Proiectarea unei unități aritmetico-logice în virgulă flotantă

Student: Jarda Adina-Ionela

Structura Sistemelor de Calcul Proiect

Universitatea Tehnica din Cluj-Napoca

2024

Cuprins

Introducere

1.1 Context

Obiectivul acestui proiect este de a proiecta o unitate aritmetico-logică (ALU) în virgulă flotantă, capabilă să efectueze operații de adunare și înmulțire. Proiectul include atât realizarea arhitecturii, cât și simularea funcționalității unității ALU utilizând limbajul de descriere hardware VHDL.

Scopul este de a dezvolta o unitate performantă și precisă pentru operațiile în virgulă flotantă, aplicabilă în sisteme de calcul care necesită un nivel ridicat de precizie și optimizare a procesării datelor numerice complexe.

1.2 Obiective

Scopul principal al acestui proiect este proiectarea și implementarea unei unități aritmetico-logice (ALU) în virgulă flotantă, capabilă să efectueze operații de adunare și înmulțire asupra numerelor reprezentate conform standardului IEEE 754. Această unitate va fi implementată și simulată utilizând Xilinx, iar rezultatele vor fi testate pe un dispozitiv hardware FPGA.

Obiectivele specifice ale proiectului includ:

- Studiul reprezentării numerelor în virgulă flotantă conform standardului IEEE 754 și înțelegerea algoritmilor de bază pentru adunare și înmulțire.
- Proiectarea arhitecturii hardware pentru operațiile de adunare și înmulțire în virgulă flotantă, utilizând descriere hardware (VHDL).
- Implementarea pe un FPGA și testarea unității aritmetico-logice pe hardware real, evaluând performanțele acesteia în termeni de consum de resurse și viteză de execuție.
- Optimizarea arhitecturii pentru a asigura un echilibru între precizie, eficiența utilizării resurselor FPGA și performanța operațiilor.

Aceste obiective vor asigura integrarea unei unități ALU fiabile și precise, esențiale în sistemele de calcul ce necesită performanță și acuratețe în calcule numerice complexe.

Studiu Bibliografic

2.1 Ce este ALU?

Unitatea Aritmetico-Logică (ALU - Arithmetic Logic Unit) reprezintă una dintre componentele esențiale ale unui procesor, responsabilă de efectuarea operațiilor de bază, aritmetice și logice, asupra datelor. Printre aceste operații se numără adunarea, scăderea, multiplicarea și împărțirea, dar și operațiuni logice precum AND, OR, XOR și negarea.

Într-o arhitectură de calcul tradițională, ALU este coordonată de unitatea de control, care determină ce operații trebuie realizate și furnizează datele necesare. Datele procesate de ALU provin, de regulă, din registrele procesorului, iar rezultatele sunt salvate fie în aceleași registre, fie în memoria sistemului.

Deși ALU-urile clasice sunt concepute, în principal, pentru operații cu numere întregi, computerele moderne includ și unități specializate, precum unitatea de calcul în virgulă flotantă (FPU - Floating Point Unit), dedicată manipulării numerelor reale (în virgulă flotantă). În arhitecturile ce utilizează standardul IEEE 754, FPU este frecvent considerată o extensie a ALU, având rolul de a procesa operațiile complexe cu precizie ridicată.

Evoluția ALU-urilor a influențat semnificativ performanța calculatoarelor, optimizarea acestora fiind crucială pentru toate aplicațiile care cer procesare rapidă, cum ar fi grafica pe calculator, simulările științifice și procesarea avansată a semnalelor.

2.2 Soluții pentru erorile de trunchiere și precizia calculelor

Erorile de trunchiere, cauzate de limitările reprezentării binare, pot afecta precizia calculelor. Tehnicile de rotunjire și unitățile de verificare a depășirilor de capacitate (overflow) și subcapacitate (underflow) minimizează acumularea acestor erori. Totodată, se asigură conformitatea cu standardul IEEE 754, îmbunătățind astfel precizia calculelor.

Standardul IEEE 754 definește și modalități de reprezentare pentru valori speciale:

- **Zero**: Valoarea 0 este reprezentată distinct în formă pozitivă (+0) și negativă (−0) prin convenția semnului. În IEEE 754, un 0 binar în zona de exponent și mantisă indică această valoare.
- **NaN** (Not a Number): Reprezintă o valoare nedefinită sau rezultatul unei operații imposibile (precum 0/0 sau rădăcina pătrată a unui număr negativ). IEEE 754 definește două tipuri de NaN — *quiet NaN* (qNaN), utilizat pentru valori nedefinite, și *signaling NaN* (sNaN), folosit pentru operațiuni invalid.
- **Infinitul** (∞): Reprezintă valori foarte mari care depășesc capacitatea de stocare a formatului în virgulă flotantă. Infinitul pozitiv (+∞) și infinitul negativ (−∞) sunt

specificate în IEEE 754 prin setarea exponenților la valoarea maximă și a mantisei la zero.

Aceste valori speciale sunt esențiale în testarea și evaluarea unităților aritmetice pentru a asigura funcționarea corectă conform standardului IEEE 754. În testele de performanță și acurateţe pentru o ALU sau FPU, este crucial să verificăm:

1. **Comportamentul în prezența lui 0**: Verificarea operațiilor ce implică 0 (+0 și -0), cum ar fi 0 + X și 0 * X, trebuie să returneze valori conforme cu semnul și valoarea.
2. **Detecția și propagarea valorilor NaN**: Asigurarea că operațiile care implică NaN returnează NaN, semnalând operații invalide unde este cazul.
3. **Depășiri și subcapacități în prezența** ∞: Testarea calculului cu infinit pentru a garanta rezultate corecte (ex. X + ∞ = ∞ sau 1/∞ = 0), precum și identificarea corectă a overflow-ului și underflow-ului.

Astfel de teste contribuie la îmbunătățirea conformității cu IEEE 754 și la reducerea erorilor de trunchiere, contribuind la acuratețea și stabilitatea calculului în aplicațiile sensibile.

2.3 Reprezentarea numerelor în virgula mobilă (VM)

Un numar ***N*** poate fi reprezentat in virgulă mobilă (VM) sub forma generală:

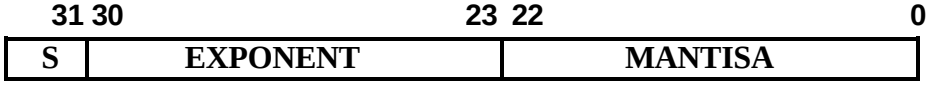
$$N = \pm M \cdot B^{\pm E}$$

unde avem două componente principale:

1.Mantisa (M) – Aceasta reflectă valoarea exactă a numărului în cadrul unui anumit interval și este, în general, exprimată ca un număr fracționar cu semn.

2. Exponentul (E) – Această componentă indică ordinul de mărime al numărului, definind scara la care se situează valoarea reprezentată.

În această formulă, ***B*** reprezintă baza exponentului, care determină scara numerică pe care o poate lua exponentul ***E*** și, implicit, influențează gama de valori ce pot fi reprezentate în virgulă mobilă.
Această reprezentare poate fi memorată într-un cuvânt binar cu trei câmpuri: semnul, mantisa şi exponentul. De exemplu, presupunând un cuvânt de 32 de biţi, o asignare posibilă a biţilor la fiecare câmp poate fi urmatoarea:



Aceasta este o reprezentare în mărime și semn, deoarece semnul are un câmp separat față de restul numărului. Câmpul de semn constă dintr-un bit care indică semnul numărului, 0 pentru un număr pozitiv și 1 pentru un număr negativ. Nu există un câmp rezervat pentru baza B, deoarece această bază este implicită și ea nu trebuie memorată, fiind aceeași pentru toate numerele.

De obicei, câmpul rezervat exponentului nu conține exponentul real, ci o valoare numită caracteristică, care se obține prin adunarea unui deplasament la exponent, astfel încât să rezulte întotdeauna o valoare pozitivă. Astfel, nu este necesar să se rezerve un câmp separat pentru semnul exponentului. Caracteristica C este deci exponentul deplasat:

$$C = E + deplasament\ (deplasament=127)$$

2.4 Reprezentarea numerelor în formatul IEEE 754

Standardul IEEE 754 definește următoarele formate sau precizii: precizie simplă, precizie simplă extinsă, precizie dublă şi precizie dublă extinsă. Parametrii principali ai acestor formate sunt prezentaţi în Tabelul 2.7. Standardul nu precizează ca obligatorie implementarea tuturor formatelor, dar recomandă implementarea combinației formatelor cu precizie simplă și precizie simplă extinsă, sau a formatelor cu precizie simplă, precizie dublă și precizie dublă extinsă.

	Precizie simplă	**Precizie simplă extinsă**	**Precizie dublă**	**Precizie dublă extinsă**
Biti ai mantisei	24	≥32	53	≥64
Exponent real maxim	127	≥1023	1023	≥16383
Exponent real minim	-126	≤1022	-1022	≤-16382
Deplasament exponent	127	Nespecificat	1023	Nespecificat

Tabelul 2.1 Parametrii formatelor definite de standardul IEEE 754.

Pentru toate formatele, baza implicită este 2. Formatele cu precizie simplă, precizie dublă şi precizie dublă extinsă sunt prezentate în Figura 2.2. Coprocesoarele matematice şi unităţile de calcul în virgulă mobilă ale procesoarelor implementează de obicei aceste formate.

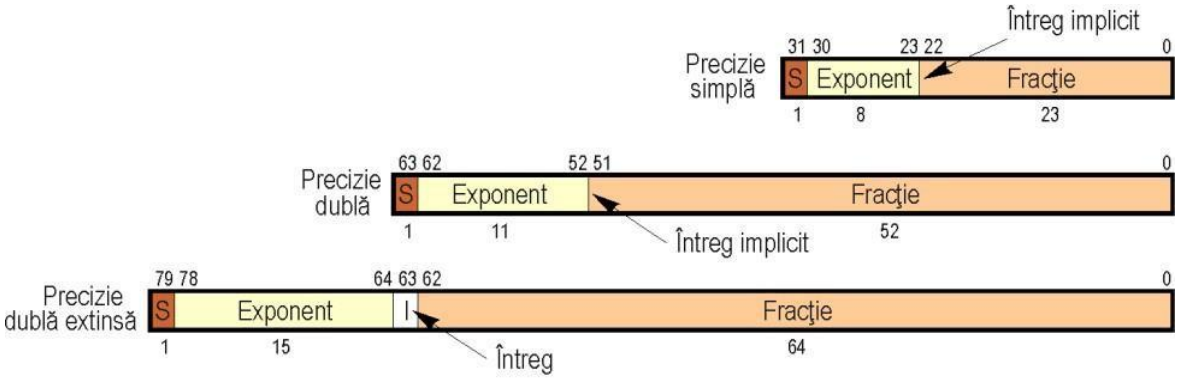


Figura 2.2. Formatele cu precizie simplă, precizie dublă şi precizie dublă extinsă definite de standardul IEEE 754.

2.5 Adunarea și Înmulțirea în virgulă mobilă

Adunarea în virgulă flotantă

Adunarea în virgulă flotantă este mai complexă decât adunarea simplă în aritmetica pe numere întregi, din cauza formatului specific al numerelor în IEEE 754. Operația necesită următorii pași principali:

Alinierea exponenților

Dacă exponenții celor două numere sunt diferiți, este necesară alinierea acestora. Exponentul mai mic este crescut pentru a fi egal cu cel mai mare, ajustând mantisa corespunzătoare. Această aliniere asigură că cele două numere au aceeași ordine de mărime.

Adunarea mantiselor

Odată ce exponenții sunt aliniați, mantisele celor două numere pot fi adunate. Dacă unul dintre numere are semn negativ, operația va deveni o scădere a mantiselor (similară cu scăderea descrisă mai sus). Adunarea se efectuează folosind algoritmi clasici de adunare binară.

Normalizarea rezultatului

După adunarea mantiselor, rezultatul poate necesita normalizare pentru a respecta formatul IEEE 754, prin ajustarea exponenței și a mantisei astfel încât mantisa să fie în intervalul permis (de obicei, între 1 și 2 pentru numere normalizate).

Rotunjire

După normalizare, rezultatul este rotunjit conform regulilor IEEE 754, pentru a asigura precizia. Acest lucru ajută la minimizarea erorilor de trunchiere, păstrând cât mai exact rezultatul final.

Înmulțirea în virgulă flotantă

Înmulțirea în virgulă flotantă este o operație care combină componentele celor două numere în mod diferit față de adunare. Pașii pentru înmulțirea numerelor în virgulă flotantă sunt următorii:

Adunarea exponenților

Exponenții celor două numere sunt adunați pentru a obține exponentul rezultatului. Dacă baza sistemului este 2 (cum este în majoritatea calculatoarelor), adunarea exponenților se face în binar, luând în considerare deplasarea exponenților (bias-ul).

Înmulțirea mantiselor

Mantisele celor două numere sunt înmulțite între ele. Întrucât mantisele sunt reprezentate în format binar fracționar, înmulțirea se efectuează folosind algoritmi clasici de înmulțire binară, ținând cont de orice cifre care depășesc intervalul standardizat.

Normalizarea rezultatului

După înmulțirea mantiselor, rezultatul poate necesita normalizare. Aceasta implică ajustarea mantisei și a exponentului pentru a păstra forma standard IEEE 754, astfel încât mantisa să se încadreze în intervalul dorit.

Rotunjire

Rezultatul final este rotunjit conform standardului IEEE 754, pentru a păstra precizia maximă în limitele reprezentării. Acest pas minimizează erorile de trunchiere și asigură conformitatea cu standardul de precizie dorit.

2.6 Sumatorul Pe Un Bit

Sumatorul este un element fundamental în arhitectura circuitelor digitale, având un rol esențial în funcționarea Unității Aritmetico-Logice (UAL). Importanța sa este atât de mare încât performanța întregului sistem depinde în mod direct de eficiența sumatoarelor utilizate cu cât acestea operează mai rapid, cu atât viteza de calcul a întregului circuit crește.
La nivel elementar, sumatorul este un circuit combinațional care operează cu:
Intrări:

- Doi biți care trebuie adunați (a și b)
- Un bit de transport (carry) de la poziția precedentă, mai puțin semnificativă (cin)

Ieșiri:

- Bitul sumă (s)
- Bitul de transport (carry) generat pentru poziția următoare, mai semnificativă (cout)

Această structură de bază permite efectuarea operațiilor de adunare la nivel de bit, reprezentând fundamentul pentru operații aritmetice mai complexe.

Expresiile booleene ale ieșirilor sunt după cum urmează:

- si = (xi XOR yi) XOR cIN.
- cOUT = (xi AND yi) OR ((xi XOR yi) AND cIN).

Operația “sau exclusiv” ( ) are următoarea formulă de calcul:

- xi XOR yi = xi$\underline{yi}$ + $\underline{xi}$yi

x_i	y_i	T_i	T_{i+1}	S_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

Tabel 2.2 Tabelul de adevar al Sumatorului pe un bit.

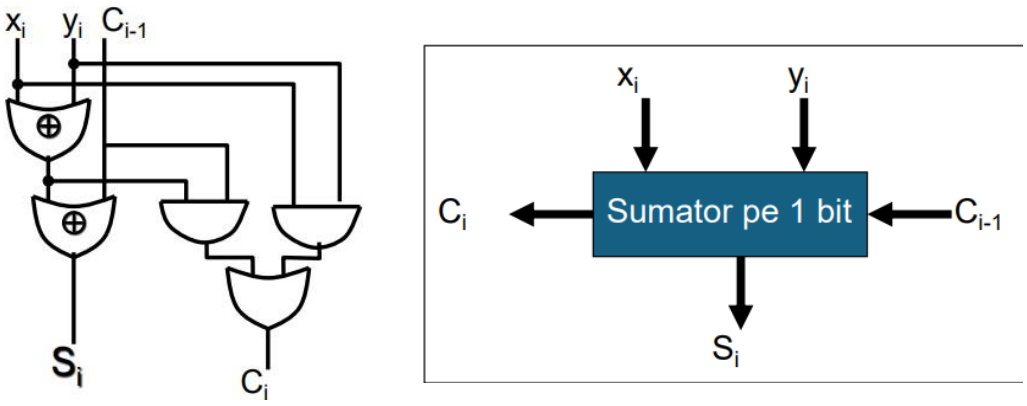


Figura 2.3 Schema sumatorului elementar

Sumator pe 24 de biți adună doi vectori de 24 de biți (xi și yi), folosind un **sumator pe 1 bit** pentru fiecare poziție, într-o arhitectură de tip "ripple-carry".

- Fiecare poziție calculează suma folosind transportul de la poziția anterioară.
- **cOUT** reprezintă transportul final (bitul de transport al rezultatului).

Componenta folosește un semnal intermediar carry pentru a propaga transportul între biți.

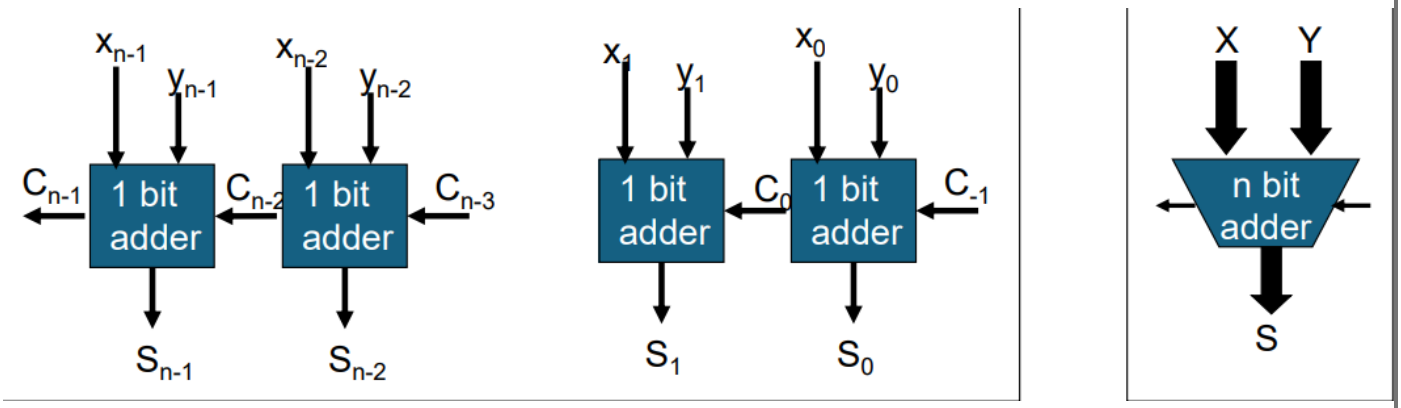


Figura 2.4 Schema sumatorului pe 24 de biti

Scăzător se bazează pe metoda complementului față de 2 pentru scădere:

- Complementul față de 2 al unui număr se obține prin inversarea tuturor bitilor (not(yi)) și adăugarea unei unități (cIN='1').
- Similar cu sumatorul, folosește un **sumator pe 1 bit** pentru fiecare poziție, iar rezultatul este obținut prin propagarea transportului.

Ieșirea finală suma este diferența, iar cOUT reprezintă transportul complementar.

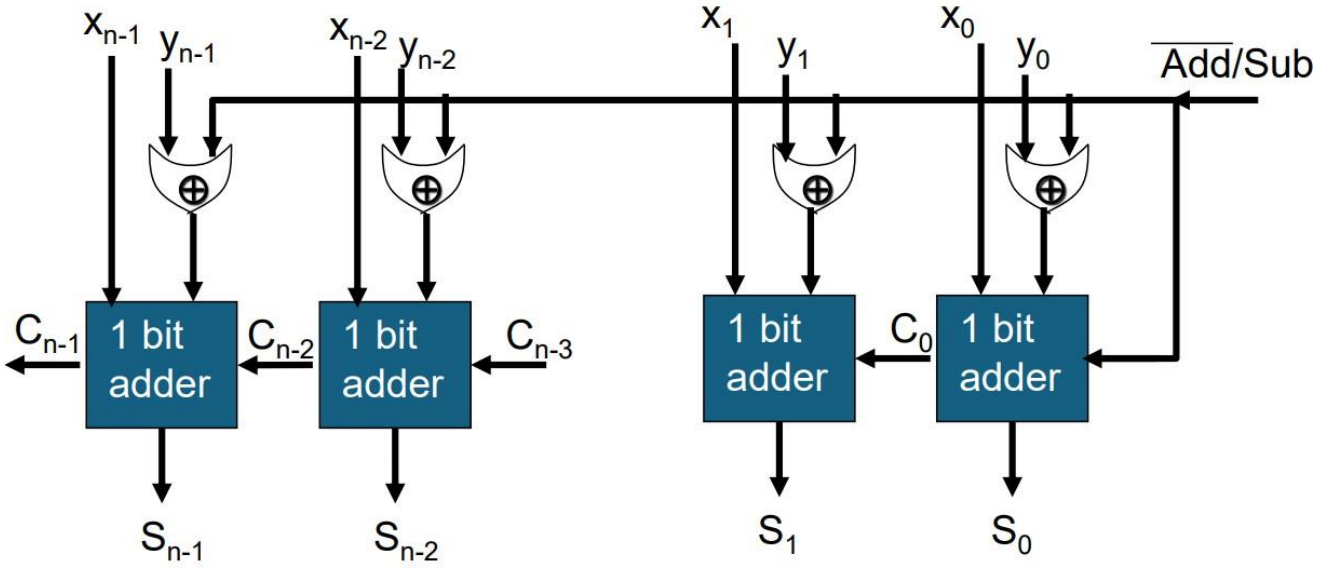


Figura 2.5 Schema sumatorului pe 24 de biti

2.7 Inmultirea Matriceala

Circuitele de înmulțire matriceală au o caracteristică distinctivă care le diferențiază de circuitele secvențiale: ele folosesc logică combinațională suplimentară pentru a efectua înmulțirea într-o singură etapă. Această operație este realizată prin componente combinaționale simple, care îndeplinesc două funcții de bază:

- Adunarea biților
- Deplasarea biților

Aceste operații elementare sunt aplicate fie la nivel de bit individual, fie pe grupuri mici de biți, permițând astfel un calcul rapid și eficient al produsului final.

Înmulțirea se va realiza pentru două numere întregi fără semn, X și Y, fiind reprezentați sub formele binare corespunzătoare: $X = x_{n-1},...x_1, x_0$, respectiv $Y = y_{n-1},...y_1, y_0$. În acest caz, produsul P se poate scrie sub forma:

$$P = X * (\sum_{i=0}^{n-1} 2^i * yi)$$

Ecuația de mai sus se poate rescrie astfel:

$$P = \sum_{i=0}^{n-1} 2^i * (\sum_{j=0}^{n-1} xi * yi * 2^j)$$

Pentru realizarea înmulțirii, avem nevoie de două componente principale:

1. Blocul de Calcul al Produsului Elementar:
- Se folosesc porți ȘI pentru această operație
- Alegerea porților ȘI este naturală deoarece înmulțirea aritmetică a doi biți este identică cu înmulțirea lor logică
- Toate produsele elementare dintre biții înmulțitorului și deînmulțitului sunt calculate simultan (concurent)

2. Blocul de Însumare:
- Utilizează sumatoare elementare aranjate într-o matrice
- Funcționează similar cu un sumator bidimensional cu propagare succesivă a transportului
- În anumite poziții, sumatoarele complete pot fi înlocuite cu semisumatoare pentru optimizare

3. Mecanismul de Deplasare:
- Deplasarea biților (reprezentată de factorii 2^i și 2^j în formula înmulțirii)
- Se realizează prin poziționarea fizică deplasată a nivelurilor de sumatoare
- Deplasarea se face pe două direcții (x și y)

Această arhitectură permite calculul eficient al produsului final prin combinarea calculului paralel al produselor elementare cu însumarea și deplasarea corespunzătoare a rezultatelor intermediare.

Exemplul unei astfel de înmulțiti descrise până acum este redat în figura.

				x_3	x_2	x_1	x_0 *
				y_3	y_2	y_1	y_0
0	0	0	0	x_3*y_0	x_2*y_0	x_1*y_0	x_0*y_0
0	0	0	x_3*y_1	x_2*y_1	x_1*y_1	x_0*y_1	0
0	0	x_3*y_2	x_2*y_2	x_1*y_2	x_0*y_2	0	0
0	x_3*y_3	x_2*y_3	x_1*y_3	x_0*y_3	0	0	0
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

Figura 2.6 Inmultirea matriceala pe 4 biti

Biții produsului final se pot calcula după următoarele formule:

$$P_0 = x_0 y_0$$

$$P_1 = x_1 y_0 + x_0 y_1$$

$$P_2 = x_2 y_0 + x_1 y_1 + x_0 y_2$$

$$P_3 = x_3 y_0 + x_2 y_1 + x_1 y_2 + x_0 y_3$$

$$P_4 = x_3 y_1 + x_2 y_2 + x_1 y_3$$

$$P_5 = x_3 y_2 + x_2 y_3$$

$$P_6 = x_3 y_3$$

Analiza
3.1 Algoritmul De Adunare

Algoritmul de adunare în virgulă mobilă este un proces complex care necesită mai multe componente interconectate. Adunătorul reprezintă elementul central al acestui sistem, iar eficiența sa influențează direct performanța întregului proces.

COMPONENTE PRINCIPALE:

1. **Scăzător pe 8 biți**

- Calculează diferența dintre exponenții numerelor (EA - EB)
- Determină direcția de aliniere a mantiselor

2. **Componentă de deplasare (shifter) pe 24 biți**

- Deplasează mantisa numărului cu exponentul mai mic
- Aliniază mantisele pentru adunarea corectă

3. **Adunător pe 24 biți**

- Realizează adunarea mantiselor aliniate
- Gestionează complementarea când este necesar

ALGORITMUL DE ADUNARE (Pași):

1. **Încărcarea Operanzilor Exemplu:**

- X = 1.010 (1.25 în zecimal)
- Y = 0.110 (0.75 în zecimal)

2. **Compararea Exponenților - 5 Cazuri Posibile:**

a) Exponenți egali (ex = ey)

- Se adună direct mantisele
- Se păstrează exponentul

b) ex > ey, diferență < lungime_mantisă

- Deplasare mantisă Y la dreapta

- Se păstrează exponentul lui X

c) ex > ey, diferență ≥ lungime_mantisă

- Se păstrează X ca rezultat
- Y este prea mic pentru a influența rezultatul

d) ex < ey, diferență < lungime_mantisă

- Deplasare mantisă X la dreapta
- Se păstrează exponentul lui Y

e) ex < ey, diferență ≥ lungime_mantisă

- Se păstrează Y ca rezultat
- X este prea mic pentru a influența rezultatul

3. Adunarea Mantiselor

- Se adună mantisele aliniate

4. **Determinarea Semnului**

- Negativ dacă unul din operanzi este negativ
- Pozitiv în caz contrar

5. **Formarea Rezultatului Final**

- Combinarea semnului, exponentului și mantisei rezultate

3.2 FlowChart pentru algoritmul de adunare

FlowChart-ul pentru algoritmul de adunare:

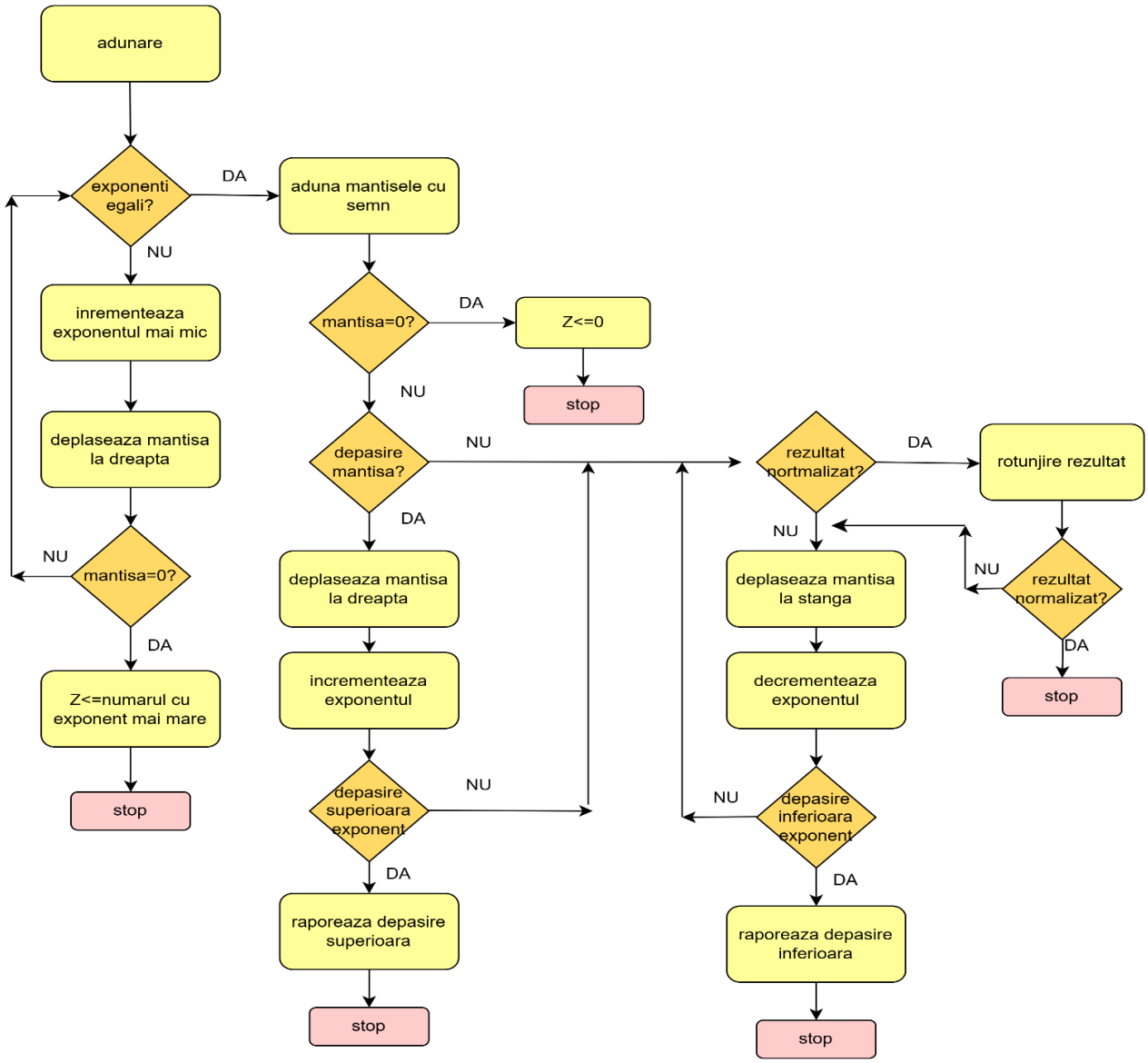


Figura 2.8 Diagrama de flux de date pentru adunare

3.3 Algoritmul De Inmultire

PROCESUL DE ÎNMULȚIRE ÎN VIRGULĂ MOBILĂ

1. **Calculul Exponentului Final:**

- Se adună exponenții celor doi operanzi
- Se scade bias-ul (127) din rezultat
- Aceasta dă exponentul final al rezultatului

2. **Înmulțirea Mantiselor:**

- Se utilizează o componentă de înmulțire specializată
- Intrare: două mantise pe 24 de biți fiecare
- Ieșire: produs pe 48 de biți

3. **Normalizarea Rezultatului:**

- Cazul 1: Primul bit este 1
    - Se selectează următorii 23 de biți pentru mantisa finală
    - Se normalizează rezultatul
- Cazul 2: Primul bit este 0
    - Se selectează 23 de biți începând cu al treilea bit
    - Se ajustează rezultatul corespunzător

4. **Calculul Semnului:**

- Se aplică operația XOR între semnele operanzilor inițiali
- Aceasta determină semnul rezultatului final

5. **Componenta de Adunare**:

- Se utilizează un adunător cu propagare anticipată a transportului
- Dimensionat la 24 de biți (nu 23)
- Bitul extra (al 24-lea) este "bitul ascuns"
    - Nu apare în rezultatul final în virgulă mobilă
    - Este esențial pentru precizia calculelor intermediare

Această arhitectură asigură precizia calculelor și respectă standardele pentru operațiile în virgulă mobilă, păstrând atât acuratețea cât și eficiența computațională.

3.4 FlowChart pentru algoritmul de inmultire

FlowChart pentru algoritmul de inmultire:

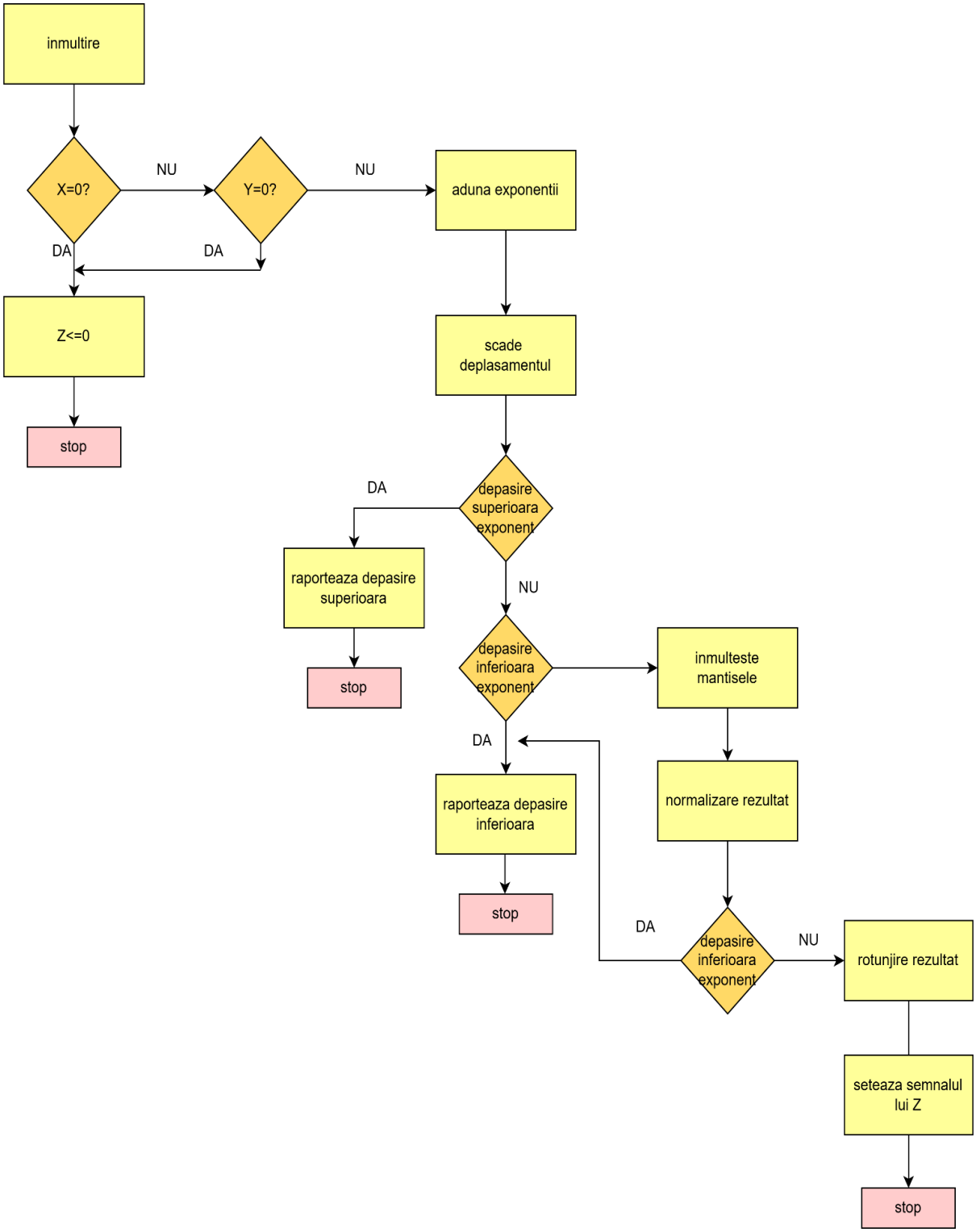


Figura 2.8 Diagrama de flux de date pentru inmulti

3.5 FlowChart pentru ALU

FlowChart pentru cei doi algoritmi:

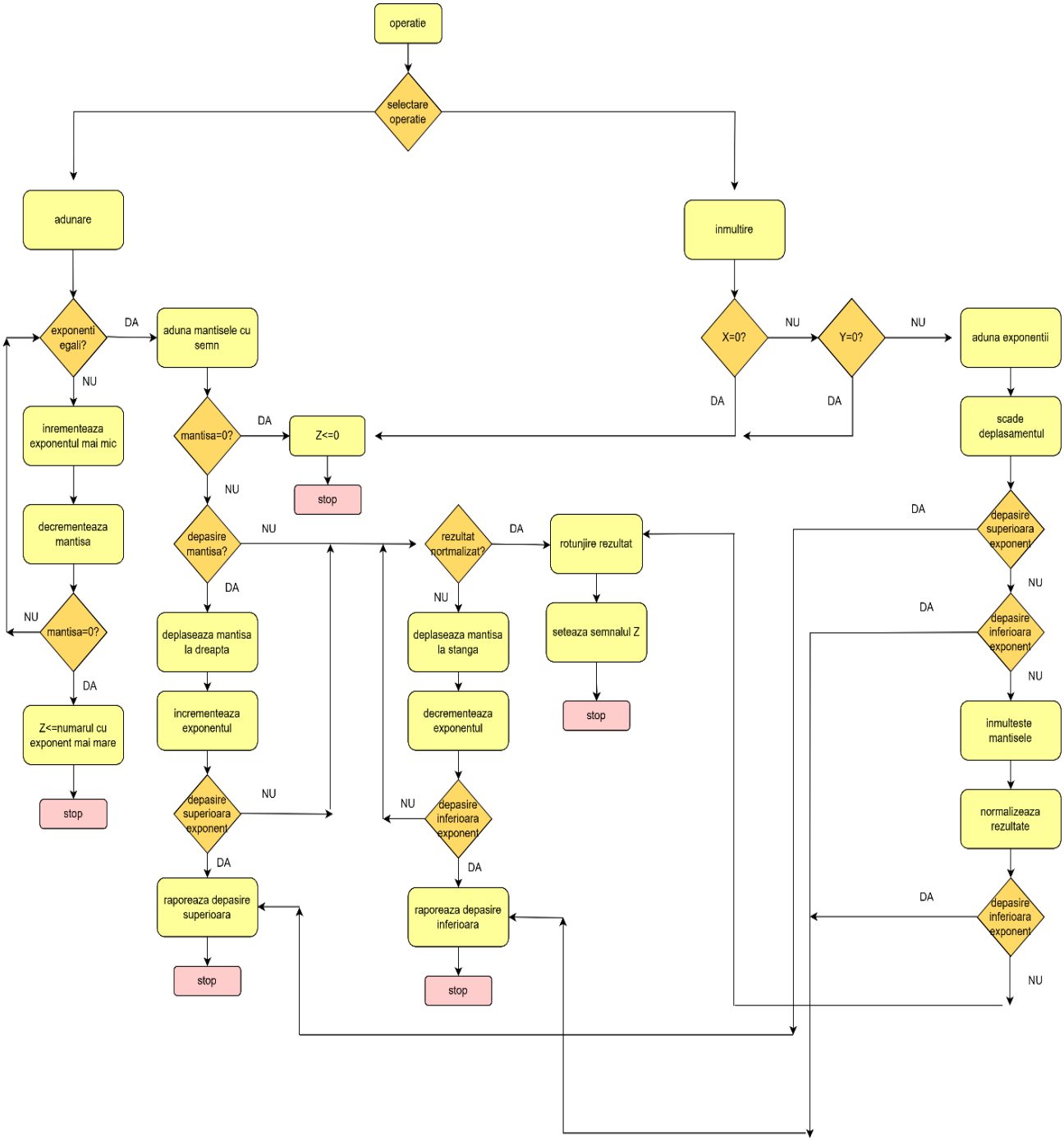
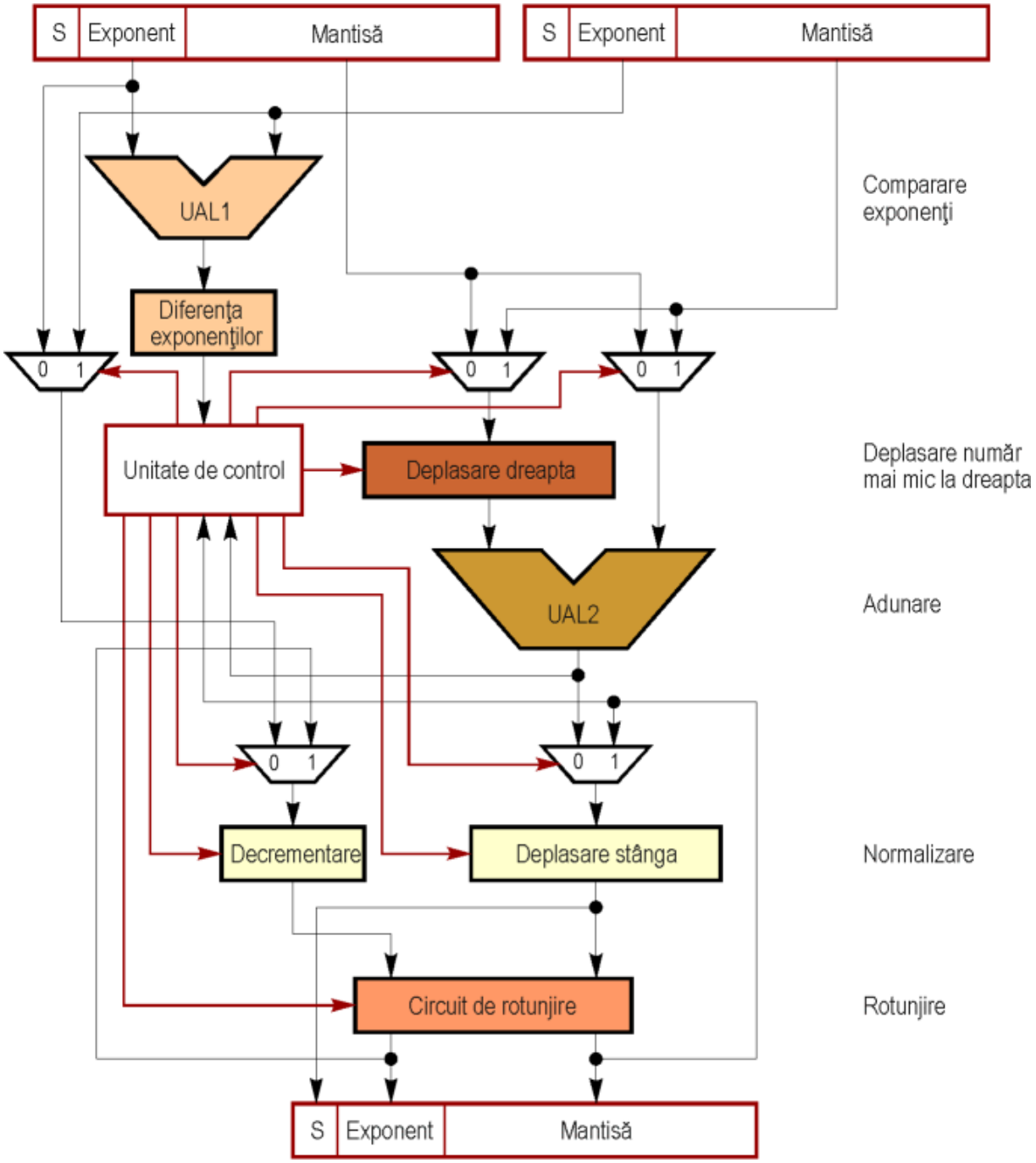


Figura 2.9 Diagrama de flux de date pentru ALU

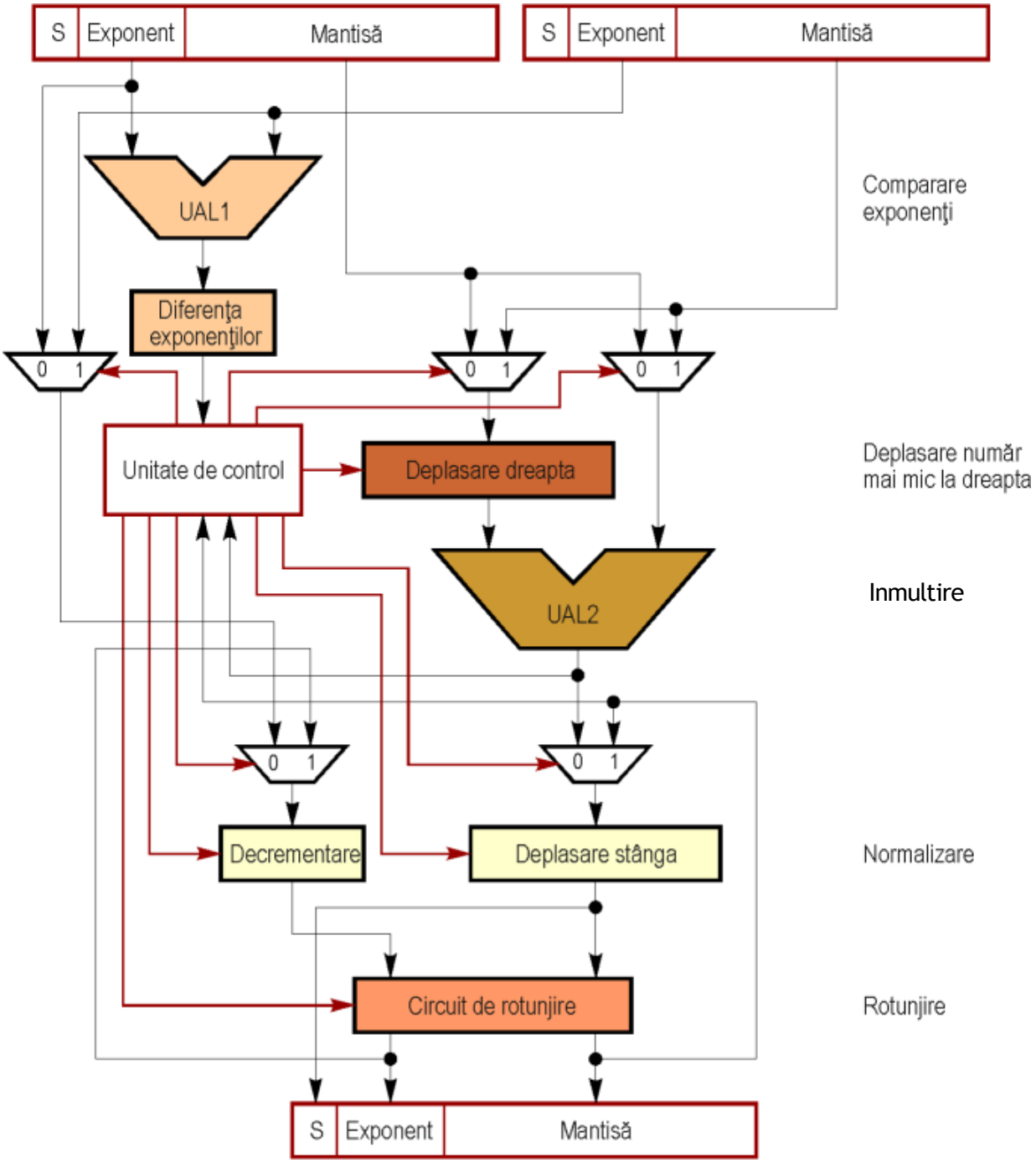
Proiectare

4.1 Schema Bloc

Schema bloc a algoritmului de adunare:



Schema bloc a algoritmului de inmultire:



Implementare
5.1 Adunarea

Implementarea operațiilor în virgulă mobilă necesită o arhitectură complexă de componente interconectate care lucrează împreună pentru a asigura precizia calculelor. În centrul acestui sistem se află procesul de manipulare a exponenților și mantiselor extrase din operanzii A și B.

Arhitectura include componenta UAL1, care are rolul crucial de a calcula diferența dintre exponenți și de a determina când și cum trebuie ajustate mantisele. Sistemul integrează module specializate precum reg, sumatorMantise, UAL2, rightShifter, normalizare si rotunjire fiecare având un rol specific în manipularea precisă a exponenților și mantiselor pentru a asigura acuratețea operației de adunare.

Pentru adunarea mantiselor pe 24 de biți, implementarea folosește o combinație de componente fundamentale. La bază se află elementele sumatorMantise și sumatorPeBit, care sunt integrate într-un sumator cu propagare în cascadă succesivă a transportului (ripple carry adder). sumatorMantise procesează doi biți de intrare împreună cu un bit de transport anterior (carry-in), generând suma și bitul de transport pentru următoarea poziție (carry-out).

Sumatorul cu propagare în cascadă succesivă a transportului gestionează întregul proces de adunare a celor 24 de biți ai mantiselor, asigurând propagarea corectă a transportului între poziții. Această arhitectură garantează că rezultatul final, stocat în componenta rezultat, respectă cu strictețe formatul numerelor în virgulă mobilă, incluzând semnul, exponenții calculați și mantisa rezultată.

Întregul sistem este proiectat pentru a asigura nu doar corectitudinea matematică a operațiilor, ci și precizia necesară în manipularea numerelor în virgulă mobilă, făcând posibilă efectuarea calculelor complexe cu un grad înalt de acuratețe.

1.Componenta de stocare a datelor- reg

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity reg is
6  Port(adress : in  std_logic_vector(1 downto 0);
7       dataOut : out std_logic_vector (31 downto 0));
8  end reg;
9
10 architecture Behavioral of reg is
11      type reg is array (0 to 3) of std_logic_vector(31 downto 0);
12
13      signal valori : reg := (
14     "10111111110000000000000000000000",-- -1.5
15     "01000000001000000000000000000000", --2.5
16      "01000101000011111111111111001100", -- 2303.9873
17      "11000101000100000001001110110110" -- -2305.232
18      );
19
20
21 begin
22
23 dataOut <= valori(to_integer(unsigned(adress)));
24
25 end Behavioral;
```

Este ca o mini memorie ROM cu 4 sertare (poziții de memorie), unde fiecare sertar conține un număr în virgulă mobilă stocat în format binar. Când primește o adresă (0-3), componenta returnează numărul

stocat în acea poziție - similar cu un bibliotecar care găsește rapid cartea cerută după numărul raftului.

2. Sumator pe 1 bit

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity sumatorPe1Bit is
    Port (
        xi : in std_logic;
        yi : in std_logic;
        cIN : in std_logic;
        cOUT : out std_logic;
        si : out std_logic
    );
end sumatorPe1Bit;

architecture Behavioral of sumatorPe1Bit is

begin
    process(xi, yi, cIN)
    begin

        si <= (xi xor yi) xor cIN;
        cOUT <= (xi and yi) or ((xi xor yi) and cIN);

    end process;

end Behavioral;
```

Acest cod VHDL implementează un sumator pe 1 bit, care calculează suma a două biți de intrare (xi și yi) și a unui bit de transport (cIN), generând un bit de sumă (si) și un bit de transport de ieșire (cOUT) pe baza expresiilor logice corespunzătoare.

3. Sumator pe 24 de biti

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity sumatorMantise is
Port (xi: in std_logic_vector(23 downto 0);
      yi: in std_logic_vector(23 downto 0);
      cIN: in std_logic;
      cOUT: out std_logic;
      suma: out std_logic_vector(23 downto 0));
end sumatorMantise;

architecture Behavioral of sumatorMantise is

component sumatorPe1Bit is
    Port (
        xi : in std_logic;
        yi : in std_logic;
        cIN : in std_logic;
        cOUT : out std_logic;
        si : out std_logic
    );
end component;

signal carry: std_logic_vector(23 downto 0);

begin

sumator0: sumatorPe1Bit port map
          (xi=>xi(0),
           yi=>yi(0),
           cIN=>cIN,
           cOUT=>carry(0),
           si=>suma(0));
```

```vhdl
sumatori: for i in 1 to 23 generate
            sumator: sumatorPe1Bit port map
                     (xi=>xi(i),
                      yi=>yi(i),
                      cIN=>carry(i-1),
                      cOUT=>carry(i),
                      si=>suma(i));

end generate;
cOUT<=carry(23);
end Behavioral;
```

Acest cod VHDL implementează un sumator pentru mantise pe 24 de biți, folosind o structură ierarhică în care un sumator pe 1 bit este instanțiat în mod repetat printr-o generație iterativă (generate), pentru a calcula suma pe toți cei 24 de biți, propagând transportul între etape.

4. Componentea ce verifica exponentii-UAL1

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ual1 is
  Port ( a: in std_logic_vector(31 downto 0);
         b: in std_logic_vector(31 downto 0);
         diferentaExponenti: out std_logic_vector(7 downto 0);
         nrMaiMare: out std_logic;
         enable: in std_logic);
end ual1;

architecture Behavioral of ual1 is

signal exponent_a: std_logic_vector(7 downto 0);
signal exponent_b: std_logic_vector(7 downto 0);

signal mantisa_a: std_logic_vector(23 downto 0);
signal mantisa_b: std_logic_vector(23 downto 0);

signal diferenta: std_logic_vector(7 downto 0);

signal mare: std_logic;

begin

exponent_a <= a(30 downto 23);
exponent_b <= b(30 downto 23);
mantisa_a <= '1' & a(22 downto 0);
mantisa_b <= '1' & b(22 downto 0);
```

```
process(exponent_a, exponent_b, mantisa_a, mantisa_b, enable)
begin
if enable = '1' then
   if unsigned(exponent_a) > unsigned(exponent_b) then
       mare <= '0';
   elsif unsigned(exponent_a) < unsigned(exponent_b) then
       mare <= '1';
   elsif unsigned(exponent_a) = unsigned(exponent_b) then
          if unsigned(mantisa_a) >= unsigned(mantisa_b) then
              mare <= '0';
          elsif unsigned(mantisa_a) < unsigned(mantisa_b) then
              mare <= '1';
          end if;
   end if;
   end if;
end process;

   nrMaiMare <= mare;

process(mare, exponent_a, exponent_b, enable)
begin
if enable = '1' then
   if mare = '0' then
       diferenta <= std_logic_vector(unsigned(exponent_a) - unsigned(exponent_b));
   elsif mare= '1' then
       diferenta <= std_logic_vector(unsigned(exponent_b) - unsigned(exponent_a));
   end if;
end if;
end process;

process(diferenta, enable)
begin
 if enable = '1' then
    if diferenta <= "00011000" then
        diferentaExponenti <= diferenta;
    else
        diferentaExponenti <= "00011001";
    end if;
 end if;
 end process;

end Behavioral;
```

Acest cod VHDL implementează o unitate aritmetică logică (UAL) care compară două numere pe 32 de biți în format floating-point. Extrage exponenții și mantisele numerelor de intrare, determină care dintre ele este mai mare și calculează diferența dintre exponenți. Rezultatul este limitat la o valoare maximă de 25 (în binar 00011001) dacă diferența depășește această valoare. Semnalul nrMaiMare indică care număr este mai mare, iar diferența exponenților este returnată în diferentaExponenti.

5. Componenta de shiftare rightShifter

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity main is
    Port ( a : out STD_LOGIC_VECTOR(31 downto 0);
           b : out STD_LOGIC_VECTOR(31 downto 0);
           clk : in STD_LOGIC;
           rst : in STD_LOGIC;
           op : in STD_LOGIC;
           rezultat : out STD_LOGIC_VECTOR(31 downto 0));
end main;

architecture Behavioral of main is
component reg is
Port(adress : in  std_logic_vector(1 downto 0);
     dataOut : out std_logic_vector (31 downto 0));
end component;

component FSM is
    Port (clk: in  std_logic;
          rst: in  std_logic;
          a: out std_logic_vector(31 downto 0);
          b: out std_logic_vector(31 downto 0);
          rezultat: out std_logic_vector(31 downto 0));
end component;

component FSM2 is
Port ( clk : in  std_logic;
       rst : in  std_logic;
       a: out std_logic_vector(31 downto 0);
       b: out std_logic_vector(31 downto 0);
       rezultat: out std_logic_vector(31 downto 0));
```

```
end component;

signal rezultat1, rezultat2: std_logic_vector(31 downto 0);
signal operatie : std_logic;
signal data_a, data_b : std_logic_vector(31 downto 0);
signal adress_a, adress_b: std_logic_vector(1 downto 0);

begin

adress_a <= "00";
adress_b <= "01";

primulNumar: reg port map
                  (adress => adress_a,
                   dataOut => data_a);

alDoileaNumar: reg port map
                  (adress => adress_b,
                   dataOut => data_b);

FSM_mapare: FSM port map
             (clk => clk,
              rst => rst,
              a => a,
              b => b,
              rezultat => rezultat1);

FSM2_mapare: FSM2 port map
             (clk => clk,
              rst => rst,
              a => a,
              b => b,
                rezultat => rezultat2);

 rezultat <= rezultat1 when op = '0' else rezultat2;

end Behavioral;
```

Acest cod VHDL implementează un shifter logic spre dreapta pentru mantise pe 23 de biți, extinzând mantisa de intrare cu un bit suplimentar (1) pentru a include bitul ascuns din formatul floating-point. Dacă semnalul enable este activ (1), valoarea mantisei este deplasată spre dreapta cu un număr specificat de poziții (shift_amount). Rezultatul deplasării este returnat ca o mantisă pe 24 de biți în semnalul mantissa_out.

6. Componenta de adunare a mantiselor-UAL2

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity UAL2 is
    Port (
        mantisa_a : in std_logic_vector(23 downto 0);
        mantisa_b : in std_logic_vector(22 downto 0);
        semn_a : in std_logic;
        semn_b : in std_logic;
        mantisa_out : out std_logic_vector(23 downto 0);
        semn_out : out std_logic;
        carry_out_shift : out std_logic;
        nrMare : in std_logic;
        aluCTRL: in std_logic;
        enable: in std_logic);
end UAL2;

architecture Behavioral of UAL2 is

    signal carry_out_adunare : std_logic;
    signal carry_out_scadere : std_logic;
    signal suma_mantisa : std_logic_vector(23 downto 0);
    signal diferenta_mantisa : std_logic_vector(23 downto 0);
    signal diferenta_mantisa2 : std_logic_vector(23 downto 0);

    signal mantisa_a_aux : std_logic_vector(23 downto 0);
    signal mantisa_b_aux : std_logic_vector(23 downto 0);

    component sumatorMantise is
        Port (
            xi : in std_logic_vector(23 downto 0);
            yi : in std_logic_vector(23 downto 0);
            cIN : in std_logic;
            cOUT : out std_logic;
            suma : out std_logic_vector(23 downto 0)
        );
    end component;

    component scazatorMantise is
        Port (
            xi : in std_logic_vector(23 downto 0);
            yi : in std_logic_vector(23 downto 0);
            cIN : in std_logic;
            cOUT : out std_logic;
            suma : out std_logic_vector(23 downto 0)
        );
    end component;

begin

    mantisa_a_aux <= mantisa_a;
    mantisa_b_aux <= '1' & mantisa_b;

    sumator: sumatorMantise
        port map (
            xi => mantisa_a_aux,
            yi => mantisa_b_aux,
            cIN => '0',
            cOUT => carry_out_adunare,
            suma => suma_mantisa
        );

    scazator: scazatorMantise
```

```
65          port map (
66              xi => mantisa_b_aux,
67              yi => mantisa_a_aux,
68              cIN => '0',
69              cOUT => carry_out_scadere,
70              suma => diferenta_mantisa
71          );
72
73      process(semn_a, semn_b, suma_mantisa, diferenta_mantisa, carry_out_adunare, nrMare, enable)
74      begin
75
76      if aluCTRL='1' then
77          if enable='1' then
78              if carry_out_adunare = '1' then
79                  mantisa_out <= suma_mantisa;
80                  carry_out_shift <= '1';
81              else
82                  mantisa_out <= suma_mantisa;
83                  carry_out_shift <= '0';
84              end if;
85              end if;
86              semn_out <= semn_a;
87
88          elsif aluCTRL = '0' then
89              if enable = '1' then
90              if nrMare = '0' then
91                  mantisa_out <= diferenta_mantisa;
92                  semn_out <= semn_a;
93              elsif nrMare = '1' then
94                  mantisa_out <= diferenta_mantisa;
95                  semn_out <= semn_b;
96              end if;
97              end if;
98              end if;
99          end process;
100
101     end Behavioral;
102
```

Acest cod VHDL implementează o unitate aritmetică logică (UAL) care realizează adunarea sau scăderea între două mantise de 24 de biți, determină semnul rezultatului și gestionează transportul de deplasare (carry_out_shift), în funcție de semnalele de control (aluCTRL, nrMare) și de activare (enable).

7. Componenta de normalizare-normalizare

```vhdl
use ieee.std_logic_unsigned.all;
entity normalizare is
  Port ( mantisa_in : in std_logic_vector(23 downto 0);
         exponent_in : in std_logic_vector ( 7 downto 0);
         mantisa_out : out std_logic_vector(22 downto 0);
         exponent_out : out std_logic_vector ( 7 downto 0);
         carry_out   : in  std_logic;
         enable: in std_logic);
end normalizare;
architecture Behavioral of normalizare is
begin
    process(mantisa_in, exponent_in, carry_out, enable)
    variable exp : unsigned (7 downto 0) ;
    variable mantisaTemp : unsigned(23 downto 0);
    begin
    if enable = '1' then
        exp := unsigned(exponent_in);
        mantisaTemp := unsigned(mantisa_in);
   if carry_out = '1' then
            mantisaTemp :=  mantisaTemp srl 1;
            exp := exp + 1;
   elsif mantisaTemp(23) = '0' then
            while mantisaTemp(23) = '0' and exp > 0 loop
               mantisaTemp:= mantisaTemp sll 1;
                exp := exp - 1;
            end loop;
        end if;
     mantisa_out <= std_logic_vector(mantisaTemp(22 downto 0));
     exponent_out <= std_logic_vector(exp);
     end if;
    end process;
end Behavioral;
```

Acest cod VHDL implementează un modul de normalizare pentru numere floating-point, care ajustează mantisa și exponentul pe baza semnalului de transport (carry_out) și a biților mantisei; dacă există un transport, mantisa este deplasată spre dreapta, iar exponentul crește, iar în caz contrar, mantisa este deplasată spre stânga până ce bitul cel mai semnificativ devine 1, reducând corespunzător exponentul.

8. Componenta de rotunjire -rotunjire

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity rotunjire is
    Port ( mantisa : in  STD_LOGIC_VECTOR(22 downto 0);
           rezultat : out STD_LOGIC_VECTOR(22 downto 0);
           enable: in std_logic);
end rotunjire;

architecture Behavioral of rotunjire is
begin
    process (mantisa, enable)
    begin
    if enable = '1' then
        if mantisa(0) = '1' then
            rezultat <= mantisa + "00000000000000000000001";
        else
            rezultat <= mantisa;
        end if;
    end if;
    end process;
end Behavioral;
```

Acest cod VHDL implementează un modul de rotunjire pentru o mantisă pe 23 de biți, care adaugă 1 la mantisă dacă ultimul bit (mantisa(0)) este 1, realizând astfel o rotunjire către cel mai apropiat număr superior, activată de semnalul enable.

5.2 Inumltirea

Algoritmul de înmulțire în virgulă mobilă este un sistem complex care gestionează înmulțirea a două numere în format special. Acest sistem este construit modular, cu componente separate care lucrează împreună pentru a realiza toate operațiile necesare, începând cu organizarea exponenților și mantiselor numerelor de intrare A și B.

La baza sistemului se află mai multe componente specializate precum sumatorExponenti, inmultireaMantiselor, verificaExponenti, normalizareInmultire și incrementeazaExponent, fiecare având un rol specific în manipularea datelor. Rezultatul final combină exponentul rezultat, mantisa rezultantă și semnul rezultatului, toate respectând standardul virgulă mobilă.

În centrul acestui sistem se află componenta inmultireaMatricelor, care realizează înmulțirea mantiselor folosind o abordare similară înmulțirii matriceale. Aceasta folosește multiple sumatoare sumatorMantise conectate secvențial pentru a calcula produsul dintre fiecare bit al unei mantise cu întreaga mantisă a celuilalt număr. Rezultatul final este un vector de 48 de biți care conține produsul complet al mantiselor.

Componenta inmultireaMantiselor gestionează operația de înmulțire propriu-zisă a mantiselor, folosind inmultireaMatricelor ca motor de calcul. Aceasta procesează mantise de 48 de biți și un vector de control "zero" de 2 biți. Pentru calcule precise, primii 23 de biți sunt folosiți pentru mantisele a și b, cu al 24-lea bit setat la 1 pentru extinderea corectă a semnului.

În final, sumatorExponenti se ocupă de adunarea exponenților, procesând vectori de 8 biți și generând un rezultat de 8 biți. Această arhitectură complexă și precisă asigură că înmulțirea numerelor în virgulă mobilă este realizată eficient și corect, respectând toate cerințele și standardele necesare pentru calcule în virgulă mobilă.

1. Componenta de adunare a exponentilor- sumatorExponent

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity sumatorExponenti is
 Port (exponent_a:in std_logic_vector(7 downto 0);
       exponent_b:in std_logic_vector(7 downto 0);
       rezultat: out std_logic_vector(7 downto 0);
       enable: in std_logic);--aici
end sumatorExponenti;

architecture Behavioral of sumatorExponenti is

component sumatorPe1Bit is
 Port (xi : in std_logic;
       yi : in std_logic;
       cIN : in std_logic;
       cOUT : out std_logic;
       si : out std_logic);
end component;

signal exp_a: std_logic_vector(7 downto 0);
signal exp_b: std_logic_vector(7 downto 0);
signal aux: std_logic_vector(7 downto 0);
signal c: std_logic_vector(7 downto 0);

begin

exp_a <= exponent_a;
exp_b <= exponent_b;

sumator0: sumatorPe1Bit port map
          (xi => exp_a(0),
           yi => exp_b(0),
           cIN => '0',
           cOUT => c(0),
           si => aux(0));

sumatori: for i in 1 to 7 generate
     sumator: sumatorPe1Bit port map
              (xi => exp_a(i),
               yi => exp_b(i),
               cIN => c(i-1),
               cOUT => c(i),
               si => aux(i));

end generate;

process(aux, enable)
begin
    if enable = '1' then--aici
    if exp_a = "00000000" or exp_b = "00000000" then
        rezultat <= "01111111";
    else
        rezultat <= aux;
    end if;
    end if;     --aici
end process;

end Behavioral;
```

Acest cod implementează un modul VHDL pentru un sumator de exponenți pe 8 biți, care utilizează componente de sumare pe 1 bit pentru a calcula suma celor două intrări exponent_a și exponent_b. Dacă oricare dintre exponenți este zero, rezultatul este setat la valoarea implicită 01111111. Operația este activată de semnalul enable.

2. Componenta de inmultire a mantiselor-inmultireaMantiselor

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity inmultireaMantiselor is
  Port (mantisa_in: in std_logic_vector(47 downto 0);
        mantisa_out: out std_logic_vector(23 downto 0);
        rezultat: out std_logic_vector(47 downto 0);
        semn: out std_logic;
        clk: in std_logic;
        enable: in std_logic);--aici
end inmultireaMantiselor;

architecture Behavioral of inmultireaMantiselor is

component inmultireaMatricelor is
Port (a:in std_logic_vector(23 downto 0);
      b:in std_logic_vector(23 downto 0);
      rezultat: out std_logic_vector(47 downto 0));
end component;

signal a,b: std_logic_vector(23 downto 0);
signal rezultat1: std_logic_vector(47 downto 0);
signal mantisa_aux: std_logic_vector(23 downto 0);
signal semn1: std_logic;

begin

a <= "1" & mantisa_in(22 downto 0);
b <= "1" & mantisa_in(46 downto 24);

inmultire: inmultireaMatricelor port map
                    (a => a,
                     b => b,
                     rezultat => rezultat1);

 semn1 <= mantisa_in(23) xor mantisa_in(47);

process(clk, enable)
begin
if rising_edge(clk) then
 if enable = '1' then--aici
   if a(22 downto 0) = "00000000000000000000000" or b(22 downto 0)= "00000000000000000000000" then
        mantisa_out <= (others => '0');
    else
        mantisa_out <= semn1 & rezultat1(47 downto 25);
    end if;
    end if;--aici
end if;
end process;
    rezultat <= rezultat1;
    semn <= semn1;
end Behavioral;
```

Acest cod VHDL descrie un modul pentru multiplicarea mantiselor a două numere cu virgulă mobilă. Modulul primește o mantisă de intrare de 48 de biți și o împarte în două părți: una de 24 de biți pentru fiecare operand al multiplicării. Se folosește o componentă externă pentru efectuarea multiplicării celor două părți ale mantisei, iar rezultatul este redimensionat și semnul este calculat prin xor între semnele celor două mantise. Dacă oricare dintre părțile mantisei este zero, ieșirea este setată la zero, altfel rezultatul multiplicării este plasat într-o ieșire de 24 de biți. Modulul este sincronizat cu semnalul de ceas și controlat de un semnal de activare.

3. Componenta de verificare a exponentilor – verificaExponenti

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity verificaExponenti is
 Port (exp_in: in std_logic_vector(7 downto 0);
       exp_out: out std_logic_vector(7 downto 0);
       enable: in std_logic);--aici
end verificaExponenti;

architecture Behavioral of verificaExponenti is

begin
process(exp_in, enable)
begin
    if enable = '1' then--aici
        if(exp_in /= "UUUUUUUU") then
            exp_out <= std_logic_vector(unsigned(exp_in)-127);
        end if;
        end if;--aici
    end process;

end Behavioral;
```

Acest cod VHDL descrie un modul care verifică și ajustează exponentul unui număr cu virgulă mobilă. Modulul primește un exponent de 8 biți (exp_in) și, dacă semnalul de activare (enable) este activat, verifică dacă exponentul nu este invalid (adică nu este "UUUUUUUU", ceea ce ar însemna un exponent nedefinit). Dacă exponentul este valid, acesta este ajustat prin scăderea valorii 127 și rezultatul este transmis prin ieșirea exp_out. Astfel, se efectuează conversia exponentului pentru a-l adresa corect într-un sistem cu exponent de 8 biți, folosind ajustarea standard în formatul IEEE 754 pentru numerele în virgulă mobilă.

4. Componenta de normalizare a inmultirii – normalizareInmultire

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use ieee.std_logic_unsigned.all;

entity normalizareInmultire is
 Port (mantisa_in: in std_logic_vector(23 downto 0);
       semn_a: in std_logic;
       semn_b: in std_logic;
       ultimulBit: out std_logic;
       mantisa_out: out std_logic_vector(23 downto 0);
       enable: in std_logic);--aici
end normalizareInmultire;

architecture Behavioral of normalizareInmultire is
signal semn: std_logic;

begin
ultimulBit <= mantisa_in(22);
semn <= semn_a xor semn_b;


    process(mantisa_in, enable)
    variable mant_aux: unsigned(23 downto 0);

    begin

        mant_aux := unsigned(mantisa_in);
        if enable = '1' then---aici
        if(mant_aux(23) = '1') then
            mant_aux(23) := '0';
            while mant_aux(23) = '0' loop
        mant_aux := mant_aux sll 1;
        end loop;
        mantisa_out(23)<= semn;
        mantisa_out(22 downto 0) <= std_logic_vector(mant_aux(22 downto 0));
      else
        if mantisa_in = "00000000000000000000000" then
            mantisa_out(23) <= semn;
            mantisa_out(22 downto 0) <= "1"& mantisa_in(22 downto 1);
            mantisa_out(0) <= '0';
        else
            while mant_aux(23) = '0' loop
                mant_aux := mant_aux sll 1;
             end loop;
                mantisa_out(23)<= semn;
                mantisa_out(22 downto 0) <= std_logic_vector(mant_aux(22 downto 0));
        end if;
        end if;
      end if;--aici
  end process;

end Behavioral;
```

Acest cod VHDL descrie un modul care normalizează mantisa într-un proces de multiplicare a numerelor cu virgulă mobilă. Primește o mantisă de 24 de biți și semnele celor două numere de intrare, calculând semnul rezultatului prin xor între semnele de intrare. Dacă bitul cel mai semnificativ al mantisei este 1, se elimină și se deplasează mantisa pentru a aduce primul 1 în poziția corespunzătoare. Rezultatul este plasat într-o ieșire de 24 de biți, iar semnul este setat corespunzător. Procesul este controlat de semnalul de activare enable.

5. Componenta de incrementare a exponentilor - oincrementareExponent

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity incrementeazaExponent is
 Port (exp_in: in std_logic_vector(7 downto 0);
       ultimulBit: in std_logic;
       exp_out: out std_logic_vector(7 downto 0);
       enable: in std_logic);--aici
end incrementeazaExponent;

architecture Behavioral of incrementeazaExponent is
begin

process(exp_in, ultimulBit, enable)
begin
if enable = '1' then --aici
    if(ultimulBit = '1') then
        exp_out <= std_logic_vector(unsigned(exp_in) +1);
    else
        exp_out <= exp_in;
    end if;
    end if;--aici
    end process;

end Behavioral;
```

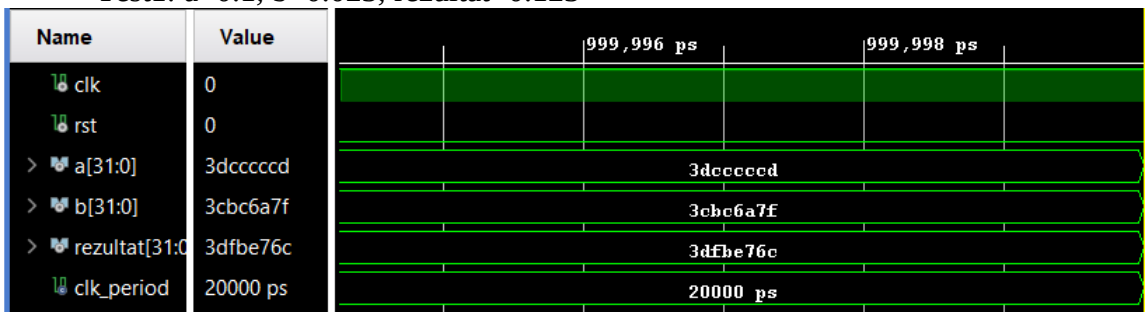
Acest cod VHDL implementează un modul care incrementează exponentul unui număr cu virgulă mobilă dacă semnalul ultimulBit este 1 și semnalul de activare enable este activ. Dacă ultimulBit este 0, exponentul rămâne neschimbat, iar rezultatul este transmis prin ieșirea exp_out.

Testare

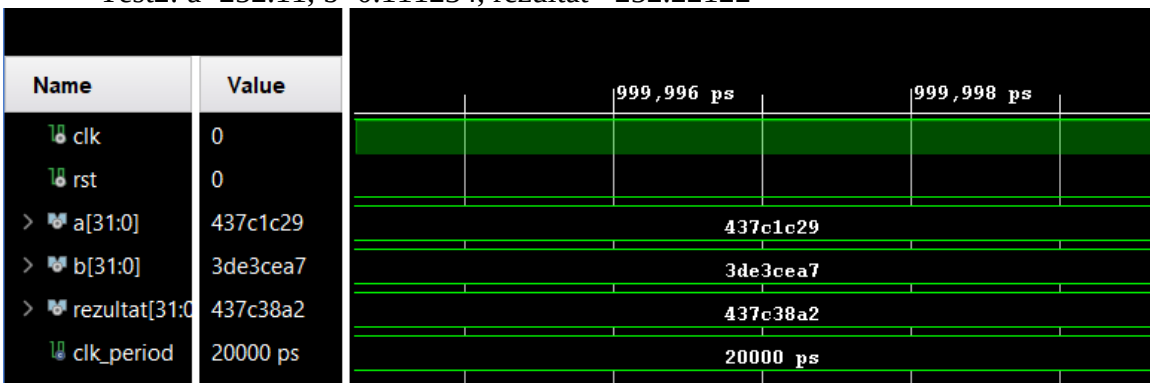
Pentru a testa corectitudinea și funcționalitatea componentelor noastre, am efectuat simulări detaliate folosind mediul Vivado. Acest proces a implicat crearea unui mediu de testare în care am introdus diferite scenarii și date de intrare pentru a evalua comportamentul fiecărei componente în parte.

Adunarea:

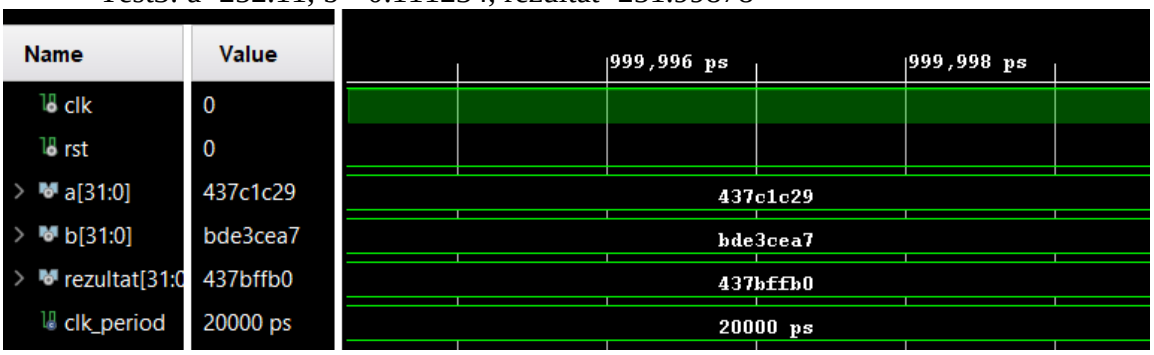
Test1: a=0.1, b=0.023, rezultat=0.123



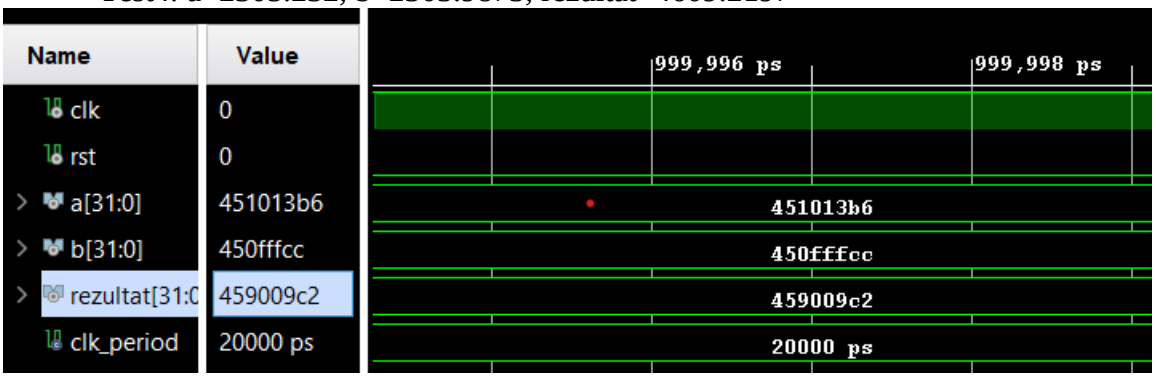
Test2: a=252.11, b=0.111234, rezultat= 252.22122



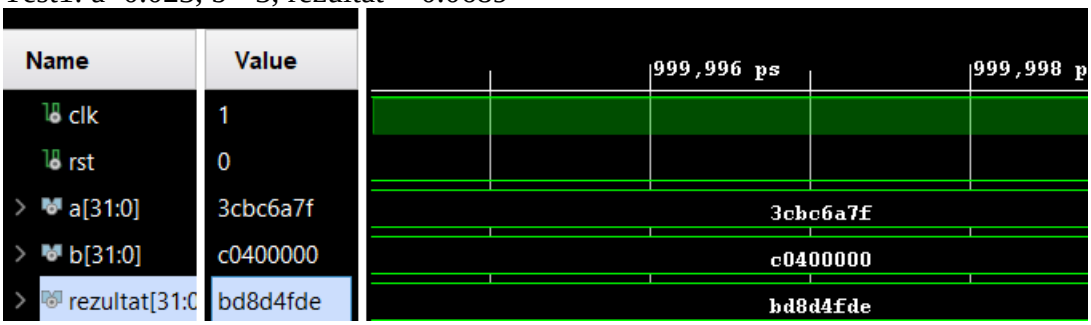
Test3: a=252.11, b=-0.111234, rezultat=251.99878



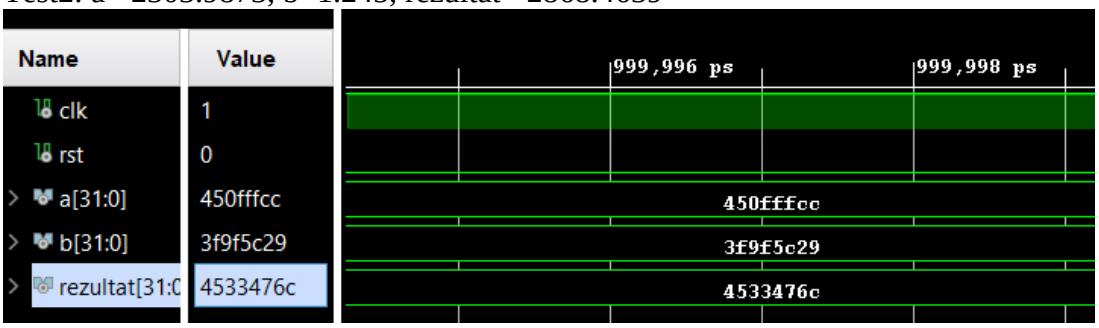
Test4: a=2305.232, b=2303.9873, rezultat=4609.2197



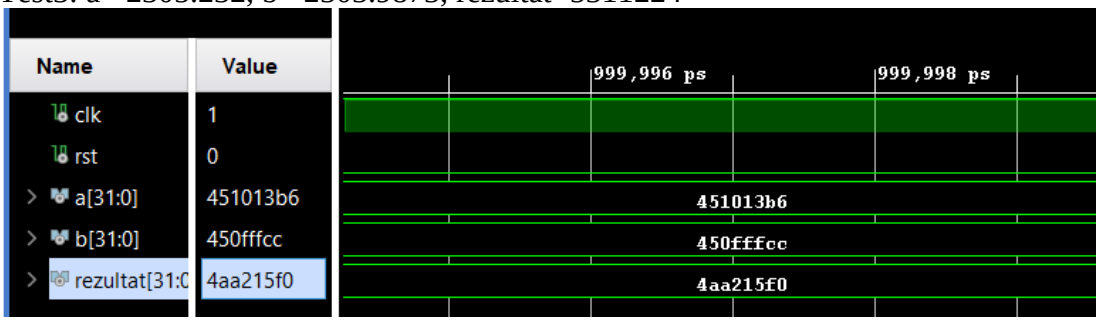
Inmultirea:
Test1: a=0.023, b=-3, rezultat= -0.0689



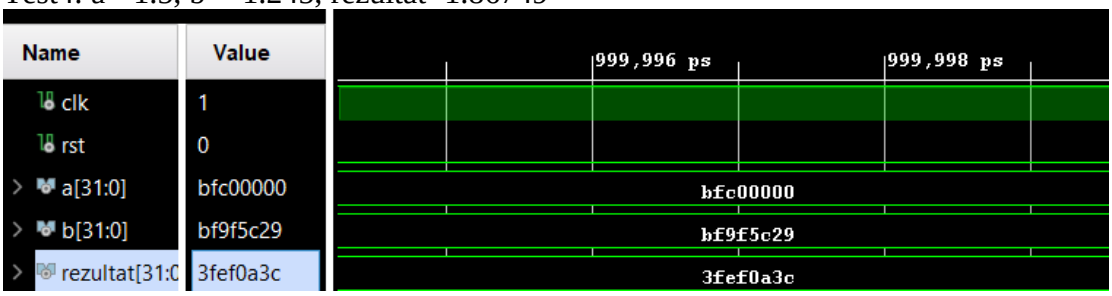
Test2: a= 2303.9873, b=1.245, rezultat= 2868.4639



Test3: a= 2305.232, b= 2303.9873, rezultat=5311224

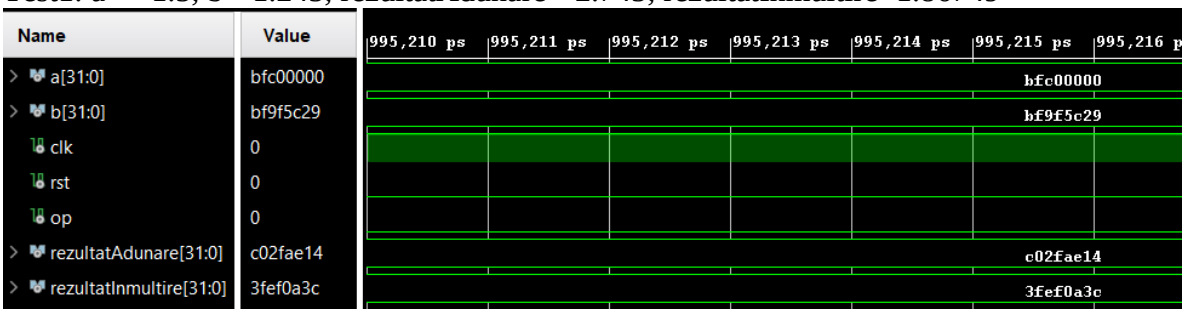


Test4: a=-1.5, b= -1.245, rezultat=1.86749

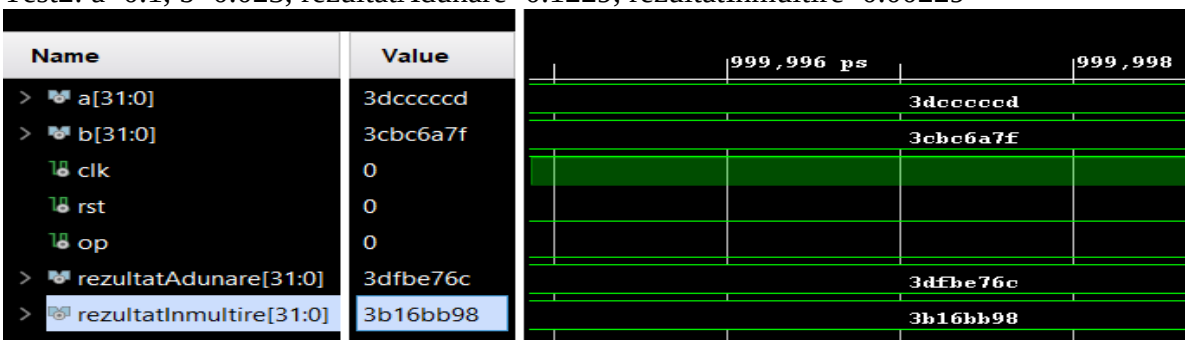


Main:

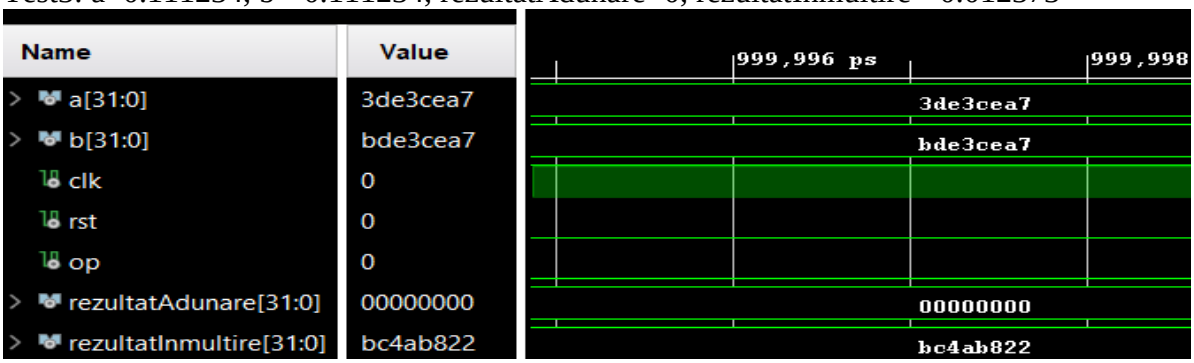
Test1: a==-1.5, b=-1.245, rezultatAdunare=-2.745, rezultatInmultire=1.86749



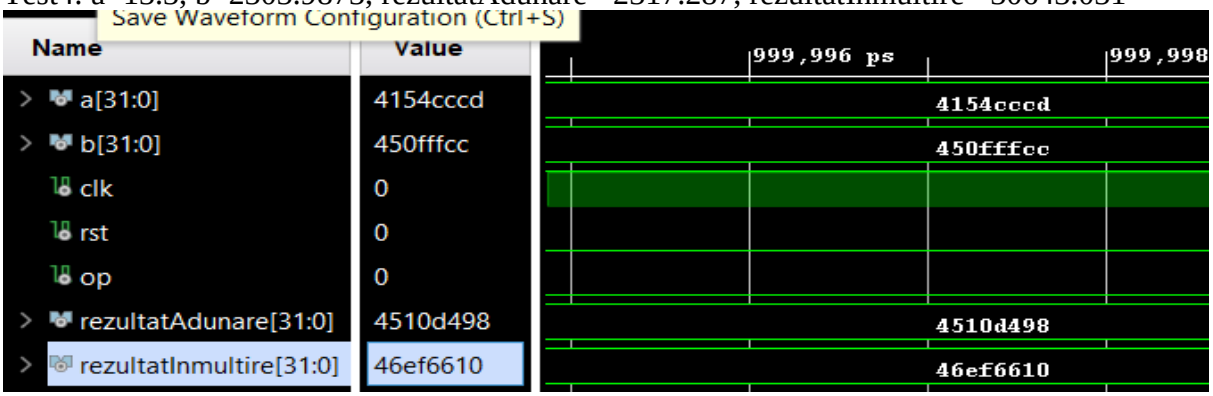
Test2: a=0.1, b=0.023, rezultatAdunare=0.1229, rezultatInmultire=0.00229



Test3: a=0.111234, b=-0.111234, rezultatAdunare=0, rezultatInmultire=-0.012373



Test4: a=13.3, b=2303.9873, rezultatAdunare= 2317.287, rezultatInmultire= 30643.031



Concluzii

Implementarea algoritmilor de adunare și înmulțire în virgulă mobilă reprezintă un fundament esențial în numeroase domenii precum calculul numeric, grafica computerizată și inginerie. Realizarea acestor operații necesită o atenție meticuloasă și o precizie deosebită în gestionarea reprezentării numerelor conform standardului IEEE 754.

Succesul acestor implementări se bazează pe manipularea exactă a exponenților și mantiselor, elemente critice pentru asigurarea acurateței operațiilor numerice. Aderarea la standardul IEEE 754 este vitală, deoarece garantează că rezultatele vor fi consistente și compatibile între diverse platforme și implementări.

Pentru a obține rezultate precise și fiabile, este esențială interacțiunea armonioasă între componentele principale ale sistemului. Acestea includ adunătoarele, înmulțitorii, precum și modulele responsabile cu verificarea și ajustarea exponenților, normalizarea și verificarea mantiselor. Fiecare componentă trebuie să își îndeplinească rolul specific cu maximă precizie, contribuind la funcționarea optimă a întregului sistem.

Astfel, crearea unor algoritmi robuști pentru operațiile în virgulă mobilă necesită o combinație de expertiză tehnică avansată și meticulozitate în abordarea detaliilor. Pilonii care susțin succesul acestor implementări sunt respectarea riguroasă a standardelor, integrarea eficientă a tuturor componentelor și procesul riguros de testare, toate acestea contribuind la asigurarea preciziei și fiabilității operațiilor numerice în virgulă mobilă.

Bibliografie

[1] Florin Oniga, De la bit la procesor. Introducere in arhitectura calculatoarelor, cap 5.5 Unitatea Aritmetica-Logica

[2] John L. Hennessy, David A. Patterson,Computer Architecture A Quantitative Approach (5th edition)

Floating-Point J.3 pag 1089

IEE format number cap J.3, J-16 pag 1091

Floating-Point Multiplication J.4, J-17 pag 1093

Floating-Point Addition J.5 pag 1097

[3]. "floating point arithmetic on division," [Online]

https://www.tutorialspoint.com/computer_organization/floating_point_arithmetic_on_division.asp.

[4]. "Adunarea şi scăderea în virgulă mobilă" [Online].

https://users.utcluj.ro/~baruch/book_ac/AC-Adunare-VM.pdf