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:

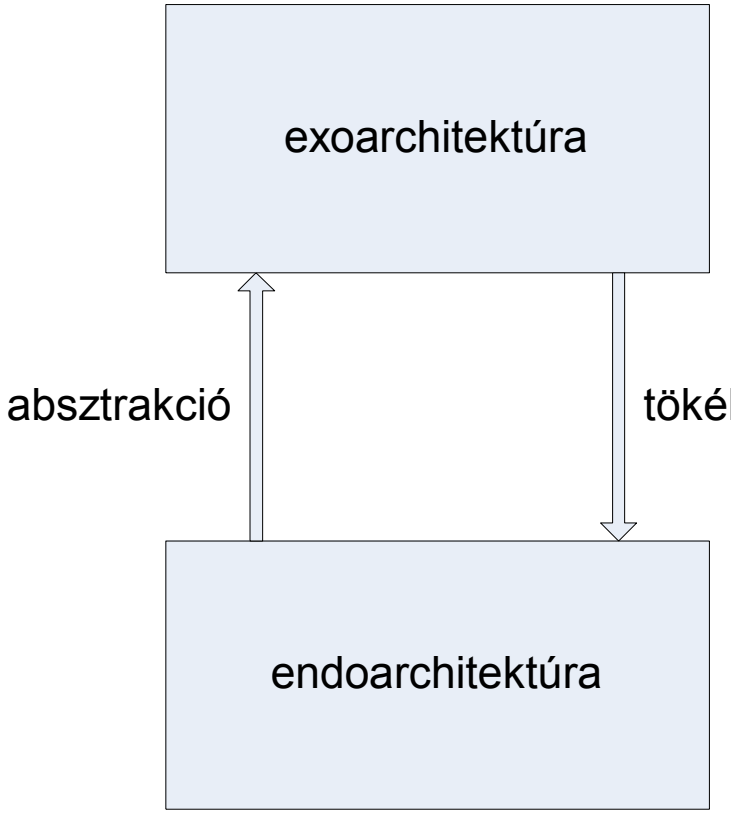
- Angol nyelvű könyv:
  http://www.knt.vein.hu
  → Oktatás → Tantárgyak → Digitális Rendszerek és Számítógép Architektúrák (nappali)
  (chapter1/…/8.pdf)
  - Bevezetés: Számítógép Generációk (chapter01.pdf)

- 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

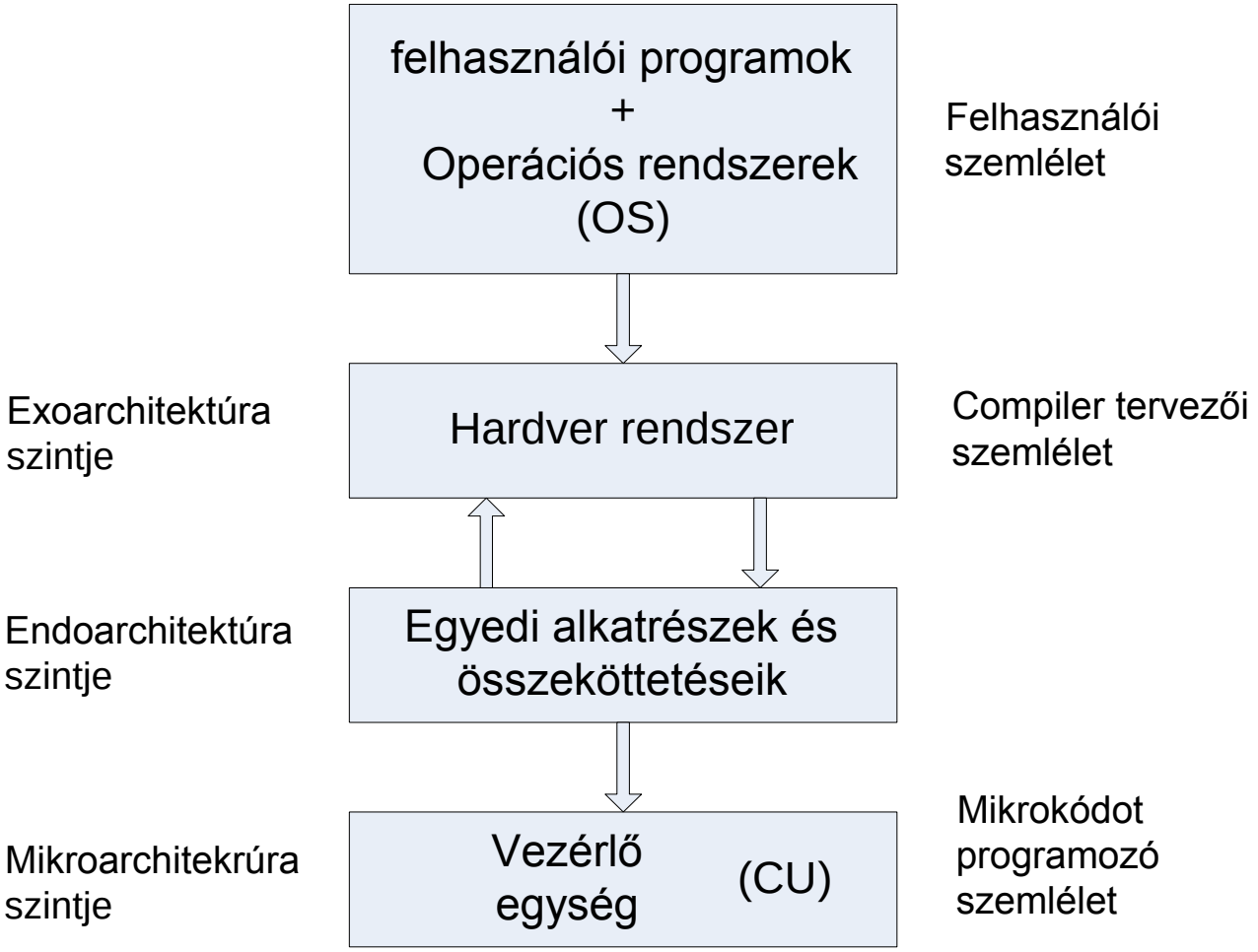
Exoarchitektúra – endoarchitektúra:



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

Vezérlő egység tervezői aspektusai:
- jelek
- komponensek

Számítógép architektúra definíciója:



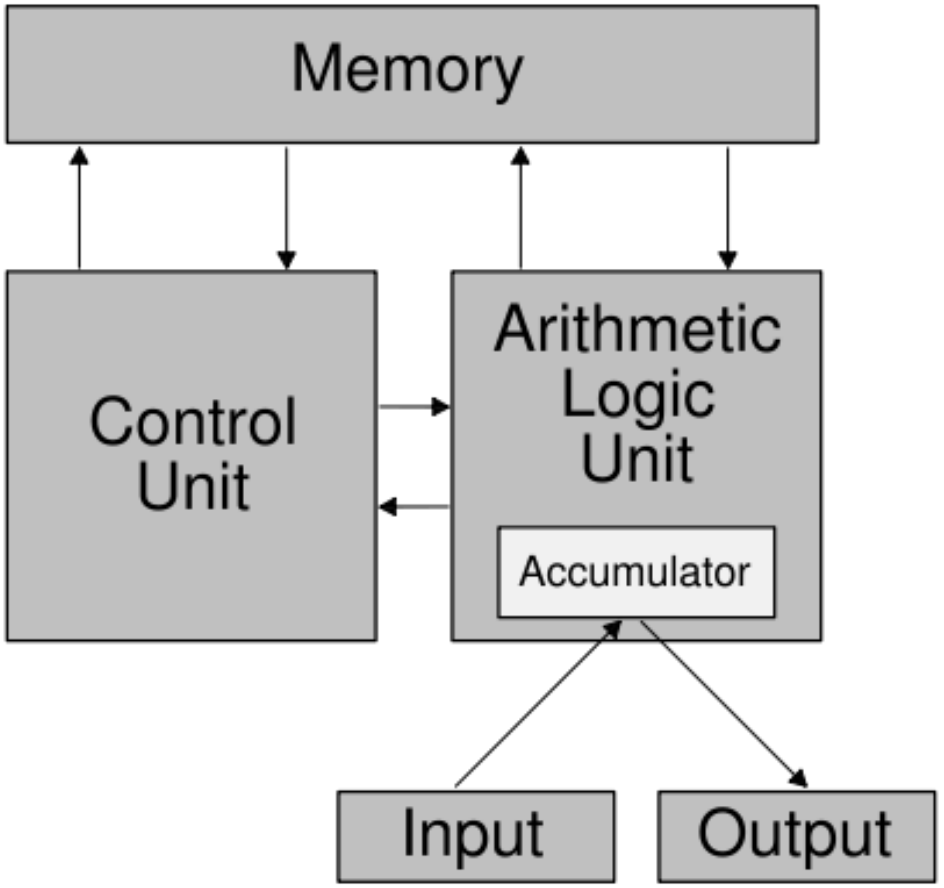
Számítógépes rendszerekkel szembeni tervezői követelmények:

- 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.)

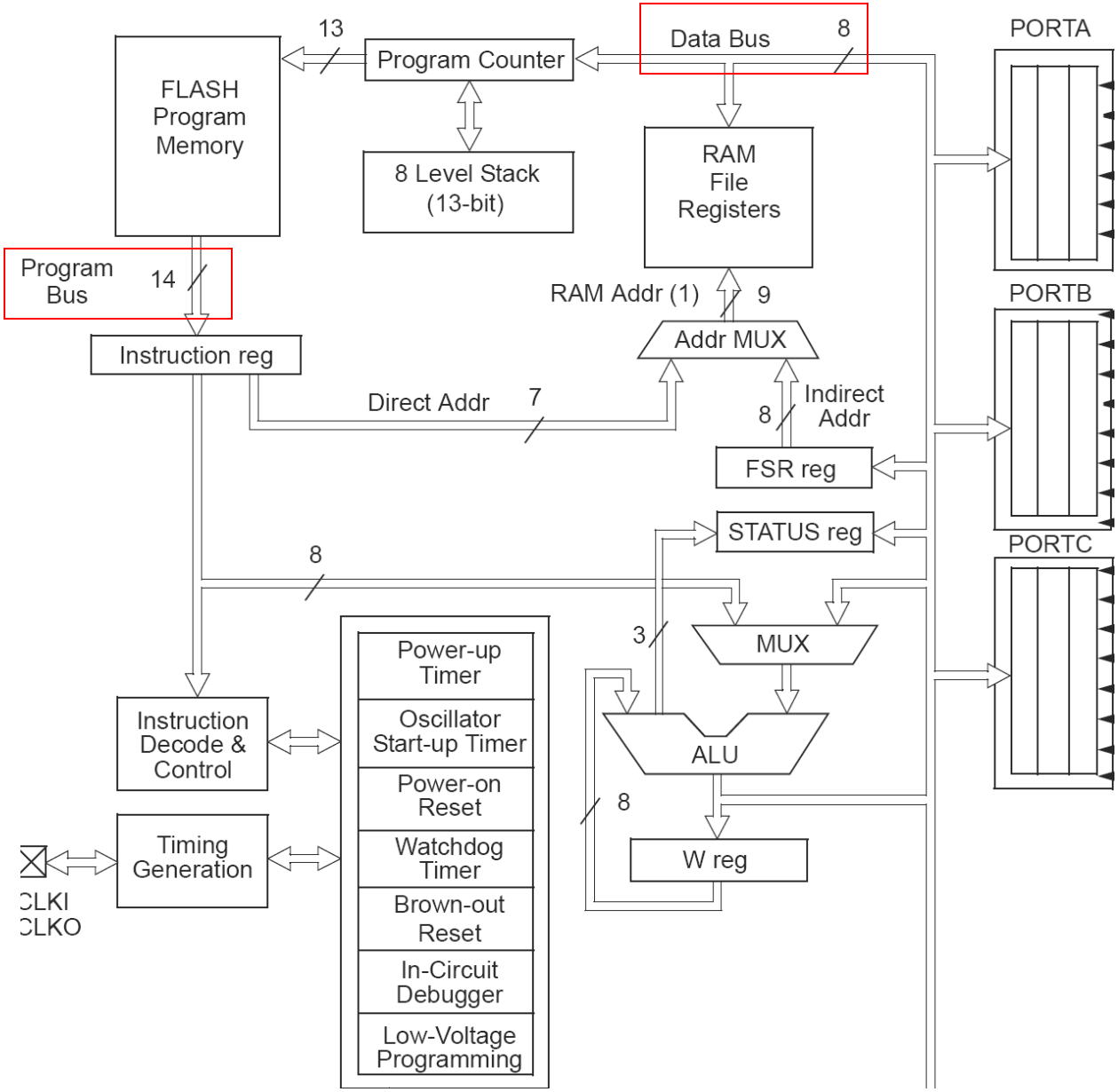
Neumann architektúra hátrányai

- „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

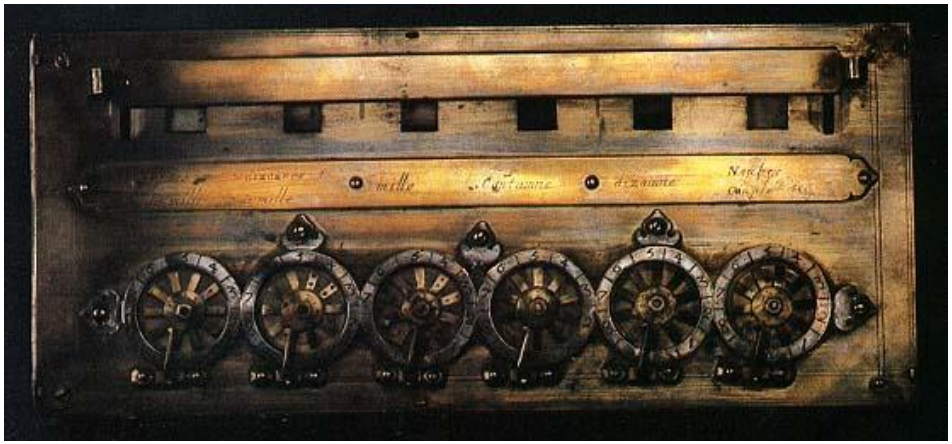
Harvard – Neumann együttes architektúra megvalósítás

- 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:

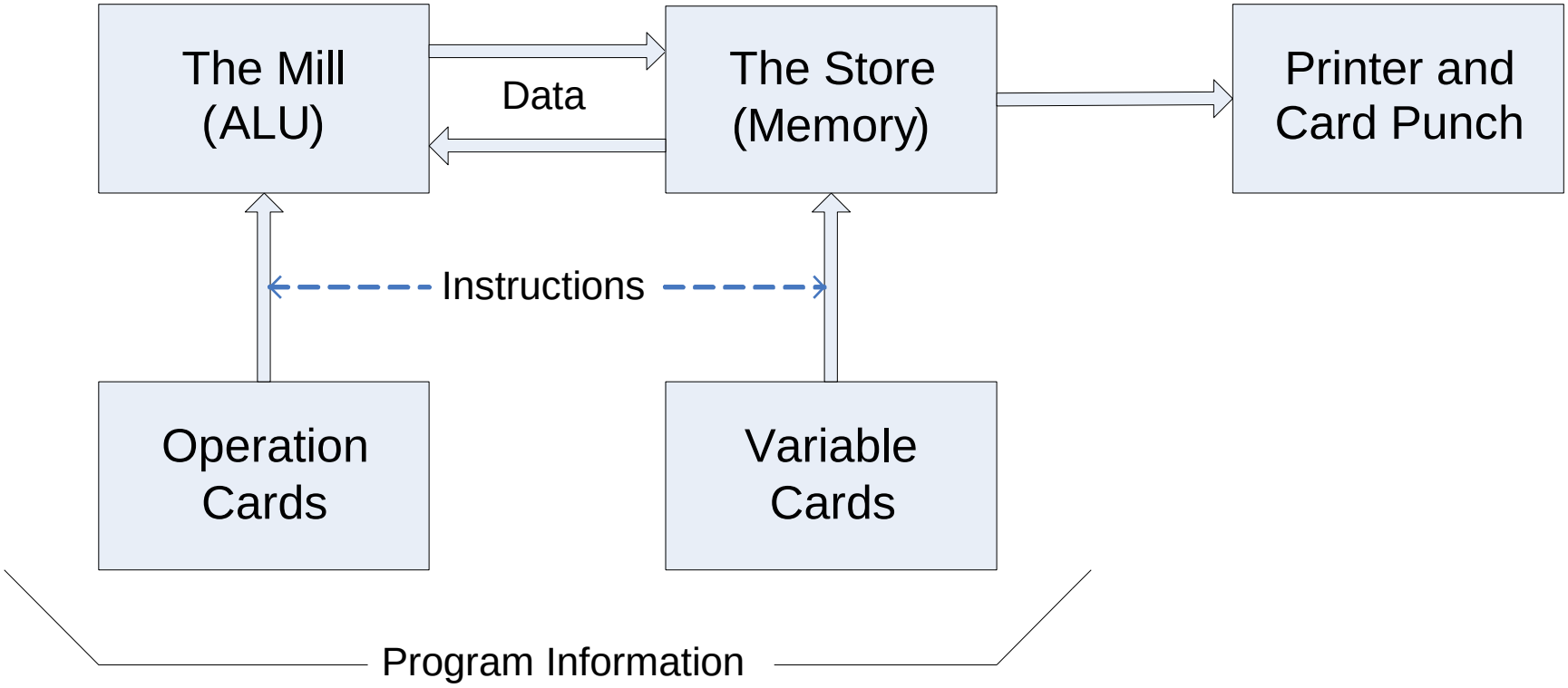
- 1642: Pascal – mechanikus kalkulátor (+,-)
- 1671: Leibnitz – kalkulátor 4 alapműv.



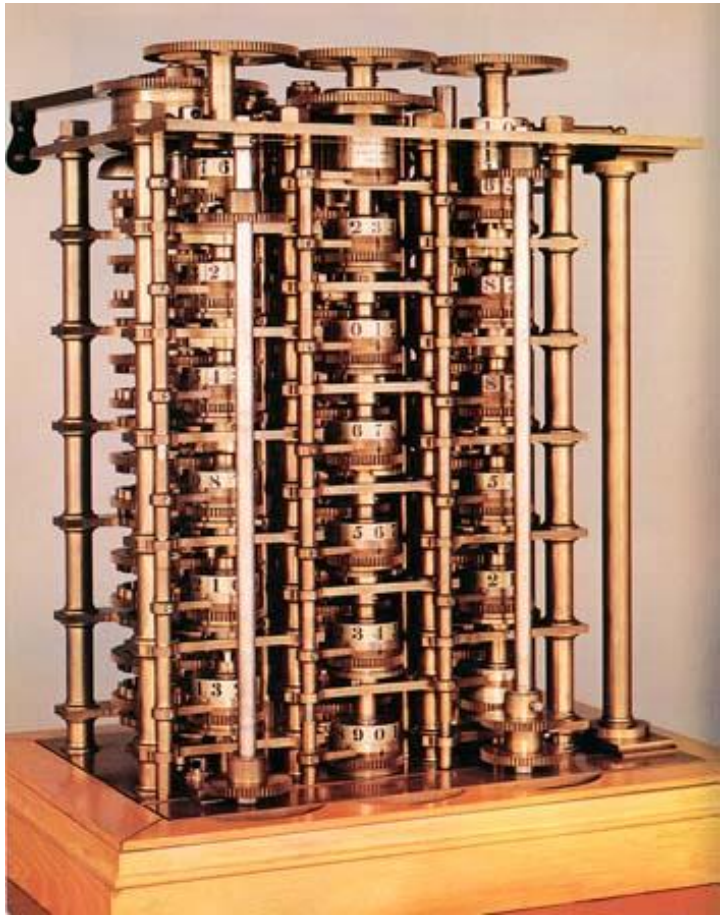
Eredet - korai számítási eszközök I (folyt.)

- 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.

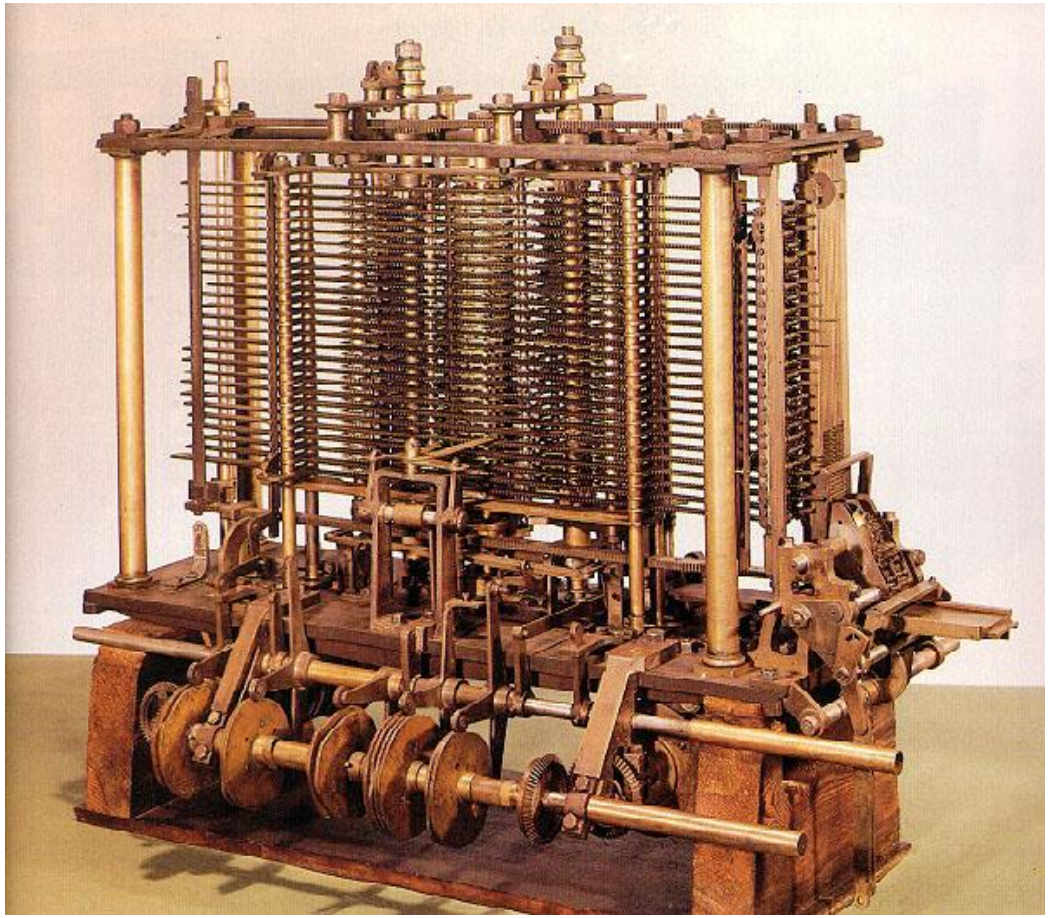
Babbage – Analitikus Gép



Babbage



Differencia gép

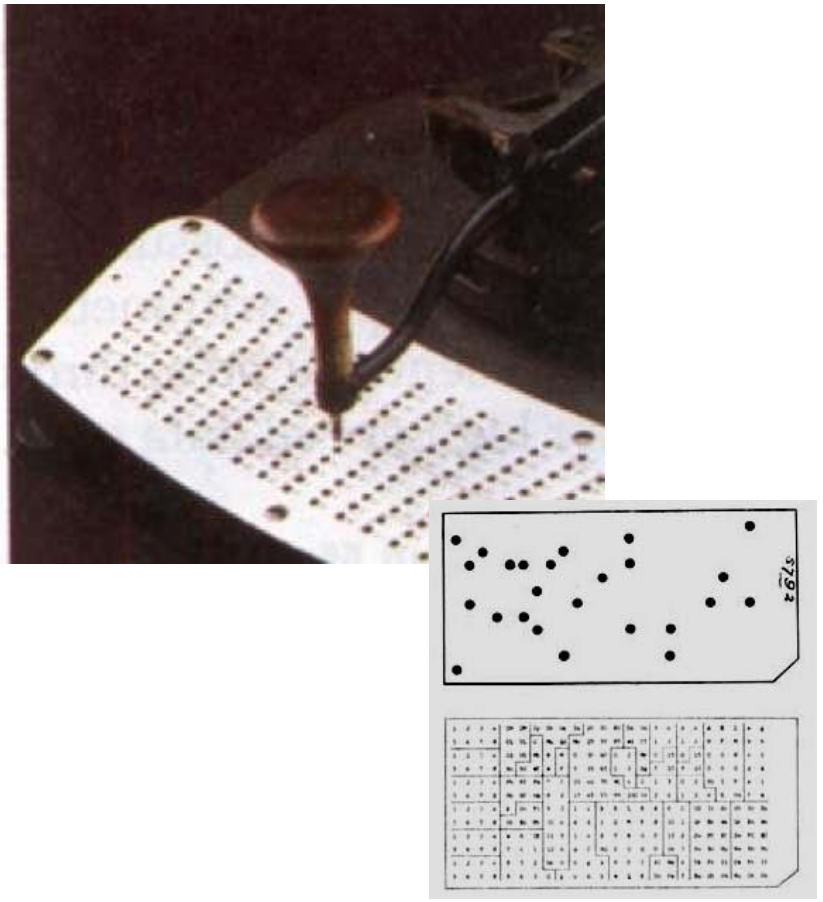


Analitikus gép

Eredet - korai számítási eszközök II (folyt.):

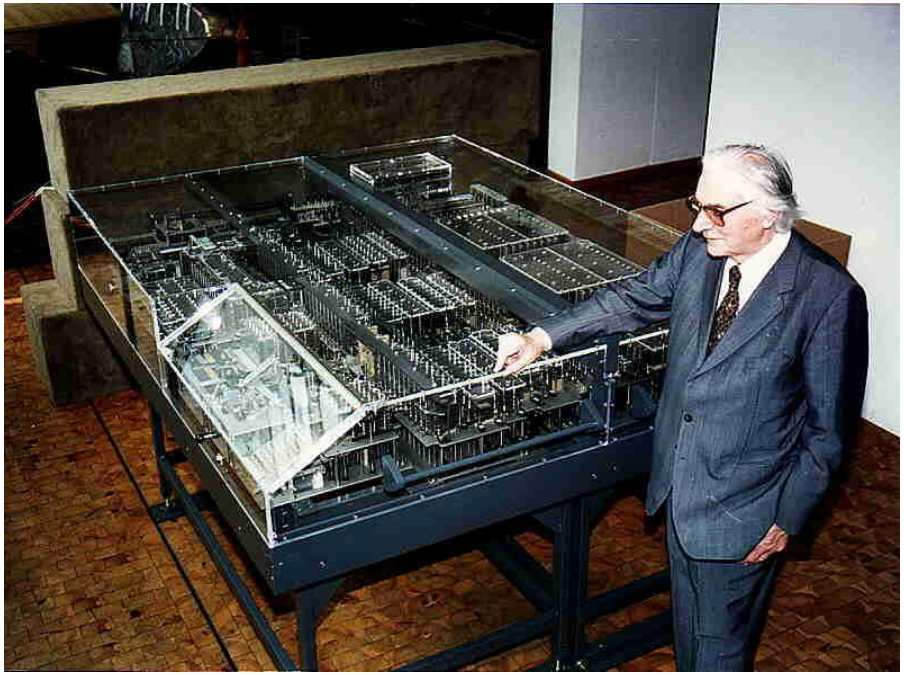
- 1801: Joseph Marie Jacquard: „loom” („szövőszék”) – „lyukkártya szerű” szalag, (számítási folyamat automatizálása)
- 1890: Hollerith – lyukkártya – US népszámlálás adatainak feldolgozására (1911 – IBM)
- 1930: Zuse: elektromechanikus gép
  - Z1: mechanikus relék, 2-es számrendszer!
  - Z3 (1941): első 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

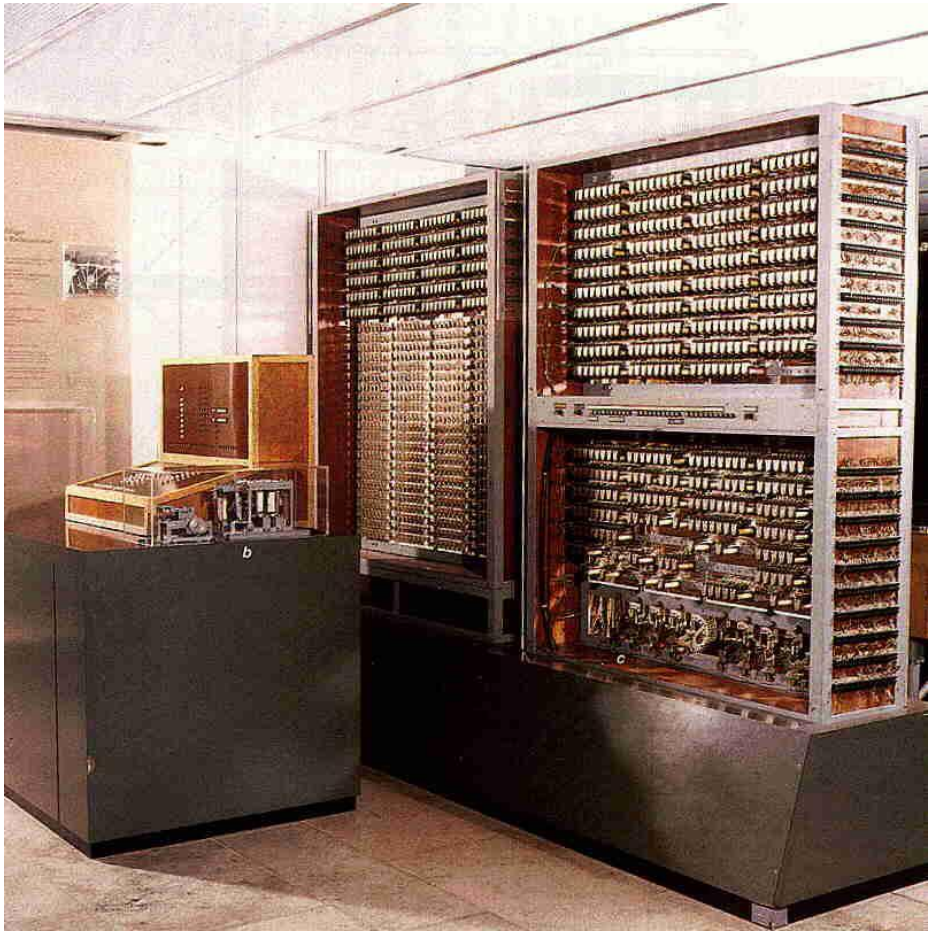
Jacquard „szövőgépe”



Hollerith - lyukkártya

Zuse Z1 és Z3

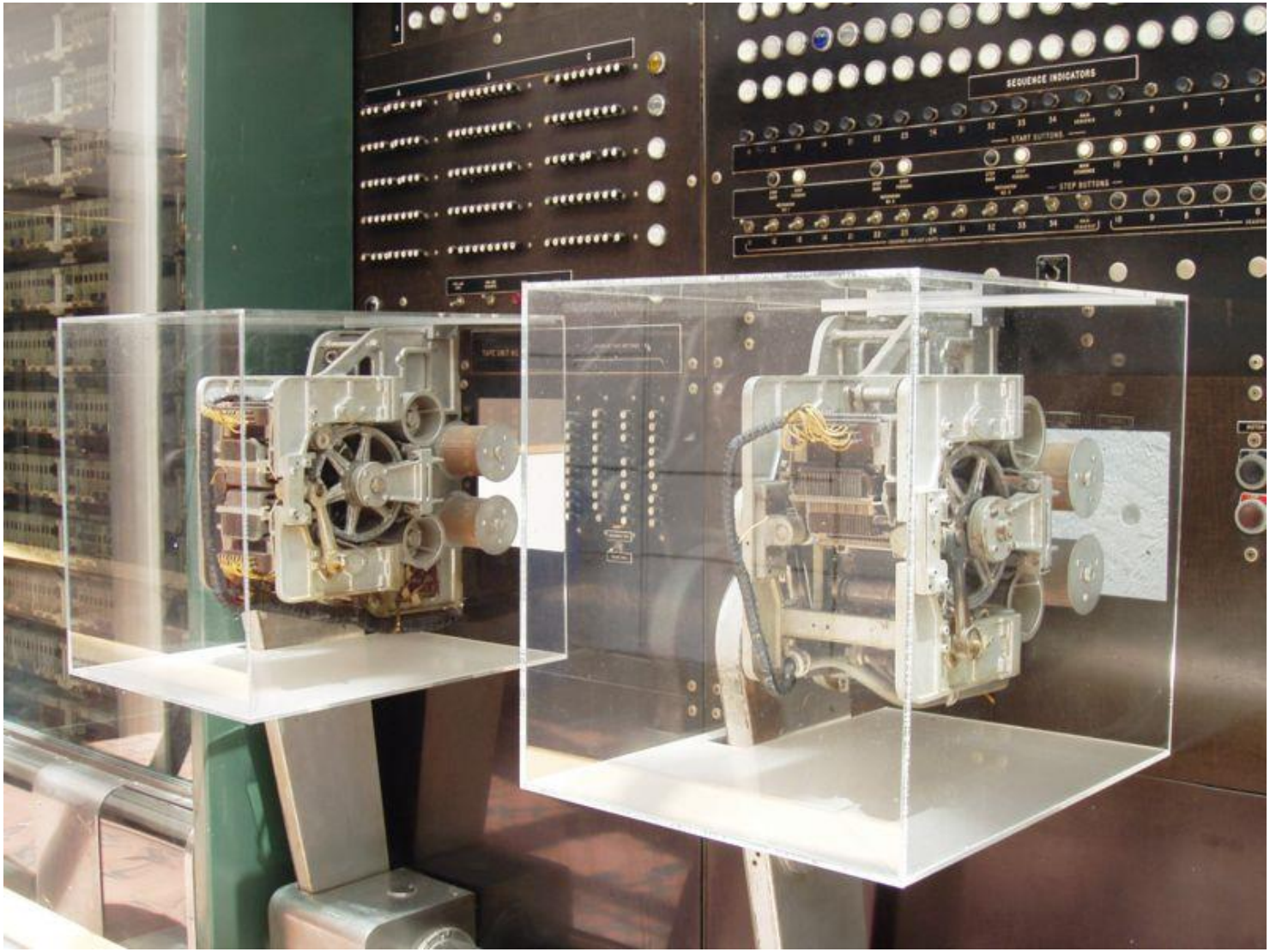




Harvard MARK I

- 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.):

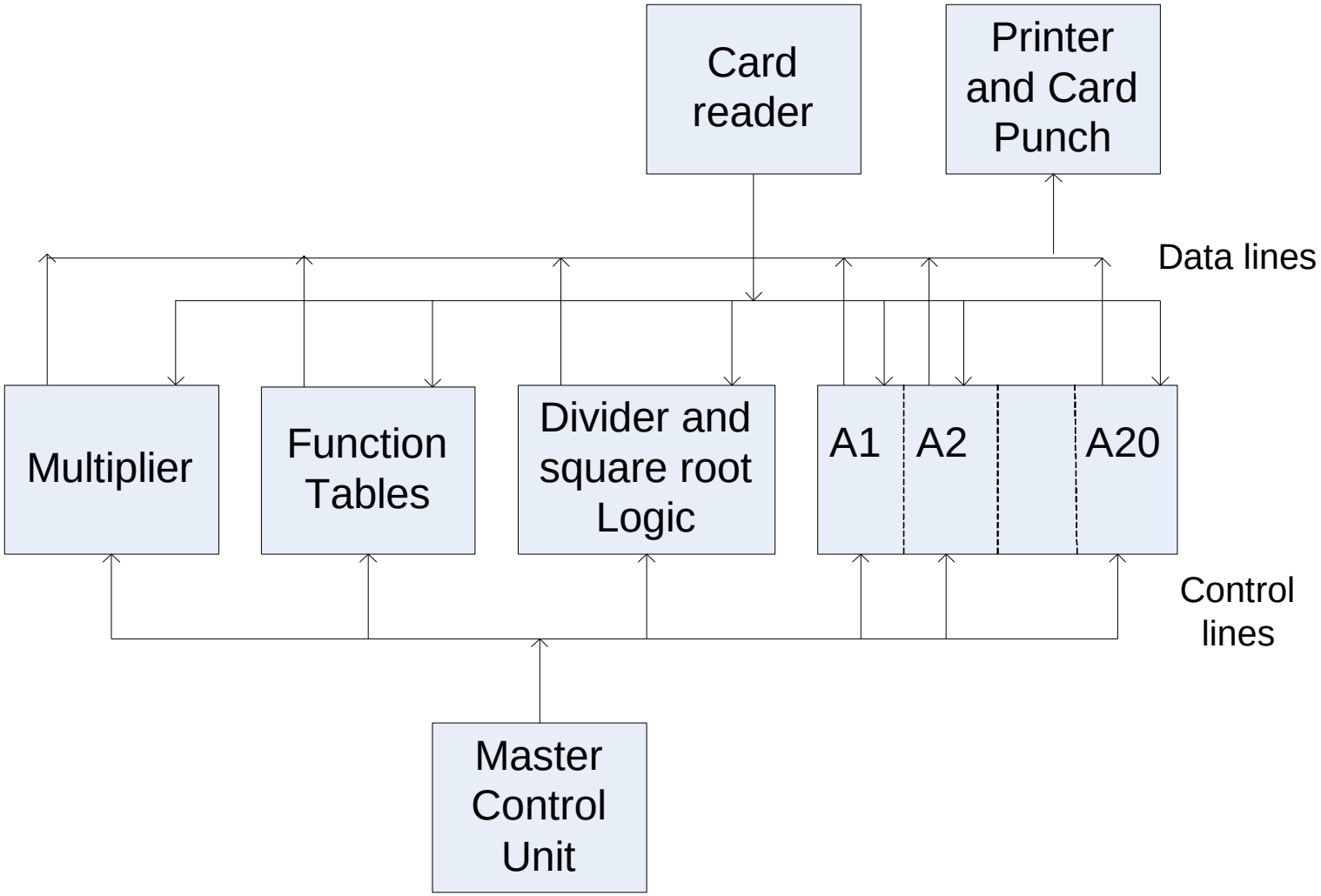
- 1937: Berry computer (Iowa Egyetem) – John Atanasoff első elektronikus számítógép rendszer
  - egyenlet rdsz.ek Gauss eliminációjára
  - 2-es számrendszer
  - Tárolás: kondenzátoron (mint DRAM-nál)
  - ALU: aritmetikai / logikai szeparáció
  - Részek teljes elkülönítése: memória, I/O perifériák, ALU

Atanasoff - Berry

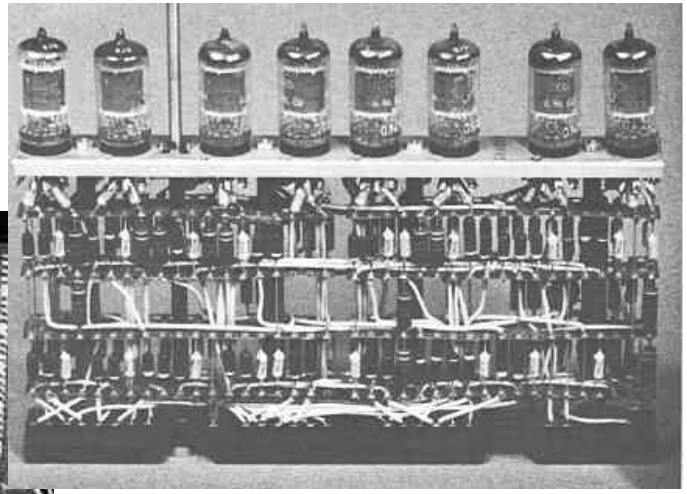
I. Generáció (1952-ig)

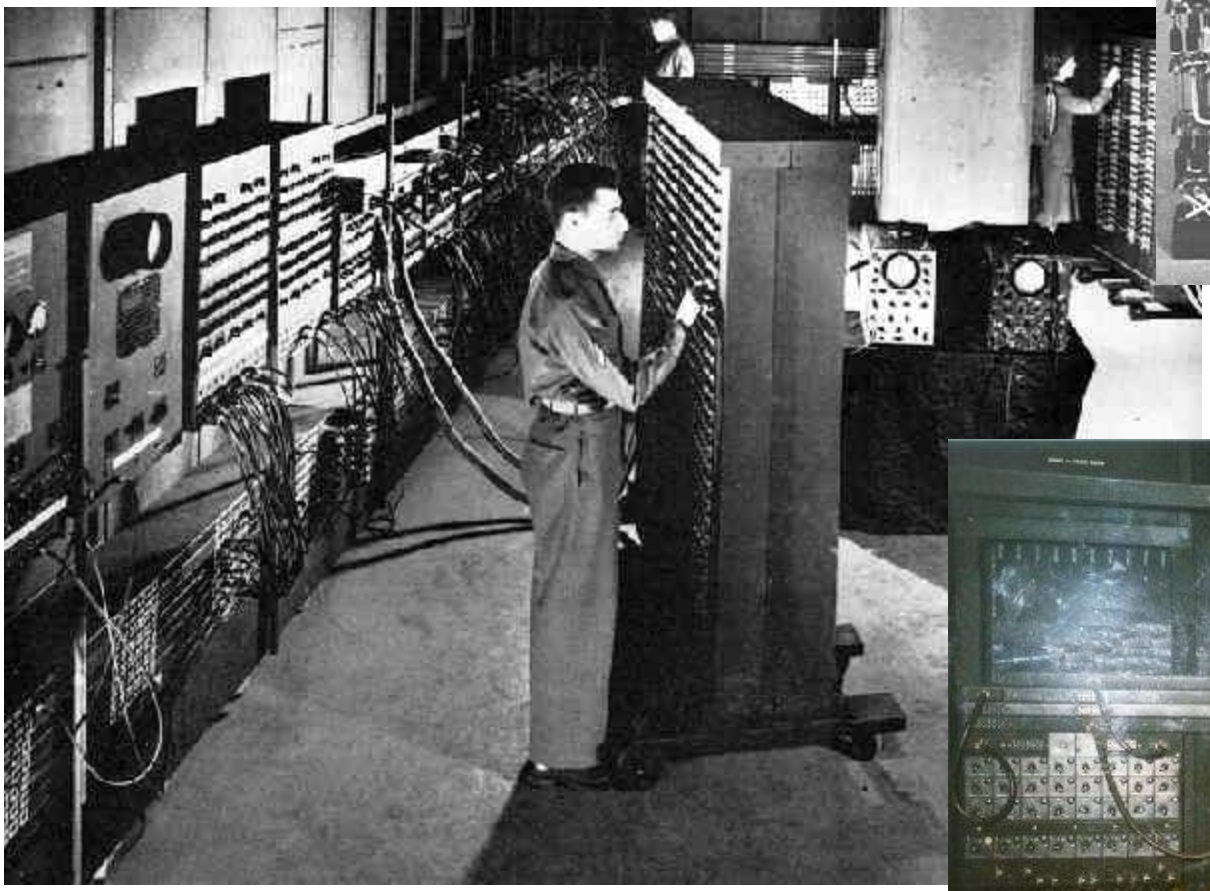
- 1943: **ENIAC**: elektromos numerikus integrátor és kalkulátor (Pennsylvania) Mauchly, Eckert
  - 18000 elektroncső, mechanikus, kapcsolók
  - Gépi szintű programozhatóság, tudományos célokra
  - Összeadás: 3ms
  - 20 ACC reg. – 10 jegyű decimális számra
  - 4 alapművelet + gyökvonás
  - Kártyaolvasó-író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

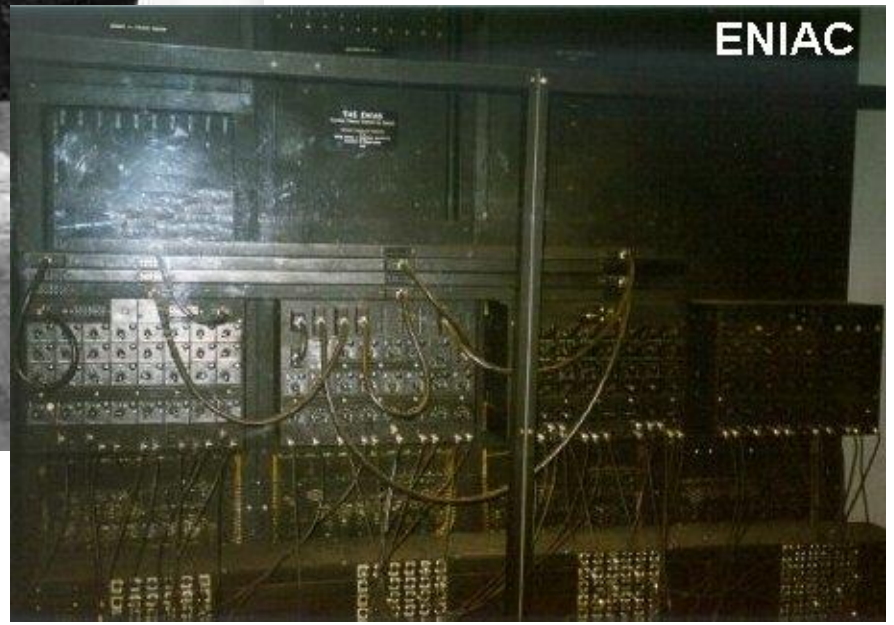
ENIAC


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



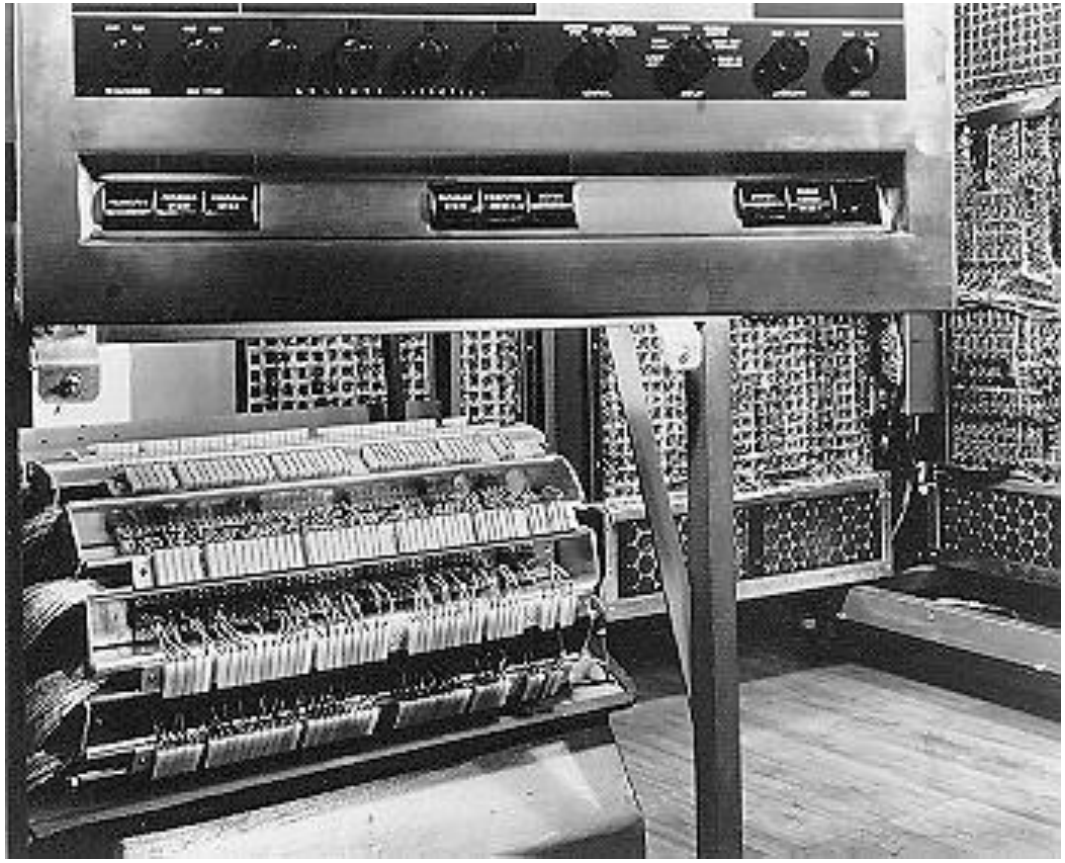




I. Generáció (folyt.):

- 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

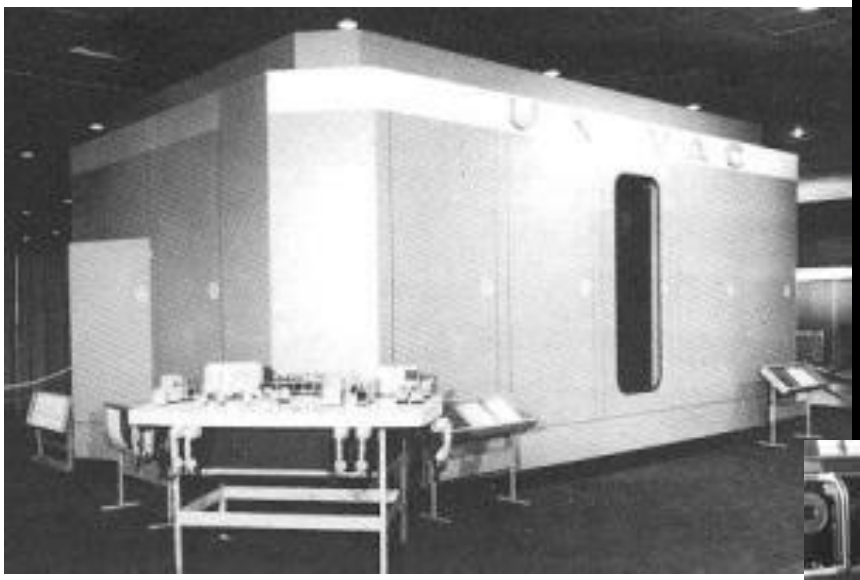


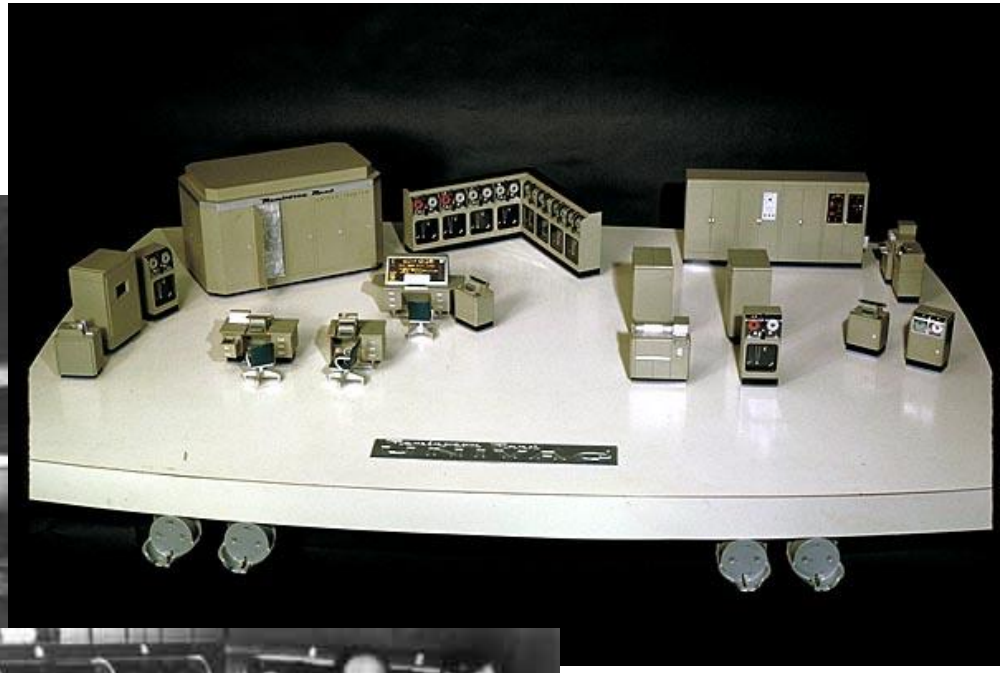
Neumann János

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

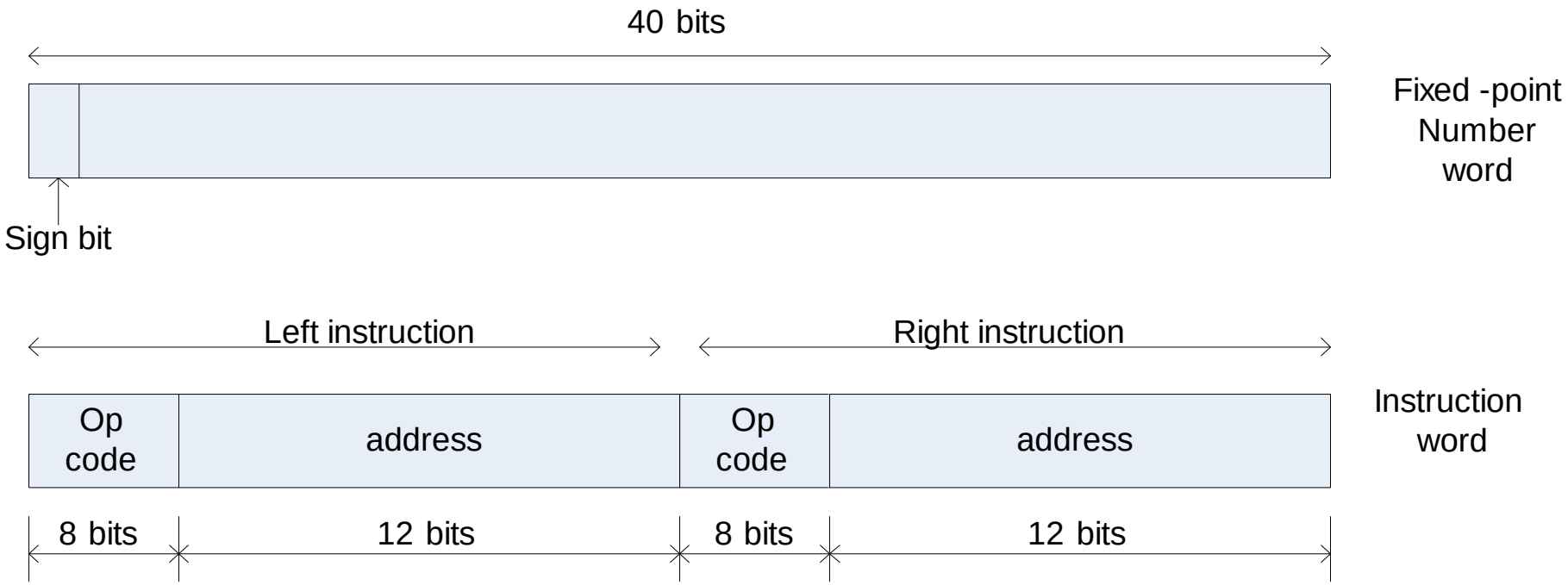




I. Generáció (folyt.):

- 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.

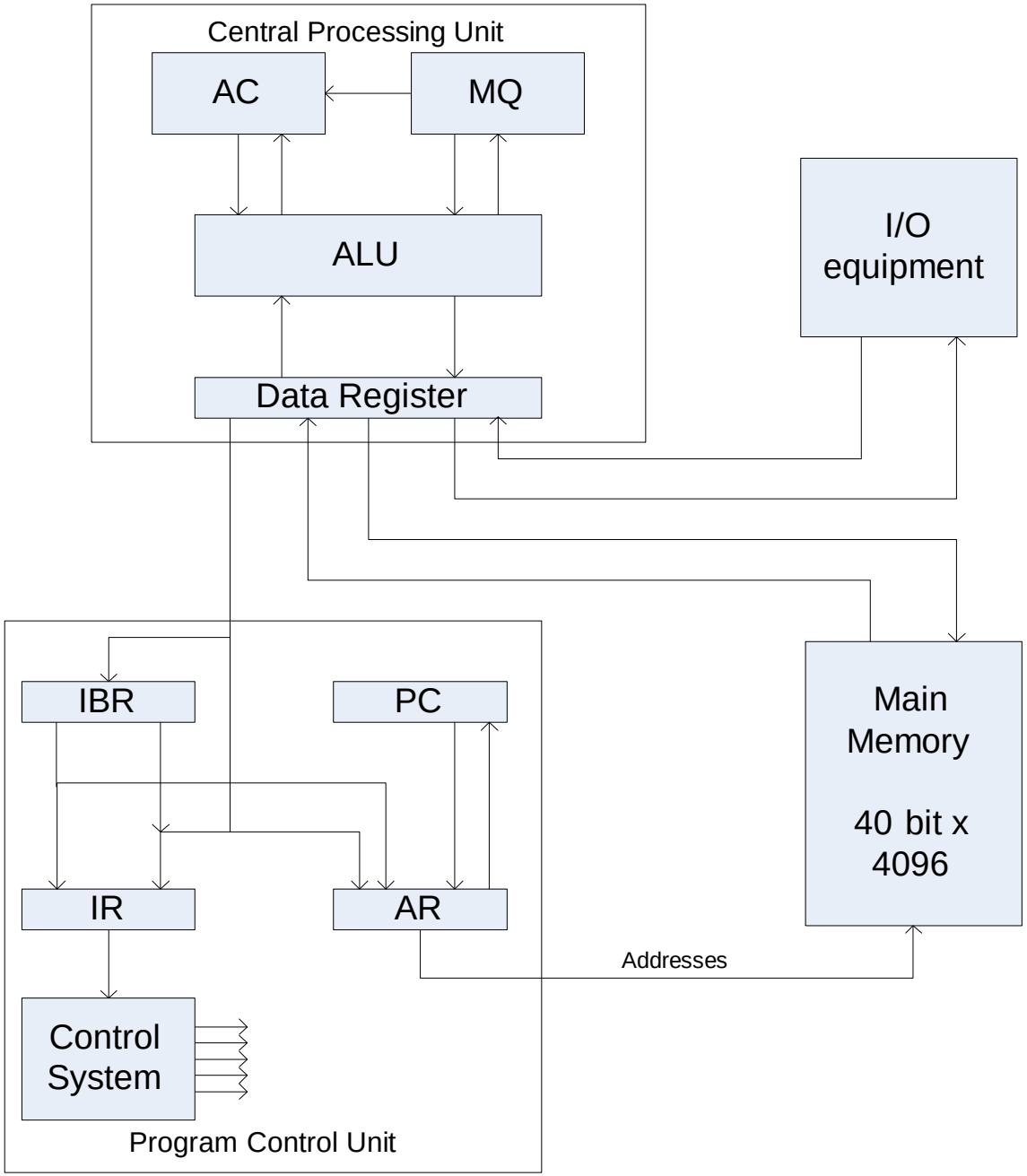
Arithmetic instructions

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.

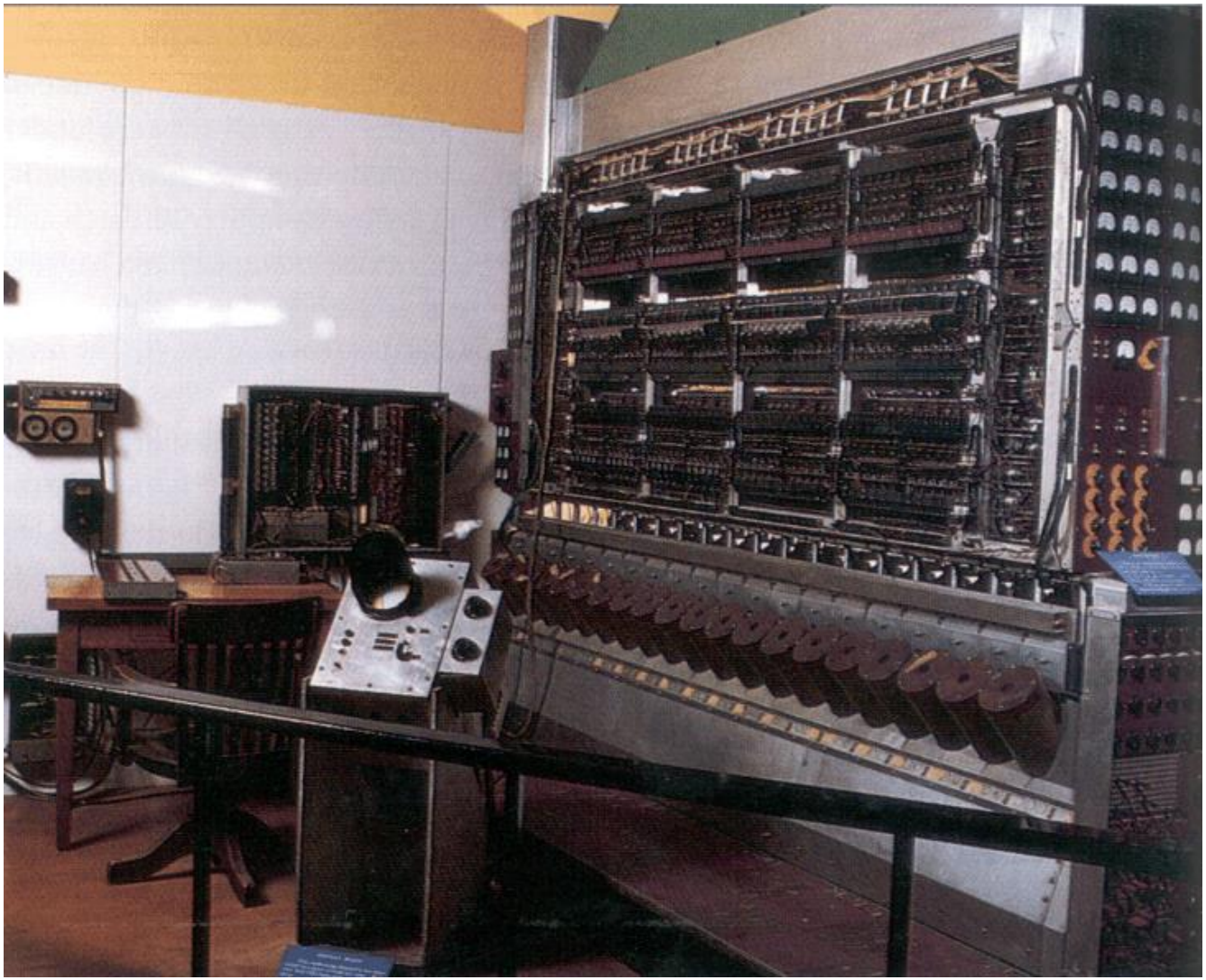
1.1 Táblázat: IAS utasítások (folyt.)

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



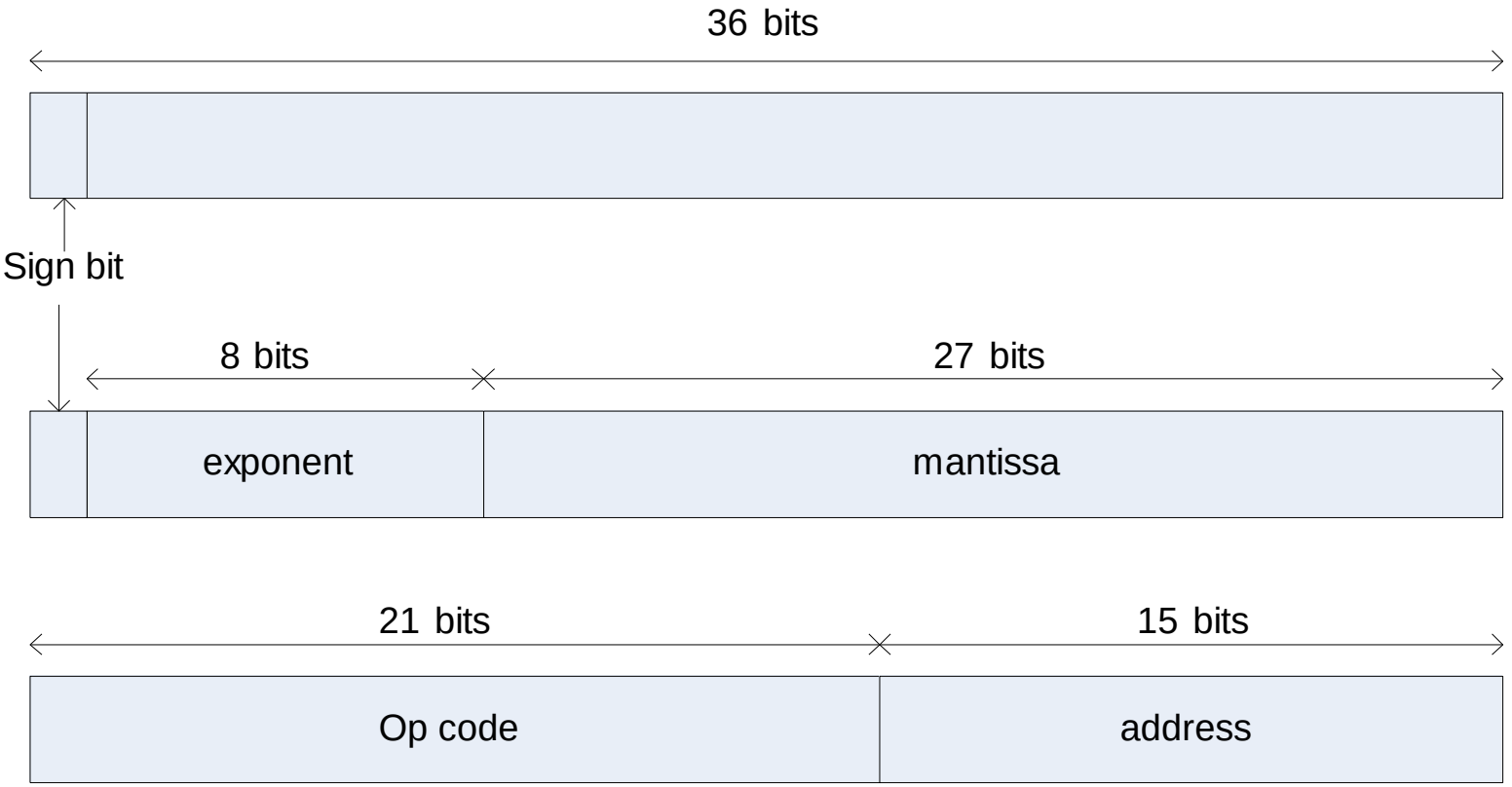
II. Generáció (1952-63):

- 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

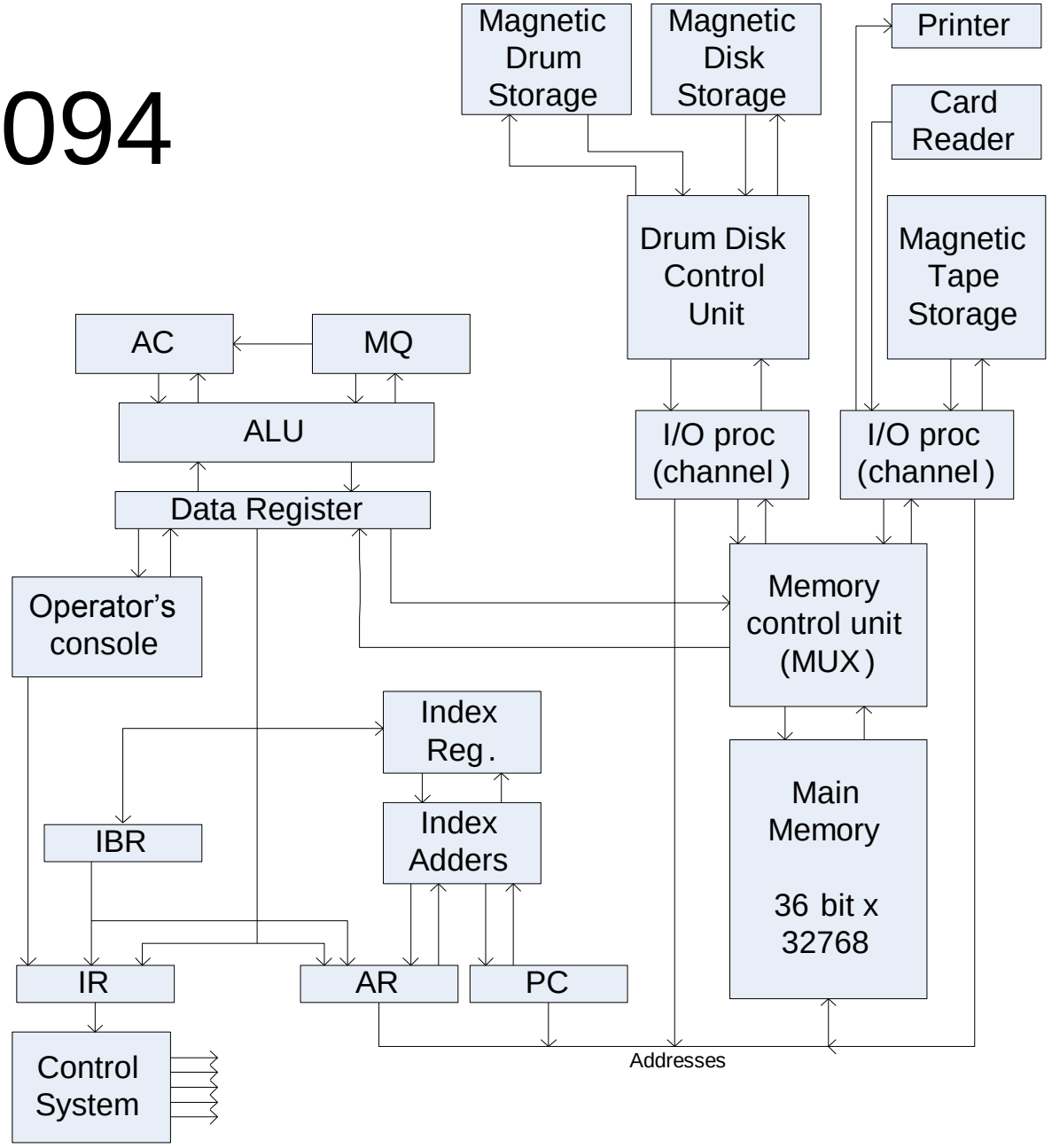
II. Generáció (folyt.):

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

IBM 709x adat- és utasítás formátum:



IBM 7094

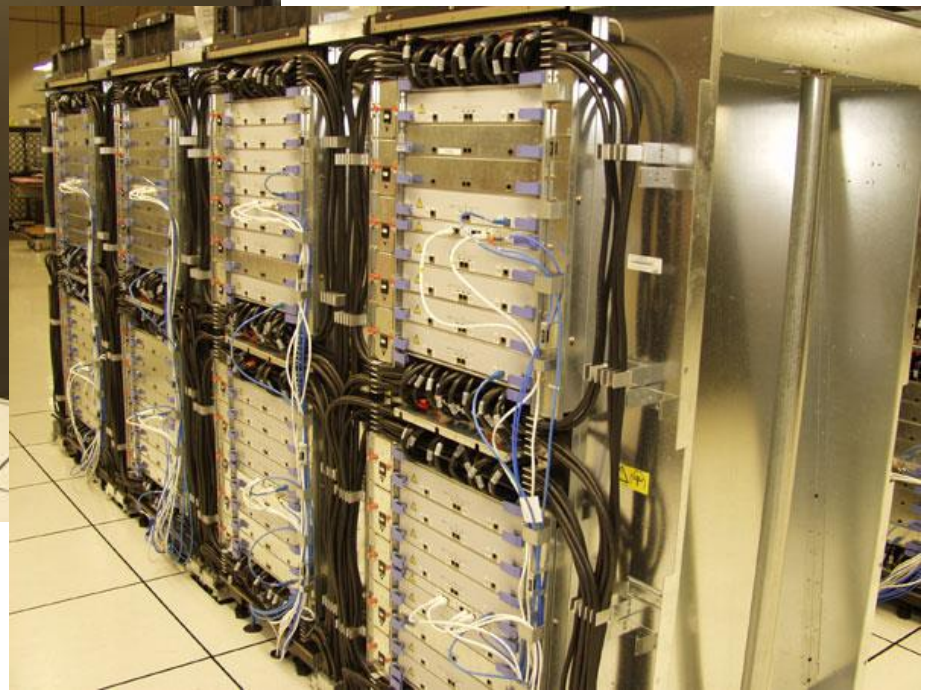

Magnetic Drum Storage
Magnetic Disk Storage
Printer
Card Reader
Drum Disk Control Unit
Magnetic Tape Storage
AC
MQ
ALU
I/O proc (channel)
I/O proc (channel)
Data Register
Operator's console
Memory control unit (MUX)
Index Reg .
Main Memory
36 bit x 32768
IBR
Index Adders
IR
AR
PC
Control System
Addresses

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



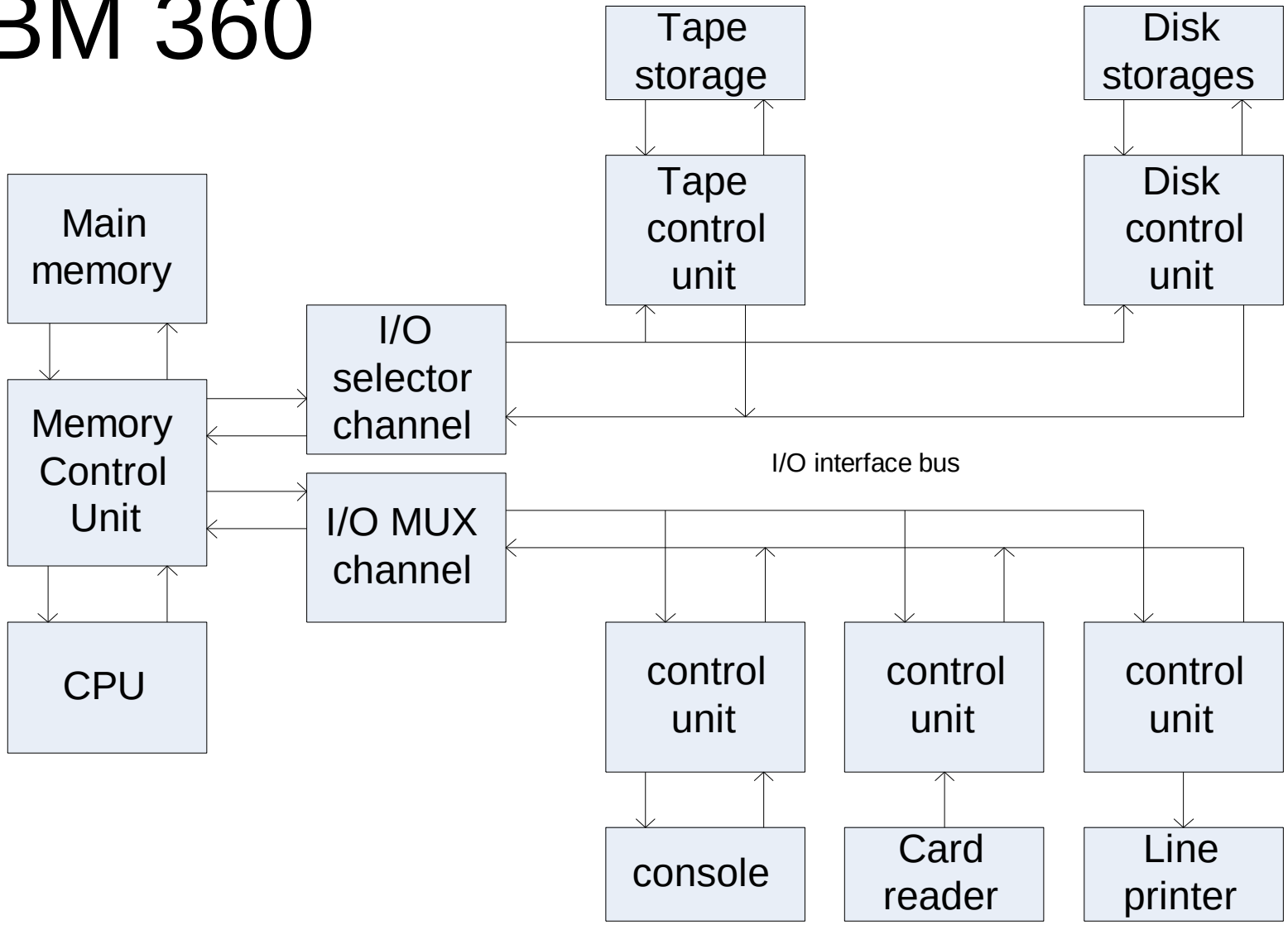
III. Generáció (1962-75):

- 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

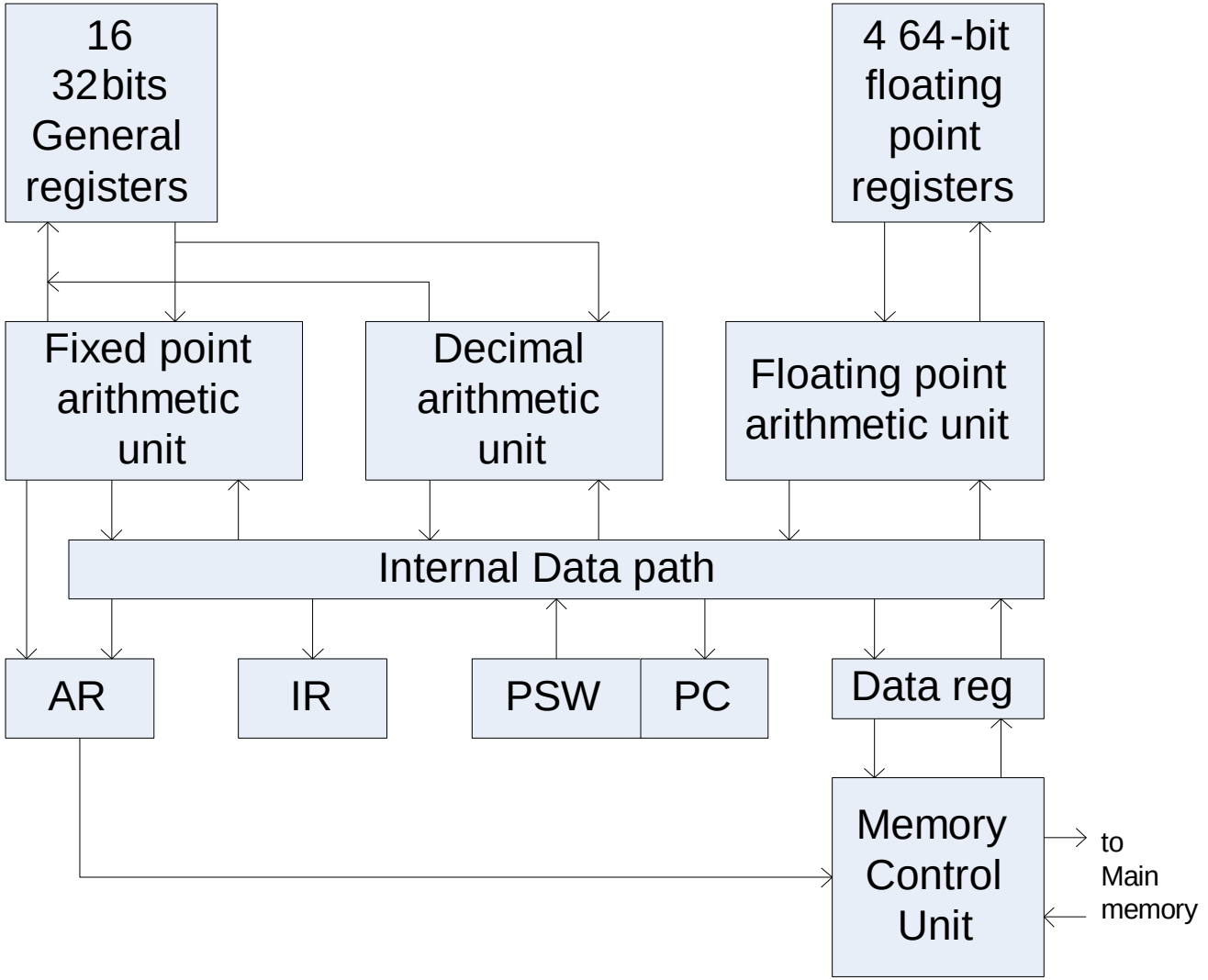
IBM 360

- 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)

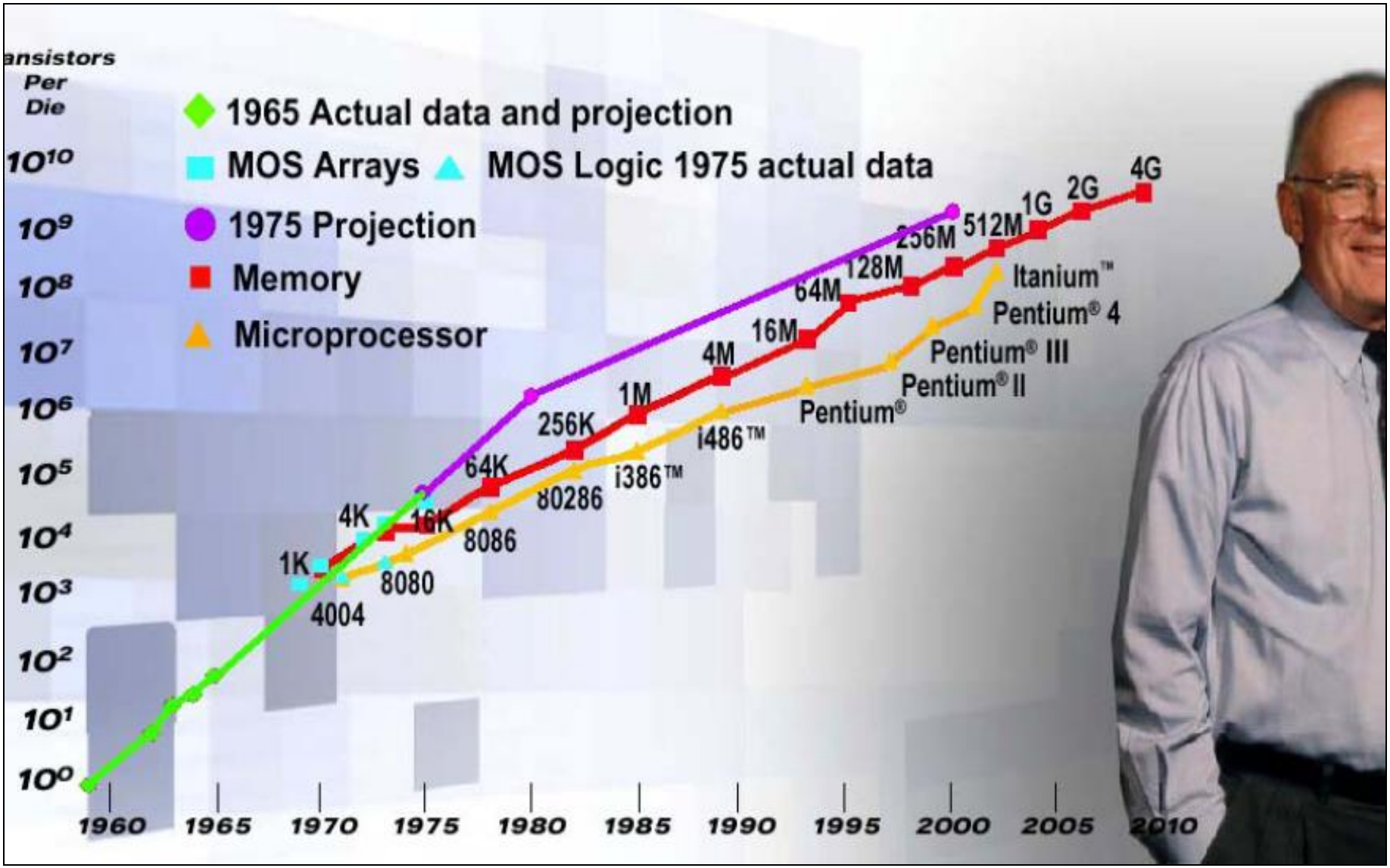
Year	Smallest feature [μm]	Dynamic Ram		Microprocessors			Wiring Levels / chip	I/O /chip
		Chip size [mm²]	Billions of bits / chip	Chip size [mm²]	Millions of transistors / cm²	On-Chip Clock (MHz)		
1995	0.35	190	0,064	250	4	300	4-5	900
1998	0.25	280	0,256	300	7	450	5	1350
2001	0.18	420	1	360	13	600	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	

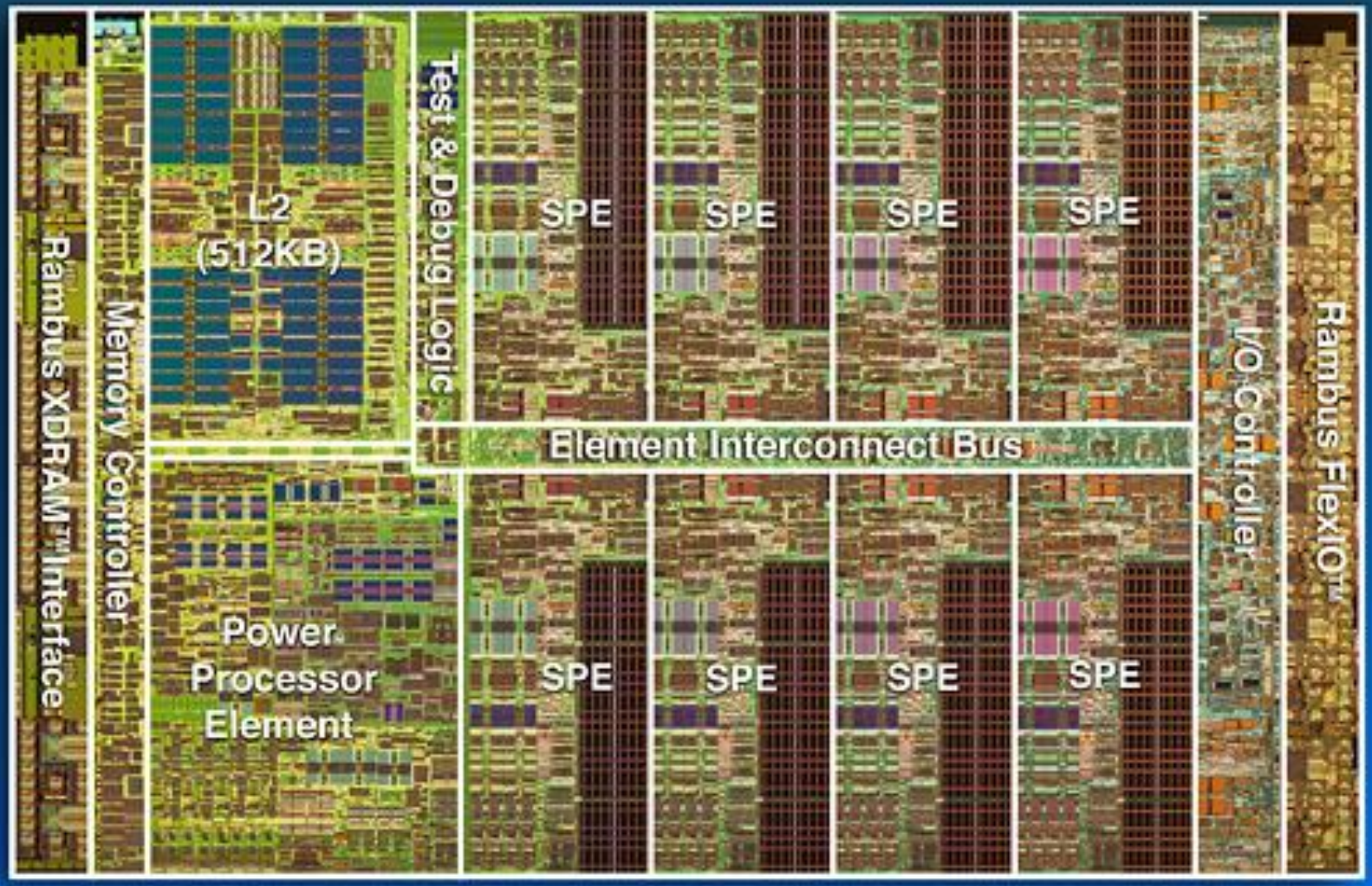
Órajelnövelés helyett Párhuzamosítás!

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

Pannon Egyetem
Képfeldolgozás és Neuroszámítógépek Tanszék

Digitális Rendszerek (BSc)

1. előadás: Logikai egyenletek leírása I.
Boole-algebra axiómái és tételei

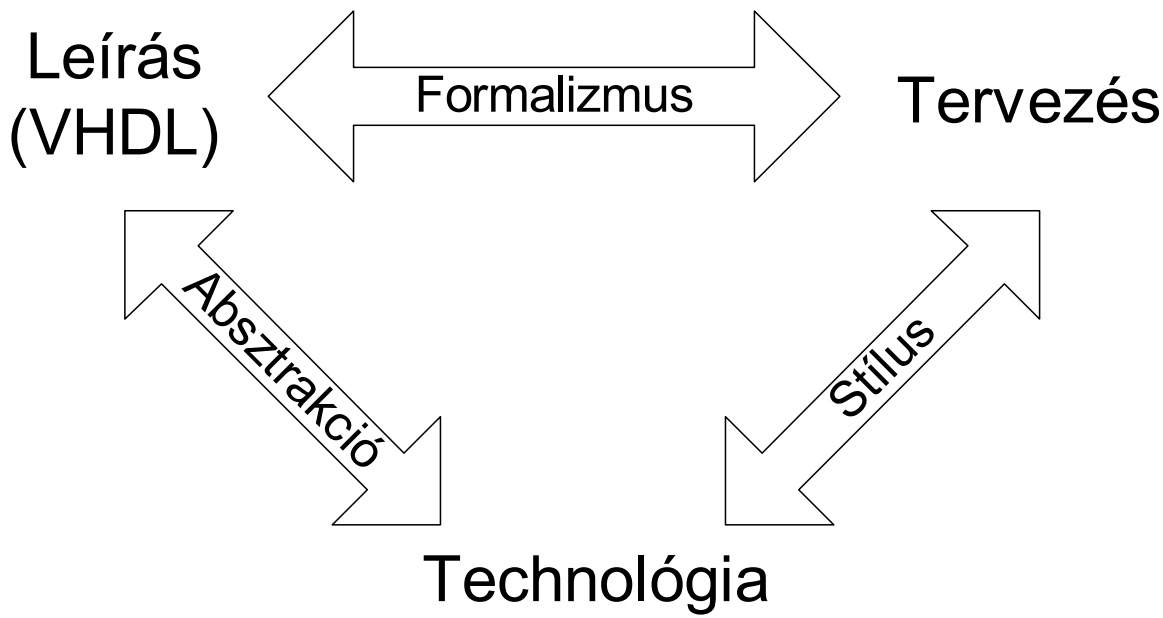
Előadó: Vörösházi Zsolt
voroshazi@vision.vein.hu

Jegyzetek, segédanyagok:

- Könyvfejezetek:
  - http://www.knt.vein.hu
    -> Oktatás -> Tantárgyak -> Digitális Rendszerek (BSC).
    (00_chapter, 01_chapter.pdf)
- Fóliák, óravázlatok .ppt (.pdf)
- Feltöltésük folyamatosan

I. Logikai egyenletek leírása

- Stílus – Absztrakció – Formalizmus



Stílus

- Komplex feladat $\Rightarrow$ egyszerűbb, kezelhető részfeladatokra bontása
  - Szisztematikus
  - Érthető módszerek kellenek

  Pl: programozási stílusok (Top-down, strukturált)

- Jó stílus kialakításának szabályai:
  - Top-down módszer szerinti tervezés
  - Csak kiforrott, biztos technikákat szabad alkalmazni
  - Fontos a dokumentálás!

Absztrakció

- Digitális tervezés „elvi-fogalmi” szintje
  - Kezdeti absztrakció a tervezés során meghatározó, kritikus pont!
    - 1. koncepcionális modell (elvi elgondolás)
    - 2. megvalósítható, realizálható modell (HW)
  - Magas-szintű absztrakció $\Rightarrow$ elvi modell szintenkénti finomítása és felépítése

Formalizmus

- A rendszer viselkedésének leírására szolgál
  - Szisztematikus szabályok és eljárások
  - Minden absztrakciós szinten fontos a használatuk
    - Pl: alapvető formalizmus a Boole-algebra (bináris logika elmélete) – de csak alsóbb szinteken használható

(felsőbb-, rendszer-szint)		(alsóbb-, áramköri szint)
Absztrakció	$\Leftrightarrow$	Boole algebra (konkretizálás)

Digitális tervezés

- Logikai konstansok:
  - Logikai állítás: Igaz / Hamis, True / False, 1 / 0
- Logikai (bináris) változók:
  - Pl: ‘A’ logikai változó esetén legyen,
    A:=fotódióda hiba
    ‘A’ lehet: T / F (A=F nincs hiba; A=T hiba)
  - Logikai változó neve utaljon a funkciójára
- Logikai operátorok:
  - Felírásuk igazság táblázattal (Truth Table)

Igazságtábla: logikai operátorok felírása

- ‘X’ logikai függvény megadása az ‘A,C,B’ logikai változók összes lehetséges értékétől függően
  Jel: X(A,C,B) //3 változó -> 2^3 = 8 sor//

A	C	B	X
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

A	C	B	X	sor
F	F	F	F	0.
F	F	T	T	1.
F	T	F	T	2.
F	T	T	F	3.
T	F	F	T	4.
T	F	T	F	5.
T	T	F	F	6.
T	T	T	F	7.

- Kanonikus „standard” igazság tábla:
  000 – 111 -ig (3 változó esetén)

Logikai operátorok és igazság táblázatuk (NOT)

- Jel: $\mathrm{NOT\,A} = \overline{A}$
- Formális definíció igazságtáblával:

A	NOT A
0	1
1	0

- Def:
  - ha A hamis, NOT A igaz
  - ha A igaz, NOT hamis

Logikai operátorok és igazság táblázatuk (AND)

- Jel: B **AND** C = B·C
- Formális definíció igazságtáblával:

B	C	B·C
0	0	0
0	1	0
1	0	0
1	1	1

- Def: B·C értéke pontosan akkor ‘igaz’ ha ‘B’ és ‘C’ is egyszerre ‘igaz’, különben hamis

Logikai operátorok és igazság táblázatuk (OR)

- Jel: B **OR** C = B**+**C
- Formális definíció igazságtáblával:

B	C	B+C
0	0	0
0	1	1
1	0	1
1	1	1

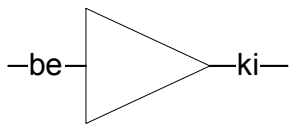
- Def: B**+**C értéke pontosan akkor ‘igaz’ ,ha ‘B’ és ‘C’ közül legalább az egyik ‘igaz’, különben hamis

Egy ill. kétváltozós logikai függvények bemutatása és szabványos jelöléseik

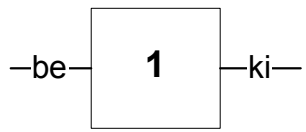
Egyváltozós logikai függvények:

- Jelmásoló (jel-erősítés):

be	ki
0	0
1	1



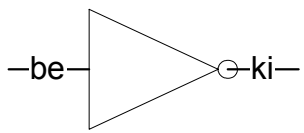
Nemzetközi szabvány

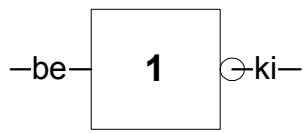


Magyar szabvány

- Negálás - Inverter (NOT):

be	ki
0	1
1	0

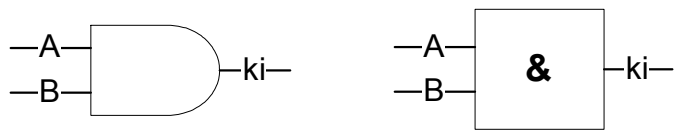




Kétváltozós logikai függvények:

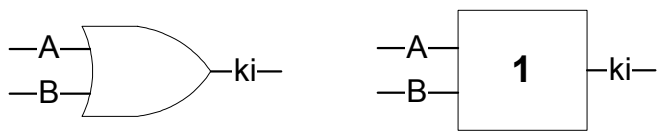
- ÉS (AND):

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



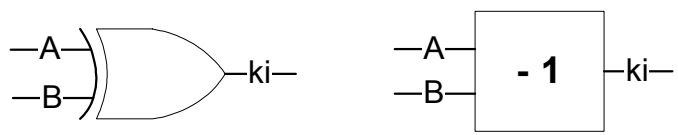
- VAGY (OR):

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



- Antivalencia (XOR):

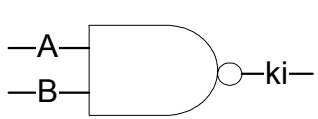
A	B	ki
0	0	0
0	1	1
1	0	1
1	1	0

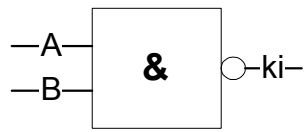


Kétváltozós log.függv. (folyt.):

- NEM-ÉS (NAND):

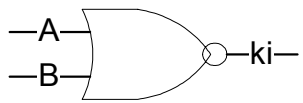
A	B	ki
0	0	1
0	1	1
1	0	1
1	1	0

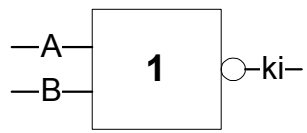




- NEM-VAGY (NOR):

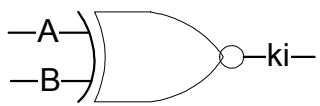
A	B	ki
0	0	1
0	1	0
1	0	0
1	1	0

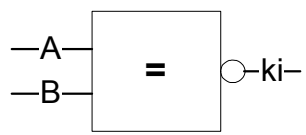




- Ekvivalencia (NXOR):

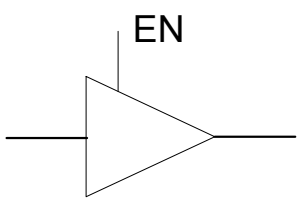
A	B	ki
0	0	1
0	1	0
1	0	0
1	1	1



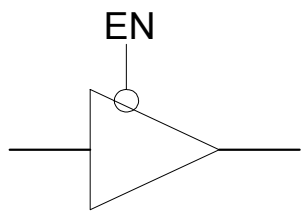


Tri-State Buffer:

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



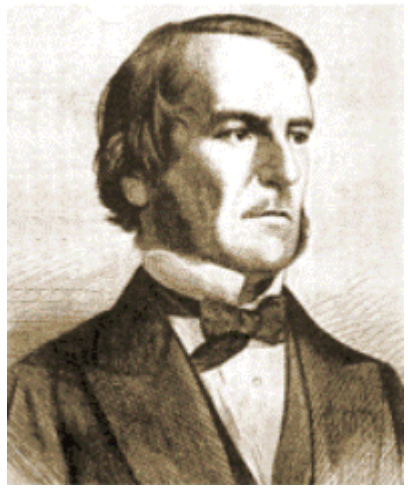
High-true enable



Low-true enable

Boole algebra

Boole-algebra



(1815-1864)

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

Boole algebra elemei:

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

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

Boole-algebrai azonosság igazolása igazságtáblával

- Pl: De Morgan

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

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

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

Dualitás elve

- Példa: egyszerűsítésre

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

Logikai egyenletek megadása igazság-táblázatokból

- PI:

sor	A	B	W
0	0	0	0
1	0	1	0
2	1	0	1
3	1	1	0

W pontosan akkor lesz **igaz**, ha A igaz és B hamis, egyébként W **hamis** lesz. Vagyis egyenletként kifejezve:

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

- PI:

sor	A	B	Y
0	0	0	1
1	0	1	0
2	1	0	1
3	1	1	1

Y pontosan akkor lesz **igaz**, ha A és B is hamis, vagy A igaz és B hamis, vagy A és B is igaz, egyébként W **hamis** lesz. Vagyis egyenletként kifejezve:

$$Y = \overline{A} \cdot \overline{B} + A \cdot \overline{B} + A \cdot B \Rightarrow \overline{B} + A \cdot B \Rightarrow \overline{B} + A$$

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

1.) Sum-of-Products (szorzat„termek” összege)

- Szorzat (AND) termek összeg (OR) kapcsolata
- Emberi szemléletmódhoz közelebb áll: a táblázat soraiból azokat a függvényértékeket (Y) vesszük amelyek ‘1’-esek
- Def: Triviális forma: ha egy változó egy adott szorzat termben vagy ponáltan, vagy negáltan legfeljebb egyszer szerepel.
  Ezt hívják még **mintermnek (m_i)** vagy **kanonikus szorzat termnek** is.
  - Pl: *valós / triviális* / kanonikus formulák: $A, \overline{A}, A \cdot B, \overline{A} \cdot B \cdot \overline{C}$
  - Pl: érvénytelen formulák (de ettől még Boole kifejezés), ami jelenti azt is, hogy tovább egyszerűsíthetők:

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

Diszjunktív Normál Forma:

- Jel: $Y(DNF): \sum_{i=0}^{2^n-1} m_i$
- n változó esetén 2^n lehetséges minterm van.
- Képzésük: az igazságtáblázatból azoknak a mintermeknek a VAGY kapcsolatát vesszük, ahol függvényértékek sorában (*Y*) ‘1’ -es szerepel, vagy ahol a függvény komplemensének ($\overline{Y}$) értéke ‘0’.
- minterm: m_i (i. sora a kanonikus táblának, ahol Y értéke ‘1’).

Példa: DNF felírása

- Igazságtábla:

sor	A	B	Y
0	0	0	1
1	0	1	0
2	1	0	1
3	1	1	1

- Kapott egyenlet: $Y = \overline{A} \cdot \overline{B} + A \cdot \overline{B} + A \cdot B = m_0 + m_2 + m_3$

  [0 0] [1 0] [1 1]

- Komplemens: $\overline{Y} = \overline{A} \cdot B = m_1$

  [0 1]

2.) Product-of-Sums: összeg„termek” szorzata

- összeg (OR) termek szorzat (AND) kapcsolata
- **Maxterm** (**M_i**): olyan kanonikus összeg term, amelyben mindegyik logikai változó pontosan egyszer fordul elő, ponált, vagy negált alakban.
  - Valós maxterm: $\overline{A}+B+\overline{C}$, de nem valós: $\overline{A}+\overline{C}$
  - Kanonikus forma: $W=(P+Q+R)(P+\overline{Q}+\overline{R})$
  - Nem kanonikus forma: $W=(P+Q)(P+\overline{Q}+\overline{R})$
- Gyakorlatban kevésbé használt forma.

KNF: Konjunktív Normál Forma

- Jel: $W(KNF) = \prod_{i=0}^{2^n-1} M_i$
- Képzésük: a kanonikus igazságtábla azon maxtermjeinek ÉS kapcsolatát vesszük, ahol a függvény (W) értéke ‘0’, vagy a komplemens függvény ($\overline{W}$) értéke ‘1’.
- Pl: $W = \overline{A} + \overline{B}$ vagy

$$\overline{W} = (A+B)\cdot(\overline{A}+B)\cdot(A+\overline{B}) \overset{\text{disztributív}}{=\!=\!=} A\cdot B = A\cdot B$$

- Maxterm (M_i): az igazságtáblázat i. sora, ahol a kimeneti függvényérték ‘0’.

Példák: KNF

- Legyen: $M_i = \overline{A} + B + \overline{C}$ ahol a kimeneti függvényérték hamis volt. Ez az M_i maxterm igaz A,B,C változók értékének kombinációjára, kivéve egyet, ahol A=1, B=0, C=1. Tehát [101]=5. $\rightarrow$ M_5 (táblázat 5.sora)

- Legyen: $M_i = A + B + C$ ahol a kimeneti függvényérték hamis volt. Ez az M_i maxterm igaz A,B,C változók értékének kombinációjára, kivéve egyet, ahol A=0, B=0, C=0. Tehát [000]=0. $\rightarrow$ Így M_0 (táblázat 0.sora)

Példa: KNF felírása

- Igazságtábla

sor	J	K	L	W
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	1
4	1	0	0	0
5	1	0	1	0
6	1	1	0	0
7	1	1	1	0

- Igazságtáblából kapjuk, hogy:

$$W(KNF) = (J + K + L) \cdot (\overline{J} + K + L) \cdot (\overline{J} + K + \overline{L}) \cdot (\overline{J} + \overline{K} + L) \cdot (\overline{J} + \overline{K} + \overline{L})$$

$$W(KNF) = [000] \cdot [100] \cdot [101] \cdot [110] \cdot [111] = M_0 \cdot M_4 \cdot M_5 \cdot M_6 \cdot M_7$$

$$W(DNF) = (\overline{J} \cdot \overline{K} \cdot L) + (\overline{J} \cdot K \cdot \overline{L}) + (\overline{J} \cdot K \cdot L)$$

$$W(DNF) = [001] + [010] + [011] = m_1 + m_2 + m_3$$

Igazságtábla felírása logikai kifejezésekből I.

- a.) DNF-ből: felírás egyszerű
  - Kanonikus mintermből: egy sor képződik (ahol Y igaz),
  - Nem kanonikus, kevesebb változót tartalmazó termből: több sor is képződhet, mivel egy ilyen term egy adott logikai változó ponált és negált értékére is igaz kimeneti eredményt (Y) ad,
  - Egy sorhoz több term is tartozhat!

Példa: DNF -> Igazságtábla

- Eredeti egyenlet:

$$Y(DNF) = J \cdot \overline{K} + \overline{J} \cdot K \cdot L + J \cdot K \cdot \overline{L} + K \cdot L$$

term1 term2 term3 term4

kanonikus (minterm)

- Kapott igazságtábla:

sor	J	K	L	W	
0	0	0	0	0	
1	0	0	1	0	
2	0	1	0	0	
3	0	1	1	1	term2 és term4
4	1	0	0	1	term1
5	1	0	1	1	term1
6	1	1	0	1	term3
7	1	1	1	1	term4

Igazságtábla felírása logikai kifejezésekből II.

- b.) KNF-ből: felírás nehezebb (az egyes logikai változók negált értékeit kell venni)
  - Kanonikus maxtermből: egy sor képződik (ahol Y hamis),
  - Nem kanonikus, kevesebb változót tartalmazó termből: több sor is képződhet, mivel egy ilyen term egy adott logikai változó ponált és negált értékére is hamis kimeneti eredményt (Y) ad,
  - Egy sorhoz több term is tartozhat!

Példa: KNF -> Igazságtábla

- Eredeti egyenlet:

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

term1 term2 term3

kanonikus (maxterm)

- Kapott igazságtábla:

sor	A	B	C	G	
0	0	0	0	1	
1	0	0	1	1	
2	0	1	0	1	
3	0	1	1	1	
4	1	0	0	0	term1 és term2
5	1	0	1	0	term2
6	1	1	0	1	
7	1	1	1	0	term3

Igazság táblák tömörebb felírási formája

- Eml: Kanonikus ig. táblánál: n változó -> 2^n sor (összes lehetséges változó kombináció felírásával)
- Egyszerűsített / tömörebb felírás:
  - „**X**”: Don’t Care változó két értéke: 0 és 1 is lehet.
- Pl: $Y = J \cdot \overline{K} + \overline{J} \cdot K \cdot L + J \cdot K \cdot \overline{L} + K \cdot L$

term1 term2 term3 term4

kanonikus (minterm)

J	K	L	Y	
0	0	0	0	
0	0	1	0	
0	1	0	0	
X	1	1	1	**term2 és term4**
1	0	X	1	**term1**
1	1	0	1	**term3**

Term1: L don’t care (0 v. 1)

Term4: J dont’care (1 v. 0)

NTSH: Nem Teljesen Specifikált Hálózat (Don’t Care kimenet)

- Bizonyos bemeneti kombinációkra ugyanazt a kimeneti eredményt kapjuk (irreleváns)
- Jele: „–” Don’t care kimeneti állapot
- Pl.

A	B	Y
0	X	1
1	0	-
1	1	0

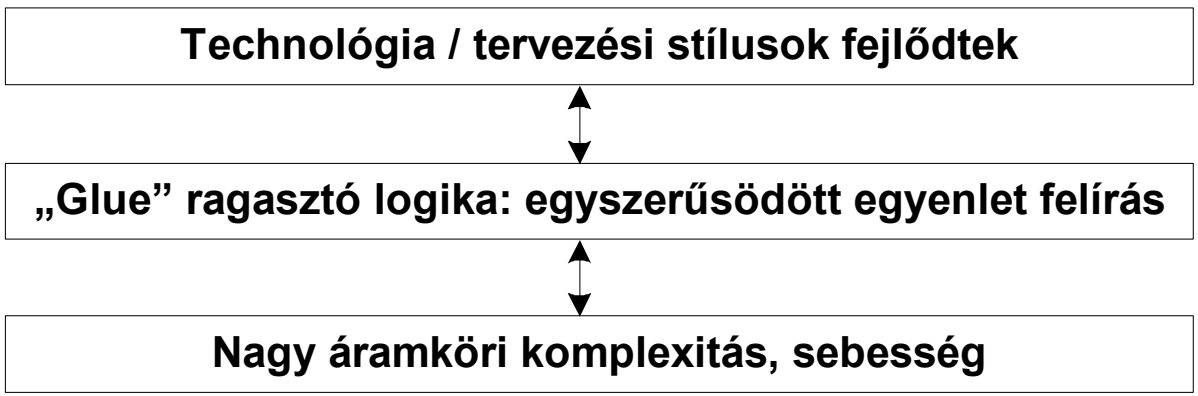
ha ‘–’=1, $Y = \overline{A} + A \cdot \overline{B} = \overline{A} + \overline{B}$

ha ‘–’=0, $Y = \overline{A}$

KARNOUGH TÁBLÁK

Karnough táblák

- Korai időszakban: logikai elemek hatalmas, nehezen tervezhető, nagy energiát disszipáló eszközökből álltak
- Logikai kifejezések egyszerűsítése. Ma: HW olcsó elemekből épül fel. Cél: az áramköri minimalizáció (modularitás, egyszerűség)

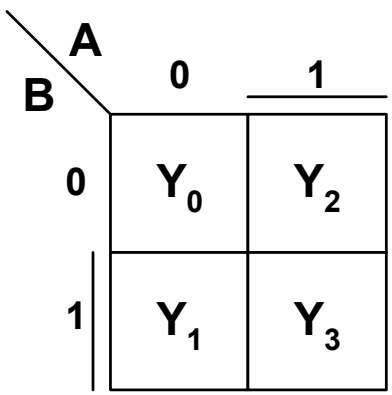


- **K-Map / Veicht diagram**: grafikus ábrázolási és egyszerűsítési mód, a kanonikus igazságtábla egy újrarendezett formája (több forma is létezik, és fontos a betűk, cimkék sorrendje)

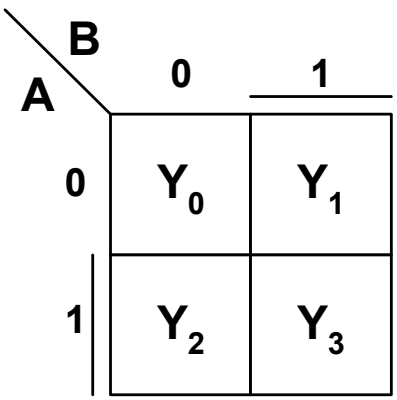
Karnough tábla felírása igazság táblázatból

- Igazságtábla mindenegyes sorának kimeneti értékéhez (Y_i) a Karnough tábla egy négyzete feleltethető meg.
- Pl. n=2 változó esetén lehetséges táblák:

sor	A	B	Y
0	0	0	Y0
1	0	1	Y1
2	1	0	Y2
3	1	1	Y3



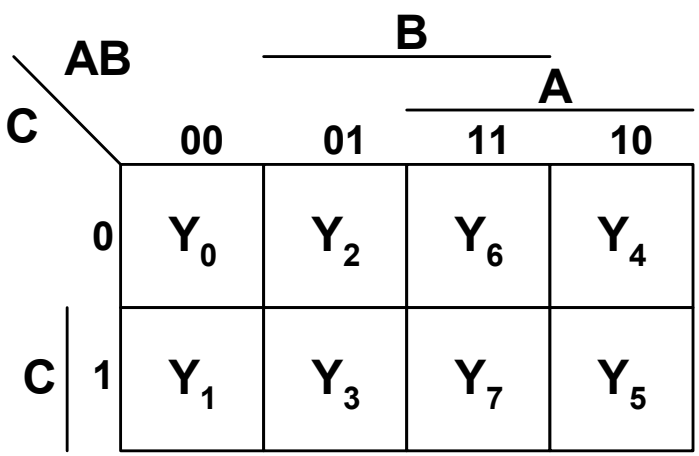
könyvbeli jelölés

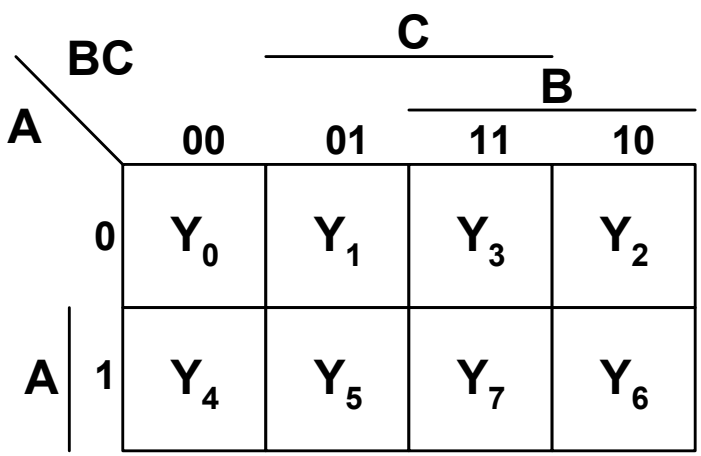


általános jelölés

Karnough táblák

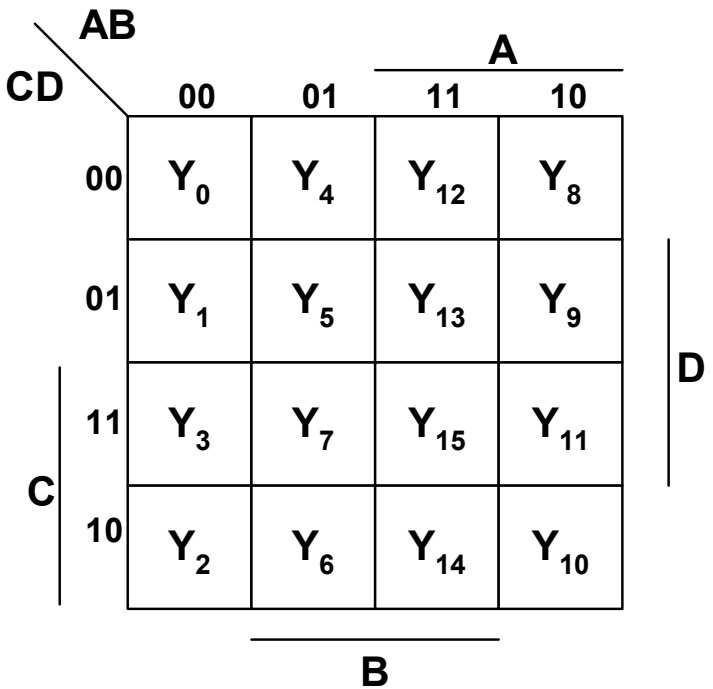
- n=2, 3, 4 változóval még könnyű felírni (>4 változó felett már más technikát használunk)
- Pl: n=3 változó esetén lehetséges táblákra:

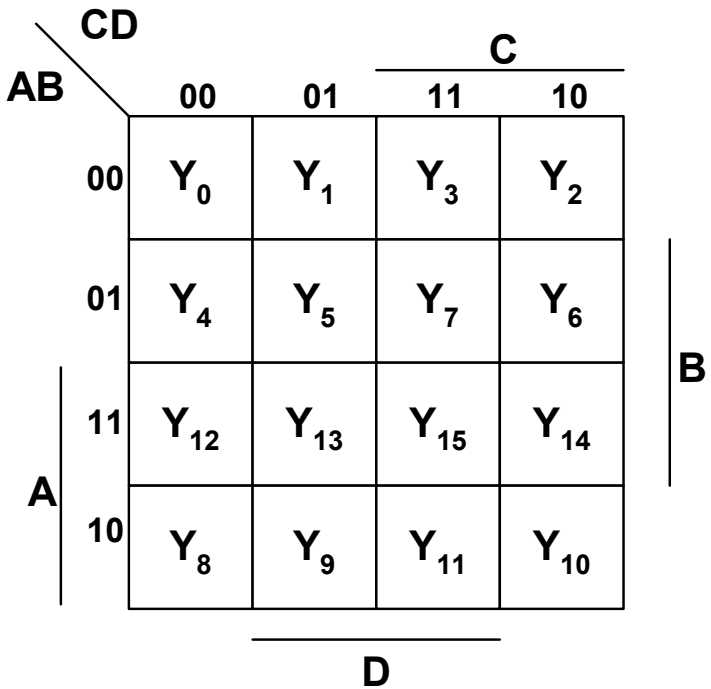




Karnough táblák

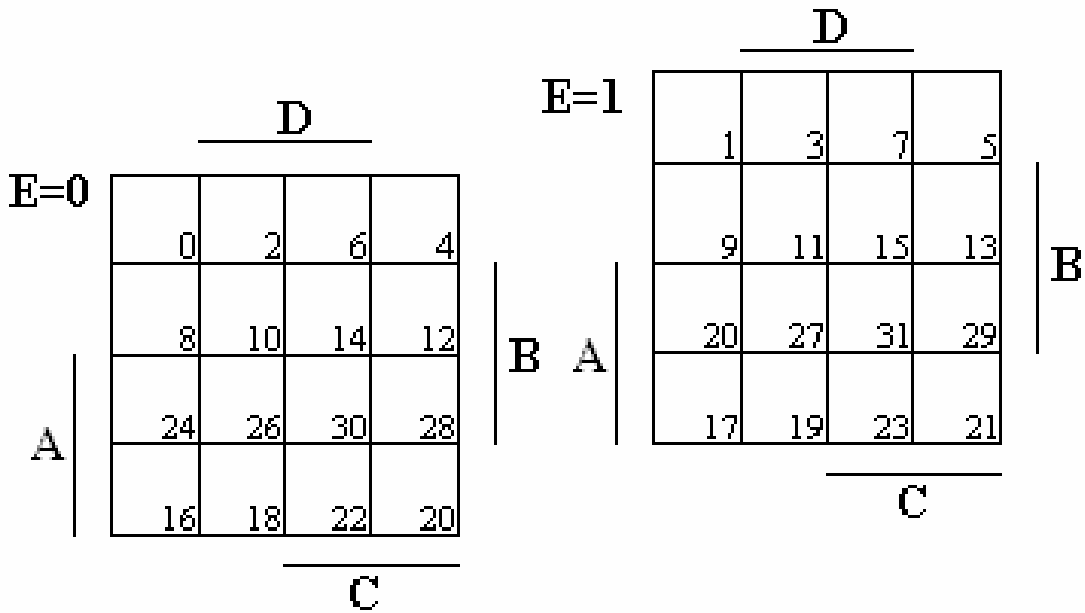
- Pl: n=4 változó esetén lehetséges táblákra:



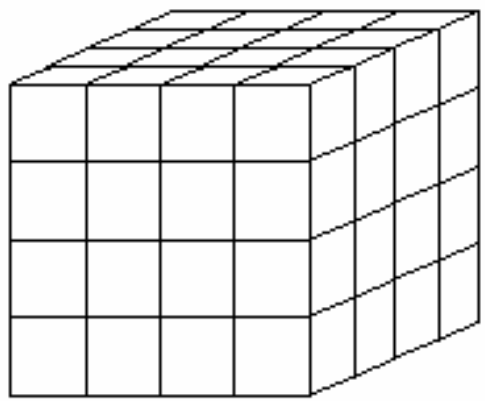


Karnough táblák

- n= 5 változó esetén



- n=6 változó esetén



Boole függvény ábrázolási módjai

- Boole-algebrai kifejezés: $Y = \overline{A} \cdot \overline{B} + A \cdot \overline{B}$
- Igazságtábla:

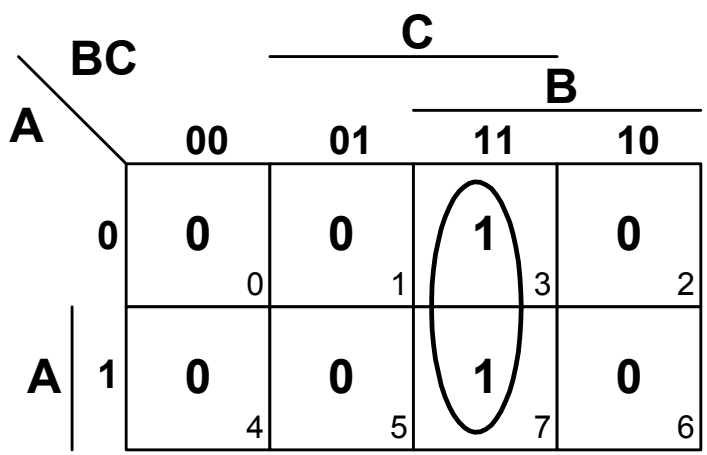
sor	A	B	Y
0	0	0	1
1	0	1	0
2	1	0	1
3	1	1	0

- Karnough tábla:

A \ B	0	1
0	1	0
1	1	0

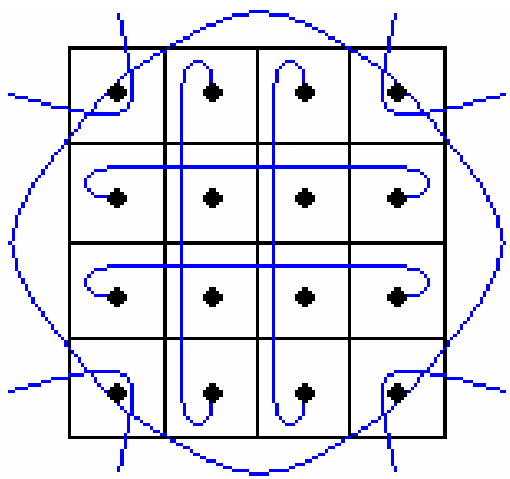
Szomszédosság – adjacencia

- Def: Ha egy Karnough táblában két szomszédos (adjacent) cella csak egy változó értékében különbözik (egységnyi távolság)!
- Pl. $Y_3 = \overline{A} \cdot B \cdot C$ és $Y_7 = A \cdot B \cdot C$



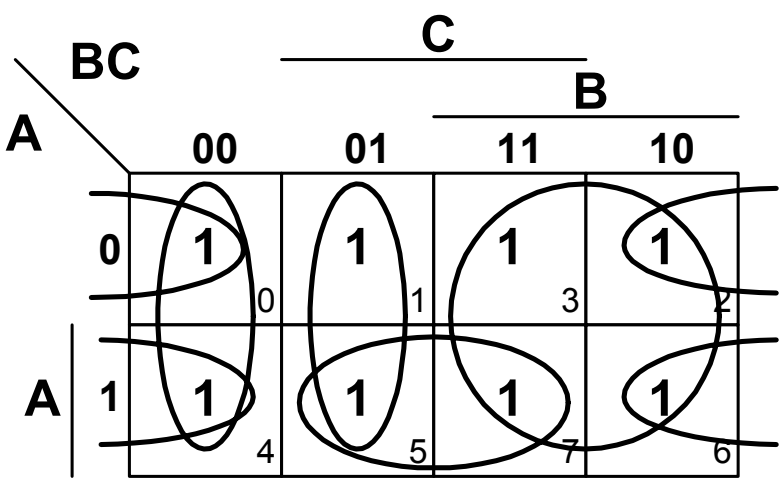
Egyszerűsítés Karnough táblákkal

- Tömörítés szabályai:
  - 2^n (n=0,1,2..) term vonható be egy tömbbe,
  - Egyetlen term több tömbben is szerepelhet (átlapolódás lehetséges)
  - Egyik tömb, a másikat nem tartalmazhatja teljes mértékben, (redundancia)
  - Mindig a lehető legnagyobb lefedéseket keressük, és haladjunk a legkisebb méretű tömbök/lefedések felé
  - Don't care ('-') kimeneti függvényértékeket a jobb lefedésnek megfelelően kell megválasztani (NTSH)
  - Egymás mellett lévő (adjacens) sorokra és oszlopokra érvényes:



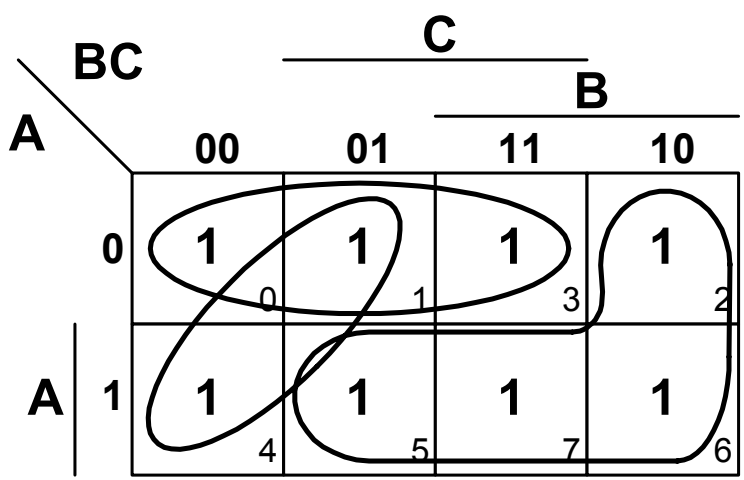
Példa: Karnough táblák egyszerűsítése

- érvényes



Nem összes, de lehetséges egyszerűsítések - érvényes

érvénytelen



Átlós, és nem 2^n számú ‘1’-es lefedés érvénytelen

Lehetséges módszerek Karnough tábla értelmezésére:

- M1: $Y(DNF)$ ‘1’-esek lefedésével képzett (normál, eddig használt ált. módszer)
- M2: $\overline{Y}(DNF)$ ‘0’-k lefedésével képzett inverz függvény felírás
- M3: $Y(KNF)$ ‘0’-k lefedésével képzett
- M4: $\overline{Y}(KNF)$ ‘1’-esek lefedésével képzett inverz függvény felírás

- Ajánlott: fejezetek végén a feladatok (Exercises) részek áttekintése.

Pannon Egyetem
Képfeldolgozás és Neuroszámítógépek Tanszék

Digitális Rendszerek (BSc)

2. előadás: Logikai egyenletek leírása II:
Függvény-egyszerűsítési eljárások

Előadó: Vörösházi Zsolt
voroshazi@vision.vein.hu

Jegyzetek, segédanyagok:

- Könyvfejezetek:
  - http://www.knt.vein.hu
    -> Oktatás -> Tantárgyak -> Digitális Rendszerek (BSC).
    (01_chapter.pdf)
- Fóliák, óravázlatok .ppt (.pdf)
- Feltöltésük folyamatosan

Függvényminimalizálás

- Általánosan:
  - Függvényminimalizálást a szomszédos mintermek megkeresésével tehetjük meg.
  - A szomszédosság megállapítása után egyszerűsítünk.
  - Minterm → implikáns (egyszerűsíthető) → prímimplikáns (tovább nem egyszerűsíthető)

Függvényegyszerűsítési eljárások

- 1.) Algebrai módszer (Boole algebrai azonosságokkal)
- 2.) Kifejtési módszer
- 3.) Grafikus módszer: (Karnough tábla, igazság tábla)
- 4.) Normálformák:
  - DNF: Diszjunktív Normál Forma
  - KNF: Konjunktív Normál Forma
- 5.) Számjegyes minimalizálás: **Quine-McCluskey**

1.) Algebrai módszer

- A Boole-algebra azonosságait használjuk fel az egyszerűsítéshez:

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

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

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

2.) Kifejtési módszer:

- Komplexebb függvények esetén egy adott változó értékét először ponáltnak, majd negáltnak definiáljuk, végül pedig az így kiszámított két logikai kifejezést összeadjuk. Ezáltal leegyszerűsödik a függvényminimalizálási feladat.

Példa: kifejtési módszer

- Legyen F_1 függvény a következő:

$$F_1(A,B,C) = \overline{A}\cdot B\cdot\overline{C} + \overline{A}\cdot B\cdot C + A\cdot\overline{B}\cdot\overline{C} + A\cdot B\cdot\overline{C}$$

  - Ha A:=1

$$F_1(1,B,C) = \cancel{0\cdot B\cdot\overline{C}} + \cancel{0\cdot B\cdot C} + 1\cdot\overline{B}\cdot\overline{C} + 1\cdot B\cdot\overline{C}$$

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

  - Ha A:=0

$$F_1(0,B,C) = 1\cdot B\cdot\overline{C} + 1\cdot B\cdot C + \cancel{0\cdot\overline{B}\cdot\overline{C}} + \cancel{0\cdot B\cdot\overline{C}}$$

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

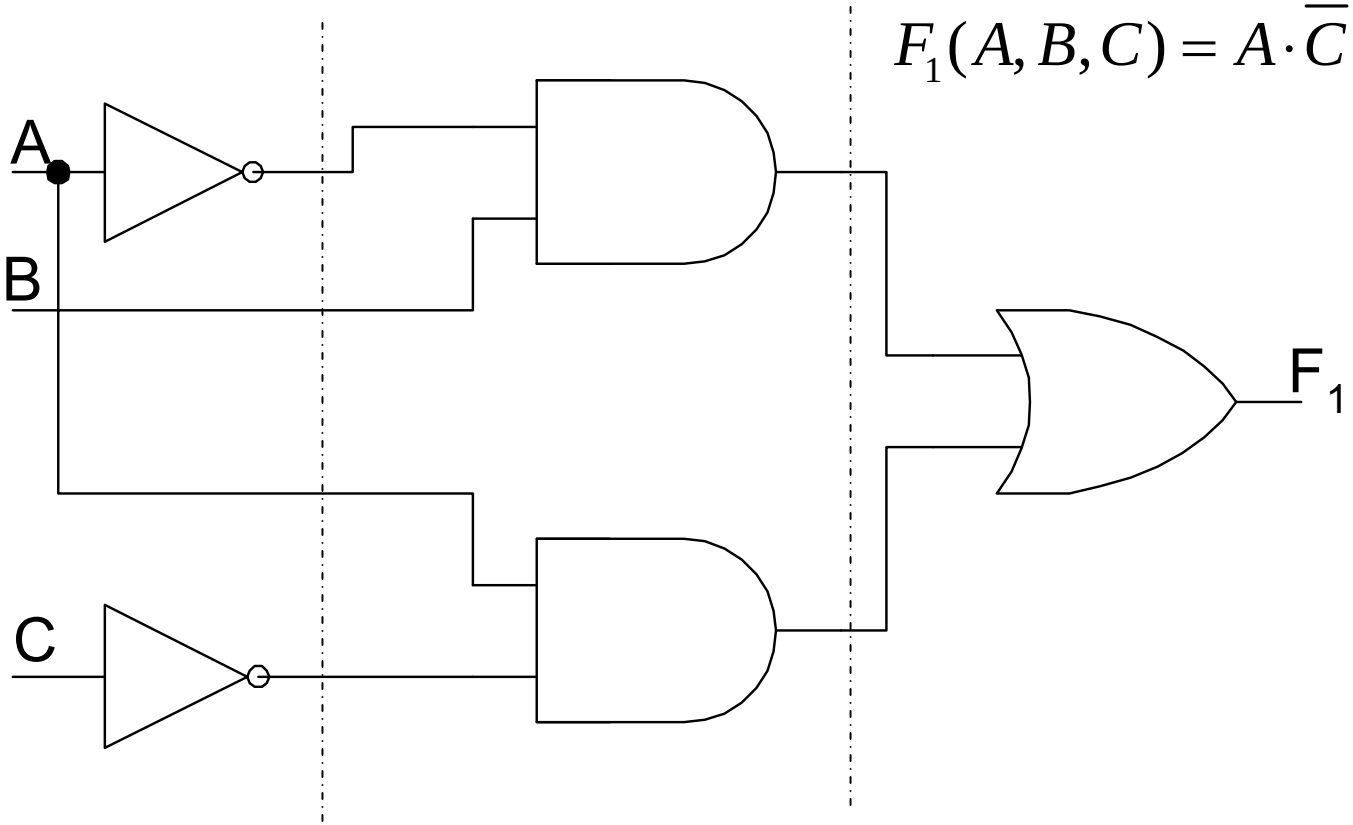
  - Végül összeadjuk a kettőt (egyszerűsített alak):

$$F_1(A,B,C) = A\cdot F_1(1,B,C) + \overline{A}\cdot F_1(0,B,C) =$$

$$= A\cdot\overline{C} + \overline{A}\cdot B$$

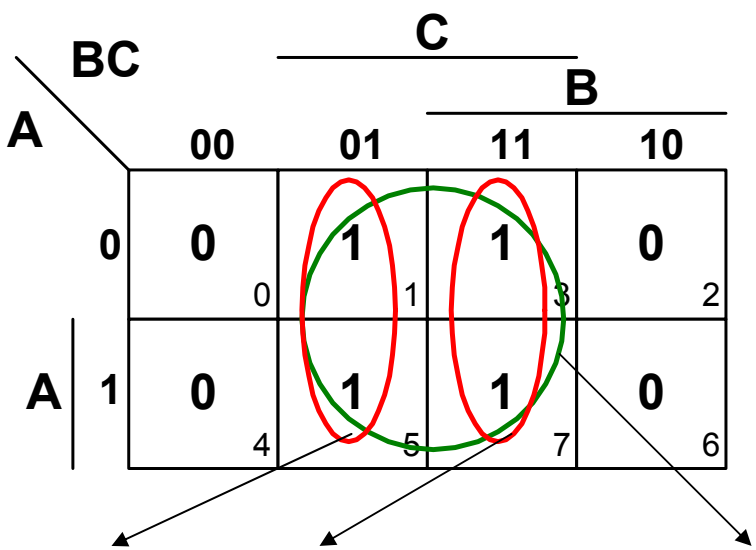
Az egyszerűsített függvény logikai áramköri realizációja

$$F_1(A,B,C) = A \cdot \overline{C} + \overline{A} \cdot B$$



3.) Grafikus módszer

- Karnough (Veicht) diagramm
  - Tömbösítés szabályainak betartása!
- Példa:



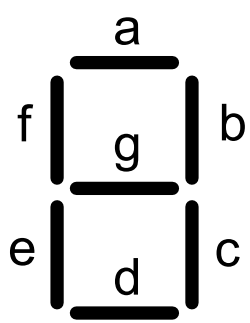
$$\overline{B} \cdot C + B \cdot C = C \cdot (\overline{B} + B) \Leftrightarrow C$$

Lehetséges, de nem tömör összevonások

Legtömörebb összevonás

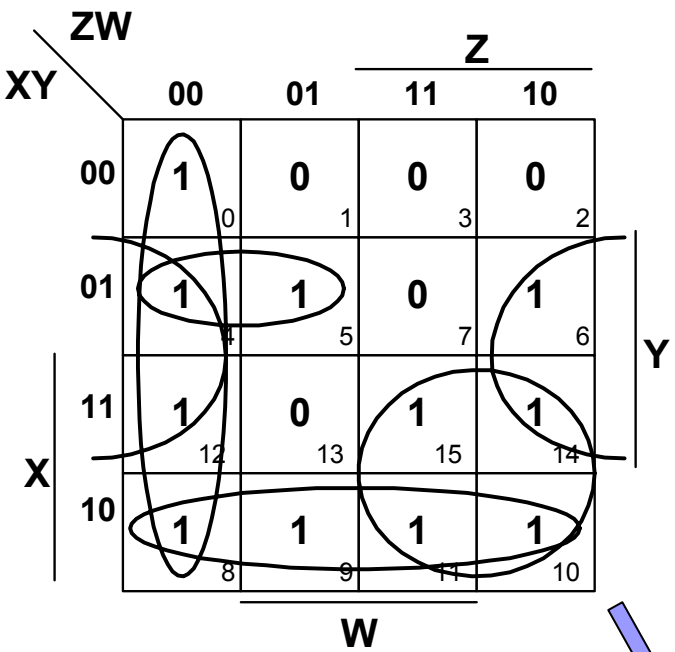
Példa 1: 7-szegmenses dekóder áramkör tervezése

- ***Számjegyek*** (0-9) és spec. ***hexadecimális karakterek*** megjelenítésére (A b C d E F)
- nemzetközi elnevezései a szegmenseknek: (a, b, c, d, e, f, g)
  - 16 érték (4 biten ábrázolható): F(X,Y,Z,W)



Példa: 7-szegmenses dekóder tervezése (folyt)

- Igazságtábla (**f** szegmensre)
- Karnough tábla:

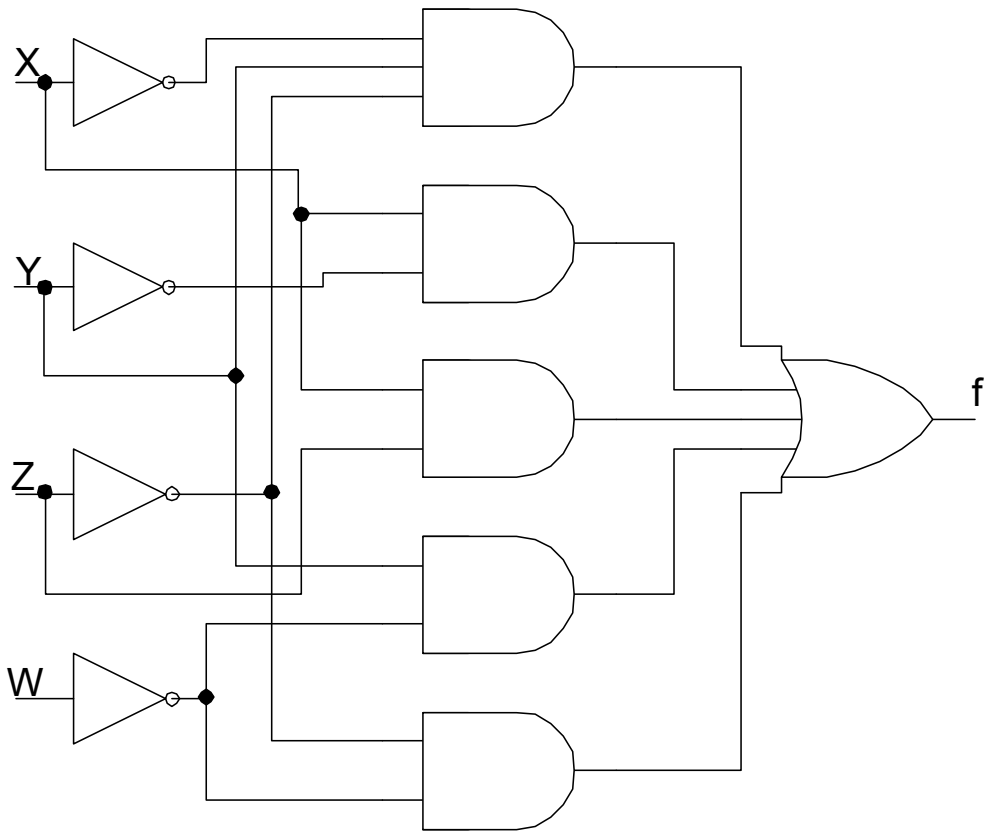


sor	X	Y	Z	W	f
0	0	0	0	0	1
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	1
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	0
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	1
11	1	0	1	1	1
12	1	1	0	0	1
13	1	1	0	1	0
14	1	1	1	0	1
15	1	1	1	1	1

- Kapott **f** kimeneti függvény:

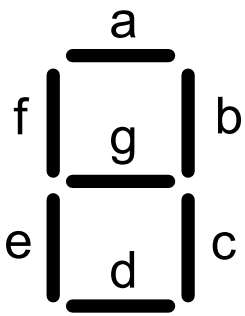
$$f(X,Y,Z,W) = \overline{Z}\cdot\overline{W} + X\cdot\overline{Y} + Y\cdot\overline{W} + X\cdot Z + \overline{X}\cdot Y\cdot\overline{Z}$$

Példa 1: A 7-szegmenses dekóder logikai áramköri realizációja



$$f(X,Y,Z,W) = \overline{Z} \cdot \overline{W} + X \cdot \overline{Y} + Y \cdot \overline{W} + X \cdot Z + \overline{X} \cdot Y \cdot \overline{Z}$$

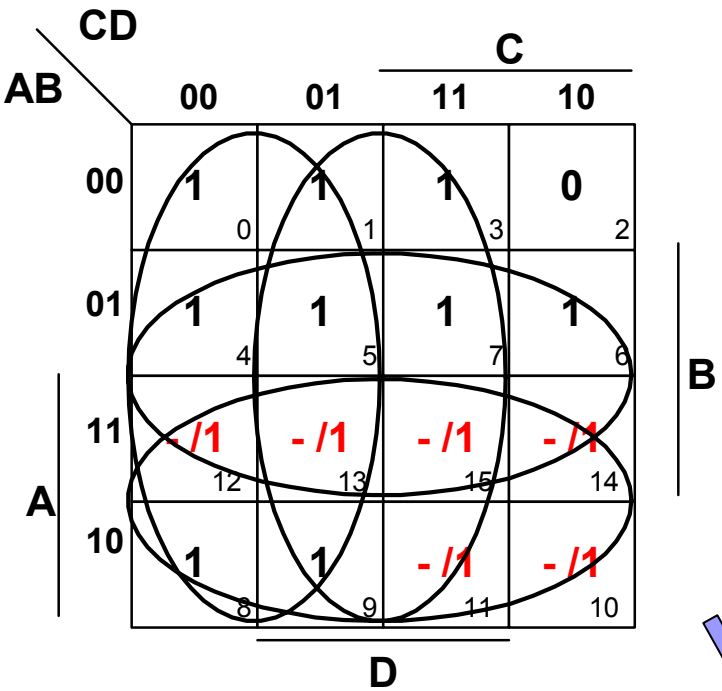
Példa 2: 7-szegmenses dekóder áramkör tervezése



- Csak számjegyeket (0-9) megjelenítésére
  - BCD: Binárisan kódolt decimális számokra
- Nemzetközi elnevezései a szegmenseknek: (a, b, c, d, e, f, g)
  - 10 érték (4 biten ábrázolható): F(A,B,C,D)
- **NTSH**: használjunk Nem Teljesen Specifikált Hálózatot (igazságtábla kimeneti függvényértékeiben lehetnek don't care '-' definiált állapotok)
  - Feladat: $F = \sum_{i=0}^{2^n-1}(0,1,3,4,5,6,7,8,9)_{n=4}$ $\quad x:10,11,12,13,14,15$

Példa 2: 7-szegmenses dekóder tervezése (folyt)

- Igazságtábla (**c** szegmensre)
- Karnough tábla:

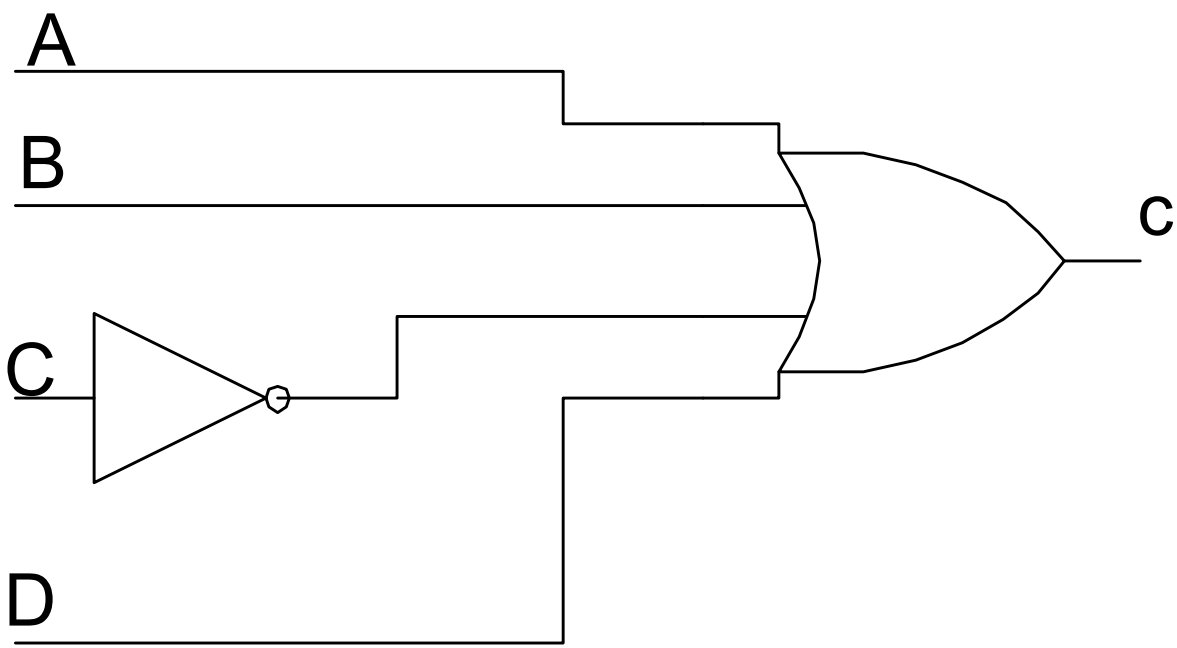


sor	A	B	C	D	c
0	0	0	0	0	**1**
1	0	0	0	1	**1**
2	0	0	1	0	**0**
3	0	0	1	1	**1**
4	0	1	0	0	**1**
5	0	1	0	1	**1**
6	0	1	1	0	**1**
7	0	1	1	1	**1**
8	1	0	0	0	**1**
9	1	0	0	1	**1**
10	1	0	1	0	**–**
11	1	0	1	1	**–**
12	1	1	0	0	**–**
13	1	1	0	1	**–**
14	1	1	1	0	**–**
15	1	1	1	1	**–**

- Kapott **c** kimeneti függvény:

$$c(A,B,C,D) = A + B + \overline{C} + D$$

Példa 2: 7-szegmenses dekóder logikai áramköri realizációja (BCD)



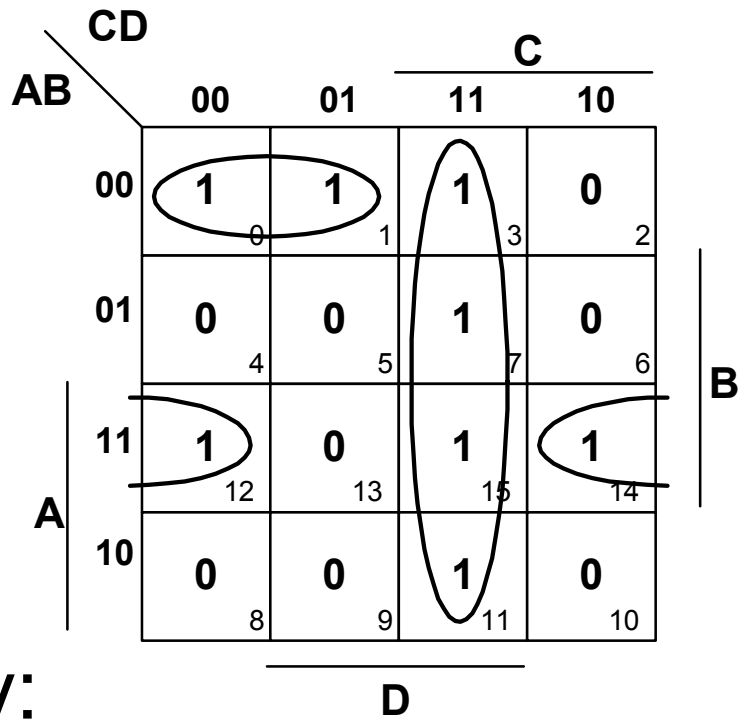
$$c(A,B,C,D) = A + B + \overline{C} + D$$

4.) Normálformák (NF)

- DNF: Diszjunktív Normál Forma
  - mintermek (szorzattermek) VAGY kapcsolata
- KNF: Konjunktív Normál Forma
  - Maxtermek (összegtermek) ÉS kapcsolata

Példa 1: Diszjunktív Normál Forma

- Legyen: $F = \sum_{i=0}^{2^n-1}_{n=4} (0,1,3,7,11,12,14,15)$
- Karnough tábla:

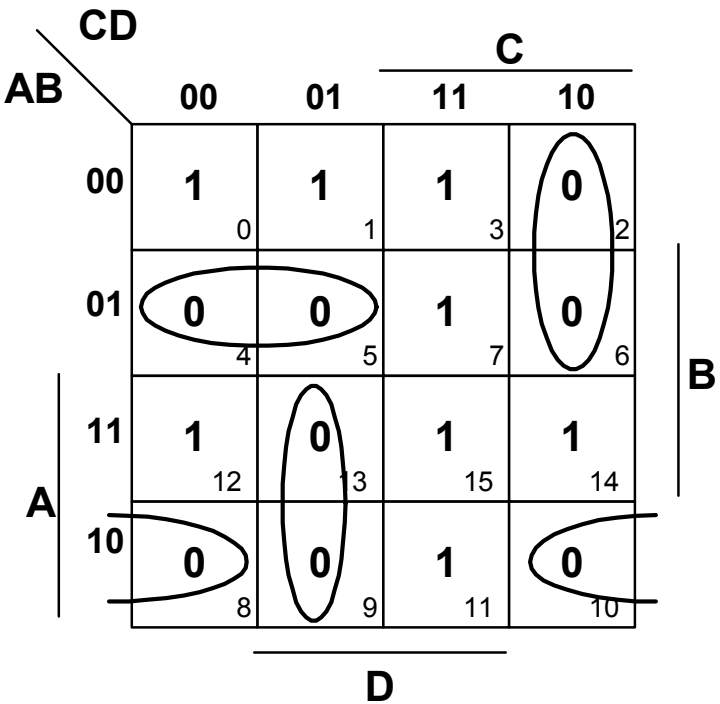


- Kapott F függvény:

$$F(A,B,C,D) = C \cdot D + \overline{A} \cdot \overline{B} \cdot \overline{C} + A \cdot B \cdot \overline{D}$$

Példa 2: Konjunktív Normál Forma

- Legyen: $F = \prod_{i=0}^{2^n-1}{}^{n=4}(2,4,5,6,8,9,10,13)$
- Karnough tábla:



- Kapott F függvény:

$$F(A,B,C,D) = (A + \overline{C} + D) \cdot (A + \overline{B} + C) \cdot (\overline{A} + C + \overline{D}) \cdot (\overline{A} + B + D)$$

5.) Számjegyes minimalizálás (**Quine-McCluskey módszer)**

- Szomszédosság szükséges feltételei:
  - Decimális indexek különbsége 2^n kell legyen (szükséges, de nem elégséges feltétel!)
    - Pl: i: 6-2=4 (szomszédos), de i:10-6=4 (nem szomszédos)
  - Bináris súlyuk különbsége 1. (Hamming távolság)
    - Pl: 0111 (7) vagy 1001 (9)
      0011 (3) 0111 (7)
      0x00 xxx0
      jó rossz

    (szükséges, de nem elégséges feltétel!)
  - A nagyobb decimális indexűnek kell nagyobb bináris súllyal szerepelnie! (szükséges, de nem elégséges feltétel!)

	00	01	11	10
00	Y_0	Y_1	Y_3	Y_2
01	Y_4	Y_5	Y_7	Y_6
11	Y_{12}	Y_{13}	Y_{15}	Y_{14}
10	Y_8	Y_9	Y_{11}	Y_{10}

Példa: Számjegyes minimalizálásra **(Quine-McCluskey módszer)**

- Oldjuk meg a következő feladatot a Quine-McCluskey módszerrel
- Ha adott az F függvény DNF alakban:

$$F = \sum_{i=0}^{2^n-1}{}^{n=4}(0,1,3,7,11,12,14,15)$$

- Karnough tábla:

CD \ AB	00	01	11	10
00	1 (0)	1 (1)	1 (3)	0 (2)
01	0 (4)	0 (5)	1 (7)	0 (6)
11	1 (12)	0 (13)	1 (15)	1 (14)
10	0 (8)	0 (9)	1 (11)	0 (10)

A (00, 01, 11, 10) — B (00, 01, 11, 10) — C (11, 10) — D

Számjegyes minimalizálás
Quine-McCluskey módszer I.lépés

- Csoportosítás bináris súlyuk szerint:
  - ahol a kimeneti értékük ’1-s’ volt.

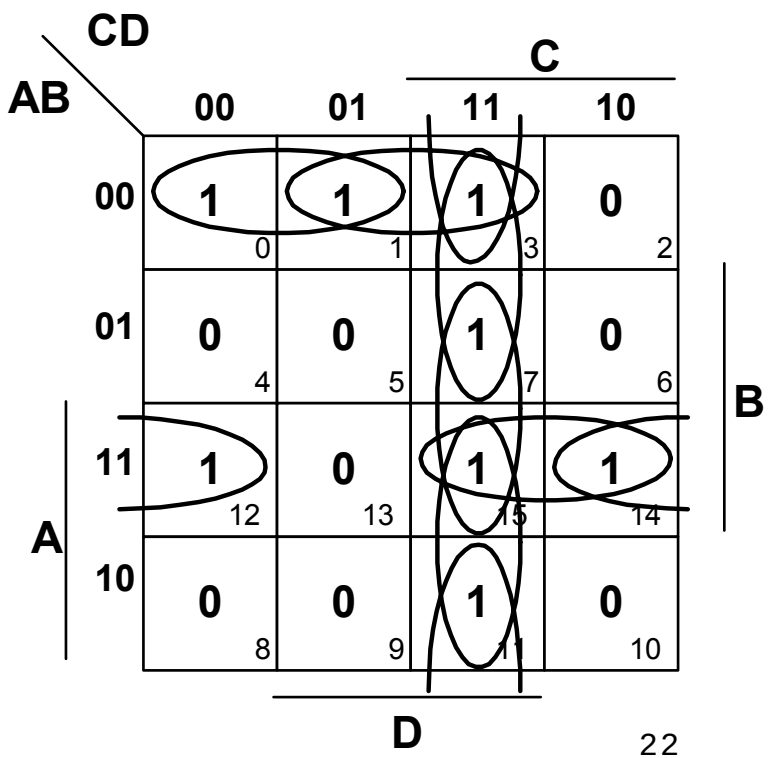
0	0000	[0 bináris súly]
1	0001	[1 bináris súly]
3	0011	[2 bináris súly]
12	1100	
7	0111	[3 bináris súly]
11	1011	
14	1110	
15	1111	[4 bináris súly]

bináris súly szerinti csoportképzések

Számjegyes minimalizálás
Quine-McCluskey módszer II.lépés

- II. Összes létező szomszédos **kételemű** lefedő tömb összevonása (Karnough tábla alapján)

Minterm	Decimális különbség
0,1	(1)
1,3	(2)
3,7	(**4**)
3,11	(8)
12,14	(2)
7,15	(8)
11,15	(**4**)
14,15	(1)



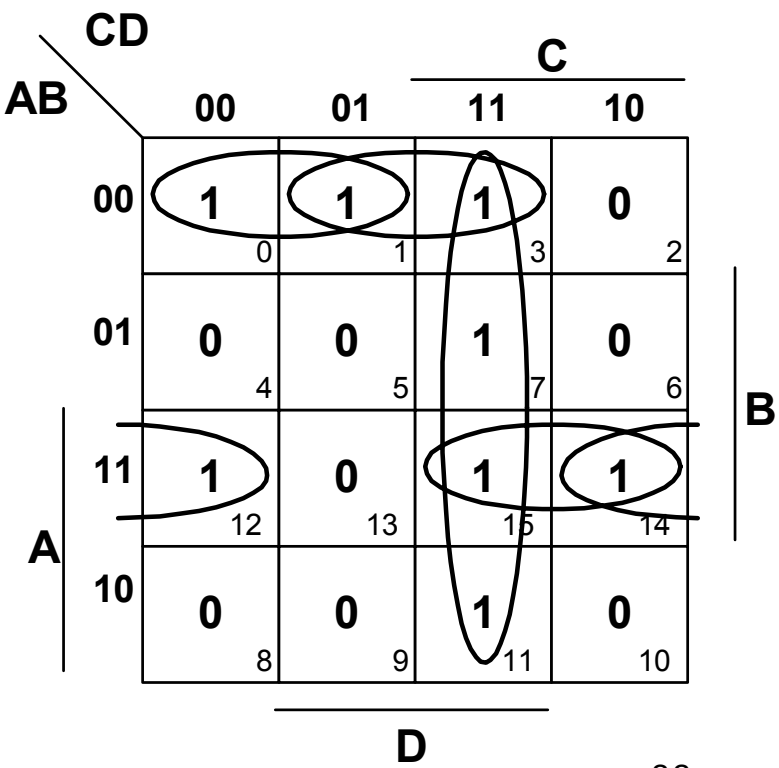
Számjegyes minimalizálás
Quine-McCluskey módszer III.lépés

- III. Összes létező szomszédos kettesekből képzett **négyelemű** lefedő tömb összevonása (Karnough tábla alapján)

Minterm	Decimális különbség
0,1	(1)
1,3	(2)
3,7	(**4**)
3,11	(8)
12,14	(2)
7,15	(8)
11,15	(**4**)
14,15	(1)

Négyes Összevonás

3,7,11,15 (4,8)



Számjegyes minimalizálás
Quine-McCluskey módszer IV.lépés

- IV. Prímimplikáns tábla felírása a megmaradt összevonásokkal (III. lépés alapján)

sor		0	1	3	7	11	12	14	15
*	**0,1** (1)	*	*						
	1,3 (2)		*	*					
*	**12,14** (2)						*	*	
	14,15 (1)							*	*
*	**3,7,11,15** (4,8)			*	*	*			*

* : ahol egy adott mintermhez tartozó oszlopban csak egy '*' van, az a sor jelöli a **lényeges prímimplikánst** (ahol az implikáns tovább már nem egyszerűsíthető!). Az a sor nem elhagyható!

Számjegyes minimalizálás
Quine-McCluskey módszer V.lépés

- V. Prímimplikánsokból képzett kimeneti függvény megadása (IV. lépés alapján):
  - **(0,1):** 0000, 0001 ⇒ 000**-**
  - **(12,14):** 1100, 1110 ⇒ 11**-**0
  - **(3,7,11,15):** 0011, 0111, 1011, 1111 ⇒ **--**11

A mintermen belüli egyszerre 0/1 tagok kiesnek!

- Tehát a kimeneti minimalizált F függvény a következő:

$$F = 000\text{-} + 11\text{-}0 + \text{--}11 \Rightarrow \quad F = \overline{A} \cdot \overline{B} \cdot \overline{C} + A \cdot B \cdot \overline{D} + C \cdot D$$

- Ajánlott: fejezetek végén a feladatok (Exercises) részek áttekintése.

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

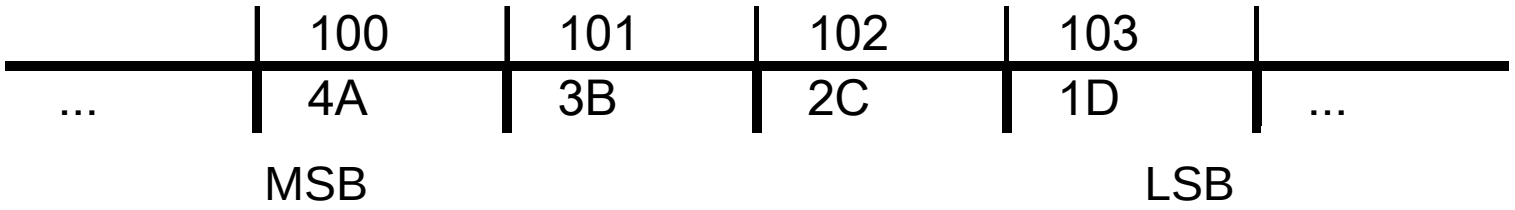
Endianitás (endianness)

- 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

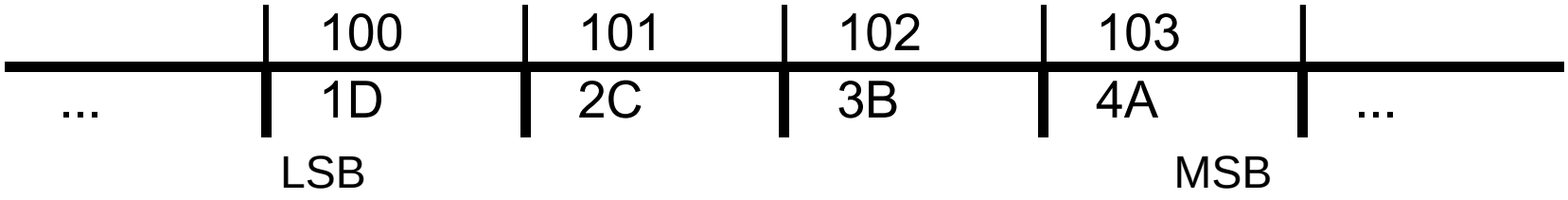
- A 32 bites egész értéket, (ami legyen „4A 3B 2C 1D”, a 100 címtől kezdve) tároljuk a memóriában, **1 byte-os** elemi tárolókból, 1 byte-onként növekvő címekkel rendelkezik, ekkor a tárolást a következők szerint végzi:



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

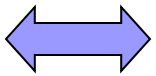
b.) előjeles (kettes komplemens) rendszer:

- 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.

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



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

2.2 táblázat:

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

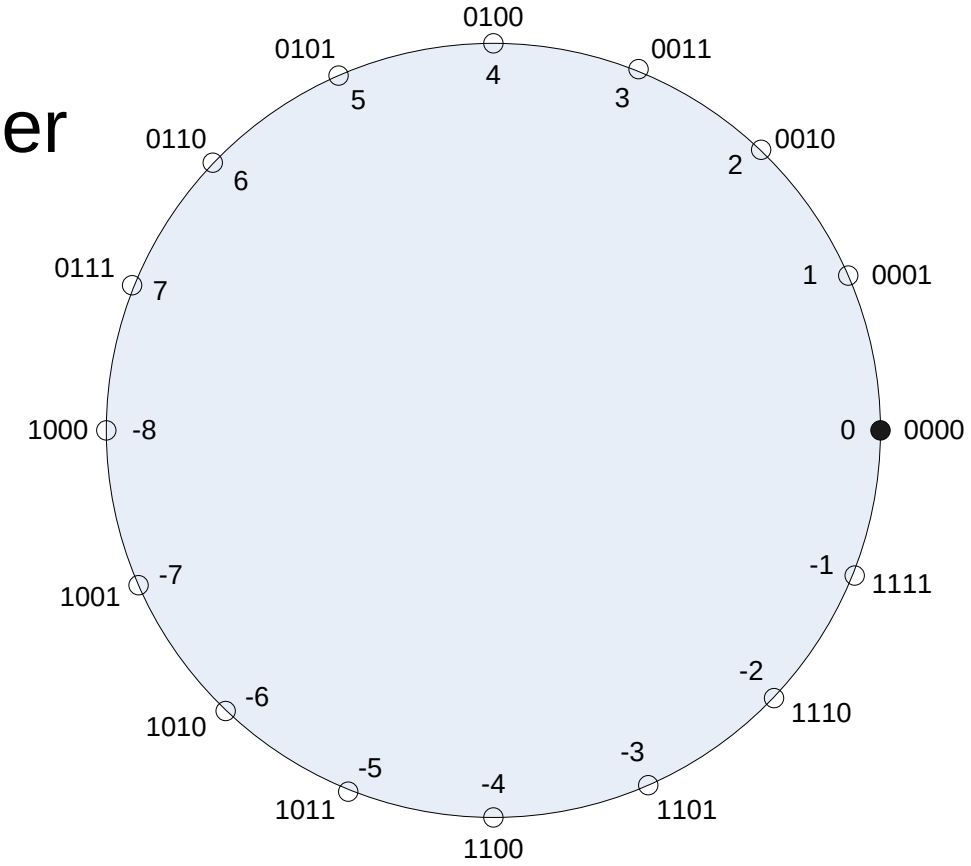
Bit Pattern	*Value*	*Note*
01111111	127	Largest representable value.
01111110	126	
01111101	125	
...	...	
...	...	
00000010	2	Note that leading zero indicates
00000001	1	positive number.
00000000	0	Unique representation of **zero.**
11111111	–1	Minus one is always all ones.
11111110	–2	Note that leading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

Pl: Körkörös számláló (circular nature):

- 4 bites 2’s komplemens rendszer
- Overflow:

  0111 -> 1000
- Underflow:

  1000 -> 0111



c.) 1-es komplemens rendszer:

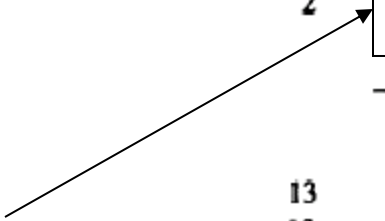
- V értékű, N bites rendszer: $\mathbf{2^N–1–V}$.
- „0” lesz ott ahol „1”-es volt, „1”-es lesz ott ahol „0” volt (mivel egy szám negatív alakját, bitjeinek kiegészítésével kapjuk meg).
- csupán minden bitjét negálni, (gyors műveletet)
- Értékhatár: $2^{(N-1)}–1$ től $–(2^{(N-1)}–1)$ ig terjed,
- Nem 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

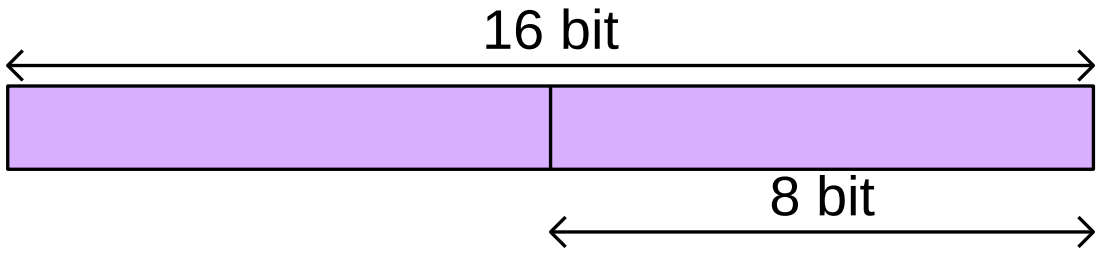
$$V_{\text{FIXEDPOINT}} = -b_{N-1} \times 2^{N-p-1} + \sum_{i=0}^{N-2} b_i \times 2^{i-p}$$

- 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:

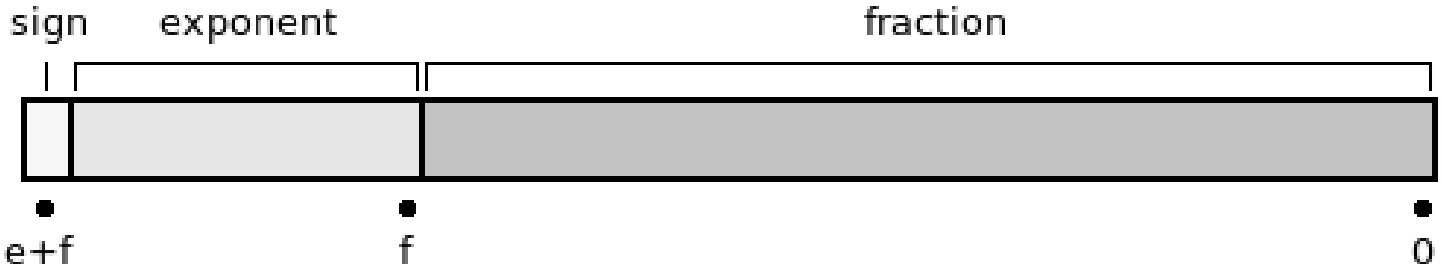
- 7 különböző tényező: *a számrendszer alapja, előjele és nagysága, a mantissza alapja, előjele és hosszúsága, ill. a kitevő alapja*.
- matematikai jelölés:

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

- 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_2 $= \frac{15}{16}$

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

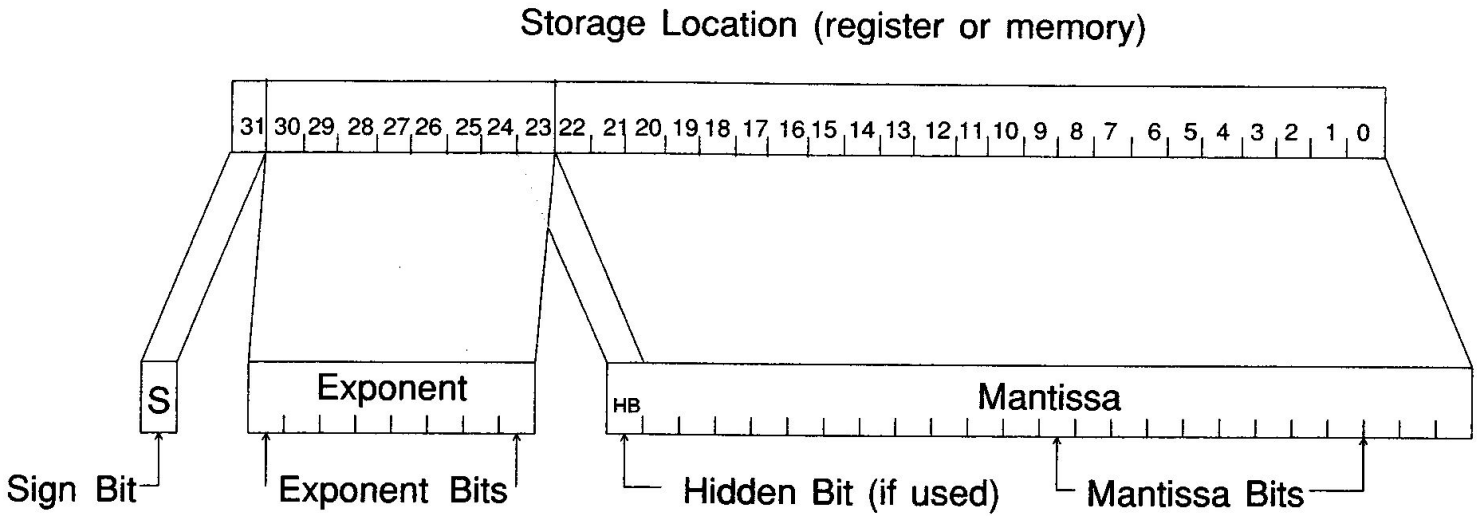
Largest number = $0.1111_2 \times 2^3$ $= 7\frac{1}{2}$

Number of fractions = $1 \times 2 \times 2 \times 2 = 8$

Number of values = 8×4 $= 32$

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).

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

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

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

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

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

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

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

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

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

- *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

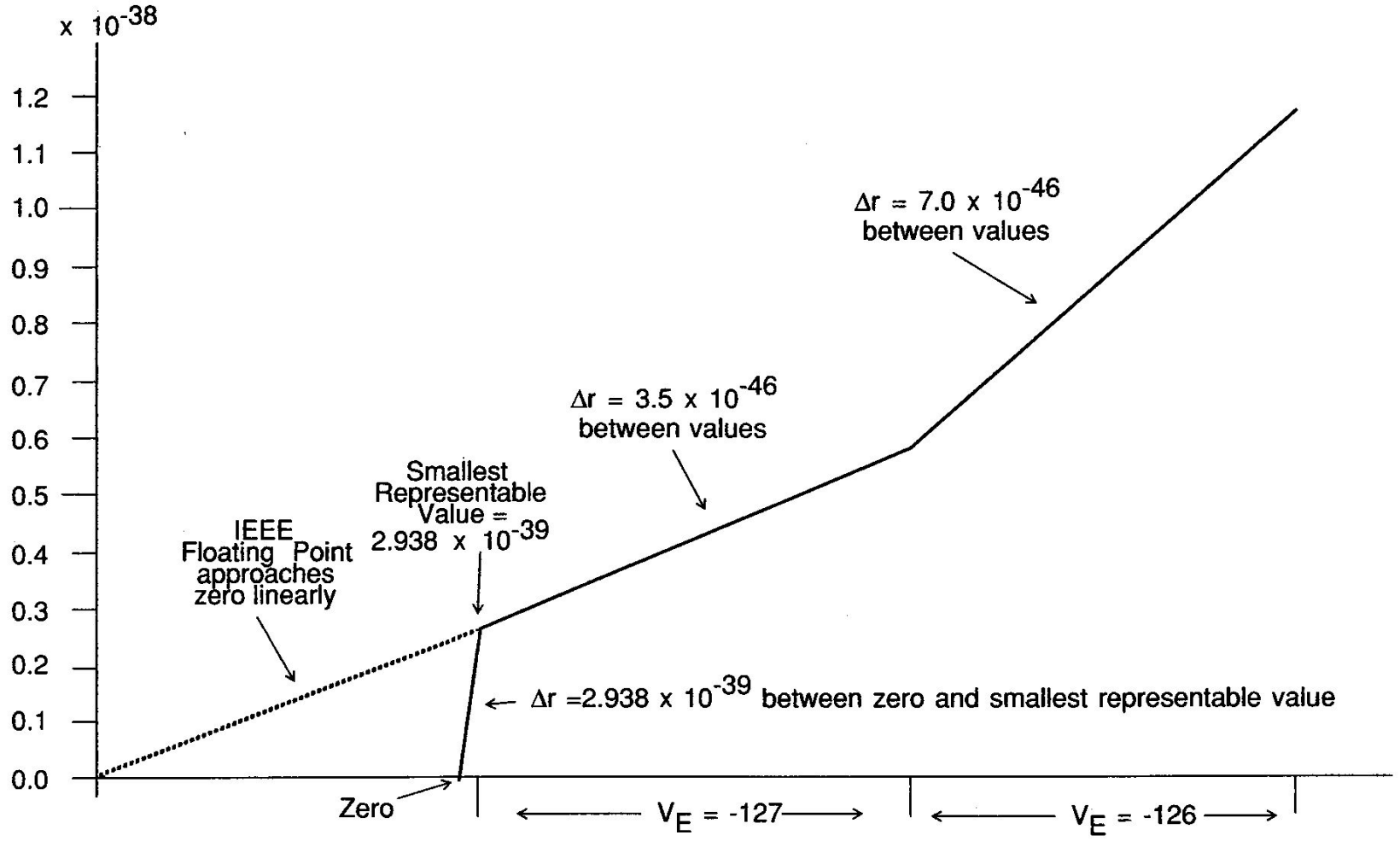


Figure 2.3. Values of the DEC Normalized Floating Point System Near Zero.

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

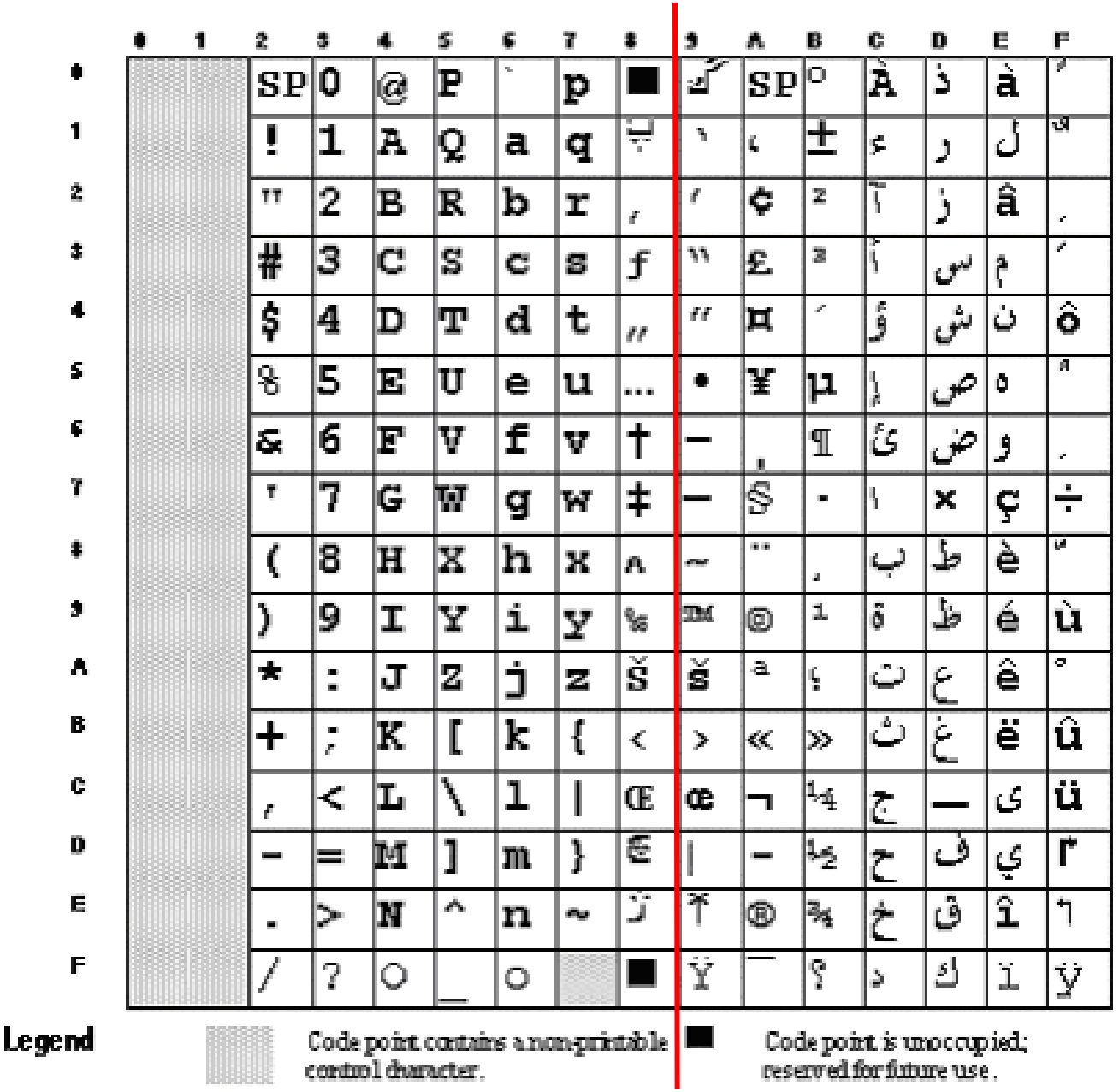
Szöveges információ kódolás

- 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	÷ SA080000	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

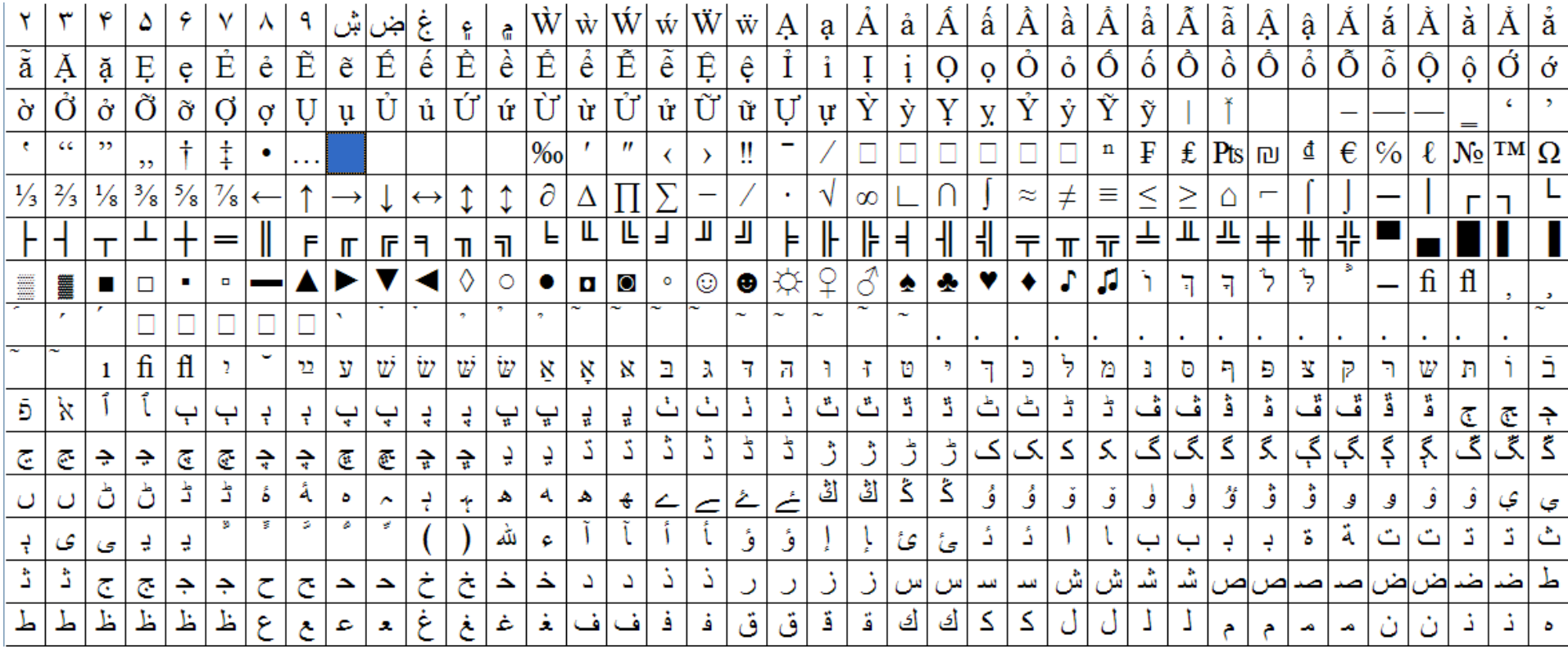
Extended ASCII (1 byte)



Unicode

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

Általános írásjelek, matematikai, pénzügyi, mértani szimbólumok stb.



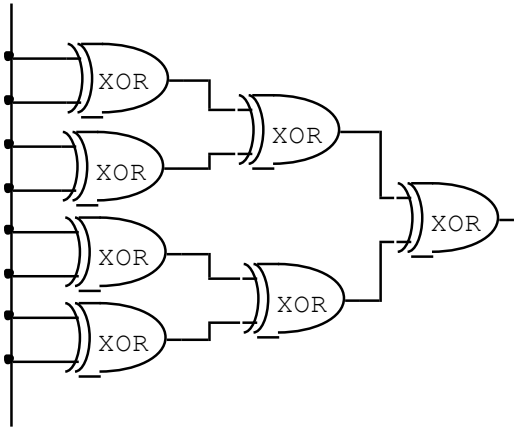
*részlet

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

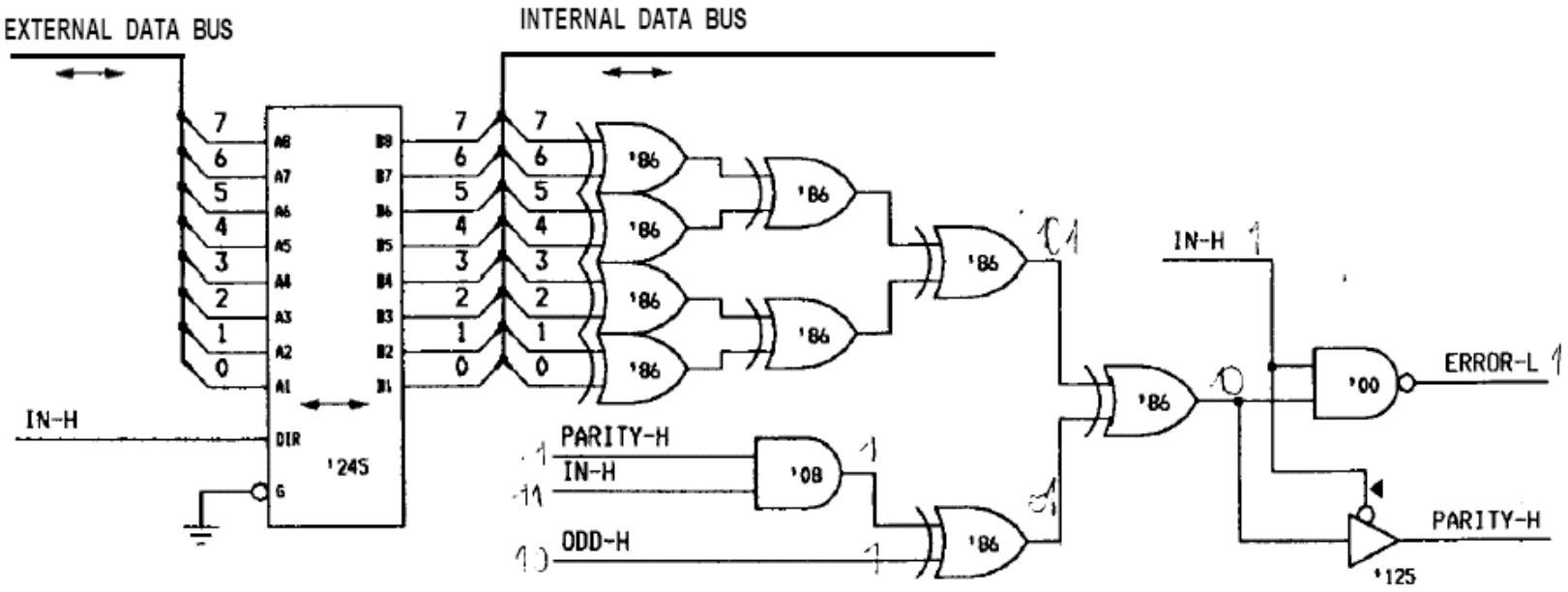
Paritás bit ellenőrzés

- 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

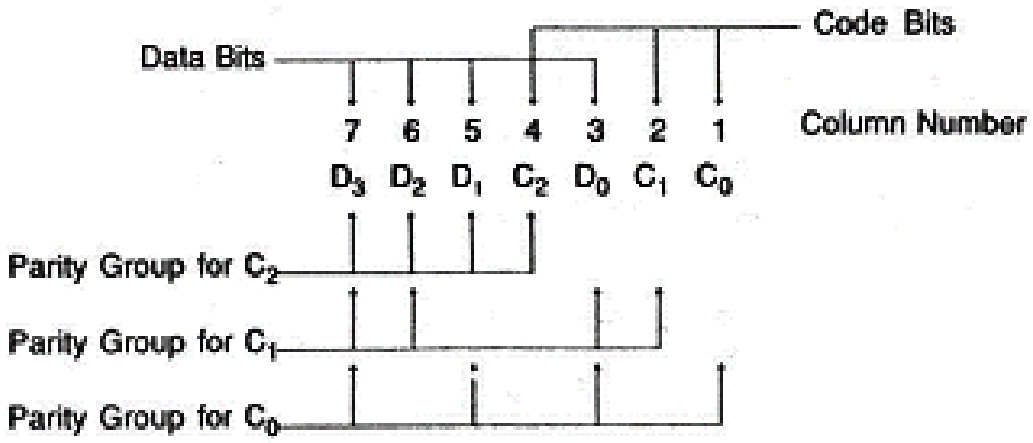


Hamming kód

- 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)

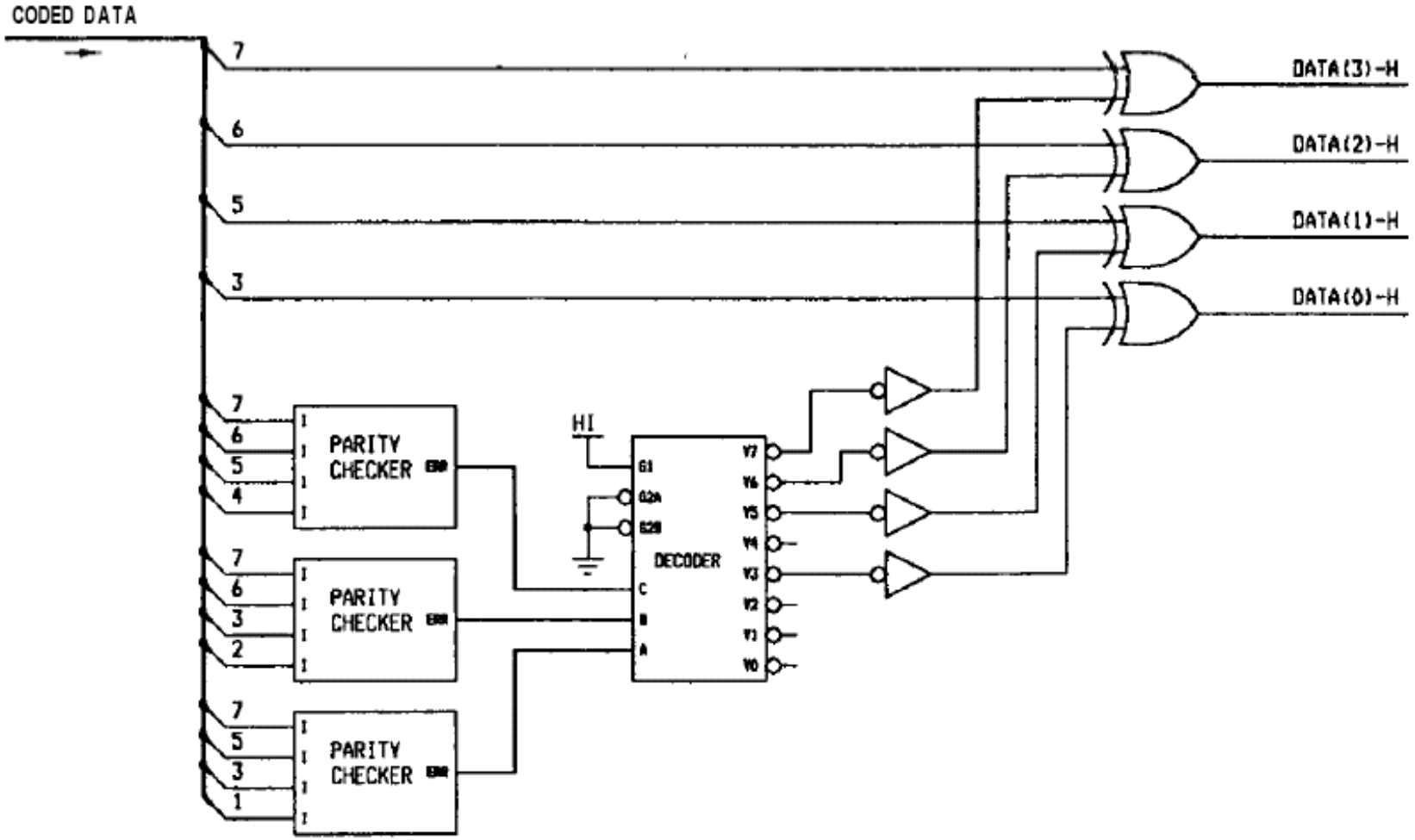


Paritás-csoportok	Bit pozíciók	Bitek jelölései
0	1, 3, 5, 7	C0, D0, D1, D3
1	2, 3, 6, 7	C1, D0, D2, D3
2	4, 5, 6, 7	C2, D1, D2, D3

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

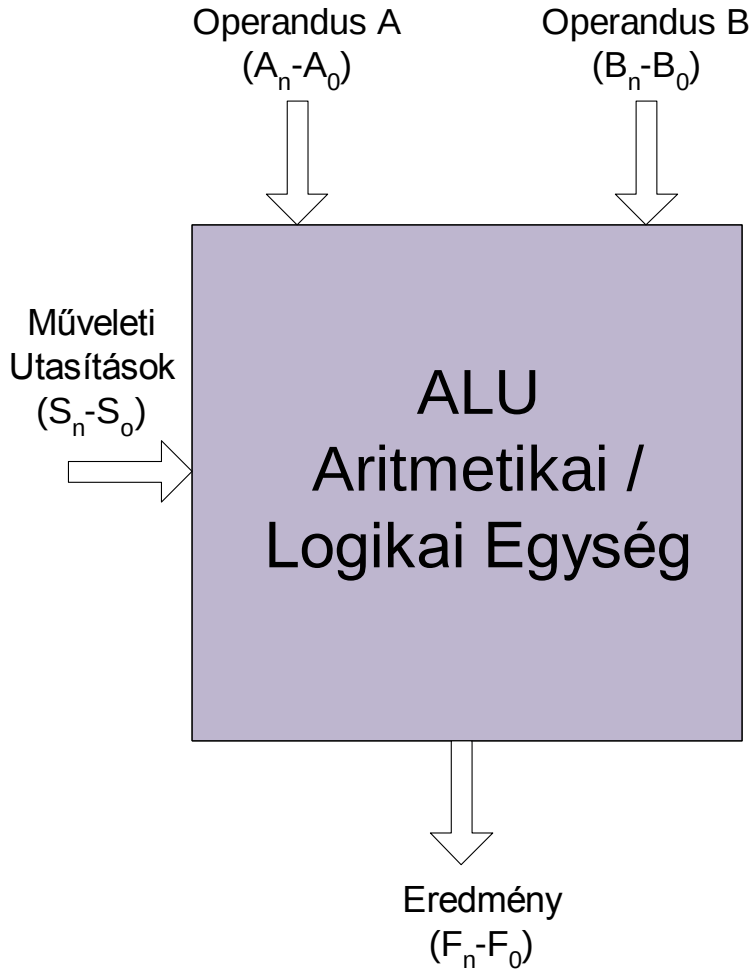
- Korai számítógépek teljesítményét főként ballisztikus számításoknál (hadászatban)
- Információ ábrázolás
- Használt utasításkészlet
- Adatkezelő / műveletvégző egység:
  - Alapvető ALU (Aritmetikai és Logikai funkciók)
    - Aritmetikai operátorok: +,-,*,/ (alapműveletek)
    - Logikai operátorok: 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

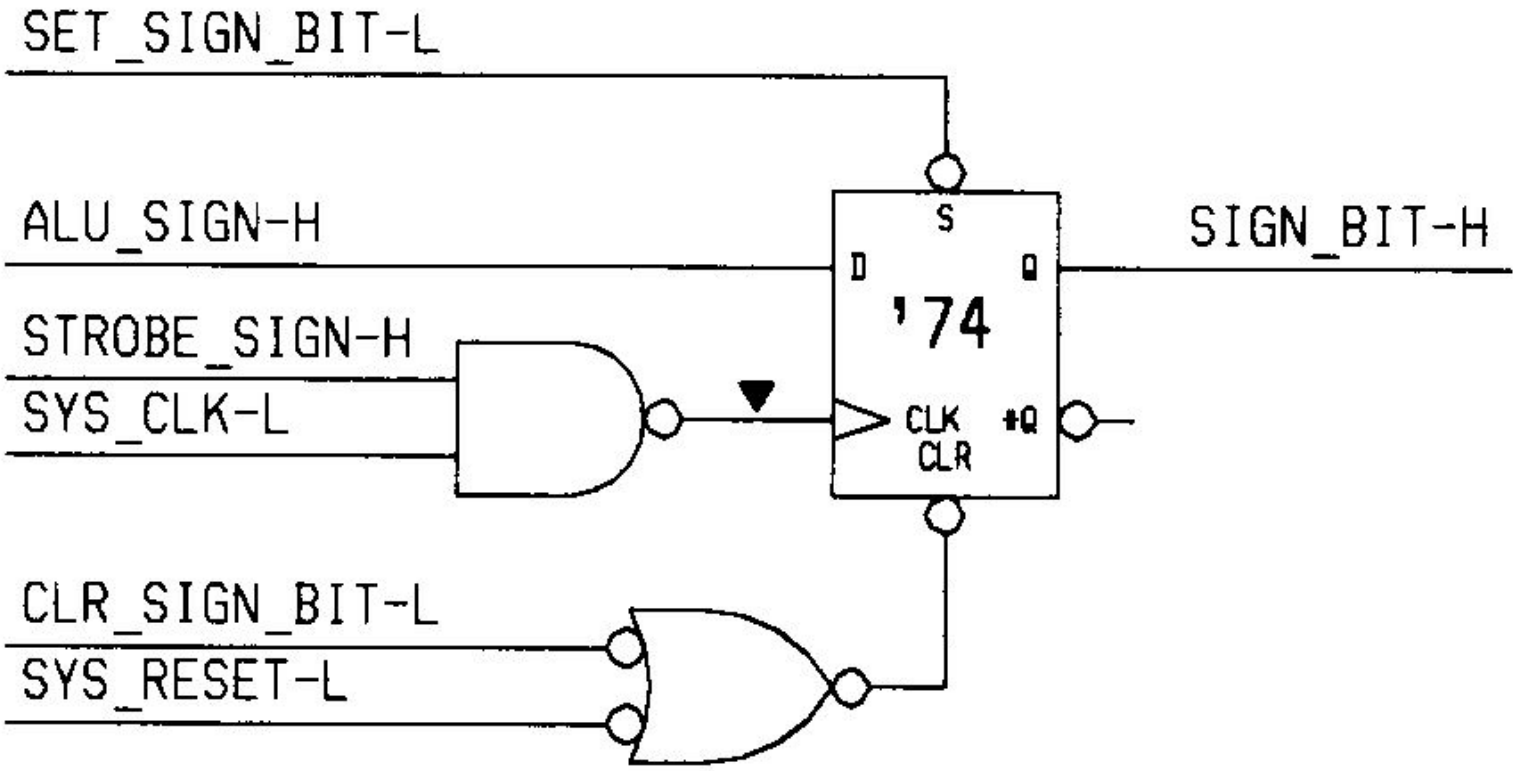
- Utasítások hatására a (S_n-S_0) vezérlőjelek kijelölik a végrehajtandó aritmetikai / logikai műveletet. További adatvonalak kapcsolódhatnak közvetlenül a státusz regiszterhez, amely fontos információkat tárol el: pl.
  - *zero bit*
  - *carry-in, carry-out* átviteleket,
  - *előjel* bitet (sign),
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



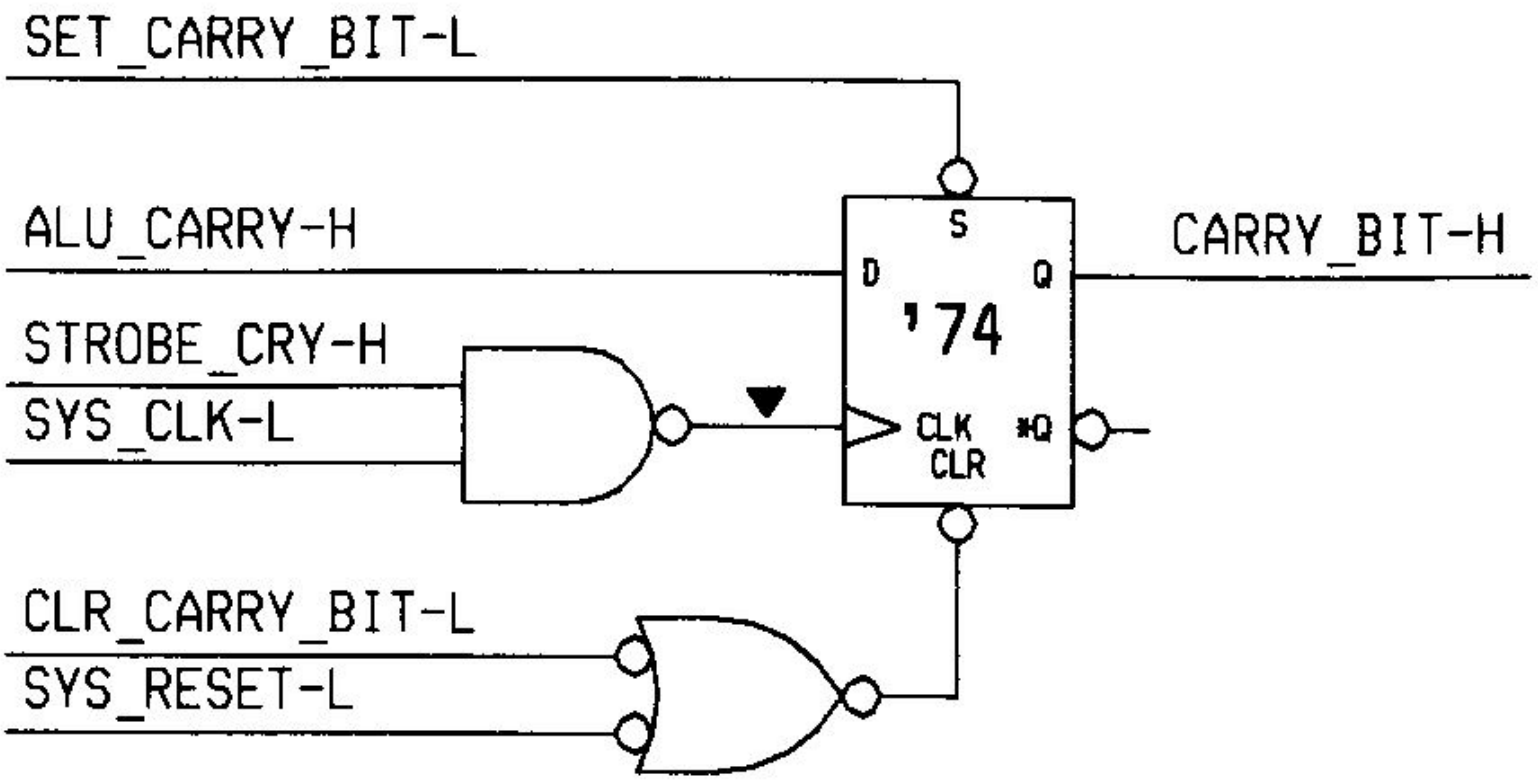
Státusz- (flag) jelzőbitek

- 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?

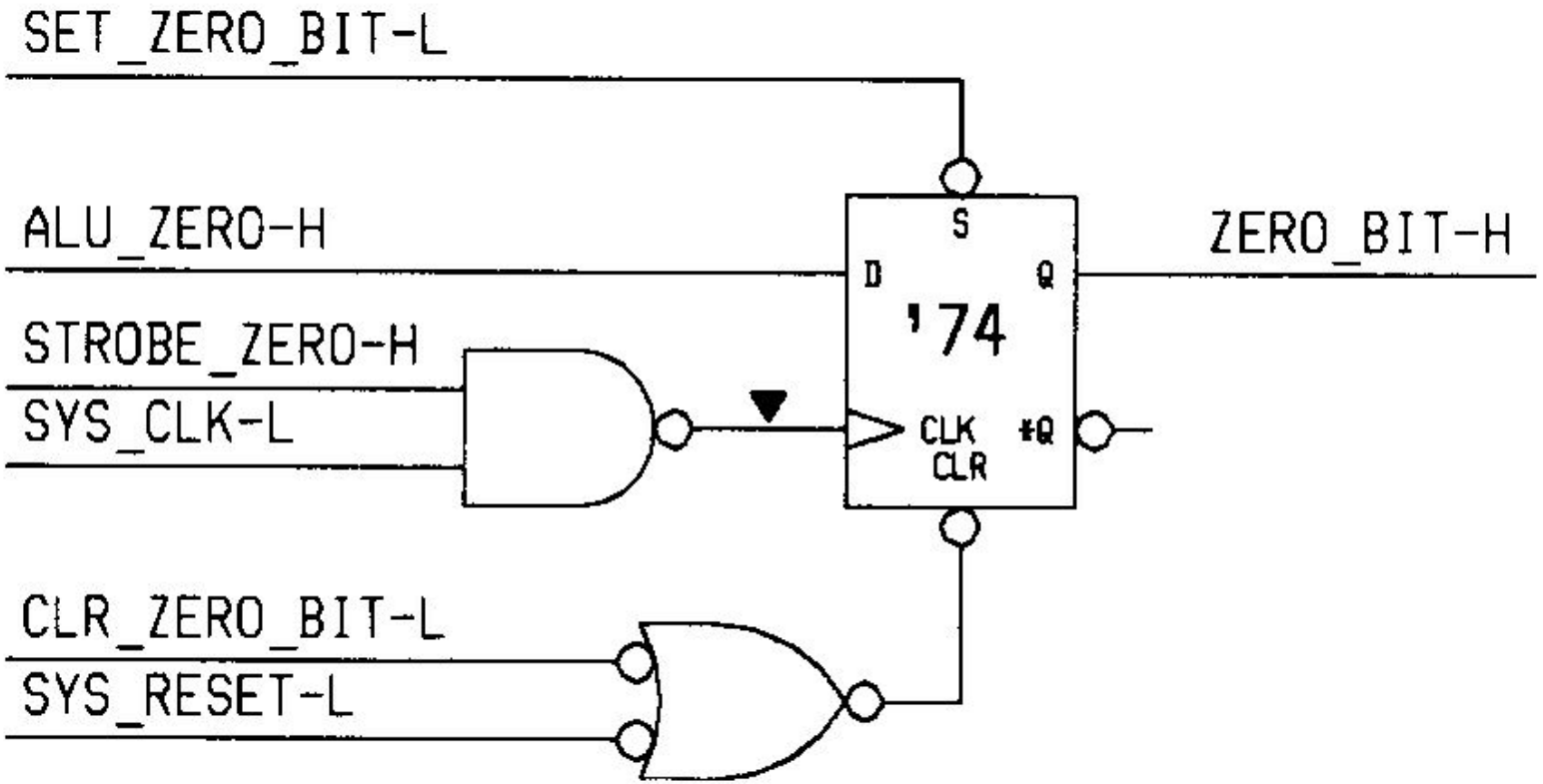
a.) Előjelbit (sign)



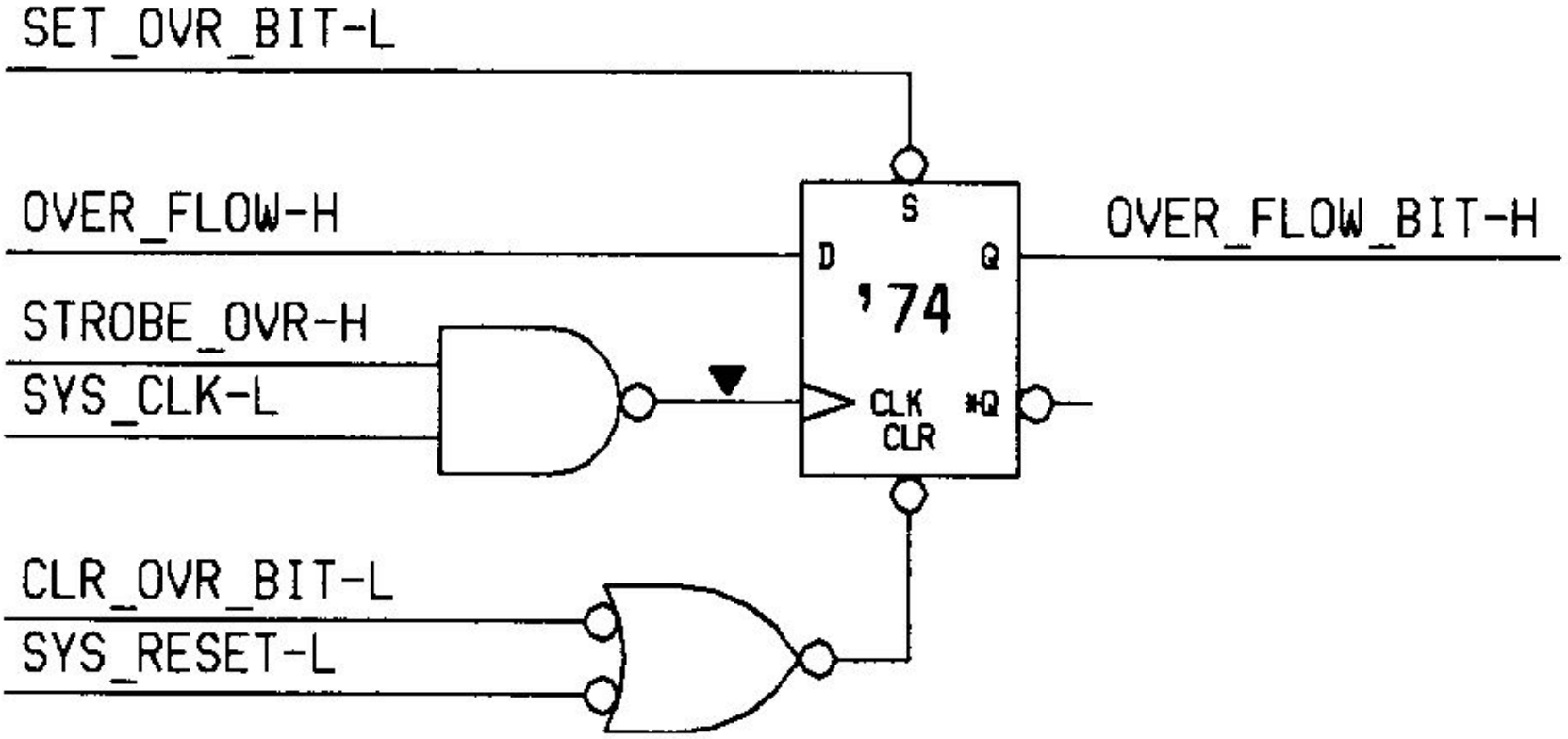
b.) Átvitel-kezelő bit (carry)



c.) Zéró bit

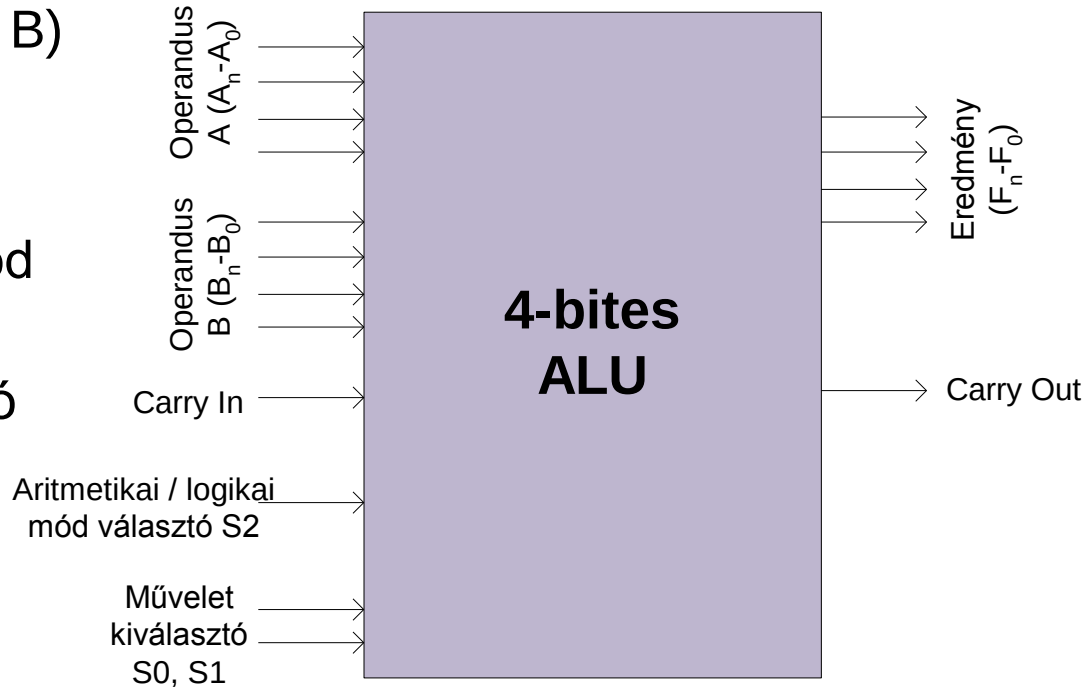

SET_ZERO_BIT-L
ALU_ZERO-H
STROBE_ZERO-H
SYS_CLK-L
CLR_ZERO_BIT-L
SYS_RESET-L
S
D
Q
'74
CLK
CLR
*Q
ZERO_BIT-H

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)



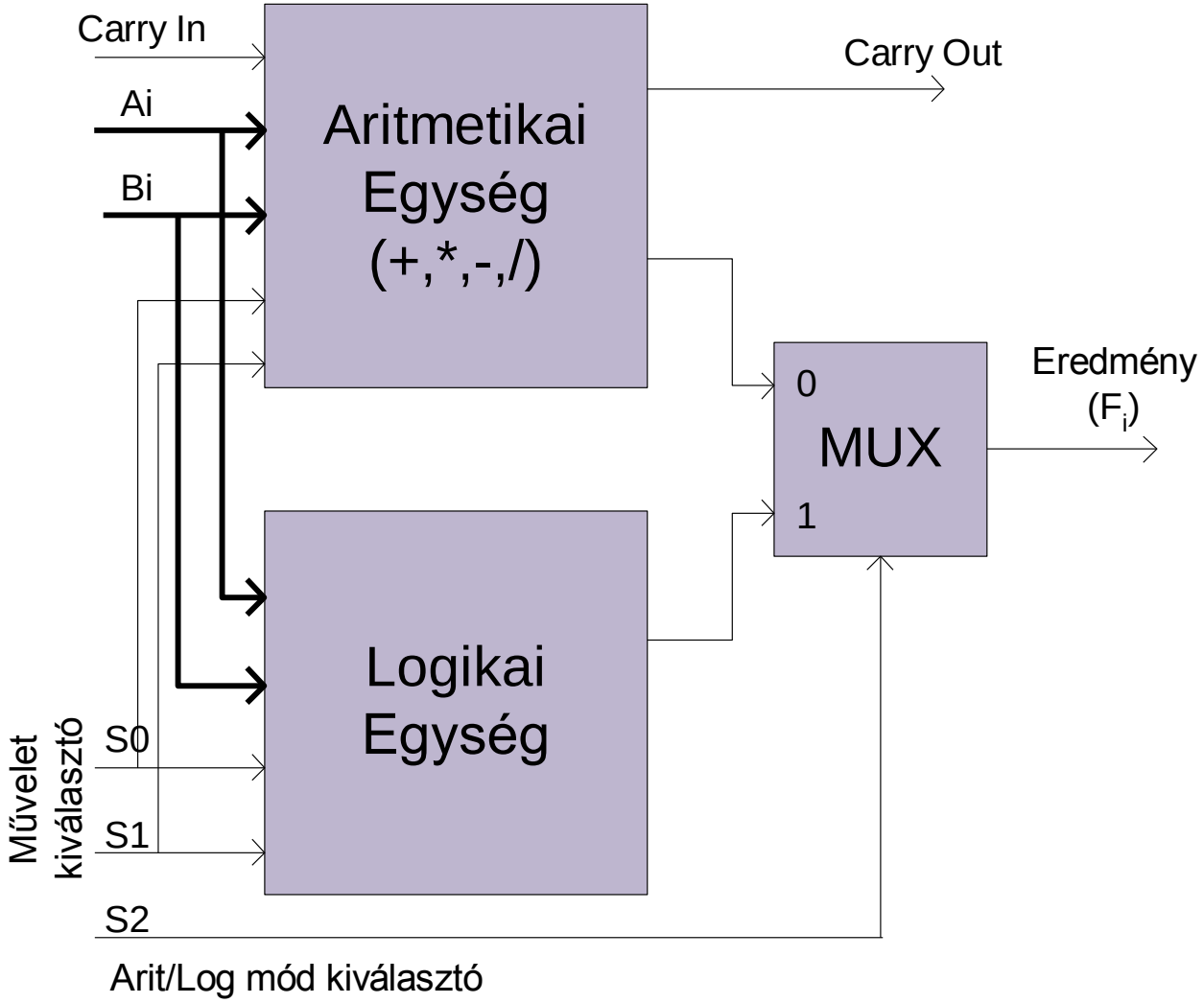
4-bites ALU szimbolikus rajza

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:



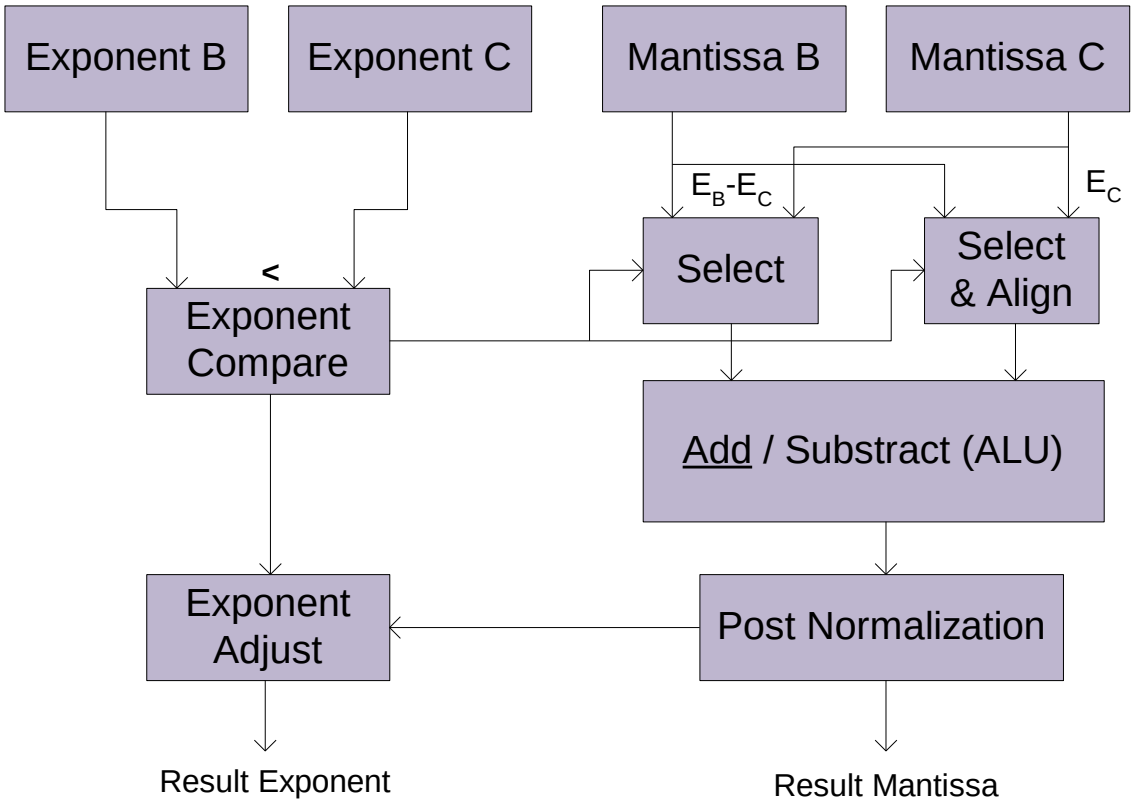
Lebegőpontos műveletvégző egységek

Lebegőpontos műveletvégző egységek

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

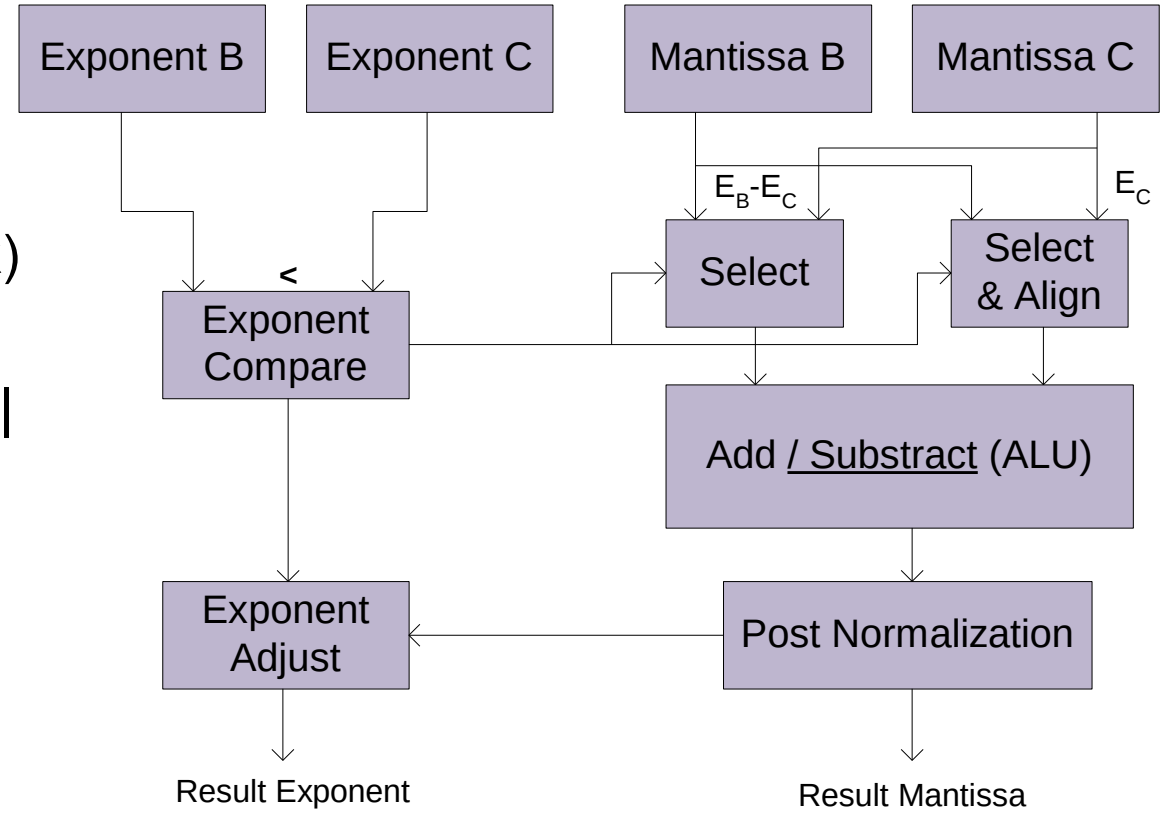
a.) Lebegőpontos összeadó

- 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



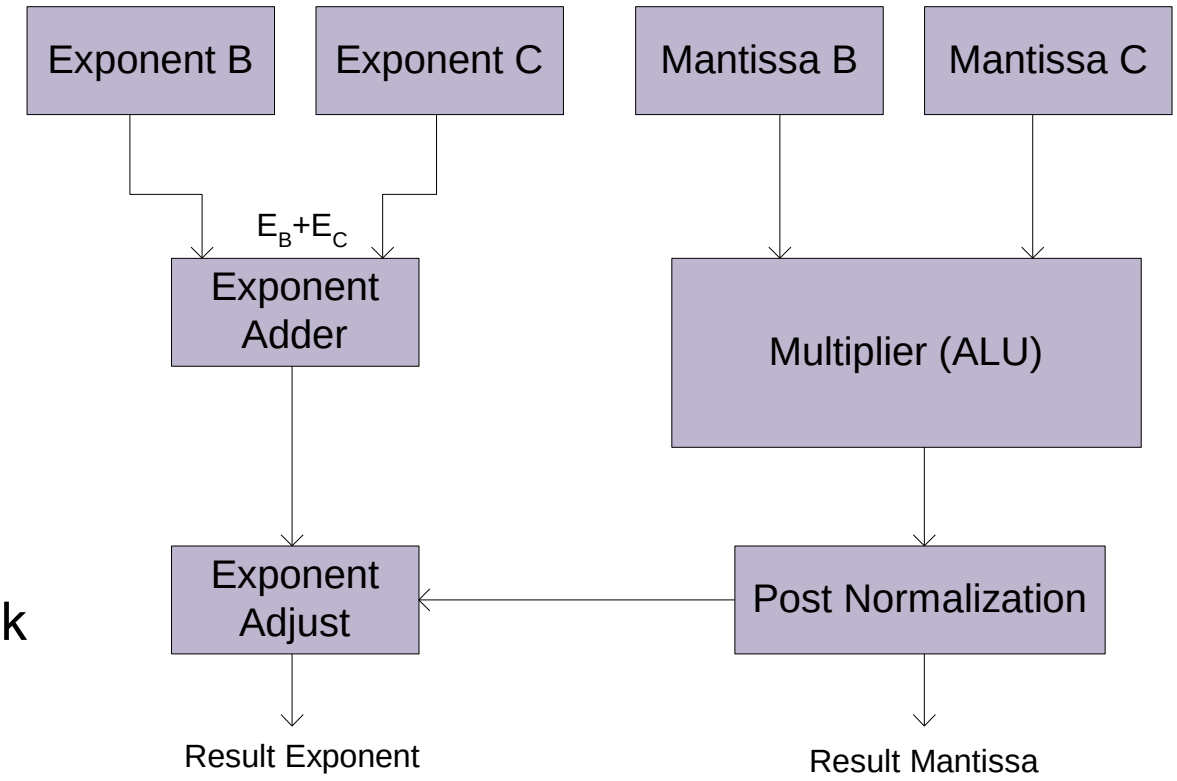
b.) Lebegőpontos kivonó

- 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)



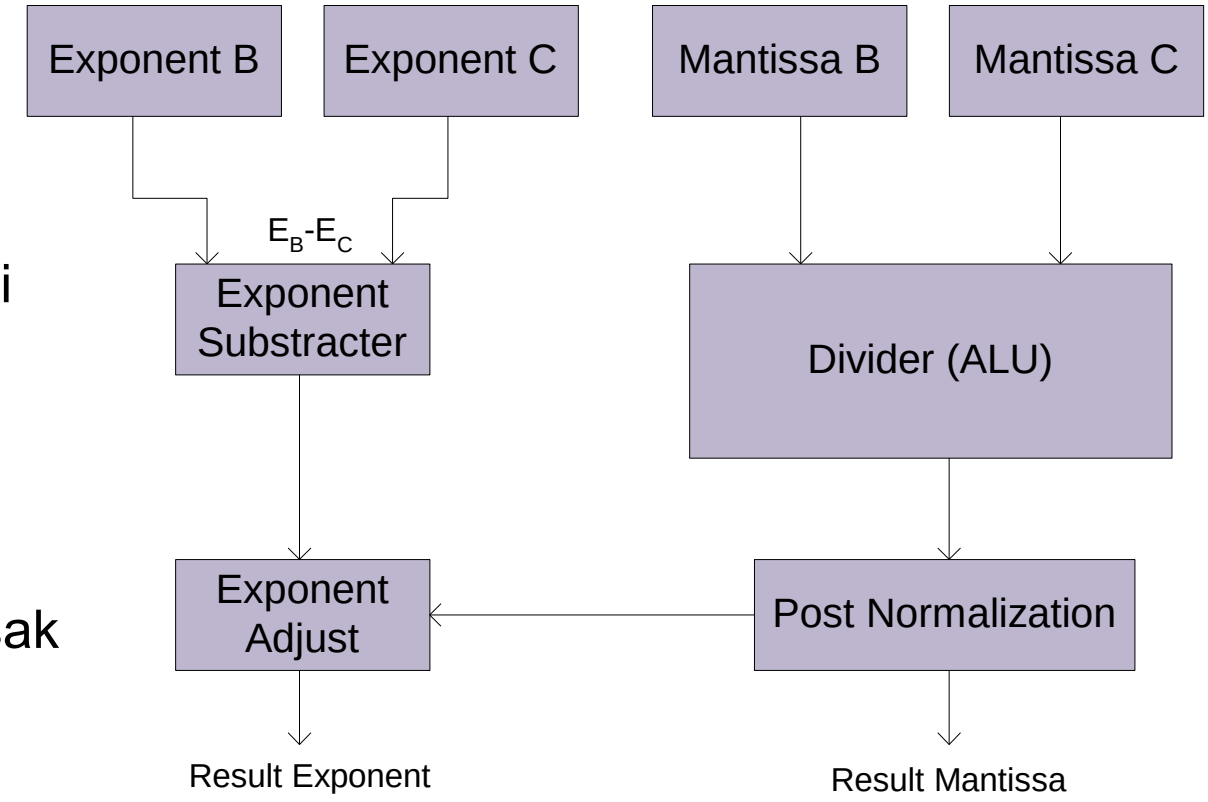
c.) Lebegőpontos szorzó

- 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

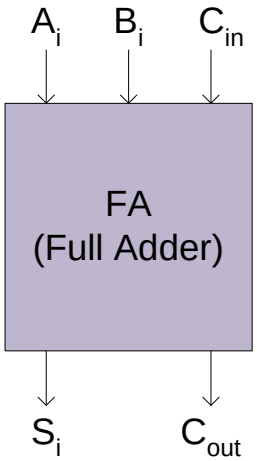
a.) Teljes összeadó – Full Adder

- FA: 1-bites Full Adder

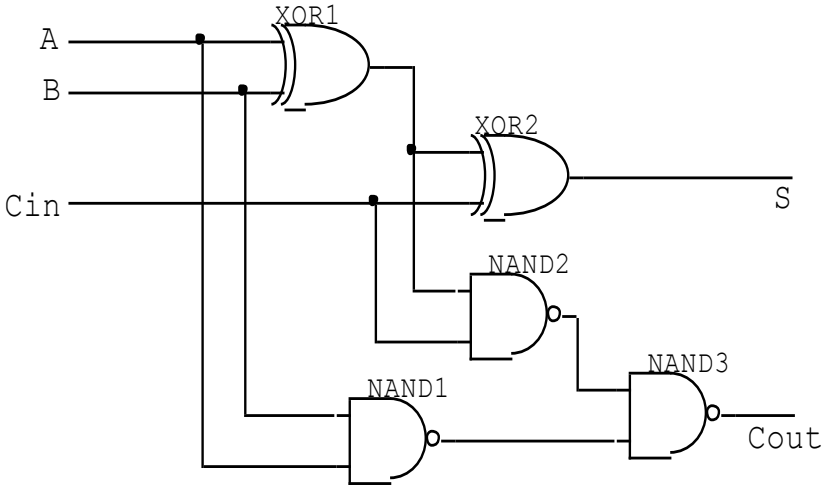
igazságtáblázat

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

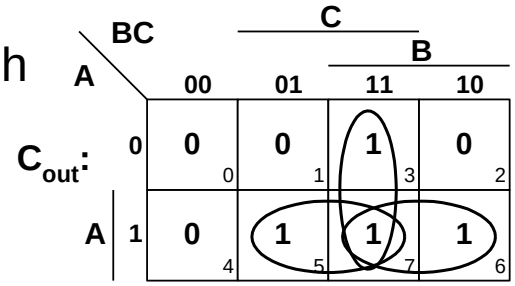
szimbólum

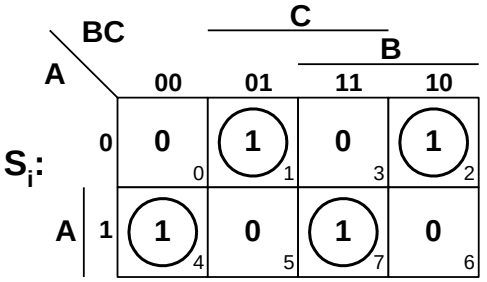


A FA egy CMOS kapcsolási rajza:



Karnaugh táblái:





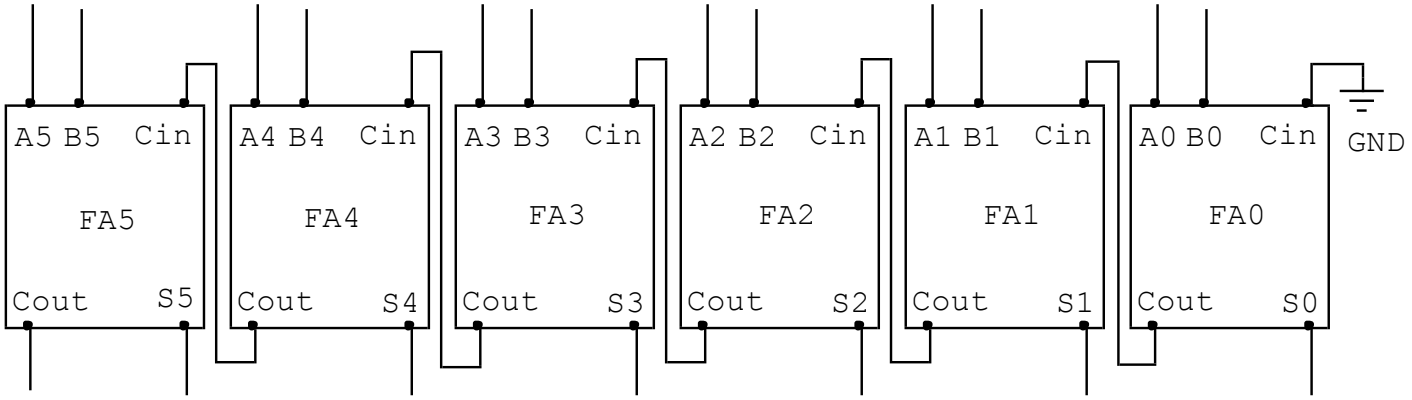
Kimeneti fgv-ei:

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

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

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.

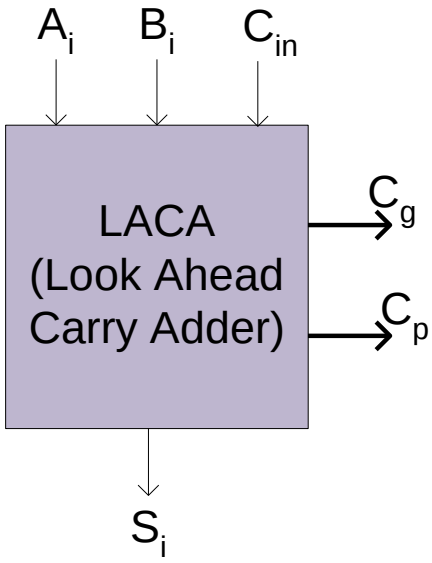
c.) LACA: Look Ahead Carry Adder:

- Képlet (FA) átírásából kapjuk:

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

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

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

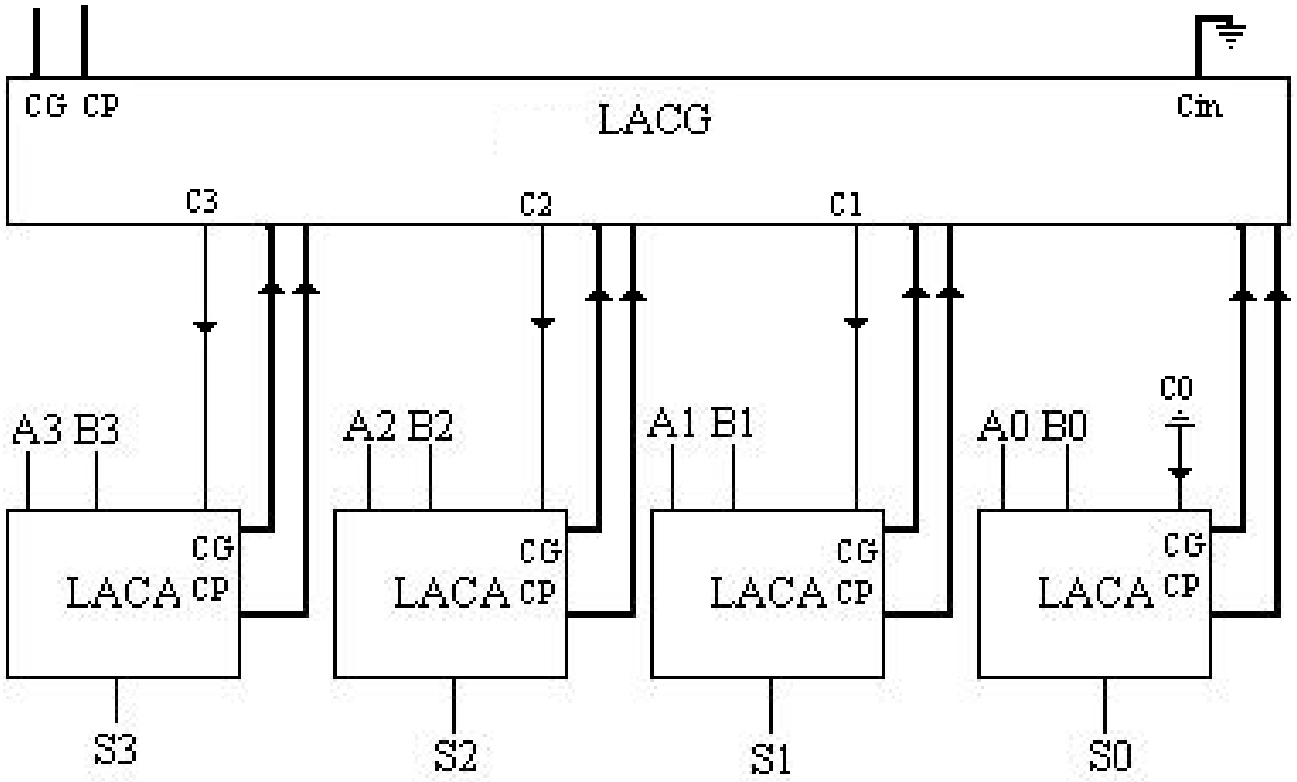


LACG: Look Ahead Carry Generator áll egy **b** bites ALU-ból, mindenegyes állapotban a Carry generálásáért felel a CP és CG (LACA) vonalakon érkező jeleknek megfelelően.
LACA számítási időszükséglete: $T_{LACA} = 2 + 4 \times (\lceil \log_b(N) \rceil - 1)$

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?



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

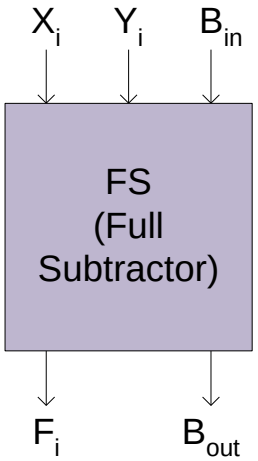
d.) Teljes kivonó - Full Subtractor (FS)

- FS: 1-bites Full Subtractor

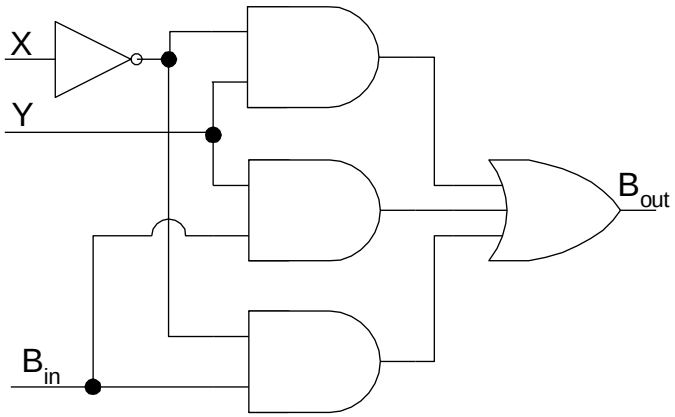
igazságtáblázat

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

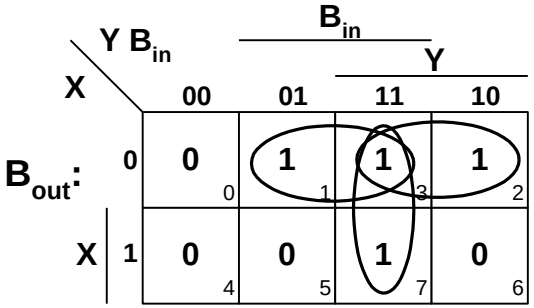
szimbólum

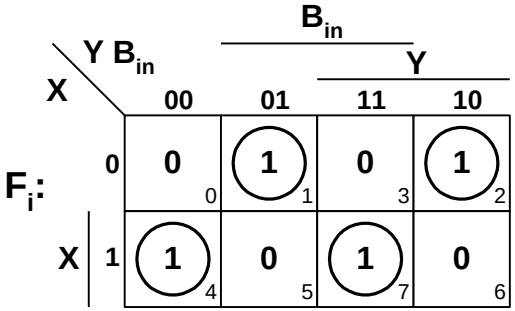


Logikai kapcsolási rajz (B out-ra)



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:

$$P = A \times B$$

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

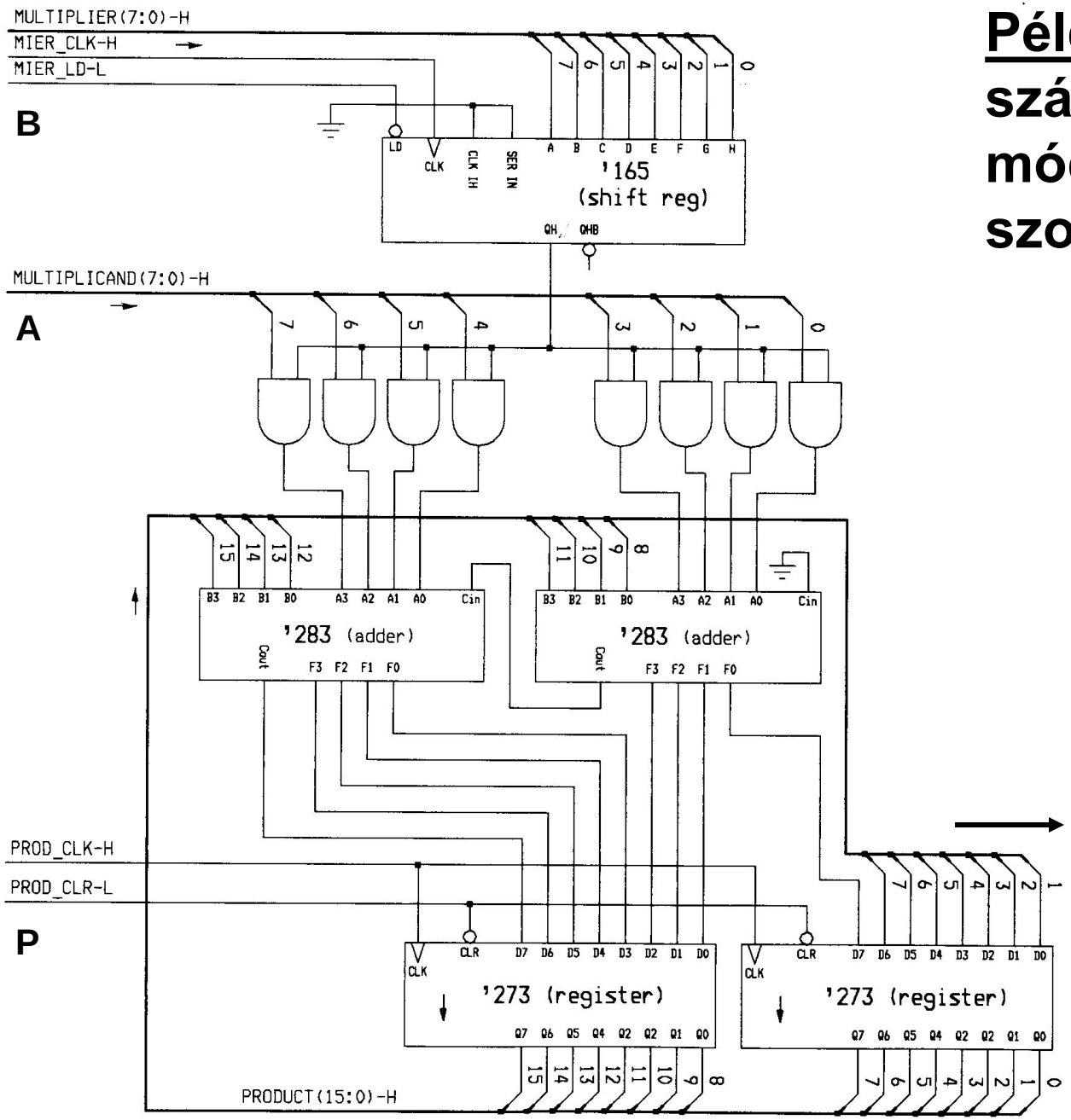
- Fordított sorrendű (MSB → LSB):

	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						

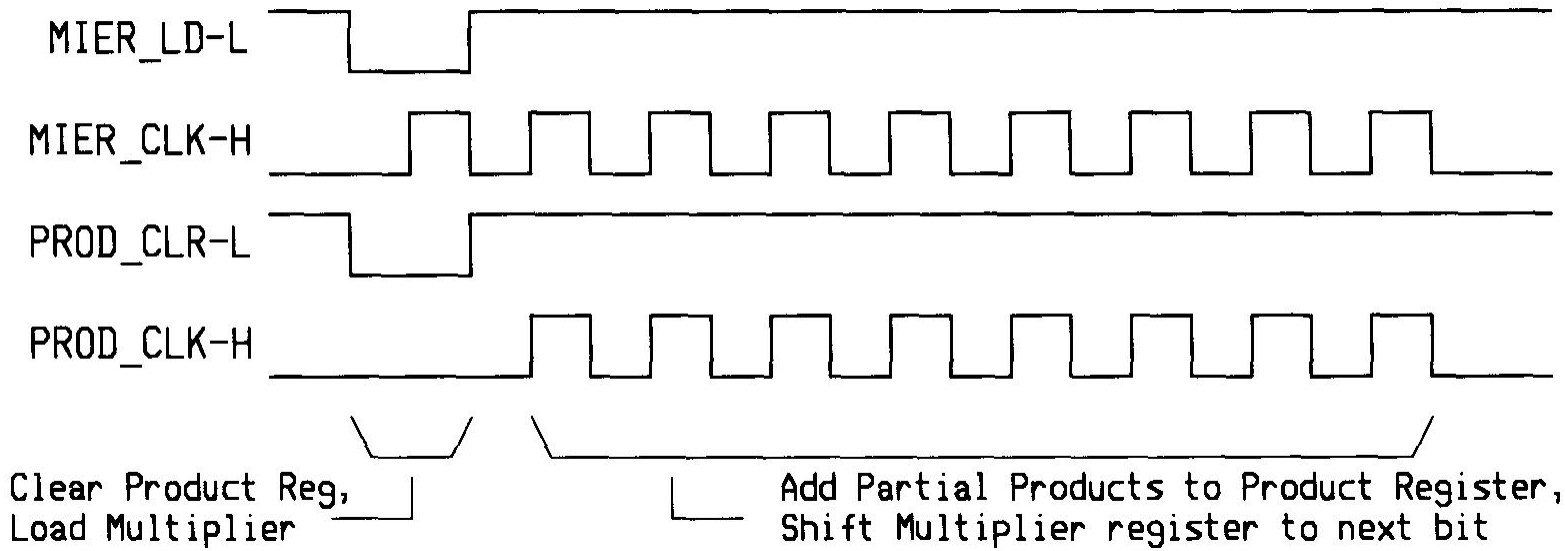
1.) Általános Shift&Add módszer

- 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



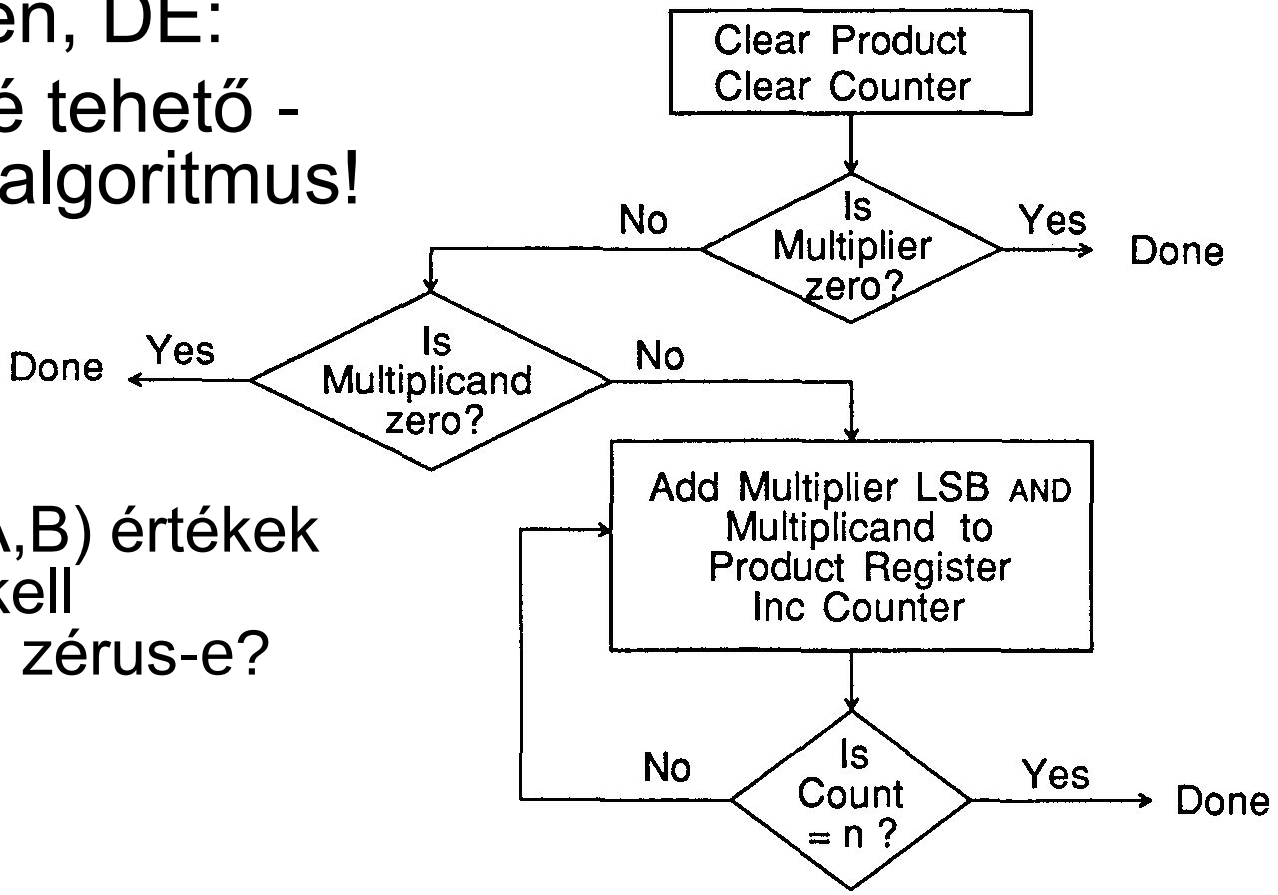
Időzítési jelek:



- 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)

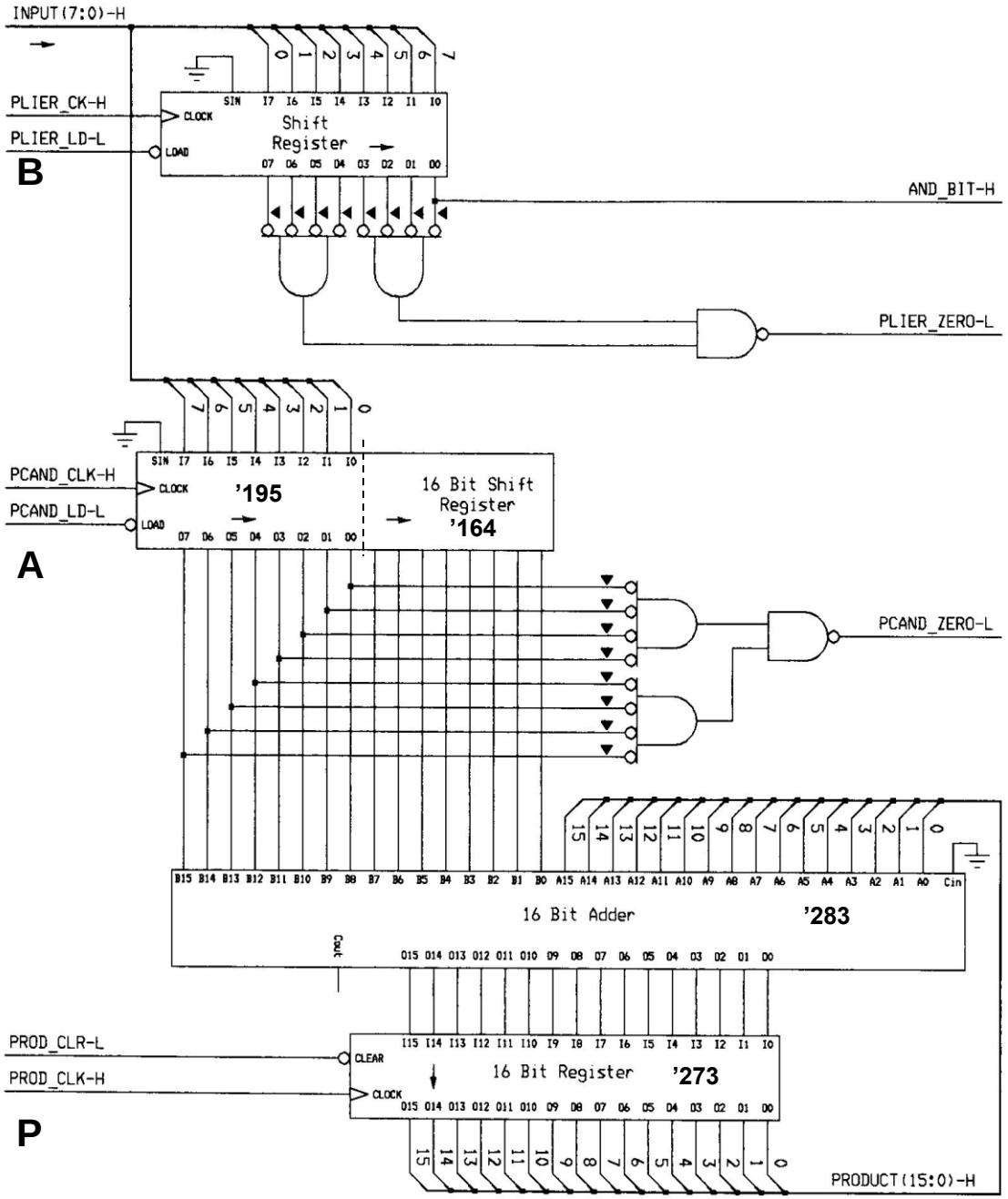
- Alapvetően adatfüggetlen, DE:
- Adatfüggővé tehető - gyorsítható algoritmus!
  - Bemenő (A,B) értékek figyelését kell megoldani: zérus-e?



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.

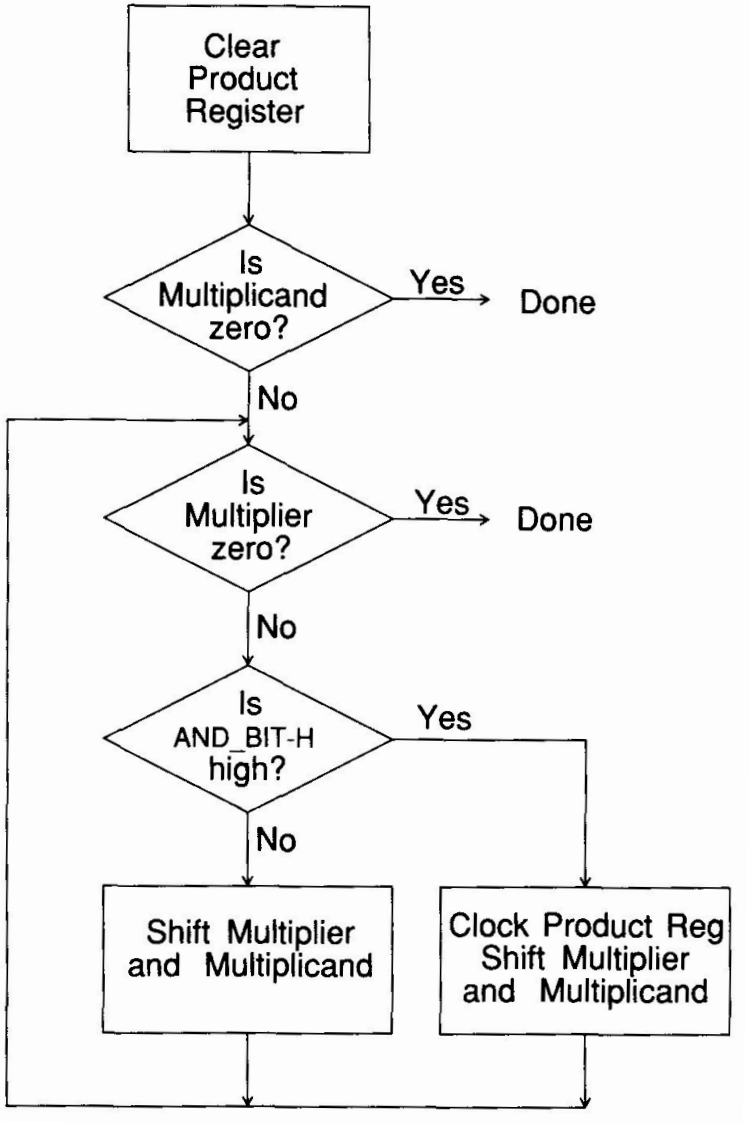
Folyamatábra (fordított sorrendű)

- 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”

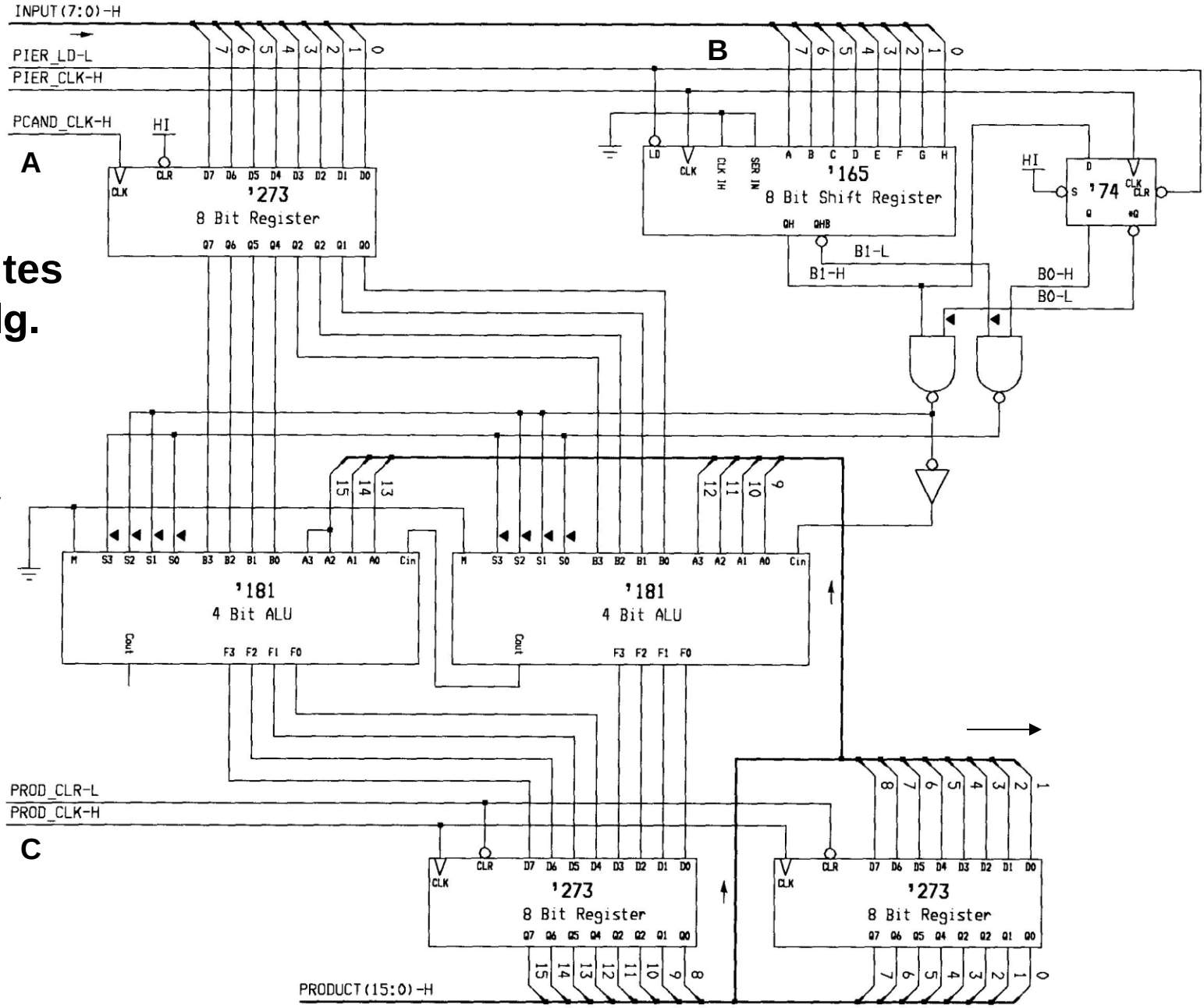
  101101=–19 „B” Végrehajtjuk **A*B**-t!

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.



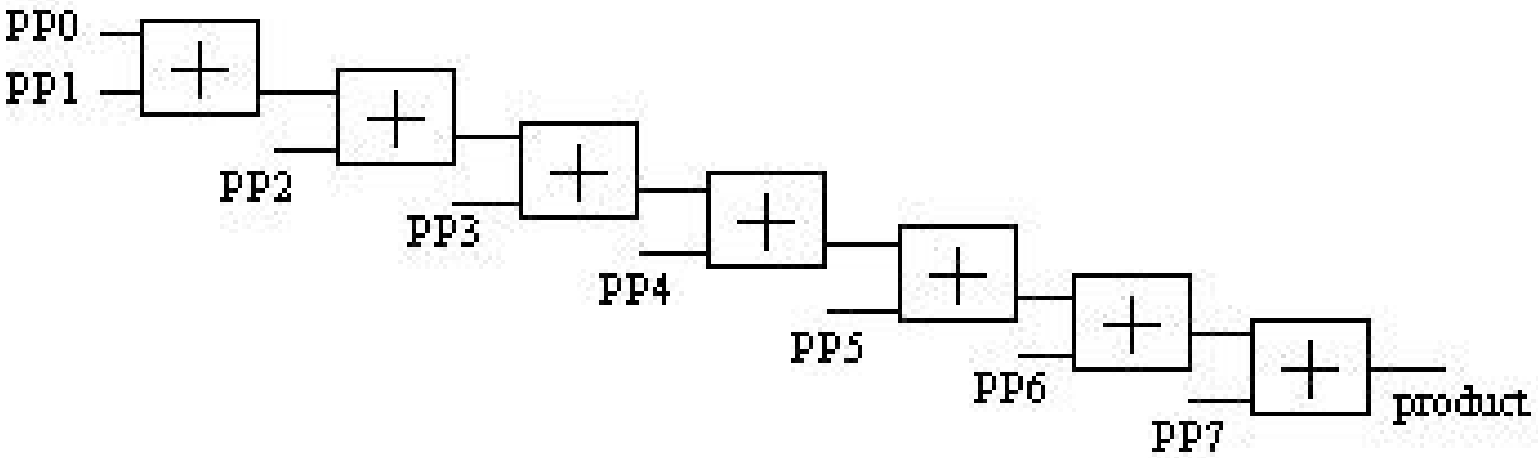
II. Közvetlen szorzási módszerek

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).

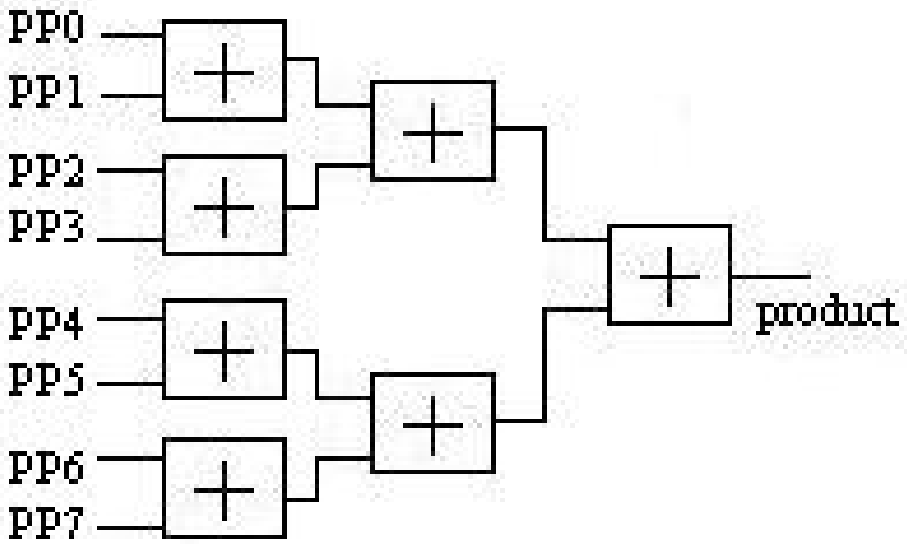
a.) Lineáris modell

- 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

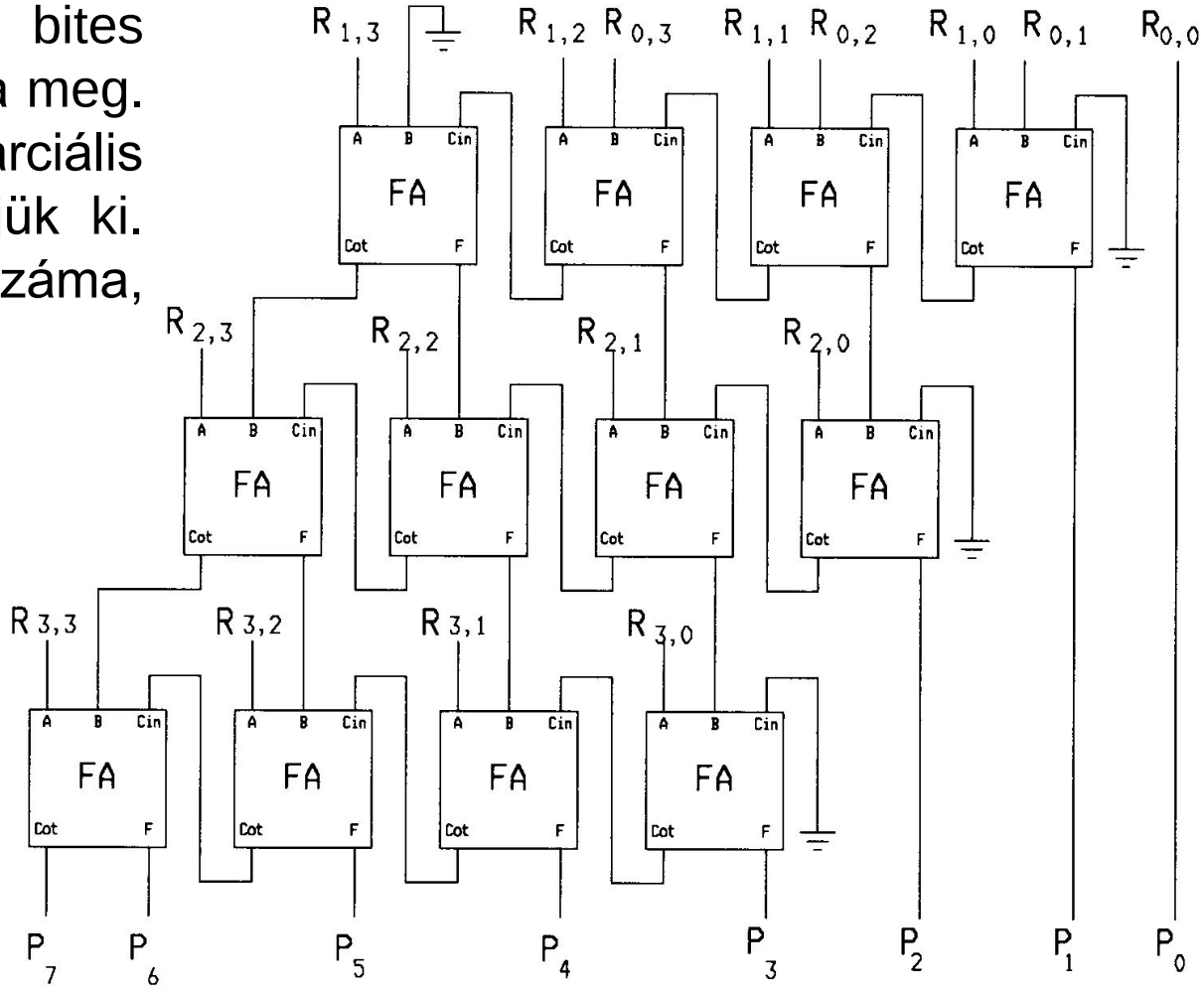
- A parciális szorzatok, képzésük után szintén azonnal összeadhatók, gyorsabban megkapjuk az eredményt, mint a lineáris modellnél, mivel ebben az esetben (N=8 bit esetén) csak 3 szintű a hierarchia, így kevesebb a késleltetés. N bites számok esetén (N-1) db összeadóra van szükségünk.
- Időszükséglet: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



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	



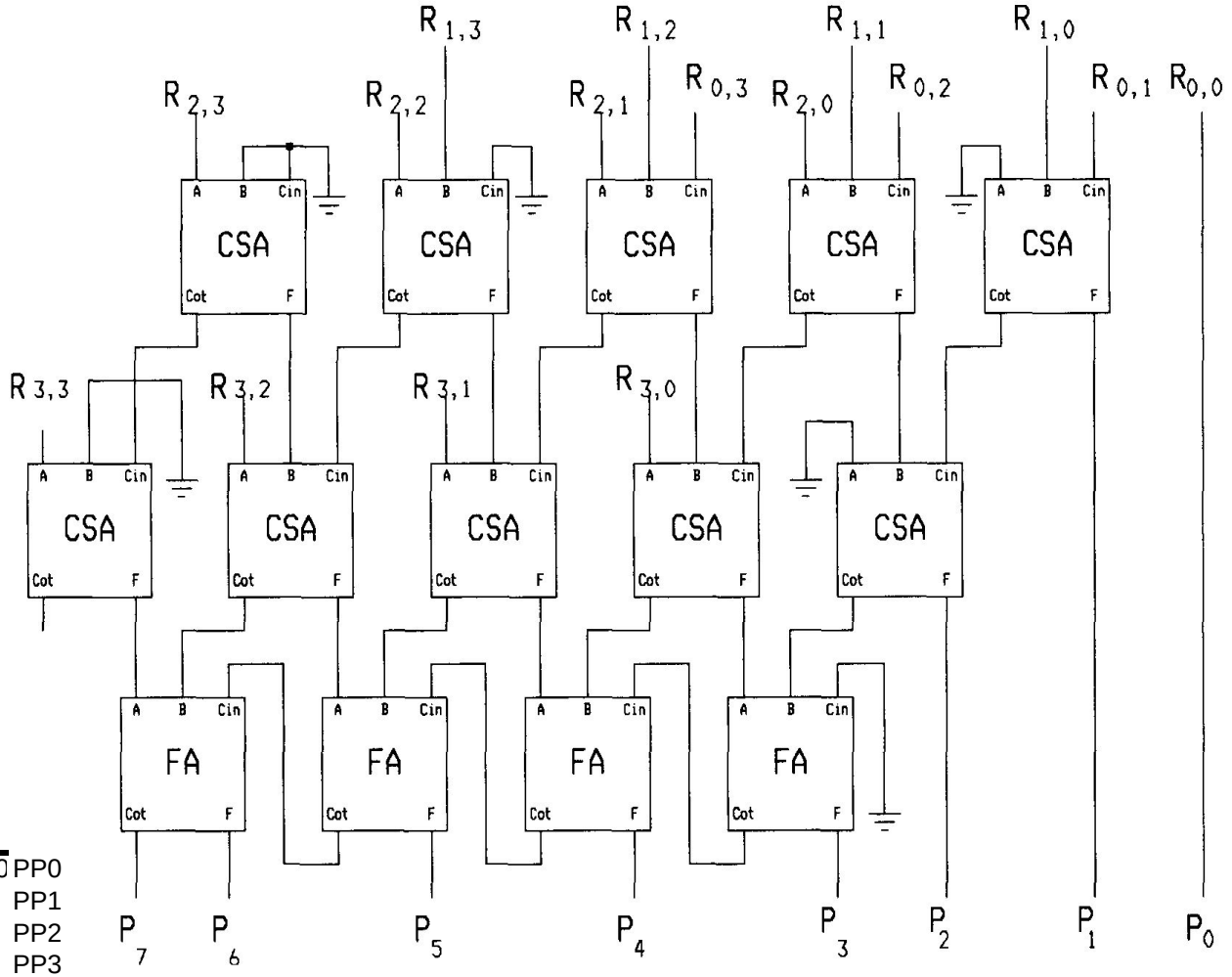
d.) CSA: Carry Save Adder

•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	

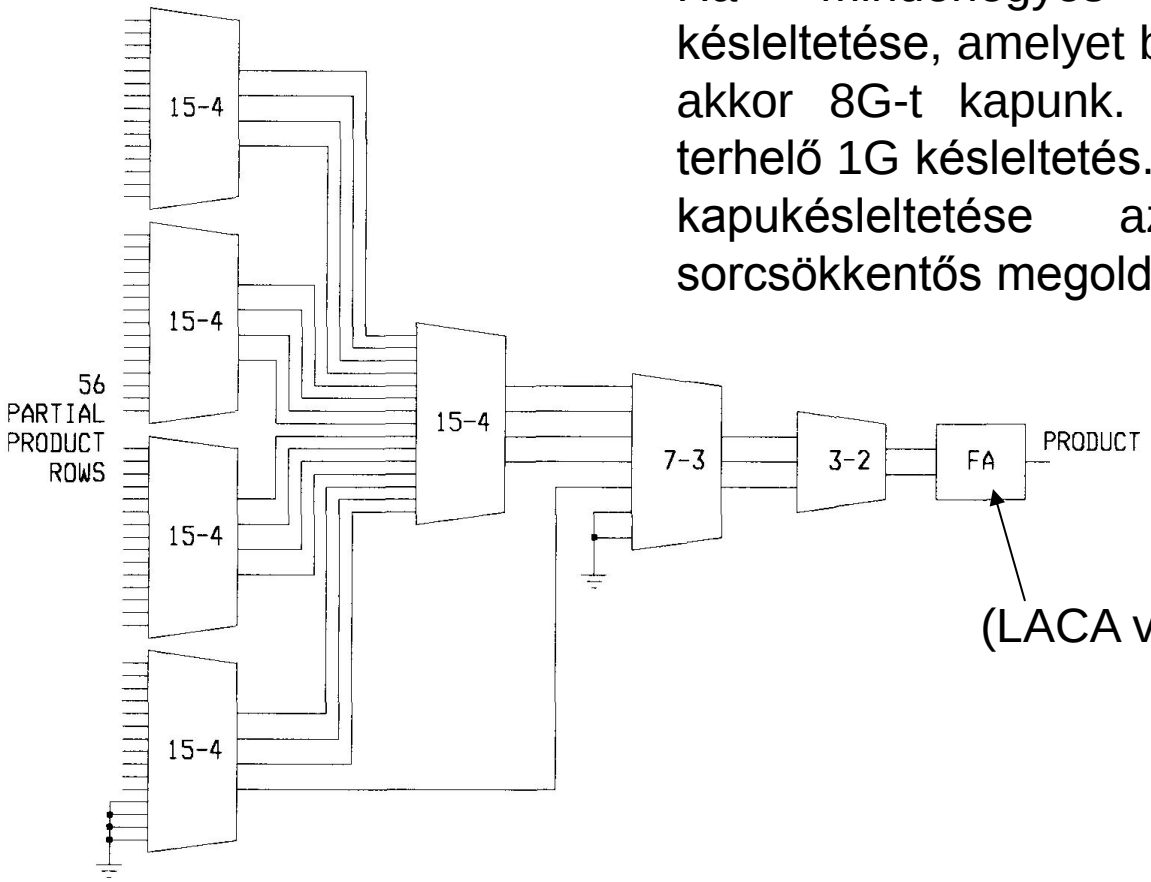


e.) Sorcsökkentős megvalósítás (row reduction)

- 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:

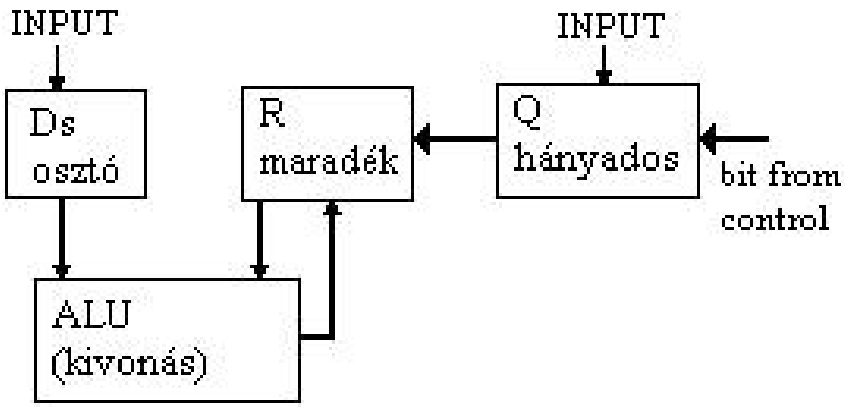
- I.) Hagyományos közvetlen osztási algoritmus
- II.) Iteratív osztási algoritmusok

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:

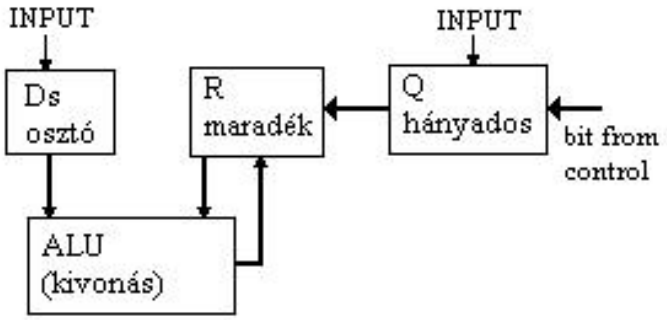
5|8 : 5=1|1
0 8
3

Bináris számok esetén hasonlóan

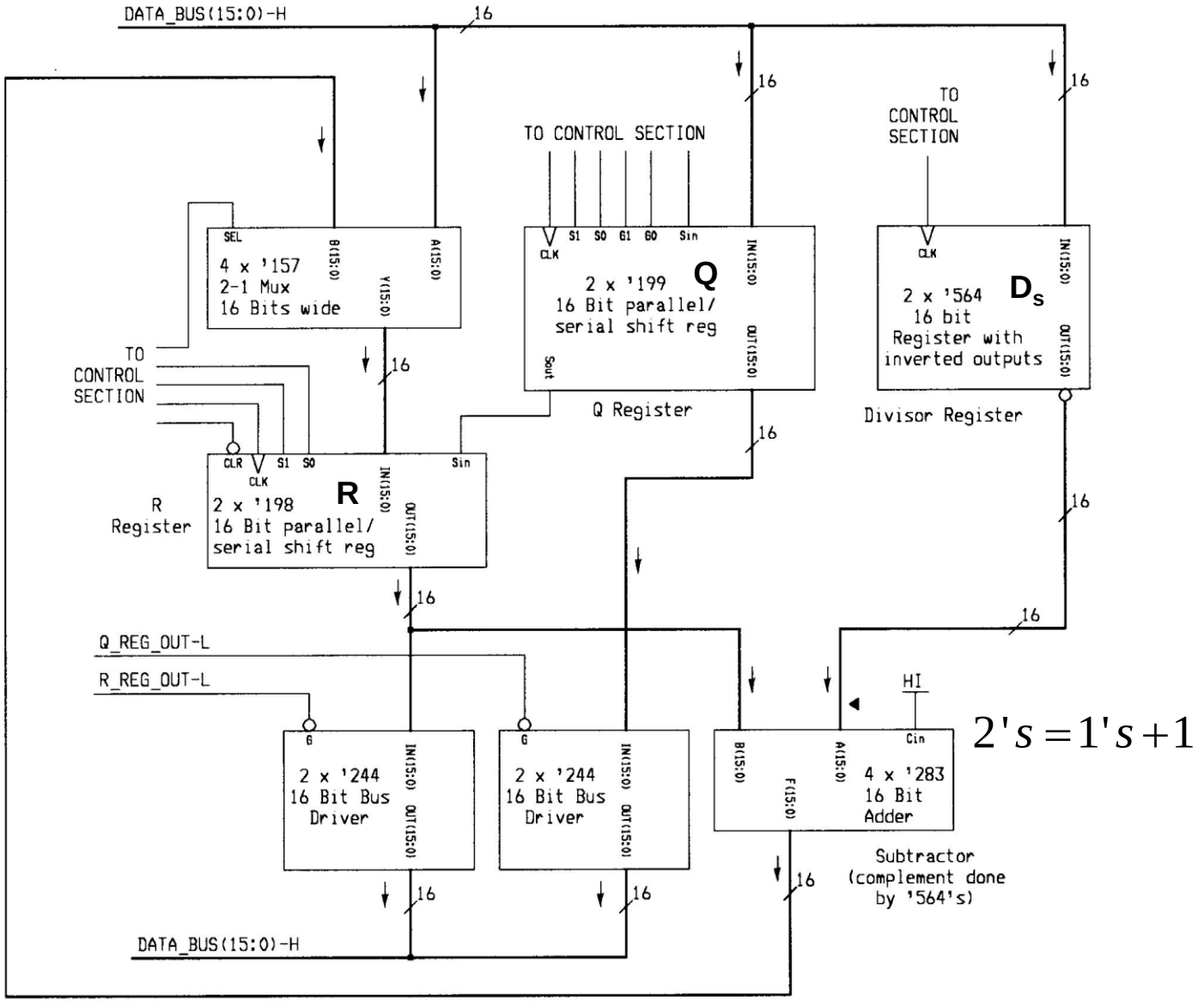
1011 Q hányados (Ds hányszor van meg Dd-ben)

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!

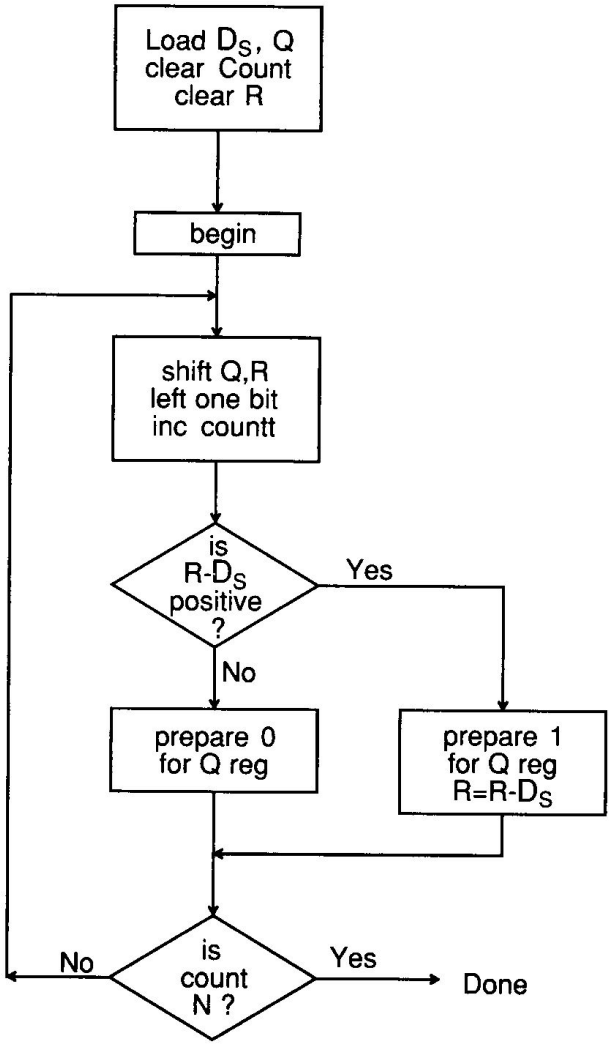


Hagyományos osztó áramkör



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

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

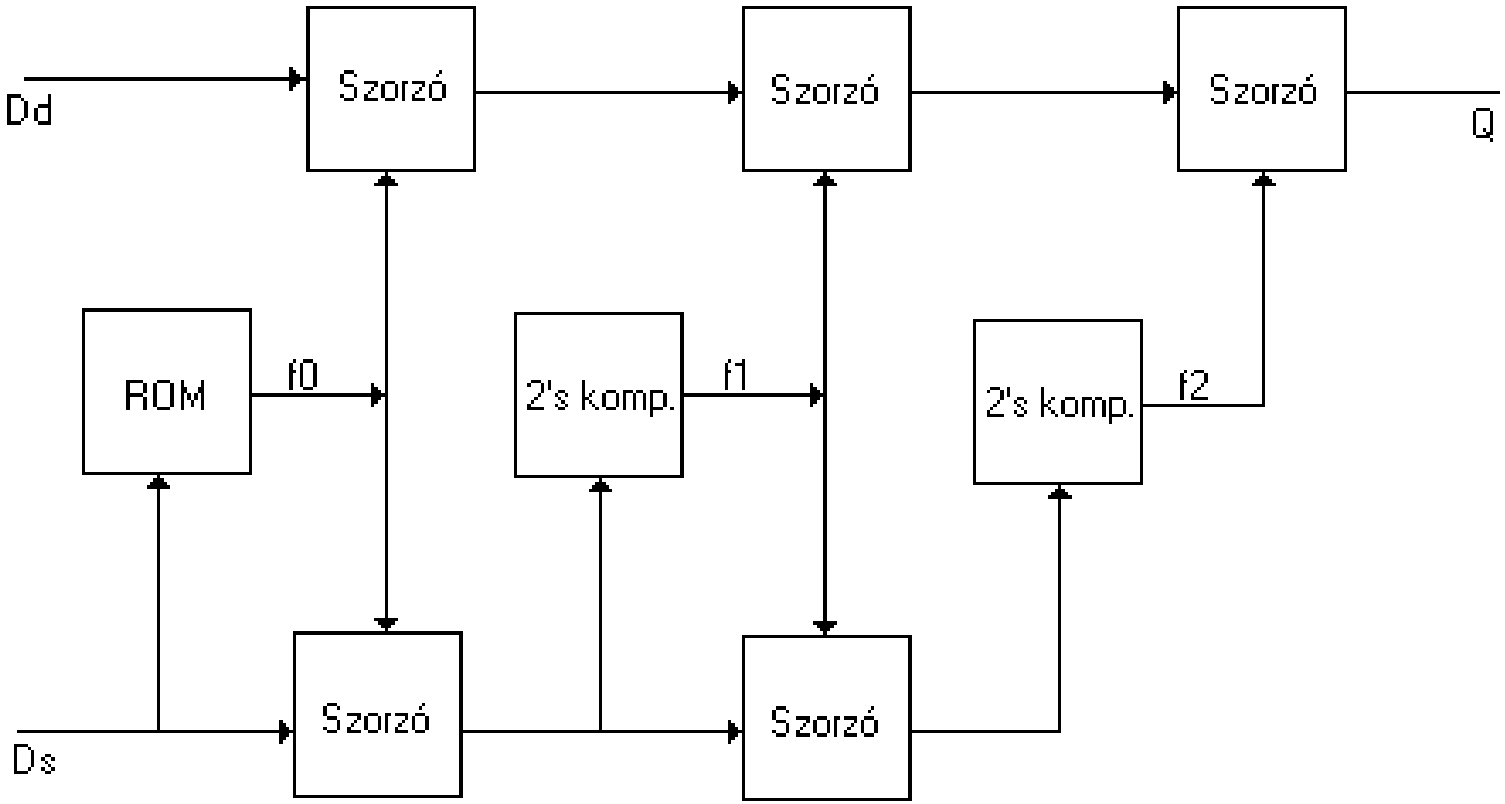
- Az iteratív osztási művelet másik módszere a következő: $Q=D_D/D_S$ kiszámolható a következő egyenlettel, ha a successive (egymást követő) $\mathbf{f_k}$ –k úgy vannak megválasztva, hogy a nevező az 1-hez konvergáljon.

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

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. $D_{D6}=0{,}6666464$ $D_{S6}=0{,}9999696$ ....

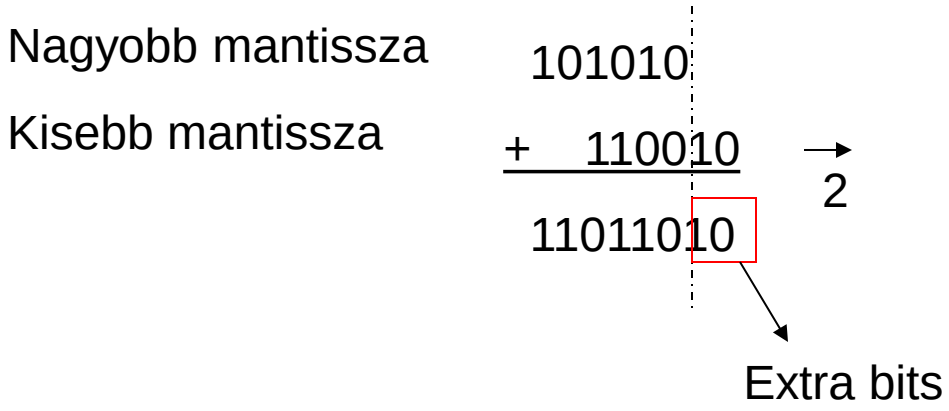
Látható, hogy itt nem kapjuk meg a kivánt értéket ($D_{D6}=0{,}6666464$) 6 iterációs lépés alatt. Ezért hogy elérjük a kivánt pontosságot véges számú lépés alatt, ROM-ot kell használni (ahol fo kezdeti értékét tároljuk).

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.

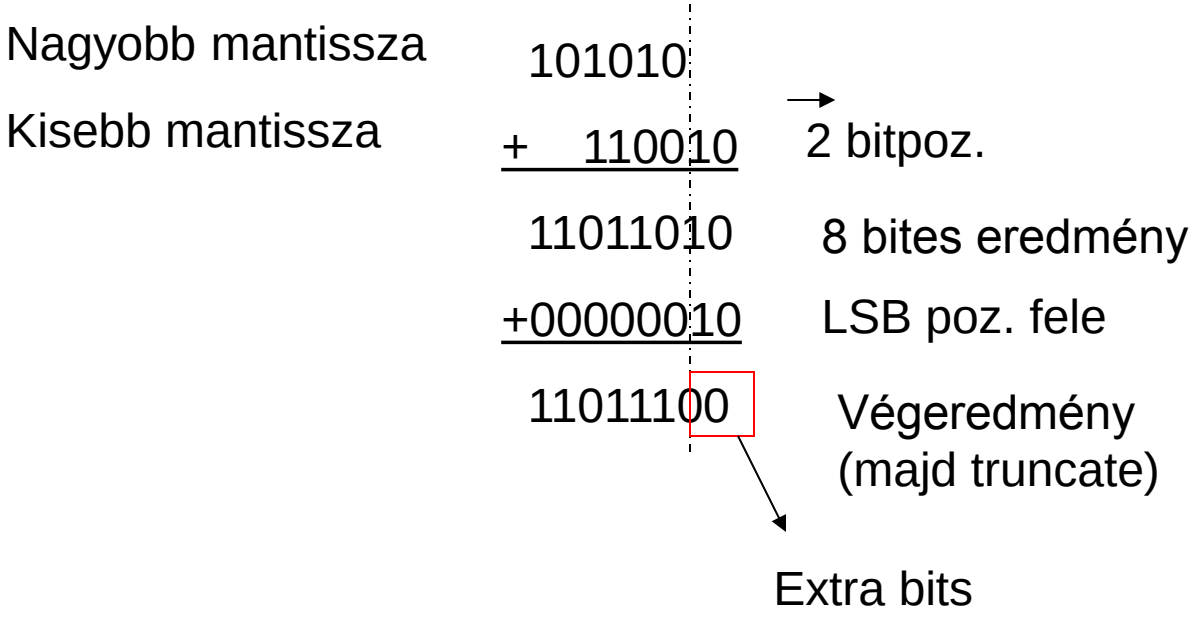
Truncation: példa

- 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}).

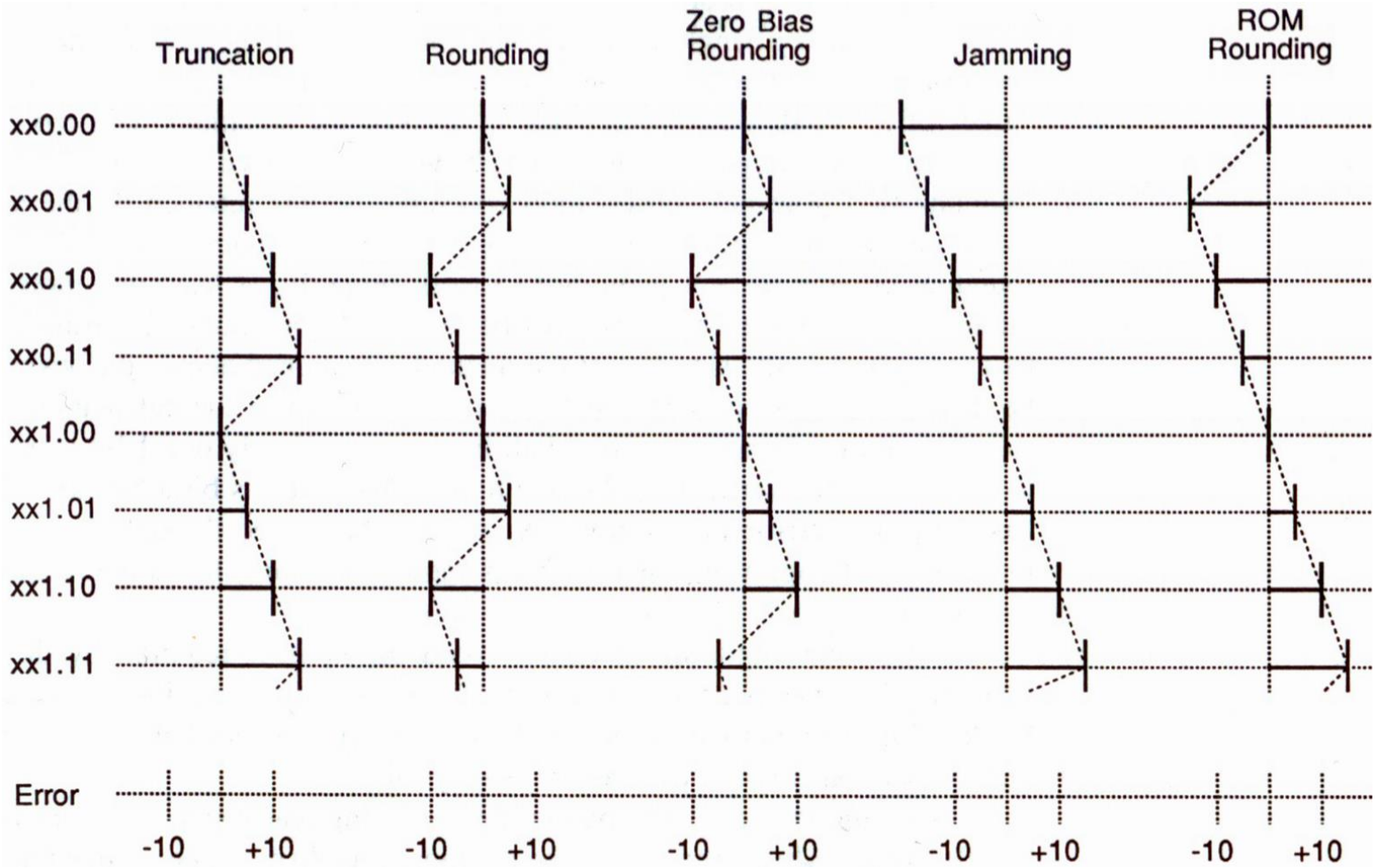
d.) Jamming (~fix értéken rögzítés)

- 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 (BSc)

5. előadás: Szekvenciális hálózatok I.
Szinkron és aszinkron tárolók, regiszterek

Előadó: Vörösházi Zsolt
voroshazi@vision.vein.hu

Jegyzetek, segédanyagok:

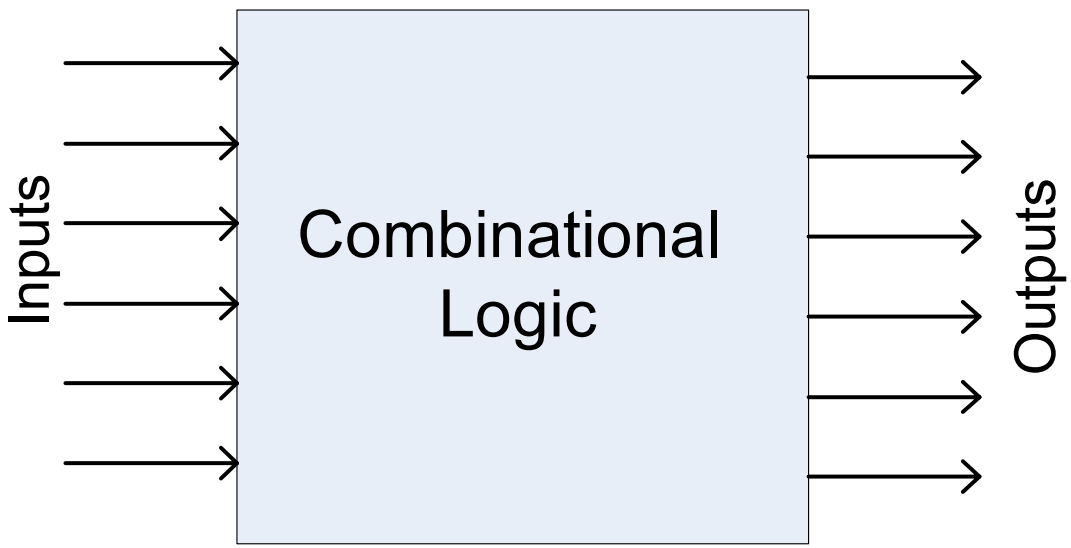
- Könyvfejezetek:
  - http://www.knt.vein.hu
    ⇒ Oktatás ⇒ Tantárgyak ⇒ Digitális Rendszerek (BSc).
    (04_chapter.pdf + további részek amik a könyvben nem szerepelnek!)
- Fóliák, óravázlatok .ppt (.pdf)
- Feltöltésük folyamatosan

Digitális-logikai hálózatok csoportosítása:

- 1.) Kombinációs Hálózatok (K.H.)
- **2.) Sorrendi Hálózatok (S.H.)**
  [könyv 4. és 12. fejezete]

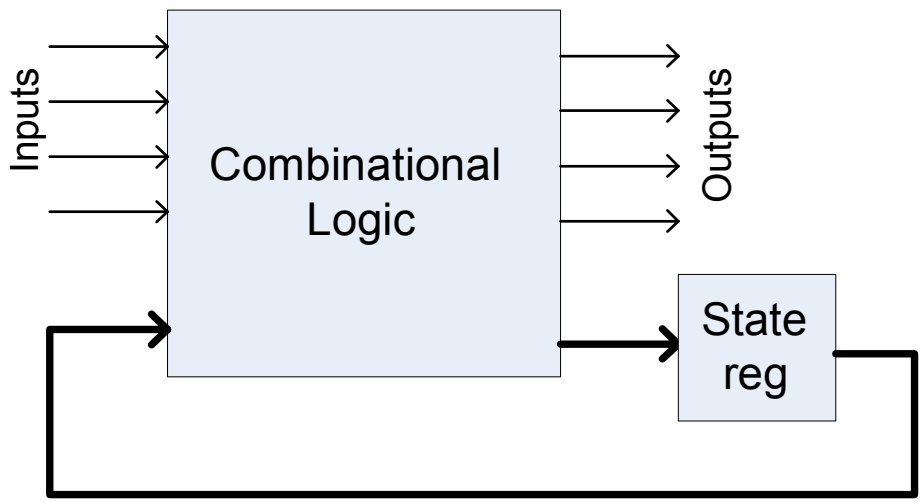
Ism: Kombinációs hálózatok

- **(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).



Ism: 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)



Szekvenciális (sorrendi) hálózatok (S.H.)

Szekvenciális hálózatok

- Hazárdjelenségek
- Visszacsatolás szerepe
- Építő elemek:
  - Szinkron tárolók és flip-flopok (szint- vagy él-vezérelt)/
  - Aszinkron tárolók (latch)
  - Regiszterek (Flip-flopok összekapcsolásából)
  - Számlálók (counters)
  - Memóriák: RAM, ROM – nagy memóriatömbök
  - Időzítő-vezérlő egységek

Eddig:

- a kommunikáció (két kapu közötti információátvitel) sebességét végtelenül gyorsnak tekintettük (K.H).
  - (S.H.) Valóságban azonban a kapuknak véges kapukésleltetéssel (propagálási idő) rendelkeznek, amelyet figyelembe kell venni!
- A kimeneti értékek generálása csak az aktuális bemeneti állapottól függött. (K.H)
  - (S.H.) Azonban a korábbi állapotok értékét is figyelembe kell vennünk!

Hazárd jelenségek

- Def: **Hazárdok:** Késleltetés okozta nem-kívánt kimenetek, állapotok.
- *Hazárd* alakulhat ki, ha egy kapu kimenete a bemenetek változásához képest csak véges időn belül változik (szilícium lapkán lévő elektron- és lyuk- vezetés következtében). $T_{propagation\ delay}$ (Nem feltétlenül alakul ki, de lehetséges!)
- *Hazárdoknak* több fajtája lehetséges:
  - Funkcionális / Statikus
  - Dinamikus

Hazárdok kialakulása I.

- **a.) Jelterjedési (propagation delay) késleltetés**:
  a logikai kapu bemeneteinek és a kimeneteinek változása közötti időkülönbség miatt.
- Függ:
  - Jelalak a bemeneten (waveform)
  - Hőmérséklet
  - Kimenet terhelése (output loading – Fan-out)
  - Disszipált teljesítmény (operating power)
  - Logikai eszköz típusa (type / device family)

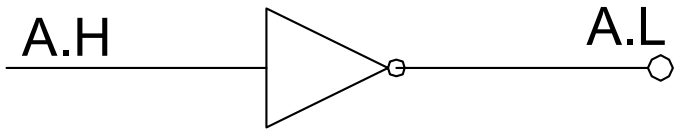
Példa: egy TTL 74LS eszközöknél, 1-gates kapu esetén a propagációs késleltetés kb. 5ns lehet.

Hazárdok kialakulása II.

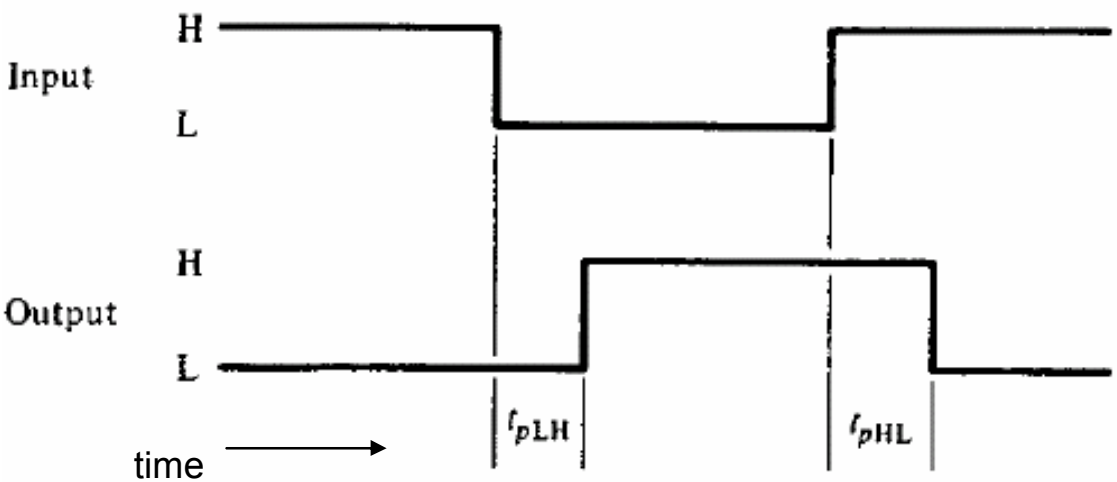
- **b.) Összeköttetési (interconnection delay) késleltetés**:
  a logikai kapukat összekötő vezetéken lévő véges jelterjedés miatt.
  - Pl: ~20 cm/ns sebességű jelátvitel az elektromos vezetéken
  - bizonyos vezetékhosszúság felett léphet fel

Példa: hazárd jelenségre

- Input: A.H → A.L (fesz. polaritását változtatjuk)



- **Idődiagram analízis**: a bemenet változását a kimenet csak véges idő alatt követi ($t_{p\,LH}$ ill. $t_{p\,HL}$)

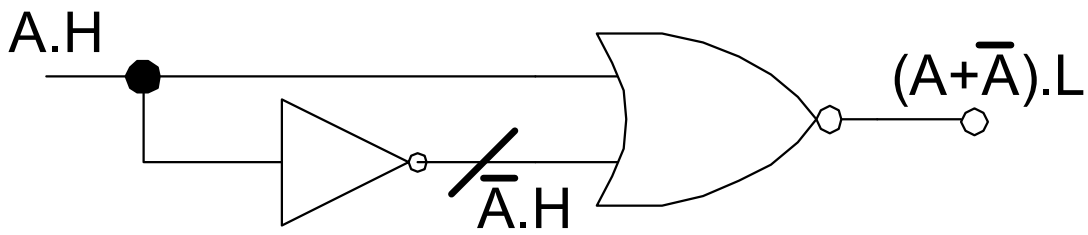


Propagációs (jelterjedési) késleltetések!

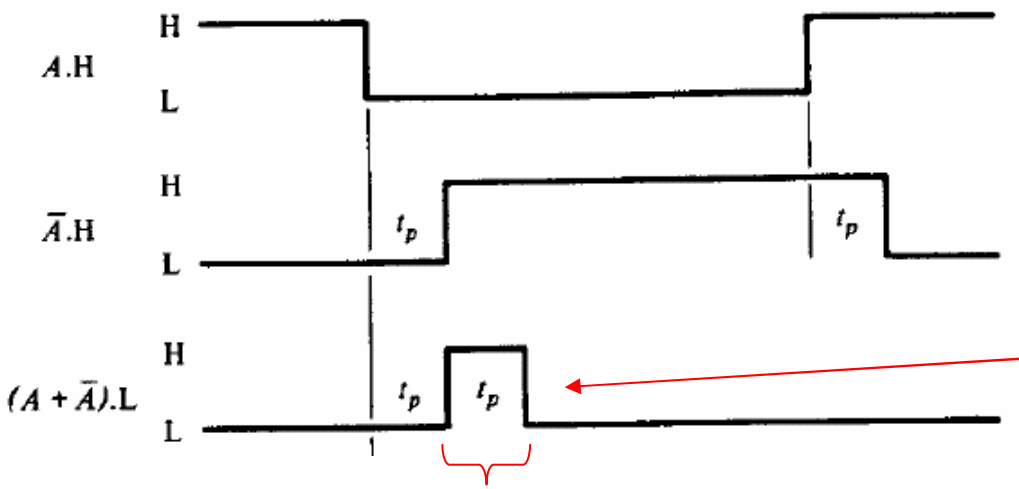
Waveform-ok

Példa: Késleltetés szerepe áramkörnél

- Tekintsük a $A + \overline{A}$ -t realizáló áramkört:



- Legyen t_p a jelterjedési késleltetés.
- Ha „A” változik T→F, akkor egy nem-kívánt („spurious”) kimenet lesz, ami egységnyi kapu-késleltetésig tart („H”)

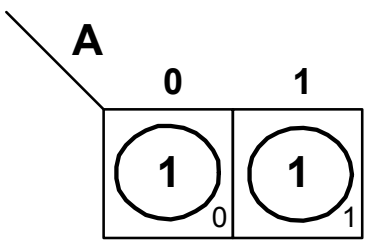


Tudjuk, hogy „A” bármely logikai értékére, a kimenet „T”. és azt is hogy „L” lesz a kimeneti fesz. értéke.

Hazárd (glitch = impulzus hiba)

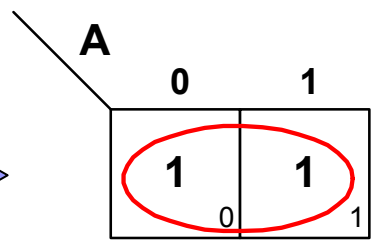
a.) Statikus / Funkcionális hazárd

- Általában elegendő időt várakozva a kimenet a megfelelő (becsült) logikai- és feszültség- értékre áll be (lásd előző példa).
- De vannak olyan hazárd-jelenségek is, melyek idővel nem szűnnek meg, ekkor a tervezőnek kell beavatkozni (**funkcionális hazárd**).
  - Pl. ha szomszédos 1-esek vannak Karnough táblában, amelyek nincsenek egy tömbbe összevonva, akkor hazárd kialakulása lehetséges:



$A + \overline{A}$

Hazárd!

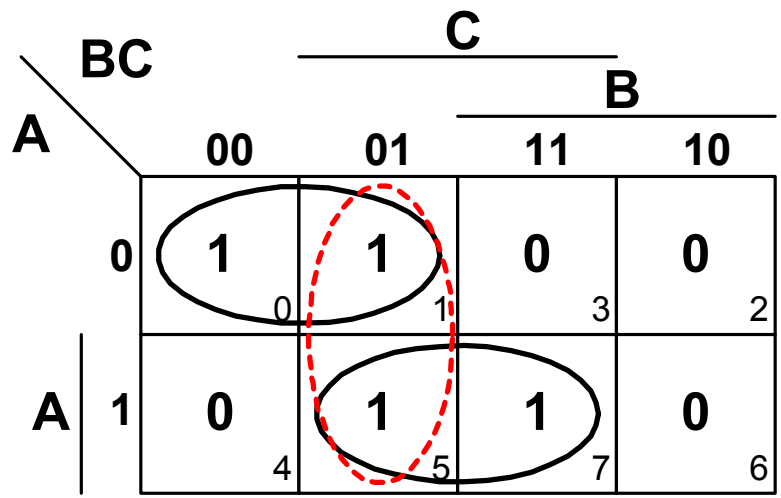


Hazárd-mentességet biztosítottunk!

$A + \overline{A} = T = 1$

Példa: Statikus hazárd

- Vegyünk egy komplexebb, 3-változós esetet:



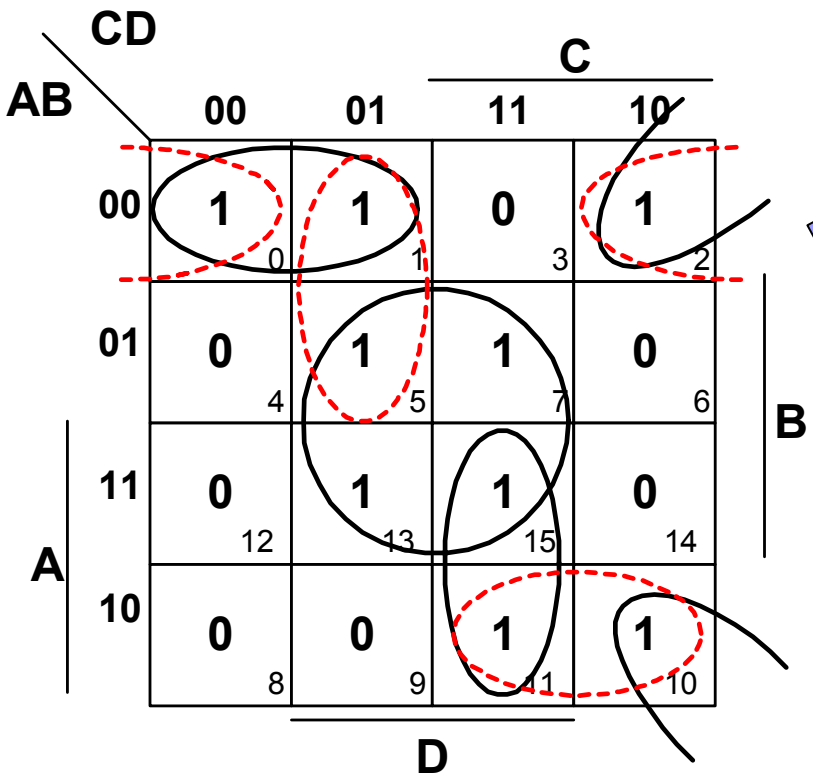
$$\overline{A}\cdot\overline{B} + A\cdot C + \overline{B}\cdot C$$

hazárdmentesítés

- Ha a szomszédos, itt kételemű tömbök között a „szaggatottal” jelölt összevonást is képezzük, akkor biztosíthatjuk a hazárdmentességet (de extra hardver szükséglet: 1 AND ill. 1 OR kapu – ezért költségesebb is.)

Példa: hazárdmentesítésre

- Legyen $F = \sum_{i=0}^{15}(0,1,2,5,7,10,11,13,15)$ //DNF!!
- Ekkor a következő K-tábla írható fel:



$$F(A,B,C,D) = \overline{A}\cdot\overline{B}\cdot\overline{C} + B\cdot D +$$
$$+A\cdot C\cdot D + \overline{B}\cdot C\cdot\overline{D} +$$
$$+\overline{A}\cdot\overline{C}\cdot D + A\cdot\overline{B}\cdot C + \overline{A}\cdot\overline{B}\cdot\overline{D}$$

Hazárdmentesítés miatt kellenek (extra kapuk)

Ebből már felrajzolható a hazárdmentesített áramkör!

b.) Dinamikus hazárd

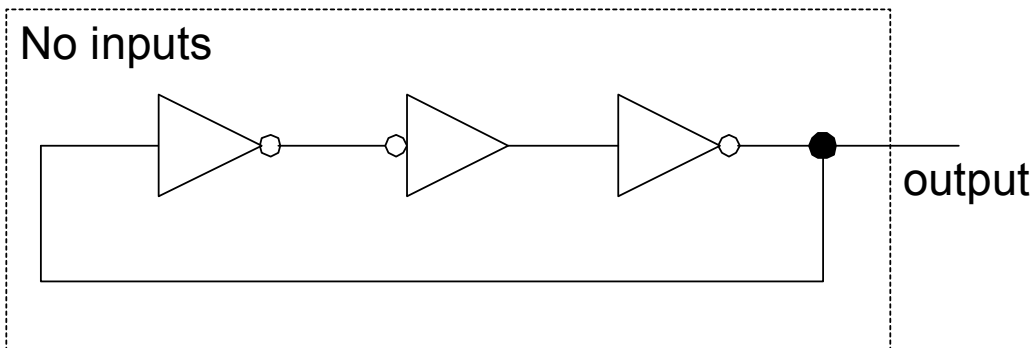
- Olyan többszintű hálózatokban jöhet létre, ahol a statikus hazárd az alacsonyabb hierarchia szinteken nem lett kiküszöbölve.
- Megszüntethető: *szinkronizálással* (órajel fel-,vagy lefutó élére működtetjük a hálózatot)

Oszcillácó - visszacsatolt áramkörökben

- K.H. + **visszacsatolás**/”**feedback**” (iteratív, szekvenciális működés): nem csak a külső bemenetek, hanem a kimenet eredeti (korábbi) értékét is figyelembe veszi az aktuális kimenet meghatározásánál. „Szokatlan” működést biztosít.
- **Példa: Ring oszcillátor:**

- Mindig páratlan számú invertert tartalmaz!

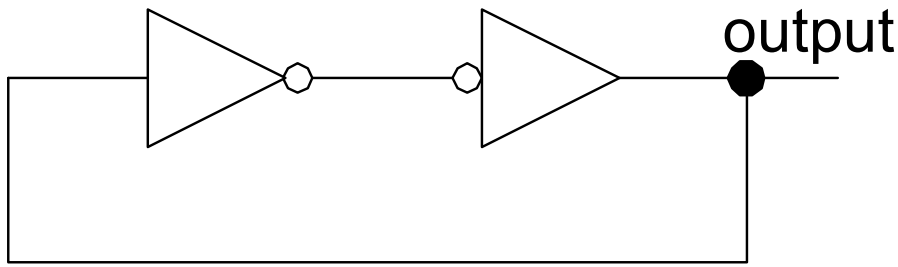
- Kimeneti feszültség/logikai-érték vissza van csatolva a bemenetre, amelynek értéke a kimeneten, mindig invertált formában jelenik meg! (***oszcillál*** a hálózat!)



Mixed-logikai kapcsolás

Stabilitás - visszacsatolt hálózatoknál

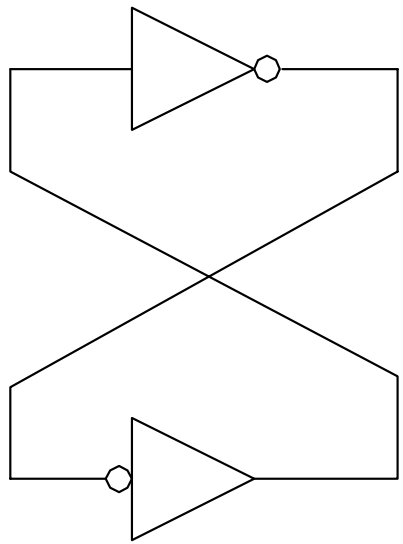
- Eltávolítva a ring oszcillátorból egy invertert a következőt kapjuk (páros számú elemmel):



- Amíg egyik inverter alacsony, addig a másik magas feszültségszintre áll be (instabilitási periódus alatt: „*settle time*”). De miután beálltak, **stabil** értéket kapunk a kimeneten (függően attól, hogy mi volt a v.cs. output értéke!)

Stabilitás:

- Előző stabilitást biztosító áramkör átrajzolásából kapjuk:
  - Mivel az áramkör nem rendelkezik külső bemenettel, ezért a viselkedésének leírására pusztán a K.H-nál megismert Boole-algebra nem elegendő.



  - „*Emlékező*” áramkör = tároló / memória (de nyilván külső bemenetek nélkül használata értelmetlen lenne)
  - Tárolók / flip-flopok esetén alkalmazzuk ezt a jelölést (más logikai kapukkal is helyettesítve az invertert).

Szekvenciális hálózatok

- Van visszacsatolás
- Korábbi állapot, és az aktuális külső bemenetek függvényében $\Rightarrow$ határozzuk meg a kimeneteket.
- Digitális rendszert – kontrollálható memóriával szekvenciális rendszernek nevezzük.
- Építőelemei:
  - Latch (retesz)
  - Flip-flop
  - Regiszter, számláló
  - Memória

Szekvenciális hálózatok csoportosítása – memória elemekre:

- 1.) órajel nélküli, **aszinkron** hálózatok:
  - a.) Latch (retesz)
  - b.) Aszinkron RS-tároló
- 2.) órajellel vezérelt / időzített **szinkron** hálózatok:
  - a.) Szinkron RS tároló – szintvezérelt
  - b.) MS Flip-Flop (tároló)
  - c.) Tiszta (pure) él-vezérelt FF
  - d.) JK Flip-Flop (tároló)
  - e.) D Flip-Flop (tároló)
  - f.) T Flip-Flop (tároló)

Él-vezérelt

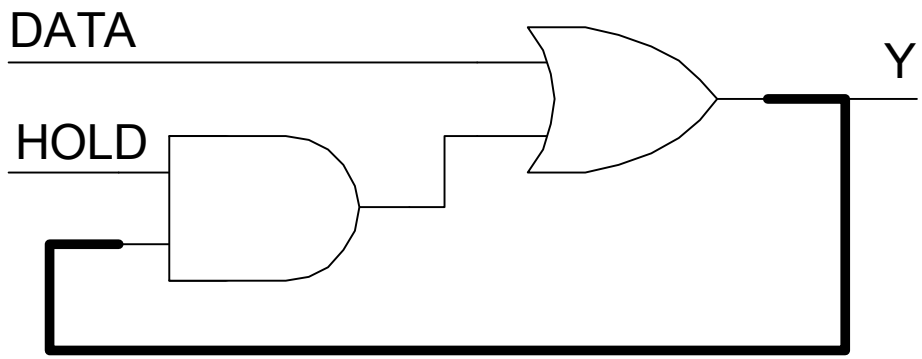
Fontos megjegyzések:

- **Aszinkron** esetben a „**tároló”** kifejezést használjuk (órajel nélküli)
- **Szinkron** esetben (órajellel vezérelt).
  - **Szint-vezérelt (level-triggered)** eszközök: adott logikai / feszültség szintet jelent – „**tároló”** kifejezést használjuk
  - **Él-vezérelt (edge-triggered)** eszközök: le / felfutó élre – „**flip-flop** kifejezést használjuk. (ekkor fontos, hogy csak egy meghatározott időpontban vegyünk mintát!)

1.) Órajel nélküli, aszinkron sorrendi hálózatok

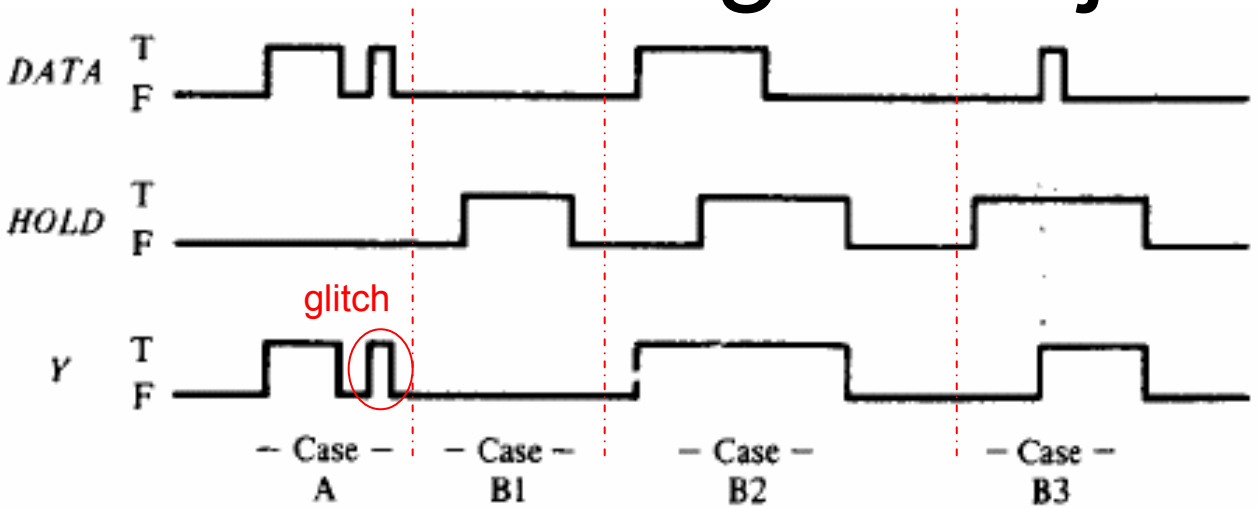
1/a.) Latch

- **Latch (retesz):** legegyszerűbb tároló elem. Az idő mint fontos paraméter játszik szerepet működésében.
- Tulajdonsága: a bemenetén lévő log. igaz (‘T’) adat azonnali kimenetre helyezése.



- Hibája, hogy amíg HOLD=T, ha egy impulzus zaj érkezik, akkor egy Y=T jelenik meg (glitch v. impulzus hiba).
- További tul. hogy a pillanatnyi ‘T’ érték a kimeneten Y=T megjelenik mindaddig, amíg HOLD=F nem lesz. (Ezt hívják **1’s catching**). Néha hasznos, de veszélyes is lehet pl. leragadásos hiba!

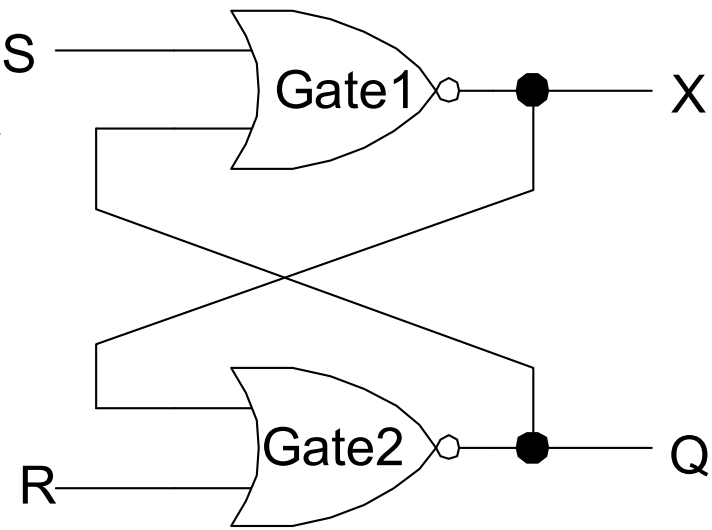
Latch időzítési diagrammja:



- Több eset – aleset lehetséges:
  - **Case A:** Ha HOLD=F, akkor Y=DATA
  - **Case B:** Ha HOLD=T, akkor bármely előfordulásakor a DATA=T esetén az adatot tárolja („hold”=tartja), mindaddig amíg HOLD=F nem lesz. Ekkor a kimenetére helyezi. Három aleset lehetséges:
    - **Case B1:** DATA=F, amikor HOLD=T. Ekkor Y=F
    - **Case B2:** DATA=T és HOLD=F. Amikor HOLD=T lesz, tárolja az adatot, addíg ameddig HOLD=F nem lesz újra. Ekkor a kimenetre helyezi
    - **Case B3:** DATA=F, amikor HOLD=F. Majd az adat DATA=T lesz, és ezáltal Y=T (DATA) lesz. Mindaddig kitartja Y-t, amíg HOLD=F.

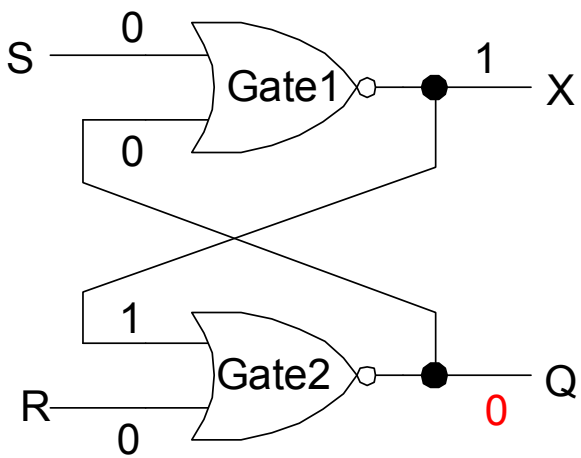
1/b.) Aszinkron RS-tároló (latch)

- Visszacsatolt hálózatok a memóriák egy sajátos típusát képezik: „visszaemlékezik” a feszültség, v. logikai szintek állapotára (1-bites tároló)
- Két stabil állapota lehetséges (**bistabil** eszköz), amelyeket eddig azonban külsőleg nem tudtunk befolyásolni.
- Ezért kell más logikai kapukat alkalmazni az inverterek helyett a visszacsatolásnál!
  így kapunk **aszinkron RS-tárolót**.
- Például: itt **NOR** kapukat használva
  - S: Set (beállítás)
  - R: Reset (újrabeállítás / törlés)
  - Kimenetek: Q (állapot),
    X (Q negáltja).
- (Megj: X-et jelölik $\overline{Q}$-al is!)

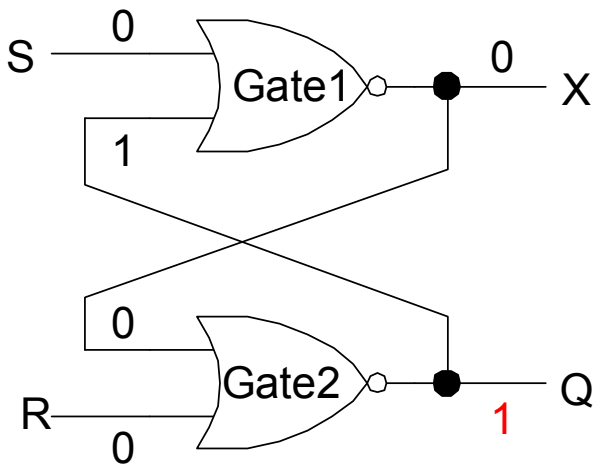


Aszinkron RS-tároló - két stabil (**bi-stabil**) állapota:

- **i.) Logikai NOR kapcsolással**



I.) RS tároló Q=‘0’-ás állapotban



II.) RS tároló Q=‘1’-es állapotban

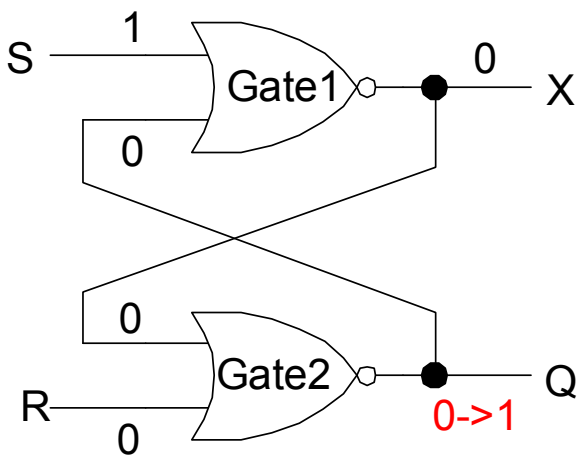
A	B	NOR
0	0	1
0	1	0
1	0	0
1	1	0

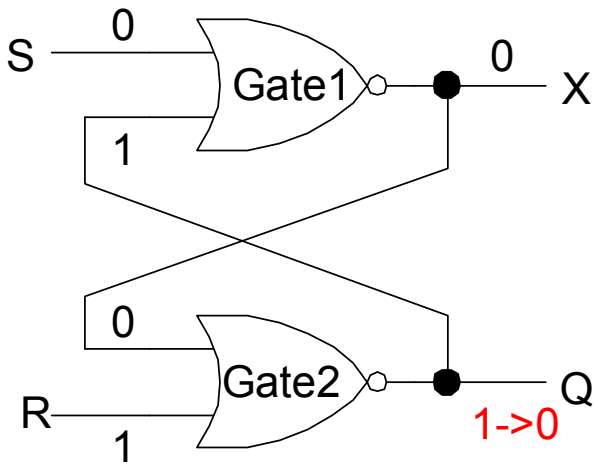
NOR igazságtáblázat

S=R=‘0’=‘F’ logikai szinten rögzítve tárolás során!

Aszinkron RS-tároló – „**Set/Reset**”:

- **i.) Logikai NOR kapcsolással (folyt)**





A	B	NOR
0	0	1
0	1	0
1	0	0
1	1	0

NOR igazságtáblázat

III.) Tfh: **S=1**, és Q=0.

Ekkor X=0

Ezután Q=1 lesz.

Tehát Q **0->1** (**Set**) történt!

IV.) Tfh: **R=1**, és Q=1.

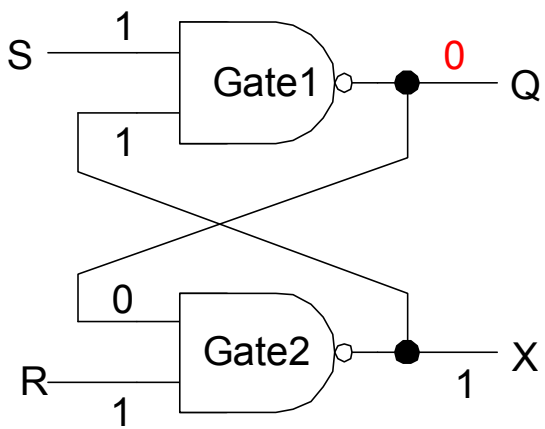
Ekkor X=1

Ezután Q=0 lesz.

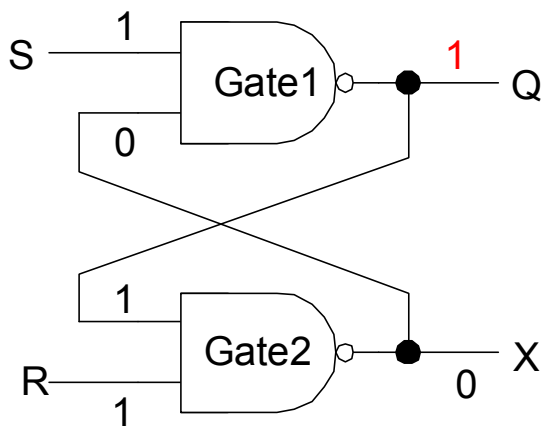
Tehát Q **1-> 0** (**Reset**) történt!

Aszinkron RS-tároló - két stabil (bi-stabil) állapota:

- **ii.) Logikai NAND kapcsolással**



I.) RS tároló Q='0'-ás állapotban



II.) RS tároló Q='1'-es állapotban

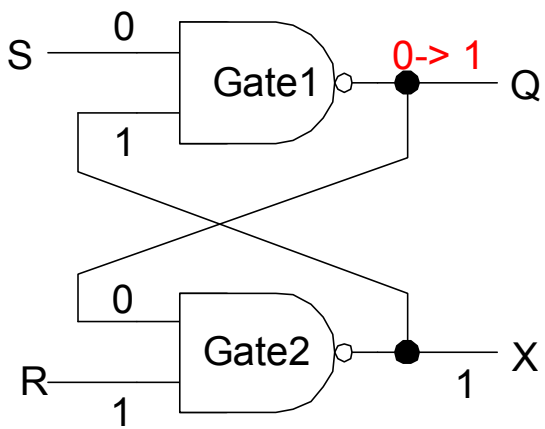
A	B	NAND
0	0	1
0	1	1
1	0	1
1	1	0

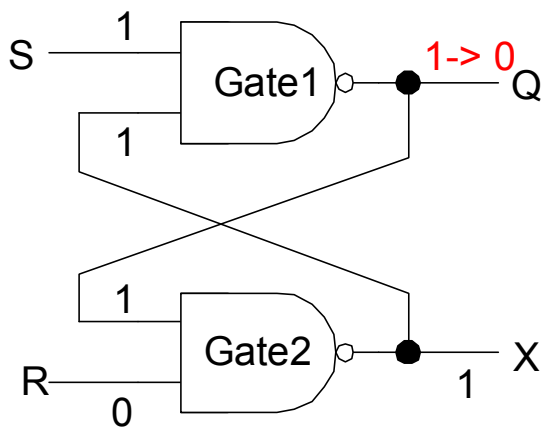
NAND igazságtáblázat

S=R='1'='T' logikai szinten rögzítve tárolás során! (Duálisa a NOR-nak!)

Aszinkron RS-tároló – „**Set/Reset**”:

- **ii.) Logikai NAND kapcsolással (folyt.)**





A	B	NAND
0	0	1
0	1	1
1	0	1
1	1	0

NAND igazságtáblázat

III.) Tfh: **S=0**, és Q=0.

Ekkor X=0

Ezután Q=1 lesz.

Tehát Q **0->1** (**Set**) történt!

IV.) Tfh: **R=0**, és Q=1.

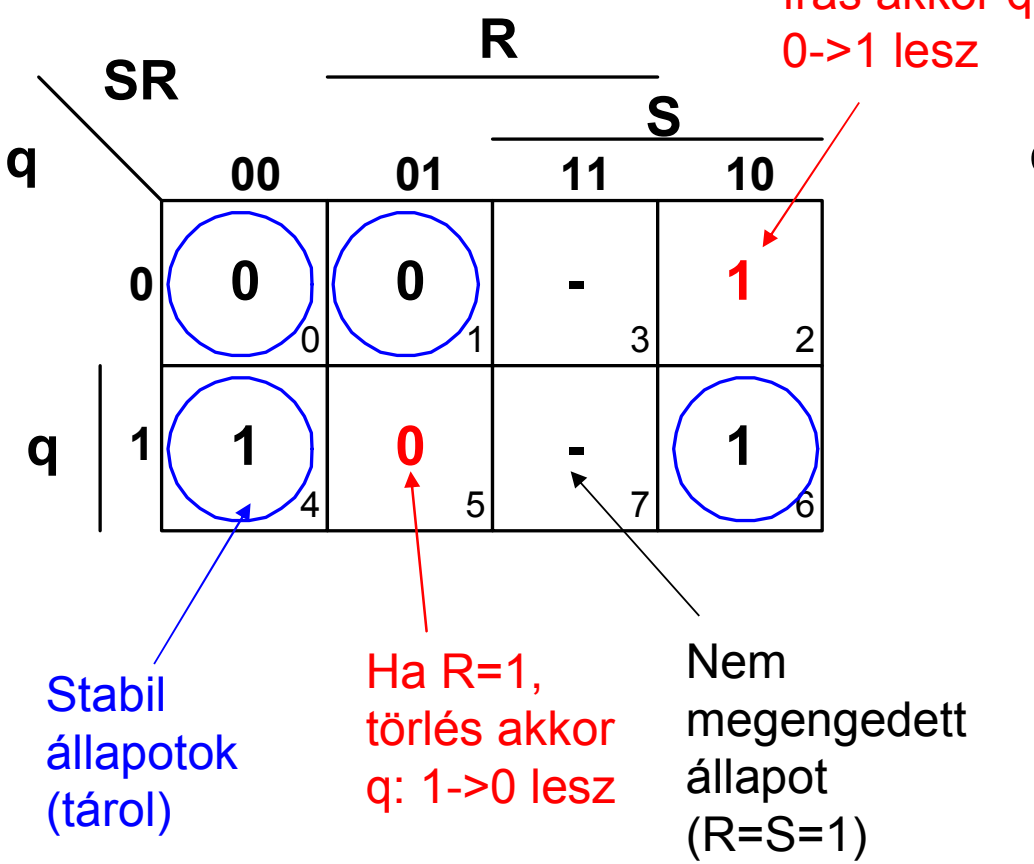
Ekkor X=1

Ezután Q=0 lesz.

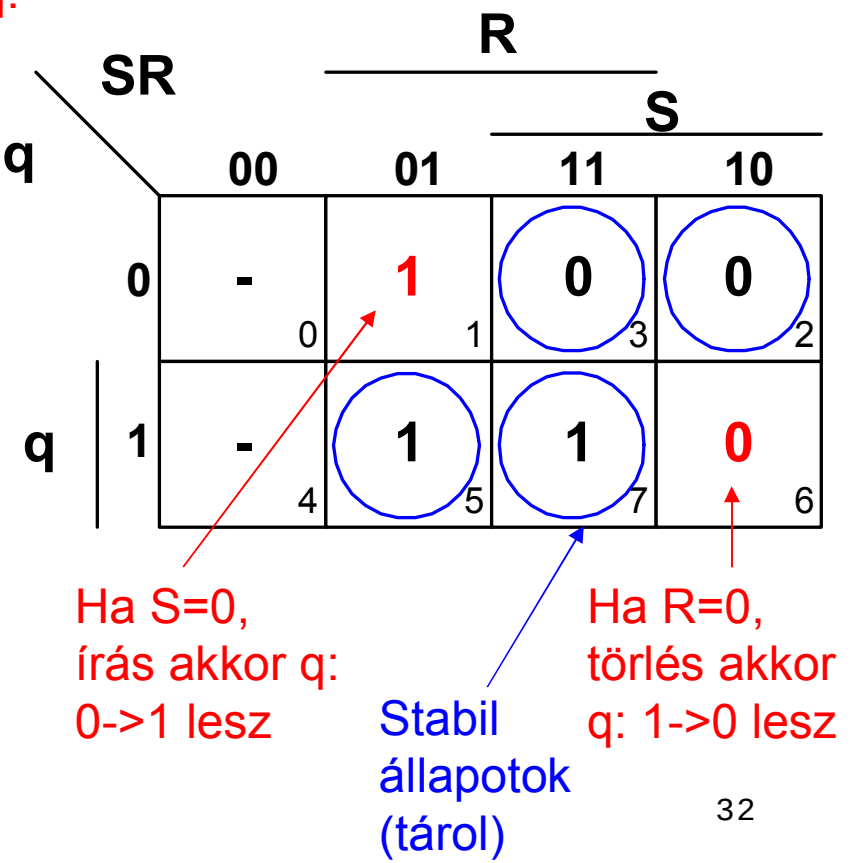
Tehát Q **1-> 0** (**Reset**) történt!

RS tárolók Karnough táblái:

- **NOR** esetben:

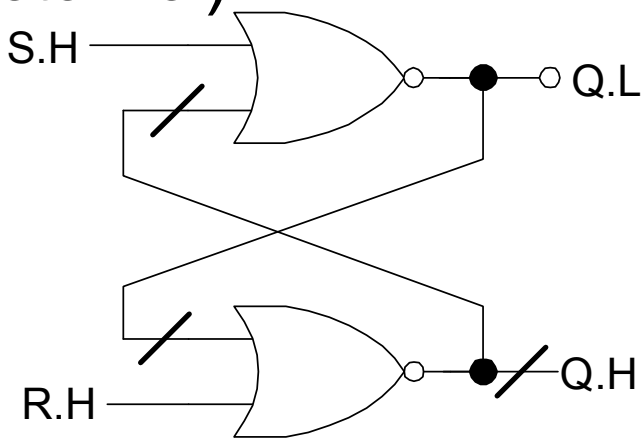


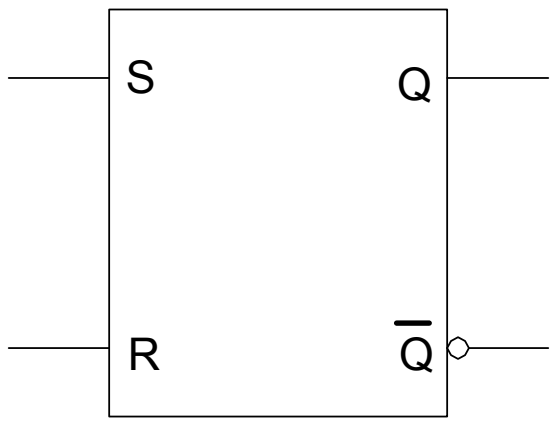
- **NAND** esetben:



Aszinkron RS-tároló feszültség-logikai viselkedése (NOR kapuval)

- Mixed-logikai kapcsolása (‘T’=‘H’ *pozitív* logika a bemeneteknél):





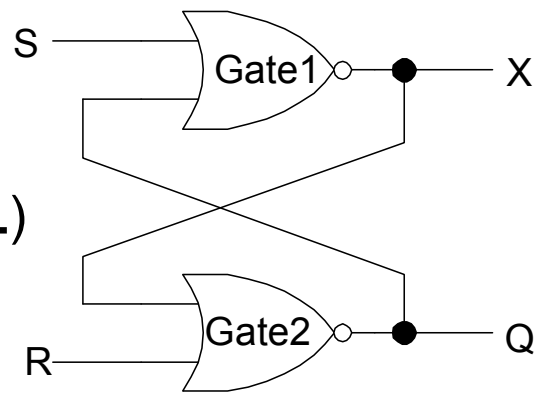
Amíg **S.H és R.H „L” („F”)** feszültségszinten van, a tároló aktuális állapotát tárolja, Q-tól függően (a bistabil állapot közül az egyikbe kerülünk).

Ha S-t, vagy R-t változtatjuk:
- S: control bemenet – Set State-be helyezi az áramkört (Q=H), **S=H (‘T’)** esetén.
- R: control bemenet – Reset State-be helyezi az á.k.-t (Q=L), amikor **R=H (‘T’).**

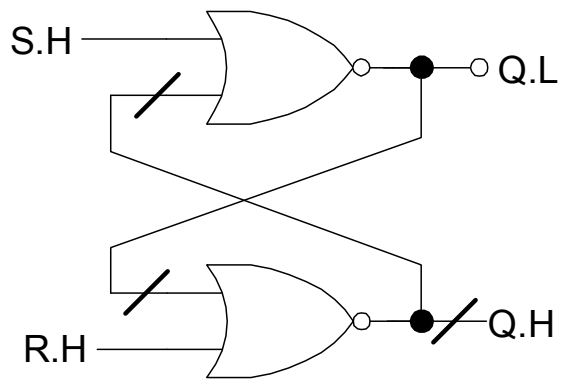
Aszinkron RS-tároló feszültség-logikai viselkedése (NOR kapuval)

- **Tekintsük a hálózatot pozitív logikával, ahol T=H (F=L)**
- **Tfh**: R.L, S.L és Q=L. Ekkor ⇒ X.H lesz. Ezt visszacsatolva Q.L lesz megint. Tehát addig nincs változás a stabil viselkedésben, ameddig az R és S állapotokon nem változtatunk! (‘L’ / ‘0’ állapotban van)
- Bi-stabilitása miatt ez igaz lesz Q.H ⇒ X.L –re is (‘H’ /‘1’ állapot).
- **Set/ Reset**:
  - Set state: (Q=L→**H**)
  - Reset state: (Q=H→**L**)

Terminológia: $\overline{Q}$-al jelöli!



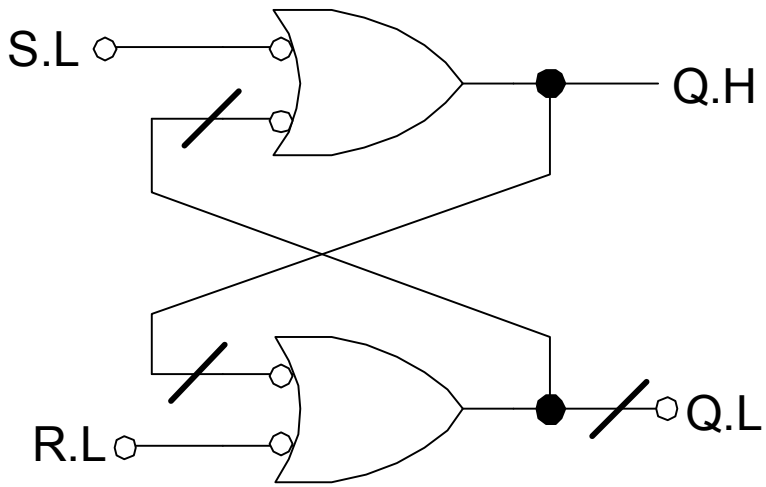
Logikai NOR kapcsolási rajza

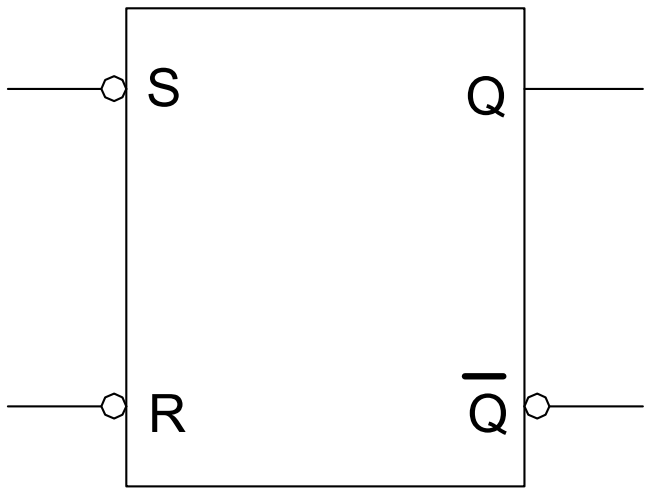


Mixed-logikai NOR kapcsolási rajza

Aszinkron RS-tároló feszültség-logikai viselkedése (NAND kapuval)

- Mixed-logikai kapcsolása (NOR duálisa) és szimbóluma ('T'='L' *negatív* logika a bemeneteknél):





Amíg **S.L és R.L „H"** feszültségszinten van, a tároló aktuális állapotát tárolja, Q-tól függően. A bistabil állapot közül az egyikbe kerülünk.

Ha S-t, vagy R-t változtatjuk:
- S: control bemenet – Set State-be helyezi az áramkört (Q=H), **S='L'** esetén.
- R: control bemenet – Reset State-be helyezi az á.k.-t (Q=L), amikor **R='L'.**

Aszinkron RS-tároló tulajdonságai

- **Aszinkronitás**: nincs közös rendszer órajel, amely működtetné. Csupán az S és R control jelek hatására válaszol a kimenet („azonnal” – véges időn belül).
- Asynchronous (unlocked) latch
- !Hibája, hogy érzékeny impulzus zajokra: *glitch*-ek lehetnek az S / R bemeneteken, amikor a másik Reset- / Set- state állapotban vagyunk.
  - Általános tervezési eszközként ezért nem ajánlatos használni.

Gerjesztési tábla:

„feszültség értékek” ábrázolására

- **Gerjesztési (Excitation) tábla**: sorrendi hálózatoknál a logikai- és feszültség- értékek felírására használatos, az *időbeliség* figyelembe vételével! (t: idő múlva, δ: beállási (settle) idő)
- A NOR kapukból felépülő RS tároló működésének leírása gerjesztési táblázat segítségével (feszültség értékekre):

S	R	$Q_{(t)}$	$X_{(t)}$	$Q_{(t+\delta)}$	$X_{(t+\delta)}$	
L	L	q	x	q	x	Hold
L	H	q	x	L	H	Reset
H	L	q	x	H	L	Set
H	H	q	x			Disallowed

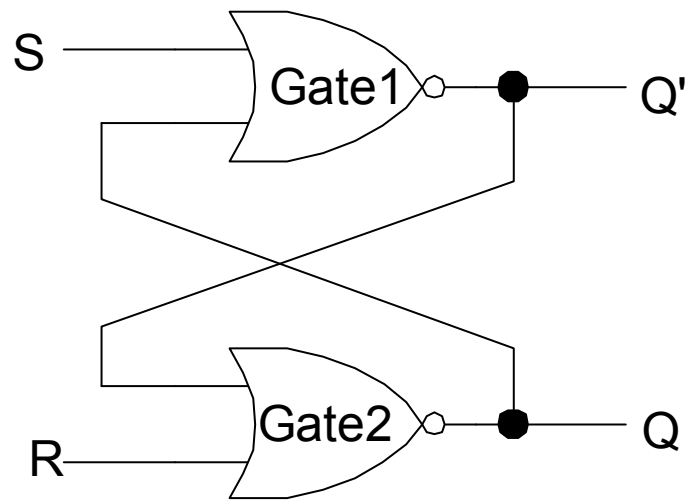
Kétértelműség! (NOR miatt)

Gerjesztési tábla:
„logikai értékek” ábrázolására

- A **NOR** kapukból felépülő RS tároló működésének leírása gerjesztési táblázat segítségével (logikai értékekre):

S	R	Q	Q'	
0	0	q	~q	Hold
0	1	0	1	Reset
1	0	1	0	Set
1	1	-	-	Disallow

Kétértelműség! (NOR miatt, ha R=S=1)

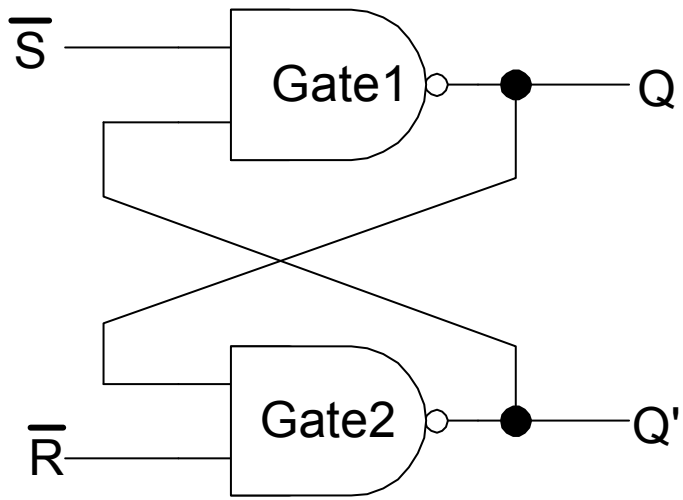


Gerjesztési tábla:
„logikai értékek” ábrázolására

- A **NAND** kapukból felépülő RS tároló működésének leírása gerjesztési táblázat segítségével (logikai értékekre):

$\overline{S}$	$\overline{R}$	Q	Q'	
0	0	-	-	Disallowed
0	1	1	0	Set
1	0	0	1	Reset
1	1	q	~q	Hold

Kétértelműség! (NAND miatt, ha R=S=0 vagyis ha R=S=1)

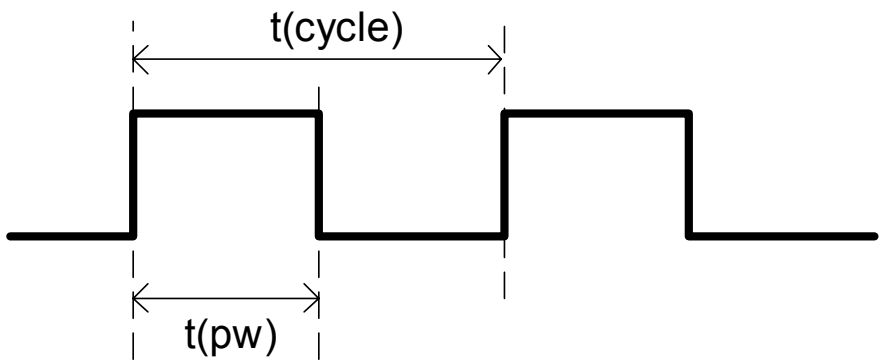


NOR duálisa: kapuk, bemenetek és állapotok felcserélésével kapjuk.

2.) Órajellel vezérelt, szinkron sorrendi hálózatok

Óra (Clock)

- 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.



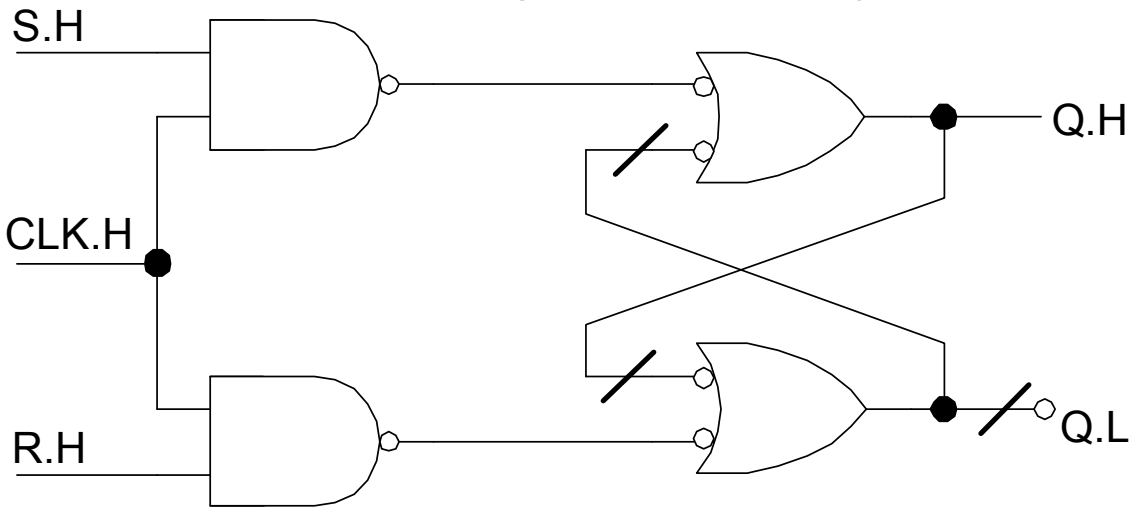
Négyszögjel.

Órajellel vezérelt szinkron sorrendi hálózatok:

- Eml: aszinkron tárolók (‘1’ / ‘T’ catching)
- Szinkron tárolók (flip-flop-ok): általánosabb építő elemek
- **Szinkronizálás**: A kimenet mindaddig nem fog változni, amíg egy rendszer órajelre (CLK) nem engedélyeződik.
- CLK: órajel impulzus ált. négyszögjel formájában adott (timing waveform)

2/ a.) Szinkron RS-tároló

- Felépítése hasonló az aszinkron RS tárolóhoz, csupán az R / S állapotok aktivitását órajellel szabályozzuk.



- A kimenet értéke csak az órajel ‘igaz’ (‘T’) állapota alatt változhat meg („**Level-driven”**: szint vezérelt eszköz)
- De az órajel (itt magas aktív) állapotát elegendően szűkre kell beállítani, különben hazárdok lehetségesek („shock”).

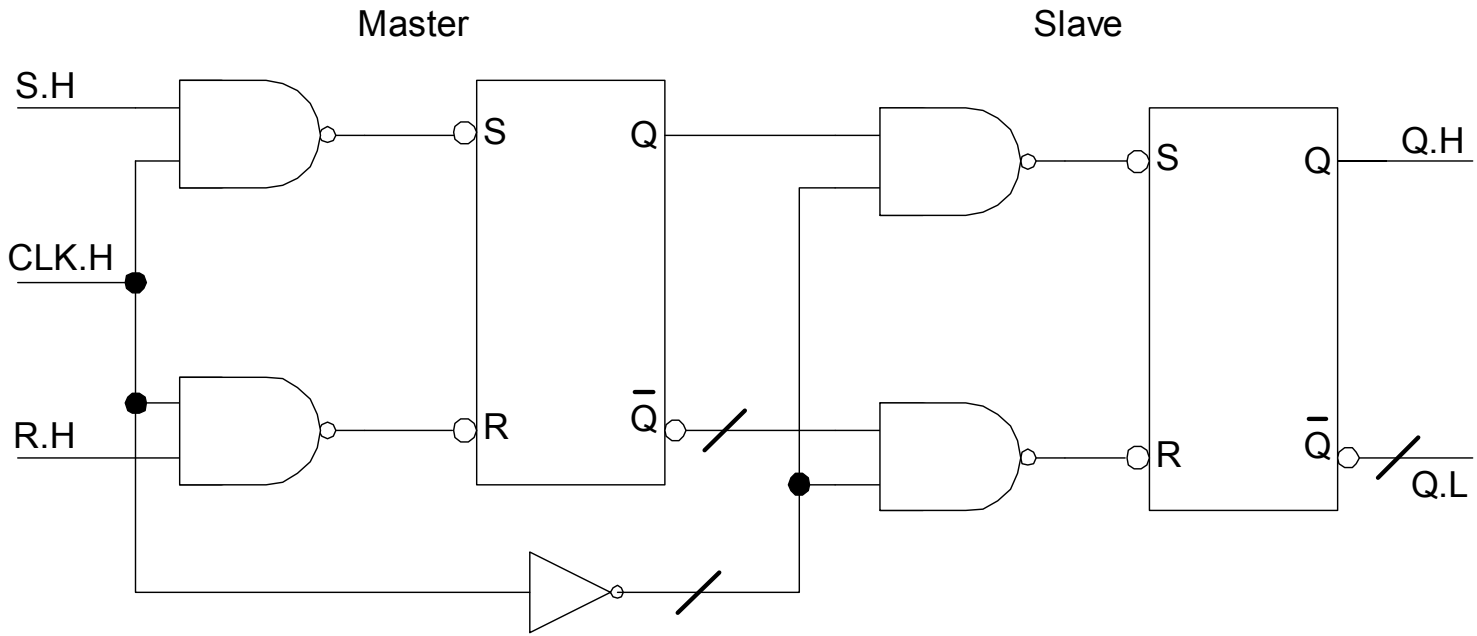
Órajel meghatározása

- *Órajel beállítása*: a célunk, hogy elegendően kis intervallumot biztosítsunk, amelynek hatására a kimenet megváltozhat, a bemenettől függően (a szint-vezérelt működés helyett, él-vezérelt működést kell biztosítani!)
- **Él-vezérelt működés**: (edge-driven / edge-triggered): órajel feszültség átmenete
  - Pozitív (felfutó) él: CLK.L→H
  - Negatív (lefutó) él: CLK.H→L 

- *Megj*: Az órajellel vezérelt szekvenciális rendszereket a továbbiakban *él-vezérelt* működésű eszközöknek tekintjük (más megnevezés: időzített flip-flopok)

2/ b.) Szinkron MS flip-flop

- Régóta széles körben használt eszköz, egyszerű felépítésű



• Ha a CLK magas aktív (‘H’), akkor a kimenet megváltozik a bemenettől függően.

• H -> L átmenetnél (lefutó él) leválasztja az **Master** tárolót a S / R bemenetekről (ekkor tárol – változatlan tartalom)

• Feszültség invertálás miatt a **Slave** tároló ellenütemben működik: negatív (lefutó) él esetén lesz aktív (S / R bemeneteit a Master állapottól kapja)

2/ c.) Tiszta (pure) él-vezérelt FF

- Célunk: elkerüljük a hazárdokat (glitch, noise-zaj stb.)
- MS-FF helyett alkalmazzuk: *1's catching* tulajdonságot próbáljuk elkerülni használatával (clk pozitív részén)
- Az órajelciklus negatív részén viszont az R / S control bemeneteket kell stabil állapotba hozni
- Tiszta él-vezérlés: állapot-átmeneteket is használni kell
  - Aktív él: F→T amelyre az átmenet megtörténik (lehet H→L vagy **L→H**)
  - Aktív élre a bemeneteket (R / S) kell figyelni (sensing)
  - Aktív él eredményeként (adott log / fesz. szinten) szabad csak megváltoznia az állapotnak (Q)

Gerjesztési tábla: logikai és feszültség értékek ábrázolására

- Él-vezérelt Flip-flopok esetén használatos táblázat
- Jelölések:
  - Minden aktív élre vagy *Set*, vagy *Reset* állapotba jut (bistabil) a bemeneti értékeknek és az aktuális tárolt állapotnak megfelelően.
  - **Q(n):** n. órajel **trigger** („**aktív él-váltás**”) állapota
  - Ha ekkor a tároló *Set* állapotban van, akkor Q(n) = ‘T’
  - Ha ekkor a tároló *Reset* állapotban van, akkor Q(n) = ‘F’
  - **Q(n+1):** n. utáni (következő) aktív élre (táblázat készítése összes lehetséges kombinációjára)
  - **Setup time**: megbizonyosodni az R / S control inputok stabil voltáról, az aktív él előtt röviddel.
  - **Hold time**: R / S control inputok stabilitása az aktív él után röviddel.

2/ d.) Szinkron JK flip-flop

- Ism: Az RS tárolónál *kétértelműség* volt az R /S bemenetek azonossága esetén (ott nem-megengedett volt!)
- Azonban **JK** esetén az összes R / S bemeneti kombinációra *egyértelmű* **kimeneti eredményt kapunk.**
- **Gerjesztési** tábla a JK tároló *logikai* vizsgálatához:
  - Megfeleltetés: **J:=Set / K:=Reset** - mint control bemenetek

Clock	J	K	$Q_{(n)}$	$Q_{(n+1)}$	
F	X	X	q	q	Szint-vezérelt viselkedés: Stabil ‘T’ v. ‘F’ esetén a JK kimenete érzéketlen
T	X	X	q	q	
↑	F	F	q	q	Hold
↑	F	T	q	F	Reset
↑	T	F	q	T	Set
↑	T	T	q	$\overline{q}$	Toggle (complement)

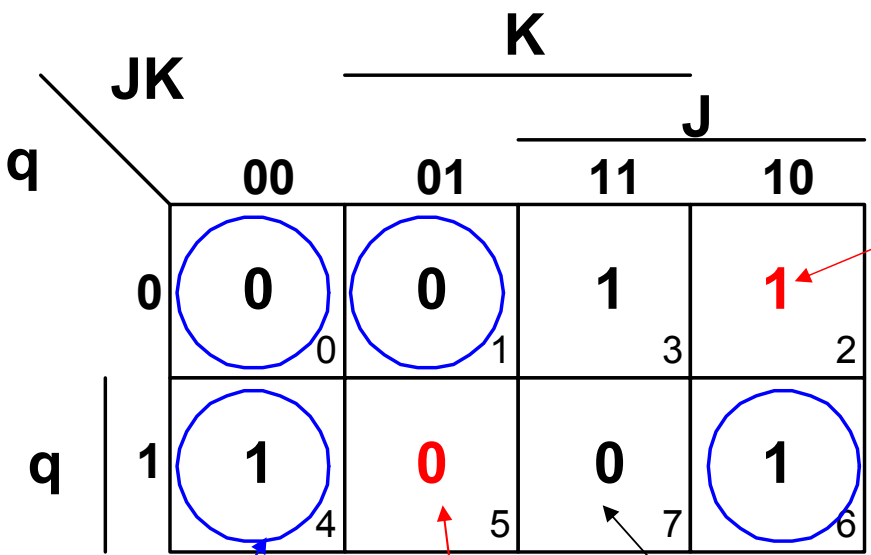
RS tárolóval analóg működést biztosít.

negált

↑ : CLK felfutó élére (pozitív) vezérelt (F → T)

JK flip-flop Karnough táblája

- Nagyon hasonló az RS-tárolóhoz, de itt az J=K=1 állapot is megengedett (toggle)!



Ha J=1, írás akkor q: 0->1 lesz

(Set)

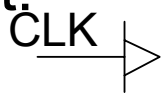
Stabil állapotok (tárol)

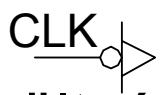
Ha K=1, törlés akkor q: 1->0 lesz

(Reset)

Megengedett állapot „toggle” mód (J=K=1)

JK tárolók típusai:

- Kereskedelmi forgalomban több típussal, jelöléssel is találkozhatunk:
  - Aktív-éle az órajelnek lehet:
    - Pozitív (felfutó) él-vezérelt: 

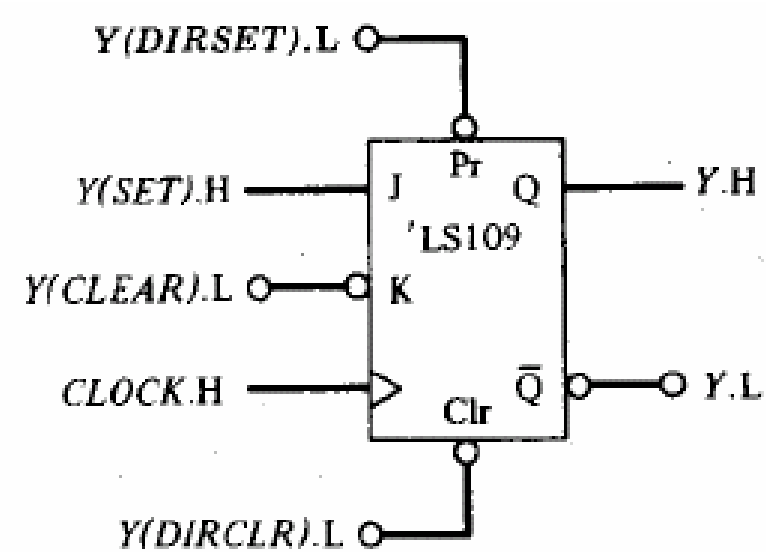
(F → T)
    - Negatív (lefutó) él-vezérelt: 

(T → F)
  - J / K control jelek aktív feszültség-szintjei:
    - J / K is magas aktív (T=H): pozitív logika
    - J aktív magas (‘H’) / K aktív alacsony (‘L’)
  - Aszinkron R / S módú viselkedés (availability):
    - Direct-clear (Pre-clear): aszinkron Reset (szimbólum alján)
    - Direct-set (Pre-set): aszinkron Set (szimbólum tetején)
  - Szinkron R / S módú viselkedés (alap)
    - Clear: lehet J, vagy K – control (logikai clear: Q=F)
    - Set: lehet J, vagy K – control (logikai set: Q=T)

Példa: 74LS109 Dual JK Flip-flop

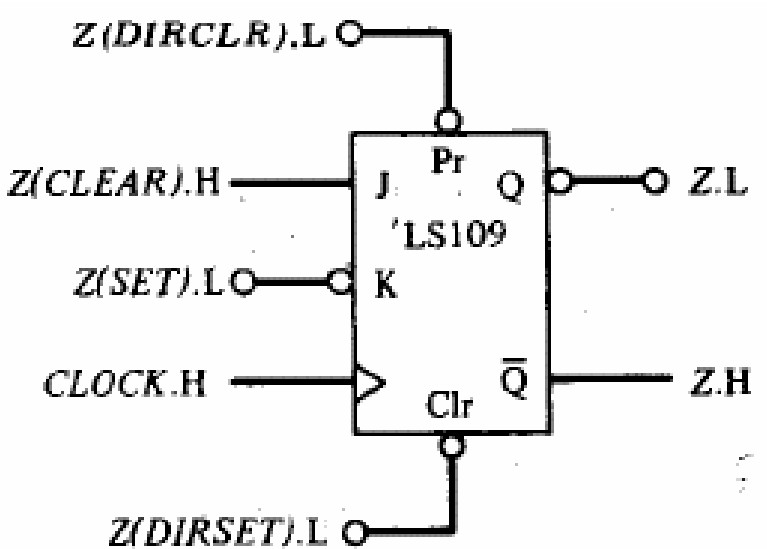
- Pozitív (felfutó) él-vezérelt SSI tároló elem
- Control jelei: J magas aktív (H) / K alacsony aktív (L)
- Set: Q-t ‘T’ be állítja (logikai set)
- Clear: Q-t ‘F’ be állítja, törli (logikai clear)

Q (T=H) esetén



(a) Setting with J, clearing with K

Q (T=L) esetén



(b) Setting with K, clearing with J

Példa: 74LS109 Gerjesztési tábla – feszültség értékek esetén

Q: T=H eset!

Preset	Preclear	Clock	J	K	$Q_{(n+1)}$	Action if Q is active-high
L	H	X	X	X	H	Direct set
H	L	X	X	X	L	Direct clear
L	L	X	X	X	—	Disallowed configuration
H	H	↑	L	L	L	Clear (Reset)
H	H	↑	L	H	$Q_{(n)}$	Hold
H	H	↑	H	L	$\sim Q_{(n)}$	Toggle
H	H	↑	H	H	H	Set

- $\sim Q_{(n)} = Q_{(n)}$ negáltja (komplemense: „toggel mód”-ban)
- X: don’t care állapot
- –: nem megengedett állapot

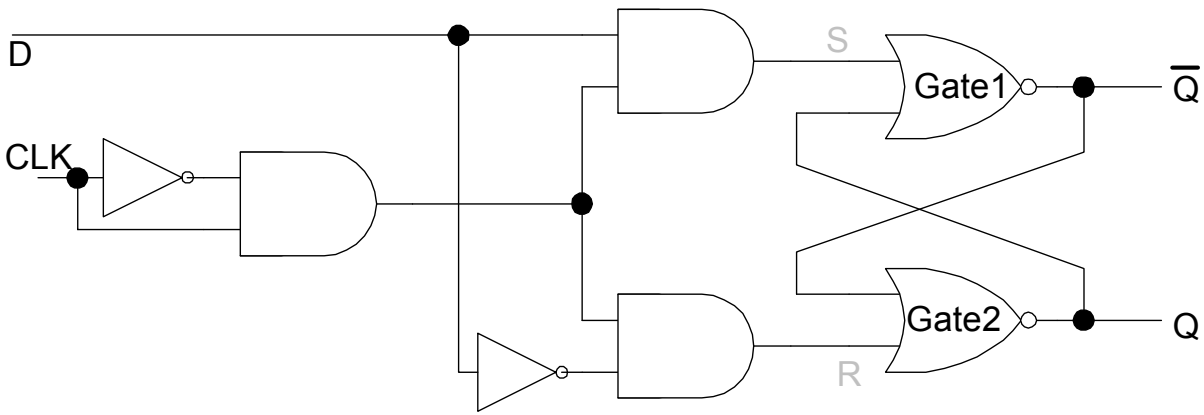
Aszinkron viselkedés

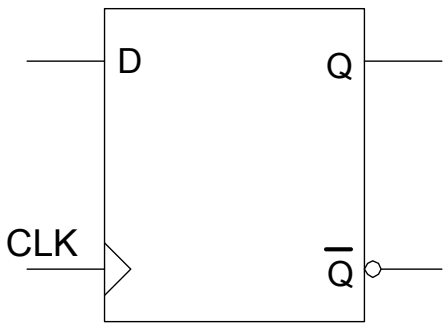
JK Flip-flopok általános tulajdonságai

- Általánosan használt, jól controllálható működésű tároló (flexibilitás)
- Adattárolásra (hold): gerjesztési tábla alapján
  - Q(n+1) = Q(n) = q lesz bármikor, ha J=K=F (felfutó élre) – ez egy egyszerű adattárolási mód
- Adatbevitelre: J / K nem adat, hanem vezérlő vonalak! Háromféleképpen használható:
  - a.) Clear -> majd Set
  - b.) Set -> majd Clear
  - c.) Tárolás: egy órajel ciklus ideig (részletesen a könyvben)

2/ e.) Szinkron D-flip-flop

- Rendkívül egyszerű működésű, általános tároló elem.
- **D** (Delay) tároló: késleltetéses-alapú működés (egy órajel ciklus ideig tartja az értéket, amelyet a következő ciklusban a kimenetére helyez)
- **CLK**=0→**1** értékére működik (1→**0**-nál tárolt érték (q(n))
- Kapcsolása, szimbóluma, és logikai gerjesztési táblázata:

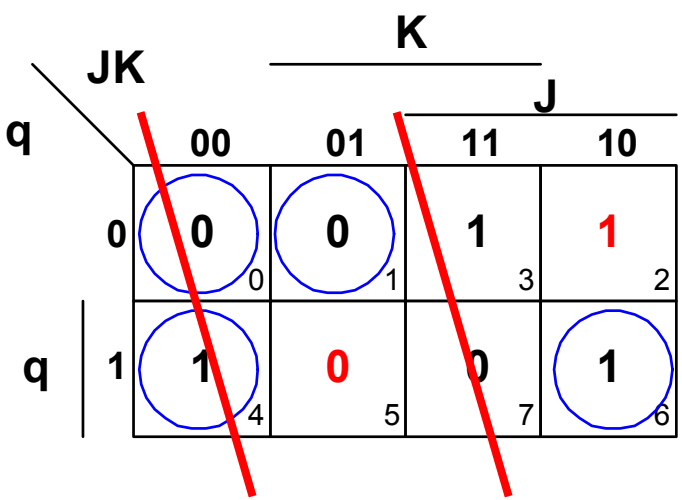


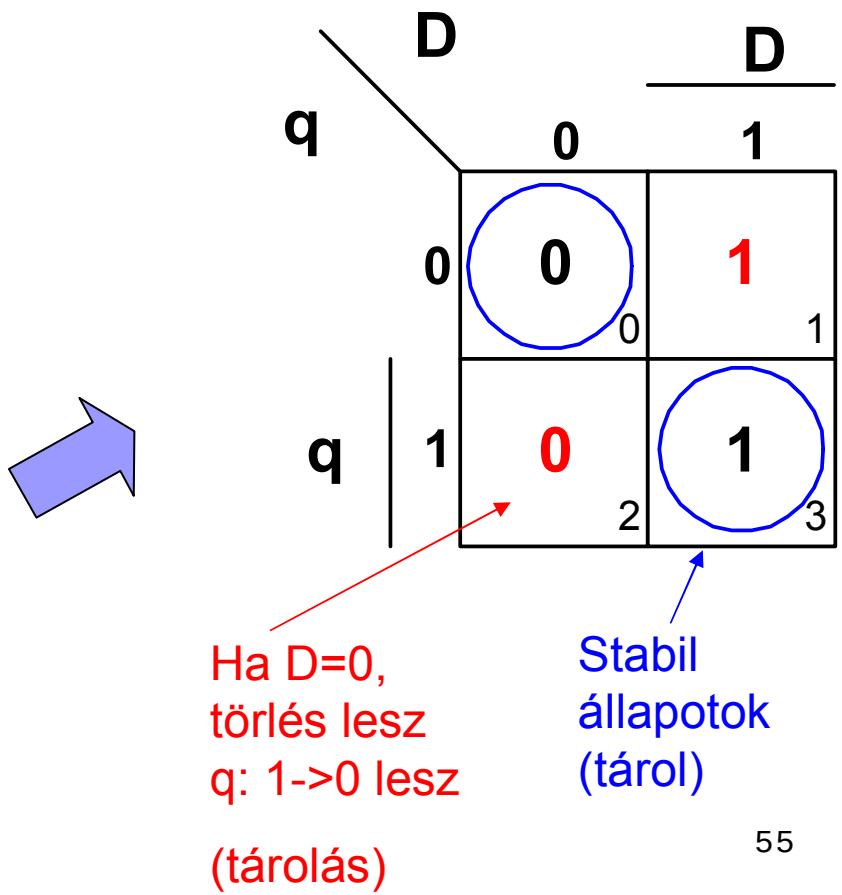


CLK	D	Q(n+1)
0	0	Q(n)
0	1	Q(n)
1	0	0
1	1	1

D-flip-flop Karnogh táblája

- JK tárolóból származtatható, ahol csak a **különböző** bemeneti értékű JK kombinációk a megengedettek → D



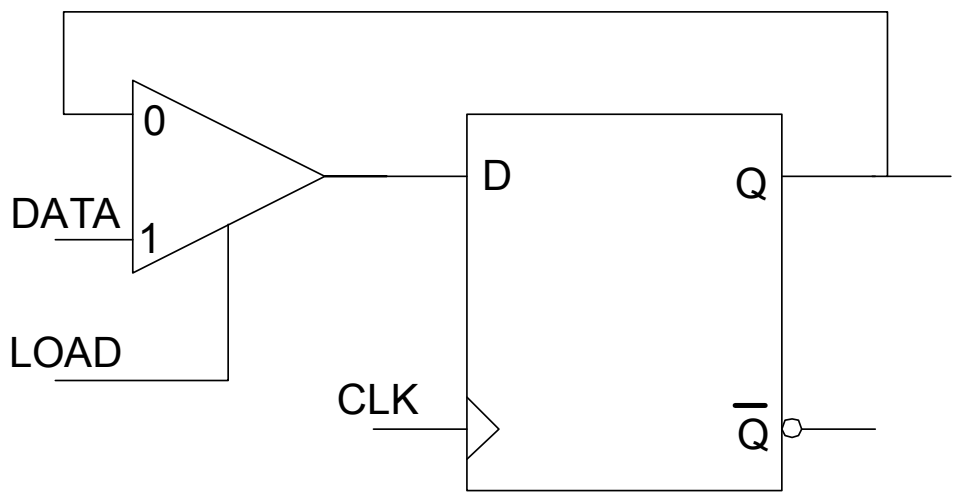


D-tároló jelei

- Kereskedelmi forgalomban a következő variációkban, jelölésekkel kapható
  - i.) CLK órajel aktív éle:
    - Pozitív (felfutó): L → H
    - Negatív (lefutó): H → L (kis kör jelöli)
  - ii.) Direct (aszinkron) Set / Clear (ha jelölik akkor általában alacsony aktív állapotúak – 1's catcher)
  - iii.) Legtöbb esetben csak a Q kimenete van feltüntetve, vagy a negált Q-val biztosítja mindkét polarítást

D-tároló alkalmazása, mint:

- **Késleltető elem**: egy órajel ciklusig késlelteti a bemenetre adott értéket, mire az a kimenetre kerül
- **Szinkronizáló elem**: különböző jelek rendszer órajelhez való időbeli ütemezése (szinkronizálatlan jel a bemenetén).
- **Adat tároló elem**: adatbevitel és tárolás
- **Engedélyező elem**: LOAD engedélyező bemenet (MUX-on keresztül választódik ki a bemenet, vagy v.cs. értéke

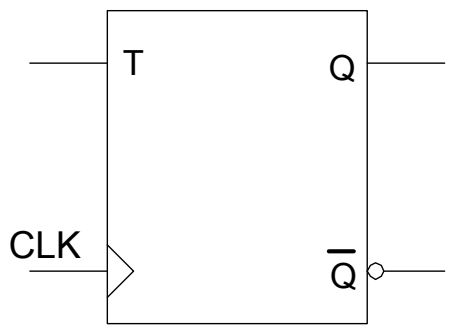


• LOAD = 0, visszacsatolt Q állapot kering

• LOAD = 1, külső bemenet (DATA)

2/ f.) Szinkron T-flip-flop

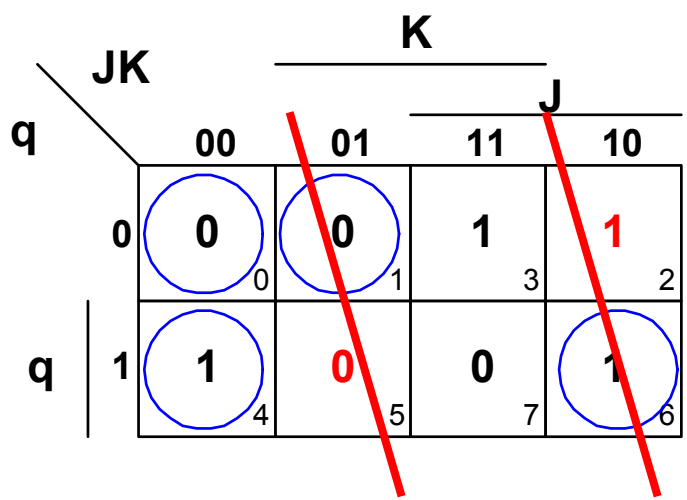
- Rendkívül egyszerű működésű
- **T** (toggle) tároló: késleltetéses-alapú működés (egy órajel ciklus ideig tartja az értéket, a következő ciklusban viszont az érték **negáltját** teszi a kimenetére)
- CLK=0->1 értékére működik (tárol)
- szimbóluma, és logikai gerjesztési táblázata:

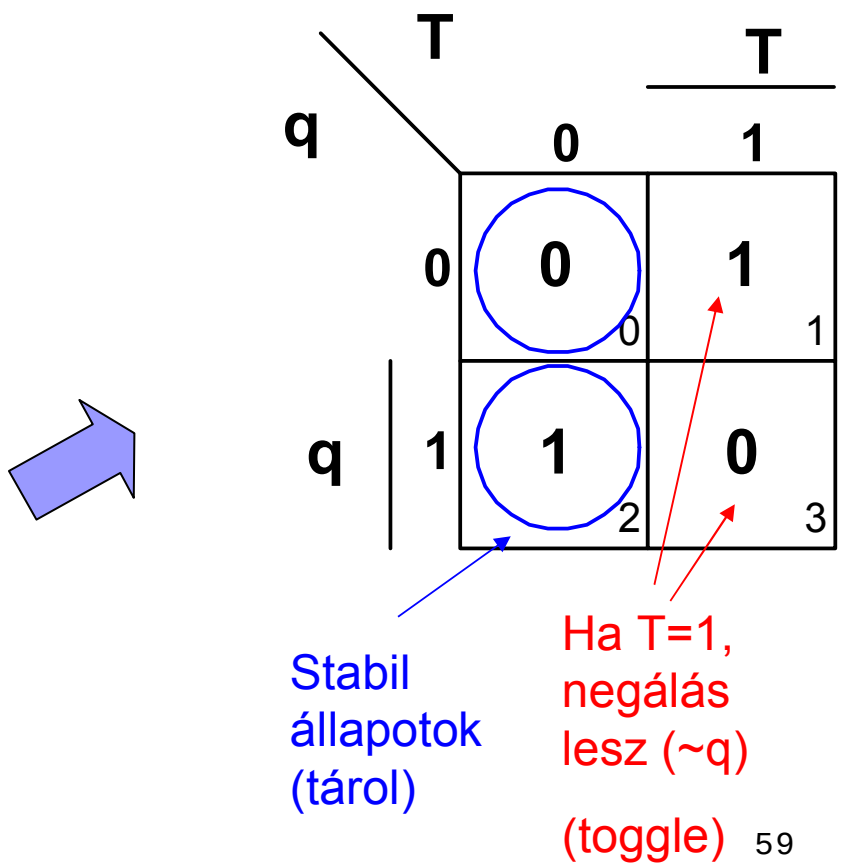


T	Q(n+1)
0	1
1	0

T-flip-flop Karnogh táblája

- JK tárolóból származtatható, ahol csak az **azonos** bemeneti értékű JK kombinációk a megengedettek → T
- A tárolót a JK bemenetek összekötéséből kapjuk!

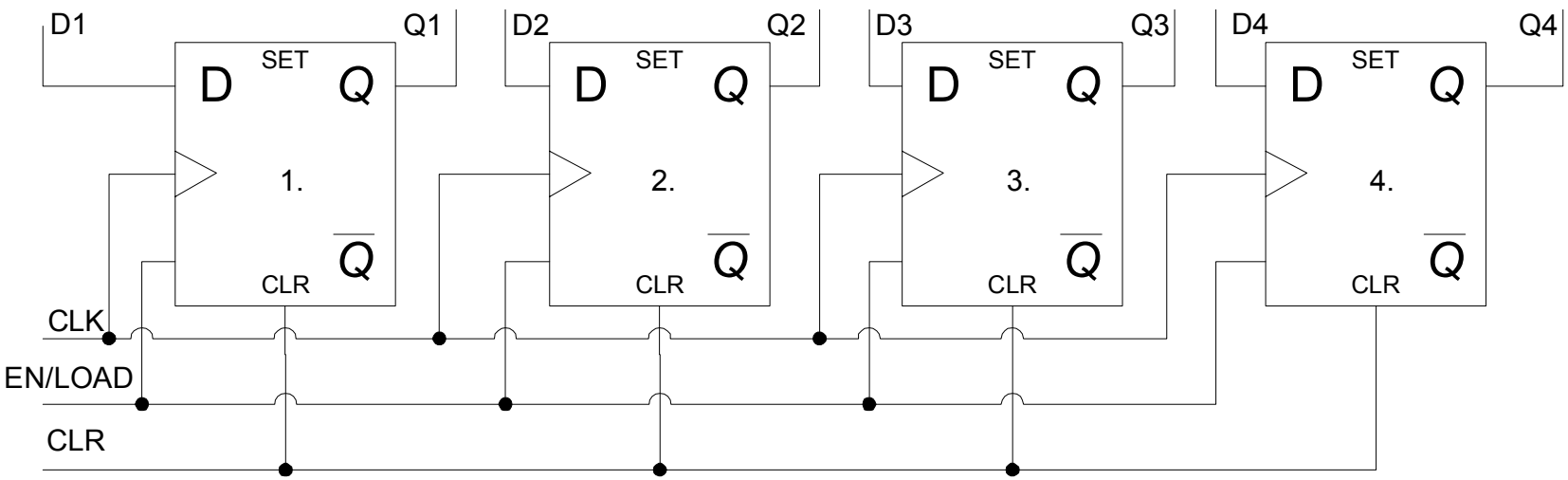




Regiszterek

- **Regiszter**: n db tároló (Flip-flop) elemekből felépülő rendezett tömb
- Ideiglenes adattárolásra használjuk (néhány bites érték tárolása aritmetikai műveletekhez)
- Byte, vagy word (szóhosszúság) szervezésű
- MSI-szintű építőelem
- Példák:
  - Engedélyező bemenetű D-regiszter (EN)
  - Tiszta (pure) D-regiszter (EN=1 nek feltételezzük)
    - 4, 6, 8, 16, 32 …számú D Flip-flop-ból épülhetnek fel.
    - Közös órajel (esetleg törlő és engedélyező jel)
  - Shift-regiszter (kimenetek sorba kötésével – léptetés)

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



LSB MSB

LOAD: a D1…D4 bemeneteknek párhuzamos beírására is lehetőség van. (LOAD=1)

Számlálók (counters)

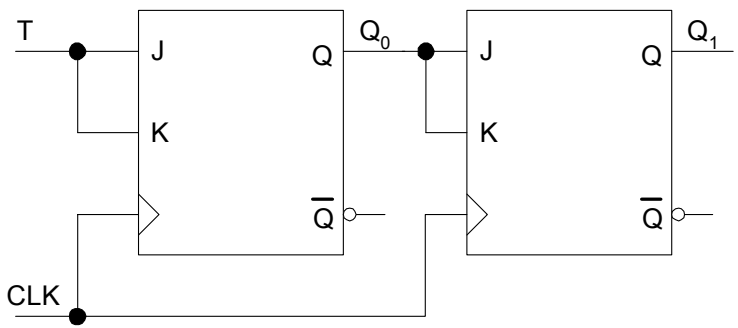
- Bináris számlálók
  - Moduló-N számláló: M moduló N (M / N utáni maradék) értékét tárolja
  - 3-bites számláló: 10^3 különböző értékre 000-999 –ig, majd 999 után újból 000-tól indul, inicializálódik. (Ez egy „M moduló 1000” számláló.)
- Számláló JK tárolókból: „toggle mode”-ban használjuk a tárolót
  - Modulo-2 számláló: (q negált).
    - CLK impulzusa: 0 1 2 3 4 5 6 7 8 …
    - FF Q kimenete: 0 1 0 1 0 1 0 1 0 … (q / $\overline{q}$) //alternáló jelleg
  - Szinkron Modulo-4 számláló: (közös CLK)
    - CLK impulzusa: 0 1 2 3 4 5 6 7 8 …
    - FF Q_1,Q_0 kimenete: 00 01 10 11 00 01 10 11 00 …

Logikai értékeket reprezentál

LSB bit: Q_0 balra (alternáló jelleg)

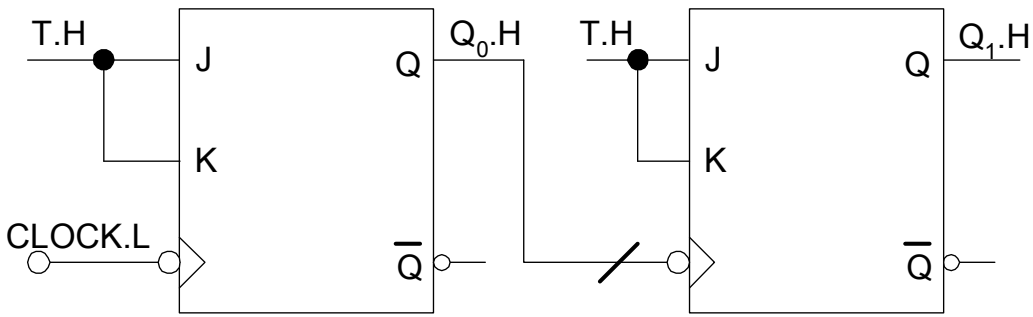
MSB bit: Q_1 jobbra (akkor változtatja az értékét alternáló jelleggel, amikor Q_0 =‘T’ / ‘1’)

Ezekből nagyobb méretű modulo-N számláló is felépíthető sorbakötésükkel!



Aszinkron bináris moduló-4 számláló JK tárolóból

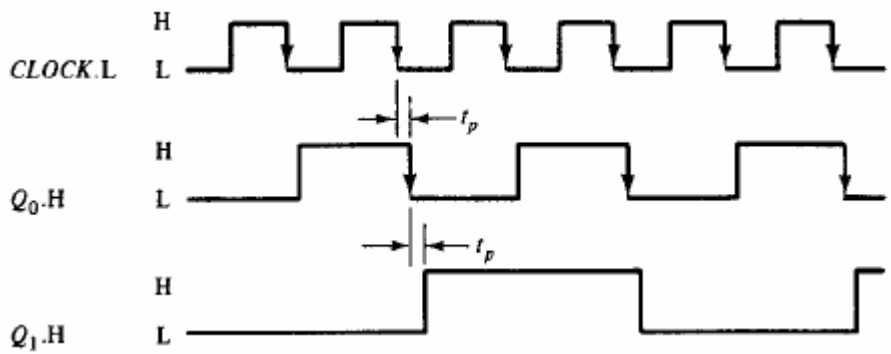
- Hasonlóan működik az előzőhöz:
  - Q_0 alternál („toggle mode”) minden CLK-ra (mivel J=K=‘T’) //bal FF
  - Q_0 generál egy T→F (H→L ebben az esetben) átmenetet
  - Q_1 is alternál, de Q_0 tól függően (J=K=‘T’ re) //jobboldali FF
  - Q_1 –es FF órajelét a Q_0 biztosítja (aszinkron működés)



LSB bit: Q_0 balra (alternáló jelleg)

MSB bit: Q_1 jobbra (akkor változtatja az értékét alternáló jelleggel, amikor Q_0 =‘T’ / ‘1’)

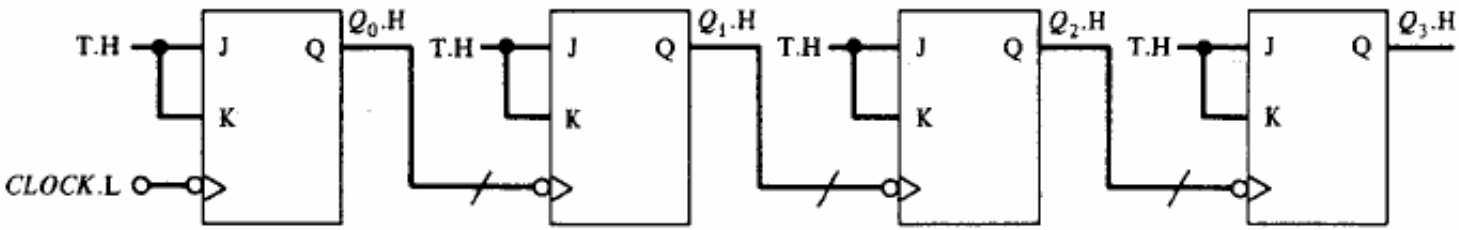
- Idődiagram:



t_p: propagációs késleltetés [ns]

4-bites moduló-16 számláló (counter)

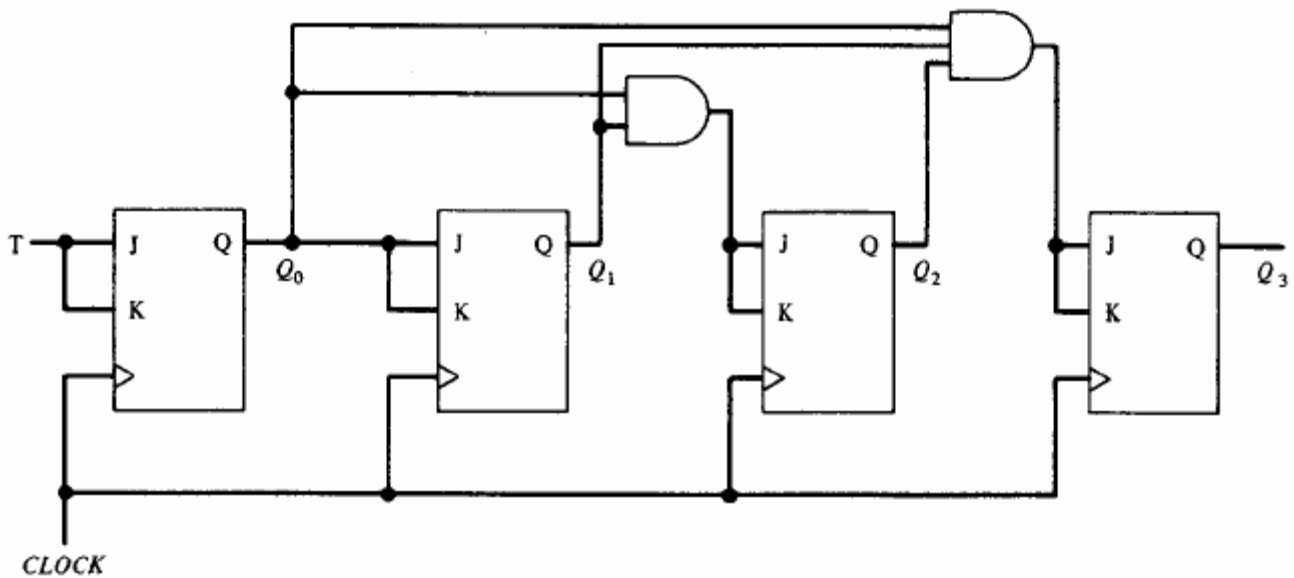
- Aszinkron (ripple) counter:



Pl. átmenet 3 -> 4 között

Time	Q_2	Q_1	Q_0
0	0	1	1
t_p	0	1	0
$2t_p$	0	0	0
$3t_p$	1	0	0

- Szinkron counter:



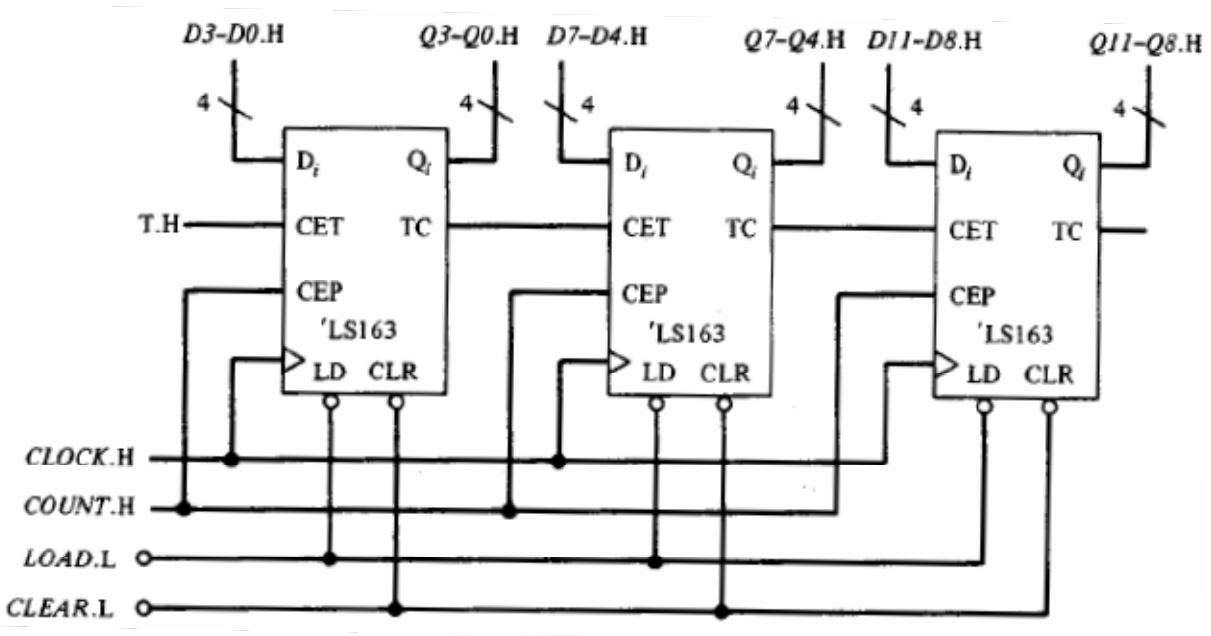
Extra AND kapuk:

„toggle-mode” szinkronizációja az egyes FF-nál

Közös CLK!

Szinkron MSI számlálók:

- Kereskedelmi forgalomban is kaphatóak
  - Moduló-10 (dekád) számláló
  - Moduló-16 (4-bites bináris) számláló
  - **Példa: 12-bites bináris számláló** 3 db ‘LS163 – 4-bites szinkron bináris számláló összekapcsolásából.



Közös CLK: szinkron működés

CLOCK.H: rendszer órajel

Szinkron Clear (0) / Load (Set)

TC: terminal count

CET: Count Enable Trickle

CEP: Count Enable Parallel (master)

Eszköz vezérlők

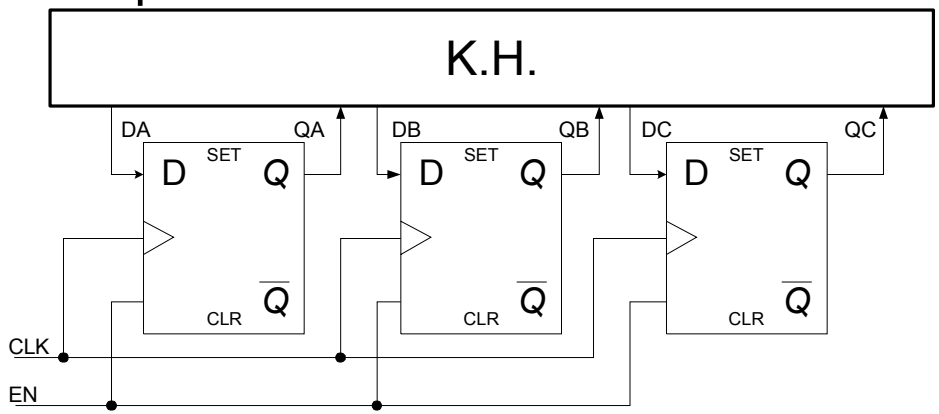
Példa: számláló tervezése D-FF-ből

- Tervezzünk bináris számlálót szinkron D-tárolóból, amely 0…5-ig tárolja az értékeket. (6 érték → összesen 3 db D-tárolóra lesz szükségünk).
- Igazságtáblázata:

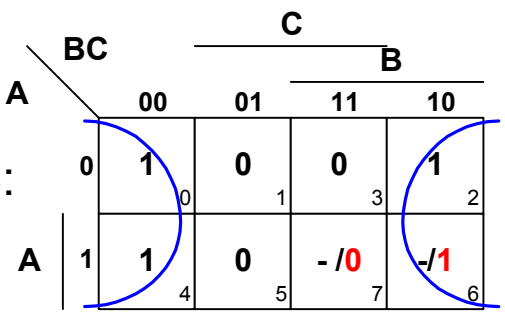
count	QA(i)	QB(i)	QC(i)	DA(i-1)	DB(i-1)	DC(i-1)	i. clk
0	0	0	0	0	0	1	1
1	0	0	1	0	1	0	2
2	0	1	0	0	1	1	3
3	0	1	1	1	0	0	4
4	1	0	0	1	0	1	5
5	1	0	1	0	0	0	6
0	-	-	-	-	-	-	-
1	-	-	-	-	-	-	-

egy órajellel később jelennek meg D*x* értékei a Q*x* kimeneteken!

Kapcsolása:



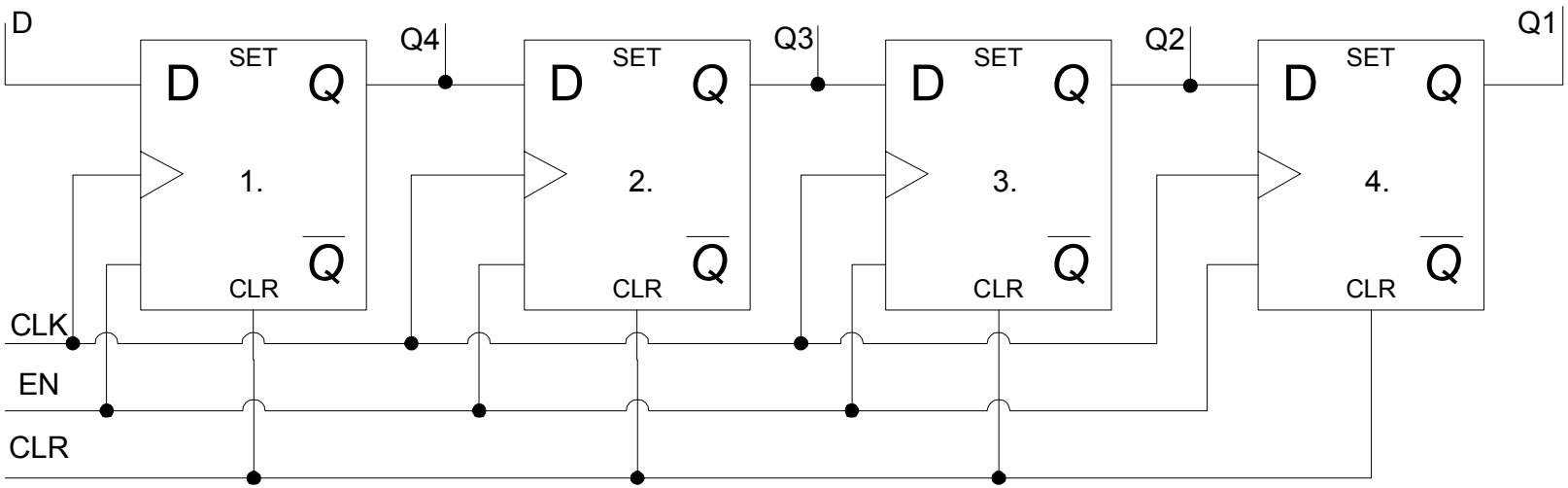
Realizáljuk pl. a DC-t (A,B,C kimenetű K.H-ból) ⇨ DC:



$$DC = \overline{C}$$

(Hasonlóan lehet képezni a Karnough táblákat a DA, DB-kre is!)

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



Shift regiszter. Oldalirányú (laterális) léptetés, az egyik bitpozíciótól a szomszédosig.

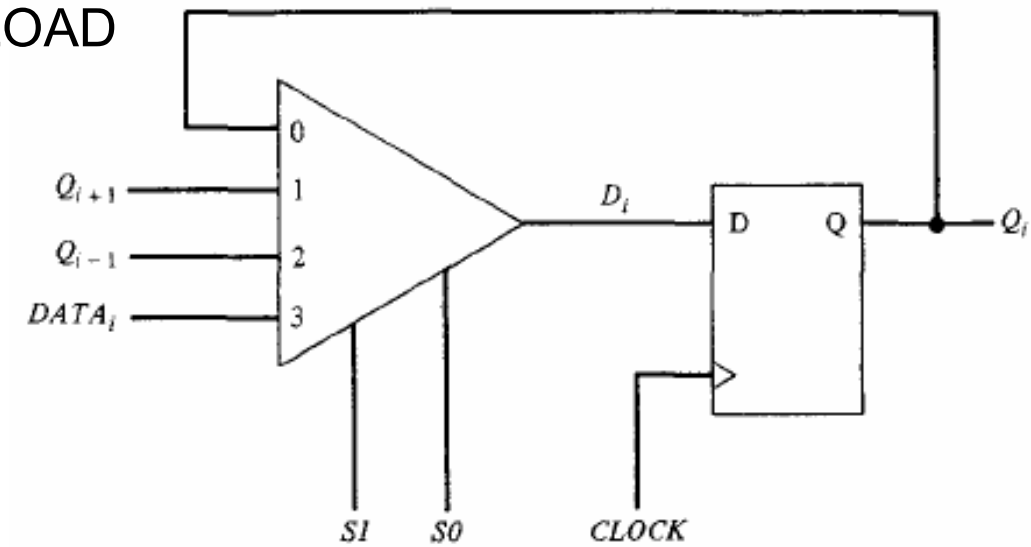
D: Data bemeneten lévő adatot lépteti sorosan balról-jobbra minden egyes órajel ciklusban. 1.clk-ban a 1. tárolóba a D1-et, majd 2 clk-ban 2. tárolóba a D1-et és az 1. be a D2-t.

Q: a kimeneteken párhuzamosan kapjuk az adatot

CLK	Q4	Q3	Q2	Q1
1	D1	-	-	-
2	D2	D1	-	-
3	D3	D2	D1	-
4	**D4**	**D3**	**D2**	**D1**
5	D5	D4	D3	**D2**
6	D6	D5	D4	**D3**
7	D7	D6	D5	**D4**

4-bites Shift-regiszter működése

- Közös szinkron CLK
- Két szelektor jel: S0, S1 (az állapotok kiválasztásához! – 4:1 MUX)
- 4-állapot: HOLD/ SHL /SHR /LOAD
- Kapcsolási rajz:



- Táblázat:

Clock	S1	S0	Result desired	Selected mux position	Required mux input
↑	0	0	Hold present data	0	Q_i
↑	0	1	Shift right	1	Q_{i+1}
↑	1	0	Shift left	2	Q_{i-1}
↑	1	1	Load new data	3	$DATA_i$

- Ajánlott: a fejezetek végén lévő feladatok (Exercises) részek áttekintése.

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

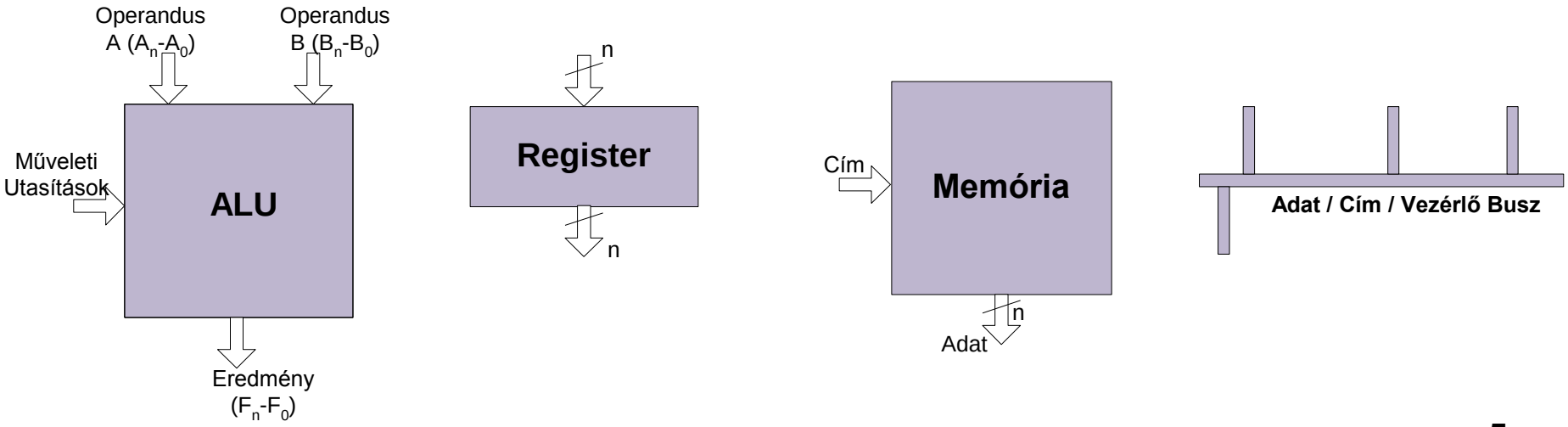
Utasítás végrehajtás folyamata

- 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

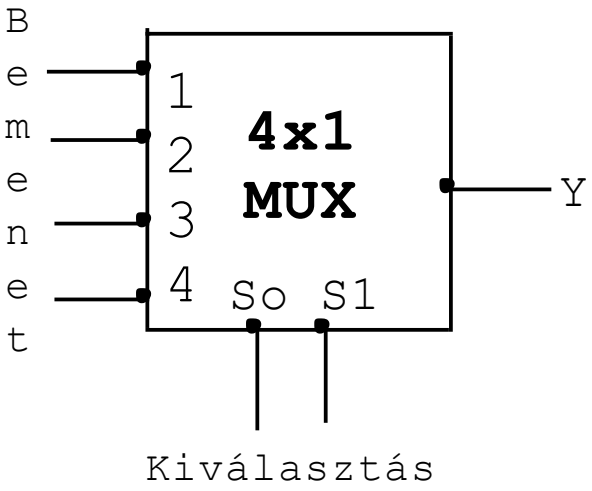
- Az ALU által kezelt / végrehajtott adatok a **memóriában** (tároló rekeszek lineáris tömbjében) tárolódnak el. A memória rekeszei általában olyan szélesek, amilyen széles az adatbusz. Például, legyen n-bit széles, és álljon M számú rekeszből. Ekkor $\log_2(M)$ számú címvezetékkel címezhető meg. Az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

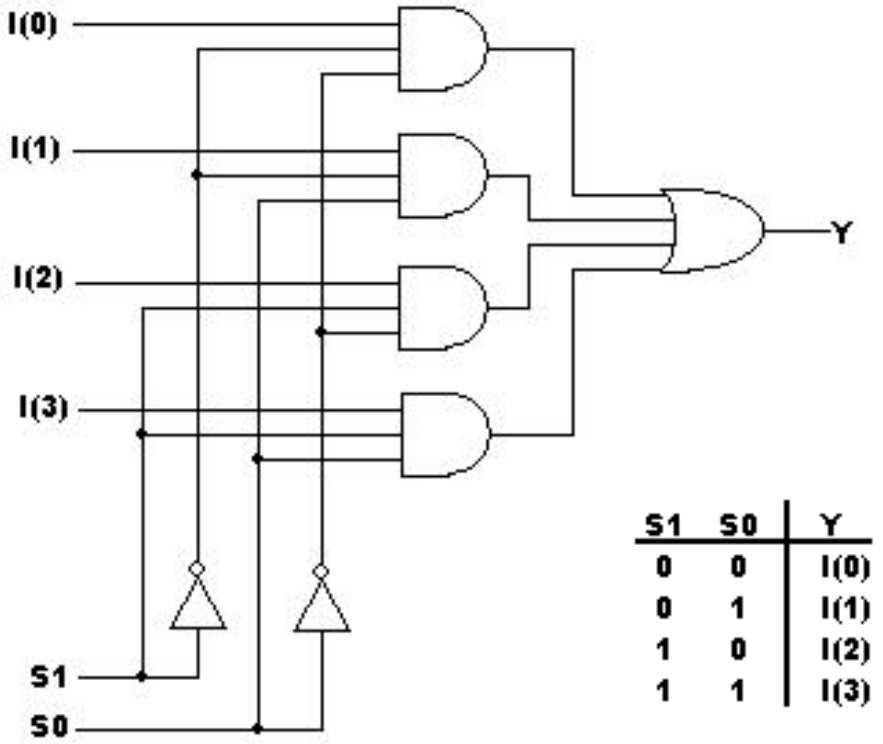
Adatbuszok – adatvonalak

- 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

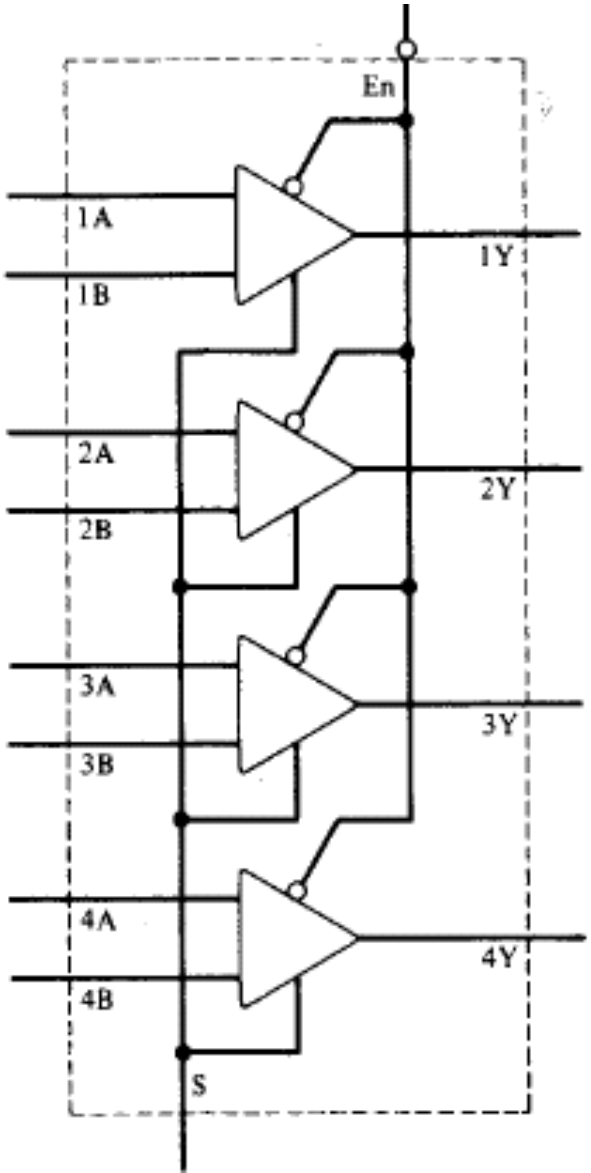




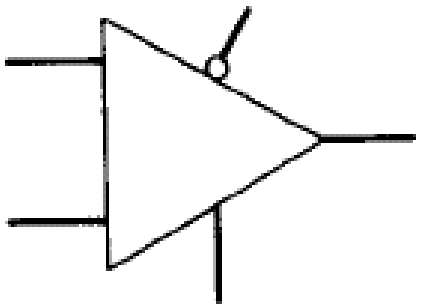
S1	S0	Y
0	0	I(0)
0	1	I(1)
1	0	I(2)
1	1	I(3)

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

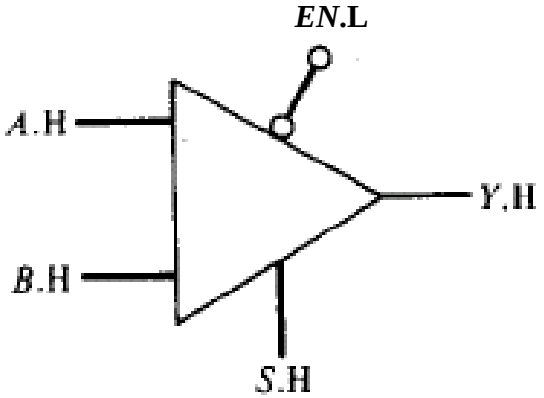
TTL’74LS157 pl. 2:1 MUX



- 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

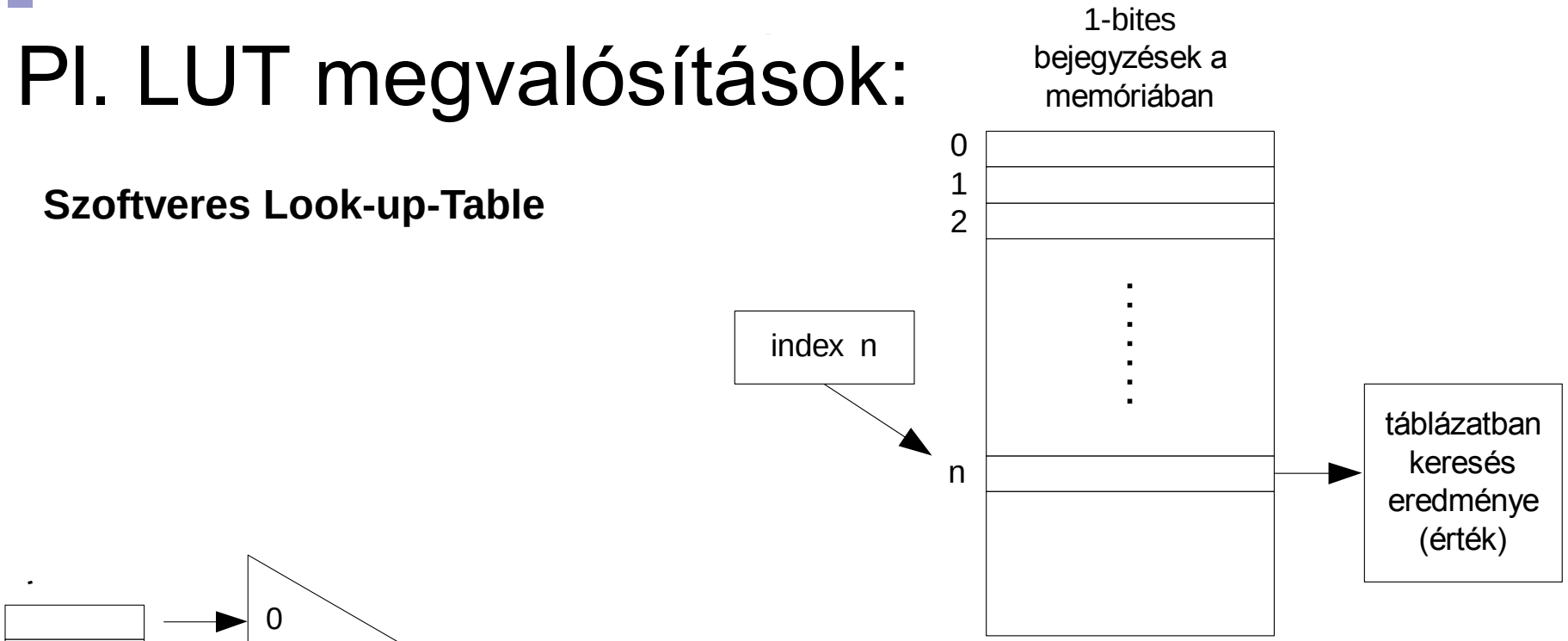


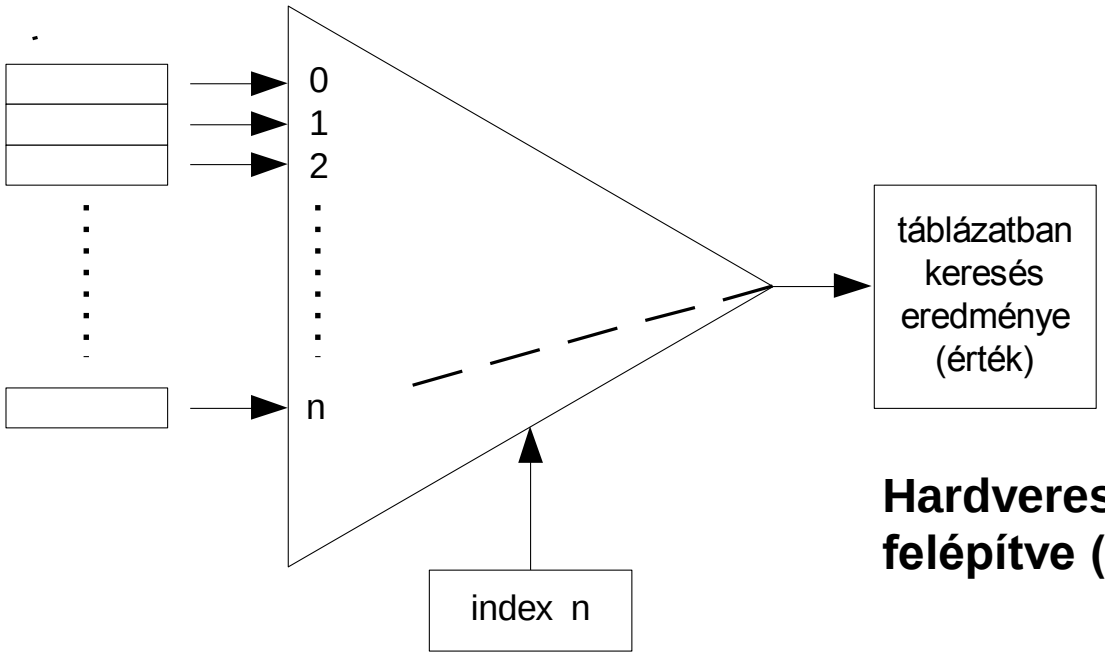
2:1 MUX áramköri szignál jelölésekkel

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

Pl. LUT megvalósítások:

Szoftveres Look-up-Table

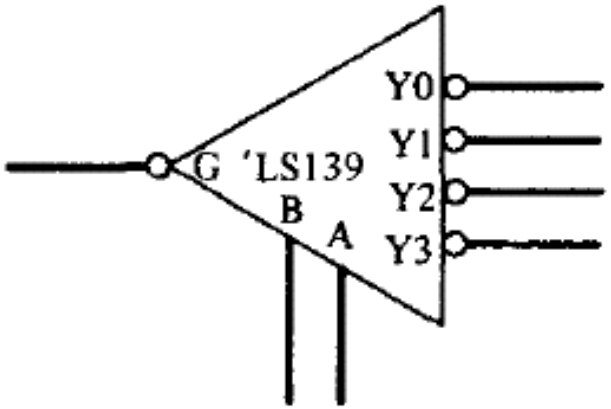




Hardveres Look-up-Table, MUX-ból felépítve (1 bites táblázatkeresés)

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

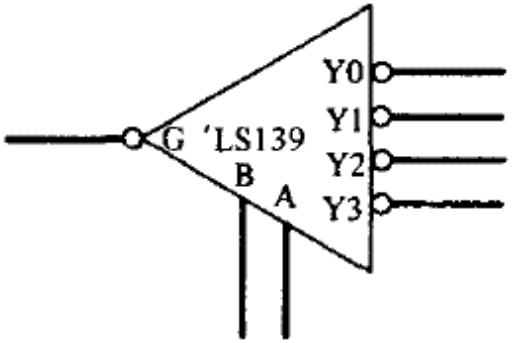
Demultiplexer logika						
G	B	A	Y0	Y1	Y2	Y3
0	x	x	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

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:



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

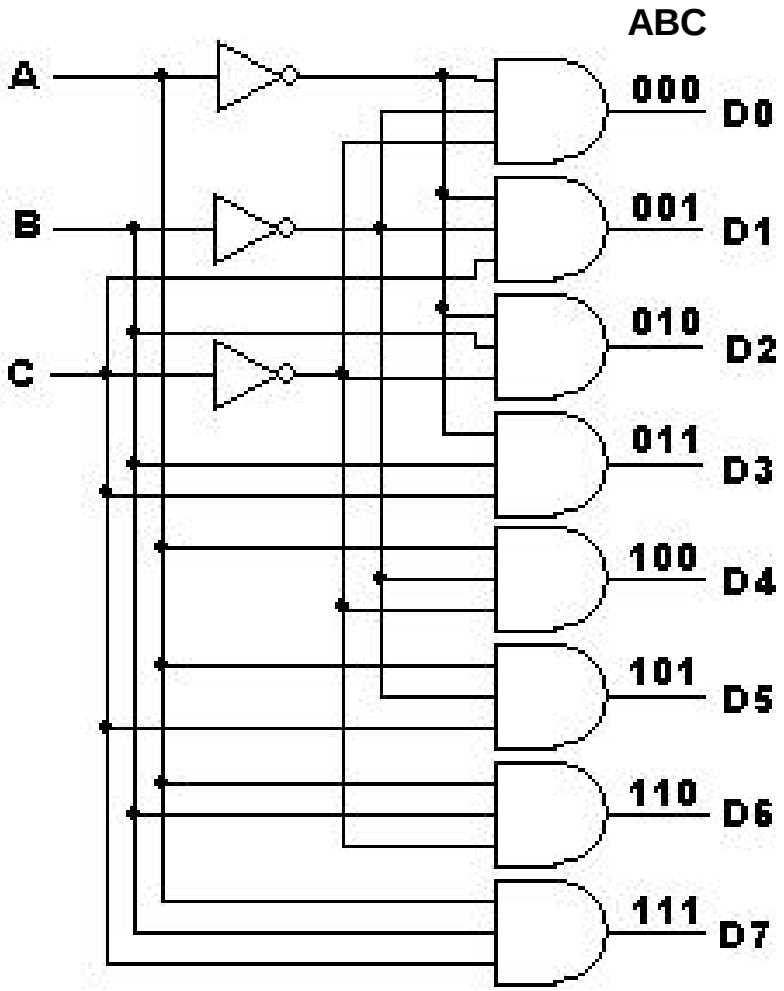
$$Y1 = \overline{B} \cdot A \cdot G$$

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

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

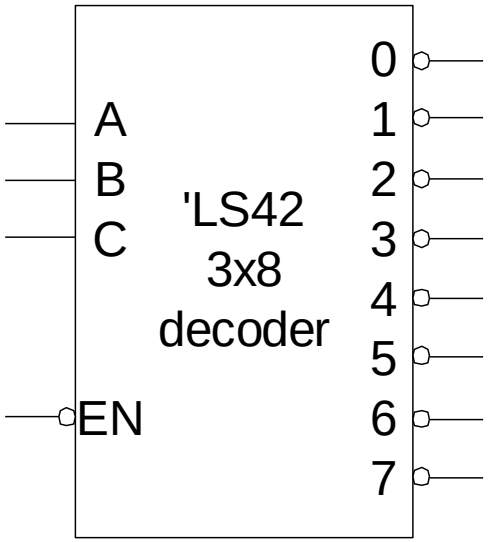
c.) Dekódoló áramkörök

- 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

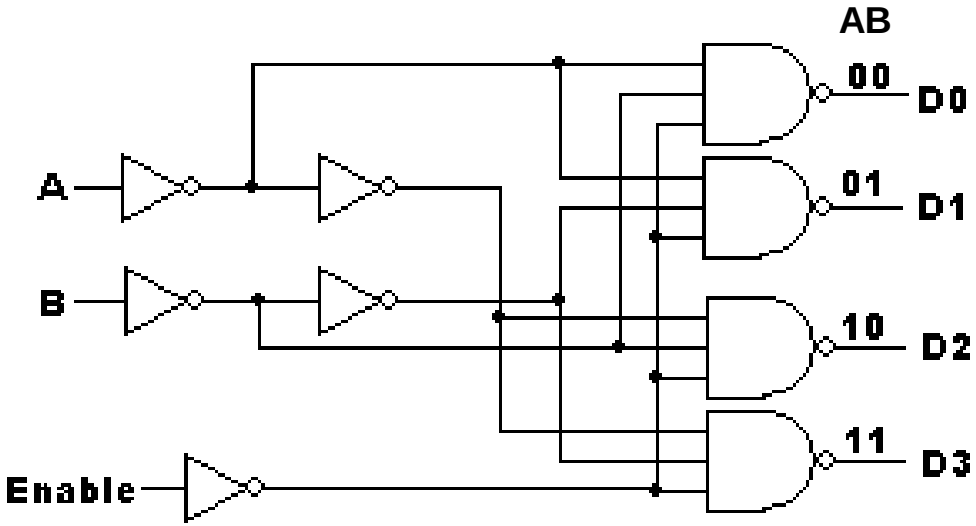


Mixed logic szimbólum

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

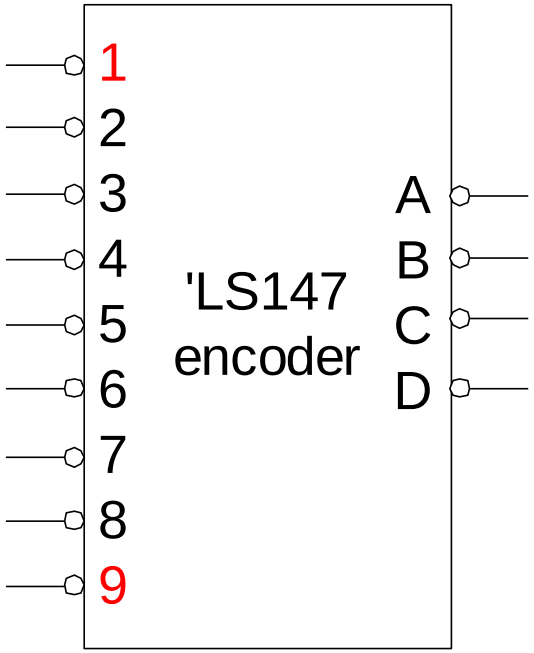
D3	*D2*	*D1*	*B*	*A*
F	F	F	F	F
F	F	T	F	T
F	T	X	T	F
T	X	X	T	T

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

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

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

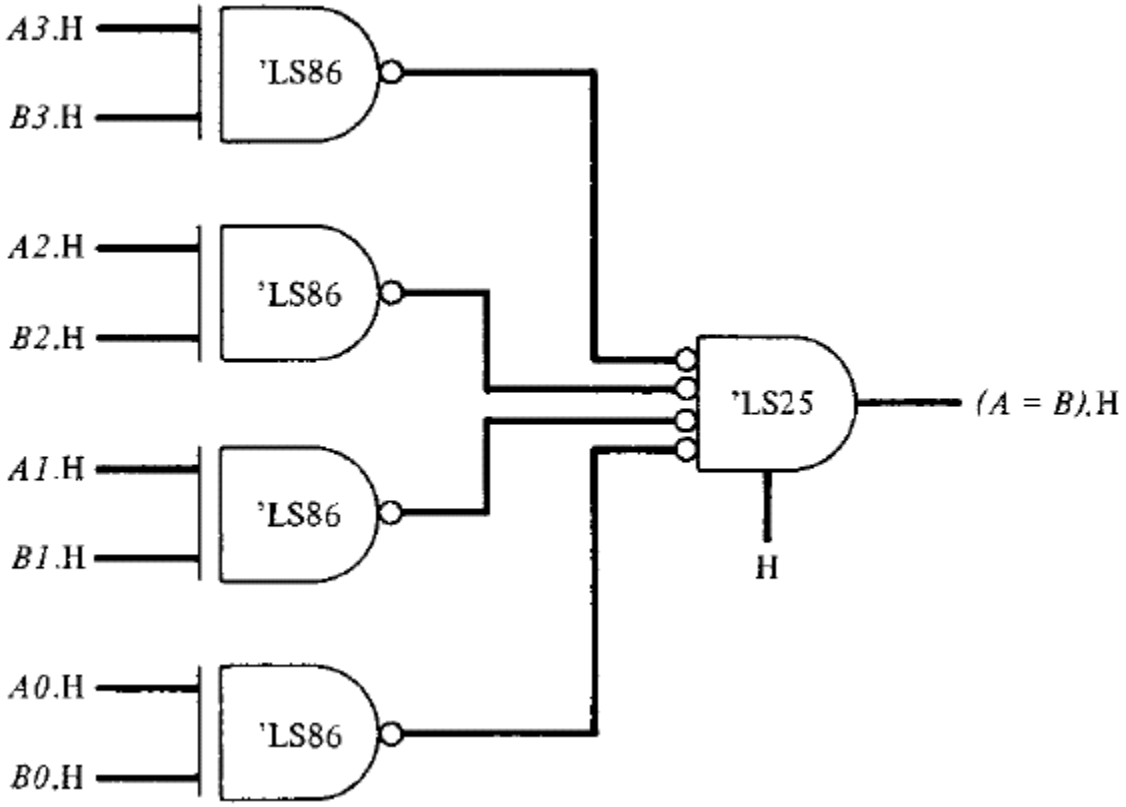
- Példa: legyen B egy 4-bites konstans mennyiség (B=TFFT), és A tetszőleges, akkor:

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

Példa: 4-bites komparátor

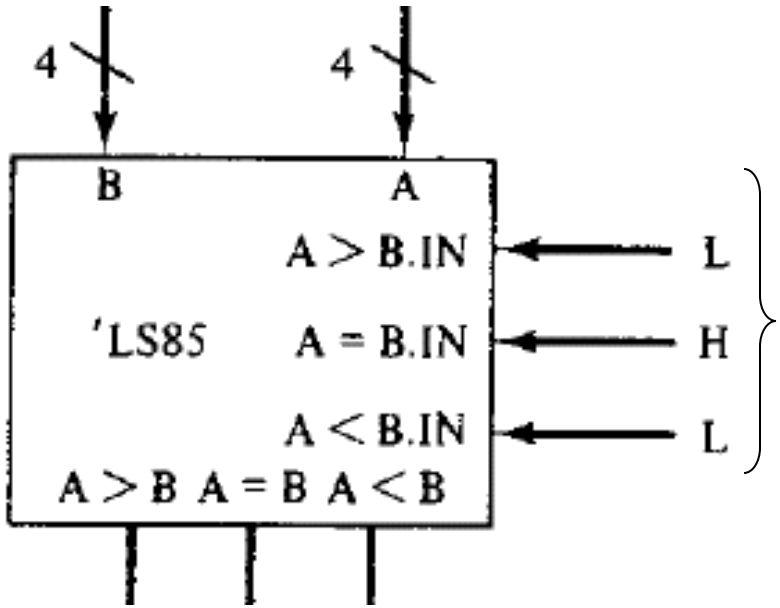
- Mixed-logic kapcsolási rajza, és log.egyenlete:

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



’74LS85 4-bit Magnitude Comparator

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

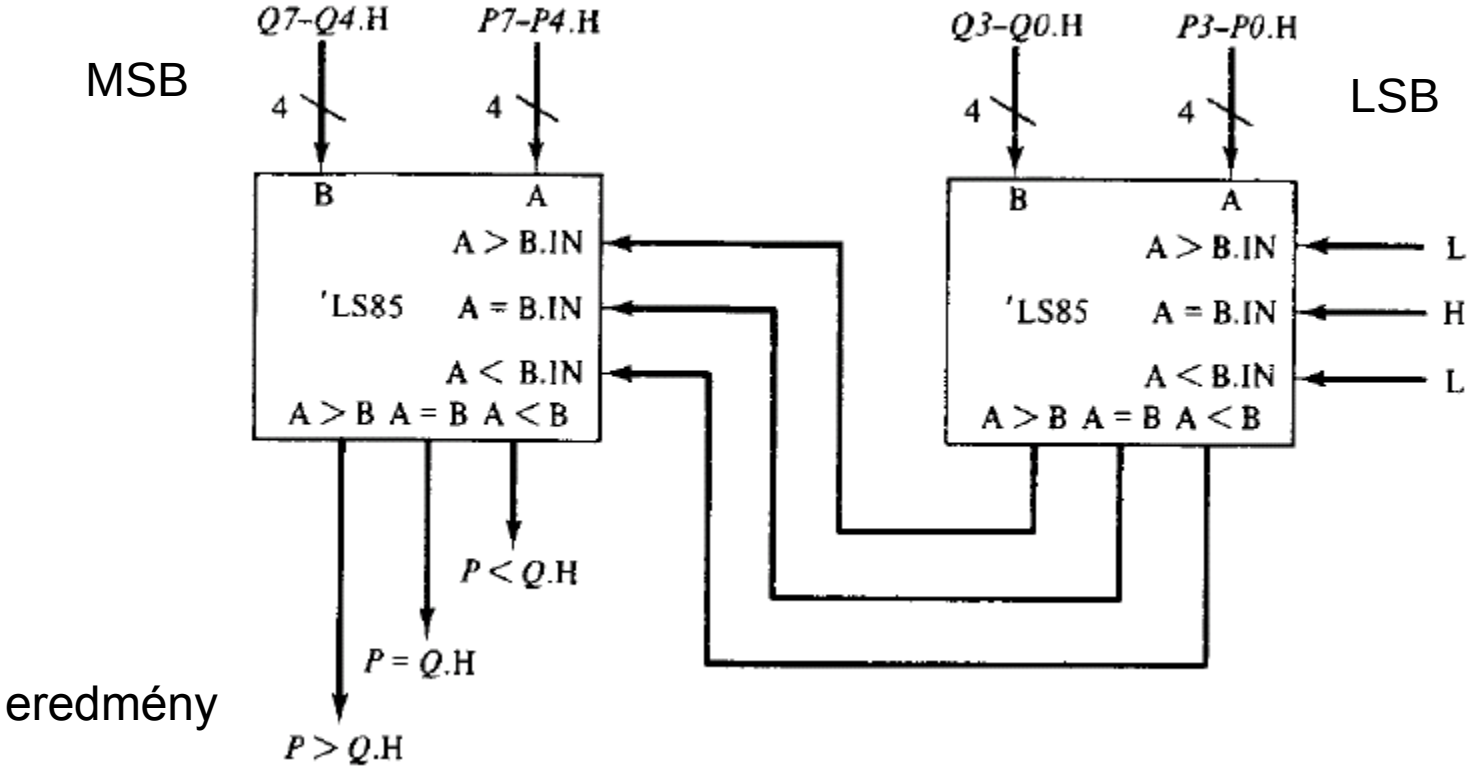


3 állapot (státusz) –bemenet

Ha B.IN=H: egyenlőség

Példa: 8-bites Magnitude Comparator – egyenlőség esetén

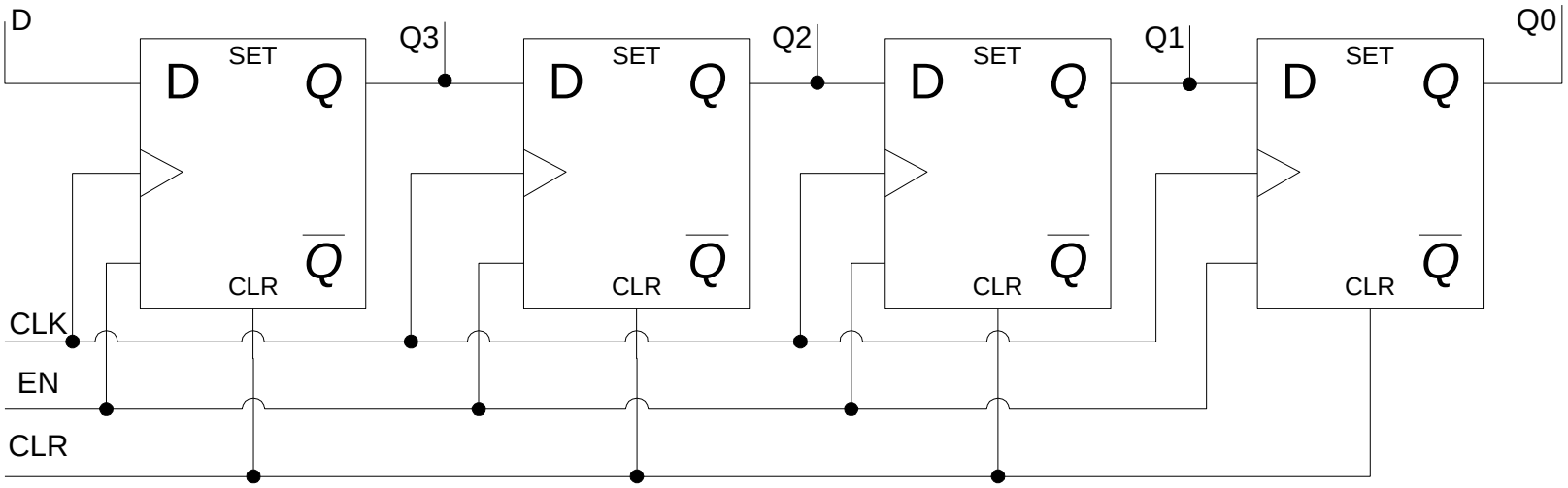
- Kettő 4-bites ’74LS85 Magnitude komparátor sorbakötéséből („cascading”) kapjuk a 8 bites (P,Q) értékek összehasonlítását



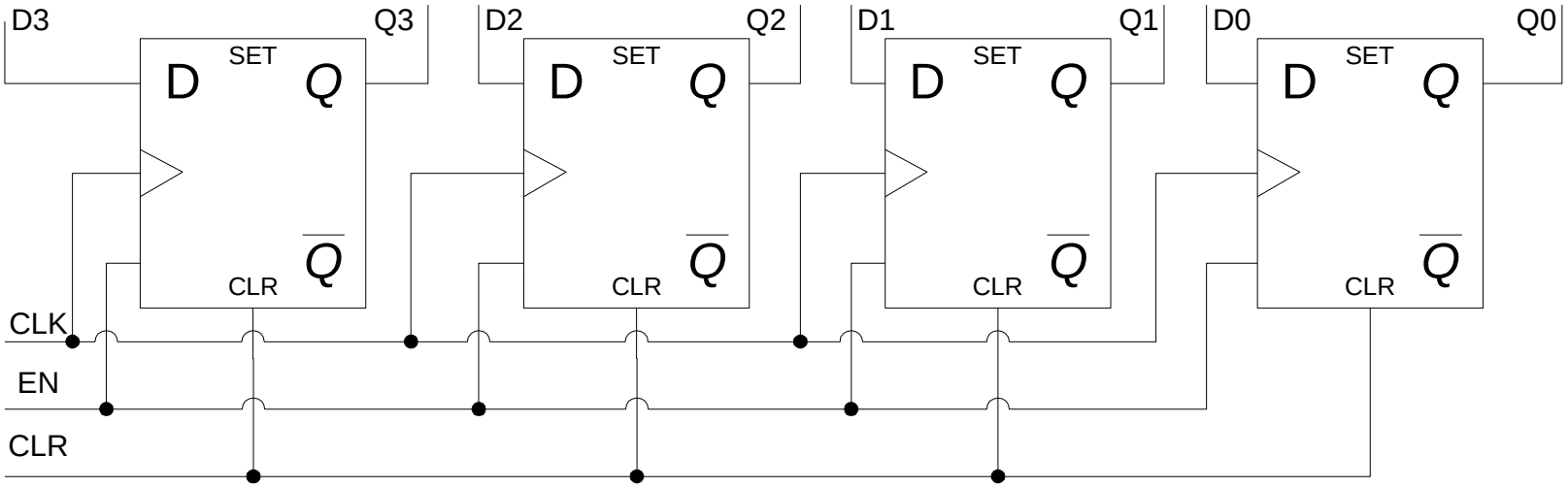
Regiszterek

- 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)

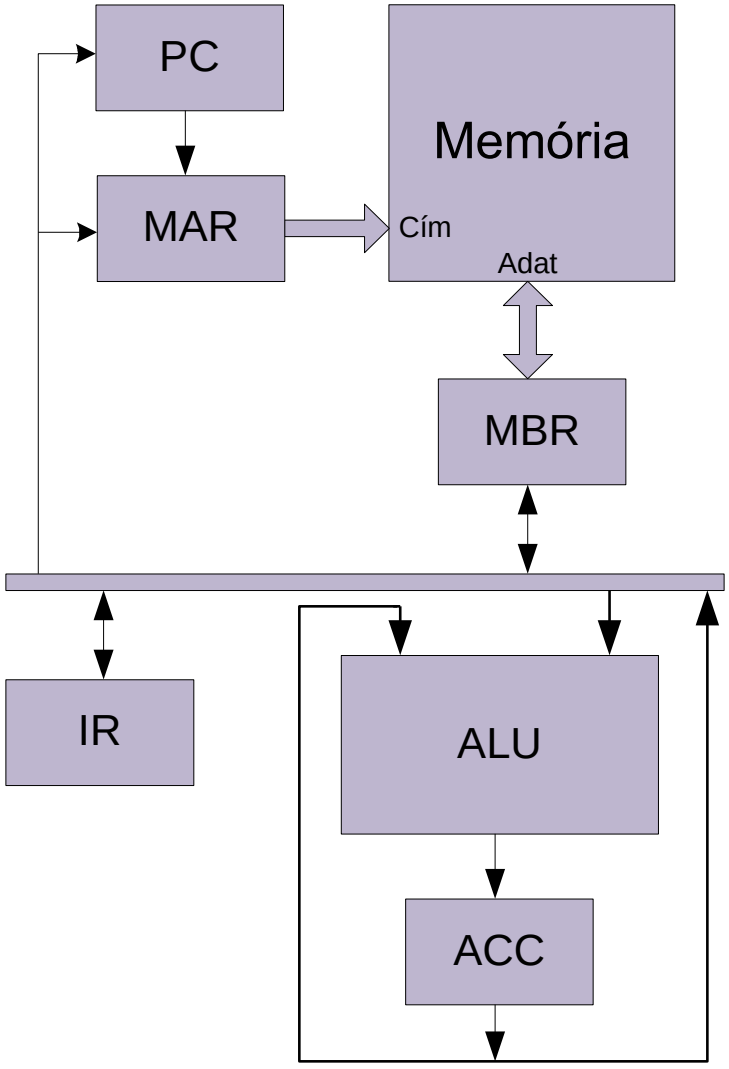


4-bites Parallel In/ Parallel Out regiszter (D-tárolókból felépítve)



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)

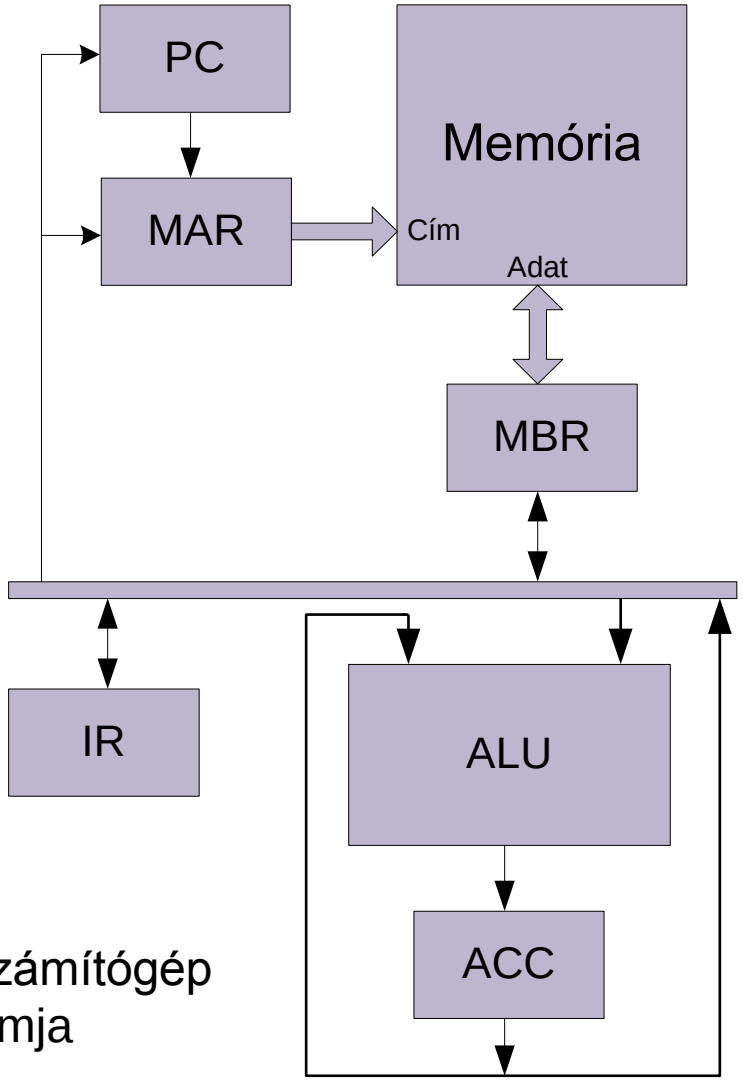
Execute:(végrehajtás)

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

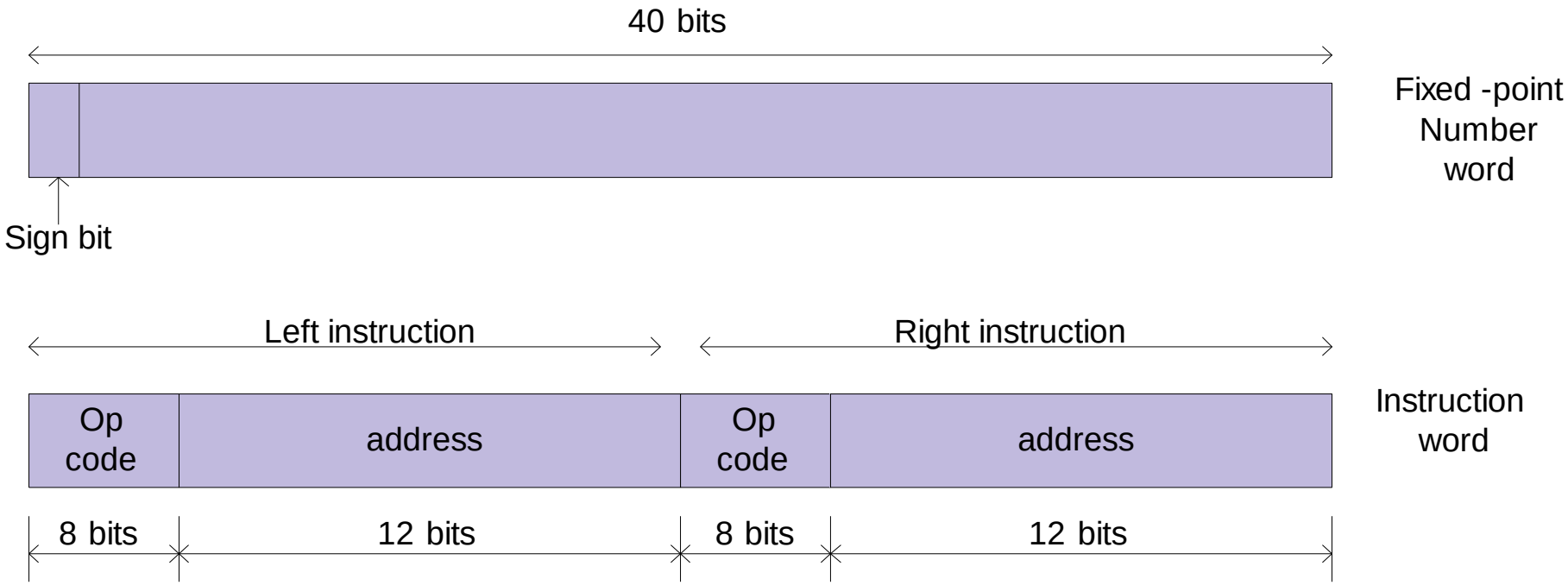
Decode: (a dekódolást általában 0 idejűnek feltételezzük)

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)

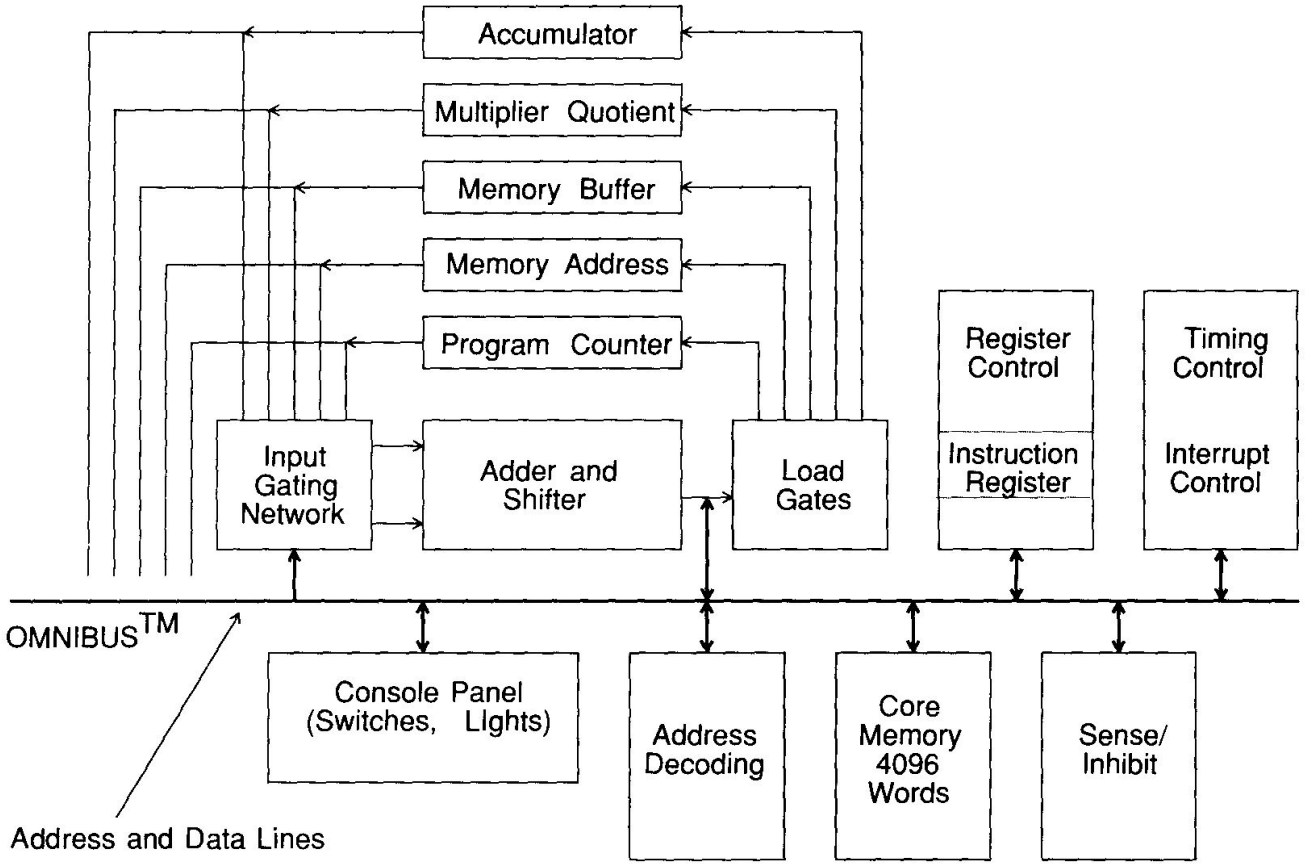
Execute: (végrehajtás)

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

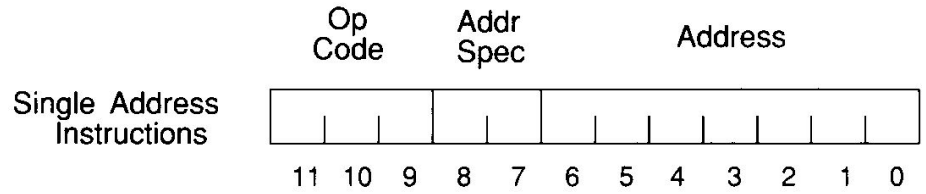
Példa: DEC PDP 8 (egycímű gép)

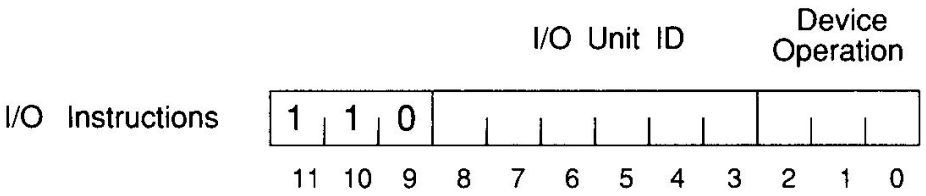
- 12-bites szóhossz: 3-bites opcode (8 művelet) +
  9-bit utasítás cím (speciális operandus címzési módokat tett lehetővé)



DEC PDP-8 utasításformátuma

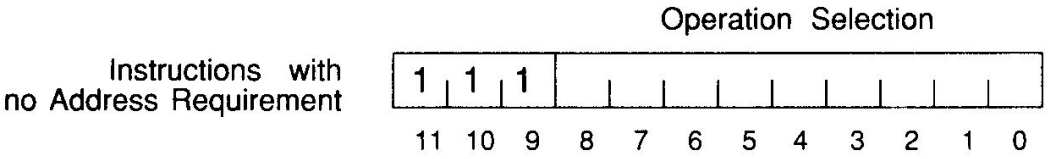
- 3-bites opcode [11:9]: 8 művelet , ebből:
  - hatnak van saját címe (itt az alsó 9 bit adja a címet)
    pl: Add, And, Jmp, Inc_Skip_IfZero, DepositClearAcc
  - kettőnek nem kell cím: pl. ACC műveletekhez





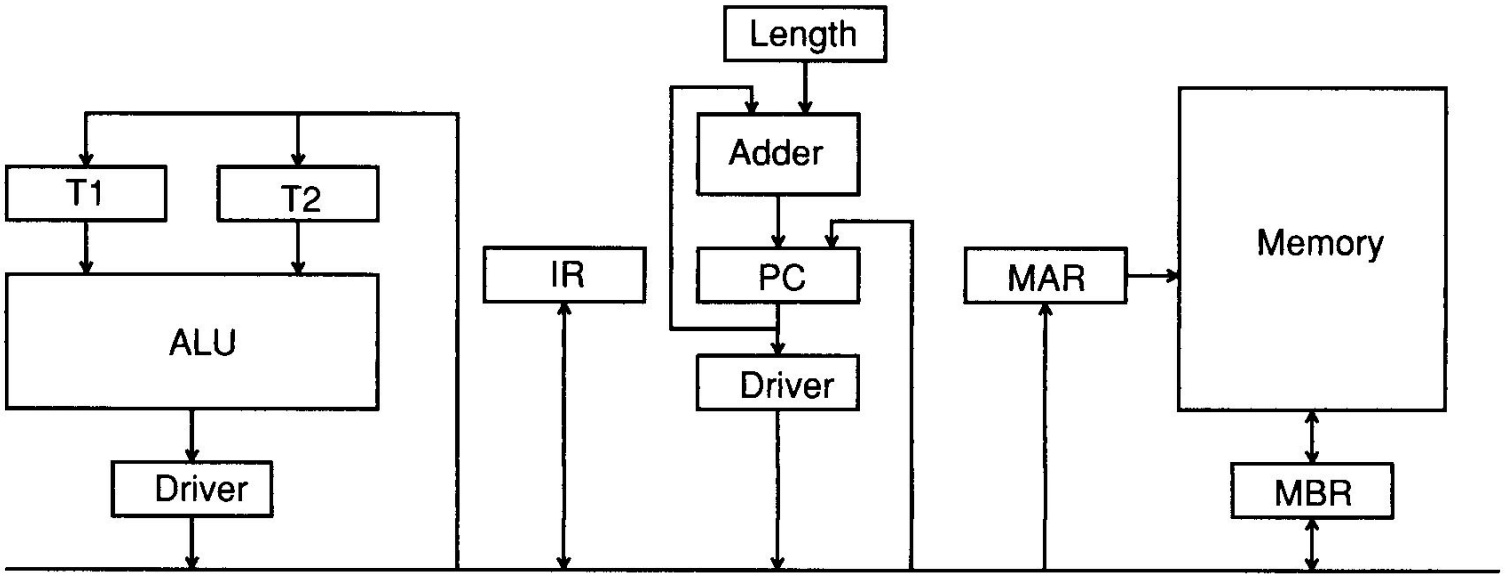
[8:3] bit -> 64 I/O eszköz

[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!

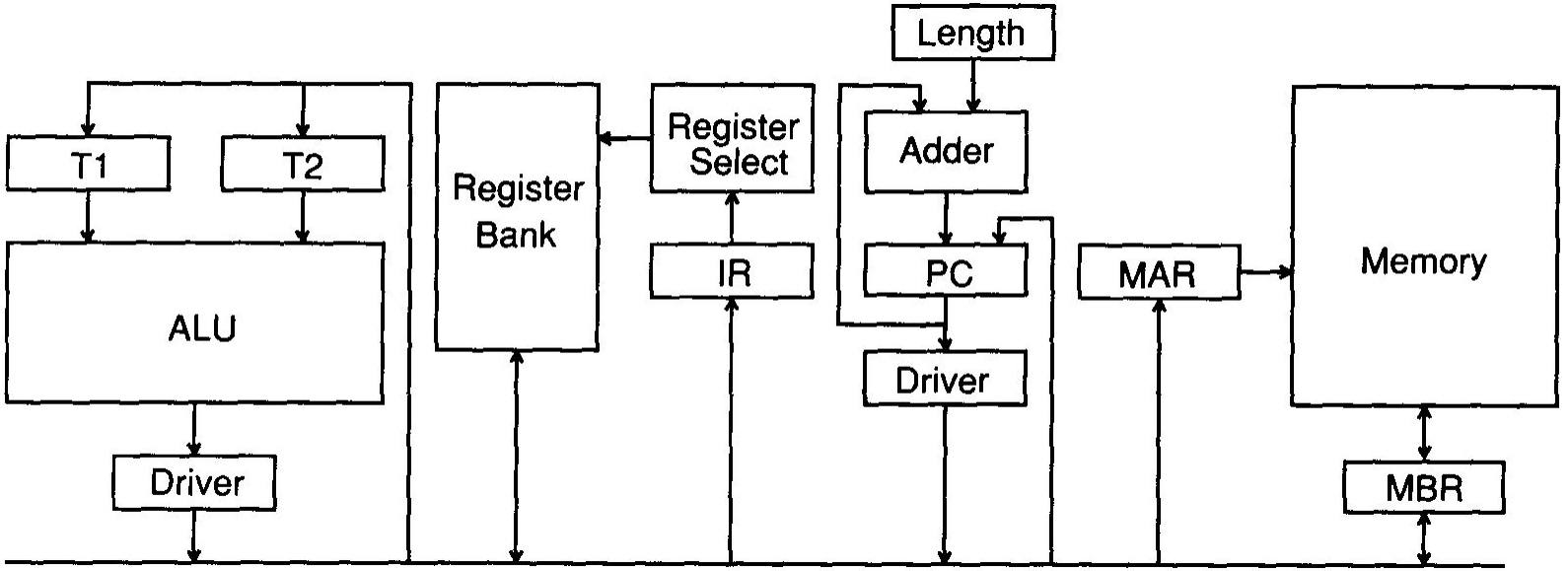
Komplex műveletek: ADD3

- 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)

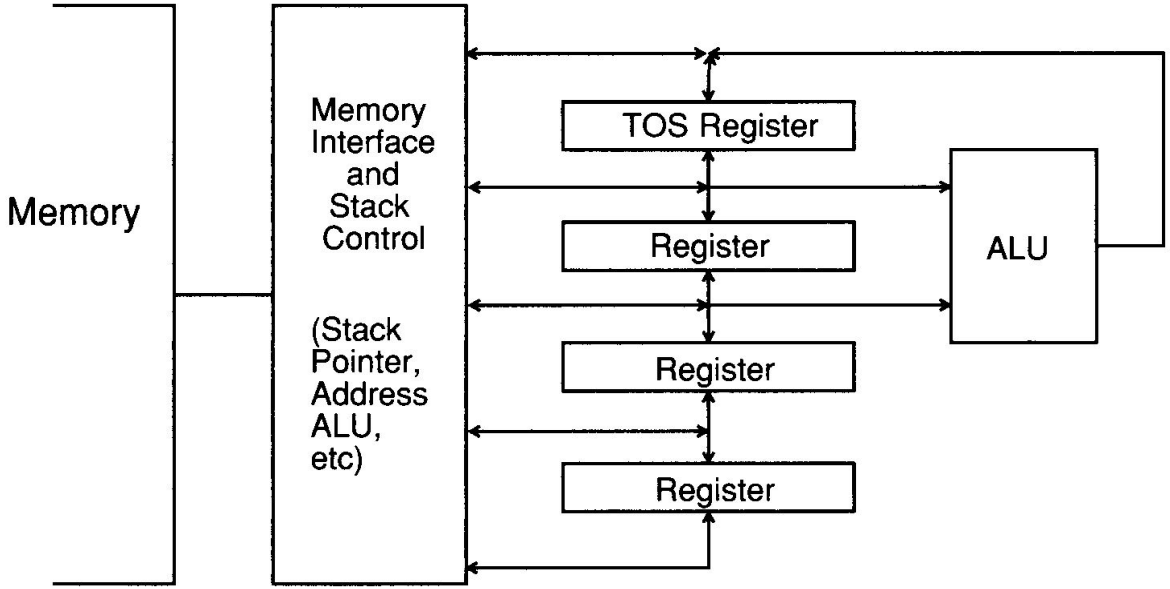
RX→T1	[5ns] **RX értékét** T1-be töltjük
RY→T2	[5ns] **RY értékét** T2-be töltjük
T1 + T2→RZ	[10+5ns] ADD2 művelet elvégzése, **RZ**-be töltjük

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

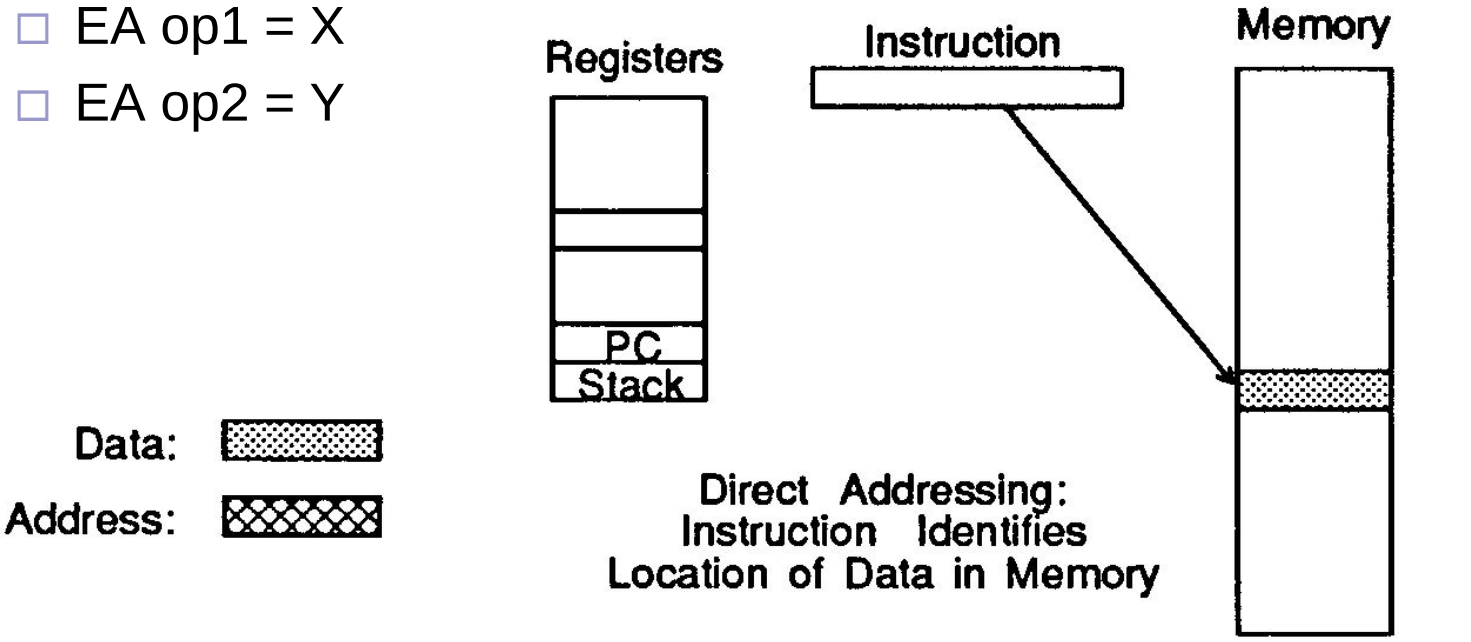
Operandus címzési módok

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

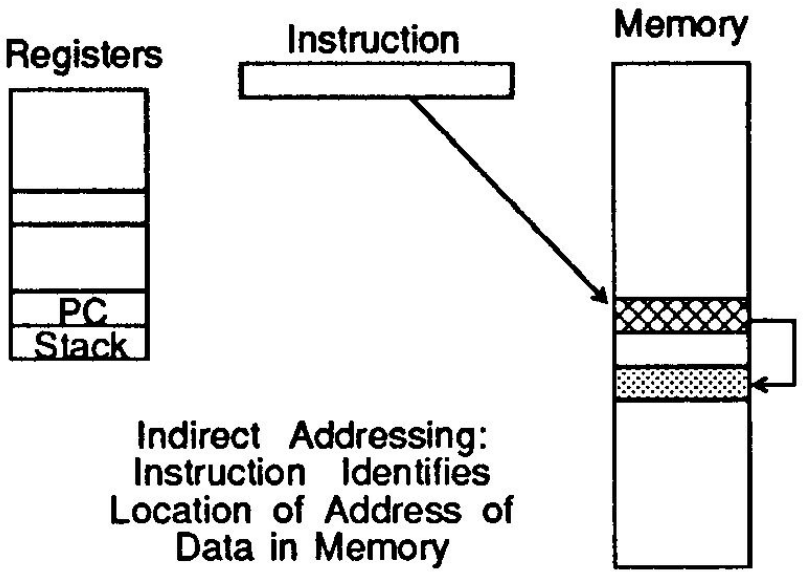
1. Direkt címzés (X)

- 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)

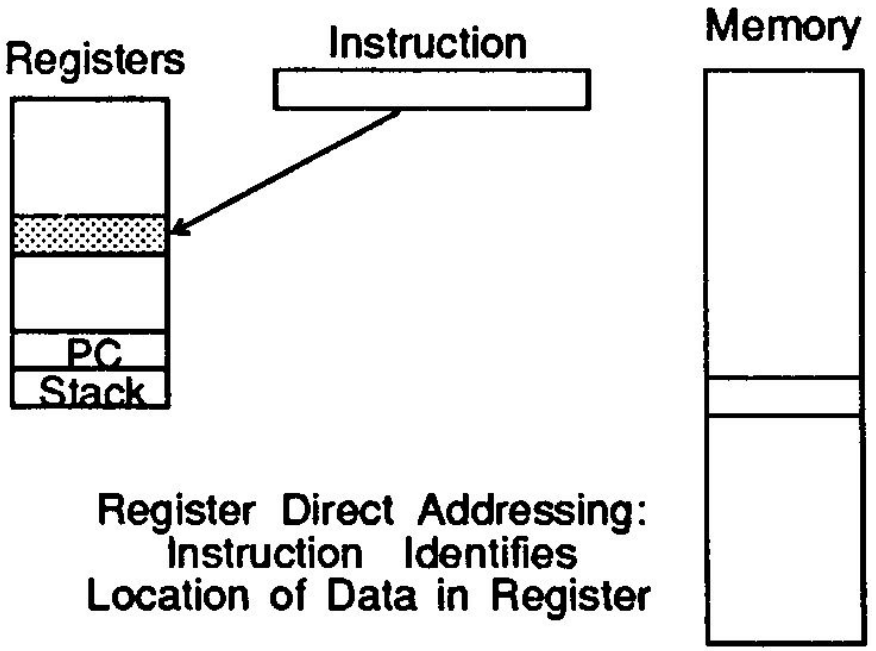
PC→MAR	[5ns] PC-vel a következő (**X**) *címének címére* mutatunk
PC+X_Alen →PC	[5ns] X operandus címének hosszával növeljük a PC-t
M[MAR]→MBR	[30ns] *X* címének címét az MBR-be írjuk
MBR→MAR	[5ns] Ezt a címet a MAR-ba töltjük
M[MAR]→MBR	[30ns] X *címét* megkapjuk az MBR-ben
MBR→MAR	[5ns] X címét a MAR-ba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

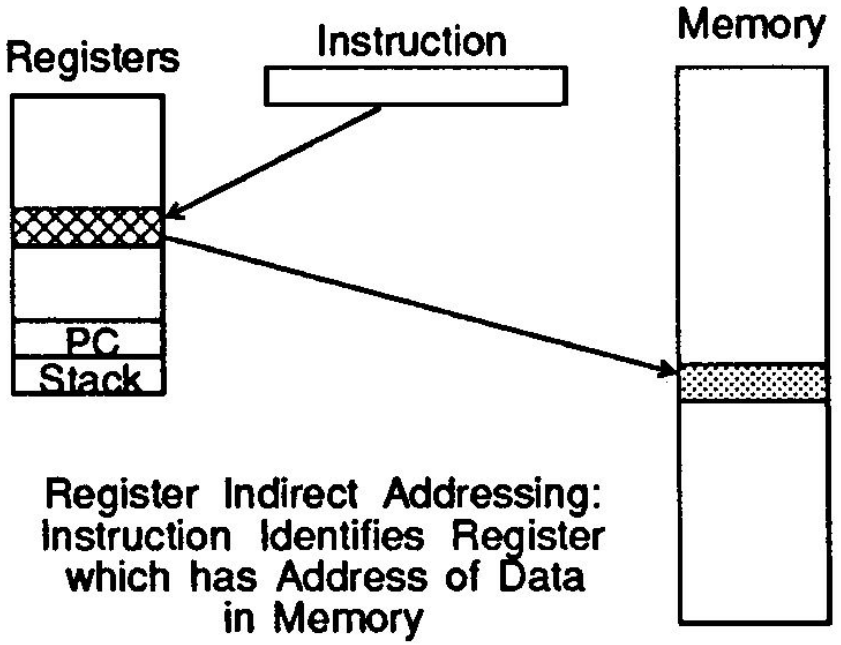
3. Regiszteres direkt címzés (R_X)

- 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)

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

Regiszteres indirekt-címzést használunk itt!

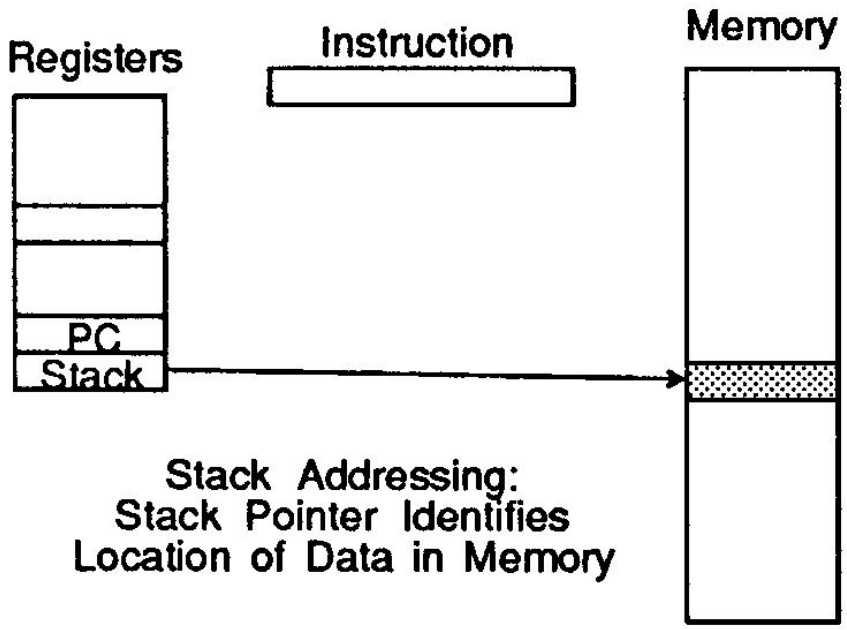
Összehasonlító táblázat – I.

- 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.

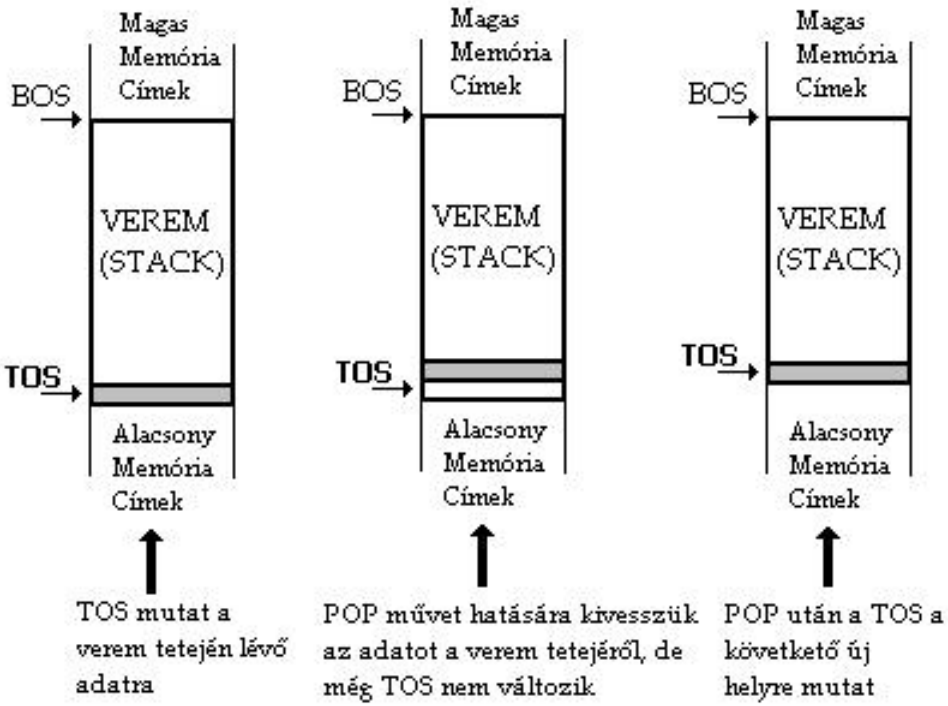


Verem – PUSH, POP műveletek

- **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**

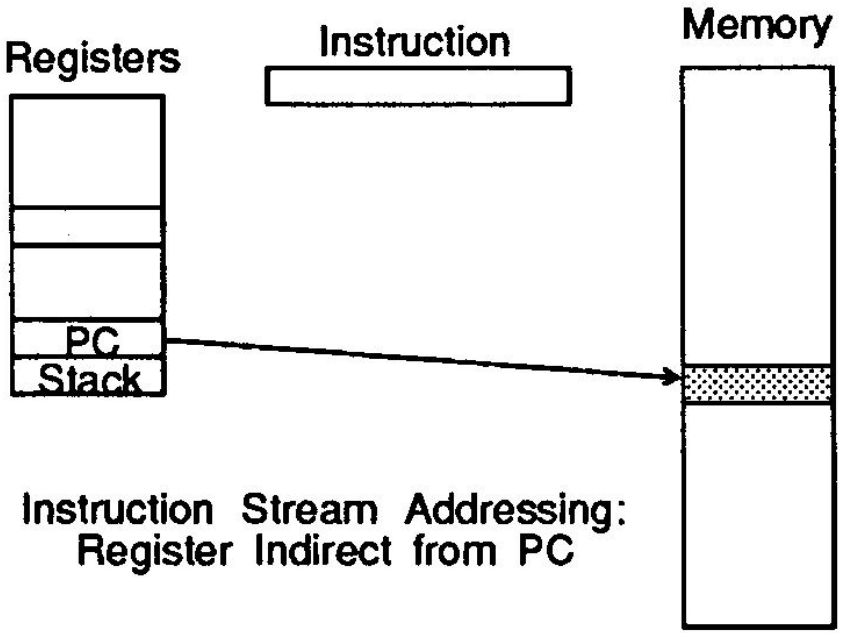
- **PUSH művelet:** a stack pointer automatikusan dekrementálja a címet, amivel a verem tetején lévő elemet azonosítja. Majd ezután berakjuk az Rx regiszterben lévő elemet a stack tetejére, a pointer átlat mutatott címre.
  Jel: **MOVE Rx, *R (stack pointer)–**

Példa: POP műveletre



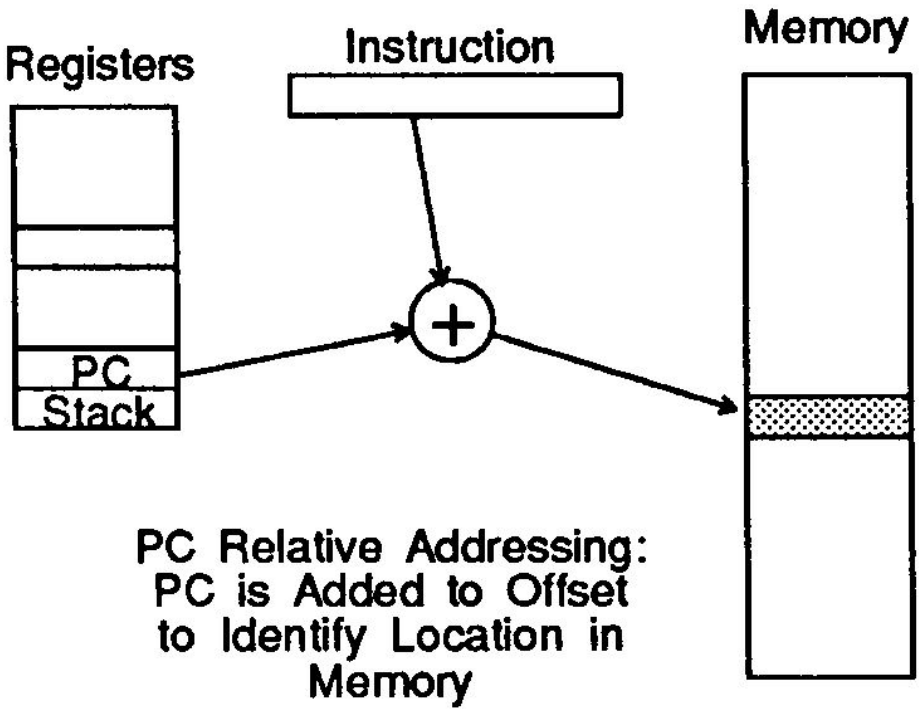
6. Instruction Stream címzés

- 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.



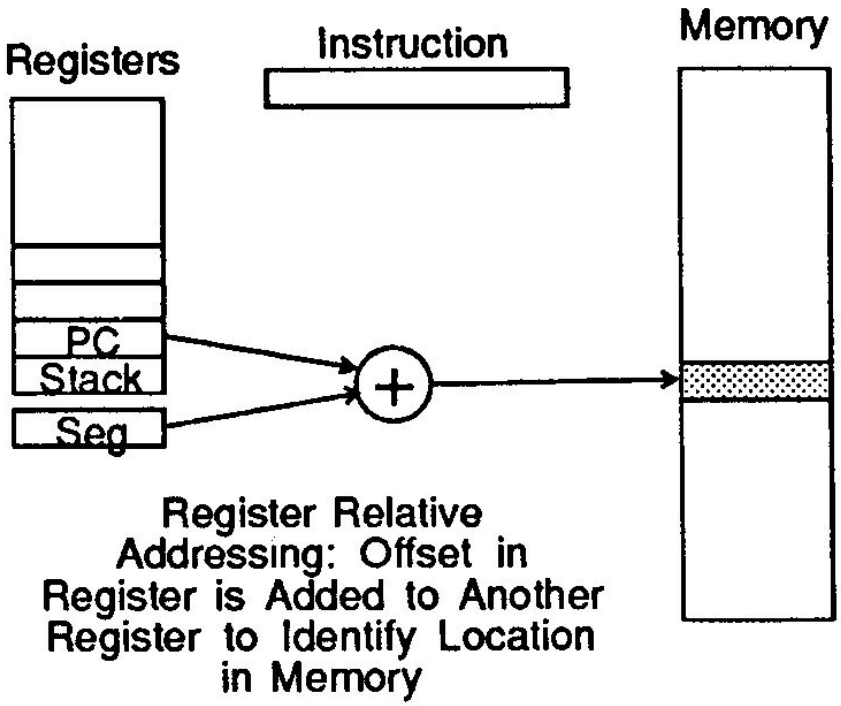
7. PC relatív címzés

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



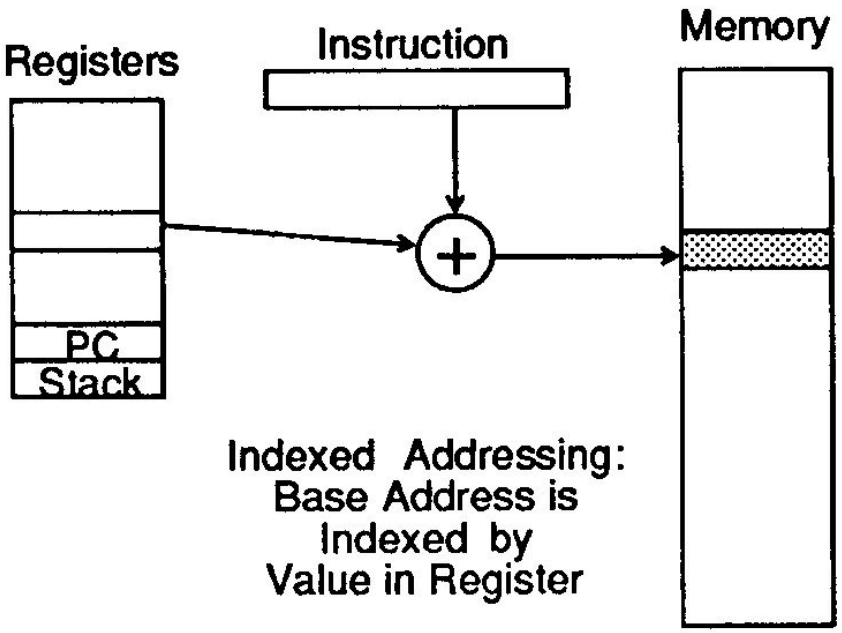
8. Regiszter relatív címzés

- 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.

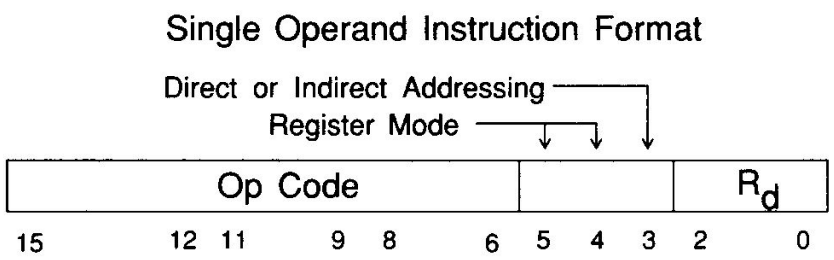
- Címzési módok és jelöléseik összefoglaló táblázata

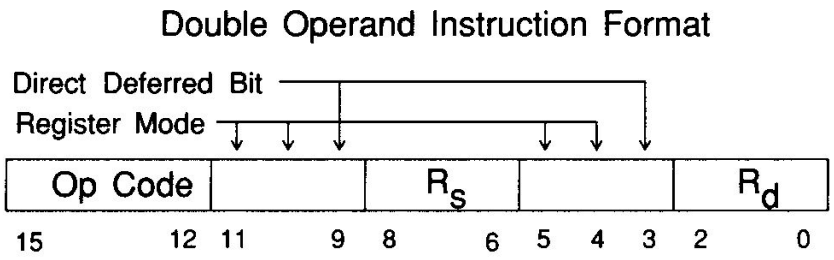
Table 4.1. Addressing Modes and their Nomenclatures.

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

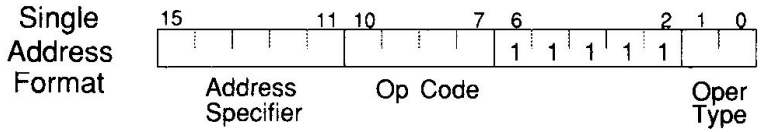
Addr bits		*Addressing mode*
0	0 0	Register direct
0	0 1	Register indirect
0	1 0	Register indirect — autoincrement
0	1 1	Two level indirect, autoincrement register
1	0 0	Register indirect — autodecrement
1	0 1	Two level indirect, autodecrement register
1	1 0	Indexed
1	1 1	Indexed indirect

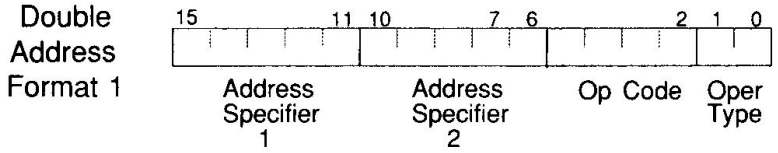
Addressing modes when PC is target register

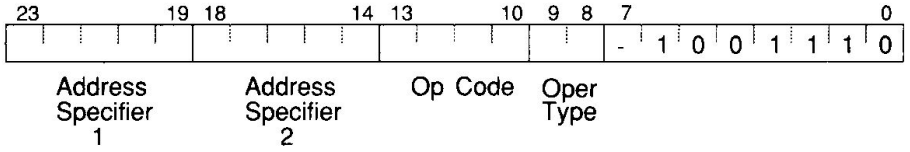
Addr bits		*Addressing mode*
0	1 0	Immediate mode (Instruction stream)
0	1 1	PC absolute mode
1	1 0	PC relative
1	1 1	PC relative, indirect

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.

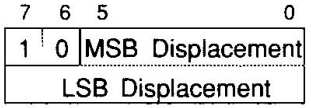




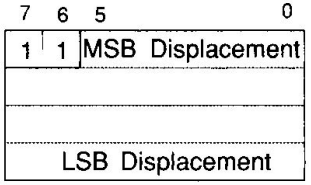


Double Address Format 2

One Byte Displacement; Range: -64 to 63



Two Byte Displacment; Range: -8192 to 8191



Four Byte Displacement; Range:
as an Address: Entire Address Space;
as a value: +/- 2^{29}-1

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):

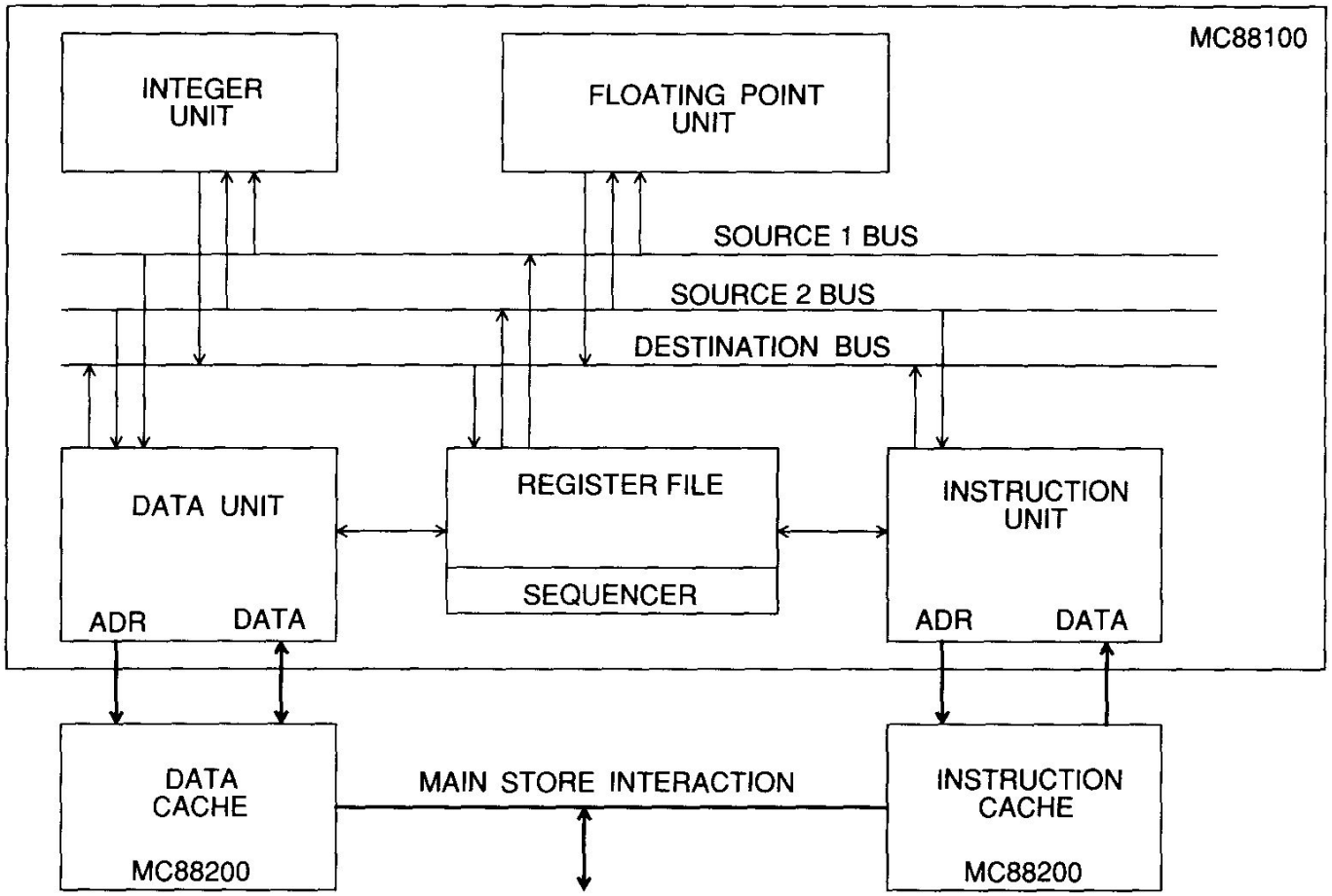
- *Huzalozott (hardwired) utasításdekódolás* (a hardveres dekódolás megvalósításához kombinációs logikát használ, azonban a mai memóriaalapú mikro-kódú gépeknél ez lassabb).
- *Egyszeres ciklusvégrehajtás*: (minden egyes ciklusban egy utasítást hajt végre, ha ezt sikerülne elérni optimális lenne az erőforrás 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

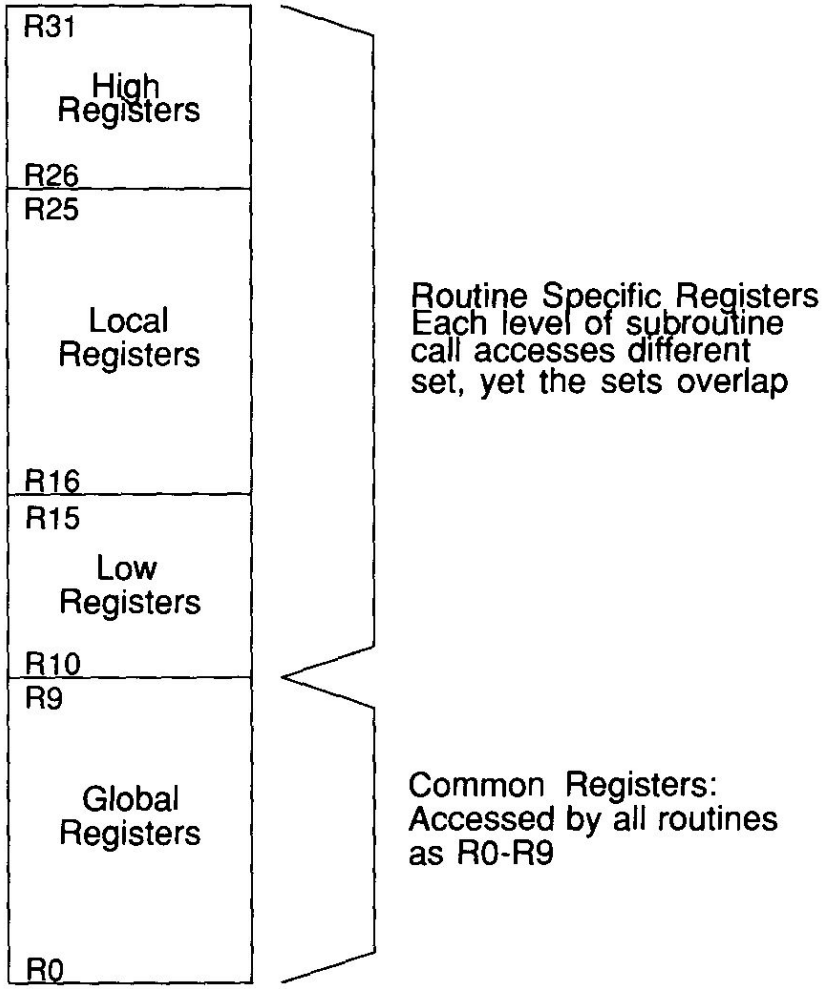


Hardveres párhuzamosítás:

- Buszok
- Cache
- Data / Instruction Unit

Berkeley RISC-I rendszere: Regiszter használat

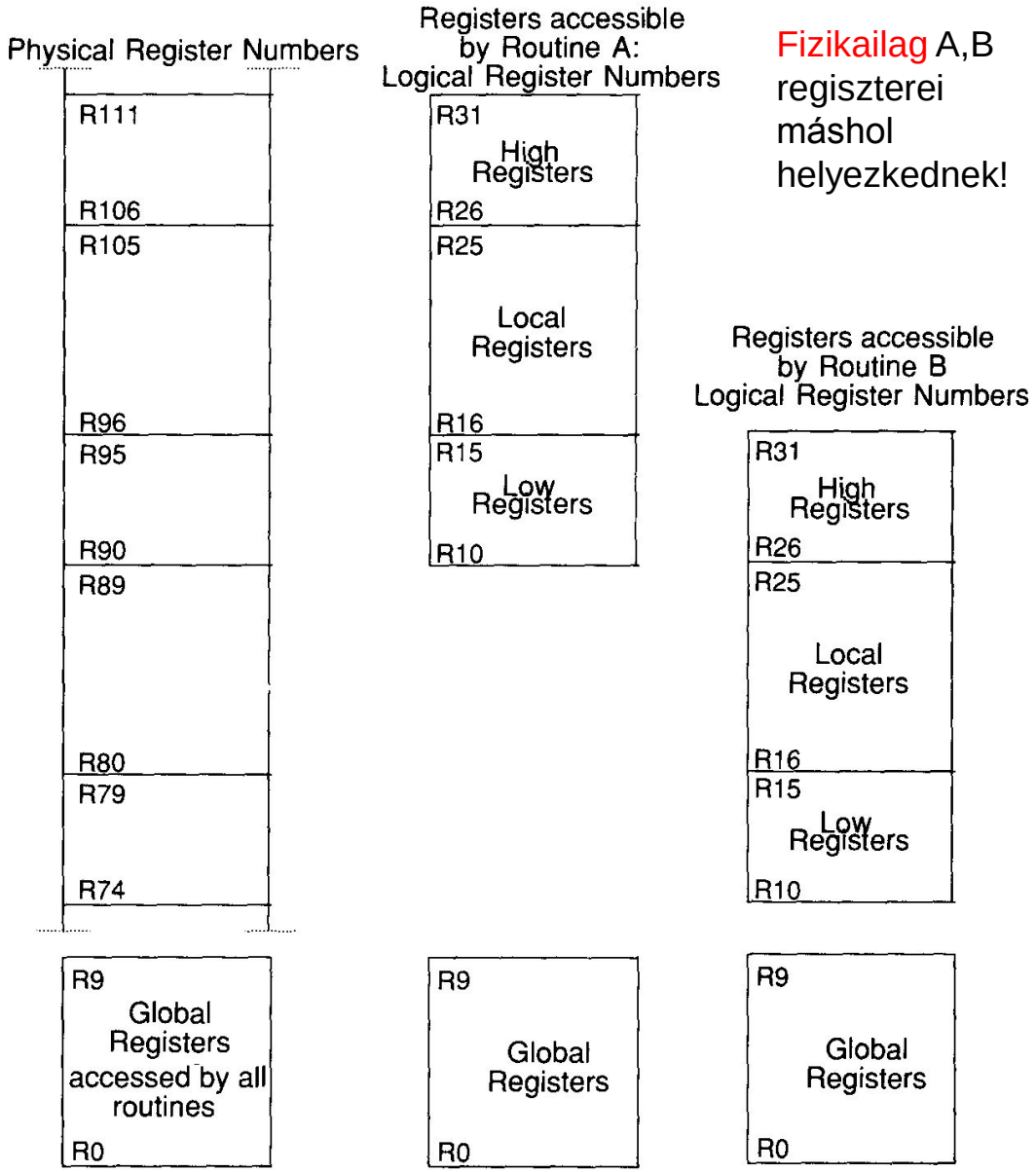
- Regiszter ablak: a szubrutin híváshoz (call) / visszatéréshez (ret) szükséges processzor-időt kívánták minimalizálni nagy számú regiszter használatával.
- Regisztereknek csak egy kis része érhető el („ablak”). Egy pointer mutat az ablakra, amely azonosítja a benne található aktuális regisztereket.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



Fizikailag A,B regiszterei máshol helyezkednek!

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