Pannon Egyetem
Képfeldolgozás és Neuroszámítógépek Tanszék

Digitális Rendszerek és Számítógép Architektúrák

4. előadás: Aritmetikai egységek - adatkezelés

Előadó: Dr. Szolgay Péter
Vörösházi Zsolt

Jegyzetek, segédanyagok:

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

Ismétlés

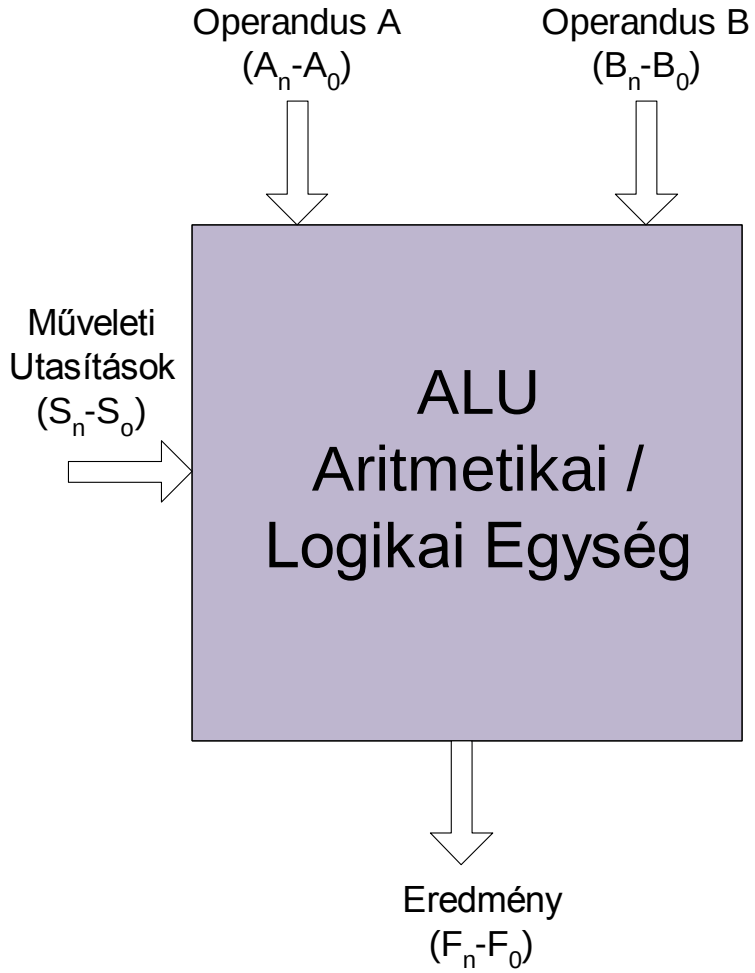
- Korai számítógépek teljesítményét főként ballisztikus számításoknál (hadászatban)
- Információ ábrázolás
- Használt utasításkészlet
- Adatkezelő / műveletvégző egység:
  - Alapvető ALU (Aritmetikai és Logikai funkciók)
    - Aritmetikai operátorok: +,-,*,/ (alapműveletek)
    - Logikai operátorok: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

Tervezői célkitűzés

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

ALU felépítése

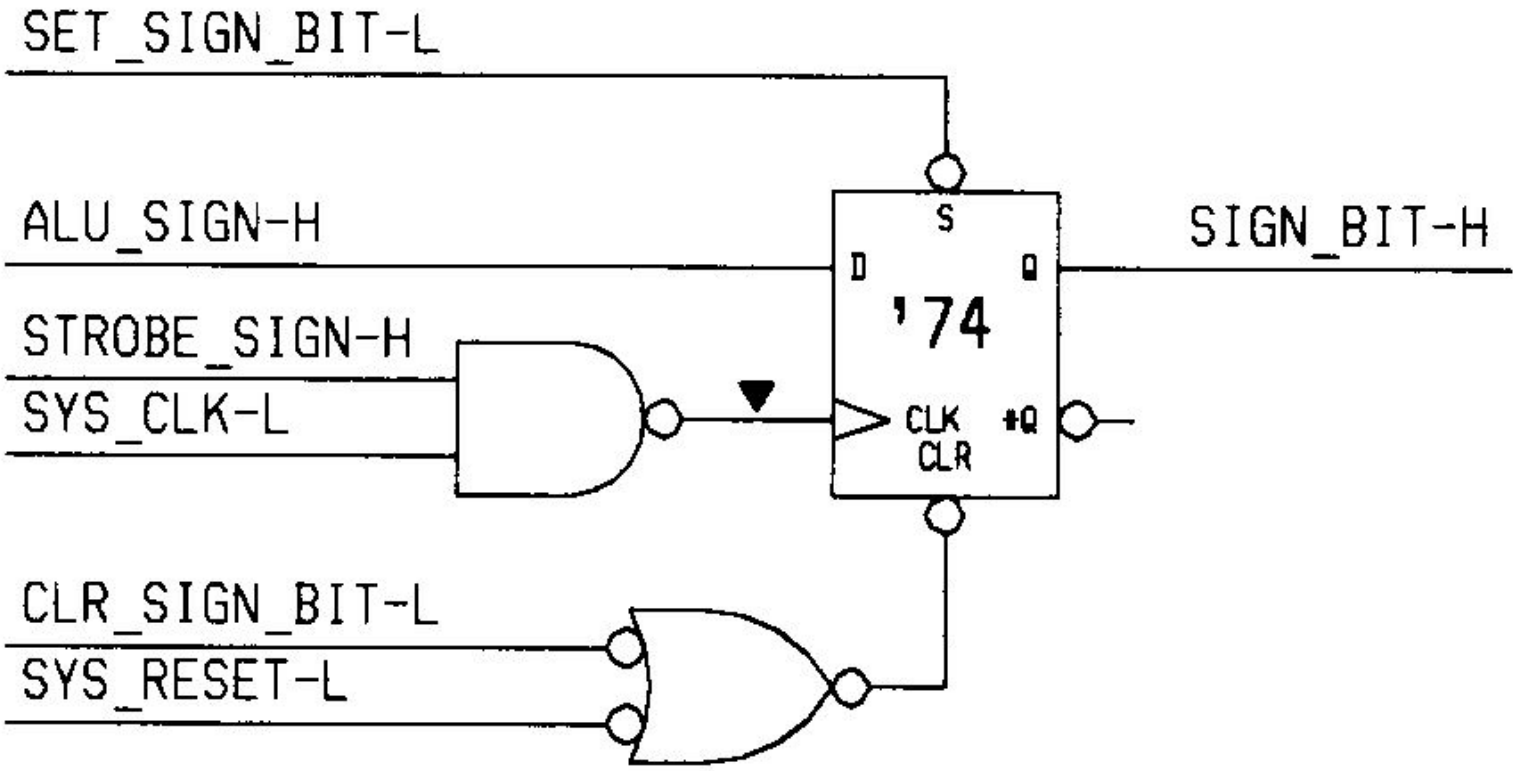
- Utasítások hatására a (S_n-S_0) vezérlőjelek kijelölik a végrehajtandó aritmetikai / logikai műveletet. További adatvonalak kapcsolódhatnak közvetlenül a státusz regiszterhez, amely fontos információkat tárol el: pl.
  - *zero bit*
  - *carry-in, carry-out* átviteleket,
  - *előjel* bitet (sign),
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



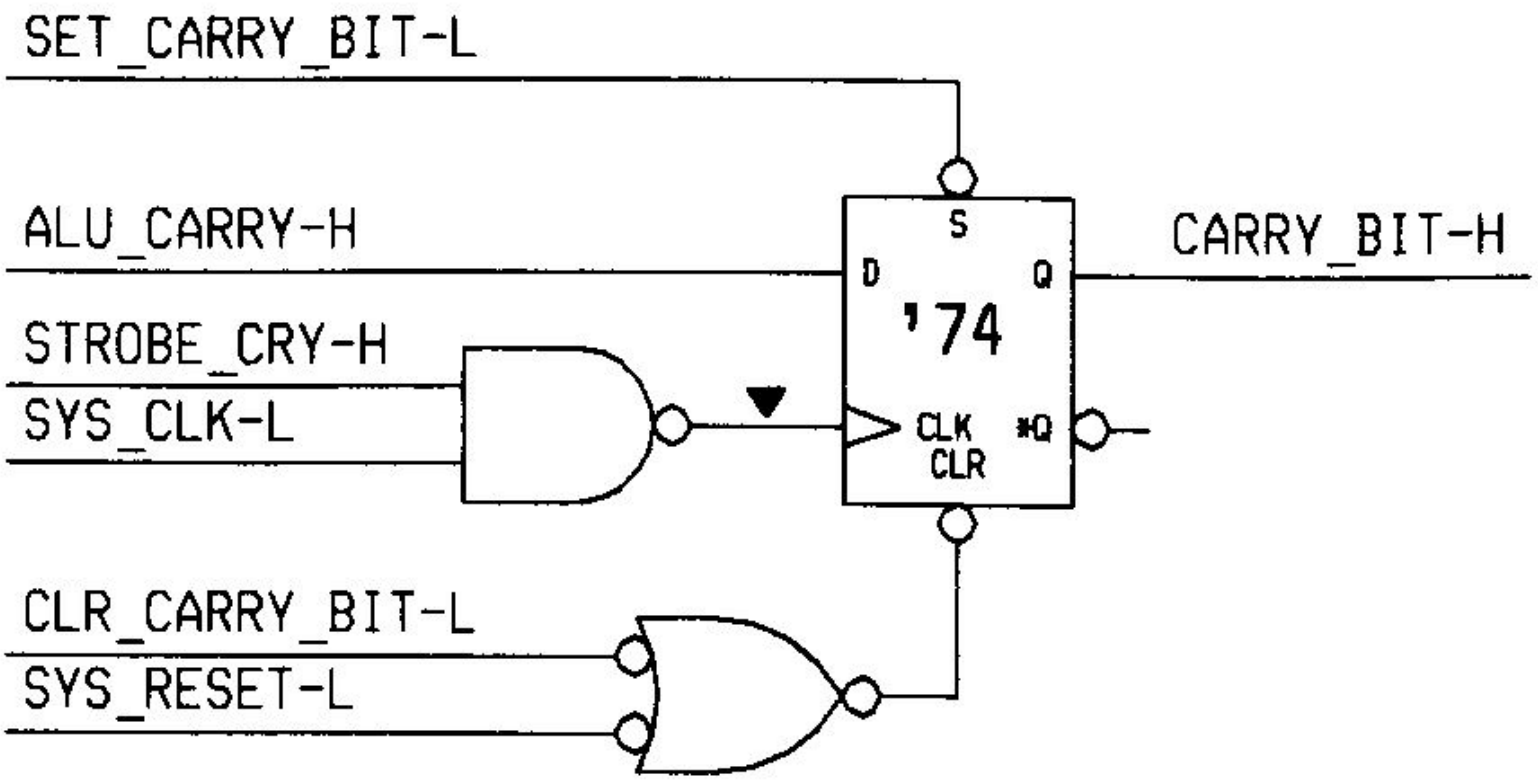
Státusz- (flag) jelzőbitek

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

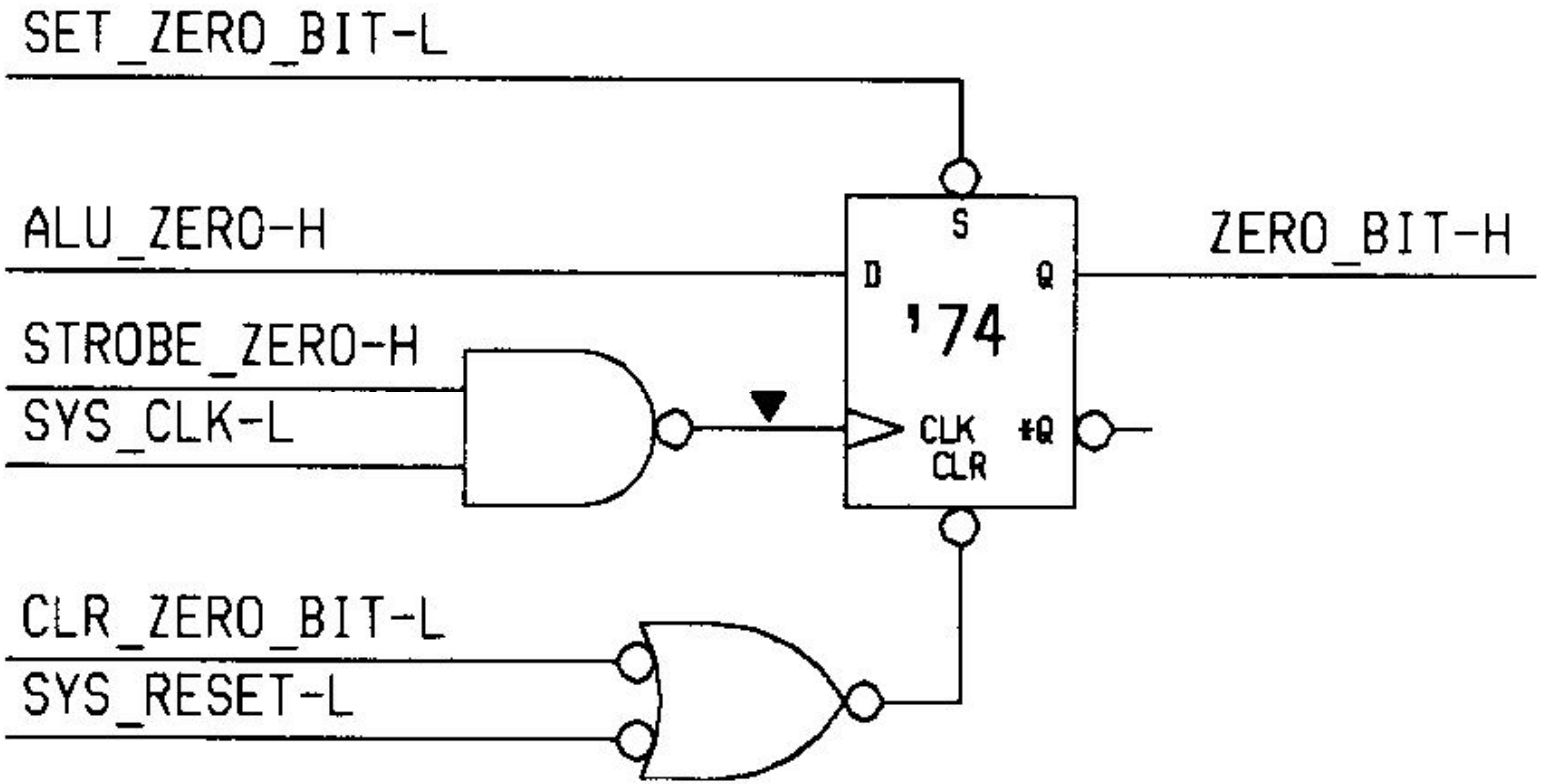
a.) Előjelbit (sign)



