PANNON EGYETEM, Veszprém
Villamosmérnöki és Információs Rendszerek Tanszék

Digitális Rendszerek és Számítógép Architektúrák

3. előadás: Aritmetikai egységek - adatkezelés

Előadó: Dr. Vörösházi Zsolt
voroshazi.zsolt@virt.uni-pannon.hu

Jegyzetek, segédanyagok:

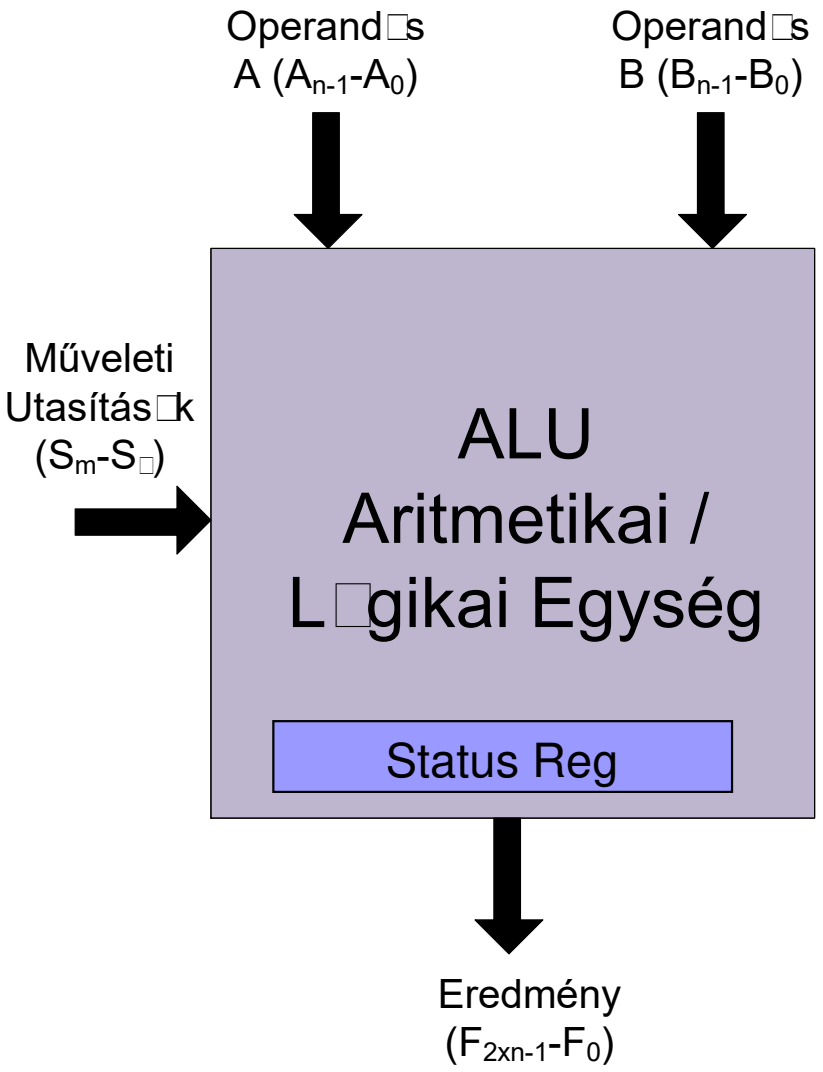
- 🕮 Könyvfejezetek:
  - http://www.virt.uni-pannon.hu
    → Oktatás → Tantárgyak → Digitális Rendszerek és Számítógép Architektúrák (nappali)
    (chapter03.pdf)
- Fóliák, óravázlatok .ppt (.pdf)
- Frissítésük folyamatosan

Ismétlés

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

ALU felépítése

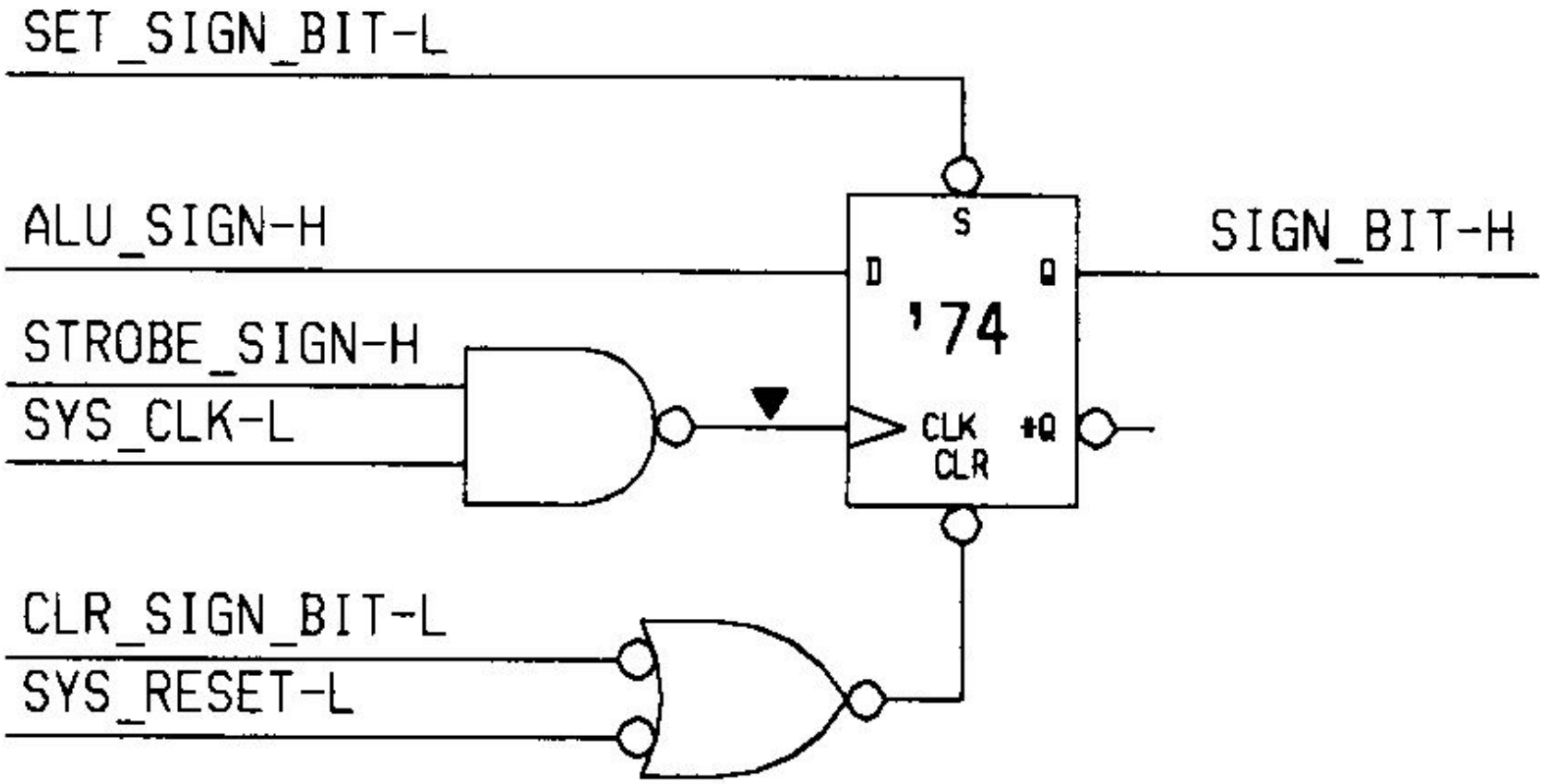
- Utasítások hatására a (S_m-S_0) „vezérlőjelek” jelölik ki a végrehajtandó aritmetikai / logikai műveletet. További adatvonalak kapcsolódhatnak közvetlenül a státusz regiszterhez, amelyben fontos információkat tárolunk: pl.
  - *zero bit*
  - *carry-in, carry-out* átviteleket,
  - *előjel* bitet (sign),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



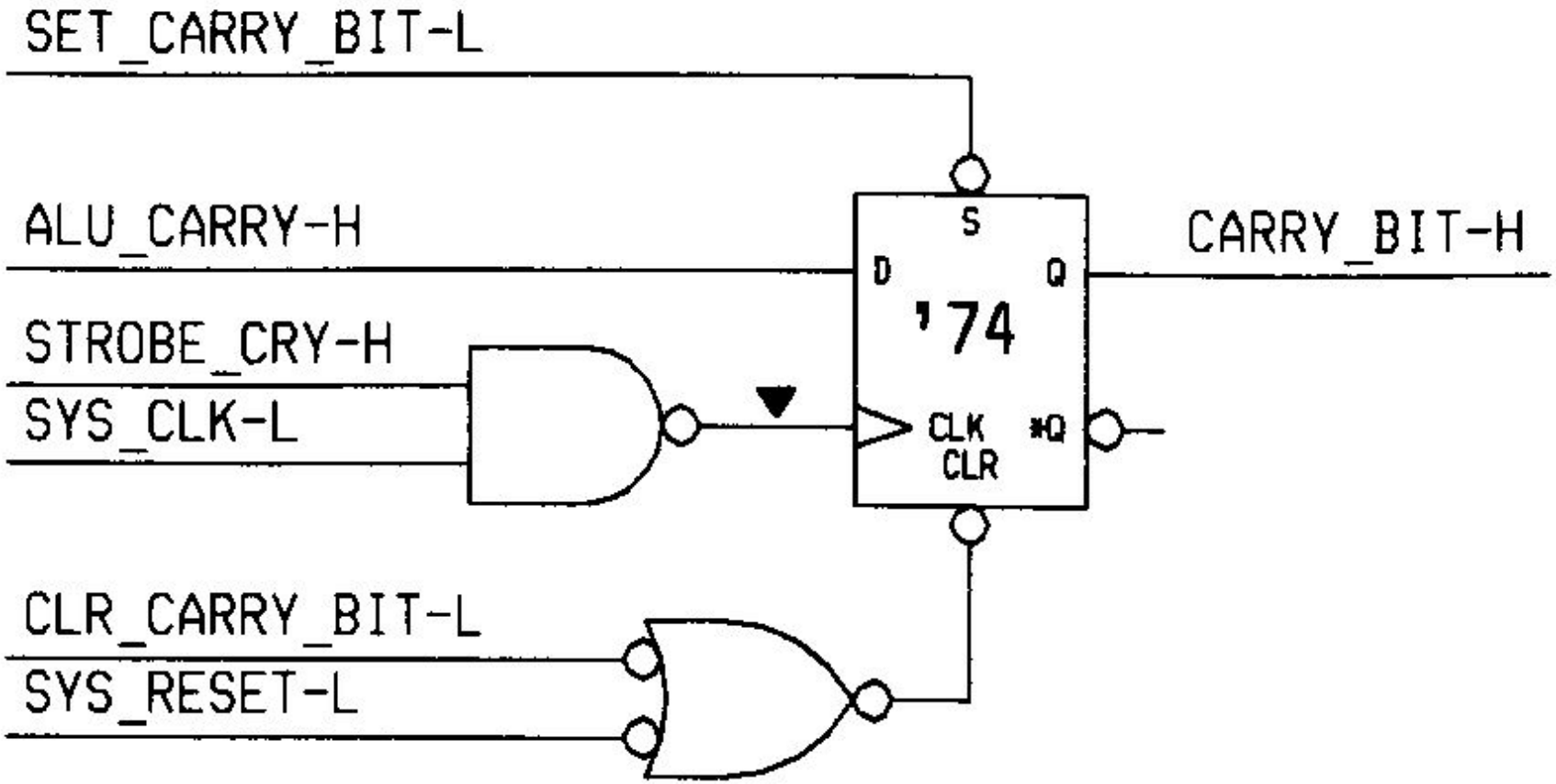
Státusz- (flag) jelzőbitek

- Az aritmetikai műveletek eredményétől függően hibajelzésre használatos jelzőbitek. Ezek megváltozása az utasításkészletben előre definiált utasítások végrehajtásától függ.
  - a.) Előjelbit (sign): 2's 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?
    - Pl: 0-val való osztás!
    - (szorzásnál egyszerűsíthetőség – adatfüggés)
  - e.) Paritás bit: páros, páratlan
  - …

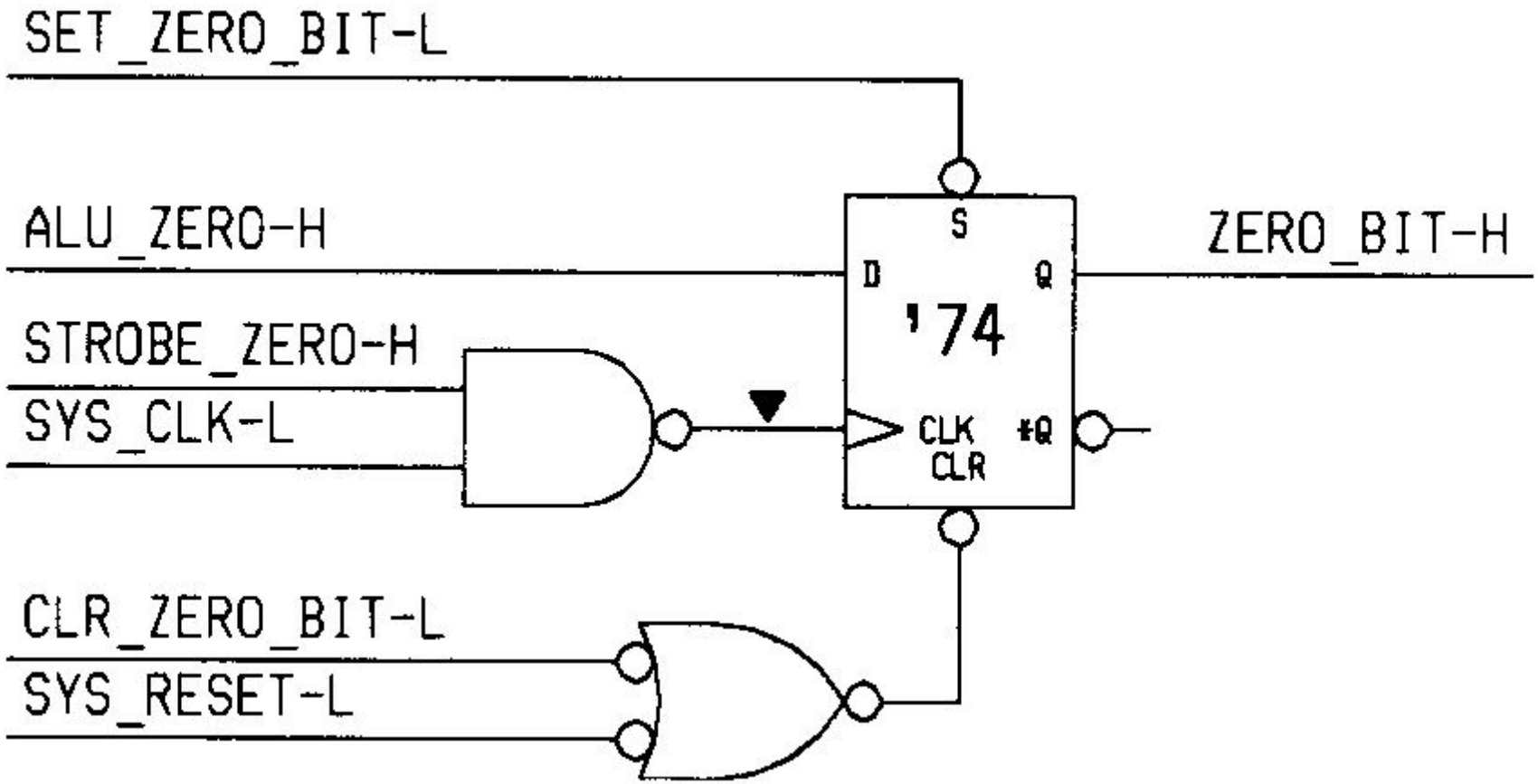
a.) Előjelbit (sign)



