7. OPERATIILE ARITMETICE

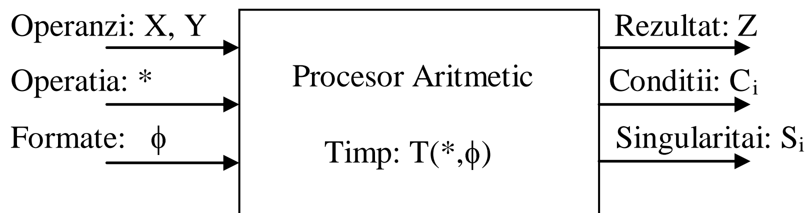
1. Procesorul Aritmetic.

Un procesor aritmetic reprezinta un dispozitiv capabil sa efectueze operatii simple sau complexe asupra unor operanzi furnizati in formate corespunzatoare. Ca exemple se pot da:

- Unitatea Aritmetica simpla;
- Incrementatorul;
- Dispozitivul pentru Transformata Fourier Rapida etc.

Procesorul Aritmetic poate fi examinat atat din punctul de vedere al utilizatorului, cat si al proiectantului.

1.1. Din *punctul de vedere al utilizatorului*, procesorul aritmetic reprezinta o cutie neagra, cu un numar de intrari si iesiri, capabila sa efectueze o serie de operatii asupra unor operanzi cu formate specificate. Rezultatele se obtin intr-un timp, care depinde de tipul operatiei si de formatul operanzilor si sunt insotite de indicatorii de conditii.



Intrari:

- una sau mai multe marimi numerice: X, Y;
- un simbol al operatiei, operatorul: *;
- un simbol de format: ϕ.

Operanzii de la intrare sunt caracterizati prin trei proprieteti:

- apartin unei multimi finite *M* de marimi numerice, caracterizate printr-o gama:

$$X_{min} \leq X \leq X_{max}$$

- sunt cunoscute cu o precizie data:

$$X - \Delta X_l \leq X \leq X + \Delta X_h$$

- sunt reprezentate cu ajutorul unor simboluri/cifre, in cadrul unui sistem de numeratie, sub forma unor *n*-tupluri:

$x_{n-1}\ x_{n-2}......................x_1\ x_0$,

care sunt interpretate ca marimi/valori numerice, pe baza unor reguli date.

Operatorii sunt codificati cu ajutorul unor simboluri *, care corespund unui set redus sau extins de operatii aritmetice:

$$* \in \{+, -, \times, :\}$$

Formatul. Atunci cand sunt posibile mai multe formate, pentru reprezentarea operanzilor, acest lucru poate fi specificat la intrarea *format*, printr-un simbol dat ϕ.

Iesiri:

- una sau mai multe marimi numerice Z, care reprezinta rezultatul;
- unul sau mai multe simboluri C_i, reprezentand conditiile in care apare rezultatul;
- unul sau mai multe simboluri S_i, reprezentand singularitati.

Iesirile numerice poseda aceleasi proprietati ca si operanzii de la intrare: gama, precizie si reprezentare.

Conditiile specifica caracteristici ale rezultatului: < 0, $= 0$, > 0 etc.

Singularitatile sunt asociate cu situatiile in care rezultatul obtinut este invalid:

- *depasire*: rezultatul depaseste posibilitatile hardware de reprezentare a numerelor, in *sistemul numeric dat;*
- *pierderea excesiva de precizie*, la operatiile in virgula mobila;
- *erori datorate hardware-lui*.

In aceste situatii apare un pseudorezultat $Z(S_i)$, impreuna cu singularitatea S_i, care sunt tratate atat prin hardware, cat si cu ajutorul unor rutine specifice ale sistemului de operare.

Timpul de operare T(*) este dat pentru fiecare operatie (*), efectuata de catre procesor. Timpul de operare, in unele cazuri, poate fi variabil:

$$T(*)_{min} \leq T(*) \leq T(*)_{max}$$

Observatii:

- definitia data procesorului aritmetic cuprinde un spectru larg de cutii negre", de la un contor simplu (ADD-ONE), pana la un generator de functii trigonometrice sau un procesor FFT.
- structura interna este specificata in termenii timpului asociat cu executia diferitelor operatii, cat si cu formatul de reprezentare a datelor.
-

1.2. Din *punctul de vedere al proiectantului* intereseaza specificarea detaliata a structurii interne.

Aceasta specificare trebuie sa considere:

- algoritmii aritmetici (proiectarea aritmetica),
- structura logica a procesorului (proiectarea logica).

Proiectarea aritmetica pleaca de la secificatiile date de catre utilizator si le transforma in specificatii de operatii aritmetice detaliate, la nivel de ranguri individuale/bit, in cadrul reprezentarii concrete a datelor. Aceste specificatii, la nivel de rang individual, reprezinta, in continuare, datele initiale (tabele de adevar, diagrame etc) pentru proiectarea logica.

Proiectarea logica pleaca de la specificatiile furnizate de catre proiectarea aritmetica si, in cadrul unei tehnologii date, selecteaza tipurile de circuite logice, pe care le interconecteaza, in mod corespunzator, in vederea implementarii operatiilor aritmetice impuse de catre algoritmii aritmetici. In cazul in care algoritmii aritmetici nu se pot executa intr-un singur pas, se proiecteaza secvente, constand in pasi aritmetici elementari, efectuati sub controlul unor semnale de comanda. Astfel, proiectantul la nivel logic trebuie sa elaboreze, atat proiectul unitatii de executie, cat si proiectul unitatii de comanda.

Specificatiile de tip "black box", pentru proiectarea unui procesor aritmetic, se obtin prin transformarea specificatiilor date de catre utilizator, astfel incat, ele sa corespunda posibilitatilor de implementare. In acest context trebuie sa se aibe in vedere ca:

- datele se reprezinta sub forma unor vectori binari;
- la baza circuitelor, care efectueaza operatiile aritmetice, se afla circuite logice, ce opereaza cu semnale binare.

Avand in vedere cele de mai sus:

- intrarile X si Y vor deveni:

  $X \equiv x_{n-1}\, x_{n-2} \ldots\ldots\ldots\ldots\ldots\ldots x_1\, x_0$,

  $Y \equiv y_{n-1}\, y_{n-2} \ldots\ldots\ldots\ldots\ldots\ldots y_1\, y_0$,

- operatorul (*) va fi codificat printr-un cod de operatie:

  $\Omega \equiv \omega_{n-1}\, \omega_{n-2} \ldots\ldots\ldots\ldots\ldots\ldots \omega_1\, \omega_0$,

  care va indica atat operatia, cat si formatul.

- iesirile reprezinta vectori numerici:

  $Z \equiv z_{n-1}\, z_{n-2} \ldots\ldots\ldots\ldots\ldots\ldots z_1\, z_0$, rezultatul;

  $C \equiv c_{p-1}\, c_{p-2} \ldots\ldots\ldots\ldots\ldots\ldots c_1\, c_0$, indicatorii de conditii

$S \equiv s_{q-1}\ s_{q-2} \ldots\ldots s_1\ s_0$, indicatorii de pseudorezultat.

In continuare se vor examina algoritmii operatiilor aritmetice in virgula fixa si in virgula mobila.

2. Operatiile aritmetice in virgula fixa.

2.1. Adunarea si scaderea.

Operatiile de adunare si scadere ale numerelor in virgula fixa se implementeaza, in majoritatea covarsitoare a cazurilor, cu numere reprezentate in complementul fata de doi. Astfel, operatiile de adunare si scadere se reduc la operatia de adunare a codurilor complementare ale celor doi operanzi. Adunarea se efectueaza rang cu rang, incepand cu rangurile mai putin semnificative, inclusiv rangurile de semn. Transportul, care apare la stanga rangului de semn, se neglijeaza.

Fie operanzii:

$x = +/-\ x_{n-2} \ldots\ldots x_1\ x_0,$

$y = +/-\ y_{n-2} \ldots\ldots y_1\ y_0,$

in conditiile :

$-2^{n-1} \leq x \leq 2^{n-1} - 1$

$-2^{n-1} \leq y \leq 2^{n-1} - 1$

La adunarea/scaderea celor doi operanzi, de mai sus, apar urmatoarele situatii:

a) $x > 0,\ y > 0$, dar $x + y \leq 2^{n-1} - 1$

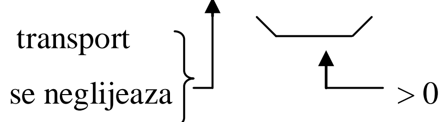
$[x]_c + [y]_c = |x| + |y| = [x + y]_c$

Exemplu:

$[0011]_c + [0010]_c = |0011| + |0010| = |0101| = [0101]_c$

b) $x < 0,\ y > 0$, dar $0 < x + y \leq 2^{n-1} - 1$

$[x]_c + [y]_c = 2^n - |x| + |y| = [x + y]_c$



Exemplu:

$[-0110]_c + [0111]_c = 2^4 - |0110| + |0111| =$

$10000 - |0110| + |0111| =$

$1010 + 0111 = 1\ 0001 = 0001$

transport, se neglijeaza

c) $x < 0,\ y > 0$, dar $x + y < 0$

$[x]_c + [y]_c = 2^n - |x| + |y| = [x + y]_c$

(underbrace over $- |x| + |y|$, pointing to: < 0)

Exemplu:

$[-0111]_c + [0110]_c = 2^4 - |0111| + |0110| =$

$10000 - |0111| + |0110| =$

$1001 + 0110 = [1111]_c$

d) $x < 0$, $y < 0$, dar $|x| + |y| \leq 2^{n-1} - 1$

$[x]_c + [y]_c = 2^n - |x| + 2^n - |y| = [x + y]_c$

transport se neglijeaza (first 2^n); (underbrace over $- |x| + 2^n - |y|$, pointing to: < 0)

Exemplu:

$[-0011]_c + [-0010]_c = 2^4 - |0011| + 2^4 - |0010| =$

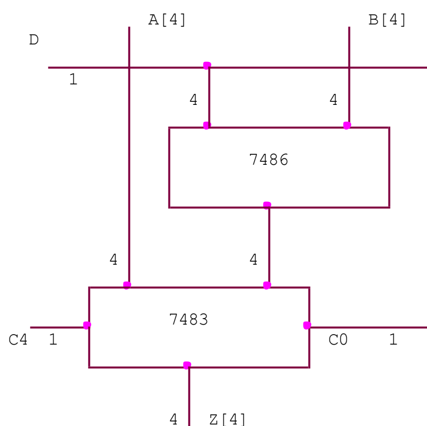
$10000 - |0011| + 10000 - |0010| =$

$1101 + 1110 = 1\ 1011 = [1011]_c$

transport, se neglijeaza (leading 1)

Schema unui sumator/scazator

Schema se bazeaza pe utilizarea a doua circuite: XOR (7486) si ADD (7483)



Operatia	Descrierea	Comanda $D \equiv \Omega$
ADD	Z = C4,ADD(A,B)	0
SUB	Z = C4,SUB(A,B)	1

Codul de operatie specifica operatorul: $D \equiv \Omega$

care ia valorile "0" pentru "+" si "1" pentru "-"

Pozitionarea indicatorilor de conditii.

Indicatorii de conditii specifica o serie de proprietati ale rezultatului, care apare in registrul acumulator al rezultatului AC. De regula, ei sunt stocati in bistabile notate cu mnemonice, care formeaza un registru, incorporat in cuvantul de stare al programului/procesului. Indicatorii de conditii specifica diverse situatii:

- rezultat = 0 - mnemonica Z, unde: Z=~(|(AC));
- semnul rezultatului > 0 sau < 0 - mnemonica S, unde: $S = AC_{n-1}$;
- aparitia transportului, la stanga rangului de semn, mnemonica C, unde: $C_n \leftarrow 1$;
- rezultatul verificarii paritatii - mnemonica P, unde: P =~(^(AC).

Pozitionarea indicatorilor de pseudorezultat.

Printre situatiile care conduc la un pseudorezultat, in cazul operarii in virgula fixa, este si aceea cand apare o depasire. Depasirea se manifesta in conditiile in care cei doi operanzi care se aduna au acelasi semn. Daca rezultatul obtinut are un semn diferit de semnul comun, al celor doi operanzi, s-a inregistrat o depasire. Depasirea poate constitui o cauza de intrerupere/suspendare a programului in cadrul careia a aparut. Aceasta situatie este semnalizata sistemului de operare, in vederea luarii unei decizii corespunzatoare.

Situatia de depasire se semnaleaza prin generarea unui semnal D egal cu *suma modulo doi* intre transportul in rangul de semn si transportul in afara rangului de semn, in cadrul registrului acumulator, al rezultataului:

$$D = C_n \wedge C_{n-1}$$

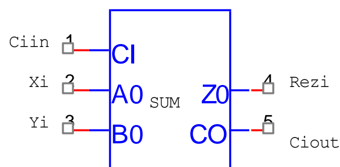
Implementari.

Operatia de adunare pe un singur rang se realizeaza prin generarea sumei si a transportului, folosind circuitele logice necesare realizarii fizice a expresiilor logice de mai jos:

$$Cout_i = (x_i \,.\, Cin_i) \cup (y_i \,.\, Cin_i) \cup (x_i \,.\, y_i)$$

$$Sum_i = (x_i \,.\, \overline{y_i} \,.\, \overline{Cin_i}) \cup (\overline{x_i} \,.\, y_i \,.\, \overline{Cin_i}) \cup (\overline{x_i} \,.\, \overline{y_i} \,.\, Cin_i) \cup (x_i \,.\, y_i \,.\, Cin_i),$$

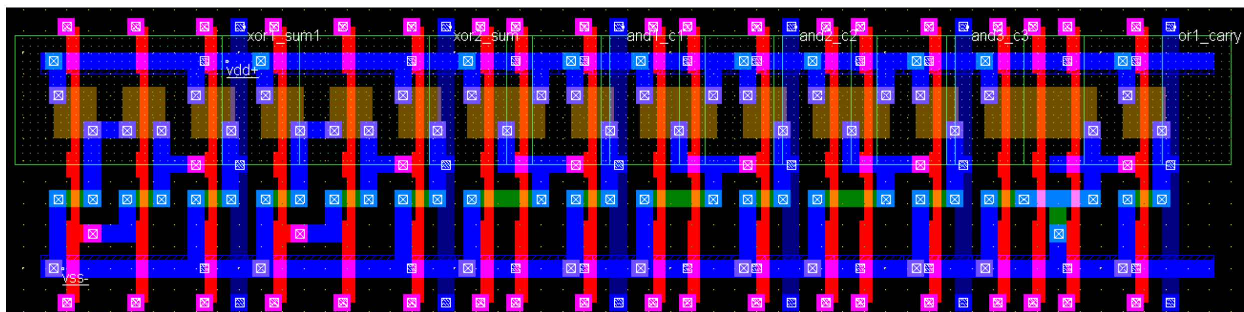
se concretizeaza intr-un circuit de tip “schema bloc”, denumit SUM, prezentat in continuare:



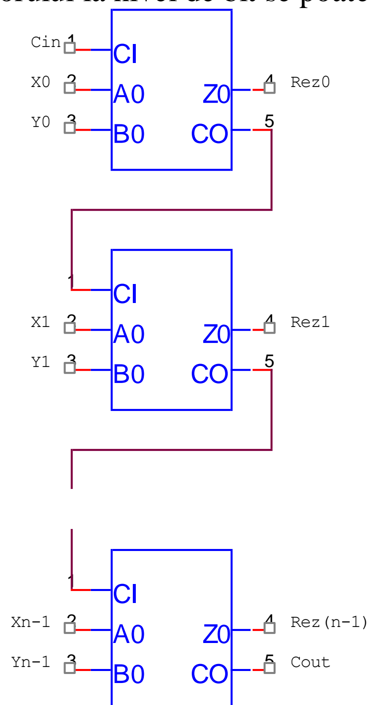
Descrierea in Verilog a unui sumator cu 3 intrari si doua iesiri este urmatoarea:

```
module fulladd(sum,carry,a,b,c);
input a,b,c;
output sum,carry;
wire sum1;
xor xor1(sum1,a,b);
xor xor2(sum,sum1,c);
and and1(c1,a,b);
and and2(c2,b,c);
and and3(c3,a,c);
or or1(carry,c1,c2,c3);
endmodule
```

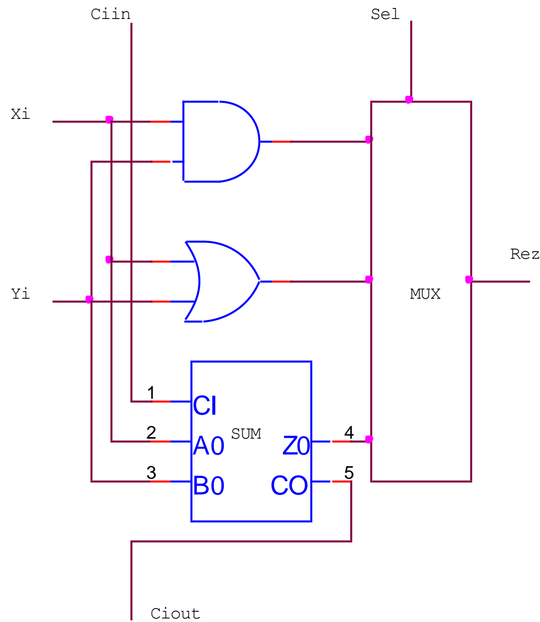
Compilarea acestei descreri la nivel de masti conduce la urmatorul rezultat:



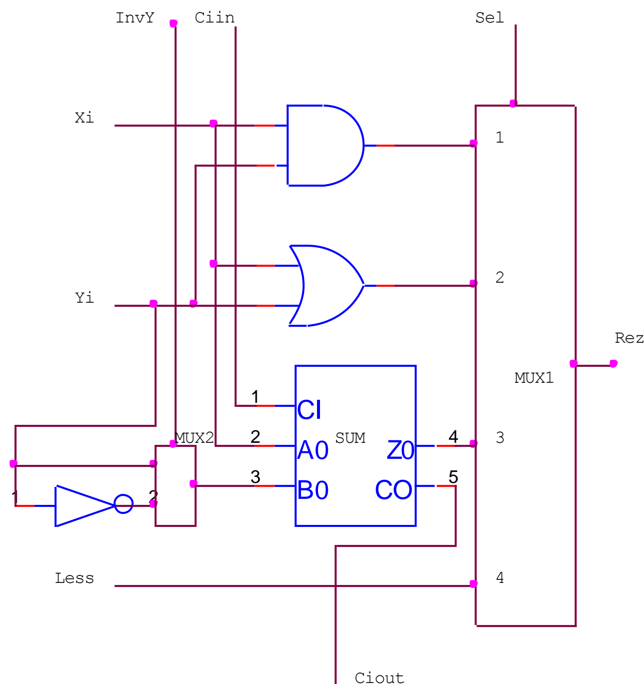
Cu ajutorul sumatorului la nivel de bit se poate realiza un sumator pe *n* biti:



Sumatorul la nivel de bit poate fi extins si cu operatiile logice SI/AND, SAU/OR, ca in figura de mai jos, in cadrul careia iesiriele sumatorului, circuitului AND si circuitului OR sunt aplicate la intrarile unui multiplexor MUX. Codul de operatie, pentru selectarea operatorului, se forteaza la intrarea *Sel*, a multiplexorului.

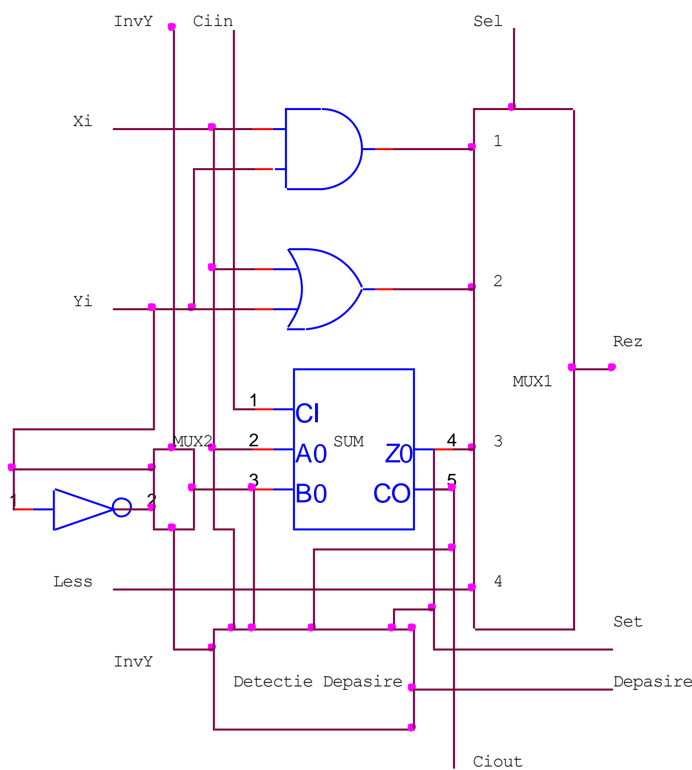


In conditiile in care se doreste si implementarea operatiei de scadere, este necesar sa se creeze posibilitatea inversarii intrarii y_i, dupa cum se prezinta in continuare:



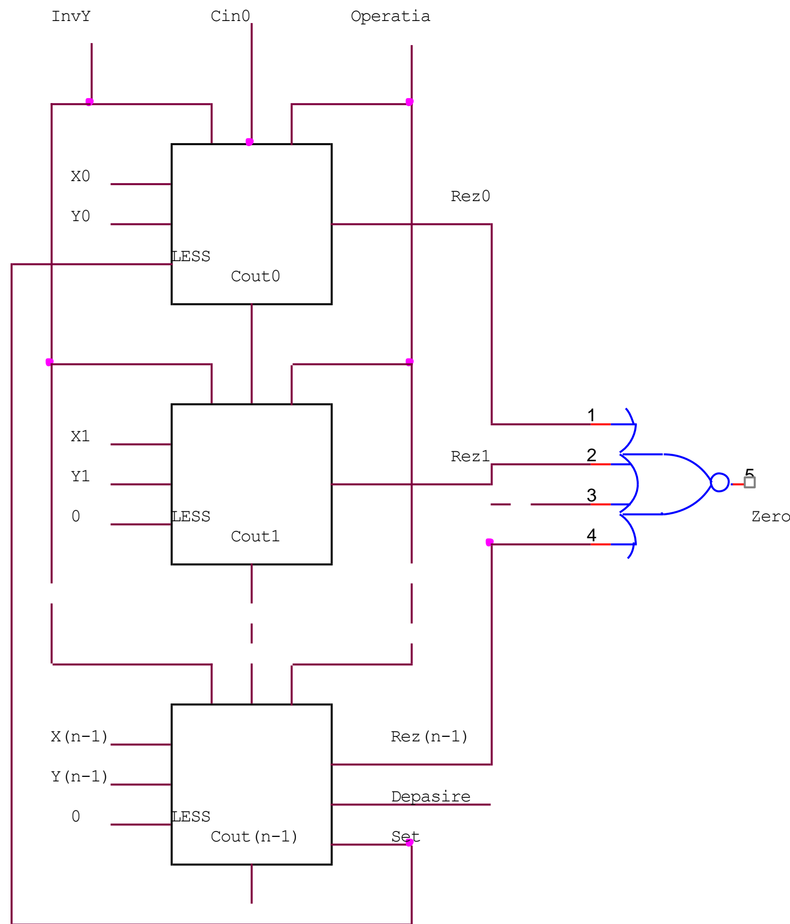
Inversarea lui *y* se realizeaza sub controlul semnalului de selectie *InvY*, aplicat la intrarea de selectie a celui de-al doilea multiplexor

Operatiile ADD, SUB, AND si OR se regasesc in Unitatile Aritmetice Logice ale oricarui procesor. Exista unele procesoare care au implementata instructiunea "*set-on-les-than*", ce se traduce prin "*forteaza in* (1), *bitul cel mai putin semnificativ al rezultatului, daca operandul x este mai mic decat operandul y*", in conditiile in care toti ceilalti biti ai rezultatului vor fi 0. Astfel, in figura de mai sus a aparut o noua intrare la multiplexor "less".
Structura poate fi completata cu detalierea schemei UAL pentru bitul cel mai semnificativ, in care se pune in evidenta depasirea aritmetica.



Implementarea operatiei "*set-on-less-than*" presupune efectuarea scaderii lui *y* din *x*.

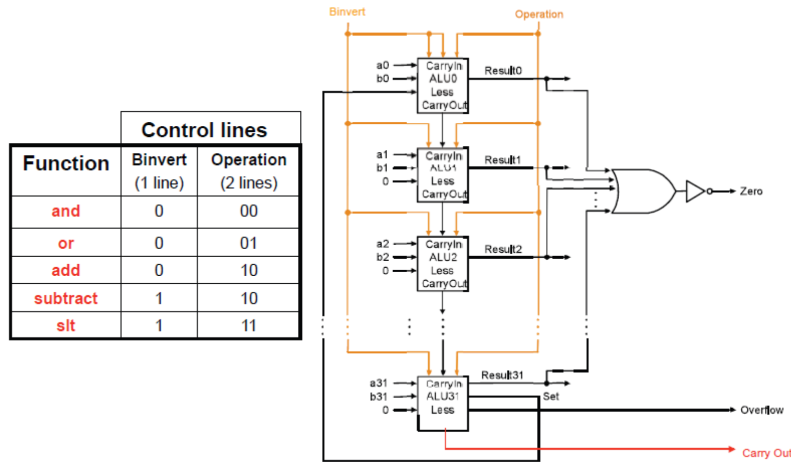
Daca $x < y$, se va pozitiona in 1 semnul rezultatului, adica $Sum_{(n-1)}$. Acest lucru trebuie sa se reflecte insa in fortarea in unu a bitului cel mai putin semnificativ al rezultatului Rez_0 si in zero a bitilor $Rez_{(n-1)}$….. Rez_1. Aceasta solutie este ilustrata in urmatoarea schema bloc.



In figura de mai sus a fost implementa si pozitionarea in "1" , an indicatorului de conditii, care specifica aparitia unui rezultat egal cu "0", la iesirea unitatii arimetice logice. Bistabilul *Z* se pozitioneaza in "1".

Nota: UAL cu 5 operatii:

32-bit ALU with 5 Functions and Zero



Function	Control lines	
	Binvert (1 line)	Operation (2 lines)
and	0	00
or	0	01
add	0	10
subtract	1	10
slt	1	11

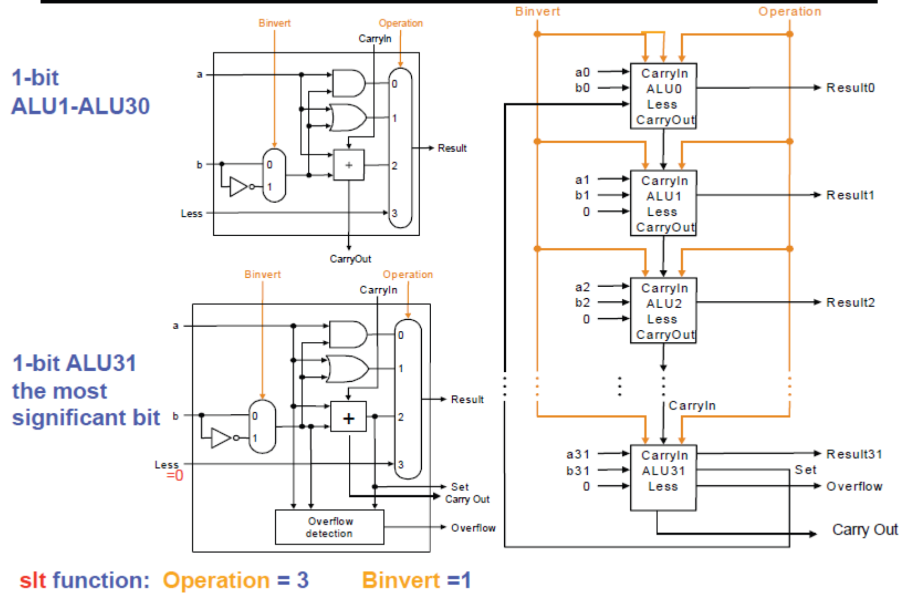
Set Less Than (slt) Function

- slt function is defined as:

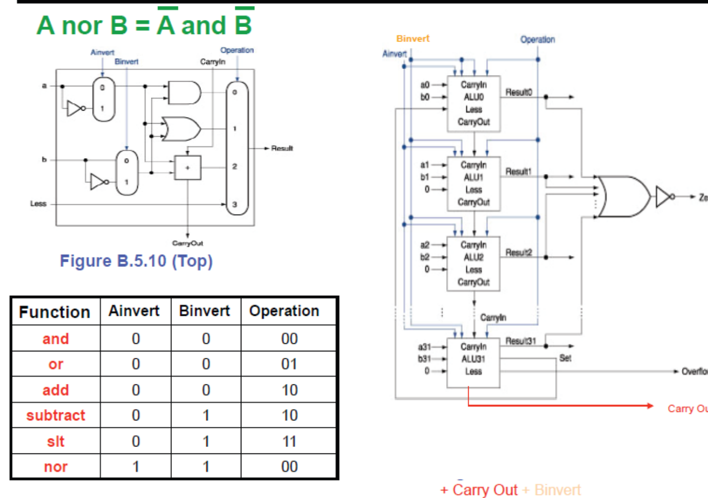
$$A \text{ slt } B = \begin{cases} 000 \ldots 001 & \text{if } A < B \text{, i.e. if } A - B < 0 \\ 000 \ldots 000 & \text{if } A \geq B \text{, i.e. if } A - B \geq 0 \end{cases}$$

- Thus, each 1-bit ALU should have an additional input (called "Less"), that will provide results for slt function. This input has value 0 for all but 1-bit ALU for the least significant bit.
- For the least significant bit Less value should be sign of A – B

32-bit ALU With 5 Functions



32-bit ALU with 6 Functions



Function	Ainvert	Binvert	Operation
and	0	0	00
or	0	0	01
add	0	0	10
subtract	0	1	10
slt	0	1	11
nor	1	1	00

Sumatoare performante.

Sumatorul cu transport succesiv.

Tipul cel mai simplu de sumator paralel este sumatorul cu transport succesiv, obtinut prin interconectarea unor sumatoare complete. Ecuatiile pentru transport si suma, la nivelul unui etaj, sunt date mai jos:

$$Cout_i = (x_i \,.\, Cin_i) \cup (y_i \,.\, Cin_i) \cup (x_i \,.\, y_i)$$

$$Sum_i = (x_i \,.\, \overline{y_i} \,.\, \overline{Cin_i}) \cup (\overline{x_i} \,.\, y_i \,.\, \overline{Cin_i}) \cup (\overline{x_i} \,.\, \overline{y_i} \,.\, Cin_i) \cup (x_i \,.\, y_i \,.\, Cin_i),$$

Se poate observa ca cele doua functii combinationale se implementeaza pe doua niveluri logice. In conditiile in care intarzierea pe un nivel logic este Δt, rezulta ca intarzierea minima pe rang va fi $2.\Delta t$. Intrucat, in cazul cel mai defavorabil, transportul se propaga de la rangul cel mai putin semnificativ pana la iesirea celui mai semnificativ rang, rezulta o intarziere egala cu $2.n.\Delta t$, unde n este numarul de ranguri. Este evident ca o asemenea solutie nu este acceptabila.

Sumatorul cu intarziere minima.

Considerand *ecuatia sumei* , Sum_0 , de la iesirea celui mai putin semnificativ rang, se incearca stabilirea unei expresii a acesteia ca functie de intrarile pentru rangul curent (n-1), cat si de intrarile pentru rangurile mai putin semnificative (n-2),…,1,0. In acest mod se va obtine o expresie logica implementabila in doua trepte.

$$Sum_0 = [x_0 . \overline{y_0} . \overline{Cin_0}] \cup [\overline{x_0} . y_0 . \overline{Cin_0}] \cup$$
$$[\overline{x_0} . \overline{y_0} . Cin_0] \cup [x_0 . y_0 . Cin_0]$$

Pentru iesirea, care reprezinta transportul, din rangul 0 s-a inlocuit $Cout_0$ cu Cin_1, pentru a simplifica relatiile, care urmeaza a se obtine.

$$Cin_1 = [x_0 . Cin_0] \cup [y_0 . Cin_0] \cup [x_0 . y_0]$$

Pentru rangul urmator 1, se obtine urmatoarea expresie a sumei:

$$Sum_1 = [x_1 . \overline{y_1} . \overline{Cin_1}] \cup [\overline{x_1} . y_1 . \overline{Cin_1}] \cup$$
$$[\overline{x_1} . \overline{y_1} . Cin_1] \cup [x_1 . y_1 . Cin_1]$$

In expresia lui Sum_1 se va inlocui Cin_1 cu componentele care-l alcatuiesc, conform formulei precedente. Calculul lui $\overline{Cin_1}$ conduce la urmatoarea expresie:

$$\overline{Cin_1} = [\overline{x_0} . \overline{Cin_0}] \cup [\overline{y_0} . \overline{Cin_0}] \cup [\overline{x_0} . \overline{y_0}],$$

Astfel, pentru Sum_1 se va obtine o expresie formata prin adunarea logica a 12 produse logice de cate 4 variabile fiecare. Aceasta presupune utilizarea unei porti OR cu 12 intrari si a 12 porti AND, cu cate 4 intrari. Extinzand procedeul la urmatoarele ranguri se vor obtine expresii cu numerosi termeni de tip produs, care, la randul lor, vor avea multe componente. Un calcul simplu arata ca, in cazul unui sumator pe 64 de biti, vor fi necesare circa 10^{12} porti, practic de nerealizat in prezent.

Principiul anticiparii transportului.

Intrucat sumatorul cu transport succesiv este lent, iar sumatorul cu intarziere minima este imposibil de realizat, se cauta o solutie intermediara. Aceasta grupeaza relatiile obtinute la sumatorul cu intarziere minima, astfel incat sa se obtina dimensiuni rezonabile.

Ideea de baza este aceea de a caracteriza comportarea unui rang al sumatorului din punctul de vedere al *generarii/propagarii* unui transport. Astfel, rangul *j* al unui sumator genereaza, G_j , transport daca are loc relatia:

$$G_j = x_j \,.\, y_j$$

De asemenea, rangul *j* al unui sumator propaga, P_j , transport in situatia de mai jos:

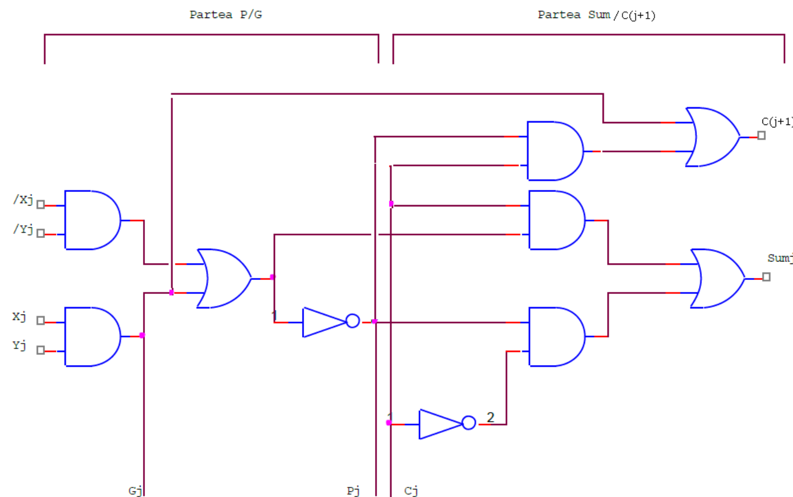
$$P_j = x_j \oplus y_j = (\overline{x_j} \,.\, y_j) \cup (x_j \,.\, \overline{y_j}).$$

In aceste conditii se pot elabora noi expresii pentru transportul $C_{(j+1)}$ si pentru suma Sum_j , la nivelul fiecarui rang al sumatorului:.

$$C_{(j+1)} = G_j \cup (P_j \,.\, C_j)$$

$$Sum_j = (P_j \,.\, \overline{C_j}) \cup (\overline{P_j} \,.\, C_j)$$

Astfel, un sumator complet va consta in doua parti: partea P_j/G_j si partea $Sum_j/C_{(j+1)}$



Relatiile de mai sus se pot structura la nivelul a 4 sectiuni ale unui sumator, bazat pe ideea mentionata anterior. Incepand cu rangul cel mai putin semnificativ se obtin:

- pentru sume:

$$Sum_0 = (\overline{P_0} \,.\, C_0) \cup (P_0 \,.\, \overline{C_0})$$

$$Sum_1 = (\overline{P_1} \,.\, C_1) \cup (P_1 \,.\, \overline{C_1})$$

$$Sum_2 = (\overline{P_2} \,.\, C_2) \cup (P_2 \,.\, \overline{C_2})$$

$$Sum_3 = (\overline{P_3} \,.\, C_3) \cup (P_3 \,.\, \overline{C_3}),$$

si:

- pentru transporturi:

$C_1 = G_0 \cup (P_0 \, . \, C_0)$

$C_2 = G_1 \cup (P_1 \, . \, C_1) = G_1 \cup (P_1 \, . \, G_0) \cup (P_1 \, . \, P_0 \, . \, C_0)$

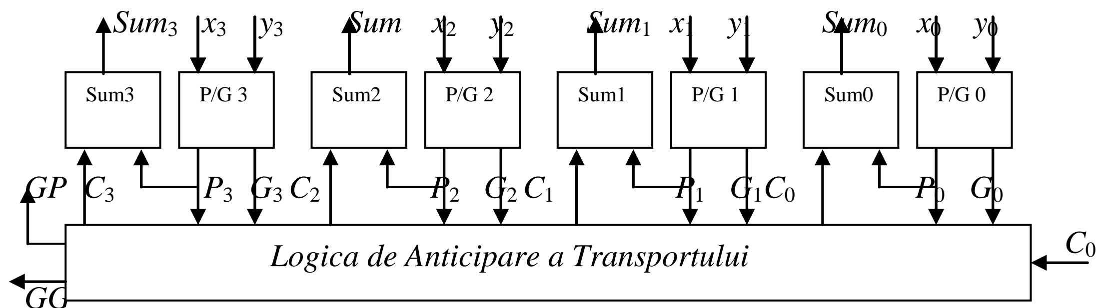
$C_3 = G_2 \cup (P_2 \, . \, C_2) = G_2 \cup (P_2 \, . \, G_1) \cup (P_2 \, . \, P_1 \, . \, G_0) \cup (P_2 \, . \, P_1 \, . \, P_0 \, . \, C_0)$

$C_4 = G_3 \cup (P_3 \, . \, C_3) = G_3 \cup (P_3 \, . \, G_2) \cup (P_3 \, . \, P_2 \, . \, G_1) \cup$

$\cup (P_3 \, . \, P_2 \, . \, P_1 \, . \, G_0) \cup (P_3 \, . \, P_2 \, . \, P_1 \, . \, P_0 \, . \, C_0)$

Aceste relatii sunt implementate in unitatea de anticipare a transportului *UAT*.

Pentru o sectiune de 4 biti, sumatorul cu transport anticipat arata astfel:



Efectuand un calcul al intarzierii, se obtine pe 4 biti de sumator o intarziere de 6.Δt, in comparatie cu intarzierea de 8.Δt, pentru sumatorul cu transport succesiv. Din schema bloc, de mai sus, rezulta ca *Logica de Anticipare a Transportului* mai genereaza doua semnale la nivel de grup. Grupul consta intr-un ansamblu de patru ranguri de sumator. Astfel, grupul poate genera transport (*GG* =1) sau poate propaga transport (*GP* = 1).

Ideea anticiparii transportului se poate fi extinsa atat la nivel de grup, cat si la nivel de sectiune (ansamblu de patru grupuri), realizand o structura piramidala de *UAT*-uri.

Unitatile de anticipare la nivel de grup si la nivel de sectiune sunt identice cu logica de anticipare a transportului la nivelul sumatorului cu patru biti.

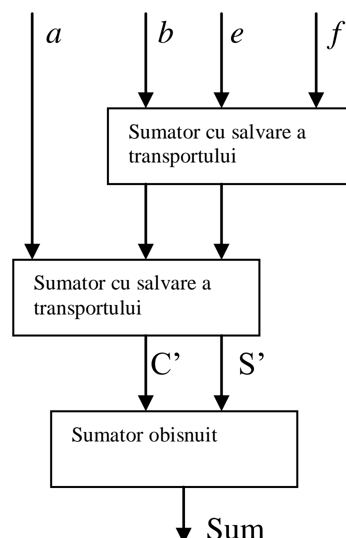
Analiza performantelor arata ca in cazul unui sumator pe 16 biti, cu transport de grup, se inregistreaza o intarziere de 8.Δt, fata de intarzierea de 32.Δt, pentru sumatorul cu transport succesiv. In cazul unui sumator pe 64 de biti, in situatia anticiparii transportului pe sectiuni, se obtine o intarziere maxima de 14.Δt, comparativ cu intarzierea de 128.Δt, pentru sumatorul cu transport succesiv.

Se poate aprecia ca numarul de terminale de intrare si iesire ale portilor poate reprezenta un criteriu de cost, intr-o implementare data. Comparatia cost/performanta pentru cele doua implementari, de mai sus, arata ca in cazul unui sumator pe 64 de biti, la o crestere a numarului de terminale cu 50%, pentru solutia cu transport anticipat, se obtine o viteza de 9 ori mai mare, decat in cazul transportului succesiv.

Sumatorul cu salvare a transportului.

In cazul adunarii mai multor vectori binari simultan se poate recurge la o schema mai rapida, constand in utilizarea mai multor sumatoare in cascada.

Fie cazul adunarii a patru numere de cate 4 biti, notate cu *a*, *b*, *e*, *f*. Rangurile numerelor *b*, *e*, *f* se vor aduna intr-un *sumator cu salvare a transportului*, care va avea iesiri pentru suma si transport. Acestea, la randul lor, se vor aduna cu rangurile numarului *a*, intr-un alt *sumator cu salvare a transportului*. Iesirile reprezentand rangurile sumei si rangurile transportului se vor aduna intr-un sumator obisnuit.



Detaliile de implementare sunt lasate pe seama cititorului.

Comparatie intre cateva tipuri de sumatoare, pe *n* biti, in privinta intarzierii si a ariei ocupate:

Tip sumator	**Intarziere**	**Aria ocupata**
*Trans*port succesiv	$2.n.\Delta t$	$9.n$
Intarziere minima	$2.\Delta t$	$2^{(2n+1)}/(2n+1)$
Transport anticipat	$2.(\log_2 n + 1).\Delta t$	$7.n + 22.\log_2 n$

2.2.Inmultirea.

Inmultirea numerelor consta in generarea si adunarea produselor partiale obtinute prin inmultirea rangului curent al inmultitorului cu deinmultitul. In cazul numerelor binare, produsul partial curent este egal cu deinmultitul, deplasat spre stanga conform pozitiei rangului inmultitorului, daca bitul curent al inmultitorului este unu, sau egal cu zero, daca bitul curent al inmultitorului este zero. Daca produsul partial curent este diferit de zero, el se aduna la suma produselor partiale anterioare.

Exista si posibilitatea ca, dupa fiecare adunare, suma produselor partiale sa fie deplasata spre dreapta cu un rang.

2.2.1. Inmultirea in cod direct.

Inmultirea in cod direct/semn si modul presupune tratarea separata a semnelor si inmultirea modulelor.

Fie numerele X si Y,

$X \equiv x_{n-1}\, x_{n-2} \ldots\ldots\ldots\ldots\ldots\ldots x_1\, x_0,$

$Y \equiv y_{n-1}\, y_{n-2} \ldots\ldots\ldots\ldots\ldots\ldots y_1\, y_0,$

care urmeaza sa fie inmultite, pentru a furniza produsul Z:

$Z \equiv z_{2n-2}\, z_{2n-3} \ldots\ldots\ldots\ldots\ldots z_1\, z_0.$

Semnul produsului z_{2n-1} va fi obtinut astfel:

$z_{2n-2} = x_{n-1} \oplus y_{n-1}.$

Produsele partiale: $P_0, \ldots\ldots, P_{n-3}, P_{n-2}$ se obtin dupa cum urmeaza:

$P_0 = |x| \,.\, y_0 .\, 2^0$

$P_1 = |x| \,.\, y_1 .\, 2^1$

……………………

$P_{n-2} = |x| \,.\, y_{n-2} .\, 2^{n-2}$

iar suma lor va conduce la produsul modulelor:

$z_{2n-3} \ldots\ldots\ldots\ldots\ldots z_1\, z_0$

Fie cazul a doua numere reprezentate in cod direct pe 5 biti:

- deinmultitul $x = 11000$ si
- inmultitorul $y = 01000$

Pentru semn rezulta: $z_8 = 1 \oplus 0 = 1$,

in timp ce modulul produsului se va obtine dupa cum se arata mai jos:

```
     1000
   × 1001
   ------
     1000
    0000
   0000
  1000
  -------
 01001000
```

$z_7 \ldots\ldots\ldots z_1\ z_0 =$ 01001000.

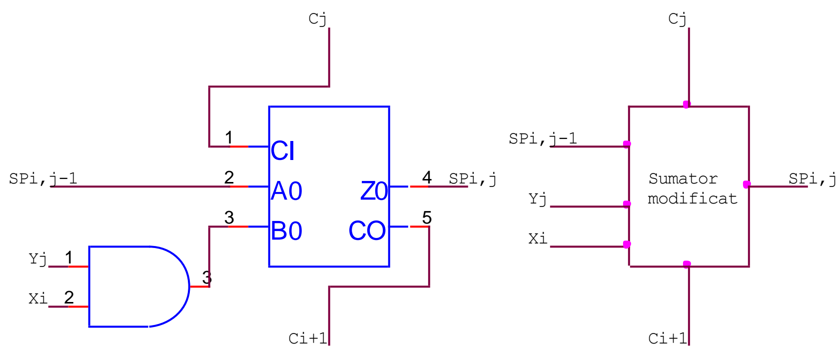
Astfel, produsul celor doua numere va avea 9 biti:

$z_8\ z_7 \ldots\ldots\ldots z_1\ z_0 =$ 101001000.

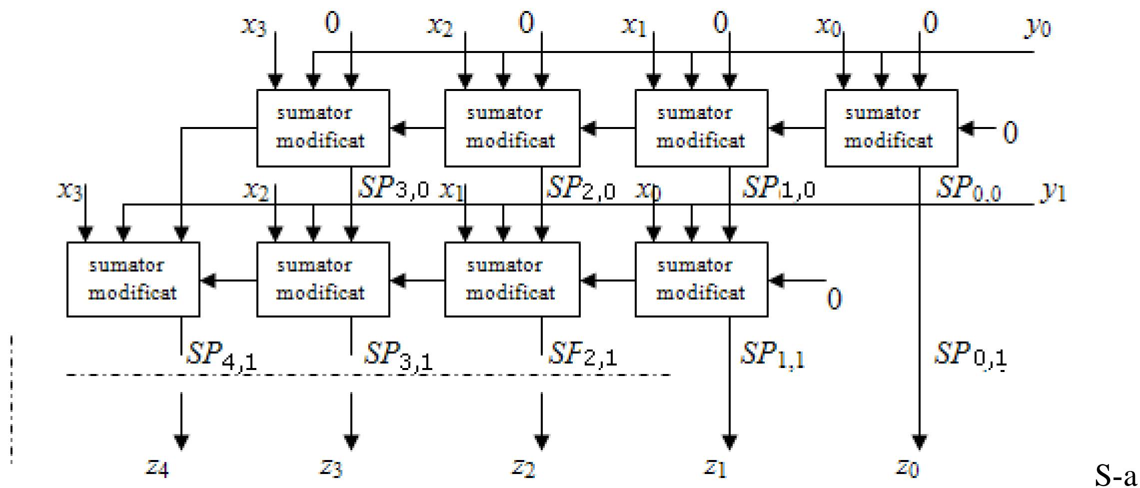
In exemplul de mai sus produsele partiale s-au obtinut prin deplasarea spre stanga a deinmultitului, inmultit cu rangul corespunzator al inmultitorului.

Solutie paralela.

O implementare combinationala presupune utilizarea sumatoarelor complete si a unor porti AND. Astfel, se obtine un sumator modificat, pentru un rang, conform schemei de mai jos:



a) Sumator modificat b) Schema bloc a sumatorului modificat



S-a notat cu $SP_{i,j}$ bitul i al sumei produselor partiale j. Schema prezinta numai primele doua etaje ale dispozitivului paralel de inmultire.

Sumatorul modificat poate fi utilizat in cadrul unei structuri de inmultire paralela, ca in figura de mai sus. Din punctul de vedere al implementarii algoritmului, se poate afirma ca, in acest caz, este vorba de o "programare spatiala", care conduce la viteza ridicata de operare. Solutia necesita, pentru implementare, $(n\text{-}1)^2$ sumatoare modificate. In cel mai defavorabil caz, rezultatul inmultirii se obtine dupa propagarea semnalelor prin $4.(n - 1)$ porti.

Se lasa pe seama cititorului elaborarea unei solutii bazate pe "sumatoare cu salvare a transportului" si sumatoare cu transport anticipat.

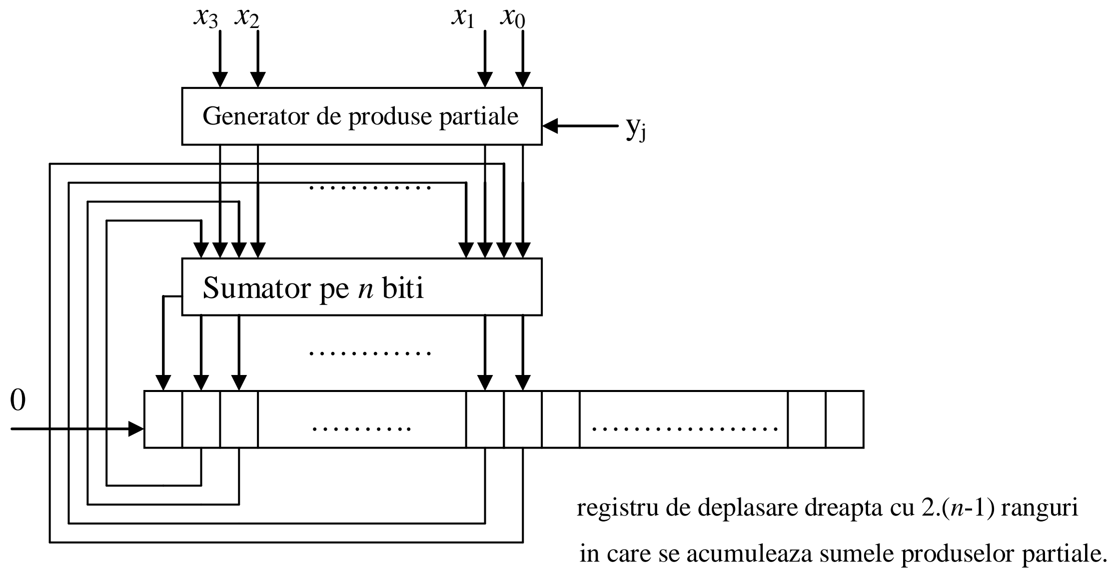
Solutie serial - paralela.

In scopul reducerii cantitatii de hardware, dispozitivele de inmultire se realizeaza sub forma unei structuri paralele hardware, pentru un pas de inmultire, cu operare secventiala.

Daca se noteaza *produsul partial j* cu pp_j, atunci un pas de inmultire va avea urmatoarea forma:

$$pp_j = pp_{j-1} + x \,.\, y_j.\, 2^j$$

O schema bloc, pentru solutia mentionata mai sus, este prezentata in continuare:



registru de deplasare dreapta cu 2.(n-1) ranguri in care se acumuleaza sumele produselor partiale.

Operarea se efectueaza conform urmatoarei secvente:

1. anuleaza continutul registrului in care se acumuleaza sumele produselor partiale;
2. initializeaza numarul rangului bitului inmultitorului $j = 0$;
3. formeaza produsul partial $x \cdot y_j$.
4. aduna produsul partial la jumatatea superioara a registrului sumei produselor partiale;
5. efectueaza $j = j + 1$ si daca $j = n$ treci la 8.
6. deplaseaza la dreapta cu un rang continutul registrului sumei produselor partiale;
7. treci la pasul 3;
8. produsul cu 2.(n –1) ranguri s-a obtinut in registrul de deplasare.

Rezultatul se poate stoca fie sub forma unui singur cuvant, pastrand jumatatea superioara, fie sub forma unui cuvant dublu. In primul caz va fi afectata precizia.

Inmultirea numerelor in cod complementar.

Intrucat, in marea majoritate a cazurilor, numerele negative se reprezinta in calculatoare in codul complementar, in continuare vor fi examinate cateva metode de inmultire in acest cod.

Mai intai se vor examina *deplasarile aritmetice la stanga si la dreapta in cod complementar.*

Se considera urmatoarele cazuri:

$x > 0$.

$[x]_c = 0\ x_{n-2} \ldots\ldots x_1\ x_0$,

Deplasarea la stanga:

$[2.x]_c = x_{n-2} \ldots\ldots x_1\ x_0 0$,

Deplasarea la dreapta:

$[2^{-1}.x]_c = 00x_{n-2}\ldots\ldots x_1,$

$x < 0.$

$[x]_c = 1\tilde{x}_{n-2}\ldots\ldots\tilde{x}_1\,\tilde{x}_0,$

Deplasarea la stanga:

$[2.x]_c = \tilde{x}_{n-2}\ldots\ldots\tilde{x}_0\,0,$

Deplasarea la dreapta:

$[2^{-1}.x]_c = 11\tilde{x}_{n-2}\ldots\ldots\tilde{x}_1.$

Cateva metode pentru inmultirea numerelor reprezentate in cod complementar.

Metoda 1.

- Se modifica , daca este cazul, inmultitorul si deinmultitul astfel incat inmultitorul sa fie pozitiv.
- Produsele partiale se calculeaza in mod obisnuit.
- Deplasarea spre stanga/ dreapta a deinmultitului/sumei produselor partiale se realizeaza conform regulilor de deplasare in cod complementar.

***Metoda* 2.**

Aceasta metoda presupune inmultirea numerelor cu semn in maniera obisnuita, ca si cand ar fi vorba de numere intregi fara semn. Rezultatul va fi corect numai in cazul in care cei doi operanzi sunt pozitivi. In caz contrar sunt necesare corectii, care se incadreaza in trei cazuri.

1. $x > 0,\ y > 0.$

$[x]_c\ .\ [y]_c = |x|\ .\ |y| = [x\ .\ y]_c$

2. $x < 0,\ y > 0;\ [x]_c = 2^n - |x|;\ [y]_c = |y|;$

$[x]_c\ .\ [y]_c = 2^n.|y| - |x|\ .\ |y|$; *rezultat incorect*

$[x]_c\ .\ [y]_c = 2^{2n} - |x|\ .\ |y|$; *rezultat corect*

Rezulta necesitatea unei corectii egala cu: $2^{2n} - 2^n.|y|$, care se va aduna la rezultatul incorect.

3. $x > 0,\ y < 0;\ [x]_c = |x|;\ [y]_c = 2^n - |y|;$

$[x]_c \cdot [y]_c = 2^n.|x| - |x| \cdot |y|$; *rezultat incorect*

$[x]_c \cdot [y]_c = 2^{2n} - |x| \cdot |y|$; *rezultat corect*

Rezulta necesitatea unei corectii egala cu: $2^{2n} - 2^n.|x|$, care se va aduna la rezultatul incorect.

4. $x < 0$, $y < 0$; $[x]_c = 2^n - |x|$; $[y]_c = 2^n - |y|$;

$[x]_c \cdot [y]_c = 2^{2n} - 2^n.|y| - 2^n.|x| + |x| \cdot |y|$; *rezultat incorect*

$[x.y]_c = |x| \cdot |y|$; *rezultat corect*

Rezulta necesitatea unei corectii egala cu: $-2^{2n} + 2^n.|y| + 2^n.|x|$, care se va aduna la rezultatul incorect.

Metoda este greoaie si are un caracter pur teoretic.

***Metoda 3. Algoritmul lui Booth**.*

In acest caz se pleaca de la observatia ca, valoarea unui numar *Y*, reprezentat in cod complementar, se poate calcula astfel:

$$Y = - y_{n-1} \cdot 2^{n-1} + y_{n-2} \cdot 2^{n-2} + y_{n-3} \cdot 2^{n-3} + \ldots\ldots + y_1 \cdot 2^1 + y_0 \cdot 2^0 =$$

$$= \sum_{i=0}^{n-1} (y_{i-1} - y_i) .2^i$$

unde:

- y_{n-1} reprezinta rangul de semn, codificat cu 0/1 in cazul numerelor pozitive/negative;
- y_{-1} constituie rangul aflat la dreapta rangului 0, avand initial valoarea 0.

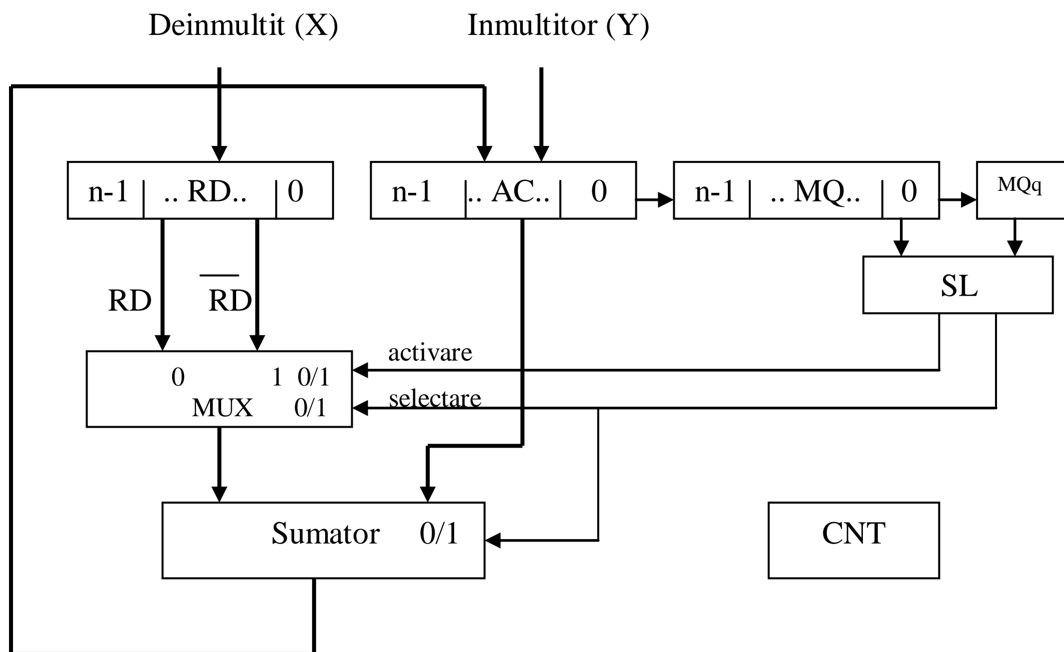
In aceste conditii valoarea produsului X.Y se va exprima dupa cum urmeaza:

$$X.Y = \sum_{i=0}^{n-1} x.(y_{i-1} - y_i) .2^i$$

Pe baza formulei de mai sus se poate calcula produsul partial de rang i :

y_{i-1}	y_i	produsul partial
0	0	0
0	1	$- x .2^i$
1	0	$x .2^i$
1	1	0

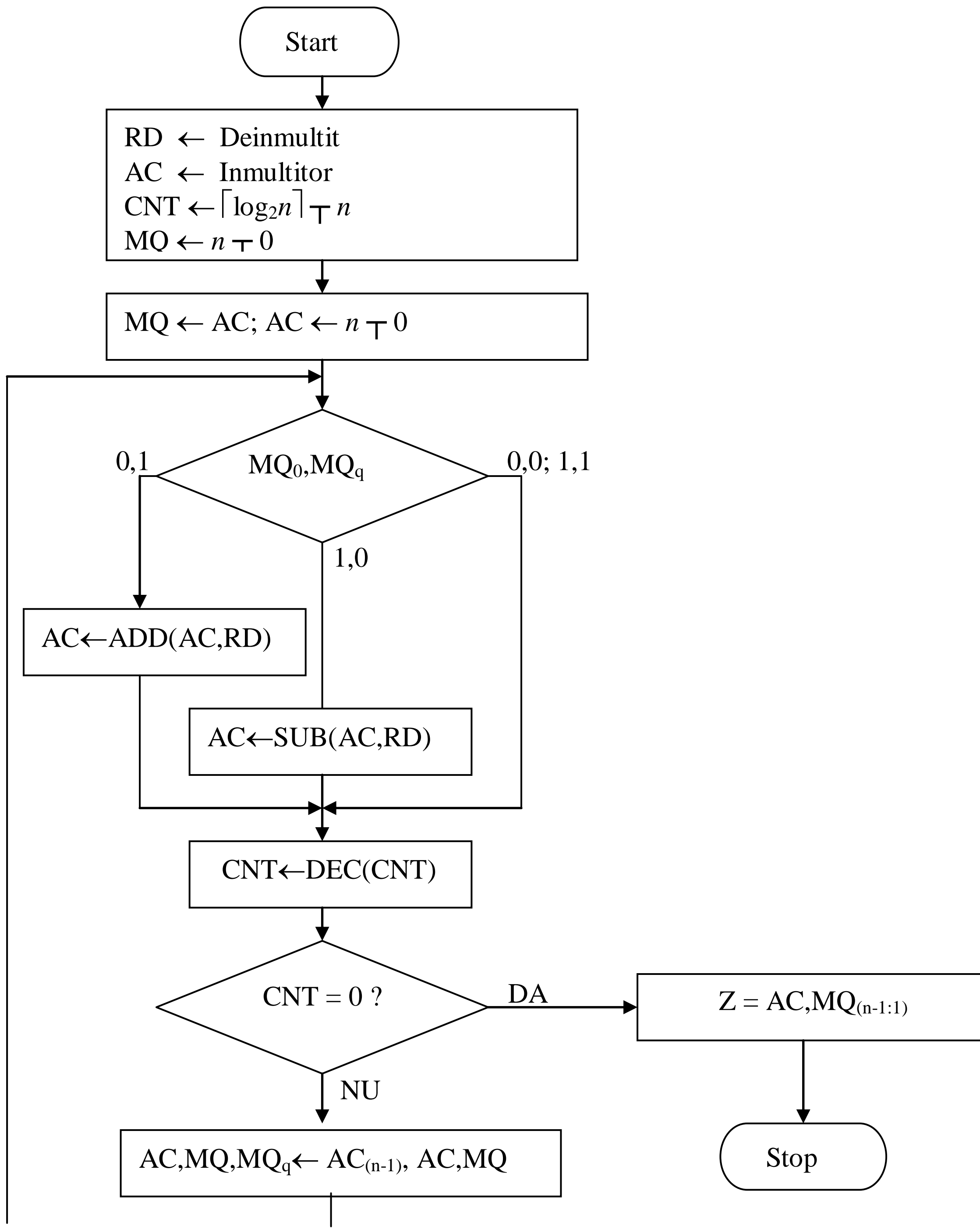
Pentru implementarea hardware se considera resursele corespunzatoare unei structuri orientate pe un singur acumulator:



Resursele hardware:

- AC – registru acumulator;
- RD – registru de date al memoriei;
- MQ – registru de extensie al acumulatorului;
- MQq – registru de un bit, extensia lui MQ;
- CNT – contor de cicluri ($\rho CNT = \lceil \log_2 n \rceil$);
- SL – structura logica pentru activarea si selectarea intrarilor multiplexorului, cat si pentru controlul transportului la sumator;
- Sumator- sumator combinational cu *n* ranguri binare.

Descrierea operarii dispozitivului se poate face cu ajutorul urmatoarei organigrame:


Start
RD ← Deinmultit
AC ← Inmultitor
CNT ← ⌈log2n⌉ ┬ n
MQ ← n ┬ 0
MQ ← AC; AC ← n ┬ 0
0,1
MQ0,MQq
0,0; 1,1
1,0
AC←ADD(AC,RD)
AC←SUB(AC,RD)
CNT←DEC(CNT)
CNT = 0 ?
DA
Z = AC,MQ(n-1:1)
NU
AC,MQ,MQq← AC(n-1), AC,MQ
Stop

Programul AHPL:

MODULE: Dispozitiv_de_inmultire

MEMORY: RD[n]; AC[n]; MQ[n]; MQ_q; CNT[$\lceil \log_2 n \rceil$]

INPUTS: X[n]; Y[n]

OUTPUTS: Z[2n-1]

// se considera operanzii X si Y adusi in RD si AC

1. MQ $\leftarrow$ AC; $MQ_q \leftarrow 0$; CNT $\leftarrow \lceil \log_2 n \rceil \top n$
2. AC $\leftarrow n \top 0$
3. $\rightarrow (\overline{MQ_0}\,\overline{MQ_q}, \overline{MQ_0}\,MQ_q, MQ_0\,\overline{MQ_q}, MQ_0\,MQ_q)/(6, 4, 5, 6)$
4. AC$\leftarrow$ADD(AC,RD)

   $\rightarrow$ (6)

5 AC$\leftarrow$SUB(AC,RD)

6 CNT$\leftarrow$DEC(CNT)

7 $\rightarrow (\overline{\cup/CNT}, \cup/CNT)/(9,8)$

8 AC,MQ,$MQ_q \leftarrow AC_{(n-1)}$, AC,MQ

   $\rightarrow$ (3)

9 ENDSEQ

Z = AC,$MQ_{(n-1:1)}$

END

In continuare se va prezenta un program Verilog pentru simularea unui dispozitiv de inmultire.

```
//Simularea unui dispozitiv de inmultire a numerelor reprezentate in complementul fata de doi,
//folosind Algoritmul lui Booth
module inmultitorb;
parameter n=8;
reg [n-1:0] RD,AC,MQ;
reg [2*n-1:0] Z;
reg [2:0] CNT;
reg MQq;
initial begin: init
AC=-7;RD=-5;MQ=0;MQq=0;CNT=n-1;
$display("timp  RD     AC       MQ    MQq CNT");
```

```
$monitor("%0d %b %b %b %b %b",$time,RD,AC,MQ,MQq,CNT);
wait (CNT==0)
        begin
        $monitor("Produs= %b",{AC[n-1],AC,MQ[n-1:1]});
        #1 $stop;
        end
end
always
begin
#1; MQ=AC;
#1; AC=0;
while(CNT>0)
  begin
   case({MQ[0],MQq})
2'b00:begin
       #1;{AC,MQ,MQq}={AC[n-1],AC,MQ};
      end
2'b01:
      begin
       #1;AC=AC+RD;
       #1;{AC,MQ,MQq}={AC[n-1],AC,MQ};
      end
2'b10:
      begin
       #1;AC=AC-RD;
       #1;{AC,MQ,MQq}={AC[n-1],AC,MQ};
      end
2'b11:
      begin
       #1;{AC,MQ,MQq}={AC[n-1],AC,MQ};
      end
  endcase
```

```
  #1;CNT=CNT-1;
end
end
endmodule
```

Veriwell -k C:\Program Files\VeriWell\exe\VeriWell.key -l C:\Program Files\VeriWell\exe\VeriWell.log inmultitorb.V
VeriWell for Win32 HDL <Version 2.1.1> Tue Dec 19 16:26:21 2000

Memory Available: 0
Entering Phase I...
Compiling source file : inmultitorb.V
The size of this model is [3%, 2%] of the capacity of the free version

Entering Phase II...
Entering Phase III...
No errors in compilation
Top-level modules:
   inmultitorb

//Cazul: RD = 5; AC = 7;

```
timp  RD     AC      MQ    MQq CNT
0 000101 00000111 00000000 0 111
1 00000101 00000111 00000111 0 111
2 00000101 00000000 00000111 0 111
3 00000101 11111011 00000111 0 111
4 00000101 11111101 10000011 1 111
5 00000101 11111101 10000011 1 110
6 00000101 11111110 11000001 1 110
7 00000101 11111110 11000001 1 101
8 00000101 11111111 01100000 1 101
9 00000101 11111111 01100000 1 100
10 00000101 00000100 01100000 1 100
11 00000101 00000010 00110000 0 100
12 00000101 00000010 00110000 0 011
13 00000101 00000001 00011000 0 011
14 00000101 00000001 00011000 0 010
15 00000101 00000000 10001100 0 010
16 00000101 00000000 10001100 0 001
17 00000101 00000000 01000110 0 001
Produs= 0000000000100011
Stop at simulation time 19
C1> $finish;
```

Top-level modules:

inmultitorb
// Cazul: RD = 5; AC = -7;

timp RD AC MQ MQq CNT
0 00000101 11111001 00000000 0 111
1 00000101 11111001 11111001 0 111
2 00000101 00000000 11111001 0 111
3 00000101 11111011 11111001 0 111
4 00000101 11111101 11111100 1 111
5 00000101 11111101 11111100 1 110
6 00000101 00000010 11111100 1 110
7 00000101 00000001 01111110 0 110
8 00000101 00000001 01111110 0 101
9 00000101 00000000 10111111 0 101
10 00000101 00000000 10111111 0 100
11 00000101 11111011 10111111 0 100
12 00000101 11111101 11011111 1 100
13 00000101 11111101 11011111 1 011
14 00000101 11111110 11101111 1 011
15 00000101 11111110 11101111 1 010
16 00000101 11111111 01110111 1 010
17 00000101 11111111 01110111 1 001
18 00000101 11111111 10111011 1 001
Produs= 1111111111011101
Stop at simulation time 20
C1> $finish;

// Cazul: RD = -5; AC = -7;

timp RD AC MQ MQq CNT
0 11111011 11111001 00000000 0 111
1 11111011 11111001 11111001 0 111
2 11111011 00000000 11111001 0 111
3 11111011 00000101 11111001 0 111
4 11111011 00000010 11111100 1 111
5 11111011 00000010 11111100 1 110
6 11111011 11111101 11111100 1 110
7 11111011 11111110 11111110 0 110
8 11111011 11111110 11111110 0 101
9 11111011 11111111 01111111 0 101
10 11111011 11111111 01111111 0 100
11 11111011 00000100 01111111 0 100
12 11111011 00000010 00111111 1 100
13 11111011 00000010 00111111 1 011
14 11111011 00000001 00011111 1 011
15 11111011 00000001 00011111 1 010
16 11111011 00000000 10001111 1 010

17 11111011 00000000 10001111 1 001
18 11111011 00000000 01000111 1 001
Produs= 0000000000100011
Stop at simulation time 20

2.3.Impartirea.

Ca regula generala, impartirea numerelor se realizeaza prin scaderea repetata a impartitorului, deplasat spre dreapta cu un rang, din restul de la scaderea precedenta, pana la obtinerea unui rest egal cu zero sau mai mic decat impartitorul; ca prim rest se considera deimpartitul.

In general se practica doua metode:

- metoda bazata pe refacerea ultimului rest pozitiv (metoda cu restaurare) si
- metoda in care nu se reface ultimul rest pozitiv (metoda fara restaurare).

Pentru prima metoda se prezinta exemplul urmator, in care deimpartitul este 22, iar impartitorul este 9:

$22 : 9 = q_0, q_{-1}q_{-2}$

22	40	40
- 9	- 9	- 9
13	31	31
- 9	- 9	- 9
4	22	22
- 9	- 9	- 9
- 5	13	13
$q_0 = 2$	- 9	- 9
	4	4
	- 9	- 9
	- 5	- 5
	$q_{-1} = 4$	$q_{-2} = 4$

22 : 9 = 2,44…

Cea de-a doua metoda pleaca de la ideea ca, in loc de a scadea impartitorul deplasat la dreapta cu un rang, din ultimul rest pozitiv, se va aduna impartitorul deplasat la dreapt, la la ultimul rest negativ. Metoda se va ilustra prin exemplul de mai jos:

$22 : 9 = q_0, q_{-1}q_{-2}$

22	-50	40
- 9	+ 9	- 9
13	- 41	31
- 9	+ 9	- 9
4	- 32	22
- 9	+ 9	- 9
- 5	- 23	13
$q_0 = 3$	+ 9	- 9
	- 14	4
	+ 9	- 9
	- 5	- 5
	+ 9	
	4	
	$q_{-1} = \bar{6}$	$q_{-2} =$

Catul va avea forma: 3, $\bar{6}$ 5 $\bar{6}$ 5 $\overline{6}$...... , in care termenii cu bara sunt negativi [3, (-6)5(-6)5(-6)], ceea ce va impune efectuarea unei corectii care va aduce rezultatul la forma:

22 : 9 = 2,44444…

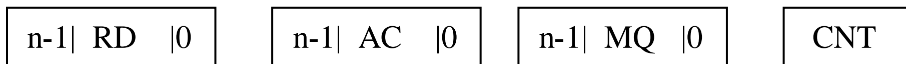
Ultima metoda necesita un numar mai mare de pasi, astfel incat, in continuare, se va examina implementarea metodei bazata pe restaurarea ultimului rest pozitiv.

Algoritmul impartirii numerelor reprezentate in complementul fata de doi, cu restaurarea ultimului rest pozitiv.

Pentru ilustrarea algoritmului se vor considera urmatoarele resurse hardware:

- AC – acumulator;
- RD – registrul de date al memoriei;
- MQ – registrul de extensie a acumulatorului;
- CNT – registru incrementator, contor de cicluri.

Impartitorul Y va fi incarcat in RD, iar deimpartitul X in AC. In MQ se va acumula catul.



Descrierea algoritmului:

1. Se deplaseaza continutul lui RD cu *p* ranguri spre stanga, in conditiile in care 2^p este prima incercare de multiplu binar al impartitorului.

   Se verifica daca RD = 0, in caz afirmativ operatia se termina cu semnalizarea unei erori.

   Daca RD ≠ 0, se incarca CNT cu vectorul binar avand valoarea p .
2. Daca deimpartitul si impartitorul au semne identice/diferite se scade/se aduna impartitorul din/la deimpartit, pentru a obtine restul.
3. a) Daca restul si deimpartitul au semne identice sau restul este egal cu zero, se introduce o unitate in bitul cel mai putin semnificativ a lui MQ, iar restul va lua locul deimpartitului.

   b) Daca restul si deimpartitul au semne diferite si restul curent nu este zero, se introduce un zero in bitul cel mai putin semnificativ al lui MQ, fara a mai modifica deimpartitul.
4. Se deplaseaza continutul registrului imparitorului cu un rang spre dreapta, extinzand rangul de semn.

   Se decrementeaza CNT.

   Se deplaseaza MQ spre stanga cu un bit.
5. Se repeta pasii 2- 4 pana cand continutul lui CNT devine egal cu zero.
6. Daca deimpartitul si impartitorul au avut semne identice rezultatul se afla in registrul MQ. In cazul in care semnele celor doi operanzi au fost diferite, rezultatul se obtine prin complementarea continutului registrului MQ.

Realizarea practica a algoritmului impune introducerea unor resurse hardware suplimentare, fata de AC, RD, MQ, CNT si anume:

- R[*n*] – registrul in care se obtine restul curent;
- z[1] – bistabil in care se stocheaza informatia (1/0) referitoare la semnele identice/diferite ale celor doi operanzi;
- e[1] – bistabil care semnalizeaza conditia de eroare/non-eroare (1/0);
- S – unitate logica combinationala, care genereaza semnalul de sfarsit al operatiilor de deplasare spre stanga in registrul RD;

  $S = RD_{n-1} . RD_{n-2} . \overline{RD_{n-3}} \cup \overline{RD_{n-1}} . RD_{n-2}$

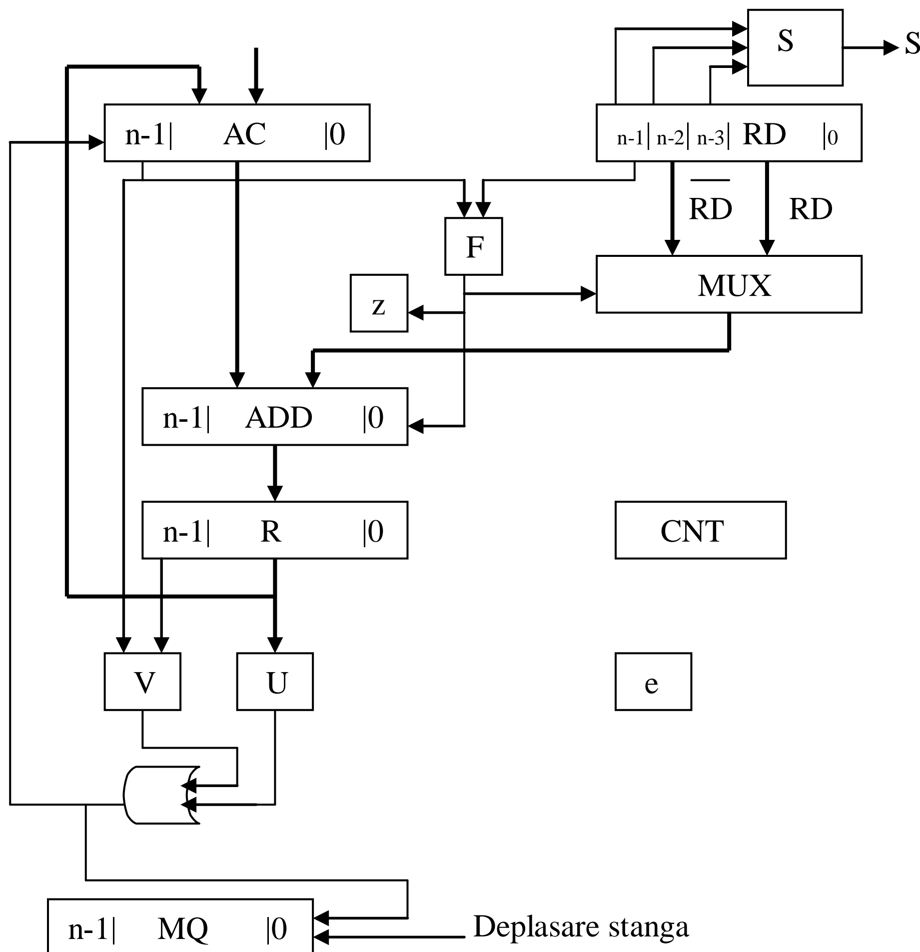
- F- unitate logica combinationala, care calculeaza identitatea/neidentitatea semnelor operanzilor;

  $F = \overline{AC_{n-1} \oplus RD_{n-1}}$
- V – unitate logica combinationala, care verifica semnele operanzilor din acumulator (deimpartit) si din registrul restului curent R;

  $V = \overline{AC_{n-1} \oplus R_{n-1}}$
- U – unitate logica combinationala, care verifica existenta unui rest curent egal cu zero;

  $U = \overline{\cup / R}$.

Structura generala, la nivel de schema bloc, a dispozitivului de impartire:



Programul AHPL pentru dispozitivul de impartire.

MODULE: Dispozitiv_de_impartire

MEMORY: RD[n]; AC[n]; MQ[n]; R[n]; CNT[$\lceil \log_2 n \rceil$]; z[1]; e[1]

INPUTS: X[n]; Y[n]

OUTPUTS: Z[n]

// se considera operanzii X si Y adusi in AC si RD.

1. $MQ \leftarrow n \top 0$; $CNT \leftarrow \lceil \log_2 n \rceil \top 0$; $Z \leftarrow 0$; $e \leftarrow 0$

   $\rightarrow (\overline{\cup/RD})/(12)$ // eroare

2. $\rightarrow (\overline{S})/(4)$

3. $RD \leftarrow RD_{n-2:0}, 0$; $CNT \leftarrow INC(CNT)$

   $\rightarrow (2)$

4. $z \leftarrow F$

5. $R \leftarrow (ADD(AC,RD) \; ! \; SUB(AC,RD)) * (\overline{F}, F)$

6. $\rightarrow (V \cup U)/(8)$

7. $MQ \leftarrow MQ_{n-2:0}, 0$

   $\rightarrow (9)$

8. $MQ \leftarrow MQ_{n-2:0}, 1$

9. $\rightarrow (\overline{\cup/CNT})/(11)$

10. $RD \leftarrow RD_{n-1}, RD_{n-1:1}$; $CNT \leftarrow DCR(CNT)$

    $\rightarrow (5)$

11. $MQ \leftarrow (MQ \; ! \; ADD(\overline{MQ},1)) * (z, \overline{z})$

12. $e \leftarrow (0 \; ! \; 1) * (\cup/RD, \overline{\cup/RD})$

ENDSEQ

$Z = MQ$

END

```
module Disp_Impartire;
// Deimpartit: AC (initializat); Impartitor: RD(initializat); Cat: MQ; Contor cicluri: CNT;
//Cei doi operanzi sunt adusi initial la valori pozitive.
// Semnul catului este stocat in registrul S
//Catul se obtine in MQ
parameter n =8;
reg[n-1:0] AC,RD,CNT,Z1,R,MQ;
reg S,Z;
initial begin:init
AC=127;RD=-30;CNT=0;S=0;Z=0;Z1=0;R=0;MQ=0;
$monitor("AC=%d RD = %d CNT = %d S=%b Z=%b
Z1=%d,R=%d,MQ=%d",AC,RD,CNT,S,Z,Z1,R,MQ);
end
always
begin
#1 S=(AC[n-1] ^ RD[n-1]);
#1 AC =(AC[n-1]==1)?-AC:AC;
#1 RD = (RD[n-1]==1)?-RD:RD;
#1 Z1=(AC-RD);
#1 Z=(Z1[n-1] == 1)|(RD==0);
 if(Z==1)                          //20
   begin
    #1 $display("Eroare: fie  RD nul , fie |RD| >|AC|");
    #1 $stop;
   end
  else
    begin
    #1 while (Z1[n-1]==0)
       begin
        #1  RD=RD<<1;
        #1  CNT=CNT+1;                   //30
       #1  Z1= AC-RD;
       end
        #1 RD=RD>>1;
        #1 CNT=CNT-1;
 #1 while ((CNT[n-1]==0))
  begin
   begin
 #1 if((AC[n-1]^RD[n-1]) ==0)
    begin
    #1  R=AC-RD;                          //40
    end
  else
    begin
    #1 R=AC+RD;
    end
   end
   begin
 #1 if( (!(AC[n-1]^R[n-1])||(R==0))== 1)
  begin
   #1 CNT=CNT-1;                          //50
   #1 MQ=MQ<<1;
   #1 AC=R;
   #1 RD={RD[n-1],RD[n-1:1]};
```

```
   #1 MQ=MQ^1;
   end
  else
   begin
  #1 CNT=CNT-1;
  #1 MQ=MQ<<1;
  #1 RD={RD[n-1],RD[n-1:1]};                //60
  end
 end
 end
 begin
#1 if(S==0)
  begin
   #1 MQ=MQ;
  end
  else
   begin                                     //70
  #1MQ=-MQ;
   end
 #1 $stop;
     end
     end
     end
endmodule
```

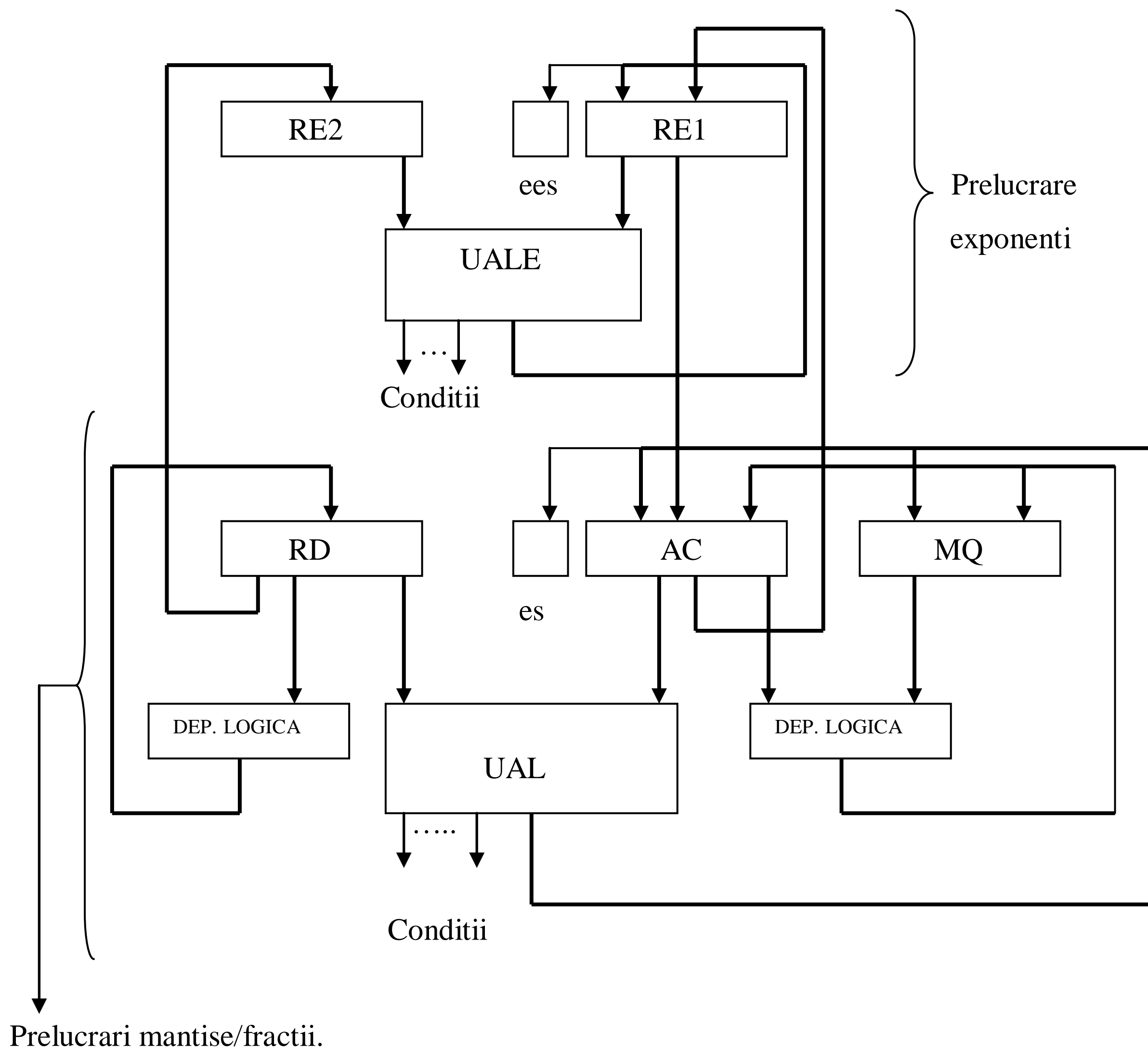
Operatii aritmetice in virgula mobila.

Operatiile aritmetice in virgula mobila vor fi examinate la nivelurile schemei bloc, pentru unitatea aritmetica, si al organigramelor pentru adunare/scadere si inmultire. In analiza care urmeaza se considera operanzii de intrare X ,Y si rezultatul Z, care vor avea urmatoarele formate:

X $\leftarrow$ xs, XE, XF ; Y $\leftarrow$ ys, YE, YF; Z $\leftarrow$ zs, ZE, ZF

3.1. Schema bloc a unitatii aritmetice in virgula mobila.

Schema bloc a unitatii aritmetice in virgula mobila, in cazul de fata, se bazeaza pe structura schemei unitatii aritmetice in virgula fixa, la care s-au mai adaugat o serie de resurse, pentru manipularea exponentilor (registre si unitate aritmetica).



Prelucrari mantise/fractii.

Partea care prelucreaza exponentii contine urmatoarele resurse:

- RE1 si RE2 - registre pentru exponenti;
- ees- bistabil de extensie a registrului exponentului la stanga;
- UALE- Unitate Aritmetica Logica pentru Exponenti.

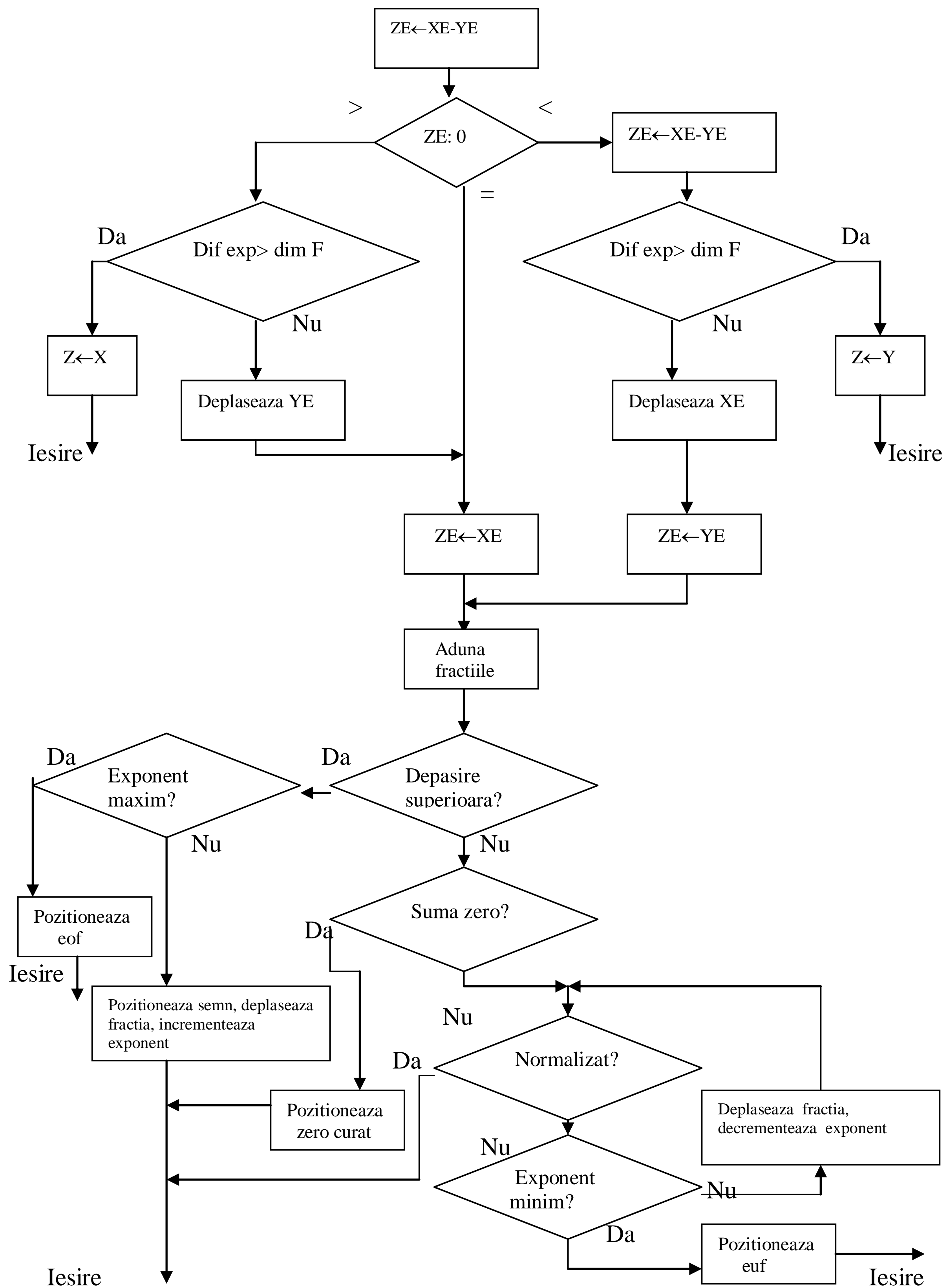
In partea care prelucreaza mantisele/fractiile, fata de resursele hardware pentru prelucrarea in virgula fixa au mai aparut doua circuite de deplasare logica si un bistabil de extensie a acumulatorului la stanga *es*.

Cele doua unitati aritmetice logice sunt prevazute cu circuite logice pentru generarea indicatorilor de conditii si de eroare.

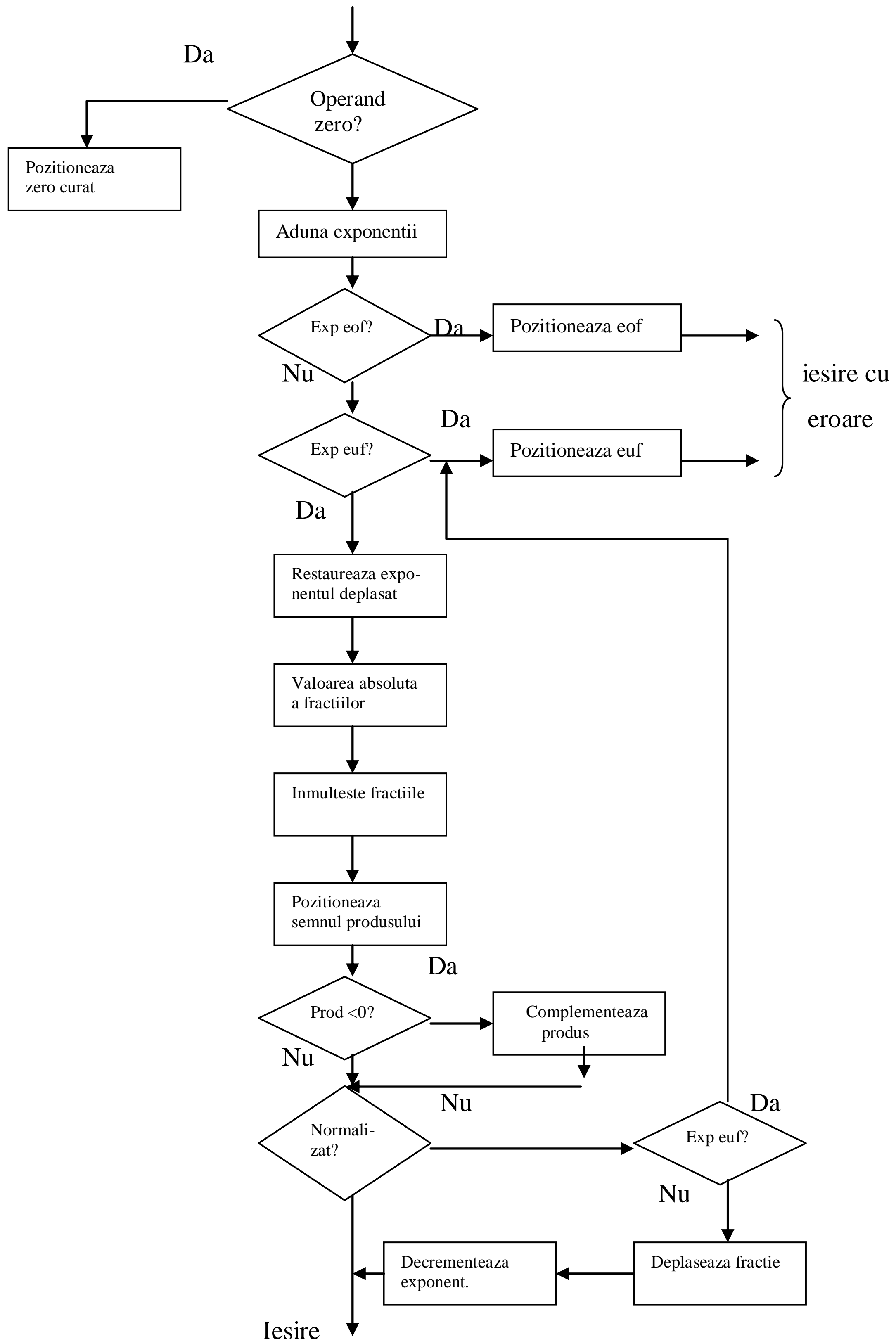
Operanzii de prelucrat se afla initial in registrele AC si RD. Din acestea se extrag exponentii (operatia de despachetare), care sunt incarcati in RE1 si RE2. Fractiile sunt deplasate spre stanga in AC si RD, pentru a beneficia de o precizie maxima.

Dupa terminarea operatiilor asupra exponentilor si fractiilor, are loc o insertie (operatia de impachetare) a exponentului rezultatului in registrul AC, prin deplasarea fractiei din AC spre dreapta.

3.2 Organigrama adunarii/scaderii in virgula mobila.



3.3. Organigrama inmultirii in virgula mobila.

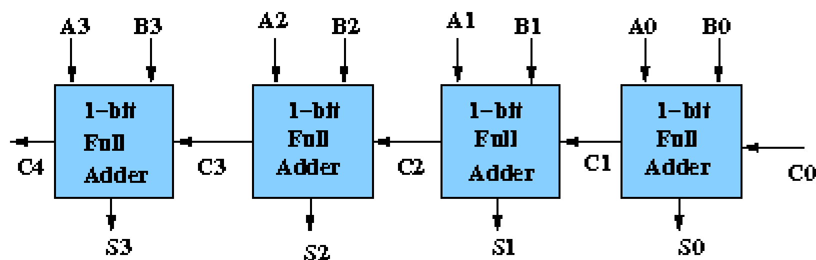


ANEXA 1.

Design of Ripple Carry Adders :

Arithmetic operations like addition, subtraction, multiplication, division are basic operations to be implemented in digital computers using basic gates likr AND, OR, NOR, NAND etc. Among all the arithmetic operations if we can implement addition then it is easy to perform multiplication (by repeated addition), subtraction (by negating one operand) or division (repeated subtraction).

Half Adders can be used to add two one bit binary numbers. It is also possible to create a logical circuit using multiple full adders to add N-bit binary numbers.Each full adder inputs a **Cin**, which is the **Cout** of the previous adder. This kind of adder is a **Ripple Carry Adder**, since each carry bit "ripples" to the next full adder. The first (and only the first) full adder may be replaced by a half adder.The block diagram of 4-bit Ripple Carry Adder is shown here below -



The layout of ripple carry adder is simple, which allows for fast design time; however, the ripple carry adder is relatively slow, since each full adder must wait for the carry bit to be calculated from the previous full adder. The gate delay can easily be calculated by inspection of the full adder circuit. Each full adder requires three levels of logic.In a 32-bit [ripple carry] adder, there are 32 full adders, so the critical path (worst case) delay is 31 * 2(for carry propagation) + 3(for sum) = 65 gate delays.

Design Issues :

The corresponding boolean expressions are given here to construct a ripple carry adder. In the half adder circuit the sum and carry bits are defined as

sum = A ⊕ B

carry = AB

In the full adder circuit the the Sum and Carry outpur is defined by inputs A, B and Carryin as

Sum=ABC + ABC + ABC + ABC

Carry=ABC + ABC + ABC + ABC

Having these we could design the circuit.But,we first check to see if there are any logically equivalent statements that would lead to a more structured equivalent circuit.

With a little algebraic manipulation,one can see that

Sum= $ABC + ABC + ABC + ABC$

$= (AB + AB)\,C + (AB + AB)\,C$

$= (A \oplus B)\,C + (A \oplus B)\,C$

$= A \oplus B \oplus C$

Carry= $ABC + ABC + ABC + ABC$

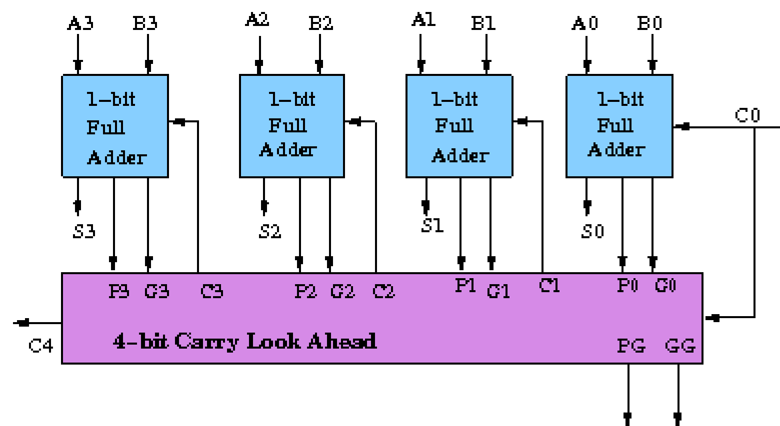
$= AB + (AB + AB)\,C$

$= AB + (A \oplus B)\,C$

ANEXA 2.

Design of Carry Lookahead Adders :

To reduce the computation time, there are faster ways to add two binary numbers by using carry lookahead adders. They work by creating two signals P and G known to be **Carry Propagator** and **Carry Generator**. The carry propagator is propagated to the next level whereas the carry generator is used to generate the output carry ,regardless of input carry. The block diagram of a 4-bit Carry Lookahead Adder is shown here below -



The number of gate levels for the carry propagation can be found from the circuit of full adder. The signal from input carry C_{in} to output carry C_{out} requires an AND gate and an OR gate, which constitutes two gate levels. So if there are four full adders in the parallel adder, the output carry C_5 would have 2 X 4 = 8 gate levels from C_1 to C_5. For an n-bit parallel adder, there are 2n gate levels to propagate through.

Design Issues :

The corresponding boolean expressions are given here to construct a carry lookahead adder. In the carry-lookahead circuit we ned to generate the two signals carry propagator(P) and carry generator(G),

$\mathbf{P_i = A_i \oplus B_i}$

$\mathbf{G_i = A_i \cdot B_i}$

The output sum and carry can be expressed as

$\mathbf{Sum_i = P_i \oplus C_i}$

$\mathbf{C_{i+1} = G_i + (P_i \cdot C_i)}$

Having these we could design the circuit. We can now write the Boolean function for the carry output of each stage and substitute for each Ci its value from the previous equations:

$\mathbf{C_1 = G_0 + P_0 \cdot C_0}$

$\mathbf{C_2 = G_1 + P_1 \cdot C_1 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0}$

$\mathbf{C_3 = G_2 + P_2 \cdot C_2 = G_2\ P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0}$

$\mathbf{C_4 = G_3 + P_3 \cdot C_3 = G_3\ P_3 \cdot G_2\ P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0}$

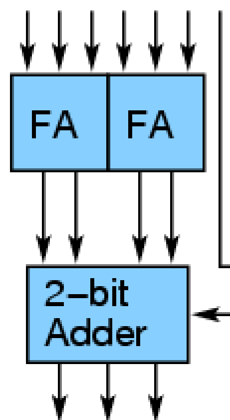
ANEXA 3.

Design of Wallace Tree Adders :

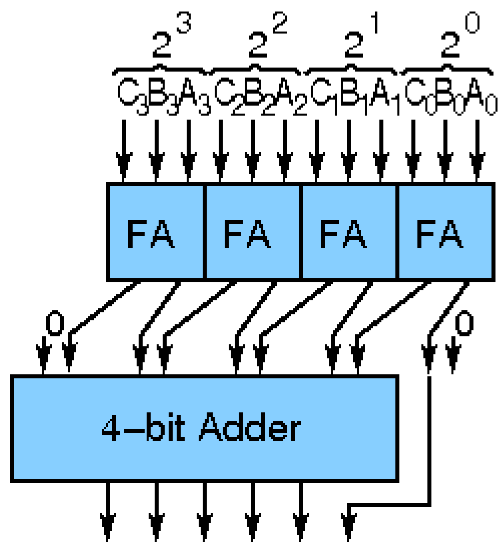
There are many cases where it is desired to add more than two numbers together. The straightforward way of adding together m numbers (all n bits wide) is to add the first two, then add that sum to the next using cascading full adders. This requires a total of m – 1 additions, for a total gate delay of O(m lg n) (assuming lookahead carry adders). Instead, a tree of adders can be formed, taking only O(lg m · lg n) gate delays.

A Wallace tree adder adds together n bits to produce a sum of $\log_2 n$ bits.

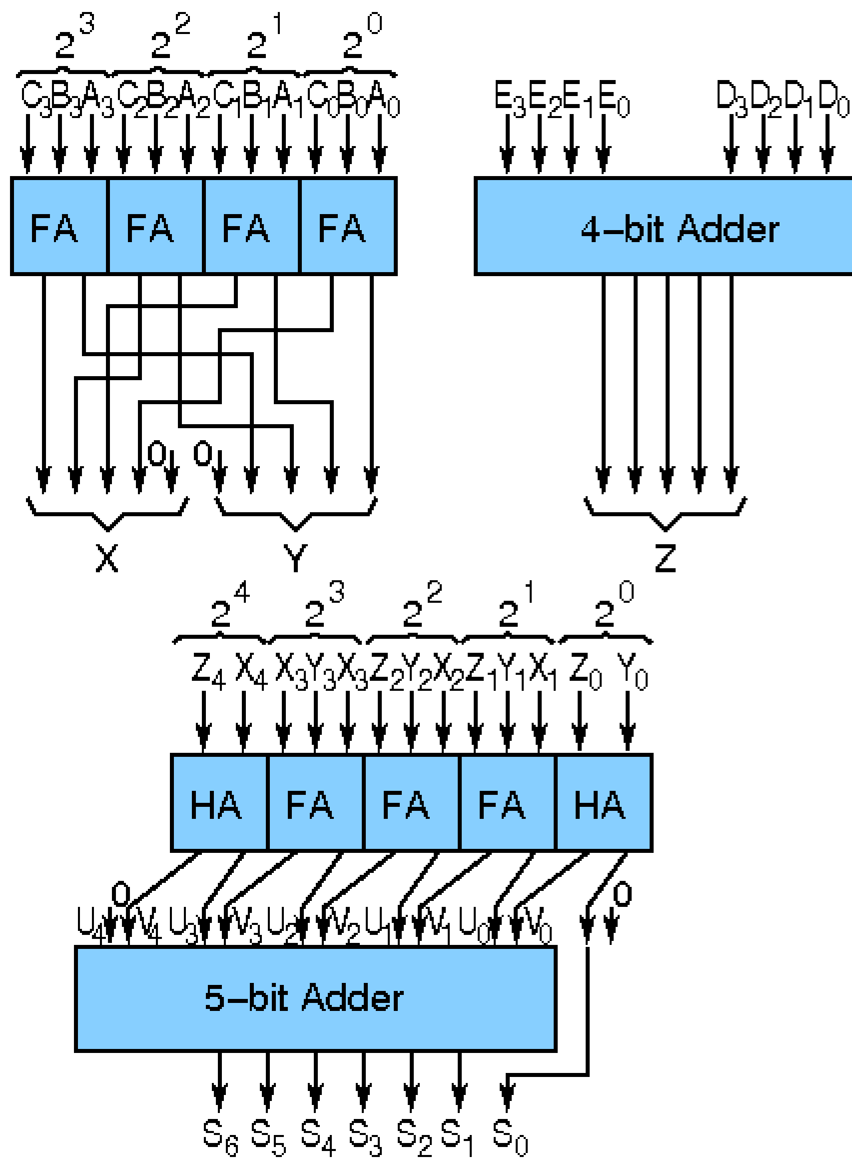
The design of a Wallace tree adder to add seven bits (W_7) is illustrated below:



An adder tree to add three 4-bit numbers is shown below:



An adder tree (interconnections incomplete) to add five 4-bit numbers is shown below:



ANEXA 4.

Design of Combinational Multiplier :

Combinational Multipliers do multiplication of two unsigned binary numbers.Each bit of the multiplier is multiplied against the multiplicand, the product is aligned according to the position of the bit within the multiplier, and the resulting products are then summed to form the final result. Main advantage of binary multiplication is that the generation of intermediate products are simple: if the multiplier bit is a 1, the product is an appropriately shifted copy of the multiplicand; if the multiplier bit is a 0, the product is simply 0.

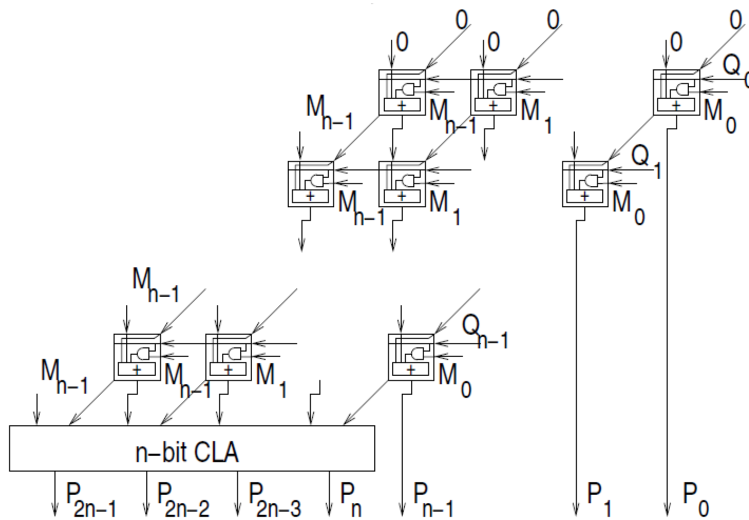
The design of a combinational multiplier to multiply two 4-bit binary number is illustrated below:

			A3	A2	A1	A0
			B3	B2	B1	B0
			A3 •B0	A2 •B0	A1•B0	A0•B0
		A3 •B1	A2•B1	A1•B1	A0•B1	
	A3 •B2	A2•B2	A1•B2	A0•B2		
A3 •B3	A2 •B3	A1•B3	A0•B3			
S_6	S_5	S_4	S_3	S_2	S_1	S_0

If two n-bit numbers are multiplied then the output will be less than or equals to 2n bits.

Some features of the multiplication scheme:

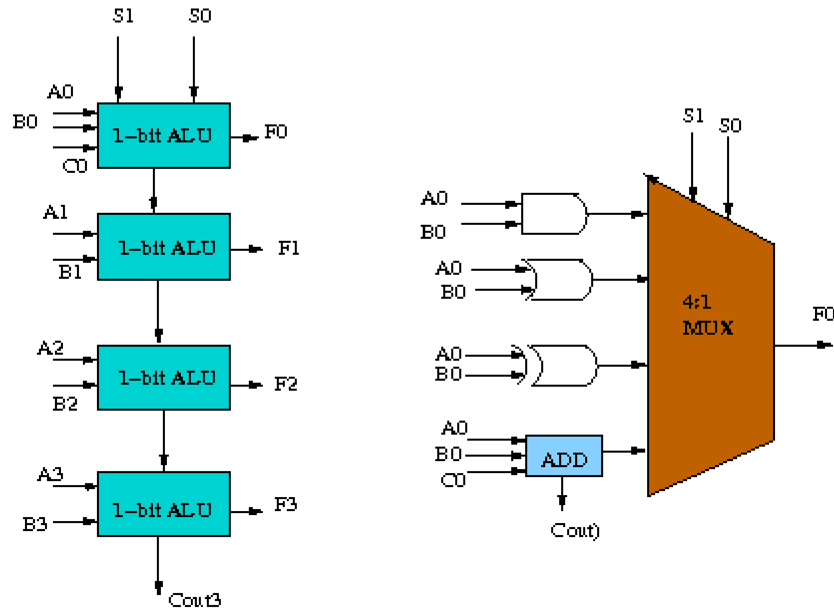
- it can be designed by unrolling the multiplier loop
- instead of handling the carry out of partial product summation bit,the carry out can be sent to the next bit of the next step
- this scheme of handling the carry is called carry save addition
- this scheme is more regular and modular
- Logic diagram:



ANEXA 5

Design of ALU :

ALU or Arithmetic Logical Unit is a digital circuit to do arithmetic operations like addition, subtraction,division, multiplication and logical oparations like and, or, xor, nand, nor etc. A simple block diagram of a 4 bit ALU for operations and,or,xor and Add is shown here :



The 4-bit ALU block is combined using 4 1-bit ALU block

Design Issues :

The circuit functionality of a 1 bit ALU is shown here, depending upon the control signal S_1 and S_0 the circuit operates as follows:

for Control signal $\mathbf{S_1 = 0}$, S_0 = 0, the output is **A And B**,

for Control signal $\mathbf{S_1 = 0}$, S_0 = 1, the output is **A Or B**,

for Control signal $\mathbf{S_1 = 1}$, S_0 = 0, the output is **A Xor B**,

for Control signal $\mathbf{S_1 = 1}$, S_0 = 1, the output is **A Add B**.

The truth table for 16-bit ALU with capabilities similar to 74181 is shown here:

Required functionality of ALU (inputs and outputs are active high)

Mode Select Inputs				**F_n for active HIGH operands**	
				Logic	**Arithmetic (note 2)**
S3	S2	S1	S0	(M = H)	(M = L) **(C_n=L)**
L	L	L	L	A'	A
L	L	L	H	A'+B'	A+B
L	L	H	L	A'B	A+B'
L	L	H	H	Logic 0	minus 1
L	H	L	L	(AB)'	A plus AB'
L	H	L	H	B'	(A + B) plus AB'
L	H	H	L	A ⊕ B	A minus B minus 1
L	H	H	H	AB'	AB minus 1
H	L	L	L	A'+B	A plus AB
H	L	L	H	(A ⊕ B)'	A plus B
H	L	H	L	B	(A + B') plus AB
H	L	H	H	AB	AB minus 1
H	H	L	L	Logic 1	A plus A (Note 1)
H	H	L	H	A+B'	(A + B) plus A
H	H	H	L	A+B	(A + B') plus A
H	H	H	H	A	A minus 1