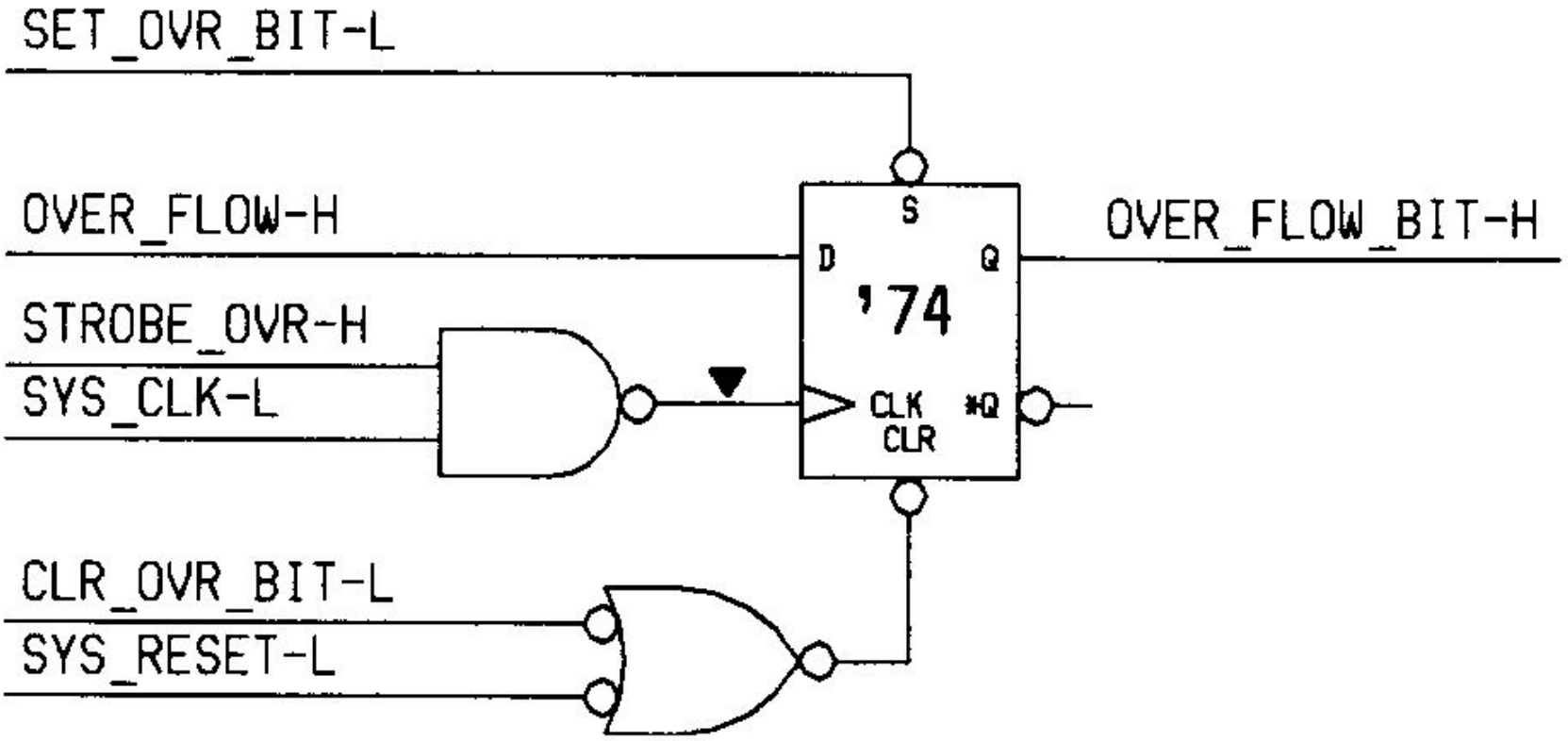
b.) Átvitel-kezelő bit (carry)



c.) Zéró bit

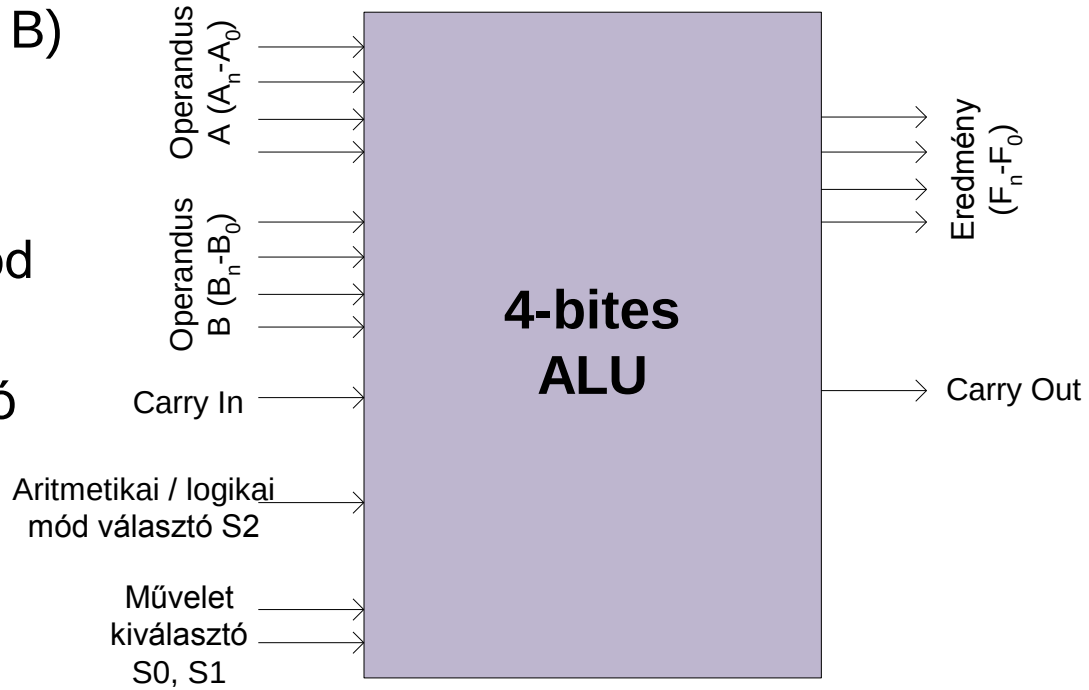

SET_ZERO_BIT-L
ALU_ZERO-H
STROBE_ZERO-H
SYS_CLK-L
CLR_ZERO_BIT-L
SYS_RESET-L
S
D
Q
'74
CLK
CLR
*Q
ZERO_BIT-H

d.) Túlcsordulást jelző bit (overflow)



Pl: 4-bites ALU felépítése és működése

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



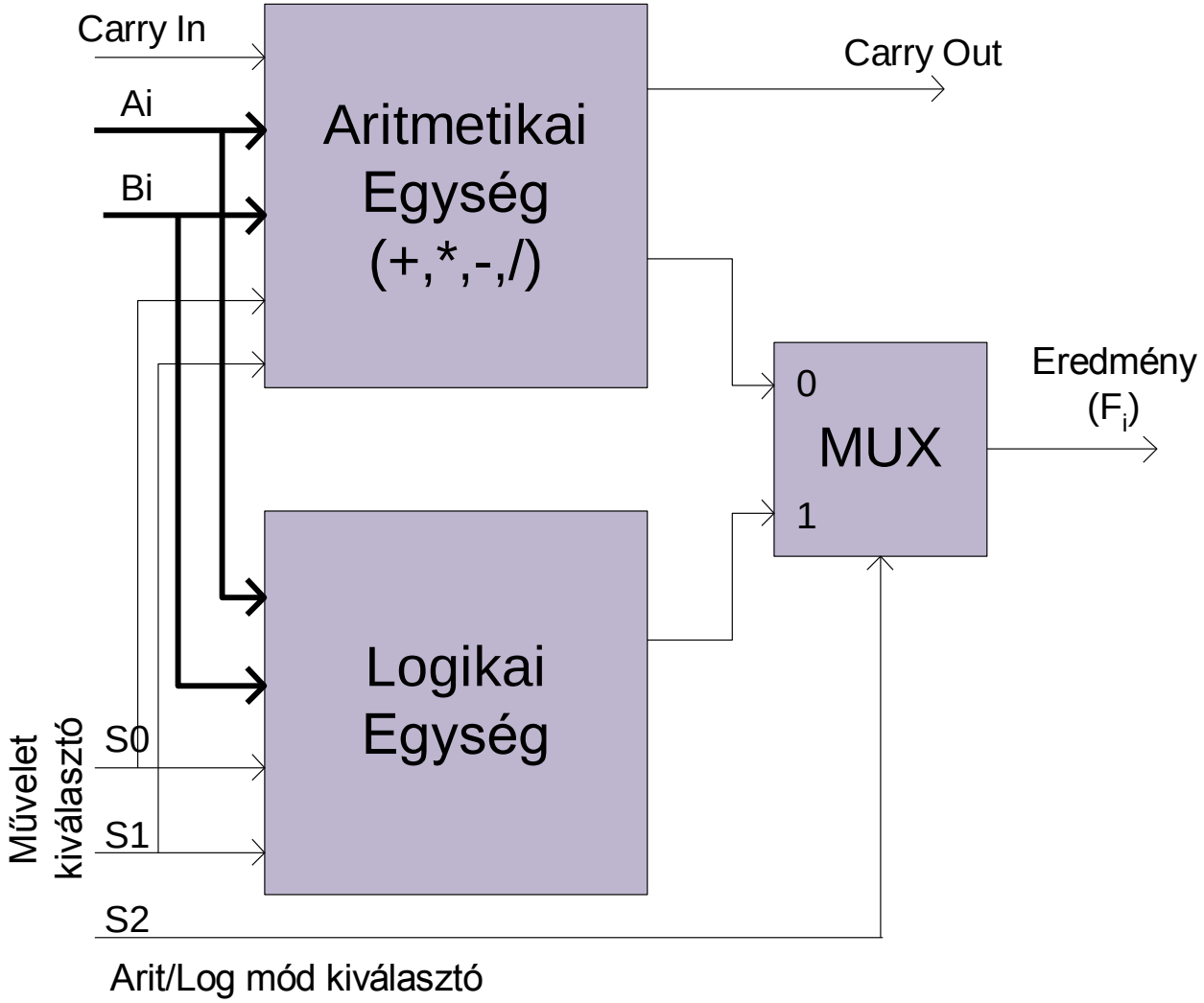
4-bites ALU szimbolikus rajza

ALU működését leíró függvénytáblázat:

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

* Könyv függeléke (appendix)

ALU felépítése:



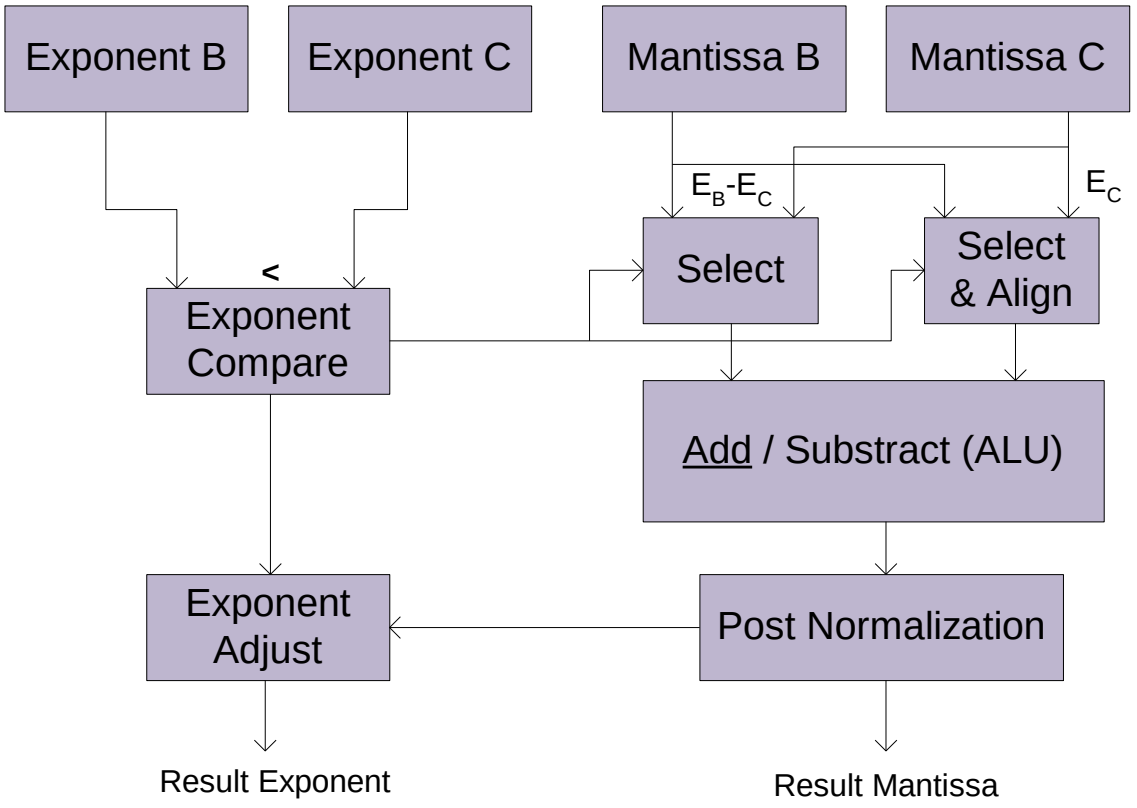
Lebegőpontos műveletvégző egységek

Lebegőpontos műveletvégző egységek

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

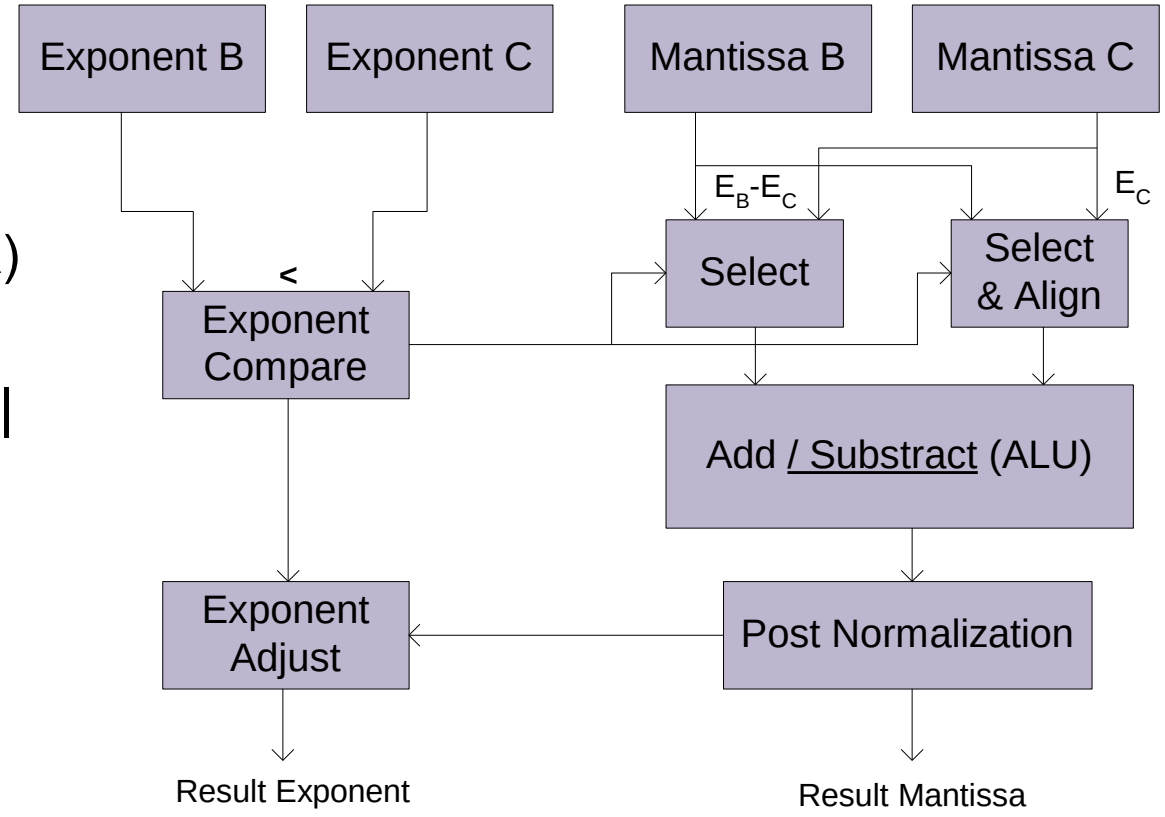
a.) Lebegőpontos összeadó

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



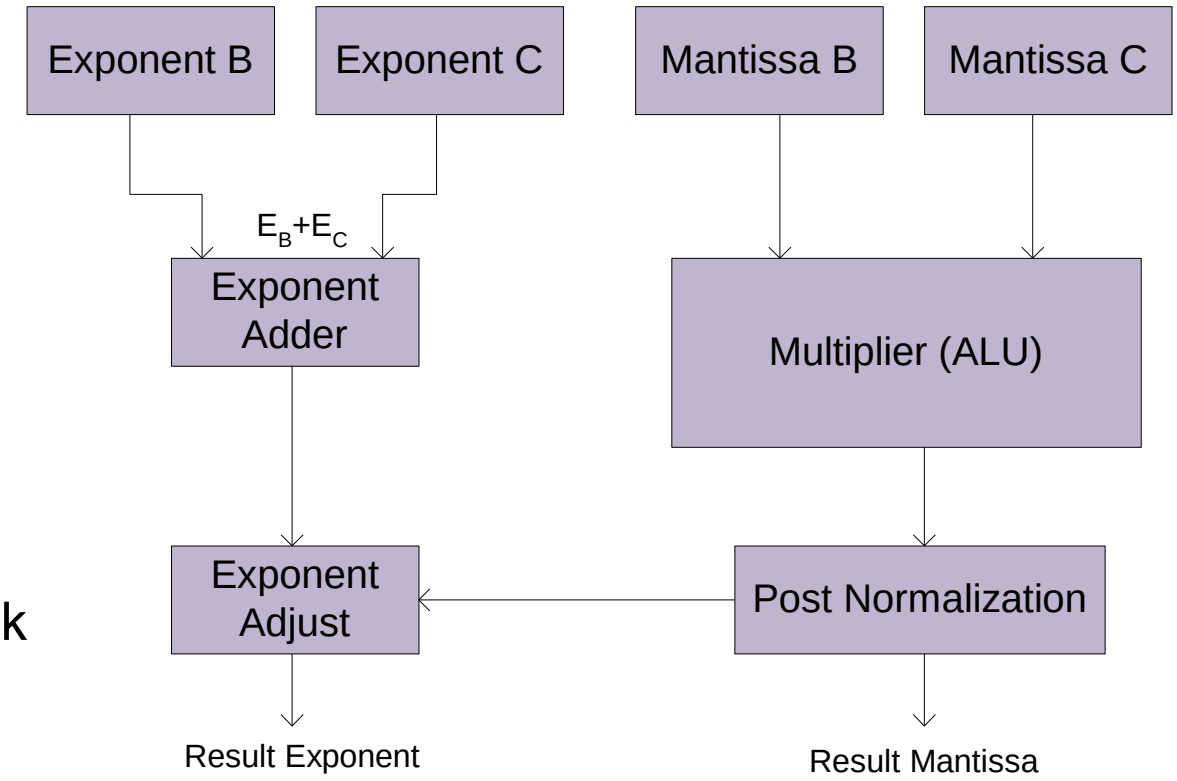
b.) Lebegőpontos kivonó

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



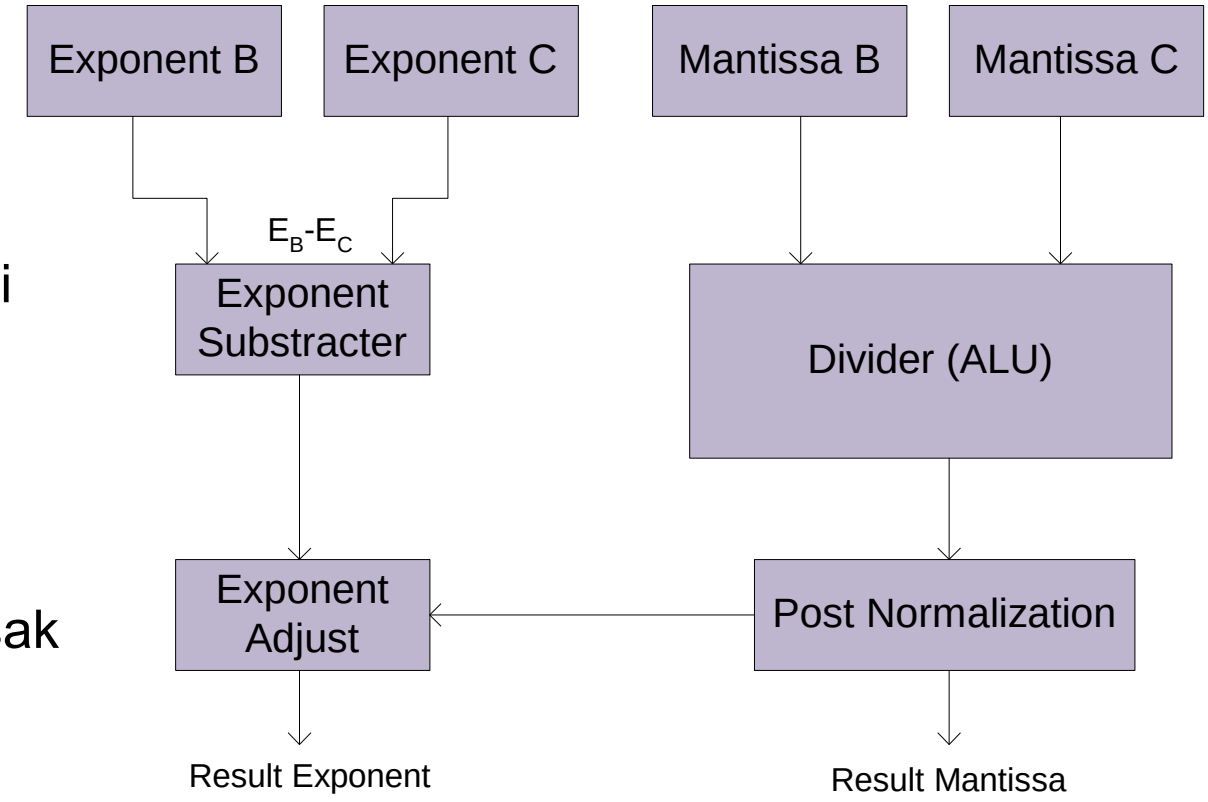
c.) Lebegőpontos szorzó

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



d.) Lebegőpontos osztó

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



Összeadó / Kivonó áramkörök

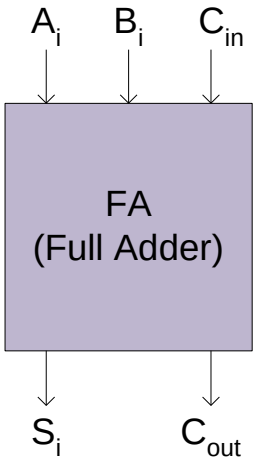
a.) Teljes összeadó – Full Adder

- FA: 1-bites Full Adder

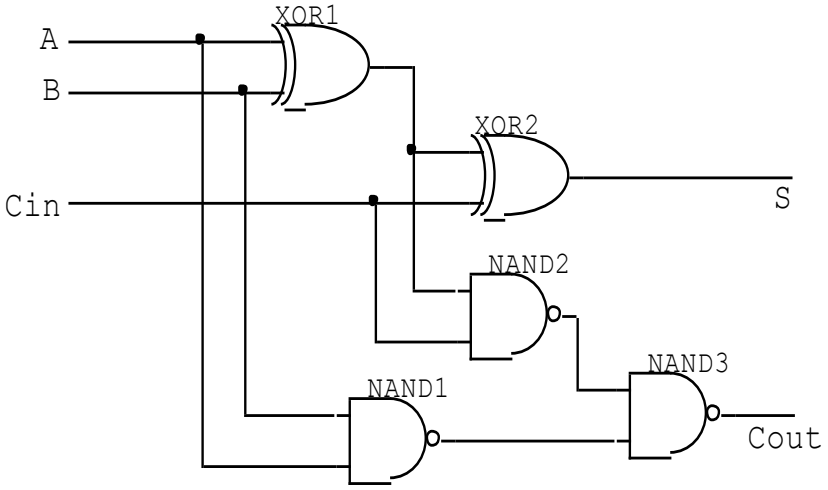
igazságtáblázat

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

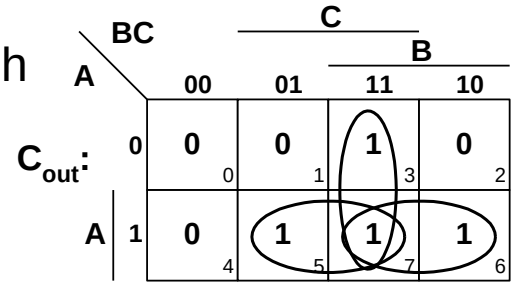
szimbólum

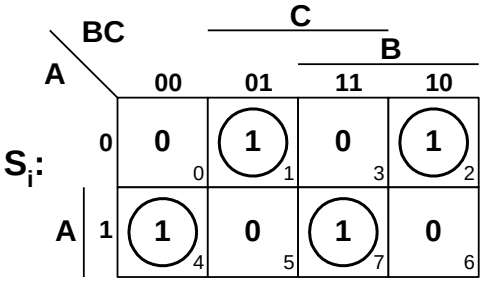


A FA egy CMOS kapcsolási rajza:



Karnaugh táblái:





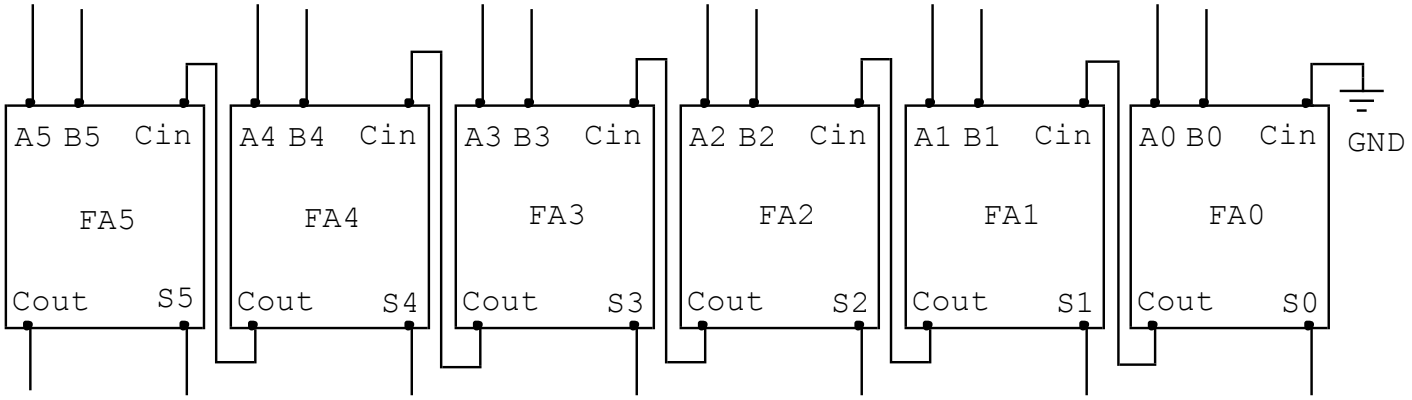
Kimeneti fgv-ei:

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

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

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

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



- Számítási időszükséglet (RCA):

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

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

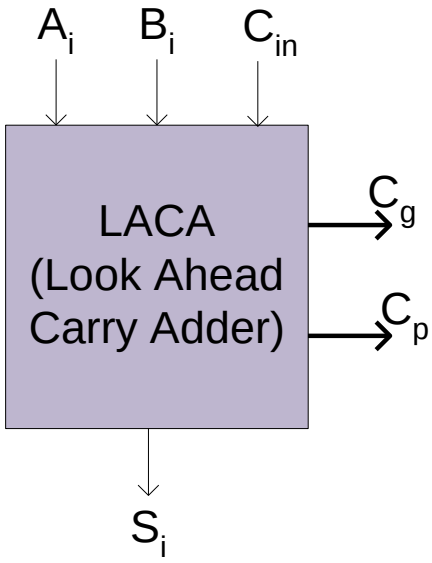
c.) LACA: Look Ahead Carry Adder:

- Képlet (FA) átírásából kapjuk:

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

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

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

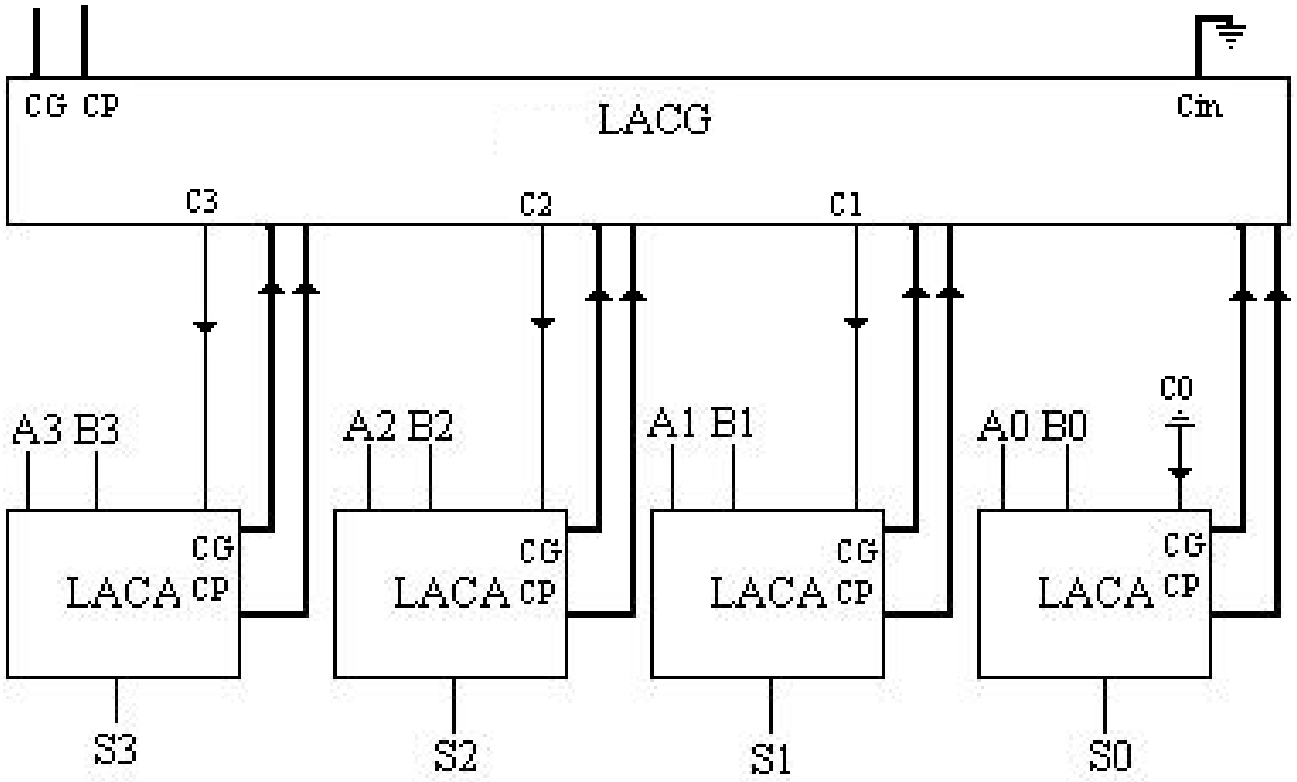


LACG: Look Ahead Carry Generator áll egy **b** bites ALU-ból, mindenegyes állapotban a Carry generálásáért felel a CP és CG (LACA) vonalakon érkező jeleknek megfelelően.
LACA számítási időszükséglete: $T_{LACA} = 2 + 4 \times (\lceil \log_b(N) \rceil - 1)$

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

Példa: 4-bites LACA

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



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

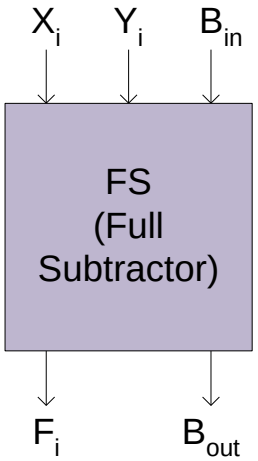
d.) Teljes kivonó - Full Subtractor (FS)

- FS: 1-bites Full Subtractor

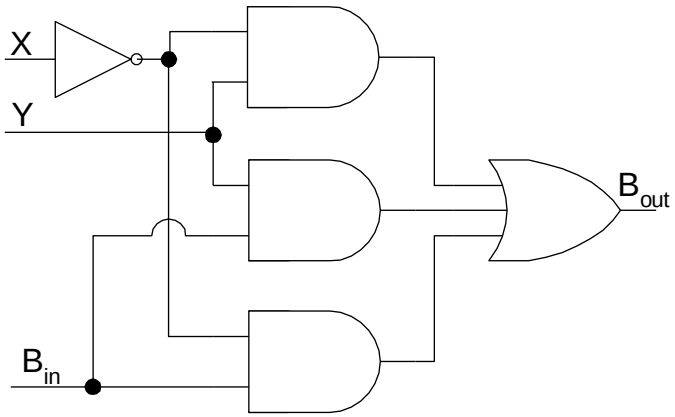
igazságtáblázat

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

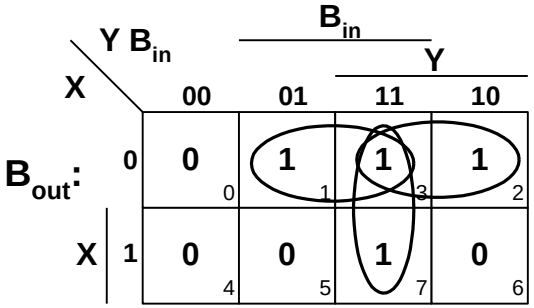
szimbólum

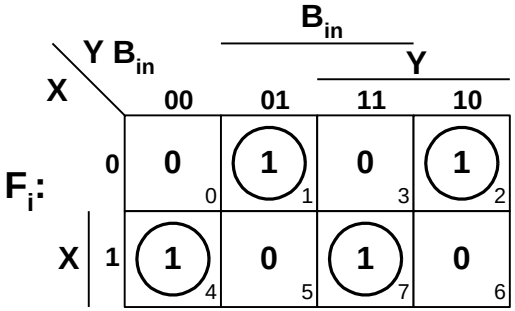


Logikai kapcsolási rajz (B out-ra)



Karnaugh táblái:





Kimeneti fgv-ei:

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

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

Szorzó áramkörök

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

I.) Iteratív szorzási módszerek

Iteratív szorzási módszerek alapjai

- Tényezők:

$$P = A \times B$$

  - P: szorzat, A:szorzandó, B:szorzó
  - Pl: Legyenek:
    - ‘A’ és ‘B’ 5-bites számok (0…2^5-1)=0…31
    - Maximálisan P=31*31=961 lehet (10 bites)
- Tehát: N bites számok szorzatát 2*N biten tudjuk eltárolni!

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

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

Iteratív szorzási műveletek:

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

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

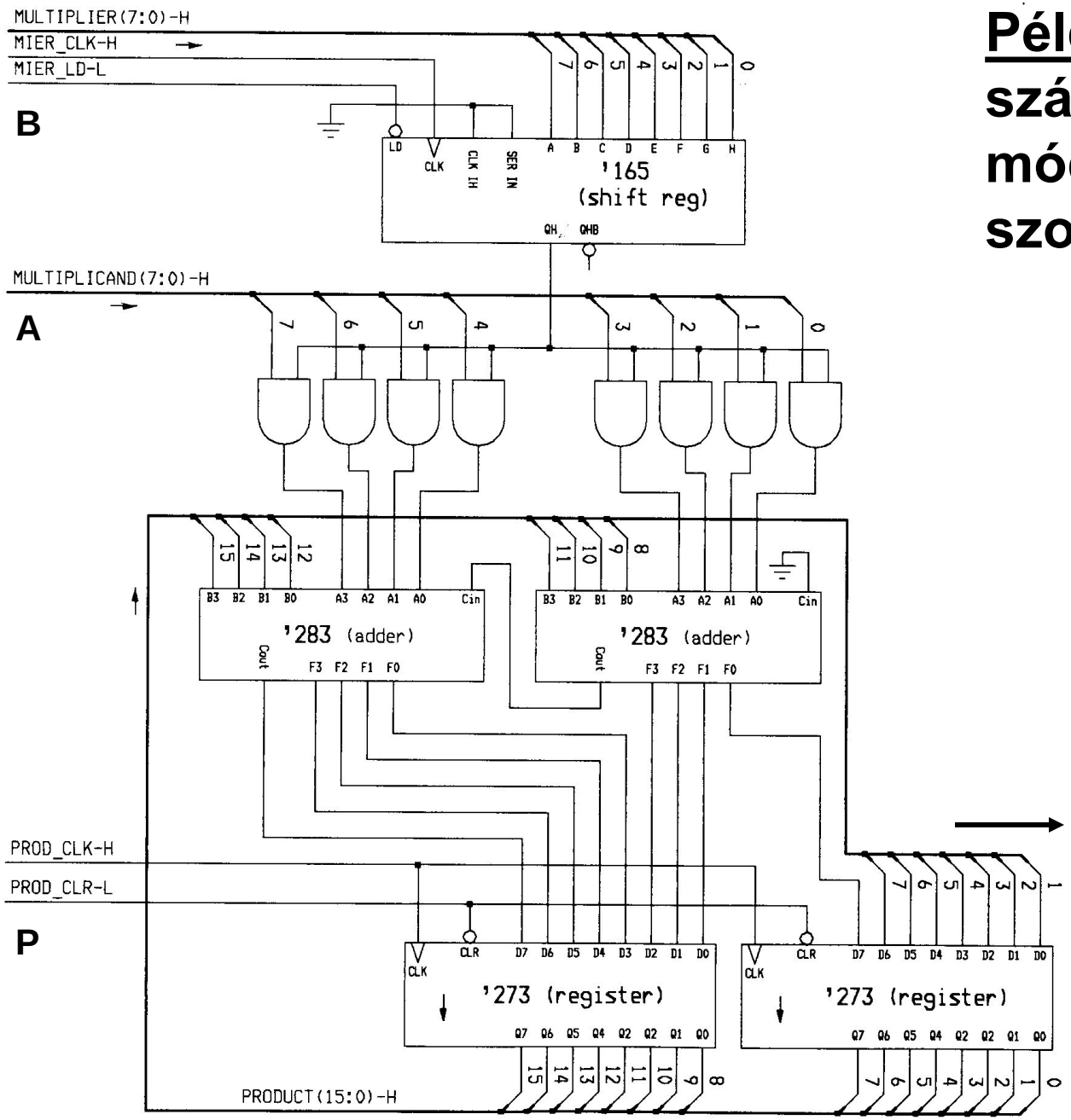
- Fordított sorrendű (MSB → LSB):

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

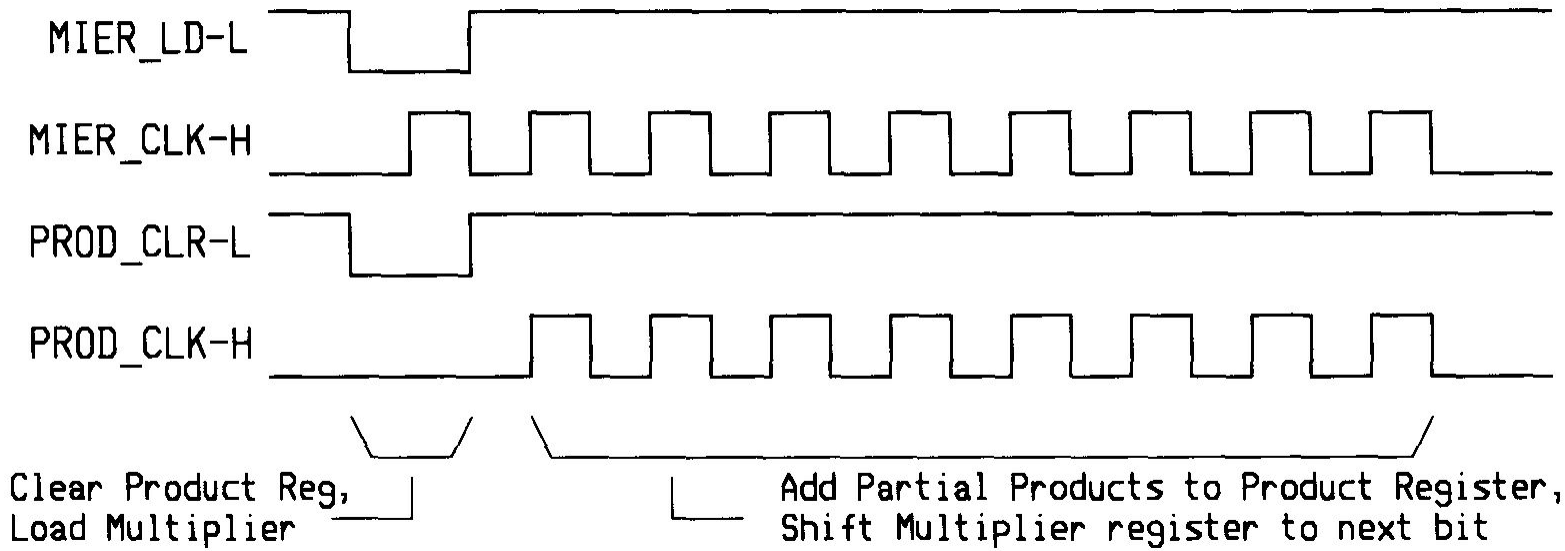
1.) Általános Shift&Add módszer

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

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



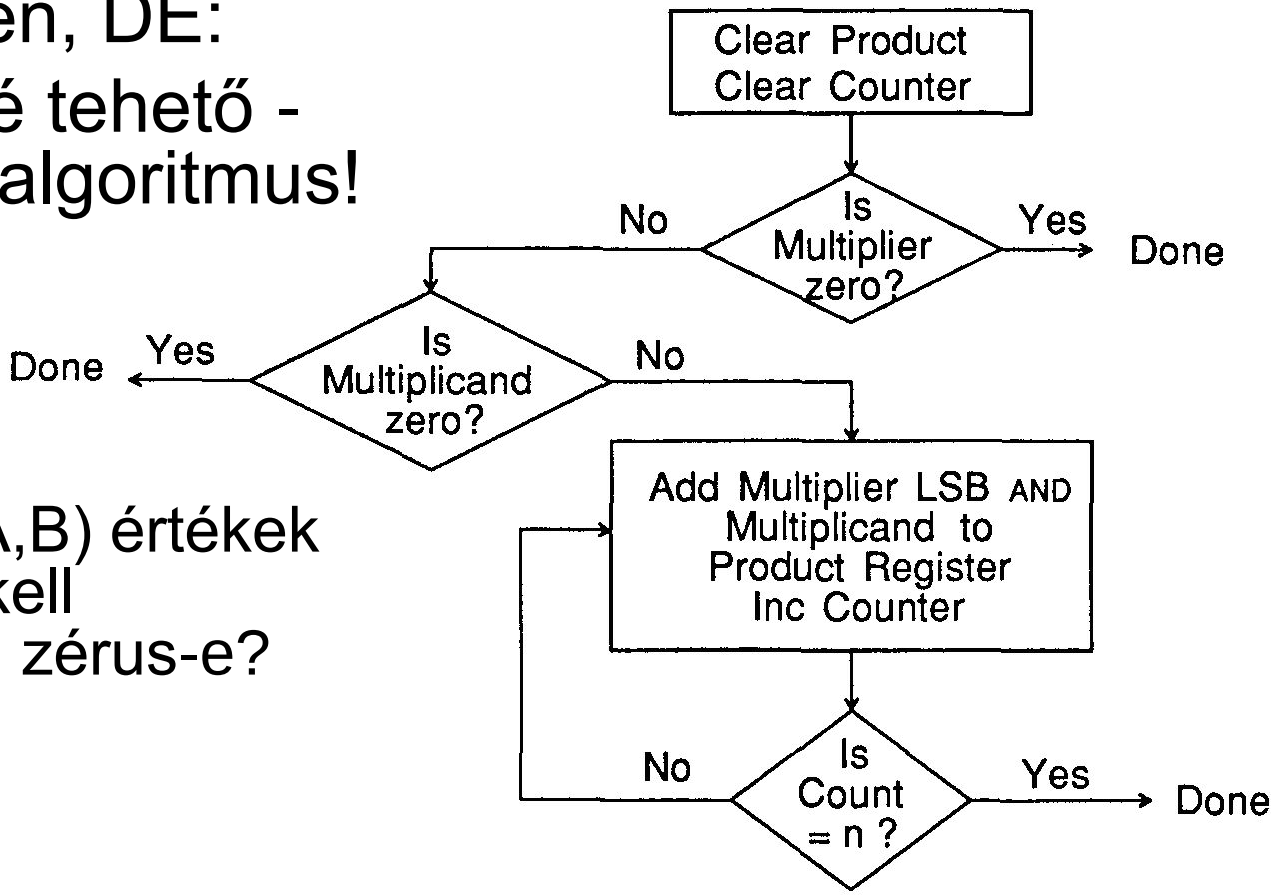
Időzítési jelek:



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

Folyamatábra (Shift-add)

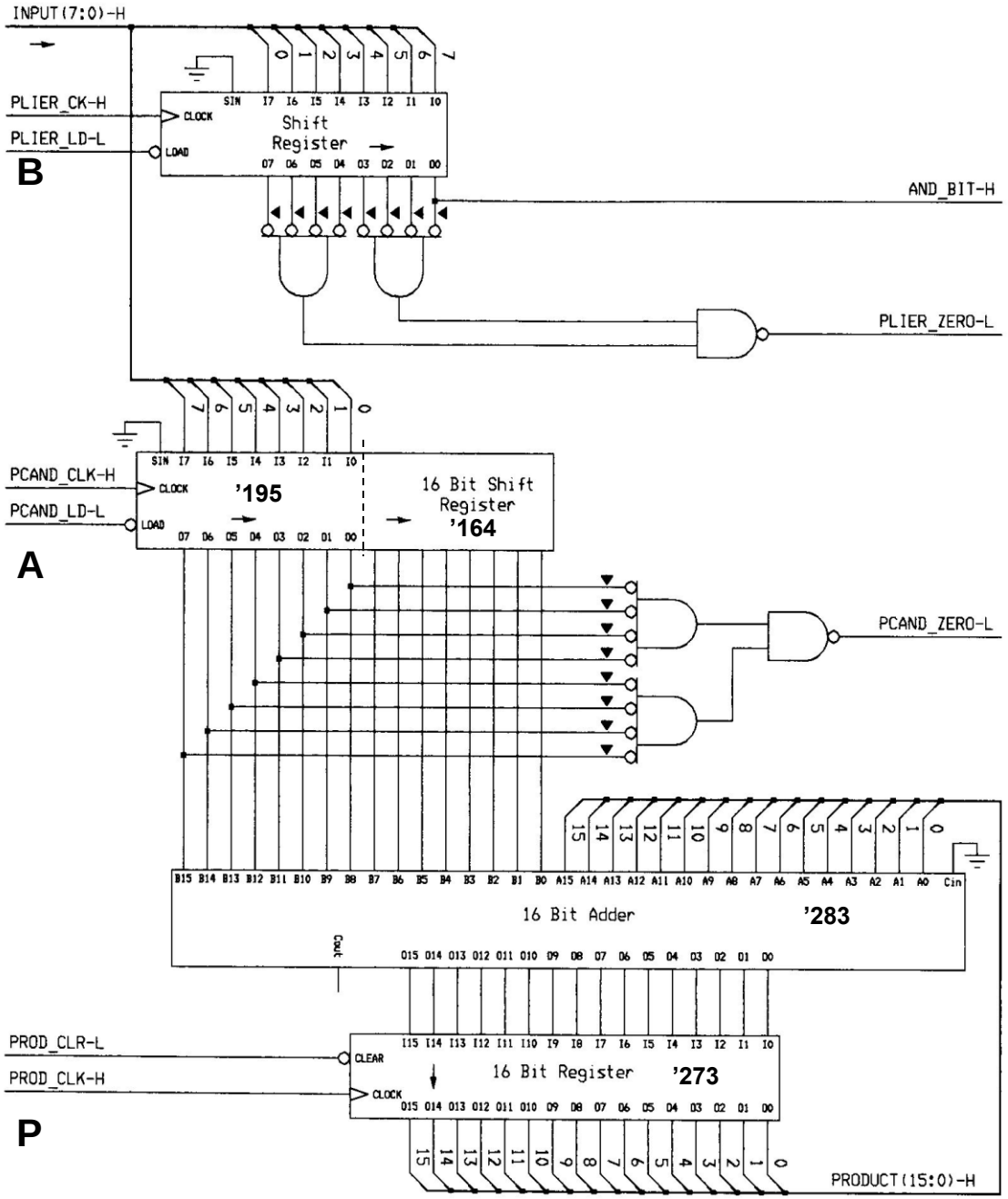
- Alapvetően adatfüggetlen, DE:
- Adatfüggővé tehető - gyorsítható algoritmus!
  - Bemenő (A,B) értékek figyelését kell megoldani: zérus-e?



2.) Fordított sorrendű módszer

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

Példa: két 8-bites szám Fordított sorrendű szorzására



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

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

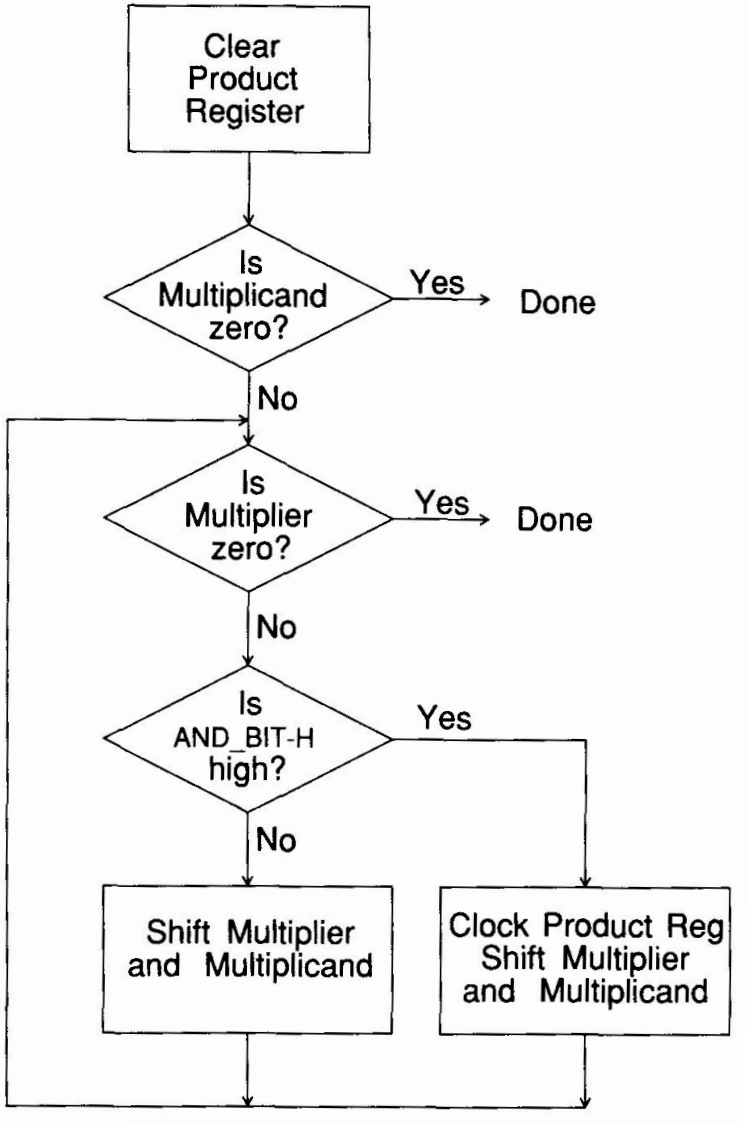
Folyamatábra (fordított sorrendű)

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

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

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

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



c.) Előjeles szorzás Booth-algoritmussal:

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

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

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

    (azonos numerikus értékek összerendelése)

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

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

Példa: előjeles Booth algoritmus

- Az előző oldalon lévő *átzárójelezéssel* a megfelelő értékeket két bitpár kivonásával kapjuk (a **zárójeles** kifejezés értéke ha **+**:akkor kivonás,/ ha **0**:akkor áteresztés / ha – összeadás történik). A súlytényezők 2 hatványai, és a szorzást a súlytényezők shiftelésével oldják meg (’165 Shift-regiszterrel). Egy összeadást mindig egy kivonás követ alternáló jelleggel.
- Példa: 543210 (bitpozíciók)

  011001= 25 „A”

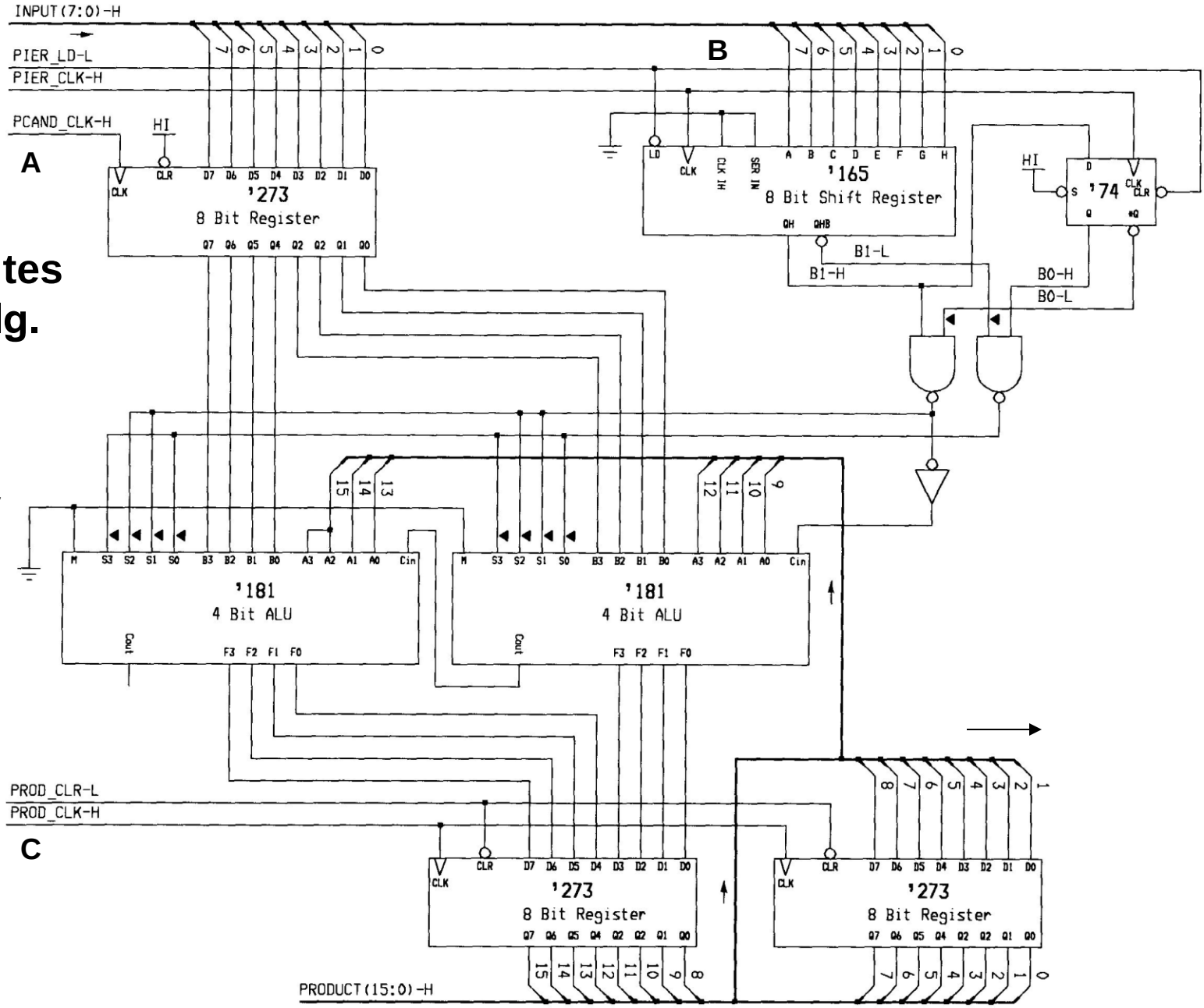
  101101=–19 „B” Végrehajtjuk **A*B**-t!

Az újrakódolást bitpárokon végezzük el:

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

Példa: két 8-bites szám Booth alg. szorzására

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



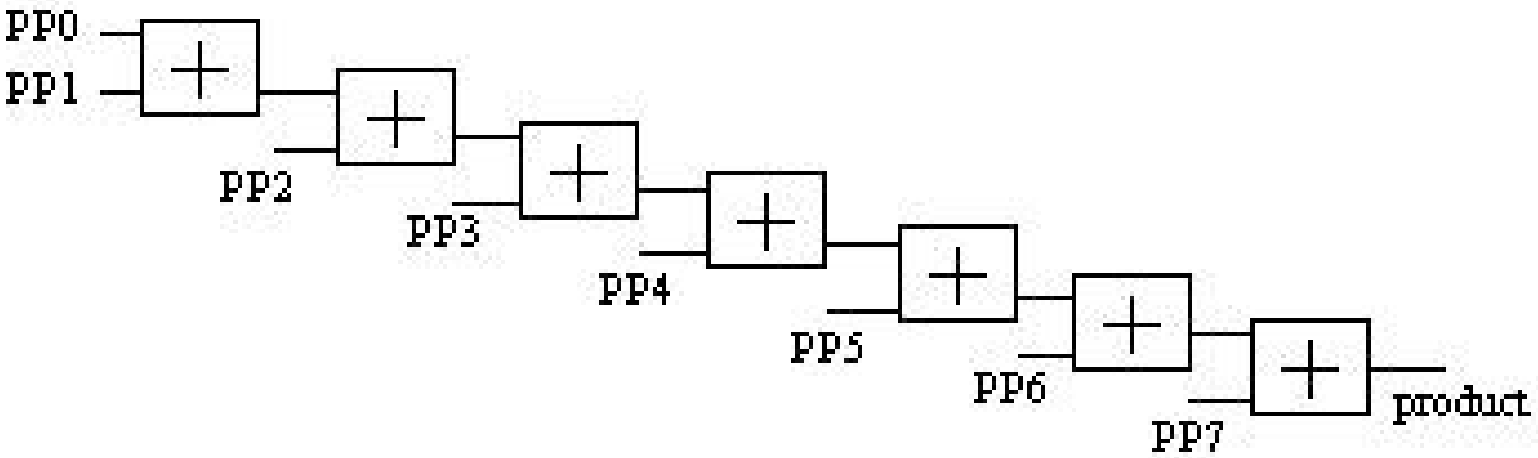
II. Közvetlen szorzási módszerek

Közvetlen szorzási módszerek

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

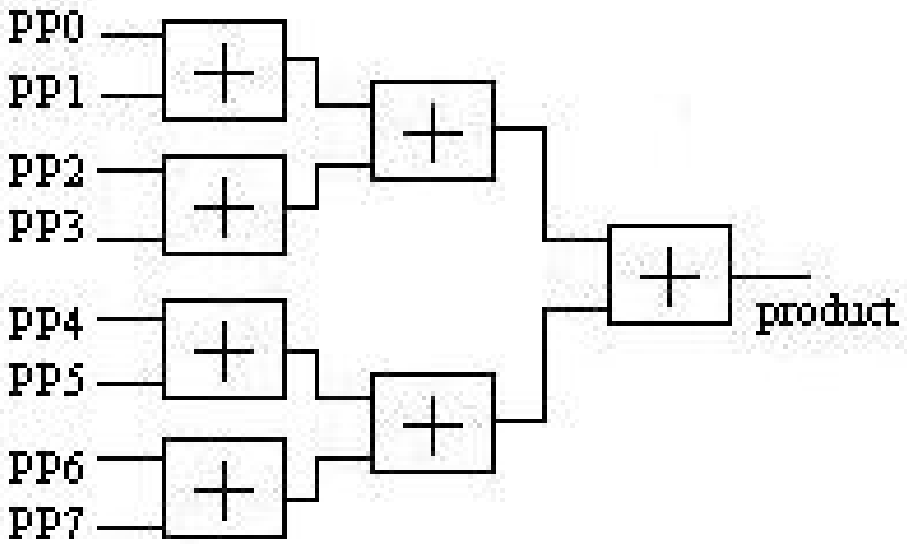
a.) Lineáris modell

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



b.) Fa modell

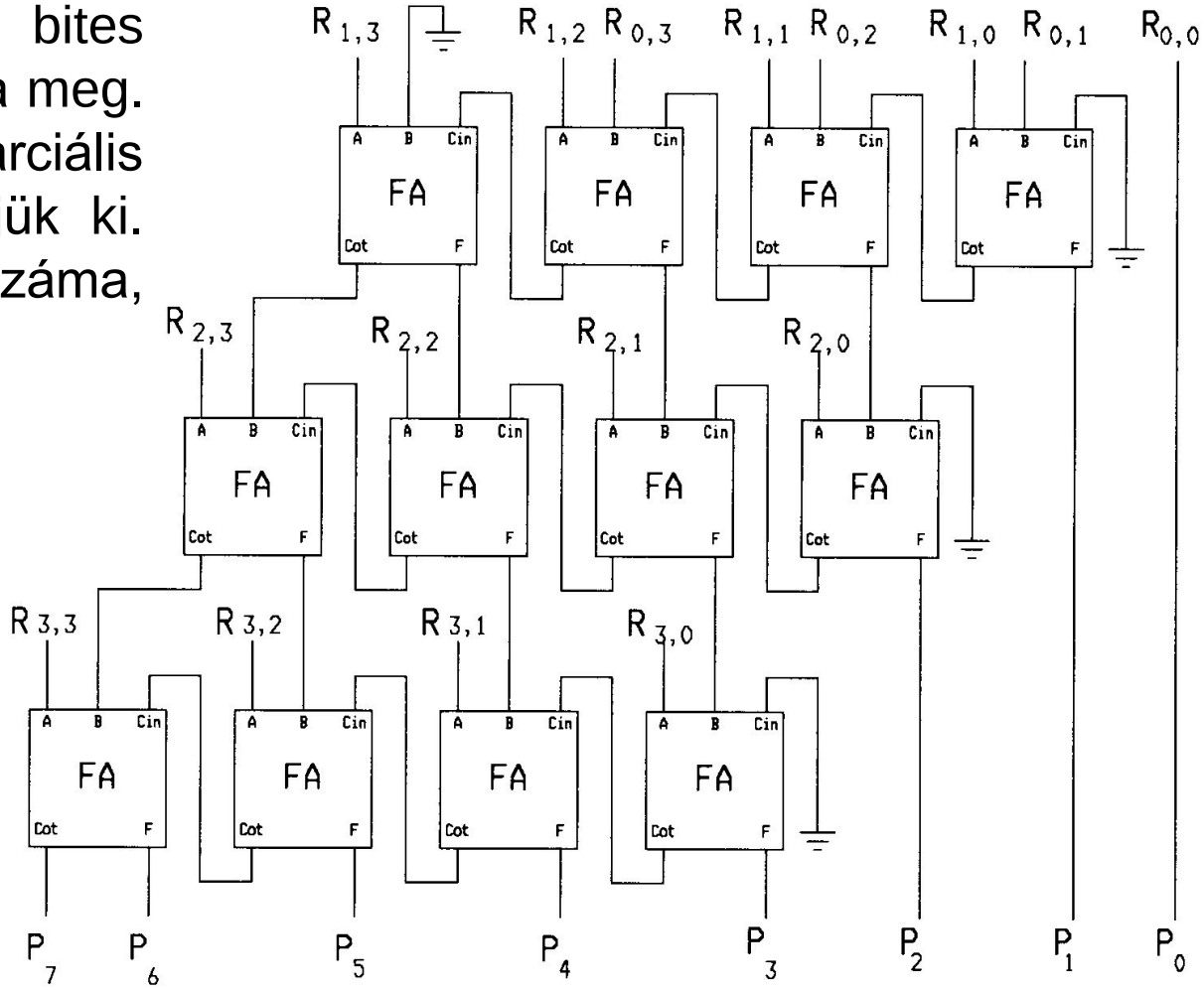
- A parciális szorzatok, képzésük után szintén azonnal összeadhatók, gyorsabban megkapjuk az eredményt, mint a lineáris modellnél, mivel ebben az esetben (N=8 bit esetén) csak 3 szintű a hierarchia, így kevesebb a késleltetés. N bites számok esetén (N-1) db összeadóra van szükségünk.
- Időszükséglet: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



c.) Full Adder-es megvalósítás

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

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



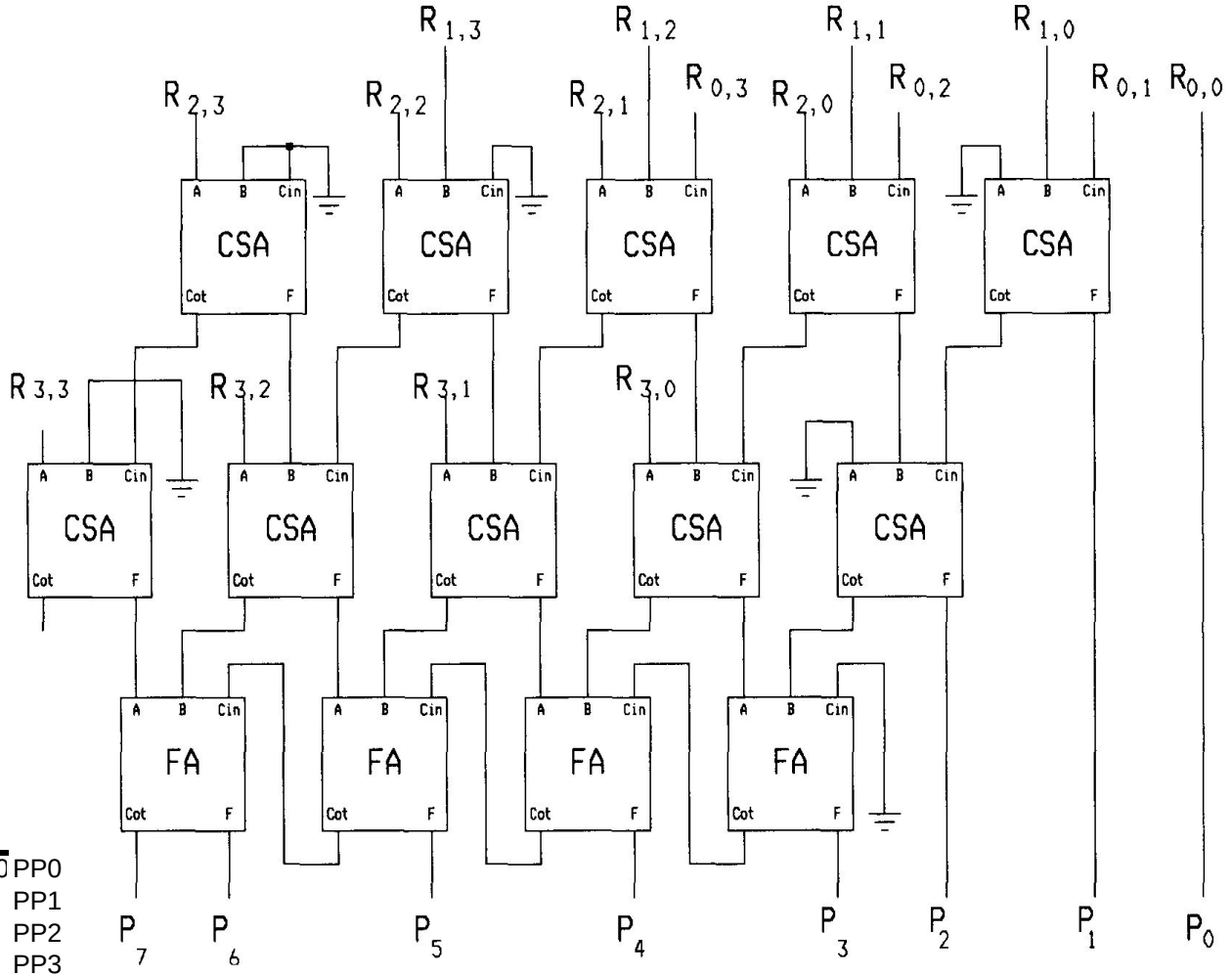
d.) CSA: Carry Save Adder

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

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

P=AxB

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

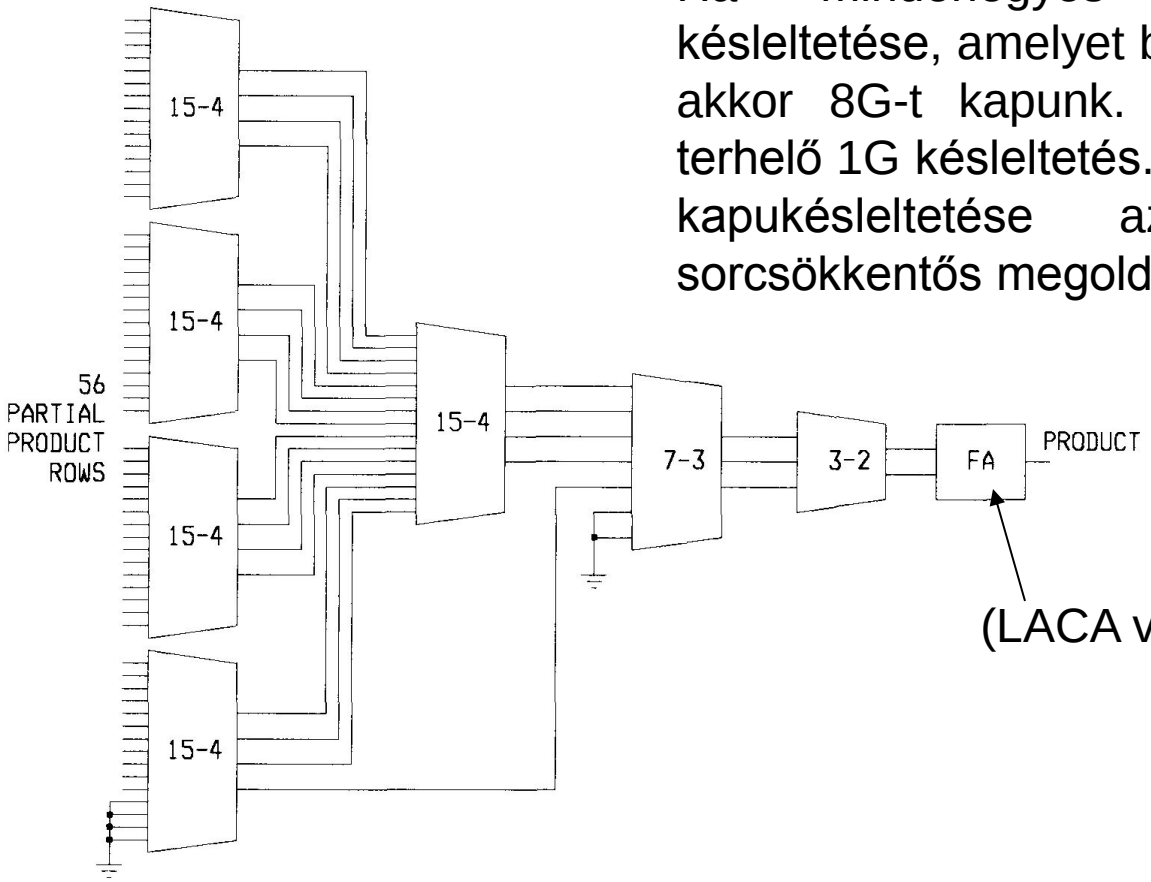


e.) Sorcsökkentős megvalósítás (row reduction)

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

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

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



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

Osztó áramkörök

Osztó áramkörök:

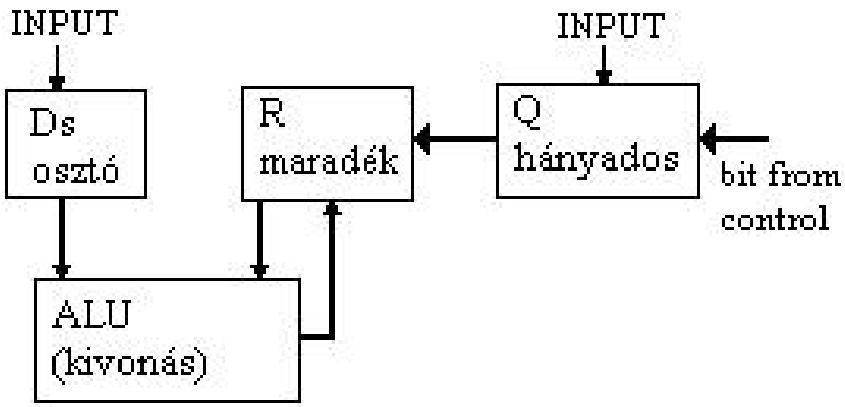
- I.) Hagyományos közvetlen osztási algoritmus
- II.) Iteratív osztási algoritmusok

I.) Hagyományos közvetlen osztási algoritmus:

Ez az osztási folyamat igen lassú eljárás. Lépései:

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

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



Példa: Hagyományos osztási algoritmus

Dd=Q*Ds + R

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

Decimális számok esetén:

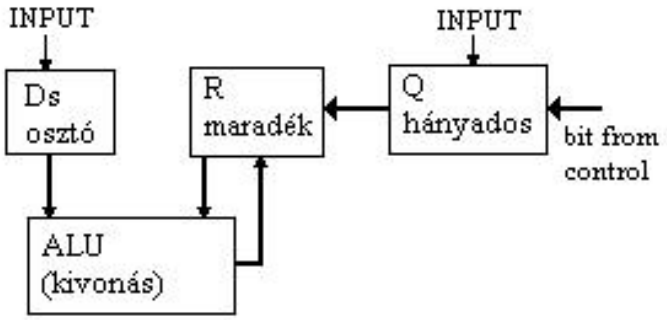
5|8 : 5=1|1
0 8
3

Bináris számok esetén hasonlóan

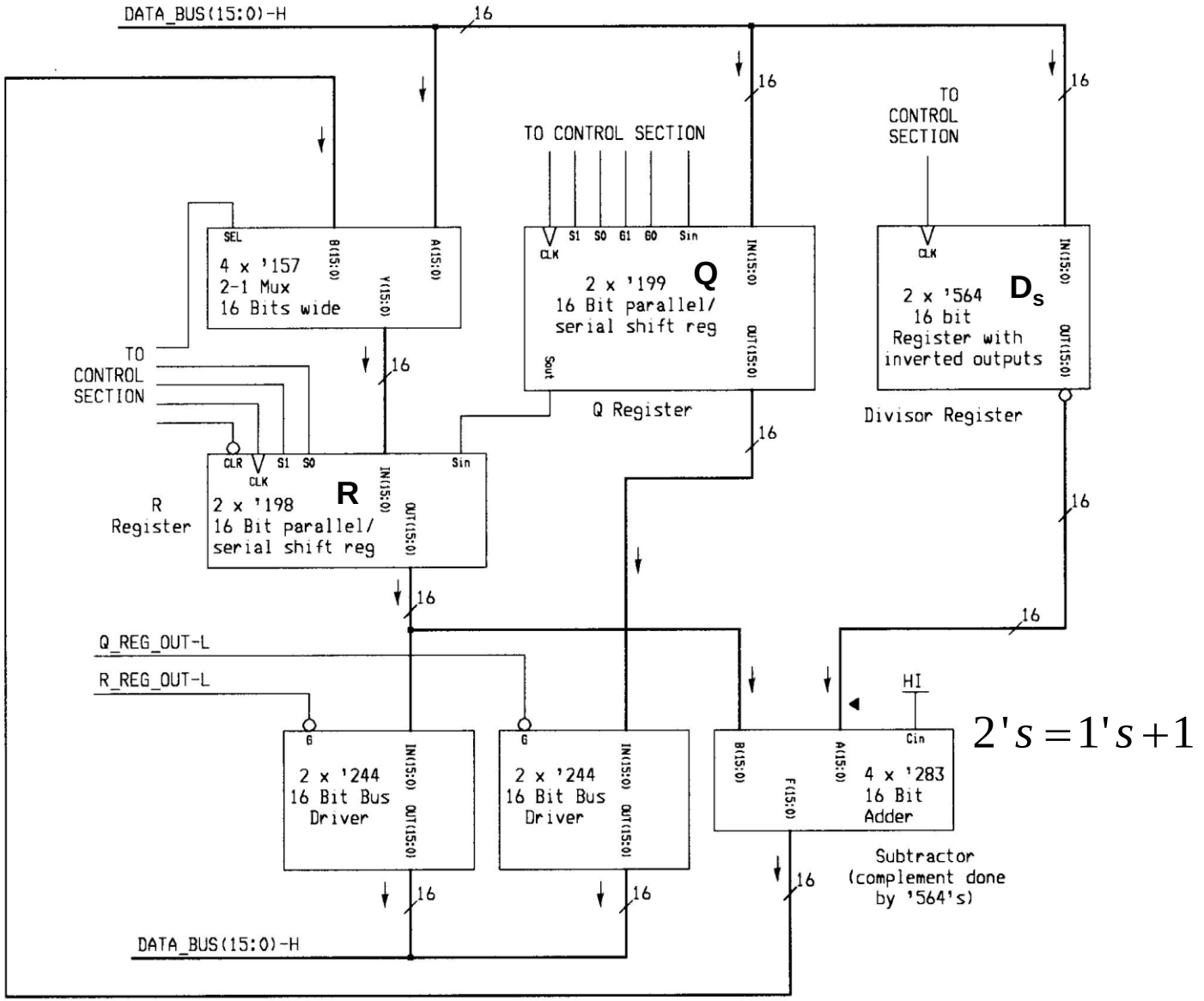
1011 Q hányados (Ds hányszor van meg Dd-ben)

111010 Dd osztandó
101/ Ds osztó

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

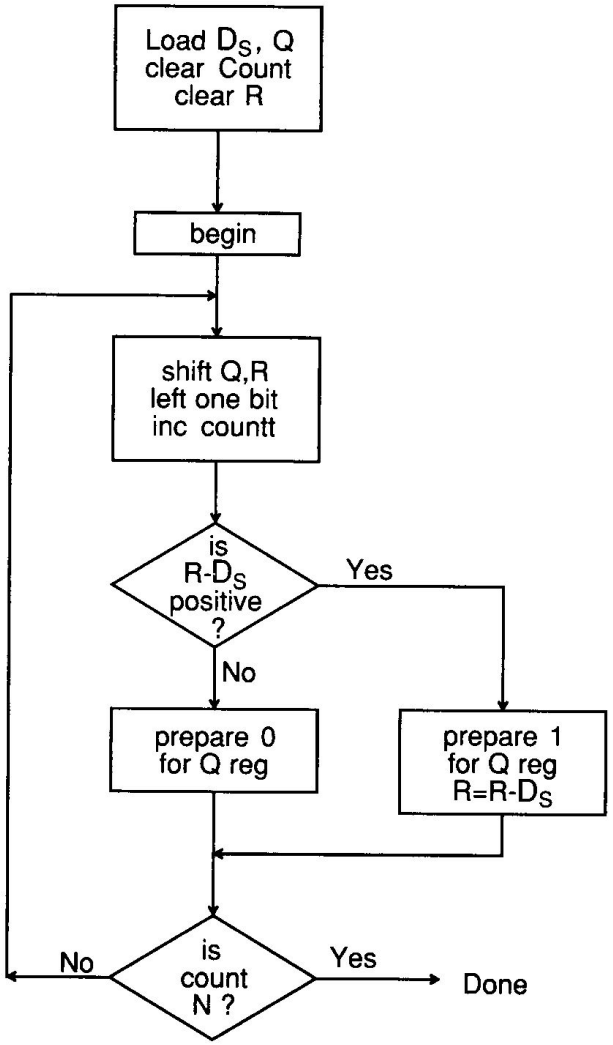


Hagyományos osztó áramkör



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

Folyamatábra: osztási algoritmus



II.) Iteratív osztási algoritmus

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

a.) Gyors osztás Newton- Raphson módszerrel

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

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

Van egy megfelelő f függvényünk és egy x0 kezdeti értékünk. Iterációs lépésekkel megkapjuk az osztás eredményét az f(x)=0 egyenlet megoldásaként. Az f-et úgy kell (jól) megválasztanunk, hogy a reciprok gyökkel rendelkezzen. Legyen

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

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

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

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

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

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

b.) Közvetlen gyors osztó

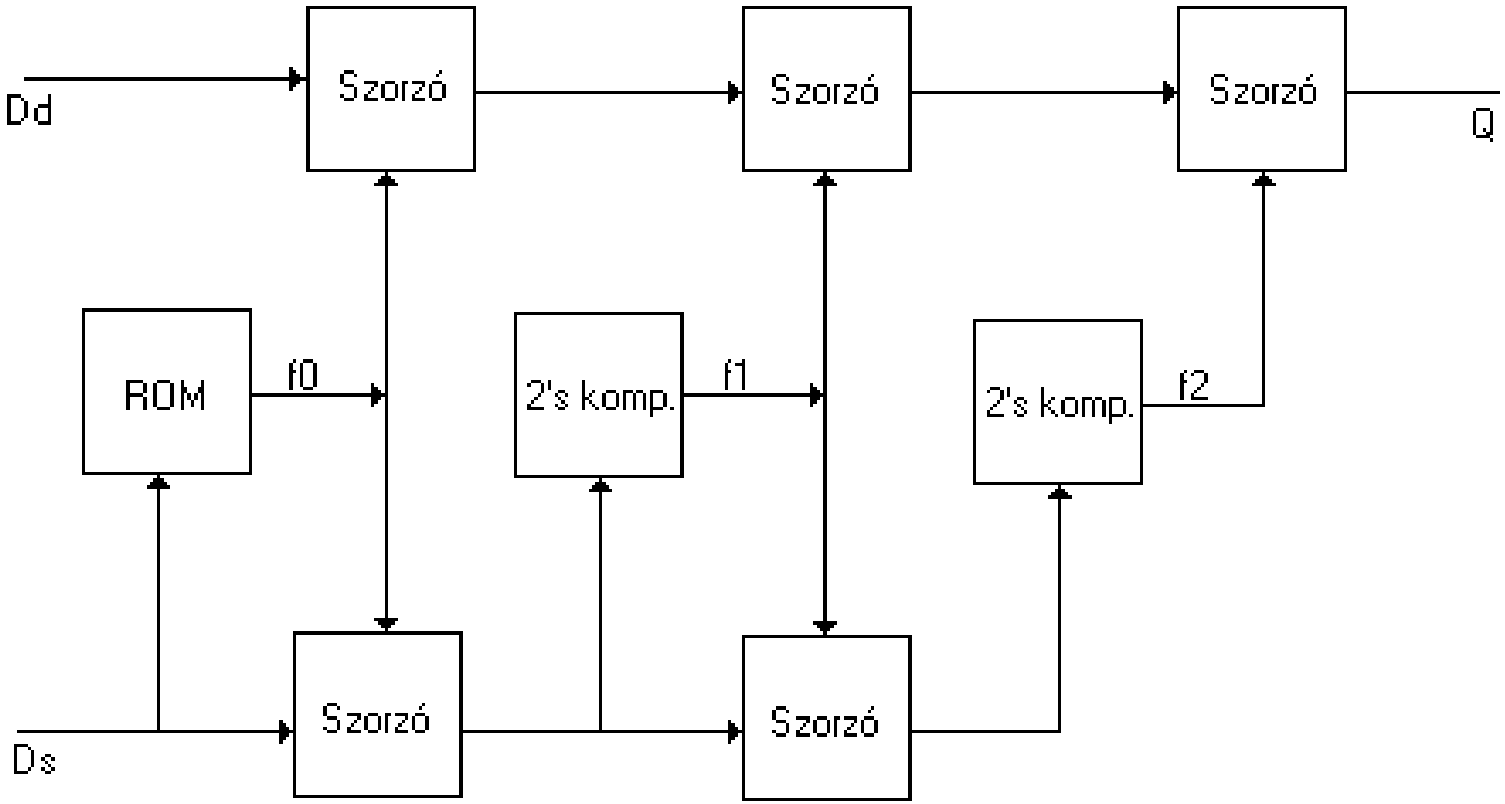
- Az iteratív osztási művelet másik módszere a következő: $Q=D_D/D_S$ kiszámolható a következő egyenlettel, ha a successive (egymást követő) $\mathbf{f_k}$ –k úgy vannak megválasztva, hogy a nevező az 1-hez konvergáljon.

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

Közvetlen gyors osztó működése

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

Közvetlen gyors osztó áramköri felépítése



Közvetlen gyors osztó

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

Példa 1.

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

- 0. D_{DO}=0,4000000 D_{SO}=0,7000000 fo=1,3000000
- 1. D_{D1}=0,5200000 D_{S1}=0,9099999 f1=1,0900000
- 2. D_{D2}=0,5668000 D_{S2}=0,9918999 f2=1,0081000
- 3. D_{D3}=0,5713911 D_{S3}=0,9999344 f3=1,0000656
- 4. D_{D4}=0,5714286 D_{S4}=0,9999999 f4=1,0000000
- 5. D_{D5}=0,5714286 D_{S5}=1,0000000 f5=1,0000000
- 6. D_{D6}=0,5714286 D_{S6}=1,0000000

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

Példa 2.

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

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

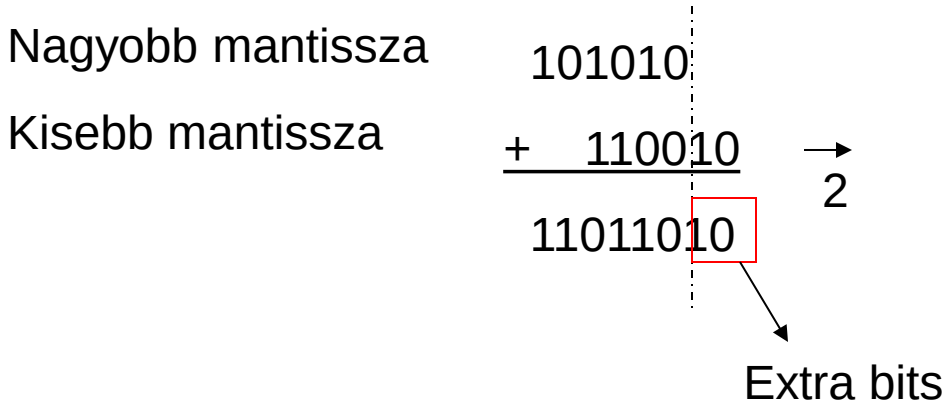
Látható, hogy itt nem kapjuk meg a kivánt értéket ($D_{D6}=0{,}6666464$) 6 iterációs lépés alatt. Ezért hogy elérjük a kivánt pontosságot véges számú lépés alatt, ROM-ot kell használni (ahol fo kezdeti értékét tároljuk).

Extra bitek kezelése

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

Extra bit: probléma

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



a.) Truncation (levágás)

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

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

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

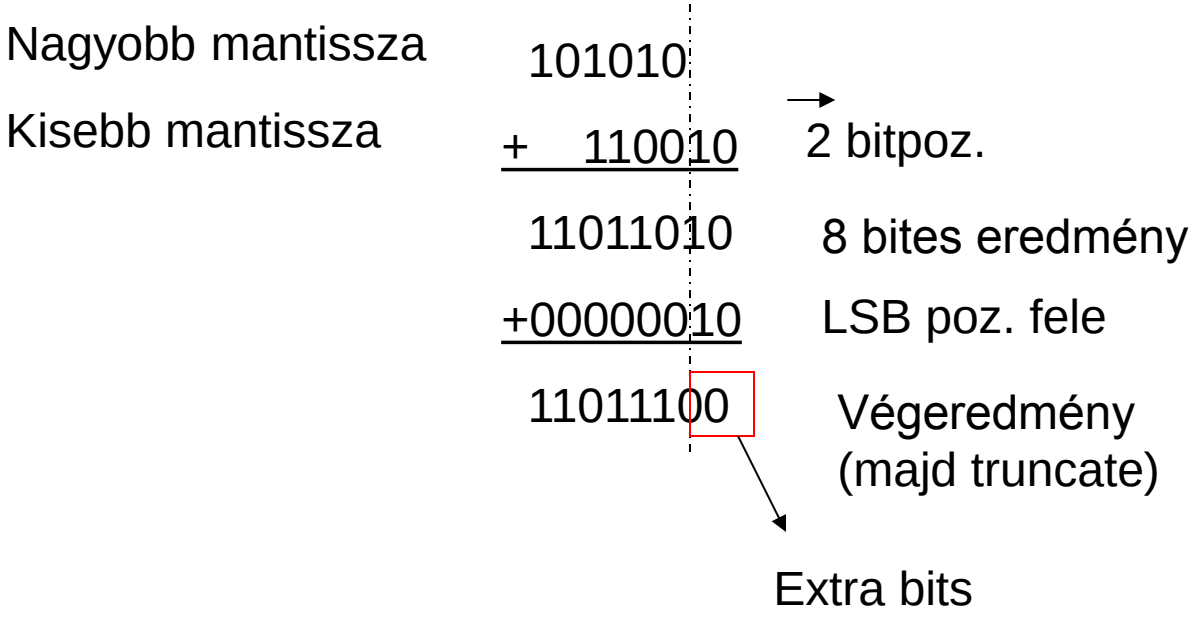
Truncation: példa

- ERR → bias: mindig pozitív

cases	MR	MF	ERR_{TRUNC}
a	xx0.00	xx0.	0.00
b	xx0.01	xx0.	+0.01
c	xx0.10	xx0.	+0.10
d	xx0.11	xx0.	+0.11
e	xx1.00	xx1.	0.00
f	xx1.01	xx1.	+0.01
g	xx1.10	xx1.	+0.10
h	xx1.11	xx1.	+0.11

b.) Rounding (kerekítés)

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



PI: Rounding (kerekítés)

- ERR → bias: hiba itt is ugyan megmarad, de már pozitív és negatív is lehet (bias-a kisebb, mint a levágás esetén)

case	MR	MR+1/2 LSB	MF	ERR_{ROUND}
a	xx0.00	xx0.10	xx0.	0.00
b	xx0.01	xx0.11	xx0.	+0.01
c	xx0.10	xx1.00	xx1.	-0.10
d	xx0.11	xx0.11	xx1.	-0.01
e	xx1.00	xx1.10	xx1.	0.00
f	xx1.01	xx1.11	xx1.	+0.01
g	xx1.10	xy0.00	xy0.	-0.10
h	xx1.11	xy0.01	xy0.	-0.01

Változás a truncate-hez képest!

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

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

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

Pl: Round-to-Zero (R*)

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

Változás a rounding-hoz képest!

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

d.) Jamming (~fix értéken rögzítés)

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

e.) ROM rounding

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

Extra-bit kezelő módszerek összehasonlítása: