ARCHITEKTÚRÁK ALAPJAI 2025/26

Vizsga témák

- gyakori
- kevésbé ritka
- ritka
- nem volt, de fontos
- kihalt/nem fontos

magyarázat

TARTALOMJEGYZÉK

1. AZ UTASÍTÁSSZINTŰ (LOGIKAI) ARCHITEKTÚRA

1.1. ARCHITEKTÚRA

nem volt

1.1.1. Fogalma, implementáció, megvalósítás

Számítógép-architektúra: „A számítógép struktúra, amit a gépi kódú programozónak értenie kell annak érdekében, hogy helyes programot tudjon írni az adott gépre”.

Ide tartoznak: **regiszterek, memória**, **utasítás-készlet, címzési módok**, **utasításkódok**

Implementáció: Az aktuális hardver struktúra *(hogyan vannak a drótok bekötve).*
Megvalósítás: A logikai technológia és a fizikai kapcsolódási pontok *(milyen tranzisztort vagy chipet használtak).*

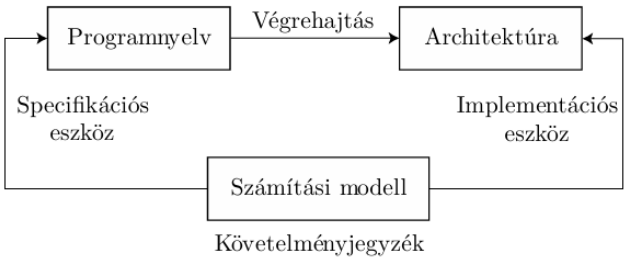
1.2. A SZÁMÍTÁSI MODELL

ritka

Definíció: A számítási modell a számításra vonatkozó alapelvek egy absztrakciója.

A modell három fő kérdésre válaszol (ez FONTOS lesz!):

1. **Min** hatjuk végre a számítást? *(Hardver/Architektúra)*
2. **Hogyan** van kifejezve a számítási feladat? *(Specifikációs)*
3. **Milyen módon** vezérlik a végrehajtási sorrendet? *(Végrehajtás/Implementációs eszköz)*



Ez az ábra a számítógép megtervezésének körforgását mutatja be.

1. **Számítási modell (A TERV):** Ez az alap, a „kívánságlista” (**követelményjegyzék**). Elméletben megmondja, hogyan akarunk számolni (pl. adatvezérelten vagy sorosan).
2. **Programnyelv (A LEÍRÁS):** A modell alapján választunk egy nyelvet (**specifikációs eszköz**), amivel pontosan leírjuk a feladatot a gépnek.
3. **Architektúra (A VÉGREHAJTÓ):** Ez maga a fizikai gép vagy rendszer, ami lefuttatja (**végrehajtás**) a megírt programot.

Az Architektúrát (a hardvert) végső soron úgy kell megépíteni *(implementációs eszköz),* hogy az képes legyen megvalósítani az eredeti **Számítási modellt**.

1.2.1. Csoportosítás

kevésbé ritka

A modellek több szempont szerint is csoportosíthatók.

1. **Számítási modelljük szerint**: „mennyi műveletet hajtunk végre egyszerre”
   - **Szekvenciális**: Egy utasítás fut egyszerre, egymás után lépünk.
   - **Párhuzamos**: Több művelet fut egyszerre, külön végrehajtó egységeken.

2. **Vezérlés meghajtása szerint: „***mitől indul el egy művelet”*
   - **Vezérlés-meghajtott**: A program lépésről lépésre végrehajtja az utasításokat (PC). *Példa: Neumann-modell*
   - **Adat-meghajtott**: Egy művelet akkor indul, ha **minden szükséges bemenete** megjelent. *Példa: Adatfolyam modell*
   - **Igény-meghajtott**: Akkor fut le valami, ha szükség van az eredményére. (lusta kiértékelés)

3. **Probléma leírás szerint**: *„hogyan írjuk le a feladatot programként”*
   - **Procedurális**: Lépéseket írunk le, hogyan számoljuk ki az eredményt. *Példa: Neumann modell, C, Java, Python alapműködése.*
   - **Deklaratív**: Azt írjuk le, hogy *mit* akarunk, nem azt, hogyan számoljuk ki. *Példa: Predikátum-logikai modellek, tudásalapú rendszerek.*

Min hajtjuk végre a számítást?

1. Adatalapú modellek:
   a. Neumann modell
   b. Adatfolyam modell
   c. Applikatív modell
2. Objektum alapú modellek
3. Predikátum logika alapú modellek
4. Tudás alapú modellek
5. Hibrid modellek

1.2.2. Adatalapú modellek

nem volt

Legfontosabb közös tulajdonságuk, hogy az adatok általában **típussal rendelkeznek**, azon belül is **elemi** *(pl.: integer 16 bit)* vagy **összetett** adattípussal. Az adattípus meghatározza az **értelmezési tartományt, értékkészletet** és az **adaton elvégezhető műveletek halmazát**. Ezeket a követelményjegyzékben és a programnyelvben pontosan meg kell határozni.

Minden adatnak típusa van. *Miért fontos ez? Ha egy adat típusa szöveg, akkor nem oszthatod el kettővel. Ha 8 bites szám (0-255), akkor nem írhatod bele az 1000-et.*

gyakori

1.2.3. A Neumann-féle számítási modell *(vezérlés-meghajtott, szekvenciális modell)*

Absztrakciós jellemzők *(procedurális nyelvekre igaz)*

- Változókat hozunk létre (deklarálunk)
- Adatokat manipuláló utasításokat deklarálunk
- Vezérlést átadó utasításokat deklarálhatunk (pl. JUMP)

1. Min hatjuk végre a számítást?

- A számítást **adatokon** hajtjuk végre
- Az **adatokat változók képviselik**.
- Változók értéke korlátlan számban megváltoztatható (**többszörös értékadás**)
- **Adatok és utasítások ugyanabban a memória térben** helyezkednek el
- A számítás **elemi műveletek sorozataként** fogható fel

2. Hogyan van kifejezve a számítási feladat?

- **Adatmanipuláló utasítások** sorozatával
- A változók és az adatmanipuláló utasítások a memóriában helyezkednek el.
- Az adatmanipuláló utasítások **szekvenciálisan megváltoztatják a változók értékét**.
- A végrehajtási sorrendet a **Program Counter (PC)** regiszter biztosítja (tartalmazza a következő utasítás címét)

3. **Milyen módon vezérlik a végrehajtási sorrendet?**

- A PC minden utasítás után inkrementál, de explicit vezérlésátadó utasításokkal (pl. JMP) a sorrend megváltoztatható

Ezért a modell **vezérlés-meghajtott**, **szekvenciális**, **procedurális** és **adat-alapú**

Következmények:

- **Előzmény érzékenység**
- **Szekvenciális végrehajtás**
- **Egyszerűen implementálható**
- **Állapotmódosulások, bizonyos utasítások nem szándékoltan** *(pl túlcsordulás)*
- Mellékhatások is kialakulhatnak *(pl. túlcsordulás: két 16 bites integer szám szorzása)*

1.2.4. Adatfolyam számítási modell *(adatvezérelt, párhuzamos modell)*

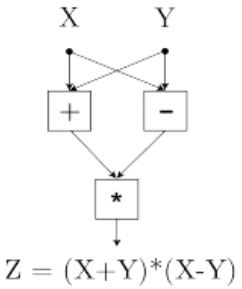
gyakori

1. Min hatjuk végre a számítást?

- A számítást adatokon hajtjuk végre
- Az **adatokat bemenő adathalmaz képviseli**
- Egyszeres értékadás

2. Hogyan van kifejezve a számítási feladat?

- Adatfolyamgráf és input adatok halmaza
- A végrehajtást sok kicsi, **szakosodott műveletvégző egység** végzi
- Párhuzamos végrehajtás lehetséges



A példában Z előállítása Neumann modell esetén 3 lépésben (ADD, SUB, MUL), adatfolyam modell esetén 2 lépésben (ADD és SUB párhuzamosan, MUL) elvégezhető, mely ilyen kis feladat esetén is 33%-os végrehajtási idő csökkenést jelent.

3. Milyen módon vezérlik a végrehajtási sorrendet? *(Stréber modell)*

- **Adatvezérelt**: amint megjelenik az adat, azonnal végrehajtja a műveletet
- Nem vár PC-re mivel nincs *(Nincs előre rögzített utasítássorrend)*
- Az adatok az utasításokban vannak tárolva
- Érzékelni kell az adatok rendelkezésre állását

Következmények:

- Nincs előzményérzékenység
- Párhuzamos végrehajtás
- Nehezebb implementálni

1.2.5. Neumann vs Adatfolyam modell

ritka

	Neumann modell	Adatfolyam modell
Min hajtjuk végre a számítást	adatokon	
Adatokat mik képviselik	változók	bemenő adathalmaz
Értékadás	többszörös	egyszeres
Adattárolás helye	közös operatív tár	utasításon belül (regiszterek)
Számítási feladat leképezése	adatmanipulációs utasítások	adatfolyam gráf
Végrehajtás	szekvenciális	Párhuzamos végrehajtás sok műveletvégző, azonnali műveletvégzés, szakosodott végrehajtó egységek
Végrehajtás vezérlése	vezérlés meghajtott: implicit szekvencia (PC), explicit vezérlésátadás	adatvezérelt: azonnali műveletvégzés amint lehetséges, adatok és utasítások nem rendezettek
Végrehajtás jellege	procedurális	
Következmények	előzményérzékeny	nincs előzményérzékenység
Implementáció	egyszerű	nehézkesebb (magasabb kommunikációs és szinkronizációs igény)

1.3. LOGIKAI ARCHITEKTÚRA

Logikai architektúra: A modell és a programozó által látott specifikáció. $\{M, S\}_L$

Számítógép szintű: *Azt vizsgáljuk, hogy adott bemenetekre milyen eredményt produkál. Központi eleme az operációs rendszer.*

Processzor szintű: A programozó feladata, hogy olyan bementet adjon amire az elvárt kimenet kapható. A processzornak van egy utasításkészlete (ISA). Minden programnyelven írt utasítás lefordítódik a processzor utasításkészletére és az alapján hajtódik végre.

Komponensei:
- Adattér
- Adatmanipulációs fa
- Állapottér
- Állapotműveletek

kevésbé ritka (processzor szintű logikai + fizikai és rendszerszintű fizikai)

1.3.1. Adattér (memória tér és regiszter tér)

kevésbé ritka (memória tér vs regisztertér)

Def: Minden olyan tároló, amelynek tartalmát a CPU **közvetlenül manipulálni tudja**.

Felbontás:
- **Memóriatér** *nagyobb, lassabb, külső lapkán, olcsóbb*
- **Regisztertér** *kisebb gyorsabb, CPU lapkán, drágább*

1.3.2. Memóriatér

Minél több a memória, annál több adatot tudunk a processzor közelében tárolni.

Címtér: A memória eléréséhez szükséges címek halmaza. A címbusz mérete meghatározza a memóriatér maximális méretét. Két féle címtért különböztetünk meg: Elméleti címtér Valós címtér

Virtuális és Fizikai memóriatér

gyakori

Memória tér ketté vált:
- A programozó által látott virtuális memória
- A CPU által látott fizikai memória

Két külön folyamat:
1. **Virtuális memória címek → Fizikai címekké alakítása:** Transzparens egyirányú folyamat, program futása során dinamikusan (AGU vagy MMU végzi)
2. **Nem használt adatok valós memóriából → virtuális memóriába**: Transzparens kétirányú folyamat, az éppen nem használt adatokat mozgatja, szükség esetén vissza.

Virtuális memória: nagyobb, lassabb, háttértáron, programozó látja, itt várnak az adatok
Fizikai memória: kisebb, gyorsabb, külön lapkán, CPU látja, itt fut a program

1.3.3. Regiszter tér

Az adattér nagy teljesítményű, kis része. Nem része a címtérnek (vagy saját címtere van, vagy huzalozott).

nem volt még

1. **Egyszerű regiszterkészletek**
   - Egyetlen regiszter *pl: akkumulátor regiszter – ALU rész eredményeit tárolja*
   - Dedikált adatregiszterek *pl: forrásregiszter, célregiszter*
   - Univerzális regiszterkészlet *gyakran ismételt változók regiszterben maradtak*
   - Stack regiszter:
     - Előnye: nem kell címezni és egyszerű utasítás -> nagyon gyors
     - Hátránya: operandus kiolvasás csak szekvenciálisan

nem volt még

2. **Adattípusonként elkülönített regiszterkészletek**
   - Külön **egyszerű regiszterkészletek külön adattípusokra** (FX, FP, SIMD)
   - **Párhuzamos végrehajtás** lehetősége több műveletvégző esetén

3. **Többszörös regiszterkészlet**

ritka

Legfejlettebb regiszterkészlet, célja az egymásba ágyazott eljárások gyorsítása.

Kontextus: regisztertér állapota + állapottér

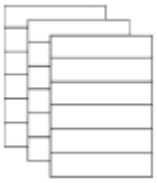
- Új eljárás meghívásakor a régi eljárás kontextusát a memóriába kell menteni → **jelentős teljesítménycsökkenés**.
- A kontextus váltást a kontextus kapcsolók végzik
- Ha a váltás regiszterkészletek között történik → nagyon gyors

Cél: minden kontextushoz külön regiszterkészlet biztosítása.
Szükséges: Egy általános regiszterkészlet, amely a regiszterkészletek közötti kommunikációt biztosítja.

Típusok:

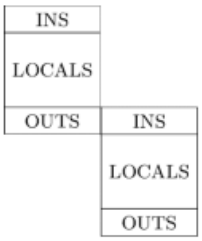
gyakori, általában legfejlettebb jellemzése

a, Több egymástól független regiszterkészlet



Nincs átfedés köztük.
Paraméterátadás csak az operatív táron (memórián) keresztül.

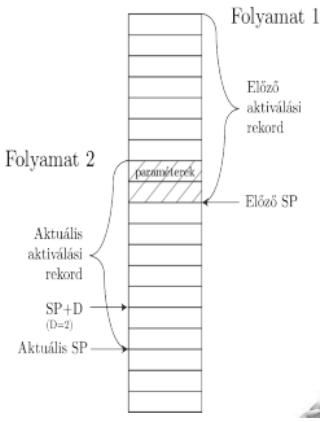
b, Átfedő regiszterkészlet (INS-LOCALS-OUTS)



Egymásba ágyazott eljárásokhoz fejlesztették
Három féle paraméterre oszlik: **INS**(bemenő), **LOCALS** (lokális), **OUTS** (kimenő)
Az OUTS és a következő eljárás INS része azonos címen helyezkedik el
Paraméterátadás egyszerű és gyors

Hátránya: merev felépítés és lehetséges túlcsordulás

c, Stack cache



Stack és közvetlen elérésű cache kombinációja.
A compiler a **stack pointer (SP)** segítségével jelöl ki egy pontot a regisztertérben
Ettől a ponttól meghatározott számú regisztert allokál → **aktiválási rekord**

- A SP az aktiválási rekord első elemére mutat
- Displacement (eltolás) segítségével bármely regiszter közvetlenül címezhető
- Aktiválási rekord hossza rugalmas
- Nincsenek üres helyek
- Nincs túlcsordulás

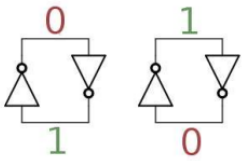
Egymást követő eljárások aktiválási rekordjai átfedik egymást
(Input–output paraméterátadás megoldott)

Minden folyamatnak a compiler pontosan úgy fogja lefoglalni az aktiválási rekordját, hogy az átfedő regiszterekben benne legyenek az átadandó paraméterek.

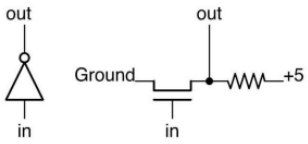
volt már 1x
1 bites fizikai regiszter

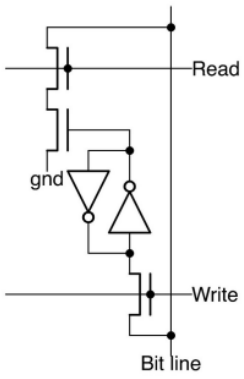
1.3.4. Regiszter fizikai kialakítása

Inverter pár: ez tárolja a bitet, statikus tárolás



- in = 0 → tranzisztor zár → az ellenállás felhúzza az out-ot out = 1
- in = 1 → tranzisztor nyit → out leföldelődik out = 0





Regiszter cella felépítése (3 tranzisztor):

- **Read**: rákapcsolja a kimenetre
- **Write**: a bit line értékére módosítja a tárolt bitet
- **Erősítő tranzisztor:** Az inverter pár jele gyenge, nem módosítja a tárolt bitet, megakadályozza, hogy az olvasás visszahatva átírja a memóriát.

1.3.5. Adatmanipulációs fa

kihalt, nem fontos

1. Adattípusok: leírja, hogy milyen adattípusokat értelmezünk az adott arcgitektúrán
2. Műveletek: megmutatja, hogy a adott adattípusokon milyeen műveletek értelmezhetőek
3. Operandus típusai: megkülönböztetjük az operandusokat számuk és típusaik szerint
4. Címzési módok: engedélyezett címzések
5. Gépi kód: minden architektúrán más

FX: fixpontos
FP: lebegőpontos
r: regiszter
m: memória
D: eltolás
PC: program számláló
RI: indexregiszter

1.3.6. Adattípusok

nem volt még

Elemi adattípusok: numerikus, karakteres, logikai, pixel

Összetett adattípusok:

- Különböző típusú részekből áll: rekordok
- Azonos típusokból áll: tömb, vektor, szöveg, verem, sor (FIFO), lista, fa, halmaz

Numerikus elemi adattípusok:

1. **FX** (fixpontos)
2. **FP** (lebegőpontos)
3. **BCD** (binárisan kódolt decimális)

Karakteres elemi adattípusok: ASCII, UNICODE

1.3.7. Főbb CPU regiszterek

régen: CPU rajza kellett, de ezek fontosak

Memory Address Register **(MAR):** *azt a memóriacímet tartja, ahol a CPU olvasni vagy írni akar.*
Memory Data Register **(MDR):** *a konkrét adatot tartja, ami a memória és a CPU között mozog.*
Program Counter **(PC):** *a következő végrehajtandó utasítás címét tartja; minden lépésnél nő vagy ugrik.*
Instruction Register **(IR):** *az éppen végrehajtott utasítást tartja, amit már kiolvastak a memóriából*
Általános célú regiszterek: *gyors ideiglenes tárolók, operandusoknak, címeknek, rész-eredményeknek.*
Dekóder **(DEC):** *az IR-ben lévő utasítást belső vezérlőjelekké fordítja, hogy a hardver tudja mit kell csinálni*
Vezérlő: *a CPU „karmestere"; időzít, engedélyez, kapcsol, meghatározza a műveletek sorrendjét*
Aritmetical Logical Unit **(ALU):** *műveletek helye (összeadás, kivonás, AND, OR, stb.).*
Accumulator Register **(AC):** *az ALU **fő munkaregisztere**, ide kerülnek a műveletek eredményei.*

1.3.8. Utasítás lehívása, végrehajtása és tárolása

ritka (lehívás + tárolás)

1. **FETCH (utasítás lehívása)**:
   MAR ← PC
   MDR ← [MAR]
   IR ← MDR
   PC ← PC + 1

   PC tartalmazza a következő utasítás memóriacímét
   → tartalma átkerül a **MAR**-ba
   → innen memóriából kiolvassa az utasítást és **MDR**-be helyezi
   → MDR tartalma átkerül az **IR**-be
   → PC automatikusan növekszik és a következő utasítás címére mutat

2. **Execution 1-2**:
   DEC ← IR
   MAR ← $DEC_{címrész}$
   MDR ← [MAR]
   AC ← MDR (Execution 2: AC ← AC+MDR)

   Vezérlőegység dekódolja az IR tartalmát
   → dekódolja az IR tartalmát
   → előkészíti az operandusokat
   → ALU végrehajtja az utasítást

3. **STORE**:
   DEC ← IR
   MAR ← $DEC_{címrész}$
   MDR ← AC
   [MAR] ← MDR

   Memóriában tárolás esetén
   → tárolási utasítás dekódolása
   → dekódolt címrész betöltése MAR-ba
   → alu eredményének betöltése MDR-be
   → MDR tartalma bekerül a MAR által megadott címre

Gépi kódú utasítás: A számítógép által végrehajtható alapvető feladatok ellátására szolgáló elemi művelet leírása. A processzor csak gépi kódú utasításokat tud értelmezni.

- **Műveleti kód (MK):** tartalmazza, hogy MIT kell csinálni (utasítás mező)
- **Címrész:** megmondja, hogy hol található (operandus mező)

MK	Címrész

gyakori (gépi kódú utasítás részei)

1.3.9. Utasítás típusok (0, 1, 2, 3, 4 címes)

kevésbé ritka (főleg 2 és 3 címes)

opd := ops1 @ ops2 d: cél operandus s: forrás operandus

4 címes utasítás: *d, ops1, ops2, következő utasítás címe*

- **Hátrány**: nehézkes, merev struktúra

3 címes utasítás: *d, ops1, ops2*

- PC tárolja a következő utasítás címét
- **Előny**: egyik forrás operandus sem íródik felül, párhuzamosítás lehetséges
- **Hátrány**: még mindig hosszú utasítások, sok memóriát igényel

2 címes utasítás: ops1, ops2

- Az eredmény felülírja ops1-et: (ops1 = ops1 @ ops2)
- Mai CISC architektúrákban alkalmazzák

1 címes utasítás:

- Operandus betöltése az AC-be, művelet az AC tartalmával
- Pl: LOAD[100], ADD[101]
- **Előny**: rövidebb utasítások
- **Hátrány**: több utasítás szükséges

0 címes utasítás:

- Nincs operandus (pl. NOP, CLEAR D, RST)
- Előny: nagyon rövid, gyors
- Hátrány: növeli az utasításkészletet

nem volt még, legfontosabbak előny, hátrány

Operandus típusok

- AC akkumulátor (E: gyors H: csak egy van belőle)
- memória (m) (E: nagy méret H: hosszú címe van, lassú)
- regiszter (r) (E: gyors H: kevés van belőle)
- verem (s) (E: gyors H: csak a tetejét látjuk)
- immediate (i) Közvetlen érték Pl: ADD r1, 3
  (E: gyors, nincs szükség regiszterre H: csak programból változtatható)

Címzési mód

nem volt még, annyira nem fontos

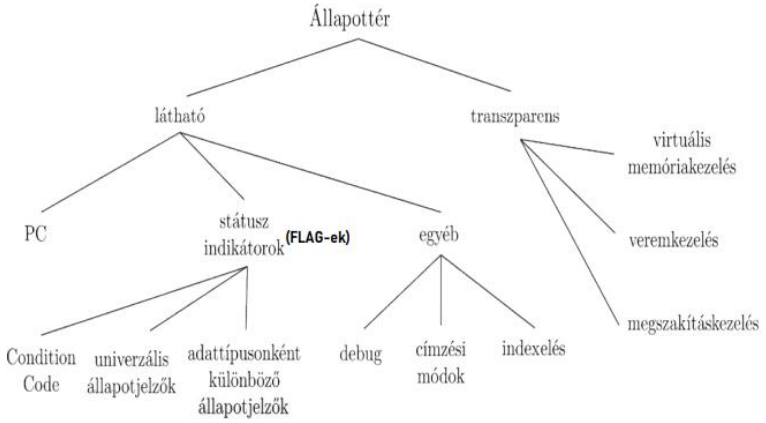
Címszámítás:

- **Abszolút cím**: pontosan megmondja melyik címet használjuk (teljes cím)
- **Relatív címzés**: bázis cím (S) + eltolási cím (D)

gyakori + ágrajz

1.3.10. Állapottér

Def: Olyan programból látható és nem látható (transzparens) tárolókból áll, melyek az adott programra vonatkozó állapotinformációkat hordozzák.



