Tartalom

1. Numerikus információ ábrázolás: egész/fix/lebegőpontos

Egész:

- Bináris számrendszer: ’1’/’0’ (I/H, T/F)
- N biten 2^N lehetséges érték reprezentálható

Előjel nélküli egész:

- $V_{UNSIGNED\ INTEGER} = \sum_{i=0}^{N-1} b_i \times 2^i$
- Itt b_i az i. pozícióban lévő ’0’ vagy ’1’
- Reprezentálható értékek 0-2^N-ig terjednek
- Helyiértékes rendszer
- Negatív számok nem ábrázolhatók

1’s komplemens rendszer:

- V értékű, N bites rendszer: $2^N - 1 - V$
- Előállítása: Eredeti bitmintázat bitenkénti negálása (pl. 1100 esetén 0011)
- Gyors (csak negálást hajtunk végre)
- Reprezentálható értékek $2^{(N-1)}$-1-től –($2^{(N-1)}$-1)-ig terjednek
- Nem helyiértékes rendszer (zérus kétféle ábrázolása: -0 és +0 – ellenőrzés szükséges)

Előjeles (2’s komplemens) rendszer

- $V_{2'S\ COMPLEMENT} = -b_{N-1} \times 2^{N-1} + \sum_{i=0}^{N-2} b_i \times 2^i$
- Reprezentálható értékek –($2^{(N-1)}$)-$2^{(N-1)}$-1-ig terjednek
- MSB = ’1’ – negatív

Fixpontos számrendszer:

- Lehet előjel nélküli, vagy előjeles 2's komplemens
- Műveletek:
  - +,-: U.a., mint egész számrendszer esetén
  - *,/: Ellenőrizni kell, hogy a tizedespont a helyén maradt-e

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

  - p: radix (tizedes) pont helye, tizedes jegyek száma
- Számrendszer finomsága: $\Delta r = 2^{-p}$
- Excess kód:
  - Lebegőpontos számok kitevőit (exponenseit) tárolják (NE legyen negatív a kitevő)
  - Előállítása: Eredeti számhoz hozzáadjuk az Excess-N (ahol N lehet 64,127,128…) N tagját (pl. Excess-128 esetében 128-at) és vesszük az így előállt szám bitmintázatát
  - S1+S2=(V1+E)+(V2+E) = (V1+V2)+2E

Lebegőpontos rendszer:

- Matematikai jelölés: (előjel) Mantissza * $\text{Alap}^{\text{Kitevő}}$
- Fixpontosnál nagyságrendekkel kisebb vagy nagyobb számok ábrázolására is mód van.
- Normalizált rendszerek: DEC-32, IEEE-32, IBM-32
    - DEC-32:
        - Számrendszer alapja ($r_e=r_b$): Bináris (2-es)
        - Mantissza hossza (m=p): 24 bit
        - Exponens bitek száma (e): 8 bit
        - Van rejtett bit (HB)
        - Exponens bitek Excess-128-ban tárolva
    - IBM-32
        - Számrendszer alapja: $r_e=2$ (bináris), $r_b=16$ (hexadecimális)
        - Mantissza hossza (p=m): 6 (HB-el együtt), de 1 hexadecimális érték tárolásához 4 bit szükséges, ezért 6*4=24
        - Exponens bitek száma (e): 7
        - Exponens bitek Excess-64-ben tárolva
        - Van HB, tároljuk is
    - IEEE-32
        - Számrendszer alapja ($r_e=r_b$): Bináris (2-es)
        - Mantissza hossza (p<m): 23 (HB ’1’, de nem tároljuk el)
        - Exponens bitek száma (e): 8
        - Exponens bitek Excess-127-ben tárolva

