



Számítógép Architektúrák (MIKNB113A)

3. előadás: Aritmetikai egységek - adatkezelés

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

Jegyzetek, segédanyagok:

- Könyvfejezetek:

- <http://www.virt.uni-pannon.hu>

- Oktatás → Tantárgyak → Digitális
Rendszerek és Számítógép Architektúrák
([chapter03.pdf](#))

- Fóliák, óravázlatok .ppt (.pdf)

- Frissítésük folyamatosan

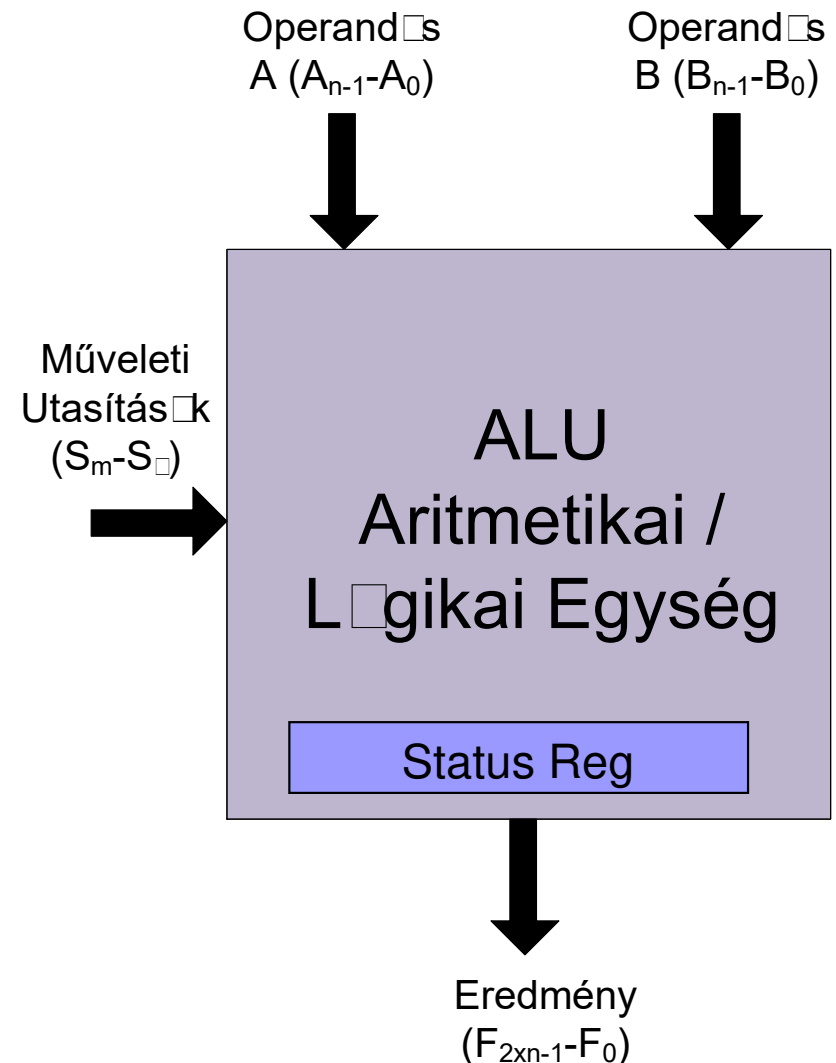
Ismétlés

- 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: $+$ $\rightarrow$ $-$, $*$, $/$ (alapműveletek)
 - Logikai operátorok:
NAND, NOR $\rightarrow$ 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.

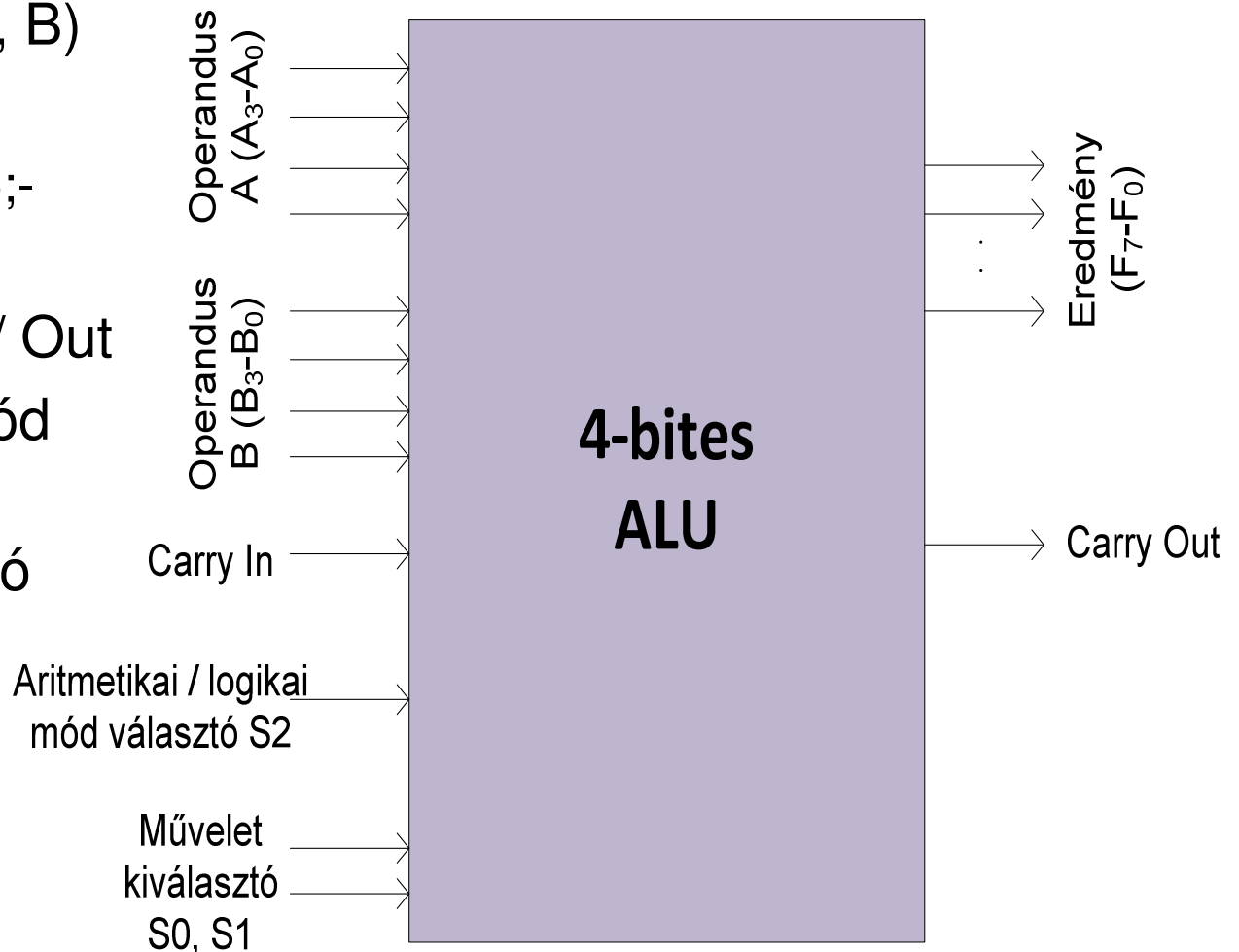


Státusz- (flag) jelzőbitek

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

PI: 4-bites ALU felépítése és működése

- Két 4-bites operandus (A, B)
- Eredmény (F)!
 - $N+(1 \text{ CarryOut})$ bit, ha +;-
 - $2 \times N$ bites, ha *; /
- H.értékes átvitel: CarryIn/ Out
- S2: Aritmetikai/ logikai mód választó (MUX)
- S0, S1: művelet kiválasztó ($S2$ értékétől függően)
 - Aritmetikai vagy Logikai

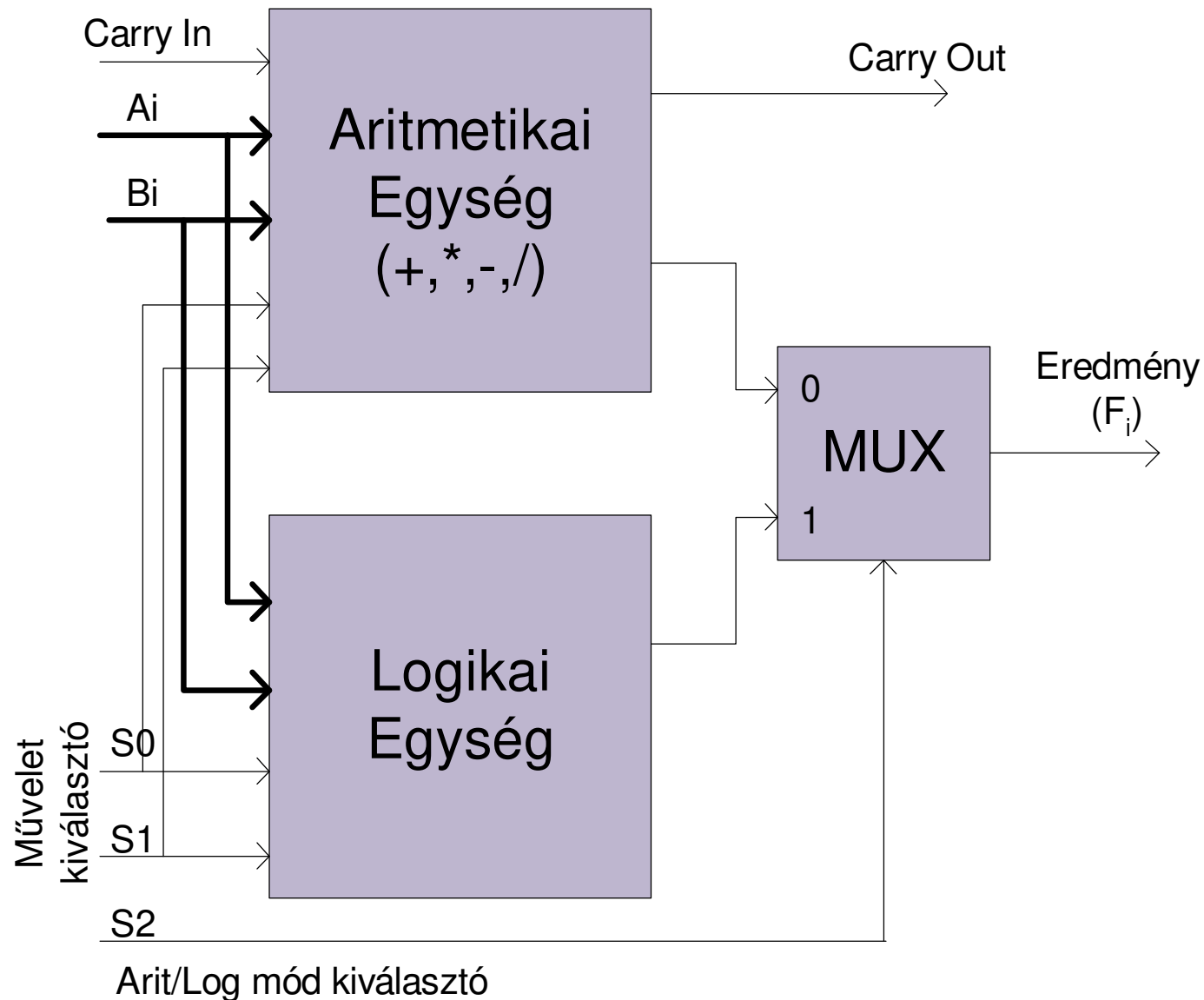


ALU működését leíró függvénytáblázat

(egy lehetséges működés, a funkciók bővíthetők):

Művelet kiválasztás:				Művelet:	Megvalósított függvény:
S2	S1	S0	Cin		
0	0	0	0	$F=A$	'A' átvitele
0	0	0	1	$F=A+1$	'A' értékének növelése 1-el (increment)
0	0	1	0	$F=A+B$	Összeadás
0	0	1	1	$F=A+B+1$	Összeadás carry figyelembevételével
0	1	0	0	$F = A + \overline{B}$	A + 1's komplement 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)

ALU felépítése:





Lebegőpontos műveletvégző egységek

Lebegőpontos műveletvégző egységek

■ Probléma:

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

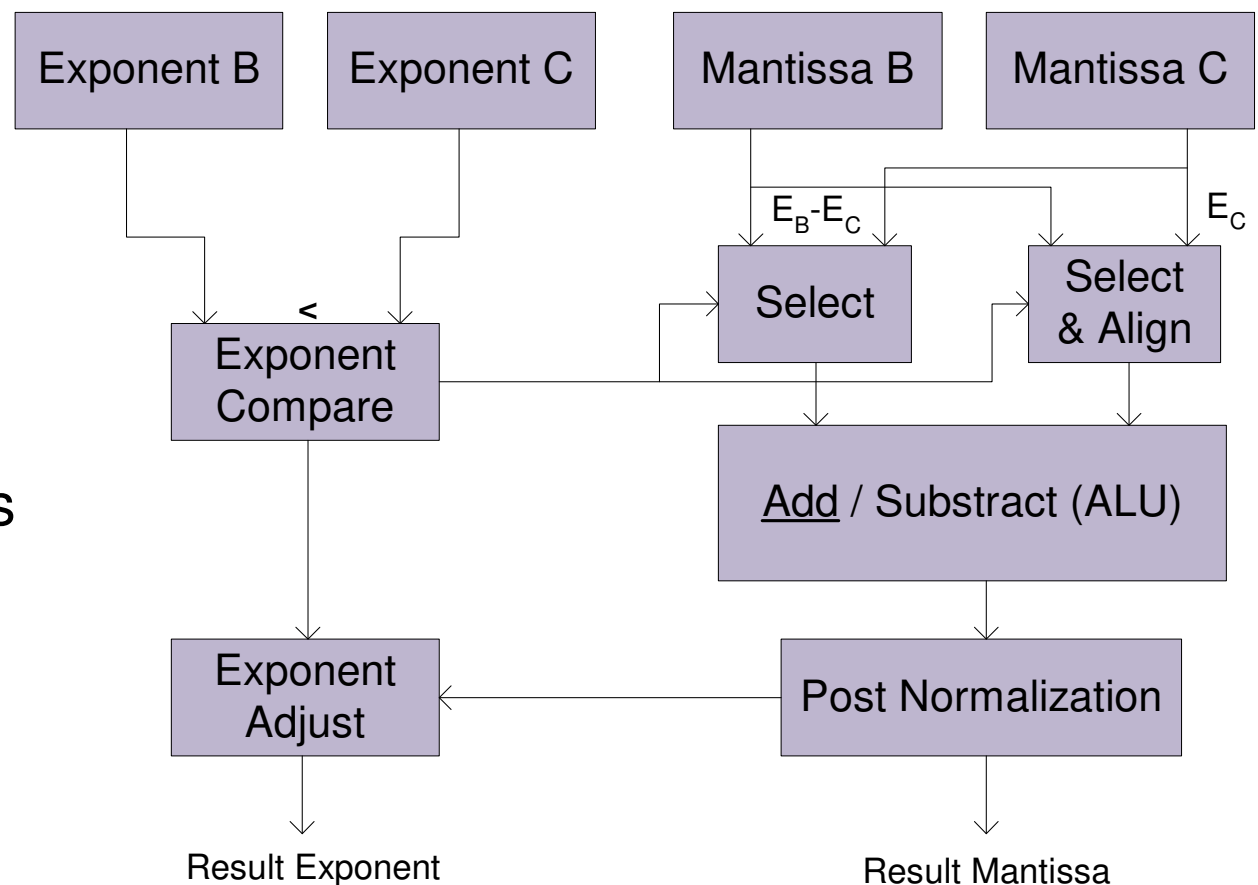
■ Műveletvégző elemek:

