

Pázmány Péter Catholic University,
Information Technology Department

Digitális Rendszerek és Számítógép Architektúrák

Szolgay Péter

szolgay@itk.ppke.hu

Jegyzetek, segédanyagok:

- Könyvfejezetek:
  - http://www.knt.vein.hu
    -> Oktatás -> Tantárgyak -> Digitális Rendszerek és Számítógép Architektúrák
    (chapter01.pdf)
- Fóliák, óravázlatok .ppt (.pdf)
- Feltöltésük folyamatosan

Boole-algebrai azonosságok:

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

2.) $A + 0 = A$

3.) $A + 1 = 1$

4.) $A + A = A$

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

6.) $A \cdot 1 = A$

7.) $A \cdot 0 = 0$

8.) $A \cdot A = A$

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

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

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

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

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

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

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

16.) $A + B \cdot C = (A + B) \cdot (A + C)$

17.) $A \cdot (B + C) = A \cdot B + A \cdot C$

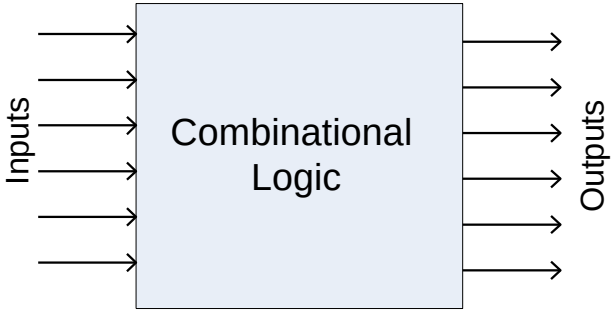
De-Morgan azonosságok:

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

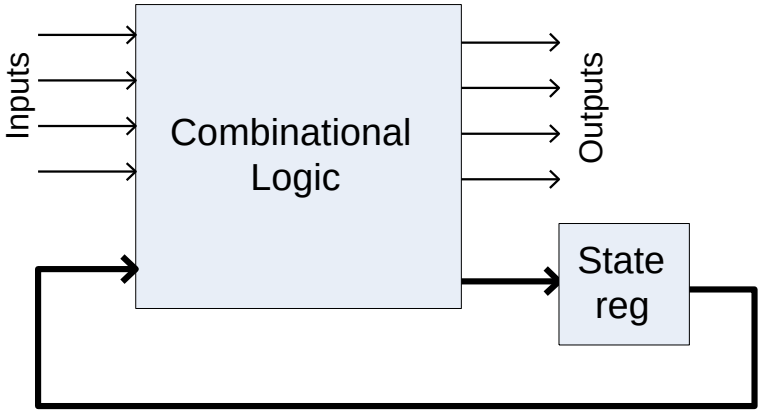
19.) $\overline{A \cdot B} = \overline{A} + \overline{B}$

Kombinációs vs sorrendi hálózatok:

- Kombinációs hálózat:



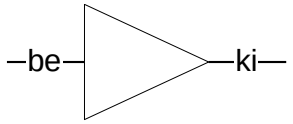
- Sorrendi hálózat:

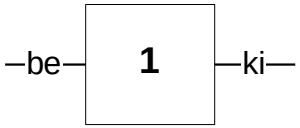


Egyváltozós logikai függvények:

- Jelmásoló (jel-erősítés):

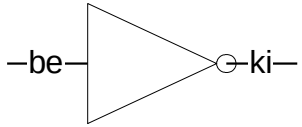
be	ki
0	0
1	1

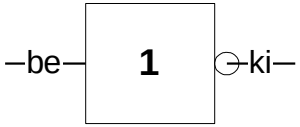




- Negálás - Inverter (NOT):

be	ki
0	1
1	0

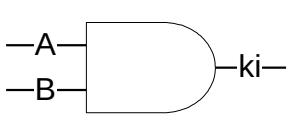


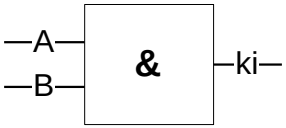


Kétváltozós logikai függvények:

- ÉS (AND):

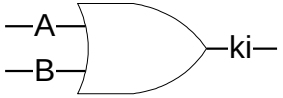
A	B	ki
0	0	0
0	1	0
1	0	0
1	1	1

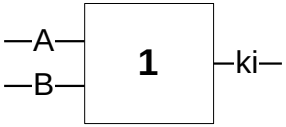




- VAGY (OR):

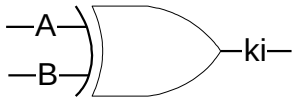
A	B	ki
0	0	0
0	1	1
1	0	1
1	1	1

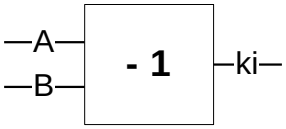




- Antivalencia (XOR):

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



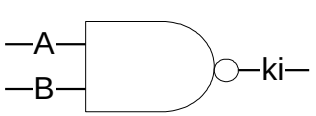


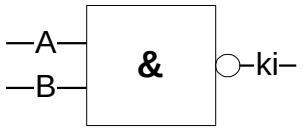
Kétváltozós log.függv. (folyt.):

Univerzálisan teljes rendszert a NAND illetve NOR függvény alkot!

- NEM-ÉS (NAND):

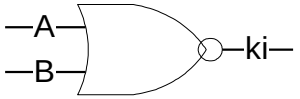
A	B	ki
0	0	1
0	1	1
1	0	1
1	1	0

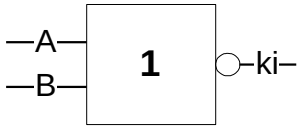




- NEM-VAGY (NOR):

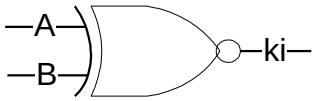
A	B	ki
0	0	1
0	1	0
1	0	0
1	1	0

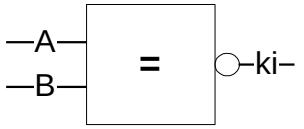




- Ekvivalencia (NXOR):

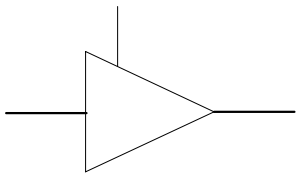
A	B	ki
0	0	1
0	1	0
1	0	0
1	1	1



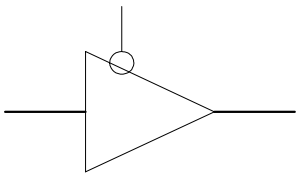


Tri-State Buffer:

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



High-true enable



Low-true enable

Digitális Rendszerek és Számítógép Architektúrák

1. előadás: Bevezetés, számítógép generációk

Előadó: Vörösházi Zsolt
Szolgay Péter

Feltételek:

- Könyv: L. Howard Pollard –
  Computer Design and Architecture (Prentice-Hall 1990)
- 8 fő fejezet
- Követelmények: 2 évközi ZH lesz
  - *2014. ápr.2*és *május 21.*
  - kisZH-k illetve nagy beadandó feladatok (plusz pontokért)
- Megajánlott jegy: **mindkét ZH >= 4**
- Vizsgára bocsátás feltétele: **mindkét ZH >= 2**
- Óralátogatás kötelező
- Vizsga: szóbeli-írásbeli
- Szigorlati/Államvizsga tárgy!

Kapcsolódó jegyzet, segédanyag:

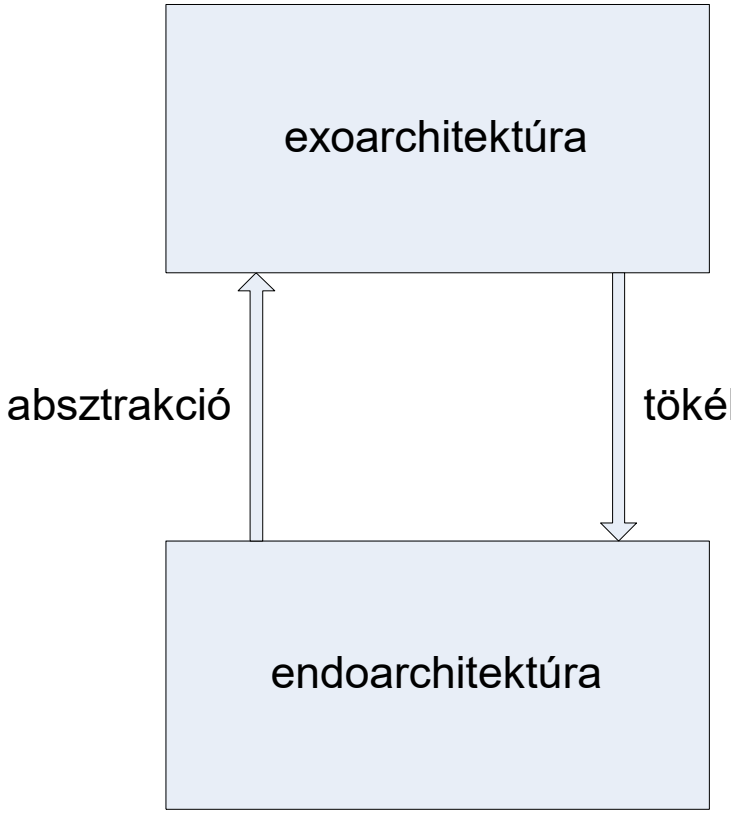
- Angol nyelvű könyv:
  http://www.knt.vein.hu
  → Oktatás → Tantárgyak → Digitális Rendszerek és Számítógép Architektúrák (nappali)
  (chapter1/…/8.pdf)
  - Bevezetés: Számítógép Generációk (chapter01.pdf)

- Fóliák, óravázlatok .ppt (.pdf)
- Feltöltésük folyamatosan
- Magyar nyelvű szigorlati segédlet

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

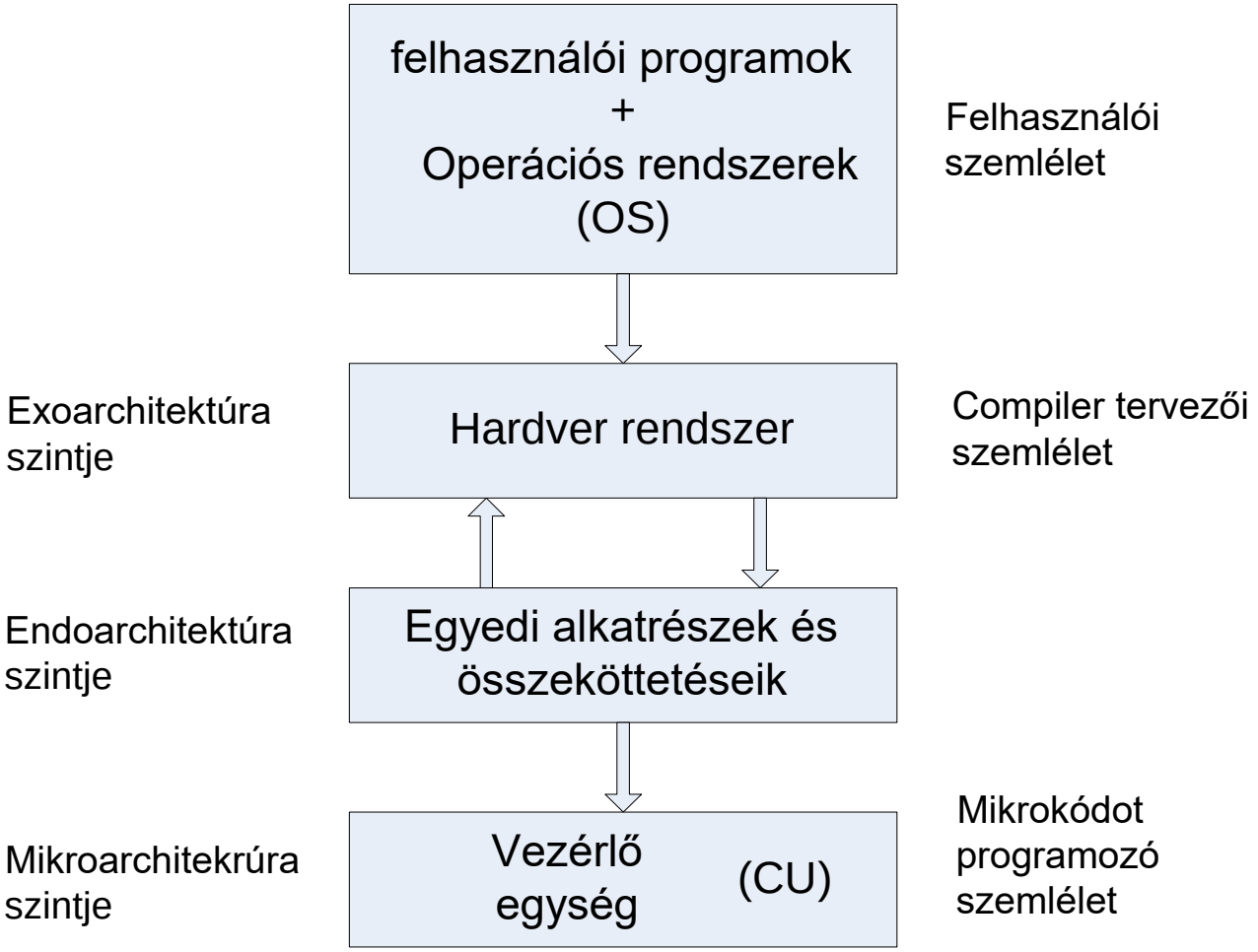
Exoarchitektúra – endoarchitektúra:



Compiler tervezői aspektus:
- utasítás készlet
- utasítás regiszterek
- címzési módok

Vezérlő egység tervezői aspektusai:
- jelek
- komponensek

Számítógép architektúra definíciója:



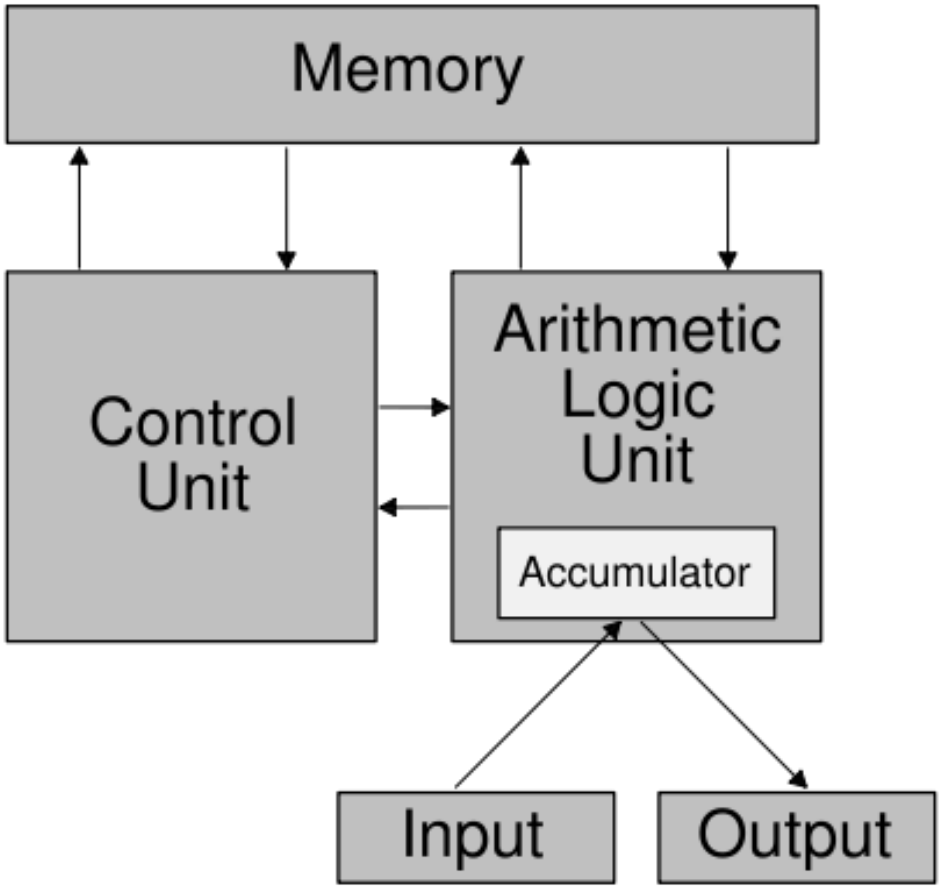
Számítógépes rendszerekkel szembeni tervezői követelmények:

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

Neumann, Harvard számítógép architektúrák

A.) Neumann architektúra

- **Számítógépes rendszer modell:**
  - CPU, CU, ALU
  - Egyetlen különálló tároló elem (utasítások és adatok részére)
  - Univerzális Turing gépet implementál
  - „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 számítógépei is ezt az architektúrát követik.

Neumann elvek

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

Fix vs. **tárolt** programozhatóság

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

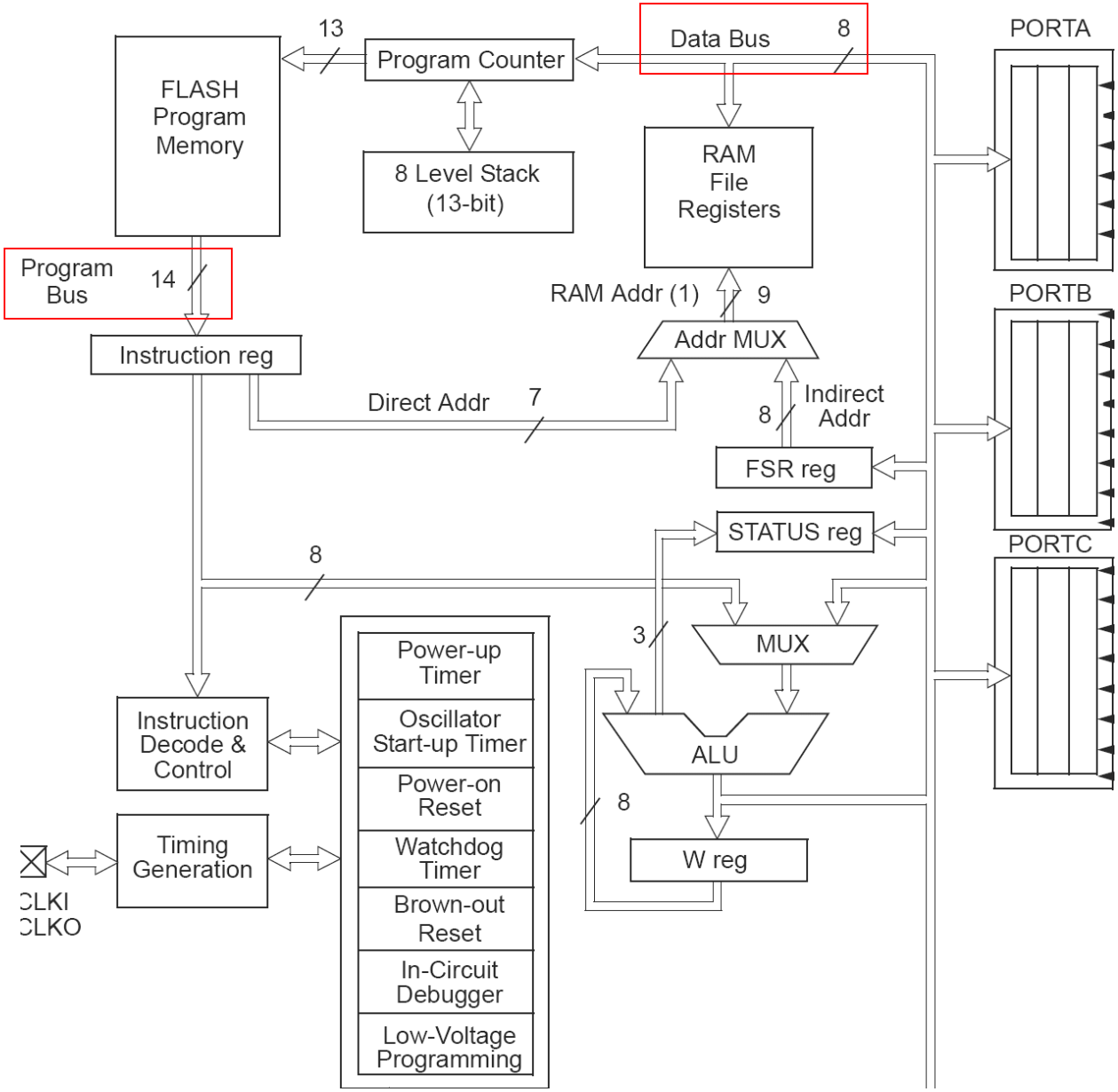
Neumann architektúra hátrányai

- „Önmagát változtató” programok (self-modifying code):
  - Már eleve hibásan megírt program „kárt” okozhat önmagában ill. más programokban is: „malware”=„malfunction”+„sw”.
  - OS szinten: rendszer leállás
  - Pl., Buffer túlcsordulás: kezelése hozzáféréssel, memória védelemmel
- **Neumann „bottleneck”**: sávszélesség korlát a CPU és memória között (ezért kellett bevezetni a cache memóriát), amely probléma a nagymennyiségű adatok továbbítása során lépett fel.
- 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.
  - Eredet: Harvard MARK I. (relés alapú rdsz.)
  - További példák:
    - Intel Pentium processzor család L1- szintű különálló *adat- és utasítás-cache* memóriája.
    - Ti320 DSP jelfeldolgozó processzorok (RAM, ROM memóriái) - / bővebben a DSP-s fóliákon /
    - Beágyazott (embedded) rendszerek: MicroBlaze, PowerPC (FPGA-n) buszrendszerei, memóriái.
    - Mikrovezérlők (MCU) különálló utasítás-adat buszai és memóriái (PIC-MicroChip, Atmel stb.)

Példa: PIC 14-bites mikrovezérlő



Harvard arch. tulajdonságai

- Nem szükséges a memória (shared) osztott jellegének kialakítása:
  - **+** Szóhosszúság, időzítés, tervezési technológia, memória címzés kialakítása is különböző lehet.
  - Az utasítás (program) memória gyakran szélesebb mint az adat memória (mivel több utasítás memóriára lehet szükség)
  - Utasításokat a legtöbb rendszer esetében ROM-ban tárolják, míg az adatot írható/olvasható memóriában (pl. RAM-ban).
  - **+** A számítógép különálló buszrendszere 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 *utasítás-szót* is olvasható *adatként* lehet elérni.
  - *Konstans adat* (pl: string, inicializáló érték) utasítás memóriába töltésével a változók számára további helyet spórolunk meg az adatmemóriában
  - Mai modern rendszereknél a Harvard architektúra megnevezés alatt, ezt a módosított változatot értjük.
  - *Gépi* (alacsony) szintű assembly utasítások

Harvard architektúra hátrányai

- Az olyan egychipes rendszereknél (pl. SoC: System On a Chip), ahol egyetlen chipen van implementálva minden funkció, nehézkes lehet a különböző memória technológiák használata az utasítások és adatok kezelésénél. Ezekben az esetekben a Neumann architektúra alkalmazása lehet megfelelőbb.
- A magas szintű nyelveket (pl ANSI C szabvány) sem közvetlenül támogatja (nyelvi konstrukció hiánya az utasítás adatként való elérésére) – assembler szükséges

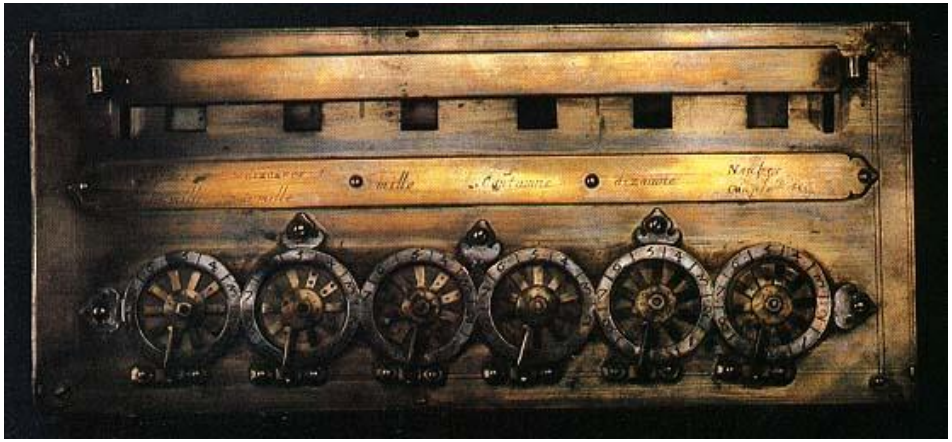
Harvard – Neumann együttes architektúra megvalósítás

- Mai, nagy teljesítményű rendszereknél a kettőt együtt is lehet, kell alkalmazni.
- Példa: Cache rendszer
  - Programozói szemlélet (Neumann): cache ‘miss’ esetén a fő memóriából kell kivenni az adatot (cím -> adat)
  - Rendszer, hardver szemlélet (Harvard): a CPU on-chip cache memóriája különálló adat- és utasítás cache blokkokból áll.

Számítógép generációk

Eredet - korai számítási eszközök I:

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

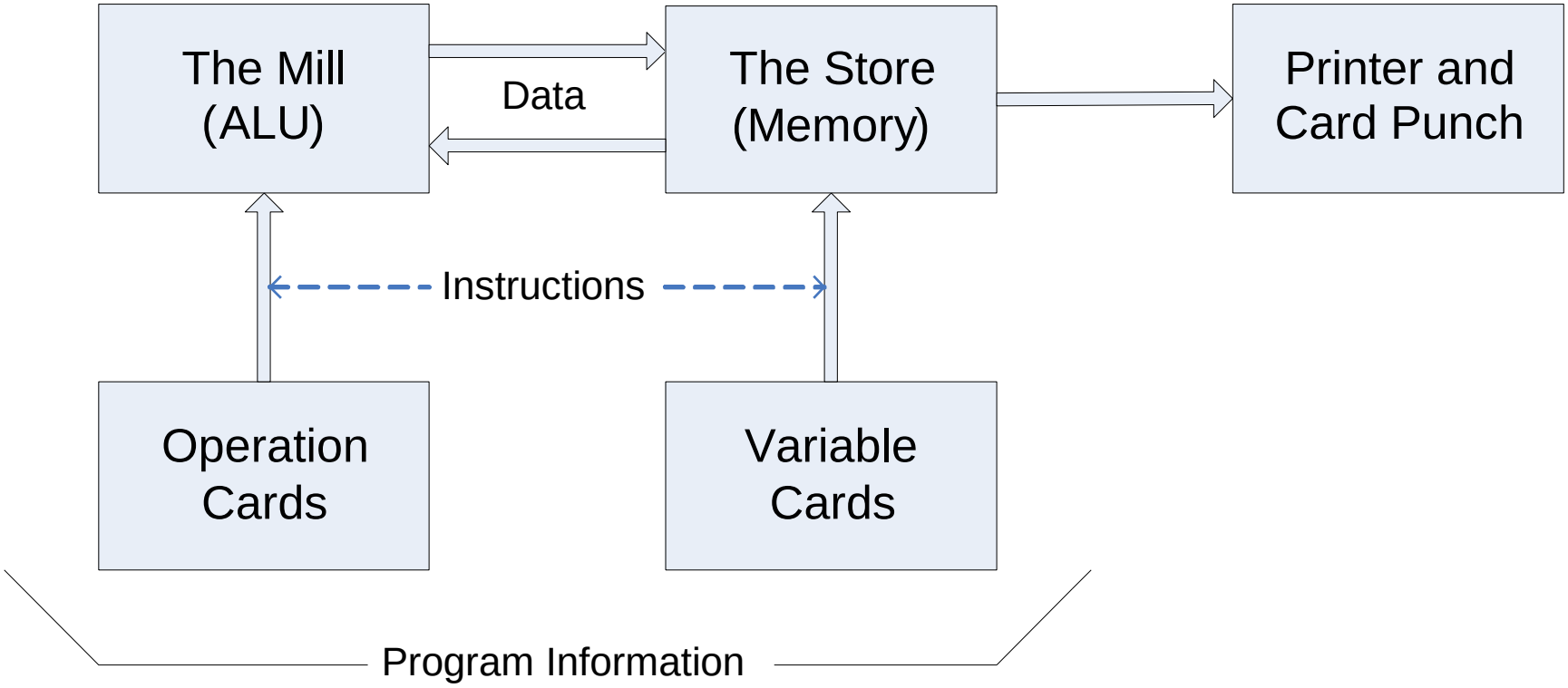




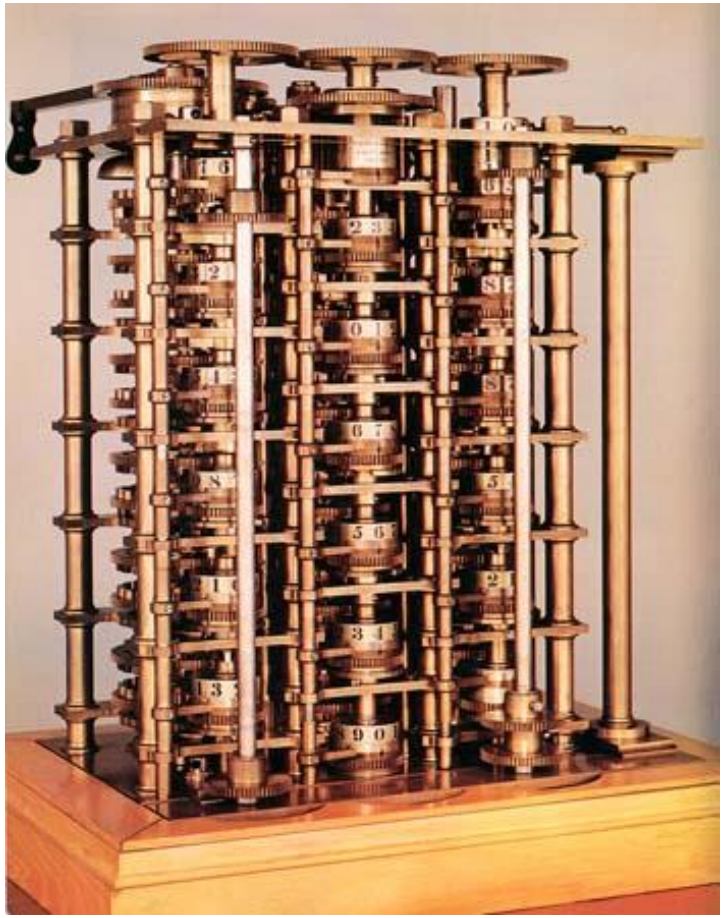
Eredet - korai számítási eszközök I (folyt.)

- 1823: Babbage
  - *Differencia Gép*: véges differencia módszer, ciklusos végrehajtás, automatikusan generált mat. táblákat
  - *Analitikus Gép*: mai gépekkel szembetűnő hasonlóság, mat. fgv-ek végrehajtása. MILL – aritmetika: 4 alapműv (‘+’ 1sec, ‘*’ 1 min alatt), felt. elágazást is támogatta. Memóriája számoló „korongos”: 1000 db 50 jegyű számot tárolt.

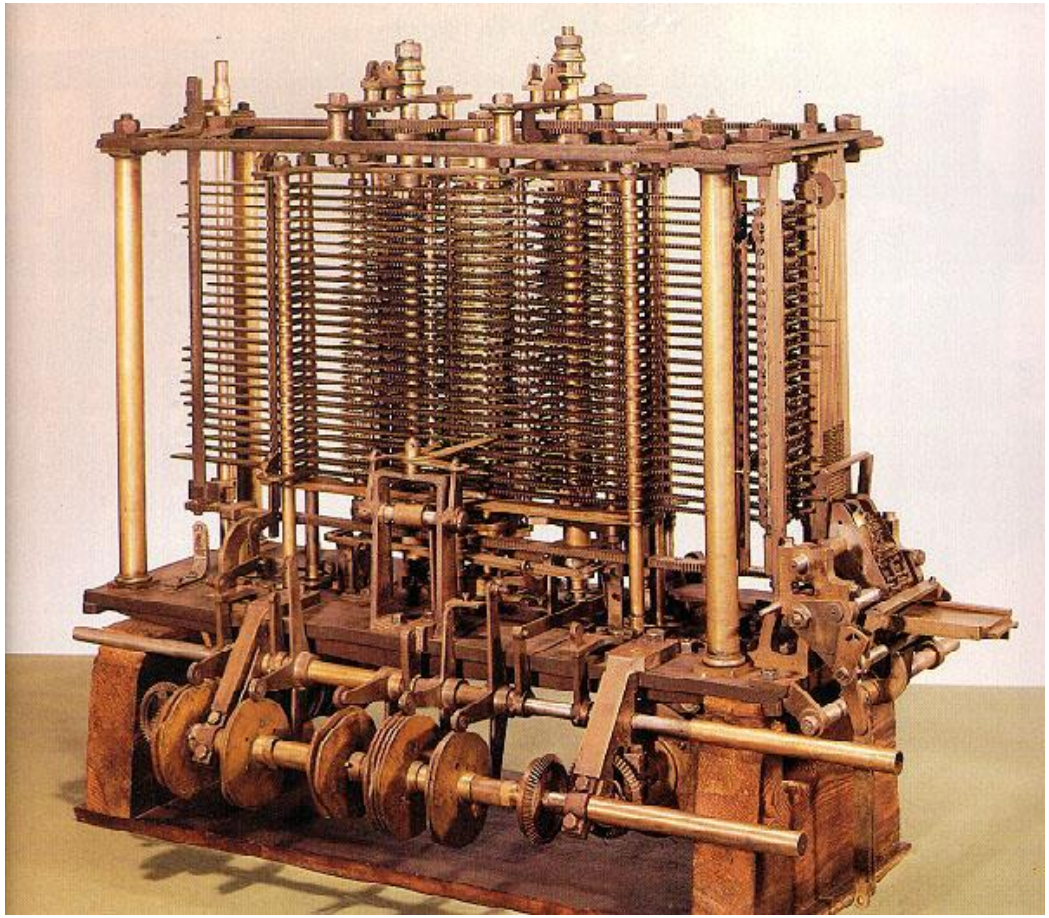
Babbage – Analitikus Gép



Babbage



Differencia gép

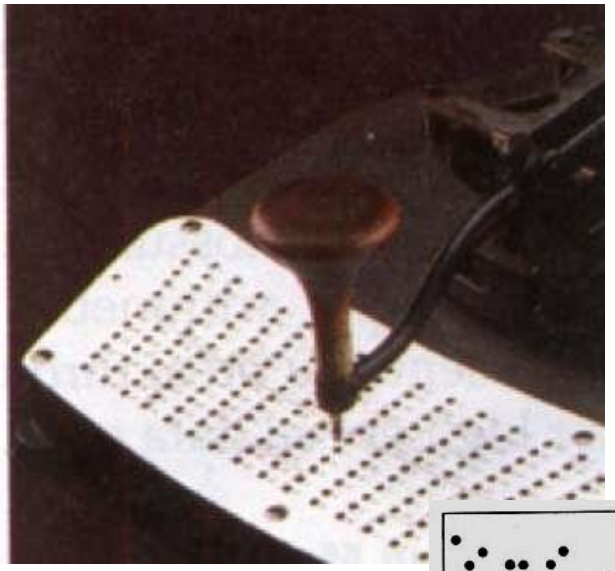


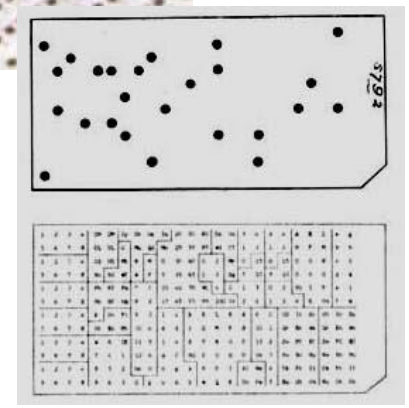
Analitikus gép

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



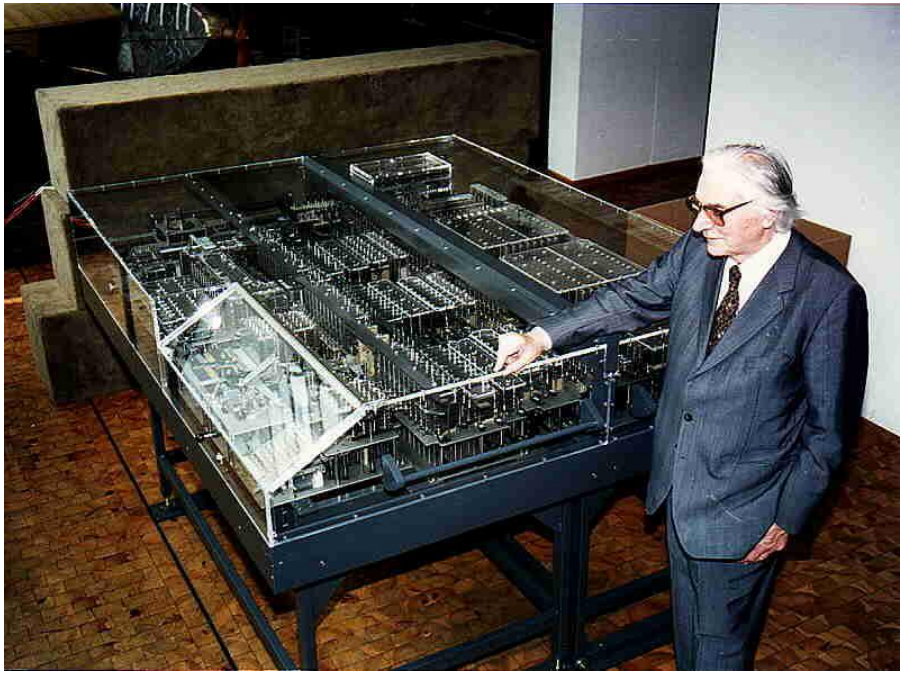


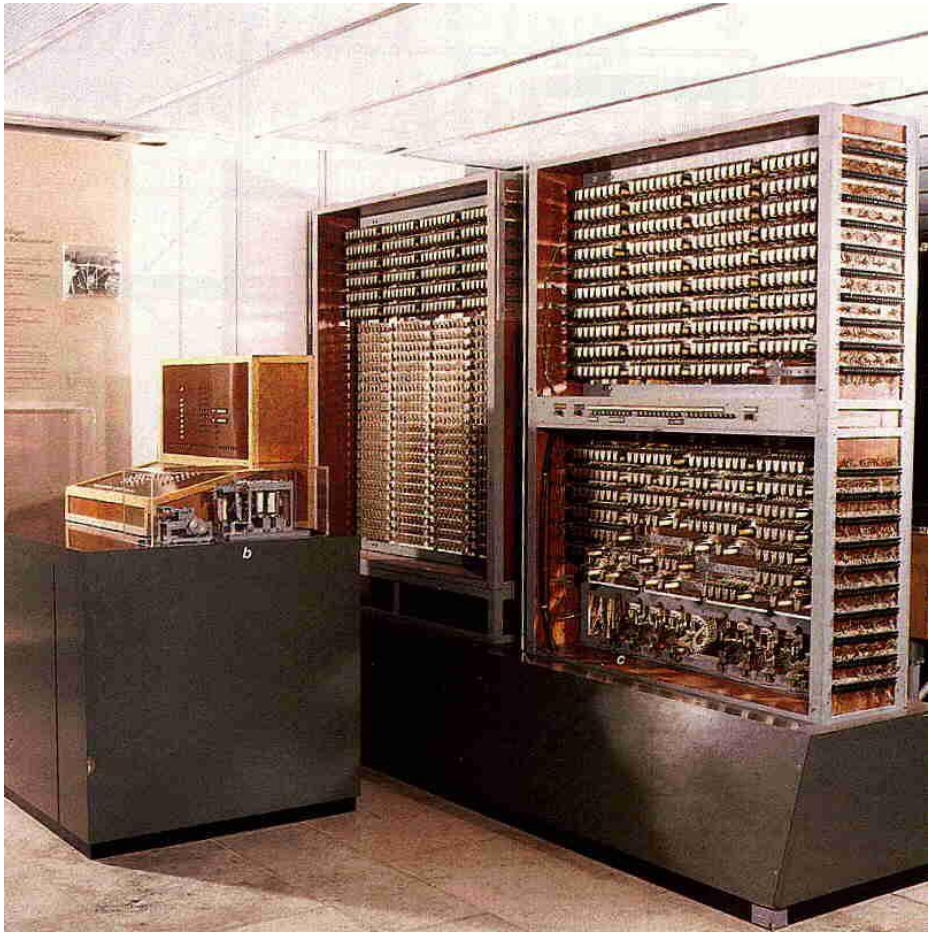


Jacquard „szövőgépe”

Hollerith - lyukkártya

Zuse Z1 és Z3

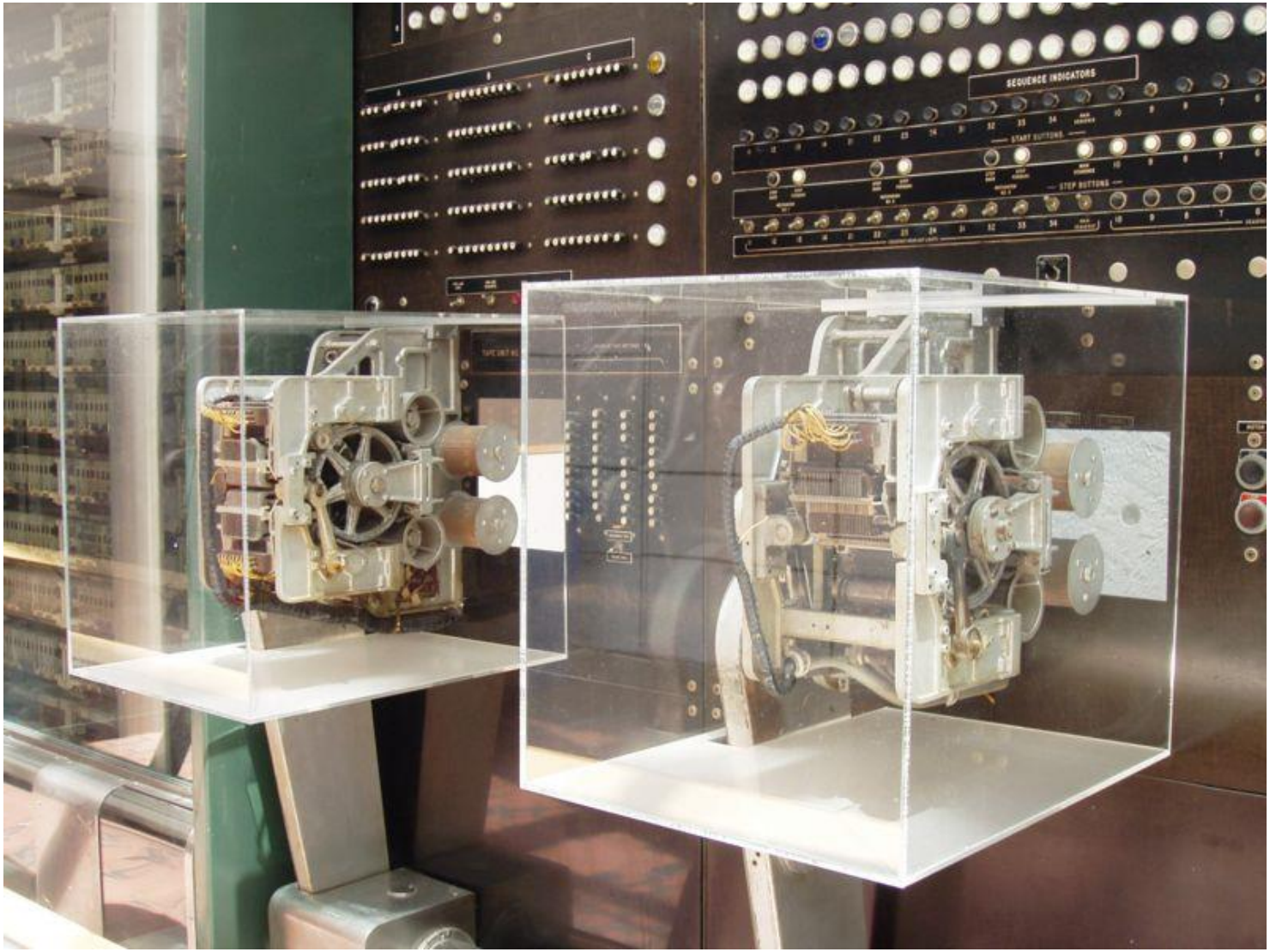




Harvard MARK I

- Howard H. Aiken (Harvard University) – 1944
- Relés alapú aritmetika, mechanikus, korai szgép rendszer. Korlátozott adattároló képesség. (72 db 23 bites decimális számot tárol)
- ***Harvard architektúra***
  - Lyukszalagon 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

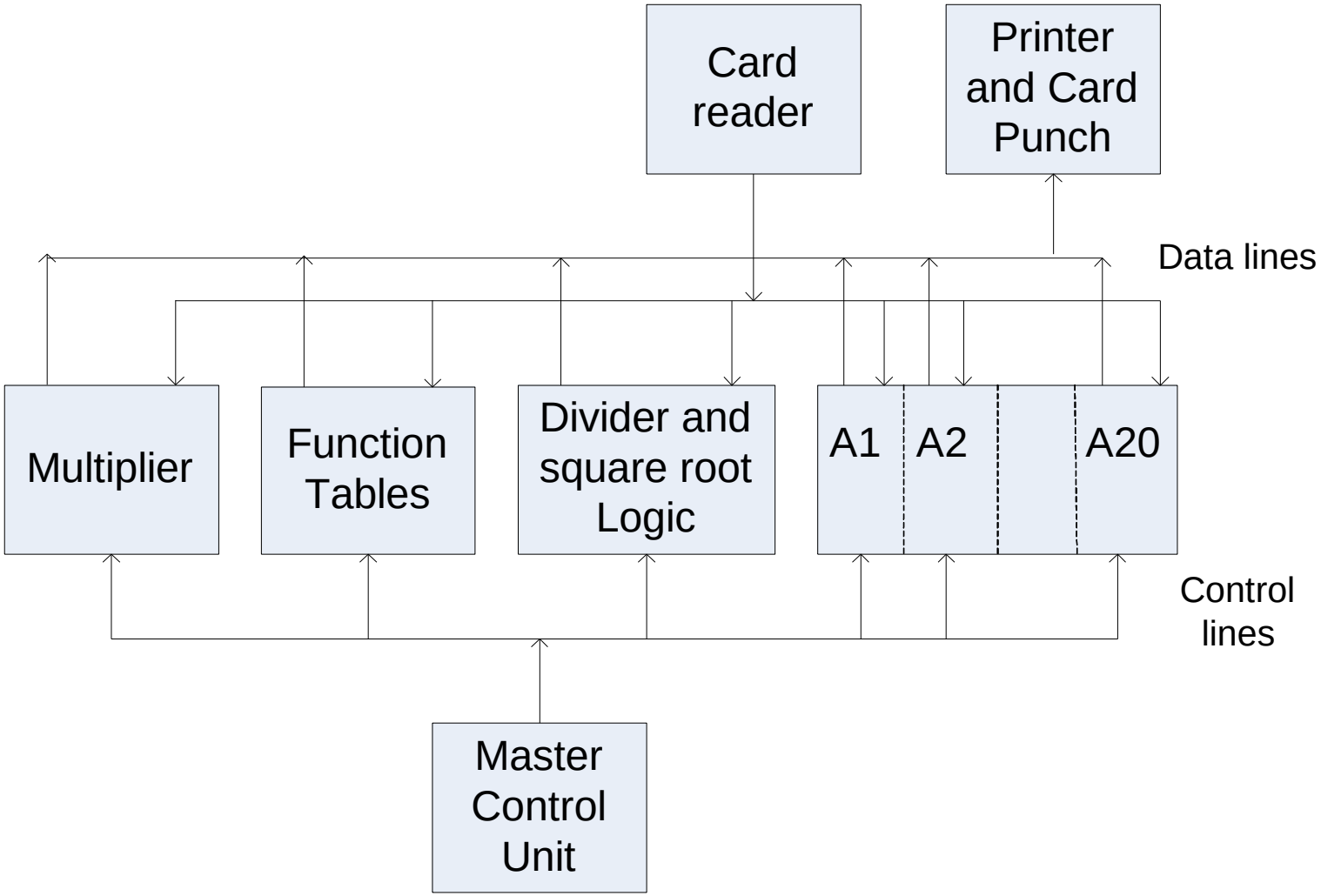
Atanasoff - Berry



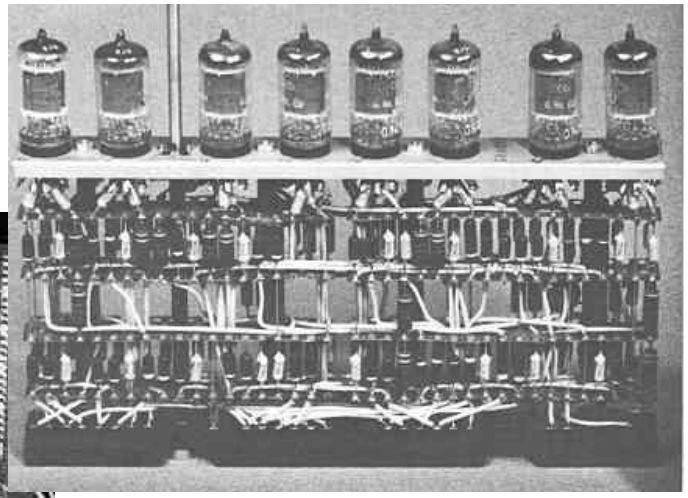
I. Generáció (1952-ig)

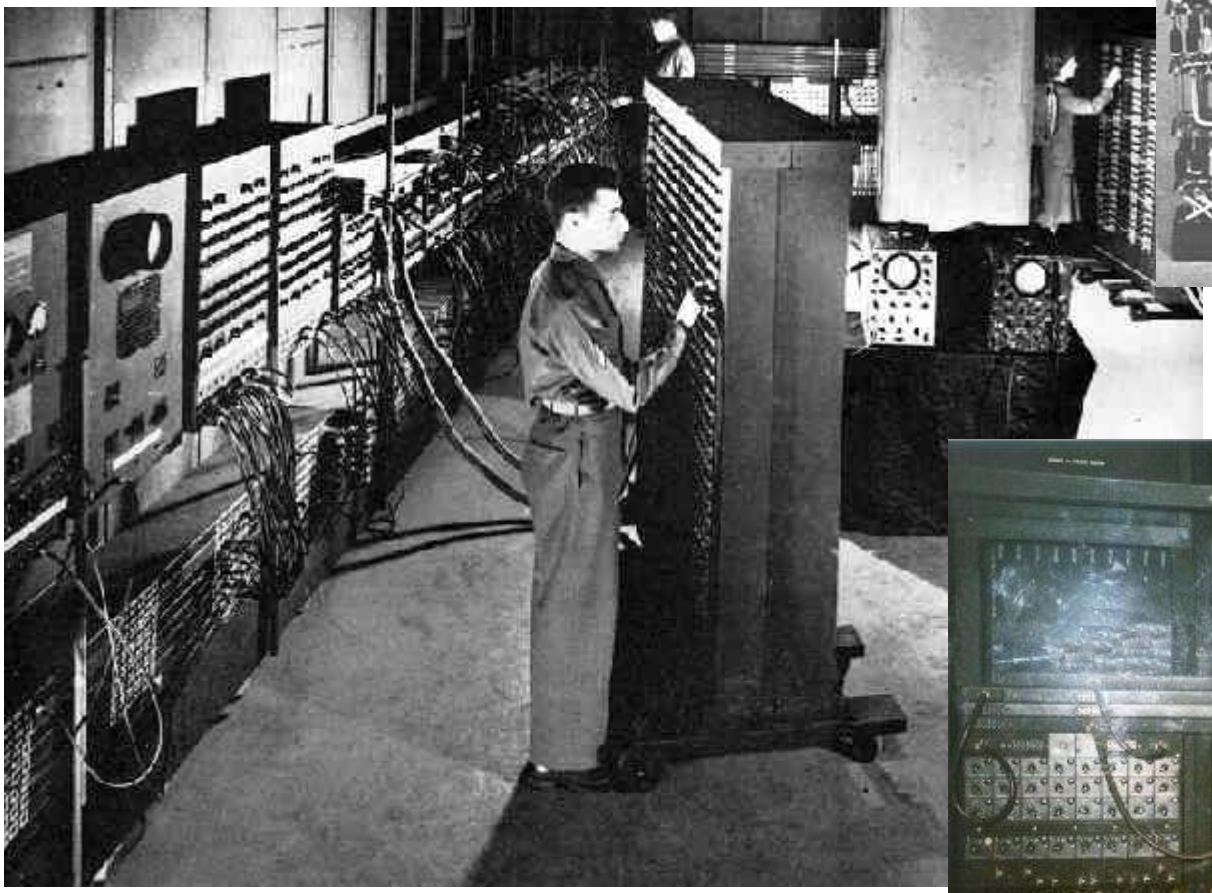
- 1943: **ENIAC**: elektromos numerikus integrátor és kalkulátor (Pennsylvania) Mauchly, Eckert
  - 18000 elektroncső, mechanikus, kapcsolók
  - Gépi szintű programozhatóság, tudományos célokra
  - Összeadás: 3ms
  - 20 ACC reg. – 10 jegyű decimális számra
  - 4 alapművelet + gyökvonás
  - Kártyaolvasó-író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

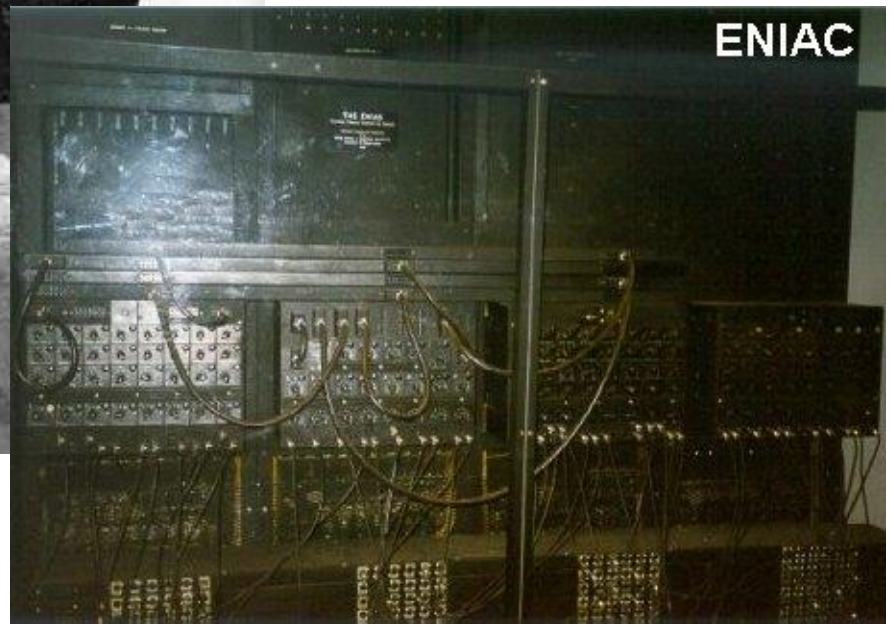
ENIAC


Card reader
Printer and Card Punch
Data lines
Multiplier
Function Tables
Divider and square root Logic
A1
A2
A20
Control lines
Master Control Unit

ENIAC



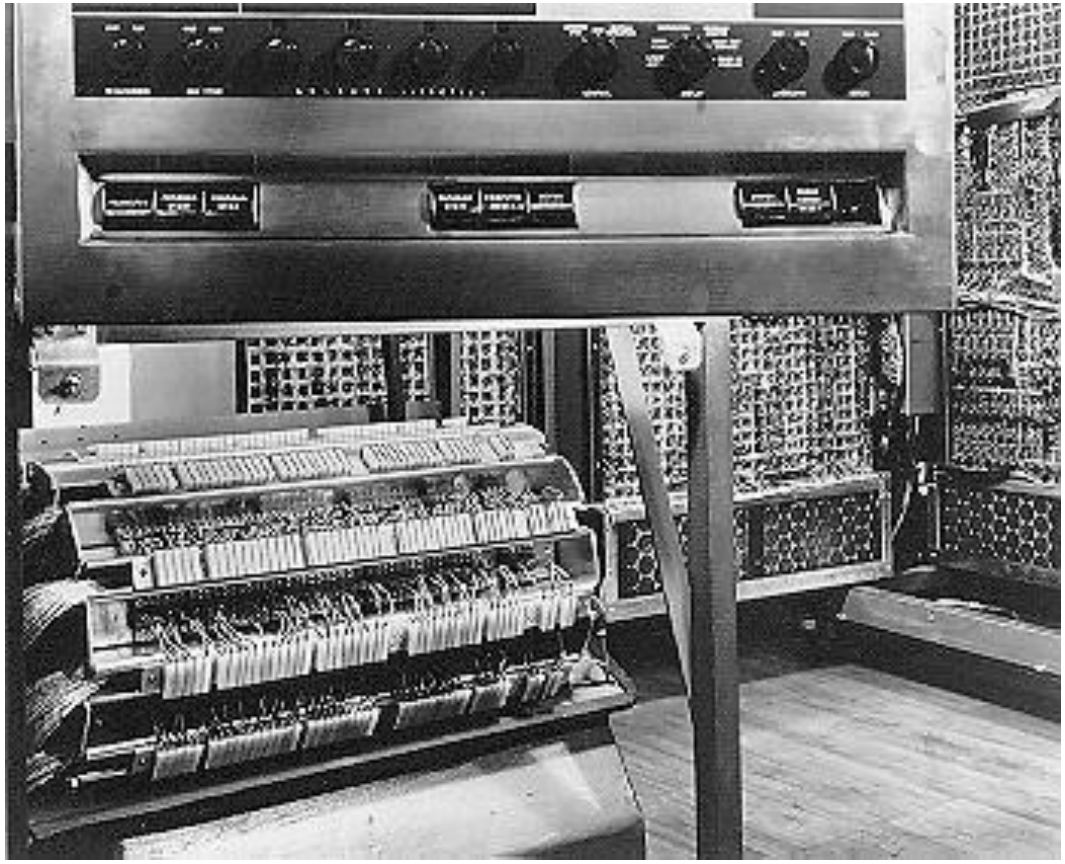




I. Generáció (folyt.):

- 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]

EDVAC



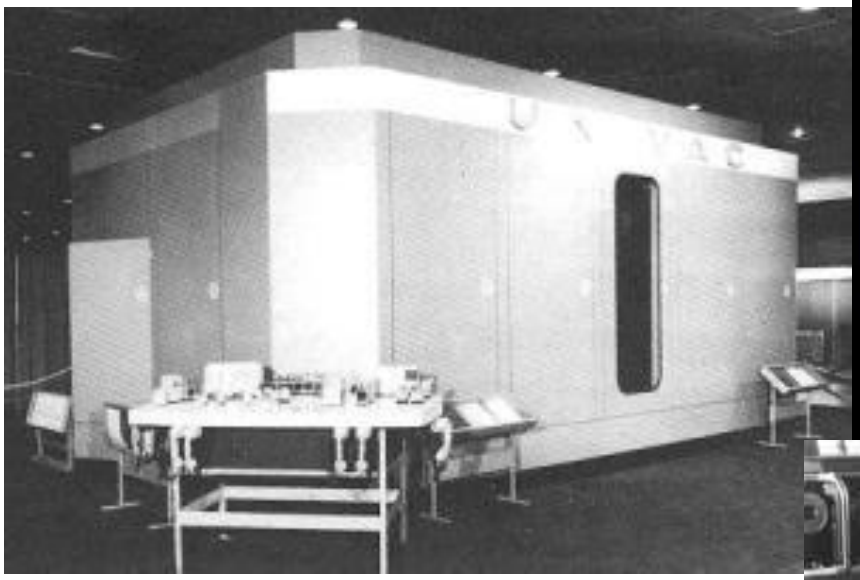


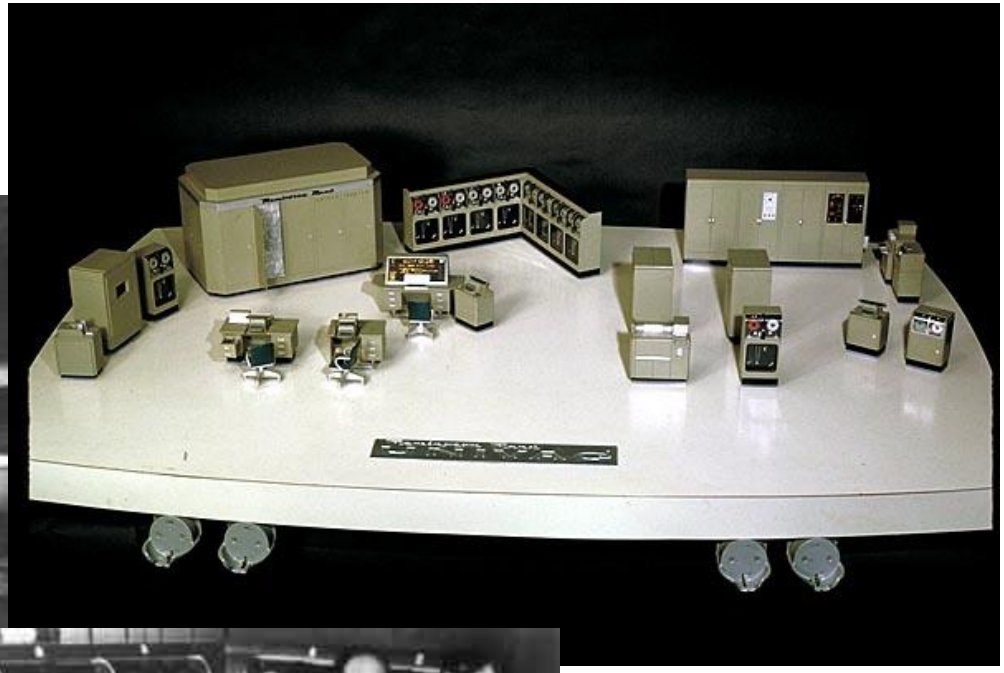
Neumann János

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



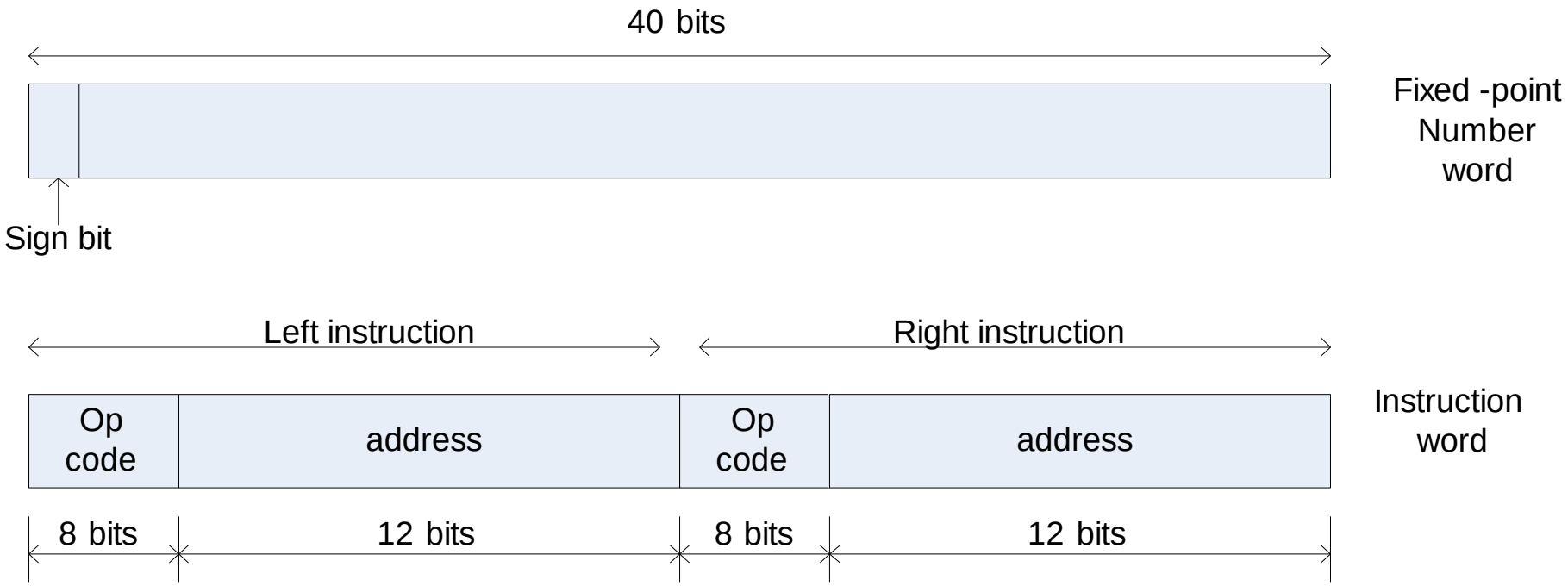




I. Generáció (folyt.):

- 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

Instruction	*Description*
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.

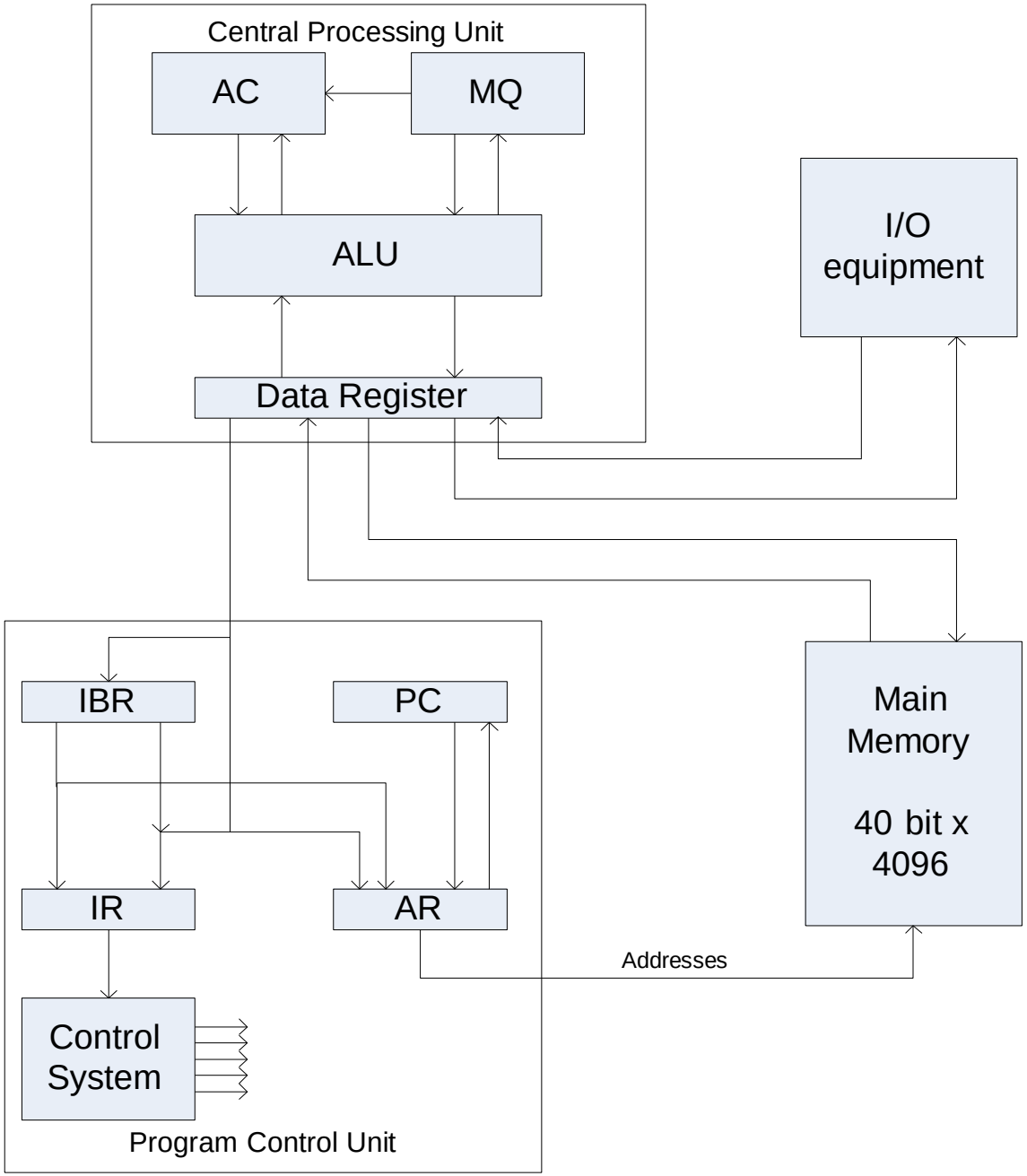
Arithmetic instructions

Instruction	*Description*
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.

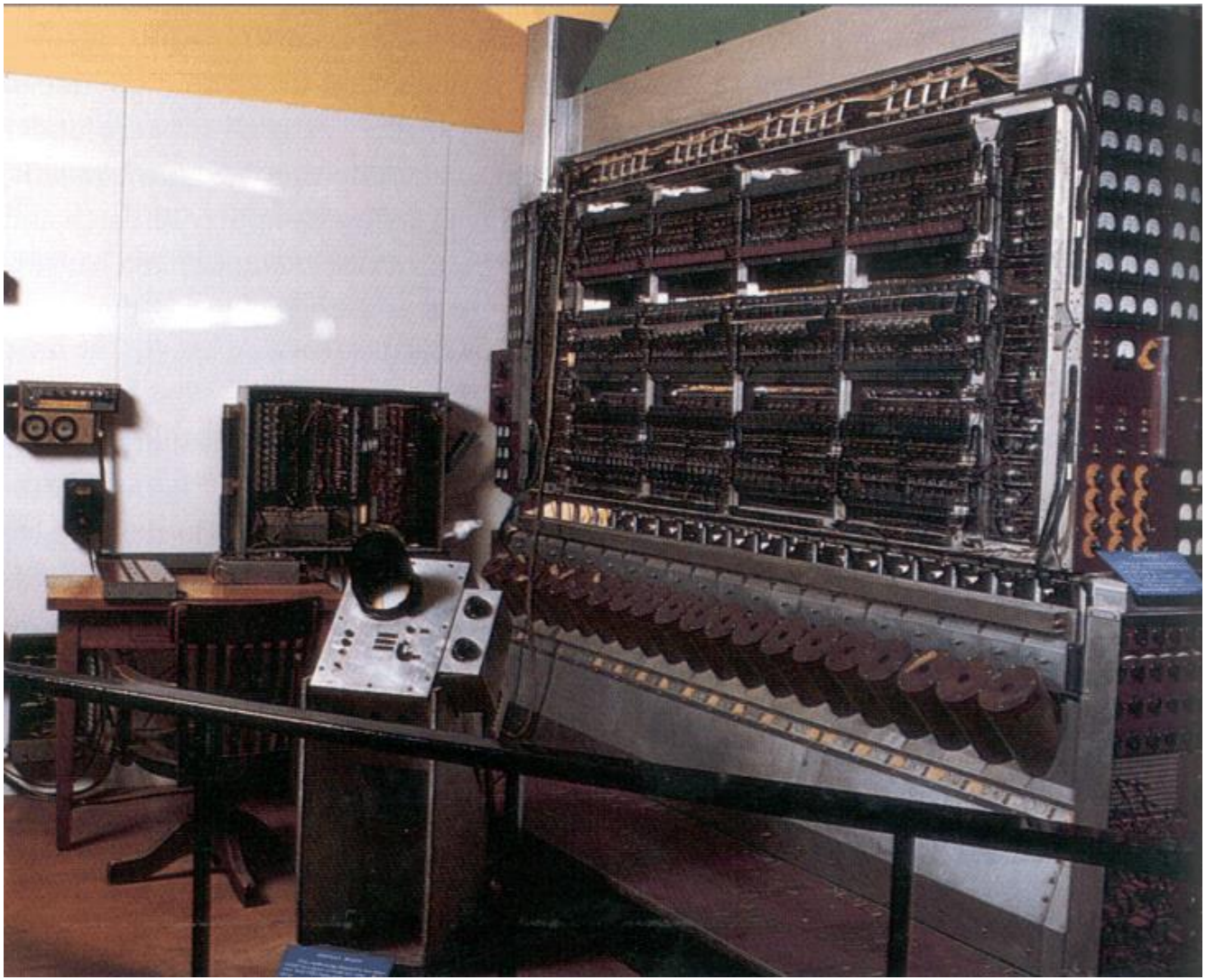
1.1 Táblázat: IAS utasítások (folyt.)

Jump instructions	
Instruction	*Description*
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.
Conditional branch instructions	
Instruction	*Description*
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.
Address modification instructions	
Instruction	*Description*
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


Central Processing Unit
AC
MQ
ALU
Data Register
I/O equipment
IBR
PC
IR
AR
Control System
Program Control Unit
Main Memory
40 bit x 4096
Addresses

IAS computer



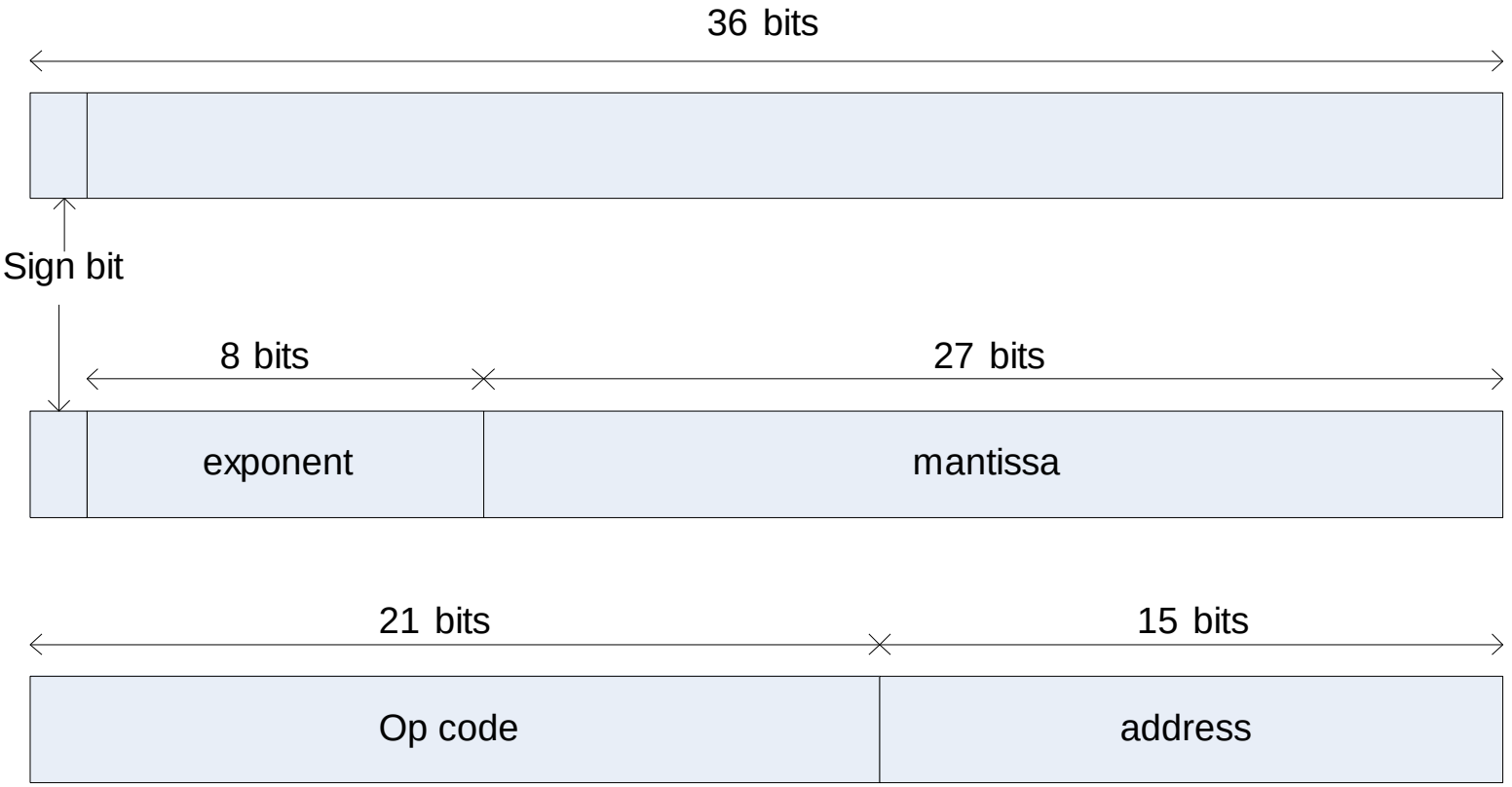
II. Generáció (1952-63):

- 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

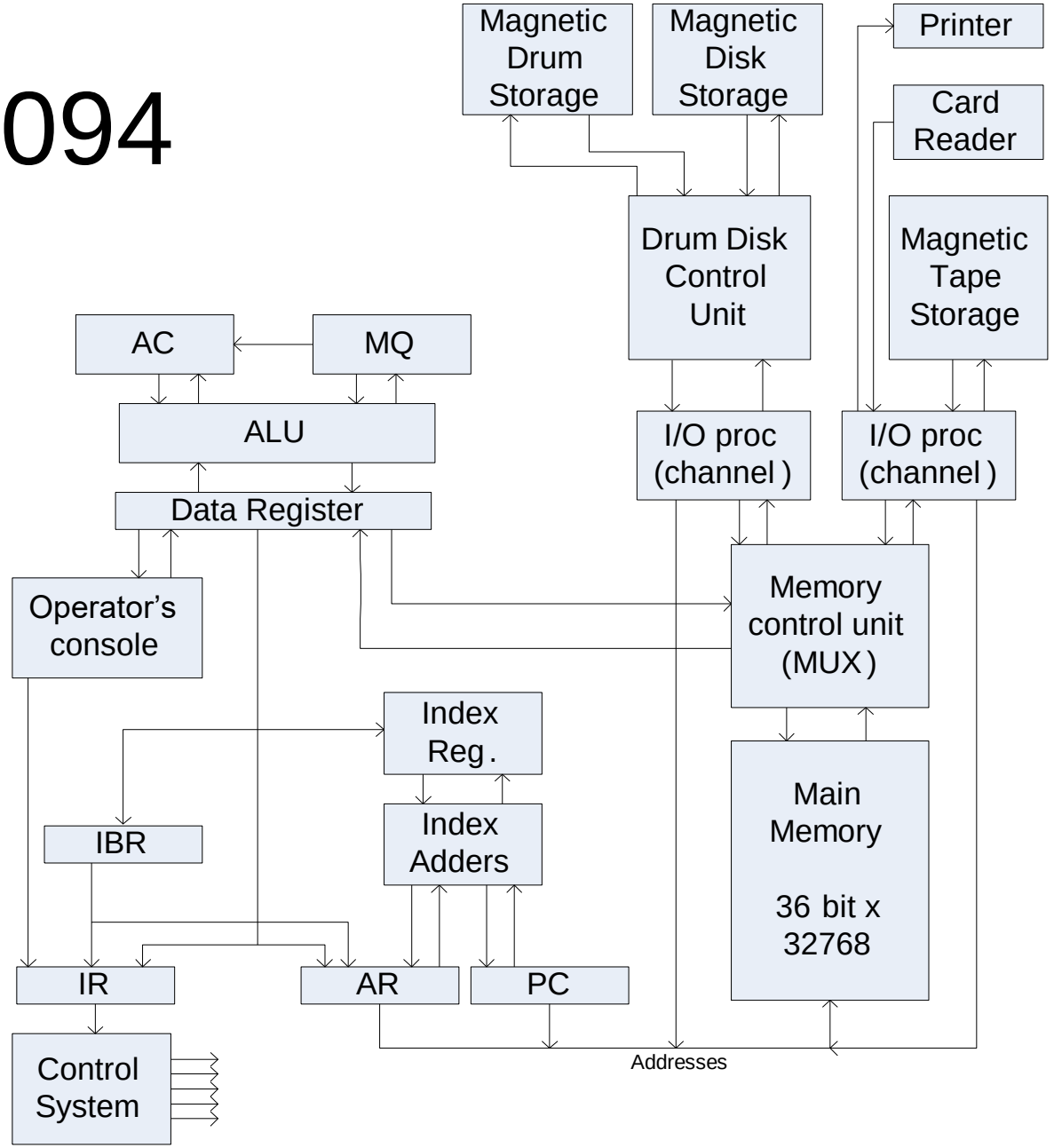
II. Generáció (folyt.):

- IBM 709x
  - 36 bites utasítás, műveleti kód (1.1 tábl.)
  - egycímű gép (AR$\Leftarrow$ PC+IR tartalma)
  - 72 bites adatút
  - I/O processzorok

IBM 709x adat- és utasítás formátum:



IBM 7094


Magnetic Drum Storage
Magnetic Disk Storage
Printer
Card Reader
Drum Disk Control Unit
Magnetic Tape Storage
AC
MQ
ALU
I/O proc (channel)
I/O proc (channel)
Data Register
Operator's console
Memory control unit (MUX)
Index Reg .
Main Memory
36 bit x 32768
IBR
Index Adders
IR
AR
PC
Control System
Addresses

IBM 7094

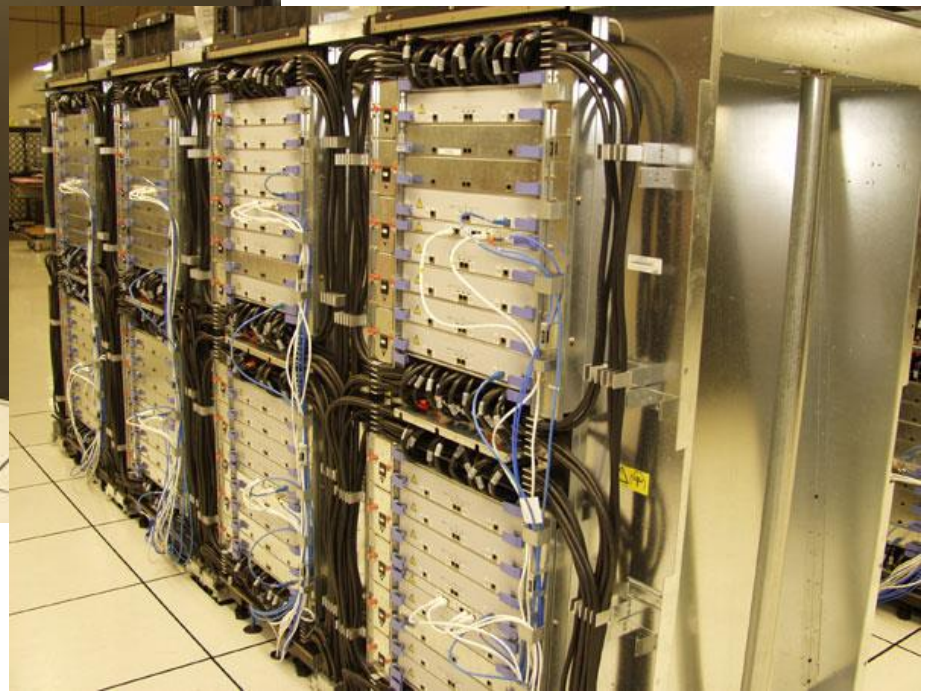


II. Generáció (folyt.):

- Szuperszámítógépek
  - LARC: (Livermore) atom-kutatásokra
  - IBM 7030 / Strech
  - **MA (2007. nov.)!:** IBM Blue Gene/L – eServer (www.top500.org) – Livermore
    - 212992 processzoros rendszer (IBM PowerPC 440s)
    - 73728 GB memória
    - 478200 GFLOPs teljesítmény!
- FDE – parallelizmus
  - átlapolt végrehajtás (látszólagos) - pipeline
  - teljesen párhuzamos végrehajtás (több processzor) – CELL BE

IBM Blue Gene/L supercomputer





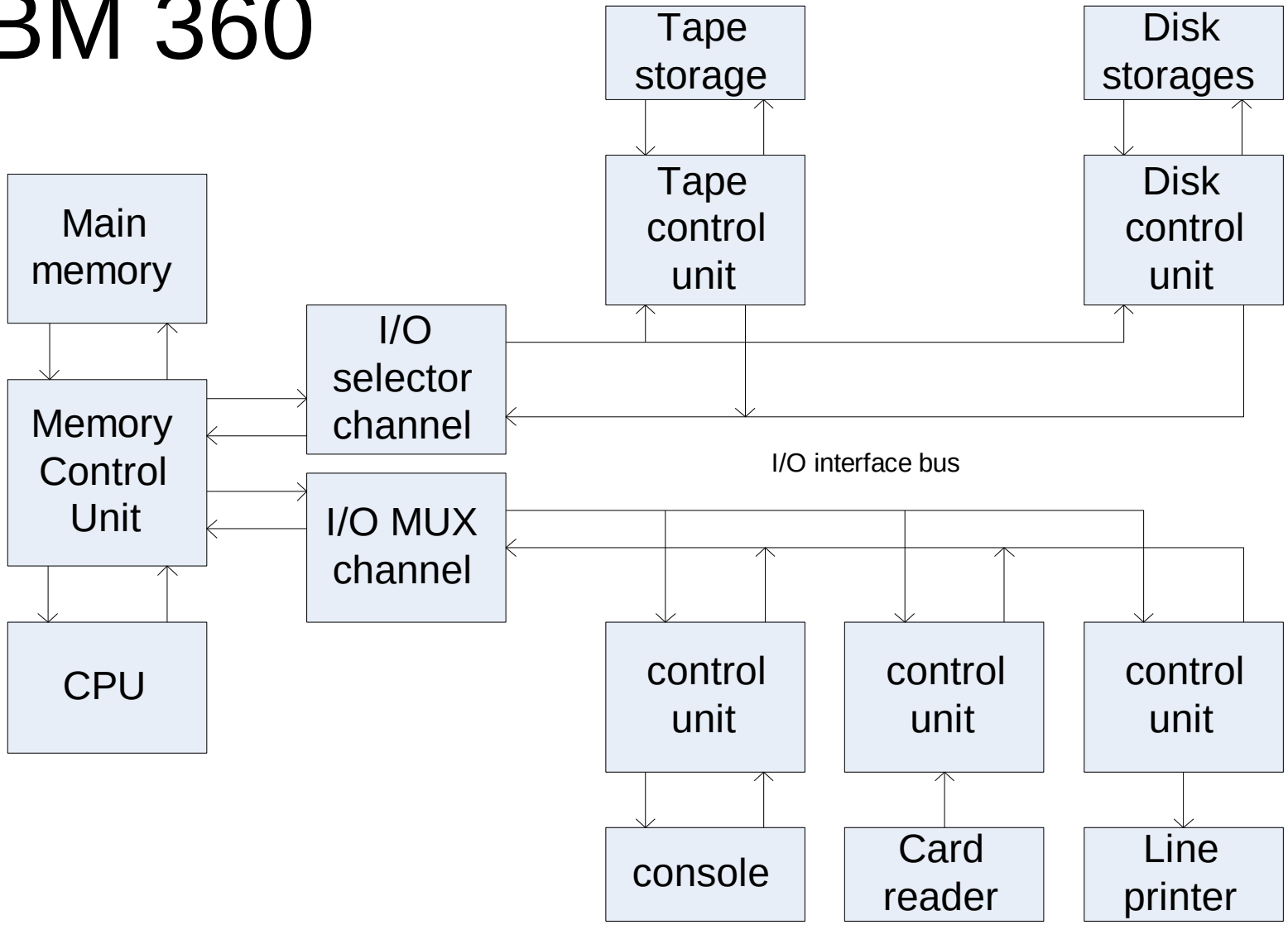
III. Generáció (1962-75):

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

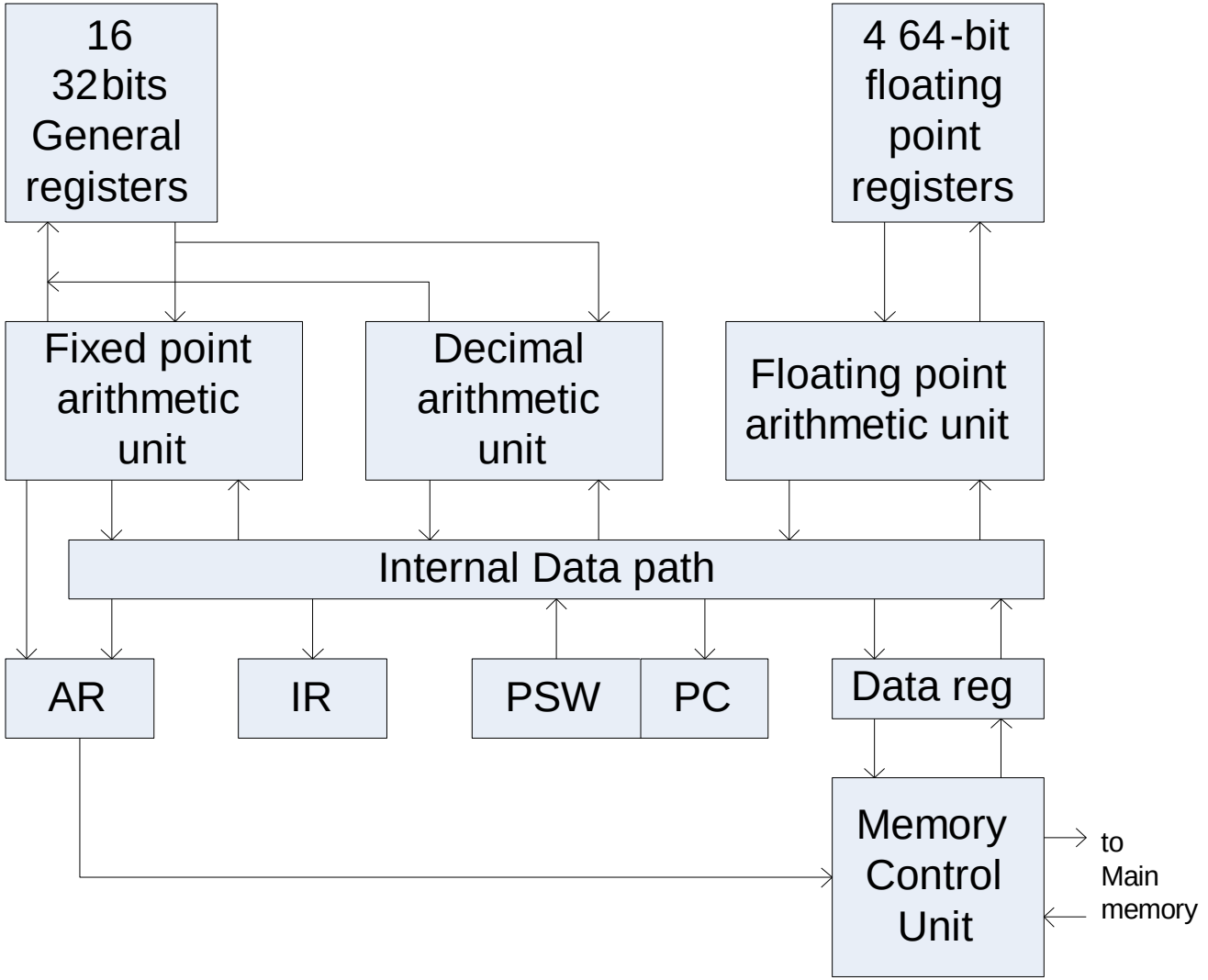
IBM 360

- 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


Tape storage
Disk storages
Tape control unit
Disk control unit
Main memory
I/O selector channel
Memory Control Unit
I/O interface bus
I/O MUX channel
CPU
control unit
control unit
control unit
console
Card reader
Line printer

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 – 32bites proc.
  - ALU, Regiszterek, virtuális memória egy chipen
- 4, majd 16 … megabites memóriák
- PC: személyi számítógépek megjelenése
- Száloptika $\Rightarrow$ hálózatok (INTERNET)

V. Generáció (napjainkban):

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

Merre tart a technológia?

1. Roadmap projections for Semiconductor technology (prediction)

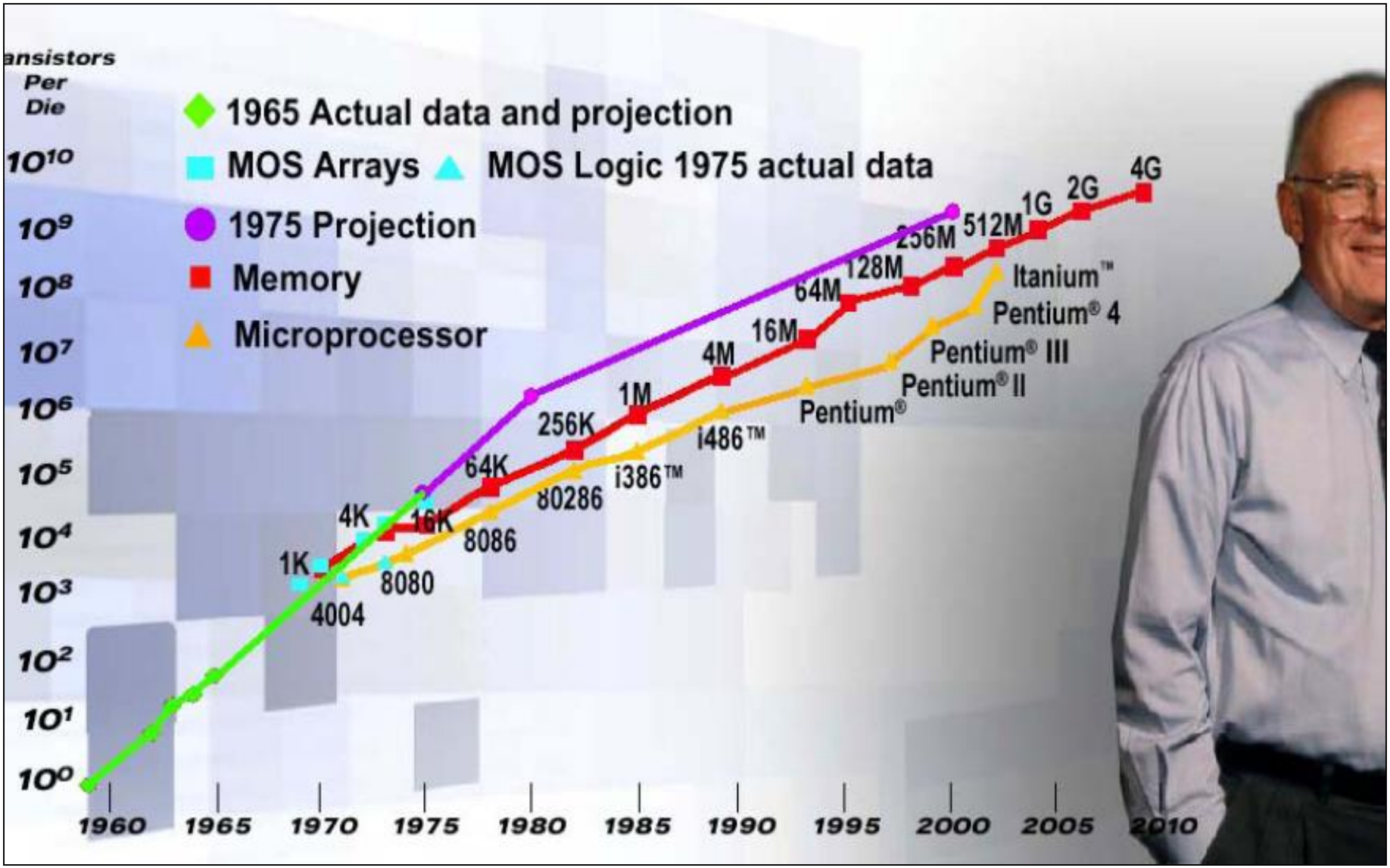
Year	Smallest feature [μm]	Dynamic Ram		Microprocessors			Wiring Levels / chip	I/O /chip
		Chip size [mm²]	Billions of bits / chip	Chip size [mm²]	Millions of transistors / cm²	On-Chip Clock (MHz)		
1995	0.35	190	0,064	250	4	300	4-5	900
1998	0.25	280	0,256	300	7	450	5	1350
2001	0.18	420	1	360	13	600	5-6	2000
2004	0.13	640	4	430	25	800	6	2600
2007	0.09	960	16	520	50	1000	6-7	3600
2010	0.07	1400	64	620	90	1100	7-8	4800

2. NOW and near future:

2004	0.09-0.13					3600	7	
end of 2005	0.065	110	70 Mbit		500	?	8	
2009*	0.03							

***EUV: extrem UV lithographical technique**

Moore törvénye



Flash memory

2006-II.	<50 nm	16 Gbit (max 32 GB)		16 milliárd!	Samsung CF NOR technology	
2008	40-20 nm	32 Gbit (max 64 GB)			Samsung NAND CF (PRAM t)	
szept.08	~20 nm	16 Gbit (max 32 GB)			Samsung SSD PRAM technology	

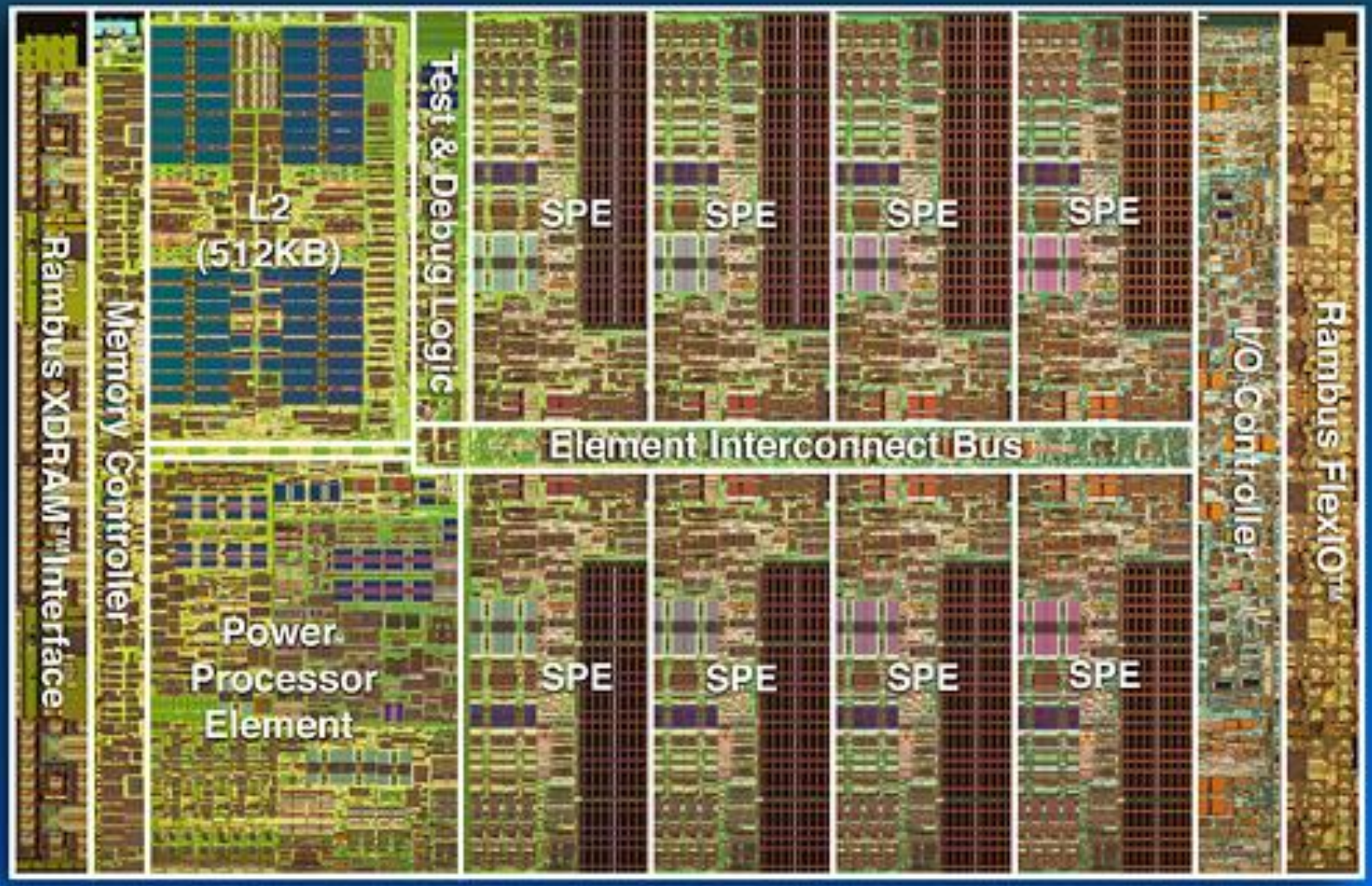
Órajelnövelés helyett Párhuzamosítás!

Többmagos tech.	**Feature**	**Size**	**Millions of trans.**		**Dissipation (TPD/ACP)**	
Intel Core2 Duo/Extreme	65nm	143 mm^2	291 millió	2.9 GHz	65-125 W	Conroe
AMD Athlon 64 X2 5200+	90nm SOI	199 mm^2	233 millió	2.6 GHz	65-80 W	Windsor
IBM Power6 (2 magos)	65nm SOI	341 mm^2	700 millió	<5 GHz	>100 W	
Intel Core2 Duo/Extreme	45nm (HKMG)	2x107mm^2	2 x 410 millió	2.6 GHz	130W	Penryn
Intel Core2 Quad (4 mag)	65nm	2x143mm^2	2 x 291 millió	2.6 GHz	130 W	Conroe
AMD Phenom Quad(4 m)	65nm SOI	285 mm^2	463 millió	2.2GHz	95 W	Agena
IBM Cell (8 magos) - PS3	90nm SOI	221 mm^2	231 millió	3.2 GHz	85 W	

Kvantumszámítógép

D-Wave	2007.febr.	sokváltozós feladatokra: biometrika, parametrikus adatbázisok , pénzügyi számítások számára

Cell Broadband Engine Processor



IBM

Logikai szorzás, ÉS kapcsolat azonosságai:

- $0 \cdot 0 = 0$
- $0 \cdot 1 = 0$
- $1 \cdot 0 = 0$
- $1 \cdot 1 = 1$
- $A \cdot 1 = A$
- $A \cdot 0 = 0$
- $AA = A$ (idempotencia)
- $A\bar{A} = 0$
- $AB = BA$ (kommutativitás)
- $(AB)C = A(BC) = ABC$ (asszociativitás)

Logikai összeadás, VAGY kapcsolat azonosságai:

- $0 + 0 = 0$
- $0 + 1 = 1$
- $1 + 0 = 1$
- $1 + 1 = 1$
- $A + 0 = A$
- $A + 1 = 1$
- $A + A = A$ (idempotencia)
- $A + \bar{A} = 1$
- $A + B = B + A$ (kommutativitás)
- $(A - B) + C = A + (B - C) = A - B + C$ (asszociativitás)

Logikai tagadás, NEM (negáció) azonosságai:

- $\bar{0} = 1$
- $\bar{1} = 0$
- $\bar{\bar{A}} = A$

További azonosságok:

- $A(B + C) = AB + AC$ (disztributivitás)
- $A + BC = (A + B)(A + C)$ (disztributivitás)
- $A + AB = A$ (elnyelési tulajdonság)
- $A(A + B) = A$ (elnyelési tulajdonság)
- $AB + A\bar{B} = A$
- $(A + B)(A + \bar{B}) = A$
- $A + \bar{A}B = A + B$
- $A(\bar{A} + B) = AB$

De-Morgan azonosságok

- $\overline{A + B} = \bar{A}\bar{B}$
- $\overline{A + B + C} = \bar{A}\bar{B}\bar{C}$
- …
- $\overline{AB} = \bar{A} + \bar{B}$
- $\overline{ABC} = \bar{A} + \bar{B} + \bar{C}$
- …

Algebrai módszer

1. Példa:

Egy logikai kifejezést több féle képen is le lehet egyszerűsíteni. Legyen $\overline{A\bar{B}+\bar{A}\bar{B}+A\bar{B}}$ az egyszerűsítendő kifejezés. Első lépésben a De-Morgan azonosság segítségével eltüntethető a teljes kifejezés negálása. Eredményül három negált minterm ÉS kapcsolata adódik: $\overline{(A\bar{B})}\,\overline{(\bar{A}\bar{B})}\,\overline{(A\bar{B})}$. Ahhoz, hogy a mintermek ne legyenek negálva, ezeken is külön-külön alkalmazni kell a De-Morgan azonosságot. Ez a függvény konjunktív normál formáját eredményezi: $(\bar{A}-\bar{B})(A+\bar{B})(\bar{A}+B)$. Következő lépésben a disztributivitás tulajdonságot kihasználva össze lehet vonni a második két zárójelet: $(\bar{A}-\bar{B})(A\bar{A}+AB+\bar{A}\bar{B}+\bar{B}B)$. Ezután kihasználható, hogy egy logikai változó ÉS kapcsolata a változó negáltjával 0-át és egy logikai változó VAGY kapcsolata 1-el a változót adja eredményül: $(\bar{A}-\bar{B})(AB+\bar{A}\bar{B})$. Ötödik lépésben a disztributivitást ismét felhasználva felbonthatóak a zárójelek: $\bar{A}AB+\bar{A}\bar{A}\bar{B}+AB\bar{B}+\bar{A}\bar{B}B$. Ezután ismét felhasználható az előző tulajdonság valamint az, hogy bármely kifejezés ÉS kapcsolata 0-val 0: $\bar{A}AB+\bar{A}\bar{B}B$. Az ÉS művelet idempotencia tulajdonsága alapján mind a két szorzatot egyszerűsíthető: $\bar{A}\bar{B}+\bar{A}\bar{B}$. Végül a VAGY művelet idempotencia tulajdonsága alapján ismét egyszerűsíteni lehet: $\bar{A}\bar{B}$. Az egyszerűsítés teljes folyamata alább látható.

$$\overline{A\bar{B}+\bar{A}\bar{B}+A\bar{B}}=$$
$$\overline{(A\bar{B})}\,\overline{(\bar{A}\bar{B})}\,\overline{(A\bar{B})}=$$
$$(\bar{A}-\bar{B})(A+\bar{B})(\bar{A}+B)=$$
$$(\bar{A}-\bar{B})(A\bar{A}+AB+\bar{A}\bar{B}+\bar{B}B)=$$
$$(\bar{A}-\bar{B})(AB+\bar{A}\bar{B})=$$
$$\bar{A}AB+\bar{A}\bar{A}\bar{B}+AB\bar{B}+\bar{A}\bar{B}B=$$
$$\bar{A}AB+\bar{A}\bar{B}B=$$
$$\bar{A}\bar{B}+\bar{A}\bar{B}=$$
$$\bar{A}\bar{B}$$

Ugyanez a logikai függvény másként is egyszerűsíthető, most más sorrendben használva a Boole algebrai azonosságokat. Első lépésben kihasználható a disztributivitás tulajdonság így az első két mintermből kiemelhető B: $\overline{(\bar{A}+\bar{A})B+A\bar{B}}$. A logikai VAGY kapcsolat idempotencia tulajdonsága alapján az $(\bar{A}+\bar{A})$ kifejezés kiesik: $\overline{B+A\bar{B}}$. Következő lépésben az $A+\bar{A}B=A+B$ azonosságot felhasználva a kifejezés átalakítható: $\overline{B+\bar{A}}$. Majd utolsó lépésben egy De-Morgan azonosság segítségével megkapható a végső formát: $\bar{A}\bar{B}$. Az egyszerűsítés teljes menete alább látható.

$$\overline{A\bar{B}+\bar{A}\bar{B}+A\bar{B}}=$$
$$\overline{(\bar{A}+\bar{A})B+A\bar{B}}=$$
$$\overline{B+A\bar{B}}=$$
$$\overline{B+\bar{A}}=$$
$$\bar{A}\bar{B}$$

Kifejtési módszer

Az algebrai módszer komplexebb, több változót tartalmazó logikai kifejezések $F(x_1,x_2,\ldots,x_n)$ egyszerűsítése esetén nehezen alkalmazható. Ilyen esetekben alkalmazható a kifejtési módszer. Ekkor egy adott változó értékét először ponáltnak $F(1,x_2,\ldots,x_n)$, majd

negáltnak $F(0, x_2, \dots, x_n)$ kell definiálni. Ezután a két egyszerűbb, kevesebb változót tartalmazó függvényt egyszerűsíthető (lehet rekurzívan kifejtési módszert is alkalmazni). Az eredeti függvény egyszerűsített formája kétféleképpen is megkapható.

- A változó ponált rögzítésével kapott függvényhez ÉS logikai kapcsolattal hozzá kell fűzni a változót ponált értékben, a negált rögzítéssel kapott függvényhez pedig negált értékben. Végül az így kiszámított két logikai kifejezést VAGY logikai művelettel kell összekötni.
  $F(x_1, x_2 \dots, x_n) = x_1 \cdot F(1, x_2 \dots, x_n) - \overline{x_1} \cdot F(0, x_2 \dots, x_n)$
- A változó negált rögzítésével kapott függvény negáltjához VAGY logikai kapcsolattal kell hozzáfűzni a változót ponált értékben, a ponált rögzítéssel kapott függvény negáltjához pedig negált értékben. Végül az így kiszámított két logikai kifejezést ÉS logikai művelettel kell összekötni és negálni.
  $F(x_1, x_2 \dots, x_n) = \overline{(x_1 + \overline{F(0, x_2 \dots, x_n)}) \cdot (\overline{x_1} + \overline{F(1, x_2 \dots, x_n)})}$

Legyen adott diszjunktív normál formában egy háromváltozós F függvény, ahol $F(A, B, C) = \bar{A}B\bar{C} + \bar{A}BC + A\bar{B}\bar{C} + AB\bar{C}$. Legyen az A változó értéke először 1, ekkor $F(1, B, C) = 0B\bar{C} + 0BC + 1\bar{B}\bar{C} + 1B\bar{C} = \bar{B}\bar{C} + B\bar{C} = (\bar{B} + B)\bar{C} = \bar{C}$. Ezek után legyen A értéke 0, azaz $F(0, B, C) = 1B\bar{C} + 1BC + 0\bar{B}\bar{C} + 0B\bar{C} = B\bar{C} + BC = B(\bar{C} + C) = B$.

- Az egyszerűsített logikai függvény meghatározása az első módszer szerint, azaz $F(A, B, C) = A \cdot F(1, B, C) - \bar{A} \cdot F(0, B, C) = A\bar{C} + \bar{A}B$.
- Az egyszerűsített logikai függvény meghatározása a második módszer szerint, azaz $F(A, B, C) = \overline{(A + \overline{F(1, B, C)}) \cdot (\bar{A} - \overline{F(0, B, C)})} = \overline{(A - \bar{B})(\bar{A} - C)} = \bar{A}B + A\bar{C}$.

Kifejtési módszer esetében fontos, hogy melyik változó kerül rögzítésre, ennek hatékony kiválasztása az algebrai módszerhez hasonlóan nagy gyakorlatot igényel.

Grafikus módszer

Hurkok felrajzolásának szabályai

A hurkok optimális felrajzolásának, vagy más néven a tömbösítésnek, vannak szabályai, amelyeket az egyszerűsítés során be kell tartani.

- 2^n (n = 0, 1, 2, …) darab minterm (maxterm) vonható be egy hurokba.
- A hurkok téglalap alakúak.
- Minden 1-est (maxtermek esetében 0-ást) tartalmaznia kell legalább egy huroknak és 0 (maxterm esetben 1) nem kerülhet hurokba.
- Egyik hurok a másikat nem tartalmazhatja teljes mértékben.
- Egy term több hurokban is szerepelhet.
- A hurkok felrajzolásánál törekedni kell a lehető legnagyobb méretű hurkok használatára.
- Közömbös (’-’) kimeneti függvényértékeket a jobb (optimálisabb) lefedésnek megfelelően kell megválasztani (NTSH).

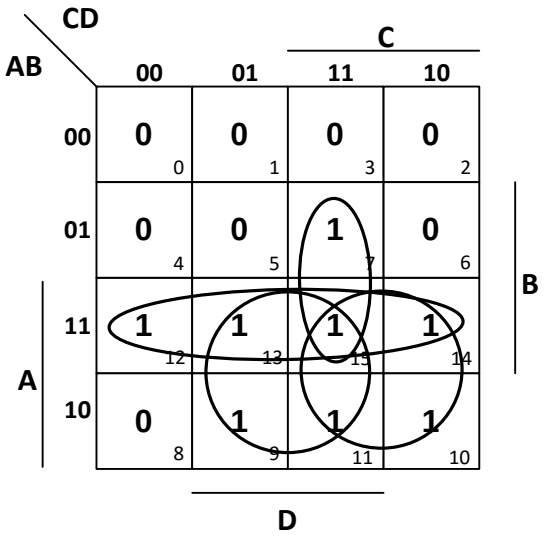
2. Példa

Készíteni kell egy logikai hálózatot, amely bemeneteihez egy súly tartozik és az F kimeneten akkor lesz 1 logikai érték, ha a bemenetek összsúlya nagyobb, mint 1 azon bementekre, ahol a logikai érték 0. Az A bemenet súlya legyen 3, a B bemeneté 2, a C és a D bemeneteké pedig 1. Az igazságtáblázatot ez alapján fel lehet írni.

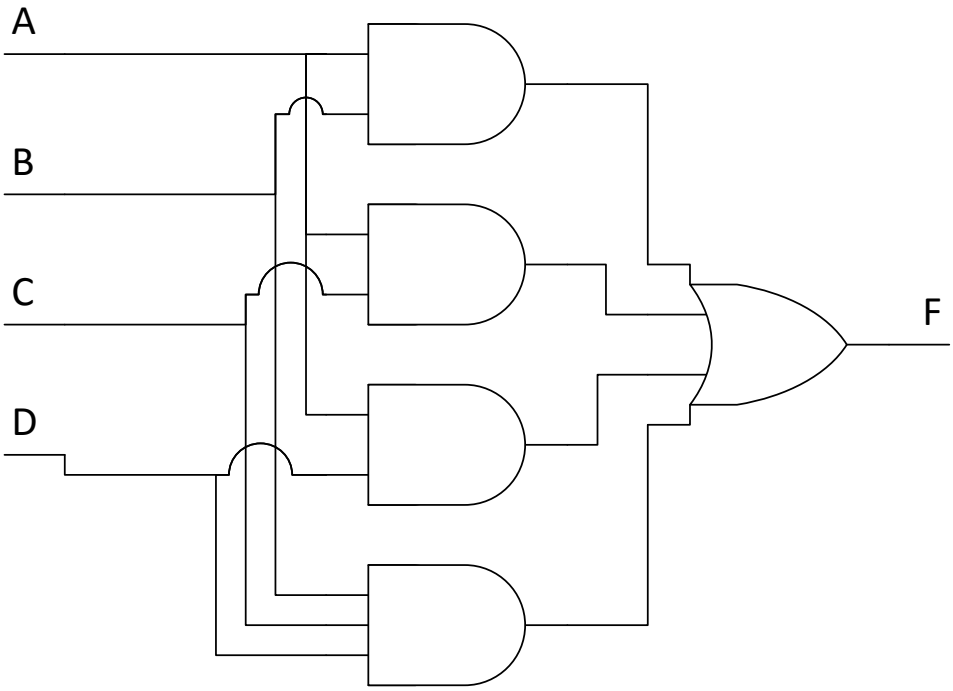
A	B	C	D	F
0	0	0	0	0

0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

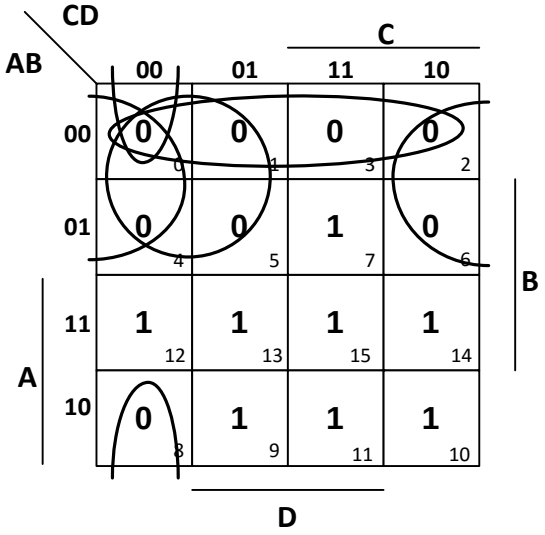
Az igazságtáblázatból fel lehet rajzolni a Karnaugh táblát. Először a diszjunktív alak meghatározása a cél. A prímimplikánsok meghatározhatók a Karnaugh tábla segítségével. Ebben az esetben az összes megtalált prímimplikáns lényeges prímimplikáns.



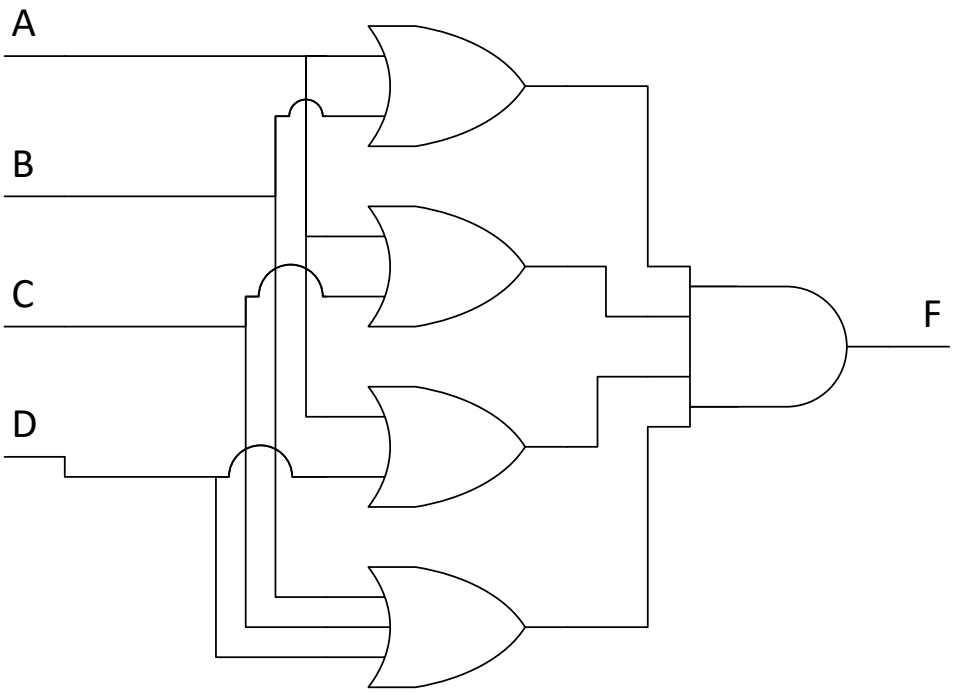
Mivel az összes lényeges prímimplikánst tartalmaznia kell az egyszerűsített függvényalaknak, ezért az egyszerűsített függvényalak a következő $F = AB + AC + AD + BCD$, melynek a megvalósítása az alábbi ábrán látható.



Konjunktív alak meghatározása esetén a Karnaugh táblában a 0-kat kell hurkolni a prímimplikánsok megtalálásához.



Az itt megtalált összes prímimplikáns is lényeges prímimplikáns, ezért nem hagyható el a megoldásból. Az eredményül kapott $F = (A + B)(A + C)(A + D)(B + C + D)$ logikai áramköri realizációja alább látható.



A megoldásként kapott diszjunktív és konjunktív alak ugyannyi kapubemenetet tartalmaz, ezért mind a kettő ugyanolyan egyszerűnek számít. Ezek után a rendelkezésre álló konkrét logikai áramkörök tulajdonságai (ár, rendelkezésre állás, méret, …) határozzák meg, hogy melyiket kell megvalósítani.

Nem teljesen specifikált hálózatok minimalizálása

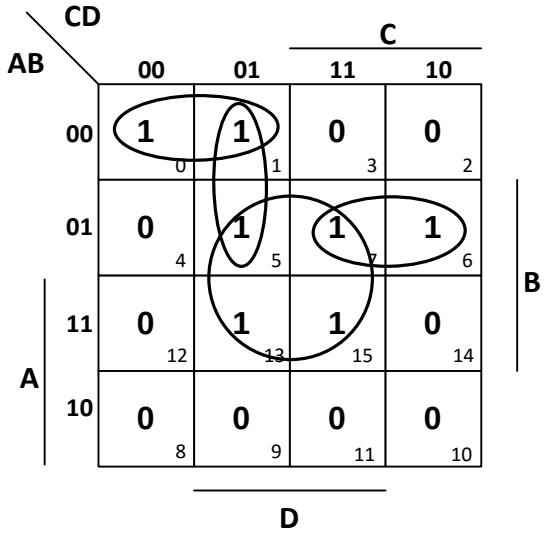
Egy példán keresztül bemutatásra kerül, hogy a határozatlan állapotok 0-nak vagy 1-nek választása mennyiben befolyásolja a megvalósítandó logikai kapcsolási vázlatot. A következő igazságtáblában nyolc darab 1-es és kettő határozatlan állapot szerepel. Karnaugh tábla segítségével meghatározhatók a prímimplikánsok.

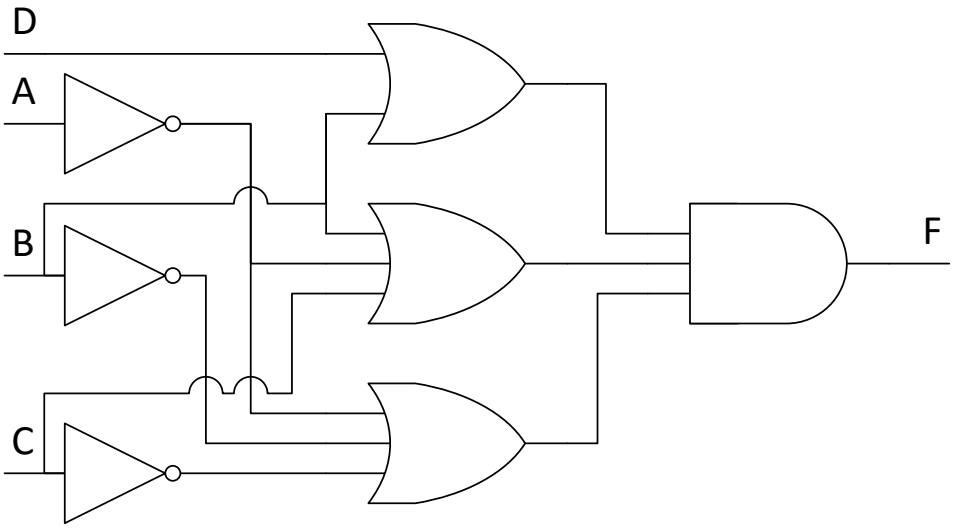
A	B	C	D	F
0	0	0	0	1
0	0	0	1	1
0	0	1	0	0
0	0	1	1	-
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	-
1	1	1	1	1

Először megvizsgálásra kerül, hogy mi történik, ha minden határozatlan állapotot 0. Ekkor az igazságtábla a következő képen módosul.

A	B	C	D	F
0	0	0	0	1
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

A Karnaugh tábla három darab kettes és egy négyes hurkot eredményez. A Karnaugh táblából a korábbiaknak megfelelően meghatározható az egyszerűsített függvényalak, mely három prímimplikánst tartalmaz és a következő függvényt eredményezi $F = \bar{B}\bar{D} + \bar{A}\bar{B}C + A\bar{B}\bar{C}$.

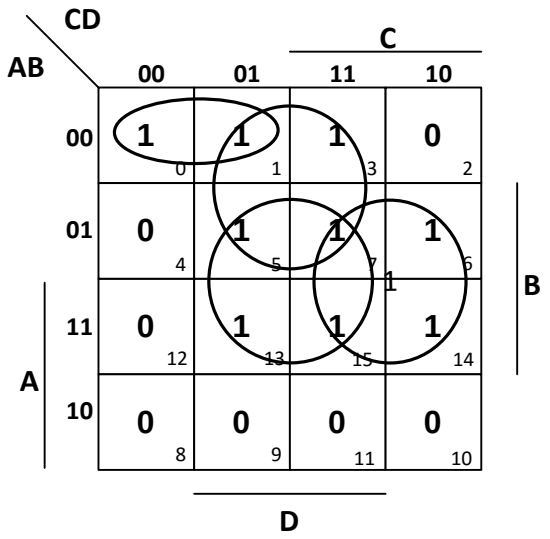


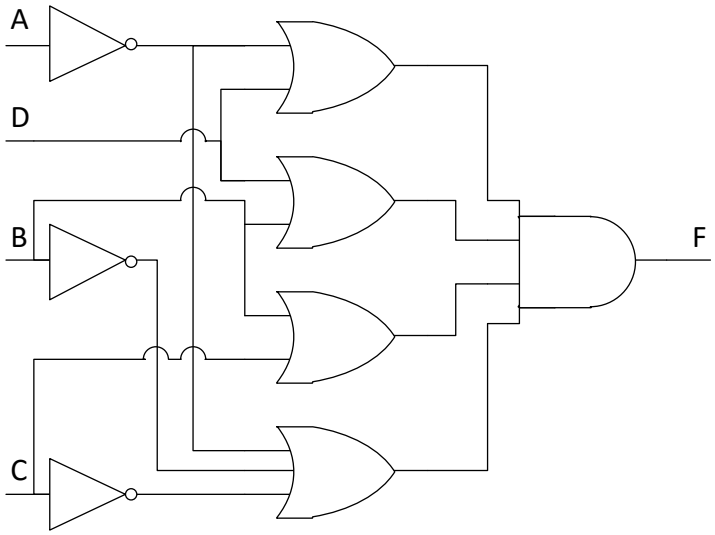


Következő esetben minden határozatlan állapot 1.

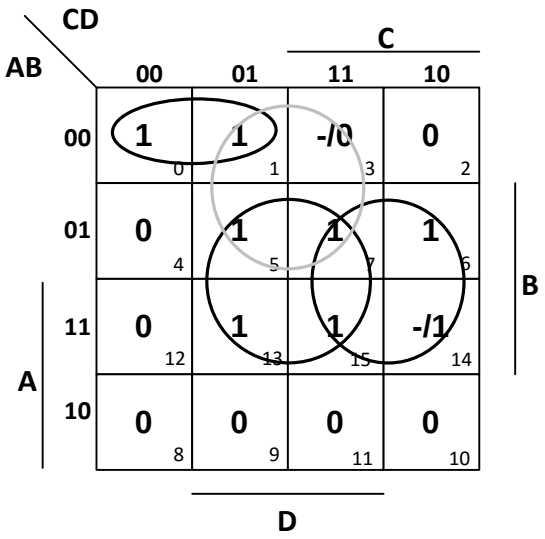
A	B	C	D	F
0	0	0	0	1
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

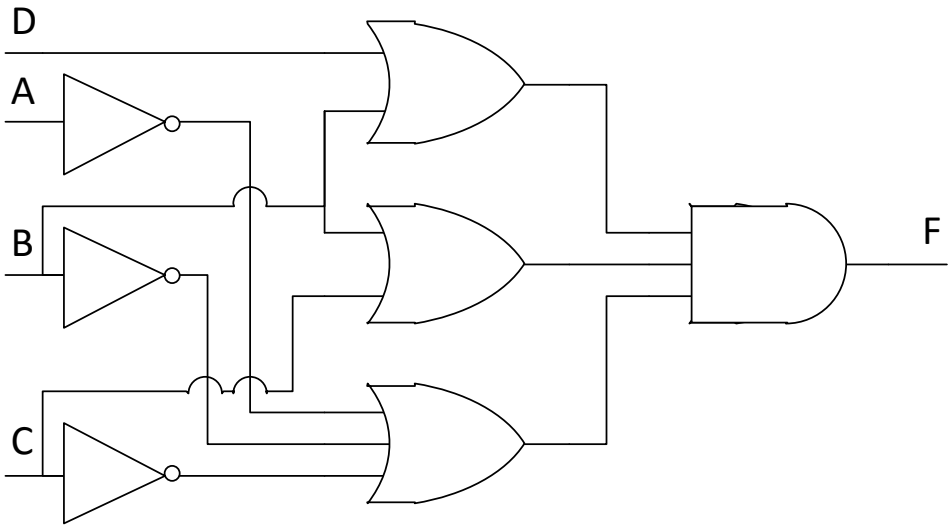
Ebben az esetben egyszerűbb prímimplikánsokat sikerül elérni, így ez egy darab kettes és három darab négyes hurkot eredményez. Az egyszerűsített függvényalak a következő $F = \bar{A}D + BD + BC + A\bar{B}\bar{C}$. A prímimplikánsok valóban egyszerűbbek, kevesebb mintermet tartalmaznak, de több prímimplikánsra van szükség a függvény megvalósításához.





Egyik előző eset sem eredményezett optimális megoldást. A fejezet elején leírt módszert alkalmazva ugyanazok a hurkok keletkeznek, abban az esetben, amikor a határozatlan állapotok 1-esek, de most a prímimplikánsok kiválasztásánál a határozatlan állapotokat ne kelljen feltétlenül lefedni. Így a következő ábrán szürkével jelölt (1, 3, 5, 7) hurkot nem kell az egyszerűsített logikai függvénynek tartalmaznia, mivel annak az 1-es, 5-ös és 7-es cellája le van fedve másik hurokkal is, a negyedik cellája viszont határozatlan állapot. Az egyszerűsített függvényalak az $F = B\bar{D} + \bar{D}C + \bar{A}\bar{B}C$ függvény, amely kevesebb prímimplikánst tartalmaz, mint ha minden határozatlan állapot 1-es és egyszerűbb, mint ha 0.





Ez a megvalósítás a következő igazságtáblázatnak felel meg, amely nem mond ellent az eredeti specifikációval.

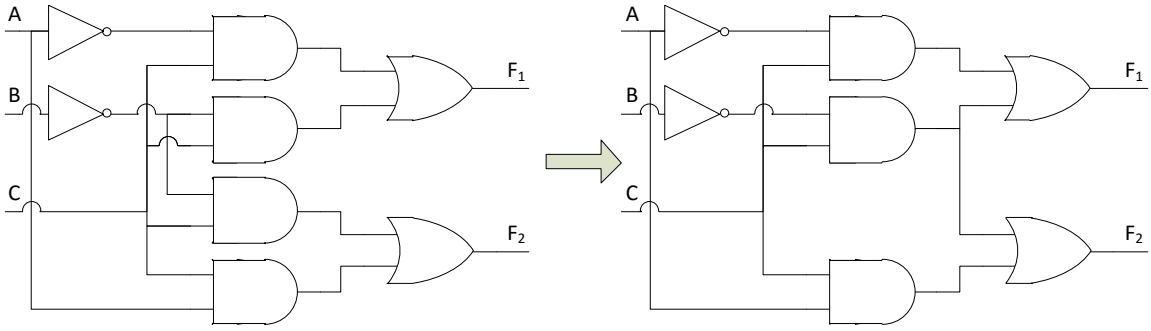
A	B	C	D	F
0	0	0	0	1
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0

1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

1. Többkimenetű logikai függvények egyszerűsítése

Többkimenetű kombinációs hálózatok esetén minden kimenethez külön logikai függvény tartozik. A logikai függvények egyszerűsítését végre lehet hajtani külön-külön egyszerűsítve őket így jutva el a legegyszerűbb diszjunktív vagy konjunktív alakhoz. Mivel mindegyik függvényhez ugyanazok a bemeneti változók tartoznak, ezért további egyszerűsítésre is lehetőség nyílhat, a közös részeket nem kell többször megvalósítani.

Legyen egy három bemenetű két kimenetű logikai hálózat, ahol a kimeneti függvények legegyszerűbb diszjunktív alakja a következő $F_1 = \bar{A}C + \bar{B}C$ és $F_2 = AC + \bar{B}C$. A két függvényt külön-külön is meg lehet valósítani, de észrevehető, hogy mind a kettő tartalmazza a $\bar{B}C$ prímimplikánst, azaz ezt a logikai kapcsolatot elegendő egyszer megvalósítani. Mind a két megvalósítás logikai áramköri realizációja az alábbi ábrán látható. A második megvalósítás természetesen egyszerűbb, azt kell megvalósítani.

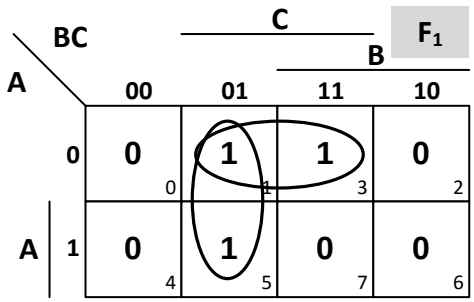


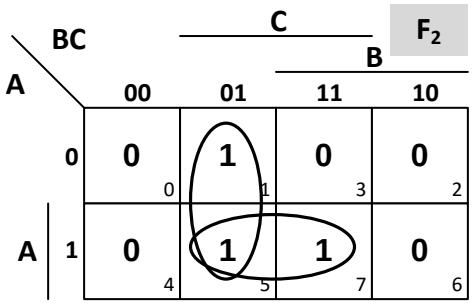
A cél a továbbiakban a közös prímimplikánsok megtalálása, hogy azokat felhasználva kevesebb áramköri elemmel legyen megvalósítható a hálózat.

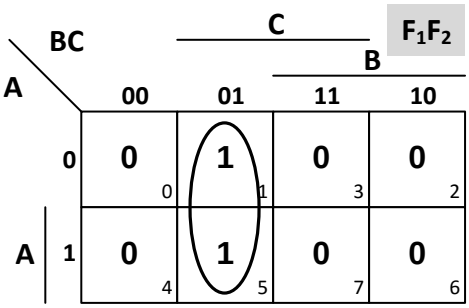
Grafikus minimalizálás

A minimalizálási eljárást módosítani kell úgy, hogy a közös prímimplikánsokat csak egyszer kelljen megvalósítani. Az egyik legegyszerűbb megoldás a szorzatfüggvény vizsgálata. A szorzatfüggvény ott vesz fel 1 értéket, ahol mind a két függvény 1 értéket vesz fel, azaz a szorzatfüggvény Karnaugh táblájában ott lesz 1, ahol mind a két Karnaugh táblában 1 volt.

Az előző példát nézve a szorzatfüggvény Karnaugh táblájában egyetlen hurok szerepel, mégpedig a közös $\bar{B}C$ hurok.

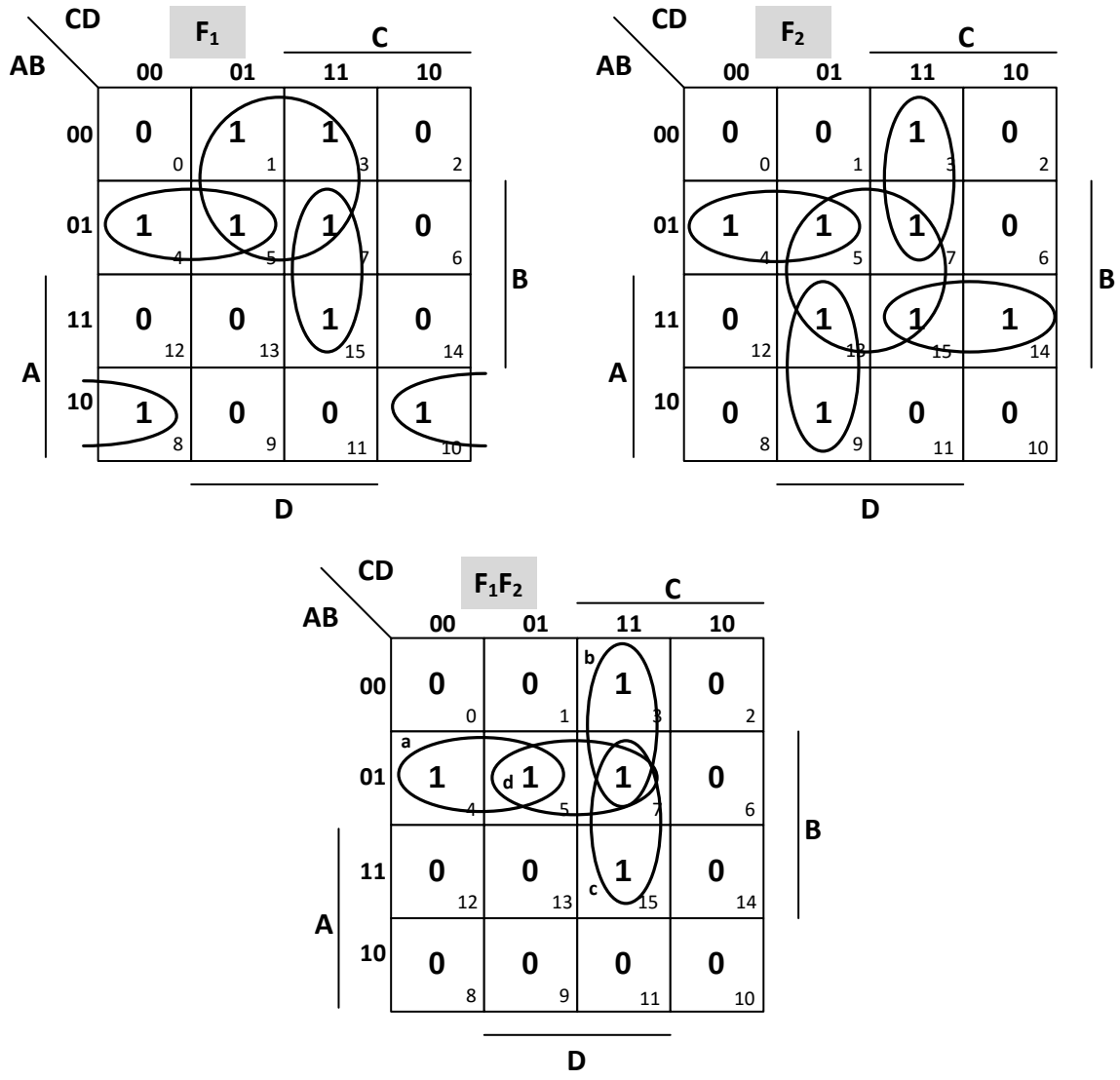






A szorzatfüggvénynek természetesen lehet olyan prímimplikánsa is, melyek nem prímimplikánsa valamely vagy mindegyik eredeti függvényeknek. Az ilyen prímimplikánsokat nem lehet figyelmen kívül hagyni a szorzatfüggvény optimális lefedéséhez.

A következő példa legyen egy négy bemenetű két kimenetű logikai hálózat, melynek Karnaugh táblái az alábbiakban adottak. Továbbá adott a szorzatfüggvény Karnaugh táblája is, ahol az implikánsok betűkkel vannak megjelölve a későbbi könnyebb hivatkozás érdekében.



Megvizsgálva a Karnaugh táblákat megállapítható, hogy a szorzatfüggvény prímimplikánsai között vannak közös és nem közös prímimplikánsok. Az **a** prímimplikáns közös, mivel F_1-ben és F_2-ben is szerepel. A **b** prímimplikáns nem közös, mivel F_2-ben ugyan prímimplikáns, de F_1-ben nem prímimplikáns, hanem egy másik prímimplikánsnak csak egy része, azaz tovább egyszerűsíthető implikánsként szerepel. A **c** prímimplikáns szintén nem közös, mivel F_1-ben ugyan prímimplikáns, de F_2-ben nem prímimplikáns, hanem egy másik prímimplikánsnak csak egy része, azaz tovább egyszerűsíthető implikánsként szerepel. A **d** prímimplikáns nem közös, mivel sem F_1 -ben sem F_2 -ben nem prímimplikáns, hanem egy másik prímimplikánsnak csak egy része, azaz tovább egyszerűsíthető implikánsként szerepel.