A **flagek** nem adatot, hanem **állapotot** tárolnak, és speciális utasításokkal menthetők, beállíthatók vagy törölhetők. Minden architektúrának saját flag-készlete van. *kihalt*

A **transzparens állapottér** olyan mechanizmusokból áll, amelyeket a program nem kezel közvetlenül, mégis hatnak a végrehajtásra, például a virtuális memóriakezelés, a veremkezelés és a megszakításkezelés.

gyakori

1.3.11. Állapotműveletek

PC állapotműveletek:

- inkrementálás
- dekrementálás
- felülírás (utasításból vett címmel)

Flagek állapotműveletei:

- Save
- Set
- Reset
- Load
- Clear

2. AZ ARITMETIKAI EGYSÉGEK FELÉPÍTÉSE

Fizikai architektúra: A modell és az implementáció együttes leírása. $\{M, I\}_L$

Számítógép szintű: processzor, memória, buszrendszer

Processzor szintű:

- **műveletvégző egység**
- **vezérlő**
- I/O egység
- megszakítás rendszer

gyakori

Műveletvégző (ALU) részei:

- Regiszterek
- Adatutak
- Kapcsolópontok
- Szűkebb értelemben vett ALU

gyakori

Két féle CPU létezik:

	Szinkron CPU	**Aszinkron CPU**
Leírás	Órajel generátorral működik	Egy utasítás befejezése után közvetlen indul a következő
Előny	Egyszerű. gyors	Nincs holt idő
Hátrány	A következő órajelet mindig meg kell várni, így holtidő keletkezhet	Speciális áramkör kell az utasítás befejezésének érzékelésére

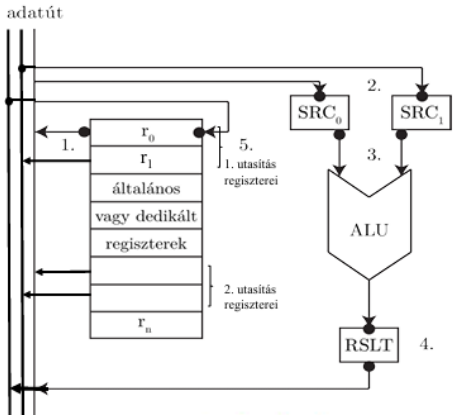
Regiszterek:

nem volt még

- **Látható (programozó által elérhető):**
  - Univerzális regiszterek: tetszőleges adat tárolható bennük, méretük korlátozott
  - Dedikált regiszterek: speciális célra, pl. veremmutató (stack pointer)
- **Rejtett (nem hivatkozható)**
  - Pufferregiszterek, adatfeldolgozást segítik
  - Alacsony szintű programozásnál figyelembe kell venni őket

gyakori

Adatutak (NEM BUSZ): Adatbuszon értelmezett a címzés, adatúton nem. A **processzor belső egységeit köti össze** (regiszterek, pufferregiszterek, ALU). **Vezetékrendszerként** értelmezhető. Egy adatúton **egyszerre csak egy adat** haladhat.



Kapcsolópontok:

A regiszterek bemenetén és kimenetén elhelyezkedő **tranzisztorok**

- Állapotukat a vezérlőegység határozza meg
- **Kimeneti kapcsoló** háromállapotú (1, 0, zárt)
- **Bemeneti kapcsoló** kétállapotú (nyitott, zárt)

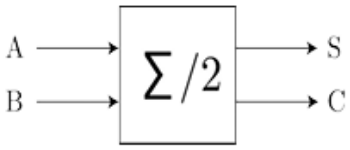
2.1. ÖSSZEADÓK

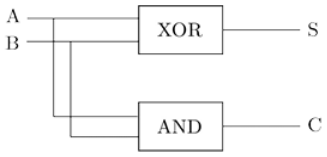
ritka

2.1.1. Egybites fél összeadó

S = $\bar{A}B + A\bar{B}$ = A XOR B
C = AB

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1





gyakori + előny/hátrány

2.1.2. Egybites teljes összeadó

$S = \overline{AB}C_{in} + A\overline{BC_{in}} + \bar{A}B\bar{C}_{in} + ABC_{in}$

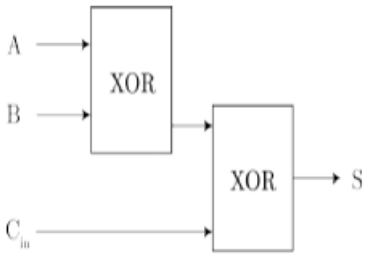
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Egyszerűsítés (C_{in} kiemelése és X = A XOR B):

$S = (\overline{AB} + AB)C_{in} + (A\bar{B} + \bar{A}B)\overline{C_{in}}$

$S = \bar{X}C_{in} + X\overline{C_{in}}$ = X XOR C_{in}

S = A XOR B XOR C_{in}



Carry out kiszámítása:

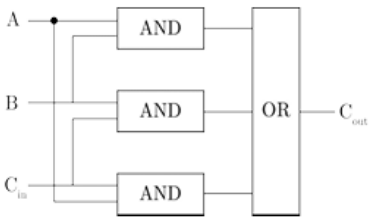
$C_{out} = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$

Egyszerűsítés (ABC-ből többet is bele rakhatunk, eredmény nem változik, de kiemelhetünk, $A + \bar{A} = 1$):

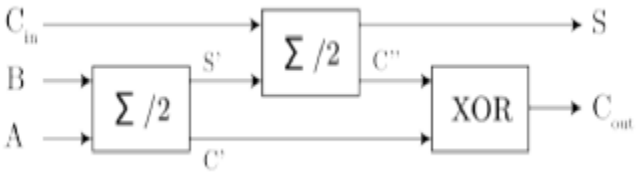
$C_{out} = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$ + ABC + ABC

$C_{out} = (A + \bar{A})BC_{in} + (B + \bar{B})AC_{in} + (C_{in} + \overline{C_{in}})AB$

C_{out} = BC_{in} + AC_{in} +AB



Megvalósítása félösszeadókkal:



gyakori + előny/hátrány

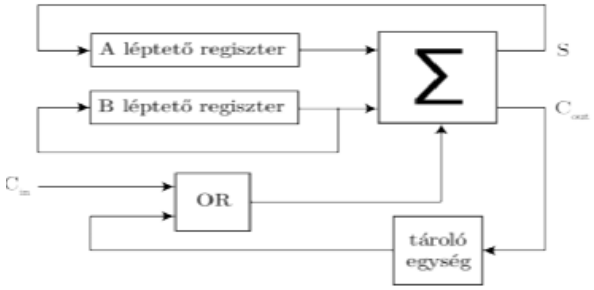
2.1.3. N bites soros összeadó

Míg az előző két összeadó csak egyetlen bit összeadására képes, a gyakorlatban a CPU nem izolált bitekkel dolgozik, hanem több bites adatokat kezel. Az összeadások tipikusan byte, fél szó, szó vagy dupla szó méretű operandusokon történnek, ezért szükség van több bites összeadás megvalósítására.

Előny: egyszerű, olcsó
Hátrány: lassú

Végrehajtási idő: T = n * t

„n”: bitek száma
„t”: egybites összeadó végrehajtási ideje



Carry-t flip-flop-ok tárolják (tároló egység): feladata a szinkronizálás. (1bites késleltető)

B léptető regiszter esetén mikor a legkisebb helyiértékű bit kikerül a regiszterből egyrészt bekerül az összeadóba, másrészt visszakerül a legnagyobb helyiértékre és minden bit egyel jobbra lép. A végére a B regiszterbe visszakapjuk az eredeti számot, az A regiszterben az eredményt. C_{in} kell ahhoz, hogy teljes összeadó legyen (korábbi eredmény bevezetése), amely csak az első bit esetén létezhet.

gyakori + előny/hátrány

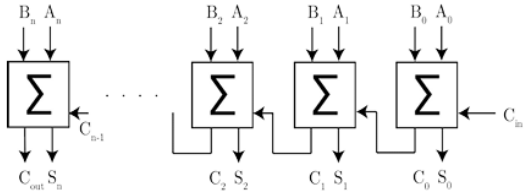
2.1.4. N bites párhuzamos összeadó (Ripple Carry Adder)

A soros megoldással az a probléma, hogy a végrehajtása arányos a bitek számával. A folyamatot fel lehet gyorsítani azzal, hogy soros helyett párhuzamos összeadást használunk.

n bit esetén n db teljes összeadó

Végrehajtási ideje: T ~ n * t

*(közel n*t, carry-k számától függ, mert ha van azt be kell kötni a következő összeadóba =* **hullámzó***)*



Előny: párhuzamos összeadás
Hátrány: hullámzó végrehajtási idő

gyakori „módosított összeadó”+ előny/hátrány
Mi a P és G, C_0-ig levezetni

2.1.5. **CLA** (Carry Look Ahead)

Rekurzív módszer.

Az egybites teljes összeadó esetén már kiszámoltuk a C_{out} értékét, amelyet fel tudunk használni.

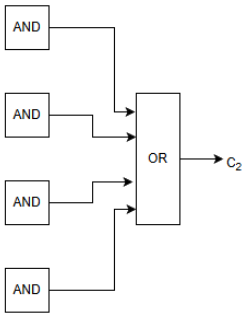
C_{out} = $BC_{in} + AC_{in} + AB$ => AB + (A + B)C_{in}

AB = G (Generate – A és B kapcsolata generálja a Carry-t)
A+B = P (Propagate – A vagy B propagálja a carry-ket)

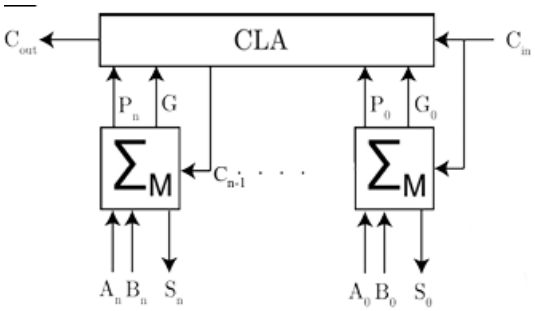
Ennek segítségével meg tudjuk határozni behelyettesítésekkel a Carryket előre.

$C_{out} = G + PC_{in}$

$C_0 = G_0 + P_0C_{in}$
$C_1 = G_1 + P_1C_0 = G_1 + P_1(G_0+P_0C_{in}) = G_1 + P_1G_0 + P_1P_0C_{in}$
$C_2 = \ldots. = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_{in}$ ← 4 tagból áll, ami azt jelenti, hogy elég 4 ÉS kapu, amit rákötünk 1 db VAGY kapura, így 2 lépésből megkapjuk a C_2 értékét és csak a Cin kell hozzá, a többi carry nem

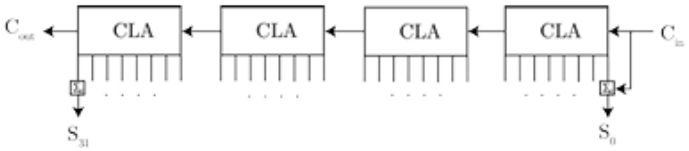


A teljes összeadók carry-jét nem használjuk mert azt a CLA intézi. Az összeadók beküldik a P és G-t. Így pl 8 bit esetén 3 órajel ciklus alatt megvan az összeadás.

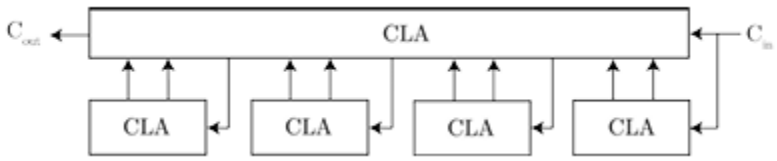


A gond: Egy OR kapu általában max 8 bemenetet kezel.

Pl: 32 bit összeadása esetén 4 db CLA-t kellene egymás után kötni. Így lassul az összeadás sebessége.



Megoldás: CLA-knak CLA



CLA CLA-ja nem volt még

2.2. FIXPONTOS SZORZÁS

A szorzást is összeadás sorozatára vezetjük vissza.
Szorzás sebességének gyorsításához rendelkezésre álló elemi műveletek:

- összeadás
- invertálás
- léptetés

nem volt még, nem fontos

X=A*B=13*123

0000
39
0039
26 *léptetés*
0299
13 *léptetés*
1599

Léptetéses módszer: létrehozunk egy **gyűjtő regisztert**, amit a művelet megkezdése előtt kinullázunk. A gyűjtőhöz hozzáadjuk a szorzó legkisebb helyiértékétől haladva annyiszor a szorzandót, amennyiszer a szorzó adott helyiértéke kívánja. Majd léptetünk és folytatjuk a szorzó eggyel nagyobb helyiértékével.

A szorzás eredménye általában több bit helyet foglal.

A *m bit*
B *n bit*
A*B ≤ *n*+*m bit*

2.2.1. Gyorsítási lehetőségek (Bitcsoport, Booth algoritmus)

gyakori

Bitcsoporttal történő szorzás.

Ahelyett, hogy bitenként hajtjuk végre a szorzást, történhet ez bitcsoportonként:

0111*10|01

0000 *gyűjtő*
01,11 *egyszerese és léptetés kétszer*
01,11
1,11000 *kétszerese és léptetés kétszer*
1,11111

00: léptetünk kettőt
01: hozzáadjuk az egyszeresét és léptetünk kettőt
10: hozzáadjuk a kétszeresét, majd léptetünk kettőt
11: négyszeresét adjuk hozzá és kivonjuk az egyszeresét és léptetünk kettőt

2-vel való szorzás (10) olyan mint 10-es számrendszerben 10-el → végére írunk egy nullát.

Booth algoritmus

Ha a szorzóban sok 1-es van, akkor lassú a szorzás, mert sok összeadást kell végezni.
X*62 (111110) esetén a szorzást több lépcsőben kellene elvégezni.

Keressük meg a szorzóhoz legközelebbi számot és állítsuk elő az eredményt úgy, hogy:

X*(64-2) = X*64 – X*2

64 (1000000) → 1 db összeadás
2 (10) → 1 db összeadás
kivonás → 1 db összeadás

5 db összeadás helyett csak 3

2.3. LEBEGŐPONTOS SZÁMOK

A fixpontos számok (FX) értelmezési tartománya viszonylag kicsi, pontosság sem túl jó.

Például 16 bit esetén -32768 →32767

Jellemzői:

FP = M * r^k *M: mantissza, **r**: radix (számrendszer alapja), **k**: karakterisztika*

Elvárás, hogy a radix egyezzen meg a mantisszánál használt számrendszer alapjával!

Normalizált formátum: az első értékes jegy a tizedespont után van és a karakterisztikát ennek megfelelően adjuk meg.

$0{,}001*2^k \rightarrow 0{,}1*2^{k-2}$

Mantissza értéke (általában 1/r):

- 10-es számrendszerben: $0{,}1 \leq M < 1$
- 2-es számrendszerben: $½ \leq M < 1$

2.3.1. Értelmezési tartomány:

gyakori ÉT + számegyenes

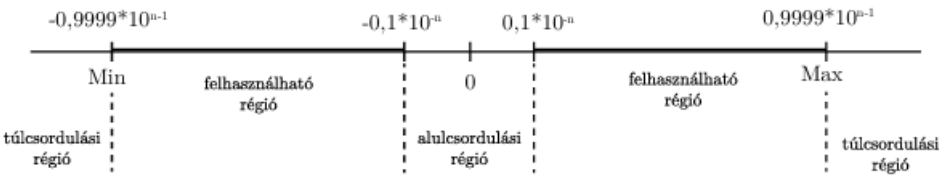
függ: 1. a karakterisztika számára rendelkezésre álló bitek számától és 2. a radixtól

Karakterisztika bitek száma	Legnagyobb érték$_{10}$	Értelmezési tartomány$_{10}$	Legnagyobb érték$_2$	Értelmezési tartomány$_2$
1	±9	$10^{\pm 9}$	1=1	$2^{\pm 1}$
2	±99	$10^{\pm 99}$	11=3	$2^{\pm 3}$
3	±999	$10^{\pm 999}$	111=7	$2^{\pm 7}$
4	±9999	$10^{\pm 9999}$	1111=15	$2^{\pm 15}$ (=FX16 32768)

2.3.2. Pontosság:

Mantissza bitjeinek számától függ.
Például, ha 3 bit a mantissza, akkor a $0{,}3014*10^6$ esetén nem tudjuk az egészet ábrázolni.

10-es számrendszer 4 mantissza bit esetén:



Az architektúrának biztosítania kell a túlcsordulás és alulcsordulás felfedezését, jelzését és kezelését

gyakori

Túlcsordulás esetén:

- Kijelzi és beállítja a legnagyobb megengedett értéket vagy
- Előjeles végtelent jelez ki (minden bitje 1-es)

Alulcsordulás esetén:

- kijelzi és 0-ra konvergál, vagy
- a denormalizált számot jelzi ki (nem normalizál)

Ha a mantissza 0, akkor a karakterisztika is 0 legyen.

2.3.3. Pontosság növelése (Rejtett bit, örző bit)

gyakori

1. **Rejtett bit használata:** Mivel normalizált formátumban az első értékes bit van a tizedespont után és az minden esetben 1-es, így tárolásnál annak nincs információ tartalma. *Ha a mantissza 23 bit, akkor 24 bitet tudunk eltárolni → nő a pontosság*
   0,1|01
   0,1|110
   0,1|0001

2. **Örző bitek**: A relatív hiba kisebb legyen, mint a normalizált eredmény legkisebb számjegye (utolsó)!

   4 biten: 0,0001|111 ezek levágásra kerülnek és értékes biteket vesztünk.

   Megoldás: A CPU-n belül a regiszterek több biten tudják tárolni a mantisszát. (+3-15 bit)

   Felhasználás:
   - A rejtett bit balra léptetésekor 1 értékes bitet tudunk beléptetni
   - Tárolási formátum kérésekor kerekített értéket tárolhatunk
     pl: 0,01000|1111 -> 0,01001
   - Normalizáláskor értékes biteket tudunk felhasználni (elején sok 0 van akkor több értékes bitet be tudunk tolni)
     pl: 0,000101|111 -> 0,101111

2.3.4. Lebegőpontos számok kódolása

gyakori

- **Mantissza kódolása:** kettes komplemens
- **Karakterisztika kódolása**: többletes kódolással

Miért?

Karakterisztika esetén a többletes kód kialakítása gyorsabb, de mantisszánál azért nem használjuk, mert csak alap műveletet lehet elvégezni (+,-).

2.3.5. IEEE754 szabvány

gyakori

Célja megkönnyíteni a különböző CPU-k esetén az adatszintű kompatibilitást, portabilitást.
Rendszerszintű megoldás, vagyis a hardvernek és a szoftvernek együtt kell biztosítania a szabványnak való megfelelést.

Adattípusok:

- egyszeres (32 bit)
- kétszeres (64 bit)
- kiterjesztett (80 bit)
- négyszeres (128 bit)

Formátumok:

1. Szabványos: háttértáron való tároláshoz kötött
   a. egyszeres pontosságú (32 bit, kisebb, gyorsabb, pontatlanabb)

1 előjel bit	8 bit karakterisztika	23 bit mantissza

   b. kétszeres pontosságú (64 bit, nagyobb, lassabb, jóval pontosabb)

1 előjel bit	11 bit karakterisztika	52 bit mantissza

2. Bővített: CPU-n belül, nagyobb szabadság
   a. egyszeres pontosságú (min 43 bit)
   b. kétszeres pontosságú (min 79 bit)

Kerekítések: *(tárolás esetén bővítettből át kell alakítani szabványos formátumba)*

- Legközelebbire való kerekítés. *nem tudjuk, hogy felfele vagy lefele kerekítettünk*
- 0-ra kerekítés: örző bitek levágása, *M ≤ pontos érték -> mindig lefele kerekítünk*
- Kerekítés pozitív végtelen felé
- Kerekítés negatív végtelen felé

Kivételkezelés: felbukkanásuk általában megszakítást eredményez.
Példák: túlcsordulás, alulcsordulás, 0-val való osztás, gyökvonás negatív számból… stb.

2.3.6. Dedikált Lebegőpontos Műveletvégző

gyakori + rajz

Párhuzamos műveletvégzés: mantissza és a karakterisztika számítása párhuzamosan is végezhető

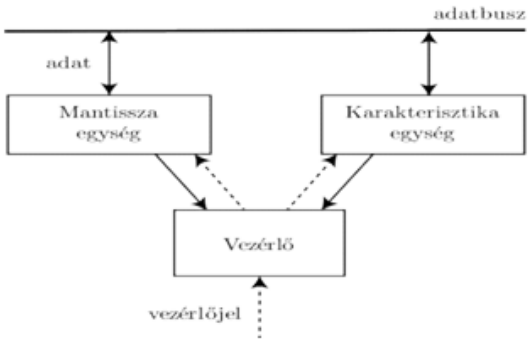
$$A = \pm m_a \times r^{ka}$$

$$B = \pm m_b \times r^{kb}$$

A×B:

- $m_a \times m_b$
- $k_a + k_b$

} párhuzamosan végezhető



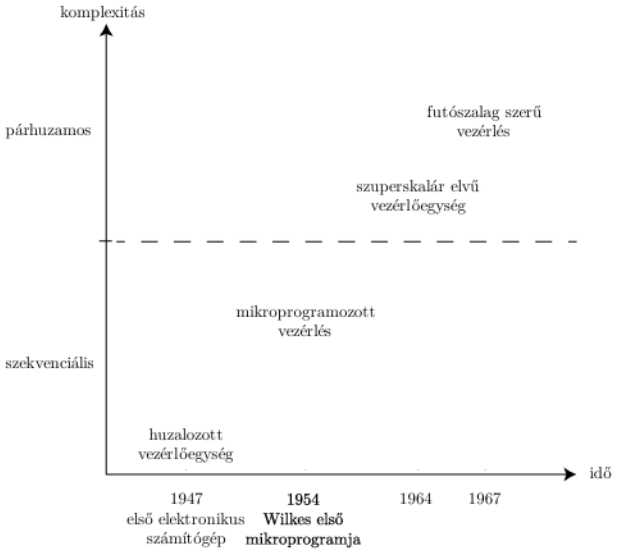
3. VEZÉRLŐEGYSÉG

kihalt

A processzorban ez az egyik legbonyolultabb áramkör. A benne található **ütemező** felelős azért, hogy úgy állítsa elő a vezérlőjeleket, hogy azok szinkronban legyenek az órajellel. Belső megvalósítása általában titkos.

Fejlődése:

- **Szekvenciális**
  - centralizált (egy vezérlő van)
  - huzalozott vagy mikroprogramozott
- **Párhuzamos**
  - decentralizált (több vezérlő van)
  - hibrid vezérlők
    (tartalmaznak huzalozott és mikroprogramozott vezérlőrészeket is)



Két vezérlési elv van:

1. Huzalozott vezérlő
2. Mikroprogramozott vezérlő

kihalt

3.1.1. Huzalozott vezérlő

Az utasításvégrehajtását fix logikai áramkörök valósítják meg (igazságtáblák, logikai függvények).

Előnye: nagy sebesség
Hátránya: merevség, nehéz módosíthatóság

kihalt

3.1.2. Mikroprogramozott vezérlő

Az utasításvégrehajtást egy mikroprogram írja le, amely mikroutasításokon keresztül ad ki vezérlőjeleket.

Előnye: rugalmasság és áttekinthetőség
Hátránya: lassabb, mint a huzalozott vezérlés

4. BUSZRENDSZEREK

A buszrendszer a számítógép egységei közti kommunikáció infrastruktúrája (CPU, RAM, perifériák). A felhasználó számára transzparens, a működés egységes és szervezett. Az egységek kizárólag a buszon keresztül kommunikálnak. Teljesítménykritikus elem: a leggyorsabb eszköz is lelassul, ha a busz szűk keresztmetszet.

Egy időben egyszerre több eszköz csatlakozik a rendszerre → kezelni kell:

- az adatátvitel irányát
- az aktív eszközök kijelölését
- az időzítést és szinkronizációt

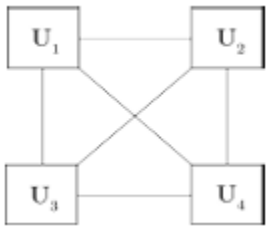
Ezt a **szabványosítás** biztosítja (jelek, vezetékek, csatlakozók) → eszközök könnyen cserélhetők.

Példák buszokra: SATA, USB, PCIe

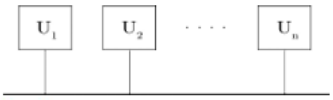
4.1. CSOPORTOSÍTÁS

kevésbé ritka

1. **Az átvitel iránya szerint**
    a. **szimplex:** csak egy irányú adatátvitel (pl.: címbusz, órajel - CLK, reset - RST)
    b. **félduplex:** két irány, de egyszerre csak az egyik aktív
    c. **full-duplex:** két irány egy időben (pl.: PCIe)



2. **Átvitel jellege szerint**
    a. **Dedikált busz:** minden egység mindegyikkel össze van kötve (pl.: Intel QPI)
    Előny: gyors, közvetlen kommunikáció
    Hátrány: merev, nehezen bővíthető
    b. **Shared (megosztott):** Minden egység közös vezetéken kommunikál. Buszvezérlő utasításokra van szükség az ütközések elkerülésére.
    Előny: olcsó, egyszerű megvalósítás, könnyen bővíthető
    Hátrány: lassabb, vezérlése bonyolult, hiba több eszközt érinthet



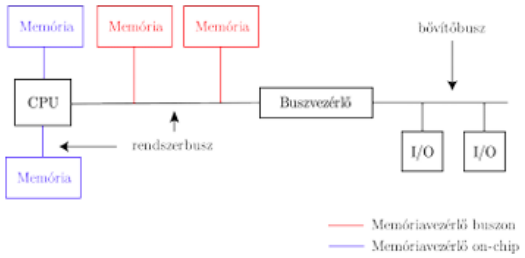
3. **Átvitt tartalom szerint**
    a. **Címbusz:** eszközök és memória területek címzésére szolgál
    Modern rendszerekben blokkos címzés: csak kezdő címet adunk meg
    b. **Adatbusz:** Adatok szállítása CPU-ba memória és perifériákból meg vissza.
    Sávszélessége nőtt az idők során (8, 16, 32, 64 bit ...)
    Lehet közös a vezeték a címbusszal → időbeli multiplexeléssel (vezérlőjelekkel)

c. **Vezérlővonalak:** a rendszerben az időzítő jelek és az egység állapotáról szóló információ átvitelére szolgálnak

- **Adatátvitelt vezérlő jelek:**
  - **M/"I/O":** memória vagy periféria van címezve
  - **R/W:** adat iránya CPU felől nézve
  - **WD/B:** adat mérete
  - **D/S:** adat felhelyezését jelzi a memória számára
  - **A/S:** cím felhelyezését jelzi a memória számára
  - **RDY:** átvitel befejezése / busz szabad
- **Megszakítást vezérlő jelek:**
  - Megszakítást kérő jel
  - Megszakítást visszaigazoló jel
- Buszvezérlő jelek
  - Buszkérés, buszfoglalás, visszaigazolás
- Egyéb vagy speciális vezérlőjelek
  - CLK
  - Reset

4. **Összekapcsolt területek alapján**

   a. **Rendszerbusz:** címbusz + adatbusz
   b. **Bővítőbusz:** pl PCI, PCIe, USB



5. **Átvitel módja szerint**

   a. **Párhuzamos:** több bit egyszerre (memória, régi PCI)
   b. **Soros:** nagy órajel, kevesebb vezeték (USB, SATA, PCIe)

4.2. PCI SZABVÁNY (PÁRHUZAMOS BUSZ)

gyakori PCI vs PCIe

- Párhuzamos, megosztott buszrendszer.
- Minden csatlakoztatott eszköz azonos címbuszt, adatbuszt és vezérlővonalakat használ

A busz specifikációja egységesen definiálja:

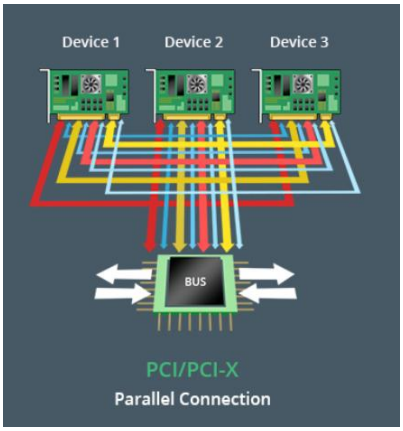
- fizikai méreteket (aljzat, érintkezők),
- elektromos jellemzőket,
- adatátviteli időzítést,
- kommunikációs protokollokat.

Működési sajátosság:

- A PCI-eszközöket a CPU úgy látja, mintha közvetlenül a rendszerbuszra lennének kötve.
- Az eszközök a CPU által látott címtérből kapnak memóriacímeket (memory-mapped I/O).
- Több buszmester esetén busz-arbitrálás (buszfoglalás) szükséges.
- Egy időben csak egyetlen master végezhet adatátvitelt.

Előnye:

- Gyorsabb adatátvitel (mondhatni közelebb kerülnek a perifériák a CPU-hoz)
- Viszonylag olcsó megoldás
- Egyszerű felépítés, széles körben elterjedt
- Egységes, jól dokumentált szabvány



Hátrányok:

- Megosztott busz → ütközésveszély, arbitrálás miatti késleltetés
- Párhuzamos átvitel → órajel-növelés fizikailag korlátozott
- Sok eszköznél jelentősen lassul

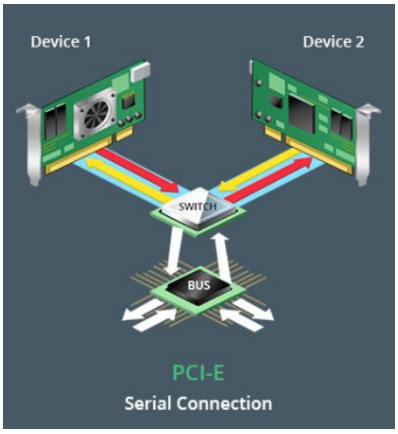
4.3. PCI EXPRESS SZABVÁNY (SOROS BUSZ)

- Soros, pont-pont topológiájú buszrendszer
- Nem megosztott busz: minden eszköz külön összeköttetést kap a vezérlő felé.

Működési sajátosság:

gyakori PCI vs PCIe

- Különálló vonalakból áll (külön küldés és fogadás)
- Full-duplex működés
- Buszprotokoll: csomagokba ágyazott adatátvitel
- Többféle szélesség: *x1, x4, x8, x16, x32*
  (szélesség a párhuzamosan használt soros sávok számát jelenti.)
- Kisebb sávszámú kártya nagyobb aljzatba illeszthető.



Előnyök:

- Nagyon nagy adatátviteli sebesség
- Kevesebb érintkező – jobb jelminőség
- Full- duplex, párhuzamos kommunikáció több eszközzel
- Hot-plug támogatás

Hátránya:

- Összetett protokoll → nagyobb vezérlési overhead
- Teljesítménye erősebben függ a CPU-tól
  mivel a CPU-nak kell kezelni a csomagkapcsolt üzeneteket is
- Drágább implementáció, mint a régi PCI

gyakori PCI vs PCIe

4.4. PCI vs PCIe

PCIe	PCI
pont-pont topológia, soros adatátvitel	megosztott párhuzamos architektúrát használ minden eszköz
különálló vonalak kapcsolják az eszközöket a buszvezérlőhöz	minden eszköz közös cím- adat- és vezérlővonalat használ
Full duplex, bármely két végpont között	több master esetén arbitrálás (buszfoglalás) történik
egy időben több végpont párhuzamosan kommunikálhat	egy időben csak egyetlen master működhet egy irányban
többféle szélességű aljzat (1, 4, 8, 16, 32-szeres) → rugalmas	egyetlen nagy teljesítményű, de közös busz
Buszprotokoll: csomagokba ágyazza az adatokat!	

Miért gyorsabb egy NVMe SSD a hagyományos SATA SSD-nél?

- *Kihasználja a félvezető alapú tároló eszközök alacsony késleltetését és belső párhuzamosságát.*
- *4 sávos PCIe interface-t biztosítanak.*

4.5. USB szabvány

periféria busz

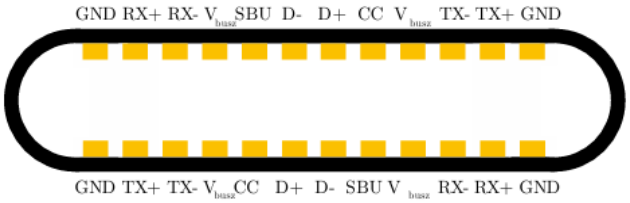
gyakori: USB C rajz + miben fejlődött

4.5.1. Hagyományos (USB-A / USB-B)

- 2 érpárt tartalmaz
- Teljesítménye maximum 5W

4.5.2. USB-C

- 4 érpárt tartalmaz
- 24 pines csatlakozó: 2x12 érintkező
- Teljesítmény: 15W – 100W



Csatlakozók:

- **GND:** mind a két végén van egy földcsatlakozó
- **RX+, RX-:** nagysebességű adatfogadási vezetékpár
- **TX+, TX-:** nagysebességű adatküldési vezetékpár
- **V_{busz}:** áramellátást biztosítja
- **D+, D-:** USB 2.0 adatátviteli vezeték
- **CC, SBU:** alternatív vezetékek

4.5.3. USB 4.0 szállított csomagok típusai

gyakori USB 4.0 gyorsítás

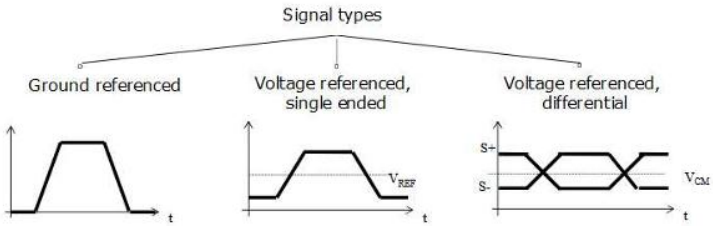
- **Aszimmetrikus adatátvitelre** is képes (80-120 Gbit/sec = 10-15 GB/s)
- Csomagkapcsolt adatátvitel

4 típusú csomag:

- **Időkritikus csomag**: Állandó sebességgel közlekednek (pl.: audio- és video adatok), adatátvitelben fellépő hibák javítása nem történik meg
- **Nagy adatcsomag**: Alacsony prioritásúak, sok adat továbbítására (pl.: backup, printer)
- **Megszakítási csomag**: Az egységek kiszolgálási kéréseire használják, kis adattartalmú csomag, ciklikus lekérdezés
- **Vezérlési csomagok**: címkiosztáshoz, eszközök azonosításához ún. handshake elven

nem volt még, de elvileg fontos

4.6. JELÁTVITEL TÍPUSOK



Ground referenced: PCI, SDRAM:

- Ha a vezeték 0 V körül van → 0, ha 5 V körül → 1
- olcsó implementáció
- érzékeny földzajra
- nagy feszültség szintek → nagy fogyasztás
- modern, nagy sebességű buszoknál használhatatlan

Voltage referenced, single ended: SSTL (DDR), AGP

- A jel nem a földhöz, hanem egy referenciafeszültséghez (V_{ref}) van mérve
- Alacsonyabb feszültség → kisebb fogyasztás
- Pontosabb küszöbérték a GND-hez képest
- Továbbra is zajérzékeny

Voltage referenced, differential LVDS (SATA, PCIe, Hypertransport)

- Két vezetéken terjed a jel (S+, S-), a vevő a két vezeték különbségét méri
- Zajtűrés: közös modusú zaj kiesik
- Kis amplitúdó → alacsony fogyasztás
- Két vezeték / jel → bonyolultabb áramkör

4.7. JELKÓDOLÁS (PAM3 – PAM4)

eddig nem volt, de kiemelte többször, hogy új kérdés lehet

Hogyan lehet az adatsűrűséget növelni? → jelkódolás

impulzus amplitúdó modulációs eljárások:

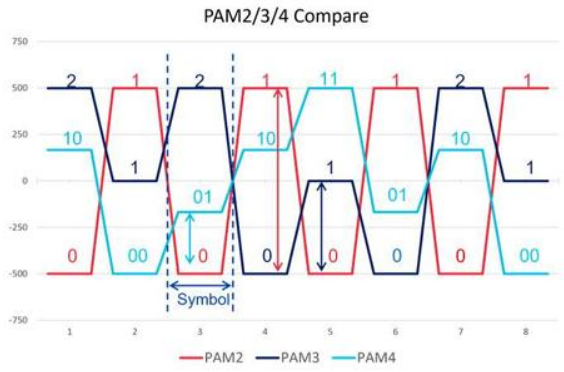
Miért sűrűbb a jel?

- Nem gyorsabban küld jelet, hanem több információ kerül egyetlen jelalakba

PAM3 (USB 4 2.0, GDDR7 memória)

A kép csak vizualizáció, nem követelmény

- 3 amplitúdó szint -1, 0, +1 (vagy 0, 1, 2)
- Tritek: hármas számrendszer
- Középső szint stabil referencia
- fizikailag stabilabb
- ÓRAJELENKÉNT 1,58 BITET TUD TOVÁBBÍTANI



PAM4 (2024 Nvidia GPU)

- 2 bit / szimbólum
- nagyobb bithiba arány miatt nem gyorsabb, mint a PAM3
- ÓRAJELENKÉNT 2 BITET TUD TOVÁBBÍTANI

gyakori PCI vs PCIe felhasználható

4.8. PÁRHUZAMOS ÉS SOROS BUSZOK

Soros buszok:

- Elég egy vezetékpár is
- Biteket bitsorozatként kódolva lehet átküldeni (ehhez plusz hardver szükséges)
- Egy gyors soros érpáron több adat továbbítható, mint több lassún
- Nagyobb frekvencián nagyobb távolságra biztosít adatátvitelt **Jitter nélkül**

Párhuzamos buszok:

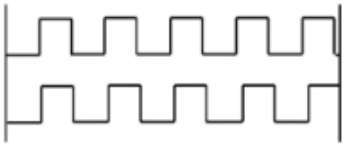
- Több vezetéket használ, ami egyszerre több adat átvitelét biztosítja
- Sok vezeték hátránya, hogy komplex, drága, sok helyet foglal
- Viszont hardver vonatkozásban viszonylag könnyen implementálhatók
- Magasabb frekvencián problémák jelennek meg (**Jitter**)

Manapság az egyetlen hely, ahol párhuzamos buszt használnak az a CPU és a Memória között található. Itt a soros adatátvitel helyett több száz vezetéken párhuzamos adatátvitel történik.

4.8.1. Jitter típusai:

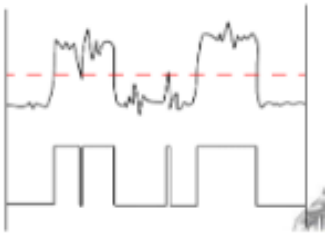
korszerűn volt, data valid window fontos

1. Delay Skew (időbeli eltérés):

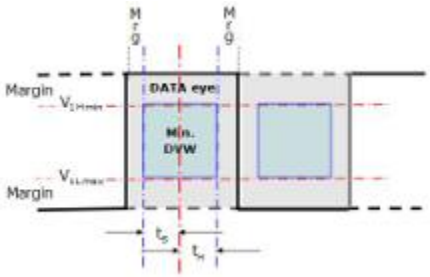


- Már kis vezetékhossznál is magas frekvenciánál megtörténik, hogy a párhuzamos adatok nem egyszerre érkeznek meg.

2. Elektromágneses interferencia (EMI):



- Sok párhuzamos vezeték és külső elektromágneses sugárzások generálhatják, ezt hívják „**zaj”**-nak. Emiatt egy érték kieshet az értelmezési tartományból, akkor azt nem tudjuk értelmezni
- **Data valid window:** Egy logikai érték pontos értelmezéséhez elengedhetetlen, hogy egy bizonyos feszülségszint alatt vagy felett legyen a jel legalább „Δt” ideig!



3. Vezetékek közötti keresztirányú áthallás:

- Interferenciát generál
- Minél hosszabb a vezeték, annál nagyobb

4.9. MODERN TECHNOLÓGIÁK

4.9.1. HyperTransport rendszerbusz

ritka

- Egy kétirányú soros/párhuzamos szélessávú, alacsony késleltetésű kapcsolat
- Fő feladata a front-side bus kiváltása *(FSB: processzoron kívüli kommunikáció)*
- CPU lapkájára van integrálva, de nagy sávszélességű I/O buszként is alkalmazzák
- Két féle egységet tartalmaz:
  - Alagút (tunnel): végén található két HT-port, amiknek segítségével több HT-egységet össze tudunk fűzni egymással
  - Barlang (cave): Ez zárja le az előbb említett HT láncot
- PCI-tól eltérően a HT nem rendelkezik dedikált I/O címtérrel, ehelyett memóriából leképzett I/O-val rendelkezik

4.9.2. QuickPath Interconnect (QPI)

kevésbé ritka

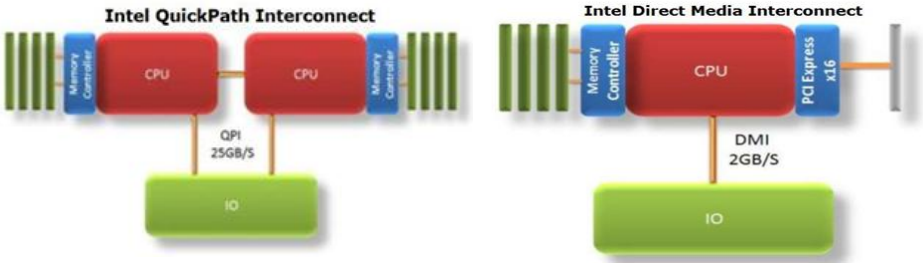
- Feladata a front side bus kiváltása (FSB) + Intel válasza a HT-ra
- A QPI-t használó processzorok is lapkára integrált memória vezérlőkkel és non-uniform memory access-el (NUMA) rendelkeznek
- Distributed shared memory: fizikailag elosztott memória egyetlen, logikailag közös címtérben
- 5 réteges architektúrát használ (2 vezeték minden egység között): 1 órajel alatt 20 adatbitet tud átvinni párhuzamosan

4.9.3. Direct Media Interface

nem volt még

Alsóbb kategóriás processzor családoknál.

Olcsóbb, nincs szükség olyan nagy adatátviteli sebességre

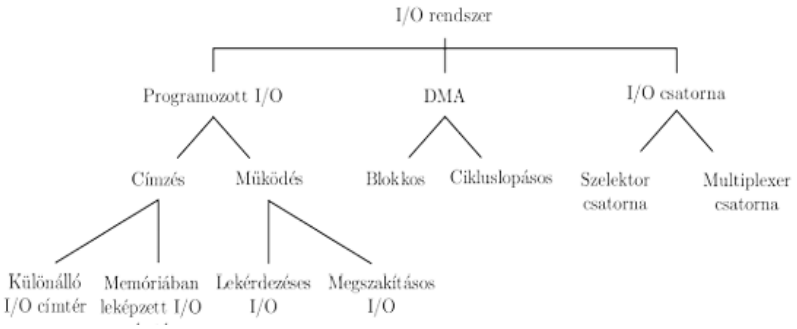


5. I/O RENDSZER

kevésbé ritka

Főbb I/O típusok:

- Programozott I/O
- DMA
- I/O csatorna



5.1. PROGRAMOZOTT I/O

A programozott I/O olyan adatátviteli módszer, ahol a CPU közvetlenül irányítja az I/O műveletet: elindítja, vezérli, figyeli az állapotot és lezárja az adatátvitelt.

Jellemzők:

- CPU-vezérelt adatátvitel
- Lekérdezéses vagy megszakításos vezérlés
- Közös buszhasználat a memória és az I/O között
- Egyszerű hardver megvalósítás
- Jelentős CPU-időt igényel

új kérdés, részei, átviteli módjai (feltételes, feltétel nélküli)

Előnyök:

- Egyszerű felépítés
- Olcsó megvalósítás
- Könnyen programozható

Hátrány:

- Lassú, erősen terheli a CPU-t
- Nagy adatmennyiségnél nem hatékony

5.1.1. I/O port

Az I/O port azon regiszterek összessége, amelyeken keresztül a CPU kommunikál a perifériákkal. Fizikailag az I/O vezérlőkártyán helyezkedik el, de logikailag címezhető, hasonlóan a memóriához.

I/O port vs memória: *nem volt külön kérdés*

- Feladatuk eltérő (periféria vezérlés vs adat tárolás)
- Mindkettő címezhető
- Mindkettő regisztereket tartalmaz

I/O port részei

Alap regiszterek:

- **Parancsregiszter (command):** CPU ide írja a „kívánságát” (I/O művelet típusát)
- **Adatregiszterek:** az adatátvitel végpontjai
  - **Data input regiszter:** CPU innen olvas adatot
  - **Data output regiszter:** CPU ide ír adatot
- **Állapotregiszter:** periféria aktuális állapotát jelzi (Ready, busy, Interrupt)

Lehet közös: Állapot és parancsregiszter vagy a data input és output

Egyéb regiszterek:

- **Jelenlét ellenőrző regiszter:** jelzi, hogy van-e csatlakoztatott eszköz az I/O portra
- **Eszköz tulajdonságait tartalmazó regiszter**
- **Bonyolult perifériáknál több funkcióhoz több regiszter**

5.1.2. Különálló I/O címtér

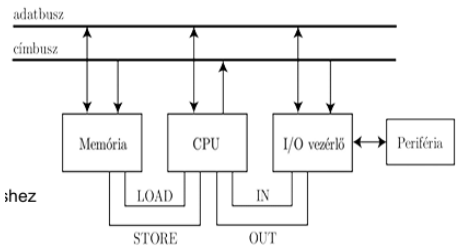
nem volt külön kérdés, ritkább zv-n

A CPU külön címtérben kezeli a memóriát és az I/O eszközöket. Az I/O műveletekhez speciális utasítások tartoznak. (pl.: PS2, RS232)

Nem mondta, hogy kéri

Jellemzők:

- Memória címzés: LOAD/STORE utasítások
- I/O vezérlő címzés: IN/OUT utasítások
- Azonos cím szerepelhet memóriacímként és I/O címként
- M/”I/O” vezérlőjel jelzi az aktuális cím típusát



Előnye:

- Egyszerű architektúra
- Olcsó hardver

Hátránya:

- CPU-terhelés
- Plusz utasítások az I/O kezeléséhez

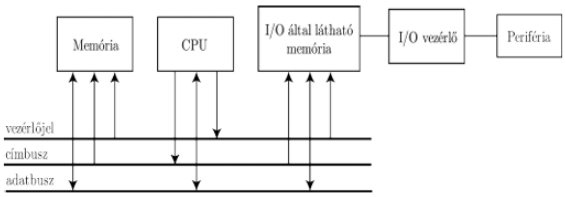
5.1.3. Memóriában leképzett I/O

nem volt külön kérdés, zv-n gyakoribb

A perifériák számára a CPU memóriaterületet foglal le, és a perifériák regiszterei ezen a címtartományon keresztül érhetők el.

Jellemzők:

- LOAD/STORE utasításokkal történik az I/O is
- A cím dönti el, hogy memória vagy I/O művelet történik
- I/O vezérlő közvetlenül hozzáfér a rendszerbuszhoz → gyorsabb átvitel



Előny:

- Nincs szükség külön I/O utasításokra
- Gyorsabb átvitel
- Egységes címzési modell

Hátrány:

- CPU-t továbbra is terheli
- Memóriacímtartományt foglal

Pl: kijelző, frame buffer közvetlen CPU-címzéssel

Működéseinek módjai:

5.1.4. Feltétel nélküli adatátvitel (direkt programozott)

Az adatátvitel ellenőrzés és visszacsatolás nélkül történik. (nincs szükség se ellőtte, sem utána)

Előfeltétel:

- Perifériának mindig adatátvitelre alkalmas állapotban kell lennie
- Nincs ellenőrzés
- Semmilyen szinkronizálás nincs a CPU és a periféria között

gyakori

Hátrány:

- Nincs visszajelzés
- Adatvesztés lehetséges

Pl: kijelző, nincs visszacsatolás → adatvesztés előfordulhat

5.1.5. Feltételes adatátvitel

új kérdés

Az adatátvitel egy feltétel teljesüléséhez kötött.

1. Lekérdezéses adatátvitel

A CPU folyamatosan olvassa az állapotregisztert, amíg az eszköz készen nem áll. Beírja kívánságát az I/O vezérlőbe és várja a választ. *(kész vagy már? kész vagy már? .....)*

Jellemzők:
- Aktív várakozás
- Nem ismert a várakozási idő

Hátrány:
- Rendkívűl pazarló
- Lassú perifériáknál különösen rossz

2. Megszakításos adatátvitel

A CPU elindítja az I/O műveletet, majd más feladatot végez. A periféria megszakítással jelzi, ha készen áll. *pl: nyomtató (nem várja meg míg kinyomtatja az egész oldalt)*

Működése:
- A CPU beírja kívánságát az I/O vezérlőbe
- Periféria dolgozik
- Kész állapotban megszakítást küld
- Megszakítás kiszolgálása eredményezi az adatátvitelt

Előny:
- CPU kihasználtság jobb
- Gyorsabb, mint a lekérdezéses

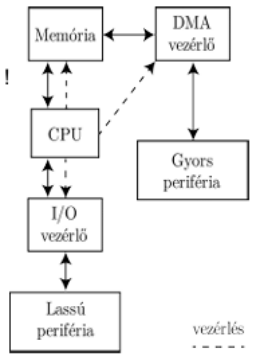
Hátrány:
- Nagy mennyiségű adat esetén sok megszakítás
- Még mindig a CPU kezdeményezi és vezérli az átvitelt

5.2. DMA

Direct Memory Address: nagy mennyiségű adatot közvetlenül a memória és a periféria között mozgat, a CPU közreműködése nélkül végzi az adatátvitelt. A CPU csak az indítást és a lezárást végzi.

jó tudni a következő két részhez

Jellemzők:
- Nagy adatmennyiség → blokkos átvitel
- **Csak gyors perifériáknál használják!**
- CPU nem mozgat adatot
- *Jelentősen csökkenti a megszakítások számát*



Feltételei:

- Közvetlen memória-címgenerálás
- Buszvezérlési funkciók (busz request / acknowledge)
- DMA vezérlő jelenléte

Regiszterei:

- DC (Data Counter): Átvivendő adategységek száma
- I/O címregiszter (I/O AR): memória kezdőcím
- I/O adatregiszter (I/O DR): aktuális adat
- Belső transzparens regiszterek

kevésbé ritka

5.2.1. DMA felparaméterezés

Felparaméterezés a CPU által, a programozott I/O-n keresztül történik és ezeket a fontos információkat viszi át:

1. Átvitel iránya (olvasás / írás)
2. I/O egység címe
3. Memória kezdőcím (I/O AR)
4. Adat típusa (byte / félszó / szó)
5. Átvivendő egységek száma (DC-be)
6. Átvitel módja (blokkos / ciklusloPásos)
7. DMA csatorna prioritása
8. Résztvevő egységek típusa (I/O-memória, memória-memória, I/O-I/O)

5.2.2. Blokkos átvitel lépései – 6 lépés

kevésbé ritka (régen rajz is kellett)

1. CPU felparaméterezi és elindítja a DMA-t
2. DMA buszhasználatot kér (DMA request)
3. CPU lemond a buszról (DMA acknowledge)
4. DMA adatot kér a perifériától az I/O DR-be
5. DMA adatot ír a memóriába az I/O AR által meghatározott memóriacímre
6. DC--, cím++, ismétlés vagy megszakítás küldése *(A DMA ellenőrzi a DC tartalmát, ha nem 0 akkor vissza 4. pont)*

rég volt, de kérheti

5.3. I/O CSATORNA

A DMA továbbfejlesztése lassabb perifériákhoz. A CPU nem paraméterez, hanem egy memóriában tárolt I/O programot indít el, amelyet az I/O csatorna hajt végre.

- CPU csak kezdeményez
- I/O csatorna hatja végre az átvitelt
- Nincs felparaméterezés
- Önálló vezérlőegység

Típusai:
- szelektor csatorna
- multiplexer

5.3.1. Szelektor csatorna

Lassabb perifériák közül is gyorsabb perifériákhoz

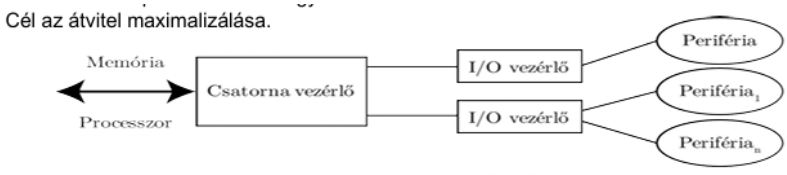
- Gyorsabb perifériákhoz
- Egyetlen I/O vonal, amire több I/O vezérlő is rácsatlakozhat
- Egyszerre csak egy aktív

Előny: gyors átvitel
Hátrány: nincs párhuzamosság

5.3.2. Multiplexer csatorna

Lassú perifériák közül is a lassabb perifériákhoz → párhuzamos kezelés

Byte multiplexer: Byte-onkénti adatküldés
Blokk multiplexer: Blokkonkénti adatküldés



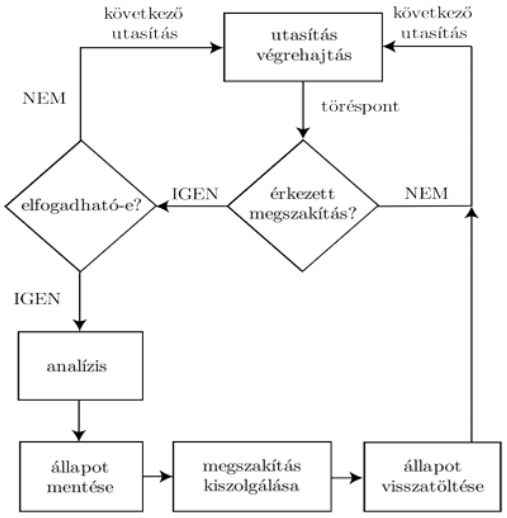
gyakori, definíció + célok

6. MEGSZAKÍTÁSI RENDSZER

Definíció: A feldolgozás szempontjából váratlannak tekinthető események kezelésére szolgáló művelet.

Célja nem csak a reagálás, hanem a folyamatosan változó körülmények között az optimális működés biztosítása.

Lényege: Csak akkor foglalja le a CPU figyelmét amikor arra tényleg szükség van → *CPU-nak nem szabad felesleges időt töltenie várakozással az I/O-ra vagy a külső eseményekre.*



ábra: nem nagyon volt még, de fontos

6.1. MEGSZAKÍTÁSI OKOK PRIORITÁSI SORREND:

1. **Géphibák**: nem letilthatók. Legmagasabb priorítás, nem késleltethetők.
   Példa: túlhevülés, tápegység-ingadozás, parításhibák, watchdog timeout
2. **I/O források**: perifériák jelzik
   Példa: adat érkezett, adatátvitel befejeződött, DMA művelet véget ért *gyakori*
3. **Külső források**: felhasználó vagy külső rendszer
   Példa: reset gomb, hálózati kommunikáció
4. **Programozási források**:
   a. **Szándékos:** amikor egy program megszakítást kér
   b. **Nem szándékos (hibakezelés):** mindig valamilyen utasítás végrehajtása vagy a végrehajtás megkísérlése során alakul ki

előzővel lehet

6.1.1. Programozási okok miatt fellépő megszakítások

1. **Memóriavédelem megsértése**: Ha egy folyamat belenyúl más memóriaterületbe.
   Minden program számára lefoglalódik egy memória terület, amit védeni kell véletlen felülírástól! Amennyiben egy program idegen címre hivatkozik, az alkalmazott memóriavédelem működésbe lép.
2. **Fizikai cím túlcímzése**: *Pl: CPU 32 bitet tud címezni → 4 GB címtér*
   De a hardverben lehet csak 1 GB RAM → túlcímzés → megszakítás
3. **Címzési előírások megsértése**: *Pl: 32 bitesek a címek és a címzés byte-osan történik → minden 4-ik byte címezhető ilyenkor, de a program ezt megsérti → megszakítás*
4. **Aritmetikai hibák**: végrehajtás közben fellépő hibák
   Pl: overflow, underflow, 0-val való osztás … stb.

6.2. MEGSZAKÍTÁSOK CSOPORTOSÍTÁSA

1. **Szinkron vs aszinkron megszakítások**
   a. **Szinkron**: a program mindig ugyan ott jelentkezik a megszakítás (pl. 0-val osztás)
   b. **Aszinkron**: véletlenszerűen lépnek fel
      - várható: I/O egység
      - nem várható: hardverhibák

gyakori, alrészek nem kellenek pontosan max példa

2. **Utasítások között vagy közben**
   a. **Között**: Utasítás végrehajtásának eredményeképp következik be (pl. overflow)
   b. **Közben**: Utasítás végrehajtás alatt, nincs szinkronban a ciklussal (pl: hardverhiba)

3. **Felhasználó által kért vs nem kért**
   a. **Kért**: rendszerhívások (pl: OS rutinok, BIOS rutinok)
   b. **Nem kért**: pl: overflow, I/O egység által kért, hardverhiba

4. **Megszakított program folytatható vs nem folytatható**
   a. **Folytatható**: pl I/O megszakítás
   b. **Nem folytatható**: pl: hardverhiba

5. **Maszkolható vagy nem maszkolható**
   a. **Maszkolható**: le lehet tiltani, prioritás alapján nem lép érvénybe pl: I/O kérés
   b. **Nem maszkolható**: pl: súlyos hardverhiba (túlmelegedés)

6.3. KISZOLGÁLÁS ÁLTALÁNOS FOLYAMATA

nem volt még külön

Alapelv: Megszakítás segítségével egy futó program felfüggeszthető, majd egy másik (megszakítási rutin) lefuttatása után folytatható.

1. Kiszolgálás előkészítése:

- Egy egység megszakításkérést ad ki → aktiválja az INTR vezérlő vonalat
- CPU:
  - befejezi az aktuális utasítást
  - Minden **utasítás töréspontban** megnézi, van-e megszakítás kérés
- A megszakítást a CPU megszakítási bemenetén érzékeli

  Régen egy szintű (van/nincs), ma már megszakításvezérlő áramkör kezeli a prioritásokat

2. Elfogadhatóság:

- Az aktuális folyamat **megszakítható**
- A megszakítás **prioritása megfelelő**
- A megszakítás **nincs maszkolva**

gyakori, mikor érvényes a megszakítás?

3. **Elfogadás és visszajelzés**: Ha a megszakítás elfogadott a CPU aktiválja az INTACK vezérlővonalat → periféria deaktiválja az INTR vonalat

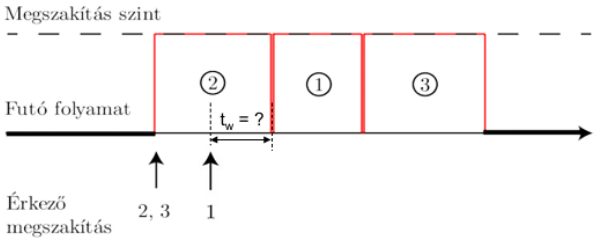
4. **Kiszolgálás végrehajtása**: A CPU eltárolja a megszakított folyamat kontextusát (programtól független tárolóban) → a PC-be betölti a megszakítási rutin kezdő címét (he szükséges állapotinformációkat is)

gyakori, fontos főleg az utolsó + néha rajzok is

6.4. MEGSZAKÍTÁSI RENDSZEREK SZINTJEI:

6.4.1. Egyszintű megszakítási rendszer

- Nem megszakítható
- Új megszakítás csak normál állapotba való visszatérés után kezdődhet.

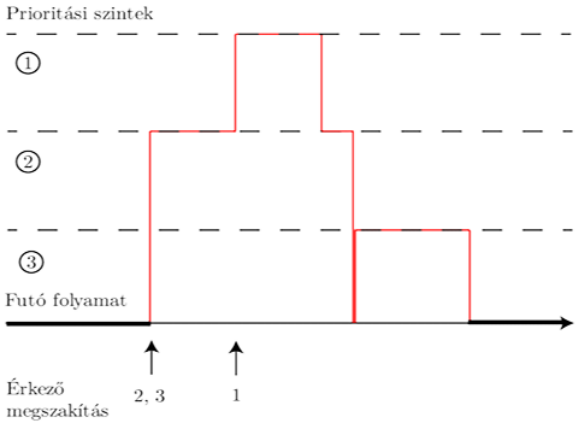


Prioritás: 1 > 2 > 3

Probléma: még a legnagyobb prioritású megszakítás is várakozhat.

6.4.2. Többszintű megszakítási rendszer

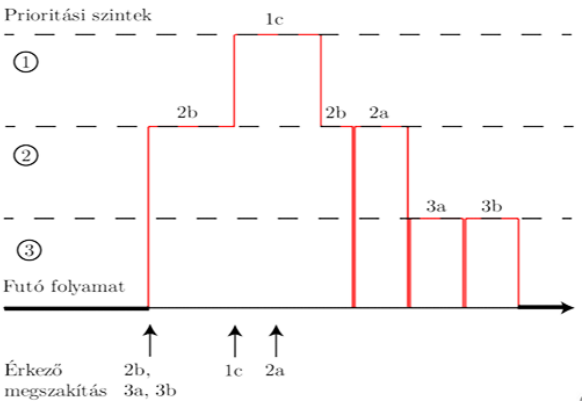
- Megszakítások megszakíthatják egymást
- Végrehajtás fontossági sorrendben



Hátrány: túl sok megszakítás van → nem lehet mindegyikhez külön prioritási szintet rendelni

6.4.3. Többszintű, többvonalú megszakítási rendszer

- A megszakítási okokat osztályokba sorolják
- Minden osztályhoz prioritási szint tartozik
  - Osztályok: 1, 2, 3
  - Prioritások: 1, 2, 3
- Osztályon belül:
  - megszakítás típusok: a, b, c, d, …
- Működés:
  - osztályok között: **többszintű**
  - osztályon belül: **egyszintű**
  - osztályon belül is léteznek prioritások



7. GYORSÍTÓTÁRAK (CACHE)

Definíció: A cache az adatok és utasítások átmeneti tárolására szolgáló gyors működésű, a programozó számára **nem** elérhető tároló.

A cache egy nagyon gyors, a CPU-hoz közeli memória, amely a gyakran használt adatokat és utasításokat tartalmazza átmenetileg, hogy a processzor ne a lassú RAM-ból olvasson.

Miért kell? CPU sebessége sokkal nagyobb tempóban nőtt, mint a memóriáé. (10 000x)

- Általában a processzor lapkán helyezkedik el
- Nem címezhető → nem része a memóriatérnek
- Cache és memória között **mindig** blokkos adatátvitel

7.1. CACHE SZINTEK

ritka

Alapvető tervezési probléma:

- nagy cache → jobb találati arány, de lassabb keresés
- kicsi cache → gyors, de több a *cache miss*

	Elérési idő	
	nanosec	ciklus
L1	1.3-1.5 ns	3-4
L2	4.5-8	10
L3	12-20	20-40
RAM	60-80	50-200

csak szemléltetés

Ez vezetett a többszintű cache-ek bevezetéséhez.

Többszintű cache célja:

- a gyakran használt adatok a lehető leggyorsabb szinten legyenek
- ritkábban használt adatok lassabb, de nagyobb cache-ben

Jellemző szintek: L1, L2, L3

Az L1 cache a regiszter után a leggyorsabb tároló, általában a CPU órajelén működik.

7.2. FELOSZTÁS: UTASÍTÁS- ÉS ADATCACHE

A tapasztalat szerint célszerű az utasításokat és az adatokat külön cache-ben tárolni.
A modern processzorok ezt az elvet követik.

nem volt még, de fontos

Okai:

- Az adatok és utasítások viselkedése eltérő
- Ha **sok adattal** dolgozik, közös cache esetén kiszoríthatná az **utasításokat**
- **Utasítás cache**: csak olvasást kell optimalizálni
- **Adat cache**: olvasást és írást is

Modern, három szintű feloszlás:

L1 Utasítás cache
L1 Adat cache
L2, L3 Mixed cache

Cache és memória kapcsolata

A cache nem önálló tároló, hanem a főmemória egyes részeinek másolatát tartalmazza.
Fontos követelmény: a cache és a memória azonos adatainak egyezniük kell.

Ha egy cache-ben lévő adat módosul, azt előbb-utóbb vissza kell írni az operatív tárba.

Adatátvitel:
- RAM ↔ cache: mindig blokkos
- CPU ↔ cache: byte szintű is lehet

A blokkos átvitel oka: nagy valószínűséggel egymást követő memória-címek kerülnek felhasználásra

gyakori, mi az a TAG, mire való

7.3. TAG

A cache-ben a memória egyes, egymást követő rekeszeinek **tartalmát tároljuk**.
Ezek mellé el kell tárolni az adatok **memória címét** is!

Tárolni kell:
- adatokat
- adatokhoz tartozó memória címet vagy annak egy részét

TAG: memóriacím cache-ben tárolt része.
A TAG alapján dönthető el, hogy a keresett adat az adott cache sorban található-e.

A TAG származhat:
- fizikai címből (cache az MMU után)
- virtuális címből (cache az MMU előtt)

MMU: Memory Management Unit

Virtuális TAG hátránya: → *modern architektúrák fizikai TAG-et használnak*
- nagyobb TAG méret, mivel a virtuális címtér is nagyobb
- Virtualizációból adódó helyettesítések kezelése - *ugyanaz a virtuális cím különböző fizikai memóriacímekre mutathat, attól függően, hogy melyik folyamat fut.*

Virtuális TAG előnye: kisebb cache miss késleltetés → *előbb derül ki a hiba, mert nem kell megvárni a címfordítást*

7.4. CACHE HIT ÉS CACHE MISS

gyakori

A cache-ben történő visszakeresés tartalom szerinti asszociatív kereséssel történik.
(CAM – Content Addressable Memory)

A CPU a keresett adatt címét hasonlítja össze a cache-ben tárolt TAG-ekkel.

Cache hit: a keresett adat a cache-ben megtalálható
- az adat gyorsan elérhető

Cache miss: a keresett adat nincs a cache-ben
- a CPU a RAM-ból olvas – lassú
- az adat betöltődik a regiszterbe és a cache-be is

Modern rendszerekben a találati arány közelít a 100%-hoz, elvárt miss 1-2%.

kevésbé ritka
csak felsorolás

7.5. HELYETTESÍTÉSI STRATÉGIA

A cache tartalmának cseréjekor a találati arány fenntartása érdekében lényeges a megfelelő „**replacement policy**” kiválasztása.

Állapot: a cache tele van

Főbb típusok:
- **FIFO** *(First In First Out): legrégebben betöltött blokk*
- **LIFO** *(Last In First Out): legutóbb betöltött blokk*
- **LFU** *(Least Frequently Used): legritkábban használt blokk*
- **LRU** *(Least Recently Used): legrégebben használt blokk*

7.6. VEZÉRLŐ BITEK (DIRTY/VALID BIT)

nagyon gyakori

A cache nemcsak adatot és TAG-et tárol, hanem állapotinformációkat is.

Legfontosabb bitek:

- **D (dirty) bit**: jelzi, hogy az adat módosult-e.
    - Az ilyen blokk helyére nem lehet új adatot betölteni
    - Előbb a módosított adatokat ki kell írni a memóriába
- **V (valid) bit**: jelzi, hogy a cache sor vagy blokk érvényes-e
    - Ha **V = 1** akkor érvényes az adat
    - Törlés után **V = 0**, ezzel jelzi a CPU-nak, hogy **szabadon írható** terület

7.7. FONTOSABB PARAMÉTEREK

nem volt még de fontos

Adat aktualizálási módszer

- **write through**
  - adat módosításakor azonnali visszaírás a főmemóriába
  - nem alakul ki inkonzisztencia
  - lassabb megoldás
- **write back**
  - visszaírás csak cache line cseréjekor
  - gyorsabb és gyakrabban használt
  - adatvesztés lehetséges (pl. áramszünet)
  - *dirty bit használata szükséges*

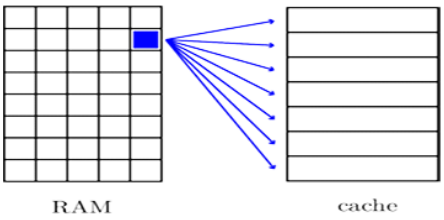
Átlagos elérési idő: AAT = „Hit time” + („Miss rate” x „Miss penalty”)

Hit time: cache találat kiszolgálási ideje
Miss rate: cache hibák aránya
Miss penalty: cache miss esetén a memóriaelérés többletideje

gyakori, általában 1-1 mondat + előny/hátrány vagy n-way külön

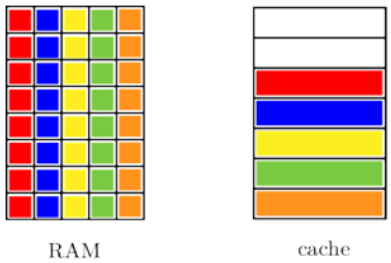
7.8. CACHE TÍPUSOK

7.8.1. Teljesen asszociatív cache (full associative)



- egy memóriablokk bármely cache sorba kerülhet
- elhelyezés sorát a helyettesítési stratégia határozza meg
- kereséskor a CPU a tageket vizsgálja minden sorban egyszerre, mivel az adat bárhol lehet
- nagy találati arány és rugalmasság *(legoptimálisabb elrendezés)*
- drága, bonyolult, nagy fogyasztás *(szükséges n darab párhuzamos összehasonlítókör)*

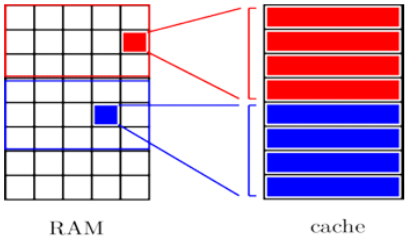
7.8.2. Direct mapping (1 way set associative cache)



- egy memóriablokk csak egy adott cache line-ba kerülhet
- egy cache line-hoz több memória blokk hozzá van rendelve, előfordulhat, hogy gyakran cserélni kell a cache tartalmát
- Előny: gyors, olcsó
- Hátrány: rugalmatlan, alacsonyabb találati arány

7.8.3. N-way associative cache

gyakori



Kompromisszum az előző kettő között

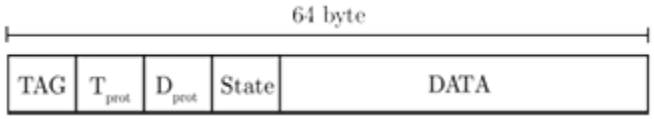
- egy blokk N különböző cache line-ba kerülhet
- N tipikusan: 2, 4, 8, 16
- Kevesebb összehasonlító áramkör szükséges
- jó találati arány és elfogadható költség

Pl: 4 utas esetén egy blokk 4 cache line-ba kerülhet → a CPU a csoport index alapján 4 darab cache line-ra tudja szűkíteni a keresést így 4 db összehasonlító áramkör szükséges

7.8.4. Sector mapping cache

A gyakorlatban nagyon ritkán használt.
Itt a csoport bárhová kerülhet, viszont a blokknak a helye a csoporton belül kötött.

7.9. CACHE LINE FELÉPÍTÉSE



64 byte-os cache line esetén a felépítés:

- **Directory Entry:** TAG és egyéb állapotjelzők
- **Tprot:** Tag protection bitek – biztosítják a tag egyezőséget elírások ellen
- **Dprot:** data protection bitek – biztosítják az adat egyezőséget elírások ellen
- **State:** állapot bitek (V bit, D bit …)
- **Data:** maga az adat

gyakori

7.10. EXCLUSIVE VS INCLUSIVE CACHE

Exclusive cache

nem igazán volt, de fontos

- cache szintek nem tartalmazzák egymás adatait
- ugyanaz az adat csak egy szinten jelenhet meg
- több CPU vagy mag esetén komplikált
- Adatok betöltése két féle képpen történhet:
    - Először L3-ba és ha szükséges akkor onnan L2-be, L1-be
    - Először L1-be, majd, ami nem fér be L2-be, ami oda se azt L3-ba *(victim cache)*

Inclusive cache

- a magasabb szintű cache tartalmazhatja az alacsonyabb szint adatait
- **Hátrány**: duplikáció és csökken az alacsonyabb szintű cache mérete
- **Előnye**:
    - Magasabb szintű cache sora szabadon cserélhető, mert alacsonyabban is megtalálható az adat
    - több mag esetén, másik mag cache-ében kell keresni akkor duplikálás miatt elég az alacsonyabb szintűben (pl: L2)

Megjegyzés: több magos CPU-knál L1, L2 gyakran inclusive, L3 nem – koherencia bonyolultság

7.11. CACHE KOHERENCIA MECHANIZMUSOK *gyakori*

Több CPU-s vagy többmagos rendszerek esetében mindig figyelni kell arra, hogy az egyes CPU-k (vagy magok) gyorsítótáraiban megegyező adatok legyenek!

Cél, hogy a módosított adat a lehető leggyorsabban bekerüljön az összes processzor gyorsítótárába, mielőtt a többi esetlegesen műveletet végezne rajta.

Adatérvényesítés módjai

- **Invalidáció**: Az érintett cache line érvénytelenítése más cache-ekben (V bit = 0)
  → gyakoribb mert kis forgalmat generál
- **Felülírás**: A módosított adat közvetlen elküldése a többi cache-be
  → ritkább, nagyobb adatforgalom miatt

Cache koherencia protokollok:

- Snoopy: *minden cache figyeli a közös buszt és reagál az érintett tranzakciókra*
- Snorf: *Snoopy egy variánsa*
- Könyvtár alapú*: Központi nyilvántartás kezeli, mely cache-ek tartalmazzák az adatot*
- **MESI**
- **MOESI**

7.11.1. MESI protokoll *gyakori*

Állapotok:

- **M – Modified:** Az adat módosult, csak ez a tár valid, a többi invalid, még nincs visszaírva memóriába
  → csak ez írható szabadon
- **E – Exclusive:** Az adat megegyezik a memóriával, **csak ebben a cache-ben van**
  → íráskor nem kell más cache-ekkel egyeztetni
- **S – Shared:** Az adat több cache-ben is megtalálható, és **egyezik a memóriával**
- **I – Invalid:** A cache line érvénytelen, nem tartalmaz használható adatot

7.11.2. MOESI *gyakori*

A MESI kibővítése egy extra állapottal

- **O – Owned:** A módosult és megosztott állapotot helyettesíti. Egyetlen cache tulajdonában van. Ezzel elkerülhető, hogy vissza kelljen írni a memóriába megosztás előtt. **Előny:** A módosított adat közvetlenül átadható egyik cache-ből a másikba memória nélkül

Akkor előnyös, ha a kommunikáció két CPU között lényegesen jobb, mint a memóriával.

MESIF/MOESIF: *kérheti pluszban*

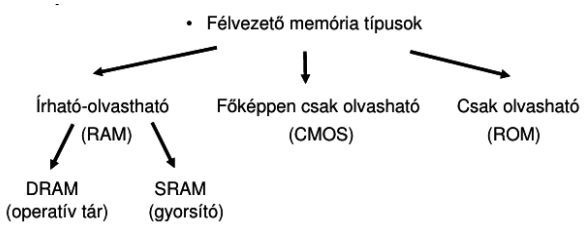
- **F – Forward:** hasonló az Owned-hez, de itt nem módosult cache line van kinevezve arra, ha egy másik CPU vagy mag igényli ezt a blokkot így nem a memóriából tölti be.

8. MEMÓRIÁK

8.1. FÉLVEZETŐ MEMÓRIÁK

A félvezető memóriák nagyságrenddel gyorsabbak a háttértáraknál, ezért közvetlenül a CPU működését szolgálják ki. Funkció szerint három fő csoportjuk van:

ritka (csoportosítás)



RAM (Random Access Memory) - írható, olvasható, nem maradandó:
→ a futó programok és adatok tárolására

- **SRAM** - statikus
- **DRAM** - dinamikus

ROM (Read Only Memory) - csak olvasható, maradandó:
→ rendszerindítás és alap hardverfunkciók

- BIOS
- POST tesztek *(részegységek működőképességét tesztelő programok)*
- eszközspecifikus adatok (pl. hálókártya MAC-cím)

CMOS – speciális, elemről táplált memória
→ BIOS beállítások és valós idejű óra

- kikapcsolt állapotban is megőrzi az adatokat
- nagyon alacsony fogyasztás

8.2. STATIKUS MEMÓRIA (SRAM)

nem volt még

Az SRAM-ban az adat nem töltésként, hanem logikai állapotként van tárolva.
Felhasználás: Regiszterek, cache

- **Tárolóelem**: flip-flop (4-6 tranzisztor / bit)
- Az állapot **tápfeszültség meglétéig stabil**
- **Nincs szükség tartalom frissítésre** → *Az adat nem kondenzátorban van*
- Ciklusidő = elérési idő

Energia és költség:

- Nincs szivárgásos adatvesztés (nincs refresh)
- Viszont sok tranzisztor → **nagyobb statikus fogyasztás**
- Nagy lapkaterület → **drága**

Következtetés: SRAM gyors, kis kapacitású, drága → **nem alkalmas főmemóriának**

nem volt még

8.3. DINAMIKUS MEMÓRIA (DRAM)

A DRAM-ban az adat töltésként van eltárolva
Felhasználás: Főmemória (operatív tár)

- Tárolóelem: **kondenzátor + 1 tranzisztor** *néhány pikó farad kapacitású kondenzátorok*
- A kondenzátor **kisül** (szivárog) → az adat idővel eltűnik
- **Periodikus frissítés kötelező** – a szivárgás miatt

Energia és költség:
- Kevés tranzisztor →kis lapkaterület
- Alacsony fogyasztás bitenként
- Olcsó, bővíthető

Következtetés: DRAM lassabb, de nagy és olcsó → **operatív tár**

DRAM típusok:
- **Klasszikus DRAM** (aszinkron) *Nem kötődik rendszerórához, rossz párhuzamosíthatóság*
- **SDRAM** (szinkron) *órajelre válaszol, 2000-től domináns megoldás*
    - SDR SDRAM (Single Data Rate)
    - DDR SDRAM (Double Data Rate)

8.3.1. SDR SDRAM

nem volt még

Szinkron memória, de az adatátvitel **csak az órajel felmenő élén** történik.

- Az adattároló több logikai egységre (**bankra**) van felosztva

Bank: sor-oszlop mátrixként értelmezhető, egy bankon belül a műveletek egymást blokkolják

Interleaved (futószalag elv):
- A különálló bankok el vannak csúsztatva
  egymás utáni adatok több különböző bank-ban vannak elcsúsztatva
- A memóriavezérlő egyidejűleg több memória hozzáférési parancsot hajthat végre
  Miközben az egyik bank várakozik (pl. sor megnyitása), egy másik kiszolgálható
- Az interleaving miatt az SDRAM **gyorsabb, mint az aszinkron** DRAM

Késleltetés (latency): A **futószalag elvű olvasás** azt jelenti, hogy az olvasási parancs kiadása után az adat nem azonnal, hanem fix számú órajel múlva jelenik meg. Ez a várakozási idő a **késleltetés**, ami fontos teljesítményparaméter.

kevésbé fontos

8.3.2. DDR

A **DDR SDRAM** az SDR SDRAM továbbfejlesztése.

Az adatátvitel nemcsak a felmenő, hanem a lemenő órajel élen is történik.
→ Egy órajel alatt **kétszeres adatátvitel**

Alacsonyabban tartott frekvencia (órajel) előnye, hogy **javul a jelintegritás**: hosszabb idő áll rendelkezésre az adatjelek értelmezésére, így a vezérlő a gyengébb vagy zajosabb jeleket is megbízhatóbban tudja kiértékelni.

- 2N-prefetch eljárás
- Egy órajel alatt kétszeres adatátvitel
- alacsonyabb frekvencia, javul a jelintegritás

gyakori pl 8n prefetch eljárás

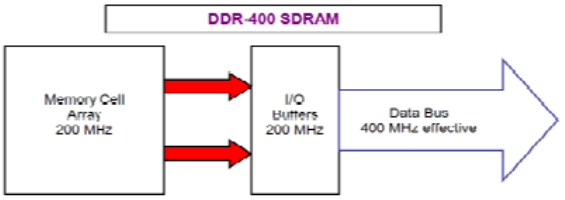
8.3.3. Prefetch eljárás

nem külön anyag, csak magyarázat

A DRAM cellák nem képesek a külső adatbusz sebességével működni.
Ezt oldja meg **a prefetch mechanizmus**.

Prefetch lényege:

- A memória belső magja egyszerre több bitet olvas ki
- Ezek egy belső pufferbe kerülnek
- A külső busz több lépésben továbbítja az adatokat



Következmény:

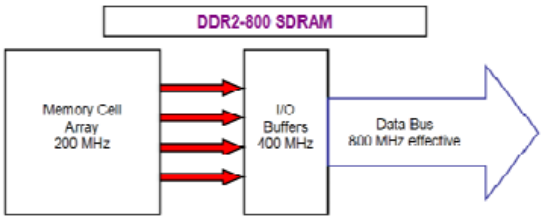
- A **belső busz szélesebb, mint a külső** (2n, 4n, 8n, 16n)
- Ez növeli a ciklusban mért késleltetést, de növeli a sávszélességet

nem fontos

8.3.4. DDR2

A DDR SDRAM közvetlen továbbfejlesztése. Alapelv nem változik, azonban a belső működés nagyobb mértékű párhuzamosítást alkalmaz.

- **4N- prefetch** eljárás – ciklusonként 4 adat
- **Nagyobb késleltetés**, mint a DDR esetében
- Alacsonyabb órafrekvencia-meghajtás
  → *alacsonyabb energiafogyasztás*
- Külső adatátviteli frekvencia növelhető
  → *sávszélesség jelentősen megnő*



tápfeszültség: 1,8V, átviteli sebesség elérheti a 6400MB/s

gyakori DDR3 vs DDR4

8.3.5. DDR3

A DDR3 SDRAM működési elve megegyezik a DDR2-évl, azonban a belső párhuzamosítás tovább növekszik.

- **8N-prefetch eljárás** – egy ciklus alatt 8 adatot készít elő
- lehetővé teszi, hogy egy chipen belül akár 8 Gbit kapacitást helyezzünk el
- A modulok fizikai kialakítása 240 DIMM lábat használ
- 8 különálló bank chipenként

tápfeszültség: 1,5V, átviteli sebessége elérheti a 17 000 MB/s

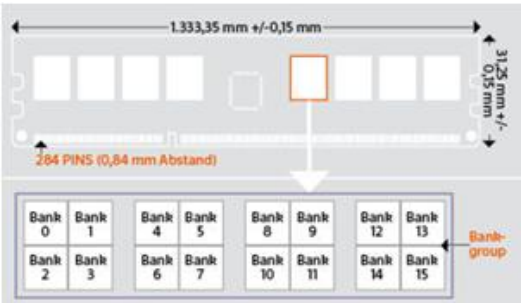
gyakori DDR3 vs DDR4

8.3.6. DDR4

Tovább csökkenti az energiafogyasztást és növeli a párhuzamosíthatóságot.
Tápfeszültség: 1,2V → kisebb hőtermelés, jobb energiahatékonyság.

gyakori 8n prefetch

- **8N-prefetch eljárás**
- Alacsonyabb órafrekvencia-meghajtás → alacsonyabb energiafogyasztás
- Külső adatátviteli frekvencia növelhető → sávszélesség megnő
- **Nagyobb késeltetés, mint DDR3** esetén ugyanazon órafrekvencia mellett
- **Bankok: 16, csoportosítva** (4 csoport x 4 bank)
  → Vezérlés egyszerre 2-4 bankot választhat ki
  → időosztásos multiplexelés elvén



Megbízhatóság javítása:

- ECC mellett **CRC** (Cyclic Redundancy Check) véletlenszerű változásokat érzékel
- Chipenként **extra paritás**
- **ODT**: On-Die Termination és feszültségszabályozás
- **Gear Down mode**: Csökkenti a prefetch értékét, ha szükséges

Fizikai kialakítás: 288 DIMM pin

paritás: egyszerű hibafelismerő, bitcsoport összegét vizsgálja (összeadja, eredmény páros vagy páratlan)

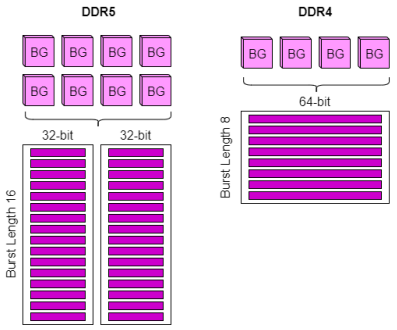
ODT: a memóriachipbe épített ellenállás, amely elnyeli a visszaverődő jeleket, javítva ezzel a jel integritását magas frekvencián.

gyakori újdonságai DDR4-el szemben

8.3.7. DDR5

Fő fejlesztések a sávszélesség, párhuzamosítás és megbízhatóság növelése. Tápfeszültség: 1,1V → még alacsonyabb energiafogyasztás

- **16N-prefetch** eljárás
- **Nagyobb sávszélesség**: magasabb külső frekvencia (4800 MT/s-től)
- **Nagyobb késeltetés, mint DDR4** esetén
- **32 bank** csoportosítva **(8 csoport x 4 bank)** *DDR4 esetén: 4x4*
- **Modulonként 2 csatorna (32 bit)** *egyszerre tudnak olvasni vagy írni*
- **On-Die ECC** memória chipbe integrálva
  → chip szintű hibajavítás
- Operációnként 64 Byte adat továbbítása
  egyetlen memória művelet során
- Akár 12 NYÁK réteg
  több sáv, jobb áramellátás nagyobb sebességnél



On-Die ECC: *automatikusan észleli és javítja a véletlenszerű bithibákat anélkül, hogy a rendszernek külön kellene kezelnie*

8.3.8. Szinkron DRAM időzítési paraméterek

A DRAM működése bank-sor-oszlop hierarchiára épül. Egy adat olvasásához először meg kell nyitni egy sort (ROW), majd azon belül oszlopokat (COLUMN) lehet elérni, végül a sort le kell zárni. Az időzítési paraméterek ezeknek a lépéseknek a kötelező várakozási idejét adják meg órajelciklusokban.

Fontosabb időzítési paraméterek: *kevésbé ritka*

- **tCL** *(CAS Latency)* – várakozás az oszlopcím kiadása után az első adat megjelenéséig
- **tRCD** *(RAS to CAS Delay)* – sor megnyitása és az oszlopcím kiválasztási parancs között
- **tRAS** *(Row Active Time)* – minimális idő amíg egy sor nyitva kell maradjon
- **tRP** *(Row Precharge)* – várakozási idő egy sor lezárása után új sor nyitásáig
- **tRC** *(Row Cycle Time)* – minimális idő két azonos bankbeli soraktiválás között

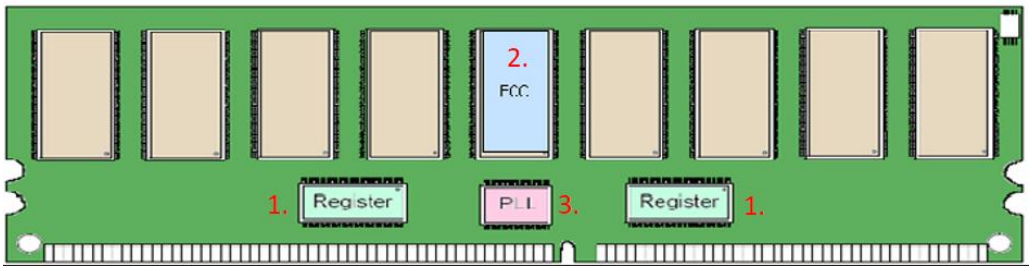
Teljes olvasási ciklus menete:

1. Bank (sor) megnyitása
2. Oszlopadatok olvasása megnyitott sorból *kihalt, nem fontos*
3. Bank lezárása (precharge)
4. Legalább tRP várakozás újra nyitás előtt

gyakori

8.4. DIMM-EK JELLEMZŐI

A DIMM (Dual Inline Memory Module) DRAM chipeket tartalmazó memória modul tipikusan 64 bites adatúttal, 168-288 érintkezővel.



8.4.1. Registered DIMM:

gyakori

- A memóriachip és a memóriavezérlő közé regiszter kerül
- Csökkenti az elektromos terhelést a vezérlőn
- Egy órajelnyi késleltetést okoz
- Nagyobb stabilitás sok modul esetén

Tipikus felhasználás: szerverek

8.4.2. ECC DIMM:

gyakori

- Extra (9.) DRAM chip az ECC/paritás bitekhez
- **Paritásbit**: egybites hiba felismerése, javítani nem tud
  többszörös hibát sem tudja konzisztensen felfedezni
- **ECC bit**:
  - Egybites hiba felfedezése és javítása
  - Több, egymás melletti bithiba felismerése és javítása
  - Kijavíthatatlan hiba esetén rendszerleállítás és naplózás

8.4.3. PLL (Phase Locked Loop):

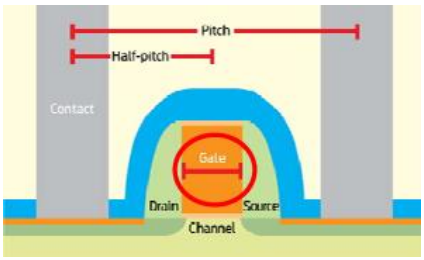
gyakori

- **Fáziszárt hurok**: órajel elcsúszás mentesítése
- Órajel szinkronizálás
- Stabilabb, pontosabb működés nagy frekvencián

9. TRANZISZTOROK

gyakori, általában mindegyik 1-1 mondat + előny előzőhöz képest

9.1. LEGFONTOSABB JELLEMZŐK



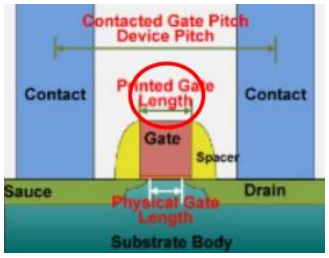
Mi az a tranzisztor?

A tranzisztor egy **félvezető alapú aktív elem**, amelyet digitális áramkörökben leggyakrabban **kapcsolóként** használunk. Feladata az áram áramlásának szabályozása a bemeneti feszültség segítségével.

- **Source (forrás)**: Az a terület, ahonnan a töltéshordozók (elektronok) elindulnak.
- **Drain (nyelő)**: Az a terület, ahová a töltéshordozók érkeznek.
- **Gate (kapu)**: A vezérlő elektróda, amely a csatorna vezetőképességét szabályozza.

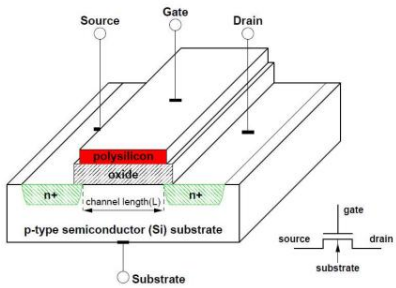
Legfontosabb méretek:

- **Pitch**: A tranzisztor teljes szélessége; minél kisebb, annál több fér egymás mellé.
- **Gate length**
  - **Elméleti hossz:** A kapu fizikai szélessége a gyártási terv szerint.
  - **Fizikai hossz:** A tényleges csatornahossz a source és drain között. Figyelembe veszi, hogy a source és a drain részben „belenyúlnak” a gate alá.



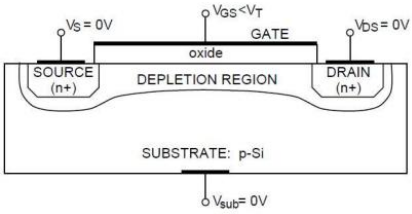
9.2. nMOSFET *Metal-Oxide-Semiconductor Field-Effect Transistor*

Felépítése (nMOS esetén): *gyakori + rajz*



- **Substrate**: p-típusú szilícium lapka
- **Source/Drain**: Erősen doppingolt (n^+) negatív szigetek a p-típusú substrate-től
- **Szigetelő (Oxid)**: Egy vékony szilícium-dioxid (SiO_2) réteg választja el a kaput a substrate-től

Hogyan működik egy hagyományos MOSFET tranzisztor (nMOS)



1. **Kikapcsolt állapot**: Alaphelyzetben a p-típusú hordozóban nincs elegendő szabad elektron, így nem folyik áram a source és a drain között ($V_{GS} < V_T$)
2. **Bekapcsolás**: Ha a kapura pozitív feszültséget kapcsolunk, az elektromos teret hoz létre
3. **Csatorna kialakulása**: Ez a tér eltaszítja a pozitív lyukakat és oda vonzza az elektronokat a kapu alá, létrehozva egy vezető csatornát .Ezen keresztül a töltések áramolni tudnak a source-tól a drain felé.

A Kapu és a Substrate közötti oxid réteg azt akadályozza meg, hogy a Gate-ből töltések kerüljenek a csatornába. (A kapu csak az elektromos mezőhöz szükséges feszültséget biztosítja, töltést nem)

9.3. FESZÍTETT SZILÍCIUM TECHNOLÓGIA

gyakori
1 mondat

Lényege: Megfelelő gyártási eljárással a szilícium atomok közötti távolságot megnövelik.

Előnye: Nő az elektronok és a lyukak mozgékonysága, ami 10-20%-os teljesítményjavulást eredményez.

Hátránya: Több lépésből áll a gyártási folyamat így az összege is.

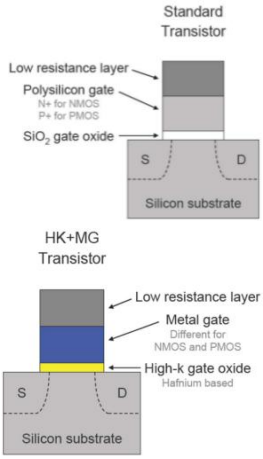
9.4. HKMG

gyakori + rajz

A probléma: Ahogy a tranzisztorok zsugorodtak, a SiO2 szigetelőréteg is vékonyodott (1nm alá). Ez hatalmas szivárgási áramhoz és disszipációhoz vezetett.

Megoldás: A SiO2 réteget lecserélték egy **High-k** (nagy dielektromos állandójú) anyagra és **fémkapura**.

- Csökkent a bekapcsoláshoz szükséges áram (-30%)
- Gyorsabban lehet a tranzisztort kapcsolni (+20%)
- Csökkent a szivárgás az oxid réteg (5x) és a source-drain között (10x)



9.4.1. Kapacitás képlete

- **A**: a kondenzátor felülete
- **t**: a szigetelő vastagsága
- **ε_0**: a vákuum permittivitása
- **κ**: relatív dielektromos állandó *(SiO_2 -nél 3,9, High-k-nél > 10)*

$$C = \frac{\kappa \varepsilon_0 A}{t}$$

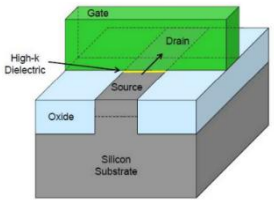
***permittivitás**: egy anyagi tulajdonság, ami megmutatja, hogy az anyag milyen mértékben képes elektromos energiát tárolni és polarizálódni*

gyakori + rajz

9.5. FINFET

más néven:3D vagy tri-gate tranzisztor

Ezt megelőző Planar (2D-s) tranzisztoroknál a Source és a Drain közötti hidat a kapu alatt alakították ki.

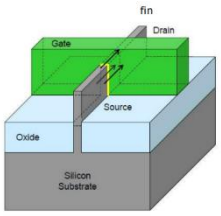


2D tranzisztor

FinFET (3D tranzisztor) *fin = fésűfog = uszony*

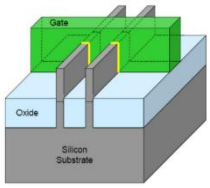
Alapvető különbség: Ez nem síkbeli, hanem **3D-s struktúra**. A csatorna egy függőleges „fésűfog" (fin), amelyet a kapu **három oldalról** vesz körbe.

A FinFET tranzisztor esetén a Substrate réteg kiemelkedik és "belehatol" a kapuba és 3 irányból veszi közre a kaput. Ha a kapu "engedélyt" ad az elektronok áramlására, akkor a "híd" egy sokkal nagyobb felületen tud kialakulni.

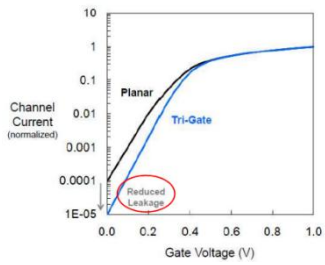


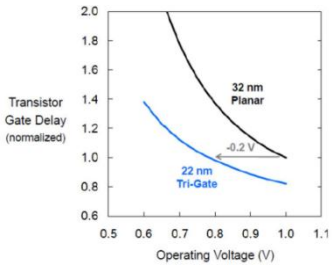
3D „fésűfogas" tranzisztor

Több fésűfog: A teljesítmény növelése érdekében több fésűfogat (fin) is össze lehet kapcsolni egy tranzisztoron belül.



- fokozott teljesítmény
- jobb energiahatékonyság





A 2D tranzisztorok magasabb szivárgási áramával szemben a Tri-Gate (3D) technológia meredekebb görbéje gyorsabb váltást tesz lehetővé az be- és kikapcsolt állapotok között.

Gate delay: A ki- és bemeneti jel közti időkülönbség.

Hatékonyság: A Tri-Gate alacsonyabb feszültségen is stabil, így több mint 50% aktív teljesítményt takarít meg.

9.5.1. 2. generációs FinFET

külön nem nagyon

A 2. generációs FinFET a fésűfogak (fin) magasságának növelésével (34nm → 42nm) és a sűrűség javításával fejlődött tovább. A kevesebb, de hatékonyabb fin alkalmazása csökkenti a kapacitást, miközben nagyobb tranzisztor sűrűséget tesz lehetővé.

9.6. GAAFET

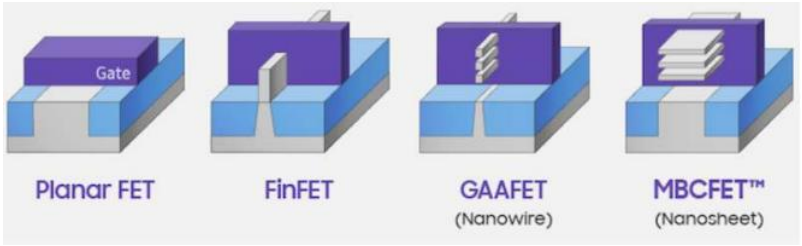
külön nem nagyon

Több áramlási rétegünk van kis szalagokban. Ezt veszi körbe a gate és így mikor megengedjük az áramlást akkor 4 irányból tud áramolni az elektron **(ez a nanowire)**

9.6.1. GAAFET vs MBCFET

Ezeknek a lapoknak/szalagoknak a szélességét, ha növeljük, akkor nanowire helyett nanosheet lesz

Az MBCFET-nél a lapok szélessége változtatható, így nagyobb áramerősséget és **jobb teljesítményt** lehet elérni ugyanazon a feszültségen.



10. PÁRHUZAMOS ARCHITEKTÚRÁK

A modern processzorok teljesítményét ma már nem csupán az órajel, hanem a párhuzamosítás mértéke határozza meg. Míg egy asztali CPU 10-20 teraFLOPS (10^{13} művelet másodpercenként) teljesítményre képes, a szuperszámítógépek már a 10^{17} FLOPS tartományban mozognak.

10.1. PÁRHUZAMOSSÁG TÍPUSAI, SZINTJEI ÉS OSZTÁLYOZÁSA

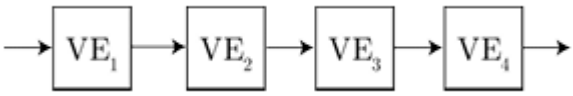
10.1.1. Párhuzamosság típusai

Funkció szerint: *„Lehetőség”* *ritka, típusok mire törekedtek*

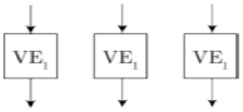
- **Rendelkezésre álló párhuzamosság**: Magában a feladatban vagy a programkódban rejlő elvi lehetőség, független a konkrét hardvertől *(elméletileg mennyire lehetne párhuzamosítani)*
- **Kihasználható párhuzamosság**: Amit az adott architektúra a végrehajtás során ténylegesen képes megvalósítani. *(gyakorlatban valójában mit lehet megvalósítani)*

Elhelyezkedés szerint: *„Geometria”*

- **Időbeli párhuzamosság**: Több végrehajtó egység működik időben elcsúsztatva (például futószalag/pipeline), jellemzően nem ugyanazon a részfeladaton. *(pl autó gyártósor)*
- **Térbeli párhuzamosság**: Több, azonos típusú végrehajtó egység egy időben ugyanazt a művelettípust végzi. *(pl 3 gyártó sorunk van, akkor egy adott műveletet egyszerre 3 helyen végez)*



időbeli párhuzamosság



térbeli párhuzamosság

Típus szerint: *„Munkadarab” Mit sokszorozunk meg: az alapanyagot vagy a munkafolyamatot?*

- **Adatpárhuzamosság**: két megvalósítás *(pl 100 számot kellene megszoroznod kettővel: ahelyett, hogy egymás után csinálnád, kiosztod a számokat 100 embernek, és mindenki egyszerre szoroz.)*
  - **Adatpárhuzamos architektúrák**: Adat-elemeken párhuzamos vagy futószalag elvű műveletek végrehajtását teszik lehetővé
  - **Átalakítás funkcionális párhuzamossággá**: Az adat-elemeken végrehajtható műveletek ciklusok formájában történő megfogalmazása *(pl for i = 1 to 100)*
- **Funkcionális párhuzamosság**: A feladat logikájából következő párhuzamosság, különböző műveletek függetlenül végezhetők (pl folyamatábra, adatfolyamgráf …)
  Példa: egy programban, fájl beolvasás, hálózat kérés, logolás – ha nincs adatfüggés egyszerre futhatnak.

10.1.2. Funkcionális programozás szintjei

ritka

A funkcionális párhuzamosságot különböző mélységekben és „szemcsézettségi” szinteken értelmezzük.

A párhuzamosság szintjei: *legkisebbtől a legnagyobbig*

- **Utasítás szint (ILP)**: Programutasítások párhuzamos végrehajtása
- **Ciklus szint**: Egymást követő iterációk párhuzamosítása (függőségek akadályozhatják)
- **Eljárás szint**: Párhuzamosan végrehajtható eljárások formájában jelenik meg, mértéke a feladat jellegétől függ
- **Program szint**: Egymástól független programok párhuzamos futtatása
- **Felhasználó szint**: Több független felhasználó egyidejű kiszolgálása (pl szerverek)

kevésbé ritka

10.1.3. Rendelkezésre álló párhuzamosságok hasznosítása

A számítások felgyorsítása érdekében az architektúrák, az operációs rendszerek és a fordítóprogramok (Compiler-ek) egyaránt törekednek a rendelkezésre álló párhuzamosságok hasznosítására.

Utasítás szintű: Párhuzamos architektúra (ILP) vagy megfelelő fordítóprogram segítségével

Ciklus vagy eljárás szintű: Szálak vagy folyamatok formájában. A szál/folyamat a tárgykód legkisebb önállóan végrehajtható része. Létrehozhatja:

- **Programozó**: Párhuzamos nyelvekkel (pl. FORK, JOIN utasítások).
- **Operációs rendszer**: Többszálas/többfeladatos támogatással
- **Párhuzamos fordító**: Magas szintű nyelvek esetén

Program és felhasználói szint: Párhuzamos rendszerek, megfelelő hardver- és szoftvertámogatással

10.1.4. Szemcsézettség

külön nem volt még

A féladatok mérete alapján két fő kategóriát különítünk el:

- **Finom szemcsézettség** (Alacsony szint): Utasítás szint, szál szint és folyamat szint
- **Durva szemcsézettség** (Magas szint): Felhasználói szint

Az alacsony szintű párhuzamosságot párhuzamos architektúrákkal vagy compilerekkel, a magasabb szintűt többszálas/többfeladatos OS-ek alatti konkurens végrehajtással lehet kihasználni.

ritka

10.1.5. Compiler

A fordítóprogram, ami lefordítja a magas szintű programnyelvben írt programot a processzor számára érthető formátumra. Kulcsszerepet játszik a párhuzamosság kihasználásában.

1. **Analízis**: Lexikális (konstansok, változók, operátorok), szintaktikai és szemantikai elemzés.
2. **Szintetizálás**: Tárgykód (gépi kód vagy assembly) generálása és **kódoptimalizálás**, amely során a fordító független, párhuzamosan futtatható részeket keres.

10.1.6. Flynn-féle osztályozás

gyakori

Négy féle fogalmat vezet be:

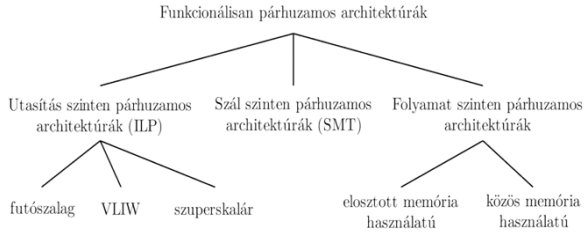
- **SI (single instruction stream):** Egyszeres utasításfolyam; a gépnek egyetlen "agya" (vezérlőegysége) van, ami mondja, mit kell tenni.
- **MI (multiple instruction stream)**: Többszörös utasításfolyam; a gép egyszerre több különböző utasításfolyamot tud egyidőben végrehajtani.
- **SD (single data stream)**: Egyszeres adatfolyam; egyszerre csak egy adaton (pl. egy számon) végzünk műveletet
- **MD (multiple data stream)**: Többszörös adatfolyam; egyszerre sok, független adaton dolgozik a gép

SISD	SIMD	MISD	MIMD
Egy parancs, egy adat	Egy parancs, sok adat	Sok parancs, egy adat	Sok parancs, sok adat
Neumann- modell (soros végrehajtás)	Multimédiás feldolgozás: ugyan azt a műveletet végezzük el rengeteg adaton (pl. kép színe)	Elméleti kategória	Teljes párhuzamos feldolgozás: minden egység mást csinál más adaton

10.1.7. Modern osztályozás

gyakori, jobb oldali ábra!
ritka bal oldali ábra
elvileg nem kéri

Figyelembe veszi a párhuzamosság fajtáját, szintjét és módját is.



10.2. PÁRHUZAMOS ARCHITEKTÚRÁK MŰKÖDÉSE

Az utasítás végrehajtás elemi műveletei:

1. **F (Fetch)**: Az utasítás lehívása a memóriából
2. **D (Decode)**: Az utasítás értelmezése (dekódolás)
    - **S/O (Source operand)**: A művelethez szükséges forrás operandusok lehívása
3. **E (Execute)**: A tényleges végrehajtás ALU-ban
4. **W/B (Writeback)**: Az eredmény visszaírása a regiszterbe vagy memóriába

A modern processzoroknál az utasítások végrehajtása és azok kibocsátása szorosan összefügg. Minél több végrehajtó egységünk van, annál több „munkát” (utasítást) kell tudnunk rájuk bízni minden egyes pillanatban.

10.2.1. Kibocsátási párhuzamosság

kevésbé ritka

Definíció: Kibocsátási párhuzamosságnak nevezzük, ha a CPU a dekódoló egysége egyetlen óraciklus alatt egynél több utasítást képes tovább küldeni végrehajtásra.

A kibocsátás az a folyamat, amikor a dekódolt utasítások átkerülnek a végrehajtó egységbe. Ha a CPU képes párhuzamosan több utasítás végrehajtására, a kibocsátási kapacitást növelni kell.

Szuperskalár processzorok: Azokat a processzorok amelyek képesek erre.

10.2.2. ILP CPU általános követelményei

kevésbé ritka, előzővel együtt szokta kérni

Minden olyan processzornak, amely **utasítás szintű párhuzamosságot** (ILP) alkalmaz, két szigorú szabálynak kell megfelelnie:

1. **Függőségek kezelése**: A processzornak figyelnie kell az utasítások közti függőségeket
   Pl. ne használjon fel egy adatot, mielőtt az előző művelet kiszámolta volna
2. **Soros végrehajtás konzisztenciája**: A programnak a párhuzamos végrehajtás mellett is pontosan úgy kell viselkednie, mintha a programozó által megírt sorrendben egymás után futottak volna le az utasítások.

10.3. FÜGGŐSÉGEK

Egy programban az egymást követő utasítások függhetnek egymástól. A függőségek akadályozzák a párhuzamos végrehajtás teljesítmény növelésének maximalizálását, valamint gátolhatják a párhuzamos adat vagy utasítás végrehajtást. *(A+B = C és D = C * 10 , D kiszámításához szükség van C-re)*

Futószalag példa:

Fokozatok számának növelésével nem sokszorozható végtelenségig a teljesítmény:

- Az egyes fokozatok gyakorlatban különböző időt igényelnek → komoly időveszteségek
- Függőségek miatt

Bizonyos határig növekszik a teljesítmény, további növeléssel csökkenést eredményez.

Függőségek 3 fő csoportja:

- Adatfüggőség
- Vezérlésfüggőség
- Erőforrásfüggőség

gyakori + műveleti/lehívási adatfüggőség

10.3.1. Adatfüggőség (RAW, WAR, WAW, Ciklusbeli)

Akkor lépnek fel, ha az utasítások ugyanazt az adatot (regiszter/memória) használják.

Csoportosítása:

- **Valós adatfüggőség** (RAW – Read After Write)
  - műveleti adatfüggőség
  - lehívási adatfüggőség
- **Ál adatfüggőség**
  - WAR (Write After Read)
  - WAW (Write After Write)
- **Ciklusbeli adatfüggőség**

csoportosítás kevésbé ritka

RAW, WAR, WAW gyakori

Valós adatfüggőség – műveleti adatfüggőség (RAW)

Probléma: Az I_2 utasítás olvasni akar egy adatot, amit az I_1 még nem írt vissza.

Példa: Két számot össze akarunk szorozni, majd az eredményeket össze akarjuk adni. *(r3 regiszter)*

	t_1	t_2	t_3	t_4	t_5
I_1	F_{MUL}	$D + S/O_{r1,r2}$	E_{MUL}	W/B_{r3}	
I_2		F_{SHL}	$D + S/O_{r3}$	E_{SHL}	W/B_{r3}

I1 MUL r3, r2, r1
I2 SHL r3

4 fokozatú futószalag: Fetch → Decode (+ Source Operandus lehívás) → Execute → Write Back (W/B)

Kezelése:

1. **NOP (No Operand)** várakozó ciklusok beiktatása (lassítja a végrehajtást)

	t_1	t_2	t_3	t_4	t_5	t_6	t_7
I_1	F_{MUL}	$D + S/O_{r1,r2}$	E_{MUL}	W/B_{r3}			
I_2		F_{SHL}	NOP	NOP	$D + S/O_{r3}$	E_{SHL}	W/B_{r3}

   Következménye: két óraciklussal hosszabb ideig tart az utasítás végrehajtása
   → minden utána következő utasítás is csúszni fog

2. **Operandus előrehozás (Forwarding)**:Extra hardverrel az eredményt az ALU végéről azonnal visszavezetik a bemenetre, megspórolva a várakozást

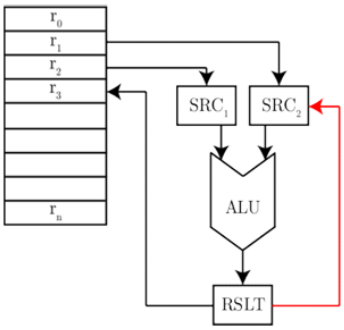
   Az eredményt a regiszterbe ugyan úgy vissza kell írni!

Valós adatfüggőség – lehívási adatfüggőség (RAW)

Az operandusokat végrehajtás előtt az adatcache/operatív tár-ból a regiszterekbe kell tölteni.

Probléma: Ha az adat nincs a regiszterben, a betöltése időigényes, ami várakozásra kényszeríti az ALU-t

Kezelés: Betöltési operandus előre hozása: extra hardverrel az adatot betöltéskor a regiszter mellett *közvetlenül* az ALU bemenetére is továbbítják.



Következmény: Legalább 1 óraciklus nyereség a végrehajtási időben.

Ál adatfüggőségek: Ezek csak a regiszterek nevei miatti ütközések.

gyakori

1. WAR (Write After Read)

Az írás hamarabb történne meg, mint az előző utasítás olvasása.

I_1	MUL r3 r2 r1	r1 és r2 értékét szorozzuk és beírjuk r3-ba
I_2	ADD r2 r4 r5	r2-be írjuk az r4 és r5 összeadás eredményét

Előfordulhat, hogy az I_2 ADD utasítás eredménye hamarabb írja felül az r2 eredményét, mint a megelőző utasítás operandusainak beolvasása *(S/O)* megtörténne.

Megoldás: regiszter átnevezés

I_1	MUL r3 r2 r1	Az r23 → r2 hozzárendelést nyilván kell tartani, majd amikor
I_2	ADD r23 r4 r5	a MUL utasítás végzett, vissza kell írni r23 tartalmát r2-be.

gyakori

2. WAW (Write After Write)

Két írás körül a későbbi logikai utasítás (pl. egy gyorsabb ADD) hamarabb írná felül a regisztert, mint a korábbi (pl. egy lassabb MUL). *– gyakoribb függőség*

I_1	MUL r3 r2 r1	r1 és r2 értékét szorozzuk és beírjuk r3-ba
I_2	ADD r3 r4 r5	ugyan úgy r3-ba írjuk az r4 és r5 összegét

Előfordulhat, hogy az I_1 *ADD* utasítás előbb fut le mint az I_2 *MUL*, így az I_1 utasítás felülírja I2 eredményét. → **Sérül a szekvenciális konzisztencia.**

Megoldás: regiszter átnevezés

Átnevezési (vagy piszkozat) regiszterek tulajdonságai:

- új, önálló regiszter
- saját címtartománnyal rendelkezik
- programozó számára transzparens
- extra hardvernek számít.

Ezáltal két féle regiszterkészletet különböztetünk meg:

1. **Architektúrális regiszterkészlet** - *ezeket látja a programozó (pl: AX, BX ... regiszterek)*
2. **Átnevezési *regiszterkészlet*** - *a vezérlés használja az ál függőségek kiküszöbölésére*

külön nem, max 1 mondat csoportosításnál

Ciklusbeli függőség

Probléma: Az aktuális iteráció kiszámításához szükség van a közvetlenül megelőző iteráció végeredményére, így a körök nem futtathatók párhuzamosan.

for *i=2 to n* **do**
$X_i \leftarrow A_i * X_{i-1} + B_i$
end for

Kezelés: Ez egy erős függőség, hardveresen nehezen feloldható.
Megoldás: az algoritmus áttervezése.

10.3.2. Vezérlésfüggőség (feltétel nélküli, feltételes)

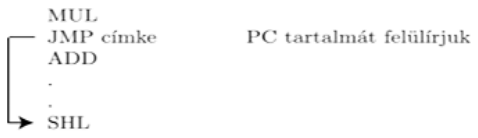
gyakori

Feltételes vagy feltétel nélküli elágazásnál léphet fel.

Feltétel nélküli elágazás (JMP)

Probléma: Az ugrás parancs későn állítja be PC-t, addigra már a futószalag végrehajtás miatt a következő parancs (vagy akár több) lehívásra került vagy akár le is futott → hibás működés.

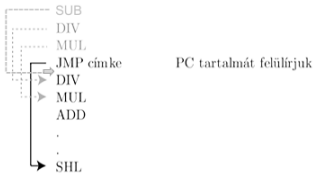
	t_1	t_2	t_3	t_4	t_5	t_6
I_1	F_{MUL}	D	E	W/B		
I_2		F_{JMP}	D	E	W/B	
I_3			F_{ADD}	D	E	W/B



Kezelési módszerek (Nincs megoldás, csak kezelni lehet): *statikus, dinamikus vagy spekulatív*

1. **Ugrási buborék**: A JMP utasítás mögé egy vagy több NOP utasítás kerül be, ezzel lassítva a futószalagot, míg elő nem áll az ugrási cím. → csökken a teljesítmény
2. **Utasítások átrendezése**: optimalizáló compiler segítségével.

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
I_1	F_{MUL}	D	E	W/B				
I_2		F_{JMP}	D	E	W/B			
I_3			NOP	NOP	NOP	F_{SHL}	D	E



Feltételes elágazás

Kezelése: ma már csak dinamikusan a végrehajtás során történhet, hiszen a feltétel teljesülésétől vagy NEM teljesülésétől függ, hogy ugrás vagy soros folytatás következik -e.

elvileg nem fontos

10.3.3. Erőforrásfüggőség

Definíció: Akkor lép fel, ha több utasítás ugyanazt a hardverelemet (pl. regisztert vagy végrehajtó egységet) akarja egyszerre használni . Ilyenkor az egyik utasítás elsőbbséget élvez, a többit várakoztatni kell.

Példa: Ha egy programban túl sok lebegőpontos művelet van, a dedikált lebegőpontos egységnél sorban állás alakul ki (szűk keresztmetszet), miközben a fixpontos egység kihasználatlanul várakozik.

Kezelése: Úgy kell megtervezni a rendszert, hogy az erőforrások ne okozzanak szűk keresztmetszetet (bottleneck) azáltal, hogy a bizonyos regiszterekből vagy végrehajtó egységekből többet építünk be a rendszerbe. A cél a kiegyensúlyozott, kb. 70-80%-os kihasználtság elérése.

10.4. A SZEKVENCIÁLIS (SOROS) KONZISZTENCIA MEGŐRZÉSE

A program szekvenciális logika szerint íródik. A hardver és a fordító párhuzamosíthat, de a programozó által elvárt sorrendi hatás nem sérülhet.

Két érintett terület: - normál utasításfeldolgozás *gyakori milyen esetekben sérülhet*
- kivételkezelés (megszakítás)

10.4.1. Utasítás feldolgozás soros konzisztenciája

A párhuzamos vagy sorrenden kívüli végrehajtás **nem változtathatja** meg azt, hogy egy utasítás logikailag melyik előző utasítás eredményét használja.

Probléma: Gyorsabb utasítás előbb fejeződik be, mint egy előtte álló lassabb → feltételes ugrás rossz eredmény alapján dönt *(JZ mindig legutoljára végzett utasítás eredményét használja)*.

```
I1    DIV r3 r2 r1
I2    ADD r5 r6 r7  ⟶  gyorsabb
I3    JZ címke        ha az eredmény 0, akkor ugrás
```

A párhuzamos végrehajtás bevezetése megsértheti a soros végrehajtás logikáját!

Megoldás: Hardveres biztosítás (pl. külön állapotjelzők, belső követés), hogy a feltételes utasítás csak a közvetlen előző utasítás eredményét lássa.

gyakori

10.4.2. A kivétel-kezelés soros konzisztenciája

A megszakításokat az utasítások eredeti programsorrendjében kell kezelni, akkor is, ha a végrehajtás párhuzamos.

Pontatlan kivételkezelés (gyenge konzisztencia) – *korai szuperskalár processzorok*

A CPU azonnal elfogadja a megszakítást, független attól, hogy az előző utasítások befejeződtek-e

```
I1    MUL r3 r2 r1
I2    ADD r5 r6 r7    túlcsordul és kér egy megszakítást
I3    JZ címke        ha az eredmény 0, akkor ugrás
```

Az ADD utasítás előbb fut le, de túlcsordulás miatt megszakítást kér.

Ha egy gyorsabb utasítás hibája miatt a CPU azonnal menti a kontextust, nem tudható, hogy a logikailag korábbi, lassabb művelet befejeződött-e.
→ Ez **definiálatlan regiszterállapotot** és súlyos működési hibákat okoz.

Pontos kivételkezelés (erős konzisztencia, precíz megszakítás)

Megszakítás csak akkor fogadható el, ha minden korábbi utasítás:

- befejeződött
- vagy szintén megszakítást kér

Eredmény: A mentett állapot mindig egy valós, szekvenciális programsorrendi állapot.

Megvalósítás:

- **átrendező puffer** (ROB – Reorder Buffer) – *később lesz róla szó*
- **címkézés**: utasítások sorszámozása, csak akkor fogadjuk el, ha egyetlen megelőző sorszámú sem kért.

11. FUTÓSZALAG PROCESSZOROK

A futószalag (pipeline) **lényege az időbeli párhuzamosság**: az utasításokat több részfeladatra bontjuk (Fetch, Decode, Execute, Writeback), amelyeket külön egységek párhuzamosan végeznek.

Gondolat: Egy *n* fokozatú futószalag elméletileg **n-szeres** sebességnövekedést jelent, ha minden fokozat azonos ideig tart.

Gátló tényezők: A függőségek (adat, vezérlés, erőforrás) miatt a gyakorlati gyorsulás kisebb.

Optimális mélység: Általában 15-30 fokozat. Efelett a függőségek kezelése rontja a teljesítményt. Speciális esetben: superpipeline (akár 200 fokozat)

11.1. 2 FOKOZATOS IDEÁLIS FUTÓSZALAG

gyakori

Ahhoz, hogy a futószalag tökéletesen működjön, az alábbi feltételek kellenek:

- 2 db egymástól teljesen független végrehajtó egységgel rendelkezik
- Egyik fokozat kimenete a másik fokozat bemenete kell legyen
- Mindkét fokozat **végrehajtási ideje azonos**
- Órajelre szinkronizáltan 1 óraciklus alatt elvégzik a feladatukat

T: szekvenciális végrehajtási idő
t: futószalagos végrehajtási idő

$$\mathbf{t = \frac{T}{2}}$$

Függőségek kezelése:

- Operandus előrehozás – *minden architektúránál használjuk*
- Újrafeldolgozás (execute többször egymás után) – *pl szorzás: összeadások sorozata*

nem szokta kérni

11.2. FUTÓSZALAGOK TÍPUSAI

1. **Előlehívás (overlapping)**: Az aktuális utasítás *(I_1)* visszaírásával *(W/B)* egy időben hívjuk le a következőt *(I_2 F)*. Kevés gyorsítást ad, de nincsenek függőségi problémák.
2. **Vektor CPU**: Csak a végrehajtási fázis (Execute) van futószalagosítva
3. **Teljes pipeline**: A teljes folyamat minden fázisa átfedésben van.
4. **Logikai futószalagok**: A CPU-n belül különböző feladatokra külön futószalagok
    - **Aritmetikai (FX/FP)** **F, DS/O, E, W/B**
    - **Ugró (Branch)** **F, E**
    - **LOAD/STORE**

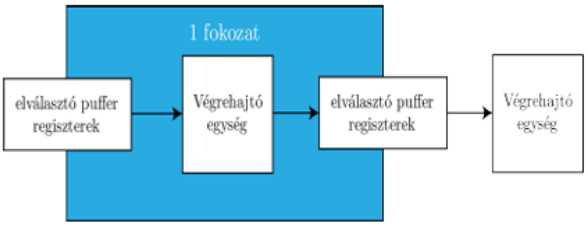
11.3. FUTÓSZALAGOK FIZIKAI MEGVALÓSÍTÁSA

Alkalmazásuk alapján 2 csoportjuk van:

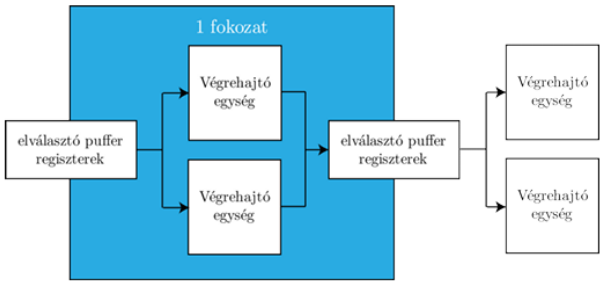
- **Univerzális**: Mindenre jó, viszont bonyolult, drága, lassú.
- **Dedikált futószalag**: Adott műveletre specializált, amik kevesebb logikai kaput igényelnek mint egy univerzális, így gyorsabbak is *(Pl: Branch, LOAD/STORE, FX, FP)*

A futószalag sebességét **mindig a leglassabb fokozat határozza meg**, ezért a cél, hogy minden fokozat ugyanolyan gyors legyen. → ennek megoldására **fokozatok előtt és után is elválasztó puffer** regiszterek kerültek.

A bemenetek és kimenetek ilyen regiszterekbe töltődnek be és itt tárolódnak, amíg végrehajtó egységek dolgoznak.



Térbeli párhuzamosítás során, mikor egymás mellé több végrehajtó egységet helyeztek vezérlőjelek segítségével határozzák meg, hogy a puffer regiszterekből melyikbe töltődjön az adat.



11.3.1. Futószalagos feldolgozás következményei:

ritka

Következmények:

- **Gyorsulás**: Jelentősen felgyorsult az utasítás lehívás és az operandus betöltés.
- **Függőségek megjelenése**: kritikus jelentőségűvé vált a különböző típusú (adat, vezérlés, erőforrás) függőségek kezelése
- **Maximális végrehajtás: 1 utasítás/ciklus**; efelett már kibocsátási vagy utasításon belüli párhuzamosság szükséges

Problémák és megoldások: *ritka*

1. A processzor sebessége gyorsabban nőtt mint a memóriáé → **Cache bevezetése**
2. **Elágazáskezelés fejlődése** (Vezérlési függőség):
    - **Feltétlen elágazás (korai RISC)**: ugrási buborék használata, blokkolja az utasítás lehívását
    - **Feltételes elágazás (korai RISC)**: minimum +2 óraciklus (feltétel kiértékelés, ugrási címszámítás)
    - **Korai CISC**: Nem használtak buborékot. A **dekódoló fokozatba építették** a címszámítót és a komparátort, így a dekódolás végére meglett az ugrási cím.
    - **Későbbi CISC (és 1. gen szuperskalár)**: **Fix előrejelzés** (pl. "mindig ugrik"). Az ugrási címen lévő utasításokat elkezdik lehívni; ha mégsem kell ugrani, azokat visszatörlik (**visszatörlés/flush**).
        - **Előnye**: Nincs blokkolás a feltétel megismeréséig
        - **Korlátja**: Ha a feltételben nagy látenciájú művelet (pl. osztás) van az blokkolja a kibocsátást
3. **II. generációs szuperskalár megoldás**
    - Elődekódolás: A CPU a feladatok egy részét már akkor elvégzi, amikor az utasítás bekerül az L1 utasítás cache-be.
    - Spekulatív elágazásbecslés

gyakori

11.4. RISC ÉS CISC ARCHITEKTÚRÁK

A processzorok tervezési stratégiája az utasításkészlet alapján két nagy csoportra osztható.

11.4.1. RISC architektúrák (Reduced Instruction Set Computing)

- **Utasítások**: Kevés (50-150) → egyszerű címzési módok,
  azonos hosszúságú utasítások (pl. 64/128 bit)
- **Műveletvégzés**: Kizárólag regisztereken végez műveletet; memória és cache elérés csak LOAD/STORE utasítással érhető el (aritmetikával nem kombinálható)
- **Felépítés**: **Sok regiszter**, **huzalozott** (hardveres) dekódolás
- **Végrehajtás**: **3 operandusos** *(eredmény nem ír felül),* cél az **1 óraciklusos** futás
- **Szoftver**: Bonyolultabb fordítóprogram (compiler)

Előnyök: alacsonyabb fogyasztás, gyorsabb utasítás szintű végrehajtás
Hátrányok: bonyolultabb feladatokat instrukció szekvenciákkal kell megoldani, kisebb kompatibilitás, kisebb teljesítmény ugyanazon a frekvencián (x86-64-hez képest)

11.4.2. CISC architektúrák (Complex Instruction Set Computing)

gyakori + miért hosszabb a CISC pipeline mint a RISC

- **Utasítások**:
  - nagy számú utasítás készlet (több száz),
  - változó hosszúságú, akár összetett utasítások → dekódolónak nem csak dekódolnia kell az utasítást, hanem azonosítani a végét (utasítás határra illesztés), plusz hardvert és időt igényel.
  - Egy utasítás több elemi műveletet is végre tud hajtani
- **Műveletvégzés**: Közvetlen memóriaelérés lehetséges (2. operandus lehet memória/cache)
- **Felépítés**: Nagy belső **mikroprogramtár**, **kevesebb regiszter**
- **Végrehajtás**: **2 operandusos** *(eredmény felülírja az elsőt)* – első operandus nem lehet memória vagy cache cím
  Utasítások feldolgozása több ciklusidő lehet → bonyolultabb feldolgozás
- **Futószalag**: Általában +2 fokozat (AG – címszámítás, cache elérés), sebességkülönbség esetén **Interlock** várakoztatás
- **Szoftver**: Egyszerűbb fordítóprogram és gépi kódú programozás
- **Extrák**: Régi programokkal kompatibilis (utasítások folyamatosan bővültek), HT (többszálúság) és virtualizáció támogatása

Előnyök: egyszerűbb compilerek, széles termékskála, kompatibilitás, nagy nyers teljesítmény
Hátrányok: komplex hardver, lassabb dekódolás, magasabb energiafogyasztás

***Interlock**: várakoztat a fokozatok között, hogy megfelelő sorrendben hatódjanak végre*

CISC: ADD [100], [102] - egy utasításban a betöltés, a művelet, és a visszaírás!

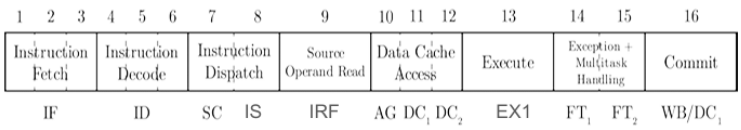
RISC: LOAD AX, [100]
LOAD BX, [102]
ADD CX, AX, BX
STORE [100], CX

- sok utasítás, de egyszerű utasítások, ezért a dekódolás is egyszerűbb, gyorsabb és kevesebb tranzisztor kell hozzá!
  ⟶ több regisztert lehet kialakítani!

CISC pipelineba pluszban kell:

gyakori, miért hosszabb a CISC pipeline mint a RISC

- AG – címszámítás
- DC1-2 – Data cache



Ábra nem fontos, csak példa

11.4.3. Hibrid architektúrák

külön nem kéri

Lényege: kívülről CISC (kompatibilitás), belül **RISC mag** (gyorsaság).

80/20 szabály: Az utasítások 20%-át használjuk az idő 80%-ában – a hibrid megoldás ezeket a gyakori utasításokat gyorsítja fel a belső RISC maggal.

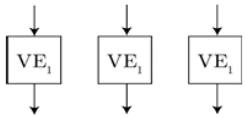
12. SZUPERSKALÁR ARCHITEKTÚRÁK

A futószalagos (pipeline) architektúrák korlátja: **1 utasítás / órajel**.
teljesítménynövelés iránya: **időbeli + térbeli párhuzamosság → szuperskalár architektúra**

12.1. KÖZÖS JELLEMZŐK

gyakori

- **Párhuzamos kibocsátás**: dekódoló egységből egy óraciklusban több utasítás kibocsátás
  - 1. gen: 2-3 utasítás / ciklus
  - 2. gen: 4-6 utasítás / ciklus
- **Időbeli + térbeli párhuzamosság**: több futószalag párhuzamosan.
- **Maguk küzdenek meg a függőségekkel**: dinamikusan, extra hardverek segítségével
- **Kompatibilitás**: régi programok is futtathatóak maradtak.



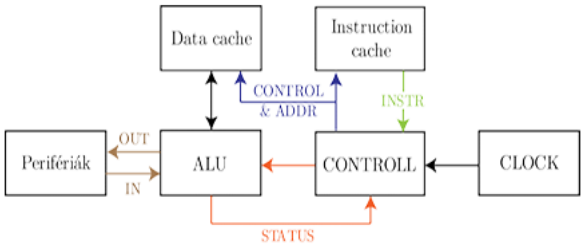
12.2. HARVARD ARCHITEKTÚRA

ritka + ábra

- **Utasítás és adat külön úton** mozog → párhuzamos elérés
- **Külön adatutak,** növekszik a teljesítmény

Szuperskalár CPU-kban:

- **L1 cache: Harvard:** utasítás + adat külön tárolódik *(lásd cache témakör)*
- **L2, L3 cache: Neumann:** utasítás + adat közösen tárolódnak
  - külön Instruction Cache
  - külön Data Cache
- → **Módosított Harvard architektúra**
  - program adatként is kezelhető
  - Mai processzorok: Harvard + Neumann elvek együtt



12.2.1. Vezérlési vázlat – működési lényege

Vezérlőegység (CONTROLL): lehívja az utasítást az Instruction cache-ből (INSTR adatút)
→ ez alapján jelet küld a **Data Cache**-nek, hogy az **ALU**-ba milyen cím kerüljön.
→ közben **Instruction Cache** felé is küld jelet: következő utasítás + aktuális adat egyszerre érkezik meg

ALU: **IN** és **OUT** adatutakon kommunikálhat a perifériákkal
→ A **vezérlőegység irányítja az ALU**-t, **STATUS adatúton** visszacsatolást biztosít felé

Minden művelet **órajelre szinkronizált**

Előnyei: - képes párhuzamosan adatot és utasítást olvasni vagy írni cache nélkül
- Az adat és utasítás tárolók különálló címtartománnyal rendelkeznek

Neumann architektúra: Harvarddal szemben a Neumann esetén egy óraciklusban vagy utasítást olvas be, vagy olvassa / írja a memóriát. (csak egy busz van)

12.3. Első generációs (keskeny) szuperskalárok

ritka

Jellemzők:

- **Kibocsátási ráta**: 2-3 kibocsátott utasítás/ciklus (RISC: 3, CISC: 2)
- **Közvetlen (nem pufferelt) kibocsátás**: CPU a dekódolt utasítást közvetlenül küldi a végrehajtó egységhez
- **Statikus elágazásbecslés**: Fetch alrendszer végzett, két féle képpen valósult meg
  Azért kell becsülni, mert az elágazás eldöntéséhez szükséges eredmény még nincs meg. Amíg a program még azt számolja, addig tudna haladni tovább a többi utasítással, ha tudná mit kéne csinálnia
  - **Fix elágazás**: elágazásnál mindig ugrik
  - A programkód bizonyos tulajdonságai alapján hozott létre egy statikus elágazásbecslést
- **Két szintű gyorsítótár**:
  - L1 cache-en **külön** van az adat és az utasítás: **Harvard** *processzor lapkán*
  - L2 cache-en és az operatív táron **közös** az adat és az utasítás tárolása: **Neumann** *különálló lapkán*

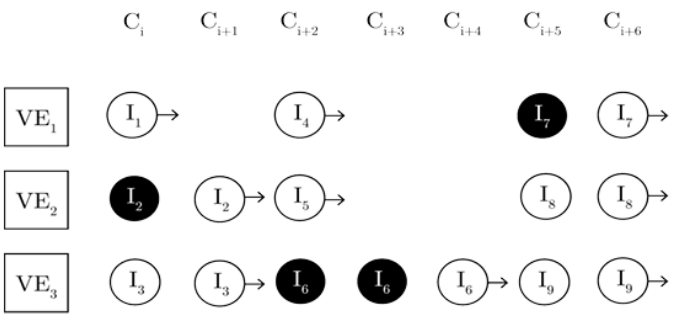
12.3.1. Közvetlen nem pufferelt kibocsátás:

külön nem nagyon, max I. gen szuperskalárral

Alapfogalom: Utasításablak

- Átmeneti puffer a dekódolt utasítások számára
- **Feladata**: Itt történik a dekódolás és a függőségvizsgálat
- **Szabály**: Csak olyan utasítás bocsátható ki a végrehajtó egységbe (VE), amelynek nincs függősége *addig marad itt amíg a függőség meg nem szűnik*

Sorrendi kibocsátás: Az utasítások csak az eredeti sorrendjükben hagyhatják el az ablakot
Egyszerre történő feltöltés: Csak akkor jönnek új utasítások, ha az előző adat (ablak) teljesen kiürült



Legnagyobb hátránya, hogy **egyetlen függő utasítás megállítja a teljes sort**, még akkor is, ha a többi utasítás már készen állna a végrehajtásra. Ez a végrehajtó egységek (VE) kihasználatlanságához vezet.

Fekete: függő utasítás

ritka

12.3.2. Szűk keresztmetszetek (Bottleneck)

- **Kibocsátás:** A közvetlen kibocsátás miatt a gyakorlati teljesítmény **(~1 utasítás/ciklus)** messze elmaradt az elméleti lehetőségektől (CISC:2, RISC: 3 utasítás)
- **Memória**: a memória lassúságot csökkentették **cache bevezetésével**
- **Elágazásfeldolgozás**: csökkentése **statikus elágazásbecsléssel**
- **Blokkoló adatfüggőségek (A fő hiba)**: Ez volt a legnagyobb probléma: az adatfüggőségek (RAW) és az ál-adatfüggőségek is teljesen **megállították (blokkolták)** a végrehajtást, mert a rendszer még nem tudta ezeket kezelni.

12.4. MÁSODIK GENERÁCIÓS SZUPERSKALÁROK

Cél: kibocsátási szűk keresztmetszet megszűntetése → **2. gen. szuperskalár CPU**

12.4.1. Második generációs szuperskalárok újításai

gyakori, 1-1 mondat mindegyikhez

Fő újítások:

- **Dinamikus utasítás ütemezés**
- **Regiszter átnevezés**
- **Dinamikus elágazásbecslés**
- Kifinomult és kibővített gyorsítótár alrendszer
- **Sorrenden kívüli kiküldés (OoO)**
- **RISC mag** (x86 architektúra)
- **(Reorder Buffer (ROB))** - *előadás nem említi felsorolásban, de fontos*

II. gen újításai

12.4.2. Dinamikus utasítás ütemezés

Lényege: A CPU nem ragad le egy hibás vagy várakozó utasításnál, hanem „átugorja” azt, és halad tovább a többivel. Két fázisra bontható: Kibocsátás és Kiküldés.

- **Kibocsátás**: A dekódolóból a várakoztató állomásba (pufferbe) még **sorrendben** érkeznek az utasítások.
- **Kiküldés**: A várakoztató állomásból a végrehajtó egységek (VE) felé már **sorrenden kívül** mennek az utasítások, amint felszabadulnak a szükséges adatok (Stréber modell)

Eredmény: Megszűnteti a kibocsátási szűk keresztmetszetet és növeli az **átbocsátóképességét**.

12.4.3. Regiszter átnevezés

A dinamikus utasítás ütemezés növeli az átbocsátóképességet, de a függő utasításokat nem.

Cél: Az ál-adatfüggőségek (WAR, WAW) felszámolása.

- A CPU minden eredeti regiszterhez egy külön **átnevezési** (piszkozat) regisztert rendel
- Az allokáció az utasításhoz kötődik, nem a fix architektúrális regiszterhez
  A regiszter neve csak egy hivatkozás, nem a fix helye. Pl két utasítás ugyan arra a helyre írna, így kapnak egy saját külön helyet és futhatnak párhuzamosan.

Extra: Támogatja a **dinamikus elágazásbecslést**: ha a CPU rossz ágon indult el, az eredmények csak a piszkozati regiszterben vannak (ami nem a végleges helye, így felülírható).

12.4.4. Dinamikus elágazásbecslés

Lényeg: Az elágazások kimenetét (ugrott vagy sem) történet bitekben tárolja a CPU, és ebből jósol a következő alkalomra. Történet bitek lehetnek: 1, 2 vagy 3 bitesek

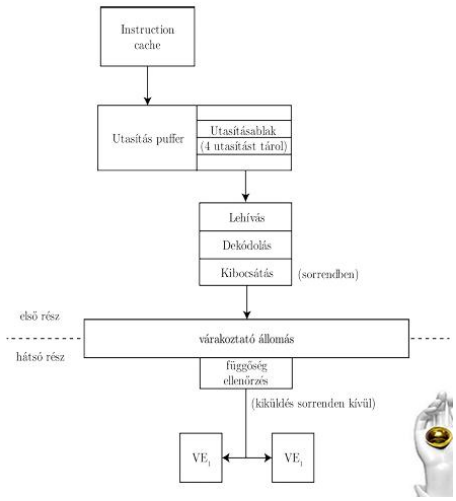
Becslés típusai:

- **1 bites**: Csak az utolsó elágazás alapján dönt. Ha akkor helyesen ugrott akkor most is.
- **2-bites**: Négy fokozatot használ a biztosabb döntéshez
  - **11 – Határozott elágazás**: (Kezdeti állapot) biztosan ugrást jósol
  - **10 – Gyenge elágazás**: Még mindig ugrást jósol, de már bizonytalanabb
  - **01 – Gyenge soros folytatás**: Inkább azt jósolja, hogy nem fog ugrani
  - **00 – Határozott soros folytatás**: Biztos benne, hogy nem ugrik

Mivel legtöbbször elágazásnál ugrás történik ezért 11 a kezdeti állapot. Ha a jóslat hibás akkor szintet vált pl 11 → 10. Csak harmadik alkalom után vált át soros (01) soros folytatásra.

12.4.5. Sorrenden kívüli kiküldés

- Kibocsátáskor már nincs függőségvizsgálat
  Az utasítások kibocsátáskor a **várakoztató állomásba** kerülnek.
  (lásd. Dinamikus utasítás ütemezés).
- Ez a puffer választja el a folyamat **első részét** (lehívás/dekódolás) a **hátsó részétől** (tényleges végrehajtás).
- Minden utasítás rendelkezik egy **állapotbittel**
- A vezérlőegység az **állapotbit segítségével** eldönti melyik utasítás függő és melyik független és amelyiket tudja, azt tovább küldi a vezérlőegységeknek **sorrendtől függetlenül** (OoO).



12.4.6. Operanduskezelés és a „Tölcsér-elv”

nem volt még

1. **Kibocsátáshoz kötött (Régebbi)**: Ha az adat még nincs kész, az utasításba az átnevezési regiszter azonosítója és egy állapotbit kerül. A CPU csak az állapotbitet figyeli: ha 0, az utasítás a várakoztató állomásban marad; ha 1, az adat betöltődik, mehet a végrehajtás
2. **Kiküldéshez kötött (Modernebb)**: A kibocsátáskor minden állapotbitet 0-ra állítanak, és csak közvetlenül kiküldés előtt ellenőrzik a függőségeket és töltik be az adatokat. Ez egyszerűbbé és gyorsabbá teszi a processzort.

Tölcsér elv: A processzor „hátsó része” szélesebb mint az eleje.
- **Kibocsátási ráta**: 3-4 utasítás/ciklus
- **Kiküldési ráta**: 5-8 utasítás/ciklus

nem volt még

12.4.7. Végrehajtás CISC architektúra esetén (RISC mag)

A modern processzorok ötvözik a két világ előnyeit (CISC, RISC), mivel a változó hosszúságú (1-17 byte) CISC utasítások futószalagos kezelése nem hatékony.

Kezelés: A dekódolás fázisában a bonyolult CISC utasításokat egységes, **128 bites RISC-szerű utasításokká alakítja.**

Hatékonyság: Átlagosan 1 CISC utasítás → 1,2 – 1,5 RISC utasítás lesz. Így kívülről látható 3 utasítás/ciklus sebesség a belső RISC magban már 4 utasítás/ciklust jelent.

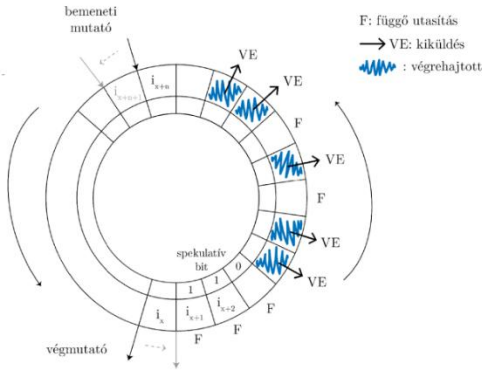
Az átalakítás a **dekódolás** fázisban, a RISC-CISC **visszaalakítás** pedig a visszaírás során történt.

12.4.8. A Reorder Buffer (ROB) működése

kevésbé ritka + ábra

A körpuffer felépítése:

- **Kör alakú puffer**: A ROB egy folytonos körpufferként működik, ahol az utasítások a bemeneti mutatónál lépnek be és a végmutatónál távoznak (íródnak ki).
- **Párhuzamos kiküldés**: A két mutató közötti területen a várakozó utasítások körül bármelyik független utasítás kiküldhető (VE) végrehajtásra, függetlenül az eredeti sorrendtől



A ROB az átnevezett regiszterekhez tartozó állapotot és eredményeket tartja nyilván.

Spekulatív végrehajtás és biztonság

- **Spekulatív bit**: minden utasításhoz tartozik egy bit: ha értéke 1, az utasítás feltétele még nem került kiértékelésre (bizonytalan)
- Amíg a spekulatív bit 1, az eredmény nem írható ki a végleges memóriába, így biztosítva a szekvenciális konzisztenciát és az architektúrális regiszterek hibás értékkel történő felülírását.

Feltétel kiértékelése után (elágazás becslés):

- **Helyes irány** esetén a **spekulatív bit értékét 0-ra** állítja és az eredmény kiírható
- **Hibás becslés** esetén a ROB-ból **törlésre kerül az utasítás**, az **átnevezési regiszterek felszabadulnak** és a helyes irányba történő **utasítás lehívásra** kerül

Kiírási szabályok: *akár külön kérdés is lehet*

- Csak akkor írható ki, ha **minden korábbi** utasítás kiírása már megtörtént.
- Spekulatív állapotban lévő utasítás nem írható ki!
- **CISC-rekonverzió** utasítás több belső RISC műveletre bomlik, a kiírás csak akkor történik meg, ha az összes hozzátartozó RISC egység elkészült. Itt történik meg a rekonverzió is!

CISC-rekonverzió: RISC utasítások vissza alakítása CISC utasítássá.

Összefoglalva:

- Erőforrás függőség → Több végrehajtó egység alkalmazása
- Ál-adatfüggőségek → Regiszter átnevezés teljes mértékben megoldja
- Vezérlés függőség → Súlyossága gyengült a spekulatív elágazáskezeléssel
- Valós adatfüggőség → Továbbra is lassították a végrehajtást!

Valós adatfüggőség részben kezelve a sorrenden kívüli kiküldéssel, várakoztató állomással és spekulatív elágazáskezeléssel, de még mindig lassítja a végrehajtást!

12.5. HARMADIK GENERÁCIÓS SZUPERSKALÁROK

A második generáció végére az utasításszintű párhuzamosság (ILP) elérte a fizikai korlátait, így a fejlesztés az **adatszintű párhuzamosság** és a multimédiás gyorsítás felé fordult.

Egyesek szerint nem nevezhető harmadik generációnak, hanem csak a másodiknak a kibővítése multimédiás/3D képességekkel.

12.5.1. Utasításon belüli párhuzamosság típusai

gyakori

- **Duál műveleti utasítások**: Egy utasítás két műveletet végez (pl. *multiply-add*: y = ax + b). Főleg numerikus számításoknál hasznos.
- **SIMD utasítások**: Egyetlen utasítás több, egymástól független operanduson hajtja végre ugyanazt a műveletet.
- **VLIW architektúrák**: *A fordítóprogram csomagol több független műveletet egyetlen hosszú utasításszóba.*

12.5.2. SIMD architektúra jellemzői

kevésbé ritka

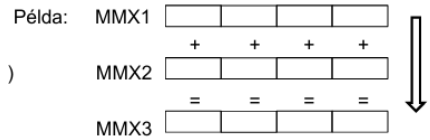
- **Logikai architektúra kiterjesztése**: Új multimédiás utasítás készletet alkottak
- **Memóriaigény**: Egy utasítás akár 8 operandust is kérhet, ami drasztikusan növelte a sávszélesség-igényt.
- **Megoldás**: Az L2 cache beépült a processzorba (korábban Pentium II-nél külön lapkán)
- **Buszrendszer:** Megjelent az **AGP**, majd a **PCI Express** a grafikai adatok gyorsabb mozgatásához.
- Elsőként **MMX (Multimedia Extension) fixpontos multimédiás utasítások**, később megjelent a **SSE (Streaming SIMD Extension) ami már lebegőpontos műveleteket** is támogatta
- **Alkalmazás**: Általános programoknál nem hozott javulást, de a multimédiás alkalmazásoknál mint: videó, kép és 3D játékok (független adatok tömeges feldolgozása) esetén óriási gyorsulást jelentett.

12.5.3. Logikai architektúra kibővítése

kérheti SIMD-vel

A processzor belső szerkezetét át kellett alakítani az új adatok fogadására:

- **Pakolt adattípusok**: 64 bites egységekbe több kisebb adatot „csomagolnak”
  - Pakolt byte: (8 bit hossz x 8 db = 64 bit)
  - Pakolt félszó: (16 bit hossz x 4 db = 64 bit)
  - Pakolt szó: (32 bit hossz x 2 db = 64 bit)



  - SSE pakolt típus – egyszeres pontosság (32 bit x 4 db = 128 bit)
  - SSE pakolt típus – kétszeres pontosság (64 bit x 2 db = 128 bit)

  Megfelelnek az IEEE 754-es szabványnak ami lebegőpontos számokra vonatkozik

- **Bit-blokk átvitel**: Az ablakokat/képeket egységes blokként kezeli a rendszer, így nem kell elemenként mozgatni a hatalmas adatmennyiséget.

Példa a sávszélesség-igényre: *Egy Full HD (1920x1080) kép 24 bites színmélységgel kb. 6 MB. 30 FPS sebességhez 180 MB/sec feldolgozási sebesség kell, amihez elengedhetetlen a blokkos kezelés és a tömörítés.*

kérheti SIMD-vel

12.5.4. Fizikai architektúra kibővítése

- SSE (Lebegőpontos): **8 darab új, 128 bites regiszterek** kerültek a CPU-ba
- **Operációs rendszer támogatása**: Az új regiszterek és a megváltozott megszakításrendszer miatt szükség volt az OS együttműködésére (elsőként Windows 98)

12.5.5. Grafikai képfeldolgozás: Vektoros vs 3D

fontos, de nem tudom, hogy kérte már

Vektoros képfeldolgozás:

- Az objektumokat **geometriai jellemzőkkel** írják le
  (pl egy egyenest két végpontja, kört a középpont és sugár tárolásával)
- **Előnye**: Sokkal kevesebb adatot kell tárolni (Full HD vonal esetén 1920 pixel helyett 2), illetve könnyen nagyítható és mozgatható
- **Hátránya**: Nagy számítási kapacitást igényel (lebegőpontos számítások), és textúrázásra/árnyékolásra van szükség az élethűséghez

3D ábrázolás követelményei:

- **Perspektíva**: A párhuzamosok a végtelenben összefutnak, a távoli tárgyak kisebbek
- **Atmoszférikus hatás**: A távoli tárgyak halványabbak és kékesebbek
- **Teljesítmény**: Legalább 15-30 FPS szükséges 3D filmekhez, ami kb 500.000 objektum/sec
- **Kapacitás**: A Pentium III (SSE) teljesítménye kb. **2 GFLOPS** (2 milliárd FP művelet/sec) volt, a mai CPU-k már a **TeraFLOPS** tartományban mozognak.

13. NETBURST ÉS SZÁLSZINTŰ PÁRHUZAMOSÍTÁS

Az 1990-es évek végére a hagyományos módszerekkel már nem lehetett növelni az utasításszintű párhuzamosságot (ILP). Az Intel válasza erre a frekvencia drasztikus növelése és egy teljesen új architektúra, a **Netburst** volt (Pentium 4).

13.1. FREKVENCIA NÖVELÉSÉNEK FORRÁSAI

gyakori + következmények

A cél a 10 GHz elérése volt, amit két módon próbáltak támogatni:

1. **Gyártási csíkszélesség csökkentése**: (180 nm → 65 nm) Kisebb tranzisztorok = gyorsabb elektronáramlás = magasabb órajel
2. **Futószalag-fokozatok rövidítése**: A logikai kapuk számának csökkentése egy fokozaton belül. Ez kényszerűen növelte a fokozatok számát (hosszabb futószalag)

Több fokozat → párhuzamosan végrehajtott utasítások száma nőtt → függőségek száma is nő → csökkenhet a hatékonyság és egy érték fölött a teljesítmény is

nem hiszem, hogy kéri

13.2. PENTIUM 4 LEGFONTOSABB ÚJÍTÁSAI

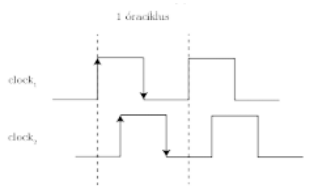
- CISC kívül (komplex utasítások), de belül egy gyors RISC mag dolgozik
- **Hosszú futószalagok**: több függőség, de magasabb frekvencia
- **Bár hatékonyságban alul maradt az AMD**-vel szemben (kevesebb végrehajtott utasítások órajelenként – függőségek miatt), de a magasabb órajel miatt magasabb teljesítmény
- **Védelem**: Bevezették a **Thermal Monitor**-t, ami túlmelegedéskor visszavette az órajelet, így a CPU nem sült ki (ellentétben a korabeli AMD-kkel)

13.3. LEGFONTOSABB ÚJÍTÁSOK

kell mindegyik, de legfontosabb: execution trace cache, rapid execution engine, replay system

- **Execution Trace Cache**: Az L1 cache már dekódolt RISC utasításokat tárol a végrehajtás sorrendjében, így spórol a dekódolási időn.
- **Hyper futószalag**: A dekódolást kihelyezték a futószalagon kívülre, hogy ne akadályozza a magas frekvenciát.
  A CISC utasítások dekódolása és átalakítása lassú (pl változó hossz), ez akadályozza a nagy frekvenciás végrehajtást. A frekvencia növelés okán 10-15 fokozatról 20-31 fokozatos futószalagot használtak, hátránya a függőségek miatt kevesebb utasítás hajtódik végre óraciklusonként illetve hibás becslés esetén nagyobb büntetés → több fokozatot kell törölni.

- **Enhanced Branch Prediction**: Új elágazásbecslő logika, 94-97%-os hatékonyságú, hogy a futószalagot ne kelljen gyakran üríteni hiba miatt
- **Quad Data Rate Bus**: Órajelenként 4x-es adattovábbítás (felfutó/lefutó él + fáziseltolás) L1 és L2 gyorsítótárak felé. Szükséges 2 órajel generátor.



- **Rapid Execution Engine**: Az egyszerű FX műveletek gyors végrehajtására. Az órajel felfutó és lefutó élére is képes műveletvégzésre → végrehajtási idő akár fél óraciklus.
  Kevesebb kapu és csak alap műveletek elvégzésére. Később 2 is kerül a CPU-ba.
- **Replay System**: RISC szerű utasítások ismételt végrehajtása. Ha egy utasítás adata még nincs kész, nem állítja meg a futószalagot, hanem beleteszi egy „várólistába" (Replay Queue) és később újra próbálja.
  Ütemező megbecsüli az utasítások végrehajtási idejét, ez alapján ennyi idővel hamarabb kiküldi a függő utasítást a végrehajtó egység felé.
  Ha rossz volt a becslés → Replay Queue

13.4. FEJLŐDÉSI KORLÁTOK

kevésbé ritka

- Hatékonysági korlát (ILP kimerülés, memória sebesség olló)
- **Disszipációs korlát**
- Párhuzamos buszok frekvencia korlátja

3,8-4 GHz környékén bekövetkezett a **hőkatasztrófa**

Disszipáció: az elektromos energia hővé alakulása a processzorban.

13.4.1. Statikus Disszipáció

kevésbé ritka
+ képlet

A szivárgási áram hőveszteséget okoz.
Szivárgási áram: a tranzisztor kikapcsolt állapotában is folyik minimális áram (Gate nem tökéletes szigetelő)

Képlete:

$$D_s = V * I_{leak}$$

Magyarázat: Ahogy csökkentették a csíkszélességet és **növelték a frekvenciát** (V), a **szivárgási áram** (I_{leak}) **exponenciálisan** növekedni kezdett.

13.4.2. Dinamikus Disszipáció

kevésbé ritka
+ képlet

A dinamikus disszipáció az aktív tranzisztorokon átfolyó áram hőtermelése.

Képlete:

$$D_d = A * C * V^2 * f_c$$

A: Az aktív kapuk (tranzisztorok) részaránya
C: A kapuk összesített elosztott kapacitása
V: Tápfeszültség
f_c: Magfrekvencia

Magyarázat: A frekvencia (f_c) növelése **lineárisan**, míg a feszültség (V) növelése **négyzetesen** növeli a hőt.
→ frekvencia emelését a feszültség csökkentésével próbálják ellensúlyozni.

13.4.3. Megoldás a disszipációra: DVFS

volt már

Mivel az órajel további növelése kezelhetetlen hőtermeléshez vezetett, a fejlesztési irány az intelligens energiagazdálkodás felé fordult. Ennek lefontosabb eszköze a **DVFS (Dynamic Voltage and Frequency Scaling)**.

Működése:

- Meghatározza a szükséges teljesítményt
- Frekvencia hozzáillesztése a teljesítményhez
- Az adott órafrekvencia fenntartásához szükséges **minimális feszültség** beállítása

Következmény: Tisztán órajel alapú fejlesztések helyett több magos architektúrák.

kevésbé ritka, tulajdonságai
+ miért rosszabb mint az ILP

13.5. Szál szinten párhuzamos architektúrák (TLP)

Mivel az egymagos processzorok komplexitásának növelése már nem hozott arányos teljesítményjavulást, a fejlesztés az **utasításszintű párhuzamosságról (ILP**) a **szálszintű párhuzamosság (TLP)** felé fordult.

szál definíció

Szál (Thread): A program legkisebb önállóan végrehajtható része. → párhuzamosan futtatható

Cél: A processzormagban keletkező üresjáratok (függőségek miatti várakozások) kitöltése más szálakból származó feladatokkal.

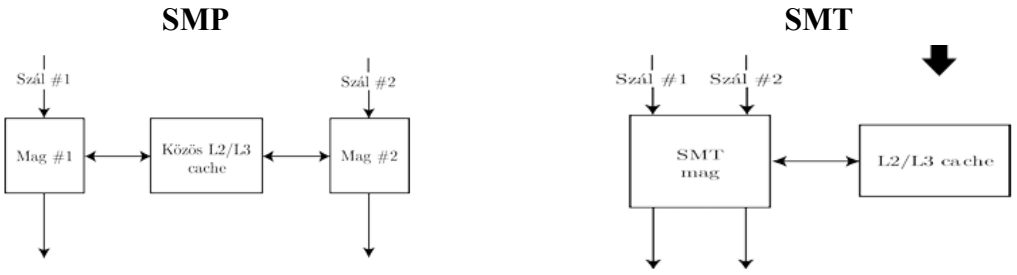
A párhuzamosság formái

- **Implicit párhuzamosság**: A programozó hagyományos, szekvenciális (soros) kódot ír. A párhuzamosítást a hardver (CPU) és a szoftver (fordítóprogram) automatikusan végzi el.
- **Explicit párhuzamosság**: A programozó tudatosan párhuzamos szerkezetű kódot ír, külön meghatározva a szálakat és azok együttműködését.

volt már elvileg

13.5.1. Többszálúság csoportosítása

- **Szoftveres**: Többszálú alkalmazások, vagy OS futtatása egyszálú CPU-n (időosztás)
- **Hardveres**: A CPU maga támogatja több szál kezelését.
    - **SMP – Symmetric multiprocessing**: Több külön mag egy lapkán
    - **SMT – Simultaneous multithreading**: Egy mag futtat egyszerre több szálat (pl. Hyper Threading)



13.5.2. Szál szinten párhuzamos architektúrák osztályozása

gyakori

Több szál futtatása P4 CPU-n (Hyper Threading nélkül): Ha nincs hardveres támogatás akkor az új szál futtatása előtt az aktív szál kontextusát el kell menteni a memóriába, ezután a másik szál kontextusát be kell tölteni. A kontextus váltás akár 2-3 ezer óraciklus → LASSÚ

Típus	Működési elv	Előny	Hátrány
Finoman szemcsézett	Órajelenként (ciklusonként) vált a szálak között	Nincs váltási késleltetés; az egyik szál elakadása nem állítja meg a CPU-t	*Az egyedi szálak végrehajtási ideje megnő, mivel ritkábban kapnak vezérlést*
Durván szemcsézett (SoEMT)	Akkor vált szálat, ha az aktuális elakad (pl függőség miatt)	*Egyetlen szál képes a teljes CPU erőforrást maximális sebességgel kihasználni*	A váltás felismerése és a kontextuscsere 1-2 óraciklus veszteséggel jár
SMT	Egyetlen órajelen belül több különböző szál utasításait futtatja egyszerre	Kitölti a függőségek miatti üresjáratokat a végrehajtó egységekben	

13.5.3. Párhuzamosság megvalósításához szükséges előkövetelmények

- **Erőforrások többszörözése**: Például szálankénti program counter (PC) és regiszter tárolókészlet. Így a szálak közötti váltás nagyon gyors
- **Erőforrások megosztása**: Erőforrásokat meg kell tudni osztani a szálak között. Szükség esetén egyesíteni is, ha egy szál megakadása esetén az aktív szál a teljes erőforrást tudja használni.

kevésbé ritka

13.5.4. Intel Hyper Threading

A **Hyper-Threading** egy **két szálas SMT (Simultaneous Multithreading)** megoldás, amely először a Pentium 4 (Northwood) processzorokban jelent meg.

- **Lényege**:
  Egyetlen fizikai mag az operációs rendszer felé **két logikai CPU-ként jelenik meg**. A processzor **két különböző szál utasításait** kezeli és küldi végrehajtásra (sorrenden kívül).
- A processzorban ekkor egyszerre **két architektúrális állapot van jelen**: (Két külön regiszterkészlet és programszámláló, de a végrehajtó egységeken osztoznak.)
- A megoldás kb. **~5% terület- és fogyasztásnövekedést** igényel, optimális esetben akár **~30% teljesítménynövekedést** eredményezhet.

Üzemmódok

ST (single task)
- **Egy szál** végrehajtása történik,
- A szál **az összes végrehajtó erőforrást megkapja (egyesítés)**
- Két lehetséges állapot:
  - **ST0** vagy **ST1**, attól függően, melyik logikai CPU aktív

MT (multi task)
- **Több szál** végrehajtása történik párhuzamosan.
- A szálak **osztoznak** a végrehajtó egységeken.

Működés:

- A CPU **alapértelmezetten MT módban** indul.
- Ha az egyik szál megakad (pl. függőség miatt), a CPU **ST módba vált**
- A függőség megszűnésekor a CPU **visszatér MT üzemmódba**
- Az üzemmód váltás **HALT utasítással** történik
  - megszakítja a CPU futást
  - energiatakarékos állapotba helyezi
- HALT után a CPU **ST0 vagy ST1** állapotba kerül attól függően, melyik logikai CPU adta ki
- A HALT utasítást kizárólag az OS vagy más alacsony szintű szoftver adhatja ki.

13.5.5. SMT tervezési célok (Összegzés):

- Kis magméret növekedés
- A szálak között váltás ne vegyen igénybe extra óraciklusokat (tárolók a magban)
- Egy szál futása esetén ne legyen lassabb mintha nem lenne megosztva a mag!