2. Nem-numerikus információ ábrázolása: Hamming kódolás

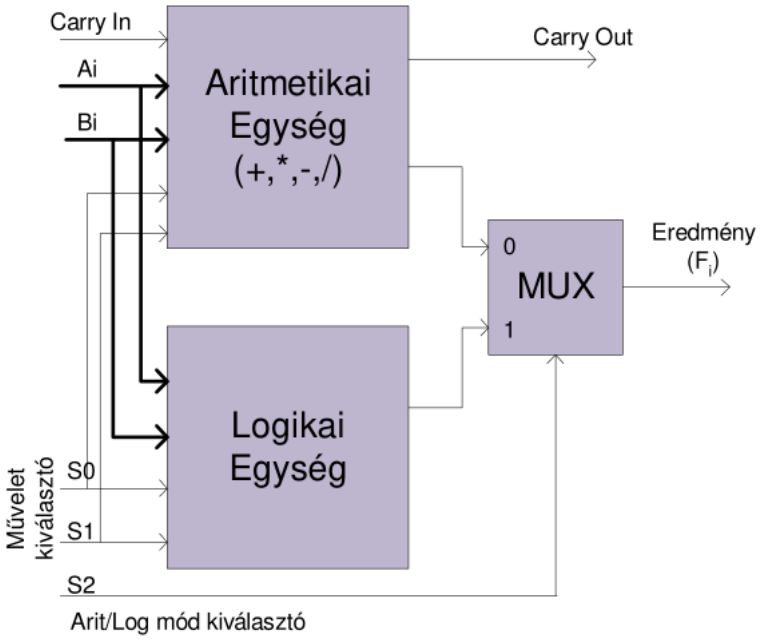
- Min: 14 karakterből álló halmaz – számjegy (0-9), tizedes pont, pozitív ill. negatív jel és üres karakter
- + ábécé (A-Z): Tartalmazza vezérlő karaktereket is (non-printable, mint pl. CR (Carriage Return), LF (Line-Feed) stb…)
- Elemek (46 db) száma 6 biten ábrázolható: $log_2 46 = 6\ bit$
- Kódolások:
    - BCD: 6 biten (nagybetűk, számok, és speciális karakterek)
    - EBCDIC: 8 biten (+kisebetűk és kiegészítő infok)
    - ASCII (alap 7 bit/extended 8 bit)
    - UTF: Változó hosszúságú karakterek

Hamming kódolás:

- Redundáns hibadetektálás és javítás
- Páros paritás: Páros számú 1-esek esetén ’0’, páratlan esetén ’1’
- Páratlan paritás: Páros számú 1-esek esetén ’1’, páratlan esetén ’0’
- Előállítása:
    - LSB!
    - Kódoló bitek (C-k) száma: $2^p >= n+p+1$
    - Felírjuk a bitmintázatot DN-től kezdődően (BE esetén a D-k sorrendjét fordítjuk meg) (D_N, D_{N-1}, …)
    - 2-es súlyok esetén kódoló bitek beszúrása (pl. 1 esetén C0 (2^0), 2 esetén C1 (2^1), 4 esetén C2 (2^2) stb…)
    - Kódolandó bitek bejelölése: Ci-től kezdődően C^i bit bejelölése és C^i kihagyása felváltva.
    - Bitek összegzése soronként és fenti paritás szabályok szerint kódolóbit értékének meghatározása
    - Hibajavítás: Érintett (megváltozott bit oszlopát lefedő) kódoló bitek negálása, és az így előállt kódoló bitmintázat XOR-ozása az eredeti kódoló bitmintával. Ebből áll elő a hiba helye (hanyadik oszlop), majd ez alapján az adott oszlopban lévő bitet negáljuk.
    - Csak egyszeres hiba detektálására és javítására alkalmas

3. ALU felépítése és működése

Felépítése:



Működése:

- Matematikai vagy logikai műveleteket végző egység
- S0-S_n vezérlőjelek jelölik ki a végrehajtandó aritmetikai vagy logikai műveletet
- Ai. Bi az operandusok, melyeken logikai vagy aritmetikai műveletet végzünk.
- Státuszbitek: Hibajelzésre, státuszregiszterben tárolódnak.
- Előjelbit: Eredmény előjelétől függ, előjelregiszterbe töltődik.
- Carry (átvitelt kezelő) bit: Ha egy aritmetikai művelet átvitelt eredményez egyik helyiértekről a másikra, akkor a státuszregiszter beállítja az átvitelt kezelő bitet
- Zéró bit: Ha eredmény nulla, beállítjuk a státuszregiszterben a zéró bitet
- Túlcsordulás (overflow) bit: Jelzi, hogy a rendszer számábrázolási tartományán egy adott aritmetikai művelet eredménye kívül esik-e vagy sem.

4. Összeadó/kivonó áramkörök

A. Összeadó áramkörök

a) Fél-összeadó (Half-adder)

1-bites Half Adder

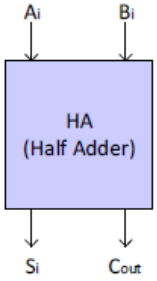
T_{HA} = 1G

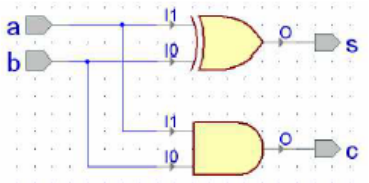
igazságtáblázat

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

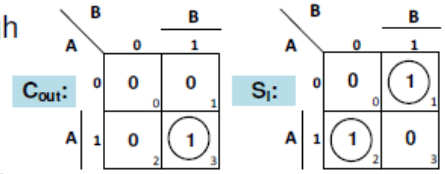
Nem kezeli a Cin-t !

szimbólum





Karnaugh táblái:

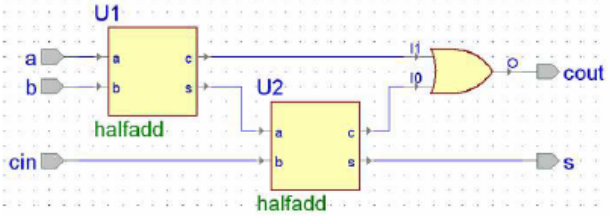


Kimeneti fgv-ei:

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

$$S_i = A_i \oplus B_i$$

1 bites FA felépítése 2 db HA segítségével:



17

b) Teljes összeadó

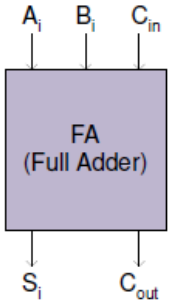
FA: 1-bites Full Adder

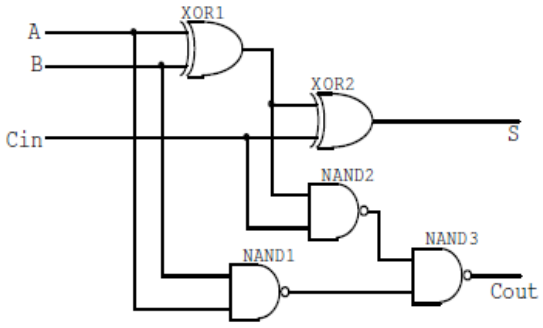
Ez a FA egy lehetséges CMOS kapcsolási rajza: (itt T_{FA} = 3G !)

igazságtáblázat

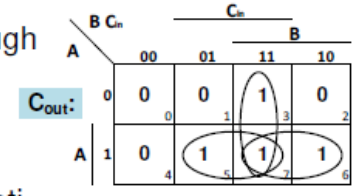
A_i	B_i	Cin	Cout	Sum_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

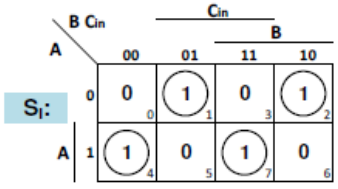
szimbólum





Karnaugh táblái:





Kimeneti fgv-ei:

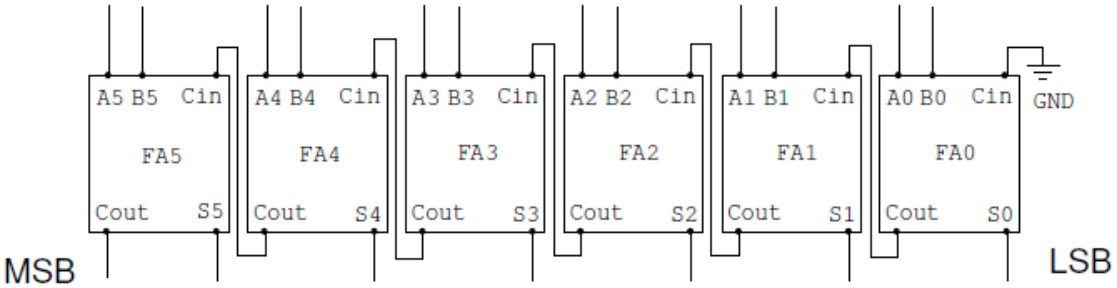
$$C_{out} = A_i \cdot B_i + A_i \cdot C_{in} + B_i \cdot C_{in}$$

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

18

c) Átvitelkezelő összeadó (Ripple Carry Adder- RCA)

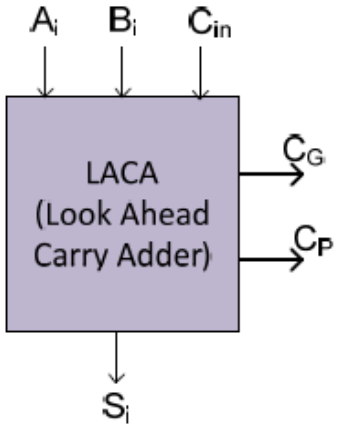
- Pl. 6-bites RCA: [5..0] (LSB Cin = GND!)



- Számítási időszükséglet (RCA):

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

d) LACA: Look Ahead Carry Adder



Képlet (FA) átalakításából kapjuk:

$$C_{out} = A_i \cdot B_i + A_i \cdot C_{in} + B_i \cdot C_{in}$$
$$= A_i \cdot B_i + C_{in} \cdot (A_i + B_i) = C_G + C_{in} \cdot C_P$$

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

LACG (Look Ahead Carry Generator): Egy b bites ALU-hoz kapcsolódik, C_{in} generálásáért felel a CP és CG (LACA-tól) érkező jeleknek megfelelően.

N-bites LACA időszükséglete:

$T_{LACA} = 2 + 4 \times (\lceil \log_b(N) \rceil - 1)$, ahol

N: bitek száma

b: LACG bitszélessége

B. <u>Kivonó áramkörök:</u>

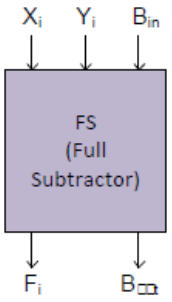
a) Teljes kivonó (Full subtractor (FS))

FS: 1-bites Full Subtractor

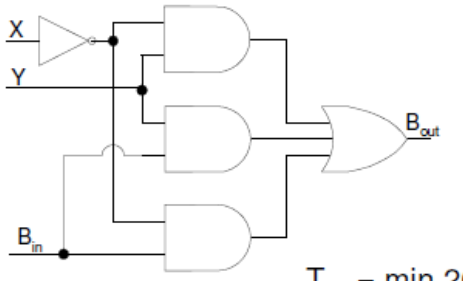
igazságtáblázat

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

szimbólum

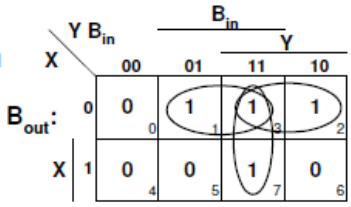


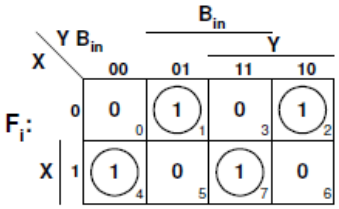
Logikai kapcsolási rajz Bout-ra
(F előállítása ugyanaz, mint FA-nál):



T_{FS} = min 2G

Karnaugh táblái:





Kimeneti fgv-ei:

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

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

25

I. módszer:
Bináris kivonás *FS* segítségével

- $X_i - Y_i \rightarrow F_i$
- 0 − 0 → 0
- 0 − 1 → 1, borrow bit '1'
- 1 − 0 → 1
- 1 − 1 → 0

```
******          *
1000000000      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 komplemens alak):

- FA, RCA, vagy LACA

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

26

5. Szorzó áramkörök

a. **Iteratív szorzó**

i. $P = A \times B$, ahol P:szorzat, A:szorzandó, B:szorzó

ii. N-bites számok szorzatát 2xN biten tudjuk eltárolni

iii. **Általános Shift & Add módszer**

- Parciális szorzat (PPi) összegeket az LSB → MSB bitek felől képzi
- AND kapuk: PPi-k képzése
- Shift-elés: Huzalozott eltolással
- Alapvetően adatfüggetlen (nem figyeli, hogy A,B bemenet zérus-e)
- De adatfüggővé tehető → Gyorsabb
- Időszükséglet: $T_{MULT} = T_{SETUP} + N \times T_{ITER}$, ahol $T_{ITER} = T_{AND} + T_{SUM} + T_{REG}$
- Fordított sorrendű: PPi-k MSB → LSB bitek felől képzése, adatfüggőség (zérus szorzó vagy szorzandó esetén nem végzi el a szorzást)

iv. **Előjeles szorzás Booth-algoritmussal**

- Példa: 543210 (bitpozíciók)

011001= 25 „A”

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

Az újrakódolást bitpárokon végezzük el:

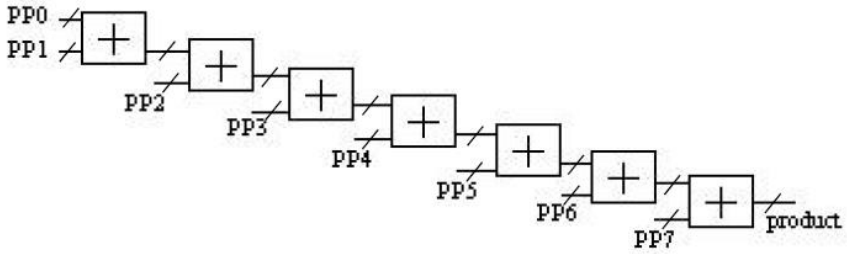
$-1\times(B_0-0)= \underline{-1}$	P0=0–1*A	Mivel – volt az érték, ezért kivonjuk a 0-ból az A-t.
$-2\times(B_1-B_0)= \underline{+2}$	P1=P0+2*A	Mivel + volt az érték, ezért hozzáadjuk P0-hoz a 2*A-t.
$-4\times(B_2-B_1)= \underline{-4}$	P2=P1–4*A	kivonjuk
$-8\times(B_3-B_2)= \underline{0}$	P3=P2	Mivel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

végeredmény értéke

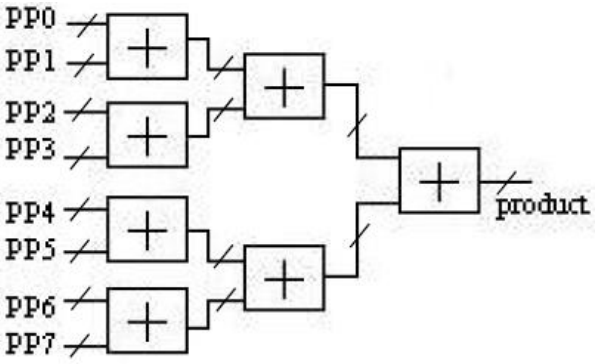
$$P_{n+1} = P_n - 2^n \times (B_n - B_{n-1}) \times A$$

b. **Közvetlen szorzási módszerek**

- **Lineáris modell:**
  - PPi-k a parciális szorzatképzés után azonnal összeadhatók soronként → gyorsabban megkapjuk az eredményt
  - N bites szám → N-1 db összeadó
  - Időszükséglet: $T_{(DIRECT-LINE)} = (N-1) \cdot T_{SUM}$, ahol $T_{SUM} = N \times T_{FA}$

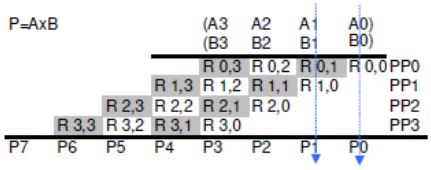


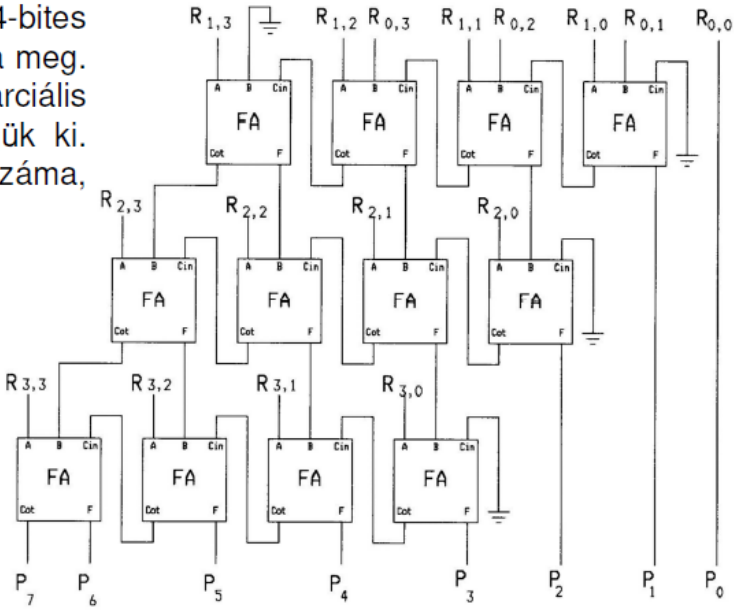
- **Fa modell:**
  - Gyorsabb, mint a lineáris modell
  - N = 8 bit esetén csak 3 szintű a hierarchia → kevesebb késleltetés
  - N bites szám → N-1 db összeadó
  - Időszükséglet: $T_{DIRECT-TREE} = \lceil \log_2(N) \rceil \cdot T_{SUM}$, ahol $T_{SUM} = N \times T_{FA}$



– **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).





6. Osztó áramkörök:

a. **Hagyományos közvetlen osztási algoritmus**

i. Osztó → Ds regiszter, Osztandó → Dd regiszter

ii. R (maradék) regiszter törlése

iii. R-Ds; Ha R-Ds>0 → Folytatás: megváltozott érték az R-be, ’1’-es a Q regiszterbe; Ha R-Ds<0 → R regiszter értéke változatlan, ’0’ a Q regiszterbe vagy osztandó bit hiányában osztás vége

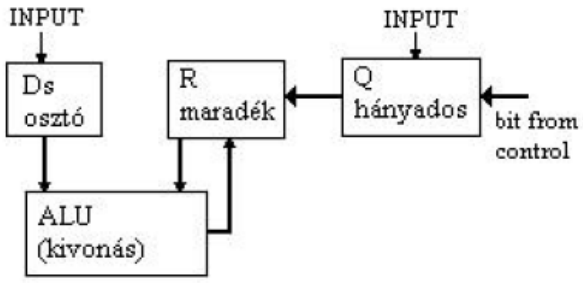
iv. Minden iterációkor egy-egy új bit létrehozása, Q regiszterbe shiftelése, és Dd (osztandó) R regiszterbe töltése

v. Osztandó MSB-jétől kezdve összehasonlítás

vi. MSB felől elsőként hányados generálódik, és a Q-ba shiftelődik 1 bittel balra

vii. Végén: Maradék R-ben, hányados Q-ban

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



b. **Gyors osztás Newton-Raphson módszerrel:**

- Gyorsabb reciprokképzéssel valósul meg az osztás
- Formulája: $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$
- x_0 (kezdeti érték) helyes megválasztása fontos
- Reciprokképzést szorzóval és egy kivonóval valósíthatjuk meg
- A függvény Taylor sorának kiterjesztésével a helyes bitek száma megduplázódik
- Megfelelő lépés kiválasztásával a kívánt pontosság elérhető
- Példa:
  - $\sqrt{612} = ?$ 612 négyzetgyökét keressük, azonos a következővel:
    $$x^2 = 612$$
  - A következő függvényt átalakítással kapjuk, amely Newton Raphson módszerben használható (gyök keresés, f(x) = 0):
    $$f(x) = x^2 - 612$$
  - Deriváltja:
    $$f'(x) = 2x$$
  - Kezdeti érték x_0 = 10-nek választásával kapjuk:

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

Várt érték: 24.73863375370…

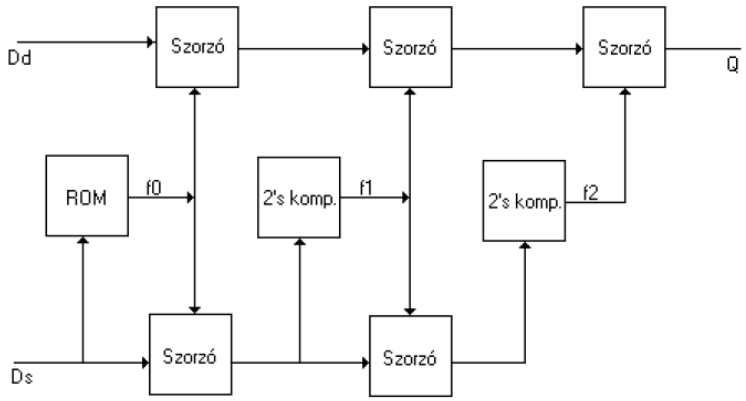
Aláhúzások, már a korrekt számjegyeket jelölik, az egyes iterációkban

c. **Közvetlen gyors osztó**

i. Iteratív osztási módszer

ii. $Q = \frac{D_D}{D_S}$ kiszámolható, ha a successive (egymást követő) f_k-k úgy vanak megválasztva, hogy a nevező az 1-hez konvergáljon

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



7. Digitális építőelemek

a. ALU

- Felépítése:
  - Két különböző n bites bemeneti rész (A, B)
  - n-bites kimeneti vonal (F)
  - Szelektáló bemenetek (S0,S1,…) ➔ Logikai vagy aritmetikai műveletek kiválasztása

b. Memória egységek

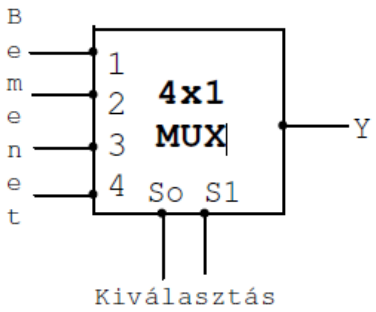
- ALU által kezelt/végrehajtott adatok itt tárolódnak
- Memória rekeszeinek szélessége (w) = adatbusz szélessége
- Pl. legyen n-bit (w) széles és álljon 2^m számú rekeszből. Ekkor m számú címvezetékkel címezhető meg.
- Neumann-architektúrát követi: Utasítások (program/kód) és adatok egy helyen. Programot is adatként tárolja.

c. Adatbuszok – adatvonalak

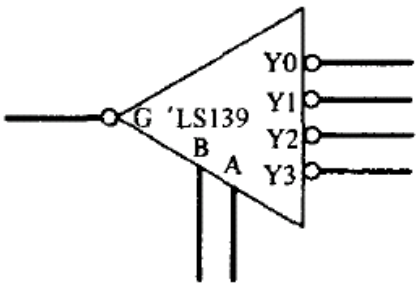
- Fontos paraméter: szélesség ($n \in \mathbb{N}$)
- Pont-pont összeköttetések különböző méretű és sebességű eszközök között
- Közvetlen kapcsolat ➔ nagy sebesség, de egyben rugalmatlanság a bővíthetőségben
- Több adatút ➔ adatbusz (különböző jelvezetékek információinak összefogása)

d. Multiplexer (MUX)

- N kiválasztó jel ➔ 2^N bemenet, 1 kimenet
- 2^N bemenet közül választ egyet (Y), mint egy kapcsoló. Rendelkezhet EN bemenettel is.



e. Demultiplexer



- Egy bemenet (G)
- Routing control jelek (A,B; Bináris kód)
- 4-kimenet mindegyike F, kivéve a kiválasztott (bemenettől függően ennek az értéke T/F)

f. Dekódoló

- N bemenet esetén 2^N kimenet
- Kódolt bemenetet fejti vissza

g. Kódoló

- Dekódoló ellentéte: bementek kódolt ábrázolásának egy formája
  - Hagyományos dekóder: Egyszerre csak egy igaz bemenet
  - Priority encoder: Egyszerre több igaz bemenet, ezek közül a legnagyobb bináris értékű (prioritású) bemenethez generál kódot

h. Komparátor

- Logikai kifejezés – referencia kifejezés (bináris számok) aritmetikai kapcsolatának megállapítására szolgáló eszköz (pl. 2 n bites szám összhasonlítása)
- Azonosság eldöntéséhez EQ/XNOR/Coincidence operátort használunk

i. Regiszter

- Regiszter szélessége = busz info + memória info + ALU info
- Vezérlőjelek hatására bementén lévő adatokat vagy betölti és ideiglenesen eltárolja, vagy kimenetére rakja a tárolt adatokat és egy plusz vezérlőjel hatására lépteti a benne lévő adatokat
- Típusok:
  - Hagyományos (parallel in/parallel out)
  - Léptető/Shift regiszter (serial in)

8. Utasítás kódolás

- Rendszer tervezéséhez regiszterek, ALU, memória, adatbuszok kevesek a végrehajtás egyes fázisainak ábrázolásánál. Szükség van egy olyan eljárásra, amely leírja ezeket az egyes egységek között végbemenő tranzakciókat.
- Ilyen leíró nyelv az assembly
- A felhasználó/programozó által adatkezelésnél használt utasítások gyűjteményét *gépi utasításkészletnek* nevezzük
- FDE (Fetch-Decode-Execute) mechanizmus:
  - F (Fetch): Utasítások memóriából utasításregiszterbe töltése (regiszter transzfer)
  - D (Decode): utasítás értelmezése, utasítás azonosítása
  - E (Execute): Dekódolt utasítások végrehajtása az adatokon, eredmény vissza a memóriába
- RTL (Register Transfer Language) leírás:
  - Utasítások végrehajtásának leírására használjuk
  - Akciók pontos sorrendjét specifikálják
  - Egyes utasításokhoz megadhatók szükséges végrehajtási idők, ezek összege fogja megadni a teljes tranzakció időszükségletét
- Néhány alapvető tranzakció specifikációja:
  - PC → MAR: Program számláló tartalma a memória cím regiszterbe töltődik
  - PC+1 → PC: PC 1-el inkrementálódik és PC-be visszatöltődik
  - M[MAR] → MBR: M memória adott celláját a MAR címregiszter tartalmával címezzük meg, melynek tartalma az MBR-be kerül
  - MBR → IR: MBR tartalma az IR-be töltődik. Egyidejű többszörös műveletvégzés esetén ezek az akciók összekapcsolhatók
  - IR<3:0> → ALU: Az információnak csak egy része, az IR regiszter 3-0 bitje töltődik az ALU-ba
  - REG[2] → MEM[MAR]: Általános célú regiszter 2. rekesze töltődik a MAR adott rekeszébe, a MAR által mutatott címre mutat az operatív memóriában
  - If(carry == 1) then PC-24 → PC: Feltételes utasítások: Ha átvitel 1, akkor PC 24-el dekrementálódik, és visszatöltődik
  - Else PC+1 → PC: Egyébként 1-el inkrementálódik

- Utasítás formák
  - Zéró című (0 című): (PUSH,POP,ACC) [operátor] (pl. STACK, vagy verem)
  - 1 című:[operátor], [operandus]
  - 2 című: [operátor],[operandus1],[operandus2]
  - 3 című: [operátor],[operandus1],[operandus2],[eredmény]
  - 4 című: [operátor],[operandus1],[operandus2],[eredmény],[következő utasítás]

9. Címzési módok

a. **Direkt címzés (X)**
   - Az utasítás egyértelműen, közvetlenül azonosítja az operandus helyét a memóriában
   - Jel: ADD2 X, Y (X-ben tárolt op1 értéket hozzáadjuk az Y-ban tárolt op2 értékhez, az eredmény az Y-ban lesz)

b. **Indirekt címzés (*X)**
   - Az utasítás közvetett módon (nem közvetlenül az operandus értékére) az operandus helyére mutat egy cím segítségével a memóriában.
   - Jel: ADD2 *X, *Y (* = indirekció) – Az első op1 értékének a címe az X-ben található, második op2 értékének a címe az Y-ban lesz, és az eredmény is az Y-ban tárolódik el

c. **Regiszteres direkt címzés (R_x)**
   - Hasonló, mint a direkt címzés, de gyorsabb, mivel ebben az esetben a memória-intenzív műveletek helyett a köztes eredményeket a gyors regiszterben tárolja, és csak a számítási eredményeket tölti a memóriába

d. **Regiszteres indirekt címzés (*R_x)**
   - Hasonló, mint az indirekt címzés, de sokkal gyorsabb, mivel a memória-intenzív műveletek helyett a köztes eredményeket a gyors regiszterekben tárolja és csak a végén tölti át a memóriába

e. **Verem (Stack) címzés:**
   - Indirekt módszerrel az operandus memóriában elfoglalt helyét a címével azonosítjuk
   - Autoincrement is, mivel a címeket automatikusan növeli**:** *R_x+
   - A stack-et a memóriában foglaljuk le, benne lévő információkra stack-pointerrel (SP) hivatkozunk
   - LIFO (Last In First Out) tároló: Ami legutoljára került be, az kerül ki legelsőnek
   - Stack pointer címe jelzi a TOS verem tetejét, ahol a hivatkozott információ található, illetve címmel azonosítható a következő elérhető hely.

f. **Indexelt címzési mód:**
   - A memóriában lévő operandus helyét legalább két érték összegéből kapjuk meg.
   - A tényleges címet az indexelt bázisértékből, és az általános célú regiszter értékéből kapjuk meg
   - Ezt a módszert használják adatstruktúrák indexelt tárolásánál (pl. tömbök)

10. RISC vs CISC számítógép architektúrák

- **RISC (Reduced Instruction Set Computer)**
  - Példák: Motorola 88000 RISC rendszere, vagy Berkeley Egyetem RISC-I rendszere stb…
  - Csak kívánt alkalmazásra jellemző utasítástípusokat tartalmaz, utasításkészlet összetettségének csökkenése végett kihagytak olyan utasításokat, amelyeket a program amúgy sem használ, ezáltal nő a sebesség
  - Minimális utasításkészletet és címzési módot (csak amit gyakran használ), gyors HW elemeket, optimalizált SW használ
  - Komplex függvények leírására szubrutinokra, és hosszabb utasítássorozatokra van szükség
  - Rendszer erőforrásainak hatékony kihasználása? → Egyszerűbb architektúra megvalósítása
  - Azonos utasításformátum (F-D-E) – Dekódolás → minimális idejű (0)
  - Huzalozott (Hardwired) utasításdekódolás
  - Egyszeres ciklusvégrehajtás
  - LOAD/STORE memóriaszervezés (2 művelet – tölt és tárol, regiszter alkalmazása a gyorsabb kiolvasás érdekében)
  - További architektúra technikák: utasítás pipe-line (párhuzamosítás), többszörös adatvonalak, nagyszámú gyors regiszterek alkalmazásával
- **CISC (Complex Instruction Set Computer)**
  - Nagyszámú utasítás típus, és címzési mód
  - Egy utasítás → több elemi feladat végrehajtása
  - Változó méretű utasításformátum → dekódolónak azonosítania kell az utasítás hosszát, az utasításfolyamból kinyerni a szükséges információt, és ezután tudja végrehajtani a feladatát
  - Szemantikus rés = a gépi nyelv és felhasználó nyelve közötti különbség
  - Megoldás: Új nyelvek, mint pl. Fortran, Lisp, Pascal, C stb… Ezek bonyolultabb problémákat is egyszerűen képesek voltak kezelni. Ezekből komplexebb gépek születtek, amelyek gyorsabbak és sokoldalúak voltak.
  - Compiler = fordító (Felhasználói (magas szintű) nyelv → Gépi (alacsony szintű) nyelv)
  - Komplex program, függvény kevesebb elemi utasítássorozattal is megvalósítható. Memória, vagy regiszter alapú technikát használ
  - Közvetlen memória elérés és összetett műveletek jellemzők a CISC-re