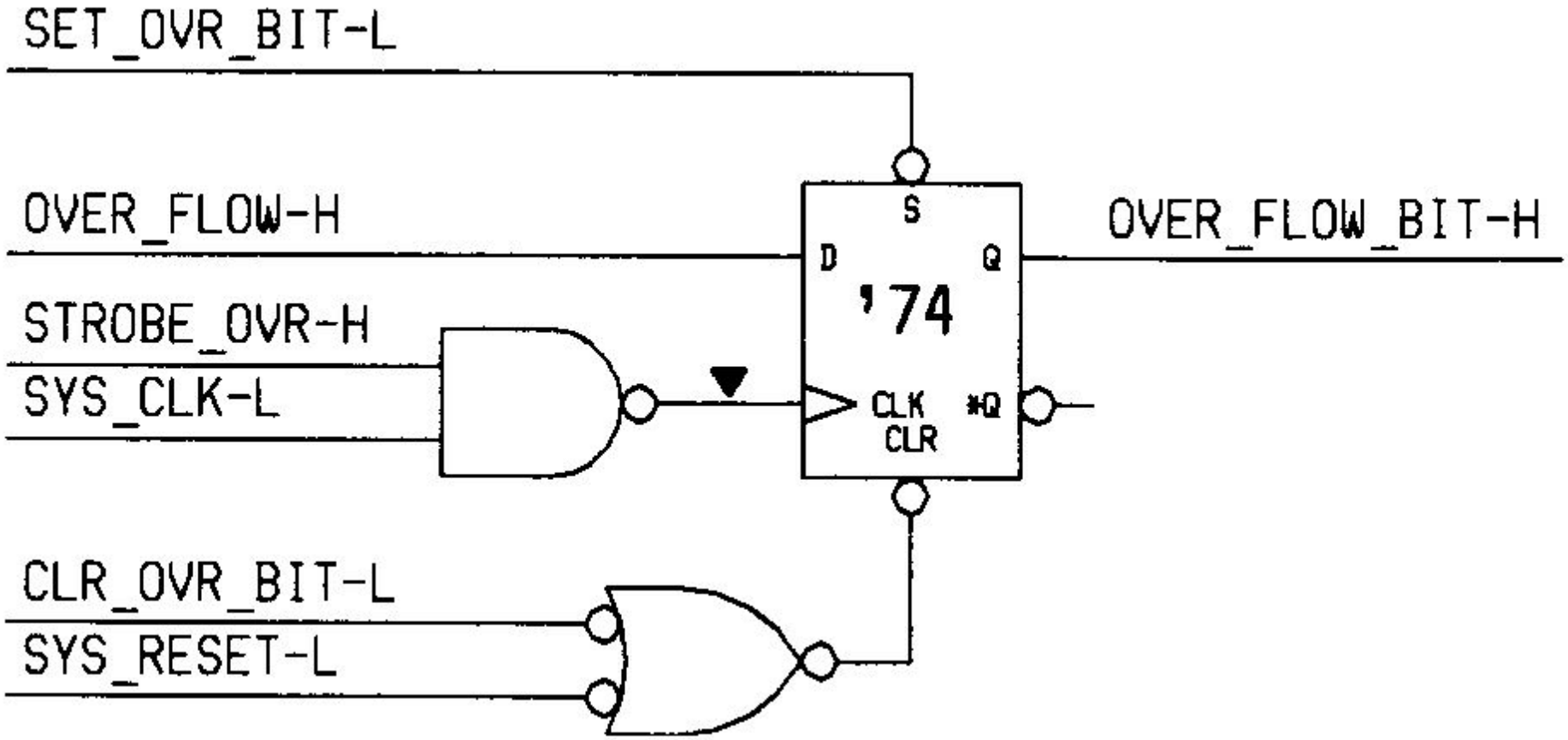
b.) Átvitel-kezelő bit (carry)



c.) Zéró bit

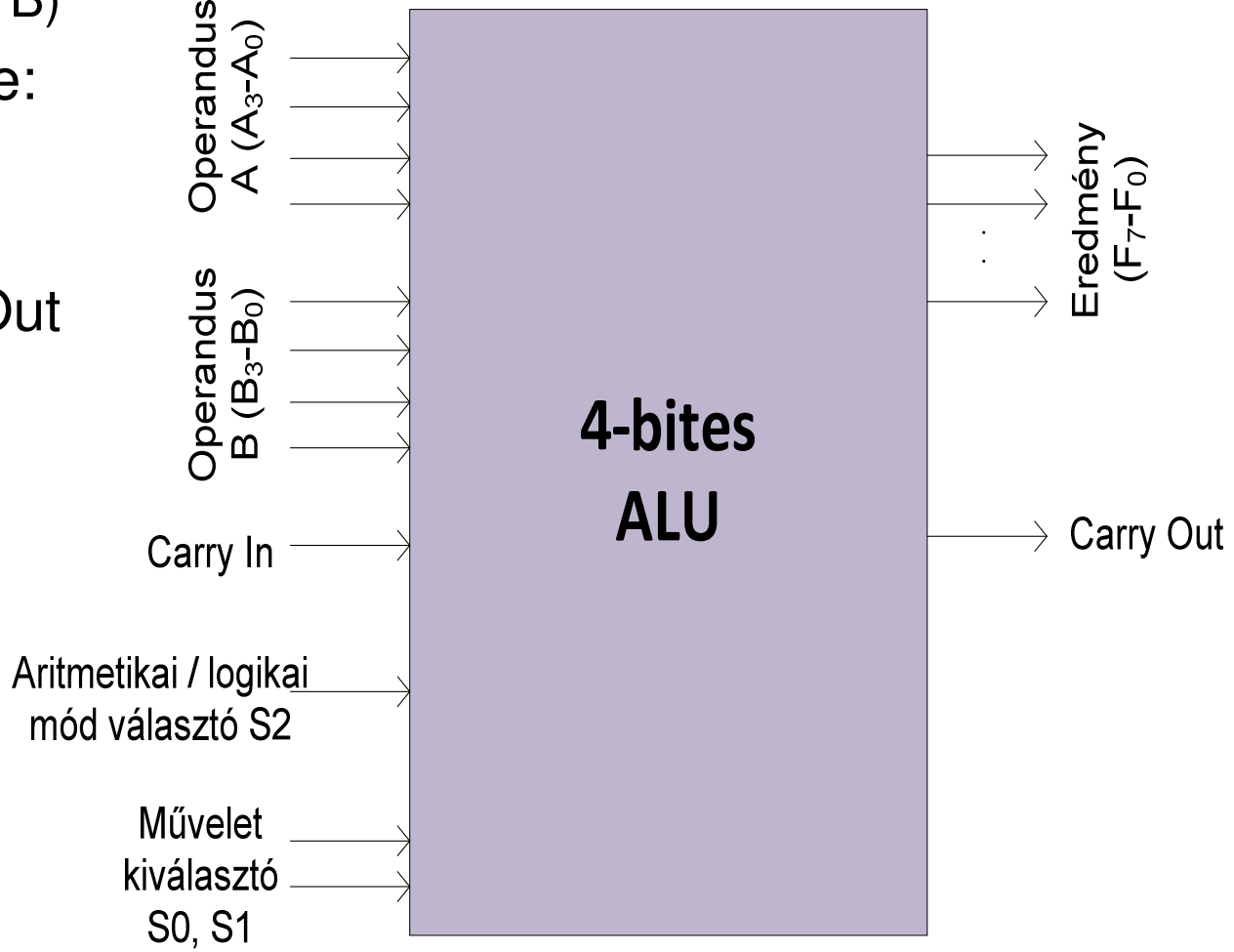

SET_ZERO_BIT-L
ALU_ZERO-H
STROBE_ZERO-H
SYS_CLK-L
CLR_ZERO_BIT-L
SYS_RESET-L
S
D
Q
'74
CLK
CLR
+Q
ZERO_BIT-H

d.) Túlcsordulást jelző bit (overflow)



Pl: N=4-bites ALU felépítése és működése

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



4-bites ALU szimbolikus rajza

IC: ‘181 ALU 4-bites (74LS181)

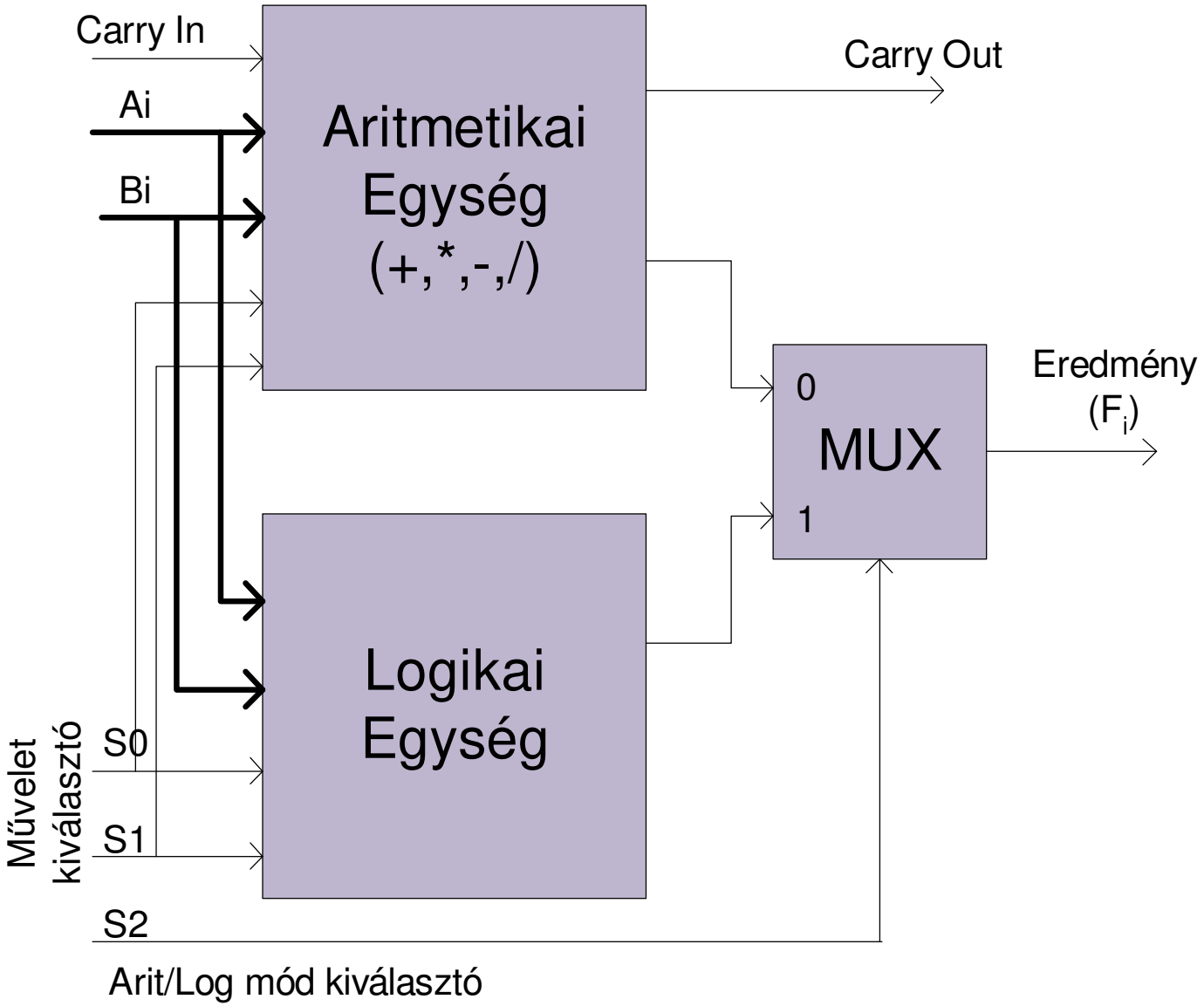
ALU működését leíró függvénytáblázat

(egy lehetséges működés, a funkciók bővíthetők):

Művelet kiválasztás:				Művelet:	Megvalósított függvény:
S2	S1	S0	Cin		
0	0	0	0	F=A	'A' átvitele
0	0	0	1	F=A+1	'A' értékének növelése 1-el (increment)
0	0	1	0	F=A+B	Összeadás
0	0	1	1	F=A+B+1	Összeadás carry figyelembevételével
0	1	0	0	$F = A + \overline{B}$	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=B	'B' átvitele
1	0	0	x	$F = A \wedge B$	AND
1	0	1	x	$F = A \vee B$	OR
1	1	0	x	$F = A \oplus B$	XOR
1	1	1	x	$F = \overline{A}$	'A' negáltja (NOT A)

* Forrás: Könyv függeléke (appendix)

ALU felépítése:



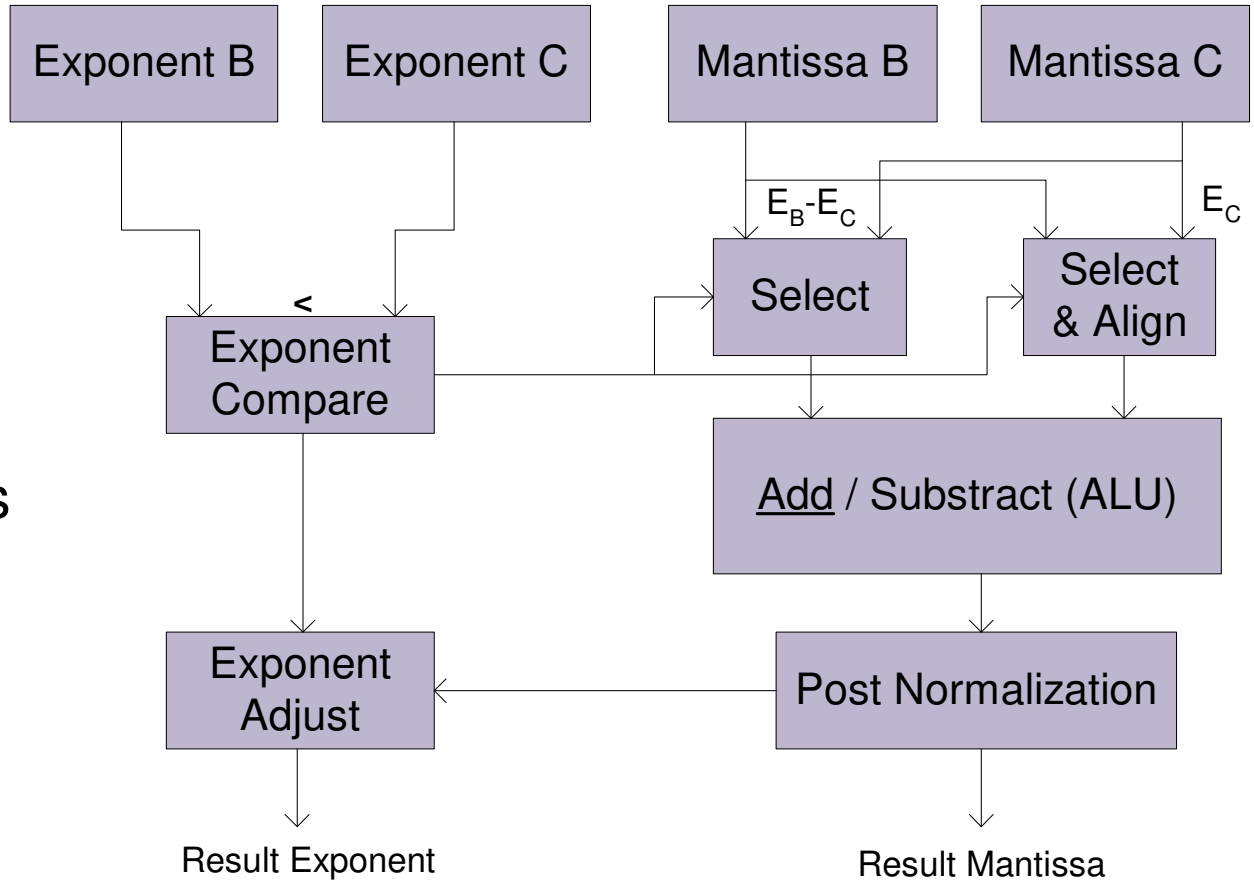
Lebegőpontos műveletvégző egységek

Lebegőpontos műveletvégző egységek

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

a.) Lebegőpontos összeadó

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



FPN: összeadás (kivonás)

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

- A = B + C = 10.0001 + 1101.1 = //B < C//

(1.00001 × 2^**1**) + (1.1011 × 2^**3**) =

(0.0100001 × 2^**3**) + (1.1011 × 2^**3**) =

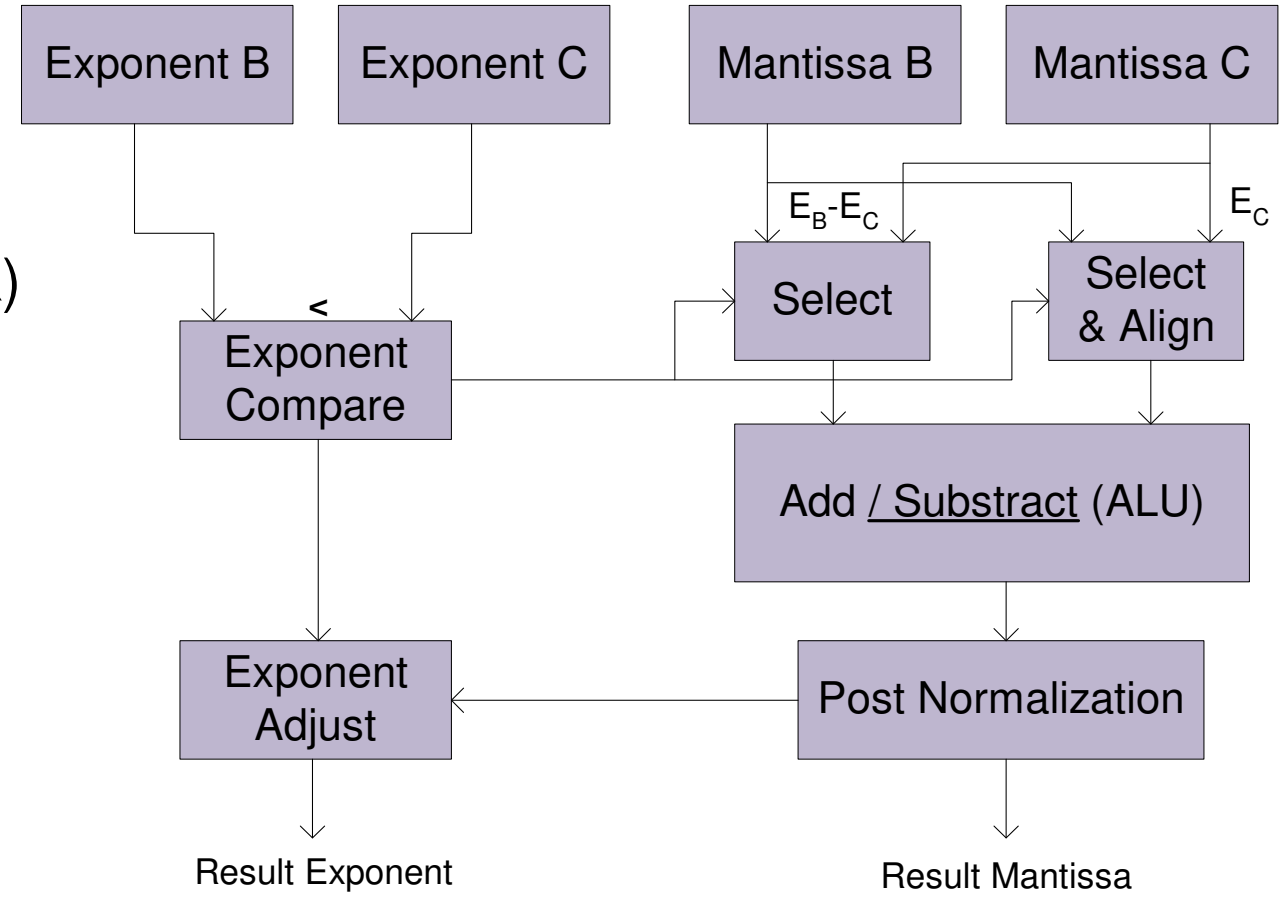
(0.0100001 + 1.1011) × 2^3 =

- 1.1111001 × 2^3 = A

  (itt éppen nem kell post-normalizálás)

