → várunk → címbusz: oszlopcím → adatbusz: megjelenik az adat

KITÉRŐ

- Hogy lesz lineáris címből sor és oszlopcím?
- Példa: decimális rendszerben (!)
  - Memória kapacitás: 1 millió (cím: 0 … 999999)
  - Tárolómező: 1000x1000

- Lineáris cím: 123456
- Sor és oszlopcím: **123**|**456** → 123. sor, 456. oszlop
- Csak „vágni” kell, osztani nem!

- Bináris rendszerben ugyanez bitekkel

DRAM PARANCSOK

- Az 5 legfontosabb:
  - **ACTIVATE**
    - Megnyit egy sort (adatok → érzékelő erősítőkbe)
  - **READ**
    - A nyitott sorból olvas egy oszlopot
    - Igazából az érzékelő erősítőkből olvas
  - **WRITE**
    - A nyitott sorba ír egy oszlopot
    - Igazából az érzékelő erősítőkbe ír
  - **PRECHARGE**
    - Bezárja a nyitott sort
    - Előfeszíti a bitvezetékeket is, hogy a köv. nyitás gyors legyen
  - **REFRESH**
    - Frissít egy sort
    - Majdnem egy sormegnyitás+lezárás
      - De ennek nem kell sorcím. Auto-inkremens.

DRAM PARANCSOK

- Példa olvasási kérések:

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

- Parancsok (tfh. előfeszített):

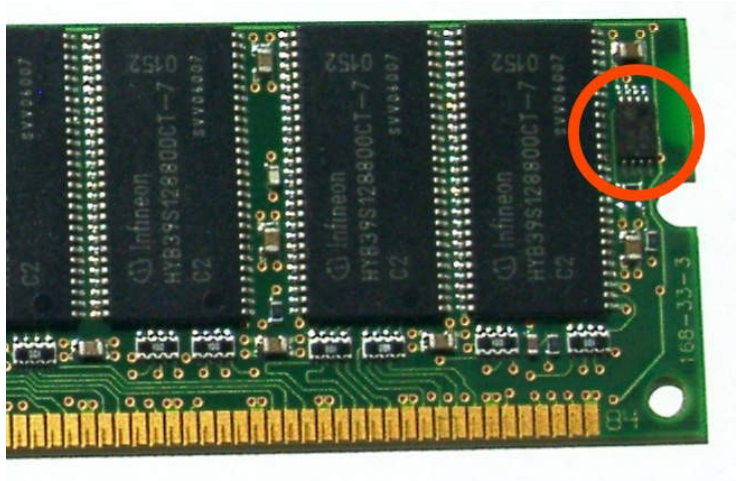
```
ACTIVATE 3
READ 8
READ 14
PRECHARGE
ACTIVATE 1
READ 3
READ 4
```

DRAM IDŐZÍTÉSEK

- A parancsok végrehajtási ideje
- 4 legfontosabb:
  - T_{RCD}: A sor megnyitás ideje
  - T_{CAS} (CL): oszlopcím ráadásától az adat megjelenéséig tartó idő
  - T_{RP}: Előfeszítés (PRECHARGE) ideje
  - T_{RAS}: Minimális sor nyitvatartási idő
- Sok-sok ilyen van még
- Mértékegység: órajel (szikron DRAM), ns (aszinkron)
- Boltban veszünk memóriát 8-9-10-11 időzítéssel:
  - T_{CAS}=8, T_{RCD}=9, T_{RP}=10, T_{RAS}=11
- Ha csak az van ráírva, hogy CL7: T_{CAS}=7

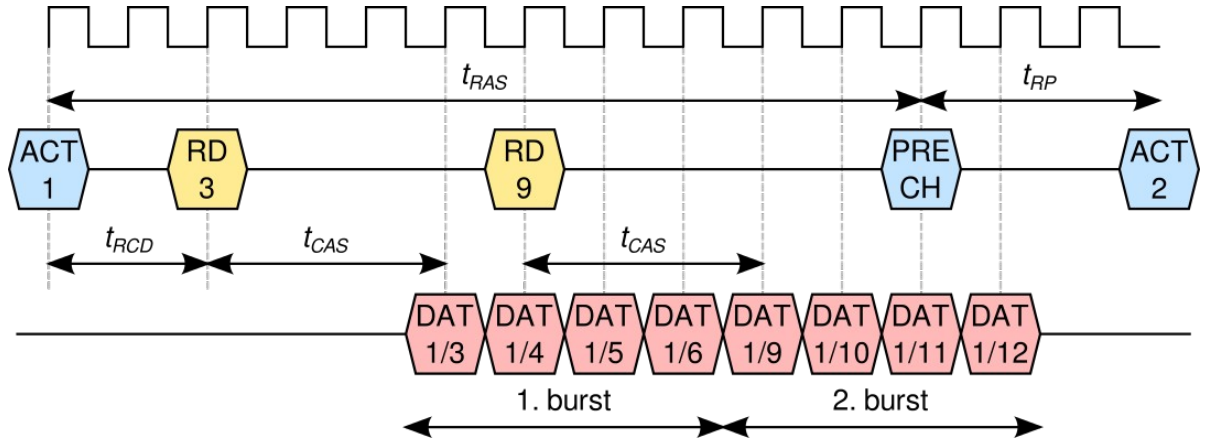
DRAM IDŐZÍTÉSEK

- Honnan tudja az időzítéseket a memóriavezérlő?
  - Megkérdezi a memóriamodultól
  - Memóriamodul tartalma:
    - DRAM chip-ek
    - ...és egy **SPD** chip! Ebben vannak a jellemzők.



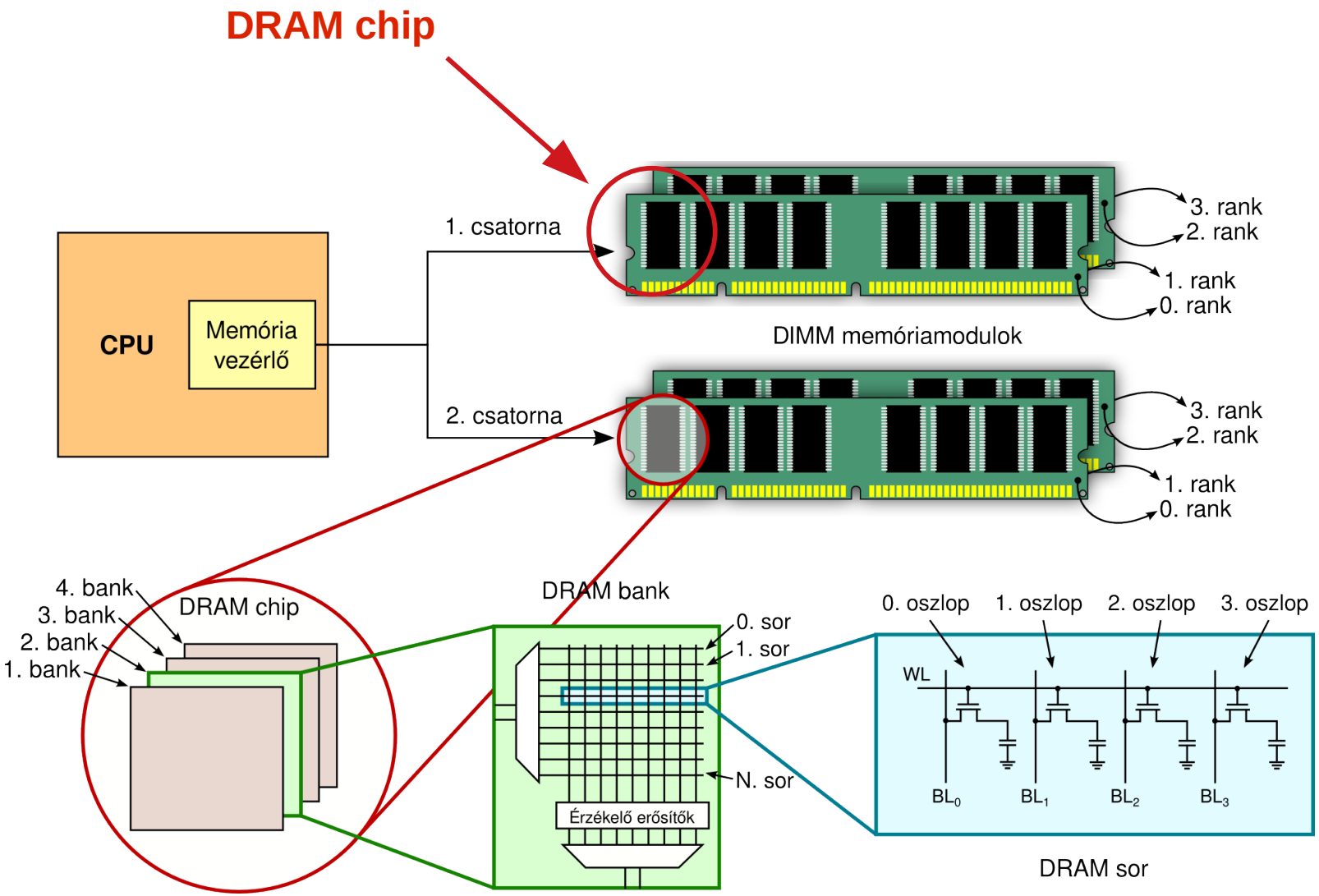
HATÉKONYSÁG JAVÍTÁSA

- A sor minden kiolvasott oszlopánál:
  - Oszlopcím → T_{CAS} → adat elvétele
  - Ez túl pazarló!
- **Burst mód**
  - Egyszer adok rá oszlopcímet
    … nem egy oszlopot ad, hanem egy egész sorozatot!
  - Burst hossz: konfigurációs paranccsal beállítható



- **Átlapolt parancskiadás és adatátvitel**
  - Nem kell megvárni az előző parancsra a választ

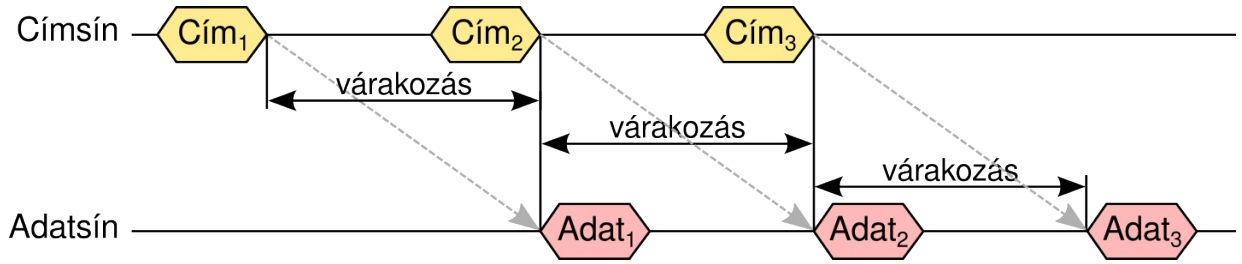
DRAM CHIP-EK


DRAM chip
CPU
Memória vezérlő
1. csatorna
2. csatorna
3. rank
2. rank
1. rank
0. rank
DIMM memóriamodulok
4. bank
3. bank
2. bank
1. bank
DRAM chip
DRAM bank
0. sor
1. sor
N. sor
Érzékelő erősítők
0. oszlop
1. oszlop
2. oszlop
3. oszlop
WL
BL0
BL1
BL2
BL3
DRAM sor

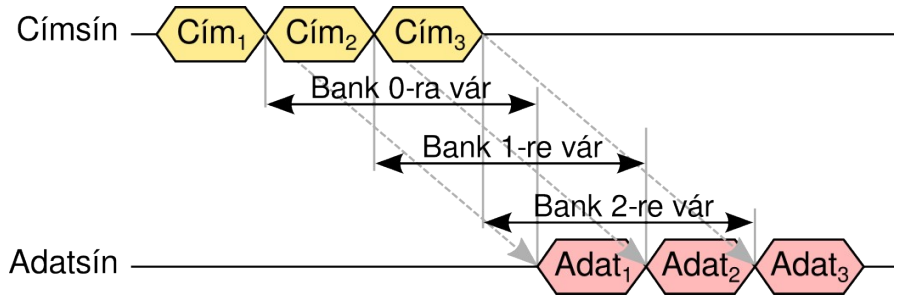
DRAM CHIP-EK

- Egy DRAM chip bankokból áll
- **Bank**:
  - Független DRAM cella mátrixok
  - Saját sordekóder, érzékelő erősítő, oszlopmultiplexer
  - Mindegyikben lehet 1-1 nyitott sor
    → Egy DRAM chip-ben több nyitott sor lehet
    → **Átlapolt adatátvitel!**

- Átlapolás nélkül:



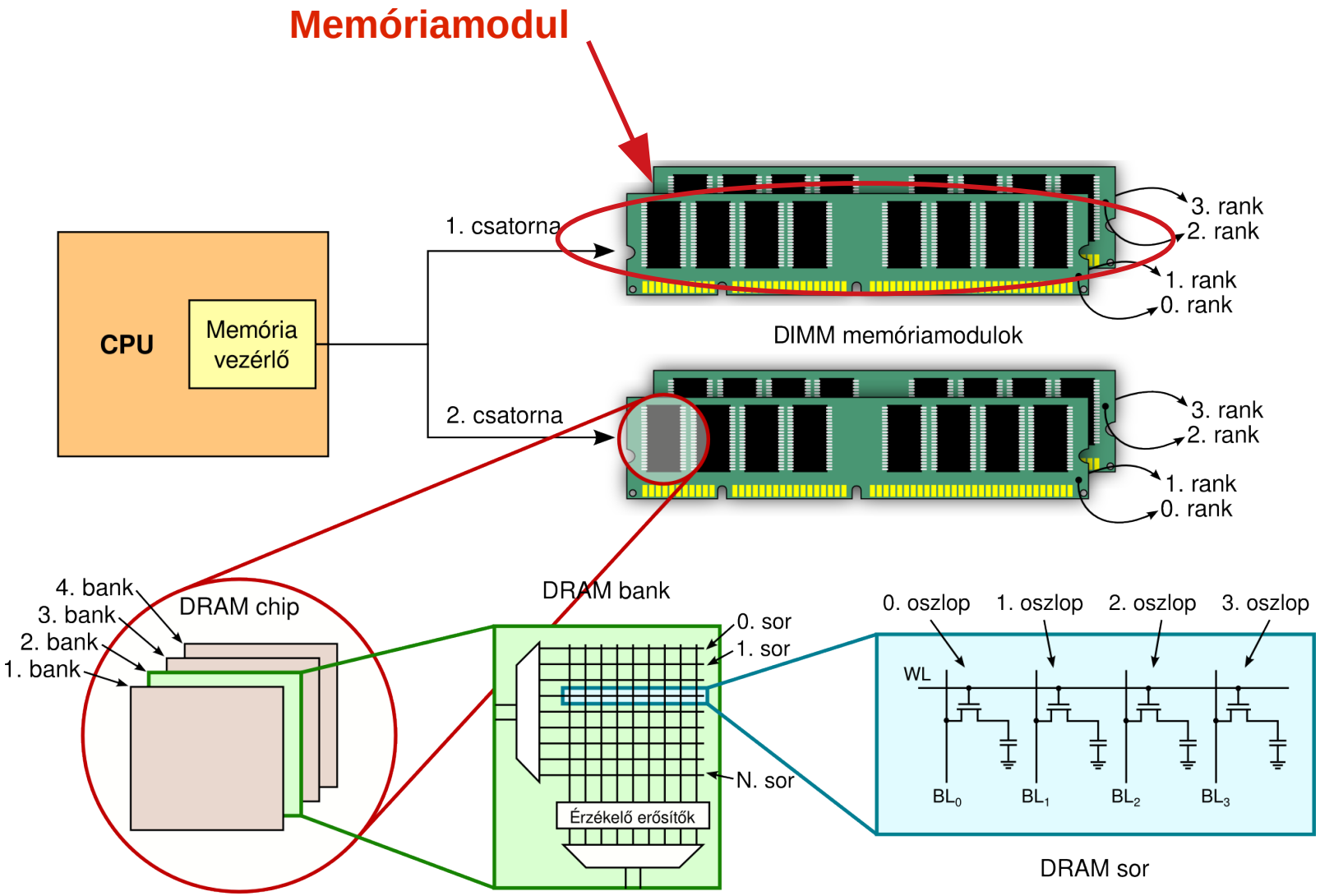
- Átlapolással:



DRAM CHIP-EK

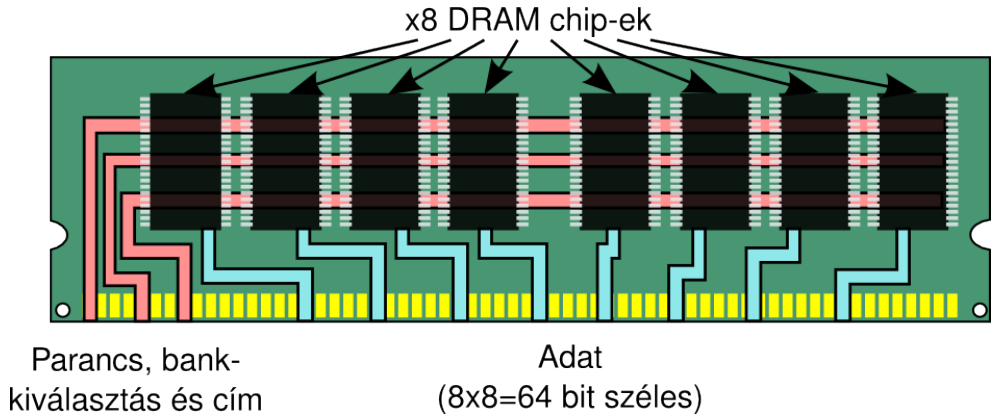
- Bankok nem biteket tárolnak
  - 1 oszlop: 4 bit, 8 bit, 16 bit (x4, x8, x16)
- Interfész:
  - **Parancs vezetékek**
    - Mit kell csinálnia
  - **Bank kiválasztó vezetékek**
    - Melyik bank-ra vonatkozik
  - **Címvezeték**
    - ACTIVATE: sorcím
    - READ/WRITE: oszlopcím
  - **Adatvezetékek**
    - 4, 8, vagy 16 bit

DRAM MEMÓRIAMODULOK



DRAM MEMÓRIAMODULOK

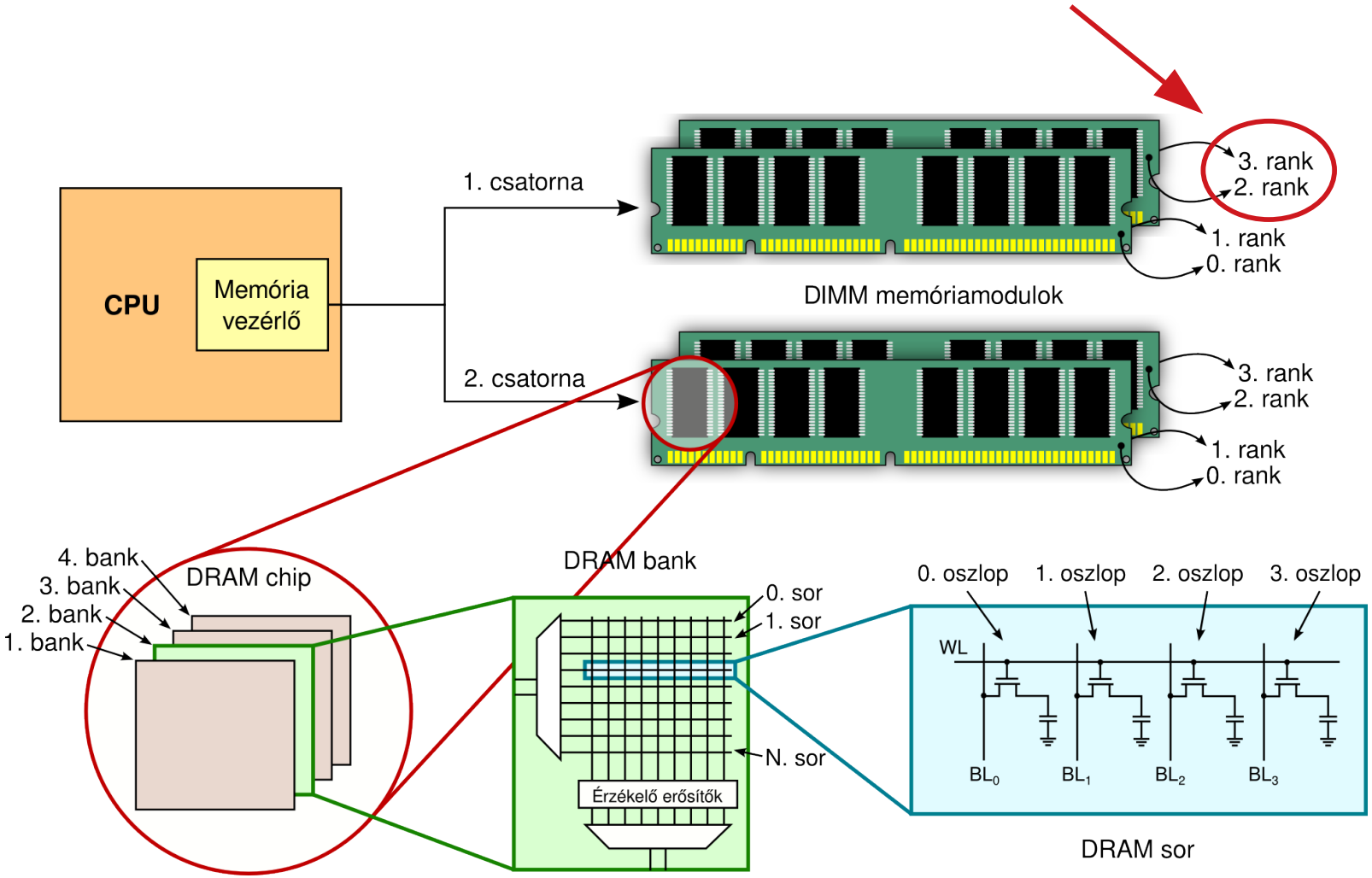
- Egy memóriamodul több DRAM chip-ből áll
- Parancs, bank kiválasztás, cím: osztott
- Adatvezetékek: összefogva



- Minden parancsot minden chip megkap
  → A megfelelő bankokban ugyanazok a sorok vannak nyitva
- Hatása:
  - Adatátviteli sebesség: 8x
  - Késleltetés: ugyanaz

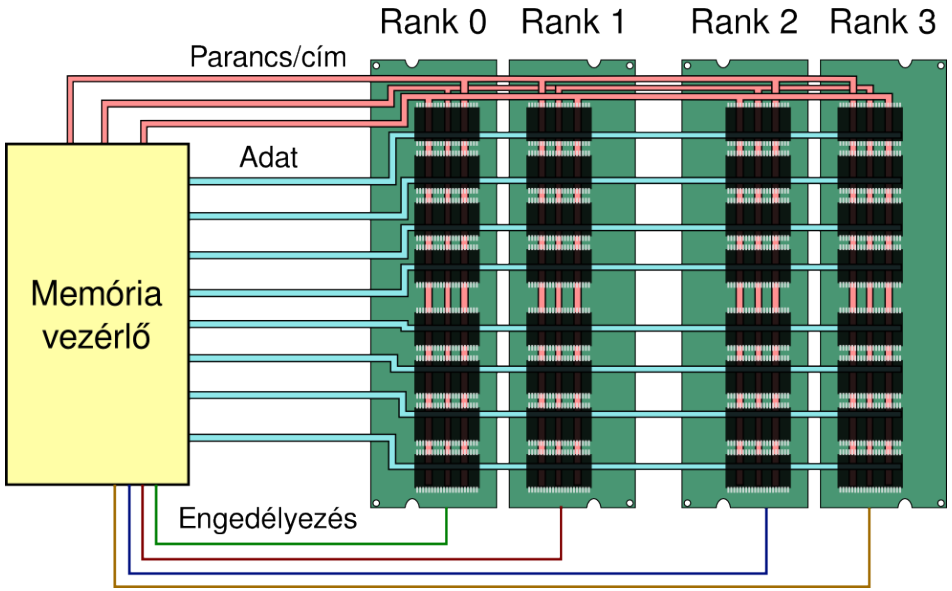
DRAM RANK-EK

DRAM rank-ek



DRAM RANK-EK

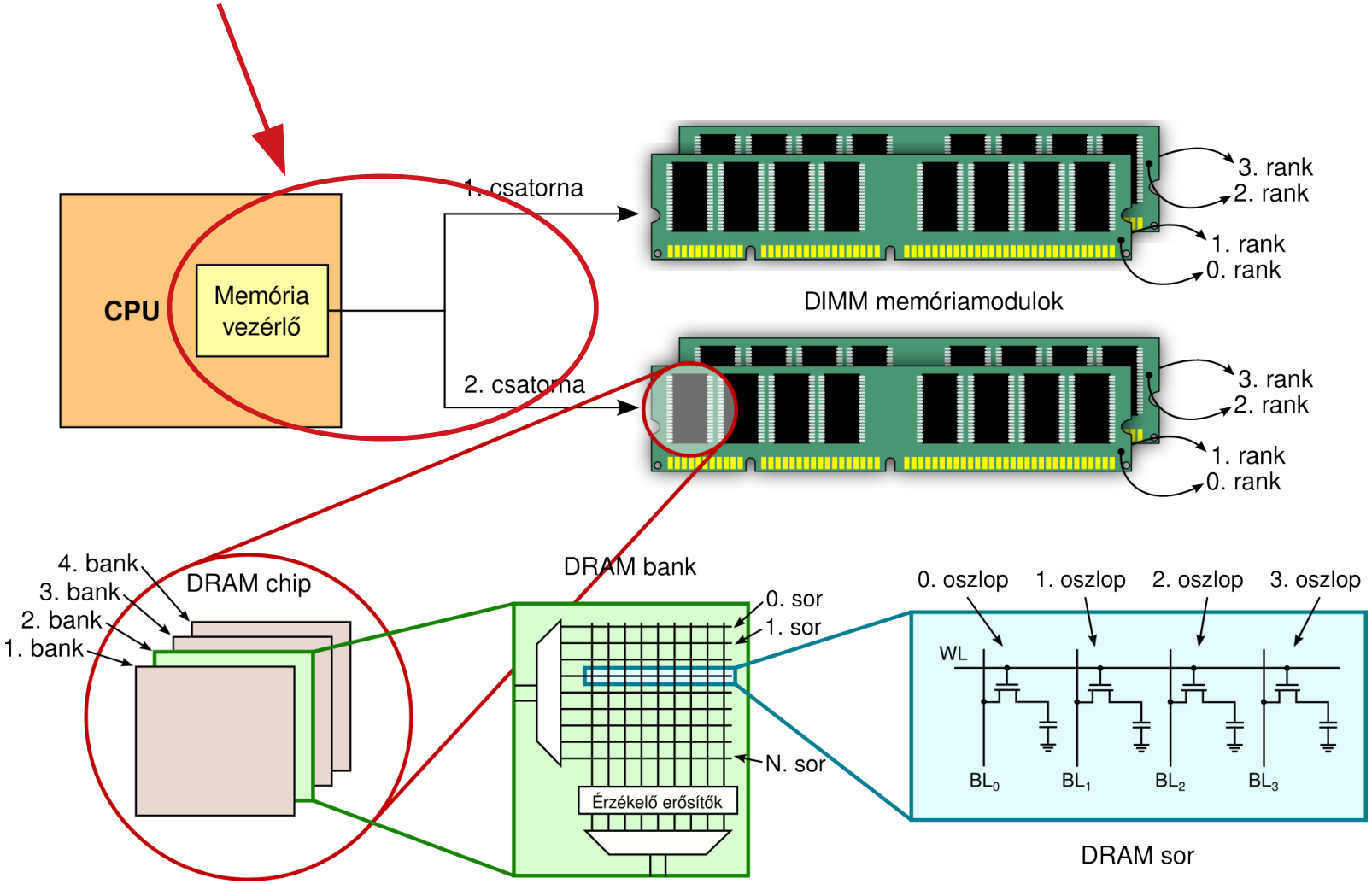
- A tárolási kapacitás növeléséhez
- Független egységek közös buszon
- Minden vezetékük osztott
  - Egyszerre csak 1 lehet bekapcsolva → chip select vezetékek



- Hatása:
  - Adatátviteli sebesség: ugyanaz
  - Késleltetés: jobb (több bank, több nyitott sor)

A MEMÓRIAVEZÉRLŐ

Memóriavezérlő



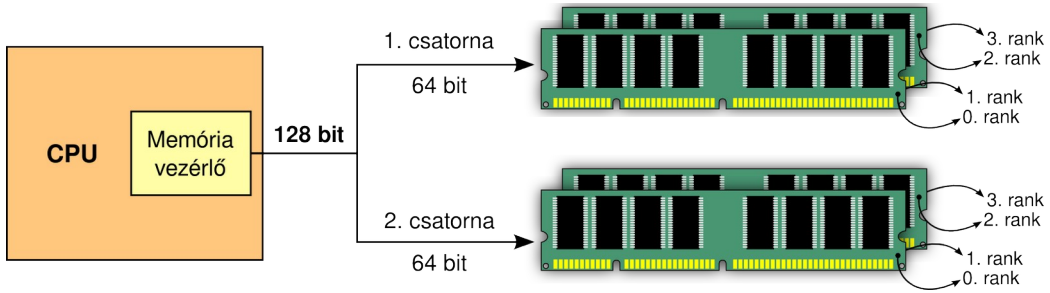
A MEMÓRIAVEZÉRLŐ

- Feladatai:
  - Memóriakérések kiszolgálása
    (CPU és perifériák felől)
- Teendők:
  - Címek leképzése csatornára/rank-re/bankra/sorra/oszlopra
  - Kérések sorrendjének optimalizálása
  - Gazdálkodás a nyitott sorokkal
  - Periodikus frissítések (REFRESH paranccsal)

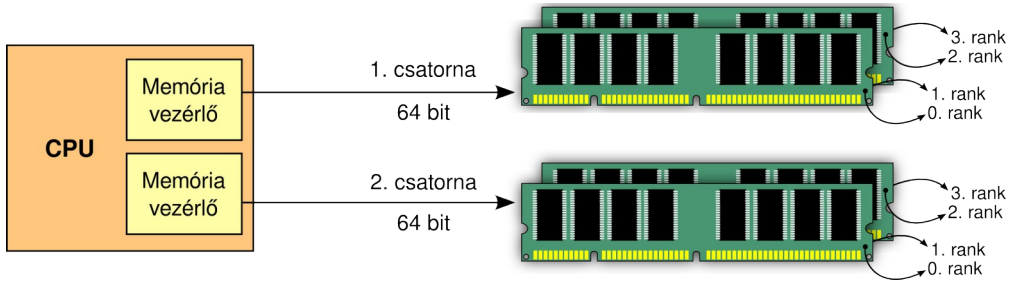
TÖBBCSATORNÁS ELÉRÉS

- Memóriavezérlő több csatornát is használhat
- *Szinkronizált* eset
  - Azonos modulok kellenek (darabra, méretre, időzítésekre)
  - Tökéletes szinkron működés

    2 db 64 bites csatorna → 1 db 128 bites csatorna



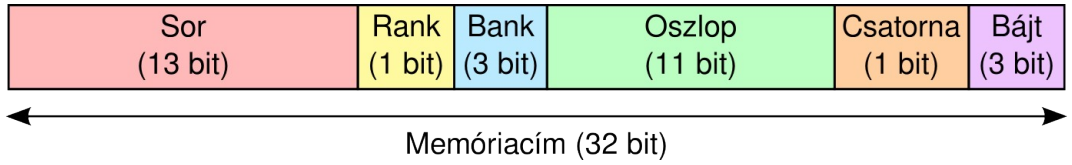
- *Független* csatornák esetén
  - Nem szükségesek azonos modulok
  - Annyi független memóriavezérlő, ahány csatorna



CÍMLEKÉPZÉS

- Címleképzés:
  - Cél: lehető legkevesebb sorváltás
  - Memóriakérések lokalitása kihasználható
  - Az egybefüggő, sorváltás nélkül elérhető címtartomány legyen nagy
  - Példa:

    (4 GB, 8 bank/chip, 2^{13} sor/bank, 2 rank, 2 csatorna, 64 bit adatszélesség)



  - 512 kB elérhető sorváltás nélkül

A MEMÓRIAVEZÉRLŐ

- Kérések kiszolgálási sorrendjének optimalizálása:
  - Spóroljunk a drága sor nyitás/zárással
    - Sorrendi kiszolgálás (FCFS)
    - Gyorsakat előre (FR-FCFS)

	FCFS	FR-FCFS
Kérések:	(3.sor, 8. oszlop) (1.sor, 3. oszlop) (3.sor, 14. oszlop)	(3.sor, 8. oszlop) (3.sor, 14. oszlop) (1.sor, 3. oszlop)
Parancsok:	ACTIVATE 3 READ 8 PRECHARGE ACTIVATE 1 READ 3 PRECHARGE ACTIVATE 3 READ 14	ACTIVATE 3 READ 8 READ 14 PRECHARGE ACTIVATE 1 READ 3

A MEMÓRIAVEZÉRLŐ

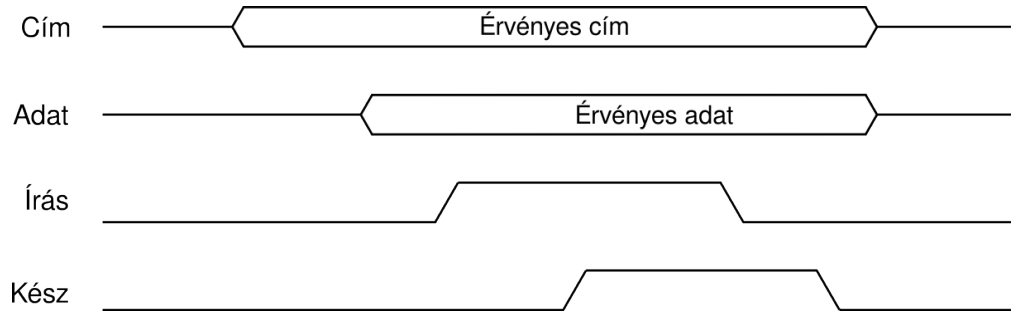
- Gazdálkodás a nyitott sorokkal:
  - Elfogynak a kérések. Bezárjuk az aktuális sort?
    - **Ne**:
      - Ha ugyanarra a sorra jön a köv. kérés, nem kell megnyitni
      - Ha más sorra jön a köv. kérés, be kell zárni (késleltetés)
    - **Igen**:
      - Ha ugyanarra a sorra jön a köv. kérés, újra meg kell megnyitni (késleltetés)
      - Ha más sorra jön a köv. kérés, nem kell bezárni
    - **Adaptív**:
      - Megtippeli, hogy ugyanerre a sorra jön-e a köv. kérés
      - APM (Active Page Management)

        Core i7: „Adaptive Page Closing” opció a BIOS-ban

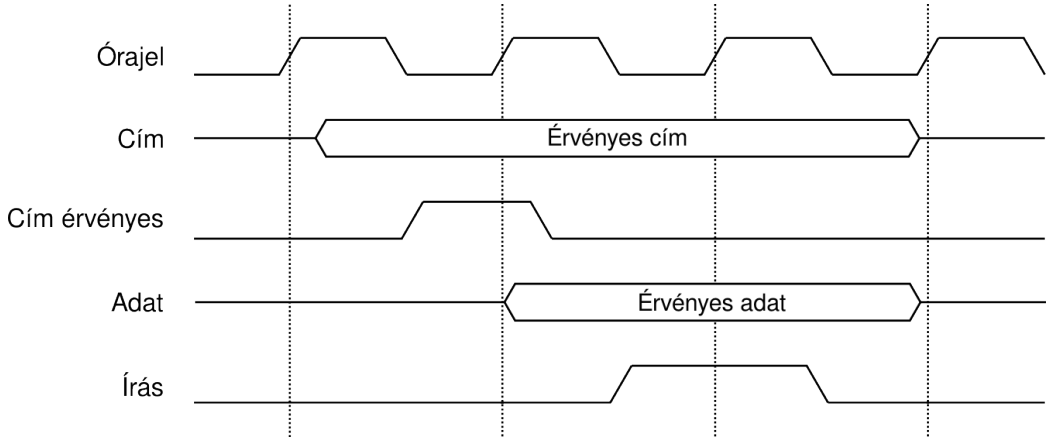
A DRAM technológiák evolúciója

DRAM EVOLÚCIÓ

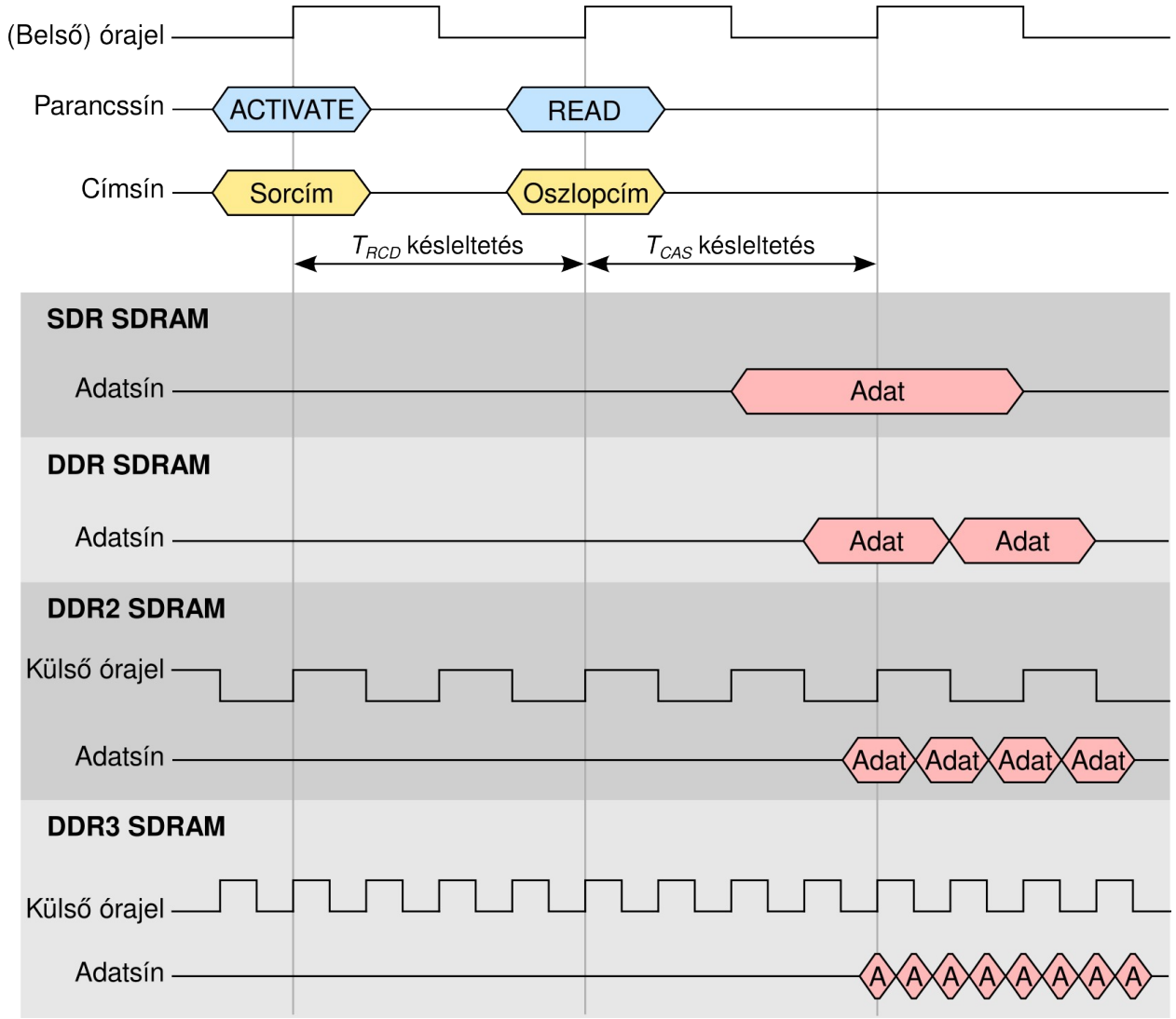
- 15 évvel ezelőttig: aszinkron interfész



- Ma: szinkron – SDRAM
  - Órajelet használ



SDR-DDR-DDR2-DDR3



SDR-DDR-DDR2-DDR3

- Szabványos jelölés:
  - Az ekvivalens SDR órajel megadásával:
    - DDR-400, DDR2-800, DDR3-1600 órajele: 200 MHz
    - **A parancsok késleltetése egyenlő!!!**
    - … csak egyre gyorsabban küldik át az adatot
  - Az adatátviteli sebesség megadásával:
    - pl. DDR2-800: adatátvitel 800 MHz-en, 8 byte/adategység
      → **PC2-6400** (800x8 = 6400)
    - pl. DDR3-1600: adatátvitel 1600 MHz-en, 8 byte/adategység
      → **PC3-12800** (1600x8 = 12800)

- Figyelem! Ez egy példa. Az órajel nem midig 200 MHz!

DDR4 SDRAM

- **A külső ↔ belső órajel aránya (burst hossz) nem nő tovább**
- Teljesítmény fokozása:
  - **Belső órajel növelése**, ehhez:
    - kisebb tápfeszültség (1.2V)
      → kisebb fogyasztás, kevésbé melegszik
    - busz órajele nő, áthallás, csúszás kezelése nehezedik
      → CRC (adatbusz) / paritásbit (cím és parancs) kell,
      kalibrálás szükséges
    - 8 helyett 16 bank (nagyobb párhuzamosság)
    - bank csoportok bevezetése, bonyolultabb időzítés
  - Nagyobb méretű (kapacitású) modulok, ehhez:
    - maximum 4 rank / modul
    - a bank-ek nem négyzet alakúak
      → több sorcím bit, mint oszlopcím bit → command jelek
      a felső címbitekkel multiplexálva (kevesebb kivezetés)
  - Max. 8 csatornás memória vezérlő

DDR5 SDRAM

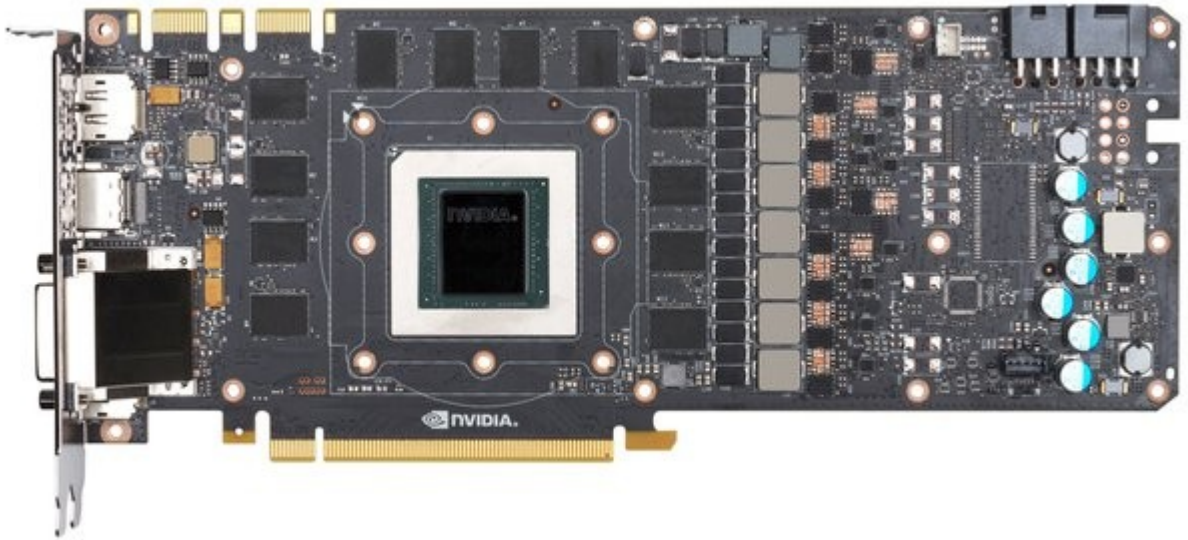
- Megjelenés: 2020
- **Burst hossz újra nő → =16**
- Teljesítmény fokozása:
  - Még kisebb tápfeszültség (1.1V)
  - 2x annyi bank
  - **Modulonként** 2 csatorna (64 bit helyett 32+32 bites adategységek)
  - Kétszeres burst méret fele szélességgel → 64 byte/burst
    - Kétszer annyi ideig tart
    - De! Két független csatornán egy időben
      → Ugyanaz a throughput
      → Jobb késleltetés

ÖSSZEHASONLÍTÁS

	SDR	DDR	DDR2	DDR3	DDR4	DDR5
Belső órajel	66-133 MHz	133-200 MHz	100-200 MHz	100-200 MHz	200-533 MHz	200-533 MHz
Adat/b. órajel	1	2	4	8	8	8
Adatátv. MB/s	528-1064	2128-3200	3200-6400	6400-12800	12800-34112	≈ DDR4
Burst hossz	1-8	2-8	4-8	8	8	16
Bank szám	2-4	2-4	4-8	8	16	32
Feszültség	3.3V	2.5V	1.8V	1.5V	1.2V	1.1V

GDDR5 SDRAM

- Cél: extrém nagy sávszélesség → grafikus kártyákhoz
- Egyszerűbb:
  - Nincs rank, nincs modul
  - Minden chip közvetlenül a vezérlőhöz kötve
- x32 adatszélesség chip-enként
  8 chip → 256 bites adatszélesség
- 16 bank, de bankonként 2 sor lehet nyitva
- Egyszerre tud írni és olvasni
- 3 -féle órajel:
  - DRAM cella órajel, pl. 375 MHz
  - Parancsórajel: 4-szeres: 1500 MHz
  - Adatátviteli órajel: 4-szeres: 6 GHz !!!
  → 256 bites adatszélességgel **196 GB/s** !
- Problémák:
  - Ár (4-5-szörös)
  - Rövid chip-vezérlő távolság
  - Nem moduláris / nem bővíthető
  - Melegszik, sokat fogyaszt
- GDDR6:
  - x32 helyett x16, de 2 csatorna + jobb gyártástechnológia → 768 GB/s



ÖSSZEGZÉS

- DRAM alapú memóriák:
  - Késleltetéssel problémák vannak (évtizedek óta változatlan)
    - Grafikus kártyának nem fontos
    - CPU-nak nagyon fontos (lenne)
  - Sávszélesség még növelhető (egy darabig)

HÁLÓZATI RENDSZEREK
ÉS SZOLGÁLTATÁSOK
TANSZÉK

HÁLÓZATI RENDSZEREK ÉS SZOLGÁLTATÁSOK TANSZÉK

MŰEGYETEM 1782

Budapest,
2022.03.22.

SZÁMÍTÓGÉP ARCHITEKTÚRÁK

A virtuális memória

Horváth Gábor, Belső Zoltán

BME Hálózati Rendszerek és Szolgáltatások Tanszék
ghorvath@hit.bme.hu, belso@hit.bme.hu

VIRTUÁLIS TÁRKEZELÉS

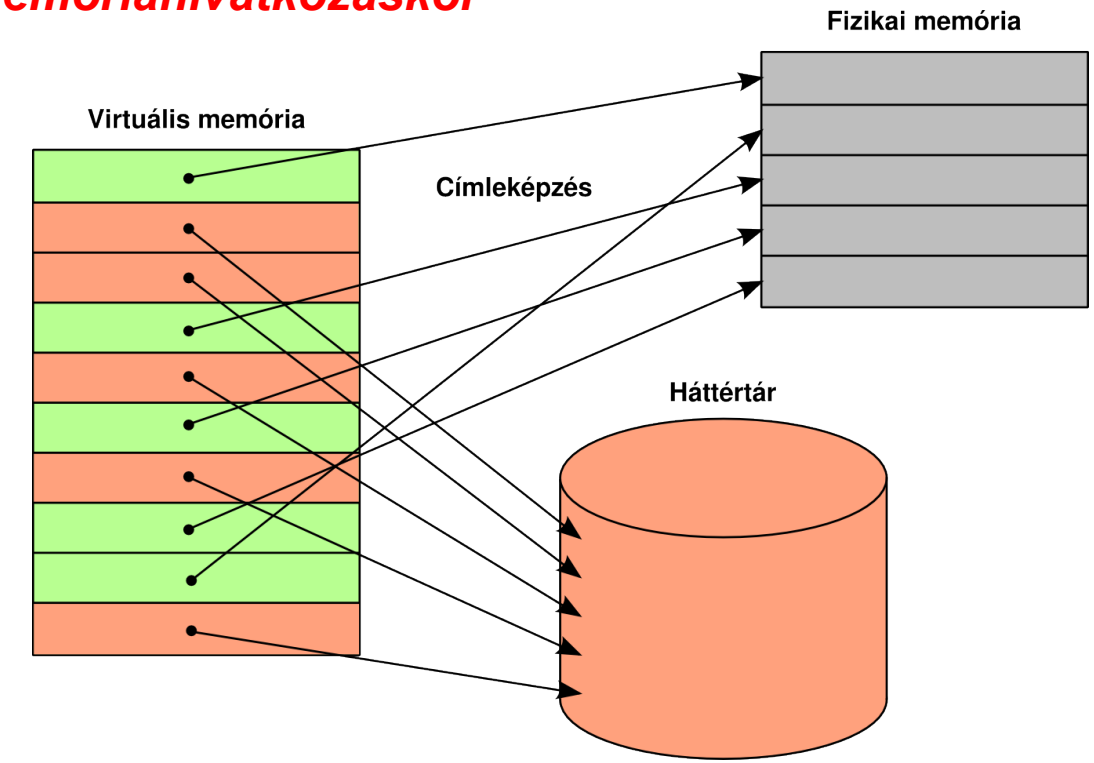
- Motiváció:
  - Multitaszking környezet
    - Taszkok jönnek-mennek
    - A programunknak jutó memóriaméret változó
  - Olcsó 64 bites processzorok
    - Ebből pl. az AMD64 48 bitet használ: 256 TB
    - Ennyi memóriát nem tudunk hozzáilleszteni

- A CPU minden programnak a teljes címtartományt felajánlja
  - 0-tól → a címtartomány tetejéig
  - Ne legyen gond a memóriaméret
- Ezt a „virtuális” memóriát kell leképezni a fizikaiba

  → ***Virtuális tárkezelés***

VIRTUÁLIS TÁRKEZELÉS

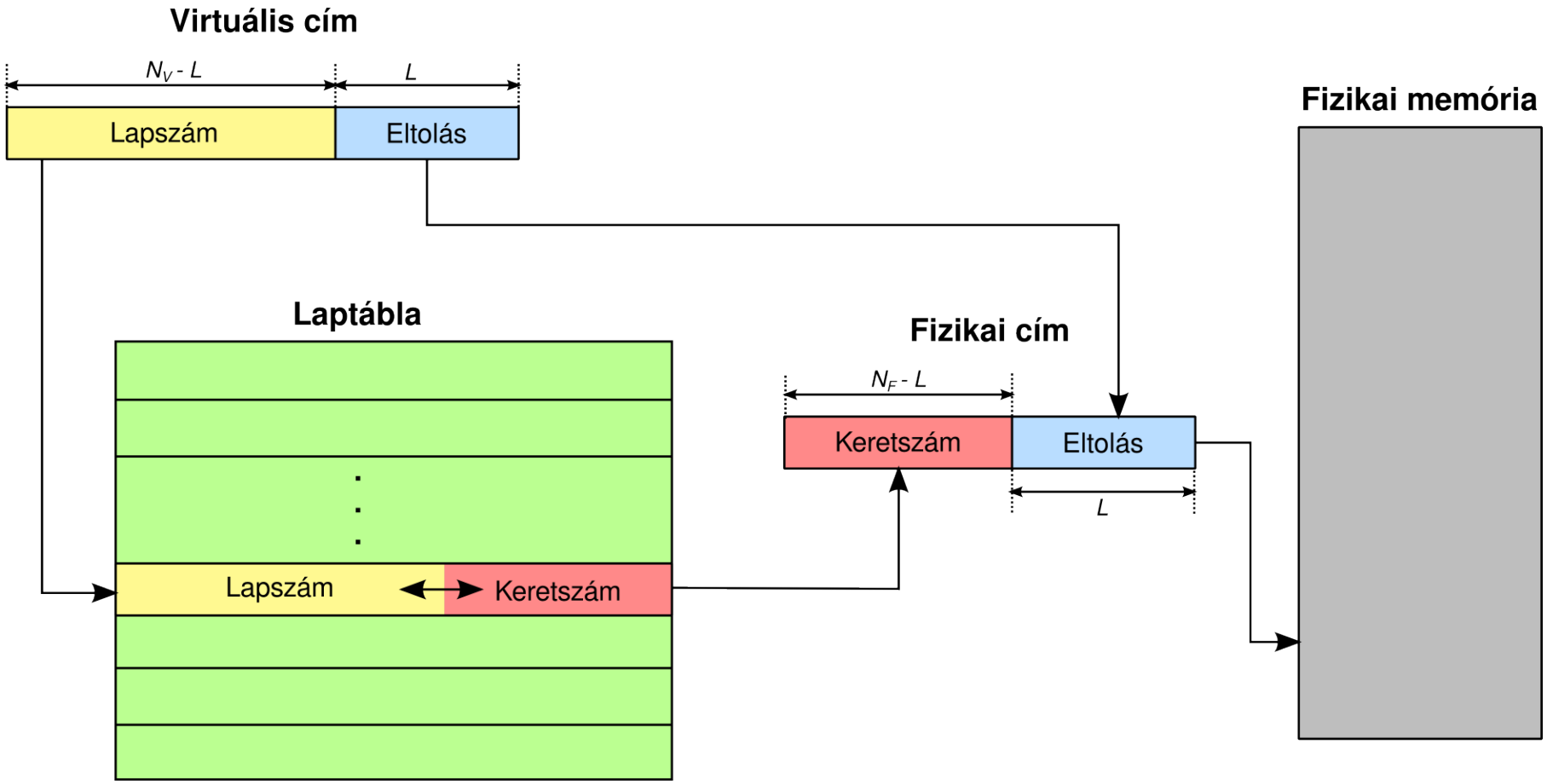
- Programok: ***Virtuális címeket*** használnak
- CPU lábain/buszon: ***Fizikai címek*** jelennek meg
- Virtuális → fizikai cím leképzés: ***Címfordítás***
  - Aki csinálja: ***MMU***
  - Amikor csinálja: ***minden memóriahivatkozáskor***
  - Egysége:
    - ***Lap*** (fix)
    - vagy ***szegmens*** (változó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



CÍMFORDÍTÁS

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

CÍMFORDÍTÁS



Laptábla implementációk

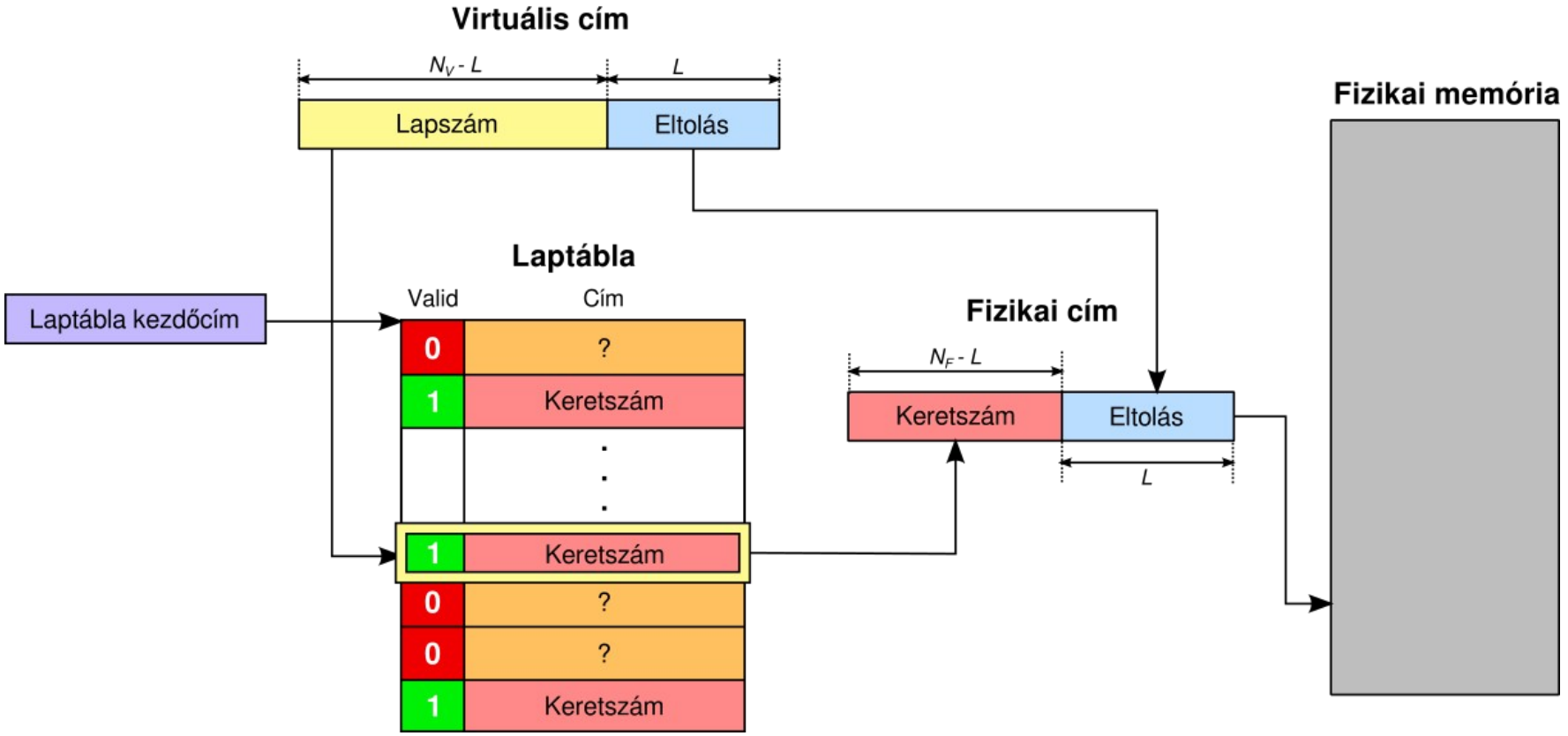
LAPTÁBLA IMPLEMENTÁCIÓK

- Cél:
  - Címfordítás legyen minél gyorsabb
    → virtuális cím szerinti keresés legyen gyors
    → minél kevesebbszer kelljen a memóriához nyúlni
  - Laptábla legyen a lehető legkisebb

- Eszköz:
  - Speciális adatszerkezetek
    - Egyszintű laptábla
    - Többszintű (hierarchikus) laptábla
    - Inverz laptábla
    - (Virtualizált laptábla)

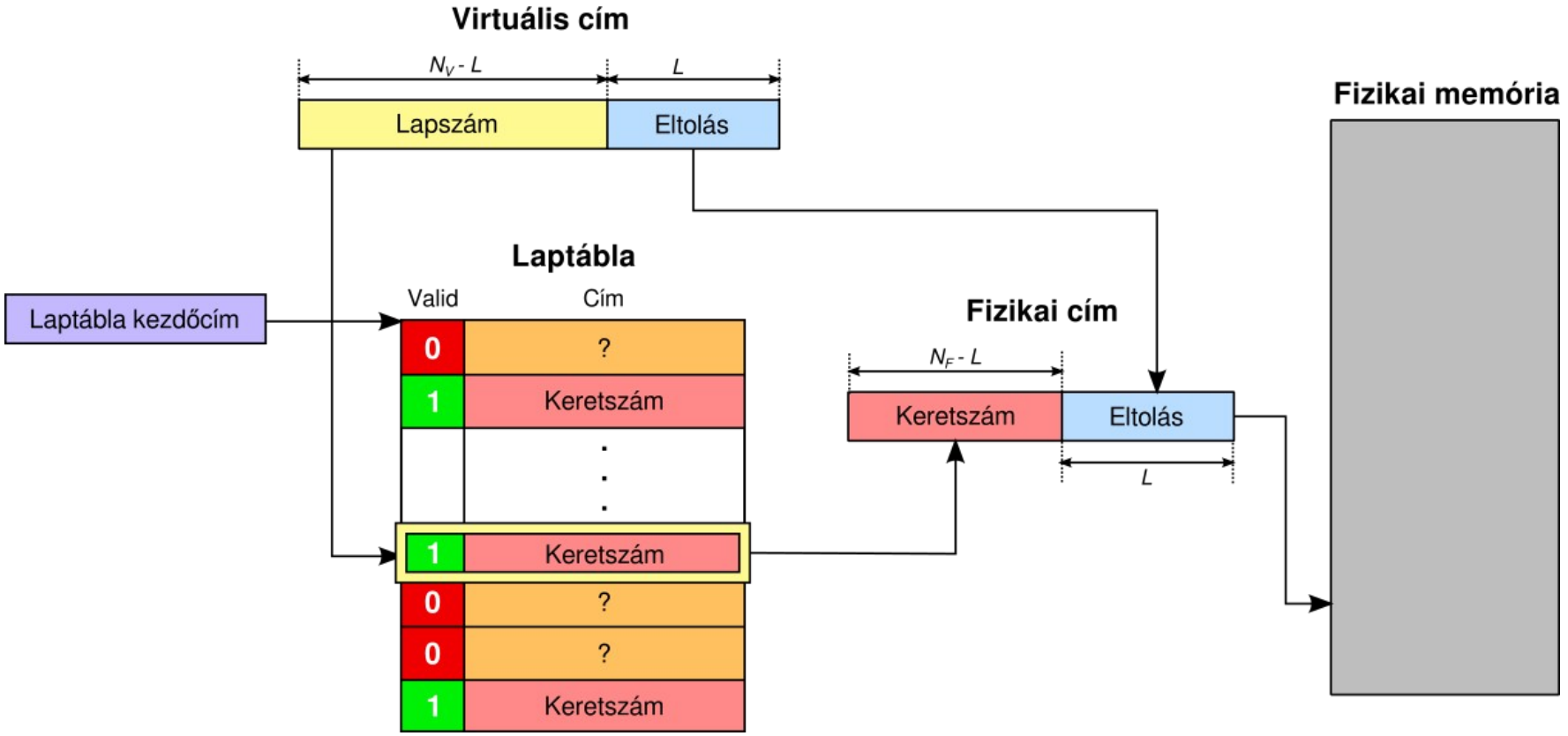
EGYSZINTŰ LAPTÁBLA

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



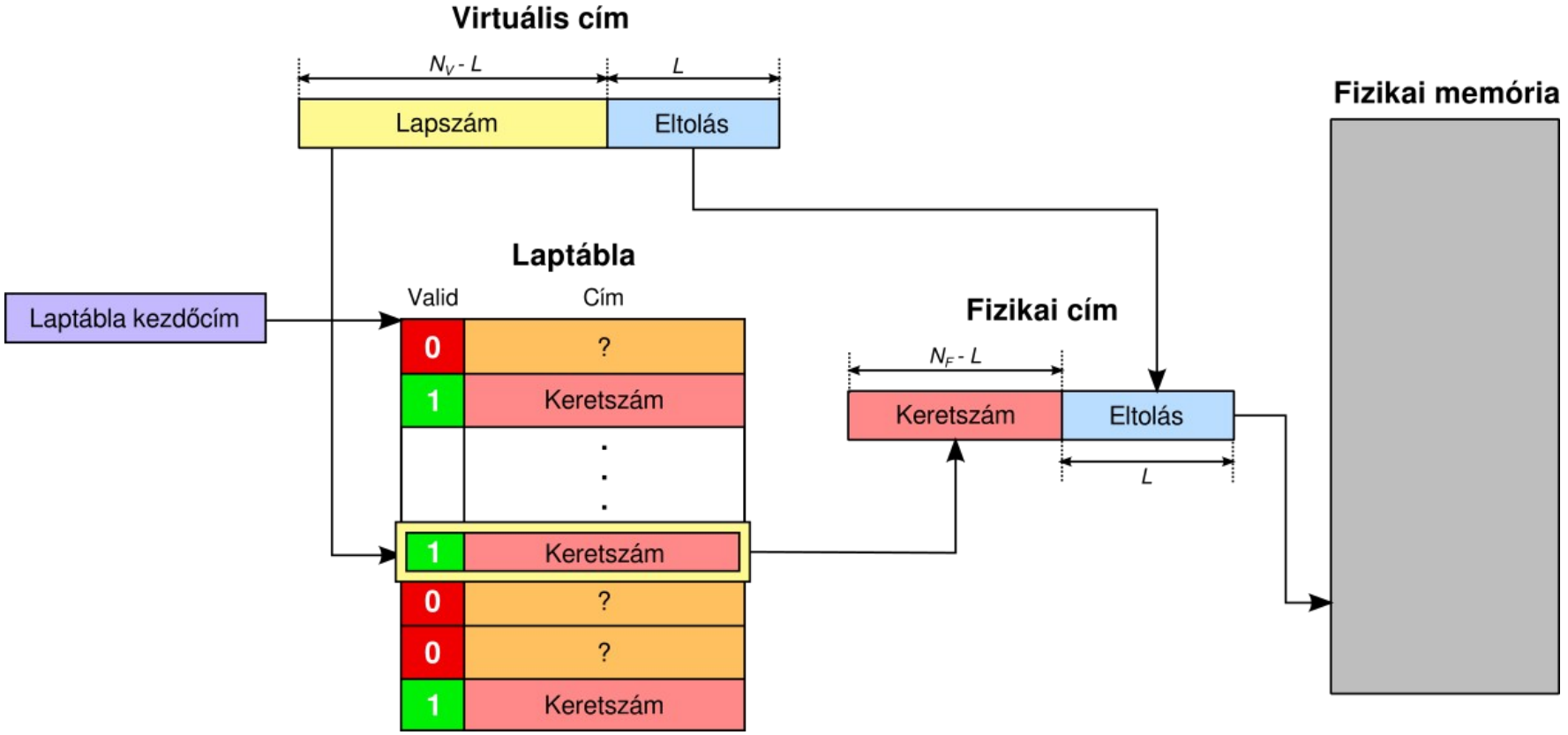
EGYSZINTŰ LAPTÁBLA

- 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

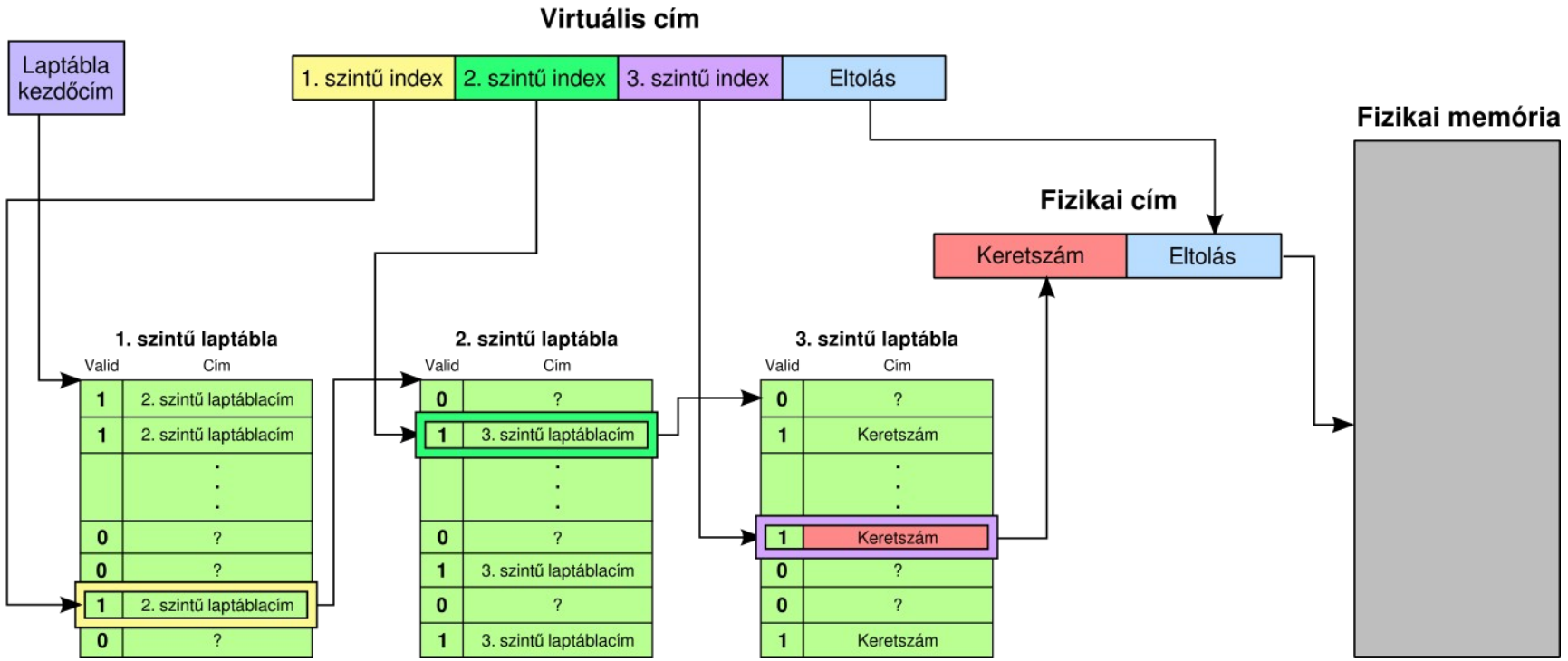
- Gyors, kevés memória-hozzáférés kell, kicsi a bejegyzés...
  ...miért nem használ mindenki ilyet?

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

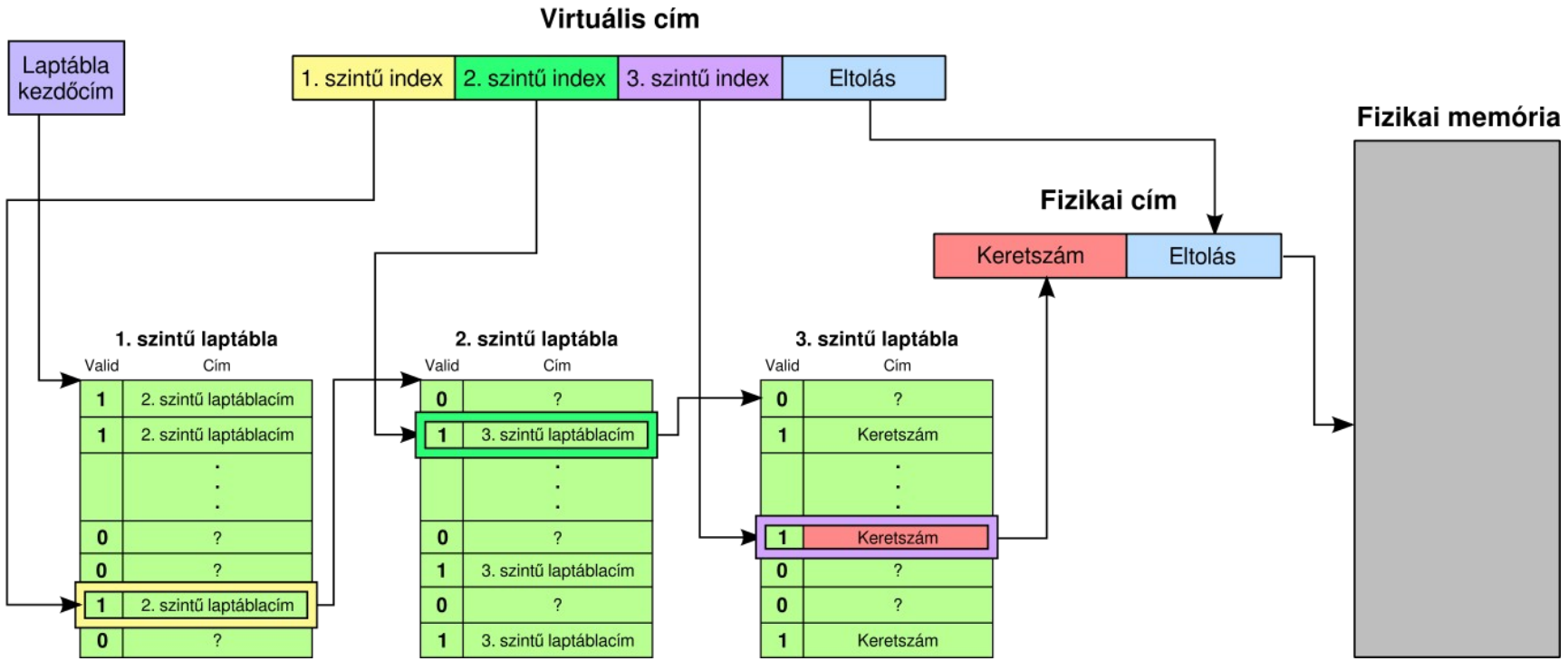
HIERARCHIKUS LAPTÁBLA

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

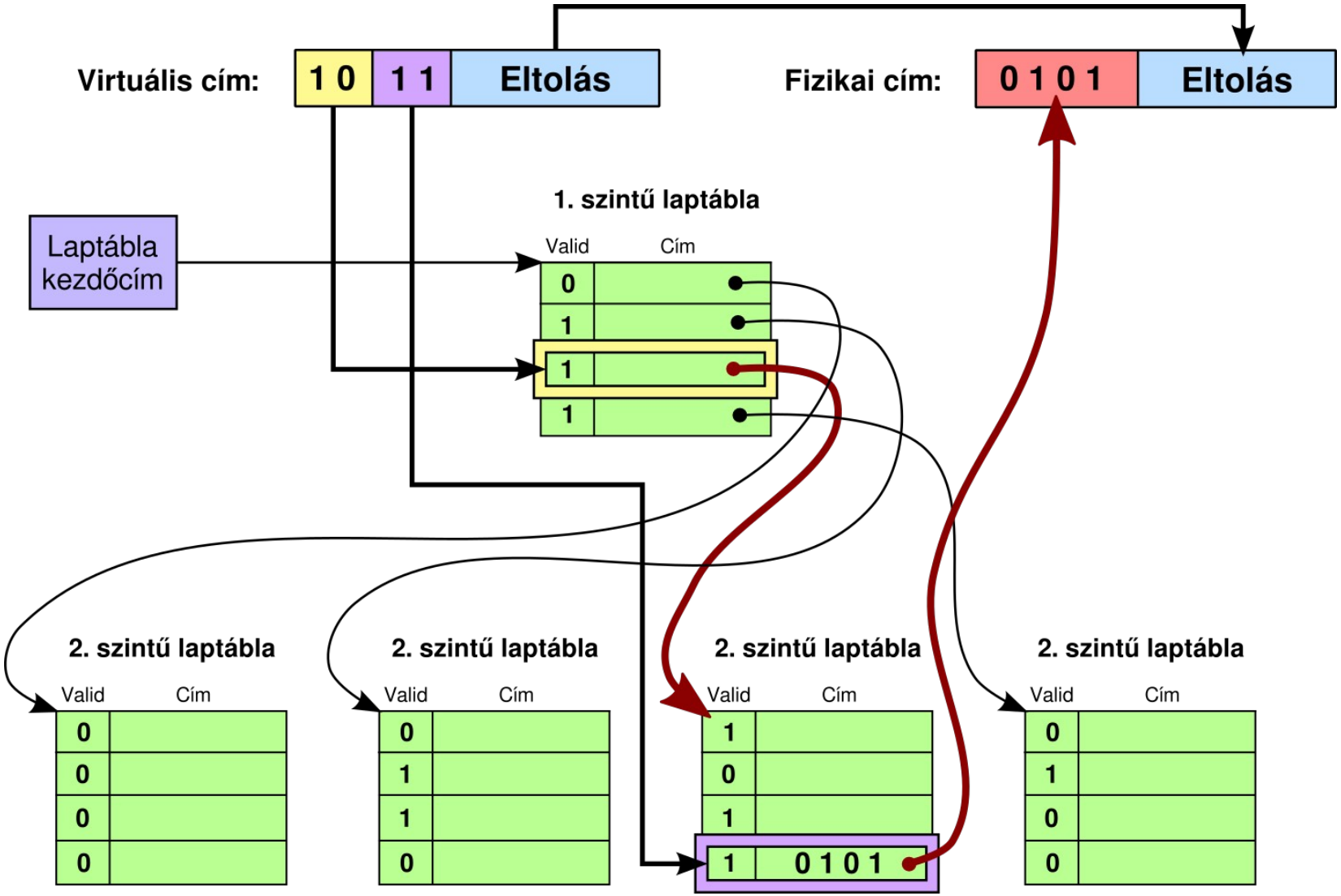


HIERARCHIKUS LAPTÁBLA

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



KÉTSZINTŰ LAPTÁBLA, PÉLDA



HIERARCHIKUS LAPTÁBLA

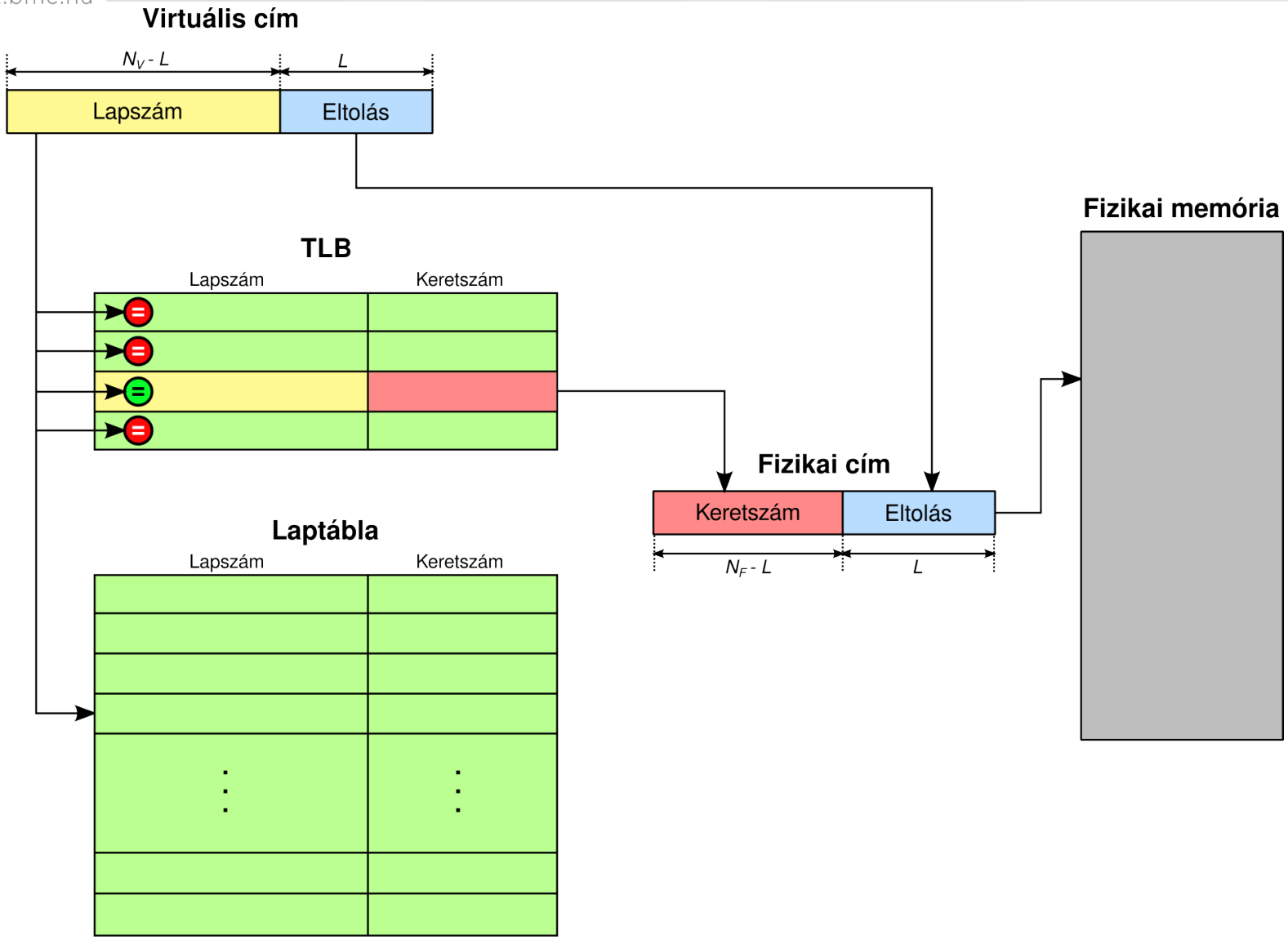
- Gyors kalkuláció:
  - 32 bites esetben, 4 kB lapokkal:
    - Lapméret: 12 bit, 1 bejegyzés: 4 bájt elég
    - Egy lapra 1024 bejegyzés fér → 10 bittel indexelhető
    - Kell 1024 laptábla lap, és még egy lap, ami ezekre mutat
    - 32 bites cím = 10 + 10 + 12 bites részek
      → 2 szintű laptábla 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

TLB

TLB

- Memória-hozzáférés 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 gyakran használt virtuális ↔ fizikai összerendelések cache-e
  - 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

  → Szűkös a tárolási kapacitása :(

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

HARDVER MENEDZSELT TLB

- Címfordítás: **CPU/MMU**...
  - először a TLB-ből próbálkozik,
  - TLB hiba esetén: ***bejárja a laptáblát***
    - Valid = 1 esetén: TLB update, kész.
    - Valid = 0 esetén: Laphiba! → operációs rendszert hív, majd újra próbálkozik
- Laphiba kezelése: **Az operációs rendszer**...
  - megkapja a keresett lap számát
  - saját nyilvántartása alapján betölti a diszkről
  - ...ha ott van!
    Ha még senki nem használta a lapot, létrehoz egy üreset
  - elhelyezi a fizikai memóriában (esetleg kidob onnan valakit)
  - ***frissíti a laptáblát***
- Példa: x86, x86-64, ARM, POWER

SZOFTVER MENEDZSELT TLB

- Címfordítás: **CPU/MMU**...
  - először a TLB-ből próbálkozik,
  - TLB hiba esetén: ***operációs rendszert hív***, majd újra próbálkozik
- TLB hiba kezelése: **Az operációs rendszer**...
  - megkapja a keresett lap számát
  - bejárja a saját laptábláját:
    - Valid = 1: beírja a CPU TLB-be a lap ↔ keret összerendelést
    - Valid = 0: mint eddig (swap-elés, laptábla frissítés)
  - ***Mindenképp frissíti a TLB-t***
- Előnye:
  - Nincs hardver megkötés, op. rendszer frissítéssel jöhet hatékonyabb laptábla
  - Bonyolultabb adatszerkezetek is használhatók
- Hátránya:
  - Sokkal lassabb címfordítás (TLB hiba esetén)
  - Példa: SPARC, Alpha, MIPS, félig a PA-RISC

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 memória

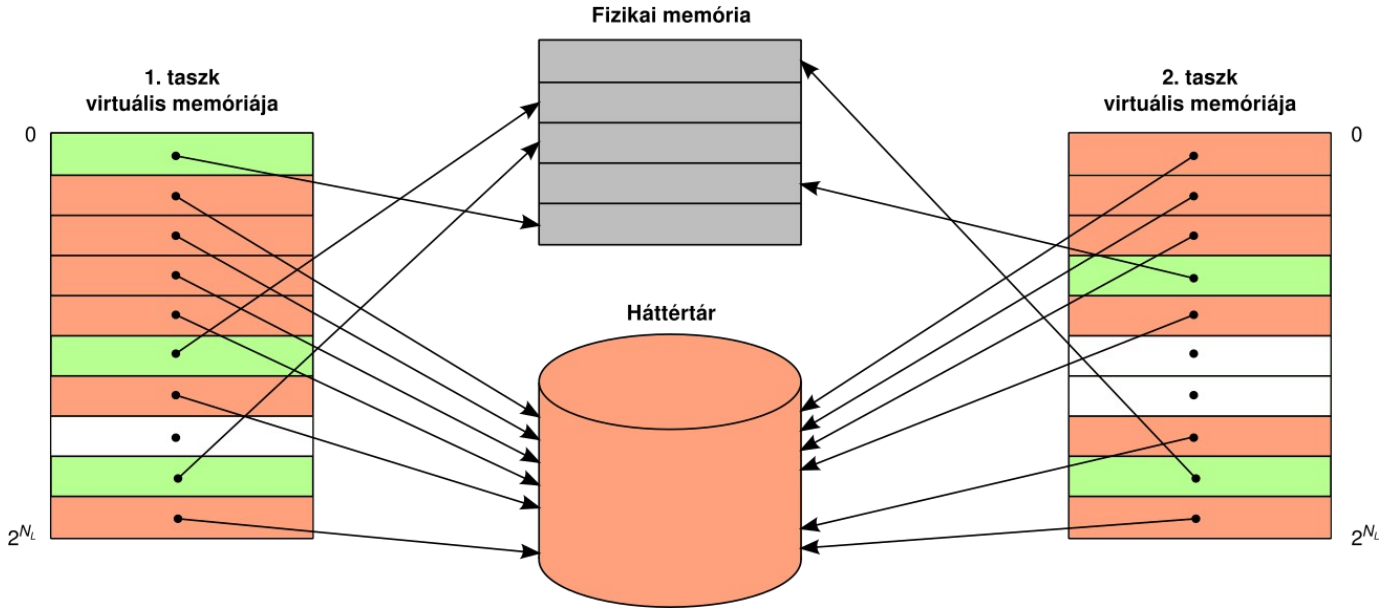
multitaszking környezetben

CÍMTÉR ELKÜLÖNÍTÉS

- Cél: minden taszk kapja meg a teljes címtartományt (pl. 0 → 4 GB)
- Előnyök:
  - **Állandó futási környezet**
    - Könnyebb szoftverfejlesztés
    - Mert fordításkor ismert
      - a belépési pont
      - a globális változók címe
      - a függvények címe
      - stb.
  - **Védelmet ad**
    - Nem tudja elérni más taszkok címterét!
- Megoldás:
  - Taszkonként külön laptábla

TASZKONKÉNTI LAPTÁBLA

- Minden taszk megkapja a teljes címtartományt
- Megoldás:
  - Taszkonként külön laptábla
  - Taszkváltáskor:
    - Laptábla kezdőpointer lecserélése
    - TLB invalidálás!



- Lehetnek osztott címtartományok, ugyanazokkal a keretekkel
  - Szükséges az op. rendszer meghívásához!

HÁLÓZATI RENDSZEREK
ÉS SZOLGÁLTATÁSOK
TANSZÉK

MŰEGYETEM 1782

HÁLÓZATI RENDSZEREK ÉS SZOLGÁLTATÁSOK TANSZÉK

MŰEGYETEM 1782

Budapest,
2022.03.23.

SZÁMÍTÓGÉP ARCHITEKTÚRÁK

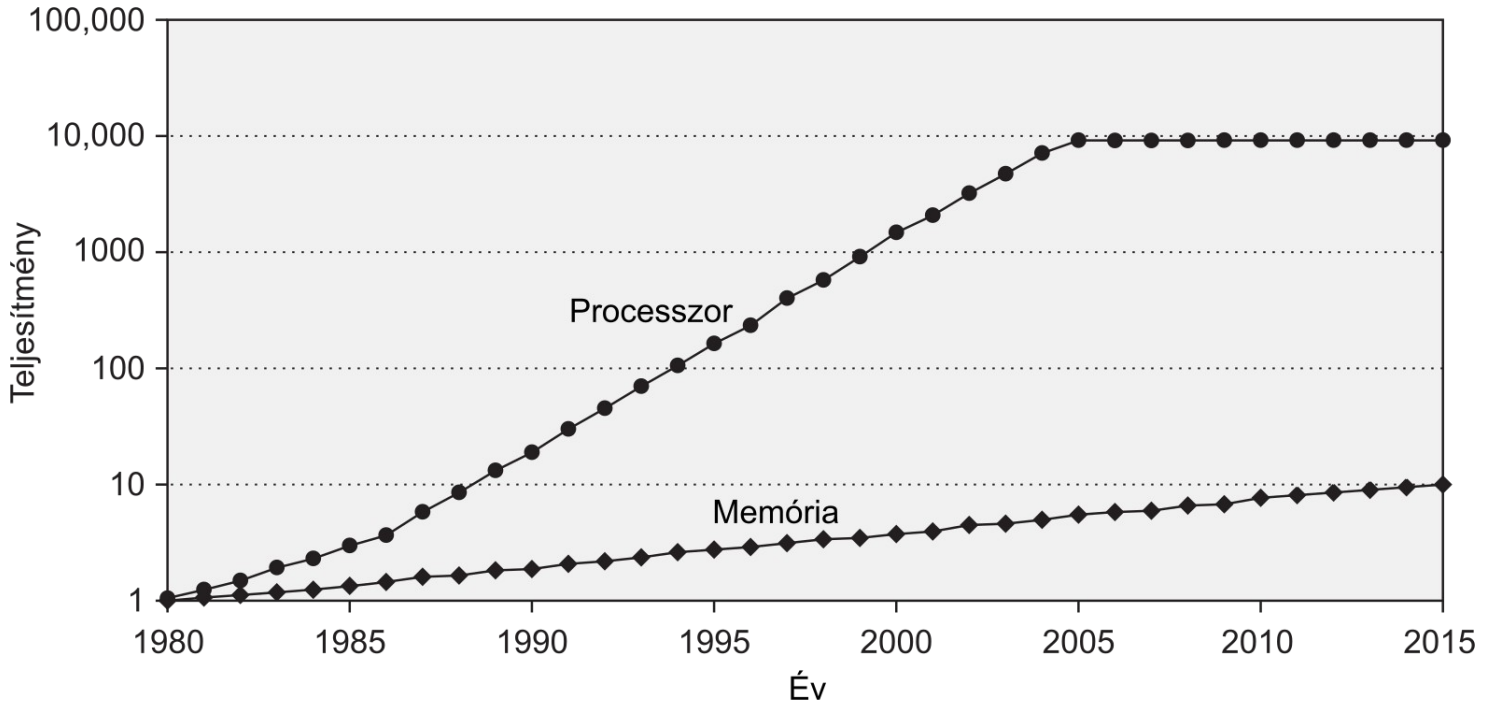
Cache memória

Horváth Gábor, Belső Zoltán

BME Hálózati Rendszerek és Szolgáltatások Tanszék
ghorvath@hit.bme.hu, belso@hit.bme.hu

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

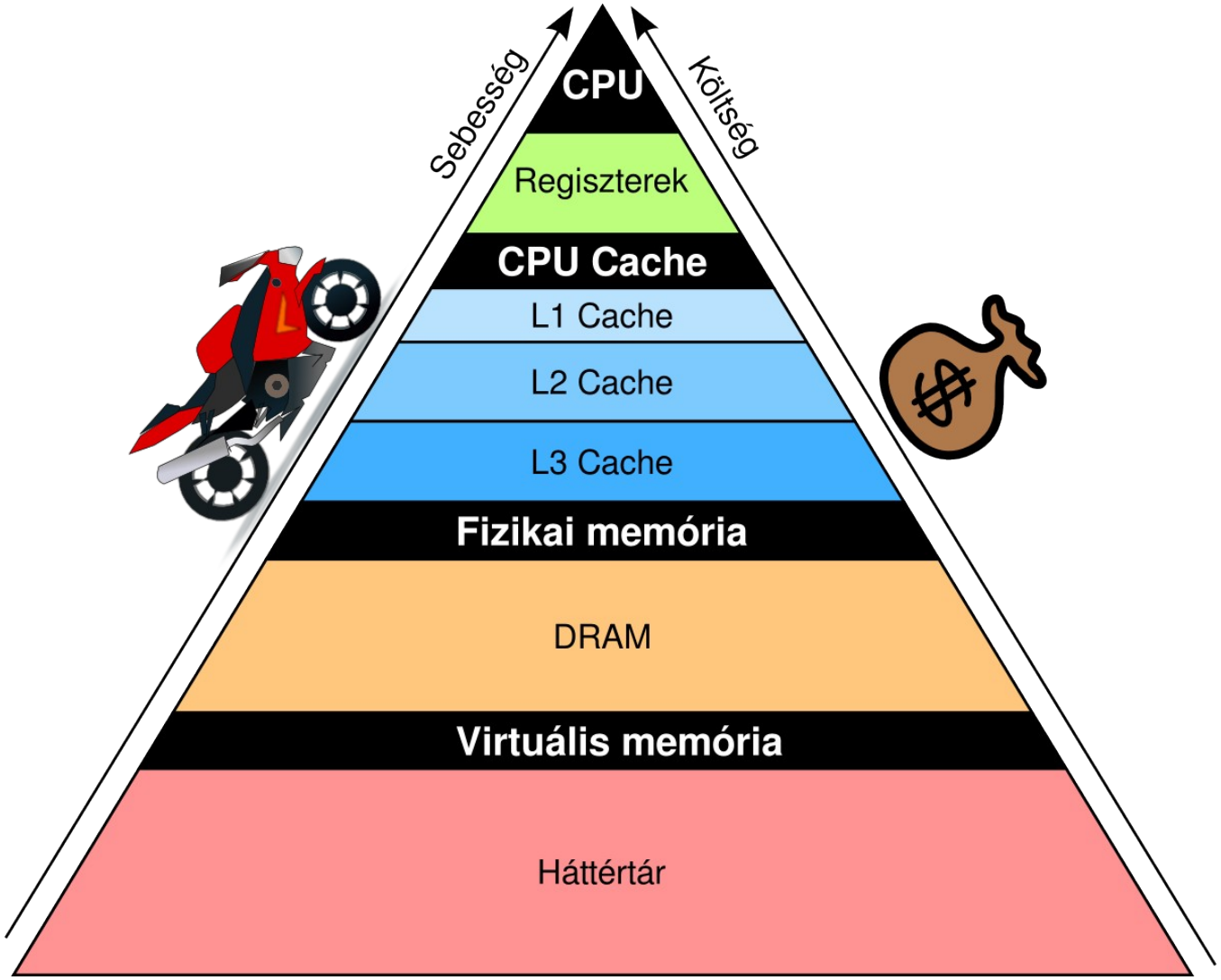
TÁRHIERARCHIA

- 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	$500 – $1000
DRAM	50 – 70 ns	$10 – $50
Flash	5 000 – 50 000 ns	$0.75 – $1.0
HDD	$5 – 20 \cdot 10^6$ ns	$0.05 – $0.1

(2012-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
  - Egzotikus 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

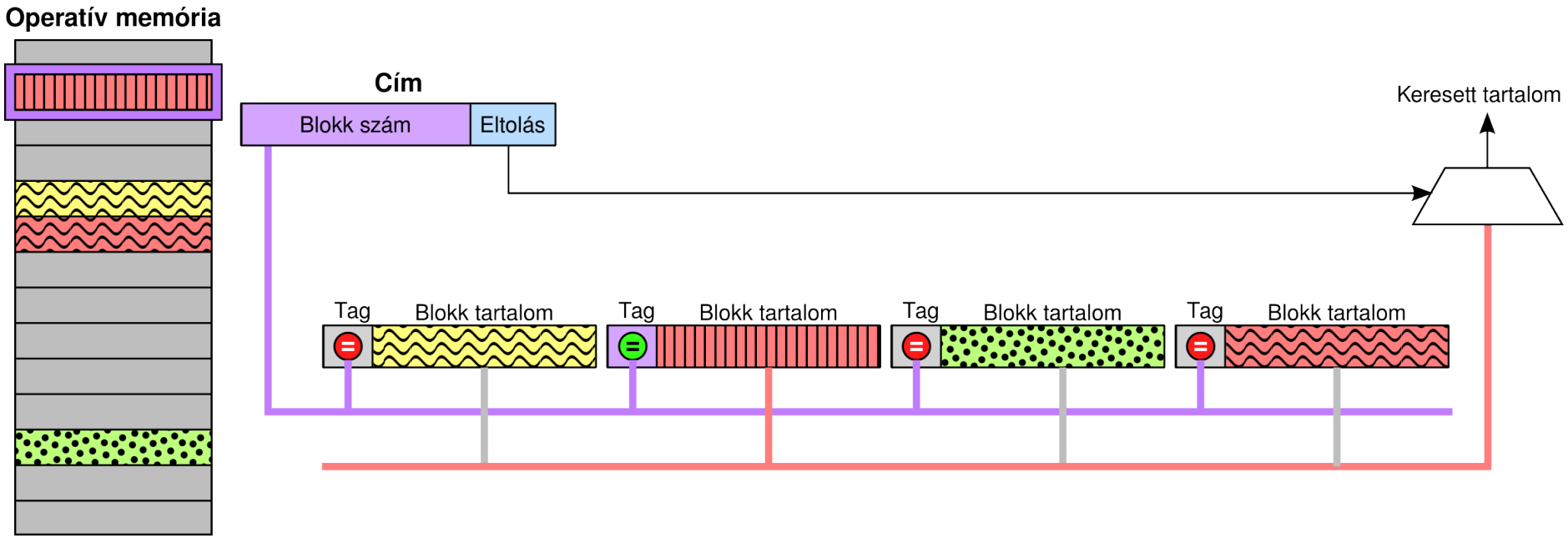
Cache szervezés

TRANSZPARENS CACHE SZERVEZÉS

- Tárolási egység: ***blokk*** (= cache line)
  - 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

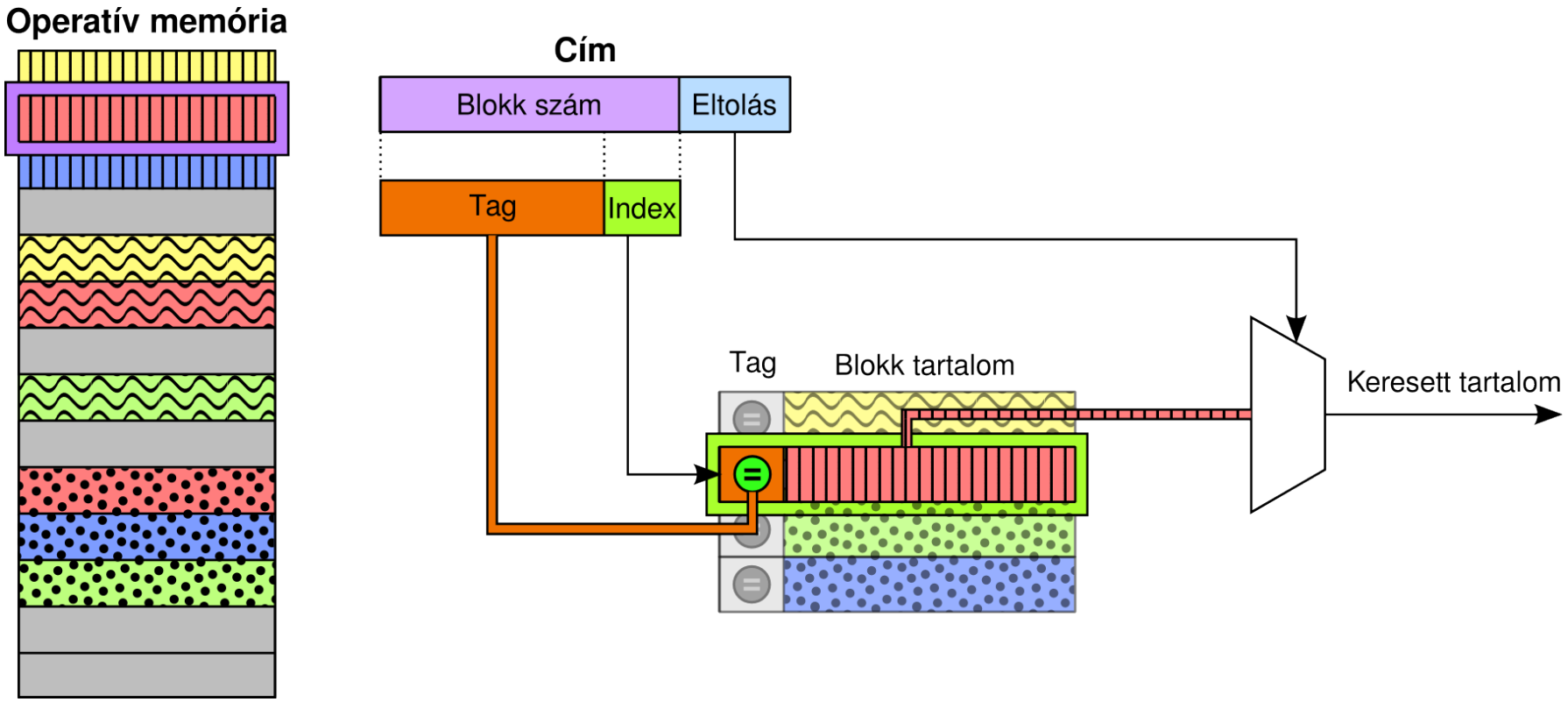
TELJESEN ASSZOCIATÍV SZERVEZÉS

- 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

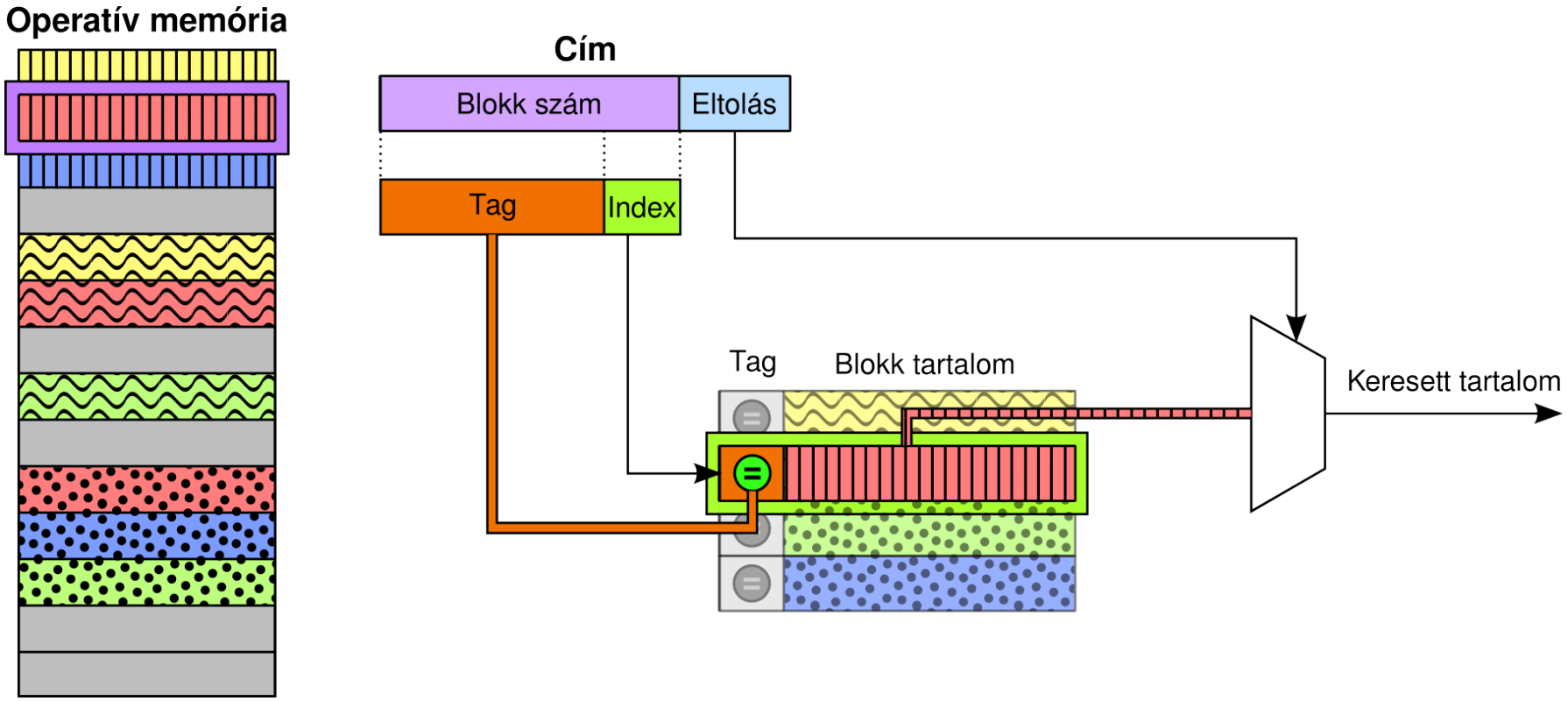
- Minden memóriabeli blokk csak egyetlen helyre kerülhet a cache-ben
- Hogy hova, azt a memóriabeli blokkszám egyértelműen eldönti
  - Pl. a blokkszám alsó bitjei alapján



DIREKT LEKÉPZÉS

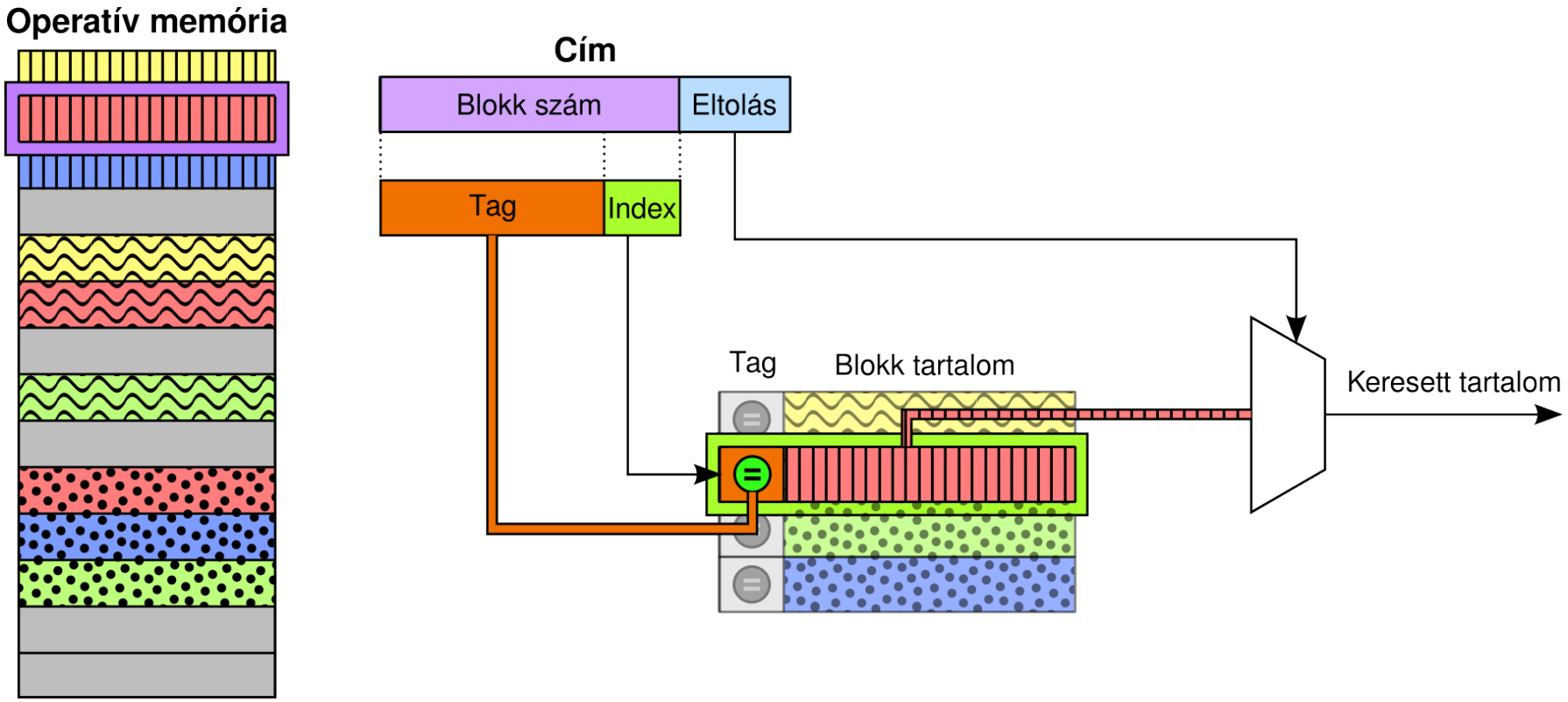
- Pl.: cache 4 blokkot tud tárolni: 1 sárgát, 1 pirosat, 1 kéket, 1 zöldet
- A memóriában a blokkok ebben a sorrendben követik egymást

  → 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)

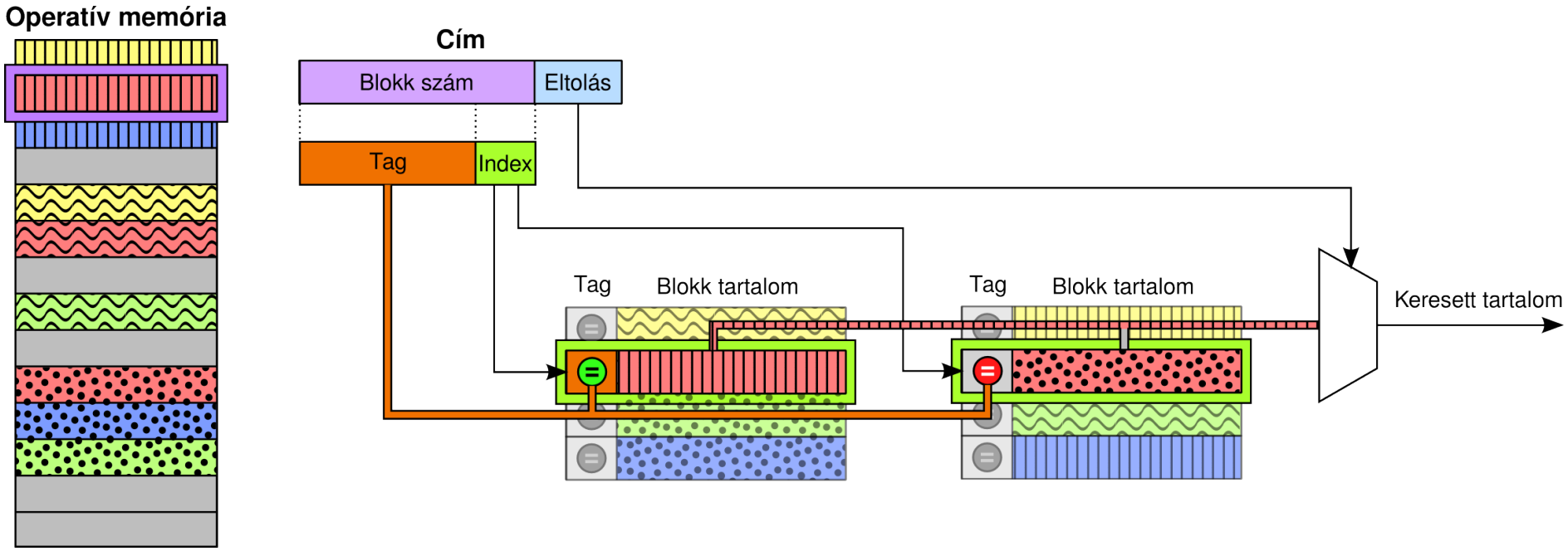


ÉRTÉKELÉS

- 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

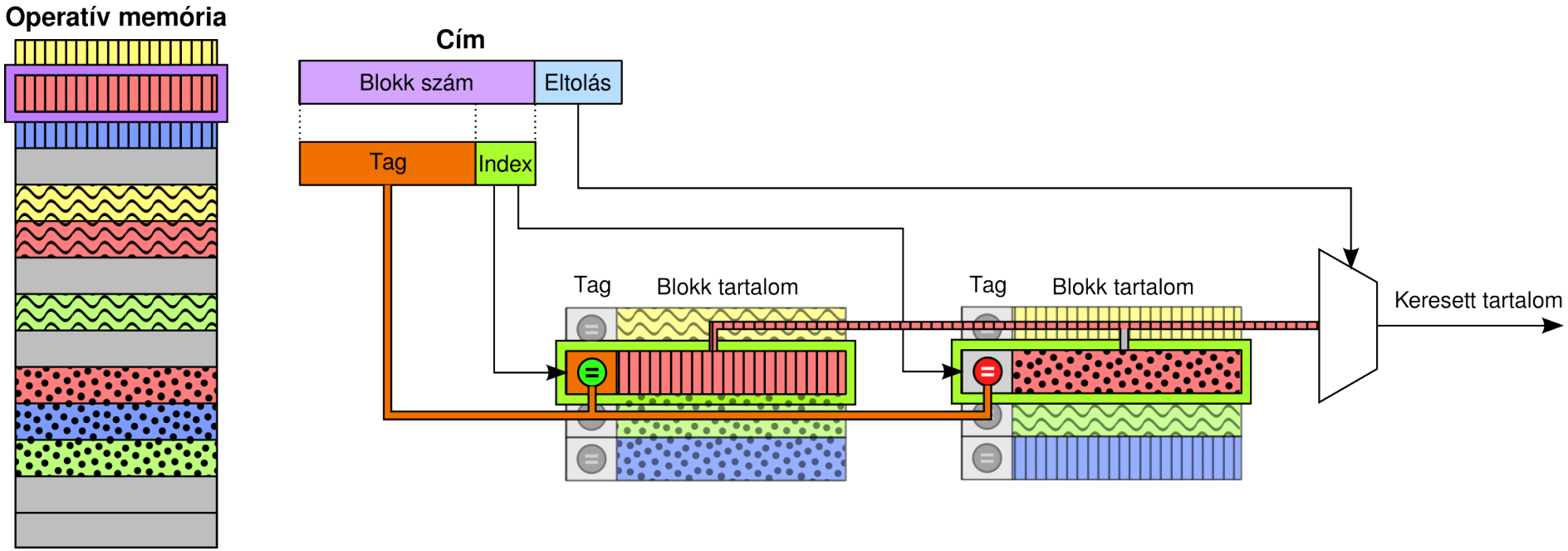
N-UTAS ASSZOCIATÍV SZERVEZÉS

- 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 (Kaby Lake)		AMD Ryzen		ARM Cortex A53		ARM Cortex A75	
	n=	méret	n=	méret	n=	méret	n=	méret
L1	8	32 KB	4/8	64/32 KB	2/4	16-64 KB	4	64 KB
L2	4	256 KB	8	512 KB	16	128-2048 KB	8	256-512 KB
L3	16	2 MB/mag	16	2 MB/mag	-	-	16	1-4 MB

Cache tartalom menedzsment

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
- Jön: hogy menedzseljük a cache tartalmát?
  - **Kit tegyünk bele?**
  - **Mikor tegyük bele?**
  - **Kit rakjunk ki?**
- Implicit / explicit

ADATOK BETÖLTÉSE A CACHE-BE

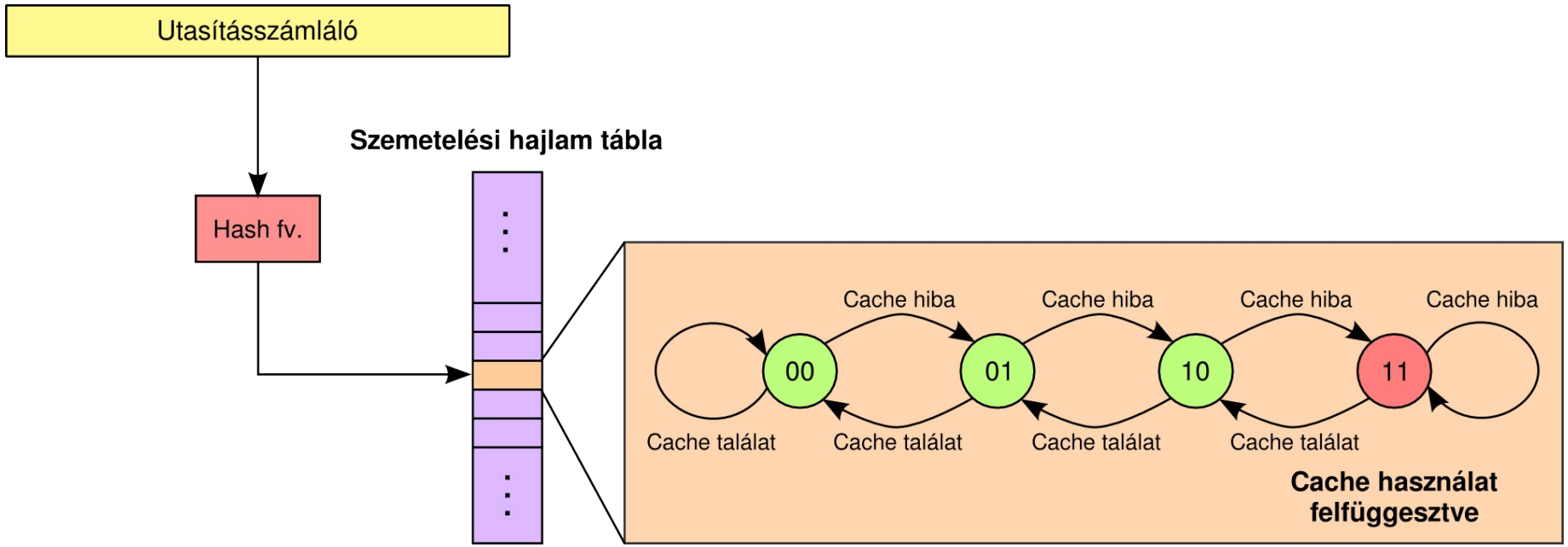
- Mikor tegyünk be egy memória blokkot a cache-be?
  - Soha
  - Amikor a futó program először hivatkozik rá
  - Jó előre, mielőtt még használnánk, jól jön az még
- **Soha**: cache szemetelés ellen véd
  - pl. médialejátszás: kijátszik egy képet, soha többet nem kell.
    → nem érdemes cache-be betölteni a képet
- **Első hivatkozáskor**: elsőre sokáig tart (memória → cache átvitel), de később gyors lesz
- **Idő előtti betöltés** (prefetch): spekulációt igényel

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:
  - Adatmozgatás a cache megkerülésével
  - Szinte minden architektúrán:
    - **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 algoritmus. 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)

- ***Adat-cache*** esetén nehéz:
  - 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)

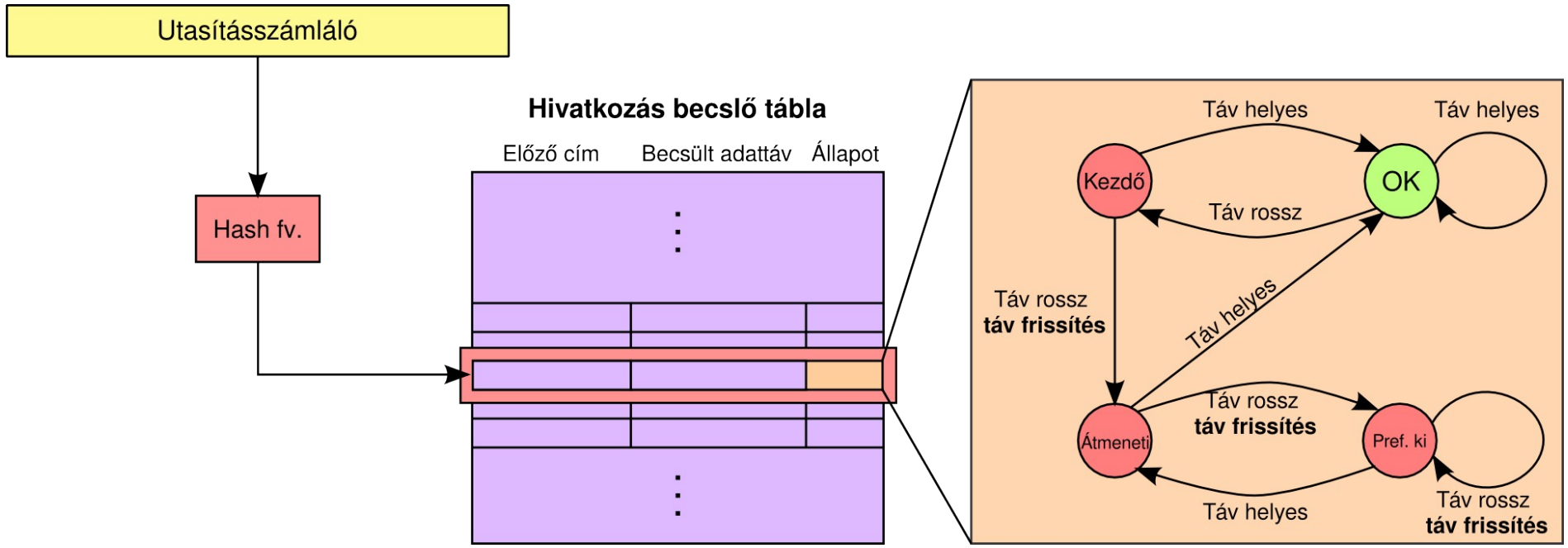
- Explicit prefetch utasítások:
  - **x86**: PREFETCHT0/1/2, PREFETCHNTA
    - behoznak egy blokkot a cache-be
    - PREFETCHNTA: a több utas cache első blokkjába teszi a behozott adatot → elkerülhető a szemetelés is!
  - **PowerPC**: DCBT, DCBTST
    - egy blokk cache-be töltése olvasásra/írásra
  - **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

- 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

ÍRÁSI MŰVELETEK

- Miért kell csínján bánni az írással?
  - Memória osztott erőforrás
    - Több processzorra közös
    - Perifériák is használják (DMA)
  - Ha módosítunk egy cache-blokkot, a memória tartalma nem lesz naprakész!
- Stratégiák a memóriatartalom frissítésére:
  - **Write-through**

    → cache-beli írást rögtön átvezeti a memóriába
  - **Write-back**

    → csak akkor vezeti át, amikor kikerül a cache-ből
- A cache gyors, a memória lassú. Bírja a tempót a sok visszaírással?
  - Az erre szánt blokk egy **write-buffer**-be kerül
  - Ahogy bírja (lassan), írja a memóriába
  - Ha valakinek kell valami, először itt kell keresni

TÖBBSZINTŰ CACHE MEMÓRIÁK

- Kisebb cache → kisebb késleltetés
- Több szintet használnak
  - Méret: nő
  - Sebesség: csökken
  - Ha nincs L1 találat, L2-t nézi, majd L3-at, stb.
  - Más lehet a blokkméret, szervezés, menedzsment, stb.
- Más a cél:
  - **L1 cache célja: minél kisebb késleltetés**
    - Kis méret, alacsony asszociativitás
  - **Ln (n>1) cache célja: cache hiba-arány minimalizálása**
    - Nagyobb méret, magas asszociativitás

HÁLÓZATI RENDSZEREK
ÉS SZOLGÁLTATÁSOK
TANSZÉK

HÁLÓZATI RENDSZEREK ÉS SZOLGÁLTATÁSOK TANSZÉK

MŰEGYETEM 1782

Budapest,
2022.03.29.

SZÁMÍTÓGÉP ARCHITEKTÚRÁK

Lokalitástudatos programozás

Horváth Gábor, Belső Zoltán

BME Hálózati Rendszerek és Szolgáltatások Tanszék
ghorvath@hit.bme.hu, belso@hit.bme.hu

A LOKALITÁS JELENTŐSÉGE

- Hogyan lehet **lassú** programot írni?
  - Címezgessünk össze-vissza a memóriában!
    - Sok cache-hiba lesz
    - Sok TLB hiba lesz
    - Sok laphiba lesz
    - Sokszor kell új sort nyitni a DRAM-ban
- Hogy lehet **gyors** programot írni?
  - Dolgozzunk a tárhierarchia keze alá!
  - Memóriahivatkozásaink legyenek
    - Térben lokálisak
    - Időben lokálisak
- Tartalomjegyzék:

1. Megéri egyáltalán?
2. Ha igen, mit tehetünk C programozóként?

A lokális viselkedés hatásának számszerűsítése

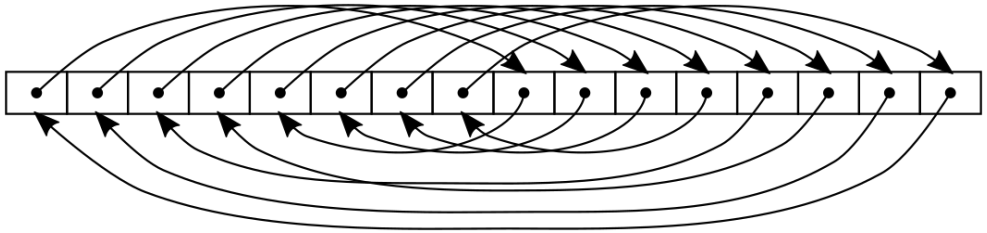
Avagy: megéri ezzel a témával foglalkozni?

MIT MÉRÜNK?

- Az időbeli lokalitás hatását
  - Mennyit számít, hogy a program minél hamarabb visszatér ugyanarra a memóriacímre?
- A térbeli lokalitás hatását
  - Mennyit számít, hogy a program a memóriát szépen sorban, szabályosan járja be?

AZ IDŐBELI LOKALITÁS HATÁSÁNAK MÉRÉSE

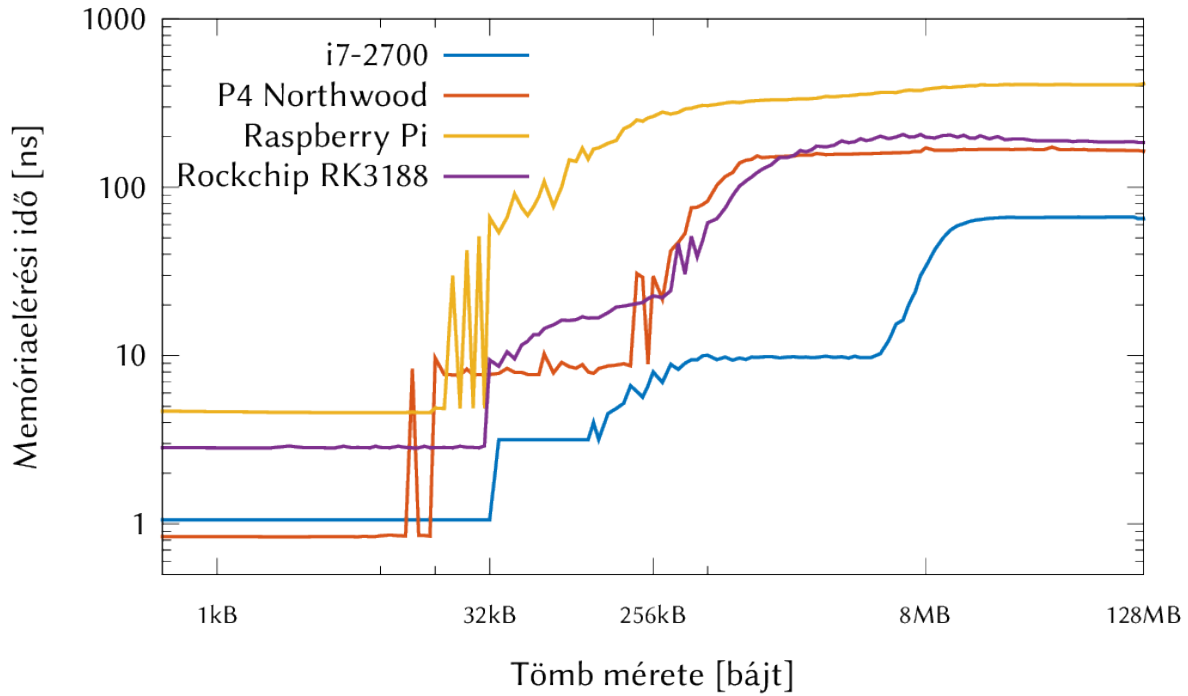
- A mérés módja:
  - Vegyünk egy tömböt (N)
  - Tömbelemek: pointerek a tömb más elemeire
  - A pointerlánc minden elemet érintsen, de legyen összevissza



- Mérjük a futási időt N függvényében

```
for (int i=0; i<iterations/100; i++) {
    p = *p;
    p = *p;
    ...
    p = *p;
}
```

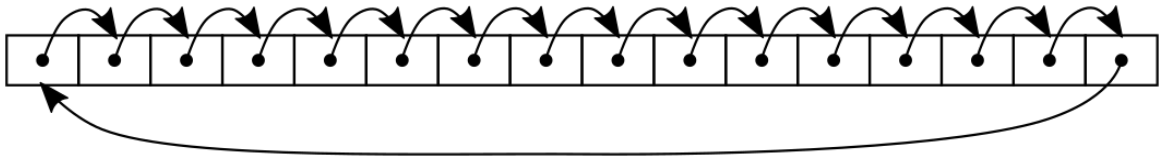
AZ IDŐBELI LOKALITÁS HATÁSÁNAK MÉRÉSE



- Értékelés:
  - Behatárolható a tárhierarchia szintjeinek mérete!
  - Üzenet:
    - Az időbeli lokalitás nagyon sokat számít
    - **Különbség akár 200-szoros!!!**

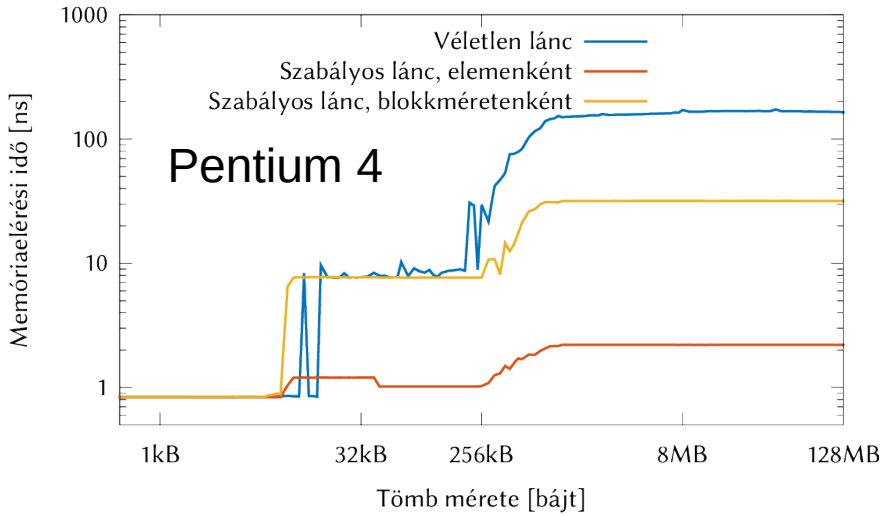
A TÉRBELI LOKALITÁS HATÁSÁNAK MÉRÉSE

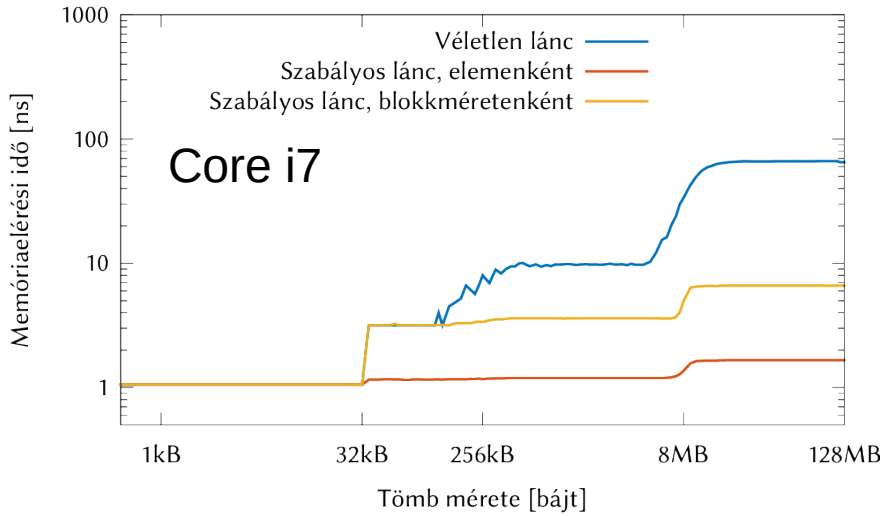
- A mérés módja
  - Mint eddig, de most láncoljuk az elemeket szépen sorba

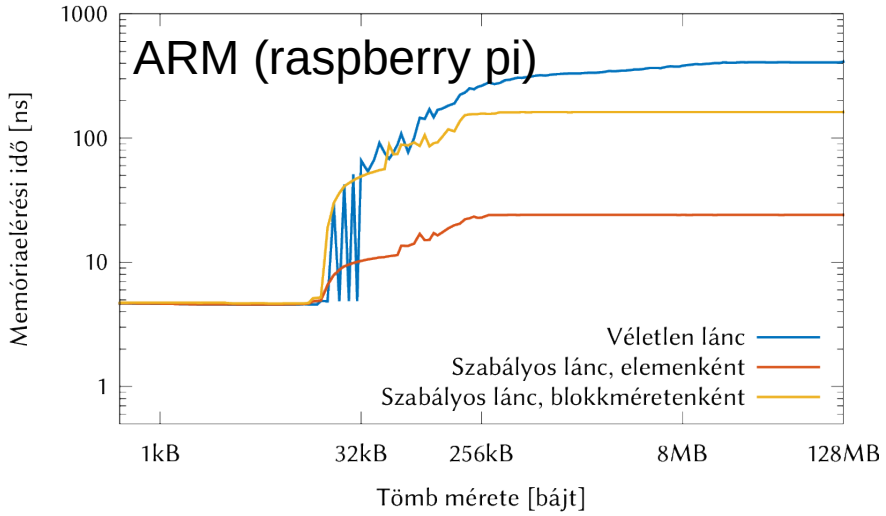


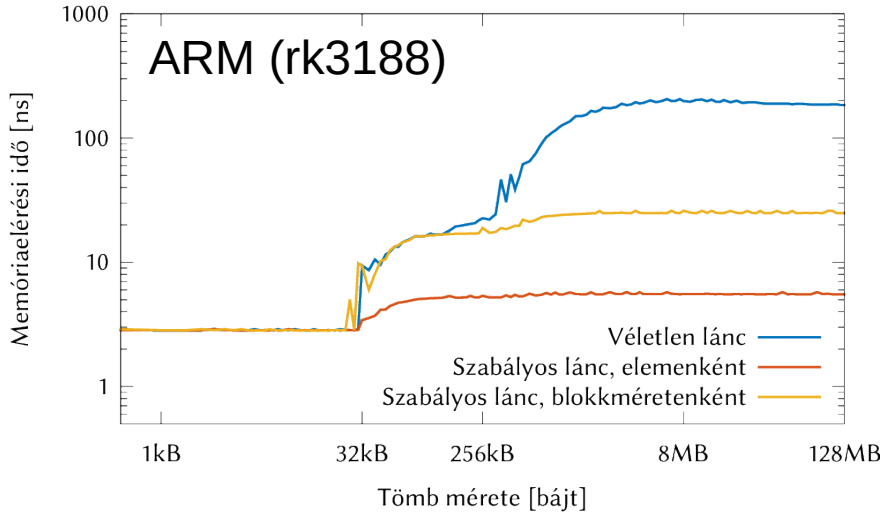
- Várt eredmény: alacsonyabb késleltetés
  - Csökken a cache hiba-arány:
    - Ha egy cache blokkra léptünk, végigmegyünk a blokk többi elemén is
    - Ha van prefetch algoritmus, rá tud tanulni, előbetölti a blokkokat
  - Csökken a TLB hiba-arány:
    - Ha egy lapra léptünk, a lap végigjárásához ugyanaz a laptábla bejegyzés kell

A TÉRBELI LOKALITÁS HATÁSÁNAK MÉRÉSE









A TÉRBELI LOKALITÁS HATÁSÁNAK MÉRÉSE

- Tanulság:
  - Kifizetődő a sorrendi bejárás
  - Nagy tömbmérettel kb. 40-80-szoros különbség

Lokalitásbarát ciklusszervezés

Avagy: mit tehet egy C programozó?

CIKLUSEGYESÍTÉS

Eredeti C kód:

```
for (i=0; i<N; i++)
    b[i] = c * a[i] + x;
sum = 0;
for (i=0; i<N; i++)
    sum += b[i];
for (i=0; i<N; i++)
    d[i] = a[i] + b[i];
```

Ciklusegyesítéssel:

```
sum = 0;
for (i=0; i<N; i++) {
    b[i] = c * a[i] + x;
    sum += b[i];
    d[i] = a[i] + b[i];
}
```

- Cache hiba-analízis: (felt.: 8 double fér egy blokkba, N nagy)
- Eredeti kód:
  - Első ciklus: 2N hivatkozás, 2N/8 cache hiba
  - Második ciklus: N hivatkozás, N/8 hiba
  - Harmadik ciklus: 3N hivatkozás, 3N/8 hiba
- Összesen: 6N hivatkozás, 6N/8 cache hiba

  → **Cache hiba-arány: 1/8 = 12,5%**

CIKLUSEGYESÍTÉS

Eredeti C kód:

```
for (i=0; i<N; i++)
    b[i] = c * a[i] + x;
sum = 0;
for (i=0; i<N; i++)
    sum += b[i];
for (i=0; i<N; i++)
    d[i] = a[i] + b[i];
```

Ciklusegyesítéssel:

```
sum = 0;
for (i=0; i<N; i++) {
    b[i] = c * a[i] + x;
    sum += b[i];
    d[i] = a[i] + b[i];
}
```

- Ciklusegyesítéssel:
  - Ciklus első sora: 2N hivatkozás, 2N/8 cache hiba
  - Második sor: N hivatkozás, 0 cache hiba!
  - Harmadik sor: 3N hivatkozás, N/8 hiba (d[i] miatt)
- Összesen: 6N hivatkozás, 3N/8 cache hiba

  → **Cache hiba-arány: 1/16 = 6,25%**

CIKLUSEGYESÍTÉS

- Tanulság:
  - Kerüljük a tömbök ismételt bejárását
  - Amit lehet, vonjunk be közös ciklusba
- Mérési eredmények:
  - $N=2^{22}$

	i7-2600	p4	Rasp. Pi	RK3188
Eredeti algoritmus	16,533 ms	109,974 ms	698,450 ms	115,354 ms
Ciklusegyesítéssel	8,469 ms	84,917 ms	203,755 ms	97,126 ms

CIKLUSSORREND OPTIMALIZÁLÁS

Bejárás sor-folytonosan:

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        sum += a[i][j];
```

Bejárás oszlop-folytonosan:

```
for (j=0; j<N; j++)
    for (i=0; i<N; i++)
        sum += a[i][j];
```

- C nyelv: tömbök tárolása sor-folytonos
- Feltételezés: 8 double/cache blokk, N nagy
- Cache hiba-analízis:
- Sor-folytonos bejárás:
  - Memóriában folytonosan végigmegy a tömb minden elemén
  - Láttuk, hogy ez milyen előnyös
  - Cache hiba: minden 8. elem elérésekor
  - **Cache hiba-arány: 1/8 = 12,5%**

CIKLUSSORREND OPTIMALIZÁLÁS

Bejárás sor-folytonosan:

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        sum += a[i][j];
```

Bejárás oszlop-folytonosan:

```
for (j=0; j<N; j++)
    for (i=0; i<N; i++)
        sum += a[i][j];
```

- Oszlop-folytonos bejárás:
  - Minden lépésben N elemnyit ugrunk
  - Ha van cache prefetch, az rátanul, és előbetölt
  - Ha nincs, és N > cache méret:
    - Mire j-t lépteti, a blokk kiszorul a cache-ből
    - Minden elérés cache-hibával jár!
    - **Cache hiba-arány: 100%**

CIKLUSSORREND OPTIMALIZÁLÁS

- Tanulság:
  - Bejáráskor igazodjunk az adatszerkezet memóriabeli elhelyezkedéséhez

- Mérési eredmények:
  - N=2048

	i7-2600	p4	Rasp. Pi	RK3188
Sor-folytonos	6,312 ms	8,973 ms	605,757 ms	14,879 ms
Oszlop-folytonos	6,926 ms	160,78 ms	4363,13 ms	60,96 ms

(az i7-nek jó cache prefetch algoritmusa van)

BLOKKOS CIKLUSSZERVEZÉS

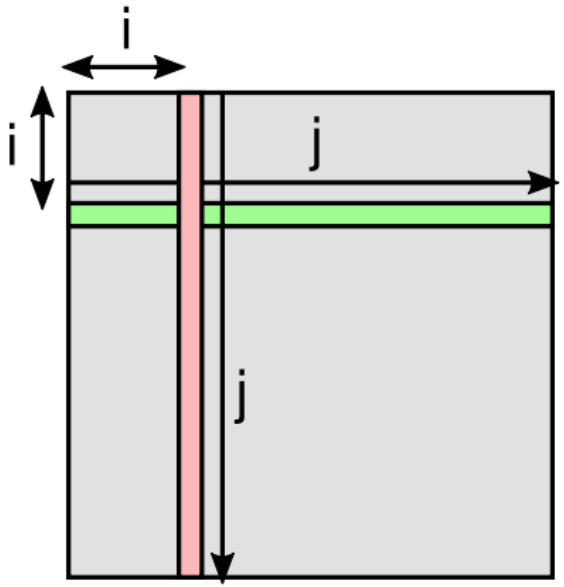
Eredeti C kód:

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        b[j][i] = a[i][j];
```

Blokkos ciklusszervezéssel:

```
for (bi=0; bi<=N-BLK; bi+=BLK)
    for (bj=0; bj<=N-BLK; bj+=BLK)
        for (i=bi; i<bi+BLK; i++)
            for (j=bj; j<bj+BLK; j++)
                b[j][i] = a[i][j];
```

- Mátrix transzponálás (kép forgatás, stb.)
- Feltételezés: 8 double/cache blokk, N nagy
- Cache hiba-analízis:
- Eredeti kód:
  - a[i][j]: sor-folytonos elérés
    → N^2 hivatkozás, $N^2/8$ cache hiba
  - b[j][i]: oszlop-folytonos elérés
    → N^2 hivatkozás, N^2 cache hiba
  - Összesen: $2N^2$ hivatkozás, $N^2/8 + N^2$ cache hiba
  - **Cache hiba-arány: 9/16 = 56,25%**



BLOKKOS CIKLUSSZERVEZÉS

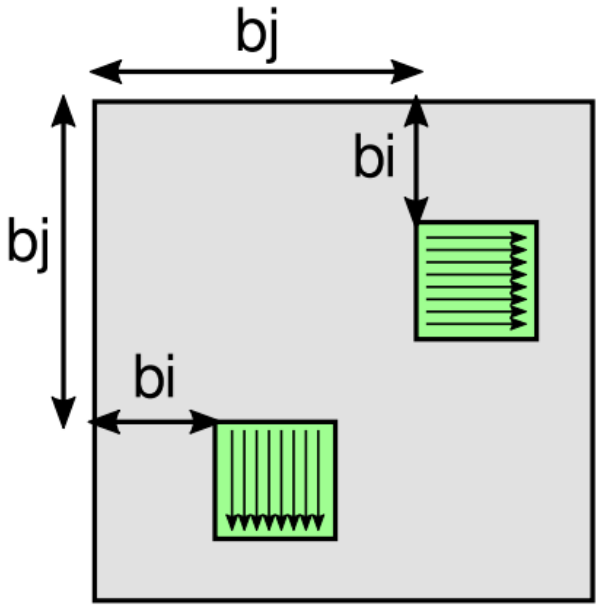
Eredeti C kód:

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        b[j][i] = a[i][j];
```

Blokkos ciklusszervezéssel:

```
for (bi=0; bi<=N-BLK; bi+=BLK)
    for (bj=0; bj<=N-BLK; bj+=BLK)
        for (i=bi; i<bi+BLK; i++)
            for (j=bj; j<bj+BLK; j++)
                b[j][i] = a[i][j];
```

- Blokkos cikusszervezéssel:
  - Blokkonként haladunk
  - Ha jó a BLK, egy BLK x BLK méretű tömbdarab befér a cache-be
  - a[i][j] és b[j][i] is a cache memóriában lesz!
  - a[i][j]: sor-folytonos elérés, → N^2 hivatkozás, $N^2/8$ cache hiba
  - b[j][i]: oszlop-folytonos elérés, → N^2 hivatkozás, $N^2/8$ cache hiba
  - Összesen: $2N^2$ hivatkozás, $N^2/8$ +$N^2/8$ cache hiba
  - **Cache hiba-arány: 1/8 = 12,5%**

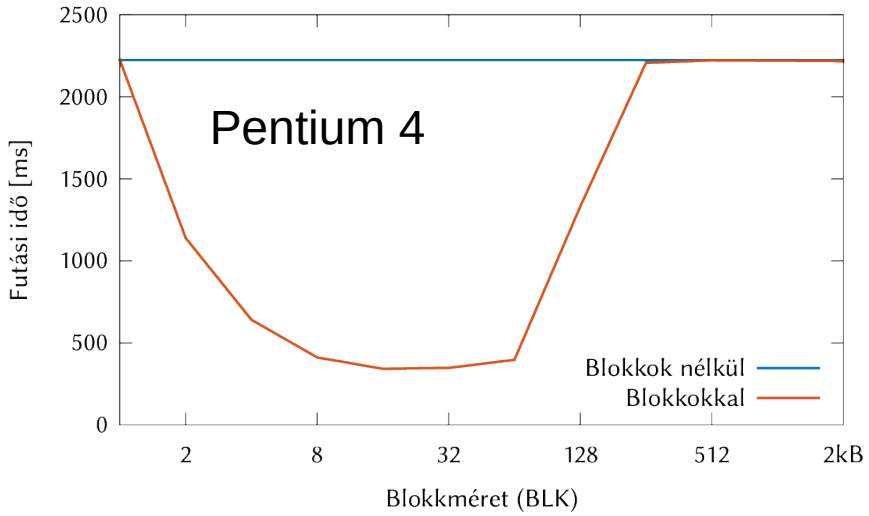


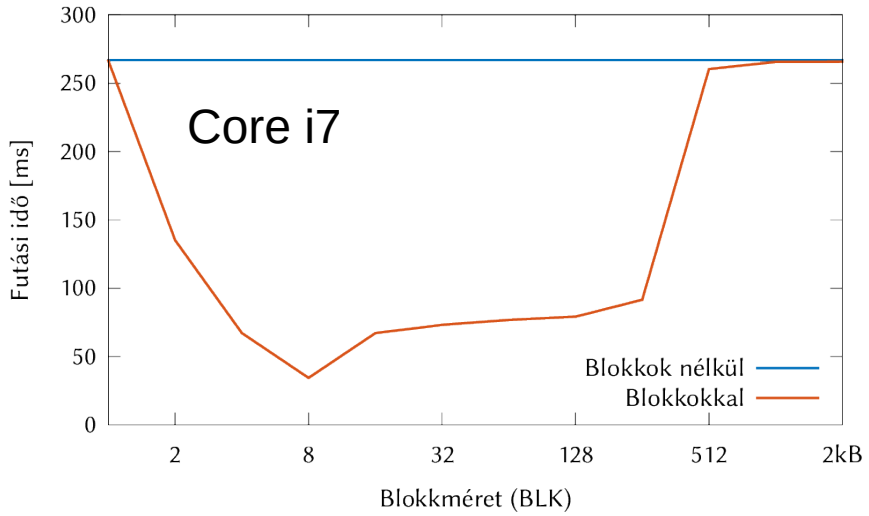
BLOKKOS CIKLUSSZERVEZÉS

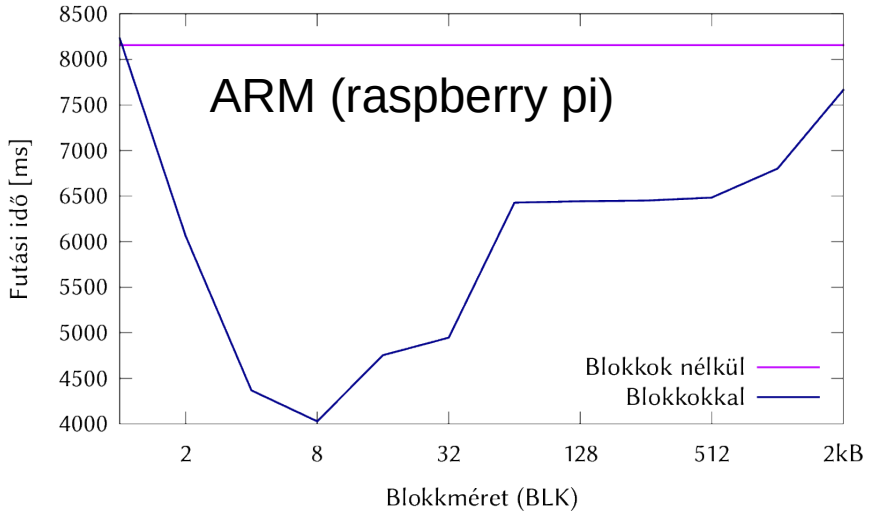
- Mi legyen a blokkméret?
  - Túl kicsi → olyan mintha nem lenne blokkszervezés
  - Túl nagy → olyan mintha nem lenne blokkszervezés
  - Architektúrafüggő!

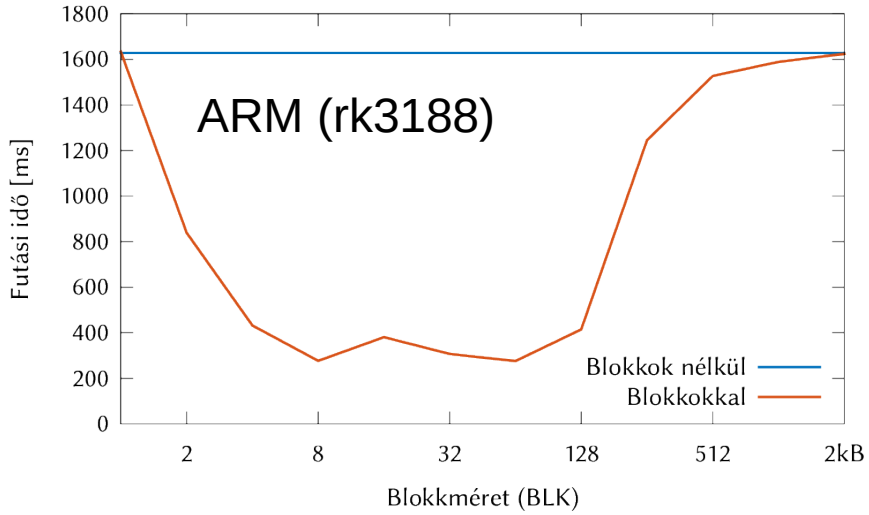
- Mérési eredmények:
  - N=2048
  - BLK 1-től 2048

BLOKKOS CIKLUSSZERVEZÉS









Esettanulmány: mátrixszorzás hatékonyan

MÁTRIXSZORZÁS

- $C = A \cdot B$
  - $c_{ij} = \sum_k a_{ik} \cdot b_{kj}$
- Naiv algoritmus:

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        for (k=0; k<N; k++)
            c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

- Kipróbáljuk a
  - ciklussorrend optimalizálását
  - blokkos ciklusszervezést

CIKLUSSORREND OPTIMALIZÁLÁS

- (ijk) sorrend:

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        for (k=0; k<N; k++)
            c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

- $4N^3$ memóriaművelet:
  - c[i][j] olvasás, a[i][k] olvasás, b[k][j] olvasás, c[i][j] írás
- c[i][j] olv.: sor-folytonos bejárás+belső ciklus ugyanazt az elemet éri el: $N^2/8$ cache hiba
- a[i][k] olv.: sor-folytonos bejárás, minden sort N-szer: $N^3/8$ cache hiba
- b[k][j] olv.: oszlop-folytonos bejárás N-szer: N^3 cache hiba
- c[i][j] írás: soha nincs cache hiba, az olvasás behozta a cache-be
- Összegzés:
  - Aszimptotikus cache hiba-arány: $\lim_{N\to\infty} (N^2/8+N^3/8+N^3)/4N^3 =$ **28,125%**

CIKLUSSORREND OPTIMALIZÁLÁS

- (ikj) sorrend:

```
for (i=0; i<N; i++)
    for (k=0; k<N; k++)
        for (j=0; j<N; j++)
            c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

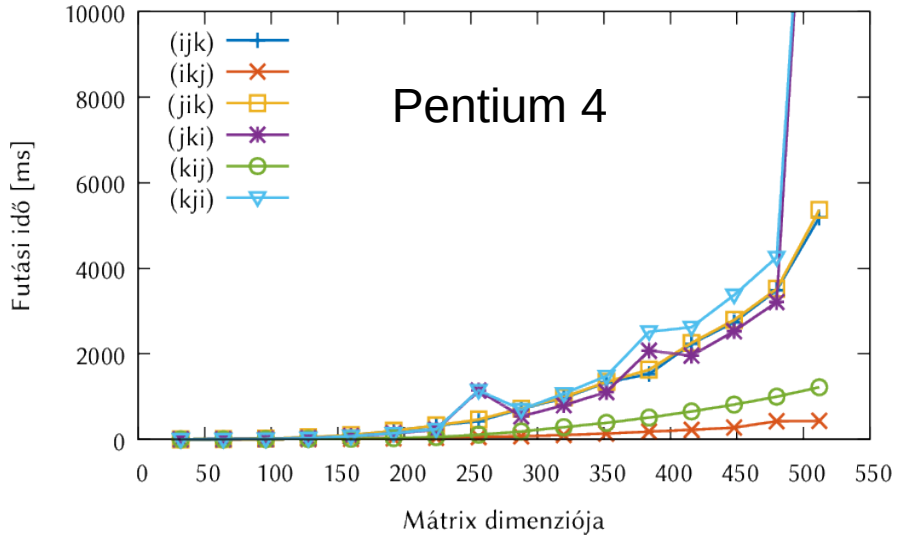
- c[i][j] olv.: sor-folytonos bejárás, minden sort N-szer:
  → $N^3/8$ cache hiba
- a[i][k] olv.: sor-folytonos bejárás+belső ciklus ugyanazt az elemet éri el: → $N^2/8$ cache hiba
- b[k][j] olv.: sor-folytonos bejárás N-szer: → $N^3/8$ cache hiba
- c[i][j] írás: soha nincs cache hiba, az olvasás behozta a cache-be
- Összegzés:
  - Aszimptotikus cache hiba-arány:
    $\lim_{N\to\infty} (N^3/8+N^2/8+N^3/8)/4N^3 =$ **6,25%**

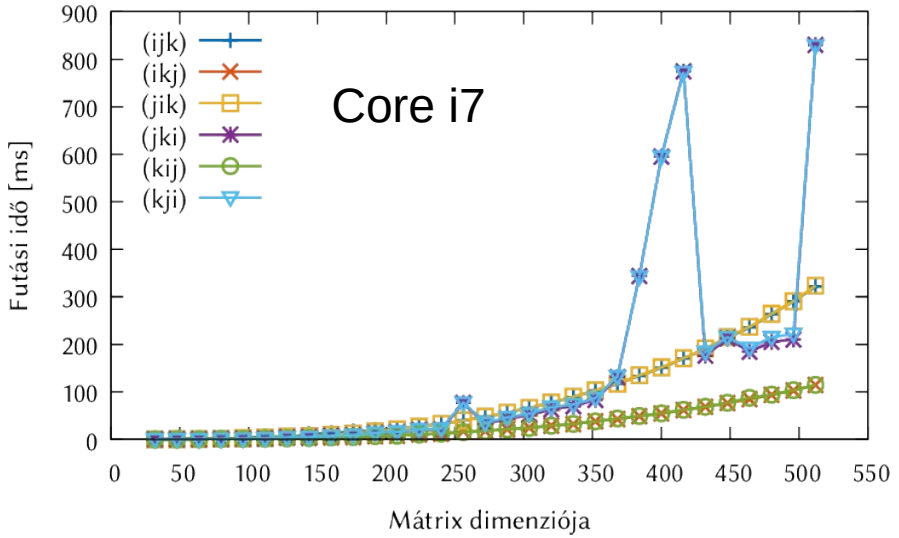
CIKLUSSORREND OPTIMALIZÁLÁS

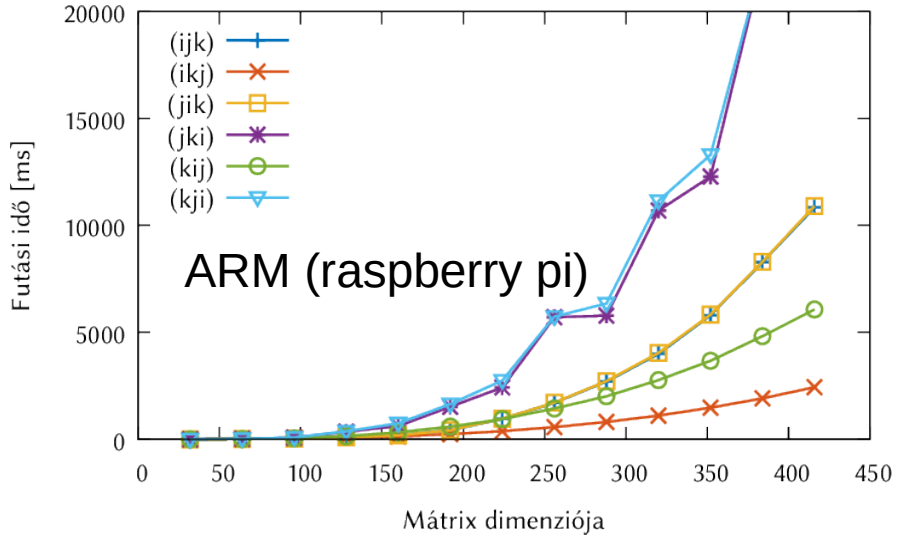
- 3 ciklus, 6-féle sorrend lehet
- Cache hiba-arányok:

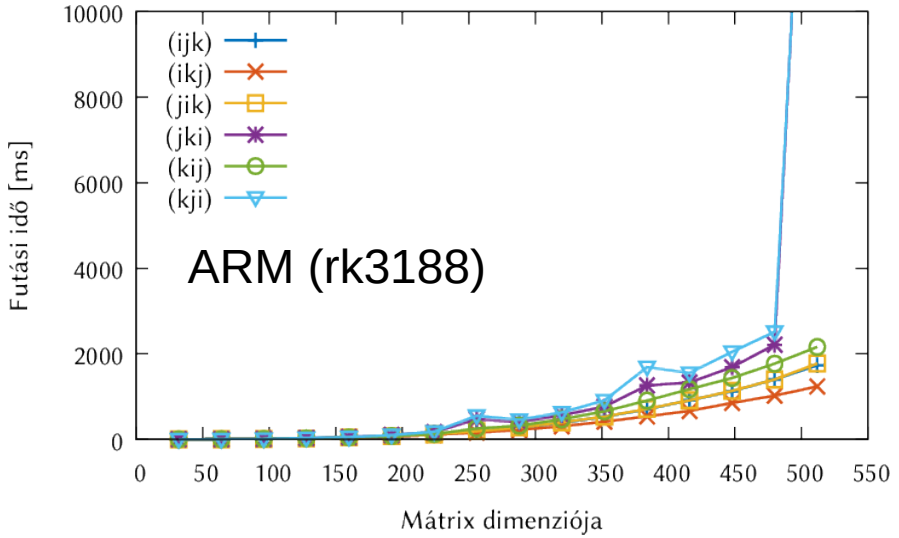
	c[i][j] olv.	a[i][k]	b[k][j]	c[i][j] ír.	Cache hiba-arány
(ijk)	$N^2/8$	$N^3/8$	N^3	0	28,125%
(ikj)	**$N^3/8$**	**$N^2/8$**	**$N^3/8$**	**0**	**6,25%**
(jik)	N^2	$N^3/8$	N^3	0	28,125%
(jki)	N^3	N^3	N^2	0	50%
(kij)	$N^3/8$	N^2	$N^3/8$	0	6,25%
(kji)	N^3	N^3	$N^2/8$	0	50%

CIKLUSSORREND OPTIMALIZÁLÁS









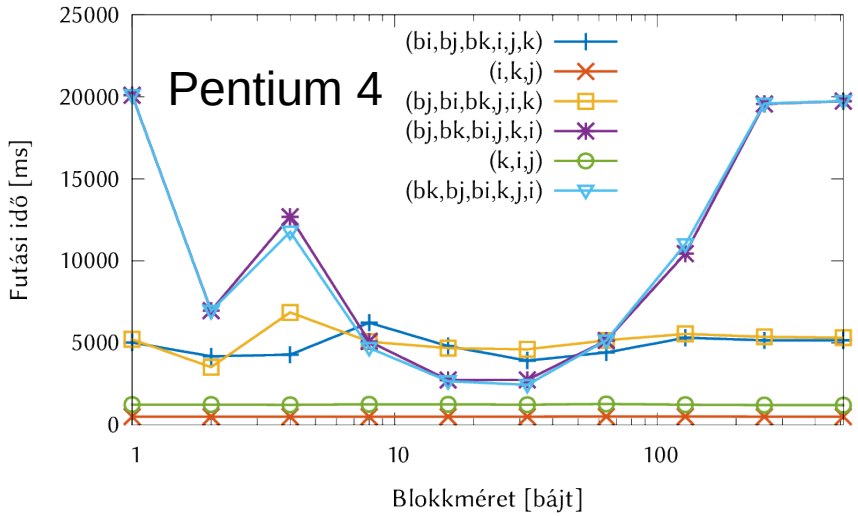
BLOKKOS CIKLUSSZERVEZÉS

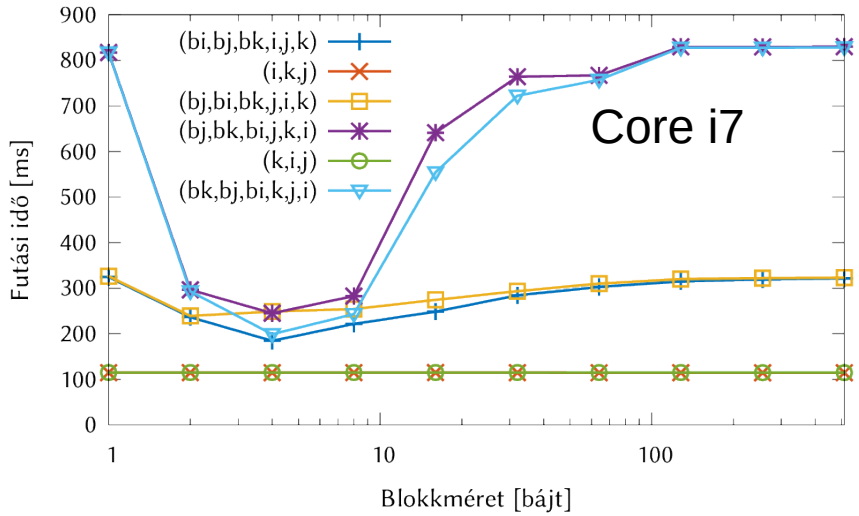
- Mivel architektúrafüggő, csak a kísérletezés marad
- Lehet mindhárom ciklus blokkosított:

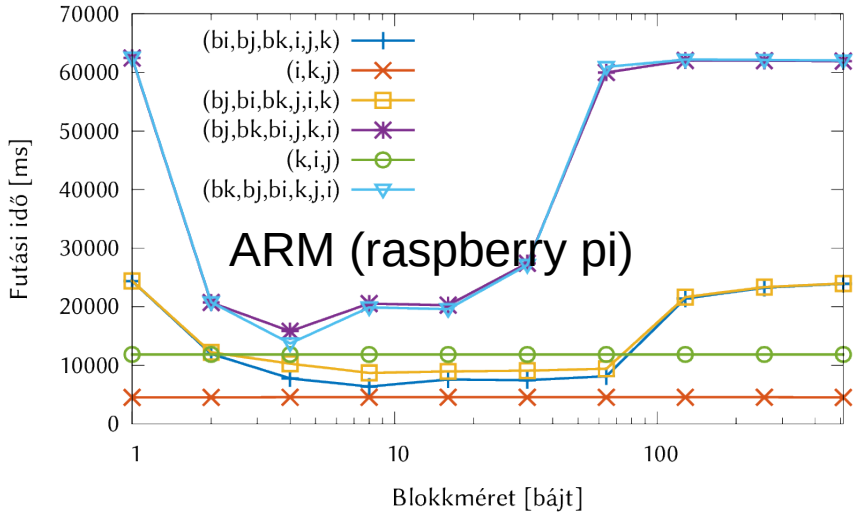
```
for (bi=0; bi<N; bi+=BLK)
    for (bj=0; bj<N; bj+=BLK)
        for (bk=0; bk<N; bk+=BLK)
            for (i=bi; i<bi+BLK; i++)
                for (j=bj; j<bj+BLK; j++)
                    for (k=bk; k<bk+BLK; k++)
                        c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

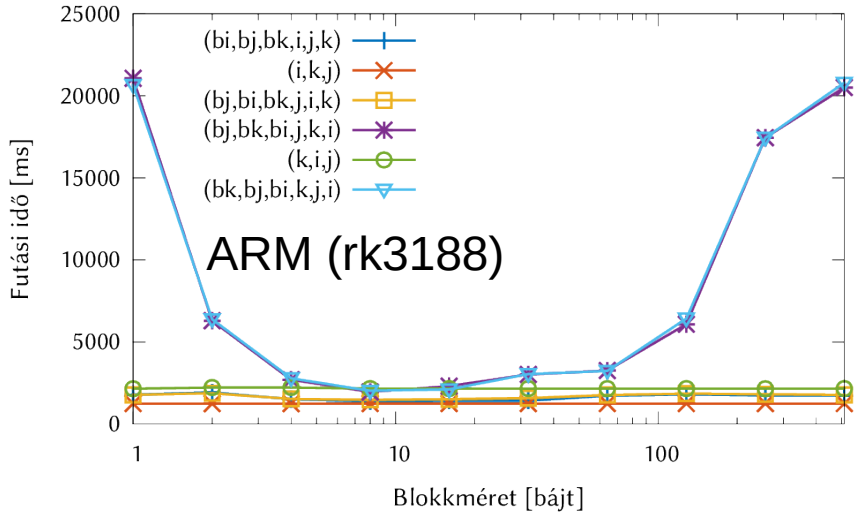
- … vagy csak kettő is.
- Mértünk
  - Az (ikj) és (kij) nem lett jobb a blokkoktól, a többi igen
  - A többinél az a legjobb, ha mindhárom ciklus blokkosított

CIKLUSSORREND OPTIMALIZÁLÁS









MÁTRIXSZORZÁS

- Konklúzió:
  - A naiv mátrixszorzó lassú
  - A blokkos szervezés sokat javít
  - De a legjobb ciklussorrend (ikj) anélkül is jobb
  - A legjobb ciklussorrend nem volt triviális, át kellett gondolni!

LOKALITÁSTUDATOS PROGRAMOZÁS

- Összegzés:
  - Sokszor lényegesen gyorsabb lesz a program, ha figyelünk a lokalitásra
  - Tanult technikák:
    - Ciklusegyesítés
    - Ciklussorrend optimalizálás
    - Blokkos ciklusszervezés
  - Több is van!
  - Vannak adatszerkezetek, melyek kifejezetten cache tudatosak
    - Rendezés: funnel sort
    - Mátrixműveletek: Morton reprezentáció
    - Stb.
- De!!!

  Premature optimization is the root of all evil
  (Donald Knuth)

HÁLÓZATI RENDSZEREK
ÉS SZOLGÁLTATÁSOK
TANSZÉK

HÁLÓZATI RENDSZEREK
ÉS SZOLGÁLTATÁSOK
TANSZÉK

MŰEGYETEM 1782

Budapest,
2022.04.06.

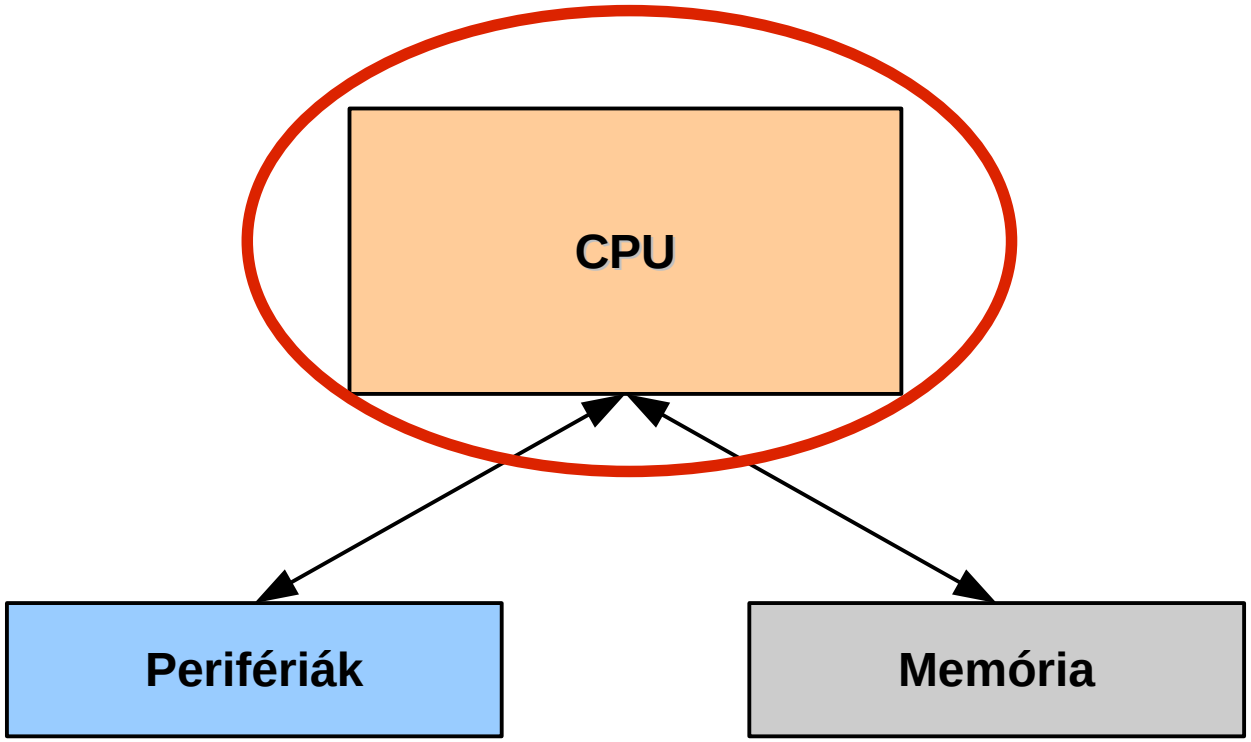
SZÁMÍTÓGÉP ARCHITEKTÚRÁK

Utasításkészlet architektúrák

Horváth Gábor, Belső Zoltán

BME Hálózati Rendszerek és Szolgáltatások Tanszék
ghorvath@hit.bme.hu, belso@hit.bme.hu

HOL TARTUNK?



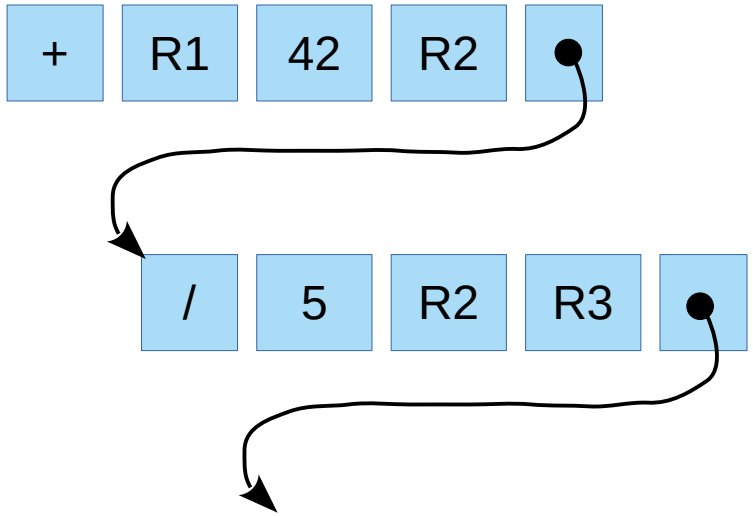
UTASÍTÁSKÉSZLETEK

- Processzorok jellemzője:

  Programozási felület / Utasításkészlet architektúra
- Részei:
  - Utasítások
  - Támogatott adattípusok
  - Regiszterek
  - Címzési módok
  - Jelzőbitek
  - Perifériakezelési mód
  - Megszakítás- és kivételkezelés

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

    `IO[42] ← R1, R1 ← IO[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

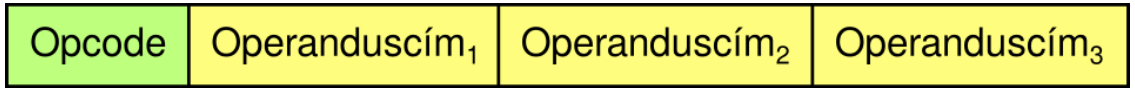
  R1 ← R2 + 32 IF P2

- Predikátumregiszter: végrehajtási feltételt tárol
- Állítása: összehasonlító műveletekkel

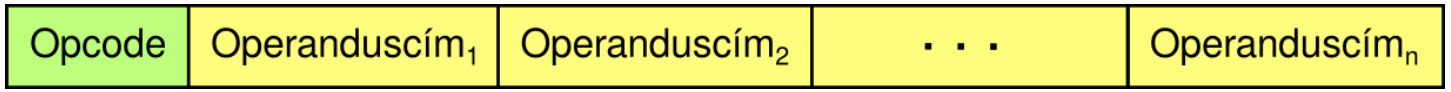
  P2 ← R3 ≤ R5

UTASÍTÁSOK JELLEMZŐI

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



  - Változó hosszú kódolás:

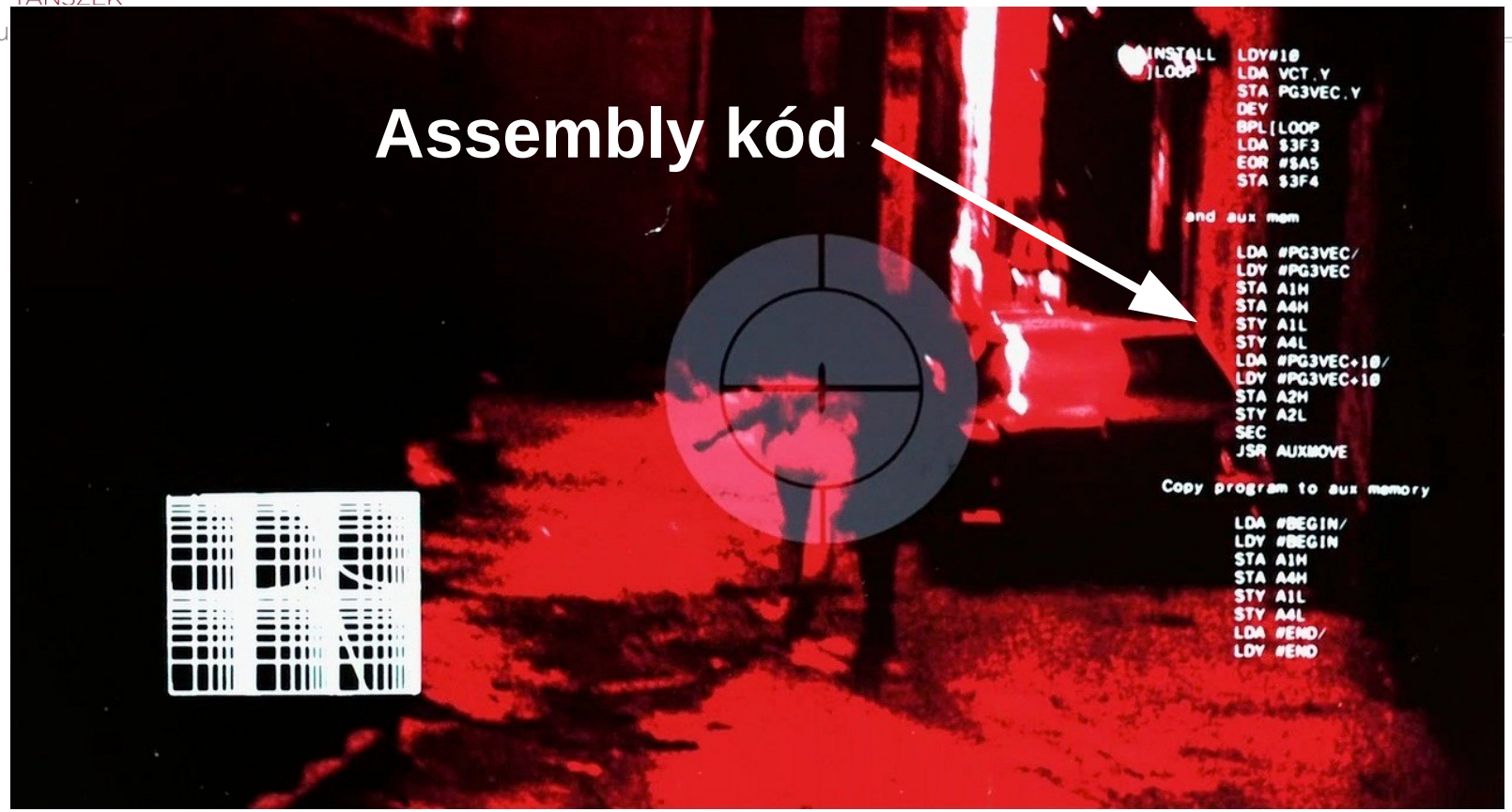


ASSEMBLY PROGRAMOZÁS

- Alacsony szintű programozás:
  - Utasítások kézi kódolása kényelmetlen
  - A binárisan kódolt utasítássorozat emberi fogyasztásra alkalmatlan
  - Eszköz: assembly programozás
- Assembly
  - A legalacsonyabb szintű programozási nyelv
  - A gépi utasítások szöveges megfelelője
    - 1 assembly „utasítás”-ból → 1 gépi utasítás lesz
  - Assembler: az assembly leírásból gépi kódot készít
  - Minden utasításkészlet architektúrára más és más!

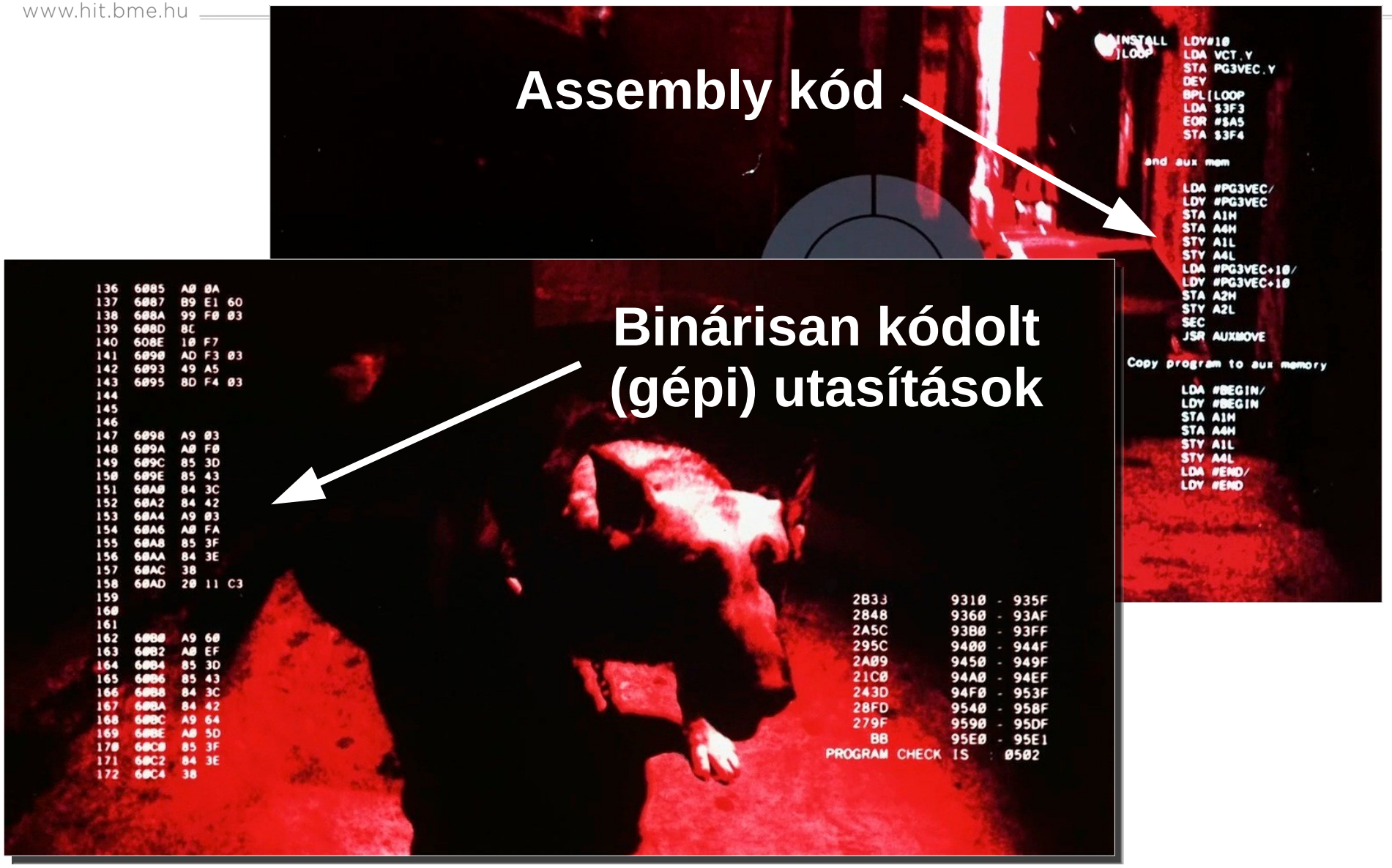
UTASÍTÁSKÓDOLÁS

UTASÍTÁSKÓDOLÁS



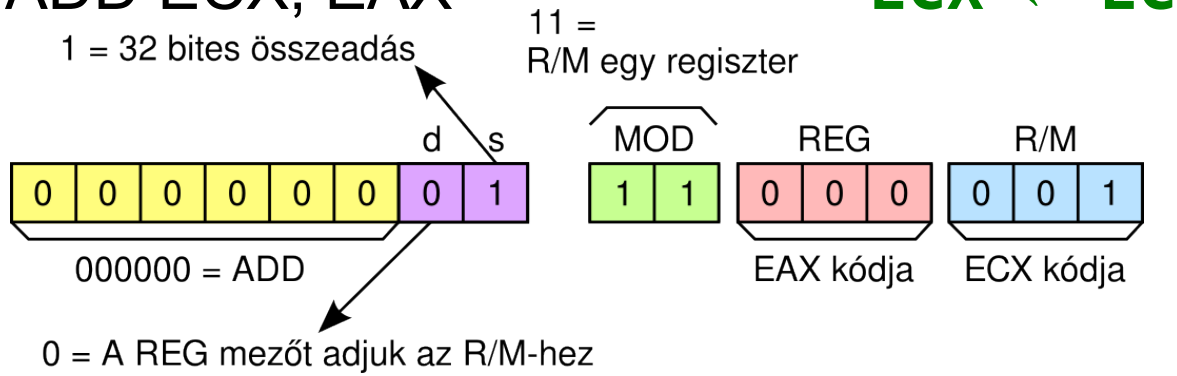
- A Terminátor processzora: MOS-6502 (akár az Apple II-é)
  (1975 és 1980 között messze a legolcsóbb CPU: a vele azonos képességű Intel és Motorola processzorok árának hatodába került)
- A Terminátor a Nibble magazin egyik példaprogramját futtatja

UTASÍTÁSKÓDOLÁS



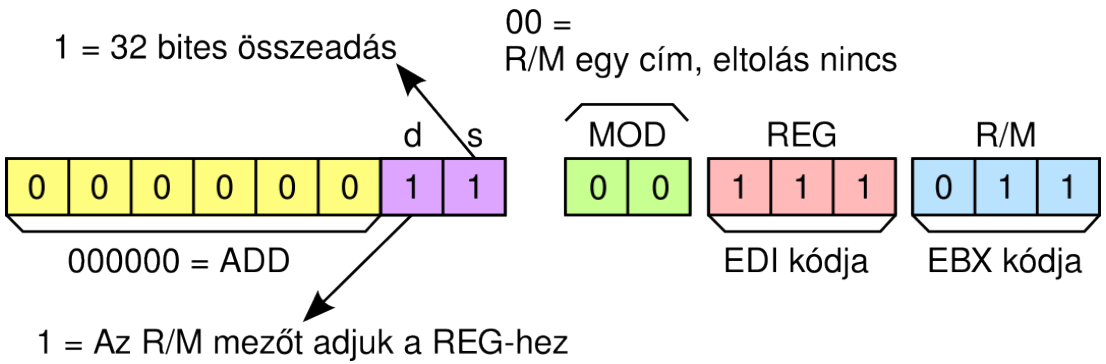
PÉLDA UTASÍTÁSKÓDOLÁSRA (X86)

- ADD ECX, EAX **ECX ← ECX + EAX**



= 01 C1
(ASCII: ☺┴)

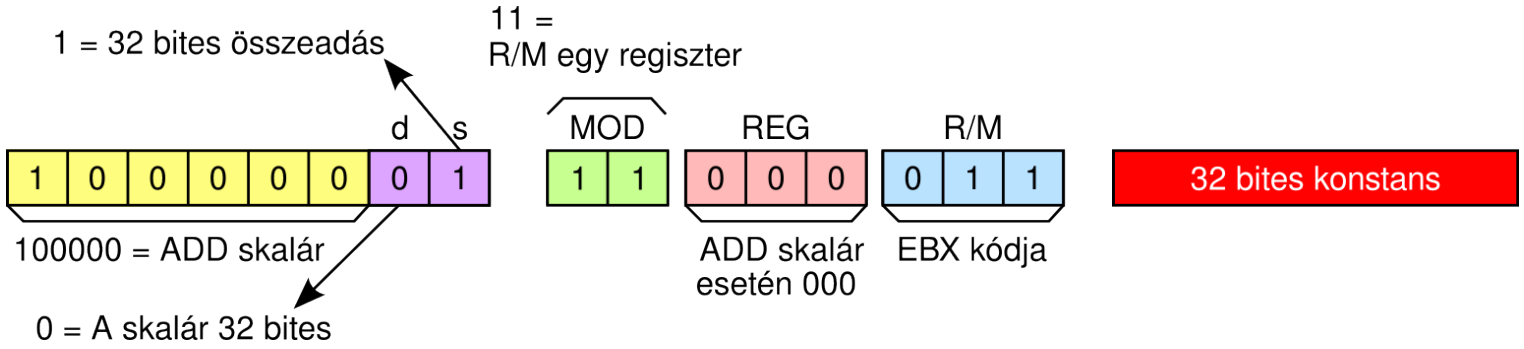
- ADD EDI, [EBX] **EDI ← EDI + MEM[EBX]**



= 03 3B
(ASCII: ;)

PÉLDA UTASÍTÁSKÓDOLÁSRA (X86)

- ADD EBX, 23423765 **EBX ← 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
  - Valamennyi utasítása valamennyi címzési módot használhatja

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 Instruction Set Computer
  - pl. verem alapú
- OISC: One Instruction 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

Elterjedt utasításkészletek

ELTERJEDT UTASÍTÁSKÉSZLETEK

x86

- Első tag: 1978, Intel 8086
  - Elődök: 1971, Intel 4004, 1972: 8008, 1974: 8080
- 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
- 43 éves kompatibilitási kényszer
- x86 licenccel rendelkeznek: Intel, AMD, (VIA)
- Intel stratégia: félvezetőipari beruházások
  - hatalmas gyártástechnológiai előny, mára elolvadt

ELTERJEDT UTASÍTÁSKÉSZLETEK

ARM

- Első megvalósítás: 1987
- A legelterjedtebb utasításkészlet architektúra
- Kezdetektől 32 bites, 2011 óta 64 bites
- Elsődleges szempont: egyszerűség és a fogyasztás
- Az ARM nem gyárt processzort, de licenszel
  - Utasításkészletet
  - Saját tervezésű processzort (pl. ARM Cortex processzorcsalád)
- Licenszelők:
  - Qualcomm: csak utasításkészletet vett, a processzor saját tervezésű
  - Apple: vett processzor licenszet, de továbbfejlesztette
  - Nvidia, Samsung, Huawei, Rockchip, Broadcom, stb.: ARM Cortex magokat licenszelnek, gyártanak
- Fejlődés:
  - 2008: A legerősebb ARM processzor: 680MHz (850 DMIPS, Apple iPhone 1: 412 MHz)
  - 2016: Samsung Exynos 8890: 2.3 GHz, 4 mag, 46920 DMIPS
  - 2021: 2.84 GHz (Snapdragon 888) / 3.2 GHz Apple M1

ELTERJEDT UTASÍTÁSKÉSZLETEK

Power

- PowerPC: 1991-ben, az IBM, Apple és Motorola összefogásában
  - A PC-kben nem terjedt el, munkaállomásokban és szerverekben igen
- IBM Power processzorok:
  - Felülmúlják az x86 processzorok teljesítményét
  - Óriási memória és I/O sávszélesség
  - 2007: 5 GHz (POWER6)
  - 2017: 12 magos, magonként 8 szál (POWER8)
  - 2019 november: a világ leggyorsabb szuperszámítógépe (Summit)
    - 9216 POWER9 core + 27648 Nvidia V100 GPU
  - 2021: POWER10 (15 db SMT8 mag, vagy 30 db SMT4 mag)

ELTERJEDT UTASÍTÁSKÉSZLETEK

SPARC

- 1987, SUN, most már Oracle
- Kezdetektől 64 bites
- Nyílt platform
  A UltraSPARC T1 és T2 VeriLog szinten elérhető!!!
- 2011: a világ legerősebb számítógépe SPARC alapú
  (2013: a negyedik legerősebb)
- 2013: SPARC T5: 3.6 GHz, 8 mag, 16 szál/mag
- 2016: SPARC M7: 4.13 GHz, 32 mag, 8 szál/mag (256 szál!)
  … és 16 foglalat/szerver!
- 2017: SPARC M8: 5 GHz, 32 mag, 8 szál/mag (256 szál!)
- 2017: Oracle leállítja a fejlesztéseket
- … de a Fujitsu folytatja!

ELTERJEDT UTASÍTÁSKÉSZLETEK

RISC-V

- 2010, University of California, Berkeley
- Cél: utasításkészlet architektúra létrehozása
  - Az akadémiai szféra számára is hozzáférhető
  - Nyílt licence, royalty mentes
  - Gyakorlatban is alkalmazható
- RISC-V Fundation: 2015, 2020 márciustól Svájcba költözött
- 32 és 64 bites változat, 128 bites tervezet
- RISC architektúra, 32 általános célú regiszter
- Több részből áll, kötelező és opcionális kiegészítésekkel
- Érdekesség: szubrutin hívás nem a stack-re menti a címet, hanem egy regiszterbe: jal (jump and link) és jalr (indirekt)
  - Az ortogonalitás jegyében ugyanezt használja a sima ugráshoz is

ELTERJEDT UTASÍTÁSKÉSZLETEK

RISC-V

- Jelentősebb implementációk:
  - SiFive: ez első
    + nagy teljesítményű core-ok: szuperskalár, soron kívüli végrehajtás, stb.
  - Google Titan M2 security module for the Pixel 6
  - Seagate a diszk vezérlőiben
  - Espressif ESP32-S2 mikrokontorller
- Fejlesztés alatt:
  - NVIDIA tervezi a Falcon processzort leváltani a GeForce grafikus kártyákon
- Nyílt implementációk:
  - Alibaba group XuanTie 910
  - Western Digital 2 utatas szuper-skalár implementáció SSD vezérlő céljára
- Szoftware support:
  - GCC, LLVM, QEMU, OpenJDK
  - Linux (csak 64 bites), FreeBDS, NetBSD, OpenBSD, FreeRTOS

A MÚLT

- **Alpha** (DEC, 1992)
  - 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
- **PA-RISC (**1986, HP)
  - Eleinte 32, majd 64 bites
  - Nagyon erős lebegőpontos egység
    PA-8600 @ 552 MHz > 2x Pentium III @ 1 GHz!
  - Halála: HP az Intel Itanium mellett teszi le a voksát

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
- 2018: bejelentik a kivezetését (2021-ig)

ÉRDEKESSÉG

- Jim Keller
  - Az Alpha 21164 és 21264 processzorok tervezője
  - 1998: Az AMD Athlon K7 és K8 processzorok tervezője (gyorsabbak voltak, mint az Intel P4!)
  - 2004: P.A. Semi vezető tervezője → Apple felvásárolja (2008), Apple A4 és A5 alapja (mindmáig meghatározza az Apple CPU-k erejét)
  - 2012: Vissza az AMD-hez, az új AMD Zen architektúra tervezését irányítja (megjelenés: 2017)
  - 2016: a Tesla alkalmazottja (Autopilot Hardware Engineering)
  - 2018 április: csatlakozik az Intelhez!
  - 2020 június: elhagyja az Intelt
  - Azóta: Tenstorrent

ÖSSZEHASONLÍTÁS

	x86	ARM	Power	SPARC
Hány bites	64	32/64	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	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	RISC-V
Hány bites	32	64	64	64	32, 64, (128)
Megjelenés éve	1979	1992	1986	2001	2010
Operandusok	2	3	3	3	3
Műveletek	Reg-mem	Reg-reg	Reg-reg	Reg-reg	Reg-reg
CISC vs. RISC	CISC	RISC	RISC	EPIC	RISC
Regiszterek sz.	16	32	32	128	32
Utasításkódolás	Vált. (2-22)	Fix (4)	Fix (4)	Fix (16)	Fix (4)
Felt. utasítások	Feltétel kód	Feltétel reg.	Összeh. & ugr.	?	Összeh. & ugr.
Bájtsorrend	Big	Bi	Big	Bi	Little
Címzési módok	9	1	5	?	3
Perifériakezelés	Mem. lekép.	Mem. lekép.	Mem. lekép.	Mem. lekép.	Mem. lekép.
Predikátumok	Nincs	Nincs	Nincs	Van	Nincs

HÁLÓZATI RENDSZEREK
ÉS SZOLGÁLTATÁSOK
TANSZÉK

HÁLÓZATI RENDSZEREK ÉS SZOLGÁLTATÁSOK TANSZÉK

SZÁMÍTÓGÉP ARCHITEKTÚRÁK

Pipeline utasításfeldolgozás

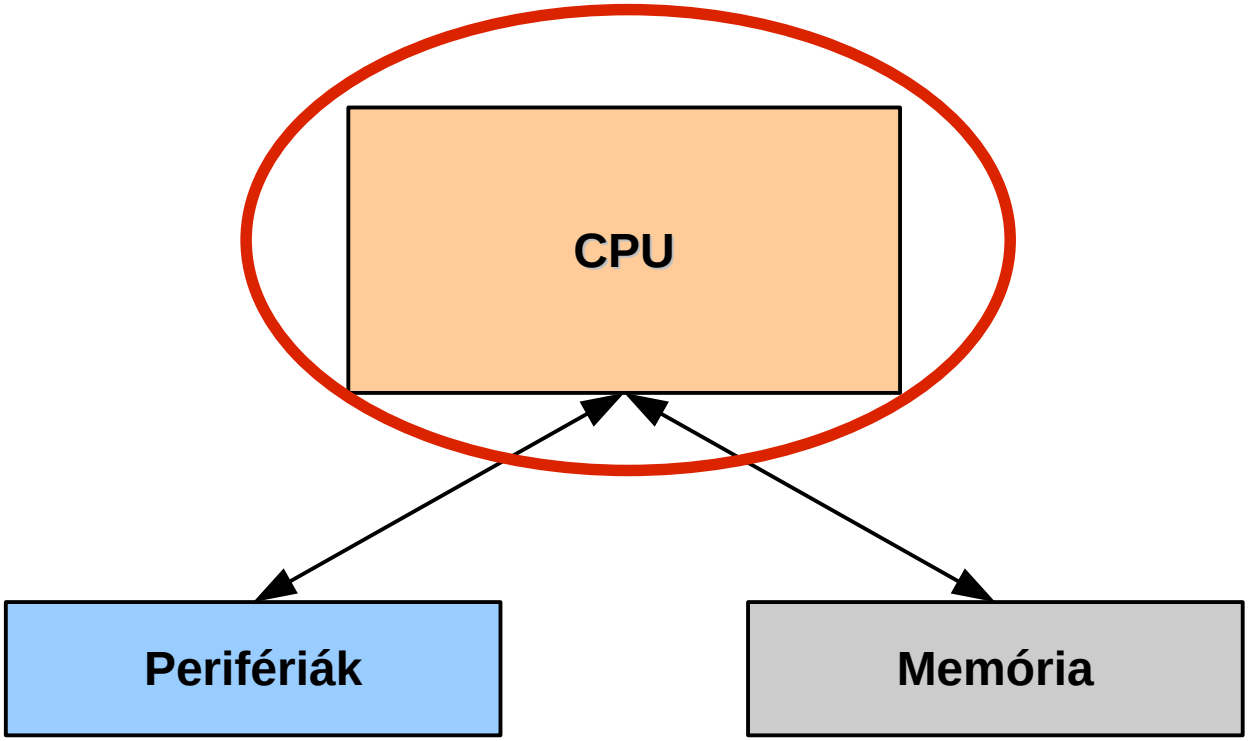
Horváth Gábor, Belső Zoltán

BME Hálózati Rendszerek és Szolgáltatások Tanszék
ghorvath@hit.bme.hu, belso@hit.bme.hu

Budapest,
2022.04.27.

MŰEGYETEM 1782

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

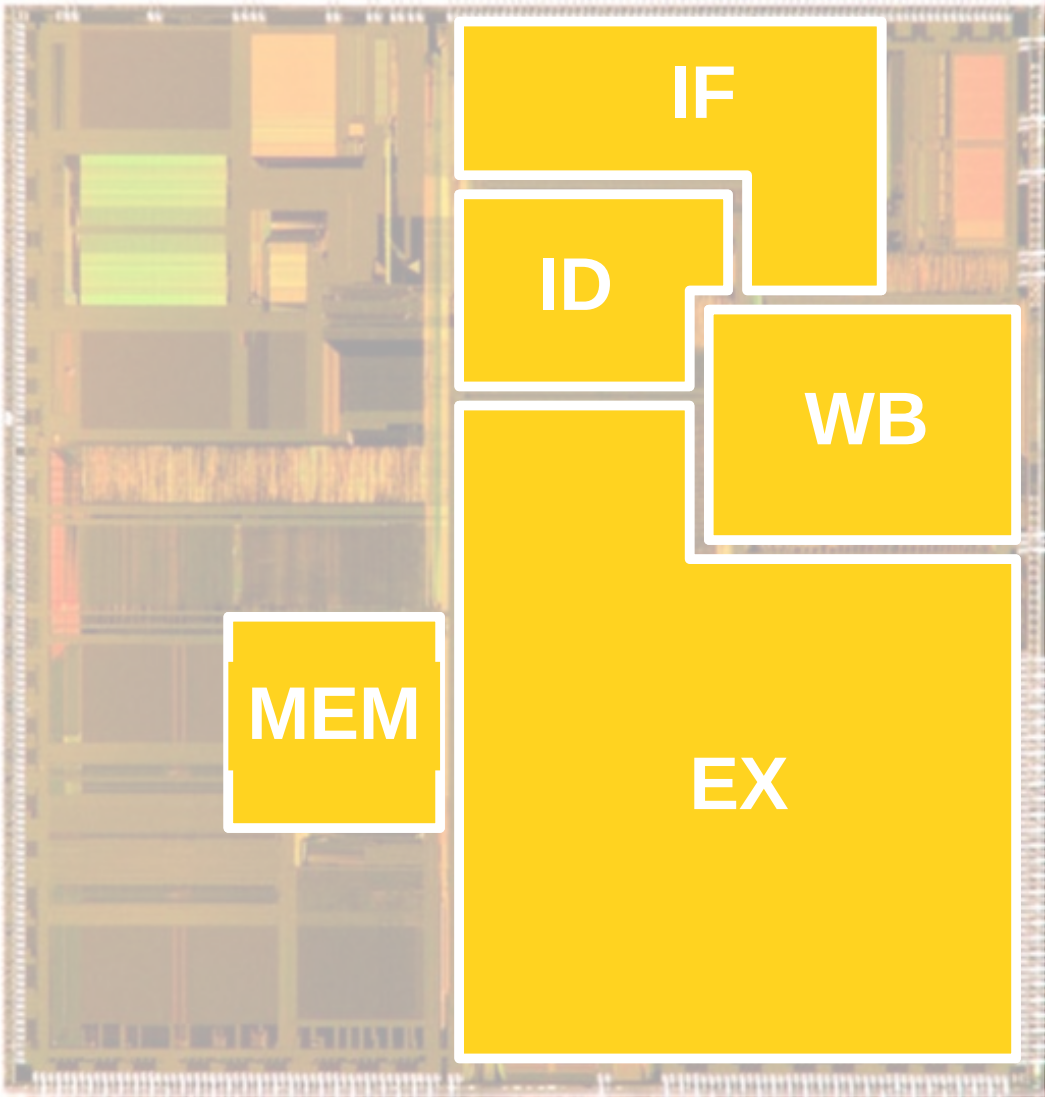
- … folytatás:

  4) Memóriaműveletek (Memory access, **MEM**)
     - Load/Store: végrehajtódik
     - Aritmetikai/logikai, ugrások: kihagyható

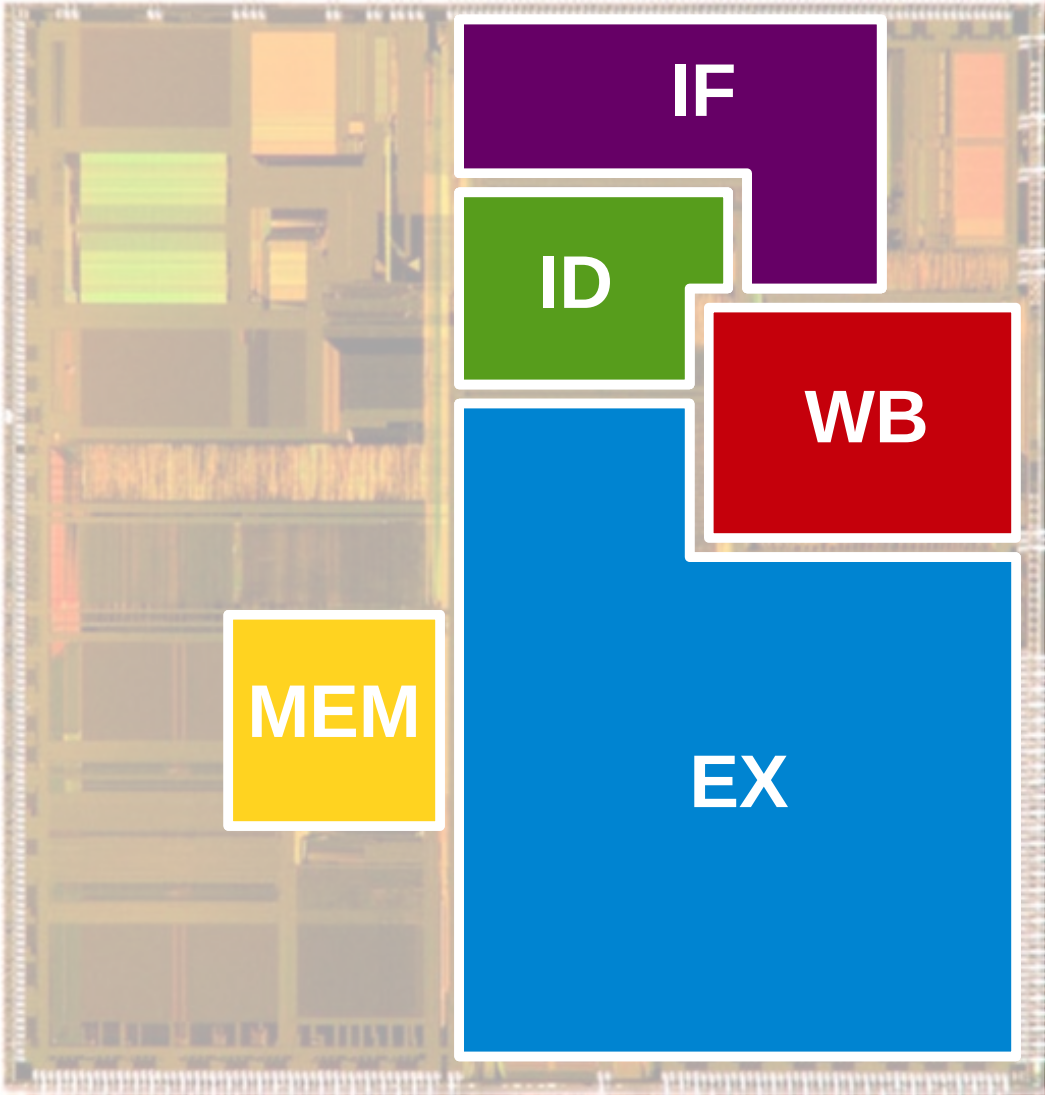
  5) Regiszterek frissítése (Write-back, **WB**)
     - Aritmetikai/logikai: eredményt tárol (`R1 ← R2*R3`)
     - Load: eredményt tárol (`R1 ← MEM[R0+42]`)
     - Store, ugrások: kihagyható

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

UTASÍTÁSOK FELDOLGOZÁSA

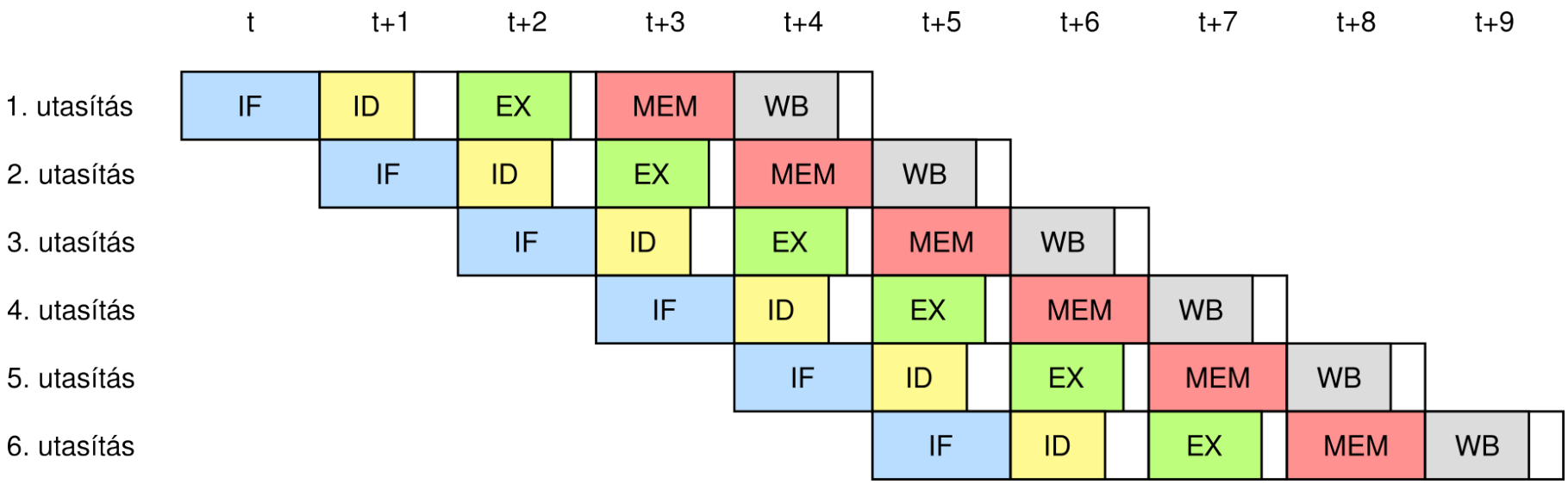


UTASÍTÁSOK FELDOLGOZÁSA



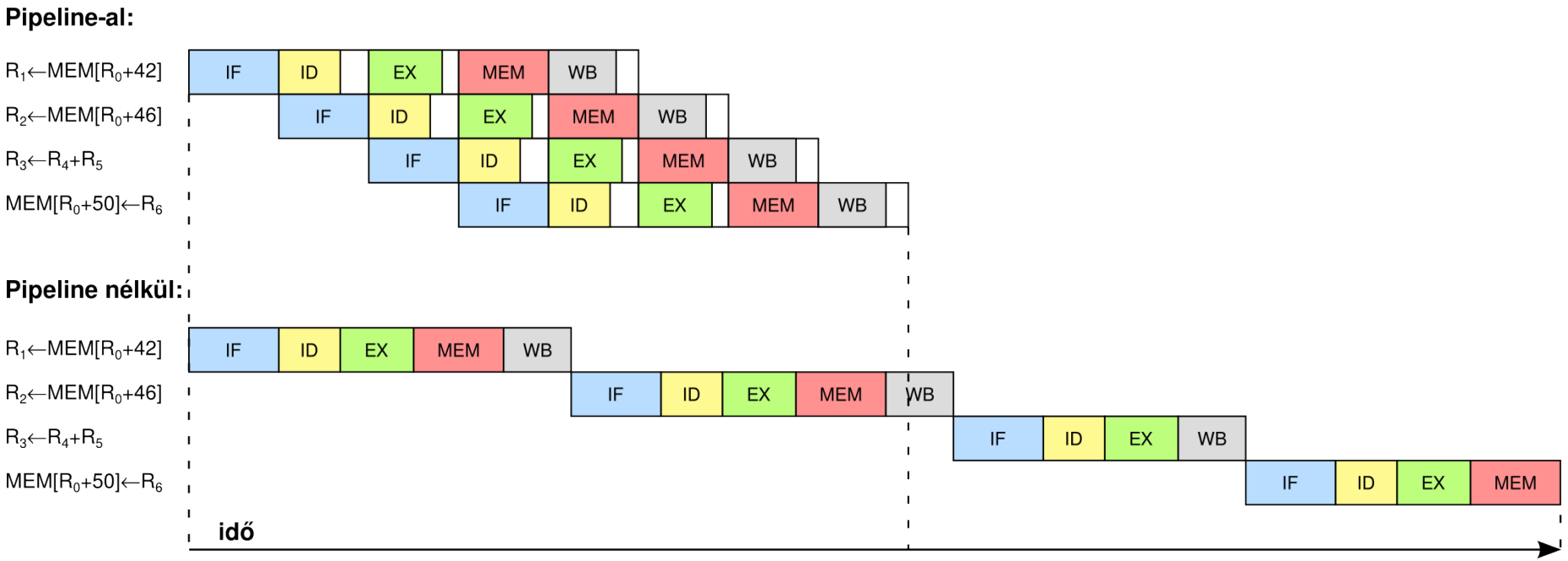
PIPELINE ELV

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



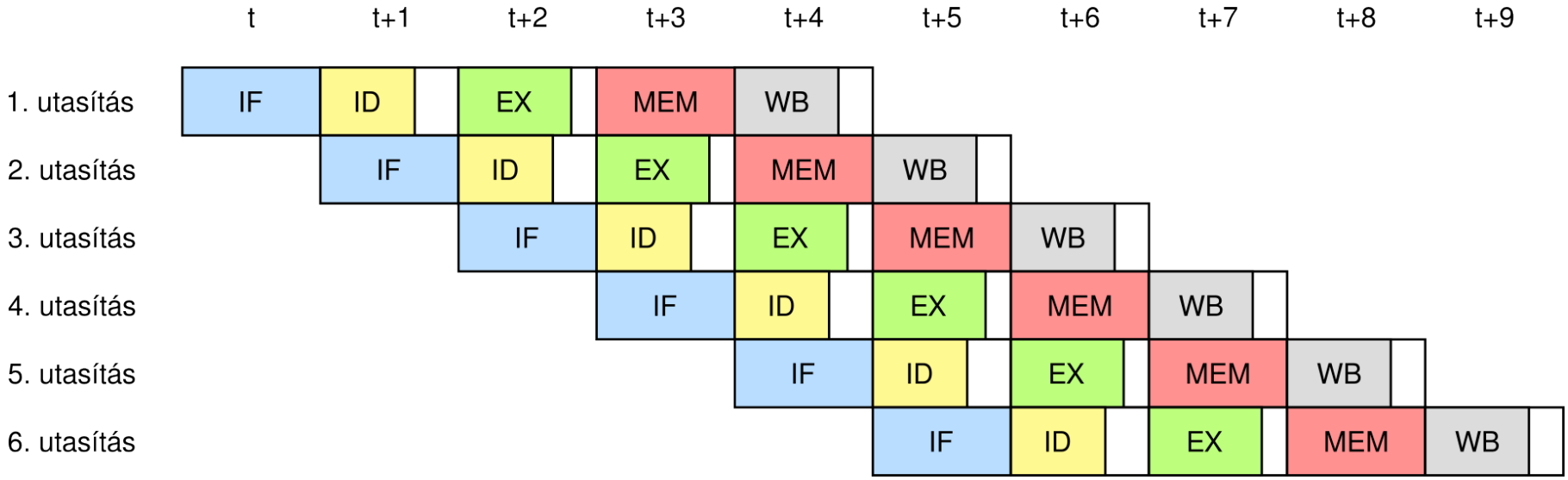
PIPELINE ELV

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



PIPELINE ELV

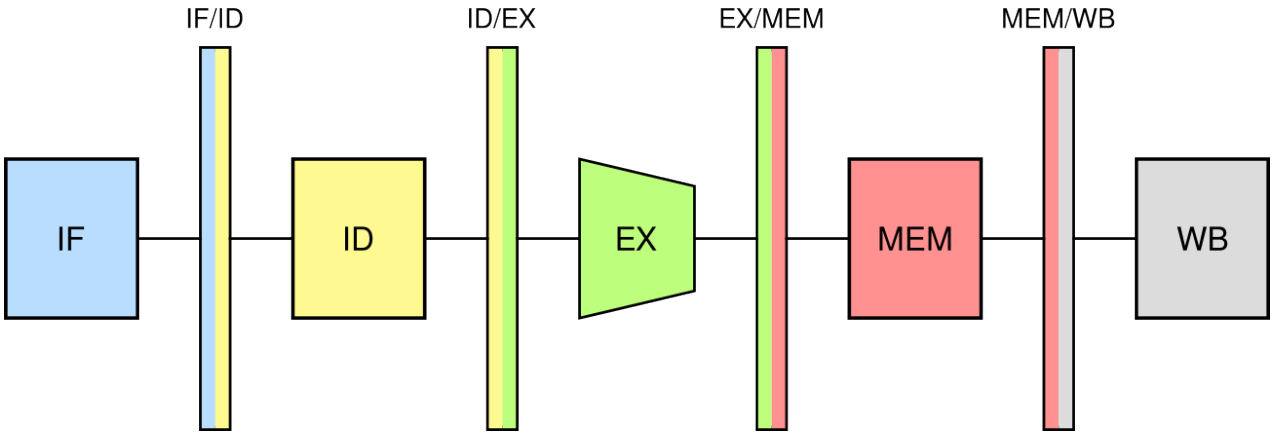
- Mennyiségek:
  - Mélység (deepness): fázisok száma
  - Késleltetés (latency): átfutási idő
  - Átviteli sebesség (throughput): kész utasítás / sec
  - Feltöltési / kiürülési idő



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

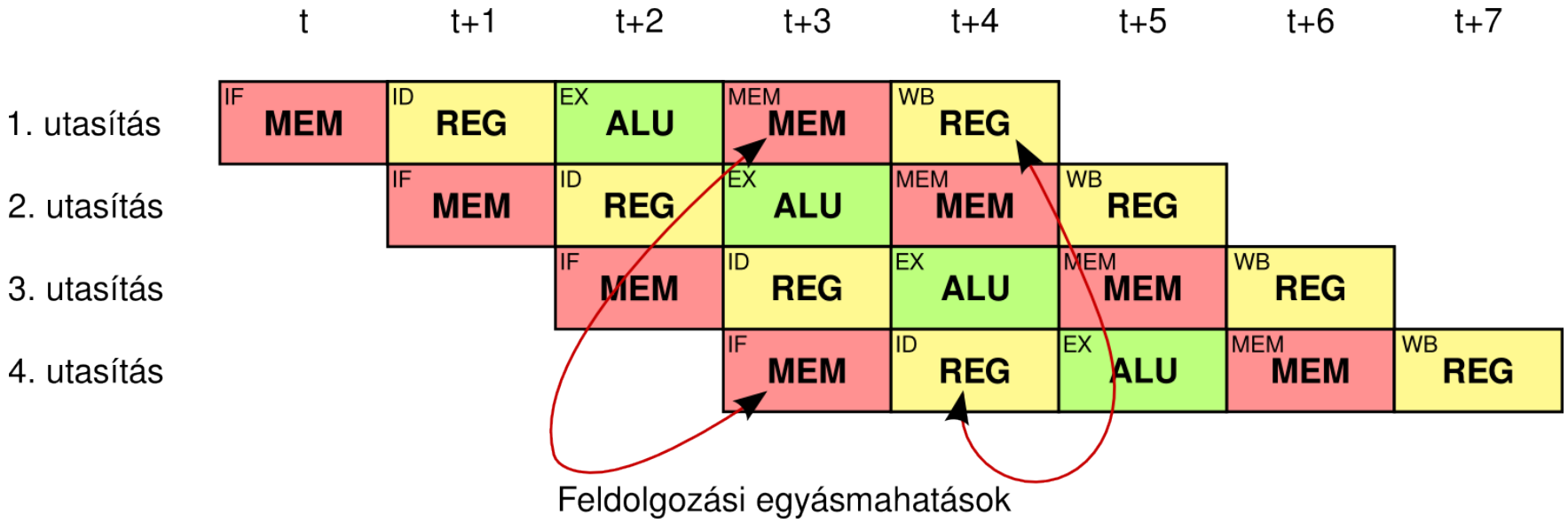
    → ***Procedurális egymásrahatás***

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

    → rontja a hatékonyságot

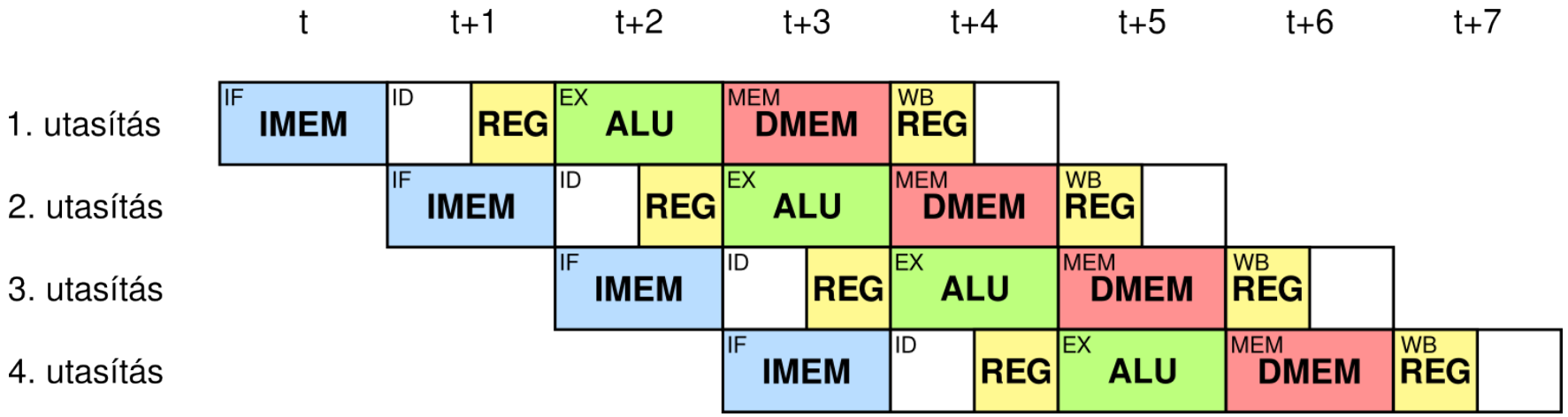
FELDOLGOZÁSI EGYMÁSRAHATÁS

- 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
- Adatfüggőség:

  Több utasítás ugyanazon a címen végez műveletet
  (ugyanaz a regiszter, ugyanaz a memóriacím)

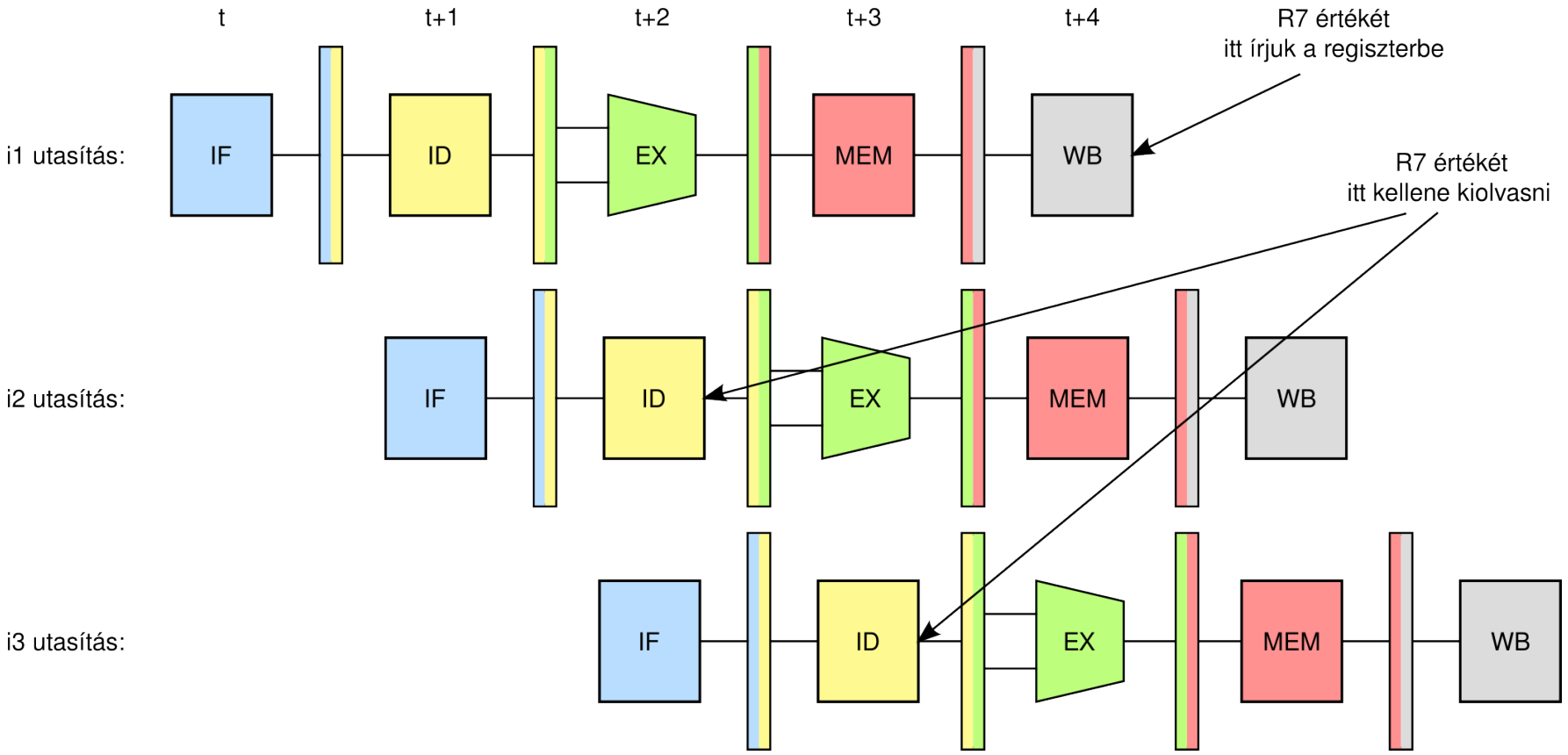
- Pl. az egyik utasítás operandusa az előző eredménye

  → ***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

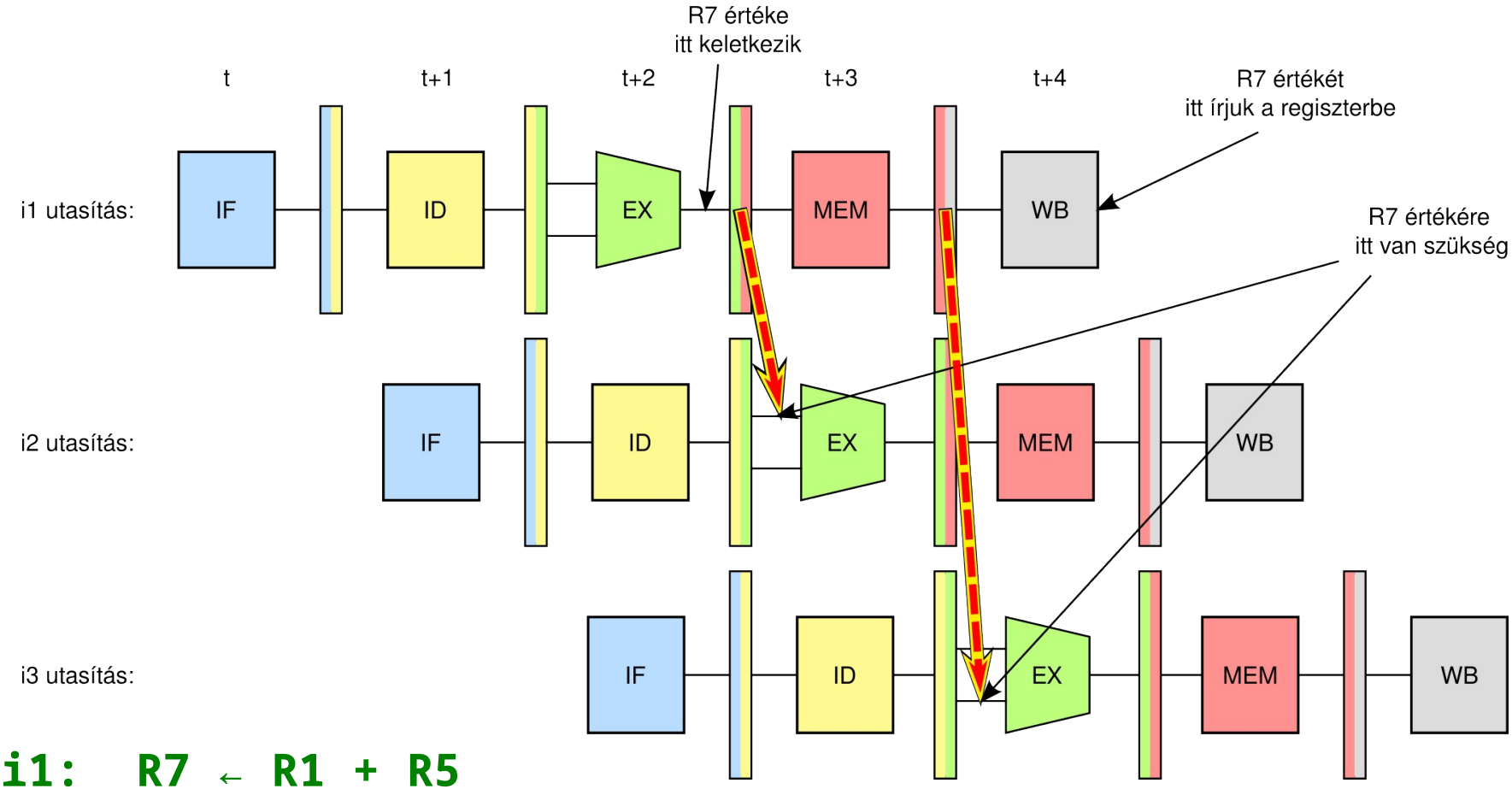
RAW EGYMÁSRAHATÁS




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

RAW EGYMÁSRAHATÁS

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



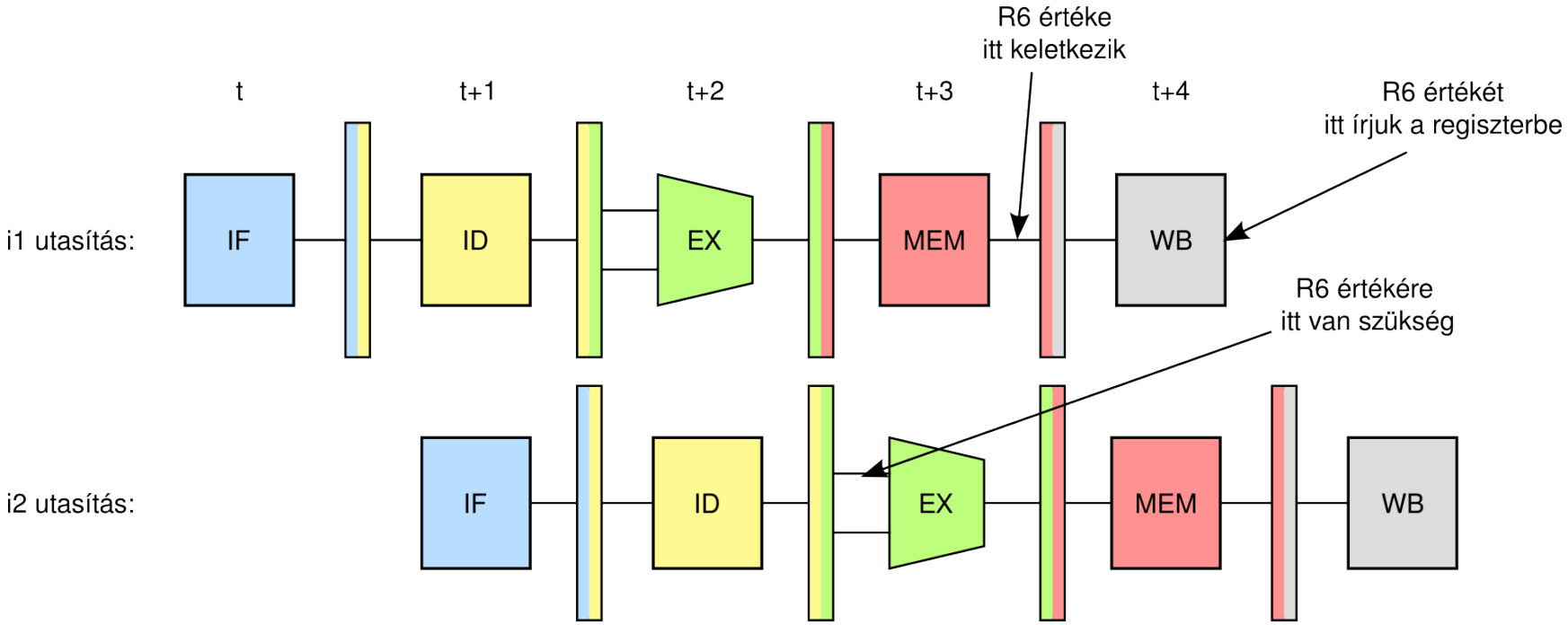

```
i1:   R7 ← R1 + R5
i2:   R8 ← R7 + R2
i3:   R5 ← R8 + R7
```

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

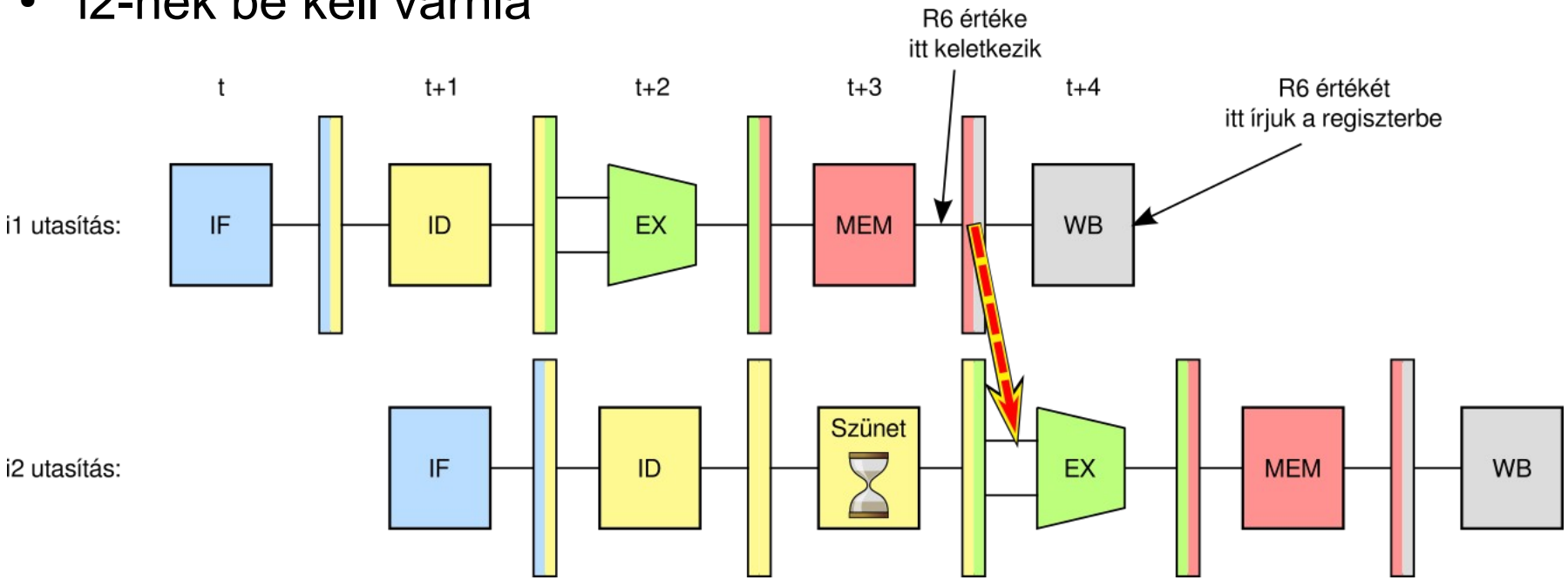


RAW EGYMÁSRAHATÁS

- 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

- Adatfüggőségek:
  - **RAW**: későbbi utasítás olvassa a korábban írt regisztert/címet
    - Megoldás: forwarding/várakozás
  - WAR: későbbi utasítás írja a korábban olvasott regisztert/címet
    - Ebben a pipeline-ban nem igényel kezelést
  - WAW: több utasítás ugyanazt a regisztert/címet írja
    - Ebben a pipeline-ban nem igényel kezelést
  - RAR: több utasítás ugyanazt a regisztert/címet olvassa
    - Sosem igényel kezelést

- Példa:

```
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

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***

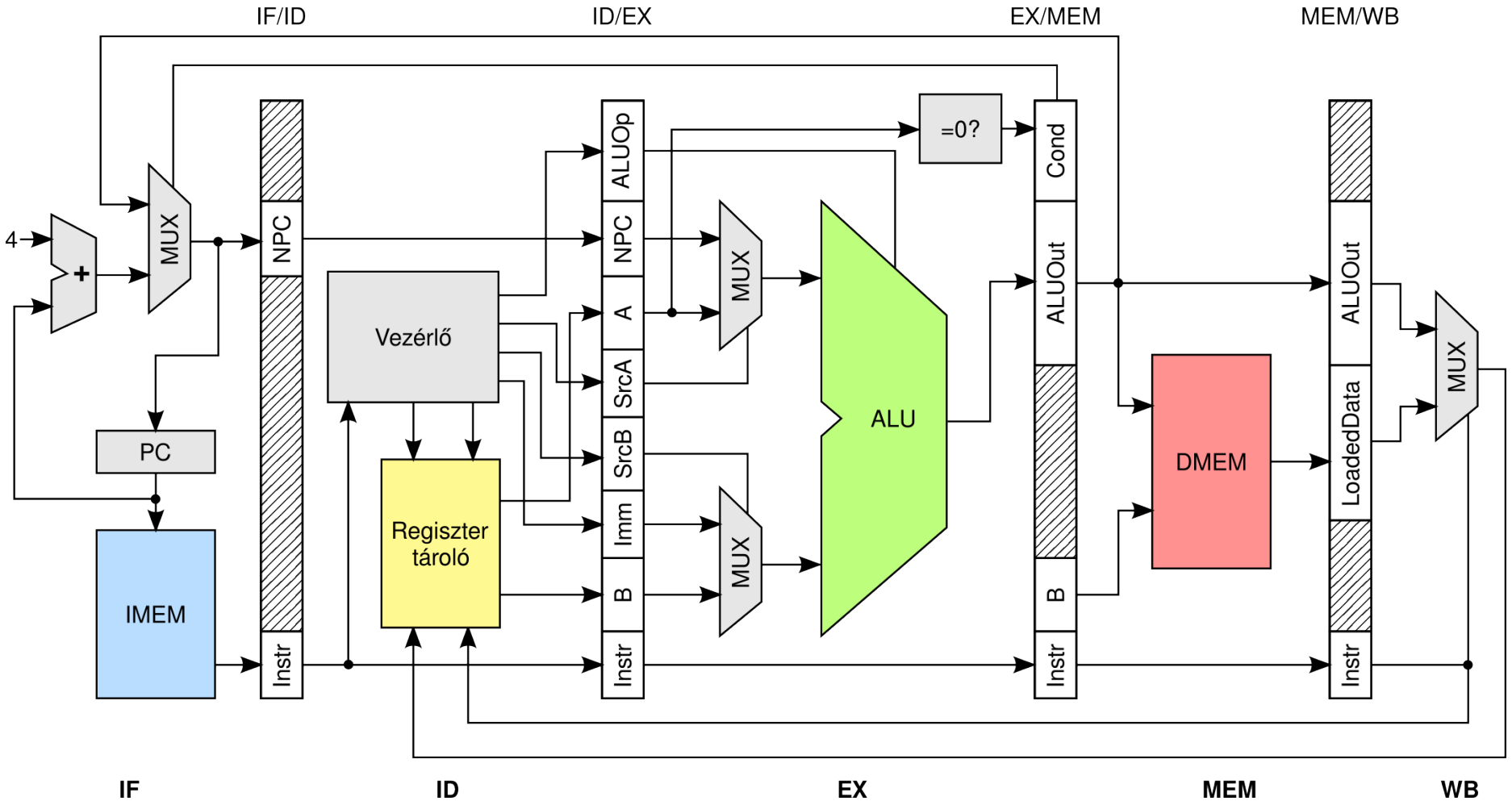
PROCEDURÁLIS EGYMÁSRAHATÁS

- **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



AZ ALU

- ALU: központi szerep
- Számol:

```
R1 ← R1 + 42
R1 ← R2 + R3
```

  - `PC ← PC + 42` (`JUMP +42`)

```
R1 ← MEM [R2 + 42]
```

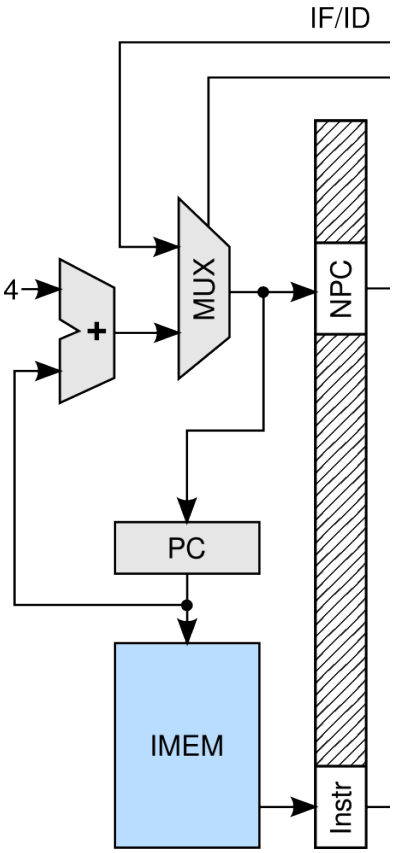
  - Első operandus: **regiszter** vagy **PC**
  - Második operandus: **regiszter** vagy **konstans** (immediate)
- Feltételt értékel ki:

```
JUMP -28 IF R1==0
```

- Az ID feladata:
  - Operandusok előkészítése
  - Operanduskiválasztó jel előállítása
  - Műveleti kód (+,-,*,/)

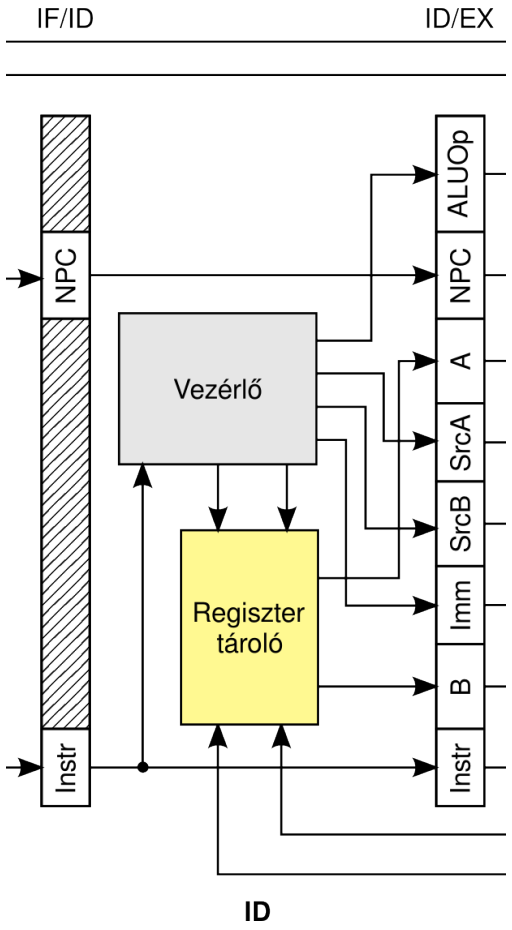
IF FÁZIS

- Utasításszámláló update:
  - Ha ugrás (**`EX/MEM.Instr.Opcode==branch`**), és ugrani kell (**`EX/MEM.Cond==TRUE`**):
    - **`PC ← EX.MEM.ALUOut`**
  - Egyébként:
    - **`PC ← PC+4`**
- Utasításszó és (új) programszámláló továbbadása:
  - **`IF/ID.NPC ← PC`**
  - **`IF/ID.Instr ← IMEM[PC]`**



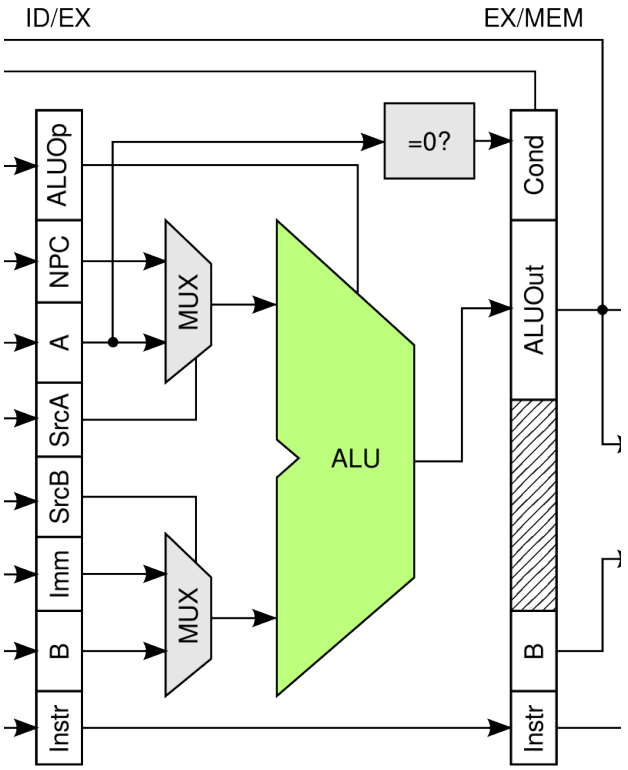
ID FÁZIS

- Operandusok előállítása:
  - Első operandus:
    - `ID/EX.NPC ← IF/ID.NPC`
    - `ID/EX.A ← Reg [IF/ID.Instr.ra]`
  - Első operandus kiválasztójel:
    - Ha ugró utasítás (`IF/ID.Instr.Opcode == branch`):
      - `ID/EX.SrcA ← npc`
    - Egyébként:
      - `ID/EX.SrcA ← regA`
  - Második operandus:
    - `ID/EX.Imm ← IF/ID.Instr.imm`
    - `ID/EX.B ← Reg [IF/ID.Instr.rb]`
  - Második operandus kiválasztójel:
    - Ha konstans (`IF/ID.Instr.HasImm`):
      - `ID/EX.SrcB ← imm`
    - Egyébként:
      - `ID/EX.SrcB ← regB`
- Műveleti kód előállítása:
  - Ha aritmetikai az utasítás (`IF/ID.Instr.Opcode == arithm`):
    - `ID/EX.ALUOp ← IF/ID.Instr.Func`
  - Egyébként (utasításszámláló növelés, címszámítás):
    - `ID/EX.ALUOp ← „+”`
- Utasításszó továbbadása:
  - `ID/EX.Instr ← IF/ID.Instr`



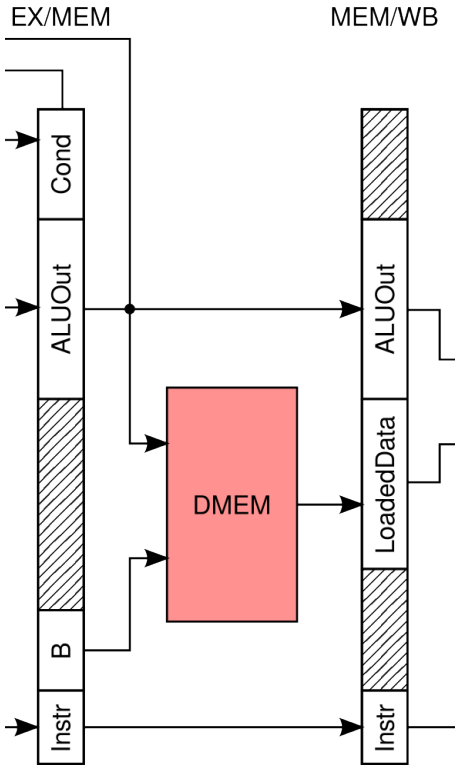
EX FÁZIS

- ALU bemenetei:
  - Ha `ID/EX.SrcA == npc`
    - `ALU.A ← ID/EX.NPC`
  - Ha `ID/EX.SrcA == regA`
    - `ALU.A ← ID/EX.A`
  - Ha `ID/EX.SrcB == imm`
    - `ALU.B ← ID/EX.Imm`
  - Ha `ID/EX.SrcB == regB`
    - `ALU.B ← ID/EX.B`
  - Művelettípus:
    - `ALU.Op ← ID/EX.ALUOp`
- Eredmény tárolása:
  - `EX/MEM.ALUOut ← ALU.Out`
- Összehasonlító egység:
  - `EX/MEM.Cond ← ID/EX.A == 0`
- Store utasítás esetén B továbbítása
  - `EX/MEM.B ← ID/EX.B` (Ez lesz majd a beírandó adat)
- Utasításszó továbbadása:
  - `EX/MEM.Instr ← ID/EX.Instr`



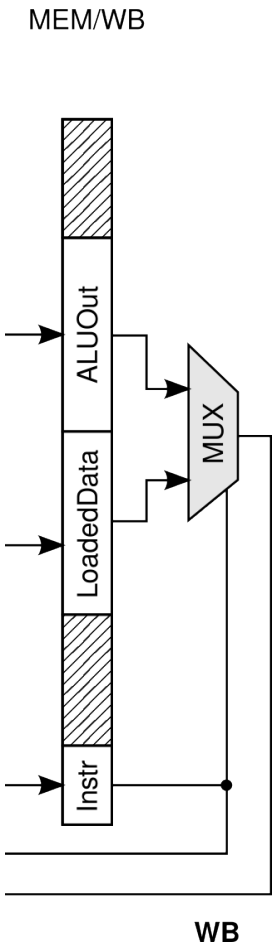
MEM FÁZIS

- Memóriacím:
  - **`ALUOut`**
- Store esetén (**`ID/EX.Instr.Opcode == Store`**)
  - **`MEM[EX/MEM.ALUOut] ← EX/MEM.B`**
    (Beírandó adat: B)
- Load esetén (**`ID/EX.Instr.Opcode == Load`**)
  - **`MEM/WB.LoadedData ← MEM[EX/MEM.ALUOut]`**
- Aritmetikai utasítás esetén:
  - **`MEM/WB.ALUOut ← EX/MEM.ALUOut`**
    (Eredmény továbbadása)
- Utasításszó továbbadása:
  - **`MEM/WB.Instr ← EX/MEM.Instr`**

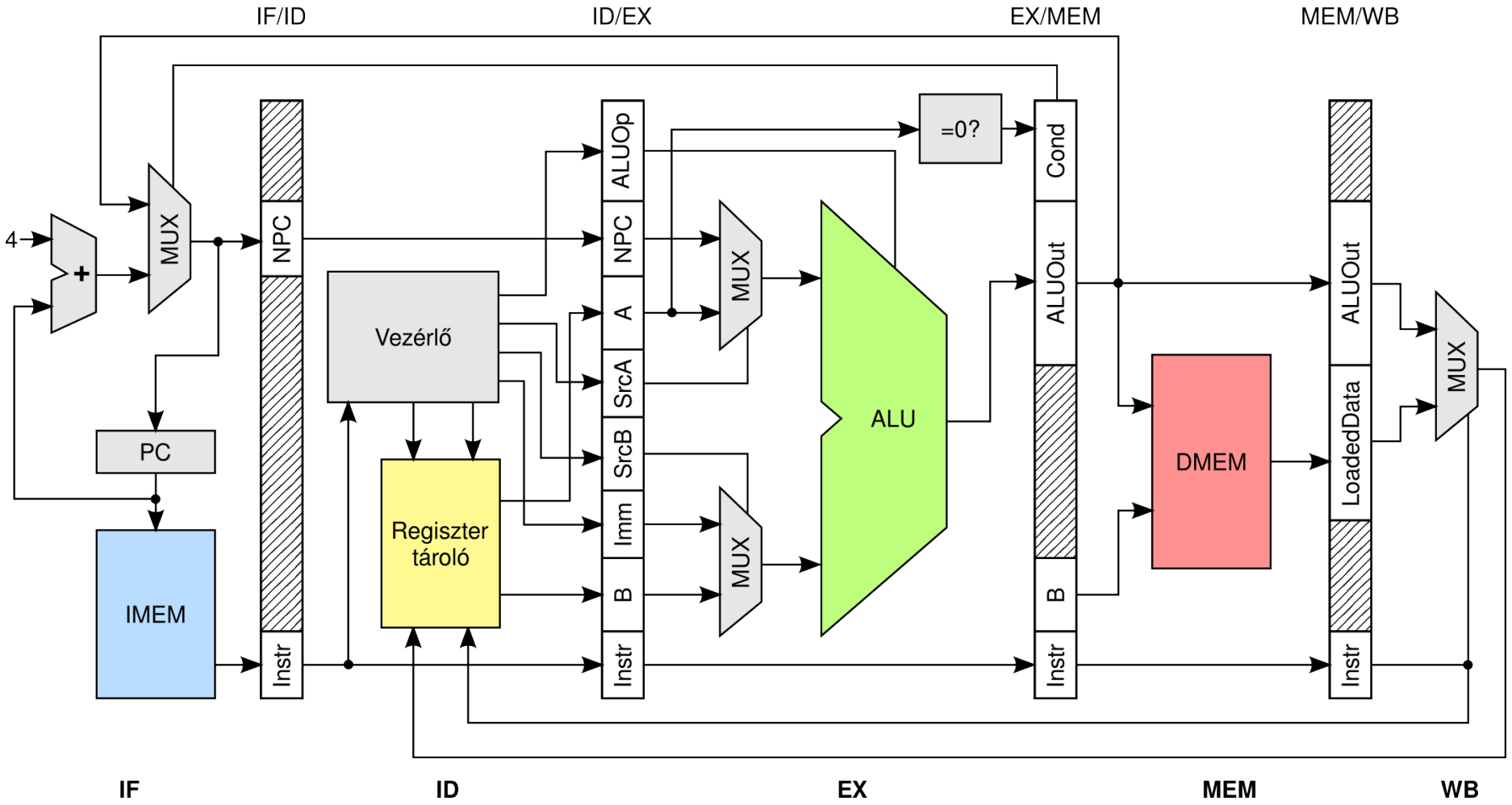


WB FÁZIS

- Regisztertároló (`Reg[]`) frissítése
  - Aritmetikai utasítás esetén
    (`MEM/WB.Instr.Opcode == arithm`)
    - `Reg[MEM/WB.Instr.rd] ← MEM/WB.ALUOut`
  - Load utasítás esetén
    (`MEM/WB.Instr.Opcode == Load`)
    - `Reg[MEM/WB.Instr.rd] ← MEM/WB.LoadedData`

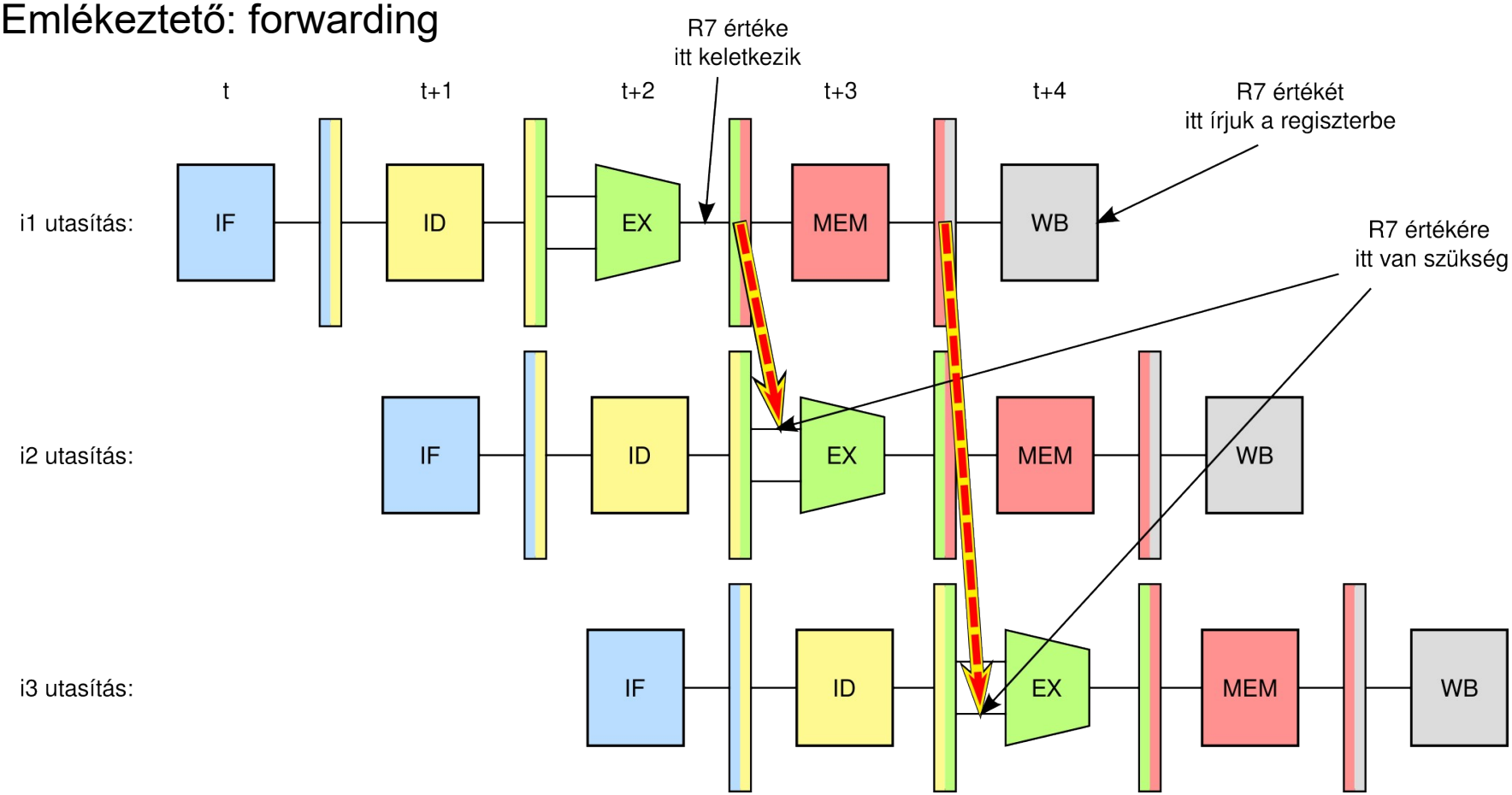


PIPELINE IMPLEMENTÁCIÓ



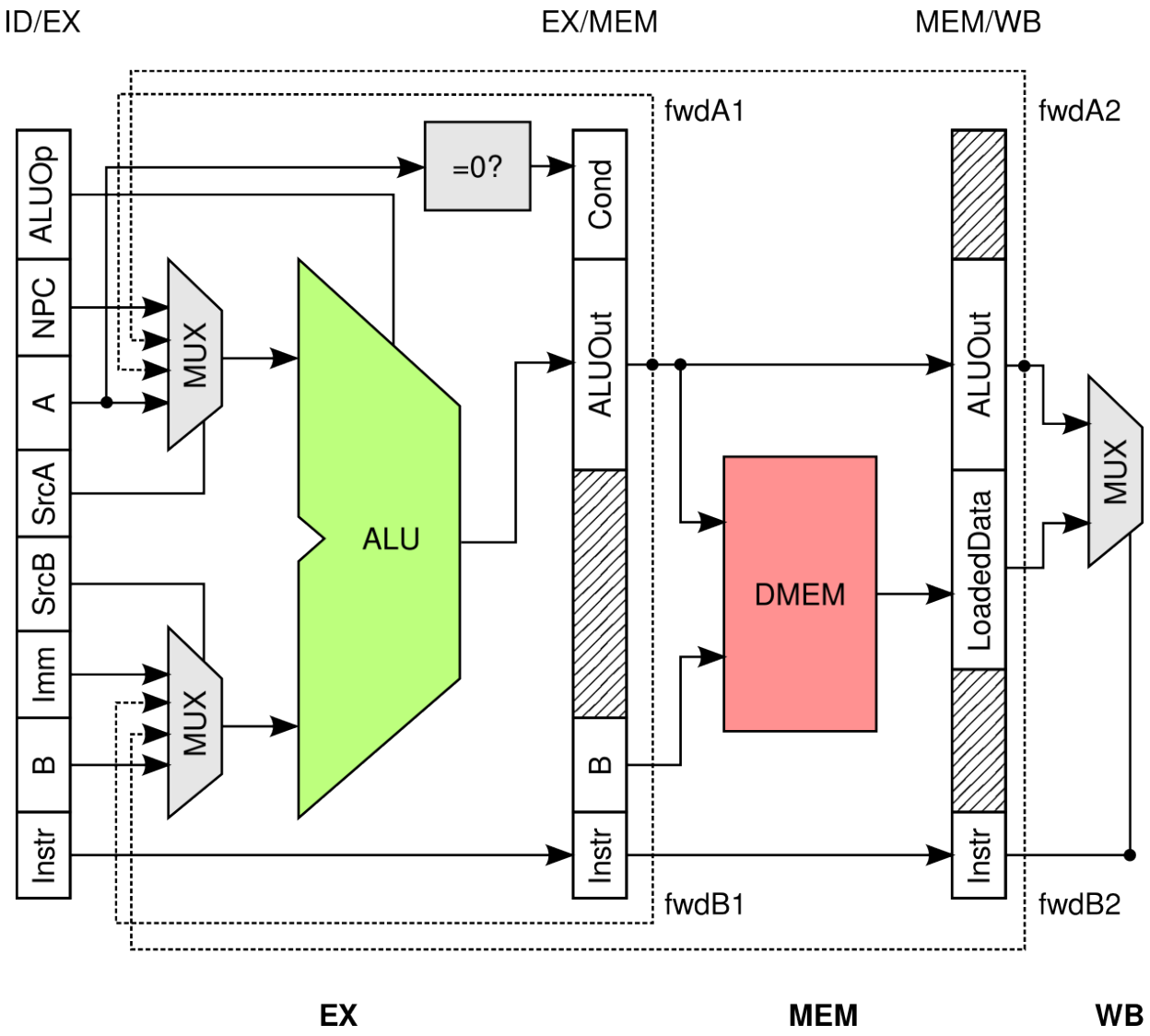
RAW EGYMÁSRAHATÁS

- 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?
  - Két operandus
  - Mindkettő kétféle lehet (NPC ↔ regiszter ill. immediate ↔ regiszter)
  - Regiszterek három helyről származhatnak:
    - ID-től
    - ALU kimenetről
    - Egyel korábbi ALU kimenetről
- Kiválasztó logika (ID-ben!):
  - pl. 1-es operandusra, **`ID/EX.SrcA`** lehet 0, 1, 2, 3:
    - **`npc: IF/ID.Instr.Opcode == branch`**
    - **`fwdA1:`** ha **`IF/ID.Instr.Opcode==arithm && IF/ID.Instr.ra == ID/EX.Instr.rd`**
    - **`fwdA2:`** ha **`IF/ID.Instr.Opcode==arithm && IF/ID.Instr.ra == EX/MEM.Instr.rd`**
    - **`regA:`** ha **`IF/ID.Instr.Opcode==arithm`**, egyébként
  - Hasonlóan a 2-es operandusra

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?
  - Kivétel!
    - Periféria kérés,
    - Laphiba,
    - Védelmi hiba,
    - Érvénytelen utasítás,
    - Stb.

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?

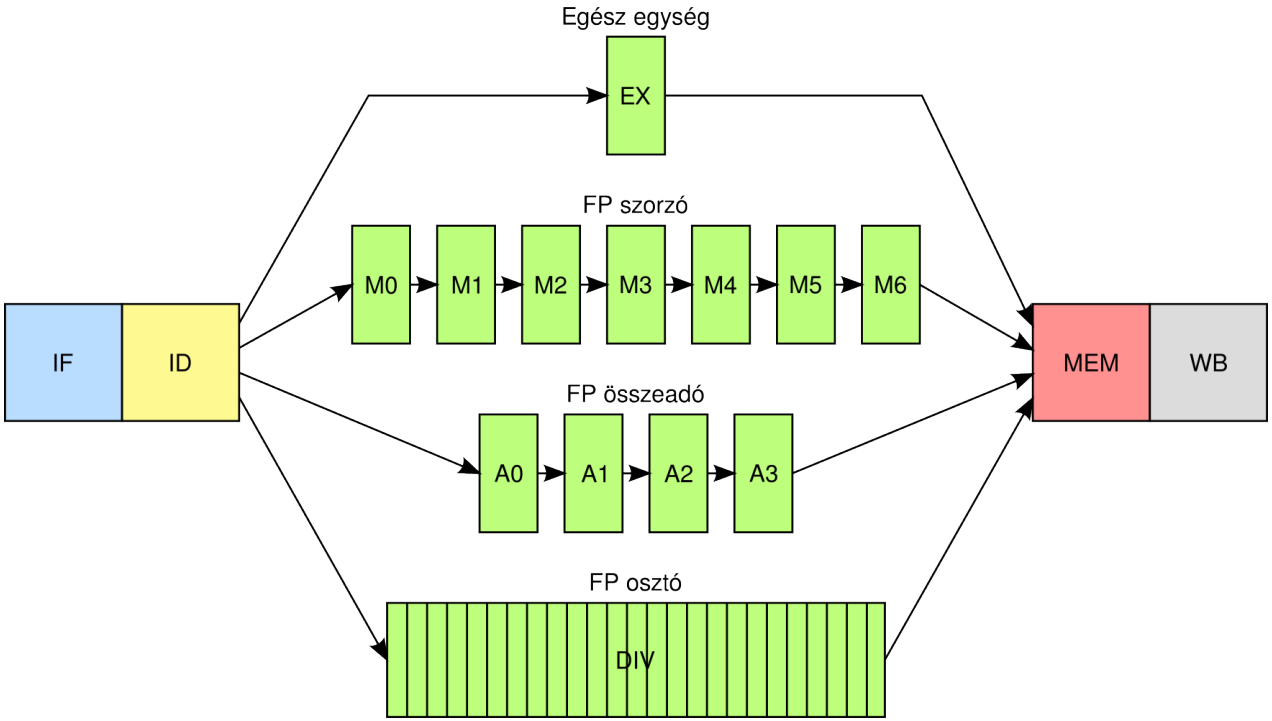
ELTÉRŐ IDEJŰ ARITMETIKAI MŰVELETEK

- Aritmetikai műveletek ***késleltetés***e:
  - 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:
  - Később 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

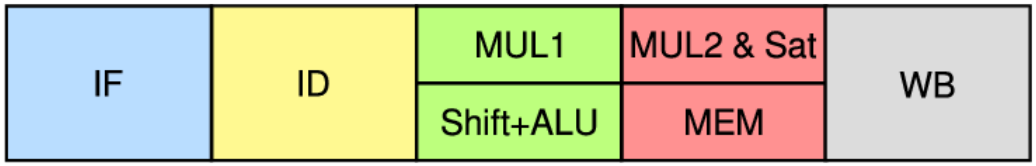
Utasítás	1	2	3	4	5	6	7	8
D2 ← D1 + D3	IF	ID	A0	A1	A2	A3	MEM	WB
...		IF	ID	EX	MEM	WB		
D2 ← MEM [R0+4]			IF	ID	EX	MEM	WB	

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

Alternatív pipeline struktúrák

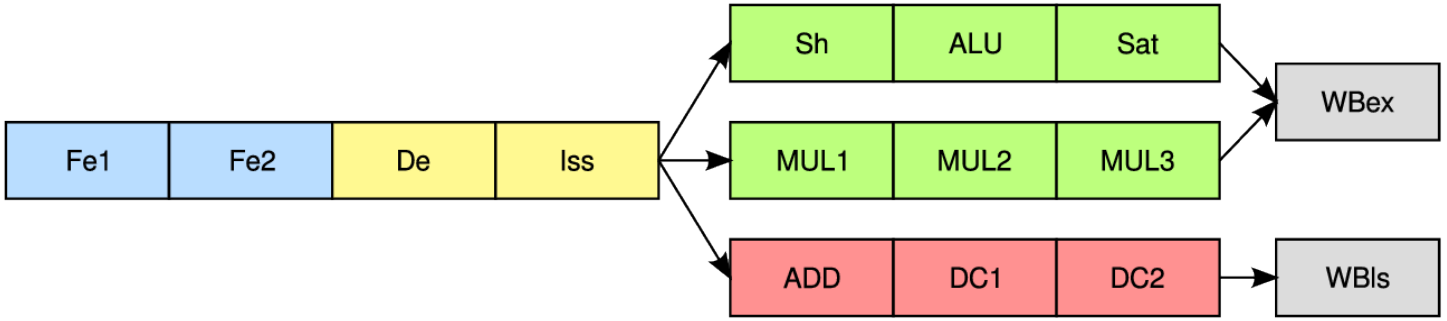
AZ ARM926EJ-S PROCESSZOR PIPELINE-JA

- 5 fokozatú, hasonlít a tanultakhoz
- Eltérés:
  - Szorzás két fázist igényel → második lépése: MEM
  - Van telítődő aritmetika → telítődés a MEM-ben



AZ ARM1176JZF-S PROCESSZOR PIPELINE-JA

- A Raspberry Pi processzora
- Mélység: 8
  - IF: 2 fázis (Fe1: lehívás, Fe2: elágazásbecslés)
  - ID: 2 fázis (De: dekódolás, Iss: regisztertároló kiolvasása)
- 3 műveleti egység, késleltetés: 3, iterációs idő: 1
  - ALU (Sh: shift, ALU: számolás, Sat: telítődés)
  - Szorzás (MUL1, MUL2, MUL3)
  - Memóriaműveletek (ADD: címszámolás, DC1, DC2: adatcache elérés)
  - Az ALU és a szorzó nem megy egyszerre, a többi igen



HÁLÓZATI RENDSZEREK
ÉS SZOLGÁLTATÁSOK
TANSZÉK

HÁLÓZATI RENDSZEREK ÉS SZOLGÁLTATÁSOK TANSZÉK

SZÁMÍTÓGÉP ARCHITEKTÚRÁK

Soron kívüli utasítás-végrehajtás

Horváth Gábor, Belső Zoltán

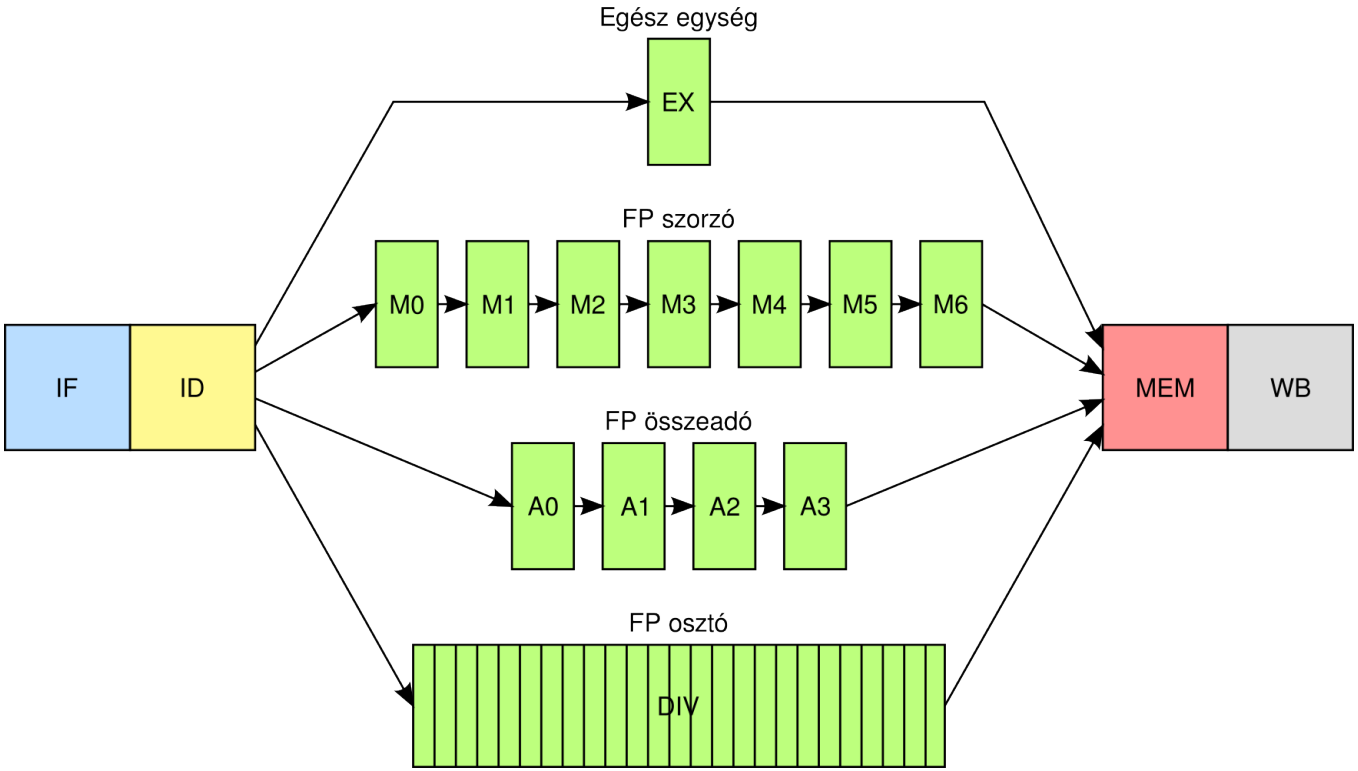
BME Hálózati Rendszerek és Szolgáltatások Tanszék
ghorvath@hit.bme.hu, belso@hit.bme.hu

Budapest,
2022.04.27.

MŰEGYETEM 1782

KÜLÖNBÖZŐ KÉSLELTETÉSŰ MŰVELETEK

- Láttuk, hogy a lebegőpontos műveletek késleltetése nagyobb



KÜLÖNBÖZŐ KÉSLELTETÉSŰ MŰVELETEK

C kód:

```
for (i=0; i<N; i++)
    Z[i]=A*X[i];
```

Elemi utasítások:

```
D2 ← MEM[R1]
D3 ← D2 * D0
MEM[R2] ← D3
R1 ← R1 + 8
R2 ← R2 + 8
```

A: D0
X[i]: MEM[R1]
Z[i]: MEM[R2]

Utasítások ütemezése: (szorzás: 5 órajel, egész és memóriaművelet: 1 órajel)

Utasítások:	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.
D2 ← MEM[R1]	IF	ID	EX	MEM	WB									
D3 ← D2 * D0		IF	ID	**A***	M0	M1	M2	M3	M4	MEM	WB			
MEM[R2] ← D3			IF	**F***	ID	**A***	**A***	**A***	**A***	EX	MEM	WB		
R1 ← R1 + 8					IF	**F***	**F***	**F***	**F***	ID	EX	MEM	WB	
R2 ← R2 + 8										IF	ID	EX	MEM	WB

OPTIMALIZÁLÁS

- Optimalizáljuk a programot a sorok átrendezésével!

Eredeti utasítássorozat:

```
D2 ← MEM[R1]
D3 ← D2 * D0
MEM[R2] ← D3
R1 ← R1 + 8
R2 ← R2 + 8
```

Optimalizált utasítássorozat:

```
D2 ← MEM[R1]
D3 ← D2 * D0
R1 ← R1 + 8
MEM[R2] ← D3
R2 ← R2 + 8
```

Utasítások:	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.
D2 ← MEM[R1]	IF	ID	EX	MEM	WB									
D3 ← D2 * D0		IF	ID	A*	M0	M1	M2	M3	M4	MEM	WB			
R1 ← R1 + 8			IF	F*	ID	EX	MEM	WB						
MEM[R2] ← D3					IF	ID	A*	A*	A*	EX	MEM	WB		
R2 ← R2 + 8						IF	F*	F*	F*	ID	EX	MEM	WB	

OPTIMALIZÁLÁS

- Utasítások átrendezésével gyorsabb lett a program
- Problémák:
  - Kihasználása a programozón / fordítón múlik
  - Minden pipeline struktúrára máshogy kell
- Kívánatos megoldás:
  - Csinálja a CPU, röptében!
  - Rendezze át az utasításokat kénye-kedve szerint
    - Úgy, hogy gyorsabb legyen
    - Úgy, hogy a szemantika ne változzon

  → **Soron kívüli (out-of-order) utasítás-végrehajtás**
- Megvalósítás:
  - Nem is túl bonyolult
  - Régóta alkalmazott elv
    - Első: Scoreboard – 1964
    - Fejlettebb: **Tomasulo** – 1967
      - Mind a mai napig használják (pl. Intel Core i7)

Soron kívüli utasítás-végrehajtás

A KEZDETEK...

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

CDC-6600

Freon alapú hűtés
minden szekrényben:

Duál vektorgrafikus megjelenítő:

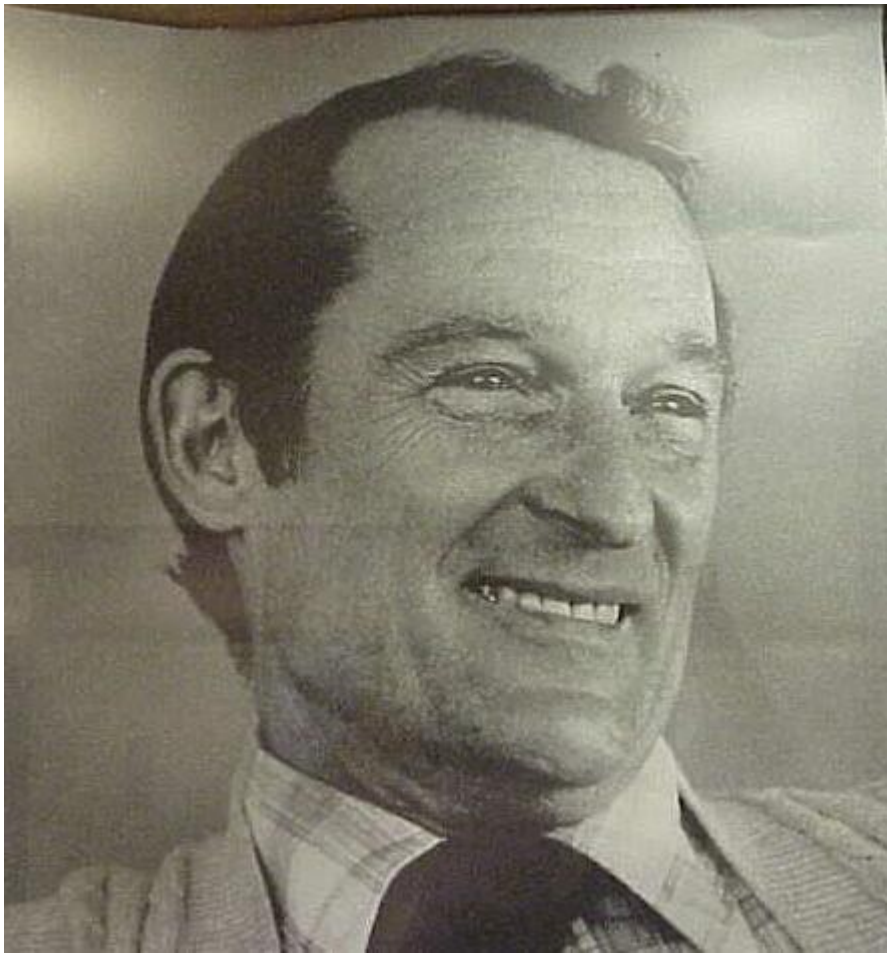
CRAY VS. IBM

- **Thomas Watson (IBM):**

  „Múlt héten megjelent a CDC 6600. Annál a cégnél összesen 34-en dolgoznak a portással együtt. Közülük csak 14 mérnök és csak 4 programozó. Nem értem, hogy a mi hatalmas fejlesztési ráfordításaink 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”



SORONKÍVÜLI VÉGREHAJTÁS HOZZÁVALÓI

- Több utasítást is le kell hívnunk, melyek közül válogatunk
  - Ezeket tárolni kell valahol:

    → **utasítástároló**
- Egy eljárás, ami eldönti az utasítások végrehajtási sorrendjét

  → **dinamikus ütemező**
- Egy jó ötlet, amivel az utasítások közötti függőségek csökkenthetők

  → **regiszterátnevezés**
- Egy trükk, hogy a külvilág sorrendi végrehajtást lásson

  → **sorrend-visszaállító buffer**

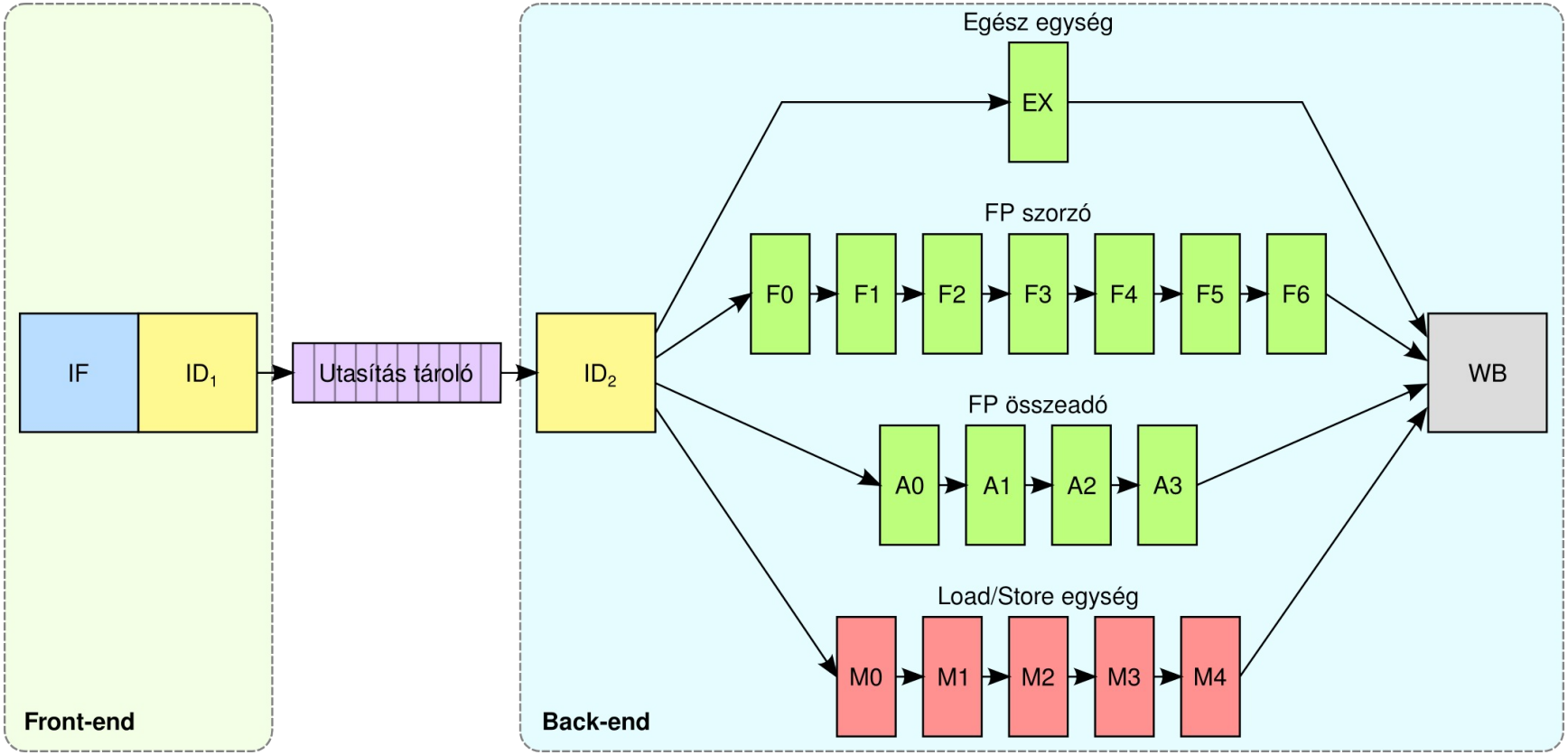
SORONKÍVÜLI VÉGREHAJTÁS HOZZÁVALÓI

- Több utasítást is le kell hívnunk, melyek közül válogatunk
  - Ezeket tárolni kell valahol:
    → **utasítástároló**
- Egy eljárás, ami eldönti az utasítások végrehajtási sorrendjét
  → **dinamikus ütemező**
- Egy jó ötlet, amivel az utasítások közötti függőségek csökkenthetők
  → **regiszterátnevezés**
- Egy trükk, hogy a külvilág sorrendi végrehajtást lásson
  → **sorrend-visszaállító buffer**

UTASÍTÁSTÁROLÓ

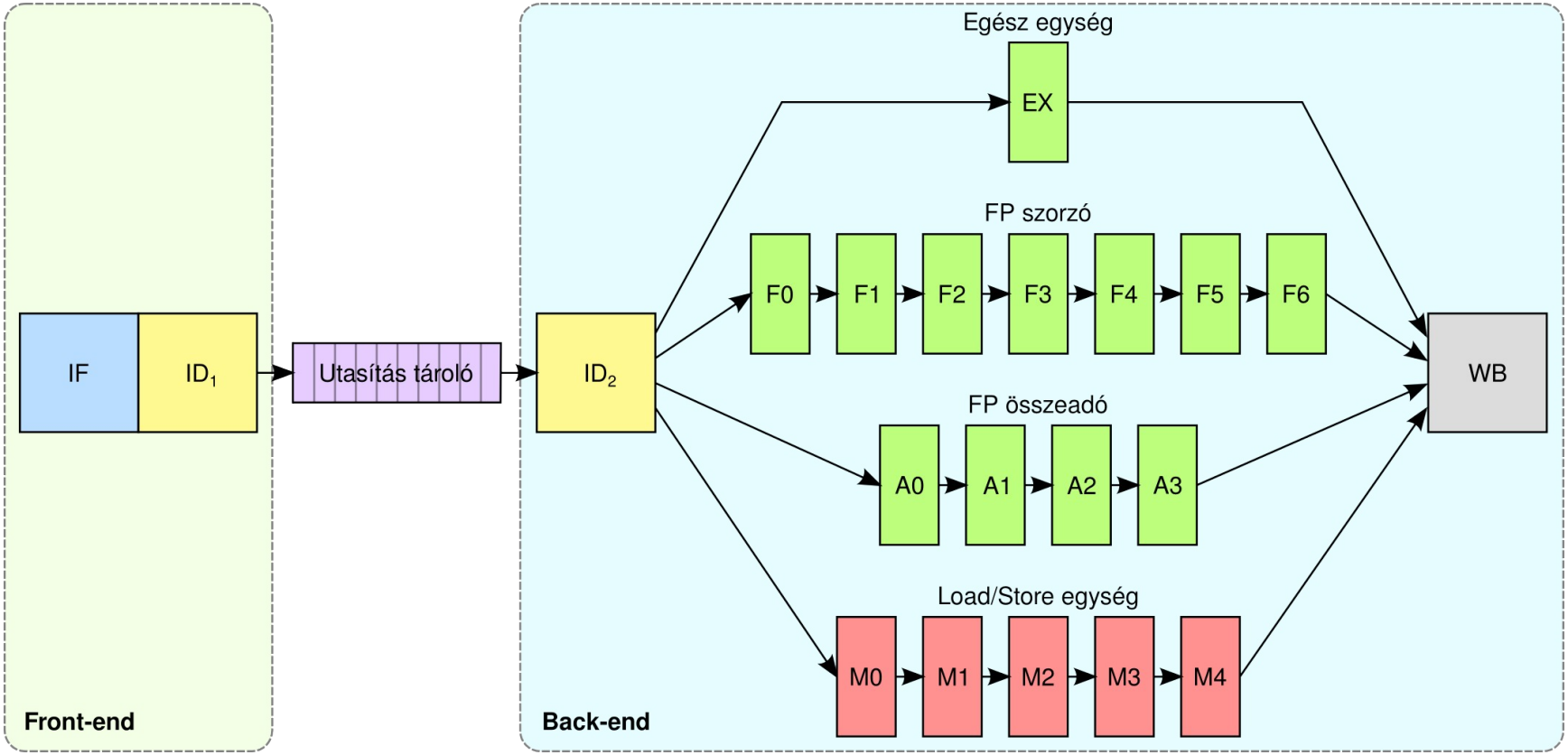
- Utasítások lehívása (IF) → sorrendben
- Utasítások végrehajtása (EX) → soron kívül

  → az utasítástároló csak az ID fázisban lehet!



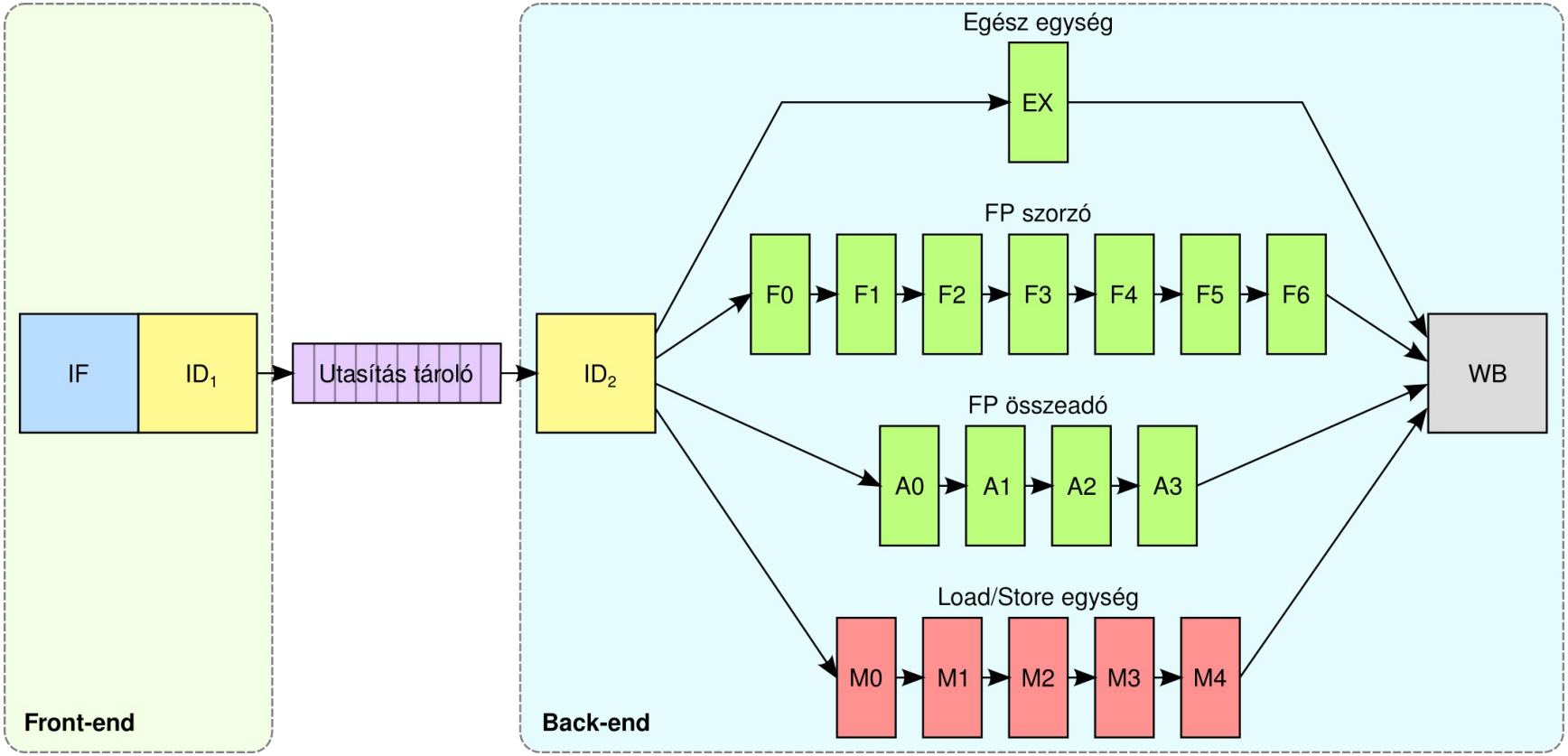
UTASÍTÁSTÁROLÓ

- Az utasítástároló két részre vágja a futószalagot:
  - Front-end: sorrendi belépés
  - Back-end: soron kívül belépés



UTASÍTÁSTÁROLÓ

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



UTASÍTÁSTÁROLÓ

- Alternatív nevek: Reservation Station, Instruction Window, stb.
- Felépítés lehet:
  - *elosztott* (műveleti egységenként külön)
  - *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

SORONKÍVÜLI VÉGREHAJTÁS HOZZÁVALÓI

- Több utasítást is le kell hívnunk, melyek közül válogatunk.
  - Ezeket tárolni kell valahol:
    → **utasítástároló**
- Egy eljárás, ami eldönti az utasítások végrehajtási sorrendjét
  → **dinamikus ütemező**
- Egy jó ötlet, amivel az utasítások közötti függőségek csökkenthetők
  → **regiszterátnevezés**
- Egy trükk, hogy a külvilág sorrendi végrehajtást lásson
  → **sorrend-visszaállító buffer**

SORONKÍVÜLI VÉGREHAJTÁS HOZZÁVALÓI

- Több utasítást is le kell hívnunk, melyek közül válogatunk.
  - Ezeket tárolni kell valahol:
    → **utasítástároló**
- Egy eljárás, ami eldönti az utasítások végrehajtási sorrendjét
    → **dinamikus ütemező**
- Egy jó ötlet, amivel az utasítások közötti függőségek csökkenthetők
    → **regiszterátnevezés**
- Egy trükk, hogy a külvilág sorrendi végrehajtást lásson
    → **sorrend-visszaállító buffer**

DINAMIKUS UTASÍTÁS-ÜTEMEZÉS

- Utasítások végrehajtási sorrendje:

  Adatáramlásos modell szerint!

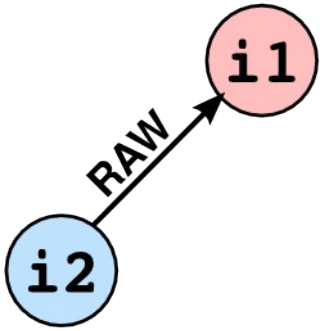
  - Az utasítássorozatból egy precedenciagráfot építünk
    - Csomópontok: utasítások
    - Élek: mely másik utasítást kell bevárni a végrehajtás előtt
  - Egy utasítás *végrehajtható,*

    ha minden utasítás, amitől függ, lefutott.
  - Ha egy utasítás végrehajtható, és van szabad egység, végrehajtjuk

PÉLDA

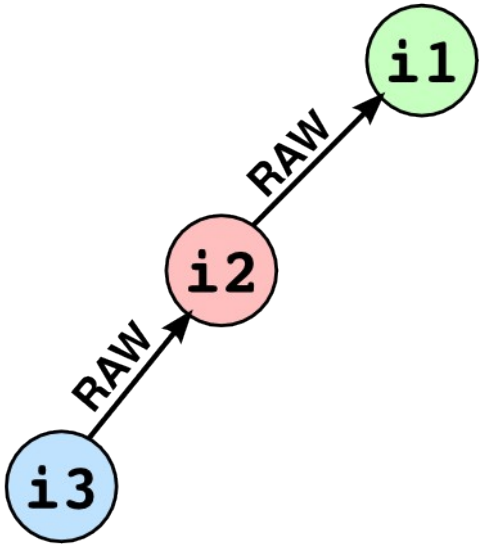

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
i5:  R2 ← R2 + 8

i6:  D2 ← MEM[R1]
i7:  D3 ← D2 * D0
i8:  MEM[R2] ← D3
i9:  R1 ← R1 + 8
i10: R2 ← R2 + 8
```

PÉLDA



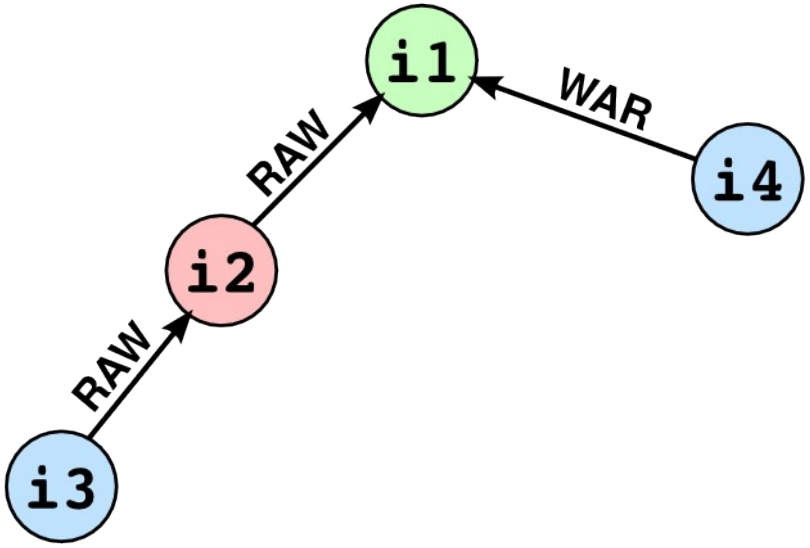

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
i5:  R2 ← R2 + 8

i6:  D2 ← MEM[R1]
i7:  D3 ← D2 * D0
i8:  MEM[R2] ← D3
i9:  R1 ← R1 + 8
i10: R2 ← R2 + 8
```

PÉLDA



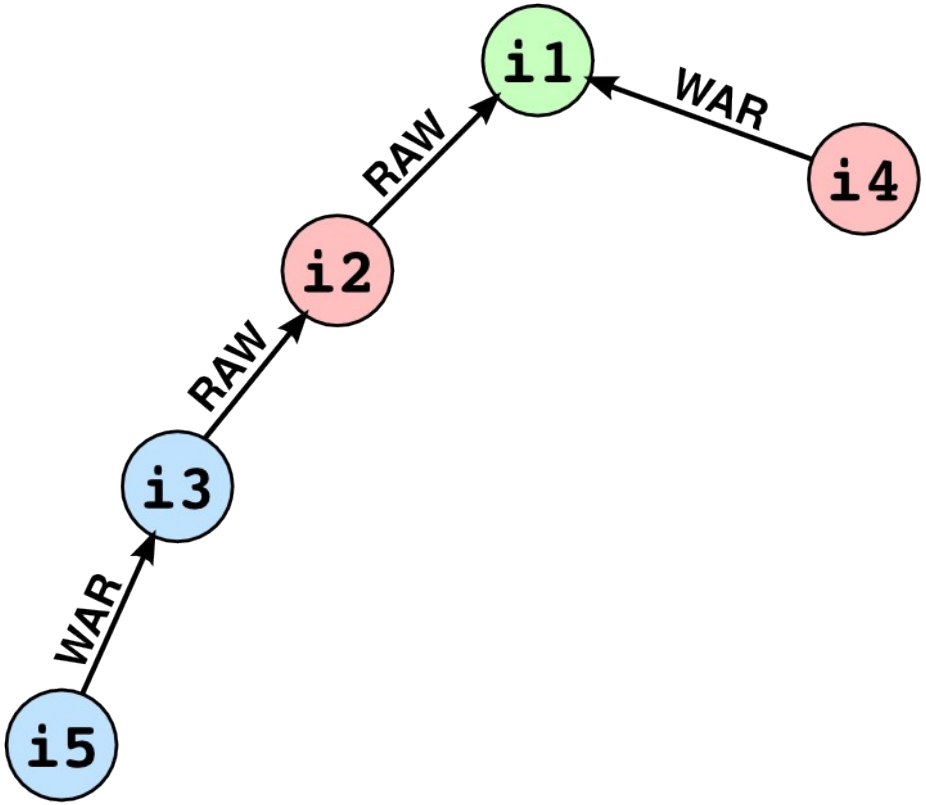

```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

PÉLDA



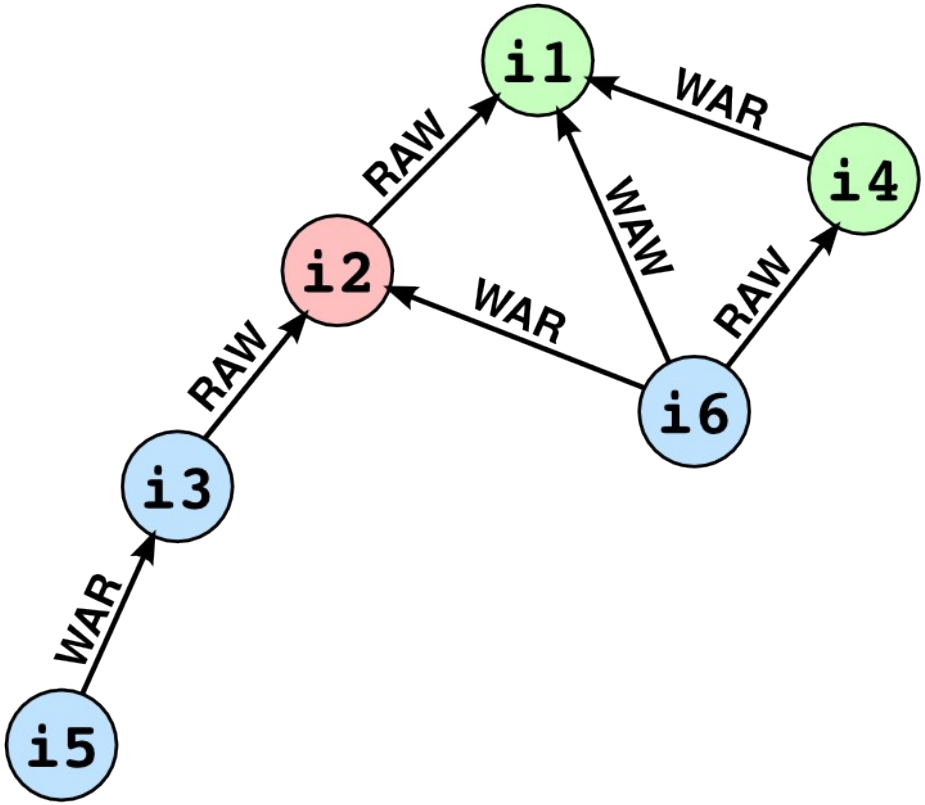

```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

PÉLDA



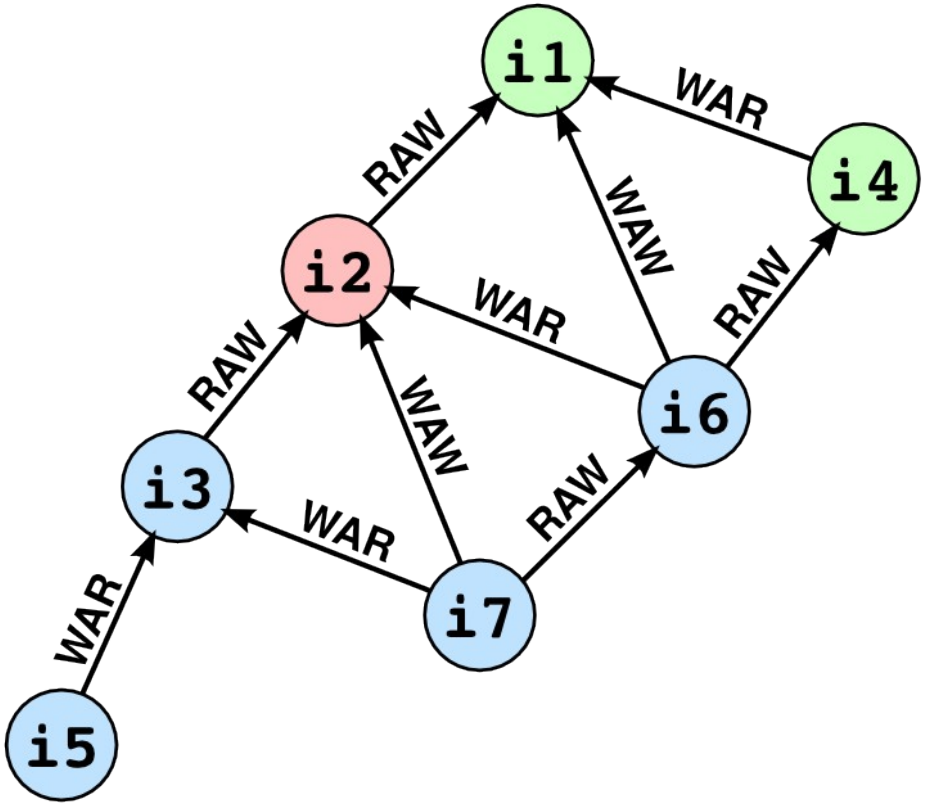

```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

PÉLDA



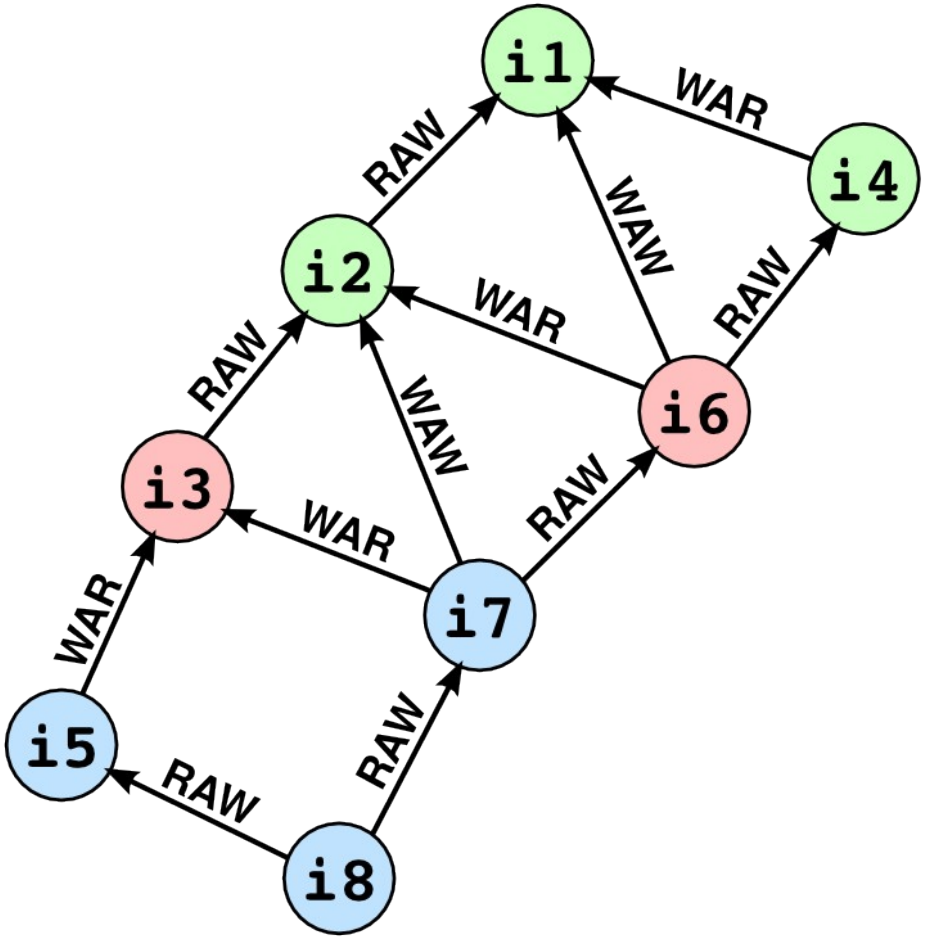

```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

PÉLDA



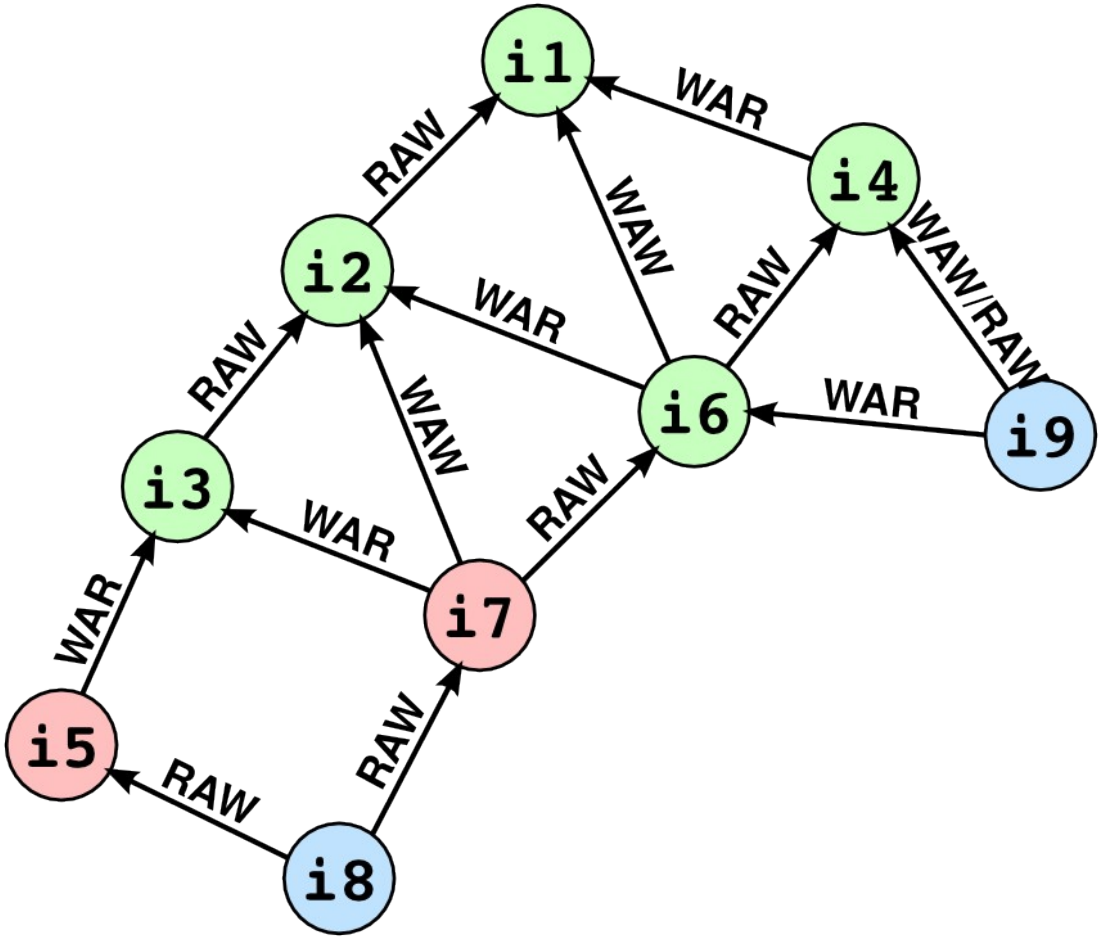

```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

PÉLDA




```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

PÉLDA




```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

SORONKÍVÜLI VÉGREHAJTÁS HOZZÁVALÓI

- Több utasítást is le kell hívnunk, melyek közül válogatunk
  - Ezeket tárolni kell valahol:
    → **utasítástároló**
- Egy eljárás, ami eldönti az utasítások végrehajtási sorrendjét
  → **dinamikus ütemező**
- Egy jó ötlet, amivel az utasítások közötti függőségek csökkenthetők
  → **regiszterátnevezés**
- Egy trükk, hogy a külvilág sorrendi végrehajtást lásson
  → **sorrend-visszaállító buffer**

SORONKÍVÜLI VÉGREHAJTÁS HOZZÁVALÓI

- Több utasítást is le kell hívnunk, melyek közül válogatunk
  - Ezeket tárolni kell valahol:

    → **utasítástároló**
- Egy eljárás, ami eldönti az utasítások végrehajtási sorrendjét

  → **dinamikus ütemező**
- Egy jó ötlet, amivel az utasítások közötti függőségek csökkenthetők

  → **regiszterátnevezés**
- Egy trükk, hogy a külvilág sorrendi végrehajtást lásson

  → **sorrend-visszaállító buffer**

REGISZTERÁTNEVEZÉS

- Cél: a függőségi gráf „megritkítása”
- Milyen függőségeket ismerünk:
  - RAW:

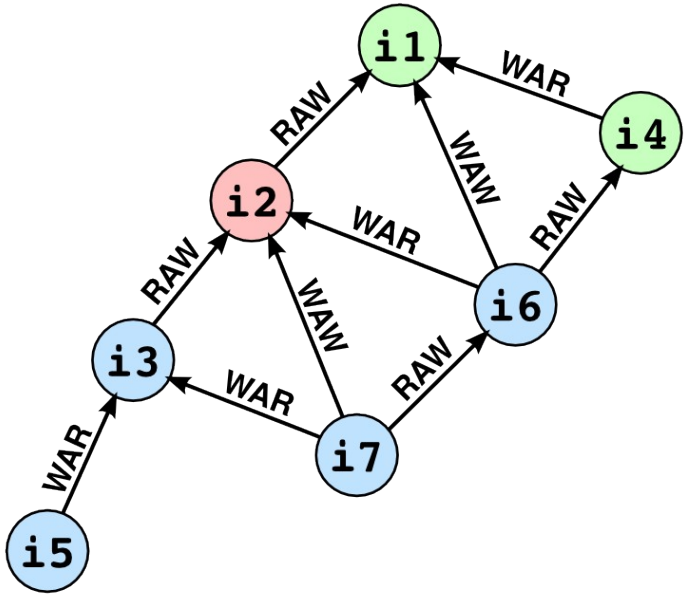
```
D3 ← D2 * D0
MEM[R2] ← D3
```

    → Valódi függőség, nem feloldható
  - WAR:

```
D3 ← D2 * D0
...
D2 ← MEM[R1]
```

  - WAW:

```
D3 ← D2 * D0
...
D3 ← MEM[R1]
```



- Meglepetés: a WAR és WAW függőségek feloldhatók!
  → Álfüggőségek

REGISZTERÁTNEVEZÉS

- Miért ír a programozó ilyen kódot?
  - WAR:

```
D3 ← D2 * D0
...
D2 ← MEM[R1]
```

  - WAW:

```
D3 ← D2 * D0
...
D3 ← MEM[R1]
```

- Újra felhasználja a regisztereket, mert nincs elég!
- Tegyünk a processzorba nagyon sok regisztert
  → **Fizikai regiszterek**
- De a programozó ezt nem látja
  → **Logikai/architekturális regiszterek**kel dolgozik
- A CPU a beolvasott programot röptében átírja
  → **Regiszterátnevezés**

REGISZTERÁTNEVEZÉS

- Eszköz: **Regiszter leképző tábla**
  - i. bejegyzése: az i. logikai regiszter melyik fizikaihoz van rendelve
- Eljárás:
  - Az utasítás operandusait fizikai regiszterekre cseréli
  - Az eredményt mindig egy új, nem használt fizikai regiszterbe teszi
  - Frissíti a regiszter leképző táblát

Eredeti:

```
i1:  D2 ← MEM[R1]
```

Átnevezés után:

```
i1:  U25 ← MEM[T3]
```

Regiszter leképző tábla:

Logikai reg.	Fizikai reg.
R0	T21
R1	T3
R2	T46
R3	T8
D0	U9
D1	U24
D2	U17
D3	U4

REGISZTERÁTNEVEZÉS

- Eszköz: **Regiszter leképző tábla**
  - i. bejegyzése: az i. logikai regiszter melyik fizikaihoz van rendelve
- Eljárás:
  - Az utasítás operandusait fizikai regiszterekre cseréli
  - Az eredményt mindig egy új, nem használt fizikai regiszterbe teszi
  - Frissíti a regiszter leképző táblát

Eredeti:

```
i1:  D2 ← MEM[R1]
```

Átnevezés után:

```
i1:  U25 ← MEM[T3]
```

Regiszter leképző tábla:

Logikai reg.	Fizikai reg.
R0	T21
R1	T3
R2	T46
R3	T8
D0	U9
D1	U24
D2	**U25**
D3	U4

REGISZTERÁTNEVEZÉS

- Eszköz: **Regiszter leképző tábla**
  - i. bejegyzése: az i. logikai regiszter melyik fizikaihoz van rendelve
- Eljárás:
  - Az utasítás operandusait fizikai regiszterekre cseréli
  - Az eredményt mindig egy új, nem használt fizikai regiszterbe teszi
  - Frissíti a regiszter leképző táblát

Eredeti:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
```

Átnevezés után:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
```

Regiszter leképző tábla:

Logikai reg.	Fizikai reg.
R0	T21
R1	T3
R2	T46
R3	T8
D0	U9
D1	U24
D2	U25
D3	U4

REGISZTERÁTNEVEZÉS

- Eszköz: **Regiszter leképző tábla**
  - i. bejegyzése: az i. logikai regiszter melyik fizikaihoz van rendelve
- Eljárás:
  - Az utasítás operandusait fizikai regiszterekre cseréli
  - Az eredményt mindig egy új, nem használt fizikai regiszterbe teszi
  - Frissíti a regiszter leképző táblát

Eredeti:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
```

Átnevezés után:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
```

Regiszter leképző tábla:

Logikai reg.	Fizikai reg.
R0	T21
R1	T3
R2	T46
R3	T8
D0	U9
D1	U24
D2	U25
D3	**U26**

REGISZTERÁTNEVEZÉS

- Eszköz: **Regiszter leképző tábla**
  - i. bejegyzése: az i. logikai regiszter melyik fizikaihoz van rendelve
- Eljárás:
  - Az utasítás operandusait fizikai regiszterekre cseréli
  - Az eredményt mindig egy új, nem használt fizikai regiszterbe teszi
  - Frissíti a regiszter leképző táblát

Eredeti:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
```

Átnevezés után:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
```

Regiszter leképző tábla:

Logikai reg.	Fizikai reg.
R0	T21
R1	T3
R2	T46
R3	T8
D0	U9
D1	U24
D2	U25
D3	U26

REGISZTERÁTNEVEZÉS

- Eszköz: **Regiszter leképző tábla**
  - i. bejegyzése: az i. logikai regiszter melyik fizikaihoz van rendelve
- Eljárás:
  - Az utasítás operandusait fizikai regiszterekre cseréli
  - Az eredményt mindig egy új, nem használt fizikai regiszterbe teszi
  - Frissíti a regiszter leképző táblát

Eredeti:

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

Átnevezés után:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
i4:  T47 ← T3 + 8
```

Regiszter leképző tábla:

Logikai reg.	Fizikai reg.
R0	T21
R1	T3
R2	T46
R3	T8
D0	U9
D1	U24
D2	U25
D3	U26

REGISZTERÁTNEVEZÉS

- Eszköz: **Regiszter leképző tábla**
  - i. bejegyzése: az i. logikai regiszter melyik fizikaihoz van rendelve
- Eljárás:
  - Az utasítás operandusait fizikai regiszterekre cseréli
  - Az eredményt mindig egy új, nem használt fizikai regiszterbe teszi
  - Frissíti a regiszter leképző táblát

Eredeti:

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

Átnevezés után:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
i4:  T47 ← T3 + 8
```

Regiszter leképző tábla:

Logikai reg.	Fizikai reg.
R0	T21
R1	**T47**
R2	T46
R3	T8
D0	U9
D1	U24
D2	U25
D3	U26

REGISZTERÁTNEVEZÉS

- Eszköz: **Regiszter leképző tábla**
  - i. bejegyzése: az i. logikai regiszter melyik fizikaihoz van rendelve
- Eljárás:
  - Az utasítás operandusait fizikai regiszterekre cseréli
  - Az eredményt mindig egy új, nem használt fizikai regiszterbe teszi
  - Frissíti a regiszter leképző táblát

Eredeti:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
i5:  R2 ← R2 + 8
```

Átnevezés után:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
i4:  T47 ← T3 + 8
i5:  T48 ← T46 + 8
```

Regiszter leképző tábla:

Logikai reg.	Fizikai reg.
R0	T21
R1	T47
R2	T46
R3	T8
D0	U9
D1	U24
D2	U25
D3	U26

REGISZTERÁTNEVEZÉS

- Eszköz: **Regiszter leképző tábla**
  - i. bejegyzése: az i. logikai regiszter melyik fizikaihoz van rendelve
- Eljárás:
  - Az utasítás operandusait fizikai regiszterekre cseréli
  - Az eredményt mindig egy új, nem használt fizikai regiszterbe teszi
  - Frissíti a regiszter leképző táblát

Eredeti:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
i5:  R2 ← R2 + 8
```

Átnevezés után:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
i4:  T47 ← T3 + 8
i5:  T48 ← T46 + 8
```

Regiszter leképző tábla:

Logikai reg.	Fizikai reg.
R0	T21
R1	T47
R2	T48
R3	T8
D0	U9
D1	U24
D2	U25
D3	U26

REGISZTERÁTNEVEZÉS

- Eszköz: **Regiszter leképző tábla**
  - i. bejegyzése: az i. logikai regiszter melyik fizikaihoz van rendelve
- Eljárás:
  - Az utasítás operandusait fizikai regiszterekre cseréli
  - Az eredményt mindig egy új, nem használt fizikai regiszterbe teszi
  - Frissíti a regiszter leképző táblát

Eredeti:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
i5:  R2 ← R2 + 8

i6:  D2 ← MEM[R1]
```

Átnevezés után:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
i4:  T47 ← T3 + 8
i5:  T48 ← T46 + 8

i6:  U27 ← MEM[T47]
```

Regiszter leképző tábla:

Logikai reg.	Fizikai reg.
R0	T21
R1	T47
R2	T48
R3	T8
D0	U9
D1	U24
D2	U25
D3	U26

REGISZTERÁTNEVEZÉS

- Eszköz: **Regiszter leképző tábla**
  - i. bejegyzése: az i. logikai regiszter melyik fizikaihoz van rendelve
- Eljárás:
  - Az utasítás operandusait fizikai regiszterekre cseréli
  - Az eredményt mindig egy új, nem használt fizikai regiszterbe teszi
  - Frissíti a regiszter leképző táblát

Eredeti:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
i5:  R2 ← R2 + 8

i6:  D2 ← MEM[R1]
```

Átnevezés után:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
i4:  T47 ← T3 + 8
i5:  T48 ← T46 + 8

i6:  U27 ← MEM[T47]
```

Regiszter leképző tábla:

Logikai reg.	Fizikai reg.
R0	T21
R1	T47
R2	T48
R3	T8
D0	U9
D1	U24
D2	**U27**
D3	U26

REGISZTERÁTNEVEZÉS

- Eszköz: **Regiszter leképző tábla**
  - i. bejegyzése: az i. logikai regiszter melyik fizikaihoz van rendelve
- Eljárás:
  - Az utasítás operandusait fizikai regiszterekre cseréli
  - Az eredményt mindig egy új, nem használt fizikai regiszterbe teszi
  - Frissíti a regiszter leképző táblát

Eredeti:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
i5:  R2 ← R2 + 8

i6:  D2 ← MEM[R1]
i7:  D3 ← D2 * D0
```

Átnevezés után:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
i4:  T47 ← T3 + 8
i5:  T48 ← T46 + 8

i6:  U27 ← MEM[T47]
i7:  U28 ← U27 * U9
```

Regiszter leképző tábla:

Logikai reg.	Fizikai reg.
R0	T21
R1	T47
R2	T48
R3	T8
D0	U9
D1	U24
D2	U27
D3	U26

REGISZTERÁTNEVEZÉS

- Eszköz: **Regiszter leképző tábla**
  - i. bejegyzése: az i. logikai regiszter melyik fizikaihoz van rendelve
- Eljárás:
  - Az utasítás operandusait fizikai regiszterekre cseréli
  - Az eredményt mindig egy új, nem használt fizikai regiszterbe teszi
  - Frissíti a regiszter leképző táblát

Eredeti:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
i5:  R2 ← R2 + 8

i6:  D2 ← MEM[R1]
i7:  D3 ← D2 * D0
```

Átnevezés után:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
i4:  T47 ← T3 + 8
i5:  T48 ← T46 + 8

i6:  U27 ← MEM[T47]
i7:  U28 ← U27 * U9
```

Regiszter leképző tábla:

Logikai reg.	Fizikai reg.
R0	T21
R1	T47
R2	T48
R3	T8
D0	U9
D1	U24
D2	U27
D3	**U28**

REGISZTERÁTNEVEZÉS

- Eszköz: **Regiszter leképző tábla**
  - i. bejegyzése: az i. logikai regiszter melyik fizikaihoz van rendelve
- Eljárás:
  - Az utasítás operandusait fizikai regiszterekre cseréli
  - Az eredményt mindig egy új, nem használt fizikai regiszterbe teszi
  - Frissíti a regiszter leképző táblát

Eredeti:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
i5:  R2 ← R2 + 8

i6:  D2 ← MEM[R1]
i7:  D3 ← D2 * D0
i8:  MEM[R2] ← D3
```

Átnevezés után:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
i4:  T47 ← T3 + 8
i5:  T48 ← T46 + 8

i6:  U27 ← MEM[T47]
i7:  U28 ← U27 * U9
i8:  MEM[T48] ← U28
```

Regiszter leképző tábla:

Logikai reg.	Fizikai reg.
R0	T21
R1	T47
R2	T48
R3	T8
D0	U9
D1	U24
D2	U27
D3	U28

REGISZTERÁTNEVEZÉS

- Eszköz: **Regiszter leképző tábla**
  - i. bejegyzése: az i. logikai regiszter melyik fizikaihoz van rendelve
- Eljárás:
  - Az utasítás operandusait fizikai regiszterekre cseréli
  - Az eredményt mindig egy új, nem használt fizikai regiszterbe teszi
  - Frissíti a regiszter leképző táblát

Eredeti:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
i5:  R2 ← R2 + 8

i6:  D2 ← MEM[R1]
i7:  D3 ← D2 * D0
i8:  MEM[R2] ← D3
i9:  R1 ← R1 + 8
```

Átnevezés után:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
i4:  T47 ← T3 + 8
i5:  T48 ← T46 + 8

i6:  U27 ← MEM[T47]
i7:  U28 ← U27 * U9
i8:  MEM[T48] ← U28
i9:  T49 ← T47 + 8
```

Regiszter leképző tábla:

Logikai reg.	Fizikai reg.
R0	T21
R1	T47
R2	T48
R3	T8
D0	U9
D1	U24
D2	U27
D3	U28

REGISZTERÁTNEVEZÉS

- Eszköz: **Regiszter leképző tábla**
  - i. bejegyzése: az i. logikai regiszter melyik fizikaihoz van rendelve
- Eljárás:
  - Az utasítás operandusait fizikai regiszterekre cseréli
  - Az eredményt mindig egy új, nem használt fizikai regiszterbe teszi
  - Frissíti a regiszter leképző táblát

Eredeti:

```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
i5:  R2 ← R2 + 8

i6:  D2 ← MEM[R1]
i7:  D3 ← D2 * D0
i8:  MEM[R2] ← D3
i9:  R1 ← R1 + 8
```

Átnevezés után:

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
i4:  T47 ← T3 + 8
i5:  T48 ← T46 + 8

i6:  U27 ← MEM[T47]
i7:  U28 ← U27 * U9
i8:  MEM[T48] ← U28
i9:  T49 ← T47 + 8
```

Regiszter leképző tábla:

Logikai reg.	Fizikai reg.
R0	T21
R1	**T49**
R2	T48
R3	T8
D0	U9
D1	U24
D2	U27
D3	U28

REGISZTERÁTNEVEZÉS

- Eszköz: **Regiszter leképző tábla**
  - i. bejegyzése: az i. logikai regiszter melyik fizikaihoz van rendelve
- Eljárás:
  - Az utasítás operandusait fizikai regiszterekre cseréli
  - Az eredményt mindig egy új, nem használt fizikai regiszterbe teszi
  - Frissíti a regiszter leképző táblát

Eredeti:

```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

Átnevezés után:

```
i1:   U25 ← MEM[T3]
i2:   U26 ← U25 * U9
i3:   MEM[T46] ← U26
i4:   T47 ← T3 + 8
i5:   T48 ← T46 + 8

i6:   U27 ← MEM[T47]
i7:   U28 ← U27 * U9
i8:   MEM[T48] ← U28
i9:   T49 ← T47 + 8
i10:  T50 ← T48 + 8
```

Regiszter leképző tábla:

Logikai reg.	Fizikai reg.
R0	T21
R1	T49
R2	T48
R3	T8
D0	U9
D1	U24
D2	U27
D3	U28

REGISZTERÁTNEVEZÉS

- Eszköz: **Regiszter leképző tábla**
  - i. bejegyzése: az i. logikai regiszter melyik fizikaihoz van rendelve
- Eljárás:
  - Az utasítás operandusait fizikai regiszterekre cseréli
  - Az eredményt mindig egy új, nem használt fizikai regiszterbe teszi
  - Frissíti a regiszter leképző táblát

Eredeti:

```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

Átnevezés után:

```
i1:   U25 ← MEM[T3]
i2:   U26 ← U25 * U9
i3:   MEM[T46] ← U26
i4:   T47 ← T3 + 8
i5:   T48 ← T46 + 8

i6:   U27 ← MEM[T47]
i7:   U28 ← U27 * U9
i8:   MEM[T48] ← U28
i9:   T49 ← T47 + 8
i10:  T50 ← T48 + 8
```

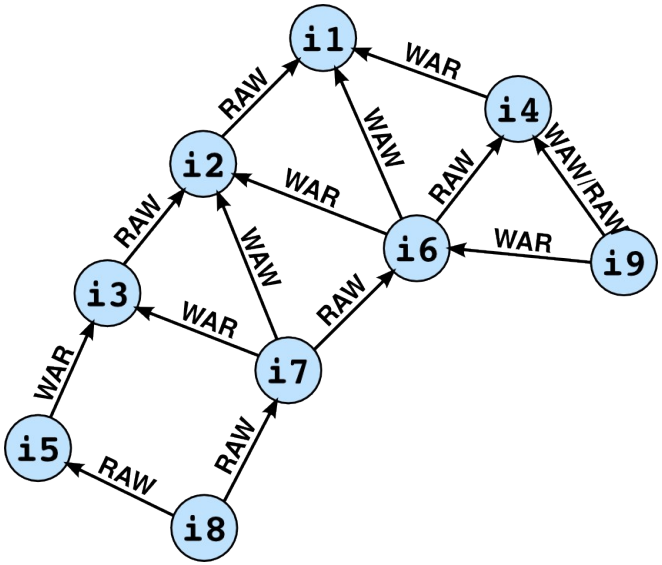
Regiszter leképző tábla:

Logikai reg.	Fizikai reg.
R0	T21
R1	T49
R2	**T50**
R3	T8
D0	U9
D1	U24
D2	U27
D3	U28

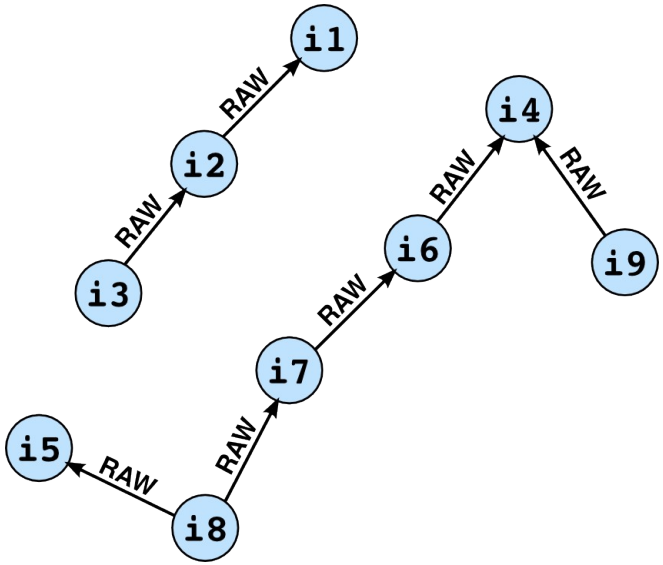
REGISZTERÁTNEVEZÉS

- Eredmény:
  - Eltűnt minden WAW és WAR!
  - ...hiszen az eredményt mindig „tiszta” regiszterbe tettük
- A precedenciagráf:

Regiszterátnevezés előtt:



Regiszterátnevezés után:

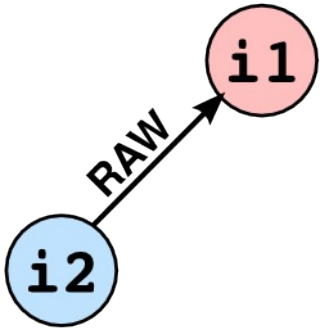


PÉLDA

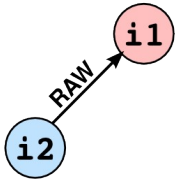

```
i1:  U25 ← MEM[T3]
i2:  U26 ← U25 * U9
i3:  MEM[T46] ← U26
i4:  T47 ← T3 + 8
i5:  T48 ← T46 + 8

i6:  U27 ← MEM[T47]
i7:  U28 ← U27 * U9
i8:  MEM[T48] ← U28
i9:  T49 ← T47 + 8
i10: T50 ← T48 + 8
```


```
i1:  D2 ← MEM[R1]
i2:  D3 ← D2 * D0
i3:  MEM[R2] ← D3
i4:  R1 ← R1 + 8
i5:  R2 ← R2 + 8

i6:  D2 ← MEM[R1]
i7:  D3 ← D2 * D0
i8:  MEM[R2] ← D3
i9:  R1 ← R1 + 8
i10: R2 ← R2 + 8
```

PÉLDA

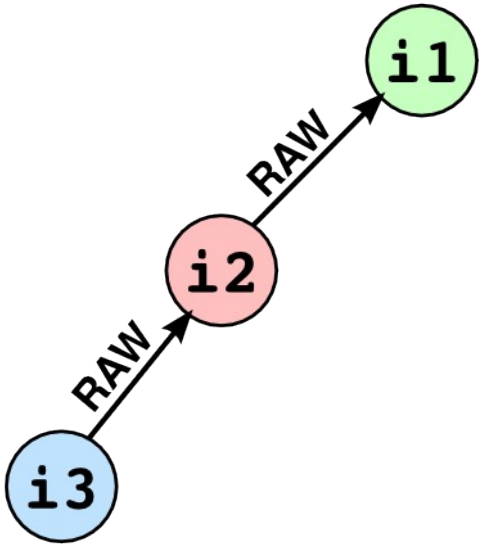



```
i1:   U25 ← MEM[T3]
i2:   U26 ← U25 * U9
i3:   MEM[T46] ← U26
i4:   T47 ← T3 + 8
i5:   T48 ← T46 + 8

i6:   U27 ← MEM[T47]
i7:   U28 ← U27 * U9
i8:   MEM[T48] ← U28
i9:   T49 ← T47 + 8
i10:  T50 ← T48 + 8
```

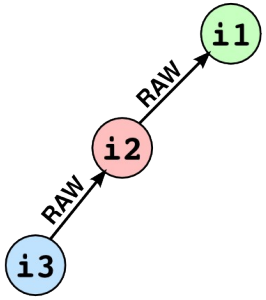



```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

PÉLDA

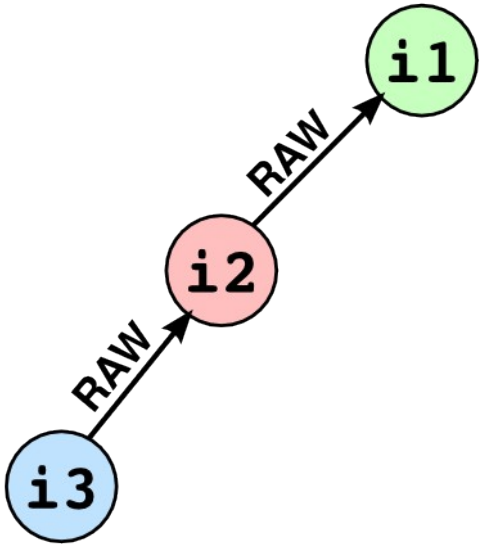



```
i1:   U25 ← MEM[T3]
i2:   U26 ← U25 * U9
i3:   MEM[T46] ← U26
i4:   T47 ← T3 + 8
i5:   T48 ← T46 + 8

i6:   U27 ← MEM[T47]
i7:   U28 ← U27 * U9
i8:   MEM[T48] ← U28
i9:   T49 ← T47 + 8
i10:  T50 ← T48 + 8
```

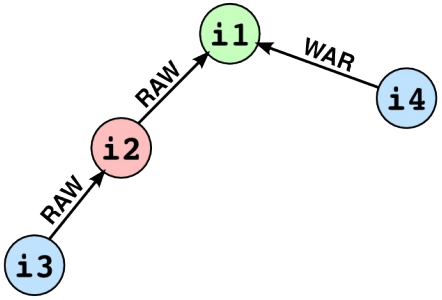



```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

PÉLDA

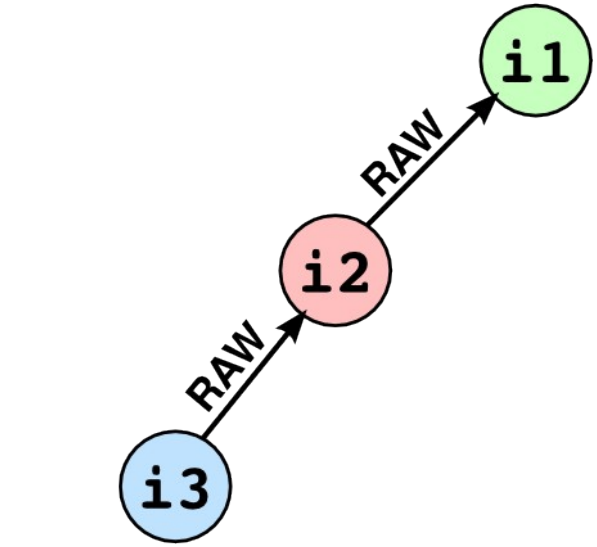


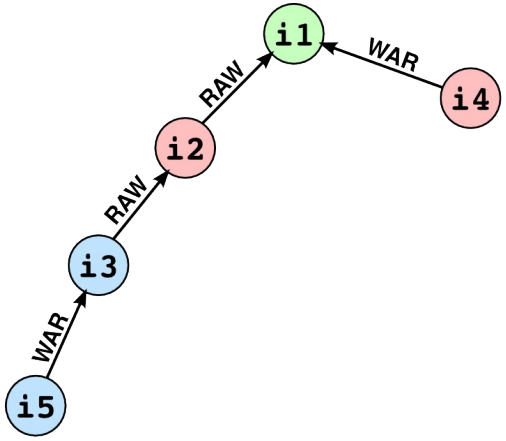

```
i1:   U25 ← MEM[T3]
i2:   U26 ← U25 * U9
i3:   MEM[T46] ← U26
i4:   T47 ← T3 + 8
i5:   T48 ← T46 + 8

i6:   U27 ← MEM[T47]
i7:   U28 ← U27 * U9
i8:   MEM[T48] ← U28
i9:   T49 ← T47 + 8
i10:  T50 ← T48 + 8
```




```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

PÉLDA

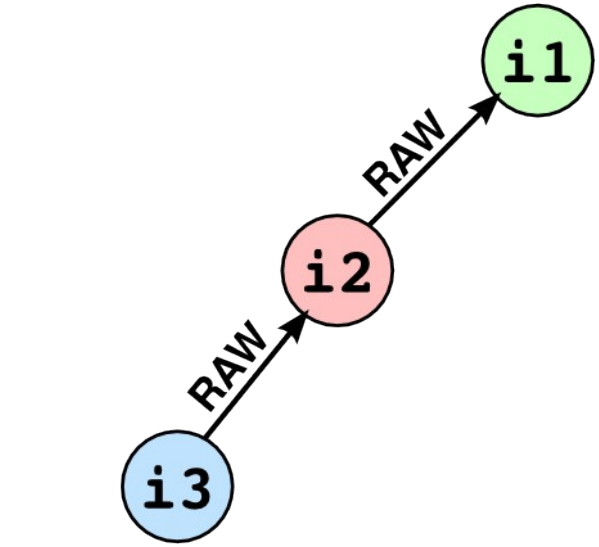


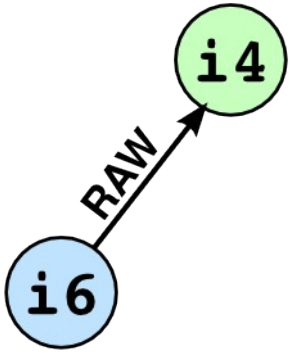


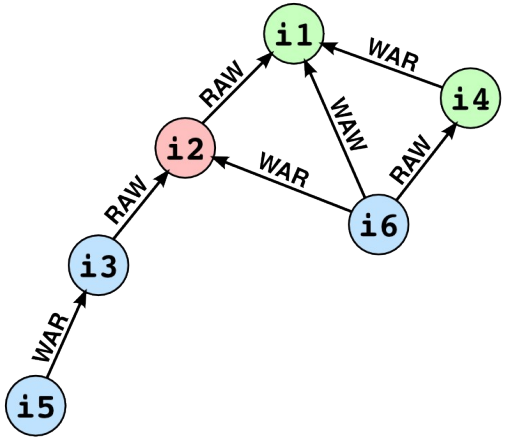

```
i1:   U25 ← MEM[T3]
i2:   U26 ← U25 * U9
i3:   MEM[T46] ← U26
i4:   T47 ← T3 + 8
i5:   T48 ← T46 + 8

i6:   U27 ← MEM[T47]
i7:   U28 ← U27 * U9
i8:   MEM[T48] ← U28
i9:   T49 ← T47 + 8
i10:  T50 ← T48 + 8
```

```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

PÉLDA



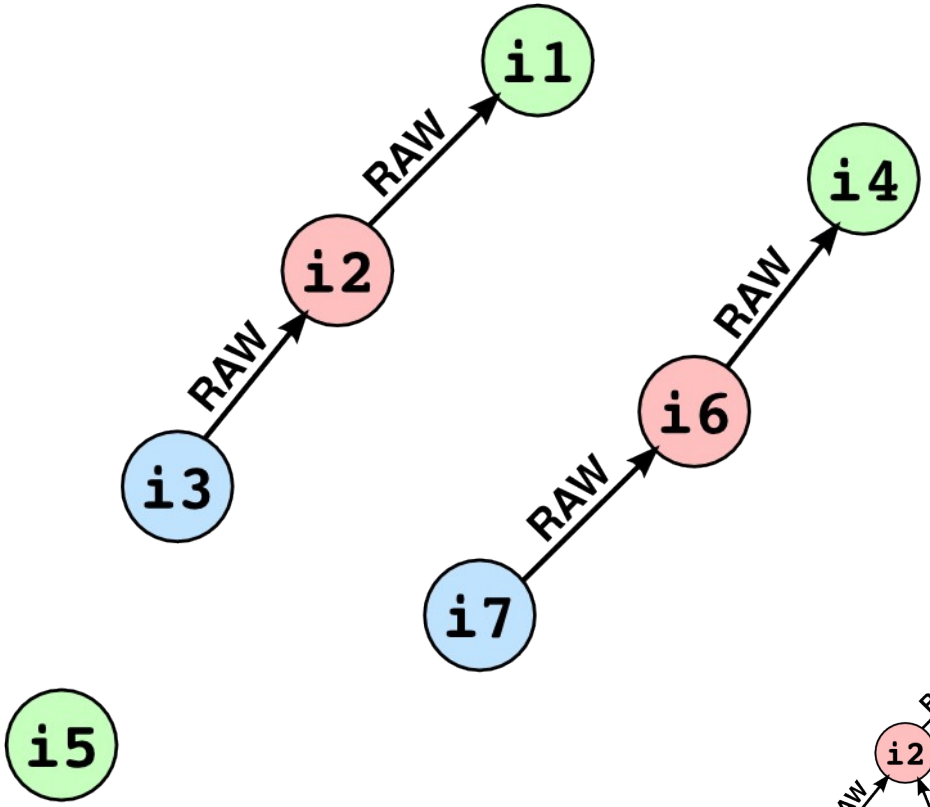




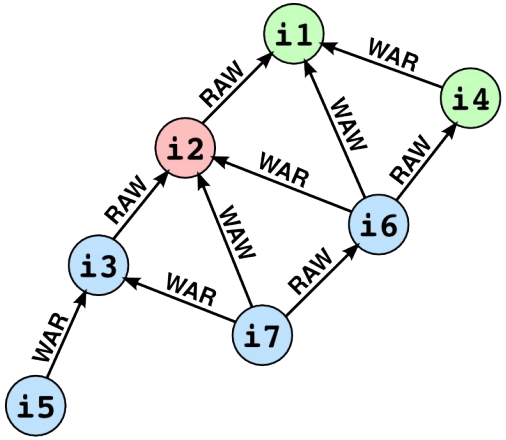

```
i1:   U25 ← MEM[T3]
i2:   U26 ← U25 * U9
i3:   MEM[T46] ← U26
i4:   T47 ← T3 + 8
i5:   T48 ← T46 + 8

i6:   U27 ← MEM[T47]
i7:   U28 ← U27 * U9
i8:   MEM[T48] ← U28
i9:   T49 ← T47 + 8
i10:  T50 ← T48 + 8
```

```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

PÉLDA

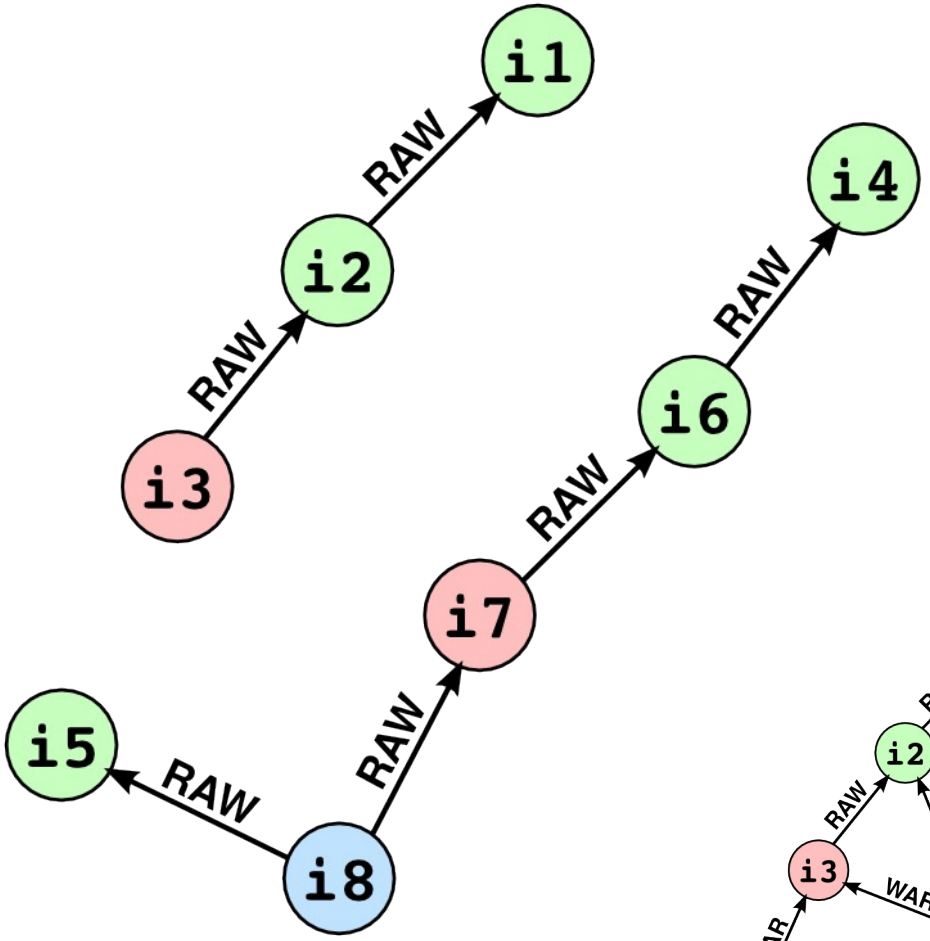


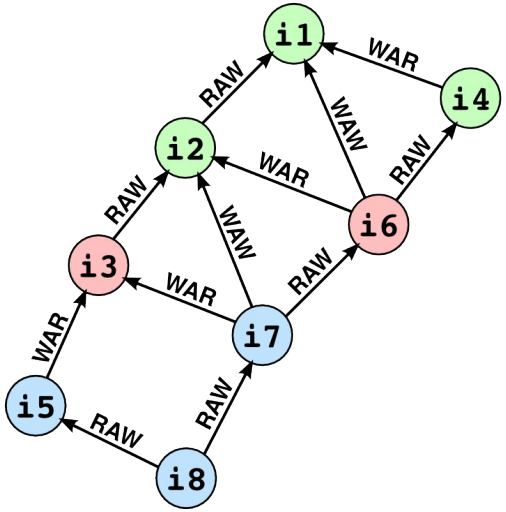

```
i1:   U25 ← MEM[T3]
i2:   U26 ← U25 * U9
i3:   MEM[T46] ← U26
i4:   T47 ← T3 + 8
i5:   T48 ← T46 + 8

i6:   U27 ← MEM[T47]
i7:   U28 ← U27 * U9
i8:   MEM[T48] ← U28
i9:   T49 ← T47 + 8
i10:  T50 ← T48 + 8
```




```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

PÉLDA






```
i1:   U25 ← MEM[T3]
i2:   U26 ← U25 * U9
i3:   MEM[T46] ← U26
i4:   T47 ← T3 + 8
i5:   T48 ← T46 + 8

i6:   U27 ← MEM[T47]
i7:   U28 ← U27 * U9
i8:   MEM[T48] ← U28
i9:   T49 ← T47 + 8
i10:  T50 ← T48 + 8
```

```
i1:   D2 ← MEM[R1]
i2:   D3 ← D2 * D0
i3:   MEM[R2] ← D3
i4:   R1 ← R1 + 8
i5:   R2 ← R2 + 8

i6:   D2 ← MEM[R1]
i7:   D3 ← D2 * D0
i8:   MEM[R2] ← D3
i9:   R1 ← R1 + 8
i10:  R2 ← R2 + 8
```

REGISZTERÁTNEVEZÉS

- Konklúzió:
  - Hatásos!
  - Minél nagyobb az utasítástároló, annál inkább

SORONKÍVÜLI VÉGREHAJTÁS HOZZÁVALÓI

- Több utasítást is le kell hívnunk, melyek közül válogatunk
  - Ezeket tárolni kell valahol:
    → **utasítástároló**
- Egy eljárás, ami eldönti az utasítások végrehajtási sorrendjét
    → **dinamikus ütemező**
- Egy jó ötlet, amivel az utasítások közötti függőségek csökkenthetők
    → **regiszterátnevezés**
- Egy trükk, hogy a külvilág sorrendi végrehajtást lásson
    → **sorrend-visszaállító buffer**

SORONKÍVÜLI VÉGREHAJTÁS HOZZÁVALÓI

- Több utasítást is le kell hívnunk, melyek közül válogatunk
  - Ezeket tárolni kell valahol:
    → **utasítástároló**
- Egy eljárás, ami eldönti az utasítások végrehajtási sorrendjét
    → **dinamikus ütemező**
- Egy jó ötlet, amivel az utasítások közötti függőségek csökkenthetők
    → **regiszterátnevezés**
- Egy trükk, hogy a külvilág sorrendi végrehajtást lásson
    → **sorrend-visszaállító buffer**

SORREND-VISSZAÁLLÍTÓ BUFFER

- Soron kívüli végrehajtás nemkívánatos mellékhatása:
  - Regiszterek/memória nem a program sorrendjében változnak
  - A memóriát több szerepélő is használhatja
    - Más processzorok,
    - perifériák,
    - stb.
  - Nem biztos, hogy erre fel vannak készülve!
- Megoldás: sorrend-visszaállító buffer (ROB)
  - Utasítás belépésekor kap egy bejegyzést
  - 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!

PÉLDA

Utasítás:	Kész?	Eredm.	Hova?
U25 ← MEM[T3]			U25

PÉLDA

Utasítás:	Kész?	Eredm.	Hova?
U25 ← MEM[T3]			U25
U26 ← U25 * U9			U26

PÉLDA

Utasítás:	Kész?	Eredm.	Hova?
U25 ← MEM[T3]	✔	<érték>	U25
U26 ← U25 * U9			U26
MEM[T46] ← U26			MEM[T46]

PÉLDA

Utasítás:	Kész?	Eredm.	Hova?
U25 ← MEM[T3]	✔	<érték>	U25
U26 ← U25 * U9			U26
MEM[T46] ← U26			MEM[T46]
T47 ← T3 + 8			T47

PÉLDA

Utasítás:	Kész?	Eredm.	Hova?
→ U25 ← MEM[T3]	✔	<érték>	U25
U26 ← U25 * U9			U26
MEM[T46] ← U26			MEM[T46]
T47 ← T3 + 8			T47
→ T48 ← T46 + 8			T48

PÉLDA

Utasítás:	Kész?	Eredm.	Hova?
U25 ← MEM[T3]	✔	<érték>	U25
U26 ← U25 * U9			U26
MEM[T46] ← U26			MEM[T46]
T47 ← T3 + 8	✔	<érték>	T47
T48 ← T46 + 8			T48
U27 ← MEM[T47]			U27

Kiszámolta a T47 értékét, de nem írhatja be a regiszter tárolóba.
Akinek kell, az innen kiveheti.

PÉLDA

Utasítás:	Kész?	Eredm.	Hova?
U25 ← MEM[T3]	✔	<érték>	U25
U26 ← U25 * U9			U26
MEM[T46] ← U26			MEM[T46]
T47 ← T3 + 8	✔	<érték>	T47
T48 ← T46 + 8	✔	<érték>	T48
U27 ← MEM[T47]			U27
U28 ← U27 * U9			U28

PÉLDA

Utasítás:	Kész?	Eredm.	Hova?
U25 ← MEM[T3]	✔	<érték>	U25
U26 ← U25 * U9	✔	<érték>	U26
MEM[T46] ← U26			MEM[T46]
T47 ← T3 + 8	✔	<érték>	T47
T48 ← T46 + 8	✔	<érték>	T48
U27 ← MEM[T47]	✔	<érték>	U27
U28 ← U27 * U9			U28
MEM[T48] ← U28			MEM[T48]

PÉLDA

Utasítás:	Kész?	Eredm.	Hova?
U25 ← MEM[T3]	✔	<érték>	U25
U26 ← U25 * U9	✔	<érték>	U26
MEM[T46] ← U26	✔	<érték>	MEM[T46]
T47 ← T3 + 8	✔	<érték>	T47
T48 ← T46 + 8	✔	<érték>	T48
→ U27 ← MEM[T47]	✔	<érték>	U27
U28 ← U27 * U9			U28
MEM[T48] ← U28			MEM[T48]
→ T49 ← T47 + 8			T49

Az alapelveken túl...

AZ ALAPELVEKEN TÚL...

- Mit tegyünk elágazás esetén?
  - Állítsuk meg az utasítások lehívását?
  - Nem! Inkább találjuk ki, merre folytatódik a program!
- RAW egymásrahatások nem csak a regiszterek között lehetnek!
  - Vannak memória egymásrahatások is
  - Ezek is kezelést igényelnek

SPEKULATÍV VÉGREHAJTÁS

- Utasítás csere-berének korlátot szabnak az elágazások
  - Két feltételes ugrás közé eső utasítások: ***basic block***
- Alapeset:
  - Utasítás csere-bere csak basic blokkon belül
  - ...hiszen ki tudja előre, hol folytatódik a program?
- ***Spekulatív végrehajtás:***
  - Megtippeljük a feltételes elágazások kimenetelét
    → elágazásbecslés
  - A tippelt ágról is töltünk be utasításokat
    ...és csere-beréljük őket basic blokkok *között* is!
  - Sokkal hatékonyabb
    - Mert általában sok a feltételes ugrás
  - Többlet adminisztációt igényel:
    - Spekulatíve betöltött utasítások kapnak egy flag-et a ROB-ban
    - Ha nem jön be a spekuláció (rossz elágazásbecslés):
      - A ROB-ból minden tévedésből betöltött utasítást törölni kell

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

ÖSSZEFOGLALÁS

- Sok mindent tanultunk:
  - Soron kívüli végrehajtás alapelve (precedenciagráf, adatáramlásos elv)
  - Eszközei:
    - Utasítástároló
    - Regiszterátnevezés
    - Sorrend-visszaállító buffer
  - Fejlett technikák:
    - Spekulatív végrehajtás
    - Memória egyértelműsítés

HÁLÓZATI RENDSZEREK
ÉS SZOLGÁLTATÁSOK
TANSZÉK

HÁLÓZATI RENDSZEREK ÉS SZOLGÁLTATÁSOK TANSZÉK

MŰEGYETEM 1782

Budapest,
2022.05.10.

SZÁMÍTÓGÉP ARCHITEKTÚRÁK

Az utasítás-pipeline szélesítése

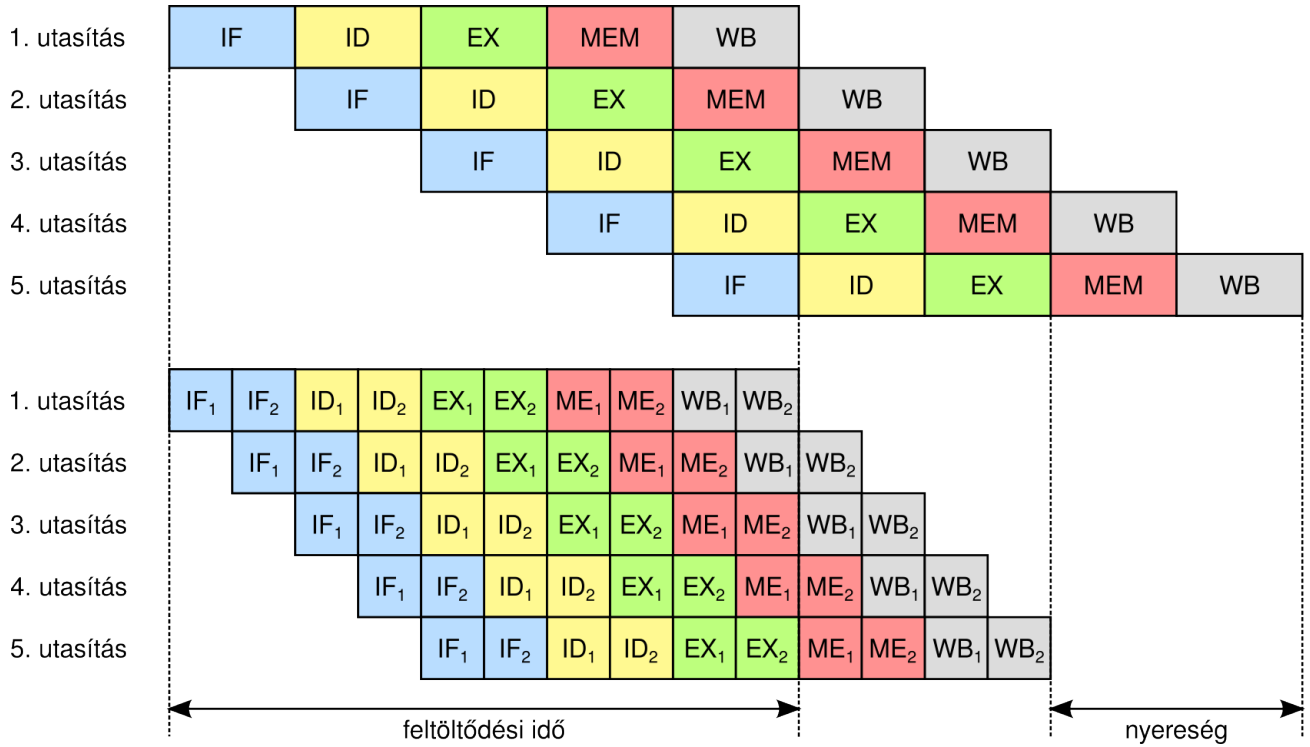
Horváth Gábor, Belső Zoltán

BME Hálózati Rendszerek és Szolgáltatások Tanszék
ghorvath@hit.bme.hu, belso@hit.bme.hu

UTASÍTÁSFELDOLGOZÁS GYORSÍTÁSA

1. Lehetőség: Mélyebb pipeline

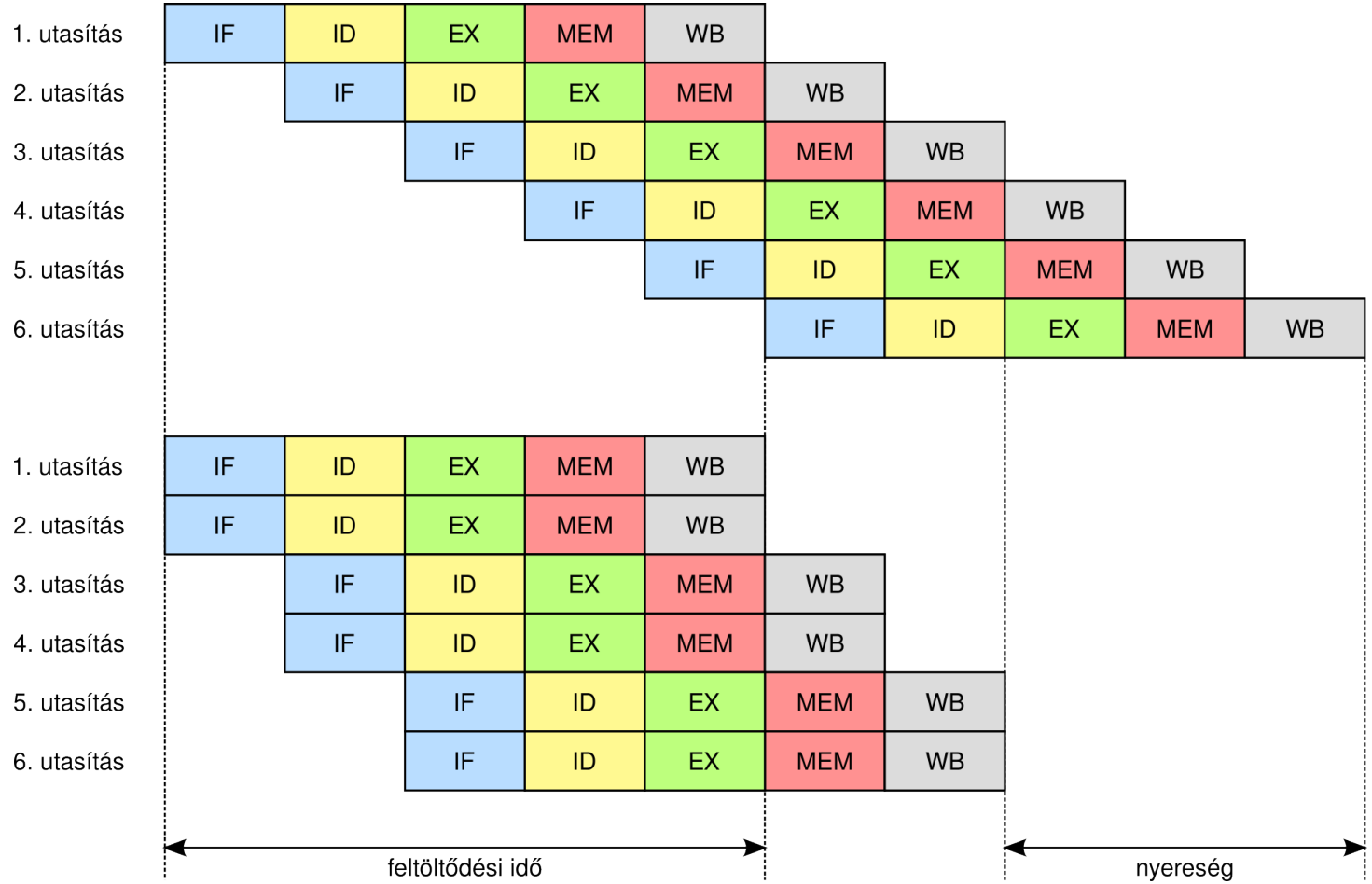
- Ciklusidőt felére csökkentjük (órajel 2x)
- Ehhez minden fázist két részre kell bontanunk
- Megdupláztuk az átviteli sebességet!



UTASÍTÁSFELDOLGOZÁS GYORSÍTÁSA

2. Lehetőség: Szélesebb pipeline

- Minden fázisban több utasítást dolgozunk fel



ÉRTÉKELÉS

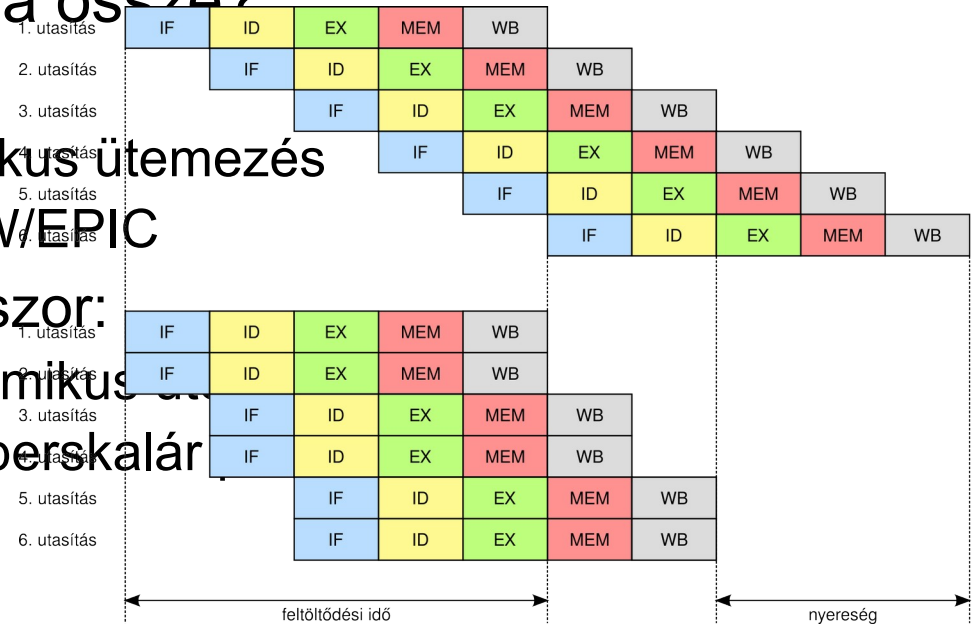
- Mélység: k-szoros, szélesség: m-szeres
  → Elméletileg m*k-szoros gyorsulás
- Gyakorlatilag több korlát:
  - m*k nagy: sok utasítás fut egyszerre → sok az egymásrahatás
    → sokszor kell megállni a feloldáshoz → romló hatékonyság
  - széles pipeline → m-el négyzetesen növő forwarding utak
  - k hiába 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

PÉLDÁK

	Mélység	Szélesség
Pentium	5	1
Pentium Pro	14	1-3 (1 bármilyen + 2 egyszerű)
Pentium 4 Prescott	31	3
Intel Core	14	4
Intel Core i7 Nehalem	16	4
Intel Core i7 Kaby Lake	14-19	4
Intel Atom	16	2
Alpha 21264	7	4
ARM Cortex A55	8-10	2
ARM Cortex A75	11-12	3
APPLE A10	?	6
POWER9	12-16	6

A PIPELINE SZÉLESÍTÉSE

- Hatékonyság kulcsa:
  - Elég sok független utasítás
- Ki válogatja össze?
  - Fordító:
    - Statikus ütemezés
    - VLIW/EPIC
  - Processzor:
    - Dinamikus ütemezés
    - Szuperskalár



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

Széles pipeline dinamikus ütemezéssel

SZUPERSKALÁR PROCESSZOROK

- A processzor több utasítást is le tud hívni egyszerre
- A processzor végzi a
  - Párhuzamosan végrehajtható utasítások kiválogatását
  - Az utasítások végrehajtó egységhez rendelését
  - A függőségi analízist (mikor hajtható végre egy utasítás)
- Ha minden fázisban m új utasítás végrehajtása kezdődik meg

  → ***m-utas szuperskalár processzor***
- Kétféle megoldás:
  - In-order:
    - végrehajtási sorrend: program követése
  - Out-of-order:
    - átrendezi az utasításokat, hogy gyorsabb legyen
    - 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:

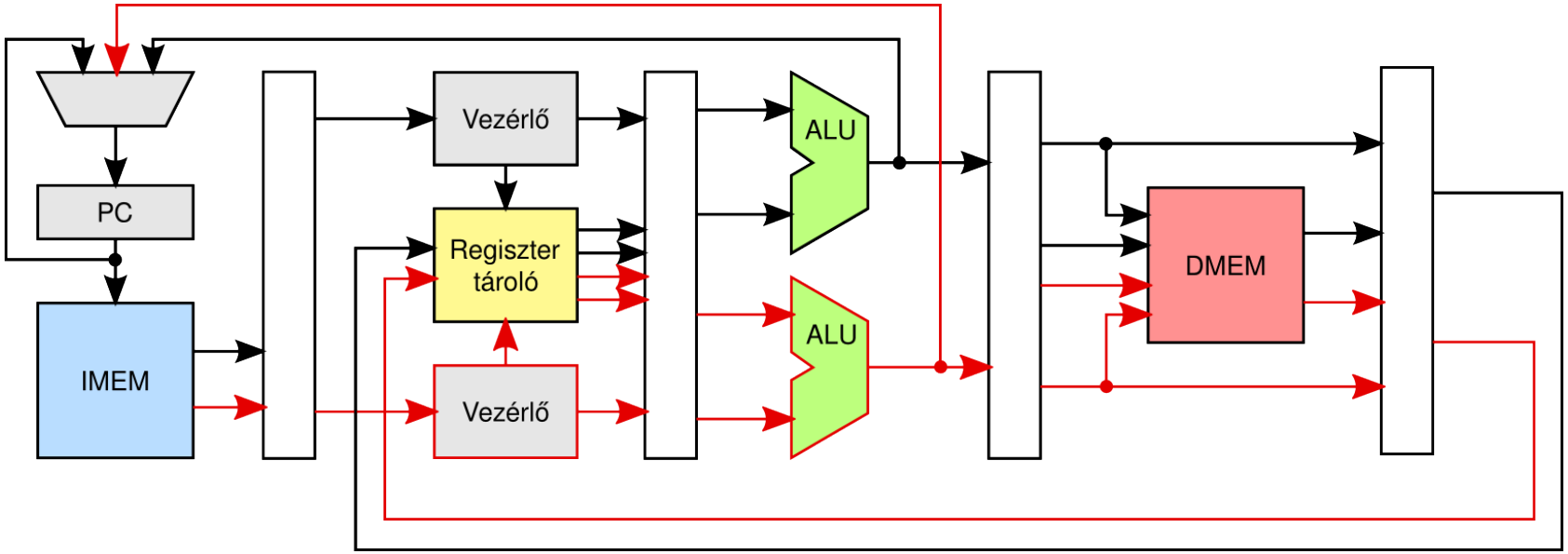
Órajel	Utasítások
1:	i1
2:	i2, i3
3:	i4

Out-of-order:

Órajel	Utasítások
1:	i1, i3
2:	i2, i4

IN-ORDER SZUPERSKALÁR PIPELINE

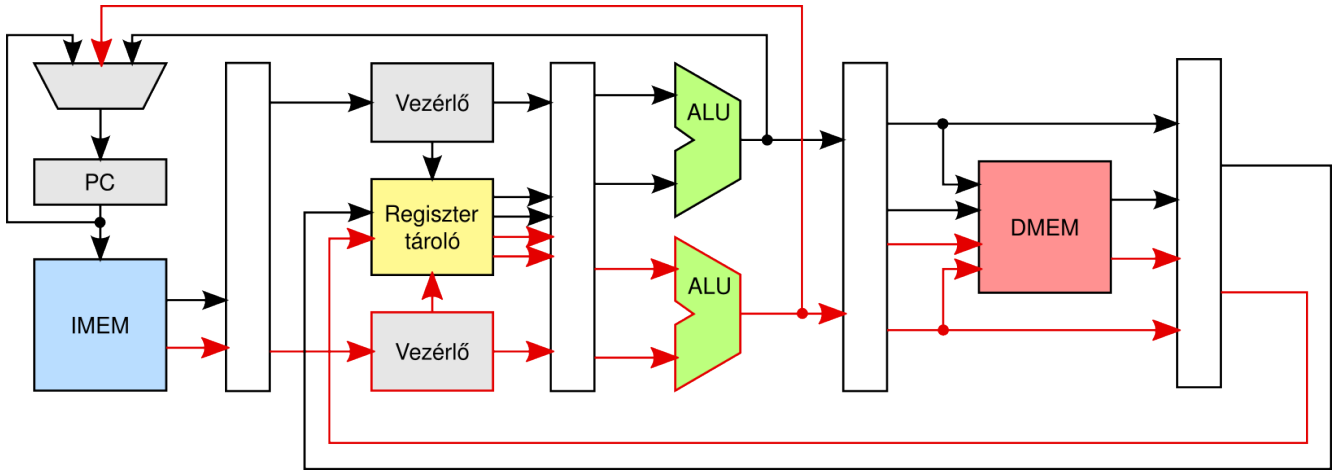
Hagyományos egy-utas nem szuperskalár processzor



Két-utas in-order szuperskalár processzor

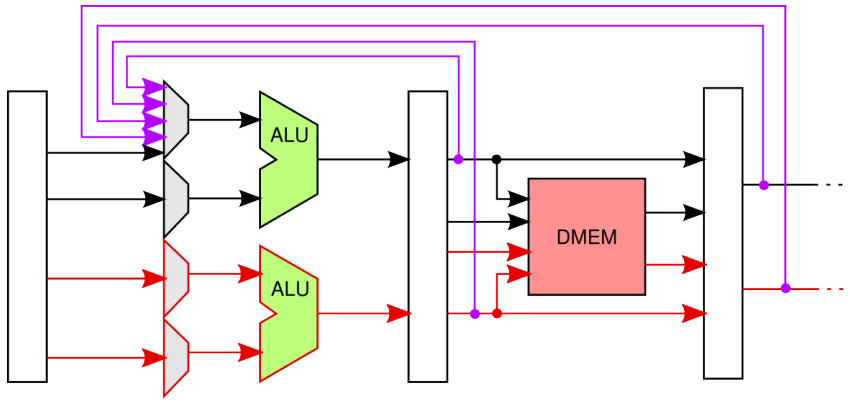
FÁZISOK

- **IF fázis:**
  - Most *m* utasítást kell lehívnia
    - Ha az *m* utasítás sorban jön, semmi gond
    - Ugrás esetén hatékonyságcsökkenés
- **ID fázis:**
  - ALU vezérlőjelek előállítása: nem nehezebb *m*-re mint 1-re
  - Operandusok kiolvasása: több portos regiszter tároló kell
  - Adatfüggőségek detektálása: m-el négyzetesen bonyolódik



FÁZISOK

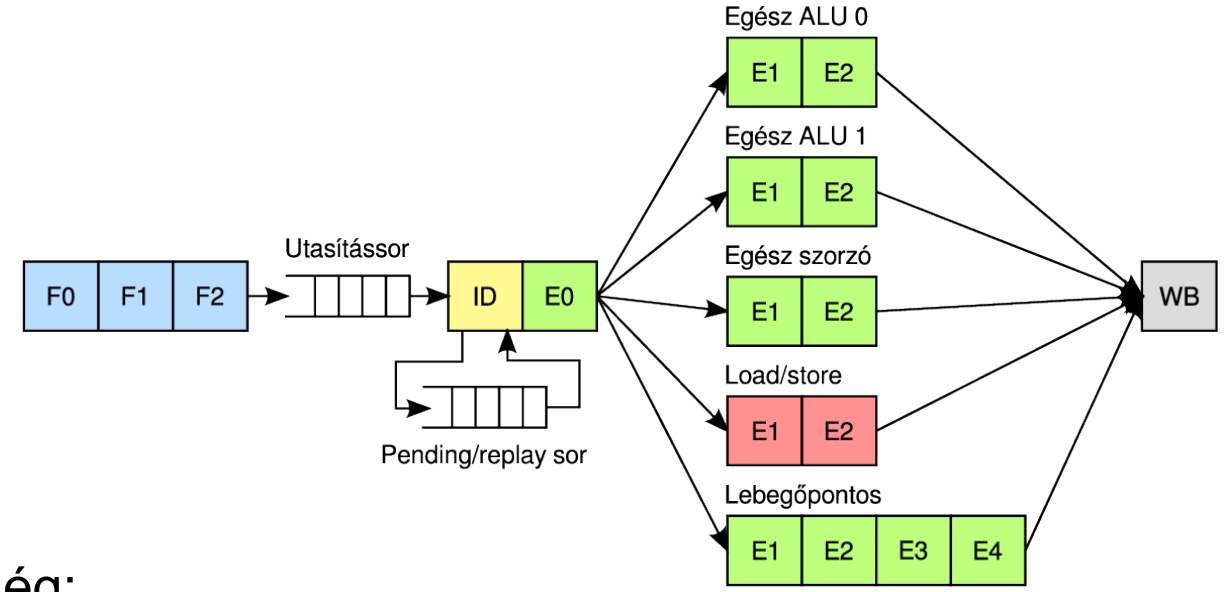
- **EX fázis:**
  - *m* db ALU beépítése nem gond
  - … de forwarding utak: m^2 van belőle
  - Egyenként 32 – 64 bites sínek → nem OK!



- **MEM fázis:**
  - Több portos adat cache kell
  - … vagy csak 1 memóriaműveletet engedünk egyszerre
- **WB fázis:**
  - Több portos regiszter tároló kell

AZ ARM CORTEX A53 PIPELINE-JA

- 2-utas in-order 8 fázisú futószalag



- IF egység:
  - F0: utasításszámláló inkrementálás, ugráskor számolás
  - F1: cím kiadása az utasításcache-nek.
  - F2: Megjön az adat az utasításcache-től → utasítássorba
- Cortex A55: ugyanez, külön Load és Store egységgel

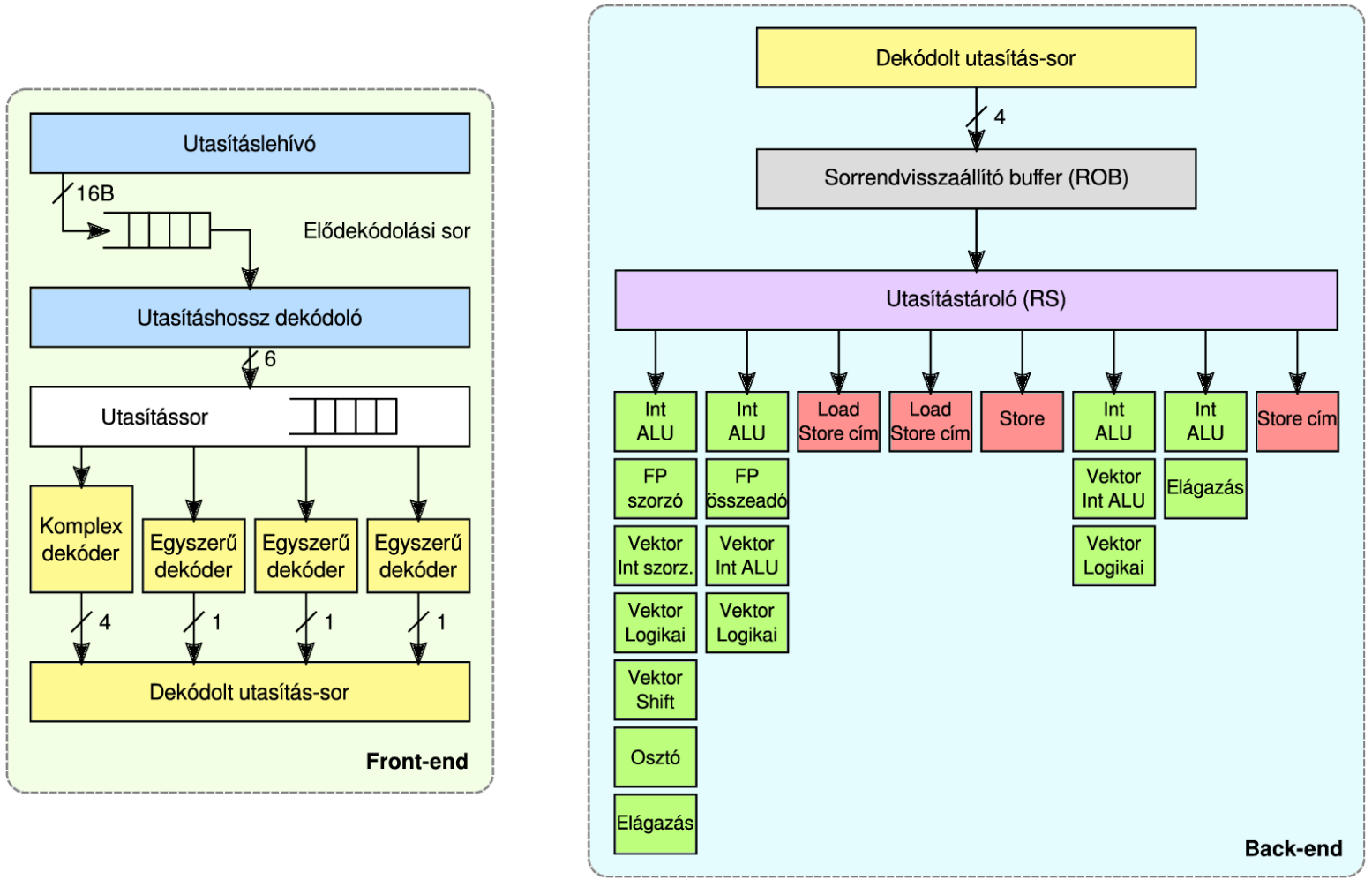
OUT-OF-ORDER SZUPERSKALÁR PIPELINE

- Egy utas esetet már láttuk
  - Nyilvántartásokat kell vezetni:
    - Az utasítások végrehajtási állapotáról
    - Műveleti egységek foglaltságáról
  - Döntéseket kell hozni
    - Utasítások műveleti egységekhez rendelése

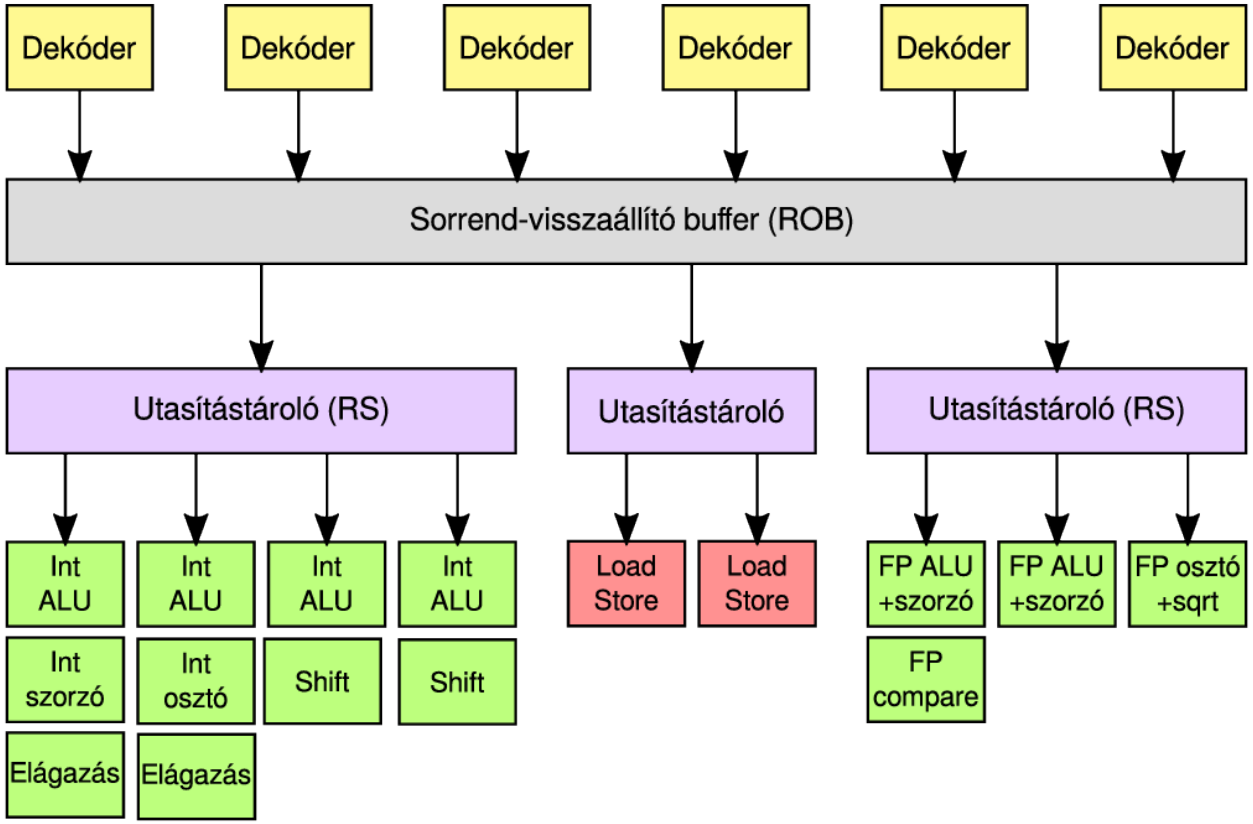
  → Bonyolult processzor
- Nem nehéz több utasra kiterjeszteni!
  - Több IF és DS egység kell
  - Egy ciklusban több utasítás lép be a tárolóba
  - Egy ciklusban több utasítás végrehajtása is elindulhat

AZ INTEL HASWELL PIPELINE-JA

- Széles, sorrenden kívüli szuperskalár



AZ APPLE CYCLONE PIPELINE-JA



SZUPERSKALÁR ÖSSZEFOGLALÓ

- In-order: egyszerűbb
- Out-of-order: bonyolultabb
  - 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

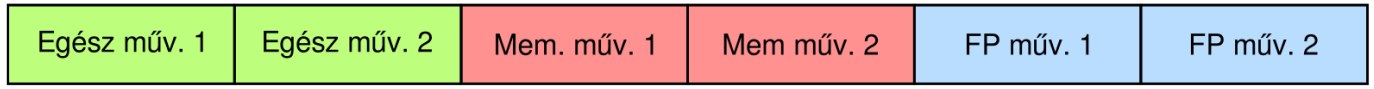
Széles pipeline statikus ütemezéssel

A MÁSIK VÉGLET

- Szuperskalár: a CPU keresgéli a független, párhuzamosan végrehajtható utasításokat
- A fordítóprogram is megteheti!
- Sőt, jobban! Több utasítást lát!
- Kapjunk vérszemet:
  - Csináljon mindent a fordító!
    - Párhuzamosan végrehajtható utasítások gyűjtése
    - Utasítások műveleti egységhez rendelése
    - Egymásrahatások detektálása és feloldása

A VLIW ARCHITEKTÚRA

- VLIW = Very Long Instruction Word
- 1 pipeline, ***utasításcsoport***okon dolgozik



- Csoportok szerepe:
  - Független utasítások kijelölése
  - Utasítások műveleti egységhez rendelése
- Nem használt pozíciókban: NOP
- 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

- 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. AMD a Radeon HD néhány generációjában: VLIW3 ill. VLIW4
- Hátrányok:
  - A program csak azon a processzoron fut, amire lefordították
  - **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 ÉS EPIC ARCHITEKTÚRÁK

- **Dinamikus VLIW architektúra**
  - A fordító csak csoportosít
  - A processzor ütemez
    - Észleli az egymásrahatásokat, szüneteket tud beiktatni
  - Előnyök:
    - Lehet cache-t csinálni!
- **EPIC**
  - VLIW-ben az utasítás csoportbeli helye a műveleti egységet is kijelölte:

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

  - 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

HÁLÓZATI RENDSZEREK
ÉS SZOLGÁLTATÁSOK
TANSZÉK

HÁLÓZATI RENDSZEREK ÉS SZOLGÁLTATÁSOK TANSZÉK

MŰEGYETEM 1782

Budapest,
2022.05.10.

SZÁMÍTÓGÉP ARCHITEKTÚRÁK

Elágazásbecslés

Horváth Gábor, Belső Zoltán

BME Hálózati Rendszerek és Szolgáltatások Tanszék
ghorvath@hit.bme.hu, belso@hit.bme.hu

PROCEDURÁLIS EGYMÁSRAHATÁSOK

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

AZ ELÁGAZÁSBECSLÉS FELADATA

- Szekvenciális futást megtörő utasítások:
  - **Feltétel nélküli ugrás:**
    - Direkt: `JUMP -28`
    - Indirekt: `JUMP R1`
    - `-28` vagy `R1`: ugrási cím
  - **Feltételes ugrás:**
    - Direkt: `JUMP -28 IF R2>0`
    - Indirekt: `JUMP R1 IF R2>0`
    - `R2>0`: ugrási feltétel
    - `-28` vagy `R1`: ugrási cím
  - **Szubrutinhívás:**
    - Direkt: `CALL -28`
    - Indirekt: `CALL R1`
    - Visszatérés: `RET`, ugrási cím a veremből
- Az elágazásbecslés feladata:
  - Kimenetel becslése, ha van ugrási feltétel
  - Ugrási cím becslése

AZ ELÁGAZÁSBECSLÉS JELENTŐSÉGE

- 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 Core 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 JELENTŐSÉGE

- 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 Skylake	16.5
Intel Atom	13
AMD K8 és K10	12
AMD Ryzen	19
Via Nano	16
ARM Cortex A53	7
ARM Cortex A72	15

Ugrási feltétel kimenetelének becslése

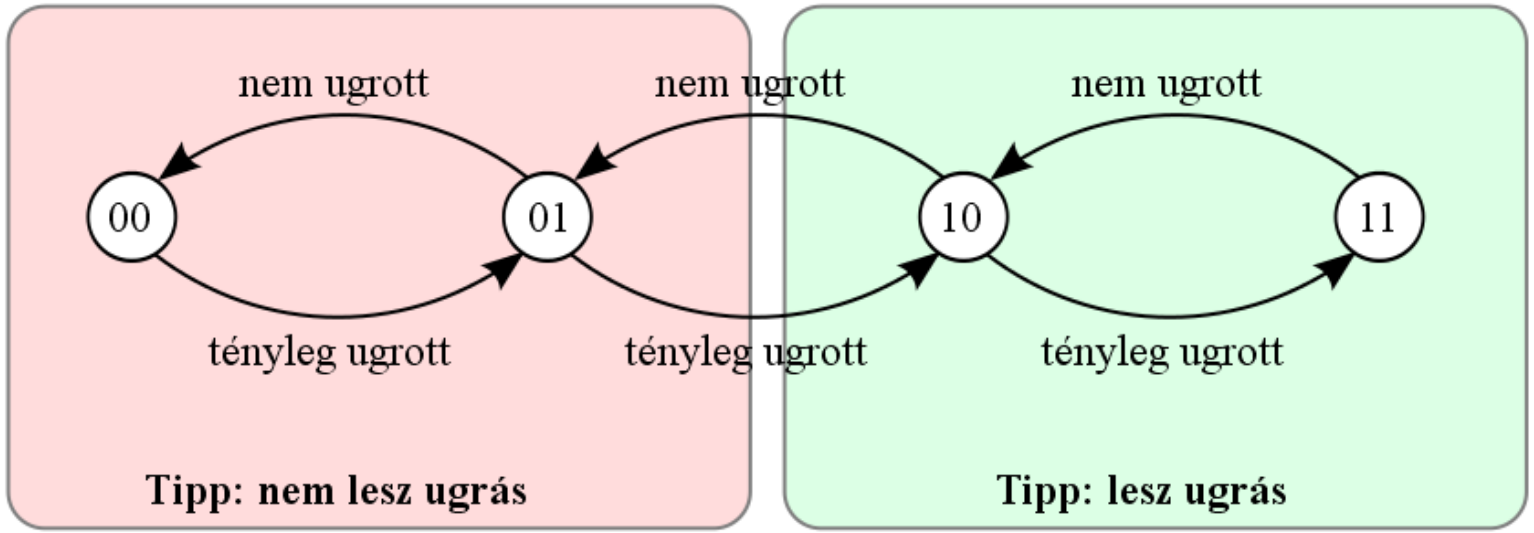
MI IS A FELADAT?

- Ismert (lehet): az ugró utasítás kimenetelei a múltban
- Feladat: mi lesz a következő kimenetel?
  - **1111111?**
  - **1111110110111110111111?**
  - **110011001100111?**
  - **11111111111000000000000?**

- Elvárások:
  - Legyen a becslő *gyors*
  - Legyen a becslő *egyszerű*
  - Legyen nagy a találati aránya

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

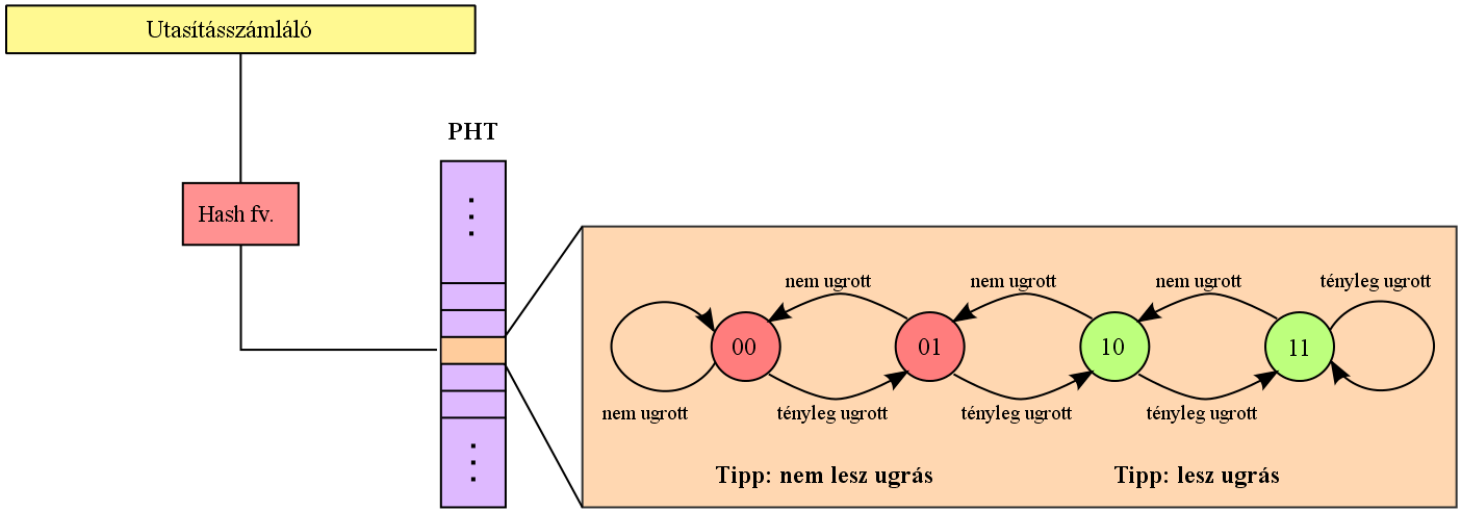
- Hatékonyságvizsgálat:

```
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)
    - Pl: Pentium 1



KORRELÁCIÓ FIGYELEMBE VÉTELE

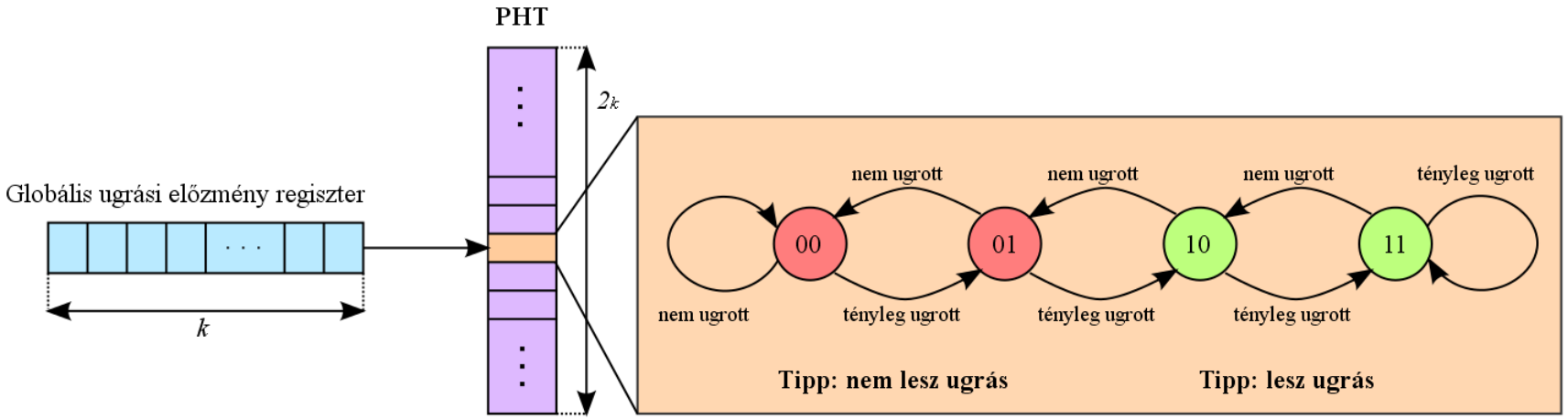
- Az ugró utasítások kimenetele gyakran függ más ugró utasítások kimenetelétől
- Példa:

```
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

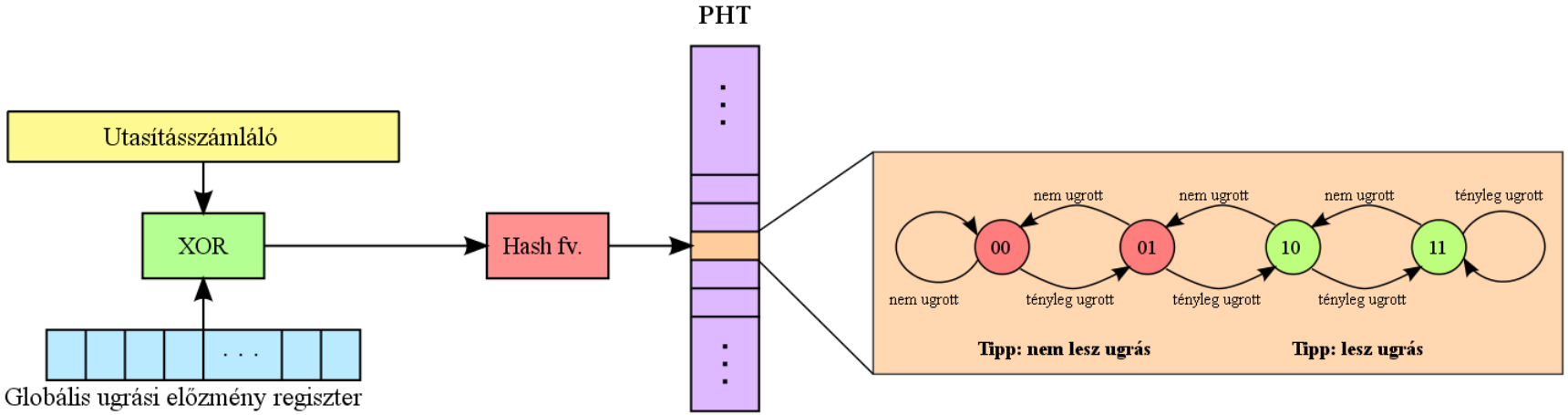


KOMBINÁLT PREDIKTOROK

- 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?

A GSHARE PREDIKTOR

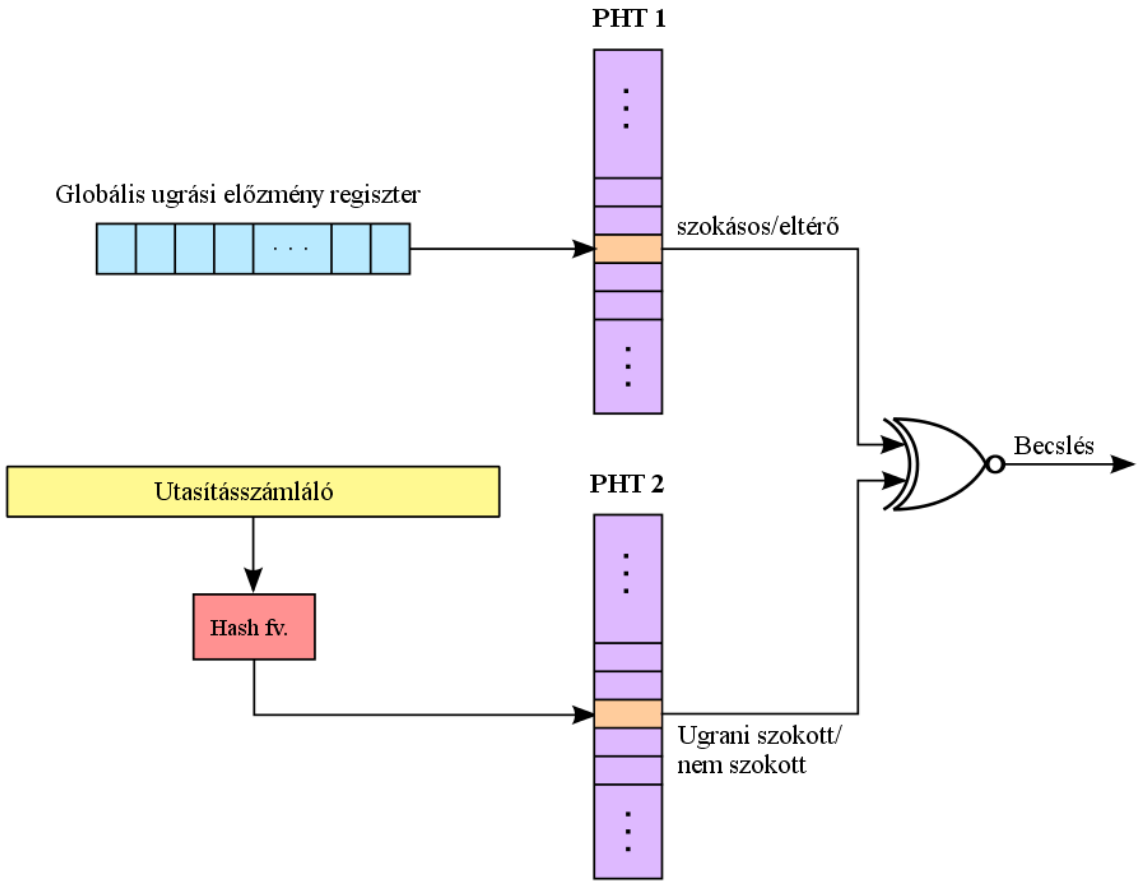
- Kombinálja a lokális és globális információkat
  - Lokális döntés: megnézzük, hogy az adott utasításhoz mit jósol a PHT
  - Globális döntés: megnézzük, hogy az előzmények mellett mit jósol a PHT
  - Kombináljuk a kettőt: PC XOR GBHR



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

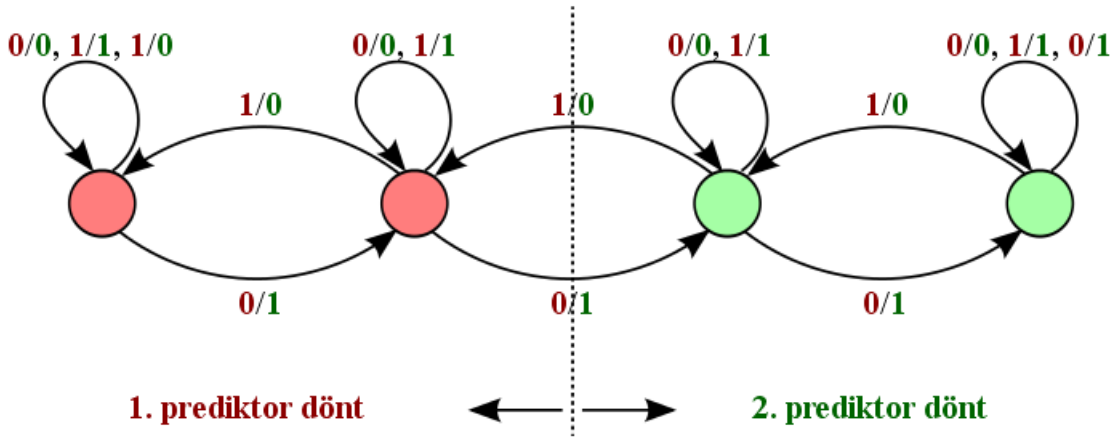
AZ AGREE PREDIKTOR

- 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

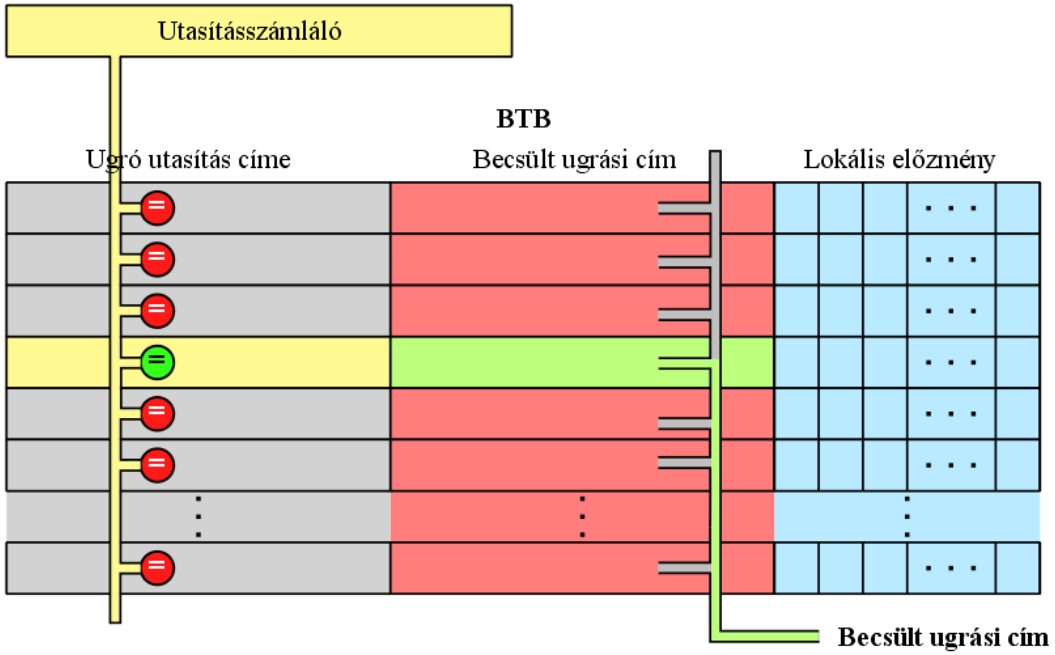
UGRÁSI CÍM BECSLÉSE

- 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 korrigáljuk 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)
- Például:



SZUBRUTINHÍVÁSBÓL VISSZATÉRÉSI CÍM BECSLÉSE

- A **RET** speciális ugró utasítás
  - Ugrási cím a lassú memóriában van (stack)
- Becsléshez hatékony adatszerkezet: **return stack**
  - A CPU-ban található
  - Nagy sebesség, kis kapacitás
  - Szubrutinhíváskor a visszatérési cím
    - az igazi (lassú) stack tetejére kerül
    - ...a return stack tetejére is bekerül!
  - Visszatéréskor:
    - Nem kell megvárni az igazi lassú stack-et!
    - A return stack ciklusidőn belül megmondja a címet
  - Addig hatékony, amíg nem telik meg
    - Ha nincs túl sok egymásba ágyazott függvény hívás

TIPIKUS UGRÁSI CÍM BUFFER PARAMÉTEREK

Processzor	BTB bej.-ek száma	BTB szervezése	Return stack
Intel Pentium I MMX	256	4 utas asszoc.	nincs
Intel Pentium 4	4096	val. 8 utas asszoc.	nincs
Intel Core2 (ciklusokhoz)	128	2 utas asszoc.	16
Intel Core2 (indir. ugr.)	8192	4 utas asszoc.	
Intel Core2 (egyéb ugr.)	2048	4 utas asszoc.	
Intel Atom	128	4 utas asszoc.	8
AMD Steamroller	L1: 512, L2: 10240	L1: 4 utas, L2: 5 utas	24
AMD Ryzen	8/256/4096	?	31
Via Nano	4096	4 utas asszoc.	Nagyon mély
ARM Cortex A8/A9	512	2 utas asszoc.	8

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
  - Pl.: ARM Cortex A15, Intel Core Nehalem, AMD Ryzen, stb.

Elágazásbecslés-tudatos programozás

ELÁGAZÁSOK SZÁMÁNAK CSÖKKENTÉSE

- 1. példa:

```
for (int i=0; i<N; i++)
    if (data[i] > 500)
        sum += data[i];
```

- A feltétel kiváltható egy logikai kifejezéssel
  - `data[i]>500` ↔ `(data[i]-501) >= 0`
  - `data[i]-501` jobbra shiftelve (aritmetikai shift!):
    - `000...0` vagy `111...1`
  - Maszkoljuk vele az összeadást!
- Elágazásmentesített kód:

```
for (int i=0; i<N; i++) {
    int t = (data[i]-501) >> 31;
    sum += ~t & data[i];
}
```

ELÁGAZÁSOK SZÁMÁNAK CSÖKKENTÉSE

- 2. példa:

```
for (int i=0; i<N; i++)
    if (min<=data[i] && data[i]<=max)
        sum += data[i];
```

- Első lépés: két feltételből egyet csinálunk
  - `min<=data[i] && data[i]<=max`

    ↔ `(unsigned)(data[i]-min)<=max-min`
  - Már csak egy feltételünk van:

```
for (int i=0; i<N; i++)
    if ((unsigned)(data[i]-min) <= max-min)
        sum += data[i];
```

- Második lépés: az előbbi trükk

```
for (int i=0; i<N; i++) {
    int t = (max-min-(unsigned)(data[i]-min)) >> 31;
    sum += ~t & data[i];
}
```

ELÁGAZÁSOK SZÁMÁNAK CSÖKKENTÉSE

- 3. példa:

```
for (int i=0; i<N; i++)
   if (!((data[i]>='a' && data[i]<='z') || (data[i]>='A' && data[i]<='Z')))
        data[i] = ' ';
```

- Trükk: segédtömb (**look-up table**, LUT)
  - Minden betűre a cserélendő karakter
- Segédtömb előkészítés + elágazásmentes kód:

```
for (int j=0; j<256; j++)
    if (!((j>='a' && j<='z') || (j>='A' && j<='Z')))
        LUT[j] = ' ';
    else
        LUT[j] = j;
for (int i=0; i<N; i++)
    data[i] = LUT[data[i]];
```

- Az előkészítés ideje fix
  - Ha N nagy, eltörpül

ELÁGAZÁSOK SZÁMÁNAK CSÖKKENTÉSE

- Mérési eredmények:

	i7-2600	Pentium 4	Rasp. Pi	RK3188
1. példa, eredeti	7,583 ms	14,122 ms	59,202 ms	11,296 ms
1. példa, elágazásmentes	1,297 ms	4,251 ms	58,410 ms	8,628 ms
2. példa, eredeti	8,211 ms	19,6 ms	73,267 ms	21,496 ms
2. példa, 1 elágazással	7,942 ms	14,295 ms	61,578 ms	13,347 ms
2. példa, elágazásmentes	1,203 ms	4,252 ms	58,328 ms	8,268 ms
3. példa, eredeti	6,533 ms	10,377 ms	48,532 ms	21,397 ms
3. példa, segédtömbbel	1,151 ms	3,641 ms	37,896 ms	18,240 ms

AZ ELÁGAZÁSOK KISZÁMÍTHATÓBBÁ TÉTELE

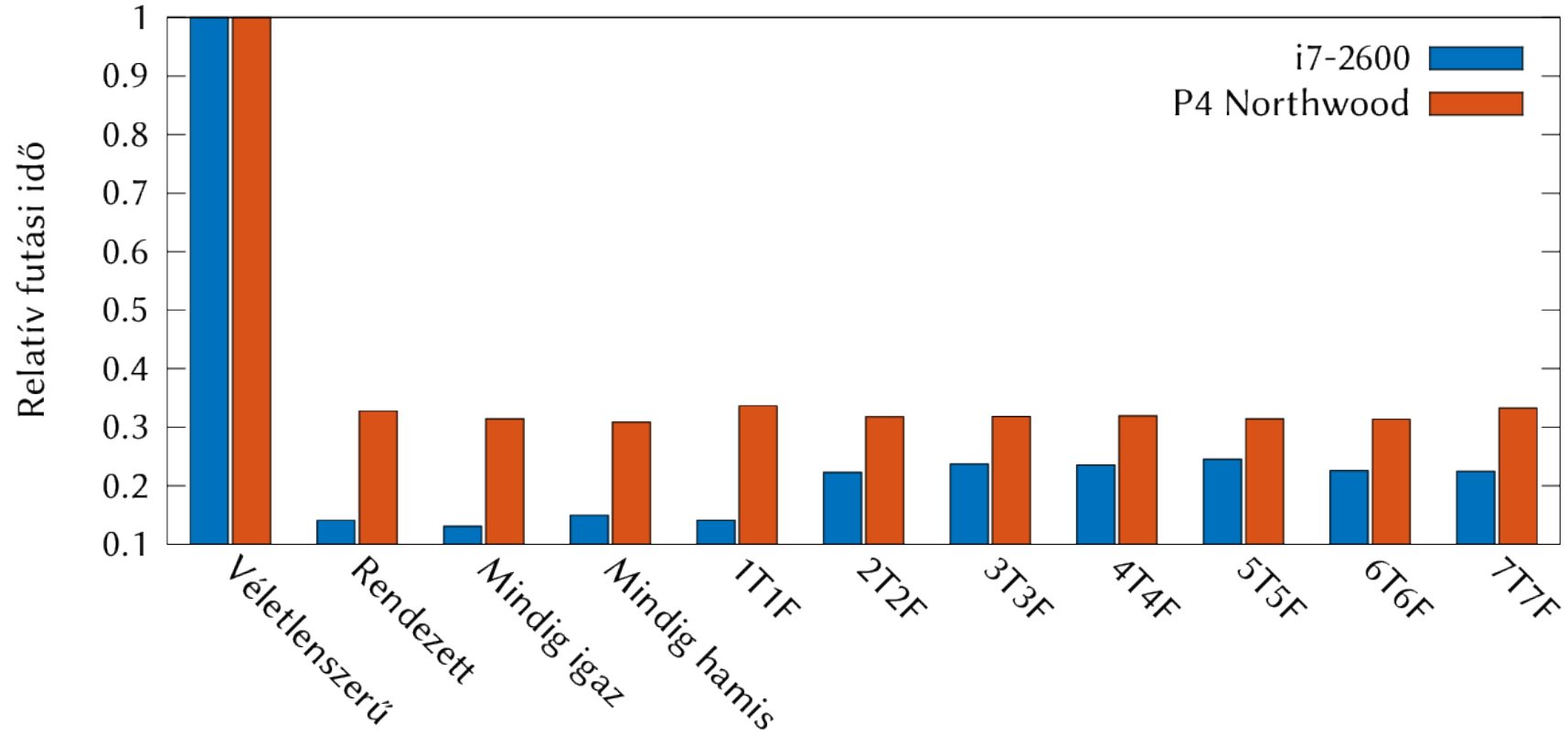
- Újra az 1. példa:

```
for (int i=0; i<N; i++)
    if (data[i] > 500)
        sum += data[i];
```

- Elemek rendezetlenek → kiszámíthatatlan kimenetel
- Rendezzük az elemeket!
- Mérések:
  - Rendezetlen eset
  - Rendezett tömb
  - Olyan tömb, amiben minden elemre teljesül
  - Olyan tömb, amiben egyetlen elemre sem teljesül
  - Adott teljesülési minta mellett

AZ ELÁGAZÁSOK KISZÁMÍTHATÓBBÁ TÉTELE

- Mérési eredmények:



AZ UGRÁSI CÍM BECSÜLHETŐSÉGÉNEK JAVÍTÁSA

- Példa: heterogén kollekció 16 leszármazottal

```
class A {
public:
    virtual int value ()=0;
    virtual int type () const =0;
    virtual ~A() {}
};
```

```
class B1 : public A {
  int v;
public:
  B1 () : v(rand()) {}
  int value () { return ++v; }
  int type () const { return 1; }
  ~B1 () {}
};
```

```
class B16 : public A {
  int v;
public:
  B16 () : v(rand()) {}
  int value () { return ++v; }
  int type () const { return 16; }
  ~B16 () {}
};
```

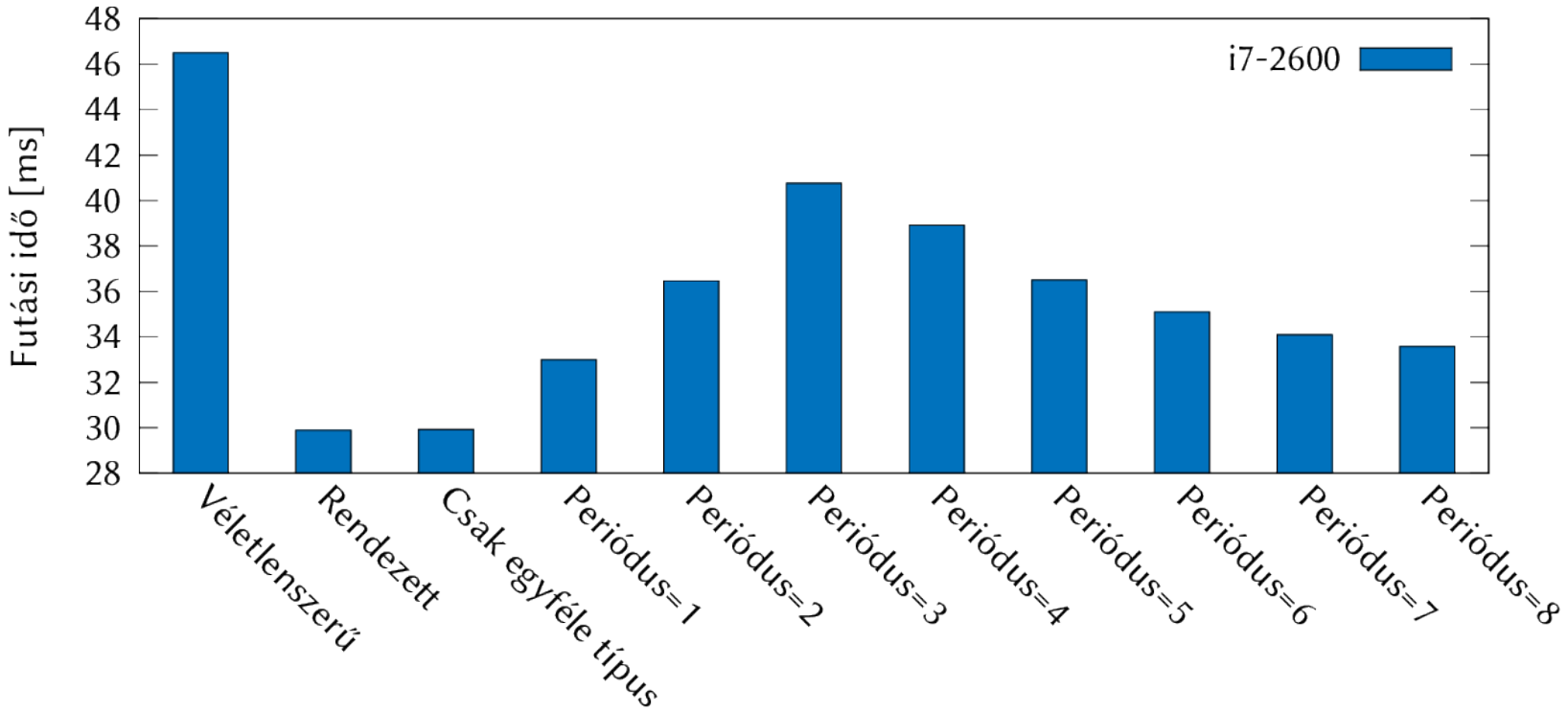
- Bejárás:

```
sum = 0;
for (i=0; i<siz; i++)
    sum += data[i]->value();
```

- Gond:
  - A virtuális függvényhívás **indirekt** ugrás
  - Az ugrási cím kiszámíthatatlan!
- Megoldás:
  - Rendezzük típus szerint a tömböt!

AZ UGRÁSI CÍM BECSÜLHETŐSÉGÉNEK JAVÍTÁSA

- Mérési eredmények:



HÁLÓZATI RENDSZEREK
ÉS SZOLGÁLTATÁSOK
TANSZÉK

HÁLÓZATI RENDSZEREK ÉS SZOLGÁLTATÁSOK TANSZÉK

MŰEGYETEM 1782

Budapest,
2022.05.10.

SZÁMÍTÓGÉP ARCHITEKTÚRÁK

Párhuzamos feldolgozás

Horváth Gábor, Belső Zoltán

BME Hálózati Rendszerek és Szolgáltatások Tanszék
ghorvath@hit.bme.hu, belso@hit.bme.hu

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.
- **MIMD** (multiple instruction, multiple data)
  - Több utasítássorozat több adaton dolgozik
  - Multiprocesszoros rendszerek
- **MISD** (multiple instruction, single data)
  - Hibatűrő rendszerekben

SIMD (single instruction, multiple data)

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

SIMD UTASÍTÁSKÉSZLET KIEGÉSZÍTÉSEK

- Vektor utasítások otthoni használatra is hasznosak
  - Képfeldolgozási célokra
  - 3D grafikai alkalmazásokban / játékokban
  - Egyszerű tudományos alapfeladatok is jól vektorizálhatók
- Sok CPU támogat vektorműveleteket - de attól még nem lesznek vektorprocesszorok!

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	32/16	64/128 bit	Int: 8x8, 4x16, 2x32 bit Float: 2x32 bit

MIMD (multiple instruction, multiple data)

HATÉKONY UTASÍTÁSFELDOLGOZÁS

- Hogyan tettük eddig hatékonnyá az utasítás-végrehajtást?
  - ***Utasítás szintű párhuzamosítással***
    - **Egyszerű pipeline:** átlapolt utasítás végrehajtás
    - **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

- Korlátok:
  - Minél több utasítás áll feldolgozás alatt
    → annál nagyobb a sansz az egymásrahatásra
  - Minél több a 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

- Miért szeretjük mégis 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
- **Alternatíva:**
  - 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
    - Van több program counter, de ezek mégsem valódi multiprocesszoros rendszerek!
  - Valódi multiprocesszoros rendszerek

TÖBB SZÁLAT KEZELŐ PROCESSZOROK

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

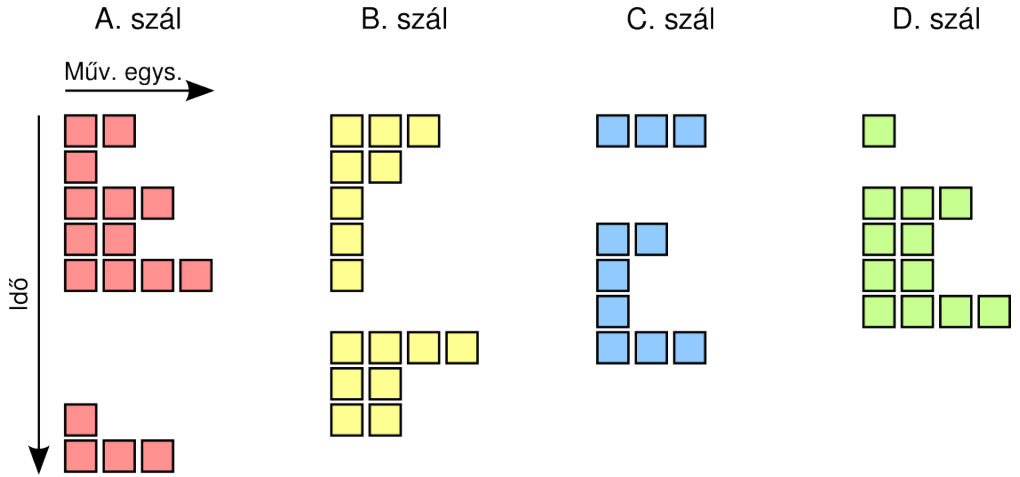
    → Szünet történik
- Megoldás:
  - Kezeljünk több utasításszámlálót!
  - Ha fennakadás van, a megoldásig tegyük félre az utasítássorozatot, és vegyük elő a másikat

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

IDŐOSZTÁSOS TÖBBSZÁLÚ VÉGREHAJTÁS

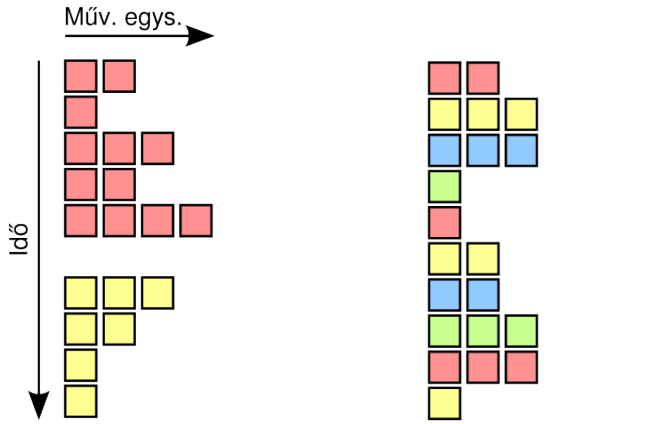
- A többszálú végrehajtást támogató processzorok egy része
  - ***Finom felbontású multi-threading***-et támogat:
    - Minden órajelben szálat (utasításszámlálót) vált
    - Mire egy szálra újra sor kerül, jó eséllyel megoldódik a problémája (nem kell külön szünet)
  - ***Durva felbontású multi-threading***-et 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

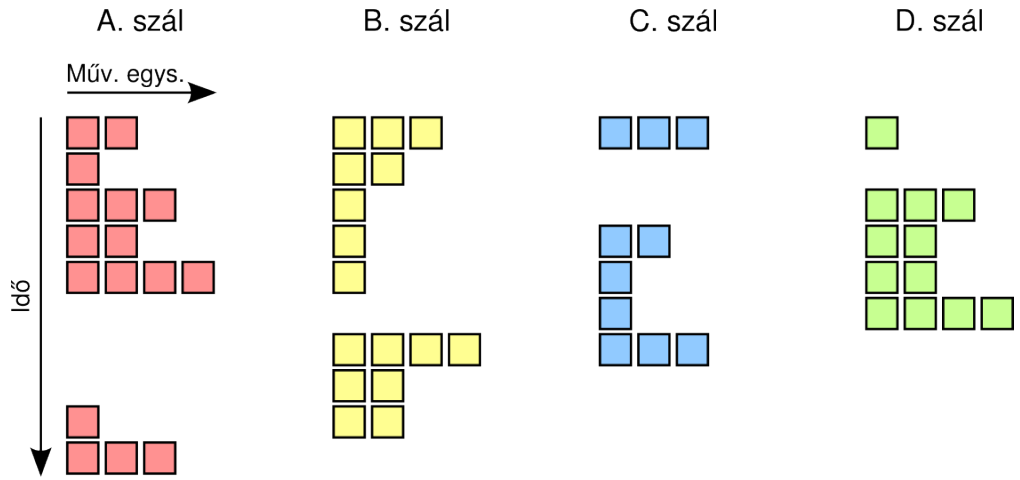
Finom felbontású többszálú végrehajtás



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
IBM POWER8	2013	Szimultán	8
UltraSPARC T1	2005	Finom felbontású	4
UltraSPARC T2	2007	Finom felbontású	8
AMD Ryzen	2017	Szimultán	2

VALÓDI MULTIPROCESSZOROS RENDSZEREK

- Teljes értékű processzorokból álló rendszerek
- Miért is jó ez nekünk?
  - Az utasításszintű párhuzamosság lehetőségei korlátozottak
    Explicit megközelítéssel nagyobb léptékű párhuzamosítás érhető el.
  - 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:

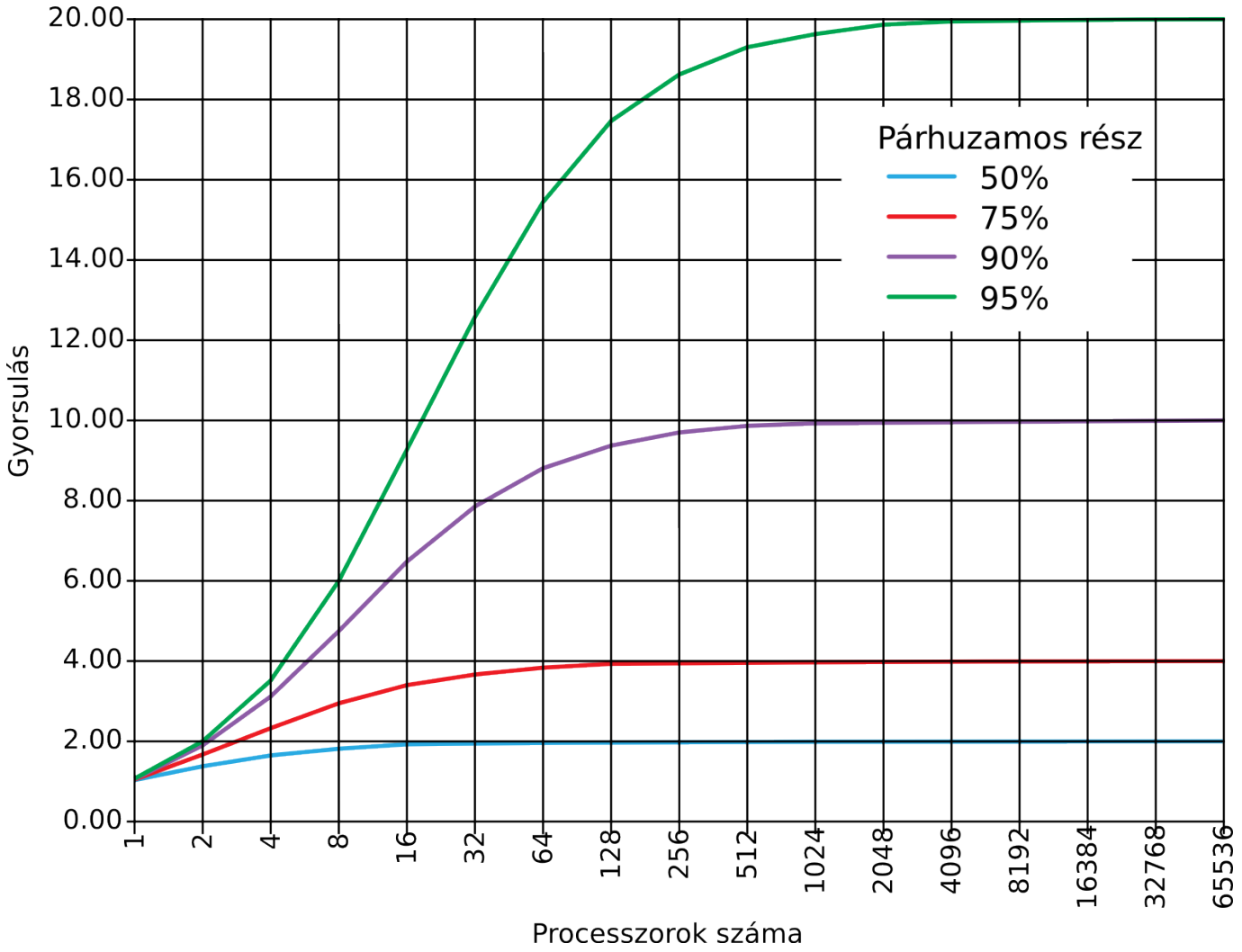
$$S_P(N) = \frac{1}{(1-P) + P/N}$$

- 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

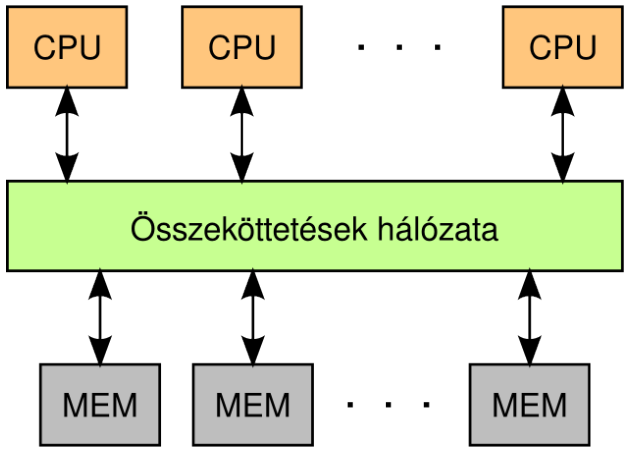
MULTIPROC. ÜZENETKÜLDÉSSEL

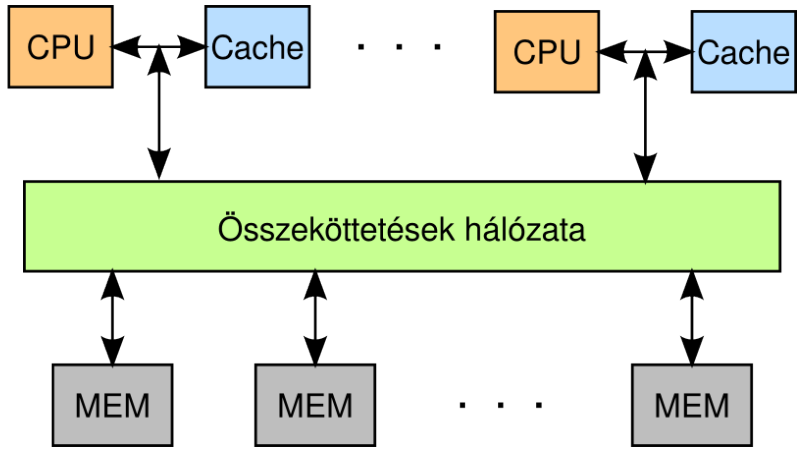
- Csomópontok: teljes értékű számítógépek, külön címtartománnyal
- Kommunikáció: üzenetekkel
  - Explicit hívások:
    - Üzenet küldése (send)
    - Üzenet fogadása (bevárása, receive)
  - Minden taszknak saját egyedi azonosítója van
  - Üzenet tartalma:
    - Rakomány (payload)
    - Küldő azonosítója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

MULTIPROC. OSZTOTT MEMÓRIÁVAL

- A taszkok kommunikációja nagyon egyszerű

  UMA eset:



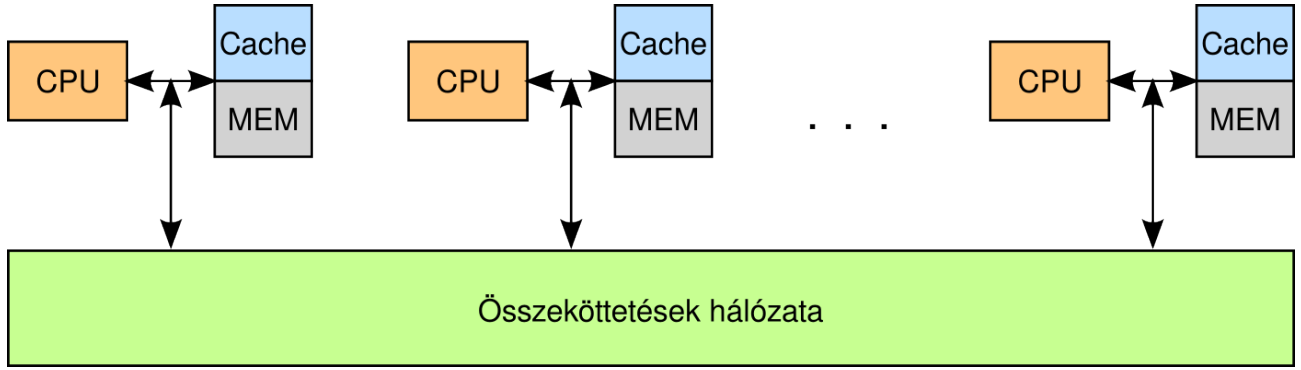


- Egyszerű megvalósítás
- Rossz skálázhatóság:
  - +1 új CPU berakásakor annak teljes memóriaforgalma terheli
- Max. 100 processzorig

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!

OSZTOTT MEMÓRIAKEZELÉS

- Koherencia probléma:

Lépés	Esemény	P1 cache	P2 cache	Memória
1				x
2	P1 olvassa	x		x
3	P2 olvassa	x	x	x
4	P1 módosítja	**x'**	**x**	**x'**

- **A memória koherens, ha a változásokról előbb-utóbb mindenki értesül**
- Speciális cache kezelés kell → **cache koherencia protokoll**
- Legegyszerűbb megoldás:
  - Write-through + write-no-allocate
    + változás esetén „invalidate” üzenet a többieknek!

OSZTOTT MEMÓRIAKEZELÉS

- Memória konzisztencia probléma

P1	P2	P3
(1) X1=1;	(3) X2=1;	(5) X3=1;
(2) Print X2,X3;	(4) Print X1,X3;	(6) Print X1,X2;

- Várt kimenet:
  - **001011** ((1),(2),(3),(4),(5),(6))
  - **101111** ((1),(3),(4),(5),(2),(6))
  - **111111** ((1),(3),(4),(6),(4),(2))
  - ...stb. → **szekvenciális konzisztencia**
- Előforduló kimenet:
  - 000000 ((2),(1),(4),(3),(6),(5))
  - Ha minden proc. soron kívüli végrehajtást csinál
  - Minden processzor önmagában szemantikailag korrekt (precedenciagráf tiszteletben tartva)
    → De az egész rendszer összességében nem!!!
  - Szoftverből kell megoldani

HÁLÓZATI RENDSZEREK
ÉS SZOLGÁLTATÁSOK
TANSZÉK

MŰEGYETEM 1782