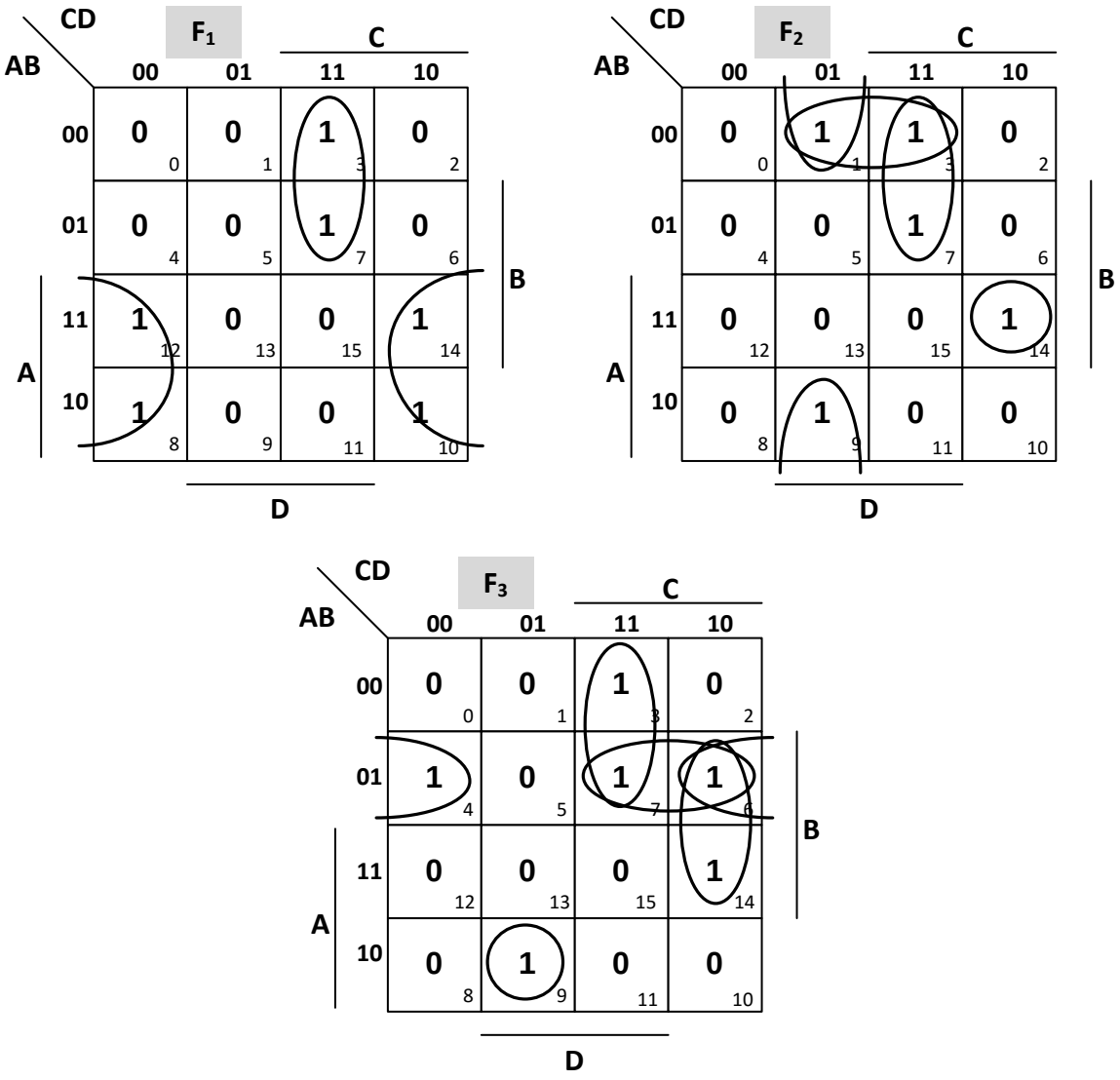
Az egyszerűsítés során a hálózat kimeneti függvényeinek összes lehetséges kombinációjára képezni kell a szorzatfüggvényt, majd ezek után minden függvényhez és szorzatfüggvényhez meg kell határozni a prímimplikánsokat. A minimalizált függvények felírásához az összes generált prímimplikánst figyelembe kell venni a függvények és a szorzatfüggvények prímimplikánsait is.

Mivel a Karnaugh táblákat nehéz lehet áttekinteni, ezért a prímimplikáns tábla valamint segédfüggvény segítségével kell kiválasztani a szükséges prímimplikánsokat. A prímimplikáns tábla tartalmazza az összes megtalált prímimplikánst a függvények és függvényszorzatok szerint csoportosítva. A táblázat kitöltését a szorzatfüggvények prímimplikánsaival kell

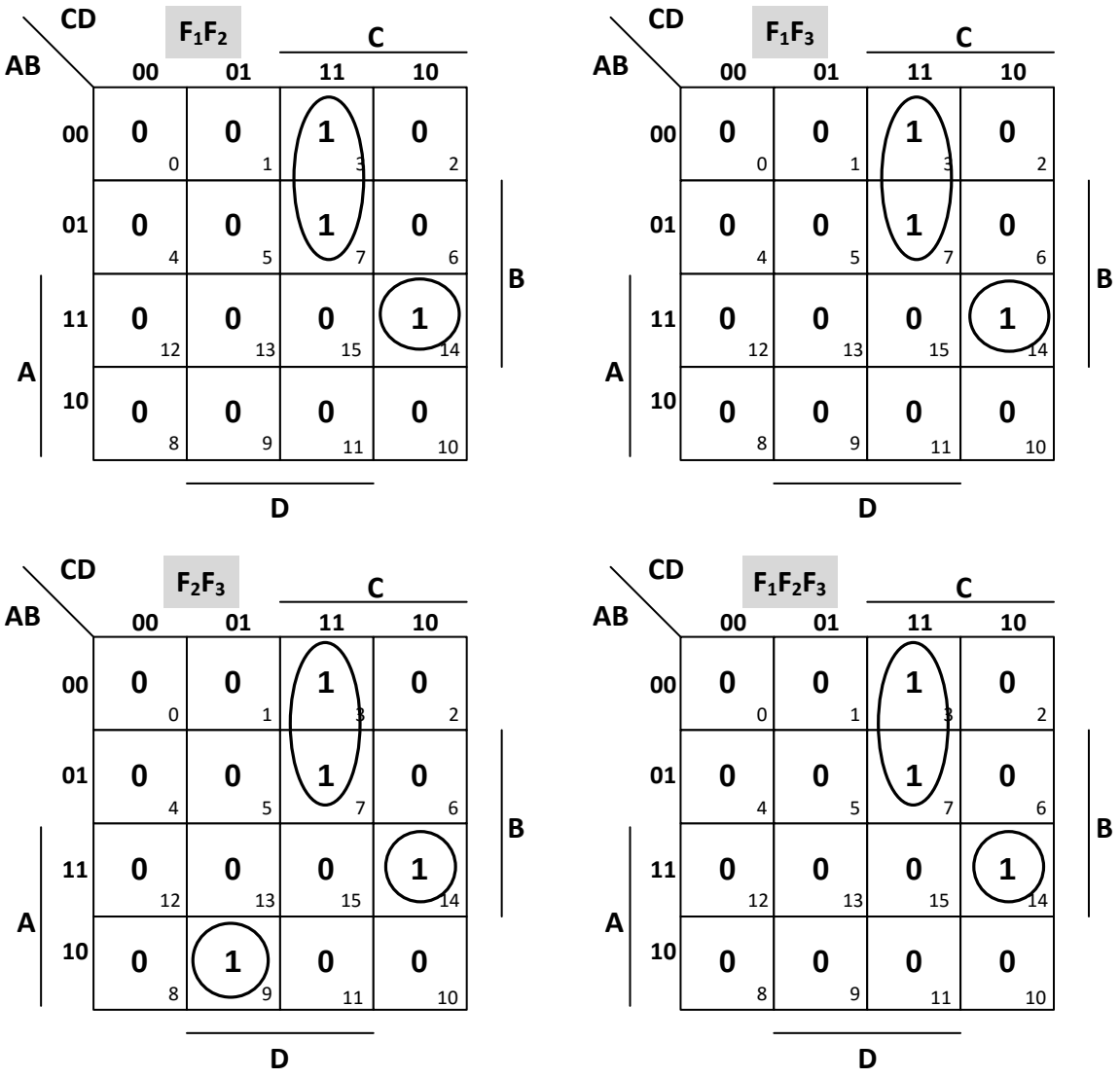
kezdeni. Ha egy szorzatfüggvénynek vagy egy függvénynek már van olyan prímimplikánsa, amely szerepel a táblázatban, akkor azt nem kell még egyszer felvenni. Ha már minden prímimplikánsa szerepel, akkor az a függvény vagy szorzatfüggvény nem jelenik meg külön a táblázatban. Így minden prímimplikáns csak egyszer fog szerepelni a táblázatban.

Amennyiben a szorzatfüggvény alapján több olyan megoldás létezik, melyekben ugyanannyi, minimális számú prímimplikáns szerepel, akkor ezek közül próbálgatással lehet kiválasztani az optimálist. A próbálgatás során csak a legkevesebb tényezőt tartalmazó szorzatokat kell figyelembe venni. A kiválasztott szorzatban szereplő prímimplikánsok az elvi logikai rajzban ÉS kapukkal lesznek megvalósítva, majd ezeknek az ÉS kapuknak a kimeneteit egy VAGY kapuk bemeneteire kell vezetni. Mivel azonos számú prímimplikáns szerepel mindegyik szorzatban, ezért a VAGY szinten ugyanannyi kapubemenet van, tehát az ÉS szinten kell minimalizálni a kapubemenetek számát.

A következő példában a feladat egy négy bemenetű és három kimenetű kombinációs hálózat minimális realizációjának meghatározása diszjunktív formában. Legyen a három kimeneti függvény diszjunktív normál formában adott a következőképpen. $F_1^4 = \sum_{i=0}^{2^4-1}(3,7,8,10,12,14)$, $F_2^4 = \sum_{i=0}^{2^4-1}(1,3,7,9,14)$ és $F_3^4 = \sum_{i=0}^{2^4-1}(3,4,6,7,9,14)$. Első lépésben meg kell határozni a kimeneti függvények prímimplikánsait Karnaugh táblák segítségével, mely alapján $F_1 = A\bar{D} + \bar{A}CD$, $F_2 = \bar{A}\bar{B}D + \bar{A}CD + \bar{B}CD + ABC\bar{D}$ és $F_3 = \bar{A}CD + \bar{A}BC + \bar{A}B\bar{D} + BC\bar{D} + A\bar{B}CD$.



Második lépésben az összes lehetséges szorzatfüggvény prímimplikánsainak meghatározása következik, mely alapján $F_1F_2 = \bar{A}CD + ABC\bar{D}$, $F_1F_3 = \bar{A}CD + ABC\bar{D}$, $F_2F_3 = \bar{A}CD + ABC\bar{D} + A\bar{B}\bar{C}D$ és $F_1F_2F_3 = \bar{A}CD + ABC\bar{D}$.



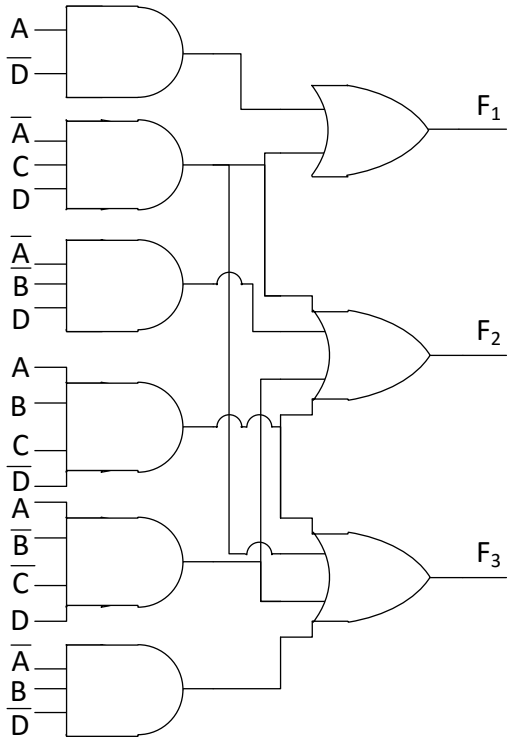
A generált prímimplikánsok alapján felírható a prímimplikáns tábla. Például az $\bar{A}C\bar{D}$ prímimplikáns szerepel az $F_1F_2F_3$ szorzatfüggvényben így bekerül a táblázatban, de hiába szerepel ugyanez a prímimplikáns az összes többi függvényben és szorzatfüggvényben, nem kerül be többször a táblázatba. Valamint például az $F_1\bar{F}_2$ szorzatfüggvény nem szerepel a táblázatban, mivel az összes prímimplikánsa már korábban bekerült.

				F_1						F_2					F_3					
	Sor	**Prímimplikáns**		**3**	**7**	**8**	**10**	**12**	**14**	**1**	**3**	**7**	**9**	**14**	**3**	**4**	**6**	**7**	**9**	**14**
$F_1F_2F_3$	*	$\bar{A}C\bar{D}$	*a*	x	x						x	x			x			x		
	*	$A\bar{B}C\bar{D}$	*b*						x					x						x
F_2F_3	*	$A\bar{B}\bar{C}D$	*c*										x						x	
F_1	*	$A\bar{D}$	*d*			x	x	x	x											
F_2		$\bar{A}\bar{B}\bar{D}$	*e*							x	x									
		$\bar{B}\bar{C}\bar{D}$	*f*							x			x							
F_3		$\bar{A}\bar{B}C$	*g*														x	x		

*	$\bar{A}B\bar{D}$	h			x	x	
	$BC\bar{D}$	i				x	x

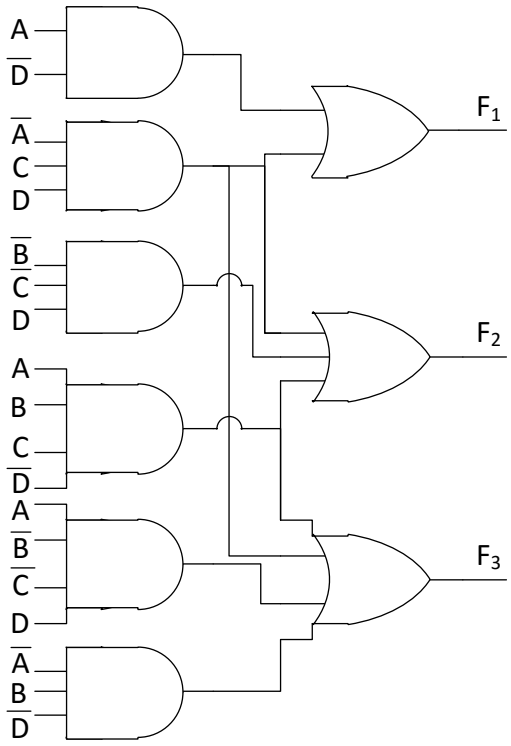
A táblázatból leolvashatóak a lényeges prímimplikánsok, de ezek nem fedik le az összes mintermet, ezért mindenképpen segédfüggvényre van szükség. Mind a három függvényhez fel kell írni a segédfüggvényt, ahol $S_1 = aaddd(b + d)$, $S_2 = (e + f)(a + e)a(c - f)b$ és $S_3 = ah(g - h + i)(a + g)c(b + i)$. Az eredő segédfüggvény a segédfüggvények ÉS kapcsolata, azaz $S = S_1S_2S_3 = aaddd(b + d)(e + f)(a + e)a(c - f)bah(g + h + i)(a + g)c(b + i)$. Az elnyelési tulajdonság segítségével a függvény egyszerűsíthető $S = abcdh(e + f) = abcdeh + abcdfh$. A segédfüggvény két lehetséges megvalósítást eredményez, ahol mind a kettőben ugyanannyi prímimplikáns szerepel. A legegyszerűbb hálózat eléréséhez próbálgatásra van szükség, azaz mind a két lehetséges megoldást elemezni kell.

Az $abcdeh$ megvalósítása esetén mind a három függvény megvalósításához csak az adott prímimplikánsokat szabad felhasználni. Mind a három függvényhez fel kell írni egy segédfüggvény és megvizsgálni, hogy melyik függvényhez melyik prímimplikánsok kellenek, de a nem felhasználható prímimplikánsok helyett 0 értékkel kell számolni. Így $S_1 = aaddd(b + d) = ad$, $S_2 = (e + 0)(a - e)a(c + 0)b = e(a + e)acb = abce$ és $S_3 = ah(0 + h + 0)(a + 0)c(b + 0) = ahhacb = abch$. Ebből meghatározhatók a kimenetek függvényei, $F_1 = \bar{A}CD + A\bar{D}$, $F_2 = \bar{A}CD + ABC\bar{D} - A\bar{B}\bar{C}D + \bar{A}\bar{B}D$ és $F_3 = \bar{A}CD + ABC\bar{D} - A\bar{B}\bar{C}D + \bar{A}\bar{B}\bar{D}$, melynek logikai áramköri realizációja lejjebb látható és a bementi negálókat nem számolva összesen 29 kapubemenetet tartalmaz.



Az $abcdfh$ megvalósítása esetén mind a három függvény megvalósításához csak az adott prímimplikánsokat szabad felhasználni. Mind a három függvényhez fel kell írni egy segédfüggvény és megvizsgálni, hogy melyik függvényhez melyik prímimplikánsok kellenek, de a nem felhasználható prímimplikánsok helyett 0 értékkel kell számolni. Így

$S_1 = aadddd(b + d') = ad$, $S_2 = (0 + f)(a + 0)a(c + f)b = faa(c - f)b = abf$ és $S_3 = ah(0 + h + 0)(a + 0)c(b + 0) = ahhacb = abch$. Ebből meghatározhatók a kimenetek függvényei, $F_1 = \bar{A}CD + A\bar{D}$, $F_2 = \bar{A}CD + ADC\bar{D} - \bar{B}\bar{C}D$ és $F_3 = \bar{A}CD + ADC\bar{D} + A\bar{B}\bar{C}D + \bar{A}B\bar{D}$, melynek logikai áramköri realizációja lejjebb látható és a bementi negálókat nem számolva összesen 28 kapubemenetet tartalmaz.



A két kapcsolás csak az F_2 függvény megvalósításában különbözik. A második megvalósításban egyel kevesebb prímimplikáns kell a függvény megvalósításához. Az első megvalósításhoz képest egy három bemenetű ÉS kapu helyett egy másik három bemenetű kell, ez az ÉS kapuk szintjén nem jelent egyszerűsítést, de a VAGY kapuk szintjén az F_2 megvalósításához négy bemenetű VAGY kapu helyett elég egy három bemenetű is, azaz egy kapubemenetet lehet megtakarítani.

Számjegyes minimalizálás

A számjegyes minimalizálás vagy más néven Quine-McCluskey módszer ugyanazt a minimalizálási alapötletet használja fel, mint a Karnaugh tábla, viszont ebben az esetben nincs korlát a változók számára. A számjegyes minimalizálás a mintermeket egyértelműen azonosító alsó indexet használja fel a számoláshoz. Ez az index egy decimális szám, amely alapján meg kell tudni mondani, hogy két minterm mikor szomszédos egymással.

Két minterm akkor szomszédos, ha csak egy változóban különböznek. Azaz ha bináris számként tekintünk rájuk, akkor csak egy helyiértékben különböznek, tehát egyetlen pozícióban van 1-es helyett 0. Három feltételnek kell teljesülnie a szomszédsághoz. A feltételekben példaként használt két bináris szám különbözzön csak az *i*-edik bitben.

Az első feltétel azt írja elő, hogy a decimális indexek különbsége kettő hatványa kell, hogy legyen. Ha a két bináris számot kivonunk egymásból (a nagyobból a kisebbet), akkor eredményül olyan bináris számot kapunk, amely mindenhol 0, kivéve az *i*-edik bitet. Ennek decimálisan értéke 2^i. Például az $m_7^3 = ABC$ mintermnek három szomszédja van, az

$m_3^3 = \bar{A}BC$, az $m_5^3 = A\bar{B}C$ és az $m_6^3 = AB\bar{C}$. Mind a három mintermmel való különbsége kettő hatványa, konkrétan 4, 2 és 1. Ez a feltétel önmagában nem elegendő, mivel például az $m_2^3 = \bar{A}\bar{B}\bar{C}$ és az $m_1^3 = \bar{A}\bar{B}C$ mintermek decimális indexének különbsége 1, de mégsem szomszédosak.

A második feltétel az indexek bináris súlyának különbséges 1 kell, hogy legyen. A bináris súly azt mondja meg, hogy a számot binárisan felírva hány darab 1-es van benne. Mivel a két bináris szám csak az *i*-edik bitben különbözik egymástól, ezért amelyikben 1-es van 0 helyett, annak a bináris súlya eggyel nagyobb. Például az $m_7^3 = ABC$ bináris súlya 3, a vele szomszédos $m_3^3 = \bar{A}BC$, $m_5^3 = A\bar{B}C$ és $m_6^3 = AB\bar{C}$ bináris súlya 2. Ez a feltétel sem elegendő önmagában, mivel például az $m_2^3 = \bar{A}\bar{B}\bar{C}$ bináris súlya 1, az $m_5^3 = A\bar{B}C$ bináris súlya 2, azaz bináris súlyuk különbsége 1, de mégsem szomszédosak.

A harmadik feltétel szerint a nagyobb decimális indexnek nagyobb a bináris súlya. Amelyik számban 1-es van 0 helyett, annak decimális értéke 2^i értékkel nagyobb. Ez a feltétel sem elegendő önmagában, mivel például az $m_6^3 = AB\bar{C}$ bináris súlya 2, az $m_3^3 = \bar{A}BC$ bináris súlya 0, azaz a nagyobb értékűnek nagyobb a bináris súlya, de mégsem szomszédosak.

A három feltétel együtt biztosítja számunkra azt, hogy a két minterm minimális legyen. Azaz összefoglalva két minterm szomszédos, ha a következő feltételek teljesülnek.

- A decimális indexük különbségének értéke 2 hatványa.
- A bináris súlyuk különbsége 1.
- A nagyobb indexű term súlya nagyobb.

A fenti feltételek megfelelően leírják, hogy két minterm mikor szomszédos, azaz mikor lehet őket összevonni. Ez viszont csak az első lépéshez elegendő, mivel az implikánsok nem csak egyetlen mintermet tartalmazhatnak. Az implikánsok összevonásánál a feltétel szerint az implikánsok által lefedett mintermek páronként szomszédosak. Ez csak abban az esetben fordulhat elő, ha két implikánsnak ugyanazokat a változókat tartalmazza, azaz ugyanazok a változók lettek belőlük kiegyszerűsítve.

Prímimplikánsok meghatározása

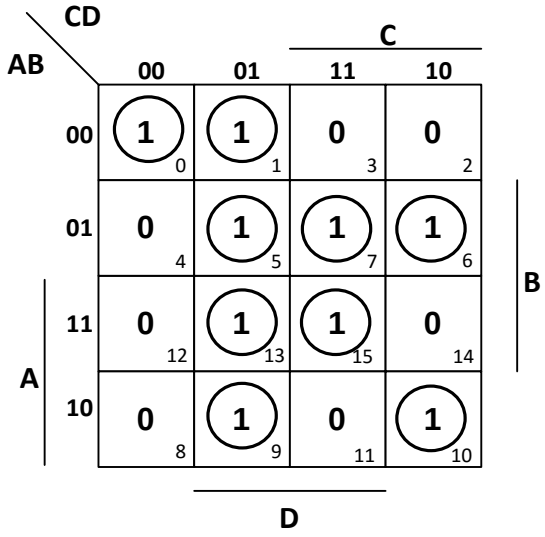
Ezek után gyakorlatilag ugyanaz a módszer használható, mint a Karnaugh tábla esetében. A kevesebb mintermet tartalmazó implikánsokat kell összevonni és nagyobbakat készíteni. Első lépésben az egy mintermet tartalmazó implikánsokat kell összegyűjteni, a Karnaugh táblában ez az egyes hurkokat jelenti. Ezeket egy táblázatba kell rendezni a bináris súlyuk alapján csoportosítva, mivel így a második feltétel ellenőrzésére a későbbiekben nem lesz szükség. A csoportok között aláhúzás szerepel a táblázatban a jobb láthatóság végett. A mintermek összevonása során a szomszédos csoportokban lévő összes mintermet párosítani kell és ellenőrizni, hogy a három korábban felírt feltétel teljesül-e. Amennyiben igen, akkor ez egy új implikánst eredményez, mely egy új táblázatban kerül összegyűjtésre. Ebben a táblázatban az implikánsok mellett fel kell jegyezni, hogy mennyi volt a decimális különbségük. A decimális különbség mutatja, hogy melyik változót lehetett kiegyszerűsíteni az összevonással, a decimális különbség 2-es alapú logaritmusa jelöli ki az elhagyható változó helyiértékét. Továbbá meg kell jelölni, hogy mely mintermeket lehetett bevonni valamely implikánsba, mivel ezek a mintermek nem lehetnek prímimplikánsok. Ilyen módon el lehet készíteni az összes kettő mintermet tartalmazó implikánst. Érdemes az újonnan kapott implikánsokat is az első mintermjük szerint bináris súlyuk alapján csoportosítani, ezt aláhúzással jelöljük a táblázatban, így csak a szomszédos csoportokban lévő implikánsokat kell megvizsgálni.

A Karnaugh tábla használatához hasonlóan itt is iteratívan addig kell új táblázatokat készíteni az előzőből, amíg keletkezik új implikáns. A ciklus végén azok az implikánsok lesznek a

prímimplikánsok, melyeket nem lehetett összevonni másik implikánssal, azaz nem lettek megjelölve az algoritmus során.

A példában legyen a minimalizálandó függvény ugyanaz, mint a Karnaugh táblás minimalizálás bemutatása során, azaz az $F^{n=4} = \sum_{i=0}^{2^n-1}(0,1,5,6,7,9,10,13,15)$. A könnyebb érthetőség miatt minden lépés után szerepel a Karnaugh tábla is. Első lépés a mintermek csoportosítása a bináris súlyuk szerint.

Minterm	Bináris érték	Bináris súly
0	0000	0
1	0001	1
5	0101	2
6	0110	
9	1001	
10	1010	
7	0111	3
13	1101	
15	1111	4



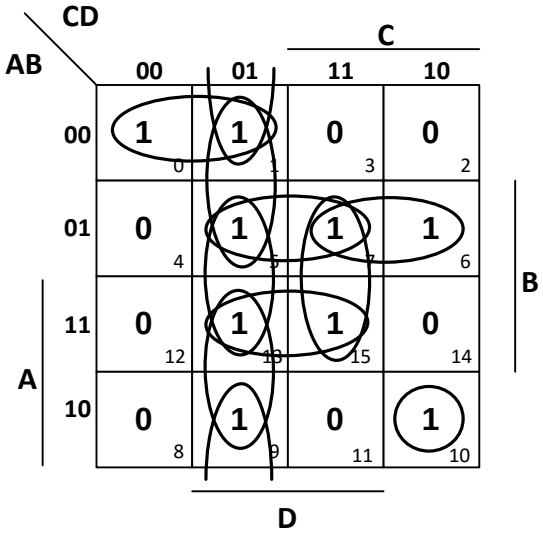
Következő lépésben a szomszédos csoportokban lévő mintermek kell összehasonlítani és amiket lehet, összevonni új implikánsá. Ahol mind a három, mintermekre vonatkozó szomszédossági feltétel teljesül, azok bekerülnek a következő táblázatba és az összevonható mintermek mellé „+” jel kerül. Például az 5 és 9 mintermre teljesülnek a feltételek, de például a 9 és 7 mintermekre nem teljesül a harmadik tulajdonság, azaz hogy a nagyobb bináris súlyú minterm decimális értéke nagyobb. Az összevonás eredményeképpen ugyanazok az implikánsok adódtak, mint a Karnaugh táblás minimalizálás során a kettes hurkok képzésénél és a 10-es minterm ugyanúgy nem vonható össze semmilyen másik mintermmel, azaz ő maga egy prímimplikáns.

Minterm	Bináris érték	Bináris súly
0+	0000	0
1+	0001	1
5+	0101	2
6+	0110	
9+	1001	
10	1010	
7+	0111	3

Implikáns	Decimális különbség
0, 1	(1)
1, 5	(4)
1, 9	(8)
5, 7	(2)
5, 13	(8)
6, 7	(1)
9, 13	(4)

13+	1101	
15+	1111	4

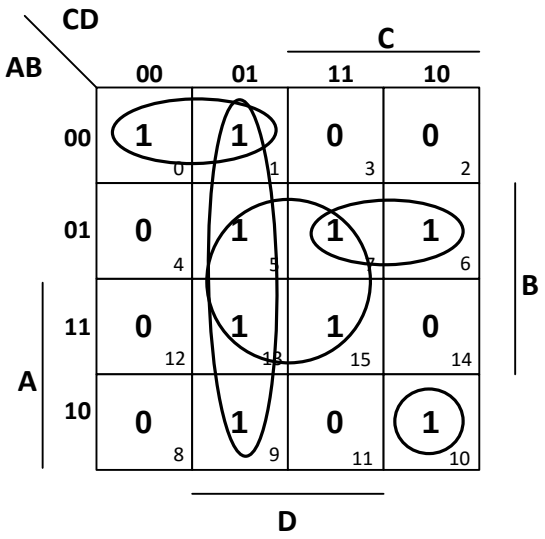
7, 15	(8)
13, 15	(2)



Ezek után következik az eredményül kapott implikánsok további összevonása. Ebben az esetben akkor vonhatóak össze az implikánsok, ha páronként teljesülnek a benne lévő mintermekre a szomszédossági feltétel és ebből következően a decimális különbségük megegyezik. Ez alapján össze lehet vonni az 1, 5 implikánst a 9, 13 implikánssal, ami a 1, 5, 9, 13 implikánst eredményezi ugyanúgy, mint az 1, 9 implikáns összevonása az 5, 13 implikánssal. A Karnaugh táblában is minden négyes hurkot két féle képen lehetett előállítani kettes hurkokból. Az összevonás során még egy új implikáns keletkezik, de annak decimális különbsége nem ugyanaz, mint a másiké, ezért ezeket a későbbiekben nem lehet összevonni.

Implikáns	**Decimális különbség**
0, 1	(1)
1, 5+	(4)
1, 9+	(8)
5, 7+	(2)
5, 13+	(8)
6, 7	(1)
9, 13+	(4)
7, 15+	(8)
13, 15+	(2)

Implikáns	**Decimális különbség**
1, 5, 9, 13	(4, 8)
5, 7, 13, 15	(2, 8)



A számjegyes minimalizálás során kapott prímimplikánsok megegyeznek azokkal a prímimplikánsokkal, amelyeket a Karnaugh táblás minimalizálás eredményezett.

A következő feladat a decimális indexekből a prímimplikánsok algebrai alakját meghatározni. Meg kell vizsgálni, hogy mely pozíciókban egyforma az összes minterm algebrai alakja és ez adja meg az prímimplikáns algebrai alakját. Ez alapján a következő képen kell meghatározni az előző példa prímimplikánsainak algebrai alakjait.

Decimális alak	Algebrai alakok	Eredő algebrai alak
10	$A\bar{B}C\bar{D}$	$A\bar{B}C\bar{D}$
0, 1	$\bar{A}\bar{B}\bar{C}\bar{D}$ $\bar{A}\bar{B}\bar{C}D$	$\bar{A}\bar{B}\bar{C}$
6, 7	$\bar{A}BC\bar{D}$ $\bar{A}BCD$	$\bar{A}BC$
1, 5, 9, 13	$\bar{A}\bar{B}\bar{C}D$ $\bar{A}B\bar{C}D$ $A\bar{B}\bar{C}D$ $AB\bar{C}D$	$\bar{C}D$
5, 7, 13, 15	$\bar{A}B\bar{C}D$ $\bar{A}BCD$ $AB\bar{C}D$ $ABCD$	BD

Egyszerűsített alak megadása

Az egyszerűsített alak meghatározásához elsősorban a prímimplikáns tábla használható. Ebből kiolvashatók a lényeges prímimplikánsok és szerencsés esetben ezek az összes mintermet tartalmazzák („lefedik”). Abban az esetben, ha a prímimplikáns tábla alapján ránézésre nem állapítható meg, hogy mely prímimplikánsokra van szükség, vagy többváltozós bonyolult függvényt kell minimalizálni, akkor segédfüggvényt használhatunk az eredmény meghatározásához.

A prímimplikáns táblában minden prímimplikánshoz tartozik egy sor és minden mintermhez tartozik egy oszlop. Ha egy prímimplikáns tartalmaz egy mintermet, akkor az adott cellába „*”-ot kell írni. A megkülönböztetett mintermek oszlopaiban egyetlen „x” szerepel, azaz azok a lényeges prímimplikánsok, amelyben van olyan minterm, melyek oszlopában egyetlen „x” van, ezeket „*”-al megjelöljük.

A $F^{n=4} = \sum_{i=0}^{2^4-1}(0,1,5,7,10,11,13,15)$ függvényhez öt prímimplikáns tartozik, azaz a prímimplikáns táblának 5 sora és 8 oszlopa lesz. A fentiek alapján kitölthető a táblázat és látható belőle, hogy 0 mintermet csak a 0, 1 prímimplikáns fedi le, azaz ez lényeges prímimplikáns. Hasonlóan a 10, 11 és az 5, 7, 13, 15 prímimplikánsok is lényeges prímimplikánsok. A példában a lényeges prímimplikánsok lefedik az összes mintermet, azaz ránézésre látszik a megoldás.

Sor	Prímimplikáns	0	1	5	7	10	11	13	15
*	0, 1	x	x						
	1, 5		x	x					
*	10, 11					x	x		
	11, 15						x		x
*	5, 7, 13, 15			x	x			X	x

Olyan eset is előfordulhat, hogy nem a legegyszerűbb prímimplikánsok maradnak meg a kiválasztás után. A következő példában egy darab négy és négy darab kettő mintermet tartalmazó prímimplikáns van. A prímimplikáns táblából jól látszik, hogy az összes kettő mintermet tartalmazó prímimplikáns lényeges prímimplikánst jelöl és ezek lefedik az összes mintermet, azaz az egyszerűsített függvényalakban a négy mintermet tartalmazó prímimplikánsra nincs szükség.

Sor	Prímimplikáns	1	5	6	7	11	12	13	15
*	1, 5	x	x						
*	6, 7			x	x				
*	12, 13						x	X	
*	11, 15					x			x
	5, 7, 13, 15		x		x			X	x

A következő példa azt az esetet mutatatja be, amikor nincs a rendszerben lényeges prímimplikáns, azaz ránézésre nem lehet egyértelműen eldönteni, hogy mely prímimplikánsokra van szükség a legegyszerűbb alak eléréséhez. Jelen esetben a prímimplikáns tábla minden oszlopában kettő darab „x” szerepel.

Sor	Prímimplikáns		1	3	7	9	13	15
	1, 3	a	x	x				
	1, 9	b	x			x		
	3, 7	c		x	x			
	7, 15	d			x			X
	13, 15	e					x	X
	9, 13	f				x	x	

Amikor ránézésre nem állapítható meg a legegyszerűbb alak, vagy amikor sok változó és prímimplikáns miatt áttekinthetetlen a táblázat, segédfüggvényt kell használni. A segédfüggvény gyakorlatilag logikai függvénye annak a feltételnek, hogy minden mintermet le kell fedni legalább egy prímimplikánssal. Ennek eléréséhez minden prímimplikánshoz be kell vezetni egy logikai változót, mely azt reprezentálja, hogy az szerepel-e az egyszerűsített függvényalakban. Ezeket jelöli az előző táblázatban az a, b, c, d, e és f. Azt, hogy egy

minterm le van fedve, úgy lehet leírni, hogy az őt lefedni tudó prímimplikánsokhoz tartozó logikai változókat VAGY kapcsolattal kell összefűzni. Azt, hogy minden minterm le van fedve, úgy lehet leírni, hogy a mintermekhez tartozó logikai kifejezéseket ÉS kapcsolattal kell összekapcsolni. Az eredményül kapott segédfüggvényt S-el jelölik.

A prímimplikáns változóknak egy olyan kombinációját kell megtalálni, hogy a segédfüggvény értéke 1-et vegyen fel. Ehhez át kell alakítani a függvény diszjunktív alakba a szorzások (ÉS kapcsolatok) elvégzésével és meg kell határozni az összes prímimplikánsát. Diszjunktív alak esetén a segédfüggvény akkor vesz fel 1 értéket, ha van olyan tagja, melynek értéke 1. Azaz ki lehet választani bármelyik tagot a segédfüggvényből, és ha az abban a tagban szereplő változókhoz tartozó prímimplikánsok segítségével lesz megvalósítva a logikai függvény, akkor minden minterm le lesz fedve. Mivel a legegyszerűbb alak megtalálása a cél, ezért azt a tagot kell kiválasztani, amelyben a legkevesebb prímimplikáns szerepel. Ha több olyan tag is van, melyek számossága minimális, akkor a választás közöttük tetszőleges.

Az előző példához tartozó segédfüggvény a következő képen néz ki $S = (a + b)(a + c)(c + d)(b + f)(e + f)(d + e)$. Ezt át lehet alakítani diszjunktív alakba a zárójelek felbontásával és egyszerűsíteni az elnyelési tulajdonság felhasználásával $S = (a + bc)(bc + cf - bd + df)(e + df) = (bc + acf + abd + adf)(e + df) = bce - bcdf - acef + acdf + abde + abdf + adef + adf = bce + adf + bcdf + acef + abde$. Az öt kapott prímimplikáns öt különböző megoldást jelöl, ezek közül az a bce és az adf a legegyszerűbbek, mivel ezek tartalmazzák a legkevesebb prímimplikánst. A bce alapján a függvény megvalósításához a b, a c és az e prímimplikánsokra van szükség, így a függvény egyszerűsített alakja $F = b + c + e = \bar{B}\bar{C}D + \bar{A}CD + A\bar{B}D$. Az adf alapján a függvény megvalósításához az a, a d és az f prímimplikánsokra van szükség, így a függvény egyszerűsített alakja $F = a + d + f = \bar{A}\bar{B}D + BCD + A\bar{C}D$.

Példa

Legyen adott a következő logikai függvény $F^{n=4} = \sum_{i=0}^{2^n-1}(4,5,10,11,13,14,15)$. A mintermek alapján fel lehet írni a kiinduló táblázatot.

Minterm	Bináris érték	Bináris súly
4	0100	1
5	0101	2
10	1010	
11	1011	3
13	1101	
14	1110	
15	1111	4

Az első lépésben meg kell határozni a kettő mintermet tartalmazó implikánsokat.

Minterm	Bináris érték	Bináris súly
4+	0100	1
5+	0101	2
10+	1010	
11+	1011	3
13+	1101	
14+	1110	
15+	1111	4

Implikáns	Decimális különbség
4, 5	(1)
5, 13	(8)
10, 11	(1)
10, 14	(4)
11, 15	(4)
13, 15	(2)
14, 15	(1)

Ezek után meghatározhatók a négy mintermet tartalmazó implikánsokat.

Implikáns	Decimális különbség
4, 5	(1)
5, 13	(8)

Implikáns	Decimális különbség
10, 11, 14, 15	(1, 4)

10, 11+	(1)
10, 14+	(4)
11, 15+	(4)
13, 15	(2)
14, 15+	(1)

Mivel csak egyetlen darab négy mintermet tartalmazó implikáns van, ezért azt már nem lehet összevonni másikkal, azaz nincs nyolc mintermet tartalmazó implikáns. A prímimplikáns tábla segítségével meghatározhatók a lényeges prímimplikánsok.

Sor	Prímimplikáns		4	5	10	11	13	14	15
*	4, 5	*a*	x	x					
	5, 13	*b*		x			x		
	13, 15	*c*					x		x
*	10, 11, 14, 15	*d*			x	x		X	x

A lényeges prímimplikánsok nem fedik le az összes mintermet, ezért a segédfüggvényt segítségével kell meghatározni a szükséges prímimplikánsokat. $S = a(a-b)dd(b+c)d(c+d) = add(b+c) = abd - abc$. Azaz a két lényeges prímimplikáns mellett tetszőleges, hogy a b vagy a c prímimplikáns szerepel a megoldásban. Most legyen b a kiválasztott prímimplikáns. Ekkor az eredő függvényhez meg kell határozni a prímimplikánsok algebrai alakjait.

Decimális alak	Algebrai alakok	Eredő algebrai alak
4, 5	$\bar{A}B\bar{C}\bar{D}$ $\bar{A}B\bar{C}D$	$\bar{A}B\bar{C}$
5, 13	$\bar{A}B\bar{C}D$ $AB\bar{C}D$	$B\bar{C}D$
10, 11, 14, 15	$A\bar{B}C\bar{D}$ $A\bar{B}CD$ $ABC\bar{D}$ $ABCD$	AC

Ebből az eredő függvényalak $F = \bar{A}B\bar{C} + B\bar{C}D - AC$.

A következő példában legyen $F^{n=5} = \sum_{i=0}^{2^n-1}(0,1,6,7,14,15,17,21,22,23,24,30,31)$ egy ötváltozós logikai függvény. A mintermek alapján fel lehet írni a kiinduló táblázatot.

Minterm	Bináris érték	Bináris súly
0	00000	0
1	00001	1
6	00110	2
17	10001	
24	11000	
7	00111	3
14	01110	
21	10101	
22	10110	
15	01111	4
23	10111	
30	11110	
31	11111	5

Az első lépésben meg kell határozni a kettő mintermet tartalmazó implikánsokat.

Minterm	Bináris érték	Bináris súly
0+	00000	0
1+	00001	1
6+	00110	2
17+	10001	
24	11000	
7+	00111	3
14+	01110	
21+	10101	
22+	10110	
15+	01111	4
23+	10111	
30+	11110	
31+	11111	5

Implikáns	Decimális különbség
0,1	(1)
1,17	(16)
6,7	(1)
6,14	(8)
6,22	(16)
17,21	(4)
7,15	(8)
7,23	(16)
14,15	(1)
14,30	(16)
21,23	(2)
22,23	(1)
22,30	(8)
15,31	(16)
23,31	(8)
30,31	(1)

Ezek után meghatározhatók a négy mintermet tartalmazó implikánsokat.

Implikáns	Decimális különbség
0, 1	(1)
1, 17	(16)
6, 7+	(1)
6, 14+	(8)
6, 22+	(16)
17, 21	(4)
7, 15+	(8)
7, 23+	(16)
14, 15+	(1)
14, 30+	(16)
21, 23	(2)
22, 23+	(1)
22, 30+	(8)
15, 31+	(16)
23, 31+	(8)
30, 31+	(1)

Implikáns	Decimális különbség
6, 7, 14, 15	(1, 8)
6, 7, 22, 23	(1, 16)
6, 14, 22, 30	(8, 16)
7, 15, 23, 31	(8, 16)
14, 15, 30, 31	(1, 16)
22, 23, 30, 31	(1, 8)

Ezek után meghatározhatók a nyolc mintermet tartalmazó implikánsokat.

Implikáns	Decimális különbség
6, 7, 14, 15+	(1,8)
6, 7, 22, 23+	(1,16)
6, 14, 22, 30+	(8,16)
7, 15, 23, 31+	(8,16)
14, 15, 30, 31+	(1,16)
22, 23, 30, 31+	(1,8)

Implikáns	Decimális különbség
6, 7, 14, 15, 22, 23, 30, 31	(1, 8, 16)

Mivel csak egyetlen darab nyolc mintermet tartalmazó implikáns van, ezért azt már nem lehet összevonni másikkal, azaz nincs tizenhat mintermet tartalmazó implikáns. A prímimplikáns tábla segítségével meghatározhatók a lényeges prímimplikánsok.

Sor	Prímimplikáns	0	1	6	7	14	15	17	21	22	23	24	30	31
*	24	*a*										x		

*	0, 1	*b*	x	x												
	1, 17	*c*		x					x							
	17, 21	*d*							x	x						
	21, 23	*e*								x		x				
*	6, 7, 14, 15, 22, 23, 30, 31	*f*			x	x	x	x			x	x			x	x

A lényeges prímimplikánsok nem fedik le az összes mintermet, ezért a segédfüggvény segítségével kell meghatározni a szükséges prímimplikánsokat. $S = b(b + c)ffff(c + d)(d + e)f(e + f)aff = bf(c - d)(d + e)a = abf(cd + ce + d - de) = abf(ce + d) = abcef + abdf$

. A két lehetséges megvalósításból az $abdf$ tartalmaz kevesebb prímimplikáns, ezért ez az optimális megoldás. Az eredő függvényhez meg kell határozni a prímimplikánsok algebrai alakjait.

Decimális alak	Algebrai alakok	Eredő algebrai alak
24	$A\bar{B}C\bar{D}\bar{E}$	$A\bar{B}C\bar{D}\bar{E}$
0, 1	$\bar{A}\bar{B}\bar{C}\bar{D}\bar{E}$ $\bar{A}\bar{B}C\bar{D}\bar{E}$	$\bar{A}\bar{B}C\bar{D}$
17, 21	$A\bar{B}\bar{C}\bar{D}\bar{E}$ $A\bar{B}C\bar{D}\bar{E}$	$A\bar{B}\bar{D}\bar{E}$
6, 7, 14, 15, 22, 23, 30, 31	$\bar{A}\bar{B}CD\bar{E}$ $\bar{A}\bar{B}CDE$ $\bar{A}BCDE$ $\bar{A}BCD\bar{E}$ $A\bar{B}CD\bar{E}$ $A\bar{B}CDE$ $ABCD\bar{E}$ $ABCDE$	CD

Ebből az eredő függvényalak $F = A\bar{B}C\bar{D}\bar{E} + \bar{A}\bar{B}C\bar{D} + A\bar{B}\bar{D}\bar{E} + CD$.

Pannon Egyetem
Képfeldolgozás és Neuroszámítógépek Tanszék



Digitális Rendszerek és Számítógép Architektúrák

1. előadás: Számrendszerek,
Nem-numerikus információ ábrázolása

Előadó: Vörösházi Zsolt
Szolgay Péter

Jegyzetek, segédanyagok:

- Könyvfejezetek:
  - http://www.knt.vein.hu
    → Oktatás → Tantárgyak → Digitális Rendszerek és Számítógép Architektúrák (nappali)
    (chapter02.pdf)
- Fóliák, óravázlatok .ppt (.pdf)
- Feltöltésük folyamatosan

Információ ábrázolás:

- A) Számrendszerek:
  - I.) Egész típusú:
    - előjel nélküli,
    - 1-es komplemens,
    - előjeles 2-es komplemens számrendszerek
  - II.) Fixpontos,
  - III.) Lebegőpontos (IBM-32, DEC-32, IEEE-32),
  - Excess kód (exponens kódolására)
- B) Nem-numerikus információ kódolása
  - Hibajavítás és detektálás (Hamming kód)

A) Számrendszerek

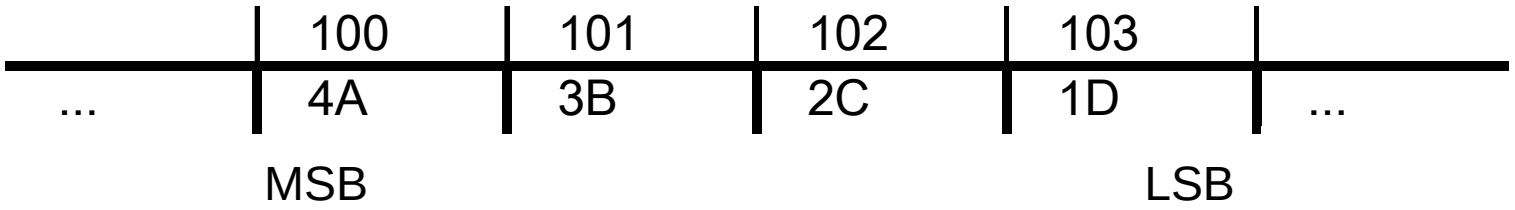
Endianitás (endianness)

- A számítástechnikában, az **endianitás** (byte-sorrend a jó fordítás) az a tulajdonság, ami bizonyos adatok - többnyire kisebb egységek egymást követő sorozata - tárolási és/vagy továbbítási sorrendjéről ad leírást (pl. két protokoll, vagy busz kommunikációja). Ez a tulajdonság döntő fontosságú az integer értékeknek a számítógép memóriájában byte-onként való tárolása (egy memória címhez relatívan), továbbítása esetében
- Byte sorrend megkötés:
  - Big-Endian formátum
  - Little-Endian formátum

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

„Nagy a végén” - Big-endian

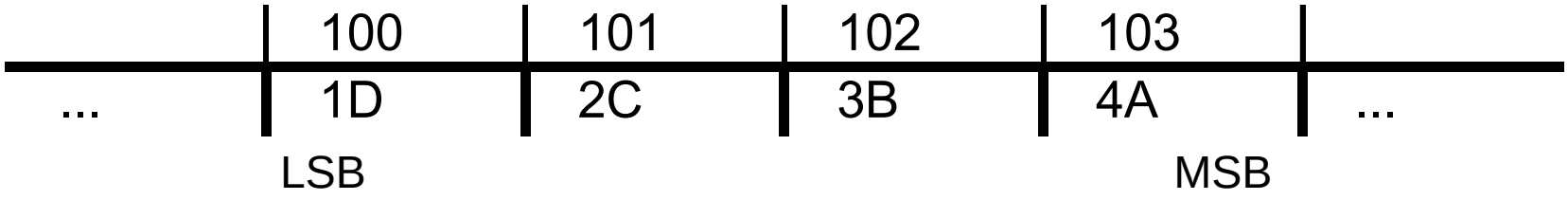
- A 32 bites egész értéket, (ami legyen „4A 3B 2C 1D”, a 100 címtől kezdve) tároljuk a memóriában, **1 byte-os** elemi tárolókból, 1 byte-onként növekvő címekkel rendelkezik, ekkor a tárolást a következők szerint végzi:



- Ebben az esetben, a „legjellemzőbb” byte - erre általában az ismert angolkifejezést "most significant byte" használják a számítástechnikában (rövidítve *MSB*, ami itt a „4A”) - a memóriában az *legalacsonyabb címen van tárolva*, míg a következő "jellemző byte" (3B) a következő, egyel nagyobb címen van tárolva, és így tovább.
- Bit-reversed format!
- Pl: beágyazott processzorok FPGA-n (MicroBlaze, PowerPC)

„Kicsi a végén” - Little-endian

- Ekkor a 32-bites „4A 3B 2C 1D” értéket a következő módon tárolják
- 100 101 102 103...”1D 2C 3B 4A”...
  Így, a kevésbé jellemző ("legkisebb") byte (az angol least significant byte rövidítéséből *LSB* néven ismert) az első, és ez az 1D, tehát a *kis vég kerül előre:*



- Általánosan használt formátum

I.) Egész típusú számrendszer:

- Bináris számrendszer: 1 / 0 (I / H, T / F)
- N biten 2^N lehetséges érték reprezentálható
- Összehasonlító táblázat:

Table 2.1. Number of Representable Values.

Number of Bits	Number of Representable Values	Machines. Uses
4	16	4004, control
8	256	8080, 6800 control, communication
16	65.536	PDP11, 8086, 32020
32	4.29×10^9	IBM 370, 68020. VAX11/780
48	1.41×10^{14}	Unisys
64	1.84×10^{19}	Cray, IEEE (dp)

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:**

101101 $\Rightarrow$1x2^5 + 0x2^4 + 1x2^3 + 1x2^2 + 0x2^1 + 1x2^0 = 45

b.) előjeles (kettes komplemens) rendszer:

- 2's comp:

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

- reprezentálható értékek határa: –(2^(N-1)) től 2^(N-1)–1 ig
- Ha MSB='1', negatív szám
- Fólia: 2.2 táblázat
- Pl. 101101 ⇒ –1x2^5 + 0x2^4 + 1x2^3 + 1x2^2 + 0x2^1 + 1x2^0 = –19
- Pl.

$$\begin{array}{r} 100000000 \\ -\ 01001100 \\ \hline 10110100 \end{array} \qquad \begin{array}{r} 256 \\ -76 \\ \hline 180 \end{array}$$



$$\begin{array}{rl} 01001100 & \\ 10110011 & \text{1's kompl.} \\ +\quad\ 1 & +1 \\ \hline 10110100 & \qquad -76 \end{array}$$

2.2 táblázat:

Table 2.2. 8-Bit Two's Complement Representations.

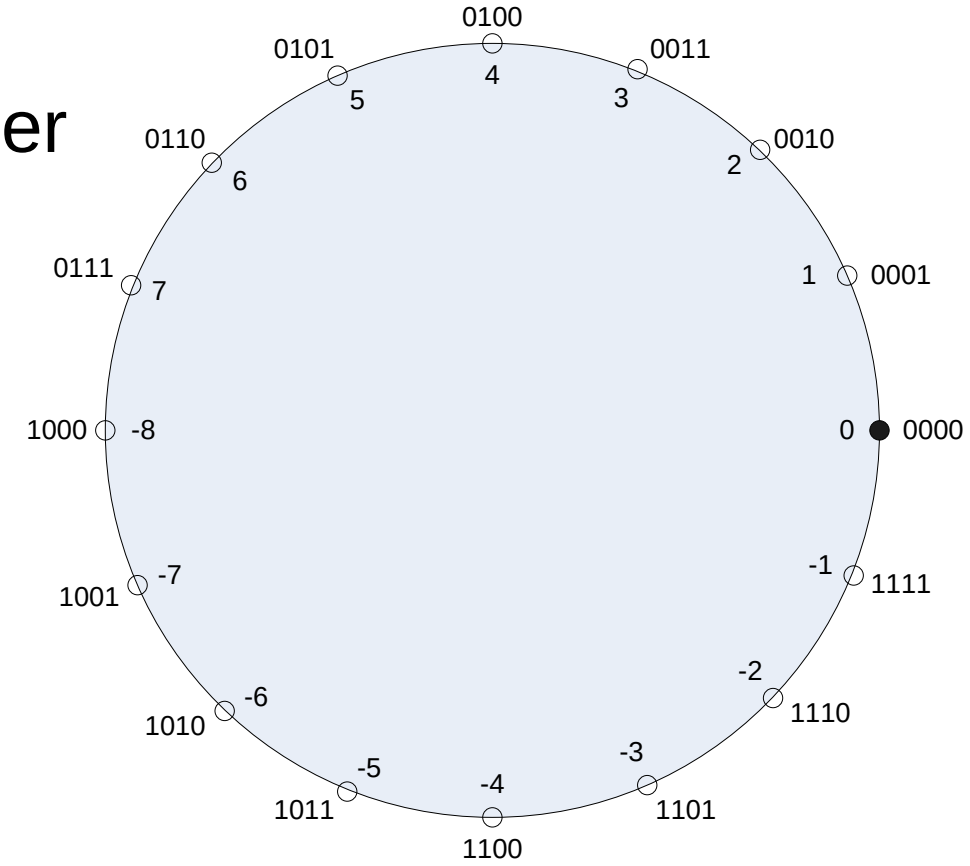
Bit Pattern	*Value*	*Note*
01111111	127	Largest representable value.
01111110	126	
01111101	125	
...	...	
...	...	
00000010	2	Note that leading zero indicates
00000001	1	positive number.
00000000	0	Unique representation of **zero.**
11111111	–1	Minus one is always all ones.
11111110	–2	Note that leading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

PI: Körkörös számláló (circular nature):

- 4 bites 2's komplemens rendszer
- Overflow:

  0111 -> 1000
- Underflow:

  1000 -> 0111



c.) 1-es komplemens rendszer:

- V értékű, N bites rendszer: $\mathbf{2^N–1–V}$.
- „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!! (ellenőrzés szükséges)
- end-around carry: amelyben a részeredményhez kell hozzáadni a végrehajtás eredményét

Példa:

Example 2.3: ***One's complement*** arithmetic: Consider a 6-bit one's complement system. Represent 15. –15. 13. and –13 in this system. Then perform the following additions: 15 + 13, 15 + **(–13),** 13 + **(–13),** and 13 + (–15).

The numbers are derived in a simple fashion:

Decimal Value	*One's Complement*	*Comment*
15	001111	Positive numbers same as two's complement.
–15	110000	Complement bits to negate.
13	001101	Positive numbers same as two's complement.
–13	110010	Complement bits to negate.
Now for the additions:		
15	001111	This proceeds just like the two's
+ 13	+ 001101	complement version.
28	0 011100	No cany out: number is correct, and the result is as we expect.
15	001111	The addition is done in the normal
+ –13	+ 110010	fashion, but
2	1 000001	the result of one is incorrect;
		however, the presence of a carry says we
	+ 000001	should add that as a 1 in the LSB
	000010	which gives the expected result.
13	001101	This time we will add a positive number
+ –13	+ 110010	to its negative (which is just complement)
0	111111	and end up with all ones — a valid zero.
13	001101	Here the positive number is smaller
+ –15	+ 110000	than the negative number, so result
–2	0 111101	is negative; no cany — the value is correct.

end-around: cirkuláris carry, körkörös átvitel

II.) Fixpontos számrendszer

- Műveletek:
  - +, - : ugyanaz, mint az egész szám rdsz. esetén
  - *, / : meg kell bizonyosodni arról, hogy a tizedespont helyén maradt-e

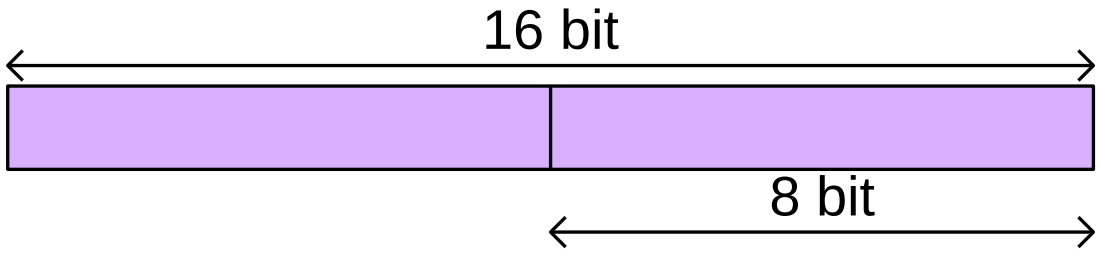
$$V_{\text{FIXED POINT}} = -b_{N-1} \times 2^{N-p-1} + \sum_{i=0}^{N-2} b_i \times 2^{i-p}$$

- p: radix (tizedes) pont helye, tizedes jegyek száma
- *differencia*, $\Delta r = 2^{-p}$ (számrendszer finomsága)
  - *Ha p=0 → Δr=1, egész rendszer, különben fixpontos*
- Alkalmazás: pl. jelfeldolgozás (DSP – Texas Instruments)

Példa: fixpontos rendszer

- Kérdés: Legyen egy 16 bites 2's comp. fixpontos rdsz. ahol p=8. V(smallest)=?, V(largest)=?, Δr = *? (decimális értékben megadva)*

- Megoldás:



- V(smallest absolute)=00000000.00000001=2^{-8}=$\underline{0{,}390625*10^{-2}}$
- V(largest absolute)=01111111.11111111=$\underline{\sim 128}$
- V(largest negative)=10000000.00000000=-2^{7}=$\underline{-128}$
- Differencia $\Delta r = 2^{-8}$=$\underline{0{,}390625*10^{-2}}$
- !!DE V(zero)=00000000.00000000 vagy 11111111.11111111

Excess kód rendszer

- Lebegőpontos számok *kitevőit* (*exponens*-eit) tárolják / kódolják ezzel a módszerrel
- S: a reprezentálni kívánt érték (eredmény), amit tárolnunk
- V: a szám valódi értéke, E: az excess
- S = V + E.
- Két számot összeadunk, akkor a következő történik:

$$S1 + S2 = (V1 + E) + (V2 + E) = (V1 + V2) + 2 \times E$$

- pontos eredmény: [(V1+V2) + E] (ki kell vonnunk E-t!)
- Fólia: 2.4, 2.5, 2.6-os példák

Példa 2.4:

Example 2.4: Number representation in excess codes: What is the representation of $+37_{10}$ in an 8-bit excess *128* code? What is the representation of -23_{10} in an 8-bit excess 128 code? What is the sum of the two numbers, in the 8-bit excess *128* code?

An 8-bit unsigned number can represent values between **0** and ***255.*** The excess representation can then represent values from **–**128 to **+127.**

+ 128	10000000	This is the excess.
+ 37	00100101	The value to be represented.
165	10100101	The representation of 37_{10} in excess 128 code.
+ 128	10000000	This is the excess.
– 23	00010111	The value to be represented.
105	01101001	The representation of -23_{10} in excess **128** code.
165	10100101	This is +37 in excess **128.**
+ 105	+ 01101001	This is –**23** in excess 128.
270	1 00001110	Note the carry out in this operation. **270** is too big to represent in **8** bits; to correct for the 2 **x** E that is in this sum, subtract **128.**
– 128	– 10000000	In binary, is this add or subtract?
142	10001110	This is the representation of **14,** the correct result. in excess **128.**

Táblázat 2.3: BCD

Table 2.3. Binary Coded Decimal (RCD) Representations.

Bit Pattern	*Value*
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	Not valid
1011	Not valid
1100	Not valid
1101	Not valid
1110	Not valid
1111	Not valid

Példa 2.5:

Example 2.5: BCD excess 3 system: Consider *a* system that works with 3-digit decimal numbers, and it stores the digits in excess 3 format. What is the representation of **573?** What is the representation of **142?** Add the two numbers, and give the correct result in excess 3 format.

The numbers are handled on a digit-by-digit basis, with the excess being included with each digit:

Decimal	Binary	
+3 3 3	0011 0011 0011	This is the excess.
5 7 3	0101 0111 0011	And the number to be represented.
8 10 6	1000 1010 0110	The excess 3 representation.
+3 3 3	0011 0011 0011	This is the excess.
1 4 2	0001 0100 0010	This is the number to be represented.
4 7 5	0100 0111 0101	The excess 3 representation.
573	1000 1010 0110	Do the addition in decimal and in
+ 142	0100 0111 0101	binary. Correct as needed to make output correct.
715	1100 1 0001 1011	

Carry out of second set of 4 bits indicates that the most significant digit should be incremented by one. Also indicates that this value is **correct** as it stands (since 2 × E = 6 and the **carry** out indicates that the number overflowed into the next digit) so we need to add 3. Therefore, the MSD needs to **be** incremented by one and decremented by 3; the middle digit needs to have 3 added; and the **LSD** needs to be decremented by 3.

(-3+1)	(+3)	(-3)
−0010	+0011	−0011
1010	0100	1000
10	4	8

Which is the correct excess 3 representation for 715_{10}.

MSD: Most significant digit

LSD: Least significant digit

III.) Lebegőpontos rendszer:

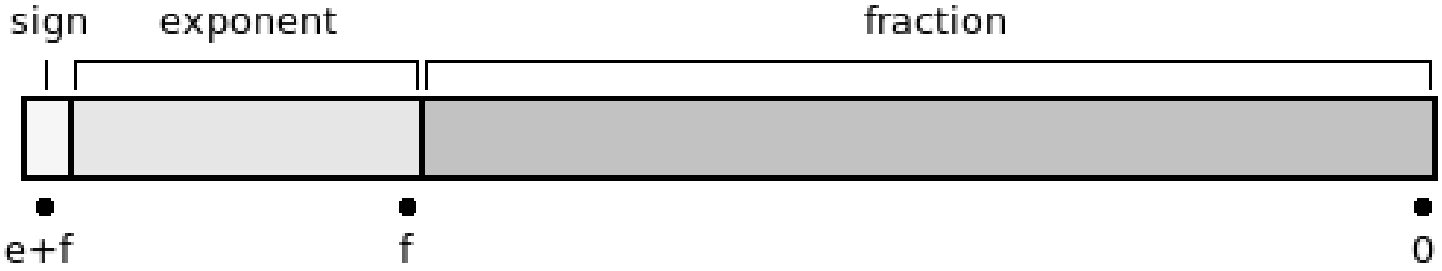
- 7 különböző tényező: *a számrendszer alapja, előjele és nagysága, a mantissza alapja, előjele és hosszúsága, ill. a kitevő alapja*.
- matematikai jelölés:

$$(\text{előjel})\ \text{Mantissza} \times \text{Alap}^{\text{Kitevő}}$$

- Fixpontosnál nagyságrendekkel kisebb vagy nagyobb számok ábrázolására is mód van:
  - Pl: Avogadro-szám: 6.022*10^23
  - Pl: proton tömege 1.673*10^-24 g

IEEE 754-1985

- Szabvány bináris lebegőpontos számok tárolásánál, amely tartalmazza még a:
  - negatív zérust: –0 = 111…1
  - Normalizálatlan számok
  - NaN: nem szám
  - – /+ végtelen



Sign-magnitude format („előjel-hossz” formátum): előjel külön kerül tárolásra (MSB), exponens eltolt (Excess-el kódolt), törtrész utána következik.

Lebegőpontos rendszer jellemzői

- Számrendszer / kitevő alapja: r_b r_e
- Mantissza értéke: $V_M = \sum_{i=0}^{N-1} d_i \times r_b^{i-p}$
  - Maximális: $V_{M_{max}} = 0.d_m d_m d_m ... = (1 - r_b^{-N})$
  - Minimális: $V_{M_{min}} = 0.100 .. = 1/r_b$
  - Radix pont helye: p
  - Mantissza bitjeinek száma: m
- Exponens értéke (max / min): V_E $V_{E_{max}}$ $V_{E_{min}}$
- Lebegőpontos szám értéke: $V_{FPN} = (-1)^{SIGN} V_M \times r_b^{V_E}$

Normalizált lebegőpontos rendszer jellemzői

- Ábrázolható maximális érték: $V_{FPN(MAX)} = V_{M(MAX)} \times r_b^{V_{E(MAX)}}$
- Ábrázolható minimális érték: $V_{FPN(MIN)} = V_{M(MIN)} \times r_b^{V_{E(MIN)}}$
- Legális mantisszák száma: $NLM_{FPN} = (r_b - 1) \times r_b^{m-1}$
- Legális exponensek száma: $NLE_{FPN} = V_{E(MAX)} + \left|V_{E(MIN)}\right| + 1_{ZERO}$
- Ábrázolható értékek száma: $NRV_{FPN} = NLM_{FPN} \times NLE_{FPN}$

- Normalizálás: mantissza értékét általában [0…~1] közé
- Pl: $32\ 768_{10} = 0.32768*10^5 = 3.2768*10^4 = 32.768*10^3 = 327.68*10^2 = 3267.8\ *10^1$

Példa: normalizált lebegőpontos rendszer

- Adott: Legyen $r_b = 10$, $r_e = 10$, $m = 3$, $e = 2$
- Kérdés: jellemző paraméterek?
- Megoldás:

$$V_{M(MAX)} = 0.999 = 1.000 - 10^{-3}$$
$$V_{M(MIN)} = 0.100$$
$$V_{E(MAX)} = (r_e \wedge e - 1) = 99$$
$$V_{E(MIN)} = -(r_e \wedge e - 1) = -99$$
$$V_{FPN(MAX)} = 0.999 \times 10^{99}$$
$$V_{FPN(MIN)} = 0.100 \times 10^{-99}$$

$$NLM_{FPN} = 9 \times 10 \times 10 = 900 = 9 \times 10^2$$
$$NLE_{FPN} = 99 + |-99| + 1_{ZERO} = 199$$
$$NRV_{FPN} = 2 \times 900 \times 199 = 358{,}200$$

Normalizált lebegőpontos rdsz.

Table 2.4. 6-Bit Normalized Floating Point System, Base 2.

$r_b = 2,\ r_e = 2,\ m = 4,\ e = 2$

				$V_E \rightarrow$	00	01	10	11
				$2^{V_E} \rightarrow$	1	2	4	8
V_M base 2				V_M	$V_M \times 2^{V_E}$			
1	0	0	0	$\frac{1}{2}$	$\frac{1}{2}$	1	2	4
1	0	0	1	$\frac{9}{16}$	$\frac{9}{16}$	$1\frac{1}{8}$	$2\frac{1}{4}$	$4\frac{1}{2}$
1	0	1	0	$\frac{5}{8}$	$\frac{5}{8}$	$1\frac{1}{4}$	$2\frac{1}{2}$	5
1	0	1	1	$\frac{11}{16}$	$\frac{11}{16}$	$1\frac{3}{8}$	$2\frac{3}{4}$	$5\frac{1}{2}$
1	1	0	0	$\frac{3}{4}$	$\frac{3}{4}$	$1\frac{1}{2}$	3	6
1	1	0	1	$\frac{13}{16}$	$\frac{13}{16}$	$1\frac{5}{8}$	$3\frac{1}{4}$	$6\frac{1}{2}$
1	1	1	0	$\frac{7}{8}$	$\frac{7}{8}$	$1\frac{3}{4}$	$3\frac{1}{2}$	7
1	1	1	1	$\frac{15}{16}$	$\frac{15}{16}$	$1\frac{7}{8}$	$3\frac{3}{4}$	$7\frac{1}{2}$

Smallest fraction = 0.1000_2 $= \frac{1}{2}$

Largest fraction = 0.1111, $= \frac{15}{16}$

Smallest number = $0.1000_2 \times 2^0$ $= \frac{1}{2}$

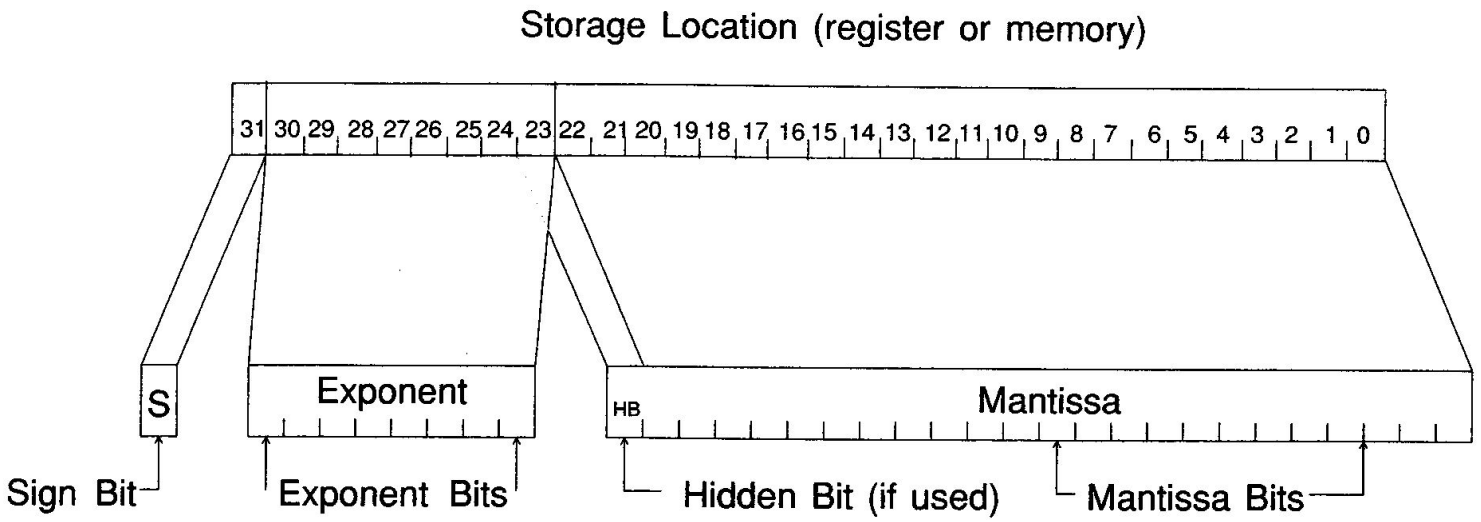
Largest number = $0.1111_2 \times 2^3$ $= 7\frac{1}{2}$

Number of fractions = 1 x 2 x 2 x 2 = 8

Number of values = 8 x 4 = 32

Lebegőpontos ábrázolás: Rejtett (hidden bit) technika

- „Vezető” egyesek számát a legtöbb rendszer tervezésekor konstans értékként definiálják (1)
  - Ilyen a *mantissza legmagasabb helyiértékű bitje*, amelyet rejtett (*hidden: HB*) bitnek hívunk, amely közvetlenül az exponensbitek mögött helyezkedik el.
  - Ezt beállítva, duplájára nő a legális mantisszák, így az ábrázolható értékek száma is. Másrészt a nullát nem könnyen tudjuk reprezentálni, mivel a legkisebb ábrázolható érték a 00 0000, ami a HB ‘1’-es konstansként való definiálása miatt $0.1000_2 \times 2^0$ = ½ -nek felel meg (m=4, e=2, r_b=r_e=2 esetén)!



Példa: 2-es alapú **DEC 32**-bites, normalizált lebegőpontos rendszer

- Adott: r_b=2, r_e=2, m=23+(1)=24 együtt, p=24 (m=p!), e=8, az exponenst tároljuk Excess-128 kódolással, és a számokat tároljuk "előjel-hossz" formátumban (~tekintsük a mantisszát pozitívnak).

$$V_{M(MIN)} = 0.\underbrace{1000\ldots_2}_{24db} = 1/2$$

$$V_{M(MAX)} = 0.1111\ldots_2 = 0.999999940395 = 1.0 - 2^{-24}$$

$$V_{E(MIN)} = -(r_e{}^{e-1} - 1) = -(2^{8-1} - 1) = -127$$

$$V_{E(MAX)} = r_e{}^{e-1} - 1 = 2^{8-1} - 1 = 127$$

$$V_{FPN(MIN)} = 0.1000\ldots_2 \times 2^{-127} = 2.9387 \times 10^{-39}$$

$$V_{FPN(MAX)} = 0.1111\ldots_2 \times 2^{+127} = 1.7014 \times 10^{38}$$

$$NLM_{FPN} = 2^{23} = 8,388,608$$

$$NLE_{FPN} = 127 + \left|-127\right| + 1_{ZERO} = 255 = (r_e{}^{e} - 1) = 2^8 - 1$$

$$NRV_{FPN} = 2^{23} \times (2^8 - 1) = 2.139 \times 10^9$$

Példa: 16-os alapú **IBM-32** bites normalizált lebegőpontos rendszer

- Adott: r_b=16, r_e=2, m=6, p=6, e=7, az exponenst tároljuk Excess-64 kódolással, és a számokat tároljuk "előjel-hossz" formátumban (tekintsük a mantisszát pozitívnak).

$$V_{M(MIN)} = 0.100000_{16} = 1/16$$

$$V_{M(MAX)} = 0.\text{FFFFFF}_{16} = 0.999999940395 = 1.0 - 16^{-6}$$

$$V_{E(MIN)} = -(r_e^{\,e-1} - 1) = -(2^{7-1} - 1) = -63$$

$$V_{E(MAX)} = r_e^{\,e-1} - 1 = 2^{7-1} - 1 = 63$$

$$V_{FPN(MIN)} = 0.100000 \times 16^{-63} = 8.636 \times 10^{-78}$$

$$V_{FPN(MAX)} = 0.\text{FFFFFF}_{16} \times 16^{+63} = 7.237 \times 10^{75}$$

$$NLM_{FPN} = 15 \times 16^5 = 15,728,640$$

$$NLE_{FPN} = 63 + |-63| + 1_{ZERO} = 127 = 2^7 - 1$$

$$NRV_{FPN} = 15 \times 16^5 \times (2^7 - 1) = 1.9975 \times 10^9$$

Bővebb tartomány mint a DEC!

7%-al kevesebb mint a DEC!!

Példa: **IEEE-32** bites normalizált lebegőpontos rendszer

- Adott: r_b=2, r_e=2, m=24, de p=23! (+HB!), , e=8, az exponenst tároljuk Excess-127 kódolással, és a számokat tároljuk "előjel-hossz" formátumban (~tekintsük a mantisszát pozitívnak).

$$V_{M(MIN)} = 1.\underbrace{000\ldots_2}_{23db} = 1$$

$$V_{M(MAX)} = 1.111\ldots_2 = 1.99999988 = 2.0 - 2^{-23}$$

$$V_{E(MIN)} = -126 \qquad \text{// Zérus pontosabb ábrázolása miatt}$$

$$V_{E(MAX)} = 127$$

$$V_{FPN(MIN)} = 1.000\ldots_2 \times 2^{-126} = 1.1755 \times 10^{-38} \qquad \text{// 4 x DEC!}$$

$$V_{FPN(MAX)} = 1.111\ldots_2 \times 2^{+127} = 3.4028 \times 10^{38} \qquad \text{// 2 x DEC!}$$

$$NLM_{FPN} = 2^{23} = 8,388,608$$

$$NLE_{FPN} = 127 + \left|-126\right| + 1_{ZERO} = 254$$

$$NRV_{FPN} = 2^{23} \times (2^8 - 1) = 2.131 \times 10^9$$

- *DEC*: zérushoz közelítve szakadás (az első érték aránytalanul messze van)
- Ezt küszöböli ki az *IEEE* rendszer (Normalizálatlan tartományban lineárisan tart a zérushoz): ezért használja a V_E= 126

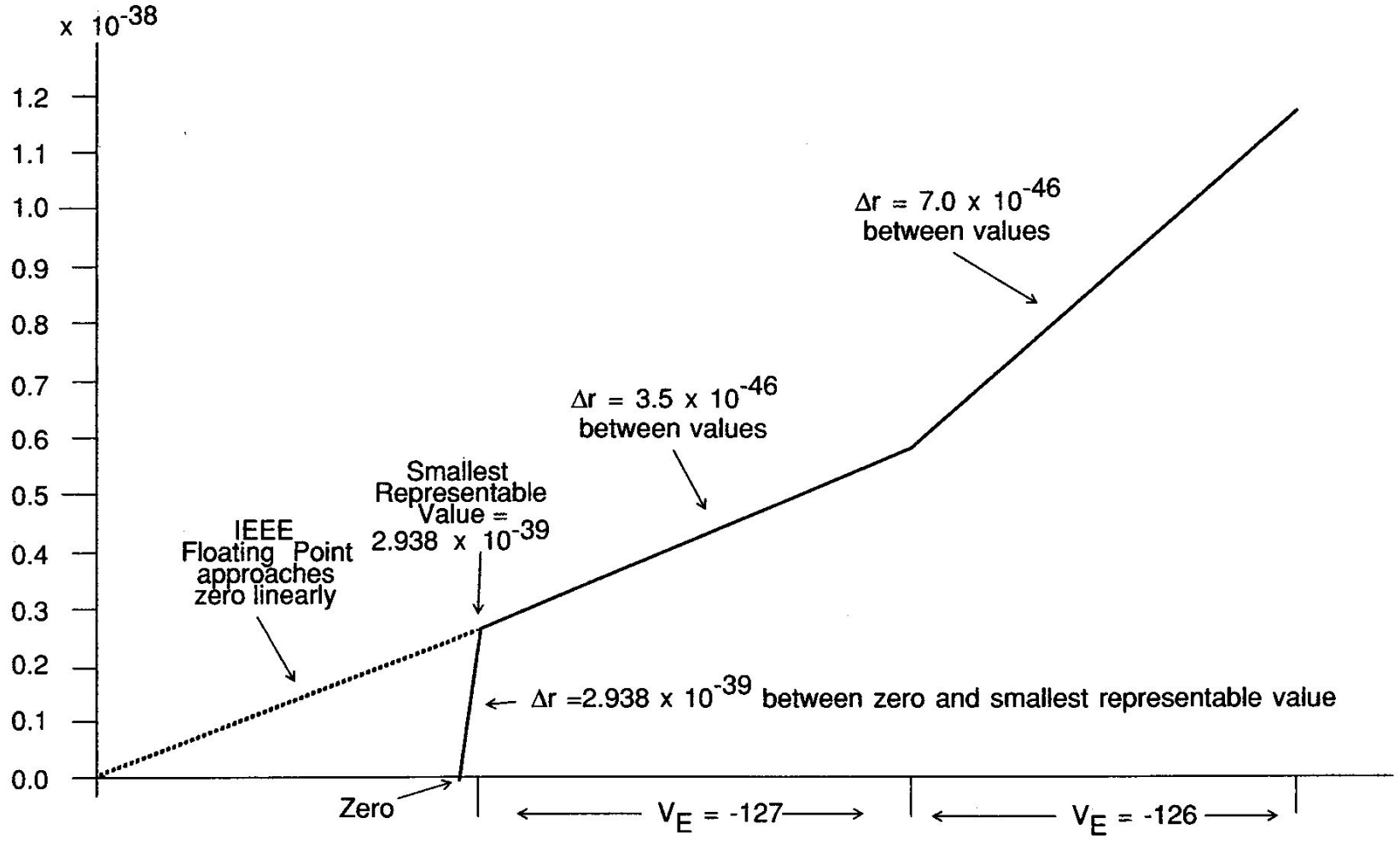


Figure 2.3. Values of the DEC Normalized Floating Point System Near Zero.

Példa: **IEEE-32** bites normalizált lebegőpontos rendszer (folyt.)

- V_E = [-127, 126] = [1, 254] eltolt Excess-127 tartomány
- Speciális jelentőség:
  - V_E = 0 értékénél (zérus ábrázolása)
  - V_E = 255 értékénél lehetőség van bizonyos információk tárolására:

V_E	Előjel	Ábrázolás jelentése
≠ 0	1,0	Nem egy szám (NaN)
0	0	+∞
0	1	-∞

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, soremelés (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

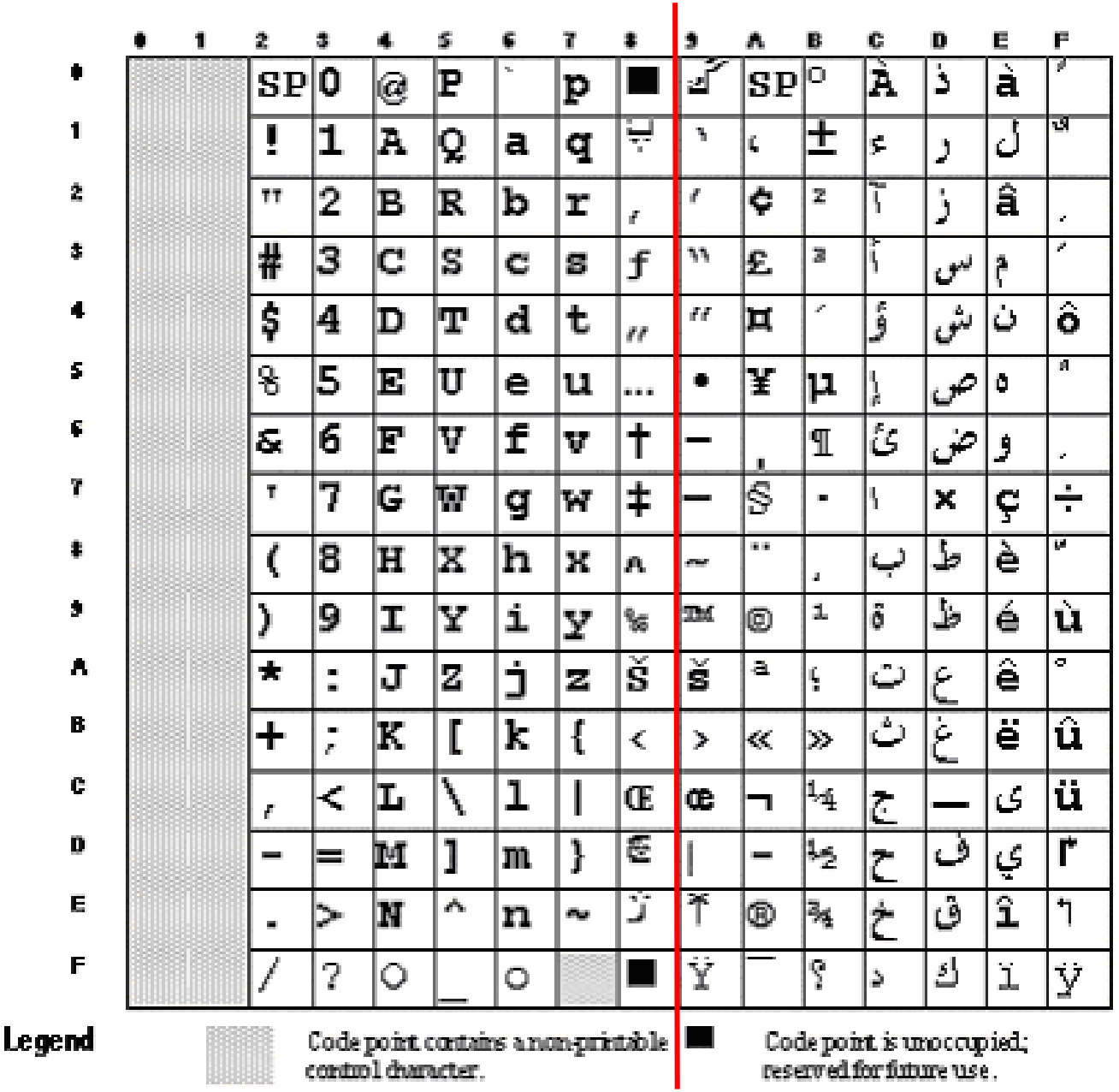
- BCD (Binary Coded Decimal): 6-biten
  - nagybetűk, számok, és speciális karakterek
- EBCDIC (Extended Binary Coded Decimal Interchange Code): 8-biten (A. Függelék)
  - + kisbetűs karaktereket és kiegészítő-információkat
  - 256 értékből nincs mindegyik kihasználva
  - Továbbá I és R betűknél szakadás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

EBCDIC

HEX DIGITS 1ST → 2ND ↓	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0	(sp) SP010000	& SM030000	- SP100000	ø LO810000	Ø LO820000	° SM190000	µ SM170000	¢ SC040000	{ SM110000	} SM140000	\ SM070000	0 ND100000
-1	(RSP) SP300000	é LE110000	/ SP120000	É LE120000	a LA810000	j LJ010000	~	£	A 000	J LJ020000	÷ SA060000	1 ND010000
-2	â LA150000	ê LE150000	Â LA160000	Ê LE160000	b LB010000	k LK010000			000	K LK020000	S LS020000	2 ND020000
-3	ä LA170000	ë LE170000	Ä LA180000	Ë LE180000	c LC010000	l LL010000			000	L LL020000	T LT020000	3 ND030000
-4	à LA130000	è LE130000	À LA140000	È LE140000	d LD010000	m LM010000			' 000	M LM020000	U LU020000	4 ND040000
-5	á LA110000	í LI110000	Á LA120000	Í LI120000	e LE010000	n LN010000	' LV010000	x SM240000	_ LE020000	N LN020000	V LV020000	5 ND050000
-6	ã LA190000	î LI150000	Ã LA200000	Î LI160000	f LF010000	o LO010000	w LW010000	¶ SM250000	F LF020000	O LO020000	W LW020000	6 ND060000
-7	å LA270000	ï LI170000	Å LA280000	Ï LI180000	g LG010000	p LP010000	x LX010000	Œ LO520000	G LG020000	P LP020000	X LX020000	7 ND070000
-8	ç LC410000	ì LI130000	Ç LC420000	Ì LI140000	h LH010000	q LQ010000	y LY010000	œ LO510000	H LH020000	Q LQ020000	Y LY020000	8 ND080000
-9	ñ LN190000	ß LS610000	Ñ LN200000	` SD130000	i LI010000	r LR010000	z LZ010000	Ÿ LY180000	I LI020000	R LR020000	Z LZ020000	9 ND090000
-A	ý LY120000	! SP020000	Š LS220000	: SP130000	« SP170000	ª SM210000	¡ SP030000	¬ SM660000	(SHY) SP320000	1 ND011000	2 ND021000	3 ND031000
-B	. SP110000	Š SC030000	, SP060000	# SM010000	» SP180000	º SM200000	¿ SP160000	š LS210000	ô LO150000	û LU150000	Ô LO160000	Û LU160000
-C	< SA200000	* SM040000	% SM020000	@ SM050000	ð LD830000	æ LA510000	Ð LD640000	¨ SD310000	ö LO170000	ü LU170000	Ö LO180000	Ü LU180000
-D	(SP080000	) SP070000	_ SP090000	' SP050000	ý LY110000	ž LZ210000	[SM080000	] SM090000	ò LO130000	ù LU130000	Ò LO140000	Ù LU140000
-E	+ SA210000	; SP140000	> SA050000	= SA040000	þ LT830000	Æ LA520000	Þ LT840000	Ž LZ220000	ó LO110000	ú LU110000	Ó LO120000	Ú LU120000
-F	| SM130000	^ SD150000	? SP150000	" SP040000	± SA000000	€ SC300000	® SM530000	× SA570000	õ LO190000	ÿ LY170000	Õ LO200000	EO



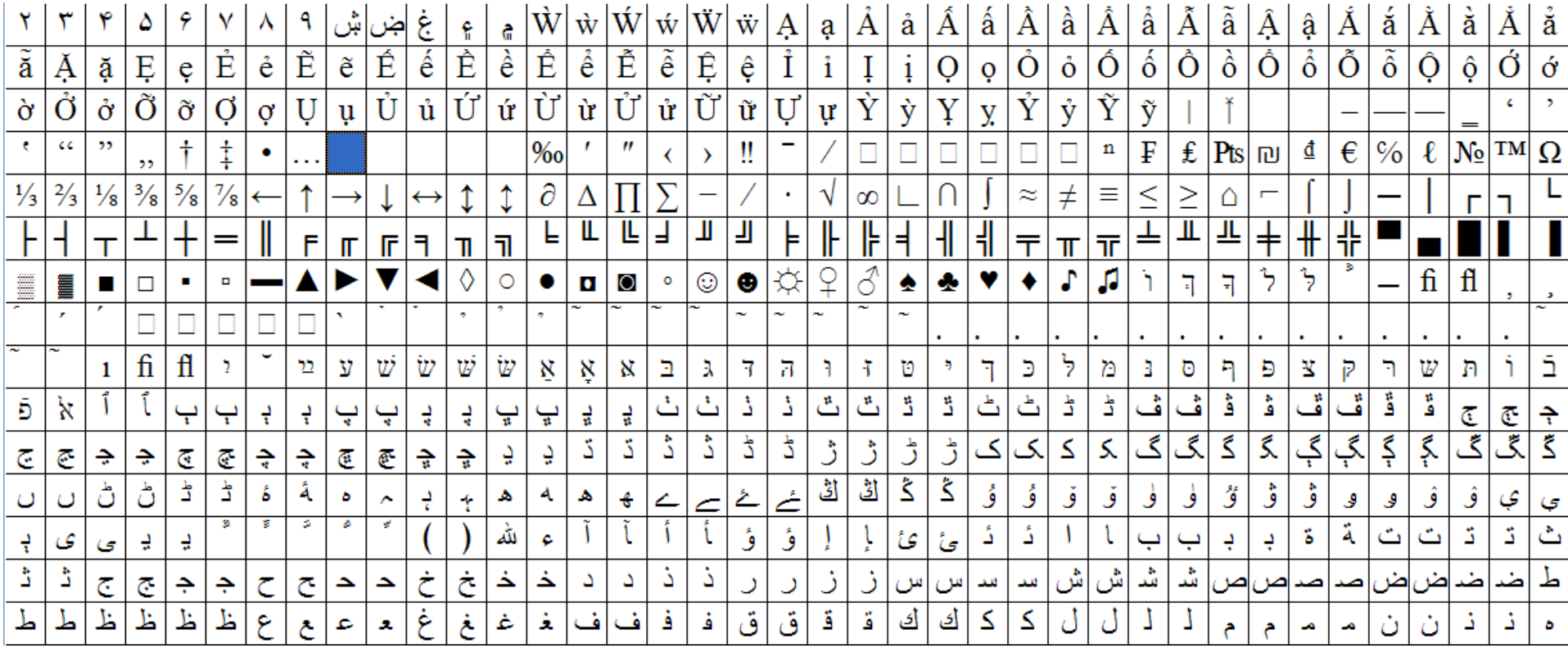
Extended ASCII (1 byte)



Unicode

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

Általános írásjelek, matematikai, pénzügyi, mértani szimbólumok stb.



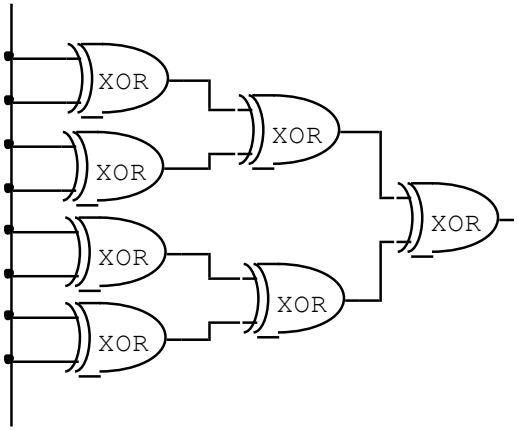
*részlet

Hibakódolás - Hibadetektálás és Javítás

- N bit segítségével 2^N különböző érték, cím, vagy utasítás ábrázolható
- 1 bittel növelve (N+1) esetén: $2^{\wedge N}$ -ről $2^{\wedge N+1}$ –re, megduplázódik
- Redundancia, detektálás, hibajavítás

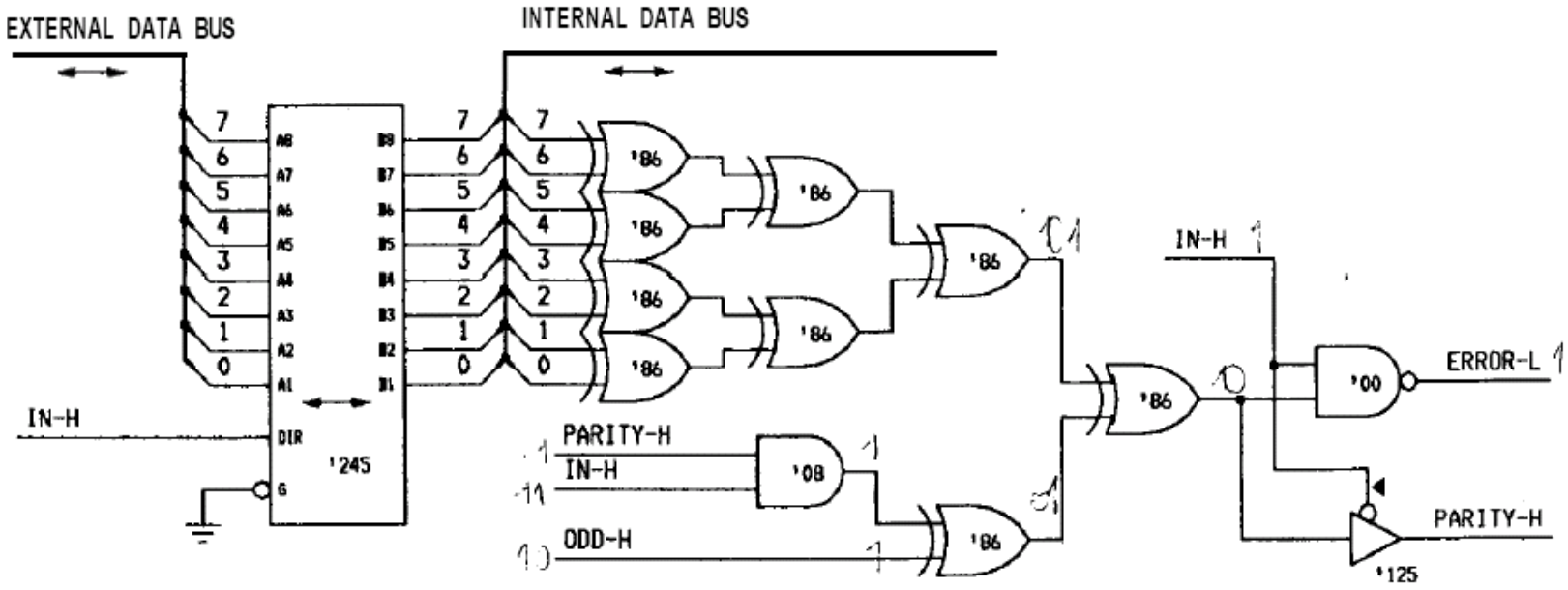
Paritás bit ellenőrzés

- Páros v. páratlan paritás: N bites információ egy kiegészítő bittel bővül → egyszeres hiba felismerése (bitek számát párosra v. páratlanra egészíti ki)
  - Pl: leragadásból: ‘0’-ból ‘1’-es lesz, vagy fordítva
  - Pl: ideiglenes, tranziens jellegű hiba
  - 8 adatbithez paritásbit generálás (XOR) (IC ‘280)



8 bites adatút kétirányú paritásbit ellenőrzéssel

- IN-H=1 → PARITY-H=1 paritás ell.
- Ha ODD-H=1 páratlan paritást használ
- Hiba esetén: ERROR-L=1

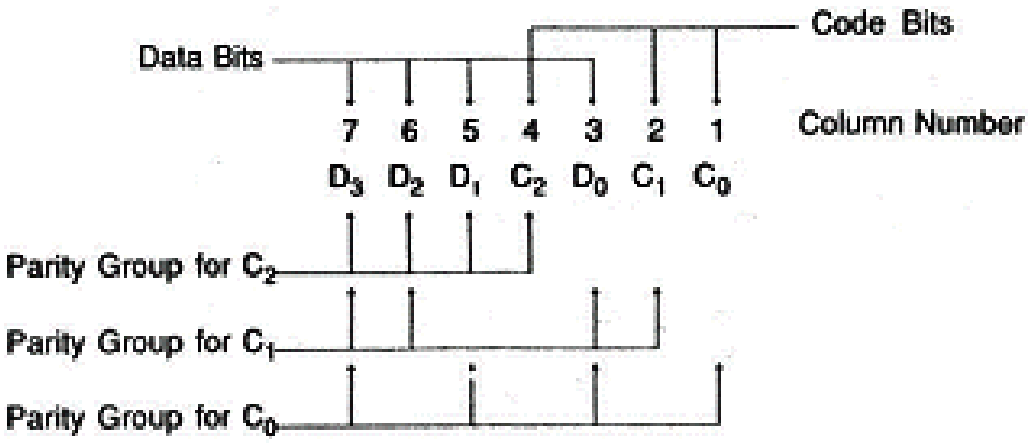


Hamming kód

- Több redundáns bittel nemcsak a hiba meglétét, és helyét tudjuk detektálni, hanem a hibás bitet javítani is tudjuk
- Hamming kód: egy biten tároljuk a bitmintázatok azonos helyiértékű bitjeinek különbségét, tehát egybites hibát lehet vele *javítani*.

Hamming kódú kódszó konstruálása (pl. 7 –bites kódszóra)

- **2^N-1 bites Hamming kód: N kódbit, 2^N-N-1 adatbit**
- Összesen 7 biten ***4*** *adatbitet (D0,D1,D2,D3),* ***3*** *kódbittel (C0,C1,C2)* kódolunk
- C_i kódbitek a bináris súlyuknak megfelelő bitpozíciókban
- A maradék pozíciókat rendre adatbitekkel töltjük fel (D_i)

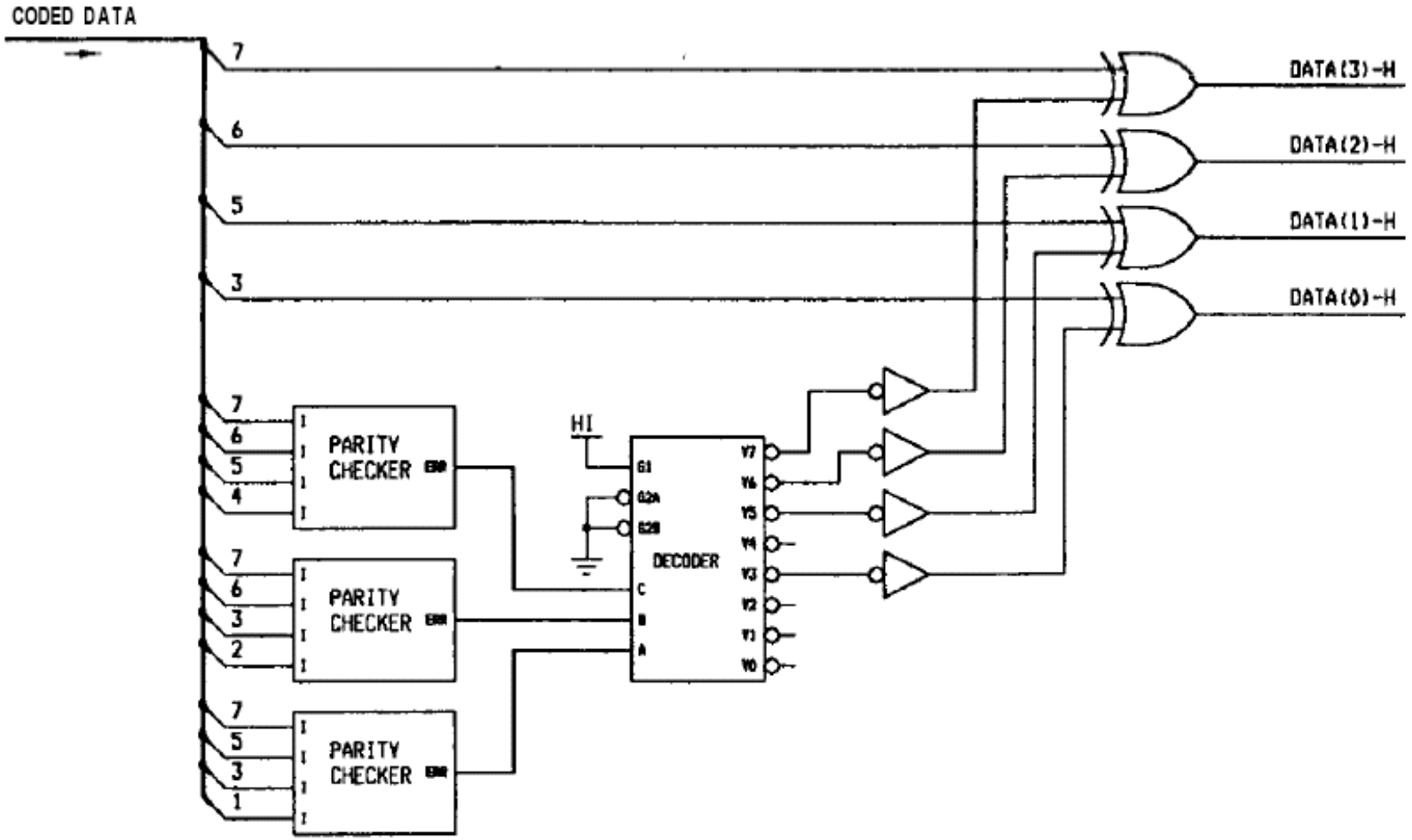


Paritás-csoportok	Bit pozíciók	Bitek jelölései
0	1, 3, 5, 7	C0, D0, D1, D3
1	2, 3, 6, 7	C1, D0, D2, D3
2	4, 5, 6, 7	C2, D1, D2, D3

Hamming kódú hibajavító áramkör tervezése

- 3 kódbitünk van, így írásnál 3 paritásbit generáló-, míg olvasásnál 3 paritás-ellenőrző áramkör kell.
- Példa: bemeneti adatbit-mintázatunk 0101.
  - Mivel páratlan paritást alkalmazunk, a megfelelő helyen szereplő kódbitekkel kiegészítve a következő szót kapjuk: 0100110. Ha nincs hiba, a paritásellenőrzők (C0, C1, C2) kimenete 000, minden egyes paritáscsoportra.
  - Hiba esetén például, ha az input mintázat 0100010, akkor a paritásellenőrző hibát észlel. Ugyan C2. paritásbitcsoport rendben, de a C1. és C0. hibás:
  - 011 az azonosított minta a harmadik oszlopban (**D0** helyén).
  - Javításként *invertáljuk* a 3. bitpozícióban lévő bitet. 0100010 $\Rightarrow$ 0100110. Ekkor a kódbitek a következőképpen módosulnak a páratlan paritásnak megfelelően: C0=1, C1=1 és C2=0.

7-bites Hamming kódú hibajavító áramkör felépítése



Példa:

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	1	0	1	0	0	0	Hozzáadott

kódbitek. Tehát a helyes ábrázolása 01011100-nek a következő:
1010111101000.

Pannon Egyetem
Képfeldolgozás és Neuroszámítógépek Tanszék



Digitális Rendszerek és Számítógép Architektúrák

4. előadás: Aritmetikai egységek - adatkezelés

Előadó: Dr. Szolgay Péter
Vörösházi Zsolt

Jegyzetek, segédanyagok:

- Könyvfejezetek:
  - http://www.knt.vein.hu
    → Oktatás → Tantárgyak → Digitális Rendszerek és Számítógép Architektúrák (Nappali)
    (chapter03.pdf)
- Fóliák, óravázlatok .ppt (.pdf)
- Feltöltésük folyamatosan

Ismétlés

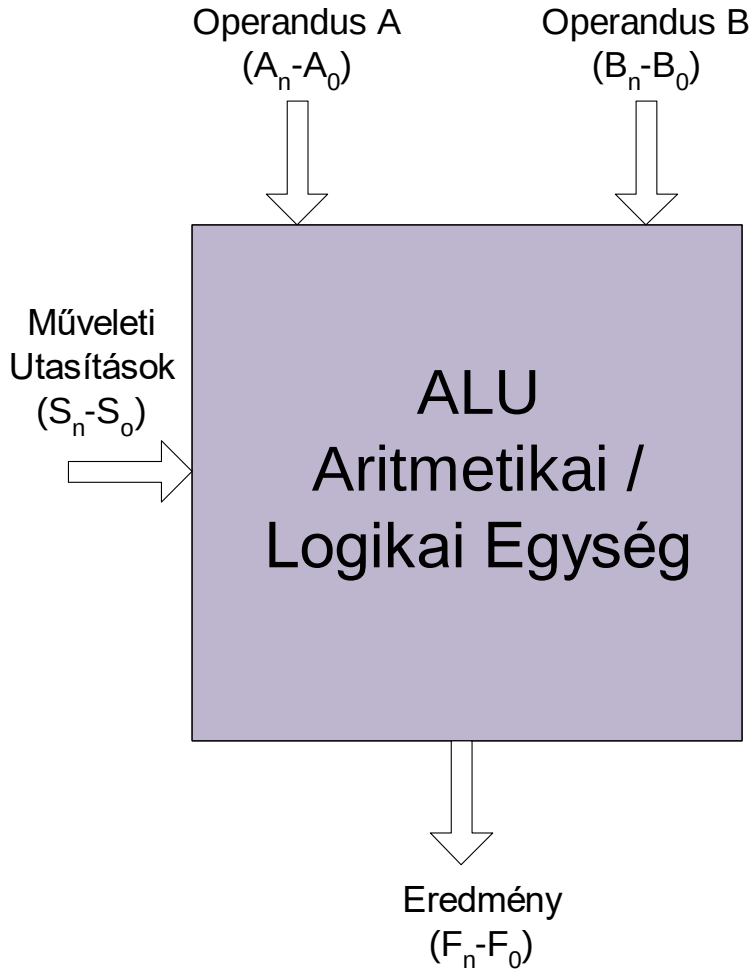
- Korai számítógépek teljesítményét főként ballisztikus számításoknál (hadászatban)
- Információ ábrázolás
- Használt utasításkészlet
- Adatkezelő / műveletvégző egység:
  - Alapvető ALU (Aritmetikai és Logikai funkciók)
    - Aritmetikai operátorok: +,-,*,/ (alapműveletek)
    - Logikai operátorok: NOT,AND,OR,NAND,NOR,XOR (AV),NXOR (EQ)

Tervezői célkitűzés

- Komplex funkció megvalósítása minimális kapu felhasználásával
  - Minimális késleltetés legyen az adat-úton
- Univerzálisan teljes leírás (K.H.):
  - NAND illetve,
  - NOR kapuk segítségével minden komplex logikai függvény felírható!
- Aritmetika alapvető építőeleme: összeadó
  - belőle a többi elem (-, *, /) származtatható!

ALU felépítése

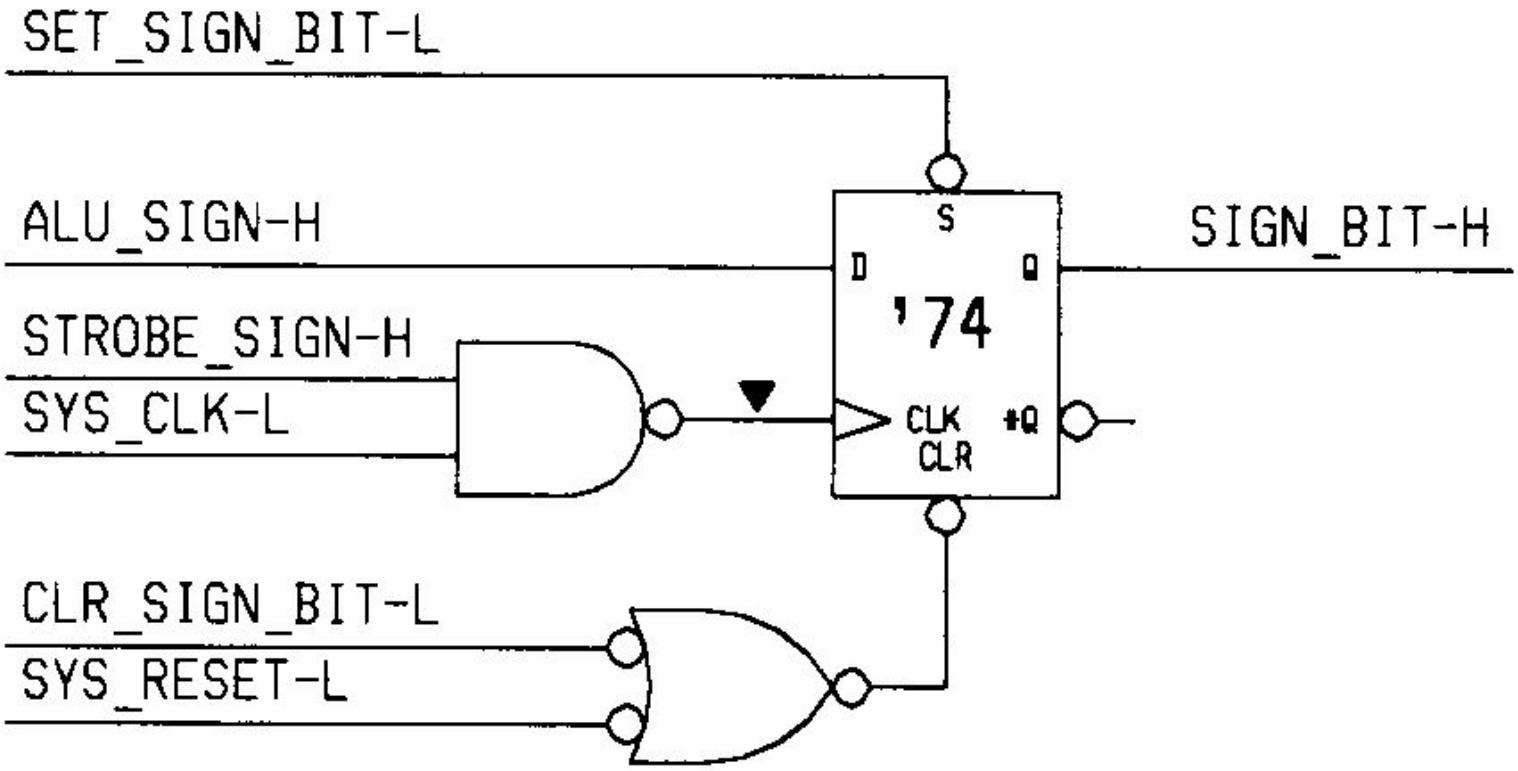
- Utasítások hatására a (S_n-S_0) vezérlőjelek kijelölik a végrehajtandó aritmetikai / logikai műveletet. További adatvonalak kapcsolódhatnak közvetlenül a státusz regiszterhez, amely fontos információkat tárol el: pl.
  - *zero bit*
  - *carry-in, carry-out* átviteleket,
  - *előjel* bitet (sign),
  - *túlcsordulást* (overflow), vagy *alulcsordulást* (underflow) jelző biteket.



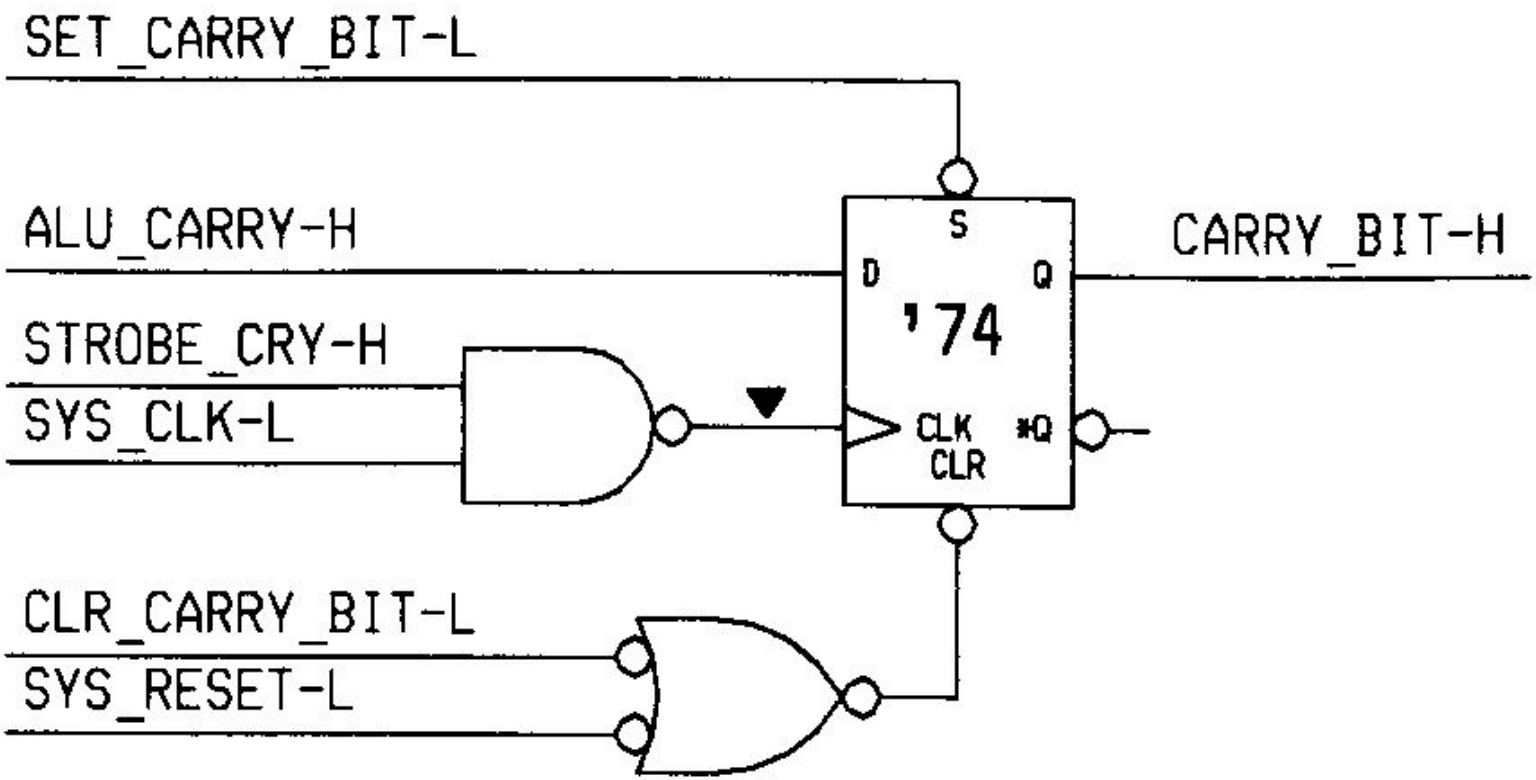
Státusz- (flag) jelzőbitek

- Az aritmetikai műveletek eredményétől függően hibajelzésre használatos jelzőbitek. Ezek megváltozása az utasításkészletben előre definiált utasítások végrehajtásától függ.
  - a.) Előjelbit (sign): 2’s komplemens (MSB)
  - b.) Átvitel kezelő bit (carry in/out): helyiértékes átvitel
  - c.) Alul / Túlcsordulás jelzőbit (underflow / overflow)
  - d.) Zero bit: kimeneten az eredmény 0-e?

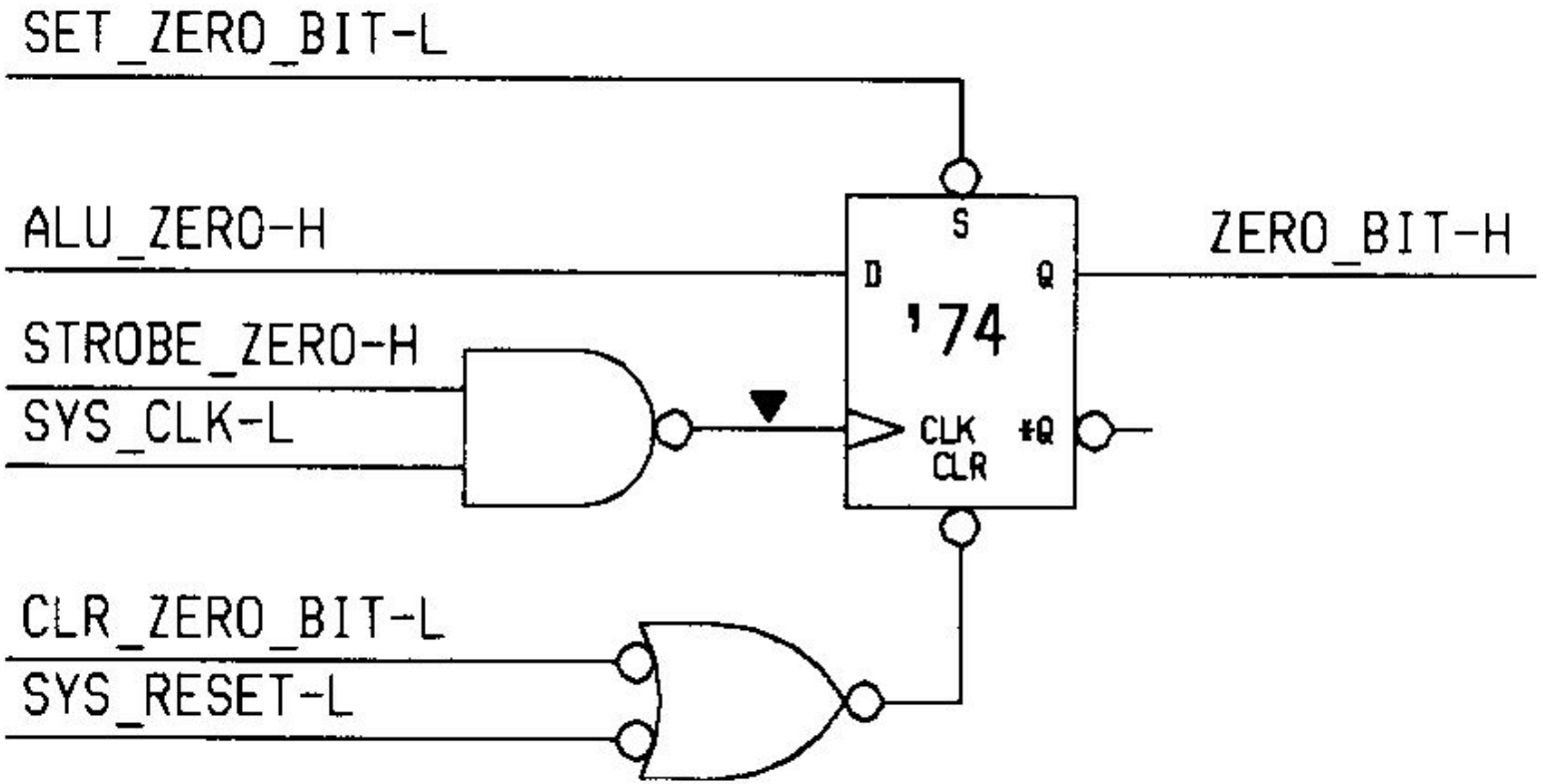
a.) Előjelbit (sign)



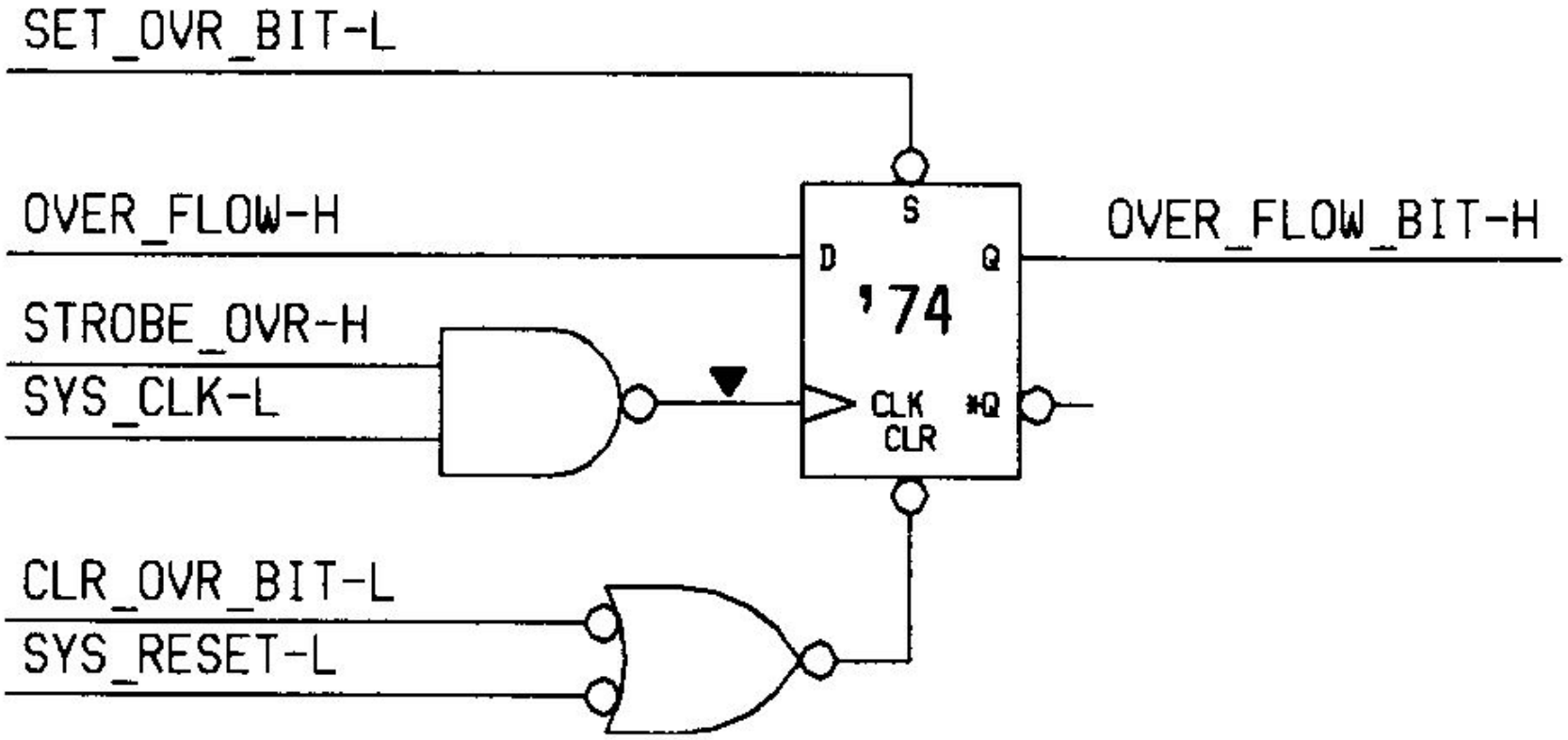
b.) Átvitel-kezelő bit (carry)



c.) Zéró bit

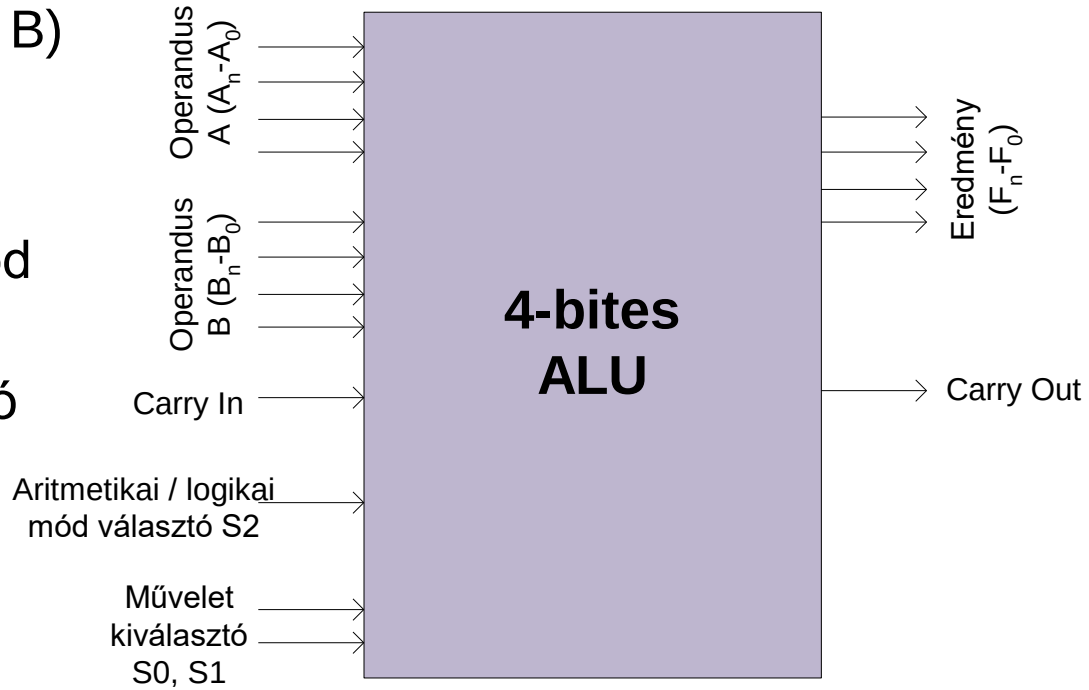

SET_ZERO_BIT-L
ALU_ZERO-H
STROBE_ZERO-H
SYS_CLK-L
CLR_ZERO_BIT-L
SYS_RESET-L
S
D
Q
'74
CLK
CLR
*Q
ZERO_BIT-H

d.) Túlcsordulást jelző bit (overflow)



Pl: 4-bites ALU felépítése és működése

- Két 4-bites operandus (A, B)
- 4 bites eredmény (F)
- Átvitel: Carry In/ Out
- S2: Aritmetikai/ logikai mód választó (MUX)
- S0, S1: művelet kiválasztó (S2 értékétől függően)



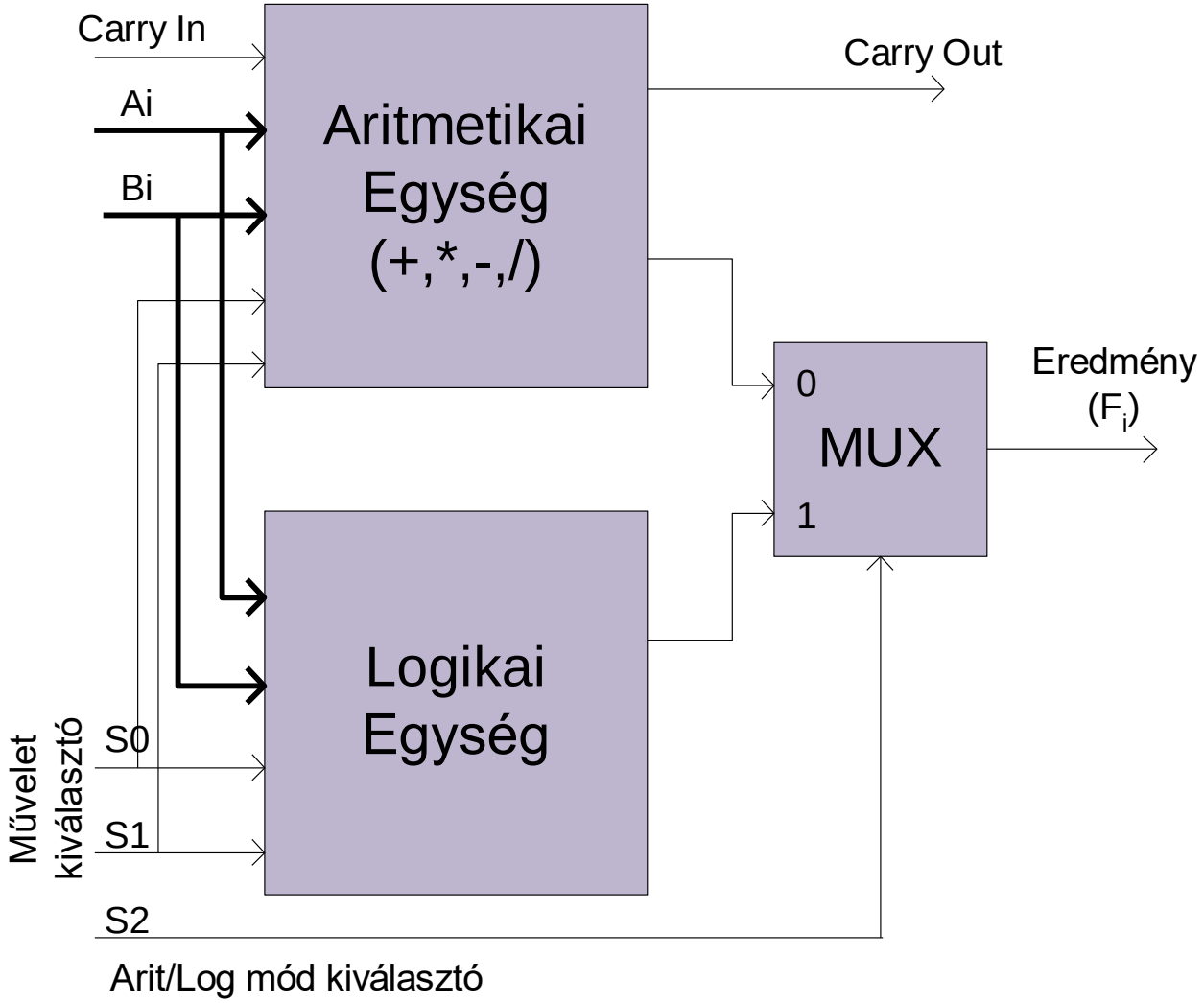
4-bites ALU szimbolikus rajza

ALU működését leíró függvénytáblázat:

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	$F = A + \overline{B}$	A + 1’s komplemens B
0	1	0	1	$F = A + \overline{B} + 1$	Kivonás
0	1	1	0	F=A-1	‘A’ értékének csökkentése 1-el (decrement)
0	1	1	1	F=A	‘A’ átvitele
1	0	0	0	$F = A \wedge B$	AND
1	0	1	0	$F = A \vee B$	OR
1	1	0	0	$F = A \oplus B$	XOR
1	1	1	0	$F = \overline{A}$	‘A’ negáltja (NOT A)

* Könyv függeléke (appendix)

ALU felépítése:



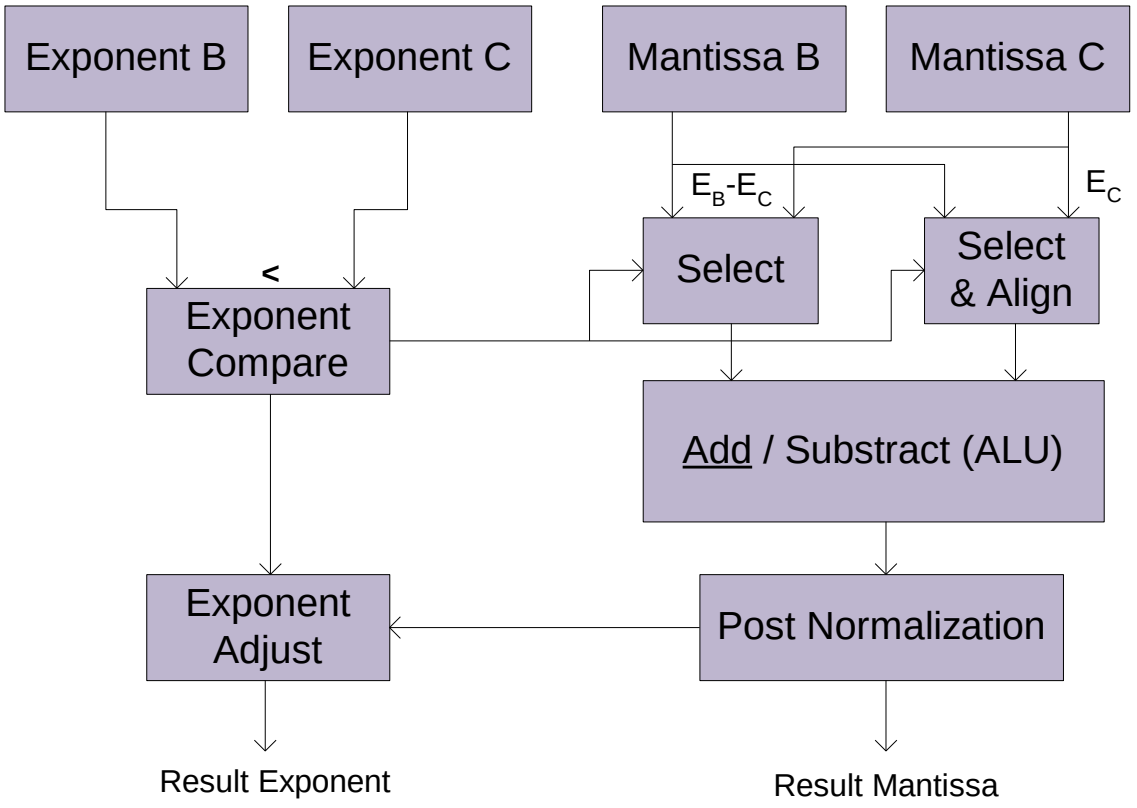
Lebegőpontos műveletvégző egységek

Lebegőpontos műveletvégző egységek

- Probléma:
  - Mantissza igazítás → Exponens beállítás
  - Normalizálás (DEC-32, IEEE-32, IBM-32)
- Műveletvégző elemek:
  - Összeadó-,
  - Kivonó-,
  - Szorzó-,
  - Osztó áramkörök.

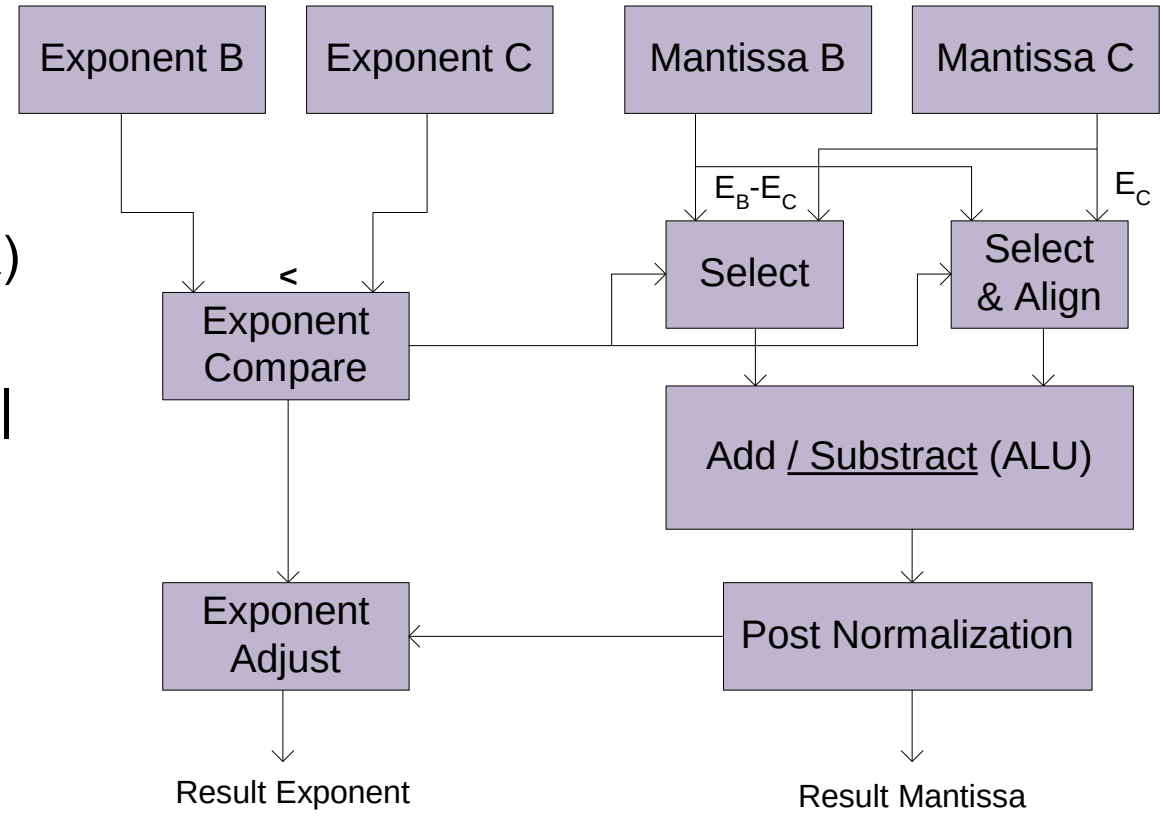
a.) Lebegőpontos összeadó

- Művelet: $A = M_B \times r^{E_B} + M_C \times r^{E_C} = (M_B \times r^{E_B - E_C} + M_C) \times r^{E_C}$
  - Komplex feladat: a mantisszák hosszát egyeztetni kell (MSB bitek azonos helyiértéken legyenek)
  - Legyen: 0<B<C
  - B → C vagyis |EB–EC| vel jobbra igazítjuk a mantisszát; ez változás az exponensben is
  - Összeadás: sign-magnitude formátum!
  - Végül minimális post-normalizáció kell



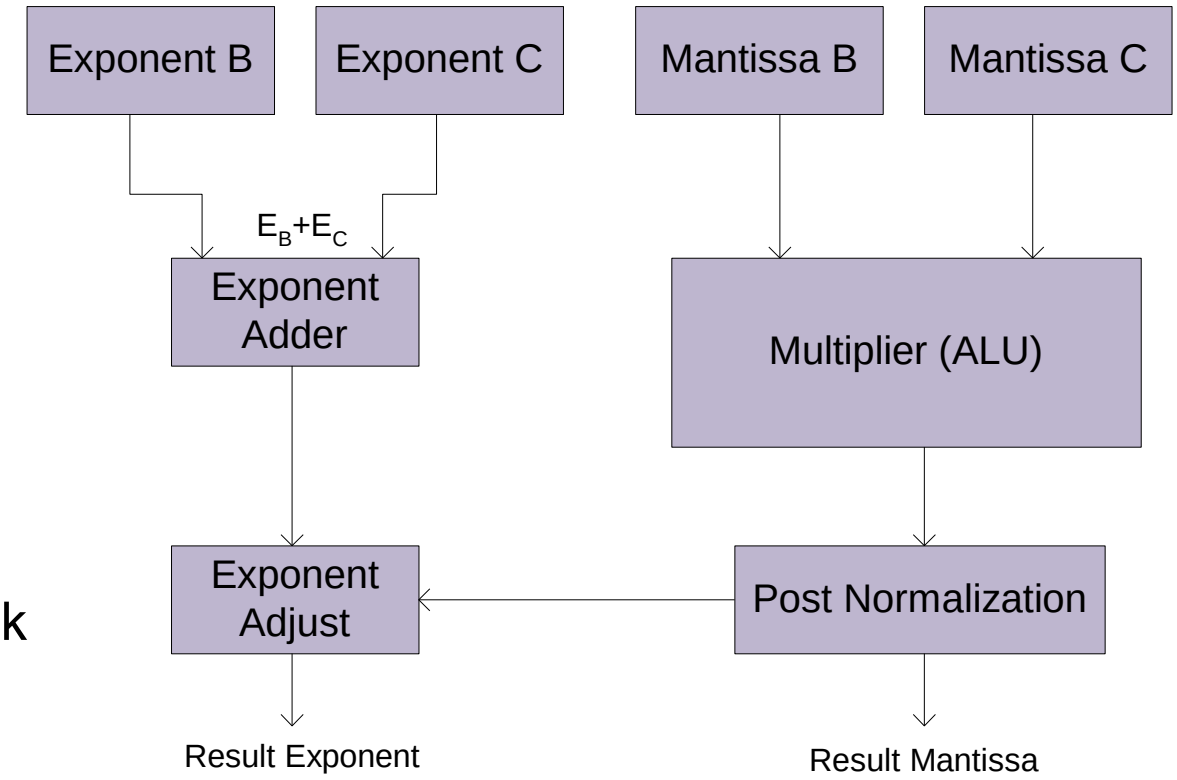
b.) Lebegőpontos kivonó

- Művelet: $A = M_B \times r^{E_B} - M_C \times r^{E_C} = (M_B \times r^{|E_B - E_C|} - M_C) \times r^{E_C}$
  - Komplex feladat: a mantisszák hosszát egyeztetni kell (MSB bitek azonos helyiértéken legyenek)
  - Legyen: 0<B<C
  - B → C vagyis |EB-EC| vel jobbra igazítjuk a mantisszát, ez változás az exponensben is
  - Kivonás! (ALU)



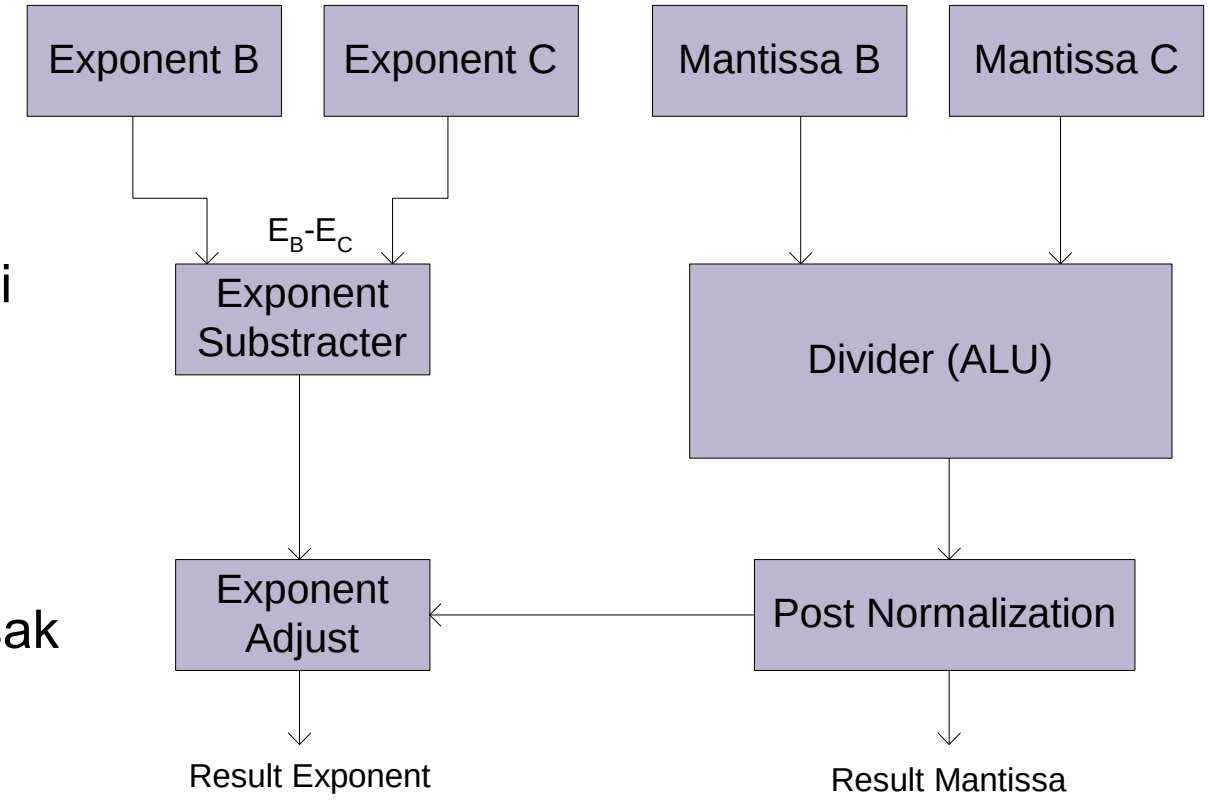
c.) Lebegőpontos szorzó

- 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



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 / Kivonó áramkörök

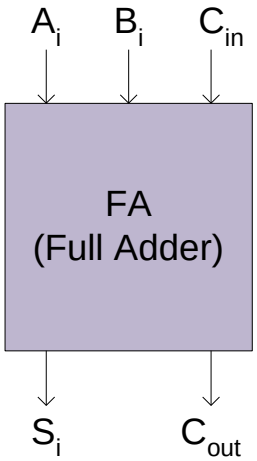
a.) Teljes összeadó – Full Adder

- FA: 1-bites Full Adder

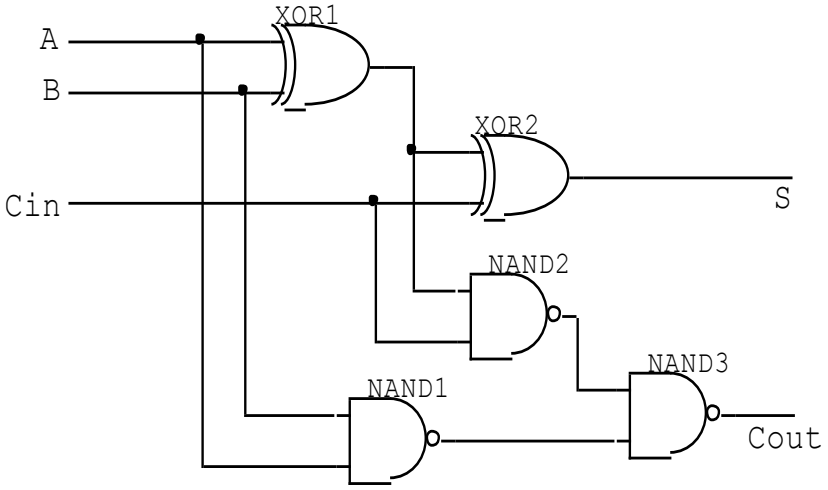
igazságtáblázat

A_i	B_i	Cin	Sum_i	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

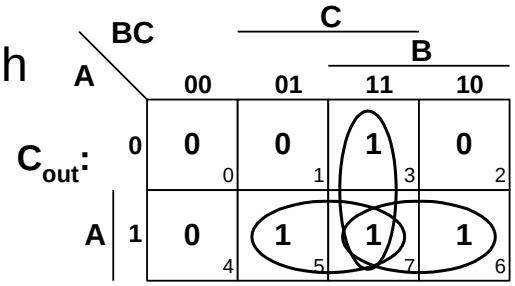
szimbólum

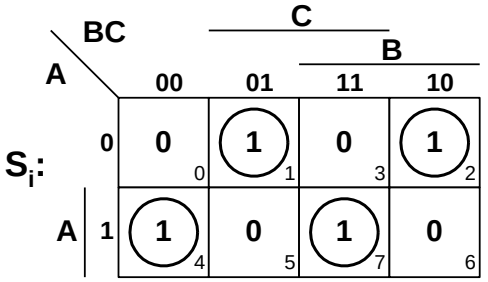


A FA egy CMOS kapcsolási rajza:



Karnaugh táblái:





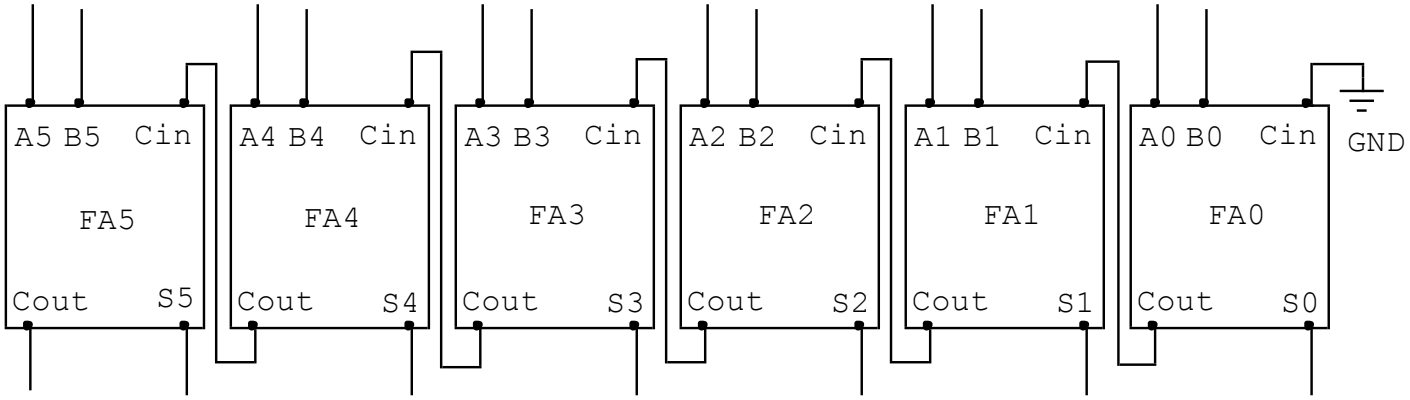
Kimeneti fgv-ei:

$$C_{out} = A \cdot B + A \cdot C_{in} + B \cdot C_{in}$$

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

b.) Átvitelkezelő összeadó – Ripple Carry Adder (RCA)

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



- Számítási időszükséglet (RCA):

$T_{(RCA)} = N*T_{(FA)} = N*(2*G) = 12\ G$ (pl. 6-bites RCA esetén)

ahol a 2G az 1-bites FA kapukésleltetése. Lassú carry terjedés.

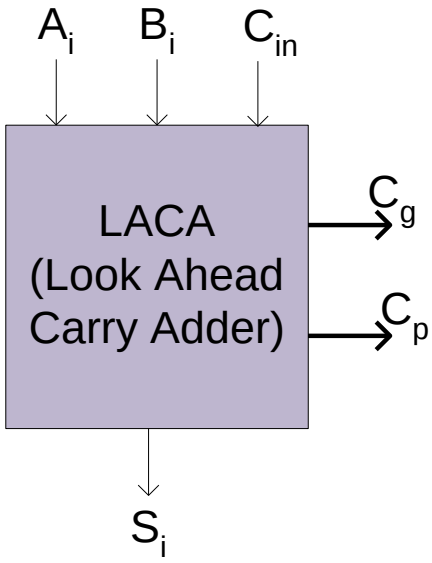
c.) LACA: Look Ahead Carry Adder:

- Képlet (FA) átírásából kapjuk:

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

$$\Rightarrow \underbrace{A_i \cdot B_i}_{CarryGenerate} + C_{in} \cdot \underbrace{(A_i + B_i)}_{Carry\,Propagate} = C_G + C_{in} \cdot C_P$$

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



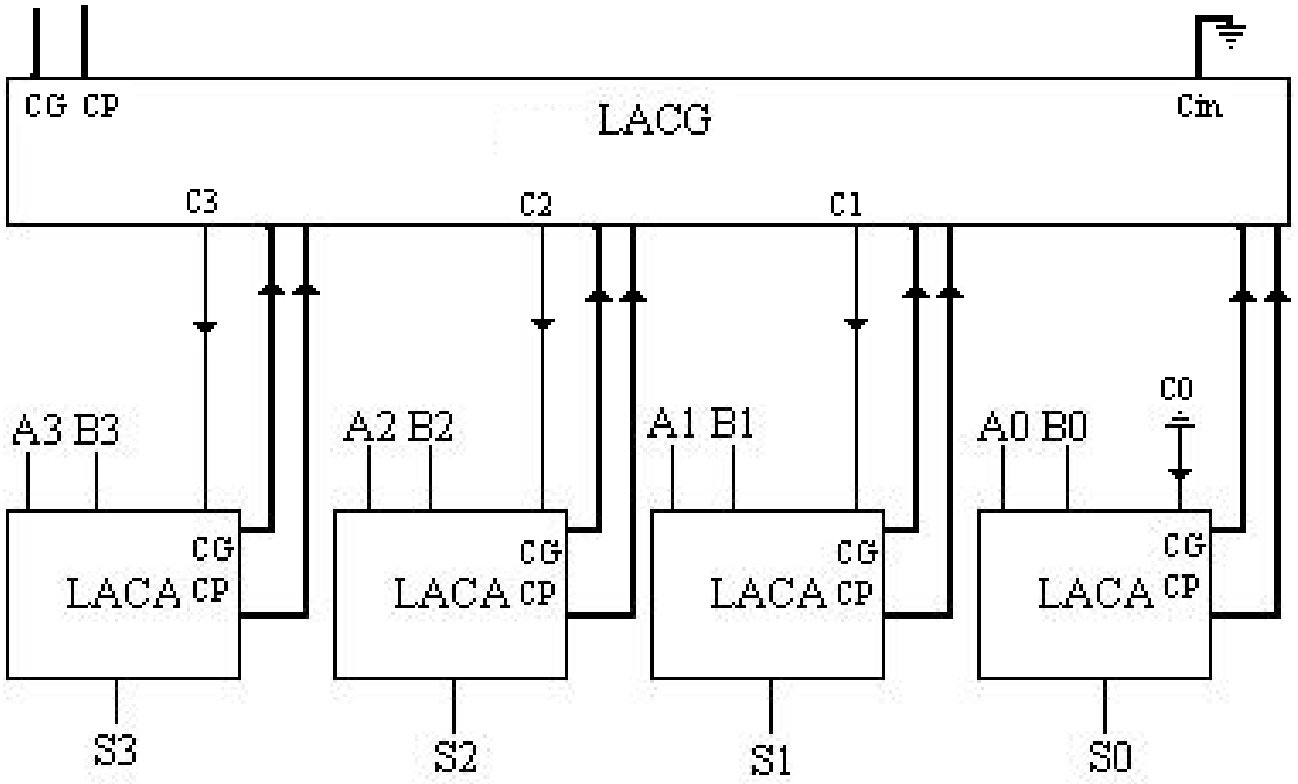
LACG: Look Ahead Carry Generator áll egy **b** bites ALU-ból, mindenegyes állapotban a Carry generálásáért felel a CP és CG (LACA) vonalakon érkező jeleknek megfelelően.

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

ahol N: bitek száma, b: LACG bitszélessége (hány LACA-ból áll egy LACG)

Példa: 4-bites LACA

- Legyen b=4, és N=4. Áramkör felépítése, és időszükséglete?



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

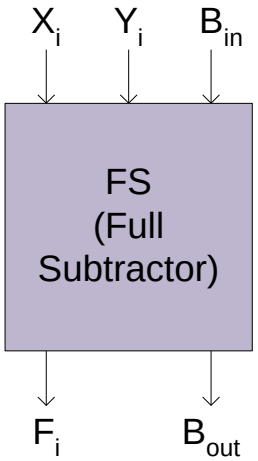
d.) Teljes kivonó - Full Subtractor (FS)

- FS: 1-bites Full Subtractor

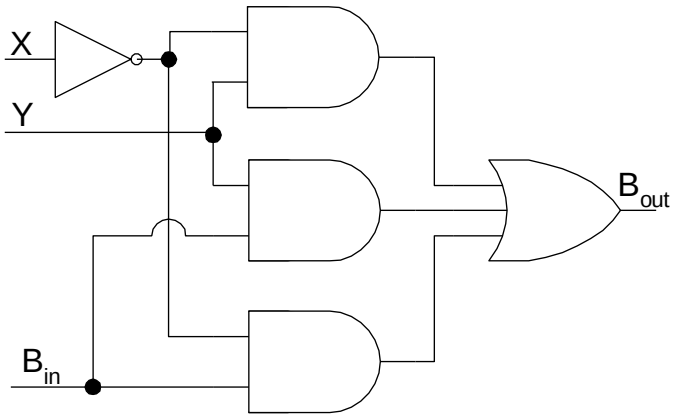
igazságtáblázat

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

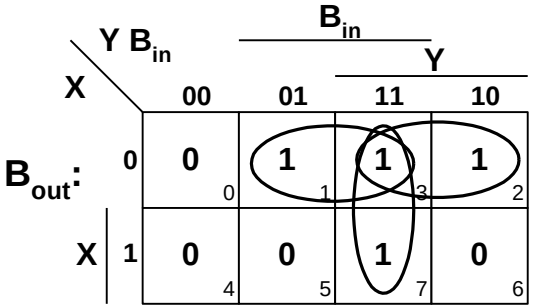
szimbólum

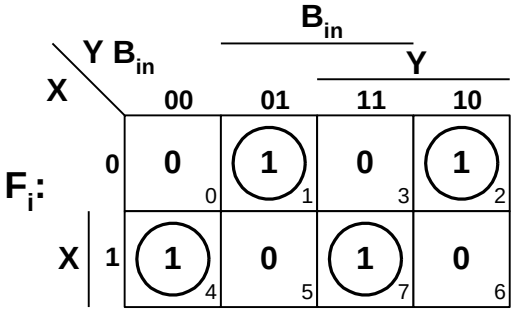


Logikai kapcsolási rajz (B out-ra)



Karnaugh táblái:





Kimeneti fgv-ei:

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

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

Szorzó áramkörök

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

I.) Iteratív szorzási módszerek

Iteratív szorzási módszerek alapjai

- Tényezők:
  - P: szorzat, A:szorzandó, B:szorzó

$$P = A \times B$$

  - Pl: Legyenek:
    - ‘A’ és ‘B’ 5-bites számok (0…2^5-1)=0…31
    - Maximálisan P=31*31=961 lehet (10 bites)
- Tehát: N bites számok szorzatát 2*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 (LSB→MSB)

					A4	A3	A2	A1	A0
				x	B4	B3	B2	B1	B0
PP0					A4*B0	A3*B0	A2*B0	A1*B0	A0*B0
PP1				A4*B1	A3*B1	A2*B1	A1*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							

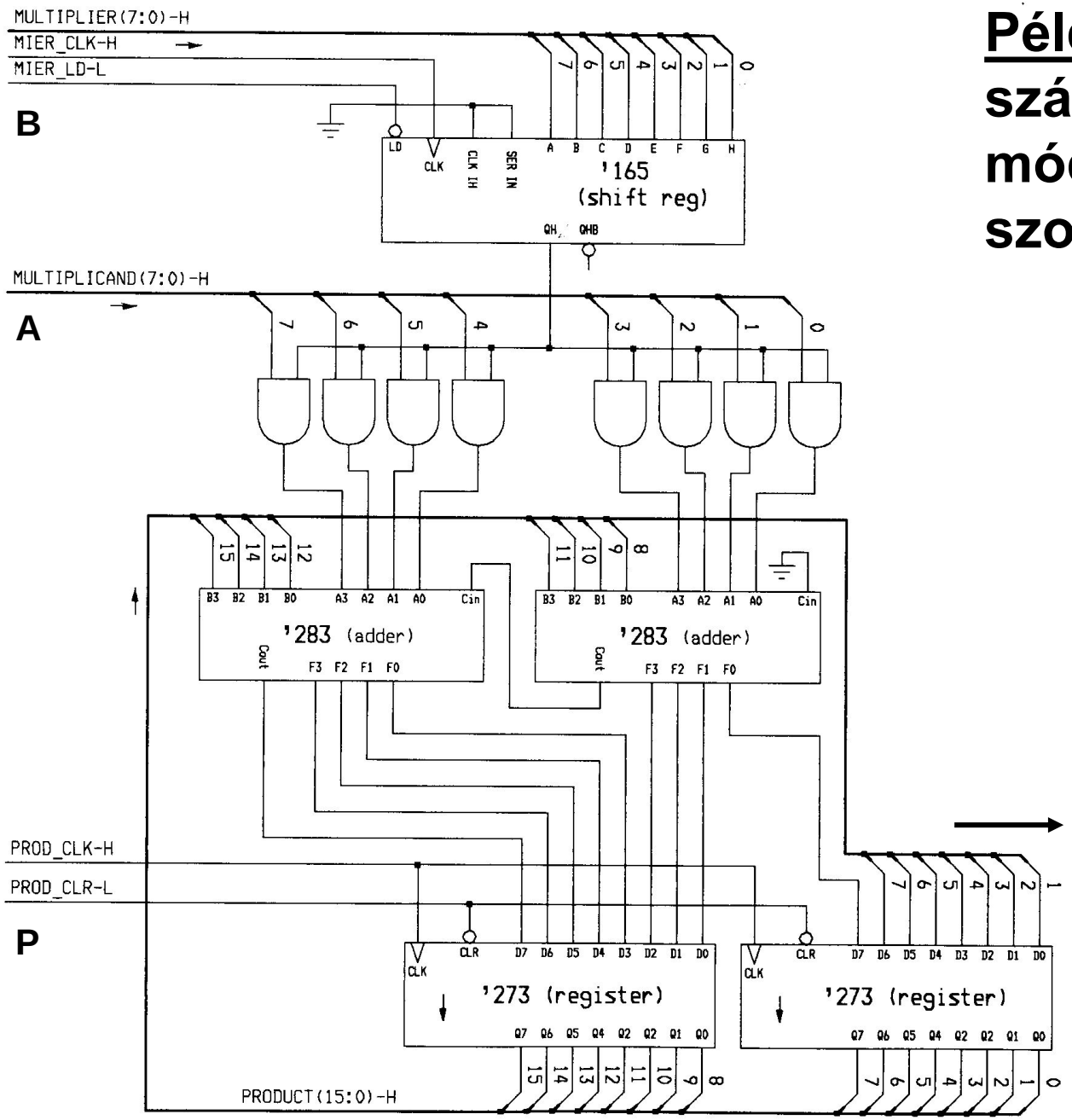
- Fordított sorrendű (MSB → LSB):

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

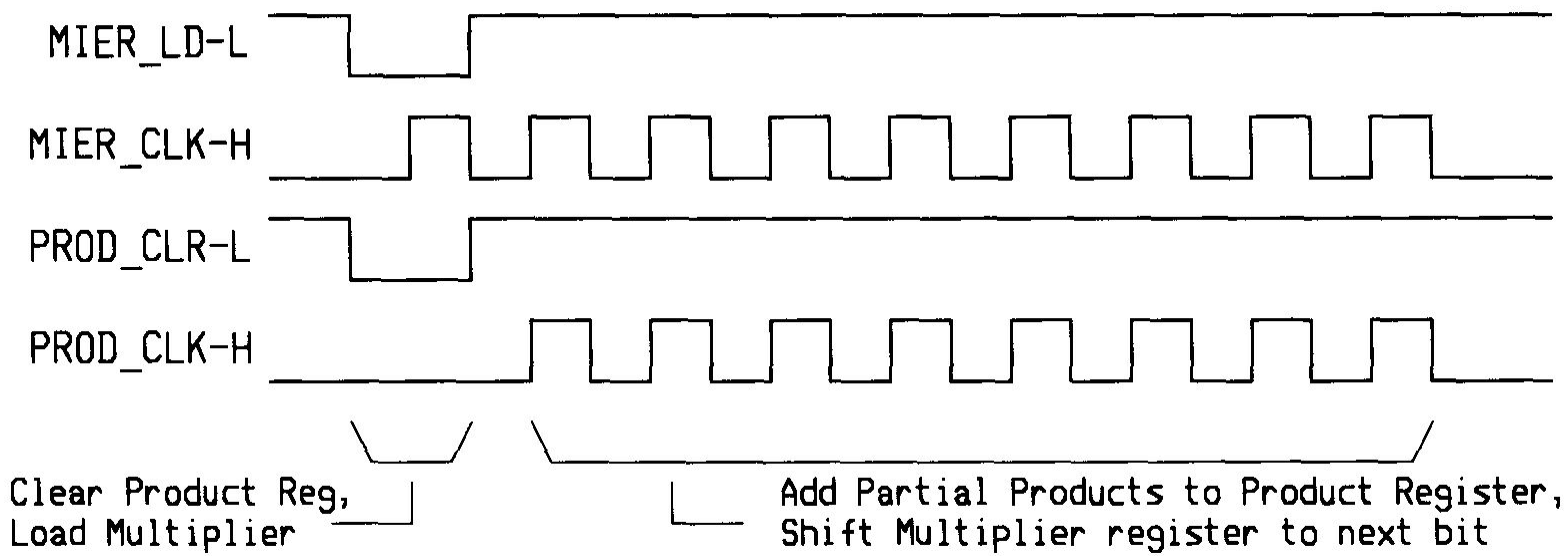
1.) Általános Shift&Add módszer

- P=A×B (A:szorzandó, B:szorzó)
- Parciális szorzat (PPi) összegeket az LSB → MSB bitek felől képzi (mivel a B szorzat biteket is ebben a sorrendben tölti be)
- AND kapuk: PPi-k képzése
- Shiftelés: *Huzalozott eltolással* (a visszacsatolt ágban)

Példa: két 8-bites szám Shift&Add módszerű szorzására



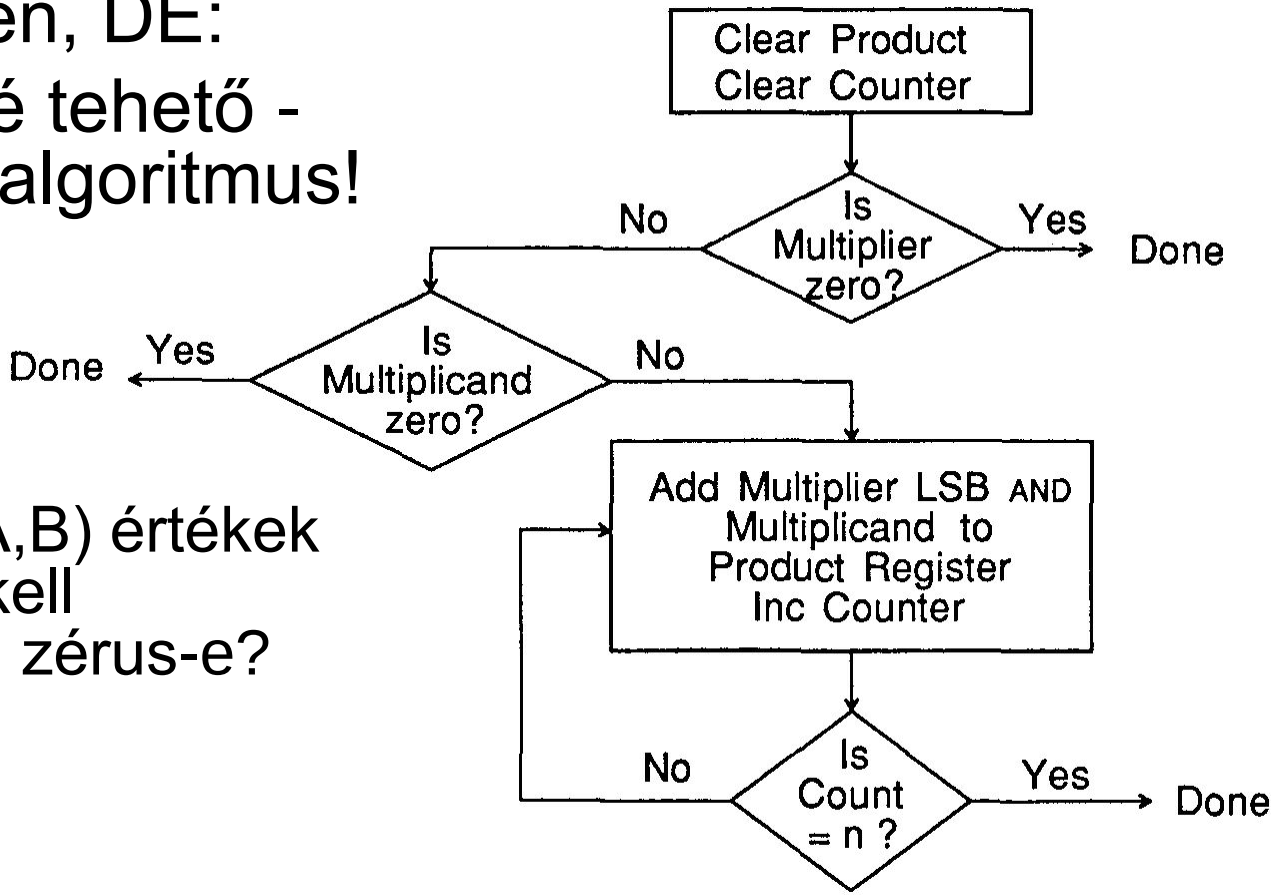
Időzítési jelek:



- A MIER_CLK-H (B): magas-aktív órajel vezérli a bemeneti 165'-ös SHIFT (paralel in- serial out) regisztert
- MIER_LD-L: load jel hatására, képes az összes bemenetére érkező jelet egy lépésben betölti
- A PROD_CLK-H: magas-aktív órajel (273' D tárolókból álló regiszternél)
- PROD_CLR-L: törlőjel, amely hozzáadás előtt törli a 273'regiszterek tartalmát

Folyamatábra (Shift-add)

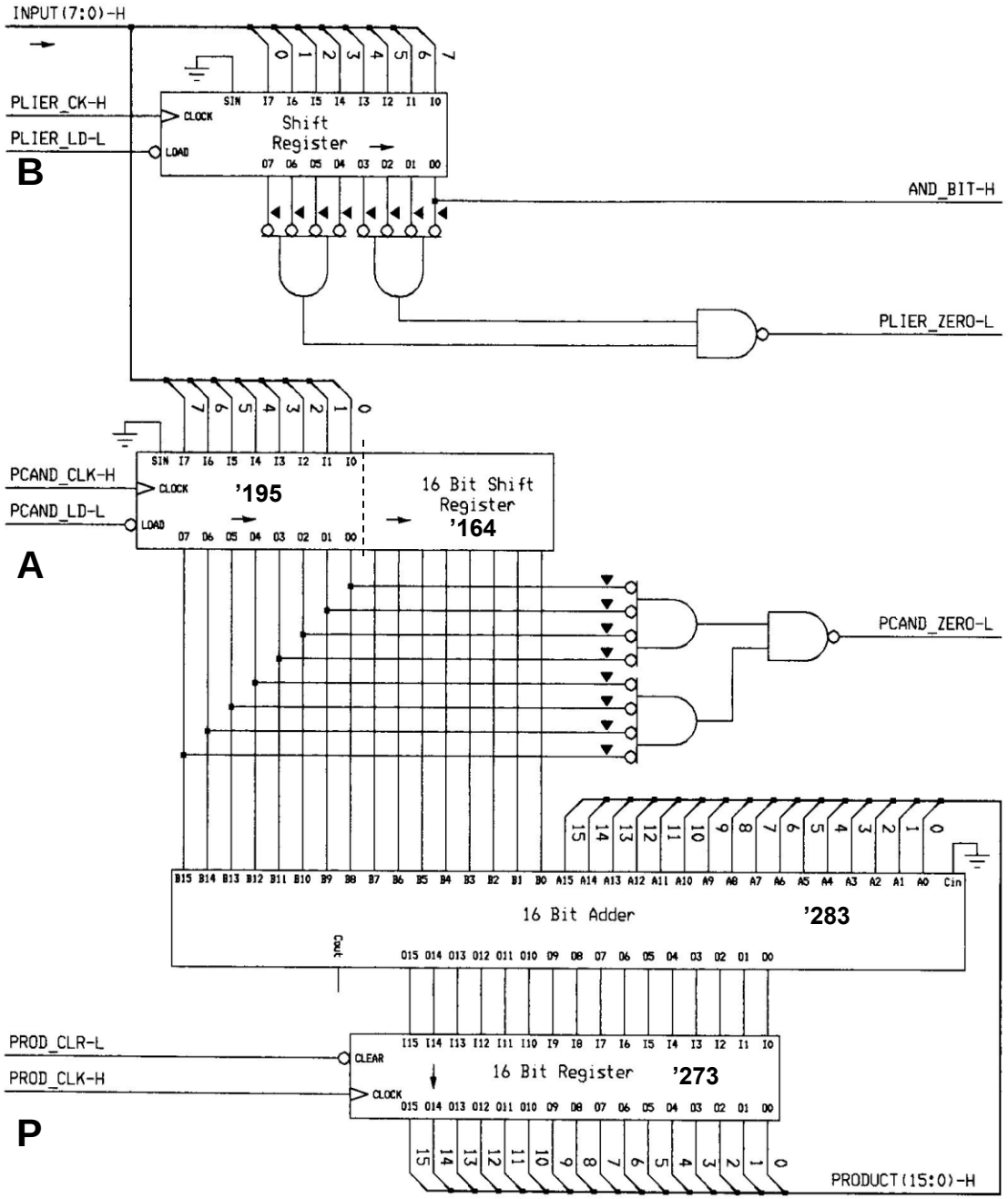
- Alapvetően adatfüggetlen, DE:
- Adatfüggővé tehető - gyorsítható algoritmus!
  - Bemenő (A,B) értékek figyelését kell megoldani: zérus-e?



2.) Fordított sorrendű módszer

- P=A×B (A:szorzandó, B:szorzó)
- Parciális szorzat (PP_i) összegeket itt fordított sorrendben, az MSB → LSB bitek felé haladva képzi (a B szorzat biteket fordított sorrendben tölti be)
- Bemenetek figyelése: ha a szorzandó, vagy szorzó bitek értéke zérus, leegyszerűsödik a művelet. (AND kapukkal)

Példa: két 8-bites szám Fordított sorrendű szorzására



Az AND_BIT_H jel ellenőrzi, hogy az „A” szorzandó regiszter hozzáadható-e a „B” szorzóregiszterhez, és az eredmény betölthető-e a „P” szorzatregiszterbe.

Ha az MSB=’1’, akkor betehető, ha ‘0’ akkor a szorzó és szorzandóregiszter 1 bitpozícióval shift-elődik.

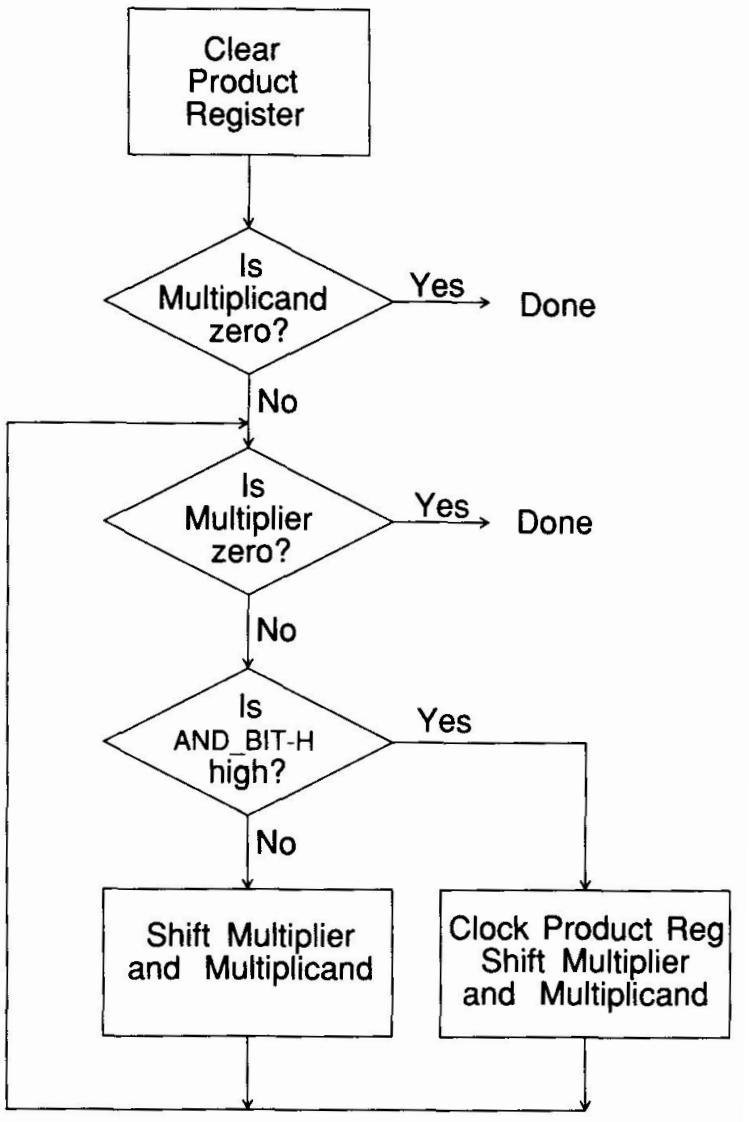
Folyamatábra (fordított sorrendű)

- Adatfüggő algoritmus - gyorsított végrehajtás
  - Bemenő (A,B) értékeket figyeli, hogy zérus-e?
- Időszükséglet:

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

$$T_{Iter} = T_{AND} + T_{Sum} + T_{\mathrm{Re}\,g}$$

•T(SETUP): kezdeti ellenőrzések, inicializálás ill. szorzat regiszter törlése
•T(AND): AND függvények végrehajtása, parciális szorzatok képzése
•T(SUM): parciális szorzatok összeadása
•T(REG): betöltésük a regiszterbe



c.) Előjeles szorzás Booth-algoritmussal:

- *Negatív számokkal* is lehet szorzást végezni!
  - Legyen a következő B 2's komplemens 6-bites szám
  - $B = B_5\ B_4\ B_3\ B_2\ B_1\ B_0 = B_5\times(-2^5) + B_4\times2^4 + B_3\times2^3 + B_2\times2^2 + B_1\times2^1 + B_0\times1$.
  - **<u>Újrakódolási technika (séma):</u>**

    $B(2'\text{ komplemens}) = B_5\times(-32) + B_4\times16 + B_3\times8 + B_2\times4 + B_1\times2 + B_0\times1 =$ TRÜKK!!

    $= B_5\times(-32) + B_4\times(32-16) + B_3\times(16-8) + B_2\times(8-4) + B_1\times(4-2) + B_0\times(2-1) =$

    (azonos numerikus értékek összerendelése)

    $= B_5\times(-32) + B_4\times32 - B_4\times16 + B_3\times16 - B_3\times8 + B_2\times8 - B_2\times4 + B_1\times4 - B_1\times2 + B_0\times2 - B_0\times1 =$

    $= -32\times(B_5 - B_4) - 16\times(B_4 - B_3) - 8\times(B_3 - B_2) - 4\times(B_2 - B_1) - 2\times(B_1 - B_0) - 1\times(B_0 - 0).$

Példa: előjeles Booth algoritmus

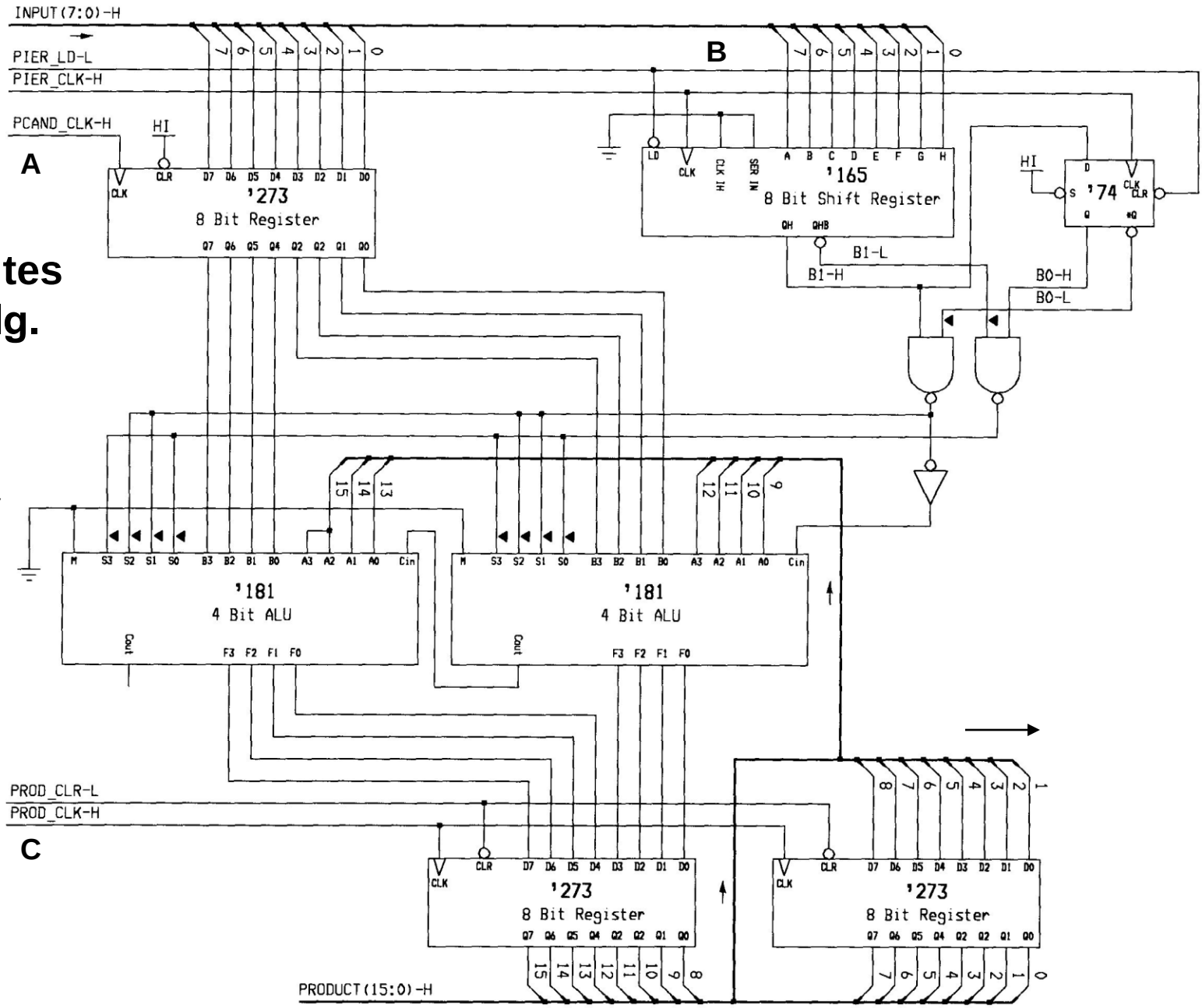
- Az előző oldalon lévő *átzárójelezéssel* a megfelelő értékeket két bitpár kivonásával kapjuk (a **zárójeles** kifejezés értéke ha **+**:akkor kivonás,/ ha **0**:akkor áteresztés / ha **–** összeadás történik). A súlytényezők 2 hatványai, és a szorzást a súlytényezők shiftelésével oldják meg (’165 Shift-regiszterrel). Egy összeadást mindig egy kivonás követ alternáló jelleggel.
- Példa: 543210 (bitpozíciók)
  011001= 25 „A”
  101101=–19 „B” Végrehajtjuk **A*B**-t!

Az újrakódolást bitpárokon végezzük el:

$-1\times(B_0-0)= \underline{-1}$	Co=0–1*A	Mivel – volt az érték, ezért kivonjuk a 0-ból az A-t.
$-2\times(B_1-B_0)= \underline{+2}$	C1=Co+2*A	Mivel + volt az érték, ezért hozzáadjuk C0-hoz a 2*A-t.
$-4\times(B_2-B_1)= \underline{-4}$	C2=C1–4*A	kivonjuk
$-8\times(B_3-B_2)= \underline{0}$	C3=C2	Mivel ‘0’ volt, Áteresztés, nem változik.
$-16\times(B_4-B_3)= \underline{+16}$	C4=C3+16*A	hozzáadjuk
$-32\times(B_5-B_4)= \underline{-32}$	C5=C4–32*A	kivonjuk

Példa: két 8-bites szám Booth alg. szorzására

A ’74 D tároló kimenetéről B0_H ill. a „B” szorzóregiszter kimenetéről B1_H jelek a szorzó shiftelődésének megfelelően generálódnak. Ezek állítják elő megfelelő kombinációs hálózat (2 NAND kapu, és 1 Inverter) segítségével az S0-S3 kiválasztó jeleket az ALU-nál.



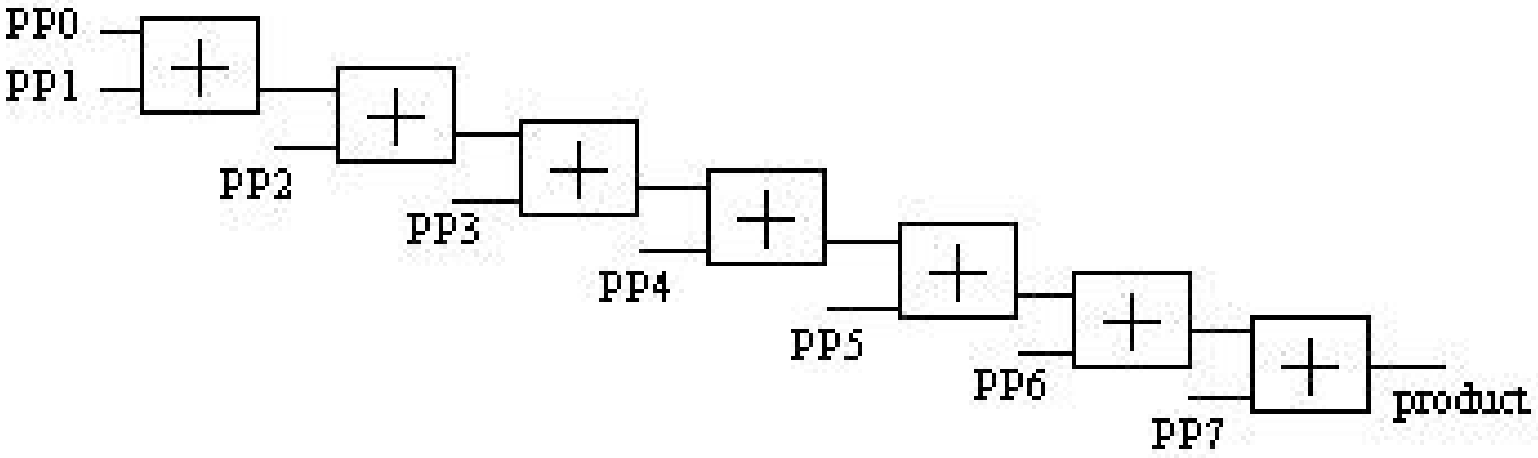
II. Közvetlen szorzási módszerek

Közvetlen szorzási módszerek

- Először a részlet-szorzatokat (Partial Products: PPi) állítjuk elő, amelyeket összeadunk. A részlet-szorzatok eltolását egységnyi kapu késleltetéssel lehet megoldani.
- Fajtái (részlet-szorzat képzés):
  - Lineáris modell,
  - Fa modell,
  - Full Adder felhasználásával,
  - CSA: Carry Save Adder,
  - Sorcsökkentős megvalósítás (row reduciton).

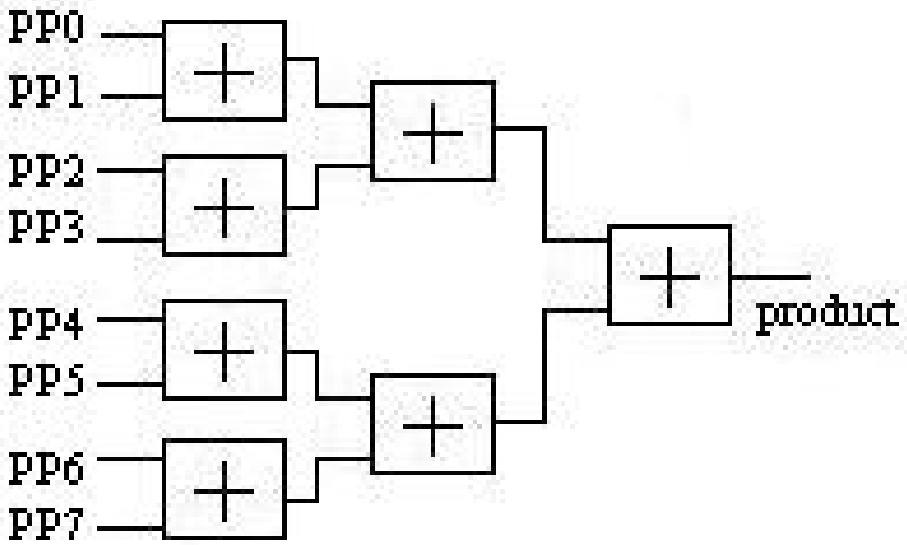
a.) Lineáris modell

- A parciális szorzatképzés után azonnal összeadhatók, 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: $T_{(DIRECT-LINE)}=(N-1)*T_{(SUM)}$



b.) Fa modell

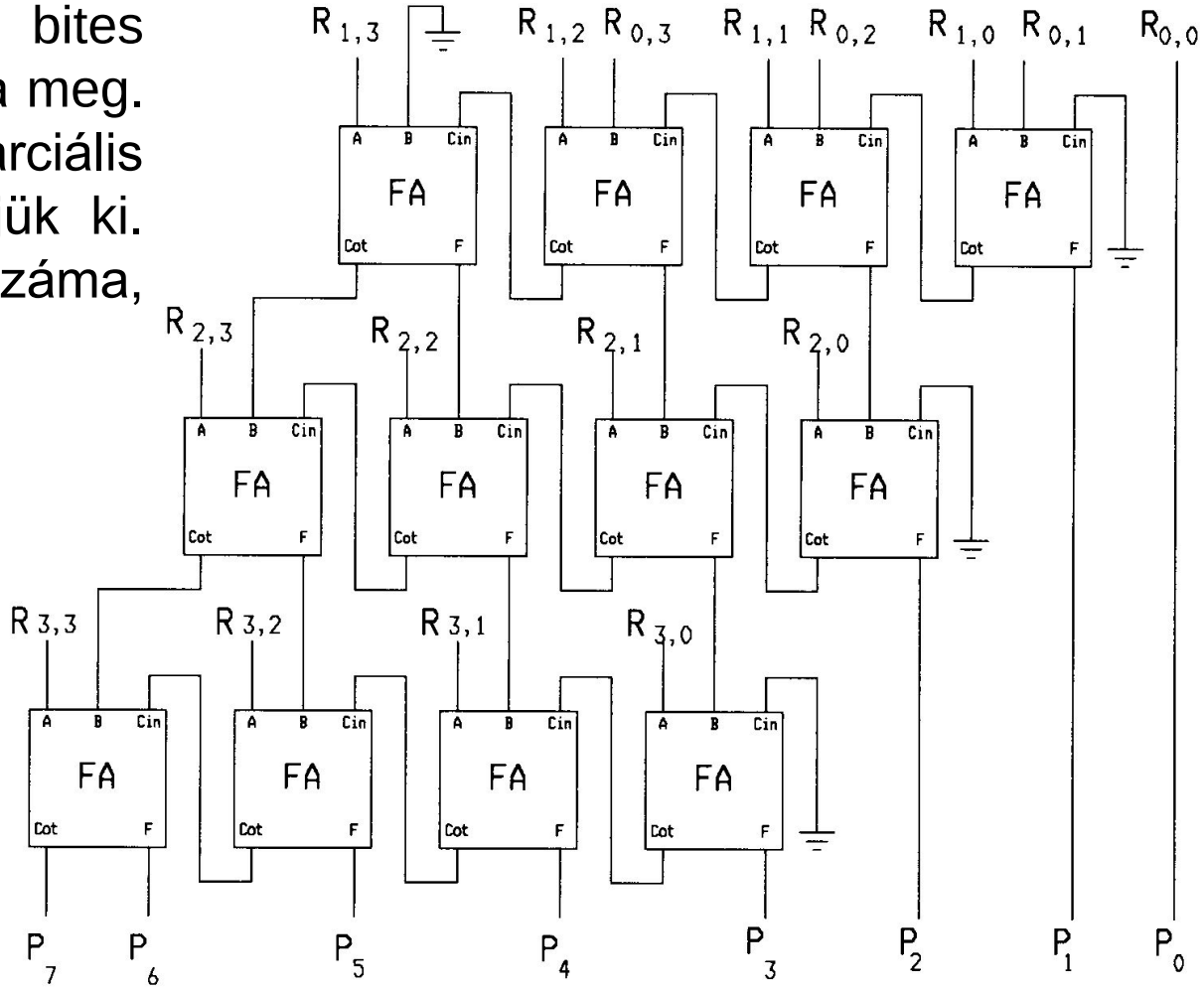
- A parciális szorzatok, képzésük után szintén azonnal összeadhatók, gyorsabban megkapjuk az eredményt, mint a lineáris modellnél, mivel ebben az esetben (N=8 bit esetén) csak 3 szintű a hierarchia, így kevesebb a késleltetés. N bites számok esetén (N-1) db összeadóra van szükségünk.
- Időszükséglet: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



c.) Full Adder-es megvalósítás

- A mellékelt ábra két 4 bites szám szorzását valósítja meg. A sorokat, mint parciális szorzat tömböket emeljük ki. Jel: R x,y, ahol x a sor száma, y a sor eleme (oszlop).

P=AxB				(A3	A2	A1	A0)	
				(B3	B2	B1	B0)	
				R 0,3	R 0,2	R 0,1	R 0,0	PP0
			R 1,3	R 1,2	R 1,1	R 1,0		PP1
		R 2,3	R 2,2	R 2,1	R 2,0			PP2
	R 3,3	R 3,2	R 3,1	R 3,0				PP3
P7	P6	P5	P4	P3	P2	P1	P0	

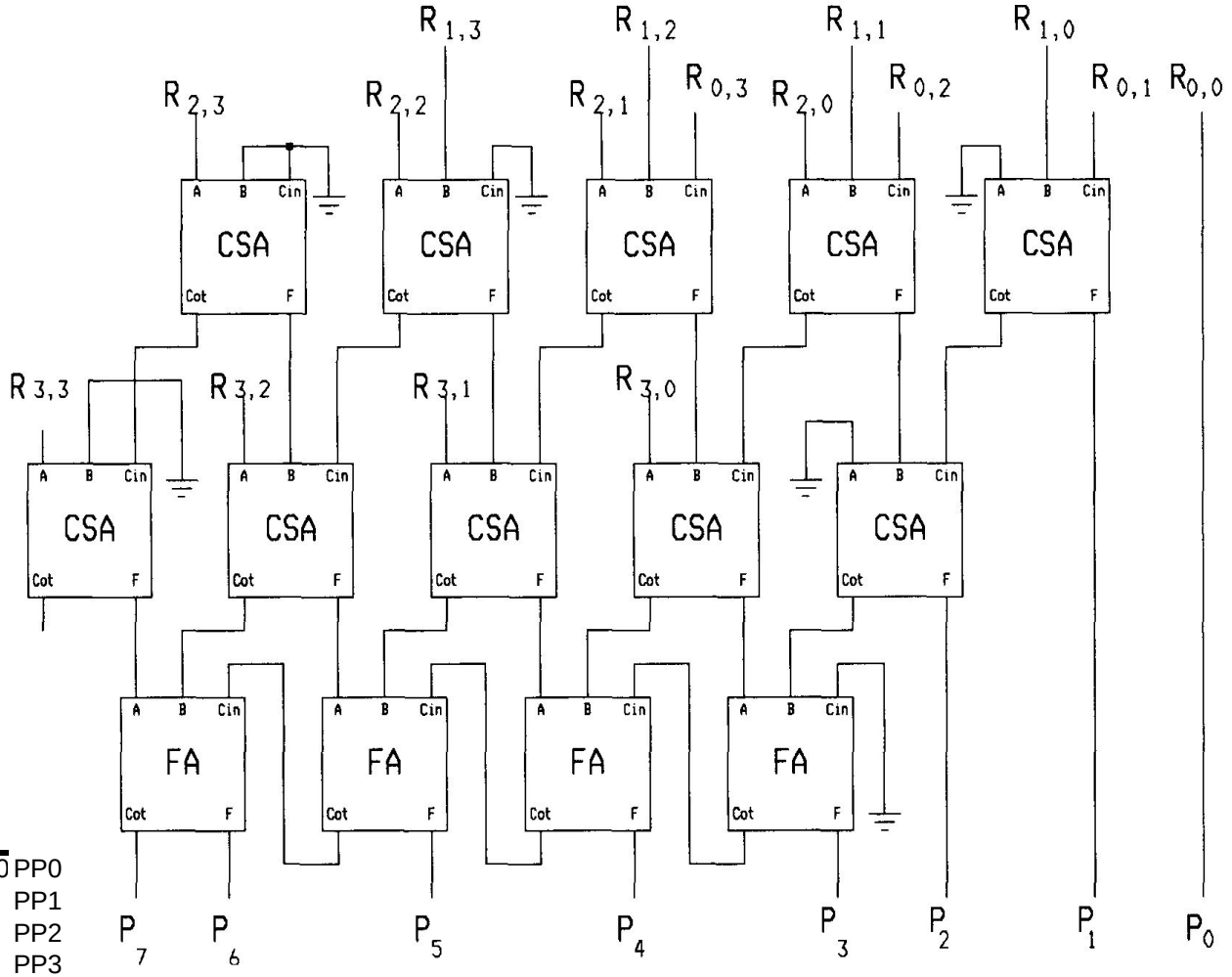


d.) CSA: Carry Save Adder

•CSA: olyan Full Adder, amely az előző szint átvitelét (Cout) eltárolja, és a következő szint Cin-jének továbbítja. Ezzel a módszerrel a szorzás tovább gyorsítható. A késleltetés *mindig 2G*.

•Az utolsó sorban FA-kat használunk, míg az első két sorban CSA-k találhatók. A CSA csökkenti az összeadandó sorok számát (3-2 sorcsökkentő egységnek felel meg).

P=AxB				(A3 (B3	A2 B2	A1 B1	A0) B0)	
				R 0,3	R 0,2	R 0,1	R 0,0	PP0
			R 1,3	R 1,2	R 1,1	R 1,0		PP1
		R 2,3	R 2,2	R 2,1	R 2,0			PP2
	R 3,3	R 3,2	R 3,1	R 3,0				PP3
P7	P6	P5	P4	P3	P2	P1	P0	

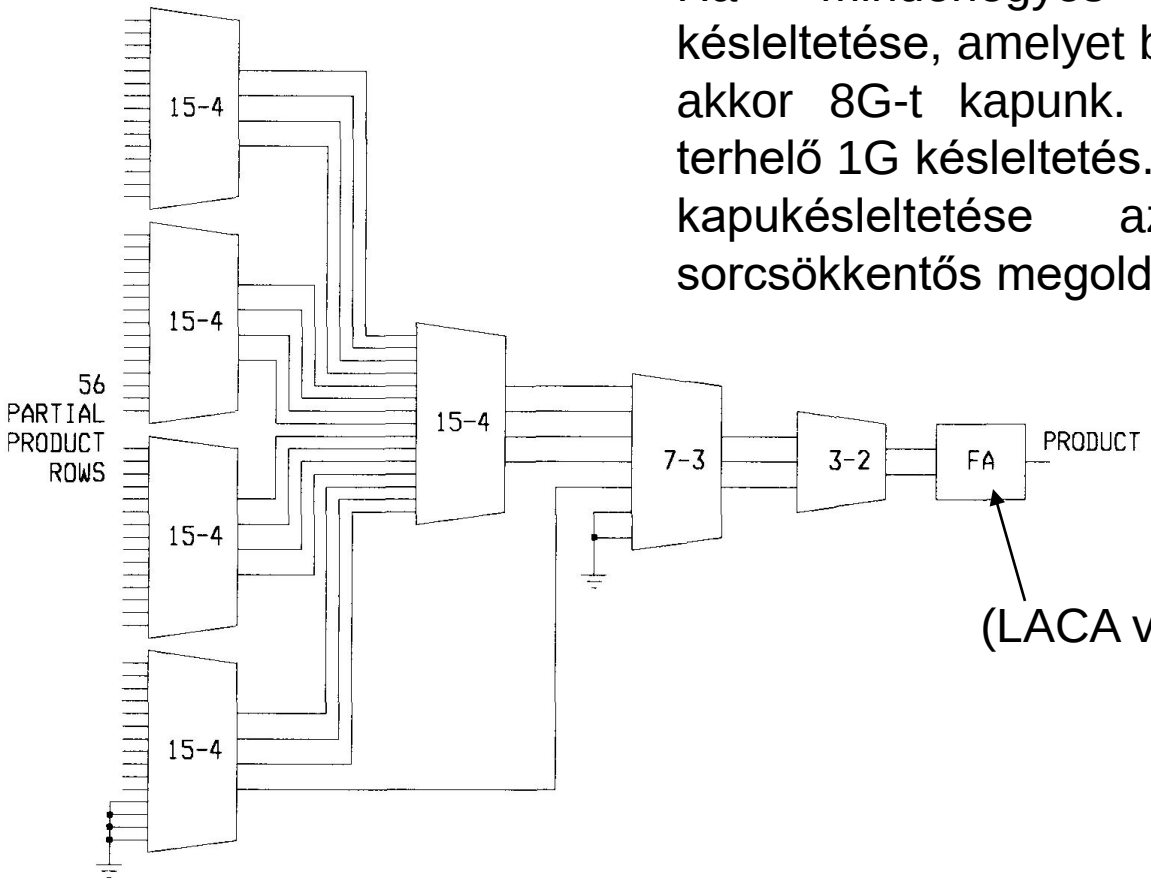


e.) Sorcsökkentős megvalósítás (row reduction)

- Példa: Két N=56 bites számot szeretnénk összeszorozni ezzel a megoldással (eredmény P= 2*N=112 bites lesz).
- Sorcsökkentő: Egy „k” kimenetű sorcsökkentő egység 0 – 2^k-1 értéket tud reprezentálni, ezért 2^k-1 bemenetet tud kezelni. Így definiálhatók 31-5, 15-4, 7-3, 3-2 sorcsökkentő egységek. Nagy előnyük, hogy a *parciális részletszorzatok (PPi) összeadását* párhuzamosan végzik. Így egy N bites bemenetet végül 2 bitesre tudunk redukálni, amely után egy egyszerű Pl. teljes összeadó (FA) vagy LACA összeadó használható.
- Most 56x56 bites szorzást végzünk: egy megkötésünk, hogy a legnagyobb alkalmazható sorcsökkentő egység 15-4. Az utolsó előtti 3-2 sorcsökkentő egység a carry save adder (CSA), amelyet végül egy LACA (tekintsük egy *b=8* bites CP-t propagáló és CG-t generáló LACG egységnek), amelyik a 2*56=112 bites eredményt számolja ki. Így LACA számítási szükséglete a következő:

$$T_{LACA} = 2 + 4 \times (\lceil \log_8(112) \rceil - 1) = 2 + 4 \times (3 - 1) = 10G$$

e.) Sorcsökkentős megvalósítás (row reduction) /folytatás/



Ha mindenegyes sorcsökkentőnek 2G a késleltetése, amelyet beszorzunk a szintek számával akkor 8G-t kapunk. Ehhez hozzájön, az inputot terhelő 1G késleltetés. Így összesen 9G+10G=19G a kapukésleltetése az eszköznek, ezzel a sorcsökkentős megoldással.

Osztó áramkörök

Osztó áramkörök:

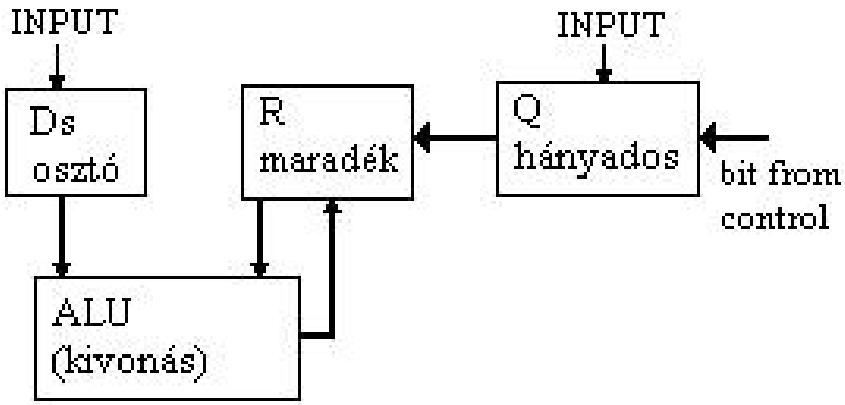
- I.) Hagyományos közvetlen osztási algoritmus
- II.) Iteratív osztási algoritmusok

I.) Hagyományos közvetlen osztási algoritmus:

Ez az osztási folyamat igen lassú eljárás. Lépései:

1. az *osztót* a Ds regiszterbe rakjuk, az *osztandót* a Q regiszterbe.
2. töröljük az R regisztert
3. iterációs lépés: kivonjuk az R-ből a Ds osztót. Ha R-Ds>0 akkor folytatódik, tehát ezt a megváltozott értéket visszatesszük az R-be, és egy **'1'-**est teszünk a Q regiszterbe. Ha R-Ds<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 shiftelünk a visszaszorzásnál)
6. A hányados generálódik elsőként az MSB felől, és a Q-ba shiftelődik 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!

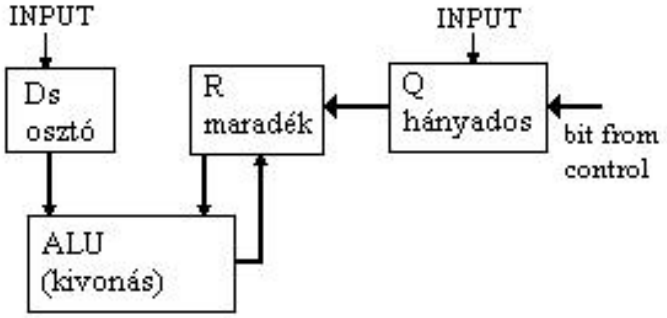
Decimális számok esetén:

```
5|8 : 5=1|1
0 8
  3
```

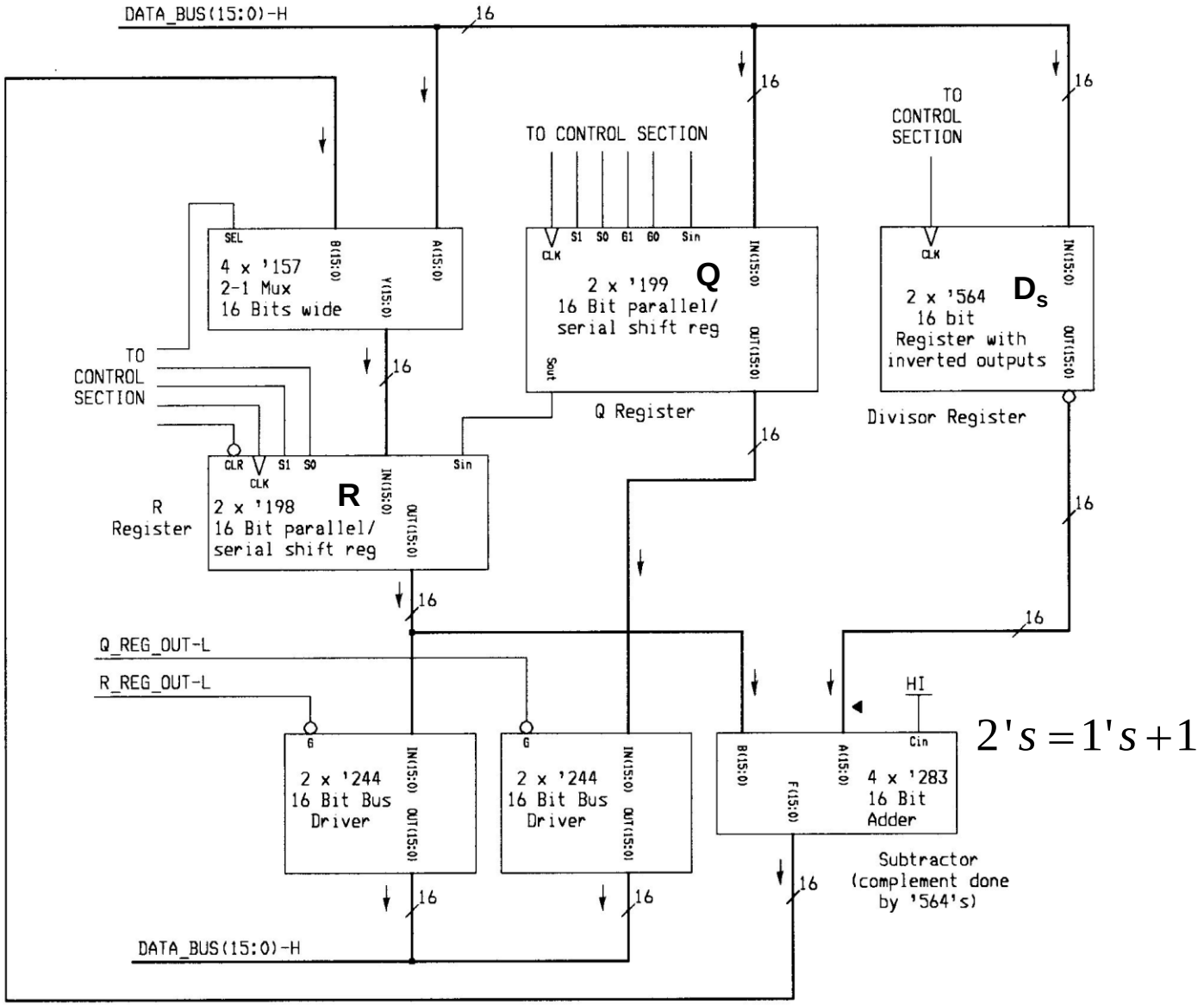
Bináris számok esetén hasonlóan

```
      1011     Q hányados (Ds hányszor van meg Dd-ben)

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

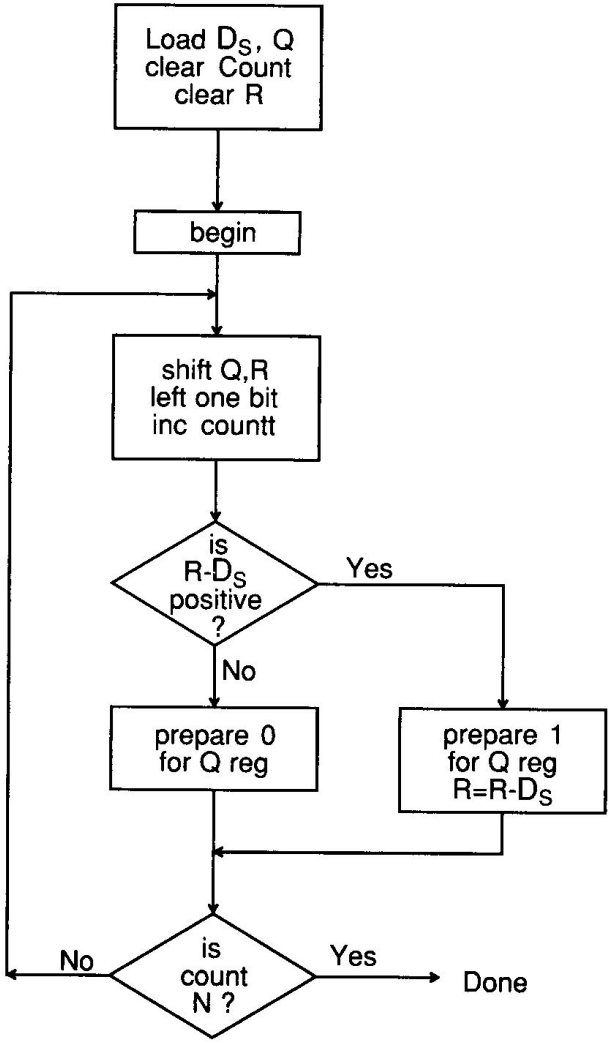


Hagyományos osztó áramkör



$$2's = 1's + 1$$

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 x0 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

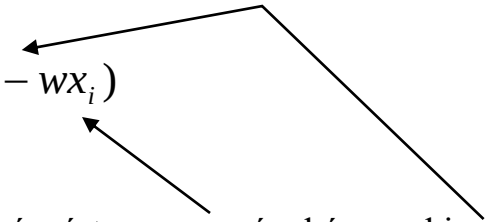
$$f(x) = \frac{1}{x} - w$$

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!

b.) Közvetlen gyors osztó

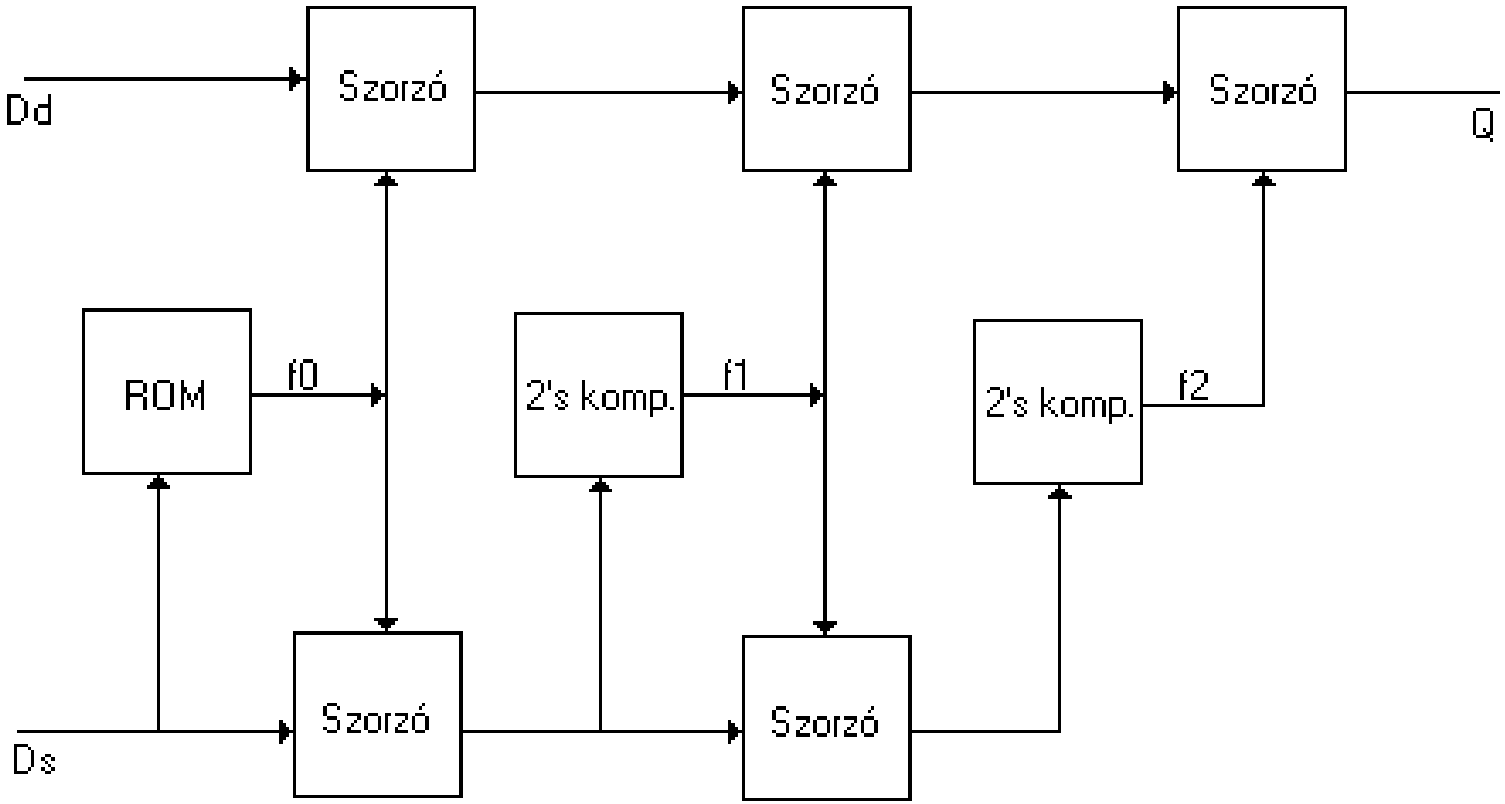
- Az iteratív osztási művelet másik módszere a következő: $Q=D_D/D_S$ kiszámolható a következő egyenlettel, ha a successive (egymást követő) $\mathbf{f_k}$ –k úgy vannak megválasztva, hogy a nevező az 1-hez konvergáljon.

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

Közvetlen gyors osztó működése

- A számláló iterációja ennél a módszernél $\mathbf{D_{Dn+1} = D_{Dn} \times f_n}$. . 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 törtkifejezésként írjuk fel (mantissza egy normalizált tört).
- Keressük a Q hányados (quotient) értékét. Hogy megkapjuk, mind az osztó, mind pedig az osztandó értékét ugyanazokkal az $\mathbf{f}_k$ 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: $\mathbf{D_S = 1 - x}$, ahol x-et D_S határozza meg, és mivel D_S kisebb 1-nél, így az x is kisebb 1-nél.

Közvetlen gyors osztó áramköri felépítése



Közvetlen gyors osztó

- Az osztás művelete f_o kiszámolásával kezdődik. Válasszuk $\mathbf{f_o} = 1+x = 1+(1-D_S) = \mathbf{2-D_S}$. 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 $\mathbf{D_{Dn+1} = D_{Dn} \times f_n}$.
- Egy kérdés vetődik fel: hogyan válasszuk meg f_k következő értékét. $\mathbf{f_1} = 1+x^2 = 1+(1-D_S \times f_o) = \mathbf{2- D_S \times f_o}$. Tehát minden egyes új f_k –t úgy kapunk meg, hogy vesszük az f_{k-1} és a Ds (nevező) szorzatának 2's komplemensét. 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)

Példa 1.

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

- 0. D_{DO}=0,4000000 D_{SO}=0,7000000 fo=1,3000000
- 1. D_{D1}=0,5200000 D_{S1}=0,9099999 f1=1,0900000
- 2. D_{D2}=0,5668000 D_{S2}=0,9918999 f2=1,0081000
- 3. D_{D3}=0,5713911 D_{S3}=0,9999344 f3=1,0000656
- 4. D_{D4}=0,5714286 D_{S4}=0,9999999 f4=1,0000000
- 5. D_{D5}=0,5714286 D_{S5}=1,0000000 f5=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 Ds elég közel volt az 1-hez, és x=0.3 volt. (x=1-Ds).

Példa 2.

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

- 0. $D_{DO}=0{,}1000000$ $D_{SO}=0{,}1500000$ $fo=1{,}8499999$
- 1. $D_{D1}=0{,}1850000$ $D_{S1}=0{,}2775000$ $f1=1{,}7224999$
- 2. $D_{D2}=0{,}3186625$ $D_{S2}=0{,}4779938$ $f2=1{,}5220062$
- 3. $D_{D3}=0{,}4850063$ $D_{S3}=0{,}7275094$ $f3=1{,}2724905$
- 4. $D_{D4}=0{,}6171659$ $D_{S4}=0{,}9257489$ $f4=1{,}0742511$
- 5. $D_{D5}=0{,}6629912$ $D_{S5}=0{,}9944868$ $f5=1{,}0055132$
- 6. $D_{D6}=0{,}6666464$ $D_{S6}=0{,}9999696$ ....

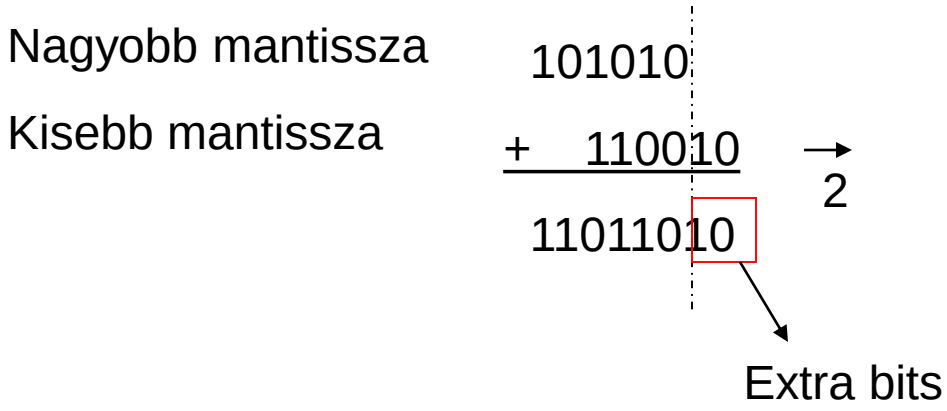
Látható, hogy itt nem kapjuk meg a kivánt értéket ($D_{D6}=0{,}6666464$) 6 iterációs lépés alatt. Ezért hogy elérjük a kivánt pontosságot véges számú lépés alatt, ROM-ot kell használni (ahol fo kezdeti értékét tároljuk).

Extra bitek kezelése

- Truncation (levágás)
- Rounding (normál kerekítés)
- Zero-bias rounding (zéróhoz kerekítés R*)
- Jamming
- ROM rounding

Extra bit: probléma

- Két 6-bites lebegőpontos szám összeadása, exponens egyeztetés után:



a.) Truncation (levágás)

- Levágás: egyszerűen elhagyjuk az extra biteket.
  - Hibája a pontosság: a kapott M_F mantissza eltér a valós mantissza M_R értékétől:

$$ERR_{TRUNC} = M_R - M_F$$

  - n-bites bias (offset) hibája: tárolt érték mindig kisebb lesz, mint a valós/aktuális érték (mindig pozitív bias-t kapunk)
  - Kevesebb extra bittel kisebb lesz a bias, így a hiba is csökken.

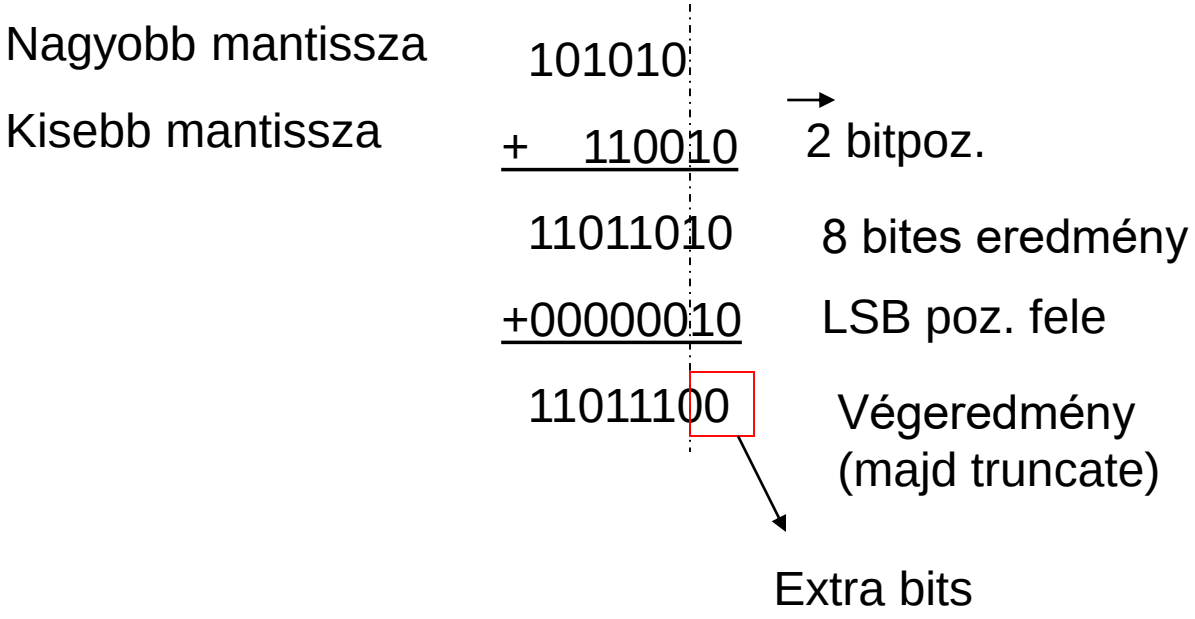
Truncation: példa

- ERR → bias: mindig pozitív

cases	MR	MF	ERR_{TRUNC}
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

b.) Rounding (kerekítés)

- Bias csökkentése a cél, úgy hogy a levágás előtt az LSB bit értékének a felét hozzáadjuk a számhoz:



PI: Rounding (kerekítés)

- ERR → 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_{ROUND}
a	xx0.00	xx0.10	xx0.	0.00
b	xx0.01	xx0.11	xx0.	+0.01
c	xx0.10	xx1.00	xx1.	-0.10
d	xx0.11	xx0.11	xx1.	-0.01
e	xx1.00	xx1.10	xx1.	0.00
f	xx1.01	xx1.11	xx1.	+0.01
g	xx1.10	xy0.00	xy0.	-0.10
h	xx1.11	xy0.01	xy0.	-0.01

Változás a truncate-hez képest!

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

c.) Round-to-Zero (R* rounding)

- Cél. A hiba minimalizálása, lehetőleg zérus bias elérése, kerekítéssel.
- ERR_{ZERO} –kat összeadva a teljes bias értéke nulla lesz.
- Kisebb a hibája mint más extra-bit kezelő technikáknak

Pl: Round-to-Zero (R*)

case	MR	MR+1/2 LSB	MF	ERR_{ZERO}
a	xx0.00	xx0.10	xx0.	0.00
b	xx0.01	xx0.11	xx0.	+0.01
c	xx0.10	xx1…	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	xx1.10	xy1…	xy1.	+0.10
h	xx1.11	xx1.11	xy0.	-0.01

Változás a rounding-hoz képest!

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

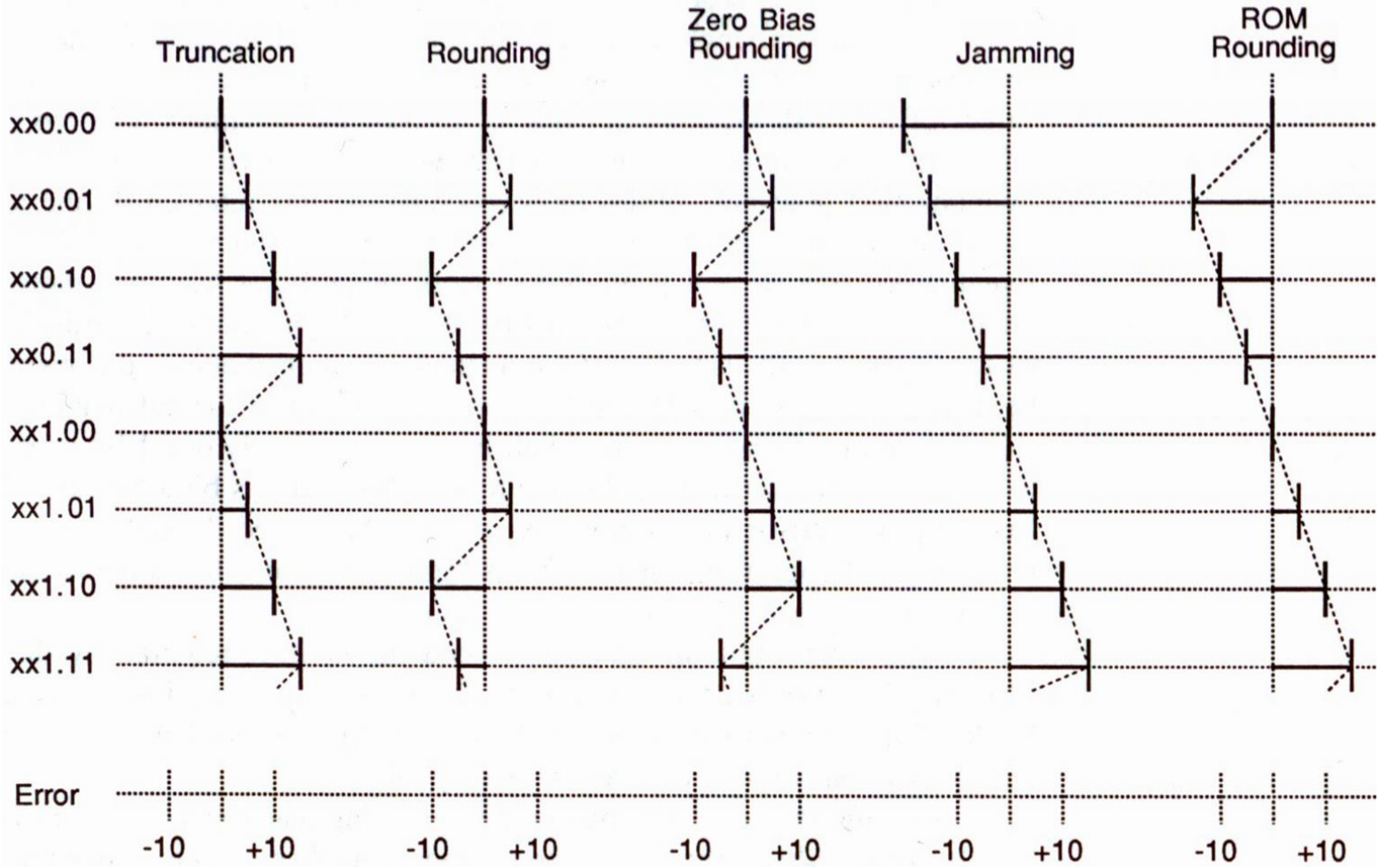
d.) Jamming (~fix értéken rögzítés)

- Neumann
- Csökkenteni a teljes hibát (jobb módszer, mint a truncation).
- Pl. *Jam to ‘1’:* LSB bitet fix-en ‘1’-re rögzítjük, az extra bitek értékétől függetlenül!
- Ennek a módszernek ugyan nagyobb a hibája, mint a legtöbb extra bit kezelő módszerek, de idővel ugyanolyan kicsi lesz a bias-a, mint a kerekítésnek.
- Olyan gyors viszont mint a truncation (itt nincs időszükséglet, mint a kerekítési fázisban, LSB-t mindig pl. ‘1’-re), ráadásul kis bias.

e.) ROM rounding

- Vizsgálat: extra biteket és LSB-biteket változatlanul hagyjuk
- Döntési folyamathoz ROM-ból való értékek kiolvasását használjuk
- Biztosítja, hogy a nagyobb bitpozíciókba nem kell „carry-t propagáltatni” (mint rounding-nál), gyorsabb is
- Bias kontrollálható (akár zérus is lehet végül)

Extra-bit kezelő módszerek összehasonlítása:



Pannon Egyetem
Képfeldolgozás és Neuroszámítógépek Tanszék



Digitális Rendszerek és Számítógép Architektúrák

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

Előadó: Vörösházi Zsolt
Dr. Szolgay Péter

Jegyzetek, segédanyagok:

- Könyvfejezetek:
  - http://www.knt.vein.hu
    → Oktatás → Tantárgyak → Digitális Rendszerek és Számítógép Architektúrák (Nappali képzés)
    (chapter04.pdf)
- Fóliák, óravázlatok .ppt (.pdf)
- Feltöltésük folyamatosan

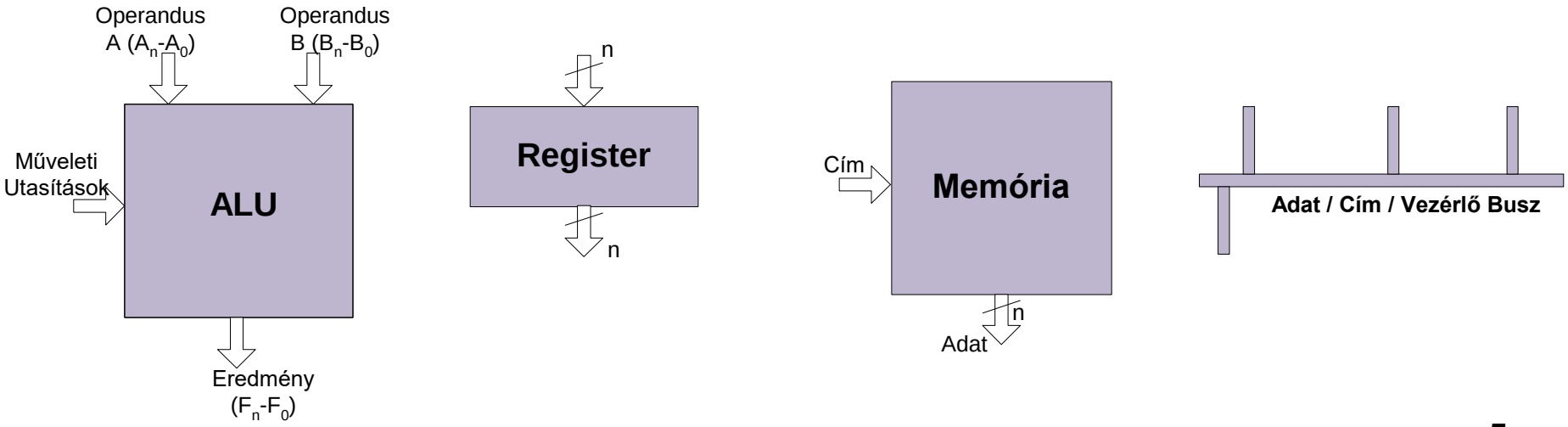
Utasítás végrehajtás folyamata

- Utasítás kódok
- Címzési módok
- Programvezérlő utasítások
- RISC, CISC processzorok

Alapvető digitális építőelemek

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.
- (Korábban részletesen: chapter_03.pdf)

Memória egységek

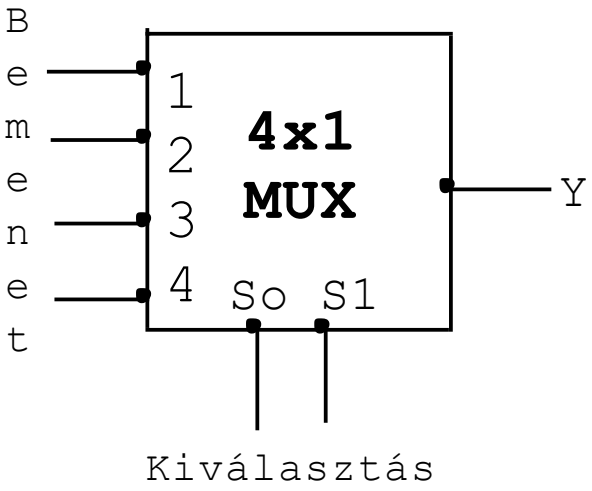
- Az ALU által kezelt / végrehajtott adatok a **memóriában** (tároló rekeszek lineáris tömbjében) tárolódnak el. A memória rekeszei általában olyan szélesek, amilyen széles az adatbusz. Például, legyen n-bit széles, és álljon M számú rekeszből. Ekkor $\log_2(M)$ számú címvezetékkel címezhető meg. Az adatbuszon kétirányú (írás/olvasás) kommunikáció is megengedett. Memória *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.

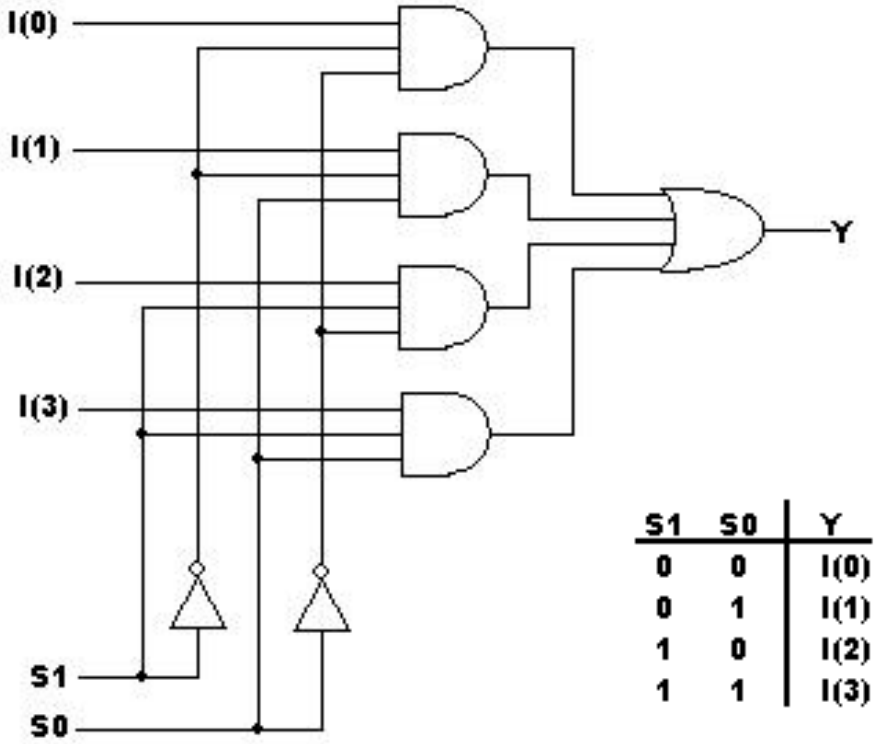
Adatbuszok – adatvonalak

- 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.

a.) Multiplexer (MUX)

- N kiválasztó jel → 2^N bemenet, 1 kimenet
- Példa: 4:1 MUX

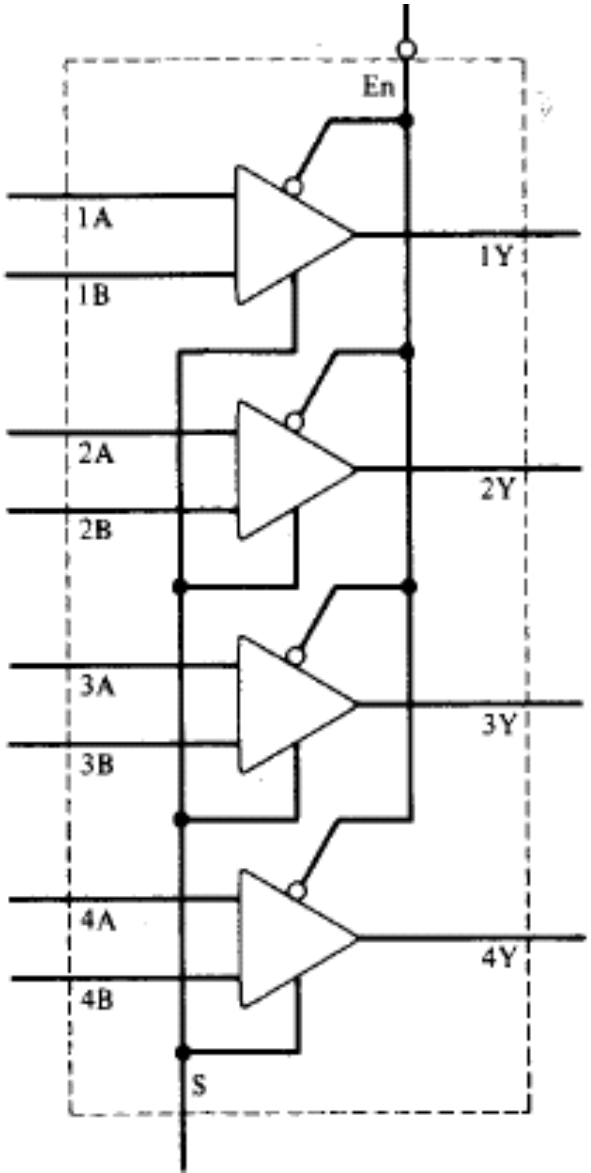




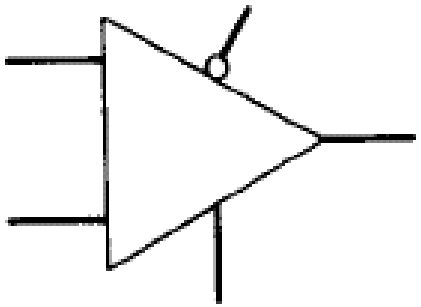
S1	S0	Y
0	0	I(0)
0	1	I(1)
1	0	I(2)
1	1	I(3)

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

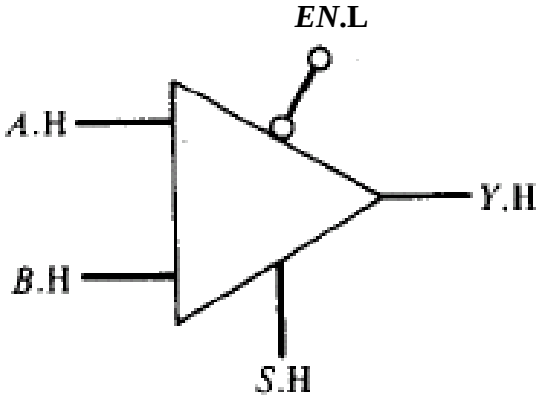
TTL’74LS157 pl. 2:1 MUX



- Quad (4 db) 2-bemenetű 2:1 MUX-ból áll.
- ← Közös S, EN jelek.
- 4 db Y(1,2,3,4) kimenet



2:1 MUX szimbóluma

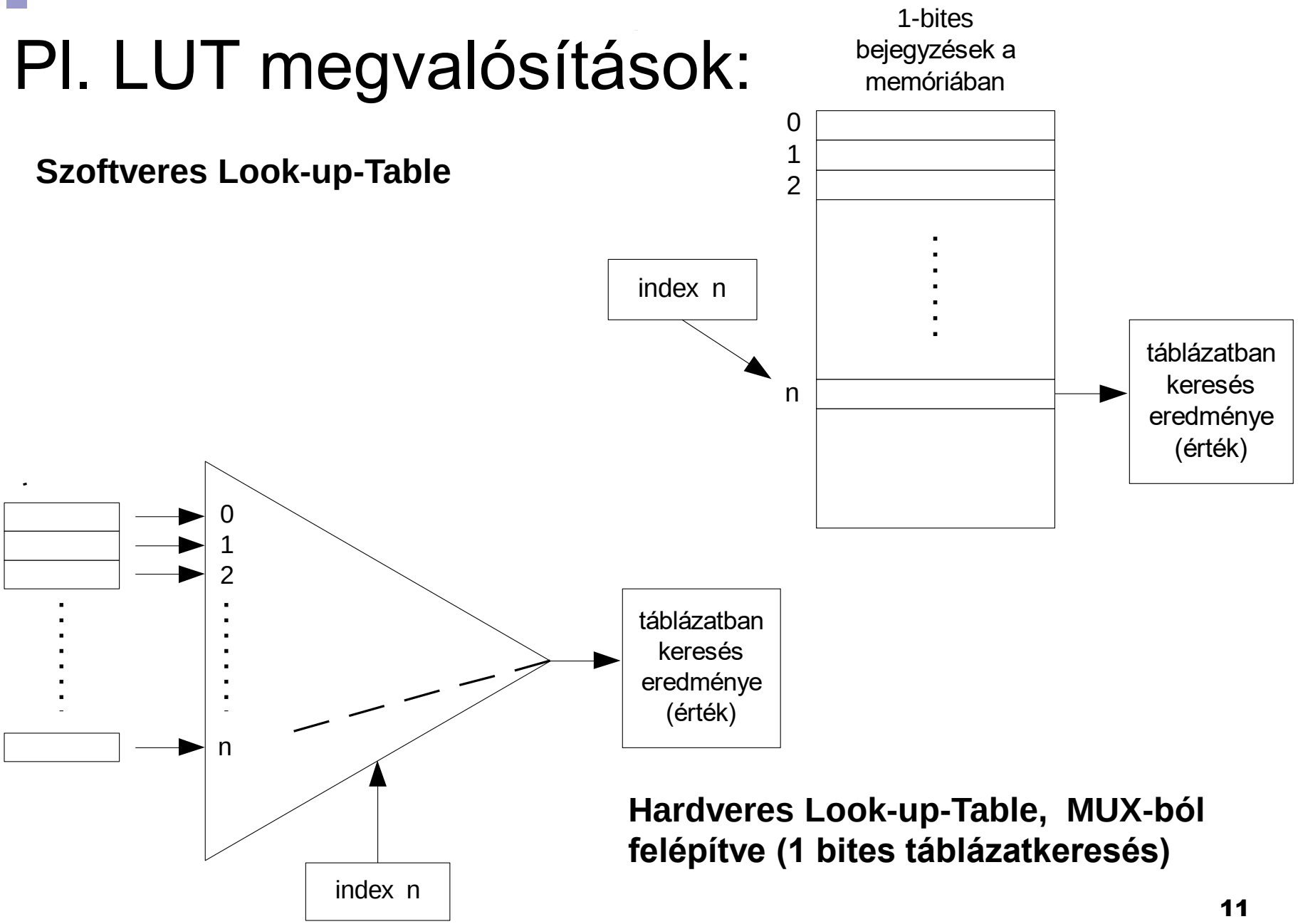


2:1 MUX áramköri szignál jelölésekkel

$$Y = EN \cdot (A \cdot \overline{S} + B \cdot S)$$

Pl. LUT megvalósítások:

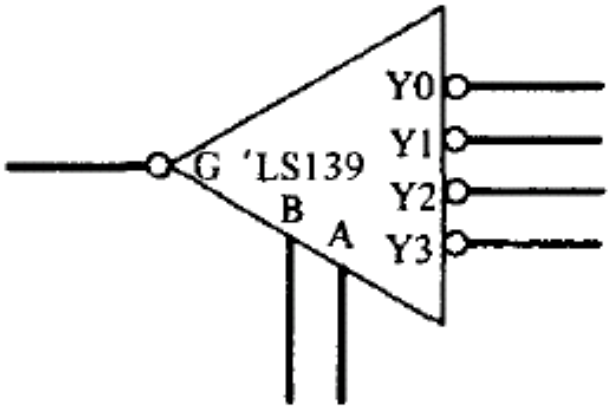
Szoftveres Look-up-Table



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

b.) Példa - 1:4 Demultiplexer

- TTL 74'LS139 duál 1:4 demultiplexer
  - Kereskedelmi forgalomban kapható
  - G: egy bemenet
  - A,B: routing control jelek (bináris kód)
  - 4-kimenet mindegyike False, egyet kivéve, amelyik a kiválasztott (annak az értéke a bemenettől függően lehet T/F)



T=L !

Példa - 1:4 Demultiplexer (folyt)

- Kanonikus táblázat

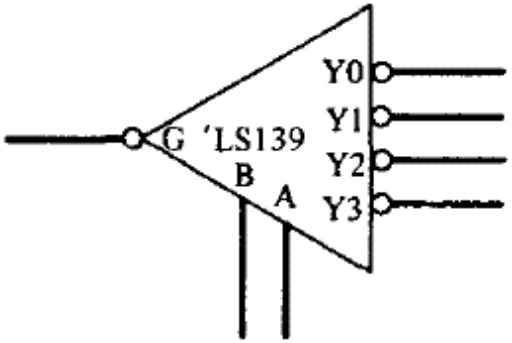
Demultiplexer logika						
G	B	A	Y0	Y1	Y2	Y3
0	x	x	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

T=L !

Feszültség-logikai tábl.

74'LS139 feszültség-logika						
G.L	B.H	A.H	Y0.L	Y1.L	Y2.L	Y3.L
H	x	x	H	H	H	H
L	H	H	L	H	H	H
L	H	L	H	L	H	H
L	L	H	H	H	L	H
L	L	L	H	H	H	L

- Demultiplexer logikai egyenletei:



$$Y0 = \overline{B} \cdot \overline{A} \cdot G$$

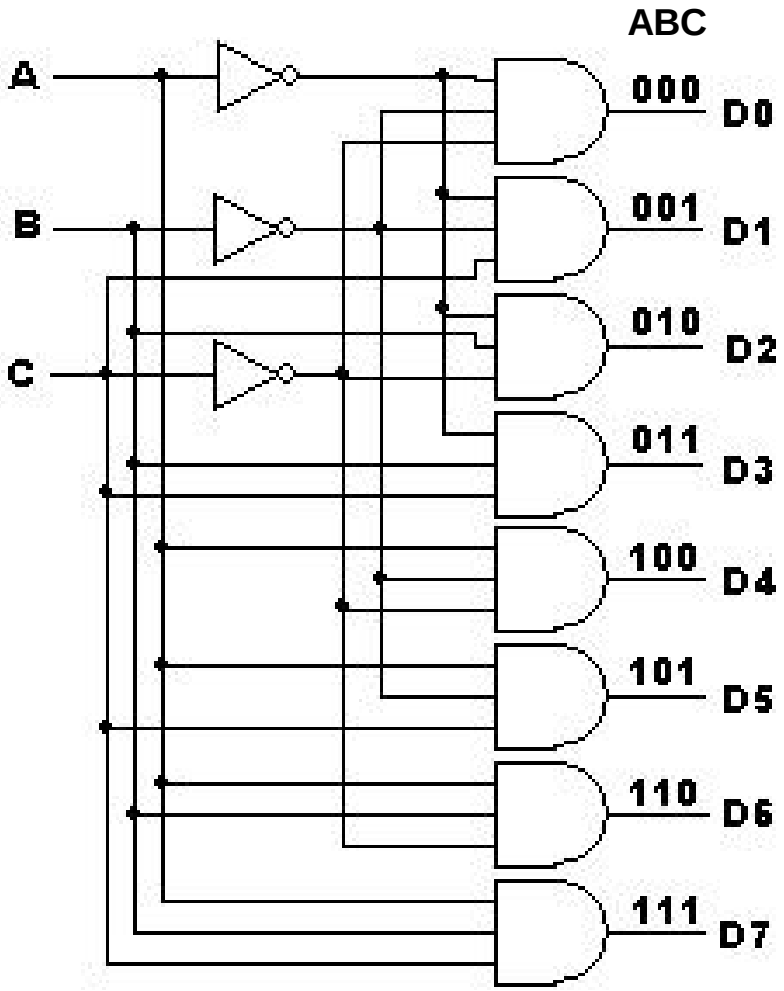
$$Y1 = \overline{B} \cdot A \cdot G$$

$$Y2 = B \cdot \overline{A} \cdot G$$

$$Y3 = B \cdot A \cdot G$$

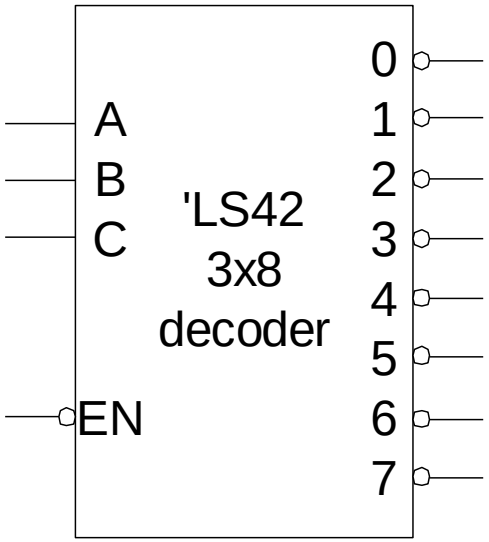
c.) Dekódoló áramkörök

- N bemenet esetén 2^N kimenete van
- Példa: 3x8 dekóder áramkör
  - Példa: Hamming-kódú hibajavító áramkör



TTL’74LS42 dekóder áramkör

- **3→8** dekóder áramkör
  - (A,B,C) 3 bemenet, (1…7) 8 kimenet
- EN: engedélyező jel, (T=L) alacsony aktív

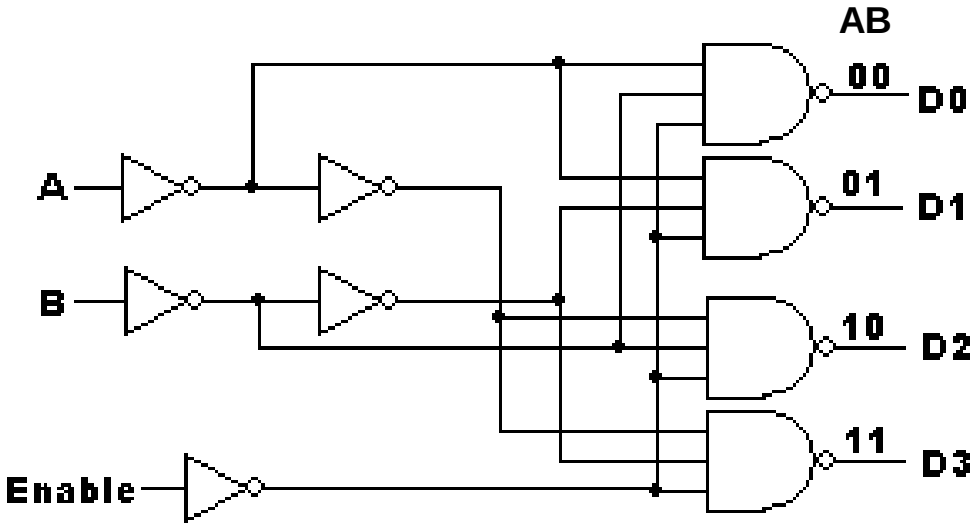


Mixed logic szimbólum

T=L!

Példa: 2x4 Dekódoló áramkör engedélyező bemenettel

- EN: alacsony aktív állapotban működik
- 2 bemenő bit (A,B)
- 4 kimenő bit (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
  - ***Hagyományos encoder***: csak egy bemenete lehet igaz egyszerre
  - ***Priority encoder***: több bemenete is igaz lehet egyszerre, de azok közül a legnagyobb bináris értékű, azaz prioritású bemenethez generál kódot! (kód: address, index lehet)
    - I/O, IRQ jelek generálásánál használják leggyakrabban

Probléma: Priority encoder esetén

- Mi van akkor, ha még sincs igaz bemenete (mindegyik hamis)? Két megoldás van:
  - *1.) módszer*: Input vonalak megszámozása 1-től (D1) kezdődően, és a 0 kimeneti kód (itt FF) jelenti, hogy mind „hamis” volt. (Továbbá: X – don’t care)
  - *2.) módszer*: input vonalak megszámozása 0-tól (D0) kezdődően, és egy külön vezérlőjelet (W) biztosítani arra, hogy nincs „igaz” bemenet. (Továbbá: X – don’t care)

Method 1

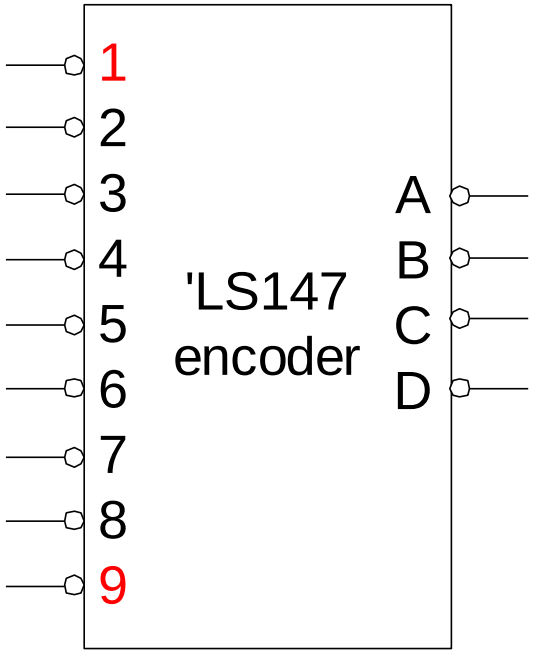
D3	*D2*	*D1*	*B*	*A*
F	F	F	F	F
F	F	T	F	T
F	T	X	T	F
T	X	X	T	T

Method 2

D3	*D2*	*D1*	*D0*	*B*	*A*	*W*
F	F	F	F	–	–	T
F	F	F	T	F	F	F
F	F	T	X	F	T	F
F	T	X	X	T	F	F
T	X	X	X	T	T	F

TTL’74LS147 Priority encoder - kódoló áramkör

- 10-input, 4-output encoder
  - „0” nincs jelölve: amikor az összes bemenet False (lefoglalt)
  - Alkalmazás:
    - Cím, indexgenerálás
    - LUT választás



Mixed logic szimbólum

T=L!

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: $A.EQ.B = A \square B$
- n-bites minták esetén:

$$A.EQ.B = (A0 \square B0) \cdot (A1 \square B1) \cdot \ldots \cdot (An \square Bn)$$

Ismétlés: EQ/XNOR/Coincidence operátor

- Logikai egyenlet: $A.EQ.B = A \square\ B = A \cdot B + \overline{A} \cdot \overline{B}$
- Referenciabit szerinti megkülönböztetés:
  - ha a referencia bit (B), amihez hasonlítunk **konstans**
  - ha a referencia bit (B) egy **változó** mennyiség
- Példa: ha B referencia konstans -> egyszerűsítése A-nak

$$A.EQ.B = A \quad if \quad B = T$$

$$A.EQ.B = \overline{A} \quad if \quad B = F$$

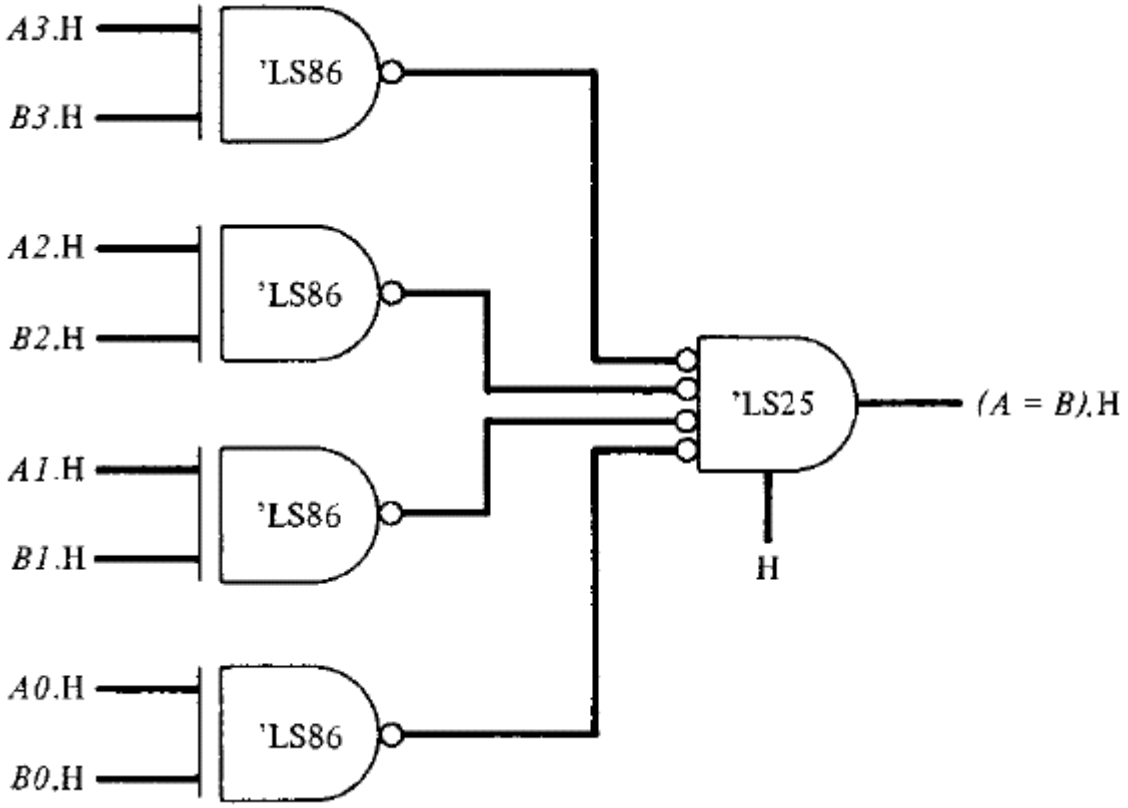
- Példa: legyen B egy 4-bites konstans mennyiség (B=TFFT), és A tetszőleges, akkor:

$$A.EQ.B = A0 \cdot \overline{A1} \cdot \overline{A2} \cdot A3$$

Példa: 4-bites komparátor

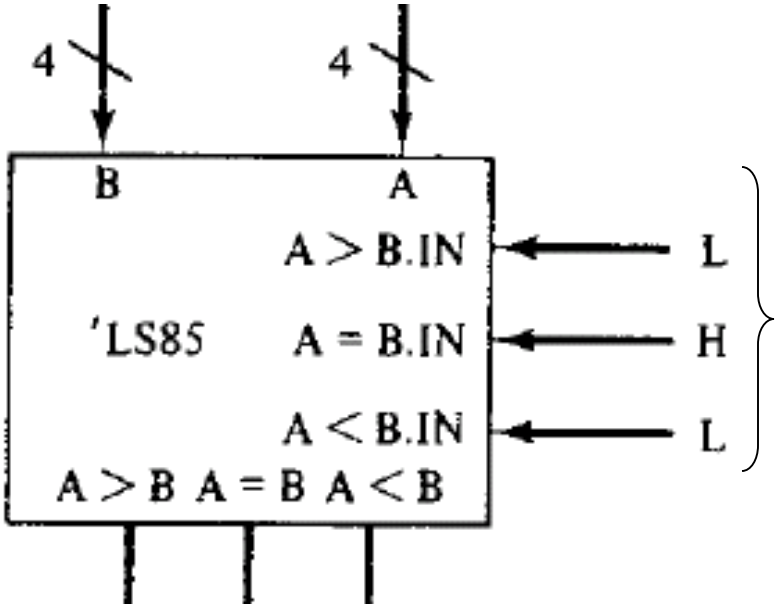
- Mixed-logic kapcsolási rajza, és log.egyenlete:

$$A.EQ.B = (A0 \square\ B0) \cdot (A1 \square\ B1) \cdot (A2 \square\ B2) \cdot (A3 \square\ B3)$$



’74LS85 4-bit Magnitude Comparator

- Magnitude comparing (~nagyságrend összehasonlítás):
  - két kifejezés **nagyságának** összehasonlítása (A<B; A=B; A>B stb.) egyszerre

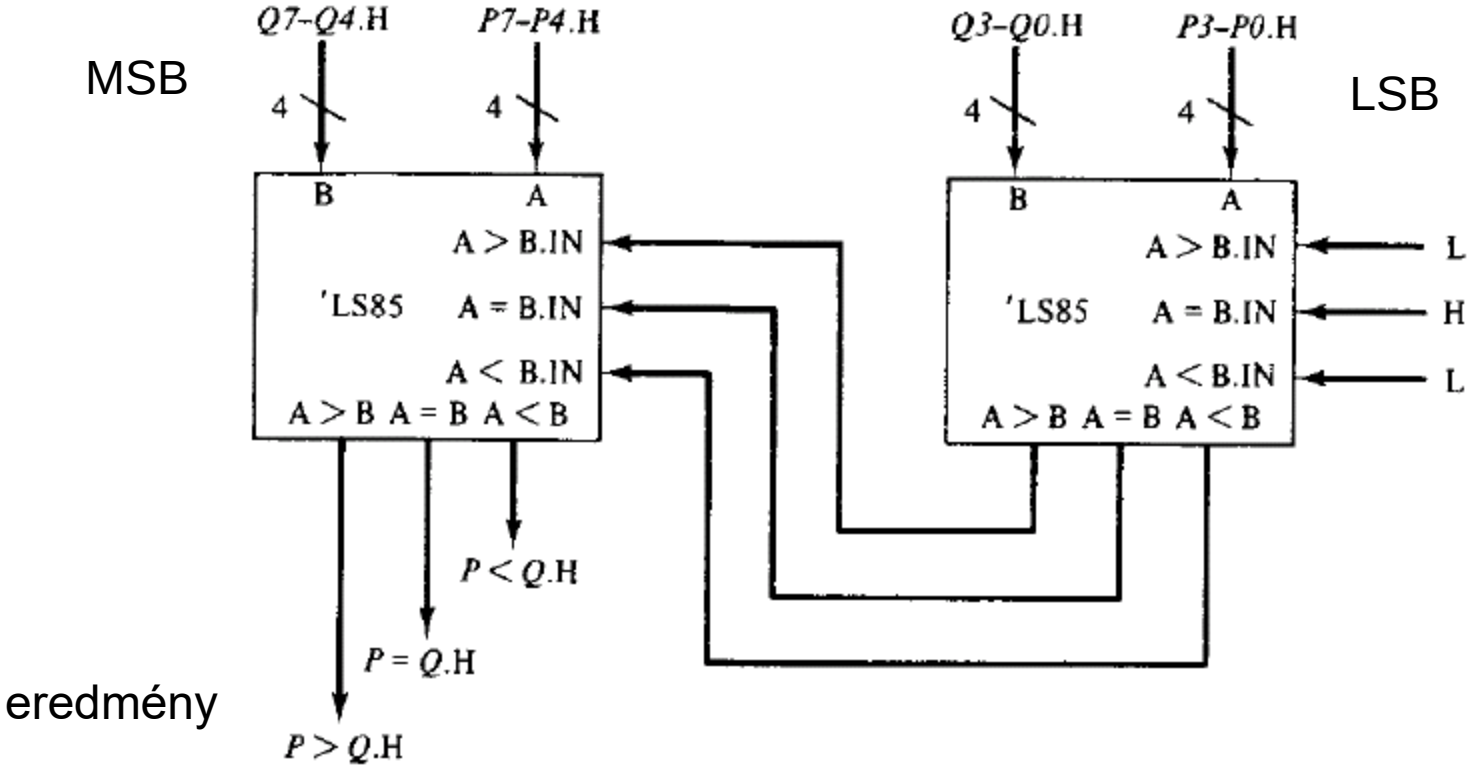


3 állapot (státusz) –bemenet

Ha B.IN=H: egyenlőség

Példa: 8-bites Magnitude Comparator – egyenlőség esetén

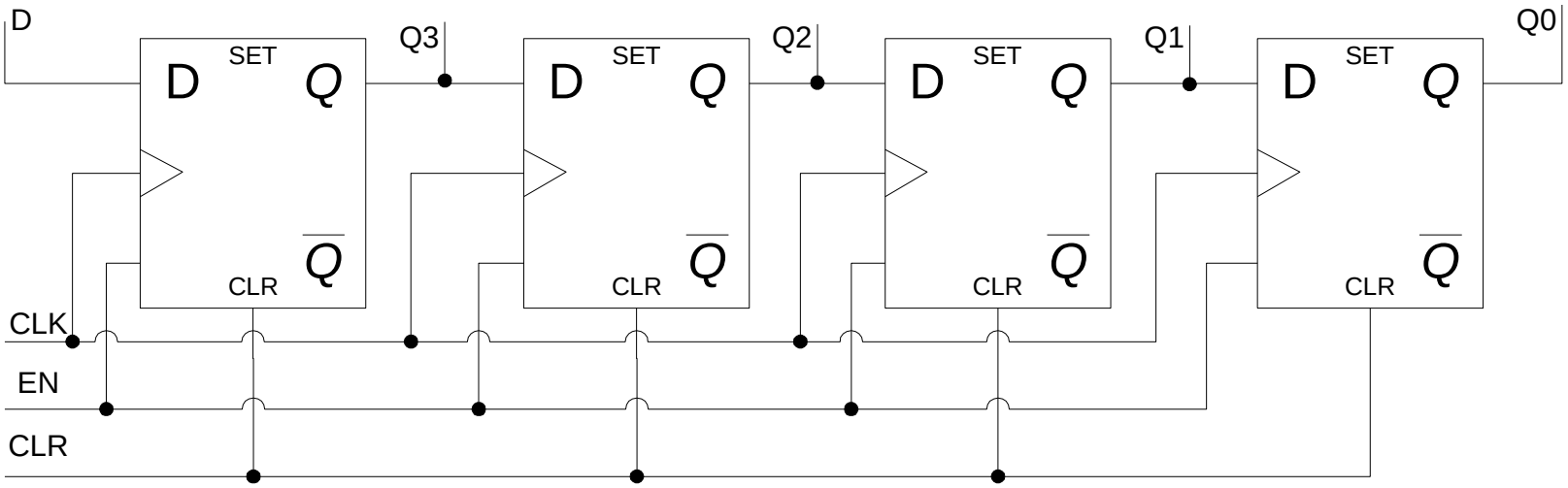
- Kettő 4-bites ’74LS85 Magnitude komparátor sorbakötéséből („cascading”) kapjuk a 8 bites (P,Q) értékek összehasonlítását



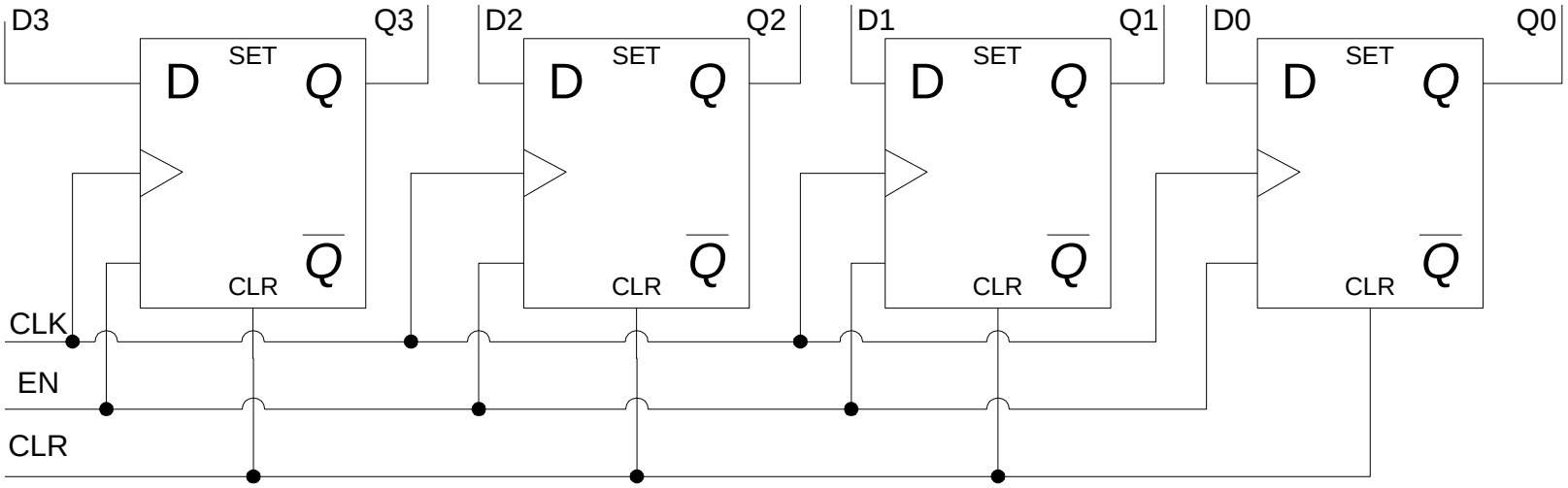
Regiszterek

- 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 rakja a tárolt adatokat, vagy például egy vezérlőjel hatására, lépteti (shift-eli) a benne lévő adatokat.

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

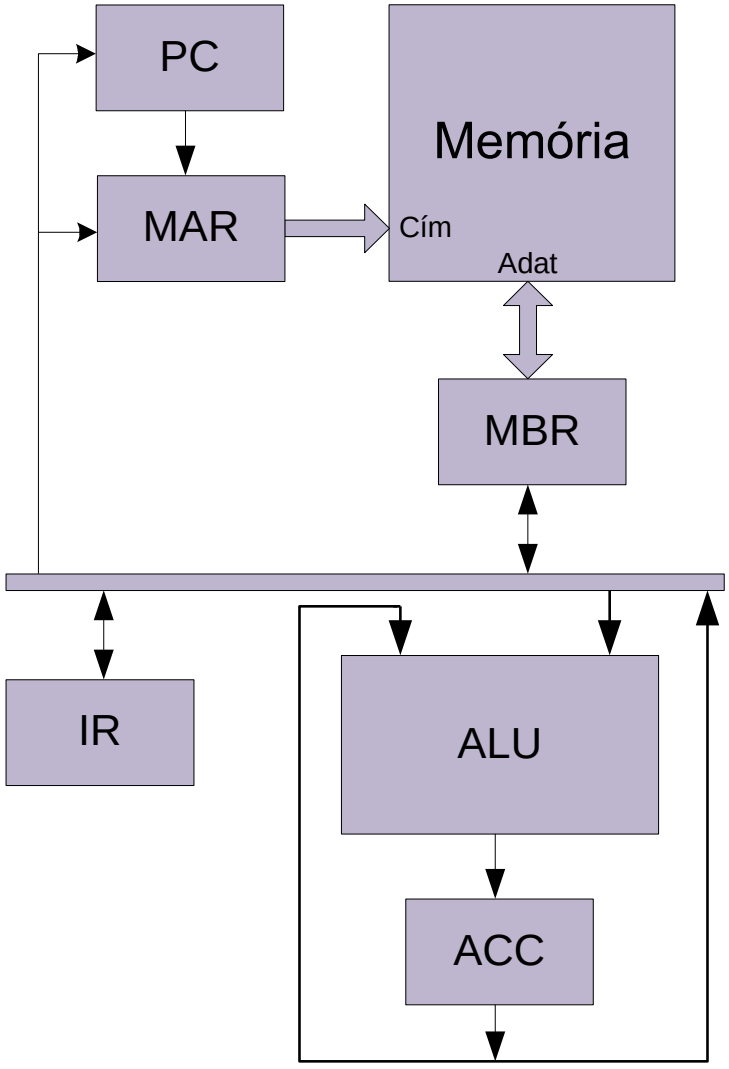


4-bites Parallel In/ Parallel Out regiszter (D-tárolókból felépítve)



Egyszerű számítógép (egycímű gép) blokkdiagramja

•***MAR: Memory Address Register*** *(Memória-cím Regiszter)*: információ helyét azonosítja adott memóriacím alapján.
•***MBR: Memory Buffer Register*** *(Memória Puffer Regiszter)*: tárolja a memóriába bevitt, ill. érkező információt. Egy adott memóriacímen lévő adat kiolvasásakor az ott lévő bejegyzés törlődhet (destruktív memória)
• ***PC: Program Counter*** *(Programszámláló)*: a soron következő (végrehajtandó) utasítás helyét azonosítja. Azon gépeknél, amelyek 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, aritmetikai egységek vezérlő vonalait a végrehajtáshoz szükséges működési módba állítsák. Az IR olyan széles, hogy az utasítás műveleti kódja ill. a hozzá tartozó egyéb utasítások ideiglenes másolatai eltárolhatók legyenek.
•***ACC: Accumulator regiszter*** *(tároló regiszter):* eredmény ideiglenes tárolására használjuk (összes adatkezeléshez tartozó utasítás tárolása).



Utasítások kódolása

Utasítás kódok

- A rendszer-tervezéshez szükséges erőforrások a *regiszterek, ALU, memória, adatbuszok* nem elegendőek a végrehajtás egyes fázisainak (tranzakcióknak) ábrázolásánál. Szükség van egy olyan eljárásra, amely leírja ezeket az egyes egységek között végbemenő tranzakciókat.
- Utasítások végrehajtásának leírására szolgáló programnyelv az **assembly**. Az utasítások gyűjteményét - amelyeket a felhasználó/programozó használ az adatkezelésnél - *gépi utasításkészletnek* nevezzük.

FDE mechanizmus

- Egy utasítás végrehajtásának három fő lépését a **Fetch-Decode-Execute** (FDE) mechanizmussal definiálhatjuk:
  - **Fetch:** az utasítás betöltődik a memóriából az utasításregiszterbe (regiszter-transzfer művelet)
  - **Decode:** utasítás dekódolása (értelmezése), azonosítja az utasítást
  - **Execute:** a dekódolt utasítást végrehajtjuk az adatokon, aminek eredménye visszakerül a memóriába

RTL leírás:

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

Néhány alapvető tranzakció specifikációja a következő:

- **PC → MAR** :A Program Számláló tartalma a Memória Cím Regiszterbe töltődik
- **PC+1→ PC** :A PC 1-el inkrementálódik, és PC-be visszatöltődik
- **MBR→IR** :MBR tartalma az IR-be töltődik. Ha az adatbusz megenged többszörös műveletvégzést egyidejűleg, akkor az egyes akciók összekapcsolhatók!
- **IR <3:0> → ALU** :Az információnak csak egy része, az IR regiszter 3-0 bitje töltődik az ALU-ba
- **REG[2] → MEM[MAR]** :A Regiszter 2. rekesze töltődik a Memória Cím Regiszter adott rekeszébe, a MAR által mutatott címre
- **If (carry==1) then PC-24 → PC** :Feltételes utasítások: Ha átvitel 1, akkor PC 24-el dekrementálódik, és visszatöltődik
- **Else PC+1 → PC** :egyébként 1-el inkrementálódik.

Utasítás formák:

- **Zéró-című (0 című):** (PUSH, POP, ACC) [operátor] (példa: STACK, vagy verem)
- **1-című:** [operátor],[operandus] (Példa: Egyszerű számítógép blokkdiagramja)
- **2-című:** [operátor],[operandus1],[operandus2]
- **3-című:** [operátor],[operandus1],[operandus2], [eredmény]
- **4-című:** [operátor],[operandus1],[operandus2], [eredmény],[következő utasítás]
- …

Példa: ADD utasítás RTL leírása

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)
MBR → IR	Majd az MBR-ben lévő adatot az IR-be tesszük
PC+I_len → PC	Az utasítás hosszával (I_len) növeli a PC értékét

Decode: (a dekódolást általában 0 idejűnek feltételezzük)

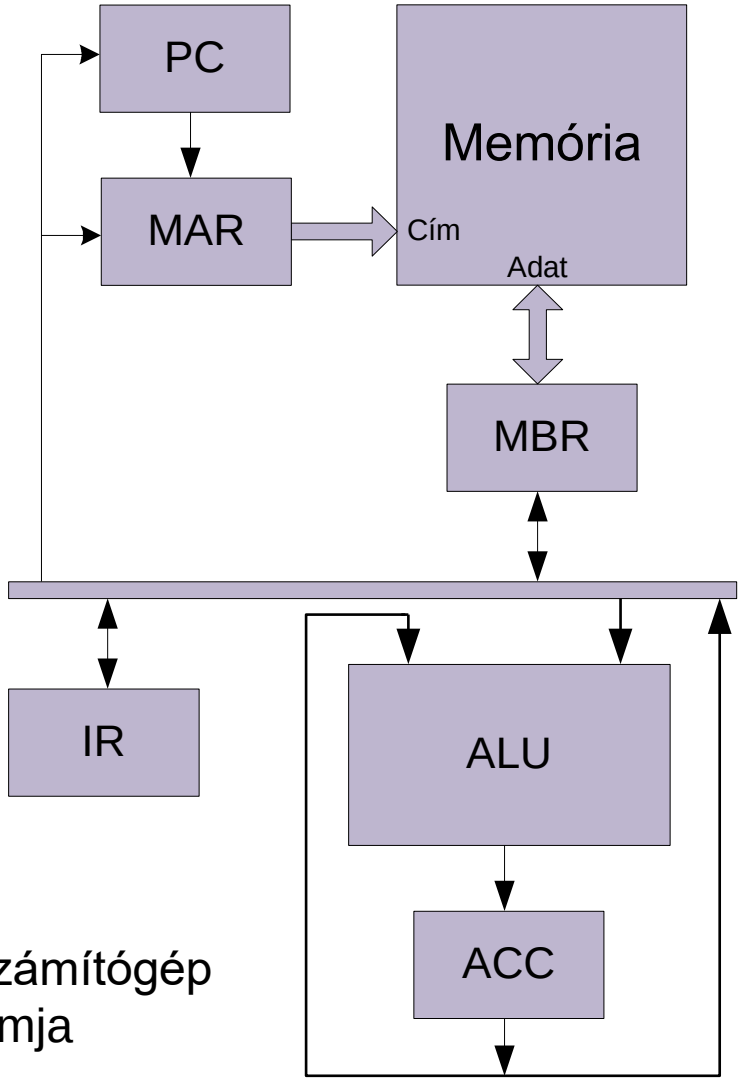
Execute:(végrehajtás)

IR <addr> → MAR	operandus címét a MAR-ba töltjük
M[MAR] → MBR	ezt az értéket kell az ACC-vel összeadni
ACC+MBR → ACC	összeadás (eredmény az ACC-ben)

Időszükségletek itt még nincsenek feltüntetve!

Egy-című gépek (Egyszerű számítógép)

Példa: PDP-8 számítógép



Egyszerű számítógép blokkdiagramja

Egy-című gép

- **Megadása:** [operátor],[operandus]
- A műveletekhez csak 1 operandus szükséges. Ilyen művelet lehet például: 1's vagy 2's komplemens képzés, inkrementálás, törlés, keresés az *ACC*-ben (akkumulátor). Az eredmény az ACC-ben tárolódik. (ACC egy olyan speciális regiszter, amelyben az aritmetikai és logikai műveletek eredménye ideiglenesen tárolódik.)
- Két operandus esetén az első operandus ACC-ben tárolt értékét használjuk fel, és a másik operandust *egyetlen címmel* azonosítjuk.

Példa: Egy-című gép
2's komplemens képzés

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	Az utasítás hosszával (I_len) növeli a PC értékét
MBR→IR	Majd az MBR-ben lévő adatot az IR-be tesszük

Decode: (a dekódolást általában 0 idejűnek feltételezzük)

Execute: (végrehajtás)

$\overline{\text{ACC}}$→ACC	ACC komplemensét az ACC-be töltjük
ACC+1→ACC	majd ACC-t 1-el inkrementáljuk (eredmény)

Időszükségletek itt még nincsenek feltüntetve!

Példa: 1-című gép
Kivonás (SUB X) egy operandusra

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	Az utasítás hosszával (I_len) növeli a PC értékét
MBR→IR	Majd az MBR-ben lévő adatot az IR-be tesszük

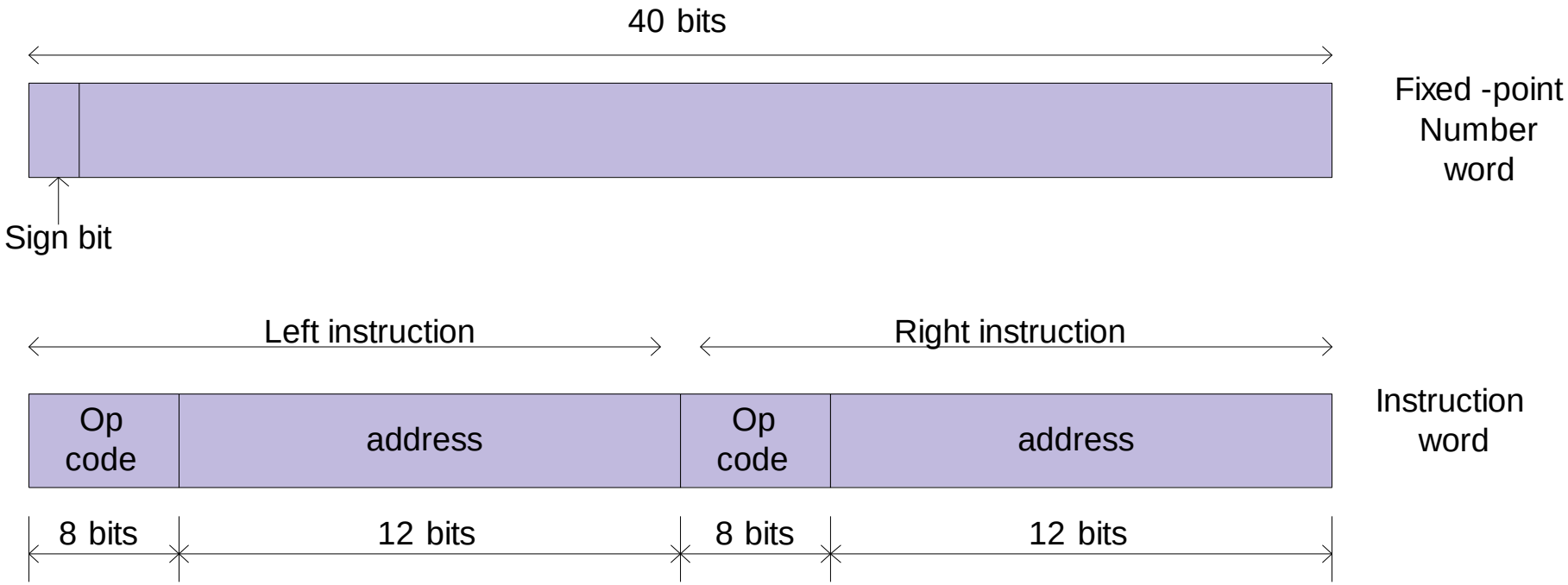
Decode: (a dekódolást általában 0 idejűnek feltételezzük)

Execute: (végrehajtás)

X→MAR	X operandus címét a *közvetlenül* a MAR-ba töltjük
M[MAR]→MBR	X címén lévő értéket az MBR-be tesszük
ACC – MBR→ACC	ACC-ből kivonjuk az X-et, és ACC-be töltjük

Időszükségletek itt még nincsenek feltüntetve!

Példa: IAS adat és utasításformátum (Neumann – 1947)



40 bites adat-szóhosszúság

2x20 bites utasításhossz: 8-bit -> 256 művelet, 12 bit-> 4096 memóriacím érhető el

Példa: 1-című gép (Kivonás SUBX)

Mostantól: „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	Az utasítás hosszával (I_len) növeli a PC értékét
MBR→IR	Majd az MBR-ben lévő adatot az IR-be tesszük

Decode: (a dekódolást általában 0 idejűnek feltételezzük)

Execute: (végrehajtás)

PC→MAR	PC-vel a következő címre mutatunk
M[MAR]→MBR	Ezt címet az MBR-be tesszük
MBR→MAR	Ez a cím lesz az X operandus címe
M[MAR]→MBR	Címen lévő értéket az MBR-be töltjük
PC+X_len →PC	X operandus címének hosszával növeljük a PC-t
ACC – MBR→ACC	ACC-ből kivonjuk az X-et, és ACC-be töltjük

Példa: 1-című gép (kivonás SUBX)

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

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: (a dekódolást általában 0 idejűnek feltételezzük)

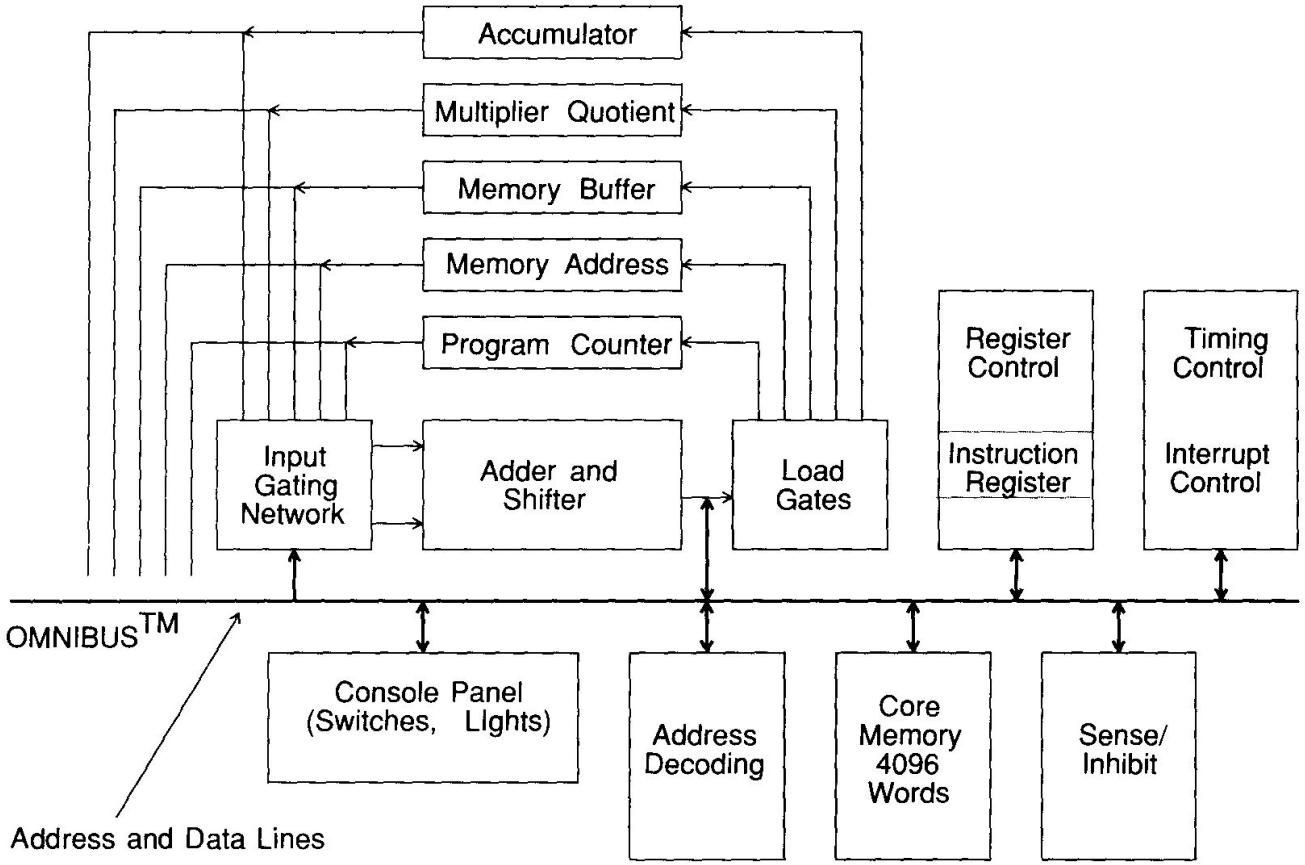
Execute: (végrehajtás)

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

Σ 135ns

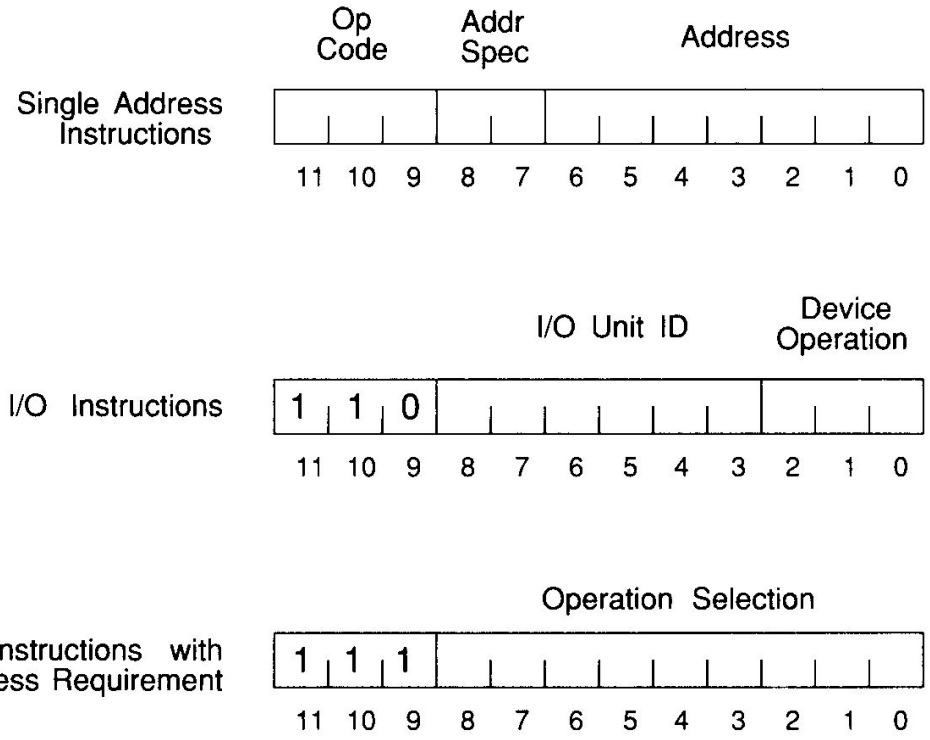
Példa: DEC PDP 8 (egycímű gép)

- 12-bites szóhossz: 3-bites opcode (8 művelet) +
  9-bit utasítás cím (speciális operandus címzési módokat tett lehetővé)



DEC PDP-8 utasításformátuma

- 3-bites opcode [11:9]: 8 művelet , ebből:
  - hatnak van saját címe (itt az alsó 9 bit adja a címet)
    pl: Add, And, Jmp, Inc_Skip_IfZero, DepositClearAcc
  - kettőnek nem kell cím: pl. ACC műveletekhez

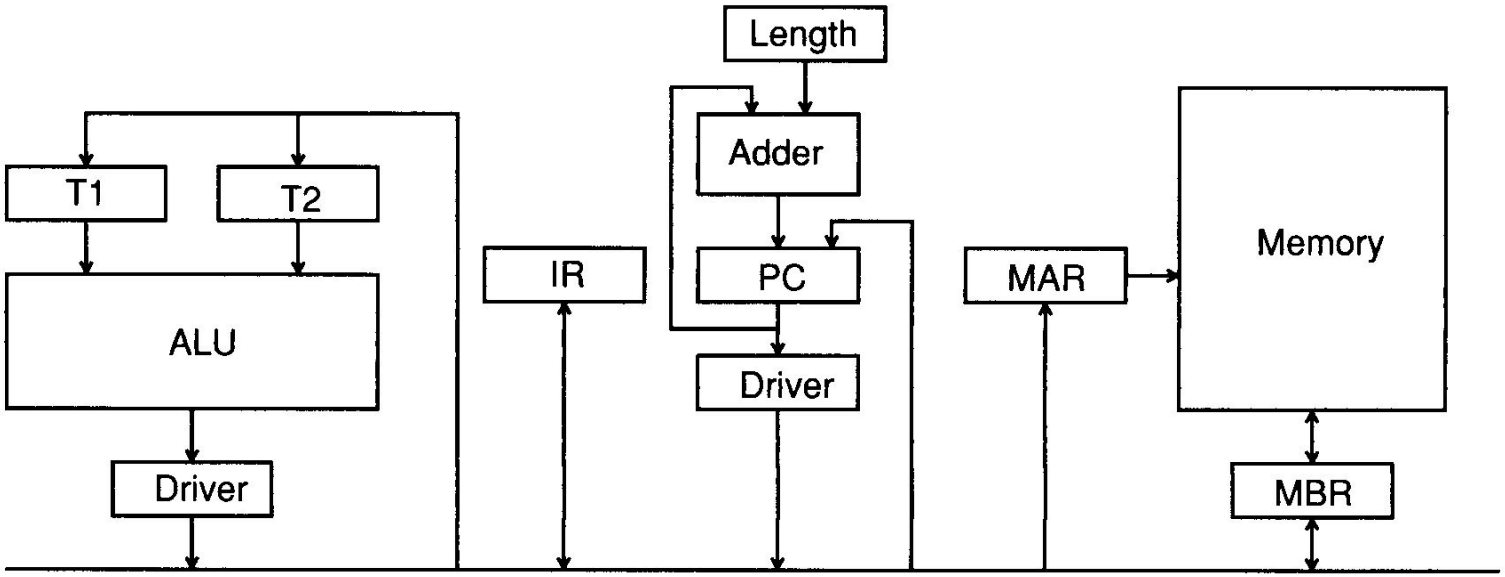


[8:3] bit -> 64 I/O eszköz

[2:0] bit: eszköz művelet

b.) Kettő- és többcímű gépek (regiszter nélküli változat)

- Egy utasítással több műveletet lehet azonosítani,
- Kevesebb utasítás-sorral, összetett módon írhatók le az RTL nyelven a folyamatok, (az egycímű gépekkel ellentétben)
- A többcímű utasítások meghatározzák, mind a *forrás*, mind a *cél*információt. A célinformáció helyét az utolsó operandus címe adja meg! [operátor],[operandus1],[operandus2]…



Jelölés: Kettő- és többcímű gép

- Jelölés: **ADD2 X, Y** kétcímű 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: **ADD3 X,Y,Z** háromcímű utasítás (*műv, op1, op2, eredmény*): hasonló az előzőhöz, csak az összeadás eredménye egy új helyen, a Z cím által azonosított helyen tárolódik el.
- Fontos megjegyezni hogy ebben az esetben (regiszter nélküli változat) a T1, ill. T2 regiszter nem az utasítás-készlet architektúra része! (ezért nem keverendő össze a később említésre kerülő regiszteres címzéssel!) Ebben az esetben csak az adatok tárolásához használjuk, nem pedig a rendszer gyorsítását kívánjuk alkalmazásukkal elérni.

Példa: Összeadás kétcímű géppel ADD2(X,Y)

Időszükségletek feltüntetésével!

T_{MEM}=30ns, T_{ALU}=10ns, T_{REG}=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: (a dekódolást általában 0 idejűnek feltételezzük)

Execute: (végrehajtás)

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

Σ 250ns

Direkt címzést használunk itt!

Példa 2: Összeadás háromcímű géppel ADD3(X,Y,Z)

Időszükségletek feltüntetésével!

T_{MEM}=30ns, T_{ALU}=10ns, T_{REG}=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: (a dekódolást általában 0 idejűnek feltételezzük)

Execute: (végrehajtás)

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

Σ 295ns

Direkt címzést használunk itt!

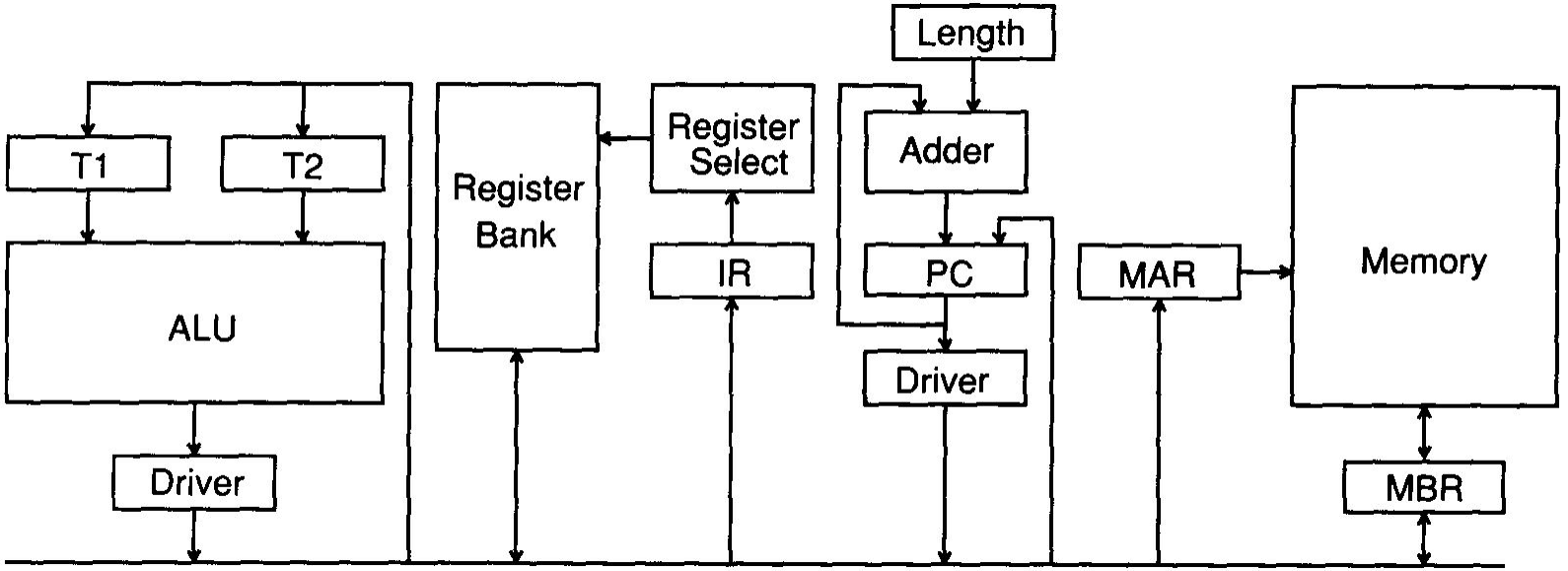
Komplex műveletek: ADD3

- Az **ADD3** végrehajtásánál (ahogy az RTL leírásból is látszik) több időt vesz igénybe az utasítások F-D-E fázisa, mint az ADD2 esetén, mivel egyel több címre kell hivatkozni.
- Azonban, az **ADD3** jelentősége a komplexebb műveletek elvégzésekor mutatkozik meg: tömörebb forma, kevesebb utasítással
- Példa: Legyen X = Y * Z + W * V (oldjuk meg ADD2-vel és ADD3-al)

ADD2	**ADD3**
MOVE Y to X	AND3 Y,Z,T
AND2 Z,X	AND3 W,V,Y
MOVE W to Y	ADD3 T,Y,X
AND2 V,Y	
ADD2 Y,X	

c.) Kettő- és többcímű gépek (regiszteres változat)

- Egy utasítással több műveletet lehet megadni,
- Kevesebb utasítás-sorral, összetett módon írhatók le az RTL nyelven a folyamatok, (az egycímű gépekkel szemben)
- A T_i regiszterek 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 ADD2(R_X,R_Y)

Időszükségletek feltüntetésével!

T_{MEM}=30ns, T_{ALU}=10ns, T_{REG}=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: (a dekódolást általában 0 idejűnek feltételezzük)

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

Σ 70ns

Regiszteres direkt-címzést használunk itt!

Példa 2: Összeadás kétcímű géppel ADD3(R_X,R_Y,R_Z)

Időszükségletek feltüntetésével!

T_{MEM}=30ns, T_{ALU}=10ns, T_{REG}=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: (a dekódolást általában 0 idejűnek feltételezzük)

Execute: (végrehajtás)

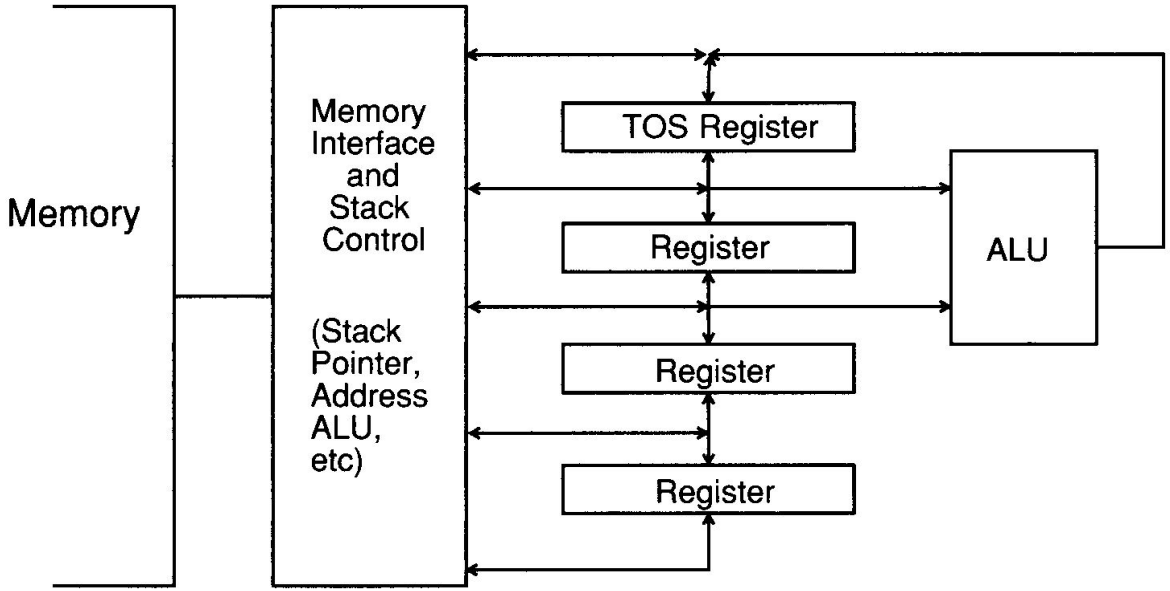
RX→T1	[5ns] **RX értékét** T1-be töltjük
RY→T2	[5ns] **RY értékét** T2-be töltjük
T1 + T2→RZ	[10+5ns] ADD2 művelet elvégzése, **RZ**-be töltjük

Σ 70ns

Regiszteres direkt-címzést használunk itt!

d.) Zéró- vagy 0-című gépek (Stack)

- Egy vermet (Stack) használunk: LIFO- típusú tároló, amelyből az utoljára betett adatot vesszük ki elsőként. A stack a memóriában található egy elkülönített részen.
- Különböző aritmetikai kifejezések hajthatók végre elég hatékonyan stack használatával: a szükséges operanduskat a stack egy-egy rekeszében tároljuk. A megfelelő operandusokat (felső kettő regiszterből) vesszük ki, elvégezzük rajtuk a műveleteket, és az eredményt a verem tetejére tesszük. Azért nevezzük **zéró címűnek**, mivel az operandusok azonosítására szolgáló utasításhoz nem használunk címeket. Az ábra a HW orientált stack rendszert ábrázolja.



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 a végrehajtáshoz: **PUSH, POP**, **ADD, DIVIDE, MULTIPLY**

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

•Az **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.

•A **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.

1. módszer: (arit. kif. elejétől haladva)	**2. módszer: (arit. kif. végétől visszafelé 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

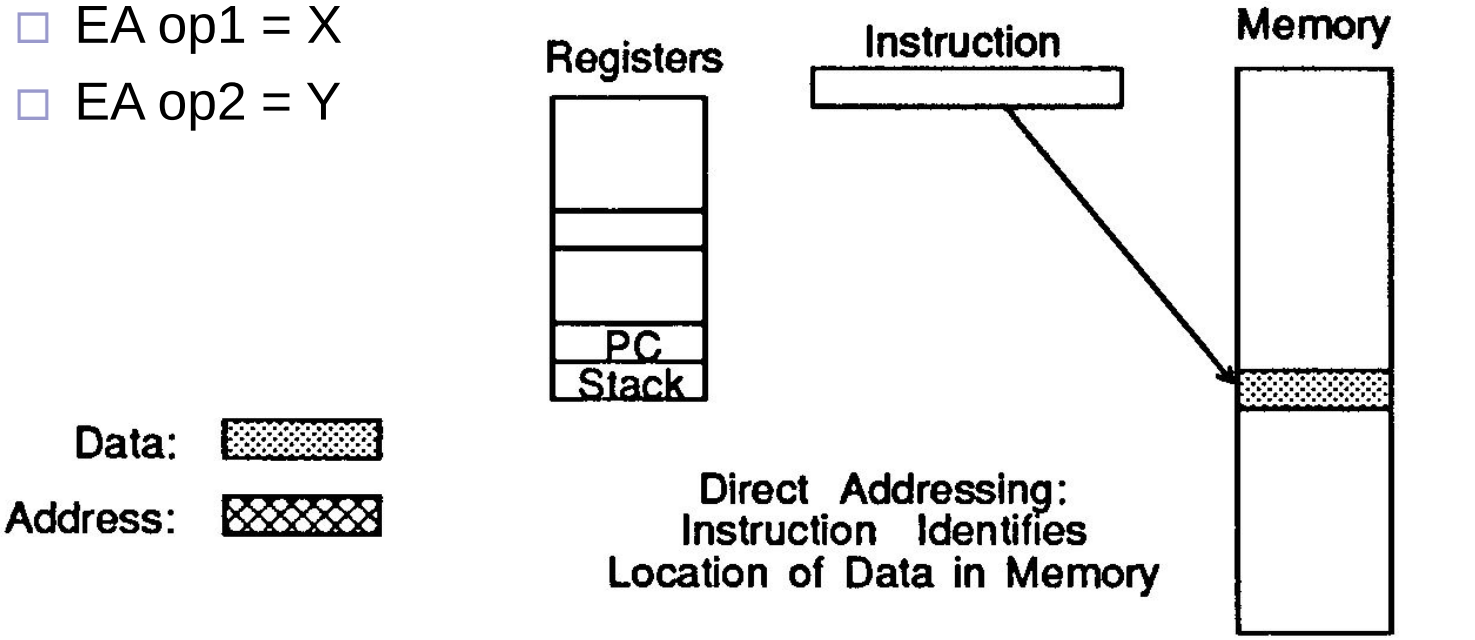
Operandus címzési módok

Operandus címzési módok

- Utasítás végrehajtásakor a kívánt operandust el szeretnénk érni, címével hivatkozhatunk a pontos helyére, azonosítjuk őt.
- Többféle címzési mód is létezik:
  - közvetlen (directed),
  - közvetett (indirected),
  - indexelt (indexed),
  - regiszteres megvalósítású (register relative).
- Ezek kombinációja igen sokféle, összesen 10-féle azonosítási módot tesz lehetővé.
- Jelölés: EA= Effektív (valódi) címe egy operandusnak

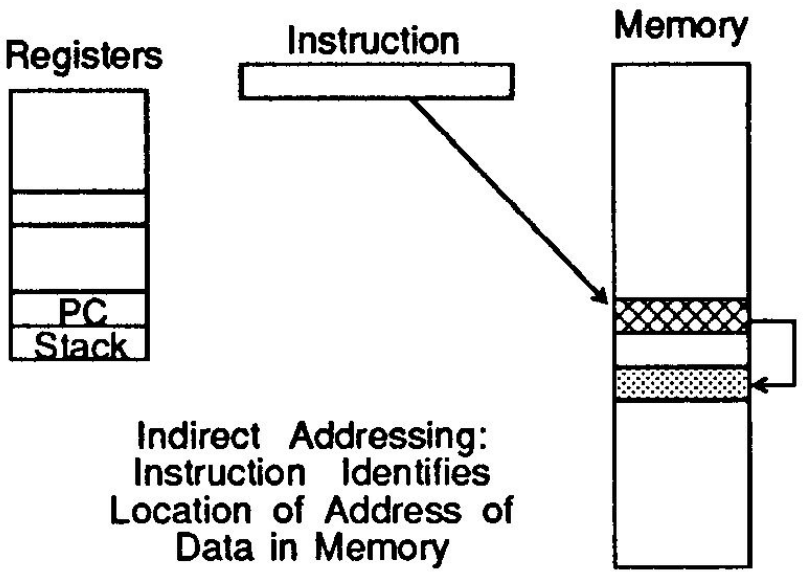
1. Direkt címzés (X)

- Az utasítás egyértelműen, közvetlenül azonosítja az operandus helyét a memóriában. (effektív cím EA= valódi címén tárolt érték) Jel: EA=A.
- **Jel: ADD2 X,Y** (X-ben tárolt op1 értéket hozzáadjuk az Y-ban tárolt op2 értékhez, az eredmény az Y-ban lesz.)
  - EA op1 = X
  - EA 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: **ADD2 *X,*Y (*: indirekció)**
  - EA op1 = MEM[X]
  - EA op2 = MEM[Y]
- Ezt különböző gyártók többféleképpen jelölik. Általában az indirekt címzési módot (*)-al jelölik:
- Példa: **ADD2 *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!



Direkt-indirekt kombináció is lehetséges:

Pl: ADD2 X,*Y

Példa: Összeadás kétcímű géppel ADD2(*X,*Y)

Időszükségletek feltüntetésével!

T_{MEM}=30ns, T_{ALU}=10ns, T_{REG}=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: (a dekódolást általában 0 idejűnek feltételezzük)

Execute: (végrehajtás)

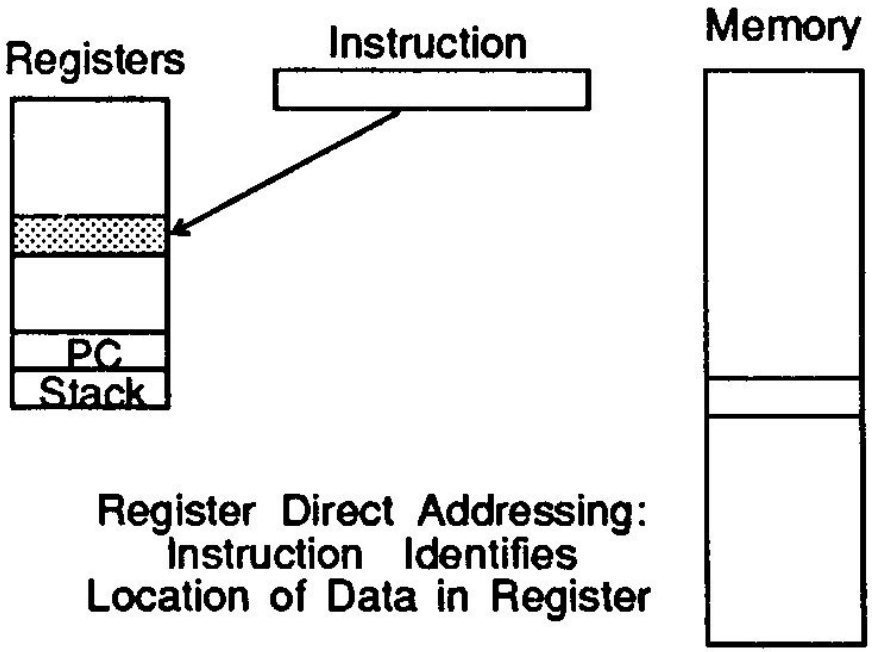
PC→MAR	[5ns] PC-vel a következő (**X**) *címének címére* mutatunk
PC+X_Alen →PC	[5ns] X operandus címének hosszával növeljük a PC-t
M[MAR]→MBR	[30ns] *X* címének címét az MBR-be írjuk
MBR→MAR	[5ns] Ezt a címet a MAR-ba töltjük
M[MAR]→MBR	[30ns] X *címét* megkapjuk az MBR-ben
MBR→MAR	[5ns] X címét a MAR-ba töltjük
M[MAR]→MBR	[30ns] X címén lévő *értékét* megkapjuk MBR-ben
MBR→T1	[5ns] **X értékét** T1-be töltjük
PC→MAR	[5ns] PC-vel a következő (**Y**) *címének címére* mutatunk
PC+Y_Alen →PC	[5ns] Y operandus címének hosszával növeljük a PC-t
M[MAR]→MBR	[30ns] Y címének címét az MBR-be írjuk
MBR→MAR	[5ns] Ezt a címet a MAR-ba töltjük
M[MAR]→MBR	[30ns] Y *címét* megkapjuk az MBR-ben
MBR→MAR	[5ns] Y címét a MAR-ba töltjük
M[MAR]→MBR	[30ns] Y címén lévő *értékét* megkapjuk MBR-ben
MBR→T2	[5ns] **Y értékét** T2-be töltjük
T1 + T2→MBR	[10+5ns] ADD2 művelet elvégzése, MBR-be töltjük
MBR→M[MAR]	[30ns] Eredményt a memóriában tároljuk el (ahol Y volt)

Indirekt címzést használunk itt!

Σ 320ns

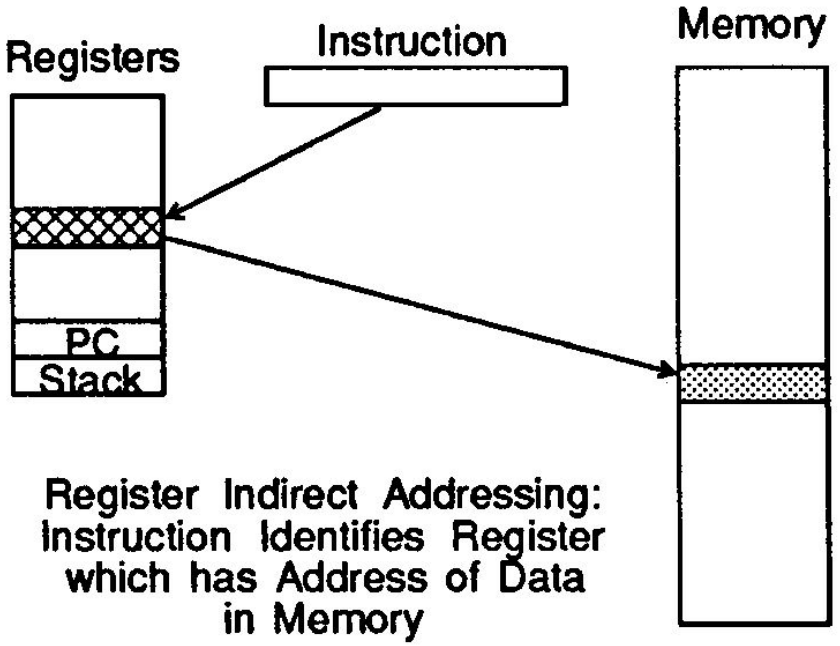
3. Regiszteres direkt címzés (R_X)

- Hasonló, mint a direkt címzés, de sokkal gyorsabb, mivel a memória intenzív-műveletek helyett a köztes eredményeket a gyors regiszterekben 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 eredményeket a gyors regiszterekben tárolja, és csak a végén tölti át a memóriába. A 3.) regiszteres módszer után ez a második leggyorsabb.



Példa: Összeadás kétcímű géppel ADD2(*R_X,*R_Y)

Időszükségletek feltüntetésével!

T_{MEM}=30ns, T_{ALU}=10ns, T_{REG}=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: (a dekódolást általában 0 idejűnek feltételezzük)

Execute: (végrehajtás)

Regiszteres indirekt-címzést használunk itt!

RX→MAR	[5ns] **RX címét** a MAR-ba töltjük
M[MAR]→MBR	[30ns] Kinyerjük az **RX** címén lévő **értéket**, amit MAR-ba 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 MAR-ba 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

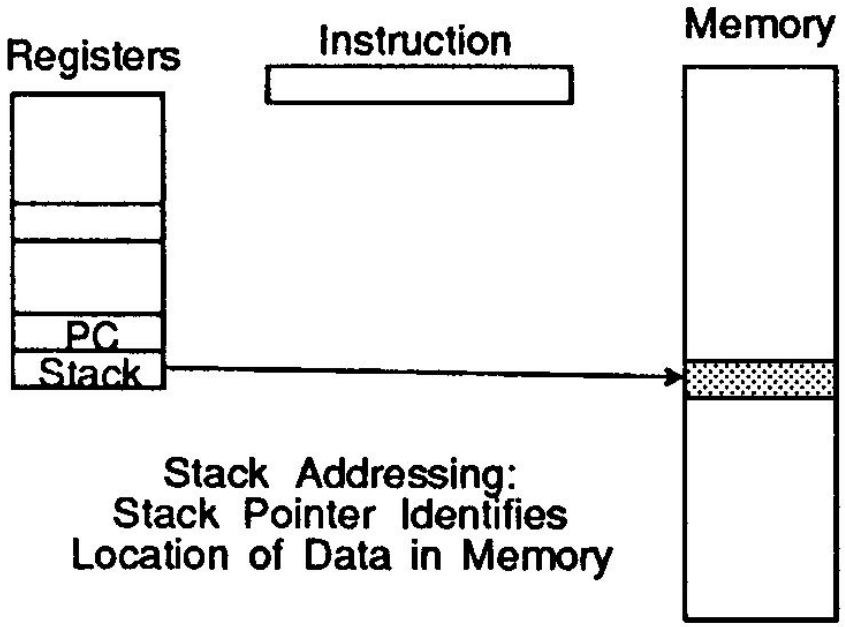
Összehasonlító táblázat – I.

- 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

- **<u>Verem (STACK) címzés, vagy regiszteres indirekt autoincrement címzési mód</u>:** *indirekt* módszerrel az operandus memóriában elfoglalt helyét a címével azonosítjuk, és akár az összes memóriatömbben lévő elem megcímezhető. *Autoincrement* is, mivel a címeket automatikusan növeli. Ezt (+) jellel jelöljük: ***Rx+**
- Ezt a mechanizmust használjuk a verem esetében. A Stack-et a memóriában foglaljuk le. A stackben lévő információra a stack pointerrel (SP-mutatóval) hivatkozunk. A stack egy *LIFO tároló*: amit utoljára tettünk be, tehát ami a verem tetején van, azt vehetjük ki legelőször.
- A stack pointer címe jelzi a TOS verem tetejét, ahol a hivatkozott információ található, ill. címmel azonosítható a következő elérhető hely. A stack (az ábra szerint) lefelé növekszik a memóriában.

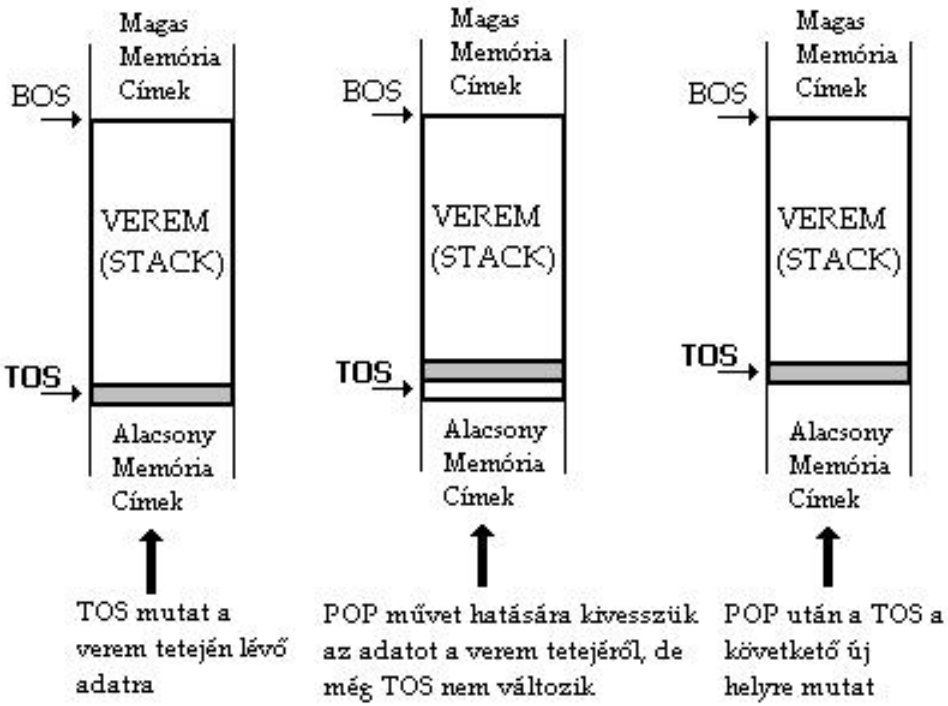


Verem – PUSH, POP műveletek

- **POP művelet** kivesszük 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**

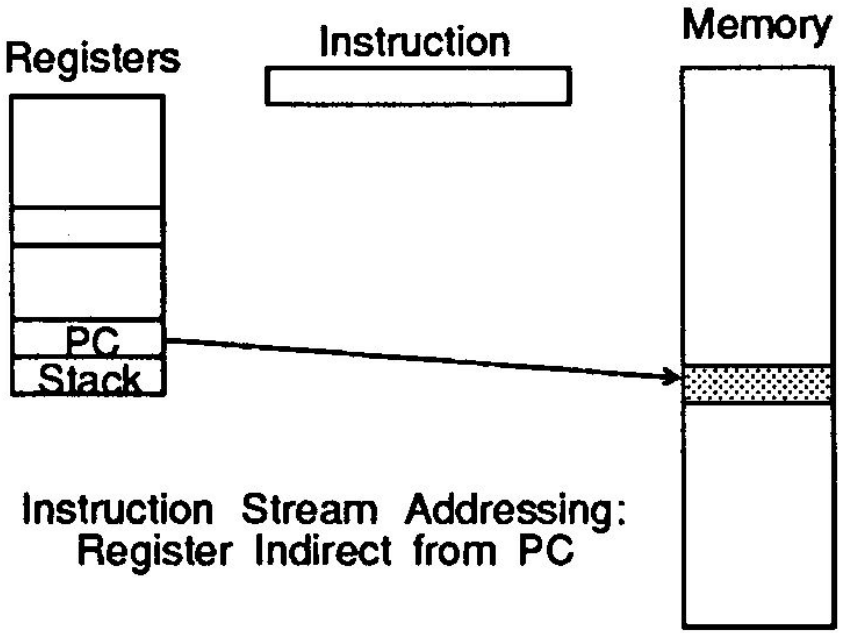
- **PUSH művelet:** a stack pointer automatikusan dekrementálja a címet, amivel a verem tetején lévő elemet azonosítja. Majd ezután berakjuk az Rx regiszterben lévő elemet a stack tetejére, a pointer átlat mutatott címre.
  Jel: **MOVE Rx, *R (stack pointer)–**

Példa: POP műveletre



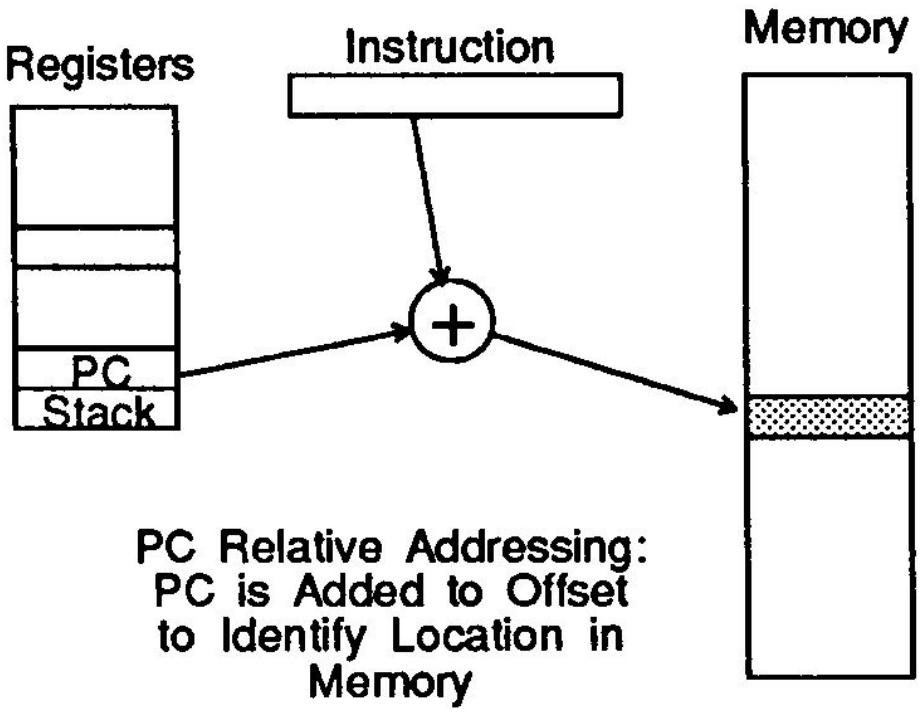
6. Instruction Stream címzés

- A PC közvetlenül azonosítja a memóriában lévő adatot és címet. Az utasítás végrehajtásakor visszakapjuk az utasításfolyamból magát az utasítást, amelyet a *PC* azonosít. Nevezik még *azonnali módszernek* is, mivel az adatok és címek azonnal a rendelkezésünkre állnak. Konstansok, előredefiniált címek szerepelhetnek az utasítás-folyamban.



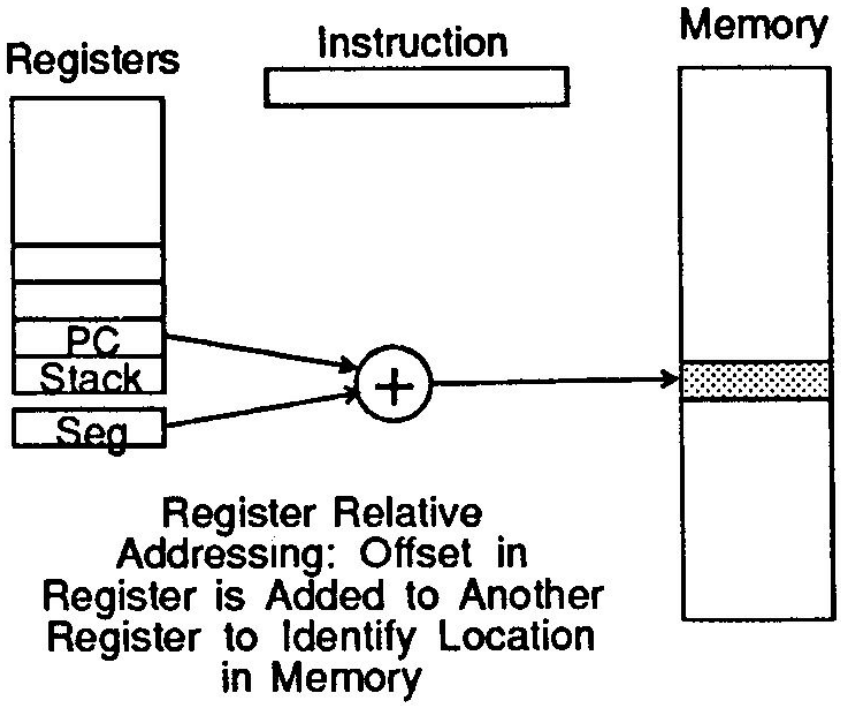
7. PC relatív címzés

- Memóriabeli adatot a regiszteren belüli PC értéke, és az utasítás eltolási (offset) értéke azonosítja.
  - Effektív cím = Regiszteren belüli PC értéke + Eltolás (offset) értéke



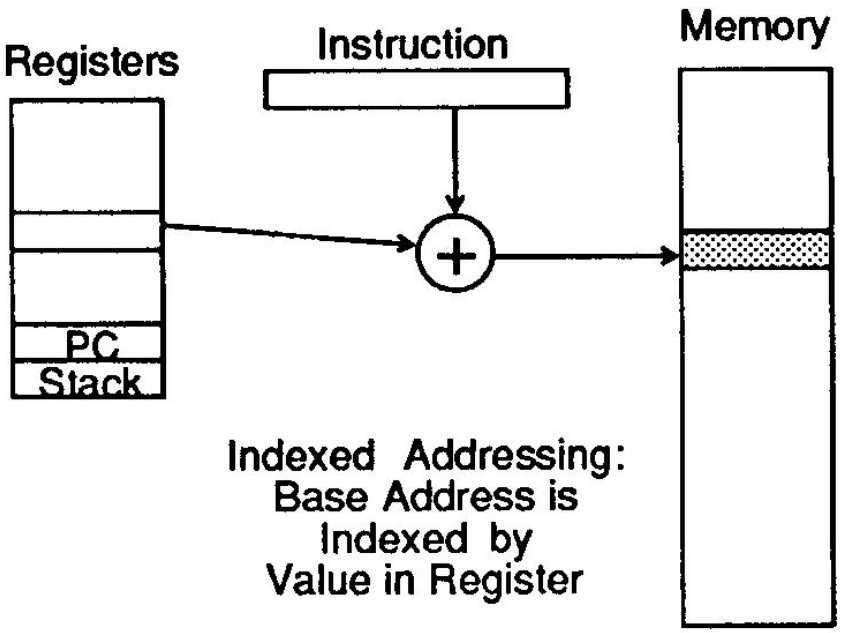
8. Regiszter relatív címzés

- A PC-regiszter eltolási értékéhez hozzáadódik *egy vagy 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



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



Összehasonlító táblázat – II.

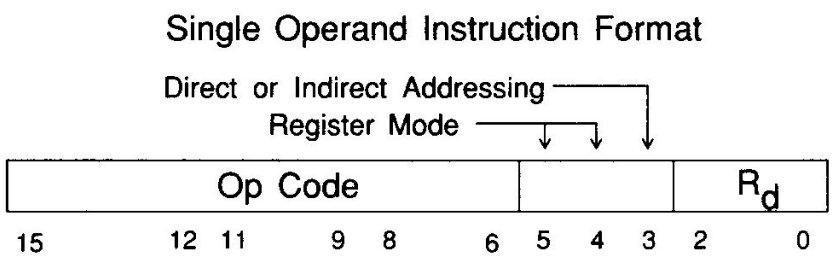
- Címzési módok és jelöléseik összefoglaló táblázata

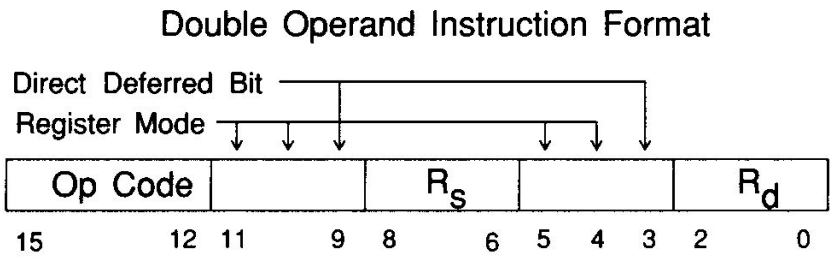
Table 4.1. Addressing Modes and their Nomenclatures.

Addressing Mode	*Represented By*	*Comment*
Direct	@<address>	Address is part of instruction.
Register Direct	Rn	Operand in 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 indentifies 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

- Különböző címzési módokat használt egyszerre
- 16-bites gép: utasítás opcode-ja + további infók (cím)
  - 3 biten: 8 általános célú regiszter ([2:0]) – **Rd**-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





PDP11 utasítás-kódolása

Table 4.2. Encoding of Instructions for the PDP 11 Architecture.

Op Code	*Function Performed*
0 0 0 0	Single address and special function instructions
0 0 0 1	Move instruction
0 0 1 0	Compare instruction
0 0 1 1	Bit test instruction
0 1 0 0	Bit clear instruction
0 1 0 1	Bit set instruction
0 1 1 0	ADD2 instruction
0 1 1 1	Single address instructions
1 0 0 0	Single address and special function instructions
1 0 0 1	Move instruction (byte)
1 0 1 0	Compare instruction (byte)
1 0 1 1	Bit test instruction (byte)
1 1 0 0	Bit clear instruction (byte)
1 1 0 1	Bit set instruction (byte)
1 1 1 0	Subtract instruction
1 1 1 1	Special purpose instructions

Dupla operandusú címzés esetén a maradék 4-biten [11…9] ig azonosítja az utasításokat.

11. 9. bit

PDP11 címzésének kódolása

Table 4.3. Encoding of Addressing Information in the PDP 11 Architecture.

Addressing modes for PDP 11 operands

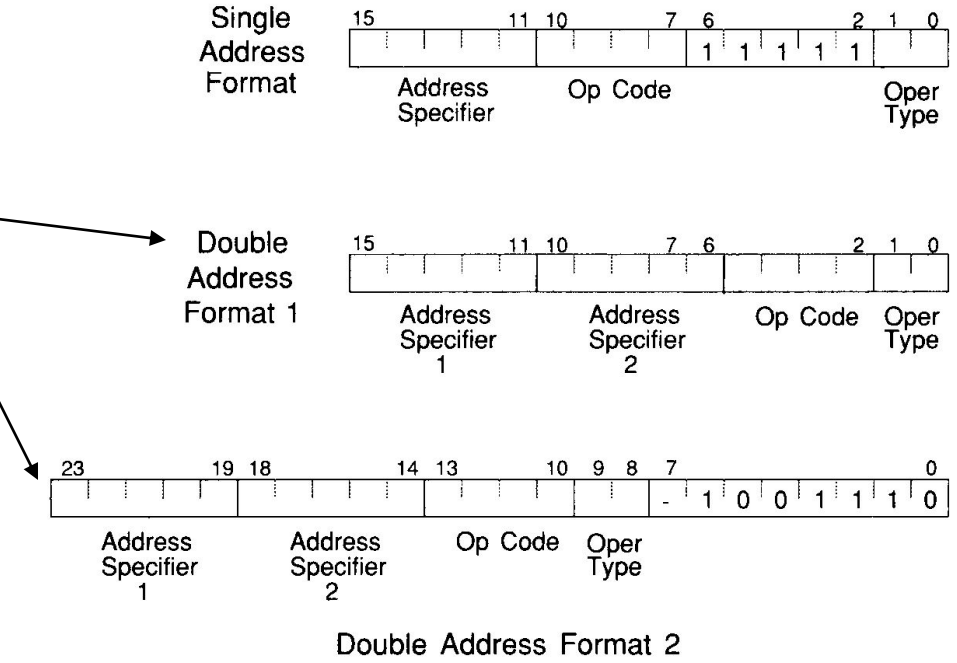
Addr bits		*Addressing mode*
0	0 0	Register direct
0	0 1	Register indirect
0	1 0	Register indirect — autoincrement
0	1 1	Two level indirect, autoincrement register
1	0 0	Register indirect — autodecrement
1	0 1	Two level indirect, autodecrement register
1	1 0	Indexed
1	1 1	Indexed indirect

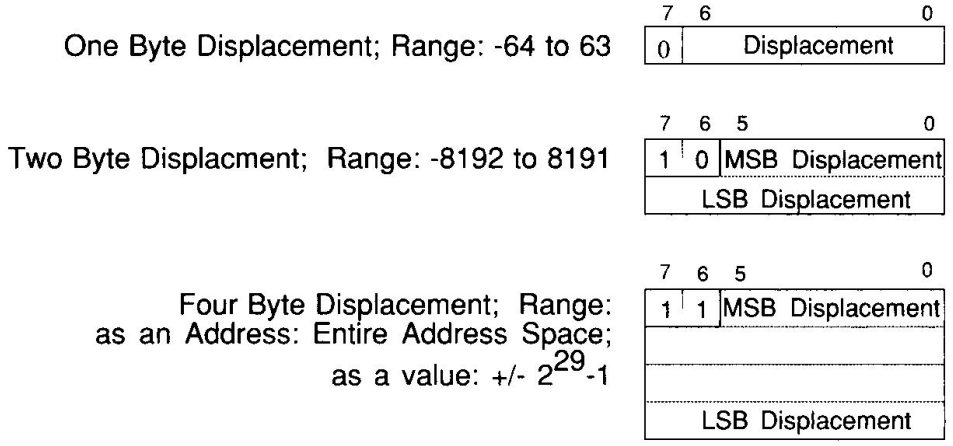
Addressing modes when PC is target register

Addr bits		*Addressing mode*
0	1 0	Immediate mode (Instruction stream)
0	1 1	PC absolute mode
1	1 0	PC relative
1	1 1	PC relative, indirect

Példa 2. NS32032 – bővíthető műveleti kód

- National Instruments – 32 bites processzora
- Egy (JUMP, JSR) és kétcímű utasítások dekódolása
  - Format 1: általános használatra (Add, Sub, Comp, Mov. Stb)
  - Format 2: bővített (Div, Test, Shift, Abs)
- Több címzési mód
- Addr. Specifier(ek): 5 biten (32 lehetséges címkombináció)
- Addr. Displacement (eltolás): 1, 2, 4 byte-on.





Programszervező utasítások

Programszervező utasítások

- Program végrehajtásának szabályozása:
  - Feltételes, feltétel nélküli utasítások (branch)
  - Szubrutin (eljárás) hívás / visszatérés
  - Ciklus (iteratív végrehatás)
  - Ugró utasítások (Jump)

JUMP utasítás RTL leírása (PDP 11 gépen)

Fetch: (regiszterek feltöltése, utasításhívások):

PC→MAR	Elsőként a PC-ből az utasítás címe (tartalmazza a cél címét is) a MAR-ba töltődik
M[MAR]→MBR	Memóriában lévő utasítás beírása az MBR-be
PC+2→PC	Az utasítás hosszával (2) növeli a PC értékét
MBR→IR	Majd az MBR-ben lévő adatot az IR-be tesszük

Decode:

Execute: (végrehajtás)

PC→MAR	PC-vel a cél (ugrás) címét szeretnénk kiolvasni
M[MAR]→MBR	Ezt az ugrási címet az MBR-be tesszük
MBR→PC	Aktualizáljuk a PC értékét (program számláló adott pontra fog mutatni)

IF feltételes elágazás RTL leírása (PDP 11 gépen)

Fetch: (regiszterek feltöltése, utasításhívások):

PC→MAR	Elsőként a PC-ből az utasítás címe (tartalmazza a cél címét is) a MAR-ba töltődik
M[MAR]→MBR	Memóriában lévő utasítás beírása az MBR-be
PC+2→PC	2-vel növeli a PC értékét
MBR→IR	Majd az MBR-ben lévő adatot az IR-be tesszük

Decode:

Execute: (végrehajtás)

IF (carry == 1)	feltételes elágazás (itt a carry bit tartalmától függően)
{	Feltétel ellenőrzés, ha Igaz akkor…
PC+(2xIR<7:0>)→PC	PC-vel a következő utasítás (cél) címére mutatunk (PC+offset)
}	IR<7:0> : 8 bites offset cím (előjel kiterjesztés 16-bitre ,páros szóhatárra)
ELSE	
{	
PC+2→PC	Ha a feltétel Hamis, megváltoztatjuk PC címét
}	

Call (szubrutinra hívás) RTL leírása (PDP 8 gépen)

Fetch: (regiszterek feltöltése, utasításhívások):

PC→MAR	Elsőként a PC-ből az utasítás címe (tartalmazza a cél címét is) a MAR-ba töltődik
PC+1→PC	Az utasítás hosszával növeli a PC értékét
M[MAR]→MBR	Memóriában lévő utasítás beírása az MBR-be
MBR→IR	Majd az MBR-ben lévő adatot az IR-be tesszük

Decode:

Execute: (végrehajtás)

IR→MAR	Feltételezzük, hogy utasítás már tartalmazza a címet
PC→ M[MAR]	A visszatérési címet a szubrutin elejére fogja tenni (indirekt ugrás)
IR→PC	Aktualizáljuk a PC értékét (a visszatérési címet fogja tartalmazni)
PC+1→PC	A szubrutin első utasítására fogunk mutatni PC-vel (*hívás*)

„Subroutine linkage”: biztosítja hogy az alprogram végrehajtása után a hívó (call) eljáráshoz vissza tudjunk térni (return)

Return (visszatérés) RTL leírása (PDP 8 gépen)

Fetch: (regiszterek feltöltése, utasításhívások):

PC→MAR	Elsőként a PC-ből az utasítás címe (tartalmazza a cél címét is) a MAR-ba töltődik
PC+1→PC	Az utasítás hosszával növeli a PC (nem aktuálisan használt) értékét
M[MAR]→MBR	Memóriában lévő utasítás (ugrás) beírása az MBR-be
MBR→IR	Majd az MBR-ben lévő adatot az IR-be tesszük

Decode:

Execute: (végrehajtás)

IR→MAR	Feltételezzük, hogy utasítás már tartalmazza a címet (szubrutin elejét azonosítja ez a cím)
M[MAR]→MAR	A visszatérési címet tesszük a MAR-ba (szubrutin elején található)
M[MAR]→PC	Aktualizáljuk a PC értékét (program számláló a visszatérési címre fog mutatni, amely után a főprogram folytatja feladatát, a szubrutinból való visszatéréskor) - *visszatérés*

„Subroutine linkage”: biztosítja hogy az alprogram végrehajtása után a hívó (call) eljáráshoz vissza tudjunk térni (return)

PDP-8 Call, Return utasításai

- Problémák:
  - Memóriát használ (lassú)
  - Egy időben csak egy hívó rutin „hívhatja” a szubrutint (visszatérési cím mindig a memória egy adott részén helyezkedik el) → nem biztosít „Time Sharing” üzemmódot
  - Nem lehet rekurzió (egymásba ágyazott hívás)
  - ROM-ot alkalmazó rendszerekben nem működik
- **Megoldás:** visszatérési cím tárolása általános célú regiszterben , Stack-et használ (SP)
  - Példa: Motorola 68000 **JSR, RTS** műveletekkel

További programvezérlési (program-control) módszerek

- I/O vezérlés:
  - memory-mapped I/O
  - DMA: Direct Memory Access (chapter6.pdf)
- Megszakítás (interrupt) [IRQ]
  - SW (nem/maszkolható interrupt)
  - HW
- Trap (csapda): programvezérelt megszakítás
  - Exception: kivétel

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
- CISC, RISC architektúrák

RISC processzorok jellemzői (1):

- Például: Motorola 88000 RISC rendszere, vagy Berkeley Egyetem RISC-I rendszere, Alpha, SPARC, PIC, DSP sorozatok, IBM PowerPC stb.
- **RISC:** Reduced Instruction Set Computer (Csökkentett utasításkészletű számítógép):
- Csak a kívánt alkalmazásra jellemző utasítástípusokat tartalmaz, az utasításkészlet összetettségének csökkentése végett kihagytak 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)

RISC processzorok jellemzői (2):

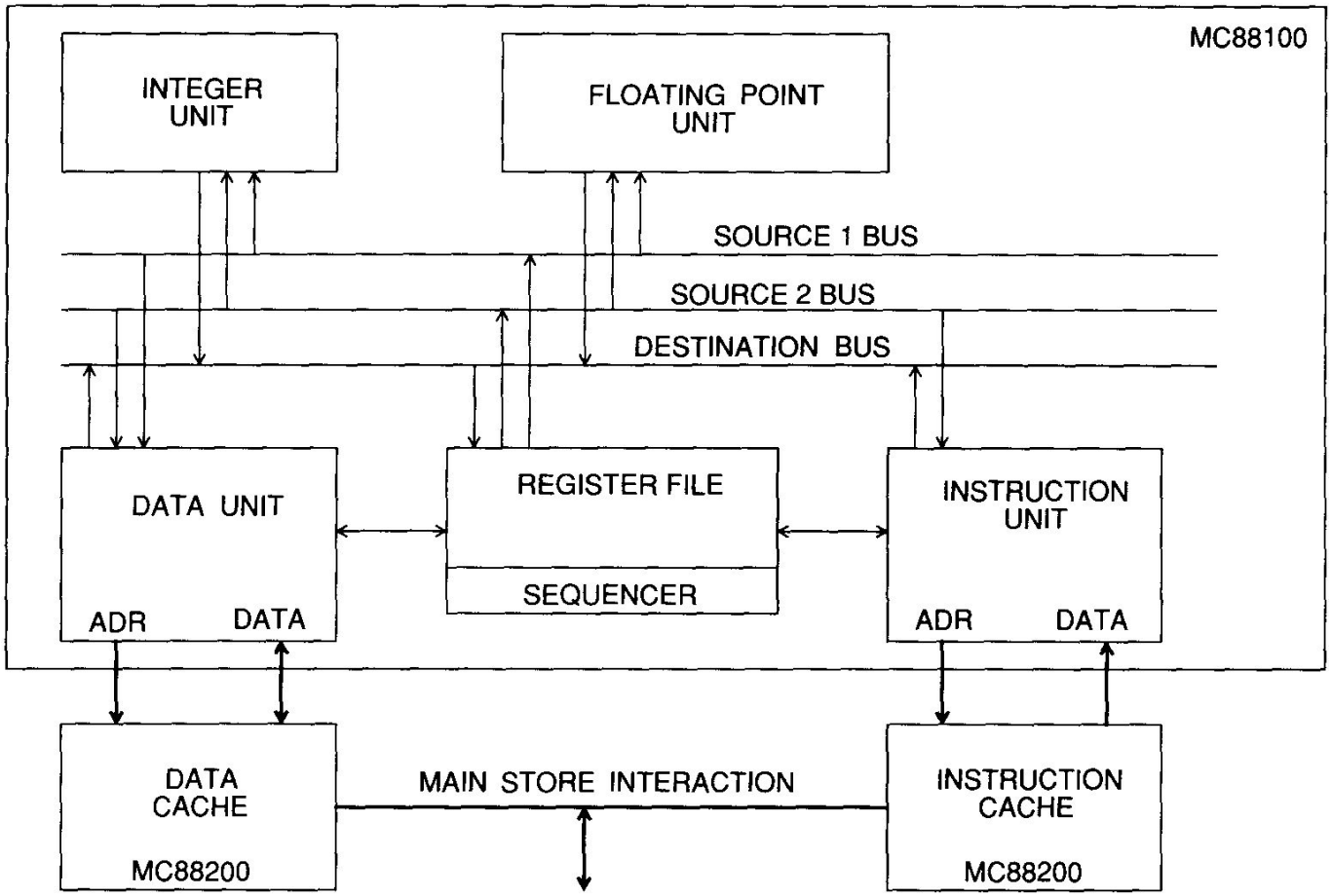
- *Huzalozott (hardwired) utasításdekódolás* (a hardveres dekódolás megvalósításához kombinációs logikát használ, azonban a mai memóriaalapú mikro-kódú gépeknél ez lassabb).
- *Egyszeres ciklusvégrehajtás*: (minden egyes ciklusban egy utasítást hajt végre, ha ezt sikerülne elérni optimális lenne az erőforrás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

RISC: Pipeline technika

- A soros feldolgozással ellentétben, egy feladat egymástól független részei (fázisai) a rendszer különböző pontjain egyszerre, egyidőben hajtódnak végre, ezáltal növekszik a sebesség. Az operandusokat gyors regiszterekben tároljuk. Azonos hosszúságú utasítások gyors F-D-E eljárása. Egy ciklusban egyszerre történik különböző utasításrészek 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. SourceI-1,2, Destination adatbuszok a Motorola 88000 rendszerben.)

	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

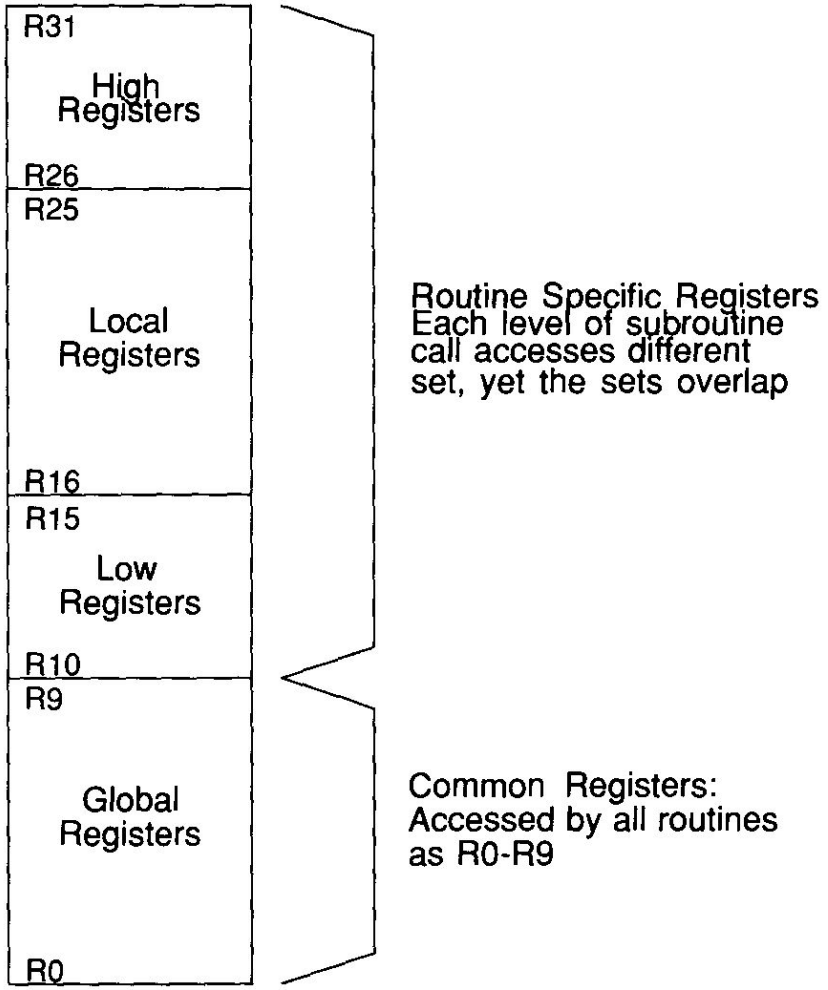


Hardveres párhuzamosítás:

- Buszok
- Cache
- Data / Instruction Unit

Berkeley RISC-I rendszere: Regiszter használat

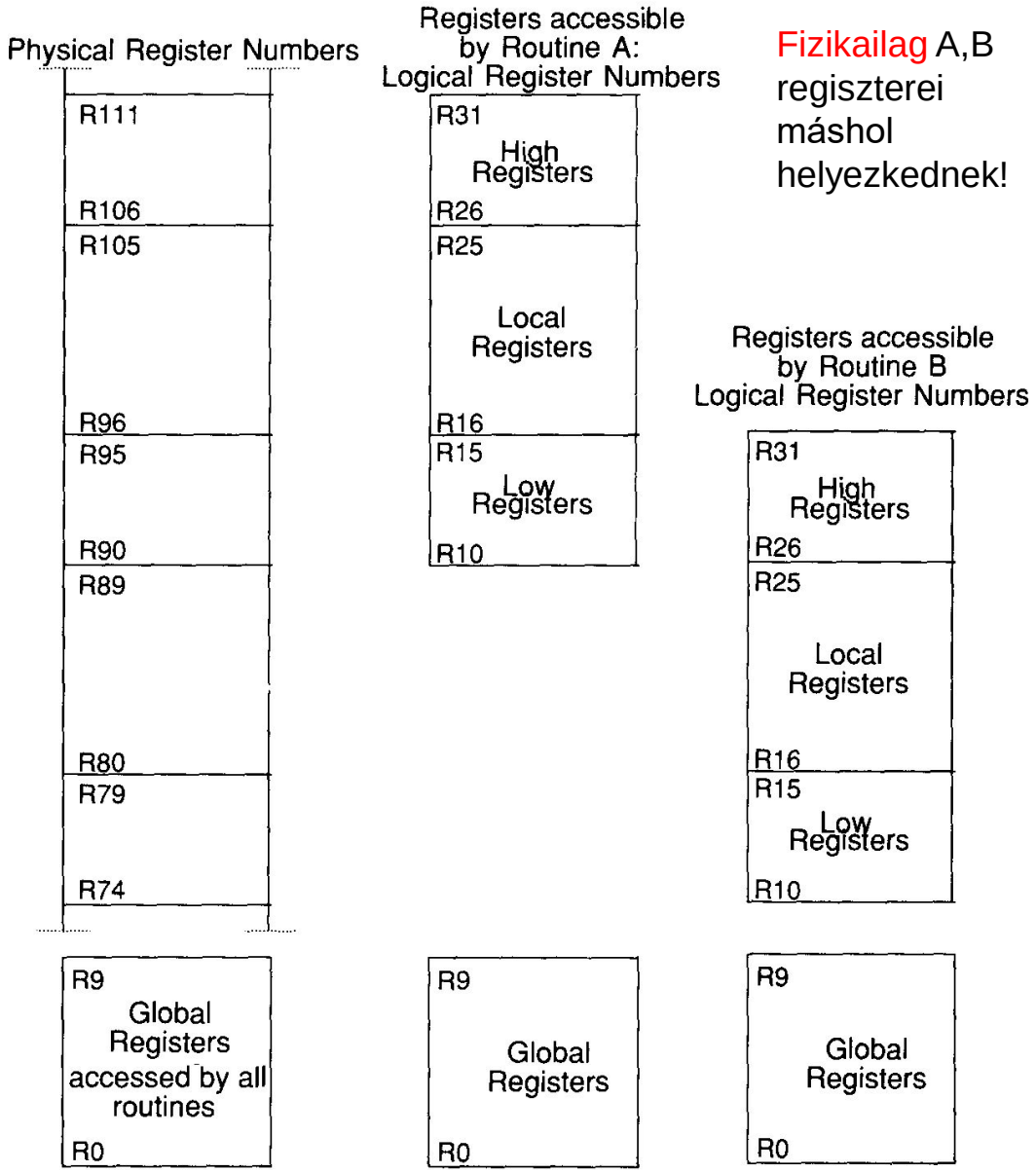
- Regiszter ablak: a szubrutin híváshoz (call) / visszatéréshez (ret) szükséges processzor-időt kívánták minimalizálni nagy számú regiszter használatával.
- Regisztereknek csak egy kis része érhető el („ablak”). Egy pointer mutat az ablakra, amely azonosítja a benne található aktuális regisztereket.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



„Regiszter ablakozási„ technika

- Paraméter átadás „ablakozással”: a globális regisztereken keresztül történik, amelyet mindkét (A,B) 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.
- A korai gépeknek egyszerű volt a felépítése, de bonyolult a nyelvezete. Összetett problémákat kívántak vele megoldani, a gépi kódnál magasabb szintű nyelven. *Szemantikus* rés= a gépi nyelv és felhasználó nyelve közötti különbség. Ennek áthidalására új nyelvek születtek: Fortran, Lisp, Pascal, C, amelyek bonyolultabb problémákat is egyszerűen képesek voltak 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 és összetett műveletek jellemzők rá.
- Mikro-programozott vezérlési mód
  - 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.
- PI: Példa:
  - System/360, VAX, PDP-11, Motorola 68000 family, and AMDx86 and Intel x86-32/64 CPUs

Pannon Egyetem
Képfeldolgozás és Neuroszámítógépek Tanszék



Digitális Rendszerek és Számítógép Architektúrák

4. előadás: Vezérlő egységek

Előadó: Vörösházi Zsolt
voroshazi@vision.vein.hu

Jegyzetek, segédanyagok:

- Könyvfejezetek:
  - http://www.knt.vein.hu
    ⇒ Oktatás ⇒ Tantárgyak ⇒ Digitális Rendszerek és Számítógép Architektúrák
    (chapter05.pdf)
- Fóliák, óravázlatok .ppt (.pdf)
- Feltöltésük folyamatosan

Vezérlő egységek általánosan

- A számítógép vezérlési funkcióit ellátó *szekvenciális* egység.
- **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 tervezési 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

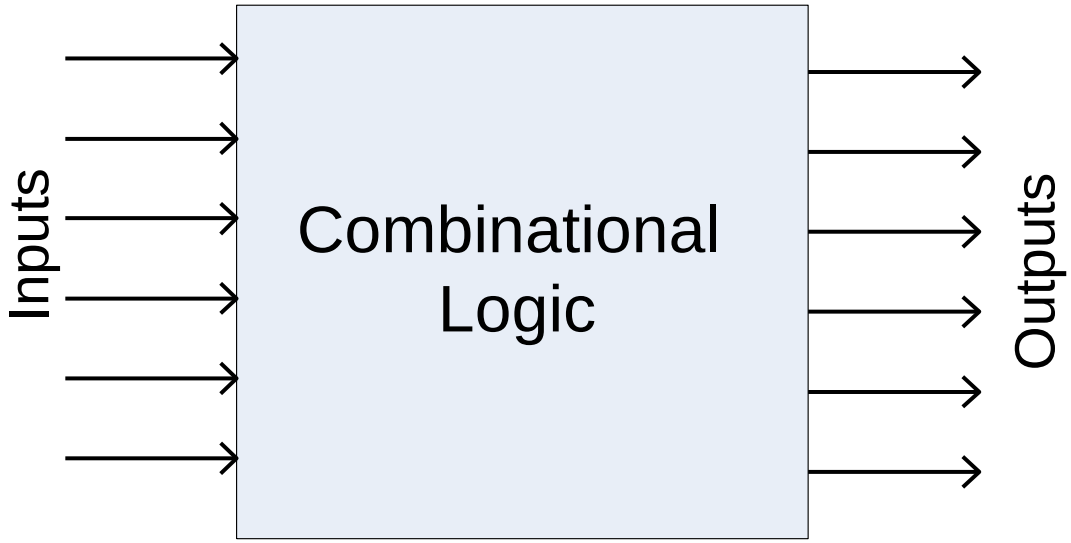
- Gazdaságosság (költség)
- Interfész szükséglet (protokollok)
- Sebesség
- Energia (disszipált teljesítmény)
- Dinamikai tartomány (számrendszerek)
- Rugalmasság (többcélúság)
- Kezelhetőség (probléma, hiba során)
- Környezet (pl. ipari v. irodai használat?)

Vezérlő egységek fajtái:

- Huzalozott (klasszikus módszerrel):
  - Mealy-modell,
  - Moore-modell (Példa: FIR szűrő tervezése – FSM állapotgép segítségével).
    - Multiplexeres / késleltetéses / Shift-regiszteres megvalósítások
- Mikroprogramozott (reguláris vezérlési szerkezettel):
  - Horizontális mikrokódos vezérlő,
  - Vertikális mikrokódos vezérlő.
- Programozható logikai eszközök (PLD):
  - PLA, PAL, PROM, CPLD,
  - FPGA!

Kombinációs hálózatok

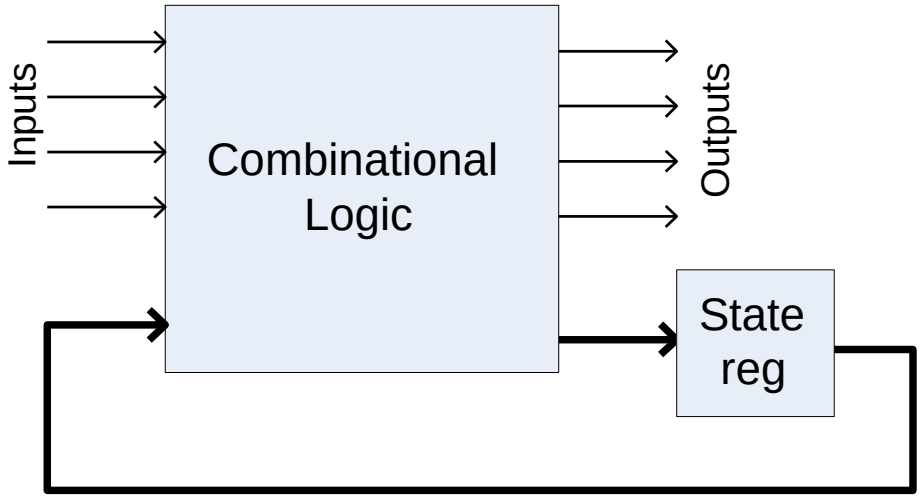
- **(K.H.) Kombinációs logikai hálózat**ról beszélünk: ha a mindenkori kimeneti kombinációk értéke csupán a bemeneti kombinációk pillanatnyi értékétől függ (tároló „kapacitás”, vagy memória nélküli hálózatok).



Sorrendi hálózatok:

- **(S.H.) Sorrendi (szekvenciális) logikai hálózat**ról beszélünk**:** ha a mindenkori kimeneti kombinációt, nemcsak a pillanatnyi bemeneti kombinációk, hanem a korábban fennállt bementi kombinációk és azok sorrendje is befolyásolja. (A *szekunder /másodlagos kombinációk* segítségével az ilyen hálózatok képessé válnak arra, hogy az ugyanolyan bemeneti kombinációkhoz 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 hálózat!



Időzítő – vezérlő egység:

- Az **időzítő (ütemező)** határozza meg a vezérlő jelek előállításának *sorrendjét*.
- Egy **időzítő-vezérlő** 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 sorrendiségét* biztosítják – melyek az aktuális *kimenet* értékét a *bemenet*, és az *állapotok* függvényében határozzák meg.

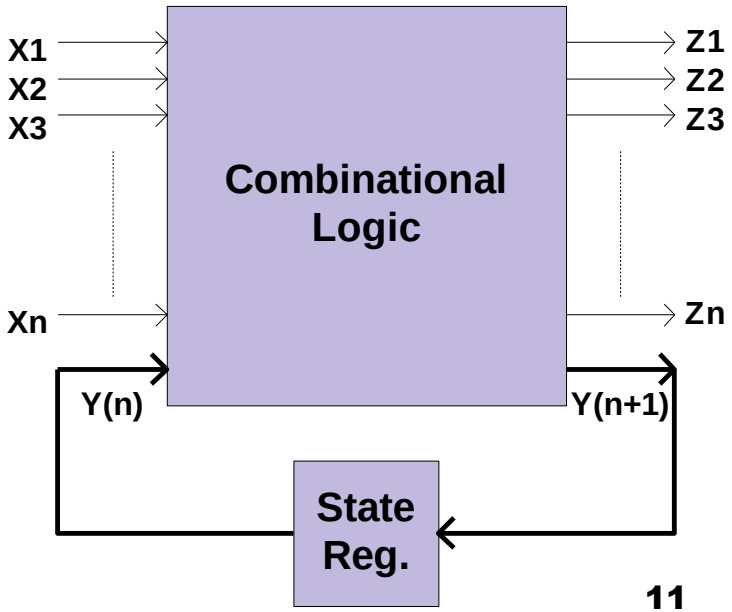
Az időzítő-vezérlő lehet:

- **huzalozott**: áramkörökkel, *dedikált* összeköttetésekkel fizikailag megvalósított (Mealy, Moore, MUX-os modellek alapján, illetve programozható PLD-k)
- **mikroprogramozott**: az adatútvonal (data-path) vezérlési pontjait memóriából (ROM) kiolvasott *vertikális*- vagy *horizontális*-mikrokódú utasításokkal állítják be

Huzalozott vezérlő egységek

1.) Mealy-modell

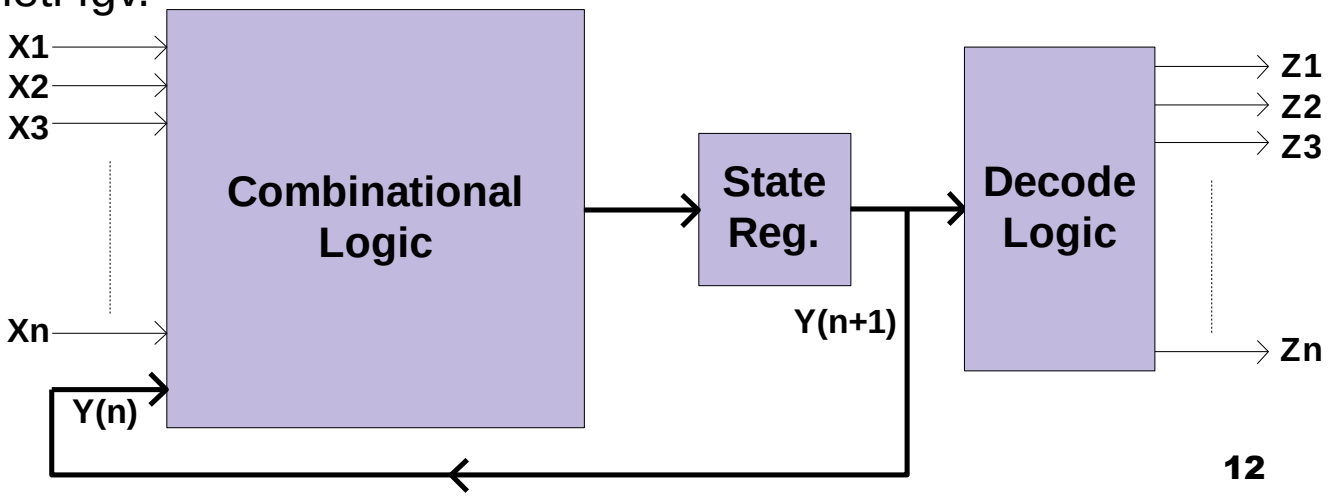
- A sorrendi hálózatok egyik alapmodellje. Késleltetés: a kimeneten az eredmény véges időn belül jelenik meg! Korábbi értékek visszacsatolódnak a bemenetre: kimenetek nemcsak a bemenetek pillanatnyi, hanem a korábbi állapotoktól is függenek. Problémák merülhetnek fel az állapotok és bemenetek közötti szinkronizáció hiánya miatt (változó hosszúságú kimenetet - dekódolás). Ezért alkalmazzuk 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:
  - $\delta(Xn,Yn) \rightarrow Yn+1$: következő állapot fgv.
  - $\mu(Xn,Yn) \rightarrow Zn$: kimeneti fgv.



2.) Moore-modell

- A kimenetek közvetlenül csak a pillanatnyi állapottól függenek (bemenettől függetlenek). 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 szabályok
  - $\delta(Xn,Yn) \rightarrow Yn+1$: köv. állapot fgv.
  - $\mu(Yn) \rightarrow Z$: kimeneti fgv.

Input ⇒ Next-State ⇒ Present-State ⇒ Output



Adatút (data-path) tervezése FIR szűrőhöz

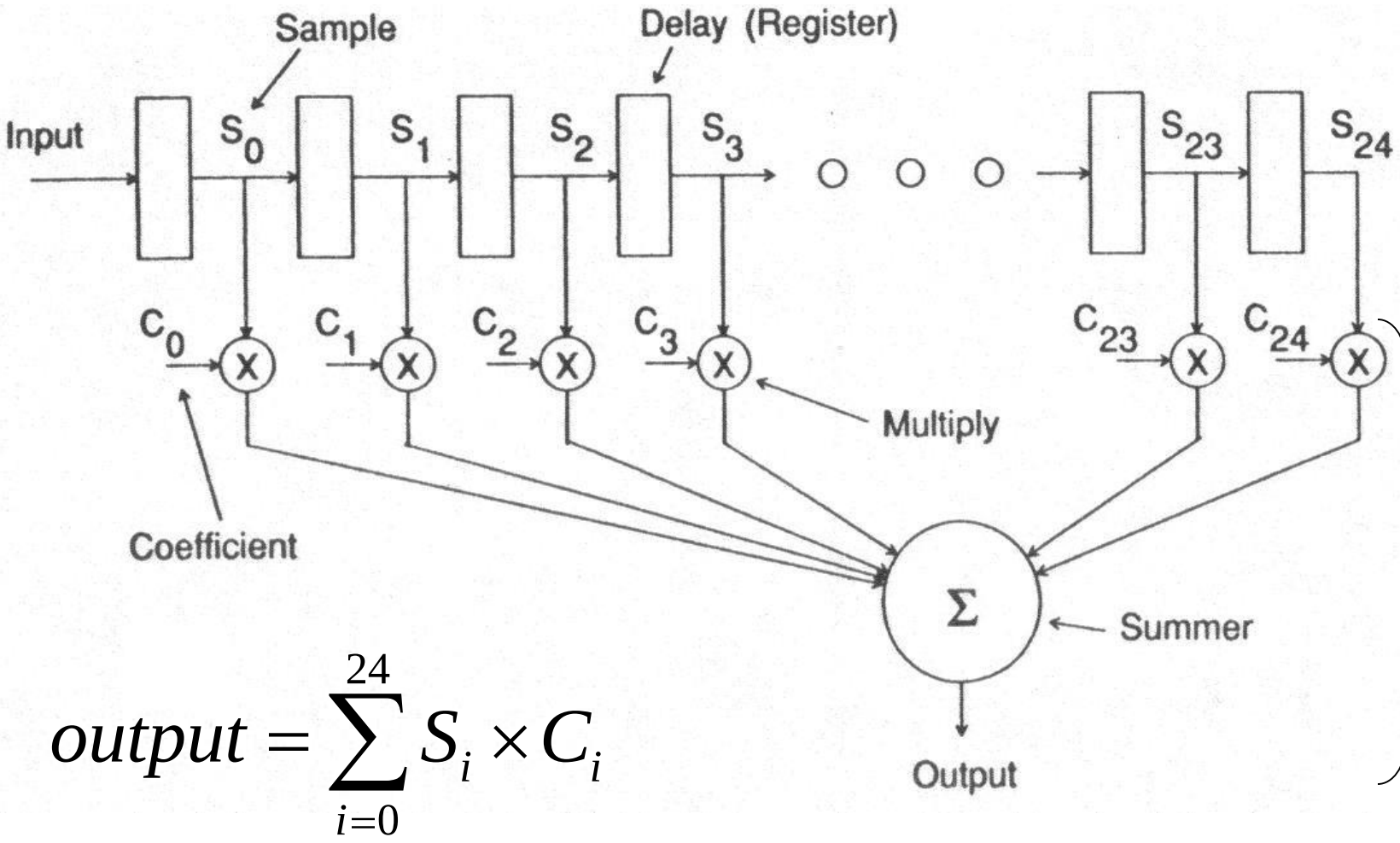
Adatút tervezés - FIR szűrő

- **FIR = Finite Impulse Response** szűrővel egy digitális rendszerben könnyen kiszámíthatók pl. skaláris/ *belső szorzat*, mátrixműveletek stb. Alkalmazzák a digitális jelfeldolgozásban (DSP) áteresztő szűrőknél. **Feladat:** tervezzünk egy FIR szűrőt, amely 25 együtthatót képes kezelni, és a kimenetét képező eredményt a következő egyenlettel kapjuk meg:

$$output = \sum_{i=0}^{24} S_i \times C_i$$

- ahol *Si* a bemeneti adatfolyam i. mintája, ill. *Ci* az i. konstans értéke. A/D konverterrel állítja elő a mintákat, ill. D/A konverterrel alakítja vissza analóg jellé, további feldolgozás szempontjából elfogadható formába. Minden egyes mintát meg kell szorozni a neki megfelelő együtthatóval (koefficiens), majd a szorzatokat össze kell adni. Egyenként 25 mintát veszünk késleltetve, (elvileg 25 különböző szorzó áramkör kellene). De ezt a feladatot a MAC egység látja el.

Példa: FIR szűrő - feldolgozás



MAC:

Multiply & Accumulate

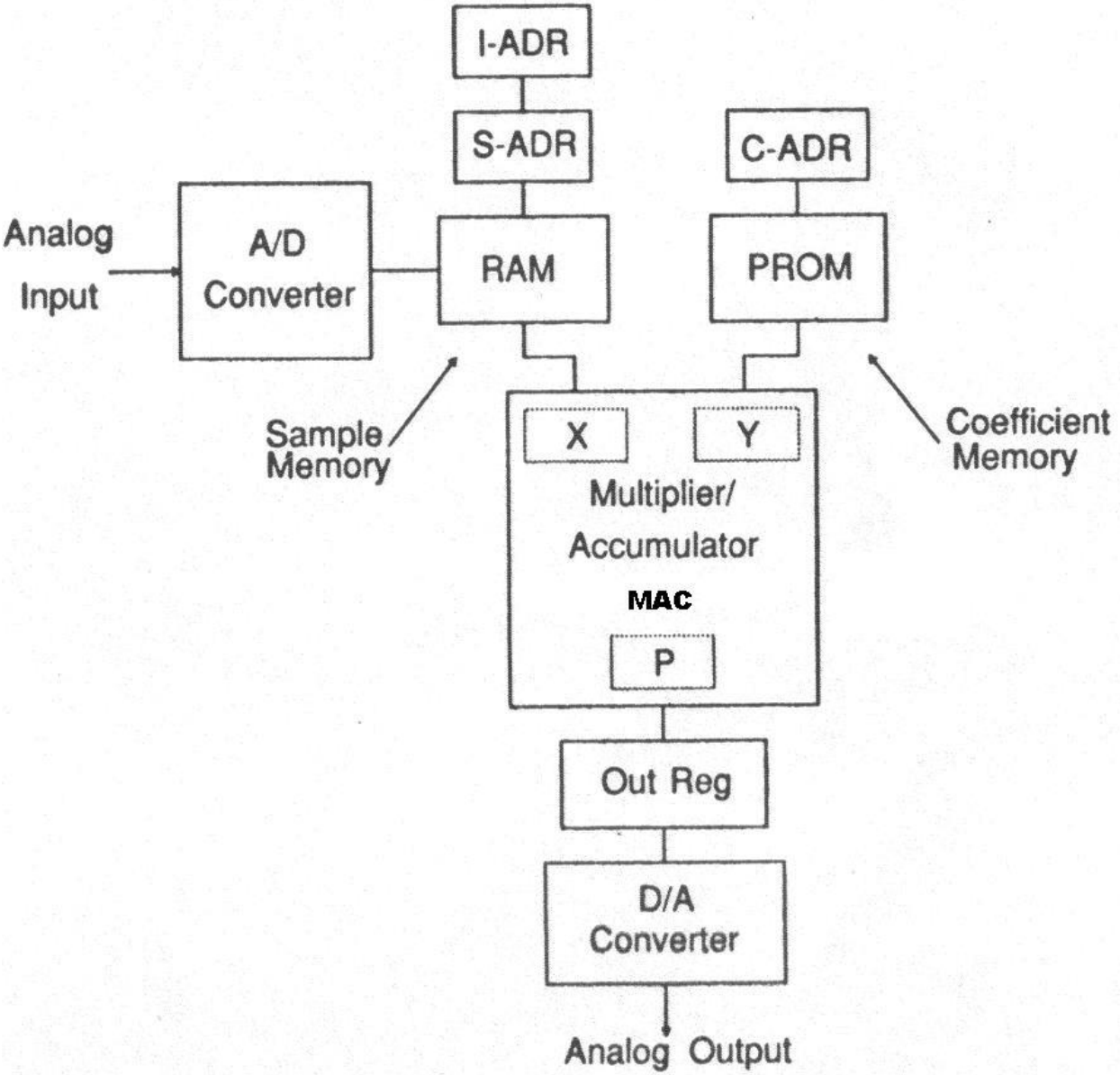
(1 clk)

$$output = \sum_{i=0}^{24} S_i \times C_i$$

Példa: FIR szűrő felépítése

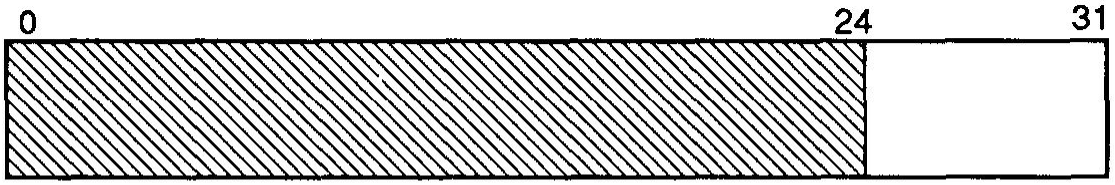
- **IMAC: (Multiplier/Accelerator):** a szorzást, és az összeadást egy órajel ciklusban képes elvégezni. Három regiszterből áll: két bemeneti (X és Y) és egy kimeneti regiszterből (P).
- **Együttható- memória (Coefficient memory):** C_MEM tárolja a Ci konstansok értékeit. Ez egy kisméretű PROM memória.
- **Együttható- memóriacímző regiszter: (C_ADDR)** egy számláló, amely a következő együtthatót azonosítja. 0-ról indul minden egyes iterációkor és az együtthatók címei egyesével inkrementálódnak.
- **Minta-memória (S_MEM):** az éppen aktuális és az azt megelőző 24 mintát tároljuk el. RAM, amely 32 értékből csak 25-t használ.
- **Minta- memóriacímző regiszter: (S_ADDR)** Ez egy számláló, amely az aktuális minta címét inicializálja, ezután pedig a (sorrendben) előző minta címére inkrementálódik (mutat)
- **Kezdő minta címregiszter (I_ADDR):** azonosítja az algoritmus kezdőpontját mindenegyes átmenetnél (akt. iteráció kezdőcíme mindig 1-el kevesebb lesz)
- **A/D konverter: (ADIN)** Minden iterációban ezen az egységen keresztül kapjuk az új adatot.
- **D/A konverter: (DAOUT)** A kimeneti regiszterből kapja az adatot, és analóg jellé alakítja, a további feldolgozás számára elfogadható formában.
- **Kimeneti regiszter: (OUT)** MAC-ből jövő értéket tárolja és a D/A konverternek továbbítja.

Példa: FIR szűrő blokkszintű felépítése



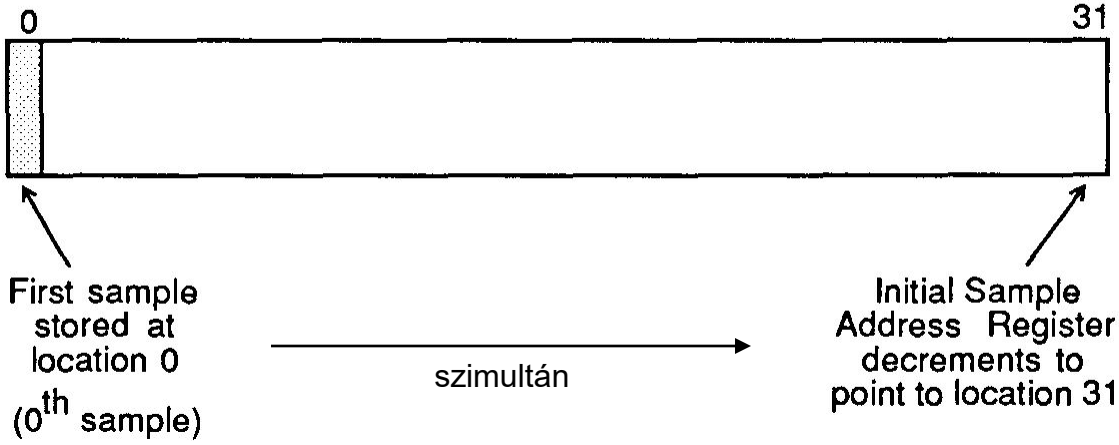
Koefficiens- és minta-adat tárolása

- Mindkettő C_MEM, S_MEM is 32 elemű tárolók



Coefficients stored in first 25 locations of PROM (0…24)

- Első betöltött minta után:

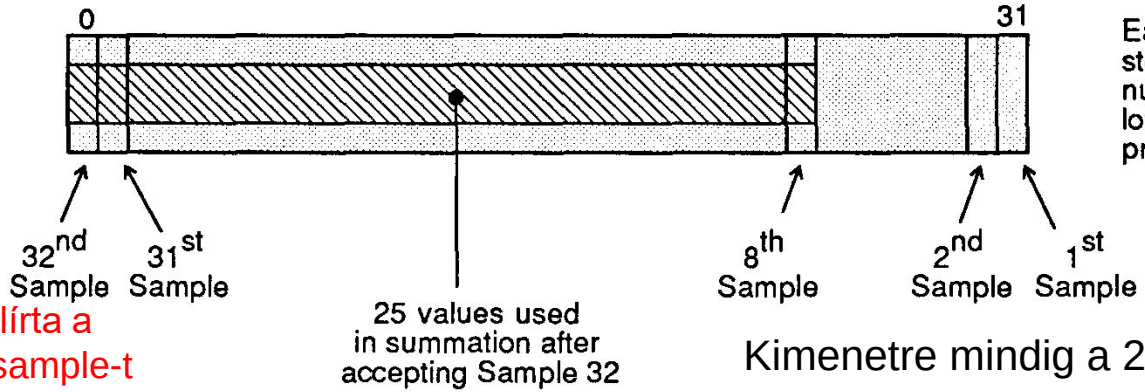


Samples start at location 0 of RAM (0-…)

I_ADR-- dekrementálása (31-…)

Koefficiens és adat tárolás (folyt)

- 33 betöltött minta után:

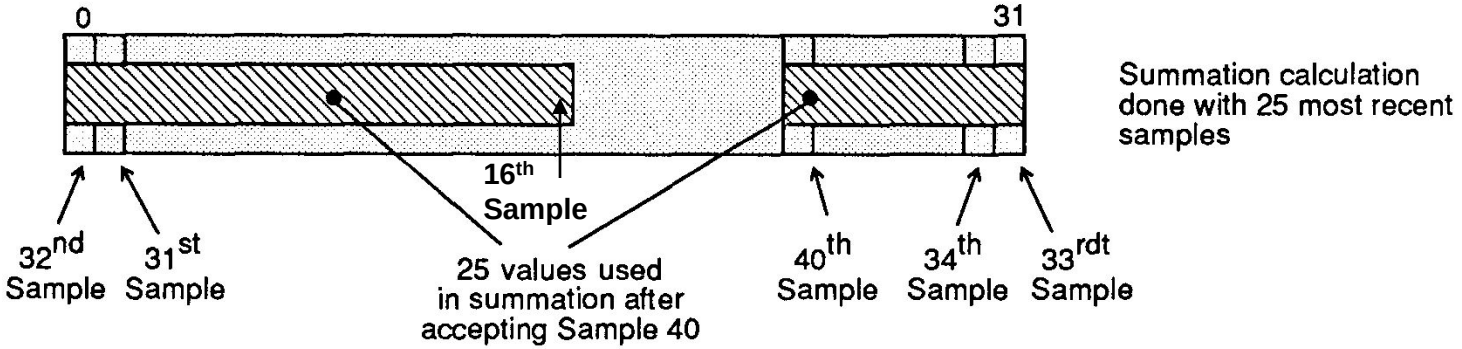


(31,30,…,0)

Felülírta a 0th sample-t

Kimenetre mindig a 25 **legutolsó** érték kerül!

- 40 betöltött minta után:



FIR - RTL leírása

```
start: if (ADIN not ready)   goto start          Check input data.
                                                 Várakozik..., új adat érkezéséig

       0          →  C_ADR                       Clear coefficient address.
       I_ADR      →  S_ADR                       Load sample address.

       ADIN       →  S_MEM[S_ADR]                Data to sample memory.
       I_ADR − 1  →  I_ADR                       Decrement initial address.
                                                 Szimultán betöltés és dekrementálás

       S_MEM[S_ADR]  →  X                        Load sample to X reg.
       C_MEM[C_ADR]  →  Y                        Load coefficient to Y reg.
       S_ADR + 1     →  S_ADR                    Increment sample address.
       C_ADR + 1     →  C_ADR                    Increment coefficient address.
                                                 Első minta (X) és első koefficiens (Y) betöltése a MAC bemenetére

       X × Y         →  P                        Load product register.
       S_MEM[S_ADR]  →  X                        Load sample to X reg.
       C_MEM[C_ADR]  →  Y                        Load coefficient to Y reg.
       S_ADR + 1     →  S_ADR                    Increment sample address.
       C_ADR + 1     →  C_ADR                    Increment coefficient address.
                                                 Szorzat (P) számításával egyidejűleg új mintát (X) és új koefficienst (Y) töltünk be

over:  P + X × Y     →  P                        Add product into P register.
       S_MEM[S_ADR]  →  X                        Load sample to X reg.
       C_MEM[C_ADR]  →  Y                        Load coefficient to Y reg.
       S_ADR + 1     →  S_ADR                    Increment sample address.
       C_ADR + 1     →  C_ADR                    Increment coefficient address.

       if (not done)   goto over                 Repeat for all samples.
                                                 Work is done when C_ADDR is 25.
                                                 Iterációs ciklus, amíg „not Done”, tehát nem érjük el az utolsó mintát!

       P  →  OUT                                 Update output value.

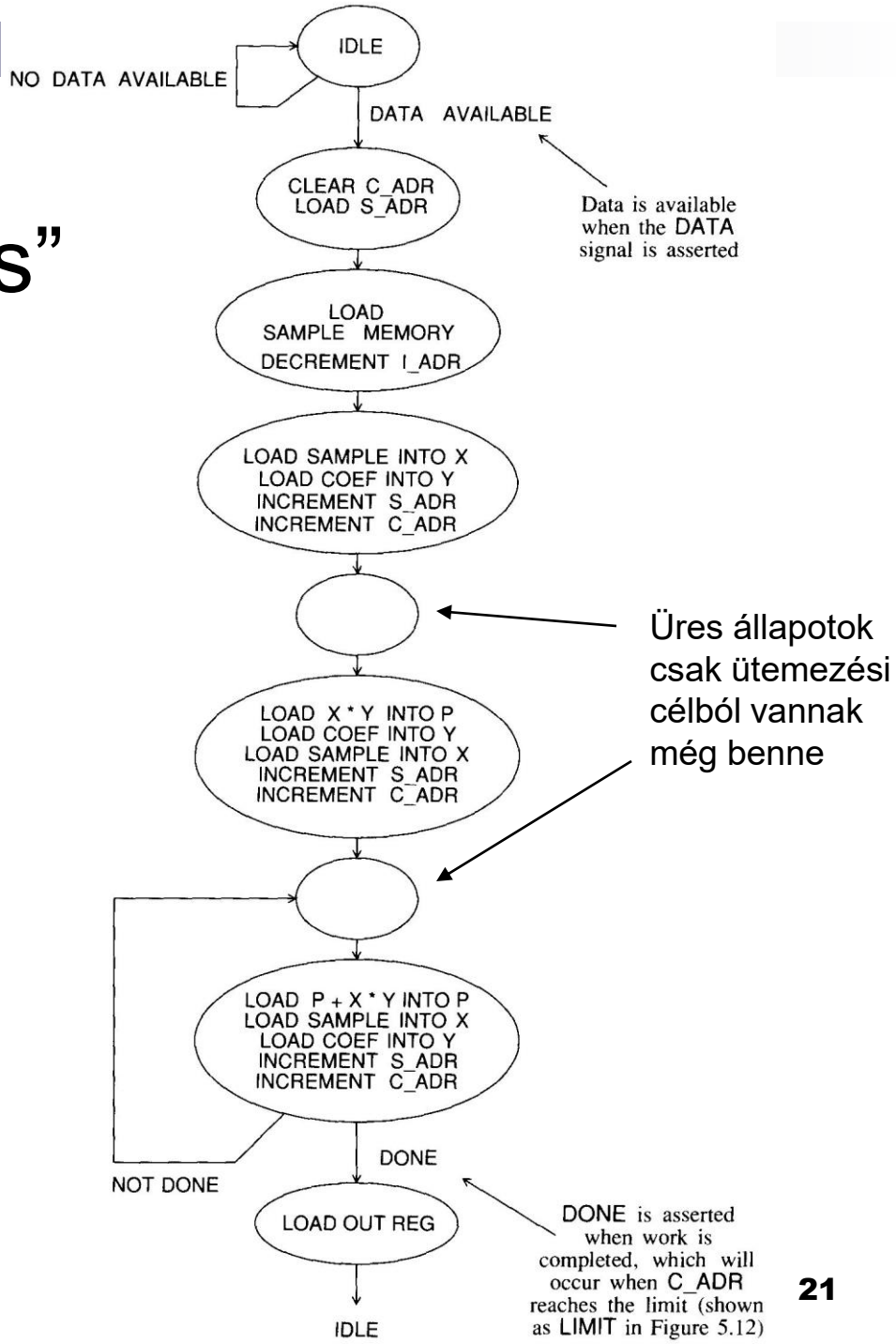
              goto start                         Start over.
```

RTL leírásnak megfelelő „előzetes” állapotdiagram

Vezérlési szekció: adatátvitel nem azonnal, hanem véges idő alatt megy végbe!

Ezt a tervezőnek kezelnie kell tudni (előzetes diagramm).

RTL leírásból képzett diagramnak lehetnek még *üres* állapotai is, azonban később az optimalizálás, véglegesítés során ezek kiesnek – egyszerűsödnek!



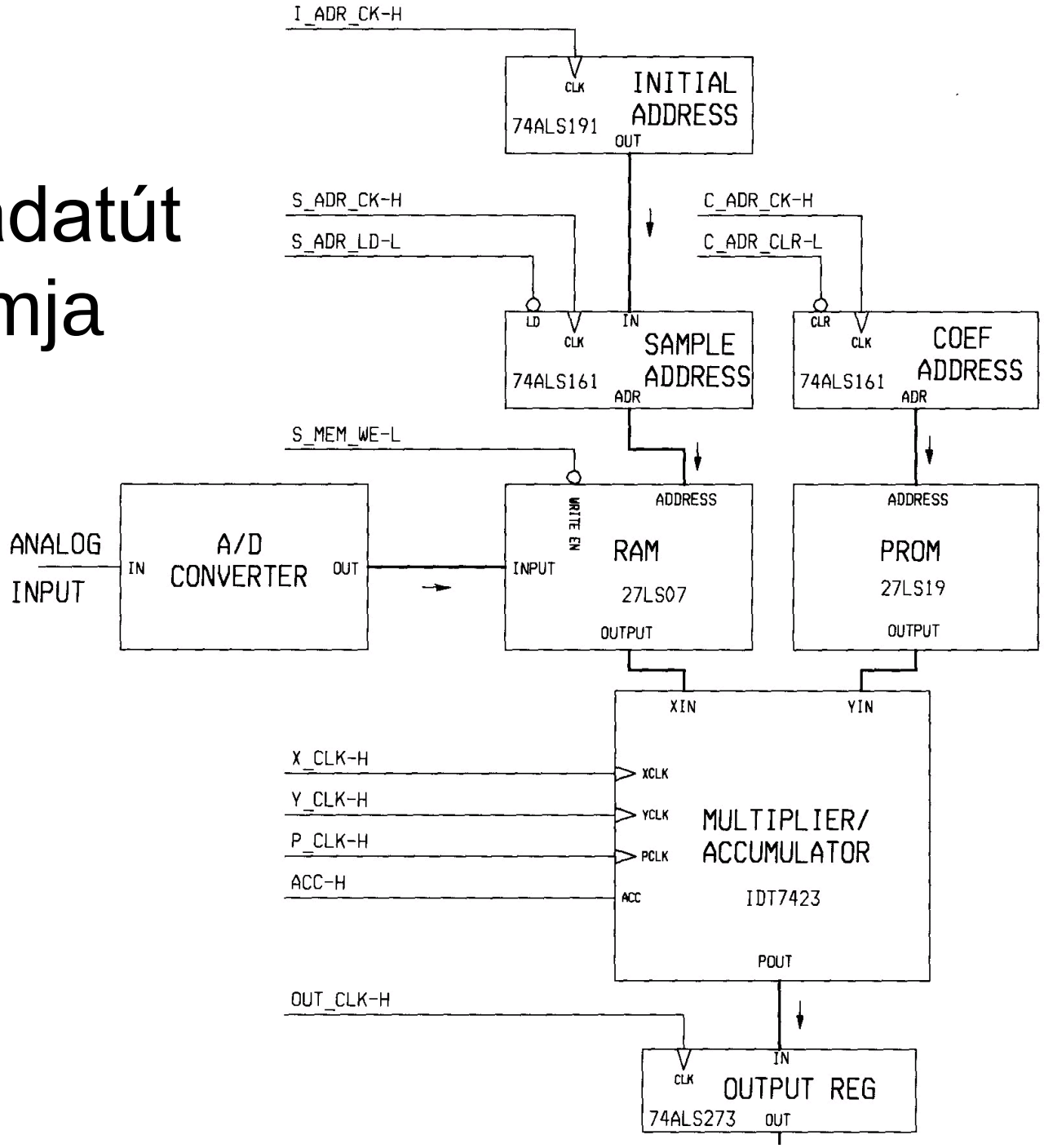
FIR szűrő „részletes” adatút blokkdiagramja

Végleges adatút diagramm:

RTL leírás + előzetes diagramm alapján kapjuk!

Itt már a **vezérlő jelek** is fel vannak tüntetve!

További tervezői feladatunk lesz ezeknek a vezérlő-jeleknek a megfelelő ütemben való generálása!



Vezérlő egység tervezése egyszerű állapotgép (FSM) segítségével – FIR szűrőhöz

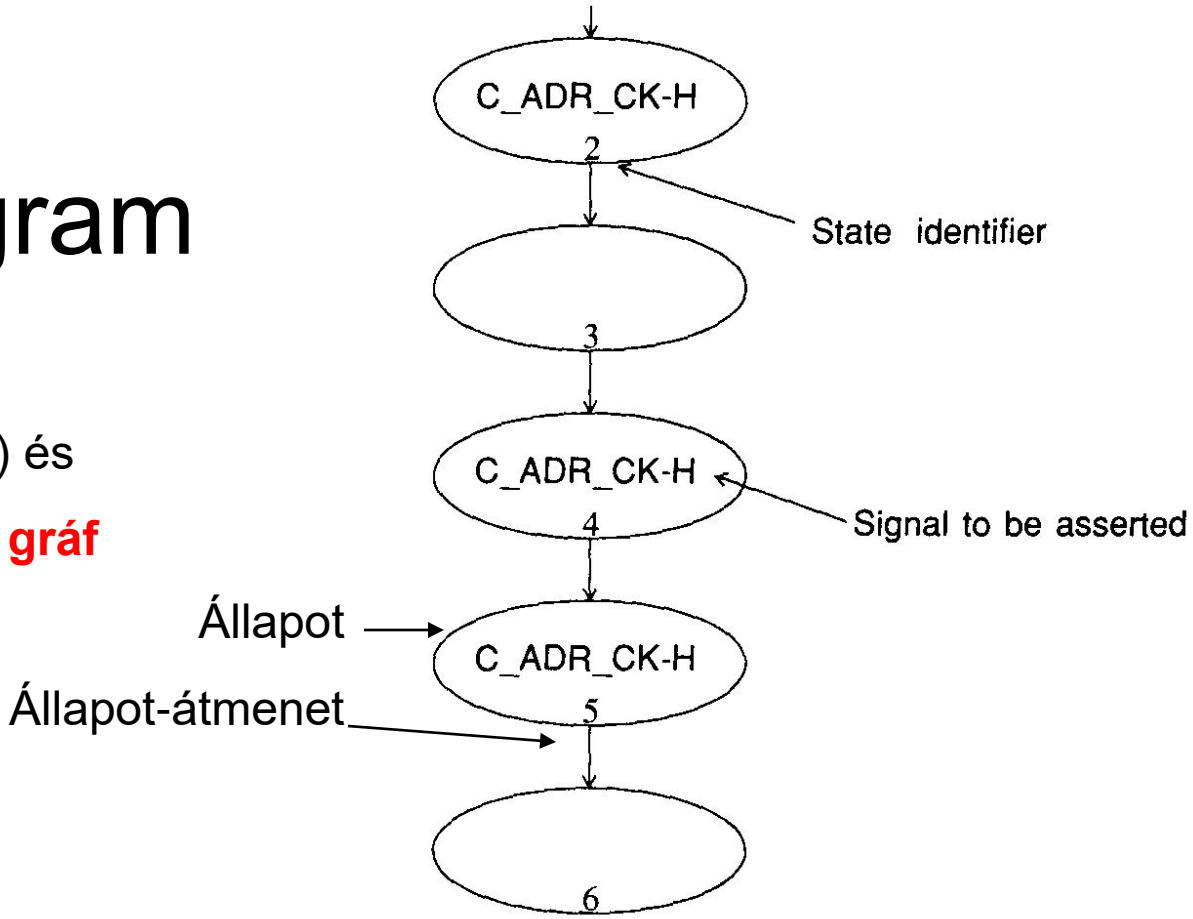
Tervezői feladatok (FIR szűrő):

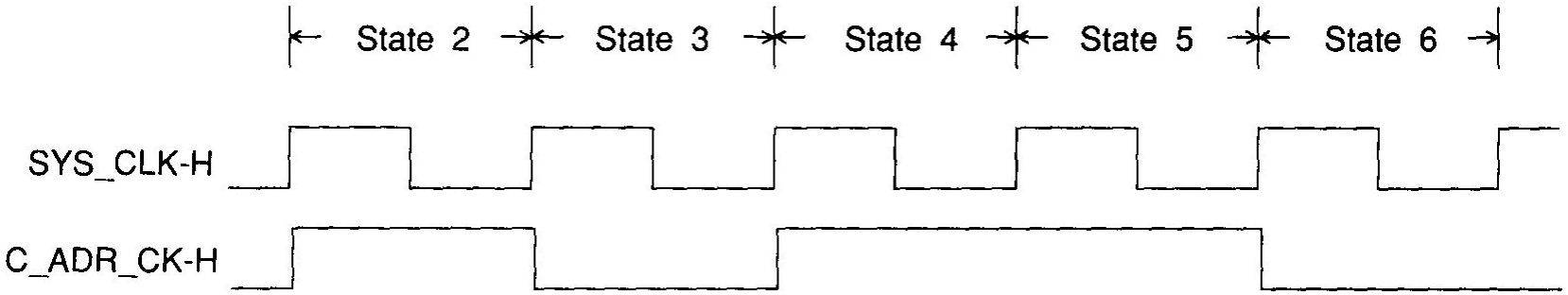
- Eddig rendelkezésre áll:
  - RTL leírás,
  - Előzetes és
  - részletes blokk diagram
- Most: a vezérlési rész tervezése következhet
  - 1.) Vezérlő jelek beállítása (assertion levels)
  - 2.) „Mapping” (hozzárendelés): a Moore modellt vesszük alapul, amelyhez a Véges Állapotú Automatát (FSM) próbáljuk illeszteni / hozzárendelni.

Példa: Állapot diagram

Gráf reprezentáció:

- (DFG: Adat-folyam gráf) és
- **CFG: Vezérlés-folyam gráf**



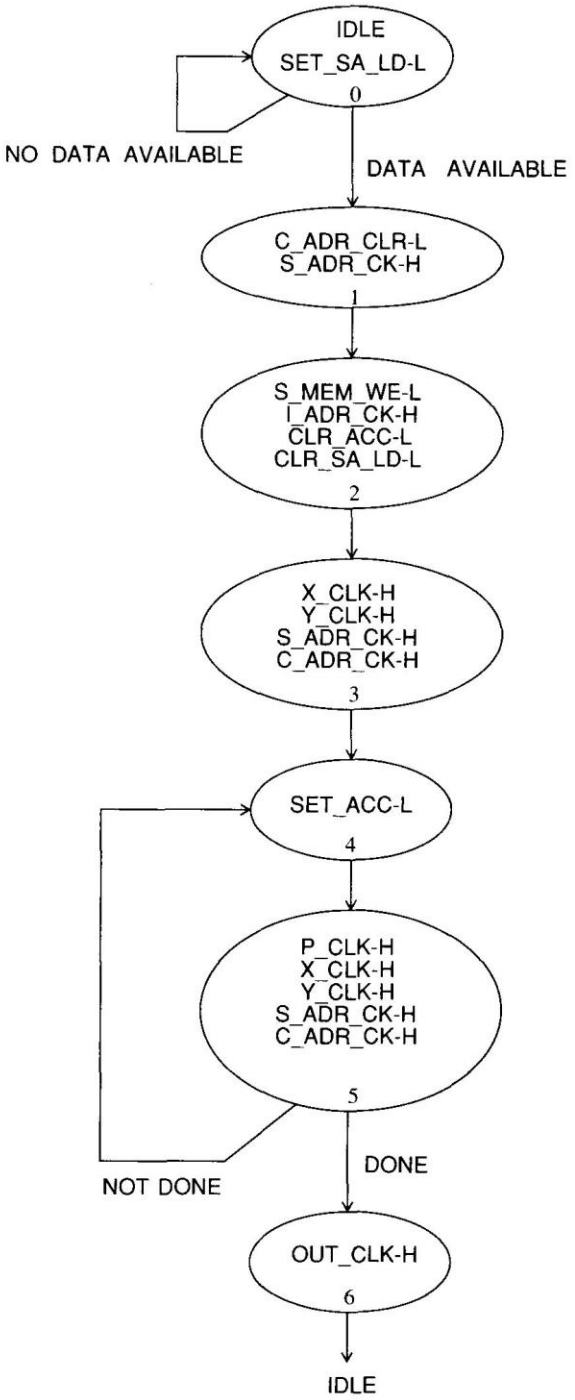


1.) FIR szűrő részletes FSM állapot diagrammja

SET-RESET FF-kal:

- Set S_ADR_LD (State0)
- Reset / Clear S_ADR_LD (State2)
- Reset ACC_L (State2)
- Set ACC_L (State4)

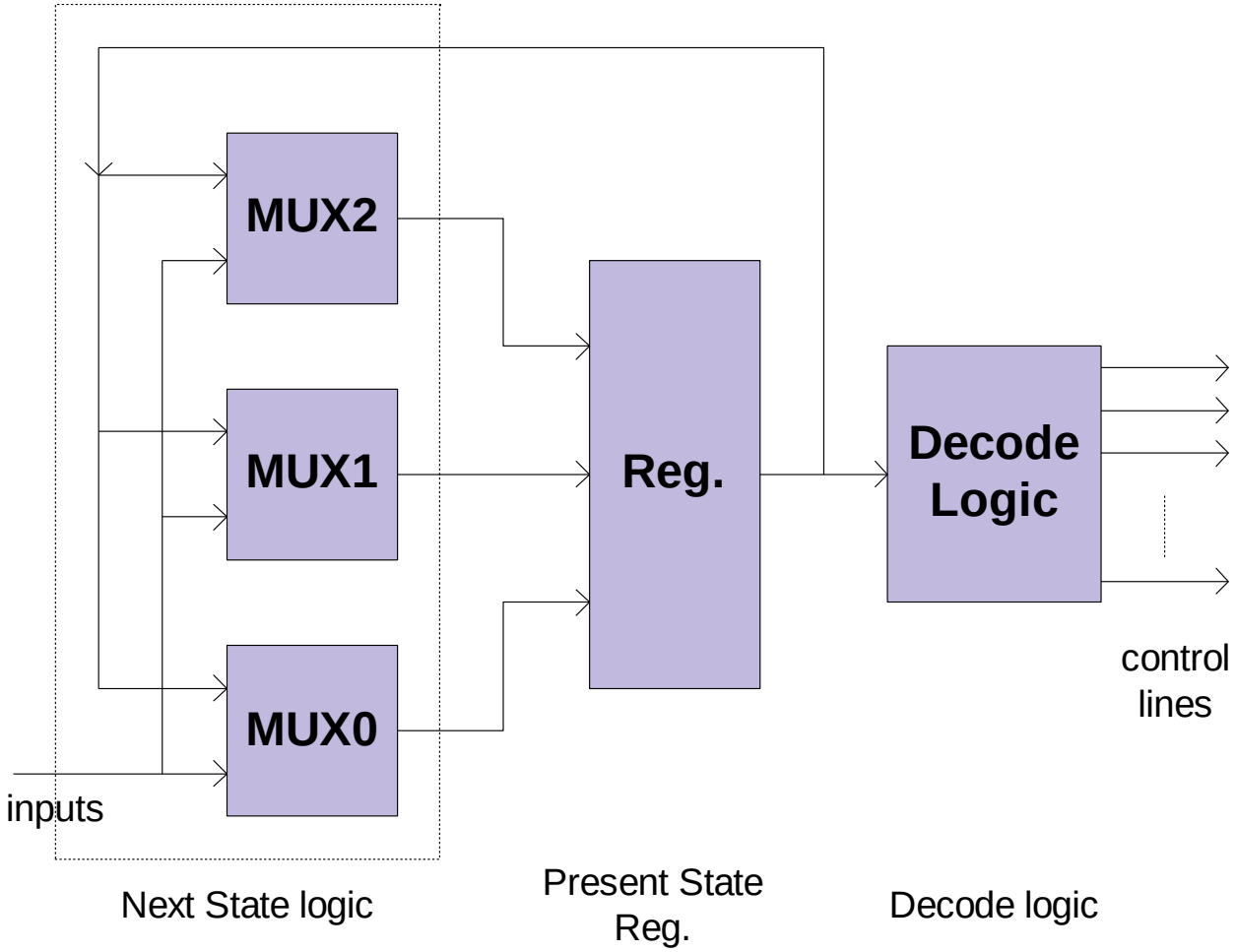
- Vezérlés folyam: $State_0$, … ,$State_6$



2.) „Mapping”: FIR szűrő vezérlőjének multiplexeres megvalósítása

- **Present State Reg**: aktuális állapotot tárolja
- **Next State Logic**: következő állapot kiválasztása a bemenetek, és a visszacsatolt állapotoktól függően (multiplexerek!)
- **Output Logic**: az aktuális állapot dekódolt formája kerül a kimenetekre (vezérlőjel generálás)

FIR szűrő vezérlőjének multiplexeres megvalósítása



FIR szűrő vezérlőjének multiplexeres megvalósítása

- **További két input signal**
  - DATA-H: A/D felől új feldolgozandó adat érkezett (flag)
  - DONE-H: szükséges számú iteráció végrehajtódott

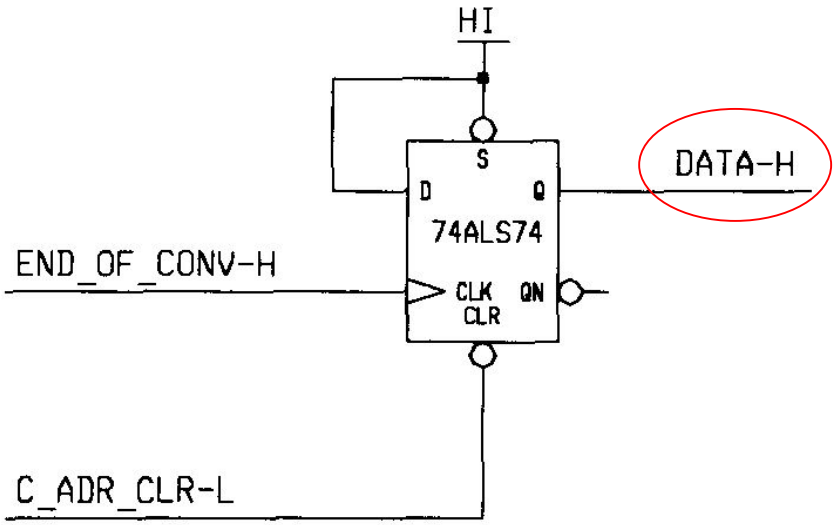
Table 5.1. Next State Multiplexer Specifications.

	MUX 2	*MUX 1*	*MUX 0*
State 0	0	0	DATA-H
State 1	0	1	0
State 2	0	1	1
State 3	1	0	0
State 4	1	0	1
State 5	1	DONE-H	0
State 6	0	0	0

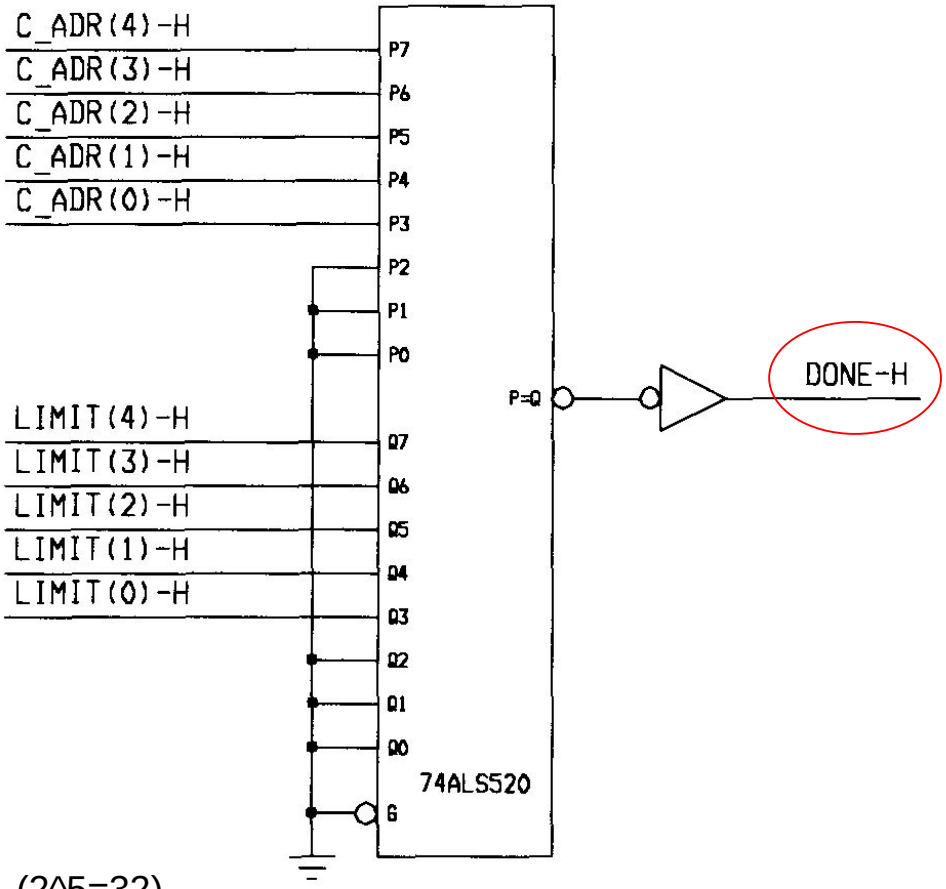
FIR szűrő részletes FSM állapot-diagram alapján kapjuk a táblázatot!

FSM vezérlőjelei

DATA-H



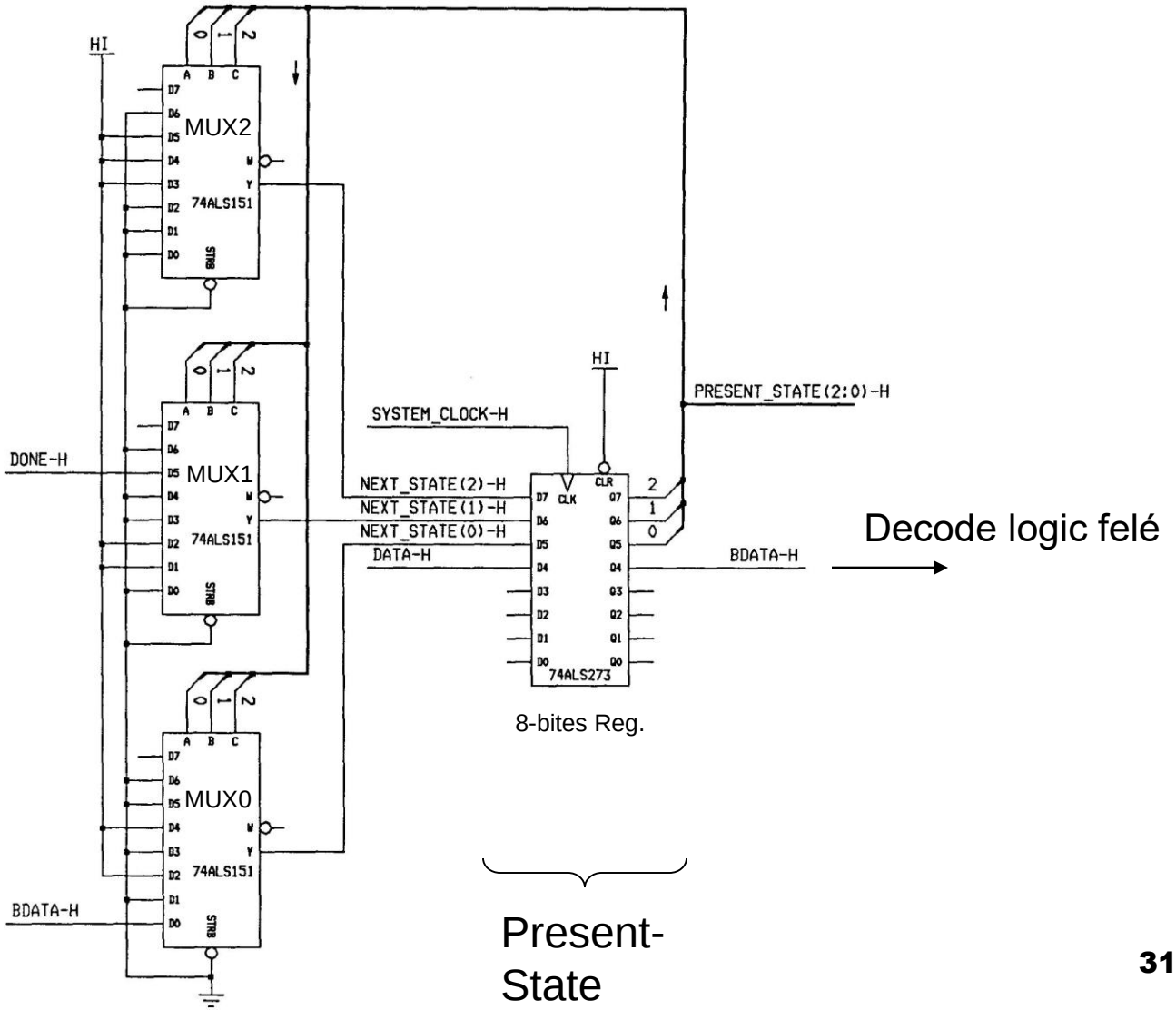
DONE-H



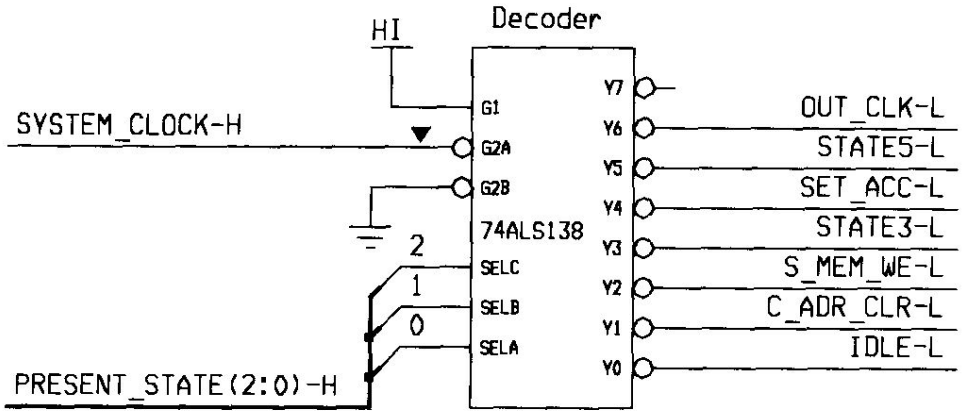
(2^5=32)

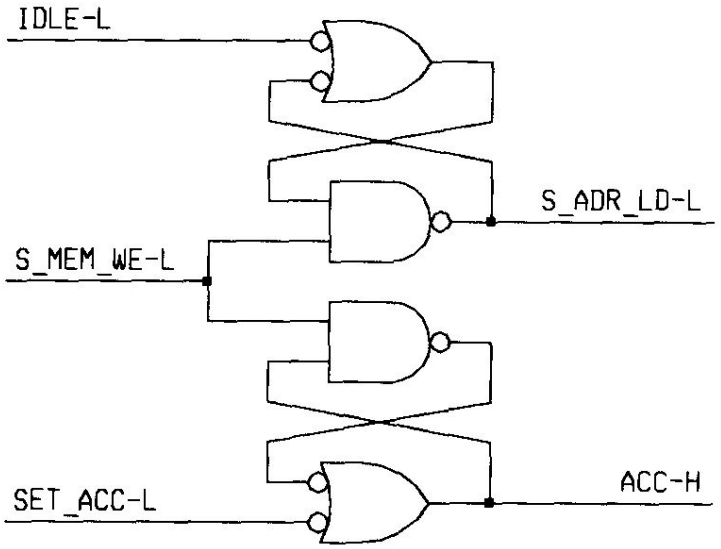
komparátor

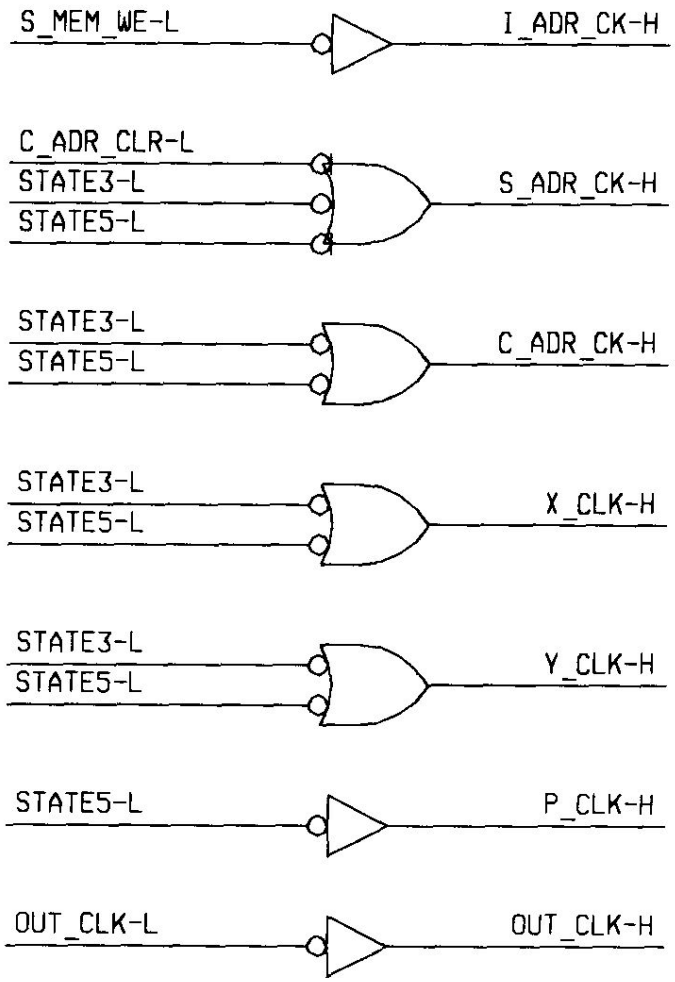
FSM vezérlő egységének Next-State és Present-State logikája (blokk diagramm)



Present-State logikai áramkörei





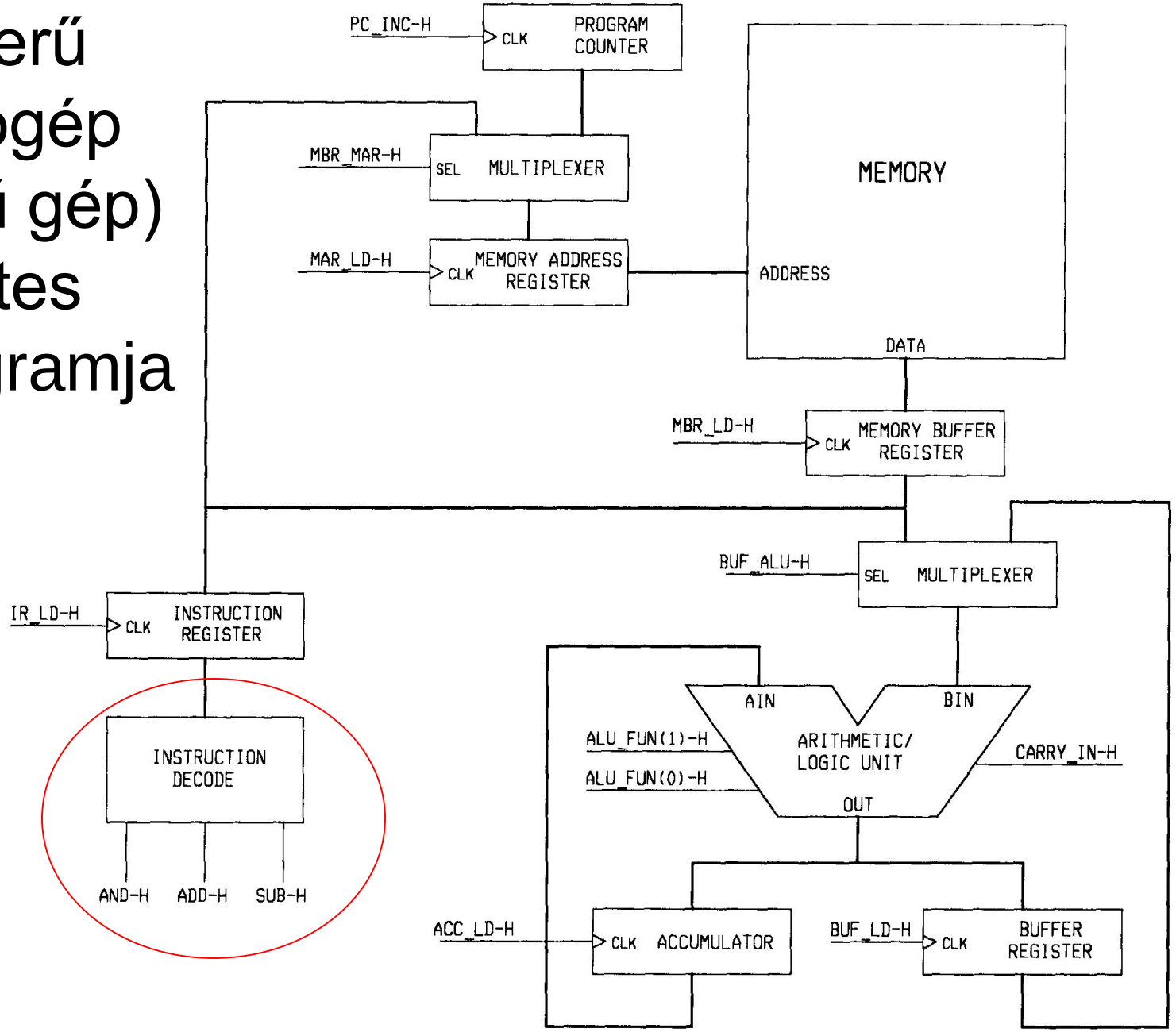


Szekvenciális vezérlő rendszerek – egyedi késleltetéses módszer

Szekvenciális vezérlő rendszerek

- Ahelyett – mint azt az állapotgépnél (FSM) láthattuk – hogy egyetlen regiszter tömbben tárolnánk a rendszer állapotait, ebben az esetben *külön regisztereket* definiálunk egy egyszerű számítógép (egycímű gép) blokkdiagramját felhasználva.
- Regiszter-transzfer műveletek sora mutatja be ennek a *késleltetéses* vezérlő rendszernek a működését.

Egyszerű számítógép (egycímű gép) részletes blokkdiagramja



Példa: 3 egyszerű utasítás

- ADD
- SUB
- AND

Példa: Egycímű gép vezérlési funkciója

- Mivel példaként három egyszerű két-operandusú utasítást (AND, 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}= 40ns

T_{MEM}= 200ns

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]

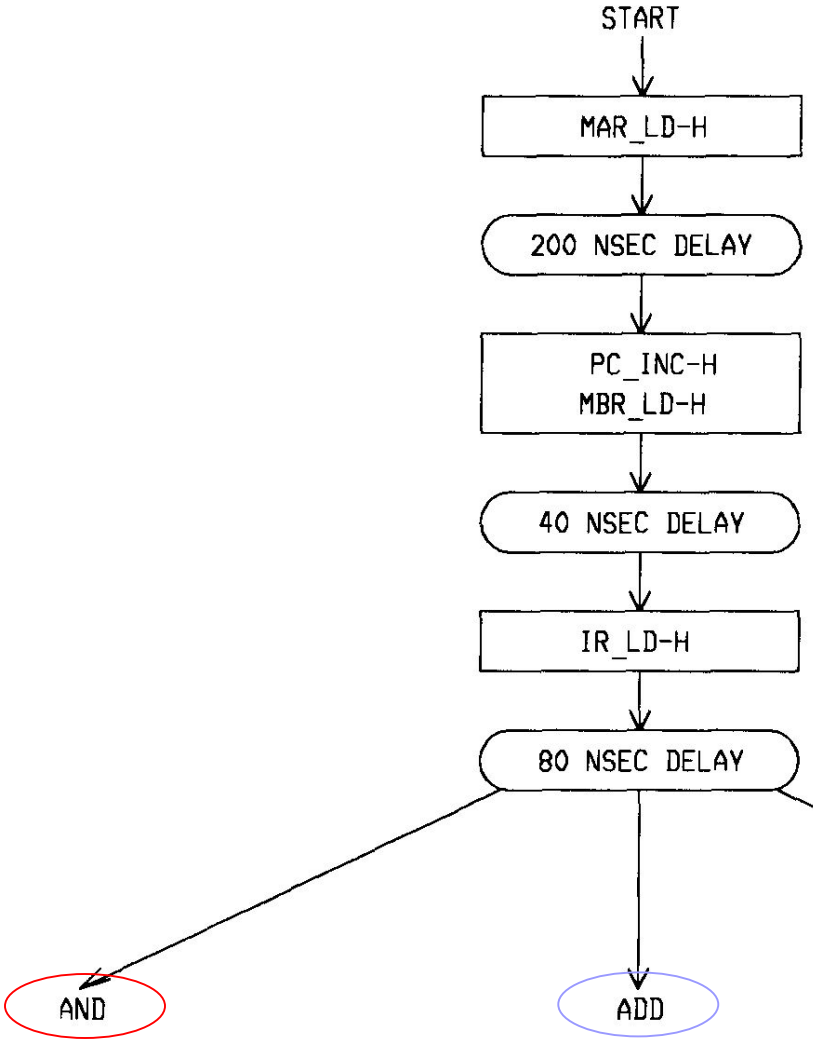
Három utasítás RTL leírása

Table 5.2. Register Transfers for Three Instructions.

Register Transfers for Example

AND *Instruction*	ADD *Instruction*	SUBTRACT *Instruction*
PC → MAR	PC → MAR	PC → MAR
M [MAR] → MBR	M [MAR] → MBR	M [MAR] → MBR
PC + 1 → PC MBR → IR	PC + 1 → PC MBR → IR	PC + 1 → PC MBR → IR
MBR → MAR	MBR → MAR	MBR → MAR
M [MAR] → MBR	M [MAR] → MBR	M [MAR] → MBR
MBR • ACC → ACC	MBR + ACC → ACC	¬ MBR → BUF (2's Comp)
		BUF + ACC + 1 → ACC (2's Comp)

Három utasítás késleltetéses folyamat ábrája (delay flowchart)



Utasítás Fetch:

$$PC \rightarrow MAR$$

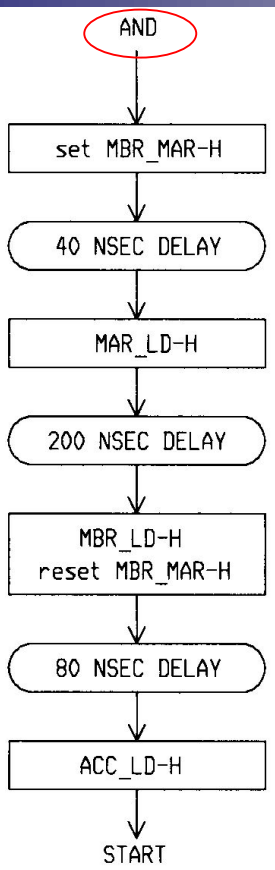
$$M[MAR] \rightarrow MBR$$

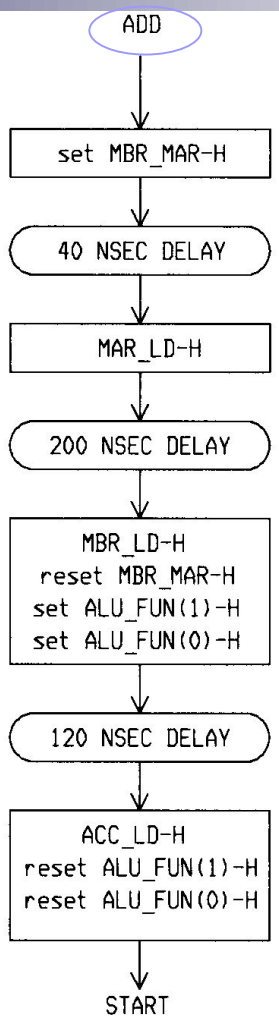
$$PC + 1 \rightarrow PC$$

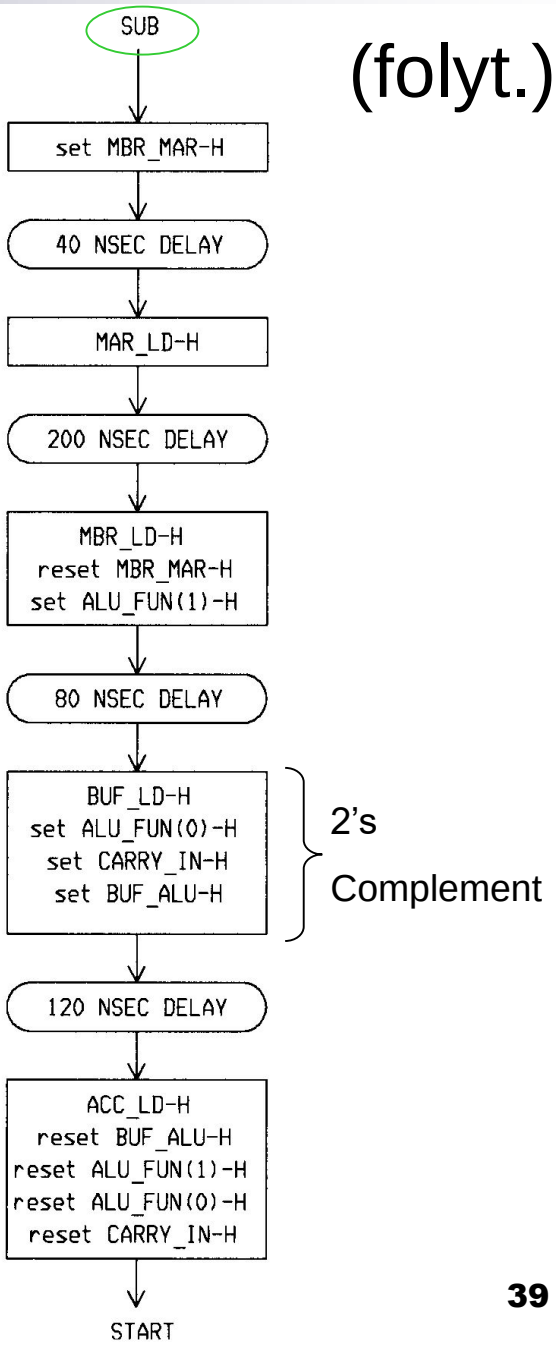
$$MBR \rightarrow IR$$

Mindhárom esetben ugyanaz.

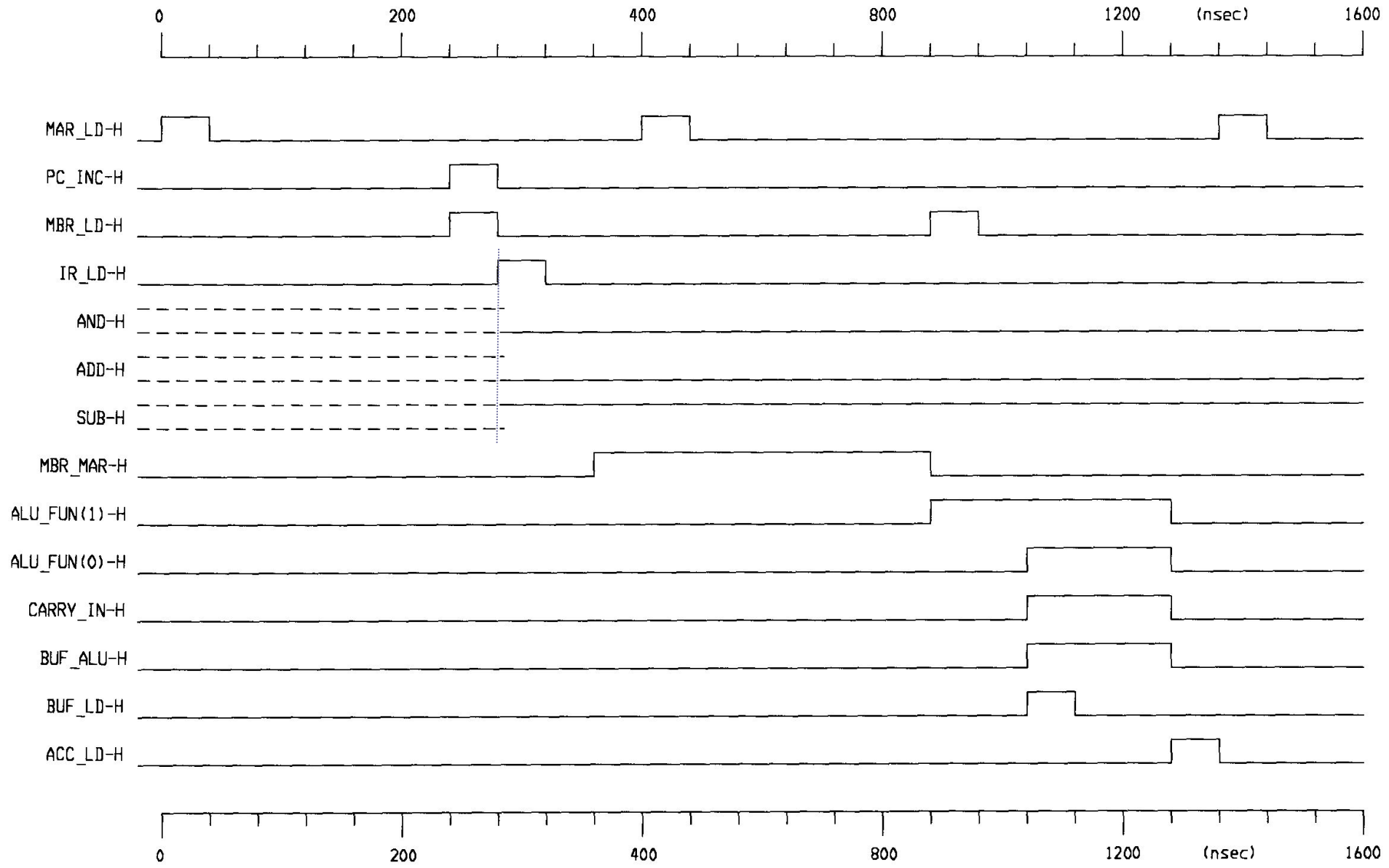
(folyt.)


AND
set MBR_MAR-H
40 NSEC DELAY
MAR_LD-H
200 NSEC DELAY
MBR_LD-H
reset MBR_MAR-H
80 NSEC DELAY
ACC_LD-H
START

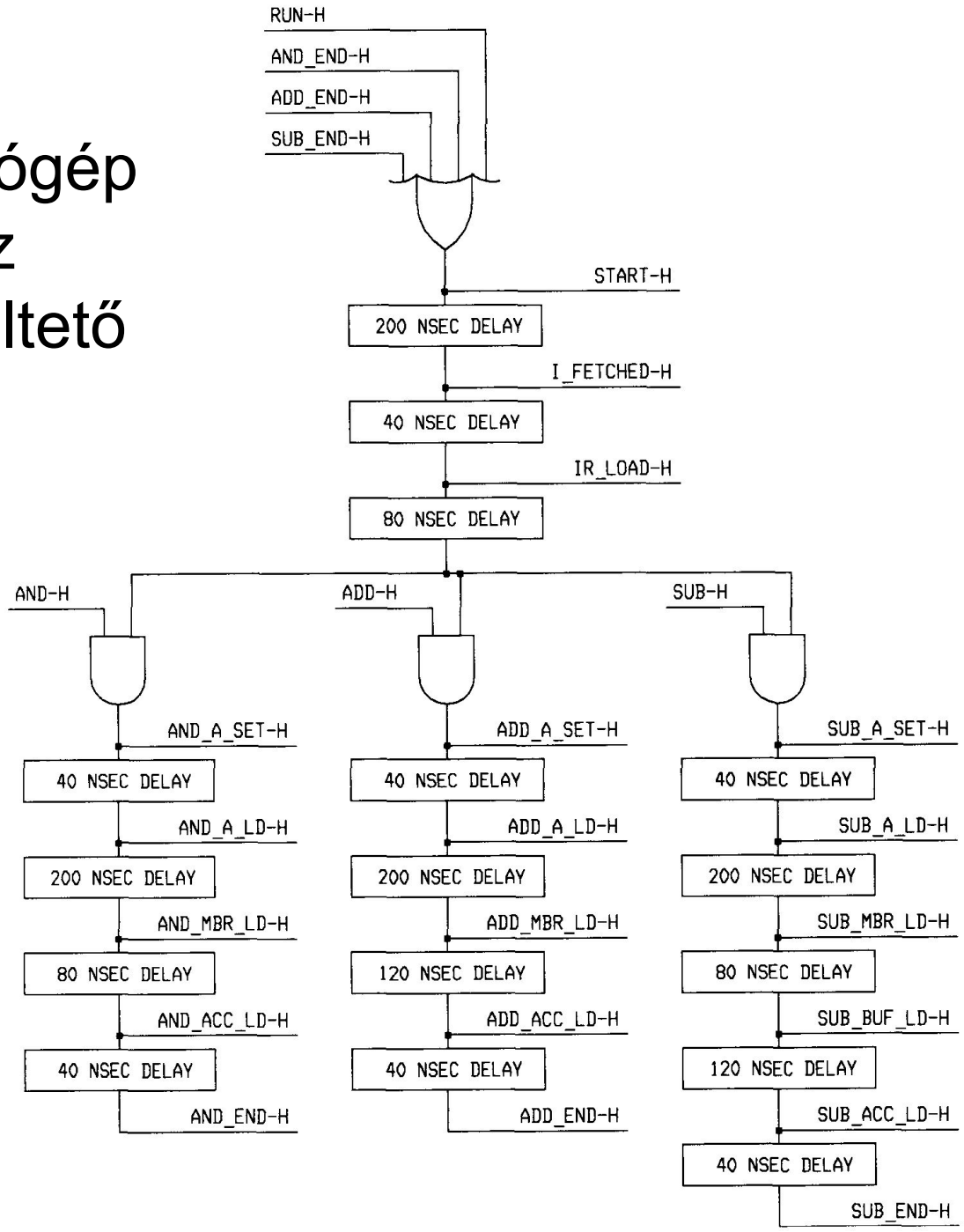

ADD
set MBR_MAR-H
40 NSEC DELAY
MAR_LD-H
200 NSEC DELAY
MBR_LD-H
reset MBR_MAR-H
set ALU_FUN(1)-H
set ALU_FUN(0)-H
120 NSEC DELAY
ACC_LD-H
reset ALU_FUN(1)-H
reset ALU_FUN(0)-H
START


SUB
set MBR_MAR-H
40 NSEC DELAY
MAR_LD-H
200 NSEC DELAY
MBR_LD-H
reset MBR_MAR-H
set ALU_FUN(1)-H
80 NSEC DELAY
BUF_LD-H
set ALU_FUN(0)-H
set CARRY_IN-H
set BUF_ALU-H
2's
Complement
120 NSEC DELAY
ACC_LD-H
reset BUF_ALU-H
reset ALU_FUN(1)-H
reset ALU_FUN(0)-H
reset CARRY_IN-H
START

PI: SUB művelethez tartozó időzítési diagram



Egyszerű számítógép vezérléséhez szükséges késleltető elemek

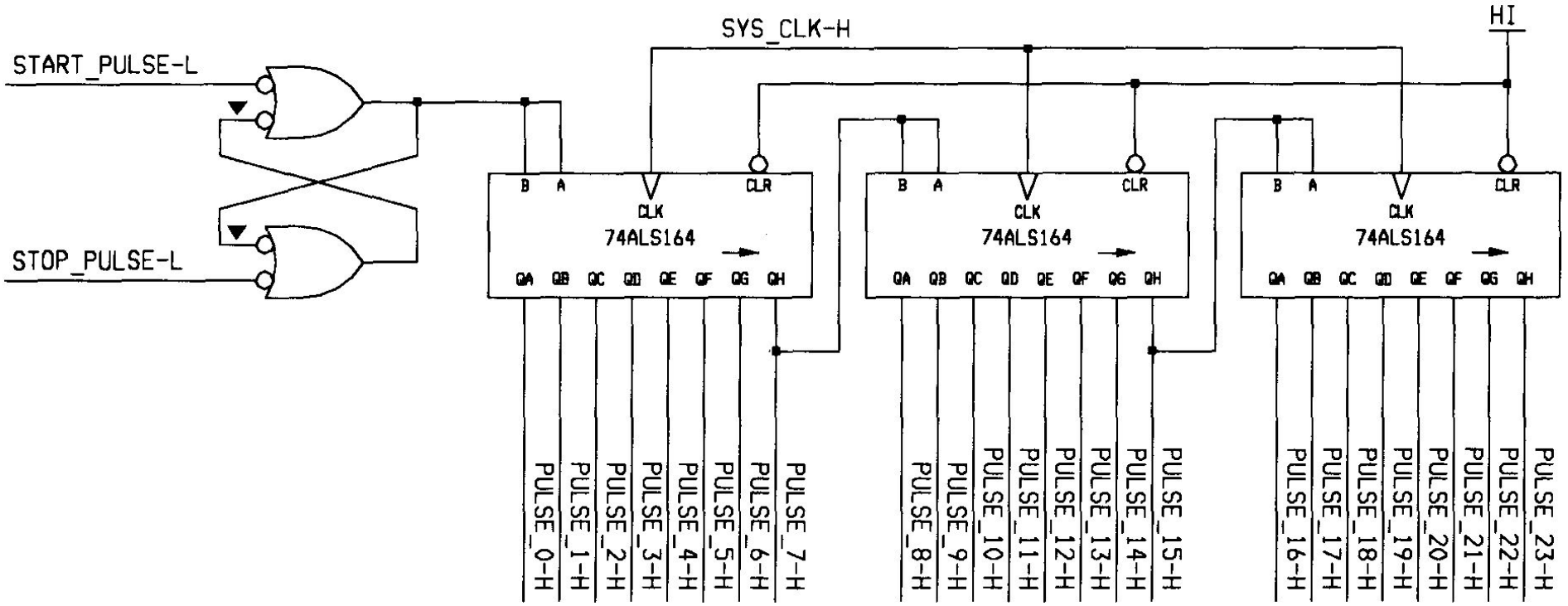


Szekvenciális vezérlő rendszerek tervezése Shift-regiszteres időzítővel

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.
- Három 8-bites Shift-regiszter segítségével generálja az impulzusokat (közvetve a vezérlő jeleket is).

Sorrendi vezérlő egységek megvalósítása Shift-regiszterekkel



Vezérlő impulzusok

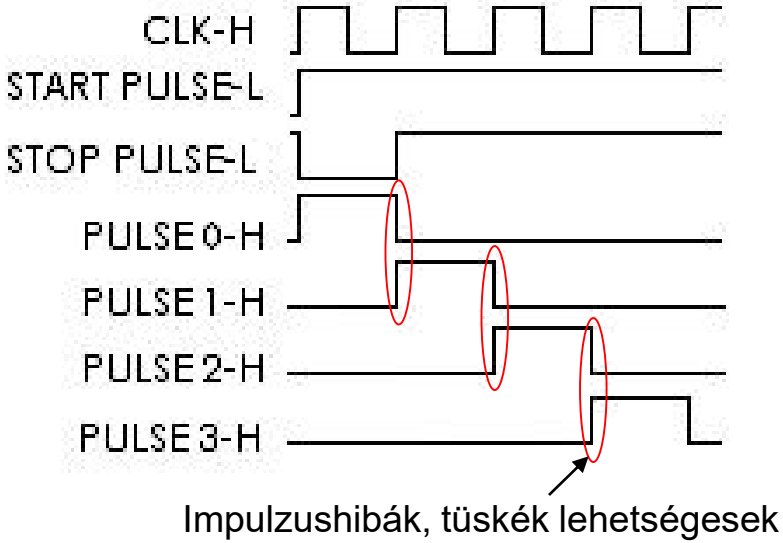
1. módszer: Nem-átlapoló impulzusok

- 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.

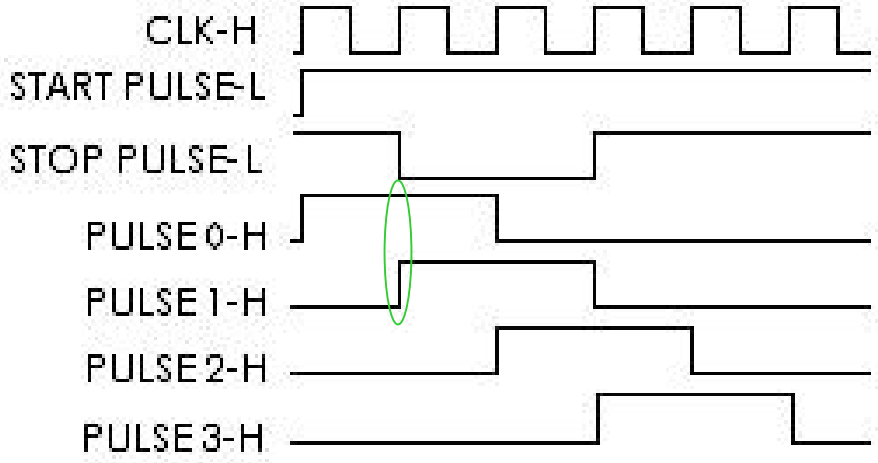
2. módszer: Átlapoló impulzusok

- 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 időd iagramon



Nemátlapoló rövid impulzusok (40ns)



Átlapoló hosszú impulzusok (80ns)

Mikrokódos vezérlők – reguláris vezérlési struktúrák

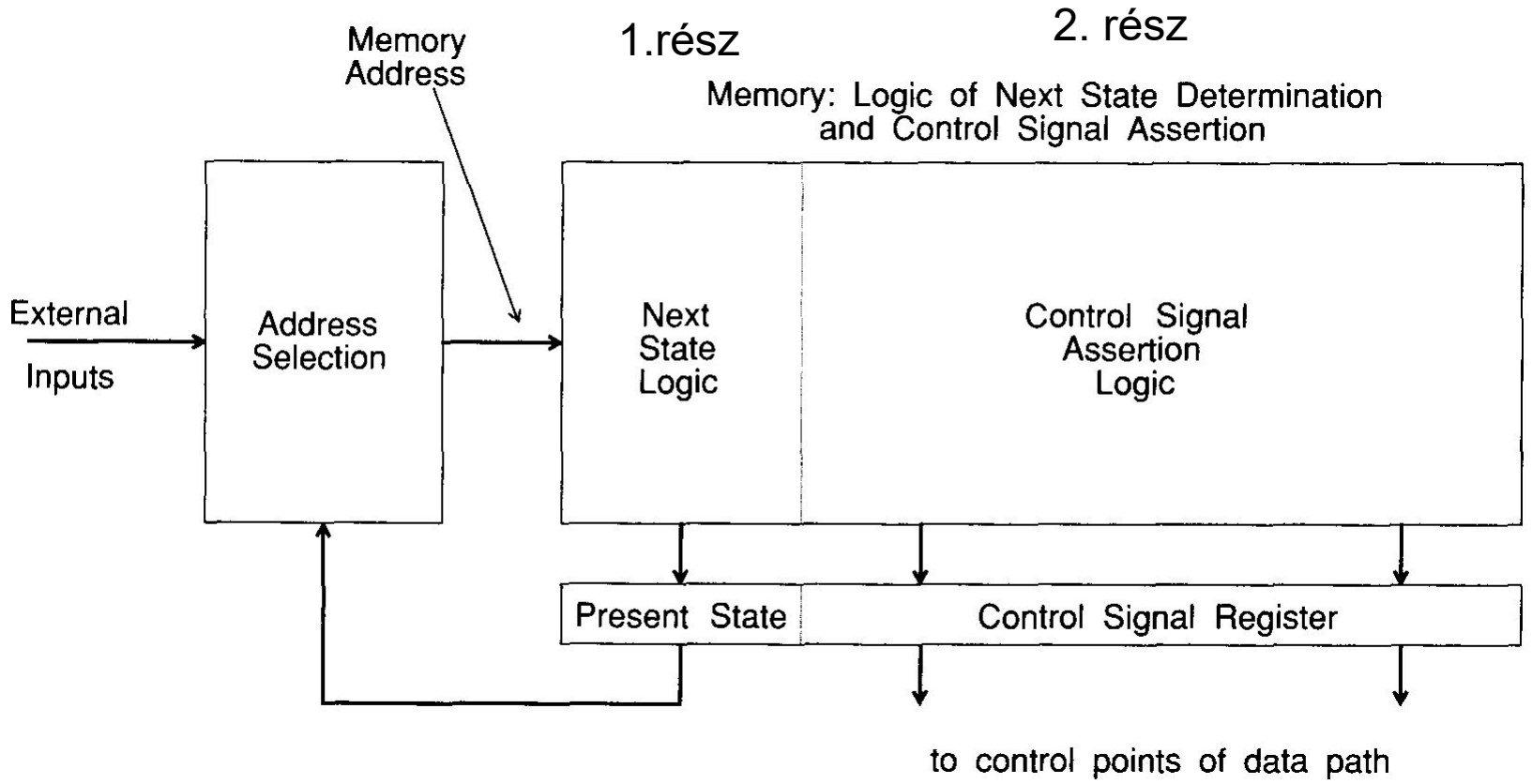
Ismétlés: Vezérlő egységek

- Általánosságban: a vezérlő egység 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 vagy reguláris módszer

- Eddig a „klasszikus”, késleltetéses módszereket tárgyaltuk (multiplexeres 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 - VHDL)
- Wilkes (1951): A komplex vezérlési folyamatokat **„reguláris módszerrel”** lehet egyszerűsíteni: nevezetesen **gyors memória elemeket** kell használni az utasítássorozatok tárolásánál. Állapotgépekkel (FSM) modellezik a vezérlő egység működését, és ezt a modellt transzformálják át mikrokódot használva. Az adatútvonal vezérlési pontjait memóriából (ROM) kiolvasott *vertikális-* vagy *horizontális-mikrokódú utasításokkal állítják be*!

FSM megvalósítása Memóriával

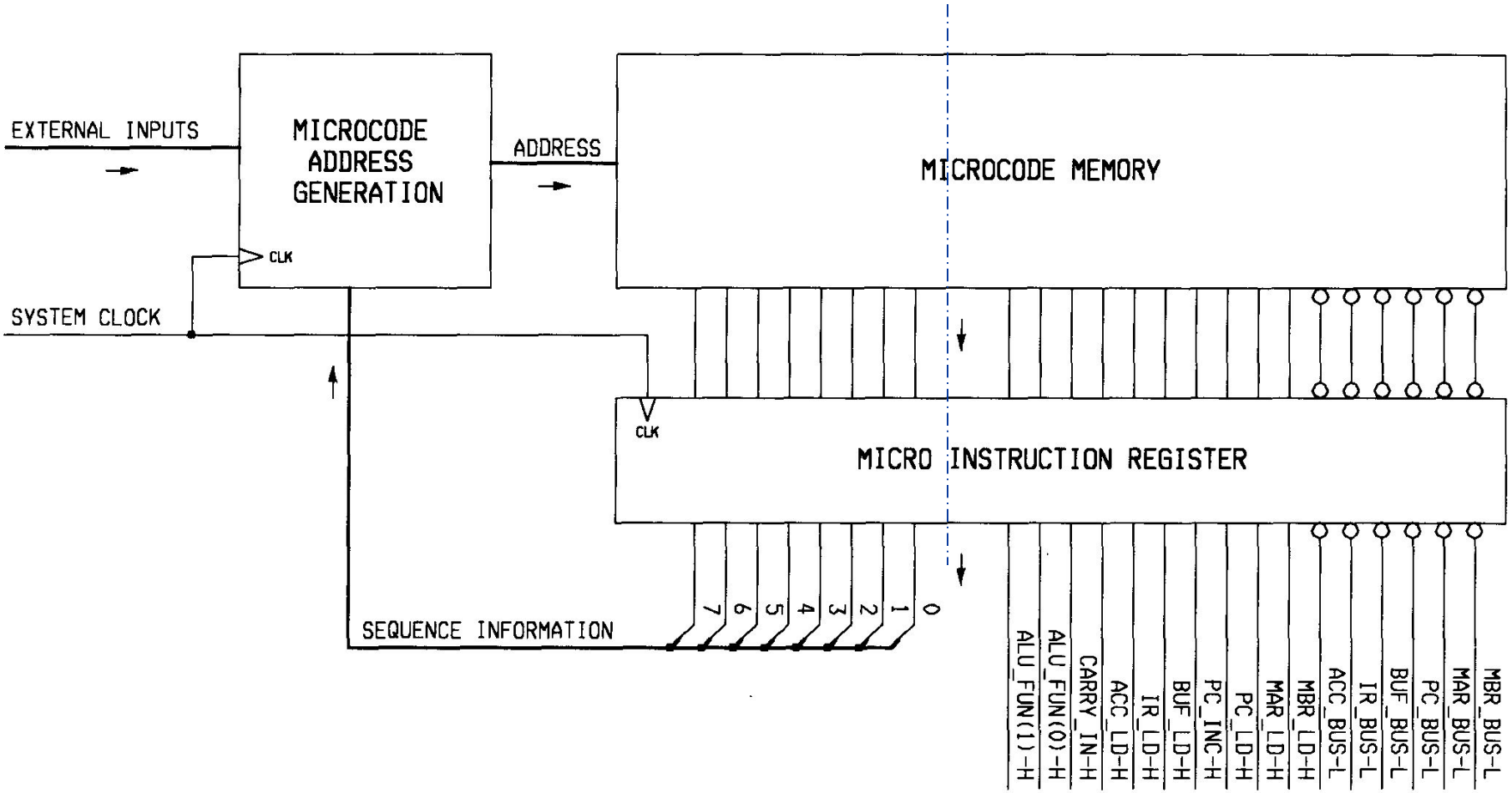


- 1.rész: szabályozza az eszköz működését a megfelelő állapotok *sorrendjében*
- 2.rész: szabályozza az adatfolyamot a megfelelő *vezérlőjelek beállításával* (assertion) az adatúton (vezérlési pontokon)

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.
- A memória címet (***memory address***-t) külső bemenő jelek és a present state határozzák meg együttesen. E cím segítségével megkapjuk az adott vezérlő információ pontos helyét a memóriában, ill. ez az információ, mint új állapot betöltődik a vezérlőjel regiszterbe (*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 lépéseiként 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.

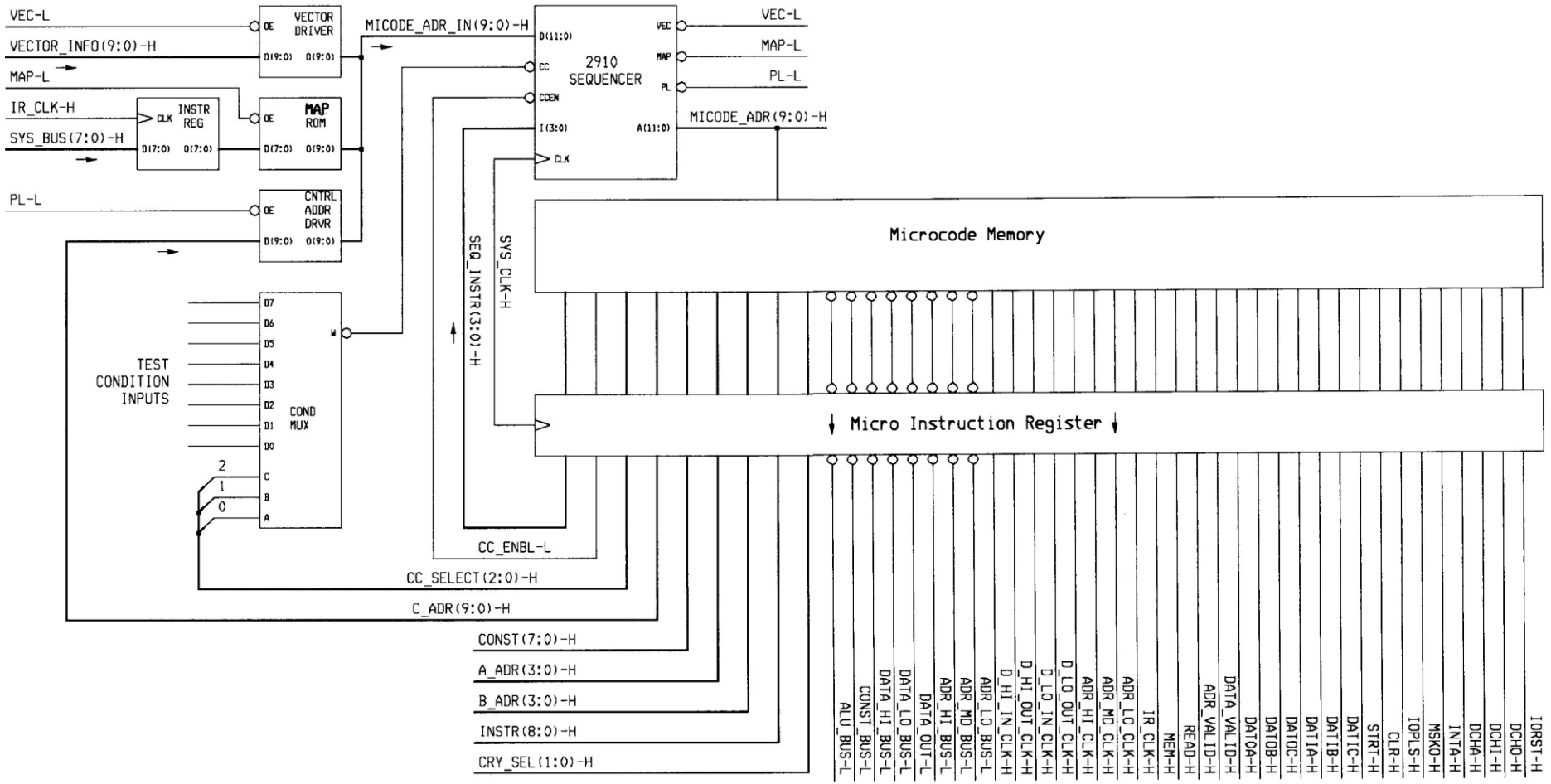
Általános Mikrokódos vezérlők tulajdonságai

- Egy gépi ciklus alatt egy **mikroprogram** fut le (amely mikroutasítások sorozatából áll). A műveleti kód a végrehajtandó mikroprogramot jelöli ki. A mikrokódú memória általában csak olvasható 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.
- Lassabb, mint a huzalozott vezérlő egységek, hiszen itt a memória elérési idejével is számolni kell (nem csak a visszacsatolt aktuális állapot késleltetésével.)

1.) Horizontális mikrokódos vezérlő

- Mindenegyes 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 regisztereket (memória, ACC) egyszerre, egyidőben tudjuk az órajellel aktiválni, ezáltal egy órajelciklus alatt az információ mindkét irányba átvihető. Növekszik a sebesség, mivel nincs szükség a vezérlőjelek dekódolását végző dekódoló logikára. Így minimálisra csökken a műveletek ciklusideje.
- Azonban nagyobb az **erőforrás** szükséglete, **fogyasztása**.

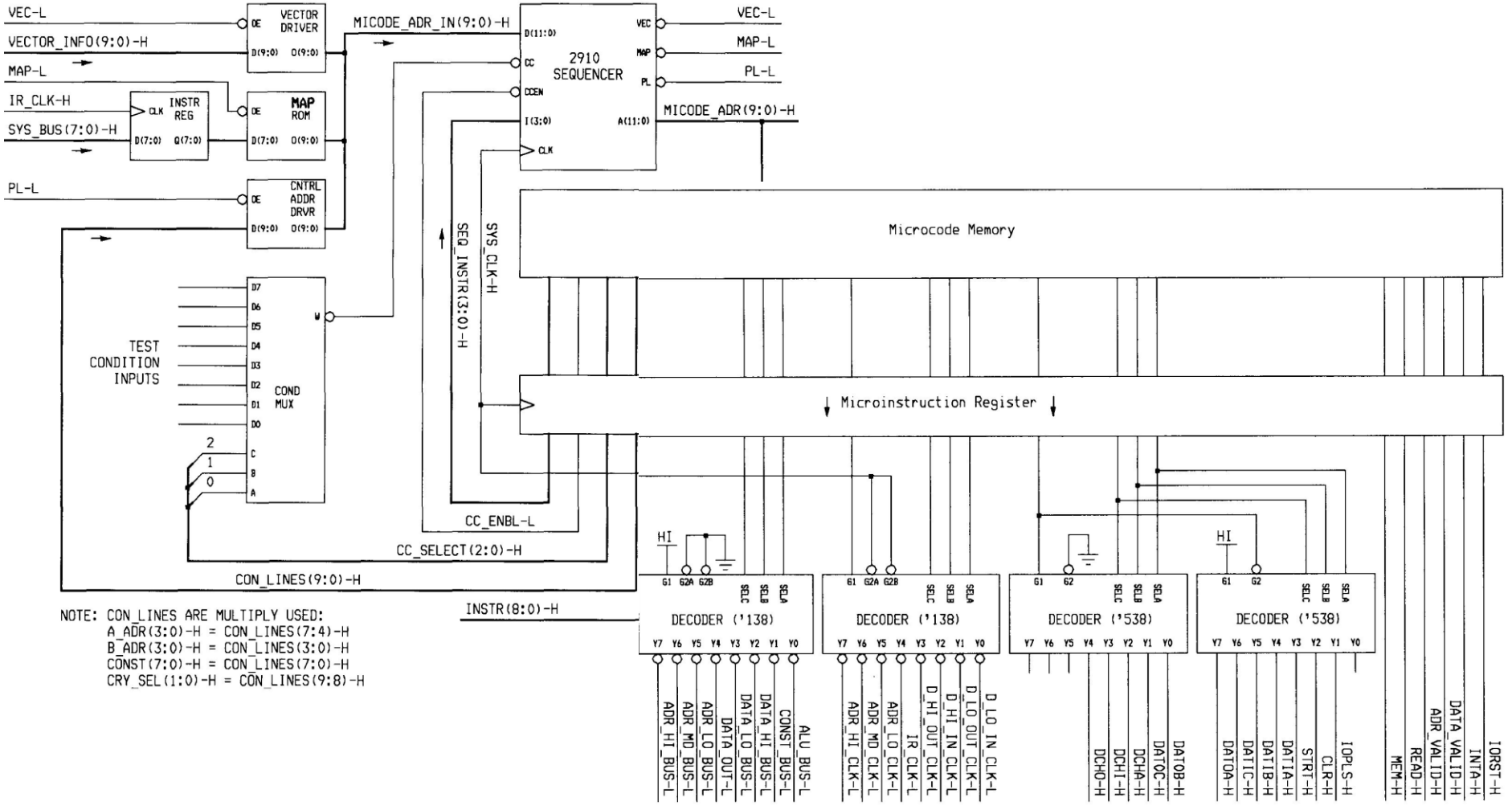
Horizontális mikrokódos vezérlő



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 közülük egyszerre csak az egyiket állítjuk be. A jeleket ezután **dekódolni** kell (a dekódolás több időt vesz igénybe). A kiválasztott biteket megpróbáljuk minimális számú vonalon keresztül továbbítani.
- A műveletek párhuzamos (konkurens) végrehajtása korlátozott. Dekódolás: N bites buszt, $\log_2(N)$ számú bittel próbálunk dekódolni. Több mikroutasítás szükségeltetik $\Rightarrow$ így a mikrokódú memóriát „**vertikálisan**” meg kell növeli.

Vertikális mikrokódos vezérlő

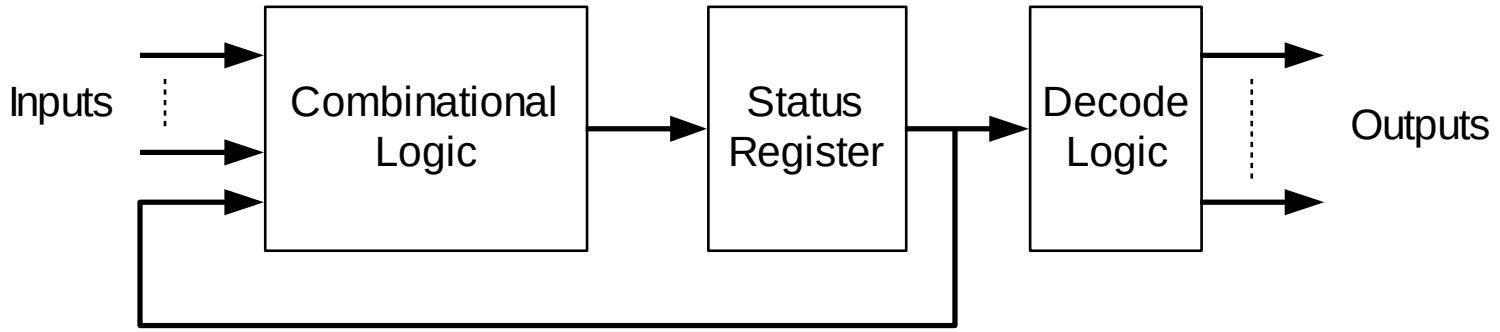


NOTE: CON_LINES ARE MULTIPLY USED:
A_ADR(3:0)-H = CON_LINES(7:4)-H
B_ADR(3:0)-H = CON_LINES(3:0)-H
CONST(7:0)-H = CON_LINES(7:0)-H
CRY_SEL(1:0)-H = CON_LINES(9:8)-H

Programozható logikai eszközök (PLD)

Állapotgép FSM tervezés tulajdonságai

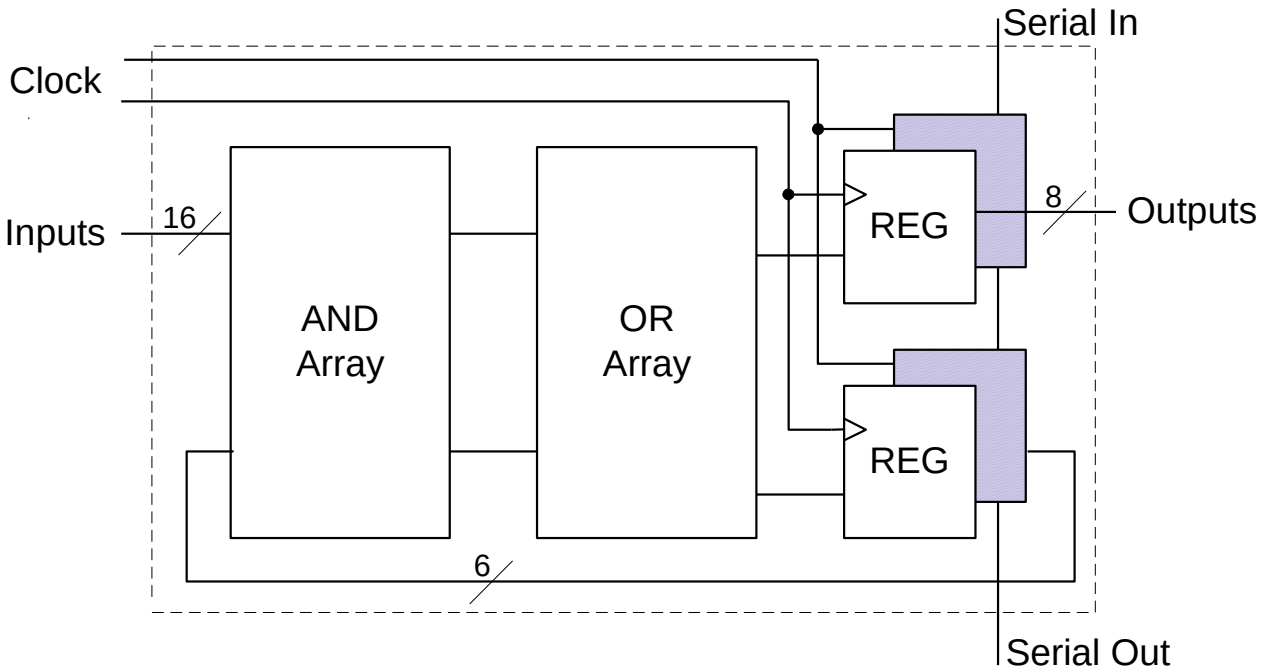
- Két kombinációs logikai hálózatból és egy regiszterből áll
- Tervezés során az állapot-átmeneteket vesszük figyelembe, **DE**
- Hibavalószínűség nagy,
- Szimulációs eszközök (Tools) hiánya,
- Hibák lehetségesek a prototípus fejlesztése során is,
- Könnyen konfigurálható / flexibilis eszközök szükségesek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



Field Programmable Logic Sequencer (FPLS) – Logikai sorrendvezérlő

- Egy vezérlő egység Next-State logikai blokkjának megvalósítása a tervező, gyártó feladata. Általában változó logikát használnak a függvények megvalósításánál.
- A felhasználó által programozható logikai sorrendvezérlő (**Field Programmable Logic Sequencer**) programozható alkatrészekből építhető fel, amelyek a következőkben részletesen ismertetésre kerülnek.

Field Programmable Logic Sequencer (FPLS) – Logikai sorrendvezérlő

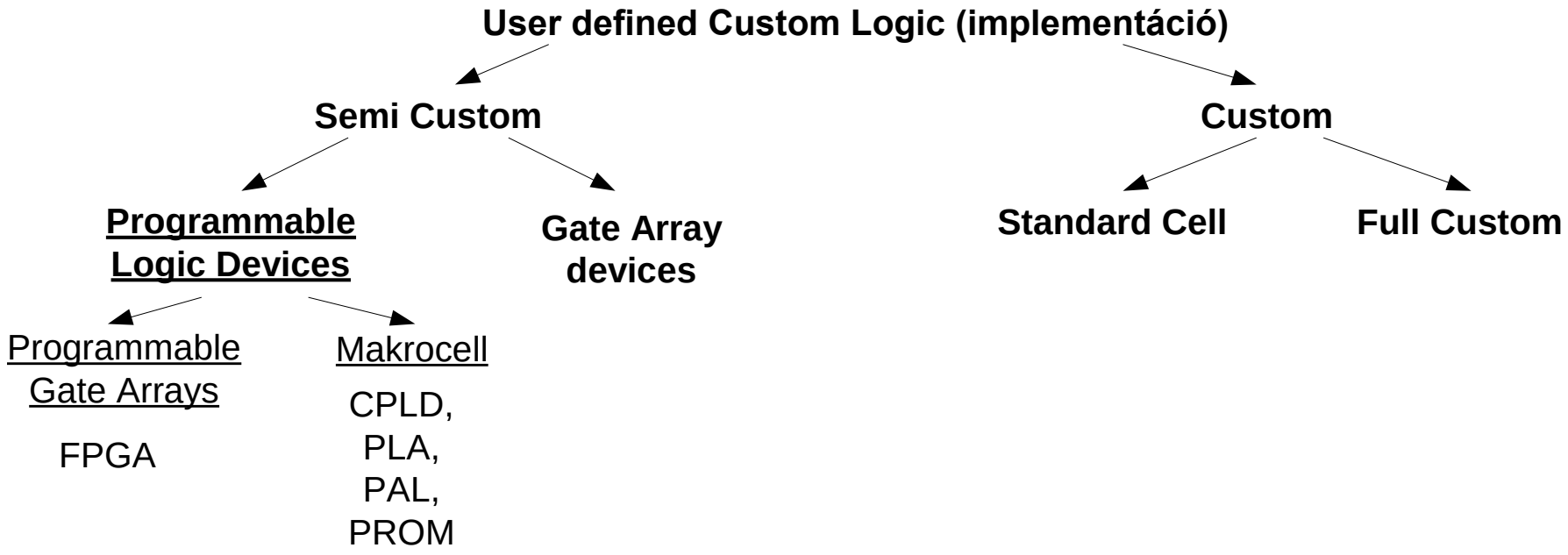


Működése:

- Normál (D-FF),
- Debug (Shift Reg.)

A logikai sorrendvezérlőnek 16 külső bemeneti vonala van, 1 RESET és 1 kimenet-engedélyező vonala, ill. 8 kimeneti vonala. A regiszterek egy-egy állapotot tárolnak, amelyek az órajel hatására a kimenetre íródnak, vagy a 6 belső, *visszacsatoló* vonalon keresztül visszacsatolódnak. A Next-State ill. a kimeneti szintek meghatározásánál programozható AND/OR tömböket használnak.

Felhasználó által definiált logikai implementációk:



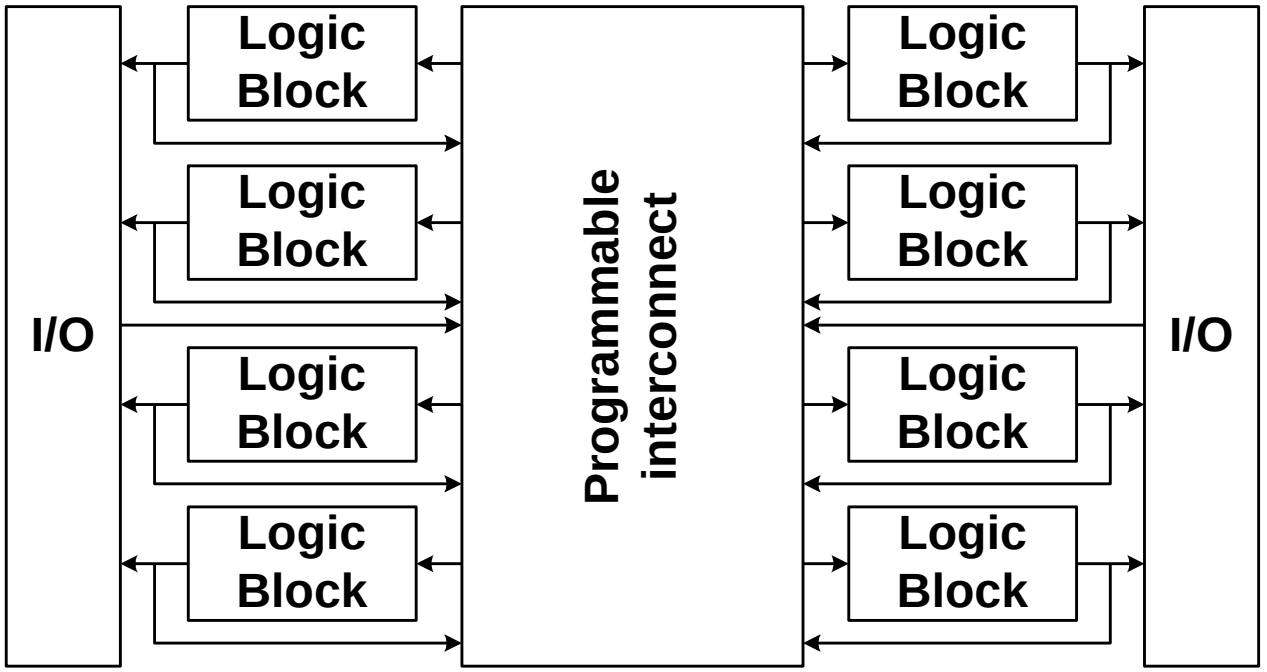
* Fontos: Ezek a részek nem találhatóak meg a könyvben!

Programozható logikai eszközök (PLD-k) két fő típusa:

- 1.) Makrocellás PLD-k: (Programmable Logic Devices):
  - PLA
  - PAL
  - PROM
  - EPLD, CPLD
- 2.) FPGA (Field Programmable Gate Array): Programozható Gate Array áramkörök
  - **XILINX** (Spartan, Virtex)
  - Altera,
  - Actel (főleg űrkutatásban alkalmazott) sorozatok

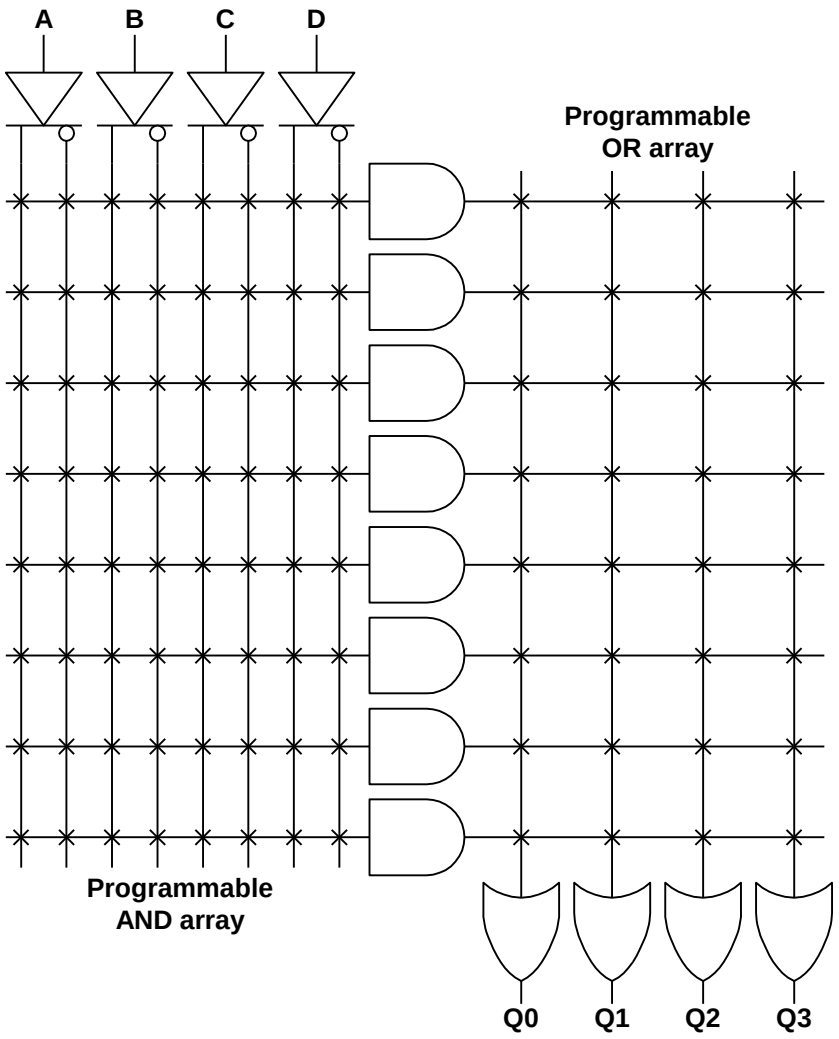
Complex Programmable Logic Device (CPLD)

- Logikai Blokk-ban
  - PAL / PLA
  - Regiszterek
- I/O Blokkok
- Programozható összeköttetések (Interconnect)
  - Teljes (Full-crossbar), vagy
  - Részleges összeköttetés hálózat



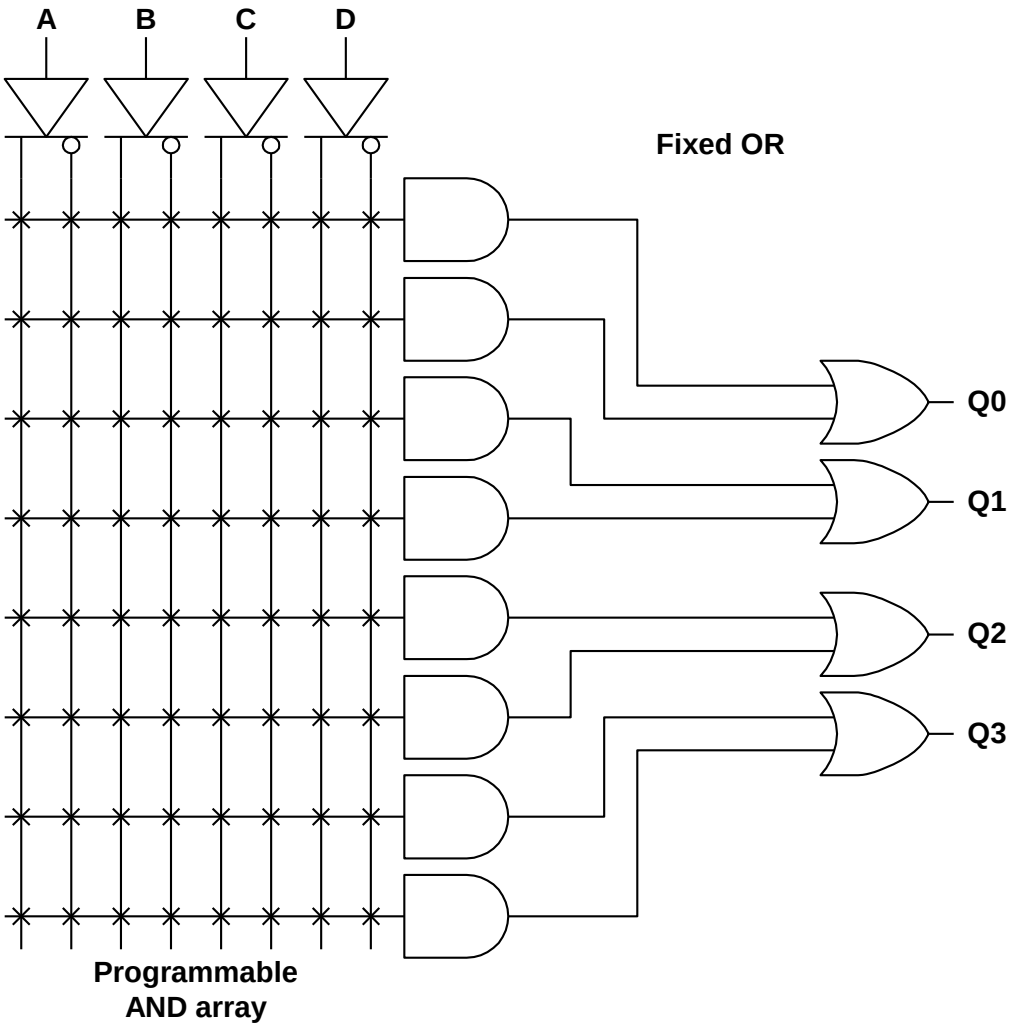
Programmable Logic Array (PLA)

- 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! (v.csat. a bemenetekre)



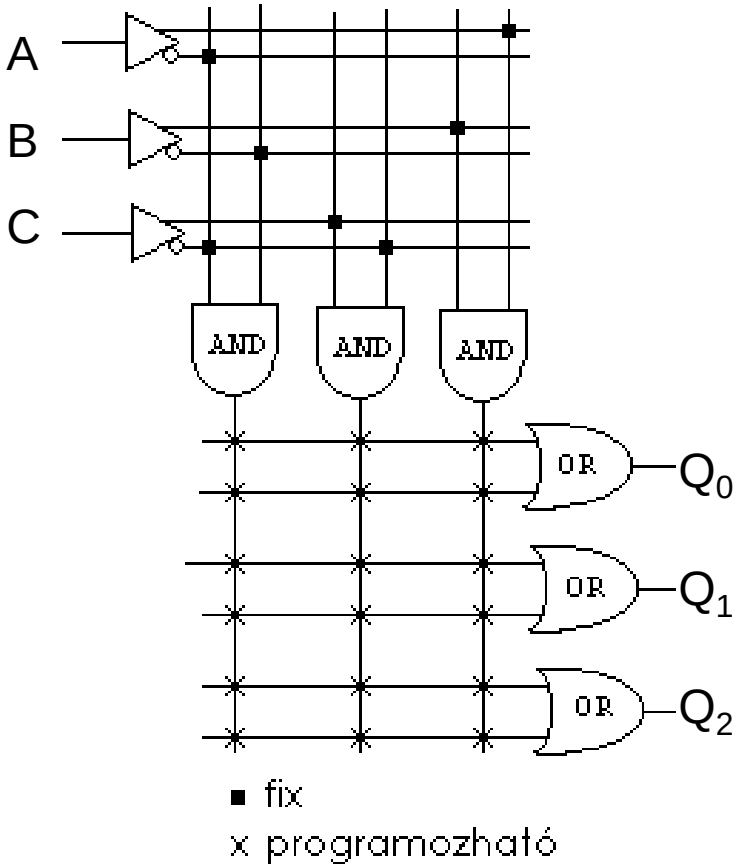
Programmable AND Logic (PAL)

- 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)



Programmable Read Only Memory (PROM)

- Egy programozható rész - OR / míg az AND 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)



Hogyan programozhatók a VSLI alkatrészek?

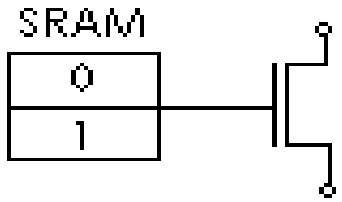
Programozási technikák

Programozási technikák

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

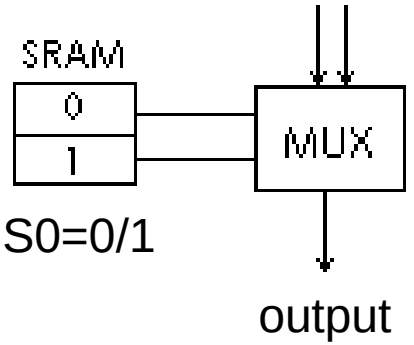
a.) SRAM cellás

- Tulajdonságai:
  - végtelen sokszor újraprogramozható (statikus RAM)
  - táp kikapcsolása után az SRAM elveszti tartalmát
  - bekapcsoláskor (inicializáláskor) a programot be kell tölteni, fel kell programozni
  - az SRAM cellára egy áteresztő tranzisztor van csatolva. A tranzisztor vagy kinyit (vezet), vagy lezár. Az SRAM értéke, ami egy bitet tárol (’0’ vagy ’1’) letölthető. Összeköttetéseket, vagy MUX-ok állását is eltárolja.
  - 1 bit tárolása az SRAM-ban (min. 6 tranzisztorból áll)
  - sok tranzisztor (standard CMOS), nagy méret, nagy disszipáció
  - nem kell frissíteni az SRAM-ot
  - nagy 0.5-2 kΩ átmeneti ellenállás
  - nagy 10-20 fentoF parazita kapacitás



b.) MUX - multiplexeres

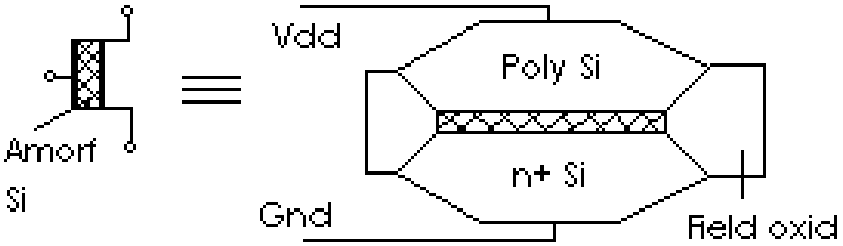
- Tulajdonságai:
  - az SRAM-ban tárolt ’0’ vagy ’1’ értéket használunk a Multiplexer bemeneti vonalának kiválasztásához. (Működése hasonló az SRAM celláéhoz.) /Bemenetek közül választ a selektáló SRAM-beli érték segítségével és a kimenettel köti össze./

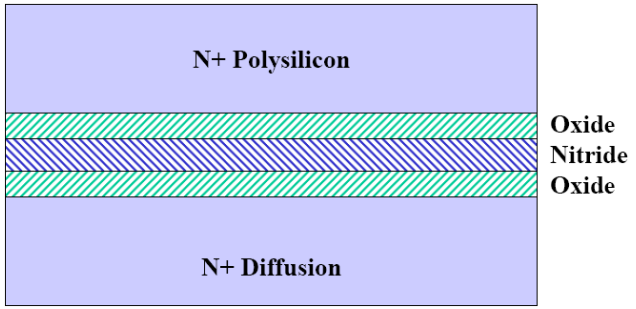


c.) Antifuse

A tranzisztor Gate-jét amorf kristályos Si alkotja, amelyet relatíve nagy feszültség (kb 20-30V) hatására átkristályosítunk, így vezetővé válik véglegesen. Pl. Texas Instruments alkalmazza ezt a technológiát. Tulajdonságai:

- az „átégetés” irreverzibilis folyamat, nem lehet újraprogramozni
- csak egyetlen egyszer programozható
- kis méreten megvalósítható, kis disszipáció
- kis átmeneti ellenállás 300 Ω
- kis parazita kapacitás 1.1-1.3 fentoF
- előállításához sok maszkréteg szükséges, drága technológiát igényel
- Típusai
  - ONO (Oxid-Nitrid-Oxid)
  - Amorf Si

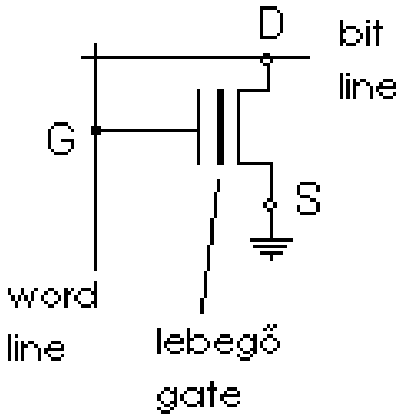


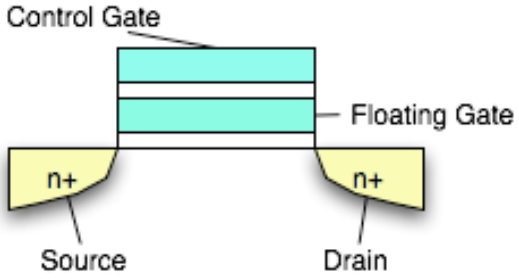


d.) Floating gate

Két-gates tranzisztort használunk, amelynek középső gate-je a lebegő gate. A másik gate fix. INTEL alkalmazza. Programozható összeköttetéseknél, csomópontokban használatos.

- Tulajdonságai:
  - programozása: töltéseket viszünk fel a lebegő Gate-re, kinyit a tranzisztor
  - írás: nagy feszültség hatására töltést viszünk fel a lebegő Gate-re
  - többször törölhető (kis ablakon keresztül UV fénnyel)
  - kikapcsoláskor is megőrzi tartalmát (non-volatile, akár 99 évig), töltések nem sülnek ki
  - nagy 2-4 kΩ átmeneti ellenállás
  - nagy 10-20 fentoF parazita kapacitás





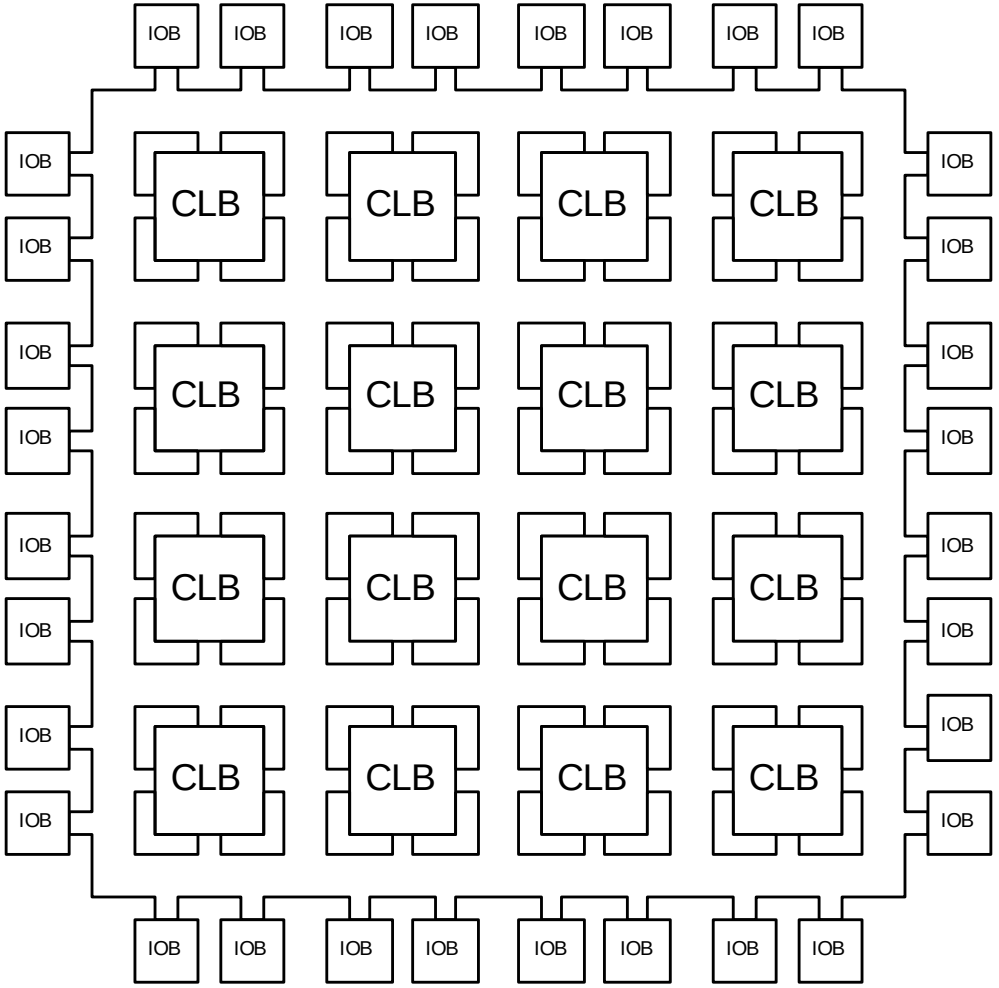
e.) EPROM / EPROM / Flash

- Tulajdonságai
  - **„Floating-gate” technológiát alkalmazza**!
  - 10000x szer programozható
  - Megőrzi tartalmát (Non-volatile)
  - UV-fénnyel törölhető (EPROM)
  - Elektromosan törölhető (EEPROM / Flash)
  - Nagy felület
  - Nagy parazita ellenállás
  - Nagy parazita kapacitás
  - További CMOS gyártási lépések (sok maszk réteg) szükségesek - drága

FPGA (Field Programmable Gate Array) architektúrák

Field Programmable Gate Array (FPGA)

- Konfigurálható Logikai Blokk (CLB)
  - Look-up table (LUT)
  - Regiszter
  - Logikai áramkörök
    - Összeadók (Adder)
    - Szorzók (Multiplier)
    - Memóriák (Memory)
    - Microprocessor
- Input/Output Block (IOB)
- Programozható összeköttetés hálózat

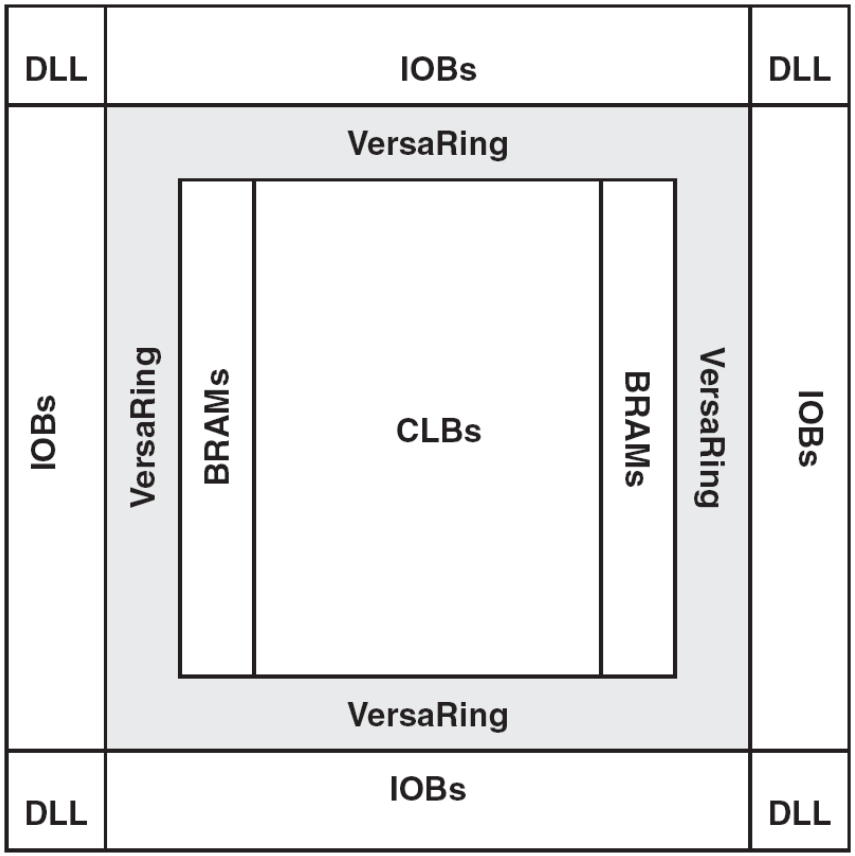


Xilinx FPGA családok

- Nagy teljesítmény
  - Virtex (1998)
    - 50K-1M gates, 0.22µm
  - Virtex-E/EM (1999)
    - 50K-4M gates, 0.18µm
  - Virtex-II (1999)
    - 40K-8M gates, 0.15µm
  - Virtex-II Pro/X (2002)
    - 50K-10M gates, 0.13µm
  - Virtex-4 (2004)
    - 50K-10M gates, 90nm
  - Virtex-5 (2006)
    - 65nm
- Alacsony költség
  - Spartan-II (2000)
    - 15K-200K gates, 0.22µm
  - Spartan-IIE (2001)
    - 50K-600K gates, 0.18µm
  - Spartan-3 (2003)
    - 50K-5M gates, 90nm

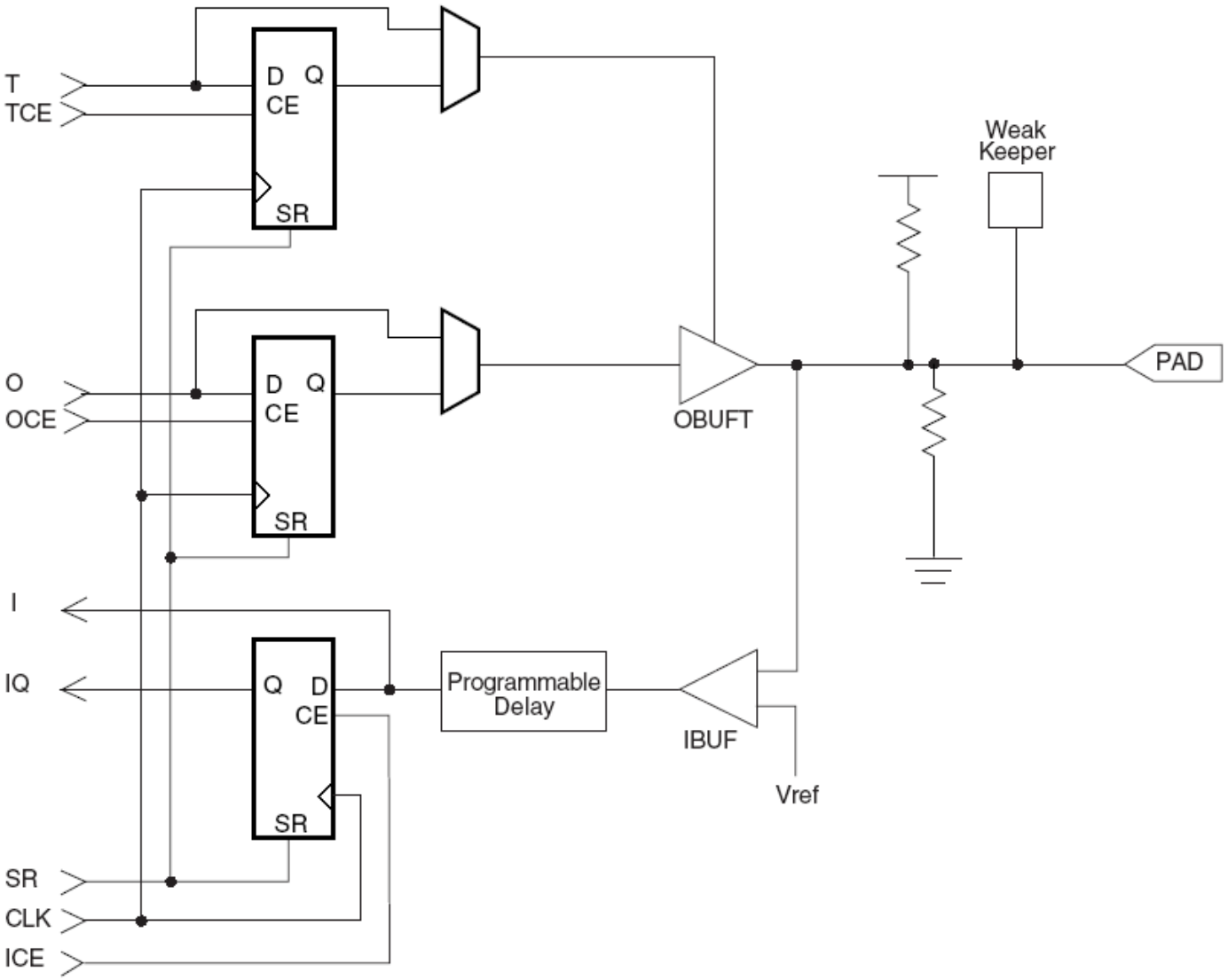
http://www.xilinx.com

Virtex architektúra

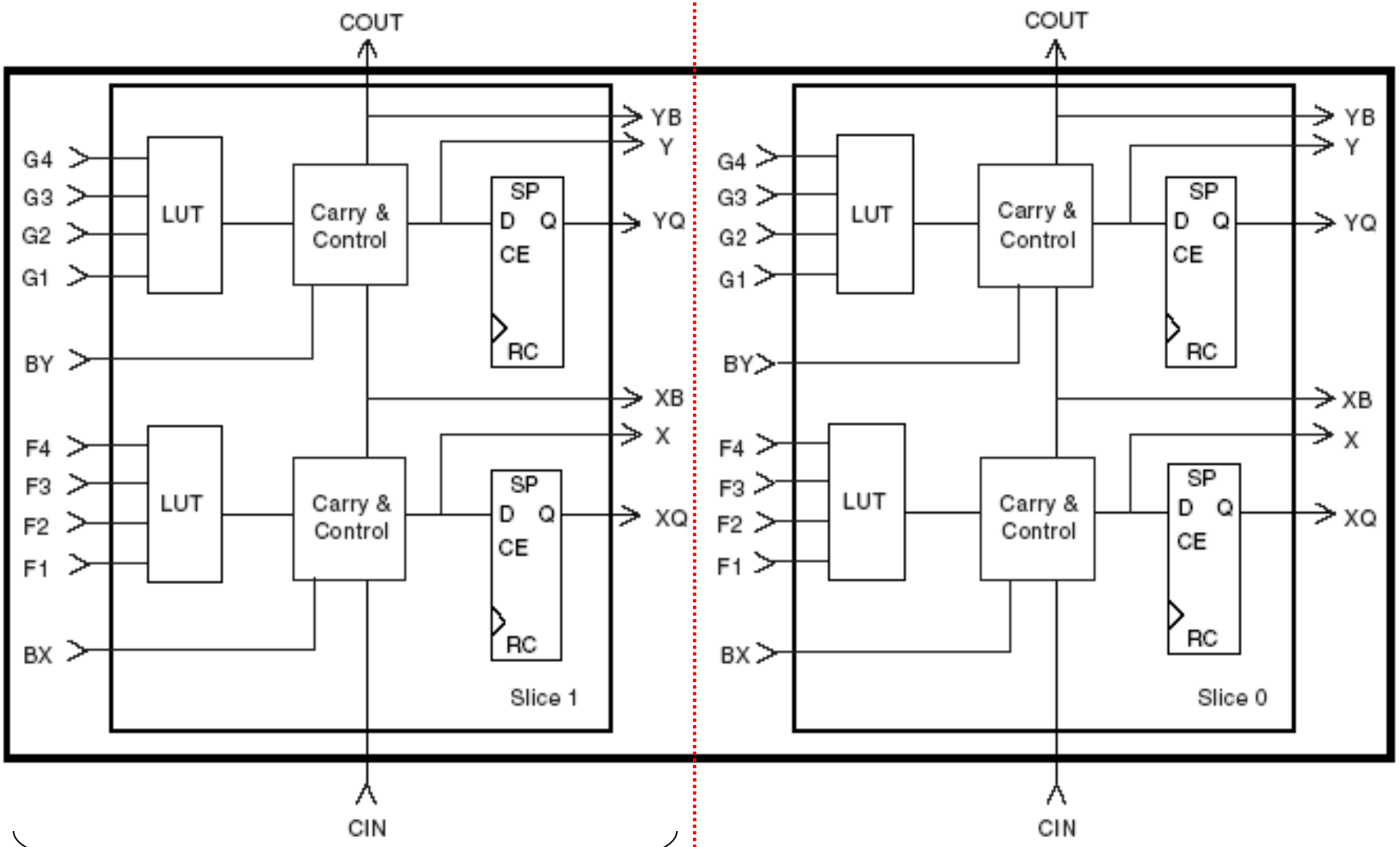


- CLB: Konfigurálható Logikai Blokk
- BRAM: Blokk SelectRAM
  - szinkron dual-portos 4096-bit RAM
- DLL: Delay-Locked Loop
  - Órajel menedzselési funkciók
  - Órajel osztás/ szorzás 1.5, 2, 2.5, 3, 4, 5, 8, vagy 16
  - Fázis tolás: 0º, 90º, 180º, 270º
- IOB: Input/Output Blokk
  - 13 különböző I/O szabványt támogat (pl: LVDS)

Virtex - (IOB) Input/Output Blokk



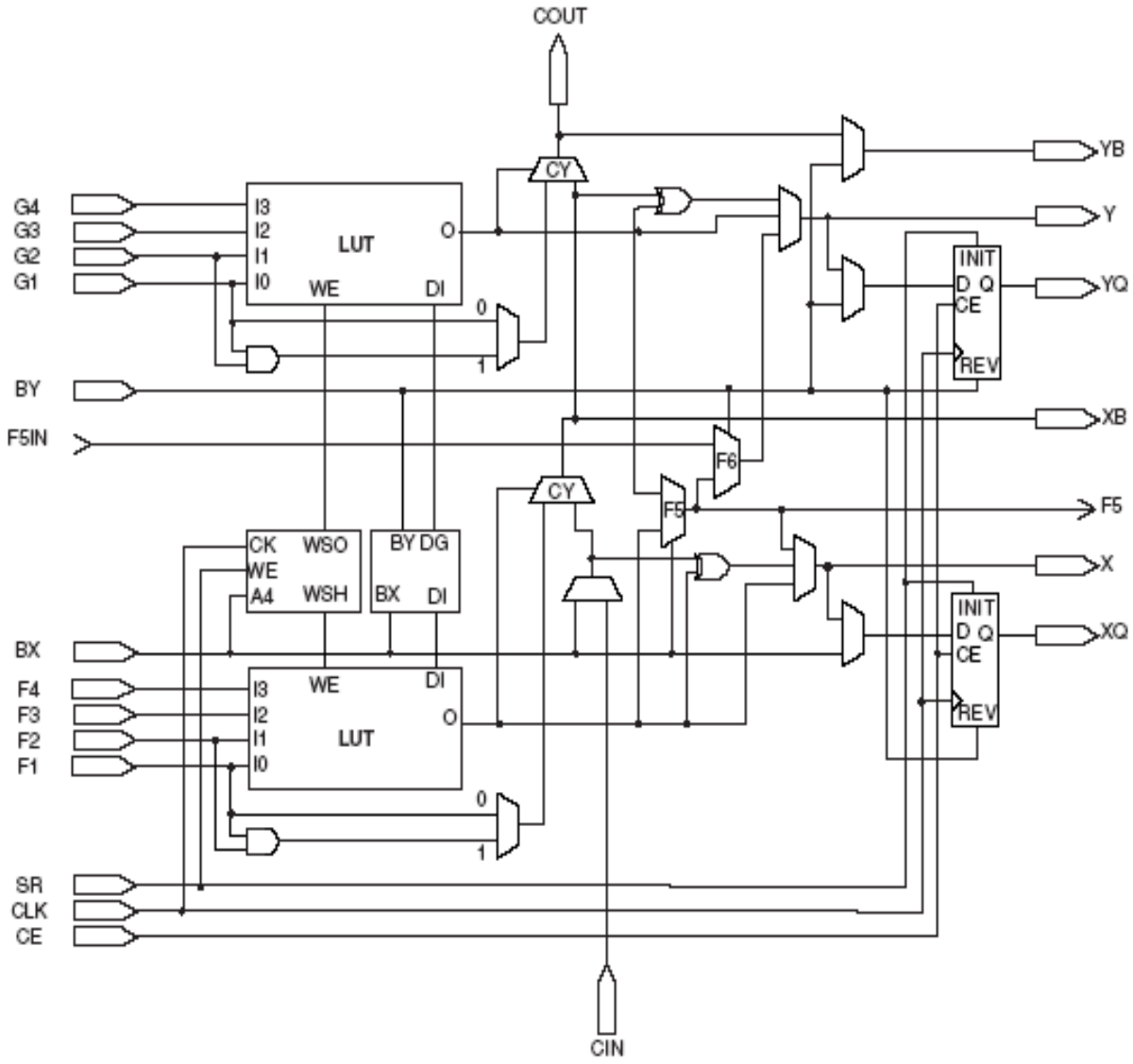
Virtex (CLB) Konfigurálható Logikai Blokk



„slice” - szelet

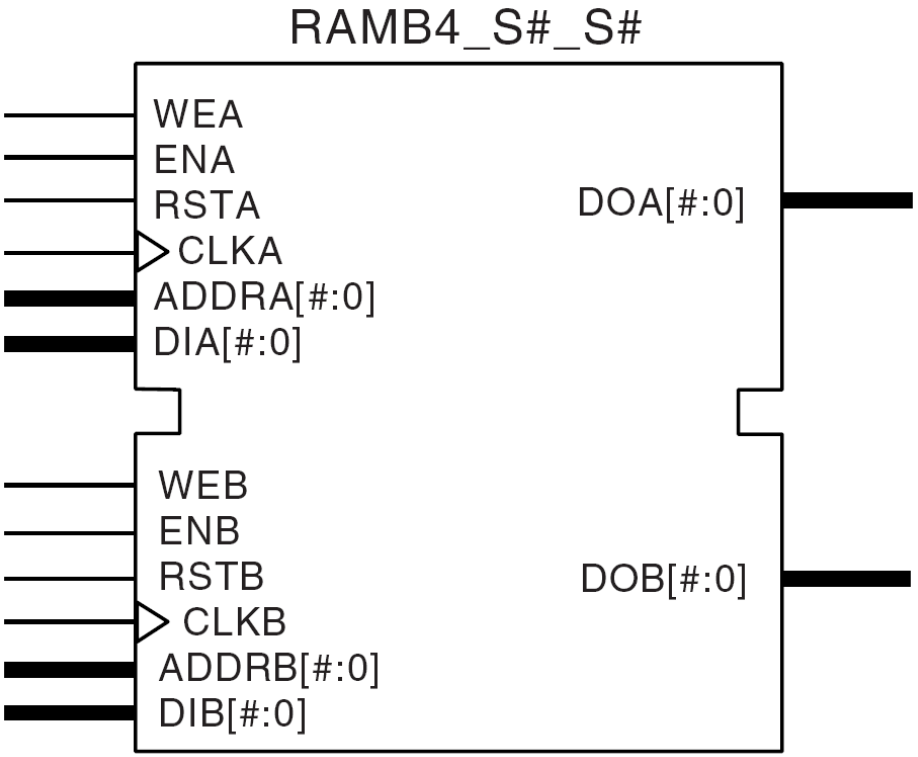
Virtex: Slice - szelet

FPGA-k esetében egy alapvető mérőszám a „slice” – szelet.



- Look-Up Table
  - 4-bemenetű LUT
  - 16 x 1-bit szinkron RAM
  - 16-bit shift regiszter
- Tároló elemek
  - ÉI-vezérelt D-típusú flip-flopok
- További Logikai áramkörök
  - F5 multiplexer
    - bármely 5-bemenetű függvény
    - 4:1 multiplexer
  - F6 multiplexer
    - bármely 6-bemenetű függvény
    - 8:1 multiplexer
- Aritmetikai Logikai Egység
  - Dedikált carry logika
  - Dedikált AND kapuk

Virtex Duál-portos Blokk SelectRAM

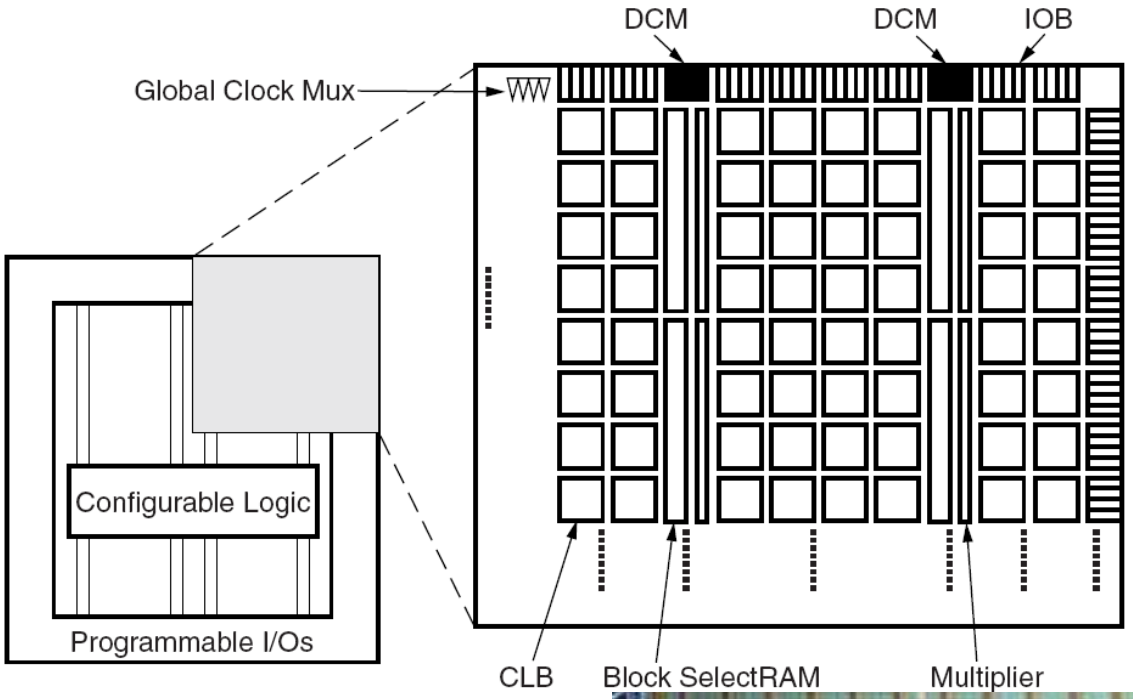


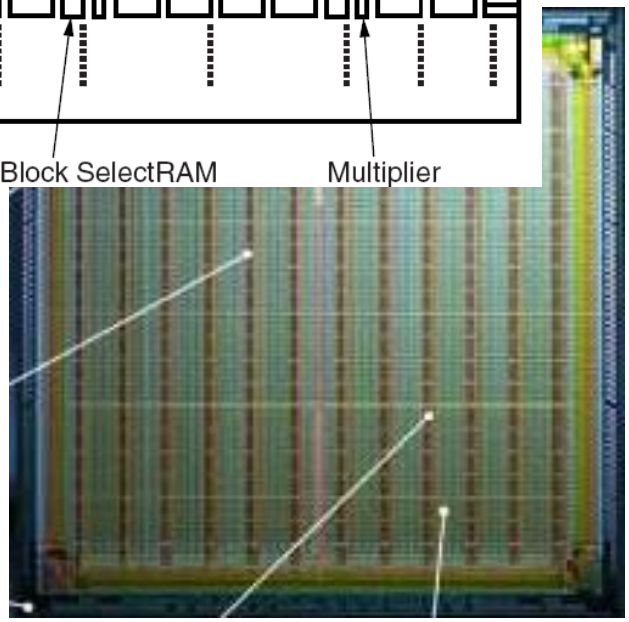
- Minkét portját (A, B) külön címezhetjük (RW)
- Független adatbusz szélesség definiálható:
  - Bus-szélesség konverzió

Table 4: **Block SelectRAM Port Aspect Ratios**

Width	Depth	ADDR Bus	Data Bus
1	4096	ADDR<11:0>	DATA<0>
2	2048	ADDR<10:0>	DATA<1:0>
4	1024	ADDR<9:0>	DATA<3:0>
8	512	ADDR<8:0>	DATA<7:0>
16	256	ADDR<7:0>	DATA<15:0>

Virtex-II architektúra





Dedikált blokkok:

- Blokk SelectRAM
  - 18Kb dual-port RAM
- Multiplier (szorzó)
  - 18x18-bit szorzó
- DCM (Digital Clock Manager)
  - Órajel ütemezés
  - Frekvencia szintézis
  - Fázis tolás stb.

Memóriák

Digitális Rendszerek és
Számítógép Architektúrák 2011

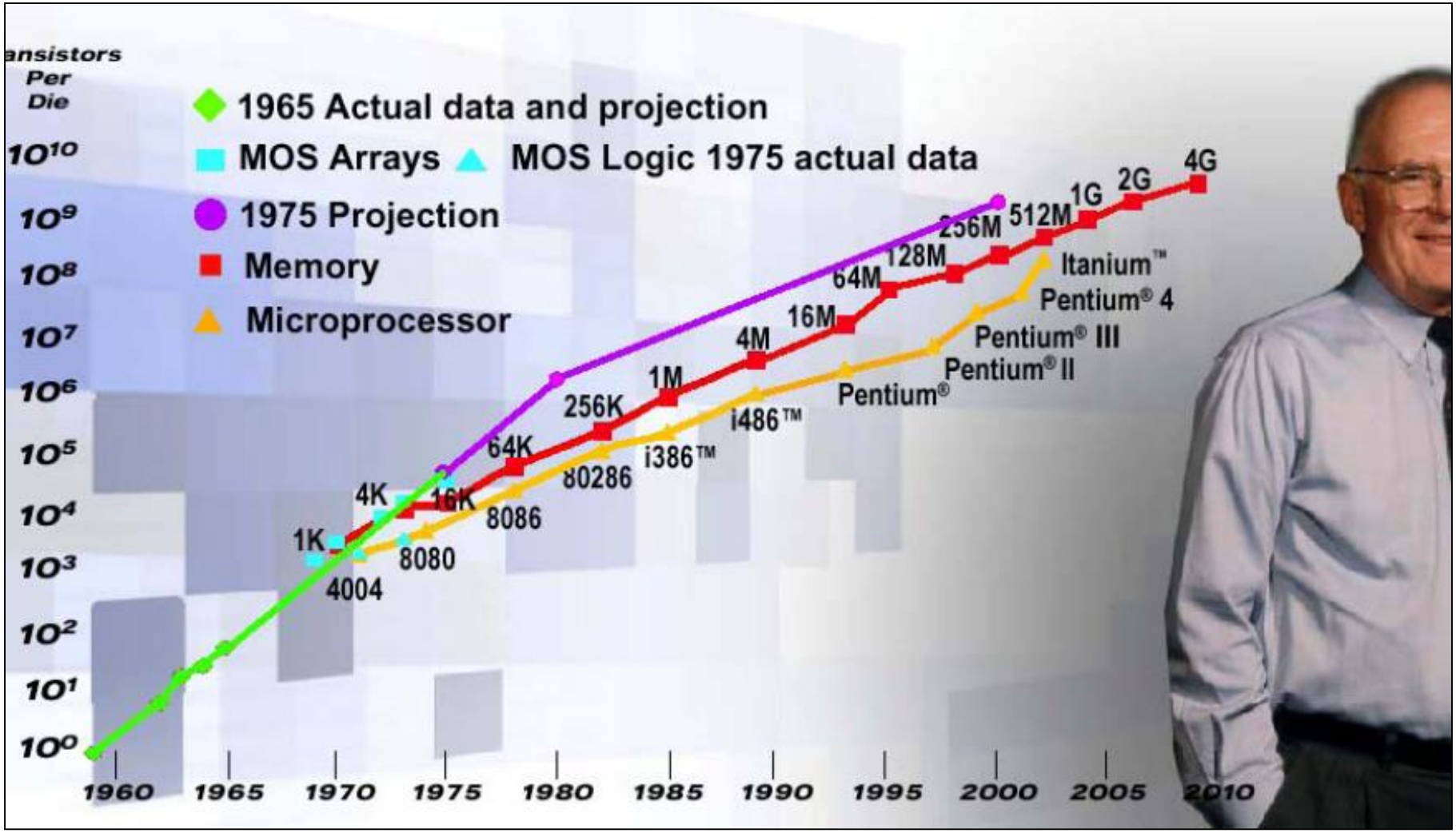
Memóriák, memória rendszerek

- RAM, ROM memóriák: nagyméretű, lineáris tároló tömb
- Virtuális memóriakezelés
  - Lapozás
  - Szegmentálás
- Cache memóriák

Mikro-minimalizálás elve:

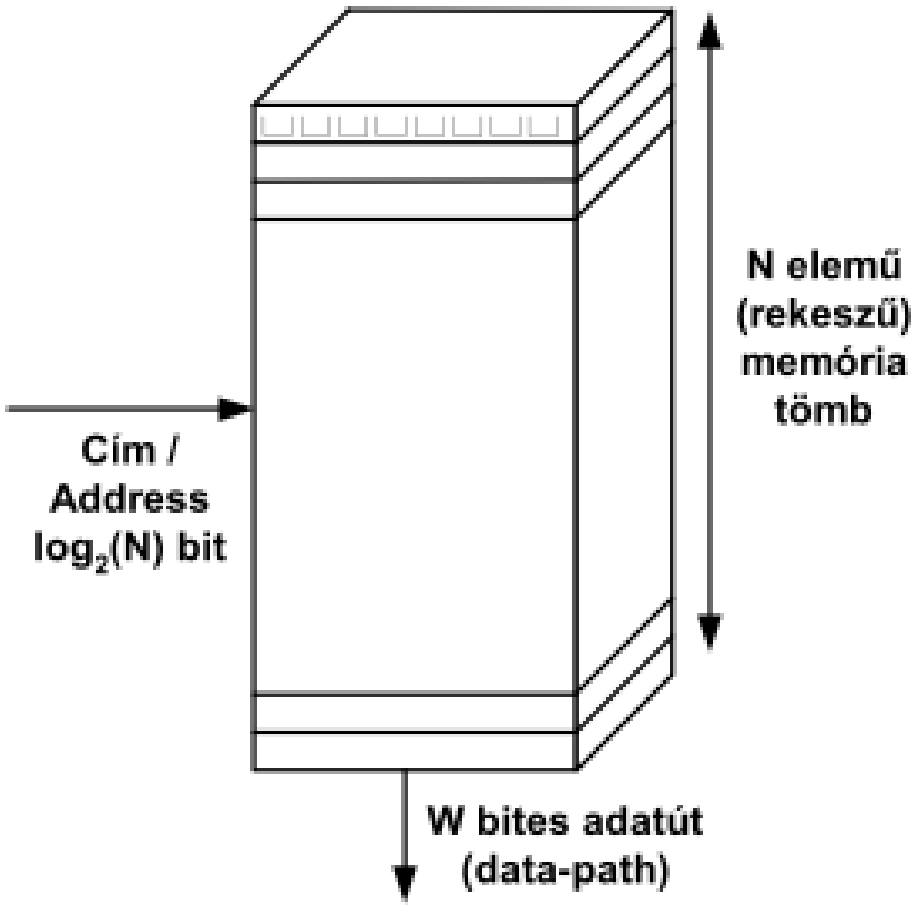
- **Gordon Moore törvénye (1965)**: rendkívül fontos a memóriák és a félvezető áramkörök méretcsökkenése esetén.
  - Tanulmány: félvezető áramkörök fejlődése (prognózis)
  - A technológia fejlődésével minden 18 hónapban az 1 felületegységre (mm² Si) eső tranzisztorok száma közel megduplázódik (integritási sűrűség)
  - Ezzel szemben az eszközök ára csökken, vagy stagnál.

Moore törvénye – „Road Map”



*Die: Si felület egysége (mm²) - megvalósított tranzisztorszám

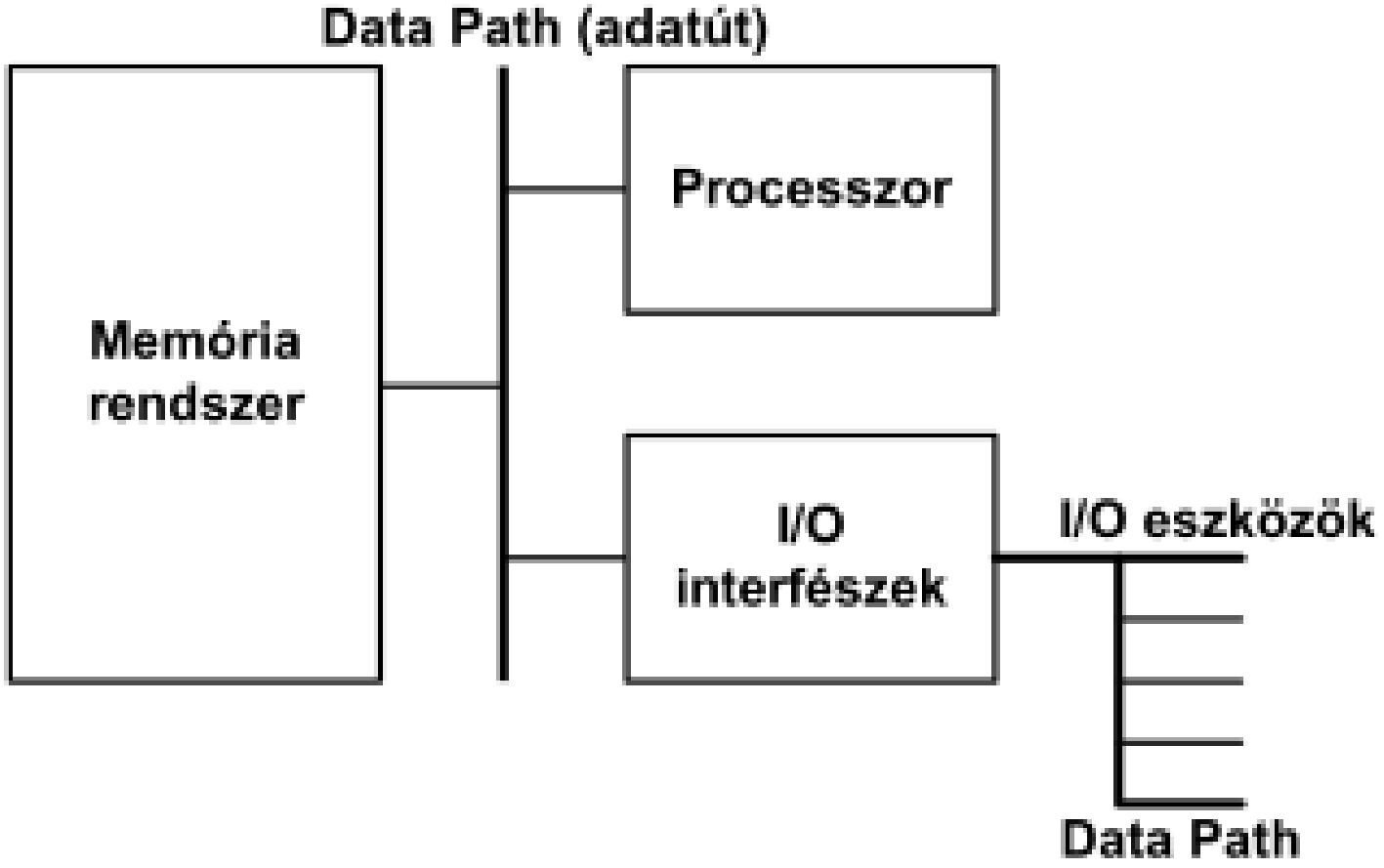
Általános memória modell:



Fontos paraméterek:

- memória mérete (N elemű, rekeszű)
- címvonalak száma N függvényében: log2(N)
- w: adatvonal szélessége
- memóriaszervezési módszer
- sebesség
- T_a: minimális (access time) elérési idő: az a legrövidebb idő, ami a cím kiadásától az adat megjelenéséig tart.
- T_{ct}: memória (cycle time) ciklus idő: minimális idő két elérés között (burst módban)

Memória – CPU – I/O interfészek kapcsolatának blokkdiagramja:



Tár-hierarchia:



• Regiszterek: a gyorsabbak, mivel fizikailag is a legközelebb vannak a processzorhoz: kis számú, nagyon gyors regiszterbankot használunk. De nagyon drágák. (ns)
• Cache memória: a leggyakrabban előforduló adatokat tárolja, növeli a műveleti (végrehajtási) sebességet
• Main Store/Memória: programok (kód) és az adatok itt tárolódnak, együtt vagy külön (Neumann v. Harvard elvű architektúra szerint!). Elegendő mennyiség szükséges az OS (operációs rendszer) zavartalan működéséhez. (ns-os elérési idő)
• Secondary Store/Disk. háttértárolók, másodlagos-merevlemezes tároló elemek, nagy kapacitásúak. Fájlok, könyvtárak, swap-terület. Hosszabb időre tárolunk. (ms-os elérési idő)
• Extended Storage/Tape: szalagos, lemezes meghajtók, mágneslemezek. Nagyon lassúak.

Memóriafajták elérésük és hozzáférésük szerint:

- **RAM:** Random Access Memory: Véletlen hozzáférésű memória, írható és olvasható
- **ROM:** Read Only Memory: Csak olvasható memória (véletlen elérésű - RAM tulajdonság)
- SAM Volatile Memória: Serial Access Memory. Soros elérési idejű, tápfesz. kimaradásakor felejtő ún. „elpárolgó” tartalmú memória
- PROM: Programmable ROM: programozható vezetékek közötti ellenállások/biztosítékok átégetésével ill. átkötésével.
- EPROM: elektromosan programozható fel, de csak UV-fény segítségével (kis kvarc ablakon keresztül) törölhető.
- EEPROM: elektromosan programozható és törölhető
- **SRAM:** Statikus RAM: tápfesz. nélkül elveszti tartalmát, gyors, nem kell frissíteni, de nagy hegyet foglal (min. 6 tranzisztor)
- **DRAM:** Dinamikus RAM: tápfesz. alatt is frissíteni kell tartalmát, lassú, de kis helyet foglal
- RWM: Read/Write Memory: írható/olvasható memória (spec Ram)
- CAM: Content Address Memory: tartalom szerint címezhető memória (asszociatív memória) Pl :CACHE memória.
- FLASH: memóriakártyák

RAM és ROM memóriák

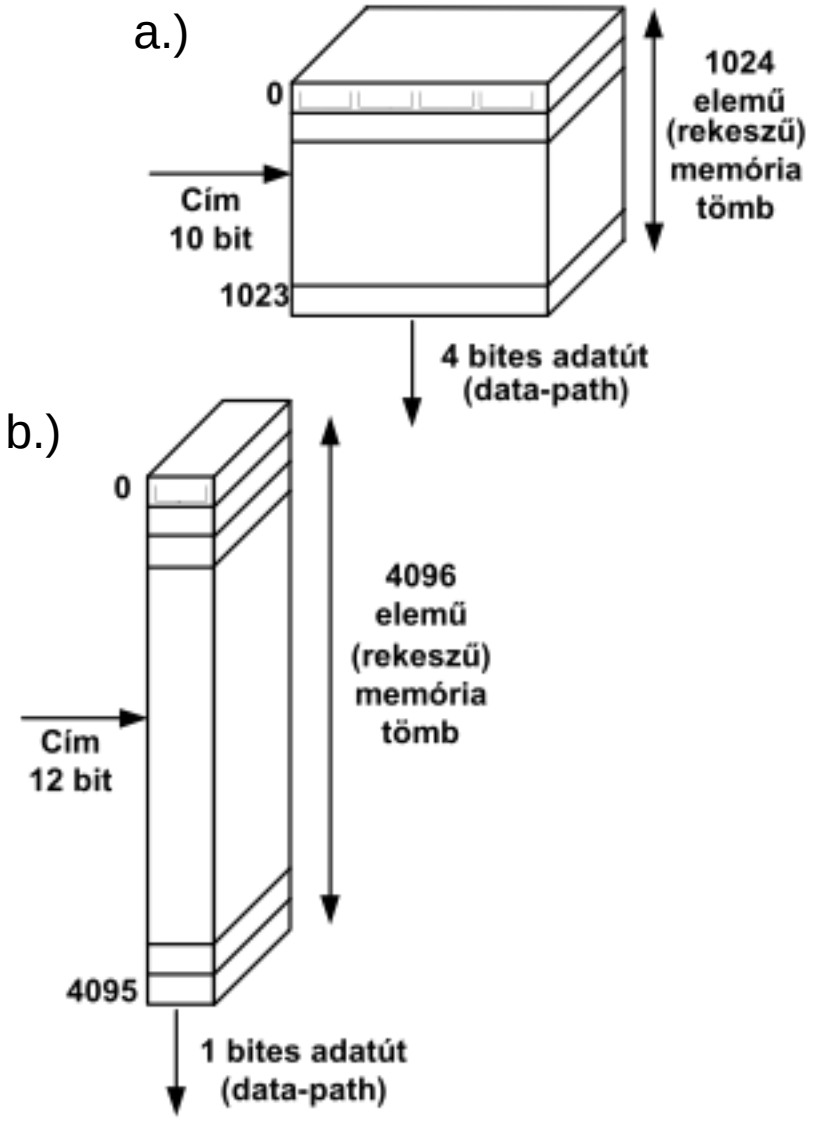
1.) ROM: Read Only Memory

- Csak egyszer írható (általában a gyártó által), utána már csak olvasható memória.
- Kikapcsoláskor megőrzik tartalmukat!
- Szervezése a RAM-hoz hasonló (de itt nem kell Write/Read vonal)
- Fontos alkalmazásai:
  - Firmware: elektronikai eszközök (pl. számítógép ROM-Bios funkcióinak tárolására)
  - Kód konverter: pl. BCD generáló 7 szegmenses kijelzőre
  - Logikai függvény generátor: tetszőleges logikai fgv. szintézisénél (előállításánál) ROM-ban tároljuk az igazságtáblázatot
- Programozható ROM fajtái:
  - PROM*: egyszer programozható
  - EPROM*: programozható, és UV-fénnyel törölhető
  - EEPROM*: elektromosan programozható/törölhető (Flash)

*Lásd: programozási módok

2.) RAM: Random Access Memory

- Véletlen hozzáférésű, írható és olvasható memória.
- Címzésnél: dekódoló és multiplexer áramköröket használunk
  - Pl. 10 lábbal -> 1024 (1K) cellát tudunk megcímezni
- Jelölés: RAM celláinak száma (4096) / tároló kapacitás 4K. Példák:
  - a.) 4K= 1K x 4 RAM tartalmaz: 1024 számú 4 bites szót (word)
  - b.) 4K= 4K x 1 RAM tartalmaz: 4096 számú 1-bites szót (word)
    (de itt az 1 bit/szó szervezéssel lábszámot spórolunk!) 12+1



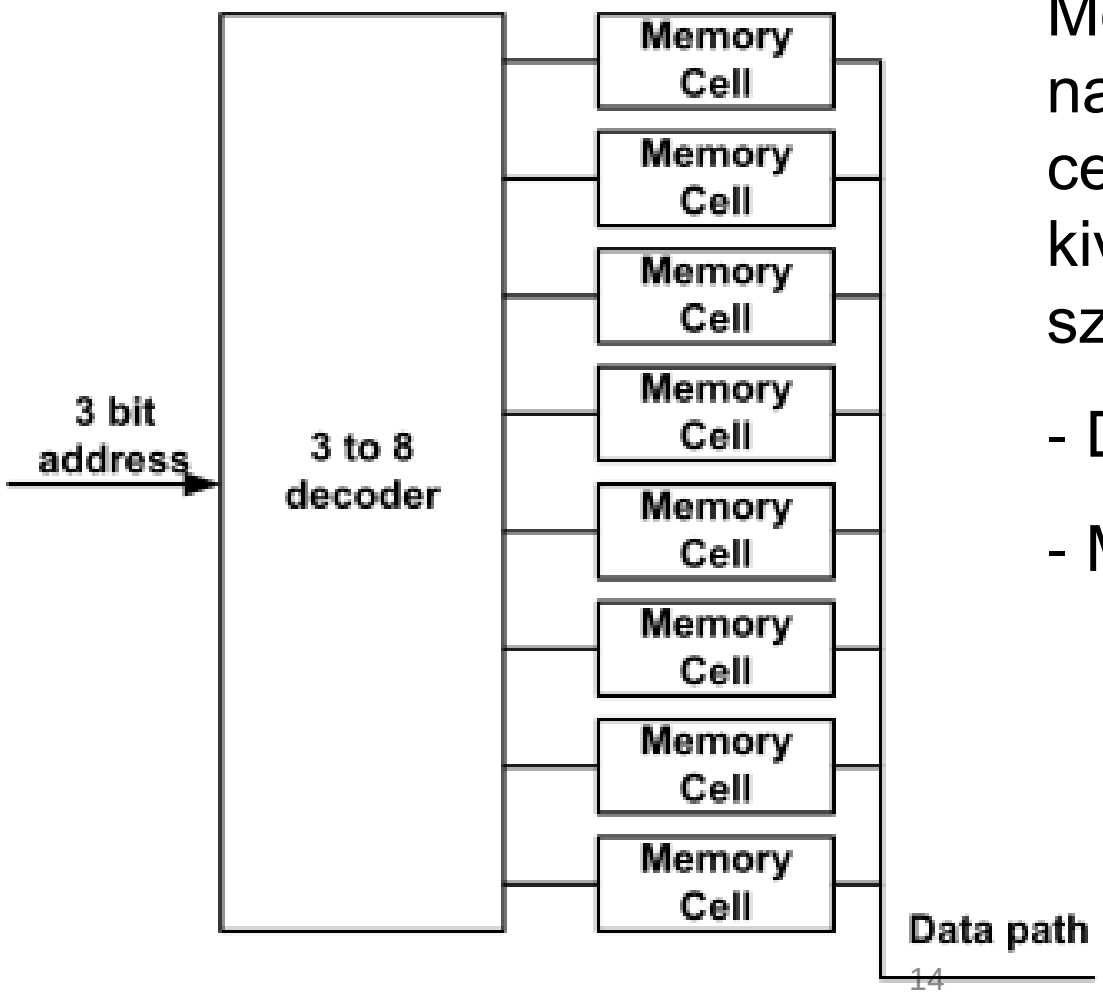
RAM időzítési paraméterei

- T(setup): T(access) elérési ideje a memóriának
  - (valid) érvényes adat megjelenése előtt a címet stabil értéken kell tartania
- T(hold): T(release): amíg az érvényes adatot kitartja az adatvonalon.
  - Ezután felszabadul a vonal és bizonyos ideig ismét stabil állapotot kell biztosítani a címnek.

RAM memória szervezés

- DRAM-ok esetén az Address (cím) felbontása:
  - nagyméretű, például „1M x 1” bites memória esetén a címet (20 bites) fel kell bontani sor és oszlopcímekre (idő-multiplexáltan)
    - Row address
    - Column address
  - Ezáltal 2D-s elrendezést kaphatunk.
- SRAM-ok esetén általában csak egy jelet (CS v. CE) használunk az adott cella azonosításához. (általában nem multiplexáltak)

RAM blokk diagram (1-D szervezés)



Memória szervezés célja: a nagyméretű (nagyszámú cella) memóriaelem kivezetéseinek (lábak) számának csökkentése!

- Dekóder

- Multiplexer áramkörökkel

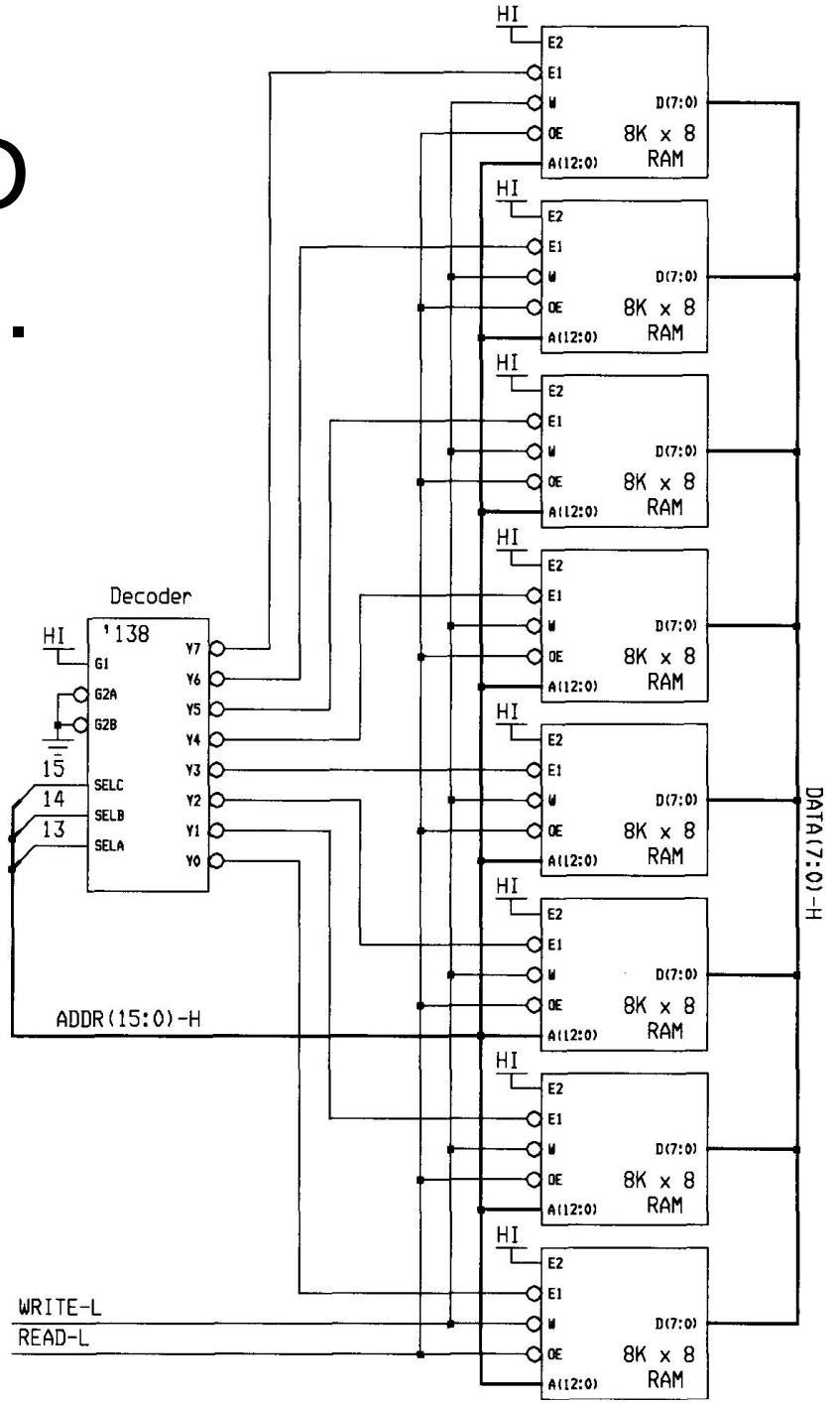
Példa: 8Kx8 – 1D szervezésű Mem.

1 cella: 2^16=64k

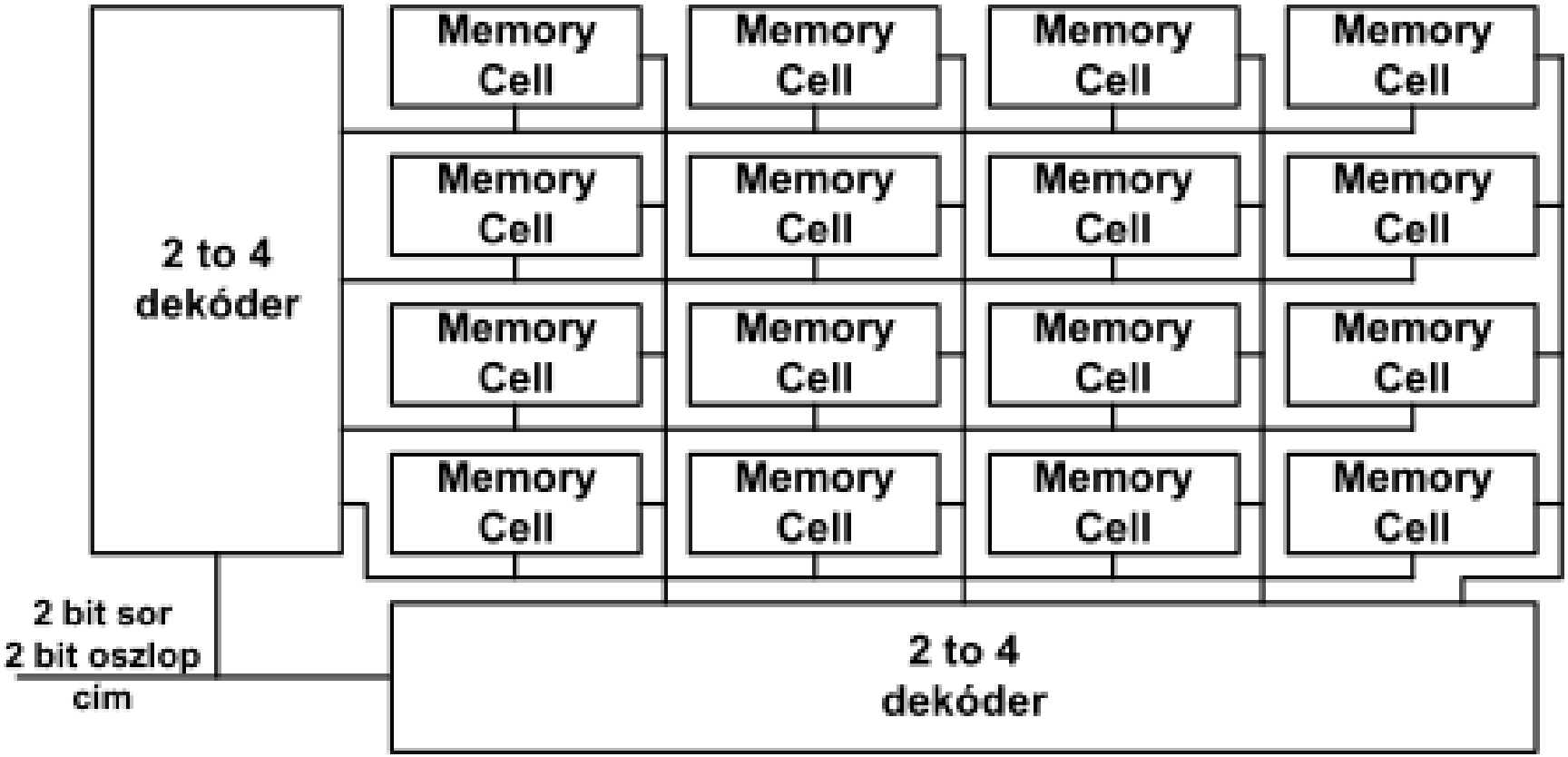
Össz: 512 K

w = 8-bites adatbusz

N= 16-bites címbusz



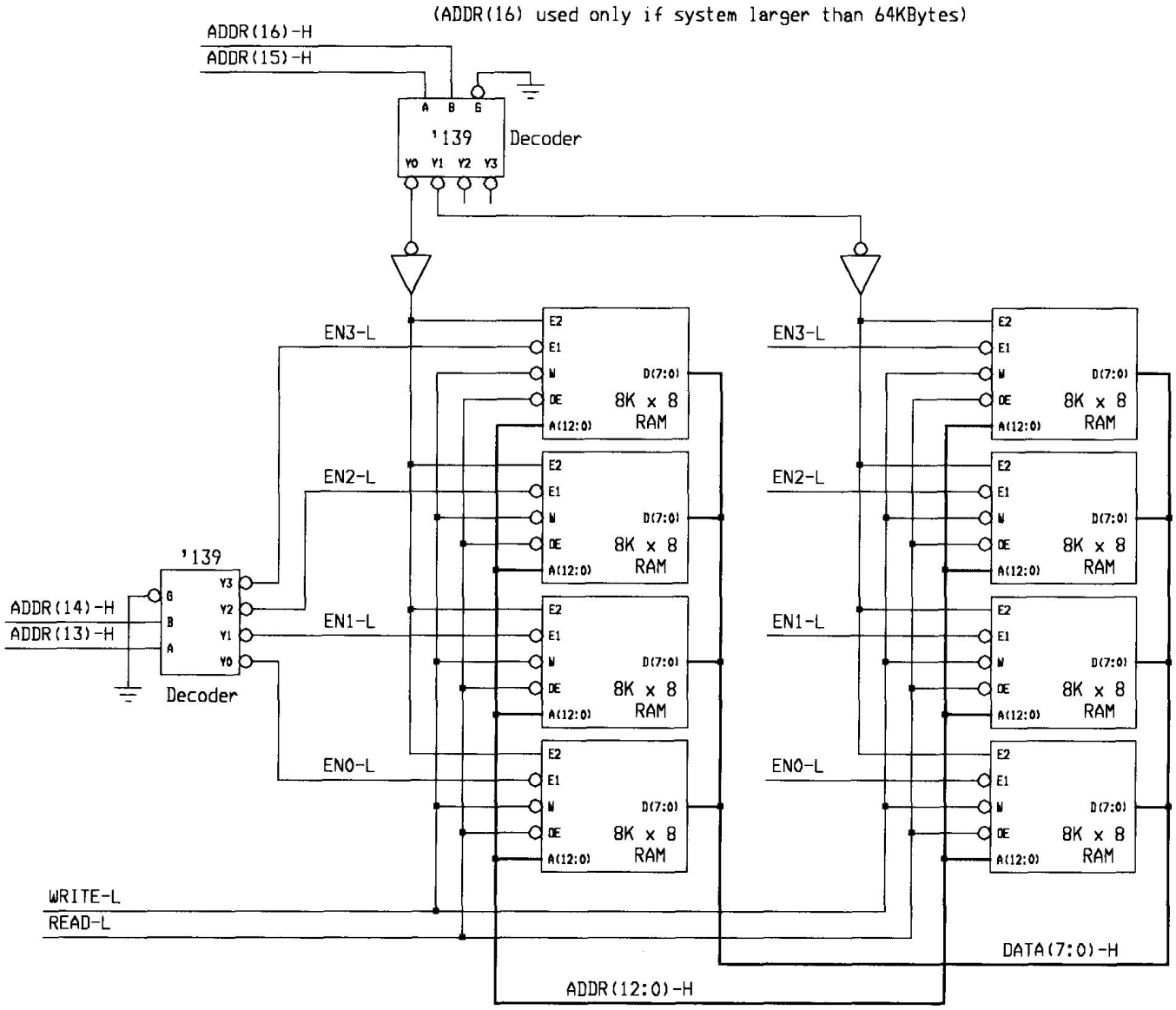
Memória cellák 2-D szervezése



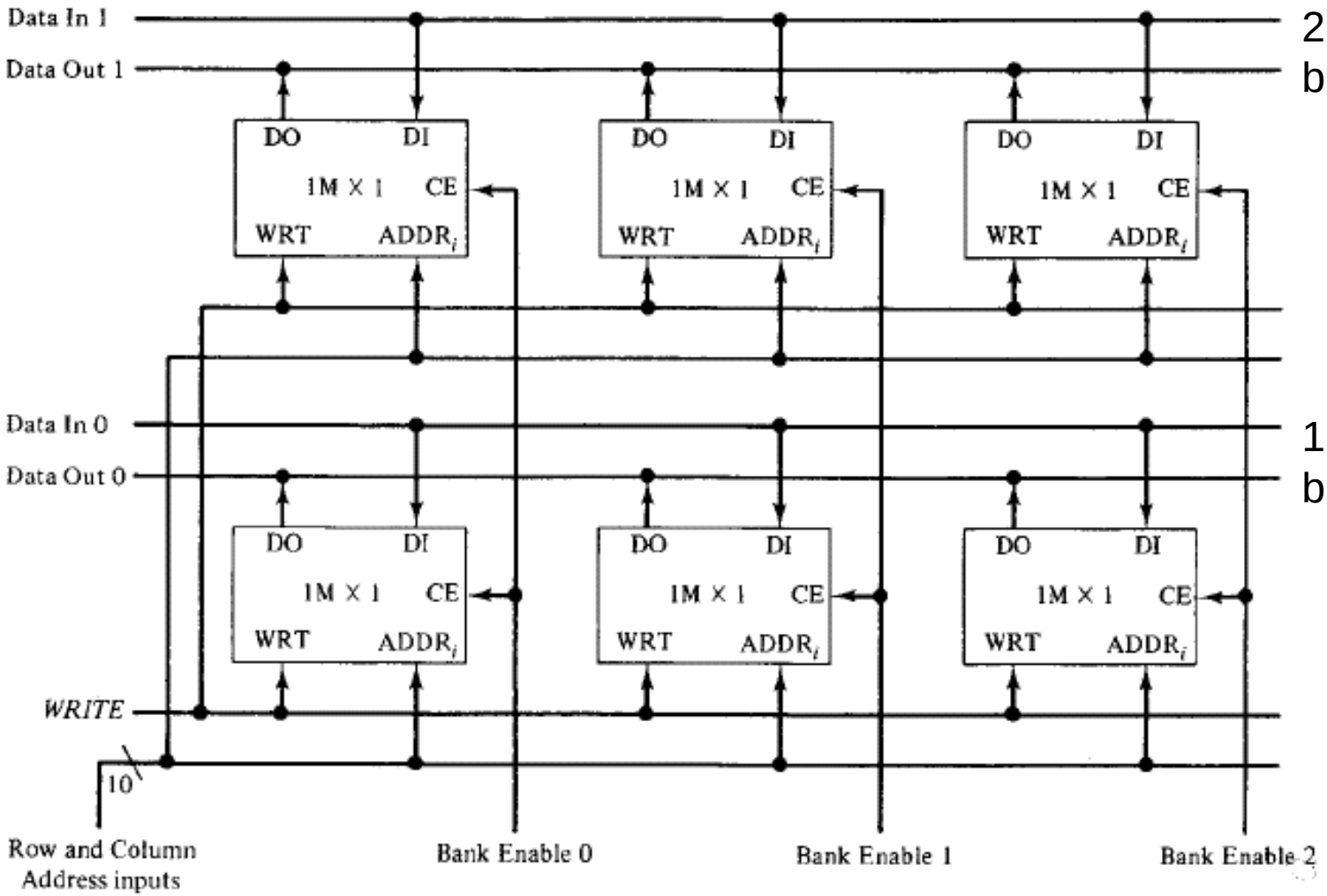
Példa: 8Kx8 2D szervezésű Mem.

2^16=64k

Össz: 512K



Példa: 3Mx2 memória 1Mx1 –ből felépítve



2. bit/word

1. bit/word

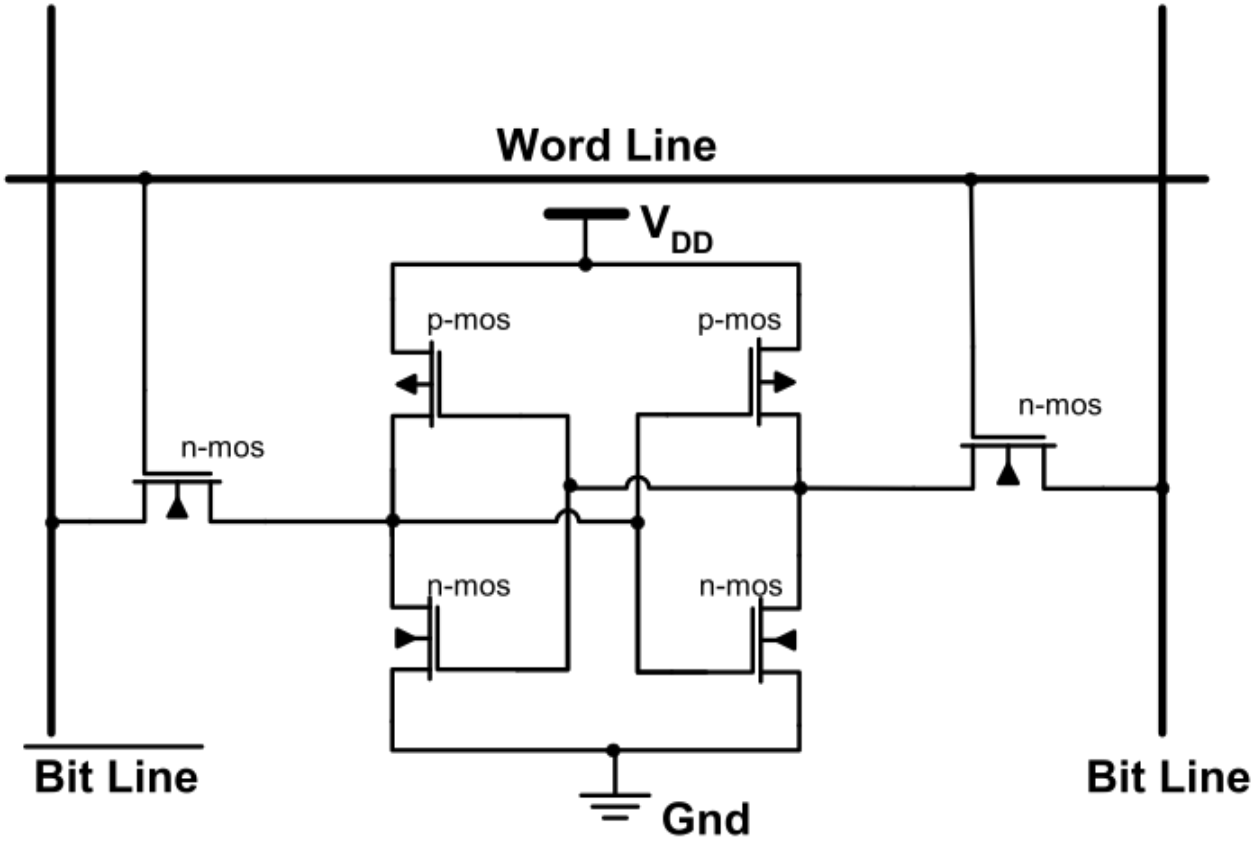
Írás: WRITE='T'

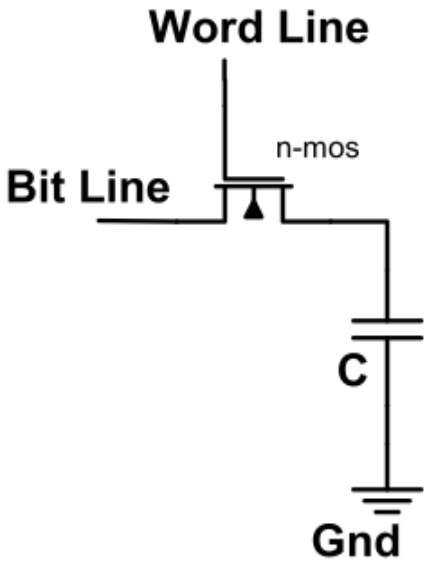
Olvasás:W WRITE='F'

1M = 20 bites cím (10 sor + 10 oszlop cím) 2-D s szervezés

3 Bank: BE engedélyező jel (melyik bank kapcsolódik a buszra)

RAM: **SRAM** (**Statikus**) és **DRAM** (**dinamikus**) memória cellák felépítése:





SRAM: n-mos és p-mos (n és p csatornás tranzisztorokból épül fel) 2-2 db, + 2 db áteresztő tranzisztor (össz. 6 CMOS tranzisztor)

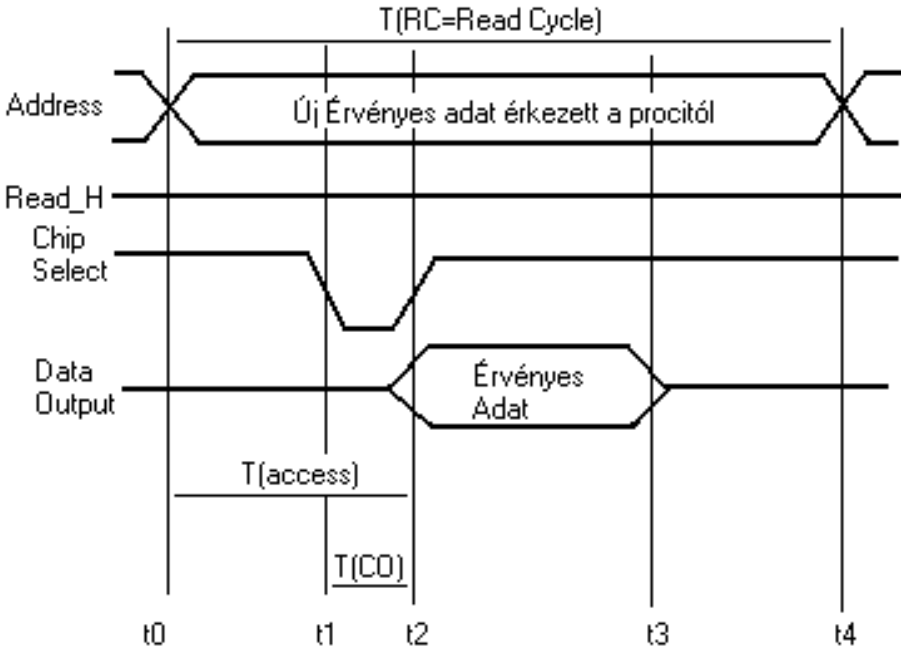
DRAM: Tárolás: egy kisméretű kondenzátoron (C), töltése és kisülése.

a.) SRAM tulajdonságai

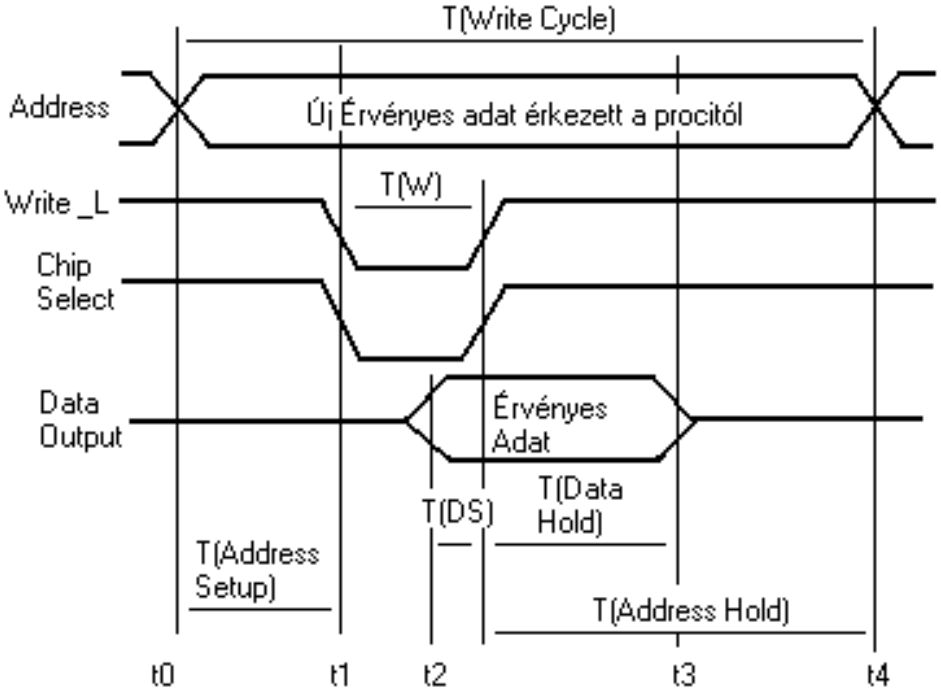
- Az információ a tápfesz. alatt is megmarad, nem kell frissíteni.
- SRAM esetén a ciklusidők T(R/W Cycle)≈T(Access Time) közel azonosak.
- Megvalósítható bipoláris (0,1) SRAM cellával, nMOS tranzisztorokkal, vagy CMOS tranzisztorokból (6 tranzisztor).
- Kisebb kapacitású, de gyorsabb a DRAM-nál, mivel tápfeszültség alatt sem kell frissíteni,
- Nagy a fogyasztása. Integritási sűrűsége 4x kisebb. Tápfeszültség kikapcsolásával elveszti a tartalmát.
- Felhasználása: cache memóriákban, digitális oszcillátorokban, logikai analizátorokban, operatív tárakban (memória).

Statikus RAM idődiagramjai:

Olvasási ciklus



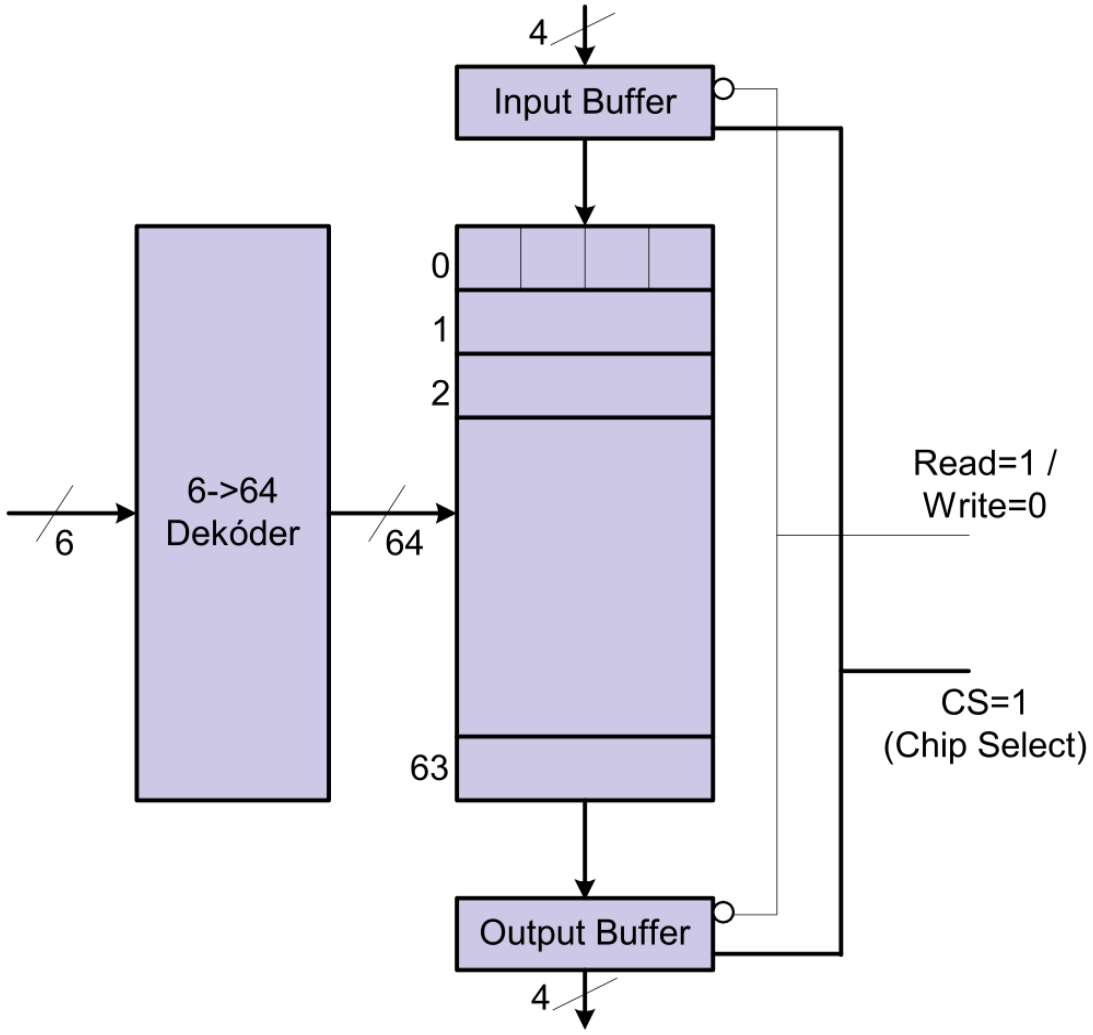
Írási ciklus



Példa: 64x4-es SRAM felépítése

- Feladat: Mekkora **lábszámot** kell biztosítani az SRAM működéséhez?

- **6+4+1(CS)+1(R/W)+ 1(Vdd)+1(Gnd) = 14**
- Tehát összesen 14 lábra (pin) van szükség!
- (64 rekesz, 4bit/szó)

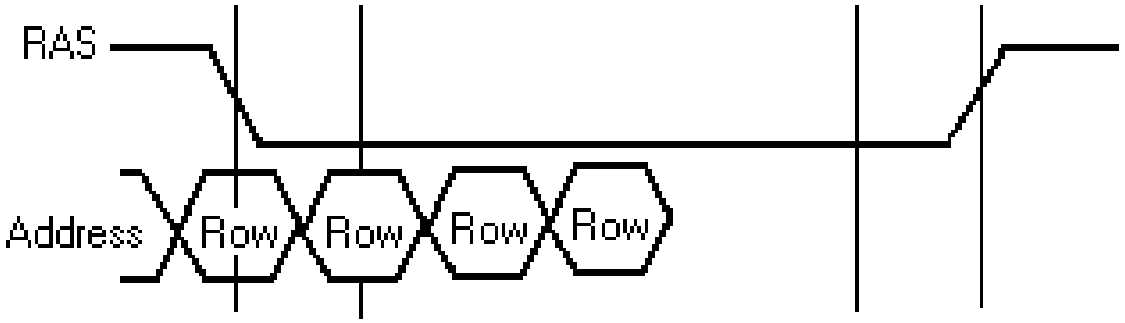


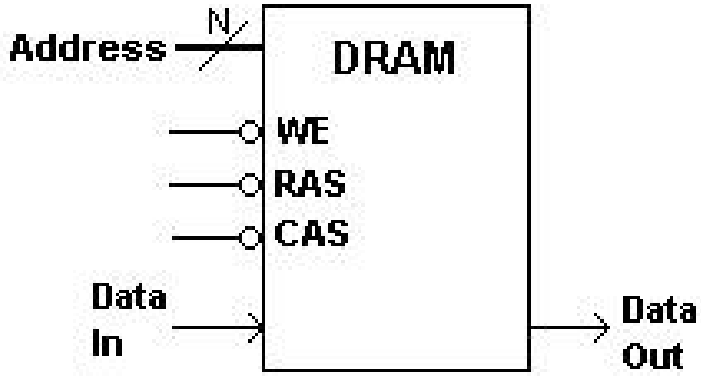
b.) DRAM tulajdonságai

- Általában CMOS technológiával készülnek.
- A tápfeszültség alatt is frissíteni kell 2-10 ms-ként, mivel idővel elvesztik tartalmukat.
- Nagyobb kapacitású, mint az SRAM, de méretben is nagyobb, bonyolultabb és lassabb is. A hozzáférési idő kétszer nagyobb a memória R/W ciklusidejénél: 2*T(R/W Cycle)=T(Access Time).
- Integritási sűrűsége **4x** nagyobb. (IRAM: az időzítő elektronika a DRAM-ra van integrálva, úgy látszik, mintha SRAM lenne, a frissítést tekintve).
- Itt lényegében a CS=Chip Select (korábban CE: Chip Enable!) jelet két részre osztottuk fel: RAS=sorkijelölő, és CAS=oszlopkijelölő komponensekre.

DRAM frissítése

- Kb. 2-10ms-ként kell frissíteni, mivel egy kisméretű kondenzátoron tároljuk az információt, melynek feszültsége idővel exponenciálisan csökken, tehát kisül.



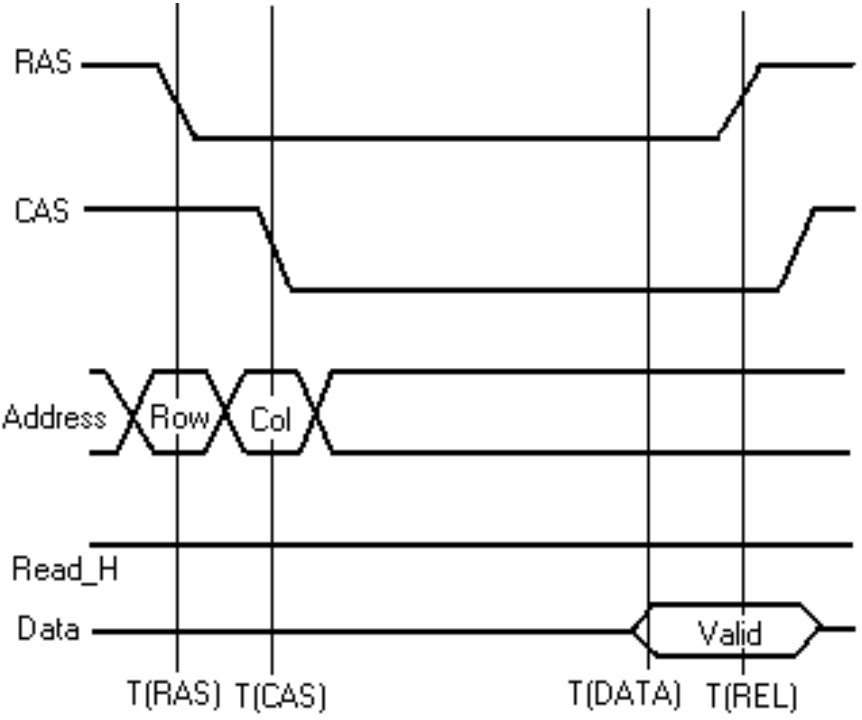


szimbóluma

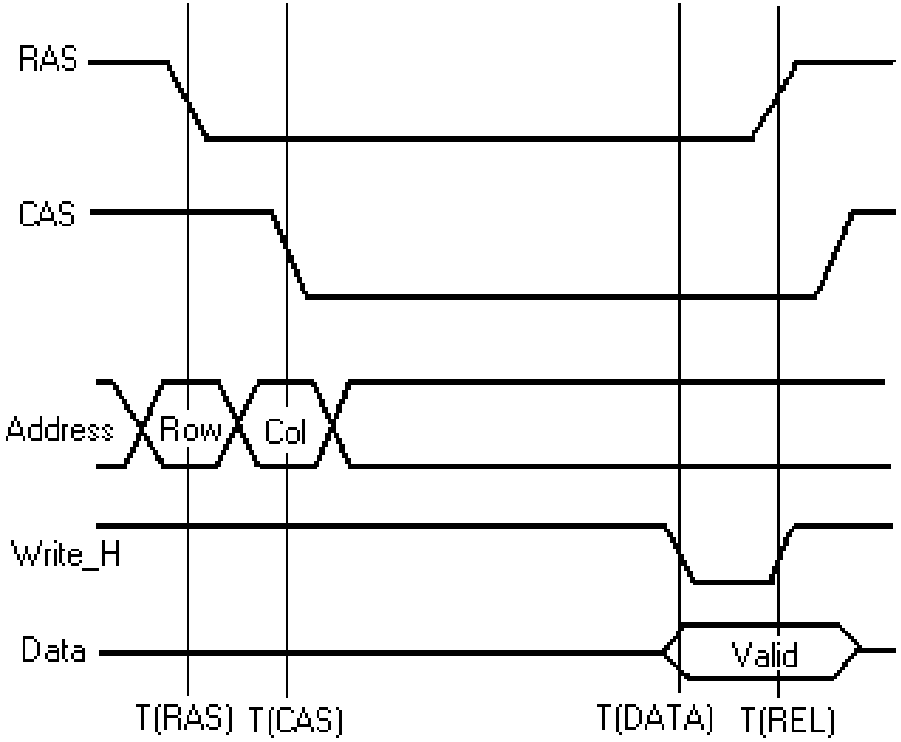
A frissítéskor például először egyetlen CAS oszlopcímet adunk ki, majd a RAS-al az összes sorcímet, hogy az összes cella frissítésre kerüljön. Frissítés történhet a kiolvasási ciklus végeztével is (még a következő beírás előtt).

Dinamikus RAM időd iagramjai:

Olvasási ciklus



Írási ciklus

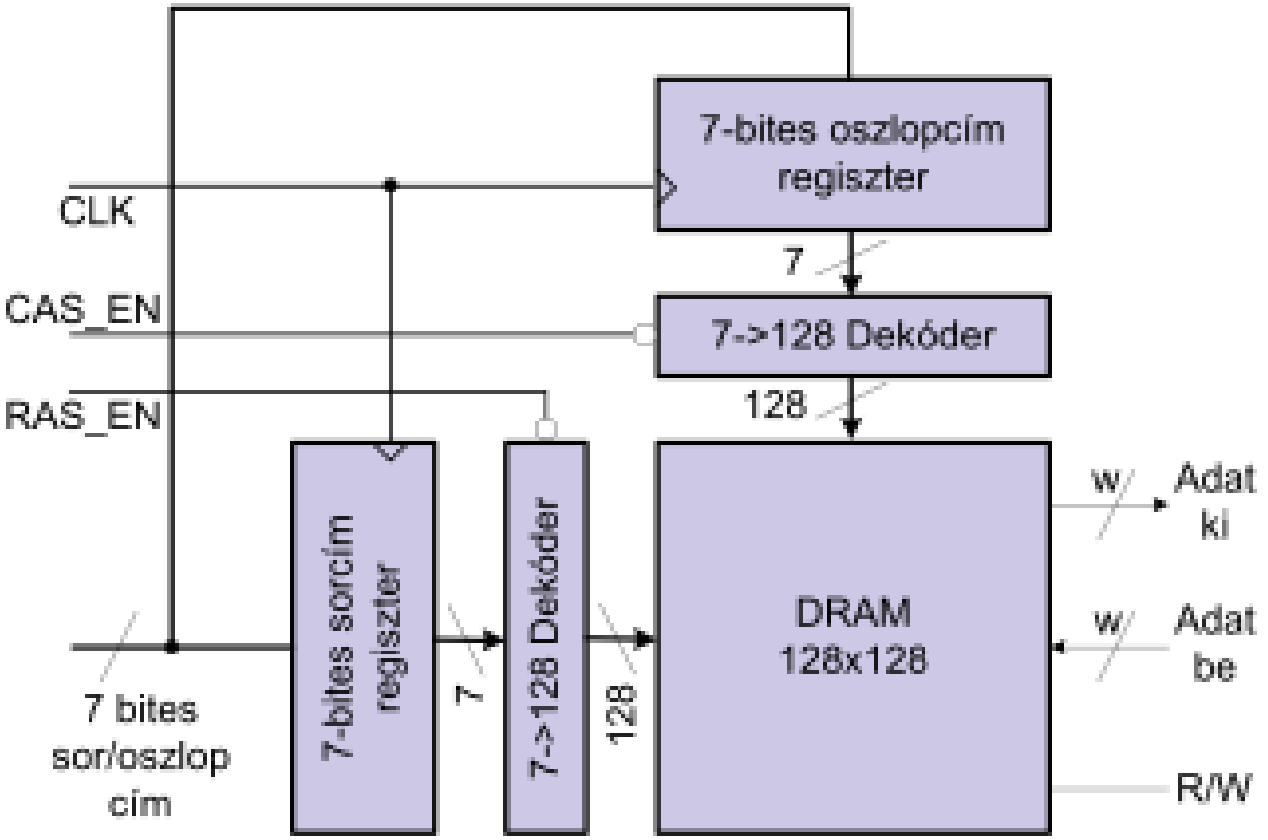


Címek: sor- és oszlop-azonosítók (RAS-CAS). A sorcím (Row), majd az oszlopcím (Col) megjelenik a címvonalon. Ezután válik elérhetővé a memória (setup time). A memória elérési idejétől (T_{acc}) függően kis idő múlva az adat érvényessé válik (Valid). A RAS felszabadul (T_{REL}) a kimeneten az adat visszatér (tri-state állapotba).

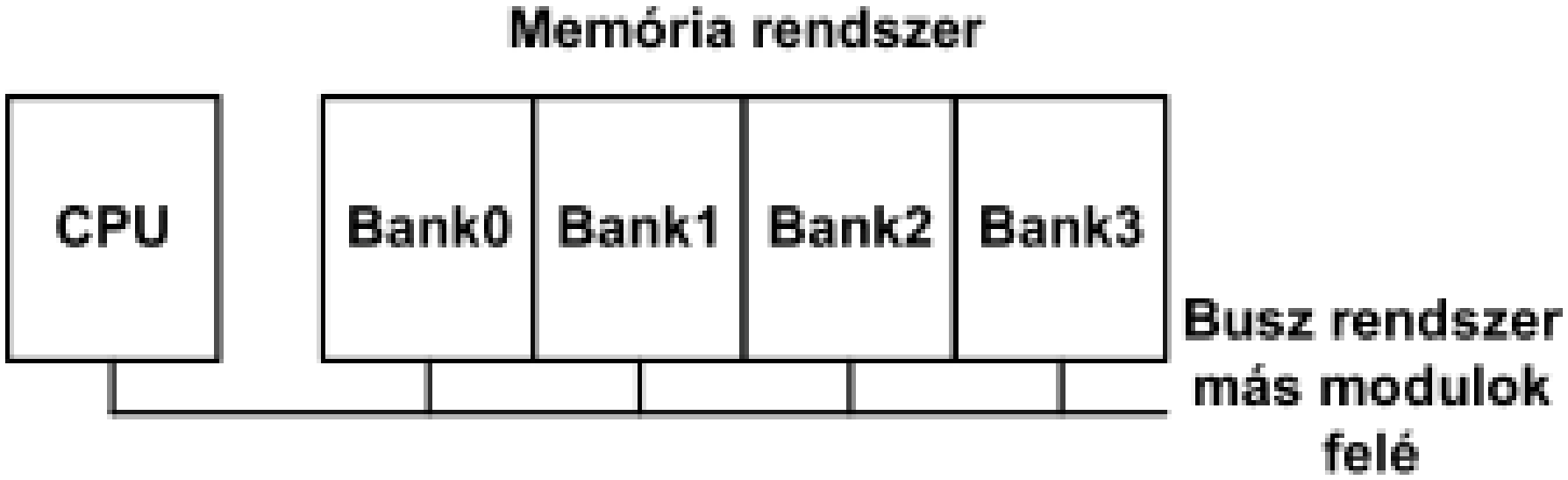
Példa: 128x128-as DRAM felépítése

Feladat: Mekkora lábszámot kell biztosítani az DRAM működéséhez?
Legyen w=16 bites

7+2*w+1(RAS_EN)+
1(CAS_EN)+1(R/W)+
1(Vdd)+1(Gnd) = 44
Tehát összesen min. 44 lábra (pin) van szükség!
(128*128 cella, 16bit/szó)



Memória „interleaving” technika

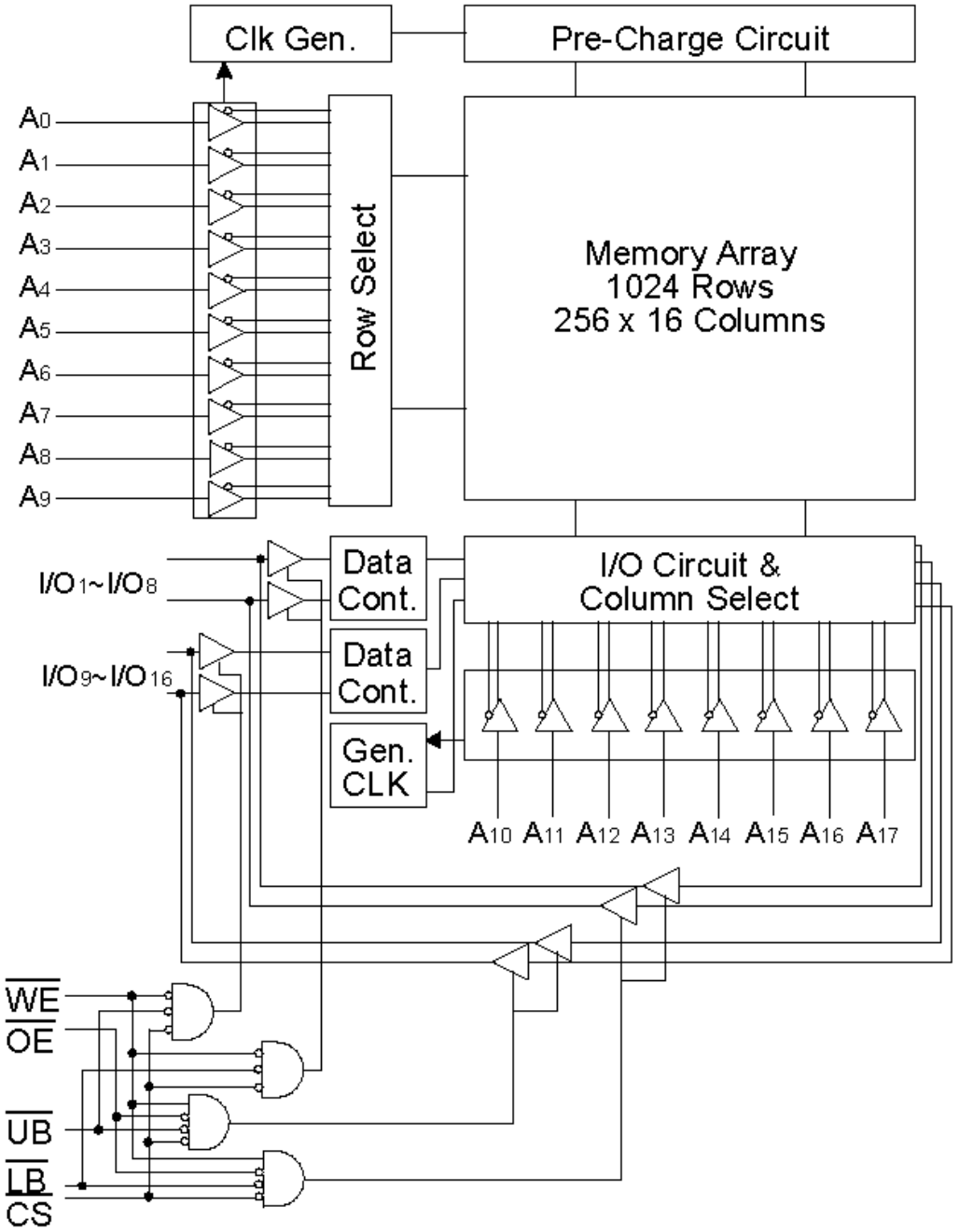


Cél: busz sávszélesség növelése

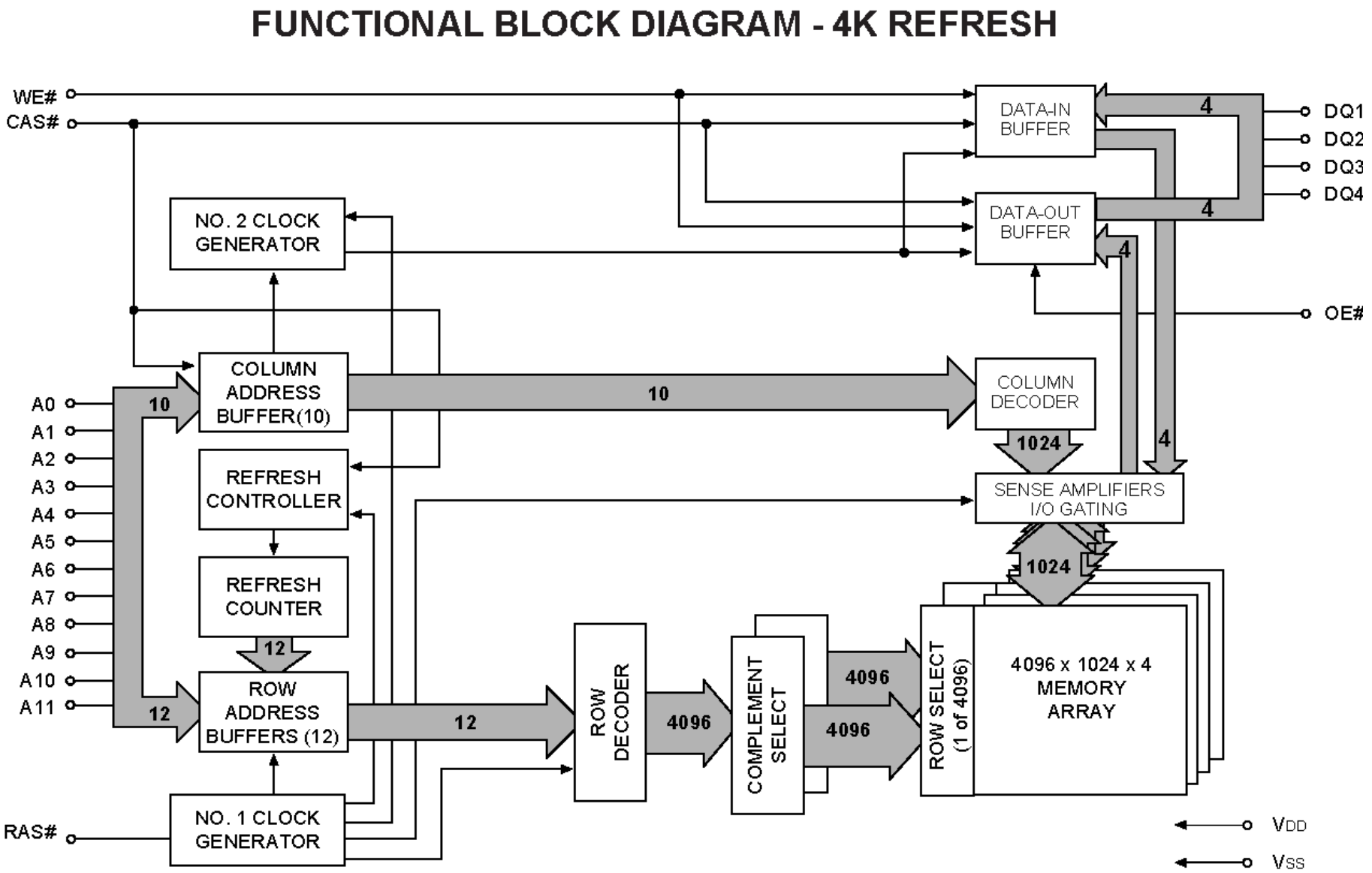
Pl. Nagyméretű memóriamodult 4-bankos szervezéssel kell kialakítani.

Interleaving: bankok közötti időbeli „átlapolódás”

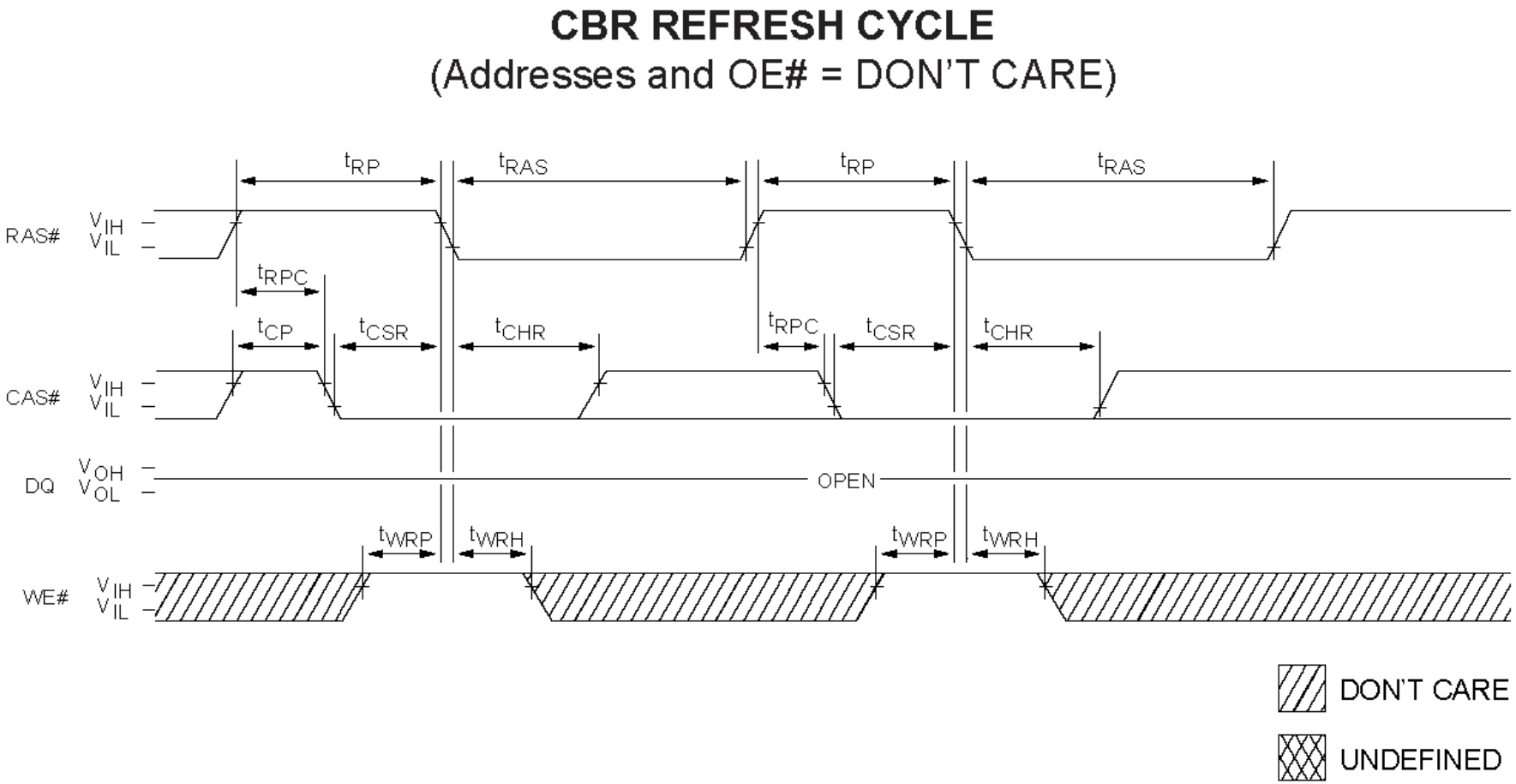
Aszinkron SRAM blokkdiagramja



Aszinkron DRAM blokk diagramja

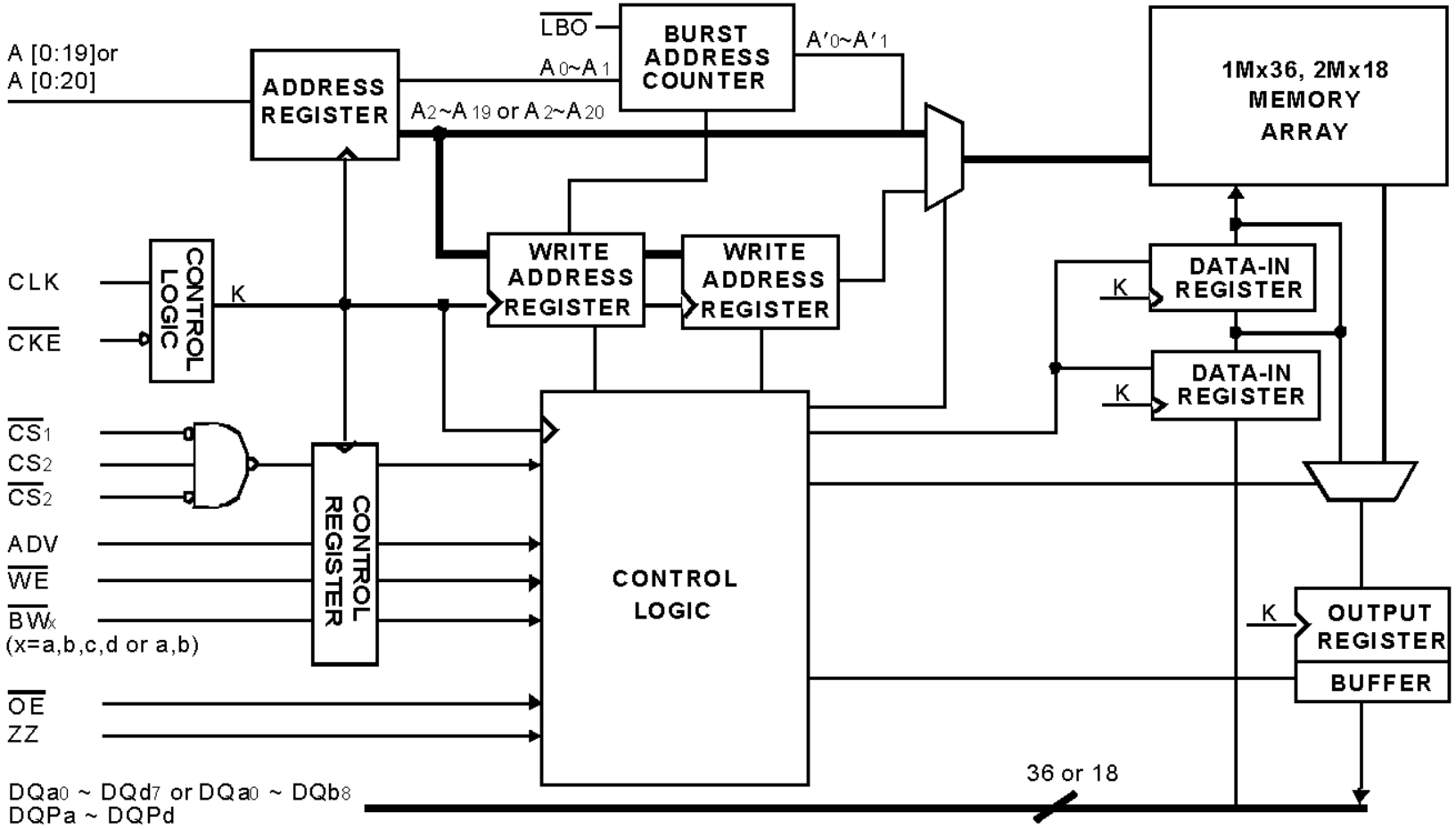


Aszinkron DRAM frissítési időd iagramja

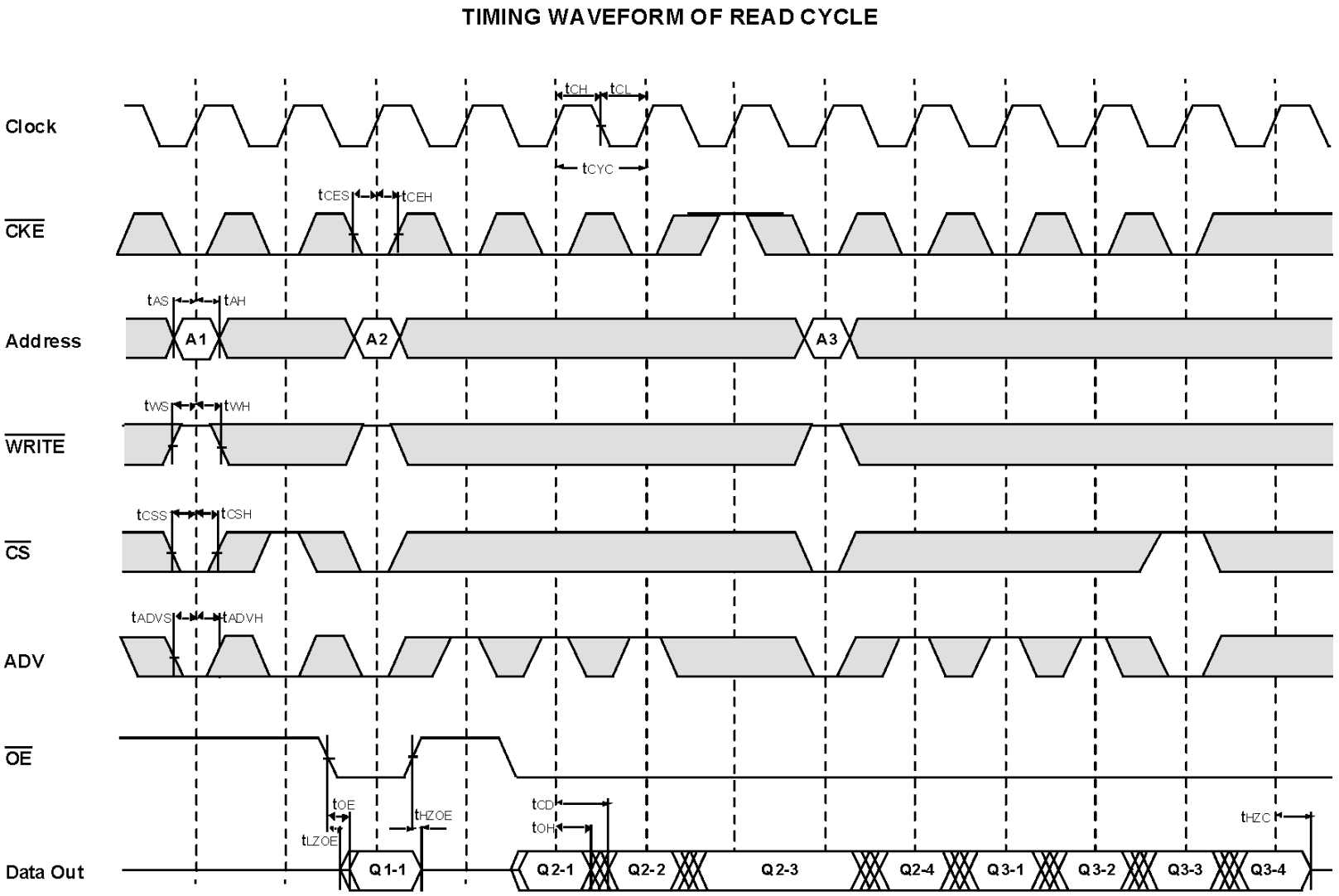


ZBT SSRAM blokk diagramja

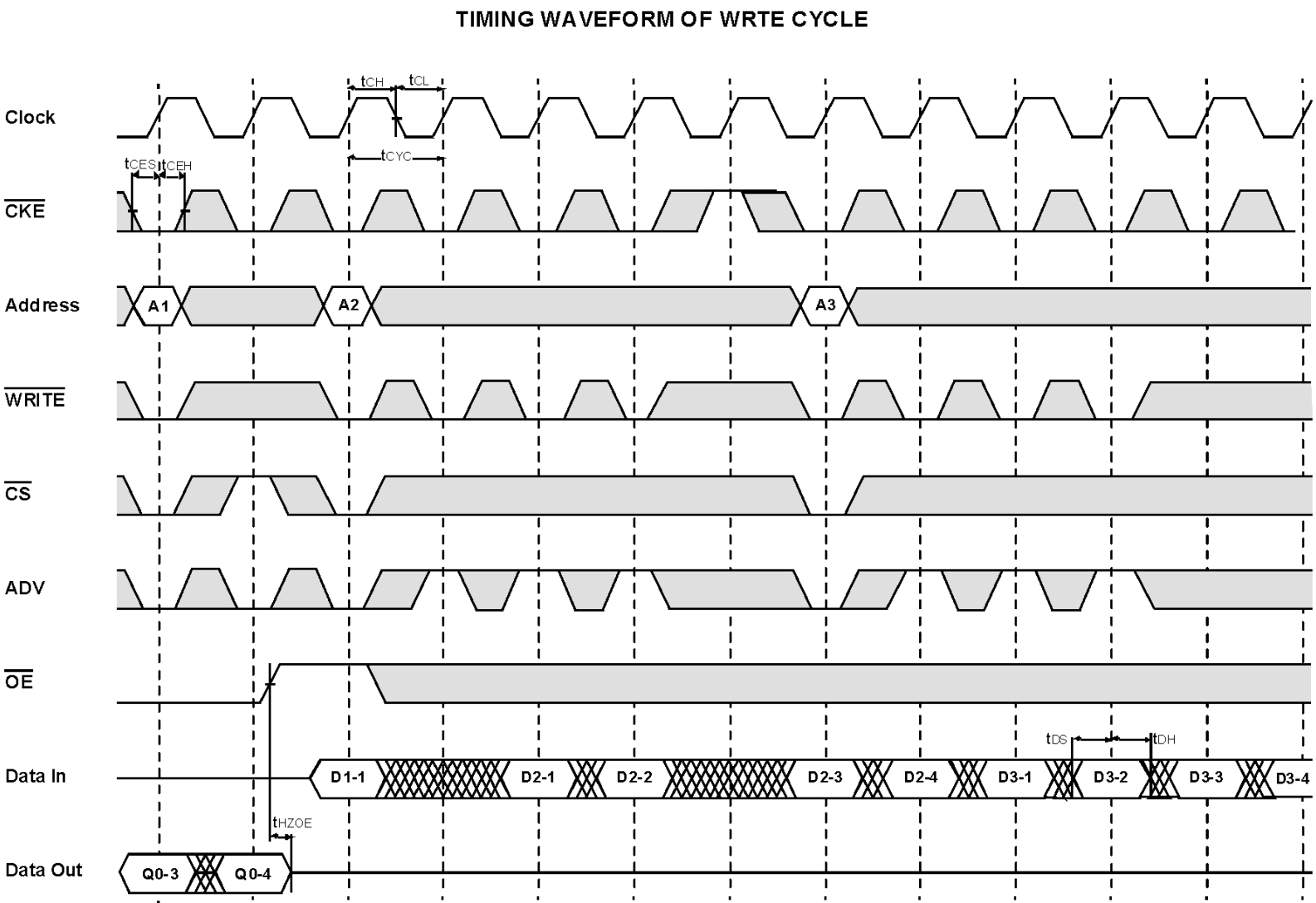
LOGIC BLOCK DIAGRAM



ZBT SSRAM olvasási ciklus



ZBT SSRAM írási ciklus



Újabb generációk

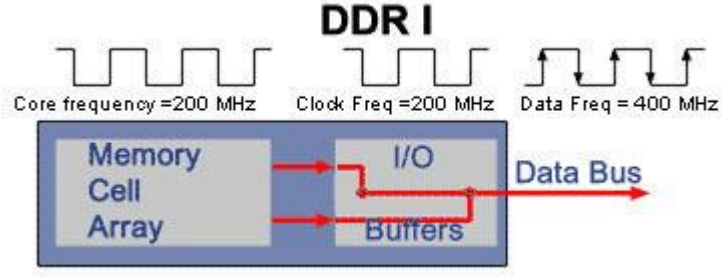
- Magasabb órajelű memóriákkal a rendszer nagyobb adatforgalmat képes lebonyolítani a processzor és a memóriák között
- processzor rendszerbuszának áteresztőképessége még 333 MHz-es FSB mellett is alacsonyabb (333 MHz esetén 10,6 GB/s)
- Mint a DDR3-as memóriák sávszélessége, persze kétcsatornás üzemmódban (12,8 GB/s DDR2/3-800 esetén)

- 2005 DDR3 bemutatása
- Az Intel P35-ös chipset megjelenésével és a 333 MHz-es FSB-vel rendelkező Core 2 processzorok bevezetésével a DDR3-as memóriák bevezetése (2007)
- 333 MHz-es Core 2-es processzorokhoz is a rendszerbusz négyszerezéséből adódóan "illik" 1333 MHz-es memóriát társítani (4 x 333).

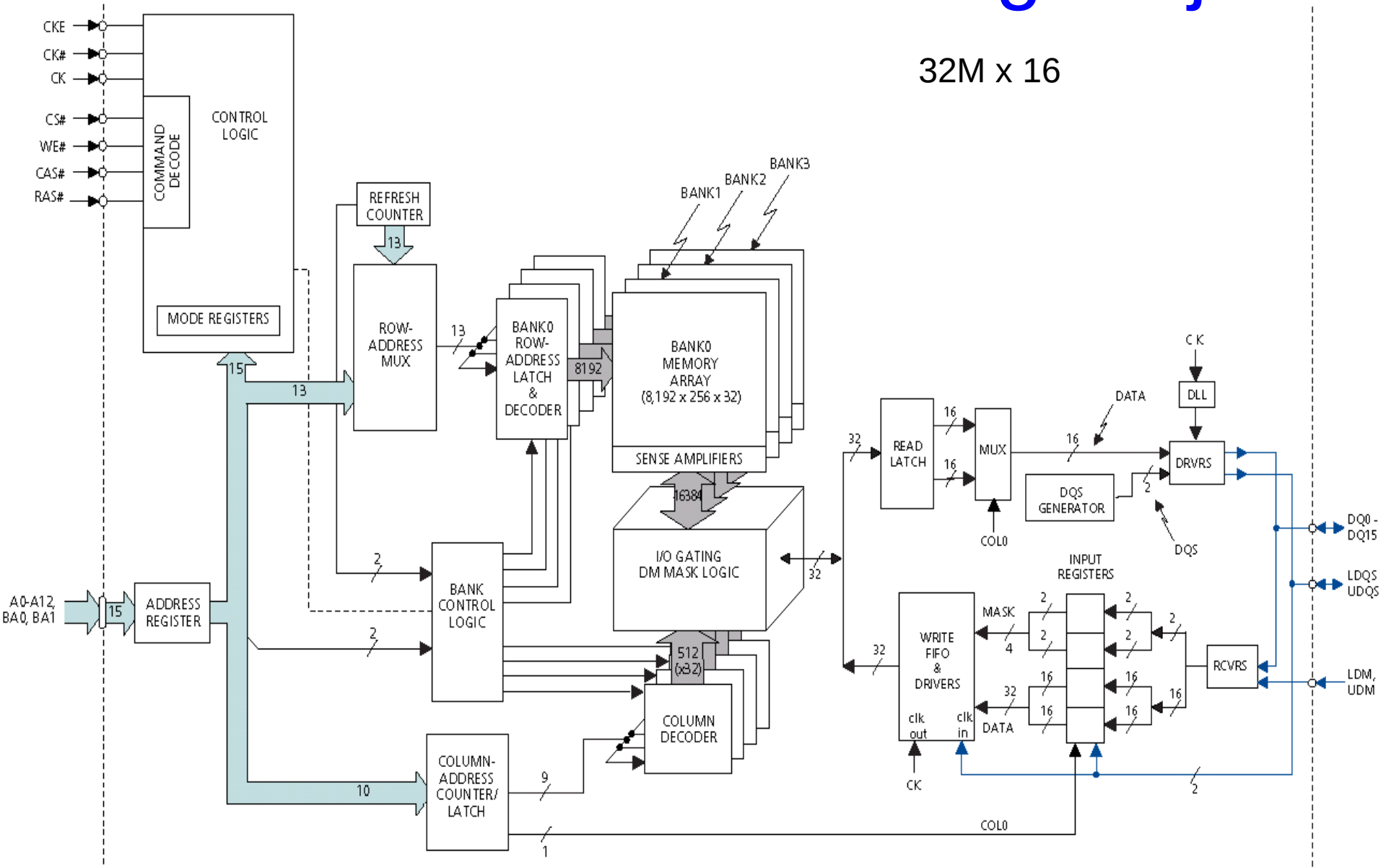
- a DDR2-800 kétcsatornás üzemmódban 12,8 GB/s-os sávszélességet biztosít,
- a DDR3 pedig a magasabb órajelek elérésével ezt növeli,
  - DDR3-1066 17 GB/s,
  - DDR3-1333 21,3 GB/s
- egy memória sebességét hasonló mértékben meghatározza az elérési idő is, a DDR3 esetében jóval magasabb, mint elődjénél.

DDR memóriák működési elve

- Double Data Rate, ami annyit jelent, hogy dupla adatmennyiség, legalábbis az SDR-DRAM-hoz képest.
- A DDR memóriák ezt úgy érik el, hogy az órajel mindkét (fel-le) ága hordoz információt, így az SDR-hez képest azonnal duplaakkora a sávszélesség.
- Ezért is duplázódik az effektív órajel, azaz 200 MHz-es DDR memória esetén DDR400-as modulokról beszélünk.
- A memóriák (nagyon leegyszerűsítve) két főbb komponensből állnak, a memóriacellákból, és az IO-pufferekből. A memóriacella órajele az első DDR esetében 100 és 200 MHz között váltakozott (DDR200 esetén 100, DDR400 esetén 200 MHz), tehát ahhoz, hogy a memóriamodul órajelenként (400 MHz) 1 bit átmozgatására legyen képes, a belső adatbuszon 2 bitnek kellett átutazni a memóriacellákból (200 MHz) az IO-pufferbe. **2n-prefetch**

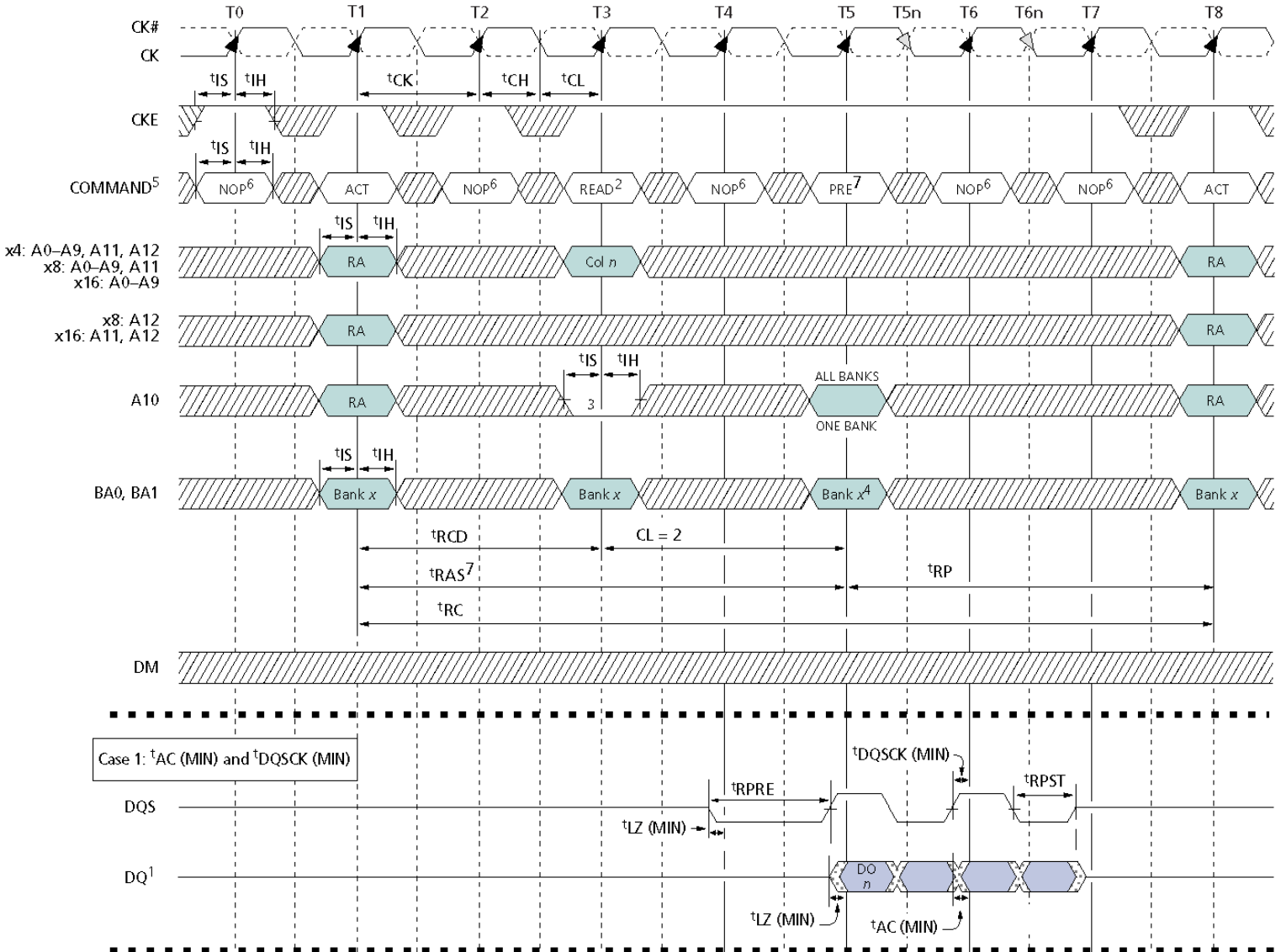


DDR SDRAM blokk diagramja



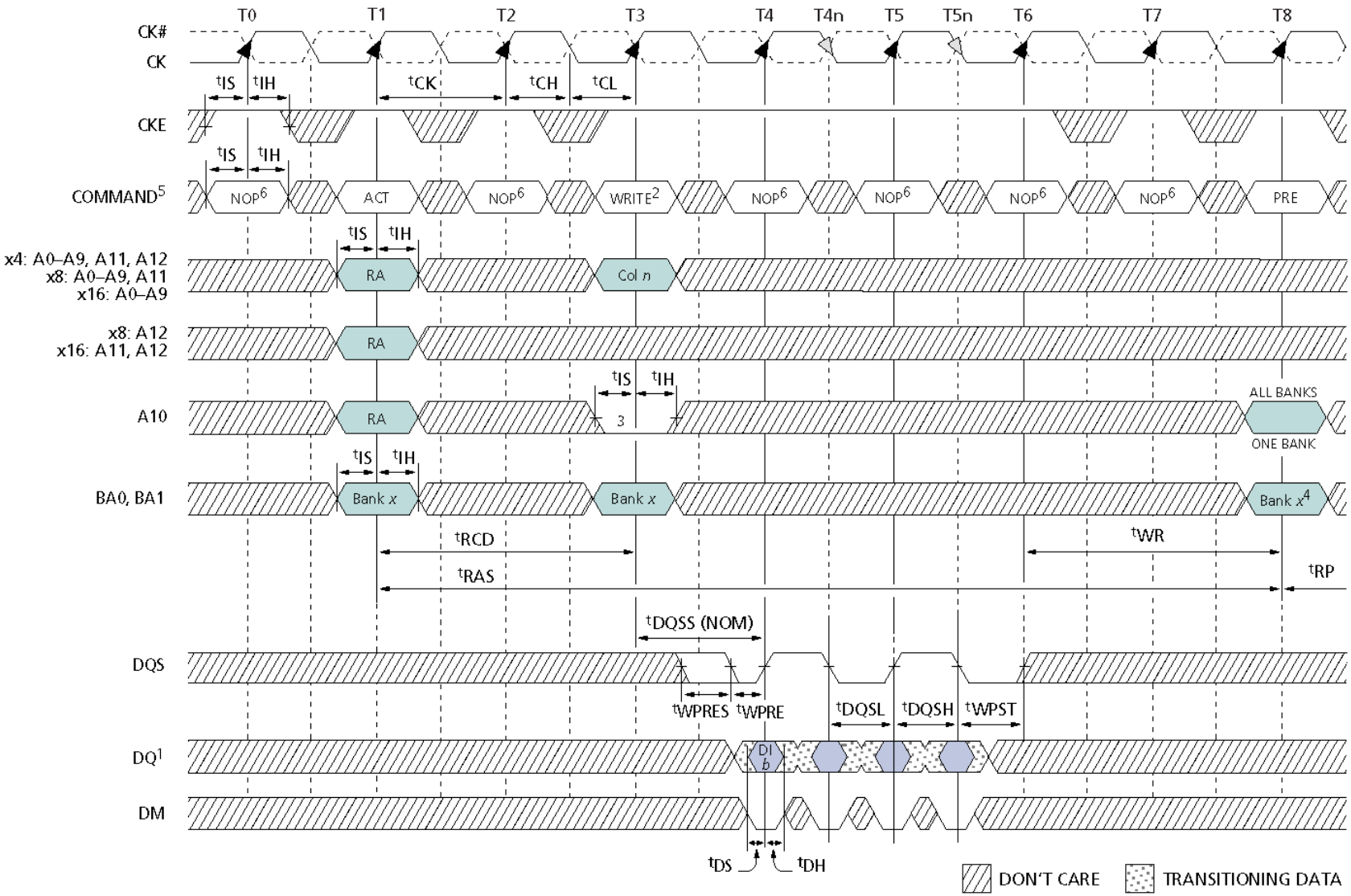
DDR SDRAM olvasási ciklus

Figure 49: Bank Read - Without Auto Precharge



DDR SDRAM írási ciklus

Figure 51: Bank Write - Without Auto Precharge



DDR-től DDR2-ig

- A DDR memóriák esetében a **JEDEC** 400 MHz-es effektív órajelnél húzta meg a határt
- Először a késleltetések csökkentése jött számításba (DDR400 esetén 3-3-3-8-ról 2-2-2-5-re),
- majd megjelentek a magasabb órajelű DDR modulok is (DDR500, DDR566, DDR600, stb), azonban ezek már nem feleltek meg a JEDEC által támasztott követelményeknek
- főleg azért nem, mert gyárilag specifikált feszültségük jóval meghaladta a DDR esetében szabványosított 2,5 V-ot (emiatt gyári hűtőbordákra volt szükség, illetve gyakoribbá vált a memóriák túlmelegedése, elhalálozása stb).
- Ekkor a gyártók kénytelenek voltak más módon növelni tovább a memóriák sebességét, így került képbe a **DDR2**.

JEDEC

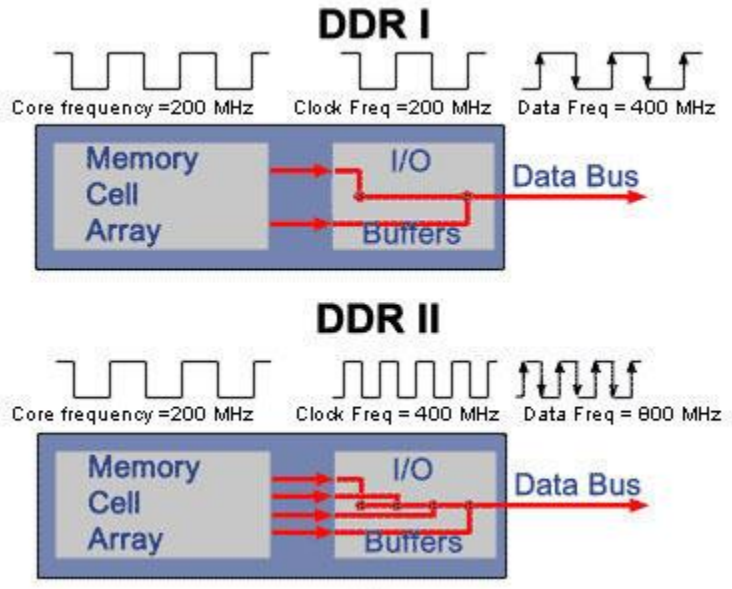
- The **JEDEC Solid State Technology Association**, formerly known as the Joint Electron Devices Engineering Council (**JEDEC**), is an independent semiconductor engineering trade organization and standardization body. Associated with the Electronic Industries Alliance (EIA), a trade association that represents all areas of the electronics industry in the United States, JEDEC has over 300 members, including some of the world's largest computer companies.

DDR-től DDR2-ig

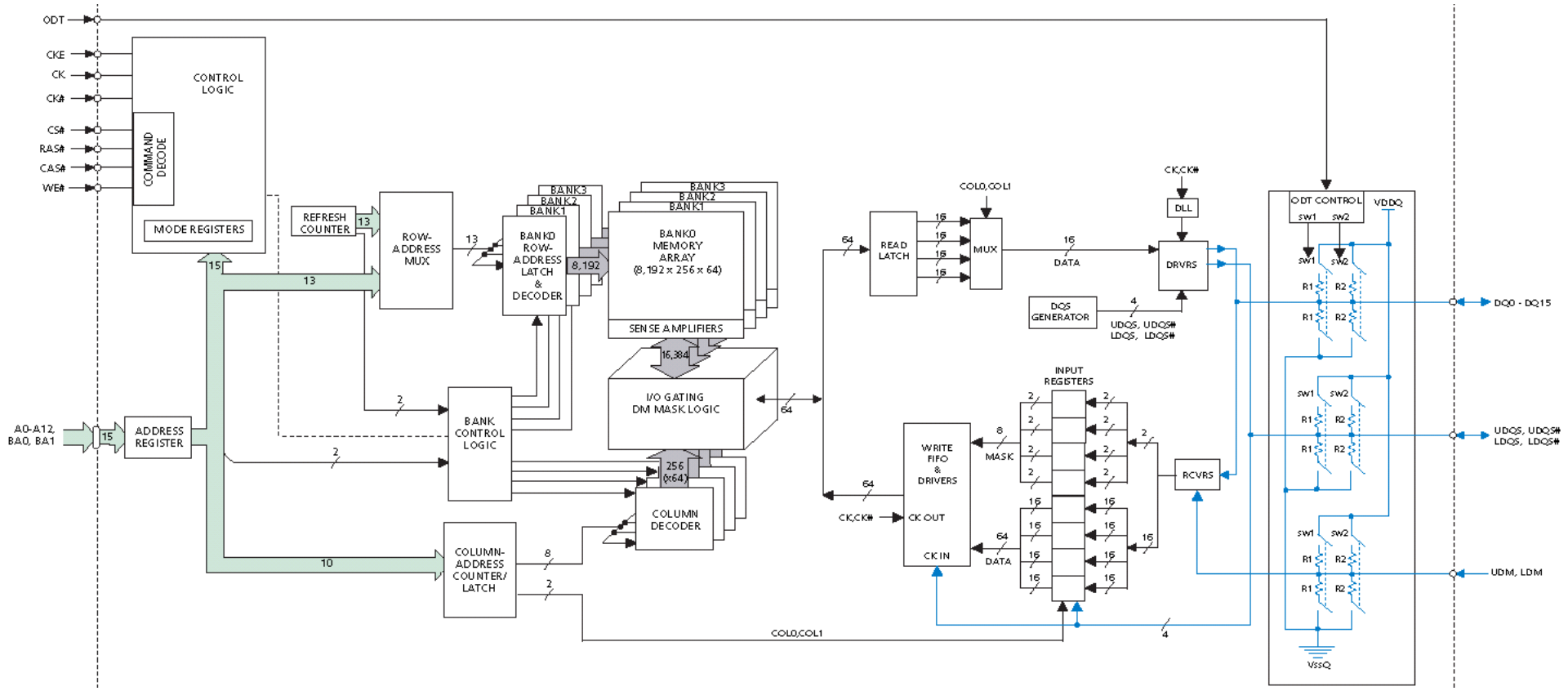
Memória típusa	Memóriacellák órajele	Adat prefetch-ek száma	IO-puffer órajele	Külső órajel	Elérhető sávszélesség
DDR200	100 MHz	2	100 MHz	200 MHz (DDR)	1600 MB/s
DDR266	133 MHz	2	133 MHz	266 MHz (DDR)	2128 MB/s
DDR333	166 MHz	2	166 MHz	333 MHz (DDR)	2656 MB/s
DDR400	200 MHz	2	200 MHz	400 MHz (DDR)	3200 MB/s
DDR2-400	100 MHz	4	200 MHz	400 MHz (DDR2)	3200 MB/s
DDR2-533	133 MHz	4	266 MHz	533 MHz (DDR2)	4256 MB/s
DDR2-667	166 MHz	4	333 MHz	667 MHz (DDR2)	5312 MB/s
DDR2-800	200 MHz	4	400 MHz	800 MHz (DDR2)	6400 MB/s

DDR2

- A DDR2-t úgy fejlesztették ki, hogy a leglogikusabb és legegyszerűbb módon nőjön tovább a külső órajel (azaz a modul órajele):
- megfelezték a memóriacellák órajelét,
- viszont a belső adatbusz órajelét megduplázták,
- vagyis elérték, hogy órajelenként már ne 2, hanem 4 bit vándorolhasson át a memóriacellákból az IO-pufferbe (**4n-prefetch**).
- Mivel továbbra is DDR-ről van szó, a sávszélesség megduplázódott (azonos órajelet feltételezve).
- Például a DDR2-800 esetében a memóriacellák órajele a DDR400-hoz hasonlóan 200 MHz, viszont az IO-puffer órajele immár 400 MHz (a DDR esetében 200 MHz), ám mivel DDR-ről van szó, vagyis ezt szorozni kell kettővel, a memória külső órajele már 800 MHz, azaz DDR2-800-ról beszélünk.
- Előnyei: Egyrészt mivel a memóriacellák órajele végeredményben nem nőtt a DDR-hez képest (elvégre megfelezték azt), **csökkenhetett a fogyasztás**, hiszen pl. egy DDR400-as modulhoz 200 MHz-es chipekre van szükség, míg egy DDR2-400-ashoz 100 MHz-es chipek kellenek, miközben a sávszélesség változatlan marad (3200 MB/s). Ugyanakkor a belső adatbusz sebességének megduplázásából adódóan viszont azonos belső órajelű chipek esetében a DDR2 **kétszer akkora sávszélesség**gel kecsegtet: a belső adatbusz 200 MHz a DDR400 és a DDR2-800 esetében is, előbbi 3200, utóbbi 6400 MB/s-os sávszélességet kínál.
- hátránya: Mivel azonos sávszélességet feltételezve a DDR2-es memóriacelláinak órajele feleakkora, mint elődjéé, az elérési idő megnőtt.

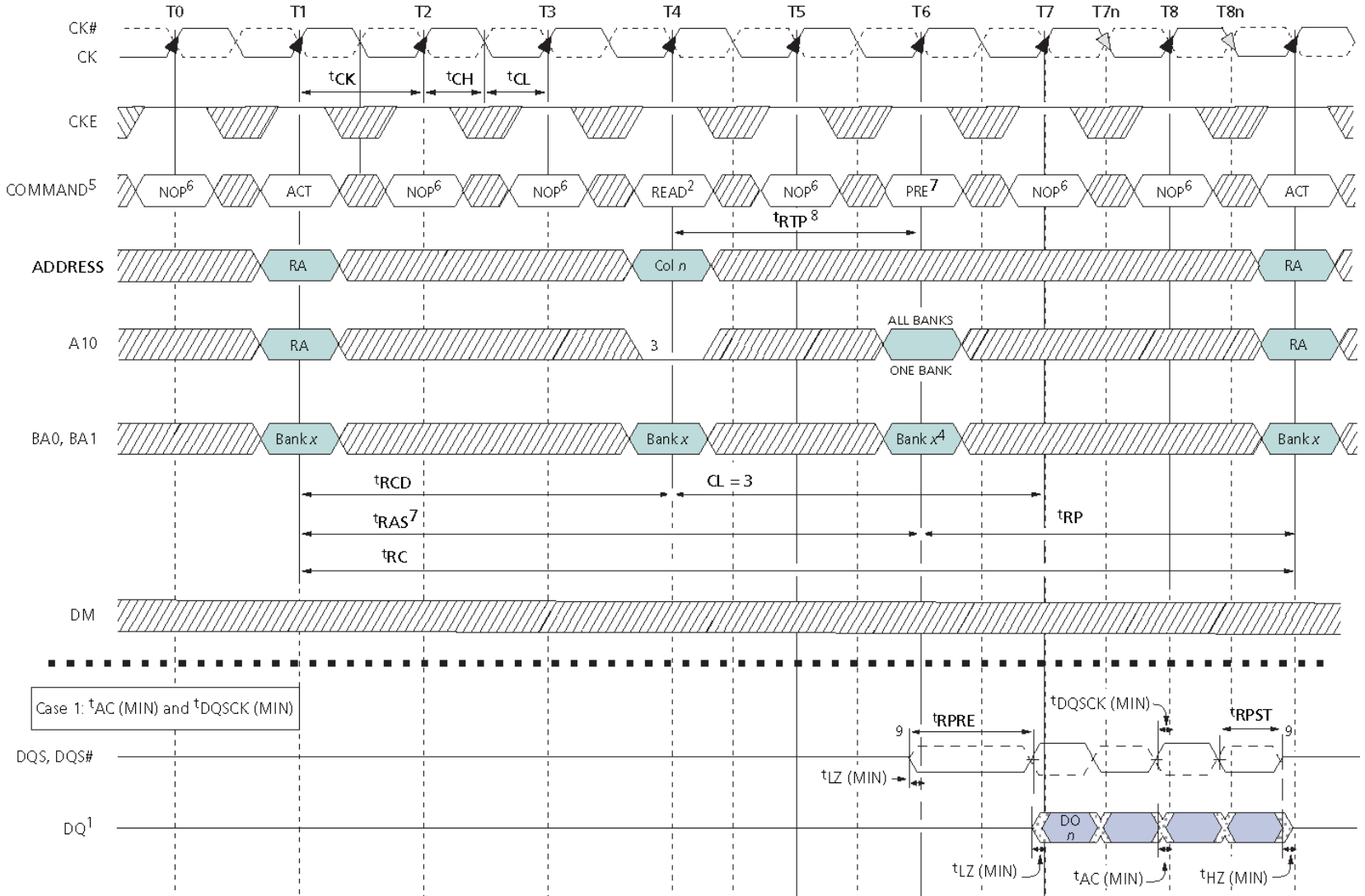


DDR-II SDRAM blokk diagram



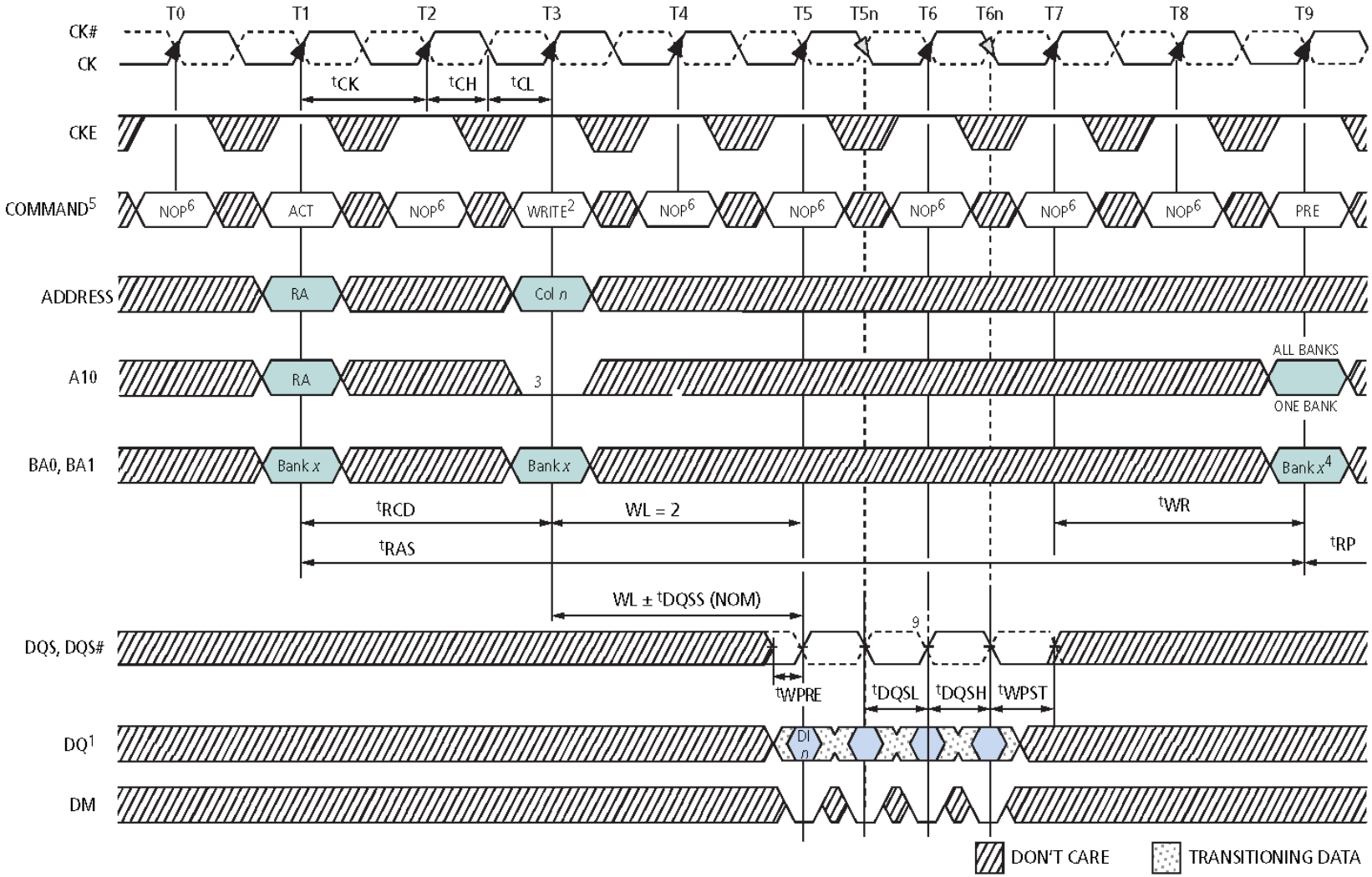
DDR-II SDRAM olvasási ciklus

Figure 28: Bank Read – Without Auto Precharge



DDR-II SDRAM írási ciklus

Figure 41: Bank Write–Without Auto Precharge



SDRAM-ok összehasonlító tábl.

		SD	DDR	DDR-II
I/O tápfeszültég	V_{DD}	3.3	2.5	1.8
CAS várakozási idő (latency)	CL	1-3	2-3	3-5
Órajel ciklus ideje (Clock cycle time)	t_{CK}	6ns	5ns	3ns
ACTIVE to PRECHARGE parancs	t_{RAS}	42ns	40ns	40ns
ACTIVE to READ or WRITE késleltetés	t_{RCD}	18ns	15ns	12ns
ACTIVE to ACTIVE/AUTO REFRESH parancs peridódus	t_{RC}	60ns	55ns	54ns
ACTIVE Bank(a) to ACTIVE Bank(b) parancs	t_{RRD}	12ns	10ns	7.5ns
Írás utáni “visszaállási idő”	t_{WR}	12ns	15ns	15ns
PRECHARGE parancs periódus	t_{RP}	18ns	15ns	12ns
REFRESH to REFRESH parancs intervallum	t_{REFC}	64ms	73us	105ns-70us
Átlagos, periódikus frissítés ideje	t_{REFI}	15us	7.8us	7.8us

DDR2-től DDR3-ig

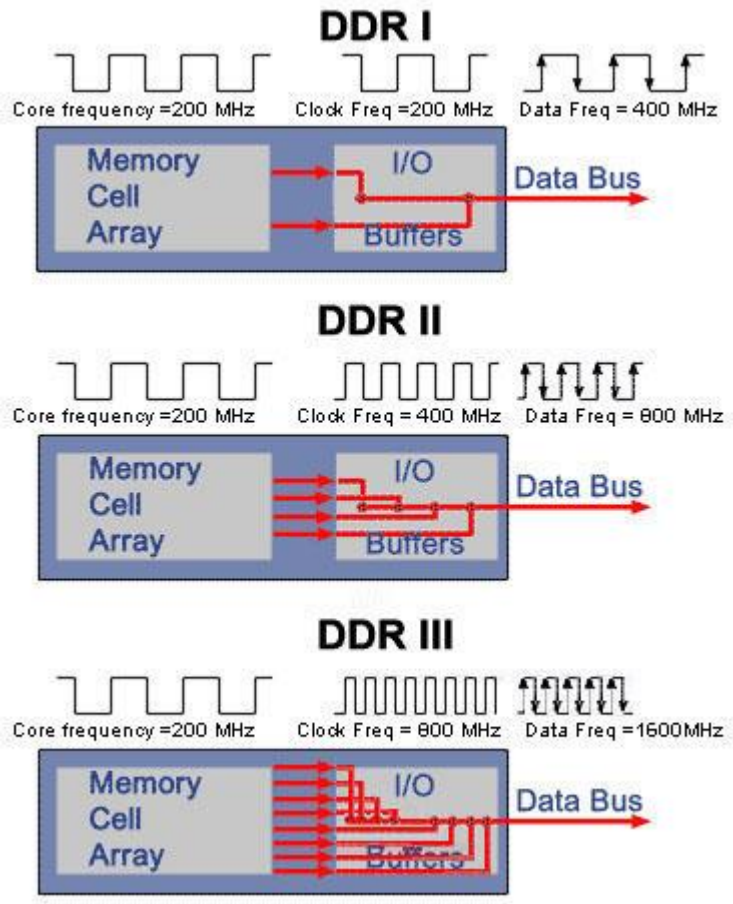
- Idővel azonban a DDR2-re is az a sors várt, ami anno a DDR-re: a gyártók a sebességet ezúttal is olyan módszerekkel tornázták feljebb, melyek már nem fértek bele a JEDEC követelményrendszerébe.
- Először csökkentek a késleltetési értékek, pl. a bevezetés után a DDR2-800-as modulok jórészt magas, 5-5-5-15-ös késleltetéssel rendelkeztek, ma pedig már találunk 4-3-3-10-es modulokat is egyes gyártók kínálatában (ugyanez lejátszódott az alacsonyabb órajelű DDR2-esekkel is).
- Az extrém alacsony késleltetések mellett a moduloknak magasabb gyári feszültségre volt szüksége, majd a technológia fejlődésével megjelentek a még magasabb órajelű DDR2-memóriák is (DDR2-1200-1300), melyek még extrémebb feszültséget igényelnek (2,3-2,4V az 1,8V helyett).
- Ezeknek külső hőelvezető hűtőbordára van szüksége a stabil működéshez, így elkerülhetetlenné és szükségessé vált a DDR memóriaszabvány harmadik generációjának kifejlesztése.

DDR2-től DDR3-ig

Memória típusa	Memóriacellák órajele	Adat prefetch-ek száma	IO-puffer órajele	Külső órajel	Elérhető sávszélesség
DDR2-400	100 MHz	4	200 MHz	400 MHz (DDR2)	3200 MB/s
DDR2-533	133 MHz	4	266 MHz	533 MHz (DDR2)	4256 MB/s
DDR2-667	166 MHz	4	333 MHz	667 MHz (DDR2)	5312 MB/s
DDR2-800	200 MHz	4	400 MHz	800 MHz (DDR2)	6400 MB/s
DDR3-800	100 MHz	8	400 MHz	800 MHz (DDR3)	6400 MB/s
DDR3-1066	133 MHz	8	533 MHz	1066 MHz (DDR3)	8512 MB/s
DDR3-1333	166 MHz	8	666 MHz	1333 MHz (DDR3)	10624 MB/s
DDR3-1600	200 MHz	8	800 MHz	1600 MHz (DDR3)	12800 MB/s

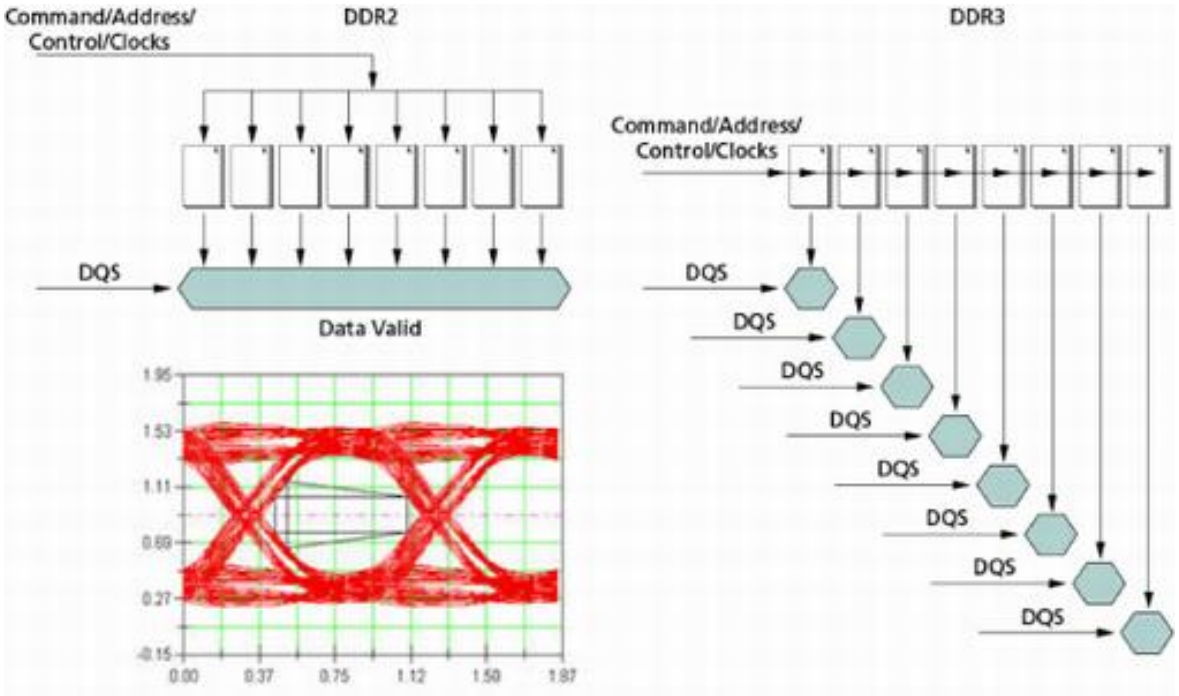
- Megfelezték a memóriacellák órajelét,
- viszont a belső adatbusz sebességét megduplázták, vagyis a DDR3 memóriacellái órajelenként 8 adatbitet továbbítanak az IO-pufferbe (**8n-prefetch**).
- Az új memóriaszabvány ott kezdi el az órajelhajhászást, ahol a DDR2 abbahagyta, azaz DDR3-800-as értéken, és a JEDEC DDR3-1600-ban határozta meg a maximumot, persze az általuk jegyzett specifikációk határain belül, aminek legfontosabb idevonatkozó része a gyári feszültség, ami a DDR3 esetében 1,5V.
- előnyök: azonos sávszélesség eléréséhez a DDR3-nak fele olyan gyors memóriacellákra van szüksége, vagyis **csökken a fogyasztás**.
- Ugyanakkor azonos memóriacella-órajelekkel **duplaakkora sávszélesség** érhető el, természetesen a DDR3 javára (DDR2-800 vs. DDR3-1600, mindkettő 200 MHz-es memóriacellákat használ).
- hátrány: a DDR3 **késleltetési értékei** a DDR2-höz képest tovább **romlottak**, ami már eleve komoly hátrányban volt a DDR-hez képest, így az **elérési idő nőtt**.

DDR3



- A 20%-kal alacsonyabb működési feszültség mellett további újításként lehet felfogni a **memóriacellák sűrűségének megnövekedését**, aminek egyenes következménye a **nagyobb memóriamodulok** megjelenése (akár 8 GB-os is, ha a gyártók is úgy akarják),
- újítás még a jeltovábbítási algoritmus megváltozását. A DDR3 modulok a DDR2-vel ellentétben úgynevezett "fly-by" topológiát alkalmaznak; ennek lényege, hogy a parancsokat/címzési műveleteket a memóriachipek egymás után kapják meg, és nem egyazon időben, mint a DDR2 esetén.
- Ezzel a módszerrel **erősödik a jelátvitel**, amire lényegében a magas órajelek elérése miatt van szükség, ám **nő a késleltetés**.
- Érdemes még megemlíteni, hogy a DDR3 modulokon megjelenhetnek a **hőmérő szenzorok** is (opcionális)
- Azt már mondanunk sem kell, hogy bár a DDR3 modulok is 240 érintkezővel rendelkeznek, a DDR2-től fizikailag eltérő kialakításuk miatt nem mennek bele a DDR2 foglalatba

Fly-by topológia



A sávszélesség kiszámítása

- DDR1 = memória eredeti órajele x 2 x 8 vagy IO-puffer órajele x 2 x 8 = sávszélesség (MB/s)
- DDR2 = memória eredeti órajele x 4 x 8 vagy IO-puffer órajele x 2 x 8 = sávszélesség (MB/s)
- DDR3 = memória eredeti órajele x 8 x 8 vagy IO-puffer órajele x 2 x 8 = sávszélesség (MB/s)

elérési időt

- (1000 / memória valódi órajele (100-200 MHz) x CAS Latency) / 1 (DDR) vagy 2 (DDR2) vagy 4 (DDR3) = elérési idő (ns)

DDR2-től DDR3-ig: sávszélesség vs. Elérési idő

Memória típusa	Memória órajele	Elérhető max. sávszélesség	Időzítések	Elérési idő
DDR2-800	200 MHz	6400 MB/s	4-4-4-12	10 ns
DDR2-800	200 MHz	6400 MB/s	5-5-5-15	12,5 ns
DDR2-1066	266 MHz	8500 MB/s	5-5-5-15	9,4 ns
DDR3-800	100 MHz	6400 MB/s	**5-5-5-15**	12,5 ns
DDR3-800	100 MHz	6400 MB/s	7-7-7-20	17,5 ns
DDR3-1066	133 MHz	8500 MB/s	5-5-5-15	9,4 ns
DDR3-1066	133 MHz	8500 MB/s	**7-7-7-20**	13,1 ns
DDR3-1066	133 MHz	8500 MB/s	8-8-8-22	15 ns
DDR3-1333	166 MHz	10666 MB/s	5-7-5-16	7,5 ns
DDR3-1333	166 MHz	10666 MB/s	7-7-7-20	10,5 ns
DDR3-1333	166 MHz	10666 MB/s	**9-9-9-22**	13,5 ns

Pannon Egyetem
Képfeldolgozás és Neuroszámítógépek Tanszék



Digitális Rendszerek és Számítógép Architektúrák

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

Előadó: Vörösházi Zsolt
voroshazi@vision.vein.hu

Jegyzetek, segédanyagok:

- Könyvfejezetek:
  - http://www.knt.vein.hu
    ⇒ Oktatás ⇒ Tantárgyak ⇒ Digitális Rendszerek és Számítógép Architektúrák
    (chapter06.pdf + további részek, amik a könyvben nem szerepelnek)
- Fóliák, óravázlatok .ppt (.pdf)
- Feltöltésük folyamatosan

I/O műveletek

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

I/O egységek

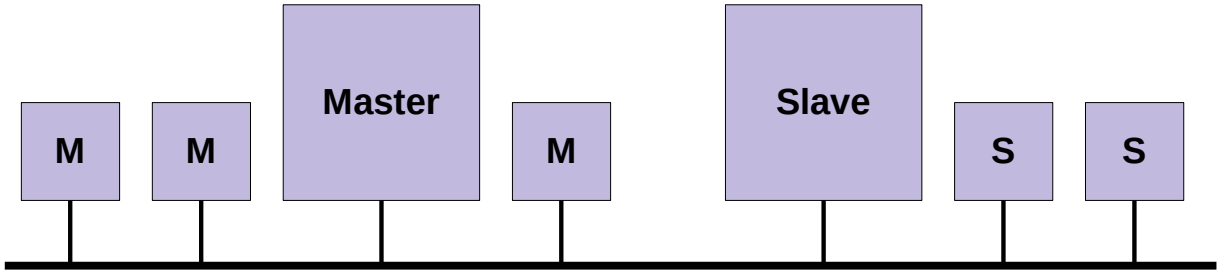
- A számítógép a külvilággal, *perifériákkal* az I/O egységeken keresztül tartja a kapcsolatot. Az információ továbbítását az egységek között buszok végzik, amelyek interfészekkel kapcsolódnak egymáshoz.
- **Interface**: azon szabályok összessége, amelyek mind a fizikai megjelenést, kapcsolatot, mind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

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 multimaster-rendszer több kezdeményezővel) is egy rendszerben.
- **Két alapvető protokoll különböztethető meg:**
  - *1.) Aszinkron* kommunikációs (pl. SCSI busz), és
  - *2.) Szinkron* kommunikációs (pl PCI busz) protokoll.

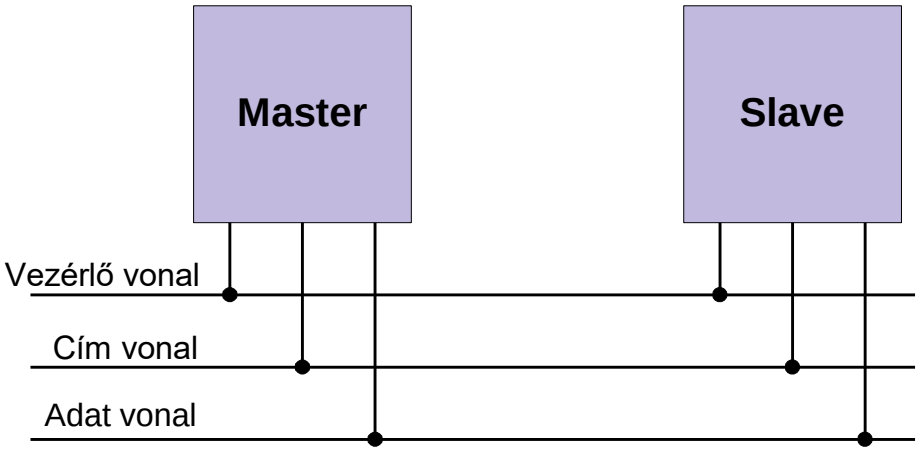
Busz rendszer Master-Slave moduljainak szervezése

- Modul-szervezés



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

- Busz-szervezés: Master - Slave



1.) Aszinkron busz protokollok

Aszinkron busz protokollok

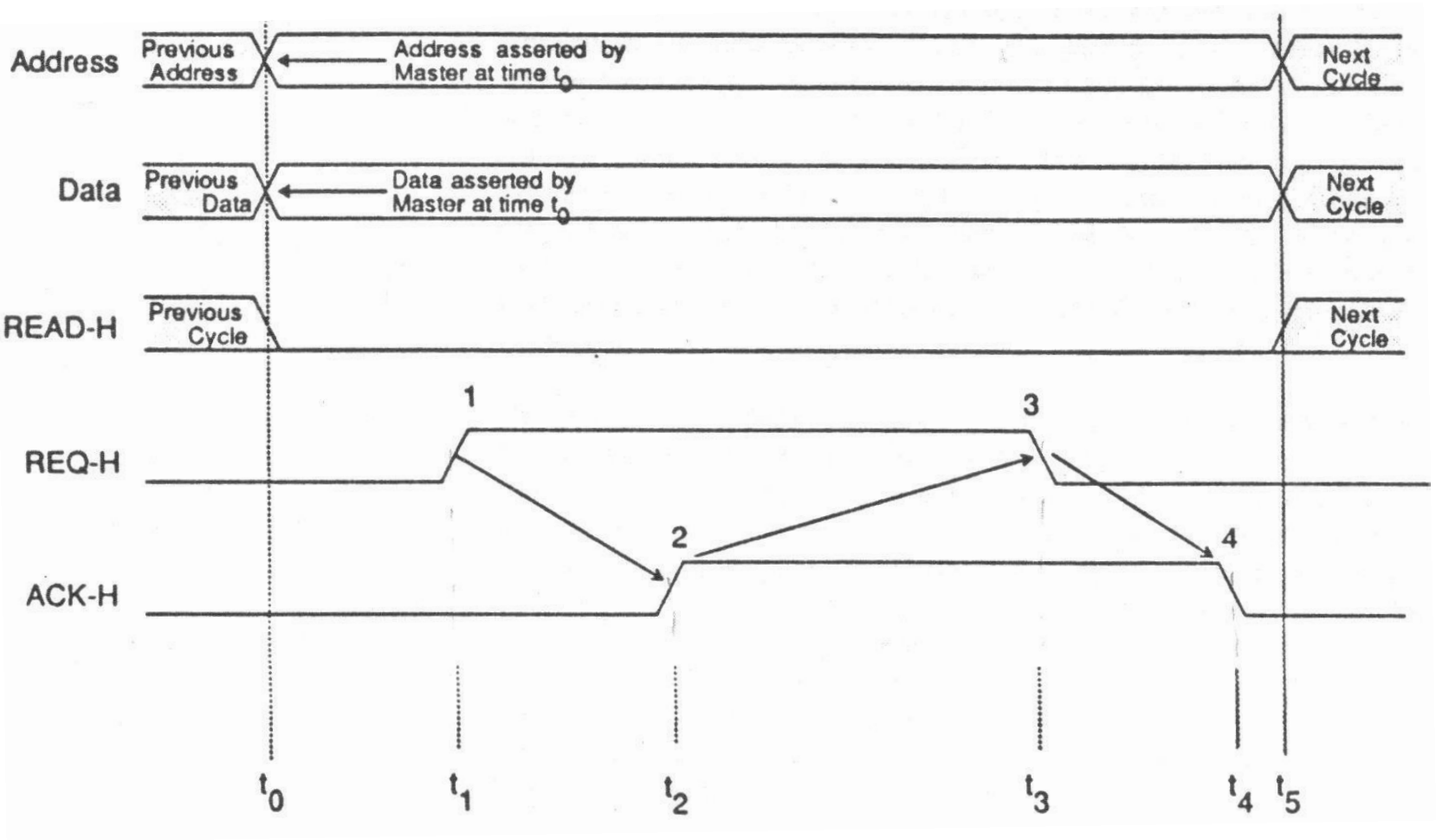
- Ebben az esetben a Master-Slave modulok **nem!** használnak közös órajelet (CLK-t): ha az egyik egység végez, elindít egy másik tranzakciót. A Master, mint (commander) kezdeményező, aktiválja a megfelelő vezetékeket. A Slave (responder) válaszol. A Slave címének azonosítására egy külön egység szolgál.
- **Vezérlőjelekkel** zajlik a kommunikáció: *írási* / *olvasási* tranzakciókat különböztetünk meg. Ilyen vezérlőjelek lehetnek a
  - READ, (ha pl. READ-H = Master olvas a Slave-ről, ha READ-L= Master ír a Slave-re, vagyis a Slave olvas.),
  - REQUEST (REQ)
  - ACKNOWLEDGE (ACK)
- *Előnye*: egyszerűen felépíthető, nem kell (közös) órajel, gyors átvitelt biztosít (modulok sebességétől függően)
- *Hátránya*: beépített késleltetések (skew-propagation time), ill. 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 *címvonal*ra 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”

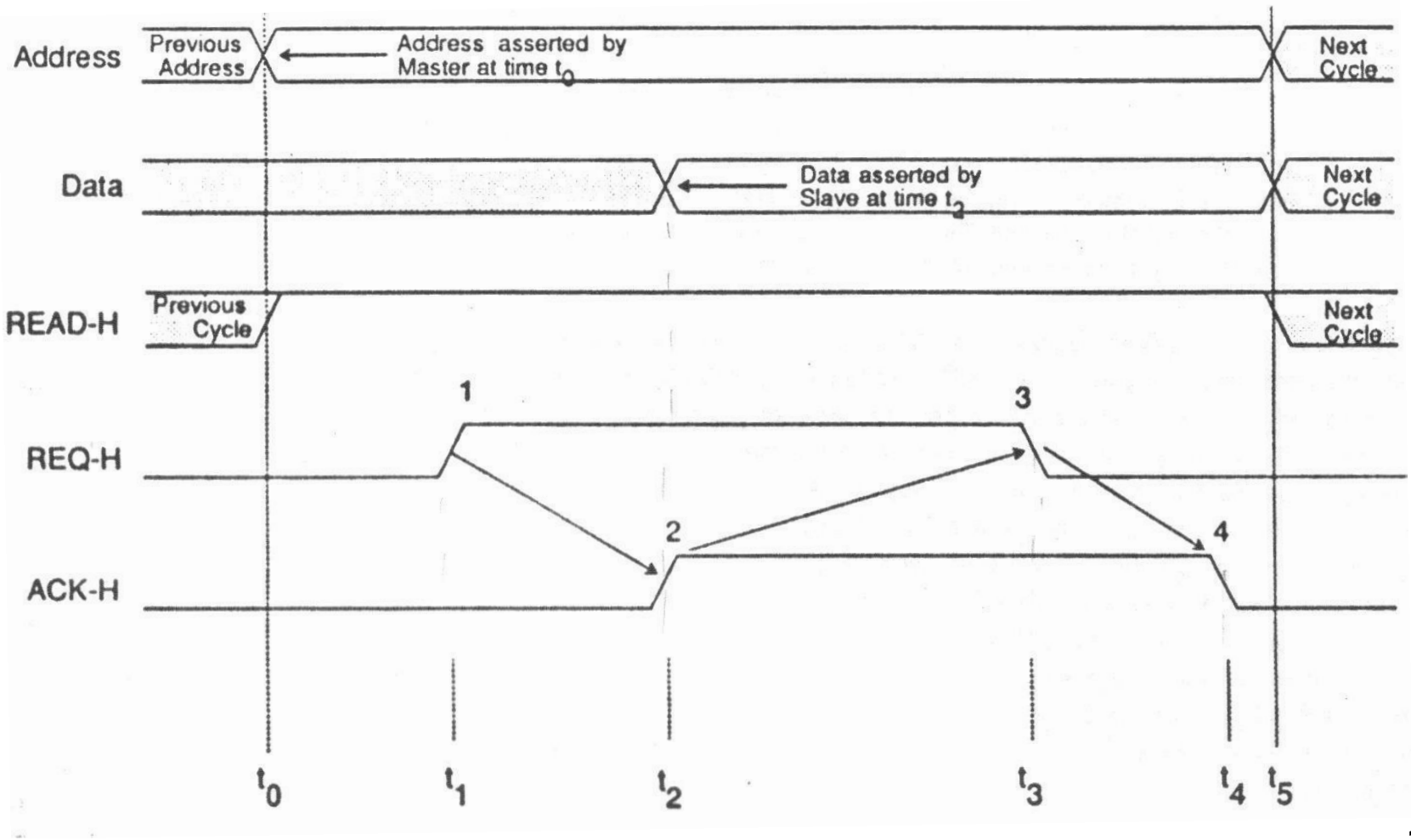


Írási (Write) ciklus lépései:

- t_o-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_o 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)

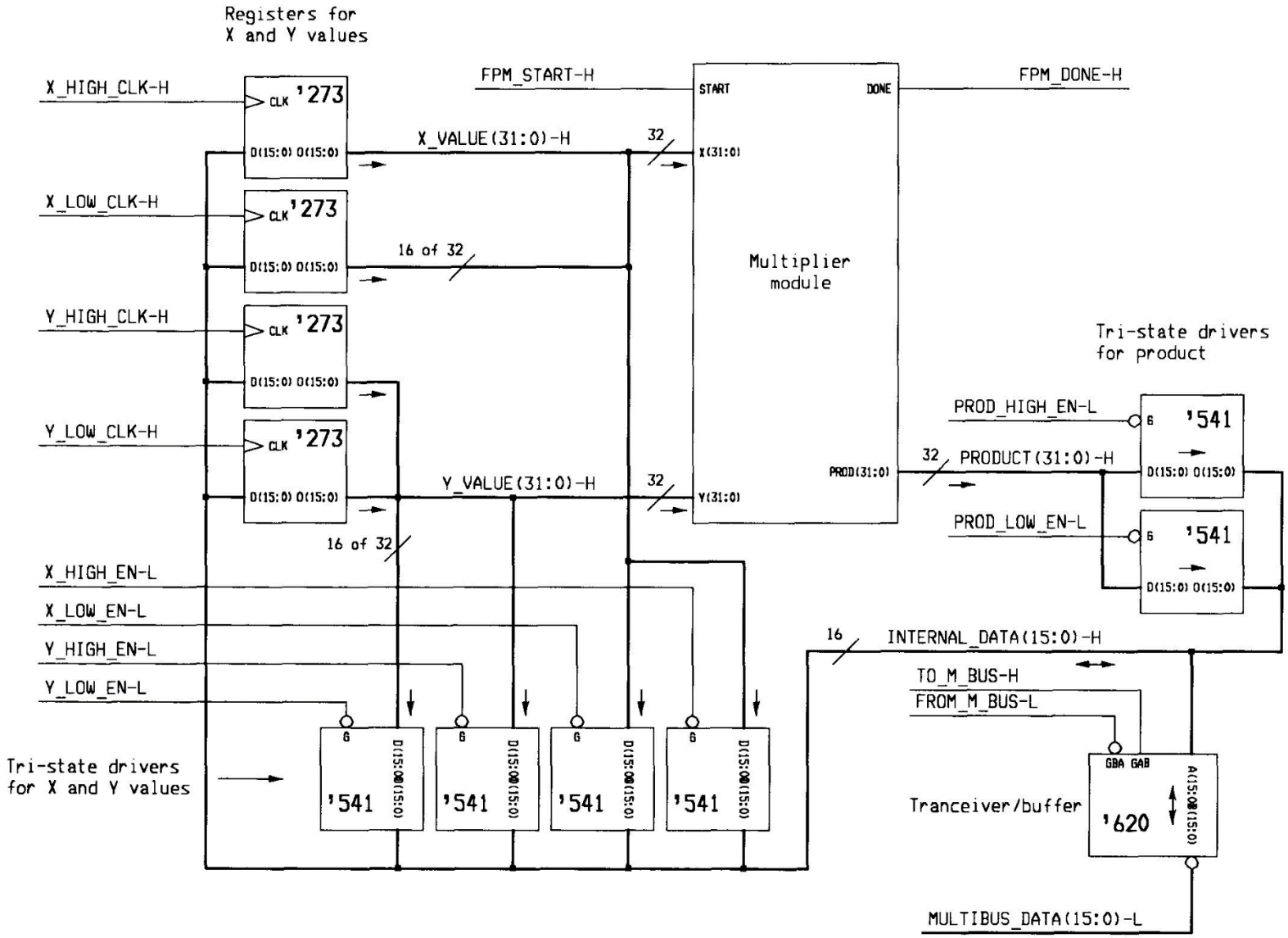
Példa 1: Multibus aszinkron protokoll

- Aszinkron handshaking-en alapul.
- Jelek alacsony aktív (negatív logika T=L / F=H) szintre definiáltak (adat/cím/vezérlő vonal)
- 20 címvonal (1 Mbyte címezhető), és 16 adatvonal
- Master beállítja az érvényes címet, majd vár (skew-time) kb 50ns-ig. → REQ.
- A Slave egység az XACK_L vonalon nyugtáz
- Nem egy READ_H vonal van, hanem két-két irányfüggő tranzakciós vonal:
  - MRDC_L: MEM olvasása
  - MWRC_L: MEM írása
  - IORC_L: I/O periféria olvasása
  - IOWC_L: I/O periféria írása

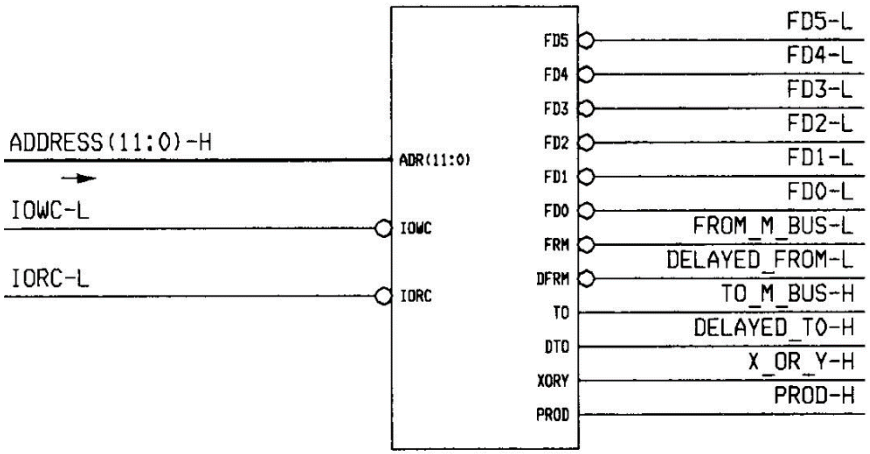
Multibusz aszinkron rendszer interfész

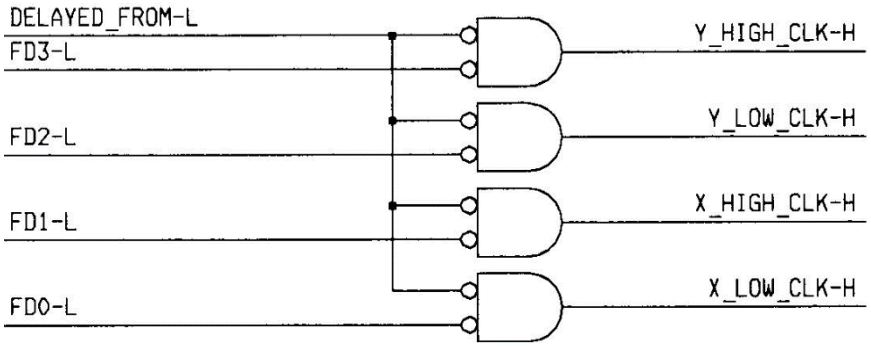
- Lebegőpontos szorzó: két 32-bites adatszó (X, Y bemeneti regiszterekkel)
- Multibus I/O címtartományhoz van a szorzó illesztve
- T(ALU): végrehajtási idő
- DONE jel: a szorzás eredménye a kimeneten
- Multibus protokoll: 16-bites bus Master 65535/16=4096 különböző tartományt érhet el, amelyből a szorzó a következőket foglalja le:
  - DF0 … DF5 IOWC_L (alsó 16 bitje a szónak)
  - DF0 … DF5 IORC_L (felső 16 bitje a szónak)
- ‘273: él-vezérelt regiszterek (X,Y adatait tárolja)
- ‘541: tri-state driver: szorzat - adatbusz illesztése
- ‘620: kétirányú transciever áramkör: belső adatbusz – Multibus illesztése

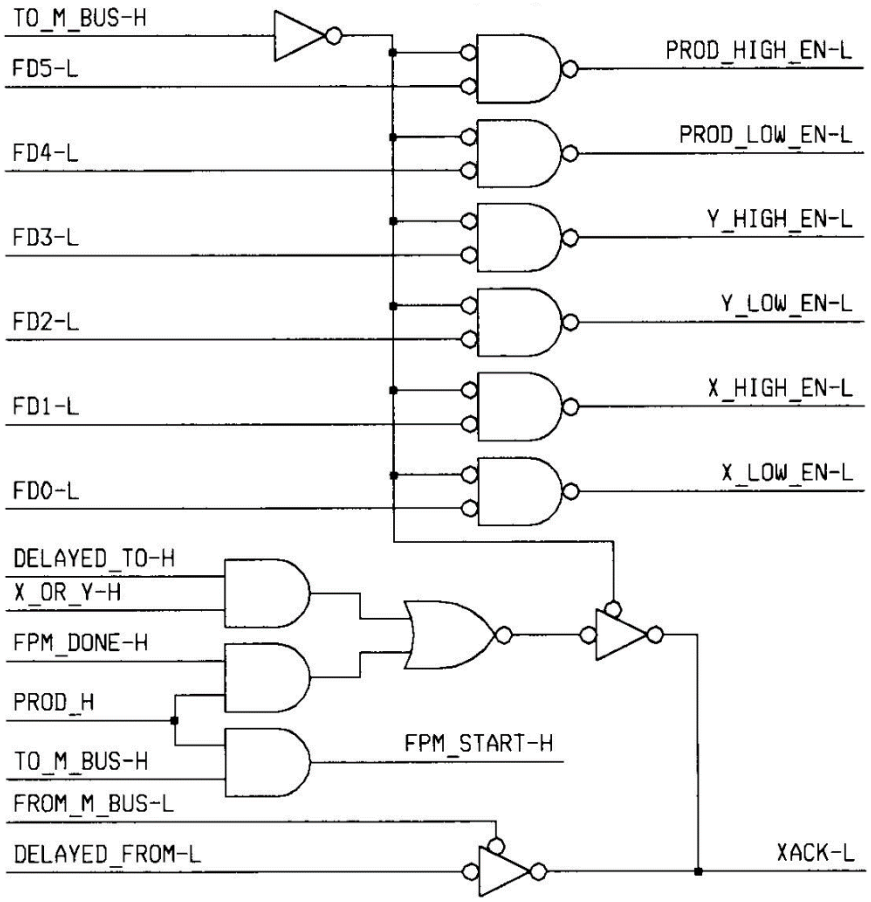
Példa: Multibus interfész adatútja



Példa: Multibus interfész vezérlő jelei



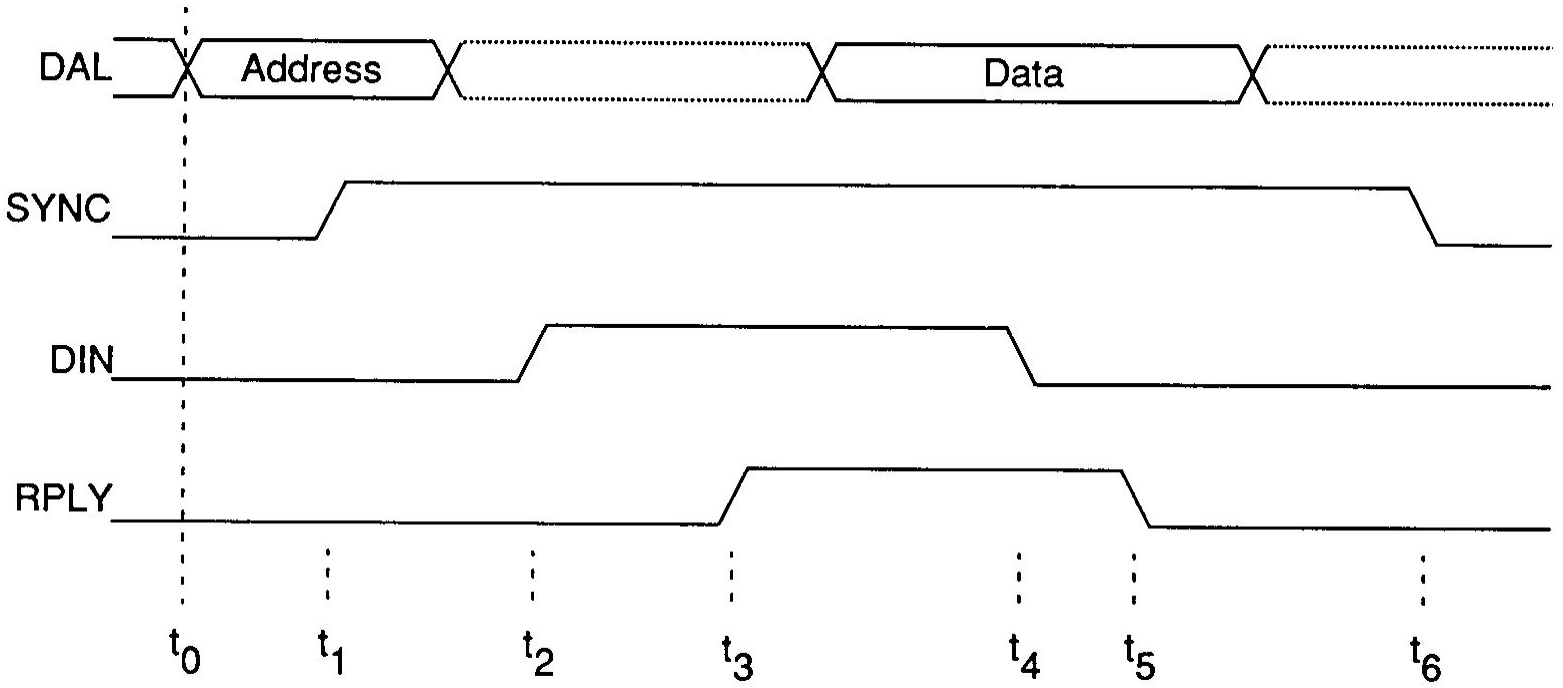




Példa 2.) Q-Bus: idő-multiplexált aszinkron protokoll

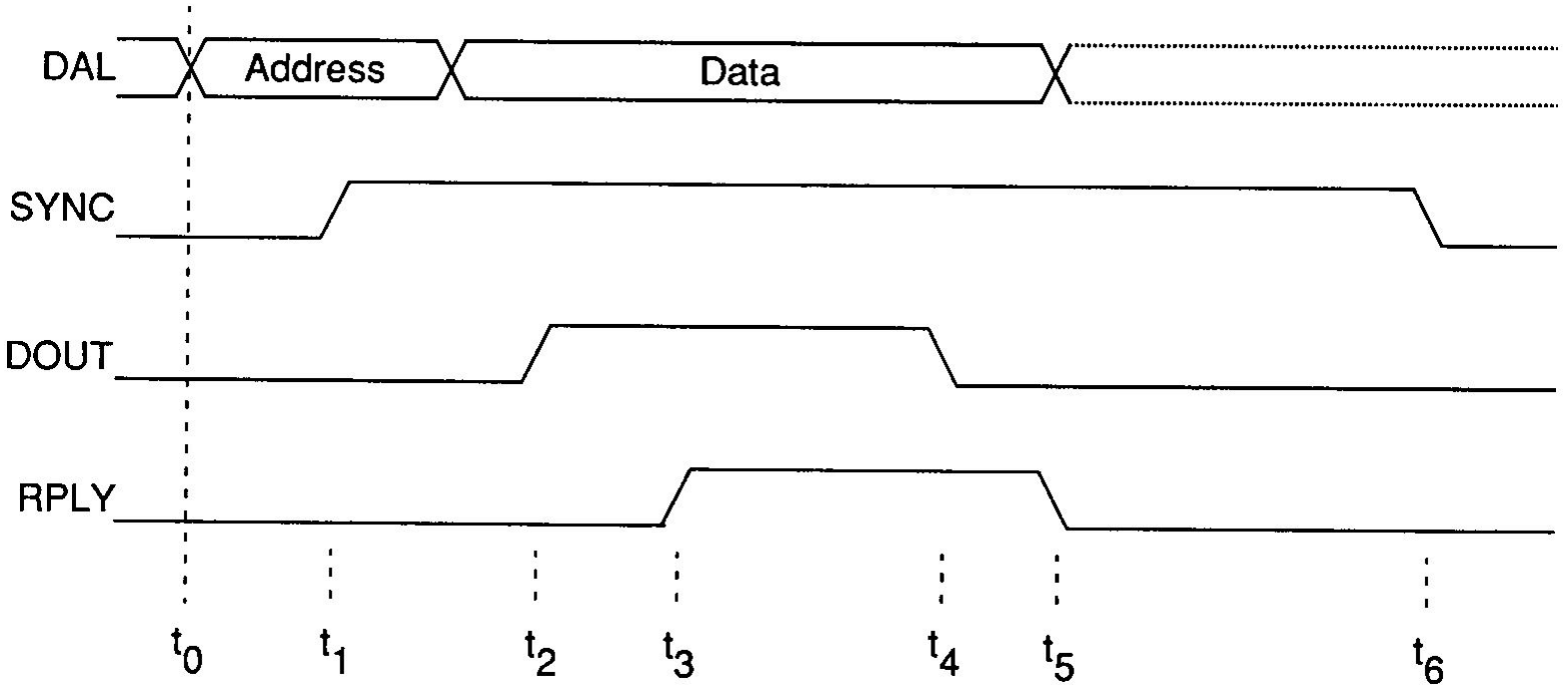
- DEC QBus: multiplexált adat/cím busz
- Olvasás: (Read) M olvas az S-től
- Írás: (Write) M ír az S-nek
- Jelei:
  - DAL: Data / Address Line (közös)
  - SYNC: Slave eszközöknél a cím esetleges tárolására (latch tárolóelemben), amelyet a Master kezel.
  - DIN: Master bemenő vonala (olvasáskor)
  - DOUT: Master kimenő vonala (íráskor)

Q-Bus: aszinkron olvasás (read)



- M beállítja a címvonalat (DAL) t_0-ban. (+ propagációs és skew time)
- S beállítja SYNC-et t_1-ben
- M felszabadítja az címnél a DAL vonalat, hogy később beállíthassa a DAL vonalat az adathoz is (címzett Slave-től fog olvasni)
- Ezt követi a 4-lépéses tranzakció, ahol DIN jelenti az igénylést t_2-ben
- A Slave nyugtázza az igényt RPLY-vonalon t_3-ban. (adat átvitel ekkor kezdődik)
- M válaszol erre, a DIN felengedésével t_4-ben.
- S észreveszi ezt és t_5-ben felszabadítja a RPLY, majd pedig DAL vonalat is.
- SYNC felszabadítása (t_6-ban) a Master által

Q-Bus: aszinkron írás (write)

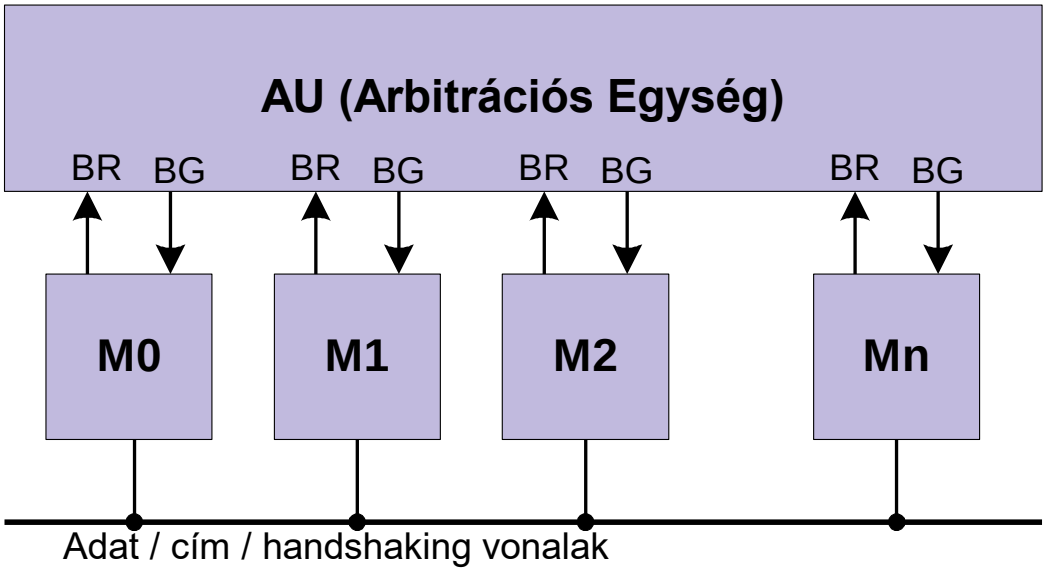


- M beállítja a címvonalat (DAL) t_0-ban. (+ propagációs és skew time)
- M beállítja SYNC-et t_1-ben
- Miután M felszabadítja az címnél a DAL vonalat, és már egyből beállítja a DAL vonalat az adathoz is (címzett Slave-nek fog írni)
- Ezt követi a 4-lépéses tranzakció, ahol DOUT jelenti az írási igényt t_2-ben
- A Slave nyugtázza az igényt RPLY-vonalon t_3-ban. (adat átvitel ekkor kezdődik a Slave felé)
- M válaszol erre, a DOUT felengedésével t_4-ben.
- S észreveszi ezt és t_5-ben felszabadítja a RPLY, és egyszerre a DAL vonalat is.
- Busz protokoll kérheti a Mastert hogy tartsa egy ideig a SYNC-t, majd felszabadítja (t_6-ban)

Arbitráció

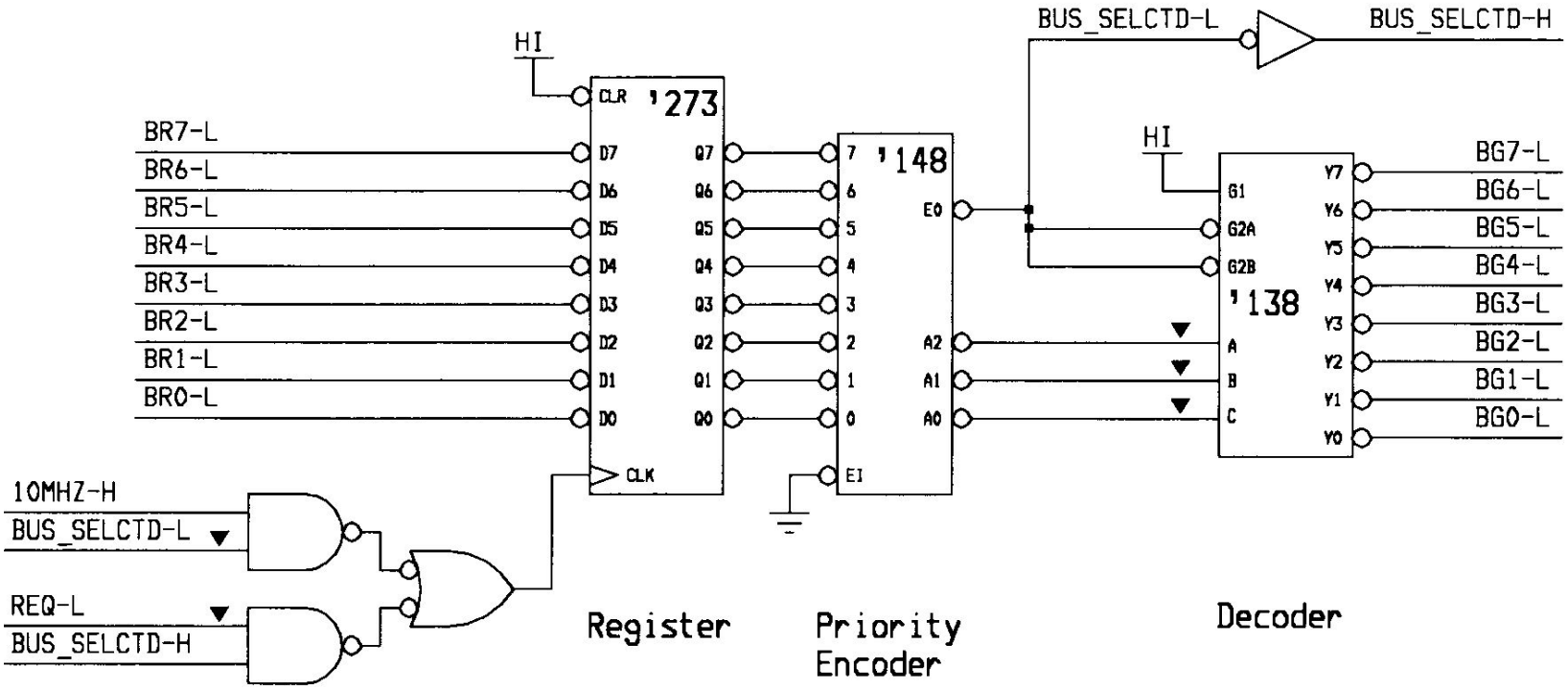
- Egy tetszőleges I/O művelet esetén (aszinkron v. szinkron buszos átvitel) több Master (commander) egység is meg akarja szerezni egyszerre a busz irányítását. Különböző előredefiniált algoritmusok segítségével egyértelműen azonosítható, hogy a „**versenyhelyzetben**” a következő átvitelt (a következő 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 M1...Mn-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



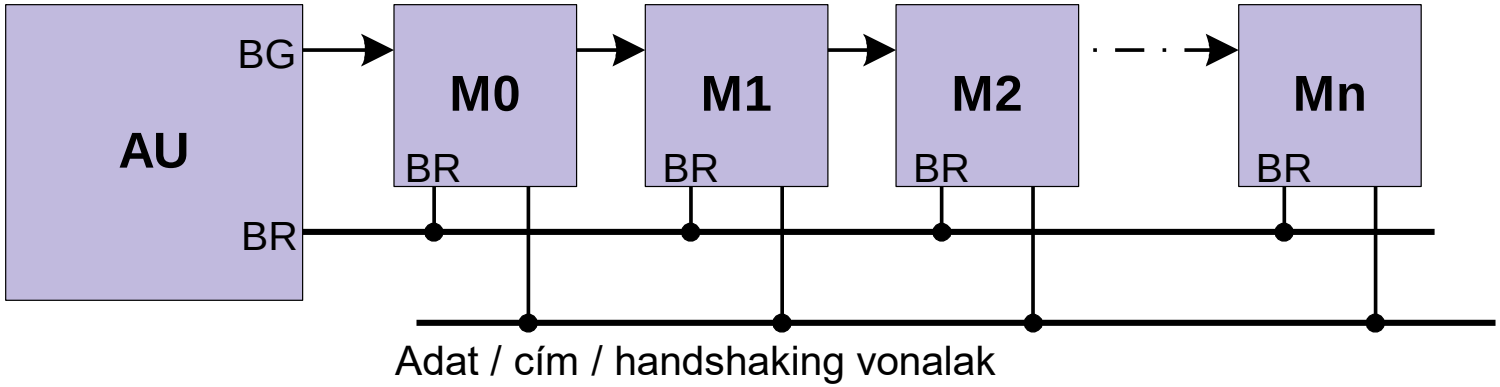
- Ez a leggyorsabb módszer, minden M_i-nek kitüntetett, egyenrangú kapcsolata van az AU-val (két vonalon: BG és BR-en). Az arbitráció lehet *(i) first asserted/first served (FIFO),* vagy (ii) *round robin*, vagy (iii) *prioritásos alapú.* Ezek a módszerek többfajta lehetőséget biztosítanak mind egyszerű, 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.

Példa: Párhuzamos arbitrációs rendszer



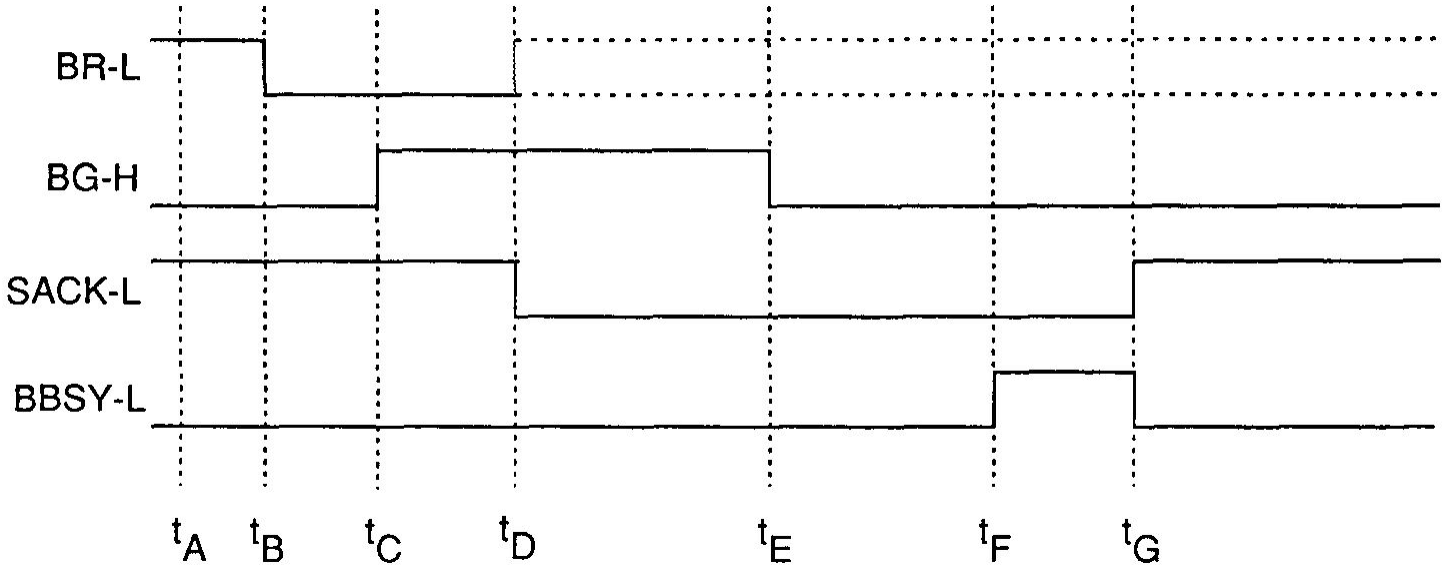
- 8 Master: egynek engedélyezi a busz használatát (BR→BG)
- Ha nincs Master, ami a tranzakciót irányítani tudná: **szinkron módban** 10 MHz-es órajellel működik. (10MHZ_H)
- Ha egy Master egység kezeli: **aszinkron handshaking mód** (REQ_L jel lefutó élére történik a szinkronizáció)
- ‘273: 8-bites Regiszter, ‘148: prioritásos 8→3 kódoló, ‘138: 3→8 dekóder

b.) Soros (daisy chain)



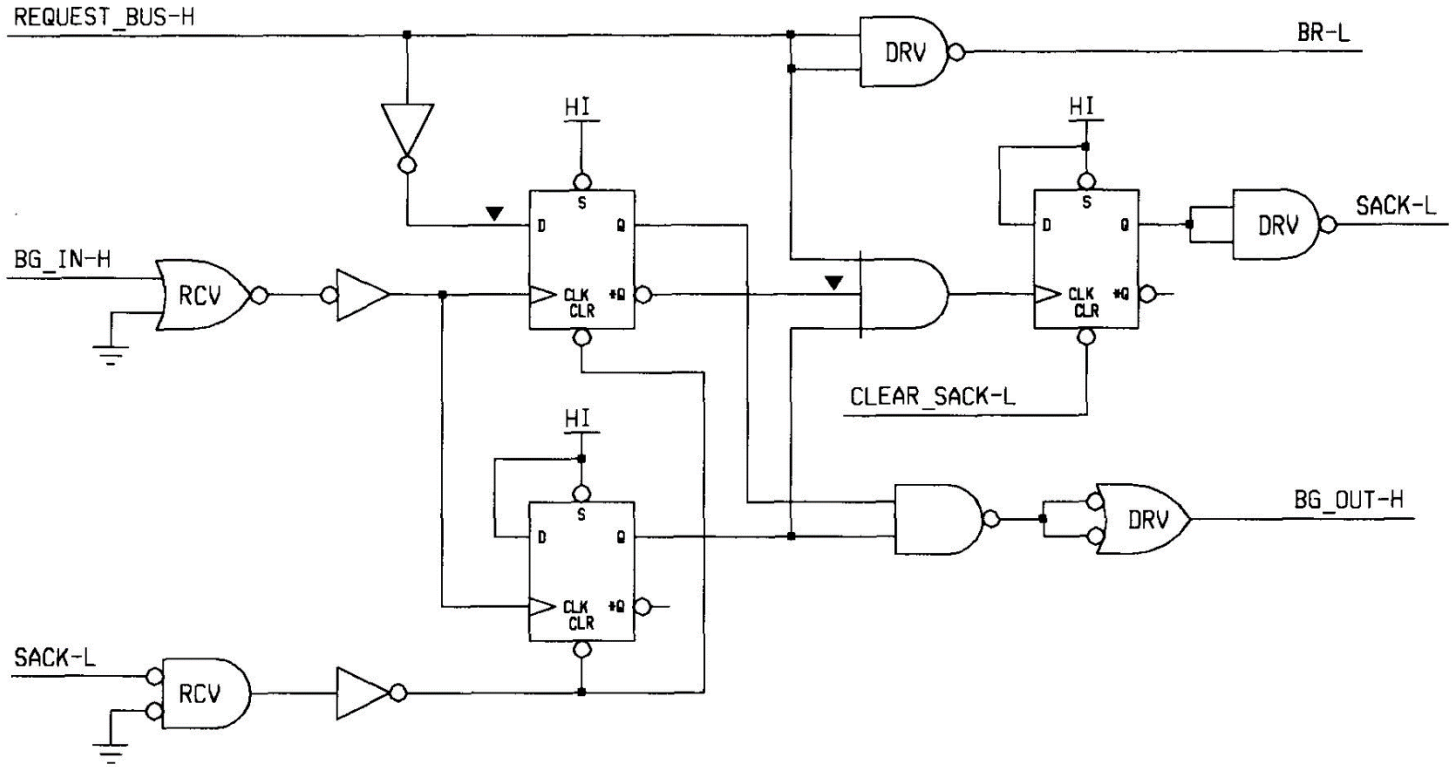
- A BG vonal sorosan van kötve, míg a BR vonal mindegyik M_i-hez csatlakozik. Az AU nem ismeri, hogy pontosan melyik M kívánja elérni a buszt, ezáltal az átvitel leegyszerűsödik: csupán azt tudja, hogy bizonyos ciklusonként BG-jelet kell kibocsátania. A soros csatlakozás miatt a legelső Master (M_0) 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.

Példa: UNIBUS - Soros arbitrációs rendszer



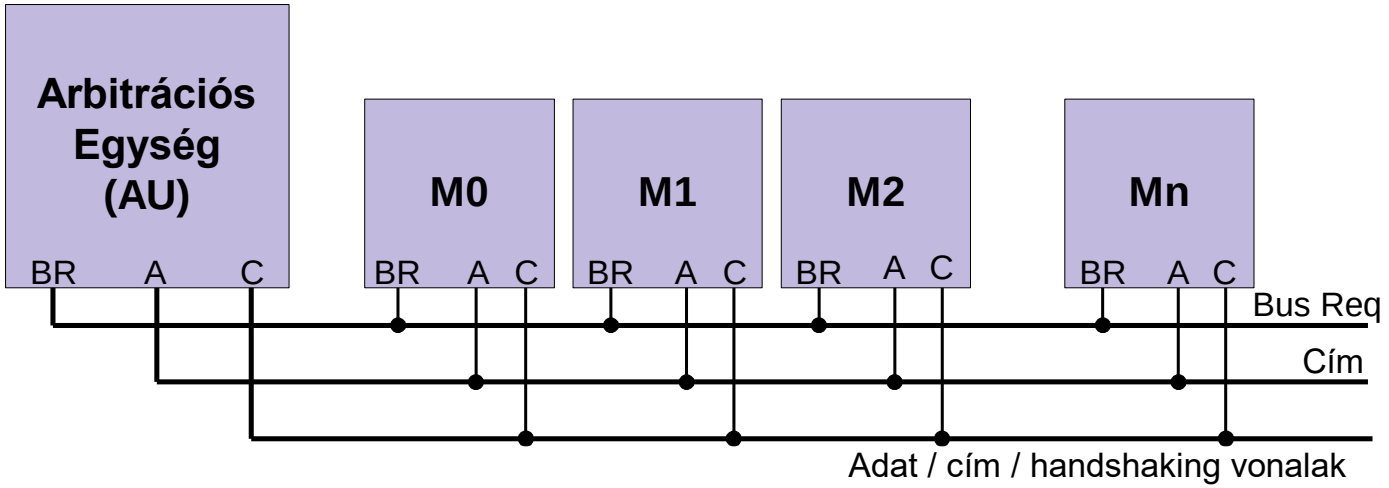
- Jelei: BR_L, BG_H, SACK_L (selection ack), BBSY_L (bus busy - foglalt)
- Lépései:
  - **t_A:** AU felismeri hogy egy új arbitrációs ciklusban vagyunk (SACK inaktív)
  - **t_B:** BR, Master igényel
  - **t_C:** AU nyugtázza a kérést, veszi az M igényét
  - **t_D:** BG jelet kap az M az AU-tól, SACK küldése
  - **t_E:** ha a tranzakció megtörtént, felengedi az AU a BG jelet
  - **t_F:** aktuális Master befejezi a ciklust, felengedei a BBSY jelet (következő Masternek adja át a döntési mechanizmusnak megfelelően, **sorban!**)
  - **t_G:** új busztranzakció kezdete a Master által (új ciklus)

Példa: UNIBUS - Soros arbitrációs rendszer felépítése



- Speciális kapuk
  - RCV: receiving bus singals
  - DRV: drive bus lines
- 3 db D-Flip-flop: BR_L, SACK_L, és BG_H vezérlőjelek tárolásához

c.) Lekérdezéses (polling)

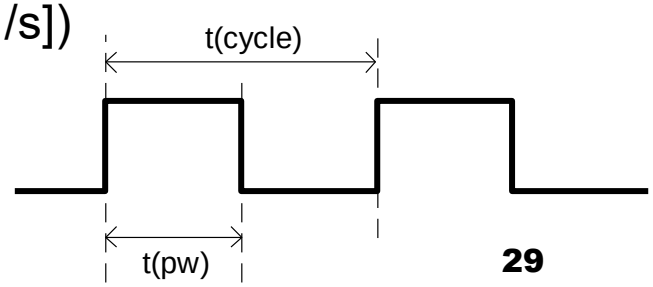


- Mindegyik Master egy közös BR buszon igényelhet, és az AU dönti el, hogy melyik Mastertől kapta a kérést. Annak a címét az *address* vonalra rakja. Minden ciklusban megnézi az igényléseket, és a legmagasabb prioritással rendelkezőt fogadja el. Itt is többféle prioritásos módszer megvalósítható: pl. (FIFO, round robin stb.).
- Hátránya, hogy nagyobb az időszükséglete a párhuzamosnál, így ritkábban alkalmazzák (pl: I/O kérések arbitrációjánál), segítségével a processzor egy program futtatásakor lekérheti az I/O eszközöket, megszakítást engedélyezhet.

2.) Szinkron busz protokollok

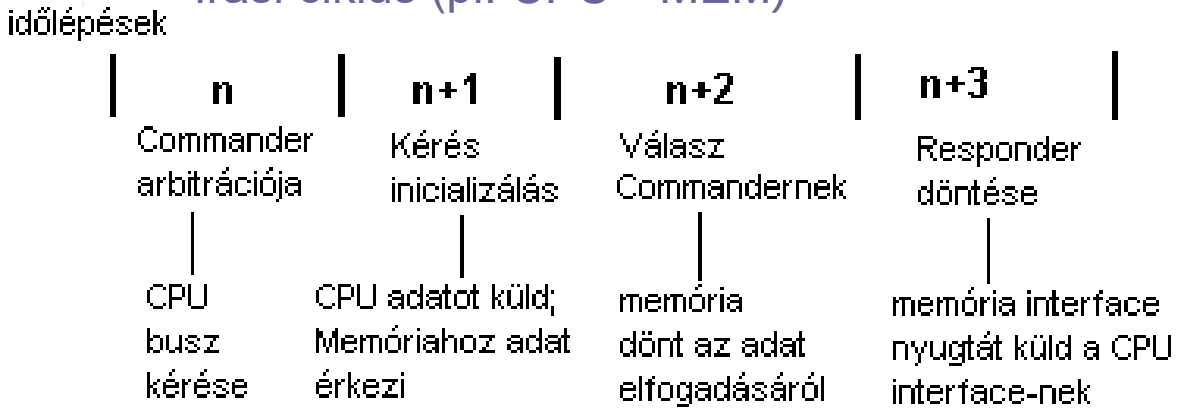
Szinkron busz protokollok

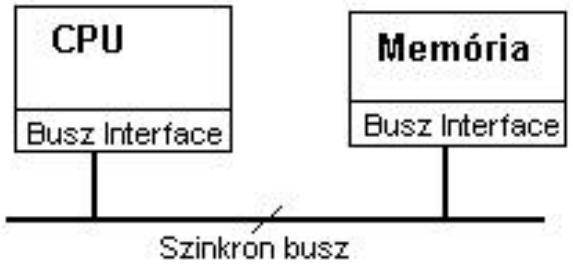
- Ebben az esetben a Master-Slave modulok **közös órajelet** (CLK-t): használnak. Itt a Master, mint Commander (kezdeményező), a Slave pedig (Responder) válaszol.
- *Előnye*: Tehát a műveleti időt az órajelciklus határozza meg. Mivel nincs szükség párbeszédre (pl. handshaking), gyorsabb lesz az aszinkron működésénél.
- *Hátránya*: Az órajelet mindig a 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 [Hz] ([1/s])
  - Ekkor T=1/f=1 / 100 000 000 = 10 [ns]
- Kristály-oszcillátor szolgáltatja ált. az órajelet.



Szinkron írási / olvasási ciklus

- Írási ciklus (pl. CPU→MEM)

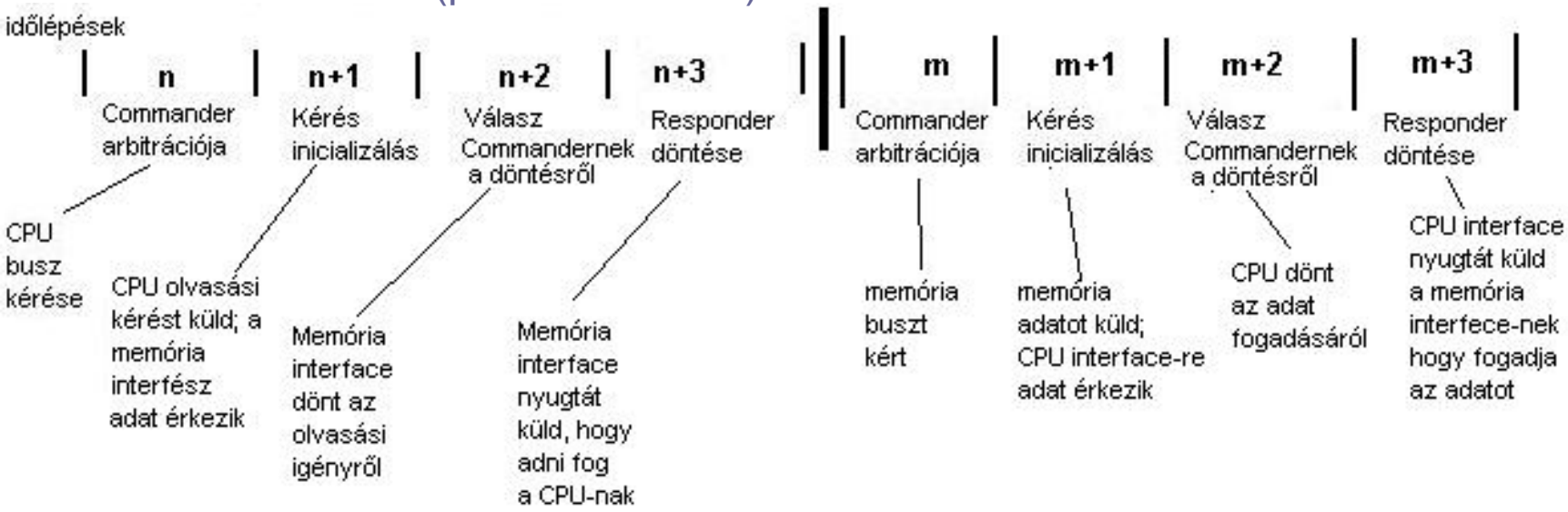




Szinkron adatátvitel 4 fő lépése:

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

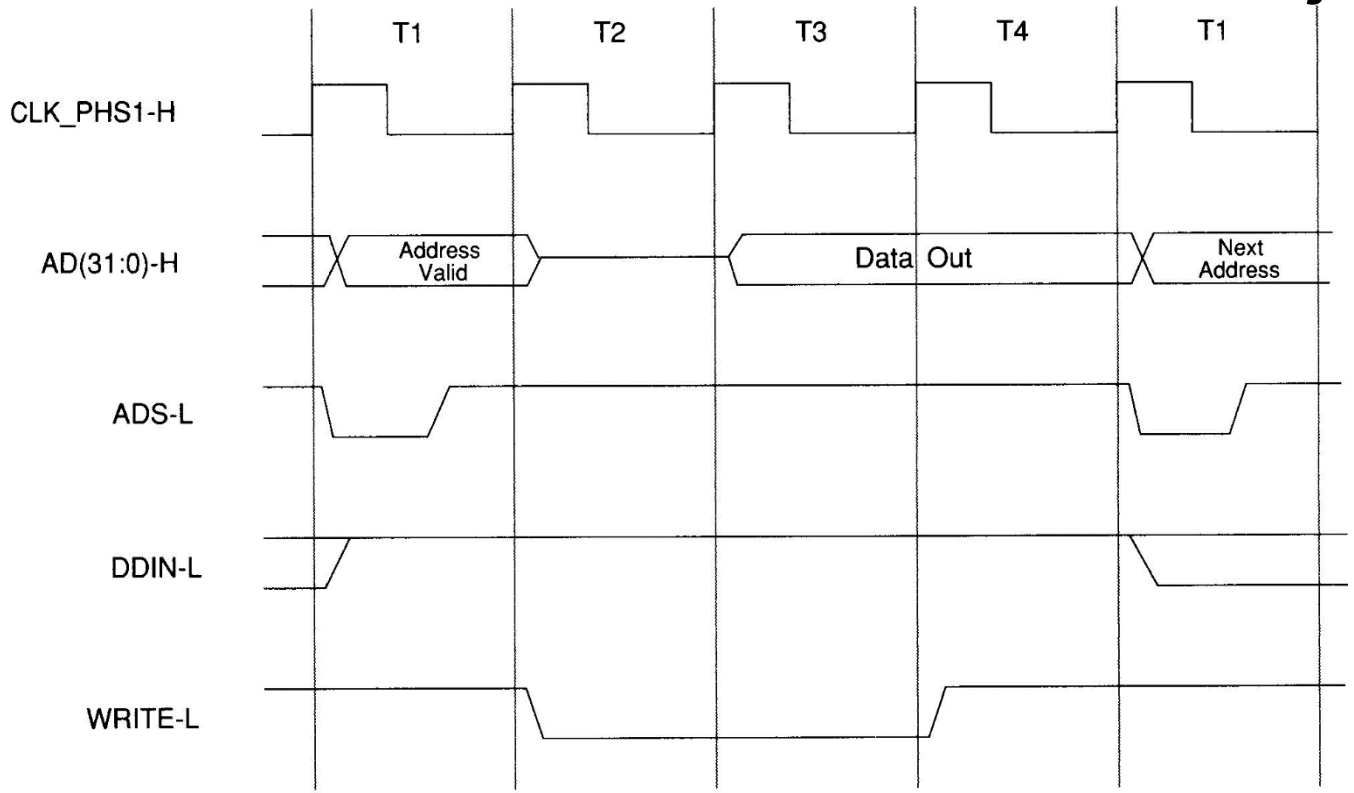
- Olvasási ciklus (pl. MEM→CPU)



Példák: Szinkron busz protokollok fajtáira

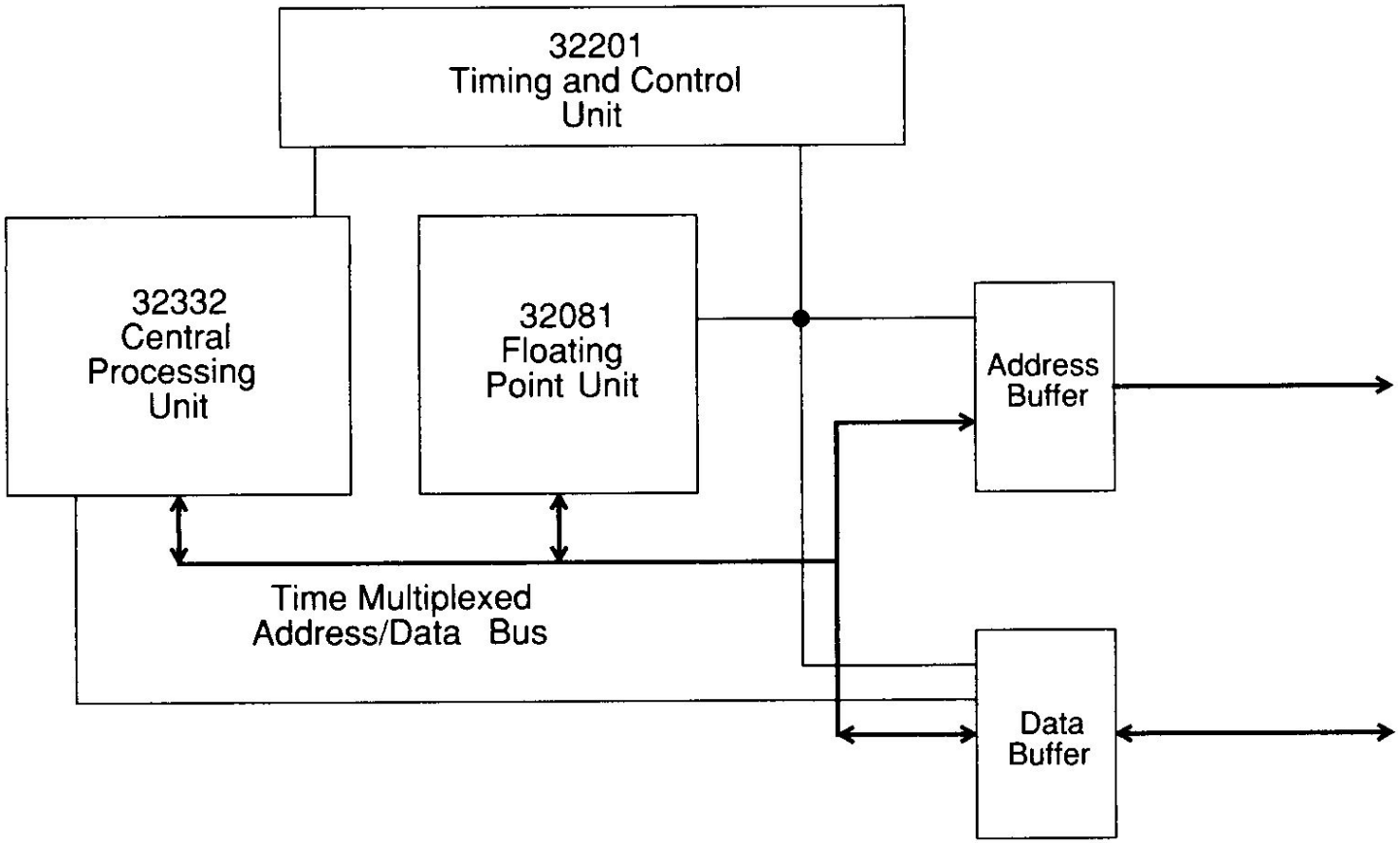
- 1.) **Általános célú busz**: CPU közvetlen vezérlése alatt áll (határozza meg a forrást ill. célt)
- 2.) **Motorola 68020**: nagyteljesítményű rendszer. Dinamikus buszméret. CLK-val szinkron kommunikáció. Ha a Slave rövid időn belül nem képes egy kérést kielégíteni, akkor a CPU automatikusan egy tétlen („idle”) állapotot, várakozó ciklust generál, amíg a Slave nem válaszol.
- 3.) **NS32332** buszrendszer: 32-bites szinkron rendszer. Idő-multiplex Addr/Data kommunikáció (IC lábszám csökkentése miatt).

Példa: NS32332 Busz írási tranzakciója



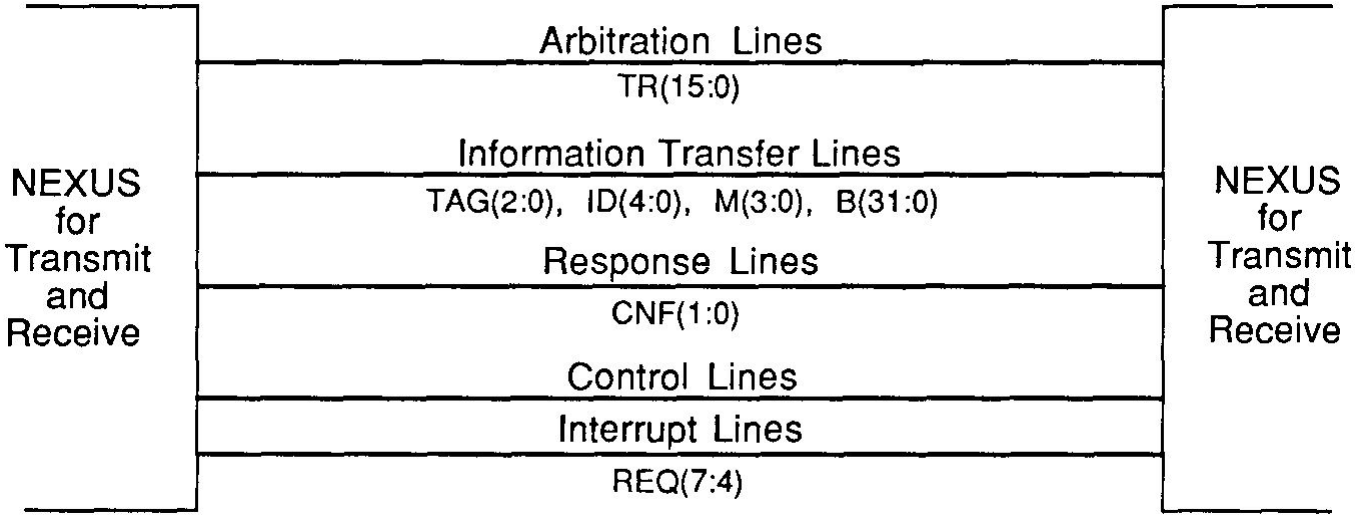
- AD (31:0)_H : 32-bites multiplexált cím/adat vonal (érvényes adat t_3-ban)
- ADS_L: érvényes cím érkezését jelzi t_1-ben
- WRITE_L: írás engedélyezés t_2-ben
- CLK_PHS1_H: közös rendszer-órajel (felfutó élre)
- DDIN_L: adatátviteli irány
- Minimális tranzakcióhoz 4 ciklus szükséges (t_1-t_4):

Példa: NS32332 Busz interfész



- 32-bites rendszer
- Idő-Multiplexált kommunikációhoz külön cím és adat bufferek kellenek

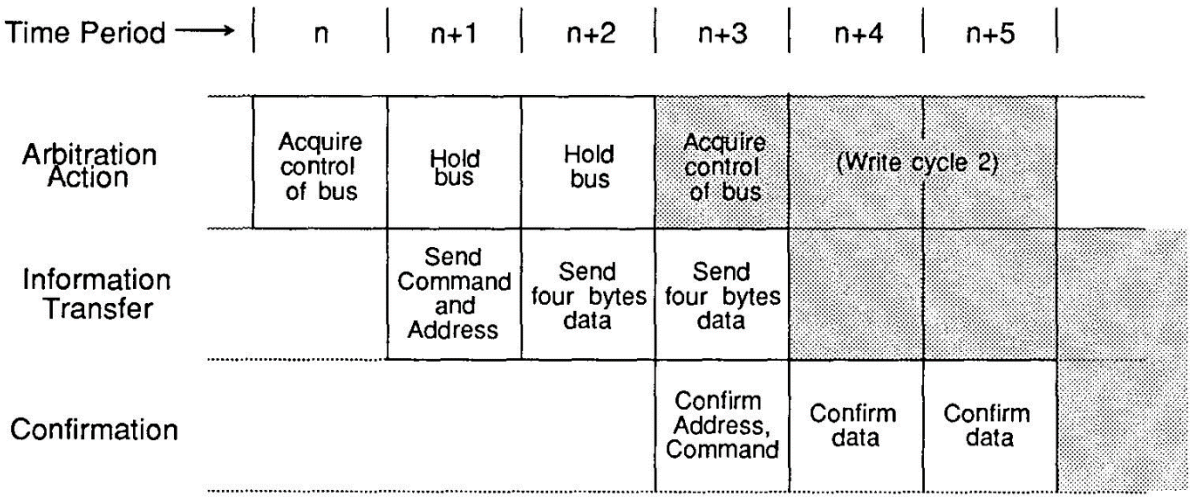
Példa 2.) Szinkron busz protokoll (SBI – VAX 11/780)



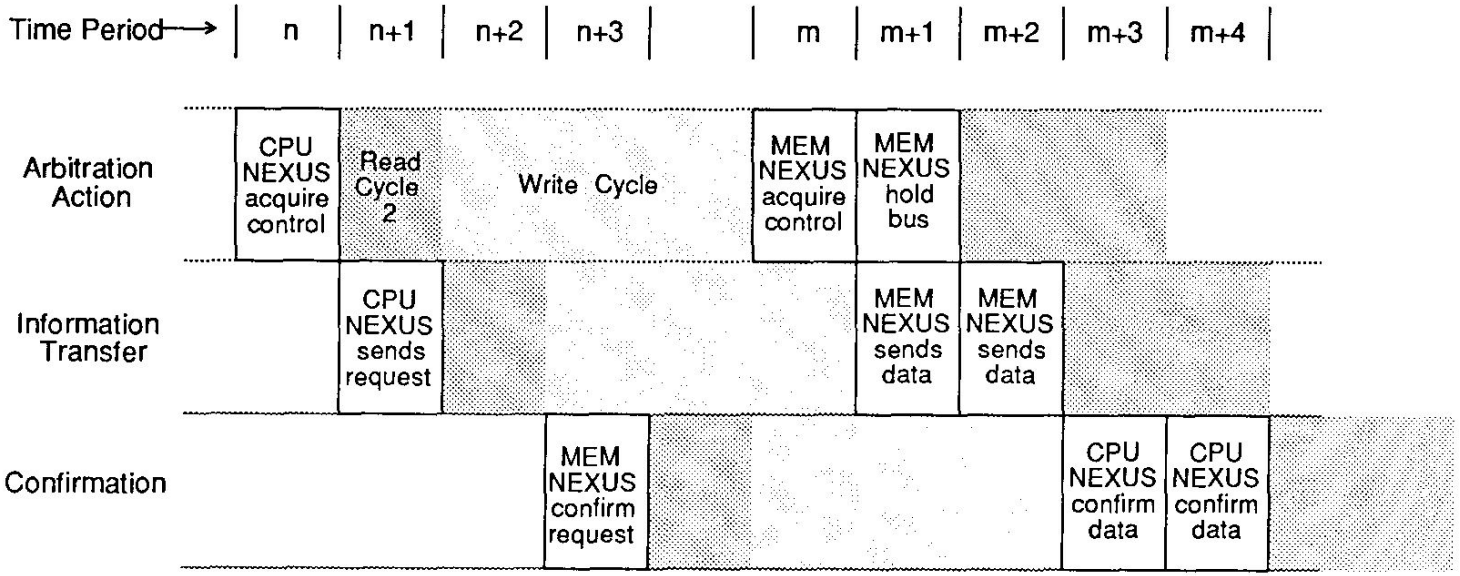
- TR<15:0>: arbitrációs vonal, 16 modult kezel
- B<31:0>: 32-bites busz (multiplexált AD)
- CNF<1:0>: Confirm – válasz, nyugta vonal
- Interrupt (megszakítás) / vezérlő vonal
- Information Transfer vonal: TAG, ID, M, B
- Írás: több mint egy ciklus ideig tart!

SBI – Szinkron busz protokoll „**pipeline**” tranzakciói

- **Írási ciklus (8 byte)**



- **Olvasási ciklus (8 byte)**

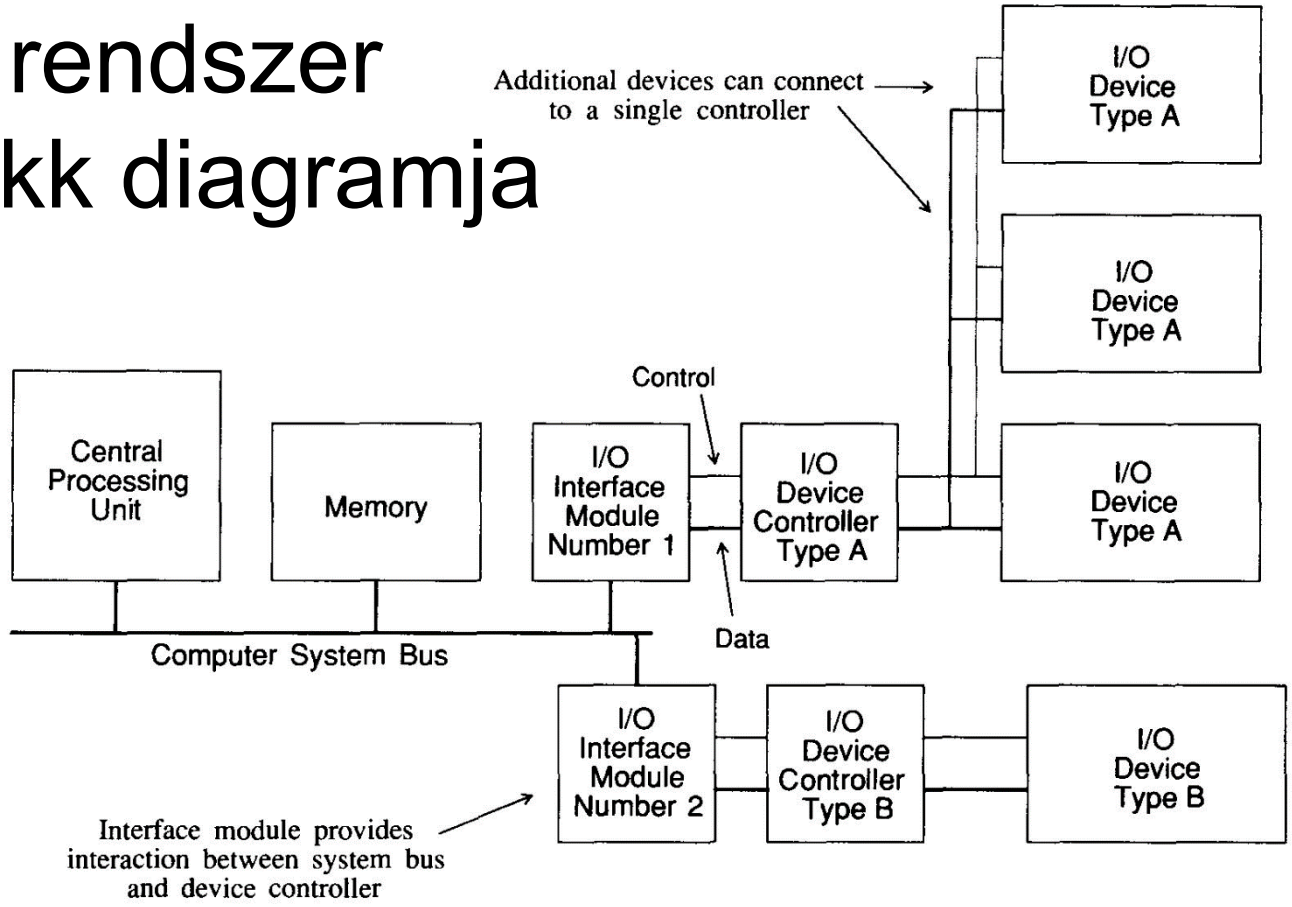


Adatmozgatás:
I/O kommunikációs technikák

I/O adatátvitel típusai:

- Programozott I/O átvitel
- Megszakításos (interrupt) átvitel
- Direct Memory Access (DMA) - Közvetlen memória-hozzáféréssel rendelkező átvitel
- I/O processzoros átvitel

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 eszközvezérlőbe kell bekötni.

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 („mappelés”)

2.) Megszakításos (interrupt) átvitel

- Megszakítással (interrupt) jelezhető a cpu-által az I/O eszköz számára az adatátviteli igény, illetve az adatátvitel befejeződése
- Megszakítás kérelem (interrupt request) – IRQ szintek
  - Megszakítási vektorok (maszkolható megszakítások – SW interrupt)

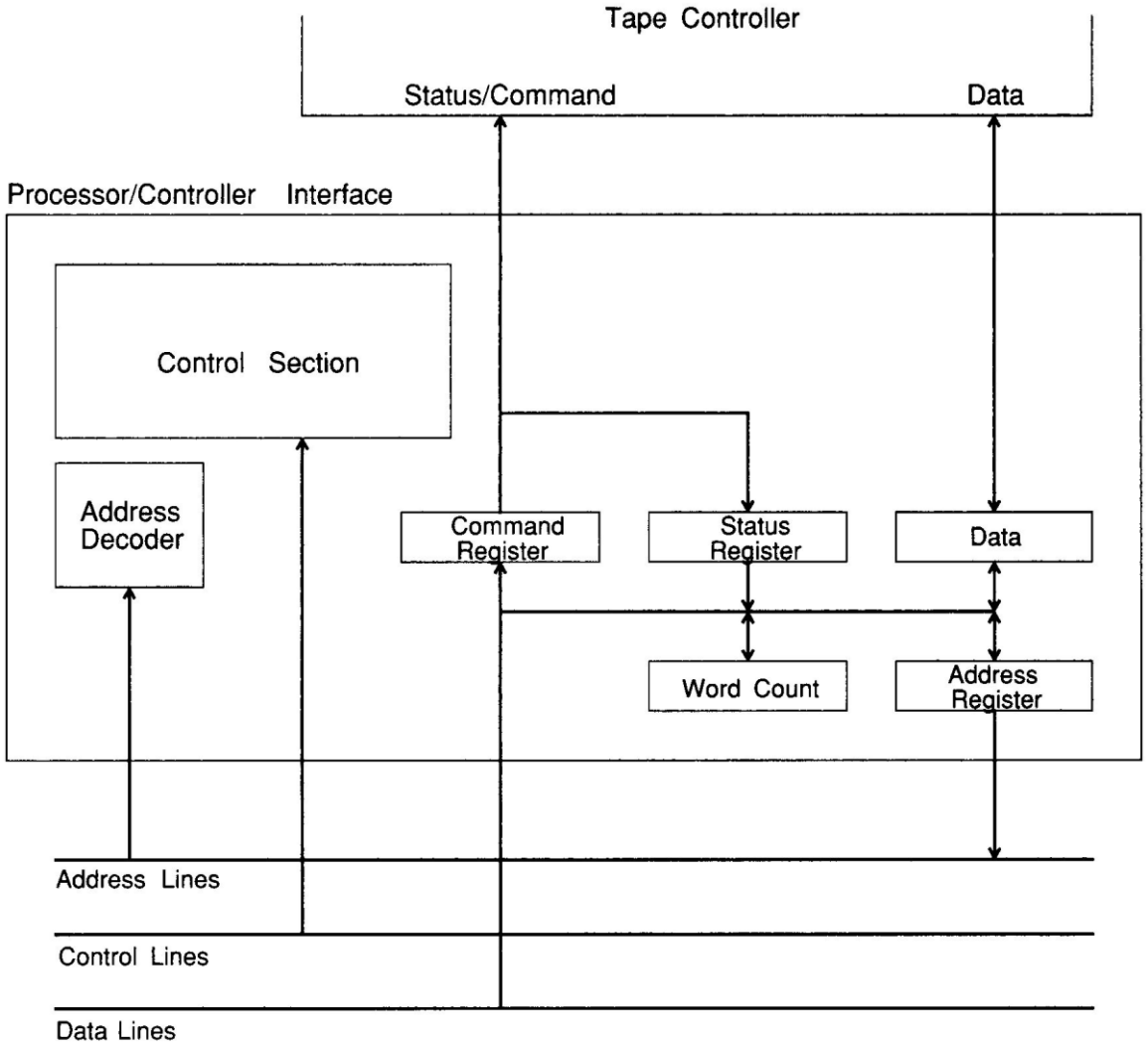
3.) Direct Memory Access (DMA)

- Közvetlen memória hozzáférés:
  az I/O eszköz és memória közötti adatátvitelt a processzortól függetlenül egy DMA eszközvezérlő végzi.
- Cél a cpu tehermentesítése a tranzakció idejére
- Cpu feladatai csupán:
  - az átvitel előkészítése (kezdőcím és adat hossza),
  - 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 processzornak, amely teljesen önállóan szabályozza a tranzakciót
- Főleg a mainframek-re jellemző módszer
- Rendkívül gyors
- I/O csatornák: eszköz sebessége szerinti osztályozás

Példa: Szalagos egység interfész modulja DMA átvitellel



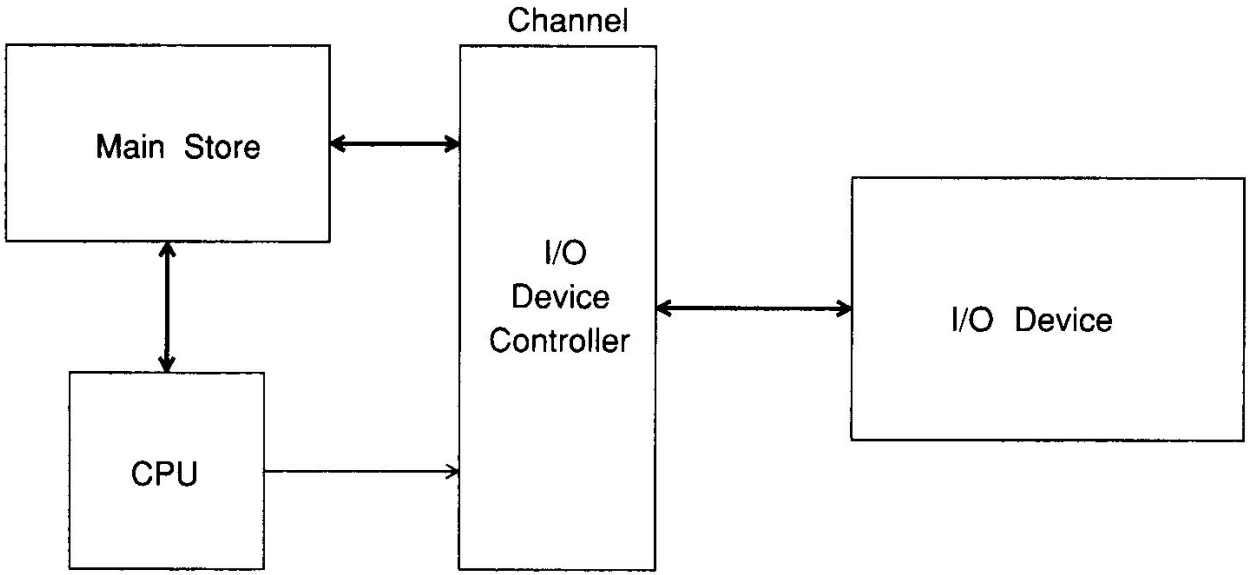
Komplex interfész:

a leggyakoribb és időigényesebb művelet a MEM↔periféria adatátvitele.

DMA: közvetlenül a MEM-val képes kommunikálni, tartalmazza:

- programozott I/O elemeit
- státusz regisztert
- parancs regisztert
- WC: word countert (számláló)
- AR: címregisztert

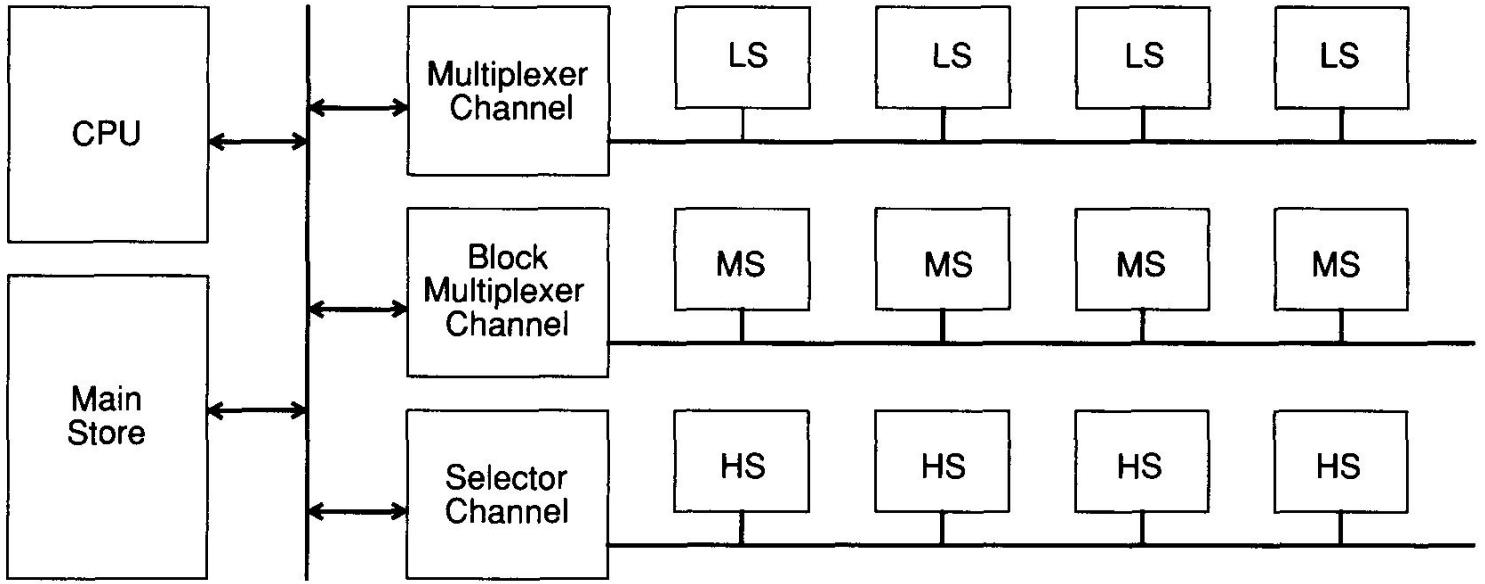
a.) I/O channel – I/O csatornák



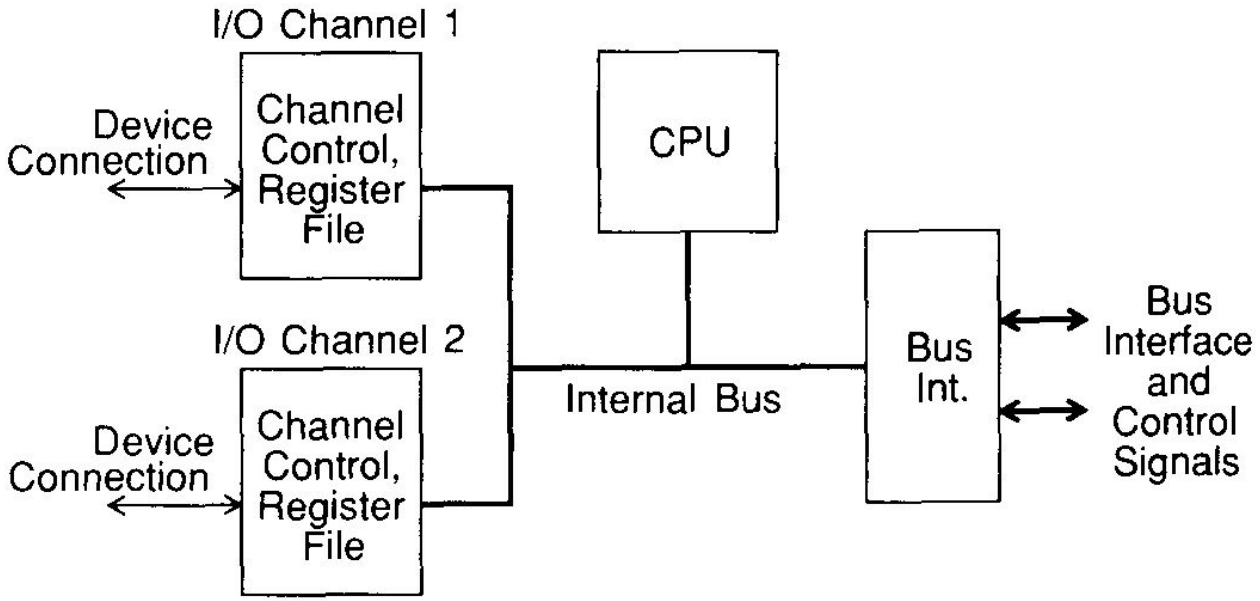
- A cpu csak az eszközvezérlőn keresztül (közvetetten) érheti el a perifériát.
- Channel = I/O Device Controller: általános célú processzáló elem.
- Channel feladata:
  - Konverzió
  - Adatmozgatás
  - Hibaellenőrzés és kezelés

Példa: I/O channel - csatornák

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

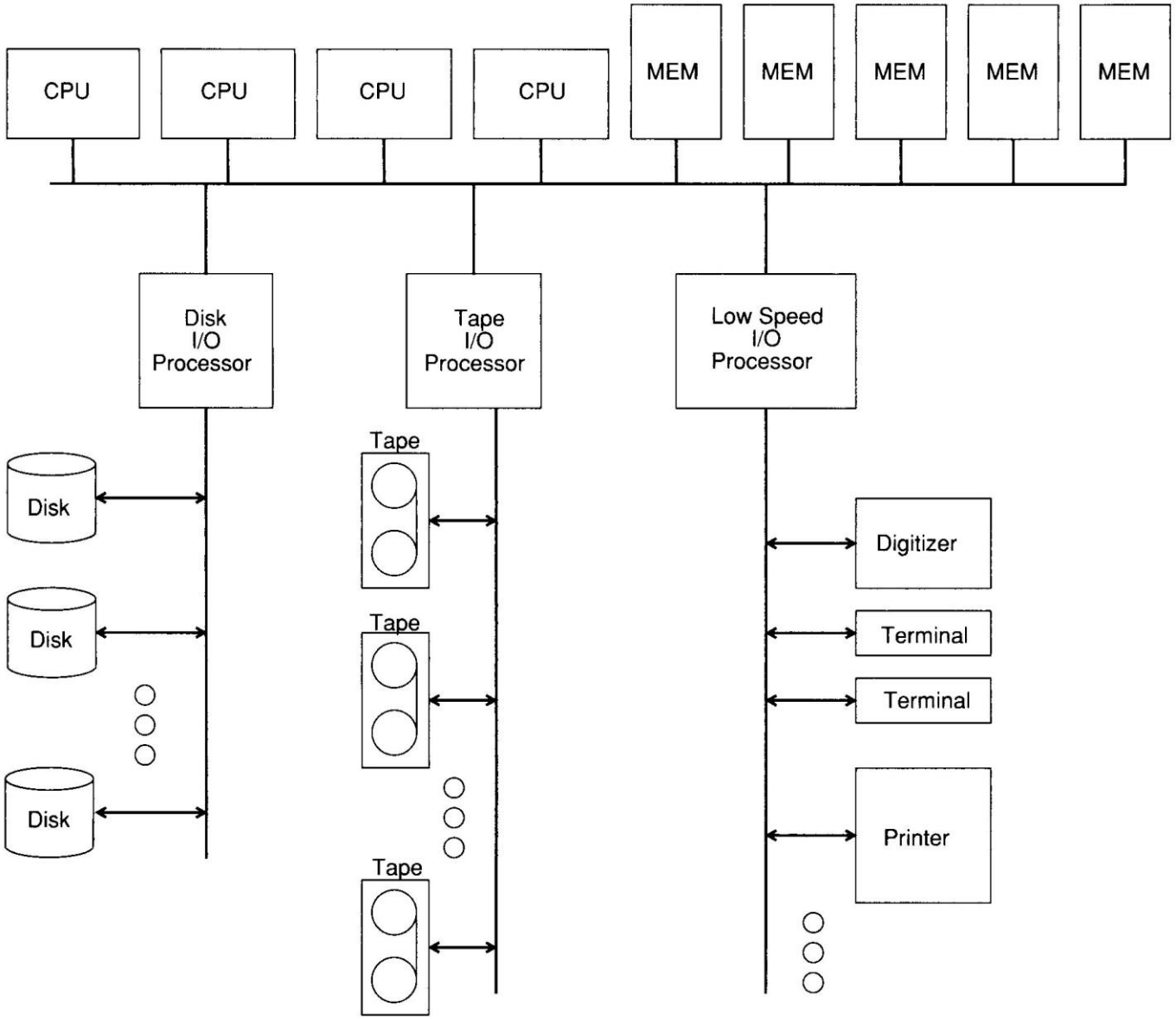


b.) I/O Processzorok (IOP)



- IOP: Intelligens eszközvezérlők.
- Saját, dedikált funkciókkal rendelkeznek (vezérlés és interfészt biztosít más rendszerekkel – különböző sebességen)
- Példa: SCSI rendszer is egy IOP

Példa: több I/O-processzoros eszköz



PCI busz

Kapcsolódó segédanyag: pci.pdf

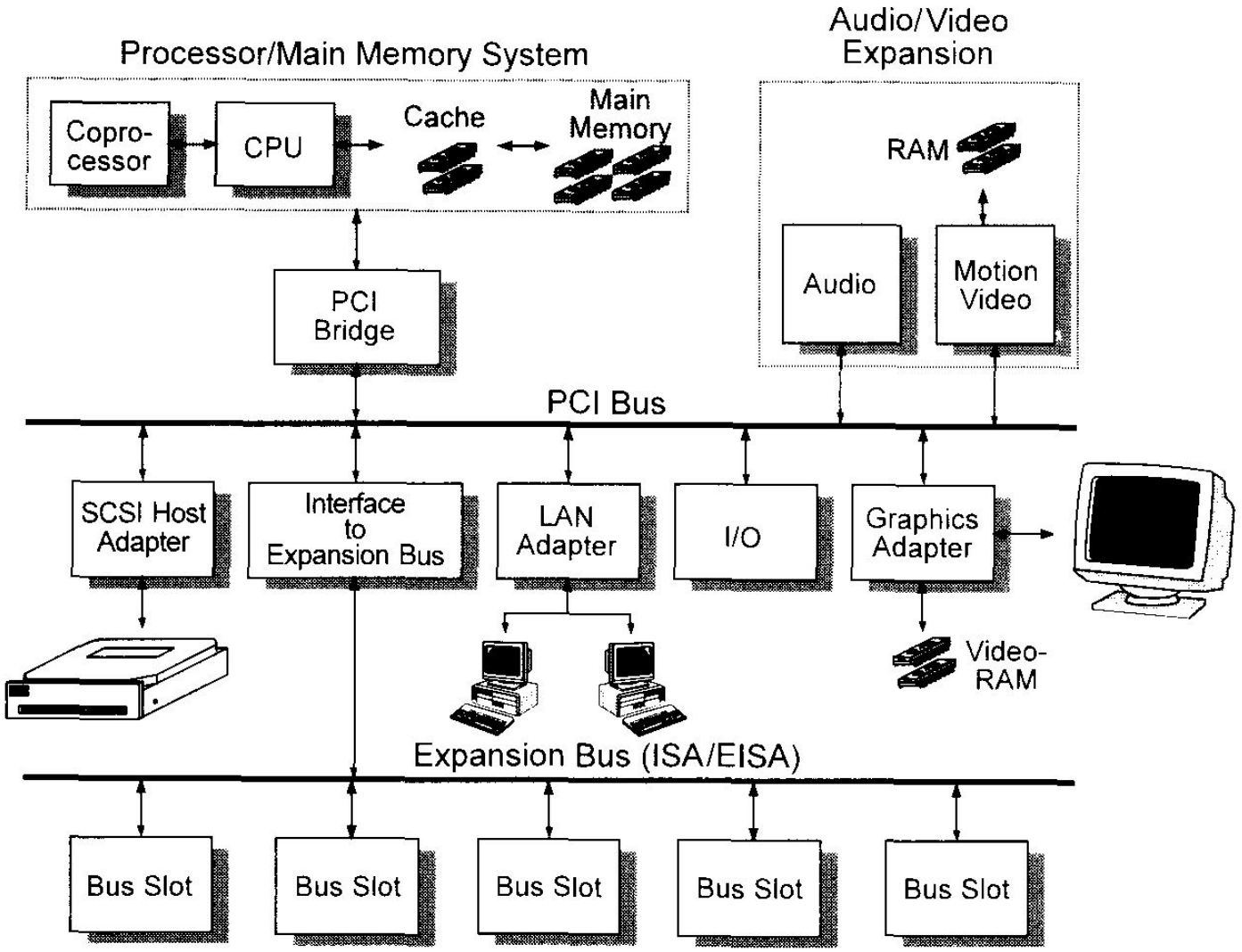
PCI busz

- **PCI= Peripherial Component Interconnect**. újszerű, a korábbiaktól önálló szabvány (ver 2.1): 33MHz-es órajel támogatás. Többprocesszoros rendszereket is támogat.
- Kapcsolatot a processzor és a PCI sin 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.

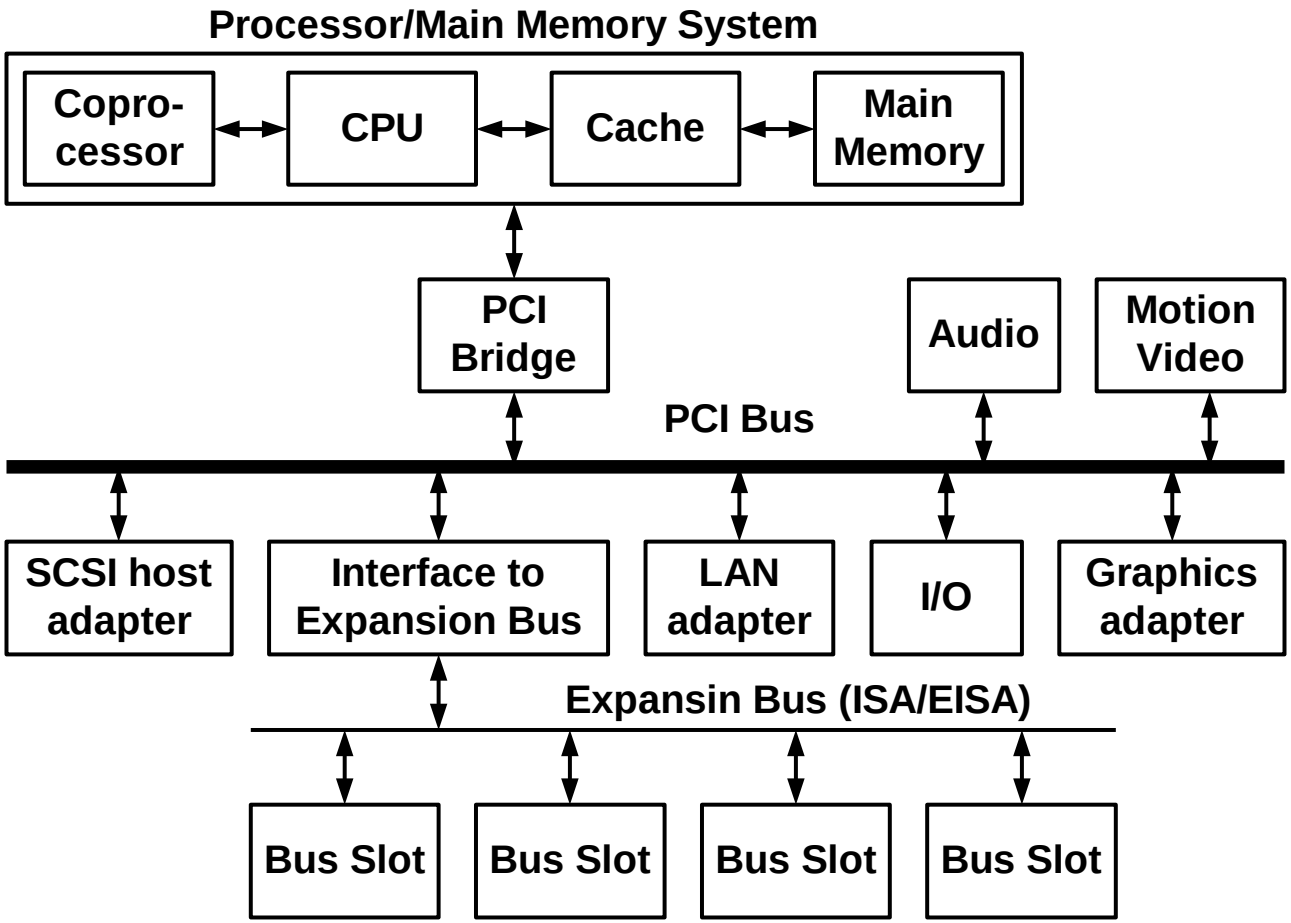
PCI buszrendszer tulajdonságai:

- A PCI 32 bites **multiplexált** adat/cím jeleket alkalmaz
  - Maximális átviteli sebessége (4-es „**burst-**onként” - löketszerűen) 133Mbyte/sec (= 4 byte*33MHz).
- (64 bites változata is létezik, főleg szerverekben alkalmazzák).
  - Ekkor a maximális átviteli sebessége 266Mbyte/sec
- 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.

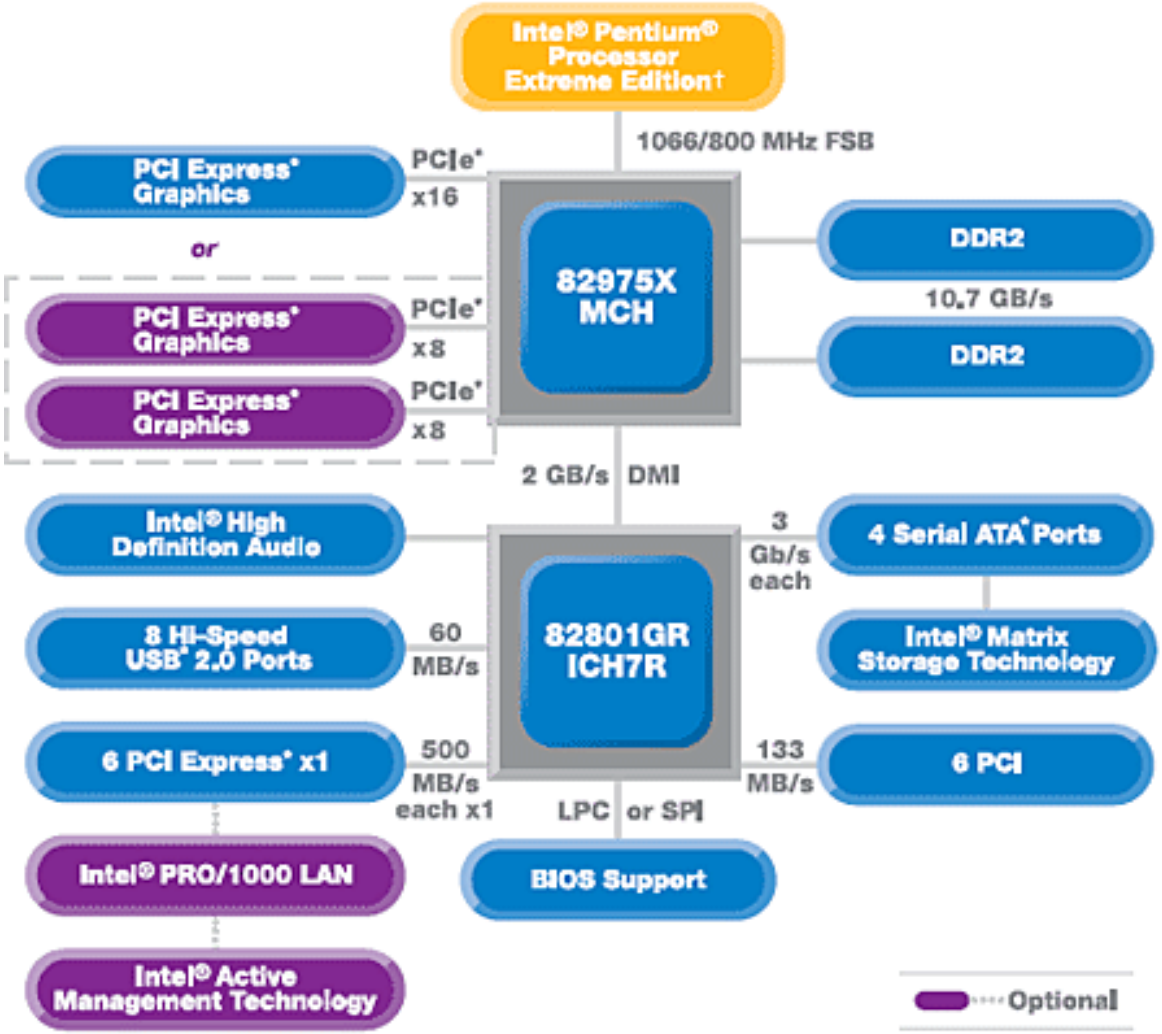
Példa: PCI buszos számítógép



Példa: PCI busz rendszer általános blokkdiagramja

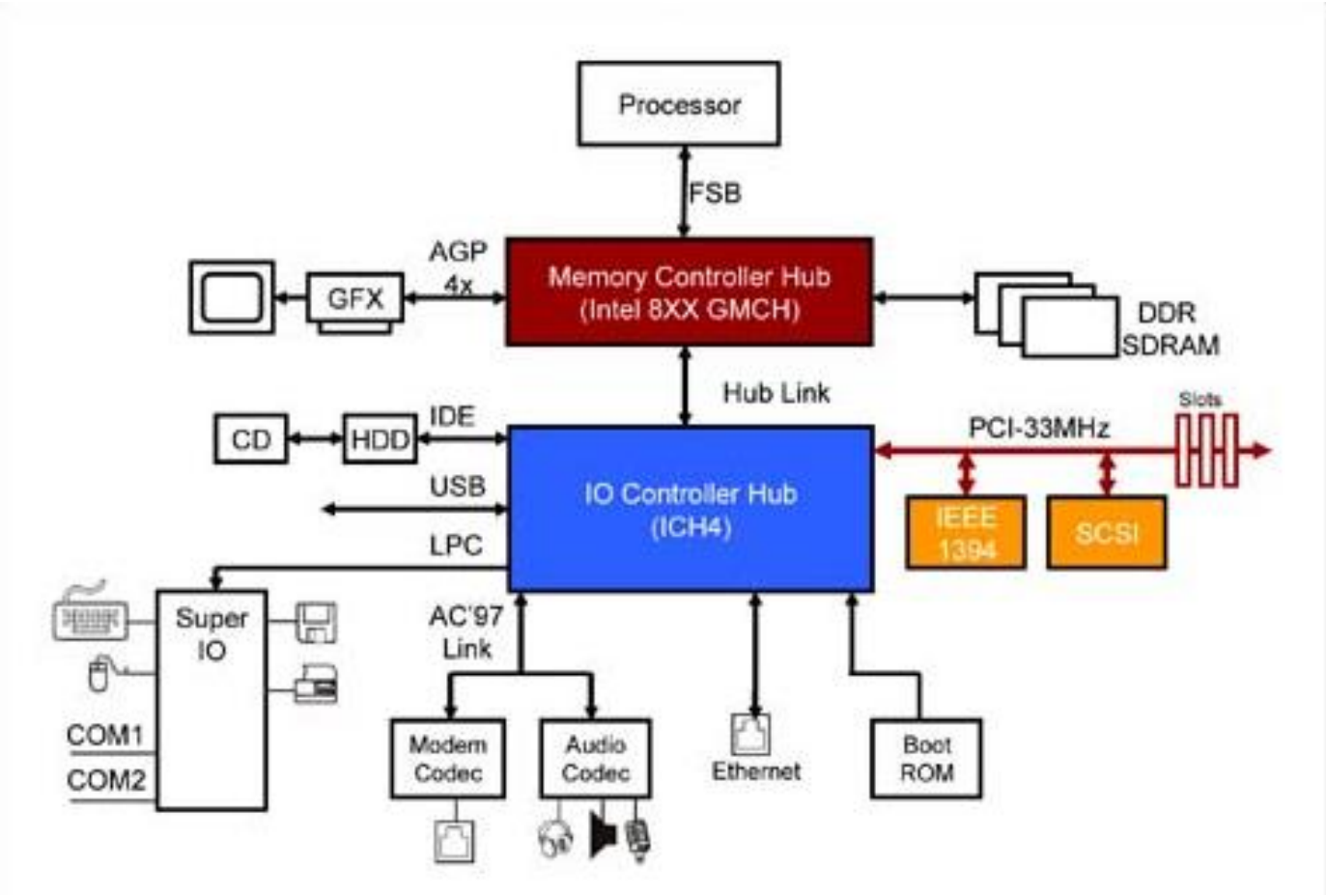


Példa: Korábbi PCI chipset (Intel ICH7R – déli híd)



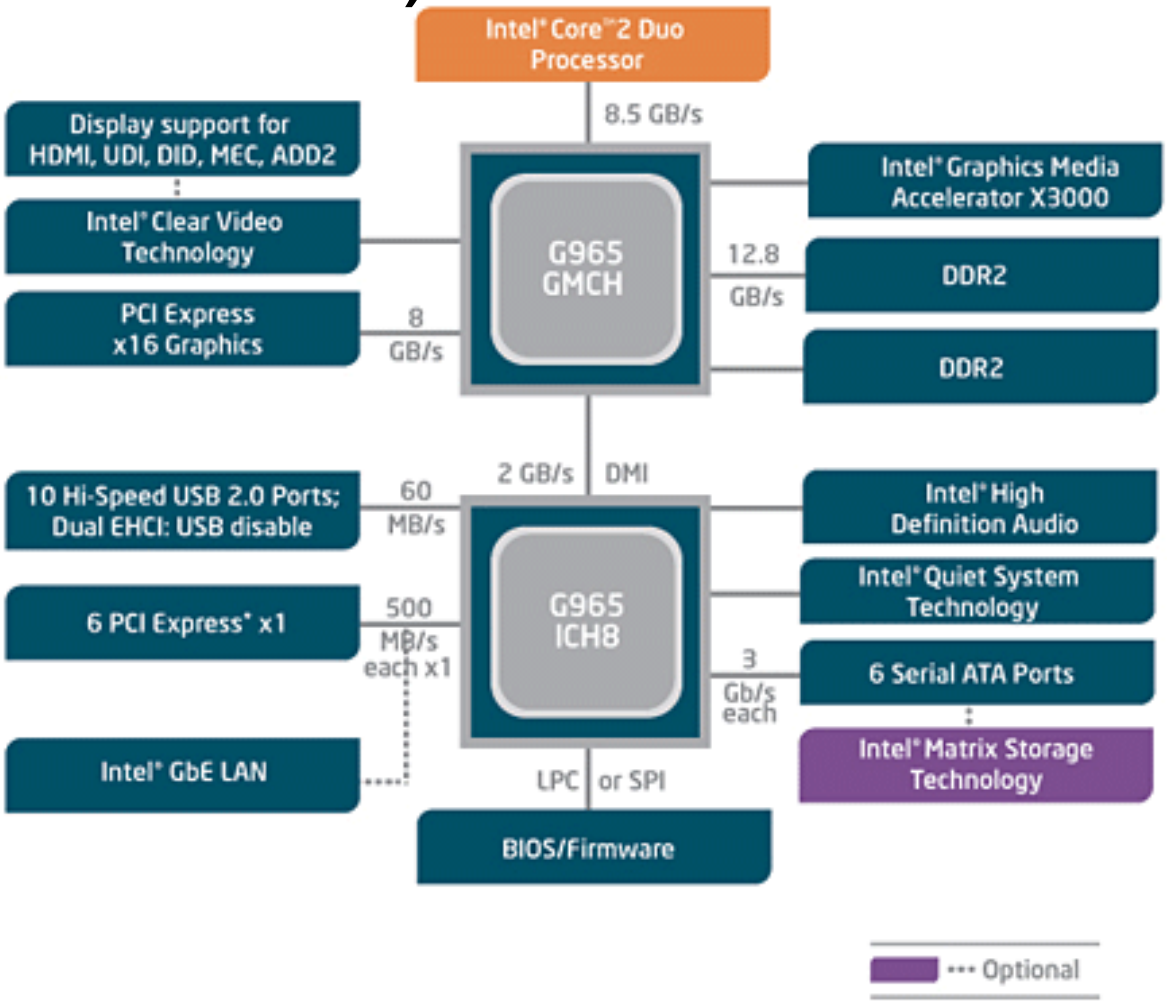
*2005. november adat

Példa: 33MHz-es PCI busz alapú platform



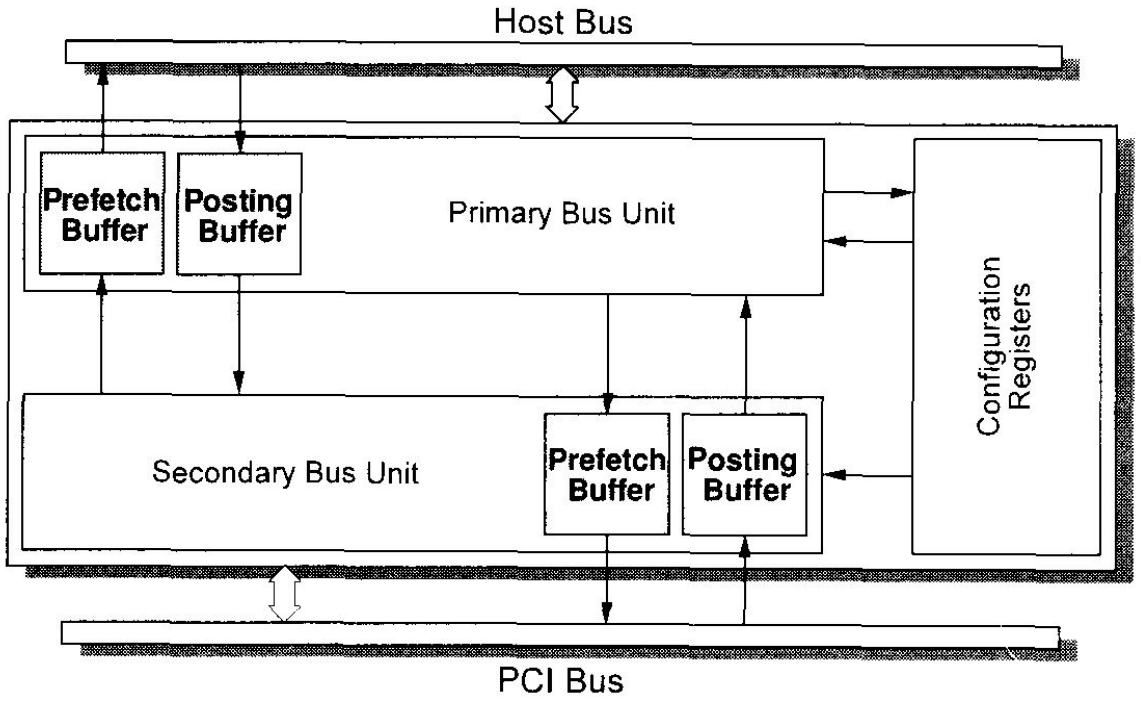
Példa: Újgenerációs PCI chipset (Intel ICH8 – déli híd)





*2006. júliusi adat

PCI Bridge (CPU – PCI)



- A PCI BRIDGE a PCI buszt köti össze a rendszerbusszal (CPU LOCAL BUS –al és MEMORY BUS –al egyben), tehát a HOST busszal. A konfigurációs regiszter a PCI BRIDGE részét képezi.
- A Host busz kapcsolódik az elsődleges busz egységhez, amelyben (Read) Prefetch és (Write) Posting bufferek találhatóak. Ez az egység közvetlenül is tud kommunikálni a másodlagos buszegységgel, amely a PCI buszhoz kapcsolódik. De a két egység megcímezhető a konfigurációs regiszteren keresztül is (szoftveres és regiszteres konfiguráció).

PCI busz 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)

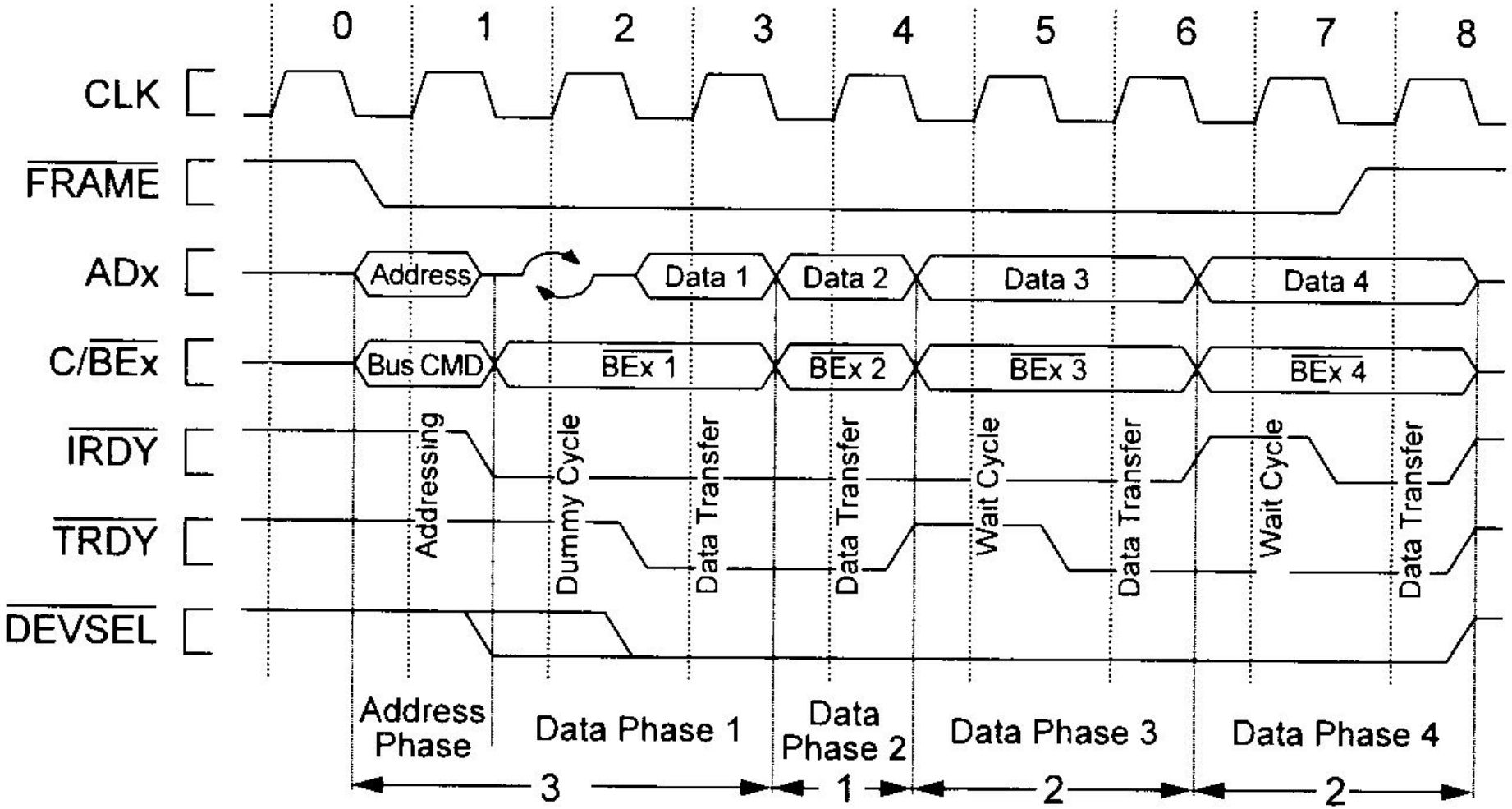
PCI busz jelei #2

- **PAR**: (Parity) adatok és címek paritás ellenőrzése
- **C/BE3 – C/BE0**: (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

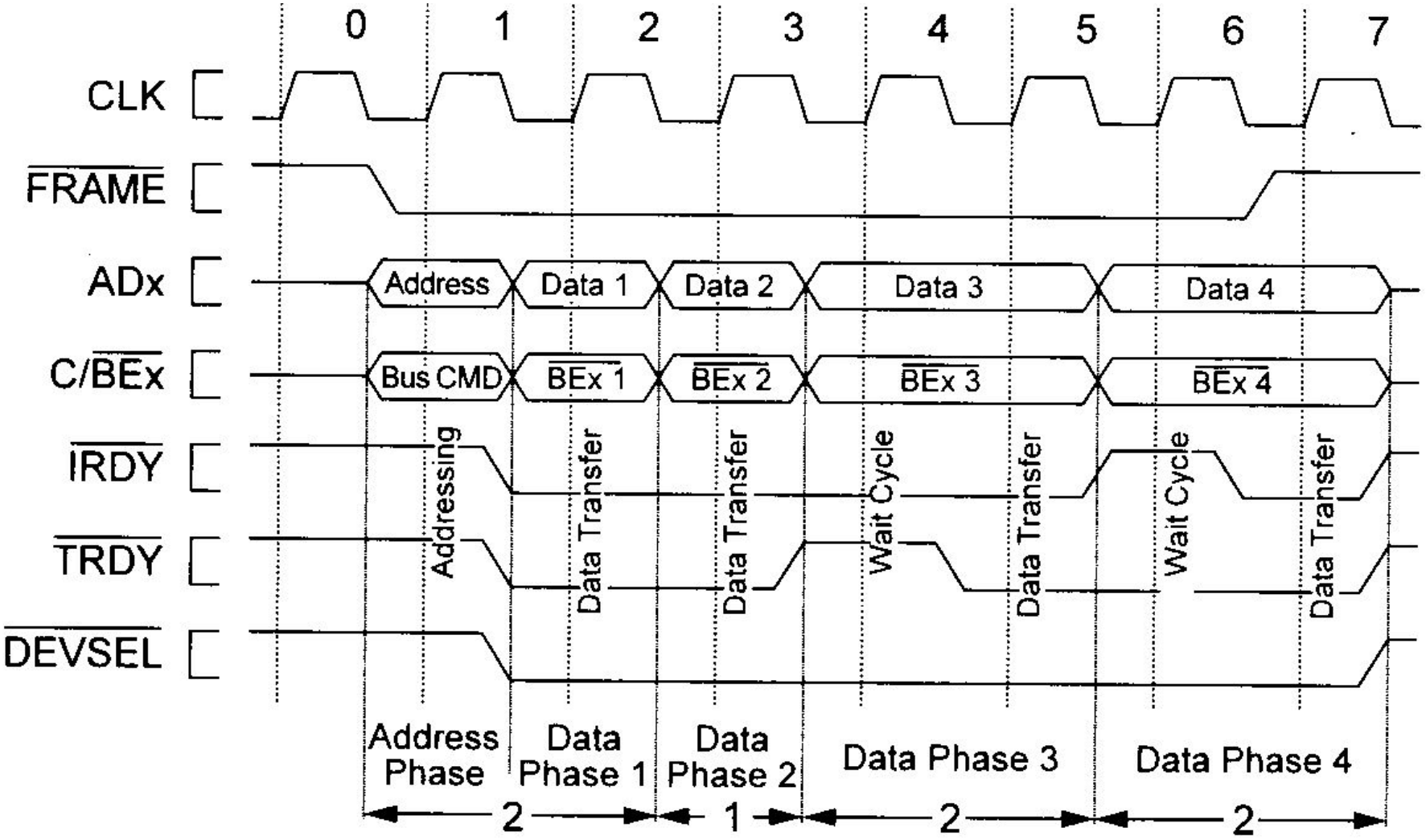
- **Burst: „löketszerűen”** egyszerre több adatot szeretnénk kiolvasni. 1 cím kiadása után 4 adat jön (4-es burst). A címzési fázis utáni első órajelciklusban az átvitel irányát módosítani kell, a közös multiplexált ADDR/DATA busz miatt. Olvasásnál ezért van szükség egy üres ciklusra (Dummy). Például: olvasásnál 3-x-x-x, vagy írásnál 2-x-x-x
- Miután az arbitráció során az egység sikeresen megkapta a vezérlést, a FRAME jellel 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= 3 órajelciklust vár az első adatig, 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 olvasási ciklus (3-1-2-2 burst)



* Optimális a 3-1-1-1 lenne.

PCI írási ciklus (2-1-2-2 burst)



* Optimális a 2-1-1-1 lenne.

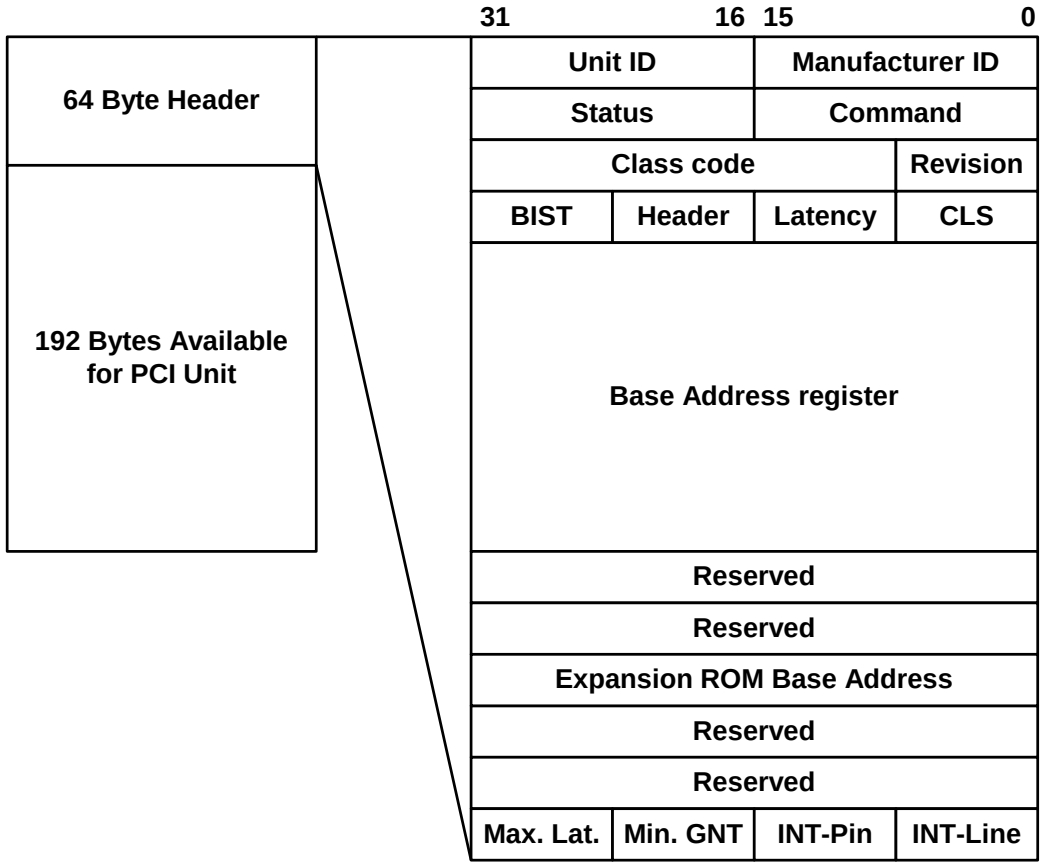
PCI átviteli módok

- Lehetséges busz arbitrációk:
  - Párhuzamos
  - Rejtett
  - Nem definiált algoritmus
- DMA
  - 4-es Burst átvitel (löketszerűen) írás / olvasásnál
- Interrupts (megszakítások)
  - INTA# aktiválása
  - Data: megszakítás vektor

PCI busz ciklusok

- INTA bitsorozat (0000)
- Speciális ciklus (0001)
- I/O read (olvasás) (0010)
- I/O write (írás) (0011)
- memory read access (0110)
- memory write access (0111)
- configuration read access (1010)
- configuration write access (1011)
- memory multiple read access (1100)
- dual addressing cycle (1101)
- line memory read access (1110)
- memory write access with invalidation (1111)

PCI Konfigurációs címtartomány



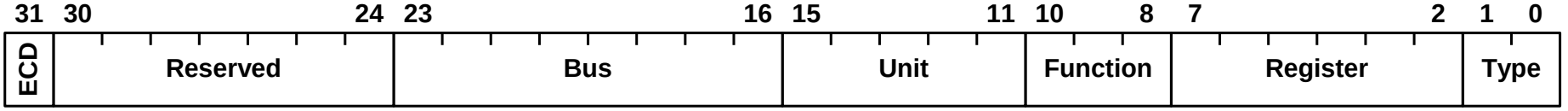
- Keret összesen 256 byte, amiből
  - 64 byte fejrész
  - 192 byte elérhető rész
- Gyártó ID
  - lefoglalva PCI SIG (PCI szabványcsaládért felelős érdekszövetség) által
- Egység ID, revision
  - Egység azonosítására
- Osztály (class) kód
  - PCI egység típusa

Státusz és Parancs Regiszterek jelei

- Status:
  - PER: Parity error
  - SER: System error
  - MAB: Master abort
  - TAB: Target abort received
  - STA: Target abort signaled
  - DEVTIM: DEVSEL timing
    - 00=fast 01=medium 10=slow 11=reserved
  - DP: Data parity error
  - FBB: Fast back-to-back cycles supported/unsupported
- Command:
  - BEE: Fast back-to-back cycles (Back-to-Back Enable)
  - SEE: SERR Enable
  - WC: Wait cycle control
  - PER: Parity error (Parity Error Response)
  - VPS: VGA palette snoop
  - MWI: Memory write access with invalidation
  - SC: Special cycle
  - BM: Busmaster
  - MAR: Activate/deactivate Memory address area
  - IOR: Activate/deactivate I/O address area

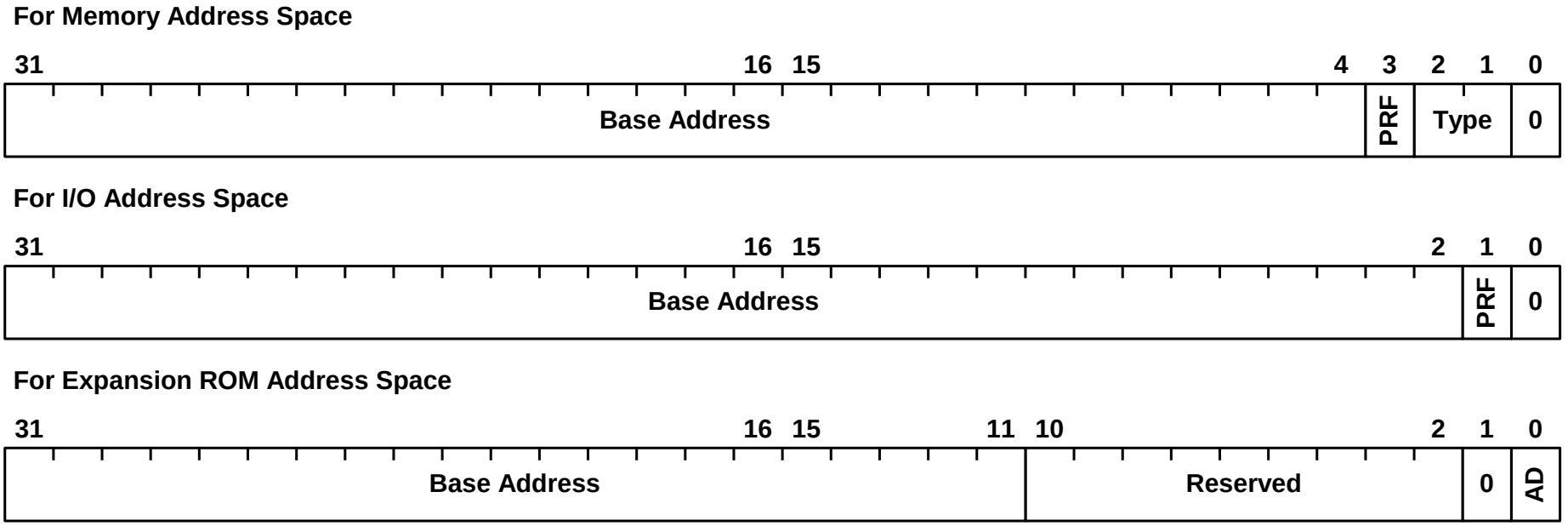
Konfigurációs címtartomány elérési módszerei

- Konfigurációs Módszer #1
  - CONFIG-ADDRESS (0cf8h) és CONFIG-DATA (0cfch) regiszterek az I/O tartományon definiáltak



- Konfigurációs Módszer #2 (PC-s rendszereknél)
  - 4k I/O címtartomány ‘c000h – cfffh’ között

Báziscím regiszterek



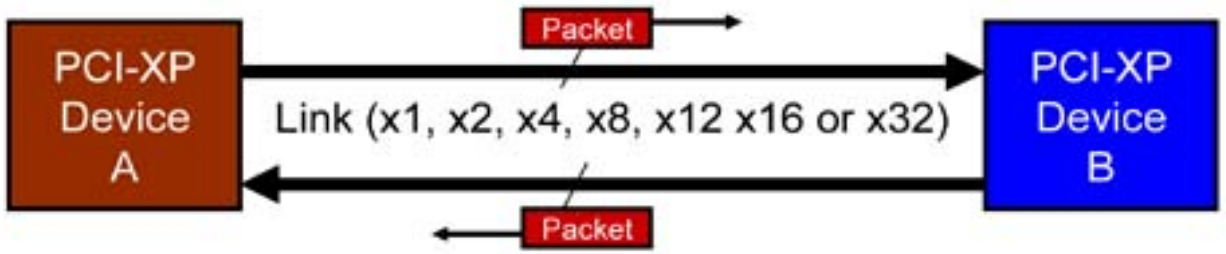
- PRF: Prefetching nem lehetséges/prefetching lehetséges
- Type: pozíció típusa
  - 00=any 32-bites cím, 01=kevesebb mint 1M, 10=bármely 64-bit cím, 11=foglalt (fenntartott)
- AD: cím dekódolása és kiterjesztése ROM inaktivált/aktivált állapotában

PCI-Express busz

Kapcsolódó segédanyag:
Addison.Wesley.PCI.Express.System.Architecture.eBook.chm

PCI-Express Busz

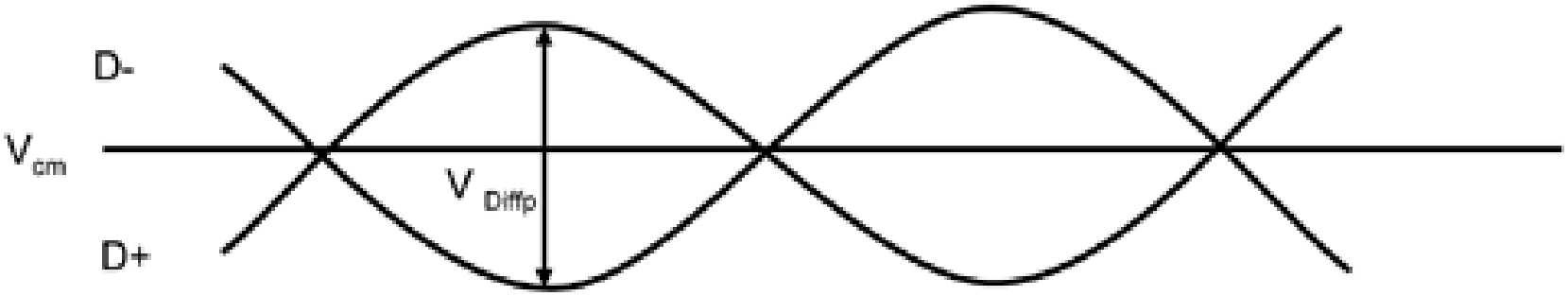
- Nagyteljesítményű, nagysebességű, P2P: pont-pont kapcsolati protokoll
- Duális simplex (egy-egyirányú) kommunikációt biztosít
  - x1, x2, x4, x8, x12, x16 or x32 link



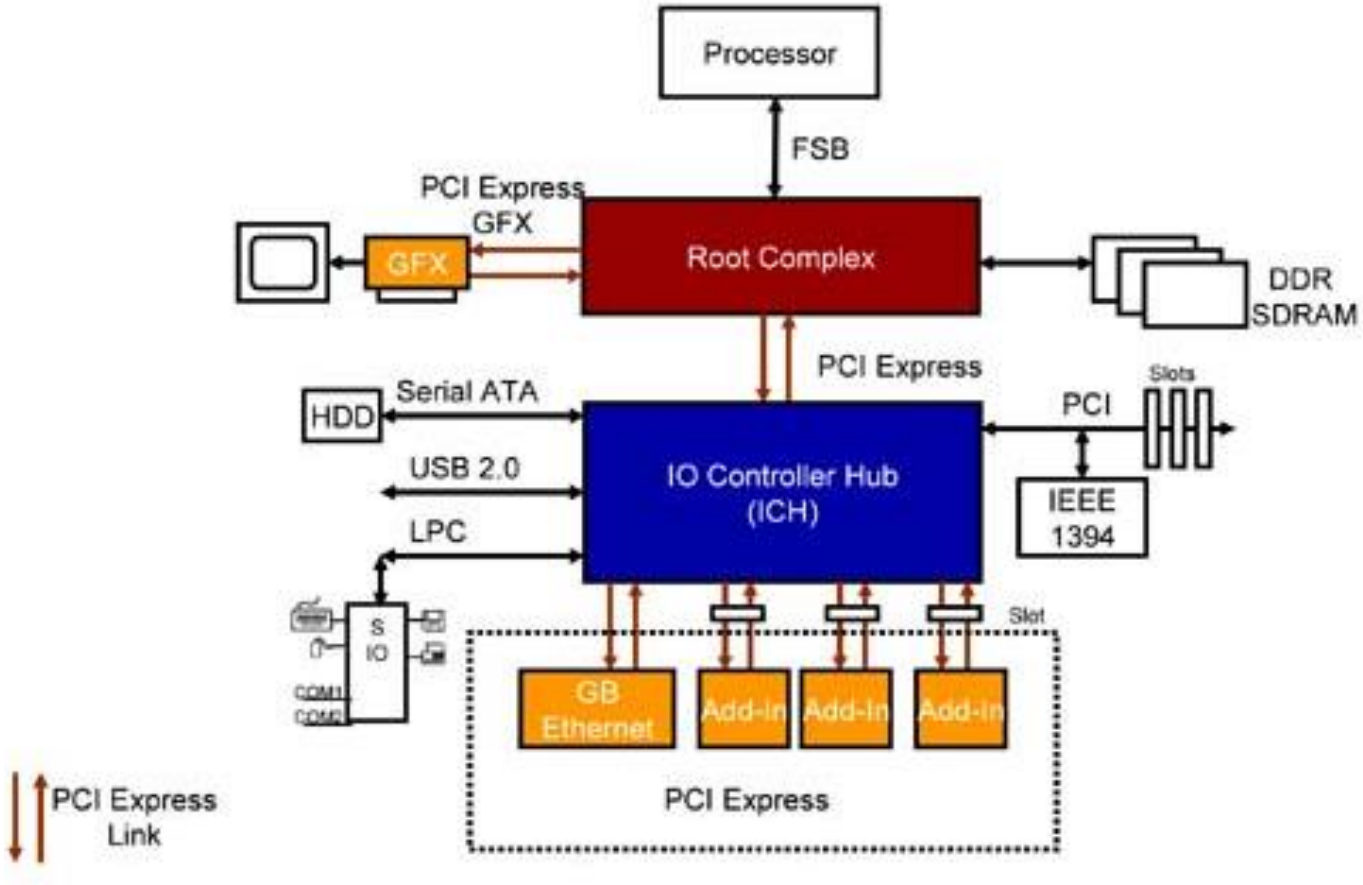
  - Lane: jelpár két irányban

Működése: Differenciális jel

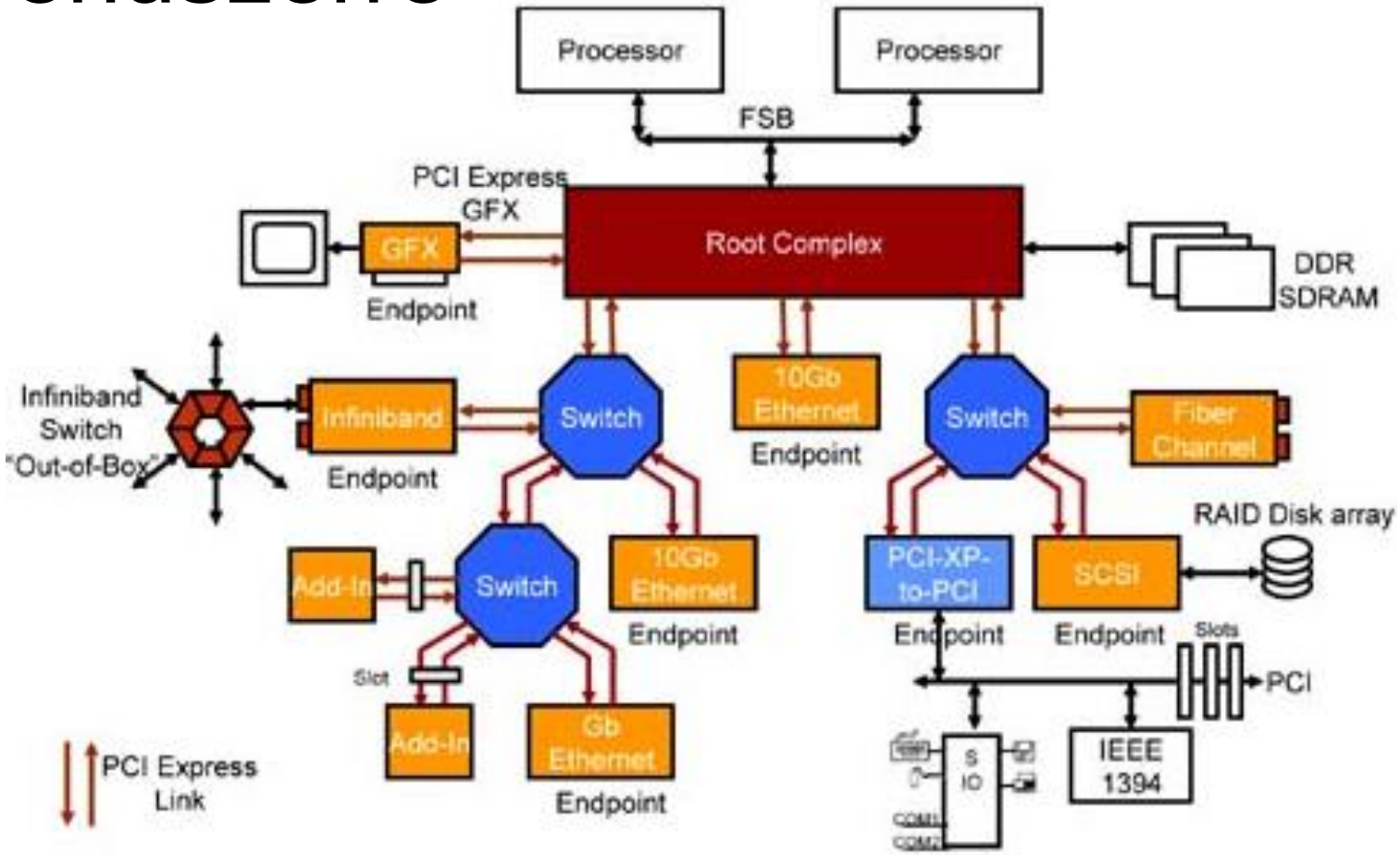
- Differenciális driver-ek és receiver-ek találhatók mindenegyes portnál.
- Ha *pozitív feszültség különbség* van a D+ és D- terminálisok között, akkor logikai ‘1’-et reprezentál.
- Ha *negatív feszültség differencia* van a D+ és D- között, akkor pedig logikai ‘0’-t mutat.
- Ha *nincs feszültség különbség*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



Olcsó asztali gépes PCI-Express rendszer



PCI-Express szerver alapú rendszerre



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

PCI-Express v1.0 tulajdonságai (2003)

- Csomag (Packet) kapcsolt protokoll
- Sávszélességek és órajelek:
  - 2.5 Gbits/sec/vonal/irány (2.5 GHz)
  - 8b/10b kódolás
  - 250 Mbytes/sec/vonal/irány
- Memória cím területek:
  - Memória
  - I/O címterület
  - Konfigurációs cím (kibővítése a PCI esetén megismert 256 Byte-ról 4 Kbyte-ra)

PCI-Express v2.0 tulajdonságai (2005. vége)

- Csomag (Packet) kapcsolt protokoll
- Sávszélességek és órajelek duplája:
  - 5 Gbits/sec/vonal/irány (5 GHz)
  - Max. elméleti határ ~10 GHz
  - Nagyobb fogyasztású (250-300W) eszközöket is támogat
  - Már külső eszközöket is támogat (10m-es kábel)
  - Input-Output Virtualization (IOV): hatékonyabbá teszi az ugyanazon hardveren futó virtuális gépek működését azáltal, hogy segíti a PCI-X-eszközök megosztását
  - Bevezetés csak 2007-körül

PCI Express tranzakciók

- Tranzakciók
  - Memória read / write
  - I/O read / write
  - Konfiguráció read / write

  } PCI-nál megismert módszerek

  - Új tranzakció típus: Üzenet (Message) alapú
- Tranzakciós modellek
  - posted (osztott kommunikáció során - nyugta)
  - non-posted (nyugta nélküli, pl. memória írás)

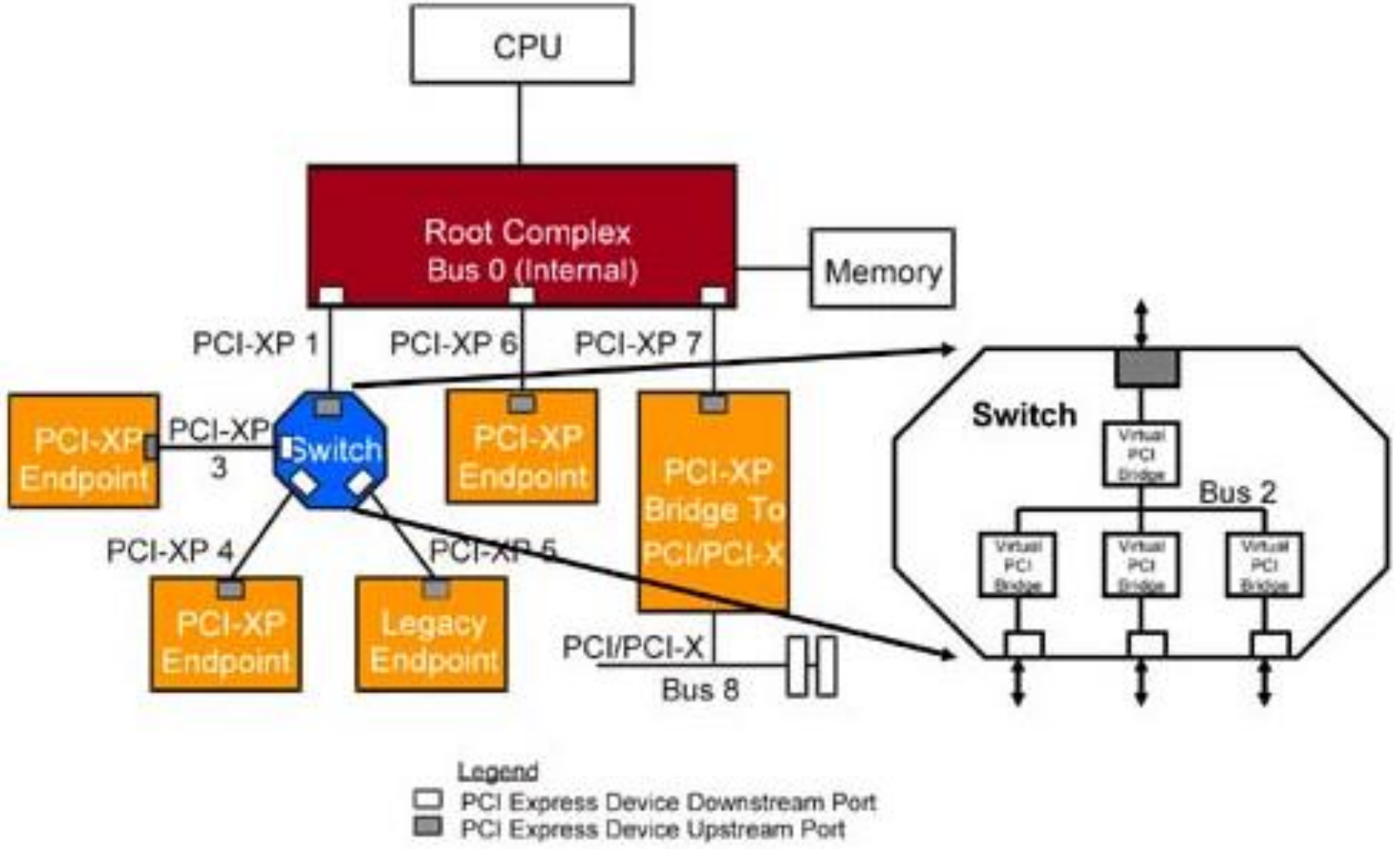
PCI Express tulajdonságai #1

- Quality of Service (QoS)
  - előre meghatározott késleltetés (latency) és sávszélesség (bandwith)
- Traffic Classes (TCs)
  - TC-k különböző (pl. sebesség) szerinti prioritás
- Virtual Channels (VCs)
  - Mindegyik Traffic Class egyedileg rendelhető hozzá egy virtuális csatornához

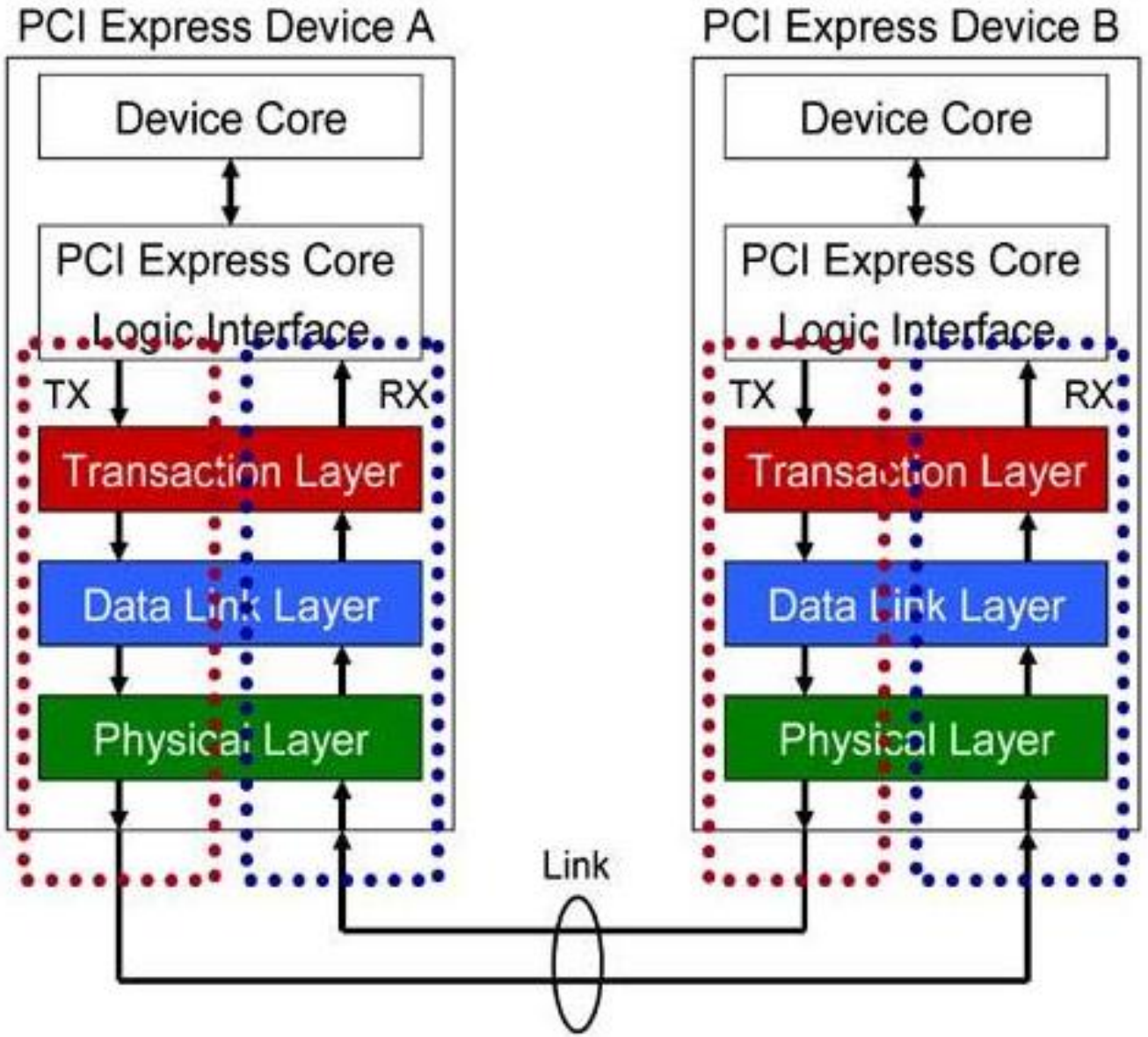
PCI Express tulajdonságai #2

- Megszakítás kezelés
  - Virtuális vezetékeken (wires)
- Power Management: energia elltátásra
  - Eszköz állapotai: D0, D1, D2, D3-Hot and D3-Cold
    - D0: teljes/ legnagyobb teljesítmény
    - D3-Cold: legkisebb teljesítmény (takarékos állapot).
  - Link / Összeköttetés állapotai: L0, L0s, L1, L2 and L3
- Hot Plug támogatás: működés közbeni eszköz csere (hibatűrő rendszereknél fontos!)
- PCI kompatibilis szoftver modell

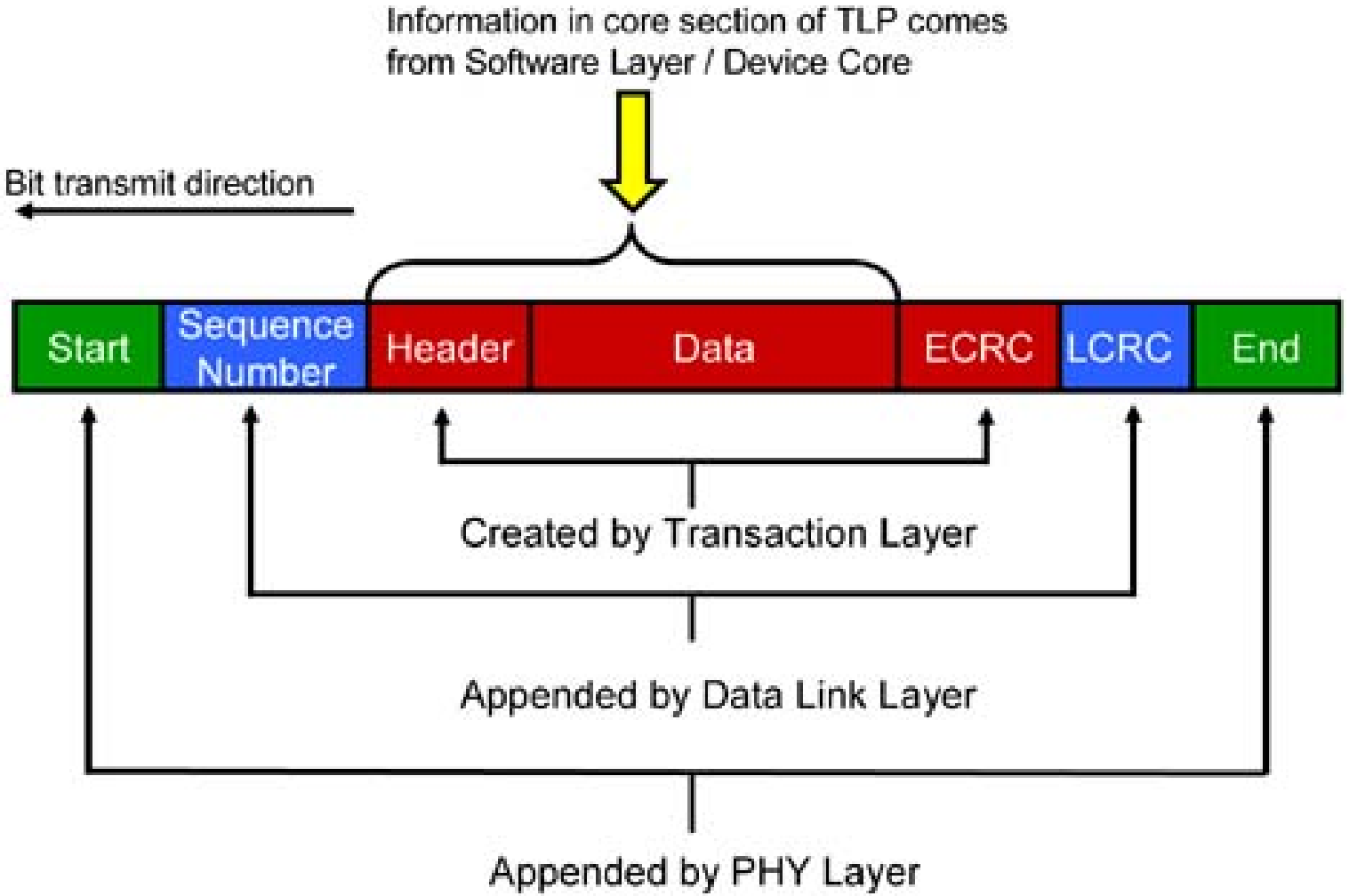
PCI Express topológia (felépítés)



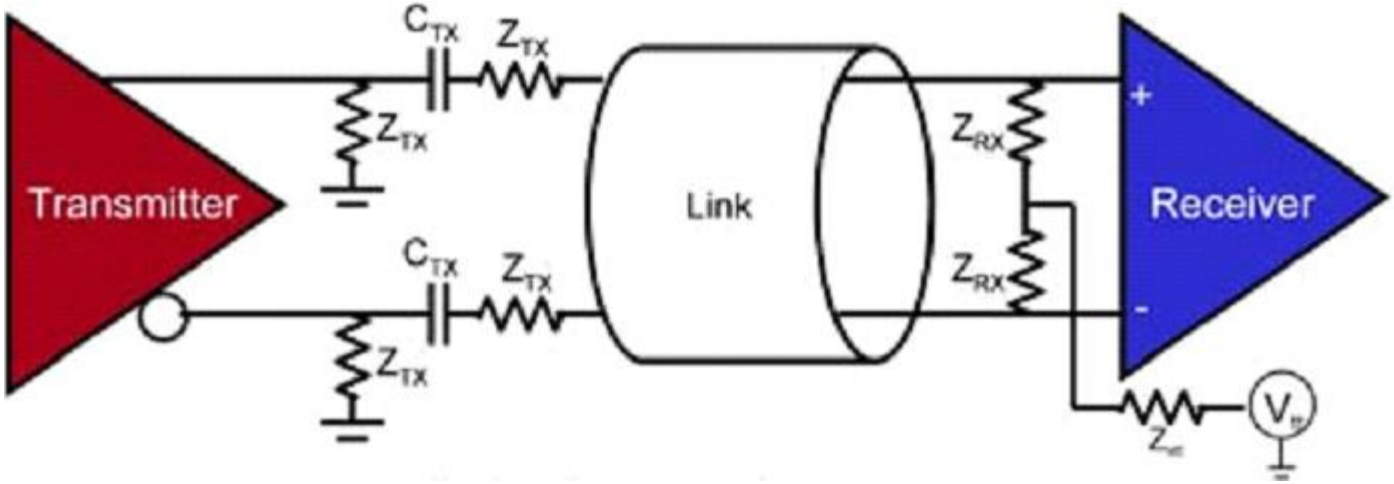
PCI Express rétegszerkezete



Tranzakciós réteg csomagjai (keret)



Elektromos Fizikai Réteg - Differenciális
Transmitter és Receiver = Port



- Port: egy PCI-X interfész, amely kapcsolatot teremt a ‘root complex’ és ‘endpoint’ között.
- Transmitter AC csatolt a receiver felé
- DC közös módú impedancia érték 50 ohm ($Z_{TX,}$, Z_{RX})
- DC Differenciális impedancia: 100 ohm
- AC Csatolt kapacitás: 75-200 nF (C_{TX})

SCSI busz

Kapcsolódó segédanyag: scsi.pdf

SCSI buszrendszer tulajdonságai

- **SCSI**= **S**mall **C**omputer **S**tandard **I**nterface: komplex, *intelligens, sínorientált eszköz* (merev-, hajlékonylemez, CD-ROM, szalagos egység, scanner...) interface. Különféle perifériák illesztésére, a processzor tehermentesítésére fejlesztették ki, az operációs rendszertől független felülettel rendelkezik. Sokoldalú eszköz, mivel nemcsak PC-s környezetbe, hanem UNIX munkaállomásokba és Apple-Mac gépekbe is integrálható.
- Tulajdonságai: A SCSI a buszok között általánosan 8 eszközt definiál (manapság max. 15 ilyen eszköz csatlakoztatható) legnagyobb sebessége 160Mbyte/s, max buszhossz 12m (UW-160). Csatlakoztatható a - típustól függően - több belső és 1 külső eszköz is.
- Az eszközöket egyetlen vezérlő a ***hostadapter*** kezeli, amely a számítógépes rendszer eszközeinek kapcsolatát építi fel a SCSI rendszerrel. Nagysebességű, párhuzamos blokkos átvitelt biztosít a processzor és perifériák között. A szabványos SCSI csatolónak 50 (v. 68) lába van, amelyből 9 vezérlő-, 9 adat-vezeték.

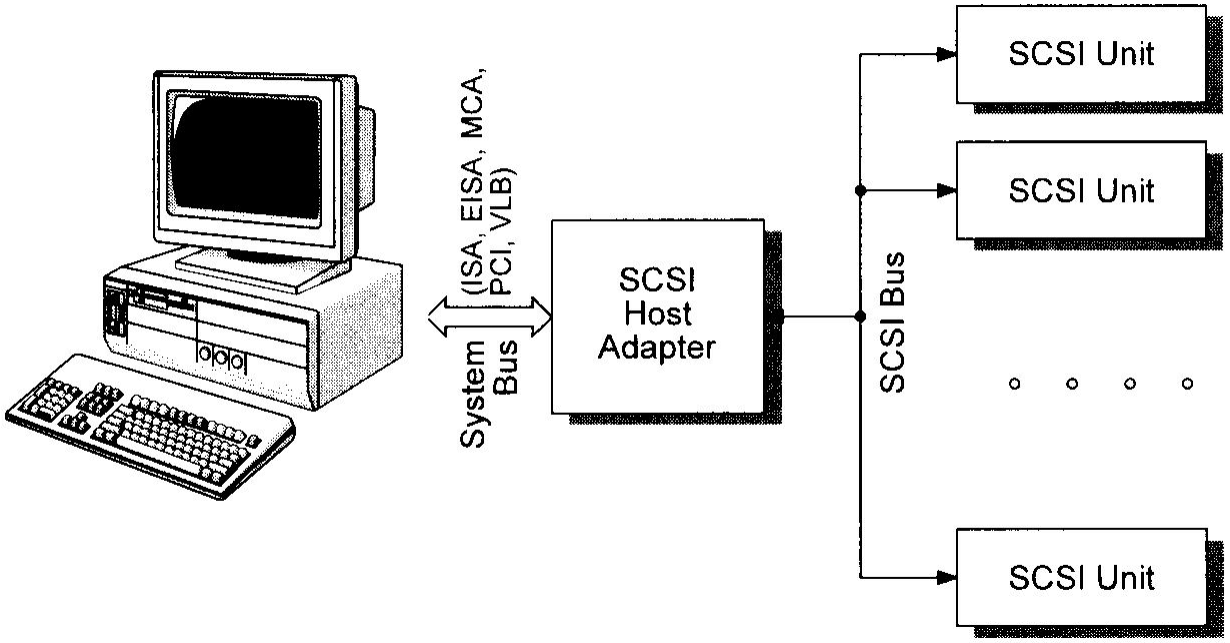
SCSI tulajdonságai (folyt. 1)

- A saját processzorral és memóriával rendelkező intelligens SCSI egységek kezdeményezőként **(*initiator*)** és fogadóként **(*target*)** is működhetnek! (lényegében egy I/O processzor eszköz)
- Maximálisan 8 initiator, és 8 target működhet 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) kell beállítani, hogy az ütközéseket a buszon elkerüljük. Az egyes SCSI szabványoknak megfelelően hostadapter/vezérlő címe mindig 0 (LSB), nem változik, a szalagos egységé pedig a 7-es (MSB), azonosítójuk SCSI-ID=0 ill. 7.

SCSI tulajdonságai (folyt. 2)

- A SCSI_ID mellett létezik a **LUN** = Logikai Egység Azonosítószám is: minden targethez hozzárendelhető további 8 logikai egység, amiket az SCSI parancsok esetén saját LUN-nal azonosíthatunk.
- A kommunikáció 8-bites adatbuszon 1-bites paritás ellenőrzés mellett zajlik. Ha a target lassú, érdemesebb magasabb prioritású szintet állítani neki. A busz elején a vezérlő hostadapter van, a másik végére pedig a lezáró ellenállást mindig az utolsó eszköznél kell tenni. Az egységeket egymás után felfűzve, egyforma (50-eres) szalagkábellel kell csatlakoztatni.
- Aszinkron / szinkron protokollt is támogat!

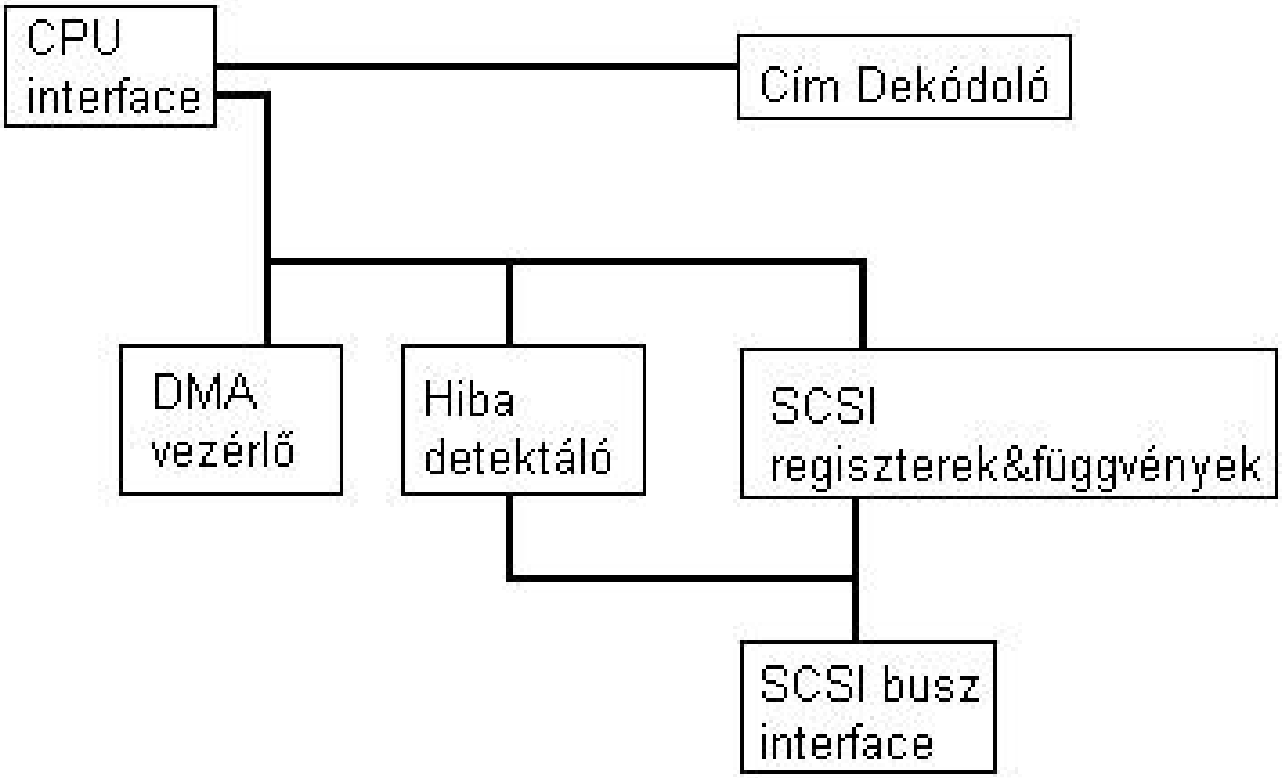
SCSI busz és PC rendszerbusz kapcsolata



- A SCSI busz egy host adapteren keresztül kapcsolódik a rendszer buszhoz

SCSI blokkdiagram

- SCSI vezérlő általános felépítése



SCSI busz 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 initiatorral 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

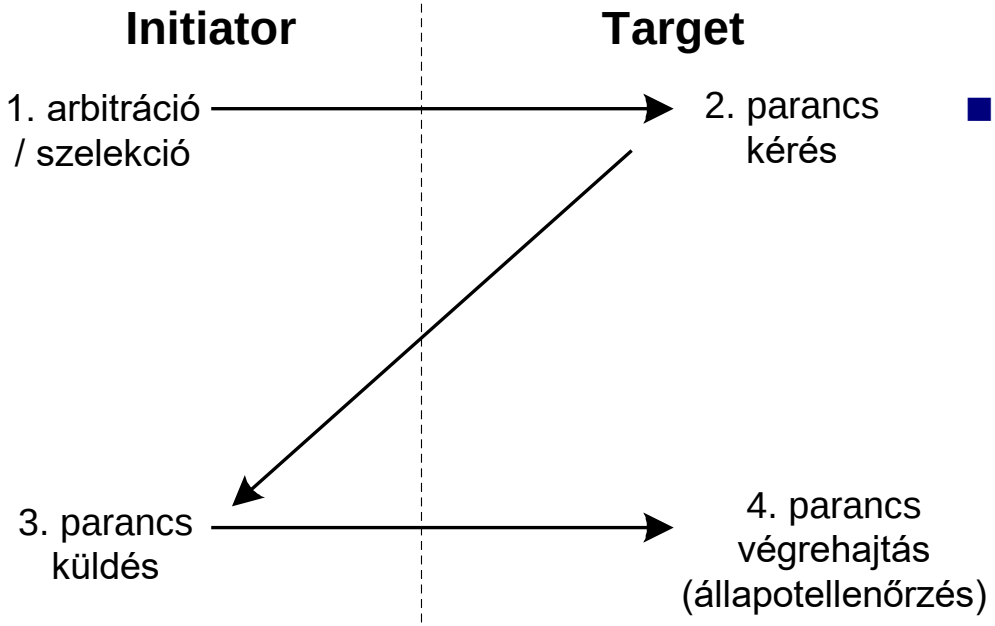
SCSI busz fázisok #1

- Bus-free (szabad?)
  - Nem használja SCSI egység a buszt, és SEL# , BSY# inaktívak
- Arbitration (döntési mechanizmus)
  - egység aktiválja a BSY# és 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 VAGY/OR kapcsolatából egy címet állít elő, melyet a buszra helyez
  - target aktiválja BSY# jelet (foglalt lesz a busz)

SCSI busz fázisok #2

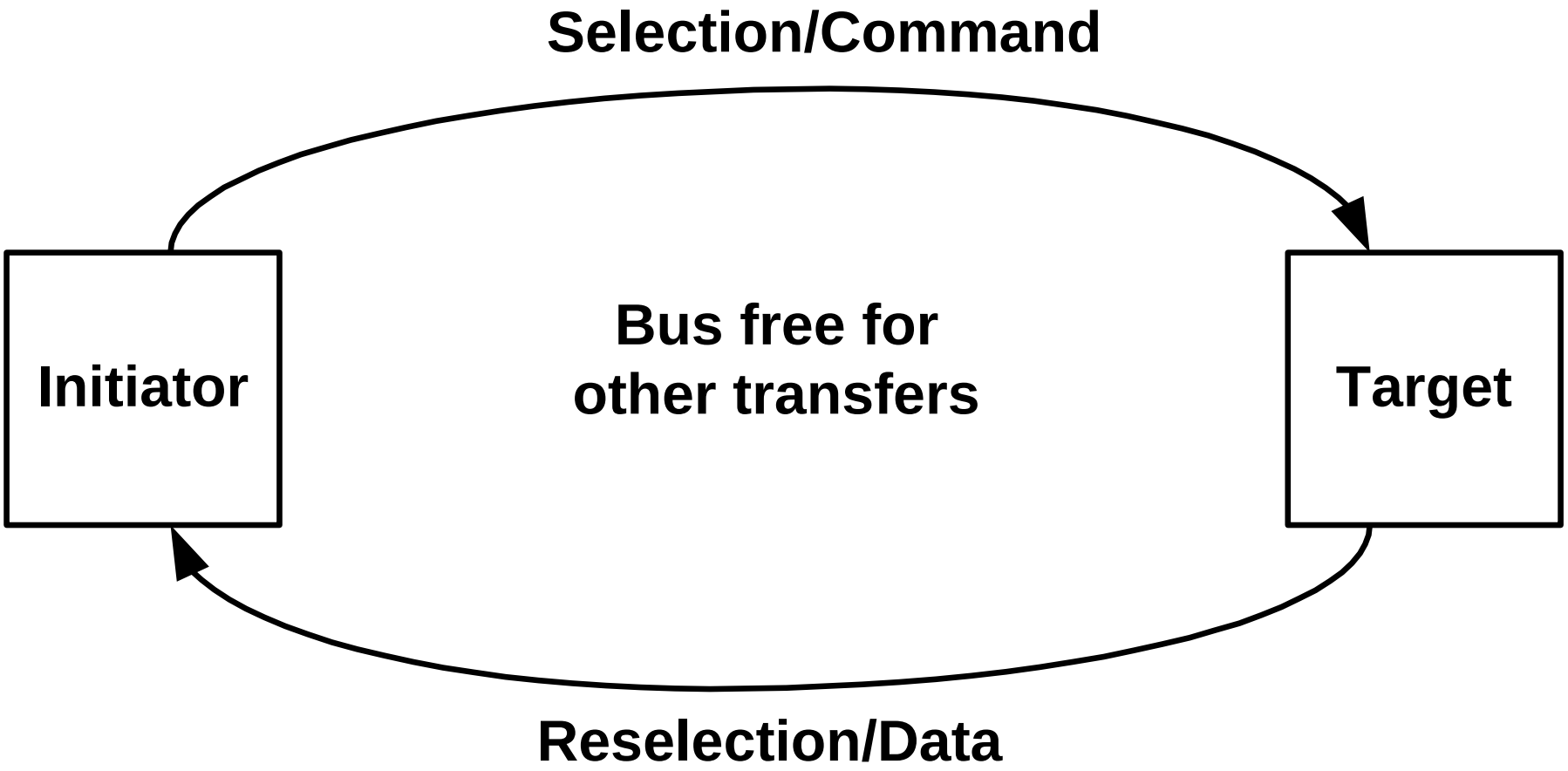
- Reselection
  - Ha szükséges, a target újra létesít kapcsolatot a targettel, miután egy megszakítás történt (folytatódhat a művelet)
- 1. Command
- 2. Data
- 3. Message
- 4. Status

SCSI – kommunikációs lépések



- (1. command → 2. data → 3. message → 4. status)
- SCSI busz fázisok a következők:
  - - Reselection (ha a target megszakítva lett, folytathatja a komm.t az initiatorral)
  - - busz szabad (bus free)?
  - - arbitráció / szelekció (ki adjon?)
  - - üzenetküldés
  - - parancs elmegy
  - - adatkapcsolat
  - - állapotellenőrzés (status)
  - - üzenetzárás, kapcsolatbontás

SCSI busz műveletek (kördiagram)



SCSI parancsok (formátum)

6-, 10-, 12- byte commands

10-byte command

Command Code		
LUN	**Reserved**	**REL**
Logical Block Address (MSB)		
Logical Block Address		
Logical Block Address		
Logical Block Address (LSB)		
Reserved		
Transfer/Allocation/Parameter Length		
Transfer/Allocation/Parameter Length		
Control Byte		

- LUN: Logikai Egység Szám
- REL: relatív cím
- Blokk címzés

SCSI busz 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 v. 32 bites szinkron busz; 20Mbyte/s; 10Mhz; 15 egység; 68 pólusú csatlakozó
- *Ultra Wide SCSI*: 16 v 32 bites; 40Mbyte/s; 20 Mhz; max 15 egység; 68 pólusú csat.
- *Ultra2 Wide SCSI:* 16 v 32 bites; 80Mbyte/s; 40Mhz; max 15 egység; 68 pólusú; max kábelhossz. 12m
- *Ultra-160 SCSI:* 16 v 32 bites; 160Mbyte/s; 80Mhz; max 15 egység; 68 pólusú; 12m (differenciális jellel működik)
- *Ultra-320 SCSI:* 16 v 32 bites; 320Mbyte/s; max 15 egység; 68 pólusú; 12m (differenciális jellel működik)
- Soros
- Optikai

Pannon Egyetem
Képfeldolgozás és Neuroszámítógépek Tanszék



Digitális Rendszerek és Számítógép Architektúrák

9. előadás: DSP (Digital Signal Processing)

Előadó: Vörösházi Zsolt
Szolgay Péter

Jegyzetek, segédanyagok:

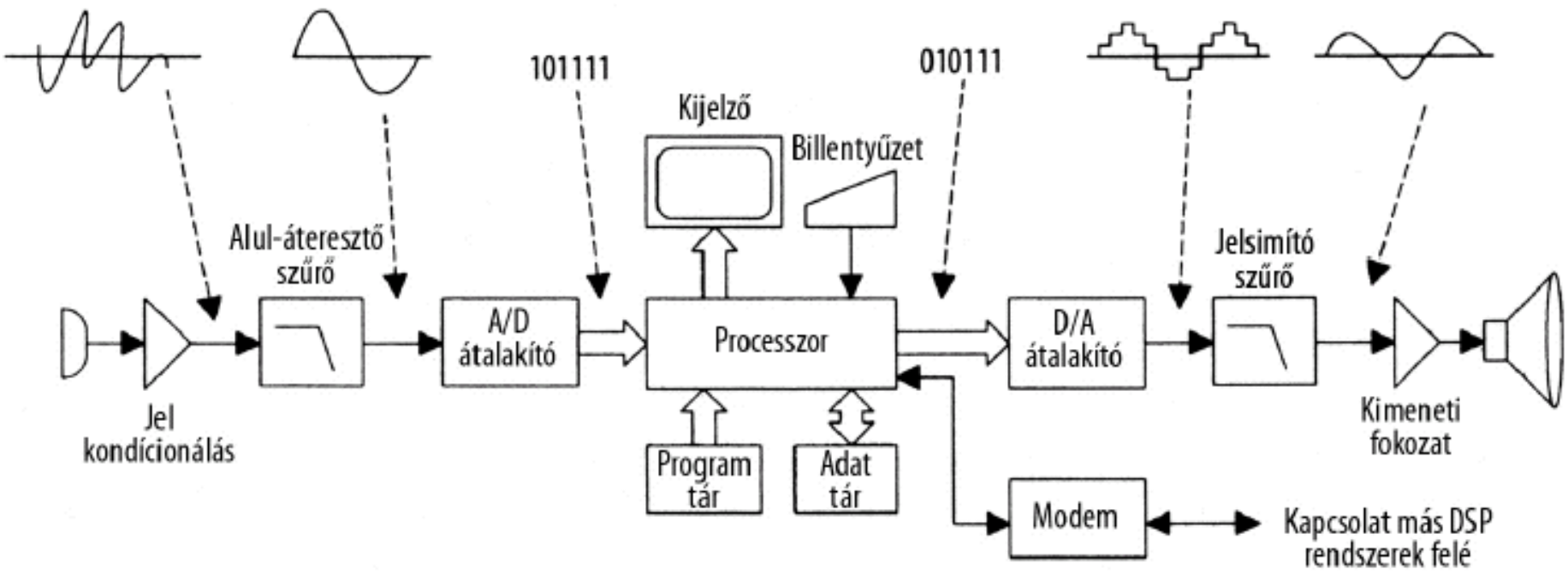
- Könyvfejezetek:
  - http://www.knt.vein.hu
    -> Oktatás -> Tantárgyak -> Digitális Rendszerek és Számítógép Architektúrák (nappali) -> Kiegészítések -> DSP
    (tms320Cx.pdf)
- Fóliák, óravázlatok .ppt (.pdf)
- Feltöltésük folyamatosan

Általános DSP rendszermodell - alapok

Jelek - Jelfeldolgozás

- Digitális jelfeldolgozás – Digital Signal Processing (DSP)
  - Analóg jelek
  - Digitális jelek
    - Hogyan ábrázolják ezeket a jelértékeket a fizikai hardver eszközök? (hardver-tervező az implementáció során saját maga szabhatja meg, hogy mit-mivel jelöl).

Általános DSP rendszermodell



* forrás: DSP: Digitális Jelfeldolgozás jegyzet
Dr. Fodor Dénes, Tóth Roland (Pannon Egyetem)

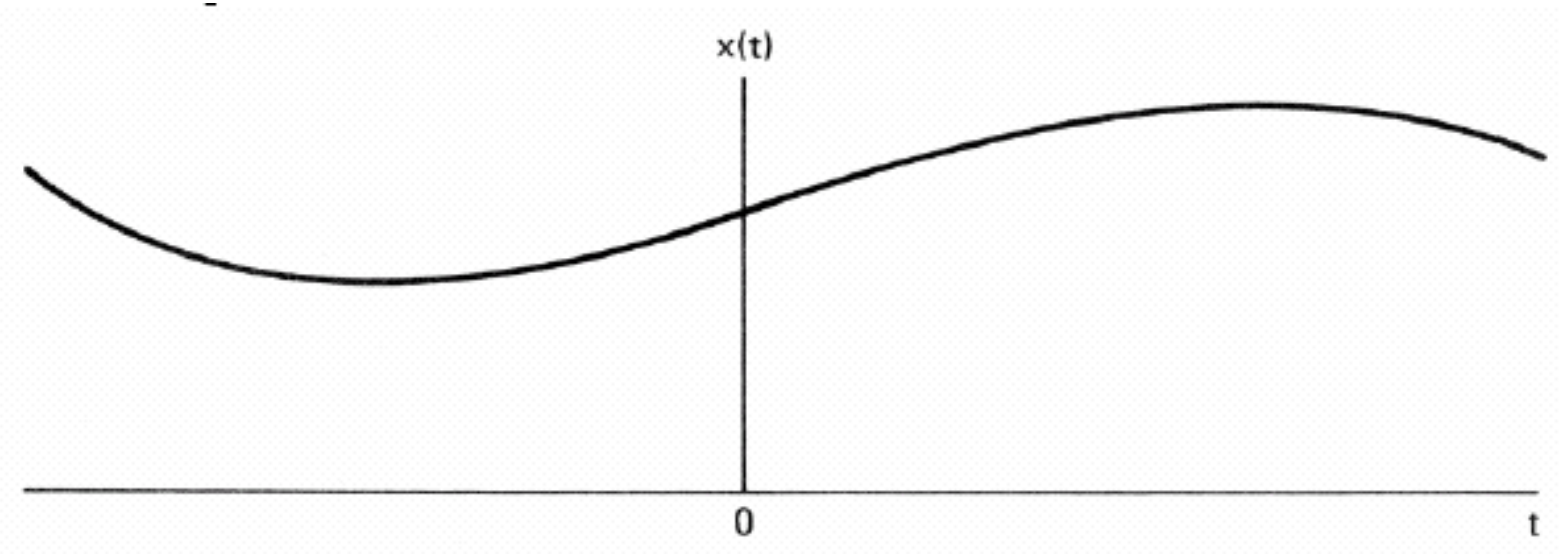
Általános rendszermodell (folyt.)

A DSP rendszermodell által végzett operációk a következő csoportokba sorolhatóak:

- Analóg jel, jelek fogadása az input csatornán.
- Ezen analóg jelek számokkal való ábrázolása, digitalizálása.
- Bizonyos, a funkciót jelentő számítások elvégzése az így kapott értékhalmazon.
- A számok visszakonvertálása analóg jelekké.
- Közben az információ feldolgozásra kerül amit megjeleníthetünk, tárolhatunk, vagy továbbíthatunk.

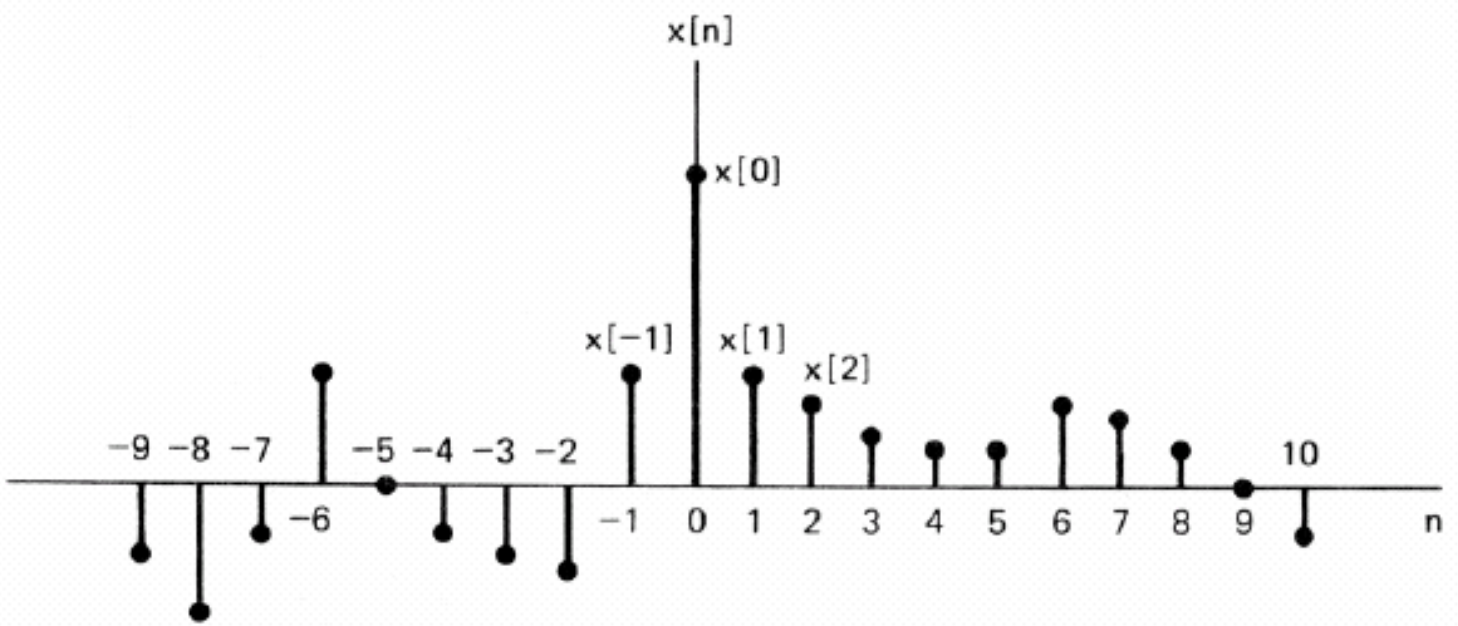
a.) Folytonos jelek

- **Def: Folytonos (analóg) változójú jelek:** Olyan jelek amelyeknek minden független változója folytonos. A mi vizsgálataink szerint az ilyen csupán az időtől folytonosan függő jeleket folytonos idejű jeleknek (Continuous Time Signals) azaz **CT** (jelölés) nevezzük. A folytonos idejű jelek két időpont között végtelen sok értékre vannak definiálva, ezért bármely időpontra felvesznek értéket.

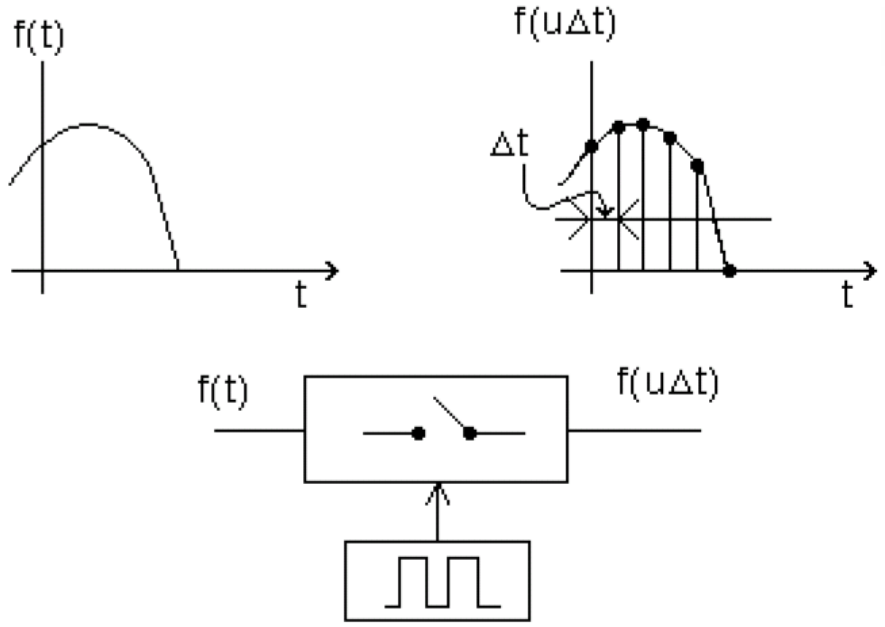


b.) Digitális jelek

- **Def: Diszkrét változójú jelek.** Olyan jelek amelyeknek minden független változója diszkrét értékű. A mi vizsgálataink szerint az ilyen csupán az időtől diszkréten függő jeleket diszkrét idejű jeleknek (Discrete Time Signals) azaz **DT** (jelölés) nevezzük. A diszkrét idejű jelek csak diszkrét időpontokra vannak definiálva, ezért mindig csak meghatározott időközökben vesznek fel értékeket, és az időközök között nem definiáltak.



Mintavételezés csoportosítása:

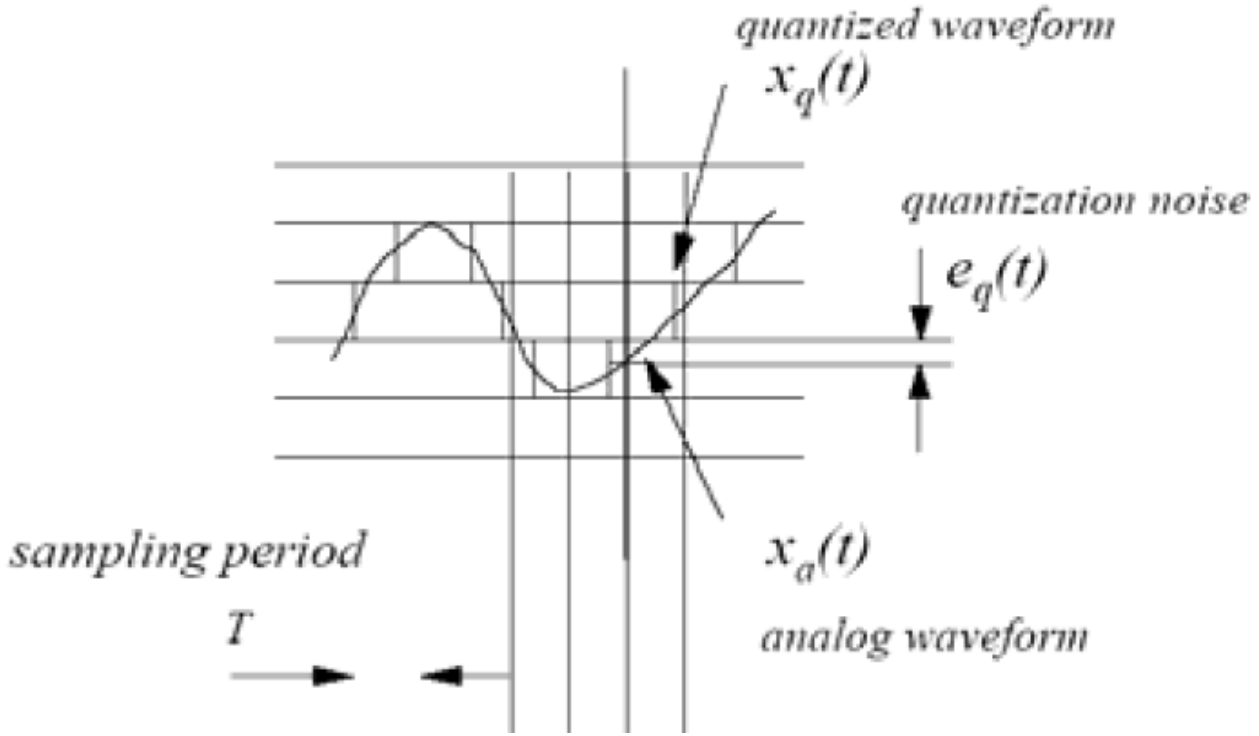


- A **jeltől független** mintavételezés, **ekvidisztáns** időközönként. Ezeket lineáris, rögzített lefolyású mintavevő rendszereknek nevezzük.
- A **jeltől függő** mintavételezés, amikor a változás sebességének növekedése pontosabb ábrázolást igényel, de gazdasági okokból a mintavételezés gyakoriságát valamilyen jellemzőnek a változásához kötjük. Ezeket ***nem lineáris***, jeltől függő mintavevő rendszereknek nevezzük.
- **Statisztikai** mintavételezés, általában a manuális mintavételezés tartozik ide.

Folyt. vs Diszkrét jelek

- Folyt (A) → diszkrét (D):
  - mintavételezéssel (sampling) és
  - kvantálással (quantization)
- Ennek következtében „aliasing” (gyors jelváltozásból) és „kvantálási zaj” léphet fel! Ezek a hibák kezelhetők, szabályozhatók megfelelően tervezett szűrők segítségével: quantizer, anti-aliasing filter stb.

Kvantálási zaj



$$x_q(t) = x_a(t) + e_q(t)$$

Jelfeldolgozó processzor (DSP) architektúrák

DSP történeti fejlődése

- Első generációs DSP chip-ek: Intel microcontroller, **TI**, AT&T, Motorola, Analog Devices (1980-as évek elején)
- Low-cost DSP-k (1980-as évek vége)
- Vo-Coder (voice-operated coder) Standards: polgári alkalmazásban (80'-as évek vége)
- DSP technológia migrációja az általános célú mikovezérlőkkel / CPU-kkal (90'-es évek)
- High-complexity (Rich Media Applicatons)
- Low-power (portable) applications

DSP feladatkörök (Application Spectrum)

- Beszéd- és audió-tömörítés
- Filtering (szűrés)
- Moduláció, demoduláció (MODEM)
- Hiba-javítás, -kódolás, -dekódolás
- Audió feldolgozás (pl. háttérzaj, zajcsökkentés, kiegyenlítés, visszhang kioltás stb.)
- Beszéd felismerés
- Jel szintézis (pl. zene-, beszéd-szintézis stb.)
- 2D-3D video processing
- Mobil kommunikáció (GPS)
- Automotive systems (ECU, Szervó vezérlés, ABS stb.)
- Military (missile guidance)

DSP-n implementálható algoritmusok

- FFT (Fast Fourier Transformation)
- Discrete Cosinus Transformation (MPEG)
- Kódolási eljárások (Huffman, Trellis, Runlength ...)
- Szűrők (FIR, IIR, Kalman, Notch ...)
- Vektor műveletek (Dot Products, Cross Product ...)
- Mátrix műveletek
- Konvolúciók
- Numerikus integrálások, deriválások és egyéb algoritmusok

Miért éppen a DSP?

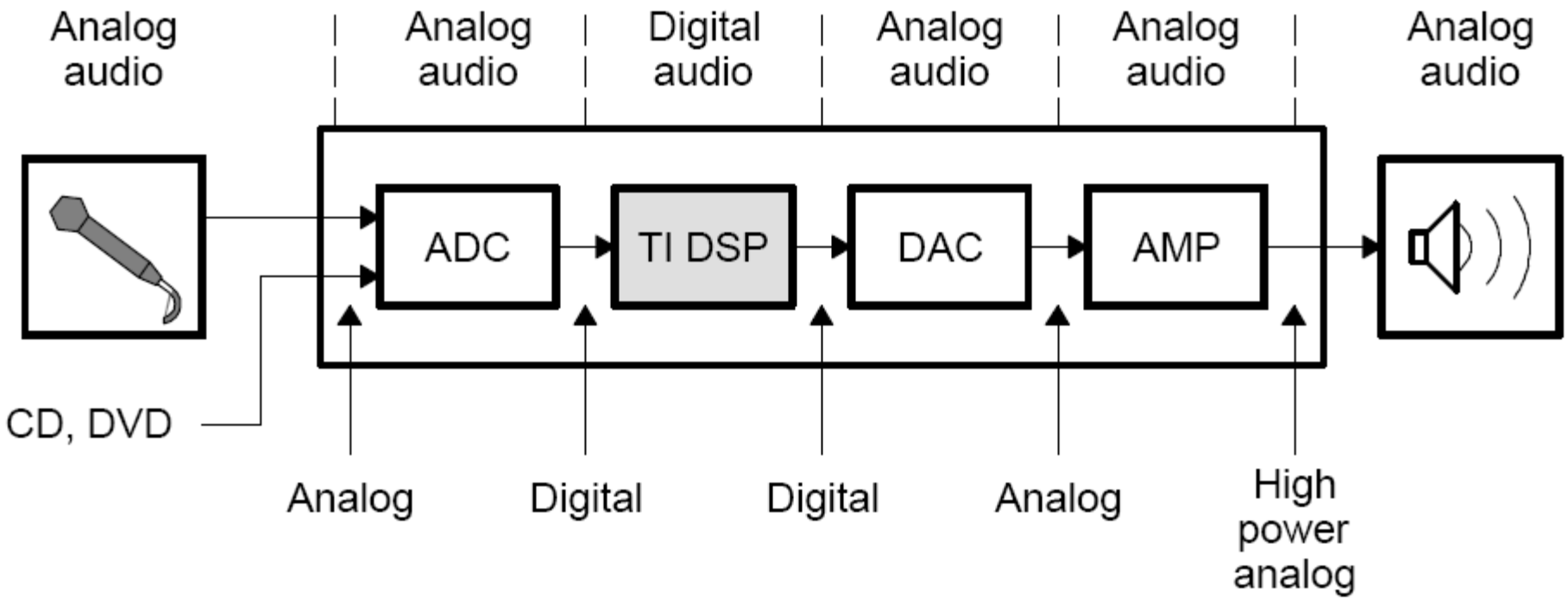
- Optimalizált architektúrák:
  - Utasításkészlete a jelfeldolgozáshoz igazított
  - Minimalizálja a numerikus problémák megoldását diszkrét, digitális jelek feldolgozásával
- Nagy teljesítmény (5 MIPS – 60 MFLOPS)
  - Komplex algoritmusok valós idejű implementálása
  - Nagy mintavételi arány (sampling rates)
  - Minimális számítási késleltetés
- Rendszer-vezérlés támogatása
  - Utasításkészlet (HLL támogatás: szimuláció, debugger stb.)
  - Dedikált perifériák I/O bit-manipulációhoz
  - Gyors környezet (context) kapcsolás (shadow regiszterekkel)
- Előnyök (digitális rendszer)
  - Megbízhatóság
  - Flexibilitás
  - Time sharing / task switching
  - Környezeti behatásoktól való függetlenség
  - Nagy sávszélesség és felbontás

Mit kell tudnia egy DSP processzornak a hatékonyság érdekében?

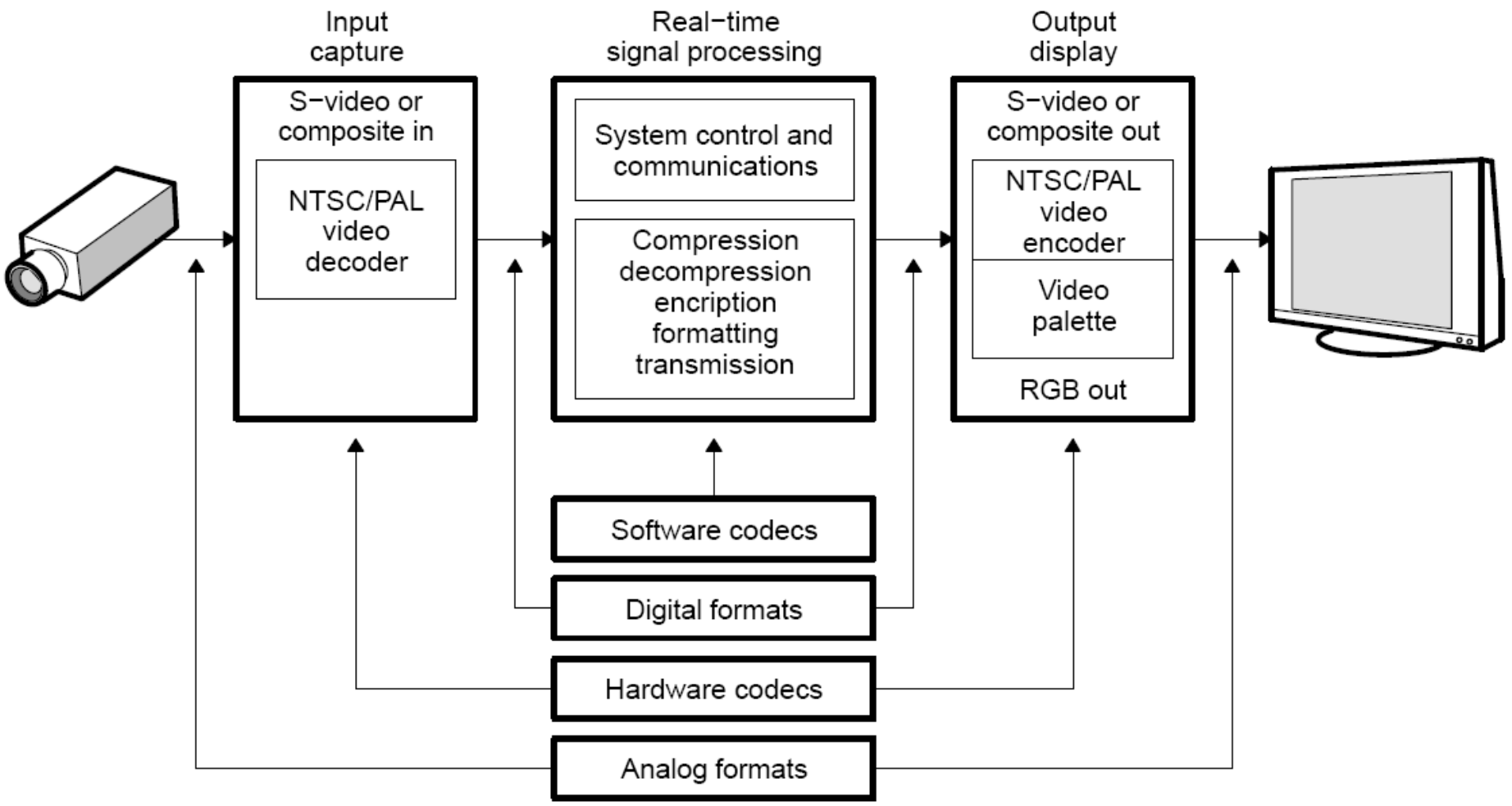
- Hatékonyság:
  - Ismétlődő numerikus számítások
  - Numerikus pontosság kezelése
  - Nagy memória sávszélesség (ált. tömb lekérdezések)
  - Real-time feldolgozás

- Minimális:
  - Költség (ár)
  - Disszipált teljesítmény
  - Memória használat
  - Fejlesztési idő

Példa: DSP-s audió rendszer

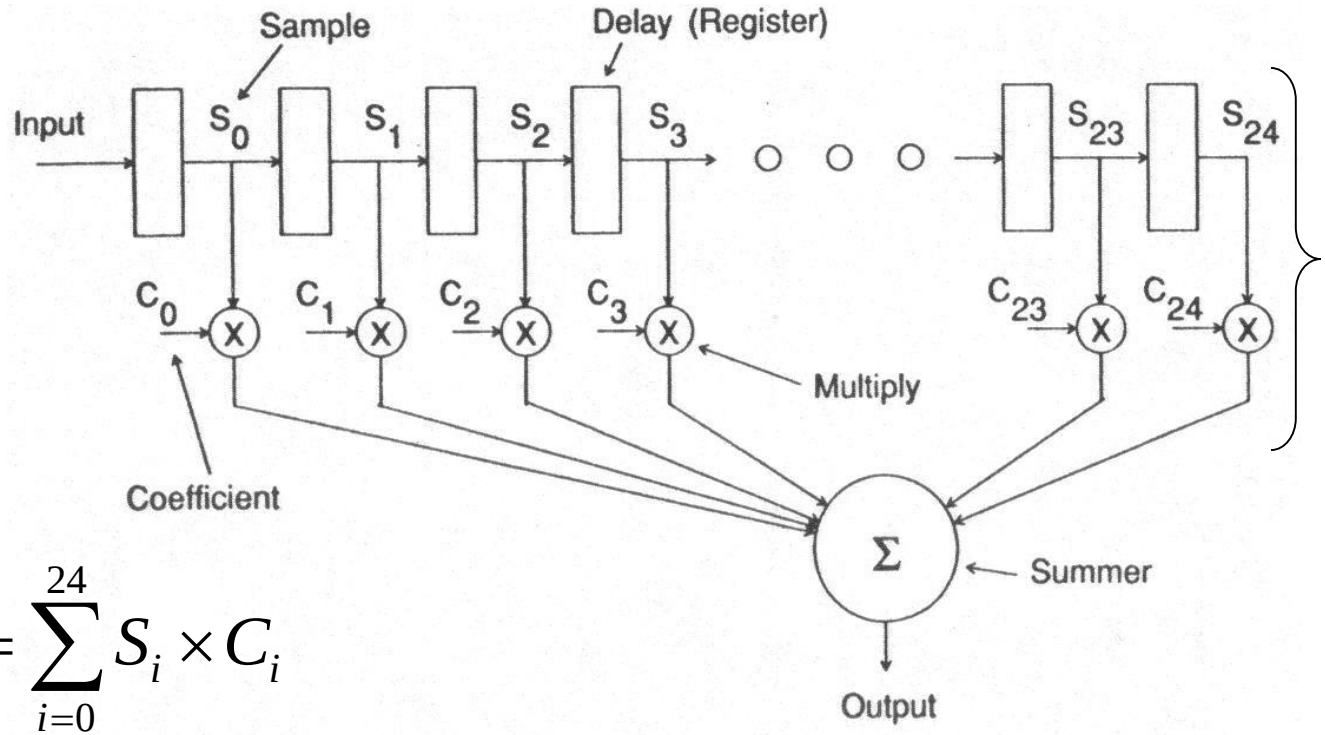


Példa: DSP-s videó feldolgozó és megjelenítő rendszer



Motiváció: FIR szűrő

- Mindenegyes elágazásban („tap”= clk összz. n):
  - Két adatfeltöltés (fetch)
  - Szorzás
  - Összeadás



MAC:

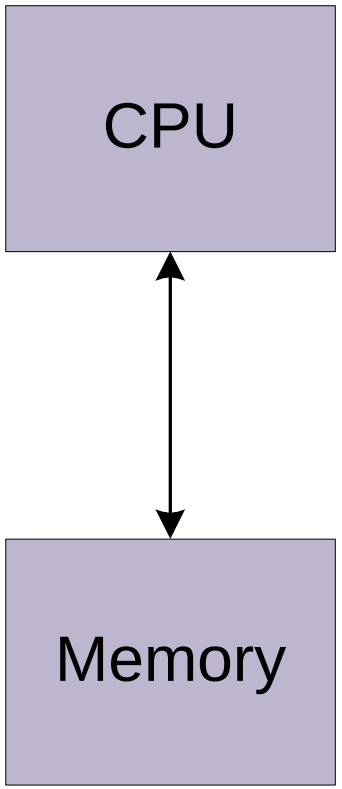
Multiply & Accumulate

(1 clk)

$$output = \sum_{i=0}^{24} S_i \times C_i$$

Probléma: FIR filter Neumann architektúrán

```
:loop
        mov     *r0, x0
        mov     *r1, y0
        mpy     x0, y0, a
        add     a, b
        mov     y0, *r2
        inc     r0
        inc     r1
        inc     r2
        dec     ctr
        tst     ctr
        jnz     loop
```



Probléma:

- memória busz / sávszélesség szűk keresztmetszete
- vezérlési kód hosszúsága

Texas Instruments TMS320C1x/2x/3x/4x/5x – processzor család

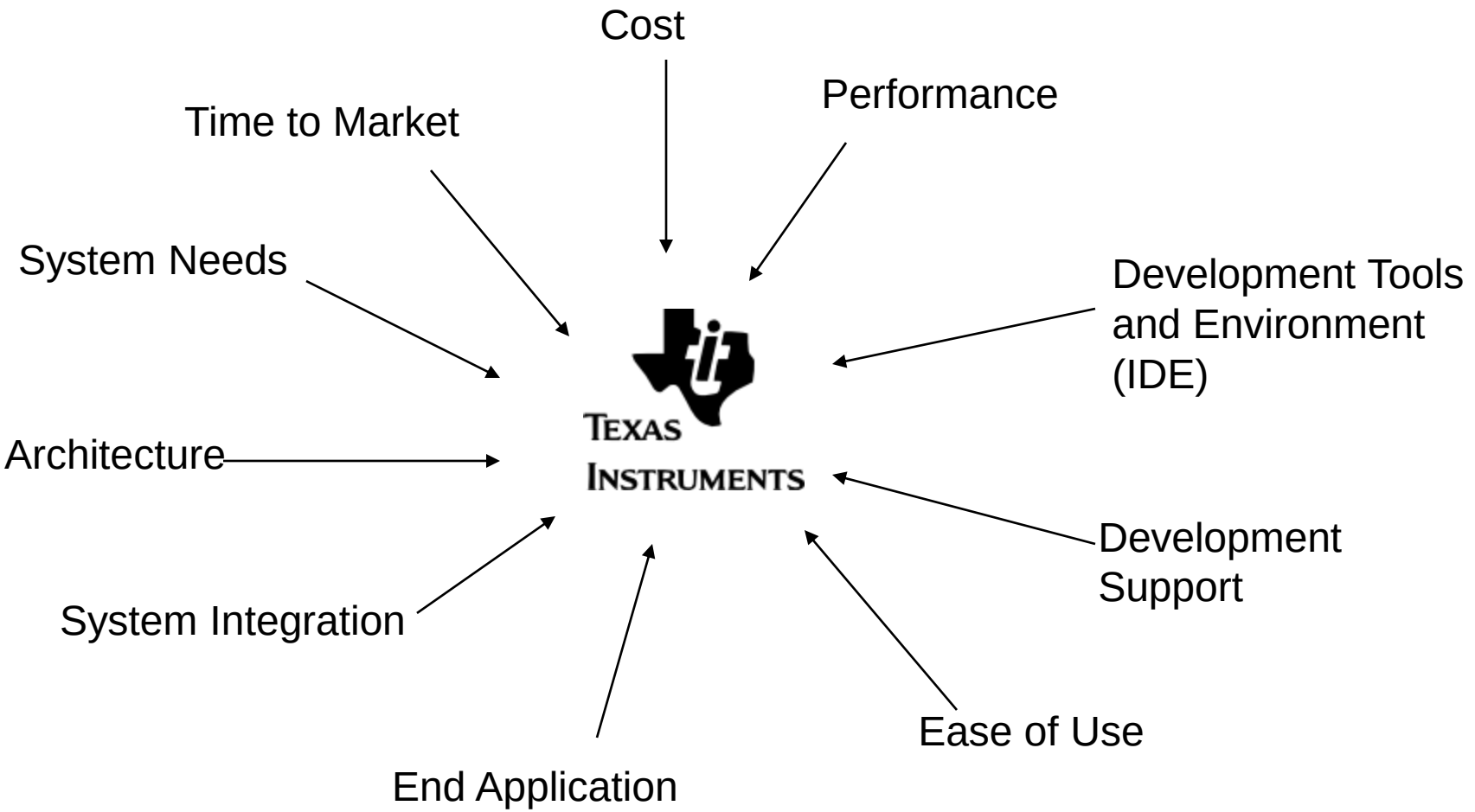


www.ti.com

Texas Instruments: Product Areas

- Amplifiers & Linear
- Clocks & Timers
- **DSP-Digital Signal Processing**
- Data Converters
- Interface
- Logic
- Microcontrollers
- Power Management
- RF/IF Components
- TI-RFid™

Milyen tulajdonságok szerint válasszuk ki a megfelelő DSP-t?

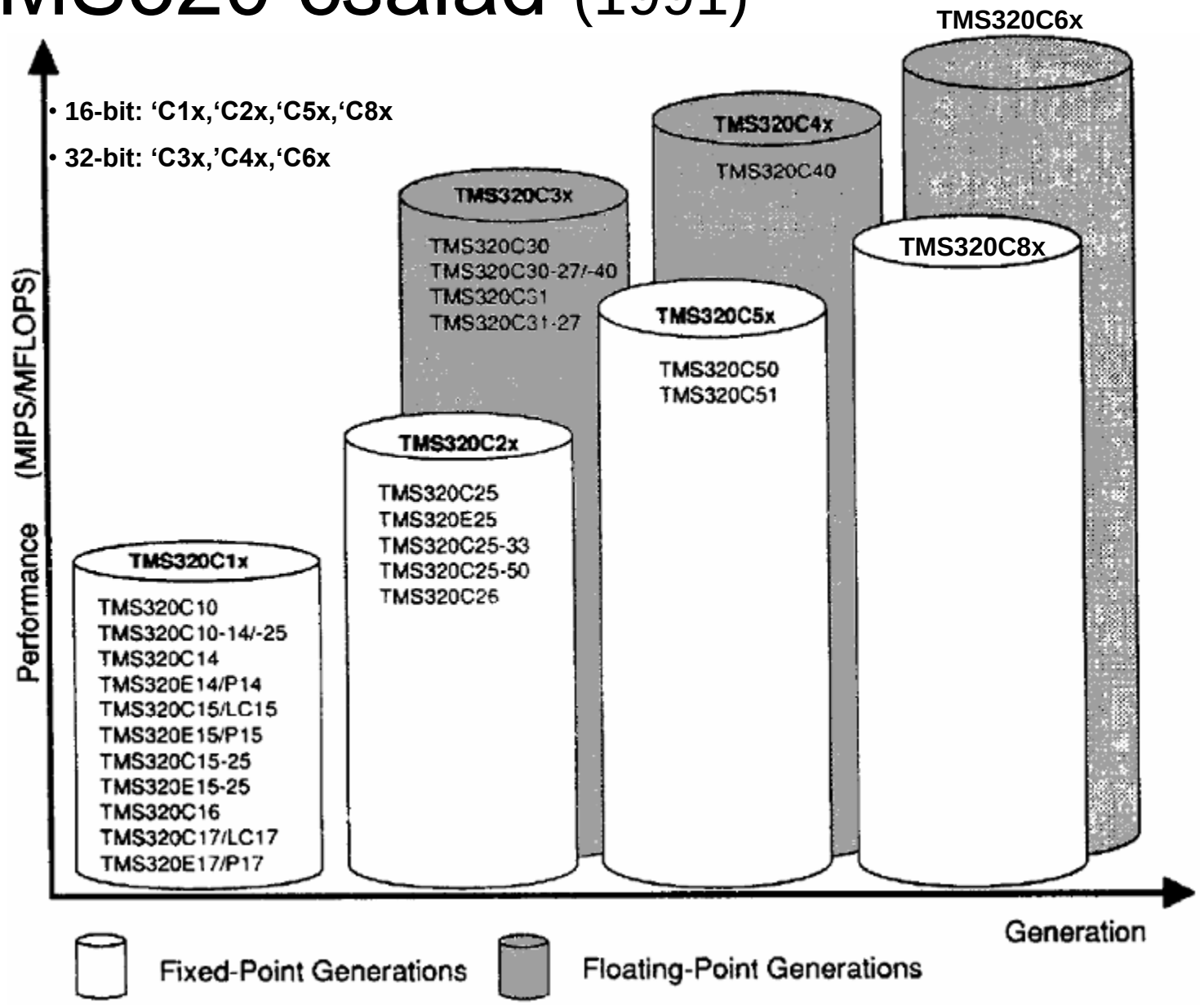


http://www-s.ti.com/sc/techlit/ssdv004 (Selection Guide)

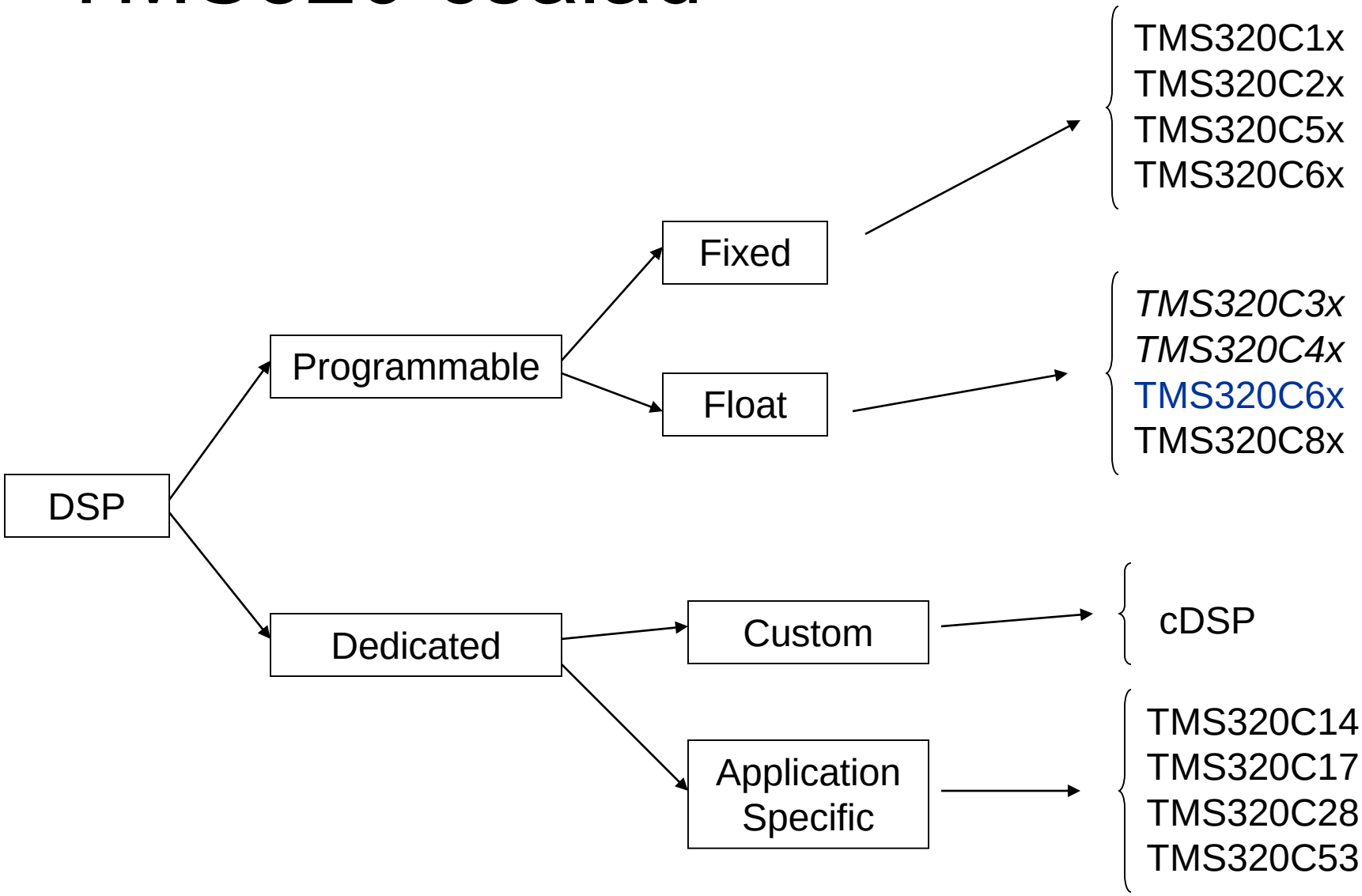
TMS320 Architectural Features

Feature	Benefit
Single-cycle instructions	High-sampling retes/control of high bandwith systems
DSP instruction set	Real-time execution of advanced control algorithms
Multiple buses	Simultaneous access of data and instructions
Hardware multiplier	Minimize computational delays
Hardware Shifters/Floating point	Fast scaling/larger dynamic range
16/32 bit wordlength	Minimize quantization errors
32/40 bit arithmetic registers	Minimize truncation errors
Hardware Stack/ Shadow Registers	Fast interrupt processing
Saturation Mode	Prevent wrap around of accumulator
Parallel Logic Unit	Fast bit-manipulation

TMS320 család (1991)



TMS320 család



TMS320Cxx család (2006)

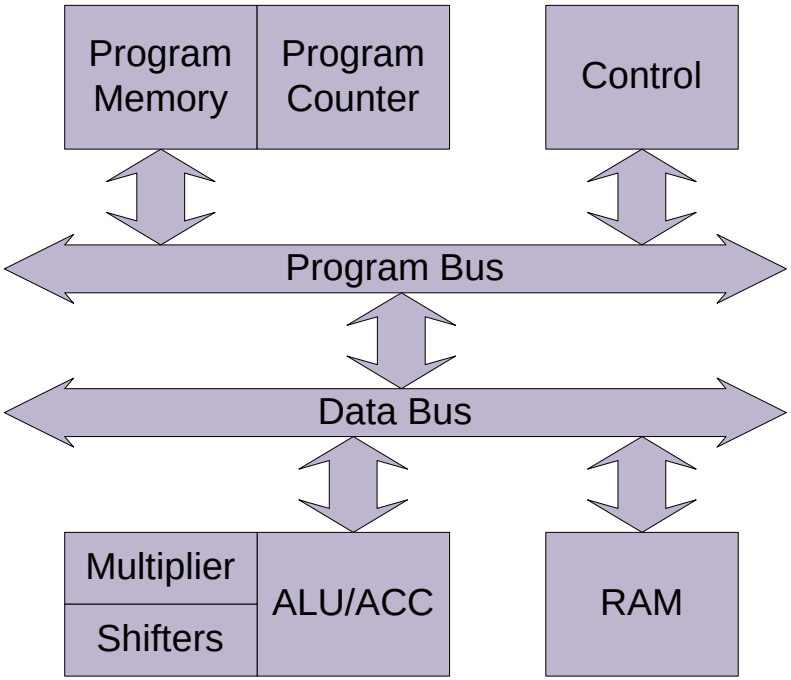
Típus	Adat	Jellemzők	Státusz
TMS320C1x	Fix-pontos	Elsőgenerációs DSP	Csak military
TMS320C2x	Fix-pontos	Első nagy szériás DSP	Élő/ népszerű
TMS320C3x	Lebegő-pontos	Első lebegőpontos DSP, DMA	Élő/ népszerű
TMS320C4x	Lebegő-pontos	4 db kommunikációs port, másik DSP felé, DSP tömb építése, DMA	Kifutó / drága
TMS320C5x	Fix-pontos	Első statikus CMOS design: low power	Élő
TMS320C6x	Fix/Lebegő pontos	Mai legmodernebb DSP-k. Nagyon széles utasítás szó (256 bit).	Élő
TMS320C8x	Lebegő-pontos	Első multimédiás 4 db TMS320C4x + RISC processzor (nem jött be, túl drága volt) – többmagos egység!	Csak military

DSP Product Tree (2007. május)

- **TMS320™ DSP Platforms** (165)
- **DaVinci™ Digital Media Processors** (21)
  - TMS320DM64x DSPs (8)
  - TMS320DM643x DSPs (10)
  - TMS320DM644x DSPs (3)
- **C6000™ High Performance DSPs** (27)
  - TMS320C6414T/15T/16T DSPs (21)
  - TMS320C645x DSPs (6)
- **C6000™ Performance Value DSPs** (23)
  - TMS320C6410/12/13/18 DSPs (8)
  - TMS320C642x DSPs (6)
  - TMS320C62x DSPs (9)
- **C6000™ Floating-point DSPs** (27)
  - TMS320C672x DSPs (17)
  - TMS320C67x DSPs (10)
- **C5000™ Power-Efficient DSPs** (36)
  - TMS320C54x DSPs (25)
  - TMS320C55x DSPs (11)
- **C2000™ High Performance 32-bit Controllers** (29)
  - TMS320F281x Controllers (8)
  - TMS320F280x Controllers (12)
  - TMS320LF240x Controllers (9)
- **Other TMS320™ DSPs** (2)
  - TMS320C33 DSPs (2)

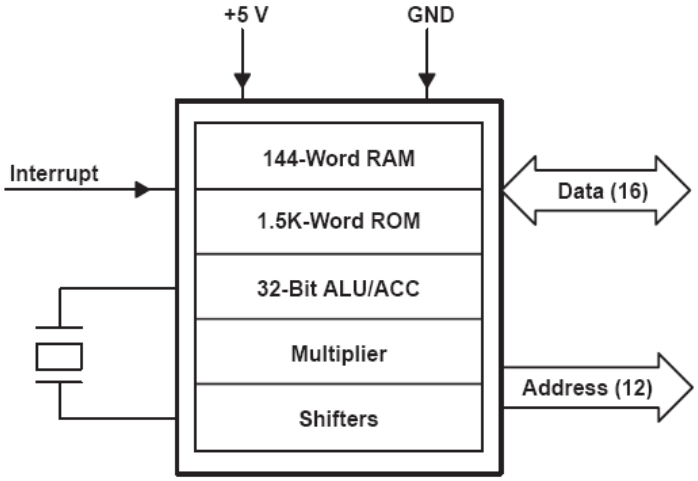
TMS320C1x High-Speed Architecture (1982)

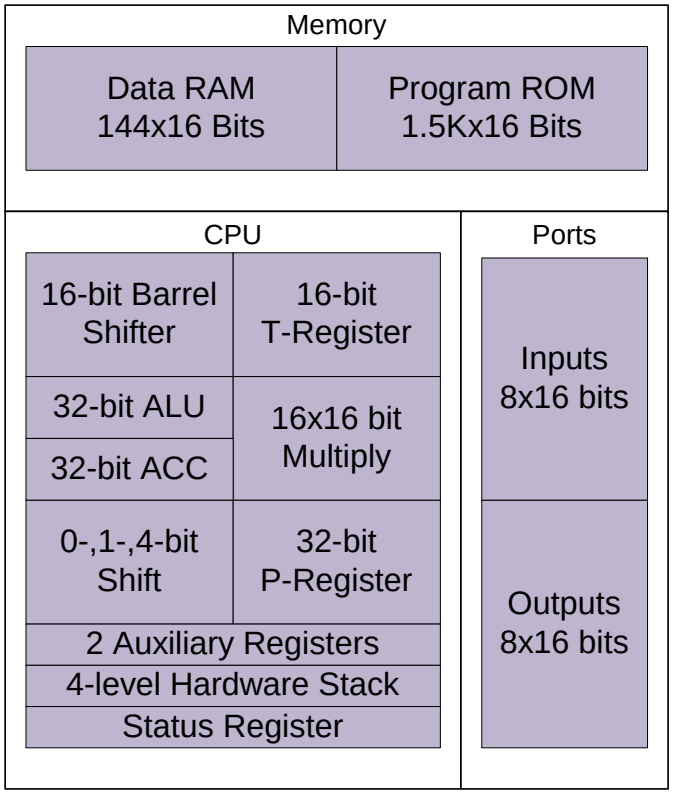
- Advanced *Harvard arcihecture*
  - Separate data/program buses
  - „Parallel” instruction fetch/operand fetch/execute
- Parallel multiplier
- Multiple Shifters
- Automatic incrementing/decrementing registers
- Most cost effective solution
- Broad line of processors
- Optimized for specific high-performance applications (telecom, control and consumer)



Első generációs DSP (1982): **TI320C10**

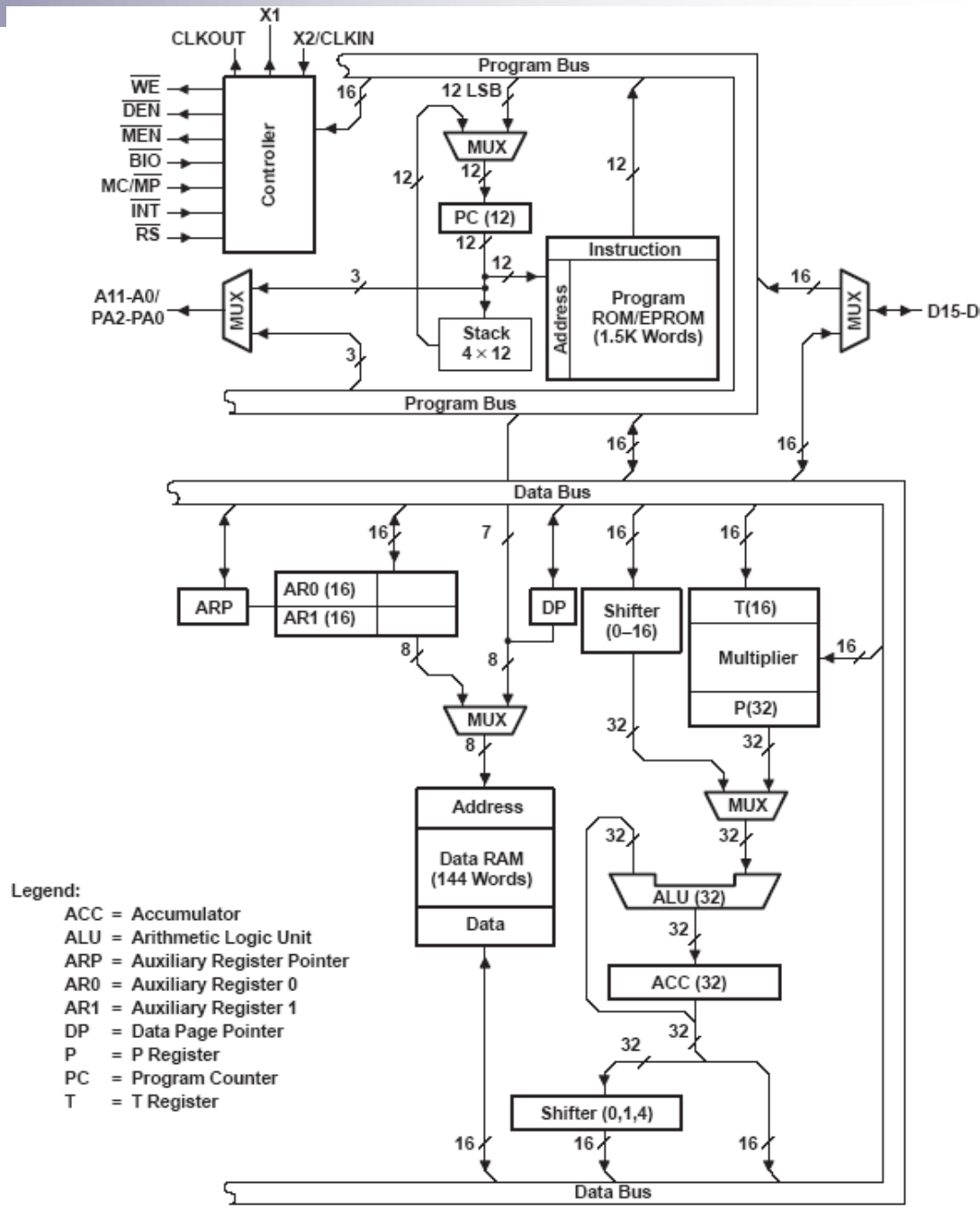
- Instruction Cycle Timing
  - — 160-ns (TMS320C10-25)
  - — 200-ns (TMS32010)
  - — 280-ns (TMS320C10-14)
- • 144 Words of On-Chip Data RAM
- • 1.5K Words On-Chip Program ROM
- • External Memory Expansion up to 4K Words Off-Chip Memory at Full Speed
- 60 General Purpose and DSP-specific instructions
- • 16 × 16-Bit Multiplier With 32-Bit Product (single-cycle)
- • 0 to 16-Bit Barrel Shifter
- • On-Chip Clock Oscillator
- • External interrupt and polled input pins
- Device Packaging:
  - — 40-Pin DIP
  - — 44-Lead PLCC
- • Single 5-V Supply
- • 8 16-bit I/O ports





TI320C10 blokk diagram

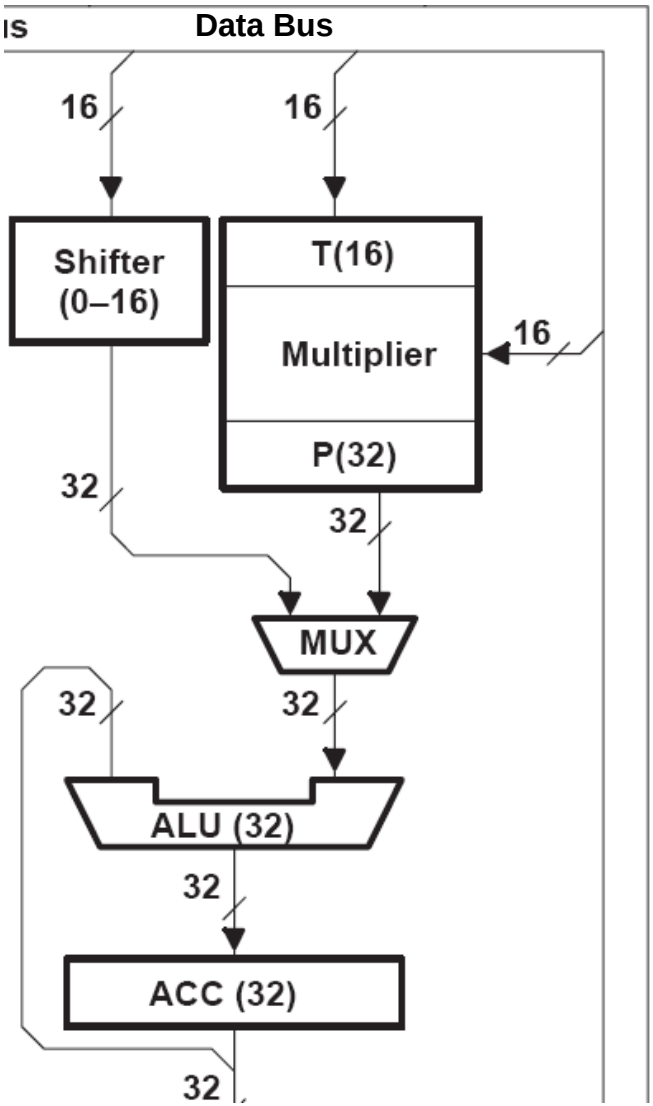
- Harvard architektúra
- 16-bites fixpontos MULT
- 16-bites Shifter
- 32-bites Akkumulátor
- 32-bites ALU
- Barrel-shifter (0-16)
- Specializált utasítás készlet
- 390ns (ma 228 ns) MAC time



Example: FIR Filter (sum of products)

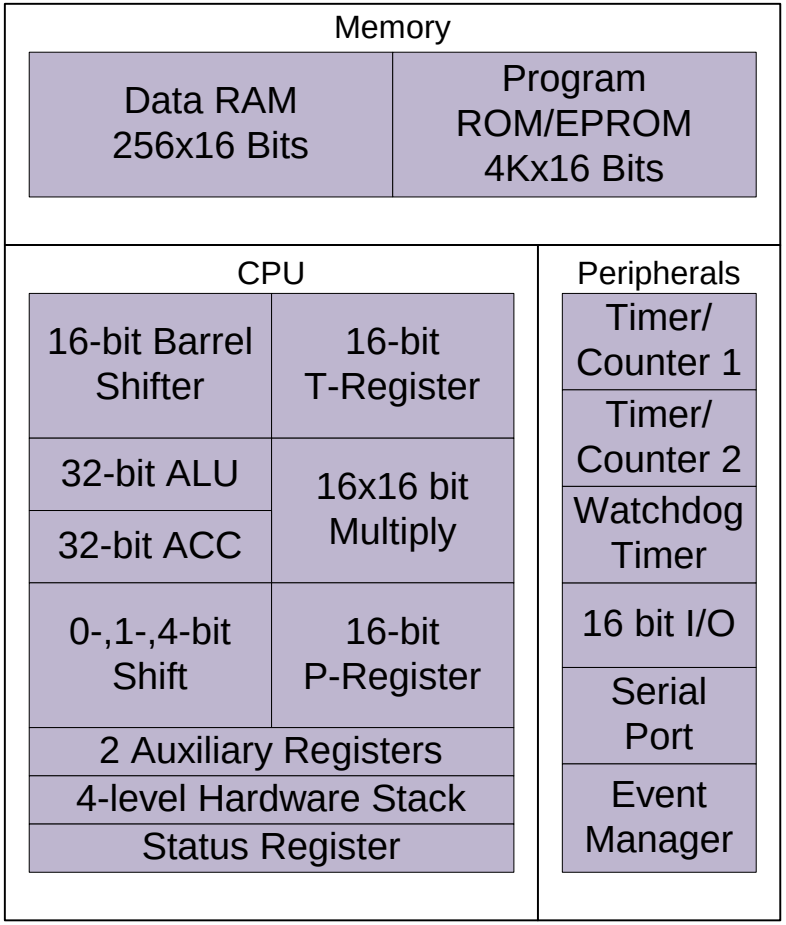
```
Compute:    Y[n]=A*X[n-1]+B*X[n-2]+C*X[n-3]+D*X[n-4]

START    IN       X1, PA0  //new input sample
   ZAC            //ACC=0
   LT    X1       //T=X1
   MPY   A        //P=A*X1
   LTA   X2       //ACC=A*X1 , T=X2
   MPY   B        //P=B*X2
   LTA   X3       //ACC=A*X1+B*X2 , T=X3
   MPY   C        //P=C*X3
   LTA   X4       //ACC=A*X1+B*X2+C*X3 , T=X4
   MPY   D        //P=D*X4
   APAC           //ACC=A*X1+B*X2+C*X3+D*X4
   SACH  Y        //STORE 32-bit result
                  //at location Y
   OUT   Y, PA0
   B START
```



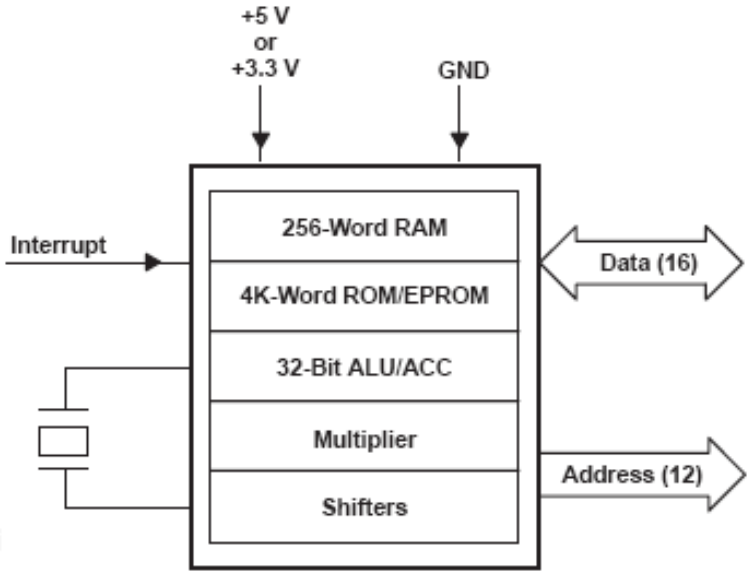
TI320C14/E14/P14

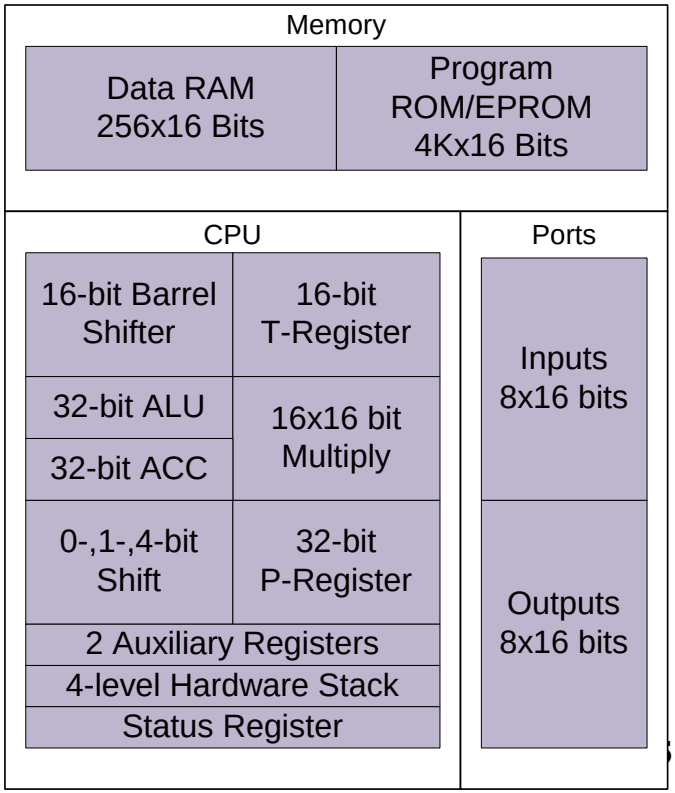
- Instruction Cycle Timing
  - — 160-ns
- 100% object-code compatible with TMS320C10/C15
- Timers and Counters
- Watchdog Timer
- Selectable I/O pins
- Serial Port
- Event Manager
- EPROM (E14) and OTP (P14)
- Microprocessor and microcomputer nodes



TMS320C15/E15/P15

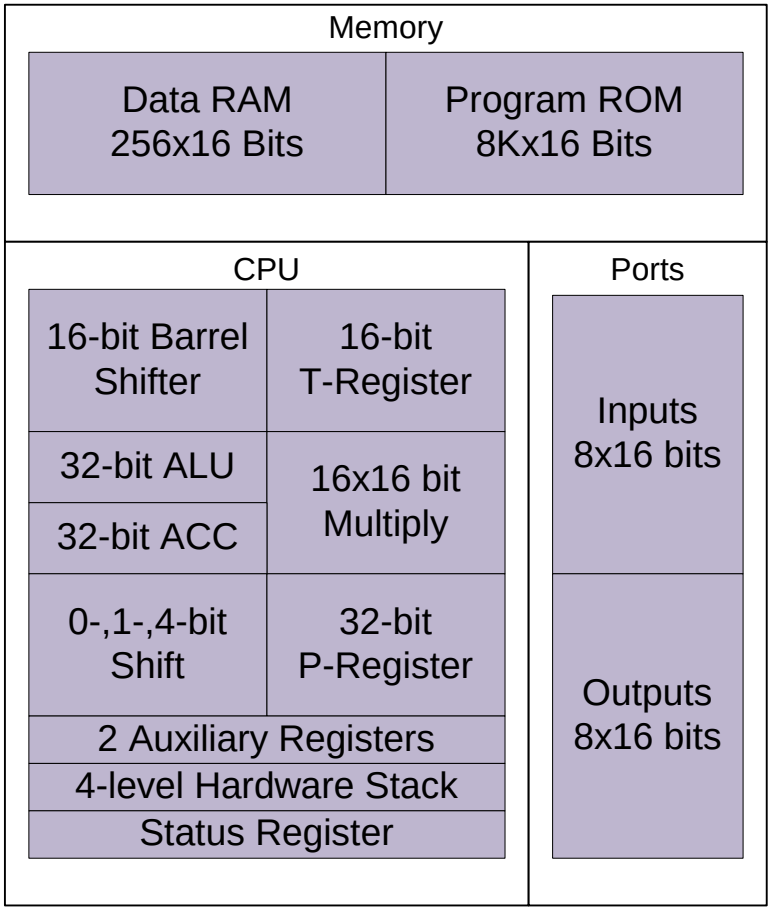
- Instruction Cycle Timing:
  - — 160-ns (TMS320C15-25/E15-25)
  - — 200-ns (TMS320C15/E15/P15)
  - — 250-ns (TMS320LC15)
- • 256 Words of On-Chip Data RAM
- • 4K Words of On-Chip Program ROM (TMS320C15/C15-25/LC15)
- • 4K Words of On-Chip Program EPROM (TMS320E15/E15-25)
- • One-Time Programmable (OTP) Windowless EPROM Version Available (TMS320P15)
- • EPROM Code Protection for Copyright Security
- • External Memory up to 4K-Words at Full Speed
- • 16 × 16-Bit Multiplier With 32-Bit Product
- • 0 to 16-Bit Barrel Shifter
- • On-Chip Clock Oscillator
- • 3.3-V Low-Power Version Available (TMS320LC15)
- • Device Packaging:
  - — 40-Pin Dip (All Devices)
  - — 44-Lead PLCC (TMS320C15/C15-25/LC15/P15)
  - — 44-Lead-QUAD (TMS320E15/E15-25)





TMS320C16

- Instruction Cycle Timing:
  - — 114ns
- • 256 Words of On-Chip Data RAM
- • 8K Words of On-Chip Program ROM
- • External Memory up to 64K-Words at Full Speed
- • 16 × 16-Bit Multiplier With 32-Bit Product
- • 0 to 16-Bit Barrel Shifter
- Separate I/O and memory write enable signals
- • On-Chip Clock Oscillator
- • Device Packaging:
  - — 64-Pin PQFP



TMS320C1x Device Overview (1991)

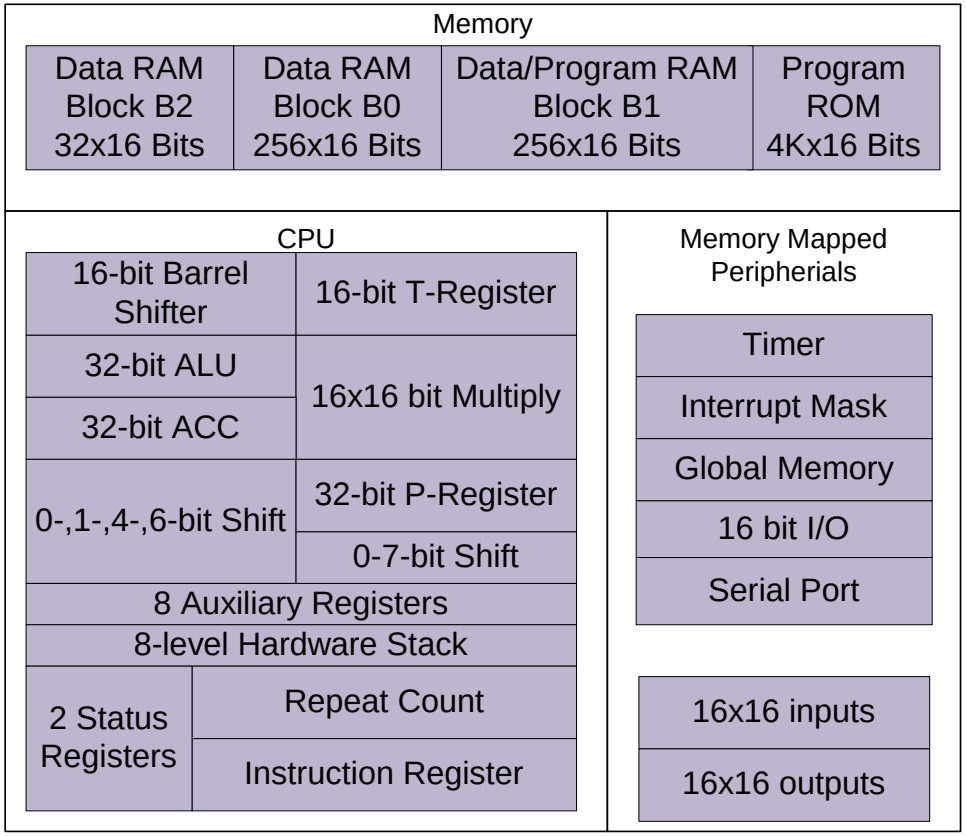
DEVICE	MEMORY				I/O		CYCLE	PACKAGE (1)		
	RAM	ROM	EPROM	PROG.	SERIAL	PARALLEL	(ns)	DIP	PLCC	CER-QUAD
TMS320C10 (2)	144	1.5K	—	4K	—	8 × 16	200	40	44	—
TMS320C10-14	144	1.5K	—	4K	—	8 × 16	280	40	44	—
TMS320C10-25	144	1.5K	—	4K	—	8 × 16	160	40	44	—
TMS320C14 (3)	256	4K	—	4K	1	7 × 16 (4)	160	—	68	—
TMS320E14 (3)	256	—	4K	4K	1	7 × 16 (4)	160	—	—	68 CER
TMS320P14†	256	—	4K	4K	1	7 × 16 (4)	160	—	68	—
TMS320C15 (3)	256	4K	—	4K	—	8 × 16	200	40	44	—
TMS320C15-25	256	4K	—	4K	—	8 × 16	160	40	44	—
TMS320E15 (3)	256	—	4K	4K	—	8 × 16	200	40	—	44 CER
TMS320E15-25	256	—	4K	4K	—	8 × 16	160	40	—	44 CER
TMS320LC15	256	4K	—	4K	—	8 × 16	250	40	44	—
TMS320P15†	256	—	4K	4K	—	8 × 16	200	40	44	—
TMS320C16	256	8K	—	64K	—	8 × 16	114	—	—	64 QFP
TMS320C17	256	4K	—	—	2	6 × 16 (5)	200	40	44	—
TMS320E17 (5)	256	—	4K	—	2	6 × 16 (5)	200	40	—	44 CER
TMS320LC17 (5)	256	4K	—	—	2	6 × 16 (5)	278	40	44	—
TMS320P17 (5)†	256	—	4K	—	2	6 × 16 (5)	200	40	44	

† One-time programmable (OTP) device is in a windowless plastic package and cannot be erased.

NOTES: 1. DIP = dual in-line package. PLCC = plastic-leaded chip carrier. CER = ceramic-leaded chip carrier. QFP = plastic quad flat pack.
2. Military version available.
3. Military versions planned; contact nearest TI Field Sales Office for availability.
4. On-chip 16-bit I/O, four capture inputs, and six compare outputs are available.
5. On-chip 16-bit coprocessor interface is optional by pin selection.

TMS320C25 Key Features – 2nd. generation (1990)

- 100-ns / 80-ns Instruction Cycle Times
- 544 Words of Programmable On-Chip Data RAM (B0-B2)
- 4K Words of On-Chip Program ROM
- 128K Words of total memory space
- 133 general purpose and DSP instructions
- 16 Input and 16 Output Channels
- 16-Bit Parallel Interface
- 16-Bit Instruction and Data Words
- 16 × 16-Bit Multiplier With a 32-Bit Product
- 32-Bit ALU and Accumulator
- Single-Cycle Multiply/Accumulate Instructions
- 0 to 16-Bit Scaling Shifter
- Bit Manipulation and Logical Instructions
- Floating-Point Operations, Adaptive Filtering, and Extended-Precision Arithmetic
- Repeat Instructions for Efficient Use of Program Space
- Eight Auxiliary Registers and Dedicated Arithmetic Unit for Indirect Addressing
- Serial Port for Direct Code Interface
- Synchronization Input for Synchronous Multiprocessor Configurations
- Wait States for Communication to Slow-Off-Chip Memories/Peripherals
- On-Chip Timer for Control Operations
- Three External Maskable User Interrupts
- 1.6-µm CMOS Technology
- Programmable Output Pin for Signaling External Devices
- Single 5-V Supply
- HLL: High-level language



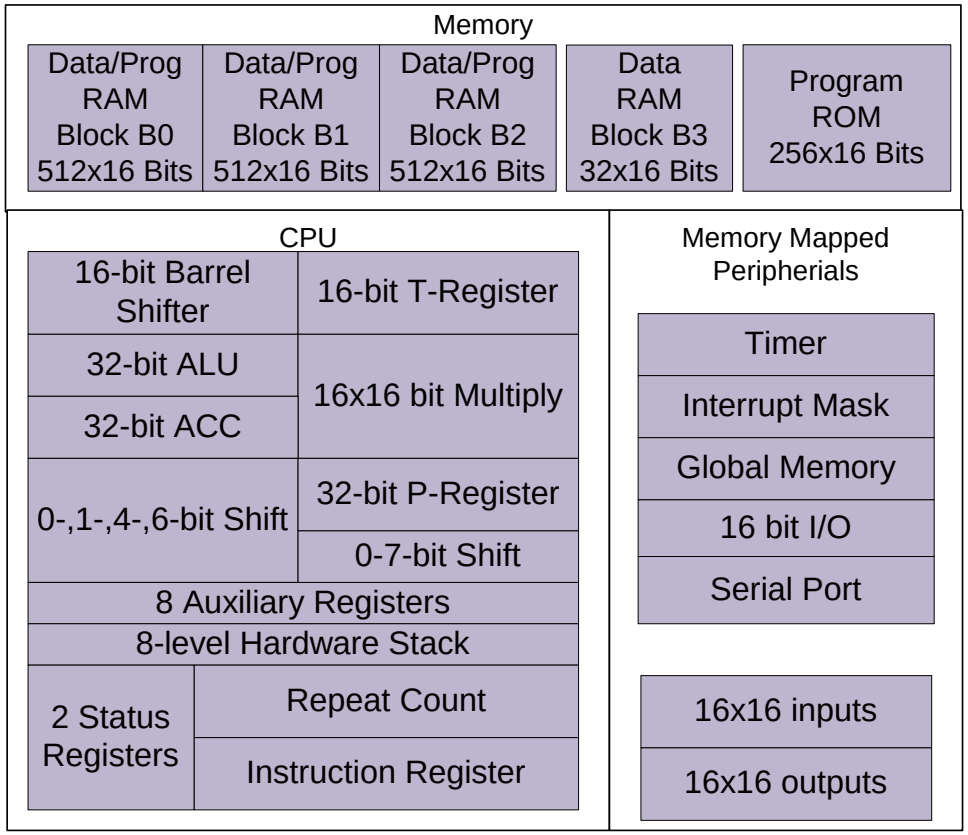
TMS320C25 Device Overview (1990)

DEVICE		MEMORY				I/O†			TIMER	CYCLE TIME (ns)	TYP POWER (mW)	PACKAGE TYPE		
		ON-CHIP		OFF-CHIP										
		RAM	ROM/EPROM	PROG	DATA	SER	PAR	DMA				PGA	PLCC	CER-QUAD
TMS32020‡	(NMOS)	544	—	64K	64K	YES	16 × 16	YES	YES	200	1250	68	—	—
TMS320C25‡	(CMOS)	544	4K	64K	64K	YES	16 × 16	CON	YES	100	500	68	68	—
TMS320C25-50§	(CMOS)	544	4K	64K	64K	YES	16 × 16	CON	YES	80	500	—	68	—
TMS320E25§	(CMOS)	544	4K	64K	64K	YES	16 × 16	CON	YES	100	500	—	—	68

† SER = serial; PAR = parallel; DMA = direct memory access; CON = concurrent DMA.
‡ Military version available; contact nearest TI Field Sales Office for availability.
§ Military version planned; contact nearest TI Field Sales Office for details.

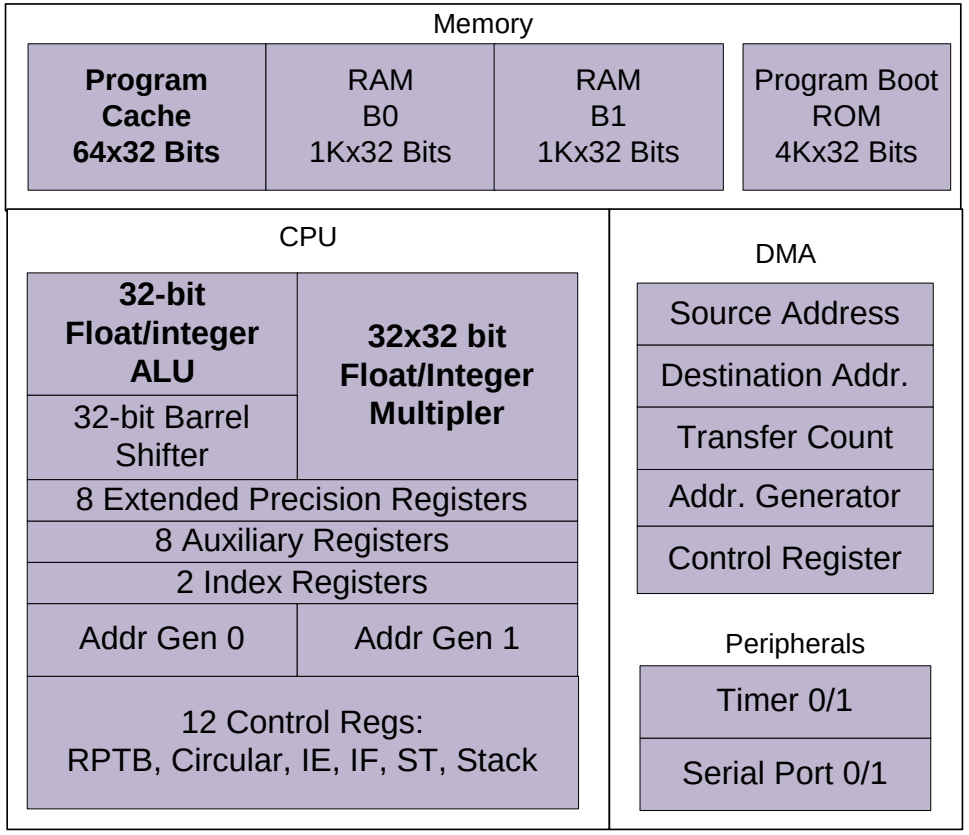
TMS320C26 Key Features

- 100-ns Instruction Cycle Times
- 1,5K Words of On-Chip Data/Program RAM (B1-B3)
- Floating-Point Operations, Adaptive Filtering, and Extended-Precision Arithmetic
- 32 Words of On-chip Data RAM (B3)
- 256 Words of On-Chip Program ROM
- 128K Words of total memory space
- 133 general purpose and DSP instructions
- 16 Input and 16 Output Channels
- 16-Bit Parallel Interface
- 16-Bit Instruction and Data Words
- 16 × 16-Bit Multiplier With a 32-Bit Product
- 32-Bit ALU and Accumulator
- Single-Cycle Multiply/Accumulate Instructions
- 0 to 16-Bit Scaling Shifter
- Bit Manipulation and Logical Instructions
- Repeat Instructions for Efficient Use of Program Space
- Eight Auxiliary Registers and Dedicated Arithmetic Unit for Indirect Addressing
- Serial Port for Direct Code Interface
- Synchronization Input for Synchronous Multiprocessor Configurations
- Wait States for Communication to Slow-Off-Chip Memories/Peripherals
- On-Chip Timer for Control Operations
- Three External Maskable User Interrupts
- 1.6-µm CMOS Technology
- Programmable Output Pin for Signaling External Devices
- Single 5-V Supply
- HLL: High-level language



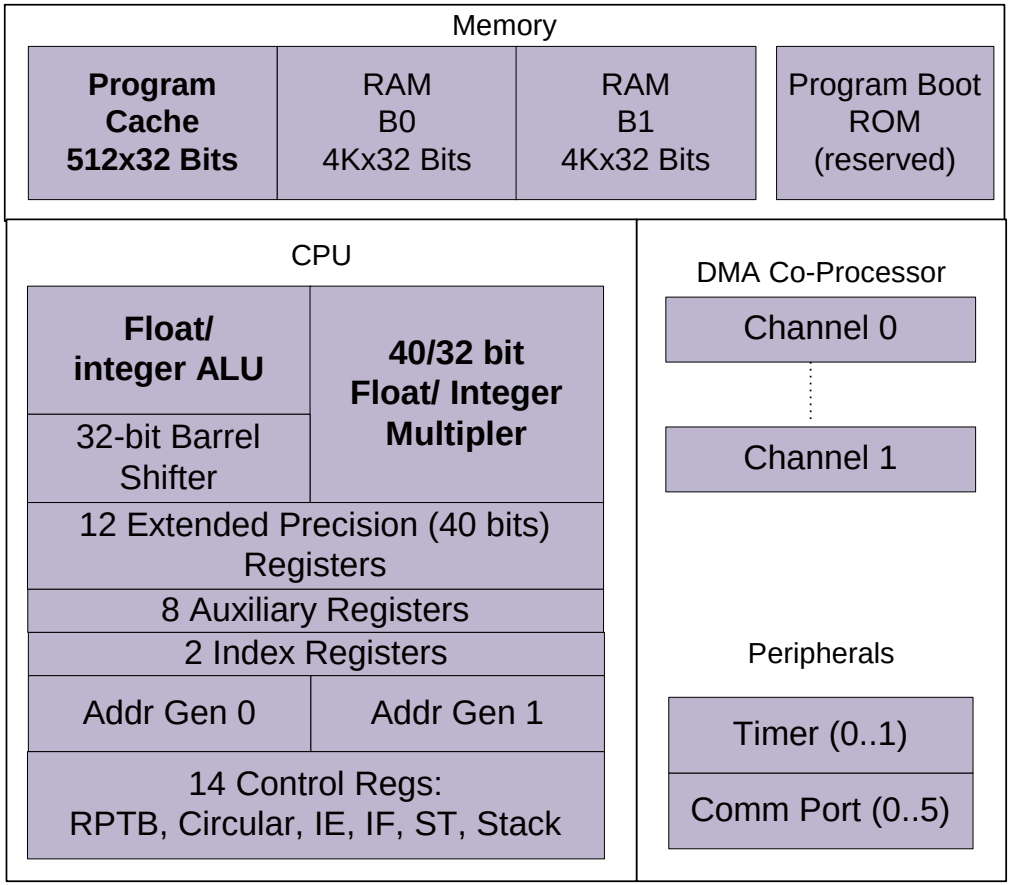
TMS320C3x Key Features (1999 upd.)

- **Parallel DSP CPU!**
- 25-75 ns instruction cycle time
- Single-cycle MAC
- 16-/32-Bit Integer and 32-/40-Bit Floating-Point Operations
- 40-50 (max 80 MFLOPS)
- Max 440 MOPS
- 16M words external address reach
- ANSI-C compiler
- 0.8-µm CMOS Technology



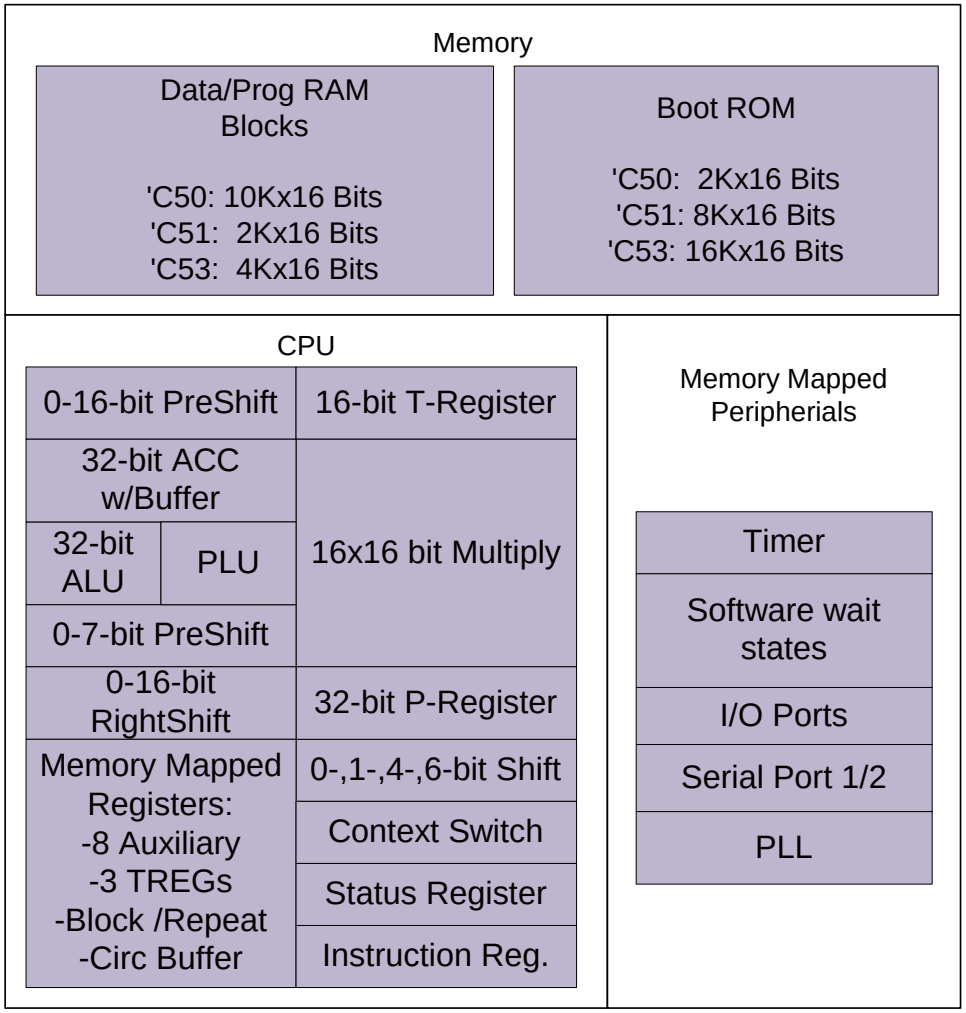
TMS320C40 Key Features (2004 upd.)

- **High-speed Parallel DSP CPU!**
  (8 operations/cycle)
  - Floating point/integer multiply
  - Floating point/integer addition
  - Two data accesses
  - Two address register updates
  - Zero overhead branch and loop counter update
- 33-50 ns cycle-time
- Max. 330 MOPS, max. 60 MFLOPS
- 40-Bit Floating-Point
- 32-Bit Integer Multipliers
- 512-byte On-chip Program CACHE
- 8K Bytes of Single-Cycle Dual-Access Program or Data RAM
- IEEE-754 Floating Point Format
- DIV and SQRT support /clk
- 6-channel DMA coprocessor
- 6 communication ports
- 325-pin CPGA package
- 0.8-µm CMOS Technology



TMS320C5x Key Features (1996 upd.)

- Source code compatible with the TMS320C1x/C2x
- 25/ 50ns instruction cycle
- 50 MIPS
- 16-bit PLU (Parallel Logic Unit) – bit manipulation
- Expanded blocks of on-chip data/program RAM and program ROM
- Single-cycle MAC
- 192K words of addressable total memory
- Added general purpose and DSP specific instructions
- 1.6-µm CMOS Technology (132-pin QFP package)
- JTAG test/emulation control
- ANSI C compiler



TMS320C5x Device Overview (1996)

TMS320 DEVICES	ON-CHIP MEMORY (16-BIT WORDS)				I/O PORTS		POWER SUPPLY (V)	CYCLE TIME (ns)	PACKAGE TYPE QFP‡
	DARAM		SARAM	ROM					
	DATA	DATA + PROG	DATA + PROG	PROG	SERIAL	PARALLEL†			
TMS320C50	544	512	9K	2K§	2	64K	5	50/35/25	132 pin
TMS320LC50	544	512	9K	2K§	2	64K	3.3	50/40/25	132 pin
TMS320C51	544	512	1K	8K§	2	64K	5	50/35/25/20	100/132 pin
TMS320LC51	544	512	1K	8K§	2	64K	3.3	50/40/25	100/132 pin
TMS320C52	544	512	–	4K§	1¶	64K	5	50/35/25/20	100 pin
TMS320LC52	544	512	–	4K§	1¶	64K	3.3	50/40/25	100 pin
TMS320C53	544	512	3K	16K§	2	64K	5	50/35/25	132 pin
TMS320LC53	544	512	3K	16K§	2	64K	3.3	50/40/25	132 pin
TMS320C53S	544	512	3K	16K§	2¶	64K	5	50/35/25	100 pin
TMS320LC53S	544	512	3K	16K§	2¶	64K	3.3	50/40/25	100 pin
TMS320LC56	544	512	6K	32K	2#	64K	3.3	35/25	100 pin
TMS320LC57	544	512	6K	32K	2#	64K + HPI‖	3.3	35/25	128 pin
TMS320C57S	544	512	6K	2K§	2#	64K + HPI‖	5	50/35/25	144 pin
TMS320LC57S	544	512	6K	2K§	2#	64K + HPI‖	3.3	50/35	144 pin

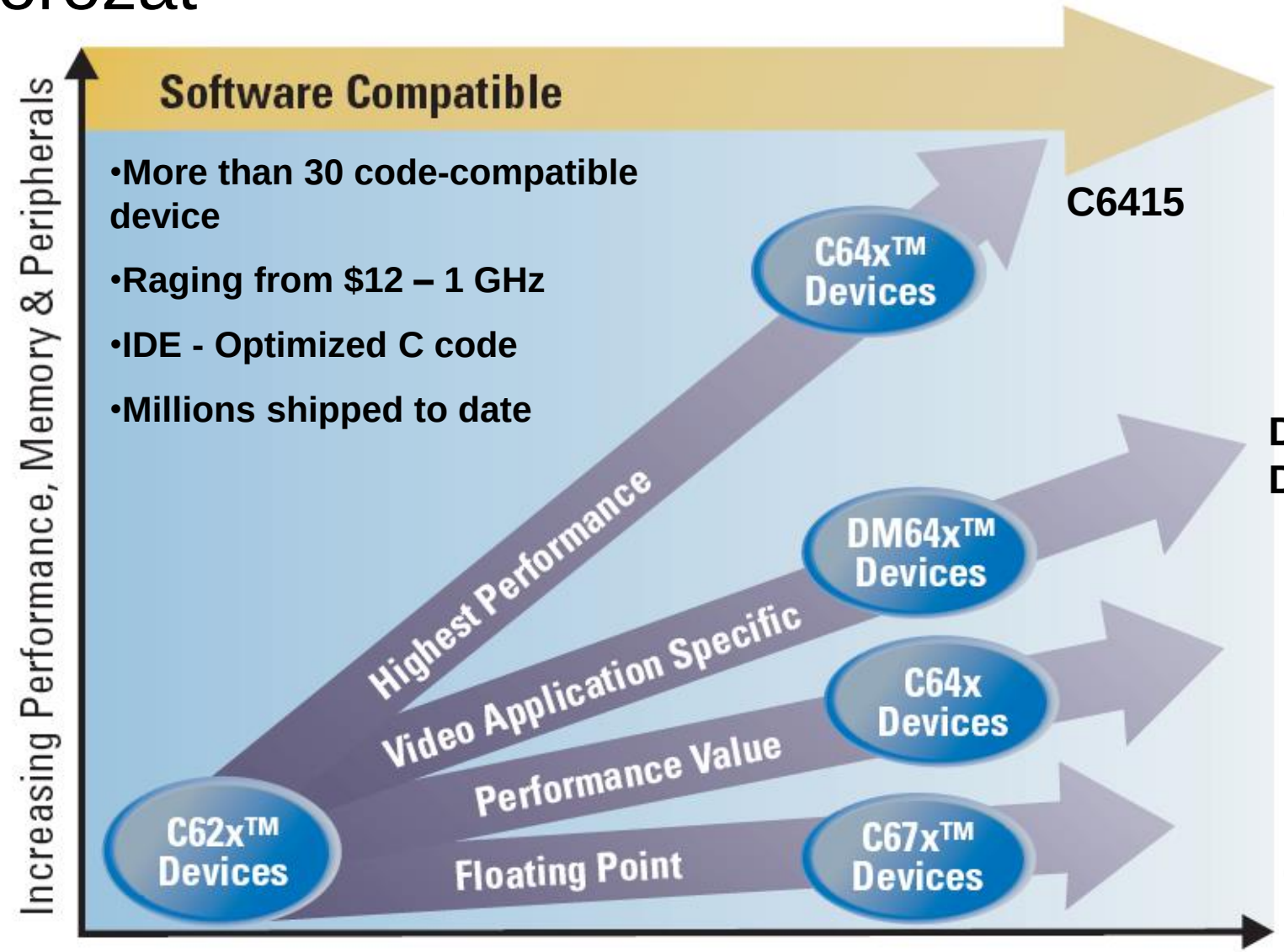
† Sixteen of the 64K parallel I/O ports are memory mapped.
‡ QFP = Quad flatpack
§ ROM boot loader available
¶ TDM serial port not available
Includes auto-buffered serial port (BSP) but TDM serial port not available
‖ HPI = Host port interface

Texas Instruments TMS320C6000 – nagyteljesítményű processzor család



www.ti.com

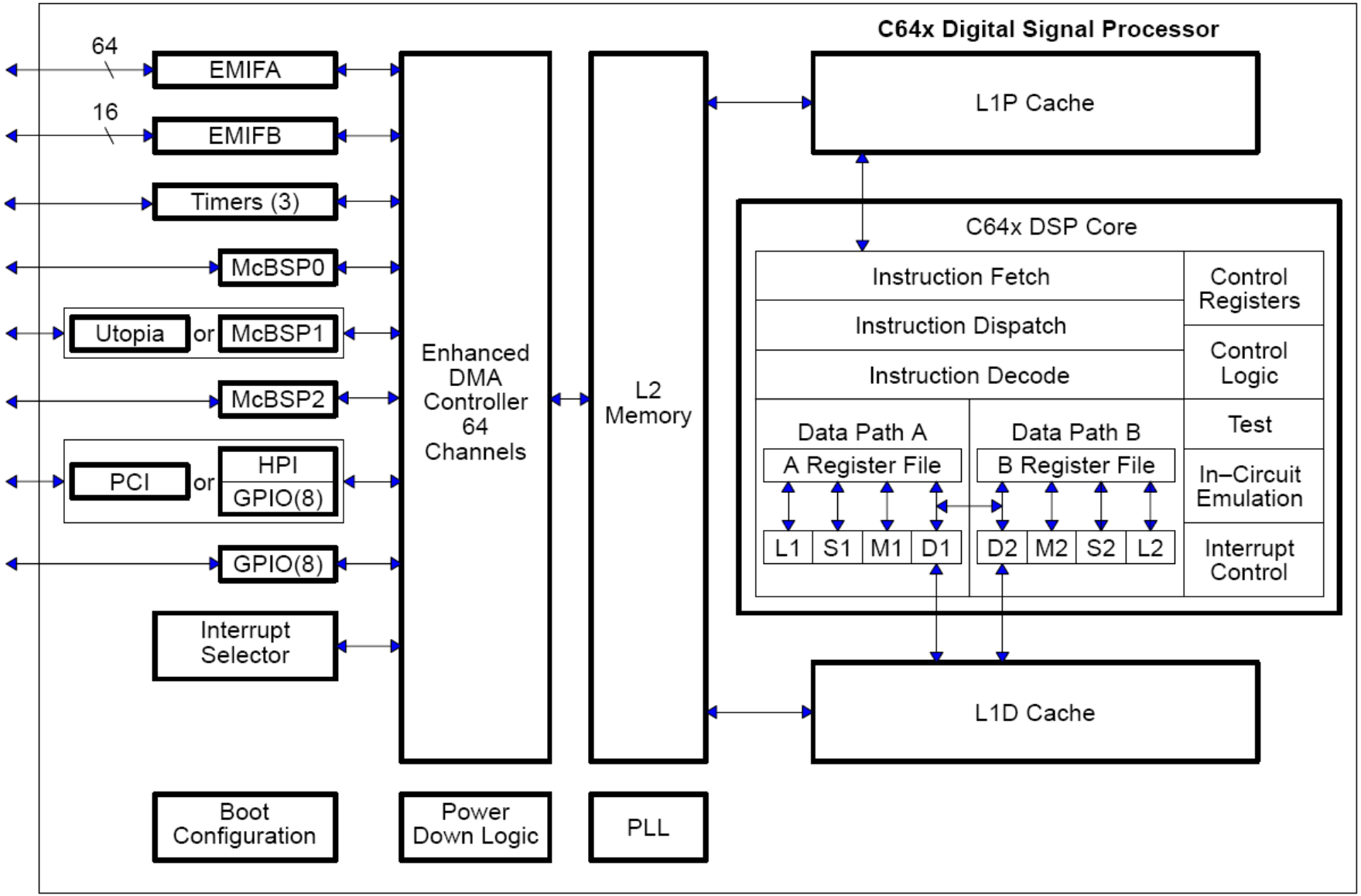
Modern DSP architektúrák: TI C6000 sorozat



A TMS320C64x gyártási paraméterei

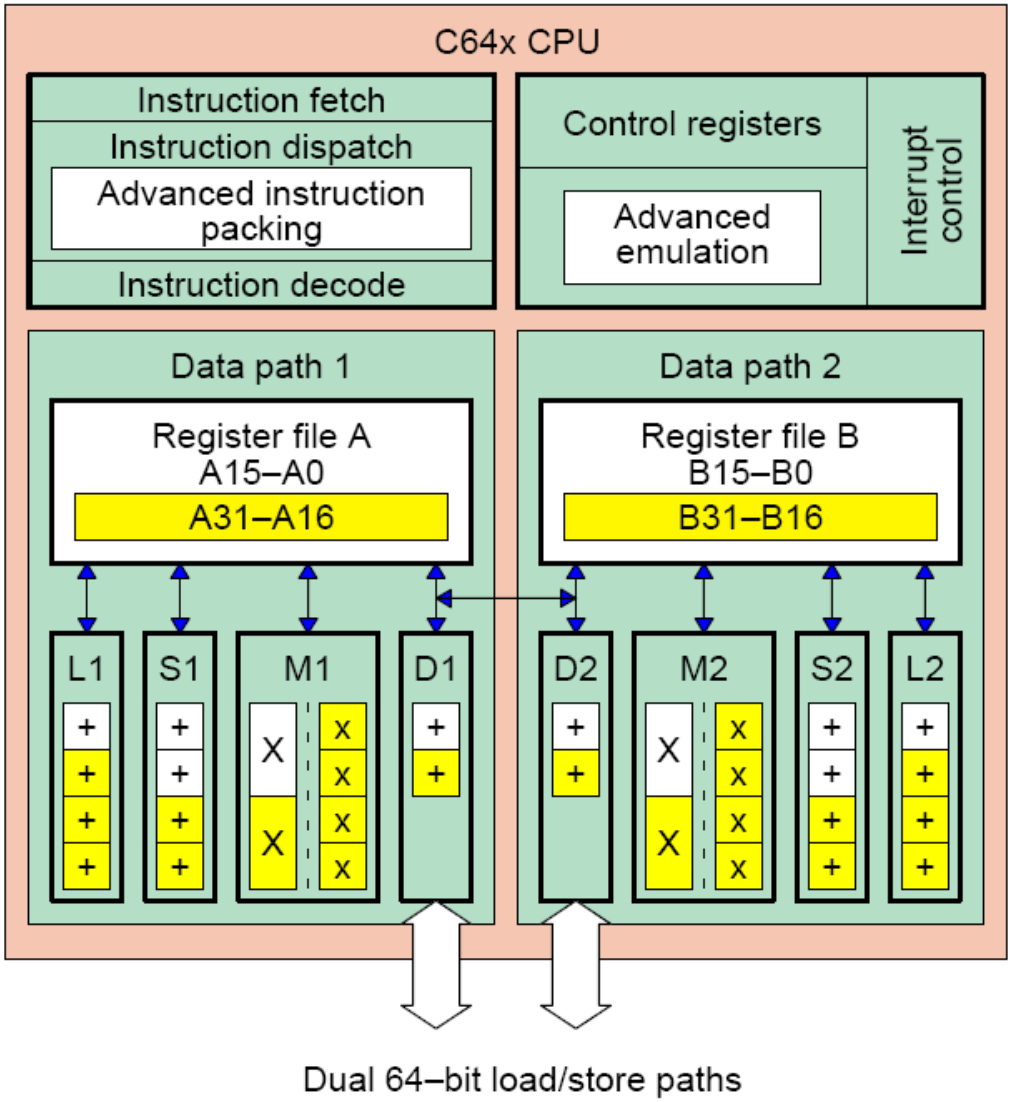
- 600 /720 /850 MHz and 1- GHz Clock Rate
- 90 nm/ 7- Level C*u* Metal Process (CMOS)
- Power consumption @1GHz ~1W
- 1GHz mass production started 2004 Q4
- 532 Pin - Ball Grid Array

A C6415 DSP architektúrája



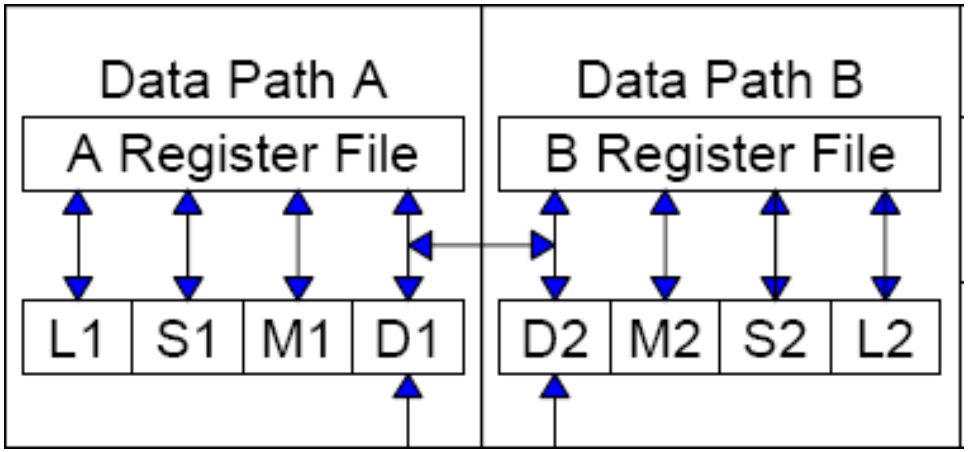
C64x processzor magja

- 2 Regiszter bank
- 8 Processzáló egység
  - L, S, M, D (2 x 4)
  - Változó bitmélységű műveletek (16, 8)
- 2 Adatútvonal
- Keresztirányú adatútvonalak
- Speciális utasítások
  - (VelociTI.2 extensions)

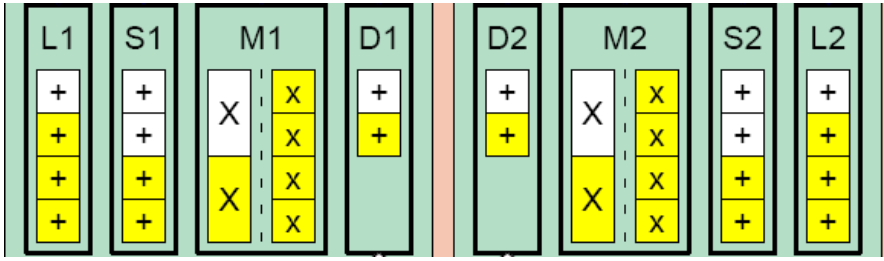


C64x processzor magja (folyt):
Regiszter bankok

- Két regiszter bank (A,B)
  - Duális struktúra
- 32db 32-bites regiszter mindkét bankban
- Három regiszter típus
  - Feltétel regiszter (3 db/regiszterbank)
  - Körkörös címregiszter (4 db/regiszterbank)
  - Adatregiszter



C64x Processzáló egységek



M1, M2 egységek
Szorzó összeadó (MAC) egység
16 x 32 multiply operations
Dual and quad multiply operations
Dual and quad multiply with add/subtract operations (MAC)
Rotation
Variable shift operations

S1, S2 egységek
Aritmetika, komaprálások, elágazások
32–bit arithmetic and logical operations
32/40–bit shifts
Branches
Dual and quad compare operations
Dual and quad saturated arithmetic operations

L1, L2 egységek
Logikai és aritmetikai egység
32/40–bit arithmetic and compare ops
32–bit logical operations
Byte shifts
Dual /quad arithmetic operations
Dual /quad min/max operations
Quad 8–bit subtract with absolute value

D1, D2 egységek
Címaritmetikai egység
32–bit add, subtract, linear and circular address calculation
Loads and stores single and double words with constant offset
32–bit logical operations
Dual 16-bit arithmetic operation

C6415 magja: adatút ‘A’

Data-path A

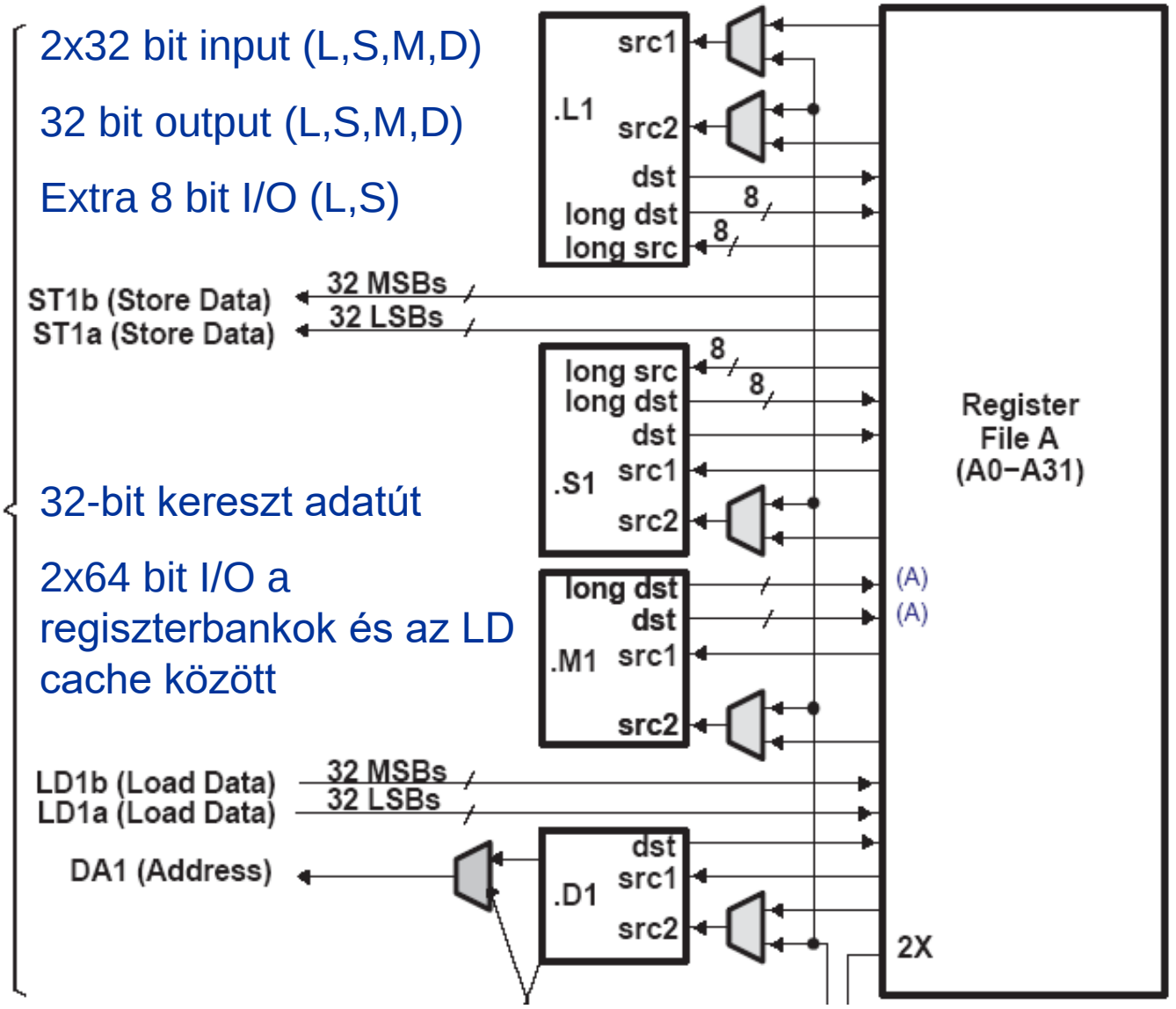
2x32 bit input (L,S,M,D)

32 bit output (L,S,M,D)

Extra 8 bit I/O (L,S)

32-bit kereszt adatút

2x64 bit I/O a regiszterbankok és az LD cache között



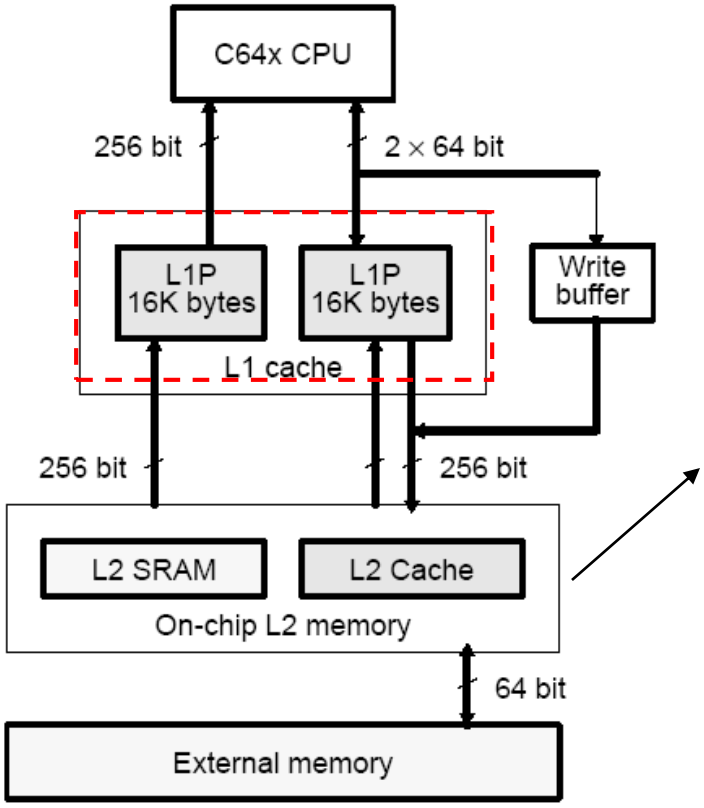
C64x Utasításbusz

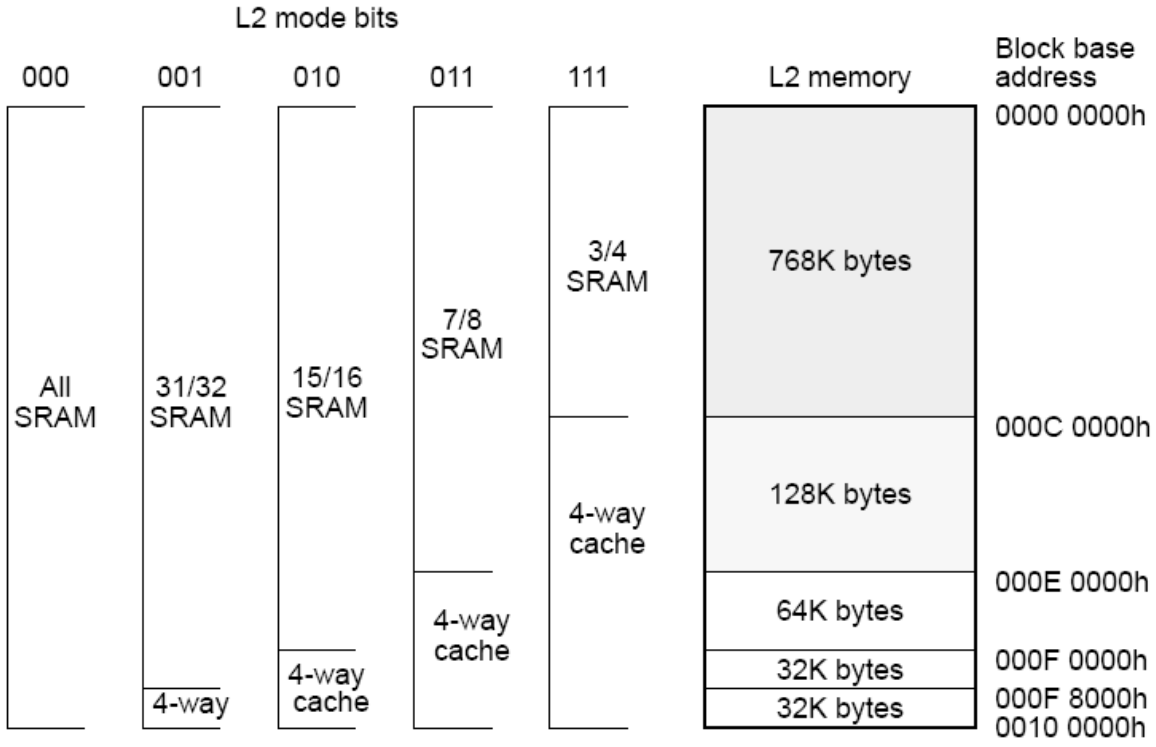
- Mind a 8 processzornak 32 bites az utasításkód szélessége → 256 bit széles utasításbusz
  - Ez egyetlen busz, azaz az a különböző processzorok nem ágazhatnak el önállóan!
  - Tipikus C kódnál 2-3 processzor megy egyszerre.

C64x: L1, L2 Cache-memória

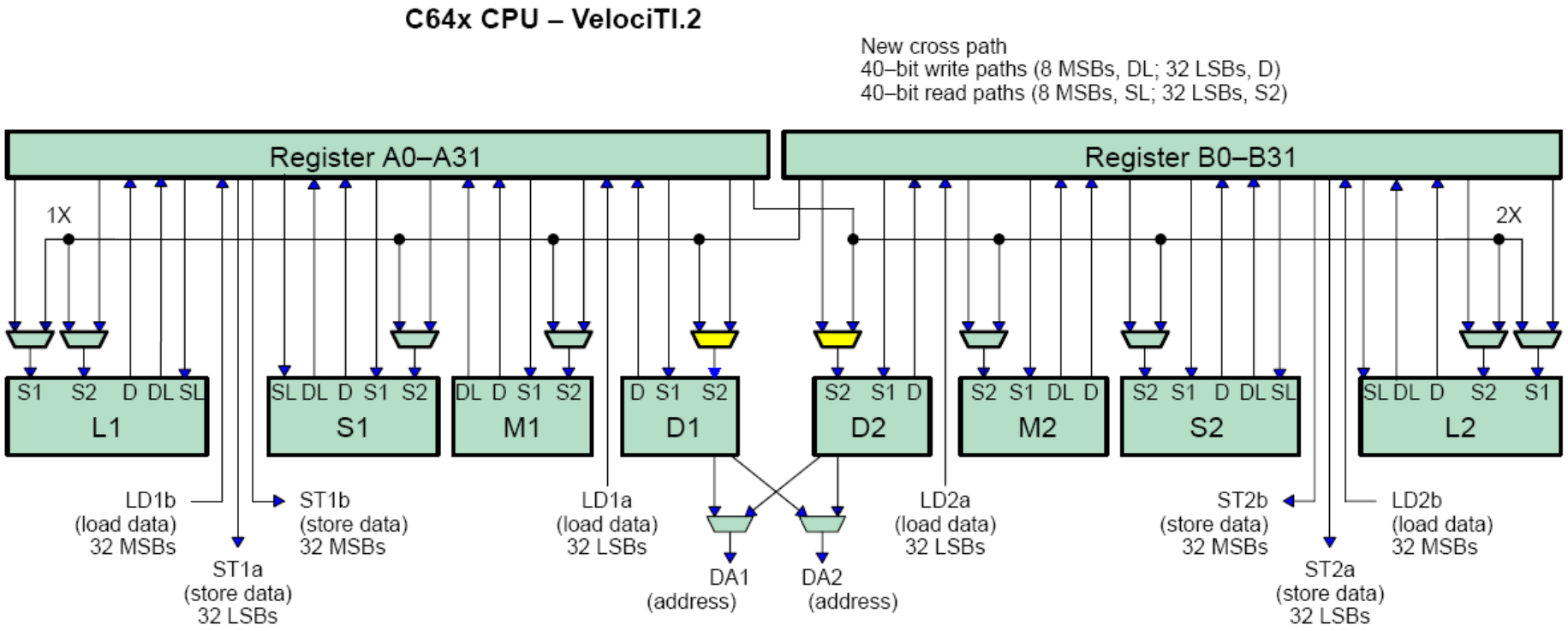
L1 cache (2x16K): külön adat és programtár

L2 cache (1024K): konfigurálható





C64x mag: Keresztirányú adatútvonalak



A C64x processzorok DMA-ja

- DSP minden I/O kérést a DMA-n keresztül éri el
- 64 DMA csatorna
- Összesen 2 Gbyte/sec
- Kapcsolt DMA csatorna (linking)
  - Eseményhez csatolt DMA csatorna, amelyik automatikusan indul az esemény hatására (intr)
- Láncolt DMA csatorna (chaning)
- DMA csatorna egy másik DMA transzfer végén indul (loop lánc)
- Egyszeri inicializálás után önjáró DMA-k
- Körkörös, illetve ping-pong adatterületek
- 1D-2D csatornák
  - – Képből egy részkép kivágása, sorfolytonos olvasással

Négy független DMA queue

- Concurrency (4 független – konkurens sor)
- Channels on different queues can interleave transfers on a cycle by cycle basis. e.g.:
  - cycle 1 queue 0
    - service a L2 cache miss to EMIFA (64 bit)
  - cycle 2 queue 1
    - Move data from a serial port to EMIFB (16 bit)
  - cycle 3 queue 3
    - the PCI/HPI could transfer data to mapped internal memory.
  - cycle 4 queue 4
    - the EMIFA could move data to a serial port (RS232)

A C64x processzorok perifériái

Három párhuzamos busz

- 64 bites (EMIFA), és egy 16 bites (EMIFB)
  - 3-3 memória vezérlő
    - SDRAM vezérlő
    - Szinkron vezérlő (szinkron burst SRAM, stb)
    - aszinkron vezérlő (SRAM, FIFO, egyéb periféria)
  - HPI vagy PCI interface
- Programozható I/O csatorna (GPIO)
- Három soros port (McBSPx)
  - Kapcsolat perifériákhoz, és más C64x processzorhoz

Texas Instruments TMS320DM64xx – Da Vinci DSP család & SW Tools



www.ti.com

TI Da-Vinci Digital Media processzorcsalád



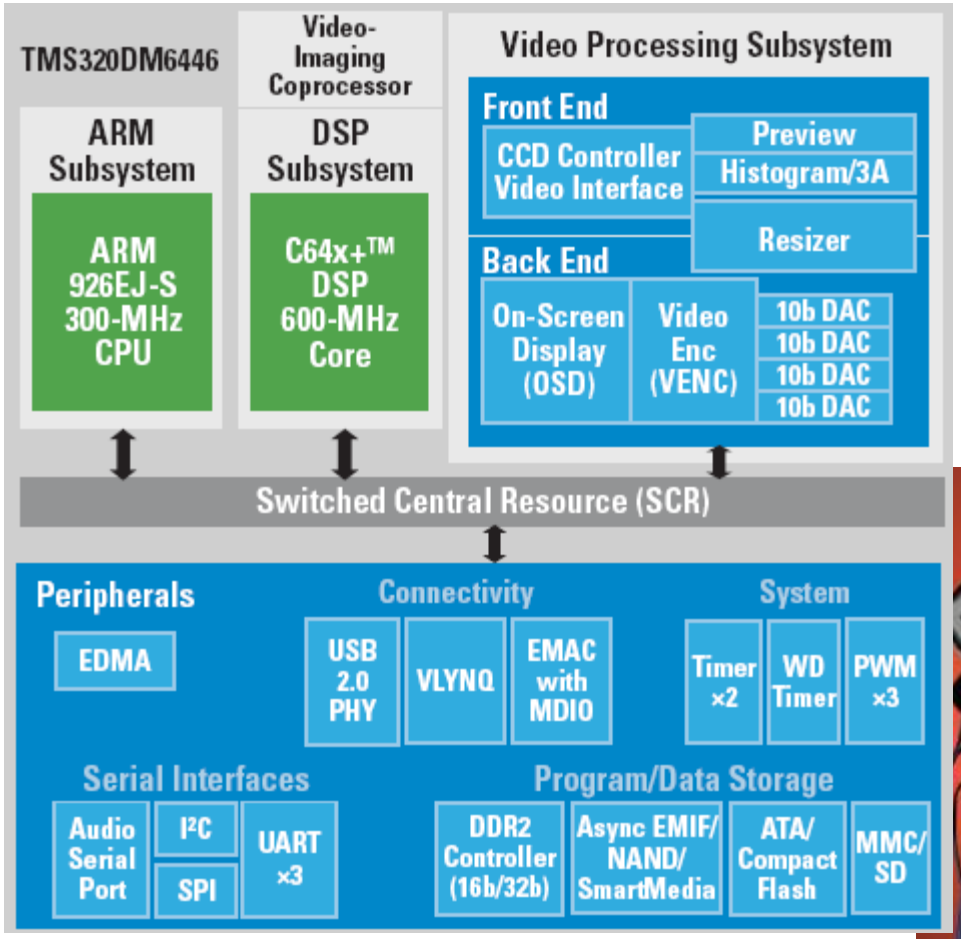
- **Da-Vinci DSP: tailored for digital video applications**
- **TMS320DM644x: highest integrated SoC**
  - video phones, automotive infotainment, digital stillcameras, streaming media and IP settop boxes.
- **TMS320DM643x: low cost processor**
  - Automotive video applications (lane departure and collision avoidance)
  - Machine-vision systems
  - Robotics
  - Video security

- Beépített kódolás/dekódolási módok:
  - H.264, MPEG-4, JPEG, AAC, MP3, G.711, HE-AAC, AC3, WMA9 stb.

TI DM643x és DM644x sorozat



TMS320DM6446 Da-Vinci DSP

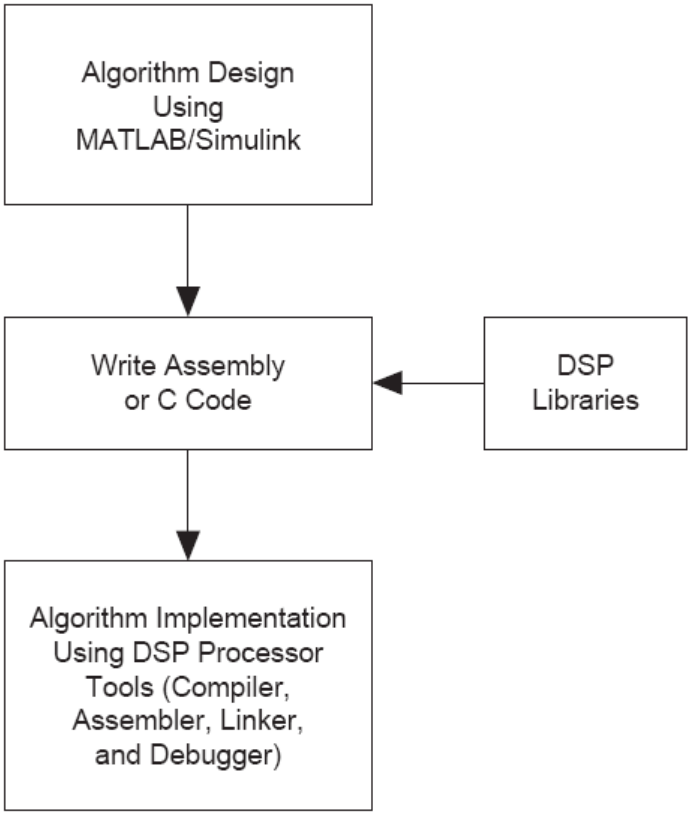


- DVEVM: Digital Video EValuation Module (HW) ~ 2500$
- SDK SW Tools: 7000$ - 11000$

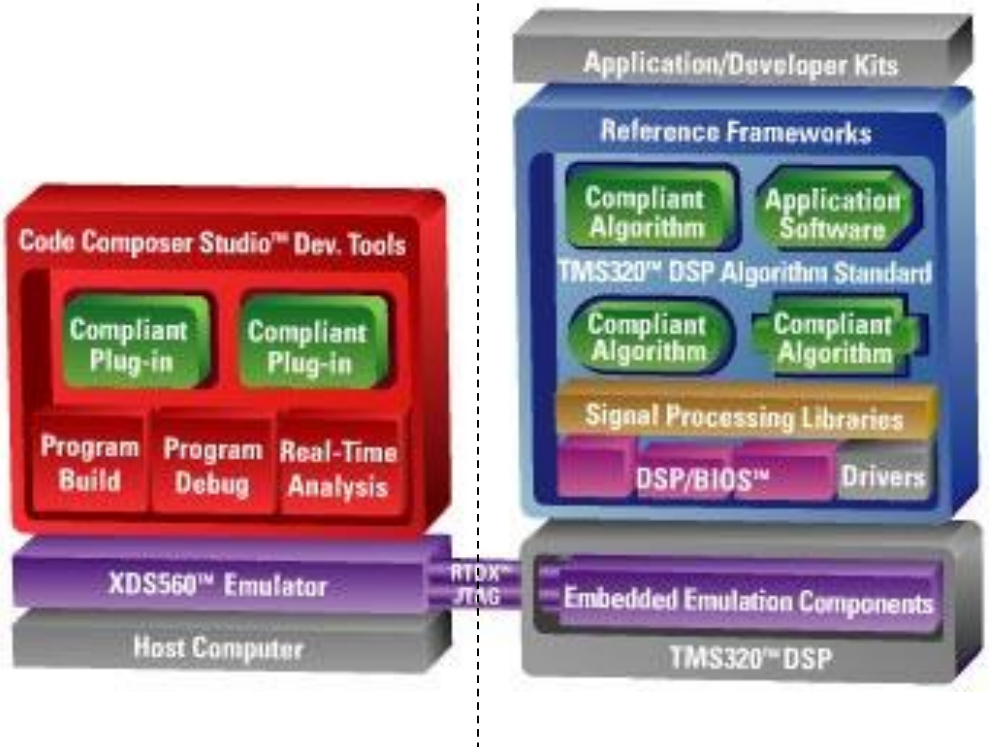
(2007 Q2)



DSP Sw-based Design-Flow



- Code Composer Studio
- eXpressDSP™ Software and Development Tool



IDE Features Throughout the Algorithm Development Cycle



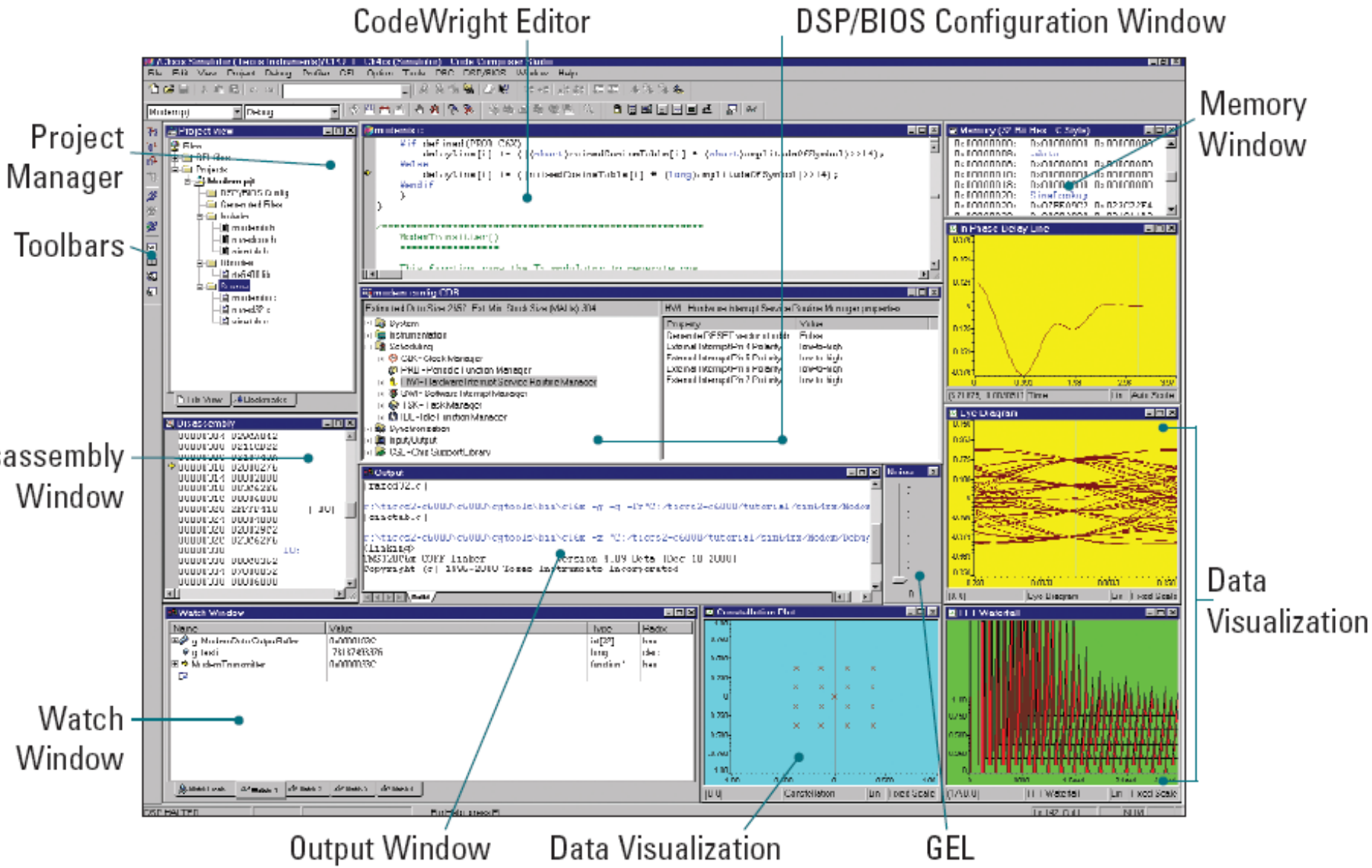
- Select Target
- DSP/BIOS Config
- eXpressDSP Algorithm Standard
- Update Advisor
- Power Planning

- Code Gen Tools
- Project Manager
- Editor
- Online Help
- Power Management

- Debugger
- Simulation
- Multi-Processor
- Data I/O & RTDX
- Advanced Event Triggering
- Scripting
- Connect / Disconnect
- Rewind

- Real-Time Analysis
- Profiler
- Visualize Data
- CodeSizeTune
- Code Coverage
- Cache Analysis
- Tuning Dashboard
- Compiler Consultant

TI Code Composer Studio



Kapcsolódó háttéranyagok:

- www.ti.com
- http://focus.ti.com/dsp/docs/dspgettingstarted.tsp?sectionId=4
  (Getting Started DSP)
- http://www-s.ti.com/sc/techlit/ssdv004.pdf
  (Selection Guide 2007 Q2)

High-level synthesis

Think about it…

- Standard-cell methodology and tools were developed for easy mapping of logic-level design into IC layout.
- BUT: example:

  32-bit: c = a + b.

  We do not write 32 Boolean expressions with up to 64 variables each to indicate this simple operation.

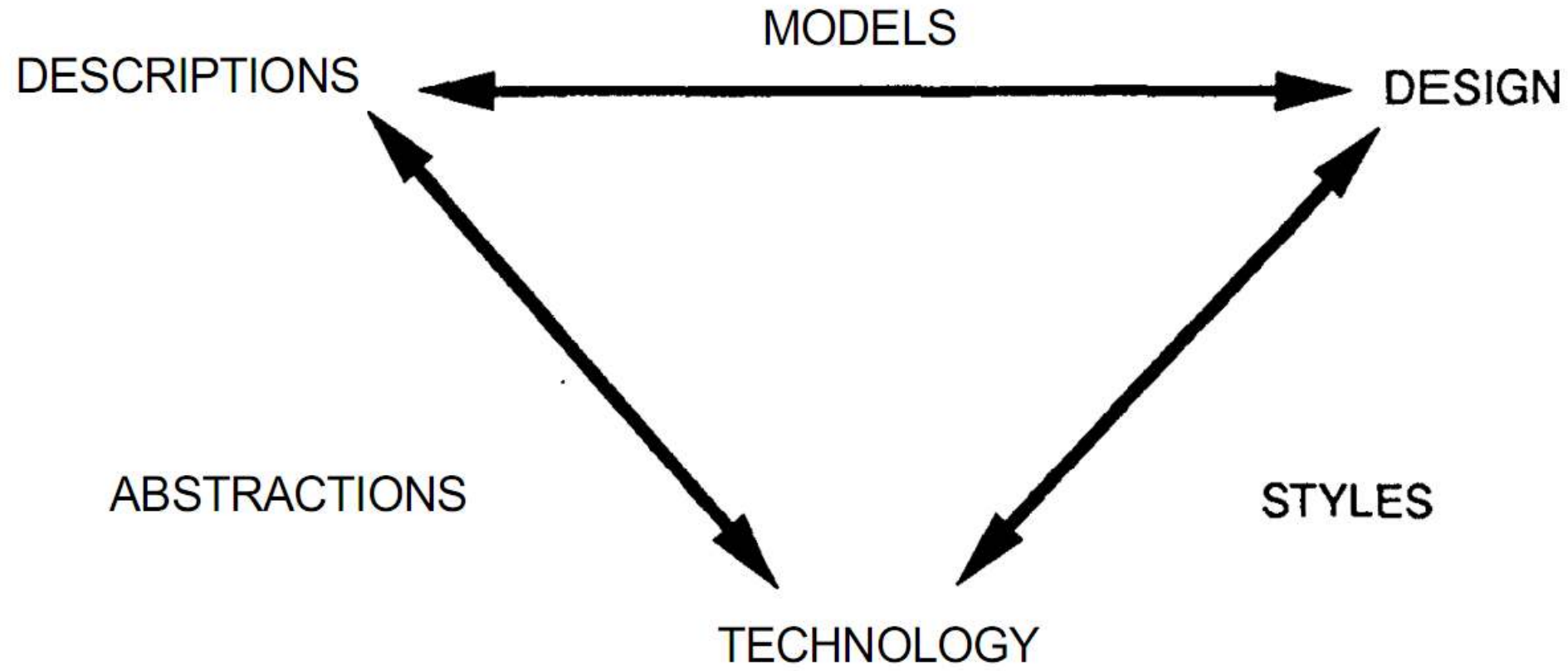
  Imagine having complex multi-chip systems described in terms of 1 million or more Boolean equations.

Main goal

- The main goal is to explore design auto-mation for synchronous digital systems at levels above the logic level in a practical environment.
- implement a system that is fast enough to allow the synthesis of large designs in a reasonable time
- high-level synthesis of synchronous digital systems

- input is a sequential specification of the function of a synchronous digital system, and possibly constraints such as the number and/or the type of the hardware modules to use, timing constraints, etc.
- assigns every operation in the specification to a control step (schedul-ing) and synthesizes the necessary hardware (allocation)
- The resulting design consists of a finite state machine (FSM) that implements the control and a netlist specifying the data path

Description-Design-Technology Dependence



Essential Issues in Synthesis

- Design Conceptualization
- Database Issues
- Technology Independence
- Design Learning
- Design Synthesis Complexity

Design Conceptualization

- Problem: design description is changing with design over time
- generate different design views for optimization or verification of different aspects of a design
- allow manual modification of synthesized design
- provide design quality metrics for exploration and design evaluation

Database Issues

- A central database stores all aspects of the design and generate different design views on demand:
  - Type-, format of a view
- initially built from the input description
- query the database for types of components, their functionality:
  - area, delay, performance, layout height, width and shape
- supply a component for any given functionality and constraint

Technology Independence

- On the layout level: technology independence can be achieved by laying dimensionless objects on a virtual grid
- Requires a change in the technology file containing spacing rules
- Requires sophisticated compaction algorithms with local optimization
- on the logic level: synthesizing design with generic gates and then mapping generic gates into library components by performing local transformations

Design Learning

- When a different ASIC library is used it is necessary to redesign higher level components to take advantage of the new library.
- learning technology-adaptation rules

Design Synthesis Complexity

- system level synthesis to translate a set of communicating processes into register-transfer components
- component synthesis to translate component descriptions into layouts

Theoretical background

- Turing-Church Thesis:

if an algorithm exists then there is an equivalent Turing machine, recursively-definable function, or applicable λ-function, for that algorithm.

Szintézis során használt algoritmusok

- THE SYNTHESIS IN-CORE MODEL
- DATA-FLOW ANALYSIS
- AS-FAST-AS-POSSIBLE SCHEDULING
- MODULE ASSIGNMENT
- PATH-BASED ALLOCATION
- Estimation
- Logic Minimization

THE SYNTHESIS IN-CORE MODEL

- Magas-szintű nyelvből gráf reprezentáció
- Vezérlés folyam gráf (CFG)
- Adatfolyam gráf (DFG)

CFG

- Irányított gráf, CFG=(N,P)
- Csúcsok halmaza (N):
  - Hozzárendelés,
  - összeadás,
  - logikai műveletek, stb…
- Élek halmaza (P):
  - Precedencia relációk,
  - egymást követő utasítások
- Összefüggő gráf

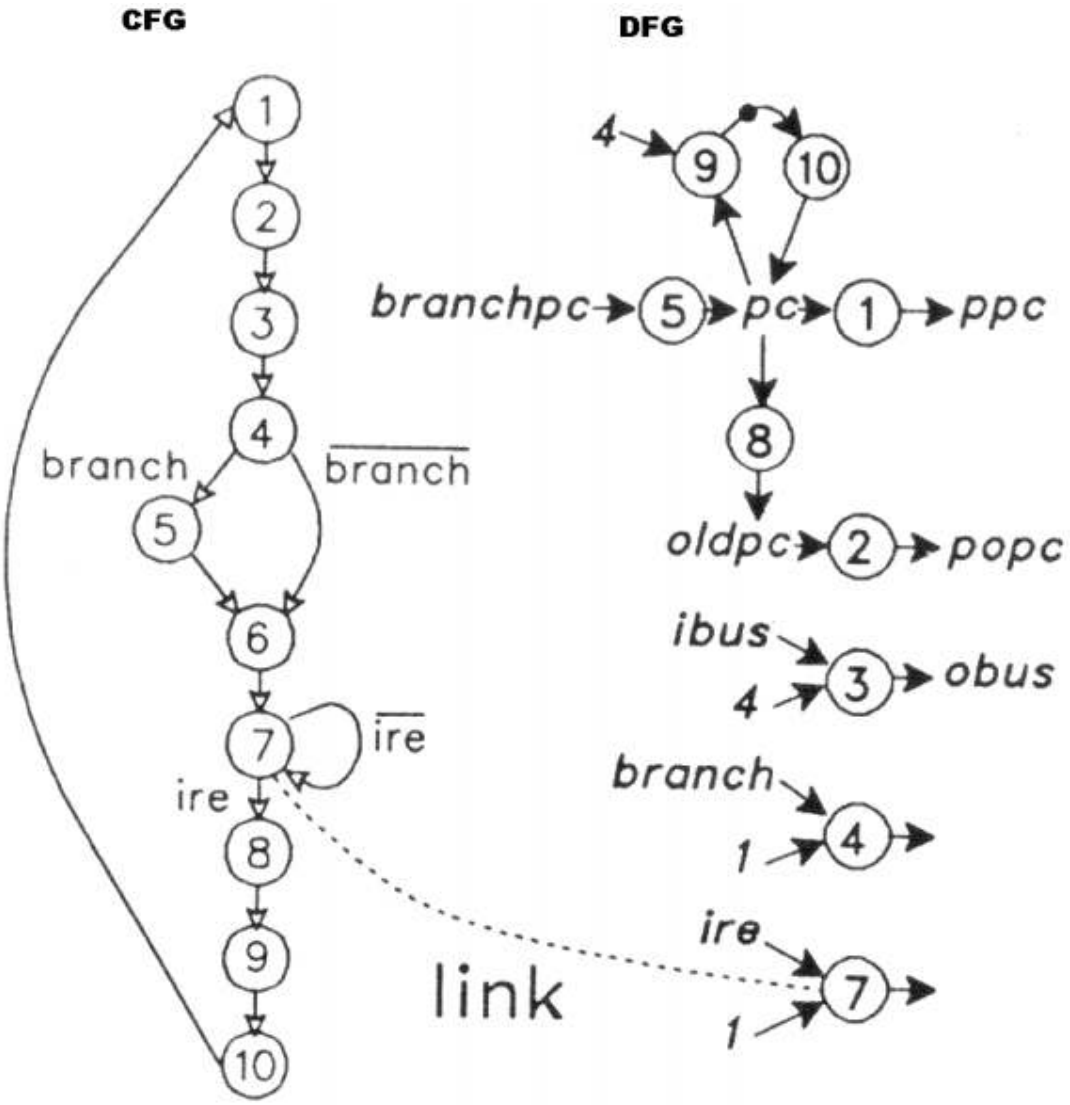
CFG

- Sorozat, szekvencia: egy él (n1,n2) eleme P-nek azt jelenti, hogy n2 következik n1 végrehajtása után
- Feltételes végrehajtás: egy művelet után egy másik művelet, ha a feltétel teljesül. A feltétel Boolean kifejezés, melynek értéke 1, akkor végrehajtódik, különben 0.
- Iteráció: Ciklusok, amelyek a folyamat iteratív viselkedését jelzik (loop).

DFG

- Irányított gráf, DFG=(N ∪ V,D)
- Csúcsok halmaza:
  - N: utasítások
  - V: változók
- Élek halmaza:
  - Adatkapcsolatok
- Nem feltétlenül összefüggő gráf

Példa




```
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;
```

IBM Magas-szintű Szintézis

- The High-level IBM Synthesis system (HIS).
- HIS uses a design representation called the SSIM (Sequential Synthesis In-core Model)
- The SSIM represents control and data separately. Before high-level synthesis, the SSIM contains only the behavior of a design, i.e., a control-flow graph (CFG) and a data-flow graph (DFG).

- Scheduling consists basically in control synthesis and adds a control finite state machine (FSM) to the SSIM. The synthesized FSM is represented by its state transition graph.
- Allocation consists basically in data-path synthesis and adds a data path to the SSIM. The synthesized data-path is also represented as a graph (netlist).
- The complete SSIM resides in memory during high-level synthesis, thus achieving the necessary speeds for design space exploration and interactive design.

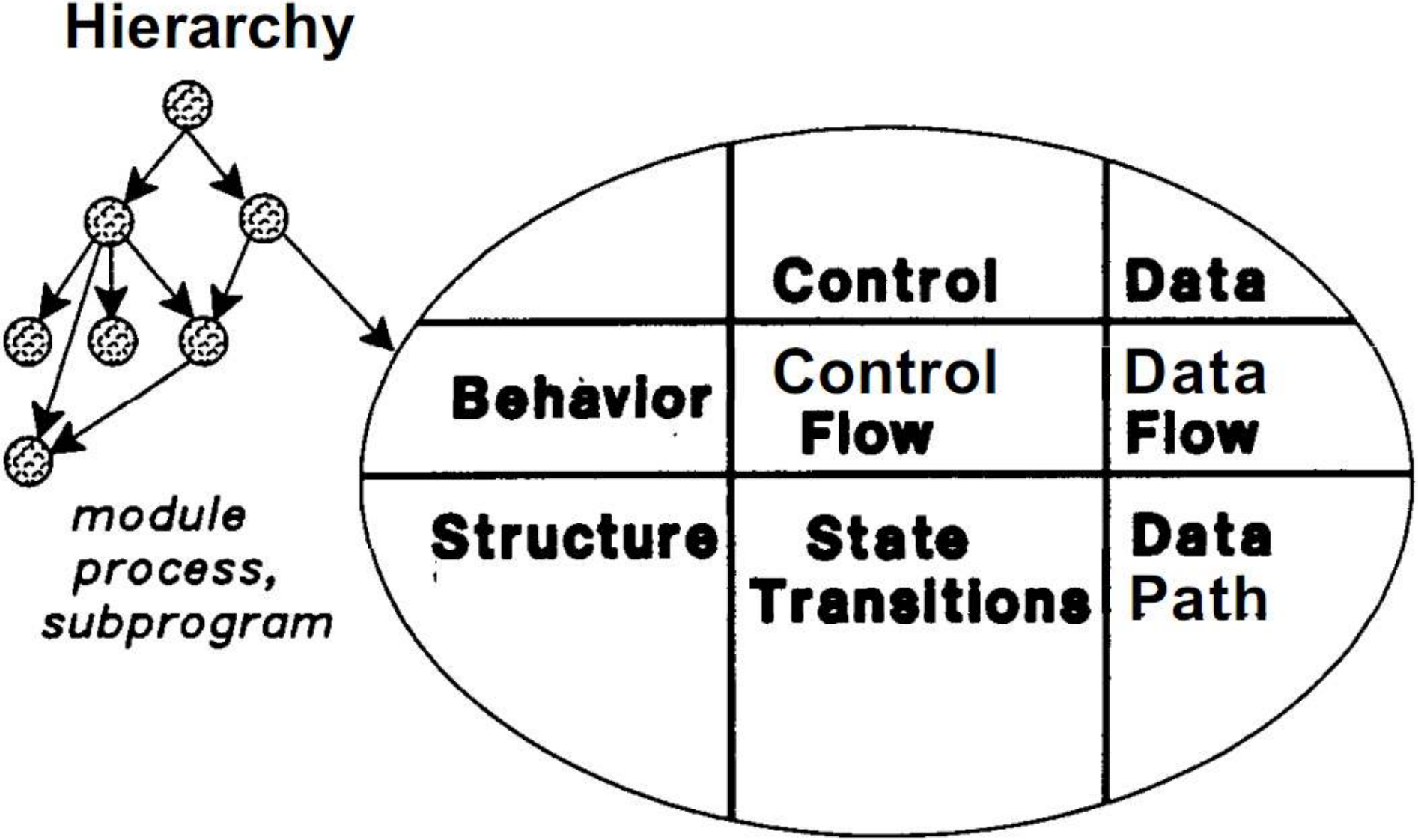


Figure 3. The Sequential Synthesis In–Core Model SSIM

DATA-FLOW ANALYSIS

- Data-flow analysis is a technique to determine the lifetime of values
- High-level synthesis often uses data-flow analysis to determine the intended behavior, for example, unfolding variables
- In HIS, lifetime information is used during allocation and scheduling

AS-FAST-AS-POSSIBLE SCHEDULING

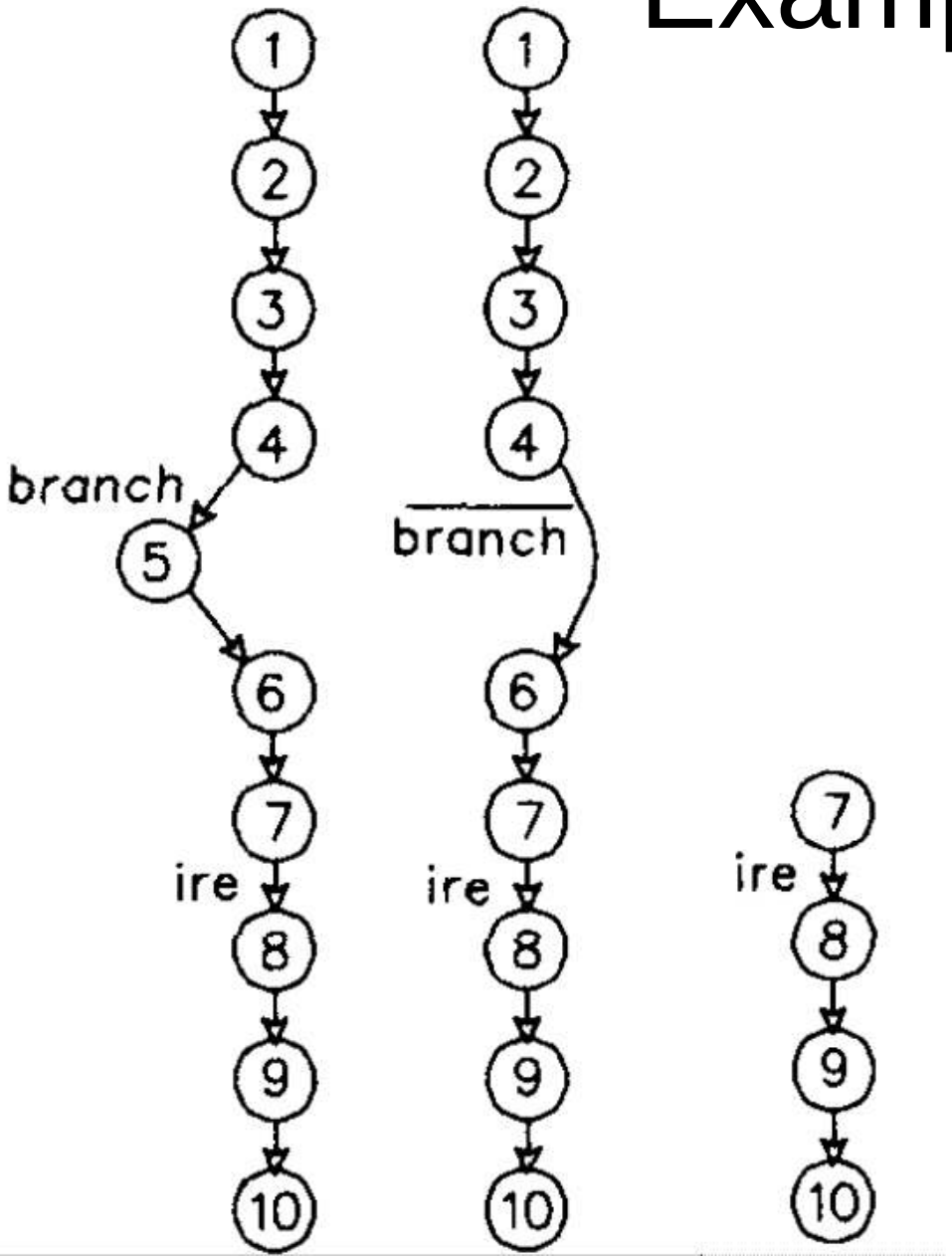
- HIS uses As-Fast-As-Possible (AFAP) scheduling for each possible path in the CFG.
- A path in the CFG is defined by conditional branches; thus one path represents one possible execution determined by the conditions on the conditional branches.
- Although the number of paths may grow exponentially with the number of nodes
- the number of paths represents the number of different functions in the circuit
- Optimizing the length of all paths (length in number of control steps), means minimizing the number of cycles in each instruction.

AS-FAST-AS-POSSIBLE SCHEDULING

- Paths are computed by performing a depth-first traversal of the acyclic CFG.
- The CFG is made acyclic by removing the feedback edges in loops
- Paths are computed starting at a given first operation

PATHS

Example



Constraints

- An implementation (and thus the schedule) is usually restricted by so called hardware constraints.
  - I/O ports can either transmit or receive only one distinct value per control step
  - registers can be written only once per control step
- Constraints explicitly specified often indicate area and delay characteristics of the desired implementation:
  - the maximum clock cycle
  - the maximum number and type of the functional units to be used

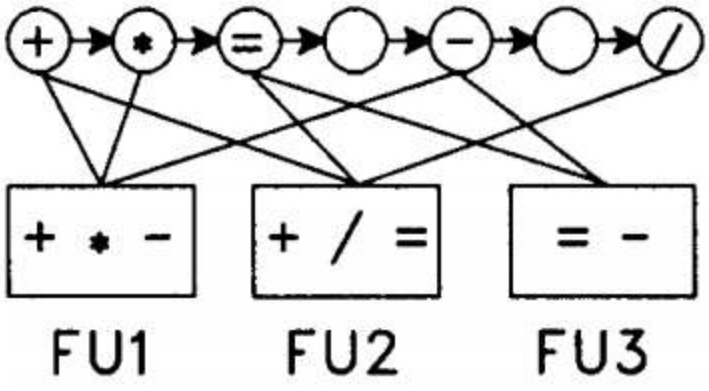
MODULE ASSIGNMENT

- The module assignment problem arises when operations can be implemented by more than one type of functional unit (FU).
- Module assignment selects the FU to implement each operation.

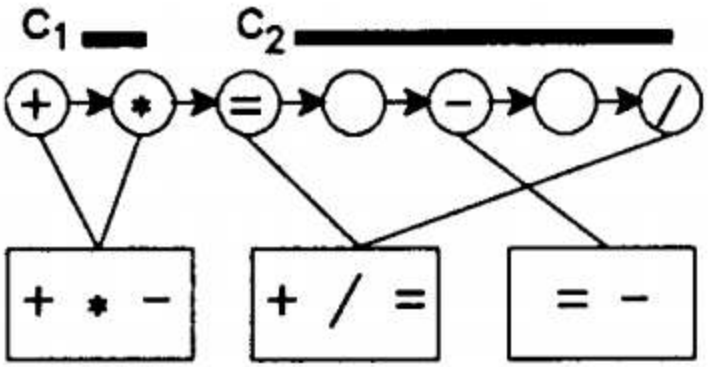
Example

```
T1    := IN1 + IN2;
T2    := IN3 * IN4;
T3    := (T2 = 15);
OUT1  := T3;
T4    := T1 - 1;
T1    := T4;
OUT2  := T1 / 4;
```

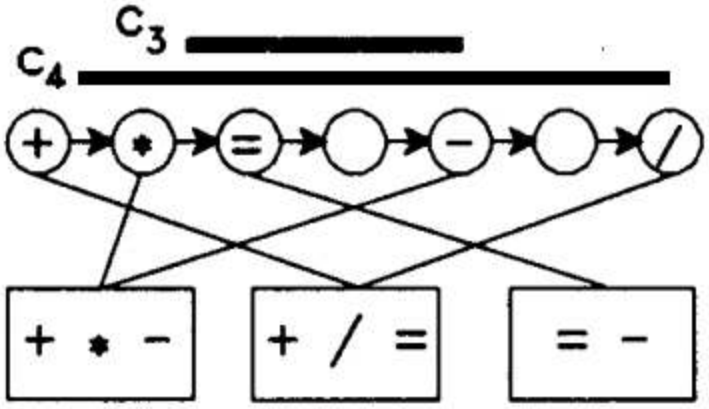
(a)



(b)



(c)



(d)

PATH-BASED ALLOCATION

- clique covering (or coloring) based allocation
- two separate steps:
  - a complete, functional initial data path is generated
  - then optimized

Initial Data-Path Allocation

- Allocating functional units
- Allocating registers
- Defining the interconnections (multiplexers) between functional units and registers
- Deriving the control signals:
  - load enable for registers
  - operation selection for functional units
  - selection for multiplexers

Data-Path Optimizations

- Optimizations in the data path comprise register, functional unit and multiplexer merging
- First a global conflict graph is generated
- The conflict graph is then colored to determine the smallest amount of hardware

Estimation

- For both constraint generation and data path optimization, it is necessary to estimate sizes and delays.
- it is extremely difficult at a high level.
- The models used are similar to typical logic synthesis delay models.

Logic Minimization

- Logic minimization within HIS is performed to optimize control signals on the fly

Output generation

- Output language generation is straightforward.
- Any language capable of representing netlists or RT-level hardware can be chosen.