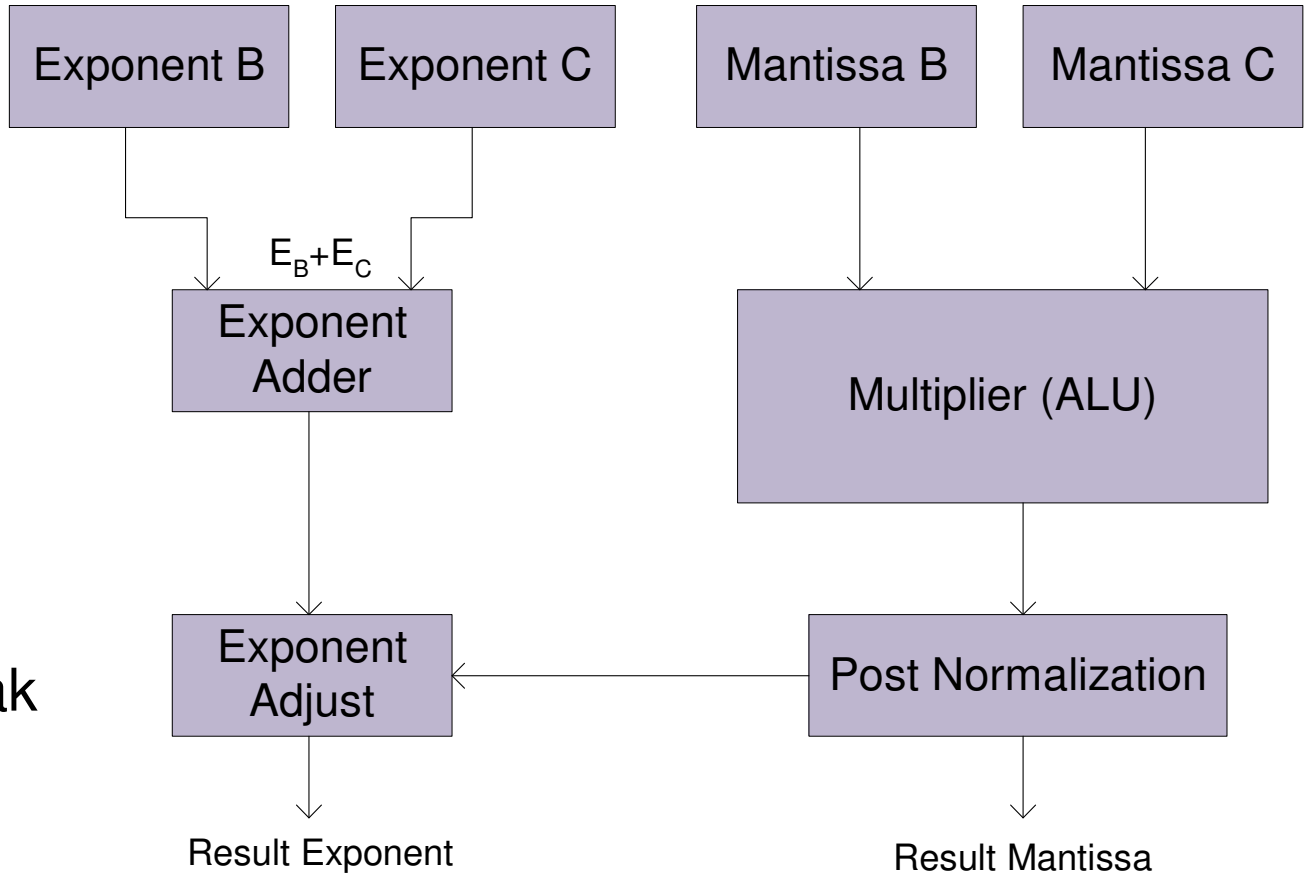
b.) Lebegőpontos kivonó

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



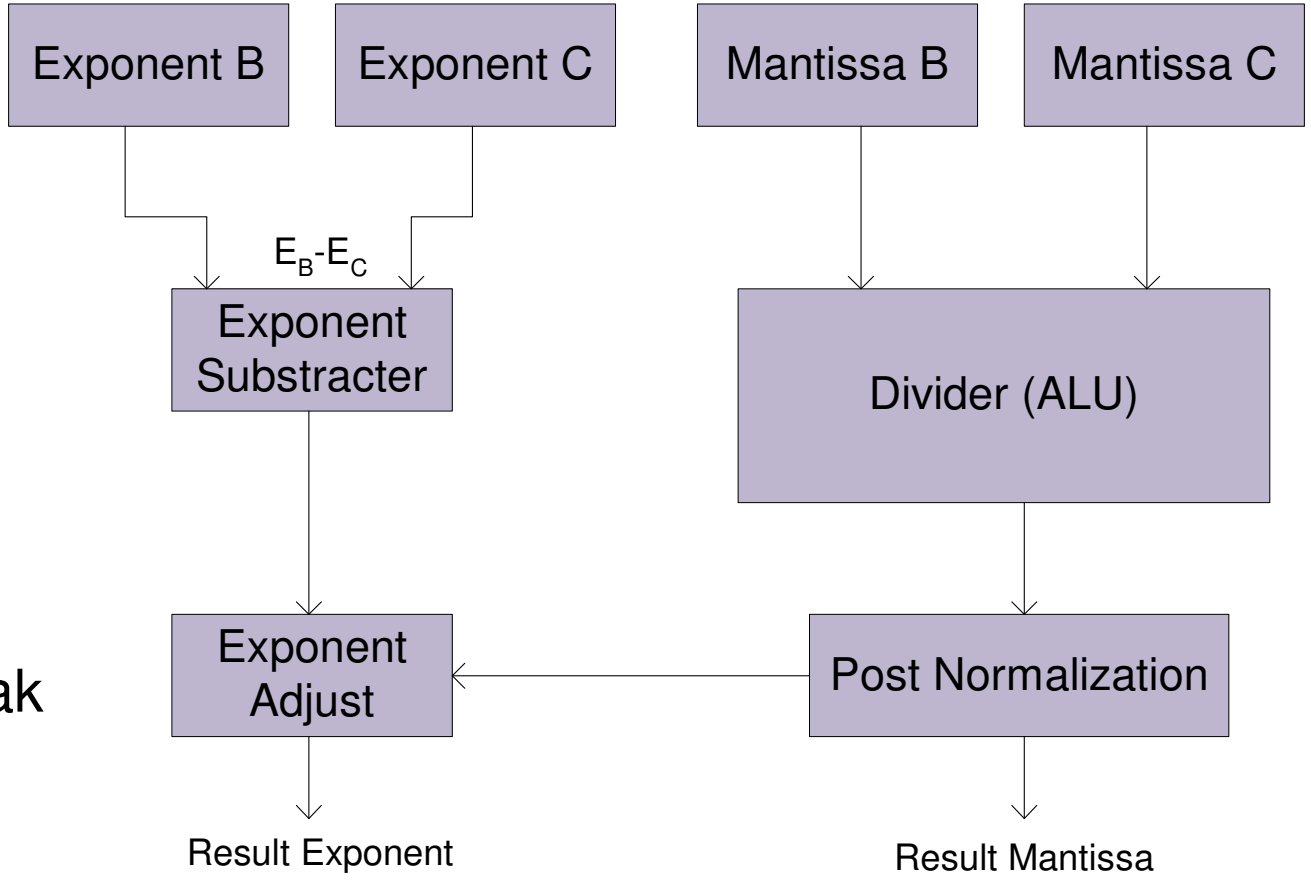
c.) Lebegőpontos szorzó

- Művelet: $A = B \times C = M_B \times r^{E_B} \times M_C \times r^{E_C} = (M_B \times M_C) \times r^{E_B + E_C}$
  - A: szorzat
  - B: szorzandó
  - C: szorzó
  - Könnyű végrehajtani
  - Nincs szükség az operandusok beállítására
  - Minimális post-normalizációt kell csak végezni
  - ALU: szorzás!



d.) Lebegőpontos osztó

- Művelet: $A = B/C = M_B \times r^{E_B} / M_C \times r^{E_C} = (M_B / M_C) \times r^{E_B - E_C}$
  - A: hányados
  - B: osztandó
  - C: osztó
  - Könnyű végrehajtani
  - Nincs szükség az operandusok beállítására
  - Minimális post-normalizációt kell csak végezni
  - Osztás! (ALU)



Összeadó / Kivonó áramkörök

a.) Fél-összeadó – Half Adder

- 1-bites Half Adder

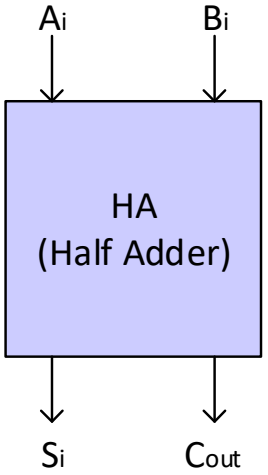
T_{HA} = 1G

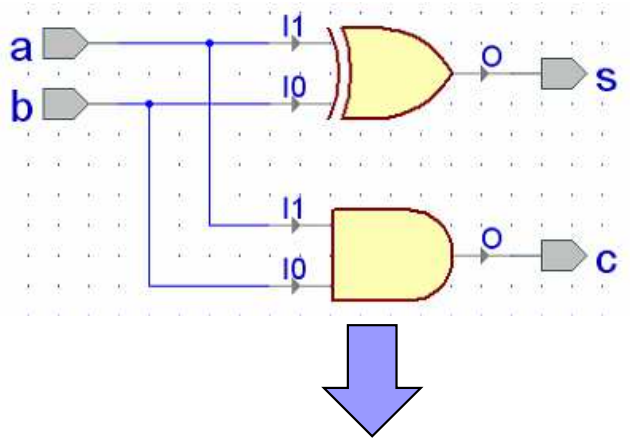
igazságtáblázat

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

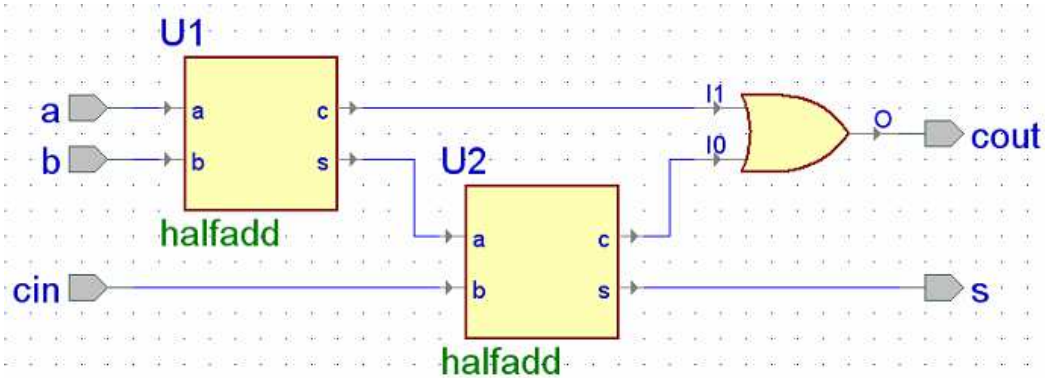
Nem kezeli a Cin-t !

szimbólum

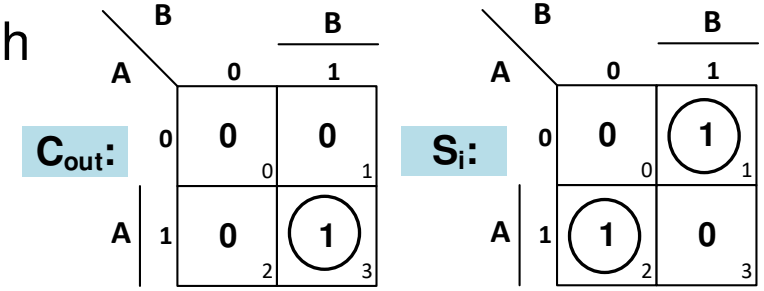




1 bites FA felépítése 2 db HA segítségével:



Karnaugh táblái:



Kimeneti fgv-ei:

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

$$S_i = A_i \oplus B_i$$

b.) Teljes összeadó – Full Adder

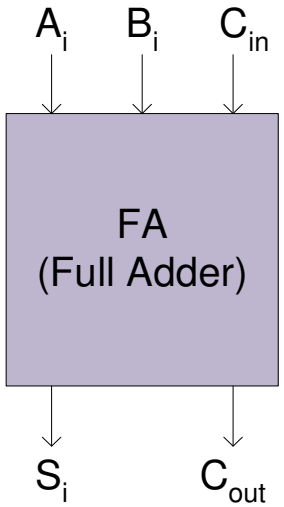
- FA: 1-bites Full Adder

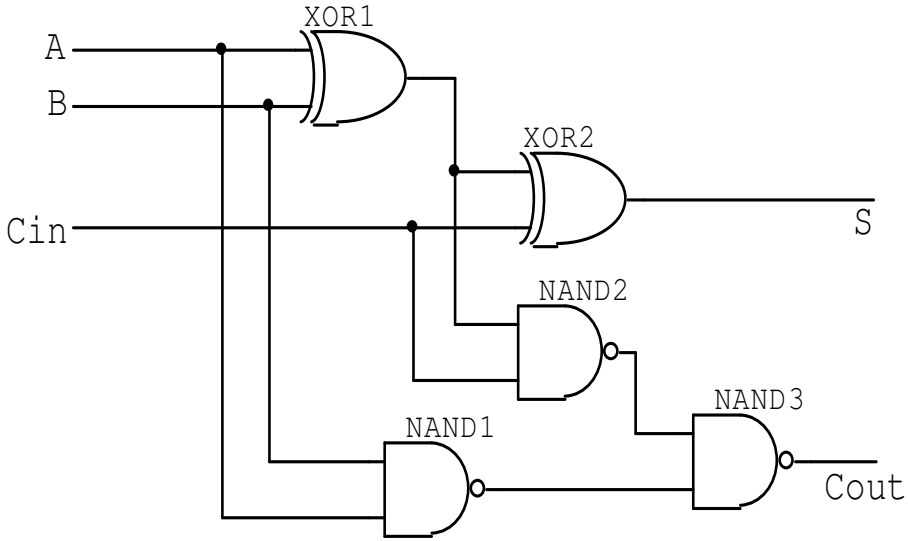
Ez a FA egy lehetséges CMOS kapcsolási rajza: (itt T_{FA} = 3G !)

igazságtáblázat

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

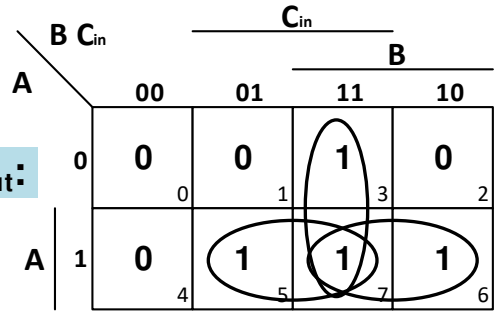
szimbólum



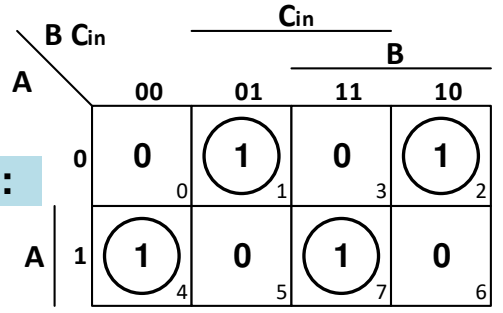


Karnaugh táblái:

C_{out}:



S_i:



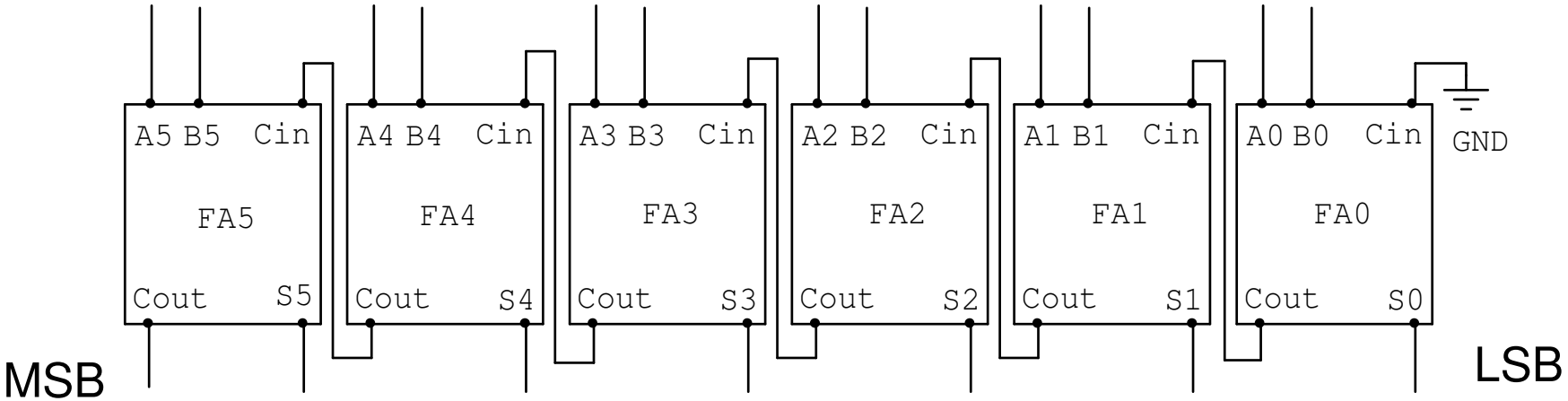
Kimeneti fgv-ei:

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

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

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

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



- Számítási időszükséglet (RCA):

$T_{(RCA)} = N*T_{(FA)} = N*(2*G) = 12\ G$ (6-bites RCA esetén)
ahol a min. 2G az 1-bites FA kapukésleltetése ([ns], [ps])