- ☐ Összeadó-,
- ☐ 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: $0 < B < C$
- $B \rightarrow C$ vagyis $|E_B - E_C|$ pozícióval jobbra igazítjuk az M_B mantisszát; ez változás az exponensben is
- ALU: Összeadás!: sign-magnitude formátumban
- Végül minimális post-normalizáció kell



FPN: összeadás (kivonás)

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

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

$$(1.00001 \times 2^{\color{red}1}) + (1.1011 \times 2^{\color{red}3}) =$$

$$(0.\underline{0}100001 \times 2^{\color{green}3}) + (1.1011 \times 2^{\color{green}3}) =$$

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

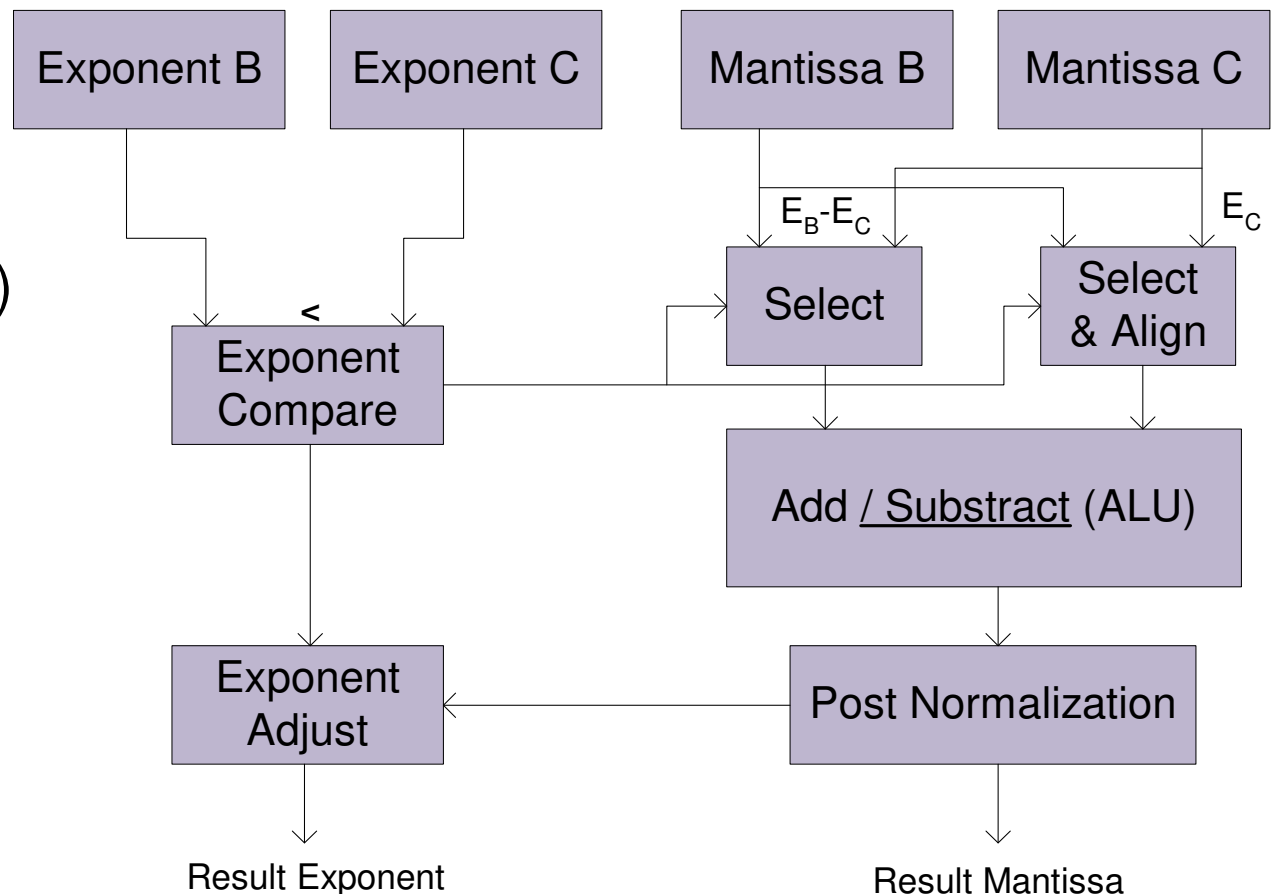
■ $\color{red}1.1111001 \times 2^{\color{red}3} = A$

(itt éppen nem kell post-normalizálás)

b.) Lebegőpontos kivonó

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

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

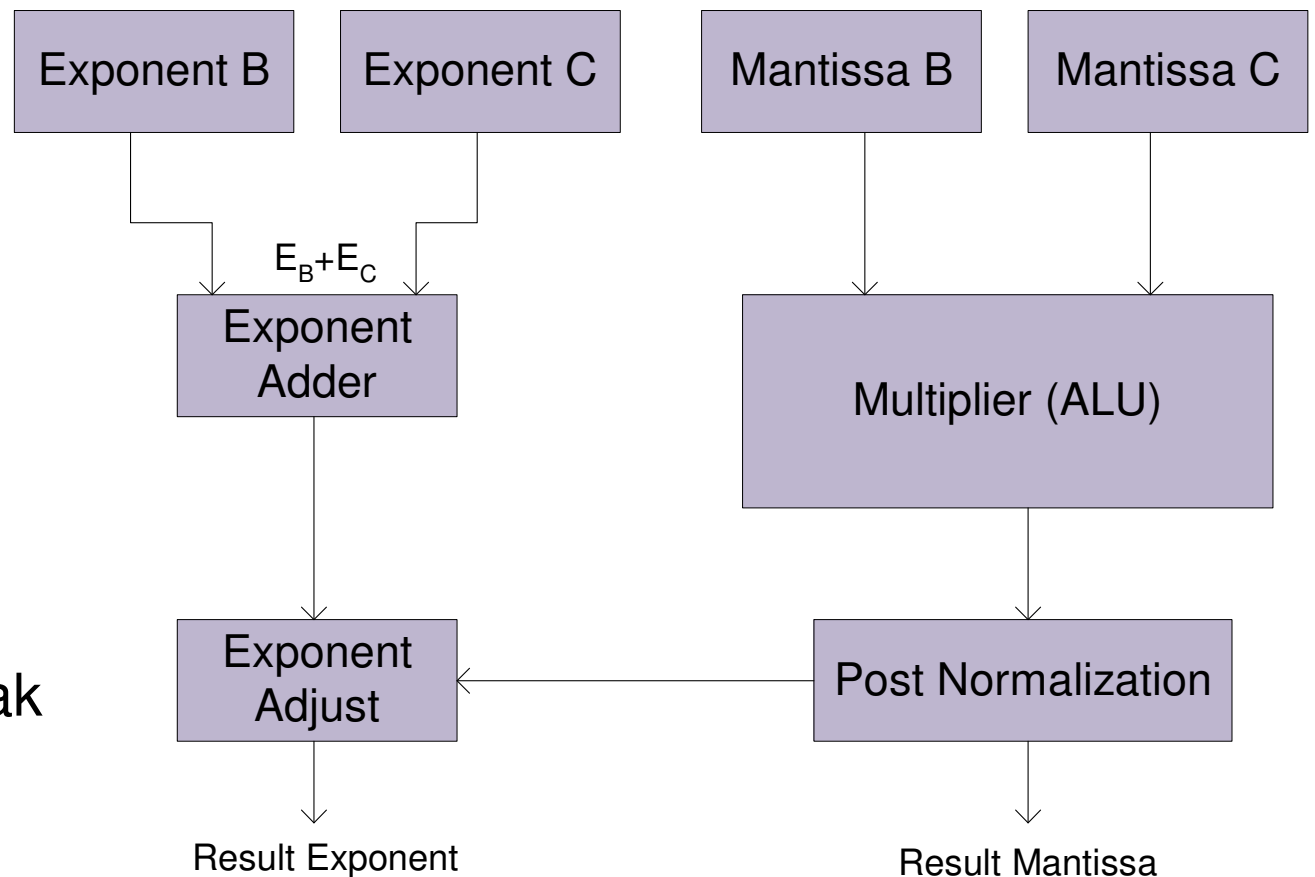


c.) Lebegőpontos szorzó

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

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

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



d.) Lebegőpontos osztó

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

□ A: hányados

□ B: osztandó

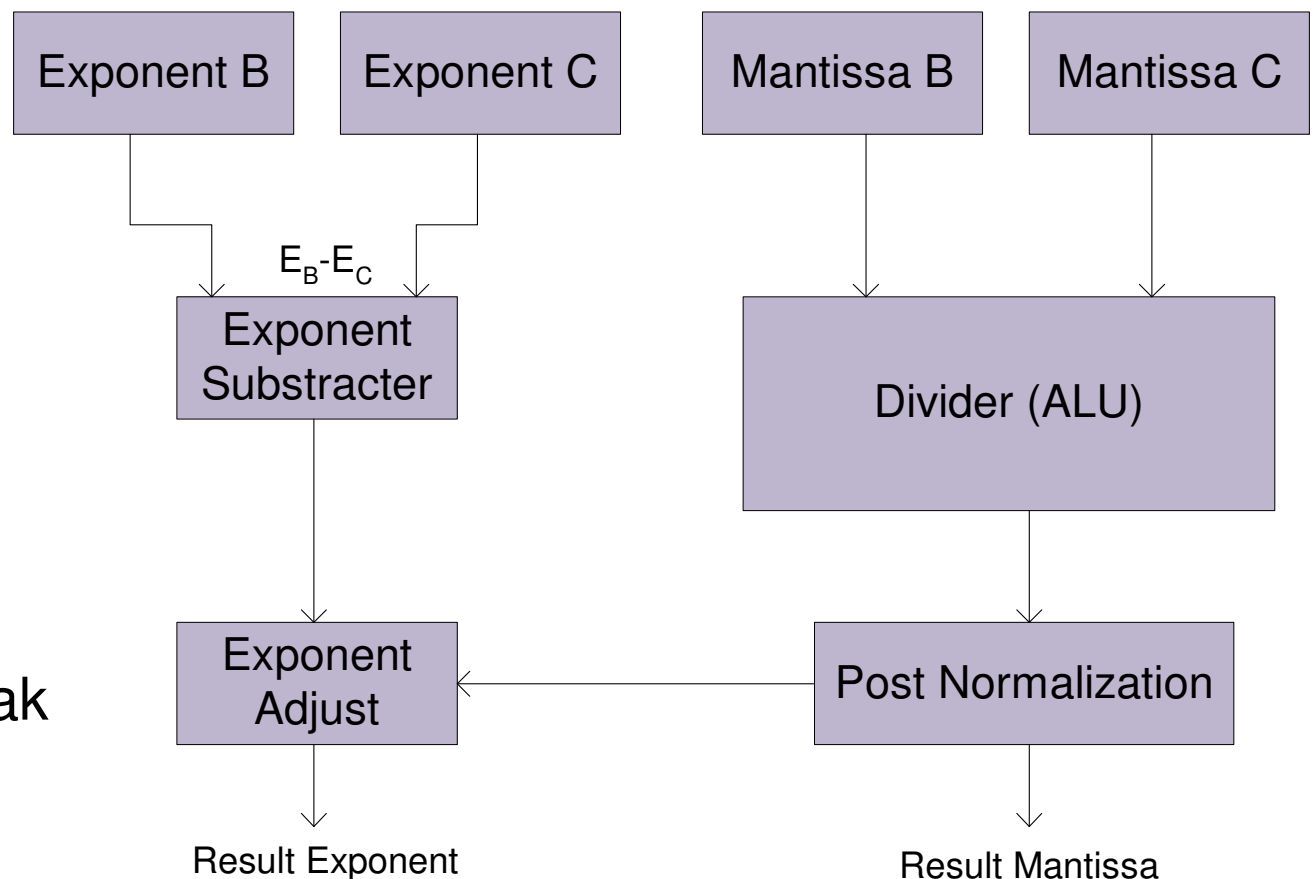
□ C: osztó

□ Könnyű végrehajtani

□ Nincs szükség az operandusok beállítására

□ Minimális post-normalizációt kell csak végezni

□ Osztás! (ALU)





Összeadó / Kivonó áramkörök

a.) Fél-összeadó – Half Adder

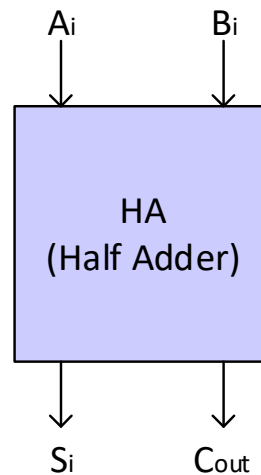
■ 1-bites Half Adder

igazságtáblázat

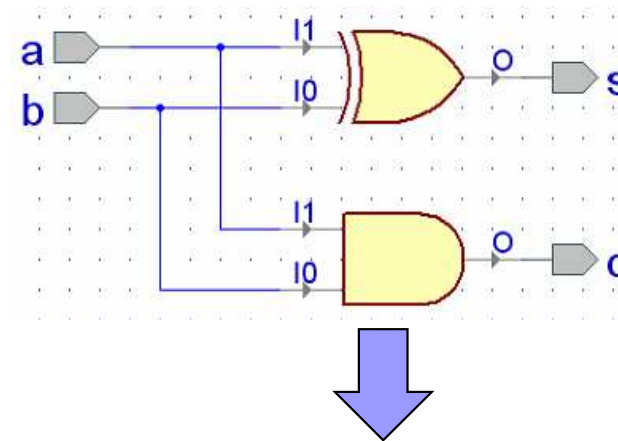
A _i	B _i	C _{out}	S _i
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Nem kezeli a Cin-t !

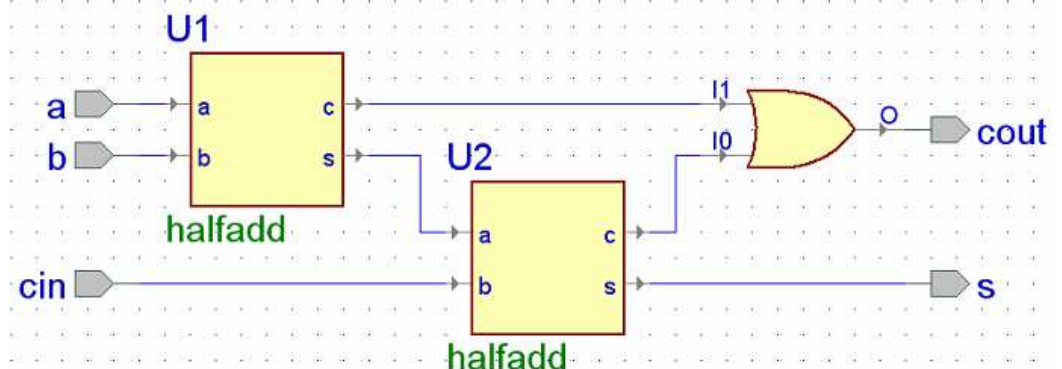
szimbólum



$$T_{HA} = 1G$$



1 bites FA felépítése 2 db HA segítségével:



Karnaugh táblái:

C_{out}:

	B	0	1
A	0	0	0
A	1	0	1

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

S_i:

	B	0	1
A	0	0	1
A	1	1	0

$$S_i = A_i \oplus B_i$$

Kimeneti fgv-ei:

b.) Teljes összeadó – Full Adder

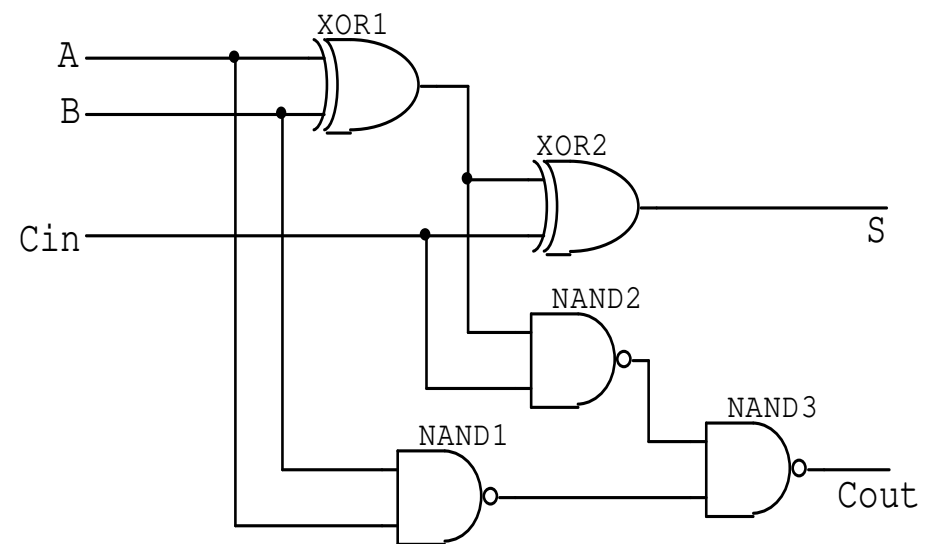
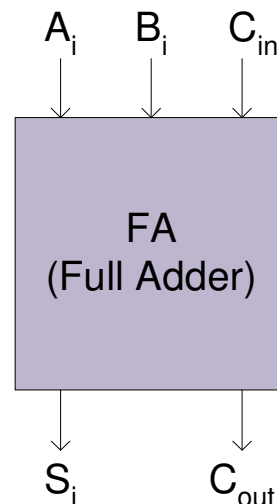
■ FA: 1-bites Full Adder

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

igazságtáblázat

A_i	B_i	C_{in}	C_{out}	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} :

		C_{in}			
		B			
A	C_{in}	00	01	11	10
	0	0	0	1	0
A	1	0	1	1	1

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

Kimeneti fgv-ei:

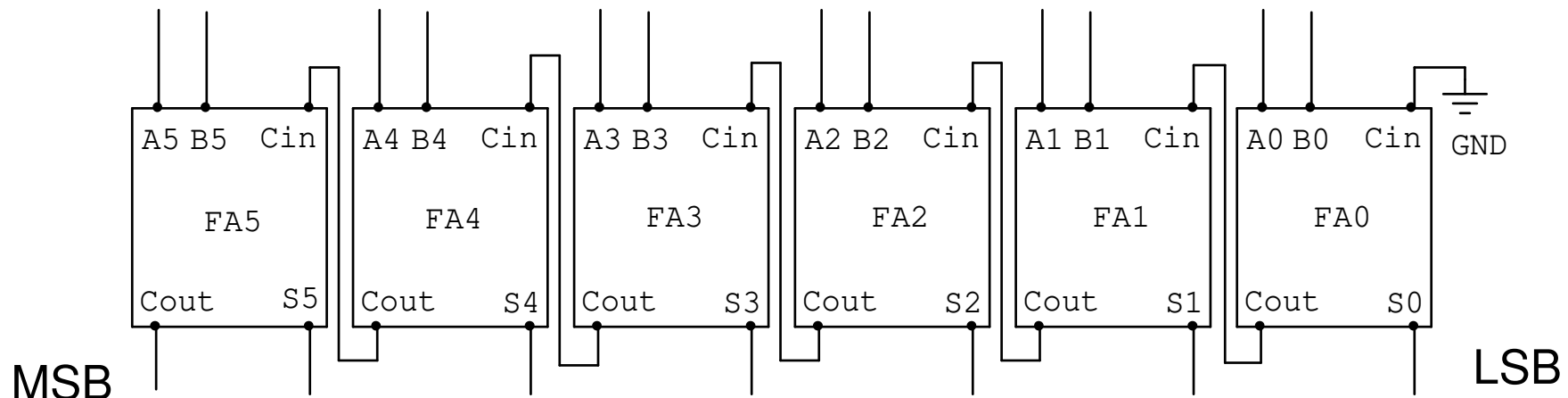
S_i :

		C_{in}			
		B			
A	C_{in}	00	01	11	10
	0	0	1	0	1
A	1	1	0	1	0

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

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

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



- Számítási időszükséglet (RCA):

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

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

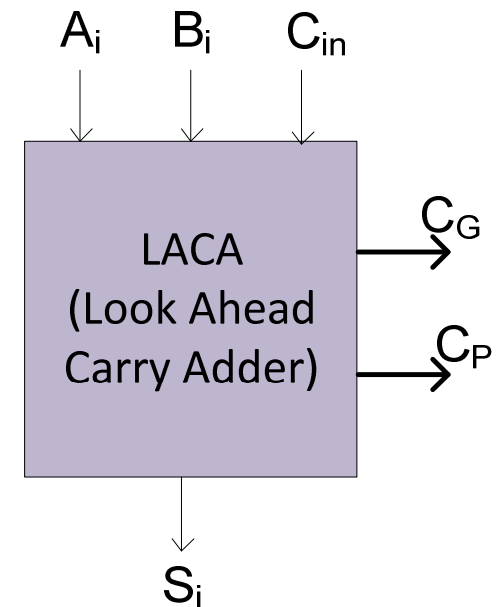
d.) LACA: Look Ahead Carry Adder:

■ Képlet (FA) átalakításából kapjuk:

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

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

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



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

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

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

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

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

igazságtáblázat

A_i	B_i	C_{in}	Sum_i	C_{out}
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

Karnaugh tábla:

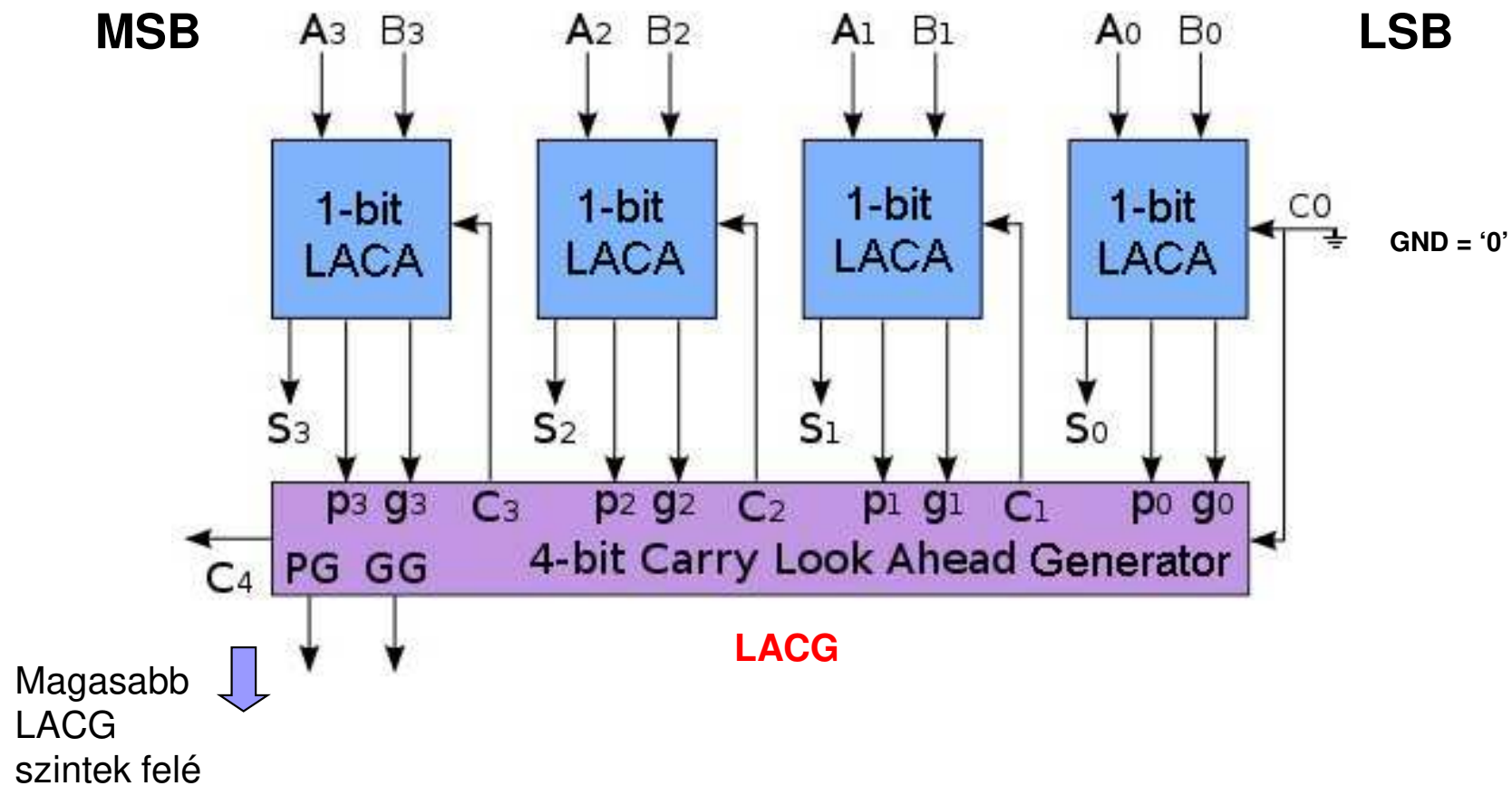
Kimeneti fgv:

		C_{in}			
		B			
A	BC_{in}	00	01	11	10
	C_{out}	0 0	0 1	1 3	0 2
1	A	0 4	1 5	1 7	1 6

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

Példa: 4-bites LACA

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



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

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

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

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

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

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

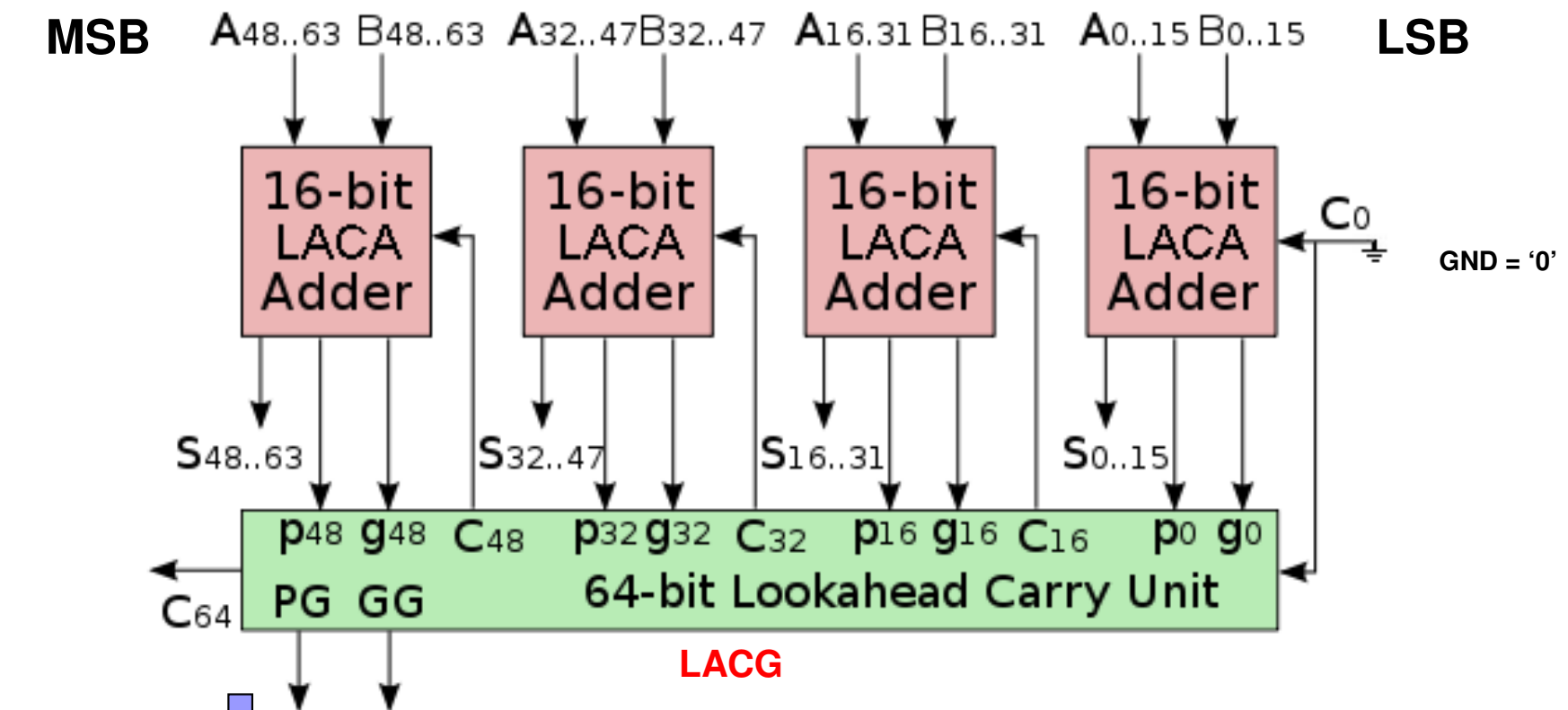
- Magasabb b-bites LACG hierarchia szintek felé GP (group propagate) és GG (group generate) számítása:

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

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

Példa: 4x16-bites LACA

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



Magasabb
LACG
szintek felé

$$T_{LACA} = 2 + 4 \times (\underbrace{\lceil \log_{64}(64) \rceil}_1 - 1) = 2$$

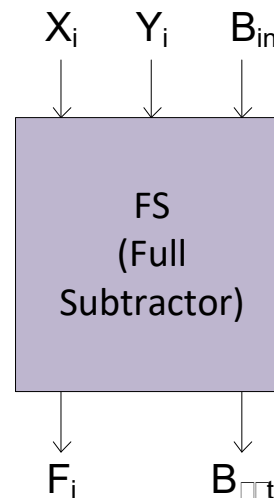
e.) Teljes kivonó - Full Subtractor (FS)

■ FS: 1-bites Full Subtractor

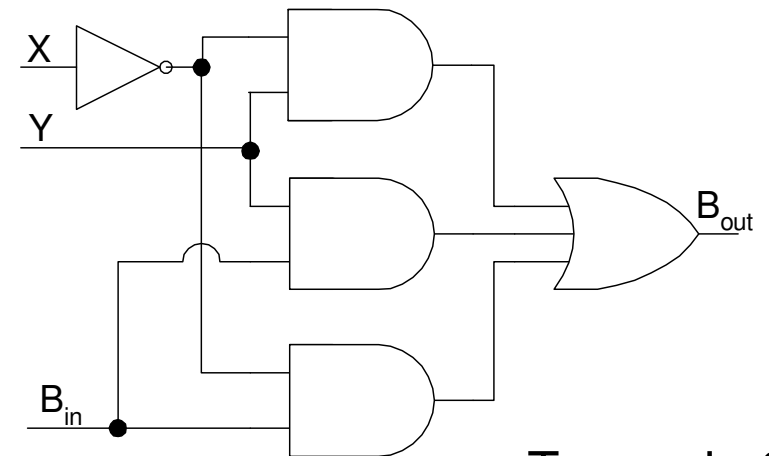
igazságtáblázat

X_i	Y_i	B_{in}	B_{out}	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} :

$X \backslash Y \ B_{in}$	00	01	11	10
0	0	1	1	1
1	0	0	1	0

Kimeneti fgv-ei:

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

F_i :

$X \backslash Y \ B_{in}$	00	01	11	10
0	0	1	0	1
1	1	0	1	0

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

e.) Bináris kivonás végrehajtása (folyt)

■ I. módszer:

Bináris kivonás *FS* segítségével

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

$$\square 0 - 0 \rightarrow 0$$

$$\square 0 - 1 \rightarrow 1, \text{ borrow bit '1'}$$

$$\square 1 - 0 \rightarrow 1$$

$$\square 1 - 1 \rightarrow 0$$

	* * * * *	*
	1 0000000	256
	-001001100	- 76
	010110100	180

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

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

□ FA, RCA, vagy LACA

$$F_i = X_i + 2^n \text{ comp}(Y_i)$$



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

□ Pl: Legyenek:

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

■ Tehát: **N**-bites számok szorzatát **2×N** biten tudjuk eltárolni!

$$\begin{aligned} P &= A \times B = A \times B_4 B_3 B_2 B_1 B_0 = \\ &= A \times B_4 \times 2^4 + A \times B_3 \times 2^3 + A \times B_2 \times 2^2 + A \times B_1 \times 2^1 + A \times B_0 \times 2^0 \end{aligned}$$

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

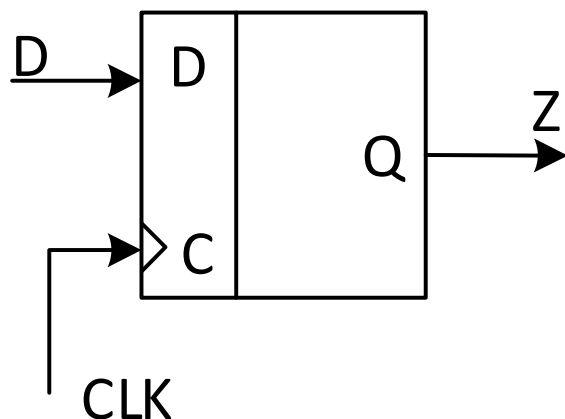
Ismétlés: D flip-flop

■ 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

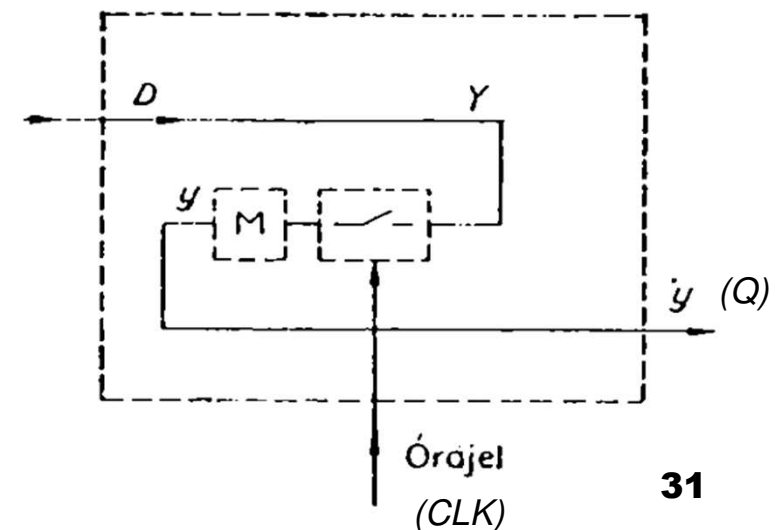


		D	
		0	1
y	0	Q 0	1
	1	0	1

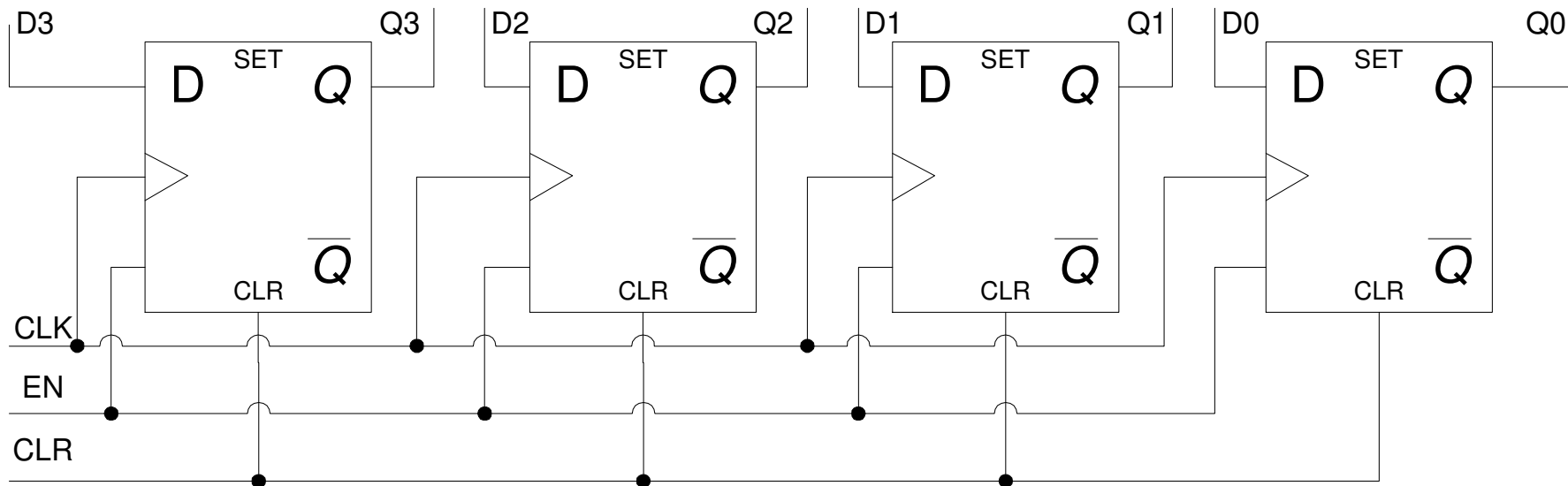
Diagram showing the truth table for the D flip-flop output Q as a function of inputs D and y (current output). The output Q is 0 when (D=0, y=0) and 1 otherwise. A circle groups the cells where Q=1, labeled with 1, 2, and 3.

Egyszerűsített DNF alak és elvi logikai rajz:

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

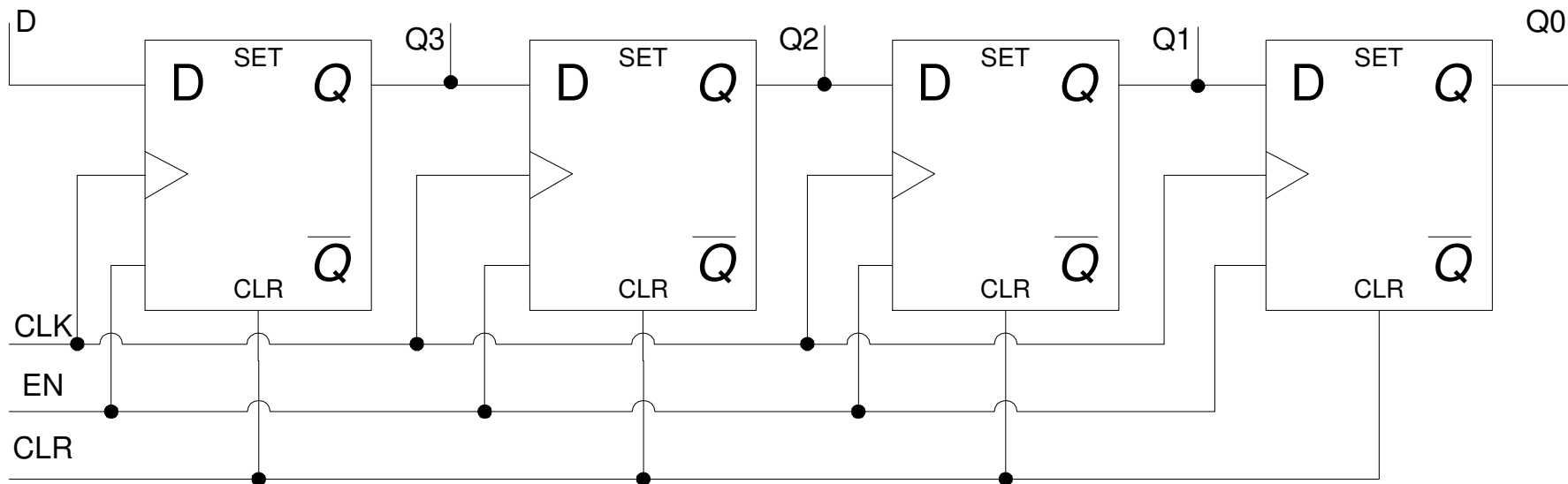


a.) 4-bites Parallel In/ Parallel Out regiszter (D-tárolókból felépítve)



Katalógus adat: [SN54/74LS175](#)

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



Katalógus adat : SN54/74LS95

Iteratív szorzási műveletek:

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

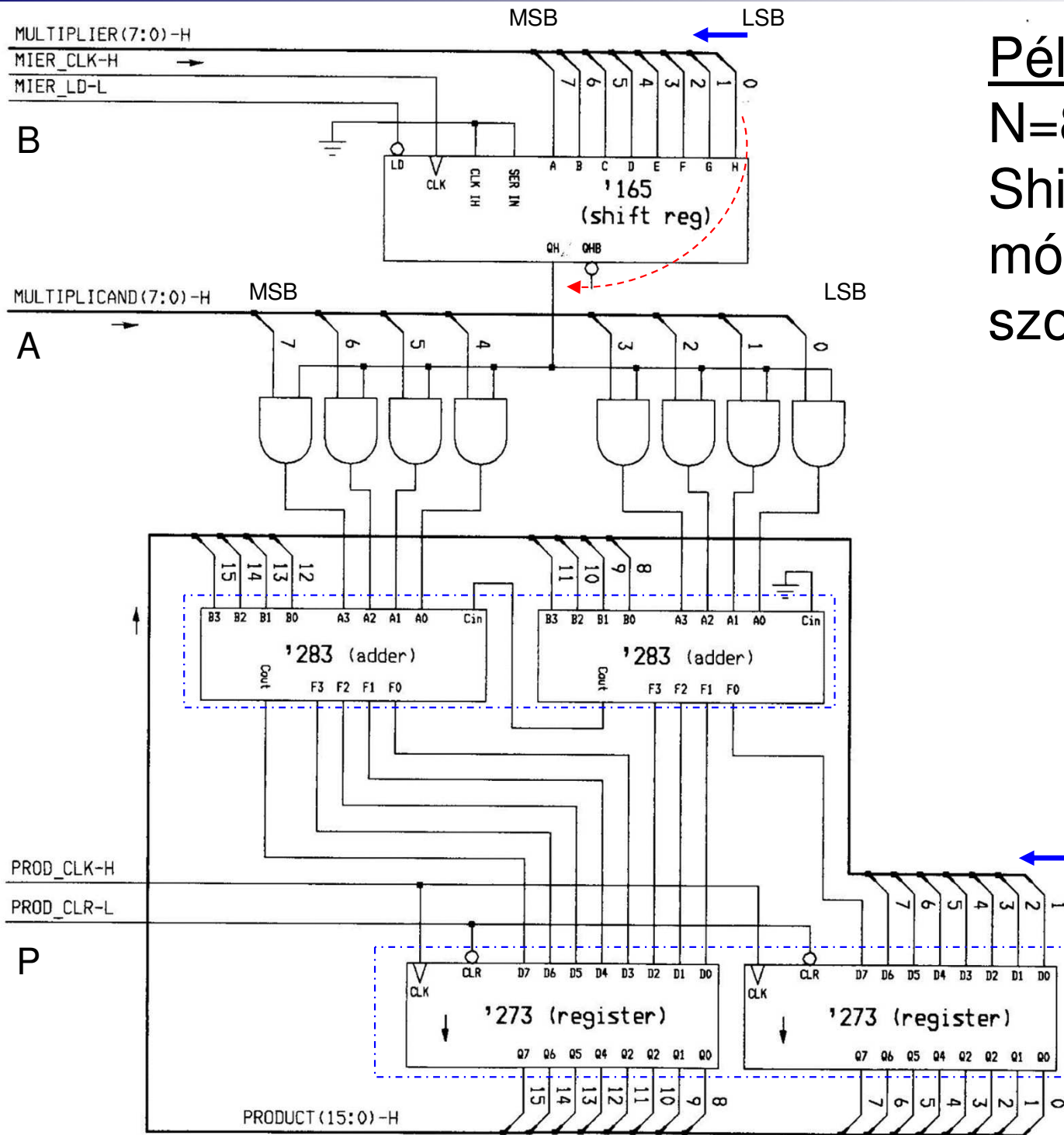
				x	A4 B4	A3 B3	A2 B2	A1 B1	A0 B0
PP0					A4*B0	A3*B0	A2*B0	A1*B0	A0*B0
PP1				A4*B1	A3*B1	A2*B1	A1*B1	A0*B1	
PP2			A4*B2	A3*B2	A2*B2	A1*B2	A0*B2		
PP3		A4*B3	A3*B3	A2*B3	A1*B3	A0*B3			
PP4	A4*B4	A3*B4	A2*B4	A1*B4	A0*B4				
PR				az egyes oszlopok összege					

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

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

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



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 parallel input – serial Shift Register
- '283 – 2db 4-bit Adder: itt helyettesíteni lehetne akár 1db 8-bites Adder-el
- '273 – 2db 8-bit Parallel Register (D-FFs): helyettesíteni lehetne, 1db 16-bites Register-el

Folyamatábra (Shift&Add)

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

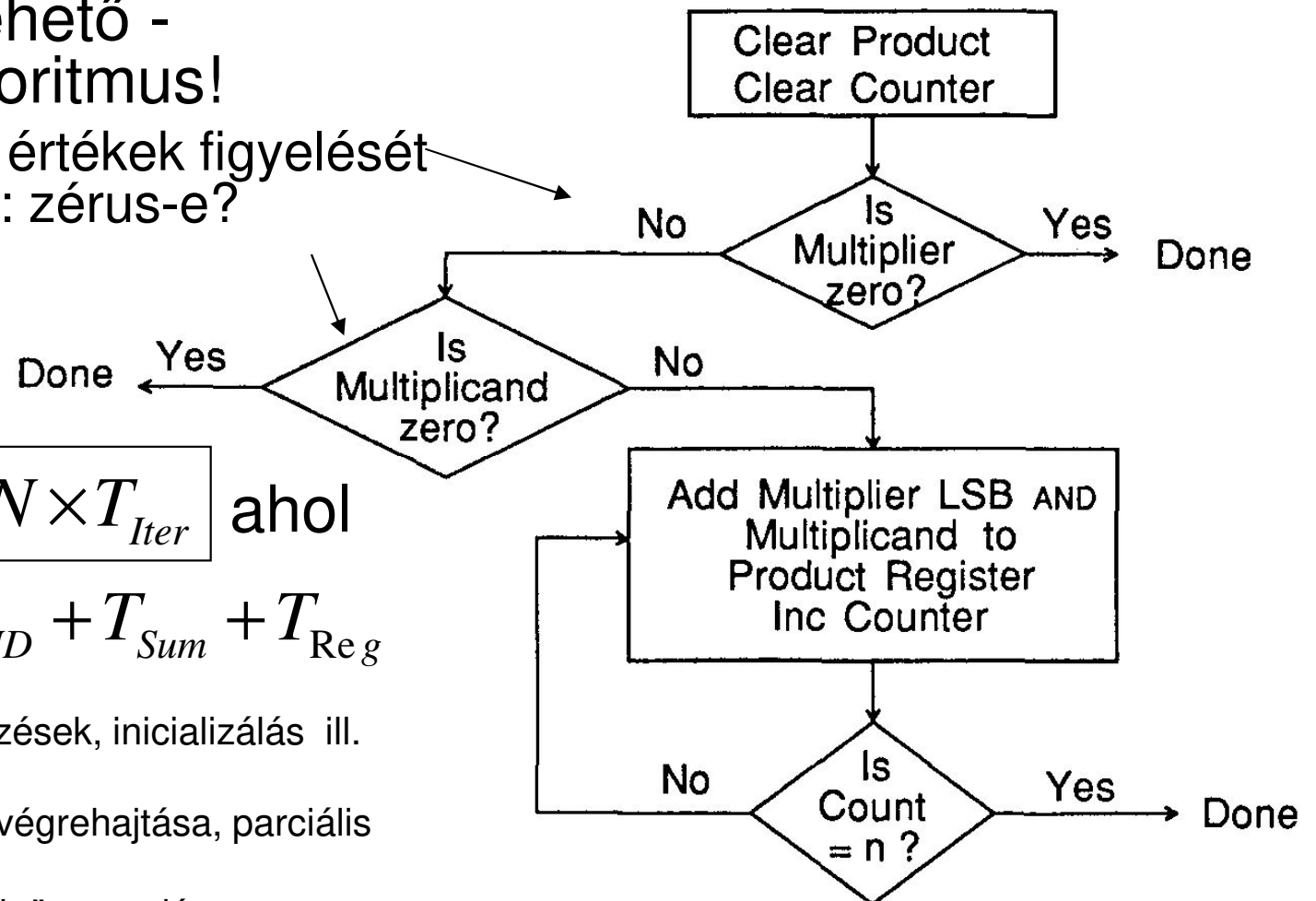
□ Bemenő (A,B) értékek figyelését kell megoldani: zérus-e?

- Időszükséglet:

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

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

- T(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

- $P = A \times B$ (A:szorzandó, B:szorzó)
- Parciális szorzat (PP_i) összegeket itt fordított sorrendben, az **MSB** $\rightarrow$ **LSB** bitek felé haladva képzik
 - Tehát a B szorzó biteket fordított sorrendben tölti be, illetve elsőként az MSB pozíción lévő PP_i értéke kerül a szorzat (P) 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

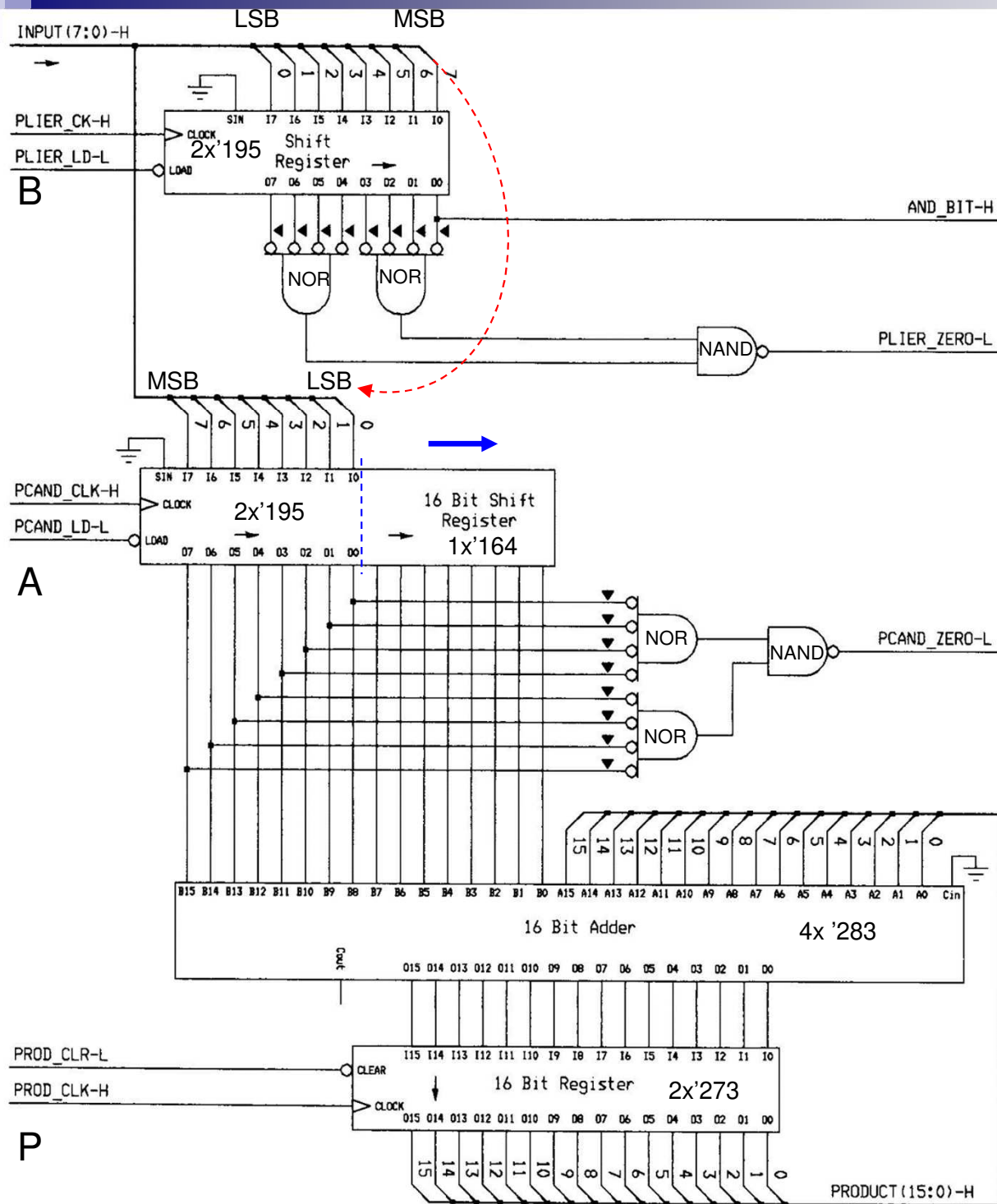
Az *AND_BIT_H* jel feltételesen határozza meg, hogy az „A” szorzandó regiszter (benne a „B” szorzó-regiszter értékeivel) a „P” szorzatregiszterbe másolható-e

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 P_{Pi}-k képzéséhez;
- **Hátránya:** viszont szélesebb összeadó, és A szorzandó reg. kell (szemben a Shift&Add módszerrel).

Folyamatábra (fordított sorrendű)

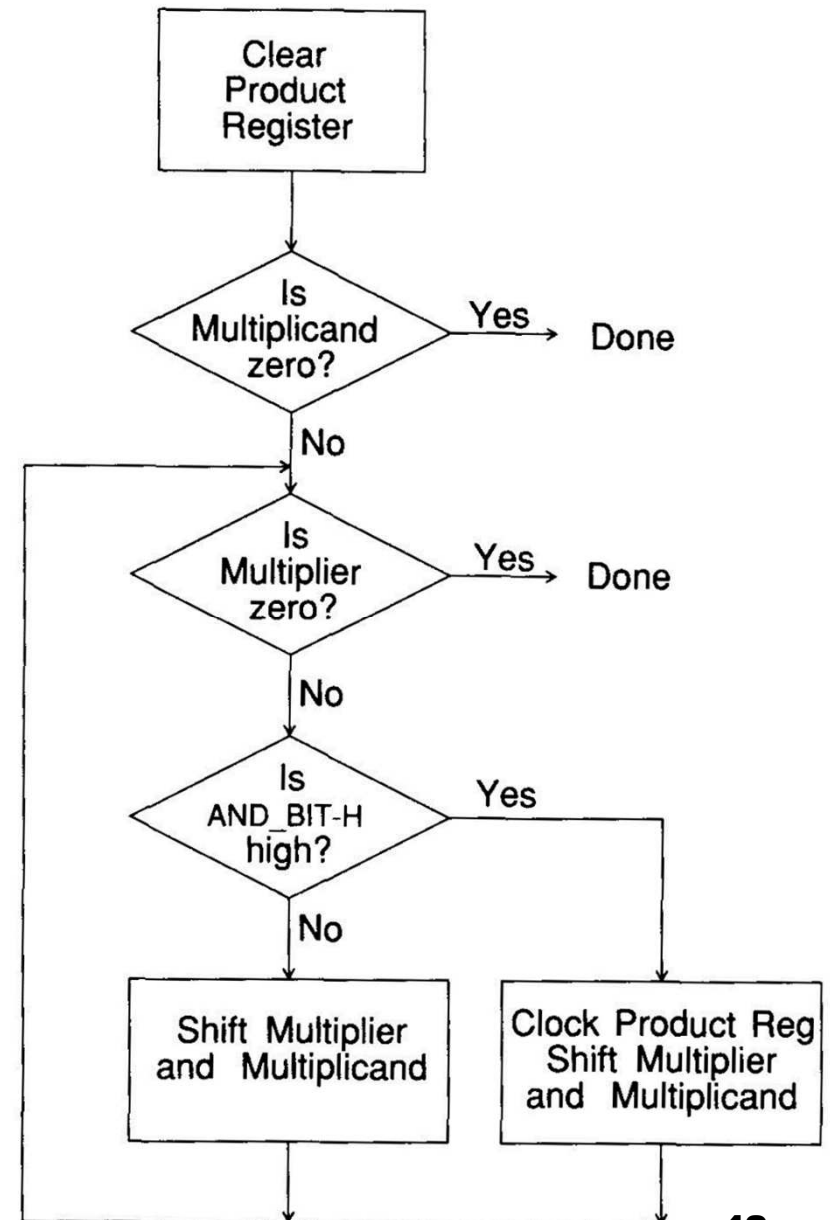
- Adatfüggő algoritmus - gyorsított végrehajtás

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

- Időszükséglet:

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

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



c.) Előjeles szorzás Booth-algoritmussal:

■ *Negatív számokkal* is lehet szorzást végezni!

□ Legyen a következő B 2's komplement 6-bites szám

$$B = B_5 B_4 B_3 B_2 B_1 B_0 = B_5 \times (-2^5) + B_4 \times 2^4 + B_3 \times 2^3 + B_2 \times 2^2 + B_1 \times 2^1 + B_0 \times 1.$$

□ Újrakódolási technika (séma):

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

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)

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

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

Példa: előjeles Booth algoritmus

- Az előző oldalon lévő *átzárójelezéssel* a megfelelő értékeket két bitpár kivonásával kapjuk (a **zárójeles** kifejezés értéke ha **+**: akkor kivonás,/ ha **0**: akkor áteresztés / ha **-**: összeadás történik). A súlytényezők 2 hatványai, és a szorzást a súlytényezők shiftelésével oldják meg ('165 Shift-regiszterrel). //Egy összeadást 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:

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

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

végeredmény értéke

Példa: Booth algoritmus (folyt.)

Végrehajtjuk $P=A \times B$ -t!

543210 (bitpozíciók) $N=6, P=2 \times N$

011001 = 25 „A” Ellenőrzés:

101101 = -19 „B” $P = -475$

// 1110_0010_0101

$$P_0 = 0 - 1 \cdot A = 0 - 1 \cdot 25 = -25$$

$$P_1 = P_0 + 2 \cdot A = -25 + 2 \cdot 25 = 25$$

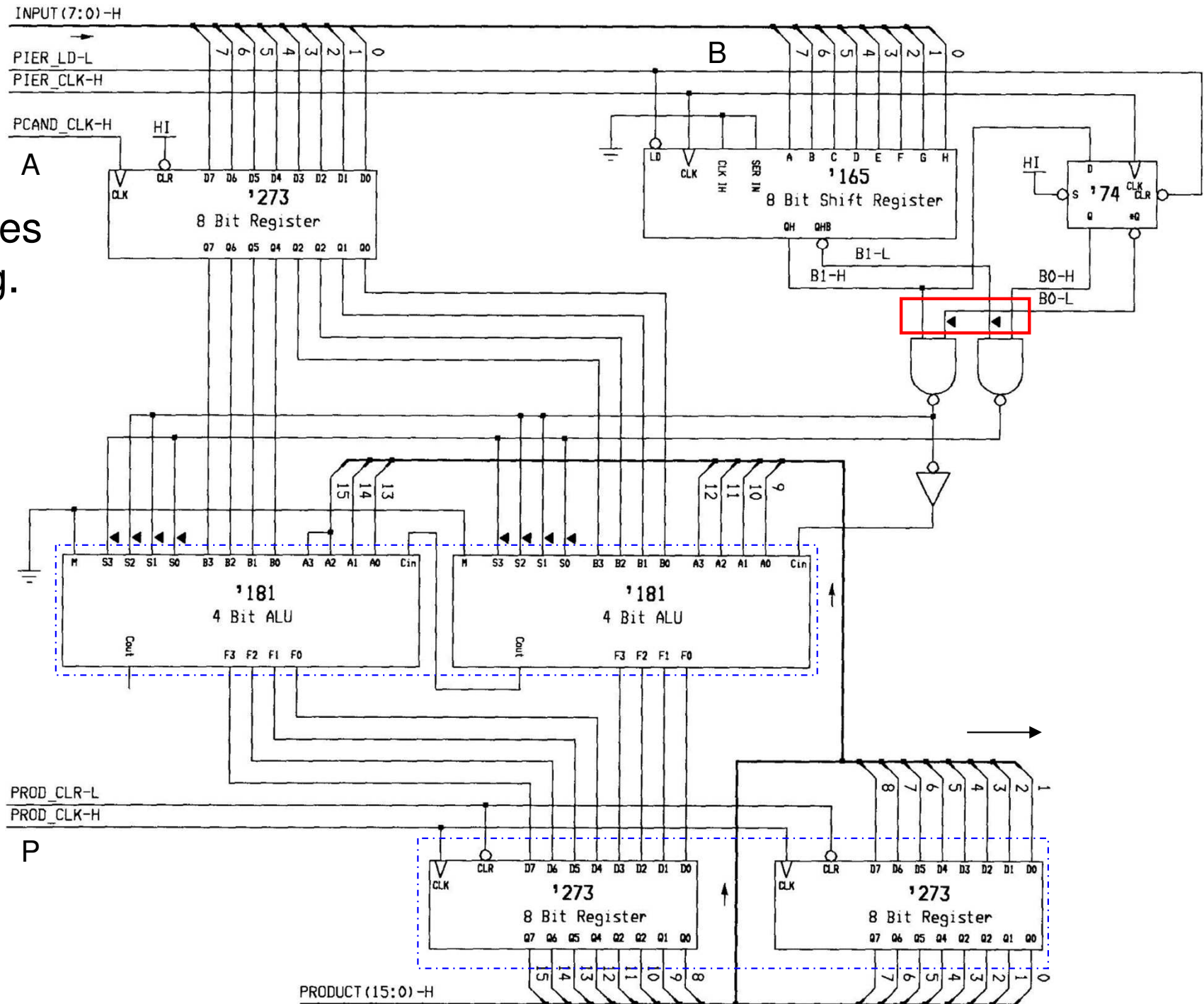
$$P_2 = P_1 - 4 \cdot A = 25 - 4 \cdot 25 = -75$$

$$P_3 = P_2 = -75$$

$$P_4 = P_3 + 16 \cdot A = -75 + 16 \cdot 25 = 325$$

$$P_5 = P_4 - 32 \cdot A = 325 - 32 \cdot 25 = -475$$

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

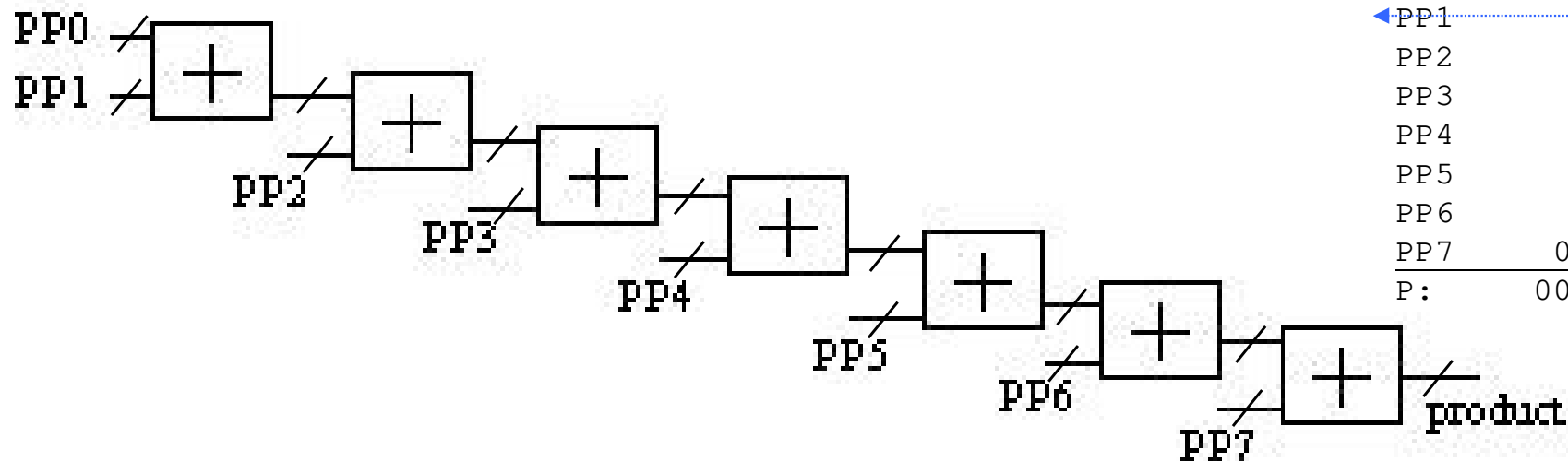
- Először a rész-szorzatokat (Partial Products: PPi) állítják elő, majd pedig azokat összeadják:
 - soronként vagy,
 - oszloponként.
- A rész-szorzatok eltolása egységnyi kapu késleltetéssel megvalósítható.
- Fajtái (rész-szorzat képzés):
 - **Lineáris modell,**
 - **Fa modell,**
 - **Full Adder felhasználásával,**
 - 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:
($T_{\text{Sum}} = N \times T_{\text{FA}}$)

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

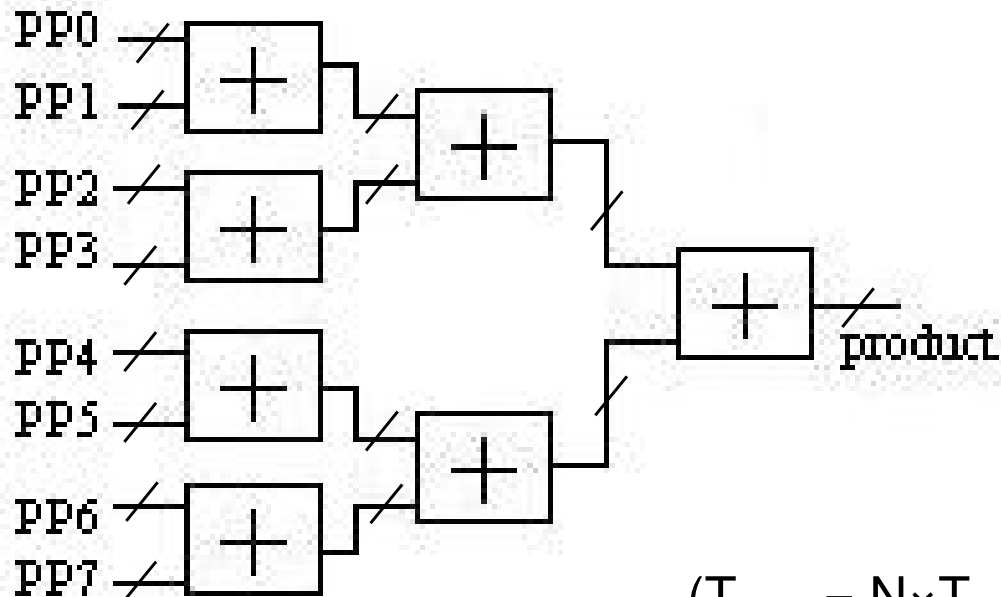


A: Szorzandó	01101001
B: Szorzó	01011010
PP0	00000000
PP1	01101001
PP2	00000000
PP3	01101001
PP4	01101001
PP5	00000000
PP6	01101001
PP7	00000000
P:	0010010011101010

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: $T_{DIRECT-TREE} = \lceil \log_2(N) \rceil * T_{SUM}$

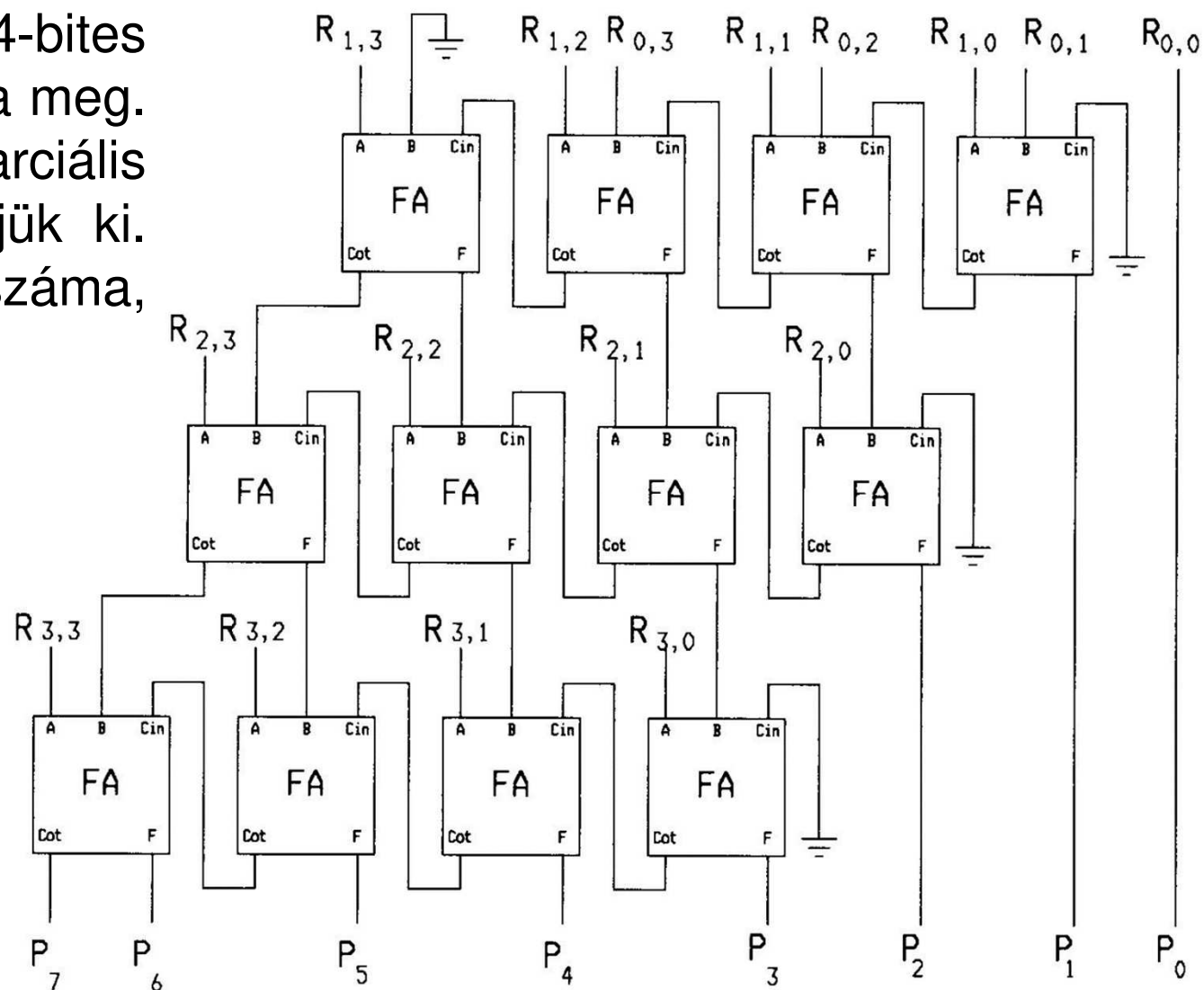
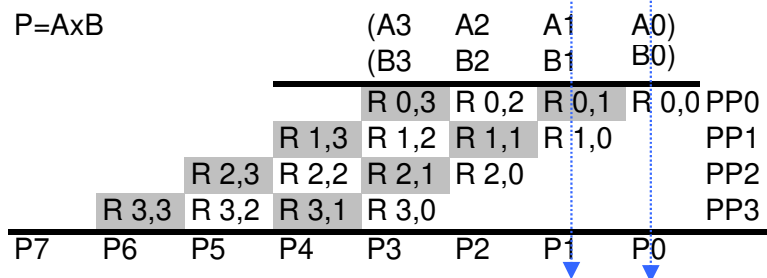


$$(T_{Sum} = N \times T_{FA})$$

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

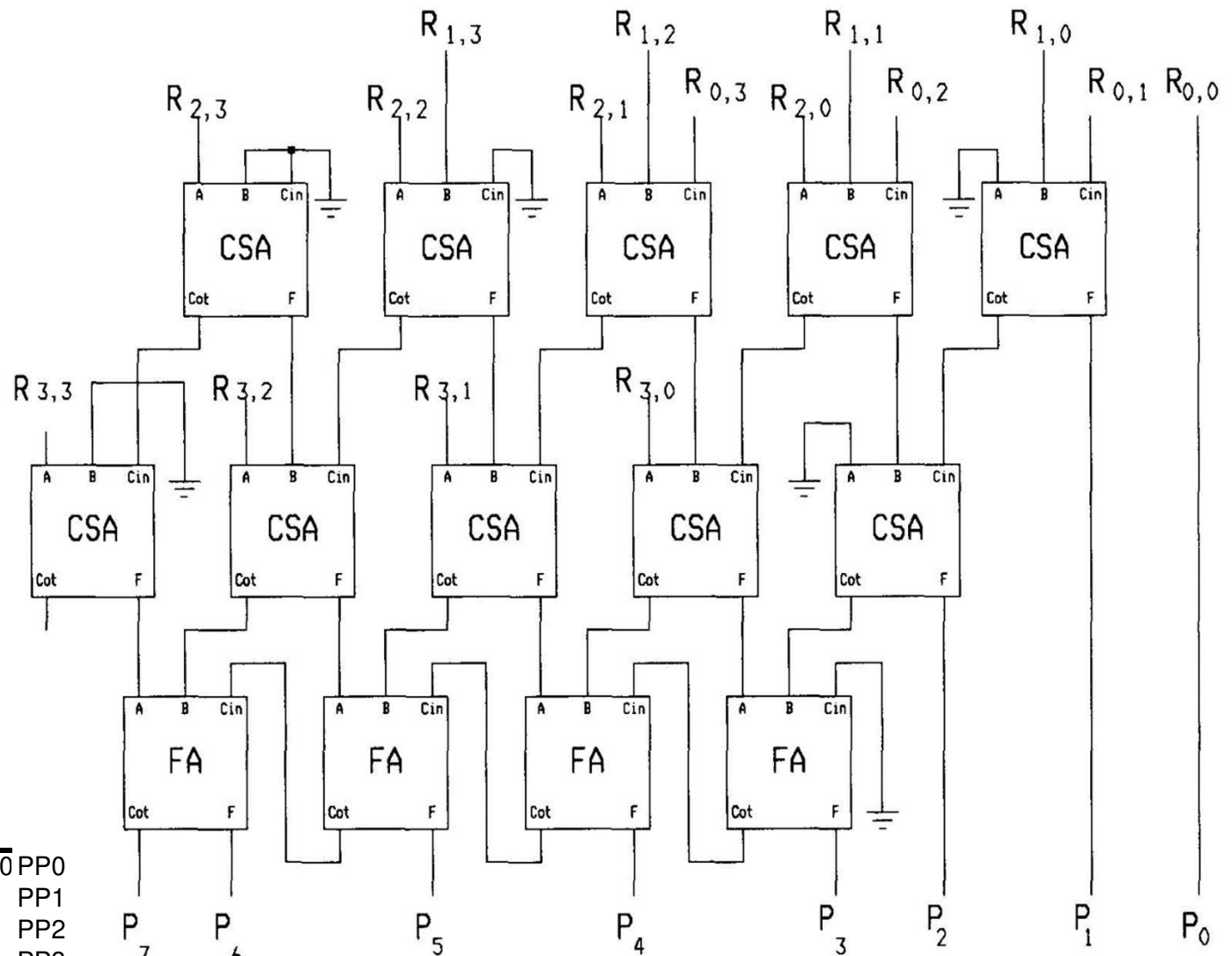
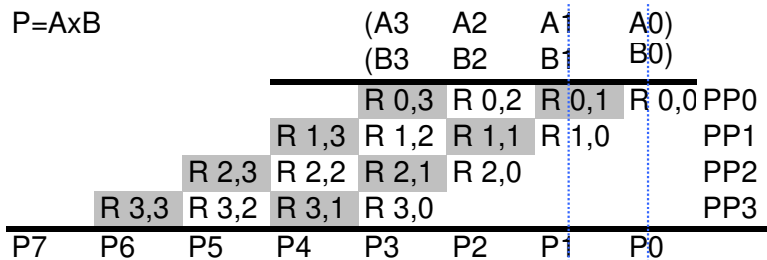
- A mellékelt ábra két 4-bites szám szorzását valósítja meg. A sorokat, mint parciális szorzat tömböket emeljük ki. Jel: $R_{x,y}$, ahol x a sor száma, y a sor eleme (oszlop).



d.) CSA: Carry Save Adder

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

•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).



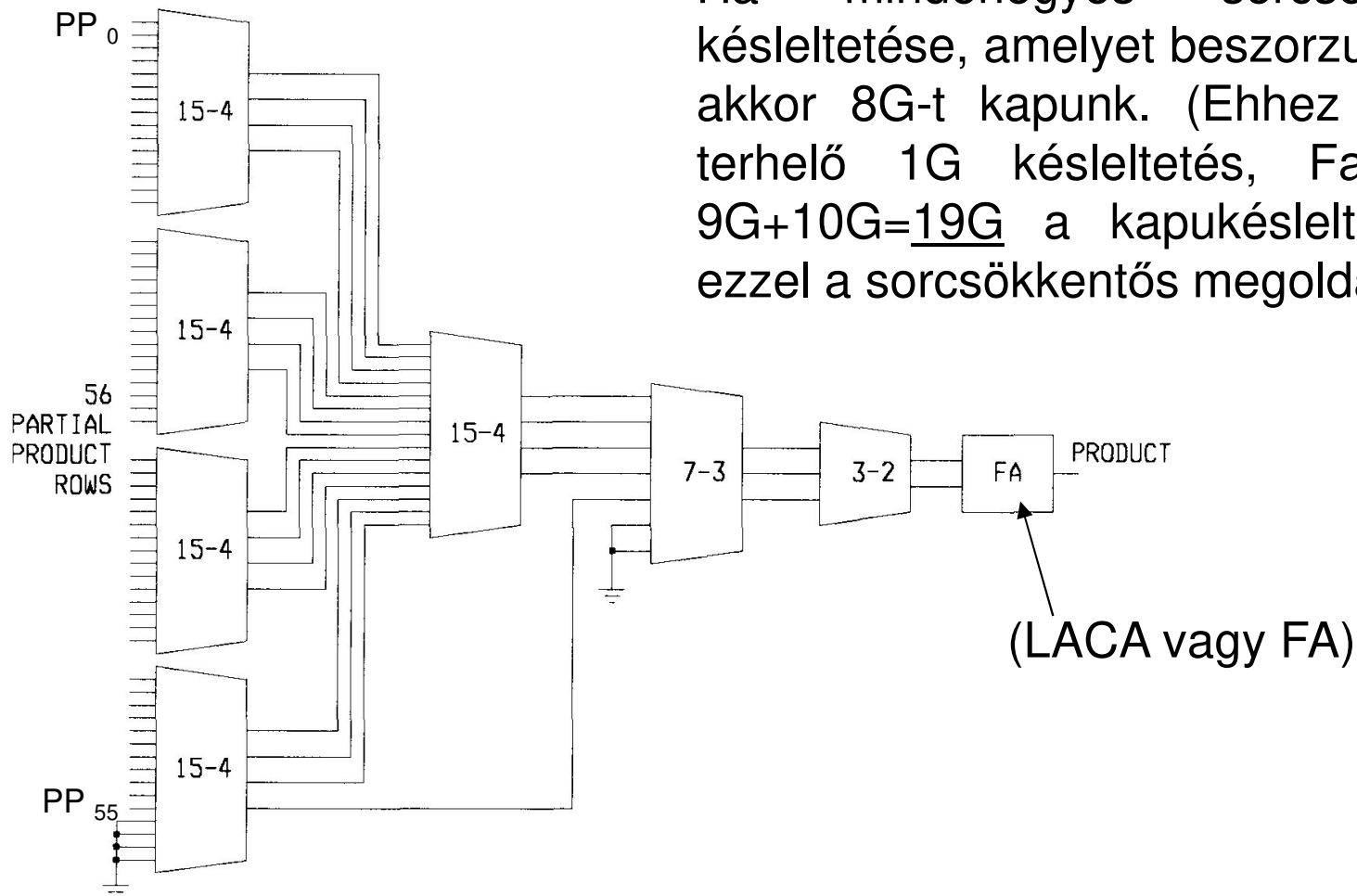
e.) Sorcsökkentős megvalósítás (row reduction)

- 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 PI. teljes összeadó (FA) vagy LACA összeadó használható.
- Most 56×56 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-in.) Így összesen $9G+10G=\underline{19G}$ a kapukésleltetése az eszköznek, ezzel a sorcsökkentős megoldással.





Osztó áramkörök

Osztó áramkörök:

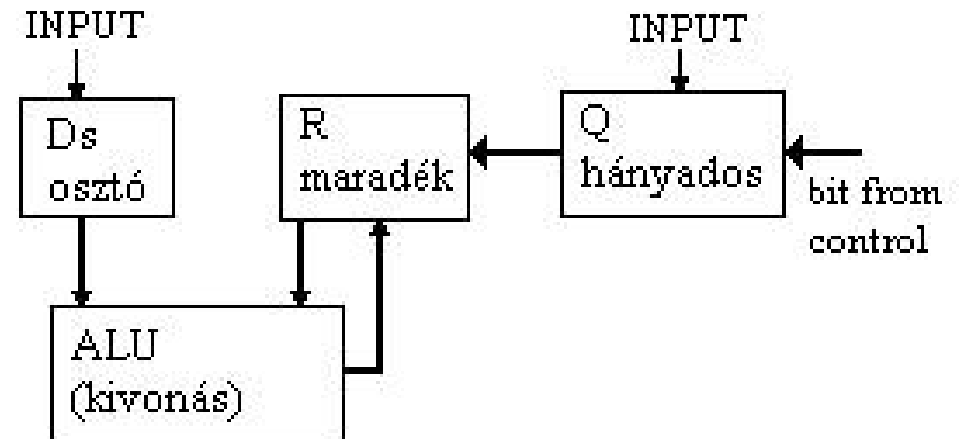
- I.) Hagyományos – „lassú” vagy közvetlen iteratív osztási algoritmus
 - Euklideszi osztás
- II.) „Gyors” - iteratív osztási algoritmusok (egyéb numerikus módszerek)

I.) Hagyományos közvetlen osztási algoritmus:

Ez az osztási folyamat igen lassú eljárás. Lépései:

1. az *osztót* a D_s regiszterbe rakjuk, az *osztandót* (D_d) pedig a Q regiszterbe.
2. töröljük az R regisztert
3. iterációs lépés: kivonjuk az R -ből a D_s osztót.
Ha $R - D_s > 0$ akkor folytatódik, tehát ezt a megváltozott értéket visszateszük az R -be, és egy '1'-est teszünk a Q regiszterbe. Ha $R - D_s < 0$ akkor R regiszter tartalma nem változik, és egy '0'-át teszünk a Q regiszterbe (vagyis hogyha nincs több osztandó bit, akkor vége az osztásnak).
4. Minden iterációs lépésben egy-egy új bit jön létre, amelyet a Q regiszterbe shiftelünk, miközben az R regiszterbe az osztandót (D_d).
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!

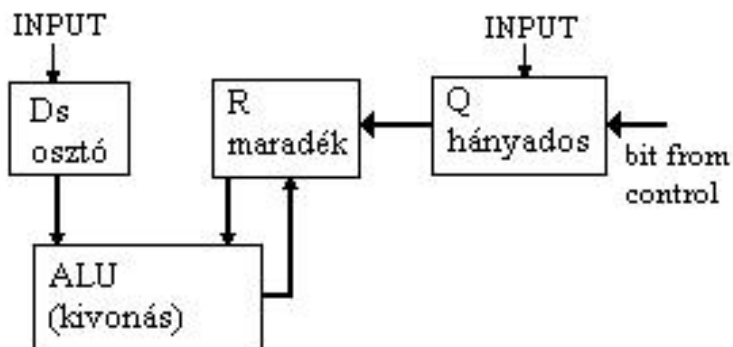
Decimális számok esetén:

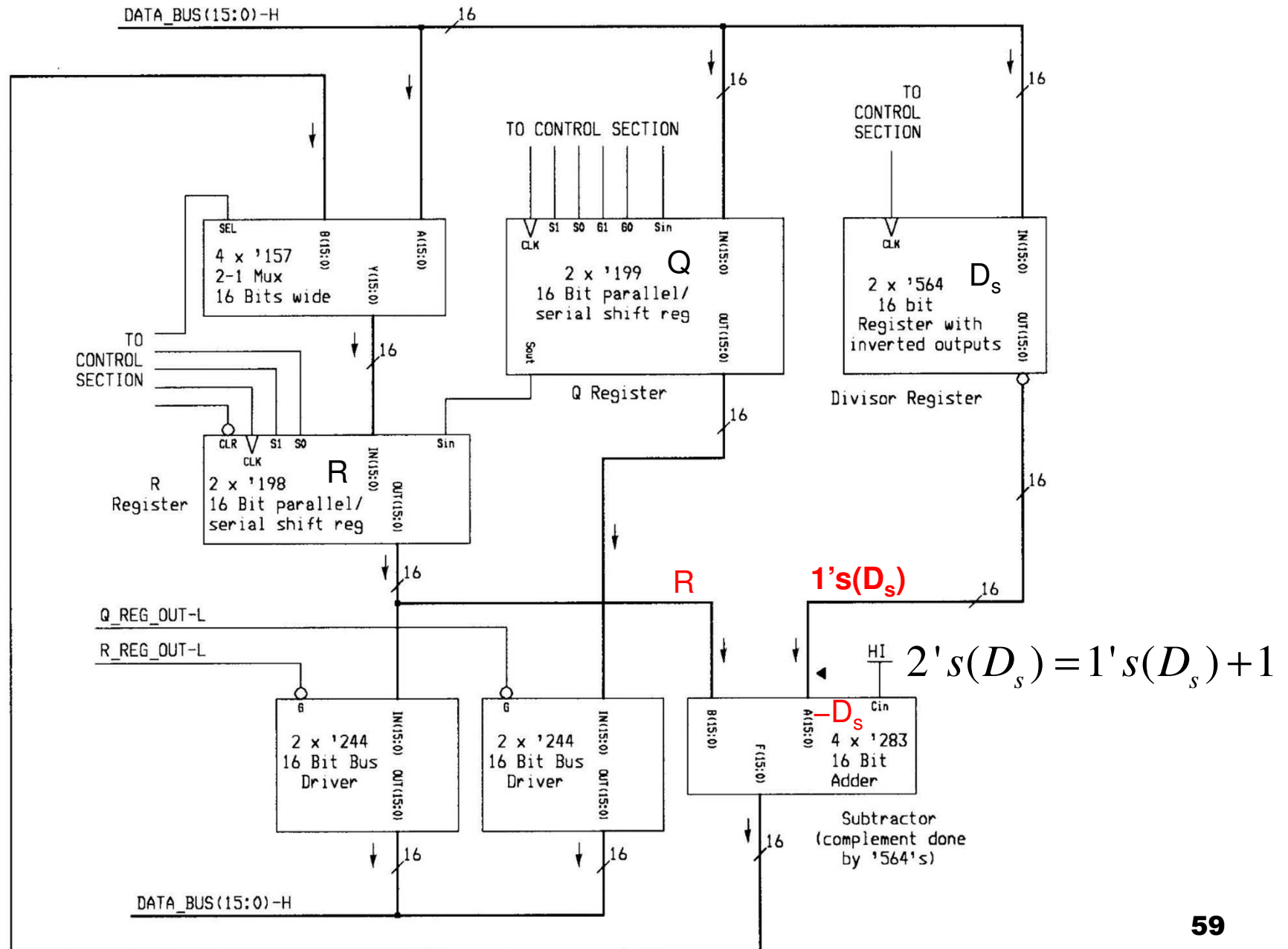
$$\begin{array}{r} 5 \overline{) 8} : 5 = 1 \overline{) 1} \\ 0 \ 8 \\ \underline{3} \end{array}$$

Bináris számok esetén hasonlóan

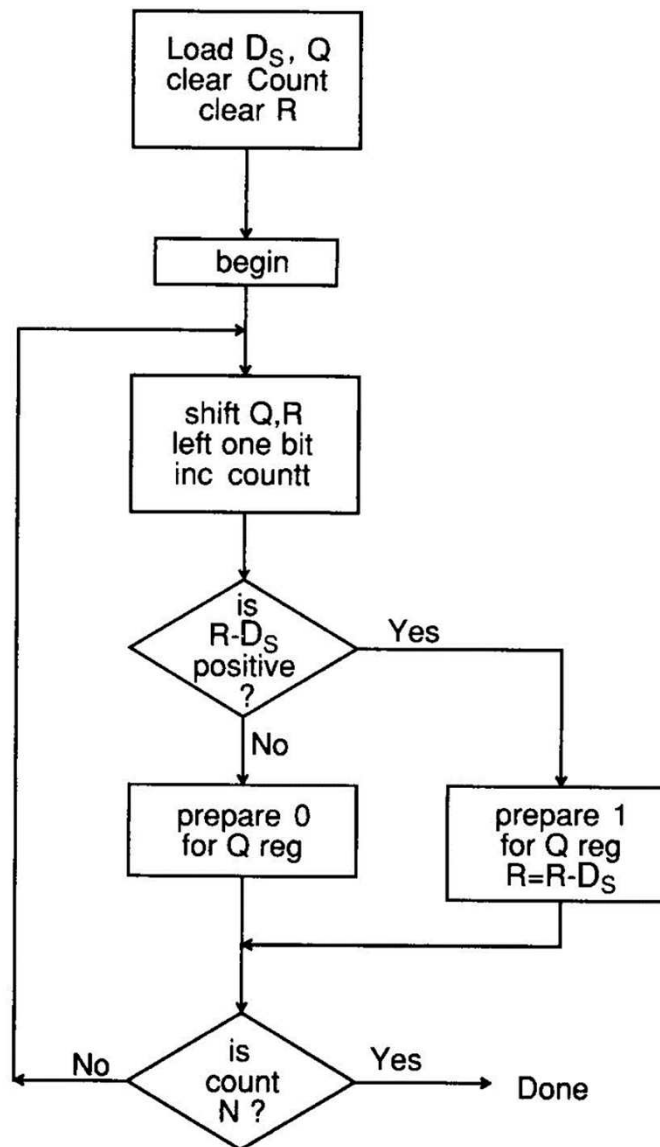
1011 Q hányados (Ds hányszor van meg Dd-ben)

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!





Folyamatábra: osztási algoritmus





II.) Iteratív osztási algoritmus

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

a.) Gyors osztás Newton- Raphson módszerrel

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

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

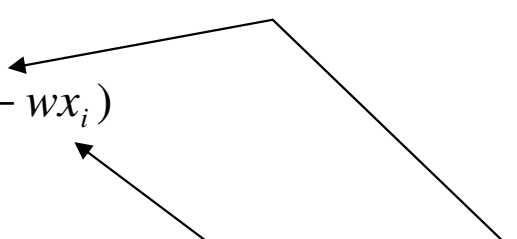
Van egy megfelelő f függvényünk és egy x_0 kezdeti értékünk. Iterációs lépésekkel megkapjuk az osztás eredményét az $f(x)=0$ egyenlet megoldásaként. Az f -et úgy kell (jól) megválasztanunk, hogy a reciprok gyökkel rendelkezzen. Legyen

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

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 $A \cdot (1/B)$ 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:

Várt érték:
24.73863375370...

x_1	$=$	$x_0 - \frac{f(x_0)}{f'(x_0)}$	$=$	$10 - \frac{10^2 - 612}{2 \cdot 10}$	$=$	35.6
x_2	$=$	$x_1 - \frac{f(x_1)}{f'(x_1)}$	$=$	$35.6 - \frac{35.6^2 - 612}{2 \cdot 35.6}$	$=$	<u>26.3955056</u>
x_3	$=$	$\vdots$	$=$	$\vdots$	$=$	<u>24.7906355</u>
x_4	$=$	$\vdots$	$=$	$\vdots$	$=$	<u>24.7386883</u>
x_5	$=$	$\vdots$	$=$	$\vdots$	$=$	<u>24.7386338</u>

Aláhúzások,
már a korrekt
számjegyeket
jelölik, az
egy-
iterációkban

b.) Közvetlen gyors osztó

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

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

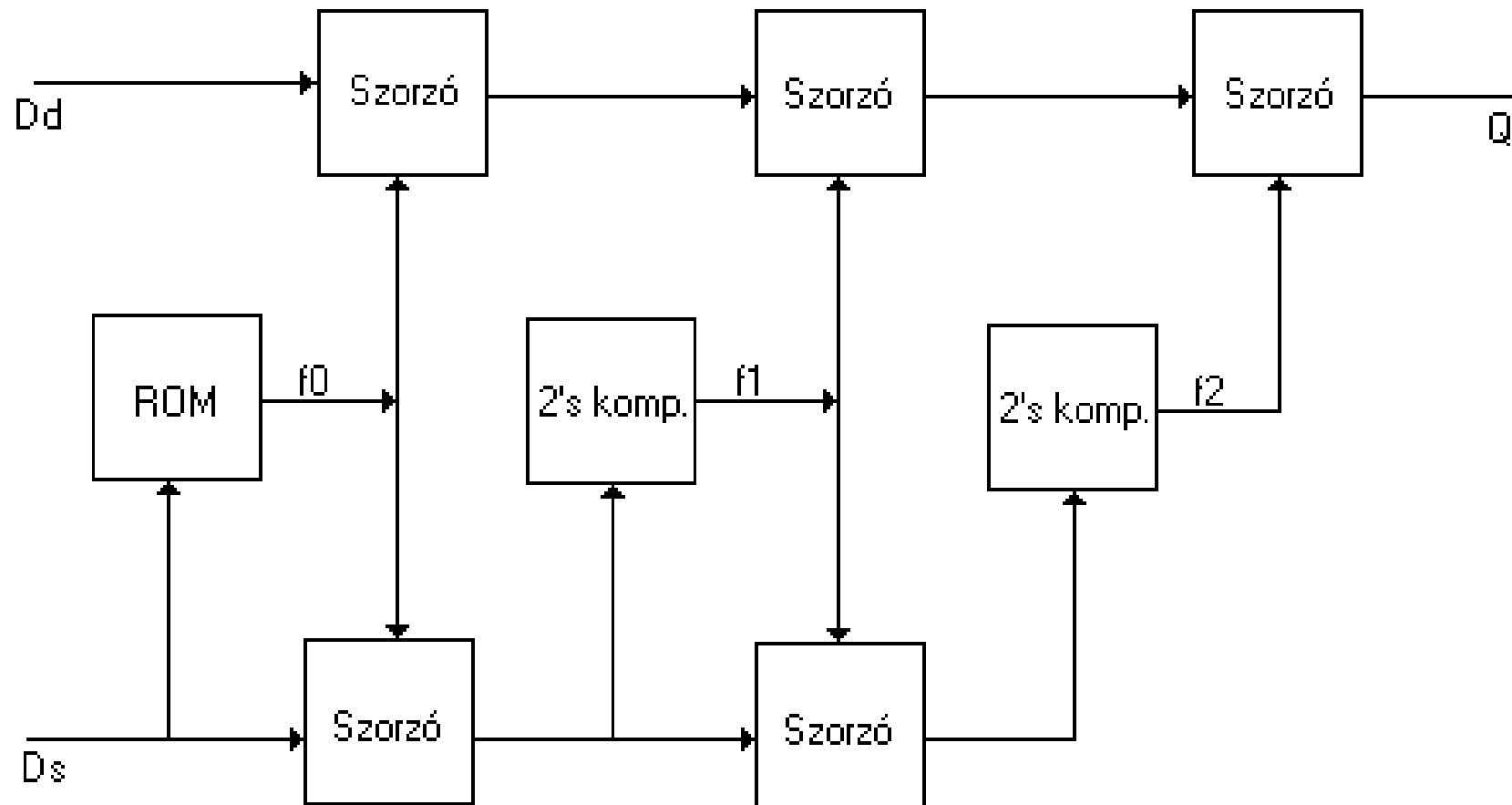
Közvetlen gyors osztó működése

- A számláló iterációja ennél a módszernél $D_{Dn+1} = D_{Dn} \times f_n$. A nevező iterációját a megismert módon használjuk a következő f -ek meghatározására.
- Tegyük fel, hogy szorzóink, 2'komplementes 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 törtekifejezésként írjuk fel (mantissza egy normalizált tört).
- Keressük a Q hányados (quotient) értékét. Hogy megkapjuk, mind az osztó, mind pedig az osztandó értékét ugyanazokkal az f_k értékekkel kell megszorozni, amelyet úgy határozzunk meg, hogy a nevező egységnyi legyen az iterációk elvégzése után. Így később a számláló értékéből megkapjuk a Q pontos értékét. Tudjuk, hogy D_S normalizált tört, ezért így ábrázoljuk: $D_S = 1-x$, ahol x -et D_S határozza meg, és mivel D_S kisebb 1-nél, így az x is kisebb 1-nél.

Közvetlen gyors osztó (számítás)

- Az osztás művelete f_0 kiszámolásával kezdődik. Válasszuk $f_0 = 1+x = 1+(1-D_S) = 2-D_S$. Így $D_S \times f_0 = (1-x)(1+x) = 1-x^2$. 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 tényezőikkel szorzódik, és közelebb kerülünk a Q pontos értékéhez. Legyen $f_1 = 1+x^2$. Így $D_S \times f_0 \times f_1 = 1-x^4$ és ez tovább ismételtető iteratív módon
- Tehát azt kapjuk, hogy $D_{Dn+1} = D_{Dn} \times f_n$.
- Egy kérdés vetődik fel: hogyan válasszuk meg f_k következő értékét. $f_1 = 1+x^2 = 1+(1-D_S \times f_0) = 2-D_S \times f_0$. Tehát minden egyes új f_k -t úgy kapunk meg, hogy vesszük az f_{k-1} és a D_S (nevező) szorzatának 2's komplementjét. Az iterációs lépéseket a kívánt pontosság eléréséig kell ismételni, amelyet f_k értéke határoz meg. Amikor f_k közelítőleg 1, akkor a Q eredmény elegendően közel lesz a kívánt eredményhez (amely az alkalmazástól és a bitek számától függ). Általában előre definiált fix számú iterációs lépést végzünk el. Ezért kell ROM-ot használni, amelyben az f_0 megfelelő kezdeti értékét tároljuk.
- (Példák: könyvben)

Közvetlen gyors osztó áramkörü felépítése



Példa 1.

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

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

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

Várt érték:

0.57142857142857142857142857142857

- 0. $D_{D0}=0,1000000$ $D_{S0}=0,1500000$ $f_0=1,8500000$
- 1. $D_{D1}=0,1850000$ $D_{S1}=0,2775000$ $f_1=1,7224999$
- 2. $D_{D2}=0,3186625$ $D_{S2}=0,4779938$ $f_2=1,5220062$
- 3. $D_{D3}=0,4850063$ $D_{S3}=0,7275094$ $f_3=1,2724905$
- 4. $D_{D4}=0,6171659$ $D_{S4}=0,9257489$ $f_4=1,0742511$
- 5. $D_{D5}=0,6629912$ $D_{S5}=0,9944868$ $f_5=1,0055132$
- 6. $D_{D6}=0,6666464$ $D_{S6}=0,9999696$

Látható, hogy itt nem kapjuk meg a kívánt értéket ($D_{D6}=0,6666464$) 6 iterációs lépés alatt. Ezért hogy elérjük a kívánt pontosságot véges számú lépés alatt, ROM-ot kell használni (ahol f_0 kezdeti értékét tároljuk).

0.66666666666666666666666666666667

Extra bitek kezelése

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

Extra bit: probléma

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

Nagyobb számhoz tartozó mantissza 101010

Kisebb számhoz tartozó mantissza + 110010

$$\begin{array}{r} 101010 \\ + 110010 \\ \hline 11011010 \end{array} \rightarrow 2$$

Extra bits?

Két bitpozícióval eltolt mantissza, hogy megegyezzen a nagyobb exponensének értékével

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:

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

- **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

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

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

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

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

b.) Rounding (kerekítés)

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

Nagyobb mantissza

101010

Kisebb mantissza

+ 110010

11011010

+00000010

11011100



2 bitpoz. igazítva

8 bites eredmény

$\frac{1}{2}$ LSB pozíció
(=rounding!)

Végeredmény (majd
truncate!)

Extra bits

PI: Rounding (kerekítés)

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

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

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

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

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. Két ilyen eset van most:

case	MR	$M_{R+1/2}$ LSB	truncated MF	ERR _{ZERO}
a	xx0.00	xx0.10	xx0.	0.00
b	xx0.01	xx0.11	xx0.	+0.01
c	xx0.10	xx1.00	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	xx1.00	xx1.	+0.10
h	xx1.11	xx1.11	xy0.	-0.01

Eltérés a Rounding-hoz képest!

$$\text{Bias: } \sum \text{ERR}_{\text{Round to Zero}} = 0 !$$

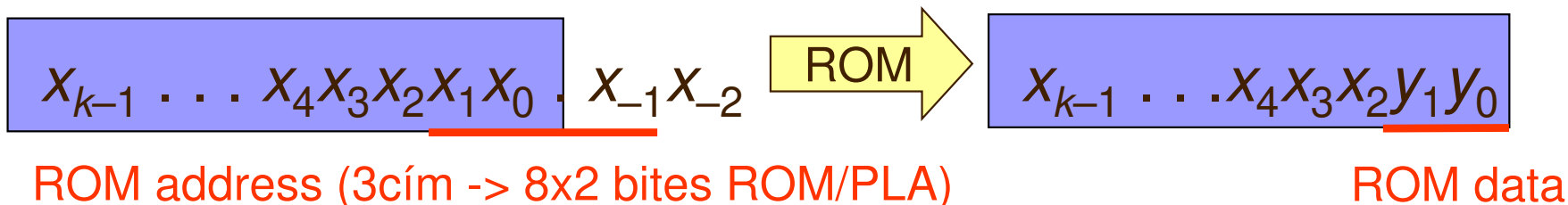
Normál kerekítéshez képest a **c.)/g.)** eseteknél egy ‘1’-es lett **direkt beállítva** (force) az LSB helyén (xx1). De csak a **g.)** esetnél lesz más a hiba értéke (ERR_{ZERO}).

d.) Jamming (~fix értéken rögzítés)

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

e.) ROM rounding

- Extra bitek vizsgálata: *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 értéket használnak.
- Biztosítja, hogy az LSB-nél nagyobb bitpozíciókba nem kell „carry-t propagáltatni” (mint rounding-nál): ezáltal gyorsabb
- Bias jól kontrollálható (akár zérus is lehet végül).



Extra-bit kezelő módszerek hibáinak összehasonlítása:

