Universitatea Tehnica din Cluj-Napoca

Calculatoare si tehnologia informatiei



Unitate de calcul in virgula mobila : adunarea si scaderea

Narita Catalin-Ioan

Tandea Alin-Dan

Grupa : 30235

Indrumator de proiect : Cristi Mocan

Data: 29.05.2017

Contents

[1.Rezumat 4](#_Toc483777176)

[2.Introducere 5](#_Toc483777177)

[3.Fundamente teoretice 5](#_Toc483777178)

[3.1 Reprezentarea numerelor in virgula mobila. Standardul IEEE 754 5](#_Toc483777179)

[3.2 Valori speciale 7](#_Toc483777180)

[3.3 Adunarea si scaderea in virgula mobila 7](#_Toc483777181)

[3.4 Organigrama operatiei de adunare 8](#_Toc483777182)

[2.5 Schema bloc a unitatii de adunare si scadere in virgula mobila 9](#_Toc483777183)

[3.6 Schema bloc a circuitului implementat 12](#_Toc483777184)

[3.7 Introducerea datelor - modulul Pmod KYPD 13](#_Toc483777185)

[4. Proiectare si implementare 13](#_Toc483777186)

[4.1 Descrierea componentelor 14](#_Toc483777187)

[3.1.1 Unitate aritmetico – logica (UAL1) 14](#_Toc483777188)

[3.1.2 Circuit de rotunjire 15](#_Toc483777189)

[3.1.3 Calcul exponent 16](#_Toc483777190)

[3.1.4 Deplasare mantisa 17](#_Toc483777191)

[3.1.5 Multiplexor generic 18](#_Toc483777192)

[3.1.6 Calcul mantisa 19](#_Toc483777193)

[5. Rezutate experimentale 20](#_Toc483777194)

[5.1 Unitate aritmetico-logica (UAL1) 20](#_Toc483777195)

[4.2 Circuit de rotunjire 20](#_Toc483777196)

[4.3 Circuit de calcul al exponentului 21](#_Toc483777197)

[4.4 Circuit de deplasare a mantisei 21](#_Toc483777198)

[4.5 Multiplexor generic 21](#_Toc483777199)

[4.6 Calcul mantisa 22](#_Toc483777200)

[6 . Manual de utilizare 22](#_Toc483777201)

[6.1 Introducerea datelor 22](#_Toc483777202)

[7. Concluzii 22](#_Toc483777203)

[7.1 Avantaje si dezavantaje 22](#_Toc483777204)

[7.2 Aplicatii proiect si dezvoltari ulterioare 23](#_Toc483777205)

[8 .Bibliografie 23](#_Toc483777206)

[9. Anexe 23](#_Toc483777207)

[9.1 Anexa A 23](#_Toc483777208)

# 1.Rezumat

Operatiile pe numere in formatul cu virgula mobila au reprezentat intotdeauna o cerinta de baza pentru realizarea unor sisteme de calcul cat mai exacte . Proiectul prezinta o implementare a unui circuit de adunare si scadere a numerelor reprezentate in virgula mobila , incarcate in circuit prin intermediul modulului PMOD KYPD . Pentru implementarea si testarea circuitului a fost folosita placa de dezvoltare Nexys 4 DDR . Cu ajutorul mediului de dezvoltare Vivado a fost descris in mod structural circuitul de adunare a numerelor reprezentate in virgula mobila folosind limbajul VHDL . Decodificare semnalelor primite de la modulul PMOD KYPD s-a realizat cu ajutorul unor surse oferite de catre producator . Pentru afisarea rezultatelor experimentale s-au folosit afisoarele cu 7 segmente disponibile pe placa de dezvoltare Nexys 4 DDR. Astfel s-a obtinut un sistem care efectueaza adunarea si scaderea a doua numere introduse cu ajutorul tastaturii , rezultatul fiind afisat pe afisoarele cu 7 segmente prezente pe placa de dezvoltare.

# 2.Introducere

Nevoia de a avea o gama mai larga de valori rationale care totodata sa fie si foarte precise a dus la aparitia unei alte metode de reprezentare a numerelor rationale , si anume reprezentarea in virgula flotanta. Tendinta in domeniu este de a se putea efectua cat mai multe operatii in virgula mobila pe secunda .

Operatia de adunare se afla la baza tuturor sistemelor de calcul moderne . Operatia de scadere este similara cu cea de adunare , diferenta constand in faptul ca scazatorul este reprezentat cu semn schimbat. Virgula mobila se refera la faptul ca in interiorul numarului virgula care separa partea intreaga si partea fractionara se poate deplasa oriunde relativ la cifrele semnificative ale numarului . [[1](#_Bibliografie)]

Obiectivul principal al proiectului este implementarea opeartiilor de adunare si scadere a numerelor reprezentate in virgula mobila respectand standardul IEEE 754 pentru numere de 32 de biti (simpla precizie) .

Sectiunea **Fundamente teoretice** descrie pe larg standardul IEEE 754 si prezinta algoritmul de adunare al numerelor in virgula mobila , precum si o schema detaliata a circuitului implementat. Componentele circuitului sunt descrise mai in detaliu in capitolul **Proiectare si implementare** , componente care mai apoi sunt testate in capitolul **Rezultate experimentale .** Capitolul **Manual de utilizare**  ne ofera o intelegere mai buna asupra modului de functionare al sistemului.

# 3.Fundamente teoretice

# 3.1 Reprezentarea numerelor in virgula mobila. Standardul IEEE 754

Standardul IEEE 754 reglementeaza operatiile aritmetice in virgula mobila , fiind specificat formatul , operatiile , conversiile , reguli de rotunjire si conditii exceptionale

În general, un număr N se poate reprezenta în virgulă mobilă în forma urmatoare:

N = ± M · B±E

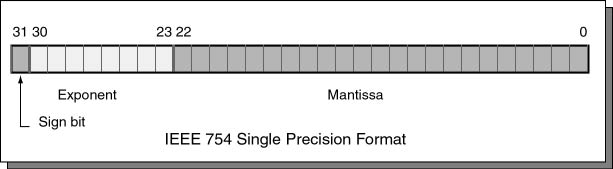
Un număr reprezentat în virgulă mobilă are următoarele componente:

* s semnul
* *c*  caracteristica
* *e*  exponentul
* m mantisa
* *d*  deplasarea

Exponentul nu este reprezentat de obicei in forma lui normala ci este folosit exponentul deplasat care este intotdeauna pozitiv

Pentru reprezentarea numerelor in precizie simpla ( 32 de biti ) sunt folositi 23 de biti pentru partea fractionara , 8 biti pentru exponent si 1 bit de semn . Reprezentarea poate fi observata in **figura 1**. Valoare numarului se determina astfel :

* daca 0 < exponent < 255 => numar = (-1)semn X c.m X 10e
* daca e = 0 si m = 0 => numar = 0
* daca e = 0 si m 0 sau e = 255 => eroare [[3](#_Bibliografie)]



**figura 1.** Reprezentare in virgula mobila (simpla precizie) [[2](#_Bibliografie)]

In cadrul reprezentarii numerelor in virgula mobila, partea fractionara nu poate avea o rezolutie la fel de mare ca reprezentarea numerelor intregi, doar cu cei 24 de biti, insa aceasta valoare este aproximata.

# 3.2 Valori speciale

Signed zero

In standardul IEEE 745 zero este considerat ca avand un semn, deci exista atat +0 cat si -0.Cele doua reprezentari se comporta normal in cadrul comparatiilor numerice, dar unele operatii dau rezultate diferite pentru +0 si -0. De exemplu, 1/(-0) returneaza -infinit, iar 1/(+0) +infinit.

Infinit

In reprezentarea in virgula mobila, valorile de +infinit si -infinit sunt reprezentate ca un numar cu bitii exponentului 1 si bitii mantisei 0. Bitul de semn face distinctia intre cele doua valori,

NaN(Not a Number)

NaN este o valoare speciala in standardul IEEE 754 care este returnata ca si rezultat in urma onr operatii invalide cum ar fi impartire la 0, infinit x 0, sau radacina patrata din numere negative.

Aceasta valoare este reprezentata ca un numar cu bitii exponentului 1 si bitul semnificativ al mantisei setat pentru a se face distinctia intre acesta si o valoare infinita.

Exista doua tipuri de NaN: quiet NaN(denota operatii nedeterminate) si signaling NaN(determina operatii invalide).

# 3.3 Adunarea si scaderea in virgula mobila

Pentru adunarea a doua numere in virgula mobila trebuie urmati urmatorii pasi : [[4](#_Bibliografie)]

1. se convertesc cele doua numere in notatia stiintifica normalizata , determinand semnul exponentul si mantisa
2. se aduc cei doi operanzi la acelasi exponent : se calculeaza dimensiune celor doi operanzi si se compara aceste dimensiuni . Mantisa operandului cu exponentul de dimensiuni mai mici este deplasata la dreapta cu un numar de pozitii care reprezinta diferenta in modul dintre dimensiunile exponentilor
3. se aduna mantisele obtinute la punctul anterior
4. se converteste numarul obtinut inpoi in formatul IEEE 754 pentru numere de 32 de biti

Sa prespunem ca avem urmatoarele numere reprezentate in standardul IEEE 745:

X = 0100 0010 0000 1111 0000 0000 0000 0000

Y = 0100 0001 1010 0100 0000 0000 0000 0000

1. cele doua numere in notatia stiintifica

pentru X:

S = 0

e = 1000 0100 = 132 – 127 = 5

deci notatia stiintifica pentru X : 1.0001111 x 25

pentru Y:

S = 0

e = 1000 0011 = 131 -127 = 4

deci notatia stiintifica pentru Y : 1.01001 x 24

1. deoarece Y are un exponent mai mic , mantisa acestuia este deplasata cu o pozitie

Y : 0.1010010 x 25

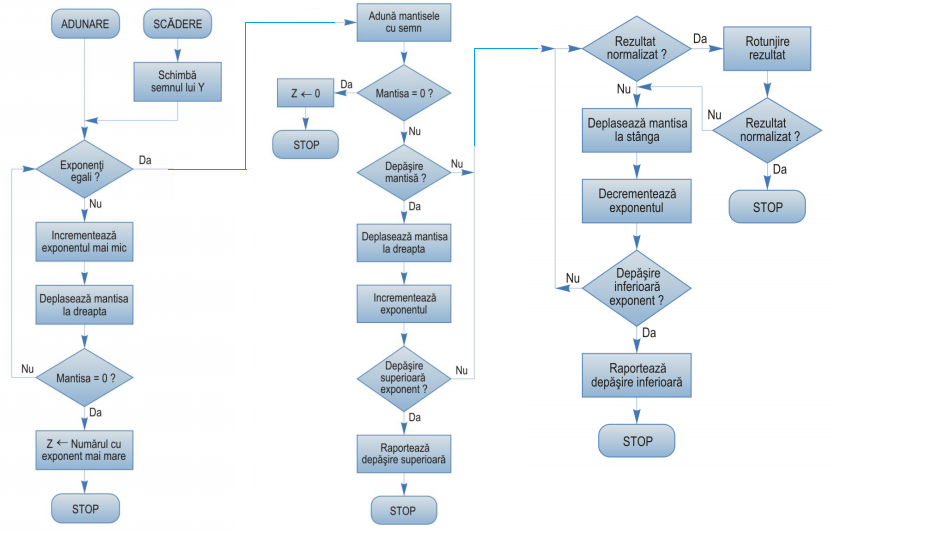
1. se aduna cele doua numere si obtinem X + Y = 1.1100001 x 25
2. rezultatul convertit inapoi in standardul IEEE 754 este

S = 0100 0010 0110 0001 0000 0000 0000 0000

# 3.4 Organigrama operatiei de adunare

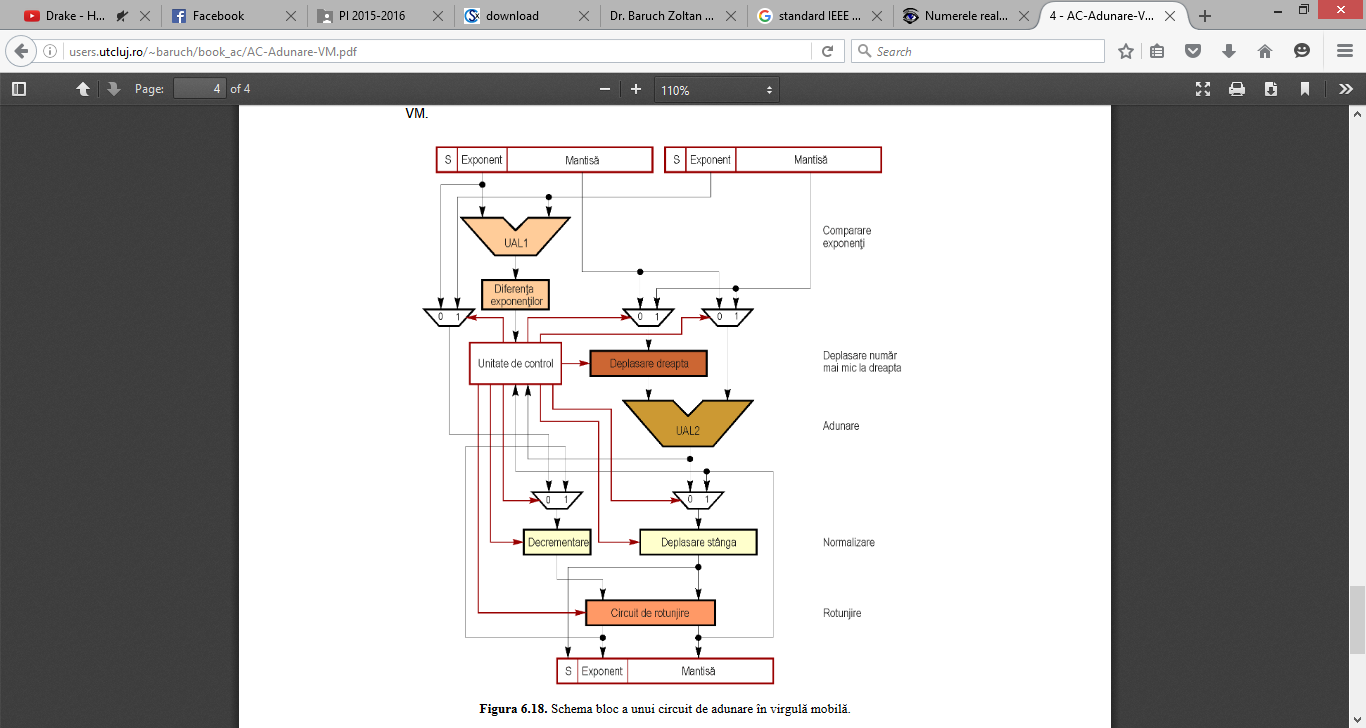
Organigrama de mai jos (**figura 2**) ne prezinta modul in care circuitul implementat realizeaza operatia de adunare sau scadere . In partea de initializare sunt comparate marimile exponentilor si este determinat numarul mai mare si este aliniata mantisa numarului cu exponentul mai mic.

Mantisele sunt apoi adunate sau scazute in functie de operatia selectata . Se efectueaza verificari pentru depasiri superioare sau inferioare care determina setarea unor flagu-uri de avertizare . Rezultatul obtinut trebuie apoi normalizat si rotunjit .



**figura 2 .** Organigrama operatiei de adunare sau scadere a numerelor in virgula mobila

# 2.5 Schema bloc a unitatii de adunare si scadere in virgula mobila



**figura 3 .**  Schema bloc a unui circuit de adunare in virgula mobila

Operanzii initiali vor fi memorati in registrele de forma [Semn|Exponent|Mantisa]. Exponentii operanzilor vor fi comparati de catre unitatea UAL1, iar diferenta lor va fi transmisa unitatii de control. Aceasta va da un semnal de control pentru selectia exponentului mai mic. Acest exponent va fi incrementat cu 1 de fiecare data cand mantisa corespunzatoare este deplasata la dreapta cu o pozitie.

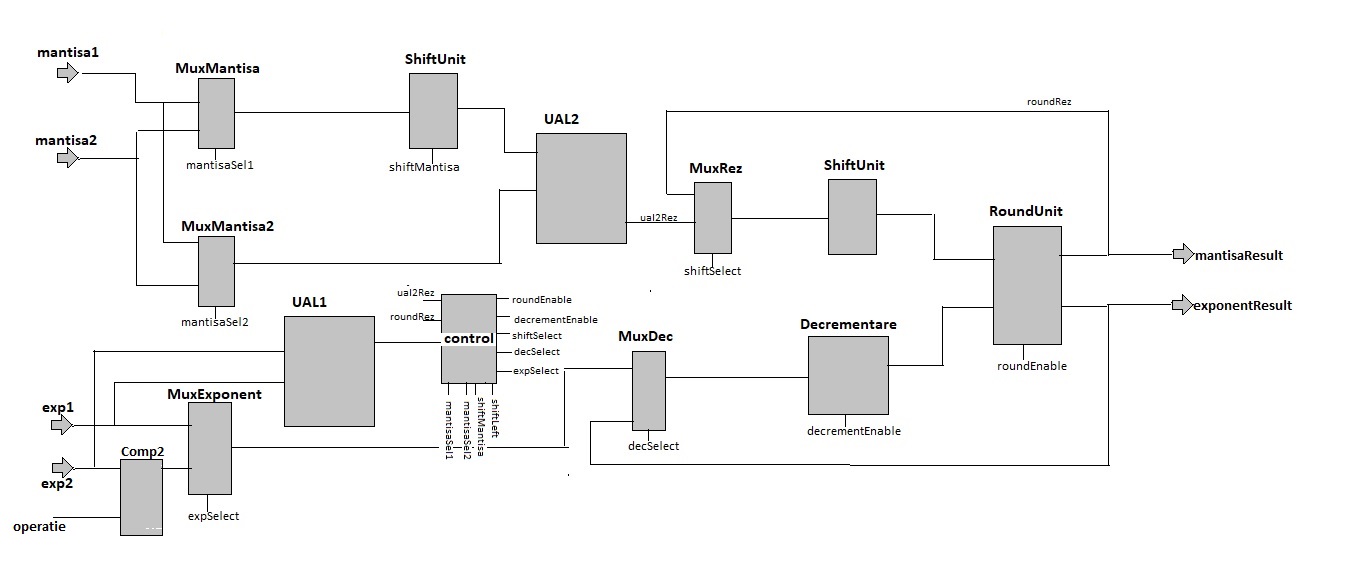
Selectia mantisei pentru deplasare se face tot in functie de diferenta exponentilor.

Unitatea UAL2 va realiza adunarea/scaderea mantiselor in momentul in care exponentii au devenit egali.

Urmatorul nivel este cel de normalizare al rezultatului. Aceasta se realizeaza prin deplasarea mantisei rezultat la stanga pana in momentul in care b.c.m.s. devine 1. Pentru deplasarile necesare vor fi folositi registri de deplasare. Dupa fiecare deplasare va avea loc si o decrementare a exponentului.

Ultima etapa a operatiei este rotunjirea rezultatului. Aceasta se realizeaza prin rotunjirea numarului la cel mai apropiat numar par prin schimbarea unlimului bit din ‘1’ in ‘0’ in cazul in care numarul este impar. Daca in urma operatiei de rotunjire rezultatul nu mai este normalizat acesta va trebui normalizat din nou.

# 3.6 Schema bloc a circuitului implementat



# 3.7 Introducerea datelor - modulul Pmod KYPD

Pentru o mai usoara furnizare a informatiilor la intrarile circuitului de adunare si scadere in virgula mobila datele vor fi introduse cu ajutorul moduluilui **Pmod KYPD** .

Modulu este o tastatura cu 16 butoane aranjate in format hexazecimal . Aceasta se conecteaza la unul dintre porturile **pmod** cu 12 pini ale placii Nexys 4 DDR folosita pentru implementarea proiectului .

Cei 12 pini ai modulului sunt descrisi in urmatorul tabel :

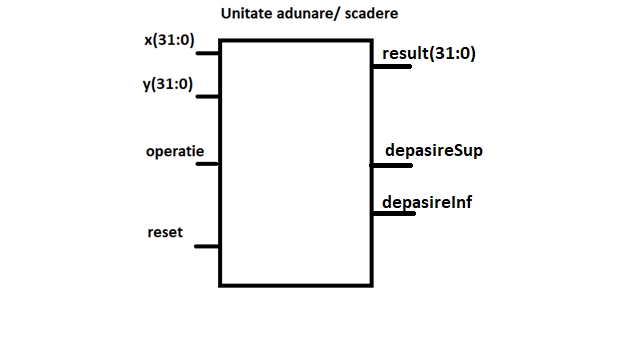
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pin | Semnal | Descriere | Pin | Semnal | Descriere |
| 1 | COL4 | Coloana 4 | 7 | ROW4 | Rând 4 |
| 2 | COL3 | Coloana 3 | 8 | ROW3 | Rând 3 |
| 3 | COL2 | Coloana 2 | 9 | ROW2 | Rând 2 |
| 4 | COL1 | Coloana 1 | 10 | ROW1 | Rând 1 |
| 5 | GND | Ground sursa de tensiune | 11 | GND | Ground sursa de tensiune |
| 6 | VCC | Sursa de tensiune(3.3V/5V) | 12 | VCC | Sursa de tensiune(3.3V/5V) |

# 4. Proiectare si implementare

Unitatea de adunare si scadere a numerelor in virgula mobila va primi ca intrare doua semnale pe 32 de biti reprezentand cele doua numere , un semnal pe un bit de selectare a operatiei ( adunare / scadere) si un semnal pe un bit de resetare . Circuitul va furniza ca iesire un semnal pe 32 de biti care reprezinta rezultatul operatiei efectuate precum si doua flag-uri pentru depasire superioara sau inferioara. Porturile circuitului pot fi observate in **figura 4**

Semnale :

* intrari
  + x : primul operand pe 32 de biti
  + y : al doilea operand pe 32 de biti
  + operatie : determina ce operatie se va efectua ( adunare / scadere )
  + reset : semnal care aduce circuitul in starea initiala
* iesiri
  + result : semnal pe 32 de biti care reprezinta rezultatul operatiei
  + depasireSup : flag care indica o depasire superioara
  + depasireInf : flag care indica o depasire inferioara



**figura 4 .** Porturi circuit de adunare in virgula mobila

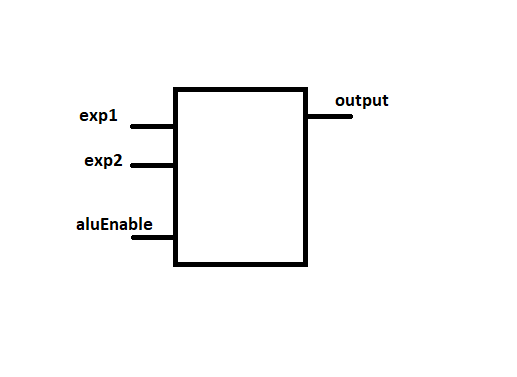
# 4.1 Descrierea componentelor

# 4.1.1 Unitate aritmetico – logica (UAL1)

Folosita pentru calcularea diferentei dintre cei doi exponenti pe 8 biti . De asemenea cu ajutorul unui semnal de iesire (compExp) se determina care dintre cei doi exponenti este mai mare . Porturile unitatii aritmetico logice pot fi observate in **figura 5.**

Semnale :

* intrari :
  + exponent1 : exponentul primului operand
  + exponent2 : exponentul celui de – al doilea operand
  + aluEnable : semnal de activare a circuitului
* iesiri
  + diferenta : rezultatul operatie de scadere al exponentilor celor doua numere primite ca semnale de intrare



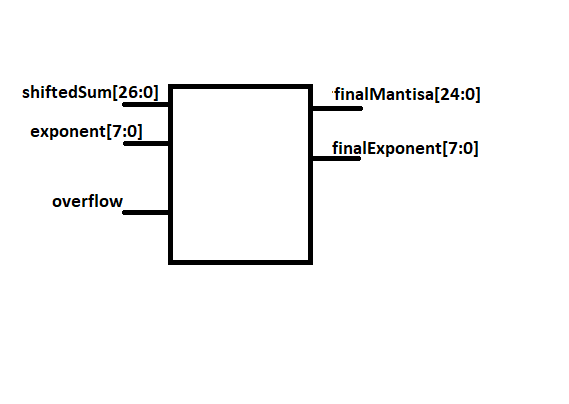
**figura 5 .** Unitate aritmetico logica – diferenta exponenti

# 3.1.2 Circuit de rotunjire

Este folosit pentru rotunjirea unui numar la cel mai apropiat numar par prin schimbarea unlimului bit din ‘1’ in ‘0’ in cazul in care numarul este impar . Circuitul primeste ca intrare un numar fara semn pe 31 de biti (**input)** , un semnal de **enable** si furnizeaza ca iesire rezultatul operatiei de rotunjire pe 31 de biti ( **output** ) .Porturile circuitului pot fi observate in **figura 6.**

Semnale :

* intrari :
  + shiftedSum : suma mantiselor shiftate
  + exponent : suma exponentilor
* iesiri :
  + finalMantisa : suma mantiselor rotunjita
  + finalExponent : exponent rotunjit



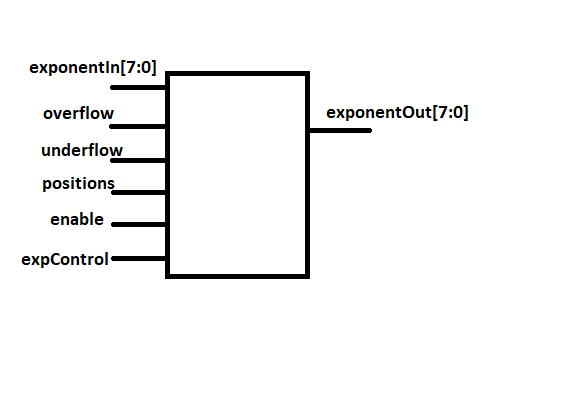
**figura 6 .** Porturile circuitului de rotunjire

## 3.1.3 Calcul exponent

Componenta este folosita pentru incrementarea eponentului la operatia de aliniere a mantiselor si la cea de normalizare(incrementare resprectiv decrementare). Pentru fiecare deplasare la dreapta a mantisei, exponentul este incrementat cu unu. Pentru fiecare deplasare la stanga a mantisei, exponentul se decrementeaza cu unu. Aceste operatii pot produce depasiri superioare sau inferioare, depasiri ce sunt semnalate prin semnalele de iesire **depasireSup**  si **depasireInf**. Circuitul primeste un semnal **exponentIn**  si genereaza o iesire **exponentOut** cu rezultatul operatiei de alese prin intrarea **operation**. Circuitul va face o incarcare a operandului prin semnalul **load**, iar operatia propriuzisa se realizeaza prin setarea semnalului de enable.

Semnale :

* intrari :
  + exponentIn : exponentul care trebuie calculat
  + overflow : flag depasire superioara
  + underflow : flag depasire inferioara
  + positions : numarul de pozitii cu care a fost deplasata mantisa
  + enable : activeaza circuitul
  + expControl : determina operatia efectuata asupra operandului
* iesiri :
  + exponentOut : rezultatul operatiei



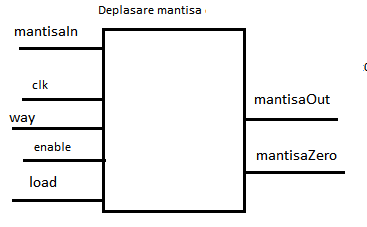
# 3.1.4 Deplasare mantisa

Circuitul de deplasare mantisa spre dreapta este folosit pentru alinierea punctelor zecimelale ale operanzilor inainte de a fi efectuata operatia selectata. Acest circuit este de asemenea folosit pentru deplasarea mantisei la stanga in cadrul operatiei de normalizare. Circuitul se comporta in principiu ca si cel de **calculExponent**, singura diferenta fiind faptul ca se realizeaza deplasari si nu adunari sau scaderi.

Circuitul primeste ca semnal de intrare (**mantisaIn**) mantisa asupra careia vom avea nevoie sa aplicam deplasarea. Semnalul **way** ne va da directia de deplasare(0 – stanga, 1 – dreapta).

Semnale :

* intrari :
  + mantisaIn : operandul care urmeaza sa fie deplasat
  + clk : semnal de ceas
  + way : determina directia deplasarii
  + enable : la activarea semnalului se efectueaza operatia de deplasare
  + load : la activarea semnalului se incarca operandul
* iesiri :
  + matisaOut : operandul dupa efectuarea operatiei de deplasare
  + mantisaZero : flag care indica faptul ca operandul este 0 dupa efectuarea operatiei de deplasare

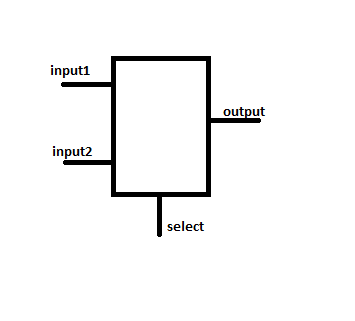


# 3.1.5 Multiplexor generic

Va fi nevoie de mai multe multiplexoare care primesc la intrari semnale de dimensiuni diferite . De aceea vom folosi un parametru generic care va indica dimensiunea intarilor si iesirilor multiplexorului . Astfel componenta va putea fi refolosita pentru toate multiplexoarele din schema bloc . Semnalele **input1**  si  **input2**  reprezinta intrarile circuitului iar semnalul **output** reprezinta intrarea selectata de multiplexor pentru a trece mai departe . Semnalul pe 1 bit **select** determina care intrare trece mai departe.

Semnale :

* intrari :
  + input1 : intrare de dimensiune variabila
  + input2 : intrare de dimensiune variabila
  + select : semnal care determina care intrare va fi trasmisa mai departe
* iesiri :
  + output : iesire determinata de semnalul **select**

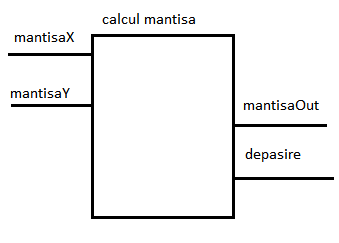


# 3.1.6 Calcul mantisa

Circuitul va aduna(scadea) mantisele in functie de operatie. Circuitul in sine face doar o adunare. In cadrul scaderii se presupune ca al doilea operand este in complement fata de 2. Iesirea **mantisaOut** va contine rezultatul operatiei asupra mantiselor. Semnalul **depasire** va fi 1 in cazul in care in urma operatiei va ramane un bit de carry in plus cu valoarea 1.

Semnale :

* intrari :
  + mantisaX : mantisa primului operand
  + mantisaY : mantisa celuilalt operand
* iesiri :
  + matisaOut : rezultatul calculului mantisei
  + depasire : flag care indica o depasire

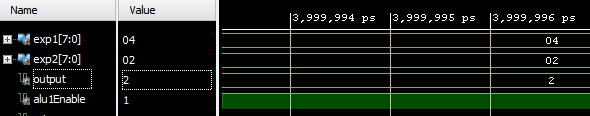


5. Rezutate experimentale

Pentru testarea componentelor unitatii de adunare si scadere in virgula mobila am elaborat module de simulare pentru fiecare dintre componentele unitatii .

# 5.1 Unitate aritmetico-logica (UAL1)

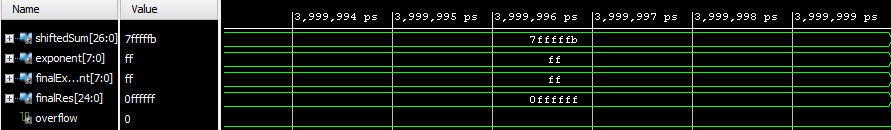
Functionarea corecta a acestei componente poate fi observata in **Figura**  care ne prezinta forma de unda a semnalelor de intrare si iesire. Operatia are loc doar daca semnalul aluEnable este activ.



**Figura .** Simulare UAL1 – calcul diferenta exponenti

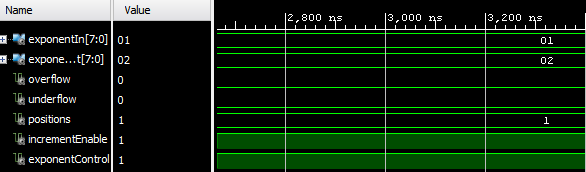
# 4.2 Circuit de rotunjire

Pentru circuitul de rotunjire am testat 3 cazuri : numar impar si semnal de enable activat , numar par si semnal de enble activat , numar impar si semnal de enable dezactivat . Rezultatul poate fi observat in urmatoarea imagine .



# 4.3 Circuit de calcul al exponentului

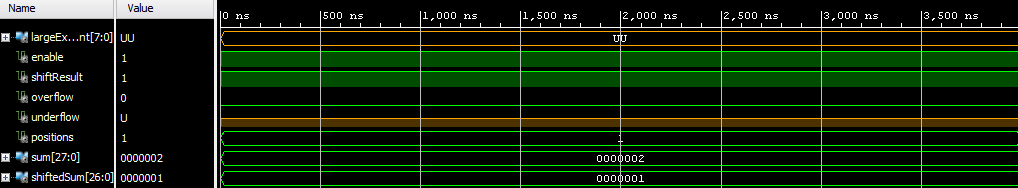
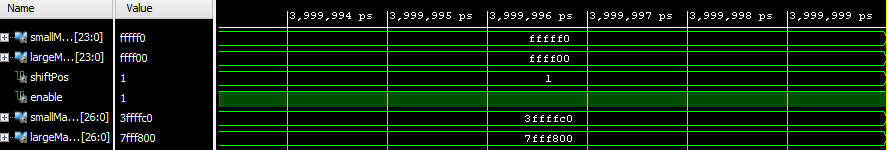
**Figura**  ne prezinta operatia de incrementare a exponentului , determinata de cele doua flaguri care indica depasiri si de semnalul de control al operatiei realizate asupra exponentului.



**Figura .** Simulare calcul exponent

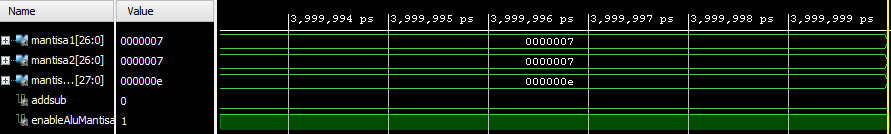
# 4.4 Circuit de deplasare a mantisei

Deplasarile cu o pozitie pot fi private ca o inmultire(deplasare stanga) sau o impartire(deplasare dreapta) a valorii initiale cu 2.



# 4.5 Calcul mantisa

Am folosit doua seturi de valori pentru simulare: cu sau fara generare de depasire.

.

# 6 . Manual de utilizare

# 6.1 Introducerea datelor

Cei doi operanzi sunt incarcati in circuit prin intermediul modulului PMOD KYPD conectat la placa Nexys 4 DDR pe portul JA . Este introdusa pe rand reprezentarea binara a operanzilor in format hexazecimal . Incarcarea celor doi operanzi are loc urmand urmatorii pasi :

1. se introduce pe rand cate o cifra hexazecimala din operand , aceasta incarcandu-se in registrul corespunzator operandului la apasarea butonului
2. se apasa butonul pentru a shimba registrul in care se memoreaza celalalt numar , astfel trecandu – se la incarcarea celui de-al doilea operand
3. se repeta pasul 1. pentru a incarca cel de-al doilea operand
4. se confirma incarcarea registrilor prin apasarea butonului care initializeaza si circuitul de adunare/scadere

6.2

# 7. Concluzii

Operatia de adunare a numerelor reprezentate in virgula mobila sporeste precizia masinilor de calcul , aceasta fiind o nevoie tot mai mare in lumea actuala . Proiectul acesta exemplifica o modalitate de efectuare a operatiilor de adunare si scadere a numerelor in virgula mobila reprezentate in formatul IEEE 754 , un format folosit la nivel international .

# 7.1 Avantaje si dezavantaje

Implementarea curenta ne prezinta un circuit de adunare a numerelor reprezentate in virgula mobila , care foloseste o unitate de control sincrona usor de inteles si implementat . Desi implementarea este usor de inteles , unul dintre dezavantajele circuitului este faptul ca unitatea de control trece dintr-o stare in alta doar la aparitia unui semnal de ceas . Din acest motiv circuitul devine mai lent , dar se elimina unele probleme care pot aparea din cauza intarzierilor fizice ale circuitelor combinationale.

# 7.2 Aplicatii proiect si dezvoltari ulterioare

Proiectul implementat este un circuit care apare in toate sistemele de calcul moderne care sunt folosite pentru realizarea unor calcule precise .

Printre dezvoltarile ulterioare ale acestui proiect putem mentiona :

* imbunatatirea vitezei de efectuare a operatiilor de adunare / scadere
* imbunatatirea controlului unitatii de calcul prin adaugarea unui automat de stari care sa permita o mai usoara introducere a operanzilor
* conversia numerelor afisate pe display din formatul hexazecimal in simpla precizie pentru o mai usoara interpretare a datelor introduse si primite de la circuit

# 8 .Bibliografie

[1] Informatii virgula mobila : <https://ro.wikipedia.org/wiki/Virgulă_mobilă>

[2]Figura reprezentare virgula mobila : <http://chortle.ccsu.edu/assemblytutorial/Chapter-30/IEEE754.jpg>

[3] https://en.wikipedia.org/wiki/Floating-point\_arithmetic

Informatii operatia de adunare :

[4]<http://andrei.clubcisco.ro/cursuri/1ii/cursuri/Aritmetica%20in%20virgula%20mobila..PDF>

[5]<https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/BinMath/addFloat.html>

# 9. Anexe

# 9.1 Anexa A

**Incrementare/decrementare:**

entity Inc\_Dec is

Port ( exponentIn : in STD\_LOGIC\_VECTOR (7 downto 0);

exponentOut : out STD\_LOGIC\_VECTOR (7 downto 0);

overflow : in STD\_LOGIC;

underflow : in STD\_LOGIC;

positions : in integer;

incrementEnable : in STD\_LOGIC;

exponentControl : in STD\_LOGIC);

end Inc\_Dec;

architecture Behavioral of Inc\_Dec is

begin

process (overflow , underflow , exponentControl , incrementEnable, positions,exponentIn)

begin

if incrementEnable = '1' then

if (overflow = '1') then

exponentOut <= "11111111";

elsif (exponentControl='0' and positions=27 and overflow = '0') then

exponentOut <= "00000000";

elsif (exponentControl='1' and overflow = '0') then

exponentOut <= std\_logic\_vector(unsigned(exponentIn) + to\_unsigned(positions,8));

elsif (exponentControl='0' and underflow = '0') then

exponentOut <= std\_logic\_vector(unsigned(exponentIn) - to\_unsigned(positions,8) + to\_unsigned(1,8));

else

exponentOut <= "00000000";

end if;

end if;

end process;

end Behavioral;

**Shiftare stanga/dreapta(rezultat):**

entity shift\_LR is

Port ( largeExponent : in STD\_LOGIC\_VECTOR (7 downto 0);

enable : in STD\_LOGIC;

shiftResult : in STD\_LOGIC;

overflow : in STD\_LOGIC;

underflow : in STD\_LOGIC;

positions : in integer;

sum : in STD\_LOGIC\_VECTOR (27 downto 0);

shiftedSum : out STD\_LOGIC\_VECTOR (26 downto 0));

end shift\_LR;

architecture Behavioral of shift\_LR is

begin

process(largeExponent , shiftResult , sum , underflow ,overflow , enable)

variable temp :STD\_LOGIC\_VECTOR (27 downto 0);

begin

if enable = '1' then

if (shiftResult='1' and overflow = '0') then

temp := std\_logic\_vector(unsigned(sum) srl positions);

elsif (shiftResult='0' and underflow = '0') then

temp := std\_logic\_vector(unsigned(sum) sll positions);

elsif (shiftResult='0' and underflow = '1') then

temp := std\_logic\_vector(unsigned(sum) sll (to\_integer(unsigned(largeExponent))));

end if;

if (overflow = '1') then

shiftedSum <= "000000000000000000000000000";

elsif (shiftResult='1') then

shiftedSum <= temp(26 downto 0);

else

shiftedSum <= temp(27 downto 1);

end if;

end if;

end process;

end Behavioral;

**Rotunjire:**

entity Round\_Unit is

Port ( shiftedSum : in STD\_LOGIC\_VECTOR (26 downto 0);

exponent : in STD\_LOGIC\_VECTOR (7 downto 0);

finalExponent : out STD\_LOGIC\_VECTOR (7 downto 0);

finalRes : out STD\_LOGIC\_VECTOR (24 downto 0);

overflow : in STD\_LOGIC

);

end Round\_Unit;

architecture Behavioral of Round\_Unit is

begin

process(shiftedSum , overflow , exponent)

variable temp : std\_logic\_vector (24 downto 0) := (others => '0');

begin

if (overflow = '1') then

temp := (others => '0');

elsif (shiftedSum(2)='0') then

temp := '0' & shiftedSum(26 downto 3);

elsif (shiftedSum(2)='1' and ((shiftedSum(1) or shiftedSum(0)) = '1')) then

temp := std\_logic\_vector(unsigned('0' & shiftedSum(26 downto 3)) + 1);

elsif (shiftedSum(2 downto 0)="100" and shiftedSum(3)='0') then

temp := '0' & shiftedSum(26 downto 3);

else

temp := std\_logic\_vector(unsigned('0' & shiftedSum(26 downto 3)) + 1);

end if;

finalExponent <= exponent;

finalRes <= temp;

end process;

end Behavioral;

**ALU exponent:**

entity ALU1 is

Port ( exp1 : in STD\_LOGIC\_VECTOR (7 downto 0);

exp2 : in STD\_LOGIC\_VECTOR (7 downto 0);

output : out integer;

alu1Enable : in STD\_LOGIC;

rst : in STD\_LOGIC;

NanFlag : out STD\_LOGIC;

ready : out STD\_LOGIC

);

end ALU1;

architecture Behavioral of ALU1 is

signal e1 , e2 , sumInteger: integer;

begin

process(rst , exp1 , exp2)

begin

if alu1Enable = '1' then

sumInteger <= to\_integer(unsigned(exp1)) - to\_integer(unsigned(exp2));

end if;

end process;

output <= sumInteger;

ready <= '1';

end Behavioral;

**Shiftare mantisa operand:**

entity shiftUnit is

Port ( smallMantisa : in STD\_LOGIC\_VECTOR (23 downto 0);

largeMantisa : in STD\_LOGIC\_VECTOR (23 downto 0);

shiftPos : in integer;

enable : in STD\_LOGIC;

smallMantisaShifted : out STD\_LOGIC\_VECTOR (26 downto 0);

largeMantisaShifted : out STD\_LOGIC\_VECTOR (26 downto 0));

end shiftUnit;

architecture Behavioral of shiftUnit is

signal round: integer;

signal shifter: std\_logic\_vector (26 downto 0);

signal mantisaShifted : STD\_LOGIC\_VECTOR (26 downto 0);

begin

RIGHT\_SHIFT: process(enable , largeMantisa , smallMantisa , shifter , mantisaShifted , shiftPos,round)

variable smallMantisaExt : STD\_LOGIC\_VECTOR (26 downto 0);

variable largeMantisaExt : STD\_LOGIC\_VECTOR (26 downto 0);

begin

if enable = '1' then

largeMantisaExt := largeMantisa & "000";

smallMantisaExt := smallMantisa & "000";

if(shiftPos < 27) then

--round <= to\_integer(unsigned(smallMantisaExt(pos downto 0)));

case shiftPos is

when 0 => round <= to\_integer(unsigned(smallMantisaExt(0 downto 0)));

when 1 => round <= to\_integer(unsigned(smallMantisaExt(1 downto 0)));

when 2 => round <= to\_integer(unsigned(smallMantisaExt(2 downto 0)));

when 3 => round <= to\_integer(unsigned(smallMantisaExt(3 downto 0)));

when 4 => round <= to\_integer(unsigned(smallMantisaExt(4 downto 0)));

when 5 => round <= to\_integer(unsigned(smallMantisaExt(5 downto 0)));

when 6 => round <= to\_integer(unsigned(smallMantisaExt(6 downto 0)));

when 7 => round <= to\_integer(unsigned(smallMantisaExt(7 downto 0)));

when 8 => round <= to\_integer(unsigned(smallMantisaExt(8 downto 0)));

when 9 => round <= to\_integer(unsigned(smallMantisaExt(9 downto 0)));

when 10 => round <= to\_integer(unsigned(smallMantisaExt(10 downto 0)));

when 11 => round <= to\_integer(unsigned(smallMantisaExt(11 downto 0)));

when 12 => round <= to\_integer(unsigned(smallMantisaExt(12 downto 0)));

when 13 => round <= to\_integer(unsigned(smallMantisaExt(13 downto 0)));

when 14 => round <= to\_integer(unsigned(smallMantisaExt(14 downto 0)));

when 15 => round <= to\_integer(unsigned(smallMantisaExt(15 downto 0)));

when 16 => round <= to\_integer(unsigned(smallMantisaExt(16 downto 0)));

when 17 => round <= to\_integer(unsigned(smallMantisaExt(17 downto 0)));

when 18 => round <= to\_integer(unsigned(smallMantisaExt(18 downto 0)));

when 19 => round <= to\_integer(unsigned(smallMantisaExt(19 downto 0)));

when 20 => round <= to\_integer(unsigned(smallMantisaExt(20 downto 0)));

when 21 => round <= to\_integer(unsigned(smallMantisaExt(21 downto 0)));

when 22 => round <= to\_integer(unsigned(smallMantisaExt(22 downto 0)));

when 23 => round <= to\_integer(unsigned(smallMantisaExt(23 downto 0)));

when 24 => round <= to\_integer(unsigned(smallMantisaExt(24 downto 0)));

when 25 => round <= to\_integer(unsigned(smallMantisaExt(25 downto 0)));

when 26 => round <= to\_integer(unsigned(smallMantisaExt(26 downto 0)));

when others => round <= to\_integer(unsigned(smallMantisaExt(0 downto 0)));

end case;

elsif(smallMantisaExt = "000000000000000000000000000") then

round <= 0;

else

round <= 1;

end if;

shifter <= std\_logic\_vector(unsigned(smallMantisaExt) srl shiftPos);

if round > 0 then

mantisaShifted <= (shifter(26 downto 1) & '1');

else

mantisaShifted <= (shifter(26 downto 1) & '0');

end if;

end if;

largeMantisaShifted <= largeMantisaExt;

smallMantisaShifted <= mantisaShifted;

end process;

end Behavioral;

**Unitate de control:**

entity controlUnit is

Port (

clk : in STD\_LOGIC;

rst : in STD\_LOGIC;

x : in STD\_LOGIC\_VECTOR (31 downto 0);

y : in STD\_LOGIC\_VECTOR (31 downto 0);

mantisaSum : STD\_LOGIC\_VECTOR (27 downto 0);

finalRes : in STD\_LOGIC\_VECTOR(24 downto 0);

exponentDifference : in integer;

ready : in STD\_LOGIC;

aluEnable : out STD\_LOGIC;

large\_exp : out STD\_LOGIC;

small\_num : out STD\_LOGIC;

large\_num : out STD\_LOGIC;

shift\_right\_pos : out integer;

addsub : out STD\_LOGIC;

sign : out STD\_LOGIC;

muxEnable : out STD\_LOGIC;

shiftEnable : out STD\_LOGIC;

enableAluMantisa : out STD\_LOGIC;

stop :out STD\_LOGIC;

shiftResult : out STD\_LOGIC;

exponentControl : out STD\_LOGIC;

shiftPos : out integer;

enableCheck : out STD\_LOGIC;

incrementEnable : out STD\_LOGIC;

shiftResultEnable :out STD\_LOGIC;

roundEnable : out STD\_LOGIC;

done : out STD\_LOGIC ;

opCheckEnable : out STD\_LOGIC

);

end controlUnit;

architecture Behavioral of controlUnit is

type adderState is ( addExp , ctrlSign , mux , shiftST , addMantisa , normalise , checkFlow , increment , shift\_LR , round , finalize);

signal currentState , nextState : adderState;

begin

STATE\_CHANGE:process(clk)

begin

if rst = '1' then

currentState <= addExp;

elsif rising\_edge(clk) then

currentState <= nextState;

end if;

end process;

CTRL : process(currentState,mantisaSum,x,y,finalRes,exponentDifference)

variable n : integer := 0;

begin

case currentState is

when addExp =>

opCheckEnable <= '0';

aluEnable <= '1';

nextState <= ctrlSign;

when ctrlSign =>

if exponentDifference > 0 then

large\_exp <= '1';

small\_num <= '0';

large\_num <= '1';

shift\_right\_pos <= exponentDifference;

sign <= x(31);

else

large\_exp <= '0';

small\_num <= '1';

large\_num <= '0';

shift\_right\_pos <= (0 - exponentDifference);

sign <= y(31);

end if;

if (x(31)=y(31)) then

addsub <= '0';

else

addsub <= '1';

end if;

nextState <= mux;

when mux =>

muxEnable <= '1';

nextState <= shiftST;

when shiftST =>

shiftEnable <= '1';

nextState <= addMantisa;

when addMantisa=>

enableAluMantisa <= '1';

nextState <= normalise;

when normalise =>

roundEnable <= '0';

if finalRes(24) = '1' then

stop <= '1';

else

stop <= '0';

end if;

if ((mantisaSum(27) = '1') or (finalRes(24) = '1')) then

shiftResult <= '1';

exponentControl <= '1';

shiftPos <= 1;

else

shiftResult <= '0';

exponentControl <= '0';

n := 0;

for i in 0 to 27 loop

if(mantisaSum(i) = '1')then

n := i;

else

n := n;

end if;

end loop;

shiftPos <= 27 - n;

end if;

nextState <= checkFlow;

when checkFlow =>

enableCheck <= '1';

nextState <= increment;

when increment =>

incrementEnable <= '1';

nextState <= shift\_LR;

when shift\_LR =>

shiftResultEnable <= '1';

nextState <= round;

when round =>

roundEnable <= '1';

if finalRes(24) = '1' then

nextState <= normalise;

else

nextState <= finalize;

end if;

when finalize =>

roundEnable <= '0';

stop <= '1';

done <= '1';

end case;

end process;

end Behavioral;

**Schimba semn:**

entity comp2 is

Port ( input : in STD\_LOGIC\_VECTOR (22 downto 0);

output : out STD\_LOGIC\_VECTOR (22 downto 0);

changeSign: out STD\_LOGIC;

sign : in STD\_LOGIC);

end comp2;

architecture Behavioral of comp2 is

constant ONE: UNSIGNED(output'RANGE) := (0 => '1', others => '0');

begin

process(sign, input)

begin

if sign = '1' then

output <= input;

changeSign <= '1';

else

output <= input;

changeSign <= '0';

end if;

end process;

end Behavioral;

**ALU mantisa:**

entity aluMantisa is

Port ( mantisa1 : in STD\_LOGIC\_VECTOR (26 downto 0);

mantisa2 : in STD\_LOGIC\_VECTOR (26 downto 0);

mantisaSum : out STD\_LOGIC\_VECTOR (27 downto 0);

addsub : in STD\_LOGIC;

enableAluMantisa : in STD\_LOGIC);

end aluMantisa;

architecture Behavioral of aluMantisa is

begin

process(addsub , mantisa1 , mantisa2 , enableAluMantisa)

begin

if enableAluMantisa = '1' then

if addsub = '0' then

mantisaSum <= std\_logic\_vector(unsigned('0'&mantisa1) + unsigned('0'&mantisa2));

else

mantisaSum <= std\_logic\_vector(unsigned('0'&mantisa1) - unsigned('0'&mantisa2));

end if;

end if ;

end process;

end Behavioral;

**Modul tastarura:**

entity PmodKYPD is

Port (

Clk : in STD\_LOGIC;

JA : inout STD\_LOGIC\_VECTOR (7 downto 0);

Rst: in std\_logic;

memDebounce: in std\_logic;

Xfinal: out std\_logic\_vector(31 downto 0);

Yfinal: out std\_logic\_vector(31 downto 0);

ready1: out std\_logic;

ready2: out std\_logic;

done: out std\_logic

);

end PmodKYPD;

architecture Behavioral of PmodKYPD is

component Decoder is

Port (

clk : in STD\_LOGIC;

Row : in STD\_LOGIC\_VECTOR (3 downto 0);

Col : out STD\_LOGIC\_VECTOR (3 downto 0);

DecodeOut : out STD\_LOGIC\_VECTOR (3 downto 0)

);

end component;

signal Decode: STD\_LOGIC\_VECTOR (3 downto 0);

signal afisare: std\_logic\_vector(31 downto 0);

signal Xout: std\_logic\_vector(31 downto 0) := (others => '0');

signal Yout: std\_logic\_vector(31 downto 0) := (others => '0');

signal count: integer := 0;

begin

C0: Decoder port map (clk=>clk, Row =>JA(7 downto 4), Col=>JA(3 downto 0), DecodeOut=> Decode);

SCHIMBA: process(Clk)

begin

if rising\_edge(Clk) then

if Rst = '1' then

afisare <= (others => '0');

count <= 0;

else

if memDebounce = '1' then

count <= count + 1;

if count <= 8 then

afisare <= afisare(27 downto 0) & x"0";

afisare <= afisare + Decode;

Xout <= afisare;

else

afisare <= afisare(27 downto 0) & x"0";

afisare <= afisare + Decode;

Yout <= afisare;

end if;

end if;

end if;

end if;

end process;

Xfinal <= Xout when count > 8;

Yfinal <= Yout when count > 8;

ready1 <= '1' when count <= 8 else '0';

ready2 <= '1' when count > 8 else '0';

done <= '1' when count > 16;

end Behavioral;