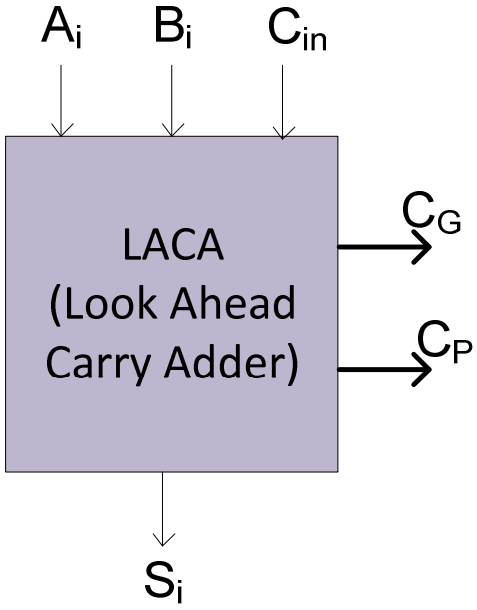
d.) LACA: Look Ahead Carry Adder:

- Képlet (FA) átalakításából kapjuk:

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

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

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



LACG: Look Ahead Carry Generator egy **b** bites ALU-hoz kapcsolódik, mindenegyes állapotban a Cin generálásáért felel a CP és CG (LACA-tól) érkező jeleknek megfelelően („*LACG looks at CP and CG from adders*”).
N-bites LACA számítási időszükséglete: $T_{LACA} = 2 + 4 \times (\lceil \log_b(N) \rceil - 1)$

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

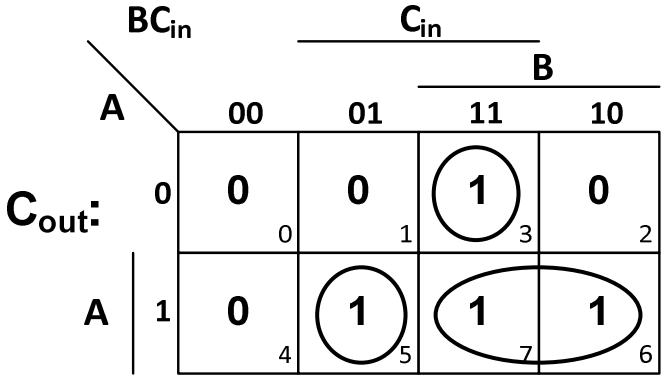
Megjegyzés: LACA – CG átírása XOR kapcsolatra (nem triviális forma)

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

igazságtáblázat

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

Karnaugh tábla:



Kimeneti fgv:

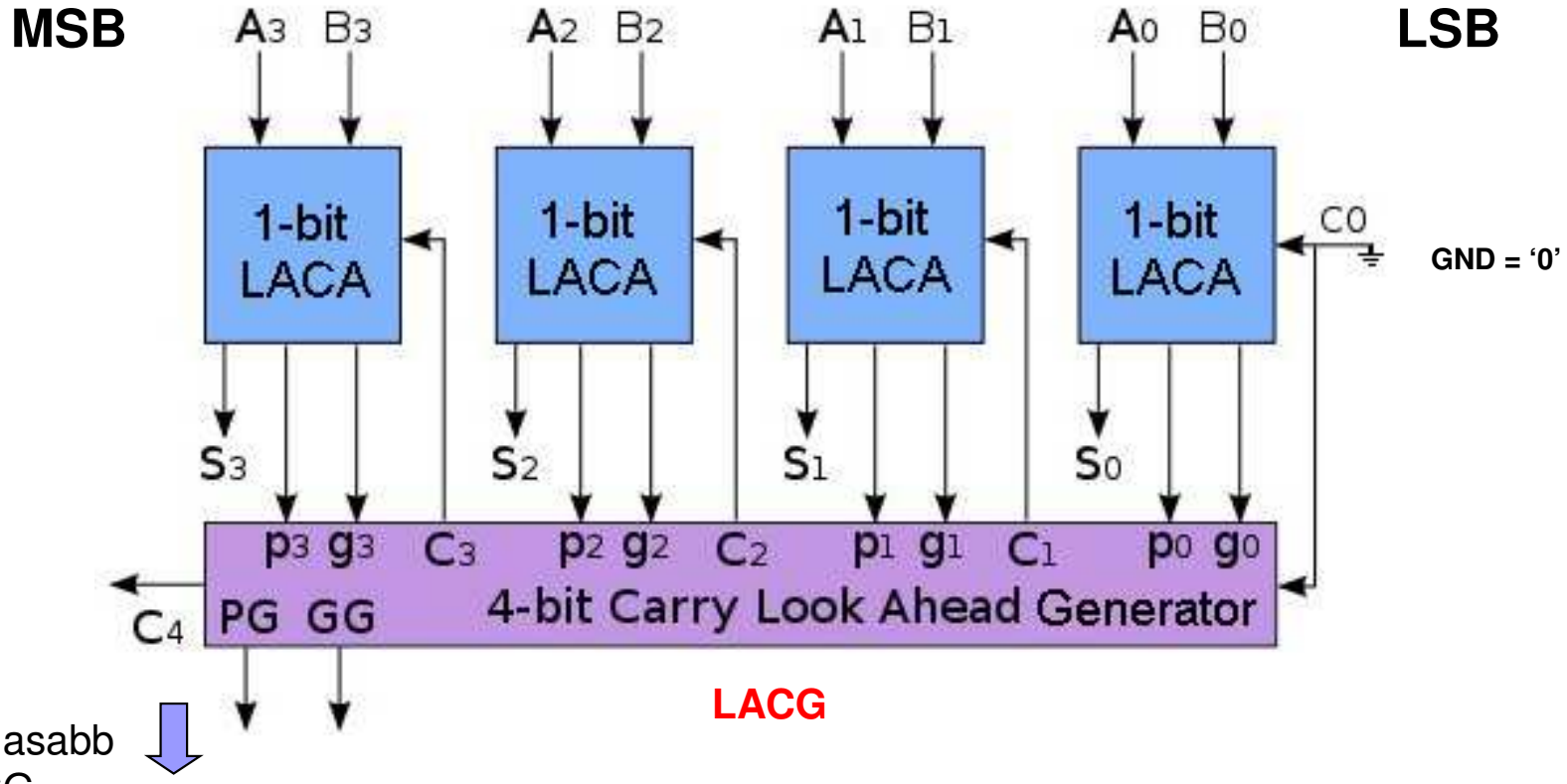
$$C_{out} = \overline{A_i} \cdot B_i \cdot C_{in} + A_i \cdot \overline{B_i} \cdot C_{in} + A_i \cdot B_i =$$

$$= A_i \cdot B_i + C_{in} \cdot (A_i \oplus B_i) =$$

$$= C_G + C_{in} \cdot C_P$$

Példa: 4-bites LACA

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



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

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

- LSB → MSB felé az összeadások

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

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

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

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

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

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

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

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

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

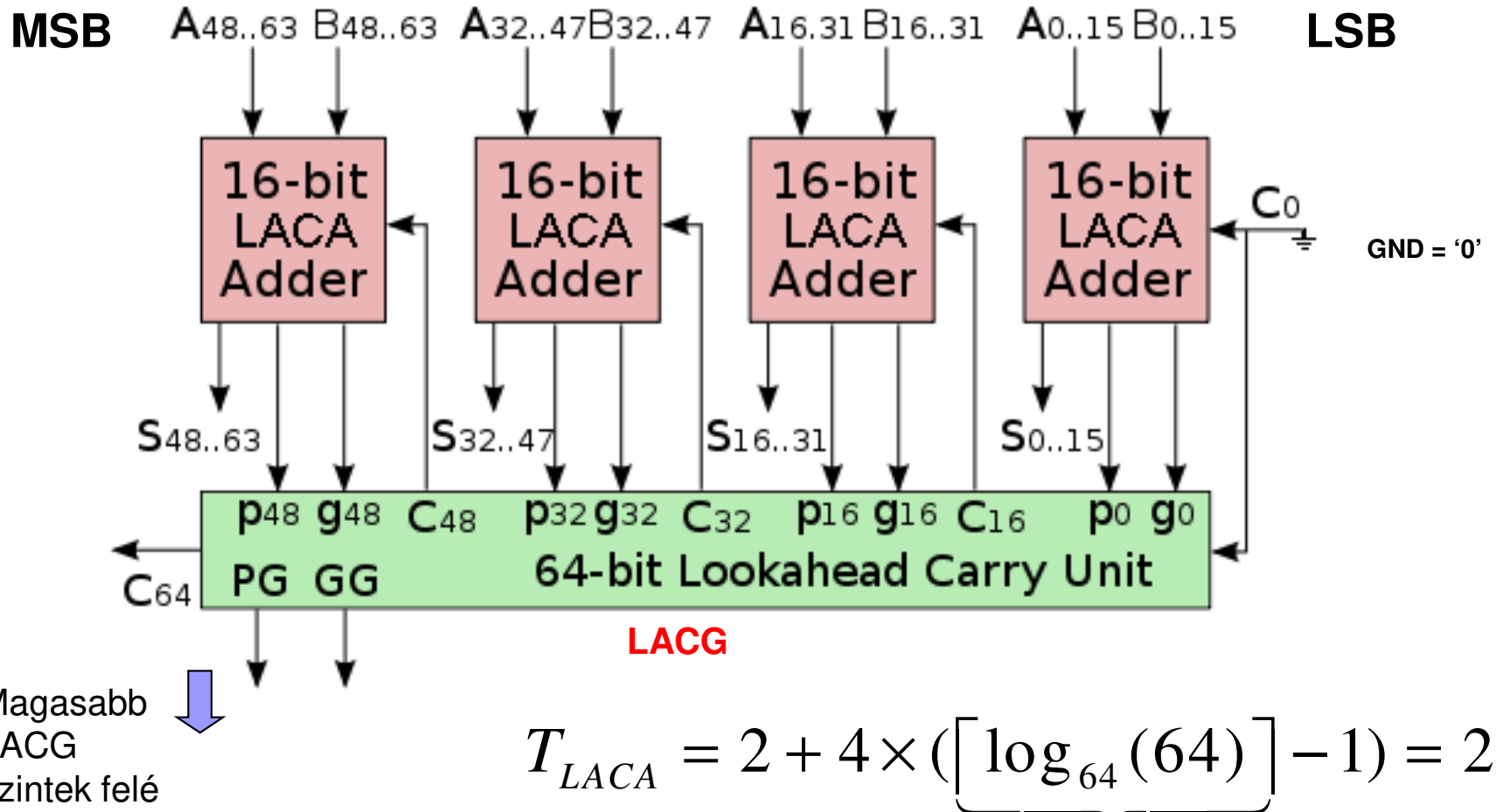
- Magasabb b-bites LACG hierarchia szintek felé GP (group propagate) és GG (group generate) számítása:

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

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

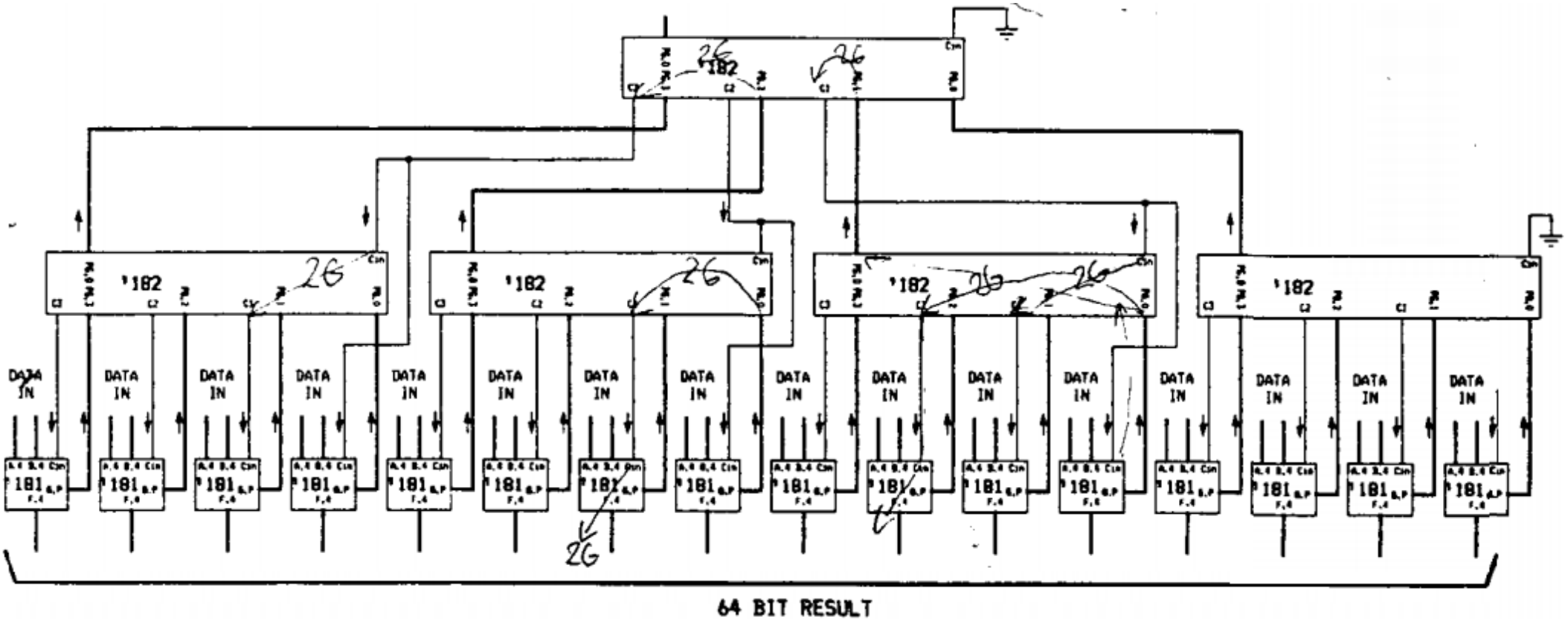
Példa: 4x16-bites LACA

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



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

64-bites (16×4) LACA összeadó



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

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

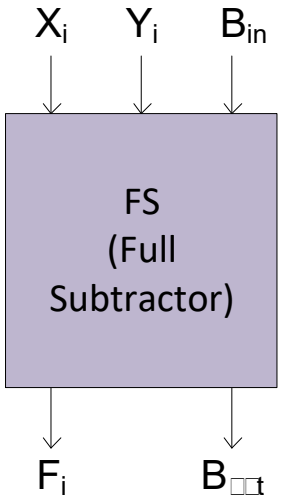
e.) Teljes kivonó - Full Subtractor (FS)

- FS: 1-bites Full Subtractor

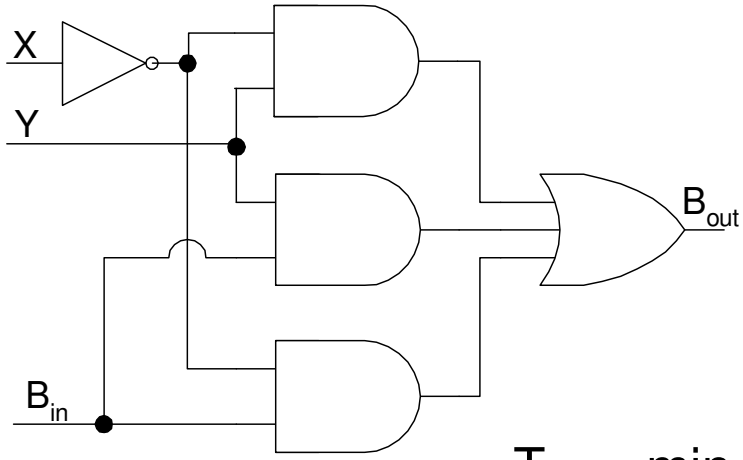
igazságtáblázat

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

szimbólum



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



T_{FS} = min 2G

Karnaugh táblái:

B_{out}:

X \ Y Bin	00	01	11	10
0	0 (0)	1 (1)	1 (3)	1 (2)
1	0 (4)	0 (5)	1 (7)	0 (6)

F_i:

X \ Y Bin	00	01	11	10
0	0 (0)	1 (1)	0 (3)	1 (2)
1	1 (4)	0 (5)	1 (7)	0 (6)

Kimeneti fgv-ei:

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

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

e.) Bináris kivonás végrehajtása (folyt)

- **I. módszer:**
  Bináris kivonás *FS* segítségével
  - $X_i - Y_i \rightarrow F_i$
  - 0 – 0 → 0
  - 0 – 1 → 1, borrow bit '1'
  - 1 – 0 → 1
  - 1 – 1 → 0

```
  ******          *
 100000000      256
-  01001100   - 76
----------     ----
  10110100      180
```

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

- **II. módszer:** Kivonás visszavezetése az *univerzálisan teljes* bináris *összeadás* segítségével (2's komplemens alak):
  - FA, RCA, vagy LACA

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

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
  - PI: Legyenek:
    - ‘A’ és ‘B’ 5-bites számok (0…2^5-1)=0…31
    - Maximálisan P=31*31=961 lehet (10 biten ábr.)
- Tehát: N-bites számok szorzatát 2×N biten tudjuk eltárolni!

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

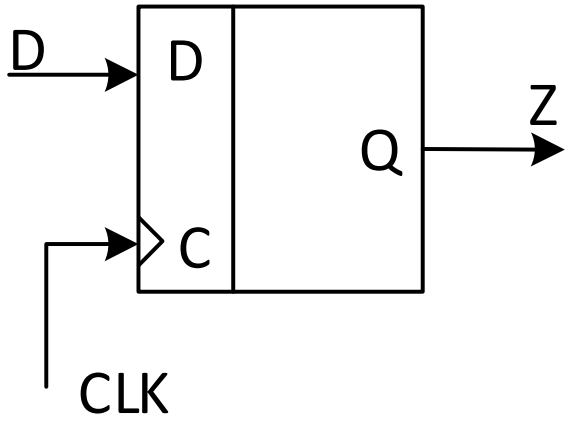
$$= A \times B_4 \times 2^4 + A \times B_3 \times 2^3 + A \times B_2 \times 2^2 + A \times B_1 \times 2^1 + A \times B_0 \times 2^0$$

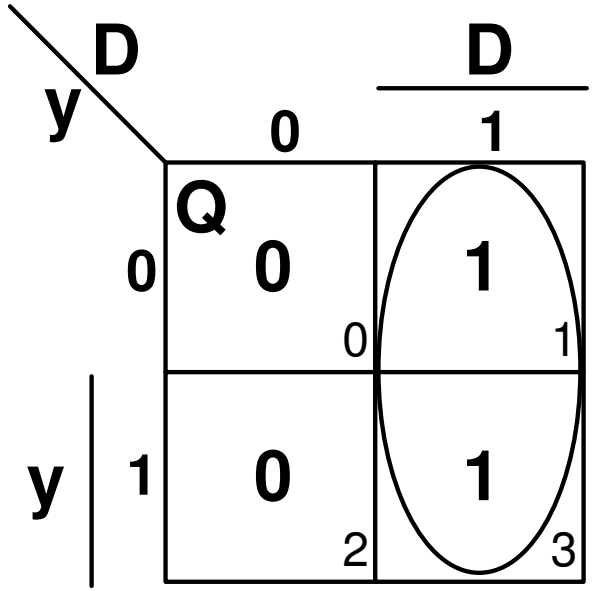
Ismétlés: Regiszterek

- 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.
- Megvalósítások működési mód szerint:
  a) *Hagyományos reg.*: párhuzamos betöltésű/kiolvasású
  b) *Léptető (Shift regiszter)*: soros betöltésű (kiolvasású)

Ismétlés: D flip-flop

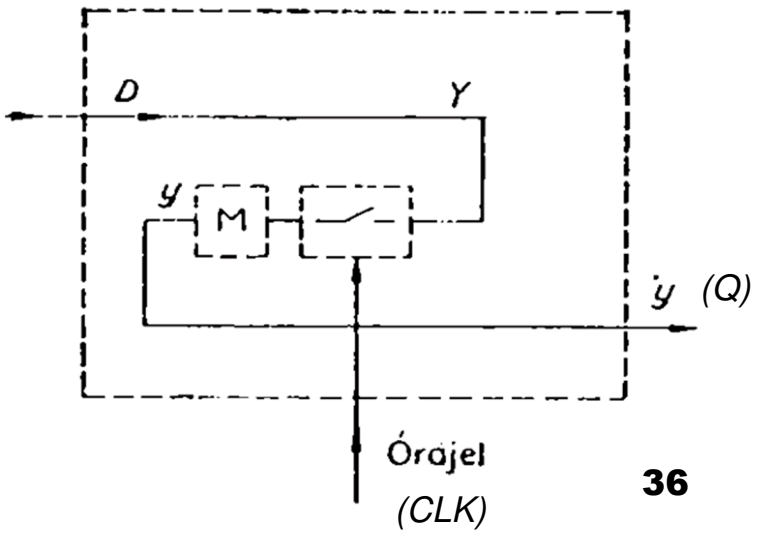
- **D tároló**
  - Csak szinkron módon értelmezhető
- **Működése:**
  - `D='0' ==` D-FF állapota változik (‘0’-t tárol)
  - `D='1' ==` D-FF állapota változik (‘1’-et tárol)
  - Tehát egy órajel ciklus (CLK) ideig tároljuk a bemenetre érkezett értéket, változás a CLK élére történhet



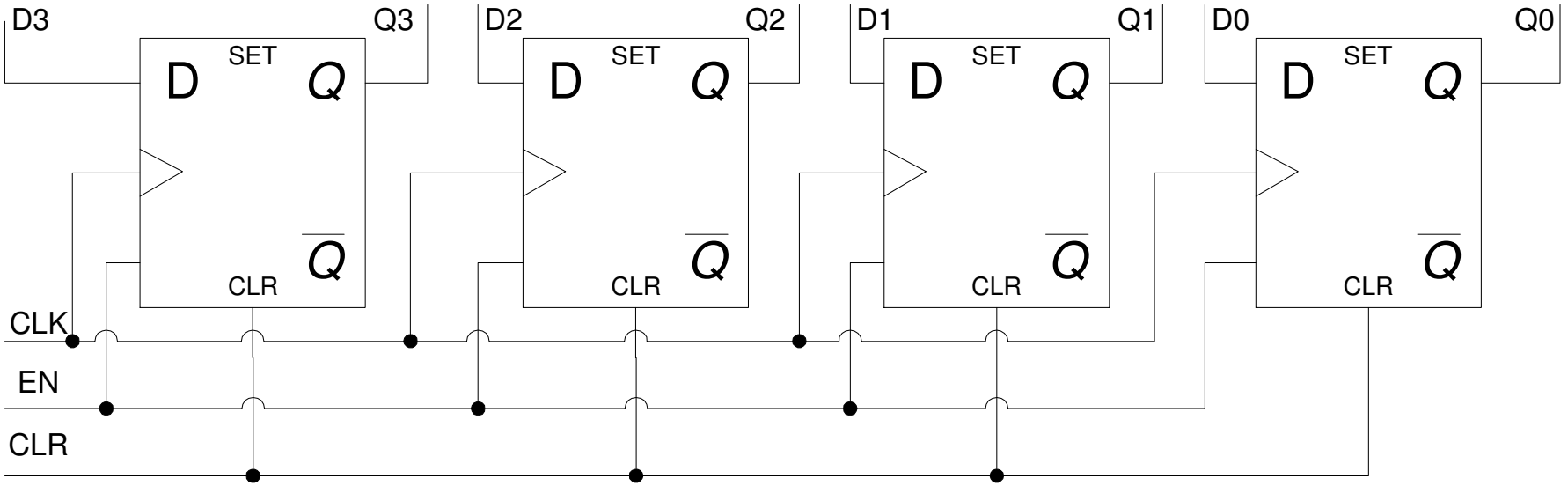


Egyszerűsített DNF alak és elvi logikai rajz:

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

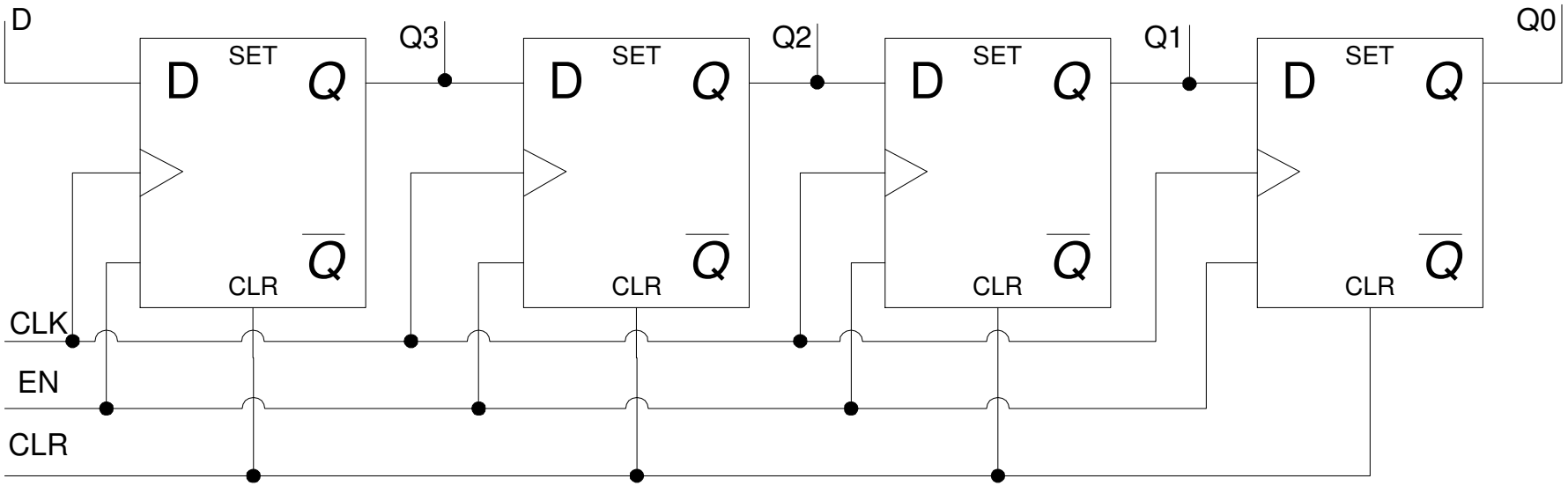


a.) 4-bites Parallel In/ Parallel Out regiszter (D-tárolókból felépítve)



Katalógus adat: SN54/74LS175

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



Katalógus adat : SN54/74LS95

Iteratív szorzási műveletek:

- Hagyományos (Shift&Add) módszer (MSB ← LSB)

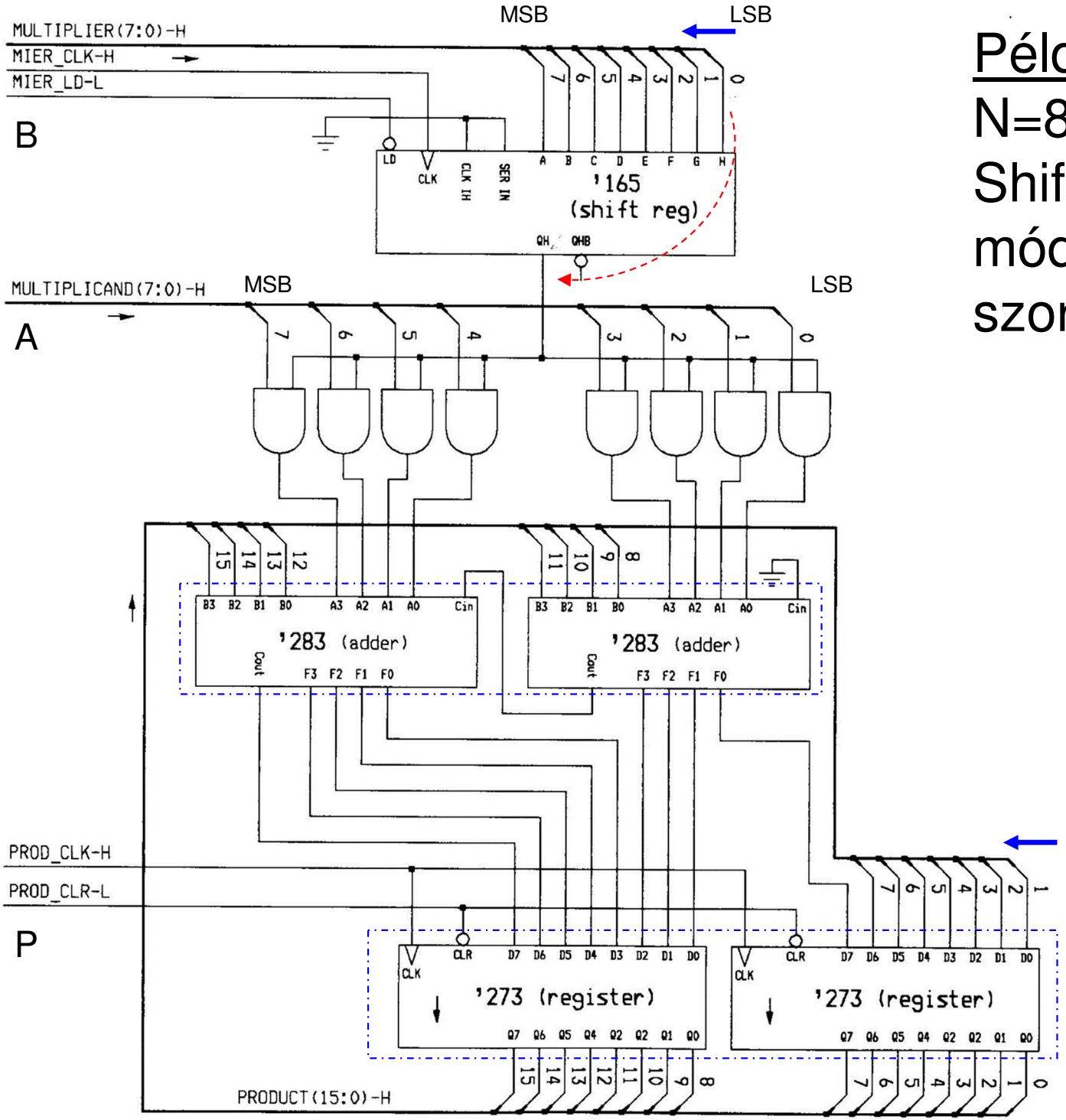
					A4	A3	A2	A1	A0
				x	B4	B3	B2	B1	B0
PP0					A4*B0	A3*B0	A2*B0	A1*B0	A0*B0
PP1				A4*B1	A3*B1	A2*B1	A1*B1	A0*B1	
PP2			A4*B2	A3*B2	A2*B2	A1*B2	A0*B2		
PP3		A4*B3	A3*B3	A2*B3	A1*B3	A0*B3			
PP4	A4*B4	A3*B4	A2*B4	A1*B4	A0*B4				
PR	az egyes oszlopok összege								

- 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 képezi (mivel a B szorzat biteket is ebben a sorrendben tölti be a Shift regiszterből N CLK alatt)
- AND kapuk: PPi-k képzése
- Shift-elés: *Huzalozott eltolással* (a visszacsatolt ágban 1. ← 0.)

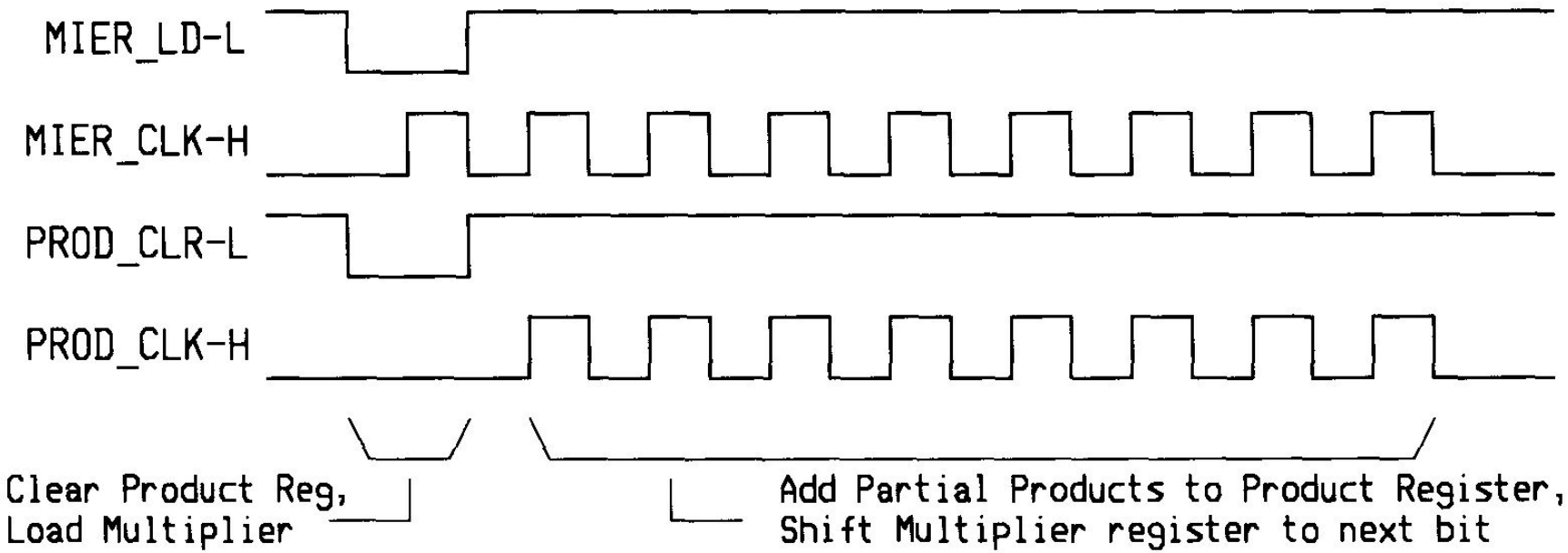


Példa: két N=8bites szám Shift&Add módszerű szorzására

Shift&Add szorzó építőelemei (folyt):

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

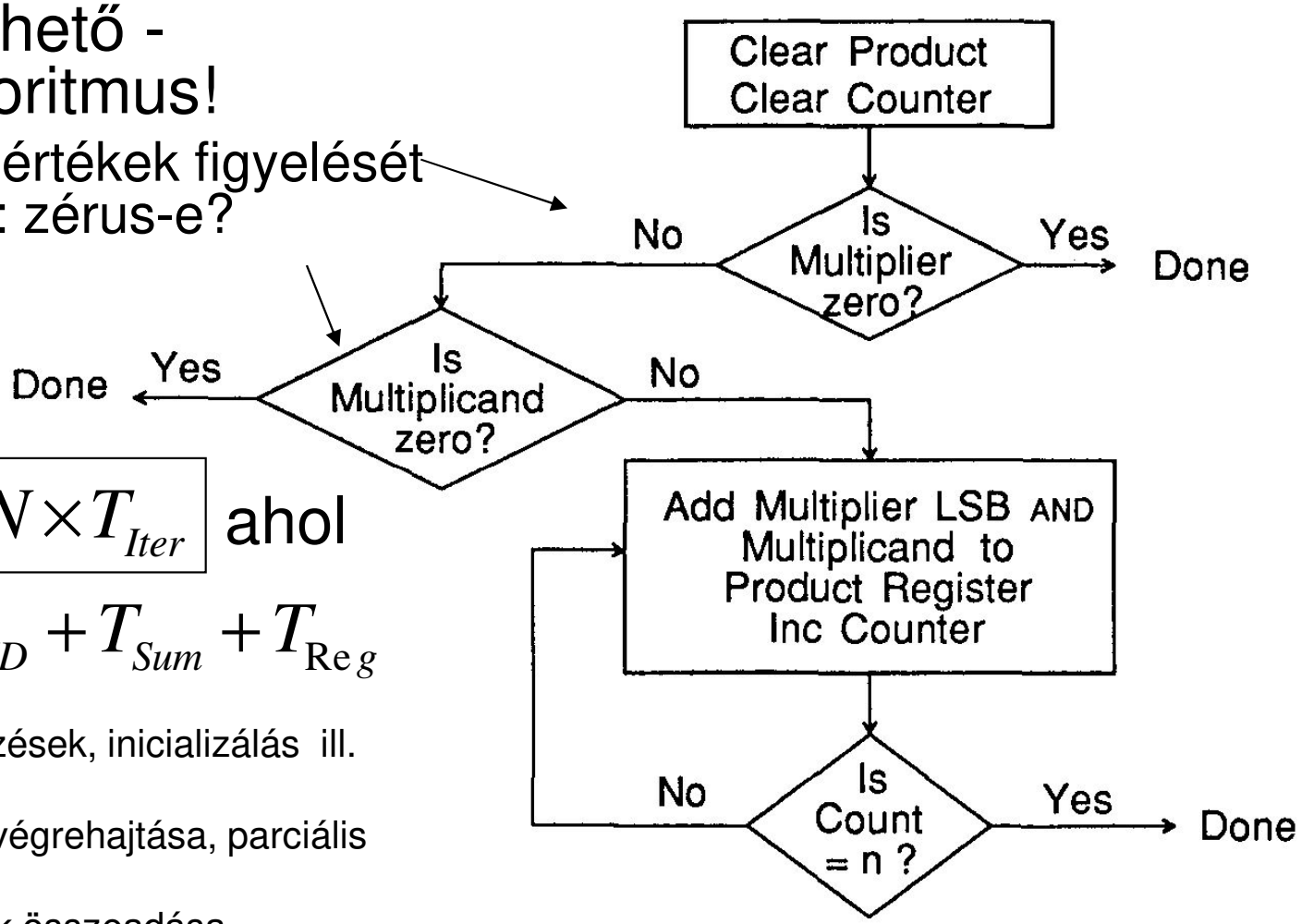
Időzítési jelek:



- 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?



