Archi alapok

/ 1. Modul : Architektúrák /

+ Mi az a számítógépes architektúra (mi az archi fogalma)?

  "A számítógép struktúra, amit a gépi kódú programozónak értenie kell annak érdekében, hogy helyes programot tudjon írni az adott gépre" – Amdahl

  - A struktúra, amit ismerni kell az a…
    × Regiszterek
    × Memória
    × Utasítás-készlet
    × Címzési módok
    × Utasításkódok
  - Tudni kell hogy az adott rendszerben ezek mik és hogyan működnek

+ A számítási modell: A számításra vonatkozó alapelvek egy absztrakciója, egy rövid, zanzásított leírása a modellnek
  - **Egy számítási modell tulajdonságai:**
    × Milyen módon hajtjuk végre a számítást
      ~ Szekvenciálisan (egymás után, egyszerre egy utasítás)
      ~ Párhuzamosan (egyszerre több utasítás)
    × Mi befolyásolja, irányítja a lefutást
      ~ Vezérlés meghajtott (egy fix utasítás sorozat alapján fut le a számítás)
      ~ Adat meghajtott (minden az adatok elérhetőségétől függ)
      ~ Igény meghajtott (mindent csak akkor számolunk ki, amikor szükség van rá)
    × Milyen módon írjuk le a problémát
      ~ Procedurális (lépésről-lépésre leírt utasítások)
      ~ Deklaratív (a végeredmény van megadva, nem az ahhoz szükséges lépések, *az SQL is ilyen*)
    × Miken hajtjuk végre a számításokat

⬇

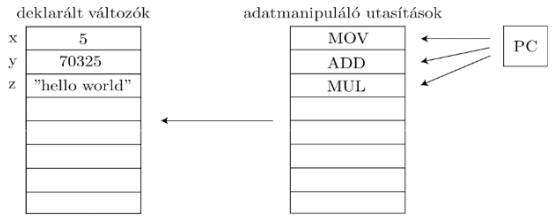
+ A legfőbb tulajdonság, ami alapján csoportosítani lehet az, hogy **min hajtjuk végre a számítást, mi a számítások alapja?**
  - **Adat alapú modellek**
    × **Neumann modell**
    × **Adatfolyam modell**
    × Applikatív modell
  - Objektum alapú modellek
  - Predikátum logika alapú modellek
  - Tudás alapú modellek (AI, előre- vagy visszafelé következtetés)
  - Hibrid modellek
+ Adat alapú modellek
  - Az adatok típussal rendelkeznek (elemi vagy összetett)

- Ez határoz meg mindent az adattal kapcsolatban (felvehető értékek, végrehajtható műveletek stb)

- Neumann modell
  - Változókat hozunk létre (más szóval: deklarálunk)
  - Adatokat manipuláló utasításokat deklarálunk
  - Vezérlést átadó utasításokat tudunk deklarálni

  - A számítást adatokon hajtjuk végre
  - Az adatokat változókban tároljuk amik értéke (korlátozott alkalommal) változtatható
  - Az adatok és az utasítások is a memóriában helyezkednek el
  - Egy számítási feladat igazából csak egyszerű, elemi műveletek sorban végrehajtva

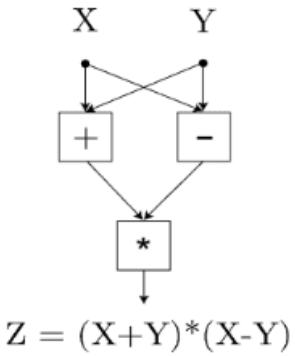
  - Az utasítások, fix sorrendben hajtódnak végre (más szóval: statikusak) és a változók értékeit módosítják
  - Az utasításokon egy Program Counter (PC) nevű regiszter halad végig, amit különböző parancsokkal lehet irányítani is (pl. JUMP: visszaugrás egy adott sorszámú parancsra)



  - Az előbb említett tulajdonságok alapján: szekvenciális, vezérlés meghajtott, adat alapú, procedurális

  - Előnye: Könnyű implementálni
  - Hátránya: Mivel a változók értéke változtatható és tudunk lépkedni az utasítások között (tehát egy változót többször is “elővehetünk”) ezért előzményérzékeny

- Adatfolyam modell
  - × Az adatok bemeneti adathalmazokban vannak
    - ~ Csak egyszer lehet nekik értéket adni
  - × A végrehajtás egy gráfként képzelhető el (lásd: kép)
  - × Kis, egy-egy feladatot végrehajtó egységek vannak
  - × Lehetséges a párhuzamos a végrehajtás
  - × Nincs PC, hanem egy művelet azonnal végrehajtódik, amint elérhetőek a szükséges adatok (ezt nevezzük **stréber modellnek**) => ezt viszont valahogyan érzékelni kell
  - × Az adatok az utasításokon belül van tárolva
  - × Előnye: Nem előzményérzékeny és párhuzamosan több feladatot tud végrehajtani
  - × Hátránya: Nehezebb implementálni



	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)

+ Architektúra fogalma := {M, S, I}
+ L: Adott absztrakciós szinten
+ M: Számítási modell
+ S: Specifikáció
+ I: Implementáció

+ Egy architektúrának kettő nagy része van
  - Fizikai architektúra := {M, I}
    - × A modell és az implementáció együttes leírása egy adott absztrakciós szinten
    - × Két szinten lehet vizsgálni a fizikai architektúra részeit
      - ~ Számítógép szinten nézve az alkatrészek:
        - Δ Processzor
        - Δ Memória
        - Δ Buszrendszer
      - ~ Processzor szinten nézve az alkatrészek (procin belüli alkotóelemek)
        - Δ Műveletvégző egység (ALU)
        - Δ Vezérlő
        - Δ I/O rendszer (Input/Output)
        - Δ Megszakítási rendszer

- Logikai architektúra := {M, S}
  - × Egy funkcionális leírás, amiben az adott modell és a programozó által látott specifikáció van. Az architektúra itt egy fekete doboz igazából (nem lényeg hogy működik pontosan, csak van)
  - × Ez is két szinten vizsgálható
    - ~ Számítógép szinten azt vizsgáljuk, hogy adott bemenetre az egész rendszer hogyan reagál, mit produkál. Ennek vizsgálására használjuk az operációs rendszert
    - ~ Processzori szinten magára a processzorra tekintünk úgy mint egy fekete doboz (mindegy hogyan működik pontosan). A programozó feladata olyan bemenetet megadni a processzor által érthető utasításkészlet alapján, ami a kívánt kimenetet adja vissza. Erre a feladatra valamilyen programnyelvet használ a programozó, ami le lesz fordítva a processzor által használt utasításokra

⬇

+ Logikai architektúra komponensei processzori szinten (erről fogunk ebben az anyagrészben innentől beszélni)
  - Adattér
  - Adatmanipulációs fa
  - Állapottér
  - Állapotműveletek

⬇

+ Logikai architektúra > Processzori szint > Adattér
  - Egy tér, ami olyan módon tárolja az adatokat, hogy a CPU tudja manipulálni **közvetlenül**
  - Kettő részre bontható:
    - × **Memóriatér**: Nagy, olcsóbb, viszont külső lapkán van és lassabb
      - ~ Létezik egy olyan transzparens folyamat ami a program futása során a valós memóriából a nem használt adatokat kiírja a virtuális memóriába
      - ~ Létezik egy olyan transzparens folyamat ami a program futása során a virtuális memóriából dinamikusan (futási időben) visszaírja a valós memóriába -> MMU (Memory Management Unit) -> AGU (Address Generation Unit)
      - ~ Kettő fajta memória létezik
        - Δ Virtuális memória: A programozó látja, ezzel dolgozik
        - Δ Fizikai memória: A CPU látja, használja
    - × **Regisztertér**: Kisebb, drágább, viszont a CPU lapkáján van és gyorsabb

	Virtuális memória	Fizika memória
Mérete	nagyobb	kisebb
Sebessége	lassabb	gyorsabb
Elhelyezkedés	háttértáron	lapkán
Láthatóság	Programozó látja	CPU látja
Mit tárol	Itt várnak az adatok	Itt fut a program

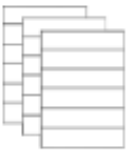
Virtuális memória Vs Fizikai memória

  - Regiszterek típusai
    - × Egyszerű regiszterkészlet
      - ~ egyetlen regiszter -> akkumulátor regiszter
      - ~ több dedikált adatregiszter
      - ~ univerzális regiszterkészlet
        - Δ gyakran használt változók folyamatosan a regiszterben maradhattak

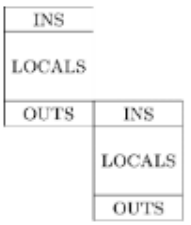
        - Δ Stack (verem) regiszterkészlet
            - Előny: nem kell címezni
            - Hátrány: Az adatok kiolvasása csak szekvenciálisan működik, lassú
    - × Adattípusonként különböző regiszterkészlet
        - ~ Lebegőpontos adattípus esetén
        - ~ Mantissza és Karakterisztika ez két részre bontva
        - ~ Párhuzamos működés
        - ~ SIMD adattípus -> Multimédiás adattípus
    - × **Többszörös regiszterkészlet** (egymásba ágyazott eljárások gyorsítására szolgál)
        - ~ A regisztertér állapota az állapottérrel együtt a **kontextus**
        - ~ Ha egymásba ágyazott eljárások esetén, meghívunk egy új eljárást (úgymond “egy szinttel mélyebbre megyünk”), akkor a “belső” eljárásnak kell egy tejes regiszterkészlet, amiben “dolgozhat”
            - Δ Ekkor el kéne a “külső” eljárás kontextusát menteni az viszont nagyon lassú
            - Δ A kontextusok közötti váltásra léteznek kontextuskapcsolók
            - Δ A kontextuskapcsolók nagyon gyorsan tudnak regiszterkészletek között váltani

            - Δ Az lenne a cél hogy tudjunk minden egyes kontextusnak saját regiszterkészletet tudjunk adni és azok között váltogatni *(+ kéne még egy extra regiszterkészlet ami a regiszterek közötti kommunikációt figyeli)*

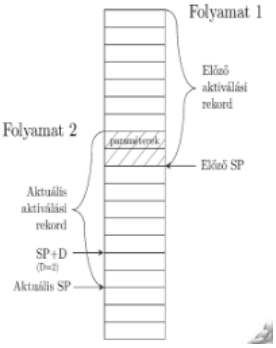
- Többszörös regiszterkészlet megoldások:
    - × Több egymástól független regiszterkészlet: Nincs átfedés a reg.készletek között, az operatív táron keresztül tudnak egymásnak paramétereket átadni



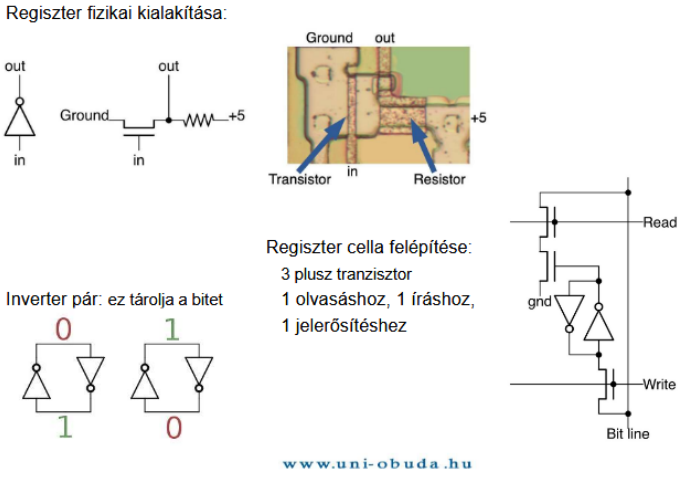
    - × Átfedő regiszterkészlet
        - ~ Egy regiszter lapot három részre osztunk
            - Δ Bemenő paraméterek (INS)
            - Δ Lokális paraméterek (LOCALS)
            - Δ Kimenő paraméterek (OUTS)
        - ~ A bemenő és kimenő paraméterek ugyanazon a címen vannak, így könnyen eléri az összes eljárás
        - ~ Hátránya hogy egy fixek a paraméterszámok, így ha több bemenő vagy kimenő paraméter van mint arra megfelelő regiszter (túlcsordulás történik), akkor azt nem tudjuk kezelni



    - × Stack cache
        - ~ Mikor egy regiszterkészletre van szükség akkor a compiler egy stack pointer-rel (SP) kijelöl egy regisztertartományt, amit az eljárás használhat
            - Δ Ez a regisztertartomány az aktiválási rekord és akkora lehet, amekkorát az eljárás igényel



Δ Az aktiválási tartományok fedik egymást (ezeken a közös helyeken vannak a közösen használt bemeneti/kimeneti paraméterek)

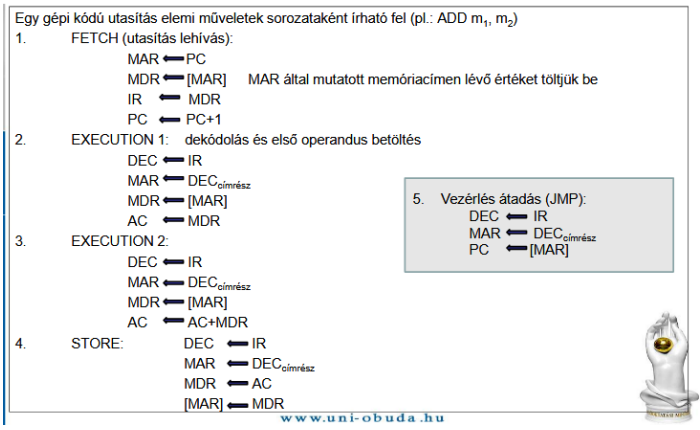


\+ Logikai architektúra > Processzori szint > Adatmanipulációs fa

- Megmutatja a potenciális adatmanipulációs lehetőségeket
- Bizonyos részfái megmutatják egy konkrét implementáció adatmanipulációs lehetőségeit
- Részei:
  - × Adat típusok
    - ~ Elemi
      - Δ Numerikus
        - Fix pontos
          - Kódolás szerint
            - Egyes komplemens
            - Kettes komplemens -> Előjeles 1 byte/2byte/4byte || Előjel nélküli
            - Többletes kódolás -> lebegőpontos Nem normalizált / **Normalizált**
        - Lebegőpontos
          - Hexára normalizált
          - Binárisra normalizált
            - VAX
            - IEEE (754)
              - egyszeres pontosságú
              - kétszeres pontosságú
              - kiterjesztett pontosságú
          - Nem normalizált
        - Binárisan Kódolt Decimális (BCD) => Pakolt, Pakolatlan
      - Δ Karakteres
        - EBDIC
        - ASCII (American Standard For International Interchange)
          - Szabványos -> 7bit
          - Kiterjesztett -> 8bit
        - UNICODE -> 2byte (16bit)
      - Δ Logikai (AND, OR)

        - 1 byte
        - 2 byte
        - 4 byte
        - Változó hossz
        - Általában 1 bit értékes és ez a legmagasabb helyiértéken van
    - Δ Pixel
  - ~ Összetett (Adatszerkezetek)
    - Δ Elemiből épülnek fel
    - Δ Különböző elemekből épülnek fel => rekord
    - Δ Azonos típusokból => Tömb, Szöveg, Verem, Sor, Lista, Fa, Halmaz
- × Műveletek
  - ~ Az adatmanipulációs fa minden művelet esetén megállapítja, hogy milyen utasítás típusok vannak megengedve és milyen operandus típus választható
  - ~ Utasításokból áll
  - ~ Műveletek szintje
    - Δ Milyen műveletek végezhetők az adott adattípusokkal
- × Operandusok típusai
  - ~ Két operandusos, Három operandusos
  - ~ Memória, regiszter, AC (akksi) típus
- × Címzési módok
  - ~ Memória / regiszter esetén
- × Gépi kód

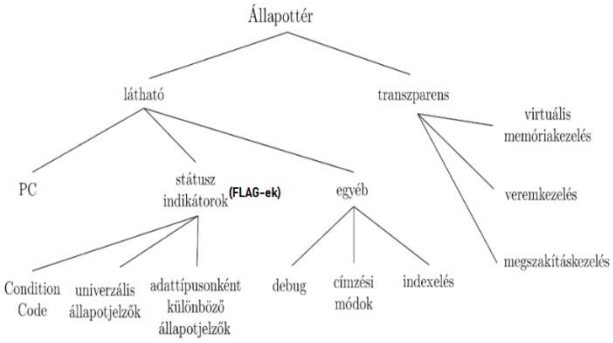
+ Mi az utasítás?
  - Számítógép által végrehajtható művelet leírása
  - Egy utasítás miből áll?
    - × Műveleti kód avagy utasítás mező (mit kell csinálni)
    - × Címrész avagy operandus mező (min kell csinálni)
+ Processzor regiszterei:
  - Memory Address Register (MAR)
  - Memory Data Register (MDR)
  - Program Counter (PC)
  - Instruction Register (IR)
  - Általános célú regiszterek



+ Utasítás típusok: Egy utasítási kódhoz több cím (operandus) tartozhat és ezek lehetnek forrásoperandusok vagy címoperandusok
  - Legfontosabb számú címes utasítások
    - × 3 címes (cél, operandus1, operandus2)
      - ~ A következő utasítás címét a PC tárolja (4 címes esetben ezt kézileg kell megadni)
      - ~ Előnye hogy lehet párhuzamosítani, viszont maga a kód túl hosszú és sok memóriát igényel az ilyen utasítás tárolása
    - × 2 címes (operandus1, operandus2)
      - ~ Az eredmény felülírja az operandus1 értékét
    - × 1 címes
      - ~ Ilyenkor egy parancs kell hogy az első operandust betegyük az AC (accumulator) regiszterbe, aki azt a művelet idejéig eltárolja (mondhatni “észben tartja”) majd egy másik paranccsal módosítani az AC-ben tárolt értéket
      - ~ pl. LOAD[A] utána pedig ADD[B] (eltároljuk A-t az AC-ben és a következő paranccsal hozzáadjuk az AC tartalmához a B-t, ezzel felülírva az A tartalmát)
      - ~ Így rövidebb és egyben gyorsabb utasításaink lesznek, viszont több is kell majd belőlük
    - × 0 címes
      - ~ POP
      - ~ PUSH
      - ~ CLEAR
+ Operandus típusok
  - AC akkumulátor (gyors, de csak egy van belőle)
  - Memória (nagy, de hosszú a címe és lassú)
  - Regiszter (gyors, de kevés van belőle)
  - Verem (gyors, de csak a tetejét látjuk)
  - Közvetlenül beírt érték (immediate) (gyors, mert nem kell hozzá regiszter, de csak programból változtatható)
+ Címzési módok = címszámítási algoritmus
  - 3 egymástól független elem kombinációja
    - × Címszámítás – jelzi, hogy abszolút (hosszú) vagy relatív (rövid) címzést használunk. Deklarálni kell egy bázis címet + címszámítási algoritmust
    - × Cím módosítás – Indexelés, auto inkrementálás, auto dekrementálás
    - × Deklarált (tényleges) cím meghatározása – A címet direkt vagy indirekt. Valós vagy virtuális címként fogjuk fel
    - × Az abszolút címzésnél a hátrány, hogy a CPU címtér nagyon nagy (4 – 64 TB), ezért Relatív címzést használunk -> bázis(S) eltolási cím (D).
    - × Bázis cím lehet pl: PC, Top of Stack => blokkos címzést használunk
  - Indexelés
    - × Adatblokkok R(egiszter) = S(cím). Y(effektív cím) = S => R(egiszter) = D <- eltolás ➔ Egy dimenziós eltolás
    - × Több dimenziós eltolásnál több index R(egiszter) kell
+ Logikai architektúra > Processzori szint > Állapottér

- Olyan tárolók, amelyek a programmal kapcsolatos állapotinfókat tartalmaznak

+ Logikai architektúra > Processzori szint > Állapotműveletek
  - × PC állapotműveletei
    - ~ Inkrementálás felfele
    - ~ Inkrementálás lefele
    - ~ Felülírás
  - × Flag-ek állapotműveletei
    - ~ Save
    - ~ Set
    - ~ Reset
    - ~ Load
    - ~ Clear



Állapottér részei

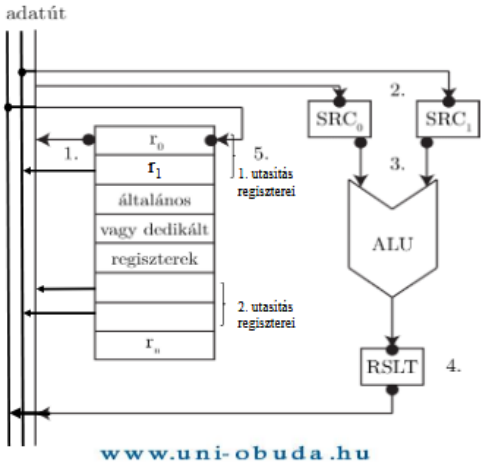
/ 2. Modul : Alkatrészek számítógép szinten #1 : A CPU /
/ Alcím: Alkatrészek CPU szinten #1 : ALU és Vezérlő /

+ Előtte röviden a CPU-ról általánosságban
  - A CPU feladata a műveletvégzés és a vezérlés
  - Kettő fő CPU típus van
    × Szinkron CPU: Van egy órajel generátora, ami alapján egy “ütem” szerint történik minden
      ~ Gyors és egyszerű, viszont mivel minden “ütemre” történik, ezért késleltetés léphet fel a rendszerben ha egy művelettel hamar végzünk, de még ugyanazon órajel alatt nem tudunk befejezni egy következő műveletet
    × Aszinkron CPU: Amint befejezünk egy utasítást, azonnal kezdjük a következőt
      ~ Nem lesz holtidő, viszont egy speciálisabb áramkör kell az utasítások végének érzékeléséhez, ami sokkal költségesebb és több időt is igényelhet mint egy szinkron CPU esetében

/ 2.1 Műveletvégző (ALU) /

+ ALU részei
  - Regiszterek
    × Kettő típusuk van
      ~ Látható regiszterek: Ebbe tehet adatokat a programozó. Lehetnek univerzálisak (bármilyen adat kerülhet beléjük) és dedikáltak (speciális feladat ellátására szánt regiszterek)
      ~ Rejtett regiszterek: Adatfeldolgozáshoz szükséges puffer regiszterek. Főként csak a CPU “nyúlhat” hozzájuk, a programozó nem *(mondjuk ha elég alacsony szinten dolgozik a géppel a programozó, akkor igen)*
  - Adatutak
    × Olyan mint egy vezetékrendszer
    × Összeköti a regisztereket, a puffer regisztereket és az ALU-t
    × Egyszerre csak egy adat lehet rajta
    × Nem lehet megcímezni, kontrollálni
  - Kapcsolópontok

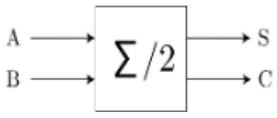
- × A regiszterek bemenetén és kimenetén lévő tranzisztorok, amik a kapcsolók állapotát változtatják
  - ~ Egy kimeneti kapcsoló állapota lehet 1, 0 vagy zárt
  - ~ Egy bemeneti kapcsoló állapota lehet nyitott vagy zárt
- Szűkebb értelemben vett ALU
  - × A legegyszerűbb logikai kapuk segítségével (AND, OR, NOT, XOR) vannak benne összeépítve különböző matematikai műveleteket végző áramkörök

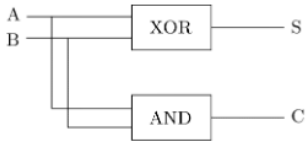


+ Néhány ilyen áramkör
  - Egybites félösszeadó: Kettő bemeneti bitet összead és tovább küldi az eredményt és a túlcsordult extra bitet ha van (1 + 1 esetén az eredmény 2 lesz, ami 10 binárisban és ez nem fér el egy bitben. Ha ilyen van akkor az C (carry) értéke 1 lesz és tovább küldjük)

Megvalósítása:
$S = \overline{A}B + A\overline{B} = A \text{ XOR } B$
$C = AB$

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





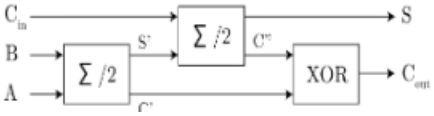
  - **Egybites teljes összeadó**: Összeadja a bemeneteket, aztán összeadja a kapott carry-t (C_{in}) a most keletkezett carry-vel (C') és ezután keletkezik túlcsordulás, akkor azt tovább küldi a végeredménnyel (C_{out})

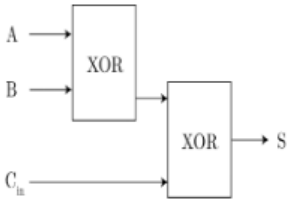
$S = \overline{A}\overline{B}C_{in} + A\overline{B}\overline{C_{in}} + \overline{A}B\overline{C_{in}} + ABC_{in} = \overline{(\overline{A}B + A\overline{B})}C_{in} + (\overline{A}\overline{B}+AB)\overline{C}_{in}$

Legyen: $X = A \text{ XOR } B$ ⟶ $\overline{X}\ C_{in} + X\ \overline{C}_{in}$

($\overline{A \text{ XOR } B}$ = A XOR B, A XOR B)

$S = \overline{X}C_{in} + X\overline{C}_{in} = X \text{ XOR } C_{in} = A \text{ XOR } B \text{ XOR } C_{in}$





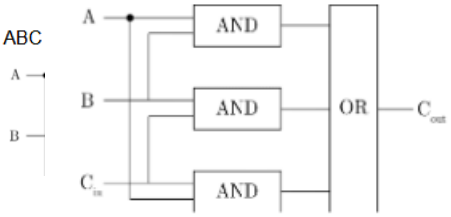
  - N bites soros összeadó (gyors és olcsó, de lassú)

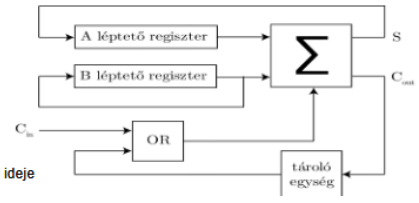
$C_{out} = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC = \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC + ABC + ABC$

Azonosságok: (A+A=A, AB+AB=AB, $A+\overline{A}$=1 !!!)

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

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





  - N bites párhuzamos összeadó (párhuzamos összeadásra képes ezért gyorsabb mint a soros HA nincs sok carry)
    - × Megoldás erre a CILA (carry look-ahead): Az összes bemenő paraméterből és a C_{in}-ből elő lehet állítani az összes carry értékét anélkül hogy a köztes carry-ket ismernénk

+ Egyéb fontos számolások
  - Fixpontos szorzás: Egy komplex és időigényes feladat *(kb x40 több órajel mint egy összeadás)*
    - × Lehet gyorsítani:

~ Szorzás bitcsoportonként

Szabály:

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

$$
\begin{array}{ll}
0111*10|01 & \\
\hline
0000 & \textit{gyűjtő} \\
01\,11 & \textit{egyszerese és léptetés kétszer} \\
\hline
01\,11 & \\
1\,11000 & \textit{kétszerese és léptetés kétszer} \\
\hline
1\,11111 &
\end{array}
$$

~ Booth algoritmus: Ha sok 1-es a szorzandó számunkban, akkor keressük meg a legközelebbi “kerek” (csupa 1-es) számot, szorozzuk meg azt a kívánt “X” számmal és abból végül vonjuk ki a “kerek” szám és az eredeti szám különbségének X-szorosát

Pl.: X*62 (111110) szorzást több lépcsőben kellene elvégezni. Helyette:
Keressük meg a szorzóhoz legközelebbi „kerek” számot és állítsuk elő az eredményt úgy, hogy:
X*(64-2) = X*64 – X*2

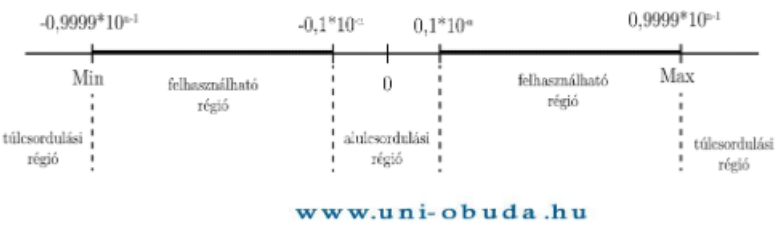
- 64 (1000000b) → sok léptetés + 1 összeadás
- 2 (10b) → 1 összeadás
- kivonás → 1 összeadás (negált-tal)

- Lebegőpontos számok: Törtszámok, melyek tizedes része és egészrésze szabadon választható
  - × Lebegőpontos számok részei:
    - ~ Mantissza (M): Az egészrész, ami csak egy bizonyos tartományban lévő érték lehet (pl. 10-es számrendszer esetén 0,1 <= M < 1; 2-es számrendszer alapján ½ <= M < 1)
    - ~ Radix (r): A számrendszer alapja (mennyivel kell megszorozni a mantiszszát hogy arrébb kerüljön a “tizedesvessző” 1-gyel)
    - ~ Karakterisztika (k): Hányszor kell a radix-szal megszorozni a mantisszát, hogy megkapjuk a kívánt számot
  - × pl. 1023,62 = 1,02362 * 10^3
  - × A lebegőpontos számok értelmezési tartománya függ a radixtól és attól hogy hány biten tároljuk a számot

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 bit: 32768)

  - × A pontosság pedig a mantissza bitszámától függ (minél több számjegyből áll a mantissza, annál kisebb számokat tudunk megjeleníteni)

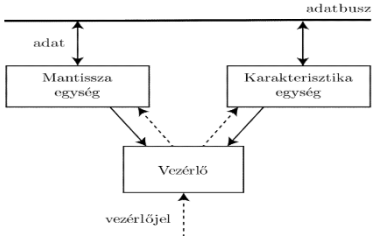
Példa: 10-es számrendszer, 4 mantissza bit esetén:



www.uni-obuda.hu

  - × A karakterisztika maximális értéke a ±∞ számára van

× A lebegőpontos számoknak tudnia kell kezelni az alul csordulást vagy a túlcsordulást
  ~ Alul csordulásnál:
    Δ Jelzi a hibát és az értéket 0-ra állítja vagy a denormalizált számot jelzi ki
  ~ Túlcsordulásnál:
    Δ Jelzi a hibát és vagy a legnagyobb érétéket jelzi ki vagy a megfelelő előjeles végtelent
× A lebegőpontos számok pontosságát lehet növelni rejtett bitek használatával
  ~ A mantissza normalizált alakban mindig 1 egész és valamennyi tört (pl. 1,1235234 vagy 1,00002)
  ~ Mivel az 1-es fix, ezért nem is szükséges eltárolni
  ~ Ezzel 1 bittel nő a pontosság
× A lebegőpontos számokhoz kapcsolódnak őrző bitek
  ~ A lebegőpontos számok nem tudnak minden számot 100%-os pontossággal eltárolni (pl. az 1/3 kettes számrendszerben végtelen hosszú)
  ~ Elvárás hogy a hibának kisebbnek kell lennie mint a normalizált eredmény legkisebb számjegye
  ~ A probléma akkor következik mikor pl. két különböző nagy karakterisztikájú számot akarunk kivonni egymásból. Ehhez egyforma értékű karakterisztikára (azonos kitevőre) kéne hozni a két számot ez viszont az egyik szám esetében azzal járna hogy meg kell növelnünk a karakterisztikáját, az viszont a mantissza méretének rovására járna => csökkenne a pontosság
  ~ Az őrző bitek pont ezt küszöbölik ki:
    Δ A mantissza csökkentésekor eltárolják az amúgy “elvesző” biteket és a mantissza újonnani növelésekor ezeket visszahelyezik a számba
× Lebegőpontos számok kódolása
  ~ A mantissza 2-es komplemensben van kódolva
  ~ A karakterisztika többletes kódolásban van
× Lebegőpontos számok formátumai
  ~ Szabványos:
    Δ Kötött és a háttértáron való tároláshoz használt
    Δ Lehet egyszeres (32 bites) vagy kétszeres (64 bites) pontosságú
  ~ Bővített
    Δ CPU-n belüli számolásra van és egy szabadabb formátum
    Δ Lehet egyszeres pontosságú vagy kétszeres
× Kerekítések
  ~ Legközelebbi értékre kerekítés
  ~ 0-ra kerekítés
  ~ Pozitív vagy negatív végtelen felé kerekítés
× A kiszámításukra külön műveletvégző volt



- × Fixpontos (FX) Vs Lebegőpontos (FP)
  - ~ Fixpotnos:
    - Δ Előny: Gyors, könnyen megvalósítható, kevés helyet igényel
    - Δ Hátrány: Kicsi az értelmezési tartomány, tört számoknál vagy osztásnál pontatlan lehet
  - ~ Lebegőpontos:
    - Δ Előny: Nagy értelmezési tartomány és általában elég nagy a pontossága
    - Δ Hátrány: Több erőforrást igényel, legtöbbször kerekíteni kell

- BCD számábrázolás
  - × A kódolása az pontos megfeleltetés kerekítési hiba
  - × Ábrázolás
    - ~ 4 biten
  - × Formátum
    - ~ Formázott 1 byte -> 1 számjegy
    - ~ Pakolt 1 byte -> 2 számjegy
  - × Hossza
    - ~ Fix
    - ~ Változó
  - × Művelet végzés
    - ~ pl: 8 + 7 = 15
    - ~ 1000 + 0111 = 1111 <- érvénytelen tetrád ezért kivonunk belőle 10-t vagyis hozzáadunk -10-t (1010 = 10) 0110 = -10 Kettes komplemenssel képezve
    - ~ 1111 + 0111 = 1 | 0101 nagyobb helyiértéken megjelenik az egyes majd utána pedig az 5 és így lesz 15d
  - × Teljesen pontos, de cserébe több helyet foglal (~40%) és lassabb, de elhanyagolható

+ Egyéb ALU által végzett műveletek
  - Boole algebra összes művelete (NOR, AND, XOR stb)
  - Léptetés, invertálás, komparálás
  - LOAD/STORE címszámítás
  - Karakteres műveletek

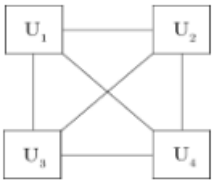
/ 2.2 Vezérlőegység /

- A CPU egyik legbonyolultabb áramköre, egyben a processzor “lelke”
- Benne található az ütemező
  - × Ő felelős a vezérlőjelek előállításáért, amiknek még szinkronban is kell lennie az órajellel

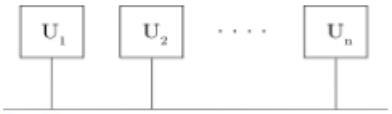
+ Két vezérlési elv van:
  - Huzalozott vezérlés
    - × Fix, csak áramköri elemekből áll -> gyors, viszont nehezen átlátható és módosítható
  - Mikroprogramozott vezérlés
    - × Minden CPU utasítás mikroutasításokból tevődik össze
    - × Rugalmasabb, áttekinthetőbb, olcsóbb mint a huzalozott megoldás és módosítható is, viszont mindig lassabb

/ 3. Modul : Alkatrészek számítógép szinten #2 : Buszrendszerek /

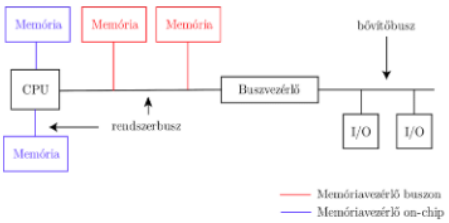
+ Mi a buszrendszer?
  - Kizárólag ezen keresztül kommunikálnak az egységek egymással egy szervezett és egységes módon
  - Egyszerre több egység van rákapcsolva ezért…
    × Meg kell valahogyan jelölni egy átvitelben mely eszközök vesznek részt
    × Meg kell határozni az átvitel irányát (honnan, hova)
    × Meg kell oldani az átvitelben résztvevő eszközök szinkronizálását
+ Buszrendszerek csoportosítása
  - Átvitel iránya szerint
    × Szimplex (csak egy irányba)
    × Félduplex (mindkét irányba, de egyszerre csak egy irányba)
    × Full-duplex (egyszerre két irányba is)
  - Átvitel jellege szerint
    × Dedikált: Minden egység mindenkivel össze van kötve
      ~ Gyors és közvetlenül (köztes fél nélkül) tud két egység kommunikálni egymással
      ~ Merev struktúra és nehezen bővíthető



    × Megosztott: Van egy közös busz és arra vannak rákapcsolva az egységek. Buszvezérlő utasításokra van szükség hogy ne történjen ütközés
      ~ Olcsó, egyszerű megvalósítani és könnyen bővíthető
      ~ Viszonylag lassú, bonyolult vezérelni és ha meghibásodik akkor több eszköz is kieshet



  - Átviteli mód szerint
    × Soros
    × Párhuzamos (pl. memória)
  - Összekapcsolt területek alapján
    × Rendszerbbusz: Adatbusz + címbusz
    × Bővítőbusz: pl. USB, PCIe
  - Átviteli tartalom szerint
    × Címbusz: Eszközök címzésére szolgál
    × Adatbusz: Adatokat juttat el az operatív tárból a perifériák és a CPU között
    × Vezérlővonal: Az időzítőjelek és az egységek állapotáról szóló információk átvitelére vannak



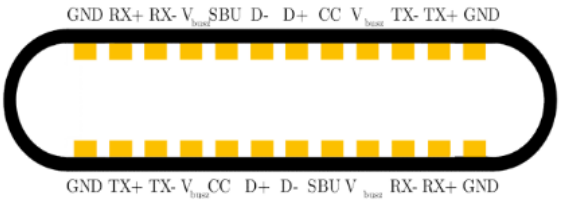
⬇

+ Vezérlőjelek típusai
  - Adatátvitelt vezérlő jelek
    × Memory/”Input/Output”: megmondja, hogy a buszon memória vagy periféria van címezve
    × Read/Write: adatáramlás irányát adja meg a CPU felől nézve
    × Word/Byte: megadja az adat méretét
    × D/S (data strobe): az adat felhelyezését jelzi a memória számára
    × A/S (address strobe): a cím felhelyezését jelzi a memória számára
    × ReaDY: az átvitel befejezését / a busz rendelkezésre állását jelzi
  - Megszakítást vezérlő jelek
    × Megszakítást kérő jel
    × Megszakítást visszaigazoló jel
  - Buszvezérlő jelek
    × Busz kérése
    × Busz foglalása
    × Busz visszaigazolása
  - Egyéb vagy speciális vezérlőjelek
    × Reset
    × CLK
+ PCI Express (PCIe)
  - Egy nagy sebességű soros busz
  - Hot-plug funkcióval rendelkezik (menet közben is le lehet kapcsolni)
  - Teljesítménye a CPU-tól is függ

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!	

PCI Express Vs Sima PCI (PCIe a lényeges)

+ USB-C szabvány
  - 4 érpárt tartalmaz adatátvitelre
  - 24 pines csatlakozója van (2x12-es szimmetrikus kiosztásban)
  - USB-C pinek
    × Mindkét végen van 1-1 földcsatlakozó (GND)
    × A TX+ és TX- nagysebességű adatküldésre használt vezetékpár
    × Az RX+ és RX- nagysebességű adatfogadásra használt vezetékpár
    × A V_{busz} feladata az áramellátás
    × A D+ és D- az USB 2.0-ás adatátvitelt végzik
    × A CC és SBU alternatív vezetékek



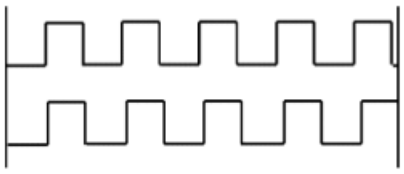
+ USB által szállított csomagok típusai
  - Időkritikus csomag: Állandó sebességgel közlekednek (pl. audio- és videóadatok)
  - Nagy adatcsomag: Alacsony prioritásúak, sok adat továbbítására szolgálnak (pl. backup)

- Megszakítási csomag: Az egységek kiszolgálási kéréseire használják. Kevés adatot tárolnak és ciklikusan kérdezik le
- Vezérlési csomagok: Címkiosztáshoz, eszközök azonosításához használják. ún. handshake elven működnek

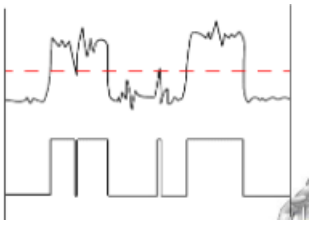
+ Párhuzamos buszok Vs Soros buszok
  - Soros buszok:
    × Elég akár egy vezetékpár is
    × A biteket bitsorozatként lehet átküldeni (viszont ez plusz hardvert igényel)
    × Egy gyors soros érpáron több adat továbbítható mint több lassún
    × Rugalmasan frissíthető, akár szoftveresen is
    × Magas frekvencián sem okoz a (mindjárt elmagyarázott) “jitter” problémát, ezért nagyobb távolságra is tudja biztosítani az adatátvitelt
  - Párhuzamos buszok:
    × Több vezetéket használ, ami egyszerre több adat átvitelét biztosítja
      ~ Emiatt a CPU és a memória közt párhuzamos adatátvitel van
    × A sok vezeték hátránya hogy drágább, több helyet foglal és komplexebbek is
      ~ Ennek ellenére hardveres vonatkozásban könnyen implementálhatók
    × Magasabb frekvencián problémák jelennek meg, az ún. “jitter” formájában

⬇

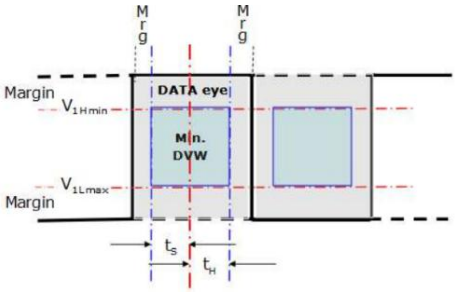
+ Jitter formái:
  - Delay Skew (időbeli eltérés)
    × Elég magas frekvencián már igen rövid vezetékhossznál is megtörténik, hogy a párhuzamos adatok nem egyszerre érkeznek meg



  - Elektromágneses interferencia (EMI)
    × Egy olyan “zaj”, ami a vezeték belső és egyéb külső elektromágneses sugárzásától alakul ki és torzítja a jelet
      ~ Ha emiatt pl. egy érték kiesik az értelmezési tartományból, akkor azt nem tudjuk értelmezni
      ~ Ez logikai értékek estén nagy probléma mert a pontos érték eléréséhez a jelnek egy bizonyos frekvenciában kell lennie egy adott ideig



  - Vezetékek közti keresztirányú áthallás
    × Minél hosszabb a vezeték, annál nagyobb az áthallás

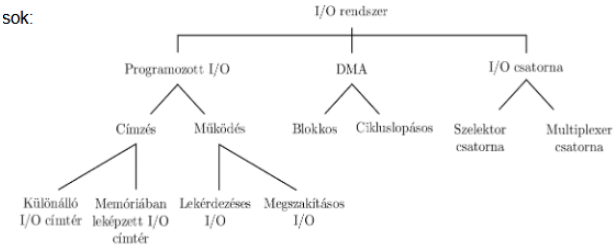


- Modern technológiák: HyperTransport (HT) rendszerbusz
  - × Egy kétirányú soros/párhuzamos szélessávú, alacsony késleltetésű kapcsolat
  - × Fő feladata a front-side bus kiváltása
  - × A CPU lapkájára van integrálva, de használják nagy sávszélességű I/O buszként is
  - × Kétféle egységet tartalmaz
    - ~ Alagút (tunnel): Végén található két HT-port, amiknek segítségével több HT-egységet össze tudunk fűzni egymással
    - ~ Barlang (cave): Ez zárja le a egy sorral feljebb említett HT láncot
  - × Memóriában leképzett I/O-val rendelkezik
- Modern technológiák: QuickPath Interconnect (QPI)
    - ~ Feladata az FSB kiváltása volt
    - ~ Az QPI-t használó processzorok lapkára integrált memória-vezérlőkkel és non-uniform memory access-el (NUMA) rendelkeznek
    - ~ Minden QPI port 2 darab 20-20 adatvonalas pont-pont linkből áll
    - ~ 5 réteges architektúrát használ: A fizikai réteg 1 órajel alatt a 20 vonalnak köszönhetően egyszerre 20 adatbitet tud párhuzamosan átvinni
    - ~ Továbbfejlesztése az UltraPath Interconnect (UPI): gyorsabb, energiatakarékosabb

/ 4. Modul: Alkatrészek CPU szinten #2 : I/O rendszer /

+ Főbb I/O típusok, részek: *(ezek igazából egymásra épülő elemei az I/O rendszernek)*
  - Programozott I/O
  - DMA
  - I/O csatorna

sok:



Ebben a modulban átvett típusok és a hozzájuk kapcsolódó fontos tulajdonságok

+ Programozott I/O
  - A CPU által irányítottan történik az adatátvitel
  - A processzor foglalkozik minden I/O művelettel (indítás, irányítás, lezárja)
  - Egyszerű a megvalósítása, de jelentős processzor időt igényel
  - Közös buszt használ a többi egységgel
  - Egy programozott I/O lehet lekérdezéses vagy megszakításos vezérlésű
  - Azokat a regisztereket, amiken keresztül a processzor a perifériákkal kommunikál I/O portoknak nevezzük és ezek a vezérlőkártyán helyezkednek el

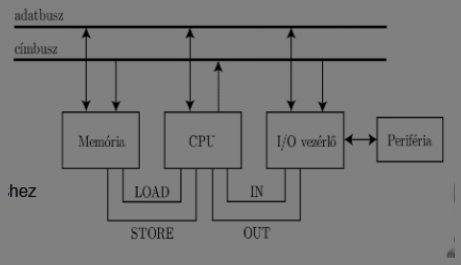
⬇

+ Az I/O port részei
  - Parancs regiszter: A CPU ide írja be a “kívánságait”
  - Adatregiszterek: A bővítőbusz “végállomása”, amin belül két típusú regiszter van
    × Data input regiszter: Ebből olvassa be a CPU a perifériától kapott adatokat
    × Data output regiszter: Ebbe írja bele a CPU az adatokat a periféria számára az adatokat
  - Állapotregiszter: Állapotinformációkat közöl a perifériáról (leggyakoribb állapot a Ready vagy Busy)

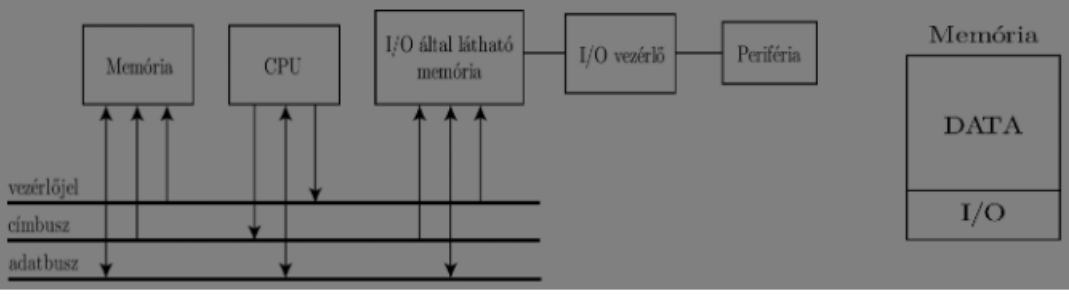
  - Lehet közös az állapot és a parancsregiszter vagy a data input és output

  - Egyéb regiszterek
    × Jelenlét jelző regiszter: Jelzi hogy van-e eszköz kapcsolva az adott I/O portra
    × Eszköz tulajdonságait tartalmazó regiszter
    × Bonyolultabb periféria esetén egy funkcióhoz több regiszter is tartozhat

+ Programozott I/O címterei: *(a szürkével jelölt részek nem nagyon vagy egyáltalán nem szokott számon kérve lenni, ez inkább csak a megértéshez és a vázlatpontok közti kapcsolathoz van itt)*
  - Különálló I/O címtér
    × A CPU két címteret lát: a perifériák címtere, a memória címtere
      ~ Ugyanaz a cím lehet memóriacím és I/O cím is!
      ~ Ennek kiküszöbölésére van egy ún. M/IO (Memória/IO) vezérlőjel, ami megmondja hogy az adott pillanatban a címsínje I/O cím vagy memóriacím van
    × A CPU a memóriával LOAD/STORE utasításokon át kommunikál
      ~ Megcímzi a memóriát és adatátvitellel kommunikál
      ~ A CPU ugyanígy kommunikál az I/O vezérlővel is: Megcímzi az I/O vezérlőt de **különálló I/O utasításokkal** biztosítja az adatátvitelt



    × Előnye: Egyszerű és olcsó a megvalósítás
    × Hátránya: Terheli a CPU-t és plusz egyedi utasítások kellenek csak az I/O kezeléséhez
  - Memóriában leképzett I/O címtér
    × A CPU lefoglal a perifériák számára egy területet a memóriában
    × A vezérlőjelen biztosítja a CPU a vezérlő információkat
    × Egy memória hivatkozásból I/O utasítás lesz ha olyan címre megy a hivatkozás, amit az I/O is lát
    × Előnye:
      ~ Nem kell külön I/O utasítás, lehet vezérelni ugyanúgy LOAD/STORE utasítással
      ~ Az I/O vezérlő hozzáfér a rendszerbuszhoz, amitől gyorsabb lesz az átvitel
    × Hátrány: Továbbra is terheli a CPU-t



  - Memóriában leképzett I/O működésének módjai (adatátviteli módszerek alapján)
    × Feltétel nélküli adatátvitel:
      ~ Feltétele hogy a perifériáknak mindig adatátvitelre alkalmas állapotban kell lennie

~ Nincs szükség ellenőrzésre sem az adat átvitele előtt sem utána
~ Semmilyen szinkronizálás nincs a CPU és a periféria között

⬇

~ Előnye hogy gyors, hátránya hogy mivel nincs visszacsatolás ezért előfordulhat az adatvesztés

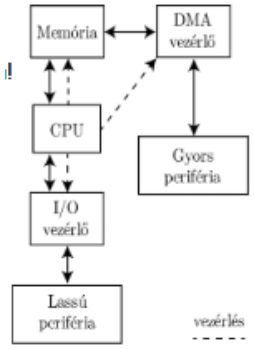
× Feltételes adatátvitel: Kettő feltételtípus van
~ Lekérdezéses adatátvitel
Δ A CPU elküldi “kívánságait” az I/O vezérlőbe és addig “kérdezgeti”, amíg nem kap választ
Δ Nagyon pazarló működés, mert a periféria sokkal lassabb
Δ Megoldás: A megszakításos adatátvitel

⬇

~ Megszakításos adatátvitel:
Δ A CPU itt is beírja a “kívánságait” az I/O vezérlőbe és közben elkezd mással foglalkozni
Δ A periféria vezérlő rendelkezik egy megszakítási rendszerrel
Δ Ha megtörténik az adatbeolvasás, akkor küld egy megszakítás felkérést a CPU-nak
Δ A következő utasítás töréspontban a CPU megnézi a vezérlő állapotregiszterét és ha “Ready” állapotot lát benne, akkor megindul az adatátvitel
Δ Előnye: Kevésbé “köti le” a CPU-t (“több ideje van dolgozni”) és sokkal gyorsabb is ez a megoldás
Δ Hátránya: Nagy mennyiségű adat esetén lassú és sok megszakítást igényel és még mindig a CPU-nak kell vezérelnie az adatátvitelt

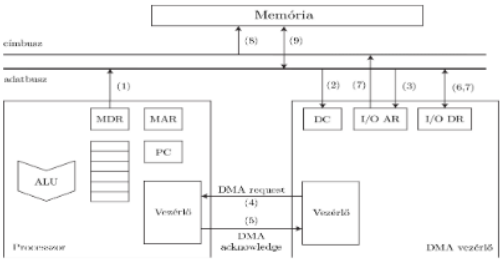
⬇

+ **DMA (Direct Memory Access)**



- Gyorsítja az adatátvitelt a perifériákról és még inkább leveszi a terhet a CPU-ról
  × Nagy mennyiségű adat esetén használt
  × Gyors perifériáknál használt
  × A CPU közreműködése nélkül végzi az adatátvitelt, a CPU csak a vezérlést biztosítja
  × (a lassú perifériáknál marad a helyzet: I/O vezérlő dolgozik a CPU közreműködésével)
- A DMA el van látva bizonyos regiszterekkel (DC, I/O címregiszter, I/O adatregiszter, belső transzparens regiszterek), amiket először fel kell paraméterezni, hogy tudja vezérelni a buszt és hozzáférjen az adattárhoz közvetlenül
- **A felparaméterezés a CPU által, a programozott I/O-n keresztül történik és ezeket a szükséges információkat viszi át:**

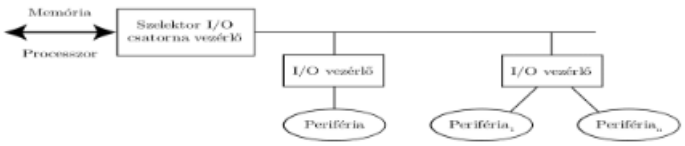
- × Írás- vagy olvasási művelet (igazából az átvitel iránya)
- × I/O egység címe
- × Memória cím kezdőértéke I/O AR-be (ahonnan olvasunk, vagy ahová írunk)
- × Átvivendő adat típusa (byte, félszó, szó)
- × Olvasandó/írandó egységek száma (Ez a DC-be kerül)
- × Átvitel módja (blokkos vagy ciklusloppásos)
- × Kik között megy az adatátvitel (memória-memória, I/O-memória, I/O-I/O)
- × A DMA csatornáihoz prioritási értékeket rendelhetünk



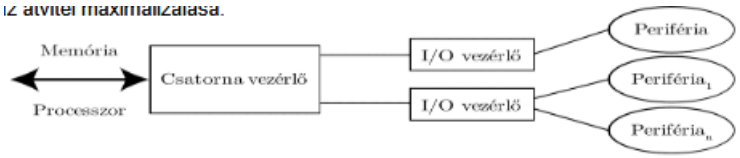
- Blokkos adatátvitel folyamata: *(ritkán elforduló kérdés)*
  - × CPU felparaméterezi a DMA vezérlőt és elindítja
  - × A DMA vezérlő küld egy “DMA Request” jelet, amivel kéri a CPU-t hogy adja át a rendszerbusz kezelés jogát neki (ezzel megszakítva CPU éppen folyó folyamatát)
  - × A CPU küld egy “DMA Acknowledge” jelet és lemond a rendszerbusz használatáról
  - × A DMA vezérlő bekéri a perifériától az első átvinni kívánt adatblokkot és azt beírja az I/O DR-be (I/O DataRegister?)
  - × A DMA vezérlő átküldi az adatot az I/O DR-ből a rendszerbuszon keresztül arra a memóriacímre, amit az I/O AR megadott
  - × A DMA vezérlő csökkenti a DC-ben található olvasandó/írandó egységek számát 1-gyel
  - × A DMA ellenőrzi a DC-t hogy vannak-e még adategységek, amiket tovább kell küldeni:
    - ~ Ha igen, akkor “az aláhúzott vázlatponttól” újrakezdi a folyamatot
    - ~ Ha nem, akkor megszakításkéréssel jelez a CPU-nak, hogy megvolt az adatátvitel, innentől újra vezérelheti ő a rendszerbuszt.
  - × A CPU is leellenőrzi, hogy megtörtént-e az adatátvitel és végül visszaveszi a kontrollt

+ I/O csatorna
  - A DMA kiterjesztése a lassabb perifériákra
  - Ebben az esetben a CPU nem hajt végre I/O műveletet, hanem csak kezdeményezi (elindítja) őket
  - Az I/O csatorna vezérlőjének utasításai a memóriában vannak tárolva, amiket onnan is hajt végre (nem kell külön felparaméterezni, azaz elküldeni minden fontos tudnivalót az adatátviteli “rendszerről” mint a DMA-nak)

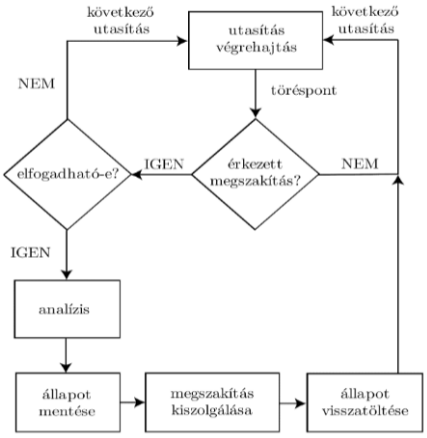
- Kettő típusú csatornája van
  - × Szelektor csatorna
    - ~ Erre a csatornára kerülnek a lassabb eszközök közül is a gyorsabban működők *(mondhatni hogy a közepesen gyors I/O eszközök csatornája)*
    - ~ A vezérlőjébe egyetlen I/O vonalból vezet, amit közösen használnak a rácsatlakoztatott I/O vezérlők.
    - ~ Egyszerre csak egy I/O vezérlő tudja használni (ezért szelektor) és ezen keresztül tud kommunikálni a CPU-val és a memóriával



  - × Multiplexer csatorna
    - ~ A kifejezetten lassú perifériáknak van fenntartva, akik folyamatosan küldik az információt, de nagy szünet van az adatok között
    - ~ A csatorna vezérlőjére több I/O vezérlő tud párhuzamosan csatlakozni
      - Δ Mivel elég lassúak a perifériák így a csatorna vezérlőjének akkor is csak 1-1 I/O vezérlővel kell foglalkoznia mikor több is rá van kötve
    - ~ Két típusai van az alapján hogy milyen formátumban vannak az adatok átküldve: byte multiplexer, blokk multiplexer
    - ~ Célja az átvitel maximalizálása



/ 5. Modul : Alkatrészek CPU szinten #3 : Megszakítási rendszer /



Megszakítási rendszer folyamatábrája

+ Mi a megszakítás definíciója?:
  - A feldolgozás szempontjából váratlan események kezelésére szolgáló művelet
  - Reagál és biztosítja a rendszer optimális működését
  - Lényege, hogy a CPU figyelmét csak tényleges szükség esetén foglalja le
+ Megszakítási okok és források (prioritási sorrendben)
  - Géphibák
    × Ezek nem letiltható megszakítások
    × Különböző hibafigyelő áramkörök jelzési tartoznak ide, aki legtöbbször egy hibakóddal jelzik a problémát
  - I/O források
  - Külső források (pl. reset gomb, hálózati kommunikáció)
  - Programozási források (több fajtájuk van)
    × Szándékos: Ilyenkor egy programszándékosan megszakítást kér
    × Nem várt: Utasítások végrehajtása közben kialakuló hiba

⬇

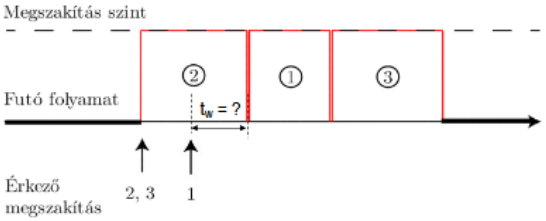
  - Programozási okokból fellépő megszakítások
    × Memóriavédelem megsértése
      ~ Ha egy program egy “nem megengedett” memóriaterületre mutat és ezzel véletlen felülírna mondjuk egy másik programot
    × Tényleges tárkapacitás túlcímkézése
      ~ Ha egy elméletileg kiadható legnagyobb cím nagyobb mint a tényleges tárkapacitás, akkor futhatunk ilyen hibába
    × Címzési előírások megsértése
    × Aritmetikai logikai végrehajtás közben történő hibák (pl. 0-val osztás, lebegőpontos (FP) számoknál overflow vagy underflow)

+ Megszakítások csoportosításának módja
  - Szinkron vagy aszinkron
    × Szinkron: Mindig ugyanott előforduló (int szám túlcsordulása)
    × Aszinkron: Véletlenszerű (lehet várható és váratlan)
  - Utasítások végrehajtása között vagy közben
  - Felhasználó által kért (pl. debug) vagy nem kért (pl. hardverhiba)
  - A megszakított program a megszakítás után tud folytatódni vagy nem
  - Maszkolható (ha kell akkor letiltható) vagy nem maszkolható (pl. súlyos hardverhiba)

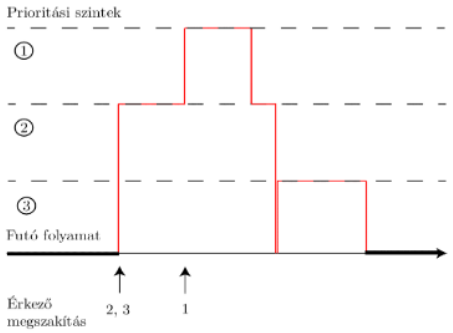
+ A megszakítás kiszolgálásának általános folyamata
  - Előkészületek: Ha valamelyik egység megszakítást kér akkor aktiválódik az INTR *(intercept)* vezérlővonal. Ezt a vezérlővonalat a processzor minden utasítások közti töréspontban megnézi
  - **Ha van megszakítás akkor jön a kérdés: Mikor elfogadható egy megszakítás (mikor juthat érvényre)?**
    × Ha az aktuális folyamat megszakítható (legyen az program vagy egy másik, fontosabb megszakítás)
    × Ha elég nagy a prioritása
    × Ha az a fajta megszakítás nincsen maszkolva (azaz letiltva)
  - Ha a megszakítás megtörténhet akkor a CPU aktiválja az INTACK *(interception acknowledgement?)* vonalat
  - Mikor a megszakítást kérő egység megkapja ezt az INTACK jelet akkor deaktiválja az INTR vonalon küldött jelét
  - A CPU kimenti a futó program kontextusát a tárolóba és annak indító kezdőcímét a betölti a Program Counter (PC) regiszterbe

Példák 3 prioritási szintes rendszereken (1 > 2 > 3)

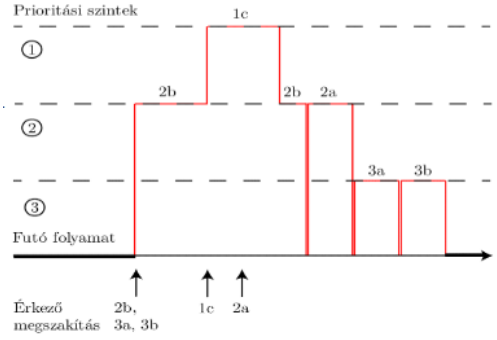
+ Megszakítási rendszerek szintjei
  - Egyszintű: Ha elindult egy megszakítási folyamat, akkor azt nem lehet megszakítani, hiába van egy nagyobb prioritású megszakítás kérve
  - Többszintű: Egy folyamatban lévő megszakítási folyamatot is meg lehet szakítani ha valami nagyobb prioritást élvez
    × Problémája, hogy rengeteg megszakítás van egy valós rendszerben és nem lehet mindegyikhez külön prioritási szintet rendelni tehát ha van egy fontos megszakításunk annak még így is nagy eséllyel várnia kell
  - **Többszintű, többvonalú**
    × A megszakításokat osztályozzuk (1-es, 2-es 3-as osztály) és osztályozzuk és azokat is priorizáljuk (1-es osztály prioritása 1, 2-es osztály prioritása 2, stb...)
    × Az osztályokon belül is prioritást adunk a megszakításoknak (“a” prioritású megszakítás, “b” prioritású, stb...)
    × Osztályok között többszintű a rendszer, egy osztályon belül pedig egyszintű a megvalósítás



Egyszintű



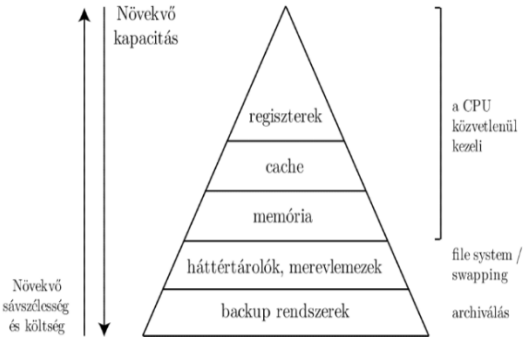
Többszintű



Többszintű és többvonalú

/ 6. Modul : Gyorsítótár (Cache) /

+ A gyorsítótár definíciója
  - Az adatok és utasítások átmeneti tárolására szolgáló gyors működésű tároló
  - Önálló szerepe nincsen, hanem mindig az operatív tárban lévő adatok (egy részének) másolatát tartalmazza
  - A programozó számra nem elérhető, nem címezhető. Ezt a feladatot a hardver látja el
  - Legtöbbször a processzor lapkáján helyezkedik el
  - A regiszterek után a leggyorsabb adatátvitelt biztosítja



Csak érdekesség: A tárolók "piramisa"

+ Felosztás *(ritkán kérdés, de azért fontos)*
  - Manapság külön cache van az utasításoknak és az adatoknak. Azért mert:
    × Az adatok és az utasítások általában egymástól függetlenül kezelhetők
    × Ha egy program vagy programkészlet annyira sok adatot igényel, hogy a cache-ből kikerülhetnek az utasítások, az lelassítja a folyamatot
    × Egy utasítás cache-nél elég az olvasás sebességét gyorsítani, míg egy adat cache esetében fontos az írás és olvasás gyorsasága is

+ Cache szintek
  - A gyorsítótárak tervezésénél két tényezőt kell mérlegelni: Sebesség Vs Találati arány
    × A gyorsítótárban eltárolt adatok közül mindig meg kell keresni azt ami éppen nekünk szükséges. Minél nagyobb a tároló, annál több idő telik el a kereséssel
    × A cél egy gyors, de nagy kapacitású gyorsítótár lenne, de ez egy **ellentmondás**

	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 érdekesség: A különböző szintű cache-ek elérési ideje

  - A “balance” megtalálása érdekében vezették be a többszintű cache-t
    × A Level 1-es (L1) cache a leggyorsabb, de a legkisebb is. Őt követi a L2, L3 majd a RAM
+ Adatok a cache-ben
  - Mivel a cache tartalma a memóriának a másolata, ezért figyelni kell arra, hogy ha egy érték megváltozik a cache-ben, akkor azt a memóriában is meg kell változtatni

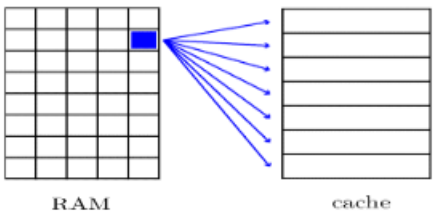
- A cache és a memória között az adatoknak blokkos formában történik az adatátvitele, míg a cache és a CPU között byte szinten is lehet
- Mivel a cache mérete mindig kisebb mint a memória mérete, ezért fontos eldöntenünk, hogy mit töltsünk a cache-be
- A cache-ben nem csak a memória tartalmát tároljuk, hanem azt is, hogy melyik memóriacímről másoltuk ki az adatokat
  - × Ezeknek a memóriacímeknek csak egy részét kell eltárolnunk. Csak annyit hogy vagy a tárolt értékből (közvetlenül) vagy a tárolt érték + a cache-ben lévő pozíciójából (közvetetten) meg tudjuk mondani, melyik blokkot másoltuk ki

- Ezeket a címeket TAG-eknek nevezzük
  - × Egy TAG lehet...
    - ~ Fizikai, ha a CPU és az MMU között helyezkedik el (Memory Management Unit)
    - ~ Virtuális, ha az MMU és a RAM között helyezkedik el
      - Δ A virtuális cím csökkenti a cache hiba késleltetését, de több helyet is foglal a cache-ben és még egy “virtuális -> fizikai” konvertálást is el kell rajta végezni

+ Visszakeresés
  - A visszakeresés módja a tartalom szerinti asszociatív keresés, amikor is a CPU megnézi, hogy van-e ugyanolyan című adat a cache-ben mint amit ő a memóriában keres
    - × Ha igen, akkor az adat benne van a cache-ben és gyorsabban el tudtuk érni mint a memóriából. Ez a **cache hit**
    - × Ha nem, akkor az adat nincs benne a cache-ben hanem a “lassú” RAM-ból kell kiolvasni és betölteni a cache-be. Ez a **cache miss**
+ Helyettesítési stratégiák
  - Alapértelmezett hogy a cache mindig tele van és ha új adatot akarunk beleírni akkor azt úgy kell kicserélni, már a cache-ben lévő adattal, hogy a találati arány megfelelően magas maradjon
  - Helyettesítési stratégiák:
    - × FIFO – A legrégebben betöltött adatblokkot írjuk felül
    - × LIFO – A legutoljára betöltött blokkot írjuk felül
    - × LFU (Least Frequently Used) – A legritkábban használt blokkot...
    - × LRU (Least Recently Used) – A legrégebben használt blokkot...
+ Vezérlő bitek
  - A cache-ben nem csak az adatok és azok memóriabeli címe van eltárolva hanem az adatok állapota is
  - Ezeket az állapotinformációkat különböző vezérlő bitek tárolják:

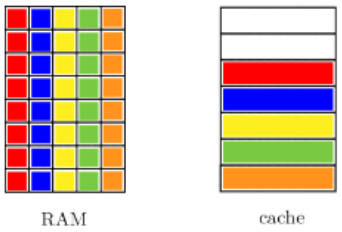
    - D (dirty) bit: Ha aktív, akkor az adott blokk valamelyik részét vagy felülírták vagy módosították. Addig nem lehet lecserélni másik adatblokkra, amíg a benne történt változások nem kerülnek át az operatív tárba is
    - V (valid) bit: Jelzi az adott szegmens érvényességét. Ha aktív ez a bit, akkor ott tárolt adat érvényes, tényleg megtalálható az általa tárolt memória címen. Ha deaktiválódik (pl. a cache line törlésénél), akkor az egy jel a CPU-nak hogy az adott cache line felülírható új adattal
+ Cache-eket jellemző fontos paraméterek
  - Méret
  - Elhelyezkedés (CPU lapkán vagy sem)
  - Blokkméret: Egyszerre mennyi adatot lehet a memória és a cache között mozgatni
  - Sorméret: Egyszerre mennyi adatot lehet összehasonlítani (kisebb vagy ugyanakkora mint a blokkméret)
  - Használt helyettesítési stratégia
  - Adat aktualizálási módszer
    - Write through: Ha változik egy adat a cache-ben akkor azt azonnal megváltoztatjuk az operatív tárban is
    - Write back: Csak akkor írjuk át a módosításokat a memóriában is ha a cache-ből el fog tűnni a módosított rész, addig nem
  - Koherencia mechanizmus: Módszer ami biztosítja hogy a fő tár és a cache tartalma egyezik
  - Átlagos elérési idő (AAT): “Cache hit valószínűsége” + (“Cache miss valószínűsége” x “A cache miss miatt a memóriából való kiolvasással járó extra idő”)

+ Cache típusok:
  - Full associative
    - Egy beolvasott adatblokk bármelyik cache line-ba bekerülhet, majd a helyettesítési stratégia alapján kiderül melyikbe
    - Mikor a CPU elkezd egy ilyen cache-be adatot keresni akkor az összes cache line TAG-jét egyszerre kezdi el vizsgálni
      ~ Ehhez viszont annyi párhuzamos összehasonlító áramkör kell amennyi cache line van

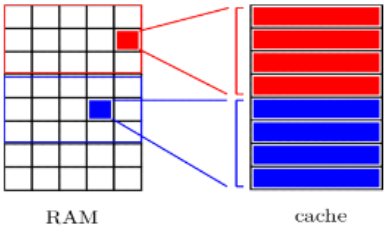


    - Konklúzió: Nagy találati arány és rugalmasság, de drága, bonyolult és nagy a fogyasztása
  - Direct mapping (1 way set associative cache)
    - Minden memóriablokk csak egy bizonyos cache line-ba kerülhet, amit a TAG határoz meg

    - × Előnye hogy csak 1 összehasonlító áramkör kell
    - × Hátránya hogy 1 line-t lehet sokszor kell cserélni, ami miatt ez a típus rugalmatlan lesz és kisebb lesz a találati arány is (ergo kevesebb haszna lesz az egész cache-nek)



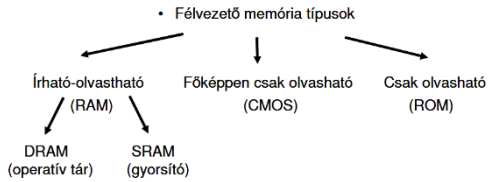
- **N-way associative cache**
    - × Egy memóriablokk csak “N” darab cache line-ba kerülhet (az “N” általában 2 hatványa szokott lenni pl. 2, 4, 8, 16, stb...)
    - × Így csak “N” darab összehasonlító áramkör szükséges
    - × Egy középút az előző két megoldás között: Gyorsabb és olcsóbb mint a “direct mapping”, de pontosabb és rugalmasabb mint a “1 way”



+ Cache line felépítése
    - Directory Entry: TAG és az állapotjellemzők
    - Tprot: TAG protection bitek (védenek az elírások ellen)
    - Dprot: Data protection bitek (védenek az elírások ellen)
    - State: Állapotbitek (pl. V bit és D bit)
    - Data: Maga az adat
+ Adatszervezési módok
    - Többszintű cache-ek esetén két típusú gyorsítótárjaink lehetnek
        - × Inclusive cache: Egy magasabb szintű tároló tartalmazhatja az alacsonyabb szintű tároló adatait
        - × Exclusive cache: A tárolók egymástól függetlenek, nem tartalmazzák egymás adatait
+ Cache koherencia mechanizmusok
    - Több magos CPU-k estén biztosítani kell hogy minden cache-ben ugyanazok az adatok legyenek tárolva
    - Egy koherencia mechanizmus az ezt biztosító módszer
    - Cél hogy a változás minél hamarabb bekerüljön minden mag cache-ébe
    - Érvényesítés módjai
        - × Invalidáció: Az adott cache line valid (V) bitjét először mindenhol 0-ra állítjuk és csak akkor írjuk át 1-re ha sikerült az egyik magtól elkérni a helyes adatot és átírni
        - × Felülírás: A változást az adott CPU mag direktben továbbküldi a többi magnak, hogy írjak felül az adott cache line-t
    - Koherencia protokollok:
        - × Snoopy
        - × Snorf
        - × Könyvtár alapú
        - × **MESI:** Különböző állapotjelzőket vezetünk be
            - ~ Modified: Egy adat módosult, de még nem lett visszaírva. Ilyenkor ő az egyetlen “valid”, a többi cache-ben az adott line “invalid”. Csak

az egyetlen “valid” line példányt lehet módosítani, a többi magban ugyanezt a line-t még nem lehet
    - Exclusive: A módosult adat már vissza lett írva az operatív tárba, de még nem lett elküldve a “többieknek”. Csak ő “valid” még mindig és csak ezt a példányt lehet módosítani, a többi magban ugyanezt a line-t még nem lehet
    - Shared: A cache line és a memória tartalma egyezik itt is és a többi tárban is
    - Invalid: Érvénytelen
- **MOESI:** A MESI kibővítése egy új állapottal:
    - Owned: Ebben az állapotban először el kell küldeni a változást a többi cache-nek és utána beírni a memóriába
    - Ehhez lehetővé kell tenni azt, hogy a cache-ek a memória elérése nélkül (közvetlenül) tudjanak egymással kommunikálni
    - Ez akkor előnyös ha két CPU mag gyorsabban tud egymással kommunikálni mint egy mag a memóriával

/ 7. Modul : Memória /



+ Memóriák csoportosítása
  - Írható és olvasható (RAM): Nem maradandó tárak
    - × Operatív tár (DRAM)
    - × Gyorsító (SRAM)
  - Főképp csak olvasható (CMOS): Nagyon alacsony feszültségen, kevés árammal működik és a gép kikapcsolása után is képes megtartani az adatokat
  - Csak olvasható (ROM): Ezen tárolódnak a számítógép elindításához szükséges programok (pl. BIOS) és adatok (pl. hálókártya MAC-címe)
+ Statikus RAM Vs Dinamikus RAM
  - Statikus RAM (SRAM)
    - × A rajta tárolt adatok addig maradnak meg amíg van tápfeszültség, nem kell az adatokat frissíteni
    - × Az adatokat egy néhány (4-6) tranzisztorból álló flip-flop memóriában tárolja
    - × Energiatakarékosak és gyorsak (gyorsabbak mint a DRAM), viszont drágák
    - × Számítógépben megtalálható statikus memória a cache és a regiszterek
  - Dinamikus RAM (DRAM)
    - × A memória cellái ebben az esetben pikofarad kapacitású kondenzátorokból + 1 tranzisztorból állnak, amik csak akkor engedik hogy az adatként használt töltések “elmenjenek” ha jelet kapnak
    - × A tranzisztorok egy idő után kisülnek, ezért az így tárolt adatokat frissíteni kell hogy megmaradjanak (ne “szökjenek ki” a töltések)
    - × Előnyük hogy olcsók, alacsony a fogyasztásuk és tömeggyárthatók
    - × Egyéb fontos tulajdonságaik: Megbízhatóság, tárkapacitás, sebesség, bővíthetőség

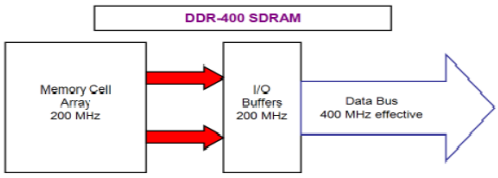
⬇

+ DRAM típusok
  - Aszinkron (klasszikus) DRAM
  - **SDRAM** (szinkron DRAM)
    - × SDR SDRAM (Single Data Rate): Csak az órajel felmenő élén történik adatátvitel
    - × DDR SDRAM (Dual Data Rate)
      - ~ Az órajel mindkét ágán történik adatátvitel => Vagy közel kétszeres sávszélesség vagy az SDR-hez képest kisebb frekvencia ugyanolyan teljesítménnyel
      - ~ A kisebb frekvencia azt is jelenti hogy egy sérült jel helyes kiolvasásra több időnk van

⬇

+ Mi az SDRAM?
  - A rendszersínnel van szinkronizálva, így a válasz mindig órajelre történik
    × A memóriacellákból az adatkiolvasás lassú, nem tudna órajelre működni
    × Egy I/O buffer-be betöltjük az adatot, ami gyorsabb mint a memória cella, elég gyors ahhoz, hogy az órajel ütemének megfelelően küldje tovább az adatot a gyors adatbuszra
  - Az adattároló logikai egységekre (logikai bankokra) van felosztva
    × Ennek az az előnye hogy a memóriavezérlő egyidejűleg több memória hozzáférési parancsot is végrehajthat
  - A logikai bakok futószalag elvnek megfelelően vannak elcsúsztatva, ezért lesz gyorsabb a szinkron RAM az aszinkronnál
    × A futószalag elv azt jelenti, hogy a kért adat az olvasási parancs kiadása után csak egy fix számú óraciklus után jelenik meg
      ~ Ez az óraciklus mennyiség a késleltetés

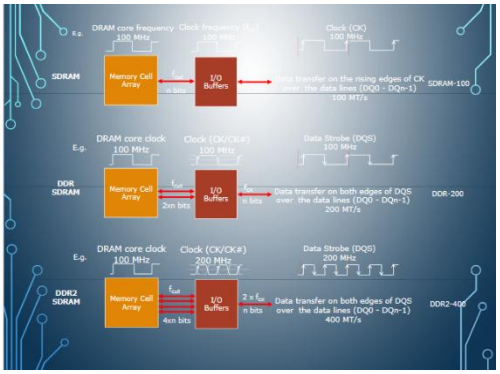
+ DDR SDRAM evolúciója



  - DDR1 (az DDR tényezőinél említettek)
    × Mivel egy órajel alatt kétszer több adatot tudunk elküldeni ezért meg kell növelnünk a puffer és a memóriacella közti buszt
    × **Ezt az eljárást, hogy egyszerre x2 annyi adatot töltünk be a töltünk be a pufferbe, mint az SDR memória "2N-prefetch"-nek nevezik**
  - DDR2
    × Alacsonyabb órafrekvencia-meghajtást igényel

⬇

      ~ Magasabb frekvencia => Több adat ugyanannyi idő alatt
      ~ Ugyanazon frekvencia => Nagyobb késleltetés => Pontosabb
    × 4 sáv => x4 adat (pl. bit)/órajel (4N-prefetch)
      ~ Az adat 4 sávból értekezését 1 sávos továbbküldésre alakítani *(lásd DDR2-es kép)* is növeli a késleltetést
  - DDR3
    × Ugyanaz a fejlődési elv mint a DDR2 esetében (Kisebb frekvencia igény)
    × 8 sáv => 8N-prefetch)
  - **DDR4**
    × Ugyanaz a fejlődési elv mint a DDR3 esetén
    × Ugyanúgy 8N-prefetch
    × Logikai bankok csoportosítása



Csak érdekesség: SDRAM Vs DDR1 Vs DDR2

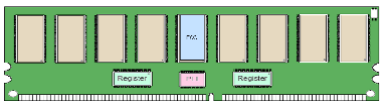
        - ~ DDR3 esetén 8 különálló logikai bank, DDR 4 esetén 16 logikai bank egy 4 bank/csoport módon. A csoportok között sokkal gyorsabban tud váltani a memória vezérlő mint egy csoporton belüli bankok között
    - × **Megbízhatóság javulása**
        - ~ CRC, ami érzékeli a véletlenszerű változásokat
        - ~ Chipenkénti lezárás és feszültség szabályozása
        - ~ Gear Down Mode: Ha szükséges, akkor a prefetch értéke csökkenthető
        - ~ Jobb fizikai kialakítás: 288 DIMM pin
- **DDR5**
    - × Ugyanaz a fejlődési elv mint a többi DDR esetében
    - × 16N-prefetch eljárás
    - × A logikai bank csoportok 8 bank/csoport-ra nőnek és 16 bank helyett most már 32 bank van

+ Adattárolás módja és az adatkiolvasás működése
    - Egy memória rengeteg kis cellából (statikus vagy dinamikus) áll ami 1-1 bit adatot tartalmaz
    - Ezeket sorokba és oszlopokba rendezve “táblázatokat” (más szóval mátrixokat vagy adat-síkokat) kapunk
    - Ezeket “táblázatokat” logikai bankokba rendezzük
    - Egy teljes adatkiolvasási ciklus
        - × Kívánt logikai bank kiválasztása és megnyitása
        - × A kívánt bank (“táblázat”) adott sorának majd oszlopának (vagy oszlopainak) kiválasztása
        - × Az adatok kiolvasása után a logikai bank lezárása (precharge)
        - × Várakozás mielőtt a ugyanaz a bank újra megnyitható (tRP-nyi idő)

⬇

+ SDRAM időzítési paraméterek
    - tRAS *(time of RowAddressStrobe)*: Minimum mennyi időnek kell eltelnie a bank megnyitása és lezárása között
    - tRCD *(time of RAS to CAS Delay)*: Külön kell elküldeni a kívánt sornak és az oszlop(ok)nak a címét. Egy bank és a sor betöltése időt igényel, addig várakozni kell. Ez a minimum várakozási idő a RAS (sorbeolvasás) to CAS (oszlopbeolvasás) delay
    - tCL *(time of ColumnAddressStrobe Latencey)*: Az oszlopkiolvasás parancsának kiadásától az (első, ha több oszlopot olvasunk ki) megjelenő adatig eltelt várakozási idő
        - × Ha a lekért adatok nem egymás mellet helyezkednek el (pl. 1-5 aztán 7, 9, 21. oszlop) akkor a címzést többször is meg kell csinálni ergo többször is ki kell várni a CL idejét

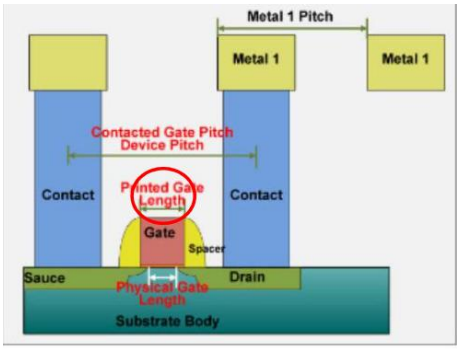
- tRP *(time of Row Precharge)*: Az adatok lekérése után a használt bankot le kell zárni, ami szintén időt igényel és addig nem lehet új bankot megnyitni. Ez a várakozási idő az RP
- tRC *(time of Row Cycle)*: Ha egy bankból több sort is le akarunk kérni akkor soronként meg kell nyitni a bankot, kiválasztani egy sort, majd lezárni és várni egy kicsit mire az adott bankot újra meg tudjuk nyitni



+ **Mik azok a DIMM-ek? (Dual In-line Memory Module)**
  - A DIMM-ek a DRAM chipeket tartalmazó memória modulok (konyhanyelven: egy RAM stick)
  - DIMM-ek típusai
    × Registered DIMM
      ~ A DRAM chipek és a memóriasín közé egy regiszter van elhelyezve
      ~ Ez a regiszter egy órajelciklus erejéig puffereli a különböző jeleket, ezzel könnyítve a memóriavezérlő elektromos terhelését, így stabilabban fog működni
    × ECC DIMM (Error Control Coding DIMM)
      ~ Van +1 DRAM chip a modulon a paritás bit (más néven ECC) tárolására
      ~ Képes egyetlen bitnyi hiba észlelésére
      ~ Neki köszönhetően a memória-vezérlő képes az egyszerre 1 bites hibákat észlelni és javítani
      ~ Egy DIMM lehet egyszerre registered és ECC
  - Rendelkeznek egy PLL-el (Phase Locked Loop)
      ~ Fáziszárt hurok: Feladata az órajel elcsúszások mentesítése

/ 8. Modul : Tranzisztortechnológiák /

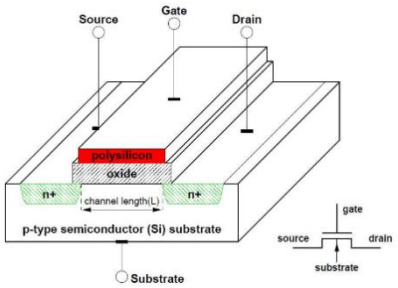
+ Mi a tranzisztor?
  - Számítógépek esetén egy kapcsolóként használjuk
    × Van egy soruce (forrás) és drain (nyelő?) része
    × A töltések a source felől a drain tud áramlani
    × A rész között található egy gate (kapu), ami csak akkor engedi át a töltéseket a source-ból a drain-be, ha ő maga kap egy bizonyos mennyiségű bemeneti feszültséget



  - Legfontosabb méretek a tranzisztoroknál
    × Pitch: A tranzisztor teljes szélessége
      ~ Minél kevesebb, annál több tranzisztor pakolható egymás mellé
    × Gate length: Két típusa van
      ~ Az elméleti hossz: A soruce és drain csatornát elválasztó kapu hossza. Ez viszont kicsit bele szokott lógni mindkét csatornába
      ~ A gyakorlati hossz: A source és a drain közötti tényleges távolság, a kapu extra “belelógása” nélkül

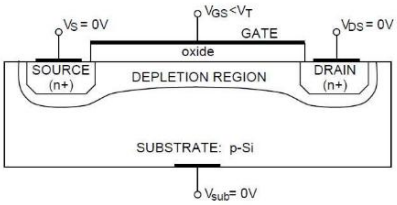
+ Tranzisztorok típusai
  - MOSFET (hagyományos) tranzisztor
  - Feszített szilíciumos tranzisztor
  - HKMG
  - FinFET
  - GAAFET

+ Mi a MOSFET tranzisztor? (Metal-Oxide-Semiconductor Field-Effect Transistor) *(pontosabban egy nMOS tranzisztor)*
  - Tranzisztor részei:
    × Kapu (Metal): A MOS-ban az “M”
    × Szigetelő oxid réteg: Elszigeteli a kaput a lapka többi részétől. Ez a MOS-ban az “O”
    × Body (vagy Substrate): A szilícium lapka, amin tranzisztor többi eleme van. Ez a MOS-ban az “S”



    × Source: Ahonnan az elektronok jönnének
    × Drain: Ide áramolnának az elektronok a source-ból

- Működése:
  - × A Source és a Drain olyan mint két, extra negatív elektronokkal feldoppingolt “sziget” a Substrate “tengerében”
  - × A Substrate-ban nagyon kevés elektron van => Nem tud egy “átjáró” a Source-ból a Drain-be való töltéseknek kialakulni



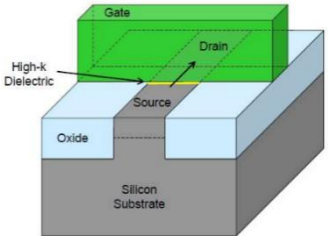
  - × Ha a Kapura elég feszültséget vezetünk, az egy elektromos teret hoz létre, ami elkezdi a kapu “alá” vonzani a Substrate-ban lévő elektronokat és a Source-ban lévő elektronokat és összegyűlik annyi töltés, hogy ki tud alakulni egy kis “híd”, amin keresztül a Soruce-ból a Drain-be tudnak áramolni az elektronok (ez a FET része a MOSFET-nek)
  - × A Kapu és a Substrate közötti oxid réteg azt akadályozza meg hogy a Gate-ből töltések kerüljenek az újonnan kialakult “hídba” (a kapu csak az elektromos mezőhöz szükséges feszültséget biztosítja, nem a töltéseket!)
  - × A kapu + szigetelő + substrate (a MOS) együttesen igazából egy kondenzátort alkot, aminek itt nem a töltések tárolása a feladata csak az elektromos tér (a FET folyamat) kialakítása
    - ~ **A kapacitás egyenlete:** $C = \frac{\kappa * \varepsilon_0 * A}{t}$
      - Δ A: Kapacitás felülete
      - Δ κ (kappa): Anyag dielektrik állandója
      - Δ ε_0 (epszilon nulla): Vákuum permittivitása, állandó
      - Δ t: szigetelő vastagsága

+ Mi a feszített szilícium technológia?
  - A szilícium megfelelő módszerekkel szétnyújtható, így szilícium atomok között több szabad hely lesz amin keresztül az elektronok áramolhatnak

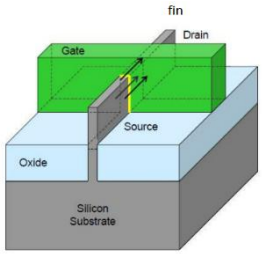
  - Picit nagyobb teljesítményt *(+10-12%)* eredményez, de picit drágább is *(+2%)*

+ Mi a HKMG tranzisztor? (High-k Metal Gate)
  - Probléma: A tranzisztorok mérete egyre csökkent, ezzel együtt a szigetelő oxid réteg vastagsága is csökkent
    - × Ez szivárgásokhoz vezetett (olyankor is áramoltak elektronok mikor még a tranzisztornak papíron zárva kellett volna lennie)
  - Megoldás: A vékony szigetelő oxid réteget lecserélték egy high-k dielektrik-re, ami sokkal vastagabb lehetett, ezzel...
    - × Csökkent a szivárgás
    - × Kevesebb áram kellett a tranzisztor kapcsolásához
    - × Gyorsabban lehetett a tranzisztort kapcsolni

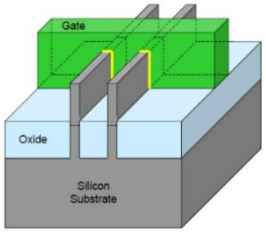
+ Mi a FinFET tranzisztor? (avagy 3Ds vagy tri-gate tranzisztor)
  - Azt megelőző planáris (2Ds) tranzisztoroknál a Source és a Drain a Kapu alatt alakított ki egy áramlási “hidat”
  - A FinFET tranzisztor esetén a Substrate réteg kiemelkedik és “belehatol” a kapuba és 3 irányból veszi közre a kaput. Ha a kapu “engedélyt” ad az elektronok áramlására, akkor a “híd” egy sokkal nagyobb felületen tud kialakulni
    × Több kiemelést (fésűfogat) is ki lehet alakítani, hogy egyszerre több elektront is át tudjunk engedni
      ~ => nagyobb teljesítmény (gyorsabb ki-be kapcsolás) ugyanazon a feszültségen
      ~ => ugyanolyan teljesítmény kisebb feszültségen
    × Ezt tovább lehet javítani, ha hosszabb és vékonyabb fésűfogakat alakítunk ki
      ~ Nagyobb teljesítmény
      ~ Kisebb szivárgás
      ~ Kisebb helyigény
  - Mi a GAAFET tranzisztor? (Gate All Around)
    × 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)**
    × Ezeknek a lapoknak/szalagoknak a szélességét ha növeljük, akkor **nanowire helyett nanosheet lesz**, és megint nagyobb áramerősséget tudunk rajtuk átvezetni **(ez az MCBFET)**

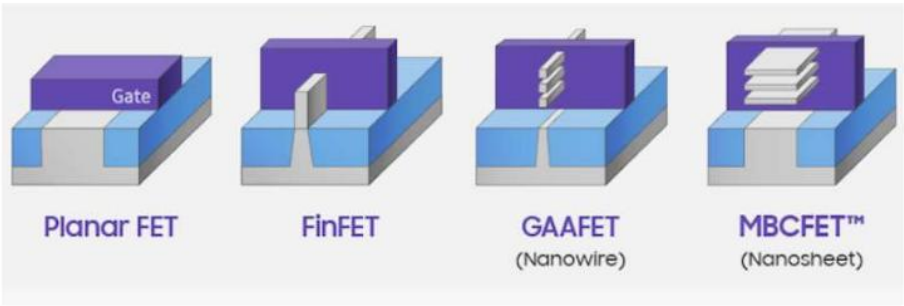


2Ds tranzisztor



Fin (fésűfogas) tranzisztor





/ 9. Modul : Párhuzamos architektúrák /

+ A párhuzamosság típusai
  - Funkció szerint:
    × Rendelkezésre álló párhuzamosság: A feladatban/programban rejlő párhuzamossági lehetőség (elméletileg mennyire lehetne párhuzamosítani)
    × Kihasználható párhuzamosság: A rendelkezésre álló párhuzamosság azon része, amit ki is lehet használni (gyakorlatban mit lehet megvalósítani az elméletből lehetőségek tárházából)
  - Elhelyezkedés szerint:
    × Időbeli párhuzamosság: A kívánt feladatot különböző szakaszokra bontjuk fel, minden szakasszal 1-1 végrehajtó egység foglalkozik. Ha egy feladat túl van egy adott szakaszon, akkor mehet tovább a következőre és a helyére érkezhet egy másik feladat. Így több feladattal is tudunk egyszerre foglalkozni, úgy hogy csak időben eltoljuk őket

Persze! Képzelj el egy **modern autómosó sort** 🚗🚿.

Ez tökéletes analógia az **időbeli párhuzamosságra** (vagyis a futószalag-feldolgozásra).

1. A régi módszer (Párhuzamosság NÉLKÜL)

Képzelj el egy egybeállásos garázst, ahol egyetlen munkás mindent megcsinál:

1. Beáll az **"A" autó**.
2. A munkás **lemossa** (5 perc).
3. A munkás **megtörli** (5 perc).
4. A munkás **kiviaszolja** (5 perc).
5. Az **"A" autó** kimegy. (Összesen: 15 perc)
6. *És csak most* jöhet be a **"B" autó**...

Ha 3 autót akarsz lemosni, az **45 percig** tart (3 x 15 perc). Egyszerre csak egy dolog történik.

Egy példa az időbeli párhuzamosságra a Gemini által

2. Az új módszer (IDŐBELI Párhuzamossággal)

Ez a futószalagos autómosó. Itt 3 külön állomás van, 3 külön munkással:

- **1. Állomás:** Csak mosás 🧼
- **2. Állomás:** Csak törlés 🧻
- **3. Állomás:** Csak viaszolás ✨

Most nézzük, mi történik, ahogy jönnek az autók (percenként léptetjük az állásokat):

T1 (Start):

- **"A" autó** bemegy az 1. Állomásra (Mosás).

T2 (5 perc múlva):

- **"A" autó** átmegy a 2. Állomásra (Törlés).
- **"B" autó** *máris* bemehet az 1. Állomásra (Mosás).

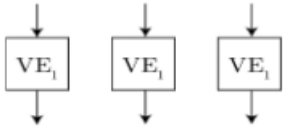
T3 (10 perc múlva):

- **"A" autó** átmegy a 3. Állomásra (Viaszolás).
- **"B" autó** átmegy a 2. Állomásra (Törlés).
- **"C" autó** *máris* bemehet az 1. Állomásra (Mosás).

T4 (15 perc múlva):

- **"A" autó** ELKÉSZÜLT! 🎉
- **"B" autó** átmegy a 3. Állomásra (Viaszolás).
- **"C" autó** átmegy a 2. Állomásra (Törlés).
- **"D" autó** bemehet az 1. Állomásra (Mosás).

    × Térbeli párhuzamosság: Több, egyforma típusú végrehajtó egységünk van, amik kapnak 1-1 feladatot, amin egyidőben egyszerre dolgoznak
      ~ Az előző példa alapján olyan mintha lenne 3 külön autómosó sorunk



- Típus szerint
  - × Adatpárhuzamosság: Két megvalósítása van
    - ~ Az adatokon vagy térbeli vagy “futószalagos” elven tudunk feladatokat végezni
    - ~ Át tudjuk alakítani funkcionális párhuzamossággá: Az adott külön-külön formában lévő műveleteket meg tudjuk adni mint kevesebb darab ciklus
  - × Funkcionális párhuzamosság:
    - ~ A feladat logikájából következő párhuzamosság
    - ~ Szintjei:
      - Δ Utasítás szintű
      - Δ Ciklus szintű
      - Δ Eljárás szintű
      - Δ Program szintű
      - Δ Felhasználó szintű

+ A számítások felgyorsítása érdekében az architektúrák, operációs rendszerek és a fordító programok (Compiler-ek) egyaránt törekednek

+ Rendelkezésre álló párhuzamosságok hasznosítása
  - Ha utasítás szintű: Párhuzamos architektúra vagy megfelelő fordítóprogram segítségével
  - Ha ciklus szintű vagy eljárás szintű: Szálak vagy folyamatok használatával (ezek a legkisebb önállóan végrehajtható kódrészletek), amiket vagy a programozó vagy az OS vagy egy Compiler tud létrehozni
  - Ha program szintű vagy felhasználói szintű: Megfelelő programok vagy párhuzamos rendszerek segítségével lehetséges, ha támogatja a hardver és szoftver is
+ Párhuzamossági szintek (más néven: szemcsézettség)
  - Alacsony szintű párhuzamosság (vagy “finom szemcsézettség”): Párhuzamos architektúrák vagy párhuzamos Compilerek segítségével közvetlenül kihasználhatók
    - × Utasítás szint
    - × Szál szint
    - × Folyamat szint
  - Magas szintű párhuzamosság (vagy “durva szemcsézettség”): Többszálas vagy többfolyamatos OS kell hozzá
    - × Felhasználó szint

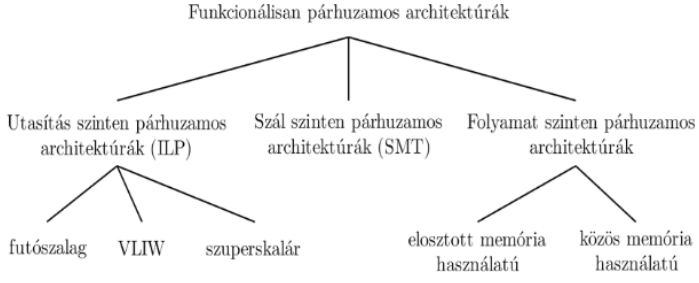
+ Mi az a compiler?
  - Fordítóprogram, ami a magasabb szinten írt kódot lefordítja a processzor által érthető kódra
  - Feladatai:
    × Elemzi a magas nyelven megírt kódot lexikálisan, szintaktikailag, semantikailag
    × **Tárgykódot szintetizál (készít)**
      ~ Kódot generál
      ~ Kódot optimalizál: Ha párhuzamosítani akarunk akkor olyan független részeket keres a kódban, amiket végre lehet külön-külön hajtani egyidőben bármilyen előfeltétel nélkül

+ Párhuzamos architektúrák osztályozása Flynn-féle módon: A kategóriák a vezérlőegységek és a feldolgozási egységek számától függenek
  - SI (Single Instruction stream): A vezérlőegység egyszerre egy utasításfolyamot tud végrehajtani és generálni is
  - MI (Multiple Instruction stream): A vezérlő egyidőben több, egymástól elkülönülő utasításfolyamot tudunk végrehajtani és generálni
  - SD (Single Data stream): Egyetlen CPU egyszerre egyetlen adatfolyamot dolgoz fel
  - MD (Multiple Data stream): Több végrehajtó egység dolgoz fel több, egymástól független adatot

  - Ezek alapján az architektúrák 4 kategóriába sorolhatók:

SISD	SIMD	MISD	MIMD
Neumann modell (szekvenciális végrehajtás)	Multimédia feldolgozás (ugyanazon műveletek végrehajtása sok adaton)	Elméleti kategória	Teljes párhuzamos feldolgozás

+ A modern osztályozás már figyelembe veszi a párhuzamosság fajtáját, szintjét és módját is és két fő csoportot hoz különböztet meg: Adatpárhuzamos architektúra vagy funkcionálisan párhuzamos architektúra



+ Egy utasítás végrehajtásának lépései (elemi műveletei):
  - F (fetch): Utasítás lehívása
  - D (decode): Utasítás dekódolása
    × S/O (source operand): Forrás operandusok („bemeneti paraméterek”) lehívás
  - E (execute): Utasítás végrehajtása (pl. összeadás)
  - W/B (writeback): Eredmény visszaírása
+ Mi a kibocsátás és a kibocsátási párhuzamosság?:
  - A kibocsátás az a folyamat, amiben a CPU dekódoló egysége megkapja az utasítást, dekódolja majd azt továbbküldi a végrehajtó egységnek
  - Ha a CPU képes több utasítás végrehajtására akkor fontos hogy egyszerre több utasítást is ki tudjunk bocsátani
  - A kibocsátási párhuzamosság azt jelenti hogy a dekódoló egy órajel alatt több utasítást is ki tud bocsátani
+ **Párhuzamos feldolgozás ILP CPU-ra vonatkozó követelményei**
  - Minden ILP CPU-nak figyelembe kell venni az utasítások között fellépő függőségeket
  - Meg kell őrizni a soros végrehajtás konzisztenciáját
    × A programozó soros végrehajtású kódot ír és így is gondolkozik
    × Amit így megír a programozó, annak ugyanúgy működnie kell párhuzamosan is

+ A függőségek
  - Az egymást követő utasítások függhetnek egymástól egy programban (pl. két számot összeadunk és utána szorozzuk tízzel. A szorzás előtt kell az összeg)
  - Emiatt (és a feladatrészek közti végrehajtási időkülönbségek miatt) nem lehet csak szimplán növelni egy futószalag teljesítményét azzal hogy több részre bontunk egy feladatot
+ Függőségek típusai:
  - Adatfüggőség
  - Vezérlésfüggőség
  - Erőforrásfüggőség (sok feladat, kevés egység hozzá -> több erőforrás kell)

+ Függőségek > Adatfüggőség
  - Egymást követő utasítások ugyanazokat az adatokat használják
  - Típusai:
    × Jellege szerint
      ~ **Valós adatfüggőség**
        Δ **Műveleti adatfüggőség**
        Δ **Behívási adatfüggőség**
      ~ **Ál adatfüggőség**
        Δ **Write After Read (WAR)**
        Δ **Write After Write (WAW)**
      ~ Ciklusban jelenlévő

    - × Operandus típus szerint
        - ~ Regiszter
        - ~ Memória

+ Függőségek > Adatfüggőség > Valós adatfüggőség

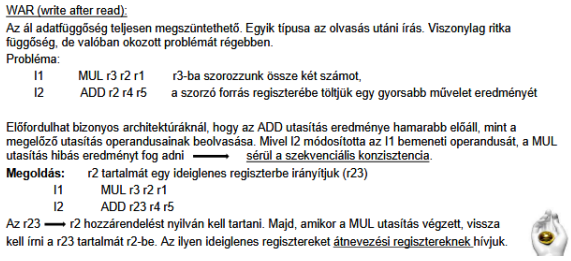
	t_1	t_2	t_3	t_4	t_5
I_1	F_{MUL}	D/ $SO_{r1, r2}$	E_{MUL}	W/B_{r3}	
I_2		F_{SHL}	D/ SO_{r3}	E_{SHL}	W/B_{r3}

A példában említett számolás időbeni lebontása szálanként

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

    - Műveleti adatfüggőség
        - × példa: Két számot akarunk összeszorozni majd az eredményt megduplázzuk
            - ~ t1-ben beolvassuk az utasítást, t2-ben még csak dekódoljuk az utasítást és lekérjük a változókat, t3-ban még csak a szorzást végezzük el, de a második (I2) szál már kérné a még nem létező r3 eredményt
            - ~ Megoldás: Várakozni kell (2. ábra) => csökken a hatékonyság!
        - × A jobb megoldás: Operandus előhozás
            - ~ Egy extra hardver segítségével az eredményt azonnal visszaírjuk és be is töltjük egy forrásregiszterbe, amit a második szál tud használni
    - Lehívási adatfüggőség
        - × A szükséges operandusokat betöltjük az ALU regiszterébe alapesetben
        - × Probléma: Ha egy adat éppen nem elérhető akkor azt “később” kell betölteni vagy a cacheből vagy az operatív tárból, ami sokkal több idő mint egy regiszterből betölteni
        - × Megoldás: Az operandusokat nem csak a regiszterekbe töltjük be hanem extra hardverekkel az ALU bemenetére is ezzel kicsit gyorsítva a végrehajtást ilyen problémás esetben

+ Függőségek > Adatfüggőség > Ál adatfüggőség (azért „ál”, mert teljesen megszüntethető)
    - A probléma az álfüggőségek esetén: Sok művelet, de kevés regiszter. Meg kell oldani az ilyen felülírási hibákat
    - Write After Read (WAR)
        - × Egy regisztert két műveletben is használunk: egyikben mint bemeneti (forrás) regiszter, másikban mint kimeneti (eredmény) regiszter
        - × Ha a második művelet gyorsabb, az át fogja írni az első művelet előtt az egyik bemeneti regisztert és az első művelet hibás lesz
        - × Megoldás: A második művelet tartalmát egy Átnevezési (más néven Piszkozat) regiszterbe tesszük, amiből az első, lassabb művelet végrehajtása után beleírhatjuk a második, gyorsabb művelet eredményét a kívánt regiszterbe
            - ~ A piszkozati regiszterek önálló regiszterek, saját regisztertérrel, amit csak a vezérlés lát és használ és extra hardvernek számítanak


WAR (write after read):
Az ál adatfüggőség teljesen megszüntethető. Egyik típusa az olvasás utáni írás. Viszonylag ritka függőség, de valóban okozott problémát régebben.
Probléma:
I1 MUL r3 r2 r1 r3-ba szorozzunk össze két számot,
I2 ADD r2 r4 r5 a szorzó forrás regiszterébe töltjük egy gyorsabb művelet eredményét
Előfordulhat bizonyos architektúráknál, hogy az ADD utasítás eredménye hamarabb előáll, mint a megelőző utasítás operandusainak beolvasása. Mivel I2 módosította az I1 bemeneti operandusát, a MUL utasítás hibás eredményt fog adni ⟶ sérül a szekvenciális konzisztencia.
Megoldás: r2 tartalmát egy ideiglenes regiszterbe irányítjuk (r23)
I1 MUL r3 r2 r1
I2 ADD r23 r4 r5
Az r23 ⟶ r2 hozzárendelést nyilván kell tartani. Majd, amikor a MUL utasítás végzett, vissza kell írni a r23 tartalmát r2-be. Az ilyen ideiglenes regisztereket átnevezési regisztereknek hívjuk.

- Write After Write (WAW)
  - × Két műveletünk van: az első lassabb, a második gyorsabb
  - × Probléma: Ha ugyanabba a regiszterbe töltjük bele mindkét művelet eredményét, akkor a lassabb művelet felül fogja írni a gyorsabb művelet eredményét
  - × Megoldás: Átnevezési (avagy Piszkozat) regiszterekbe kiírni a gyorsabb művelet eredményét

Probléma:

I_1 MUL r_3 r_2 r_1 r_3-ba szorozzunk össze két számot
I_2 ADD r_3 r_4 r_5 majd ugyanabba a regiszterbe töltjük egy gyorsabb művelet eredményét

+ Függőségek > Adatfüggőség > Ciklusbeli adatfüggőség
  - Probléma: Egy ciklus következő eredményéhez szükség van az előző iteráció eredményére
  - Megoldás: Algoritmus áttervezése

+ Függőségek > Vezérlés függőség
  - Feltételes vagy feltétel nélküli elágazásoknál tud előlépni

+ Függőségek > Vezérlés függőség > Feltétel nélküli elágazásnál
  - JMP (ugrás parancs) használatánál történik ilyen
  - Az ugrás parancs túl későn történik: Mire megtörténik az ugrás a kívánt parancsra, addigra a “rossz”/nem kívánt parancs már lefutott (lehet már több is) *(lásd: kép)*
    - × Feleslegesen hívjuk le az utasítást és ha esetleg a hibás utasítás le is fut akkor át tud írni olyan regisztereket amiket nem kellett volna

ágazás:

MUL
JMP címke — PC tartalmát felülírjuk
ADD
.
.
SHL

	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):
    - × Késleltetés: No Operation (NOP) utasítással, ami megint a hatékonyságot csökkenti
    - × Utasítások átrendezése optimalizáció compiler segítségével (dinamikus kezelés)

	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

+ Függőségek > Vezérlés függőség > Feltételes elágazásnál
  - Ma már csak dinamikusan lehet kezelni, hiszen az elágazás továbbmenetelétől függ hogy ugrani kell az utasításokban vagy mehetünk tovább a megadott sorrendben

+ Függőségek > Erőforrás függőségek
  - Akkor lép fel ha több utasítás akarja ugyanazt az erőforrást használni. Ilyenkor egy utasítást kiválasztunk, a többit várakoztatni kell
  - Cél: Úgy 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 szekvenciális (soros) konzisztencia megőrzése**
  - A programozó soros logika alapján írja meg a programot, de a rendszer próbál minél több dolgot párhuzamosítani
  - A compiler feladata a soros utasításokat minél jobban párhuzamosítani úgy hogy a logikai integritás megmaradjon
  - Típusai
    × Utasításfeldolgozás soros konzisztenciája
      ~ Utasítás végrehajtás soros konzisztenciája (más néven: processzor konzisztencia)
      ~ Memória hozzáférés soros konzisztenciája (más néven: memória konzisztencia)
    × Kivételkezelés (megszakítás) soros konzisztenciája
      ~ pontatlan kivételezés (gyenge konzisztencia)
      ~ pontos kivételezés (erős konzisztencia)

+ Soros konzisztencia > Processzor konzisztencia **(itt flag-ekkel dolgozunk nem pedig operandusokkal vagy elágazásokkal!)**
  - Egy utasítás hamarabb lefut, mint amilyen sorban jönne (az ADD gyorsabb mint a DIV)

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

  - A JZ (Jump Zero) utasítás mindig a legutoljára elvégzett megoldást vizsgálja, ha az a DIV és nem az ADD akkor hibás működéshez juthatunk mert a JMP megtörténik amikor nem kéne vagy fordítva.
  - Megoldás: Úgy kell tervezni a hardvert hogy ilyen ne forduljon elő
    × A JZ utasítás az előtte sorban lévő eredményt vizsgálja
    × Plusz flag-ek bevezetése

+ Soros konzisztencia > Kivételkezelés konzisztenciája
  - Párhuzamos végrehajtás esetén a kivételeknek a sorrendje is összekeveredhet (valami előbb küld kivételt mint ahogy sorrend szerint kéne egy gyorsabb végrehajtás miatt), ami szintén problémát okozhat a megszakítás kezelésénél
    × Ez a pontatlan megszakításkezelés

Probléma:
tegyük fel, hogy az alábbi kódrészletben az ADD túlcsordul, de a MUL még nem fejeződött be.

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 túlcsordulása miatti megszakításkérést kezelni kell. Ilyenkor a processzor a regiszterek állapotát (kontextust) elmenti egy verem regiszterbe. Miután a kivétel lekezelődött, a veremből visszatöltődik a kontextus és folytatódik a végrehajtás.
Ebben az esetben viszont nem fogjuk tudni, hogy a MUL utasítás végzett-e már, így az r3 regiszter definiálatlan állapotba kerül ⟶ hibákhoz vezethet!

  - Pontos kivételkezelés: A megszakítás kéréseket csak az utasítások sorrendjében lehet elfogadni
    × Megvalósítás: Átrendező pufferek
    × Címkézés: Az utasításokat sorszámokkal látjuk és csak akkor fogadunk el egy megszakításkérést ha a olyan utasítástól jön, aminek megfelelő a sorszáma

/ 10. Modul : Futószalagos processzorok /

Instruction$_1$	**F (t_1)**	**D (t_2)**	**E (t_3)**	**W/B (t_4)**	
Instruction$_2$		F (t_1)	D (t_2)	E (t_3)	W/B (t_4)

ahol t1=t2=t3=t4

Példa: A végrehajtás 4 részre történő osztása

+ Mi az a futószalagos (más néven: pipeline) megoldás?
  - Egy utasítást több részre bontunk és ezeket a részeket külön, egymással párhuzamosan hajtjuk végre
  - Ez az ún. időbeli párhuzamosság
    × *Ha minden rész ugyanannyi időt igényelne akkor ahány részre bontjuk a végrehajtást, annyiszor gyorsabb is lenne*
+ Futószalagos feldolgozás előfeltételei *(egy 2 fokozatú ideális futószalag esetén)*
  - Szükség van 2 db egymástól teljesen független végrehajtó egységre
  - Az egyik fokozat kimenete, a másik fokozat bemenete
  - Mindkét fokozat végrehajtási ideje azonos
    × Órajelre kapják a bemenetet és egy óraciklus alatt el is végzik a feladatot

+ A különböző függőségek gátolják a hatékony futószalagos végrehajtást, ezért ezeket kezelni kell:
  - Operandus előrehozás
  - Újrafeldolgozás (pl. egy szorzás esetén nem sok külön-külön összeadást csinálunk hanem ciklusos módon az előző összeadás kimenetét már be is töltjük mint az új összeadás bemenete és végrehajtjuk ugyanazt a fokozatot)

+ Futószalagok típusai
  - Előlehívásos (avagy: overlapping): Egy utasítás eredményének visszaírása közben már hívjuk is le a következő utasítást

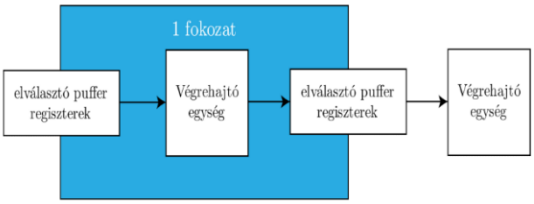
	t_1	t_2	t_3	t_4	t_5	t_6	t_7
I_1	F	D	E	W/B			
I_2				F	D	E	W/B

    × Ezzel nyerünk 1 extra óraciklust per fokozat => Minél több fokozatunk van, ez annál több megspórolt óraciklus
    × Ez a megoldás függőségeket sem okoz, mert mire beolvasnánk a következő utasítás bemeneteit, azok addigra már ki lettek írva (lásd: t_4 és t_5 idő)
  - Vektor CPU
  - Teljes pipeline
  - Logikai futószalagok
    × Az eltérő utasítások eltérő futószalagokat igényelnek, ezért a processzorban több, különböző feladatokra kialakított futószalag is lesz

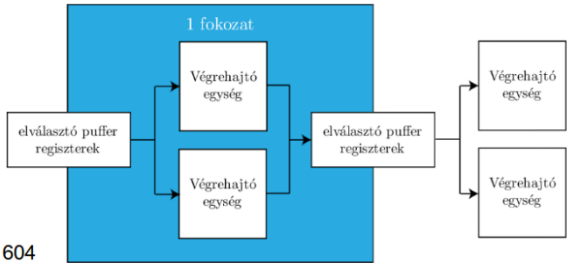
+ A futószalagok fizikai megvalósítása
  - Alkalmazásuk alapján 2 csoportjuk van
    × Univerzális futószalag: Mindenre jók, viszont bonyolultak, drágák és lassúak

× Dedikált futószalag: Adott műveletekre vannak specializálva, amik kevesebb logikai kaput igényelnek mint egy univerzális futószalag, így gyorsabbak is (pl. Aritmetikai, LOAD/STORE)

- A futószalag sebességét mindig a leglassabb fokozat határozza meg, ezért az a cél hogy minden fokozat ugyanolyan gyors legyen
  × Ennek megoldására a fokozatok előtt és után is elválasztó puffer regisztereket helyezünk el
    ~ Az bemenetek és kimenetek ilyen regiszterekbe töltődnek be, és itt tárolódnak, amíg végrehajtó egységek dolgoznak



    ~ Ha egyszerre több végrehajtó egység is van egy fokozatban (ezt nevezik térbeli párhuzamosságnak), akkor vezérlőjelek segítségével van meghatározva hogy melyik regiszterből, melyik végrehajtó egységbe töltődjön az adat



\+ RISC és CISC architektúrák

- Az utasításkészlet alapján, két különböző architektúrát különböztetünk meg:
  × RISC (Reduced Instrutction Set Computing): Csökkentett utasításkészletű (pl. ARM: Advance RISC Machine)
  × CISC (Complex Instrutction Set Computing): Bővített utasításkészletű (pl. x86 *vagy mostmár x86-64*)

+ A RISC architektúra főbb tulajdonságai
  - Kevés utasítással rendelkeznek (kb 50 - 150), ami egyszerűsíti a címzési módokat
  - Az utasítások egyforma hosszúak, ami könnyíti a futószalagos feldolgozást
  - Az utasítások egyszerűek és alapvetőek
    × Nincs olyan hogy egy utasítás pl. egyszerre betölt adatot meg számol meg stb, ez mind egy-egy külön utasítás

  - Emiatt sokkal több utasításra van szükség egyes műveletek elvégzéséhez, ami bonyolítja a fordítóprogramokat

  - Egy utasítás általában 3 operandusból áll, aminek köszönhetően pl. egy számolás esetén nem kell felülírni egyik bemenet regiszterét sem, mert van egy külön kimeneti regiszter
  - Minden utasítás szigorúan csak regisztereket használhat
    × A memória és cache csak LOAD/STORE utasítással érhető el

  - Nagyszámú regiszterkészletet igényel

  - Az utasítások végrehajtása általában csak egy óraciklust igényel

  - Előnyök: Alacsony energiafogyasztás és gyorsabb végrehajtás utasításonként a CISC architektúrához képest
  - Hátrányok:
    × A bonyolultabb feladatokhoz sokkal több utasítást kell írni, ami növeli a programok méretét
    × A kevés utasítás miatt kisebb a kompatibilitása is
    × Kisebb a teljesítménye ugyanazon a frekvencián egy CISC-es processzorhoz képest

+ A CISC architektúra főbb tulajdonságai
  - Sok utasítással rendelkezik (több száz), emiatt sok címzési mód is van és nagy belső mikroprogramtár kell
  - Az utasítások hossza változó, így a dekódolónak nem csak dekódolnia kell, hanem meg kell határozni melyik utasítás meddig tart, amihez extra hardver és idő szükséges
  - Egy utasítás egyszerre több elemi műveletet is végre tud hajtani (pl. egy utasítás egyszerre beolvas adatot, számol, majd visszaírja az eredményt)

    ⬇

  - Ezek az összetett utasítások levesznek a fordítóprogramról bizonyos „tervezési terheket”, így azok sokkal egyszerűbbek

  - Közvetlenül el lehet érni a memóriát és a cache-t is
  - Általában 2 operandusos utasítások vannak használva, így az eredménynek felül kell írnia az egyik bemeneti regisztert
    × Emiatt az egyik bemeneti operandusnak muszáj regiszternek lennie mert a memóriába való kiírás nagyon lassú lenne
  - Egy utasítás végrehajtása több óraciklust is igényelhet
  - Nagy kompatibilitás, mivel ha új utasítások kellenek, akkor csak bővítjük a régi készletet, így a régebbi utasításokat igénylő programok is működőképesek maradnak
  - Támogatja a többszálúságot és a virtualizációt

  - Előnyök: Egyszerűbb fordítóprogramok, nagyobb kompatibilitás
  - Hátrányok: Komplexebb hardver, lassabb végrehajtás per utasítás, magasabb energiafogyasztás

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!

Példa: Ugyanaz az utasítás CISC és RISC architektúrán

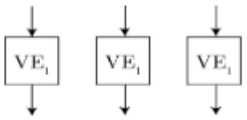
  - a

+ Hibrid architektúrák
  - *“Megfigyelték, hogy egy CISC CPU a működése során a rendelkezésre álló utasításoknak ~ 20%-át használja az idő ~80%-ában -> Célszerű ezeket gyorsítani -> RISC mag”*

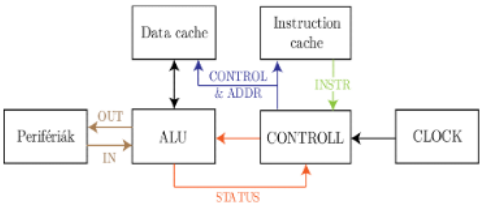
/ 11. Modul : Szuperskalár architektúrák /

+ Mi a szuperskalár architektúra?
  - Előzmények: A dekódolók csak 1 utasítást tudtak kibocsátani óraciklusonként. Ennél nem lehetett gyorsabban „dolgozni”
  - Megoldás: Időbeli és térbeli párhuzamosság a szuperskalár architektúrával, ahol 1 óraciklus alatt több utasítást is ki tud bocsátani a dekódoló
  - Ehhez már egy új architektúra is kellett
  - 3 generációra osztható

+ Szuperskalár architektúrák közös jellemzői:
  - A dekódoló 1 óraciklus alatt több utasítást is kibocsát (ez a kibocsátási párhuzamosság)
  - Időbeli és térbeli párhuzamosság => Több futószalag dolgozik, párhuzamosan
  - Maguk küzdenek meg a függőségekkel extra hardverek segítségével
  - Kompatibilitás a régi, 1 futószalagot használó programokkal



+ Mi az a Harvard architektúra?
  - Lényege: Az adat és a programkód elkülönített útvonalon mozog
  - => párhuzamos adatutak, ahol az adat és az utasítás párhuzamosan tud haladni
  - Előnyei:
    × Lehet párhuzamosan adatot és utasítást írni vagy olvasni cache nélkül is
      ~ A Neumann architektúra esetén egy óraciklus alatt vagy utasítást olvastunk be vagy adatot írtunk/olvastunk mert ezeknek egy közös busza volt
    × A külön tárolt utasítások és adatok külön címtartománnyal rendelkeznek és így a hosszuk lehet eltérő



+ Első generációs szuperskalárok (más néven: keskeny szuperskalárok)
  - 2-3 kibocsátott utasítás/óraciklus
  - Közvetlen (nem pufferelt) kibocsátás: A dekódolt utasítást közvetlenül küldjük a végrehajtó egységhez
  - Statikus elágazásbecslés
    × Ha egy program egy elágazáshoz ér (legyen az „for” ciklus vagy „if” elágazás), döntenie kell hogy visszaugrik-e egy előző utasításra mert az egy ciklus, vagy halad tovább sorban az utasításokkal
      ~ Azért kell dönteni, 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

    - × Ha a program hibásan dönt és elkezd felesleges adatokat számolni, akkor az összes hibás adatot el kéne vetni és megvárni az eredményt, ami időpazarlásnak számítana
    - × Statikus becslés esetén egy elágazásnál vagy mindig ugrunk vagy mindig haladunk tovább
  - Kétszintű gyorsítótár
  - L1 cache-en külön van az adat és az utasítás a Harvard architektúra alapján
  - L2 cache-ben és az operatív tárban marad a Neumann architektúra alapján (közös az adat és az utasítás tárolója)
  - Alapvető problémái:
    - × Nem tudták megfelelően kezelni a különböző függőségeket
    - × A közvetlen kibocsátás: Sorrendben történt az utasítások kibocsátása és ha nem sikerült megfelelően kezelni egy függőséget, akkor várni kellett, ezzel lecsökkentve a kibocsátási rátát akár 1 utasítás/ciklusra
    - × A cache megjelenésével csökkent a memória mérete
    - × Röviden: Rengeteg extra alkatrész és munka a semmiért és ennek “bottleneck”-je a dekódoló volt…

+ Második generációs szuperskalárok
  - 3-4 utasítás/óraciklus
  - Második generációs szuperskalár CPU-k kellettek hozzá
    - × Dinamikus utasítás ütemezés
    - × **Képes regiszter átnevezésre (ez és a dinamikus ütemezés kiküszöböli a függőségi “bottleneck”-et)**
    - × Elágazások előrejelzése dinamikus előrejelzéssel (90-95% pontosságú)
    - × Kifinomult és kibővített cache alrendszer
    - × **Dinamikus Sorrenden kívüli kiküldés**
    - × **RISC mag**

+ Második generációs szuperskalár CPU-k tulajdonságai részletezve
  - Dinamikus ütemezés
    - × A kibocsátás a (dekódolóból a pufferbe küldés) sorrendi módon történik, de a kiküldés (pufferből a végrehajtó egységbe) viszont sorrenden kívül, a stréber modell alapján
    - × Aminek nincs függősége, azt kiküldjük
  - Regiszter átnevezés
    - × Probléma: Vannak ún. ál adatfüggőségek, amik lassítják a végrehajtást (ál adatfüggőségre példa a kép alapján: Az R1 regisztert az egyik utasításban bemenetként használjuk, a másikban meg mivel kimenetnek használjuk, ezért felülírjuk. Ezt a két feladatot nem lehet emiatt párhuzamosan elvégezni

```
1. ADD R1, R2, R3    ; R1 = R2 + R3
2. SUB R4, R1, R5    ; R4 = R1 - R5  (Igazi adatfüggőség: várni kell az 1-esre)
3. MUL R1, R6, R7    ; R1 = R6 * R7  (Névfüggőség: felülírja R1-et)
4. ADD R8, R1, R9    ; R8 = R1 + R9
```

    - × Megoldás: Regiszter átnevezés, ami kiküszöböli az ál adatfüggőségeket

        - ~ Vannak architektúrális regisztereink (pl. R1, R2, R3), amiket a számításokban használunk
        - ~ Van egy adag extra regiszter, amiket hozzárendelünk az architektúrális regiszterekhez, mint piszkozati regiszter
        - ~ Ha egy művelet az R1 regisztert akarja használni akkor azt az egyik utasítás esetén helyettesítjük egy piszkozat regiszterrel (pl. P10) és ha a másik utasításnak is kéne az R1, akkor pedig azt R1 helyére behelyettesítünk egy másik piszkozat regisztert (pl. P11)
    - Dinamikus elágazásbecslés
        - × Probléma: Ha egy program egy elágazáshoz ér (legyen az „for” ciklus vagy „if” elágazás), döntenie kell hogy visszaugrik-e egy előző utasításra mert az egy ciklus, vagy halad tovább sorban az utasításokkal. Statikus esetben vagy mindig ugrik vagy mindig megy tovább
        - × Megoldás: Az elágazások történetét történetbitek tárolják el
        - × Az alapján hogy az előző elágazás(ok)nál ugrottunk-e vagy sem, a program eldönti hogy ebben az esetben mit csináljon: visszaugorjon egy előző utasításhoz, mert az egy ciklus vagy menjen tovább sorban az utasításokkal
    - Sorrenden kívüli kiküldés (Out of Order)
        - × Kibocsátáskor (dekóderből a pufferbe küldés) közben nincs függőségvizsgálat, ezért a dekódoló nem fogja lelassítani a rendszert, mint az első generációs szuperskalár architektúra esetén
        - × Helyette, minden utasítás rendelkezik egy állapotbittel
        - × A vezérlőegység az állapotbit segítségével eldönti melyik utasítás függő és melyik független és amelyik tudja, azt küldi tovább a vezérlőegységeknek sorrendtől függetlenül (innen jön az „Out of Order” rész)

    - Összefoglaló:
        - × Az új generáció kezelt az erőforrás függőségeket azáltal, hogy több végrehajtó egysége volt
        - × Kezelte az áladatfüggőségeket
        - × A vezérlési függőségek súlyossága gyengült a dinamikus elágazáskezeléssel
        - × A valós adatfüggőségek továbbra is lassították a végrehajtást

+ Harmadik generációs szuperskalárok
    - A második generáció elérte az architektúra korlátjait az utasítás szintű párhuzamosítás szinten
    - Újítás: Utasításon belüli párhuzamosság
    - Utasításon belüli párhuzamosság típusai:
        - × Duál műveleti utasítás (egy utasítással több művelet is elvégezhető egyszerre), nem nagyon használt
            - ~ *Nem összekeverni egy CISC paranccsal. Egy CISC parancs csak sok elemi műveletet gyűjt egy parancsszó alá, amit egymás után végzünk el, míg ebben az esetben mondjuk egy összeadás és szorzás egyszerre történik, különböző végrehajtó egységek által*

- × **SIMD (Single Instruction Multiple Data) utasítások**: Egy utasításon belül több operanduson ugyanazt a műveletet végezzük el (pl. van 4 szám mint bemenet és mindenkit összeadunk mindenkivel párosával)
- × VLIW utasítások (leálltak a fejlesztésével)

- SIMD architektúra jellemzői:
  - × Egy kibővített utasításkészletet alkottak (multimédia)
    - ~ Emiatt ki kellett terjeszteni a logikai architektúrát is
  - × Egy utasítás 8 bemeneti operandust igényel
    - ~ Emiatt nagyon megnőtt a memóriaigény
  - × Memória sávszélesség növekedés
    - ~ Emiatt a L2 cache is a processzor lapkára kerül
  - × Kibővült a buszrendszer
  - × Meggyorsította a multimédiás alkalmazásokat (kép, videó, 3Ds játék)
  - × (nem tudom kell-e a FX és FP utasítások típusai és az azokhoz szükséges kibővítések, majd kérdezd meg!)

/ 12. Modul : Netburst és szálszintű párhuzamosítás /

+ Előzmények
  - Elértük az architektúra limitjeit, de a piac igényelte a teljesítmény további növekedését
  - Erre megoldások:
    × Frekvencia erőteljes növelése (amihez új architektúra kellett)
    × Új architektúrák (pl. VLIW, *ami „meghalt”*)
    × Párhuzamosság egyéb formái (pl. szálszintű vagy folyamat szintű párhuzamosság)

⬇

+ Frekvencia növelésének forrásai:
  - Gyártási csíkszélesség csökkentése => Kisebb helyen több tranzisztor => Gyorsabb elektron áramlás => Nőhet a frekvencia
  - Futószalag fokozatok hosszának csökkentése, ami a fokozatok több kisebb alfokozatra bontásával járt
    × A több fokozat, több párhuzamosan végrehajtható utasítást jelent, viszont több függőséget is!

+ Netburst architektúra (és a Pentium 4 CPU család) (4. DIA FONTOS) (ezt majd írd ki)
  - Pentium 4 CPU jellemzői
    × CISC architektúra, RISC maggal
    × Hosszú futószalagok
    × Hatékonyságban (energia igény) alulmaradt a konkurenciával szemben, viszont teljesítményben (sebesség) jobb volt
    × **Sikere főként annak köszönhető, hogy jobb védelme volt a túlhevülés ellen** (*Nem „olvadt le” ha nem volt (megfelelő) hűtés)*
      ~ Ezt a „thermal monitor” segítségével érte el: Túlmelegedés esetén a CPU lekapcsolta az órajelet majd csökkentette a frekvenciát és a magfeszültséget
      ~ Így csak lelassul vagy lefagy, de nem sérül

  - Legfontosabb újítások
    × Execution Trace Cache: L1 utasítás cache, már dekódolt RISC utasításokkal a feltételezett végrehajtásuk szerinti sorrendben (nem kellett a megfelelő utasítás kiválasztására „időt pazarolni”)
    × Hyper futószalag: Sok fokozatból áll (több függőség => alacsonyabb teljesítmény) és nincs benne dekódoló fokozat (az már az Execute Trace Cache-ben történik), mert az túl sok időt igényelt
    × Enhanced Branch Prediction: Egy továbbfejlesztett elágazásbecslő *(94–97% pontosság*
    × Quad Data Rate Bus: Egy új belső rendszerbusz az L1 és L2 cache-ek felé, ami két órajelgenerátor segítségével a buszfrekvencia x4-én továbbítja az adatokat
      ~ *(a memória nem gyorsult annyit mint a processzor, így a külső rendszerbuszt nem lehetett gyorsítani)*

        - Rapid Execution Engine: Az egyszerű FX (fixpontos szám) műveletek gyors végrehajtására szolgáló végrehajtó egység. Csak az egyszerű műveletek elvégzésére képes de azokat fél óraciklus alatt elvégzi
        - Replay System
            - Probléma: Sok fokozatú futószalag => Sok függőség => Sokat kell várnia a CPU-nak
            - Megoldás: Megbecsüljük az utasítások végrehajtási idejét, így ki tudjuk küldeni az adott utasítást úgy, hogy mire a hiányzó operandust használnunk kéne, addigra elérhető lesz. Ezzel egy csomó felesleges várakozási időt ki lehet tölteni a CPU-ban
            - Rossz becslés esetén az utasítás egy Replay Queue-ba kerül, ahonnan egy idő múlva kivesszük és újra megpróbáljuk végrehajtani

    - Következmények: Lehet hogy egy óraciklus alatt kevesebb utasítást tudunk végrehajtani, de az órajel annyival megnőtt hogy igazából egy időegység alatt több utasítást hajtunk végre mint ezelőtt

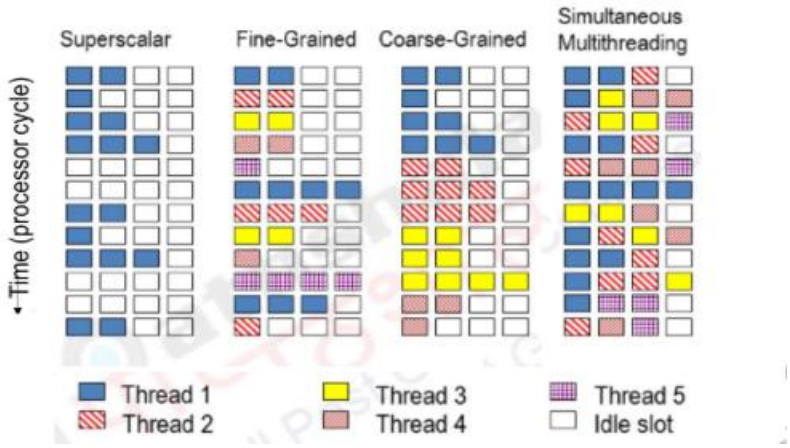
+ Szál szinten párhuzamos architektúrák
    - Mi az a szál?
        - A program legkisebb önállóan végrehajtható egysége
    - A szál szintű párhuzamosság kihasználásához már az OS támogatása is szükséges
    - A cél a CPU egy-egy fizikai magjában a feldolgozás során kialakuló üresjáratok kihasználása

    - A párhuzamosság formái:
        - Implicit: A programozó szekvenciális kódot ír és a hardver és szoftver alakítja át párhuzamossá
        - Explicit: A programozó kifejezett párhuzamos programot ír

    - Egy szálban nincsen párhuzamosság. A szál szintű párhuzamosság lényege hogy párhuzamosan több szállal dolgozunk *(nem pedig egy szálon beül párhuzamosítunk)*
    - Szálak származtatása:
        - Különböző alkalmazásokból
        - Ugyanabból az alkalmazásból
            - Multitasking
            - Multithreading
    - Többszálúság csoportosítása
        - Hardveres többszálúság: Többszálú alkalmazás fut a többszálú CPU-n
        - Szoftveres többszálúság: Többszálú alkalmazás fut az egyszálú CPU-n időosztásos technikával
    - Többszálú CPU-k típusai
        - SMP (Symmetric Multiporcessing): Egynél több (pl. kettő) mag futtatja a szálakat párhuzamosan
        - SMT (Symmetric Multithreading): Egy mag képes több szálat egyszerre két párhuzamosan futtatni (kis komplexitással nagy teljesítmény növekedés)

- Szál szinten párhuzamos architektúrák osztályzása (szemcsézettség)
  - × Finoman szemcsézett: A CPU órajelenként vált a szálak között
    - ~ Szálak váltásánál mindig el kell menteni az előző szál kontextusát ami és a másik szálét betölteni ami nagyon lassú lenne
    - ~ Megoldás: Sok regiszter, amiben el tudjuk tárolni a kontextusokat és egy kontextus kapcsoló segítségével ezek között késleltetés nélkül tudunk váltani
  - × Durván szemcsézett: Ha egy szál futása megszakad, akkor váltunk egy másik szálra
    - ~ Egyetlen probléma, hogy a megakadást érzékelni kell ami 1-2 óraciklust vesz igénybe
  - × SMT: Az Intel Hyper Threading alapja. Az összes szál futása párhuzamosan történik. Ha egy végrehajtó egység várakozik amíg egy másik szál megszerzi a szükséges függőségeit, addig „lefoglaljuk” egy másik szál utasításaival

Példa: különbség az egyszálas és az előző lapon felsorolt többszálas CPU-k működése között: (mindegyik 4 utasítás széles) jól látszik az üresjárati ciklusok számának csökkenése



- A párhuzamosság megvalósításához teljesíteni kell pár hardveres előkövetelményt
  - × Bizonyos erőforrásokat meg kell többszöröznünk (pl. egy-egy program counter szálanként)
  - × Az erőforrásokat meg kell tudni osztani valahogyan a szálak között és egyesíteni ha valamelyik szál inaktív lesz

- Intel Hyper Threading
  - × Két szálas SMT architektúra
  - × Egy mag küldi ki két szál utasításait párhuzamosan
    - ~ Az egy fizikai mag az OS számára két külön logikai magnak látszódik és úgy is dolgozik vele
    - ~ Kicsit több tranzisztor kell hozzá, de az relatív elenyésző extra
    - ~ Két üzemmód van:
      - Δ ST (Single Task): Egyszerre egy szál végrehajtása történik és az összes erőforrást az az egy aktív szál (vagy az OS „szeme” szerint az az egy CPU) használja
      - Δ MT (Multi Taks): Több száll végrehajtás párhuzamosan, megosztott erőforrásokkal