Pannon Egyetem Villamosmérnöki és Információs Tanszék



Számítógép Architektúrák II. (MIVIB344ZV)

4. előadás: Aritmetikai műveletvégző egységek – szorzás, osztás. Kerekítési eljárások.

Előadó: Dr. Vörösházi Zsolt

voroshazi.zsolt@mik.uni-pannon.hu



Jegyzetek, segédanyagok:

- Könyvfejezetek:
 - □ http://www.virt.uni-pannon.hu → Oktatás → Tantárgyak → Számítógép Architektúrák II.
 - □ (chapter03.pdf)
- Fóliák, óravázlatok .ppt (.pdf)
- Feltöltésük folyamatosan

Szorzó áramkörök

Ismétlés: Tárolók - Regiszterek

- A szorzó áramkörökben (is) egy fontos építőelem a regiszter. Olyan széles, hogy benne a buszokról, memóriákból, ALU-ból érkező információ eltárolható legyen.
- Adott vezérlőjelek hatására:
 - □ a bemenetén lévő adatokat betölti, és ideiglenesen eltárolja,
 - □ a kimenetére teszi a tárolt adatokat, vagy
 - □ lépteti (shift-eli) a benne lévő adatokat.
- Működési mód szerint:
 - a) Hagyományos reg.: párhuzamos betöltésű/kiolvasású
 - b) Léptető (Shift) regiszter: soros betöltésű (kiolvasású)

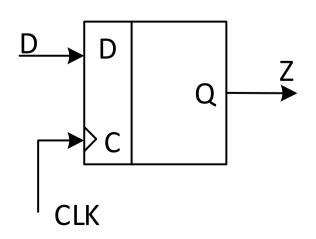
Ismétlés: D flip-flop

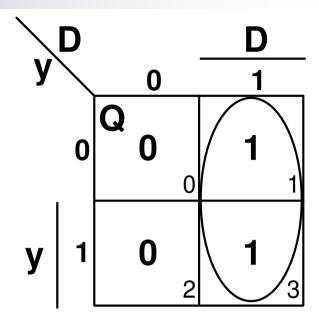
D tároló

Csak szinkron módon értelmezhető

Működése:

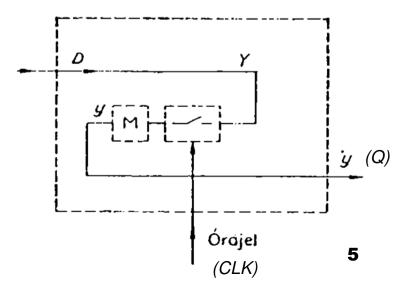
- D=`0' == D-FF állapota változik ('0'-t
 tárol)
- D=`1' == D-FF állapota változik ('1'-et tárol)
- Tehát egy órajel ciklus (CLK) ideig tároljuk a bemenetre érkezett értéket, változás a CLK élére történhet





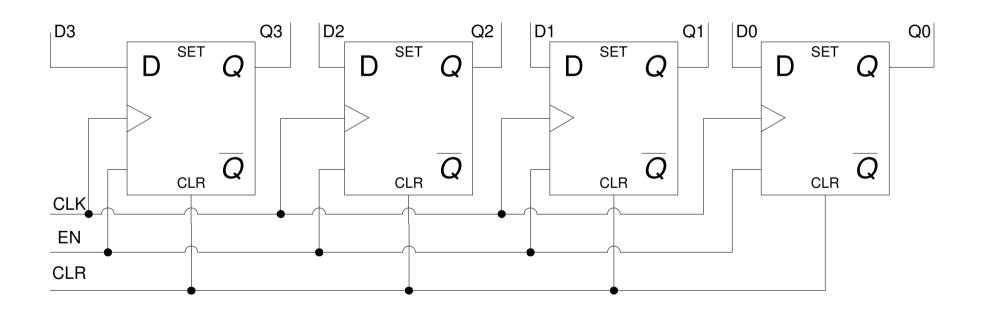
Egyszerűsített DNF alak és elvi logikai rajz:

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





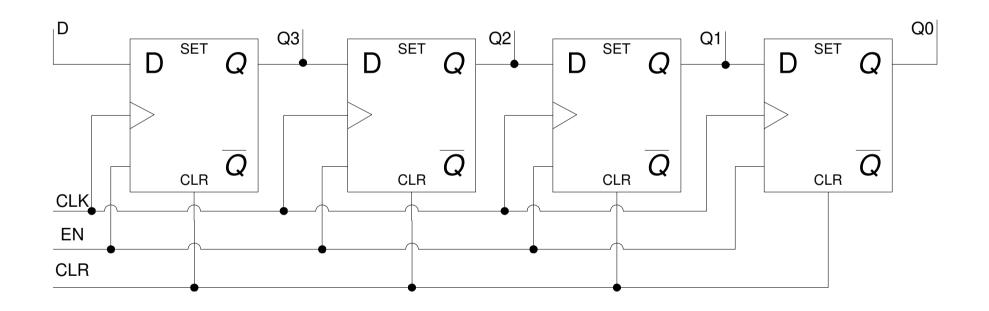
a.) 4-bites Parallel In/ Parallel Out regiszter (D-tárolókból felépítve)



Katalógus adat: SN54/74LS175



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



Katalógus adat : SN54/74LS95



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...2⁵-1)=0...31
 - Maximálisan P=31×31=961 lehet (→ 10 biten ábr.)
- Tehát: N-bites számok szorzatát 2×N biten tudjuk eltárolni!

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

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

.

Iteratív szorzási műveletek:

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

(10100×100)						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				
	DD		07.000.00	oozlopak ö	007000					

az egyes oszlopok osszege

■ Fordított sorrendű (MSB → LSB):

	A4	A 3	A2	A1	A 0				
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



Példa:

- Számolja ki szorzás műveletével az A*B = P eredményét (ha N=4 bit, LE, uint), A:=1001, B:=1010 esetén:
 - □ Shift & Add, és
 - □ Fordított sorrendű módszerekkel

Shift & Add

A:	Szorzand	ó 1001
B:	Szorzó	101 <mark>0</mark>
PP	O	0000
PP:	1	1001
PP2	2	0000
PPS	3	1001
P:		01011010

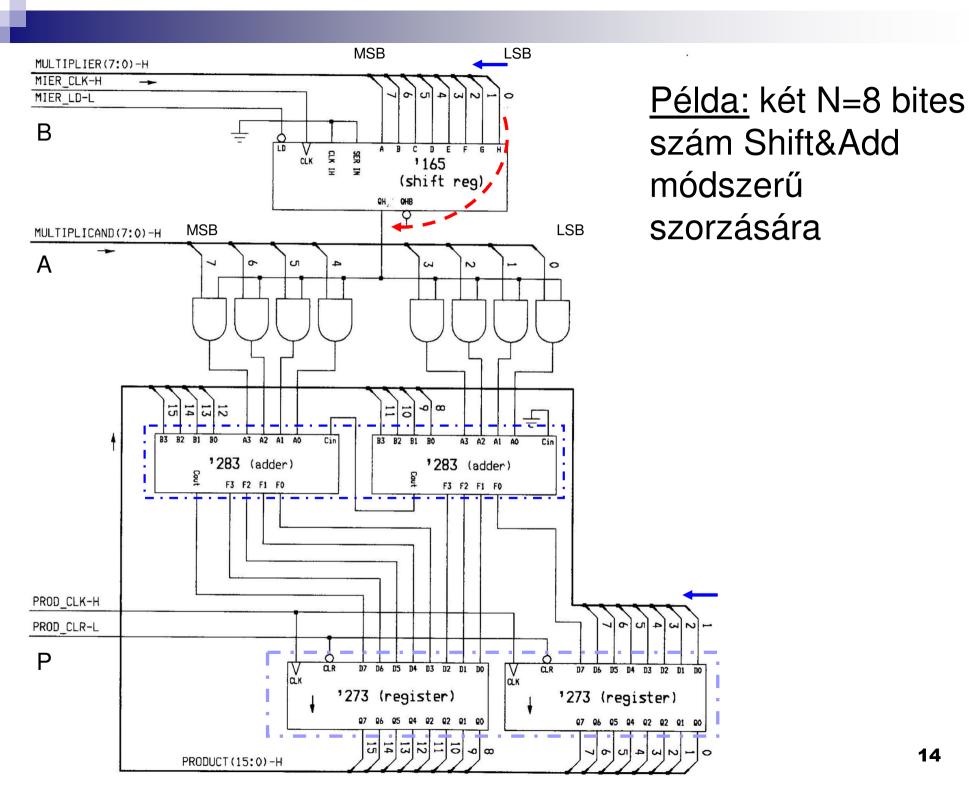
Fordított sorrendű

A:	Szorzandó	1001
B:	Szorzó	<mark>1</mark> 010
PPC)	1001
PP1		0000
PP2	2	1001
PP3	3	0000
P:		01011010

M.

1.) Általános Shift&Add módszer

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





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

Folyamatábra (Shift&Add)

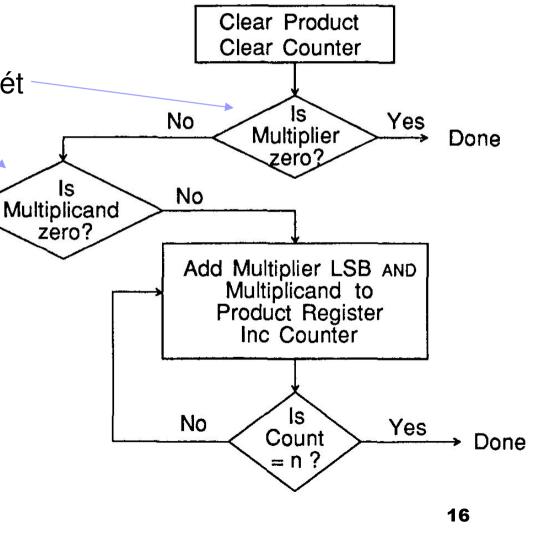
- Alapvetően adatfüggetlen, DE:
- Adatfüggővé tehető gyorsítható algoritmus!
 - Bemenő (A,B) értékek figyelését kell megoldani: zérus-e?
- Időszükséglet:

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

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

Done Yes

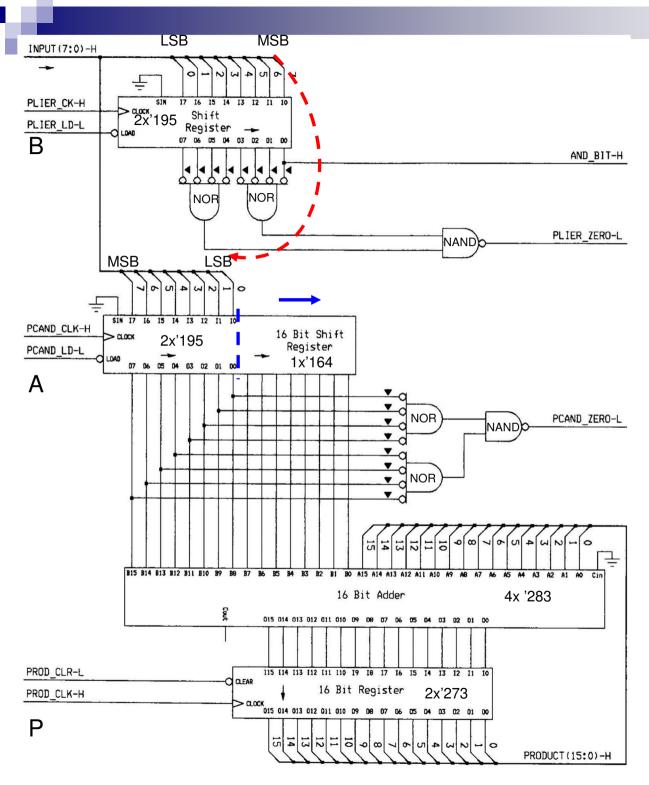
- •T(SETUP): kezdeti ellenőrzések, inicializálás ill. szorzat regiszter törlése
- •T(AND): AND függvények végrehajtása, parciális szorzatok képzése
- •T(SUM): parciális szorzatok összeadása
- •T(REG): betöltésük a regiszterbe





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

10

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).

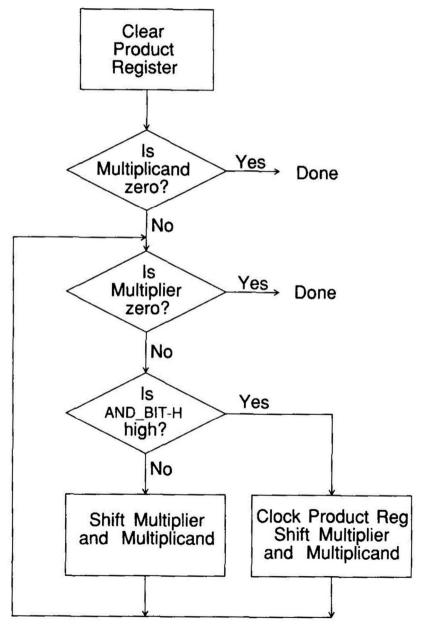


Folyamatábra (fordított sorrendű)

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

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

- •T(SETUP): kezdeti ellenőrzések, inicializálás ill. szorzat regiszter törlése •T(SUM): parcT(AND): AND függyények
- •T(SUM): parcT(AND): AND függvények végrehajtása, parciális szorzatok képzése
- •iális 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

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

□ Újrakódolási technika (séma):

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

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

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

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



- 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:

$$-1 \times (B_0 - 0) = -1$$

$$P0=0-1*A$$

Mivel – volt az érték, ezért kivonjuk a 0-ból az A-t.

$$-2 \times (B_1 - B_0) = +2$$

$$P1=P0+2*A$$

$$-4\times(B_2-B_1)=\underline{-4}$$

$$-8 \times (B_3 - B_2) = 0$$

$$-16 \times (B_4 - B_3) = +16 \mid P4 = P3 + 16 \cdot A$$

$$-32 \times (B_5 - B_4) = -32$$
 P5=P4-32*A

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

D/A

Példa: Booth algoritmus (folyt.)

Végrehajtjuk **P=A×B**-t!

```
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
```

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

011001= 25 "A" Ellenőrzés:

101101= -19 "B" P = -475

// 1110 0010 0101
```

INPUT (7:0) -H В PIER LD-L PIER CLK-H PCAND CLK-H HI Α 1165 8 Bit Shift Register ds 174 amb 1273 8 Bit Register B1-L B1-H B0-H BO-L 1181 181 4 Bit ALU 4 Bit ALU F3 F2 F1 F0 F3 F2 F1 F0 PROD CLR-L 1273 1273 8 Bit Register 8 Bit Register 8 6 6 6 6 6 6 6 6 6 S3 kiválasztó jeleket PRODUCT (15:0) -H

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 shiftelődésének megfelelően generálódnak (egymást követő pozíciókat vizsgálunk). Ezek állítják elő megfelelő PROD_CLK-H kombinációs hálózat P (2 NAND kapu, és 1 Inverter) segítségével az S0-

az ALU-nál.



100

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,
 - □ (bináris) fa modell,
 - □ FA Full Adder felhasználásával,
 - □ CSA: Carry Save Adder (Sorcsökkentős megvalósítás)



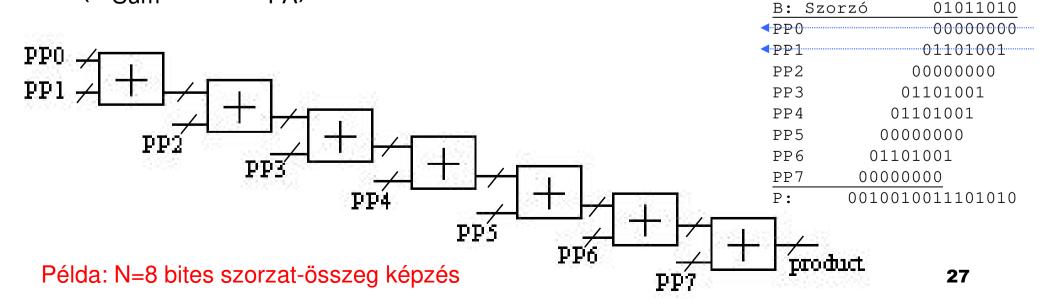
A PPi-k a parciális szorzatképzés után azonnal összeadhatók soronként, így gyorsabban megkapjuk az eredményt. N bites számok esetén (N-1) db összeadóra van szükségünk. Lassabb, mint a következő fa modell, mivel több összeadó szintű a késleltetés.

Időszükséglet:(T_{Sum} = N×T_{FA})

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

A: Szorzandó

01101001

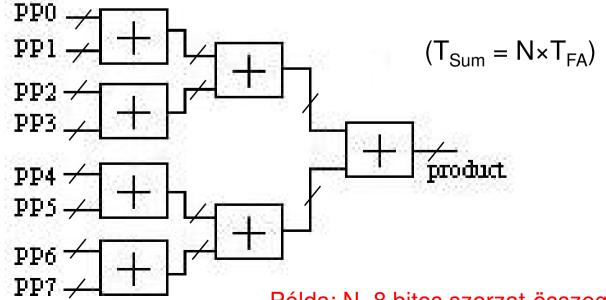




b.) Fa modell ("összeadó fa")

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



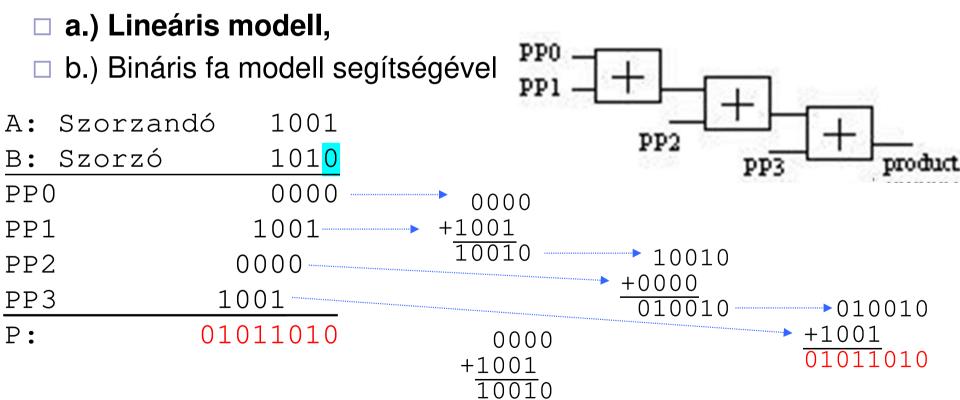
A:	Szorza	ndó 01101001
B:	Szorzó	01011010
PP()	0000000
◀ PP1	<u></u>	01101001
PP2	2	0000000
PP3	3	01101001
PP4	1	01101001
PP5	5	0000000
PP6	5	01101001
PP	7 0	0000000
P:	0.0	10010011101010

Példa: N=8 bites szorzat-összeg képzés



Példa:

 Számolja ki Shift&Add szorzás műveletével az A*B = P eredményét (ha N=4 bit, LE, uint), A:=1001, B:=1010 és adja össze a parciális szorzatokat a:





Példa (folyt):

Számolja ki Shift&Add szorzás műveletével az A*B = P eredményét (ha N=4 bit, LE, uint), A:=1001, B:=1010 és adja össze a parciális szorzatokat a:



/.... fo doll oo ... / ... / ... /

□ b.) Binaris t	a modell segitsegevel	
A: Szorzandó	1001	202 — — — — — — — — — — — — — — — — — —
B: Szorzó	101 <mark>0</mark>	pp3 + product
PP0	0000	0
PP1	$1001 \rightarrow +\frac{1001}{1001}$	0
PP2	0000	
PP3 1	001 000	0 + 10010
P: 01	011010 $+1001$ 1001	01011010

c.) Full Adder-es megvalósítás

 $R_{1,3}$

P₅

FA

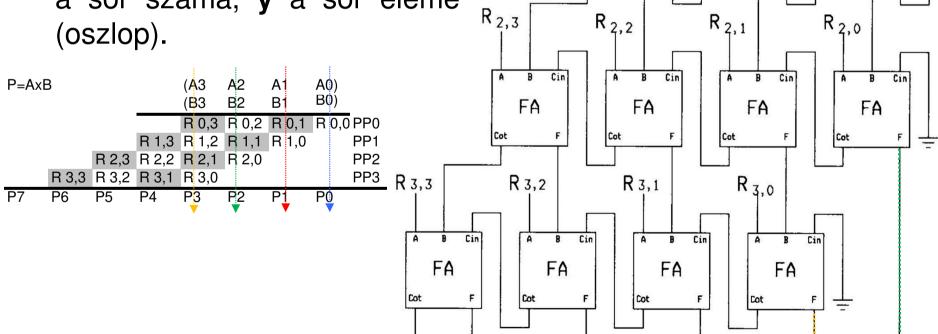
FA

 $R_{1,2} R_{0,3} R_{1,1} R_{0,2} R_{1,0} R_{0,1}$

FA

FA

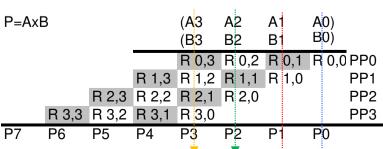
Az ábrán két 4-bites szám szorzását valósítjuk meg. Az oszlopokat, mint parciális szorzatokat adjuk össze FA-k segítségével. Jel: R x,y, ahol x a sor száma, y a sor eleme (oszlop).

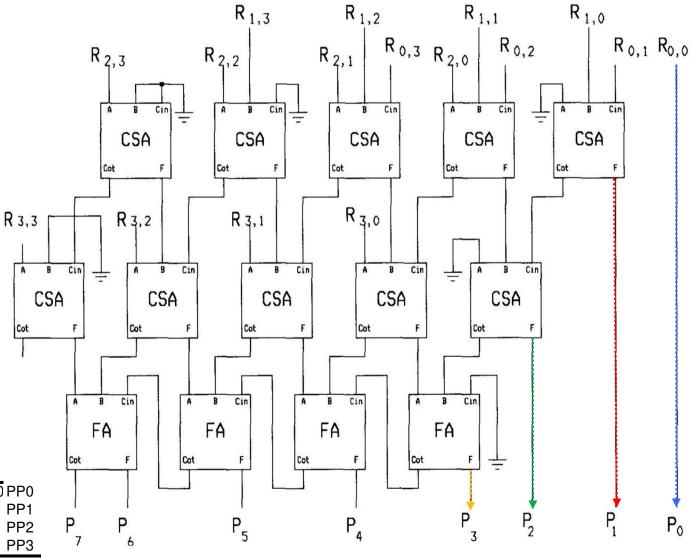


d.) CSA: Carry Save Adder

•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 / CSA*.

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





Példa: N=4 bites szorzat-összeg képzés

32

Osztó áramkörök



Osztó áramkörök:

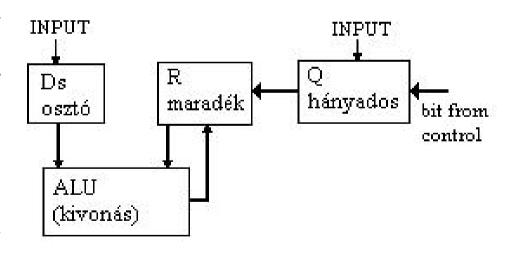
- I.) Hagyományos "lassú", vagy közvetlen iteratív osztási algoritmus
 - □ Euklideszi osztás
- II.) "Gyors" iteratív osztási algoritmusok (egyéb numerikus módszerek)

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
- 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
- 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!

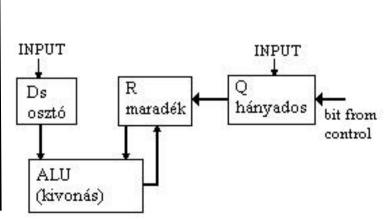
Decimális számok esetén:

Bináris számok esetén hasonlóan

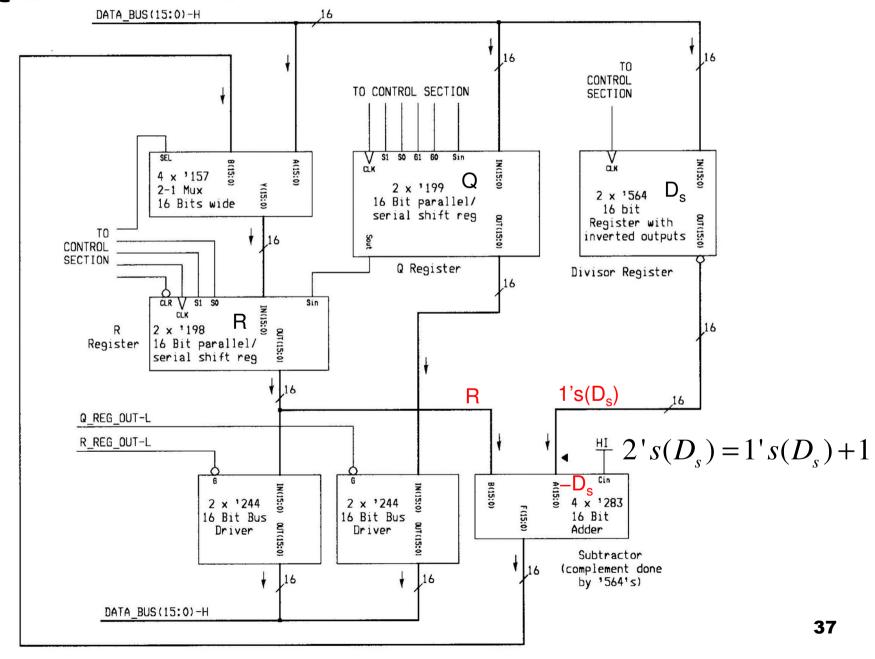
Q hányados (Ds hányszor van meg Dd-ben)

	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
from	100	Ez a kivonás eredménye: 1001-101=100
trol	1000	1000-ban megvan az '101', ezért 1 ? Q
	101	Visszaszorzás 1*'101-el
	11	Kivonás eredménye: 1000-101=11

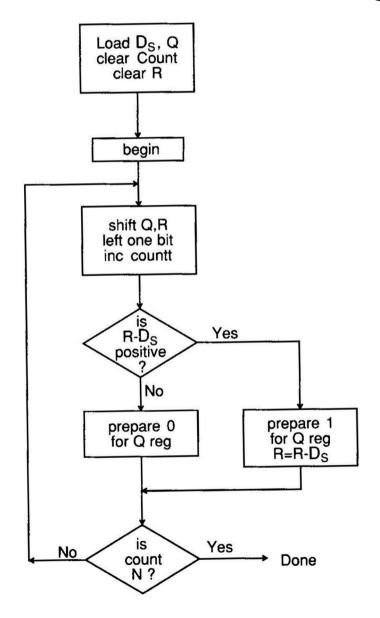
Ez a maradék R!



Hagyományos osztó áramkör



Folyamatábra: 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 x₀ = 10-nek választásával kapjuk:

Várt érték: 24.73863375370...

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

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) f_k –k úgy vannak megválasztva, hogy a nevező az 1-hez konvergáljon.

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

Közvetlen gyors osztó működése

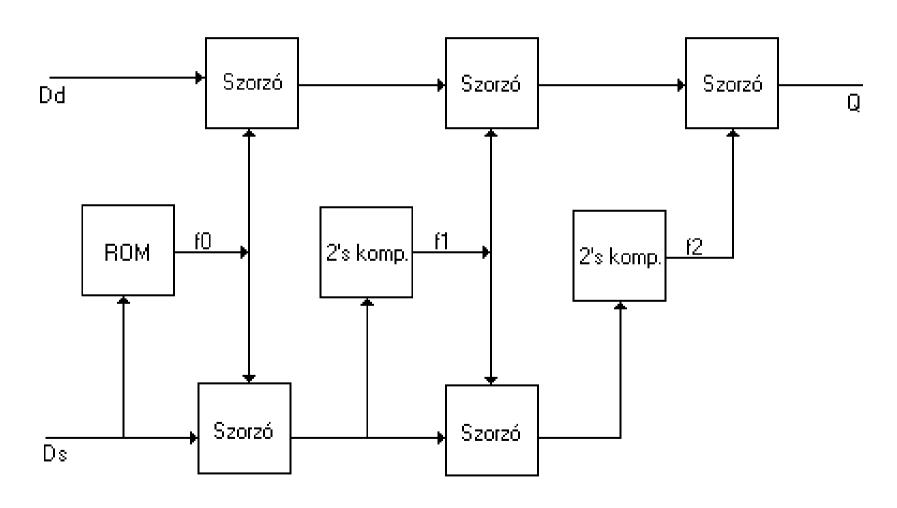
- A számláló iterációja ennél a módszernél $\mathbf{D}_{Dn+1} = \mathbf{D}_{Dn} \times \mathbf{f}_{n}$. A nevező iterációját a megismert módon használjuk a következő f-ek meghatározására.
- Tegyük fel, hogy szorzóink, 2'komplemens képző egységeink vannak, valamint a kezdő értéket tartalmazó ROM, vagy 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 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: 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 $f_o = 1+x = 1+(1-D_S) = 2-D_S$. 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 $D_{Dn+1} = D_{Dn} \times f_n$.
- Egy kérdés vetődik fel: hogyan válasszuk meg f következő értékét. f1 = 1+x² = 1+(1-D_S× f_o)= 2- D_S× 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 kértéke határoz meg. Amikor f 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 fo megfelelő kezdeti értékét tároljuk.
- (Példák: könyvben)

r,e

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 ? $D_{Dn+1} = D_{Dn} \times fn$./

- **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: 0.57142857142857142857

Példa 2.

Legyen az osztandó $\mathbf{D_D}$ = 0.1, osztó $\mathbf{D_S}$ = 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:



Extra bitek kezelése ("kerekítési eljárások")

- 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:

M

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:

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

- 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

■ ∑ ERR → bias: mindig pozitív

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

cases	MR	MF	ERR _{TRUNC}
а	xx0.00	xx0.	0.00
b	xx0.01	xx0.	+0.01
С	xx0.10	xx0.	+0.10
d	xx0.11	xx0.	+0.11
е	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 (0.1₂) hozzáadjuk az összeghez:

Nagyobb mantissza

101010

+ 110010

2 bitpoz. igazítva

11011010

8 bites eredmény

1/2 LSB pozíció
(=rounding!)

Végeredmény (majd truncate!)

Extra bits

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}	
а	xx0.00	xx0.10	xx0.	0.00	
b	xx0.01	xx0.11	<u>x</u> x0	±0.01	
С	xx0.10	xx1.00	xx1.	-0.10	
d	xx0.11	xx1.01	x <u>x</u> 1•	I <u>-</u> 0.01	
е	xx1.00	xx1.10	xx1.	0.00	Változás a
f	xx1.01	xx1.11	xx1	+0.01	truncate-hez
g	xx1.10	xy0.00	xy0.	-0.10	képest!
h	xx1.11	xy0.01	xy0.	I -0.01	Nopost:

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-tozero bias).
- ERR_{ZERO} –kat összeadva a teljes bias értéke nulla lesz.
- legkisebb a hibája (bias = 0), szemben a többi extra-bit kezelő technikával!



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.

		MR+1/2	truncated	
case	MR	LSB	MF	ERRZERO
а	xx0.00	xx0.10	xx0.	0.00
b	xx0.01	xx0.11	xx0.	+0.01
С	xx0.10	xx1	xx1.	-0.10
d	xx0.11	xx1.01	xx1.	-0.01
е	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: ∑ ERR Round to Zero = 0!

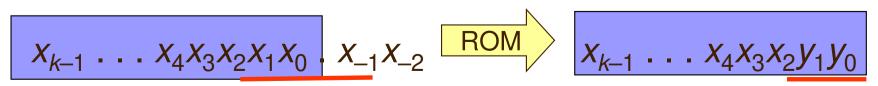
Normál kerekítéshez képest a **c.)**/**g.)** eseteknél egy '1'-es lett **direkt beállítva** (force) az LSB helyén (xx1). De csak a **g.)** esetnél lesz más a hiba értéke (ERR_{ZERO}).

d.) Jamming (~fix értéken rögzítés)

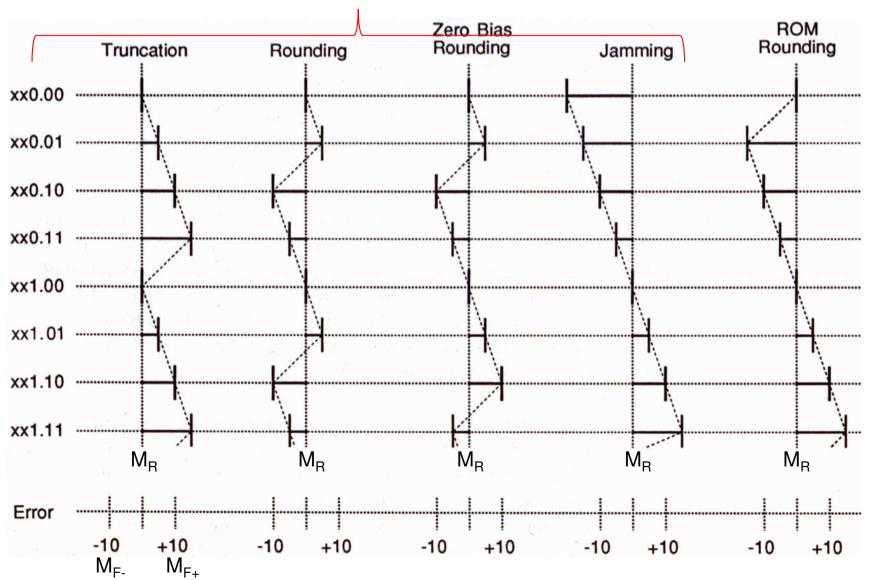
- "Von Neumann rounding" néven is ismert.
- Pl. *Jam to '1':* LSB bitet fix-en '1'-re rögzítik, az extra bitek értékétől függetlenül!
- Cél: Csökkenteni az összhibát (jobb módszer, mint a truncation).
 - □ Ennek a módszernek ugyan nagyobb a hibája, mint a legtöbb extra bit kezelő módszernek, de idővel 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: 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).



Extra-bit kezelő módszerek hibáinak összehasonlítása:



59