- Időszükséglet:

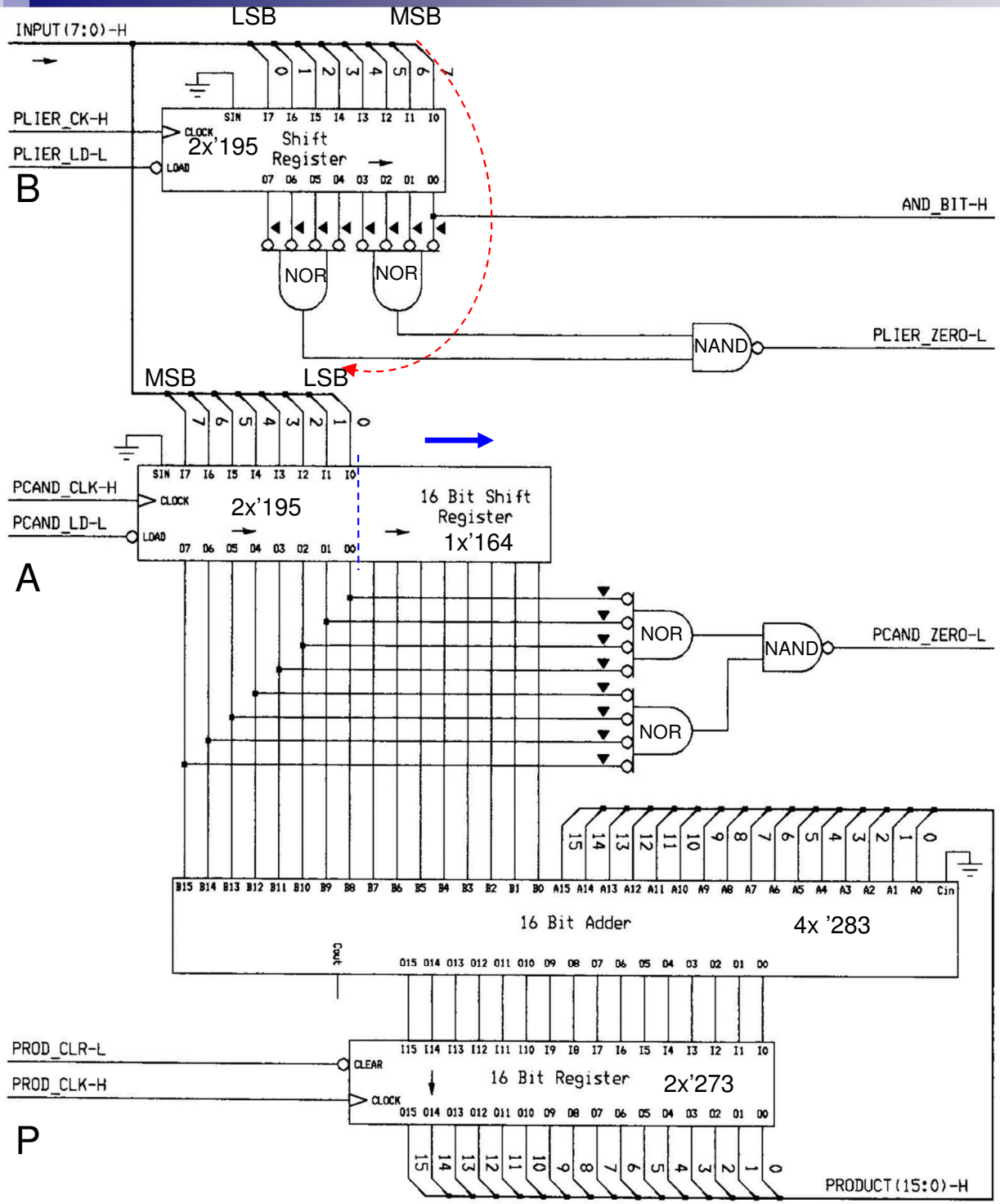
$$T_{Mult} = T_{Setup} + N \times T_{Iter}$$ ahol

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

•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

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
  - Tehát a B szorzat biteket *fordított* sorrendben tölti be, illetve elsőként az MSB pozíción lévő PPi értéke kerül a szorzat (C) regiszterbe
- Adatfüggőség: bemenetek figyelésekor ha a szorzandó, vagy szorzó bitek értéke zérus, nem kell elvégezni a szorzást (vizsgálata N-bit szélességű AND kapukkal)



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

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

Adatfüggőség: zérus-e? PLIER_ZERO_L / PCAND_ZERO_L

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

Ha az MSB=’1’ (B<7>), akkor szorzandó és szorzó reg. tartalma összeadható;
Ha MSB=’0’ a B szorzó- és az A szorzandóregiszter tartalma 1-bitpozícióval shift-elődik jobbra:

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

Fordított sorrendű szorzó építőelemei (folyt):

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

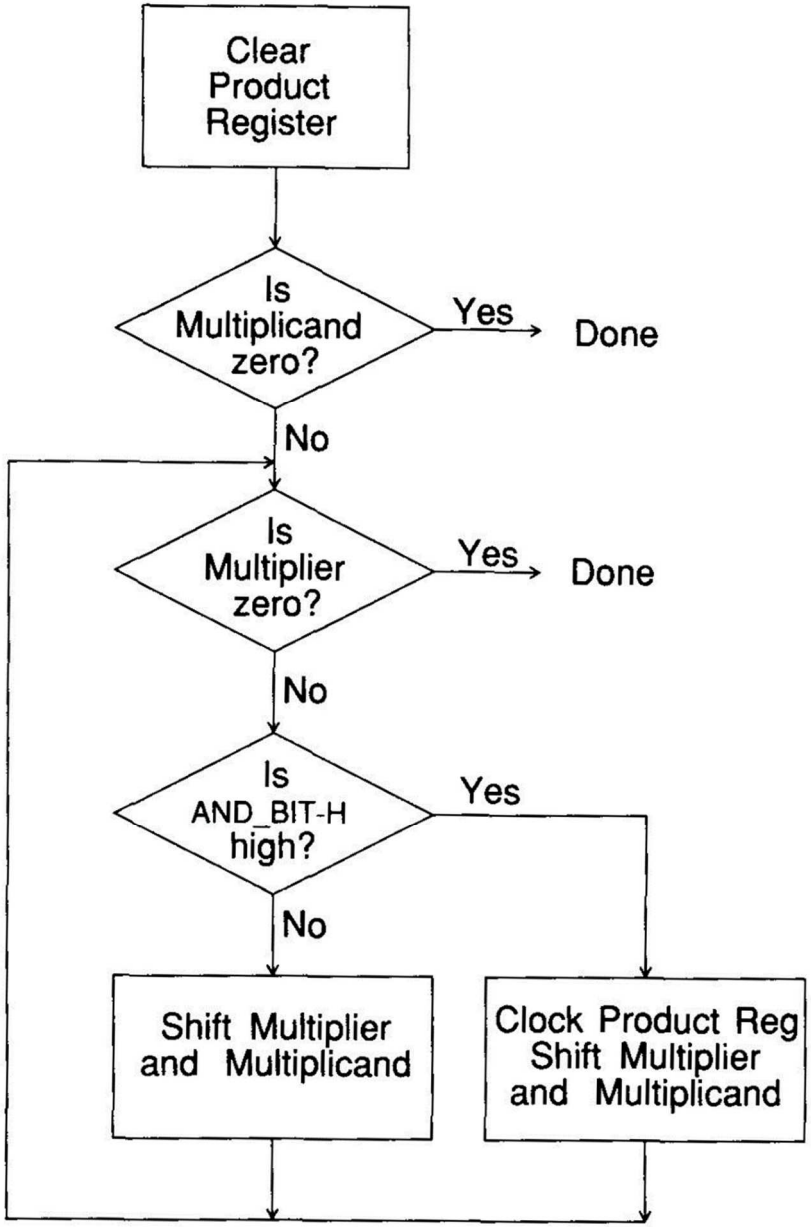
Folyamatábra (fordított sorrendű)

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

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

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

•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 **P=A×B**-t!

Az újrakódolást bitpárokon végezzük el:

$-1\times(B_0-0)=\underline{-1}$	P0=0–1*A	Mivel – volt az érték, ezért kivonjuk a 0-ból az A-t.
$-2\times(B_1-B_0)=\underline{+2}$	P1=P0+2*A	Mivel + volt az érték, ezért hozzáadjuk P0-hoz a 2*A-t.
$-4\times(B_2-B_1)=\underline{-4}$	P2=P1–4*A	kivonjuk
$-8\times(B_3-B_2)=\underline{0}$	P3=P2	Mivel ‘0’ volt, Áteresztés, nem változik.
$-16\times(B_4-B_3)=\underline{+16}$	P4=P3+16*A	hozzáadjuk
$-32\times(B_5-B_4)=\underline{-32}$	P5=P4–32*A	kivonjuk

végeredmény értéke

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

Példa: Booth algoritmus (folyt.)

Végrehajtjuk **P=A×B**-t!

543210 (bitpozíciók) N= 6, P = 2×N

011001= 25 „A” Ellenőrzés:

101101= –19 „B” P = –475

// 1110_0010_0101

P0=0–1*A =0–1*25= –25

P1=P0+2*A= –25+2*25=25

P2=P1–4*A= 25–4*25= –75

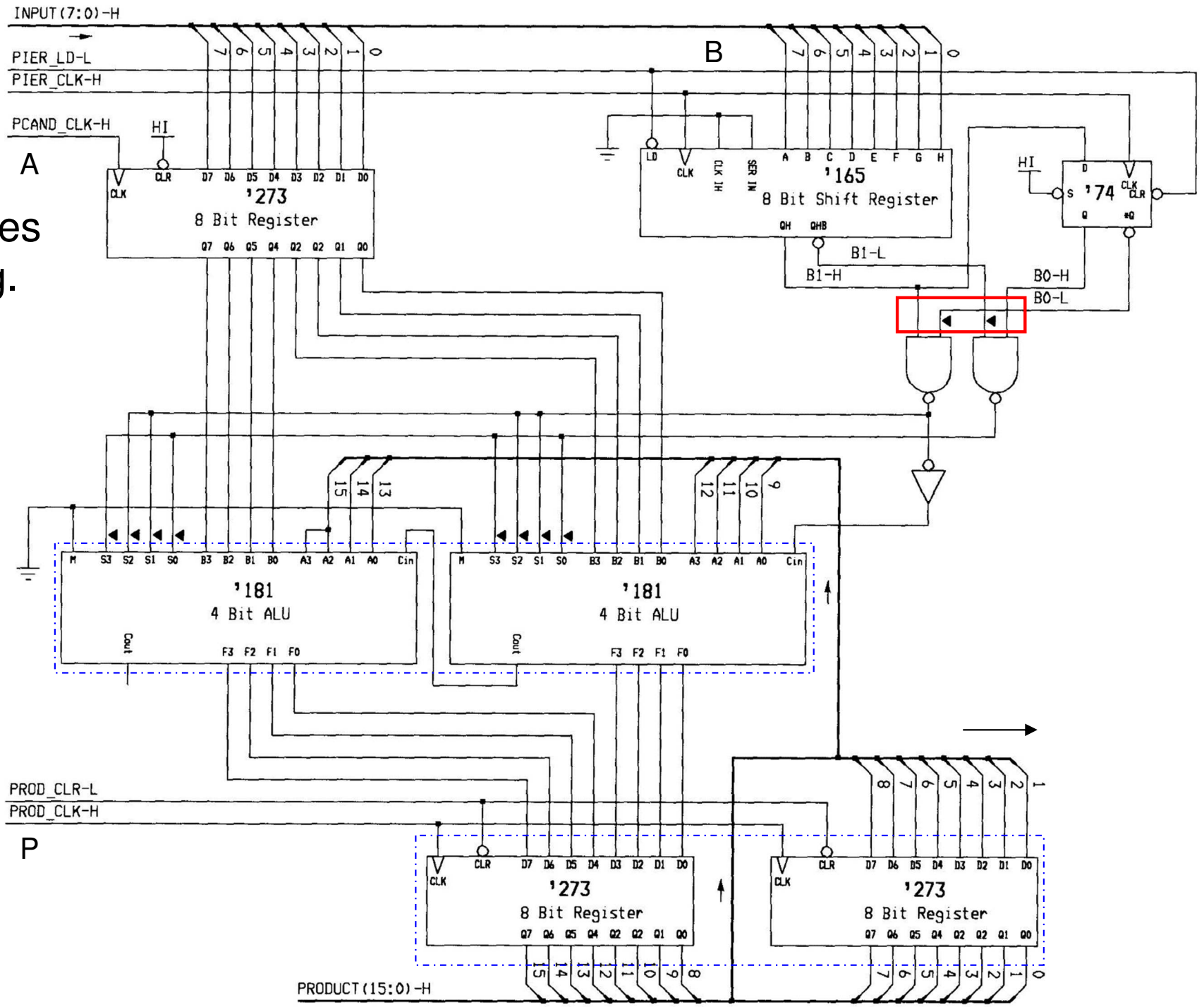
P3=P2 = –75

P4=P3+16*A = –75+16*25=325

P5=P4–32*A = 325–32*25= –475

Példa: két 8-bites szám Booth alg. szorzására

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



II. Közvetlen „szorzási” módszerek

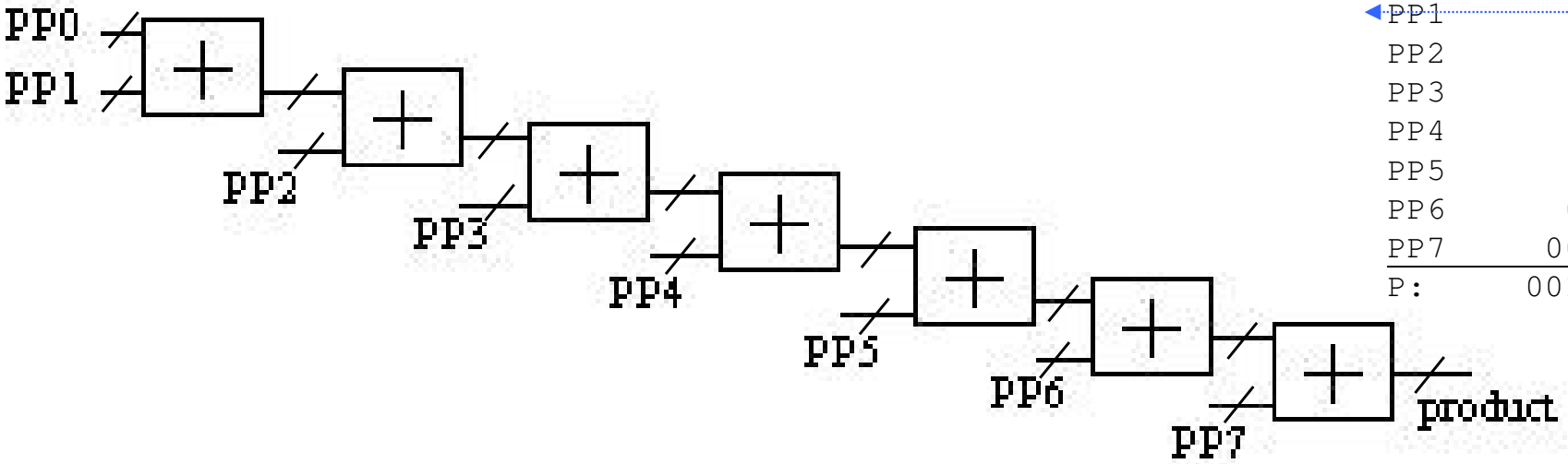
Közvetlen „szorzási” módszerek

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

a.) Lineáris modell

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

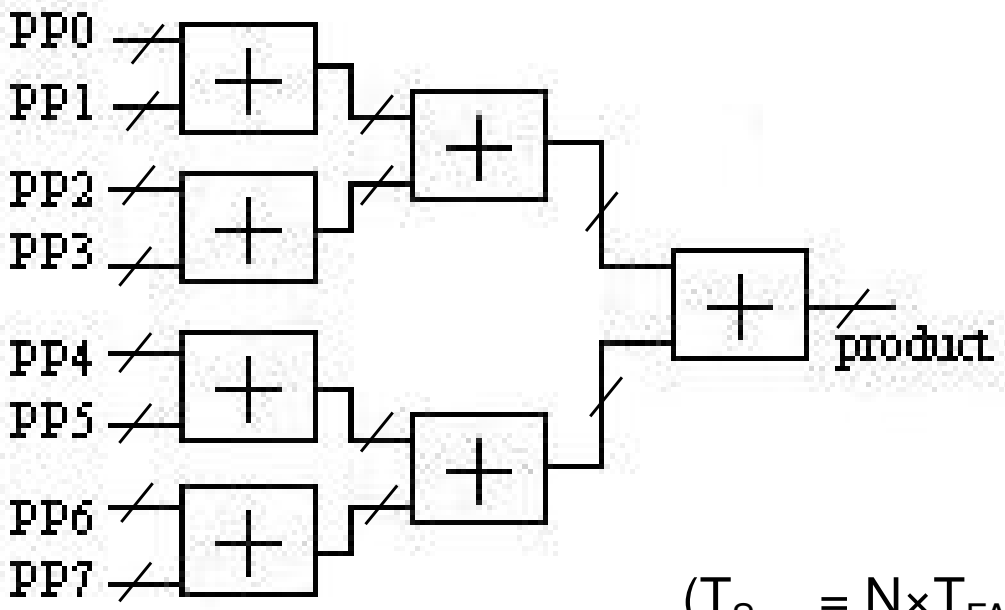
$$T_{(DIRECT\text{-}LINE)} = (N-1) * T_{(SUM)}$$




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

b.) Fa modell

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



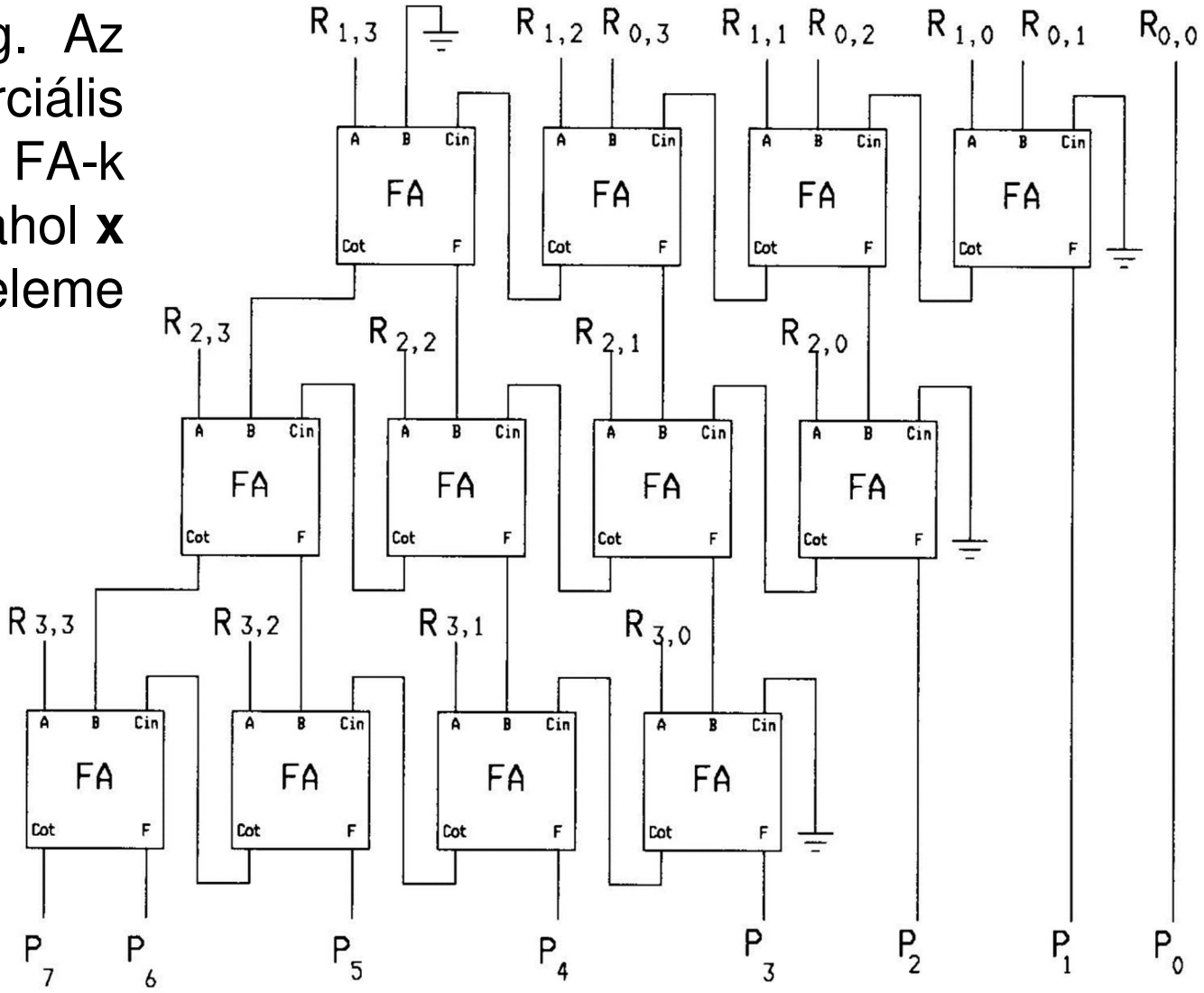
($T_{Sum} = N \times T_{FA}$)

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

c.) Full Adder-es megvalósítás

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

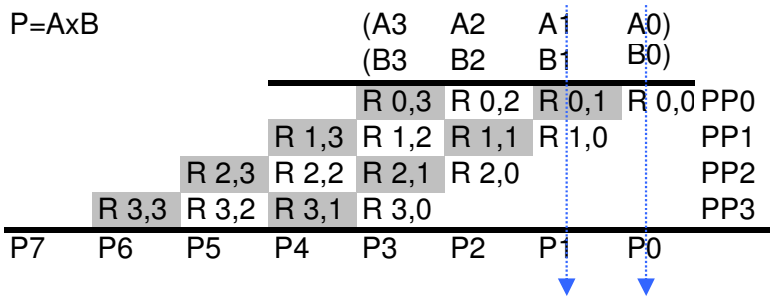
P=AxB				(A3	A2	A1	A0)	
				(B3	B2	B1	B0)	
				R 0,3	R 0,2	R 0,1	R 0,0	PP0
			R 1,3	R 1,2	R 1,1	R 1,0		PP1
		R 2,3	R 2,2	R 2,1	R 2,0			PP2
	R 3,3	R 3,2	R 3,1	R 3,0				PP3
P7	P6	P5	P4	P3	P2	P1	P0	

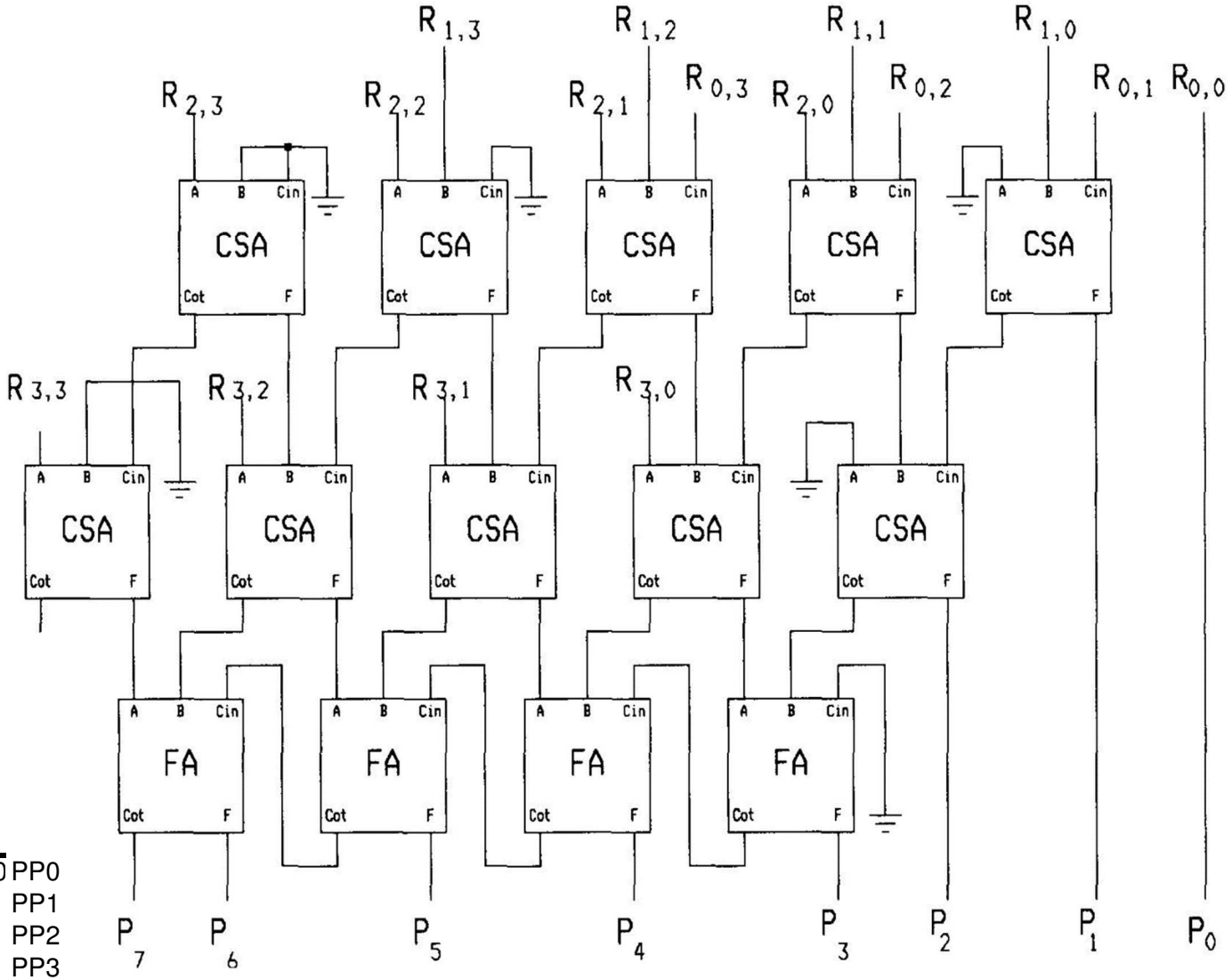


d.) CSA: Carry Save Adder

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

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





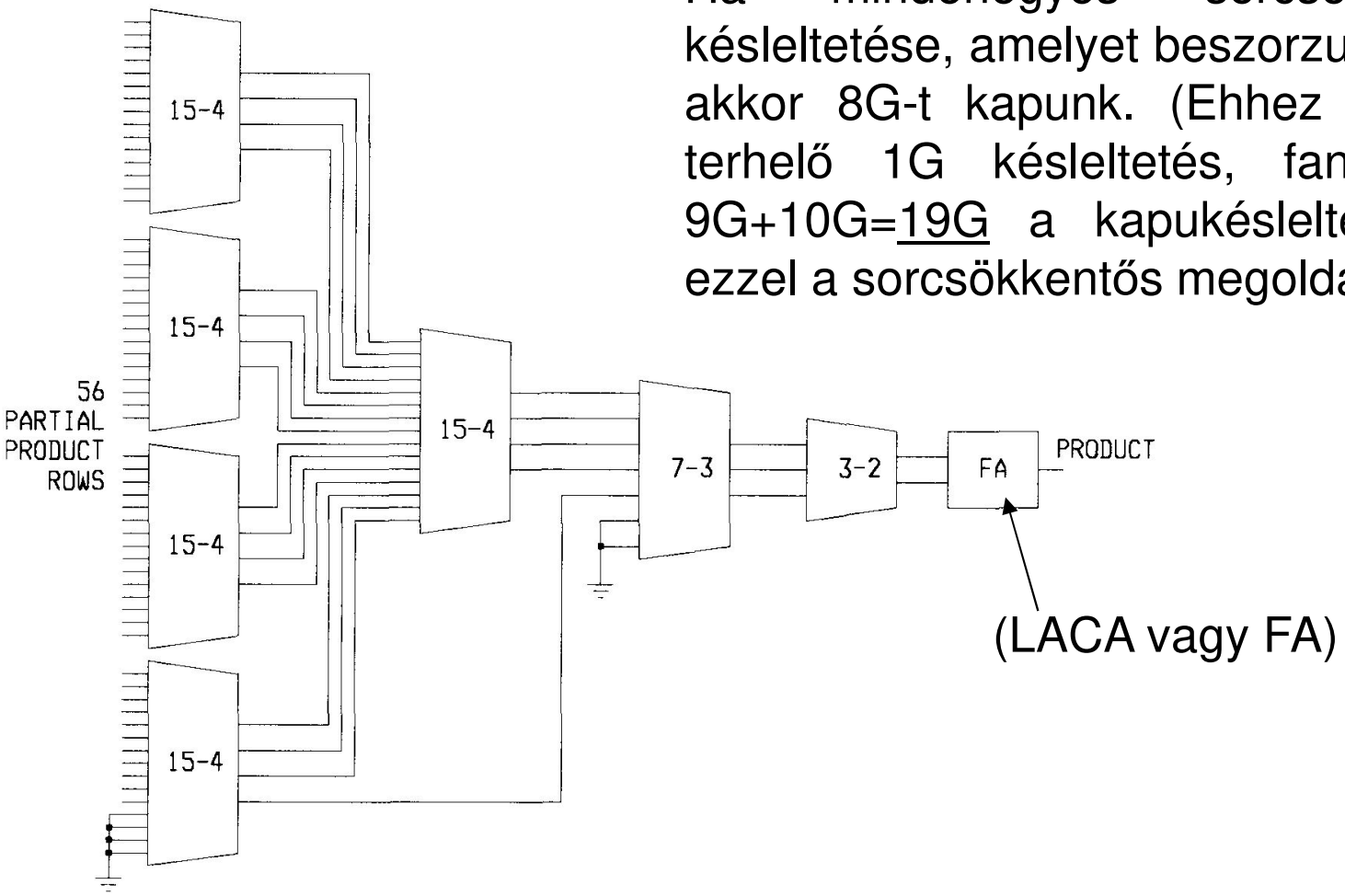
e.) Sorcsökkentős megvalósítás (row reduction)

- 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, fan-out.) Így összesen 9G+10G=19G a kapukésleltetése az eszköznek, ezzel a sorcsökkentős megoldással.



Osztó áramkörök

Osztó áramkörök:

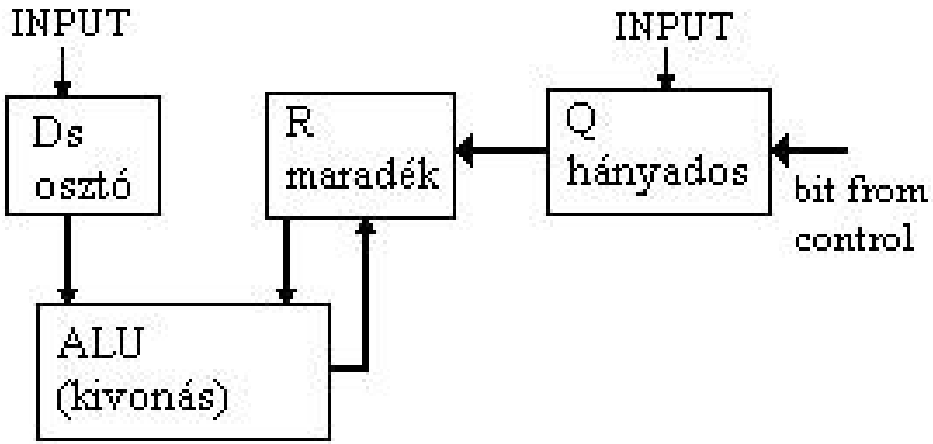
- I.) Hagyományos – „lassú” vagy közvetlen iteratív osztási algoritmus
  - Euklideszi osztás
- II.) „Gyors” - iteratív osztási algoritmusok (egyéb numerikus módszerek)

I.) Hagyományos közvetlen osztási algoritmus:

Ez az osztási folyamat igen lassú eljárás. Lépései:

1. az *osztót* a Ds regiszterbe rakjuk, az *osztandót* (Dd) pedig 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 shift-elünk a visszaszorzásnál)
6. A hányados generálódik elsőként az MSB felől, és a Q-ba shift-elődik 1 bittel balra
7. A folyamat végén a maradék Az R-ben, a hányados pedig a Q-ban lesz

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



Példa: Hagyományos osztási algoritmus

Dd=Q*Ds + R

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

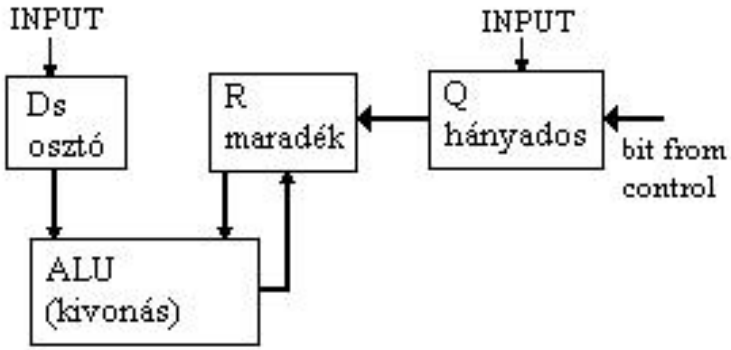
Decimális számok esetén:

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

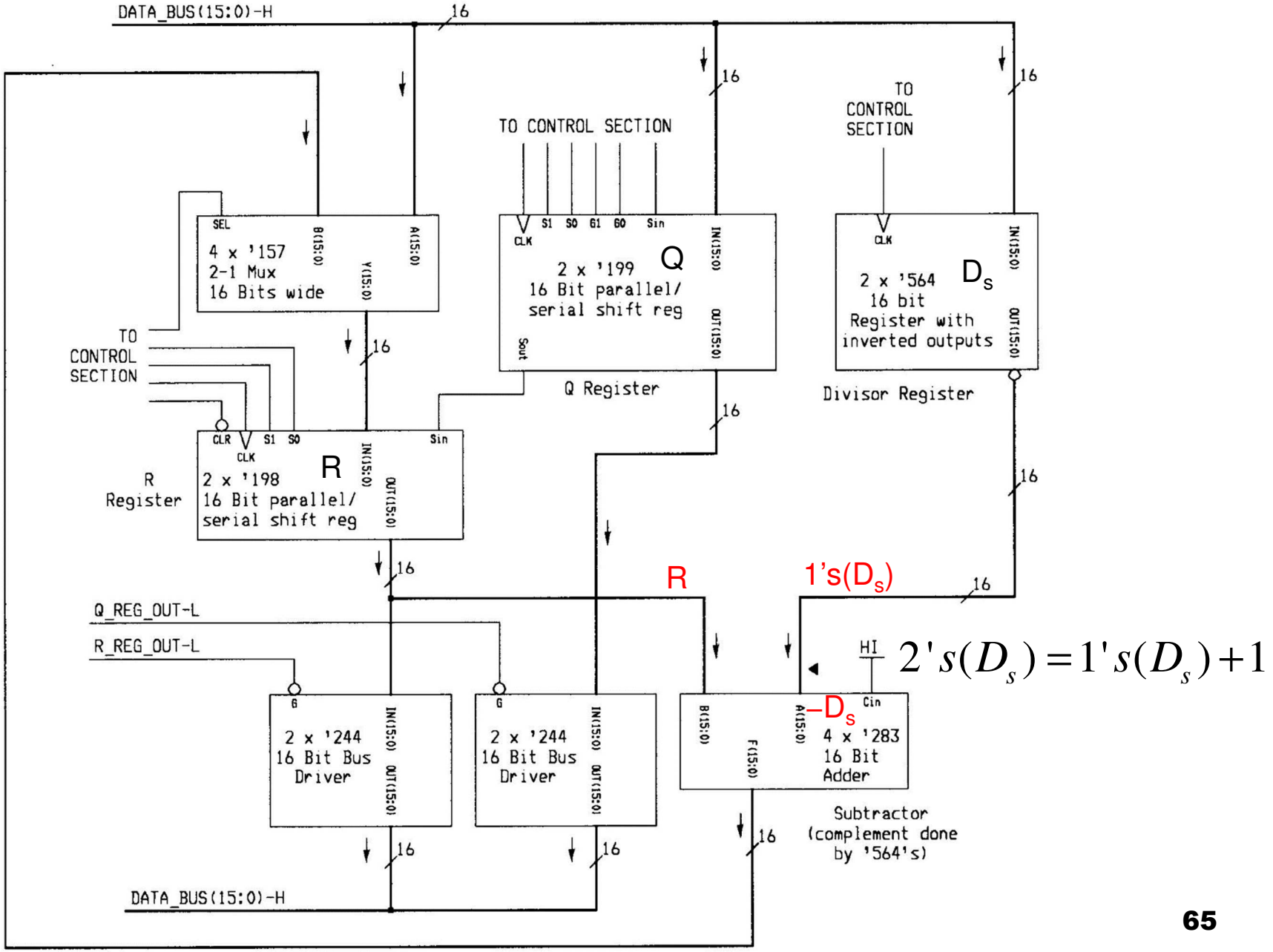
Bináris számok esetén hasonlóan

```
   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!
```

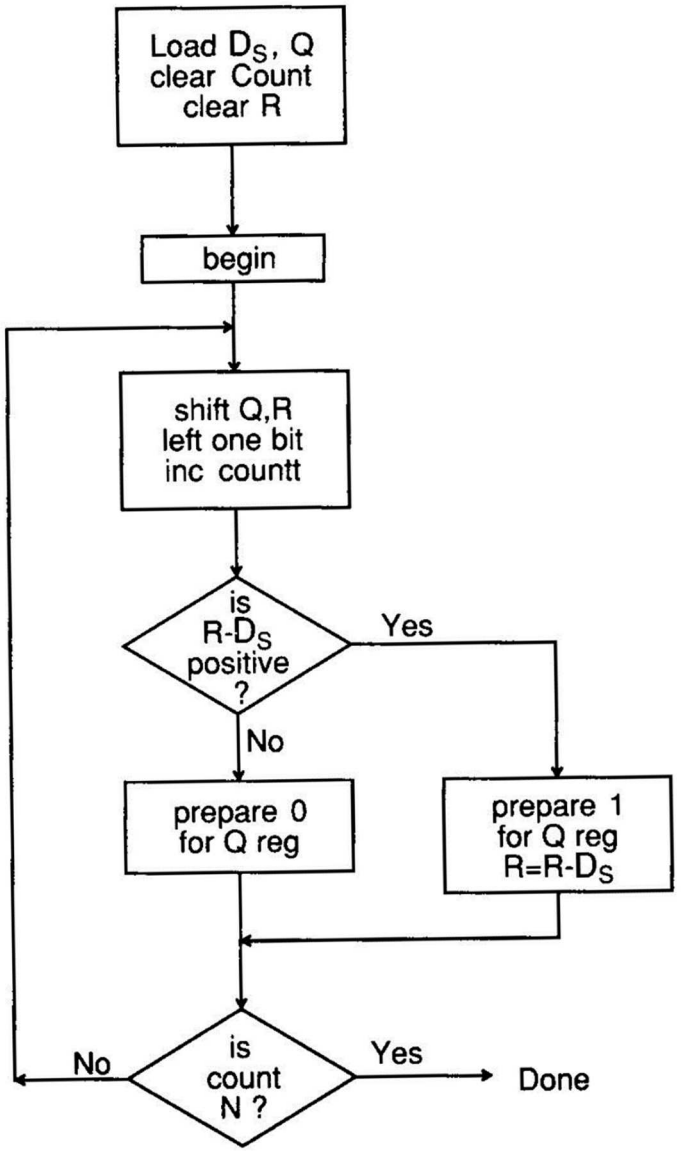


Hagyományos osztó áramkör



$$2's(D_s) = 1's(D_s) + 1$$

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!

Példa: Newton Raphson Szám négyzetgyökének közelítése

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

$$x^2 = 612$$

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

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

- Deriváltja:

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

- Kezdeti érték x_0 = 10-nek választásával kapjuk:

$$\begin{array}{lcccccl}
x_1 & = & x_0 - \frac{f(x_0)}{f'(x_0)} & = & 10 - \frac{10^2-612}{2\cdot 10} & = & 35.6 \\
x_2 & = & x_1 - \frac{f(x_1)}{f'(x_1)} & = & 35.6 - \frac{35.6^2-612}{2\cdot 35.6} & = & \underline{2}6.3955056 \\
x_3 & = & \vdots & = & \vdots & = & \underline{24.7}906355 \\
x_4 & = & \vdots & = & \vdots & = & \underline{24.7386}883 \\
x_5 & = & \vdots & = & \vdots & = & \underline{24.7386338}
\end{array}$$

Várt érték:
24.73863375370…

Aláhúzások, már a korrekt számjegyeket jelölik, az egyes iterációkban

b.) Közvetlen gyors osztó

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

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

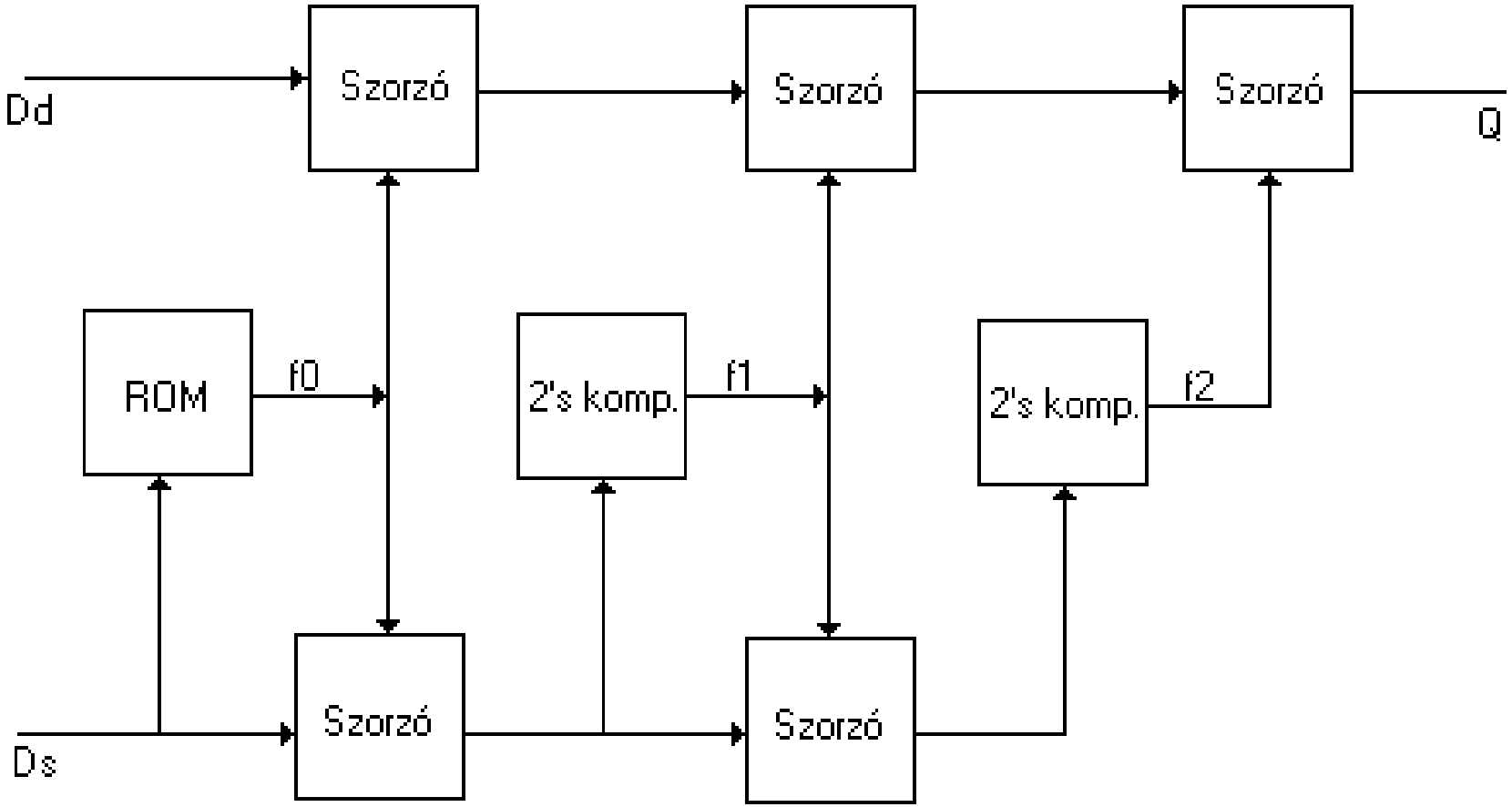
Közvetlen gyors osztó működése

- 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 (számítás)

- Az osztás művelete f_o kiszámolásával kezdődik. Válasszuk $\mathbf{f_o = 1+x = 1+(1-D_S) = 2-D_S}$. Így $D_S \times f_o = (1-x)(1+x) = 1-x^2$. Így sokkal közelebb kerültünk 1-hez, mintha csak a D_S –et használtuk volna. Minden iterációs lépésben a számláló és a nevező is f_K tényezőkkel szorzódik, és közelebb kerülünk a Q pontos értékéhez. Legyen $f_1 = 1+x^2$. Így $D_S \times f_o \times f_1 = 1- x^4$ és ez tovább ismételhető iteratív módon
- Tehát azt kapjuk, hogy $\mathbf{D_{Dn+1} = D_{Dn} \times f_n}$.
- 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)

Közvetlen gyors osztó áramköri felépítése



Példa 1.

Legyen az osztandó D_D = 0.4, osztó D_S = 0.7, és 6 iterációs lépésig számoljunk (7 tizedesjegy pontosságú számokkal). Ekkor 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).

Várt érték:
0.571428571428571428571428571428571

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≈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=1,8500000
- 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 kívánt értéket (D_{D6}=0,6666464) 6 iterációs lépés alatt. Ezért hogy elérjük a kívánt pontosságot véges számú lépés alatt, ROM-ot kell használni (ahol f_o kezdeti értékét tároljuk).

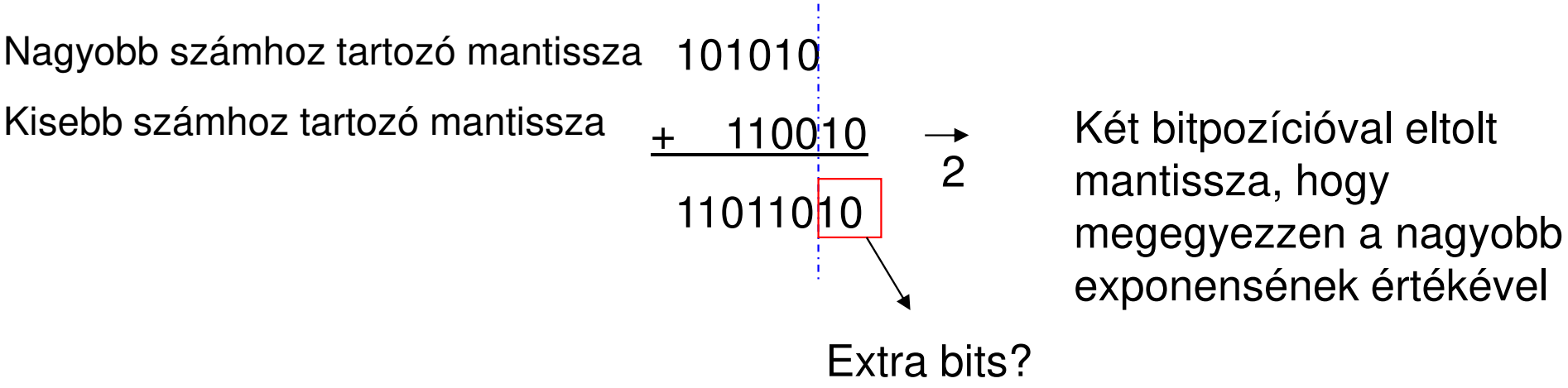
Várt érték:
0.6666666666666666666666666666667

Extra bitek kezelése

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

Extra bit: probléma

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



a.) Truncation (levágás)

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

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

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

Truncation: példa

- ∑ ERR → bias: mindig pozitív

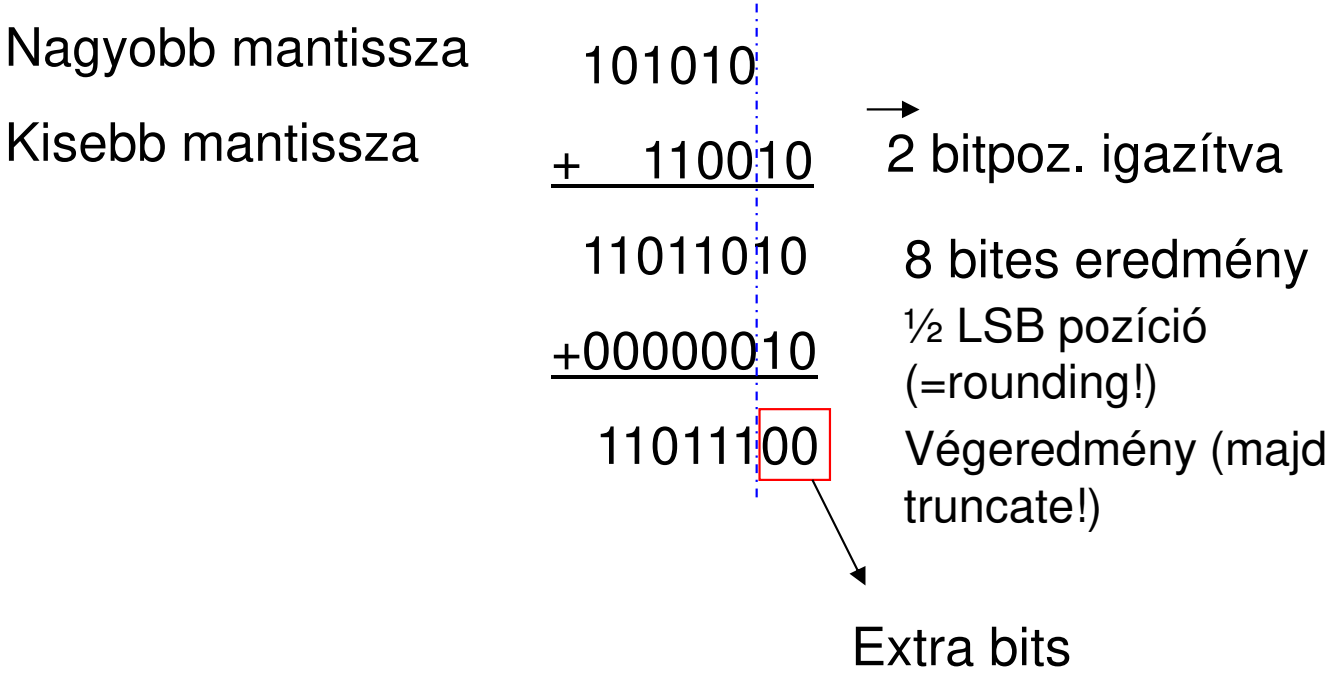
$ERR_{TRUNC} = M_R - M_F$

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

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

b.) Rounding (kerekítés)

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



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	xx1.01	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!

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

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

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

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

PI: Round-to-Zero (R*)

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

case	MR	MR+1/2 LSB	truncated 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	xx1..	xx1.	+0.10
h	xx1.11	xx1.11	xy0.	-0.01

Eltérés a Rounding-hoz képest!

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

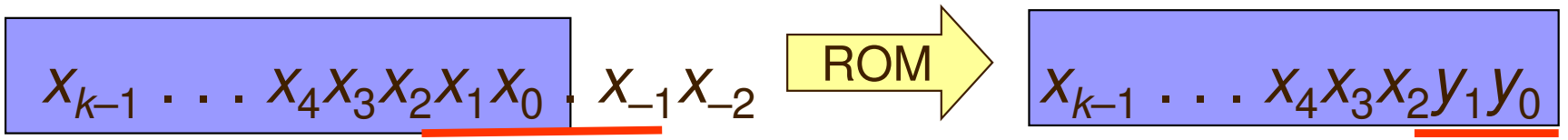
Normál kerekítéshez képest a **c.)/g.)** eseteknél egy ‘1’-es lett **direkt beállítva** (force) az LSB helyén (xx**1**). De csak a **g.)** esetnél lesz más a hiba értéke (ERR_{ZERO}).

d.) Jamming (~fix értéken rögzítés)

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

e.) ROM rounding

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



ROM address (3cím -> 8x2 bites ROM/PLA)

ROM data

Extra-bit kezelő módszerek hibáinak összehasonlítása: