

# Számítógép Architektúrák II.

(MIVIB344ZV)

1. előadás: Bevezetés, számítógép generációk.  
Neumann-Harvard architektúrák.

Előadó: Dr. Vörösházi Zsolt  
[voroshazi.zsolt@mik.uni-pannon.hu](mailto:voroshazi.zsolt@mik.uni-pannon.hu)

# Számítógép Architektúrák II. kurzus



<https://oktatas.mik.uni-pannon.hu/course/view.php?id=1609>

# Feltételek:

- Követelmények: lásd tematika
  - kisZH-k** (lásd tematika)
  - 1 ZH** (lásd tematika)
  - PótZH** (lásd tematika)
- Megajánlott jegy: **eredmény(összpontszám) >= 4**
- Aláírás/Vizsgára bocsátás feltétele:
  - Összesített eredmény >= 40%**
- Óralátogatás: **kötelező!**
- Vizsga: írásbeli (szóbeli)
-  **Könyvek:**
  - L. Howard Pollard – Computer Design and Architectures (Prentice-Hall 1990)*
  - Órai diasorok*
- **Záróvizsga tárgy:**
  - Informatika BSc/Üzemmérnök BProf - Informatika t.cs. tételei
  - Villamosmérnök BSc – választható tárgy

# Kapcsolódó jegyzet, segédanyag:

-  Angol nyelvű könyv:  
<http://www.virt.uni-pannon.hu> → Oktatás → Tantárgyak → Számítógép Architektúrák II.  
(**chapter1/.../8.pdf**) + további segédletek
  - Bevezetés: Számítógép Generációk (**chapter01.pdf**)
-  Fóliák, óravázlatok .ppt (.pdf)
- Frissítésük folyamatosan „//frissítve”

# Elérhetőségek



## Előadás-gyakorlat

**Dr. Vörösházi Zsolt**

egyetemi adjunktus

VIRT, Képfeldolgozás Kutatólaboratórium

I-208 szoba

☎ : +36 88 624 324

✉ : [voroshazi.zsolt@mik.uni-pannon.hu](mailto:voroshazi.zsolt@mik.uni-pannon.hu)

# További ajánlott irodalom

-  Dr. Holczinger, Dr. Göllei. Dr. Vörösházi:  
Digitális Technika I. (TAMOP 4.1.2A - 2012) :  
Digitalis technika I TAMOP
-  Dr. Holczinger, Dr. Göllei. Dr. Vörösházi:  
Digitális Technika II. (TAMOP 4.1.2A - 2013) :  
Digitalis technika II TAMOP

# Előzmények (PE tantárgyak)

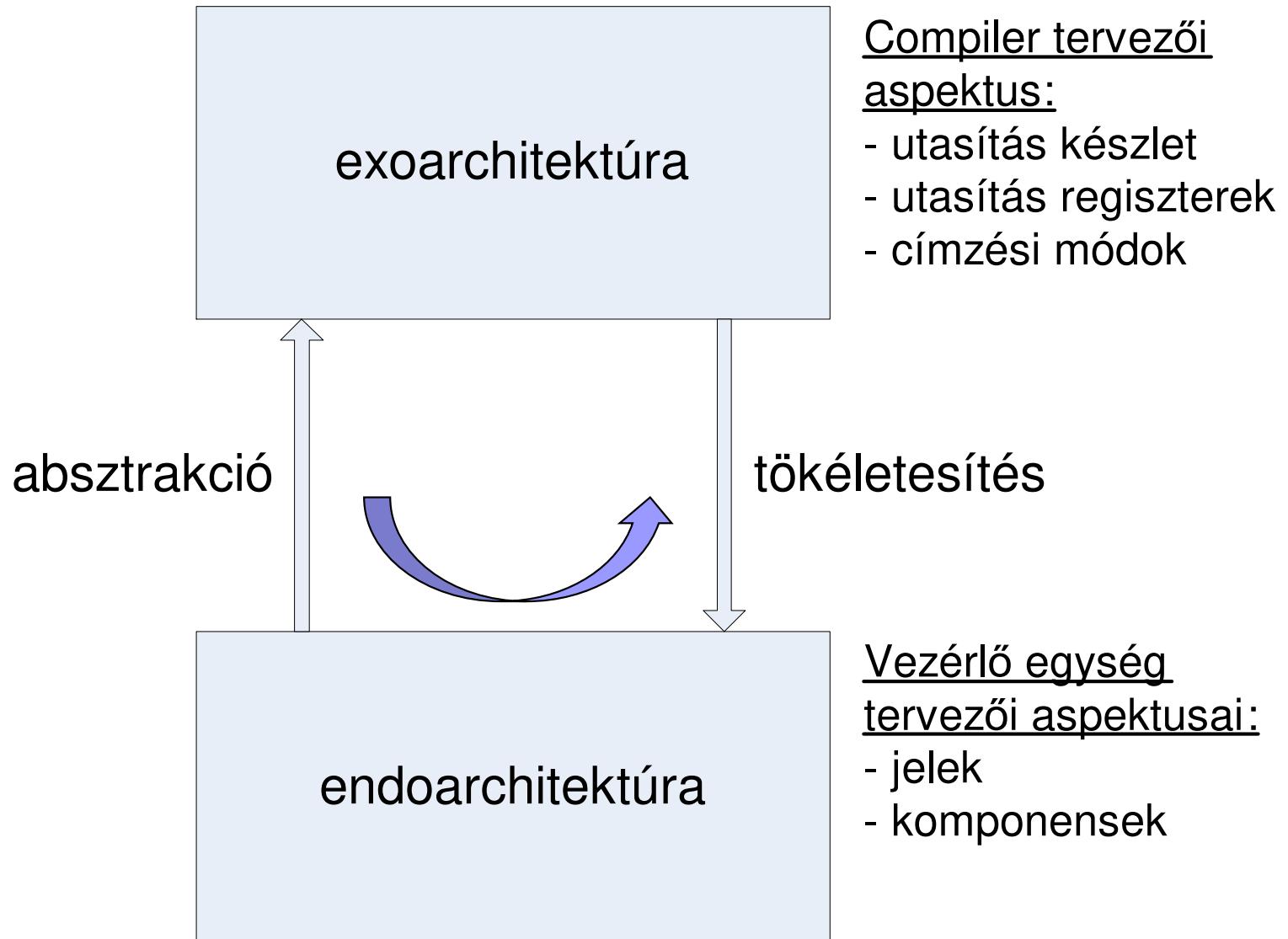
- Információs Technológia
  - Informatikai alapfogalmak
  - Számítástechnika fejlődéstörténete
  - Logikai tervezés (K.H.)
  - Számrendszerek, számábrázolás
- Digitális Áramkörök / Számítógép Architektúrák I.
  - Kombinációs-, és Szekvenciális hálózatok tervezése
- Operációs Rendszerek
  - Memória szervezés és védelem (cache \$)
- Számítógépes Perifériák

# Alapfogalmak:

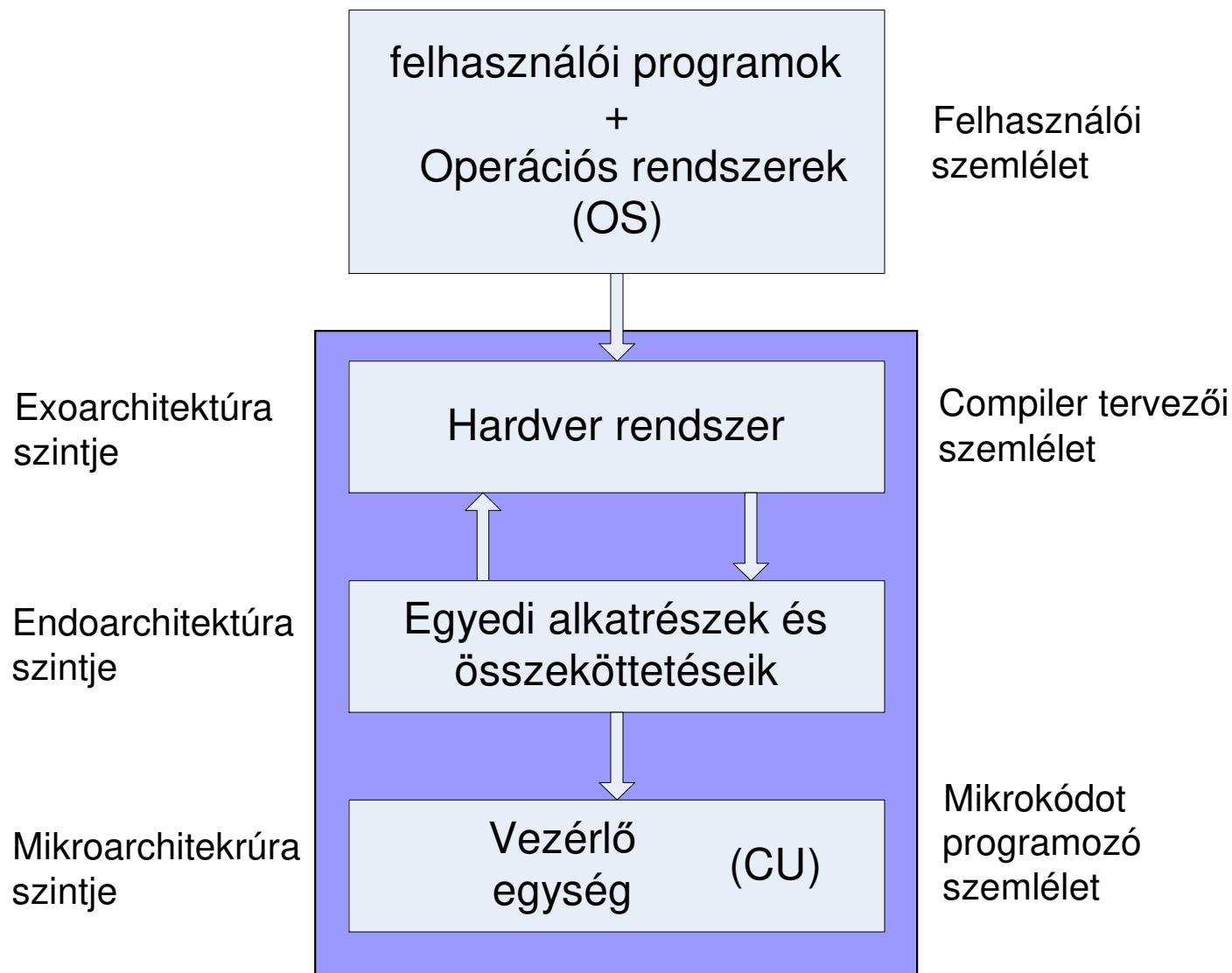
- **A számítógép architektúra** a hardver egy általános *absztrakciója*: a hardver struktúráját és viselkedését jelenti (más rendszerek egyedi, sajátos tulajdonságaitól eltekintve)
- **Architektúrális tulajdonságok** nemcsak a funkcionális elemeket, hanem azok belső felépítését, struktúráját is magába foglalják
- **Mikro-architektúra**: egy számítógép kapcsolási sémája, hardver-alapú működésének leírása.
- **Számítógép architektúra** = utasítás készlet (ISA) + rendszer mikro-architektúrája.

# Számítógép architektúra:

## Exoarchitektúra – endoarchitektúra:

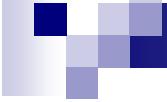


# Számítógép architektúra definíciója:



# Számítógépes rendszerekkel szembeni tervezői követelmények:

- Aritmetika (ALU) megtervezése, algoritmusok, módszerek elemzése, hogy a kívánt eredményt elfogadható *időn* belül biztosítani tudja
- Utasításkészlet – vezérlés (ISA-CU)
- A részegységek közötti kapcsolatok / összeköttetések a valós rendszert szemléltetik
  - CFG, DFG a főbb komponensek között
- Számítógép és perifériák közötti I/O kommunikációs technikák

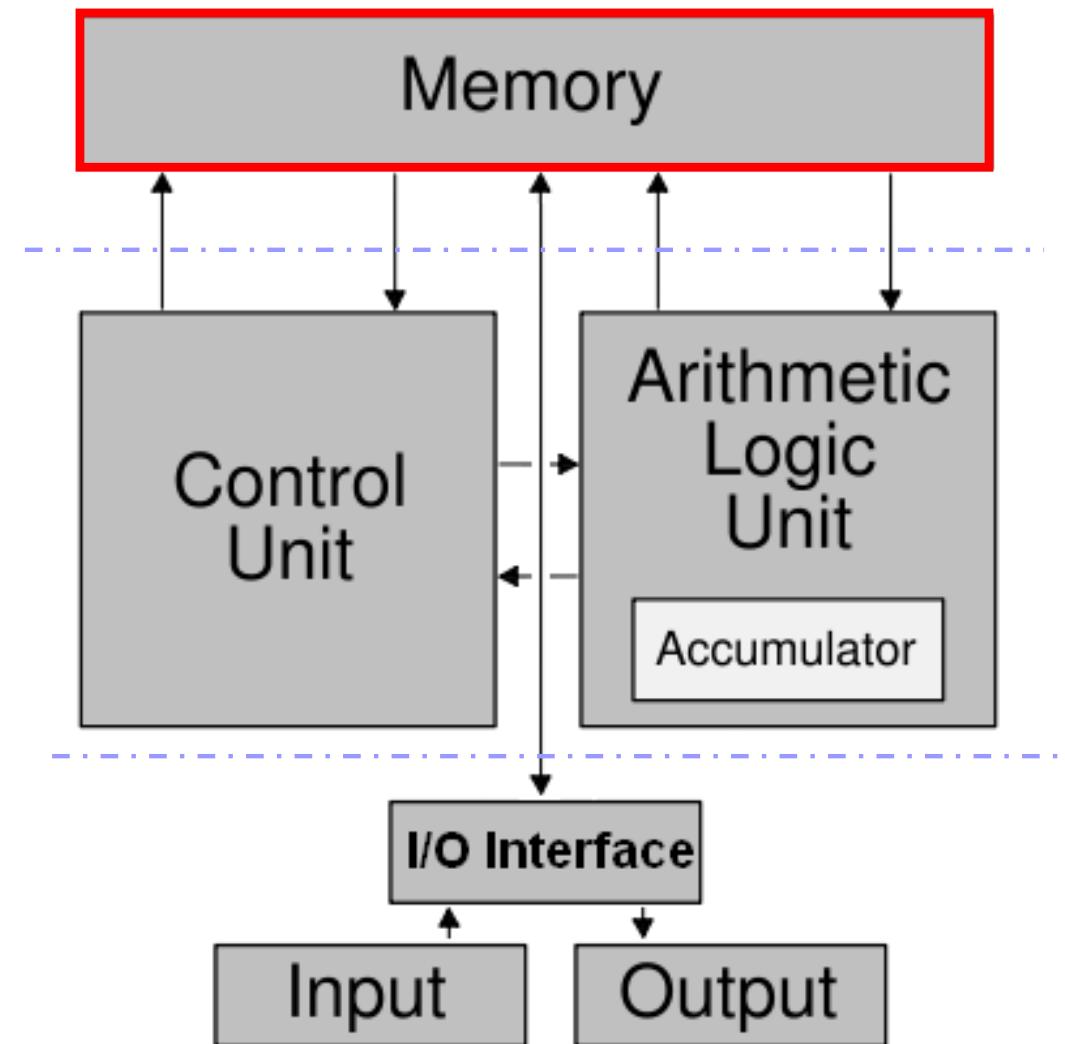


# Neumann vs. Harvard számítógép architektúrák

# A.) Neumann architektúra

## ■ Számítógépes rendszer modell:

- CPU (CU + ALU) szeparáció
- Egyetlen, de különálló tároló elem (**utasítások** és **adatok** együttes tárolására)
- Univerzális Turing gépet implementál (TM)
- „Szekvenciális” architektúra (SISD)



# Von Neumann architektúra

- „De Facto” szabvány: „*single-memory architecture*”. Az **adat-** és **utasítás**-címek a memória (tároló) ugyanazon címtartományára vannak leképezve (mapping). Ilyen típusú pl:
  - EDVAC (Neumann), egyenletmegoldó tárolt-programú gép
  - Eckert, Mauchly: ENIAC, UNIVAC (University of Pennsylvania) – numerikus integrátor, kalkulátor
  - A mai rendszerek modern mini-, mikro, és mainframe számítógépenek operatív memóriája is ezt az architektúrát követi.
    - általános programozói szemléletmód



# Neumann elvek

- számítógép működését tárolt program vezérli (Turing);
- a vezérlés leírása a vezérlés-folyam (control-flow graph - CFG) segítségével; Fontos lépés itt az adatút megtervezése.
- a gép belső operatív tárolójában a program utasításai és a végrehajtásukhoz szükséges adatok egyaránt megtalálhatók (**közös utasítás és adattárolás**, a program felülírhatja önmagát – **Neumann architektúra**);
- az *aritmetikai / és logikai* műveletek végrehajtását önálló részegység (ALU) végzi; CU – vezérlő egység szeparáció.
- az adatok és programok beolvasására és az eredmények megjelenítésére önálló egységek (IO perifériák) szolgálnak;
- 2-es (bináris) számrendszer alkalmazása.
  - PI: EDVAC computer, ENIAC stb.

# Fix vs. tárolt programozhatóság

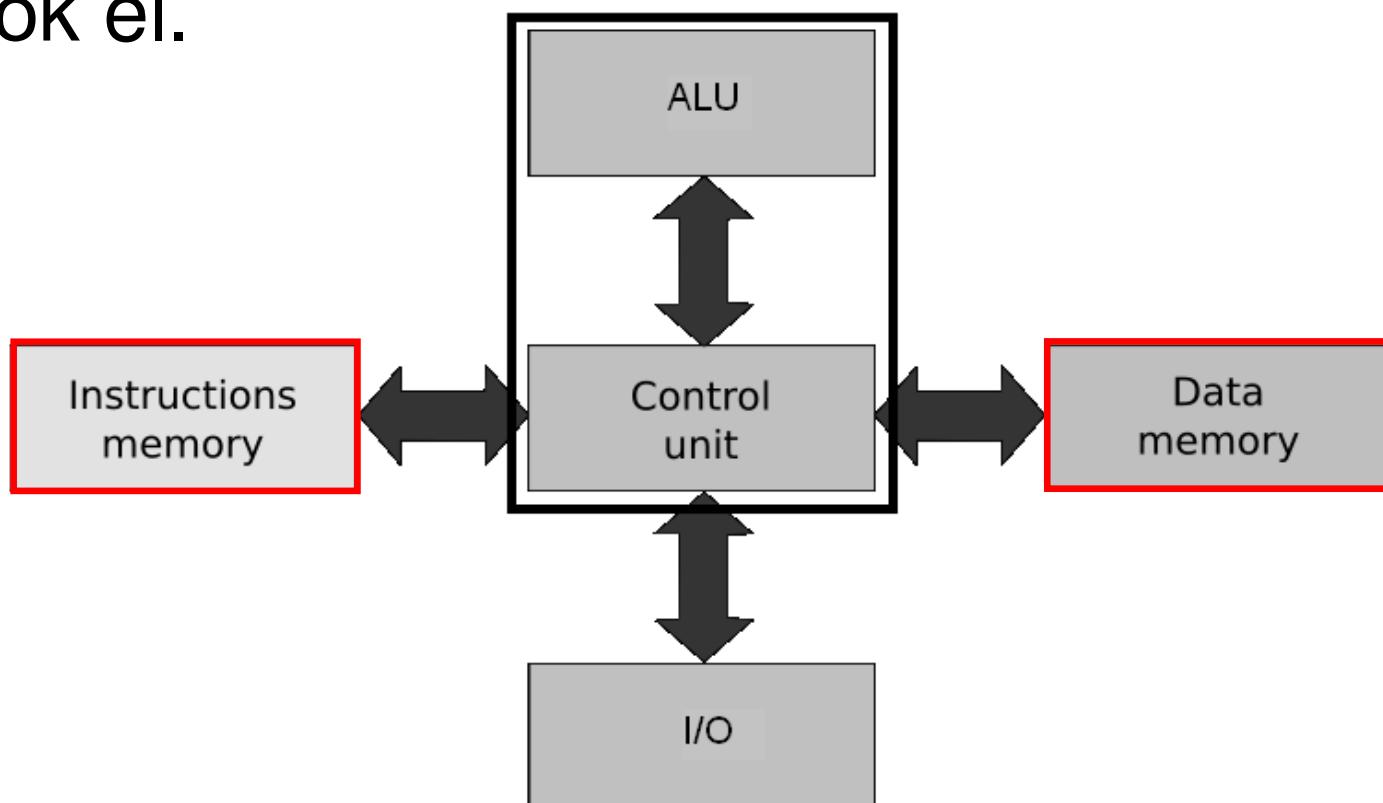
- Korai számítási eszközök **fix** programmal rendelkeztek (nem tárolt programozható): pl: kalkulátor
  - - Program változtatása: „átvezetékezés”, struktúra újratervezéssel lehetséges csak (lassú)
  - - Újraprogramozás: folyamat diagram → előterv spec. (papíron) → részletes mérnöki tervek → nehézkes implementáció (hibalehetőség)
- **Tárolt** programozhatóság ötlete:
  - + Utasítás-készlet architektúra (ISA): RISC, CISC
  - + Változtatható *program*: utasítások sorozata
  - + Nagyfokú flexibilitás, *adatot* hasonló módon tárolni, és kezelní (assembler, compiler, automata prog. eszk.)

# Neumann architektúra hátrányai

- „Önmagát változtató” – kártékony programok (self-modifying code / vulnerability - sebezhetőség):
  - Már eleve hibásan megírt program „kárt” okozhat önmagában ill. más programokban is: „malware”=„malfunction”+„software”.
  - OS szinten: rendszer leálláshoz is vezethet
  - Pl. Buffer túlcsordulás: kezelése szintenkénti hozzáféréssel, memória védelemmel!!
- **Neumann „bottleneck”:** sávszélesség korlát a CPU és memória között, amely a nagymennyiségű adatok továbbítása során léphet fel.
  - ezért kellett bevezetni a CPU –ban a Cache memóriát (\$)
- A nem-cache alapú Neumann rendszerekben, egyszerre vagy csak adat írás/olvasást, vagy csak az utasítás beolvasását lehet elvégezni (egy buszrendszer!)

## B.) Harvard architektúra

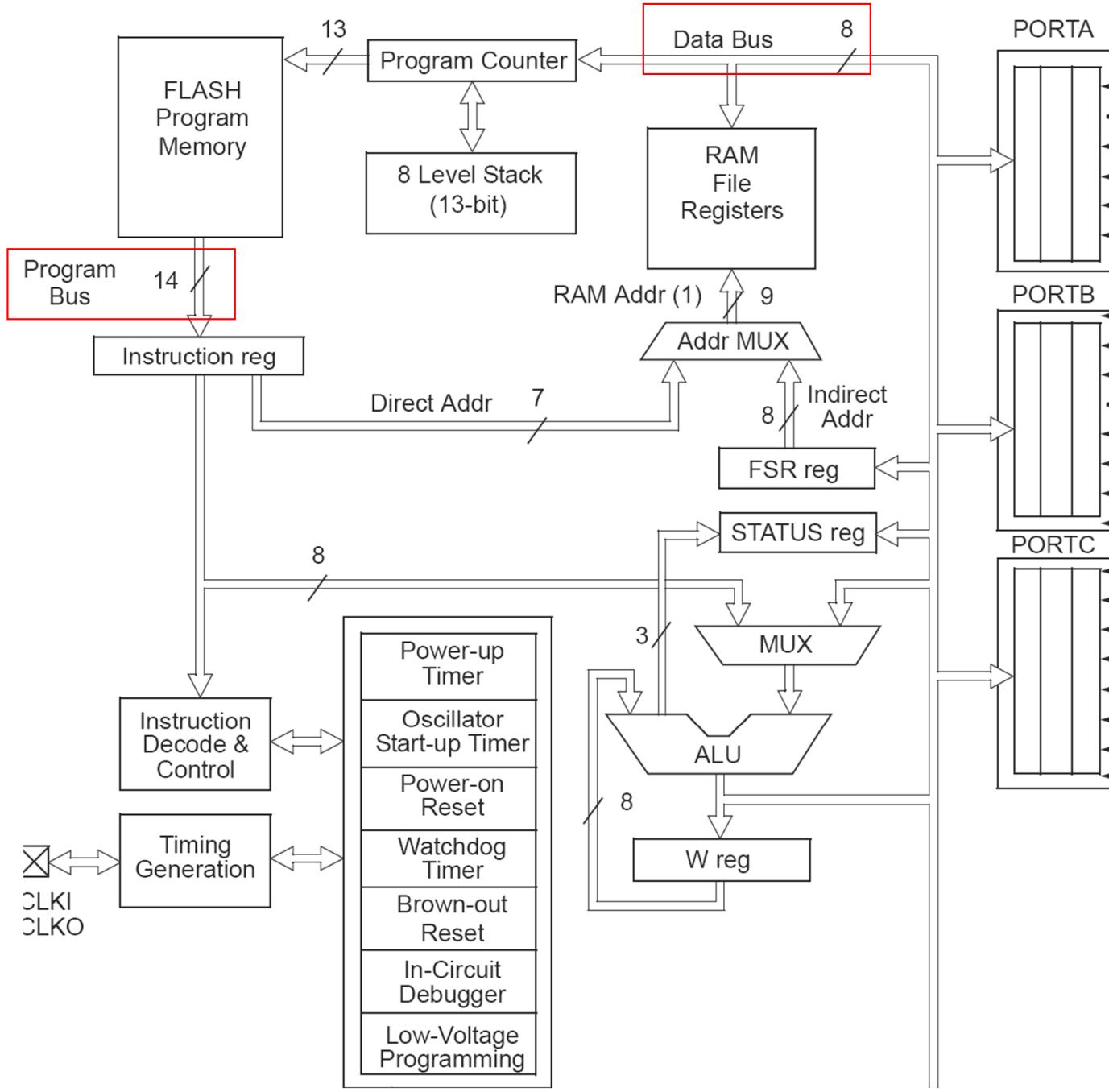
- Olyan számítógéprendszer, amelynél a program *utasításokat* és az *adatokat* fizikailag **különálló** memóriában tárolják, és külön buszon érhetők el.



# Harvard architektúra

- Eredet 1944: Harvard MARK I. (relés alapú rdsz.)
- További fontosabb példák:
  - Intel Pentium processzor család L1-szintű különálló adat- és utasítás-cache (\$) memóriája
  - ARM processzorok újabb pl. Cortex sorozatai (L1 cache)
  - Beágyazott („embedded”) rendszerek processzorai:
    - Mikrovezérlők (MCU) különálló utasítás-adat buszai és memóriái (MicroChip=Atmel, Cypress, Texas, ... stb.)
    - FPGA-alapú beágyazott rendszerek: MicroBlaze, PowerPC cache memóriái, buszrendszerei.
    - DSP jelfeldolgozó processzorok (RAM, ROM memóriái)
      - Texas Instruments

# Példa: PIC 14-bites mikrovezérlő



# Harvard arch. tulajdonságai

- Nem szükséges a memória (shared) osztott jellegének kialakítása:
  - + A memória szóhosszúsága, időzítése, tervezési technológiája, címzése is különböző lehet.
  - Az utasítás (program) memória gyakran szélesebb mint az adat memória (nagyobb utasítás memóriára lehet szükség)
  - Utasításokat a legtöbb rendszer esetében olvasható ROM-ban (esetleg PROM) tárolják, míg az adatot írható/olvasható memóriában (pl. RAM-ban).
    - Ezért nincs malware probléma (mint Neumann esetben)
  - + A számítógép különálló buszrendszer segítségével egyidőben akár egy utasítás beolvasását és adat írását/olvasását is el lehet végezni (cache nélkül is).

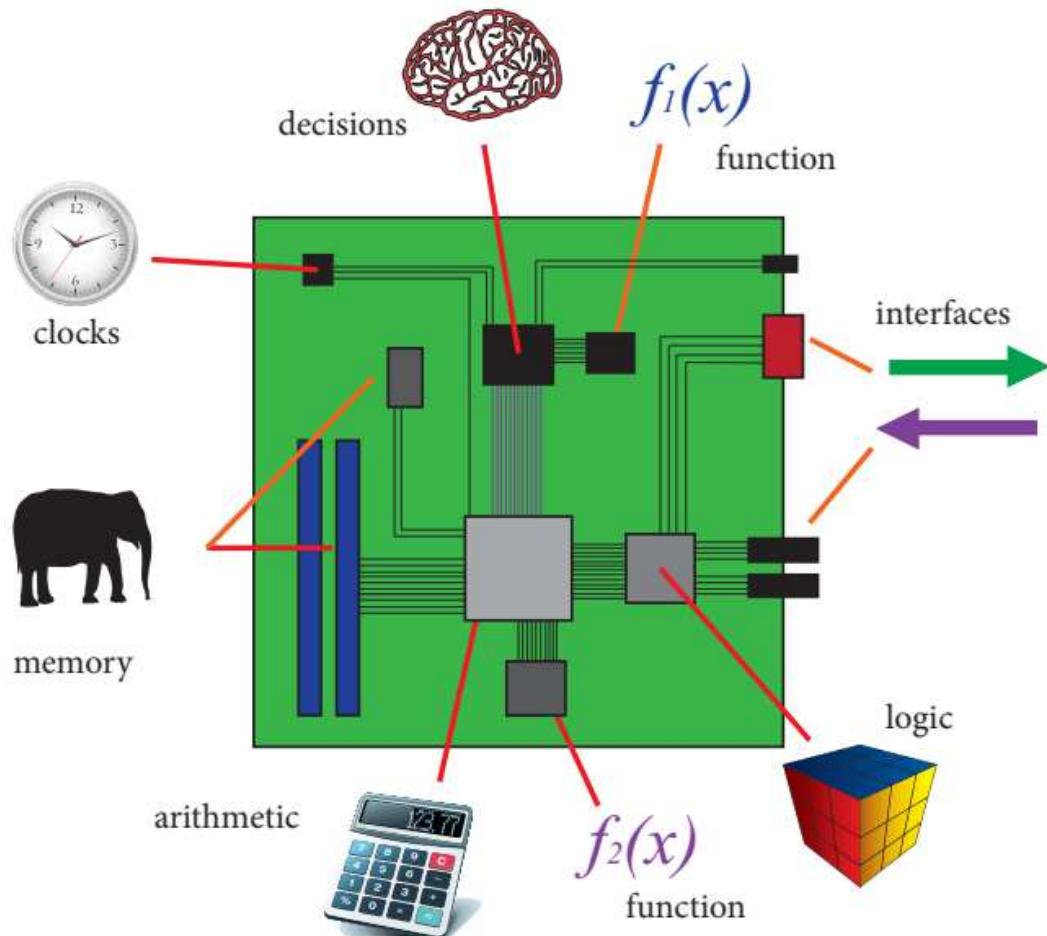
# „Módosított” Harvard architektúra

- Modern számítógép rendszerekben az utasítás-memória és CPU között olyan közvetlen adatút biztosított, amellyel az olvasható *adatot is, mint utasítás-szót* lehet elérni a program memóriából:
  - Konstans adat (pl: string, inicializáló érték) utasítás memóriába töltésével a változók számára további helyet spórol(hatunk) meg az adatmemóriában.
    - Adat intenzív műveletek
  - Mai modern rendszereknél a Harvard architektúra megnevezés alatt, **ezt a módosított változatot értjük.**
  - Gépi (alacsony) szintű assembly utasítások.

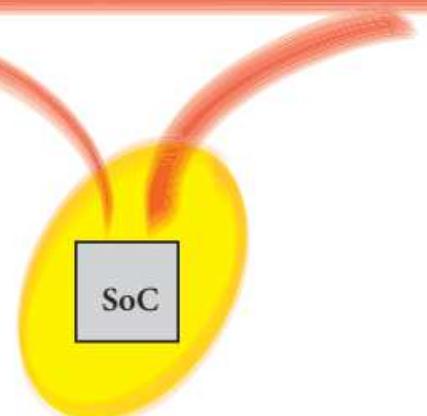
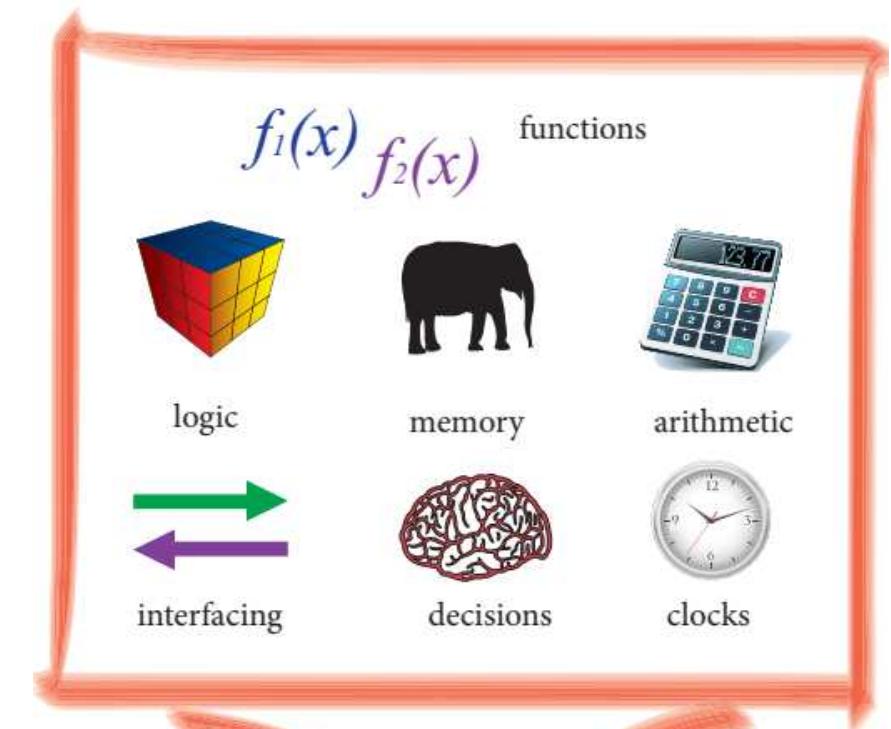
# Harvard architektúra hátrányai

- Mai korszerű egychipes rendszereknél (pl. **SoC**: System On a Chip - 2005), ahol egyetlen chipen van implementálva minden funkció, nehézkes lehet a különböző memória technológiák együttes használata az utasítások és adatok kezelésénél. Ezekben az esetekben a külső memória ált. Neumann elvű.
- Korábban hátrányként említették: a magas szintű nyelveket (pl. ANSI C szabvány) , melyek közvetlen támogatása mára sokat fejlődött (új nyelvi konstrukció az utasítás adatként való elérésére).

# System-On-a-Board vs. System-On-a-Chip



vs.



# Harvard – Neumann együttes architektúra megvalósítás

- Mai, nagy teljesítményű számítógép architektúrákban a két elvet együttesen kell értelmezni:
- Példa: Cache rendszer
  - Programozói szemlélet (Neumann): cache ‘*miss*’ esetén a fő memóriából kell kivenni az adatot (cím → adat)
  - Rendszer, hardver szemlélet (Harvard): a CPU ún. „on-chip” cache memóriája különálló adat-, és utasítás cache blokkokból áll, amelyből a CPU *cache ,hit*’ esetén közvetlenül tud adatot/utasítást venni.

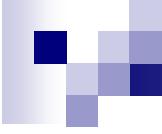


Meltdown



Spectre

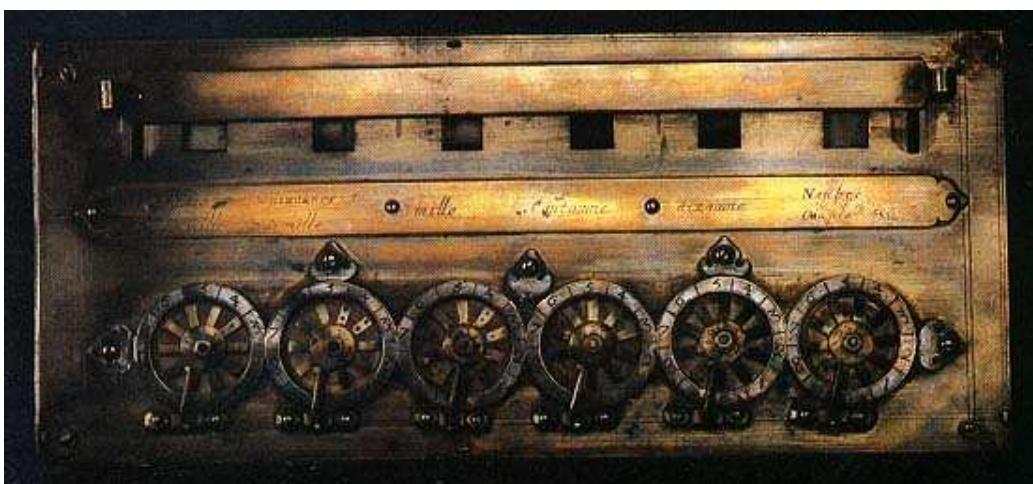
- Meltdown: illetéktelen hozzáférés memória tartalomhoz, CPU adat cache-én keresztül
  - szoftveres memória védelem (OS update - 5-30% lassulás), mikrokód javítása (FW update), később új mikroarchitektúra kell
- Spectre: branch misprediction – ún. spekulatív végrehajtás, mikroarchitektúrális támadás, illetve felhasználói módból OS kernel memória olvasható
  - Hardveres védelem, OS update, később új mikroarchitektúra kell
- Mely rendszereket érintheti:
  - Intel, AMD, ARM, RISC, Nvidia (GPU), Apple ...
  - Linux, MacOS, Windows ...



# Számítógép generációk

# Eredet - korai számítási eszközök I:

- 1642: Pascal – mechanikus kalkulátor (+,-)
- 1671: Leibnitz – kalkulátor 4 alapműv.

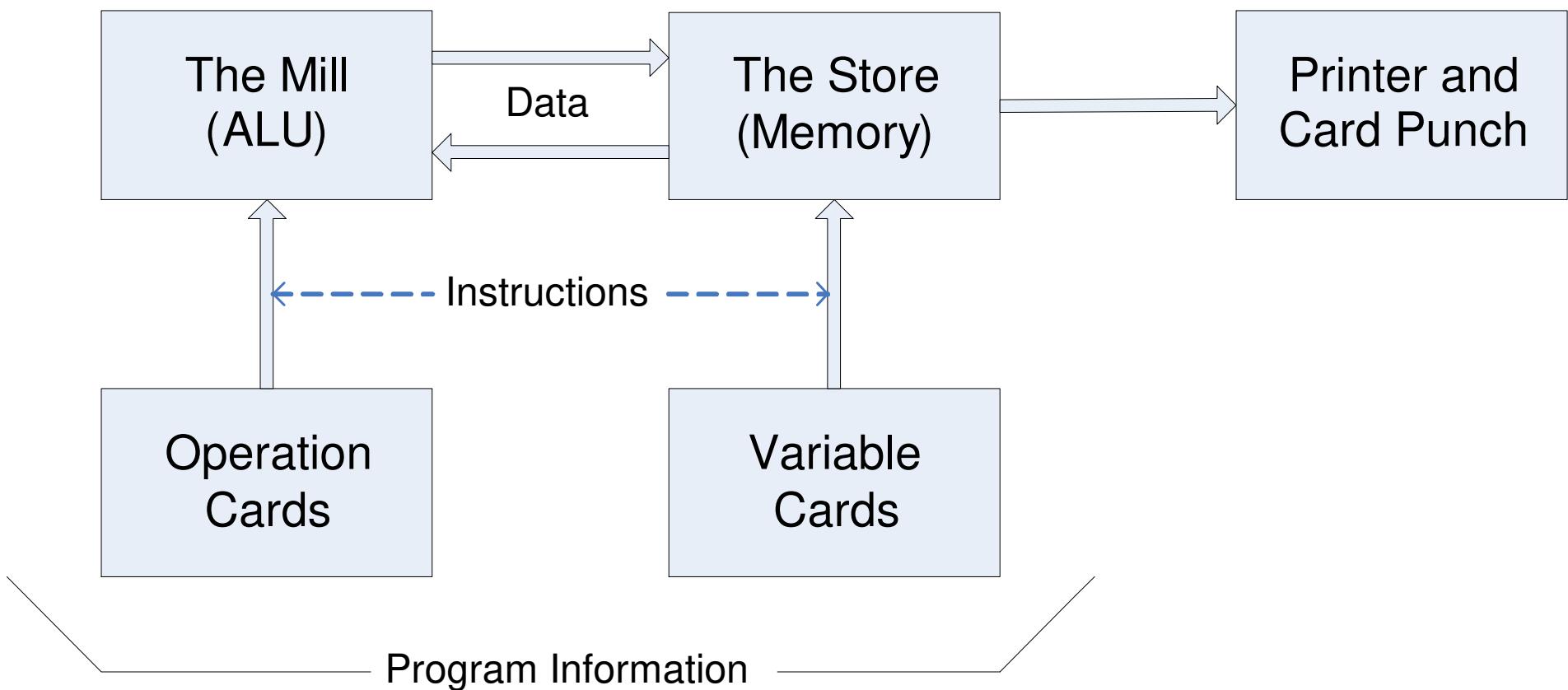


# Eredet - korai számítási eszközök I (folyt.)

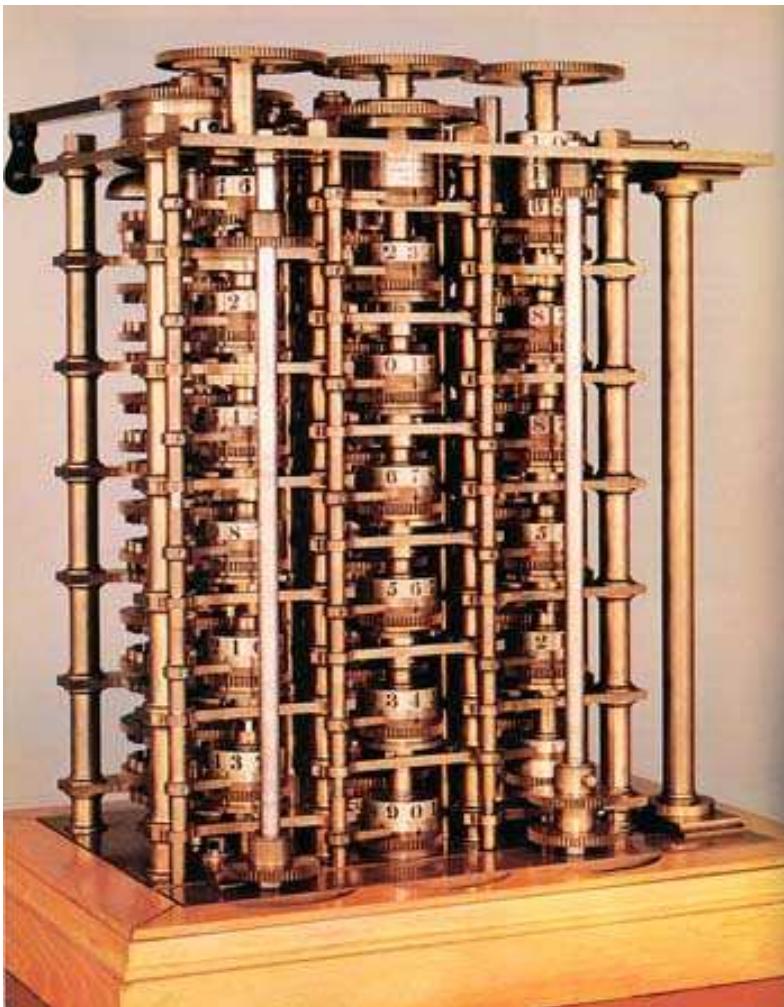
## ■ 1823: Babbage

- *Differencia Gép*: véges differencia módszer, ciklusos végrehajtás, automatikusan generált mat. táblákat
- *Analitikus Gép*: mai gépekkel szembeírható hasonlóság, mat. fgv.-ek végrehajtása. MILL – aritmetika: 4 alap.műv. ('+' 1sec, '\*' 1 min alatt), felt. elágazást is támogatta. Memóriája számoló „korongos”: 1000 db 50 jegyű számot tárolt.

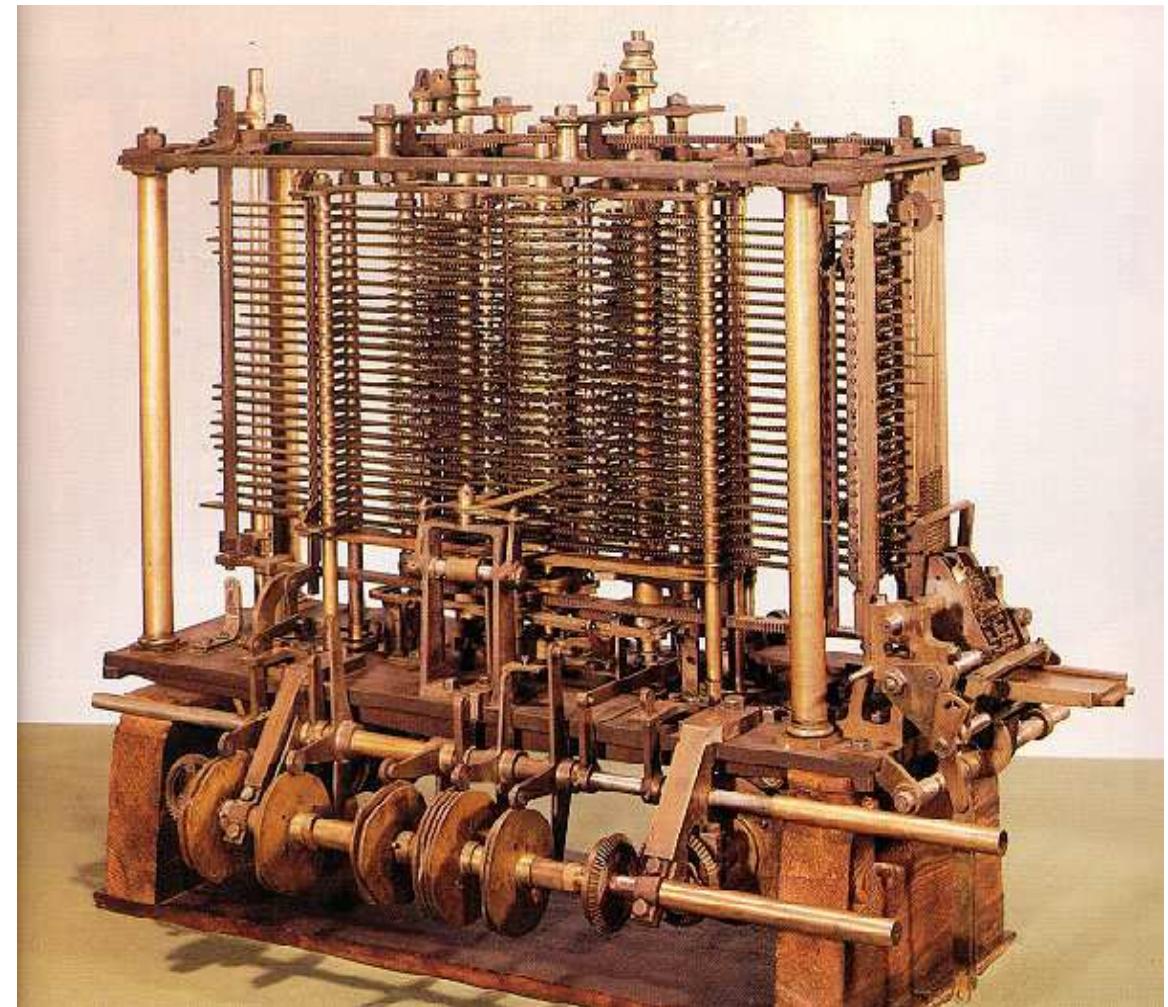
# Babbage – Analitikus Gép



# Babbage



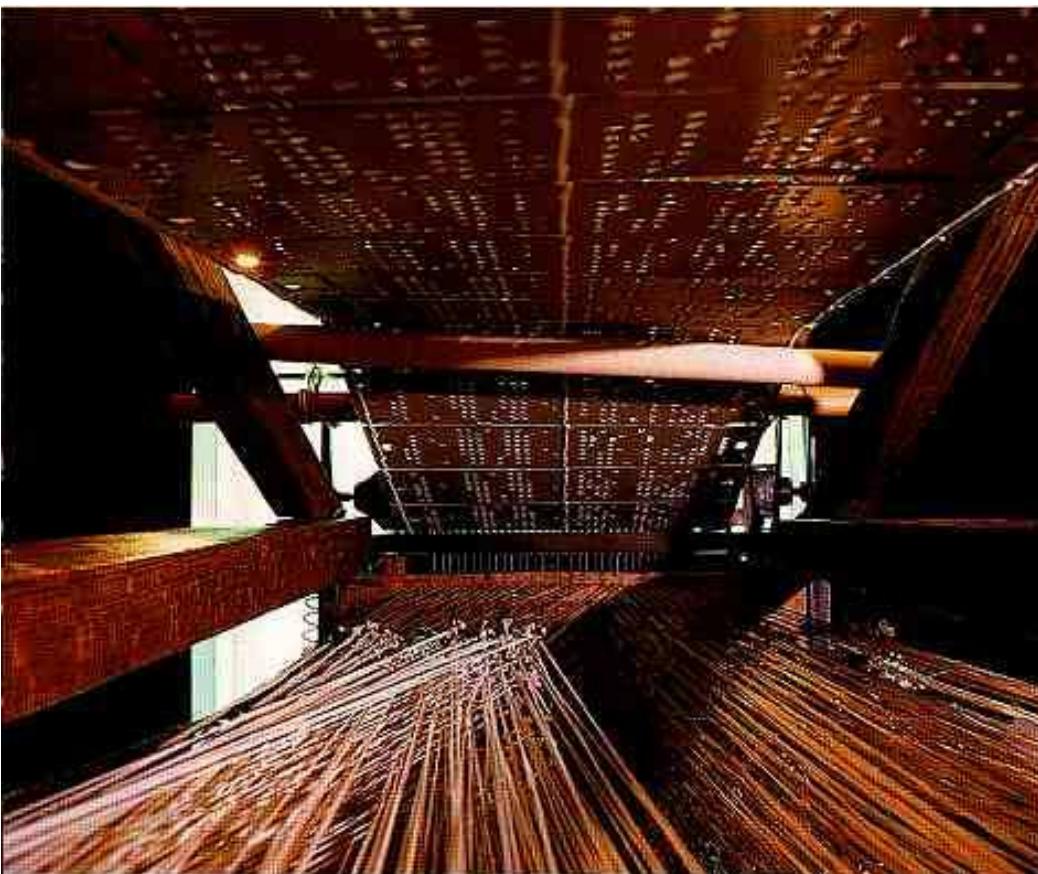
Differencia gép



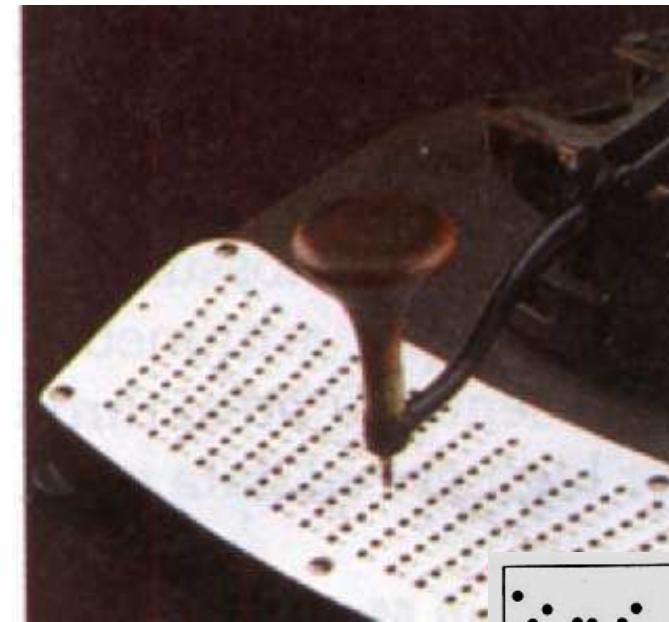
Analitikus gép

# Eredet - korai számítási eszközök II (folyt.):

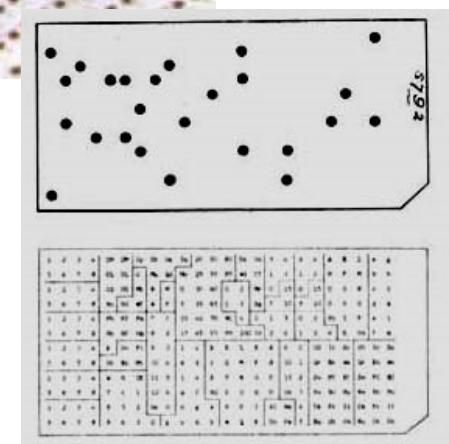
- 1801: Joseph Marie Jacquard: „loom” („szövőszék” ) – „lyukkártya szerű” szalag, (számítási folyamat automatizálása)
- 1890: Hollerith – lyukkártya – US népszámlálás adatainak feldolgozására (1911 – IBM)
- 1930: Zuse: elektromechanikus gép
  - Z1: mechanikus relék, 2-es számrendszer!
  - Z3 (1941): első műveleti programvezérelt általános célú gép, lyukszalagos bemet (Neumann elvet követő)
- 1939: Aiken – **MARK I (Harvard)** relés aritmetika, számoló fogaskerekes tároló. **Harvard architektúra**: különálló program/kód és adatmemória! 72 db 23 jegyű szám



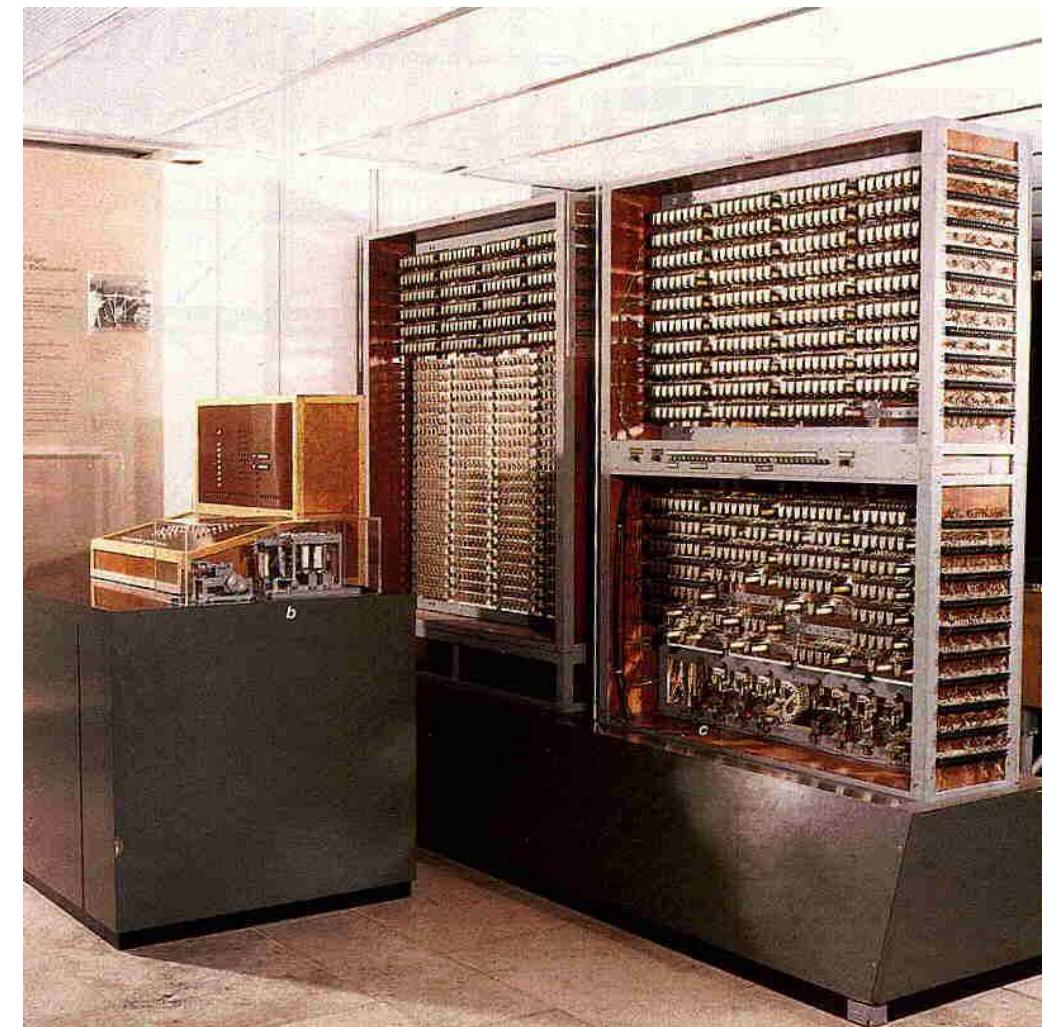
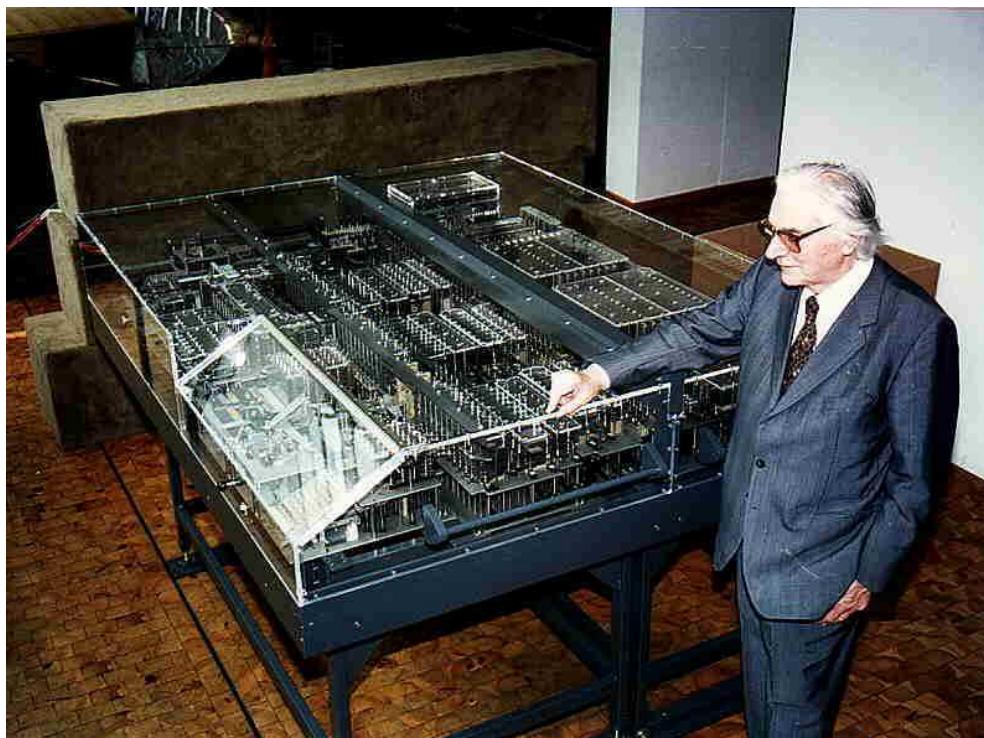
Jacquard „szövőgépe”



Hollerith - lyukkártya



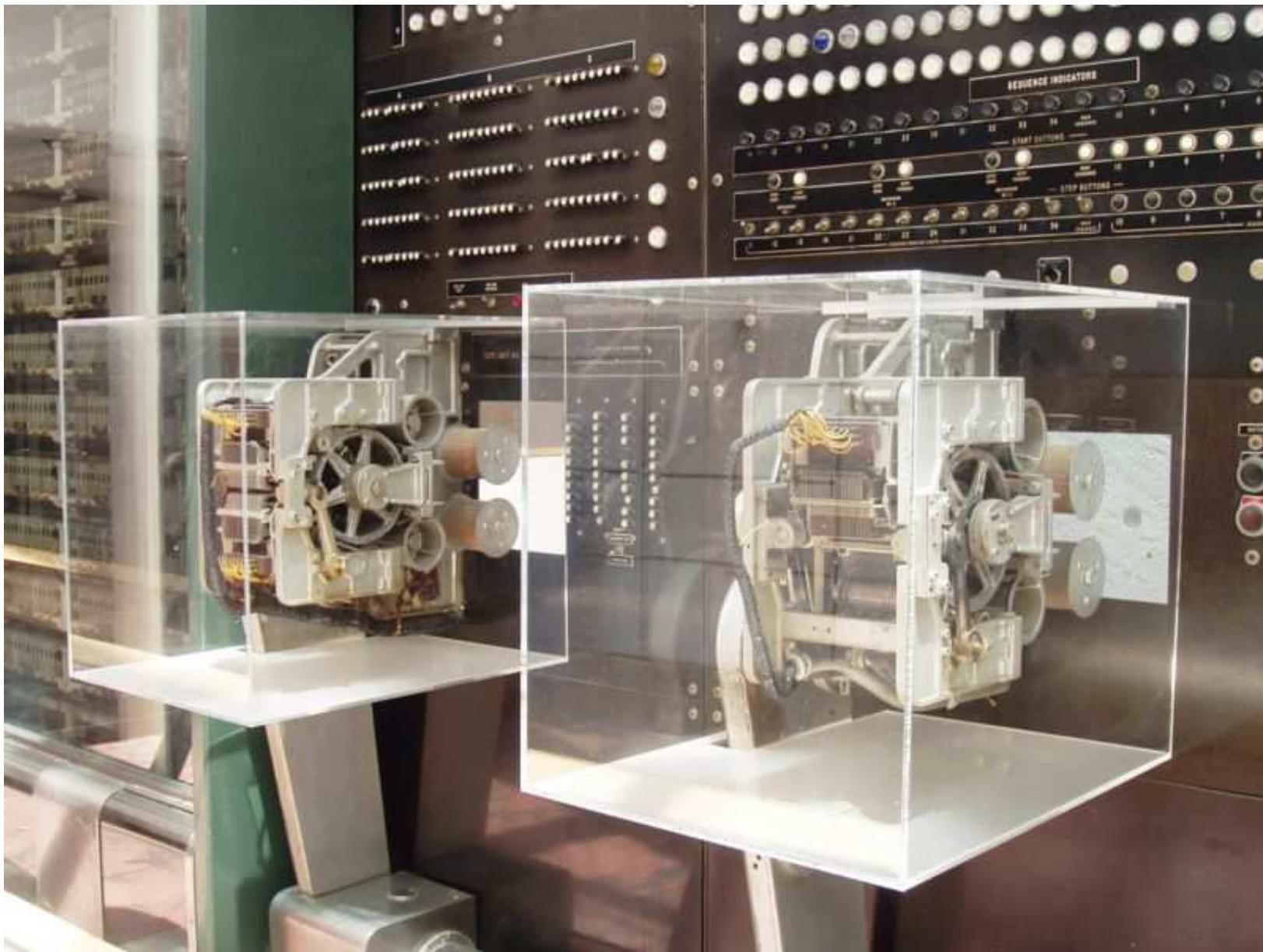
# Zuse Z1 és Z3



# Harvard MARK I

- Howard H. Aiken (Harvard University) – 1944
- Relés alapú aritmetika, mechanikus, korai sz.gép rendszer. Korlátozott adattároló képesség. (72 db 23 bites decimális számot tárol)
- ***Harvard architektúra***
  - Lyukszálon tárolt 24-bites utasítások
  - Elektro-mechanikus fogaskerekes számlálókon tárolt 23 bites adatok
  - Utasítást adatként nem lehetett elérni!
- 4KW disszipáció, 4.5 tonna, 765.000 alkatrész: relék, kapcsolók
- Műveletvégzés: +,-: 1 sec, \*: 6 sec, /: 15.3 sec
- Logaritmus, trigonometrikus fgv. számítás: 1 min

# MARK I.



# Eredet - korai számítási eszközök

## III (folyt.):

- 1937: Berry computer (Iowa Egyetem) – John Atanasoff első elektronikus számítógép rendszer
  - egyenlet rdsz.ek Gauss eliminációjára
  - ! 2-es számrendszer
  - Tárolás: kondenzátoron (mint DRAM-nál)
  - ALU: aritmetikai / logikai szeparáció
  - Részek teljes elkülönítése: memória, I/O perifériák, ALU

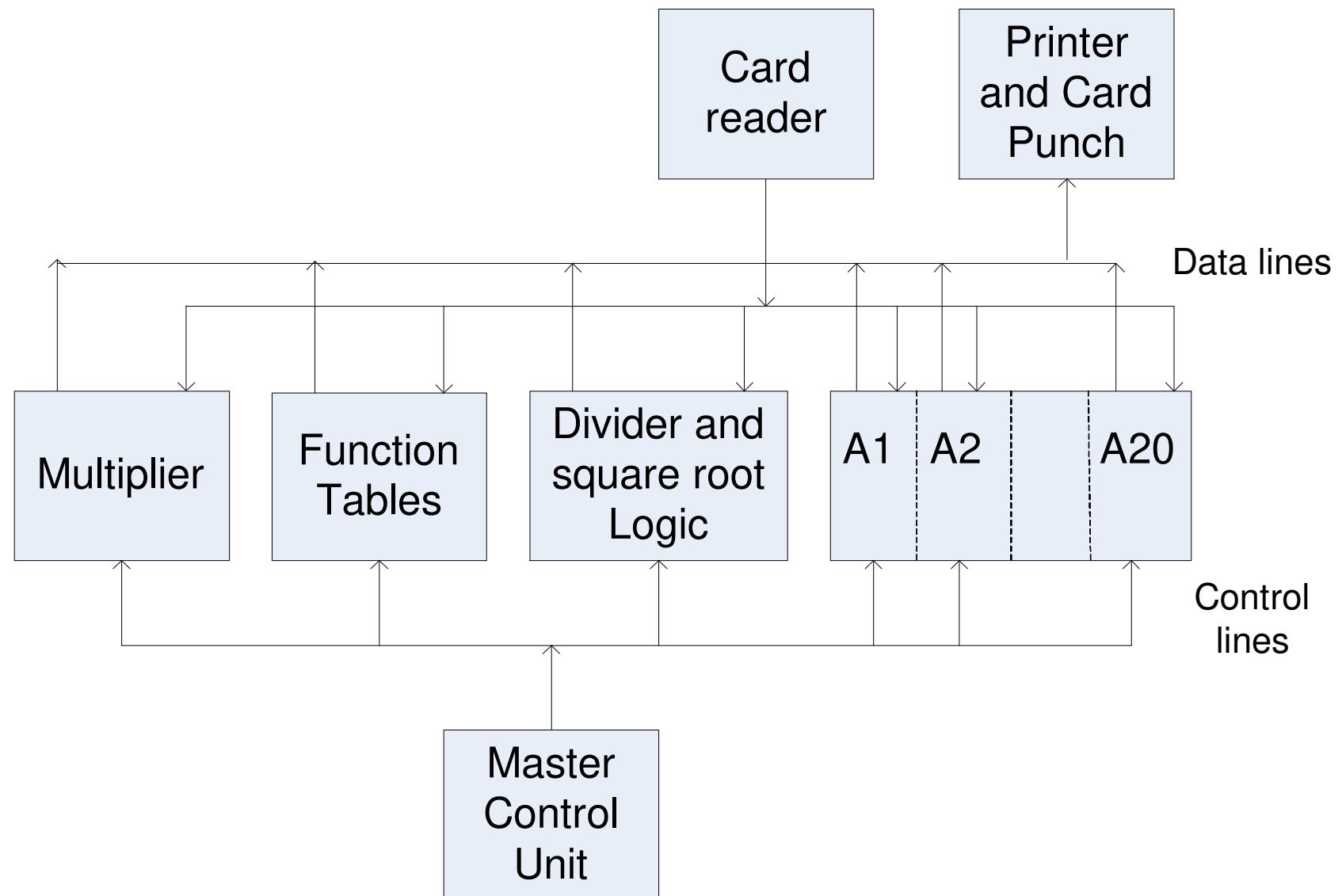
# Atanasoff - Berry



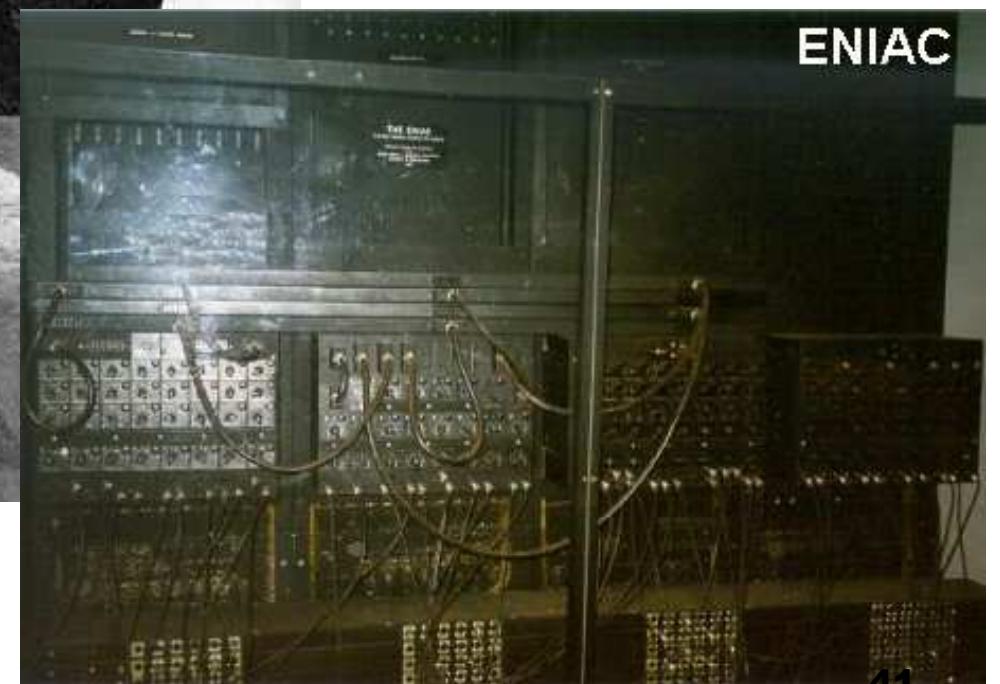
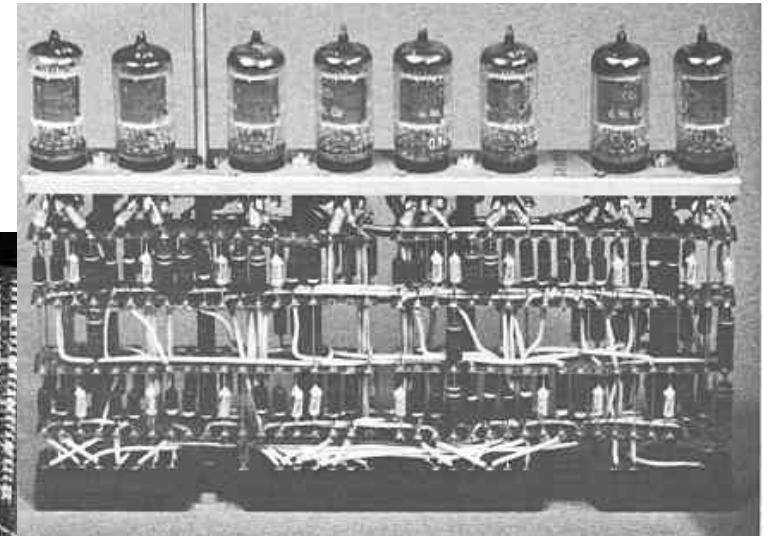
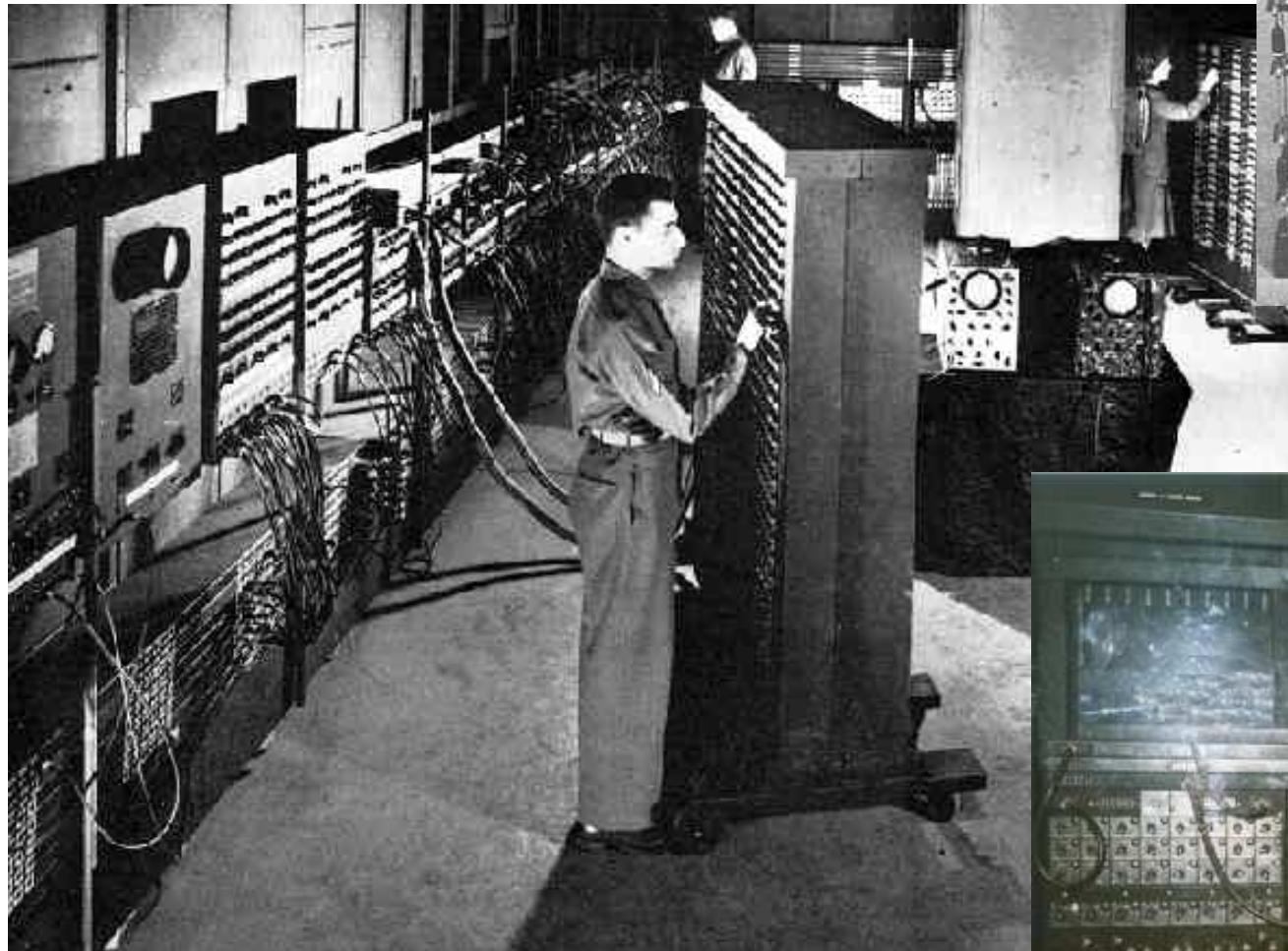
# I. Generáció (1952-ig)

- 1943: **ENIAC**: elektromos numerikus integrátor és kalkulátor (Pennsylvania) Mauchly, Eckert
  - 18000 elektroncső, mechanikus, kapcsolók
  - Gépi szintű programozhatóság, tudományos célokra
  - Összeadás: 3ms
  - 20 ACC reg. – 10 jegyű decimális számra
  - 4 alapművelet + gyökvonás
  - Kártyaolvasó-író
  - Function table: szükséges konstansok tárolása
  - Neumann elvű: közös program/kód és adat

# ENIAC



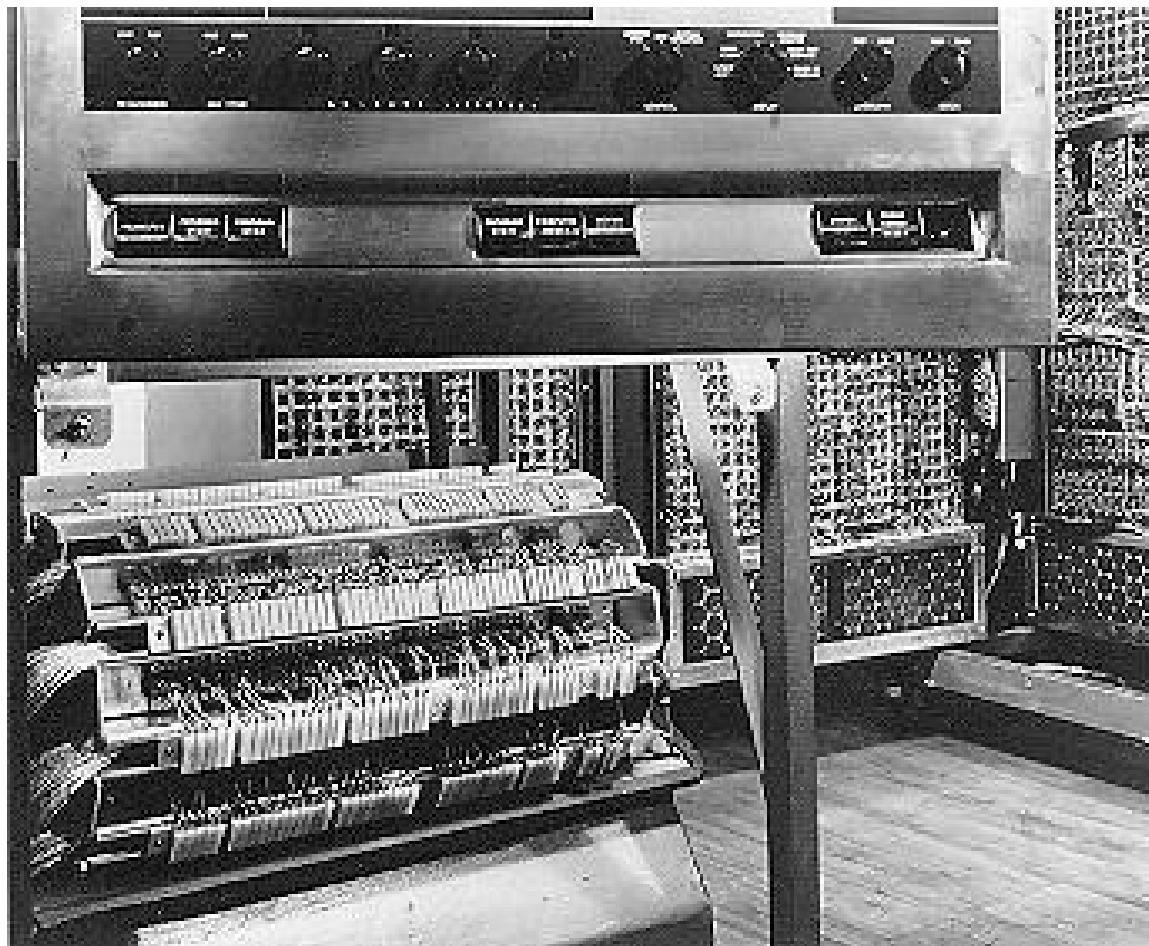
# ENIAC



# I. Generáció (folyt.):

- 1945: **EDVAC** (Electronic Discrete Variable Computer): egyenletmegoldó elektromos szgép.
  - Neumann János – „**von Neumann architektúra**”
  - Tárolt programozás
  - 2-es számrendszer
  - 1K elsődleges + 20K másodlagos tároló
  - soros műveletvégzés: ALU
  - utasítások: aritmetikai, i/o, feltételes elágazás
  - EDVAC tanulmány első teljes kivonata [pdf]
    - <http://www.virtualtravelog.net/wp/wp-content/media/2003-08-TheFirstDraft.pdf>

# EDVAC

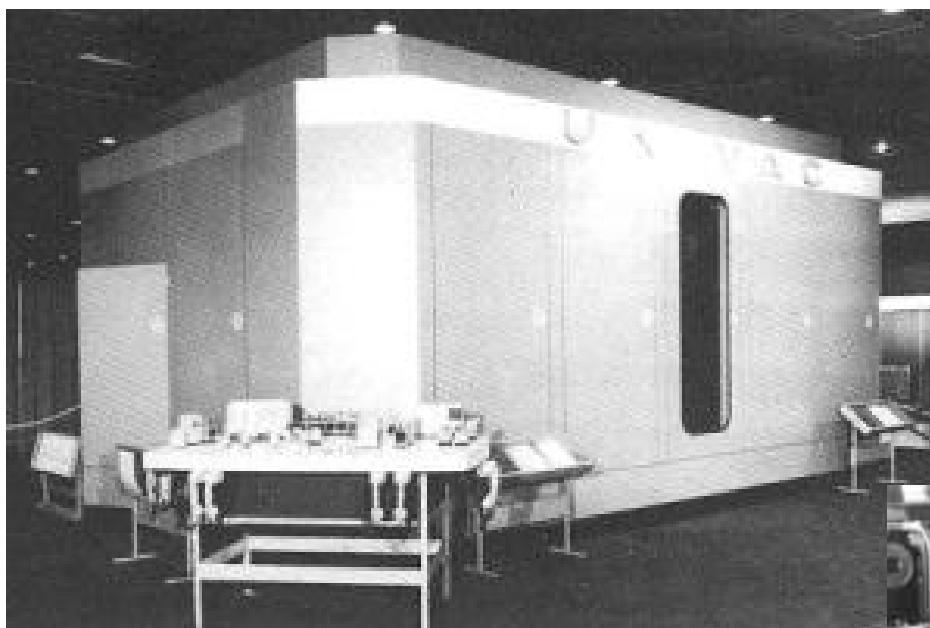


Neumann János

# I. Generáció (folyt.):

- 1951: **UNIVAC I** (UNIVersal Automatic Computer I): üzleti/adminisztratív célokra
  - Mauchly, Eckert tervezte
  - 1951-es népszámlálásra, elnökválasztásra
  - 5200 elektroncső, 125KW fogyasztás, 2.25MHz
  - 1000 szavas memória, (12 bites adat: 11 digit + 1 előjelbit, 2x6 bites utasítás formátum)
  - Összeadás: 525µs, szorzás: 2150µs
  - BCD, paritás ell., hiba ell.

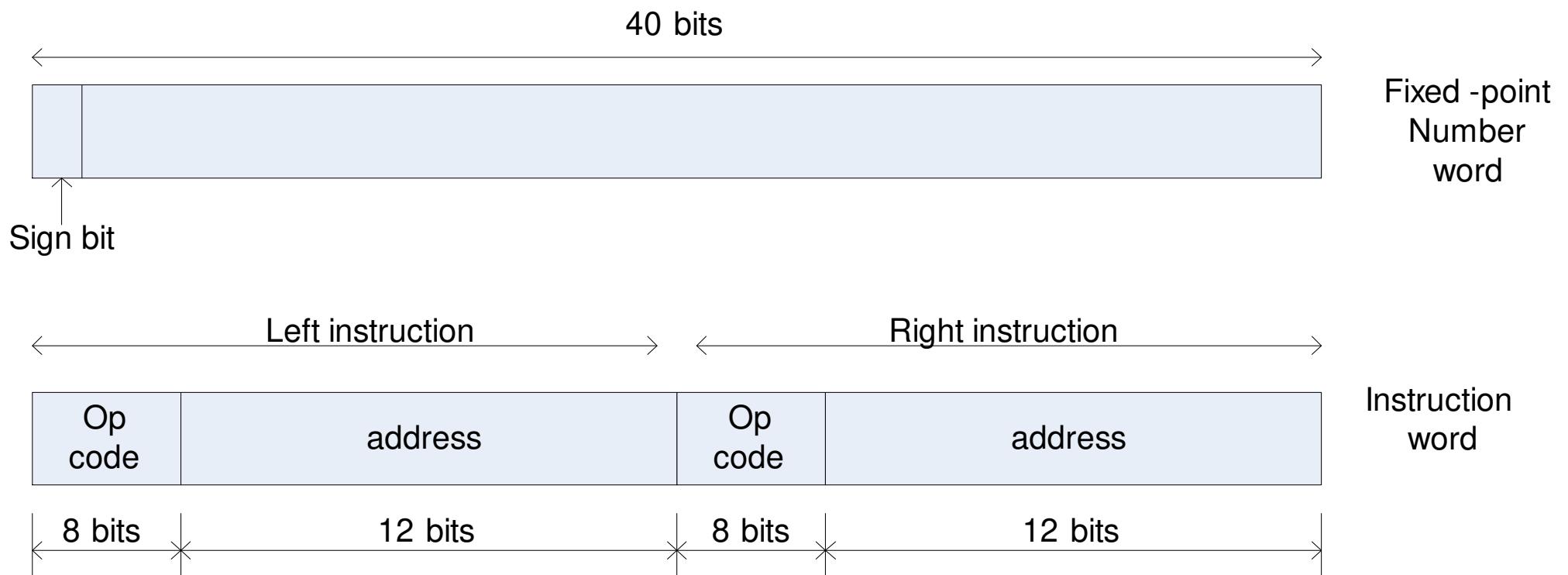
# UNIVAC - I



# I. Generáció (folyt.):

- 1952: **IAS** (Institute of Advanced Studies) Princeton
  - moduláris felépítés: mem, ALU, CU, I/O, ACC
  - köv. végrehajtható utasítás a memóriában a soron következő helyen van
  - egycímű gép – kisebb utasításhossz, (de ACC műveletek)
  - Mem:  $2^{12}=4096$  location
  - párhuzamos feldolgozás!
  - szóhosszúság a feladattípusnak megfelelő numerikus pontosságtól függ
  - Utasítás csoportok: (1.1 táblázat)
    - Adatmozgató, aritmetikai, ugró, feltételes elágazás, címmódosító
  - IAS hátrányai: program struktúráltság – szubrutin hívás (call / return) nem támogatott, nincsenek nemnumerikus adatok

# IAS adat és utasításformátum:



# 1.1 Táblázat: IAS utasítások

## *Data transfer instructions*

<i>Instruction</i>	<i>Description</i>
LDA X	Load ACCUMULATOR with value stored at location X.
LDAM X	Load ACCUMULATOR with negative of value stored at location X.
ABS X	Load ACCUMULATOR with absolute value of number stored at location X.
ABSM X	Load ACCUMULATOR with negative of absolute value of number stored at location X.
LDM X	Load MQ register with value stored at location X.
MQA	Load ACCUMULATOR with value stored in MQ register.
STOR X	The value of the ACCUMULATOR is transferred to location X.

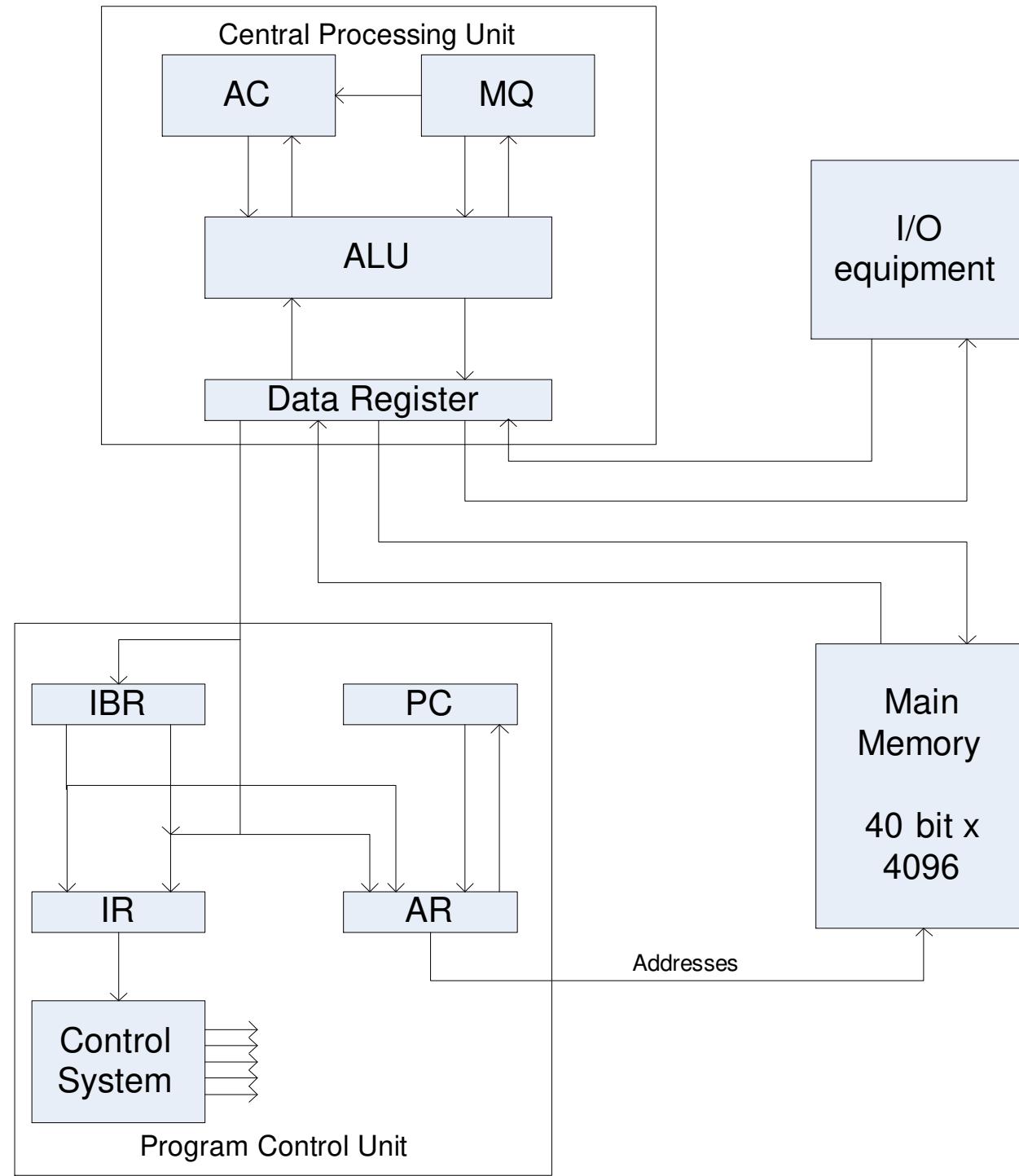
## *Arithmetic instructions*

<i>Instruction</i>	<i>Description</i>
ADD X	Add number stored at location X to ACCUMULATOR.
SUB X	Subtract number stored at location X from ACCUMULATOR.
ADDABS X	Add absolute value of number stored at location X to ACCUMULATOR.
SUBABS X	Subtract absolute value of number stored at location X from ACCUMULATOR.
MULT X	Multiply the number stored in MQ register by value stored in location X, leave 39 most significant bits in ACCUMULATOR, and leave 39 least significant bits in MQ register.
DIV X	Divide value in ACCUMULATOR by value stored at location X; leave remainder in ACCUMULATOR and quotient in MQ register.
LFTSHFT	Multiply the number in the ACCUMULATOR by 2, leaving it there.
RGTSHFT	Divide the number in the ACCUMULATOR by 2, leaving it there.

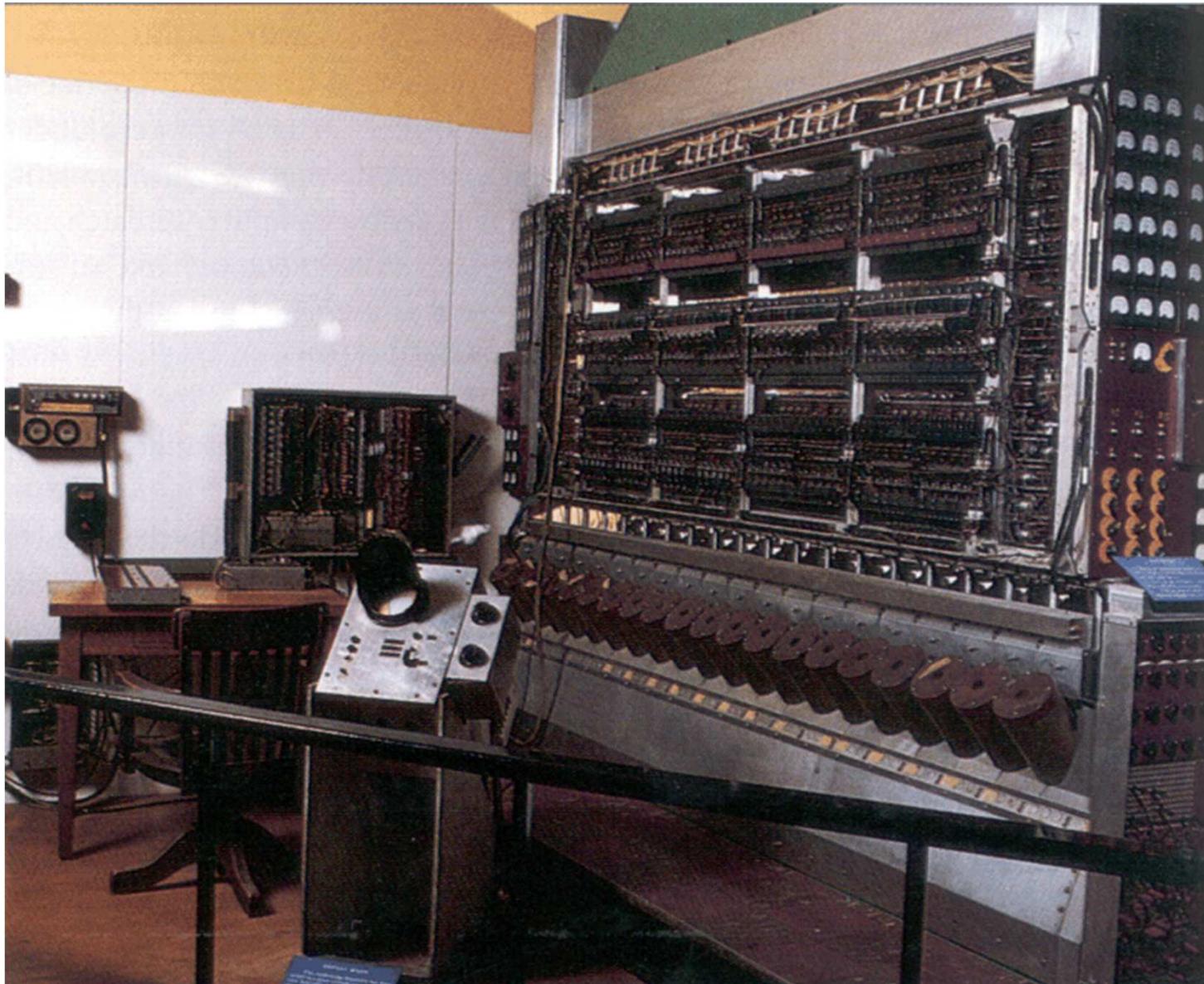
# 1.1 Táblázat: IAS utasítások (folyt.)

<i>Jump instructions</i>	
<i>Instruction</i>	<i>Description</i>
JMPL X	Next instruction to execute is in most significant half of location X.
JMPR X	Next instruction to execute is in least significant half of location X.
<i>Conditional branch instructions</i>	
<i>Instruction</i>	<i>Description</i>
BRANCHL X	If number in ACCUMULATOR is nonnegative, next instruction to execute is in most significant half of location X.
BRANCHR X	If number in ACCUMULATOR is nonnegative, next instruction to execute is in least significant half of location X.
<i>Address modification instructions</i>	
<i>Instruction</i>	<i>Description</i>
CADRL X	The address bits (12 least significant bits) of the most significant half of location X are replaced with the 12 least significant bits of the ACCUMULATOR.
CADRR X	The address bits (12 least significant bits) of the least significant half of location X are replaced with the 12 least significant bits of the ACCUMULATOR.

# IAS



# IAS computer



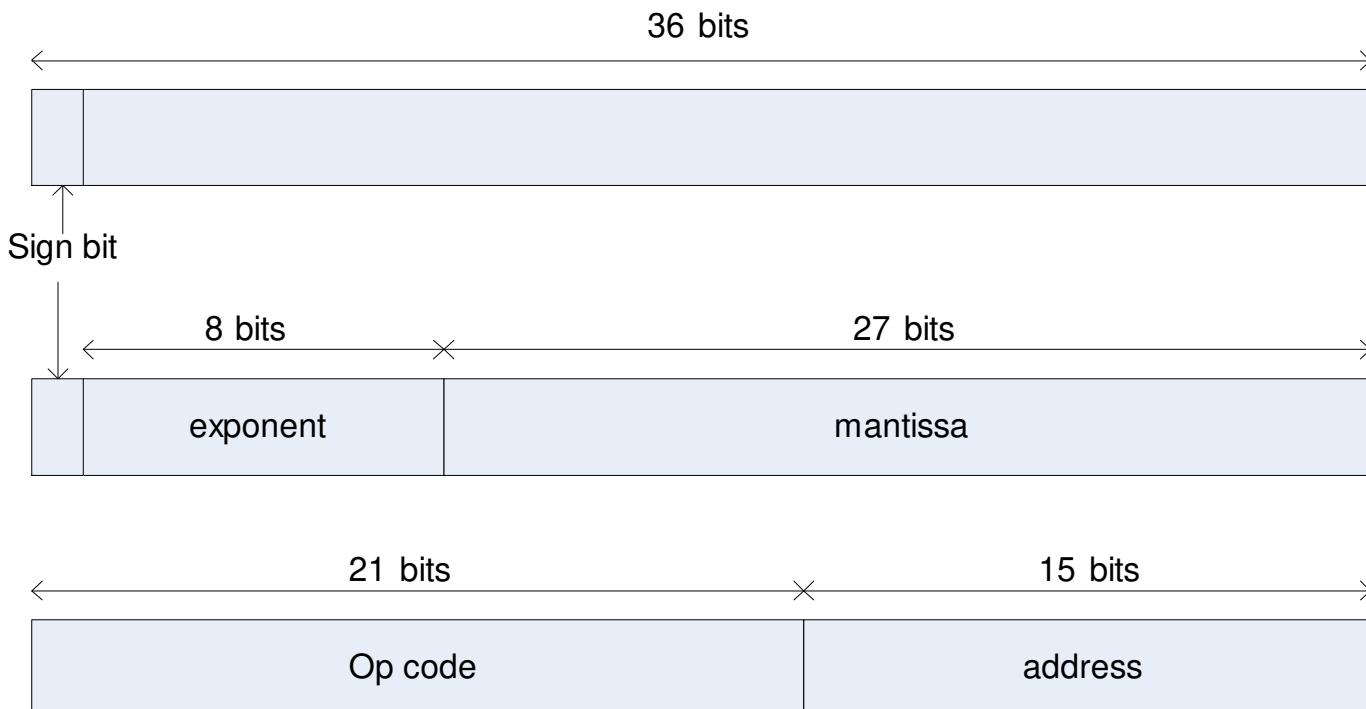
# II. Generáció (1952-63):

- Üzleti célokra (háború vége) IBM
- Tranzisztor! (1940 végétől)
- Csökkenő méret + disszipált telj. / sebesség nő
- Core memóriák – megbízható, gyors
- Lebegő pontos számok, utasítások
- Új módszer az operandus helyének azonosítására
- FORTRAN, ALGOL, COBOL nyelvek
- I/O processzorok: CPU tehermentesítése
- Batch programozás, könyvtári függvények, compilerek

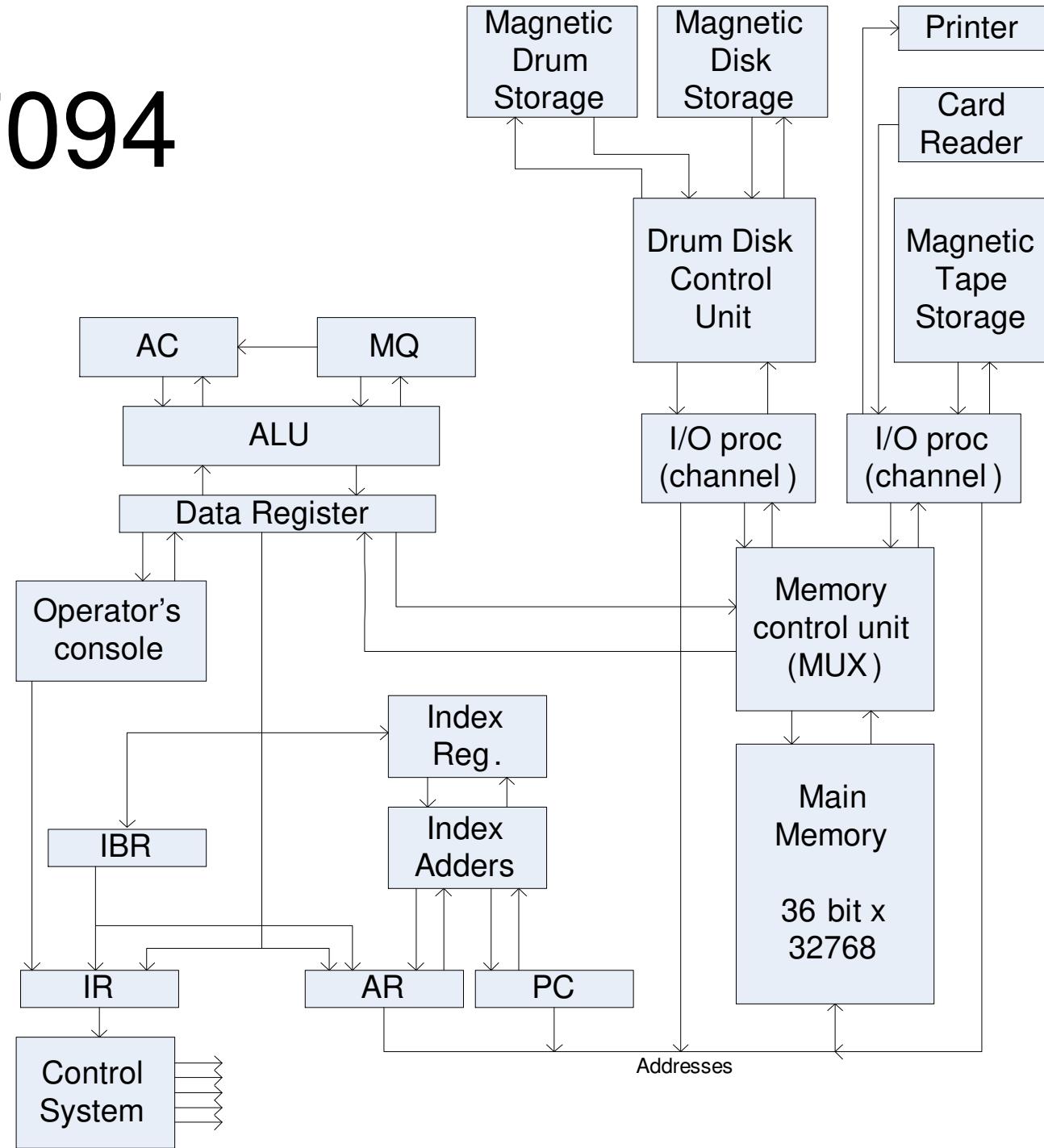
# II. Generáció (folyt.):

## ■ IBM 709x

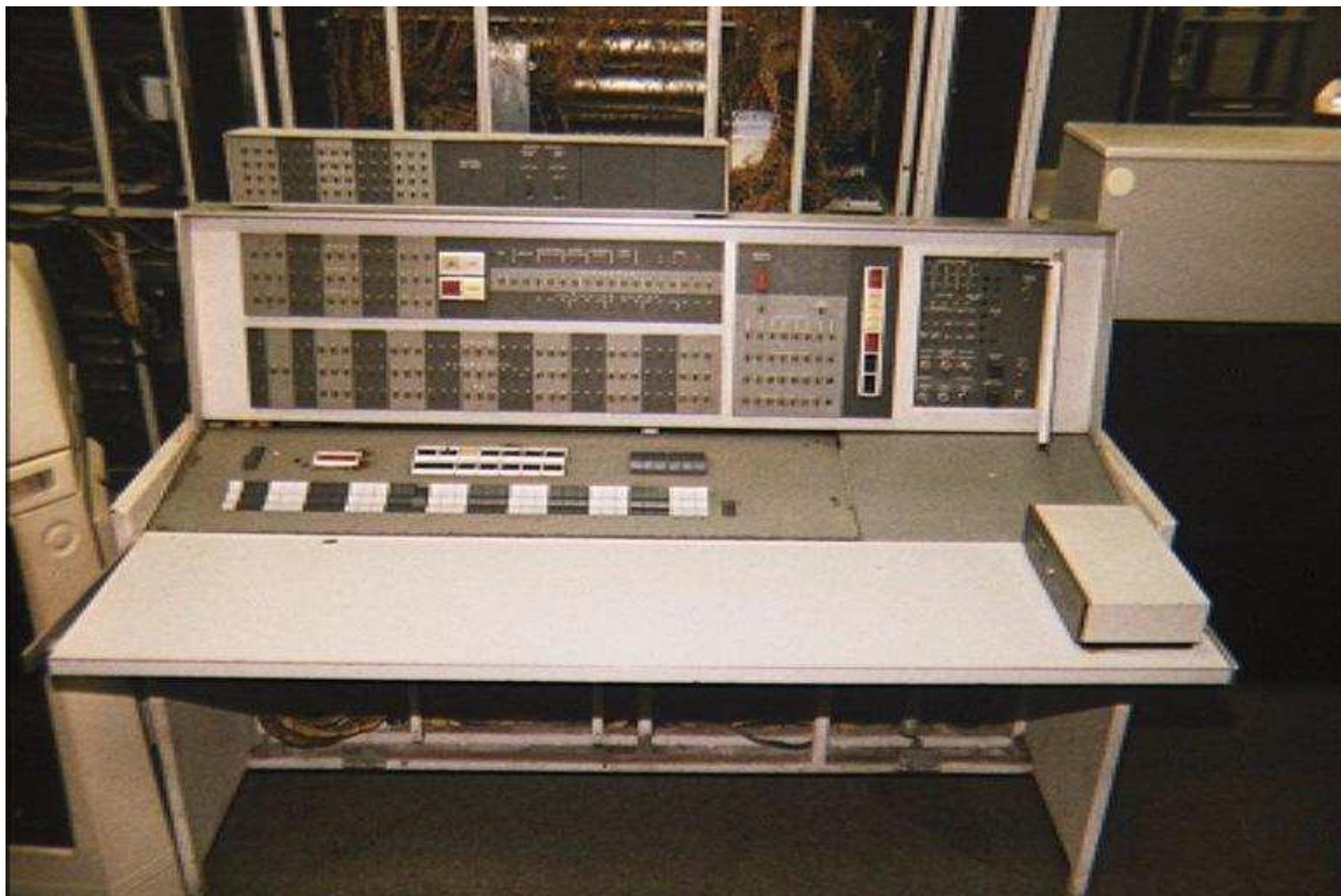
- egycímű gép ( $AR \leftarrow PC+IR$  tartalma)
- I/O processzorok
- 36 bites utasítás, és adat (72 bites adatút)



# IBM 7094



# IBM 7094



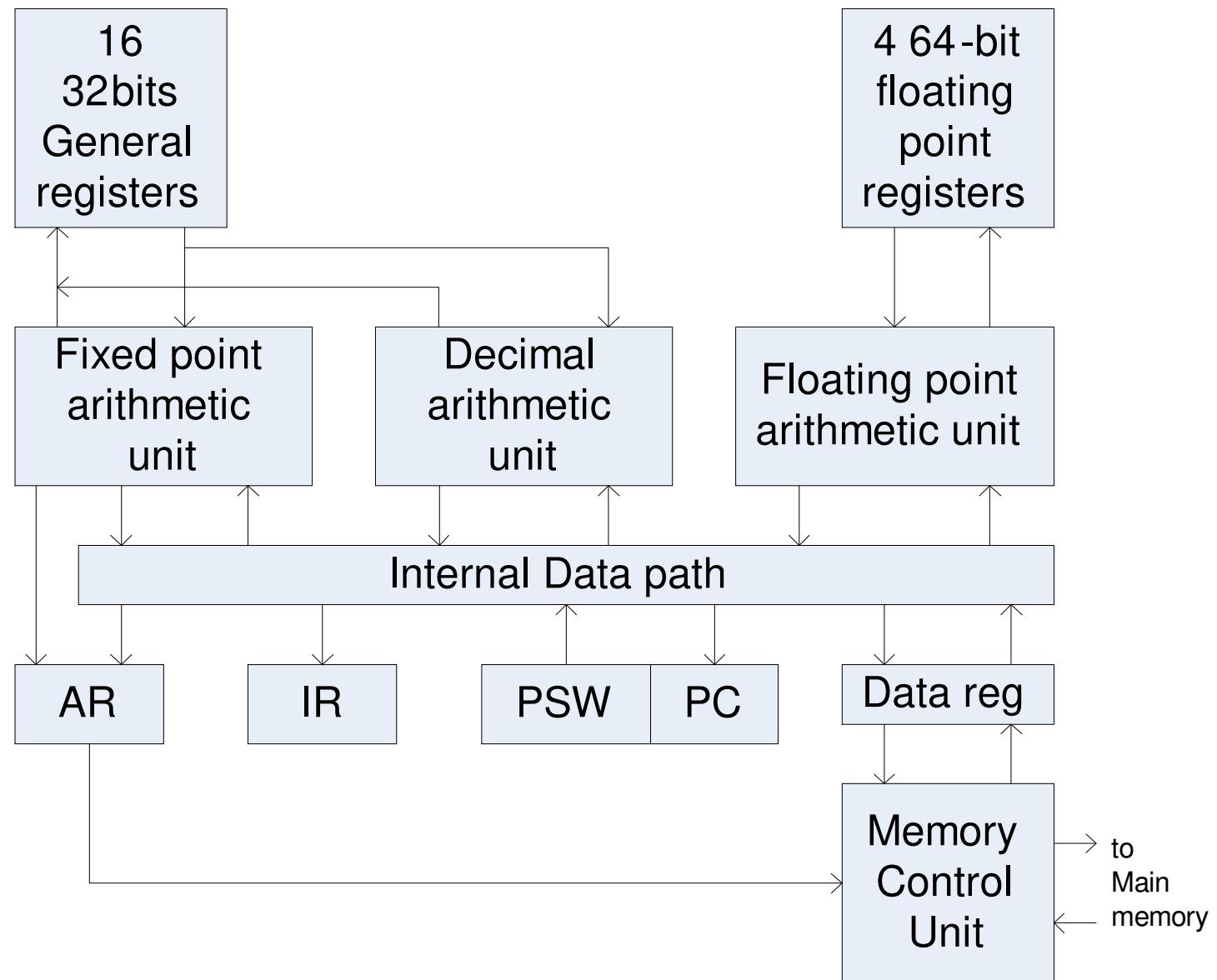
# III. Generáció (1962-75):

- IC technológia
- 1965. Gordon-Moore trv: Mikro-minimalizáció
- Félvezető memóriák
- Mikroprogramozás (Wilkes 1951)
- Multiprogramozás: „time-sharing”
- Operációs Rendszerek megjelenése
- Pipeline - parallel működés
- Numerikus programozás: vektorműveletek

# IBM 360

- első sorozatban gyártott (gépcsalád): fogyasztói célok szerinti kategóriák
- azonos utasítás készletek
- I/O csatornák seb. szerint (selector, MUX)
- 32 bites utasítások
- 8x4 bites BCD számjegyeket tárol
- 4x8 bit karakter! tárolására
- Integer / fix-point / floating-point számokat is kezel
- 16 db 32 bites ált.célú regiszter (adatok, címek)
- 4 db 64 bites lebegőpontos műveleti reg.
- Interaktív rendszer
- Virtuális memóriakezelés lehetősége
- PSW: státuszjelző regiszter (flag)

# IBM 360 utasítás készlet:



# IBM 360

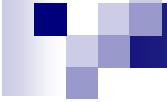


# IV. Generáció (1974 - ?):

- IC alapú technológia: komplexitás-méret
- Cache memóriák
- Virtuális memória rendszerek
- SoC: System On a Chip
  - Motorola 68000 – 32 bites proc.
  - ALU, Regiszterek, virtuális memória egy chipen
- 4, majd 16 ... megabites memóriák
- PC: személyi számítógépek megjelenése
- Száloptika ⇒ hálózatok (INTERNET)

# V. Generáció (napjainkban):

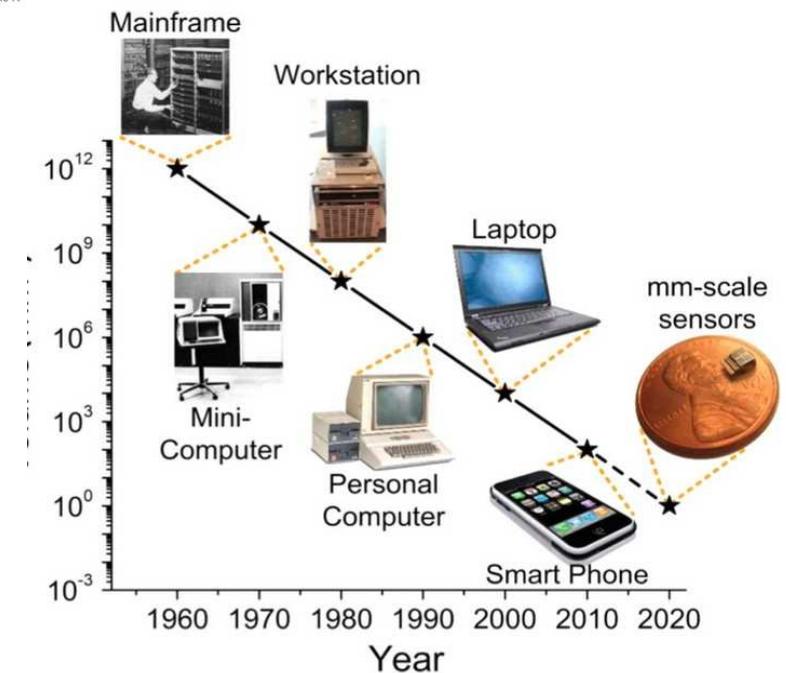
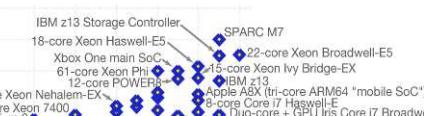
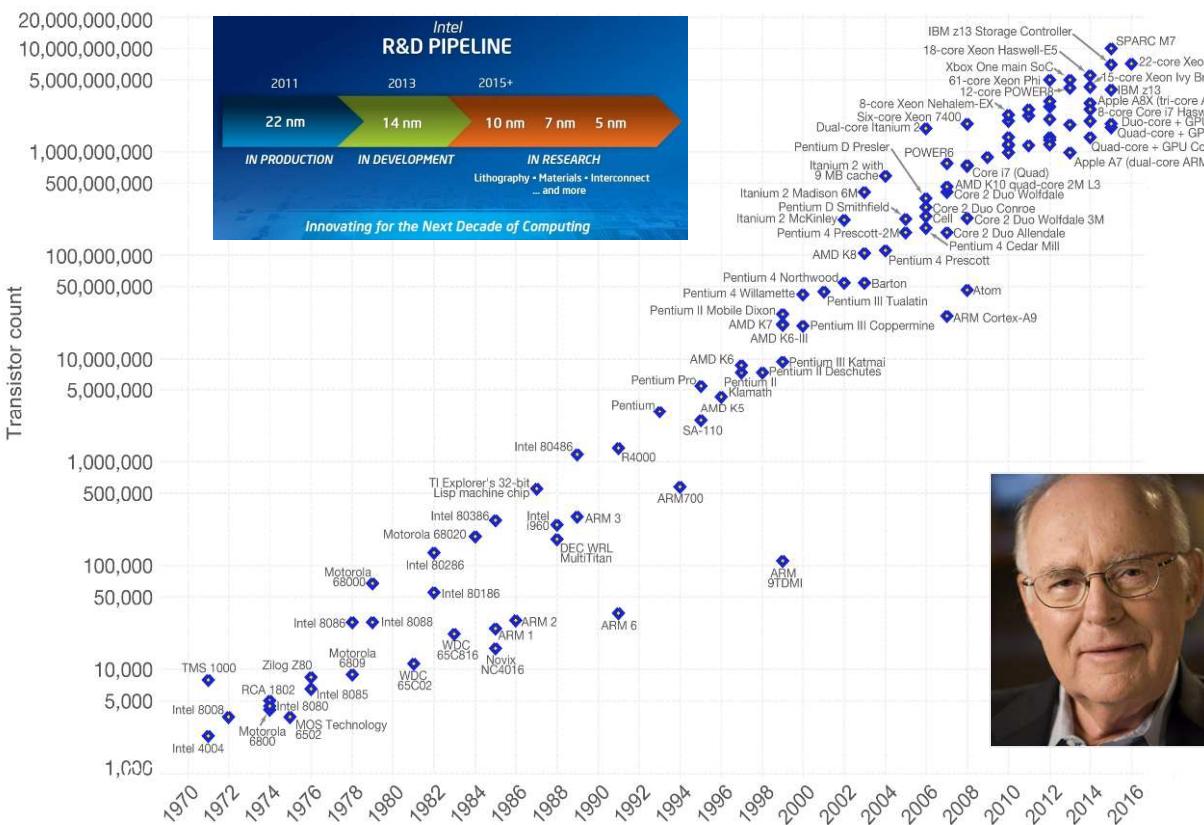
- Ember-gép interakció (HCI)
  - IPAR 4.0 ...
- Felhasználóbarát szemlélet
- Ergonómia
- Mesterséges intelligencia (AI)
  - Deep learning
- Természetes nyelvi környezet:  
fejlesztőeszközök (development tools)



# Hol tart jelenleg a technológia?

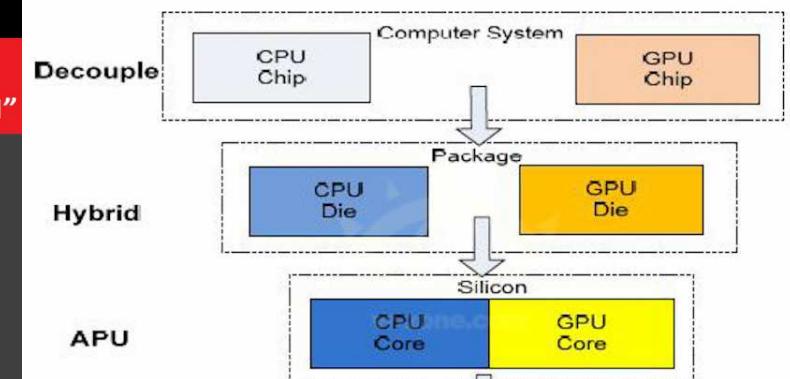
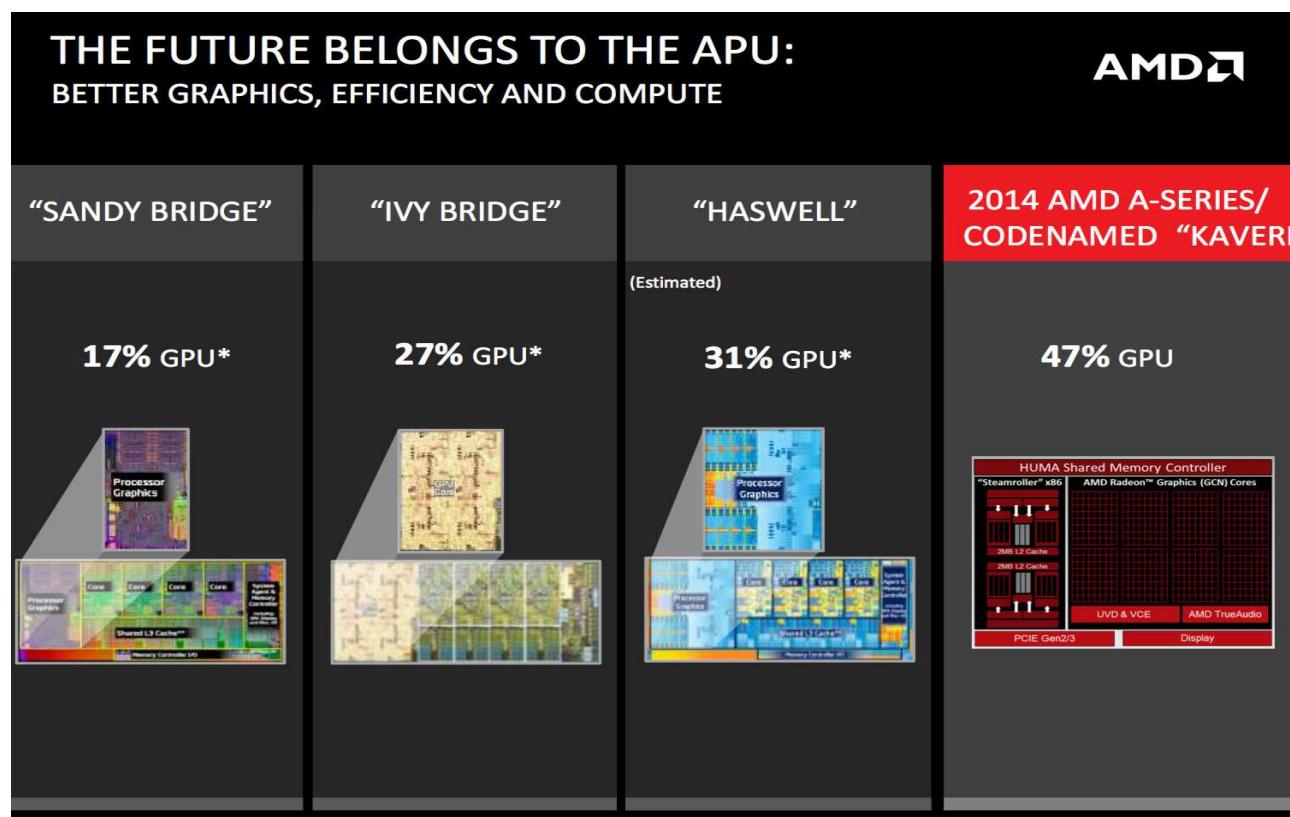
# Technológiai fejlődés

- **Moore törvénye (1965):** 1 (2, ma 3) évente adott Si felületegységre eső tranzisztorszám duplázódása
- **Bell törvénye (1972):** számítógépek méretének fokozatos csökkenése (~10 évente új számítógép osztályok, platformok megjelenése)

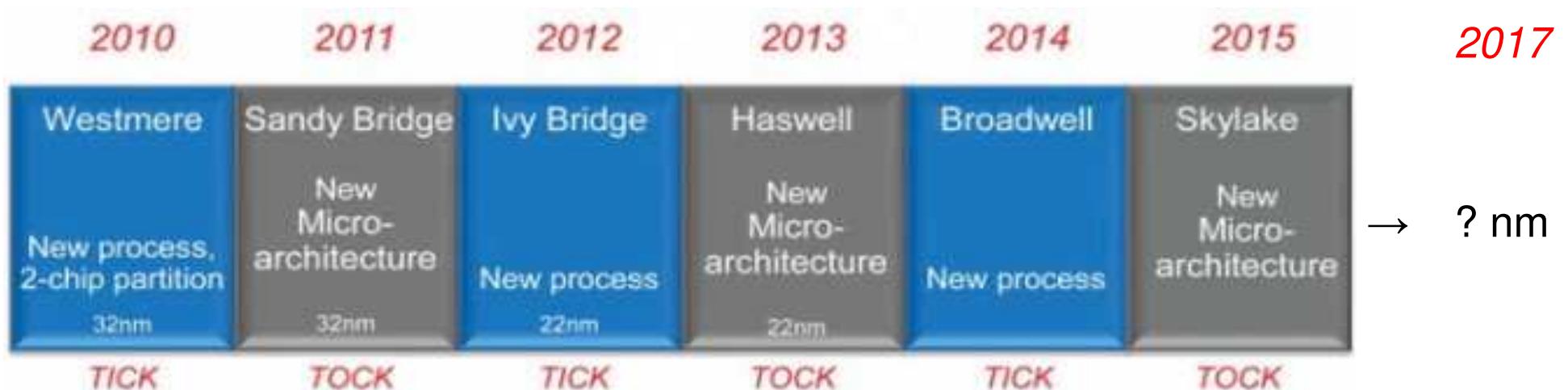


# CPU + GPU integráció = APU

## APU (Accelerator Processor Unit)



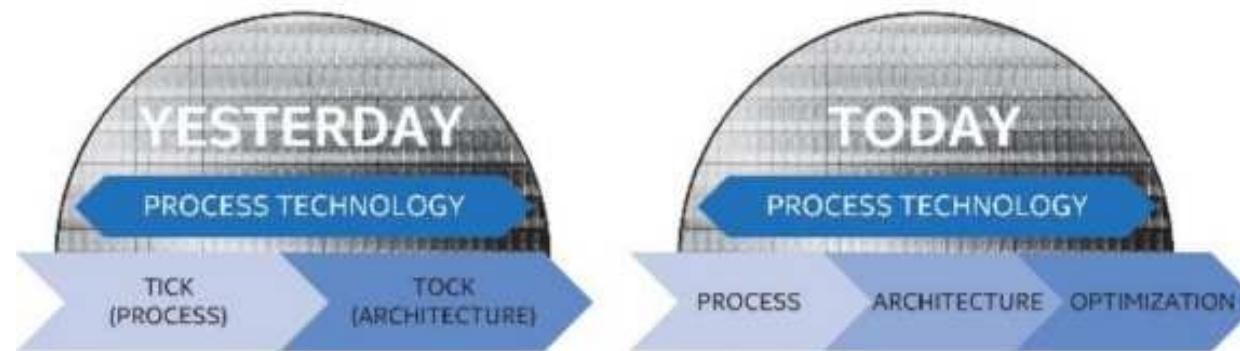
# Intel „tikk-takk” stratégiája:



Megdőlt ez a „tikk-takk” stratégia (2016):

- 2 évente új gyártástechnológiára váltottak (ez jelenti a nagyobb problémát!)  $22 \rightarrow 14 \rightarrow 10$  nm.
- 2 évente új mikroarchitektúra jelent meg
- Intel Kaby Lake: 7. gen Intel Core architektúra (még az utolsó tock fázisban készült, 14nm)

# Intel „PAO” stratégia:



## PAO – Process – Arcitecture – Optimization (2016-tól)

- 3 évente új gyártástechnológiára váltottak (ez jelenti a nagyobb problémát!) → 10 nm → ... ?
- 3 évente új mikroarchitektúra jelent meg
- 3 évente az architektúra optimalizálása
- Intel Kaby Lake: 7. gen Intel Core architektúra (még az utolsó tock fázisban készült)

# Szuperszámítógépek

## ■ Első szuperszámítógépek

- LARC: (Livermore – US) atom-kutatásokra (1960)
- IBM 7030 / Strech (1961)

## ■ MA (2024. február): [www.top500.org](http://www.top500.org)

- 1.) **Frontier – HPE Cray EX235, Oak Ridge Laboratory, USA**
  - 8.7 millió ! processzor mag (AMD Epyc 3rd gen 64C 2GHz), P: 22 700 kWh!!
  - **~1.2 ExaFLOPs = 1 194 PetaFLOP/s teljesítmény!!**
- 2.) **Aurora – Argonne National Laboratory, USA**
  - 4.74 millió ! processzor mag (Intel Xeon 9470 52C 2.4GHz), P: 24 600 kWh!!
  - **585 PetaFLOP/s teljesítmény!!**
- 3.) **Eagle – Microsoft Azure, USA**
  - 1.23 millió ! processzor mag (Intel Xeon Platinum 8080 48C 2 GHz), P: -
  - **561 PetaFLOP/s teljesítmény!!**
- 5.) **LUMI – HPE Cray EX235: EuroHPC, Finnország!**
  - 2.7 millió processzor mag (AMD Epyc 3rd gen 64C 2GHz), 2800 TB memória, P: 7 107 kWh!!
  - **379.2 PetaFLOP/s teljesítmény !**
- ...
  - x.) **IBM Roadrunner BladeCenter QS22/LS21 Cluster, (LANL, Los Alamos., US) - 2009**
    - 129 600 processzor magos rendszer (PowerXCell 8i 3.2 GHz ), 73 728 GB memória (N/A)
    - **1.105 millió GFLOPs teljesítmény! (elsőként ~ 1 PetaFLOPs sebességtartomány átlépése)**

## ■ További lehetőségek: FDE – parallelizmus

- átlapolt végrehajtás (látszólagos) – pipe-line, vagy IPL (utasítás szintű párhuzamosítás – pl. szuperskalár processzorok): párhuzamosítás egyetlen processzoron belül
- teljesen párhuzamos végrehajtás (több processzor) – pl. CELL BE
- heterogén több-magos (multi-core/many-core) rendszerek (pl. mai APU-k)

# Szuperszámítógép

## ■ #199. Magyarország (2023. január)

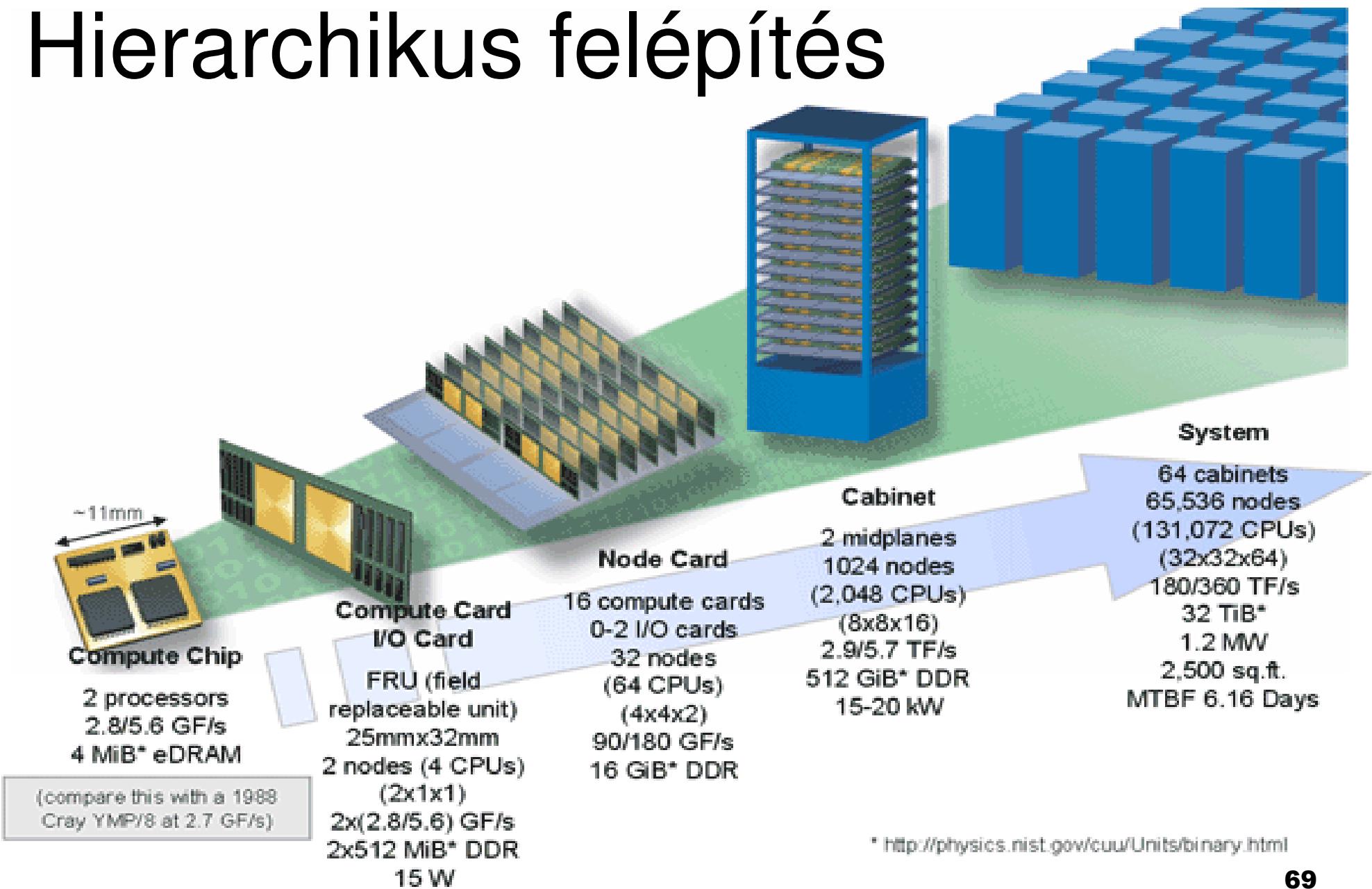
- KIFÜ-HPC: „Komondor”, Debrecen
- **5 PetaFLOP/s**, ~5 mrd Ft, 1.3 MWh
- HPE Cray EX, AMD Epyc 7763 64C 2.5GHz
- 28 768 CPU mag
- NVIDIA A100 SXM4 40 GB
- <https://hpc.kifu.hu/>



## ■ Magyarország (2014)

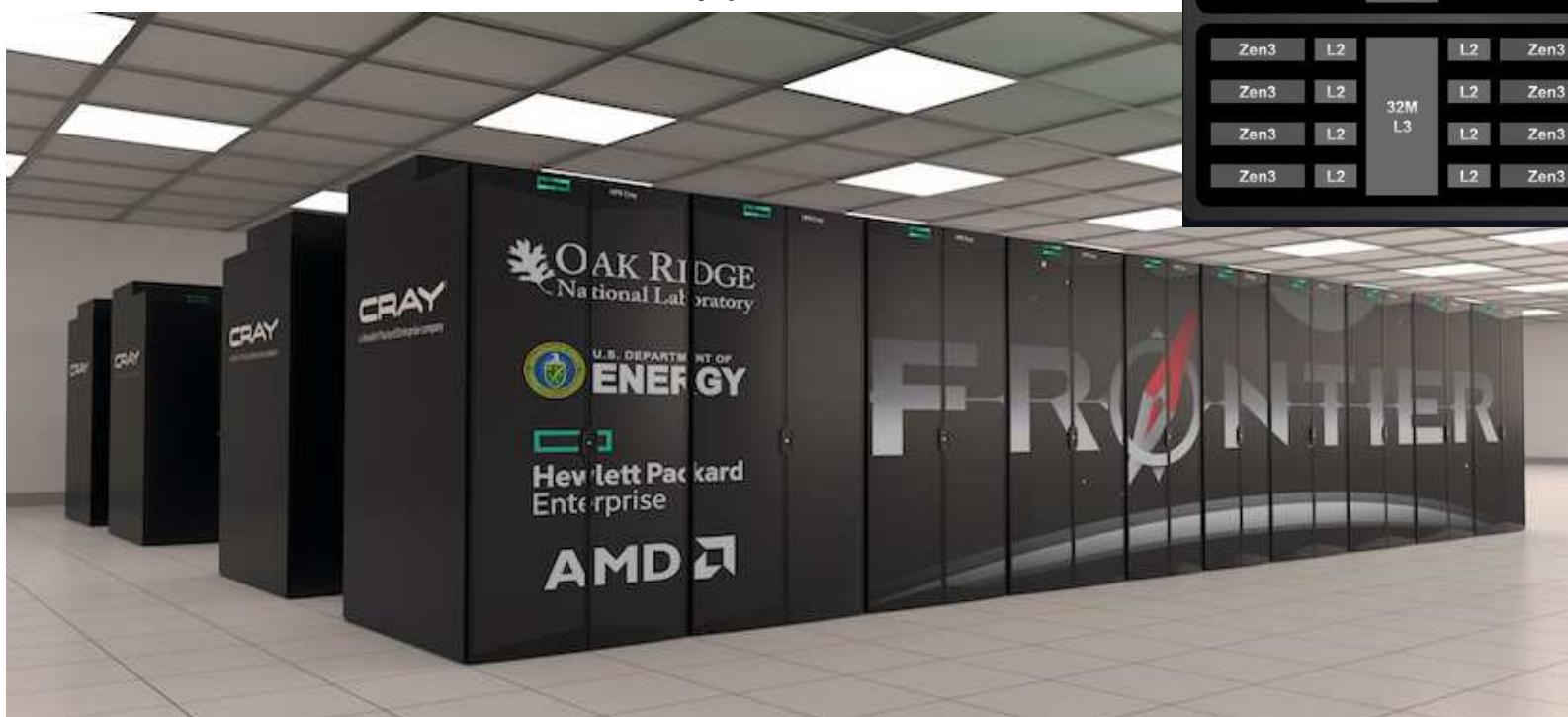
- # 307. Leó NIIFI-Debrecen - Cluster Platform SL250s Gen8, Intel Xeon E5-2650v2 8C 2.6GHz, Infiniband FDR, NVIDIA K20x/K40
- **253.6 TFLOPs** teljesítmény, 122 KWh
- HPE rendszer
- 4 890 CPU mag, 10 TB memória
- <http://www.niif.hu>

# Szuperszámítógépek – Hierarchikus felépítés



# #1 Frontier (USA) 1.2 ExaFLOPs

AMD Epyc 3rd gen 64 Cores



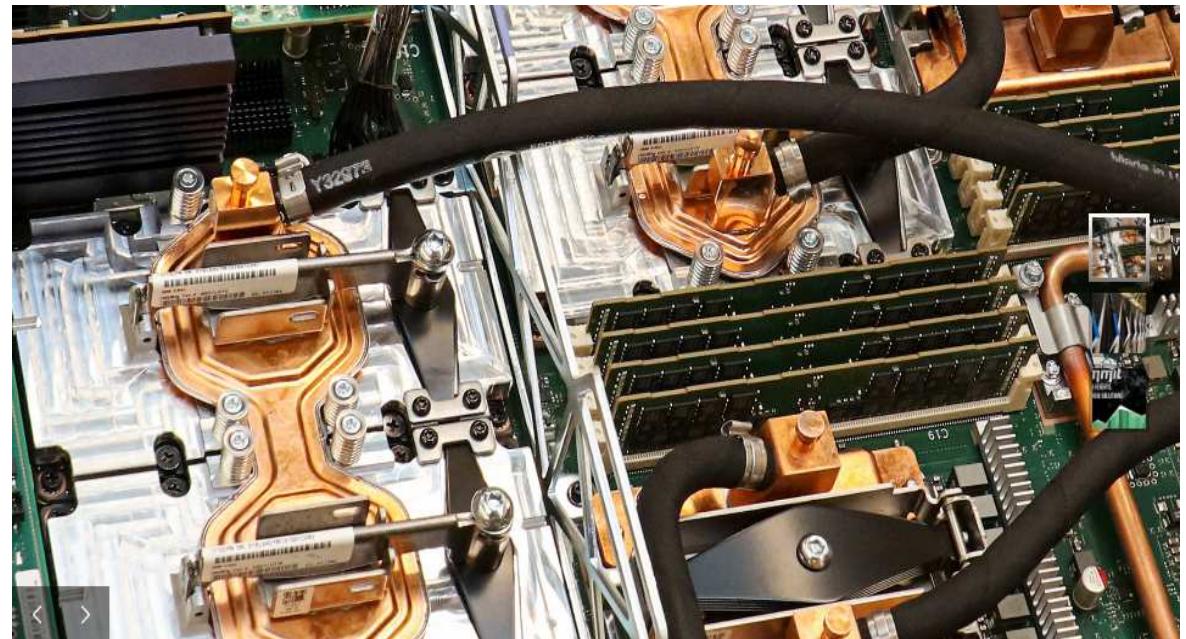
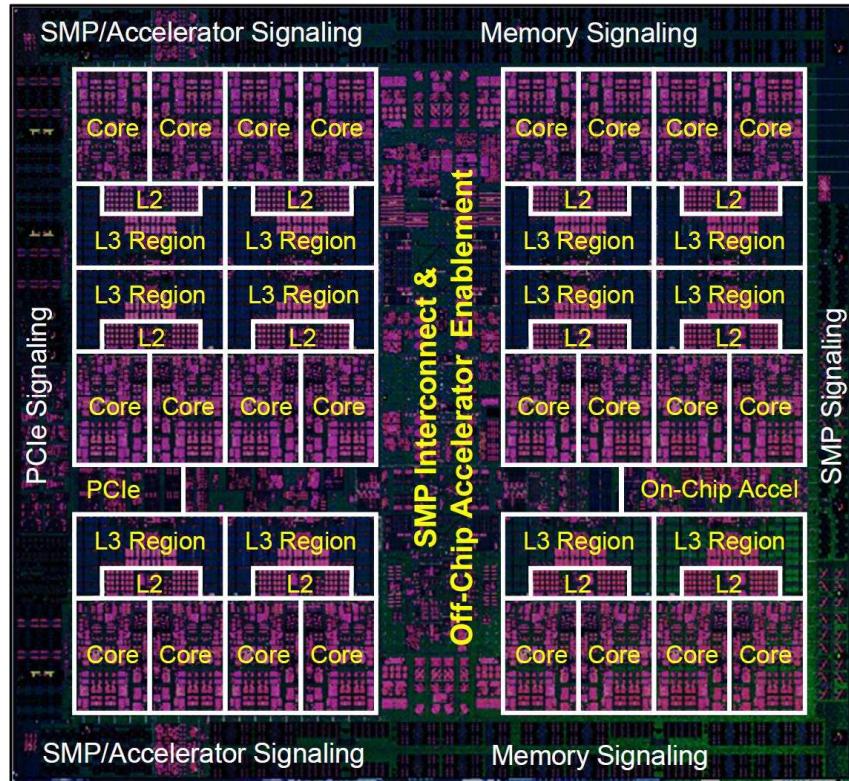
# #5 LUMI (Finnország)



AMD Epyc 3rd gen 64 Cores

# #7 Summit (USA)

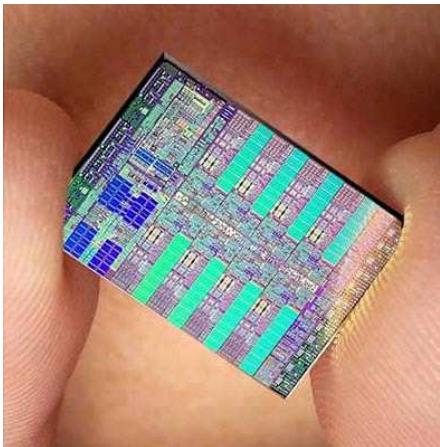
IBM POWER9 22Cores @ 3.1 GHz

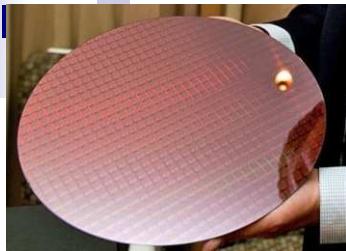


# IBM Roadrunner supercomputer

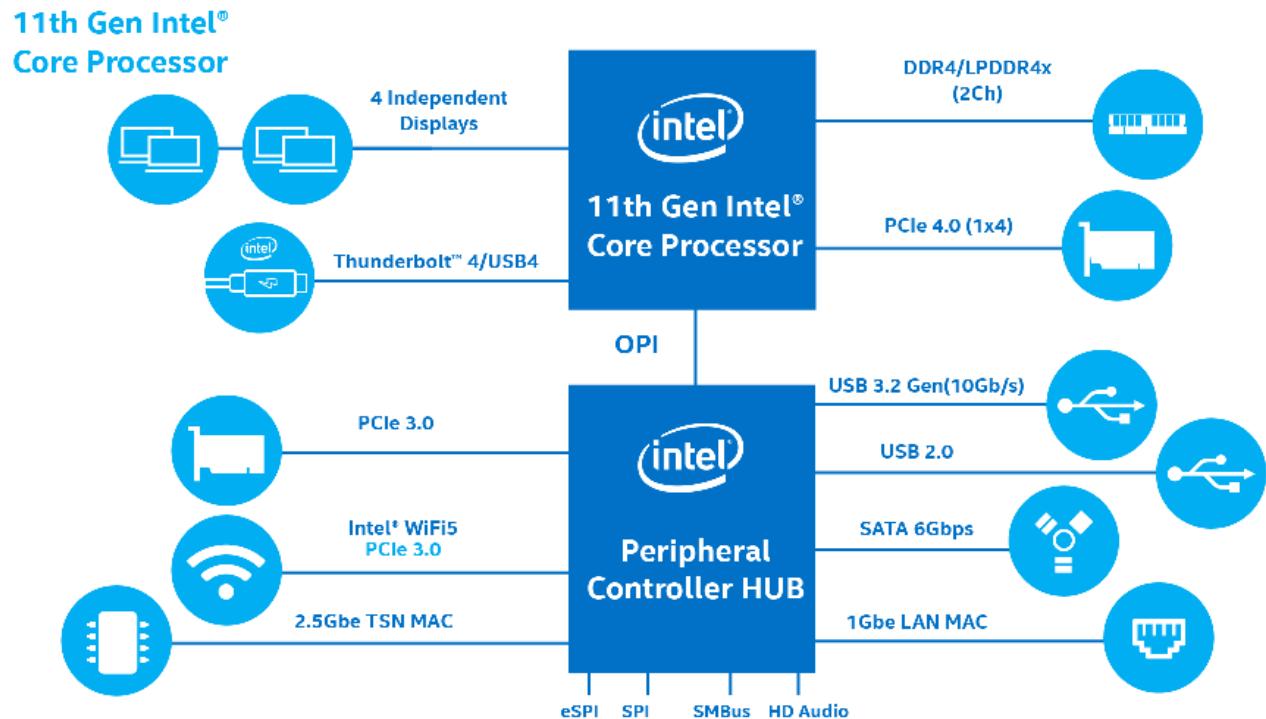
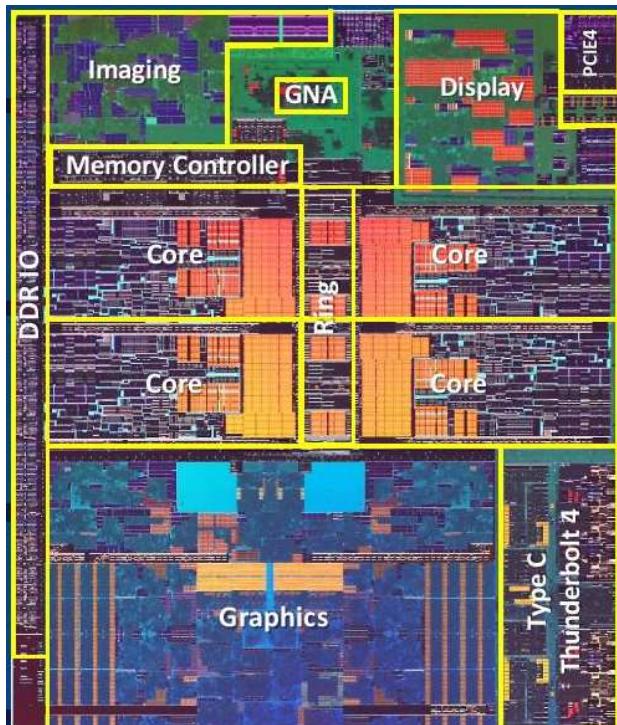


1 PetaFlops (2009): #1





# Intel „Lake” generációk



2017. **Kaby Lake**, 7.generációs APU – teljes „brand” paletta (még **TIC-TOC** stratégia: 2/4/8 mag, 14nm, 4+ GHz, 30-95 W, 1151 lábú tokozás)

2017. **Coffee Lake**, 8.gen., már **PAO**, 14nm, 4/6 mag, 65-95W, 4+ GHz

2018. **Coffee Lake Refresh**, 9. gen. (14 nm, 8 mag, 3.6 GHz, 95W)

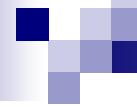
2019. **Ice Lake**, 10.gen (10 nm, 4 mag, 4+ GHz )

2020. **Tiger Lake**, 11.gen (10+ nm, 4 mag, Intel Iris Xe grafikus vezérlők, 15-30 W)

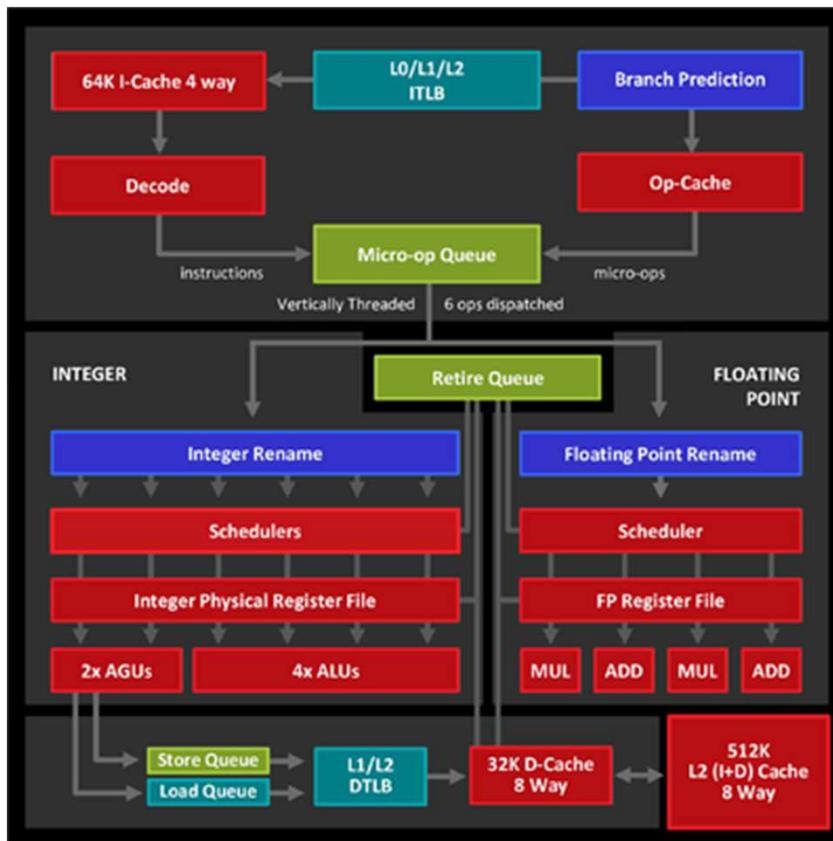
2022. **Alder Lake**, 12.gen (Intel7 nm), új hibrid CPU: P-cores, E-cores

2023. **Raptor Lake**, 13.gen (7 nm?), hibrid CPU → Raptor Lake Refresh, 14.gen (32 mag, 6GHz)

2024. **Meteor Lake** – 14. gen „Core Ultra-5-7-9”, (7nm)



# AMD Ryzen (ZEN 1/2/3/4 gen)



**APU ≠ CPU**



2017: **AMD Ryzen 3/5/7, 1.gen ZEN** architektúra, 4/6/8...16 mag – 8/12/16...32 szál, 14nm, 3-4.2 GHz, 5 milliárd tranzisztor, L3 \$: 8-16-32 MB, TDP: 65W – 100 W...180W, 1331 lábú tokozás), \$100-1000

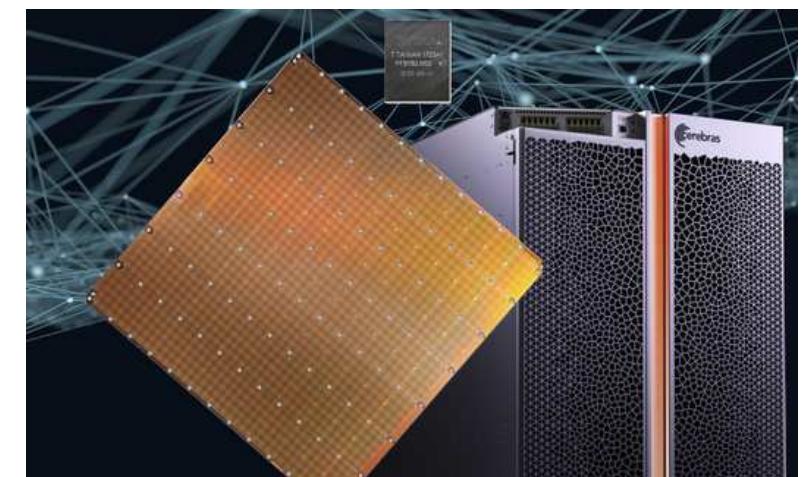
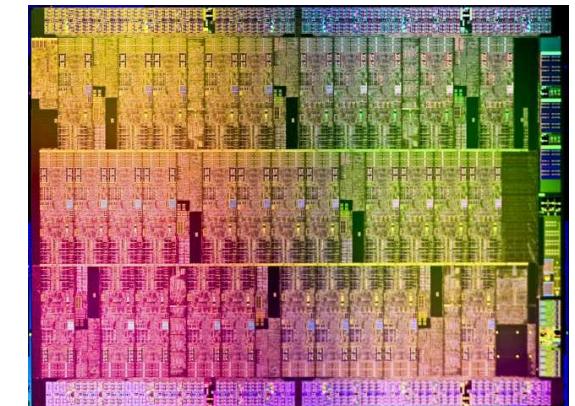
2019: **AMD Ryzen-2** 3/5/7/9 (2X00/3X00) **2. gen ZEN-2** architektúra (akár max 64 mag/128 szál, L3\$: 288MB!, 3-5 GHz).

2020: **AMD Ryzen-3** 5/7/9 (5x00X) **3. gen ZEN-3** arch, 6/12 – 16/32 mag/szál ( L3\$, 65W-100W TPD), 3.5-4.9 GHz

2022: **AMD Ryzen-4** 5/7/9 (7x000X) **4. gen ZEN-4** arch 16/32 mag/szál 4.6-5.6 GHz (TPD 170W!)

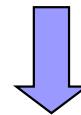
# Many-integrated cores

- Intel Xeon Phi 3100/5110/7120 (Knights Corner, Hill, Mill ...)
  - 1-1.2 TFLOPs, 12 mag, 22 nm, max 320 GB/sec memória sávszélesség, 300 W, 2000-4000 \$
  - Tianhe-2 (2013) Top 1.
- Intel Knights Landing:  
Xeon Phi „v2” (2015)
  - 14 nm, 3 TFLOPs **72 magos** (Intel Atom), 500 GB/sec memória. 200 W
- 2021 - Világ „legnagyobb processzora”
  - **Cerebras-CS1** „óriás chip”:
    - 1 200 milliárd tranzisztor (21 cm<sup>2</sup>!! )
    - 400.000 optimalizált AI mag, 20 KWh
  - Cerebras-CS2 ?, 7nm (46 cm<sup>2</sup>!)
    - 2 500 milliárd tranzisztor
    - 850 000 AI core

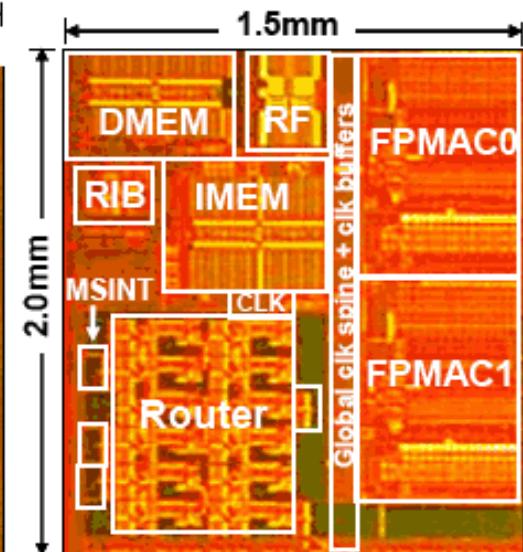
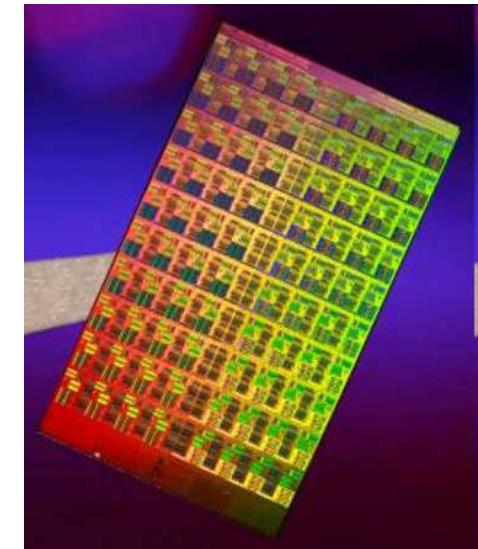
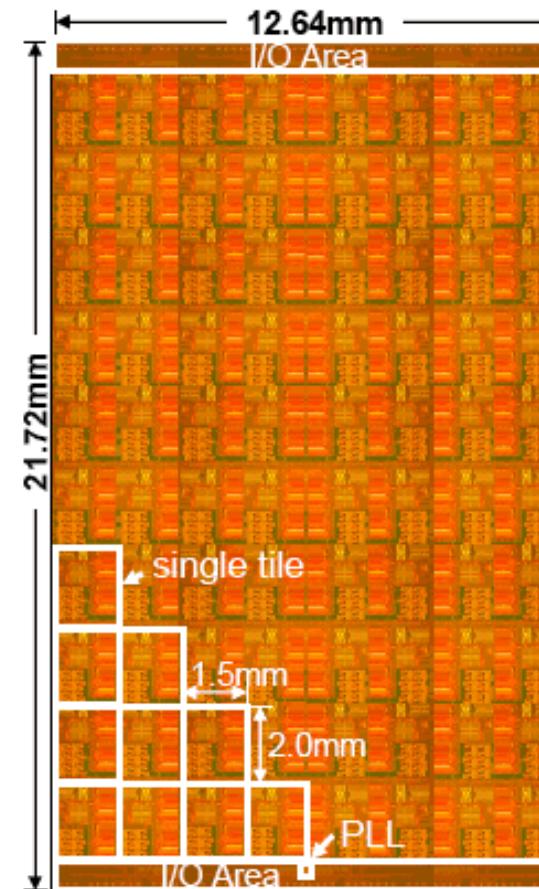
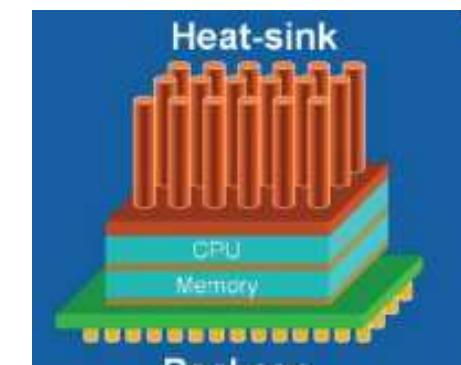


# Intel Nehalem-EX: 80 mag

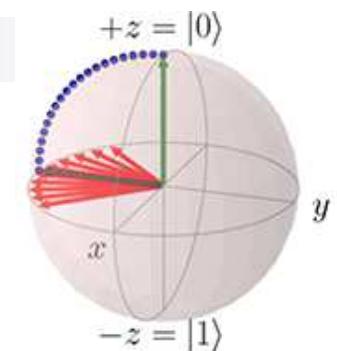
- ISSC'2007
- Polaris: 80 mag
  - 65 nm technológia
  - 3D rétegszerkezet
- 1 TeraFLOPs.
- 4 - 5.1 GHz
  - 100 – 175 W



- Intel Core i7 EE 980x
  - 32nm
  - 3.3 GHz
  - 6 mag / 12 szál
  - 2.2 milliárd tr.



Technology	65nm CMOS Process
Interconnect	1 poly, 8 metal (Cu)
Transistors	100 Million
Die Area	275mm <sup>2</sup>
Tile area	3mm <sup>2</sup>
Package	1248 pin LGA774 layers, 343 signal pins

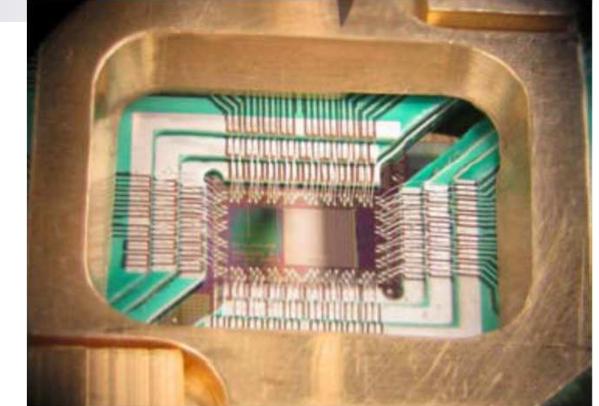


# Más alternatíva: Kvantumszámítógépek

- A *hagyományos számítógépeknél* a tranzisztorok számának duplázódása –  $2x$ -esére növeli a gép teljesítményét (~Moore-törvény szerint *lineáris* növekedés)
- A **kvantumszámítógépeknél** minden egyes kvantumbit (qubit) hozzáadása megduplázza (*hatványozza*) a gép teljesítményét ()!
  - „**qubit**” = **kvantumbit**, a kvantum-számítás alapegysége, amellyel Boole algebrában ismert ‘0’ és ‘1’ állapotok két normalizált és kölcsönösen ortogonális kvantum állapot-pár szuperpozíciójával ábrázolhatók {  $|0\rangle$  ,  $|1\rangle$  } (egyszerre lehet mindkettő, ill. 0-1 között bármely átmenet lehetséges)
  - Egy kvantumbitet úgy érdemes elképzelní, mint egy gömböt. A klasszikus bitek ennek a gömbnek mindig egy-egy meghatározott pontján találhatók, a kvantumbitek viszont bárhol lehetnek ezen a gömbön belül. Emiatt egy kvantumbit jóval több információt tárolhat, de kisebb energiafelvétel mellett!

# Más alternatíva: D-Wave Kvantumszámítógép

- D-Wave One System (2009): 128 qubit
- D-Wave Two (2012): 512 qubit
- D-Wave 2X (2015): 1000+ qubit
- D-Wave 2000Q (2017): 2000+ qubit (~ **15 m\$**)
- D-Wave 5000+Q (2020): 5000+ qubit
- Félvezetők helyett szupravezető fémet használnak mágneses vákuumban: niobium (ultra alacsony hőmérsékleten  $T=-273\text{ C}^\circ$ ,  $P = 25\text{ kW!}$ )
- HPC: High Performance Computing alkalmazásokra, Cloud
  - parallel-, elosztott számítási struktúra
  - Big data analysis - Optimization – Classification - Machine learning etc.
- Támogatók: Google, NASA, Lockheed CIA, Amazon...



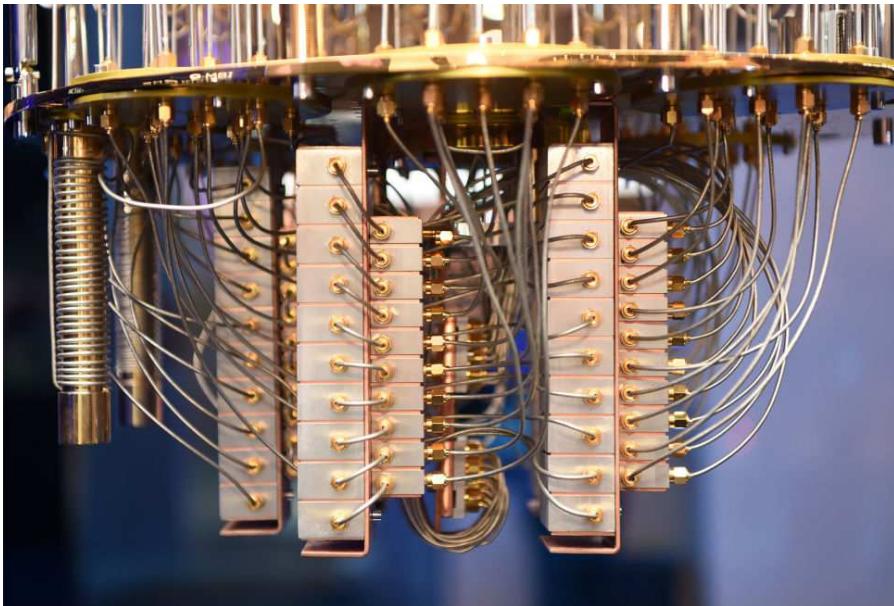
<http://www.dwavesys.com>

[http://index.hu/tech/2016/08/18/programozható\\_kvantumszámítogep](http://index.hu/tech/2016/08/18/programozható_kvantumszámítogep)

[http://index.hu/tech/2014/10/15/a\\_kvantumszike\\_es\\_a\\_kiserteties\\_kapocs/](http://index.hu/tech/2014/10/15/a_kvantumszike_es_a_kiserteties_kapocs/)

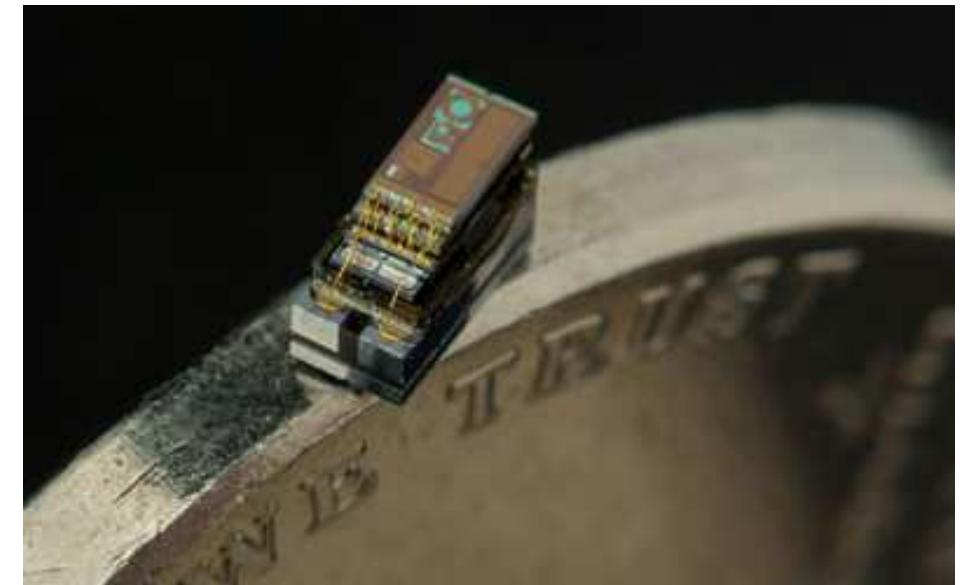
# Más alternatíva: IBM Q System One/Two - Kvantumszámítógép

- IBM Q (2017-): prototípus 50 qubit (IBM Research USA, CERN)
  - Szimulátor, SDK támogatás
  - Q Network: Cloud támogatás
- Első integrált kereskedelmi célú kvantumszámítógépe (2019 – CES): 20 qubit ...
- 2022: IBM Osprey, 433 qubit

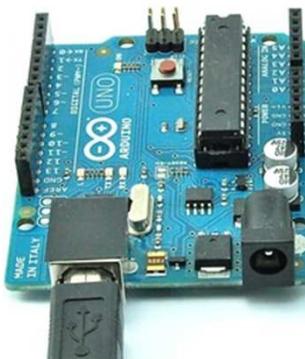


# „Világ legkisebb számítógépe”

- 2018. jún (University of Michigan, USA)
- **M<sup>3</sup> : Michigan Micro Mote: smart-sensor**
  - Hőmérséklet, nyomás,
  - Képalkotó szenzor  
(160x160 pixel)
  - 1 mm<sup>2</sup> felületű!
  - 2 nA disszipáció  
(standby mód)
  - CPU + MEM + PWR  
RF, battery



# „Legnépszerűbb” eszközök: Arduino vs. Raspberry Pi



## ■ **Arduino** – Atmel/Microchip MCU alapú fejlesztő kártya

- Kisebb órajelű (~x10 MHz) MCU mag, kis belső memória, kis bitszélesség (8-, 16 bit)
- Nincs külső memória, nincs OS kezelése, nem real-time eszköz.
- Jó bővíthetőség: „shield”-ek
- Főként egyszerű szabványokat, GPIO-kat kezel, van ADC.
- Olcsó, népszerű, rengeteg szenzor illeszthető, de kisebb komplexitású fejlesztési célokra.  
Ára: \$5-15 (platform függő)

## ■ **Raspberry Pi** – ARM alapúáltalános célú sz.gép, fejlesztő kártya („single board computer”)

- Dedikált, nagy órajelű CPU magok (ARM 32/64 bit ~x100 MHz, memória (DDR3), GPU mag, HDMI stb.)
- Bővíthetőség: SDCard (OS boot), WIFI, BLE, CamIF, de nincs ADC.
- OS/RTOS (HW-es) kezelése Nagyobb komplexitás, több funkció, de drágább.
- Ára: \$ 30- 50 (platform függő)



# Számítógép Architektúrák II.

(MIVIB344ZV)

1. előadás: Boole-algebra, logikai függvények  
(ismétlés)

Előadó: Dr. Vörösházi Zsolt  
[voroshazi.zsolt@mik.uni-pannon.hu](mailto:voroshazi.zsolt@mik.uni-pannon.hu)

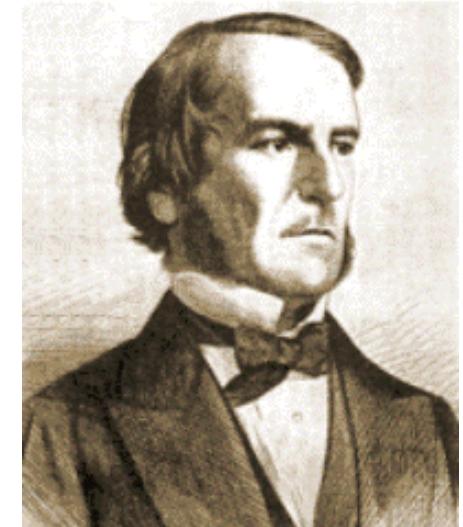
# Kapcsolódó jegyzet, segédanyag:

- Angol nyelvű könyv:  
<http://www.virt.uni-pannon.hu> → Oktatás → Tantárgyak → Számítógép Architektúrák II.  
■ (chapter01.pdf)
- Fóliák, óravázlatok .ppt (.pdf)
- Feltöltésük folyamatosan

# További ajánlott irodalom

-  Dr. Holczinger, Dr. Göllei. Dr. Vörösházi:  
Digitális Technika I. (TAMOP 4.1.2A - 2012) :  
Digitalis technika I TAMOP
-  Dr. Holczinger, Dr. Göllei. Dr. Vörösházi:  
Digitális Technika II. (TAMOP 4.1.2A - 2013) :  
Digitalis technika II TAMOP

# Boole-algebra



(1815-1864)

- „Logikai operátorok” algebrája
- George Boole: először mutatott hasonlóságot az általa vizsgált **logikai operátorok** és a már jól ismert **aritmetikai operátorok** között.
- HW tervezés alacsonyabb absztrakciós szintjén rendkívül fontos szerepe van (specifikáció + logikai egyszerűsítés = „minimalizálás”)

# Boole-algebra elemei

- A vizsgált 3 alapművelet: AND, OR, NOT
- Tulajdonságaik (AND, OR esetén):
  - **Kommutatív:**  $A+B=B+A$ ,  $A \cdot B=B \cdot A$
  - **Asszociatív:**  $A+(B+C)=(A+B)+C=A+B+C$   
 $A \cdot (B \cdot C)=(A \cdot B) \cdot C=A \cdot B \cdot C$
  - **Disztributív:**  $A \cdot (B+C)=A \cdot B+A \cdot C$ ,  
 $A+(B \cdot C)=(A+B) \cdot (A+C)$
- Operátor precedencia (csökkenő sorrendben):
  - NOT
  - AND
  - OR

átzárójelezhetőség!

# Bizonyítás: Disztributivitás

$$A + (B \cdot C) \stackrel{?}{=} (A+B) \cdot (A+C)$$



A	B	C	$B \cdot C$	$A + (B \cdot C)$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

A	B	C	$A+B$	$A+C$	$(A+B) \cdot (A+C)$
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	1	1	1
1	0	0	1	1	1
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1



Mivel a mindkét oldal kimeneti kombinációi azonosak, ezért egyenlőség áll fenn!

# Boole algebra

## azonosságai = axiómák

$$1.) \bar{\bar{A}} = A$$

$$2.) A + 0 = A$$

$$3.) A + 1 = 1$$

$$4.) A + A = A$$

$$5.) A + \bar{A} = 1$$

NOT

$$6.) A \cdot 1 = A$$

$$7.) A \cdot 0 = 0$$

$$8.) A \cdot A = A$$

$$9.) A \cdot \bar{A} = 0$$

OR

AND

$$10.) A + A \cdot B = A$$

$$11.) A \cdot (A + B) = A$$

Elnyelési  
tul.

$$12.) A \cdot B + A \cdot \bar{B} = A$$

$$13.) (A + B) \cdot (A + \bar{B}) = A$$

$$14.) A + \bar{A} \cdot B = A + B^*$$

$$15.) A \cdot (\bar{A} + B) = A \cdot B$$

### De-Morgan azonosságok:

$$16.) \overline{A + B} = \bar{A} \cdot \bar{B}$$

$$17.) \overline{A \cdot B} = \bar{A} + \bar{B}$$

DUALITÁS

# Példa-1: Boole-algebrai azonosság igazolása igazságtáblával

- De-Morgan (17.) NAND:  $\overline{A \cdot B} = \overline{\overline{A}} + \overline{\overline{B}}$

A	B	A·B	NOT (A·B)
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

A	B	NOT A	NOT B	NOT(A) + NOT(B)
0	0	1	1	1
0	1	1	0	1
1	0	0	1	1
1	1	0	0	0

Dualitás elve

- Példa-2: Hasonló módon bizonyítsa be De-Morgan NOR (16.) azonosságot is igazságtábla segítségével!

# Példa-3: Boole algebrai egyszerűsítések

- Igaz-e a következő állítás:

$$\overline{A \cdot (B + C \cdot (B + \overline{A})))} = \underline{\overline{A}} + \underline{\overline{B}}$$

- Megoldás: Belső zárójelek felbontása → egyszerűsítés

$$\overline{A \cdot (B + C \cdot (B + \overline{A})))} \underset{diszt}{=} A \cdot \overline{(B + C \cdot B + C \cdot \overline{A})} = \\ A \cdot \overline{B \cdot (1+C)} =$$

$$A \cdot \overline{(B + C \cdot \overline{A})} = A \cdot B + A \cdot \overline{A} \cdot C = A \cdot B = \underline{\overline{A}} + \underline{\overline{B}}$$



# Logikai hálózatok csoportosítása

Kétféle hálózatot különböztetünk meg:

- **(K.H.) Kombinációs logikai hálózatról** beszélünk: ha a mindenkorai kimeneti kombinációk létrehozásához *elég a bemeneti* („elsődleges”) *kombinációk* pillanatnyi értéke.
- **(S.H.) Sorrendi (szekvenciális) logikai hálózatról** beszélünk: ha a mindenkorai kimeneti kombinációt, nemcsak a *pillanatnyi* bemeneti kombináció, hanem a *korábban fennállt állapot* kombinációk és azok sorrendje is befolyásolja. (Ezen „másodlagos” kombinációk segítségével az ilyen hálózatok képessé vállnak arra, hogy az ugyanolyan bemeneti kombinációhoz **más-más kimeneti** kombinációt szolgáltassanak.)

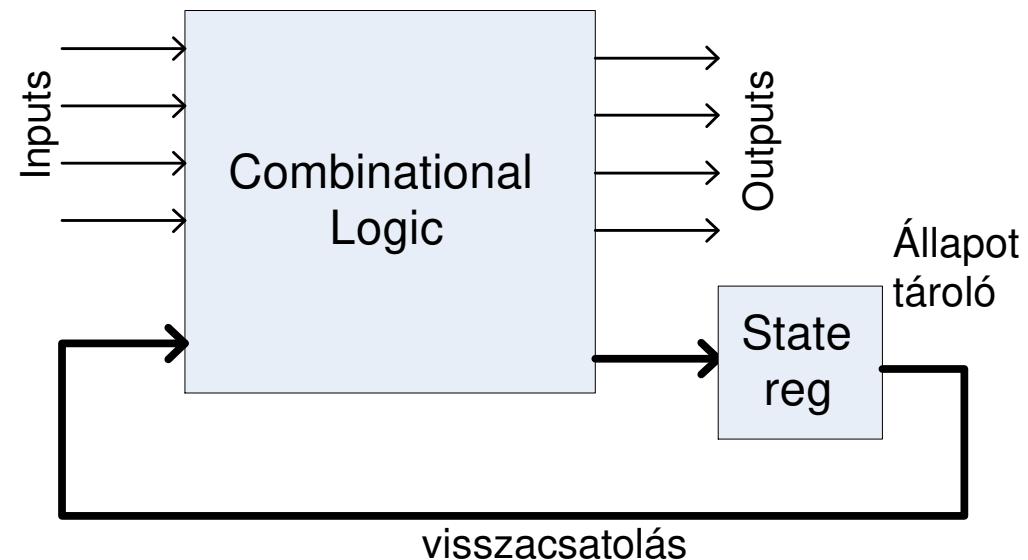
# Kombinációs vs. sorrendi logikai hálózatok felépítése

- Kombinációs hálózat:



Függvény = egyértelmű  
hozzárendelés!

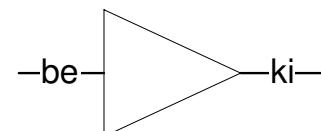
- Sorrendi hálózat:



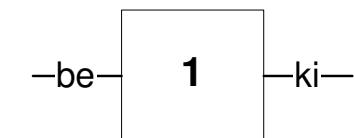
# Egyváltozós logikai függvények:

## ■ Jelmásoló („buffer” - jel-erősítő):

be	ki
0	0
1	1



Nemzetközi  
szabvány

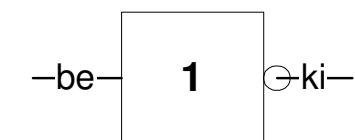
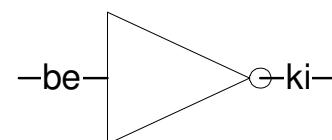


Magyar  
szabvány

## ■ Negálás - Inverter (NOT):

$\bar{A}$

be	ki
0	1
1	0

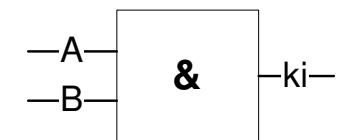
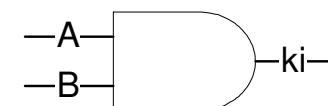


# Kétváltozós logikai függvények:

- ÉS (AND):

$$A \cdot B$$

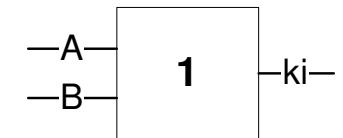
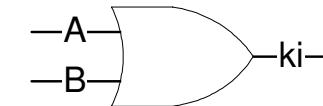
A	B	ki
0	0	0
0	1	0
1	0	0
1	1	1



- VAGY (OR):

$$A + B$$

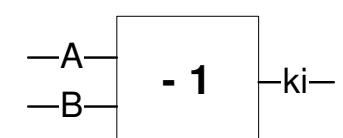
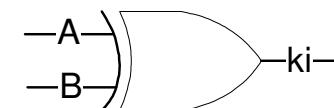
A	B	ki
0	0	0
0	1	1
1	0	1
1	1	1



- Antivalencia (XOR):

$$A \oplus B$$

A	B	ki
0	0	0
0	1	1
1	0	1
1	1	0



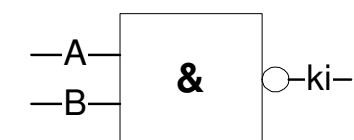
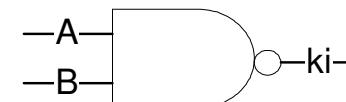
# Kétváltozós log.függv. (folyt.):

## ■ NEM-ÉS (NAND):

$$\overline{A \cdot B}$$

A	B	ki
0	0	1
0	1	1
1	0	1
1	1	0

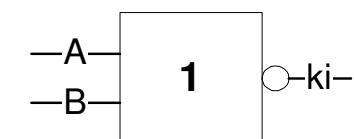
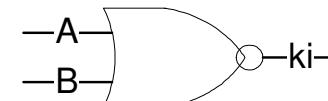
Univerzálisan teljes rendszert a NAND illetve NOR függvény alkot!



## ■ NEM-VAGY (NOR):

$$\overline{A + B}$$

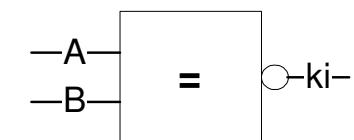
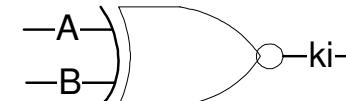
A	B	ki
0	0	1
0	1	0
1	0	0
1	1	0



## ■ Ekvivalencia (NXOR):

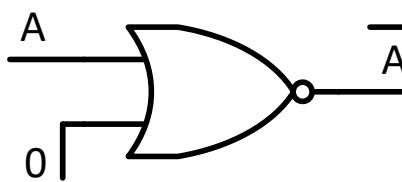
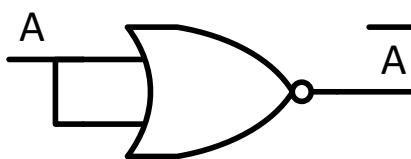
$$A \odot B$$

A	B	ki
0	0	1
0	1	0
1	0	0
1	1	1

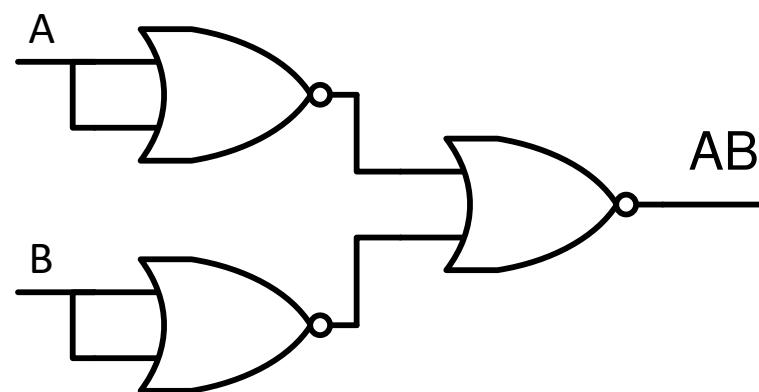


# Definíció: Funktionális teljesség

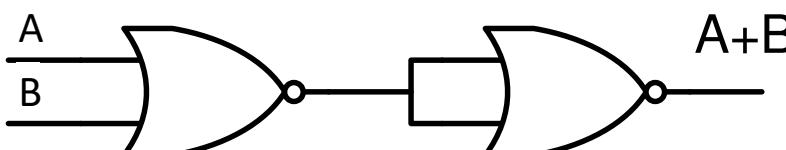
- NOR  $\rightarrow$  INV



- NOR  $\rightarrow$  AND



- NOR  $\rightarrow$  OR

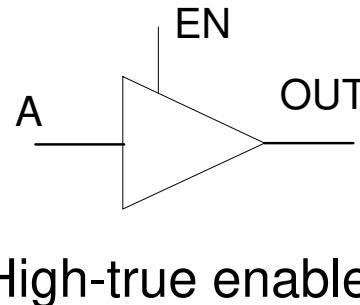


Funkcionálisan teljes / univerzális áramköri elemek:

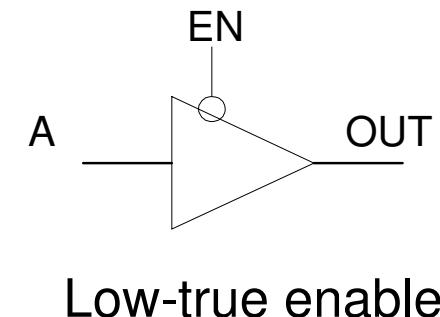
- Logikai hálózatok esetén a CMOS VLSI technológiában a NAND, illetve NOR kapuk.
- (Aritmetikai egységek esetében ilyen univerzális építőelem az „összeadó” )

# Tri-State Buffer:

- buszok esetén használatos: kommunikációs irány változhat
  - Driver: egyirányú kommunikációra
  - Transceiver: kétirányú kommunikációra
- 3 állapota lehet:
  - magas: ‘1’
  - alacsony: ‘0’ (normál TTL szintek)
  - nagy impedanciás állapot: ‘Z’ – minden tranzisztor zár



A	EN	OUT
0	1	0
1	1	1
X	0	Z



# Smart áramkörök ☺





# Számítógép Architektúrák II.

(MIVIB344ZV)

2. előadás: Számrendszerök,  
Nem-numerikus információ ábrázolása

Előadó: Dr. Vörösházi Zsolt  
[voroshazi.zsolt@mik.uni-pannon.hu](mailto:voroshazi.zsolt@mik.uni-pannon.hu)

# Jegyzetek, segédanyagok:

- Könyvfejezetek:

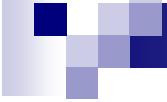
- <http://www.virt.uni-pannon.hu> → Oktatás → Tantárgyak → Számítógép Architektúrák II.  
□ (chapter02.pdf)

- Fóliák, óravázlatok .ppt (.pdf)

- Feltöltésük folyamatosan

# Információ ábrázolás:

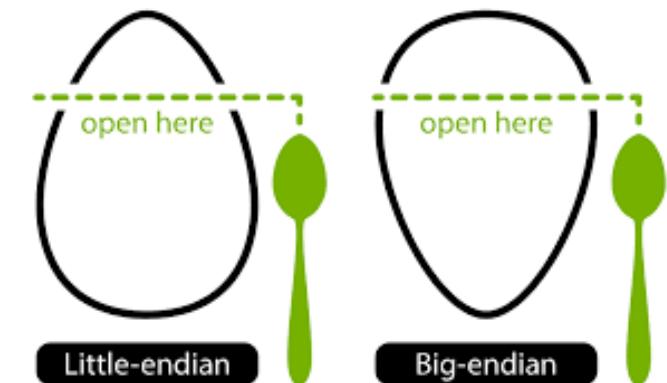
- A) Számrendszerök (numerikus információ):
  - I.) **Egész típusú:**
    - előjel nélküli,
    - előjeles:
      - 1-es komplement,
      - 2-es komplement számrendszer.
  - II.) **Fix-pontos,**
  - III.) **Lebegő-pontos** (IBM-32, DEC-32, IEEE-32),
    - Excess kód (exponens kódolására)
- B) Nem-numerikus információ kódolása
- C) Hiba-detektálás, és javítás (Hamming kód)
  - SEC-DED



# A) Számrendszerök

# Endianitás (endianness)

- A számítástechnikában, az **endianitás** („bit/byte-sorrend” a jó fordítása) az a tulajdonság, ami bizonyos adatok - többnyire kisebb adategységek egymást követő sorozata - tárolási és/vagy továbbítási sorrendjéről ad leírást (pl. két protokoll, vagy busz kommunikációja).
- Ez a tulajdonság döntő fontosságú az értékeknek a számítógép memóriájában bit/byte-onként való tárolása (egy memória címhez relatívan), ill. továbbítása esetében
- Két lehetséges sorrend:
  - Big-Endian (BE) formátum
  - Little-Endian (LE) formátum



📖 Háttér: Az eredeti angol kifejezés az *endianness* (1980) egy utalás arra a háborúra, amely a két szembenálló csoport között zajlik, akik közül az egyik szerint a lágytojás nagyobb/vastagabb végét (big-endian), míg a másik csoport szerint a lágytojás kisebb végét (little-endian) kell feltörni. Erről Swift ír a *Gulliver Kalandos Utazásai* című könyvében ☺

# „Kicsi a végén” - Little-endian (LE)

Példa: 32-bites „3A 4B 1C 2D” értéket a 0x100... címtől növekvő módon, **4×1byte**-os tárolási egységekben tároljuk:

0x100 0x101 0x102 0x103 ...

- Ekkor a kevésbé jellemző ("legkisebb") byte (az angol Least Significant Byte rövidítéséből *LSB* néven ismert) az első, ez a 2D, tehát a *kis vége kerül „előre”, azaz a legkisebb címen van tárolva (0x100)*:

0x103	0x102	0x101	0x100	memóriacímek [31:0]	
3A	4B	1C	2D		
MSB	←		LSB	adategységek	

- LE = Hagyományos, általánosan használt formátum:** ha más nem mondunk, nem kötik ki külön, ezt feltételezzük!
- pl. Intel, AMD, illetve ARM processzorok stb.

# „Nagy a végén” - Big-endian (BE)

Példa: 32-bites értéket „3A 4B 1C 2D”, a 0x100 címtől kezdve tároljuk a memóriában, **4x1 byte-os** tárolási egységekben :

- 1 byte-onként növekvő címekkel rendelkezik

0x100 0x101 0x102 0x103 ... "2D 1C 4B 3A" ...

*memóriacímek*

0x103	0x102	0x101	0x100	[0:31]
2D	1C	4B	3A	
LSB			MSB	<i>adategységek</i>

- Ekkor a „legjellemzőbb” byte - „Most Significant Byte” (**MSB**), ami itt a „3A” - a memóriában az *legalacsonyabb címen van tárolva (0x100)*, míg a következő "jellemző byte" (4B) az egyel nagyobb címen van tárolva, és így tovább.
- *Bit/byte-reversed format! (fordított formátum)*
- Pl: speciális, beágyazott rendszerek processzorai, pl. MCU – mikrovezérlők, programozható FPGA-k (MicroBlaze, PowerPC), stb<sup>7</sup>

# I.) Egész típusú számrendszerek

Bináris ( $p=2$ ) számrendszer: "1" / "0" (I / H, T / F, ...)

## ■ Átalános szabály:

N biten  $\rightarrow 2^N$  lehetséges érték ábrázolható!

- Példa: N = 2 byte-os (16 bites), vagy N = 4 byte-os (32-bites) bináris, pozitív, egész szám esetén:
  - $2^{16}-1 = 65535$  vagy
  - $2^{32}-1 = 4\ 294\ 967\ 295$  a maximálisan ábrázolható érték.

- Negatív alak = Előjel kezelése? legegyszerűbb mód, ha a szám legfelső helyiértékű (MSB) bitjét **előjelbitnek (S)** tekintjük:

„+”: 0, „-”: 1 (de nem feltétlenül ez az általános!)

- Ennek oka: célszerű olyan ábrázolási módot alkalmazni, ahol a kivonás (-) művelete  $\rightarrow$  összeadással (+) helyettesíthető.



# a.) előjel nélküli egész:

- Unsigned integer:

$$V_{\text{UNSIGNED INTEGER}} = \sum_{i=0}^{N-1} b_i \times 2^i$$

- ahol  $b_i$  az  $i$ -edik pozícióban lévő '0' vagy '1'
- Reprezentálható értékek határa: 0-tól  $2^N - 1$  -ig
- Helyiértékes rendszer
- Negatív számok ábrázolása nem lehetséges!

- **Pl: (Legyen N:=6, LE, unsigned int)**

$$10\ 1101_2 \Rightarrow 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 45_{10}$$

# b.) 1's komplementens rendszer

- V értékű, N bites rendszer:  **$2^N - 1 - V$** 
  - „0” lesz ott ahol „1”-es volt, „1”-es lesz ott ahol „0” volt (mivel egy szám negatív alakját, bitjeinek kiegészítésével kapjuk meg).
- csupán minden bitjét negálni, (gyors műveletet)
- Értékhatár:  $2^{(N-1)} - 1$  –től –  $(2^{(N-1)} - 1)$  –ig terjed,
- Nem helyiértékes rendszer
  - kétféleképpen is lehet ábrázolni a zérust!! (-0 / +0, ellenőrzés szükséges)

# Példa: 1's komplement

Példa: N:=6, LE

$$V = 01\ 0010_2 = 18_{10}$$

a.) Adja meg a fenti V szám  
1's komplementét!

$$V(1's) = 10\ 1101_2$$

b.) Milyen decimális,  
negatív, egész értéket  
ábrázol a fenti bináris 1's  
komplement szám?

$$V(1's) = 10\ 1101_2 = ? \\ = -18_{10}$$

V érték	V(Unsigned int)	V(1's comp)
01 1111	31	31
01 1110	30	30
...	...	...
010 010	18	18
...	...	...
00 0010	2	2
00 0001	1	1
00 0000	0	+0
11 1111	63	-0
11 1110	62	-1
11 1101	61	-2
...	...	...
10 1101	45	-18
...	...	...
100001	33	-30
100000	32	-31

# c.) 2's komplement rendszer

- $V$  jelöli a szám értékét,  $N$  bites, LE rendszer:

$$V_{2^{\text{S COMPLEMENT}}} = -b_{N-1} \times 2^{N-1} + \sum_{i=0}^{N-2} b_i \times 2^i$$

- Értékhatár:  $-2^{(N-1)}$  –től ...  $+2^{(N-1)} - 1$  –ig
  - ha MSB='1', a szám negatív
  - helyiértékes rendszer!

Bit Pattern	Value	Note
01111111	127	Largest representable value.
01111110	126	
01111101	125	
...	...	
...	...	
00000010	2	Note that leading zero indicates positive number.
00000001	1	
00000000	0	Unique representation of <b>zero</b> .
11111111	-1	Minus one is always all ones.
11111110	-2	Note that leading one indicates negative number.
11111101	-3	
...	...	
...	...	
10000010	-126	
10000001	-127	
10000000	-128	Smallest (most negative) representable value.

# Példa: 2's komplement

Legyen előjeles – 2's komplement rendszer,  
**N:=6, LE**

Milyen decimális, negatív, egész értéket ábrázol az alábbi bináris 2's komplement szám?

■  $V(2's) = 10\ 1101_2 = ?$

$$-1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = \\ = -32 + 8 + 4 + 1 = -19_{10}$$

■ További számítási módszerek:

\* azt jelöli, amikor az adott helyiértéken '1'-et kell kivonni még az  $X_i$  értékéből (borrow from  $X_i$ )

I.

$$\begin{array}{r} 0\ 1\ 0\ 0\ 1\ 1 & \text{a szám} \\ 1\ 0\ 1\ 1\ 0\ 0 & \text{szám 1's kompl.} \\ + & 1 & +1 \\ \hline 1\ 0\ 1\ 1\ 0\ 1 & -19 = V(2's) \end{array}$$

II.

$$\begin{array}{r} * * * * * * \\ X \quad \cancel{1} \cancel{0} \cancel{0} \cancel{0} \cancel{0} \cancel{0} \\ - Y \quad - \quad 0\ 1\ 0\ 0\ 1\ 1 \\ \hline Z \quad -19 \equiv 1\ 0\ 1\ 1\ 0\ 1 \end{array}$$

64  
- 19  
45  
13

## II.) Fix-pontos számrendszer

### Műveletek:

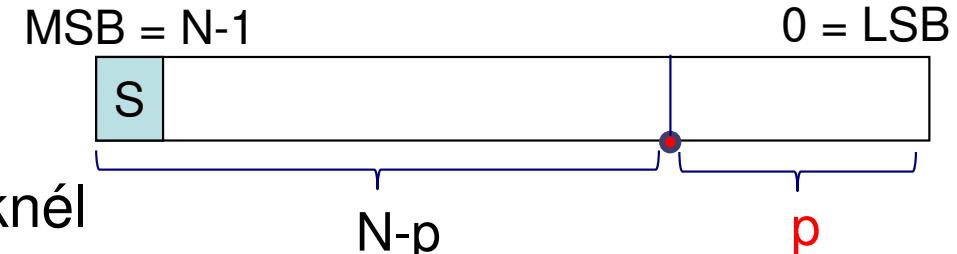
- +, - : mint egész számrendszereknél
- \*, / : vizsgálni kell a tizedespont helyét

### V jelöli az előjeles, bináris, fixpontos szám értékét (LE):

$$V_{\text{FIXED POINT}} = -b_{N-1} \times 2^{N-p-1} + \sum_{i=0}^{N-2} b_i \times 2^{i-p}$$

### Paraméterek:

- N: fixpontos szám hossza (egész + törtrész együttesen)
- p: radix (tizedes) pont** helye, törtrész hossza
- differencia**,  $\Delta r = 2^{-p}$  (számrendszer finomsága, azaz két szomszédos szám távolsága)
  - Ha  $p=0 \rightarrow \Delta r=1 \rightarrow$  egész számrendszer pl: -1, 0, 1, 2, 3, ...
  - Ha  $p=1 \rightarrow \Delta r=1/2 \rightarrow$  fixpontos számrendszer pl: -1/2, 0, 1/2, 1, 3/2, 2, ...
  - Ha  $p=2 \rightarrow \Delta r=1/4 \rightarrow$  fixpontos számrendszer pl: -1/4, 0, 1/4, 1/2, 3/4, 1, ...



# Példa: Fixpontos rendszer

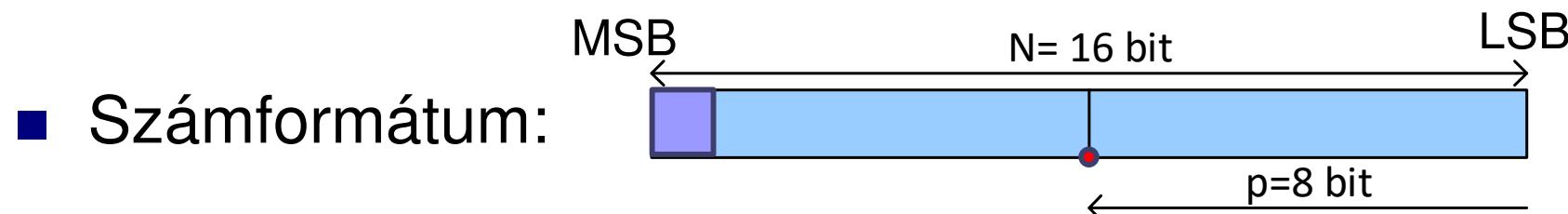
Legyen egy **N:=16 bites, LE, bináris, 2's komp. fixpontos rdsz.** ahol **p:=8**.

Kérdések:

- Ábrázoljuk az N, p paramétereknek megfelelő számformátumot
- V(legkisebb pozitív)=?
- V(legnagyobb pozitív)=?
- V(legkisebb negatív)=? // "leg-negatívabb"
- V(legnagyobb negatív)=? // "legkevésbé negatív"
- V(zéró),
- $\Delta r = ?$

*Minden paramétert számolunk ki, és ahol lehet decimális értékben / hatványalakban is megadva!*

# Megoldás: Fixpontos rendszer



- Differencia  $\Delta r = \underline{2^{-8}} = \underline{1/256} = (0,390625 \cdot 10^{-2})$
- $V(\text{zero}) = \underline{00000000.00000000} = \underline{0.0}$
- $V(\text{legkisebb poz}) = \underline{00000000.00000001} = \underline{2^{-8}} = \underline{1/256} = \Delta r = (0,390625 \cdot 10^{-2})$
- $V(\text{legnagyobb poz}) = \underline{01111111.11111111} \approx \underline{128} = \underline{128 - \Delta r}$
- $V(\text{legkisebb negatív}) = \underline{10000000.00000000} = \underline{-2^7} = \underline{-128}$
- $V(\text{legnagyobb negatív}) = \underline{11111111.11111111} = \underline{-2^{-8}} = \underline{-1/256} = -\Delta r = (-0,390625 \cdot 10^{-2})$

# III.) Lebegőpontos számrendszerek

- Nagyobb pontosság vs. több paraméter tárolása → nehezebb számolni
- 7 különböző paraméter: *a számrendszer alapja, előjele és nagysága, a mantissa alapja, előjele és hosszúsága, ill. a kitevő alapja.*
- **Egyeszerű matematikai jelölés:** (előjel) Mantissa  $\times$  Alap<sup>Kitevő</sup>
- Fixpontosnál nagyságrendekkel kisebb, vagy nagyobb számok ábrázolására is lehetőség van:
  - PI: Avogadro-szám:  $6.022 \times 10^{23}$
  - PI: proton tömege  $1.673 \times 10^{-24}$  g
- Normalizált lebegőpontos rendszerek: pl. DEC-32, IBM-32, IEEE-32
  - **32-bites, vagy egyszeres pontosságú – float (C)** vagy
  - 64-bites, vagy dupla pontosságú – double (C)

# Normalizálás (mantissa)

Példa: adott decimális ( $r_b = 10$ -es) alapú lebegőpontos szám **normalizálása** esetén:

- $32\ 768_{10} = 0.32768 \times 10^5 =$   
 $3.2768 \times 10^4 = 32.768 \times 10^3 =$   
 $327.68 \times 10^2 = 3267.8 \times 10^1$

(mindegyik érvényes alak) DE

- $[\frac{1}{r_b}, 1[$  közé normalizált alak:

**0.32768 × 10<sup>5</sup>**

Mantissa igazítása ↔ exponens változása!

# Lebegőpontos rendszer (FPN) jellemző paraméterei

- Számrendszer / kitevő alapja:  $r_b$   $r_e$
- Mantissza értéke:  $V_M = \sum_{i=0}^{N-1} d_i \times r_b^{i-p}$ 
  - Maximális:  $V_{M_{\max}} = 0.d_m d_m d_m \dots = (1 - r_b^{-m})$
  - Minimális:  $V_{M_{\min}} = 0.\textcolor{red}{1}00 \dots = 1/r_b$ 
    - Ahol  $d_m$  a számjegyek
  - Radix pont helye:  $p$ 
    - (a  $p$  helye az exponens értékével összefüggésben változik!)
  - Mantissza bitjeinek száma:  $m$
- Exponens értéke (max / min):  $V_E$   $V_{E_{\max}}$   $V_{E_{\min}}$
- Lebegőpontos szám értéke:  $V_{FPN} = (-1)^{SIGN} V_M \times r_b^{V_E}$

# Excess-kód

## Miért használják az Excess „kódolást” ?

- **Lebegőpontos** számok **kitevőjét (exponens-ét)** tárolják / kódolják e módszerrel. Cél: a kitevő **NE legyen negatív**, ezért eltolással oldják meg, hogy a negatív kitevőhöz egy pozitív számot adnak hozzá.
- Megjegyzés: egy lebegőpontos szám előjelbitje, a mantissa előjelét, és nem pedig az exponens előjelét tárolja!
  - V: az exponens valódi értéke, (lehet pozitív, negatív)
  - E: az excess (offset/eltolás értéke)
  - S: a reprezentálni kívánt érték, azaz a kitevő kódolt értéke, melyet eltárolunk (ez csak pozitív lehet)
  - $S = V + E$

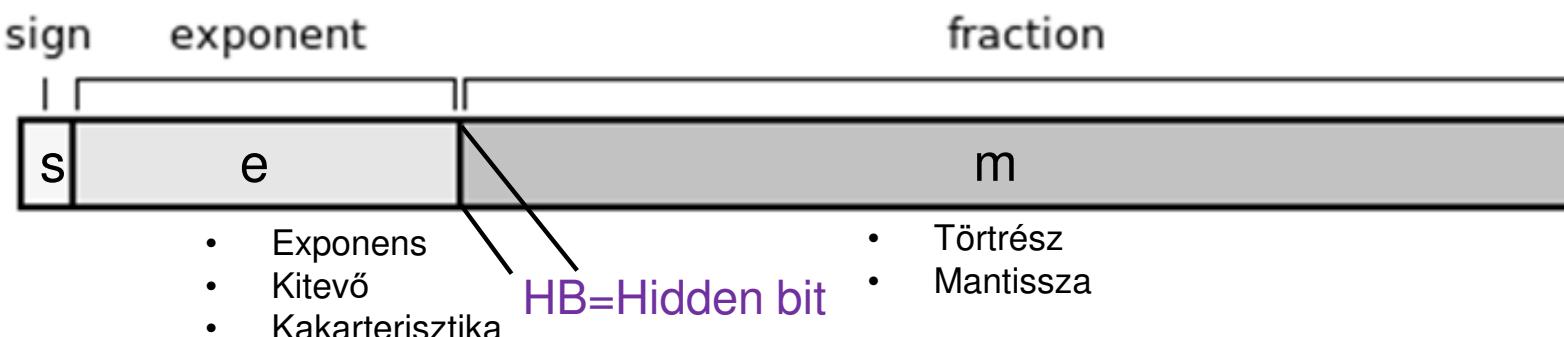
# Lebegőpontos számrendszerek

- **DEC-32:**  $r_b=2$ ,  $r_e=2$ ,  $m=24$  (nincs HB),  $p=24$  ( $m=p$ ),  $e=8$ , az exponenst Excess-128 kódolással tárolja. Normalizálás:  $[\frac{1}{r_b}, 1[$
- **IBM-32:**  $r_b=16$ ,  $r_e=2$ ,  $m=p=6$  (nincs HB),  $e=7$ , az exponenst Excess-64 kódolással tárolja. Normalizálás:  $[\frac{1}{r_b}, 1[$
- **IEEE-32:**  $r_b=2$ ,  $r_e=2$ ,  $m=24$ , de  $p=23!$  (**HB='1'**),  $e=8$ , az exponenst Excess-127 kódolással tárolja. Normalizálás:  $[\text{HB}, r_b[ = [1, 2[$

# IEEE 754-1985 számformátum

- Nemzetközileg elfogadott szabvány a bináris lebegőpontos számok tárolására, amely tartalmazza:

- negatív zérust is:  $-0 = 1\_000..0000\_00...0000$  (két zérus:  $+0$  )
- normalizálatlan (denormált) számok (Hidden-Bit = 0)
- NaN: nem szám (pl.  $\frac{\pm 0}{\pm 0} = \text{NaN}$ ; vagy  $\pm 0 \times \pm \infty = \text{NaN}$  )
- $\underline{\pm \infty}$



**Sign-magnitude format („előjel-hossz” formátum):** az előjel (sign) külön biten van el tárolva (MSB), az exponens kódolt (excess-el „eltolt”), ill. a törtrész (mantissza) következik végül. Fontos a sorrendük!

# IEEE-32 bites normalizált lebegőpontos rendszer

- $V_E [min,max] = [-126, 127] \rightarrow [1, 254]$  az Excess127-el eltolt exponens tartomány

- Speciális jelentőség:

- $V_E = 0$  értékénél (zérus ábrázolása)
  - $V_E = [255]$  értékénél lehetőség van bizonyos információk tárolására

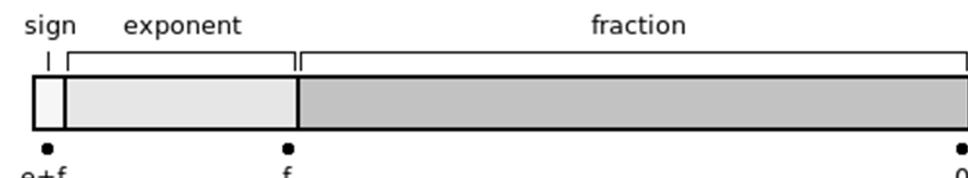
$$(+\infty) + (+7) = (+\infty)$$

$$(+\infty) \times (-2) = (-\infty)$$

$$(+\infty) \times 0 = \text{NaN}$$

$$0 / 0 = \text{NaN}$$

$$\text{Sqrt}(-1) = \text{NaN}$$



$V_E$ [255]	S (előjel)	Ábrázolás jelentése
$\neq 0$	X	Nem egy szám (NaN)
0	0	$+\infty$
0	1	$-\infty$

# Különböző pontosságú IEEE lebegőpontos rendszerek

Típus IEEE	Sign (s)	Exponens (e)	Excess-kód (Exc)	Mantissa (p < m)	Teljes szóhossz
<b>Half (IEEE 754r)</b>	1	5	15	10	16
<b>Single</b>	1	8	127	23	32
<b>Double</b>	1	11	1023	52	64
<b>Quad</b>	1	15	16383	112	128

# Példa: DEC-32

Ábrázoljuk DEC-32 rendszerben a következő decimális számot:  $-12.625_{10} = ?$

DEC-32 paraméterei:  $r_b=2$ ,  $r_e=2$ ,  $m=p=24$ , (nincs HB),  $e=8$ , Excess-128

$$-12.625_{10} = \underbrace{1100}_{\text{sign}} \cdot \underbrace{101}_e_2 = \text{(normalizálás } [\frac{1}{r_b}, 1[) \Rightarrow$$

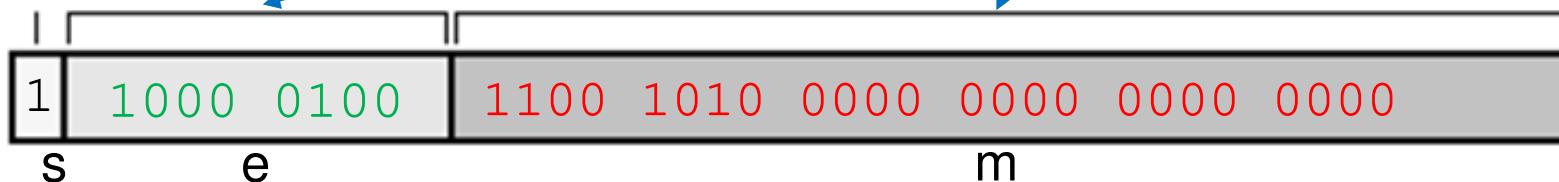
$$= 0.1100101_2 \times 2^4$$

Kitevő:  $4_{10} \Rightarrow 100_2 + \text{Exc-128} =$

$s=1$

$$= 100 + 10000000 =$$

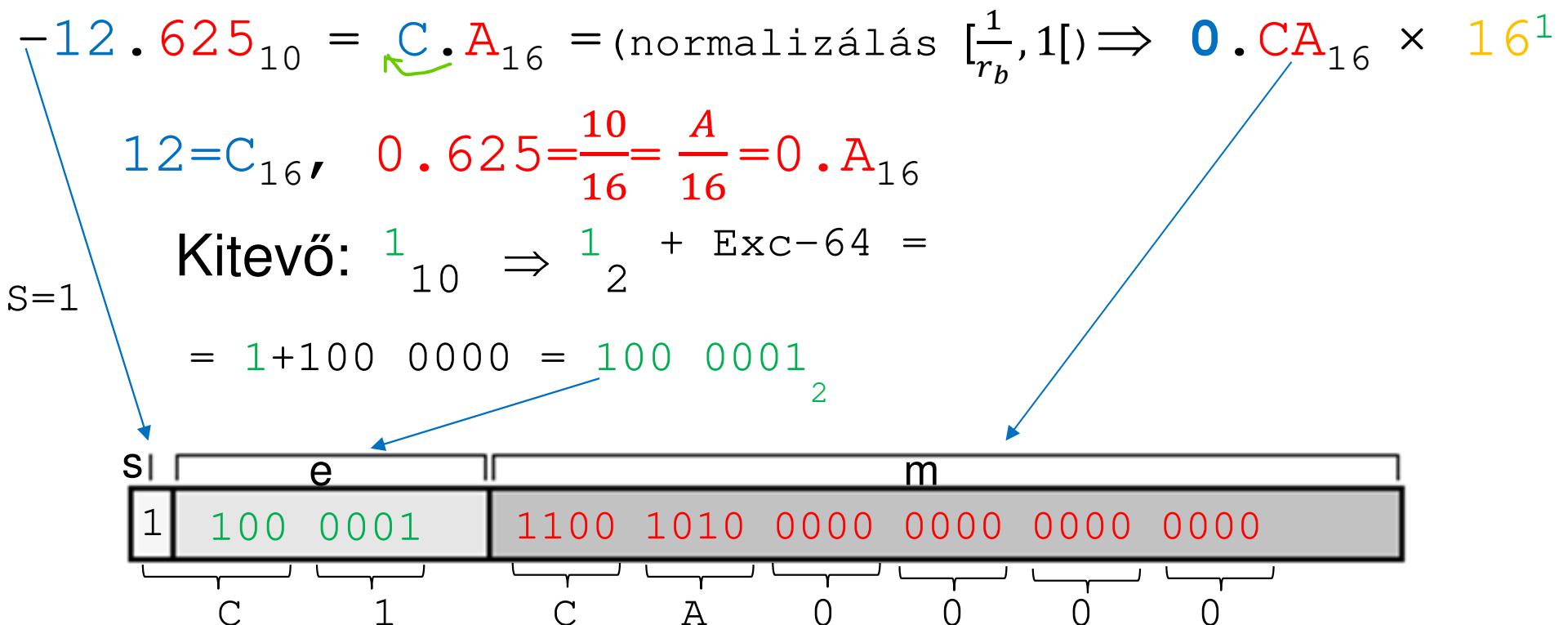
$1000_2 0100$



# Példa: IBM-32

Ábrázoljuk IBM-32 rendszerben a következő decimális számot:  $-12.625_{10} = ?$

IBM-32 paraméterei:  $r_b=16$ ,  $r_e=2$ , ( $m=p=6$ ), (nincs HB),  $e=7$ , Excess-64



# Példa: IEEE-32

Ábrázoljuk IEEE-32 rendszerben a következő decimális számot:  $-12.625_{10} = ?$

IEEE-32 paraméterei:  $r_b=2$ ,  $r_e=2$ , ( $m=24>p=23$ ), (**HB=1!**),  $e=8$ , Excess-127

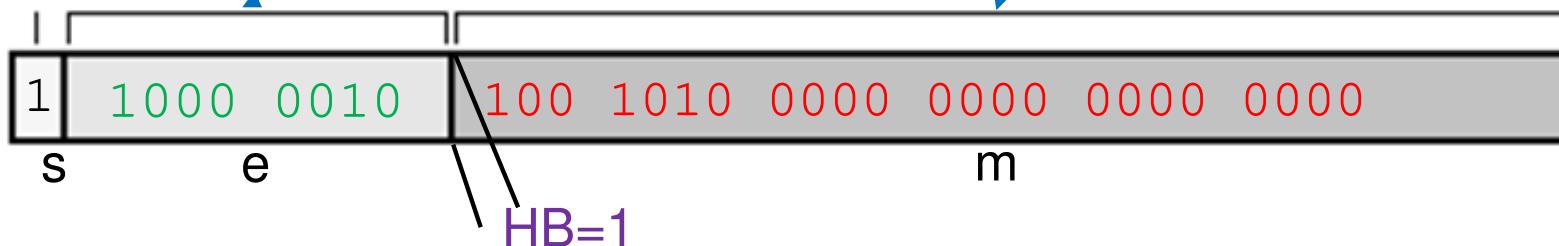
$$-12.625_{10} = \underbrace{1100}_{\text{sign}}.\underbrace{101}_2 = \text{(normalizálás [1,2])} \Rightarrow \\ = 1.\underbrace{100101}_2 \times 2^3$$

Kitevő:  $3_{10} \Rightarrow 11_2 + \text{Exc-127} =$

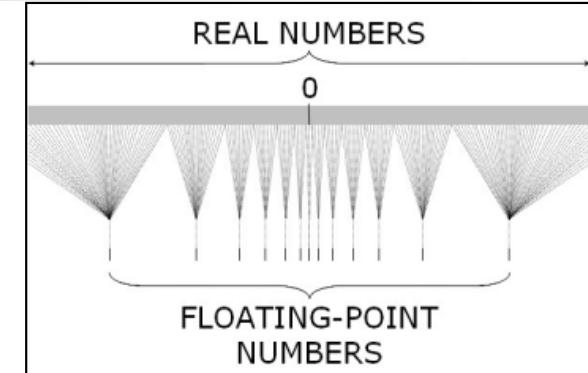
$S=1$

$$= 11 + 111\ 1111 =$$

$$1000\ 0010_2$$

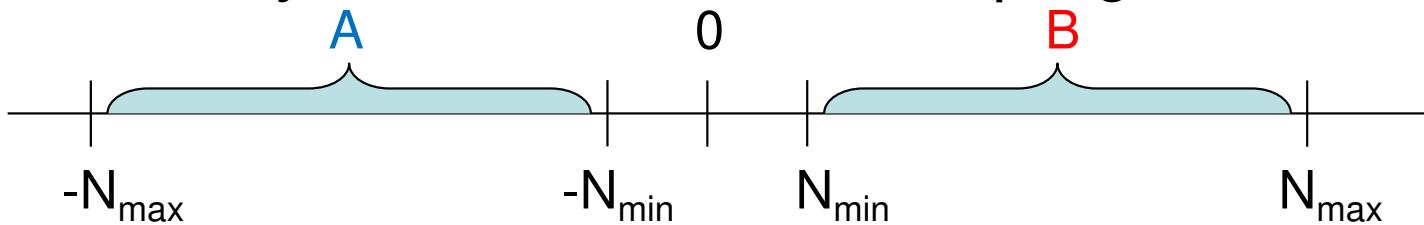


# Lebegőpontos rendszer dinamika tartománya

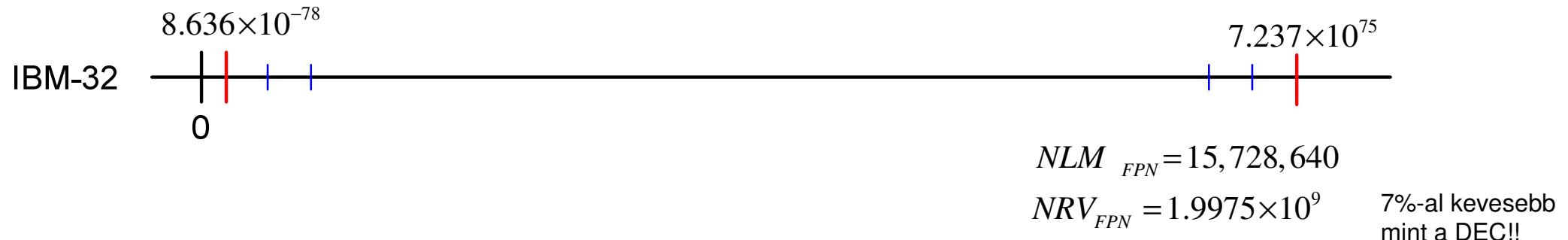
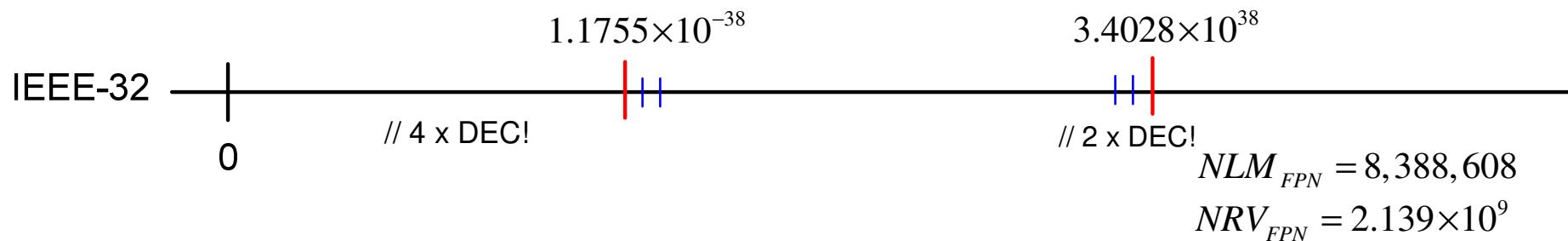
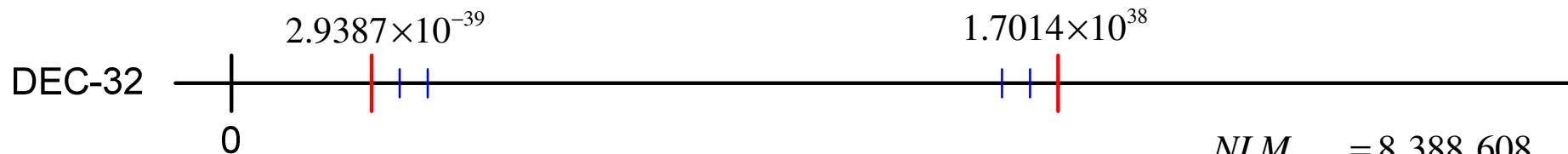


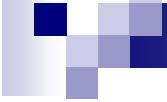
A lebegőpontos számábrázolás **A** és **B** számtartományt tudja (részben) ábrázolni.

- Ha  $x < -N_{\max}$  vagy  $x > N_{\max}$ , akkor **túlcsordulásról**,
- Ha  $-N_{\min} < x < N_{\min}$ , akkor **alulcsordulásról** beszélünk, és ekkor x nem ábrázolható.
- Az ilyen esetek kezelése a programozó feladata.



# Lebegőpontos számrendszerek összehasonlítása (ha FPN előjele pozitív):





## B) Nem-numerikus információ kódolása

# Nem-numerikus információk

- Szöveges,
- Logikai (Boolean) információt,
- Grafikus szimbólumokat,
- és a címeket, vezérlési karaktereket értjük alattuk

# Szöveges információ

- Minimális: 14 karakterből álló halmazban: számjegy (0-9), tizedes pont, pozitív ill. negatív jel, és üres karakter.
- + ábécé (A-Z), a központozás, címkék és a formátumvezérlő karakterek (mint pl. vessző, tabulátor, (CR: Carriage Return) kocsi-vissza, sormelés (LF:Line Feed) , lapemelés (FF: From Feed), zárójel)
- Így elemek száma 46: 6 biten ábrázolható  
 $\lceil \log_2 46 \rceil = 6 \text{ bit}$
- De 7 biten tárolva már kisbetűs, mind pedig a nagybetűs karaktereket is magába foglalja

# Szöveges információ kódolás

- BCD (Binary Coded Decimal): 6-biten
  - nagybetűk, számok, és speciális karakterek
- EBCDIC (Extended Binary Coded Decimal Interchange Code): 8-biten (A. Függelék)
  - + kisbetűs karaktereket és kiegészítő-információkat
  - 256 értékből nincs minden egyik kihasználva
  - Továbbá I és R betűknél szakadás van!
- ASCII (American Standard Code for Information Interchange): (A függelék) – alap 7-biten / extended 8-biten
- UTF-n (Universal Transformation Format): váltózó hosszúságú karakterkészlet (többnyelvűség támogatása)

# EBCDIC

HEX DIGITS 1ST → END ↓	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
<b>-0</b>	ØP SP010000	& SAM000000	- SP100000	ø LC010000	Ø LC020000	º SM100000	µ SM170000	€ SC040000	{ SM110000}	) SM140000	\ SM070000	0 ND100000
<b>-1</b>	ØSØP SP030000	é LT010000	/ SP120000	É LT110000	a LA210000	j LJ010000	~ ~	£ A	J 000	÷ LJ020000	1 SA060000	1 ND0610000
<b>-2</b>	å LA150000	ê LT150000	Å LA160000	Ê LT160000	b LB210000	k LK210000			K 000	S LK020000	2 LS020000	2 ND020000
<b>-3</b>	å LA170000	ë LT170000	À LA180000	È LT180000	c LC010000	l LL010000			L 000	T LL020000	3 LT020000	3 ND020000
<b>-4</b>	à LA190000	é LT190000	À LA140000	È LT140000	d LC010000	m LM010000	.		M 000	U LM020000	4 LU020000	4 ND040000
<b>-5</b>	á LA110000	i LT110000	Á LA120000	í LT120000	e UE010000	n UN010000	*	§ SM040000	W LE020000	N UN020000	V LY020000	S ND040000
<b>-6</b>	ã LA180000	í LT180000	Ã LA200000	Í LT190000	f LF010000	o LG010000	W LW010000	¶ SM050000	F LF020000	O LO020000	W LU020000	6 ND060000
<b>-7</b>	à LA270000	í LT170000	À LA280000	Í LT180000	g LG010000	p UP010000	X UG010000	Œ LO030000	G LG020000	P UP020000	X UX020000	7 ND070000
<b>-8</b>	ç LC040000	i LT130000	Ç LC030000	í LT140000	h LH010000	q LG010000	y LY010000	œ LO050000	H LH020000	Q LG030000	Y LY020000	8 ND080000
<b>-9</b>	ñ LN100000	ß LS0610000	Ñ LA020000	º SD100000	i LS010000	r LS0210000	z LZ010000	ÿ LY100000	I LI020000	R LR020000	Z LZ020000	9 ND090000
<b>-A</b>	Ý LY120000	! SP020000	ſ LS220000	: SP130000	¢ SP170000	ø SM0210000	ø SP030000	¬ SM060000	(ØY) SP120000	í RD011000	z RD021000	3 ND031000
<b>-B</b>	.	s SP110000	,	# SC030000	¤ SP080000	¤ SM010000	¤ SP160000	¤ SM020000	¤ LZ010000	ö LO160000	ü LU150000	ö LO160000
<b>-C</b>	< SA030000	* SM040000	% SM020000	@ SM050000	ð LC030000	æ LA210000	D LD040000	- SD011000	ð LO170000	ü LU170000	ö LO180000	ü LU180000
<b>-D</b>	( SP060000	) SP070000	— SP090000	' SP050000	ÿ LY110000	ž LZ210000	[ SM080000	] SM090000	ò LO180000	ù LU190000	ò LO140000	ù LU140000
<b>-E</b>	+	; SA010000	> SA020000	= SA040000	þ LT060000	Æ LA050000	þ LT040000	Þ LZ030000	ò LO110000	ú LU110000	ó LO120000	ú LU120000
<b>-F</b>	 SM010000	~ SD020000	? SP150000	" SP040000	± SA030000	€ SC010000	® SM070000	× SM090000	ò LO100000	ÿ LY170000	ö LO090000	ó LO080000

# Extended ASCII (1 byte)

		Standard								Extended							
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	•	SP	0	@	P	‘	p	█	’	SP	○	À	ò	à	á	í	ó
	!	1	A	Q	a	q	!	,	,	!	+	ñ	;	r	J	ú	ü
	”	2	B	R	b	r	,	,	,	φ	z	í	j	â	,	,	,
	#	3	C	S	c	s	f	“	“	£	z	í	ç	ç	é	é	é
	\$	4	D	T	d	t	”	”	”	H	‘	g	ş	ç	ç	ö	ö
	£	5	E	U	e	u	...	•	•	¥	µ	í	ş	ş	ø	ø	ø
	&	6	F	V	f	v	†	-	-	¶	ı	ç	ç	ç	,	,	,
	†	7	G	W	g	w	‡	-	-	\$	·	ı	x	ç	÷	,	,
	(	8	H	X	h	x	˜	~	~	..	,	ç	ç	ç	è	è	è
	)	9	I	Y	i	y	‰	‰	‰	@	ı	ö	ö	ö	é	ú	ú
	*	:	J	Z	j	z	˜	˜	˜	˜	ı	ç	ç	ç	ê	ê	ê
	+	:	K	[	k	{	<	>	>	«	»	ç	ç	ç	ë	ë	ë
	,	<	L	\	l		Œ	œ	œ	„	„	ç	ç	ç	ü	ü	ü
	-	=	M	]	m	}	Œ	œ	œ	-	-	ç	ç	ç	ï	ï	ï
	.	›	N	^	n	~	Œ	œ	œ	®	®	ç	ç	ç	î	î	î
	/	?	O	_	o	█	’	’	’	—	?	ı	ı	ı	ı	ı	ı

Legend



Code point contains a non-printable control character.

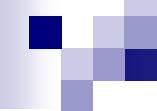


Code point is unoccupied; reserved for future use.

# Unicode

Nyelvkészletek: Alap, - Latin 1/2, görög, cirill, héber, arab stb.

**Általános írásjelek, matematikai, pénzügyi, mértani szimbólumok stb.**



## C) Hamming hibakódolás – Hiba-detektálás, és javítás

# Hibakódolás - Hibadetektálás és Javítás

- **N bit segítségével  $2^N$  különböző érték, cím, vagy utasítás ábrázolható**
- 1 bittel növelve ( $N+1$ ) bit esetén:  $2^N$  -ről  $2^{N+1}$  –re: tehát megduplázódik az ábrázolható értékek tartománya
- ***Redundancia***: „többlet bitek” segítségével lehet a hibákat detektálni, ill. akár javítani is!
  - Redundáns többlet bitek a paritás, v. kódbitek.

# Paritás bit

- Legegyszerűbb hibafelismerési eljárás, a paritásbit átvitele. Két lehetőség:

Adatbitek | Paritásbit(kódbit)

- páros paritás 1 1 0 1 | 1
- páratlan paritás 1 1 0 1 | 0

- **Páros paritás:** az '1'-esek száma páros.
  - Az adatszóban lévő '1'-esek számát '1' vagy '0' hozzáadásával **párossá** egészítjük ki.
  - '0' a paritásbit, ha az '1'-esek száma páros volt.
- **Páratlan paritás:** az '1'-esek száma páratlan.
  - A adatszóban lévő '1'-esek számát '1' vagy '0' hozzáadásával **páratlan**ná egészítjük ki.
  - '1' a paritásbit, ha az '1'-esek száma páros volt.

# Paritás bit generáló áramkör

## ■ Paritásbit képzése:

- ANTIVALENCIA (XOR) művelet alkalmazása a kódszó bitjeire, pl. 'n' adatbit esetén „n-1”-szer!

## ■ Példa:

Kódszó

Paritásbit(P)

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0



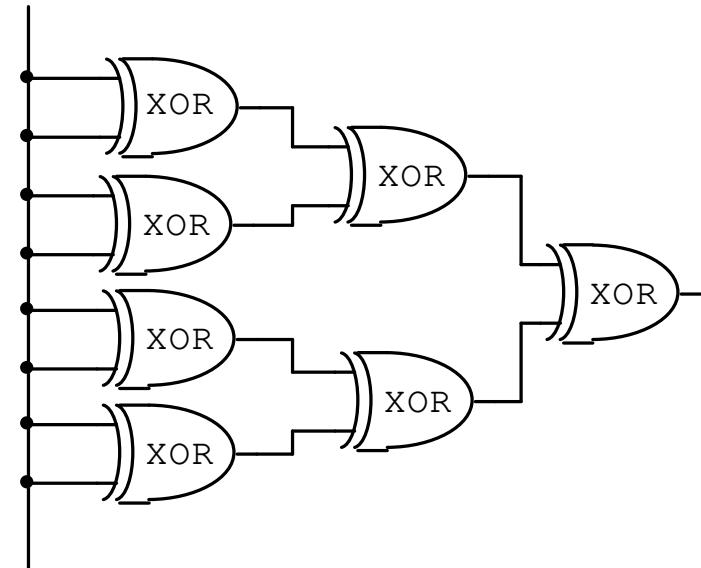
$$\begin{array}{l} 0001 \rightarrow 0 \oplus 0 \oplus 0 \oplus 1 = 1 \\ 0110 \rightarrow 0 \oplus 1 \oplus 1 \oplus 0 = 0 \\ 1110 \rightarrow 1 \oplus 1 \oplus 1 \oplus 0 = 1 \end{array}$$

Páros paritás!



# Paritás bit ellenőrzés

- Páros v. páratlan paritás: N bites információ egy kiegészítő bittel bővül → *egyszeres hiba felismerése*
- Hibák lehetséges okai:
  - Ieragadásból: '0'-ból '1'-es lesz, vagy fordítva
  - ideiglenes, tranziens jellegű hiba
  - áthallás (crosstalk)
- 8-adatbithez páros paritásbit generálás  
(IC 74'180. <http://alldatasheet.com> 9 bites paritás ellenőrző)  
(XOR gate IC 74'86)

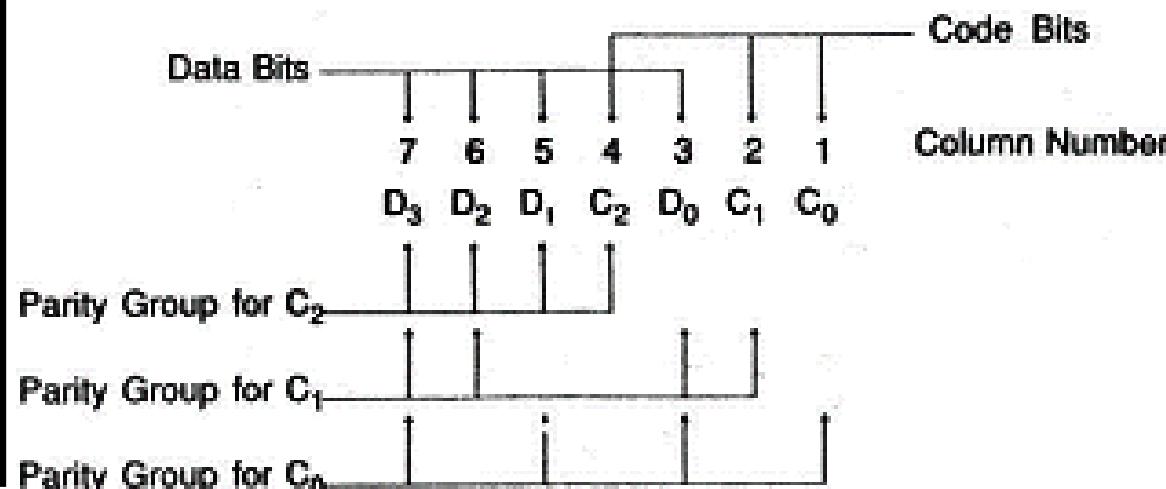


# Hamming kód

- Háttér: több *redundáns* bittel nemcsak a hiba meglétét, és helyét tudjuk detektálni, hanem akár a hibás bitet javítani is tudjuk
- **Hamming kód**: egy biten tároljuk a bitmintázatok azonos helyiértékű bitjeinek különbségét, tehát egy bites hibát lehet vele *javítani*.
  - Előny:
    - Bitcsoportokon történő paritás ellenőrzésen alapul: gyors művelet
    - Egy-szeres hibát tud javítani (adatokon)
    - SEC-DED: Single Error Correction / Double Error Detection
  - Hátrány:
    - Csoportos hibát nem képes javítani, ill. több egy-bites hibát sem
    - Adatbitekhez képest „túl sok” a javító ún. kód bitek száma

# 7-bites Hamming kódú kódszó konstruálása (pl. 4 –bites adatszóra)

- **$2^N-1$  bites Hamming kód: N kódbit  $\rightarrow 2^N-N-1$  adatbit**
- Összesen pl. 7 biten **4 adatbitet ( $D_0, D_1, D_2, D_3$ )**, **3 kódbittel ( $C_0, C_1, C_2$ )** kódolunk (LE!)
- $C_i$  kódbitek a bináris súlyuknak megfelelő bitpozíciókban
- A maradék pozíciókat rendre adatbitekkel töltjük fel ( $D_i$ ).



Paritás-csoportok	Bit pozíciók	Bitek jelölései
0	1, 3, 5, 7	$C_0, D_0, D_1, D_3$
1	2, 3, 6, 7	$C_1, D_0, D_2, D_3$
2	4, 5, 6, 7	$C_2, D_1, D_2, D_3$

# Pl. 1/a) Hamming kódú hibajavító áramkör tervezése (LittleEndian)

- Példa: bemeneti adatbit-mintázatunk 0101 (D<sub>3</sub>-D<sub>0</sub>). **LE!**
- 4 adatbit → 3 kódbitünk van
  - Alkalmazzunk páratlan paritást! A megfelelő helyen szereplő kódbitekkel kiegészítve a következő szót kapjuk: 0100110. Ha nincs hiba, a paritásellenőrzők (C<sub>2</sub>, C<sub>1</sub>, C<sub>0</sub>) kimenete '000' (**nem-létező bitpozíciót azonosít, azaz nincs hiba**), így megegyezik a kódolt mintázat paritásbitjeinek értékével, minden egyes paritáscsoportra (küldött és az azonosított Ci-minták bitenkénti XOR kapcsolata).
  - **Hiba szindróma:** Hiba esetén például, tfh. az input mintázat 0100010 - re változik, ekkor a vevő oldali paritásellenőrző hibát észlel. Ugyan C<sub>2</sub> paritásbitcsoport rendben ('0'), DE a C<sub>1</sub> ('0') és C<sub>0</sub> ('1') változott, tehát hiba van:
    - Ekkor **011** = 3 az azonosított minta, ami a 3. oszlopot jelenti (→ **D0** helyén).
    - Javításként *invertálni kell* a 3. bitpozícióban lévő bitet. 0100010 ⇒ 0100110. Ekkor a kódbitek a következőképpen módosulnak a páratlan paritásnak megfelelően: C<sub>2</sub>=0, C<sub>1</sub>=1 és C<sub>0</sub>=0.

# Pl. 1/a) folyt. Hamming kódú kódszó (LittleEndian)

- 4 adatbithez ( $D_3-D_0=0101 \rightarrow 3$  paritásbit ( $C_2-C_0$ )
  - azaz 7-bites Hamming kódú hibajavító kódszó

7	6	5	4	3	2	1	poz
D3	D2	D1	C2	D0	C1	C0	Error syndrome
0	1	0	?	1	?	?	
0	1	0	0	1	1	0	
	.		.		.		
		.	.				

	C2	C1	C0
Adó	0	1	0
Vevő	0	1	0
XOR	0	0	0

Azaz 000 = 0. pozíció  
(nem létezik,  
tehát nincsen hiba)

# Pl. 1/a) folyt. Hamming kódú kódszó (LittleEndian)

- Tfh. van **hiba**, a D0 megváltozik  $'1' \rightarrow '0'$

7	6	5	4	3	2	1	poz
D3	D2	D1	C2	D0	C1	C0	Error syndrome
0	1	0	?	0	?	?	
0	1	0	0	0	0	1	
	.		.		.		
		.	.				

$$\begin{array}{r} & C_2 & C_1 & C_0 \\ \text{Adó} & 0 & 1 & 0 \\ \text{Vevő} & 0 & 0 & 1 \\ \hline \text{XOR} & 0 & 1 & 1 \end{array}$$

Azaz  $011 = 3.$  pozíció  
(tehát D0 a hibás!)

**Javítás** = hibás D0  
invertálása  $'0' \rightarrow '1'$

# Pl. 1/b) Hamming kódú hibajavító áramkör tervezése (Big Endian)

- Példa: bemeneti adatbit-mintázatunk 0101 ( $D_0$ - $D_3$ ). **BE!**
- 4 adatbit  $\rightarrow$  3 kódbitünk van
  - Alkalmazzunk páratlan paritást! A megfelelő helyen szereplő kódbitekkel kiegészítve a következő szót kapjuk: 1001101. Ha nincs hiba, a paritásellenőrzők ( $C_0$ ,  $C_1$ ,  $C_2$ ) kimenete '000' (nem-létező bitpozíciót azonosít, azaz nincs hiba), így megegyezik a kódolt mintázat paritásbitjeinek értékével, minden egyes paritáscsoportra (azaz a küldött és azonosított  $C_i$ -minták bitenkénti XOR kapcsolata).
  - **Hiba szindróma:** Hiba esetén például, tfh. az input mintázat 1011101 - re változik, ekkor a vevő oldali paritásellenőrző hibát észlel. Ugyan  $C_2$ . paritásbitcsoport rendben ('1'), DE a  $C_1$  ('1') és  $C_0$  ('0') változott, tehát hiba van:
    - Ekkor (110) azaz  $011 = 3$  az azonosított minta, ami a 3. oszlopot jelenti ( $\rightarrow D_0$  helyén).
    - Javításként *invertálni kell* a 3. bitpozícióban lévő bitet. 1011101  $\Rightarrow$  1001101. Ekkor a kódbitek következőképpen módosulnak a páratlan paritásnak megfelelően:  $C_0=1$ ,  $C_1=0$  és  $C_2=1$ .

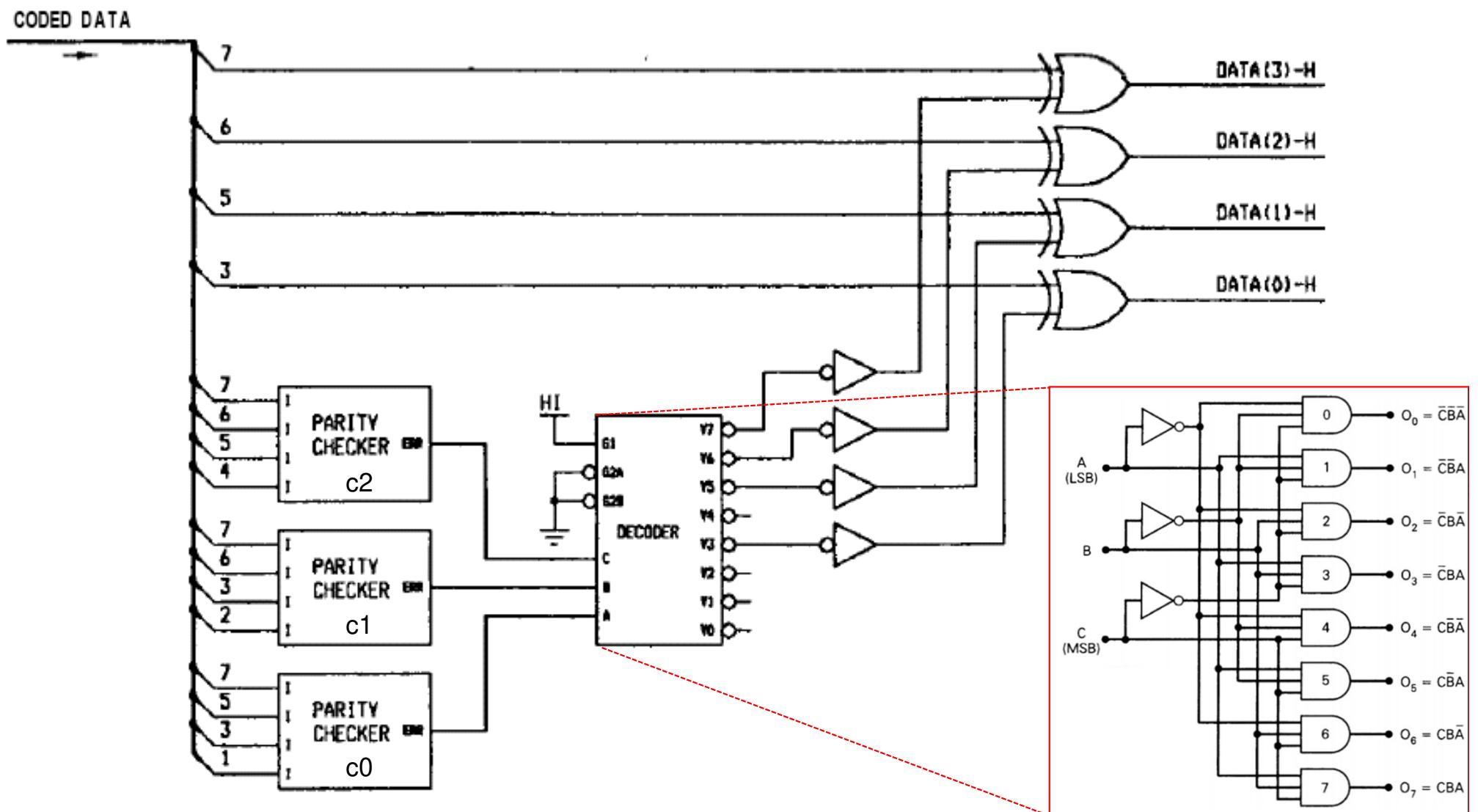
# Pl 2.) Hamming kódú hibajavító áramkör tervezése (Little Endian)

- Példa: bemeneti adatbit-mintázatunk 0101 (D<sub>3</sub>-D<sub>0</sub>). LE!
- 4 adatbit → 3 kódbitünk van
  - Alkalmazzunk **páros paritást!** A megfelelő helyen szereplő kódbitekkel kiegészítve a következő szót kapjuk: 0101101. Ha nincs hiba, a paritásellenőrzők (C<sub>2</sub>, C<sub>1</sub>, C<sub>0</sub>) kimenete '000' (**nem-létező bitpozíciót azonosít**). Igy megegyezik a kódolt mintázat paritásbitjeinek értékével, minden egyes paritáscsoportra (küldött és vett Ci-k bitenkénti XOR kapcsolata).
  - Hiba esetén például, ha az input mintázat 0111101 -re változik, ekkor a paritásellenőrző hibát észlel. Két paritásbit ellenőrző megváltozott: C<sub>2</sub> ('0'), a C<sub>0</sub> ('0'), de C<sub>1</sub> ('0') változatlan.
    - Ekkor 101 = 5, az azonosított minta, ami a 5. oszlopot jelenti (→ **D1** helyén).
  - Javításként *invertálni kell* a 5. bitpozícióban lévő bitet. 0111101 ⇒ 0101101. Ekkor a kódbitek a következőképpen módosulnak a páratlan paritásnak megfelelően: C<sub>2</sub>=1, C<sub>1</sub>=0 és C<sub>0</sub>=1.

# Pl 3.) Hamming kódú hibajavító áramkör tervezése (Little Endian)

- Példa: bemeneti adatbit-mintázatunk 0101 ( $D_3-D_0$ ). LE!
- 4 adatbit  $\rightarrow$  3 kódbitünk van
  - Alkalmazzunk páratlan paritást! A megfelelő helyen szereplő kódbitekkel kiegészítve a következő szót kapjuk: 0100110. Ha nincs hiba, a paritásellenőrzők ( $C_2$ ,  $C_1$ ,  $C_0$ ) kimenete '**000**' (**nem-létező bitpozíciót azonosít, azaz nincs hiba**), így megegyezik a kódolt mintázat paritásbitjeinek értékével, minden egyes paritáscsoportra (küldött és vett  $C_i$ -k bitenkénti XOR kapcsolata).
  - Hiba esetén például, tfh. az input mintázat 0100100 -re változik, akkor a paritásellenőrző hibát észlel.  $C_2$  paritásbit ellenőrző változatlan ('0'), és a  $C_0$  is ('0'), DE a  $C_1$  ('0') hibát észlel, tehát:
    - Ekkor **010** = 2 az azonosított minta önmaga, ami a 2. oszlopot jelenti ( $\rightarrow$  **C1** paritásbit! helyén).
  - Javításként itt már a dupla hibaellenőrzést (DEB / vagy SECDEC) kell alkalmazni, amely a **paritásbiteket is kódolja**.

# 7-bites Hamming kódú hibajavító áramkör felépítése



# Példa: SEC-DED-dupla paritáshiba ellenőrzés (LittleEndian)

DEB: extra bit, a teljes kódszóra vonatkozóan (Ci-ket is kódolja)

Hamming kód (DEB-el) 8 adatbitre: mi a helyes ábrázolása 8 biten a 01011100 adatbit mintázatnak. Szükséges 8 adatbit (D0-D7), 4 kódbit (C0-C3) és egy kettős hibajelző bit (DEB). Páratlan paritást alkalmazunk. (BW-binary weight jelenti az egyes oszlopok bináris súlyát, 1,2, 4 ill 8 biten).

13	12	11	10	9	8	7	6	5	4	3	2	1	Oszlopszám
DEB	D7	D6	D5	D4	C3	D3	D2	D1	C2	D0	C1	C0	
	1	1	1	1	1	0	0	0	0	0	0	0	BW, 8bit
	1	0	0	0	0	1	1	1	1	0	0	0	BW, 4 bit
	0	1	1	0	0	1	1	0	0	1	1	0	BW, 2 bit
	0	1	0	1	0	1	0	1	0	1	0	1	BW, 1 bit

Paritáscsoportok	Bit pozíciók	Bitek jelölései
0	1, 3, 5, 7, 9 ,11	C0, D0, D1, D3, D4, D6
1	2, 3, 6, 7, 10, 11	C1, D0, D2, D3, D5, D6
2	4, 5, 6, 7, 12	C2, D1, D2, D3, D7
3	8, 9, 10, 11, 12	C3, D4, D5, D6, D7

13	12	11	10	9	8	7	6	5	4	3	2	1	Oszlopszám
DEB	D7	D6	D5	D4	C3	D3	D2	D1	C2	D0	C1	C0	
	0	1	0	1		1	1	0		0			Adatbitek
	1	0	1	0	1	1	1	0	1	0	0	0	Hozzáadott

kódbitek. Tehát a helyes ábrázolása 01011100-nek a következő:

101011101000.



# Számítógép Architektúrák II.

(MIVIB344ZV)

3. előadás: Aritmetikai műveletvégző egységek –  
összeadás, kivonás

Előadó: Dr. Vörösházi Zsolt  
[voroshazi.zsolt@mik.uni-pannon.hu](mailto:voroshazi.zsolt@mik.uni-pannon.hu)

# Jegyzetek, segédanyagok:

- Könyvfejezetek:

- <http://www.virt.uni-pannon.hu> → Oktatás → Tantárgyak → Számítógép Architektúrák II.  
□ (chapter03.pdf)

- Fóliák, óravázlatok .ppt (.pdf)

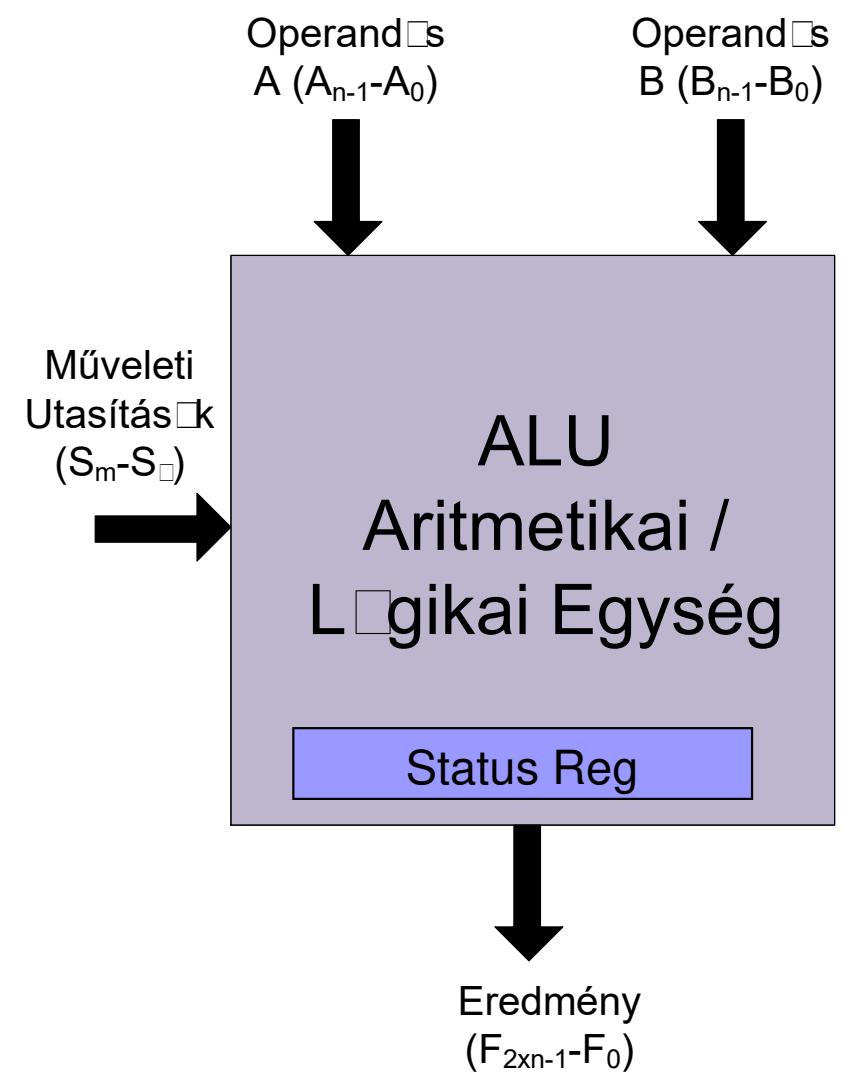
- Feltöltésük folyamatosan

# Ismétlés

- Korai számítógépek teljesítményét főként ballisztikus számításoknál (hadászatban)
- Információ ábrázolás
- Használt utasításkészlet (RISC vs. CISC)
- Adatkezelő / műveletvégző egység: Alapvető ALU (Aritmetikai és Logikai funkciók)
  - Univerzális / funkcionális teljesség
    - Aritmetikai operátorok:  $+$  →  $-$ ,  $*$ ,  $/$  (alapműveletek)
    - Logikai operátorok:  
**NAND, NOR** → NOT, AND, OR, XOR (AV), NXOR (EQ) – mai *CMOS VLSI technológia* esetén

# ALU felépítése

- Utasítások hatására a ( $S_m-S_0$ ) „vezérlőjelek” jelölik ki a végrehajtandó aritmetikai / logikai műveletet.
- További adatvonalak kapcsolódhatnak közvetlenül a **statusz regiszterhez**, amelyben fontos információkat tárolunk: pl.
  - zero bit
  - carry-in, carry-out átviteleket,
  - előjel bitet (sign),
  - túlcsordulást (overflow), vagy alulcsordulást (underflow) jelző biteket,
  - Paritás, stb.

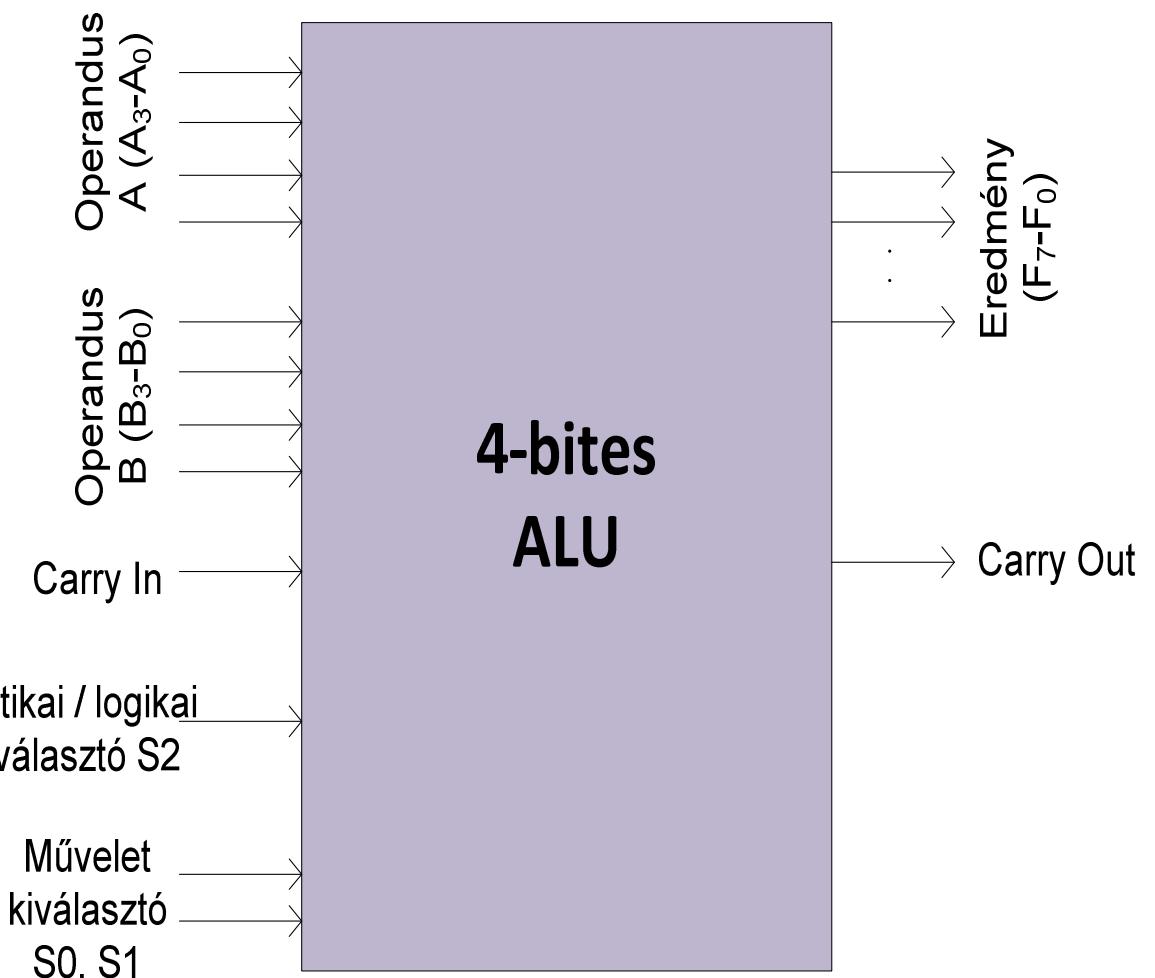


# Státusz- (flag) jelzőbitek

- Az aritmetikai műveletek eredményétől függően hibajelzésre használatos jelzőbitek. Ezek megváltozása az utasításkészletben előre definiált utasítások végrehajtásától függ.
  - a.) Előjelbit (sign): 2's komplement (MSB)
  - b.) Átvitel kezelő bit (carry in/out): helyiértékes átvitel
  - c.) Alul / Túlcordulás jelzőbit (underflow / overflow)
  - d.) Zero bit: kimeneten az eredmény 0-e?
    - Pl: 0-val való osztás!
    - (szorzásnál egyszerűsíthetőség – adatfüggés)
  - e.) Paritás bit: páros, páratlan
  - ...

# Példa: N=4-bites ALU felépítése és működése

- Két N=4-bites operandus (A, B)
- Eredmény (F) bitszélessége:
  - N+(1 CarryOut) bit, ha +;-
  - 2×N bites, ha \*
- H.értékes átvitel: CarryIn/ Out
- S2: Aritmetikai/ logikai mód választó (MUX)
- S0, S1: művelet kiválasztó (S2 értékétől függően)
  - Aritmetikai vs. Logikai

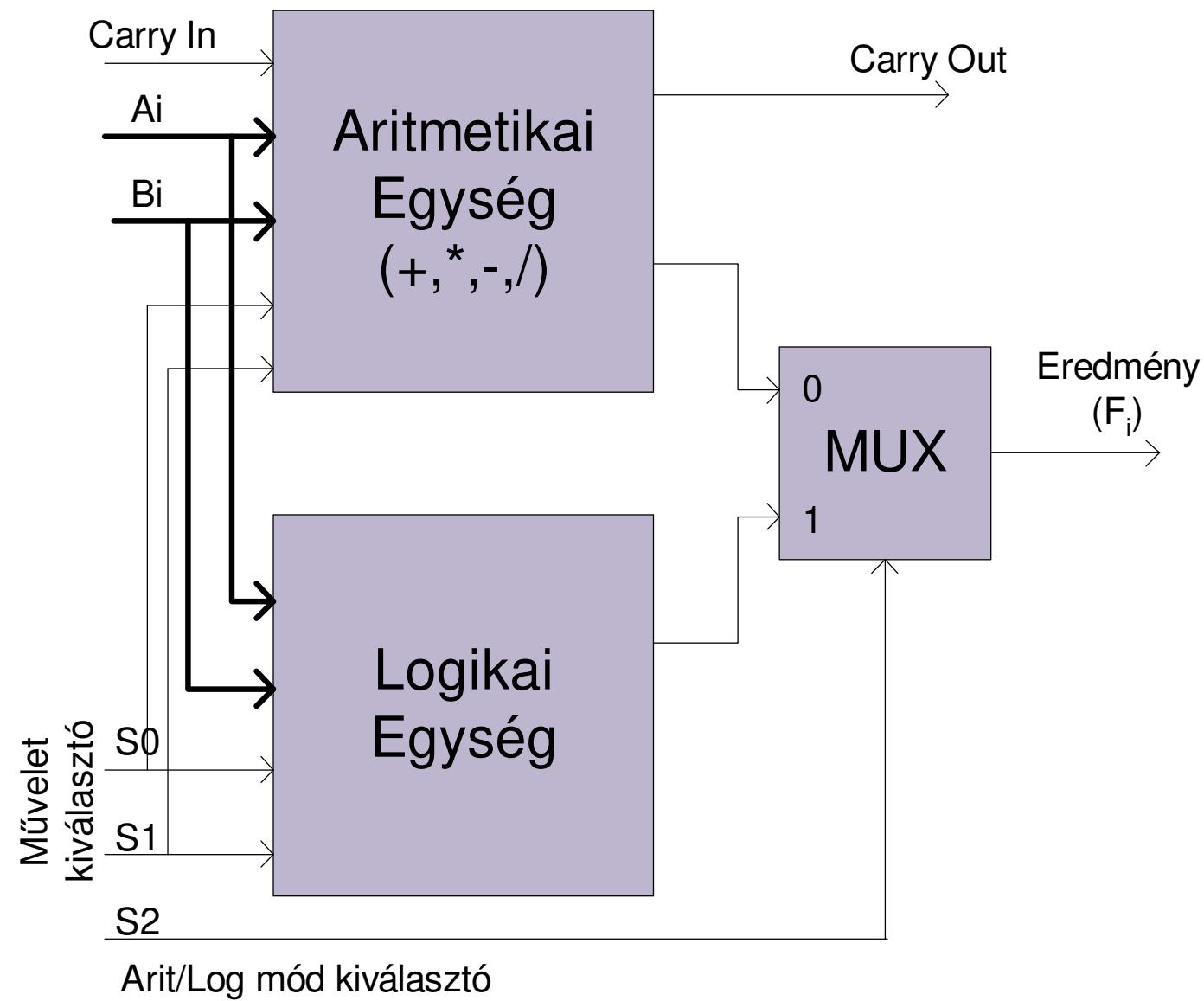


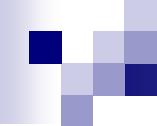
# ALU működését leíró függvénytáblázat

(egy lehetséges működés, a funkciók bővíthetők):

Művelet kiválasztás:				Művelet:	Megvalósított függvény:
S2	S1	S0	Cin		
0	0	0	0	F=A	‘A’ átvitele
0	0	0	1	F=A+1	‘A’ értékének növelése 1-el (increment)
0	0	1	0	F=A+B	Összeadás
0	0	1	1	F=A+B+1	Összeadás carry figyelembevételével
0	1	0	0	$F = A + \bar{B}$	A + 1's komplementens B
0	1	0	1	$F = A + \bar{B} + 1$	Kivonás = 2's összeadás!
0	1	1	0	F=A-1	‘A’ értékének csökkentése 1-el (decrement)
0	1	1	1	F=B	‘B’ átvitele
1	0	0	x	$F = A \wedge B$	AND
1	0	1	x	$F = A \vee B$	OR
1	1	0	x	$F = A \oplus B$	XOR
1	1	1	x	$F = \bar{A}$	‘A’ negáltja (NOT A)

# ALU felépítése:





# Lebegőpontos műveletvégző egységek

# Lebegőpontos műveletvégző egységek

## ■ Probléma:

- Mantissa igazítás → Exponens beállítás
- Normalizálás, Utó-(Post) Normalizálás
  - (DEC-32, IEEE-32, IBM-32)

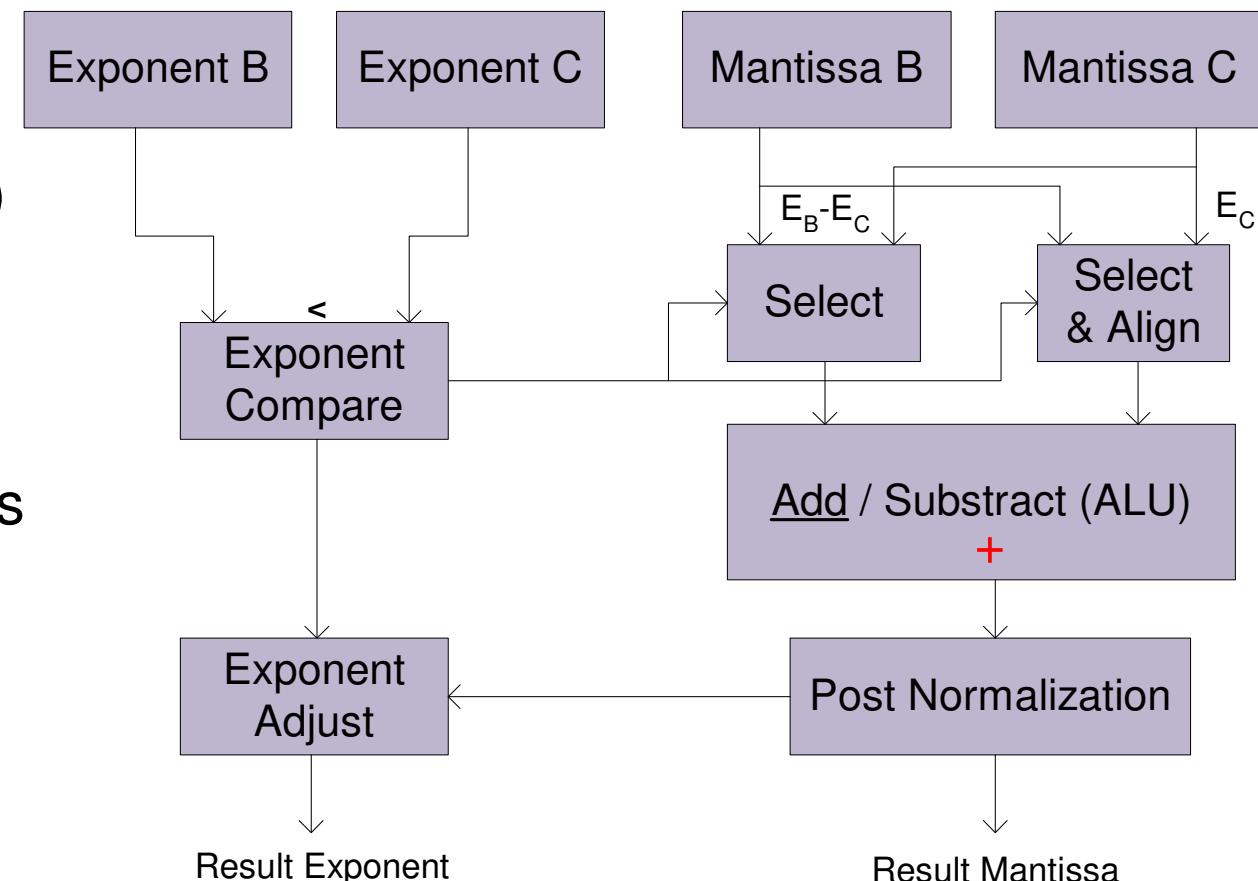
## ■ Műveletvégző elemek:

- Összeadó (adder)-,
- Kivonó (subtractor)-,
- Szorzó (multiplier)-,
- Osztó (divider) áramkörök.

# a.) Lebegőpontos összeadó

■ Művelet:  $A = B + C = M_B \times r^{E_B} + M_C \times r^{E_C} = (M_B \times r^{|E_B - E_C|} + M_C) \times r^{E_C}$

- Komplex feladat: a mantisszák hosszát egyeztetni kell (MSB bitek azonos helyiértéken legyenek)
- Legyen:  $0 < B < C$
- $B \rightarrow C$  vagyis  $|E_B - E_C|$  pozícióval jobbra igazítjuk az  $M_B$  mantisszát; ez változás az exponensben is
- ALU: Összeadás!: sign-magnitude formátumban
- Végül minimális post-normalizáció kell



# Példa: Lebegőpontos összeadás (kivonás)

$r_b=2$ , IEEE-754 (bináris 32-bites rendszer):

■  $A = B + C = 10.0001 + 1101.1 = //B < C//$

$$(1.00001 \times 2^1) + (1.1011 \times 2^3) =$$

$$(\underline{0.0100001} \times 2^3) + (1.1011 \times 2^3) =$$

$$(0.0100001 + 1.1011) \times 2^3 =$$

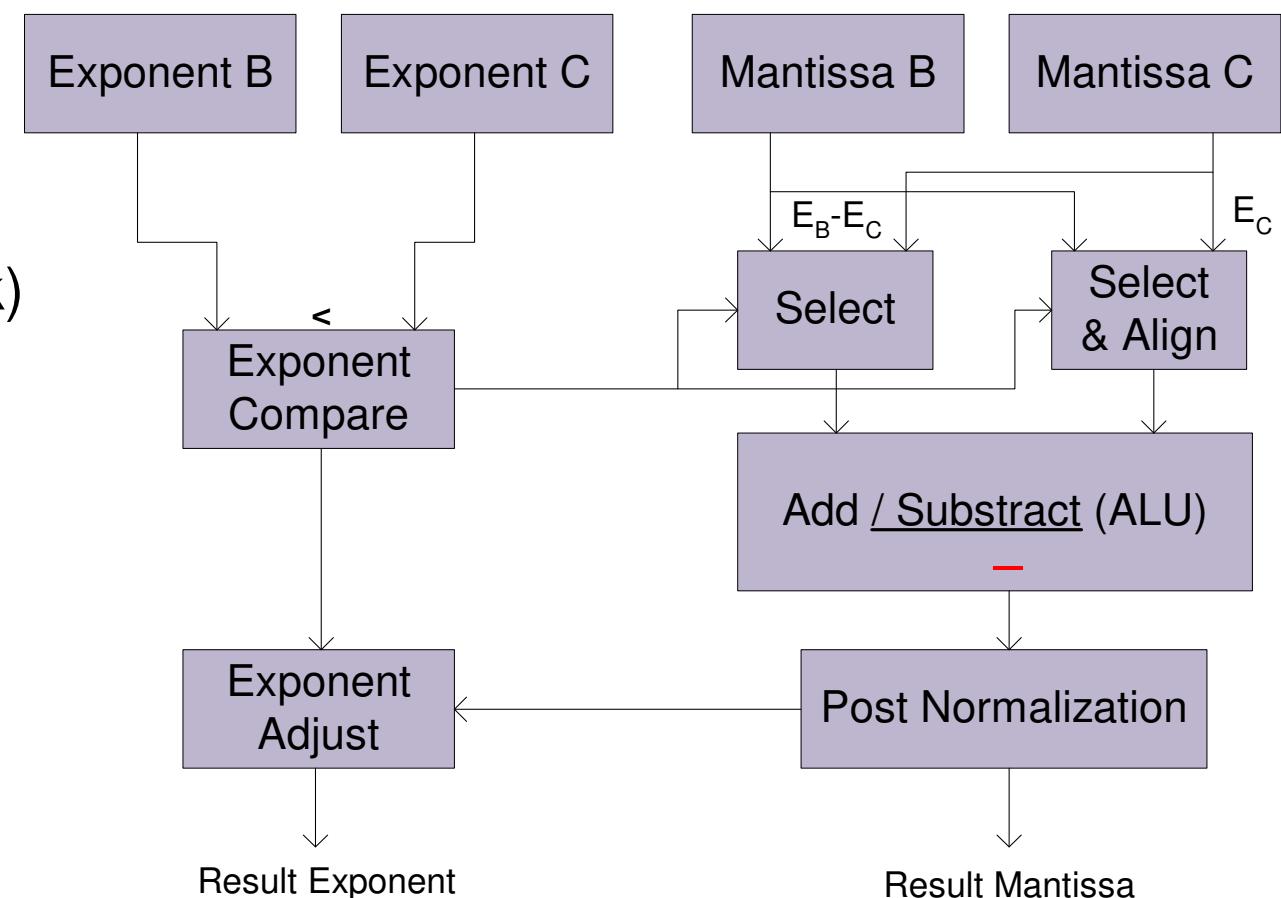
■  $1.1111001 \times 2^3 = A$

(itt éppen nem kell post-normalizálás)

# b.) Lebegőpontos kivonó

■ Művelet:  $A=B-C=M_B \times r^{E_B} - M_C \times r^{E_C} = (M_B \times r^{|E_B-E_C|} - M_C) \times r^{E_C}$

- Komplex feladat: a mantisszák hosszát egyeztetni kell (MSB bitek azonos helyiértéken legyenek)
- Legyen:  $0 < B < C$
- $B \rightarrow C$  vagyis  $|E_B - E_C|$  pozícióval jobbra igazítjuk az  $M_B$  mantisszát, ez változás az exponensben is
- Kivonás! (ALU)

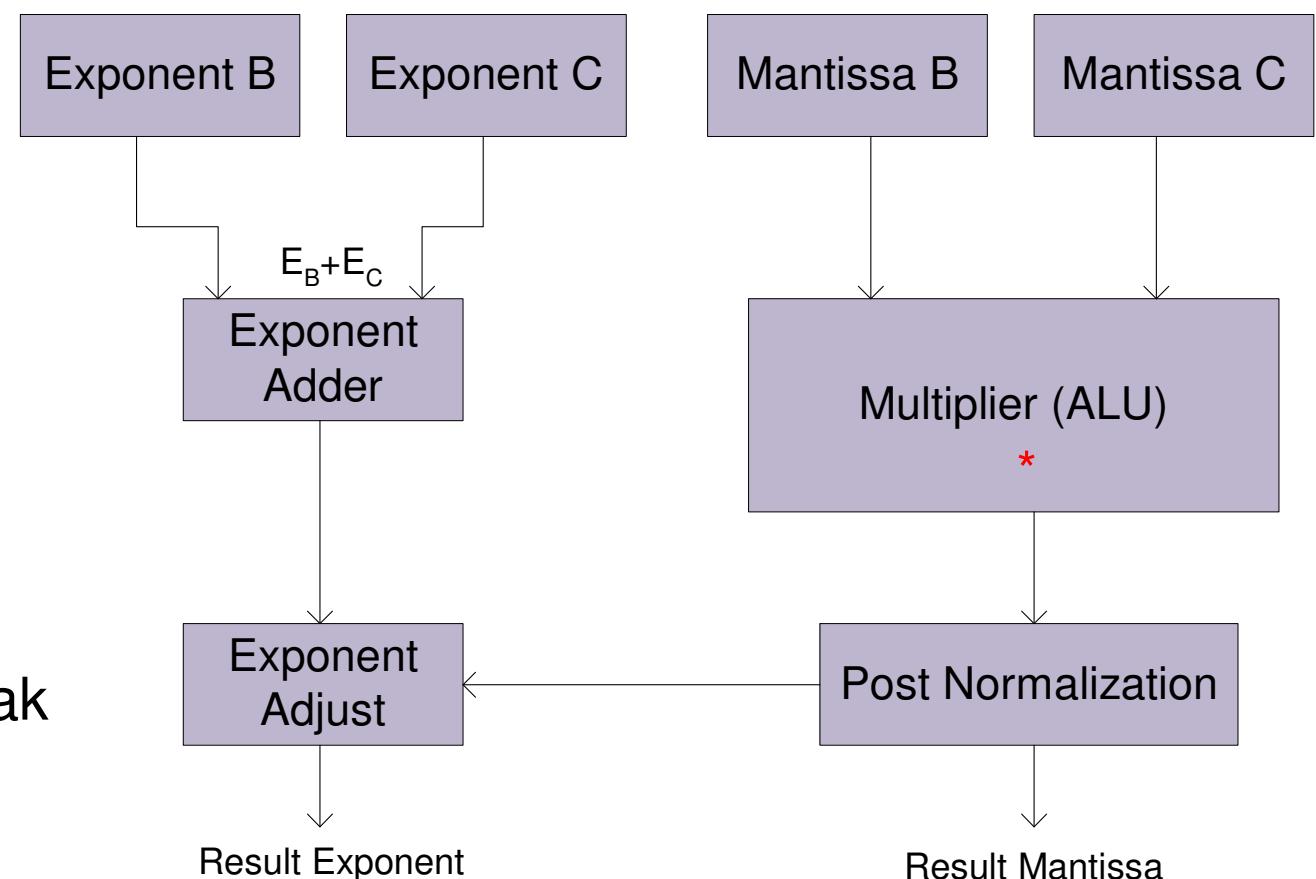


# c.) Lebegőpontos szorzó

■ Művelet:  $A = B \times C = M_B \times r^{E_B} \times M_C \times r^{E_C} = (M_B \times M_C) \times r^{E_B + E_C}$

- A: szorzat
- B: szorzandó
- C: szorzó

- Könnyű végrehajtani
- Nincs szükség az operandusok beállítására
- Minimális post-normalizációt kell csak végezni
- ALU: szorzás!

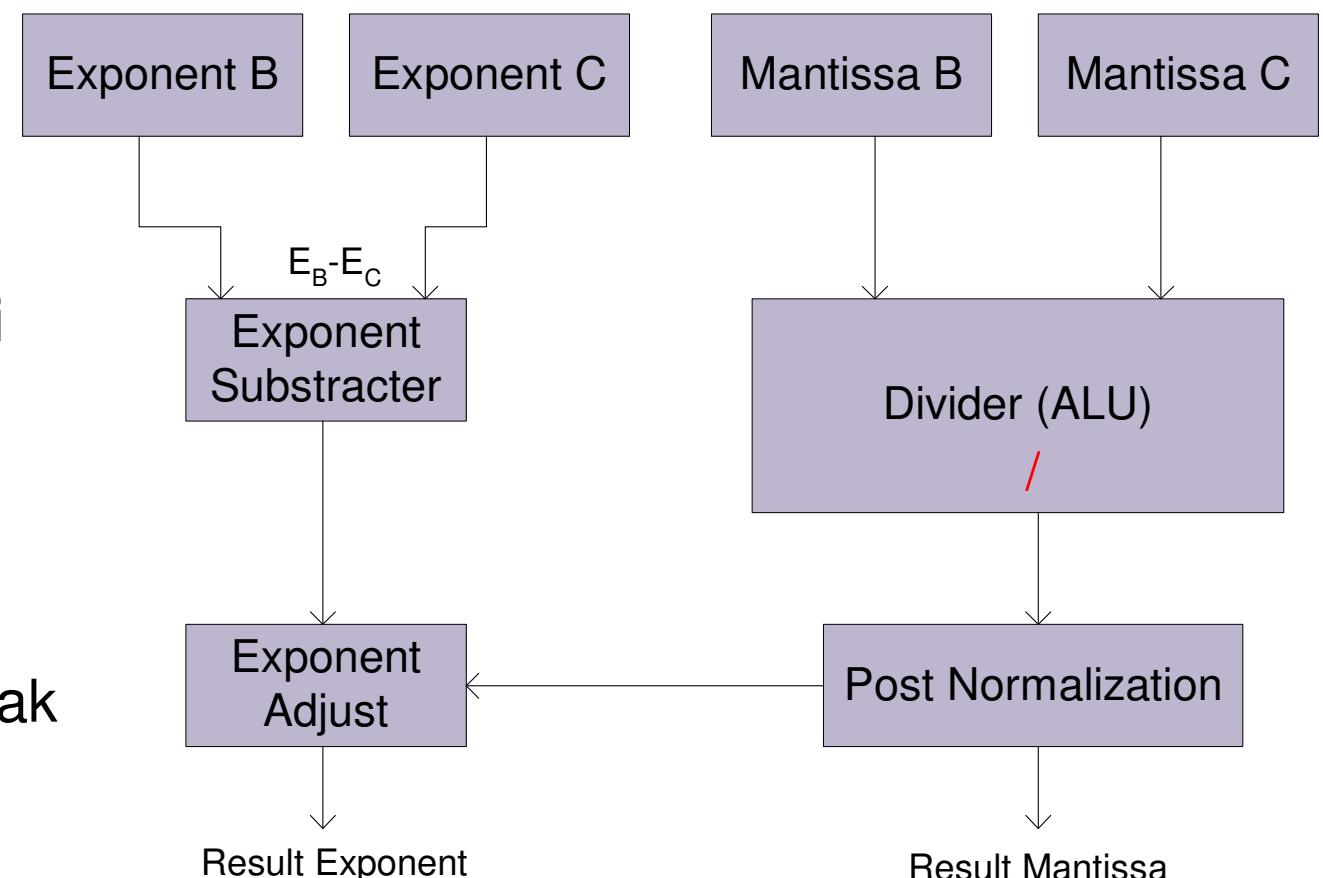


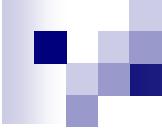
# d.) Lebegőpontos osztó

■ Művelet:  $A=B/C=M_B \times r^{E_B} / M_C \times r^{E_C} = (M_B / M_C) \times r^{E_B - E_C}$

- A: hányados
- B: osztandó
- C: osztó

- Könnyű végrehajtani
- Nincs szükség az operandusok beállítására
- Minimális post-normalizációt kell csak végezni
- Osztás! (ALU)





# Összeadó áramkörök

# a.) Fél-összeadó – Half Adder

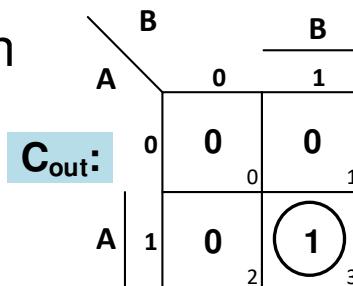
## ■ HA: 1-bites Half Adder

igazságtáblázat

Ai	Bi	Cout	Si
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

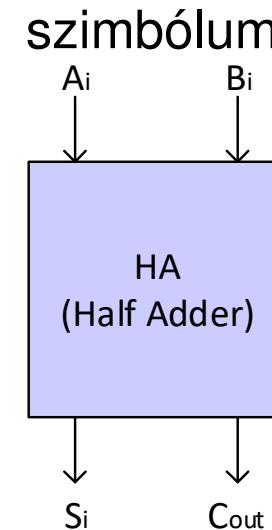
Nem kezeli a Cin-t !  
 $A + B = \text{Cout} | S$

Karnaugh  
táblái:

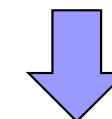
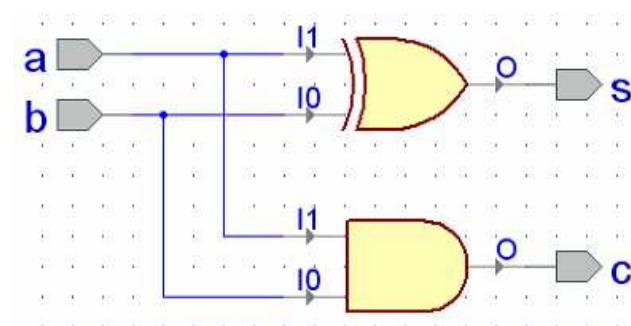


Kimeneti  
fgv-ai:

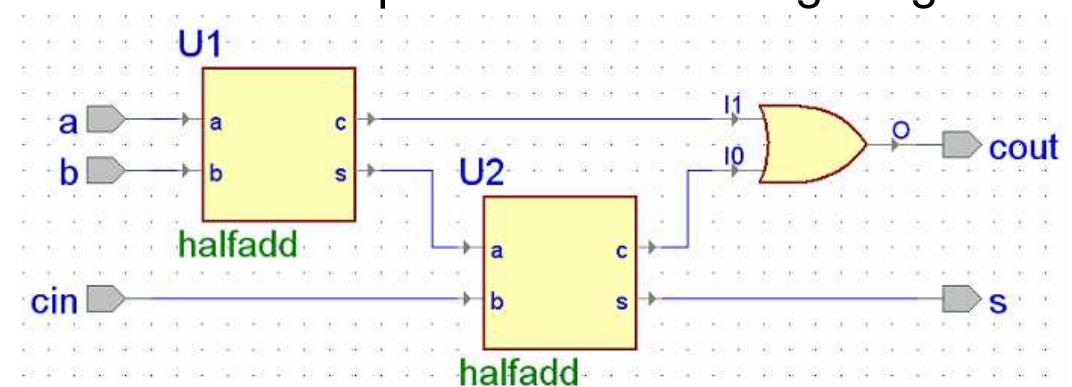
$$C_{out} = A_i \cdot B_i$$



$$T_{HA} = 1G$$



1 bites FA felépítése 2 db HA segítségével:



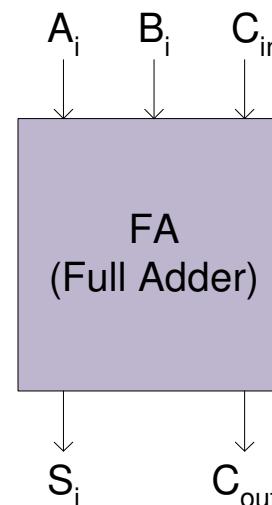
# b.) Teljes összeadó – Full Adder

## ■ FA: 1-bites Full Adder

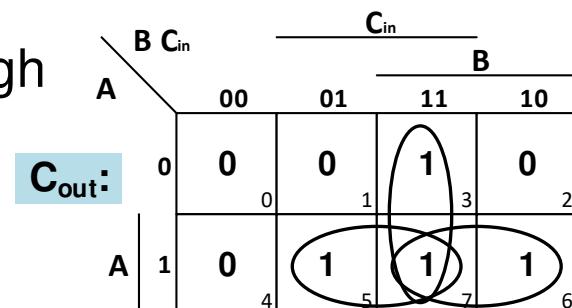
igazságtáblázat

$A_i$	$B_i$	Cin	Cout	Sum <sub>i</sub>
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

szimbólum



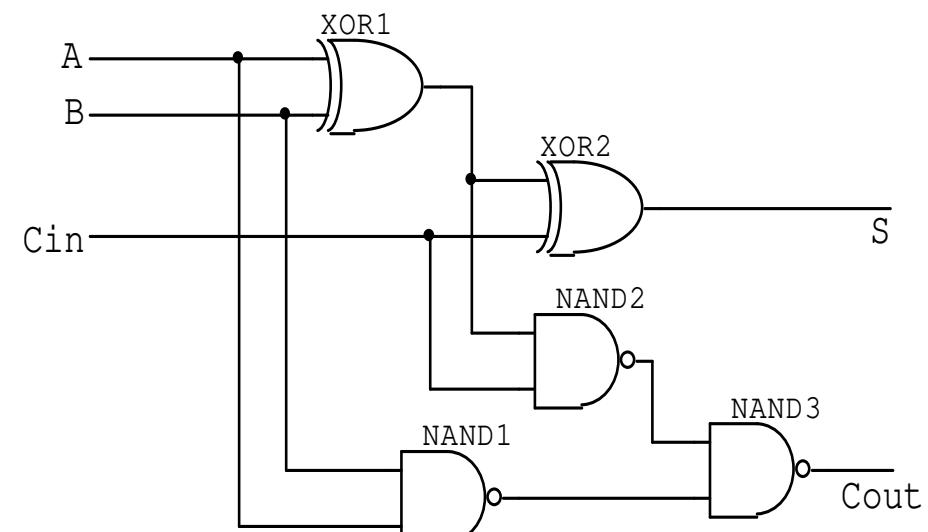
Karnaugh táblái:



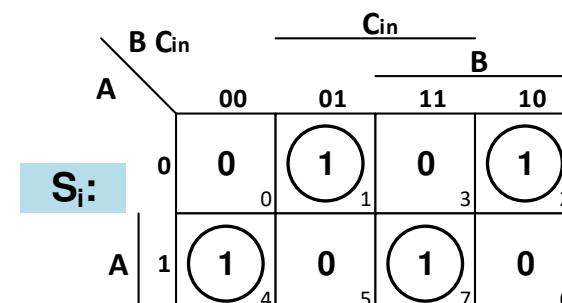
Kimeneti fgv-ei:

$$C_{out} = A_i \cdot B_i + A_i \cdot C_{in} + B_i \cdot C_{in}$$

Ez a FA egy lehetséges CMOS kapcsolási rajza: (itt  $T_{FA} = 3G$  !)



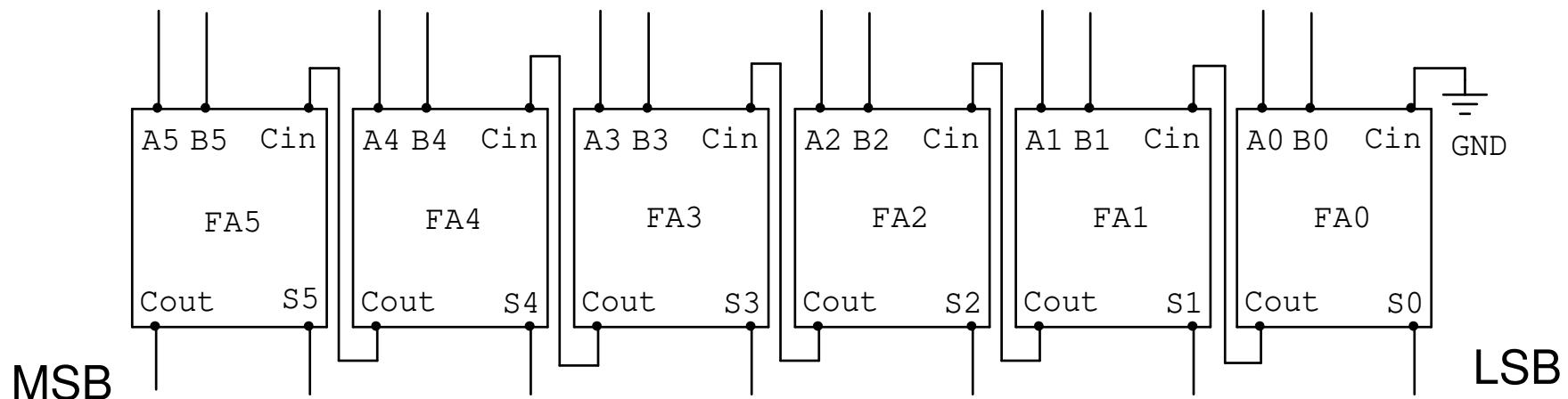
$$A + B + Cin = Cout | S$$



$$S_i = A_i \oplus B_i \oplus C_{in}$$

## c.) Átvitelkezelő összeadó – Ripple Carry Adder (RCA)

- Példa: 6-bites RCA: [5..0] (LSB Cin = GND!)



- Számítási időszükséglet (RCA):

$$T_{(RCA)} = N \cdot T_{(FA)} = N \cdot (2 \cdot G) = 12G \text{ (6-bites RCA esetén)}$$

ahol a min.  $2G$  az 1-bites FA kapukésleltetése ([ns], [ps])

# d.) LACA: Look Ahead Carry Adder

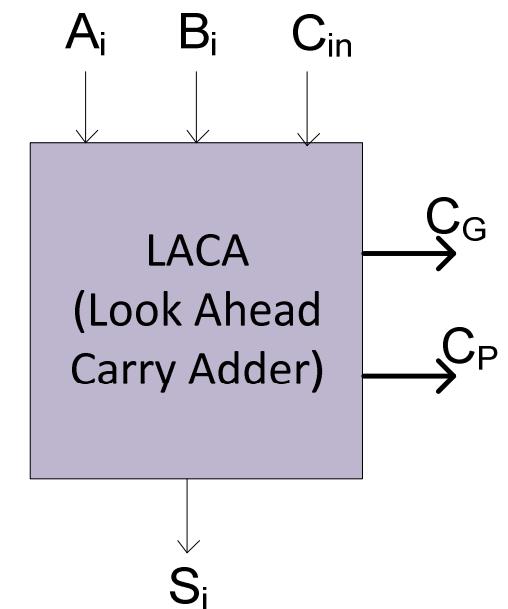
## Átvitelgyorsító összeadó

- Képlet (FA) átalakításából kapjuk:

$$C_{out} = A_i \cdot B_i + A_i \cdot C_{in} + B_i \cdot C_{in}$$

$$\Rightarrow \underbrace{A_i \cdot B_i}_{\text{CarryGenerate}} + C_{in} \cdot \underbrace{(A_i + B_i)}_{\text{CarryPropagate}} = C_G + C_{in} \cdot C_P$$

$$S_i = A_i \oplus B_i \oplus C_{in}$$



LACG: Look Ahead Carry Generator egy **b** bites ALU-hoz kapcsolódik, mindenegyes állapotban a Cin generálásáért felel a CP és CG (LACA-tól) érkező jeleknek megfelelően („*LACG looks at CP and CG from adders*”).

**N**-bites LACA számítási időszükséglete:  $T_{LACA} = 2 + 4 \times (\lceil \log_b(N) \rceil - 1)$

ahol **N**: bitek száma, **b**: LACG bitszélessége (hány LACA-hoz tartozik egy LACG)

# Megjegyzés: LACA – CG átírása XOR kapcsolatra (nem triviális forma)

- CG előállítása: alkalmazott másik érvényes forma a XOR kapcsolattal megadott kifejezés

igazságtáblázat

A <sub>i</sub>	B <sub>i</sub>	Cin	Cout	Sum <sub>i</sub>
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Karnaugh  
tábla:

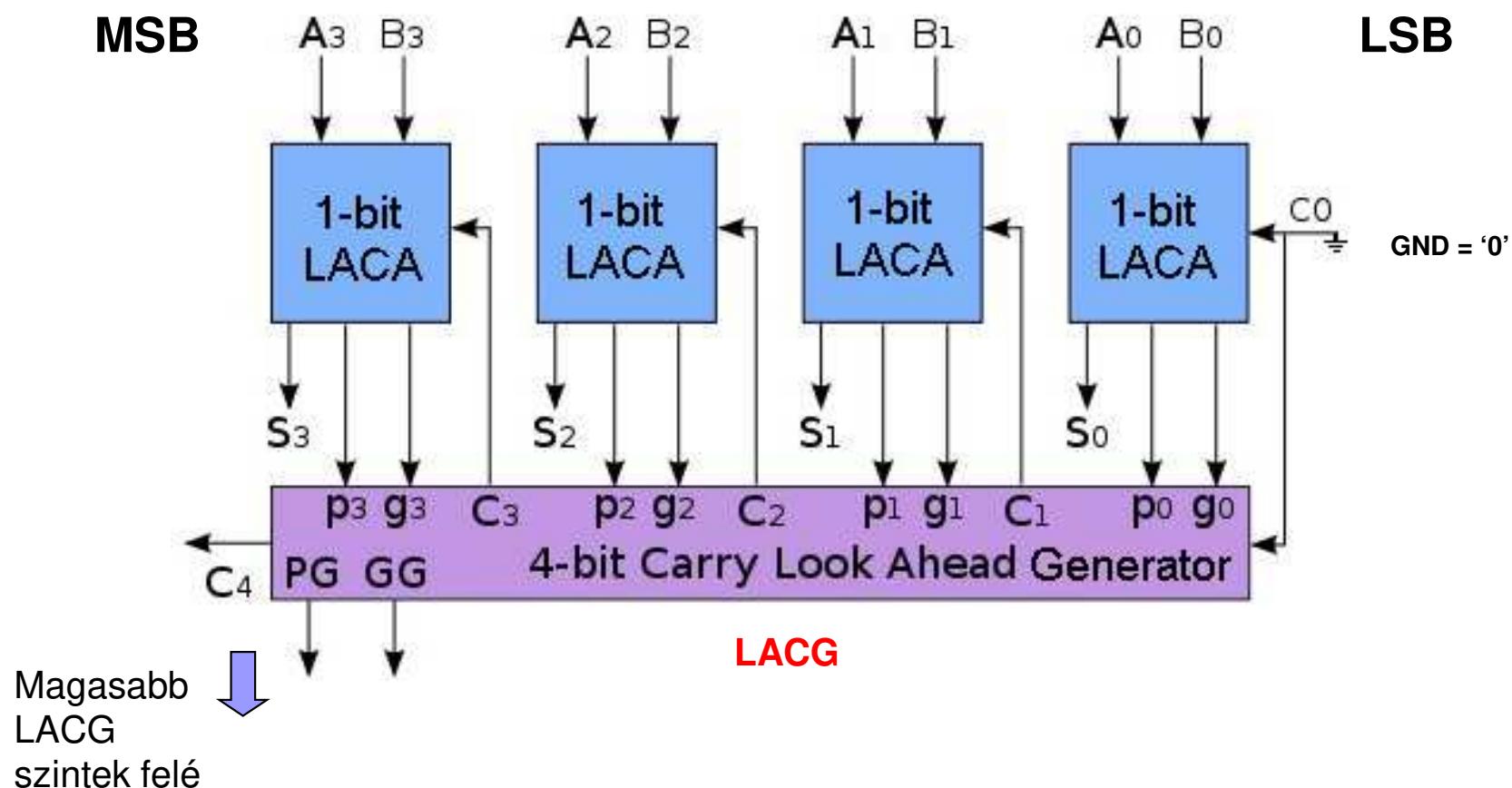
Kimeneti  
fgv:

		BC <sub>in</sub>		C <sub>in</sub>		B
		00	01	11	10	
A	0	0	0	1	0	B
	1	0	1	1	1	
C <sub>out</sub> :		0	0	1	2	
		1	4	5	6	

$$\begin{aligned} C_{out} &= \overline{A_i} \cdot B_i \cdot C_{in} + A_i \cdot \overline{B_i} \cdot C_{in} + A_i \cdot B_i = \\ &= A_i \cdot B_i + C_{in} \cdot (A_i \oplus B_i) = \\ &= C_G + C_{in} \cdot C_P \end{aligned}$$

# Példa: 4-bites LACA

- Legyen **b=4 (LACG)**, és **N=4 (LACA)**. Áramkör felépítése, és időszükséglete?



$$T_{LACA} = 2 + 4 \times (\underbrace{\log_4(4)}_1 - 1) = 2$$

# Példa (folyt.): 4-bites LACA számítási műveletei (carry terjesztés)

- LSB → MSB felé az összeadások

$$C_1 = G_0 + P_0 \cdot C_0$$

$$C_2 = G_1 + P_1 \cdot C_1$$

$$C_3 = G_2 + P_2 \cdot C_2$$

$$C_4 = G_3 + P_3 \cdot C_3$$

- Behelyettesítések: adott  $C_i$ -t → a  $C_{i+1}$ -be

$$C_1 = G_0 + P_0 \cdot C_0$$

$$C_2 = G_1 + G_0 \cdot P_1 + C_0 \cdot P_0 \cdot P_1$$

$$C_3 = G_2 + G_1 \cdot P_2 + G_0 \cdot P_1 \cdot P_2 + C_0 \cdot P_0 \cdot P_1 \cdot P_2$$

$$C_4 = G_3 + G_2 \cdot P_3 + G_1 \cdot P_2 \cdot P_3 + G_0 \cdot P_1 \cdot P_2 \cdot P_3 + C_0 \cdot P_0 \cdot P_1 \cdot P_2 \cdot P_3$$

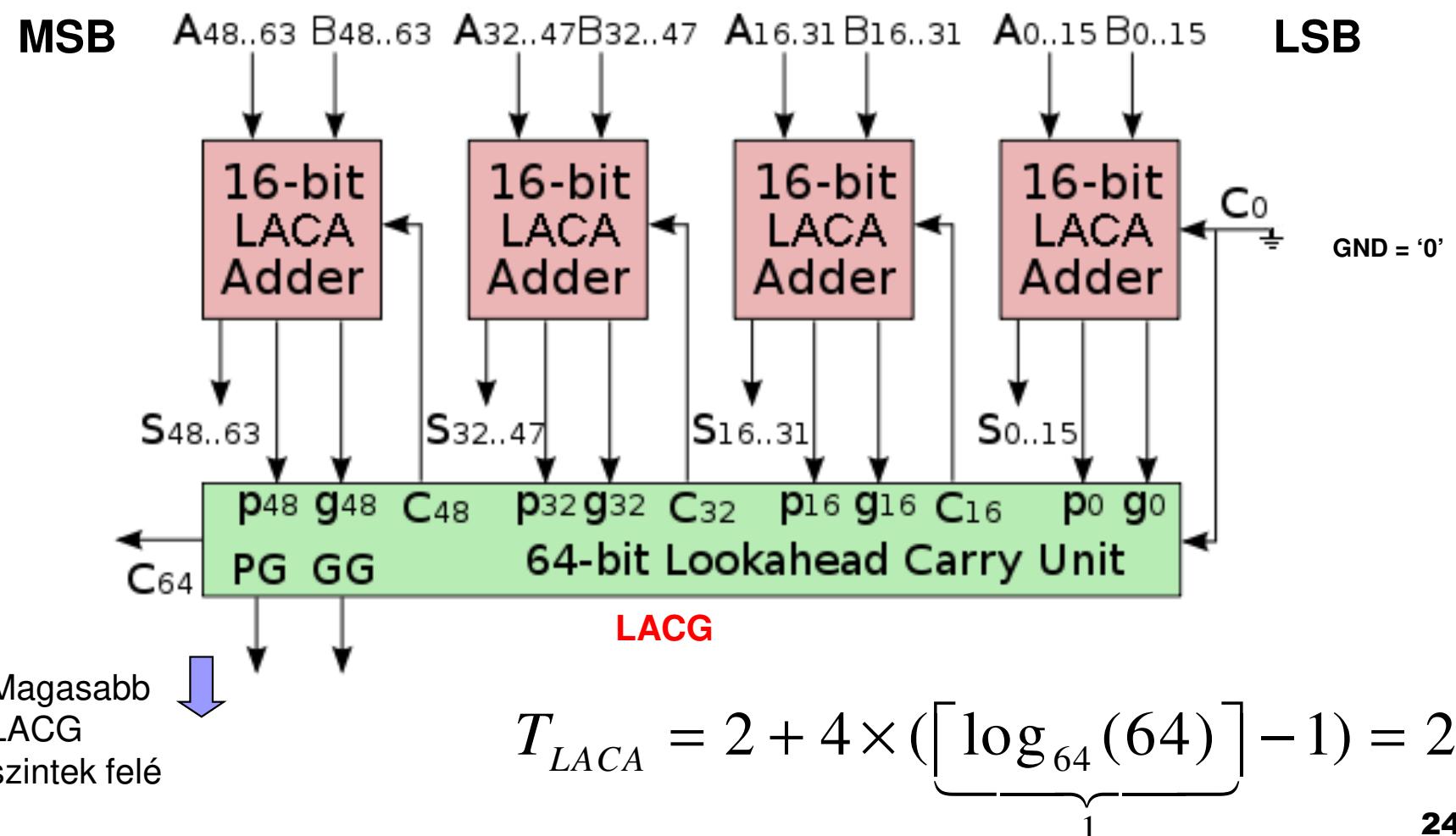
- Magasabb b-bites LACG hierarchia szintek felé GP (group propagate) és GG (group generate) számítása:

$$GG = G_3 + G_2 \cdot P_3 + G_1 \cdot P_3 \cdot P_2 + G_0 \cdot P_3 \cdot P_2 \cdot P_1$$

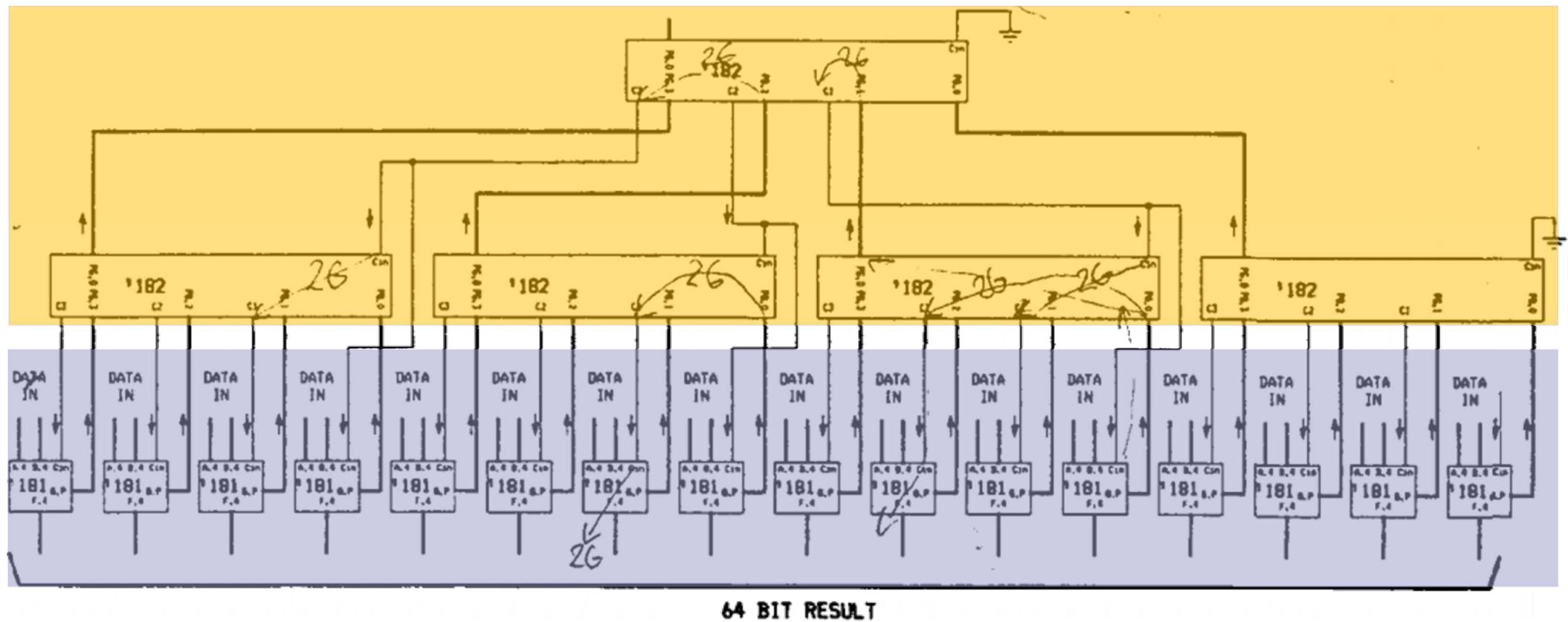
$$PG = P_0 \cdot P_1 \cdot P_2 \cdot P_3$$

# Példa: 4x16-bites LACA

- Legyen **b=64 (LACG)**, és **N=4x16 (LACA)**. Áramkör felépítése, és időszükséglete?



# 64-bites (16×4) LACA összeadó



$$T_{LACA} = 2 + 4 \times (\underbrace{\lceil \log_4(64) \rceil - 1}_{3}) = 10$$

- Komponensek:

- '182 = SN74LS181: **N=4**-bites ALU (összeadás)
- '181 = SN74LS182: **b=4** bites LACG generátor



# Kivonó áramkörök

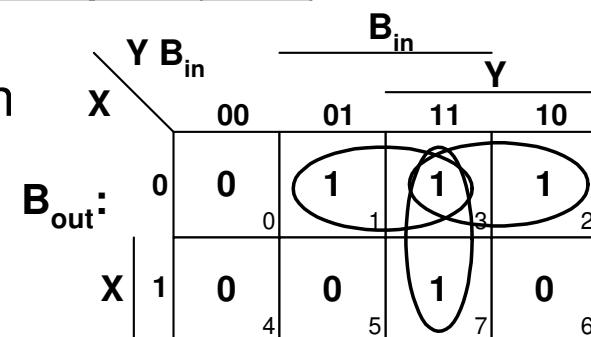
# Teljes kivonó - Full Subtractor (FS)

## ■ FS: 1-bites Full Subtractor

igazságtáblázat

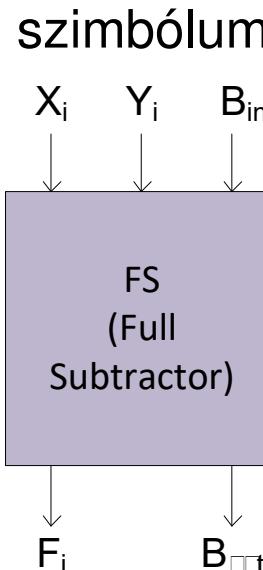
$X_i$	$Y_i$	Bin	Bout	$F_i$
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Karnaugh  
táblái:

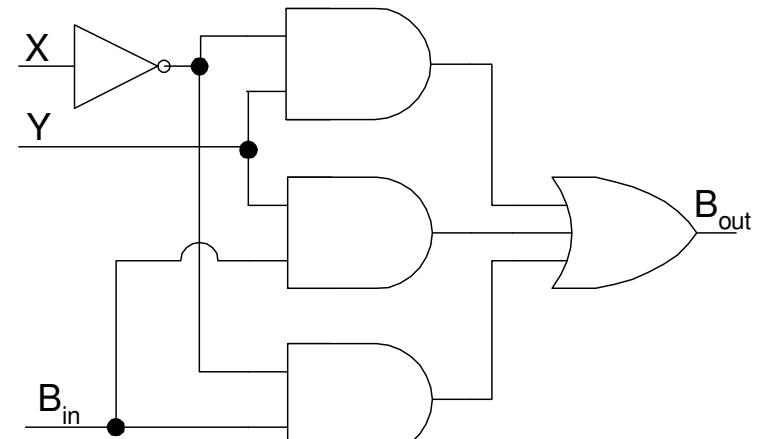


Kimeneti  
fgv-ai:

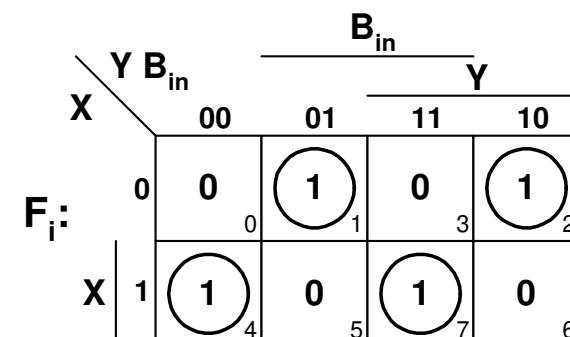
$$B_{out} = \overline{X_i} \cdot \overline{Y_i} + \overline{X_i} \cdot B_{in} + Y_i \cdot B_{in}$$



Logikai kapcsolási rajz Bout-ra  
(F előállítása ugyanaz, mint FA-nál):



$$T_{FS} = \min 2G$$



$$F_i = X_i \oplus Y_i \oplus B_{in}$$

# Bináris kivonás módszerei

## ■ I. módszer:

Bináris kivonás FS segítségével

- $\frac{X_i - Y_i}{0 - 0 \rightarrow 0} \rightarrow F_i$

- $0 - 1 \rightarrow 1$ , borrow bit '1'

- $1 - 0 \rightarrow 1$

- $1 - 1 \rightarrow 0$

*	* * * * *	*
<del>100000000</del>	-	<del>256</del>
- 01001100	-	- 76
	—————	—————
10110100		180

\*: azt jelöli, amikor az adott helyiértéken '1'-et kell kivonni még az  $X_i$  értékéből (borrow from  $X_i$ )

## ■ II. módszer: Kivonás visszavezetése az *univerzálisan teljes bináris összeadás* segítségével (2's komplement alak – korábban tanultuk):

- FA, RCA, vagy LACA

$$F_i = X_i + 2^s \ comp(Y_i)$$



# Számítógép Architektúrák II.

(MIVIB344ZV)

4. előadás: Aritmetikai műveletvégző egységek – szorzás, osztás. Kerekítési eljárások.

Előadó: Dr. Vörösházi Zsolt  
[voroshazi.zsolt@mik.uni-pannon.hu](mailto:voroshazi.zsolt@mik.uni-pannon.hu)

# Jegyzetek, segédanyagok:

- Könyvfejezetek:

- <http://www.virt.uni-pannon.hu> → Oktatás → Tantárgyak → Számítógép Architektúrák II.  
□ (chapter03.pdf)

- Fóliák, óravázlatok .ppt (.pdf)

- Feltöltésük folyamatosan



# Szorzó áramkörök

# Ismétlés: Tárolók - Regiszterek

- A szorzó áramkörökben (is) egy fontos építőelem a **regiszter**. Olyan széles, hogy benne a buszokról, memóriákból, ALU-ból érkező információ eltárolható legyen.
- Adott vezérlőjelek hatására:
  - a bemenetén lévő adatokat betölti, és ideiglenesen eltárolja,
  - a kimenetére teszi a tárolt adatokat, vagy
  - lépteti (shift-eli)* a benne lévő adatokat.
- Működési mód szerint:
  - a) *Hagyományos reg.*: párhuzamos betöltésű/kiolvasású
  - b) *Léptető (Shift) regiszter*: soros betöltésű (kiolvasású)

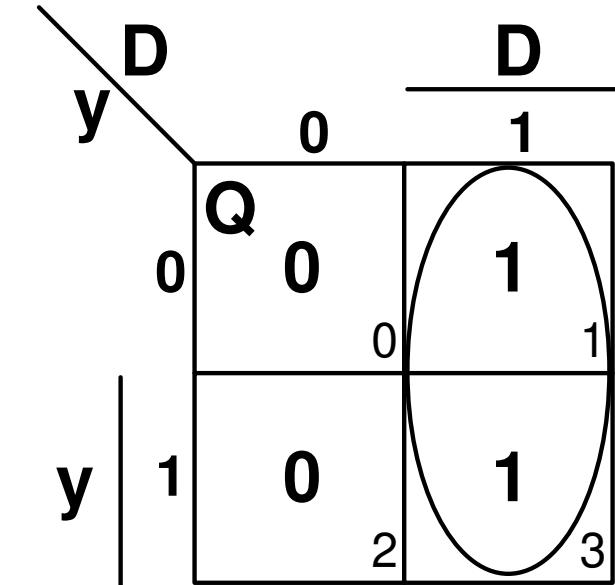
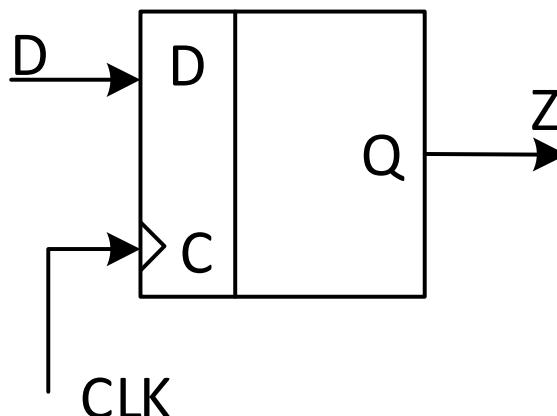
# Ismétlés: D flip-flop

## ■ D tároló

- Csak szinkron módon értelmezhető

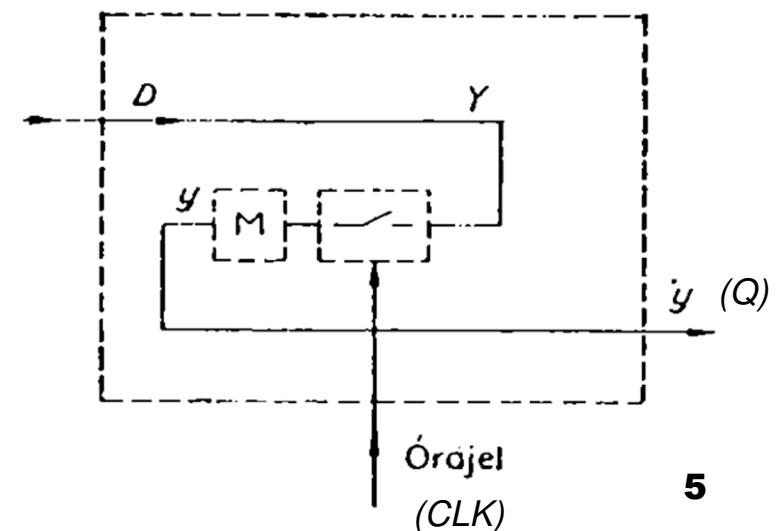
## ■ Működése:

- $D = '0'$  == D-FF állapota változik ('0'-t tárol)
- $D = '1'$  == D-FF állapota változik ('1'-et tárol)
- Tehát egy órajel ciklus (CLK) ideig tároljuk a bemenetre érkezett értéket, változás a CLK élre történhet

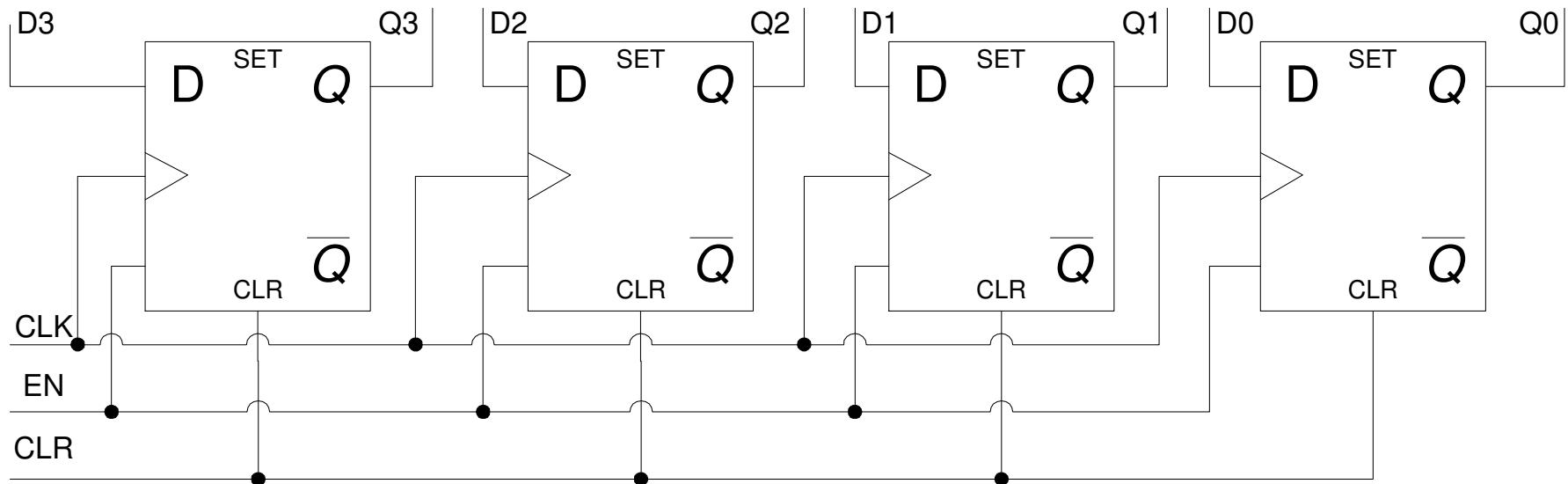


Egyszerűsített DNF alak  
és elvi logikai rajz:

$$Q = f_y(D, y) = D$$

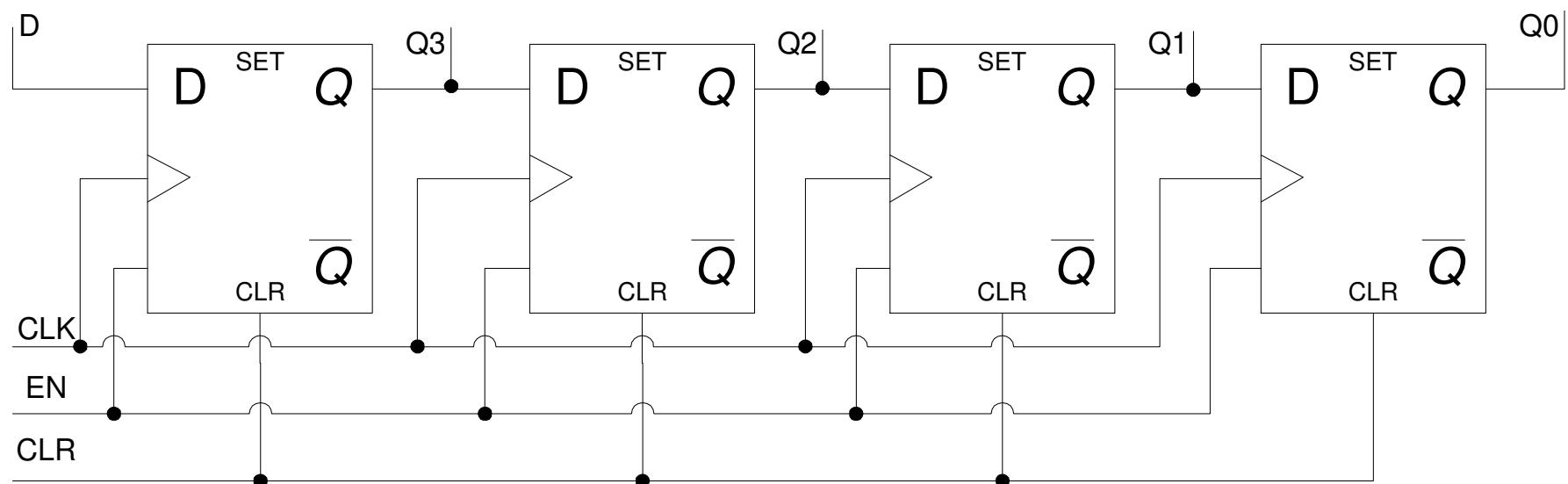


# a.) 4-bites Parallel In/ Parallel Out regiszter (D-tárolókból felépítve)



Katalógus adat: [SN54/74LS175](#)

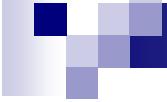
# b.) 4-bites Shift/léptető regiszter (Serial in/Paralel Out – D-tárolós)



Katalógus adat : SN54/74LS95

# Szorzó áramkörök

- I. Iteratív szorzási módszerek
- II. Közvetlen „szorzási” módszerek



# I.) Iteratív szorzási módszerek

# Iteratív szorzási módszerek alapjai

- Tényezők:

$$P = A \times B$$

- P: szorzat, A:szorzandó, B:szorzó

- Pl: Legyenek:

- 'A' és 'B' 5-bites számok ( $0 \dots 2^5 - 1 = 0 \dots 31$ )
    - Maximálisan  $P = 31 \times 31 = 961$  lehet ( $\rightarrow 10$  biten ábr.)

- Tehát: N-bites számok szorzatát  $2 \times N$  biten tudjuk eltárolni!

$$P = A \times B = A \times B_4 B_3 B_2 B_1 B_0 =$$

$$= A \times B_4 \times 2^4 + A \times B_3 \times 2^3 + A \times B_2 \times 2^2 + A \times B_1 \times 2^1 + A \times B_0 \times 2^0$$

# Iteratív szorzási műveletek:

- Hagyományos (Shift&Add) módszer  
(MSB  $\leftarrow$  LSB)

	x	A4	A3	A2	A1	A0
		B4	B3	B2	B1	B0
PP0			A4*B0	A3*B0	A2*B0	A1*B0
PP1			A4*B1	A3*B1	A2*B1	A0*B1
PP2		A4*B2	A3*B2	A2*B2	A1*B2	A0*B2
PP3		A4*B3	A3*B3	A2*B3	A1*B3	A0*B3
PP4	A4*B4	A3*B4	A2*B4	A1*B4	A0*B4	
PR	az egyes oszlopok összege					

- Fordított sorrendű (MSB  $\rightarrow$  LSB):

	A4	A3	A2	A1	A0
x	B4	B3	B2	B1	B0
PP0	A4*B4	A3*B4	A2*B4	A1*B4	A0*B4
PP1		A4*B3	A3*B3	A2*B3	A1*B3
PP2			A4*B2	A3*B2	A2*B2
PP3				A4*B1	A3*B1
PP4					A4*B0
PR	az egyes oszlopok összege				

# Példa:

- Számolja ki szorzás műveletével az  $A^*B = P$  eredményét (ha N=4 bit, LE, uint), A:=1001, B:=1010 esetén:
  - Shift & Add, és
  - Fordított sorrendű módszerekkel

Shift & Add

A:	Szorzandó	1001
B:	Szorzó	1010
		<hr/>
PP0		0000
PP1		1001
PP2		0000
PP3		1001
		<hr/>
P:		01011010

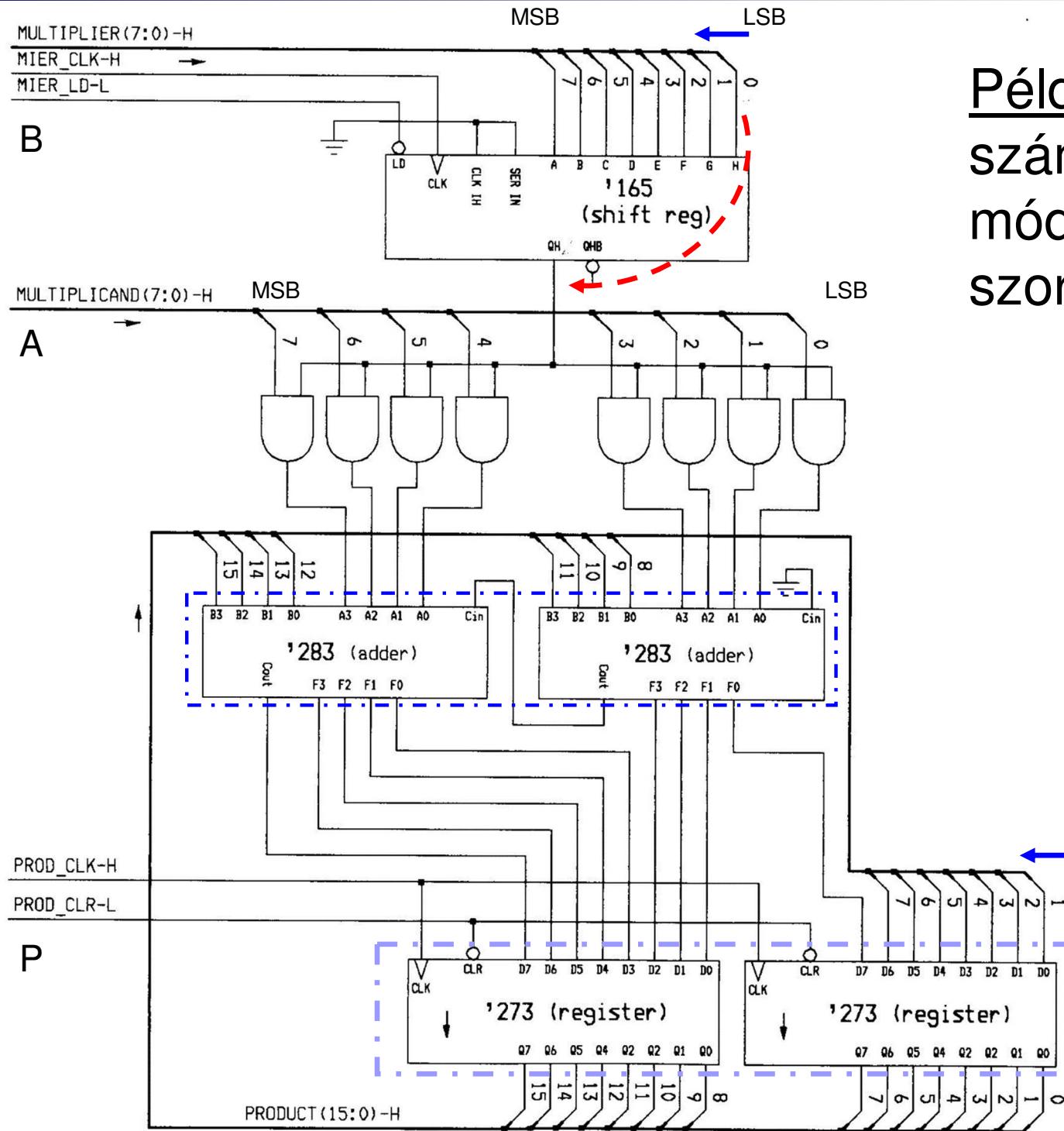
Fordított sorrendű

A:	Szorzandó	1001
B:	Szorzó	1010
		<hr/>
PP0		1001
PP1		0000
PP2		1001
PP3		0000
		<hr/>
P:		01011010

# 1.) Általános Shift&Add módszer

- $P = A \times B$  ( $A$ :szorzandó,  $B$ :szorzó)
- Parciális szorzatok (PPi) összegét az **MSB**  $\leftarrow$  **LSB** bitek szerint képezi (mivel a  $B$  szorzat biteket is ebben a sorrendben tölti be a Shift regiszterből  $N$  órajel CLK alatt)
- AND kapuk: PPi-k képzése
- Shift-elés: *Huzalozott eltolással* (a visszacsatolt ágban  $1. \leftarrow 0.$ )

Példa: két N=8 bites szám Shift&Add módszerű szorzására



# Shift&Add szorzó építőelemei (folyt):

- 2-input AND gates (prepare partial products)
- ‘165 – 1db 8-bit paralle input – serial Shift Register
- ‘283 – 2db 4-bit Adder: itt helyettesíteni lehetne akár 1db 8-bites Adder-el
- ‘273 – 2db 8-bit Parallel Register (D-FFs): helyettesíteni lehetne, 1db 16-bites Register-el

# Folyamatábra (Shift&Add)

- Alapvetően adatfüggetlen, DE:
- Adatfüggővé tehető - gyorsítható algoritmus!

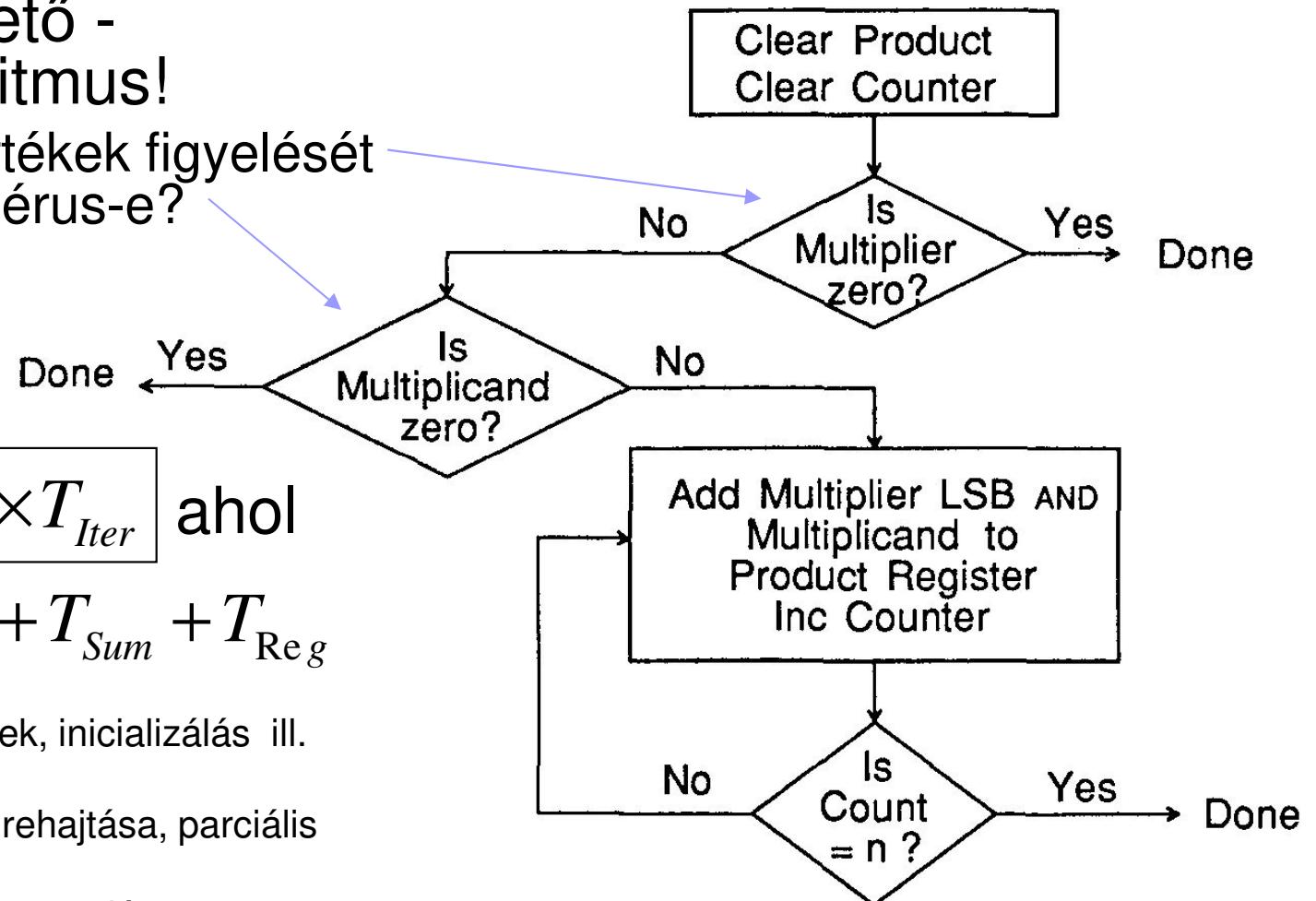
□ Bemenő (A,B) értékek figyelését kell megoldani: zérus-e?

- Időszükséglet:

$$T_{Mult} = T_{Setup} + N \times T_{Iter} \quad \text{ahol}$$

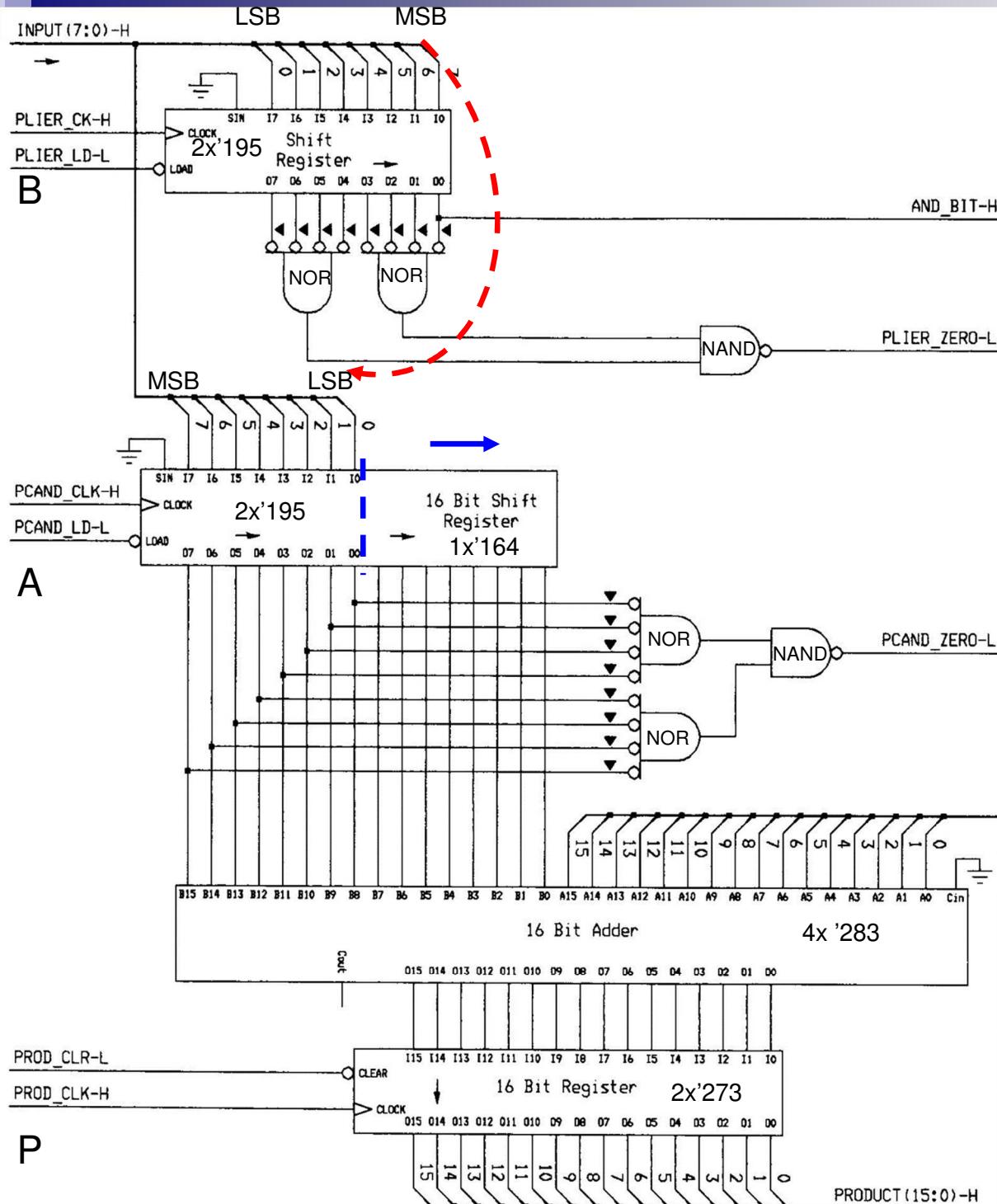
$$T_{Iter} = T_{AND} + T_{Sum} + T_{Reg}$$

- $T(\text{SETUP})$ : kezdeti ellenőrzések, inicializálás ill. szorzat regiszter törlése
- $T(\text{AND})$ : AND függvények végrehajtása, parciális szorzatok képzése
- $T(\text{SUM})$ : parciális szorzatok összeadása
- $T(\text{REG})$ : betöltésük a regiszterbe



## 2.) Fordított sorrendű módszer

- $P=A \times B$  ( $A$ :szorzandó,  $B$ :szorzó)
- Parciális szorzat ( $PP_i$ ) összegeket itt fordított sorrendben, az **MSB** → **LSB** bitek felé haladva képzi
  - Tehát a B szorzat biteket *fordított* sorrendben tölti be, illetve elsőként az MSB pozíión lévő PP<sub>i</sub> értéke kerül a szorzat (C) regiszterbe
- Adatfüggőség: bemenetek figyelésekor ha a szorzandó, vagy szorzó bitek értéke zérus, nem kell elvégezni a szorzást (vizsgálata N-bit szélességű AND kapukkal)



## Példa: két N=8-bites szám Fordított sorrendű szorzására

Az Adder olyan széles, mint a P szorzat regiszter (16)

Adatfüggőség: zérus-e?

**PLIER\_ZERO\_L /**  
**PCAND\_ZERO\_L**

Az **AND\_BIT\_H** jel feltételesen határozza meg, hogy az „A” szorzandó regiszter (benne a „B” szorzó-regiszter értékeivel) a „P” szorzatregiszterbe másolható-e

Ha az  $\text{MSB} = '1'$  ( $B < 7 >$ ), akkor szorzandó és szorzó reg. tartalma összeadható;

Ha  $\text{MSB} = '0'$  a B szorzó- és az A szorzandóregiszter tartalma 1-bitpozícióval shift-elődik jobbra:

- Szorzó: alacsonyabb bitpozíciók felé.
- Szorzandó: magasabb bitpozíciók felé egyszerre

# Fordított sorrendű szorzó építőelemei (folyt):

- 2-input AND gates (partial product)
- '195 – 4-bites Shift Regiszter
  - 2 db szorzó reg. (B) tárolására, shiftelésére
  - 2 db szorzandó reg. (A) tárolására és shiftelésére
- '164 – 1 db 8-bites Shift regiszter
- '283 – 4db 4-bit Adder: itt a 4 db Adder olyan széles kell legyen, mint a P szorzat regiszter
  - (helyettesíthető lenne egyetlen 16-bites Adder-el)
- '273 – 2db 8-bit Parallel Regiszter (2x8 D-FFs)
  - Helyettesíthető lenne egyetlen 16-bites Parallel Regiszterrel
- **Előnye:** nem kellenek AND-ek a PPi-k képzéséhez;
- **Hátránya:** viszont szélesebb összeadó, és A szorzandó reg. kell (szemben a Shift&Add módszerrel).

# Folyamatábra (fordított sorrendű)

- Adatfüggő algoritmus - gyorsított végrehajtás

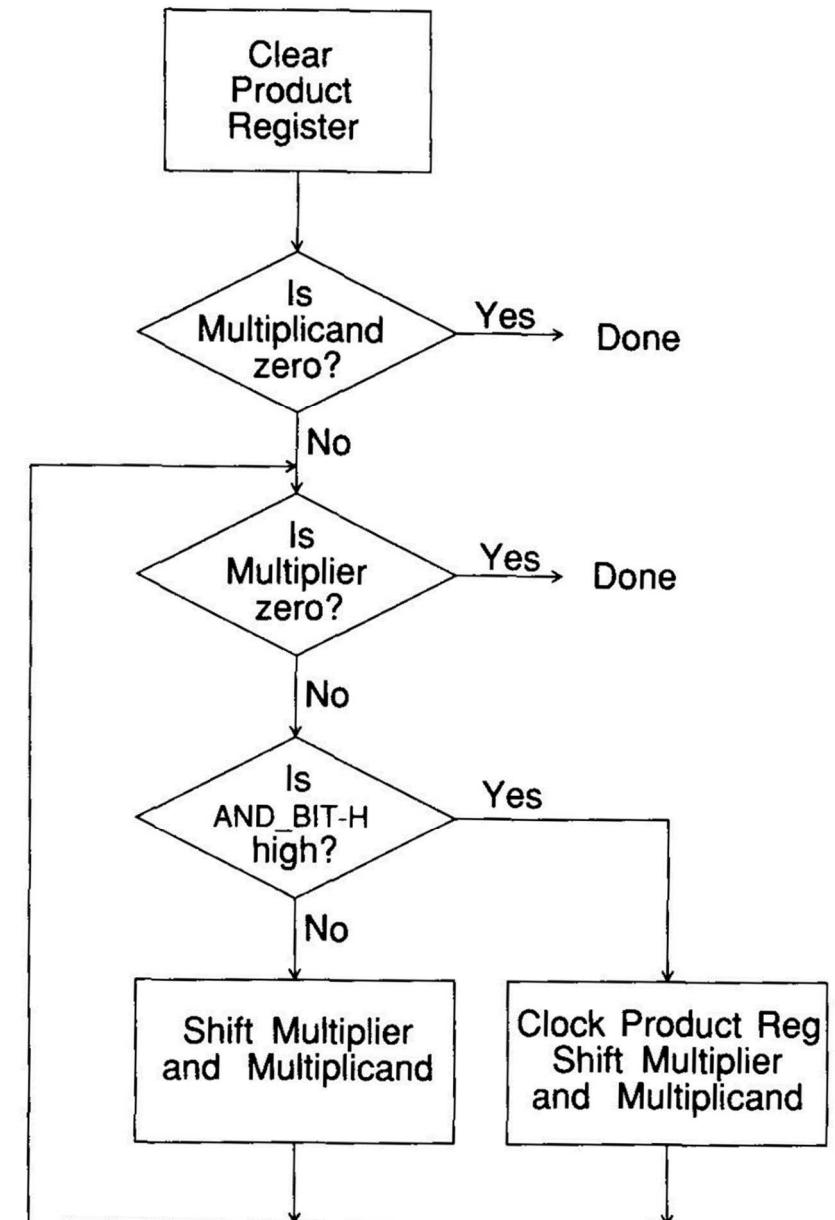
- Bemenő (A,B) értékeket figyeli, hogy zérus-e?

- Időszükséglet:

$$T_{Mult} = T_{Setup} + N \times T_{Iter} \quad \text{ahol}$$

$$T_{Iter} = T_{Shift\_Reg} + T_{Sum} + T_{Reg}$$

- $T(\text{SETUP})$ : kezdeti ellenőrzések, inicializálás ill. szorzat regiszter törlése
- $T(\text{SUM})$ : parc $T(\text{AND})$ : AND függvények végrehajtása, parciális szorzatok képzése
- jálás szorzatok összeadása
- $T(\text{REG})$ : betöltésük a regiszterbe



## c.) Előjeles szorzás Booth-algoritmussal:

### ■ Negatív számokkal is lehet szorzást végezni!

- Legyen a következő B 2's komplement 6-bites szám
- $B = B_5 B_4 B_3 B_2 B_1 B_0 = B_5 \times (-2^5) + B_4 \times 2^4 + B_3 \times 2^3 + B_2 \times 2^2 + B_1 \times 2^1 + B_0 \times 1$ .
- Újrakódolási technika (séma):**

$$B(\text{2' komplement}) = B_5 \times (-32) + B_4 \times 16 + B_3 \times 8 + B_2 \times 4 + B_1 \times 2 + B_0 \times 1 = \\ \text{TRÜKK!!}$$

$$= B_5 \times (-32) + B_4 \times (32 - 16) + B_3 \times (16 - 8) + B_2 \times (8 - 4) + B_1 \times (4 - 2) + B_0 \times (2 - 1) = \\ (\text{azonos numerikus értékek összerendelése}) \\ = B_5 \times (-32) + B_4 \times 32 - B_4 \times 16 + B_3 \times 16 - B_3 \times 8 + B_2 \times 8 - B_2 \times 4 + B_1 \times 4 - B_1 \times 2 + \\ B_0 \times 2 - B_0 \times 1 = \\ = -32 \times (B_5 - B_4) - 16 \times (B_4 - B_3) - 8 \times (B_3 - B_2) - 4 \times (B_2 - B_1) - 2 \times (B_1 - B_0) - \\ 1 \times (B_0 - 0).$$

# Példa: előjeles Booth algoritmus

- Az előző oldalon lévő átzárójelezéssel a megfelelő értékeket két bitpár kivonásával kapjuk (a **zárójeles** kifejezés értéke ha **+**: akkor kivonás,/ ha **0**: akkor áteresztés / ha **-**: összeadás történik). A súlytényezők 2 hatványai, és a szorzást a súlytényezők shiftelésével oldják meg ('165 Shift-regiszterrel). //Egy összeadást minden egy kivonás követ alternáló jelleggel.
- Példa:      543210 (bitpozíciók)  
 $011001 = 25 \quad \text{„A”}$   
 $101101 = -19 \quad \text{„B”}$

Végrehajtjuk  $\mathbf{P}=\mathbf{A} \times \mathbf{B}$ -t!

Az újrakódolást bitpárokon végezzük el:

$-1 \times (B_0 - 0) = \underline{-1}$	$P_0 = 0 - 1 * A$	Mivel – volt az érték, ezért kivonjuk a 0-ból az A-t.
$-2 \times (B_1 - B_0) = \underline{+2}$	$P_1 = P_0 + 2 * A$	Mivel + volt az érték, ezért hozzáadjuk P0-hoz a $2 * A$ -t.
$-4 \times (B_2 - B_1) = \underline{-4}$	$P_2 = P_1 - 4 * A$	kivonjuk
$-8 \times (B_3 - B_2) = \underline{0}$	$P_3 = P_2$	Mivel ‘0’ volt, Áteresztés, nem változik.
$-16 \times (B_4 - B_3) = \underline{+16}$	$P_4 = P_3 + 16 * A$	hozzáadjuk
$-32 \times (B_5 - B_4) = \underline{-32}$	$P_5 = P_4 - 32 * A$	kivonjuk

végeredmény értéke

$P_{n+1} = P_n - 2^n \times (B_n - B_{n-1}) \times A$

# Példa: Booth algoritmus (folyt.)

Végrehajtjuk  $\mathbf{P} = \mathbf{A} \times \mathbf{B}$ -t!

543210 (bitpozíciók) N= 6, P = 2×N  
011001= 25 „A” Ellenőrzés:  
101101= -19 „B” P = -475  
// 1110\_0010\_0101

$$P_0 = 0 - 1 * A = 0 - 1 * 25 = -25$$

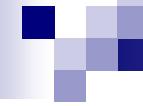
$$P_1 = P_0 + 2 * A = -25 + 2 * 25 = 25$$

$$P_2 = P_1 - 4 * A = 25 - 4 * 25 = -75$$

$$P_3 = P_2 = -75$$

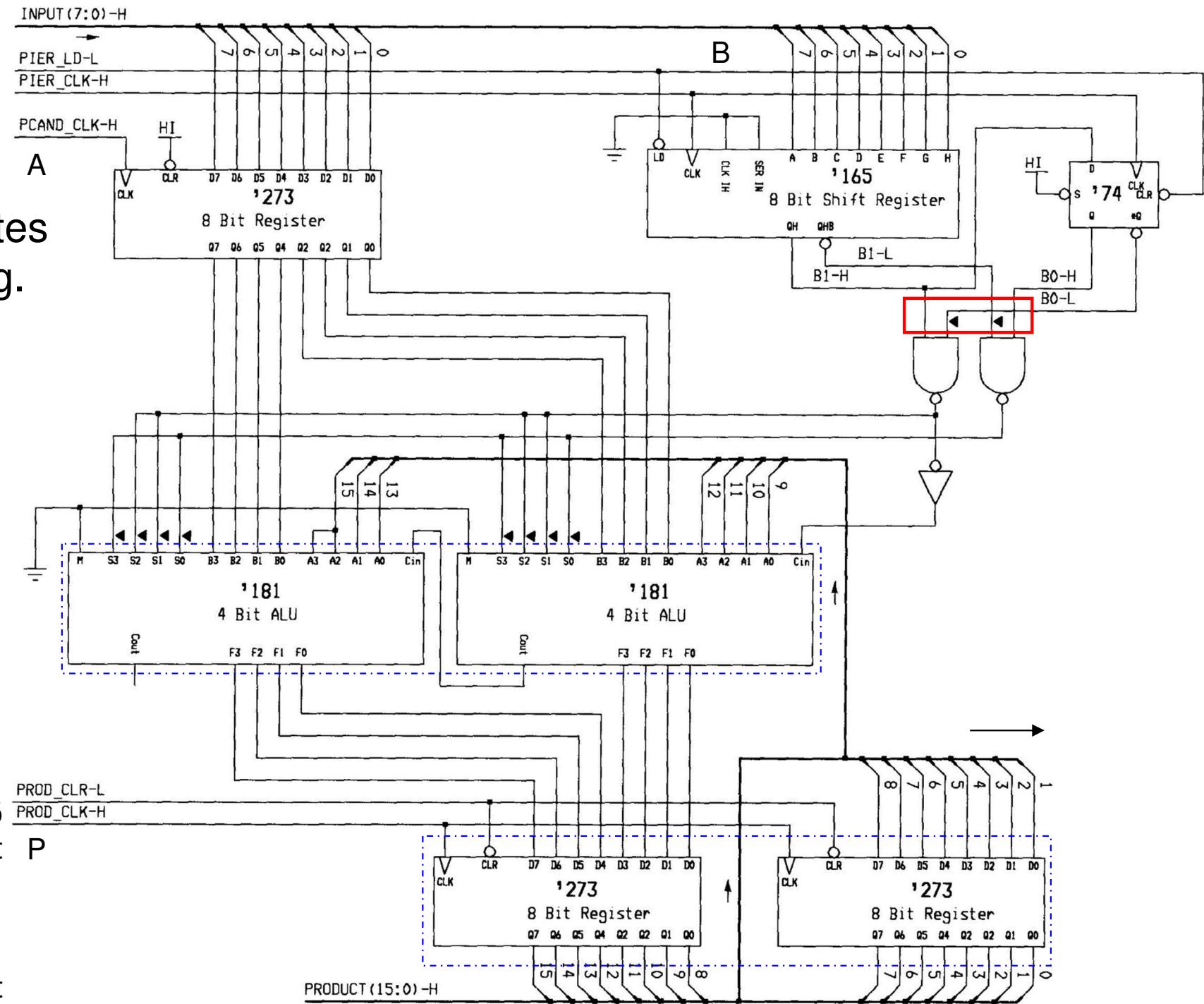
$$P_4 = P_3 + 16 * A = -75 + 16 * 25 = 325$$

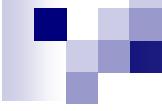
$$P_5 = P_4 - 32 * A = 325 - 32 * 25 = \textcolor{red}{-475}$$



Példa: két 8-bites szám Booth alg. szorzására

A '74 D tároló (késleltetés!) B0\_H ill. a „B” szorzóregiszter kimenetéről B1\_H jelek a szorzó shift-elődésének megfelelően generálódnak (egymást követő pozíciókat vizsgálunk). Ezek állítják elő megfelelő kombinációs hálózat (2 NAND kapu, és 1 Inverter) segítségével az S0-S3 kiválasztó jeleket az ALU-nál.





## II. Közvetlen „szorzási” módszerek

# Közvetlen „szorzási” módszerek

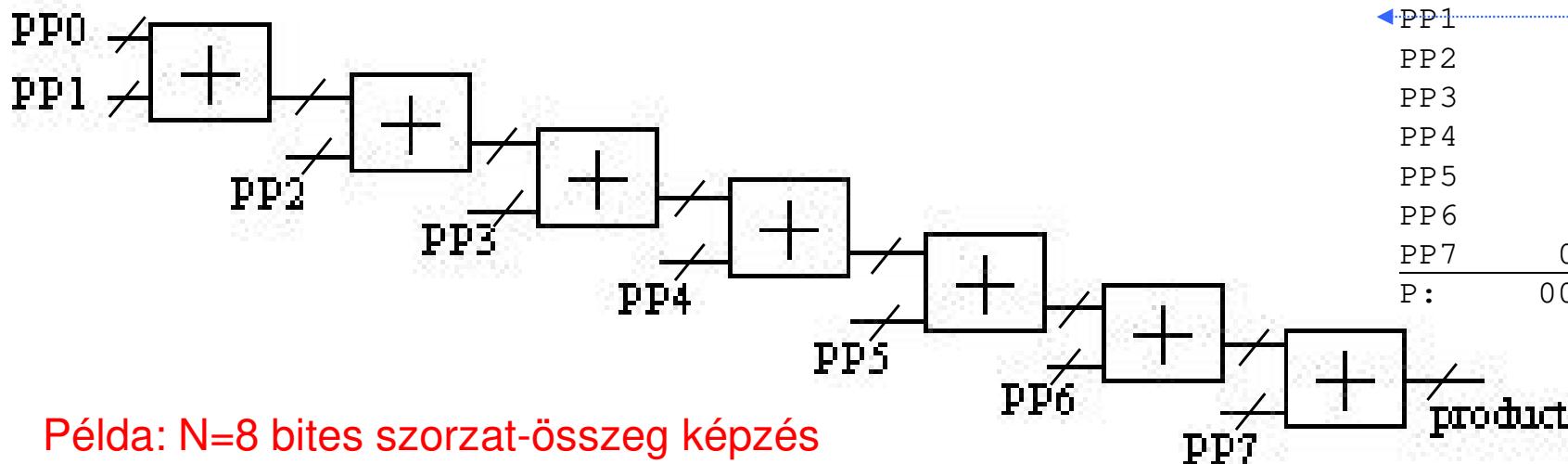
- Először a **rész-szorzatokat (Partial Products: PPi)** állítják elő, majd pedig azokat összeadják:
  - soronként vagy,
  - oszloponként.
- A rész-szorzatok eltolása egységnyi kapu késleltetéssel megvalósítható.
- Fajtái (rész-szorzat képzés):
  - lineáris modell**,
  - (bináris) fa modell**,
  - FA - Full Adder felhasználásával**,
  - CSA: Carry Save Adder (Sorcsökkentős megvalósítás)

# a.) Lineáris modell

- A PPi-k a parciális szorzatképzés után azonnal összeadhatók *soronként*, így gyorsabban megkapjuk az eredményt. N bites számok esetén ( $N-1$ ) db összeadóra van szükségünk. Lassabb, mint a következő fa modell, mivel több összeadó szintű a késleltetés.
- Időszükséglet:  
 $(T_{Sum} = N \times T_{FA})$

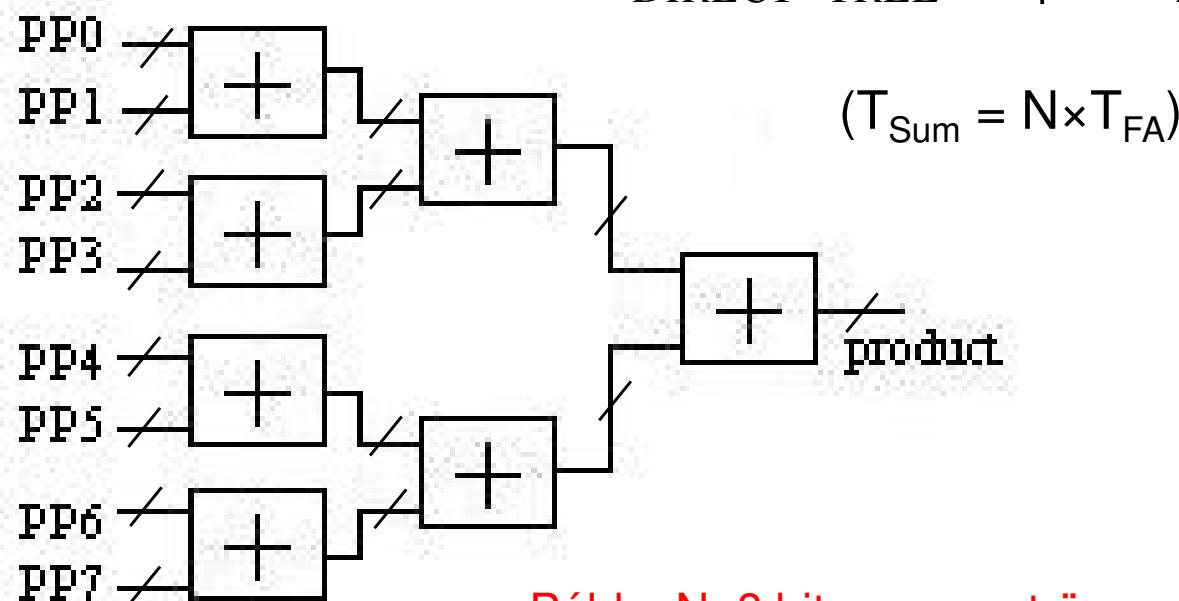
$$T_{(DIRECT-LINE)} = (N-1) * T_{(SUM)}$$

A:	Szorzandó	01101001
B:	Szorzó	01011010
PP0		00000000
PP1		01101001
PP2		00000000
PP3		01101001
PP4		01101001
PP5		00000000
PP6		01101001
PP7		00000000
P:		0010010011101010



## b.) Fa modell („összeadó fa”)

- A PPi-k parciális szorzatok azonnal összeadhatók soronként. Gyorsabb a lineáris modellnél, mivel ebben az esetben ( $N=8$  bit esetén) csak 3-szintű a hierarchia, így kevesebb a késleltetés.  $N$  bites számok esetén ( $N-1$ ) db összeadóra van szükségünk.
- Időszükséglet:  $T_{DIRECT-TREE} = \lceil \log_2(N) \rceil * T_{SUM}$



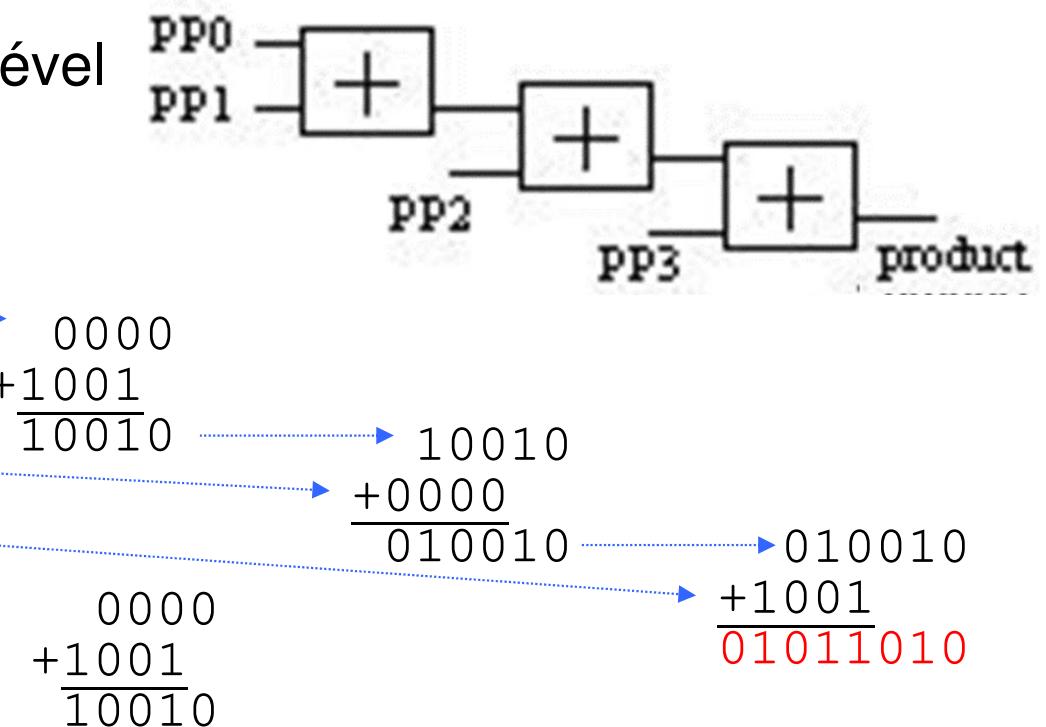
Példa:  $N=8$  bites szorzat-összeg képzés

A:	Szorzandó	01101001
B:	Szorzó	01011010
PP0		00000000
PP1		01101001
PP2		00000000
PP3		01101001
PP4		01101001
PP5		00000000
PP6		01101001
PP7		00000000
P:		0010010011101010

# Példa:

- Számolja ki Shift&Add szorzás műveletével az  $A^*B = P$  eredményét (ha  $N=4$  bit, LE, uint),  $A:=1001$ ,  $B:=1010$  és adja össze a parciális szorzatokat a:
  - a.) Lineáris modell,
  - b.) Bináris fa modell segítségével

$$\begin{array}{r} A: \text{Szorzandó} & 1001 \\ B: \text{Szorzó} & 1010 \\ \hline \end{array}$$



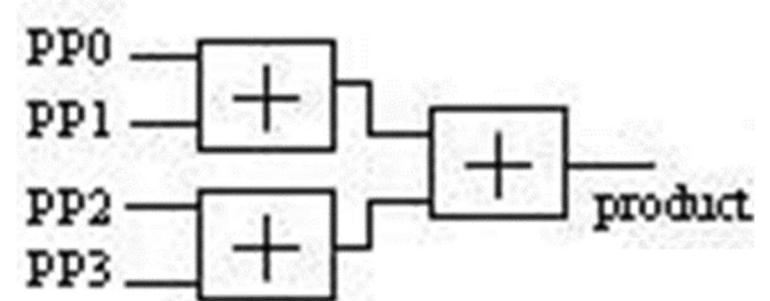
$$\begin{array}{r} 0000 \\ +1001 \\ \hline 10010 \end{array}$$
$$\begin{array}{r} 0000 \\ +0000 \\ \hline 0000 \end{array}$$
$$\begin{array}{r} 10010 \\ +0000 \\ \hline 10010 \end{array}$$
$$\begin{array}{r} 010010 \\ +1001 \\ \hline 01011010 \end{array}$$

# Példa (folyt):

- Számolja ki Shift&Add szorzás műveletével az  $A^*B = P$  eredményét (ha  $N=4$  bit, LE, uint),  $A:=1001$ ,  $B:=1010$  és adja össze a parciális szorzatokat a:
  - a.) Lineáris modell,
  - b.) Bináris fa modell segítségével**

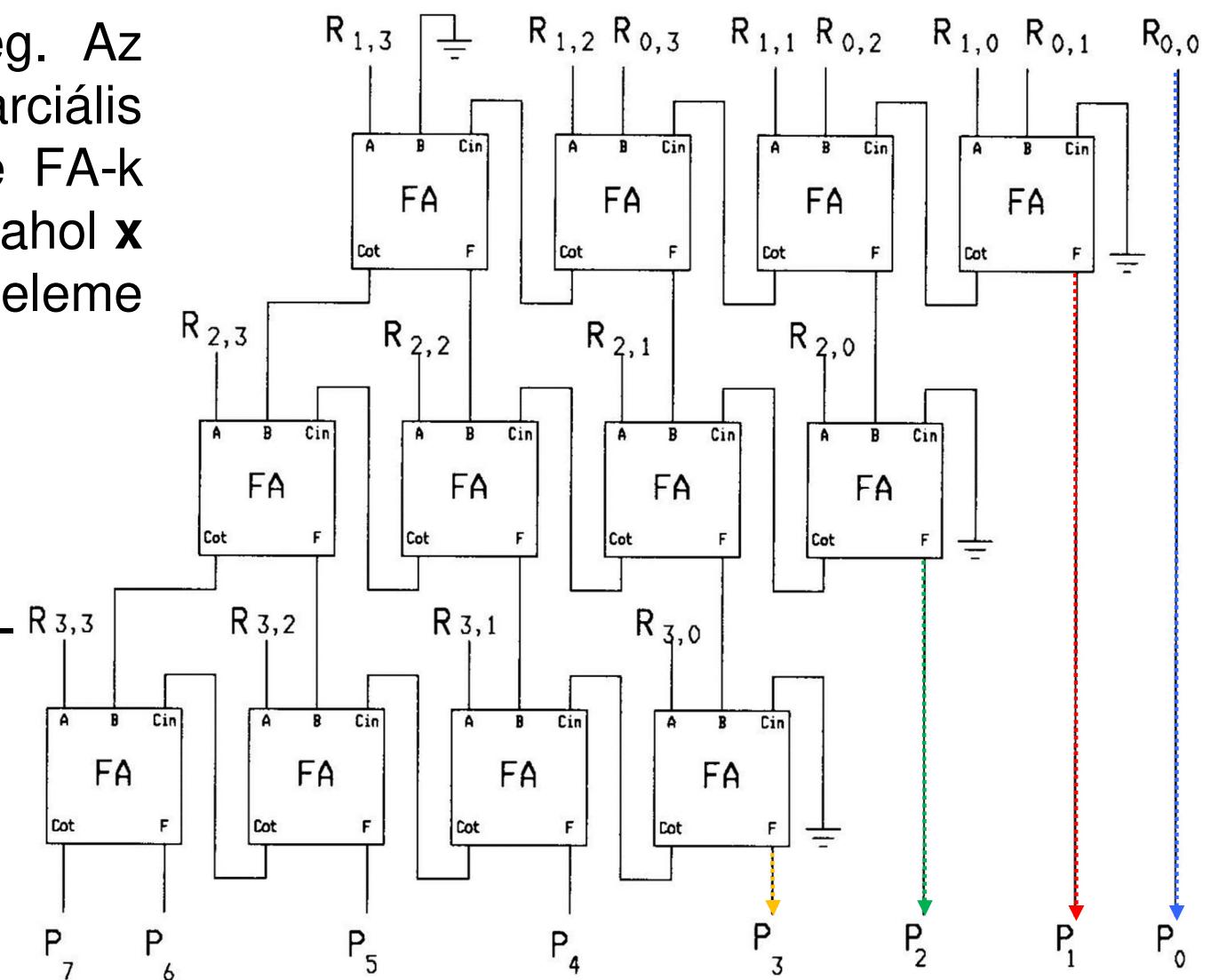
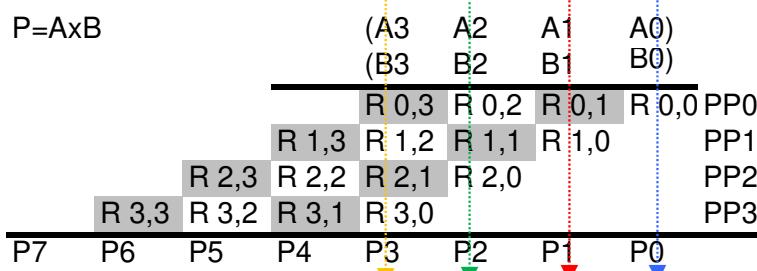
$$\begin{array}{r} A: \text{Szorzandó} & 1001 \\ B: \text{Szorzó} & 1010 \\ \hline \end{array}$$

PP0            0000            0000  
PP1            1001            +1001  
PP2            0000            10010  
PP3            1001            +1001  
P :            01011010            10010  
                                  + 10010  
                                  01011010



# c.) Full Adder-es megvalósítás

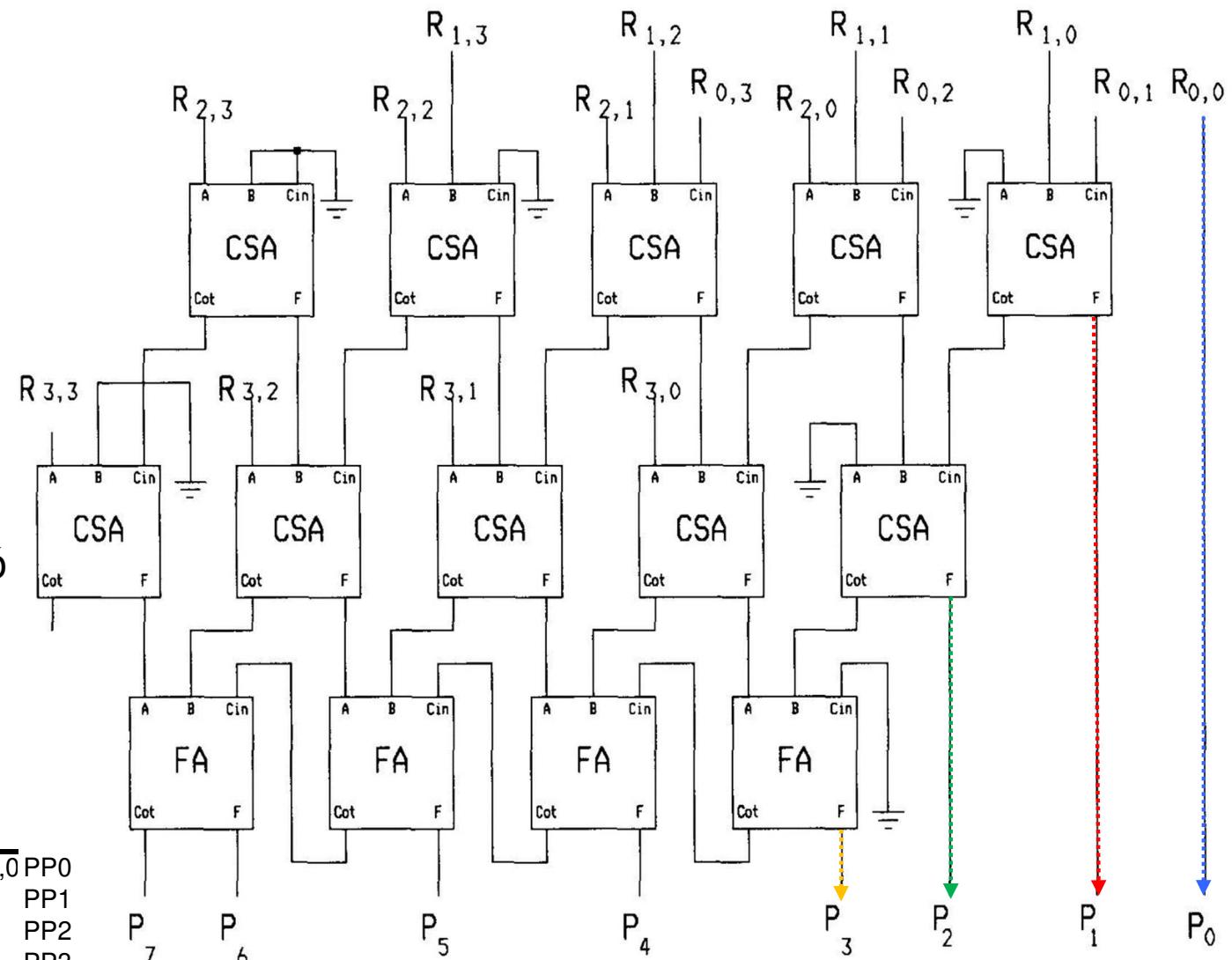
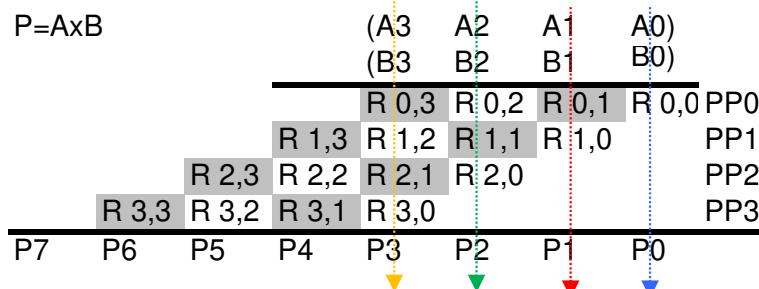
- Az ábrán két 4-bites szám szorzását valósítjuk meg. Az oszlopokat, mint parciális szorzatokat adjuk össze FA-k segítségével. Jel:  $R_{x,y}$ , ahol  $x$  a sor száma,  $y$  a sor eleme (oszlop).



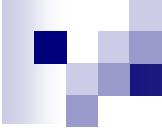
Példa: N=4 bites szorzat-összeg képzés

# d.) CSA: Carry Save Adder

- CSA: olyan Full Adder, amely az előző szint átvitelét (Cout) eltárolja, **és a következő szint Cin-jének továbbítja**. Ezzel a módszerrel a szorzás sebessége tovább növelhető. A késleltetés *mindig 2G / CSA*.
- Az utolsó sorban **FA**-kat használunk, míg az első két sorban CSA-k találhatók. A CSA csökkenti az összeadandó sorok számát ( $3 \rightarrow 2$  sorcsökkentő).



Példa: N=4 bites szorzat-összeg képzés



# Osztó áramkörök

# Osztó áramkörök:

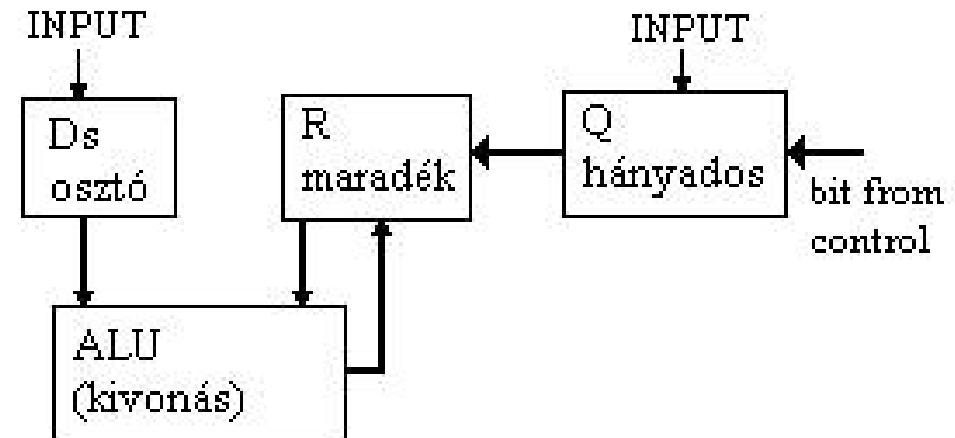
- I.) Hagyományos – „lassú”, vagy közvetlen iteratív osztási algoritmus
  - Euklideszi osztás
- II.) „Gyors” - iteratív osztási algoritmusok (egyéb numerikus módszerek)

# I.) Hagyományos közvetlen osztási algoritmus:

Ez az osztási folyamat igen lassú eljárás. Lépései:

1. az osztót a  $D_s$  regiszterbe rakjuk, az osztandót ( $D_d$ ) pedig a Q regiszterbe.
2. töröljük az R regisztert
3. iterációs lépés: kivonjuk az R-ből a  $D_s$  osztót. Ha  $R - D_s > 0$  akkor folytatódik, tehát ezt a megváltozott értéket visszatesszük az R-be, és egy '1'-est teszünk a Q regiszterbe. Ha  $R - D_s < 0$  akkor R regiszter tartalma nem változik, és egy '0'-át teszünk a Q regiszterbe (vagy hogyha nincs több osztandó bit, akkor vége az osztásnak).
4. minden iterációs lépésben egy-egy új bit jön létre, amelyet a Q regiszterbe shiftelünk, ahogyan az R regiszterbe az osztandót
5. Az osztandó legnagyobb helyiértékű (MSB) bitjével kezdjük az összehasonlítást (míg a legkisebbtől a legnagyobb helyiértékek felé, balra haladva shift-elünk a visszaszorznál)
6. A hányados generálódik elsőként az MSB felől, és a Q-ba shift-elődik 1 bittel balra
7. A folyamat végén a maradék Az R-ben, a hányados pedig a Q-ban lesz

$$D_d = Q \times D_s + R$$



# Példa: Hagyományos osztási algoritmus

$$Dd = Q * Ds + R$$

Egy kikötésünk van:  $R < Ds$  esetén leáll az osztás!

Decimális számok esetén:

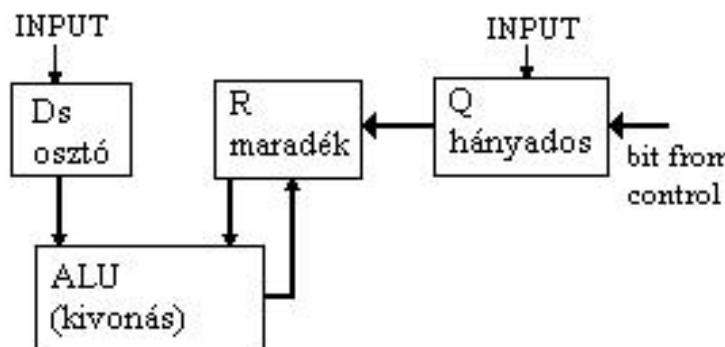
$$\begin{array}{r} 5|8 : 5=1|1 \\ 0\ 8 \\ \quad\quad\quad 3 \end{array}$$

Bináris számok esetén hasonlóan

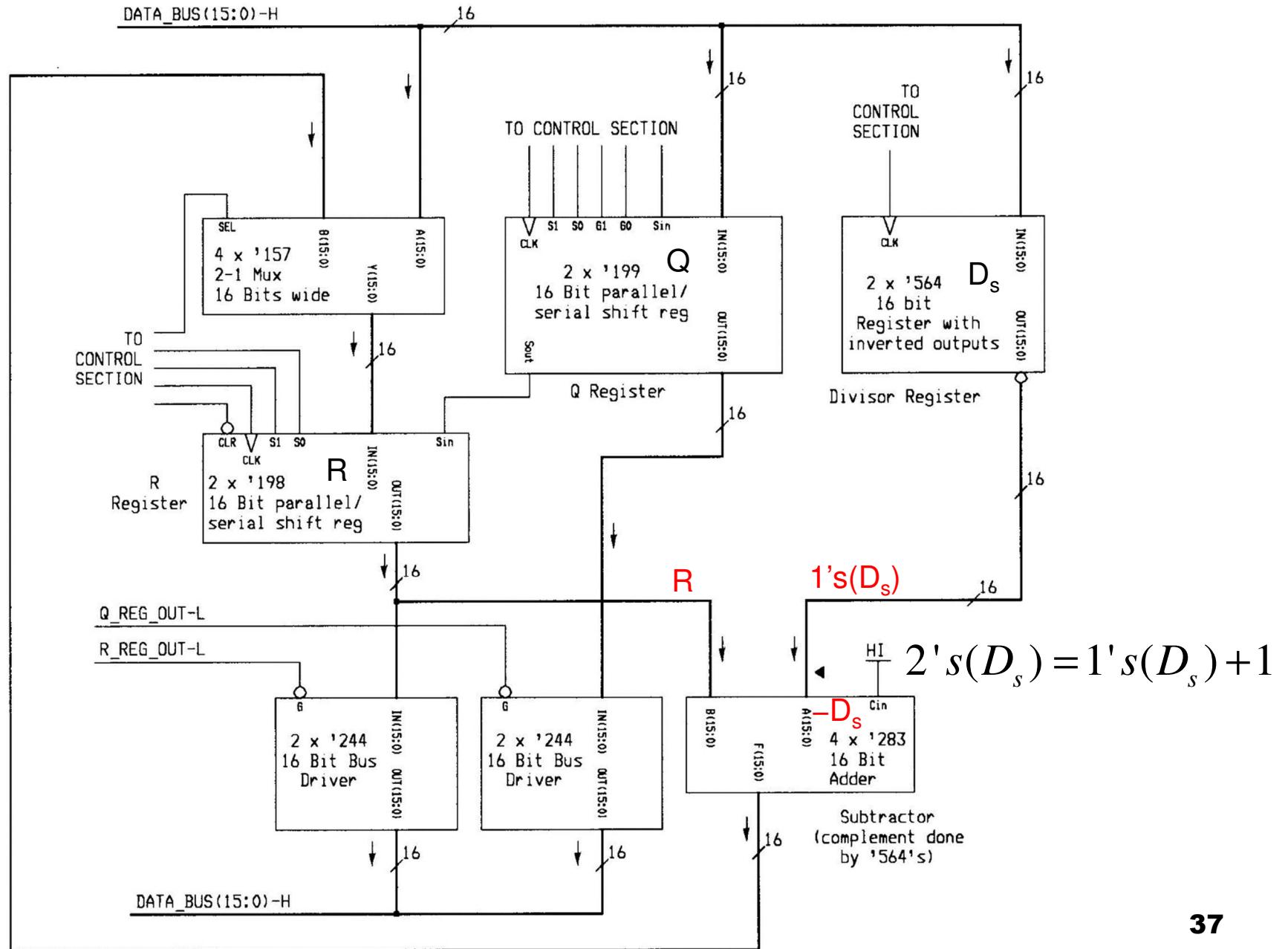
**1011**

Q hánnyados (Ds hányszor van meg Dd-ben)

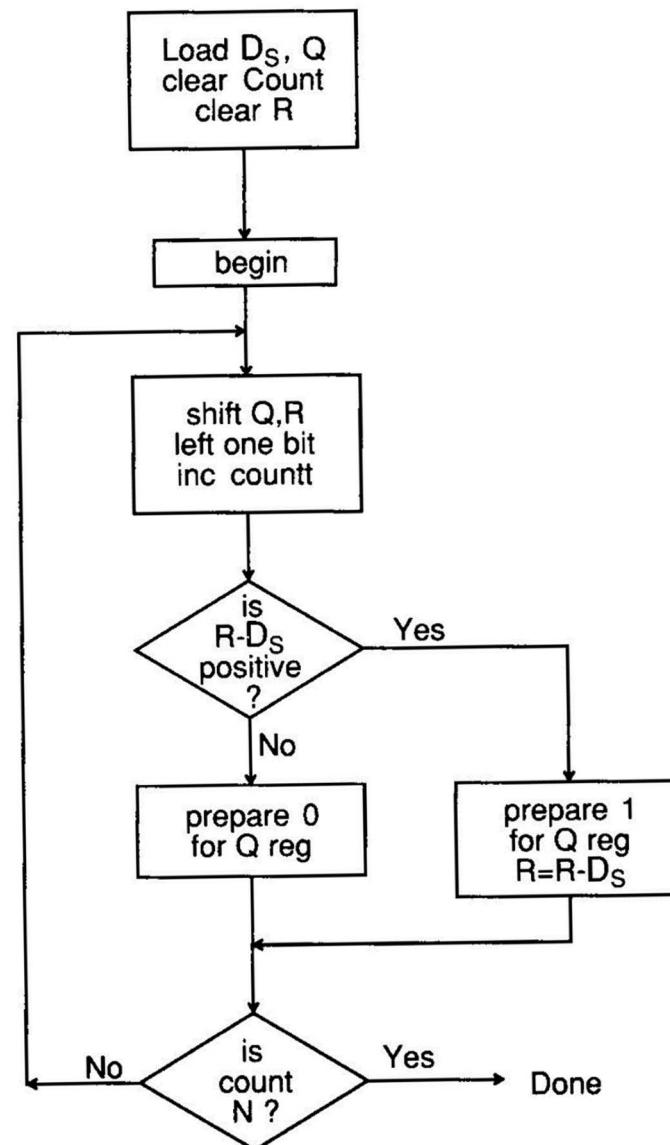
101/	111010	Dd osztandó Ds osztó
	111 010	111-ben megvan „101” ezért <b>1</b> ? Q
	101	Visszaszorzás $1 * '101'$ -el
	10	Ez a kivonás eredménye $111-101=10$
	100 00	100-ban nincs meg az ’101’, ezért <b>0</b> ? Q
	000	Visszaszorzás $0 * '101'$ -el
	100	Ez a kivonás eredménye $100-000=100$
	1001 0	1001-ban megvan ’101’, ezért <b>1</b> ? Q
	101	Visszaszorzás $1 * '101'$ -el
	100	Ez a kivonás eredménye: $1001-101=100$
	1000	1000-ban megvan az ’101’, ezért <b>1</b> ? Q
	101	Visszaszorzás $1 * '101$ -el
	11	Kivonás eredménye: $1000-101=11$
		Ez a maradék R!



# Hagyományos osztó áramkör



# Folyamatábra: osztási algoritmus



## II.) Iteratív osztási algoritmus

- a.) „Gyors” osztás Newton- Raphson módszerrel
- b.) Közvetlen „gyors” osztó

# a.) Gyors osztás Newton- Raphson módszerrel

Az előzőnél gyorsabb osztási művelet reciprokképzéssel valósul meg. Szorzó segítségével végezzük el az osztást. A Newton-Raphson iteráció alapformulája a következő:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Van egy megfelelő f függvényünk és egy x<sub>0</sub> kezdeti értékünk. Iterációs lépésekkel megkapjuk az osztás eredményét az f(x)=0 egyenlet megoldásaként. Az f-et úgy kell (jól) megválasztanunk, hogy a reciprok gyökkel rendelkezzen. Legyen

$$f(x) = \frac{1}{x} - w$$

Az fenti egyenlet gyöke, f(x)=0 esetén az x= 1/w. Ha f(x)=1/x-w, akkor

$$f'(x) = -\frac{1}{x^2}$$

Ekkor visszahelyettesítve az eredeti Newton-Raphson iterációs képletbe a következőt kapjuk:

$$x_{i+1} = x_i - \frac{\frac{1}{x_i} - w}{-\frac{1}{x_i^2}} = x_i + (x_i - wx_i^2) = 2x_i - wx_i^2 = x_i(2 - wx_i)$$

Tehát az A/B műveletet A\*(1/B) alakra írtuk át, és az 1/B reciprokképzést egy szorzóval és egy kivonóval valósíthatjuk meg. A függvény Taylor sorának kiterjesztésével (négyzetes konvergencia) belátható, hogy minden egyes iterációs lépésben a helyes bitek száma megduplázódik. Tehát megfelelő iterációs lépés kiválasztásával a kívánt pontosság elérhető!

# Példa: Newton Raphson Szám négyzetgyökének közelítése

- $\sqrt{612} = ?$  612 négyzetgyökét keressük, azonos a következővel:

$$x^2 = 612$$

- A következő függvényt átalakítással kapjuk, amely Newton Raphson módszerben használható (gyök keresés,  $f(x) = 0$ ):

$$f(x) = x^2 - 612$$

- Deriváltja:

$$f'(x) = 2x$$

- Kezdeti érték  $x_0 = 10$ -nek választásával kapjuk:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = 10 - \frac{10^2 - 612}{2 \cdot 10} = 35.6$$

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} = 35.6 - \frac{35.6^2 - 612}{2 \cdot 35.6} = 26.3955056$$

$$x_3 = \vdots = \vdots = 24.7906355$$

$$x_4 = \vdots = \vdots = 24.7386883$$

$$x_5 = \vdots = \vdots = 24.7386338$$

Várt érték:  
24.73863375370...

Aláhúzások,  
már a korrekt  
számjegyeket  
jelölik, az  
egyes  
iterációkban

## b.) Közvetlen gyors osztó

- Az iteratív osztási művelet másik módszere a következő:  $Q = D_D / D_S$  kiszámolható a következő egyenlettel, ha a **successive** (egymást követő)  $f_k$ -k úgy vannak megválasztva, hogy a nevező az 1-hez konvergáljon.

$$Q = \frac{D_D \times f_0 \times f_1 \times f_2 \dots}{D_S \times f_0 \times f_1 \times f_2 \dots}$$

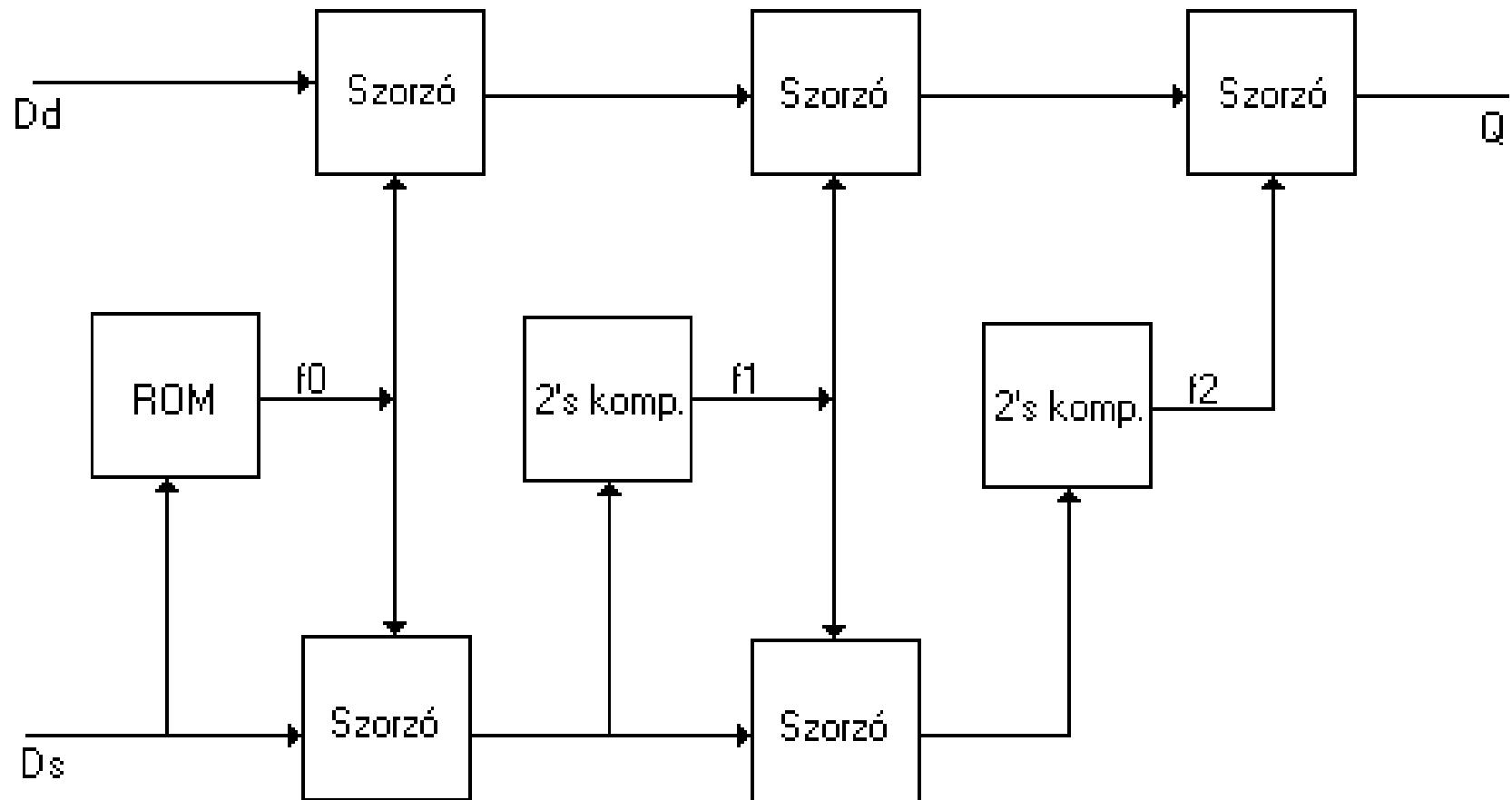
# Közvetlen gyors osztó működése

- A számláló iterációja ennél a módszernél  $D_{Dn+1} = D_{Dn} \times f_n$ . A nevező iterációját a megismert módon használjuk a következő f-ek meghatározására.
- Tegyük fel, hogy szorzóink, 2'komplemens képző egységeink vannak, valamint a kezdő értéket tartalmazó ROM, vagy regiszter.
- Ezzel az iteratív osztási módszerrel az eredményt *közvetlenül* megkapjuk. Feltételezzük, hogy a számok itt normalizált lebegőpontos számok, az osztót és osztandót egy törkifejezésként írjuk fel (mantissa egy normalizált tört).
- Keressük a Q hányados (quotient) értékét. Hogy megkapjuk, mind az osztó, mind pedig az osztandó értékét ugyanazokkal az  $f_k$  értékekkel kell megszorozni, amelyet úgy határozunk meg, hogy a nevező egységnyi legyen az iterációk elvégzése után. Így később a számláló értékéből megkapjuk a Q pontos értékét. Tudjuk, hogy  $D_s$  normalizált tört, ezért így ábrázoljuk:  $D_s = 1-x$ , ahol x-et  $D_s$  határozza meg, és mivel  $D_s$  kisebb 1-nél, így az x is kisebb 1-nél.

# Közvetlen gyors osztó (számítás)

- Az osztás művelete  $f_o$  kiszámolásával kezdődik. Válasszuk  $f_o = 1+x = 1+(1-D_S) = 2-D_S$ . Így  $D_S \times f_o = (1-x)(1+x) = 1-x^2$ . Így sokkal közelebb kerültünk 1-hez, mintha csak a  $D_S$ -et használtuk volna. minden iterációs lépésben a számláló és a nevező is  $f_k$  tényezőkkel szorzódik, és közelebb kerülünk a Q pontos értékéhez. Legyen  $f_1 = 1+x^2$ . Így  $D_S \times f_o \times f_1 = 1-x^4$  és ez tovább ismételhető iteratív módon
- Tehát azt kapjuk, hogy  $D_{Dn+1} = D_{Dn} \times f_n$ .
- Egy kérdés vetődik fel: hogyan válasszuk meg  $f_k$  következő értékét.  $f_1 = 1+x^2 = 1+(1-D_S \times f_o) = 2 - D_S \times f_o$ . Tehát minden egyes új  $f_k$ -t úgy kapunk meg, hogy vesszük az  $f_{k-1}$  és a  $D_S$  (nevező) szorzatának 2's komplementjét. Az iterációs lépéseket a kívánt pontosság eléréséig kell ismételni, amelyet  $f_k$  értéke határoz meg. Amikor  $f_k$  közelítőleg 1, akkor a Q eredmény elegendően közel lesz a kívánt eredményhez (amely az alkalmazástól és a bitek számától függ). Általában előre definiált fix számú iterációs lépést végzünk el. Ezért kell ROM-ot használni, amelyben az  $f_o$  megfelelő kezdeti értékét tároljuk.
- (Példák: könyvben)

# Közvetlen gyors osztó áramköri felépítése



# Példa 1.

Legyen az osztandó  $D_D = 0.4$ , osztó  $D_S = 0.7$ , és **6 iterációs lépésig** számoljunk (7 tizedesjegy pontosságú számokkal). Ekkor  $f_0=2-D_S=2-0.7=1,3000000$ . Kérdés  $Q=D_D/D_S?$  / $D_{Dn+1}=D_{Dn} \times f_n/$

- 0.  $D_{D0}=0,4000000$   $D_{S0}=0,7000000$   $f_0=1,3000000$
- 1.  $D_{D1}=0,5200000$   $D_{S1}=0,9099999$   $f_1=1,0900000$
- 2.  $D_{D2}=0,5668000$   $D_{S2}=0,9918999$   $f_2=1,0081000$
- 3.  $D_{D3}=0,5713911$   $D_{S3}=0,9999344$   $f_3=1,0000656$
- 4.  $D_{D4}=0,5714286$   $D_{S4}=0,9999999$   $f_4=1,0000000$
- 5.  $D_{D5}=0,5714286$   $D_{S5}=1,0000000$   $f_5=1,0000000$
- 6.  $D_{D6}=0,5714286$   $D_{S6}=1,0000000$

Látható, hogy már a 4. Iterációs lépésben megkaptuk a helyes eredményt ( $D_{D4}=0,5714286$ ), mivel  $D_S$  elég közel volt az 1-hez, és  $x=0.3$  volt. ( $x=1-D_S$ ).

Várt érték:

0.57142857142857142857142857142857142857142857

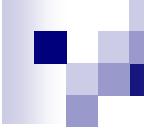
# Példa 2.

Legyen az osztandó  $D_D = 0.1$ , osztó  $D_S = 0.15$ , és 6 iterációs lépésig számoljunk (7 tizedesjegy pontosságú számokkal). Ekkor  $f_0=2-D_S=2-0.15\approx1,8499999$ . Kérdés  $Q=D_D/D_S?$   $D_{D_{n+1}}=D_{D_n} \times f_n.$

- 0.  $D_{DO} = 0,1000000$   $D_{SO} = 0,1500000$   $f_0 = 1,8500000$
  - 1.  $D_{D1} = 0,1850000$   $D_{S1} = 0,2775000$   $f_1 = 1,7224999$
  - 2.  $D_{D2} = 0,3186625$   $D_{S2} = 0,4779938$   $f_2 = 1,5220062$
  - 3.  $D_{D3} = 0,4850063$   $D_{S3} = 0,7275094$   $f_3 = 1,2724905$
  - 4.  $D_{D4} = 0,6171659$   $D_{S4} = 0,9257489$   $f_4 = 1,0742511$
  - 5.  $D_{D5} = 0,6629912$   $D_{S5} = 0,9944868$   $f_5 = 1,0055132$
  - 6.  $D_{D6} = 0,6666464$   $D_{S6} = 0,9999696$  ....

Látható, hogy itt nem kapjuk meg a kívánt értéket ( $D_{D_6}=0,6666464$ ) 6 iterációs lépés alatt. Ezért hogy elérjük a kívánt pontosságot véges számú lépés alatt, ROM-ot kell használni (ahol  $f_o$  kezdeti értékét tároljuk).

## Várt érték:



# Extra bitek kezelése

# Extra bitek kezelése („kerekítési eljárások”)

- Truncation (levágás)
- Rounding (normál kerekítés)
- Zero-bias rounding (zéróhoz kerekítés  $R^*$ )
- Jamming to fix value (von Neumann)
- ROM-Rounding

# Extra bit: probléma

- Pl. Két lebegőpontos szám pl. „összeadása” (mantissa 6-bit), exponens egyeztetés után:

Nagyobb számhoz tartozó mantissa 101010

Kisebb számhoz tartozó mantissa

$$\begin{array}{r} + 110010 \\ \hline 11011010 \end{array} \rightarrow 2$$

Extra bits?

Két bitpozícióval eltolt  
mantissa, hogy  
megegyezzen a nagyobb  
exponensének értékével

# a.) Truncation (levágás)

- Levágás: egyszerűen elhagyjuk az extra biteket.
  - Hibája a pontosság: a kapott/korrigált végleges  $M_{\text{Final}}$  mantissza eltér a valós mantissza  $M_{\text{Real}}$  értékétől:

$$\text{ERR}_{\text{TRUNC}} = M_R - M_F$$

- **Bias:** akár pozitív, akár negatív eltérések (hibák) összege
  - n-bites bias (offset) hibája: tárolt érték mindenkor kisebb lesz, mint a valós/aktuális érték (mindig pozitív bias-t kapunk,  $M_R > M_F$ )
  - Kevesebb extra bittel (pl. 2 helyett 1-el) kisebb lesz a bias, így a hiba is csökken.

# Truncation: példa

- $\sum \text{ERR} \rightarrow \text{bias: mindig pozitív}$

$$\text{ERR}_{\text{TRUNC}} = M_R - M_F$$

cases	MR	MF	ERR <sub>TRUNC</sub>
a	xx0.00	xx0.	0.00
b	xx0.01	xx0.	+0.01
c	xx0.10	xx0.	+0.10
d	xx0.11	xx0.	+0.11
e	xx1.00	xx1.	0.00
f	xx1.01	xx1.	+0.01
g	xx1.10	xx1.	+0.10
h	xx1.11	xx1.	+0.11

$$\text{Bias: } \sum \text{ERR}_{\text{TRUNC}} = +11.0_2 = +3_{10}$$

## b.) Rounding (kerekítés)

- Bias csökkentése a cél, úgy hogy a levágás (truncate) előtt az LSB pozíció értékének felét ( $0.1_2$ ) hozzáadjuk az összeghez:

Nagyobb mantissza	101010	↓	→	2 bitpoz. igazítva
Kisebb mantissza	<u>+ 110010</u>			
		11011010		8 bites eredmény
		<u>+00000010</u>		½ LSB pozíció (=rounding!)
		11011100		Végeredmény (majd truncate!)
				Extra bits

# Pl: Rounding (kerekítés)

- $\sum \text{ERR} \rightarrow \text{bias}$ : hiba itt is ugyan megmarad, de már pozitív és negatív is lehet (bias-a kisebb, mint a levágás esetén)

case	MR	MR+1/2 LSB	MF	ERR <sub>ROUND</sub>
a	xx0.00	xx0.10	xx0.	0.00
b	xx0.01	xx0.11	xx0. <del>1</del>	+0.01
c	xx0.10	xx1.00	xx1. <del>0</del>	-0.10
d	xx0.11	xx1.01	xx1. <del>1</del>	-0.01
e	xx1.00	xx1.10	xx1.	0.00
f	xx1.01	xx1.11	xx1. <del>1</del>	+0.01
g	xx1.10	xy0.00	xy0. <del>1</del>	-0.10
h	xx1.11	xy0.01	xy0. <del>0</del>	-0.01

Változás a  
truncate-hez  
képest!

$$\text{Bias: } \sum \text{ERR}_{\text{ROUND}} = -1.0_2 = -1_{10}$$

**xy:** g.)/h.) eseteknél „carry propagate” van, xx helyett („xx incremented to xy”)

## c.) Round-to-Zero ( $R^*$ rounding)

- Cél. A hiba minimalizálása, lehetőleg zérus bias elérése, kerekítéssel (round-to-zero bias).
- $ERR_{ZERO}$  –kat összeadva a teljes bias értéke nulla lesz.
- **legkisebb a hibája (bias = 0), szemben a többi extra-bit kezelő technikával!**

# Pl: Round-to-Zero ( $R^*$ )

Ha az  $M_R$  „levágandó” tizedes jegyeinek legfelsőbb helyiértékű (MSB) bitje ‘1’, a többi ‘0’, akkor az  $M_R + 1/2$  legkisebb helyiértékű (LSB) bitjére egy ‘1’-est rakunk (majd levágunk), egyébként csak levágás.

case	$M_R$	$M_R + 1/2$ LSB	truncated $M_F$	$ERR_{ZERO}$
a	xx0.00	xx0.10	xx0.	0.00
b	xx0.01	xx0.11	xx0.	+0.01
c	xx0. <del>10</del>	xx1 <del>0</del> ...	xx1.	-0.10
d	xx0.11	xx1.01	xx1.	-0.01
e	xx1.00	xx1.10	xx1.	0.00
f	xx1.01	xx1.11	xx1.	+0.01
g	xx1. <del>10</del>	xx1 <del>0</del> ...	xx1.	<del>+0.10</del>
h	xx1.11	xx1.11	xy0.	-0.01

Eltérés a  
Rounding-hoz  
cékest!

$$\text{Bias: } \sum ERR_{\text{Round\_to\_Zero}} = 0 !$$

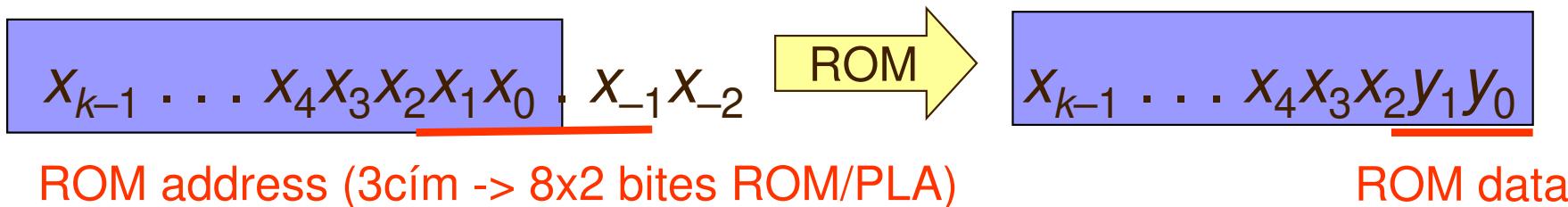
Normál kerekítéshez cékest a c.)/g.) eseteknél egy ‘1’-es lett **direkt beállítva** (force) az LSB helyén (xx1). De csak a g.) esetnél lesz más a hiba értéke ( $ERR_{ZERO}$ ).

## d.) Jamming (~fix értéken rögzítés)

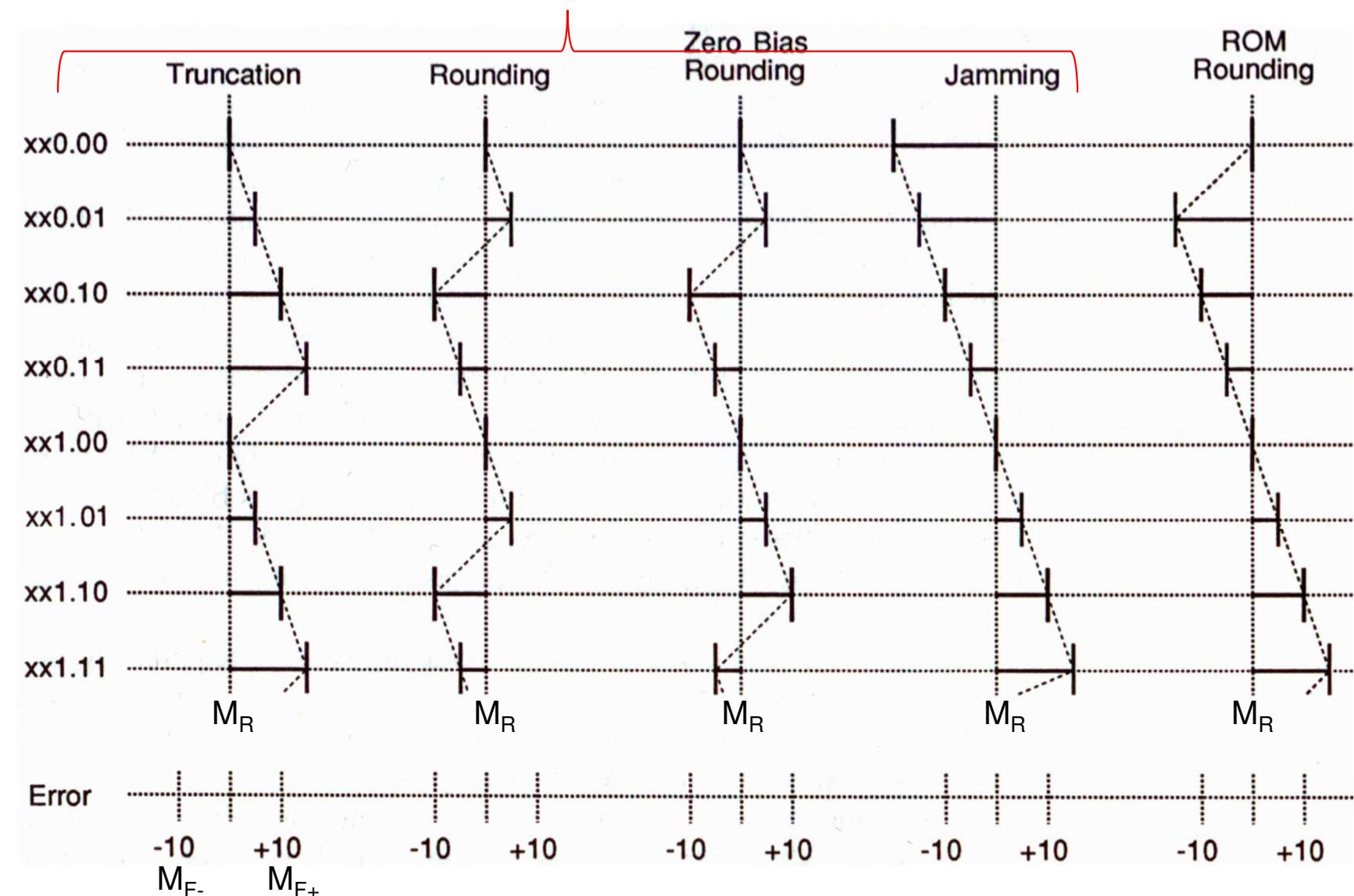
- „Von Neumann rounding” néven is ismert.
- Pl. ***Jam to ‘1’***: LSB bitet fix-en ‘1’-re rögzítik, az extra bitek értékétől függetlenül!
- Cél: Csökkenteni az összhibát (jobb módszer, mint a truncation).
  - Ennek a módszernek ugyan nagyobb a hibája, mint a legtöbb extra bit kezelő módszernek, de idővel ugyanolyan kicsi lesz a bias-a (összhiba), mint a normál kerekítésnek (rounding).
- Nagyon gyors módszer viszont: mint pl. a truncation.
  - itt nincs időszükséglet, mint a kerekítési fázisban, LSB-t minden (pl. ‘1’-re) rögzítjük, ráadásul kicsi lesz bias értéke.

# e.) ROM rounding

- Extra bitek vizsgálata: rounding, majd *döntés* alapján az LSB-biteket hozzáadják a számhoz
- Döntési folyamathoz ROM-ból való értékek kiolvasását használjuk (ROM LUT táblázat kiolvasás). Elve hasonló a hagyományos kerekítéshez, azonban *gyorsabb nála*:
  - Összeadás helyett ROM-ból kiolvasott értékeket használnak.
- Biztosítja, hogy az LSB-nél nagyobb bitpozíciókba nem kell „carry-t propagáltatni” (mint rounding-nál): ezáltal gyorsabb
- Bias jól kontrollálható (akár zérus is lehet végül).



# Extra-bit kezelő módszerek hibáinak összehasonlítása:





# Számítógép Architektúrák II.

(MIVIB344ZV)

5. előadás: CU – vezérlő egységek: huzalozott,  
mikroprogramozott módszerek

Előadó: Dr. Vörösházi Zsolt  
[voroshazi.zsolt@mik.uni-pannon.hu](mailto:voroshazi.zsolt@mik.uni-pannon.hu)

# Jegyzetek, segédanyagok:

- Könyvfejezetek:

- <http://www.virt.uni-pannon.hu> → Oktatás → Tantárgyak → Számítógép Architektúrák II.  
□ (chapter05.pdf)

- Fóliák, óravázlatok .ppt (.pdf)

- Feltöltésük folyamatosan

# Vezérlő egységek általánosan

- A számítógép vezérlési funkcióit ellátó *szekvenciális* egység (CU – Control Unit)
- **Feladata:** az operatív tárban lévő gépi kódú utasítások értelmezése, részműveletekre bontása, és a szekvenciális (sorrendi) hálózat egyes funkcionális részeinek vezérlése (vezérlőjel- és cím-generálás)
- Vezérlő egység tervezésének lépései:
  - megfelelő technológia, és rendszerkomponensek kiválasztása
  - komponensek összekapcsolása a működési sorrendnek megfelelően
  - RTL leírás alkalmazása az akciók ill. adatátvitel pontos leírására
  - **adatút (data-path)** megtervezése (*legfontosabb!*)
  - kívánt vezérlő jelek azonosítása, meghatározása

# Adatút (Data-path) tervezés szempontjai

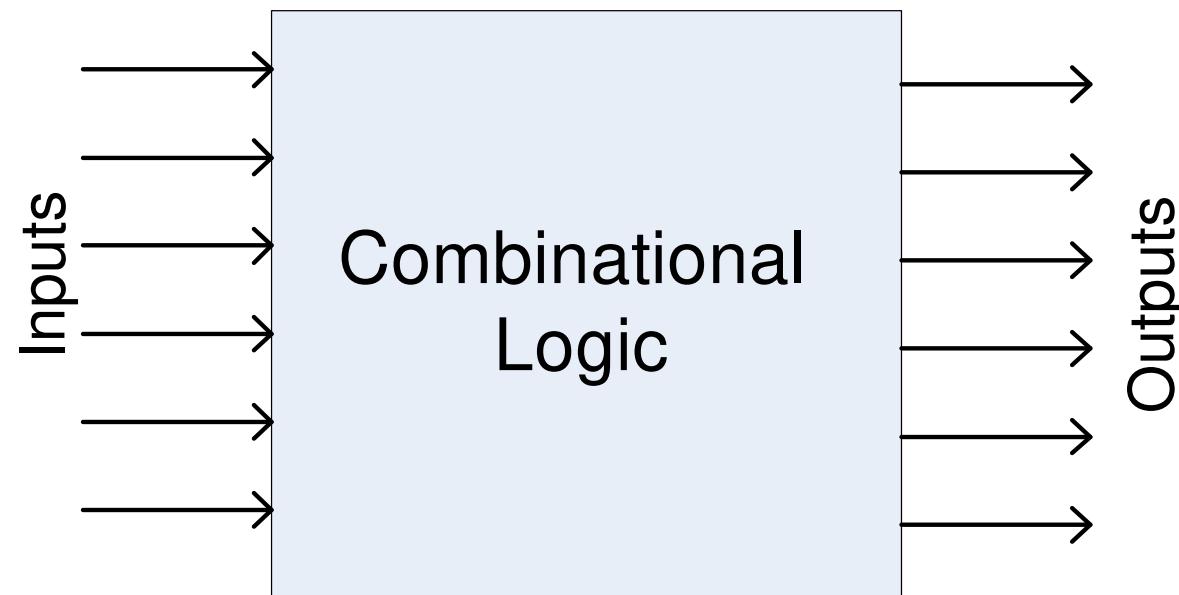
- Gazdaságosság (költség)
- Interfész szükséglet (protokollok)
- Sebesség (S)
- Felület (A)
- Energia (disszipált teljesítmény) (P, D)
- Dinamikai tartomány (számrendszerek)
- Rugalmasság (többcélúság)
- Kezelhetőség (probléma, hiba során)
- Környezet (pl. ipari v. irodai használat?)

# Vezérlő egységek fajtái:

- I. **Huzalozott (klasszikus)** módszerek (pl. korai RISC architektúrák):
  - Mealy-modell,
  - Moore-modell
- II. **Mikrogramozott („reguláris” vezérlési szerkezettel** – pl. CISC, ill. mai RISC architektúrák):
  - Horizontális mikrokódos vezérlő,
  - Vertikális mikrokódos vezérlő.
- III. **Programozható logikai eszközök (PLD):**
  - Maszk-programozható: PLA, PAL, PROM, CPLD,
  - Újrakonfigurálható (szoftveresen): FPGA

# Ismétlés:Kombinációs hálózatok

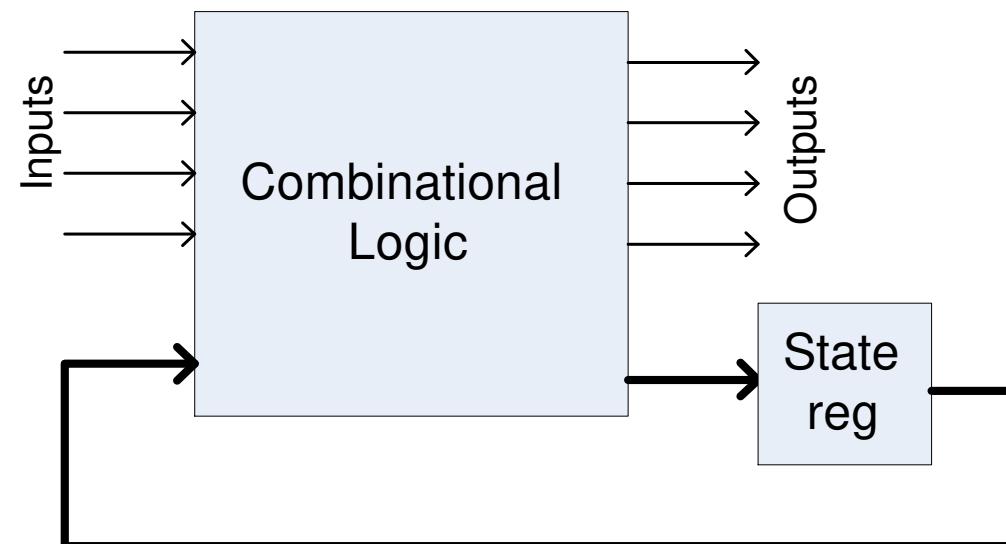
- **(K.H.) Kombinációs logikai hálózatról** beszélünk:  
ha a mindenkor kimeneti kombinációk értéke  
csupán a bemeneti kombinációk pillanatnyi értékétől  
függ
  - tároló „kapacitás”, vagy memória nélküli hálózatok!



# Ismétlés: Sorrendi hálózatok

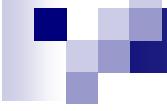
- **(S.H.) Sorrendi (szekvenciális) logikai hálózatról** beszélünk: a mindenkorai kimeneti kombinációt, nemcsak a pillanatnyi (elsődleges) bemeneti kombinációk, hanem a korábban fennállt állapot kombinációk és azok sorrendje is befolyásolja. A **szekunder /másodlagos kombinációk** = **tárolt állapotok** segítségével a hálózatok képessé válhatnak arra, hogy az ugyanolyan bemeneti kombinációhoz más-más kimeneti kombinációt szolgáltassanak, attól függően, hogy a bemeneti kombináció fellépésekor, milyen értékű a szekunder kombináció, pl. a State Register tartalma.

Vezérlő egységek  
alapjául szolgáló  
sorrendi hálózat!



# TAC – Időzítő-vezérlő egység:

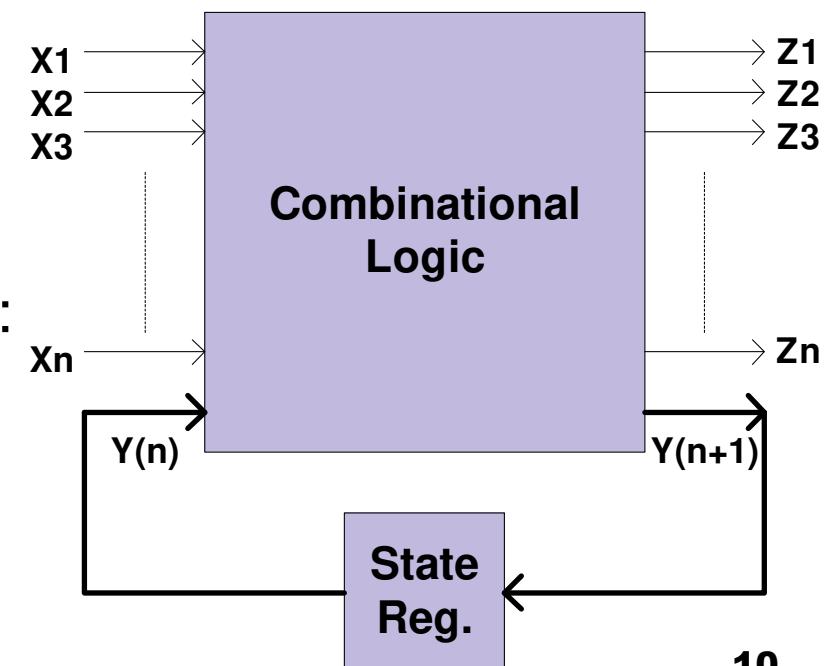
- Az időzítő (ütemező) határozza meg a vezérlő jelek előállításának sorrendjét.
- Egy időzítő-vezérlő (TAC) egység általános feladata az egyes funkciók megvalósítását végző áramköri elemek (pl. ALU, memória elemek) összehangolt működésének biztosítása.
- Az időzítő-vezérlő áramkörök ***szekvenciális rendszerek*** – mivel az áramköri egységek tevékenységének egymáshoz viszonyított *időbeli sorrendisége*t biztosítják – melyek az aktuális *kimenet* értékét a *bemenet*, és az *állapotok* függvényében határozzák meg.



# I. Klasszikus vezérlési módszerek: Huzalozott vezérlő egységek

# 1.) Mealy-modell

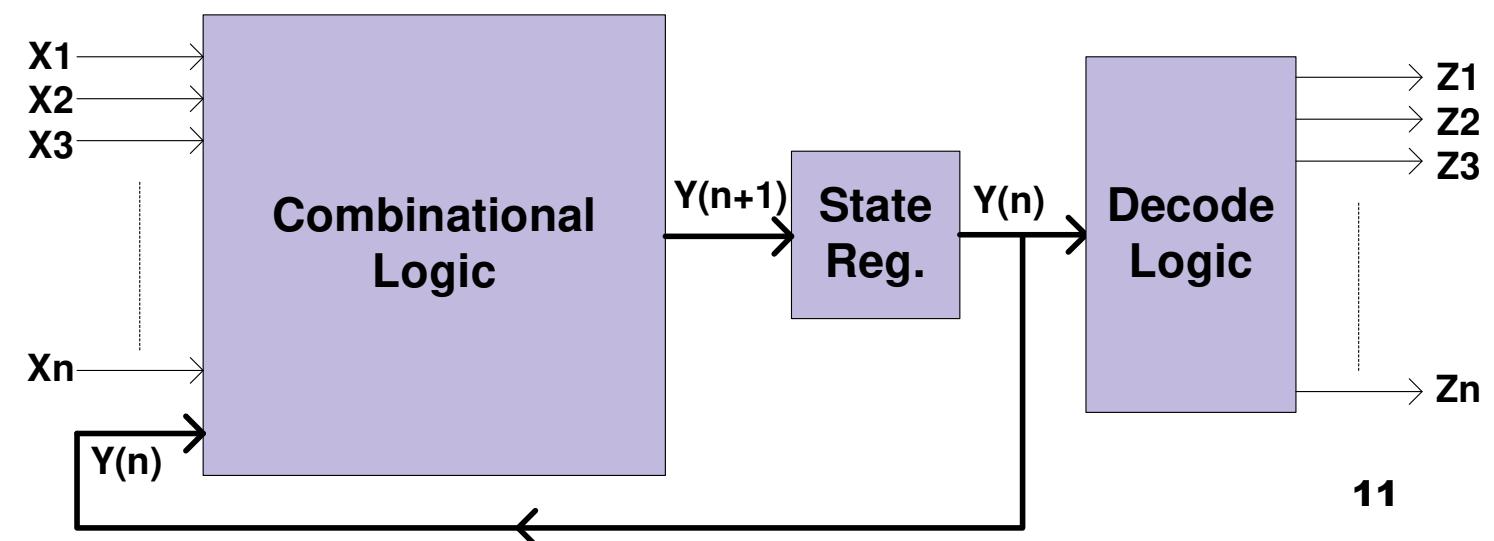
- A sorrendi hálózatok egyik alapmodellje. Késleltetés: a kimeneten az eredmény véges időn belül jelenik meg! Korábbi értékek visszacsatolódnak a bemenetre: kimenetek nemcsak a bemenetek pillanatnyi értékétől, hanem a korábbi állapotuktól is függnek. Problémák merülhetnek fel az állapotok és bemenetek közötti szinkronizáció hiánya miatt (változó hosszúságú kimenet - dekódolás). Ezért alkalmazzák legtöbb esetben a második, Moore-féle automata modellt.
- Három halmaza van: (Visszacsatolni az állapotregisztert a késleltetés miatt kell)
  - X – a bemenetek,
  - Z – a kimenetek,
  - Y – az állapotok halmaza.
- Két leképezési szabály a halmazok között:
  - $\delta(X_n, Y_n) \rightarrow Y_{n+1}$  : következő állapot fgv.
  - $\mu(X_n, Y_n) \rightarrow Z_n$  : kimeneti fgv.

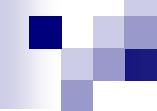


# 2.) Moore-modell

- A kimenetek közvetlenül csak a pillanatnyi állapottól függnek (bemenettől függetlenek v. közvetve függenek). Tehát a kimenetet nem a bemenetekhez, hanem az állapotoknak megfelelően szinkronizáljuk.
- Három halmaza van:
  - X – a bemenetek,
  - Z – a kimenetek,
  - Y – az állapotok halmaza.
- Két leképezési szabály:
  - $\delta(X_n, Y_n) \rightarrow Y_{n+1}$  : köv. állapot fgv.
  - $\mu(Y_n) \rightarrow Z_n$  : kimeneti fgv.

**Input  $\Rightarrow$  Next-State  $\Rightarrow$  Present-State  $\Rightarrow$  Output**





# Szekvenciális vezérlő rendszerek – egyedi késleltetéses módszer

# Szekvenciális vezérlő rendszerek

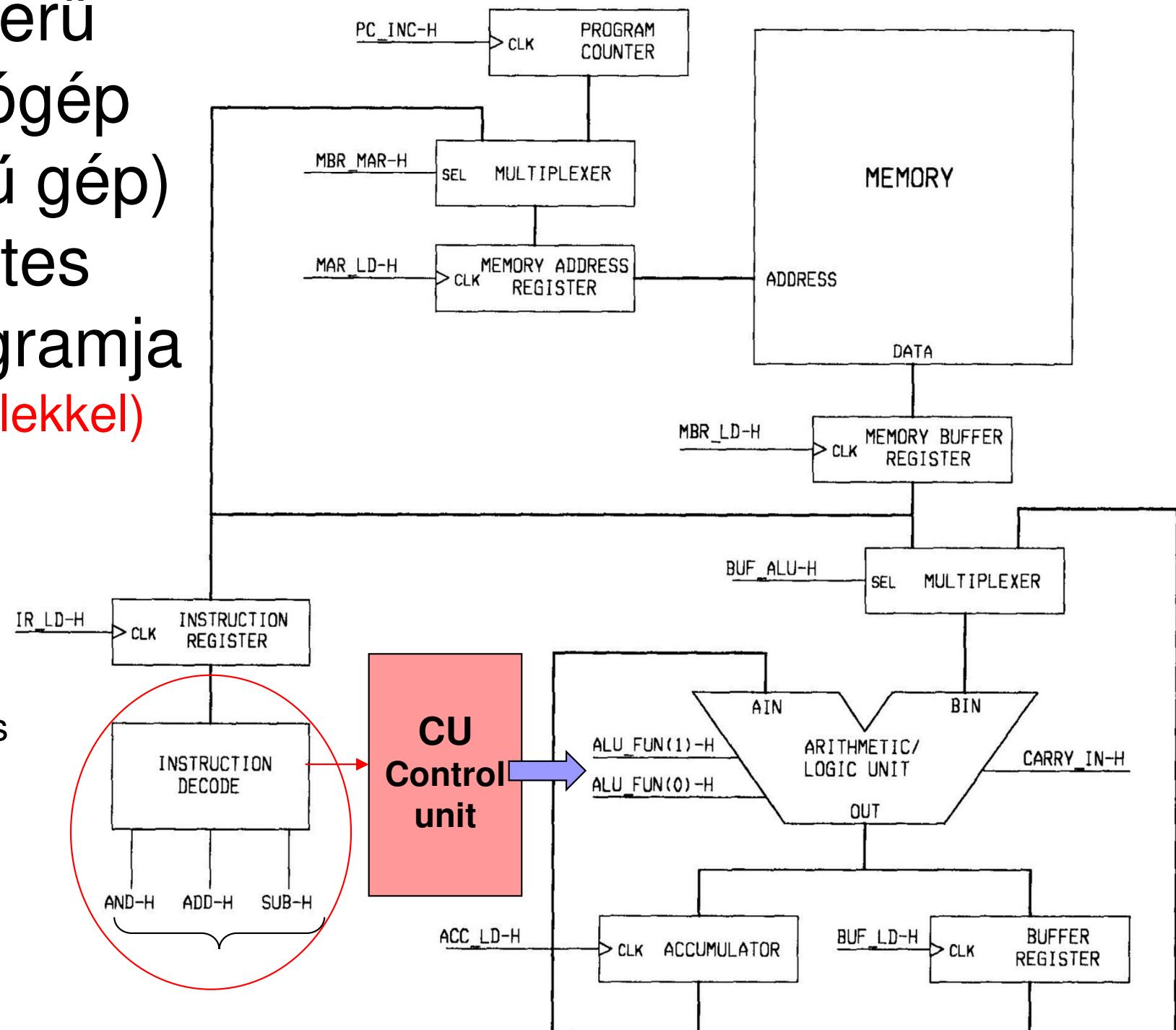
- A rendszer állapotait tároljuk, *külön regisztereket* definiálunk egy egyszerű számítógép (egycímű gép) blokkdiagramját felhasználva.
  - FSM: Finite State Machine – véges állapotú autómata
- Regiszter-transzfer műveletek sora mutatja be ennek a *késleltetéses* vezérlő rendszernek a működését.



# Egyszerű számítógép (egycímű gép) részletes blokkdiagramja (vezérlő jelekkel)

Példa: 3  
egyszerű utasítás

- ADD
- SUB
- AND



# Példa: Egycímű gép vezérlési funkciója

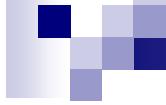
- Mivel példaként három egyszerű két-operandusú utasítást (AND, ill ADD, SUB) akarunk végrehajtani egy egycímű gépen, ezért a második operandus értékét az ACC-ből kell betölteni!
- Ehhez az ALU néhány alapvető funkciója:

**Késleltetések!**

$T_{REG} = 40\text{ ns}$

$T_{MEM} = 200\text{ ns}$

ALU_FUN		OUT function	
0	0	bitenkénti AND (A_In, B_In)	[40 ns]
0	1	bitenkénti OR (A_In, B_In)	[40 ns]
1	0	inverz NOT (B_In)	[40 ns]
1	1	bináris ADD (A_In, B_In)	[80 ns]

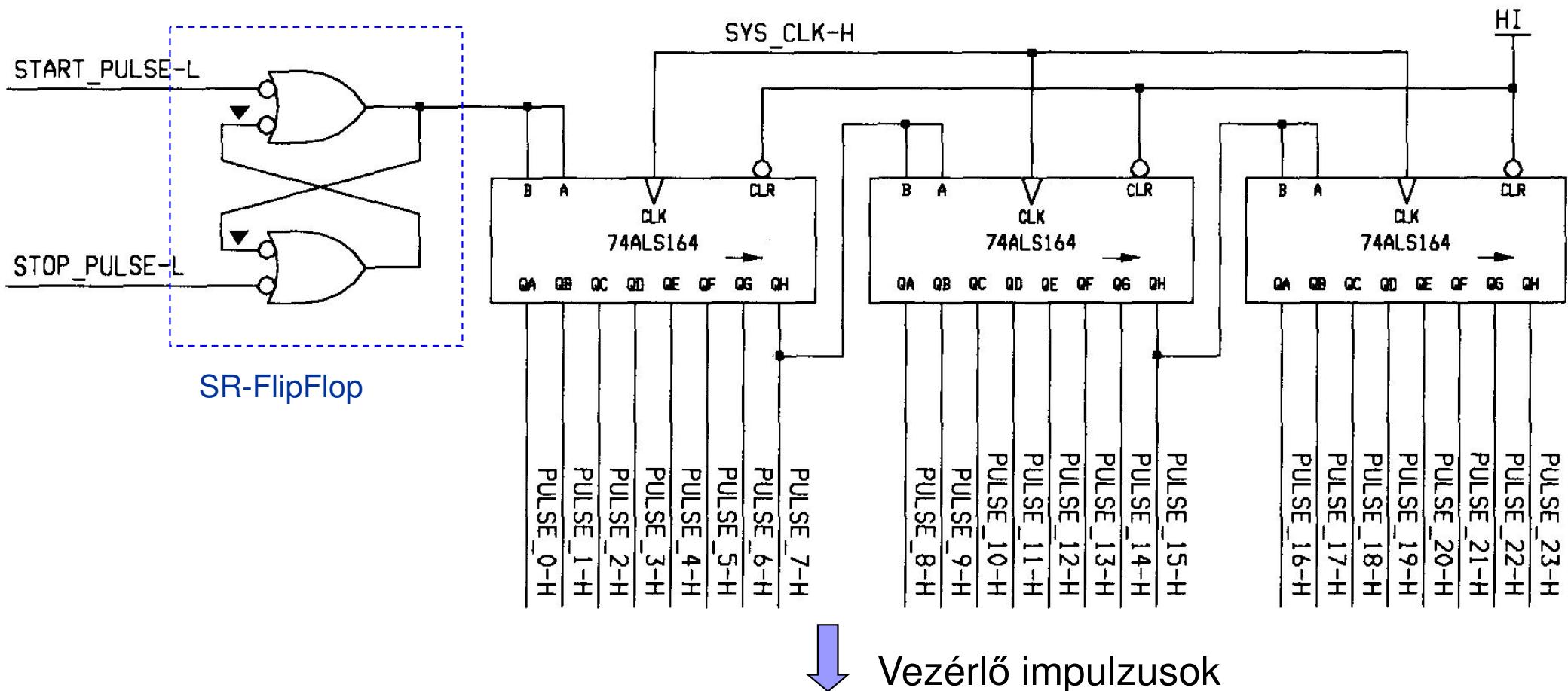


# Szekvenciális vezérlő rendszerek tervezése Shift-regiszteres időzítővel

# Vezérlő Shift-regiszteres időzítővel

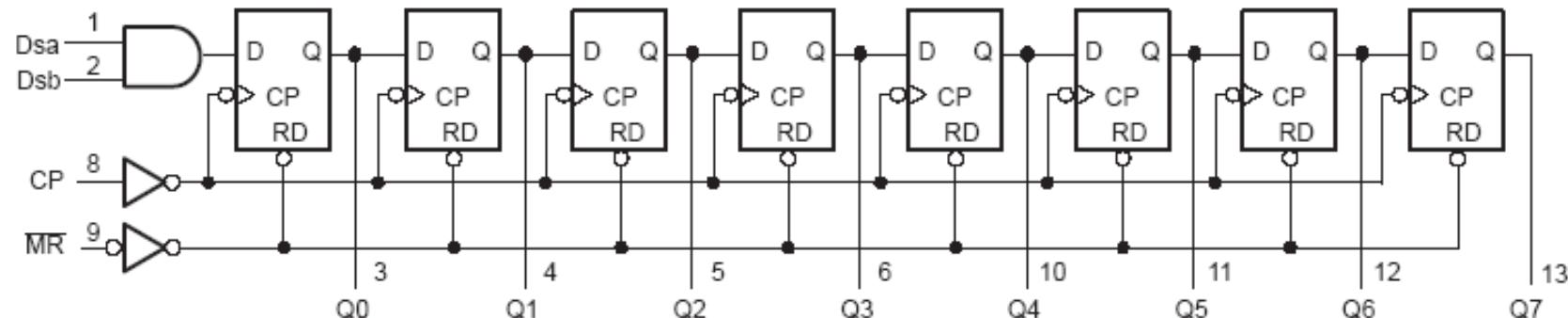
- Ez a megvalósítás az egyedi késleltetéses módszerhez nagyban hasonlít, ugyanis:
  - *Adatút-diagrammot* használunk a vezérlőjelek azonosítására,
  - *Folyamat-diagrammot* a regiszter-transzferek (RTL) ábrázolására, míg
  - *Idő-diagrammot* a vezérlőjelek kölcsönhatásának leírására.
- Hárrom 8-bites Shift-regiszter segítségével generálja az impulzusokat (közvetve a vezérlő jeleket is).

# Sorrendi vezérlő egységek megvalósítása Shift-regiszterekkel



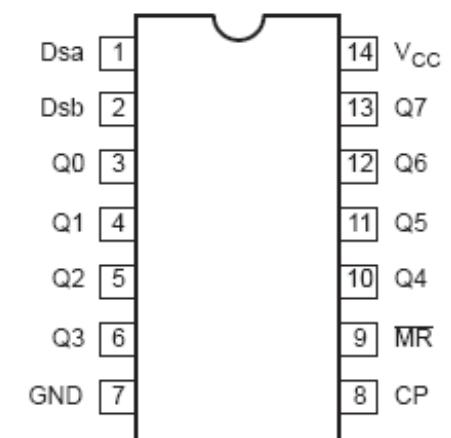
# 74ALS164

## ■ 8-bites serial in/paralel out shift regiszter



- D<sub>sa</sub> / D<sub>sb</sub>: két adatbemenet (egyiket lehet engedélyezőnek is definiálni)
- Q<sub>n</sub>: adatkimenetek
- CP: clock pulse
- MR: low master-reset

Több 74ALS164-et összekötve szinkron shift reg. kapunk



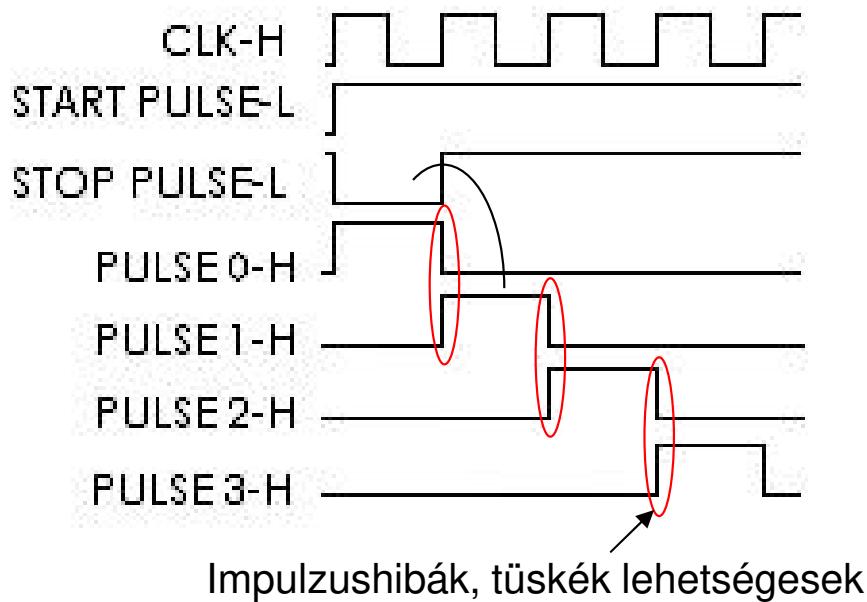
# 1. módszer: „Nem-átlapoló” impulzusok

- Inicializáláskor a kívánt impulzus előállítását a START\_PULSE\_L beállításával érhetjük el, amelyet a teljes folyamat végéig „L” alacsony-aktív szinten tartunk. A következő órajelciklusban a PULSE\_0-H jel állítódik be magas jelszintre *rövid* 40 ns-os impulzus ideig.
- A STOP\_PULSE-L vezérlőjelet a PULSE\_0-H jel negálásával kapjuk meg (abban az esetben, ha egy ciklus, azaz 40ns ideig tart). Az órajel minden egyes felfutó élére shiftelődik az impulzus. Ekkor nem lapolódnak át, mivel egymás utáni 40ns –os részekből állnak össze, és a megfelelő PULSE\_XX kimeneti vezérlőjelek OR kapcsolatából képződnek.

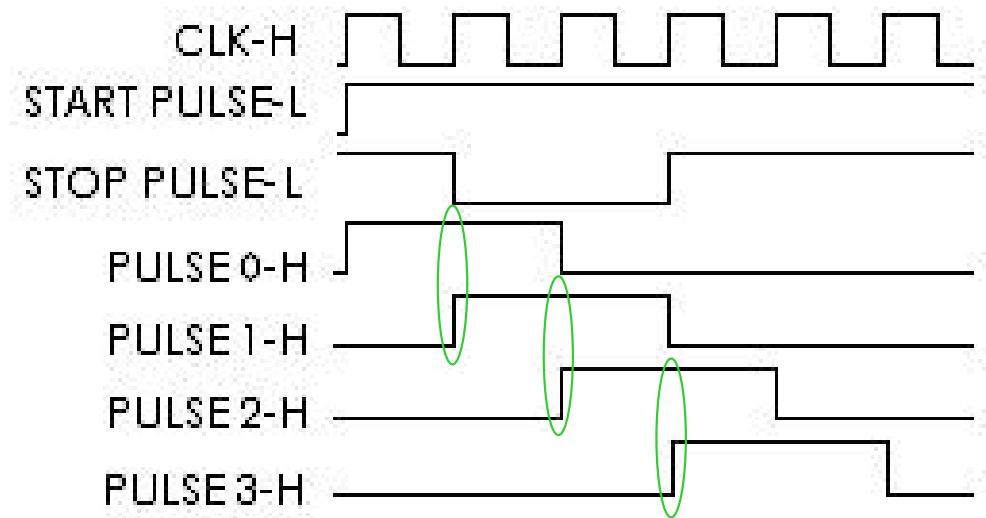
## 2. módszer: „Átlapoló” impulzusok

- Bizonyos esetekben azonban nem kívánt impulzushibák ún. tüskék (glitch) keletkezhetnek (pl. ha az egyik jel alacsony szinten marad, a másik viszont magas jelszintre vált). Ezeket a nem megfelelő jelváltásokat vagy SET-RESET flip-flopok használatával küszöbölhetjük ki, vagy pedig alkalmazni kell az **átlapoló impulzusok** technikáját.
- Ezt az idődiagramot a jobboldali ábra mutatja. Két egység *hosszú impulzusokat* (80ns) egyszerűen létrehozhatunk a PULSE\_1-H jel és az invertált STOP\_PULSE-L jel OR kapcsolatával (a bal oldali ábra jeleiből!). Az így kapott impulzus mentes lesz a hibáktól, és kiküszöbölhetők a hazárdjelenségek.

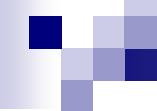
# „Nem-átlapoló” és „átlapoló” impulzusok bemutatása idődiagramon



Nemátlapoló rövid impulzusok (40ns)



Átlapoló hosszú impulzusok (80ns)



# Mikrokódos vezérlők – reguláris vezérlési struktúrák

# Ismétlés: Vezérlő egységek

- Általánosságban: a vezérlő egység (CU) feladata a memóriában lévő gépi kódú program utasításainak
  - értelmezése (decode),
  - részműveletekre bontása,
  - és ezek alapján az egyes funkcionális egységek vezérlése (**a vezérlőjelek megfelelő sorrendben történő előállítása**).

# Klasszikus vs. reguláris módszer

- Eddig a „klasszikus”, késleltetéses módszereket tárgyaltuk (*huzalozott* és *shift-regiszteres* példákkal). A rendszer tervezésekor, miután a feladat elvégzéséhez szükséges vezérlőjeleket definiáltuk, meg kell határozni a kiválasztásuk sorrendjét, és egyéb specifikus információkat (rendszer ismeret, tervezési technikák, viselkedési leírások – pl.VHDL). Vezérlő egység:
  - Kombinációs hálózat (hard-wired = huzalozott), vagy
  - FSM: véges állapotú automata alapú.
- Wilkes (1951): A komplex, többcímű (operandusú), illetve vezérlési szerkezeteket „**reguláris módszerrel**” lehet gyorsítani, egyszerűsíteni: nevezetesen gyors memória elemeket kell használni az utasítássorozatok tárolásánál. Ugyan a klasszikus módszernél használt állapotgépekkel (FSM) modellezik a reguláris vezérlő egység működését, majd ezt a modellt transzformálják át mikrokódos memóriát (ami nem azonos az operatív memóriával!) használva. Az adatútvonal vezérlési pontjait memóriából (ROM) kiolvasott *vertikális-* vagy *horizontális-mikrokódú utasításokkal* állítják be!

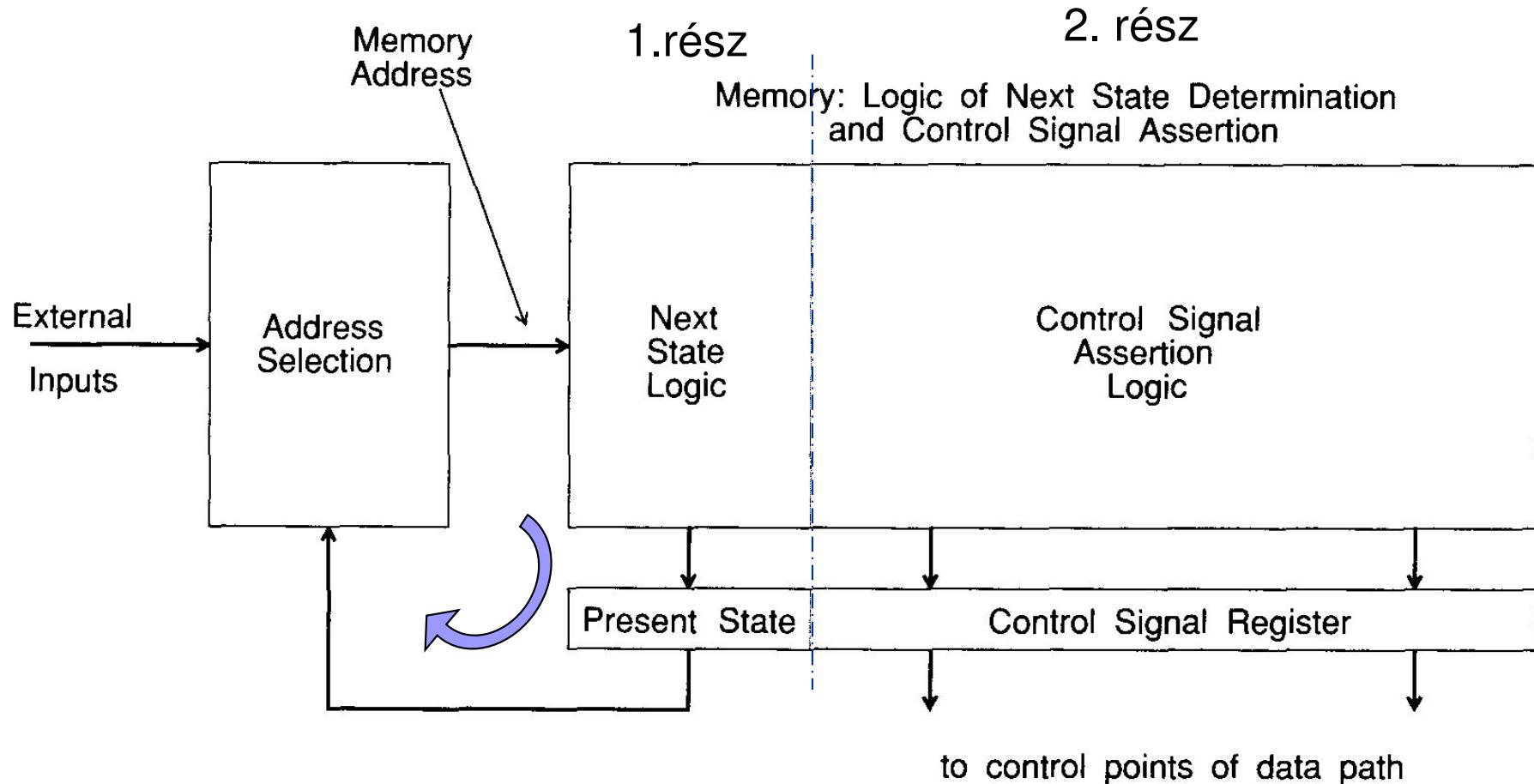
# Reguláris módszer: mikrokódos vezérlés tulajdonságai

- **Mikrokód:** gépi kódú utasításokat (IR) legalacsonyabb szintű áramköri (hw) utasítások sorozatára leképező köztes kód leképezés →
- Szerepe: **értelmezés** (interpreter / translator) a fenti két szint között:
  - A gépi kódú utasítások változtatásának lehetősége (RISC, CISC), anélkül hogy a HW változna
  - Mai rendszerek olvasható mikrokódját gyors memóriában (általában ROM), vagy PLD-ben tárolják (írható esetben RAM, vagy Flash is lehet)
- Alkalmazás: CPU, GPU, lemezvezérlők, NPU (network processor unit)

# Mikroprogram = mikroutasítások sorozata

- Példa: Tipikus mikroprogram (~RTL-szintű utasítás szekvenciák sorozata):
  1. Load Reg[1] to the "A" side of the ALU
  2. Load Reg[7] to the "B" side of the ALU
  3. Select the ALU to perform 2's-comp addition
  4. Select the ALU's carry input to zero
  5. Store the result value in Reg[8]
  6. Update the "condition codes" with the ALU status flags ("Negative", "Zero", "Overflow", and "Carry")
  7. Microjump (MicroPC) for the next microinstruction

# FSM megvalósítása Memóriával

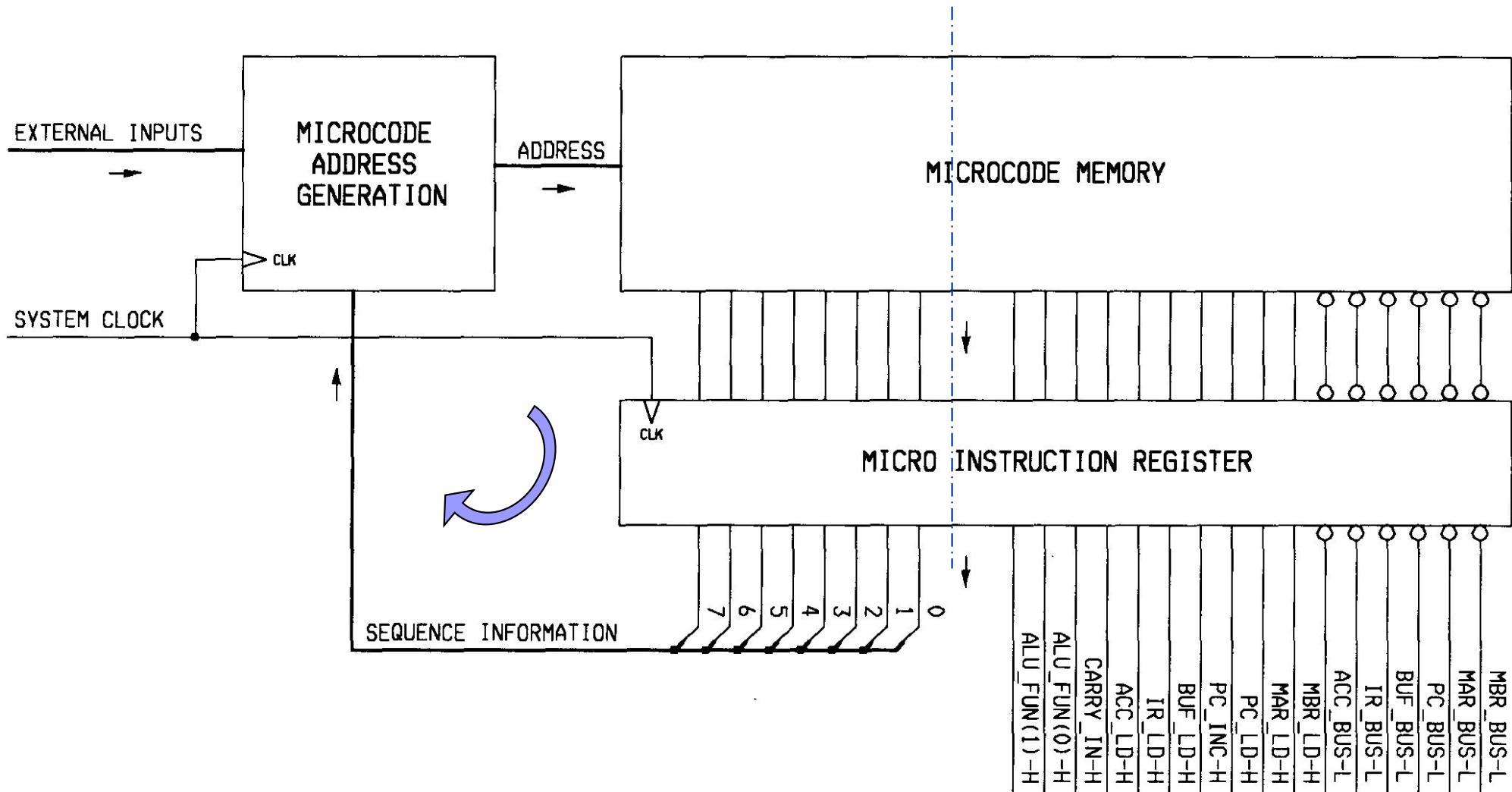


- 1.rész: szabályozza az eszköz működését a megfelelő állapotok *sorrendjében*
- 2.rész: szabályozza az adatfolyamot a megfelelő vezérlőjelek *beállításával* (assertion) az adatúton (vezérlési pontokon)

# FSM megvalósítása Memóriával (folyt)

- **Address Selection:** (mint új elem) a következő utasítás (*Next State*), és beállítani kívánt vezérlőjel (*control signal assertion*) címére mutat a memóriában (~ Id. MAR).
- A memória címet (**memory address**-t) külső bemenő jelek és a *present state* együttesen határozzák meg. E cím segítségével megkapjuk az adott vezérlő információ pontos helyét a memóriában, ill. ez az információ, mint új állapot betöltődik a generált vezérlőjeleket tároló (*Control Signal Register*).
- **Next-State** kiválasztásához szükséges logikai memória méretét az aktuális állapotok száma, az állapotdiagram komplexitása, és a bemenetek száma határozza meg.
- **Control Signal** generálásához szükséges logikai memória méretét a bemenetek száma, a függvény (vezérlő jel) komplexitása, és a vezérlőjelek száma határozza meg.

# „Általános” Mikrokódos vezérlő



# Általános Mikrokódos vezérlő felépítése

- **Micro Instruction Register:** a „Present State” (aktuális állapot) regisztert + a Control Signal regisztert egybeolvasztja (az adatút vezérlővonalainak beállítása / kiválasztása). Mikroutasítások sorrendjében generálódik a vezérlőjel!
- **Microcode Memory:** a Control Signal Assertion Logic vezérlőjel generálás/beállítás + „Next-State” kiválasztása (mikroprogram eltárolása) összevonása
- **Microcode Address Generator:** a vezérlő jelet az aktuális mikroutasítások lépéseként sorban generálja, de címkiválasztási folyamat komplex. **Sebesség a komplexitás rovására változhat!** (komplexebb vezérlési funkciót alacsonyabb sebességgel képes csak generálni). A következő cím kiválasztása még az aktuálisan futó mikroutasítás végrehajtása alatt végbemegy! Számlálóként működik: egyik címről a másik címre inkrementálódik (mivel a mikroutasításokat tekintve szekvenciális rendszerről van szó). Kezdetben resetelni kell.

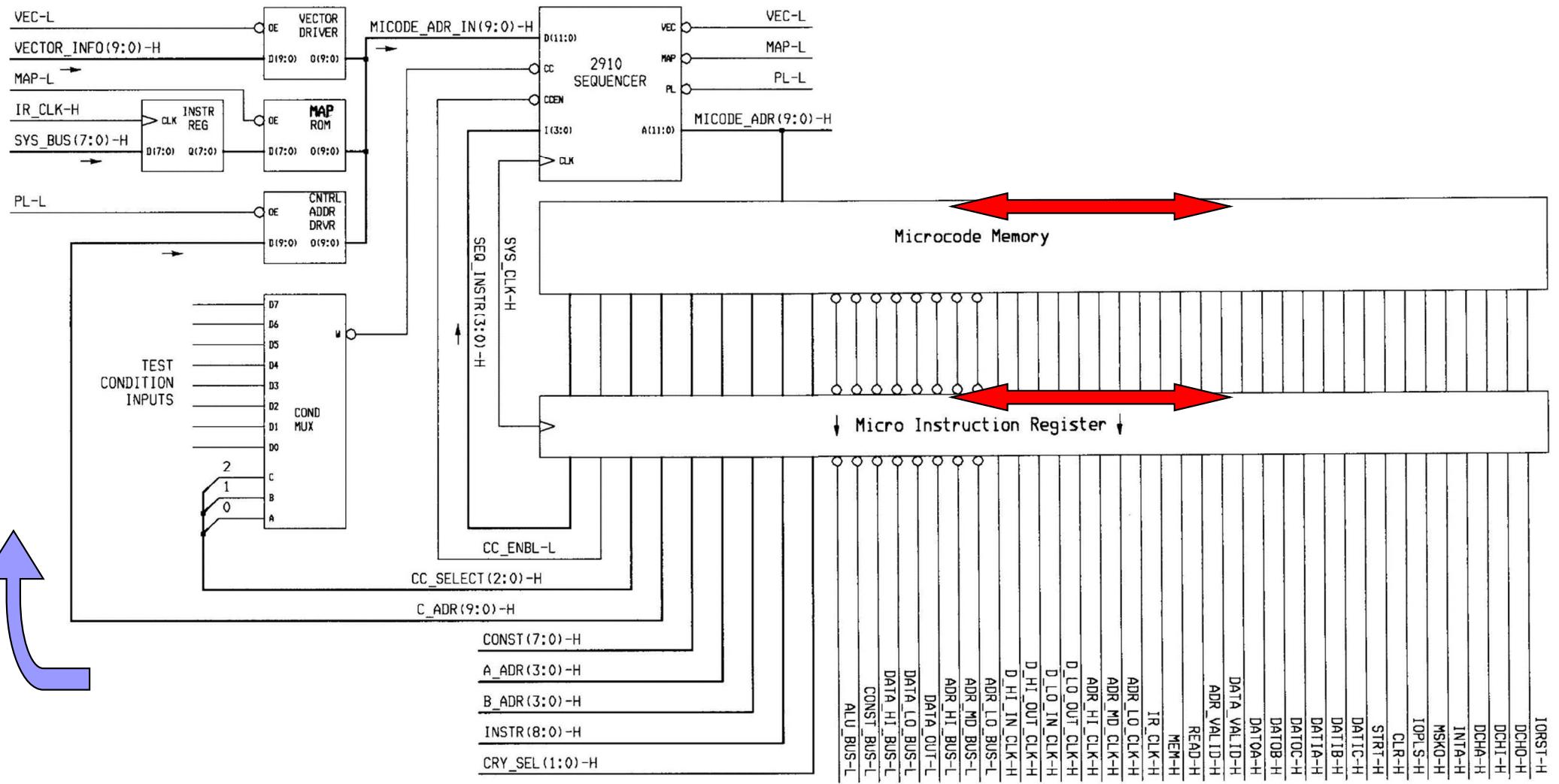
# „Általános” Mikrokódos vezérlők tulajdonságai

- Egy gépi ciklus alatt egy **mikroprogram** fut le (amely mikroutasítások sorozatából áll). A műveleti kód (utasítás opcode része) a végrehajtandó mikroprogramot jelöli ki. A mikrokódú memória általában csak statikus módon olvasható gyárilag konfigurált ROM, ha írható is, akkor dinamikus mikroprogramozásról beszélünk.
- Ha a mikroprogram utasításai szigorúan *szekvenciálisan* futnak le, akkor a címüket egy egyszerű számláló inkrementálásával megkaphatjuk. Memóriából érkező bitek egyik része a következő *cím kiválasztását* (Sequence Information), míg a fennmaradó bitek az *adatáramlást* biztosítják.
- Mai gyors félvezető alapú memóriáknak köszönhetően kis mértékben lassabb, mint a huzalozott vezérlő egységek, mivel ekkor a memória elérési idejével (~ns) is számolni kell (nem csak a visszacsatolt aktuális állapot késleltetésével.)

# 1.) Horizontális mikrokódos vezérlő

- minden egyes vezérlőjelhez saját vonalat rendelünk, ezáltal horizontálisan megnő a mikro-utasításregiszter kimeneteinek száma, (**horizontálisan** megnő a mikrokód). Minél több funkciót valósítunk meg a vezérlőjelekkel, annál **szélesebb** lesz a mikrokód.
- Ennek köszönhetően ez a **leggyorsabb** mikrokódos technika, mivel minden bit független egymástól ill. egy mikrokóddal többszörös (konkurens) utasítás is megadható.
  - Pl: a megfelelő funkcionális egységeket (memória, ALU, regiszterek stb.) egyszerre tudjuk az órajellel aktiválni, ezáltal egy órajelciklus alatt az információ minden két irányba átvihető. Növekszik a sebesség, mivel nincs szükség a vezérlőjelek dekódolását végző dekódoló logikára. Így minimálisra csökken a műveletek ciklusideje.
- Azonban nagyobb az **erőforrás** szükségléte, **fogyasztása**.

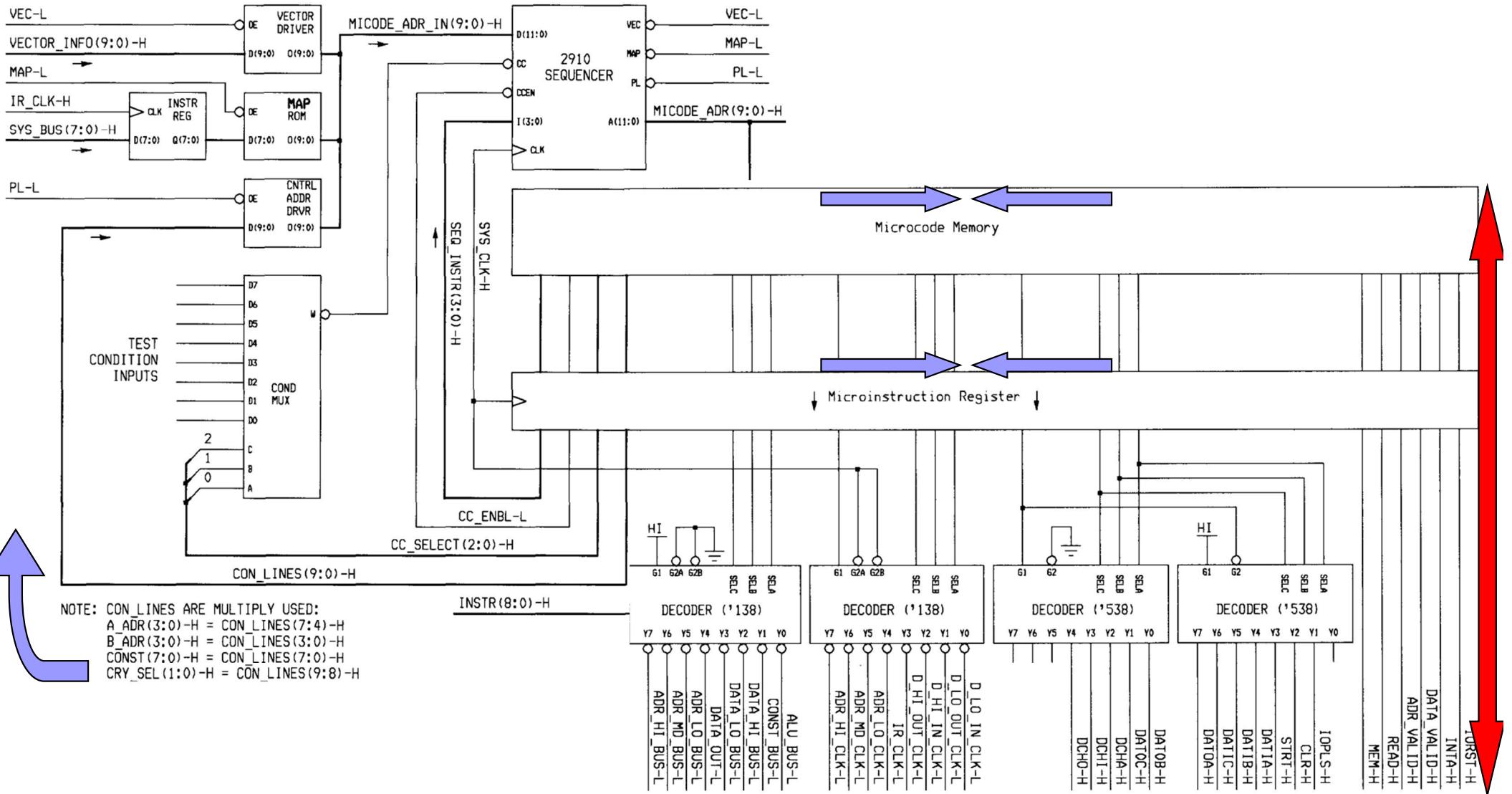
# Horizontális mikrokódos vezérlő



## 2.) Vertikális mikrokódos vezérlő

- Nem a sebességen van a hangsúly, hanem hogy **takarékoskodjon** az erőforrásokkal (fogyasztás, mikrokódban a bitek számával), ezért is **lassabb**.
- Egyszerre csak a szükséges (korlátozott számú) biteket kezeljük, egymástól nem teljesen függetlenül, mivel közük egyszerre csak az egyiket állítjuk be (deködöljük). A jeleket ezután **deködolni** kell (több időt vesz igénybe). A kiválasztott biteket megpróbáljuk minimális számú vonalon keresztül továbbítani.
- A műveletek párhuzamos (konkurens) végrehajtása korlátozott. Dekódolás:  $\log_2(N)$  számú dekódolandó bit -> N bites kimeneti busz. Több mikroutasítás szükségeltetik  $\Rightarrow$  így a mikrokódú memóriát „**vertikálisan**” meg kell növeli.

# Vertikális mikrokódos vezérlő





# Számítógép Architektúrák II.

(MIVIB344ZV)

6. előadás: Utasítás végrehajtás folyamata,  
címzési módok. RISC-CISC processzorok

Előadó: Dr. Vörösházi Zsolt  
[voroshazi.zsolt@mik.uni-pannon.hu](mailto:voroshazi.zsolt@mik.uni-pannon.hu)

# Jegyzetek, segédanyagok:

- Könyvfejezetek:

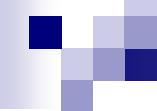
- <http://www.virt.uni-pannon.hu> → Oktatás → Tantárgyak → Számítógép Architektúrák II.  
□ (chapter04.pdf)

- Fóliák, óravázlatok .ppt (.pdf)

- Feltöltésük folyamatosan

# Utasítás végrehajtás folyamata

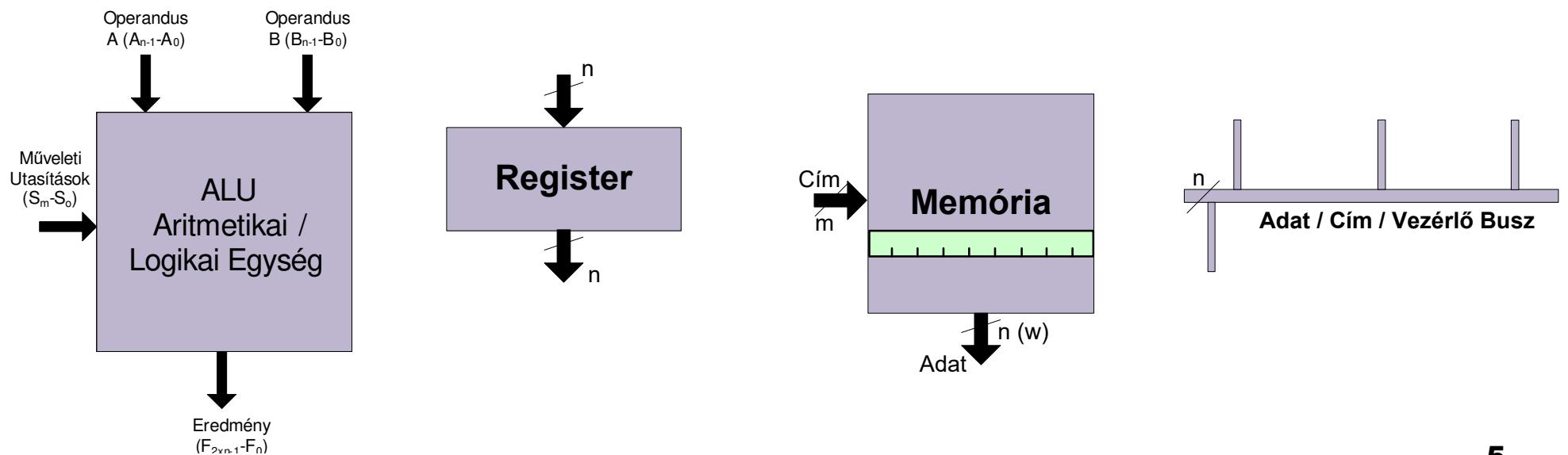
- Utasítás kódok
  - Programvezérlő utasítások
- Címzési módok
- RISC vs. CISC processzor architektúrák



# Alapvető digitális építőelemek (rövid áttekintés)

# Legfontosabb digitális építőelemeink:

- ALU
- Memóriák
- Adat / Cím / Vezérlő Buszok
- Regiszterek, De/Multiplexerek, De/Kódoló áramkörök



# ALU egység

- Az **ALU** egység két különböző  $n$ -bites bemeneti résszel (A, B) rendelkezik, és egy  $n$ -bites\*\* kimeneti vonallal (F). A szelektáló (S) jelek segítenek a megfelelő műveletek kiválasztásában. Az ALU egység egy algoritmus utasításainak megfelelően aritmetikai ill. logikai műveleteket hajt végre.
- Eml: funkcionális teljesség,  $+, -, *, /$  és Logikai fgv.
- (Korábban részletesen: [chapter\\_03.pdf](#))
- \*\* eredmény valójában  $n+1$ , vagy  $2^*n$  bites

# Regiszterek

- A következő fontos elem a **regiszter**. Olyan szélesnek kell lennie, hogy benne, a buszokról, memóriákból, ALU-ból érkező információ eltárolható legyen.
- Adott vezérlőjelek hatására a bemenetén lévő adatokat betölti, és ideiglenesen eltárolja. Más vezérlőjelek hatására a kimenetére teszi a tárolt adatokat, vagy például egy vezérlőjel hatására, *lépteti (shift-eli)* a benne lévő adatokat.

# Memória egységek

- Az ALU által kezelt / végrehajtott adatok a **memóriában** (tároló rekeszek lineáris tömbjében) tárolódnak el. A memória rekeszei általában olyan szélesek, amilyen széles az adatbusz. Például, legyen  $n$ -bit ( $w$ ) széles, és álljon  $2^m$  számú rekeszből. Ekkor  $m$  számú címvezetékkel címezhető meg. Az adatbuszon kétirányú (írás/olvasás) kommunikáció is megengedett. Memória a von Neumann architektúrát követi: tehát az utasítások (program/kód) és az adatok egy helyen tárolódnak, nem pedig külön-külön (Harvard architektúra). A programot is adatként tárolja a memória.

# Adatbuszok – adatvonalak

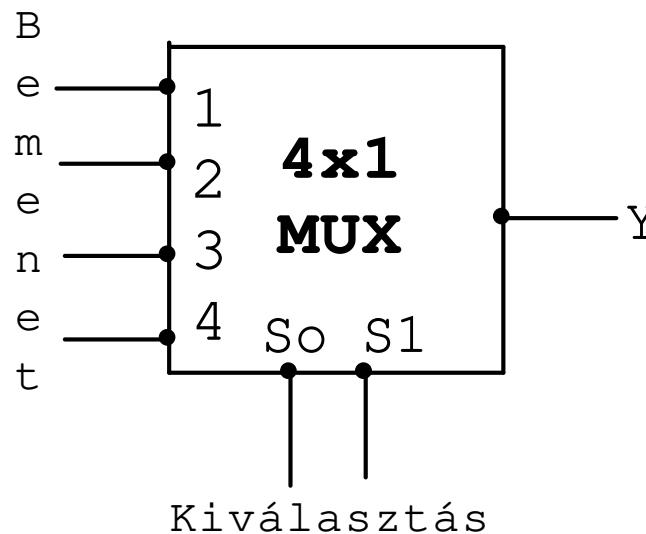
- Másik alap építőelem az **adatbusz** vagy **adatút** (**datapath**). Fontos paraméter a szélessége: egy  $n$  természetes szám.
- Az adatutak pont-pont összeköttetéseket jelentenek különböző méretű és sebességű eszközök között.
- A közvetlen kapcsolat nagy sebességet, de egyben rugalmatlanságot is jelent a bővíthetőségben.
- Ezek az adatutak adatbuszokká szervezhetők, amivel különböző jelvezetékek információi foghatók össze.

# További digitális építőelemek

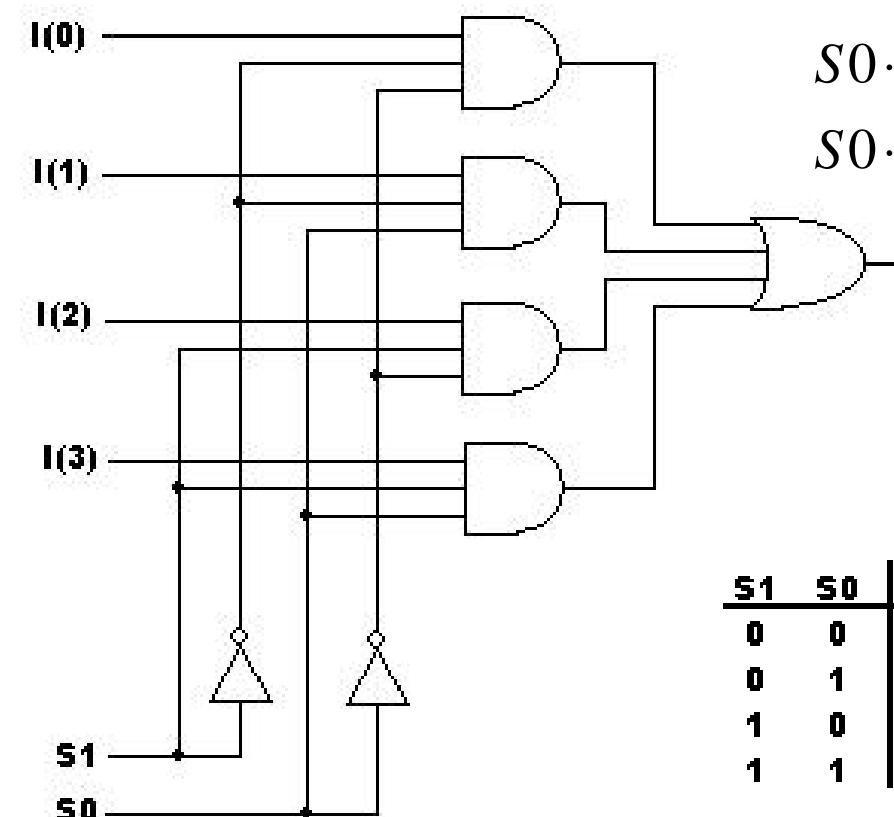
- a.) MUX
- b.) DEMUX
- c.) Dekódoló (decoder)
- d.) Kódoló (encoder)
- e.) Komparátor (Comparator)

# a.) Multiplexer (MUX) – útvonal választó

- N kiválasztó jel,  $2^N$  bemenet → 1 kimenet
- Példa: 4:1 MUX (74LS153)



$2^N$  számú bemenet közül választ egyet ( $Y$ ), mint egy kapcsoló.  
Rendelkezhet EN bemenettel is.



$$Y = \overline{S_0} \cdot \overline{S_1} \cdot I_0 +$$

$$\overline{S_0} \cdot S_1 \cdot I_1 +$$

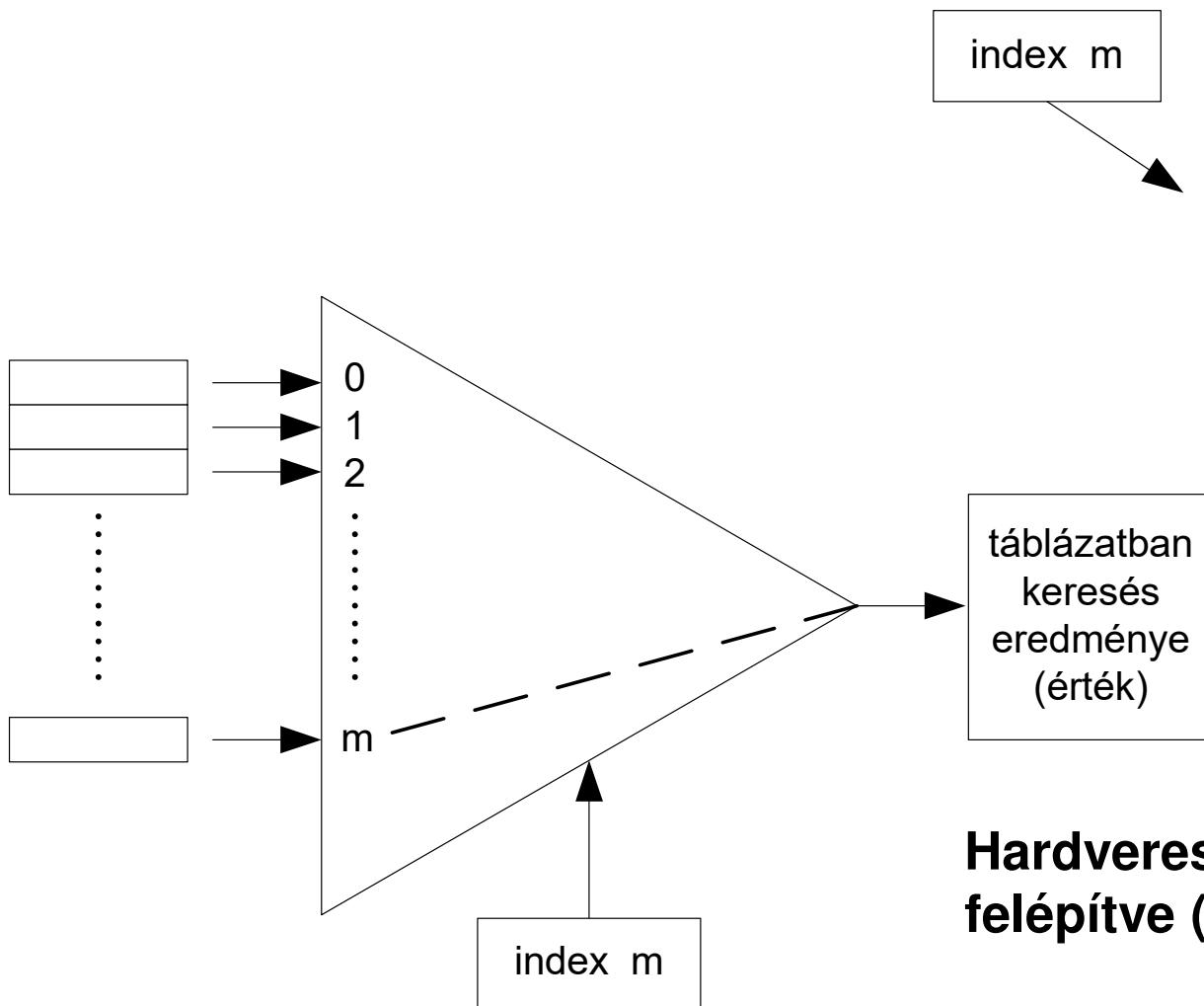
$$S_0 \cdot \overline{S_1} \cdot I_2 +$$

$$S_0 \cdot S_1 \cdot I_3$$

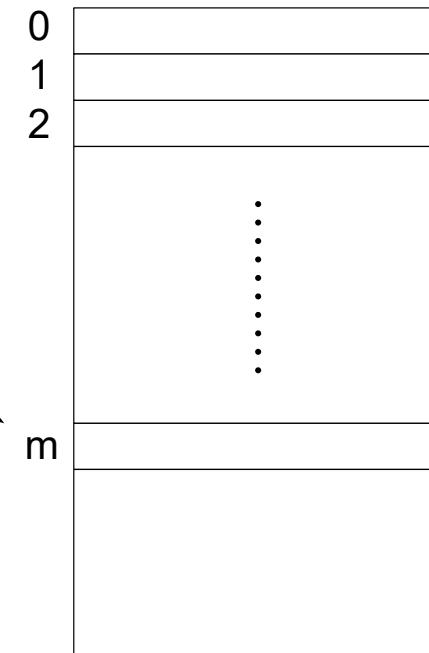
$S_1$	$S_0$	$Y$
0	0	$I(0)$
0	1	$I(1)$
1	0	$I(2)$
1	1	$I(3)$

# Pi. LUT megvalósítások:

## Szoftveres Look-up-Table



$n(w)=1$ -bites  
bejegyzések a  
memoriában



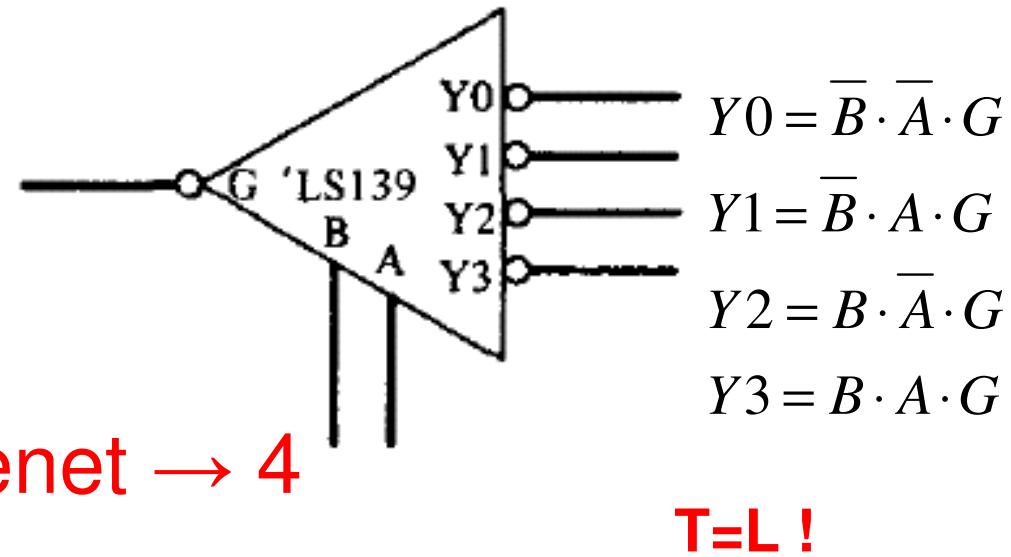
**Hardveres Look-up-Table, MUX-ból  
felépítve (1 bites táblázatkeresés)**

## b.) Példa - 1:4 Demultiplexer

- TTL 74'LS139 dual 1:4 demultiplexer

- G: egy bemenetű
- A,B: selector jelek
- N kiválasztó jel, 1 bemenet  $\rightarrow$  4 kimenet valamelyikére „továbbítja”

- 4-kimenet mindegyike False=1, egyet kivéve, amelyik a kiválasztott (annak az értéke a bemenettől függően lehet T/F)

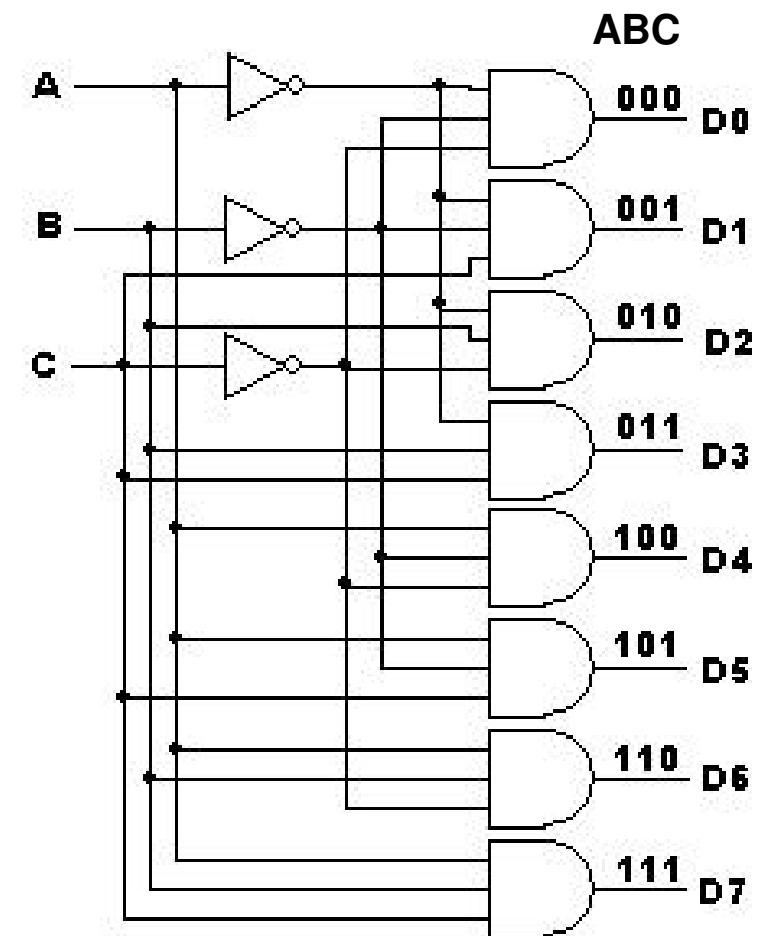


Demultiplexer logika							
G	B	A	Y0	Y1	Y2	Y3	
0	x	x	0	0	0	0	0
1	0	0	1	0	0	0	0
1	0	1	0	1	0	0	0
1	1	0	0	0	1	0	0
1	1	1	0	0	0	0	1

# c.) Dekódoló áramkör

- N bemenet esetén  $\rightarrow 2^N$  dekódolt kimenete van
- N bemenetből mindenkor csak egy aktív logikai értékű
- Példa:  $3 \rightarrow 8$  dekóder áramkör (SN 74LS138N)
  - Példa: Hamming-kódú hibajavító áramkör

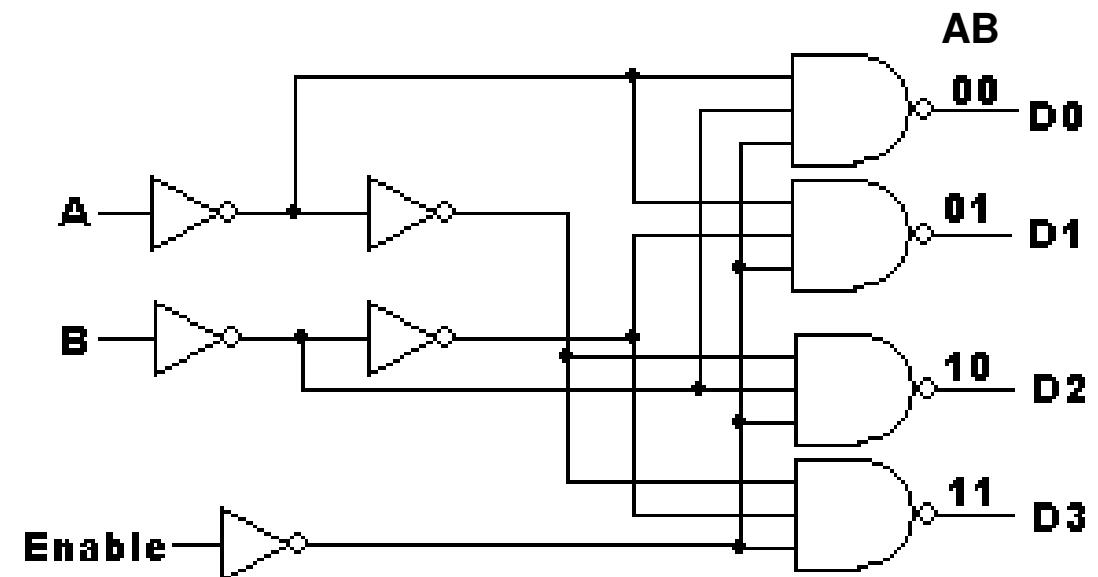
T=H



SELECT INPUTS			OUTPUTS							
C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
L	L	L	L	H	H	H	H	H	H	H
L	L	H	H	L	H	H	H	H	H	H
L	H	L	H	H	L	H	H	H	H	H
L	H	H	H	H	H	L	H	H	H	H
H	L	L	H	H	H	H	L	H	H	H
H	L	H	H	H	H	H	H	L	H	H
H	H	L	H	H	H	H	H	H	L	H
H	H	H	H	H	H	H	H	H	H	L

# Példa: $2 \rightarrow 4$ Dekódoló áramkör engedélyező bemenettel

- SN74HC139
- EN: alacsony aktív állapotban működik
- 2 bemenő jel (A,B)
- 4 kimenet (D0...D3)



En	A	B	D0	D1	D2	D3
1	x	x	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

## d.) Kódoló (encoder) áramkör

- A dekódoló áramkör ellentéte: bemenetek kódolt ábrázolásának egy formája
  - **2<sup>N</sup> bemenet esetén → N kódolt kimenete van**
    - Pl: 8→3 kódoló (pl. 74LS148)
  - **Hagyományos encoder:** csak egy bemenete lehet aktív logikai értékű egyszerre
  - **Priority encoder:** több bemenete is lehet aktív logikai értékű egyszerre, de azok közül ált. a legnagyobb bináris értékű, azaz prioritású bemenethez generál kódot!  
(pl. kód: address, index is lehet)
    - I/O, IRQ jelek, címek generálásánál használják leggyakrabban

## e.) Komparátor

- Logikai kifejezés – *referencia* kifejezés (bináris számok) *aritmetikai kapcsolatának* megállapítására szolgáló eszköz.
  - Pl: Kettő n-bites szám összehasonlítása
- **compare = összehasonlítás!** Az azonosság eldöntéséhez a EQ/XNOR/Coincidence operátort használjuk. Jele:  $A.EQ.B = A \odot B$
- n-bites minták esetén:

$$A.EQ.B = (A_0 \odot B_0) \cdot (A_1 \odot B_1) \cdot \dots \cdot (A_n \odot B_n)$$

# Ismétlés: EQ/XNOR/Coincidence operátor

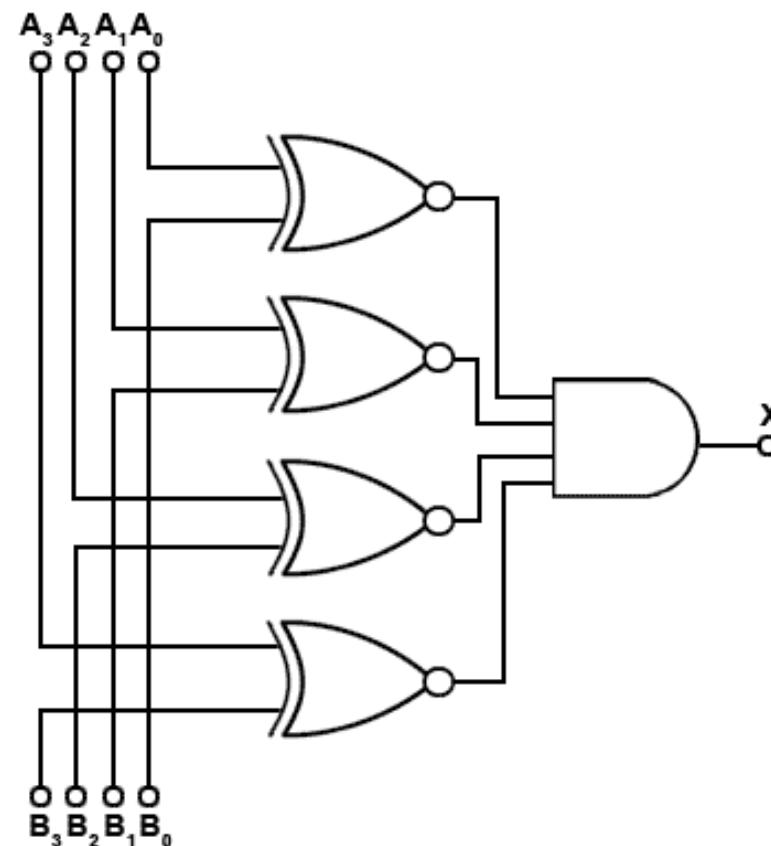
- Logikai egyenlet:  $A.EQ.B = A \odot B = A \cdot B + \overline{A} \cdot \overline{B}$
- Referenciabit szerinti megkülönböztetés:
  - ha a referencia bit (B), amihez hasonlítunk **konstans**
  - ha a referencia bit (B) egy **változó** mennyiség
- Példa: ha B referencia konstans  $\rightarrow$  egyszerűsítése A-nak
$$A.EQ.B = A \text{ if } B = T$$
$$A.EQ.B = \overline{A} \text{ if } B = F$$
- Példa: legyen B egy 4-bites konstans mennyiség ( $B=TFFT$ ), és A tetszőleges, akkor:

$$A.EQ.B = A_0 \cdot \overline{A_1} \cdot \overline{A_2} \cdot A_3$$

# Példa: n=4-bites komparátor

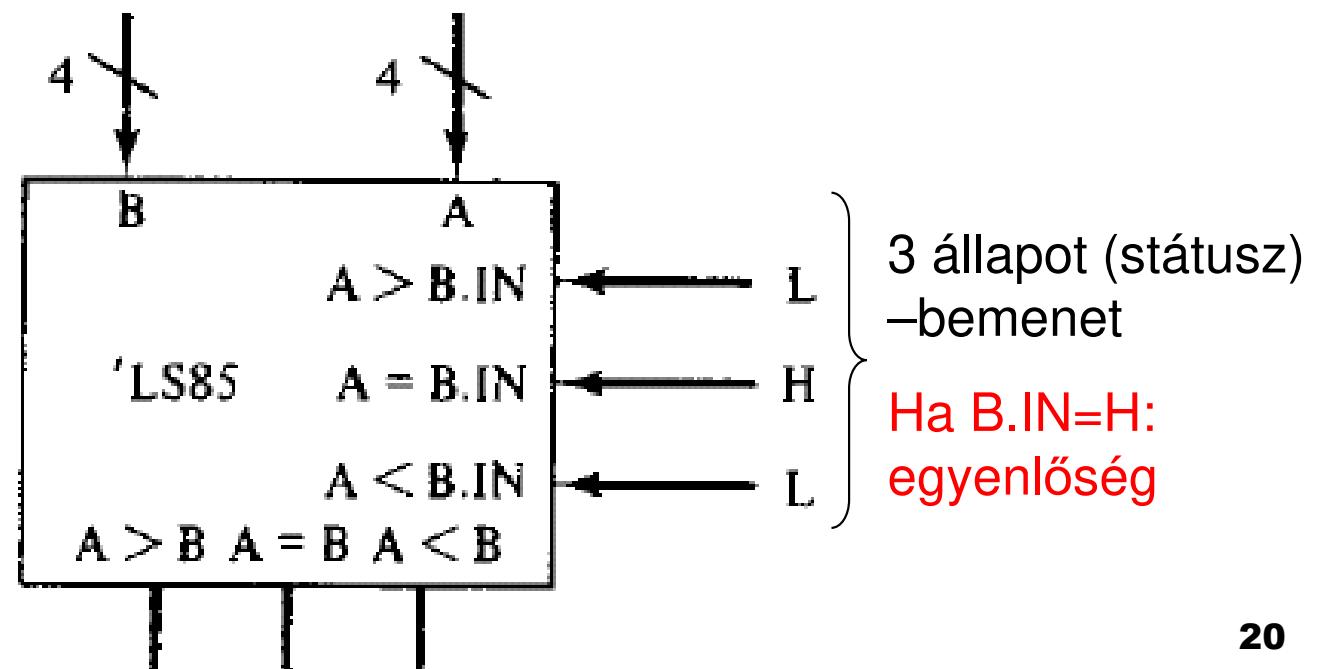
- Mixed-logic kapcsolási rajza, és log.egyenlete:

$$A.EQ.B = (A_0 \odot B_0) \cdot (A_1 \odot B_1) \cdot \dots \cdot (A_n \odot B_n)$$



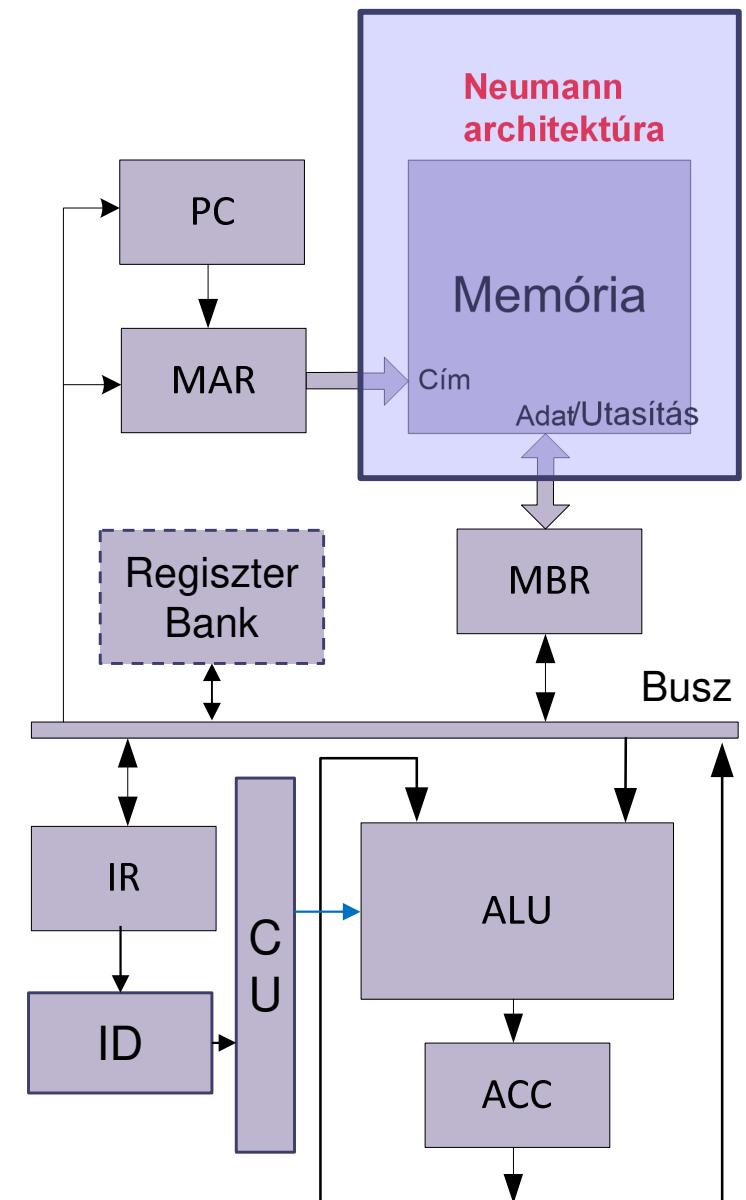
# SN74LS85 4-bit Magnitude Comparator

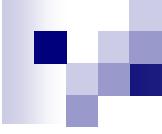
- Magnitude comparing (~nagyságrend összehasonlító, relációs műveletek):
  - két kifejezés **nagyságának** összehasonlítása ( $A < B$ ;  $A = B$ ;  $A > B$  stb.) egyszerre



# Egyszerű (1-című) számítógép felépítése

- **MAR: Memory Address Register** (Memória-cím Regiszter): adott memóriacímen lévő adat/utasítás helyét azonosítja.
- **MBR: Memory Buffer Register** (Memória Puffer Regiszter): tárolja a memóriába beírt, ill. kiolvasott információt. Mivel az adat kiolvasásakor akár törlődhet (destruktív memória).
- **PC: Program Counter** (Programszámláló): a soron következő (végrehajtandó) utasítás helyét azonosítja. Ha egy utasítást tárolnak memóriaterületenként, az utasítás végrehajtása után a PC értékét 1-el kell növelni (increment), mint egy számlálót.
- **IR: Instruction Register** (Utasítás Regiszter): tárolja az éppen végrehajtás alatt álló utasítást. Engedélyezi a gép vezérlő részeinek, hogy a regiszterek, memóriák, ALU egységek vezérlő vonalait (CU) a végrehajtáshoz szükséges működési módba állítsák. Olyan széles, hogy az utasítás műveleti kódja ill. a hozzá tartozó egyéb utasítások ideiglenes másolatai eltárolhatók.
- **ID: Instruction Decoder** (Utasítás dekódoló): az IR-be töltött aktuális utasítás értelmezése, részműveletekre bontása, majd pedig a **CU (Control Unit)** vezérli a teljes rendszert, vezérlő jelek generálása.
- **ACC: Accumulator register** (tároló regiszter): eredmény ideiglenes tárolására használjuk, bizonyos utasítás kódok esetén mindenkorban használni kell (pl. 1-című).
- **Regiszter Bank**: általános célú regisztereket tartalmazó tároló tömb, a műveletek gyorsításához (operandusok betöltése innen történhet)





# Utasítások kódolása

# Gépi kódú utasítások

- Binárisan kódolt vezérlő információ, amely a processzort valamilyen művelet végrehajtására utasítja.
- A számítógépek gépi utasításainak összességét a számítógép **gépi nyelvénnek** = **utasításkészletének** nevezzük (**ISA** = Instruction Set Architecture).
- A gépi nyelv általában kötött (fix, nem bővíthető).
- Egy gépi kódú program csak akkor futtatható egy adott számítógépen, ha a gép tudja értelmezni (interpreter) a gépi kódú a programot ↔ illető gép "nyelvén" írták.

# Gépi kódú utasítások

Két részből állnak :

- 1. rész: műveleti kód (operátor)
- 2. rész: operandus(ok)

Műveleti kód részei:

- 1. operandus címe (helye)
  - 2. operandus címe (helye)
  - végeredmény
- következő utasítás tárcíme
  - műveleti kód
  - 1. operandus címe
  - 2. operandus címe
- 
- 1-, 2-, 3-című utasításkód
- 4-, vagy töbccímű utasításkód

# Utasítás végrehajtás

M. J. Flynn taxonómia (1972) – rendszertan

## ■ Soros/szekvenciális

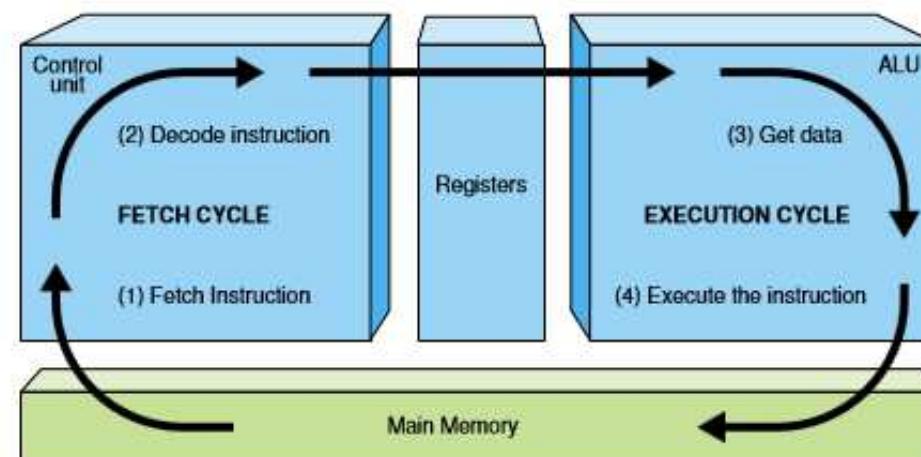
- SISD** = Single Instruction – Single Data (lásd Neumann elvek sorrendi végrehajtása)
- SIMD** = Single Instruction – Multiple Data (vektor-, mátrix műveletek)

## ■ Párhuzamos/parallel

- Több feldolgozó egység megléte szükséges (több processzor vs. több mag vs. több szál )
- MISD** = Multiple Instruction – Single Data
- MIMD** = Multiple Instruction – Multiple Data (pl. GPU-k, mai modern többmagos CPU-k, MPU-k, szuperszámítógépek)

# FDE mechanizmus

- Egy utasítás végrehajtásának 3 fő lépését a **Fetch-Decide-Execute** (FDE) mechanizmussal definiálhatjuk:
  - **F-Fetch:** az utasítás betöltődik a memóriából az utasításregiszterbe (RTL művelet)
  - **D-Decide:** utasítás dekódolása (értelmezése), azonosítása
  - **E-Execute:** a dekódolt utasítást végrehajtjuk az adatokon, aminek eredménye visszakerül a memóriába
    - Mai modern CPU architektúrák 15-30!! fázissal rendelkeznek (FDE-t további al-fázisokra osztva)
- További lépések lehetnek még (tipikusan) „5-lépcsős”:
  - **MEM: Memory operations:** következő utasítás letöltése a memóriából
  - **WB: Memory Write-Back:** eredmény visszaírása



# Gépi kódú utasítások formái

## ■ 0-című (zéró-című) utasítás

Műveleti  
kód

(PI: Verem/Stack/Push-Down-Automata)

## ■ 1-című utasítás

Műveleti  
kód

Operandus  
címe (eredmény is)

ACC: akkumulátor bevezetése

## ■ 2-című utasítás

Műveleti  
kód

1. Operandus  
címe

2. Operandus  
címe (eredmény is)

## ■ 3-című utasítás

Műveleti  
kód

1. Operandus  
címe

2. Operandus  
címe

Eredmény  
címe

PC: Program  
számláló  
bevezetése

## ■ 4-című utasítás

Műveleti  
kód

1. Operandus  
címe

2. Operandus  
címe

Eredmény  
címe

Következő  
ut. címe...

# Assembly

- **(ASM) Alacsony-szintű gépi kódú utasítások nyelve**
- minden utasítás (**mnemonic**) a processzor egy utasításának felel meg! Rövid tömör kód.
- Régen ebben programozták a CPU-kat.
- Manapság: ált. magas-szintű nyelvek alkalmazása (C, C++)
  - „**Szemantikai rés**”: magas-, vs. alacsony-szintek közötti leképezést a fordítóprogram (compiler) biztosítja!

$$Z = X + Y$$

Megfelel:  $\text{ADD}_3 \quad R_1, R_2, R_3$   
(3-című, R=Regiszteres utasítás)

LD	X, \$R <sub>1</sub>	//Load
LD	Y, \$R <sub>2</sub>	
ADD	\$R <sub>1</sub> , \$R <sub>2</sub> , \$R <sub>3</sub>	//Store
ST	\$R <sub>3</sub> , Z	

# RTL leírás:

- minden utasítás végrehajtása az **RTL leírás (Regiszter-Transzfer Nyelv)** segítségével írható le. A szükséges adatátvitelket ezzel a nyelvvel specifikáljuk az egyik fő komponenstől a másikig.
- Továbbá megadható az engedélyezett adatátvitelkhöz tartozó **blokkdiagram** (vagy **gráf** leírás) is, az éleken adott irányítással, melyek az adatátvitel pontos irányát jelölik.
- Az RTL leírások specifikálják a műveletek pontos sorrendjét. Az egyes utasításokhoz megadhatók a szükséges végrehajtási idők (pl.  $[ns, ps]$ -ban), amelyek erősen függenek a felhasznált technológia tulajdonságaitól. Ezek összege fogja megadni a teljes tranzakció időszükségletét.

# 1-című utasítás

Műveleti kód	Operandus címe (eredmény is)
--------------	------------------------------

- Tipikusan egyszerűbb CPU-k utasításai (pl. 8-, 16-bites)
- **Egy operandust azonosítunk**, melyek a memóriából olvasunk ki, a másik operandust, ill. az eredményt az CPU speciális regiszteréből (ACC – akkumulátor) olvassuk ki, ill. tároljuk el.
  - Egyetlen operandus címezhető (memória), a másik operandus már eleve a CPU belső (ACC) regiszterében van – „ACC intenzív” használat,
- Címzési mód alapján a közvetett (operandusok címe) és közvetlen adatelérést (érték szerint) is támogatja
  - Közvetett (indirekt) címzés: memória
  - Közvetlen (direkt) címzés: „speciális” tároló → általános célú regiszter
  - Regiszter utasítások gyorsak → HW felépítés + beírás/kiolvasás a memóriából nem kell.

$$\text{ADD}_1 \boxed{X} = \text{ACC} + X \Rightarrow \text{ACC} \Rightarrow X$$

# Példa: Egy-című utasítás (RTL kód)

## 2's komplementek képzés ACC-vel

Fetch: (regiszterek feltöltése, utasításhívások):

1. **PC→MAR**
2. **M[MAR]→MBR**

PC-ből a következő utasítás címe a MAR-ba töltődik  
Memóriában lévő utasítás kiolvasása az MBR-be

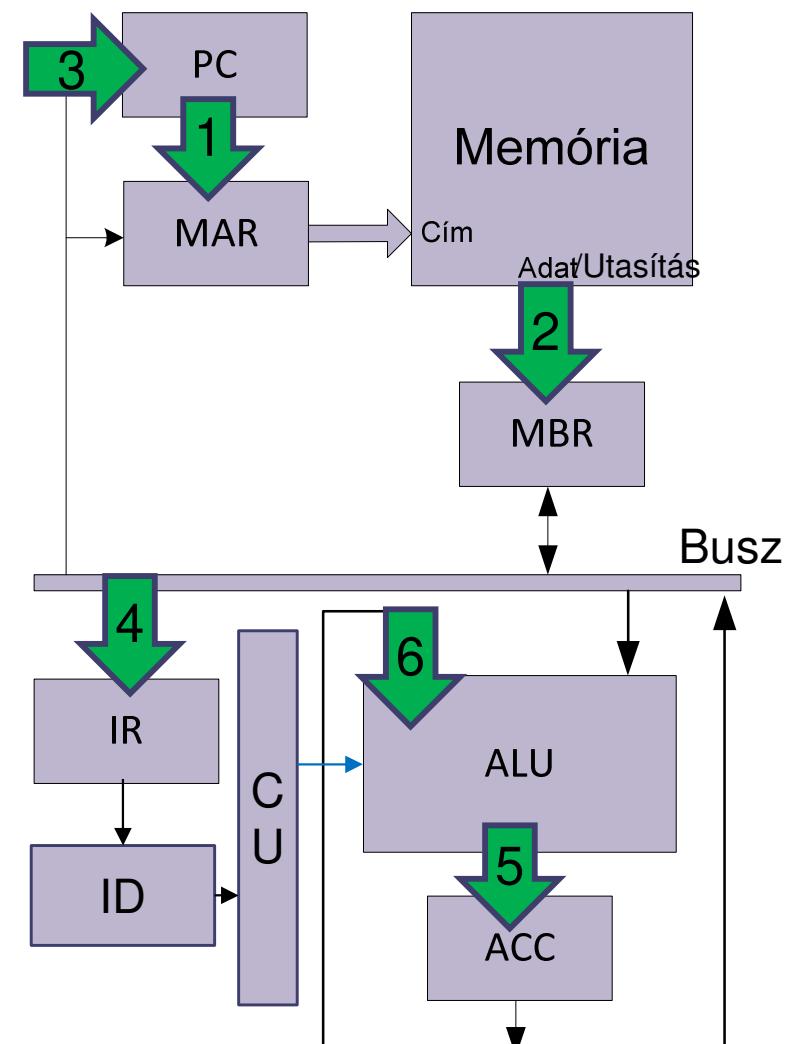
3. **PC+I\_len→PC**  
Az utasítás hosszával (*I\_len*) növeli a PC értékét
4. **MBR→IR**  
Majd az MBR-ben lévő adatot az IR-be tesszük

Decode: (~min 1 órajel ciklus idő)

Execute: (végrehajtás)

5. **ACC →ACC**  
ACC 1's komplementét az ACC-be töltjük
6. **ACC+1→ACC**  
majd ACC-t '1'-el inkrementáljuk (eredmény)

Példa: DEC PDP-8  
számítógépe



# Példa: 1-című gép (Kivonás $SUB_1 X$ )

„X” Operandus címét is a PC-vel azonosítjuk!

*Fetch: (regiszterek feltöltése, utasításhívások):*

**PC→MAR**

Elsőként a PC-ből a következő utasítás címe a MAR-ba töltődik

**M[MAR]→MBR**

Memóriában lévő utasítás beírása az MBR-be (később visszaírjuk)

**PC+I\_len→PC**  
értékét

Az utasítás hosszával ( $I\_len$ ) növeli a PC

**MBR→IR**

Majd az MBR-ben lévő adatot az IR-be tesszük

*Decode: (~min 1 órajel ciklus idő)*

*Execute: (végrehajtás)*

{ **PC→MAR**  
**PC+X\_len →PC**  
**M[MAR]→MBR**  
**MBR→MAR**  
**M[MAR]→MBR**

PC-vel a következő címre mutatunk  
X operandus címének hosszával növeljük a PC-t  
Ezt címet az MBR-be tesszük  
EZ a cím lesz az X operandus címe  
Címen lévő értéket az MBR-be töltjük

**ACC – MBR→ACC**

ACC-ből kivonjuk az X-et, és ACC-be töltjük

# Példa: 1-című gép (kivonás $SUB_1X$ )

Időszükségletek feltüntetésével!  $T_{MEM}=30\text{ns}$ ,  $T_{ALU}=10\text{ns}$ ,  $T_{REG}=5\text{ns}$

Fetch: (regiszterek feltöltése, utasításhívások):

**PC**→**MAR**

[5ns] Elsőként a PC-ből a következő utasítás címe a MAR-ba töltődik

**M[MAR]**→**MBR**

[30ns] Memóriában lévő utasítás beírása az MBR-be (később visszaírjuk)

**PC+I\_len**→**PC**

[5ns] Az utasítás hosszával (I\_len) növeli a PC értékét

**MBR**→**IR**

[5ns] Majd az MBR-ben lévő adatot az IR-be tesszük

Decode: (~min 1 órajel ciklus idő)

Execute: (végrehajtás)

**PC**→**MAR**

[5ns] PC-vel a következő címre mutatunk

**M[MAR]**→**MBR**

[30ns] Ezt címet az MBR-be tesszük

**MBR**→**MAR**

[5ns] Ez a cím lesz az X operandus címe

**M[MAR]**→**MBR**

[30ns] Címen lévő értéket az MBR-be töltjük

**PC+X\_len**→**PC**

[5ns] X operandus címének hosszával növeljük a PC-t

**ACC – MBR**→**ACC**

[10+5ns] ACC-ből kivonjuk az X-et, és ACC-be töltjük

# 2-című utasítás

Műveleti kód

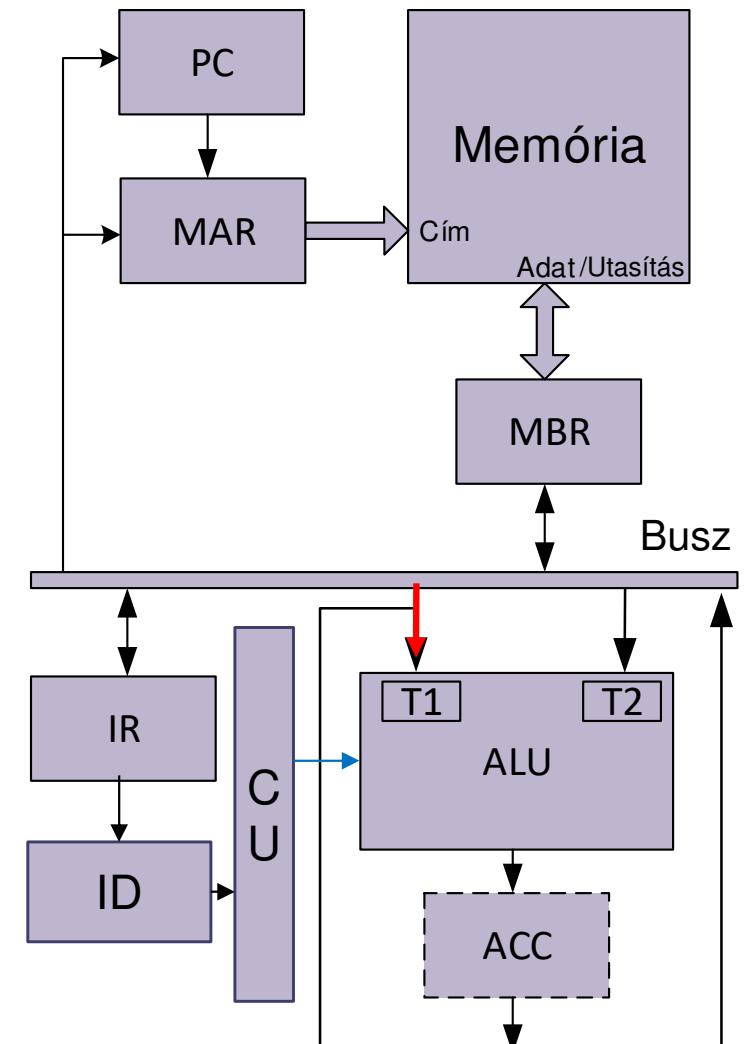
1. Operandus címe

2. Operandus címe (eredmény is)

- A korszerű CPU-k utasításai (32-, 64-bites)
- **Két operandust** azonosítunk, ill. a művelet elvégzése után az eredmény a második operandus helyén tárolódik el (memóriában)!
  - Külön gondoskodni kell a második operandus mentéséről!
- Címzési módok támogatása (lásd: 1-című utasítás)
  - Közvetett (indirekt) vs. Közvetlen módszerek
  - CPU-ban gyorsító tároló elem → általános célú regiszter

$$\text{SUB}_2 \boxed{X}, \boxed{Y} = X - Y \Rightarrow Y$$

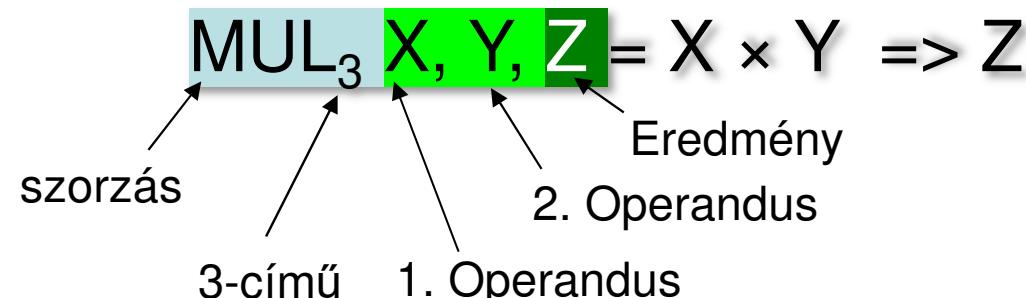
kivonás  
2-című      1. Operandus  
                  2. Operandus (eredmény is)



# 3-című utasítás

Műveleti kód	1. Operandus címe	2. Operandus címe	Eredmény címe
--------------	-------------------	-------------------	---------------

- A korszerű CPU-k utasításai
- **A két operandust és az eredményt külön azonosítjuk!**
- A művelet elvégzése után az eredmény egy új, különálló helyen tárolódik el (memóriában)
  - Az eredmény nem írja felül így már az operandus(okat)!
  - **3-, és többcímű számítógép felépítése azonos a 2-című architektúrával!**
- Közvetett (cím), és közvetlen adatelérést (érték szerint) is támogat
  - Közvetett (indirekt) vs. Közvetlen módszerek (lásd: 1-című utasítás)
  - CPU-ban gyorsító tároló elem → általános célú regiszter



# Jelölés: kettő-, és töbccímű gép

- Jelölés: **ADD<sub>2</sub> X, Y**                                    **két-című** utasítás (*műv, op1, op2*). Az X cím által azonosított helyen tárolt értéket hozzáadjuk az Y cím által azonosított helyen lévő értékhez, és az összeadás eredményét az Y címmel azonosított helyen tároljuk el.
- Jelölés: **ADD<sub>3</sub> X,Y,Z**                                    **három-című** utasítás (*műv, op1, op2, eredmény*): hasonló az előzőhez, csak az összeadás eredménye egy új helyen, a Z cím által azonosított helyen tárolódik el.
- Fontos megjegyezni hogy ebben az esetben („regiszter nélkülisége”) a T1, ill. T2 regiszter nem az utasítás-készlet architektúra része! (Ezért nem keverendő össze a később említésre kerülő *regiszteres* címzéssel!)
  - Ebben az esetben T1, T2-t „csak” az ALU részeként, nem pedig a rendszer gyorsítását szolgáló elkülönített regiszter bankként használjuk.

# Példa: Összeadás két-című géppel ADD<sub>2</sub>(X,Y)

Időszükségletek feltüntetésével!

T<sub>MEM</sub>=30ns, T<sub>ALU</sub>=10ns, T<sub>REG</sub>=5ns

Fetch: (regiszterek feltöltése, utasításhívások):

PC→MAR  
M[MAR]→MBR  
PC+I\_len→PC  
MBR→IR

[5ns] PC-ből a következő utasítás címe a MAR-ba töltődik  
[30ns] Memóriában lévő utasítás beírása az MBR-be  
[5ns] Az utasítás hosszával (I\_len) növeli a PC értékét  
[5ns] Majd az MBR-ben lévő adatot az IR-be tesszük

Decode: (~min 1 órajel ciklus idő)

Execute: (véghajtás)

PC→MAR  
PC+X\_Alen →PC  
M[MAR]→MBR  
MBR→MAR  
M[MAR]→MBR  
MBR→T1  
PC→MAR  
PC+Y\_Alen →PC  
M[MAR]→MBR  
MBR→MAR  
M[MAR]→MBR  
MBR→T2  
T1 + T2→MBR  
MBR→M[MAR]

[5ns] PC-vel a következő (**X**) címre mutatunk  
[5ns] X operandus címének hosszával növeljük a PC-t  
[30ns] Ezt az X címet az MBR-be írjuk  
[5ns] Ez a cím lesz az X operandus címe  
[30ns] X címen lévő értéket az MBR-be töltjük  
[5ns] **X értékét** T1-be töltjük  
  
[5ns] PC-vel a következő (**Y**) címre mutatunk  
[5ns] Y operandus címének hosszával növeljük a PC-t  
[30ns] Ezt a Y címet az MBR-be írjuk  
[5ns] Ez a cím lesz az Y operandus címe  
[30ns] Y Címen lévő értéket az MBR-be töltjük  
[5ns] **Y értékét** T2-be töltjük  
  
[10+5ns] ADD2 művelet elvégzése, MBR-be töltjük  
[30ns] Eredményt a MAR-ban tároljuk el (ahol Y volt)

Direkt  
címzést  
használunk  
itt!

# Példa: Összeadás három-című géppel ADD<sub>3</sub>(X,Y,Z)

## Időszükségletek feltüntetésével!

T<sub>MEM</sub>=30ns, T<sub>ALU</sub>=10ns, T<sub>REG</sub>=5ns

Fetch: (regiszterek feltöltése, utasításhívások):

PC→MAR  
M[MAR]→MBR  
PC+I\_len→PC  
MBR→IR

[5ns] PC-ből a következő utasítás címe a MAR-ba töltődik  
[30ns] Memóriában lévő utasítás beírása az MBR-be  
[5ns] Az utasítás hosszával (I\_len) növeli a PC értékét  
[5ns] Majd az MBR-ben lévő adatot az IR-be tesszük

Decode: (~min 1 órajel ciklus idő)

Execute: (végrehajtás)

PC→MAR  
PC+X\_Alen →PC  
M[MAR]→MBR  
MBR→MAR  
M[MAR]→MBR  
MBR→T1

[5ns] PC-vel a következő (X) címre mutatunk  
[5ns] X operandus címének hosszával növeljük a PC-t  
[30ns] Ezt az X címet az MBR-be írjuk  
[5ns] Ez a cím lesz az X operandus címe  
[30ns] X címen lévő értéket az MBR-be töltjük  
[5ns] X értékét T1-be töltjük

PC→MAR  
PC+Y\_Alen →PC  
M[MAR]→MBR  
MBR→MAR  
M[MAR]→MBR  
MBR→T2

[5ns] PC-vel a következő (Y) címre mutatunk  
[5ns] Y operandus címének hosszával növeljük a PC-t  
[30ns] Ezt a Y címet az MBR-be írjuk  
[5ns] Ez a cím lesz az Y operandus címe  
[30ns] Y Címen lévő értéket az MBR-be töltjük  
[5ns] Y értékét T2-be töltjük

PC→MAR  
PC+Z\_Alen →PC  
M[MAR]→MBR  
MBR→MAR  
T1 + T2→MBR  
MBR→M[MAR]

[5ns] PC-vel a következő (Z) címre mutatunk  
[5ns] Z operandus címének hosszával növeljük a PC-t  
[30ns] a Z eredmény címét az MBR-be írjuk  
[5ns] majd a MAR-ba töltjük  
[10+5ns] ADD2 művelet elvégzése, MBR-be töltjük  
[30ns] Eredményt a memóriában tároljuk el (ahol Z volt)

Direkt  
címzést  
használunk  
itt!

Σ 295ns

# Komplex műveletek: MŰV<sub>2</sub> vs. MŰV<sub>3</sub> összehasonlítása

- A MŰV<sub>3</sub> –nál (ahogy az RTL leírásból is látszott) egy sor végrehajtása több időt vesz igénybe az utasítások F-D-E fázisánál, mint az MŰV<sub>2</sub> esetén, mivel egyel több címre kell hivatkozni.
- Azonban, az MŰV<sub>3</sub> jelentősége a **komplexebb műveletek** elvégzésekor mutatkozik meg → tömörebb forma, **kevesebb** utasítás sor!
- Példa: Legyen  $X = Y * Z + W * V$  ( oldjuk meg MŰV<sub>2</sub>-vel, és MŰV<sub>3</sub>-al)

MŰV <sub>2</sub>	MŰV <sub>3</sub>
MOVE Y to X	MUL <sub>3</sub> Y,Z,T
MUL <sub>2</sub> Z,X	MUL <sub>3</sub> W,V,Y
MOVE W to Y	ADD <sub>3</sub> T,Y,X
MUL <sub>2</sub> V,Y	
ADD <sub>2</sub> Y,X	

# Példa: Kvadratikus egyenlet (2-, és 3-című utasításokkal)

Számítsuk ki 3-című utasításokkal az alábbi egyenletet!

$$X = \frac{(-b + \sqrt{b^2 - 4*a*c})}{(2*a)}$$



```
T1 := b * b
T2 := 4 * a
T3 := T2 * c
T4 := T1 - T3
T5 := sqrt(T4)
T6 := 0 - b
T7 := T5 + T6
T8 := 2 * a
X := T7 / T8
```

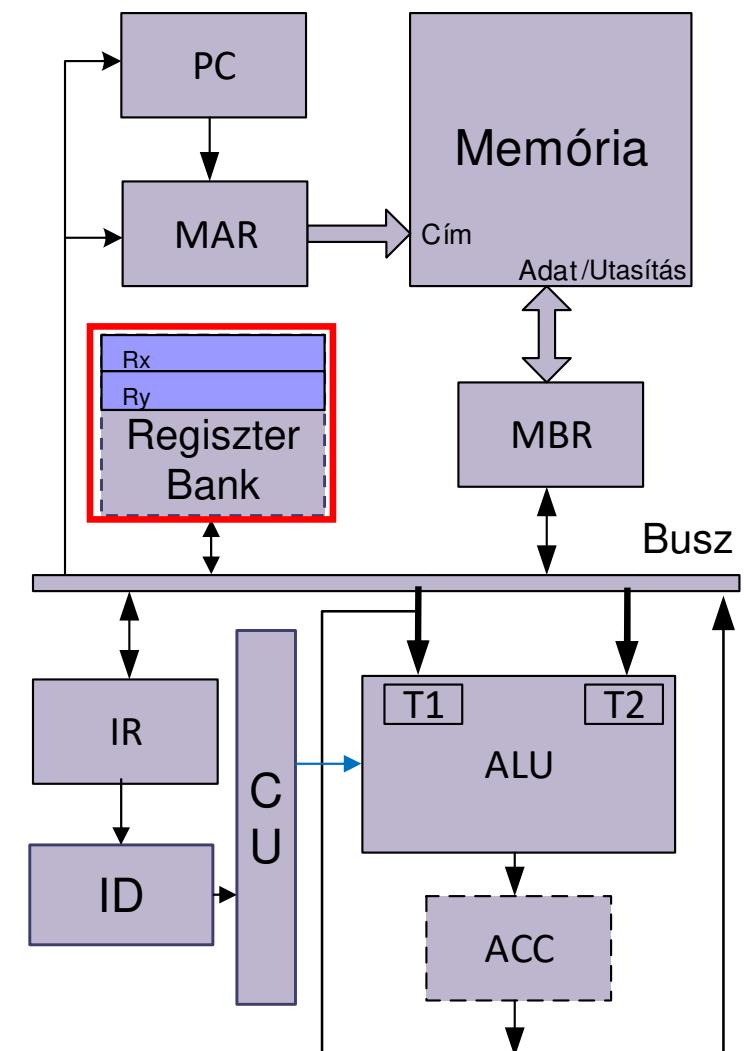


RTL utasítások

<b>MUL<sub>3</sub></b>	b, b, T1
<b>MUL<sub>3</sub></b>	4, a, T2
<b>MUL<sub>3</sub></b>	T2, c, T3
<b>SUB<sub>3</sub></b>	T1, T3, T4
<b>SQRT<sub>2</sub></b>	T4, T5
<b>SUB<sub>3</sub></b>	0, b, T6
<b>ADD<sub>3</sub></b>	T5, T6, T7
<b>MUL<sub>3</sub></b>	2, a, T8
<b>DIV<sub>3</sub></b>	T7, T8, X

## c.) Kettő-, és több-című gépek (Regiszteres változat)

- Egy utasítással több operandust / operátort lehet megadni,
- Kevesebb utasítás-sorral, összetett módon írhatók le az RTL nyelven a folyamatok, (az egycímű gépekkel szemben)
- Az általános célú regiszterek (**Reg. Bank**) használata csökkenti a végrehajtási időt, mivel a lassú memória-intenzív műveletek helyett gyorsabb regiszterműveleteket használnak. (A regiszterbank  $2^N$  számú regisztert tartalmazhat.)



# Példa 1: Összeadás kétcímű géppel ADD<sub>2</sub>(R<sub>X</sub>, R<sub>Y</sub>)

Időszükségletek feltüntetésével!

T<sub>MEM</sub>=30ns, T<sub>ALU</sub>=10ns, T<sub>REG</sub>=5ns

Fetch: (regiszterek feltöltése, utasításhívások):

PC→MAR	[5ns] PC-ből a következő utasítás címe a MAR-ba töltődik
M[MAR]→MBR	[30ns] Memóriában lévő utasítás beírása az MBR-be
PC+I_len→PC	[5ns] Az utasítás hosszával (I_len) növeli a PC értékét
MBR→IR	[5ns] Majd az MBR-ben lévő adatot az IR-be tesszük

Decode: (~min 1 órajel ciklus idő)

Execute: (végrehajtás)

RX→T1	[5ns] RX értékét T1-be töltjük
RY→T2	[5ns] RY értékét T2-be töltjük
T1 + T2→RY	[10+5ns] ADD2 művelet elvégzése, RY-ba töltjük

$\Sigma$  70ns

Regiszteres,  
direkt-címzést  
használunk  
itt!

# Példa 2: Összeadás kétcímű géppel ADD<sub>3</sub>(R<sub>X</sub>, R<sub>Y</sub>, R<sub>Z</sub>)

Időszükségletek feltüntetésével!

T<sub>MEM</sub>=30ns, T<sub>ALU</sub>=10ns, T<sub>REG</sub>=5ns

Fetch: (regiszterek feltöltése, utasításhívások):

PC→MAR	[5ns] PC-ből a következő utasítás címe a MAR-ba töltődik
M[MAR]→MBR	[30ns] Memóriában lévő utasítás beírása az MBR-be
PC+I_len→PC	[5ns] Az utasítás hosszával (I_len) növeli a PC értékét
MBR→IR	[5ns] Majd az MBR-ben lévő adatot az IR-be tesszük

Decode: (~min 1 órajel ciklus idő)

Execute: (végrehajtás)

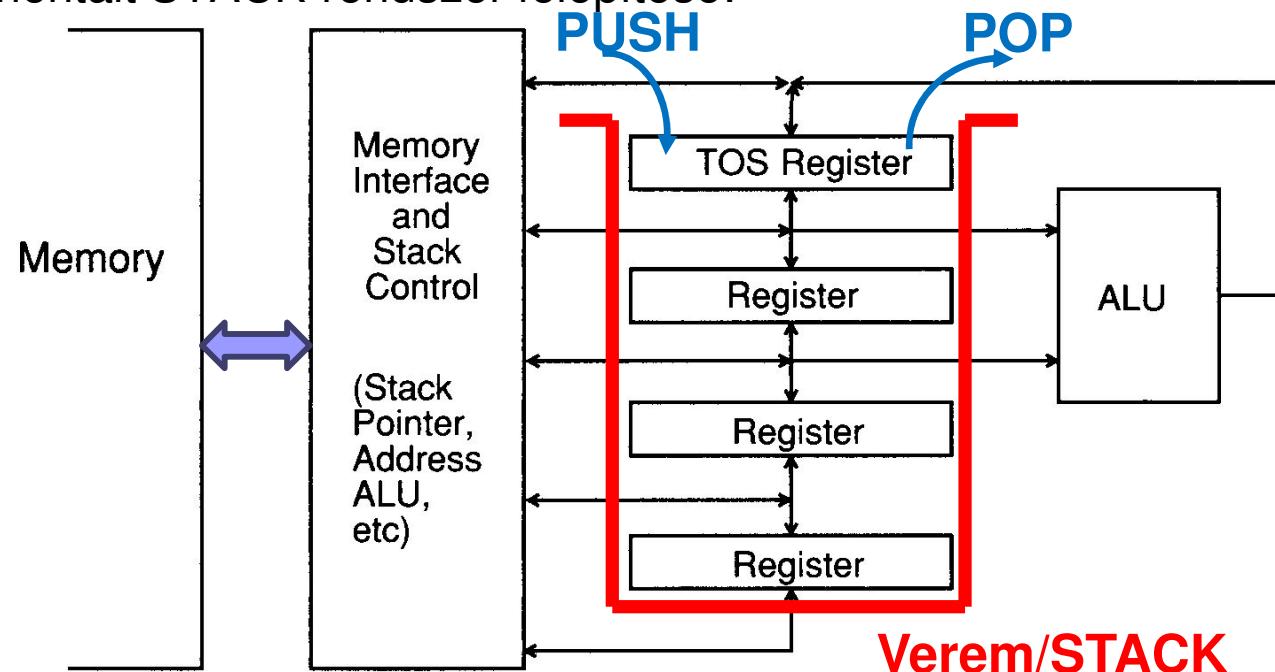
RX→T1	[5ns] RX értékét T1-be töltjük
RY→T2	[5ns] RY értékét T2-be töltjük
T1 + T2→RZ	[10+5ns] ADD2 művelet elvégzése, RZ-be töltjük

Σ 70ns

Regiszteres,  
direkt-címzést  
használunk  
itt!

# d.) Zéró-, vagy 0-című gép

- Zéró-című utasítás végrehajtásához **vermet (STACK)** használunk
- **Verem:** „**LIFO**”-típusú tároló (**Last-In, First Out**), amelyből az utoljára betett adatot vesszük ki elsőként.
- STACK alkalmazása: MŰV<sub>0</sub> () **PUSH, POP** → **Műveleti kód**
  - Függvény hívás, és visszatérés (CALL-RETURN eljárások)
  - Aritmetikai műveletek is hatékonyan végrehajthatók használatával:
    - a szükséges operandusokat a STACK egy-egy rekeszében tároljuk: minden a felső (kettő) regiszterből vesszük ki, és elvégezzük rajtuk a műveletet, majd az eredményt a verem tetejére tesszük vissza. **Azért nevezik zéró címűnek, mivel az operandusok azonosítására szolgáló utasításhoz nem használ címeit.**
  - A HW-orientált STACK-rendszer felépítése:



# Példa: Zéró-, vagy 0-című gép

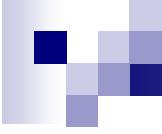
Legyen  $F = A + [ B^*C + D^*(E / F) ]$  aritmetikai kifejezés. F-et akarjuk kiszámolni verem segítségével, és eltárolni az eredményt. A következő műveletek szükségesek: **PUSH, POP, ill. ADD, DIV, MULT**

**Fontos:** minden elvégzett művelet egy szinttel csökkenti a verem mélységét!

1. **módszer** kiértékelésénél az aritmetikai kifejezés elejétől haladunk, és amint lehetséges a verem tetején lévő két értéken végrehajtjuk a soron következő műveletet, az eredményt, pedig a verem tetejére pakoljuk. A veremben max. 5 értéket tárolunk el, (**mélysége 5 lesz**) ezért lassabb, mint a második módszer.

2. **módszernél** az aritmetikai kifejezést hátulról előrefelé haladva értékeljük ki. Itt is elvégezzük a soron következő műveletet, és az eredményt a verem tetejére rakjuk. De ez gyorsabb módszer, mivel a veremben max. csak 3 értéket tárolunk el (**mélysége 3**).

1. módszer: (arit. kif. elejétől haladva)	2. módszer: (arit. kif. végétől haladva)
PUSH A	PUSH E
PUSH B	PUSH F
PUSH C	DIV [E/F]
MULT [B*C]	PUSH D
PUSH D	MULT [D*(E/F)]
PUSH E	PUSH C
PUSH F	PUSH B
DIV [E/F]	MULT [B*C]
MULT [D*(E/F)]	ADD [B*C+D*(E/F)]
ADD [B*C+D*(E/F)]	PUSH A
ADD [A+(B*C+D*(E/F))]	ADD [A+(B*C+D*(E/F))]
POP F	POP F



# Operandus „címzési módok”

# Gépi kódú utasítások (ism)

## ■ 0-című (zéró-című) utasítás

Műveleti  
kód

(PI: Verem/Stack/Push-Down-Autómata)

## ■ 1-című utasítás

Műveleti  
kód      Operandus  
             címe (eredmény is)

ACC: akkumulátor bevezetése

## ■ 2-című utasítás

Műveleti  
kód      1. Operandus  
             címe      2. Operandus  
             címe (eredmény is)

## ■ 3-című utasítás

Műveleti  
kód      1. Operandus  
             címe      2. Operandus  
             címe      Eredmény  
             címe

PC: Program  
számláló  
bevezetése

## ■ 4-című utasítás

Műveleti  
kód      1. Operandus  
             címe      2. Operandus  
             címe      Eredmény  
             címe      Következő  
             ut. címe...

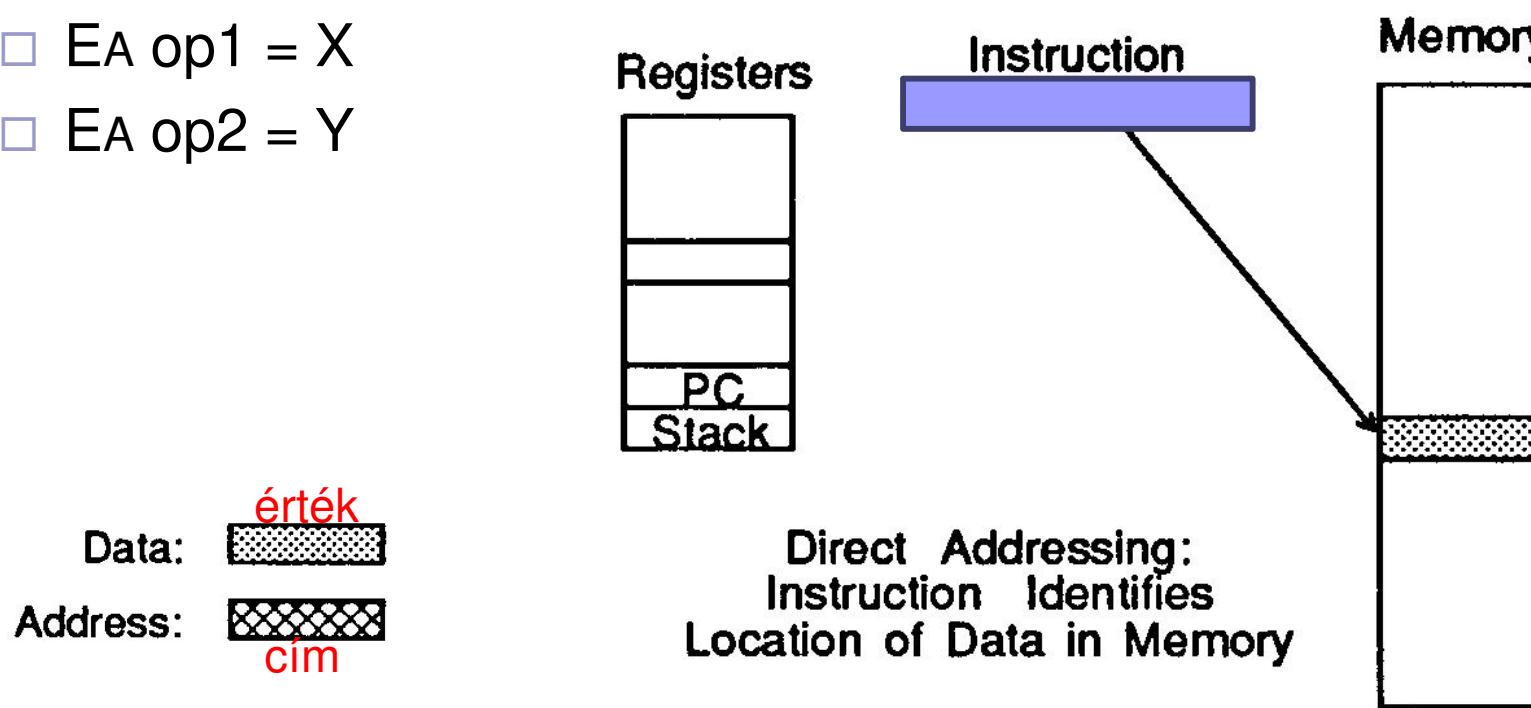
# Operandus címzési módok

- Utasítás kód része: hogyan érjük el az operandust!
- Utasítás végrehajtásakor a kívánt operandust is azonosítjuk, címével hivatkozhatunk a pontos helyére (címre, értékre).
- Többféle címzési mód létezik:
  - közvetlen (directed),
  - közvetett (indirected),
  - indexelt (indexed),
  - regiszteres megvalósítású (register relative).
- Ezek kombinációja igen sokféle lehet: összesen akár 10-féle azonosítási módot tesz lehetővé.
- Jelölés: **E<sub>A</sub>= Effektív (valódi) címe** az operandusnak

# 1. Direkt címzés (X)

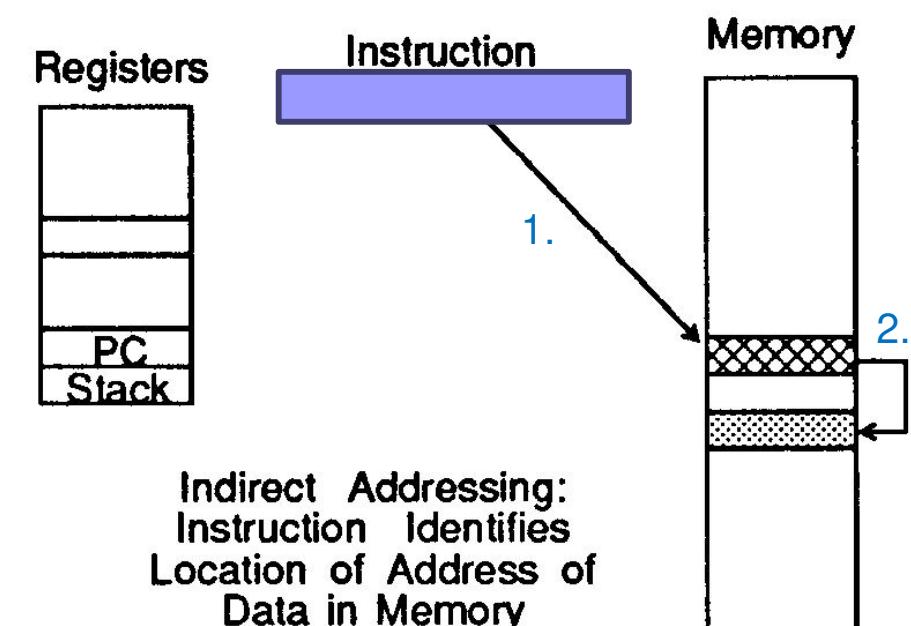
- Az utasítás egyértelműen, közvetlenül azonosítja az operandus helyét a memóriában. (Effektív cím,  $E_A =$  valódi címén tárolt érték) Jel:  $E_A = A$ .
- **Jel:** **ADD<sub>2</sub> X,Y** (X-ben tárolt op1 értéket hozzáadjuk az Y-ban tárolt op2 értékhez, az eredmény az Y-ban lesz.)

- $E_A \text{ op1} = X$
- $E_A \text{ op2} = Y$



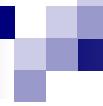
## 2. Indirekt címzés (\*X)

- Az utasítás **közvetett** módon, (nem közvetlenül az operandus értékére), hanem az operandus **helyére** mutat egy **cím** segítségével a memóriában. Ez a cím a helyet azonosítja. Ez sokkal hatékonyabb megvalósítás.
- Jel: **ADD<sub>2</sub> \*X,\*Y** (\*: **indirekció, pointer**)
  - $E_A \text{ op1} = \text{MEM}[X]$
  - $E_A \text{ op2} = \text{MEM}[Y]$
- Ezt különböző gyártók többféleképpen jelölik. Általában az indirekt címzési módot (\*)-al jelölik:
- Példa: **ADD<sub>2</sub> \*X,\*Y** (az első op1 értékének címe az X-ben található, a második op2 értékének címe az Y-ban lesz, és az eredmény is az Y-ban tárolódik el.) Az 1.), 2.), 3.), 4.), közül ez a *leglassabb* megvalósítás, de az indirekt címzés a *memóriatömb* elemeinek elérhetőségét biztosítja!



Data:

Address:



# Példa: Összeadás két-című géppel ADD<sub>2</sub>(\*X,\*Y)

**Időszükségletek feltüntetésével!**

T<sub>MEM</sub>=30ns, T<sub>ALU</sub>=10ns, T<sub>REG</sub>=5ns

Fetch: (regiszterek feltöltése, utasításhívások):

PC→MAR  
M[MAR]→MBR  
PC+I\_len→PC  
MBR→IR

[5ns] PC-ből a következő utasítás címe a MAR-ba töltődik  
[30ns] Memóriában lévő utasítás beírása az MBR-be  
[5ns] Az utasítás hosszával (I\_len) növeli a PC értékét  
[5ns] Majd az MBR-ben lévő adatot az IR-be tesszük

Decode: (~min 1 órajel ciklus idő)

Execute: (végrehajtás)

PC→MAR  
PC+X\_Alen →PC  
M[MAR]→MBR  
MBR→MAR  
M[MAR]→MBR  
MBR→MAR  
M[MAR]→MBR  
MBR→T1

[5ns] PC-vel a következő (X) címének címére mutatunk  
[5ns] X operandus címének hosszával növeljük a PC-t  
[30ns] X címének címét az MBR-be írjuk  
[5ns] Ezt a címet a MAR-ba töltjük  
[30ns] X címét megkapjuk az MBR-ben  
[5ns] X címét a MAR-ba töltjük  
[30ns] X címén lévő értékét megkapjuk MBR-ben  
[5ns] **X értékét** T1-be töltjük

PC→MAR  
PC+Y\_Alen →PC  
M[MAR]→MBR  
MBR→MAR  
M[MAR]→MBR  
MBR→MAR  
M[MAR]→MBR  
MBR→T2  
T1 + T2→MBR  
MBR→M[MAR]

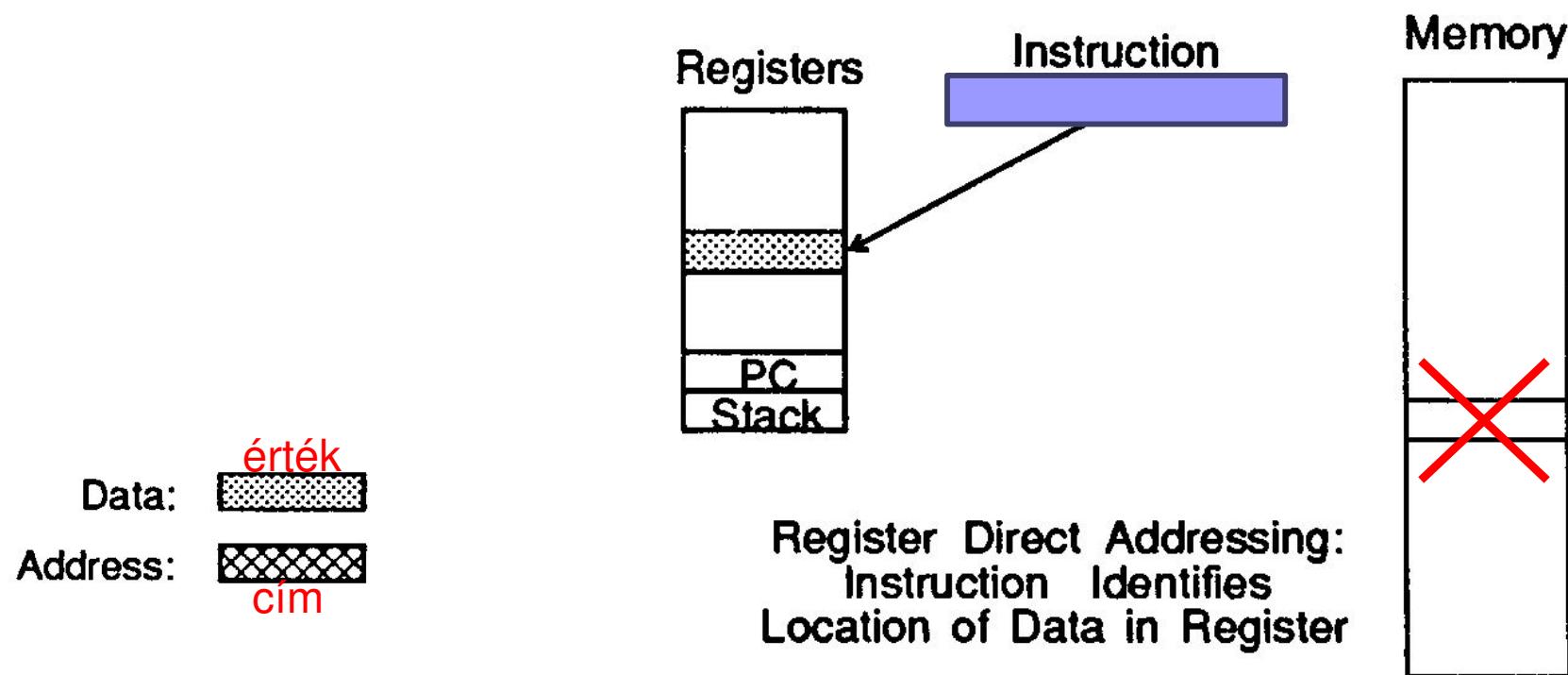
[5ns] PC-vel a következő (Y) címének címére mutatunk  
[5ns] Y operandus címének hosszával növeljük a PC-t  
[30ns] Y címének címét az MBR-be írjuk  
[5ns] Ezt a címet a MAR-ba töltjük  
[30ns] Y címét megkapjuk az MBR-ben  
[5ns] Y címét a MAR-ba töltjük  
[30ns] Y címén lévő értékét megkapjuk MBR-ben  
[5ns] **Y értékét** T2-be töltjük  
[10+5ns] ADD2 művelet elvégzése, MBR-be töltjük  
[30ns] Eredményt a memóriában tároljuk el (ahol Y volt)

**Indirekt  
címzést  
használunk  
itt!**

$\Sigma 320\text{ns}$

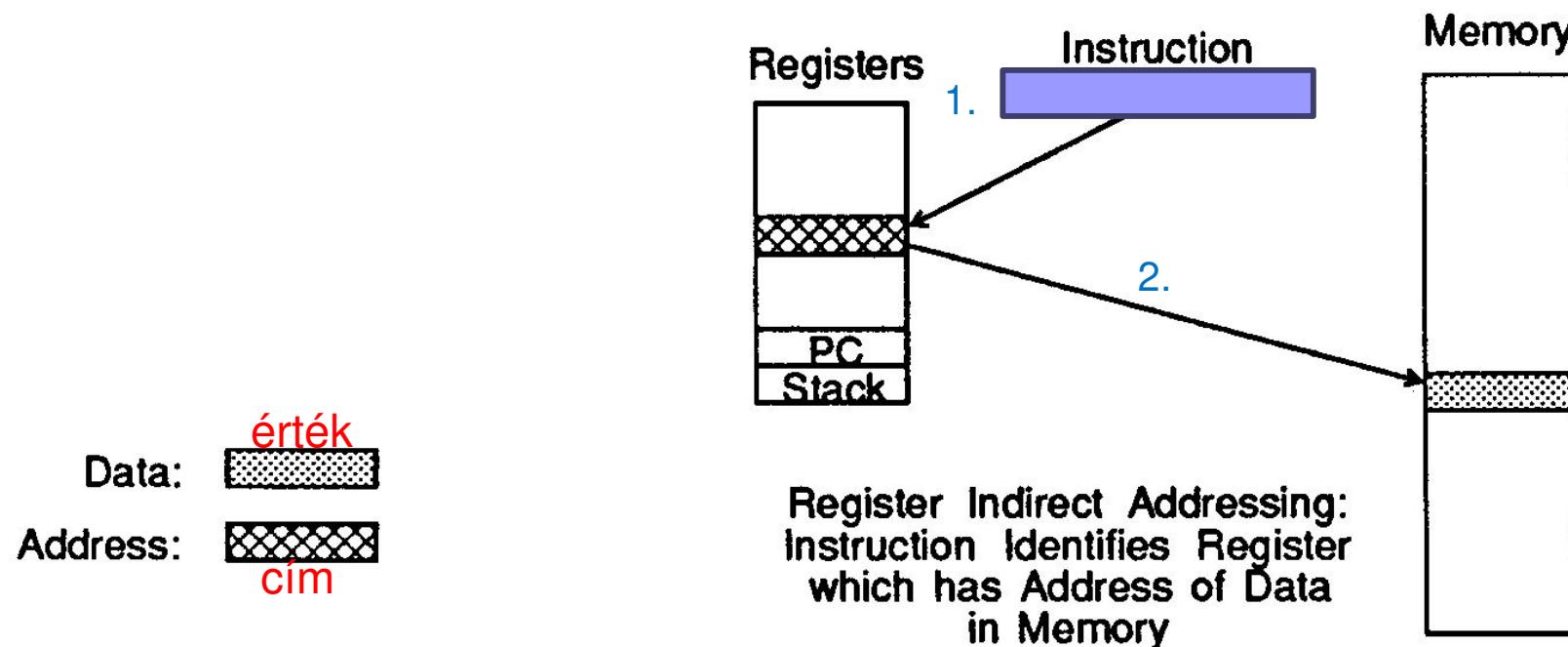
### 3. Regiszteres direkt címzés ( $R_X$ )

- Hasonló, mint a direkt címzés, de sokkal **gyorsabb**, mivel a memória intenzív-műveletek helyett az operandusok értékeit a CPU gyors **belső regisztereiben** tárolja, és csak a számítási eredményt tölti át a memóriába. Az 1), 2), 3), 4) közül ez a *leggyorsabb* módszer.



# 4. Regiszteres indirekt címzés ( $*R_X$ )

- Hasonló, mint az indirekt címzés, de sokkal gyorsabb, mivel a memória-intenzív műveletek helyett a köztes **címeket** a CPU **gyors belső regiszterekben** tárolja, és csak a végén hivatkozik a memoriára (operandus értékére). A 3.) regiszteres módszer után ez a második leggyorsabb.



# Példa: Összeadás kétcímű géppel ADD<sub>2</sub>(\*R<sub>X</sub>, \*R<sub>Y</sub>)

Időszükségletek feltüntetésével!

T<sub>MEM</sub>=30ns, T<sub>ALU</sub>=10ns, T<sub>REG</sub>=5ns

Fetch: (regiszterek feltöltése, utasításhívások):

PC→MAR	[5ns] PC-ből a következő utasítás címe a MAR-ba töltődik
M[MAR]→MBR	[30ns] Memóriában lévő utasítás beírása az MBR-be
PC+I_len→PC	[5ns] Az utasítás hosszával (I_len) növeli a PC értékét
MBR→IR	[5ns] Majd az MBR-ben lévő adatot az IR-be tesszük

Decode: (~min 1 órajel ciklus idő)

Regiszteres  
indirekt-  
címzést  
használunk itt!

Execute: (végrehajtás)

RX→MAR	[5ns] RX címét a MAR-ba töltjük
M[MAR]→MBR	[30ns] Kinyerjük az RX címén lévő értéket, amit MBR-be töltünk
MBR→T1	[5ns] RX értékét T1-be töltjük
RY→MAR	[5ns] RY címét a MAR-ba töltjük
M[MAR]→MBR	[30ns] Kinyerjük az RY címén lévő értéket, amit MBR-be töltünk
MBR→T2	[5ns] RY értékét T2-be töltjük
T1 + T2→MBR	[10+5ns] ADD2 művelet elvégzése, MBR-be töltjük
MBR→M[MAR]	[30ns] eredményt a memóriában RY operandus helyén tároljuk el

Σ 170ns

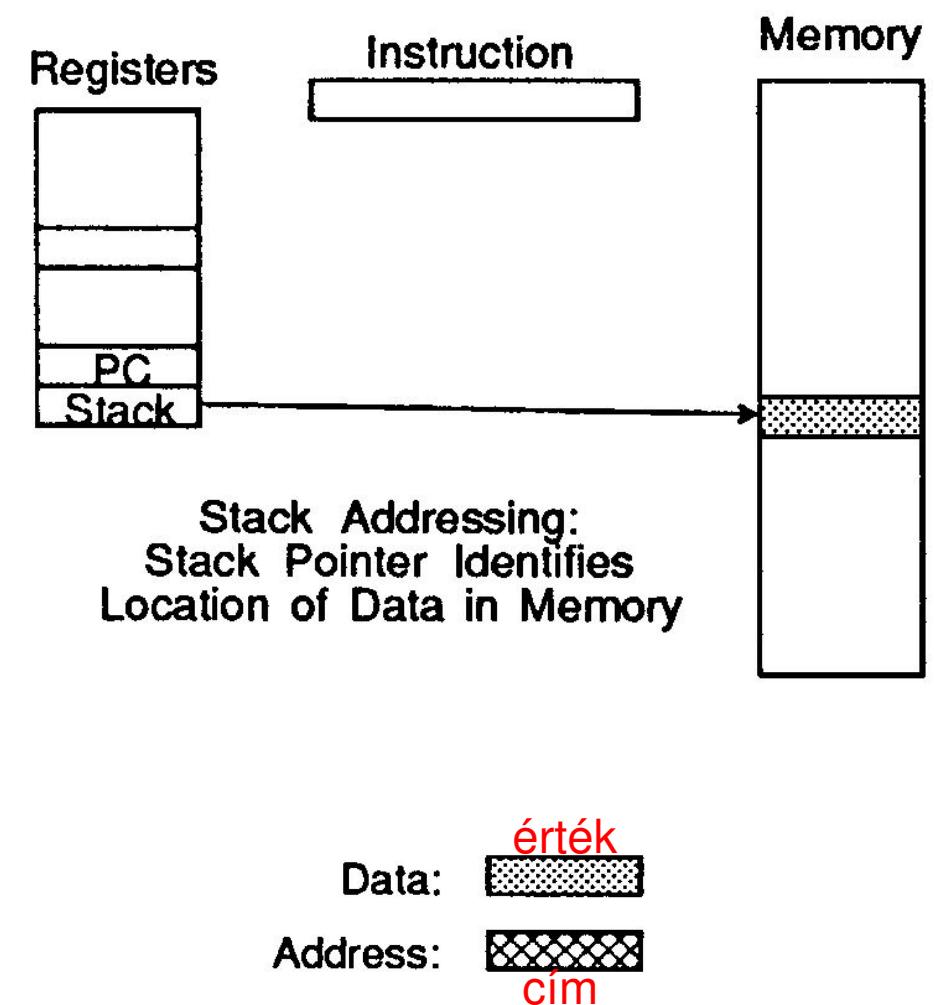
# Összehasonlító táblázat – I.

- Az 1.) – 4.) címzési módok időszükségleteinek összehasonlító táblázata:

Címzési módszer	Memória hivatkozások száma	Fetch (ns)	Execute (ns)	Total Time (ns)
Direkt	6	45	205	250
Indirekt	8	45	275	320
Regiszteres direkt	1	45	25	70
Regiszteres indirekt	4	45	125	170

# 5. Verem (Stack) címzés

- Verem (STACK) címzés, vagy regiszteres indirekt autoincrement címzési mód: indirekt módszerrel az operandus memóriában elfoglalt helyét a címével azonosítjuk, és akár az összes memóriatömbben lévő elem megcímezhető. *Autoincrement*: mivel a címeket automatikusan növeli. Ezt (+) jellet jelöljük: **\*Rx+**
- A Stack-et a regiszterekből foglalhatjuk le. A stackben lévő információra a stack pointerrel (SP-mutatóval) hivatkozunk a memóriára. A Stack egy *LIFO* tároló.
- A stack pointer (SP) címe jelzi verem tetejét (ToS-Top of Stack), ahol a hivatkozott információ található, ill. címmel azonosítható a következő elérhető hely.



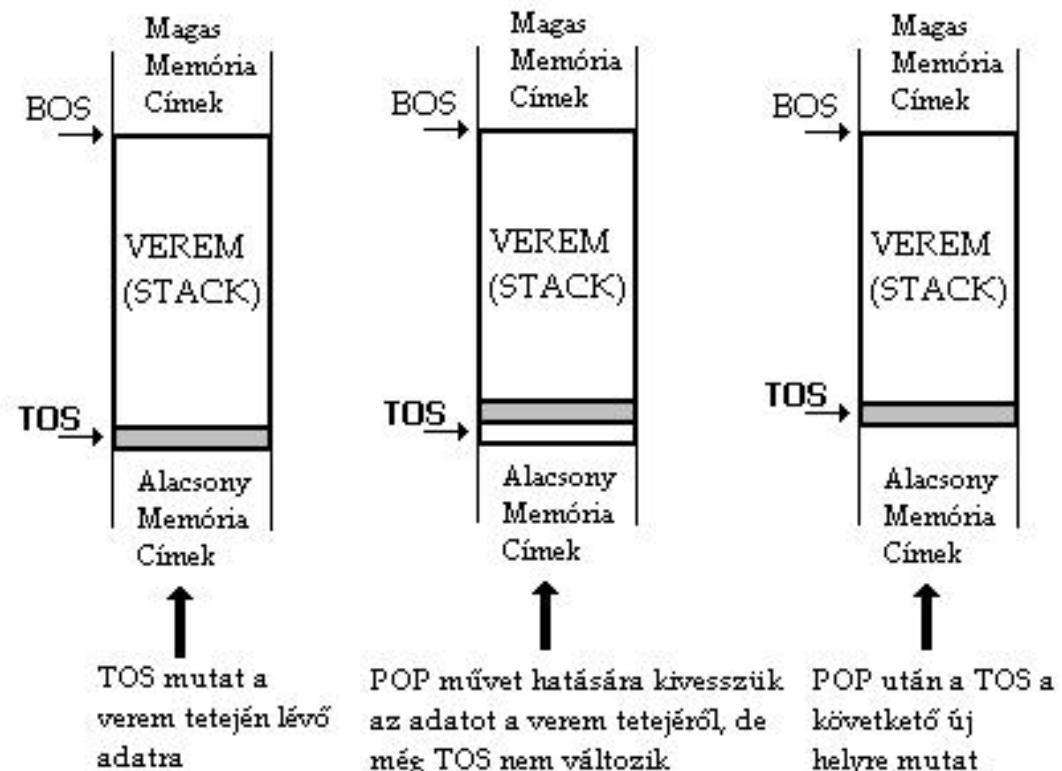
# Verem – PUSH, POP műveletek

- **POP művelet** kiveszük a stack tetején lévő adatot (TOS), és egy Rx regiszterbe rakjuk. Ezután a stack pointer automatikusan inkrementálja a címet, amivel a következő elemet azonosítja a verem tetején.  
Jel: **MOVE \*R (stack pointer)+, Rx**

- **PUSH művelet:** a stack pointer automatikusan dekrementálja a címet, amivel a verem tetején lévő elemet azonosítja. Majd ezután berakjuk az Rx regiszterben lévő elemet a stack tetejére, a pointer átlát mutatott címre.

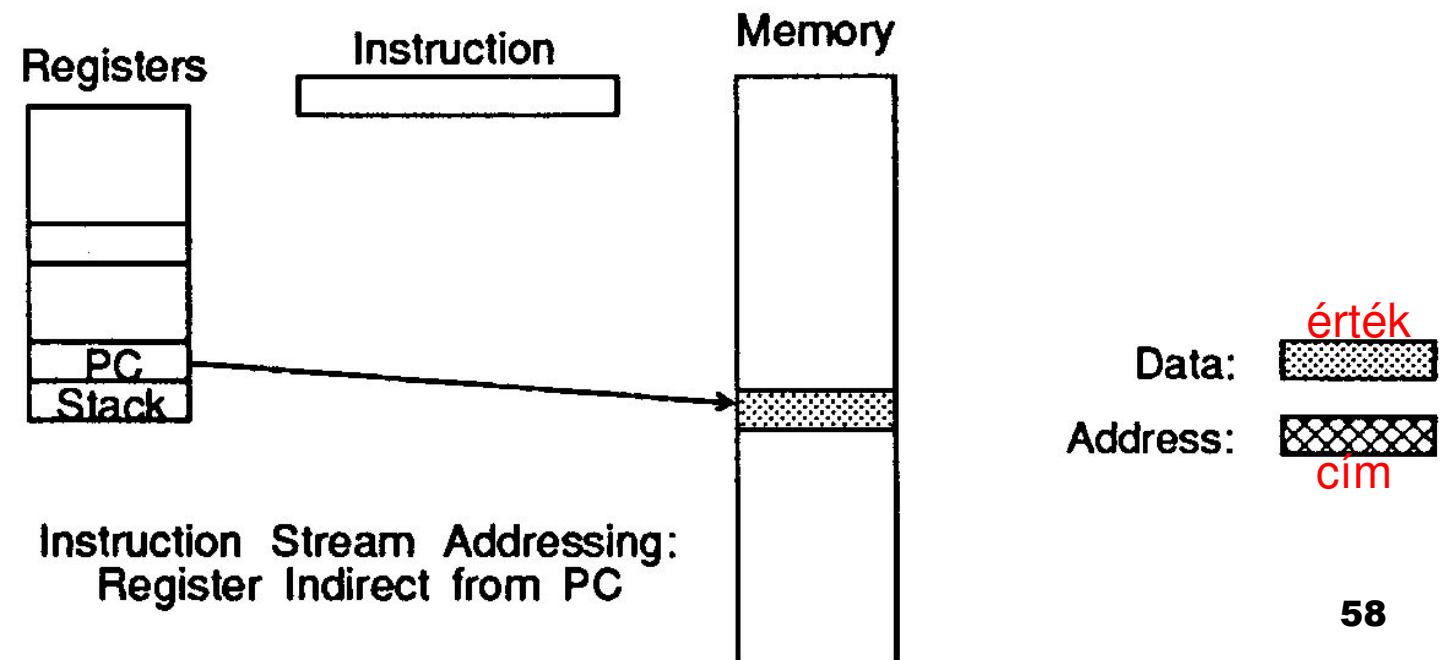
Jel: **MOVE Rx, \*R (stack pointer)-**

Példa: POP műveletre



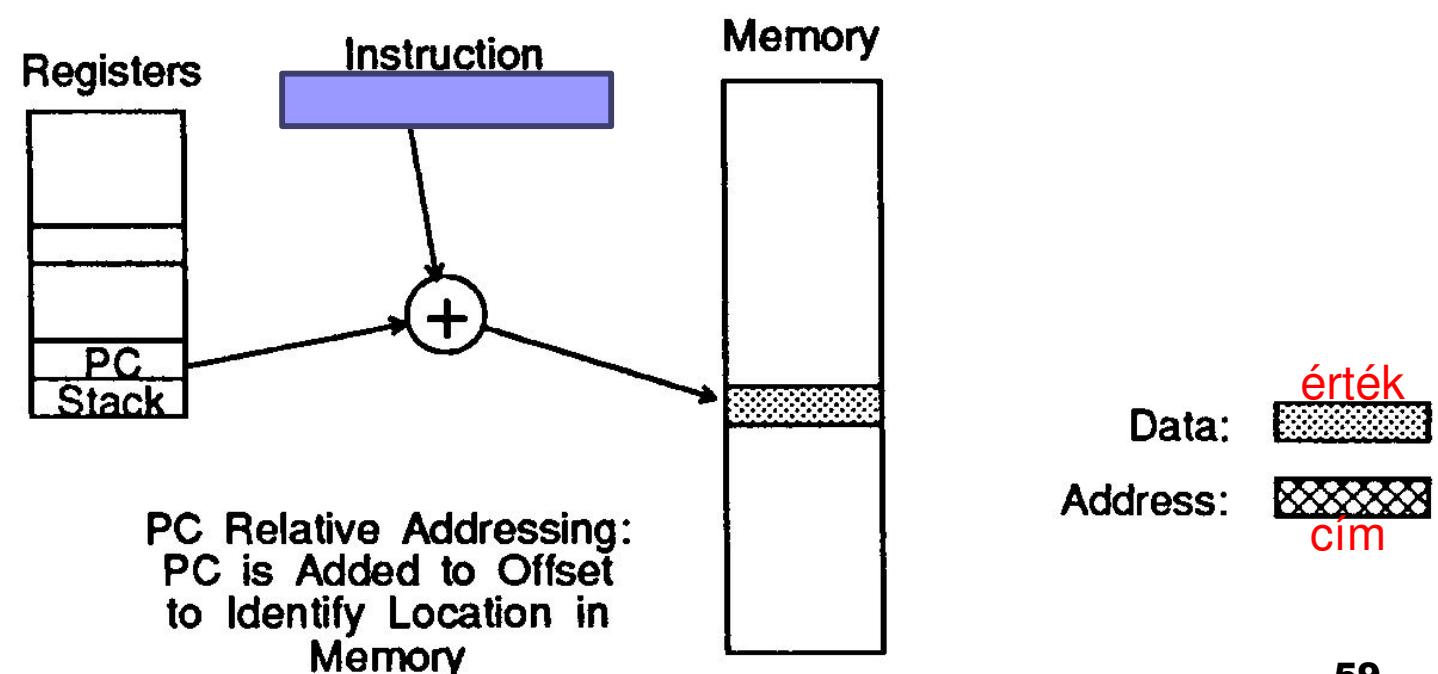
# 6. Instruction Stream címzés

- A PC-ben tárolt utasítás cím segítségével közvetlenül azonosítja a memóriában lévő operandust. Az utasítás végrehajtásakor visszakapjuk az utasításfolyamból magát az utasítást, amelyet a PC azonosít.
- Hívják még *azonnali módszernek* (*imm X*) is, mivel az adatok és címek azonnal a rendelkezésünkre állnak. Konstansok, előre definiált címek szerepelhetnek az utasítás-folyamban.



# 7. PC-relatív címzés

- Memóriában lévő operandusra a regiszteren belüli PC értéke, és az utasítás eltolási (offset) értéke együttesen azonosítja.
  - Effektív cím = Regiszteren belüli PC értéke + Eltolás (offset) értéke

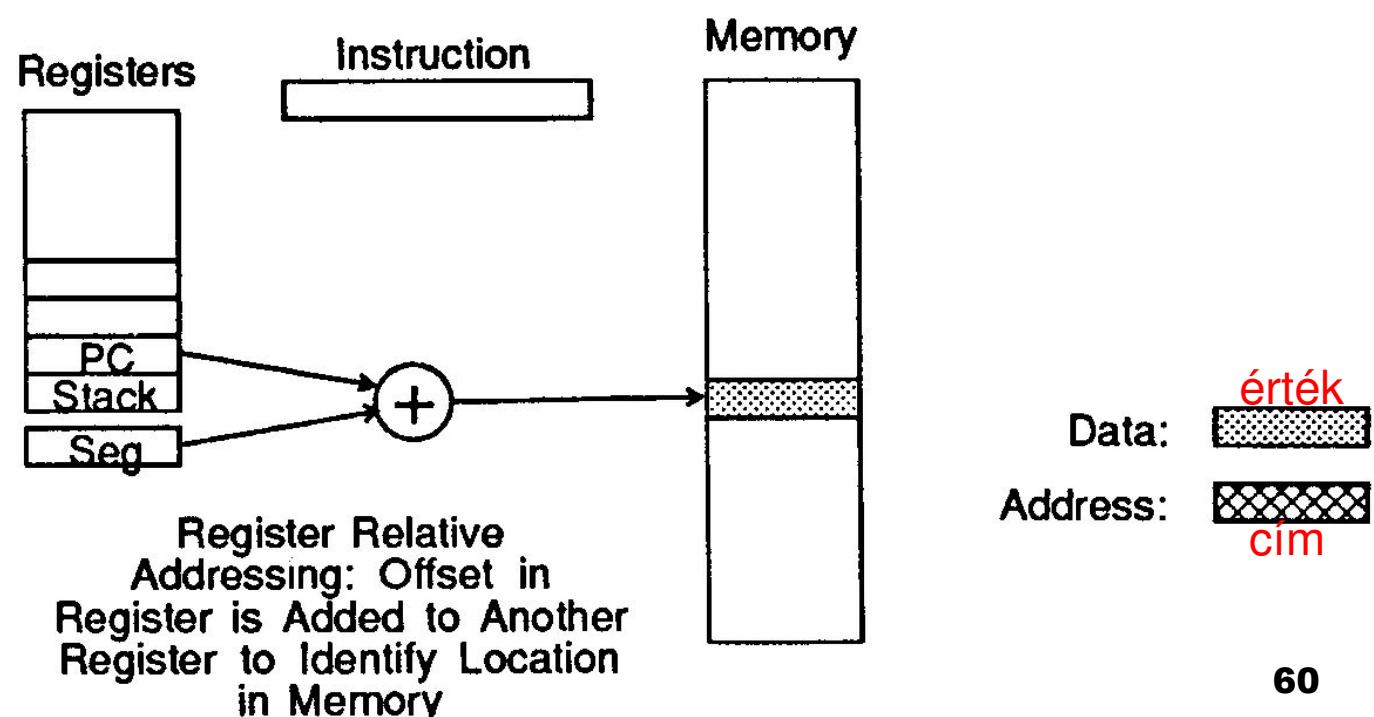


# 8. Regiszter relatív címzés

- A memóriában lévő operandusra a regiszteren belüli PC értékéhez hozzáadódik (egy, vagy akár több) másik, külső Szegmens-regiszter értéke. Tehát ez abban különbözik a PC-relatív címzéstől, hogy itt a címzés két különböző regiszter segítségével történik!
  - Effektív cím = Regiszternek a PC eltolási értéke (offset) + Szegmens regiszter értéke

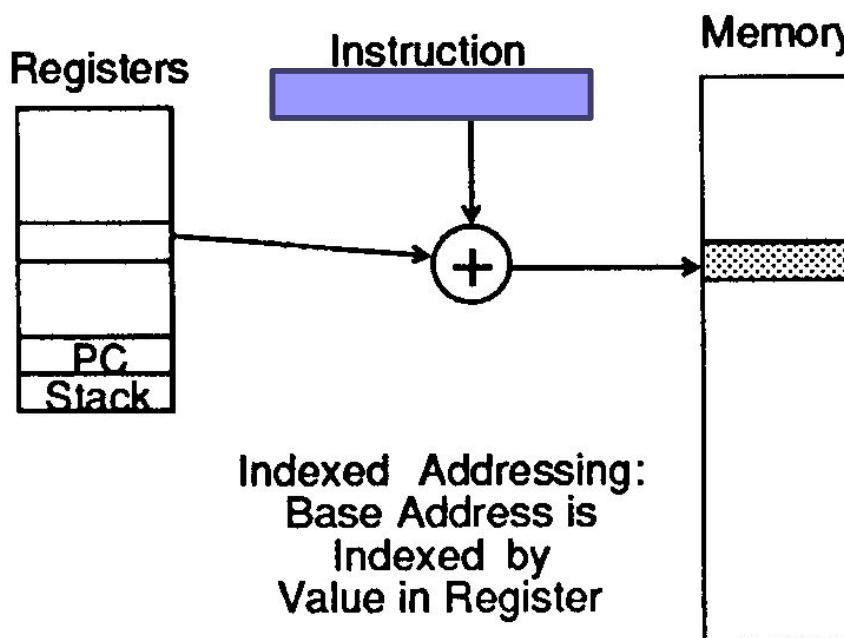
PI: Intel 80x86  
szegmensek:

ds: data  
cs: code  
ss: stack  
es: extra



# 9. Indexelt címzési mód

- A memóriában lévő operandus helyét legalább két érték összegéből kapjuk meg. Tehát a tényleges címet az *indexelt bázisértékből*, és az általános célú regiszter értékéből kapjuk meg. Ezt módszert használják adatstruktúrák indexelt tárolásánál. (Pl: tömböknél)
  - Effektív cím= utasításfolyam bázis értéke + általános célú regiszter értéke



```
int main(void) {  
    int i;  
    ptr = &my_array[0];      /* point our ptr pointer  
                           to the first element of the my_array */  
    printf("\n\n");  
    for (i = 0; i < 6; i++) {  
        printf("my_array[%d] = %d    ", i, my_array[i]);  
        /*ver.A */  
        printf("ptr + %d = %d\n", i, *(ptr + i));  
        /*ver.B */  
    }  
    return 0;  
}
```

# Összehasonlító táblázat – II.

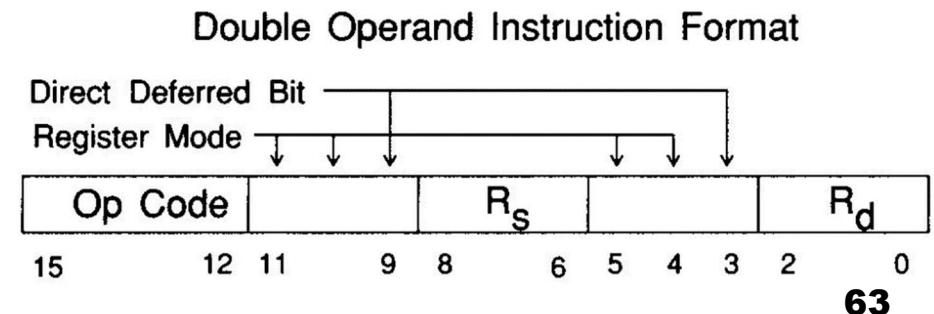
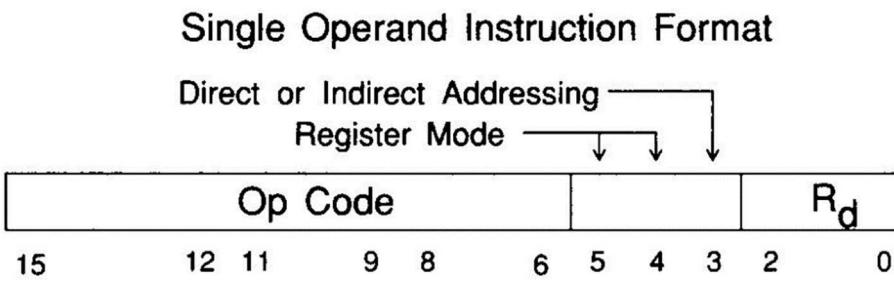
## ■ Címzési módok és jelöléseiik összefoglaló táblázata

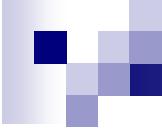
Table 4.1. Addressing Modes and their Nomenclatures.

<i>Addressing Mode</i>	<i>Represented By</i>	<i>Comment</i>
Direct	@<address>	Address is part of instruction.
Register Direct	Rn	Operand is found in register.
Indirect	*(<address>)	Address is part of instruction; operand is located in memory at that address.
Register Indirect	*Rn	Address found in register; operand in memory at that address.
Instruction Stream	#<value>	Value is stored in instruction stream.
Register Indirect Autoincrement	*Rn+	Register used as address; value in register incremented at end of instruction.
Stack Addressing	Push Pop	Stack pointer identifies location in main store for transfers; value in stack pointer adjusted as necessary.
PC Relative	\$<offset>	Offset identifies target address relative to current location identified by program counter.
Memory-Based Index	(<address> i Rm)	Operand is located in memory at address which is sum of <address> and Rm.
Register-Based Index	(Rn i Rm)	Operand is located in memory at address which is sum of Rn and Rm.

# Példa 1: DEC PDP11 működése

- Egyszerű számítógépünk támogatott utasítása(i)
- Különböző címzési módokat használt
- 16-bites gép: utasítás opcode-ja + további infók (cím)
  - 3 biten: 8 általános célú regiszter ([2:0]) – **R<sub>d</sub>**-nek lefoglalt rész
  - További 3 biten: **Rsource** (regiszter specifikáció használathoz)
  - !Dupla operandus esetén: csak 4 bit marad az utasítások kódolására (**opcode**)
  - Egyszeres operandus esetén: 10 bites **opcode**
- Egyszeres- és dupla operandusú utasítás formátum





# Programszervező utasítások

# Programszervező utasítások

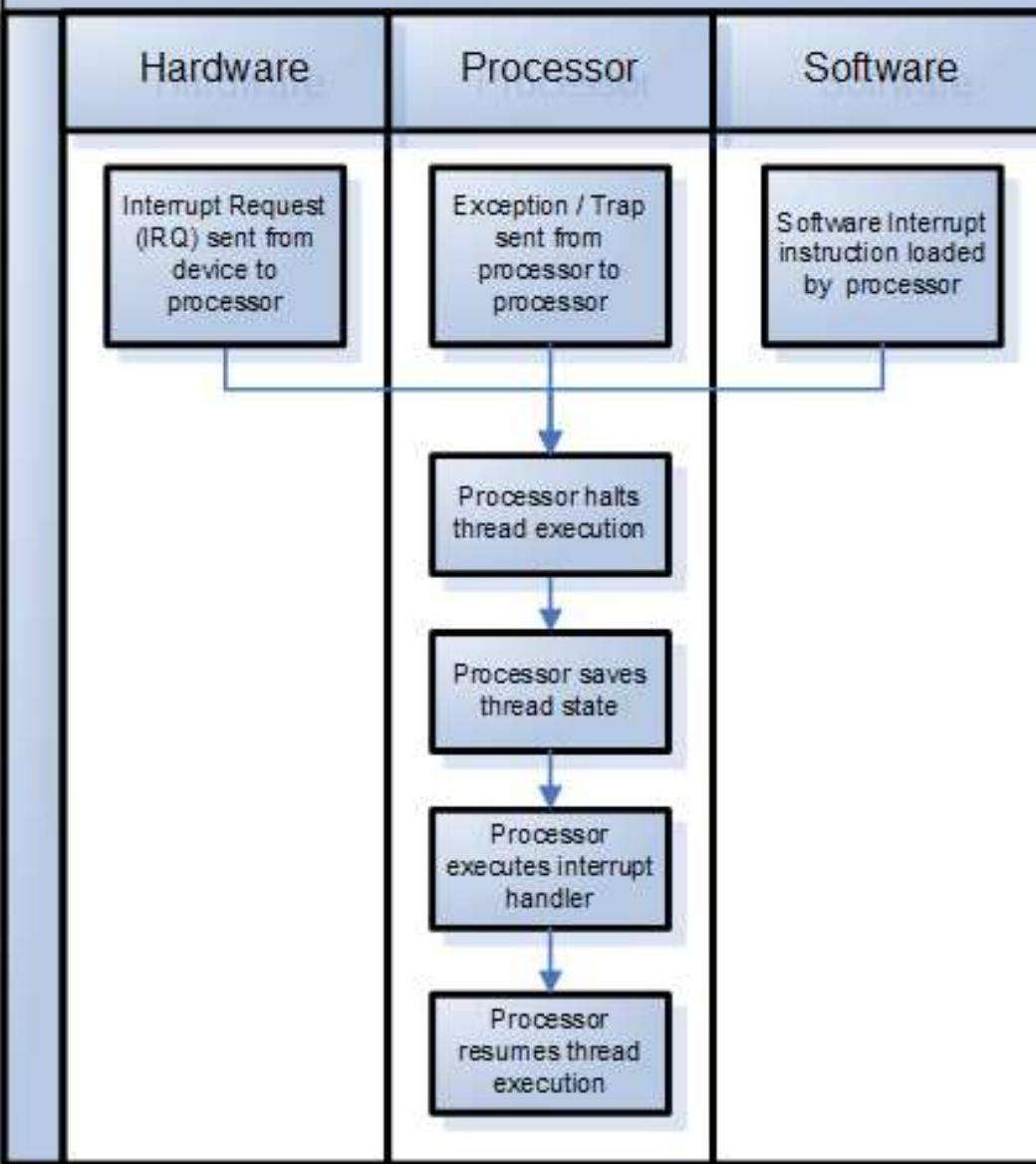
- Program végrehajtásának szabályozása:
  - Feltételes, feltétel nélküli utasítások (IF BRANCH)
  - Függvény-Szubrutin (eljárás)
    - hívás (CALL) / visszatérés (RETURN)
  - Ciklus-loop (iteratív végrehatás)
  - Ugró utasítások (JUMP)
    - Feltételes
    - Feltétel nélküli

# További programvezérlési (program-control) módszerek

- I/O vezérlés:
  - memory-mapped I/O
  - DMA: Direct Memory Access (lásd. chapter6.pdf)
- Megszakítás (interrupt) [IRQ]
  - HW: nem-maszkolható interrupt (NMI) = nem / maszkolható (ignorable)
  - CPU
  - SW: 
- Trap (csapda): programvezérelt megszakítás
  - =Exception: kivétel kezelése (pl. 0-val osztás)

# Megszakítások

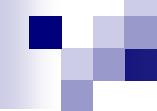
Interrupt Process (from three potential sources)



## Megszakítás (interrupt)

- HW: külső eszköz
- CPU: multiprocesszoros, társpolrocesszoros rendszerben másik CPU-tól érkező
- SW: speciális utasítás, vagy program végrehajtása végén küldi (exception is lehet)

Minden megszakításhoz saját „lekezelő” handler tartozik



# RISC és CISC processzorok utasításkészletei

# Utasítás készletek

- Fontos paraméter: utasítások száma!
- Kezdetben egyszerű felépítésű gépek
  - Egyszerű utasítások és gépi nyelv
- Azonban a komplex problémákat kívántak megoldani (magasabb szintű leírással) →  
**„Szemantikus rés”**
- Megoldás: compiler
- **ISA (Instruction Set Architecture): CISC, RISC architektúrák**

# Utasításkészlet csoportosítása

- Aritmetikai utasítások
- Logikai utasítások
- Vezérlésátadó utasítások
- Verem-kezelő utasítások
- Adat-kezelő utasítások
- Lebegőpontos utasítások  
(nem minden CPU támogatja)
- Multimédiás utasítások (Pentium utáni CPU-k)
  - MMX, SSE

# RISC processzorok jellemzői (1):

- **RISC:** Reduced Instruction Set Computer (Csökkentett utasításkészletű számítógép):
- Csak a **kívánt alkalmazásra jellemző utasítástípusokat** tartalmaz, az utasításkészlet összetettségének csökkentése végett kihagyta olyan utasításokat, amelyeket a program amúgy sem használ, ezáltal nő a sebesség.
- **Minimális utasításkészletet és címzési módot** (csak amit gyakran használ), **gyors HW elemeket**, optimalizált SW használ.
- Azonban, hogy a programozási nyelvek komplex függvényei leírhatók legyenek (ahogyan az a CISC-nél működik) szubrutinokra, és hosszabb utasítássorozatokra (**sok egyszerű utasítás**) van szükség.
- Hogyan tudjuk a rendszer erőforrásait hatékonyan kihasználni? Gyorsabb működés érhető el (**MIPS**), egyszerűbb architektúra megvalósítására kell törekedni.
- **Azonos hosszúságú utasításformátum** (korlátozott utasításformátum miatt a tárolt programú gépeknél az F-D-E folyamatban a dekódolás minimális idejű lesz (nullának feltételezzük), amely során azonosítani kell a végrehajtandó utasítást)
- Például: Beágyazott rendszerek speciális feldolgozó egységei: pl. MCU=mikrokontrollerek, DSP=Digitális jelfeldolgozó processzorok, FPGA-CPLD: Programozható logikai áramkörök beágyazott processzorai,
- További példák: Motorola 88000 RISC rendszere, vagy Berkeley RISC-I rendszere, Alpha, SPARC, MicroChip MCU, ARM, DSP sorozatok, IBM PowerPC stb.

# RISC processzorok jellemzői (2):

- **Huzalozott (*hardwired*) vezérlés és utasításdekódolás** (CU a hardveres dekódolás megvalósításához fix-kombinációs logikát használt, azonban a mai memóriaalapú mikro-kódú gépeknél ez már módosítható).
- **Egyszeres ciklusvégrehajtás:** ( minden egyes ciklusban egy utasítást hajt végre, ha ezt sikerülne elérni optimális lenne az erőforrás kihasználás - VLSI technológiáfüggő. Egy lebegőpontos művelet rendkívül kis idő alatt végrehajtható. Hátránya, hogy vannak bizonyos műveletek, amelyeket egy ciklus alatt nem kapunk meg: pl. a memoriában lévő érték inkrementálásakor az értéket előbb ki kell venni, frissíteni, majd visszaírni a memoriába).
- **LOAD/STORE memóriaszervezés:** 2 művelet – tölt és tárol (regiszter <-> memória). Regiszterre azért van szükség, mivel a betöltött adatot sokkal gyorsabban tudjuk kiolvasni, mint a memoriából. Az aritmetikai/logikai utasítások a regiszterekben tárolódnak. A regiszterek gyorsabbak, mint a memória-intenzív műveletek.
- További architektúra technikák: **utasítás pipe-line (utasítás feldolgozás „látszólagos” párhuzamosítása)**, többszörös adatvonalak, nagyszámú gyors regiszterek alkalmazásával.

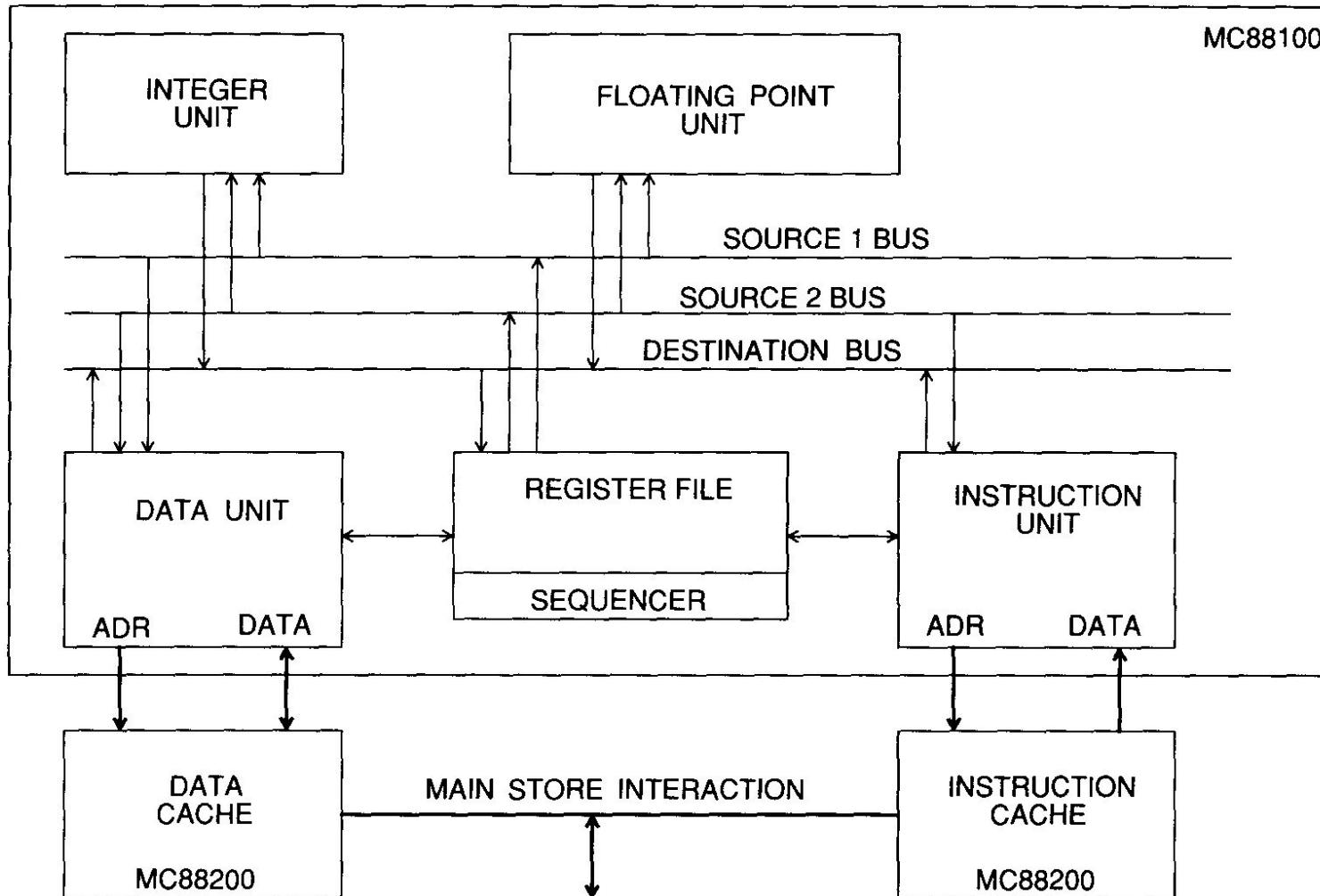
# RISC: „Pipe-line” technika

- **Utasítás szintű „látszólagos” párhuzamosítás:** A soros feldolgozással ellentétben, egy feladat egymástól független részei (fázisai) a rendszer különböző pontjain egyszerre, egy időben hajtódnak végre, ezáltal növekszik a sebesség. Az operandusokat gyors regiszterekben tároljuk. Azonos hosszúságú utasítások gyors F-D-E eljárása. Egy ciklusban egyszerre történik különböző utasításrészletek Fetch-Decode-Execute fázisok feldolgozása (gyors fetch és dekódolás).
- Többszörös adatvonalak párhuzamos végrehajtást engednek meg (hardveres párhuzamosítás). Tehát egy órajelciklus alatt több utasítást tudnak feldolgozni. (pl. Source-1,2, Destination adatbuszok a Motorola 88000 rendszerben.)

3 lépcsős pipe-line

	1 fázis	2 fázis	3 fázis	...	...
1.utasítás	F	D	E	F	D
2.utasítás	-	F	D	E	F
3.utasítás	-	-	F	D	E

# Motorola 88000 RISC rendszere

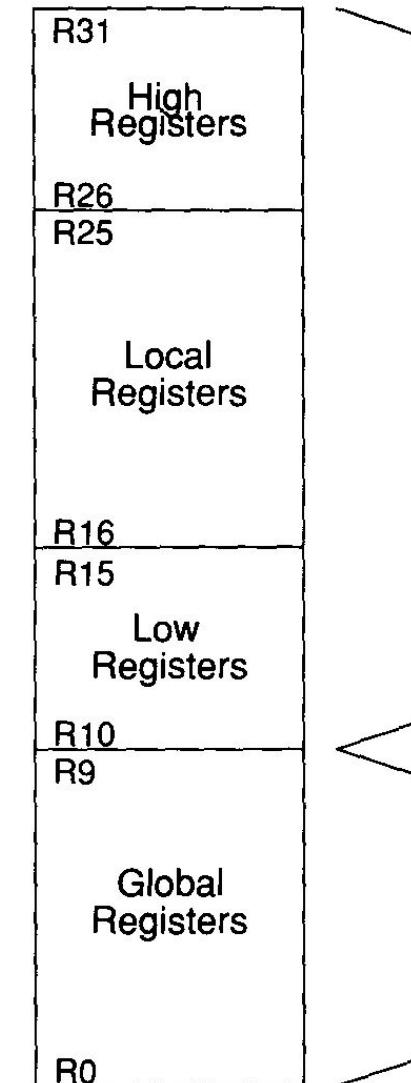


Hardveres  
párhuzamosítás:

- Buszok
  - Cache
  - Data / Instruction Unit
- 
- (Harvard Arch!)
  - Max 3-című utasítások támogatása

# Berkeley RISC-I rendszere: Regiszter használat

- **Regiszter ablak:** a szubrutin híváshoz (call) / visszatéréshez (ret) szükséges processzor-időt kívánták minimalizálni nagy számú regiszter stack használatával.
- Regisztereknek csak egy kis része érhető el („**ablak**”). Egy pointer mutat az ablakra, amely azonosítja a benne található aktuális regisztereket. Ha a szubrutinok között „átlapolódás” van az ablakon belül, akkor történhet paraméter átadás.

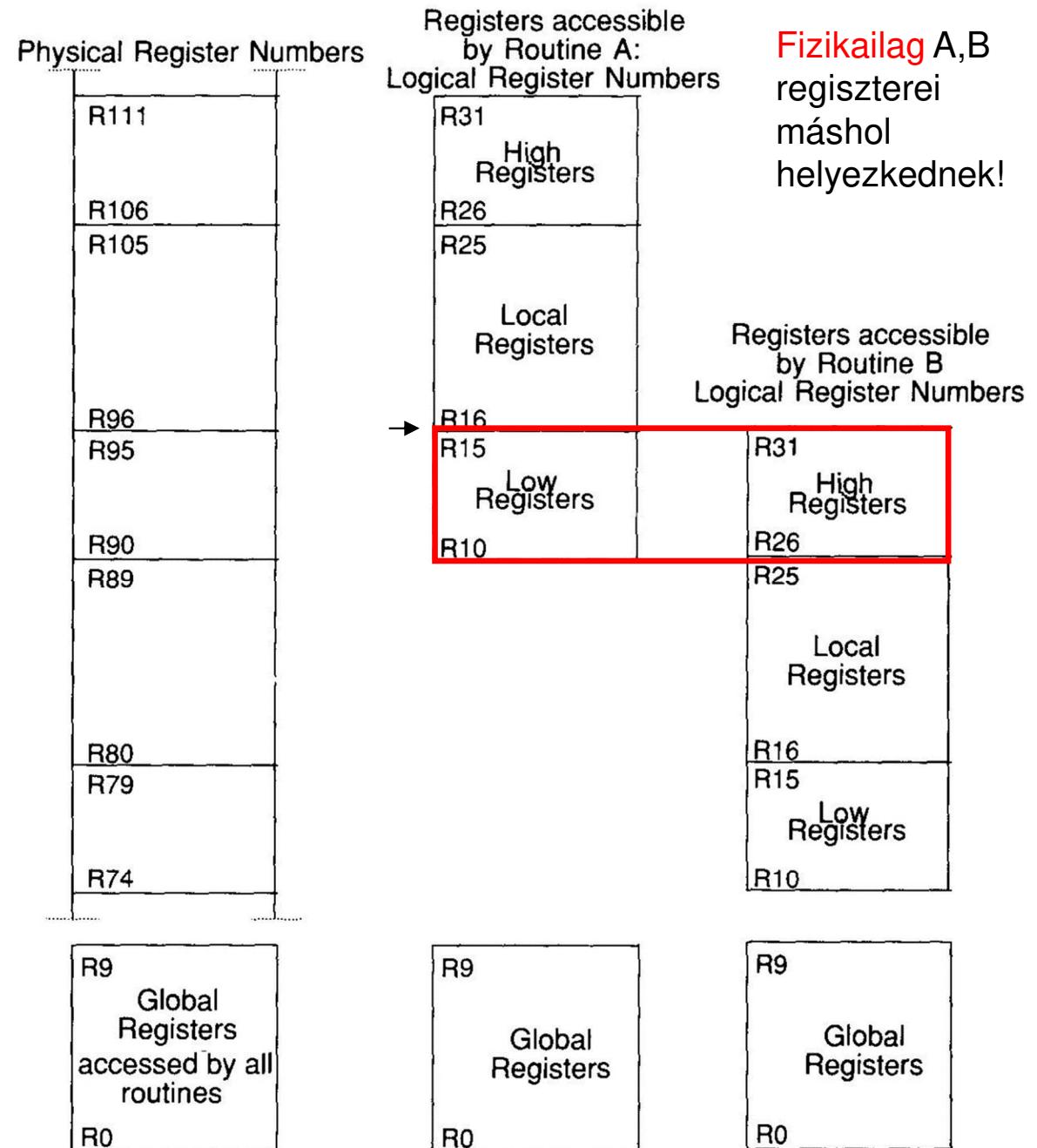


Routine Specific Registers  
Each level of subroutine  
call accesses different  
set, yet the sets overlap

Common Registers:  
Accessed by all routines  
as R0-R9

# „Regiszter ablakozási,” technika

- Paraméter átadás „ablakozással”: a globális regiszterekek keresztül történik, amelyet minden szubrutin elérhet.
- 5 bit → 32 regiszter A(R0-R31) címezhető meg B(R0-R31)
- közös (globális) regiszterek: R0-R9, minden szubrutin által elérhetők
- rutin specifikus regiszterek: R10-R31 mely további három részből áll (a regiszterek között történhet átlapolódás!)
  - Alacsony-szintű regiszterek: R10-R15
  - Lokális regiszterek: R16-R25
  - Magas-szintű regiszterek: R26-R31
- Ez az eljárás mindaddig jól működik, ameddig a paraméterek száma kisebb a regiszterek méreténél, mivel nem igényel memória-intenzív Stack műveletet.



# CISC Processzorok jellemzői

- **CISC:** Complex Instruction Set Computer
- **Nagyszámú utasítás-típust**, és **címzési módot** tartalmaz, **egy utasítással több elemi feladatot végre tud hajtani**. **Változó méretű utasításformátum** miatt a dekódolónak először azonosítania kell az utasítás hosszát, az utasításfolyamból kinyerni a szükséges információt, és csak ezután tudja végrehajtani a feladatát.
- **Lassabb** a változó méretű utasítások **dekódolása!**
- A korai gépeknek egyszerű a felépítése, de bonyolult a nyelvezete. Összetett problémákat megoldása a gépi kódnál magasabb szintű nyelven. **Szemantikus rész** = a gépi és felhasználó nyelve közötti különbség. Ennek áthidalására új nyelvek születtek: Fortran, Lisp, Pascal, C, amelyek bonyolultabb problémákat is egyszerűen képesek kezelni. **Komplexebb gépek** születtek, amelyek gyorsak, sokoldalúak voltak.
- *Compiler* = *Fordító*: a bemenetén a probléma felhasználói nyelven van leírva, míg a kimenetén a megoldást gépi nyelvre fordítja le.
- Megfigyelték, hogy a processzor munkája során a rendelkezésre álló utasításoknak csak egy részét használja (20%-os használat, az idő 80%-ában).
- Ugyanaz a komplex program, függvény *kevesebb elemi utasítássorozattal* is megvalósítható. Memória, vagy regiszter alapú technikát használ.

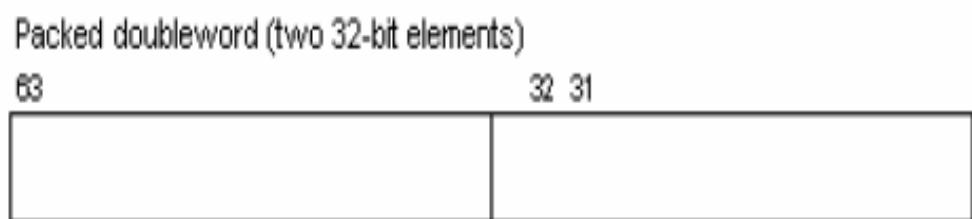
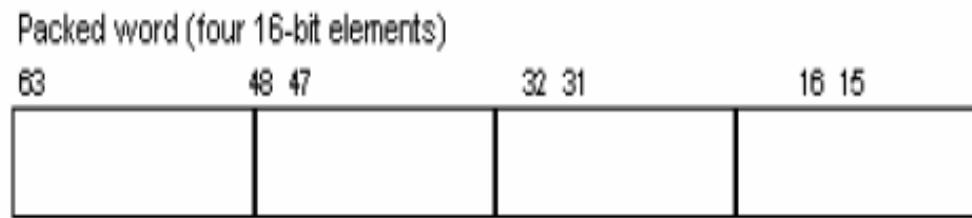
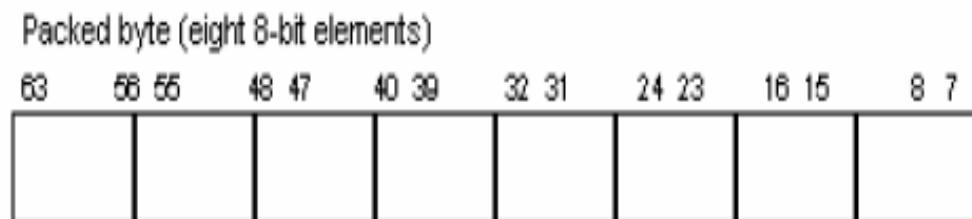
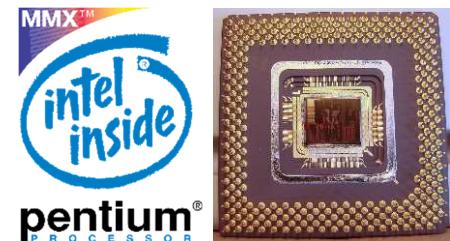
# CISC Processzorok jellemzői (2)

- Közvetlen memória-elérés (**DMA**) és összetett/bonyolult műveletek jellemzők rá.
- **Mikro-programozott vezérlés (CU)**
  - a CISC processzor esetén a fordító (compiler) a programot egyszerűbb szintre fordítja, majd ezután a mikroprogram (ami meglehetősen összetett lehet) veszi át a vezérlést – mikroutasítások sorozata a mikrokódos memóriában.
- Példák:
  - System/360, VAX, DEC PDP-11/VAX rendszerei, Motorola 68000 család, és **AMDx86-32/64** és **Intel x86-32/64 CPUs**

# Pl. MMX kiterjesztés

## ■ **MMX: Multi-Media Extension** (Intel Pentium sorozat 1996) –

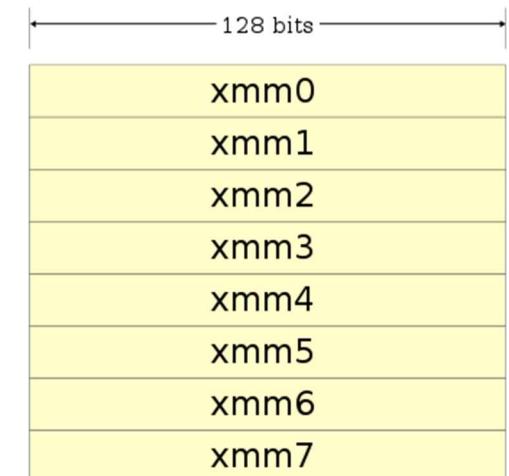
- SIMD: Single Instruction / Multiple Data alapú **integer!** stream data feldolgozásra (jelfeldolgozás)
  - 8 db MM0..7 regiszter (8 bit/reg)
  - Regiszterek adatait 4 különböző formátumban lehet tárolni (packet)
  - 57 MMX utasítás, 6 fő műveleti osztályban:
    - ADD
    - SUBTRACT
    - MULTIPLY
    - MULTIPLY THEN ADD (MAC – FIR)
    - COMPARISON
    - LOGICAL
      - AND, NAND, OR, XOR stb.



# Pl. SSE, SSE2 kiterjesztés

Eredeti nevén **KNI**: *Katmai New Instructions* (első Intel Pentium III-nál, 1999)

- **SSE**: Streaming SIMD Extension (**lebegőpontos** és **fixpontos** adatfolyam utasításai) //Intel, AMD CPU-k
- 32-bites módban 8 db, egyenként 128-bites regiszter csomag (xmm0...7)
- SSE-1:
  - 128-bit „packed” IEEE *single-precision* floating-point műveletek (~70 utasítás).
  - 2 órajel ciklus alatt számíthatóak
- SSE-2:
  - 128-bit „packed” IEEE *double-precision* SIMD floating-point műveletek (~144 utasítás), vagy
  - 128-bit „packed” egész típusú SIMD műveletek
    - 8-, 16-, 32-, és 64-bites operandusok támogatása
  - 2 órajel ciklus alatt számíthatóak





# Számítógép Architektúrák II.

(MIVIB344ZV)

7. előadás: I/O műveletek  
PCI, PCI Express, SCSI buszok

Előadó: Dr. Vörösházi Zsolt  
[voroshazi.zsolt@mik.uni-pannon.hu](mailto:voroshazi.zsolt@mik.uni-pannon.hu)

# Jegyzetek, segédanyagok:

- Könyvfejezetek:

- <http://www.virt.uni-pannon.hu> → Oktatás → Tantárgyak → Számítógép Architektúrák II.  
□ (chapter06.pdf)

- Fóliák, óravázlatok .ppt (.pdf)

- Feltöltésük folyamatosan

# I/O műveletek

- **Aszinkron protokoll**
- **Szinkron protokoll**
- **Arbitráció (döntési mechanizmus)**
- Megszakítás – kezelés (operációs rendszerek)
- Buszok – Buszrendszerek:
  - PCI,
  - PCI-Express,
  - SCSI buszok

# I/O egységek

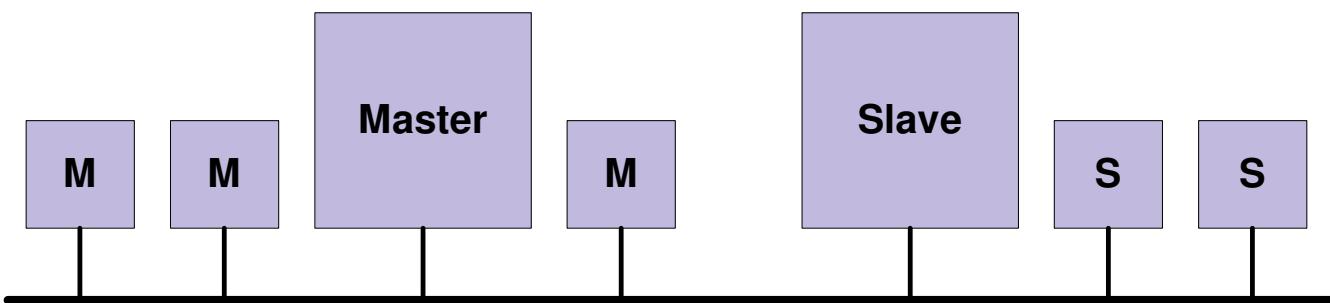
- A számítógép a külvilággal, *perifériákkal* az I/O egységeken keresztül tartja a kapcsolatot. Az információ továbbítását az egységek között buszok végzik, amelyek interfészekkel kapcsolódnak egymáshoz.
- **Interface:** azon szabályok összessége, amelyek mind a fizikai megjelenést, kapcsolatot, mind pedig a kommunikációs folyamatokat leírják. Egy busz általában 3 fő kommunikációs vonalból állhat:
  - vezérlőbusz,
  - adatbusz, és
  - címbusz (esetleg utasításbusz).

# I/O kommunikációs protokollok:

- Kommunikáció során megkülönböztetünk egy eszközpárt: a **Master-t** és **Slave-t**. A Master (pl. CPU) általában, mint kezdeményező, birtokolja és ellenőrzi a buszt, és átadja (ír / olvas) az adatokat a Slave-nek (pl. Memória).
- A kommunikációhoz előredefiniált **protokollokra** (szabályok és konvenciók gyűjteménye) van szükség, amelyek meghatározzák az események sorrendjét és időzítését. A kommunikáció feltétele a másik egység állapotának pontos ismerete. Lehetséges több M-S modul (pl. multi-master rendszer több kezdeményezővel) is egy rendszerben.
- **Két alapvető protokoll különböztethető meg:**
  - 1.) Aszinkron kommunikáció (pl. régi SCSI busz), és
  - 2.) Szinkron kommunikáció (pl. PCI busz, PCI-e, mai SCSI).

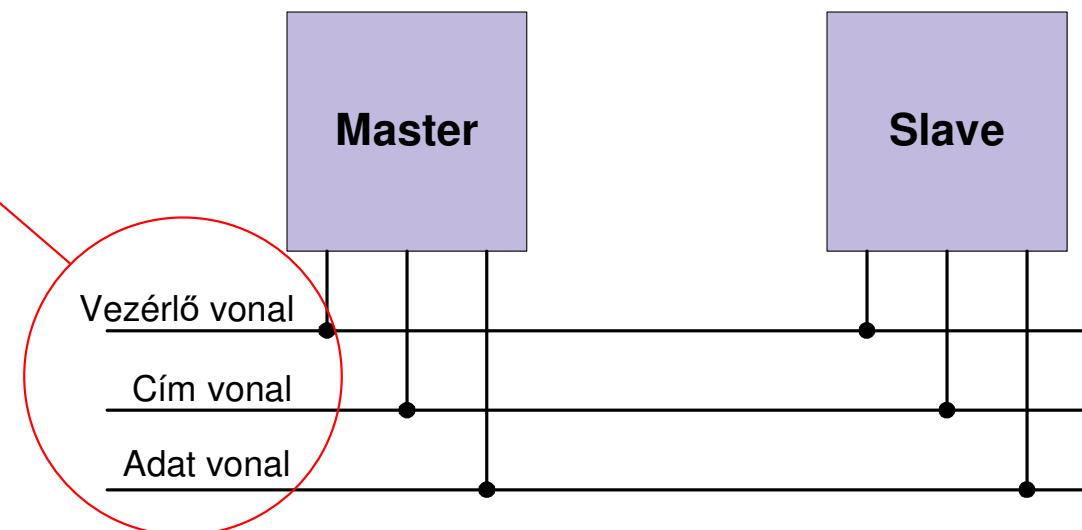
# Busz rendszer Master-Slave moduljainak szervezése

- Modul-szervezés



Busz-Master (busz-vezérlő) kommunikál a Busz-Slave-el (responder).

- Busz-szervezés: Master - Slave



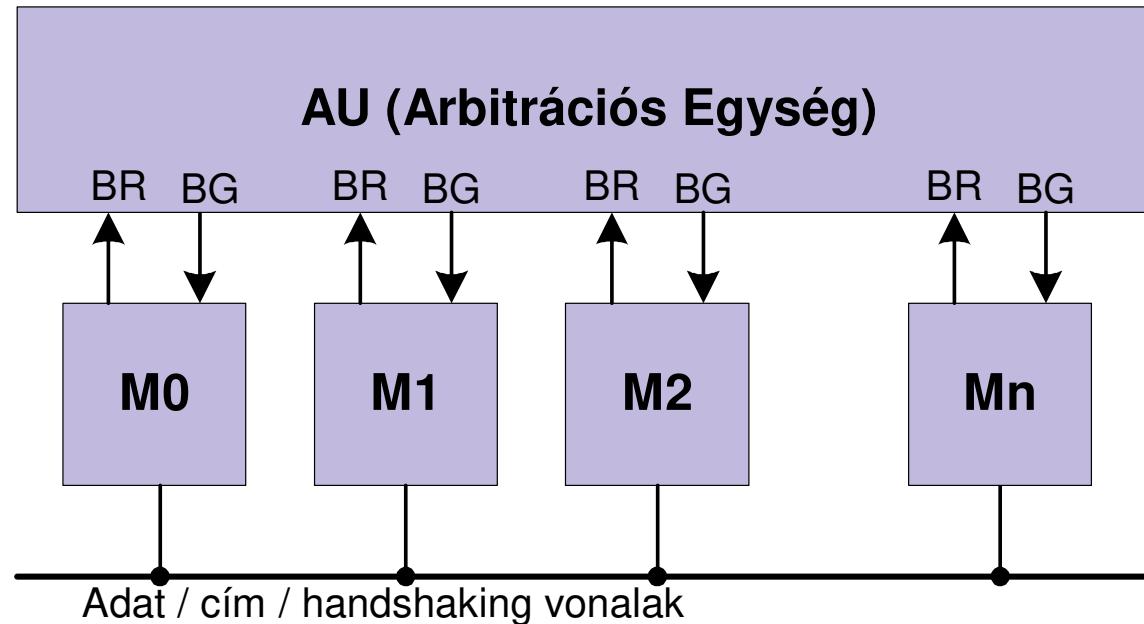


# Busz Arbitráció

# Arbitráció

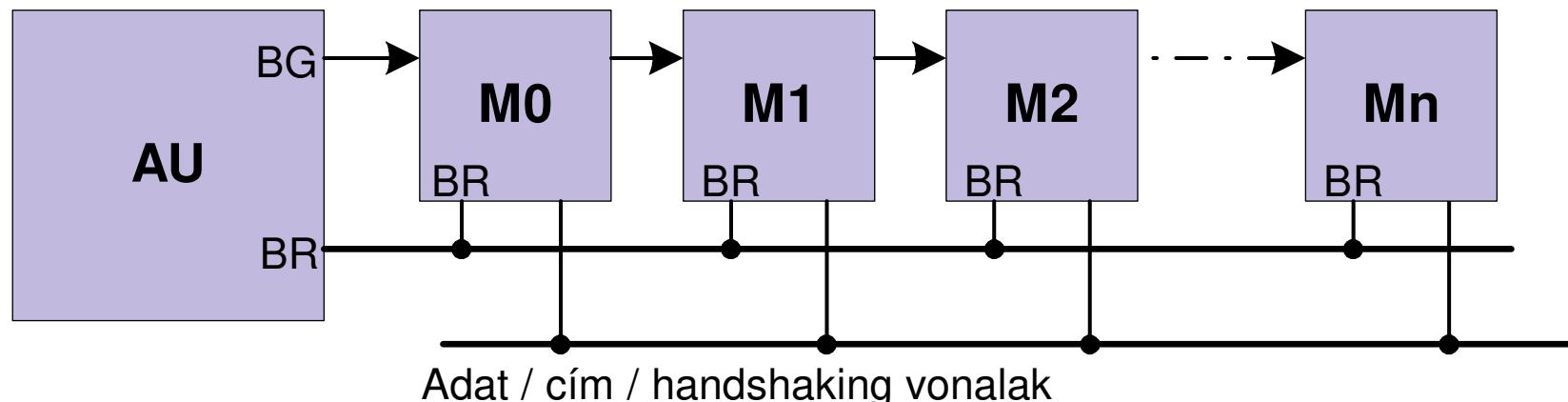
- Egy tetszőleges I/O művelet esetén (aszinkron v. szinkron buszos átvitel) több Master (commander) egység is meg akarja szerezni egyszerre a busz irányítását. Különböző előredefiniált algoritmusok segítségével egyértelműen azonosítható, hogy a „**versenyhelyzetben**” a következő átvitelt (a következő ciklusban) melyik Master fogja megvalósítani. Az **arbitrációs eljárás** egy **döntési folyamat (mechanizmus)**, amely az adatátvitellel párhuzamosan zajlik le, és még az aktuális adatátvitel befejezése előtt eldől, hogy melyik következő master adhat. A Mastereket M<sub>1</sub>...M<sub>n</sub>-el jelöljük, a *BR: Bus Request* (busz kérése, igénylése) a Master által, míg a *BG: Bus Grant* (igénylés elfogadása, engedélyezés) jelet a központi *AU: Arbitration Unit* (arbitrációs egység) bocsátja ki.
- **Az arbitrációnak 3 típusa** van:
  - a.) párhuzamos,
  - b.) soros (daisy chain), és
  - c.) lekérdezéses (polling).

# a.) Párhuzamos



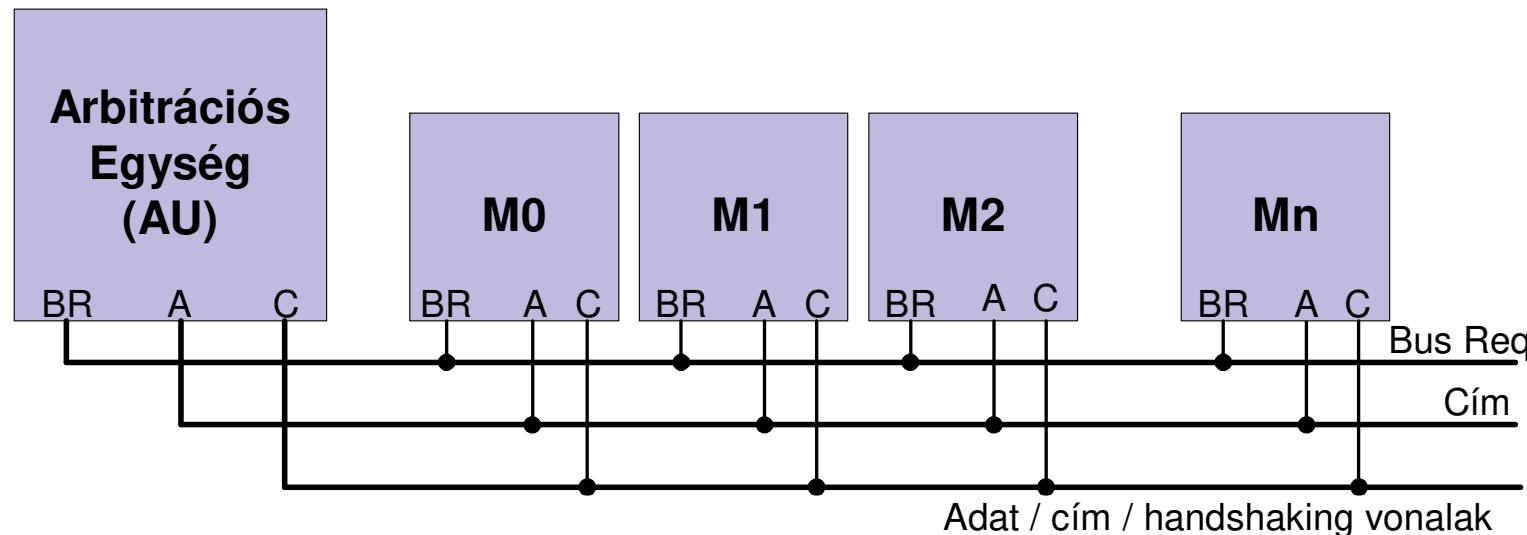
- Ez a leggyorsabb módszer, minden  $M_i$ -nek kitüntetett, egyenrangú kapcsolata van az AU-val (két vonalon: BG és BR-en). Az arbitráció lehet (i) *first asserted/first served (FIFO)*, vagy (ii) *round robin*, vagy (iii) *prioritásos alapú*. Ezek a módszerek többfajta lehetőséget biztosítanak mind egyszerű, mind pedig komplex esetben.
- Az AU vezérli a közös buszon az adatátvitelt. Az adat-cím-handshake vonalat a kijelölt master kezeli, az tranzakció befejeztével pedig kiadja ismét a BR-jelet. Hátránya, hogy drágább, mint a többi módszer (a párhuzamos ágak miatt minden  $M_i$ -hez 2 vonal kell), és az  $M_i$ -k száma korlátozott.

## b.) Soros (daisy chain)



- A BG vonal sorosan van kötve, míg a BR vonal minden egyik  $M_i$ -hez csatlakozik. Az AU nem ismeri, hogy pontosan melyik M kívánja elérni a buszt, ezáltal az átvitel leegyszerűsödik: csupán azt tudja, hogy bizonyos ciklusonként BG-jelet kell kibocsátania. A soros csatlakozás miatt a legelső Master ( $M_0$ ) rendelkezik a legnagyobb prioritással (**fizikai prioritás**), így ha ő igényelt, minden esetben megkapja a buszt.
- Bármennyi eszközt is sorba köthetünk, nincs felső korlátja. Hátránya, hogy az arbitrációs idő (ha a legutolsó  $M_i$  igényel és az előtte lévők nem) egyenes arányban van a sorba kapcsolt  $M_i$ -k számával.

# c.) Lekérdezéses (polling)



- Mindegyik Master egy **közös** BR buszon igényelhet (ID),
- AU dönti el, hogy melyik Masternek adja a buszt. Annak az M-nek a **címét** az address vonalra rakja. minden ciklusban megnézi az igényléseket, és az aktuálisan legmagasabb prioritással rendelkezőt fogadja el. Itt is többféle prioritásos módszer valósítható meg: pl. (FIFO, round robin stb.).
- Hátránya, hogy nagyobb az időszükséglete a párhuzamosnál, így a lassabb periféria műveletek esetén alkalmazzák (pl: I/O kérések arbitrációjánál), a processzor egy program futtatásával folyamatosan monitorozza az I/O eszközök busz kéréseinek állapotát
  - (a megszakításos I/O műveletek ennél a módszernél jobbak).

# 1.) Aszinkron busz protokollok

# Aszinkron busz protokollok

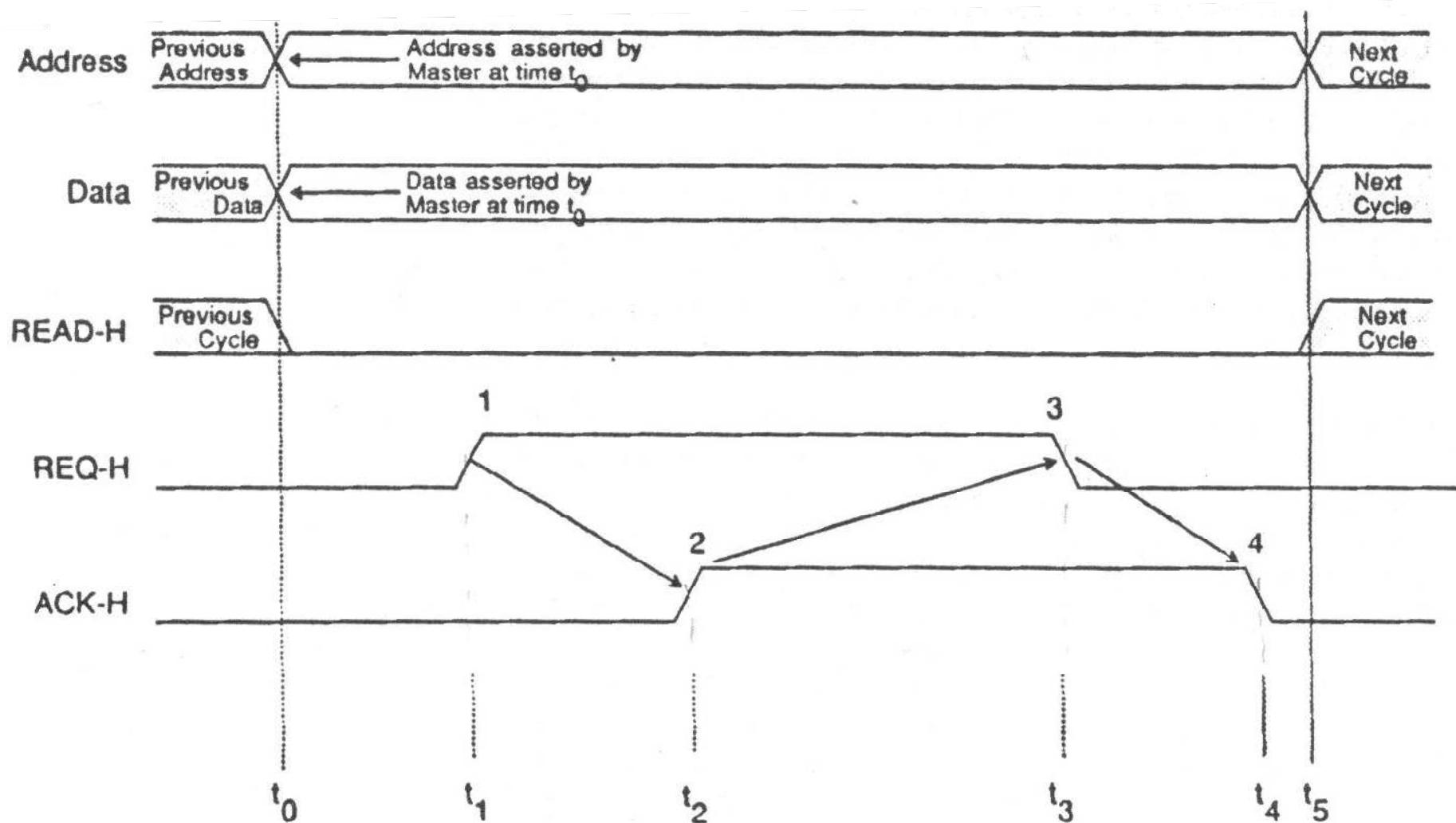
- Aszinkron **handshaking** = „**2x-es kézfogás**”
- Ebben az esetben a Master-Slave modulok **nem** közös órajelet (SysCLK-t) használnak: azaz ha az egyik egység végez, elindít egy másik tranzakciót. A Master, mint (commander) kezdeményező, aktiválja a megfelelő vezetékeket. A Slave (responder) válaszol. A Slave címének azonosítására egy külön egység szolgál.
- **Vezérlőjelekkel** zajlik a kommunikáció: írási / olvasási tranzakciókat különböztetünk meg. Ilyen vezérlőjelek:
  - **READ**, (ha pl. READ-H = Master olvas a Slave-től, ha READ-L= Master ír a Slave-nek, vagyis a Slave olvas.),
  - **REQUEST** (REQ)
  - **ACKNOWLEDGE** (ACK)
- **Előnye:** egyszerűen felépíthető, nem kell (közös) órajel, gyors átvitelt biztosít (modulok sebességétől függően)
- **Hátránya:** nagy belső késleltetések (skew-propagation time) miatt csak korlátozott hosszúságú buszok /vezetékek használhatóak.

# Aszinkron Handshaking protokoll (kétszeres „kézfogás”)

- A buszrendszer moduljai nem közös órajellel működnek, hanem **vezérlő jelek** segítségével. (READ-H, REQ, ACK)
- Háromféle buszvonalat ismerünk: *cím-, adat- és vezérlő-* buszt. A Master által *címvonalra* rakott cím (kezdeményezés) egyértelműen meghatározza a tranzakció célállomását. Meghatározott idő áll rendelkezésre, hogy a slave modulok összehasonlítsák saját címükkel a célcímet. Ha megegyezik, akkor válaszol a master-nek, és a tranzakciót a *vezérlővonalak* megfelelő beállításaival szabályozza. A vezérlővonalakat tehát a master-slave közötti kommunikáció szinkronizálására használjuk. Ezt nevezzük **handshaking** protokollnak.
- Címvonalak csoportjából (Address), adatvonalak csoportjából (Data), és három vezérlőjelből (READ-H, REQ-H, ACK-H) áll. Ha READ-H magas, akkor a Master olvas a Slave-ről, ha alacsony, akkor ír a Slave-re. A REQ (kérés) és ACK (nyugtázás) vezérlőjelek határozzák meg az események időzítését és pontos sorrendjét.

# Handshaking – írási ciklus

WRITE (READ L) ciklus: „a Master modul ír a Slave-nek”

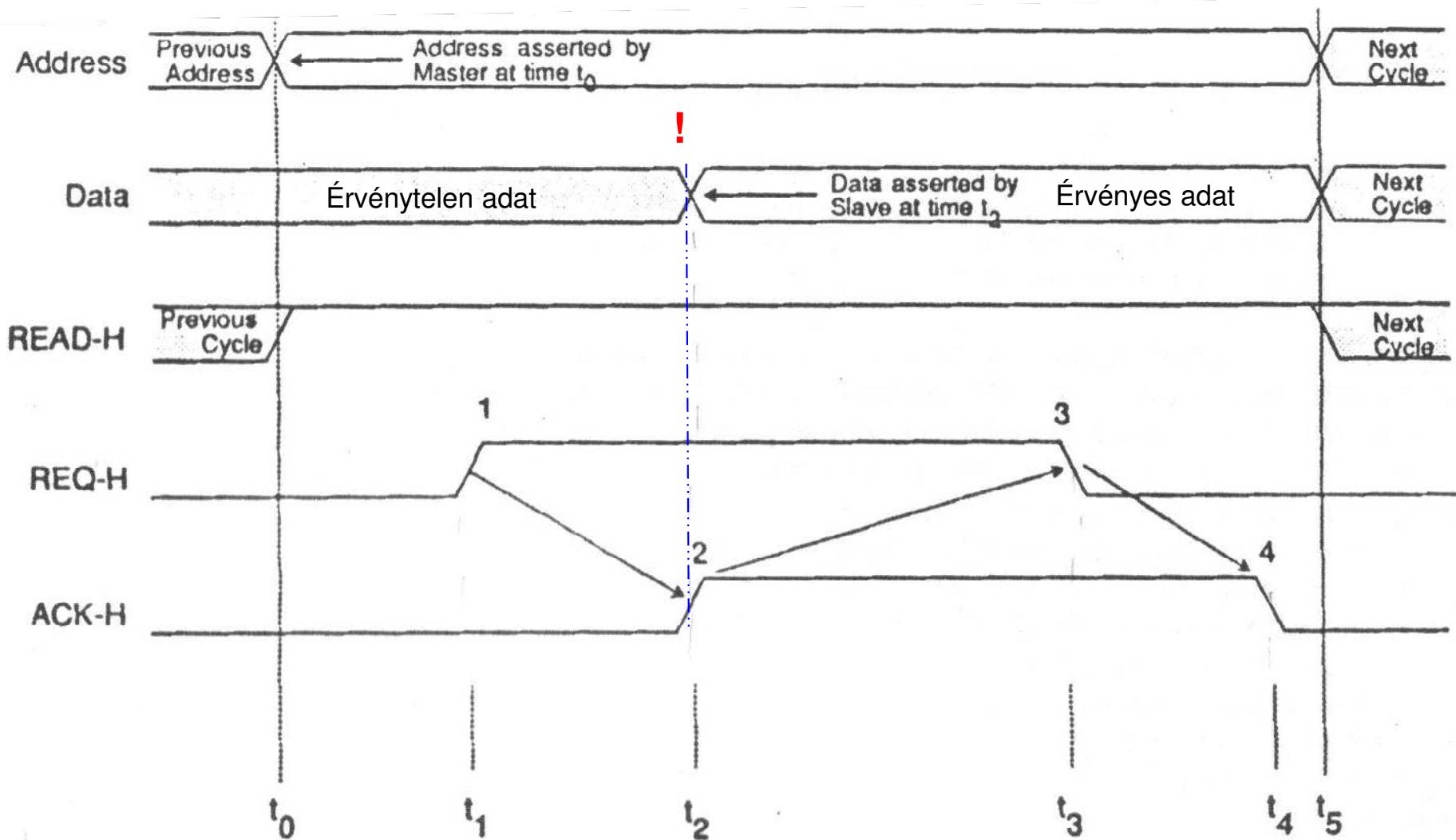


# Írási (Write) ciklus lépései:

- $t_0$ -ban a master (amely az aktuális arbitráció után már megkapta a busz vezérlését) megadja a kívánt slave címét.
- véges idő kell hogy a jel a slave modulokhoz érjen, azok dekódolják a címet, ezért a master vár bizonyos ideig mielőtt beállítja a request vonalat  $t_1$ -ben. (*skew time*= a slavekhez elsőként ill. utolsóként érkező címek közötti időkülönbség  $\Delta t=t_1-t_0$ )
- ezt a request jelet minden slave veszi ugyan, de csak az fog válaszolni, akinek a címe megegyezett a master által kért címmel.
- amikor a slave megkapta az adatokat a mastertől, nyugtázza  $t_2$ -ben.
- a master megkapja a nyugtát, így tudja, hogy az átvitel megtörtént, ezért felszabadítja (alacsony állapotba helyezi) a request vonalat  $t_3$ -ban.
- ezt (a request felszabadítását) érzékeli a slave, és felszabadítja a nyugtázó vonalat  $t_4$ -ben
- a master a címvonalat tartja még egy bizonyos ideig ( $t_5$ -ig) a request vonal felengedése után is, a cím esetleges megváltozása miatt (amelyet a Slavek dekódolnak).

# Handshaking – olvasási ciklus

**READ (READ H) ciklus:** „a Master olvas a Slave-től”



# Olvasási (Read) ciklus lépései:

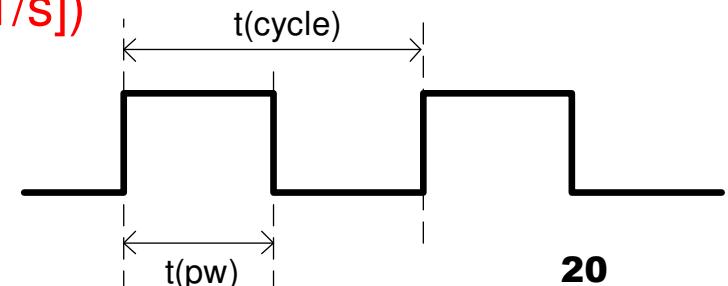
- Nagyon hasonlít az írási folyamathoz, de abban különbözik, hogy a READ-H jel magas szinten van, és az adatvonalat a slave állítja be. Ez jelenti az olvasást.
- $t_0$  hasonlóan történik, a master (amely az arbitráció után már megkapta a busz vezérlését) megadja a kívánt slave címét
- véges idő kell hogy a jel a slave modulokhoz érjen, azok dekódolják a címet, ezért a master vár bizonyos ideig mielőtt beállítja a request vonalat  $t_1$ -ben. (*skew time* = a slavekhez elsőként ill. utolsóként érkező címek közötti időkülönbség  $\Delta t=t_1-t_0$ ). Tehát  $t_1$ -ben a master a request vonal beállításával a megcímzett slave-től adatot kér, olvasni szeretne
- $t_2$ -ben veszi a kérést a slave, nyugtázza és beállítja magas szintre a nyugtázó vonalat.
- $t_3$ -ban a master megkapja az adatot a slave-től, felszabadítja a request vonalat.
- $t_4$ -ben érzékeli a slave, hogy a master felszabadította a requestet, ezért így ő is felszabadítja a nyugtázó vonalat.
- végül a master felszabadítja a címvonalat ( $t_5$ -ben)

## 2.) Szinkron busz protokollok

# Szinkron busz protokollok



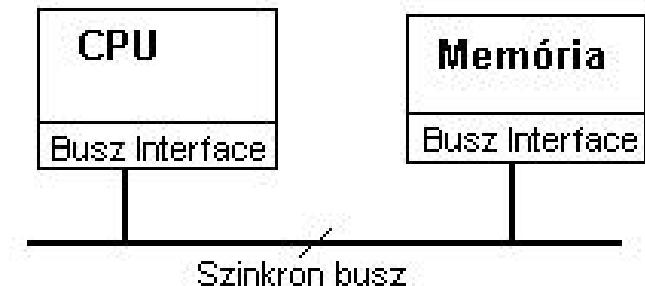
- Ebben az esetben a Master-Slave modulok **közös órajelet** (CLK-t): használnak. Itt a Master, mint Commander (kezdeményező), a Slave pedig (Responder) válaszol.
- Előnye:* Tehát a műveleti időt az órajelciklus határozza meg. Mivel nincs szükség párbeszédre (pl. handshaking), gyorsabb lesz az aszinkron működésénél.
- Hátránya:* Az órajelet mindenkor leglassabb (legtávolabb lévő) egységhez kell igazítani.
- A digitális áramkörökben az események történésének *sorrendje kritikus* (megfelelő időzítés kell  $\Rightarrow$  órajel vezérléssel)
- Óra:** impulzusok sorozatát bocsátja ki, pontosan meghatározott szélességgel [ $t(pw)$ ], és időintervallummal.
- Ciklus-idő** (clock-cycle): két egymást követő pulzus élei közötti időintervallum [ $t(cycle)$ ].
  - Példa: Órajel Frekvencia:  $f=100\ 000\ 000\ [\text{Hz}]$  ( $[1/\text{s}]$ )
  - Ekkor  $T=1/f=1 / 100\ 000\ 000 = 10\ [\text{ns}]$
- Kristály-oszcillátor szolgáltatja ált. az órajelet (lásd fenti képen CLK=16 MHz)



# Szinkron írási / olvasási ciklus

## • Írási ciklus (pl. CPU → MEM)

időlépések			
n	n+1	n+2	n+3
Commander arbitrációja	Kérés inicializálás	Válasz Commandernek	Responder döntése
CPU busz kérése	CPU adatot küld; Memóriahoz adat érkezi	memória dönt az adat elfogadásáról	memória interface nyugtát küld a CPU interface-nek



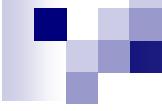
## Szinkron adatátvitel 4 fő lépése:

1. Commander arbitrációja, kiválasztása
2. átvitel megkezdése (de még nem fogad)
3. válasz a kérésre, döntéshozás
4. nyugta, Responder veszi az adatot

## • Olvasási ciklus (pl. CPU ← MEM)

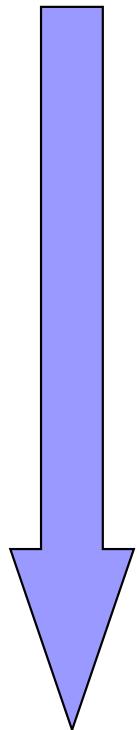
időlépések							
n	n+1	n+2	n+3	m	m+1	m+2	m+3
Commander arbitrációja	Kérés inicializálás	Válasz Commandernek a döntésről	Responder döntése	Commander arbitrációja	Kérés inicializálás	Válasz Commandernek a döntésről	Responder döntése
CPU busz kérése	CPU olvasási kérést küld; a memória interfész adat érkezik	Memória interface dönt az olvasási igényről	Memória interface nyugtát küld, hogy adni fog a CPU-nak	memória buszt kért	memória adatot küld; CPU interface-re adat érkezik	CPU dönt az adat fogadásáról	CPU interface nyugtát küld a memória interface-nek hogy fogadja az adatot





# Adatmozgatás: I/O kommunikációs technikák

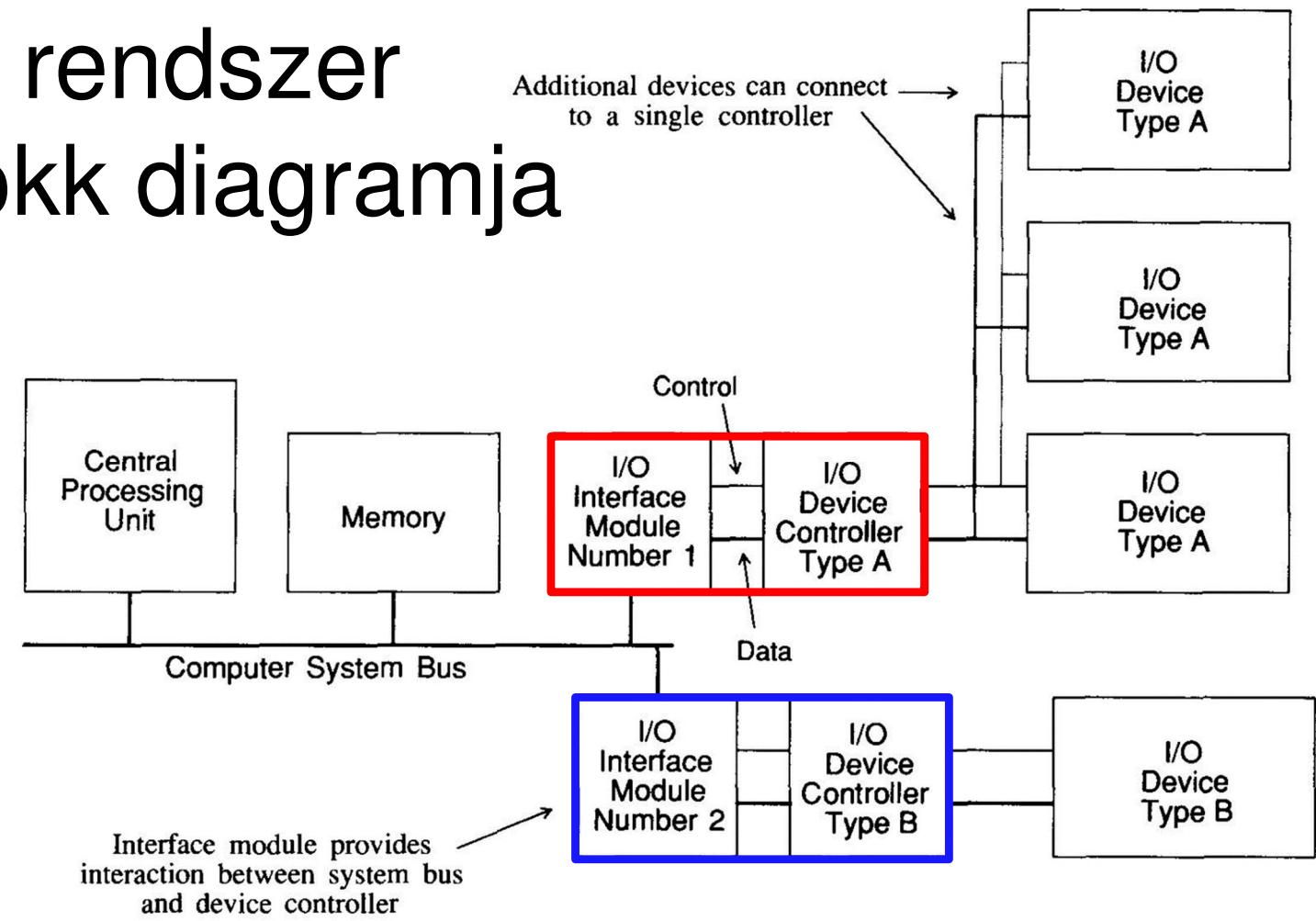
# I/O adatátvitel típusai:



- Programozott I/O átvitel („polling”)
- Megszakításos (interrupt) átvitel
- Direct Memory Access (DMA) -  
Közvetlen memória-hozzáféréssel  
rendelkező átvitel
- IOP - I/O Processzoros átvitel

CÉL: CPU  
tehermentesítése!

# I/O Interfész rendszer általános blokk diagramja



Két fő részre osztható: 1. **rendszerbusz** (CPU, memória) 2. **I/O eszközvezérlők** a különböző típusú I/O eszközökkel tartják a kapcsolatot. A rendszerbuszt és az I/O eszközvezérlőket az I/O interfacek kapcsolják (illesztik) össze. Az eszközök gépi kódú utasításokkal (assembly) vezérelhetők. Egy új I/O eszközt (**A** v. **B** típusú) a megfelelő típusú eszközvezérlőkhöz kell kapcsolni.

# 1.) Programozott I/O átvitel (Polling)

- Legegyszerűbb technika
- De leginkább ez a módszer terheli a CPU-t (adatátvitel teljes ideje alatt) – lassú eszközök esetén
  - Teljes vezérlésért, adatmozgatásért felel
  - Pl. periféria állapotának ciklikus lekérdezés folyamatosan terheli
- Csak a CPU közbeiktatásával érheti el a periféria a memóriát
- Lehet Memory Mapped I/O: amikor a program és az I/O eszköz is ugyanazt a címtartományt használja (cím leképezés = „mappelés”)

## 2.) Megszakításos (interrupt) átvitel

- Megszakítással jelezhető a CPU-által az I/O eszköz számára az adatátviteli igény, illetve az adatátvitel befejeződése
- Megszakítás kérelem (**interrupt request**) I/O eszköz által – **IRQ** szintek
  - Megszakítási vektorok (maszkolható megszakítások – SW interrupt)
- Bővebben: Operációs rendszerek tárgyból

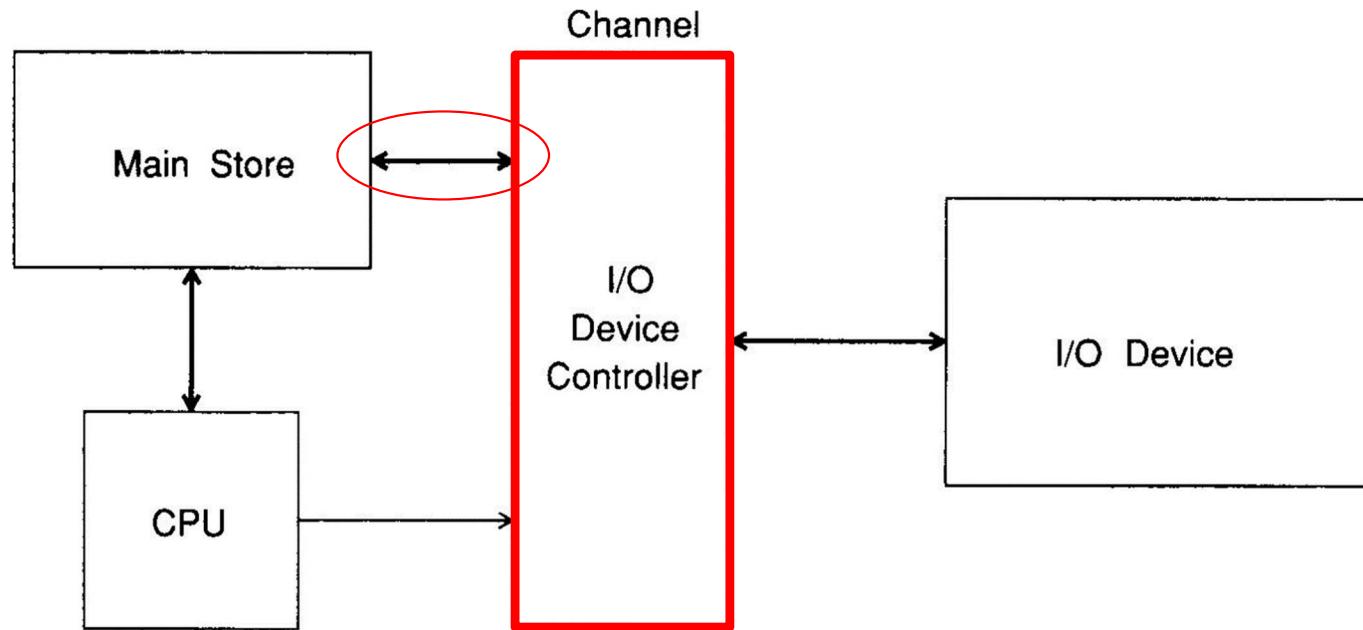
# 3.) Direct Memory Access (DMA)

- Közvetlen memória hozzáférés:  
az I/O eszköz/periféria és memória közötti adatátvitelt a processzortól függetlenül egy DMA-controller/vezérlő (eszközvezérlő) végzi el.
- Cél a CPU tehermentesítése a tranzakció idejére
- CPU feladatai csupán:
  - az átvitel előkészítése (kezdőcím és adat(ok) hossza, száma),
  - minimális vezérlés: eszköz állapot vizsgálata (busy)
  - és a befejezett művelet hibátlanságának ellenőrzése (megszakításos alapú is egyben)
- Gyors módszer

## 4.) I/O processzor (I/O csatornák)

- A cpu átadja az I/O műveletet és a végrehajtáshoz szükséges összes adatot ez intelligens eszközvezérlőnek = I/O (társ)processzornak, amely teljesen önállóan szabályozza a tranzakciót
- Főleg a mainframek-re jellemző módszer
- Rendkívül gyors
- I/O csatornák: eszköz sebessége szerinti osztályozás

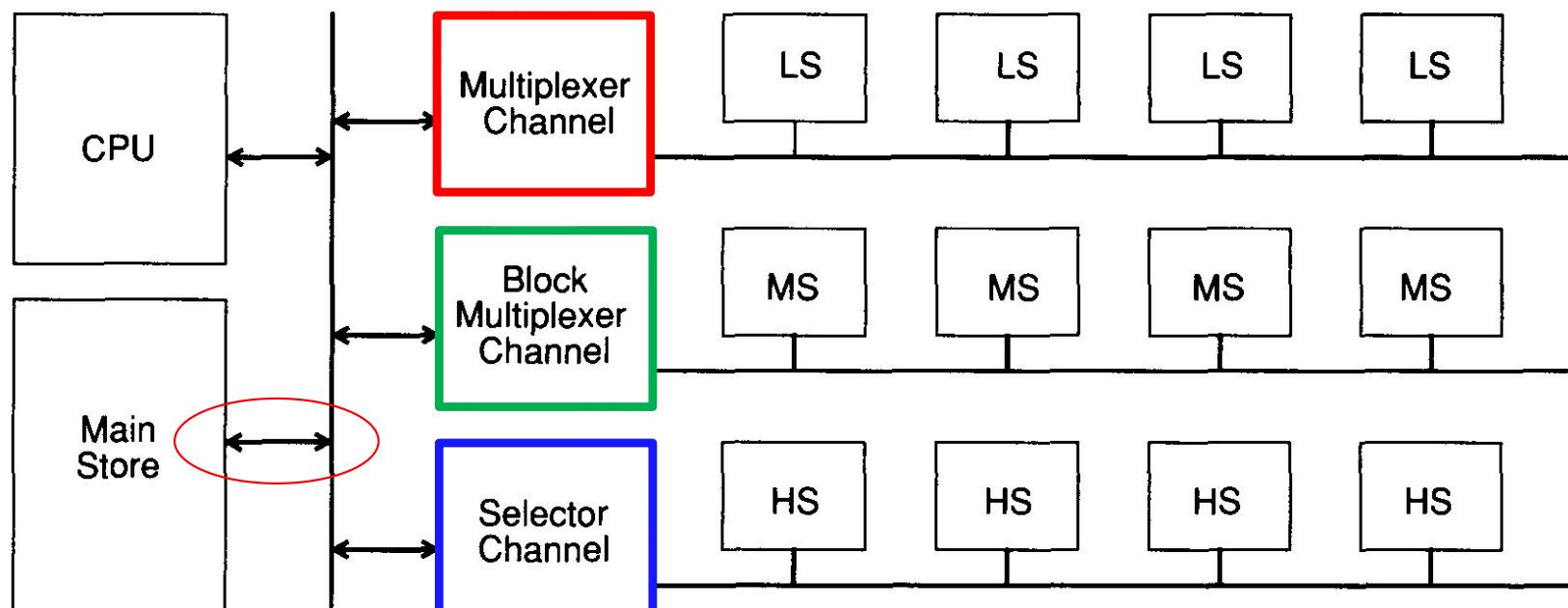
# a.) I/O channels – I/O csatornák



- A cpu csak az eszközvezérlőn keresztül (közvetetten) érheti el a perifériát.
- Channel = I/O Device Controller: általános célú processzáló elem.
- Channel feladata:
  - Konverzió
  - Adatmozgatás
  - Hibaellenőrzés és kezelés

# Példa: I/O channels – I/O csatornák

- I/O csatornák típusai (sebességük szerinti kategóriákat képeznek):
  - Multiplexer: LS (lassú)
  - Blokkos: MS (közepesen gyors)
  - Selector channel: HS (nagysebességű)

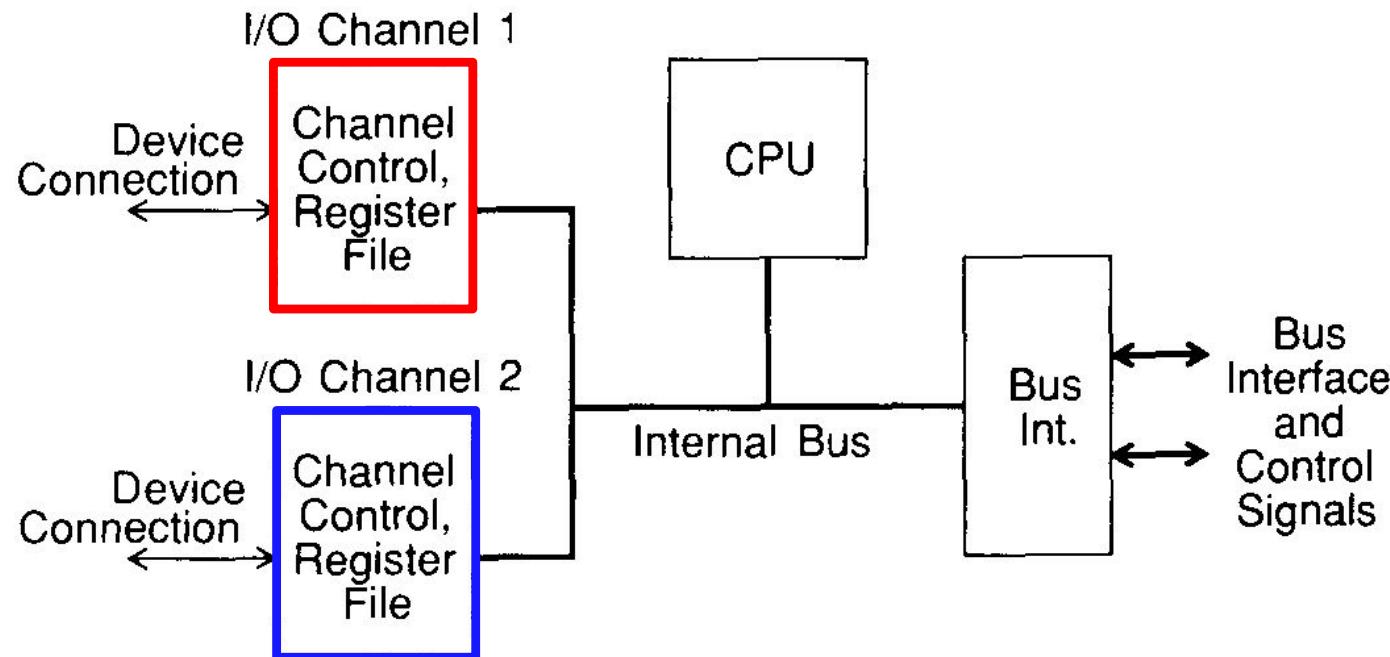


LS = Low Speed device

MS = Medium Speed device

HS = High Speed device

# b.) I/O Processzorok (IOP)



- IOP: Intelligens eszközvezérlők/processzorok
- Saját, dedikált funkciókkal rendelkeznek (vezérlést, és interfészét biztosít más rendszerekkel – különböző sebességen)
- Példa: SCSI rendszer is egy IOP

# PCI busz

# PCI busz

- **PCI= Peripherial Component Interconnect.** újszerű, a korábbiaktól önálló szabvány (ver 2.1, 2.2 és 2.3) volt: 33MHz-es órajel támogatás. Többprocesszoros rendszereket is támogat. **Párhuzamos sín!**
- Kapcsolatot a *processzor* és a PCI sín között egy *PCI HOST BRIDGE* biztosítja. Támogatja a többprocesszoros rendszereket, kompatibilis az ISA, EISA, MCA régebbi rendszerekkel. A PCI csatlakozóhelyekre speciális intelligens kártyák helyezhetők, amelyek képesek önálló adatátvitelt végrehajtani a processzor tehermentesítése céljából, így gyorsabb működés érhető el. (3.3 – 5V szabvány).
- Csatlakoztatható eszközök: SCSI-, hálózati-, hang-, videó-kártya./ Konfigurálása szoftveres úton, a BIOS-on keresztül történik. / Arbitrációs mechanizmust és **szinkron** protokollt használ.

# PCI buszrendszer tulajdonságai:

- A PCI 32 bites **multiplexált** címadat vonalat alkalmaz
  - Maximális átviteli sebessége (4-es „burst-onként” - löketszerűen) 133Mbyte/sec (= 4 byte\*33.3MHz).
- (64 bites változata is létezik, főként szerverekben alkalmazzák. Jele. PCI-X!).
  - Ekkor a maximális átviteli sebessége 266Mbyte/sec (8byte\*33.3MHz)
- Régebbi alaplapokon az ISA ill. AGP bővítőhelyek mellett általában 3-4 PCI slot is található volt.
- A külső környezeti zavarok, zaj elkerülése végett, a PCI elemeket rövid úton kell összekötni, így a PCI jelek egy oldalán vannak kivezetve (PCI Speedway), sok földelést használva.
- Saját POST (Power On Self Test) önellenőrző kóddal is rendelkezik, a hibák felderítése végett, ami a számítógép bekapcsolásakor inicializálódik.

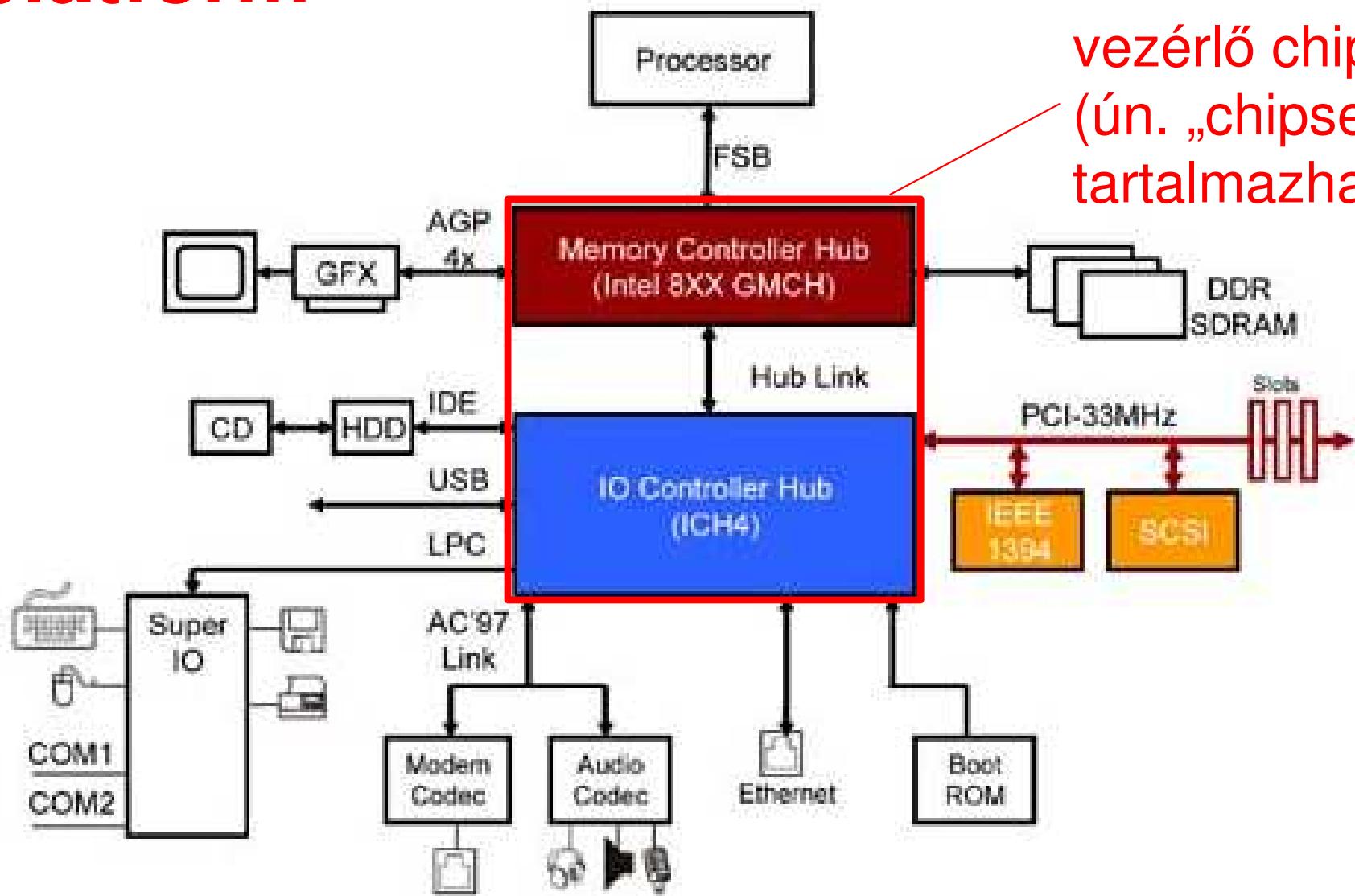


# PCI-X

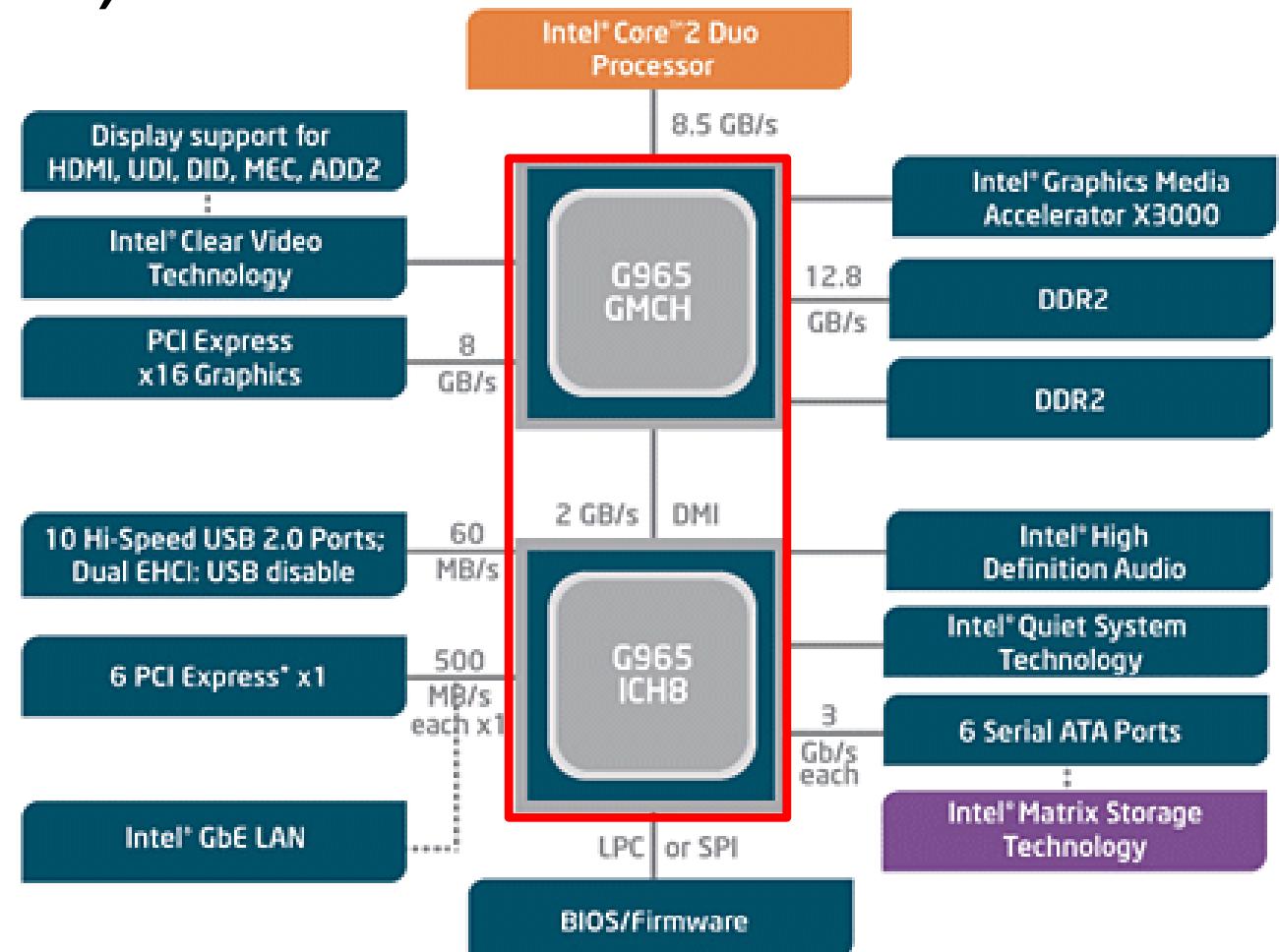
- PCI módosított változata
- **PCI-X** mint **extended**, bővített 64 bites
  - PCI-64bit/66.6MHz max 532 MByte/s
  - PCI-64bit/133.3MHz max 1064 Mbyte/s
  - PCI-X 1.0 & PCI-X 2.0
- Főként szerver alaplapokon található:
  - nagyteljesítményű kártyák integrálása

# Példa: 33MHz-es PCI busz alapú platform

PCI híd: 1 vagy 2 vezérlő chip-et (ún. „chipset”-et) tartalmazhat.

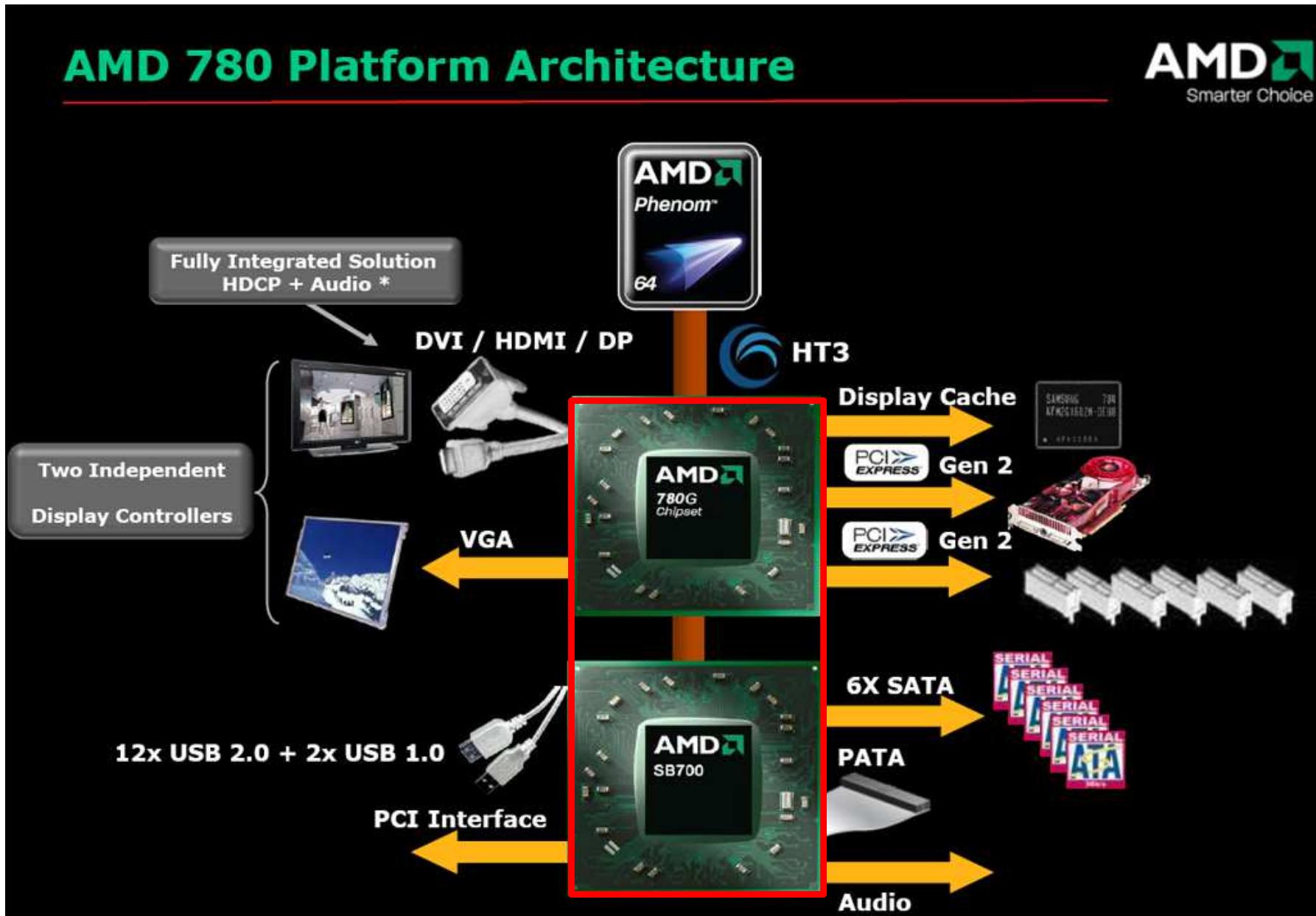


# Példa: Intel chipset (Intel CPU – É-D híd külön)



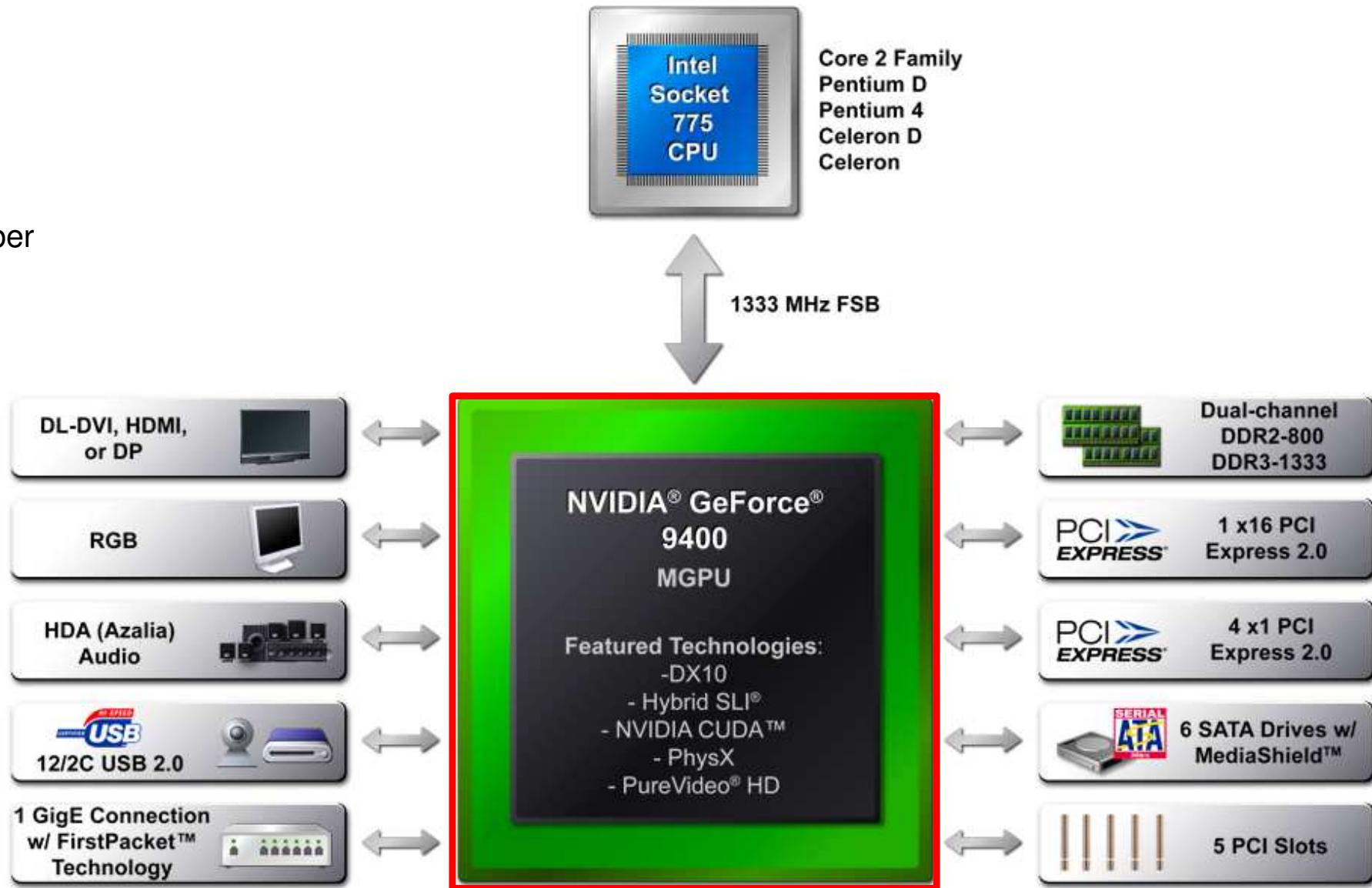
Integrált grafikus vezérlővel:  
Graphics Media Accelerator 3000 (GMA 3000)

# Pl: Újabb generációs AMD 780G



# PI: NVIDIA GF9400 MGPU Intel CPU-khoz (MGPU – Egyetlen chipen É-D híd!)

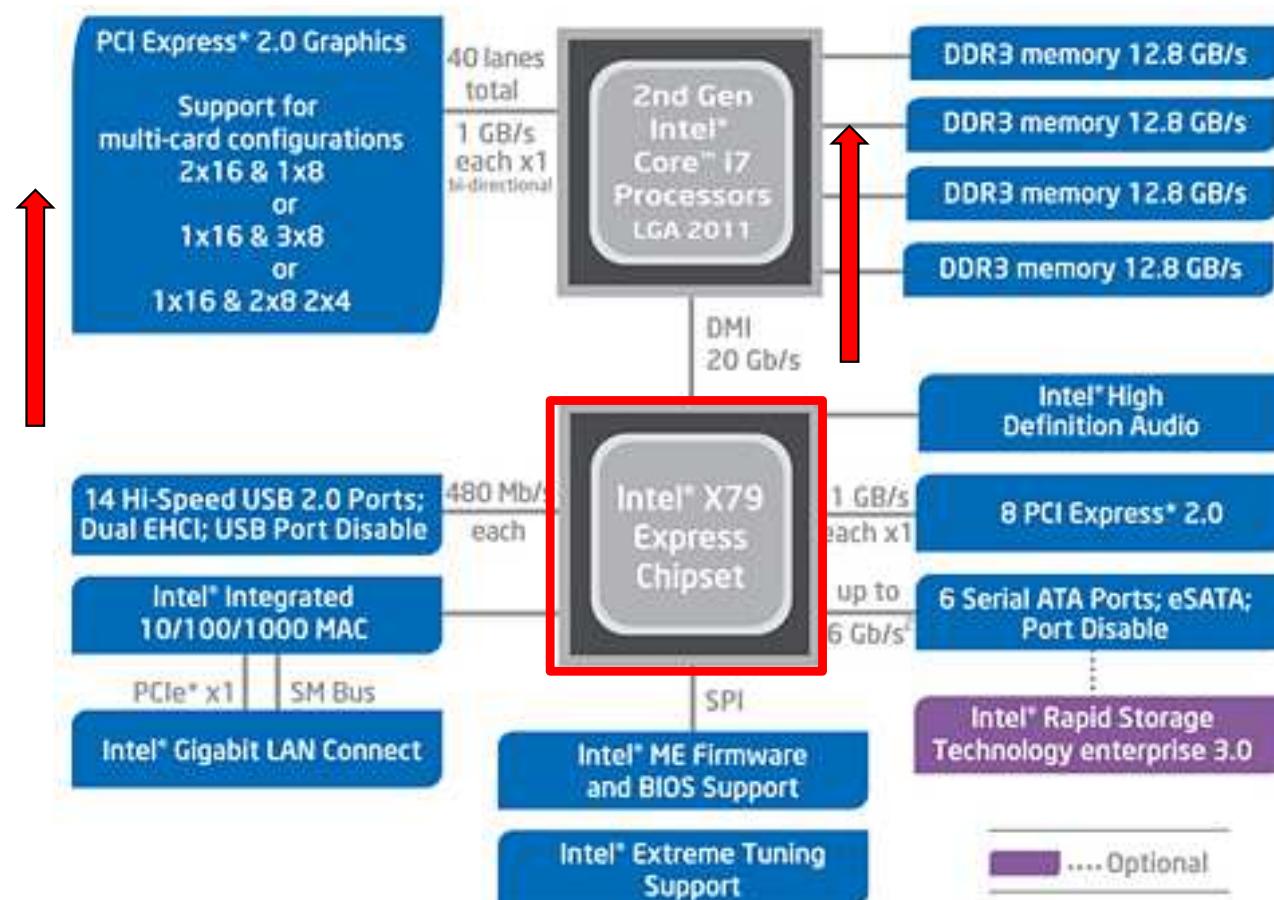
\*2008.  
november



# Pl: Intel Sandy Bridge-E (2011)

- Nincs már északi híd: PCI-E ill., a DDR memória vezérlése is a CPU-ba integrált feladat), csak déli híd maradt (X79)

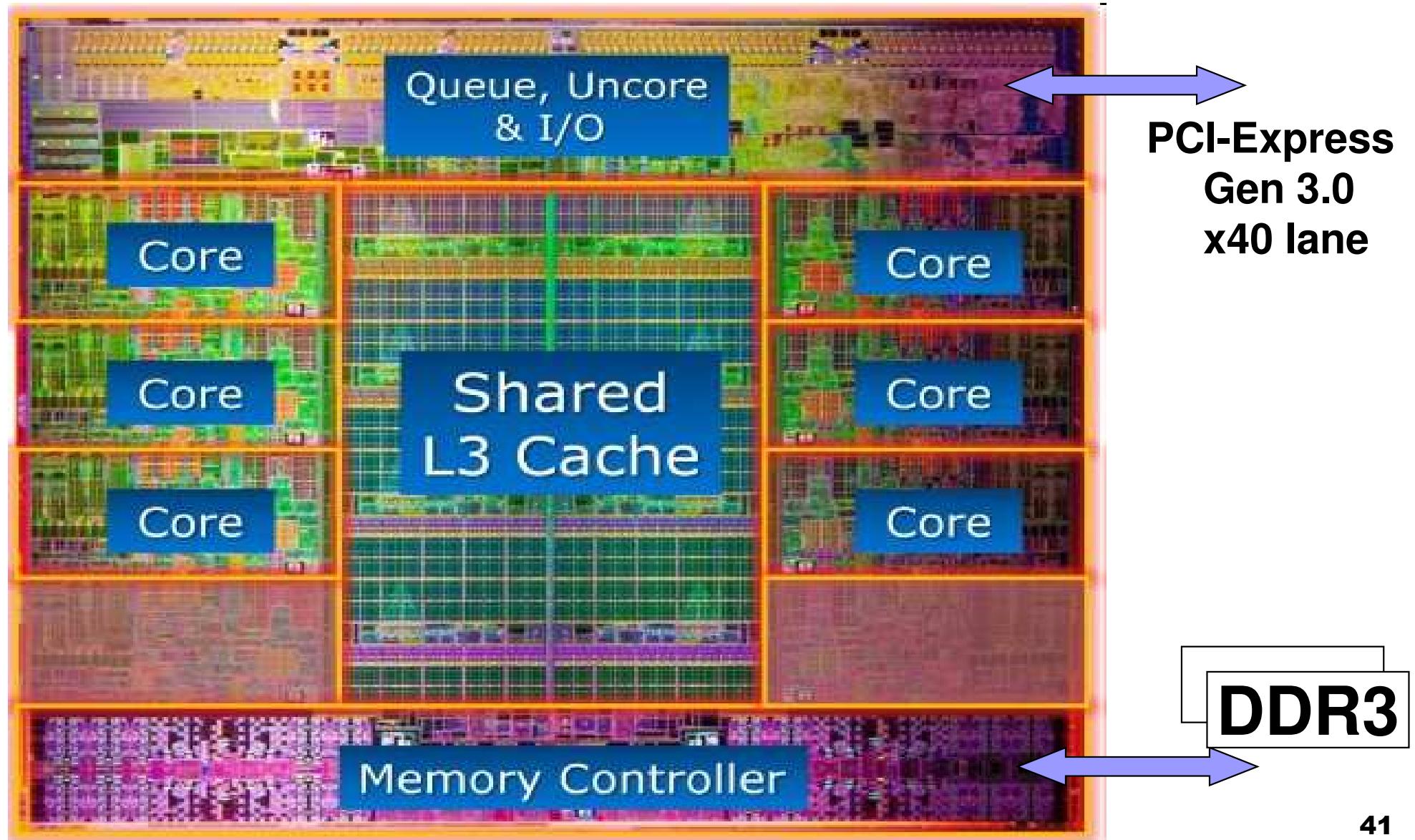
\*2011.  
november



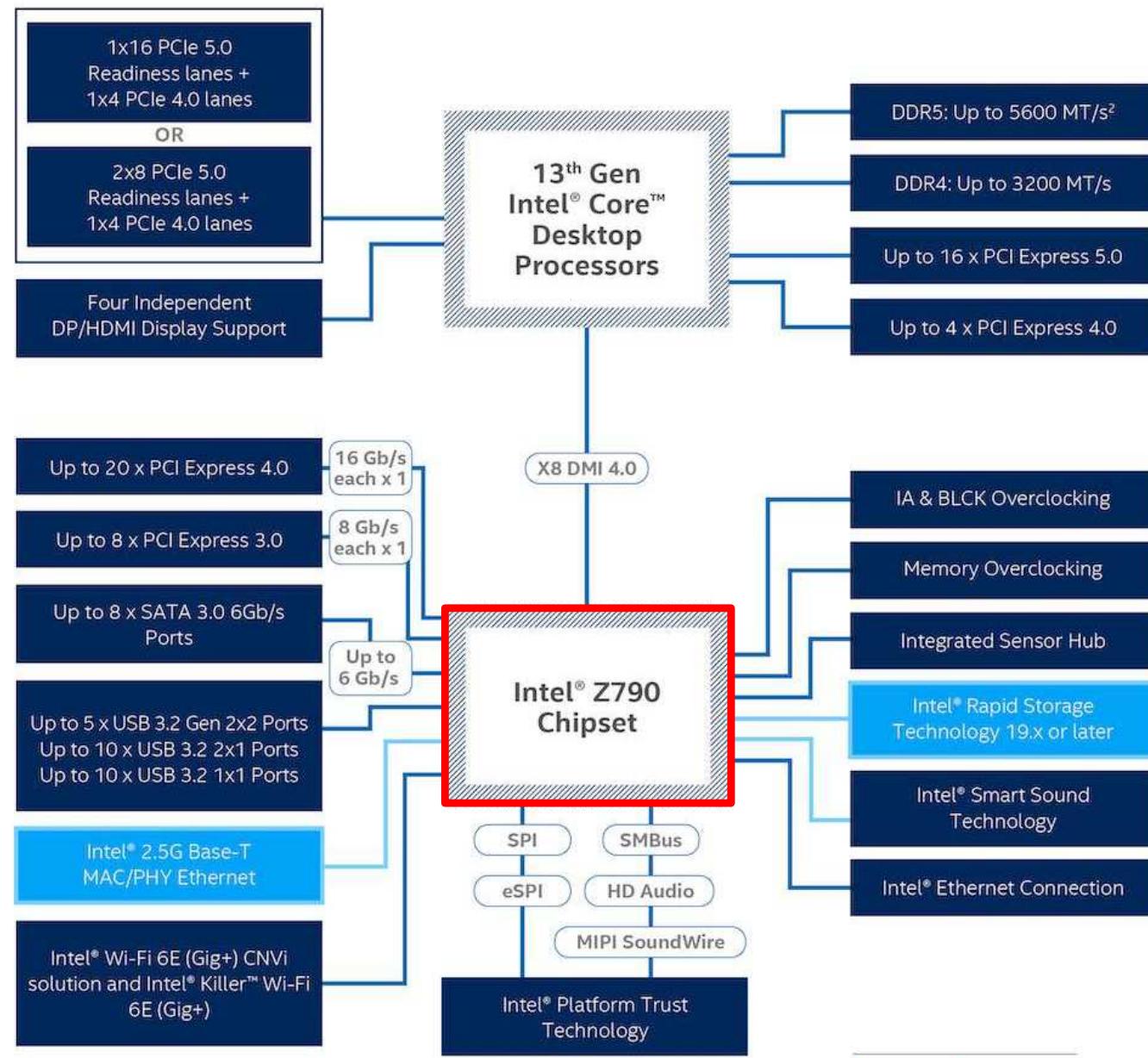
<sup>1</sup>Theoretical maximum bandwidth

<sup>2</sup>All SATA ports capable of 3 Gb/s. 2 ports capable of 6 Gb/s.

# Intel Core i7-3960X – Sandy Bridge-E



# Pl: Intel Raptor Lake (13.gen – 2023)



# PCI busz fontosabb jelei #1

A csatlakozó 188 lábú, oldalanként 94-94 lábbal. Sok GND található a zavarások elkerülése végett. A tápról jön 12V, 5V, 3.3V-os jel is.

- **CLK**: (Clock) PCI busz órajele (0-20-33MHz)
- **AD0-AD31**: (AD: Address/Data) multiplexált cím/adat vezetékek 32 bites üzemmódban (ahol AD00-AD07 byte-címzés esetén az LSB, míg AD24-AD31 byte-címzés esetén az MSB-t jelöli).
- **AD32-AD63**: 64 bites üzemmódban
- **IRDY**: (Initiator Ready) adatok olvasásakor (negált jel)
- **TRDY**: (Target Ready): adat írásakor (negált jel)
- **DEVSEL**: (Device Select) ez egy nyugtajel, a target ezzel nyugtázza, hogy a címet dekódolta
- **IDSEL**: a Chip Selectnek felel meg, adatírás vagy konfiguráció során lehet hozzáférni a chip-hez
- **STOP**: az adatforgalom megszakítását jelzi a target-nek
- **FRAME**: adatátviteli ciklus jelzése (negált jel)

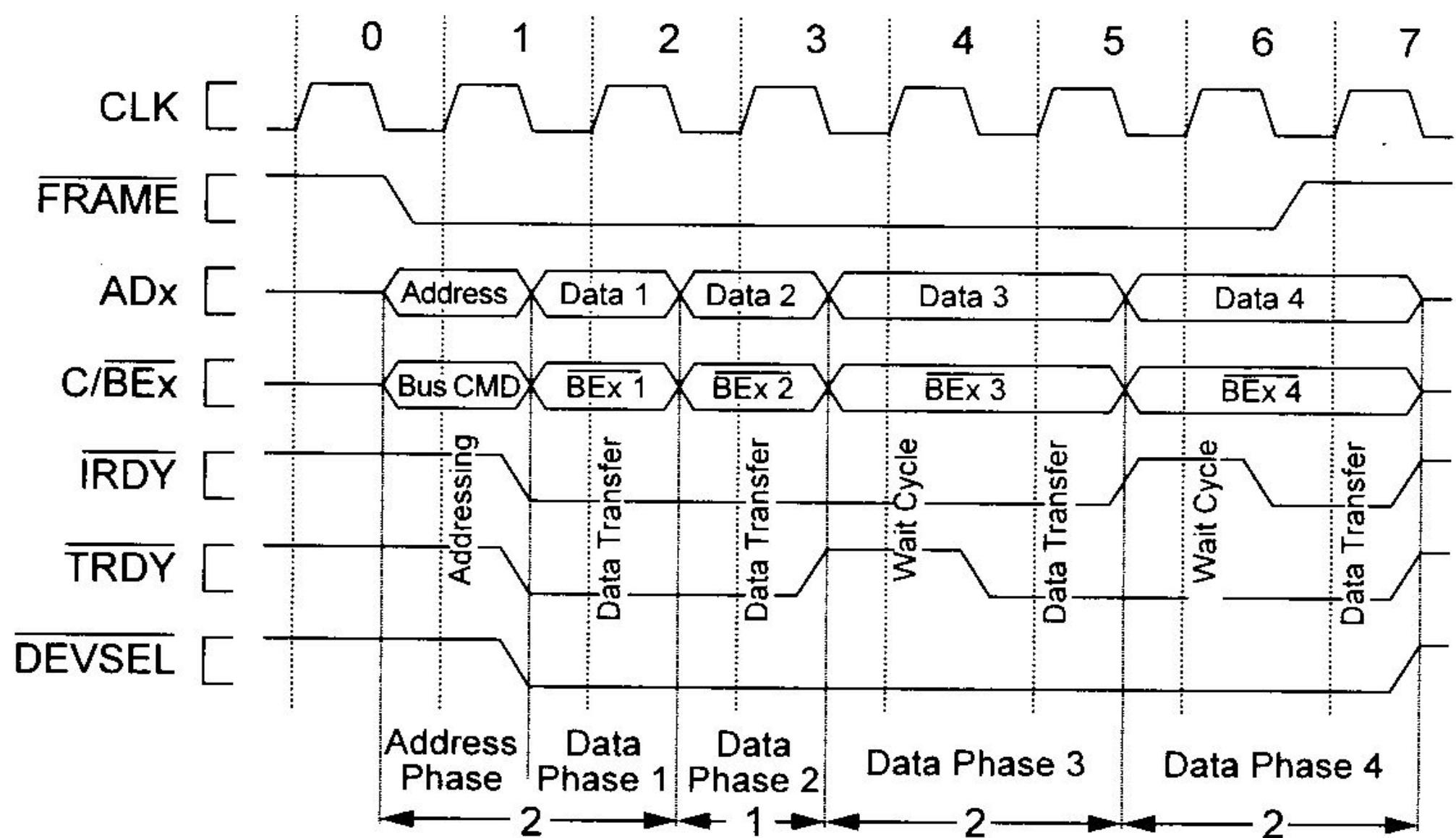
# PCI busz fontosabb jelei #2

- **PAR:** (Parity) adatok és címek paritás ellenőrzése
- **C/BE0 – C/BE3:** (Command & Byte Enable) egy-egy jel egy byte-ot foglal magába. Megmutatja, hogy melyik byte tartalmaz érvényes adatot, ill. írunk, vagy olvasunk-e.
- **PERR – SERR:** (Parity error ill System error) hibajelek
- **INTA – INTD:** megszakításjelek (felfutó élre vezéreltek)
- **REQ:** (Request) sínhozzárendelés a kéréshez
- **GNT:** (Grant) sínhozzárendelés engedélyezéshez
- **TCK, TDI, TDO, TRST:** (Test Clock, Test Data in, Test Data Out, Test Reset) PCI sín tesztelésének jelei (JTAG-hez)
- **RST:** (Reset) regiszterek tartalmának törlése, és a PCI jeleinek kiinduló helyzetbe állítása

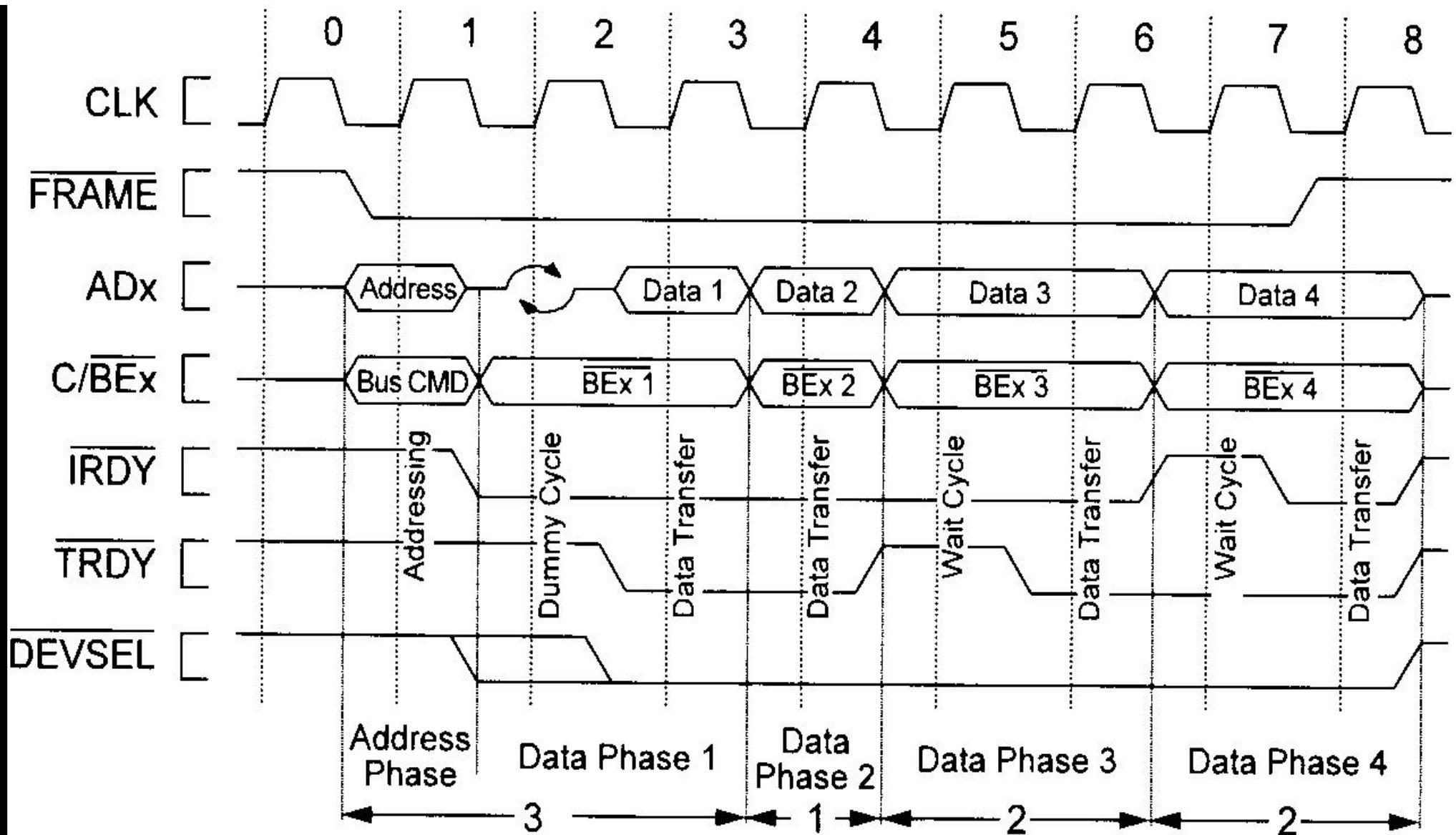
# PCI írási/olvasási tranzakciók

- **Burst = „löketszerűen”** egyszerre több adatot szeretnénk kiolvasni/írni. 1 cím kiadása után tipikusan **4 adat jön (4-es burst)**. A címzési fázis utáni első órajelciklusban az átvitel irányát módosítani kell, a közös multiplexált ADDR/DATA busz miatt. Olvasásnál ezért van szükség egy üres ciklusra (*Dummy*). Például: olvasásnál 3-x-x-x, vagy írásnál 2-x-x-x
- Miután az arbitráció során az egység sikeresen megkapta a vezérlést, a FRAME jellet inicializáljuk az adatátviteli folyamatot. Olvasásnál a címzés egy ciklus idejű, miután egy üres ciklus jön, majd az első adat megérkezéséhez szükséges ciklus (Esetünkben az első fázis 3 ciklus idejű: címzés + üres ciklus(ok)+TRDY adat). Az IRDY jel majd a TRDY jel alacsony jelszintre váltása után kezdődhet meg az adatok továbbítása löketszerűen, egymás után x-x-x ciklusonként.
- Olvasásnál: pl. 3-1-2-2= először 3 órajelciklust vár az első adatig, majd utána 1-2-2 ciklusonként jönnek az adatok (közben *wait* ciklusok lehetnek!)
- Írásnál nincs üres (dummy) ciklus, a cím kiadása után egyből mennek az adatok 2-1-2-2 ciklusonként, az **optimum 2-1-1-1**)

# PCI írásí ciklus (pl. 2-1-2-2 burst)



# PCI olvasási ciklus (pl: 3-1-2-2 burst)



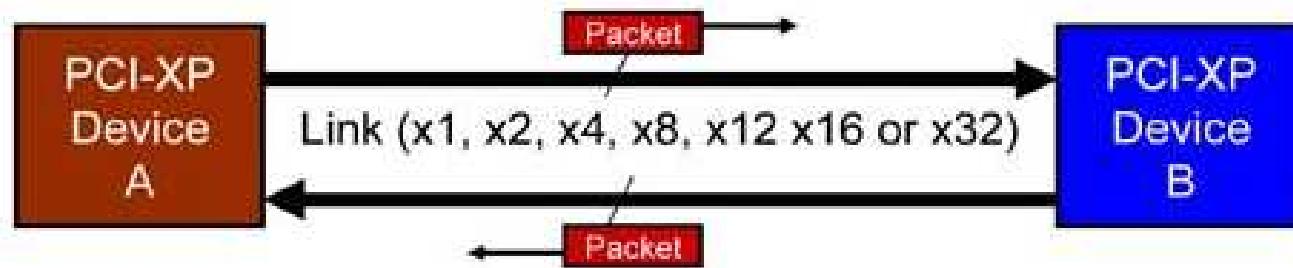
# PCI-Express busz

Kapcsolódó háttéranyag:

- <https://pcisig.com/faq>

# PCI-Express Busz

- Nagyteljesítményű, nagysebességű, P2P: pont-pont kapcsolati szinkron protokoll, soros buszrendszer. Szabványosításért PCI-SIG felel.
- **Duális simplex** (két-egyirányú) kommunikációt biztosít (ma már full-duplex)
  - **Link:** ×1, ×2, ×4, ×8, ×12, ×16 vagy akár ×32 lane-ből áll



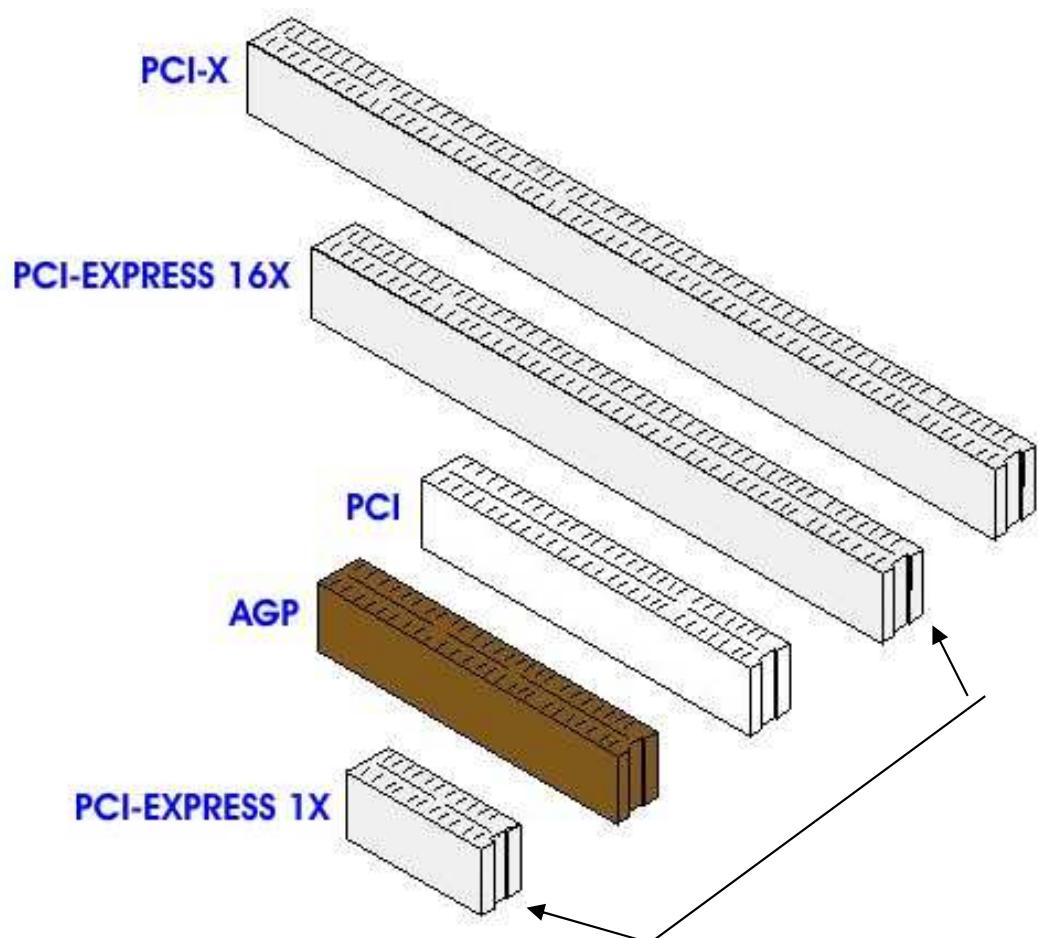
- **Lane (sáv):** jelpárok a két irányban
- Hivatalos Jelölése: PCI-E, PCIe ! ( $\neq$  PCI-X !)
- Cél: leváltani a PCI, PCI-X és AGP buszokat

# PCI Express tulajdonságai

- (Míg a korábbi PCI-nál az eszközök osztottak a sínen), addig az új PCI-Expressnél egy *switchen/hub-on* keresztül érik el (**P2P topológia**) a sínt:
  - azaz minden eszköz úgy látja, mintha saját külön sínnel rendelkezne! Soros kapcsolat!
- A **switch** gondoskodik a P2P kapcsolatok létrehozásáról és a vezérli a sín adatforgalmát. A switch és az eszközök közötti kapcsolatokat **link**-nek nevezik.
- Egy PCIe link duál-szimplex (ma már full-duplex): azaz az adó és a vevő két egyirányú csatornán keresztül forgalmaz. minden link egy vagy több **lane**-ből ( $\times n$  sáv) állhat.
- Egy **lane** egy bájt egyidejű átvitelét teszi lehetővé (250MB/s = PCIe v1.0 szabványban!), ami a gyakorlatban maximálisan ~2,5 GB/s adatátviteli sebességet jelentett.
- A PCIe  $\times 1$ ,  $\times 2$ ,  $\times 4$ ,  $\times 8$ ,  $\times 12$ ,  $\times 16$  és  $\times 32$  lane-ből álló linkek létrehozását támogatja. A switch alkalmazása lehetővé teszi a rendelkezésre álló sávszélesség jobb kihasználását és az adatforgalom fontosság szerinti osztályozását
- Alacsony fogyasztás, illetve az energiatakarékossági funkciók támogatása. Támogatás: GPU, SSD, hálózati eszközök.

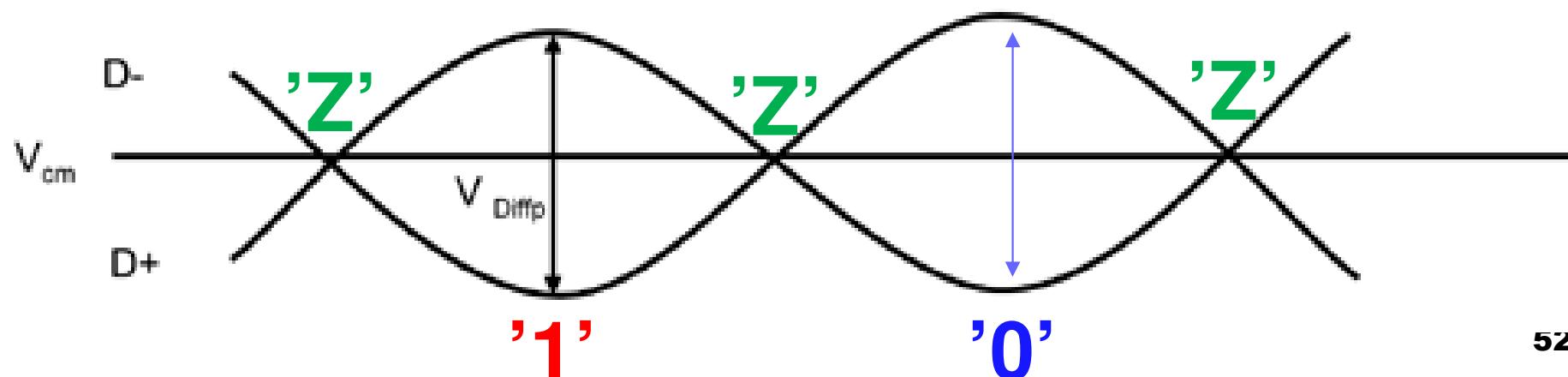
# Bővítőhelyek (slotok)

- PCI (32 bit)
- PCI-X (64 bit)
- **PCI-Express**
  - ×1
  - ×16
  - ×32
- AGP (Intel Accelerated Graphic Port)
  - ×1-×8

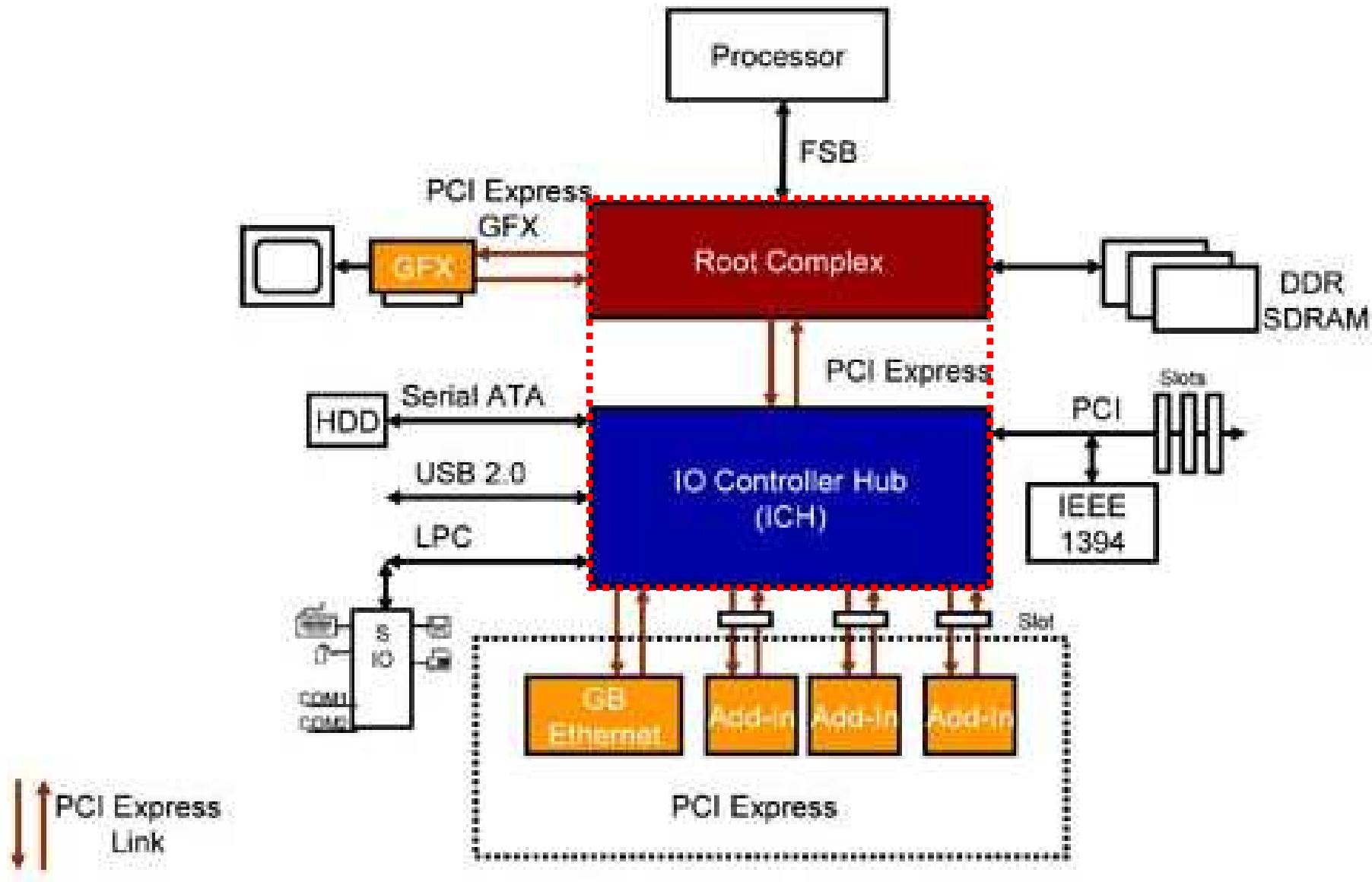


# Működése: Differenciális jel

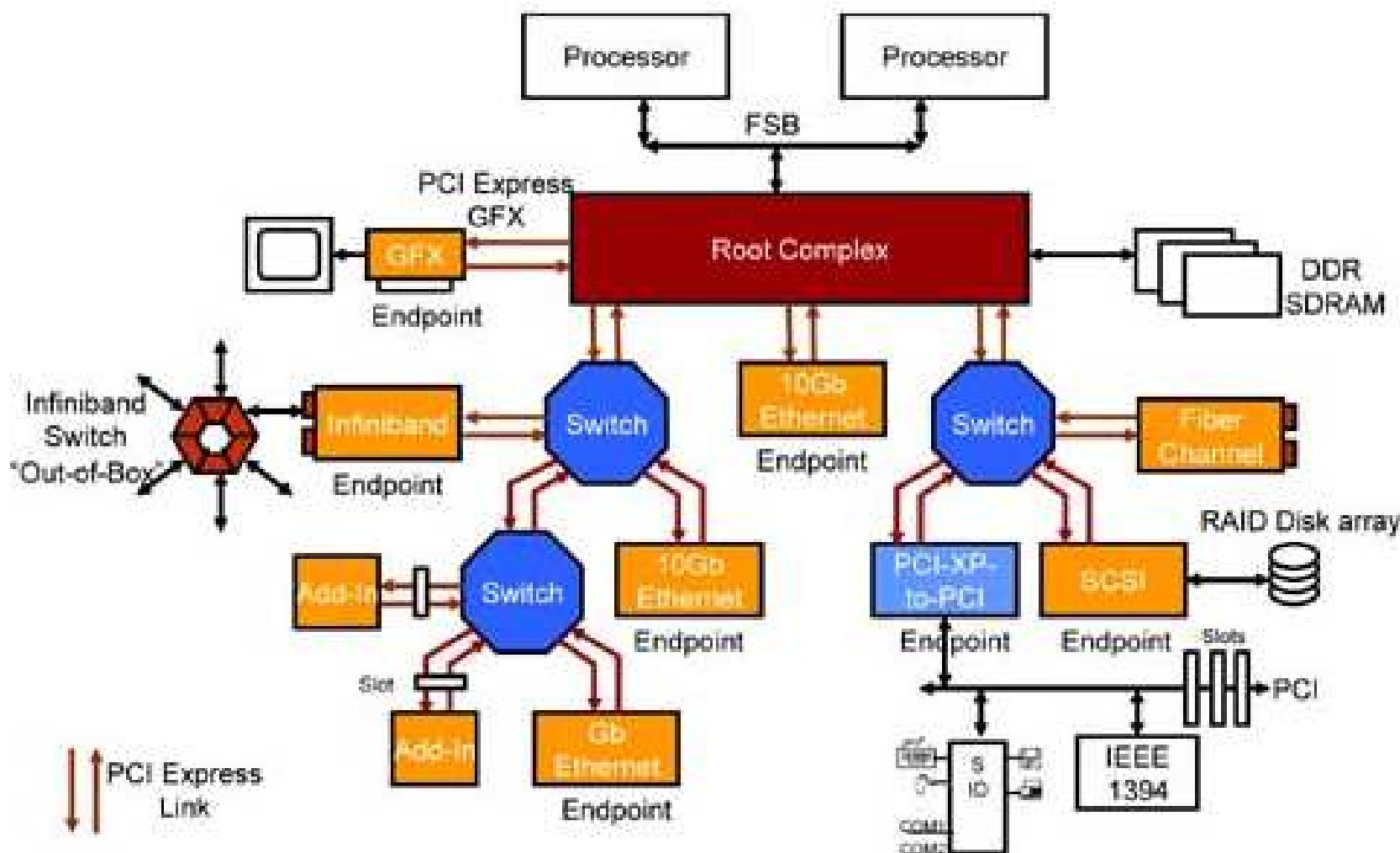
- Differenciális driver-ek és receiver-ek találhatók mindenegyes PCIe portnál.
- Ha *pozitív feszültség különbség* van a D+ és D- terminálisok között, akkor *logikai '1'*-et reprezentál.
- Ha *negatív feszültség differencia* van a D+ és D- között, akkor pedig *logikai '0'*-t mutat.
- Ha *nincs feszültség különbség* a D+ és D- között, az azt jelenti, hogy a driver nagy-impedanciás ún. *tri-state ('Z')* állapotban van, amely az eszköz (tétlen) állapotát, és a *line low-power* állapotát jelzi.
- PCI Express iel elektromos karakteristikája (VDS):



# Olcsó asztali gépes PCI-Express rendszer



# PCI-Express szerver alapú, multi-processzoros rendszer esetén



- **Switch:** több-portos eszköz, Link-ek kapcsolhatók hozzá

# PCI-Express v1.0 tulajdonságai (2004)

- Csomag (Packet) kapcsolt protokoll
- Sávszélességek és órajelek:
  - Data rate (bandwidth): 250 MB/sec/sáv/irány (x1 lane)
  - Max ~2.5 GB/sec/sáv/irány (~2.5 GHz)
  - 8b/10b kódolás (1 bytehoz, 2 kódbit)
  - **IO Transfer rate=IO Bandwidth (IO sávszélesség)**  
Max. elméleti határ ~**2.5 GT/s** (x16: 16 sávon)  
**//GigaTransfer (GT): adatműveletek száma/s**
- Memória cím területek:
  - Memória
  - I/O címtérület
  - Konfigurációs cím (kibővítése a PCI esetén megismert 256 Byte-ról 4 Kbyte-ra)

# PCI-Express v1.0 összesített sávszélesség adatok

- IO Bandwidth (IO sávszélesség) Max 2.5 GB/s
- 8b/10b kódolás (osztva 10-el)
- n Lane: „x n” (szorozva)
- szorozva 2-vel (irányonként)

Table - PCI Express v1.0 Aggregate Throughput for Various Link Widths							
PCI Express Link Width	x1	x2	x4	x8	x12	x16	x32 (nem támog.)
Aggregate Bandwidth (GB/s)	0.5	1	2	4	6	8	16

# PCI-Express v2.0 tulajdonságai (2007)

- Csomag (Packet) kapcsolt protokoll
- **IO Sávszélességek (órajelek) duplájára növekedtek:**
  - Data rate (bandwidth): 500 MB/sec/sáv/irány (x1 lane)
  - Max 5 GB/sec/sáv/irány (5 GHz)
  - 8b/10b kódolás (20% overhead)
  - **IO bandwidth:**
    - ~**5 GT/s (x1 lane)**
    - Max. elméleti határ : ~80 GT/s (x16 lane)
  - Nagyobb fogyasztású (250-300W) eszközöket (pl. GPU) is támogat
  - Már **külső eszközöket** is támogat (10 m-es kábel)
  - Input-Output **Virtualization** (IOV): hatékonyabbá teszi az ugyanazon hardveren futó virtuális gépek működését azáltal, hogy segíti a PCIe -eszközök megosztását - kapcsolati seb. SW-szintű beállítása
  - PCIe v1.1-re visszafelé kompatibilis

# PCI-Express v3.0 tulajdonságai (2011)

- Data rate (bandwidth): ~1 GB/sec/sáv/irány (x1 lane)
- IO sávszélesség ismét duplázódott: **max ~8 GT/s** (128b/130b kódolás = 16 bytehoz 2 kódbit).
  - „PCIe 2.0 delivers 5GT/s but employed an 8b/10b encoding scheme which took 20 % overhead on the overall raw bit rate. By modifying the requirement for the 128/130b encoding scheme, PCIe 3.0's effectively delivers double PCIe 2.0 bandwidth: only 1.5 % overhead!”
  - 1 Gbit/sec/sáv/irány (x1 lane)
- Továbbfejlesztett jel és adat integritás:
  - receiver-transmitter,
  - PLL (phased-locked-loop)
  - csatorna részeken optimalizálás
- PCIe v.2.x-re visszafelé kompatibilis
- Bevezetése: 2011. januárjában jelent meg.

# PCI-Express v4.0 tulajdonságai (2016)

- Data rate (bandwidth): ~2 GB/sec/sáv/irány (x1 lane, max 16 lane)
- IO sávszélesség ismét duplázódott: max ~**16 GT/s** (8b/10b kódolás módosításával).
  - Továbbfejlesztett jel és adat integritás
  - Power optimizations (active / idle states)
  - Max power consumption: 375 W total ( $1 \times 75$  W +  $2 \times 150$  W)
- PCIe v.3.0-re visszafelé kompatibilis
- Tervezett bevezetése: 2017 végtől (többszöri csúszás)
  - BroadCom 200 Gbit Ethernet vezérlőjében először
  - AMD X580 chipset (2019. jan)

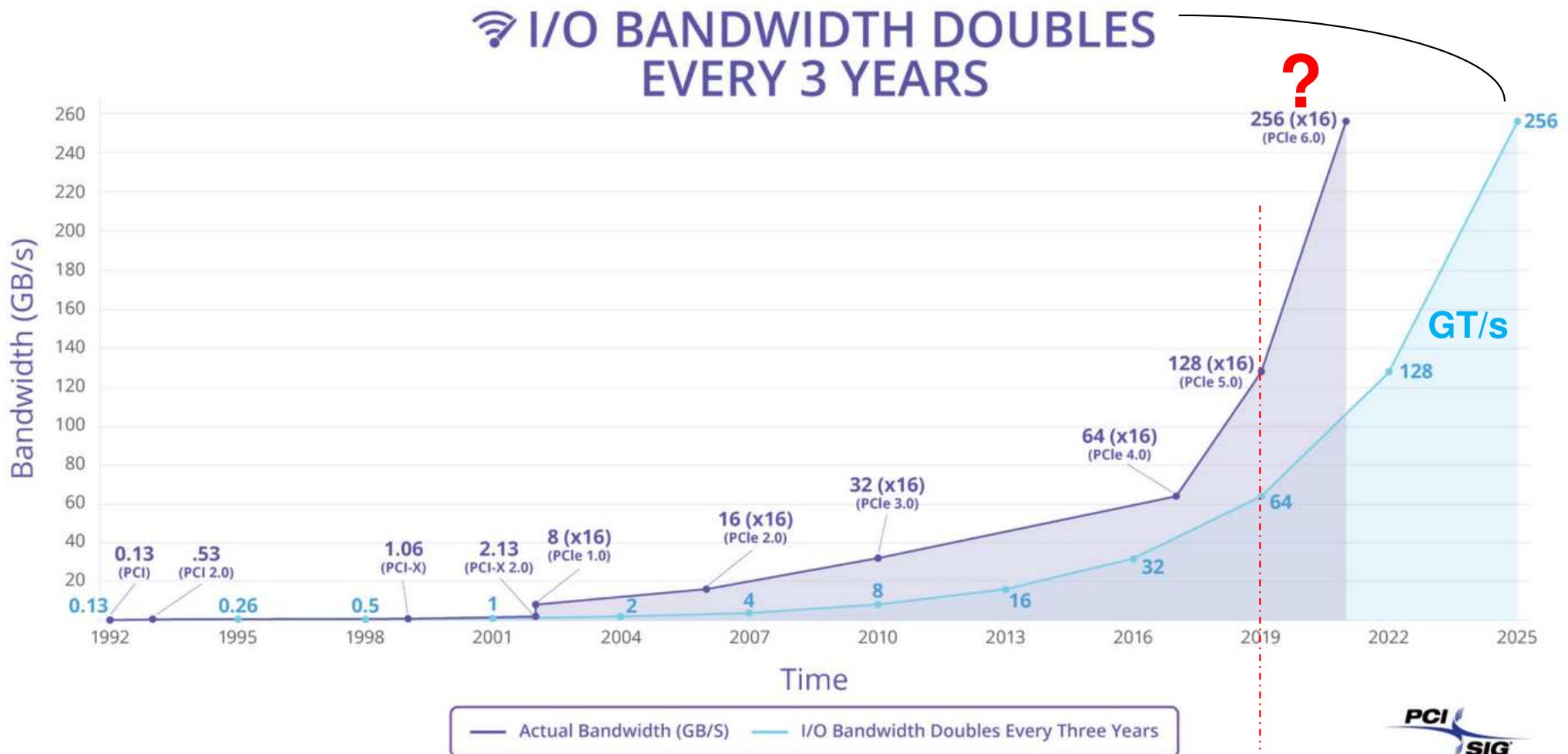
# PCI-Express v5.0 tulajdonságai (2019)

- Data rate (bandwidth): ~4 GB/sec/sáv/irány (x1 lane, x16 lane ~64GB/s)
- **IO sávszélesség ismét duplázódott: max ~32 GT/s**
- PCIe v.4.0-re visszafelé kompatibilis
- Tervezett bevezetése: 2019 (csúszás)

# PCI-Express v6.0 tulajdonságai (2021 - jelenleg)

- Data rate (bandwidth): ~8 GB/sec/sáv/irány (x1 lane, x16 lane ~128 GB/s)
- **IO sávszélesség ismét duplázódott: max ~64 GT/s**
- PCIe v.5.0-re visszafelé kompatibilis
- Tervezett bevezetése: 2021 eleje (csúszás)

# PCI-Express (v1.0 ... v6.0, v7.0? ..)



Sávszélesség duplázódása 3 évente.

2024: jelenlegi PCk-ben: PCIe v3.0, v4.0 , ill. v5.0 támogatás.

## PCIe® Speeds/Feeds - Pick Your Bandwidth

- Flexible to meet needs from handheld/client to server/HPC
- ~Max Total Bandwidth = Max RX bandwidth + Max TX bandwidth
- 35 Permutations yielding 11 unique bandwidth profiles
- Encoding overhead and header efficiency not included

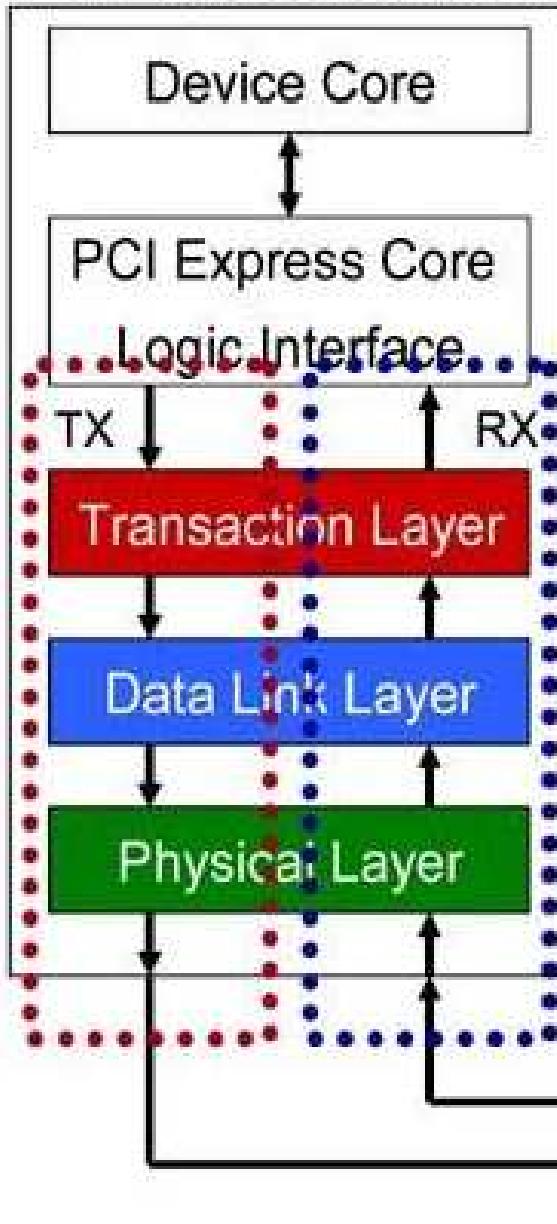
Specifications	Lanes				
	x1	x2	x4	x8	x16
2.5 GT/s (PCIe 1.x +)	500 MB/S	1GB/S	2 GB/S	4 GB/S	8 GB/S
5.0 GT/s (PCIe 2.x +)	1GB/S	2 GB/S	4 GB/S	8 GB/S	16 GB/S
8.0 GT/s (PCIe 3.x +)	2 GB/S	4 GB/S	8 GB/S	16 GB/S	32 GB/S
16.0 GT/s (PCIe 4.x +)	4 GB/S	8 GB/S	16 GB/S	32 GB/S	64 GB/S
32.0 GT/s (PCIe 5.x +)	8 GB/S	16 GB/S	32 GB/S	64 GB/S	128 GB/S
64.0 GT/s (PCIe 6.x +)	16 GB/S	32 GB/S	64 GB/S	128 GB/S	256 GB/S
128.0 GT/s (PCIe 7.x +)	32 GB/S	64 GB/S	128 GB/S	256 GB/S	512 GB/S

+ = data rate supported by this and subsequent spec revisions.

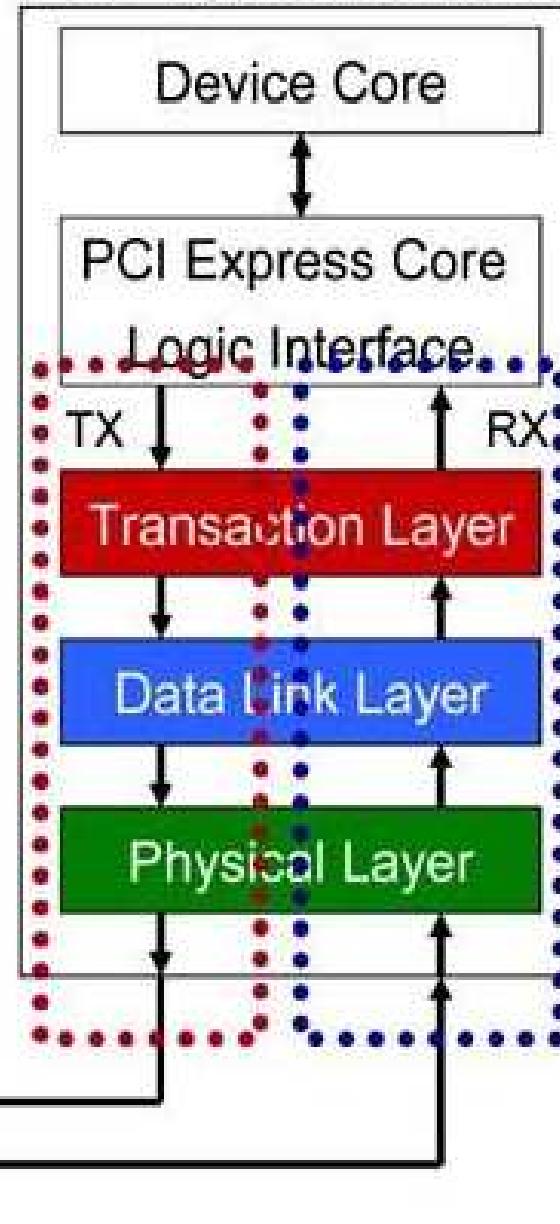
Újabb PCI-Express verziók kevesebb sávon biztosítják ugyanazt a sávszélességet, mint a korábbiak (azonos színek!)

# PCI Express rétegszerkezete

PCI Express Device A

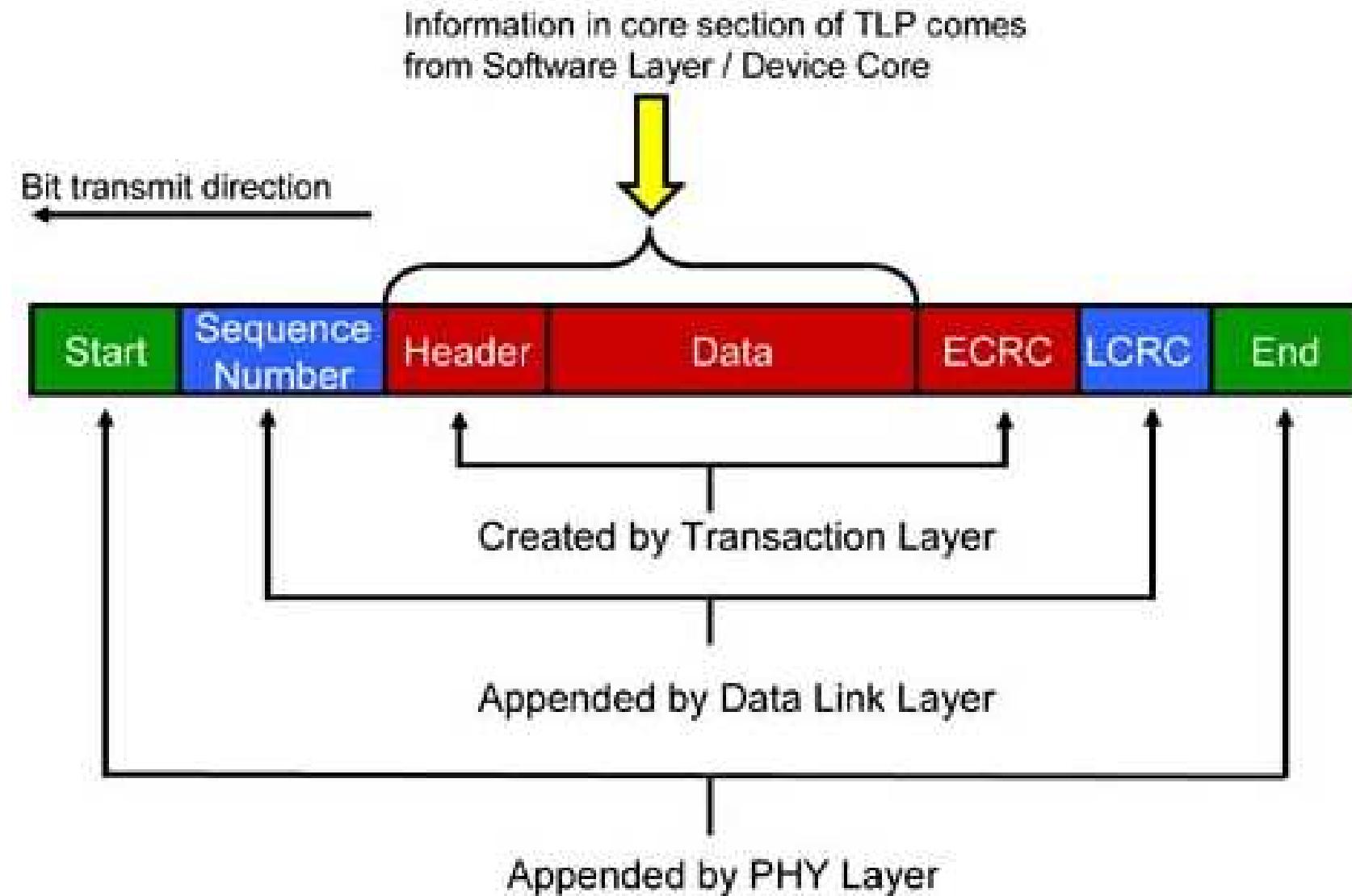


PCI Express Device B



- ISO OSI modell alsó négy! rétegét implementálja
  - Fizikai,
  - Adatkapcsolati,
  - Hálózati és Szállítási réteg egyben,

# Tranzakciós réteg csomagjai (keret)





# SCSI busz

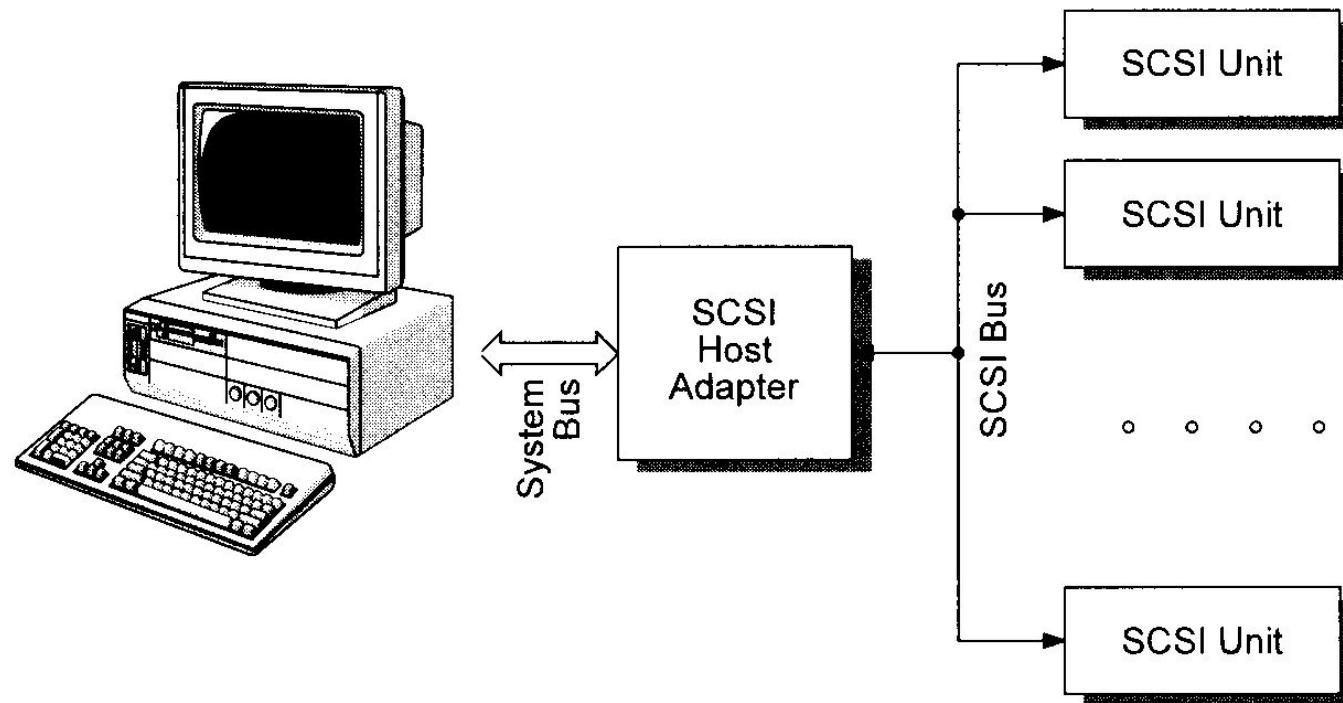


# Korai párhuzamos SCSI buszrendszer tulajdonságai



- **SCSI= Small Computer System/Standard Interface:** komplex, intelligens, síorientált (merev-, hajlékonylemez, CD-ROM, szalagos egység, scanner...) interfész
- **(IOP!).** Különféle perifériák illesztésére, CPU tehermentesítésére fejlesztették ki, az OS-től független felülettel rendelkezik. Sokoldalú eszköz, mivel nemcsak PC-s környezetbe, hanem UNIX munkaállomásokba és Apple-Mac gépekbe is integrálható.
- **Tulajdonságai:** A korai párhuzamos SCSI buszon általánosan 8 eszközt definiál (mai soros SCSI-n x1000 eszköz, sőt külső eszköz is csatlakoztatható) legnagyobb sebessége 640 Mbyte/s, max buszhossz 12m volt.
- Az eszközöket egyetlen vezérlő a **hostadapter (IOP)** kezeli, amely a számítógépes rendszer eszközeinek kapcsolatát építi fel a SCSI buszrendszerrel. **Nagysebességű, megbízható párhuzamos/soros** átvitelt biztosít a CPU és perifériák között. A korai szabványos SCSI csatolónak 50 (v. 68) lába volt.

# SCSI periféria busz és CPU rendszerbusz kapcsolata



- A *SCSI periféria busz* egy host adapteren (IOP) keresztül kapcsolódik a CPU *rendszer buszához*

# SCSI tulajdonságai (folyt. 1)

- A saját processzorral és memóriával rendelkező intelligens SCSI egységek kezdeményezőként (**initiator**) és fogadóként (**target**) is működhetnek! (lényegében egy I/O processzor eszköz)
- A korai SCSI vezérlő max **8 initiator**, és **8 target** egységet kezelt egyszerre. A kezdeményező adja ki az utasításokat, a cél pedig feldolgozza, és végrehajtja azokat. minden egységnek saját különböző címe van (0-7), amelyeket jumperekkel (rövidzár, kapcsoló) kell beállítani, hogy az interferenciát a buszon elkerüljük. A hagyományos SCSI szabványnak megfelelően hostadapter/vezérlő címe mindig 7-es, míg a szalagos egységé pl. a 0-ás, azonosítójuk **SCSI-ID=0** ill. 7.

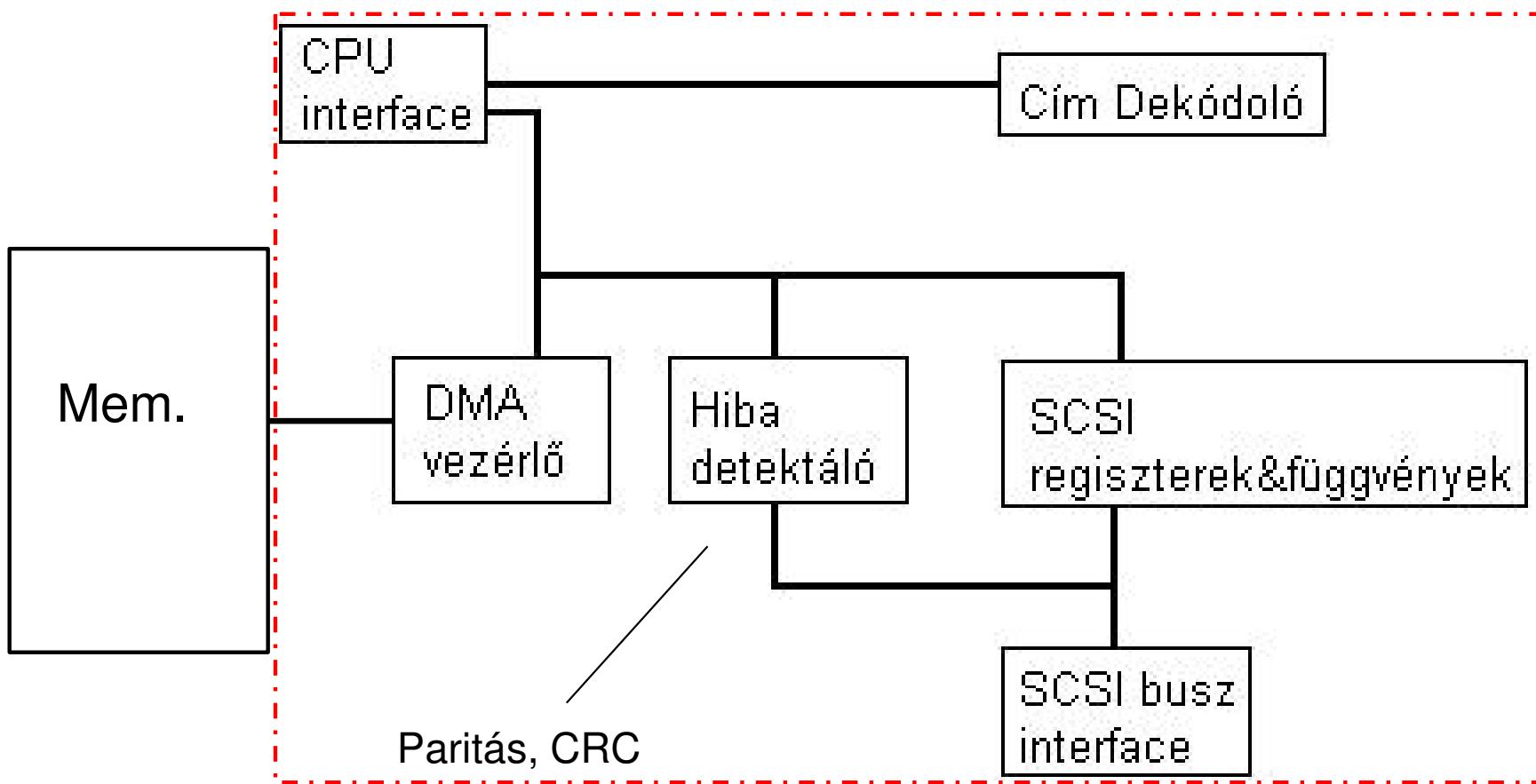
# SCSI tulajdonságai (folyt. 2)

- A **SCSI\_ID** mellett létezik a **LUN** = Logikai Egység Azonosítószám is: azaz minden targethez (perifériához) hozzárendelhető további 8 logikai egység, amiket az SCSI parancsok esetén saját LUN-nal azonosíthatunk.
- Az (korai, párhuzamos) **SCSI-I** busz kommunikáció 8-bites adatbuszon 1-bites paritás ellenőrzés mellett zajlott. Lassú target esetén, érdemesebb volt magasabb prioritású szintet állítani. A busz elején a vezérlő hostadapter van (jumper, kapcsoló, BIOS állítja), a másik végére pedig a lezáró ellenállást (mindig az utolsó eszköznél kellett tenni). Az egységeket egymás után láncba (**SCSI chain**) felfűzve, egyforma (50-eres) szalagkábellel lehetett csatlakoztatni.
- Alapvetően **Auszinkron** / de a **mai szabványok már a szinkron** protokollokat is támogatják!



# SCSI blokkdiagram

- SCSI vezérlő (host adapter = **IO Processzor**) általános felépítése



# SCSI busz fontosabb vezérlő jelei

- **REQUEST:** handshaking parancskérés, target által kezelt
- **ACKNOWLEDGE:** handshaking üzenet nyugtázása az initiator által
- **BUSY:** buszon a target foglaltságának jelzése (egy eszköz szabad, ha BUSY=0)
- **SELECTION:** initiator kiválasztja a target-et (SELECTION=0, nincs kiválasztva),
  - (SEL=0 esetén a target újra felépíti a kapcsolatot az initiator-ral a busz ideiglenes felszabadítása után)
- C/D: Control /Data: a target által kezelt jel, vezérlőadatok, parancsok, állapotinformációk jelzése a buszon
- I/O : Input/Output: szintén a target által kezelt vezérlőjel, amely az adatbusz adatforgalmának irányát mutatja
- MSG= Message: az üzenetküldés fázisának jelzésére szolgál, a target által kezelt
- ATTENTION: vezérlő figyelmezteti a célt
- RESET: az összes csatlakoztatott SCSI eszköz „reset-elése”, inicializálása, a sín alaphelyzetbe állítása

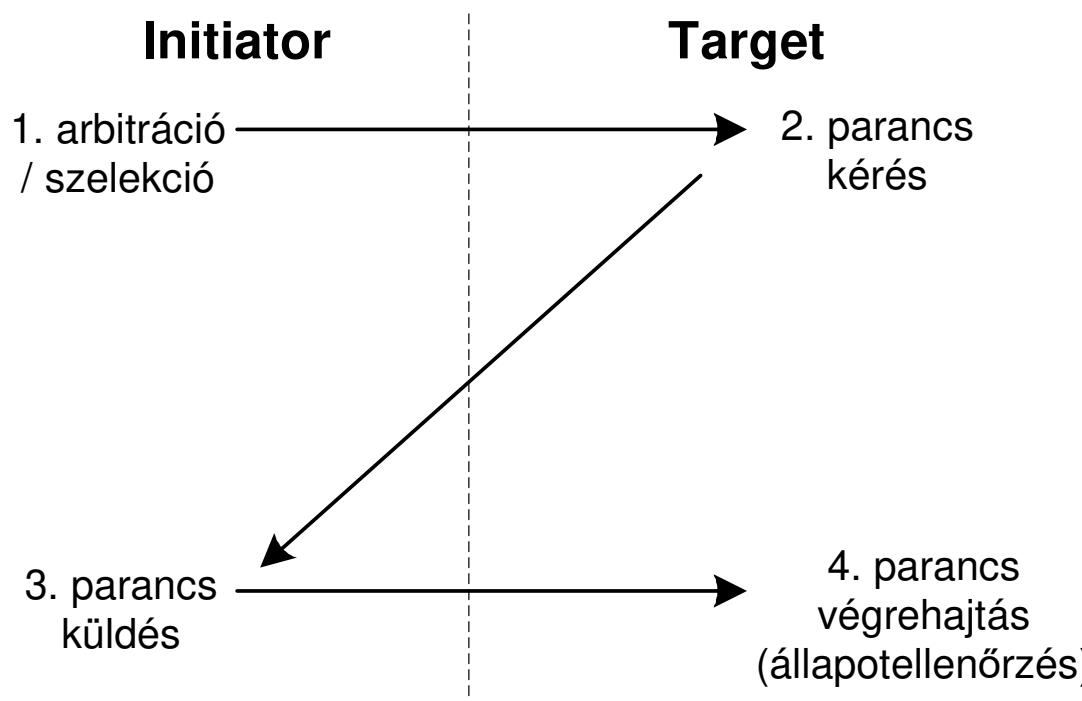
# SCSI busz fázisok #1

- Bus-free (szabad?)
  - Nem használja SCSI egység a buszt, és SEL# , BSY# inaktívak
- Arbitration (döntési mechanizmus)
  - egység aktiválja a **BSY#** és saját SCSI-ID azonosítóját a buszra helyezi
  - Rövid arbitrációs késleltetés után, ha nincs más aktív SCSI-ID magasabb prioritással, akkor az egység fogja vezérelni a buszt és aktiválja **SEL#** jelet
- Selection (kiválasztás)
  - Initiator kiválasztja a Target-et (bizonyos funkciókat kell végrehajtani)
  - az I/O# jel inaktív
  - Az initiator és target SCSI-ID-jének OR kapcsolatából egy címet állít elő, melyet a buszra helyez
  - target aktiválja BSY# jelet (foglalt lesz a busz)

# SCSI busz fázisok #2

- Reselection
  - Ha szükséges, az initiator újra kapcsolatot létesít a korábbi targettel, miután egy megszakítás történt (~folytatódhat a művelet ott ahol abbamaradt)
- 1. Command: Initiator parancsot küld (Target parancsot igényelhet)
- 2. Data: adatátvitel pl. Init -> Target
- 3. Message: üzenet továbbítás
- 4. Status: Target->Init állapot információ

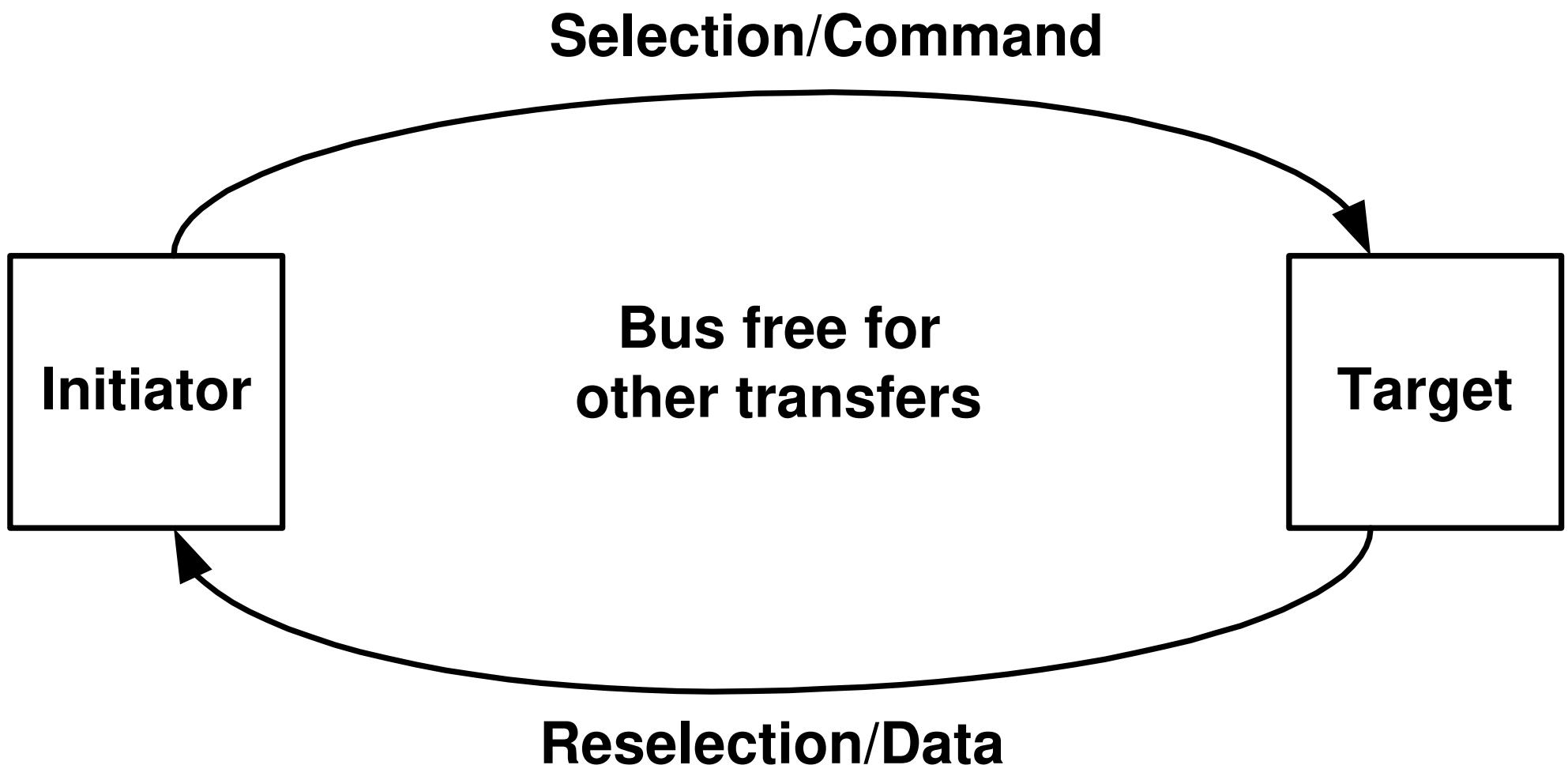
# SCSI kommunikáció főbb lépései



## SCSI busz fázisok a következők:

- Reselection (ha a target meg lett szakítva, folytathatja a komm.t az initiatorral)
- busz szabad (bus free?)
- arbitráció / szelekció (ki adjon?)
- üzenetküldés (msg)
- parancs elmegy (command)
- adatkapcsolat (data)
- állapotellenőrzés (status)
- üzenetzárás, kapcsolatbontás

# SCSI busz műveletek (kördiagram)



# Fontosabb SCSI szabványok

## ■ Párhuzamos (régi szabványok):

- *SCSI I*: 8 bites busz; átviteli seb. *aszinkron* módban 2.5Mbyte/s, *szinkron* módban 5Mbyte/s; 5Mhz busz-frekvencia; max 7 egység csatlakoztatható, 50 pólusú csatoló
- *SCSI II (Fast SCSI)*: első valódi SCSI szabvány, 8 bites busz; 10Mbyte/s; 10 Mhz buszfrekvencia; max 7 egység; 50 pólusú
- *Wide-SCSI*: 16 bites szinkron busz; 20Mbyte/s; 10Mhz; 15 egység; 68 pólusú csatlakozó
- *Ultra Wide SCSI*: 16 bites; 40Mbyte/s; 20 Mhz; max 15 egység; 68 pólusú csat.
- *Ultra2 Wide SCSI*: 16 bites; 80Mbyte/s; 40Mhz; max 15 egység; 68/80 pólusú; max kábelhossz. 12m
- *Ultra-3160 SCSI*: 16 bites; 160Mbyte/s; 80Mhz; max 15 egység; 68/80 pólusú; 12m (differenciális jellel működik)
- *Ultra3-320 SCSI*: 16 bites; 320Mbyte/s; max 15 egység; 68/80 pólusú; 12m (differenciális jellel működik)
- *Ultra3-640 SCSI*: 16 bites; 640Mbyte/s, max 15 egység; 68/80 pólusú;

## ■ Mai szabványok:

- **Soros (SAS)** 1.0-4.0: 1.5 ... 22 Gbit/s, (Adaptec.com), 6m, 16K eszköz!!
- **USB SCSI (UAS)**: **USB3.0-4.0** max 40 Gbit/s; 127 eszköz, 3m
- **Optikai (Fiber channel SCSI)**, **128GFC** ~ max 12800 MB/s!; akár 127 eszköz, 500m
- iSCSI: SCSI over TCP/IP
- ...





# Számítógép Architektúrák II.

(MIVIB344ZV)

8. előadás: Programozható logikai eszközök:  
CPLD, FPGA, HLS: magas szintű szintézis

Előadó: Dr. Vörösházi Zsolt  
[voroshazi.zsolt@mik.uni-pannon.hu](mailto:voroshazi.zsolt@mik.uni-pannon.hu)

# Jegyzetek, segédanyagok:

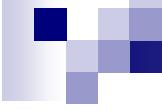
- Könyvfejezetek:
  - <http://www.virt.uni-pannon.hu> → Oktatás → Tantárgyak → Számítógép Architektúrák II.
- Fóliák, óravázlatok .ppt (.pdf)
- Feltöltésük folyamatosan

# Eml: Vezérlő egységek fajtái:

- I. Huzalozott (klasszikus) módszerek (pl. korai RISC architektúrák):
  - Mealy-modell,
  - Moore-modell.
- II. Mikro-programozott módszerek (reguláris vezérlési szerkezettel – pl. mai CISC, RISC architektúrák):
  - Horizontális mikrokódos vezérlő,
  - Vertikális mikrokódos vezérlő.
- III. Programozható logikai eszközök (PLD):
  - 1.) Maszk-programozható/"makrocellás" típusok: PLA, PAL, GAL, CPLD,
  - 2.) Tetszőlegesen újra-konfigurálható (=szoftveresen) típus: FPGA

# PLD/ FPGA ismeretkörök

1. Mik azok a.) **Programozható Logikai Eszközök** és az b.) **FPGA-k?**  
Összeköttetések programozhatósága
2. Tervezési módszerek (Design methods)
3. Tervezés folyamata (Design flow)
4. Magas-szintű szintézis (HLS – High-Level Synthesis)
5. Fejlesztő környezetek, hardver leíró nyelvek (HDL - Hardware Description Languages)

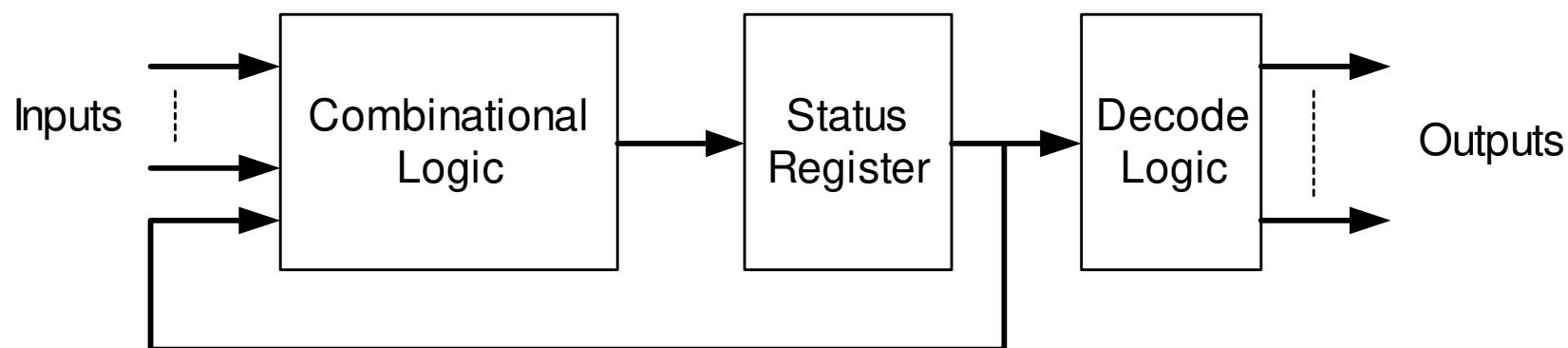


Ismertetés

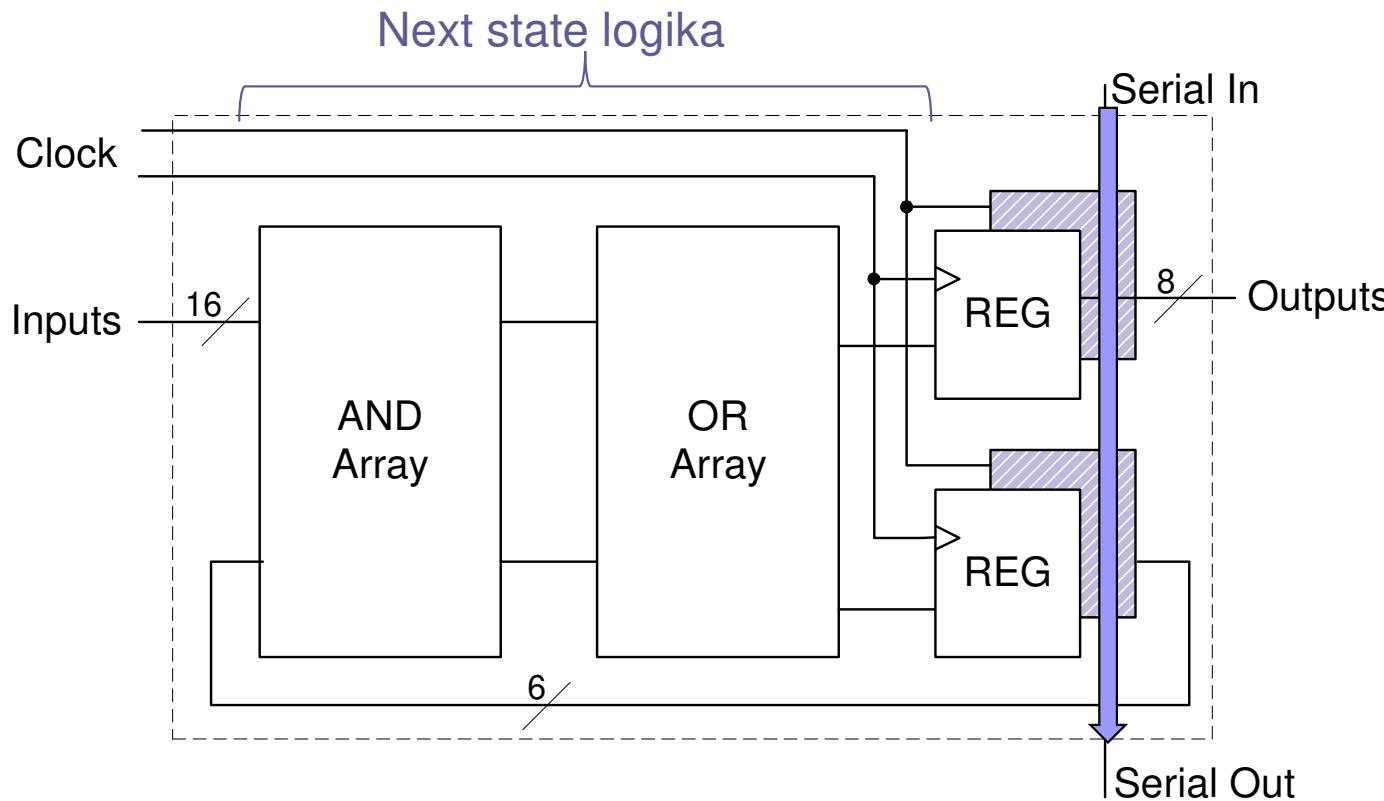
# 1.a) Programozható logikai eszközök (**PLD**)

# Állapotgép FSM tervezés tulajdonságai

- Két kombinációs logikai hálózatból és egy regiszterből áll
- Tervezés során az állapot-átmeneteket vesszük figyelembe, **DE**
- Hibavalószínűség nagy,
- Szimulációs eszközök (CAD Tools) hiánya (~1970),
- Hibák lehetségesek a prototípus fejlesztése során is,
- Könnyen konfigurálható / flexibilis eszközök kellettek:  
→ mindenek miatt használunk programozható logikai alkatrészeket



# Field Programmable Logic Sequencer (FPLS) – Programozható logikai sorrendvezérlő

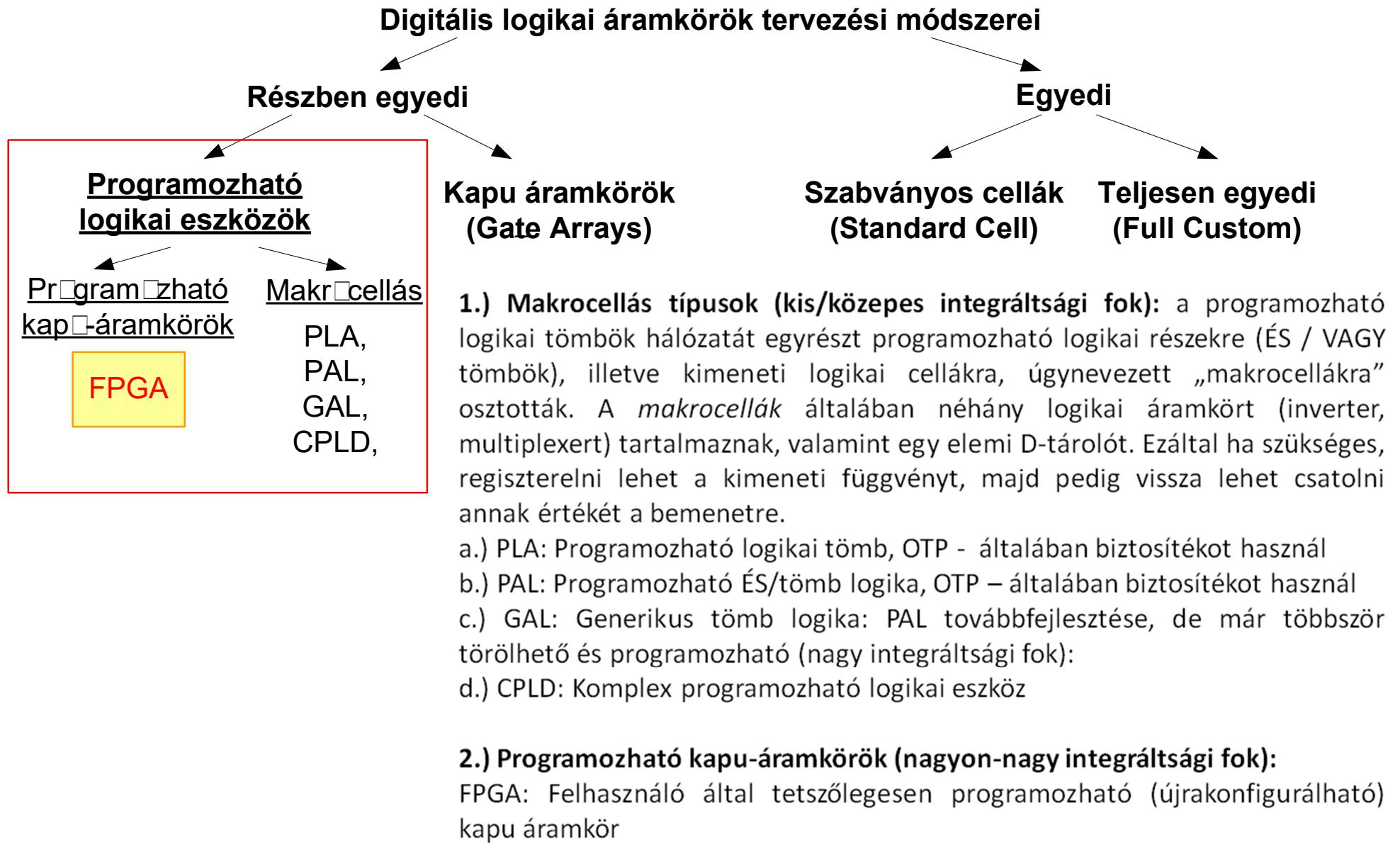


Működése:

- Normál (D-FF),
- Debug (Shift Reg.)

Bemenete (16), 1 RESET és 1 órajel, ill. 8 kimenete van. A regiszterek (REG) belső állapotot tárolnak, amelyek az órajel hatására a kimenetre kerülnek, vagy visszacsatolódnak. A Next-State logika ill. a kimeneti szintek meghatározásánál programozható AND/OR tömböket használnak.

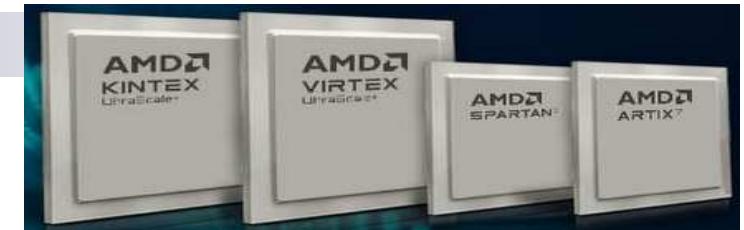
# Tervezési módszerek



\* Fontos: PLD/FPGA-s a részek nem találhatóak meg a könyvben!

# PLD – Programozható logikai eszközök

- A Programozható logikai áramköröket (**PLD: Programmable Logic Devices**) általánosan a *kombinációs logikai hálózatok* és *sorrendi hálózatok* tervezésére használhatjuk. Napjainkban ezek *VLSI* (Very-Large Scaling Integrated IC) típusú alkatrészek:
  - Több millió logikai kaput (tranzisztor) jelent!
- Azonban még a hagyományos kombinációs logikai hálózatok dedikált összeköttetésekkel, illetve kötött funkcióval (kimeneti függvény) rendelkeznek, addig a **programozható logikai eszközökben változtathatók, az alábbi lehetséges módokon:**
  - 1.) A felhasználó által *egyszer programozható / konfigurálható* logikai eszközök (**OTP**: One Time Programmable), amelynél a gyártás során nem definiált funkció egyszer még megváltoztatható (ilyenek pl. a korai PAL, PLA eszközök)
  - 2.) **Többször, akár tetszőleges módon** programozható logikai eszközök = újrakonfigurálható (ilyenek pl. a korábbi GAL, vagy a mai modern CPLD, **FPGA** eszközök)



# PLD-k két fő típusa:

## ■ 1.) Makrocellás PLD-k:

- PLA: } Elavult, 1x prog. (OTP)
  - PAL: }
  - GAL: }
  - CPLD: }
- Többször programozható

## ■ 2.) FPGA (Field Programmable Gate Array): "Felhasználó által programozható/újra-konfigurálható kapuáramkörök"

- AMD-XILINX** (Spartan, Virtex, Kintex, Artix) ~ 49% !
- Intel-FPGA** (Agilex, Stratix, Arria, Cyclone) ~ 40%
- Lattice 6%
- MicroChip (MicroSemi/Actel) ~4%
- QuickLogic & other smaller vendors <1%

AMD

XILINX

intel® FPGA

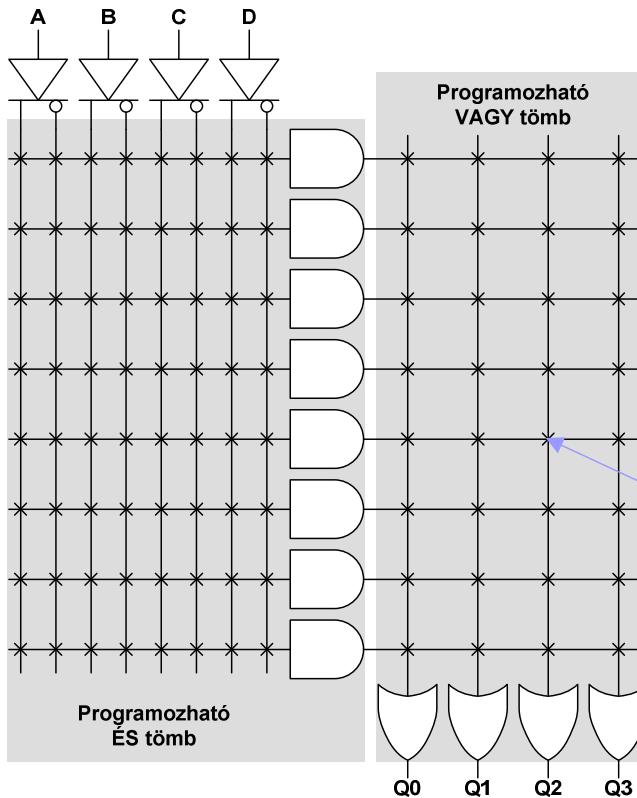
Lattice®  
Semiconductor  
Corporation

MICROCHIP  
Microsemi Product Portfolio

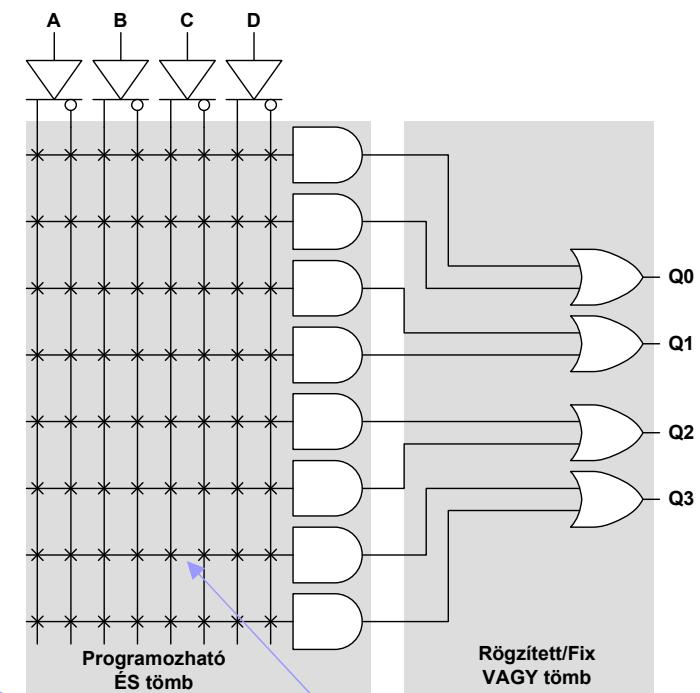
QuickLogic

# Makrocellás PLD-k

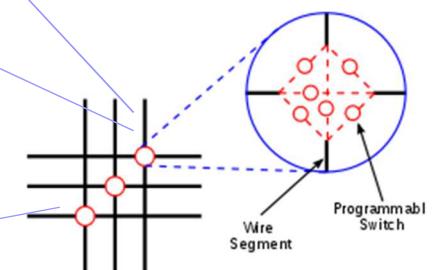
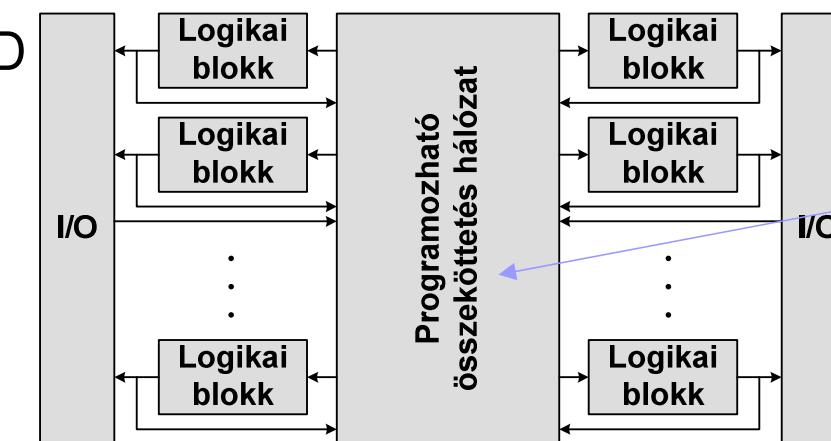
PLA



PAL

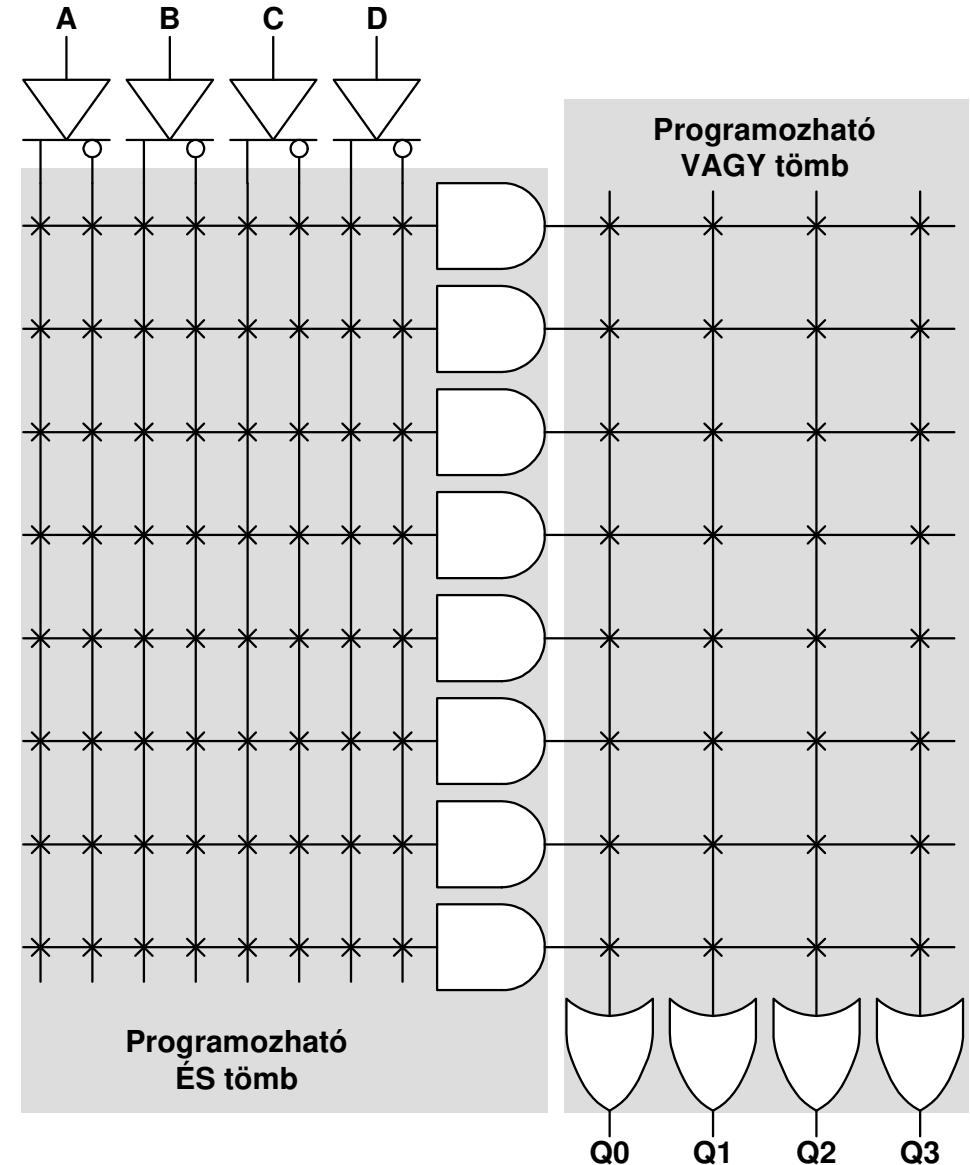


CPLD



# Programmable Logic Array (PLA)

- Mindkét része (AND, OR) programozható
- Bármely kombinációja az AND / OR-nak előállítható
- Mintermek OR kapcsolata (DNF)
- Programozható kapcsolók a horizontális/ vertikális vonalak metszésében
- $Q_n$  Kimeneteken D tárolók!  
(visszacsat. a bemenetekre)



# PLA

- 1970-ben, a TI (Texas Instruments) által kifejlesztett eszköz *mindkét részhálózata* (*ÉS*, illetve *VAGY tömb*) *programozható összeköttetéseket* tartalmazott, amelyek segítségével tetszőleges mintermek tetszőleges VAGY kapcsolata előállítható (DNF alakot), ezáltal bármilyen kombinációs logikai hálózat realizálható volt (természetesen adott bemenet, ill. kimenet szám mellett).
- A programozható ÉS / VAGY tömbökben úgynevezett „*programozható kapcsolók*” vannak elhelyezve a horizontális/ vertikális vonalak metszéspontjában.
- Amennyiben a Qn kimenet(ek)re tárolókat kötünk (pl. egyszerű D tárolót), majd pedig visszacsatoljuk a programozható logikai hálózat bemenete(i)re akár egy *sorrendi hálózati viselkedést* is meghatározhatunk.

# Programozásuk (Fuse) biztosítékok segítségével

- Biztosíték – korai típusok esetén (ma már újraprogramozhatók)
- Az összeköttetés mátrix metszéspontjaiban akár kis **biztosítékok (fuse)** helyezkednek el. Gyárilag logikai ‘1’-est definiál, tehát vezetőképes. Ha valamilyen spec. programozó eszközzel, a küszöbnél nagyobb feszültséget kapcsolunk rá, átégethető, tehát szigetelővé (nem-vezető) válik, és logikai ‘0’-át fog reprezentálni.
- A biztosíték átégetése, csak egyszer lehetséges, utána már csak a programozott állapotot fogja tárolni (**OTP** – One time programmable IC).

# Példa: PLA tervezése

- Realizálja a következő függvényeket:

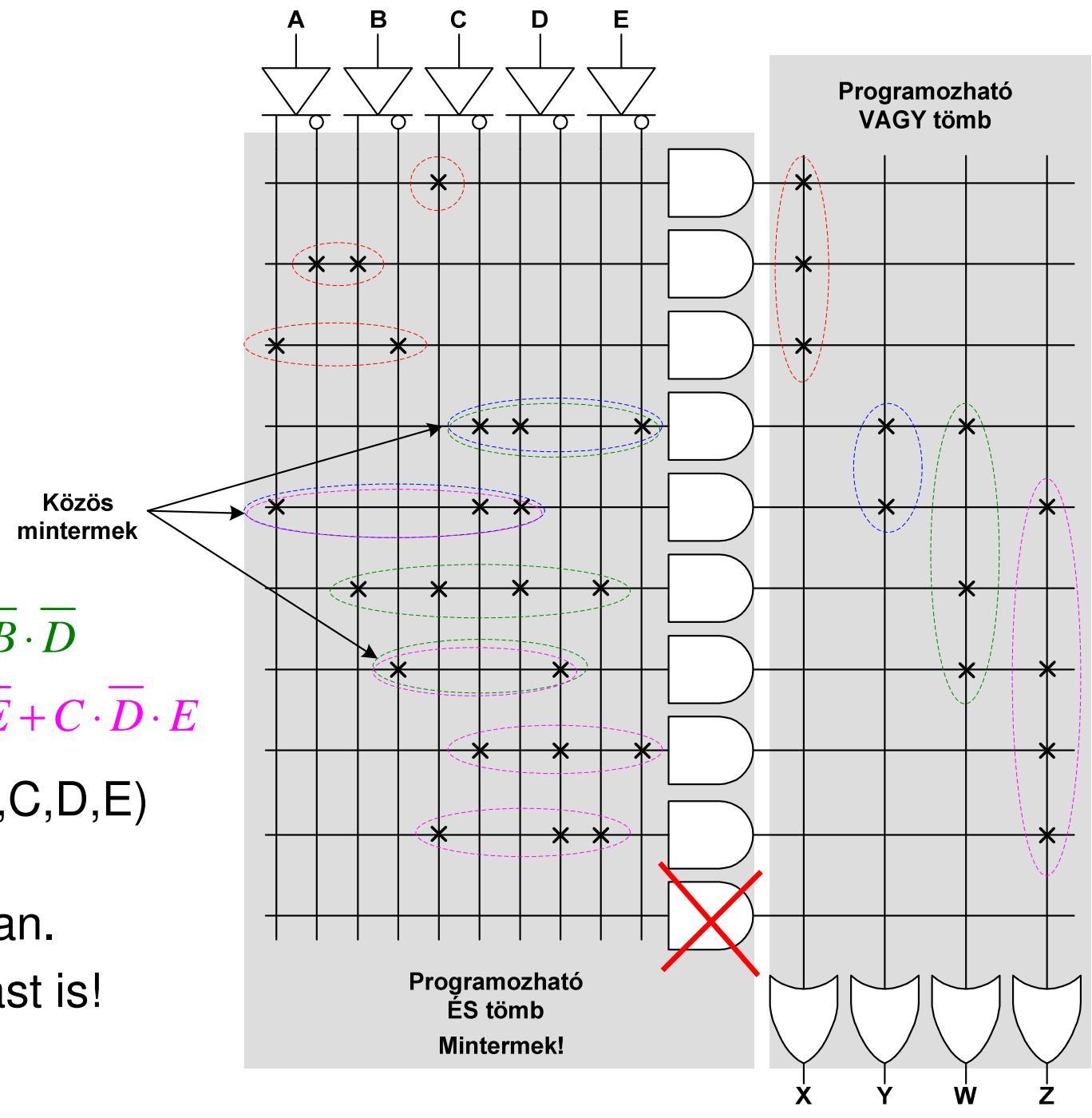
$$X = C + \bar{A} \cdot B + A \cdot \bar{B}$$

$$Y = \bar{C} \cdot D \cdot \bar{E} + A \cdot \bar{C} \cdot D$$

$$W = \bar{C} \cdot D \cdot \bar{E} + B \cdot C \cdot D \cdot E + \bar{B} \cdot \bar{D}$$

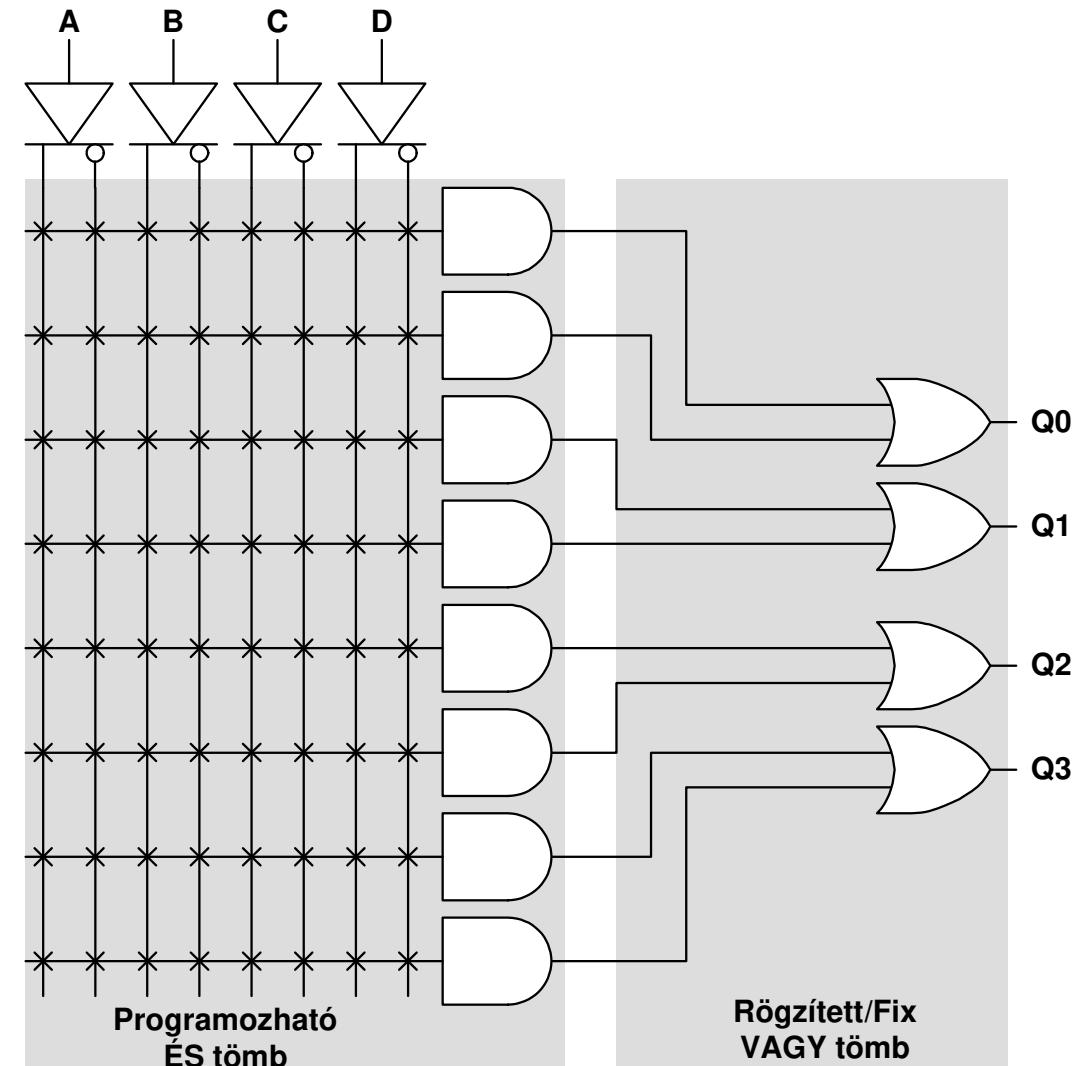
$$Z = A \cdot \bar{C} \cdot D + \bar{B} \cdot \bar{D} + \bar{C} \cdot \bar{D} \cdot \bar{E} + C \cdot \bar{D} \cdot E$$

- Tehát 5 bemenete (A,B,C,D,E)  
és
- 4 kimenete (X,Y,W,Z) van.
- Rajzoljuk fel a kapcsolást is!



# Programmable AND Logic (PAL)

- Egy programozható rész - AND / míg az OR fix
- Véges kombinációja áll elő az AND / OR kapcsolatoknak
- Metszéspontokban kevesebb kapcsoló szükséges
- Gyorsabb, mint a PLA
- $Q_n$  kimeneteken D tárolók (visszacsatolódhatnak a bemenetekre)



# PAL

- Elsőként, 1978-ban az MMI (Monolithic Memories Inc.) jelent meg ilyen programozható eszközökkel, majd pedig későbbi jogutódja a Lattice Semiconductor, illetve az AMD a 80'-as évek végén.
- A PAL hálózatban *VAGY tömb fix/rögzített* a programozható részt az *ÉS tömb* jelenti, míg az. Igy a tetszőleges mintermeknek csak egy véges kombinációja (VAGY) állítható elő: a lehetséges kimeneti függvények variálhatóságából veszünk, cserébe viszont a VAGY részek dedikált útvonalainak jelterjedési sebessége nagyobb, míg az eszköz mérete kisebb és ezáltal olcsóbb is lesz.
- Ezáltal a metszéspontokban kevesebb kapcsoló szükséges („gyorsabb”, mint a PLA). Hasonlóan a PLA-khoz, amennyiben a Qn kimenet(ek)re tárolókat kötünk (pl. egyszerű D tárolót), majd pedig visszacsatoljuk a programozható PAL logikai hálózat bemenete(i)re akár sorrendi hálózati viselkedést is könnyen valósíthatunk.

# GAL (Generic Array Logic): Általános tömb logika

- 1985-ben a Lattice Semiconductor fejlesztette ki elsőként,
  - amely a *PAL*-nak egy továbbfejlesztett változatát képviseli.
  - Ugyanolyan belső struktúrával rendelkezik, mint egy *PAL* áramkör,
  - azonban többször programozható: tehát törölhető és újraprogramozható eszköz.
  - EEPROM technológiát (lásd. lebegő-gate) alkalmaz. Később a National Semiconductor, és AMD is megjelent saját *GAL* sorozataival a piacon.

# CPLD

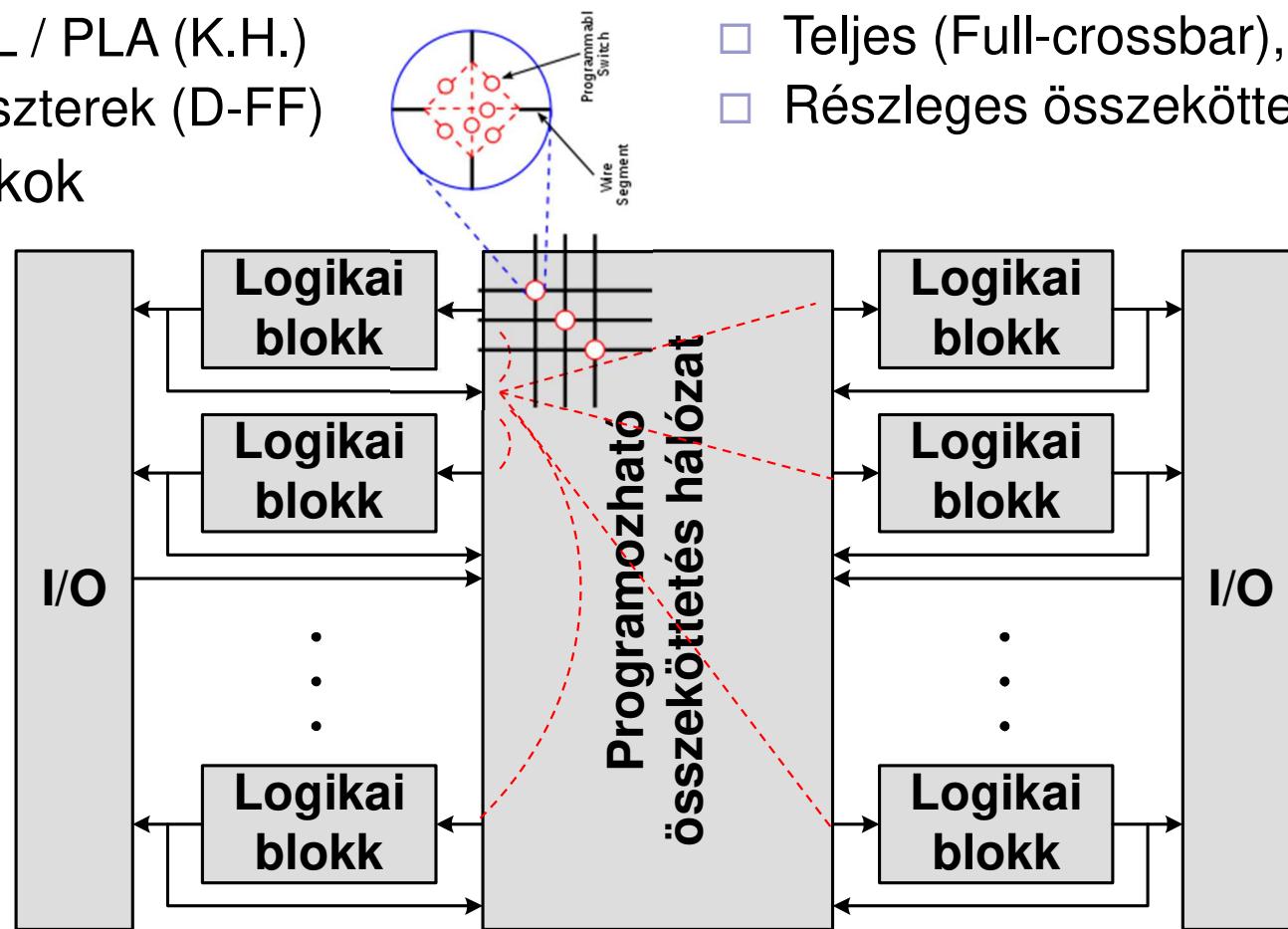
## CPLD (Complex Programmable Logic Devices): Komplex-Programozható Logikai eszközök

- Valójában átmenetet képeznek a kis/közepes integráltsági fokú **makrocellás** PLD-k GAL/PAL áramkörei, illetve a nagy integráltsági fokú FPGA kapu-áramkörök között.
- A GAL/PAL áramköröktől architekturálisan annyiban különbözik, hogy ki lett bővítve: nem egy-, hanem több logikai cellamátrixot tartalmaz, amelyek konfigurálható blokkok reguláris struktúrájában vannak elrendezve. A mai modern FPGA áramköröktől viszont az különbözteti meg felépítésben, hogy *nem tartalmaz dedikált erőforrásokat* (pl. szorzók, memória blokkok).
- a legnagyobb gyártók, amelyek jelenleg is aktív szereplői a CPLD-k piacának a következők: AMD-Xilinx, Intel, Lattice Semiconductor, MicroSemi stb.

# Complex Programmable Logic Device (CPLD)

- 1 Logikai Blokkon („makrocellán”) belül:
  - ~ PAL / PLA (K.H.)
  - Regiszterek (D-FF)
- I/O Blokkok

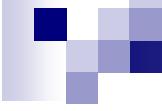
- Programozható összeköttetések (PI: Programmable Interconnection)
  - Teljes (Full-crossbar), vagy
  - Részleges összeköttetés hálózat



# CPLD (folyt)

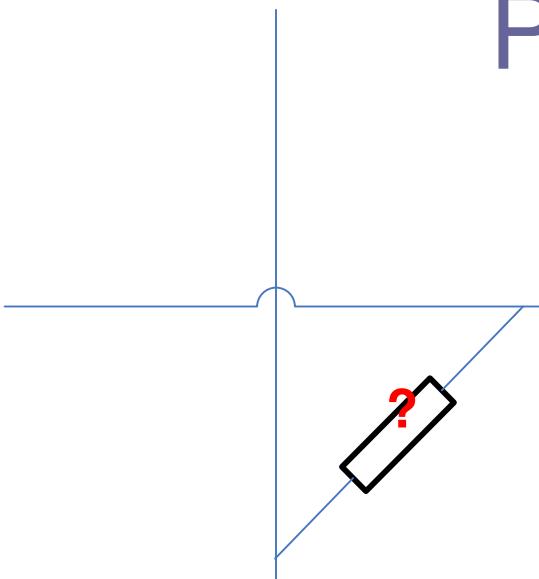
A CPLD-kben található Logikai Blokk-ok (~**makrocellák**):

- egyrészt *logikai kapuk* tömbjeit tartalmazzák (hasonlóan a PAL/GAL áramkörök felépítéséhez – DNF alak),
- másrészt *regisztereket* (D-tárolókból) tartalmaznak a logikai tömbök által előállított kimenetek átmeneti tárolásához, valamint
- *multiplexereket*, mellyel a programozható összeköttetés hálózatra, vagy I/O blokkok celláihoz lehet továbbítani a belső Logikai Blokkok által előállított kimeneti értékeket. Ezáltal nemcsak logikai kombinációs hálózatokat, hanem akár sorrendi hálózatokat is egyszerűen megvalósíthatunk CPLD-k segítségével
- A CPLD-kben található *Programozható összeköttetés hálózat*
  - teljes összeköttetést ( mindenki-mindekel), vagy
  - részleges összeköttetést (valamilyen struktúra szerint, pl. bemenetet – kimenettel, főként régi CPLD típusok esetén) biztosít az egyes blokkok között.
- Kikapcsoláskor a CPLD konfigurációs memóriája megtartja értékét (non-volatile típus), ezért nem kell egy külső pl. ROM memóriát használni az inicializációs minták tárolásához, bekapcsoláskor ezek automatikusan betöltésre kerülnek. A CPLD-ket közkedvelten alkalmazzák különböző interfészek jeleinek összekapcsolásához (*glue-logic*), amennyiben a jeleken átalakításra is szükség van, továbbá áraik az FPGA-k árainál jóval kedvezőbbek.



# Hogyan programozhatók a VLSI alkatrészek?

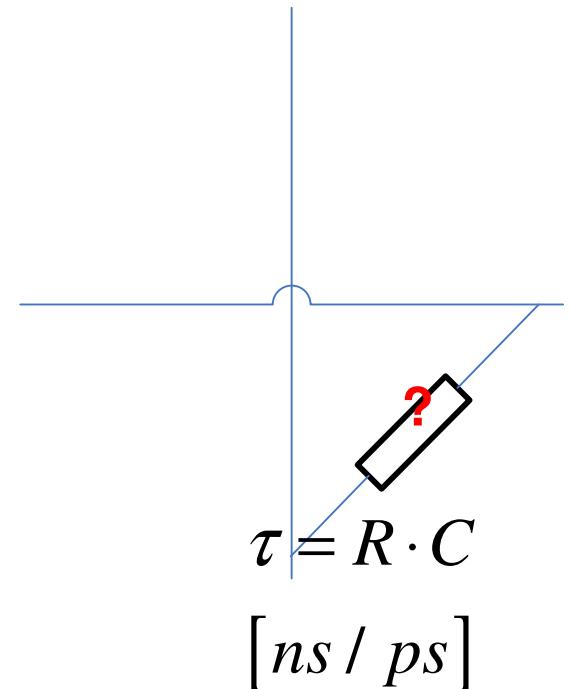
Programozási technikák  
összeköttetésekre



# Programozási technikák

Mi van a programozható összeköttetések csomópontjaiban, illetve milyen módszerrel programozhatóak?

- a.) SRAM
- b.) MUX
- c.) Antifuse
- d.) Floating Gate
- e.) EPROM/EEPROM/Flash

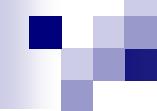


# Programozás = konfigurálás

**Konfigurálás (PLD/FPGA esetén)** – mielőtt az eszközt használni szeretnénk egy speciális (manapság általában JTAG szabványú) programozó segítségével „fel kell programozni”: le kell tölteni a konfigurációs állományt (bitfájl, vagy bitstream fájl). A programozás a legtöbb PLD esetében a belső programozható összeköttetésének fizikai típusától függően azok beállításával történik.

A programozható **összeköttetésekben** a következő lehetséges alkatrészek találhatóak:

- Biztosíték (Fuse):** átégettések után nem visszafordítható a programozási folyamat (OTP). Korábban a PAL eszközök népszerű kapcsoló elemeként használták.
- Antifuse technológia:** (OTP), az antifuse-os kristályszerkezetű kapcsoló elem 'átolvasztása' után egy nagyon stabilan működő összeköttetést kapunk, amely sajnos szintén nem visszafordítható folyamatot jelent. A technológia drága, az előállításához szükséges maszk-rétegek nagy száma miatt, nagyon jó zavarvédelettség elérése érdekében használják (pl. ūrkutatás).
- SRAM cella + tranzisztor:** tetszőlegesen programozható (FPGA-k esetén legelterjedtebb kapcsolás-technológia), az SRAM-ban tárolt inicializáló értéktől függően vezérelti a tranzisztor gate-elektródáját
- SRAM cella + multiplexer:** tetszőlegesen programozható az SRAM cellában tárolt értéktől függően (kiválasztó jel) vezérelhető a multiplexer
- Lebegő kapus tranzisztor (Floating Gate) technológia:** elektromosan tetszőlegesen programozható, a mai EEPROM/Flash technológia alapja. **24**



Ismertetés

## **1.b) FPGA (Field Programmable Gate Array) architektúrák**

# FPGA

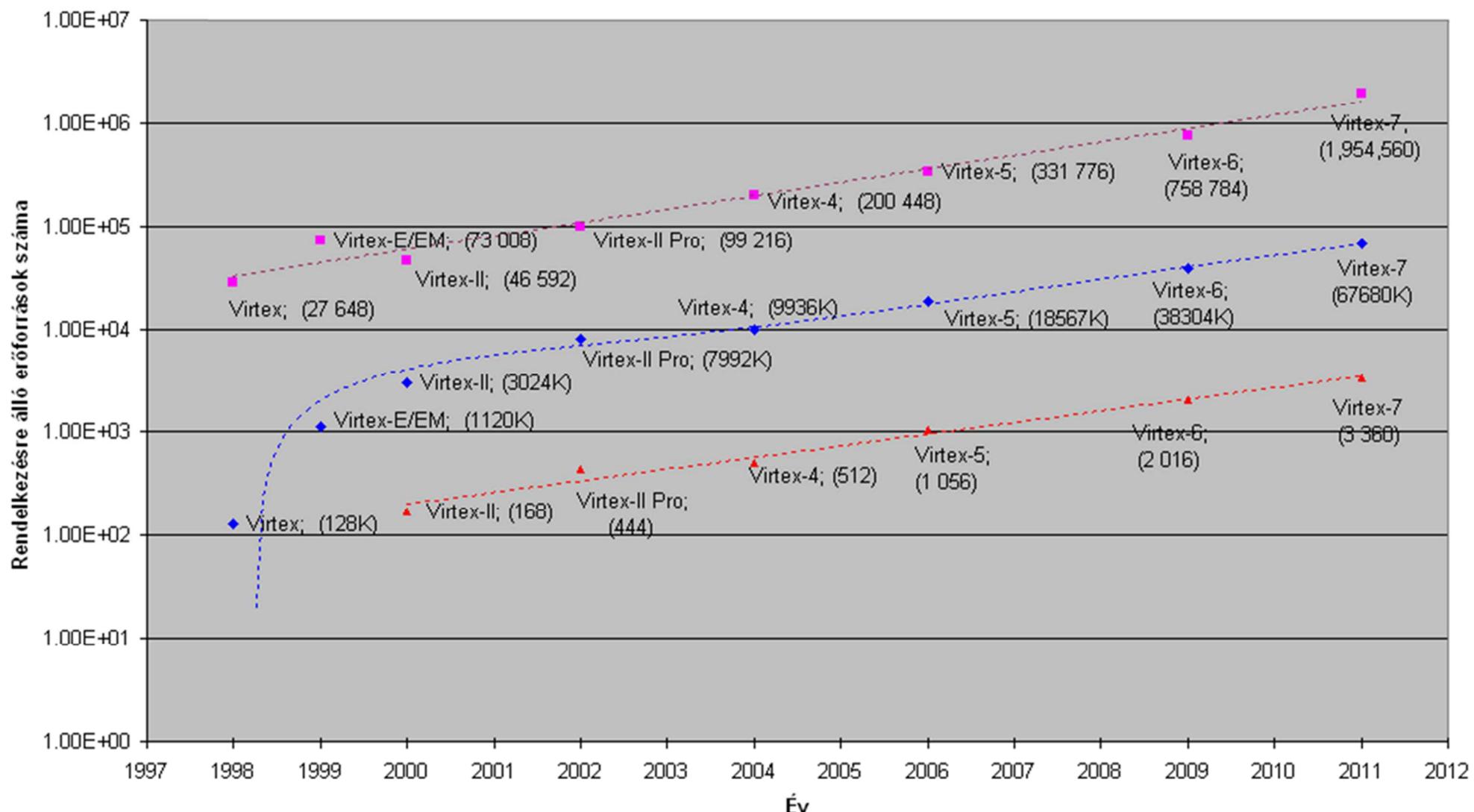
**Field Programmable Gate Array** = „Felhasználó által tetszőlegesen/többször” programozható kapuáramkör

- architekturálisan tükrözi mind a PAL, ill. CPLD felépítését, komplexitásban pedig a CPLD-ket is felülmúlják. Nagy/nagyon-nagy integráltsági fok: ~10.000 - ~10.000.000 *ekvivalens logikai kaput* is tartalmazhat gyártótól, és sorozattól függően.
- *Ekvivalens tranzisztorszám*
  - Xilinx Virtex-7 2000T FPGA esetén már meghaladta a **~6.5 milliárdot** (2012 – 28nm), amely **~2 millió logikai cellát** jelentett.
  - a kapható legnagyobb Xilinx Virtex-Ultrascale+ XCVU440 (2015 – 20nm->16nm) FPGA: **20 milliárd tr.** - **~4.4 millió logikai cella!** (~ 50 millió logikai kapu ekv.)
  - Intel/Altera Stratix-10, **30 milliárd tranz.** (2016, ~5.5 millió logikai cella, 4mag ARM-Cortex A53, 14nm)
  - Xilinx Virtex Ultrascale+ VU19P, **35 milliárd tranz.**, 16 nm (2019, ~9 millió logikai cella)
  - Intel Stratix-10 GX 10M **43.3 milliárd tranz.**, 14 nm (2020, **10.2 millió logikai cella**)  
Dual FPGA mag!
  - Xilinx Versal ACAP **50 milliárd tranz.** 7nm. (2020, **7.4 millió logikai cella**)
  - AMD Versal VP1902 SoC (2024?, **18.5 millió logikai cella**)  
Quad FPGA mag!



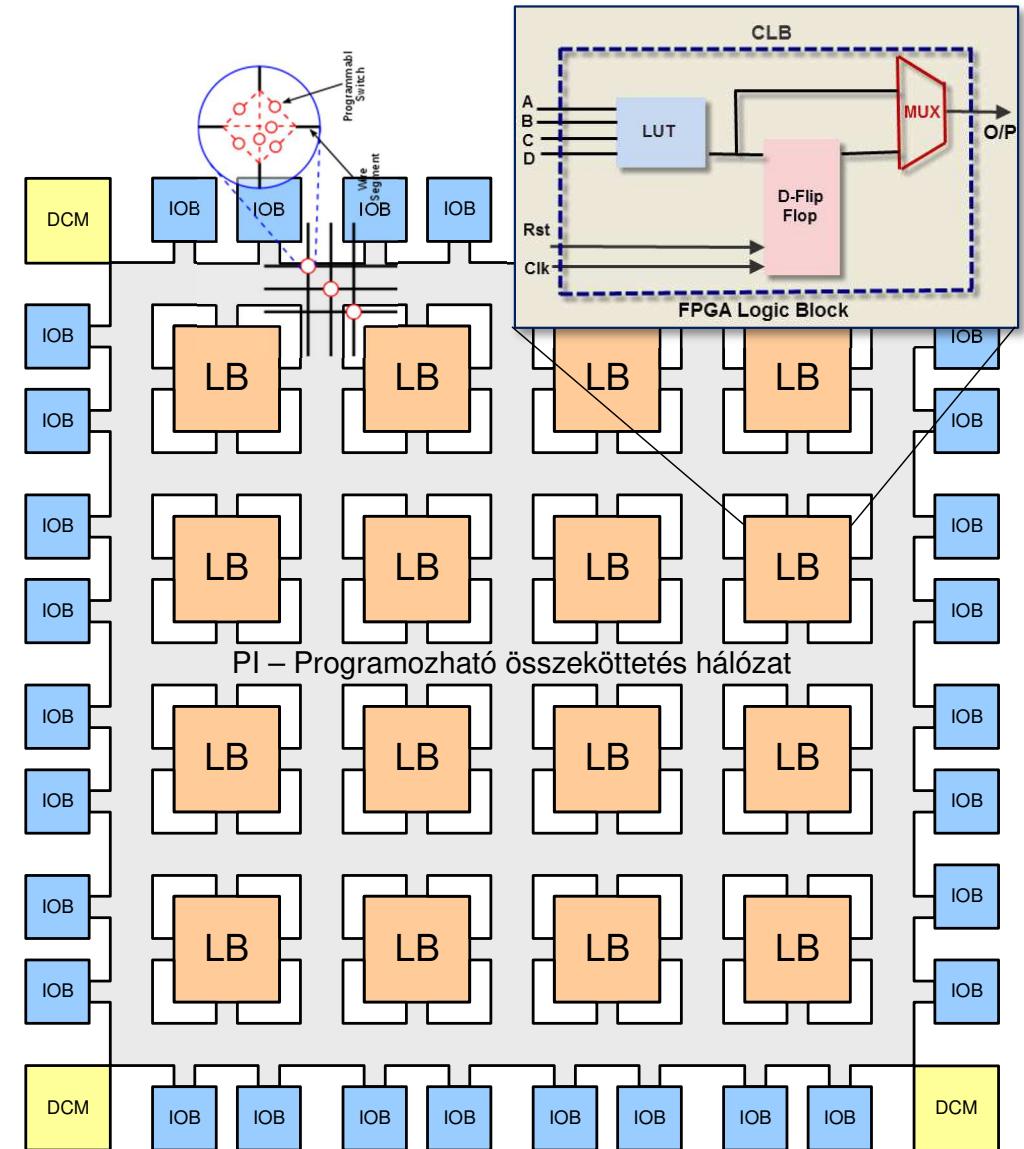
# Nagy-teljesítményű Xilinx Virtex FPGA család erőforrásainak alakulása (1998-2012 időszakban)

\*Tranzisztorok fizikai méretcsökkenésének („Scaling-down”) hatása a fejlődésre



# FPGA: „általános” erőforrások

- **LB/CLB:** Konfigurálható Logikai Blokkok, amelyekben LUT-ok (Look-up-table) segítségével realizálhatók például tetszőleges, több bemenetű (ált. 4 vagy 6), egy-kimenetű logikai függvények. Ezek a kimeneti értékek szükség esetén egy-egy D flip-flopban tárolhatók el; továbbá multiplexereket, egyszerű logikai kapukat, és összeköttetéseket is tartalmaznak.
- **IOB:** I/O Blokkok, amelyek a belső programozható logika és a külvilág között teremtenek kapcsolatot. Programozható I/O blokkok kb. 30 ipari szabványt támogatnak (pl. LVDS, LVCMOS, LVTTL, SSTL stb.).
- **PI:** az FPGA belső komponensei között a programozható összeköttetés hálózat teremti kapcsolatot (lokális, globális és regionális útvonalak segítségével, melyeket konfigurálható kapcsolók állítanak be)
- **DCM:** Digitális órajel menedzselő áramkör, amely képes a külső bejövő órajelből tetszőleges fázisú és frekvenciájú belső órajel(ek) előállítására



# FPGA – „dedikált” erőforrások

**Dedikált erőforrások** a következők (amelyek az FPGA típusuktól és komplexitásuktól függően nagy mértékben változhat):

- **BRAM**: egy/két portos Blokk-RAM memóriák, melyek nagy mennyiségű ( $\sim \times 100\text{Kbyte}$  – akár  $\sim \times 10\text{Mbyte}$ ) adat/utasítás tárolását teszik lehetővé, egyenként 18K / 36 Kbites kapacitással \*
- **MULT** / vagy **DSP** Blokkok: beágyazott szorzó áramköröket jelentenek, amelyek segítségével hagyományos szorzási műveletet, vagy a DSP blokk esetén akár bonyolultabb DSP MAC (szorzás-akkumulálás), valamint aritmetikai (kivonás) és logikai műveleteket is végrehajthatunk nagy sebességgel.
- **Beágyazott/Beágyazható processzor(ok)\*\*:**
  - Tetszés szerint konfigurálható / **beágyazható** ún. **szoft-processzor** mag(ok)
    - Példa: *Xilinx PicoBlaze*, *Xilinx MicroBlaze*, *Altera Nios II* stb.
  - Fixen **beágyazott**, ún. **hard-processzor mag(ok)**
    - Példa: *IBM PowerPC 405/450* (*Xilinx Virtex 2 Pro*, *Virtex-4 FXT*, *Virtex-5 FXT*), **ARM Cortex A9** (*Xilinx Zynq*, illetve *Altera Cyclone V*, *Stratix V*, *Arria V*, *MicroSemi Smartfusion-1,-2 FPGA chipjei*), ) stb.

\* FPGA függő adatok (AMD Xilinx)

# FPGA – „dedikált erőforrások”

Általános erőforrások mellett a további **dedikált erőforrások** a következők (amelyek száma és felépítése az FPGA típusától és komplexitásától függően akár nagy-mértékben is változhat):

- **BRAM:** egy/két-portos Blokk-RAM memóriák, melyek összessége nagy mennyiségű ( $\sim \times 100\text{Kbyte}$  – akár  $\sim \times 10\text{Mbyte}$ ) adat/utasítás tárolását teszik lehetővé
- **MULT** / vagy **DSP** Blokkok: beágyazott szorzó áramköröket jelentenek, amelyek segítségével hagyományos szorzási műveletet, vagy a DSP blokk esetén akár bonyolultabb DSP MAC (szorzás-akkumulálás), valamint aritmetikai (kivonás) és logikai műveleteket is végrehajthatunk, nagy sebességgel
- **Beágyazott/Beágyazható processzor(ok):**
  - Tetszés szerint konfigurálható / beágyazható ún. *soft-processzor* mag(ok)
    - Példa: *Xilinx PicoBlaze*, *Xilinx MicroBlaze*, *Altera Nios II stb.*
    - **ARM** Cortex M1/M3 (licenzelt soft-core magok)
  - Fixen beágyazott, ún. *hard-processzor* mag(ok)
    - Példa: *IBM PowerPC 405/450* (*Xilinx Virtex 2 Pro*, *Virtex-4 FXT*, *Virtex-5 FXT*), **ARM** Cortex A9/A53/A72 (pl: *Xilinx Zynq*, illetve *Intel Cyclone V*, *Stratix V*, *Arria V SoC*) stb.

# FPGA létjogosultsága?

A mai modern FPGA-k a

- nagyfokú flexibilitásukkal,
- nagy számítási teljesítményükkel,
- és (ASIC-el szemben) gyors prototípus-fejlesztési,
- ezáltal olcsó kihozatali (piacra kerülési) költségükkel  
igen jó alternatívát teremtenek a mikrovezérlős (uC/MCU), illetve  
DSP-alapú implementációk kiváltására (pl. jelfeldolgozás, hálózati  
titkosítás, beágyazott rendszerek, stb. alkalmazásai területén).

Fejlődésüket jól tükrözi a mikroprocesszorok és az FPGA áramköri technológia fejlődési üteme között fennálló nagyfokú hasonlóság a méretcsökkenésnek (scaling-down) - *Gordon Moore-törvénynek* megfelelően.

# Pannon Egyetem - VIRT tanszéken lévő fejlesztő kártyák

Laborokon használt HW eszközök:

- **Digilent ZYBO (Xilinx Zynq)**
- Digilent ZED (Xilinx Zynq)
- **Digilent Nexys-2 (Xilinx Spartan-3E)**
- Digilent Atlys (Xilinx Spartan-6)
- Xilinx ML506 (Virtex-5)
- Xilinx VirtexII-Pro (VirtexII-Pro)
- Celoxica RC203/RC200 (Xilinx Virtex-II)

Laborokon használt SW eszközök:

- **Xilinx Vivado+VITIS 2020.2**
- Xilinx ISE Design Suite 14.7

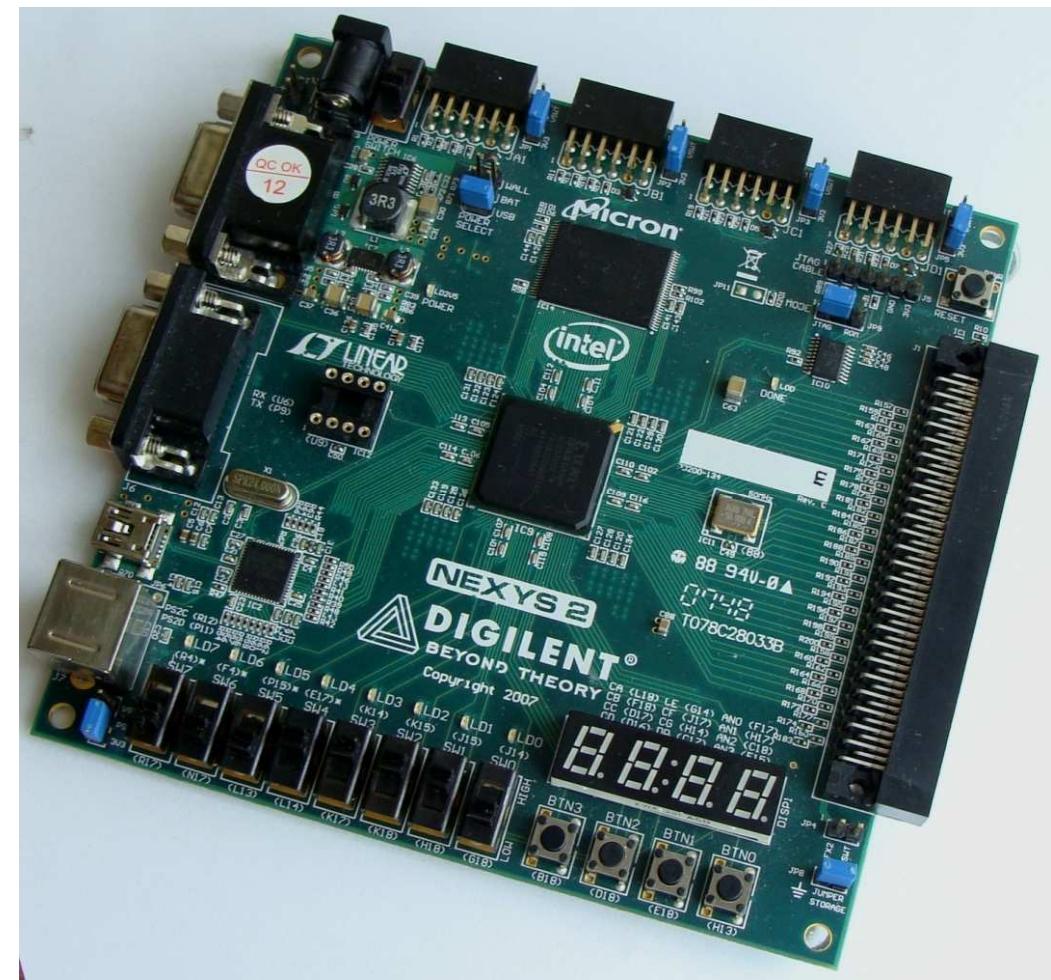
# „FPGA” témájú tárgyak a Pannon Egyetemen – VIRT tanszék (2024)

- **(VEMIVIB334TM) Tervezési módszerek programozható logikai alkatrészekkel (VHDL) – őszi félév**
  - <http://virt.uni-pannon.hu/index.php/hu/oktatas/tantargyak/170-tervezesi-modoszerek-programozhato-logikai-eszkoezoekkel-vemivib544t>
- **(VEMIVIB334BR) FPGA-alapú beágyazott rendszerek – tavaszi félév**
  - <http://virt.uni-pannon.hu/index.php/hu/oktatas/tantargyak/195-fpga-alapu-beagyazott-rendszerek>

# Digilent Nexys-2 fejlesztő kártya

## Nexys™2 Xilinx Spartan-3E FPGA fejlesztő kártya

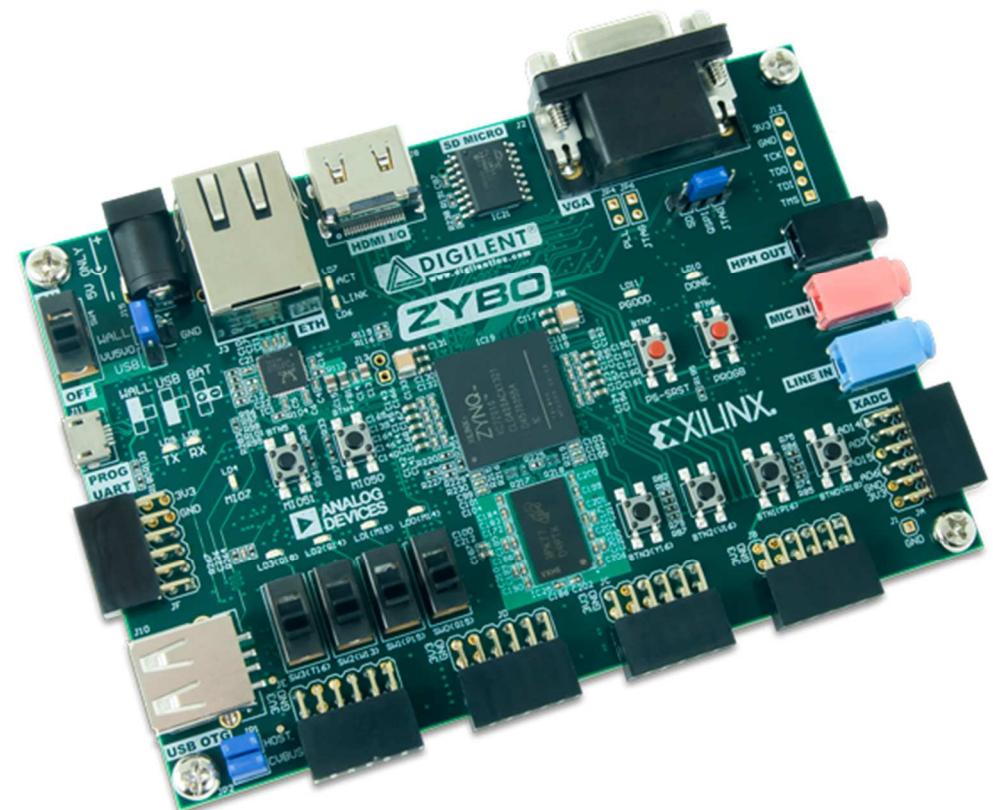
- Xilinx Spartan-3E FPGA, 500 000 / 1 200 000 ekvivalens kapuval
- USB2 port (táp, konfiguráció, adat-transzfer egyben)
- Xilinx ISE/Webpack/EDK
- 16MB Micron PSDRAM
- 16MB Intel StrataFlash Flash
- Xilinx Platform Flash ROM
- 50MHz osszcillátor
- 75 FPGA I/O's (1 nagy-sebességű Hirose FX2 konnektor és 4 db 2x6 PMOD konnektor)
- GPIO: 8 LED, 4-jegyű 7-szegmenses kijelző, 4 nyomógomb, 8 kapcsoló
- VGA, PS/2, Soros port



# Digilent ZYBO fejlesztő kártya

## ZYBO™ Zynq FPGA fejlesztő kártya

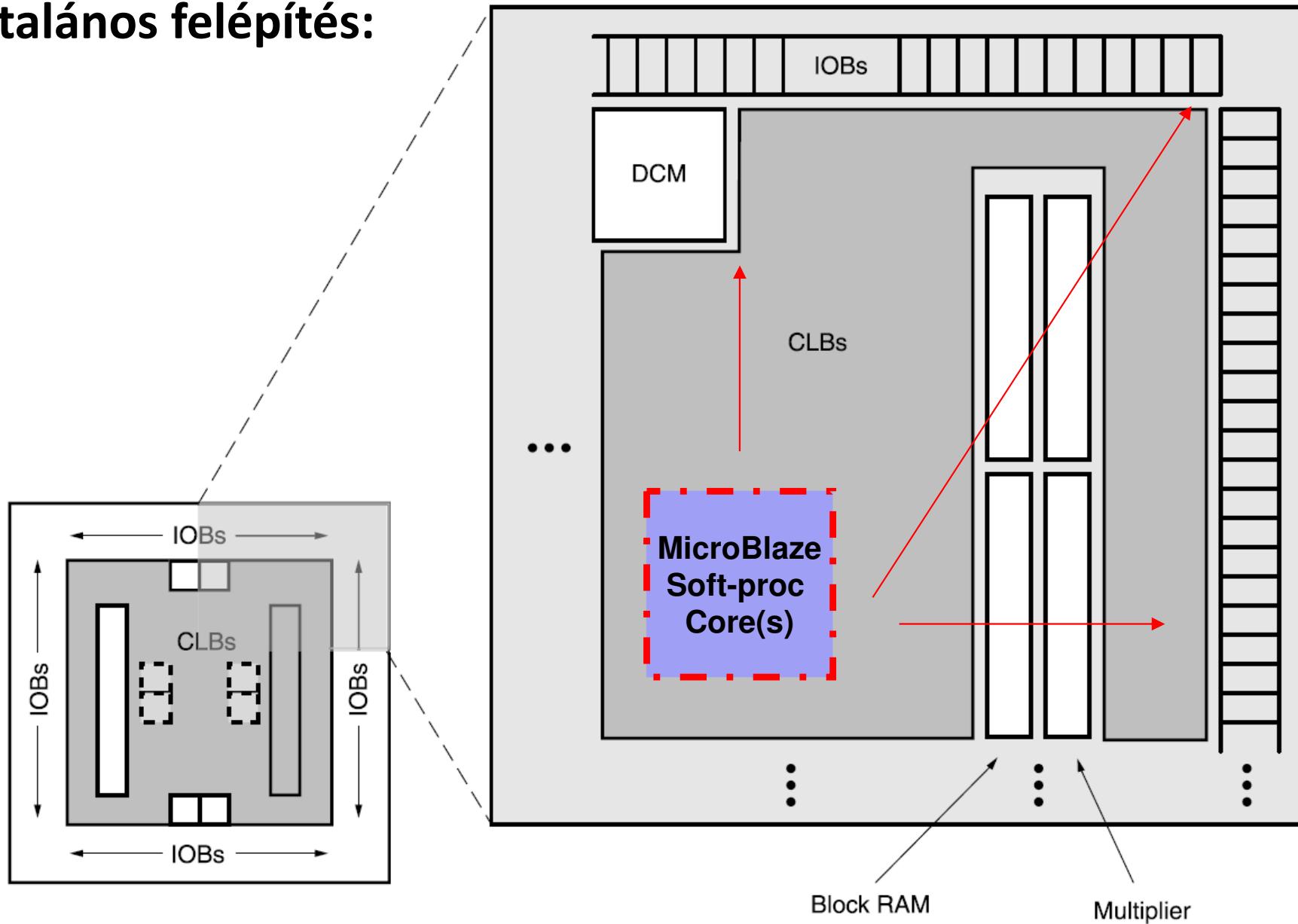
- Xilinx Zynq-7000 (Z-7010)
- 650 MHz dual **ARM Cortex-A9** magok (PS)
  - Harvard , RISC 32-bites processzor
  - 8-csatornás DMA vezérlő (PS)
  - 1G ethernet, I2C, SPI, USB-OTG vezérlő (PS)
  - Artix-7 FPGA logika (PL)
  - 28Kbyte logikai cella, 240 Kbyte BRAM, 80 DSP szorzó(PL)
  - 12-bites, 1MSPS XADC (PL)
- 512 Mbyte DDR3 x32-bit (adatbusz), 1050Mbps sávszélességgel
- Tri-mode 10/100/1000 Ethernet PHY
- HDMI port: Dual role (source/sink)
- VGA port: 16-bites
- uSD kártya: OS tartalom tárolása
- OTG USB 2.0 (host és device)
- Audio codec
- 128Mbit x Serial Flash/QSPI (konfiguráció tárolási célokra)
- JTAG-USB programozhatóság, UART-USB vezérlő
- GPIO: 5 LED, 6 nyomógomb, 4 kapcsoló
- 4+1 PMOD csatlakozó (A/D átalakítóhoz)





# Pl. Spartan-3E architektúra

Általános felépítés:



# Spartan-3E FPGA építőelemei

- CLB: Konfigurálható Logikai blokkok
    - Slice, mint *logika*: LUT, D-FF, MUX, Carry Logika
    - Slice, mint *mémória*: SRL-16×1, RAM-16×1
  - IOB: I/O blokkok
  - DCM: Digitális órajel menedzser blokkok
  - Programozható kapcsolók/ összeköttetések
- 
- MULT: 18×18 bites előjeles (2's) szorzó(k)
  - BRAM: konfigurálható 18 Kbites (~2 Kbyte + paritás) egy/két-portos memóriák
  - Beágyazható processzor(ok):
    - Csak szoft-processzor mag(ok) alakíthatóak ki
      - PI: Xilinx MicroBlaze, PicoBlaze, (esetleg külső független IP – szellemi terméke integrálható)

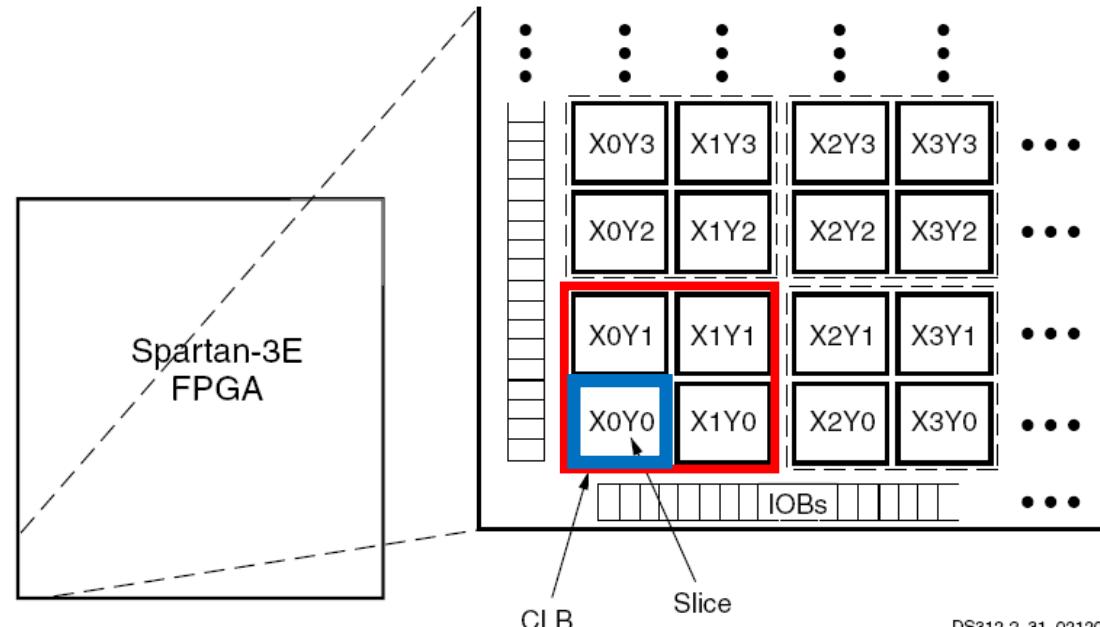
Általános erőforrások

Dedikált erőforrások

# Spartan-3E CLB tömb

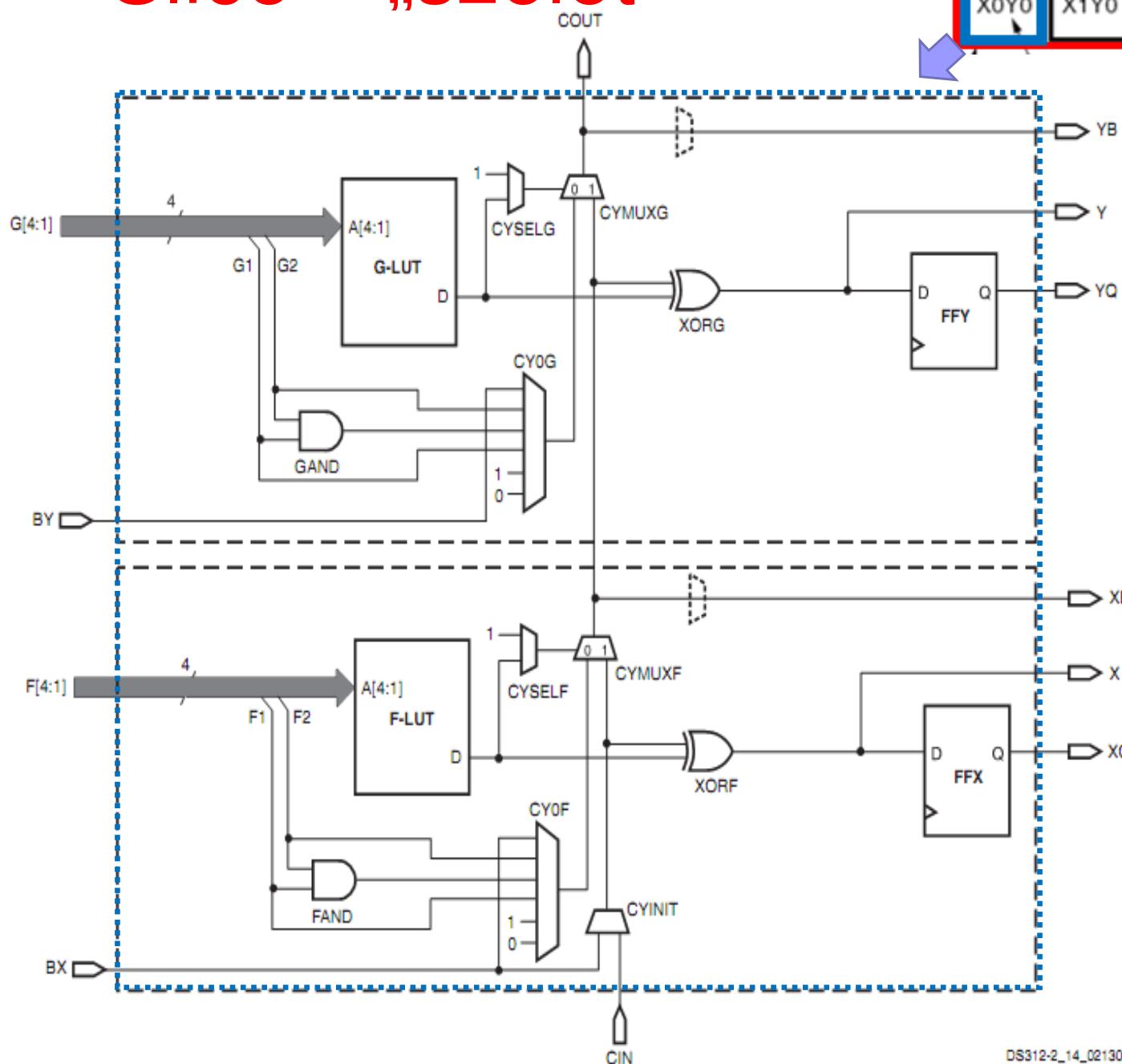
**CLB:** Konfigurálható Logikai Blokkok: fő logikai erőforrás, kombinációs és szekvenciális logikai hálózatok tervezésére

**1 CLB = 4 Slice !**



Device	CLB Rows	CLB Columns	CLB Total <sup>(1)</sup>	Slices	LUTs / Flip-Flops	Equivalent Logic Cells	RAM16 / SRL16	Distributed RAM Bits
XC3S100E	22	16	240	960	1,920	2,160	960	15,360
XC3S250E	34	26	612	2,448	4,896	5,508	2,448	39,168
XC3S500E	46	34	1,164 * <sub>4</sub> = 4,656 * <sub>2</sub> =	9,312	10,476	4,656 * <sub>16</sub> =	74,496	
XC3S1200E	60	46	2,168 * <sub>4</sub> = 8,672 * <sub>2</sub> =	17,344	19,512	8,672 * <sub>16</sub> =	138,752	
XC3S1600E	76	58	3,688	14,752	29,504	33,192	14,752	236,032

# Slice – „szelet”



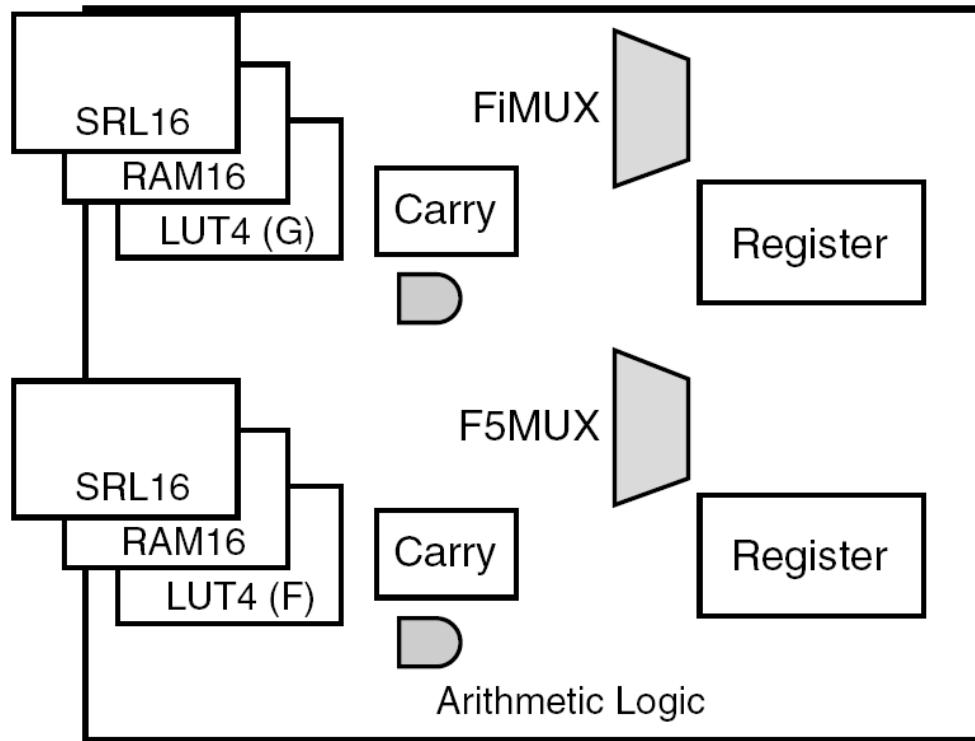
- Look-Up Table konfigurálható:
  - 4-bemenetű LUT-ként (F,G)
  - 16x1-bit szinkron RAM-ként
  - 16-bit shift regiszterként
- Tároló elemek
  - D-típusú flip-flop-ok, vagy latch-ek
- További Logikai áramkörök
  - F5MUX multiplexer
    - bármely 5-bemenetű függvény
    - 4:1 multiplexer
  - FiMUX multiplexer
    - bármely 6-bemenetű függvény
    - 8:1 multiplexer
- Aritmetikai Logikai Egység
  - Dedikált carry logika (CYSEL\_, CYMUX\_, CY0\_)
  - Dedikált AND kapuk (GAND, FAND)

FPGA-k esetében egy alapvető logikai mérőszám a „slice” – szelet.

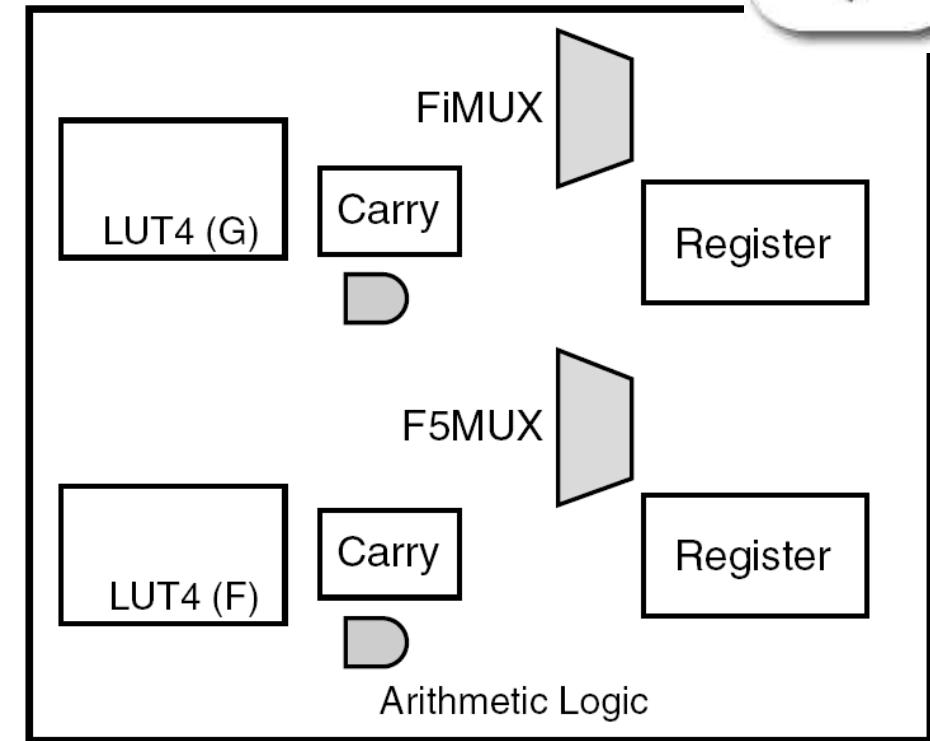
# Milyen erőforrásként konfigurálhatók a Slice-ok?



1 CLB = 4 slice (Spartan 3E esetén) = 2-2 SLICEM-SLICEL pár



**SLICEM**



**SLICEL**

DS312-2\_13\_020905

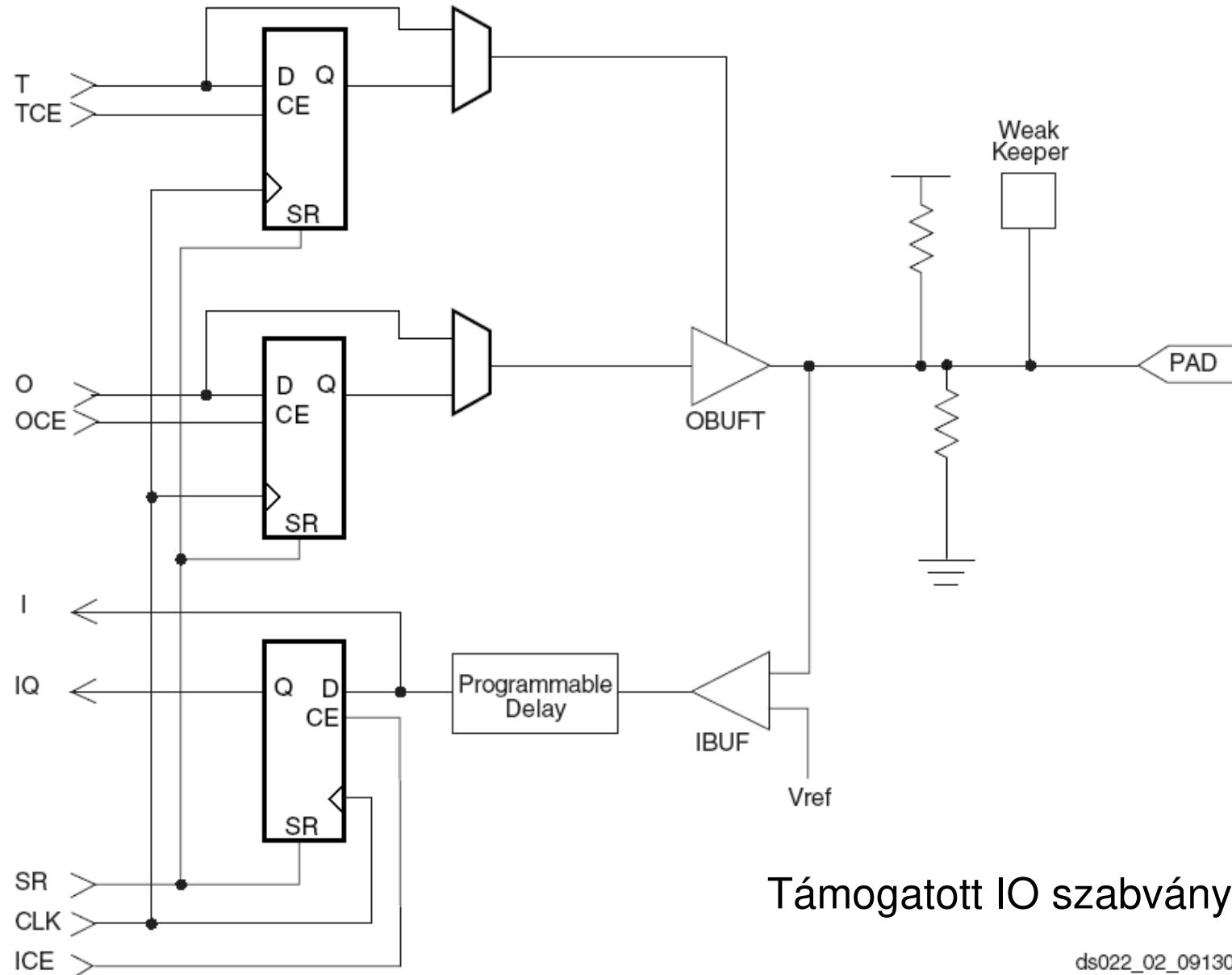
**Slice”M”:** M, mint memória elem lehet:

- A.) *LUT-4*: Logika 4-bemenetű
- B.) *RAM16*: Distributed (elosztott) RAM – regiszterekből 16x1-bit,
- C.) *SRL16*: Shift Regiszter 16x1-bit

**Slice”L”:** L, mint logika lehet:

- -.) Csak *LUT-4*!
- D.) Dedikált *MUX*
- E.) *Carry logika*

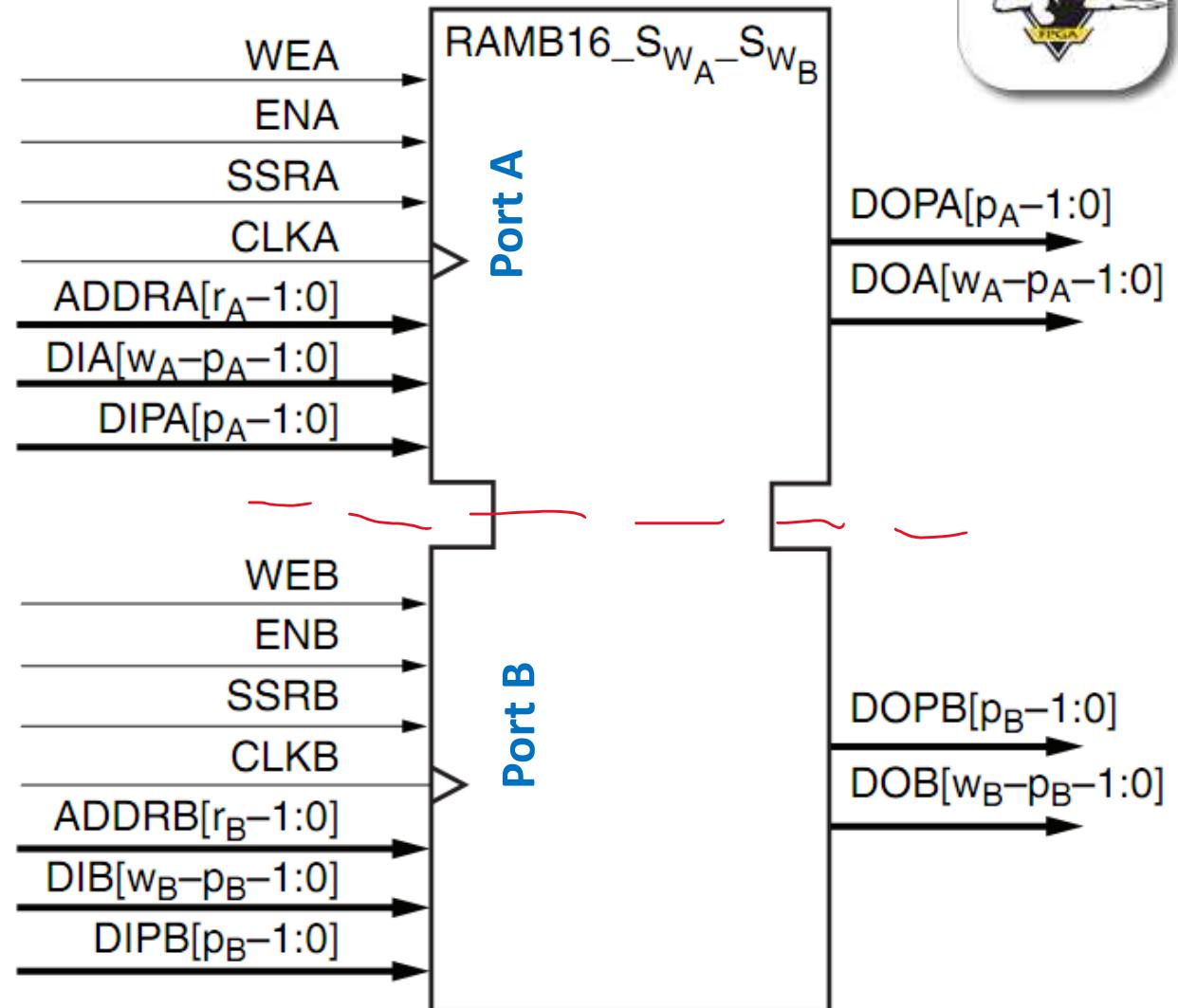
# IOB – Programozható I/O Blokkok



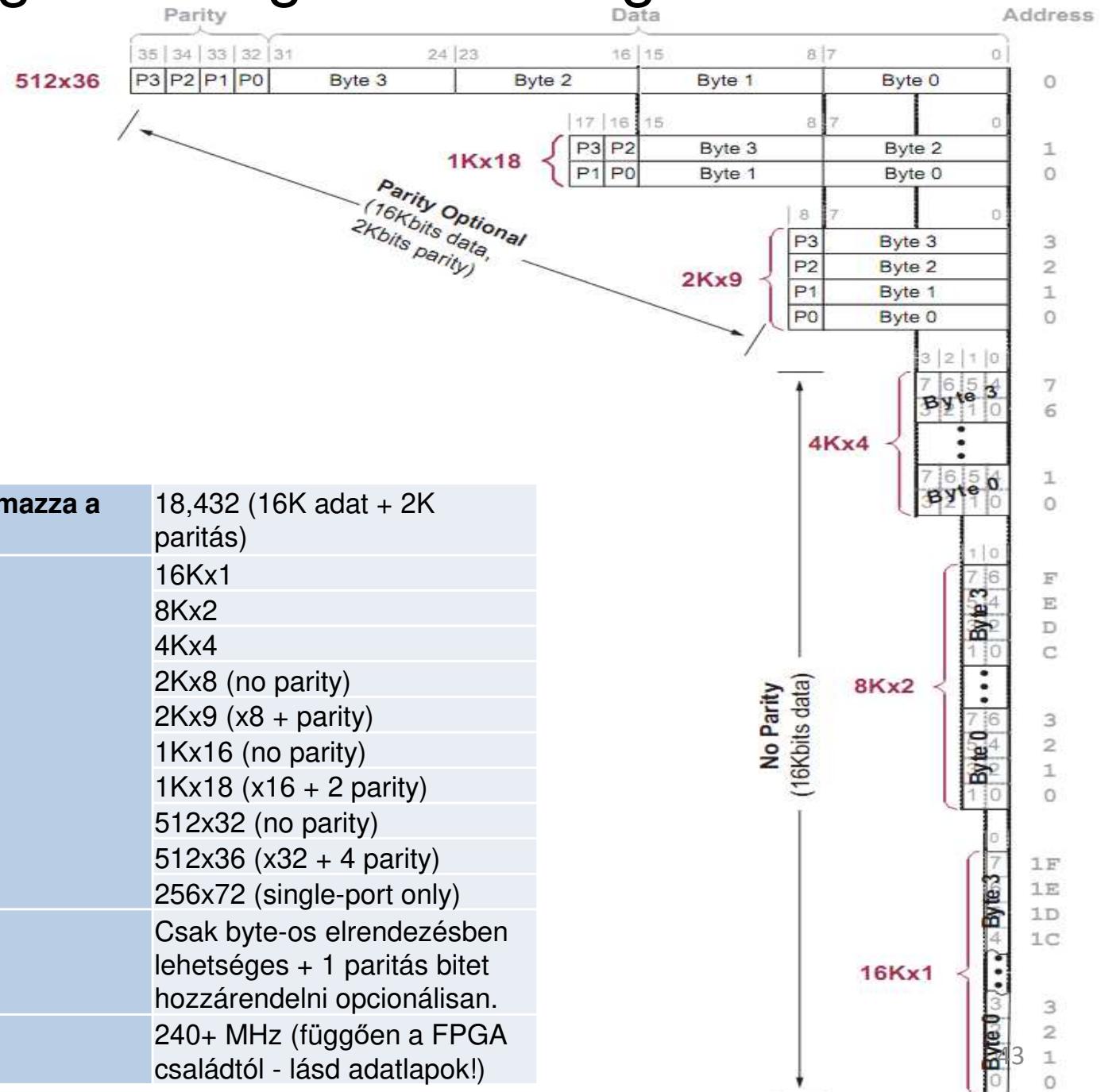
Támogatott IO szabványok (~40)

# Blokk-RAM

- „SelectRAM” tulajdonság (konfigurálható)
  - FIFO, RAM, ROM-ként is
- Dedikált BRAM lehet:
  - Egy-portos
  - Két-portos (A,B)
    - Minkét portját (A, B) külön címezhetjük (RW)
    - Független adatbusz szélesség definiálható



# BRAM tetszőleges konfigurálhatósága



**Teljes RAM kapacitás (bit), mely tartalmazza a paritásokat is**

18,432 (16K adat + 2K paritás)

**Memória szervezési módszerek:**

- 16Kx1
- 8Kx2
- 4Kx4
- 2Kx8 (no parity)
- 2Kx9 (x8 + parity)
- 1Kx16 (no parity)
- 1Kx18 (x16 + 2 parity)
- 512x32 (no parity)
- 512x36 (x32 + 4 parity)
- 256x72 (single-port only)

**Paritás:**

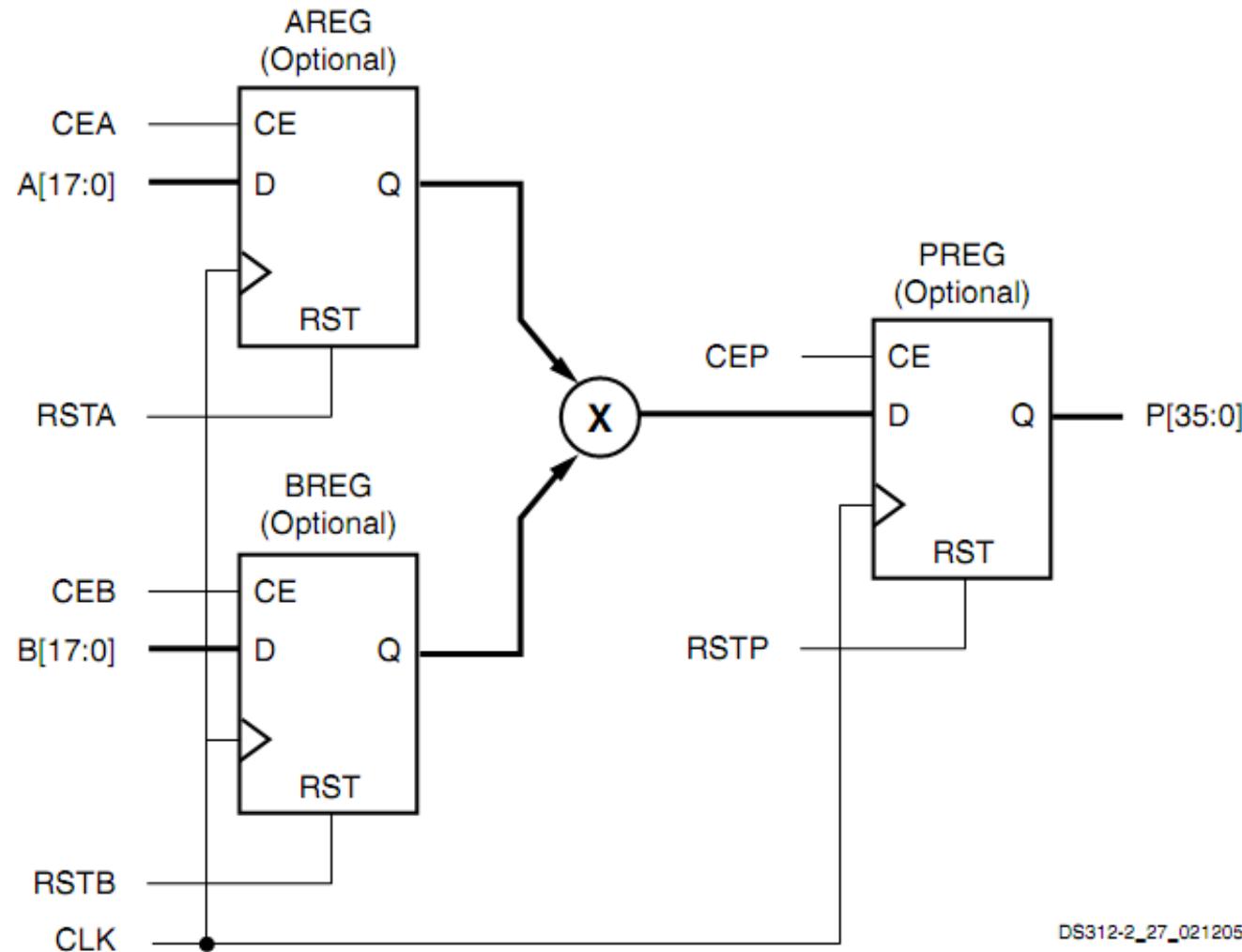
Csak byte-os elrendezésben lehetséges + 1 paritás bitet hozzárendelni opcionálisan.  
240+ MHz (függően a FPGA családtól - lásd adatlapok!)

**Sebesség:**

# Multiplier - Szorzó



- $P = A^*B$  dedikált szorzó áramkör
  - 18×18-bites, 2-komplemens szorzást támogat
  - Opcionális szorzó-, szorzandó-, szorzat-regiszterek



# DCM – Digital Clock Manager

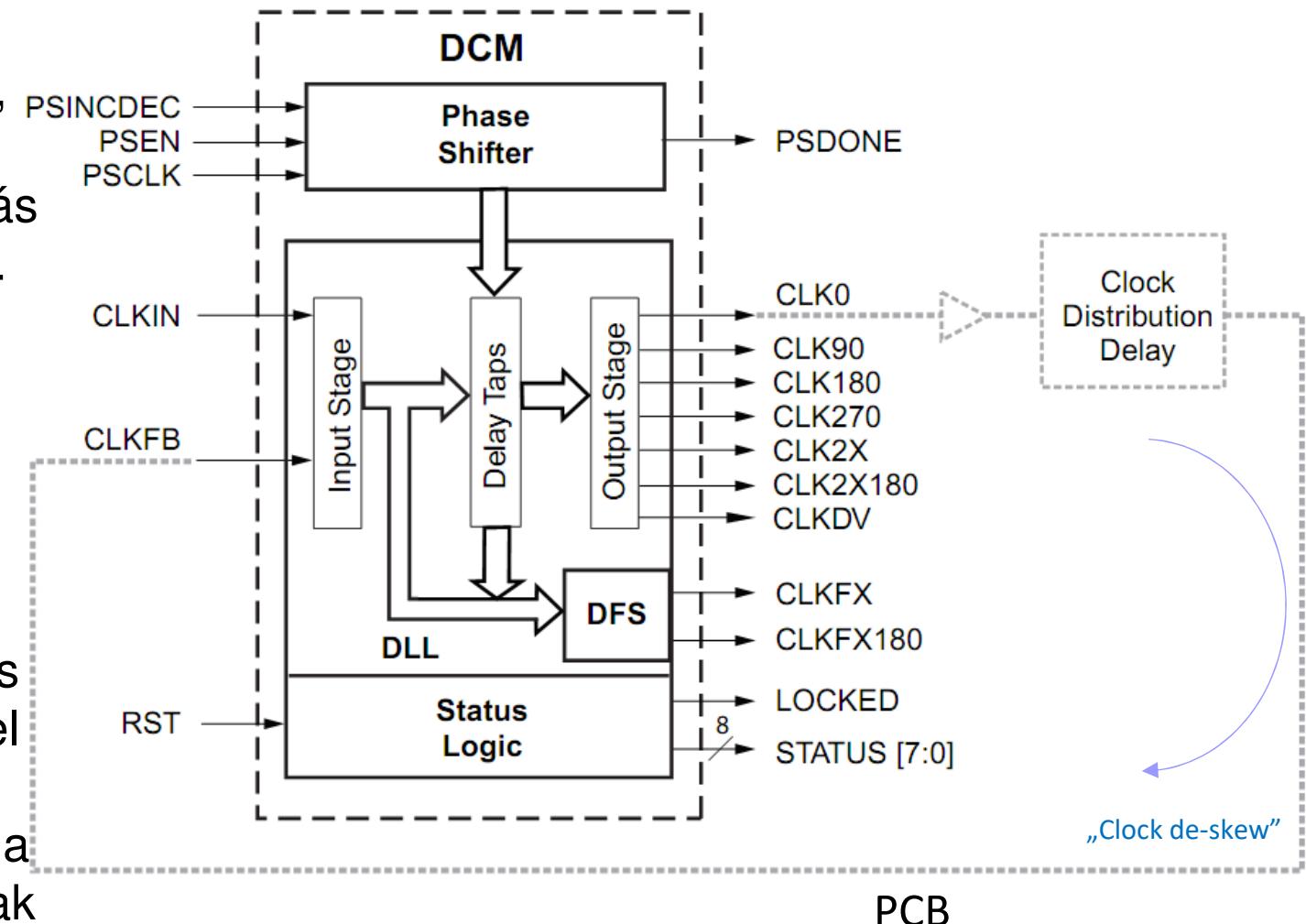
## Digitális órajel menedzser áramkör(ök)

- DCM-ek száma: 4 – 8
- **DLL:** Delayed Locked Loop
- Negyedelt fázis tolás:  $0^\circ$ ,  $90^\circ$ ,  $180^\circ$ ,  $270^\circ$
- Órajel szorzás (M)/ osztás (D) 1.5, 2, 2.5, 3, 4, 5, ... 16

- Tetszőleges órajel generálható
- 5 MHz – 333 MHz

**DFS:** Digitális Frekvencia szintézis

- Órajel duplázás / felezés
- Bemeneti, kimeneti órajel pufferelés
- Locked: DCM kimenetei a CLKIN-el fázisban vannak



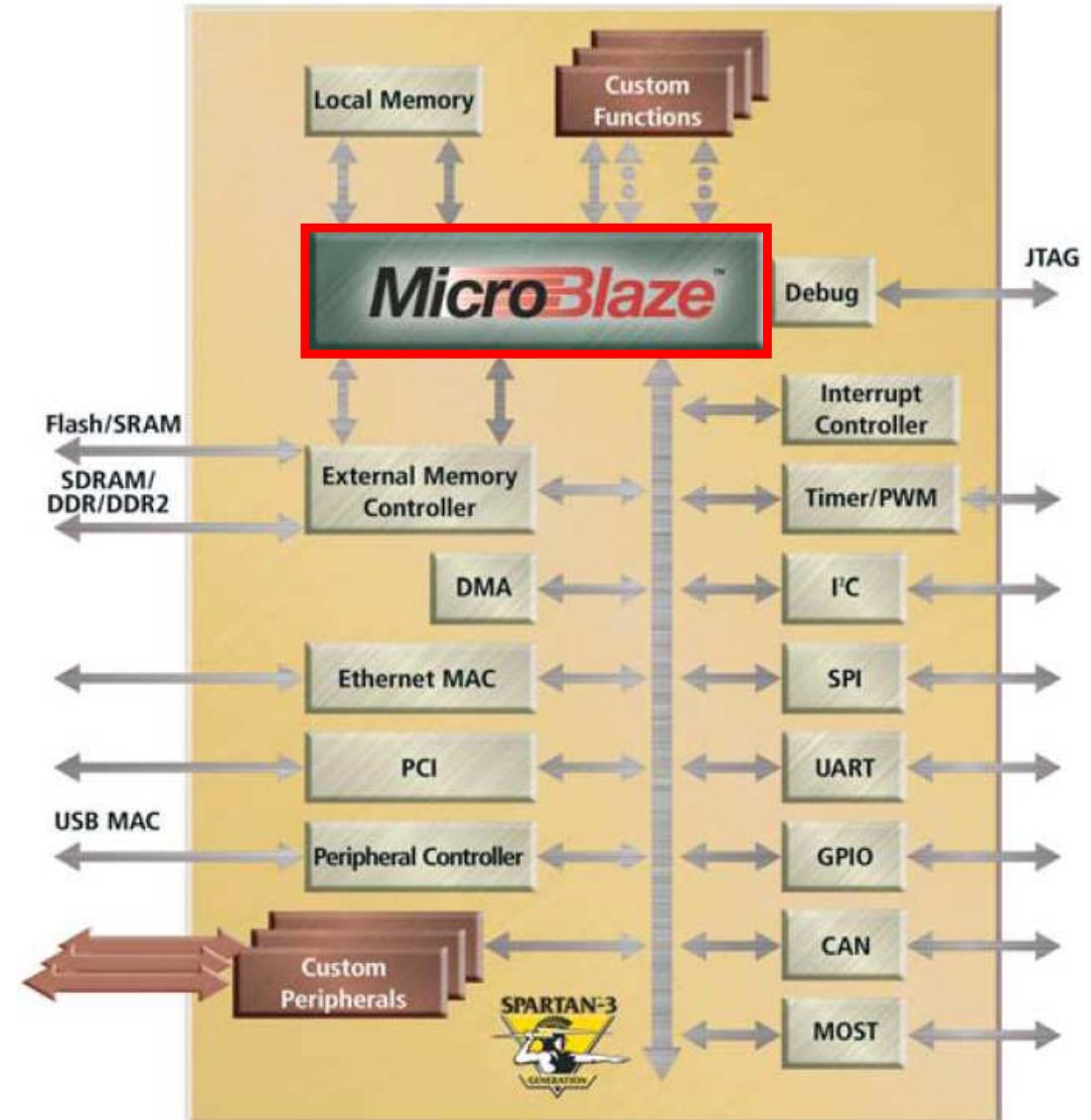
# Beágyazott/Beágyazható processzorok

- „Beágyazható” (lágynak írt) **soft-processzor** magok:
  - Xilinx *PicoBlaze*: 8-bites (VHDL, Verilog HDL forrás)
  - Xilinx *MicroBlaze*: 32-bites (XPS – EDK/SDK támogatás!)\*
    - PLB, OPB (régi), AXI buszrendserekhez is csatlakoztatható
  - 3rd Party: nem-Xilinx gyártók processzorai
    - Pl. ARM Cortex M1/M3 (licenzelt soft-core magok)
- „Beágyazott” (keményen írt) **hard-processzor** magok:
  - Korábban IBM *PowerPC* 405/450 processzor (dedikált): 32-bites (EDK/SDK támogatás), PLB buszrendszerhez integrálható
    - de kizárolag Virtex II Pro, Virtex-4 FX, Virtex-5/6 FXT FPGA-kon!
  - *Jelenleg*: ARM Cortex-A9/A53/A72 processzor (dedikált): kétmagos, AMBA-AXI buszrendszerhez integrálható (32/64 bites)
    - Xilinx Zynq APSoC: FPGA logikához integrált ARM magok

# MicroBlaze magra épülő beágyazott rendszer

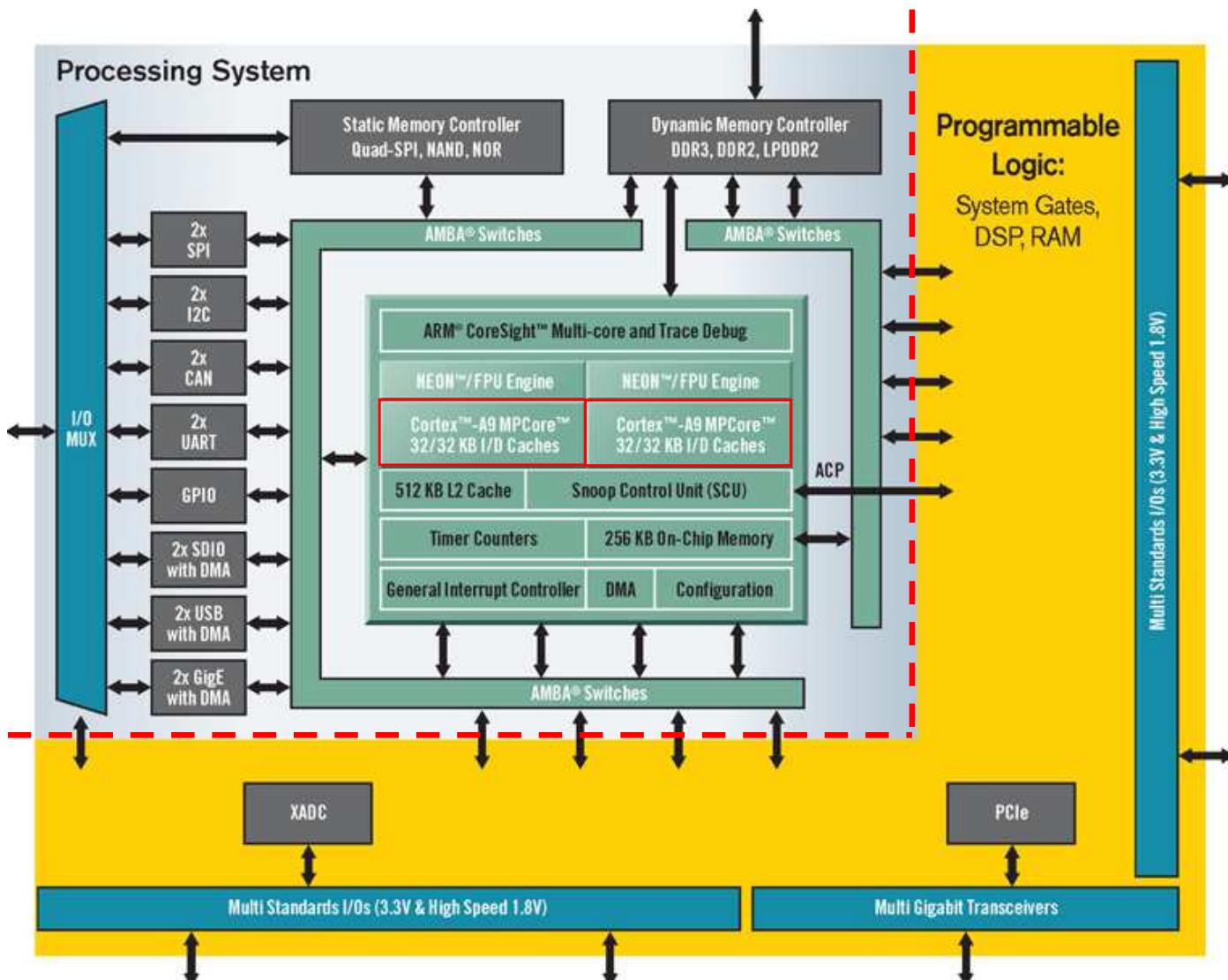
## „Beágyazható” processzor

- **RISC** utasítás készlet architektúra
- 32-bites **soft-processor** core!
- 133+ MHz órajel (PLB busz)
- **Harvard memória-architektúra**
- Kis fogyasztás: ~ mW/MHz
- 3/5 lépcsős adatvonal pipeline
- 32 darab 32-bites Általános célú regiszter
- utasítás cache / adat cache
- Időzítési lehetőségek (timer)
- Sokféle periféria, kommunikációs interfész csatlakoztatható (IP core-ok)
- **Minden** Xilinx FPGA-n implementálható, melynek elegendő erőforrása van és a fejlesztő szoftver támogatja!



# Xilinx Zynq APSoC – ARM hard-processzor

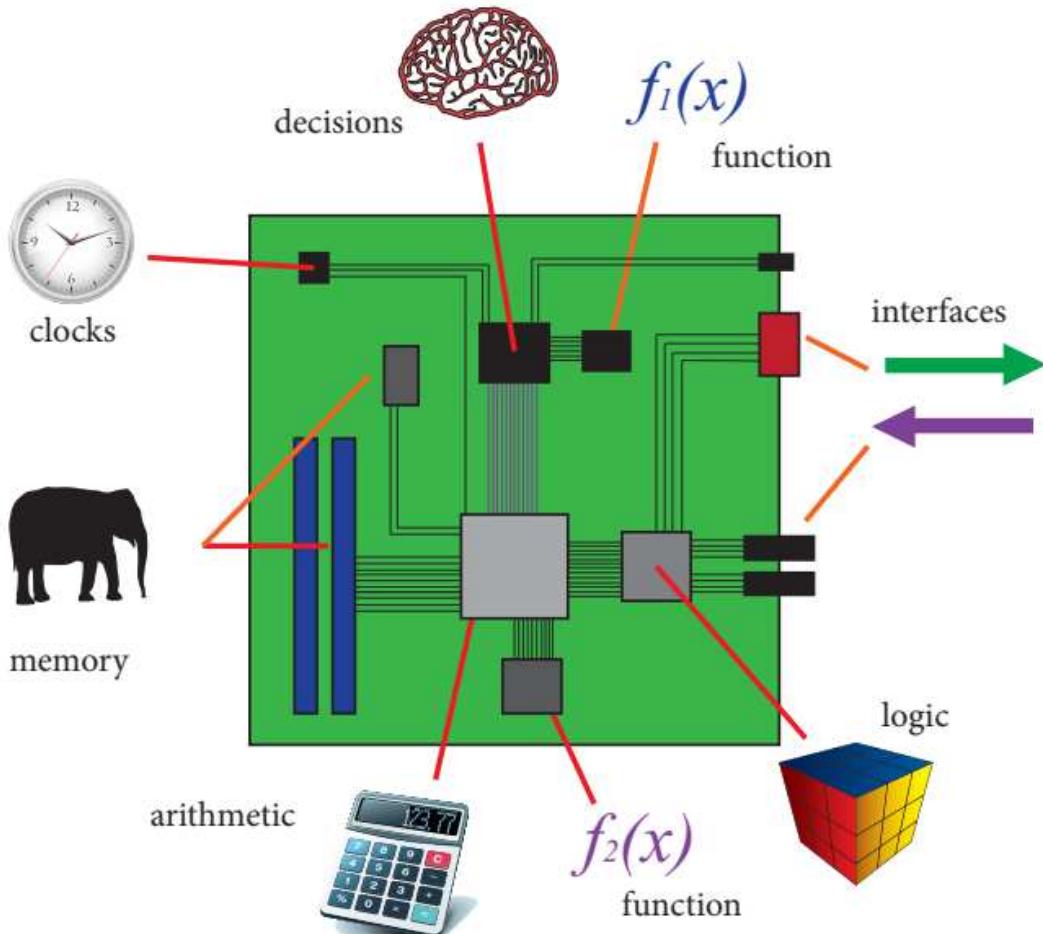
**ARM**



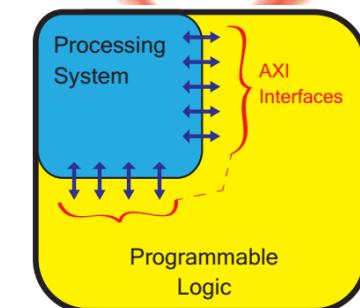
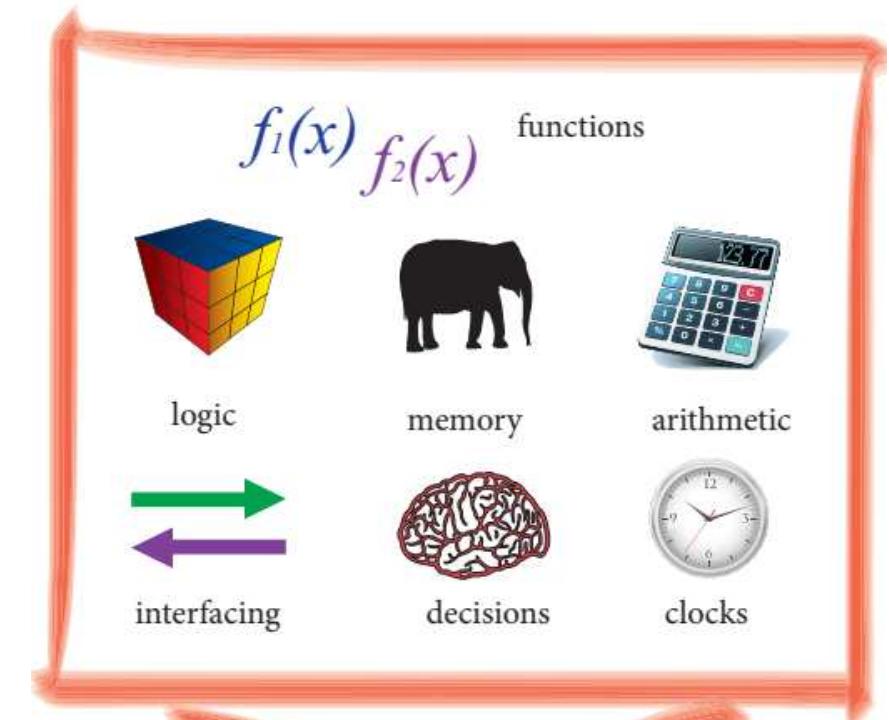
- Dupla ARM Cortex™-A9 MPCore (800MHz)
- Beágyazott RISC, Harvard
- Neon: 32-/ 64-bit FPU
- 32kB utasítás & 32kB adat L1 Cache
- Közös 512kB L2 Cache
- 256kB on-chip memória
- Integrált DDR3, DDR2 and LPDDR2 DDR vezérlő
- Integrált 2x QSPI, NAND Flash and NOR Flash memória vezérlő
- Perifériák: 2x USB2.0 (OTG), 2x GbE, 2x CAN2.0B 2x SD/SDIO, 2x UART, 2x SPI, 2x I2C, 4x 32b GPIO, PCI Express® Gen2 x8
- Két 12-bit 1Msps ADC
- 28nm Programozható FPGA Logika:
  - 28k - 350k Logikai cella (~ 430k to 5.2M ekvivalens kapu)
  - 240KB - 2180KB Block RAM
  - 80 - 900 18x25 DSP szorzó (58 - 1080 GMACS -DSP teljesítmény)

■ **FPGA → APSoC = All Programmable System on a Chip**

# System-On-a-Board vs. System-On-a-Chip



VS.



**Zynq  
APSoC**



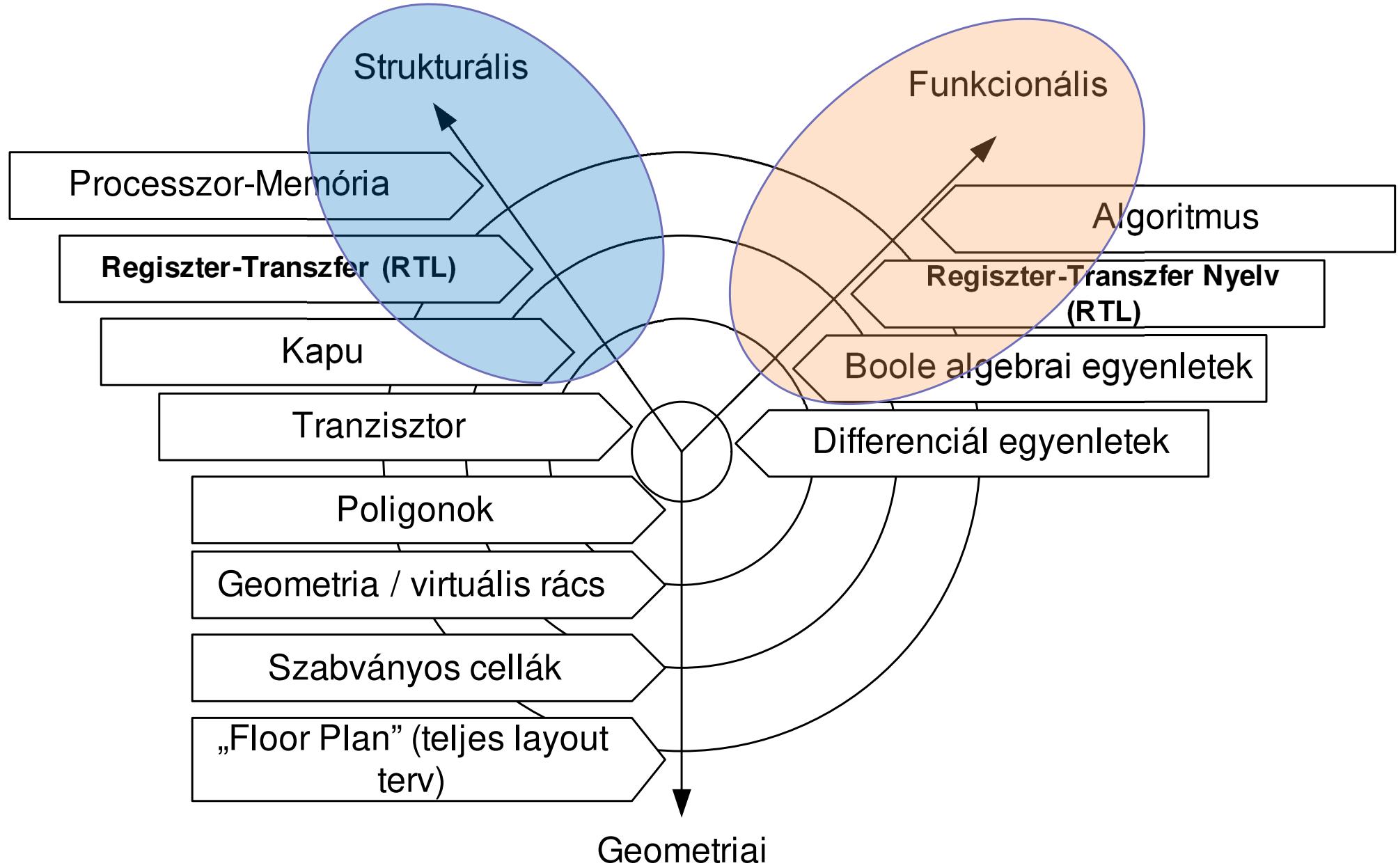
Szempontok, absztrakciós szintek, modellezés

## 2. TERVEZÉSI MÓDSZEREK

# Tartományok – absztrakciós szintek

- Különböző tervezési szempontok (metodika) szerint dolgozhatunk, azaz a rendszer modellezését különféle **tartományokon** és azokon belül eltérő **absztrakciós szinteken végezhetjük**.
- Az ismert *Gajski és Kuhn féle Y-diagramm* szerint **3 lehetséges tervezési tartományt** különböztethetünk meg:
  - Funkcionális/viselkedési,*
  - Strukturális* és
  - Geometriai.*(a HDL leírások értelmezése során a *viselkedési* és a *strukturális* tartomány szerinti tervezést alkalmazzuk)

# Gajski-Kuhn „Y-diagramm”



# I. Funkcionális tartomány

A rendszer **viselkedésének** leírását adjuk meg, de nem foglalkozunk annak belső részleteivel (felépítésével).

■ A funkcionális tartomány a 'leg-absztraktabb' megközelítést jelenti, mivel a teljes rendszer viselkedése megadható az **algoritmikus szinten**.

■ A következő szint a **regiszter-átviteli szint (Register-Transfer)** vagyis a *regiszter-memória-processzor* elemek közötti transzformációk megadása. Az adatot egy regiszter, vagy egy memória adott cellájának értéke, míg a transzformációkat az aritmetikai és logikai operátorok jellemezhetik. Az adat és vezérlési információ áramlása definiálható akár a regiszter transfer szintű nyelvi leírásokkal (RT Language) is, vagy hatékonyan szemléltethetők grafikus módon az **adat-, és vezérlési-folyam gráfok segítségével (DFG, CFG)**.

■ A harmadik szint a hagyományos logikai szint, ahol egy-egy funkció megadható a **Boole-algebrai** kifejezések, igazság táblázatok segítségével.

■ Végül az utolsó szinten az áramkör működését definiáló **differenciál-egyenleteket** kell definiálni, amely a hálózat feszültségében, és áramában történő változásokat hivatott leírni.<sup>53</sup>

## II. Strukturális tartomány

A **strukturális tartományban** viszont pontosan a rendszer *belső elemeinek felépítését*, és azok között lévő összeköttetéseket vizsgáljuk.

- A strukturális tartomány legfelső szintjén kell a fő komponenseket és összeköttetéseit definiálni, amelyet **processzor-memória kapcsolatnak** (**processor-memory switch**) is hívnak.
- A második szint itt is a **regiszter-átviteli szint**, amely egyrészt az adatútból (data path), másrészt vezérlési szakaszokból (control path, sequence), szekvenciákból áll.
- A harmadik szint a **logikai szint**, amelyben a rendszer struktúrája, az alapkapuk, és összeköttetései segítségével építhető fel.
- A negyedik, legalacsonyabb szinten a **tranzisztorok**, mint az *áramköri rajzolatok* (*layout*) elemi egységeit kell tekinteni (azokból építkezni).

# III. Geometriai tartomány

Végül a **geometriai tartomány** azt mutatja, ahogyan a rendszert elhelyezzük, leképezzük (=MAPPING) a rendelkezésre álló fizikai erőforrások felhasználásával (*felület*).

- A legfelső hierarchia szinten, a *szilícium felületén elhelyezett*, vagy ún. „kiterített” VLSI áramkört kell tekinteni (**floor-plan**): FPGA esetén tehát magukat a *logikai cellákat* és *makrocellákat*, valamint a közöttük lévő összeköttetéseket. (lásd FPGA felépítés)
- A következő szintet a **szabványos alapcellák (Standard Cell)** könyvtárai képezhetik, amelyeket, mint *technológiai adatbázist* használhatunk fel a regiszterek, memóriák, vagy akár aritmetikai-logikai egységek megvalósításához.
- A harmadik szinten az egyedi tervezésű integrált áramkörök (**ASIC**) **geometriája** egy virtuális rácson adható meg.
- Végül az utolsó, legalacsonyabb szinten a **poligonokat** kell megrajzolni, amelyek csoportjai az áramkör maszk-rétegeinek feleltethetők meg.

# **EDA**: Electronic Design Automation

- Manapság a számítógéppel segített elektronikus tervezői, fejlesztői és szimulációs eszközök széles skálája áll a rendelkezésünkre.
- Segítségükkel egy-egy tartományon belül nem kell minden szintet külön-külön pontosan definiálni (sokszor nem is lehet), elegendő a **tervezést a legfelsőbb absztrakciós szinteken** elvégezni (*design entry*), → amelyből az alacsonyabb szintek automatikusan generálódnak (**EDA**).

# PLD/ FPGA: tárgyalt ismeretkörök

1. Mik azok a.) Programozható Logikai Eszközök és az b.) FPGA-k?
  - Összeköttetések programozhatósága
2. Tervezési módszerek (Design methods)
3. **Tervezés folyamata (Design-flow)**
4. Magas-szintű szintézis (HLS – High-Level Synthesis)
5. Fejlesztő környezetek, hardver leíró nyelvek (HDL - Hardware Description Languages)

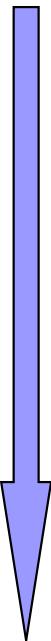


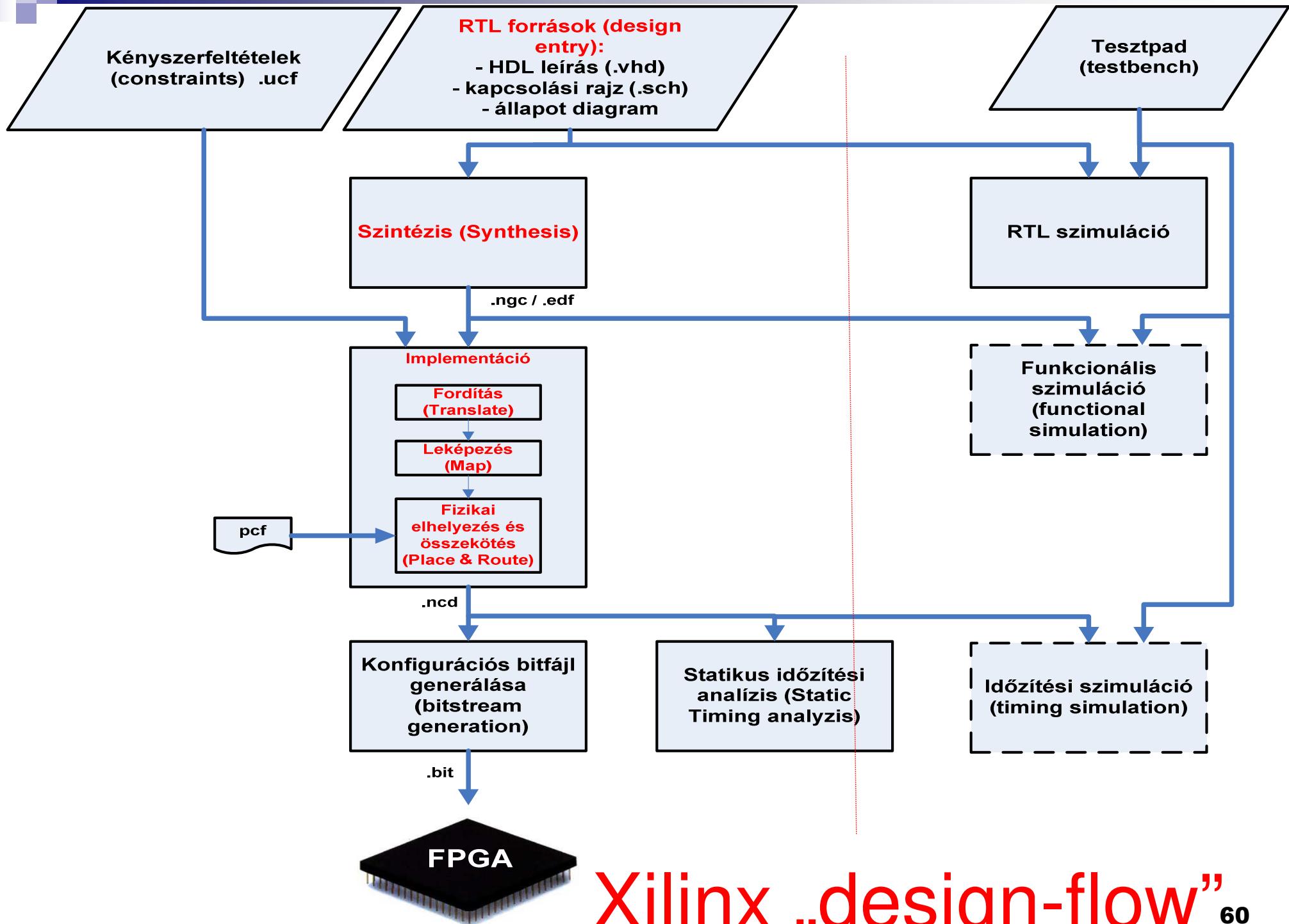
„DESIGN FLOW„ (XILINX FPGA-k esetében)

### **3. TERVEZÉS FOLYAMATA**

# Xilinx „design-flow”

- Az FPGA alapú rendszertervezés folyamatát a Xilinx fejlesztő környezetben történő egymást követő **lépések**en keresztül demonstráljuk. A fejlesztés egyszerűsített folyamatát a következő oldalon lévő ábra szemlélteti. Lépei:
  - Design Entry (pl. HDL forrás)
  - 1.) Szimuláció
  - 2.) Szintézis és
  - 3.) Implementáció
  - 4.) Időzítési analízis
  - 5.) Bitfolyam (konfiguráció) létrehozása





Xilinx „design-flow”<sup>60</sup>

# A tervezés fontosabb lépései (I.):

- **1.) Moduláris, vagy komponens alapú (hierarchikus) rendszertervezés**
  - Lentről-felfelé (bottom-up), vagy fentről-lefelé (top-down) irányú tervezési metodika alkalmazása
  - HDL leírások, kapcsolási rajzok, vagy állapot diagramok megadása a tervezés kezdeti fázisában (=design entry), illetve
  - felhasználói-tervezési megkötések, kényszerfeltételek (user constraints - *ucf*) rögzítése (lásd később a szintézis és implementáció során),
- **2.) Szimuláció:** párhuzamosan a tervezés egyes szintjein, illetve a legfelsőbb hierarchia szinten **HDL tesztkörnyezet**, tesztágy vagy **tesztpad (testbench)** összeállítása = **szimulációs modell**
  - RTL / viselkedési szimuláció tesztvektorokkal, amely még PC-n történik

# A tervezés fontosabb lépései (II.):

## ■ 3.) Szintézis és implementáció!:

- **Szintézis:** „logikai szintézis” során a HDL leírás általános kapu-szintű komponensekké transzformálódik (pl. logikai kapuk, FFs)
- **Implementáció** 3 fő lépésből áll: **TRANSLATE (Compiler) → MAP (Fit) → PAR (Placer & Router / Assembler)** lépéseinék sorozata. Ezeket a kifejezéseket a Xilinx FPGA rendszer-fejlesztési folyamatán kívül is hasonlóan nevezik. Ha az implementáció bármely adott lépésében hiba történt, akkor az implementáció további lépései már végre sem hajtódnak. Az adott lépésben meglévő hibát először ki kell javítani – ezt az ISE üzenet ablakában megjelenített információk alapján láthatjuk - és csak utána tudjuk a további implementációs lépéseket végrehajtani.

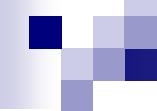
# A tervezés fontosabb lépései (III.):

## Implementációs fázisok (részletezve):

- **TRANSLATE:** több, esetleg eltérő hardver leíró nyelven megírt tervezői file összerendelése (merge) egyetlen *netlist*-fájlba (*EDIF*). A netlista fájl tartalmazza a komponensek és összeköttetéseiik szabványos szöveges leírását.
- **MAP:** az elkészült „*logikai*” tervnek egy adott technológia szerinti „*leképezése*” (*technology mapping*), amelyet az előző lépésben kapott EDIF netlistából a kapukból CLB-ket, ill. IOB-kat képez le
- **PAR:** végleges „*fizikai*” áramköri tervet hoz létre periféria kényszerfeltételeitől függően (.PCF – peripheral constraint file), amely fázisban a komponenseket az **fizikailag** az FPGA-n meglévő és azonosítható cellákon **helyezi el** (pl. XnYm). Az elhelyezett fizikai komponenseket végül a ‚*routing*’ eljárás összehuzalozza, az egyes tervezési megkötésekkel, kényszer-feltételeket figyelembe véve (.PCF). Kimenetén egy .NCD fájl jön létre.

## A tervezés fontosabb lépései (IV.):

- **4.) Statikus időzítési analízis:** időzítési paraméterek (timing parameters) meghatározása (max. órajel frekvencia, kapuk megszólalási idejének, vezetékek jelterjedési késleltetések hatásának vizsgálata stb.)
- **5.) Bit-folyam (bit-stream),** mint konfigurációs fájl generálása (.BIT) és letöltése FPGA-ra (CLB-k, programozható összeköttetések beállítása, konfigurálása minden egyes inicializációs fázis után szükséges, köszönhetően annak, hogy a Xilinx FPGA-kat alapvetően az SRAM alapú technológia jellemzi).



HLS – HIGH LEVEL SYNTHESIS

# 4. MAGAS-SZINTŰ SZINTÉZIS

# HLS elméleti háttere

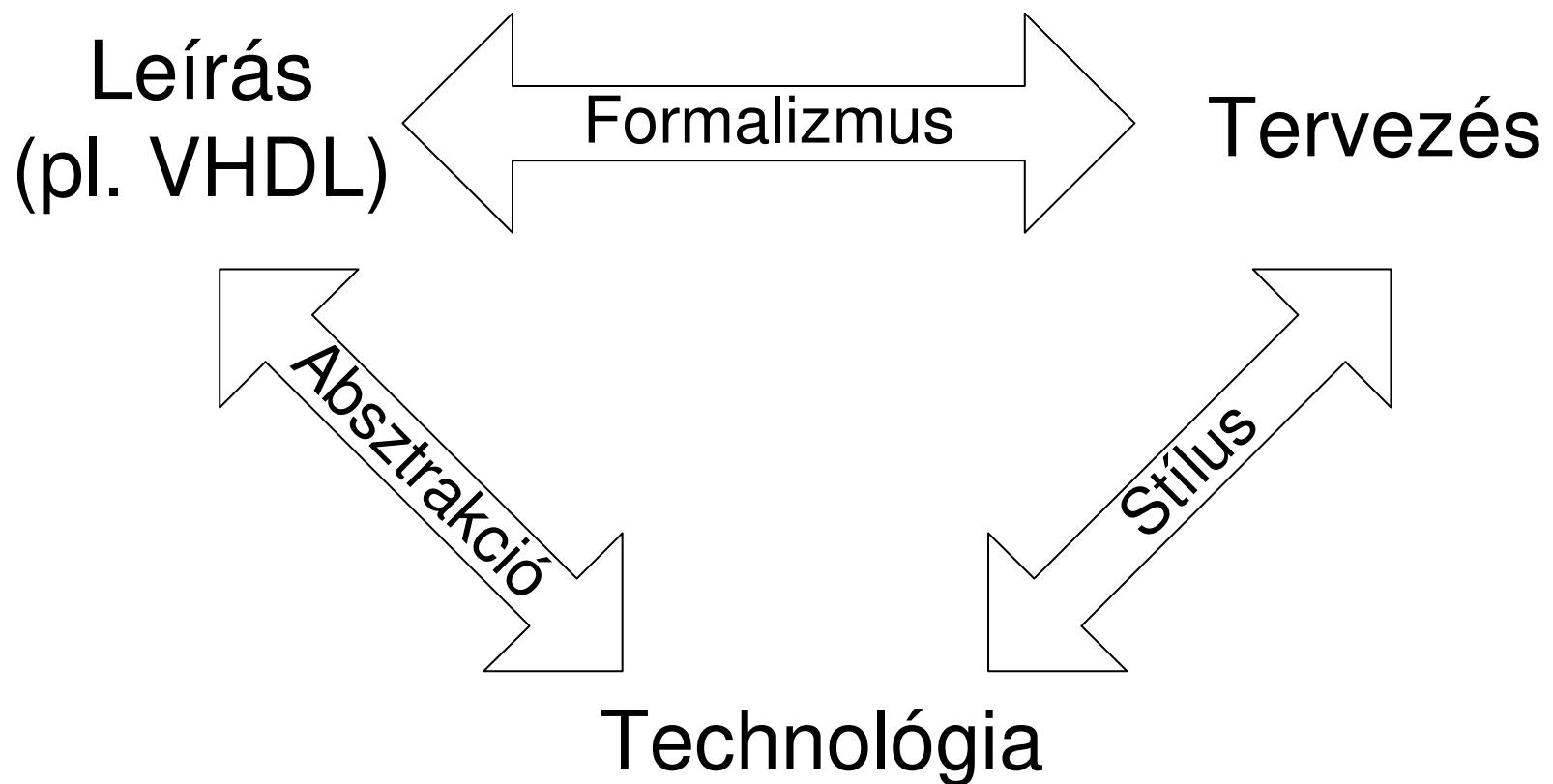
- ***Turing-Church tézis:*** HW – SW ekvivalencia.
  - a  $\mu$ -rekurzív függvények (utasítások),
  - környezet független nyelvtan (algoritmikus modell, pl. CFG), ill.
  - TM – Turing Machine (HW)  
egymásba ekvivalens módon transzformálhatóak.
- **Tervezés:** a valós rendszerek felépítéséhez különböző modellek szükségesek. Egy valós modellt implementálhatunk a számítógéppel segített (CAD tool) tervező rendszerrel: a struktúráját felépítjük, ill. viselkedését (behaviour) is tudjuk szimulálni.
- **Folyamatok:**
  - Tervezés (CAD): információ-feldolgozó folyamat:  $\Delta$
  - Gyártás (CAM):  $\pi$
  - Tesztelés v. Verifikáció (CAE) :  $\tau$

# HLS célja

- tervezés **automatizálásának** vizsgálata kombináció és, szinkron digitális hálózatok esetén, méghozzá a logikai (Boole-algebrai) szint feletti / **magasabb hierarchia szinten** definiált gyakorlati környezetben
  - Rendszer megvalósítása **gyorsan**, akár komplex rendszerek szintézisére is (elfogadható időben): „rapid prototyping, time-to-market” fogalmak
  - Bemenete magas szintű **hardver leíró nyelv** (hagyományos HDL, pl. VHDL, vagy Verilog), amelynek segítségével a rendelkezésre álló adatbázisok (DB – tervezői alap-építőelem könyvtárak) alapján digitális áramköröket építhetünk pl. újrakonfigurálható PLD, FPGA eszközökön.
- 
- „**Magas szintű szintézis**” – fő lépései:
    - Tervezés-Ütemezés (SCHEDULING)
    - Erőforrás foglalás / feladat felosztás - (LOCATION)
    - Kötött algoritmusok (BINDING) - megkötések

# Absztrakció – Formalizmus – Stílus

## ■ Erős korreláció



# 1. Stílus

- Komplex feladat ⇒ egyszerűbb, kezelhető részfeladatokra bontása
  - Szisztematikus
  - Érhető módszerek kellenek  
Pl: programozási stílusok (Top-down, Strukturált modellek, stb.)
- Jó stílus kialakításának szabályai:
  - „*Top-down*”/ „*Bottom-up*” módszerek szerinti tervezés
  - Csak kiforrott, biztos technikákat szabad alkalmazni
  - Fontos a dokumentálás (specifikáció)!

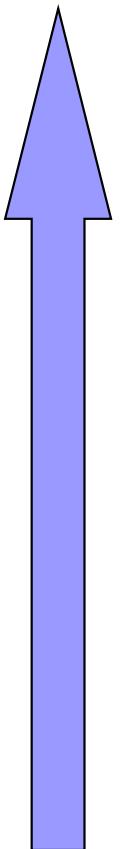
Stílushoz soroljuk a tervezés menetét (design-flow)

## 2. Absztrakció

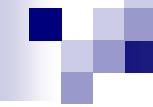
- Digitális tervezés „elvi-fogalmi” szintje
  - Kezdeti absztrakció a tervezés során meghatározó, kritikus pont!
    - i. koncepcionális modell (elvi elgondolás) vs.
    - ii. megvalósítható, realizálható modell (HW)
  - Magas-szintű **absztrakció** ⇒ elvi modell szintenkénti finomítása, és felépítése

# Absztrakciós szintek

- Rendszer szint (legfelsőbb szint)
- Algoritmikus szint
- Funkcionális szint (pl. multiplexer, dekóder, ALU stb.) – ***RTL szint*** (VHDL, Verilog stb.)
- Logikai szint (kapuk – Boole algebra)
- Fizikai áramkör: tranzisztor szint (legalsó szint)
  - erősen gyártás-technológia függő – (pl. CMOS-VLSI technology)



## 3. Formalizmus



Nyelvi eszközök



# **5. HARDVER LEÍRÓ NYELVEK ÉS SW FEJLESZTŐ ESZKÖZÖK CFG-DFG gráf reprezentációk**

# FPGA-k lehetséges „programozási nyelvei”:

## ■ I.) Hagyományos HDL nyelvek:

- VHDL,
- Verilog/SystemVerilog

## ■ II.) C-alapú nyelvek ( $C \rightarrow$ FPGA szintézis):

- a.) Xilinx Vivado HLS ( $C/C++$ , System C, OpenCV)
- b.) Intel SDK (OpenCL, HLS)
- c.) Impulse-C, Catapult-C, Handel-C, System-C, Mitrion-C, stb. (még ~10 további)

## ■ III) Modell alapú nyelvek:

- a.) MATLAB Simulink,
- b.) NI LabVIEW (FPGA Module)

# Hagyományos HDL nyelvek célja I.

- Hardver **modellezés**
- A nyelv elemkészlet jelentős része
  - csak a hardver funkciók modellezésére ill. PC-n történő **szimulációra** használható, ill.
- A nyelv elemkészlet bővebb tartománya
  - **Szintézisre** és **implementációra** is használható (FPGA-n futtatható), azonban
  - Szintetizálható részhalmaz szintézis eszköz (synthesizer, pl XST) függő
- Kapuszintű modulokból építkező, kapcsolási rajzon alapuló tervezési módszerek leváltása
- **RTL** (register transfer level) szintű leírás: VHDL, Verilog
- **Automatikus** hardver szintézis a leírásból
- Tervezői **hatékonyság** növelése

# HDL nyelvek II.

- Alapvetően **moduláris** és egyben **hierarchikus** felépítésű tervezést tesz lehetővé
- HDL modul = entitás
  - Be-, kimenetek, illetve kétirányú port-ok definiálása
  - Be-, kimenetek közötti logikai kapcsolatok és időzítések definiálása
- **NEM szoftver!!**
- Alapvetően **időben párhuzamos, konkurens** működést ír le (gondoljunk az FPGA nagyszámú, párhuzamosan elérhető blokkjainak felépítésére)
  - Ha külön definiáljuk (process, always), akkor fogja csak szekvenciálisan végrehajtani az utasításokat!

# FPGA-k, HDL nyelvek gyengeségei

- HDL nyelveken történő fejlesztés a hagyományos szoftver fejlesztéshez viszonyítva továbbra is időigényes
- Sok SW fejlesztő rendelkezik C/C++ ismerettel, azonban kevesen HDL tapasztalattal, amely a terjedésének egyik fő korlátja.



Éppen ezért születtek meg napjainkra a *magas-szintű C szintaktikát követő (HLS) nyelvek, valamint a Modell-alapú nyelvek*

- + Gyorsabb prototípus fejlesztés, szimuláció/verifikáció
- + HW(FW) / SW együttes tervezés, szimuláció és verifikáció

# Tradicionális HDL nyelvek

Szabványos HDL (Hardware Description Language)

## VHDL

- 1983-85: IBM, Texas Instruments
- 1987: IEEE szabvány (IEEE 1076-1987)
- 1994: VHDL-1993
- ...

## Verilog

- 1984: Gateway Design Automation Inc.
- 1995: IEEE szabvány (IEEE 1364)
- 2001: Verilog-2001
- ...

# I. a.) VHDL hardver leíró nyelv

- **VHDL = VHSIC HDL** (Very High Speed Integrated Circuit Hardware Description Language) nevű nyelv a DARPA program keretében a 80-as évek közepén alakult ki, melyet azóta nemzetközi szabványnak is elfogadtak (IEEE-1076 1987ben).
  - Nagyon-nagy sebességű Integrált Áramkörök - Hardver Leíró Nyelve,
  - A VHDL szintaxisa az ADA programnyelven alapul: erősen típusos, jól strukturált, objektum-orientált magas-szintű „programnyelv”,
  - elsődleges célja volt: a magasabb absztrakciós szinteken az ember számára is áttekinthetővé tenni egy hardver dokumentációját,
  - biztosítja az egyes rétegnek megfelelően egy szabványos „nyelven” a hardver leírását,
  - az alsó szinteken lehetővé teszi a tervezek átadását a gyártás CAD/CAM rendszereinek,
  - biztosítja a hardver szimulációs ellenőrzését (verifikációt).

# VHDL

- A VHDL nyelvű hardver leírás leginkább egy emberi nyelvhez közel álló, magas szintű programnyelvhez hasonlítható. A VHDL hardver leíró nyelv három alapegysége
  - *architektúra* (architecture): hw egység funkciója (viselkedése) és szerkezete
  - *egyed* (entity): hw egységek közötti kommunikációs interfész (protokoll)
  - *konfiguráció* (configuration): architektúrák és interfések egymáshoz rendelése

```
ENTITY or_gate IS
    PORT (a, b: IN bit; y: OUT bit)
END or_gate
ARCHITECTURE viselkedes OF or_gate IS
BEGIN
    y <= a OR b
END viselkedes
CONFIGURATION or2_konfig OF or_gate IS
    FOR viselkedes
        END FOR
    END or2_konfig
```

# CFG: Control Flow Graph

- Irányított gráf,  $\text{CFG}=(\mathbf{N}, \mathbf{P})$  Vezérlés-folyam gráf
- Csúcsok halmaza (**N**):
  - hozzárendelés,
  - összeadás,
  - logikai műveletek, stb.
- Élek halmaza (**P**):
  - Precedencia relációk,
  - egymást követő utasítások
- Összefüggő gráf (nem aciklikus!)
- Sorozat, szekvencia: egy él  $(n_1, n_2) \in P$  azt jelenti, hogy  $n_2$  következik  $n_1$  végrehajtása után
- Feltételes végrehajtás (branch=if): egy művelet után egy másik művelet, ha a feltétel teljesül.
  - A feltétel Boolean kifejezés, melynek értéke '1', akkor végrehajtódik, különben '0'.
- Iteráció (loop): Ciklusok, amelyek a folyamat iteratív viselkedését jelzik

# DFG: Data Flow Graph

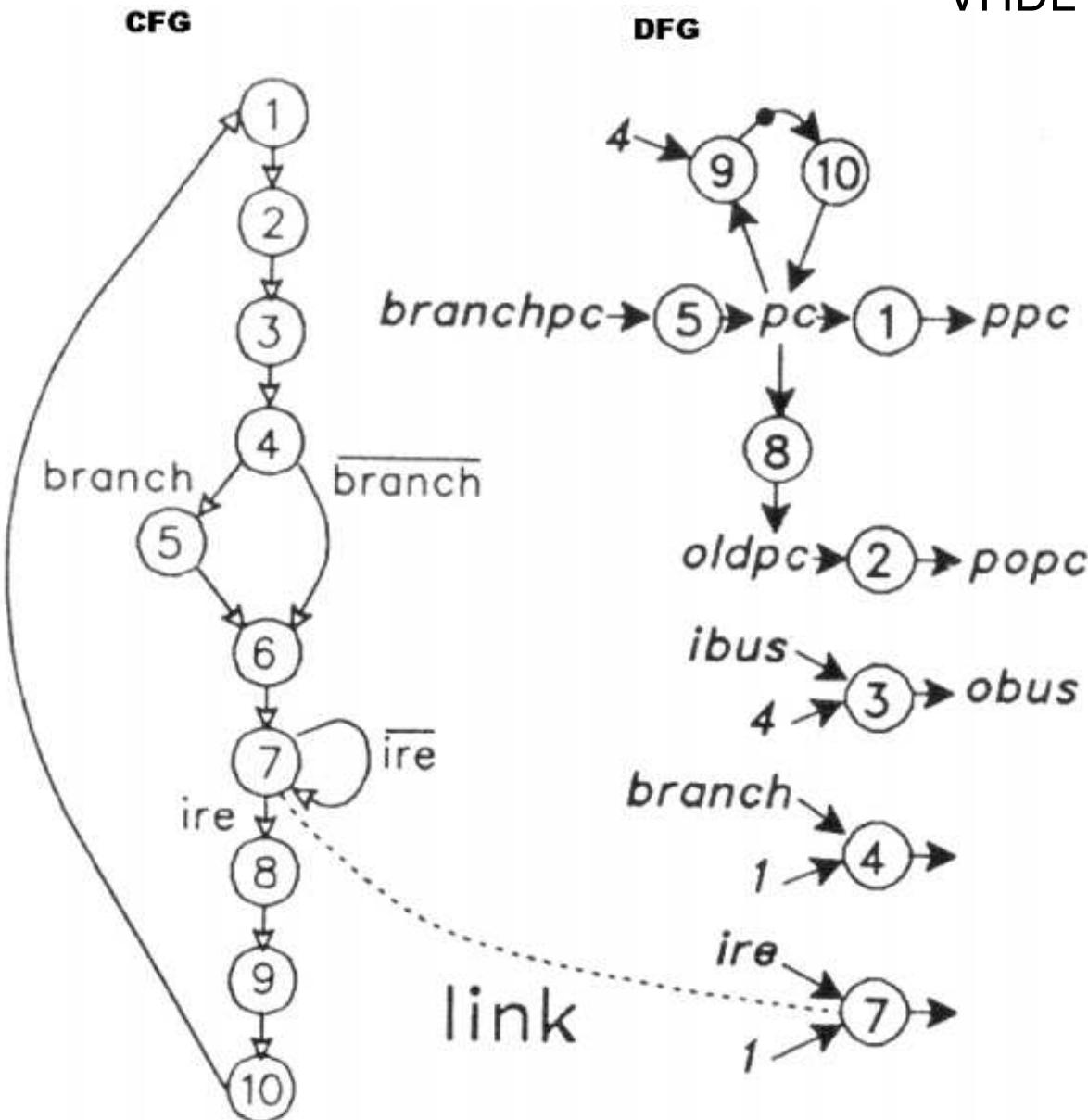
- Irányított gráf,  $\text{DFG}=(\mathbf{N} \cup \mathbf{V}, \mathbf{D})$  Adat-folyam gráf
- Csúcsok halmaza:
  - **N: utasítások**
  - **V: változók**
- Élek halmaza (D):
  - Adatkapcsolatok
- Fontos: nem feltétlenül összefüggő gráf, azaz a DFG-nél az egyes műveletek lehetnek ábrázolva úgy is, hogy nincs közöttük kapcsolódási pont (nem húzódik él), tehát részekre lehet bontani

# Példa 1.) VHDL → CFG, DFG

Csak az  
hozzárendelési utasítások  
megszámozása (begin-end részen)

Process-en belüli részek sorrendben  
(egymás után hajtódnak végre)

VHDL leírás:



Entity prefetch is  
Port ( branchpc, ibus in bit32;  
branch, ire: in bit;  
ppc, popc, obus: out bit32);  
end prefetch;

architecture *behavior* of prefetch is  
begin

```

process
variable pc, oldpc: bit32:=0;
begin
    ppc <= pc; --1
    popc <= oldpc; --2
    obus <= ibus + 4; --3
    if (branch = '1') then --4
        pc:= branchpc; --5
    end if; --6
    wait until (ire='1'); --7
    oldpc:=pc; --8
    pc:=pc+4; --9, 10
end process;
end behavior;
  
```

# Példa 2): VHDL → CFG, DFG

architecture behavioral of MODULE is

begin

```
p1: process(a, b, cin)
variable vsum : std_logic_vector(3 downto 0);
variable carry : std_logic;
begin
    carry := cin; --1
    for i in 0 to 3 loop --2
        vsum(i) := (a(i) xor b(i)) xor carry; --3
        carry := (a(i) and b(i)) or (carry and (a(i) or b(i))); --4
    end loop; --5
    sum <= vsum; --6
    cout <= carry; --7
end process p1;
end behavioral;
```

Csak az  
Utasítások  
megszámozása!



--1  
--2  
--3  
--4  
--5  
--6  
--7

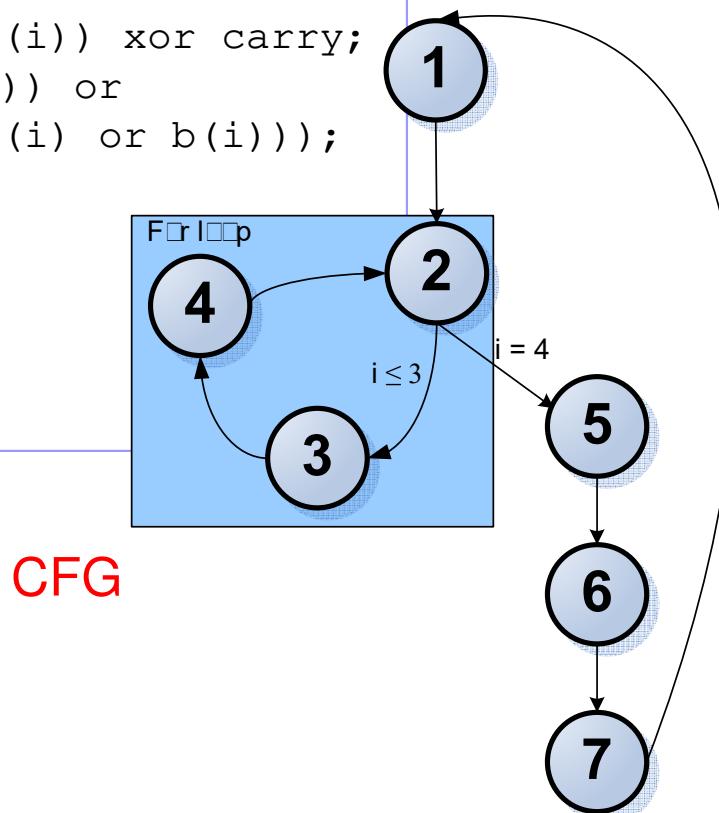
Megj. Ez egy 4-bites RCA (4 Full Adder sorba kapcsolását 0..3-ig) valósítja meg!

# VHDL → CFG, DFG

```

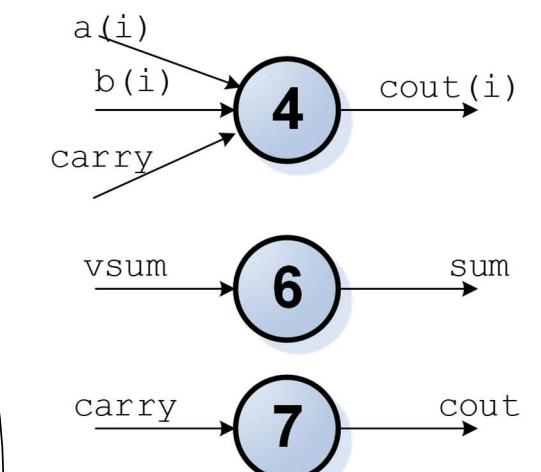
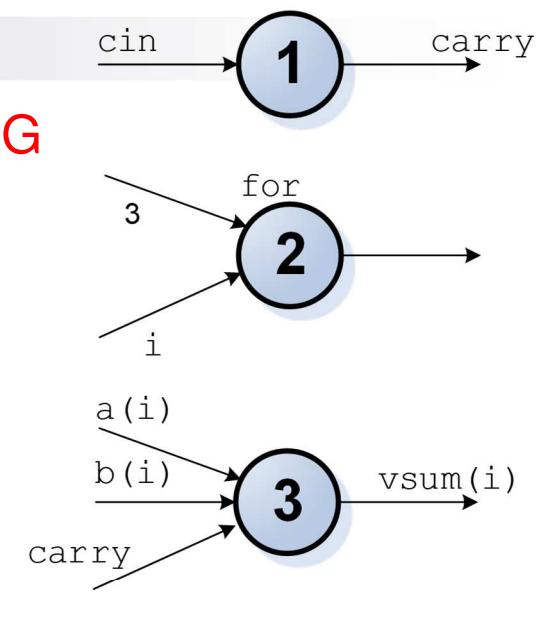
p1: process(a, b, cin)
    variable vsum : std_logic_vector(3 downto 0);
    variable carry : std_logic;
begin
    --1    carry := cin;
    --2    for i in 0 to 3 loop
    --3        vsum(i) := (a(i) xor b(i)) xor carry;
    --4        carry := (a(i) and b(i)) or
    --           (carry and (a(i) or b(i)));
    --5    end loop;
    --6    sum <= vsum;
    --7    cout <= carry;
end process p1;

```



CFG

DFG



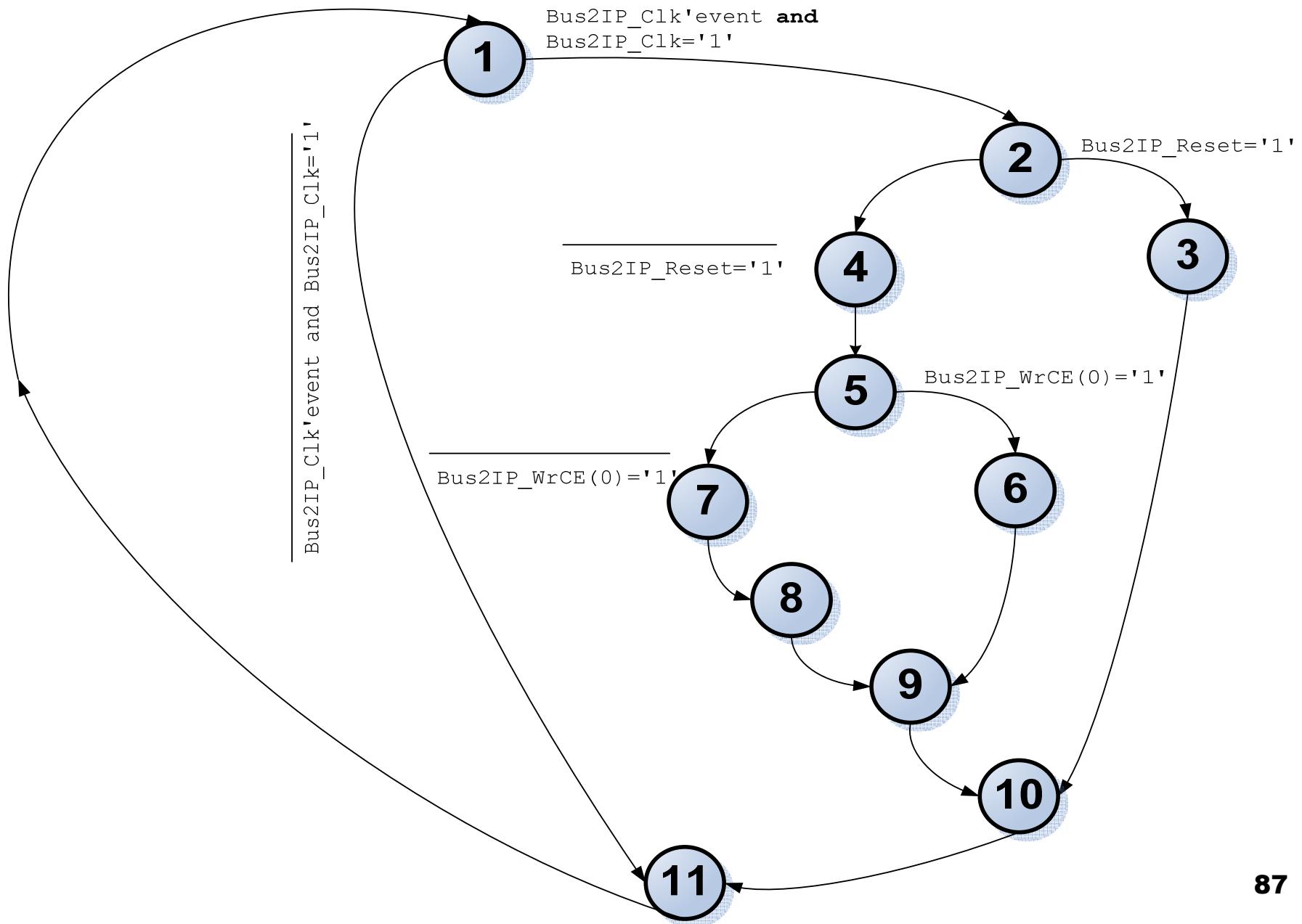
# Példa 3): VHDL → CFG

```
-- behavioral implementation of the LED_proc
architecture behavioral of LED is
begin
    LED_PROC: process (Bus2IP_Clk) is
begin
    if Bus2IP_Clk'event and Bus2IP_Clk='1' then
        if Bus2IP_Reset='1' then
            LED_i<="0000";
        else
            if Bus2IP_WrCE(0)='1' then
                LED_i<=Bus2IP_Data(0 to 3);
            else
                LED_i<=LED_i;
            end if;
        end if;
    end if;
end process LED_PROC;
end behavioral;
```

Csak az  
Utasítások  
megszámozása!

--1  
--2  
--3  
--4  
--5  
--6  
--7  
--8  
--9  
--10  
--11

# VHDL CFG



# Intel vs. AMD-Xilinx fejlesztő környezetek

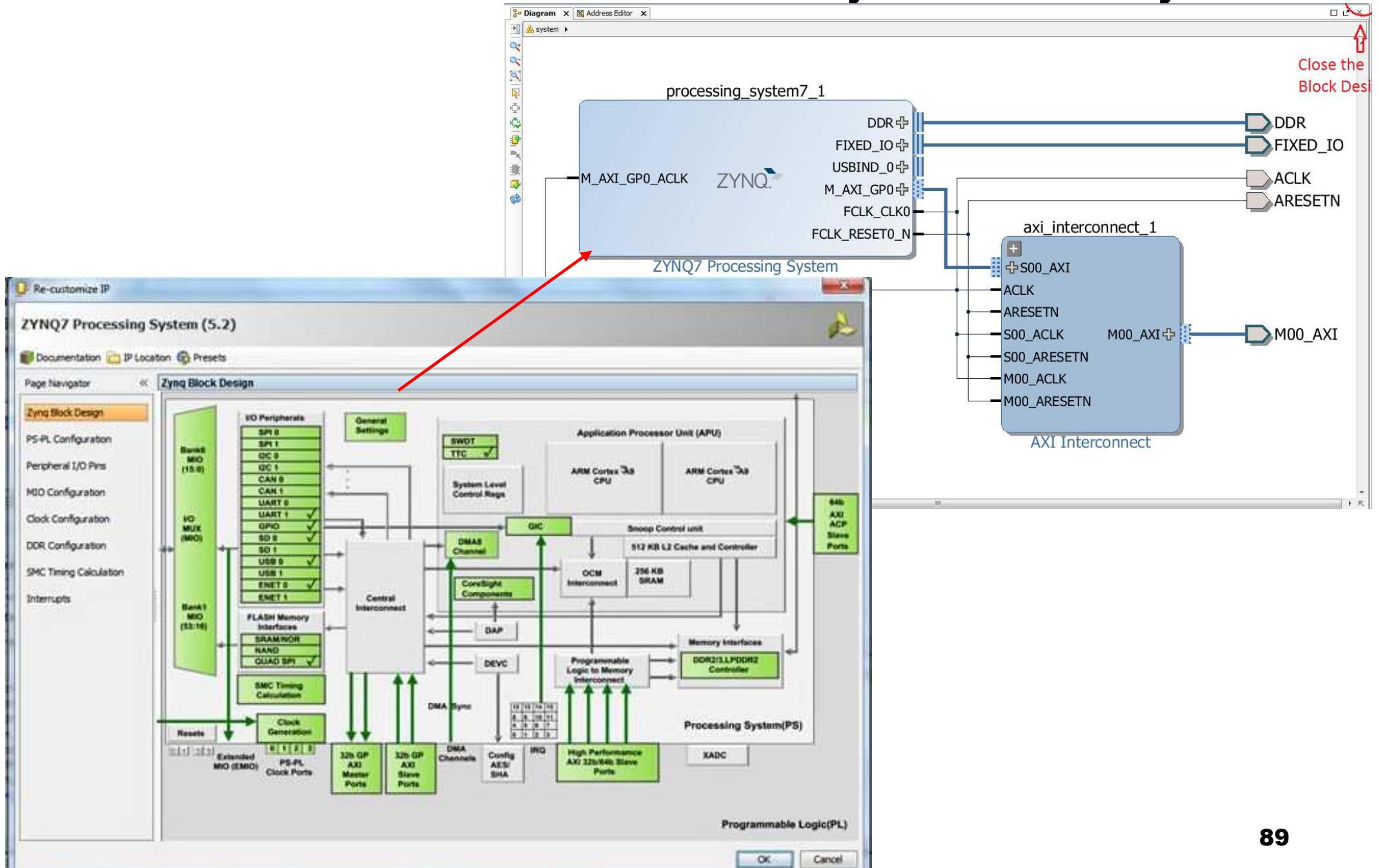
## ■ Intel-Altera

- Quartus Prime: integrált fejlesztő környezet (HW/FW fejlesztés) *HDL*-ből
  - Qsys: modularizált, hierarchikus beágyazott rendszer fejlesztés
  - EDS: beágyazott SW fejlesztés (Eclipse)
  - NiosII EDS: soft core, EDS-SOC: ARM hard-core
- Intel SDK OpenCL (2013)
- Modelsim: professzionális HDL szimulációs környezet

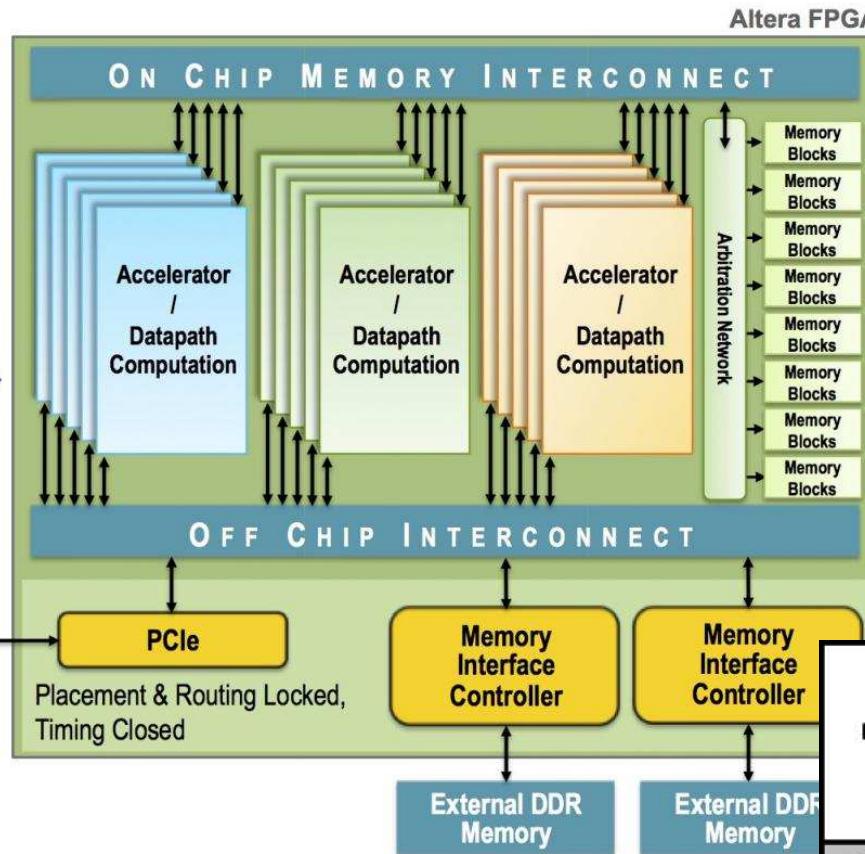
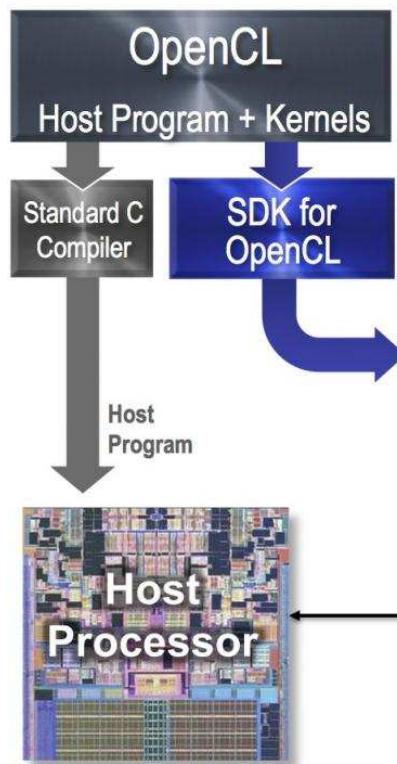
## ■ AMD-Xilinx

- VITIS: egységes SW platform (2020)
- Vivado: HDL alapú környezet, Vivado HLS: C/C++
- ISE Design Suite: integrált fejlesztő környezet (HW/FW) *HDL*-ből (C/C++ Vivado HLS (2013))
  - EDK: beágyazott rendszer fejlesztő környezet
  - SDK: beágyazott SW fejlesztő körny. (Eclipse)
- ModelSim, vagy ISim: szimulációs környezetek

# I. a. Xilinx Vivado / HLS fejlesztő környezet



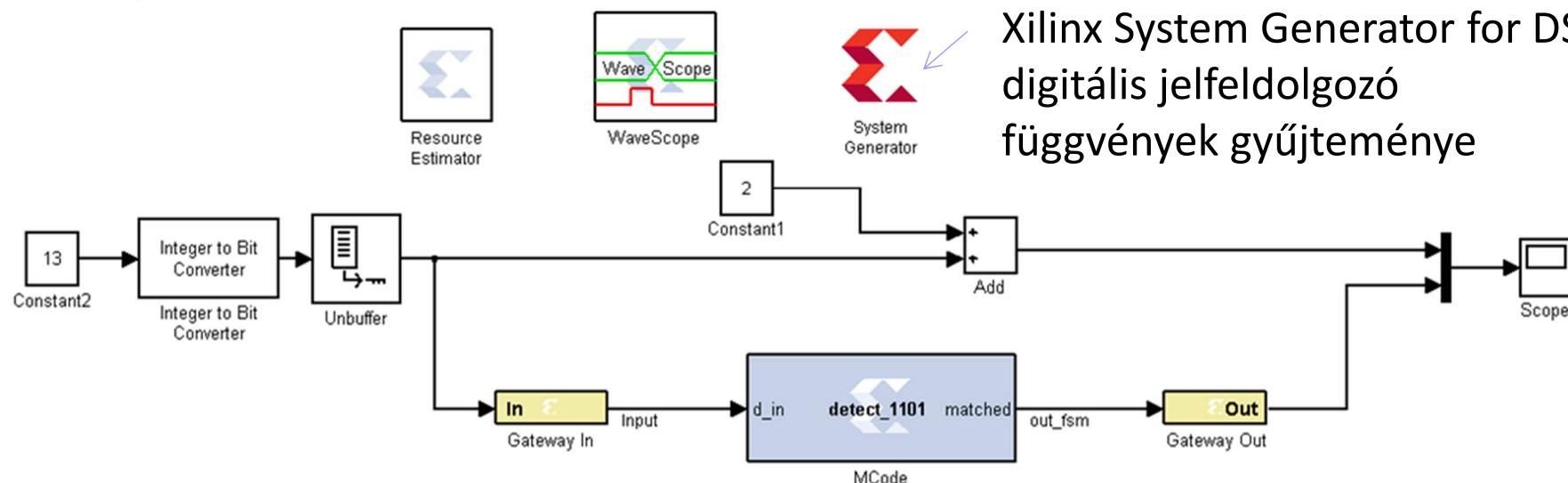
# I. b. Altera/Intel SDK for OpenCL



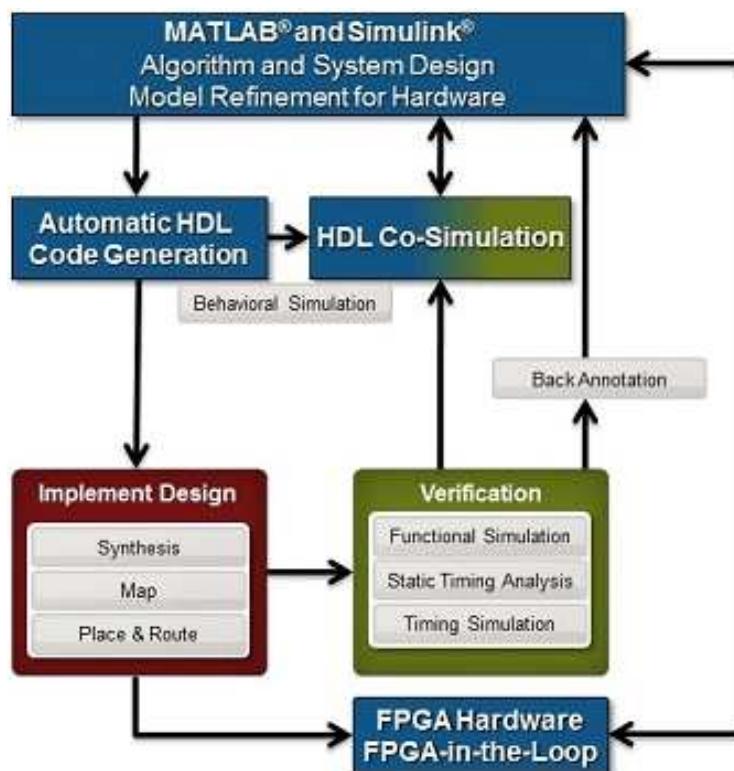
1,2,3 .... utasítások  
A, B,C ... adatok

SIMD Parallelism (GPU)	1 A	1 B	1 C	1 D	1 E	4 A	4 B	4 C	4 D	4 E
Clock Cycle	1	2	3	4	5	6	7	8	9	10
Pipeline Parallelism (FPGA)	1 A	2 A	3 A	4 A	5 A	6 A				
	1 B	2 B	3 B	4 B	5 B	6 B				
	1 C	2 C	3 C	4 C	5 C	6 C				
	1 D	2 D	3 D	4 D	5 D	6 D				
	1 E	2 E	3 E	4 E	5 E	6 E				

# III. a.) MATLAB - Simulink



Xilinx System Generator for DSP:  
digitális jelfeldolgozó  
függvények gyűjteménye



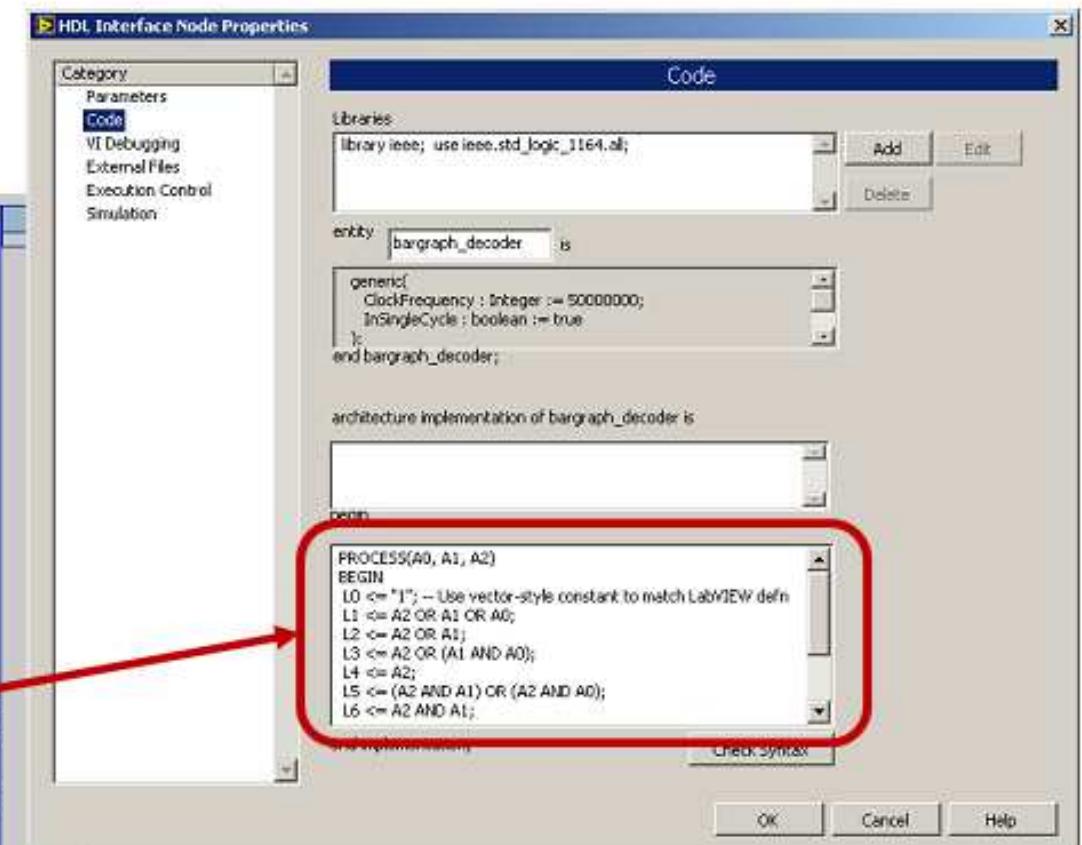
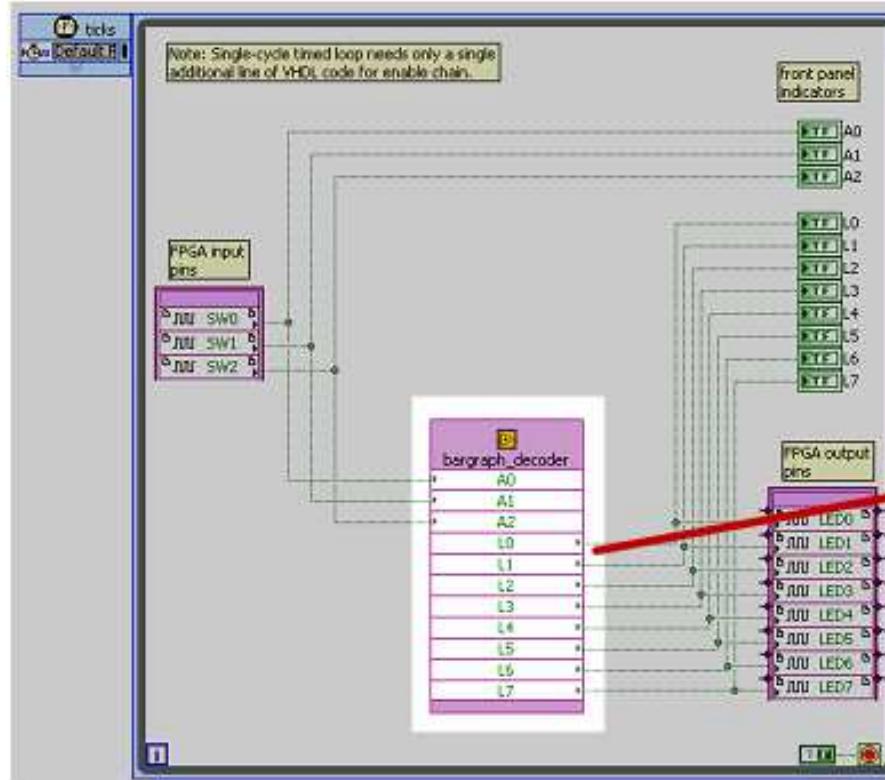
Mcode: Matlab-Code (.m nyelven  
írható egyedi modul, amely beépíthető  
akár a Simulink blokkok közé

# III. b.) LabVIEW – FPGA modul



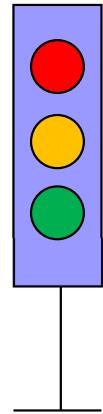
VHDL node

3->8 dekóder



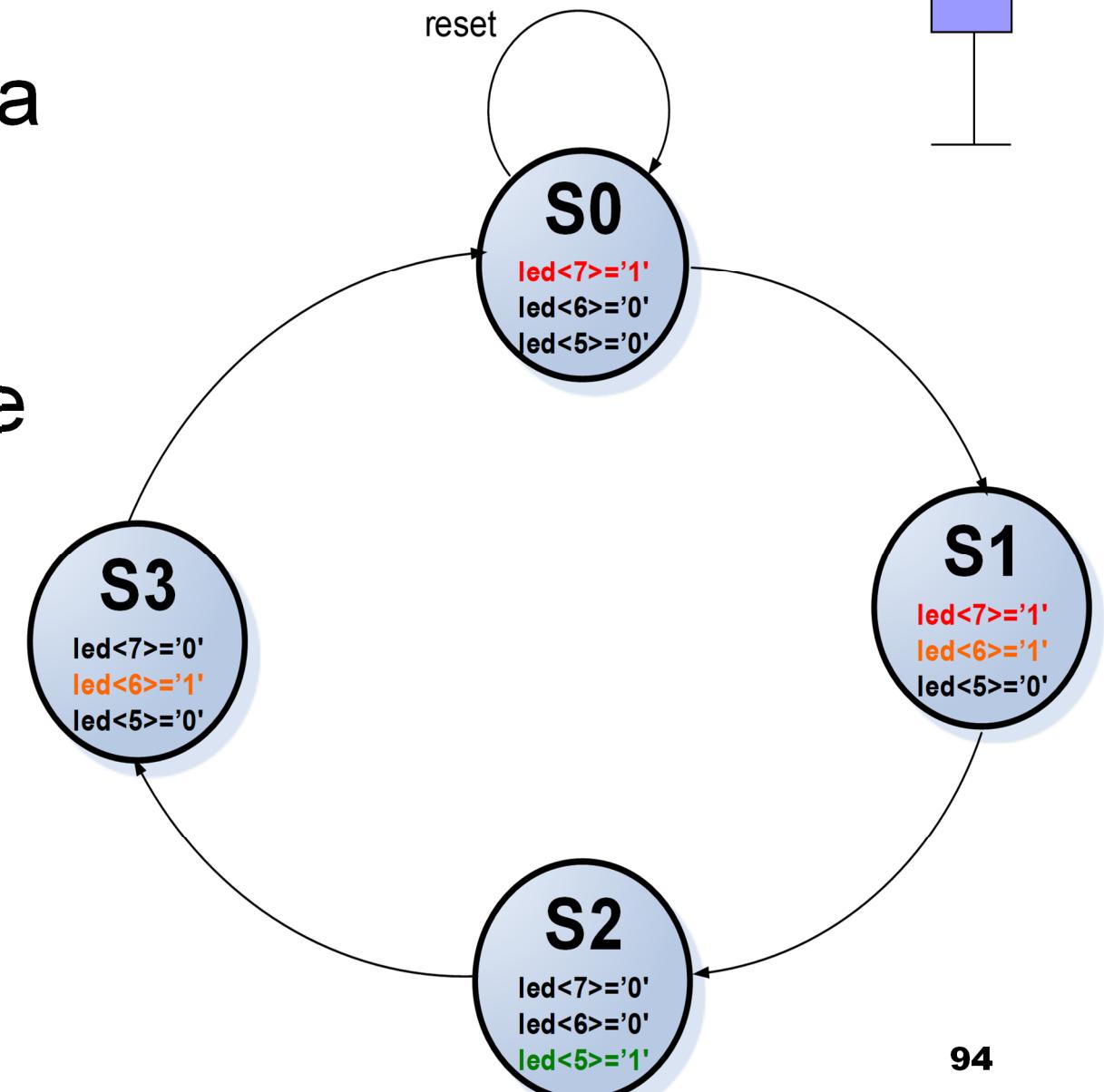
# További ajánlott irodalom

- Fontosabb FPGA gyártók oldalai:
  - <https://www.amd.com>
  - <https://www.intel.com/content/www/us/en/products/programmable.html>
  - <http://www.microsemi.com>
- FPGA és Programmable Logic Journal:
  - <http://www.fpgajournal.com>
- Xilinx FPGA Silicon Devices:
  - <http://www.xilinx.com/products/devices.htm>
- Fodor Attila, Dr. Vörösházi Zsolt: Beágyazott rendszerek és programozható logikai alkatrészek (TÁMOP 4.1.2) Egyetemi jegyzet (2011)
  - [http://www.tankonyvtar.hu/hu/tartalom/tamop425/0008\\_fodorvoroshazi/Fodor\\_Voroshazi\\_Beagy\\_0903.pdf](http://www.tankonyvtar.hu/hu/tartalom/tamop425/0008_fodorvoroshazi/Fodor_Voroshazi_Beagy_0903.pdf)
- Témához kapcsolódó téma a Pannon Egyetemen:
  - **(VEMIVIB334TM) Tervezési módszerek programozható logikai alkatrészekkel (VHDL) – őszi félév**
  - **(VEMIVIB334BR) FPGA-alapú beágyazott rendszerek – tavaszi félév**



# Példa VHDL – Jelzőlámpa

- Közlekedési lámpa vezérlő = állapotdiagram (DFA/FSM – Finite State Machine).



- Megvalósítás:  
Melyik tanult klasszikus S.H. modell ez ?

# Feladat 1/a.) Megoldás VHDL: ([traffic\\_Moore.vhd](#))

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity traffic_control is
generic (N: natural := 3);
port ( clk : in STD_LOGIC;
       reset : in STD_LOGIC;
       led : out STD_LOGIC_VECTOR (N-1
      downto 0));
end traffic_control;

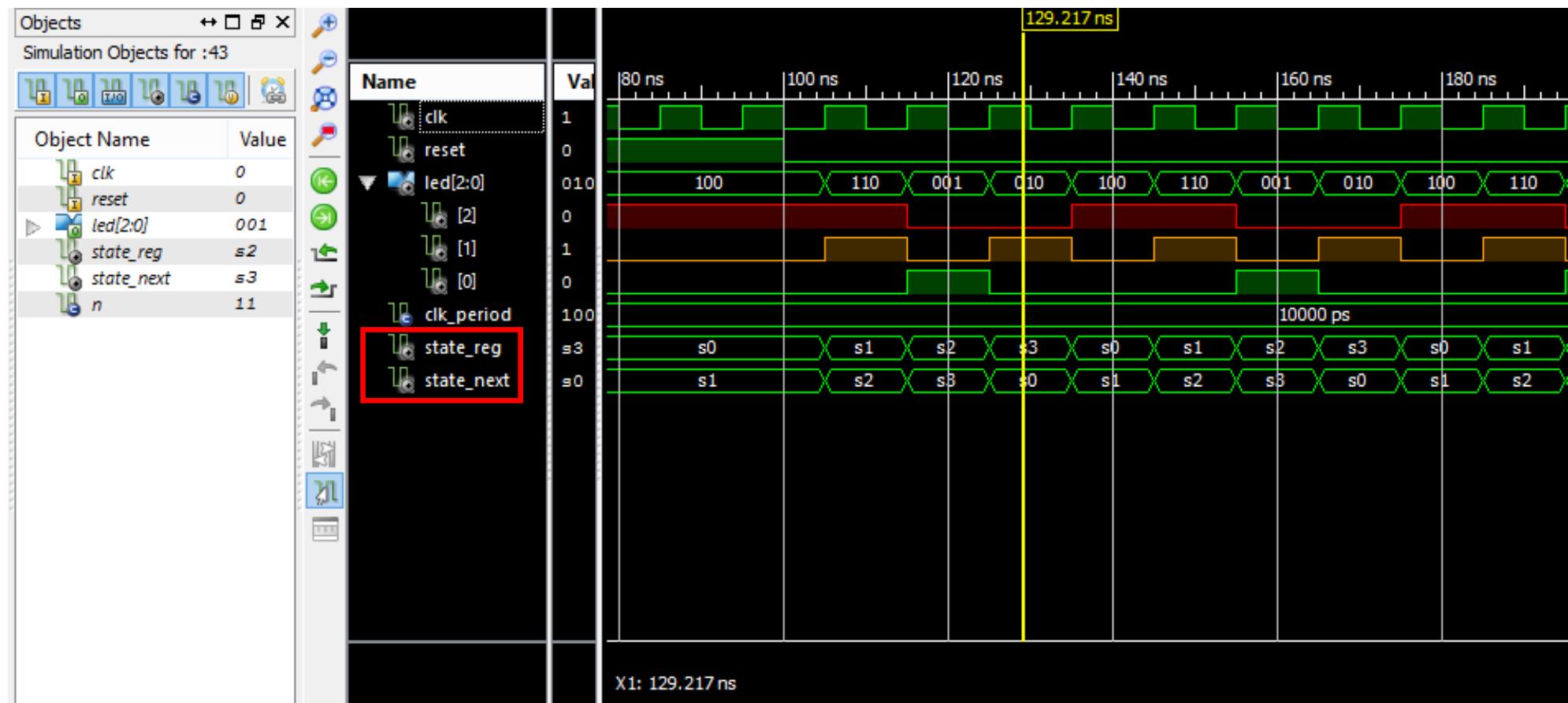
architecture Behavioral of traffic_control is
type traff_state_type is (s0, s1, s2, s3);
-- P, PS, Z ,S -> P
signal state_reg, state_next:
traff_state_type;

begin
  -- state register
  process(clk, reset)
begin
  if (reset='1') then
    state_reg <= s0;
  elsif rising_edge(clk) then
    state_reg <= state_next;
  end if;
end process;
```

```
-- next-state logic
process(state_reg)
begin
  case state_reg is
    when s0 =>
      state_next <= s1;
    when s1 =>
      state_next <= s2;
    when s2 =>
      state_next <= s3;
    when s3 =>
      state_next <= s0;
  end case;
end process;

-- Output logic (led2, led1, led0) !
process(state_reg)
begin
  case state_reg is
    when s0 =>
      led(N-1 downto 0) <= "100"; --P
    when s1 =>
      led(N-1 downto 0) <= "110"; --PS
    when s2 =>
      led(N-1 downto 0) <= "001"; --Z
    when s3 =>
      led(N-1 downto 0) <= "010"; --S
  end case;
end process;
end Behavioral;
```

# Szimulációs eredmény





# Számítógép Architektúrák II.

(MIVIB344ZV)

9. előadás: Beágyazott rendszerek alapjai.  
Mikrovezárlók (MCU-k).

Előadó: Dr. Vörösházi Zsolt  
[voroshazi.zsolt@mik.uni-pannon.hu](mailto:voroshazi.zsolt@mik.uni-pannon.hu)

# Jegyzetek, segédanyagok:

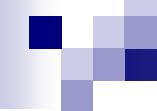
- Könyvfejezetek:
  - <http://www.virt.uni-pannon.hu> → Oktatás → Tantárgyak → Számítógép Architektúrák II.
- Fóliák, óravázlatok .ppt (.pdf)
- Feltöltésük folyamatosan

# Ajánlott és felhasznált irodalom

- Fodor Attila, Dr. Vörösházi Zsolt: Beágyazott rendszerek és programozható logikai alkatrészek (TÁMOP 4.1.2) Egyetemi jegyzet (2011)  
 [http://www.tankonyvtar.hu/hu/tartalom/tamop425/0008\\_fodorvoroshazi/Fodor Voroshazi Beagy 0903.pdf](http://www.tankonyvtar.hu/hu/tartalom/tamop425/0008_fodorvoroshazi/Fodor_Voroshazi_Beagy_0903.pdf)  
(1. Beágyazott rendszerek fejezetrész)

- MCU = Micro Controller Unit:

-  <https://en.wikipedia.org/wiki/Microcontroller>
-  <https://www.elprocus.com/microcontrollers-types-and-applications/>
-  <https://www.elprocus.com/difference-between-avr-arm-8051-and-pic-microcontroller/>
-  <https://www.elprocus.com/difference-between-arduino-and-raspberry-pi/>



# **Beágyazott rendszerek**

**Bevezetés**

# Beágyazott Rendszerek

- A beágyazott rendszer (**Embedded System**) a (számítógépes) **hardver-** és **szoftverelemeknek** kombinációja, amely kifejezetten egy adott funkciót, *specifikus* (vezérlési) feladatot képes ellátni, szemben az általános célú számítógép rendszerekkel.  
**HW + (FW) + SW + (OS) = Beágyazott rendszer**
- A beágyazott rendszerek olyan számítógépes eszközöket tartalmazhatnak, amelyek alkalmazás-orientált célberendezésekkel (**ASIC/ASSP**, **GPU**, **FPGA**, **MCU**, **CPU/MPU**, **DSP**, stb.), vagy komplex alkalmazói rendszerekkel (akár OS) szervesen egybeépülve akár azok **autonóm** működését is képesek biztosítani.
- A **programozható** beágyazott rendszerek olyan programozói interfésszel vannak ellátva, amelyek általában sajátos szoftver (firmware) fejlesztési stratégiákat és technikákat követelnek meg.

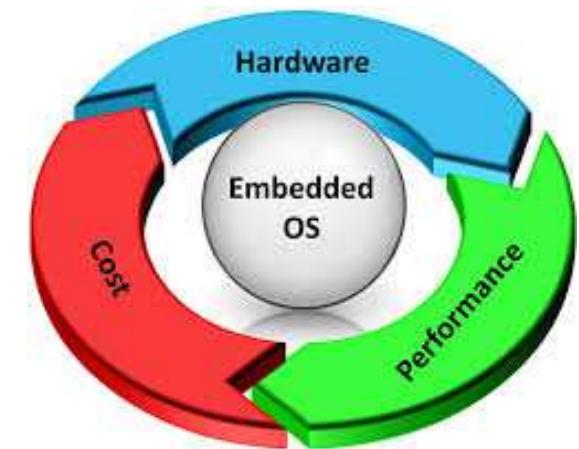
# Néhány fontos alkalmazási terület

- Autóipari alkalmazások: beágyazott elektronikus vezérlők
  - Biztonságkritikus: központi elektronikai vezérlő (ECU), motorvezérlés, fékrásegítő, sebességváltó, blokkolásgátló vezérlés (ABS), kipörgégátló (ESP) légzsák
  - Utas központú (komfort) rendszerek: szórakoztatás, ülés/tükör ellenőrzés stb.
- Repülőgép-ipari és védelmi alkalmazások
  - Repülésirányító rendszerek (fedélzeti navigáció, GPS vevő), hajtómű vezérlés, robotpilóta
  - Védelmi rendszerek, radar rendszerek, rádió rendszerek, rakétavezérlő rendszerek
- Gyógyászati berendezések:
  - Orvosi képfeldolgozás
  - Jelmonitorozás (PET, MRI, CT)
- Hálózati/ telekommunikációs rendszerek (modem, router stb.)
- WSN: Vezeték nélküli szenzorhálózatok (motes)
- IoT: Intelligens, vagy smart rendszerek
- Háztartási gépek, ill. fogyasztói elektronika
  - mobiltelefon, PDA, PNA, digitális kamera, nyomtató stb.



# Általános követelmények

- „Dedikált” funkció
  - Jól körülhatárolt (alkalmazás specifikus) funkció(k) támogatása
- Szigorú követelmények
  - Alacsony költség (**Cost**)
  - Gazdaságosság (**Economy**) - lehetőleg minimális alkatrészből épüljön fel
  - Gyors működés (**Speed**)
  - Alacsony disszipáció (**Power**)
- Valós idejű (real-time) működés és válasz
  - a környezetet folyamatos monitorozása, és beavatkozás
- Hardver-, és szoftver részek elkülönült, de együttes tervezése (co-design), tesztelése (co-simulation), ellenőrzése (co-verification)



# Alapkövetelmények:

- **Idő:** Egy bekövetkező esemény kezelését a beágyazott rendszer egy *meghatározott* időn belül kezdje el.
- **Biztonság:** olyan rendszer vezérlése, amely hibás működés esetén egészségkárosodás, és komoly anyagi kár nélkül kezeli a bekövetkező eseményt.

E filozófia mentén a beágyazott rendszerek két *alcsoporthatáját* lehet definiálni:

- **Valós idejű rendszer** (v. **idő kritikus**): melynél az időkövetelmények betartása a legfontosabb szempont,
- **Biztonságkritikus rendszer:** melynél a biztonsági funkciók sokkal fontosabbak, mint az időkövetelmények betartása.

Megjegyzés: A valóságban nem lehet ilyen könnyedén a beágyazott rendszereket csoportosítani, mert lehetnek olyan valós idejű rendszerek is, melyek rendelkeznek a biztonságkritikus rendszerek bizonyos tulajdonságaival. Szabványok és a törvények szabályozzák azt, hogy milyen alkalmazásoknál kell kötelezően biztonságkritikus rendszert alkalmazni (pl. ADAS ISO 26262).

# Valós-idejű rendszerek

A követelmények szigorúsága alapján kétféle valós-idejű (real-time) rendszert különböztethetünk meg:

- **hard real-time rendszer:** szigorú követelmények vannak előírva, és a *kritikus* folyamatok meghatározott időn belül kell, hogy feldolgozásra kerüljenek,
- **soft real-time rendszer:** a követelmények kevésbé szigorúak, és a *kritikus* folyamatokat a rendszer mindenkor nagyobb prioritással dolgozza fel.

# Ütemezés (scheduling)

A (valós-idejű) operációs rendszerek (**OS/RTOS**) számára is kritikus feladat az ütemezés és az erőforrásokkal való optimális gazdálkodás. Mivel minden rendszer, valamilyen periféria segítségével kommunikál a környezetével, ezért fontos a perifériák valós-idejű rendszer követelményeinek megfelelő módon történő kezelése: a válaszidő betartásához az eseményt lekezelő *utasítás sorozatot* végre kell hajtani. Az utasítássorozat lefutása erőforrásokat igényel, melyeket az operációs rendszernek kell biztosítani, hogy hozzá tudja rendelni az időkritikus *folyamatokhoz*.

A processzorok **ütemezésének** következő szintjeit lehet megkülönböztetni:

- Hosszú-távú (long term) ütemezés vagy munka ütemezés,
- Közép-távú (medium term) ütemezés,
- Rövid-távú (short term) ütemezés.

# Ütemezés szintjei

Az operációs rendszerek magja (kernel) tartalmazza az ütemezőt.

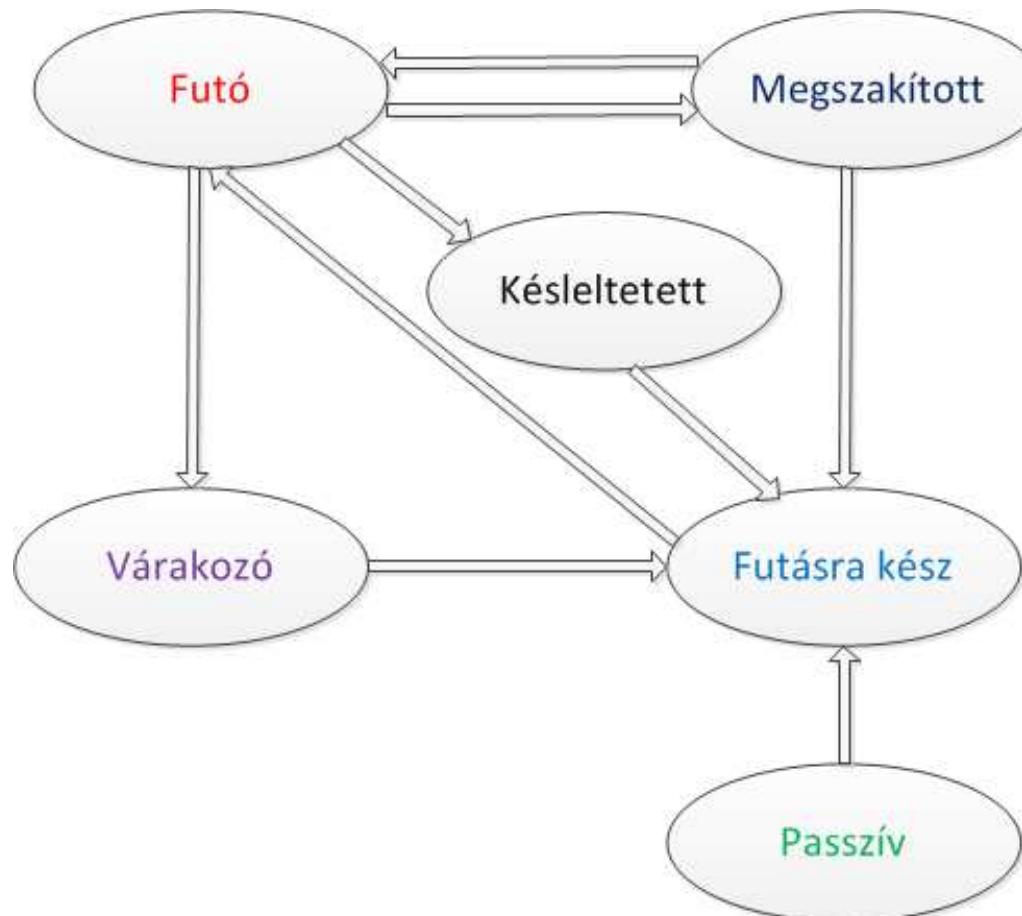
- **A hosszú-távú ütemezés** feladata, hogy a *háttértáron* várakozó, még el nem kezdett munkák közül meghatározza, melyek kezdjenek el futni, a munka befejeződésekor ki kell választania egy új elindítandó munkát. A hosszú-távú ütemezést végző algoritmusnak ezért *ritkán* kell futnia.
- **A közép-távú ütemezés** az időszakos *terhelésingadozásokat* hívhatott megszüntetni, hogy a nagyobb terhelések esetében ne legyenek időtúllépések. A középtávú ütemező algoritmus ezt úgy oldja meg, hogy bizonyos (nem időkritikus) folyamatokat *felfüggeszt*, majd *újraaktivál* a rendszer terhelésének a függvényében. Folyamat felfüggesztése esetén a folyamat a *háttértáron* tárolódik, az operációs rendszer elveszi a folyamattól az erőforrásokat, melyeket csak a folyamat újraaktiválásakor ad vissza a felfüggesztett folyamatnak.
- **A rövid-távú ütemezés** feladata, hogy kiválassza, hogy melyik futásra kész folyamat *kapja meg a processzort*. A rövidtávú ütemezést végző algoritmus *gyakran* és *gyorsan* fut le, ezért az operációs rendszer mindenkorban a *memoriában* tartja az ütemező kódját.

# Ütemezés – további fogalmak

Az ütemezéssel és a programokkal kapcsolatban a következő alapfogalmak értelmezhetők:

- **Task:** Önálló részfeladat.
- **Job:** A task-ok kisebb, rendszeresen végzett részfeladatai.
- **Process:** A legkisebb futtatható programegység, egy önálló ütemezési entitás, amelyet az OS önálló programként kezel. Van saját (védett) memória területe, mely más folyamatok számára elérhetetlen. A task-okat folyamatokkal implementálhatjuk.
- **Thread:** Saját memóriaterület nélküli ütemezési entitás, az azonos szülőfolyamathoz tartozó szálak azonos memóriaterületen dolgoznak.
- **Kernel:** Az operációs rendszer alapvető eleme, amely a task-ok kezelését, ütemezést és a task-ok közti kommunikációt biztosítja. A kernel kódja hardver-függő (device driver), valamint hardverfüggetlen rétegekből együttesen épül fel.

# Task állapotok



- **Passzív (Dormant):** Passzív (nyugvó) állapot, amely jelentheti az inicializálás előtti vagy felfüggesztett állapotot.
- **Futásra kész (Ready):** A futásra kész állapotot jelöli. Fontos a task prioritási szintje és az is, hogy az éppen aktuálisan futó task milyen prioritási szinttel rendelkezik, ezek alapján dönti el az ütemező, hogy elindítja e a taskot.
- **Futó (Running):** A task éppen tevékenyen fut.
- **Késleltetett (Delayed):** Ez az állapot akkor lép fel, mikor a task valamilyen *időintervallumig* várakozni kényszerül. Rendszerint szinkron időzítő (*timer*) szolgáltatás hívása után következik be.
- **Várakozó (Waiting):** A task egy meghatározott eseményre várakozik. (Ez rendszerint valamilyen I/O periféria művelet szokott lenni.)
- **Megszakított (Interrupted):** A task-ot megszakították, vagy a megszakítás kezelő rutin éppen megszakítja a folyamatot (IRQ, INT).

# Ütemezési algoritmusok

Az ütemezési algoritmusoknak két fő típusa van:

- **Kooperatív** (=nem preemptív): A működési elve és alapötlete, hogy egy adott program vagy folyamat *lemond* a processzorról, ha már befejezte a futását vagy valamilyen I/O műveletre vár. Ez az algoritmus addig működik jól és hatékonyan, amíg a szoftverek megfelelően működnek (nem kerülnek végtelen ciklusba) és lemondanak a processzorról. Ha viszont valamelyik a program/folyamat nem mond le a processzorról vagy kifagy, akkor az egész rendszer stabilitását képes lecsökkenteni. A kooperatív algoritmus ezért soha nem fordulhat elő valós-idejű beágyazott operációs rendszerek esetében.
- **Preemptív**: az operációs rendszer részét képező *ütemező algoritmus vezérli* a programok/folyamatok futását. A preemptív multitask esetén az operációs rendszer elveheti a folyamatoktól a *futás jogát* és átadhatja más folyamatoknak. A valós idejű operációs rendszerek ütemezői minden esetben preemptív algoritmusok, így bármely program vagy folyamat leállása nem befolyásolja számottevően a rendszer stabilitását.

# Task-ok közötti kommunikáció

Mivel a rendszer működése közben a task-ok egymással párhuzamosan futnak ezért gondoskodni kell arról, hogy egyazon I/O perifériát, erőforrást vagy memória területet két vagy több task ne használjon egyszerre, mert abból hibás rendszerműködés alakulna ki.

A következő ismert módszerek állnak rendelkezésre:

- **Mutex (kölcsönös kizárási mechanizmus)**: ún. „locking” mechanizmus (csak a task amelyik zárolta, oldhatja fel)
- **Szemafor (semaphore)**: „signaling” mechanizmus (egyik task jelez a másiknak, hogy végzett, és átveheti az erőforrást) ~ 1 bit információ
- **Események (event flags)**: melyek több bit információ kicsérélésére is alkalmasak.
- **Postaláda (mailbox)**: amely akár komplexebb adatstruktúra átadására is szolgálhat.
- **Sor (queue)**: amely több mailbox tömbjében lévő tartalom átadására szolgál.
- **Cső (pipe vagy FIFO)**: amely direkt, folyamatos (akár streaming) kommunikációt tesz lehetővé két task között.

# (Beágyazott) Operációs rendszerek

Többféle csoportosítás lehetséges:

- Általános célú, vagy **beágyazott OS**
- Valós-idejű (időkritikus), vagy nem-időkritikus
- Nyílt forráskódú, vagy licenszelhető, stb.

Általános célú processzorok operációs rendszerei (OS):

- MS-DOS, Linux, Windows, stb.

Beágyazott processzorok *valós-idejű* operációs rendszerei (RTOS):

- Linux
- Android
- Micrium uC/OS
- QNX
- RTLinux
- **Windriver VxWorks (RT)**
- Windows Embedded, IoT, stb...



# Processzorok osztályozása

- Integráltság szerint:
  - uP/CPU: hagyományos **mikroprocesszorok** + fizikailag különálló memória + külső I/O periféria chipek (chipset)
  - **uC/MCU**: mikro**kontrollerek**: egyetlen chipen integrálva a processzor, a memória (ált. flash), és néhány I/O periféria
    - System-on-a-Chip (SoC) : egychipes rendszer
    - Kis méret és költség, alacsony disszipált teljesítmény
- Utasítás készlet szerint:
  - RISC vs. Nem-RISC (=CISC) ISA – utasításkészletű architektúrák
- Utasítás / Adat memória hozzáférés szerint:
  - Von Neumann (közös) vs. Harvard architektúrák (elkülönült)

Néhány architektúra típus: Intel 8051, ARM, AVR, PIC, MIPS, IBM PowerPC, x86 (32/64), Sun SPARC, stb.

# Technológiák és stratégiák

Élenjáró *technológiák* a beágyazott rendszerek tervezéséhez és megvalósításához – processzálo egységek csoportosítása:

- **(DSP):** Digitális jelfeldolgozó processzor alapú rendszerek
- **(MCU):** Mikrovezérlő-alapú rendszerek
- **(ASIC/ASSP):** Alkalmazás specifikus (berendezés orientált) integrált áramköri technológián alapuló rendszerek
- **(FPGA):** Programozható logikai kapuáramkörök technológián alapuló rendszerek
- **(CPU/MPU/GPU):** Mikroprocesszor, vagy grafikus processzor
  
- **SoC: System-on-a-chip:** olyan egychipes rendszer, amely a fentieket akár integrálva is tartalmazhatja!

Fejlesztési *stratégiák*:

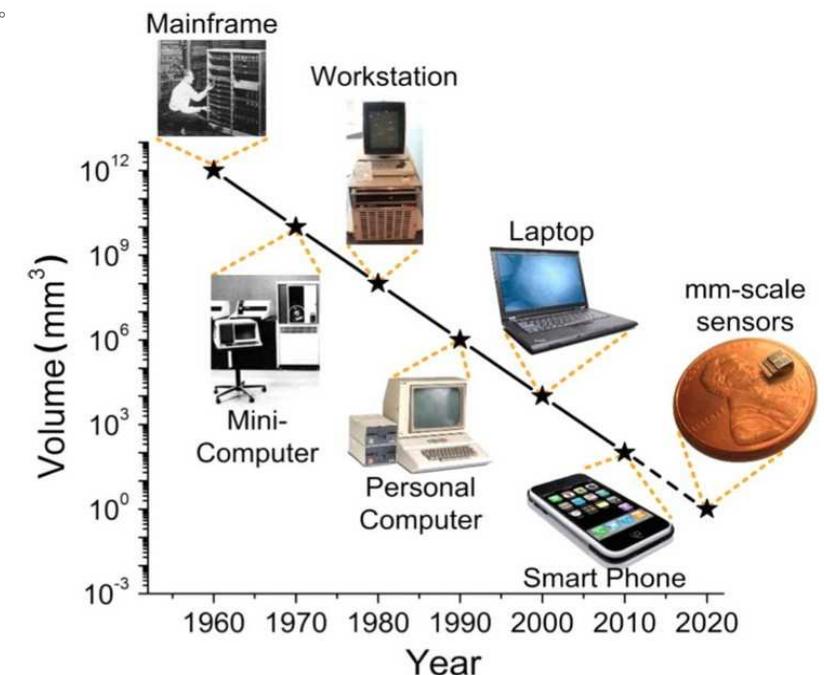
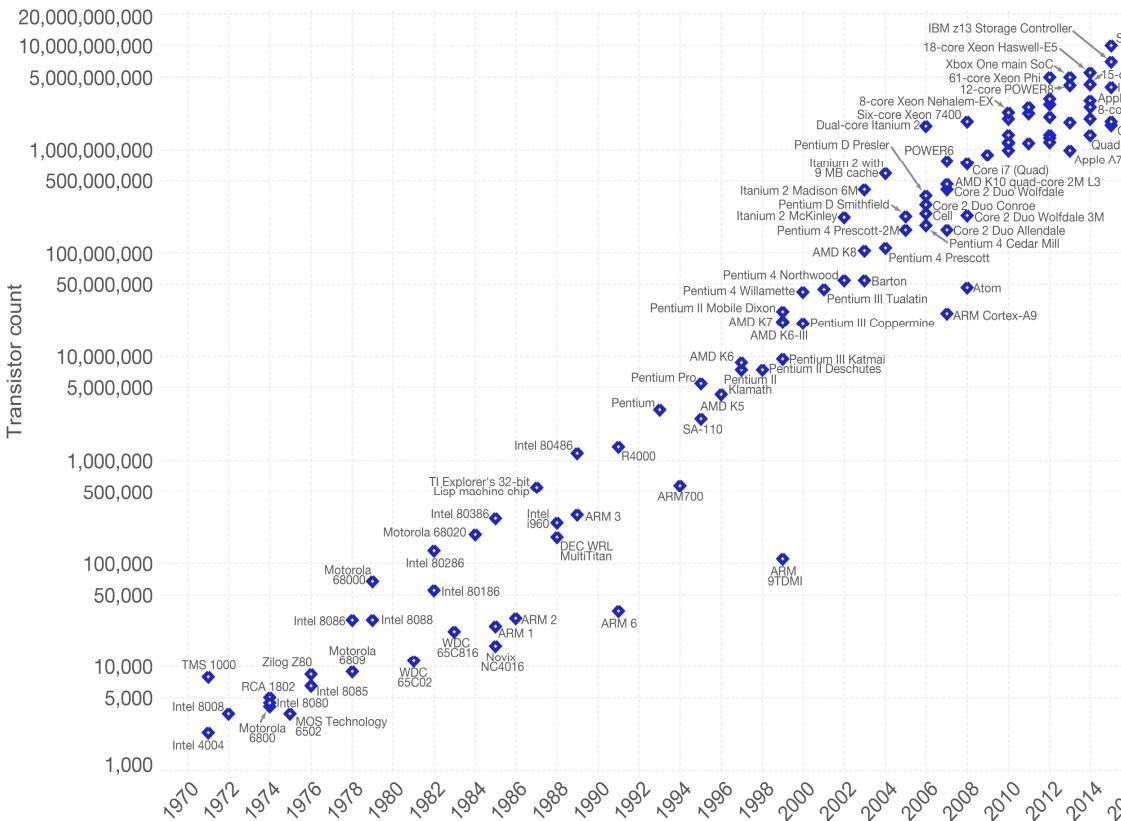
- HW/FW/SW co-design: HW/FW/SW részek együttes tervezése
- HW/FW/SW co-verification: HW/FW/SW részek együttes ellenőrzése és tesztelése

# I/O Perifériák

- Aszinkron soros kommunikációs interfészek: **RS-232, RS-422, RS-485, stb.**
- Szinkron soros kommunikációs interfészek: **I<sup>2</sup>C, SPI** stb.
- Univerzális soros busz: **USB**
- Multimédia kártyák: (SD) Smart Cards, (CF) Compact Flash stb.
- Hálózat: Ethernet (1GbE / 10 GbE / 100 GbE)
- Ipari hálózati ún. „Field-bus” protokollok: **CAN, LIN, PROFIBUS, IO Link**, stb.
- Időzítő-ütemezők: PLL(s), Timers, Counters, Watchdog timers (WDT)
- Általános célú I/O-k (General Purpose I/O - **GPIO**): LED-ek, nyomógombok, kapcsolók, LCD kijelzők, stb.
- Analóg-Digitális/Digitális-Analóg (ADC/DAC) konverterek
- Debug portok: **JTAG, ISP, ICSP, BDM, DP9**, stb.

# Technológiai fejlődés

- **Moore törvénye (1975):** 1 (ma 3) évente adott Si felületegységre eső tranzisztorszám duplázódása
- **Bell törvénye (1972):** számítógépek méretének fokozatos csökkenése (~10 évente új számítógép osztályok, platformok megjelenése)





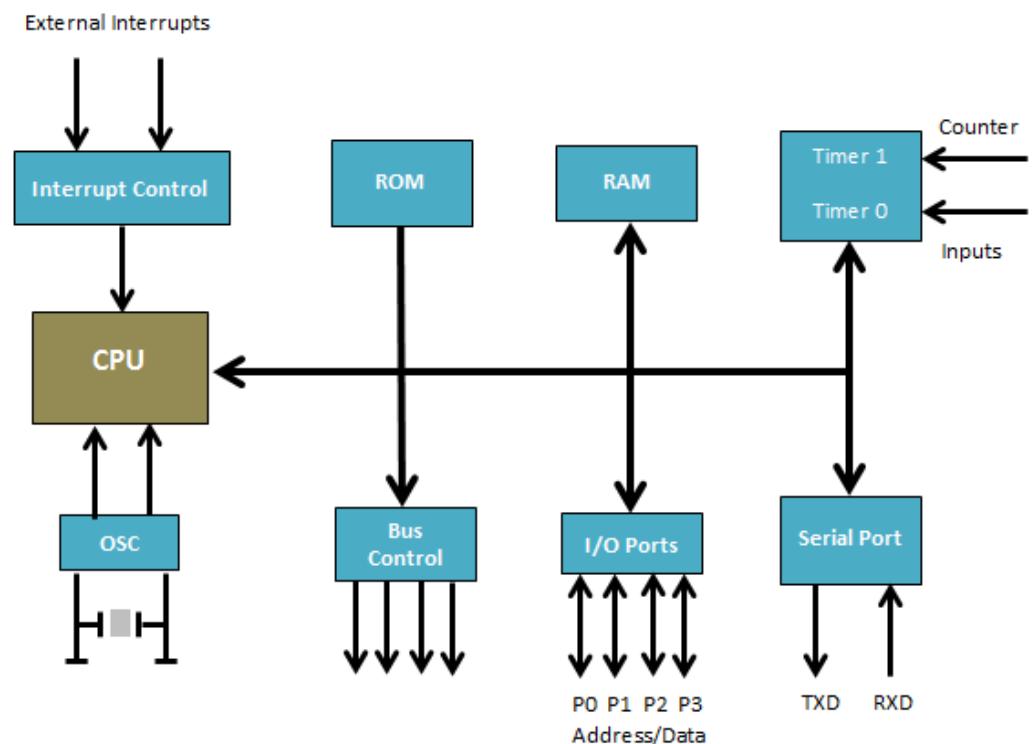
# Mikrovezérlők

# Mikrovezérlők

- Mikrovezérlő = mikrokontroller (MCU – Micro Controller Unit, vagy **UC**, vagy  **$\mu$ -controller**)
  - *Egy olcsó, kis méretű és fogyasztású „számítógép” egy integrált áramkörön*
  - **Beágyazott rendszerek** alapvető építő eleme
    - *SoC = System-on-a-Chip, egychipes számítógép, feldolgozó egysége lehet az MCU (DSP, FPGA is akár)*
    - *Alkalmazásuk: lásd Beágyazott rendszerek.*

# Mikrovezérlők általános felépítése

- Processzor (CPU MPU)  
mag(ok)
- Memória (Harvard architektúra):
  - ROM/Flash/(EE)PROM: program (utasítás) memória (KB)
  - RAM: adat memória (KB)
- Interrupt/(WD)Timer/Counter
- OSC: oszcillátor (órajel)
- I/O perifériák, portok
  - Analóg, Digitális, vagy Mixed-signal funkciók
    - Manapság: IoT funkciók (érzékelők, aktuátorok, kommunikációs IF-ek)
  - ADC/ DAC konverterek



# Mikrovezérlők tulajdonságai

- Előnyök:
  - Olcsó!
  - Egyszerű felépítés
  - Kis disszipáció ( $\text{mW}$ ,  $\mu\text{W}$ ,  $\text{nW}$  – órajel/üzemmód függő)
  - Könnyű integrálhatóság, beépíthetőség, széleskörű interfész támogatottság
  - CPU: Speciálisan DSP műveleteket is támogathat (pl. Microchip dsPIC)
- Hátrányok:
  - Kis sebesség/ órajel (~10 - 100+ MHz)
  - Relatíve kis pontosság
    - (8-,16-,32-bites adat, cím, vezérlő, utasításbuszok)
  - Korlátozott mértékű funkcionalitást biztosít
    - Komplex feladatokhoz több, vagy nagyobb számítási-tárolási kapacitású eszköz kell

# Fontosabb gyártók



- Atmel (ma Microchip): AVR8, AVR32, Intel 8051
- ARM: Cortex-M proc. magok
- Microchip: PIC, dsPIC,
- Texas Instruments: MSP, C2000
- Renesas: RL78-16, RX-32
- Freescale (ma NXP),
- NXP Semiconductors: LPC sorozat
- ST Microelectronics: STM8, STM16, STM32
- Cypress: PSoC családok

# Mikrokontrollerek osztályozása

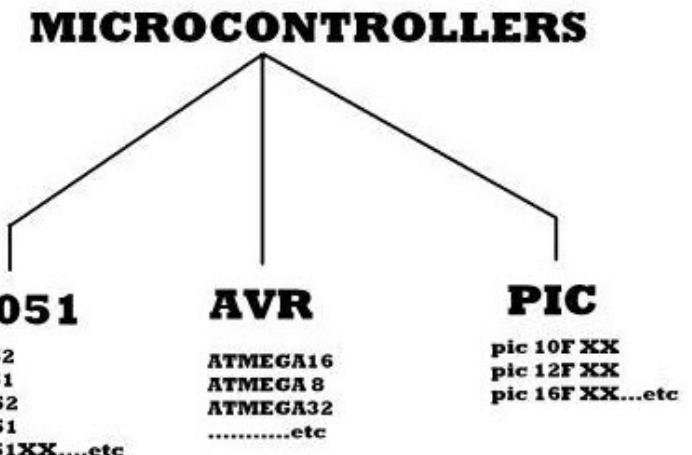
## ■ Kategóriák (gyártónként keveredhetnek):

- **Busz-szélesség?** 8-/16-/32-bites
- **Utasítás készlet (ISA)?:** RISC vs. CISC

- RISC: csökkentett utasítás készlet
- CISC: kibővített/komplex utasítás készlet

## □ **Memória elvek?**

- Neumann vs. Harvard elv
- Belső vs. külső memória



# Kategória: Busz szélesség

- **8-bit:** belső busz 8-bites, ALU műveletek.
  - Pl: Intel 8031/8051, PIC1x, Motorola MC68HC11... családok
- **16-bit:** 2x-es pontosság, belső busz, ALU
  - Pl: Intel 8051XA, PIC2x, Intel 8096, Motorola MC68HC12... családok
- **32-bit:** 4x-es pontosság, belső busz, ALU
  - Pl: Intel/Atmel 251, PIC3x... családok

# Kategória: Utasítás készlet (ISA)

ISA = Instruction Set Architecture

## ■ CISC = Komplex/kibővített utasítás készlet:

- nagyobb architektúra, sok utasítással.
- 1 utasításban több elemi utasítás van, (1 utasítás / több órajel)
- változó utasítás hossz → nehezebb dekódolni, majd végrehajtani.
- komplex művelet kevesebb CISC utasítás sorral írható le!

## ■ RISC = Csökkentett utasítás készlet:

- csak a feladatra optimalizált, kevés számú utasítás van
- azonos utasítás hossz → könnyebb dekódolás (1 utasítás / 1 órajel)
- komplex művelet sok elemi RISC utasítás sorral írható csak le.

# Kategória: Memória

## ■ Szervezés elve:

- Von Neumann (Princeton) elv: adat/utasítás fizikailag közös memóriában
- Harvard elv: adat/utasítás fizikailag elkülönült memóriában

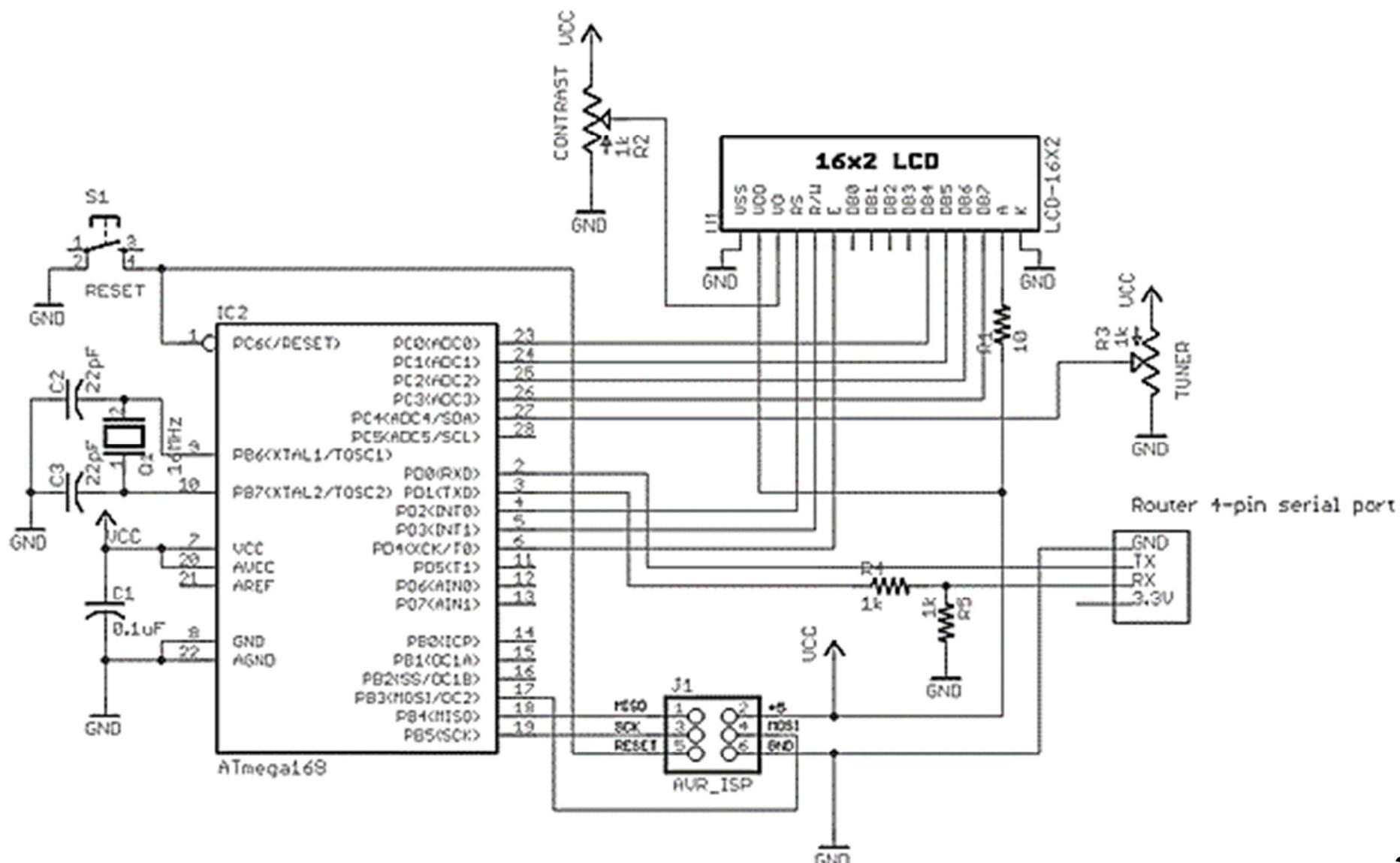
## ■ Hol található ?

- Belső (on-chip) – beágyazott memória vezérlő (pl. Intel 8051)
- Külső (off-chip) memória vezérlő: nincs belső / on-chip memória (pl: Intel 8031 – nincs program memória)

# MCU architektúrák (magok)

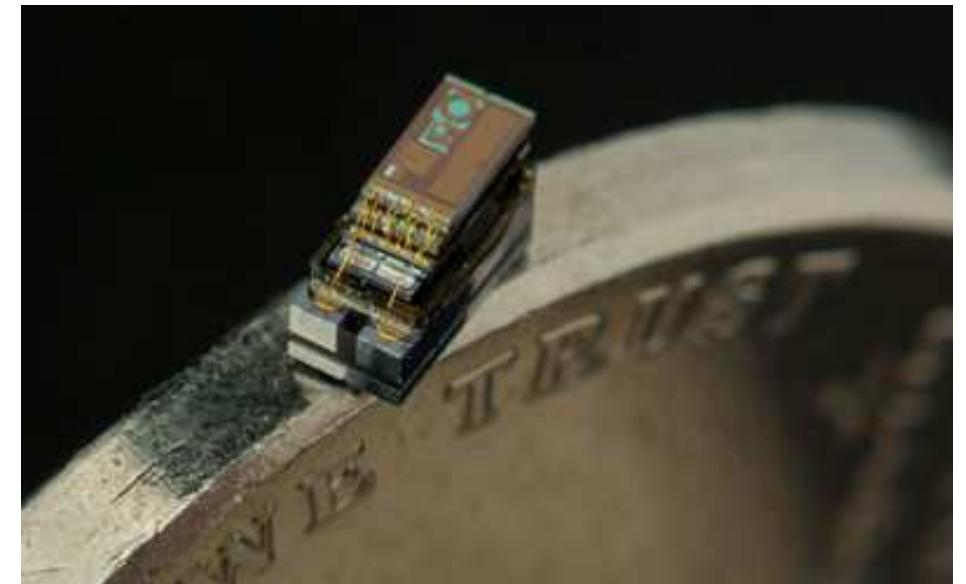
- ARM: harvard RISC
  - Manapság a legnépszerűbb, skálázható magokat gyártanak, több gyártó is integrálja a magokat (pl. STMicro)
- AVR (Atmel): Harvard RISC
  - Népszerű Atmega családok (pl. Arduino kártyák)
- Intel 8051: Harvard CISC
  - Népszerű, beágyazott MCU mag több gyártónál (pl Cypress)
- Renesas: RX-32 - Harvard CISC
- Microchip:
  - PIC16/18 ... - Harvard RISC
  - Gyorsabb, könnyebb programozhatóság

# Pi. Atmel – karakter LCD vezérlő



# „Világ legkisebb számítógépe”

- 2018. jún (University of Michigan, USA)
- **M<sup>3</sup> : Michigan Micro Mote: smart-sensor**
  - Hőmérséklet, nyomás,
  - Képalkotó szenzor  
(160x160 pixel)
  - 1 mm<sup>2</sup> felületű!
  - 2 nA disszipáció  
(standby mód)
  - CPU + MEM + PWR  
RF, battery



# Arduino vs. Raspberry Pi



## ■ Arduino – Atmel/Microchip MCU alapú fejlesztő kártya

- Kisebb órajelű (~x10 MHz) **MCU** mag, kis belső memória, kis bitszélesség (8-, 16 bit)
- Nincs külső memória, nincs OS kezelése, nem real-time eszköz.
- Jó bővíthetőség: „shield”-ek
- Főként egyszerű szabványokat, GPIO-kat kezel, van ADC.
- Olcsó, népszerű, rengeteg szenzor illeszthető, de kisebb komplexitású fejlesztési célokra.  
Ára: \$5-15 (platform függő)

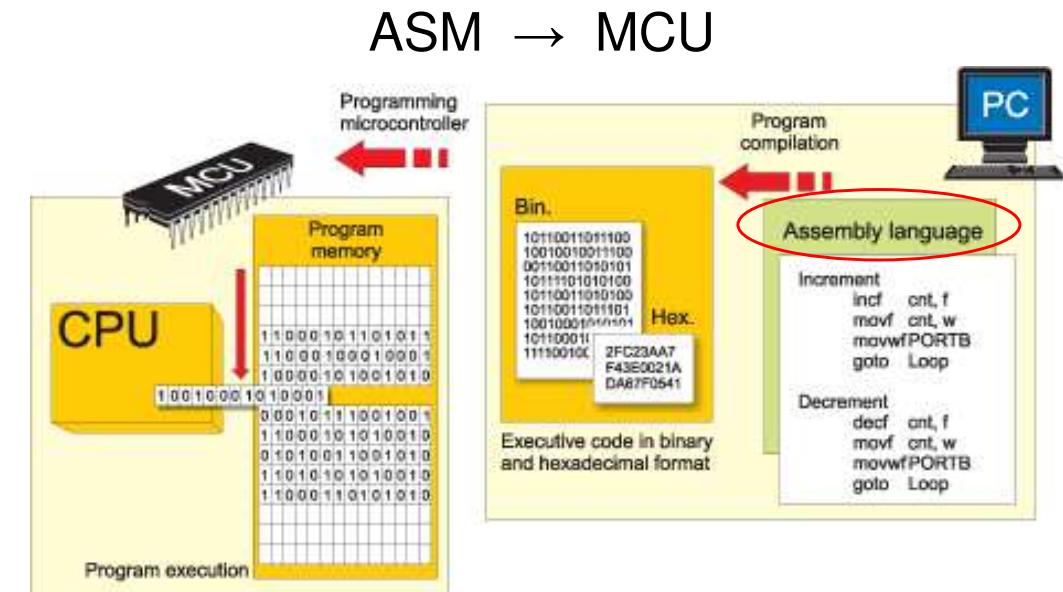
## ■ Raspberry Pi – ARM alapúáltalános célú sz.gép, fejlesztő kártya („single board computer”)

- Dedikált, nagy órajelű **CPU** magok (ARM 32/64 bit ~x100 MHz, memória (LDDR3/4), GPU mag, HDMI stb.)
- Bővíthetőség: SDCard (OS boot), WIFI, BLE, CamIF, de nincs ADC.
- OS/RTOS (HW-es) kezelése Nagyobb komplexitás, több funkció, de drágább.
- Ára: \$ 30- 50 (platform függő)

# Mikrovezérlők programozása

## ■ Programozási nyelvek (compiler függő):

- ASM – assembly (régen, hagyományos)
- C (C++)
- Python, ...
- Egyéb: Interpreter FW elérhetővé tehet más nyelveget is: BASIC...



## ■ Integrált fejlesztő környezetek (IDE), pl.:

- Texas Instruments - Code Composer Studio
- Arduino IDE
- Microchip/PIC – MPLAB
- ARM – Keil MDK / ARM DS-5, stb.

