**Unitate de calcul în virgulă mobilă: înmulțirea**

Pap Dorottya

Șimon Bianca Andreea

Grupa 30232

Profesor îndrumător: Mocan Cristi Data: 8.01.2021

**Cuprins**

[**1. Rezumat** 4](#_Toc61013537)

[**2. Introducere** 4](#_Toc61013538)

[**3. Fundamentare teoretică** 5](#_Toc61013539)

[3.1 Limbajul VHDL 5](#_Toc61013540)

[3.2 Circuitul FPGA 5](#_Toc61013541)

[3.3 Reprezentarea numerelor în virgulă mobilă 6](#_Toc61013542)

[3.4 Înmulțirea numerelor în virgulă mobilă 8](#_Toc61013543)

[3.5 Soluția finală și obiectivul proiectului 8](#_Toc61013544)

[3.5.1 Sumatorul cu anticiparea transportului 9](#_Toc61013545)

[3.5.2 Înmulțirea matricială 12](#_Toc61013546)

[**4. Proiectare și implementare** 15](#_Toc61013547)

[4.1 Unitatea de însumare 16](#_Toc61013548)

[4.2 Unitatea de înmulțire 17](#_Toc61013549)

[4.3 Registru de n biți cu resetare sincronă 17](#_Toc61013550)

[4.4 Comparator 18](#_Toc61013551)

[4.5 Normalizare și rotunjire 18](#_Toc61013552)

[4.6 Unitatea de control 19](#_Toc61013553)

[4.7 Modulul principal 20](#_Toc61013554)

[**5. Rezultate experimentale** 20](#_Toc61013555)

[5.1 Unitate de însumare 20](#_Toc61013556)

[5.2 Unitatea de înmulțire matricială 20](#_Toc61013557)

[5.3 Registru de n biți cu resetare sincronă 21](#_Toc61013558)

[5.4 Comparator 21](#_Toc61013559)

[5.5 Normalizare și rotunjire 21](#_Toc61013560)

[5.6 Unitatea de control 22](#_Toc61013561)

[5.7 Modulul principal 22](#_Toc61013562)

[**6. Manual de utilizare** 23](#_Toc61013563)

[**7. Concluzii** 24](#_Toc61013564)

[**Anexe** 25](#_Toc61013565)

[1. Comparator 25](#_Toc61013566)

[2. Registru de n biți cu resetare sincronă 25](#_Toc61013567)

[3. Înmulțitor 26](#_Toc61013568)

[4. Normalizare 27](#_Toc61013569)

[4. Sumator elementar 29](#_Toc61013570)

[5. Sumator pe 2 biți 29](#_Toc61013571)

[6. Sumator pe 10 biți 30](#_Toc61013572)

[7. Unitatea de control 31](#_Toc61013573)

[8. Sumator 33](#_Toc61013574)

[9. Prelucrare rezultat 34](#_Toc61013575)

[10. Modulul principal 35](#_Toc61013576)

[**Bibliografie** 38](#_Toc61013577)

# **1. Rezumat**

Proiectul ales de noi, și anume : “Unitatea de calcul în virgulă mobilă: înmulțirea”, are ca scop realizarea operției de înmulțire a două numere în virgulă mobilă. În continuare se va prezenta modul în care s-a realizat acest proiect, rezultatele experimentale care demonstrează corectitudinea funcționalităților implementate și utilizarea proiectului propriu-zis.

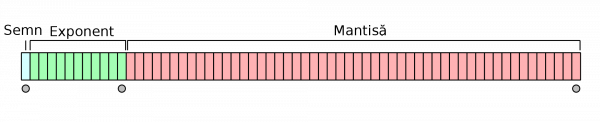
# **2. Introducere**

Proiectul are ca scop dezvoltarea unei unităţi de înmulţire a numerelor reprezentate în virgulă mobilă conform standardului IEEE 754 (32 biţi). Acest standard prevede o metodă de calcul cu numere în virgulă mobilă, care va produce acelaşi rezultat dacă prelucrarea se face în hardware, software sau o combinaţie a acestora.

Majoritatea unităţilor centrale de procesare lucrează cu două tipuri de numere: în virgulă fixă şi în virgulă mobile [1]. Acest lucru este datorat faptului că nu toate informațiiile necesare UCP-ului pentru a-și realiza operațiile specifice sunt numere întregi. Pentru asigurarea portabilității programelor între calculatoare diferite, se datorează faptului că reprezentarea internă a datelor, în calculator respectă un standard(IEEE 754) utilizat de marea majoritate a unităților de calcul și a coprocesoarelor matematice.

Multe aplicații necesită numere care nu sunt întregi. Există mai multe posibilități pentru reprezentarea acestor numere. Una dintre ele este reprezentarea în *virgulă fixă*. În acest caz, se poate utiliza aritmetica pentru numere întregi, plasând apoi virgula binară în poziția predefinită, de exemplu, după bitul de semn. În cazul reprezentării numerelor in acest format, deși virgula nu este reprezentată fizic în calculator, poziția virgulei binare, care este stabilită la proiectare, nu mai poate fi schimbată [2]. Astfel, este necesară transformarea tuturor numerelor în acest format, lucru ce necesită o serie de operații de scalare sau deplasare, atașând numerelor factori de scală.Evidența acestora trebuie realizată prin program, ceea ce mărește timpul de calcul.

Una dintre soluții este utilizarea unei tehnici de scalare automate, cunoscută sub numele de reprezentare în virgulă mobilă. Numerele în virgulă mobilă sunt numerele care sunt exprimate în notaţiile ştiinţifice, în care un număr este reprezentat ca un număr zecimal înmulţit cu o putere a lui 10. Notaţia ştiinţifică este un mod compact de a exprima numere foarte mari sau foarte mici. Acest lucru este important pentru a reprezenta grafice şi pentru munca ştiinţifică, însă aritmetica în virgulă mobilă este mai complexă şi poate dura mai mult [1].

În acest caz, factorul de scală devine o parte a cuvântului din calculator, poziția virgulei variind pentru fiecare număr în mod automat. Un număr reprezentat în virgulă mobilă se poate memora într-un cuvânt cu trei câmpuri: semnul, mantisa și exponentul (figura 2.1.).

*Figura 2.1. Reprezentarea unui număr în virgulă mobilă*

Scopul nostru principal este de a implementa algoritmul de înmulțire în virgula mobilă. De asemenea, trebuie să reprezentăm numerele conform standardului IEEE 754 pe 32 de biți, iar implementarea algoritmului, se realizează operația de rotunjire, pentru a obține rezultate cu o precizie cât mai exactă. Pentru soluționarea problemei, vom lua în considerare și unele cazuri excepționale ce pot apărea în cadrul acestei operații: depășirea inferioară, depășirea superioară și eventuale rezultate inexacte.

# **3. Fundamentare teoretică**

## 3.1 Limbajul VHDL

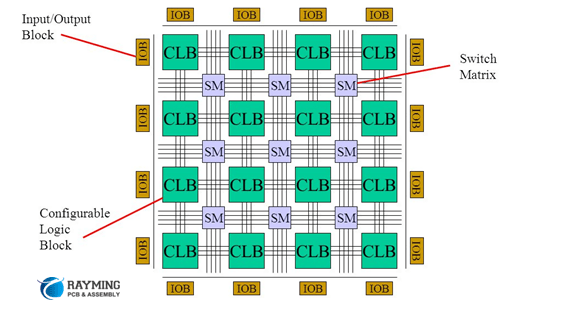
VHDL (VHSIC-HDL, Very High Speed Integrated Circuit Hardware Description Language) este un limbaj standard pentru descrierea hardware a circuitelor integrate de foarte mare viteză. Un sistem numeric poate fi reprezentat în diferite nivele de abstractizare. [3]

Cel mai înalt nivel de abstractizare ale acestui limbaj este nivelul de descriere al comportării(funcționării) numit în engleză behavioral. La acest nivel un sistem este descris prin ceea ce face, și nu prin componentele sale și prin conxexiunile dintre acestea. O astfel de descriere specifică relațiile dintre semnalele de intrare și ieșire.

Un alt nivel de abstractizare este nivelul structural care descrie un sistem ca o colecție de porți și componente care sunt conectate între ele pentru a realiza funția dorită. Aceasta este o reprezentare care se apropie mai mult de realizarea fizică a sistemului, fiind comparată cu o schemă de porți logice conectate între ele.

## 3.2 Circuitul FPGA

Un FPGA(Field Programmable Gate Array) este un circuit integrat care poate fi programat de către utilizator. Configurarea circuitului se face, în general, cu ajutorul unui limbaj de descriere hardware HDL, dezvoltându-se recent și compilatoare care traduc instrucțiuni din limbajul C în limbajul HDL. [4]

Un FPGA conține mai multe funcții versatile, interconexiuni configurabile, interfețe de I/E adaptabile conform specificațiilor utilizatorului (figura 3.1.).

*Figura 3.1. Structura FPGA*

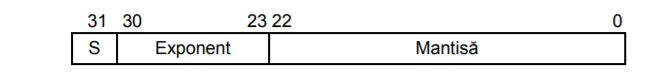
## 3.3 Reprezentarea numerelor în virgulă mobilă

Virgula mobilă este unul din sistemele care sunt folosite pentru a reprezenta numerele raționale ca șiruri de biți, separând partea întreagă de cea fracționară printr-o virgulă care se poate deplasa oriunde relativ la cifrele semnificative ale numărului. Poziția virgulei este indicată separat în cadrul reprezentării interne.

Un număr reprezentat în virgula mobilă are două componente: mantisa și exponentul.

N =

Mantisa(M) indică valoarea exactă a numărului într-un anumit domeniu, fiind reprezentată de obicei ca un număr fracționar cu semn. B este baza exponentului, care fiind implicită nu se va memora, iar E reprezintă exponentul, indicând ordinul de mărime a numărului.

Conform cerințelor, numerele vor fi reprezentate pe 32 de biți, iar reprezentarea internă al acestora se va face astfel: un bit pentru semn(0 pentru numere pozitive/1 pentru numere negative), 8 biți pentru exponent, iar restul de 23 de biți pentru mantisă (figura 3.2.).

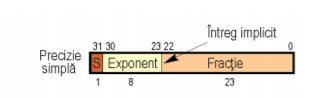
*Figura 3.2. Reprezentarea numerelor în precizie simplă*

Câmpul specific exponentului în general conține o valoare numită caracteristică, care se obține adunând la exponent un deplasament, astfel încât să avem tot timpul o valoare pozitivă, fără a mai fi necesară creerea unui câmp separat pentru semnul exponentului.

C = E + deplasament

Avantajele utilizării exponentului deplasat sunt: simplificarea operațiilor executat cu exponentul datorită lipsei exponenților negativi, modele variate de reprezentare al numărului 0 și posibilitatea de a compara numerele în virgulă mobilă, acestea fiind ordonate în același fel ca și numere întregi.

Un număr în virgulă mobilă este normalizat dacă bitul cel mai semnificativ al mantisei este 1. Acest bit nu este deobicei memorat, fiind un bit ascuns la dreapta virgulei binare.

 Standardul IEEE 754 definește următoarele formate/precizii: precizie simplă, precizie simplă extinsă, precizie dublă și dublă extinsă. În continuare vom reprezenta formatul cu precizie simplă (figura 3.3.). [5]

*Figura 3.3. Reprezentarea numerelor în precizie simplă*

S reprezintă semnul numărului(0 sau1). Exponentul este format din 8 biți, deplasamentul pentru acest format fiind de 127 (7Fh). Reprezentarea mantisei(fracției/significand) constă dintr-un bit cu valoarea 1(partea întreagă), virgula binară și biții fracției.

Standardul IEEE 754 definește 5 tipuri de excepții: depășire inferioară, depășire superioară, împărțire la 0, rezultate inexacte și rezultate invalide.

În cazul depășirii inferioare, standardul permite utilizarea numerelor care nu sunt normalizate, numite numere denormalizate. Aceste numere sunt generate printr-o tehnică care constă în deplasarea mantisei la dreapta și incrementarea exponentului până când acesta va ajunge la valoarea minimă admisă.

În cazul unei depășiri superioare, se reprezintă valoarea infinit punând la exponent valoarea maximă și mantisa egală cu 0. Sunt posibile două reprezentări pentru infinit, în funcție de bitul de semn și anume: .

În cazul operațiilor nedefinite s-a prevăzut un format special NaN (Not a Number). Exponentul are valoarea maximă posibilă, iar mantisa este diferită de 0.

Excepția de rezultat inexact apare atunci când rezultatul trebuie rotunjit, această problemă se poate rezolva în 4 moduri diferite, și anume: rotunjirea spre 0, rotunjire spre , rotunjire spre și rotunjirea spre cel mai apropiat număr reprezentabil.

Realizarea cu acuratețe a operațiilor aritmetice în virgulă mobilă necesită utilizarea unor cifre suplimentare. Dintre aceste cifre, cifrele de gardă g sunt plasate la dreapta cifrelor de precizie p ale mantisei pentru a evita pierderea acestor cifre. Aceste cifre pot fi deplasate la stânga în pozițiile biților de precizie prin operația de normalizare. După această operație se verifică valoarea cifrelor aflate la dreapta biților de precizie: dacă valorile acestora sunt diferite de 0 este necesară rotunjirea numărului. Cifra de rotunjire r este plasată la dreapta cifrelor de gardă, astfel încât după deplasarea prin normalizare a cifrelor de gardă, rezultatul este rotunjit conform valorii acestei cifre.

Doi biți suplimentari sunt suficienți pentru rotunjirea în primele moduri prezentate mai sus. Rotunjirea spre cel mai apropriat număr par necesită un bit suplimentar, bit-ul sticky, ce va fi setat când există biți diferiți de 0 la dreapta bitului de rotunjire. Acest tip de rotunjire este prevăzut pentru situațiile în care numărul real se află la jumătatea intervalului dintre două reprezentări în virgulă mobilă.

## 3.4 Înmulțirea numerelor în virgulă mobilă

Așa cum am prezentat mai sus, numerele în virgulă mobilă au următoarea structură:

Ecuația de înmulțire pentru aceste două numere se va determina astfel:

Așa cum se obervă în ecuație, vor fi necesare atât operații de înmulțire, cât și de adunare. În plus rezultatul trebuie normalizat, rotunjit și în final, trebuie stabilit semnul.

În primul rând, algoritmul verifică pe rând operanzii dacă sunt nuli. În caz afirmativ, algoritmul se finalizează, returnând valoarea 0. De menționat este că nu pot fi ambele 0.

La următorul pas se adună exponenții celor două numere. Datorită faptului că exponenții sunt deplasați, va rezulta un deplasament dublu, din care va trebui să scădem unul dintre cei doi.

La următorii doi pași se verifică dacă avem depășiri superioare/inferoare la exponent, în caz afirmativ se încheie execuția programului, semnalizând acest lucru.

Dacă exponentul se încadrează în limitele admise, se înmulțesc mantisele, se normalizează rezultatul, iar apoi se retestează din nou dacă exponentul se află în domeniul permis. Dacă nu se află, se va raporta depășirea și se va opri execuția.

În final, se rotunjește rezultatul și se setează semnul.

## 3.5 Soluția finală și obiectivul proiectului

În acest proiect este descrisă implementarea unui înmulțitor în virgulă mobilă, conform standardelor IEEE 754. Pentru o soluție optimă, trebuie să luăm în considerare mai multe aspecte care apar în rezolvarea problemei: dimensiunea și numărul redus al operanzilor (2 operanzi), dorința obținerii unui timp de execuție redus, precum și utilizarea cât mai eficientă a resurselor.

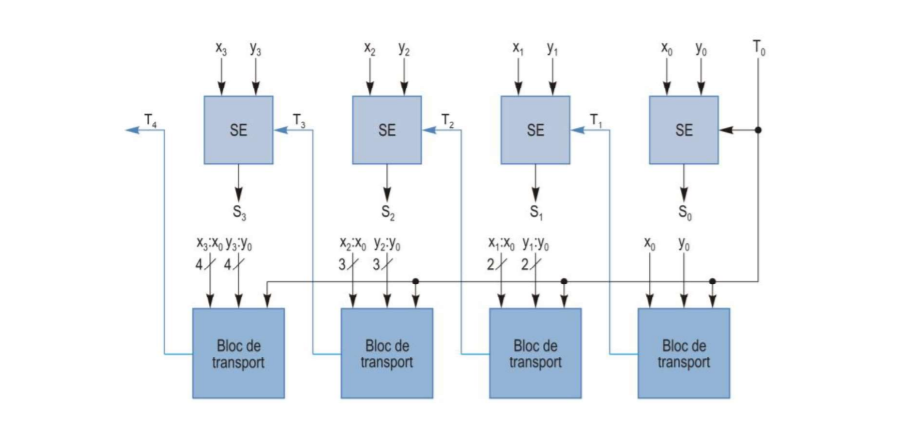
Având în vedere faptul că operațiile de adunare și înmulțire a două numere se pot realiza în diferite moduri, vom argumenta și prezenta în continuare algoritmul ales pentru fiecare tip de operație.

Pentru operația de adunare ale celor doi exponenți avem următoarele opțiuni: un sumator elementar, un sumator cu propagarea succesivă a transportului, un sumator cu anticiparea transportului sau un sumator cu salvarea transportului. Pentru cerințele actuale, am ales un sumator cu anticiparea transportului, care este foarte eficient datorită creșterii vitezei de răspuns, prin reducerea timpului necesar formării transportului. Sumatorul cu selecția transportului utilizează resurse în mod ineficient, iar cel cu salvarea transportului este potrivit în cazul adunării mai multor numere deodată, acestea fiind motivele pentru alegerea făcută.

Dintre tipurile de înmulțire pe care le avem la dispoziție, și anume: înmulțirea prin deplasare și adunare, înmulțirea matricială și arborele Wallace, am ales circuitele de înmulțire matricială. Acest tip de circuit se deosebește de cele secvențiale prin utilizarea unei logici combinaționale suplimentare ce facilitează calculul produsului într-un singur pas și se remarcă prin rapiditatea execuției.

### 3.5.1 Sumatorul cu anticiparea transportului

În sumatorul cu transport anticipat (Carry Look-Ahead), prezentat în figura 3.4. , fiecare adunare pe bit elimină dependenţa de semnalul transportului generat anterior şi impune în schimb utilizarea valorilor celor doi operanzi de intrare. El funcționează prin generarea a două noi semnale (P și G) pentru fiecare rang binar în funcție de starea intrărilor [6].

Blocurile de transport de pe fiecare etaj conțin o logică combinațională ce va calcula transportul de intrare pentru acel etaj, în funcție de transportul inițial și toți biții din etajele precedente (nu se așteaptă propagarea transporturilor de pe un etaj pe altul).

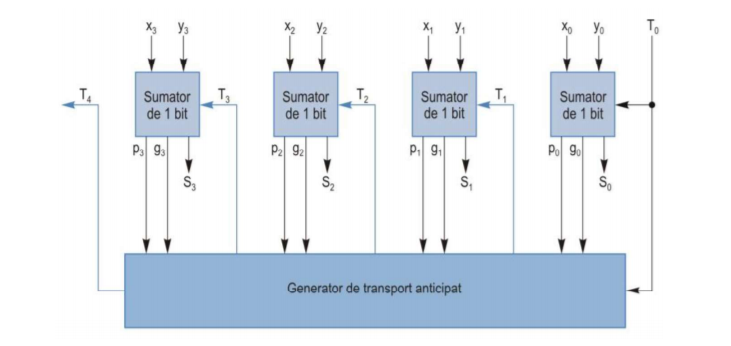
*Figura 3.4. Schema bloc a unui sumator de patru biți cu anticiparea transportului*

Expresiile booleene ale fiecărui bloc de transport pot fi definite pe baza ecuației transportului de ieșire al unui sumator elementar:

Se introduc două funcții noi, p și g, pentru simplificarea fiecărui semnal de transport. Funcția g se referă la generarea transportului de către un bloc de transport, iar funcția p se referă la propagarea intrării de transport la ieșirea de transport al blocului. Ele sunt definite astfel:

Funcția de generare este denumită astfel, deoarece etajul i va produce un semnal de transport cu valoarea logică ‘1’ () dacă ambii operanzi vor fi ‘1’ (), independent de . Alternativ, propagarea se referă la prezența unui transport () dacă având ca și intrare , unul din cei doi operanzi este setat (). Noua expresie devine:

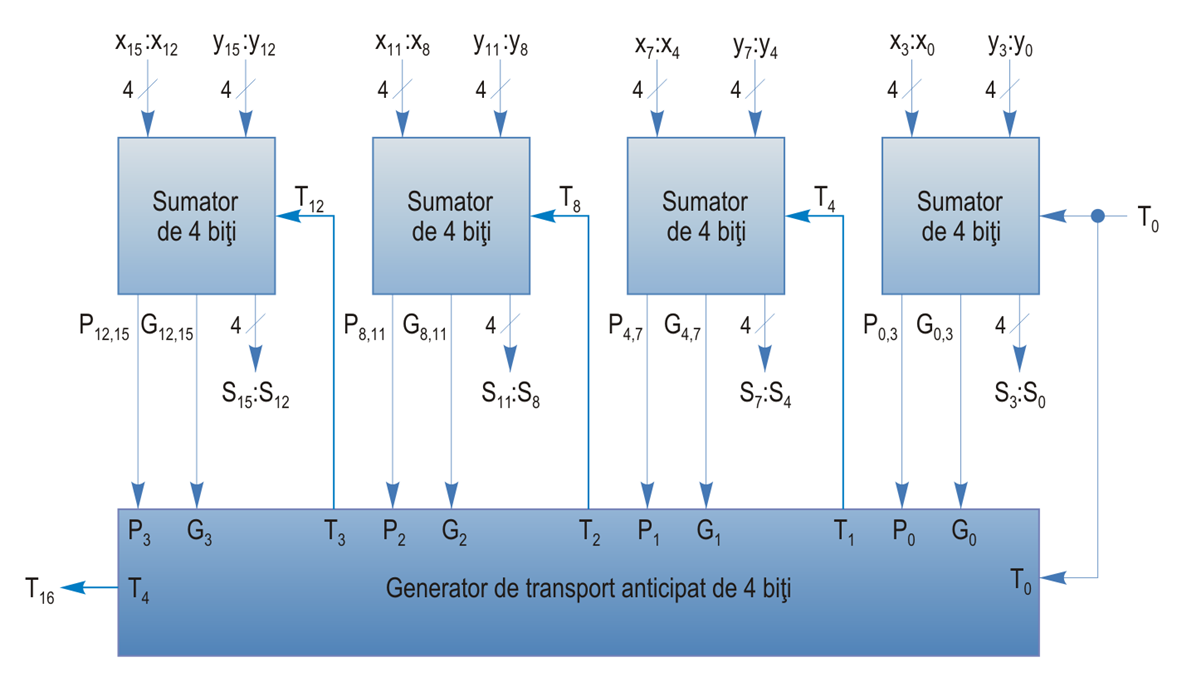
Utilizând aceste notații, ecuațiile boolene pentru semnalele de transport ale sumatorului cu anticiparea transportului de patru biti se pot exprima astfel:

În figura 3.5. este reprezentat sumatorul implementat pe baza ecuațiilor definite. Fiecare sumator de un bit produce semnalele de propagare a transportului pe bit p si de generare a transportului pe bit g, în locul ieșirilor de transport. Generatorul convertește seturile de semnale p și g în intrările de transport necesare pentru sumatoare.

*Figura 3.5. Schema bloc a unui sumator de patru biți cu anticiparea transportului care utilizează un generator de transport anticipat*

Pentru a reduce complexitatea circuitului, se poate limita numărul de intrări și de porți existente, ce va duce la adăugarea unui nou nivel logic suplimentar. Se definesc două funcții noi de generare a transportului pe grup , și de propagare a transportului pe grup , pentru blocul bițiilor de la i până la k, ecuațiile devenind:

Astfel semnalul de transport devine:

Schema bloc a unui astfel de sumator devine:

*Figura 3.6. Schema bloc a unui sumator de 16 biți format din sumatoare de patru biți conectate prin semnale de transport generare anticipat*

Intrarea de transport , are o ecuație similară cu următoarea:

unde:

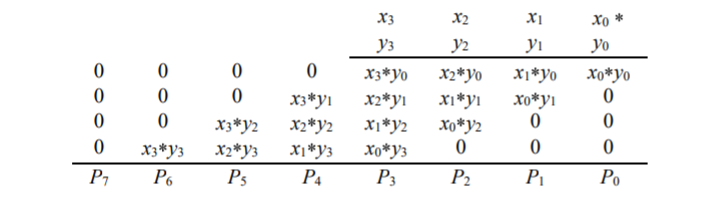
Ecuațiile intrărilor și sunt similare semnalelor și .

### 3.5.2 Înmulțirea matricială

Algoritmul de înmulțire matricială conține o logică suplimentară față de circuitele combinaționale de înmulțire, care permite calcularea produsului într-o singură etapă. Acest circuit este format dintr-o matrice de elemente combinaționale simple, fiecare din acestea implementând o operație de adunare și de deplasare pentru un bit sau pentru un număr redus de biți.

Presupunând că și sunt două numere întregi fără semn, produsul P se poate exprima astfel:

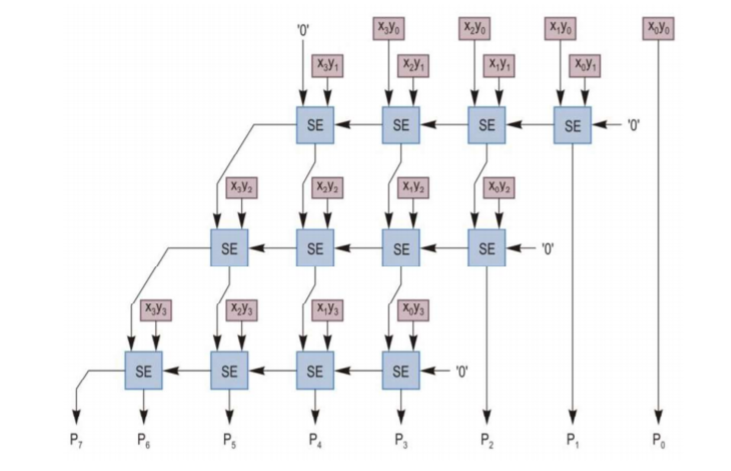
Toți cei termeni produs de un bit se pot calcula cu ajutorul unei porți ȘI cu două intrări. Astfel o matrice de porți ȘI cu două intrări poate calcula toți termenii simultan. Acești termini sunt însumați cu ajutorul unei matrice de n(n-1) sumatoare elementare, dintre care un număr de n sumatoare elementare pot fi înlocuite prin semisumatoare elemementare. Deplasările implicate de factorii și din ecuația de mai sus, sunt implementate prin deplasarea spațială a sumatoarelor pe direcția x și y.

Pentru înmulțirea numerelor pe 4 biți, înmulțirea se efectuează ca și în figura 3.7.

*Figura 3.7. Înmulțirea numerelor de câte 4 biți*

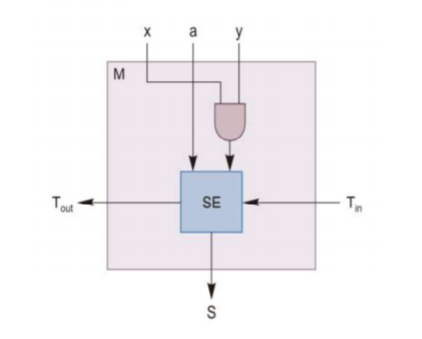
Biții produsului final au urmăroarele expresii:

În figura 3.8. este reprezentată schema pentru generarea și adunarea produselor parțiale. Dreptunghiurile notate cu reprezintă termeni produs de câte un bit, generați cu porți ȘI. Semnalele de transport din fiecare rând sunt conectate ca și la un sumator cu propagarea succesivă a transportului. În primul rând, sumatoarele elementare adună primele două produse parțiale, iar în următoarele rânduri, sumatoarele elementare adună câte un produs parțial la suma rezultată.



*Figura 3.8. Matrice de sumatoare elementare pentru înmulțirea a două numere fără semn de câte patru biți*

Funția de adunare și funcția logică ȘI necesară circuitului de înmulțire matricială pot fi combinate într-o singură celulă, notată cu M, care este reprezentată în figura 3.9. . Această celulă implementează următoarea expresie aritmetică, unde **plus** indică operatorul de adunare:

La intarea ”a” se conectează un bit al produsului parțial din linia precedentă de celule. Un circuit de înmulțire pentru biți poate fi realizat prin utilizarea a celule de acest tip, deși unele celule de la periferia matricei vor avea intrările setate la ‘0’. Timpul de execuție al operației de înmulțire este determinat de timpul de propagare al transportului pentru cazul cel mai defavorabil. Ignorând diferențele dintre celulele interne și cele periferice, acest timp este , unde este timpul de propagare al celulei de bază.

*Figura 3.9. Celula pentru înmulțirea matricială a numerelor fără semn*

# **Diagram Description automatically generated4. Proiectare și implementare**

*Figura 4.1. Schema bloc a circuitului de înmulțire în virgulă mobilă a două numere pe 32 de biți*

În figura 4.1 este reprezentată schema bloc conform abordării noastre, pentru înmulțirea în virgulă mobilă a numerelor pe 32 biți, numere a căror reprezentare respectă standardul 754.

Pentru început am folosit două registre pentru a memora numerele X și Y. În aceste registre sunt reprezentate numerele în virgulă mobilă pe 32 biți. Am folosit patru comparatoare pentru a verifica dacă numerele sunt egale cu 0, respectiv +/- infinit, aceste comparatoare apoi trimit semnalele corespunzătoare pentru cazurile menționate. Se calculează cu ajutorul sumatorului suma celor doi exponenți. Pentru a depista eventualele depășiri folosim comparatoare: pentru a determina depășirea inferioară comparăm rezultatul cu 1, iar pentru cea superioară cu 254.

Mantisele sunt înmulțite în componenta de înmulțire, apoi rezultatul este normalizat și deasemenea rotunjit în componenta aferentă acestor operații, actualizând odată cu acesta și exponentul. Pentru a detemina semnul rezultatului principal am folosit un XOR care transmite semnul direct la componenta de prelucrare a rezultatului. Această componentă va mai primi mantisa și exponentul, iar în funcție de semnalele primite de la UC se va determina rezultatul final.

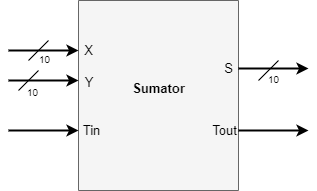
### 4.1 Unitatea de însumare

Implementarea sumatorului cu anticiparea transportului este folosit în proiect pentru a calcula exponenții rezultatului final. Acesta trebuie să adune exponenții celor două numere din virgula mobilă și apoi să scadă deplasamentul.

Sumatorul din figura 4.2. are ca și intrări două semnale de intrare pe 10 biți care reprezintă exponenții, un semnal de intrare de transport, un semnal de ieșire pe 10 biți reprezentând exponentul produs prin adunare, și un semnal de ieșire pentru transportul produs în urma adunării.

Acest sumator folosește sumatoare cu anticiparea transportului pe 2 biți pentru a determina funcțiile pentru generarea și propagarea transportului.

Generatorul de transport utilizat de sumator preia funcțiile pentru generarea și pentru propagarea transportului pe grup de 2 biți și produce transportul pentru următorul sumator pe 2 biți și așa mai departe.

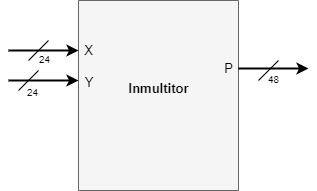


*Figura 4.2. Simbolul sumatorului*

### 4.2 Unitatea de înmulțire

Pentru înmulțirea mantiselor am ales circuitul combinațional de înmulțire matricială, care utilizează sumatoare elementare pe un bit și blocuri logice ȘI. Se înmulțesc două numere binare pe 24 de biți (bitul ascuns + 23 de biți al mantisei).

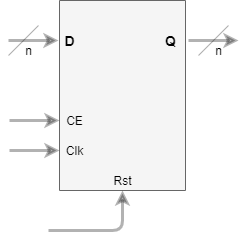
Înmulțitorul primește de produse parțiale, care vor fi de forma și sunt calculate simultan cu ajutorul porților logice ȘI. Apoi aceste produse parțiale sunt însumate cu ajutorul semisumatoarelor elementare, rezultând un produs pe maxim 48 de biți.



*Figura 4.3. Simbolul înmulțitorului*

### 4.3 Registru de n biți cu resetare sincronă

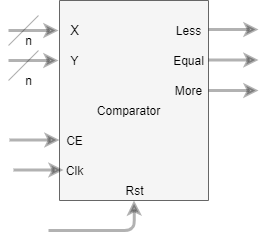
Această componentă (Figura 4.4.) o folosim pentru a reține exponenții și mantisele celor două numere. Ca și intrări avem un vector de date D, care conține mantisele, respectiv exponenții, intrarea de ceas Clk, Rst-ul care resetează registrul și semnalul CE(Clock Enable) care realizează incărcarea registrului de date. Dacă semnalul Rst este 1 logic, pe ieșirea Q va fi un vector de zerouri.În caz contrar, dacă semnalul CE este 1, pe ieșirea Q se încarcă intrarea D.



FDN

*Figura 4.4. Simbolul registrului FDN*

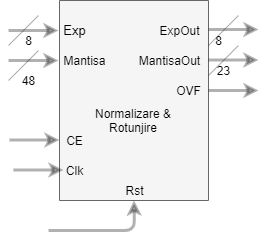
### 4.4 Comparator

Această componentă (Figura 4.5.) am implementat-o pentru a verifica fie dacă operaznii sunt egali cu zero, fie pentru a determina eventualele depășiri. O intrare este folosită pentru încărcarea valorilor din mantise și exponenți, iar pe cealaltă încărcăm valorile cu care le vom compara. De exemplu dacă dorim să determinăm dacă numărul este egal cu zero, vom folosi două comparatoare în care vom introduce pe semnalul X o data exponentul și o data mantisa numărului respectiv, iar pe semnalul Y vom introduce valoarea 0. Dacă intrările sunt egale, pe ieșirea Equal vom avea 1, iar pe celelalte două 0.

*Figura 4.5. Simbolul comparatorului*

### 4.5 Normalizare și rotunjire

Componenta de normalizare și rotunjire are rolul de a menține produsul celor două mantise în formatul standard IEEE 754.

 Așadar, ca și intrări avem exponentul(Exp), mantisa rezultată în urma produsului dintre mantisele celor două numere(Mantisa), semnalul de clock(Clk), de clock enable(CE) și de asemenea cel de reset(Rst). Dacă Rst primește valoarea 1 logic, ieșirile vor fi resetate în mod sincron pe frontul crescător al ceasului. În caz contrat, dacă CE este 1 logic, se va normaliza mantisa, se va actualiza exponentul în funcție de numărul de deplasări pe care trebuie să îl facem pentru a normaliza rezultatul, și se va determina biții g,r,s care ulterior vor fi folosiți pentru operația de rotunjire. În cazul în care exponentul are valoarea maximă, mantisa va deveni zero, iar ieșirea OVF va fi 1(adică există o depășire). La final se face rotunjirea modificând ultimul bit al mantisei cu valoarea grs-ului.

*Figura 4.6. Simbolul componentei de normalizare și rotunjire*

### 4.6 Unitatea de control

*Figura 4.7. Diagrama de stare a automatului*

### 4.7 Modulul principal

Acest modul leagă toate componentele prezentate mai sus.

# **5. Rezultate experimentale**

### 5.1 Unitate de însumare

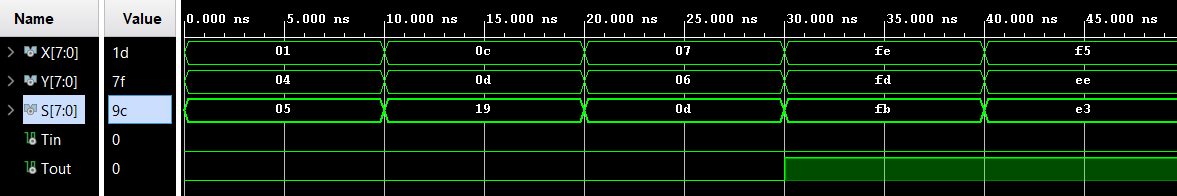
Simularea sumatorului pe 8 biți a fost realizată în mediul de lucru Vivado. Componenta de simulare testează funcționarea corectă a operației de adunare pentru câteva valori specifice unor intervale, și anume: valori apropriate de pragul minim, valori de la mijlocul intervalului și valori apropriate de limita superioară.

Pentru intervalul inferior de reprezentare al numerelor pe 8 biți am adunat următoarele valori scrise în baza 10: 1 și 4 având ca rezultat în hexazecimal x”05”, 12 și 13 rezultatul fiind x”19” și 7 adunat cu 6 rezultând x”0D”. Semnalul de transport de ieșire a fiecărei operații de mai sus este ‘0’.

Pentru intervalul din mijloc am realizat doar o operație pentru numerele 29 și 127. În urma adunării a rezultat numărul în hexazecimal x”7F”, cu transport ‘0’.

În ultimul interval, cel superior, am realizat două adunări: 254 și 253 cu rezultatul x”1FB”, 245 și 238 având ca rezultat x”1E3”. În ambele cazuri, primul bit reprezintă transportul.

La sfârșitul simulării am afișat numărul de erori, acesta fiind 0, de unde putem deduce faptul că rezultatul operațiilor de mai sus a fost calculat corect.



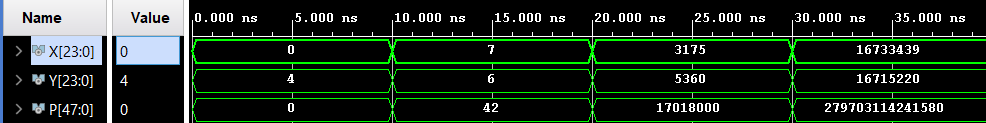
*Figura 5.1. Simularea funcționării sumatorului cu anticiparea transportlui în mediul de lucru Vivado*

### 5.2 Unitatea de înmulțire matricială

Simularea înmulțitorului a fost realizată în mediul de lucru Vivado. Componenta de simulare testează funcționarea corectă a operației de înmulțire pentru câteva valori specifice unor intervale, și anume: valori apropiate de pragul minim, valori de la mijlocul intervalului și valori apropiate de limita superioară.

Pentru intervalul inferior de reprezentare al numerelor am înmulțit următoarele valori, scrise în baza 10: 0 și 4, rezultatul fiind 0, și de asemenea am mai înmulțit și 7 cu 6, rezultând 42 în decimal, care în reprezentarea hexazecimală este egală cu valoarea x”2A”.

Pentru intervalul din mijloc am realizat doar o operație pentru numerele 3145 și 5360, rezultând 16857200, care este echivalent cu valoarea x”103AC90”.

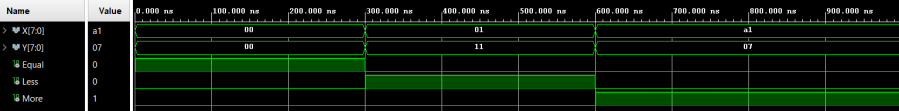
În ultimul interval, cel superior, am realizat un nou test, înmulțind x”FF54FF” cu x”FF0DD4” care a rezultat conform calculelor valoarea x”FE6374C4562C”.

*Figura 5.2. Simularea funcționării înmulțitorului matricicial în mediul de lucru Vivado*

### 5.3 Registru de n biți cu resetare sincronă

### 5.4 Comparator

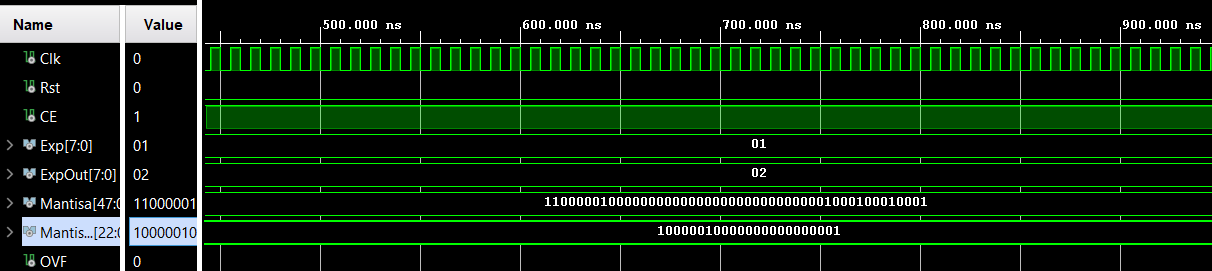
Simularea comparatorului a fost realizată în mediul de lucru Vivado. Componenta de simulare testează funcționarea corectă a operației de comparare pentru câteva valori pentru a demonstra corectitudinea componentei.

După cum se vede și în figura de mai jos se activează semnalele corespunzătoare fiecărui caz: semnalul Equal când am introdus două numere egale, semnalul Less când primul este mai mic decât al doilea număr și semnalul More când primul număr este mai mare.

*Figura 5.3. Simularea funcționării comparatorului în mediul de lucru Vivado*

### 5.5 Normalizare și rotunjire

Simularea componentei a fost realizată în mediul de lucru Vivado. Componenta de simulare testează funcționarea corectă a operației de normalizare și rotunjire. După cum se vede și în figura 5.4. pe semnalul MantisaOut(penultima linie) ies primii cei mai semnificativi 23 de biți care apar după primul bit de 1, mai exact bitul ascuns. De asemenea se poate observa și rotunjirea ultimei cifre al mantisei. Valoarea OVF este 0, deoarece în acest caz nu se pune problema unei depășiri.

* + 1. 

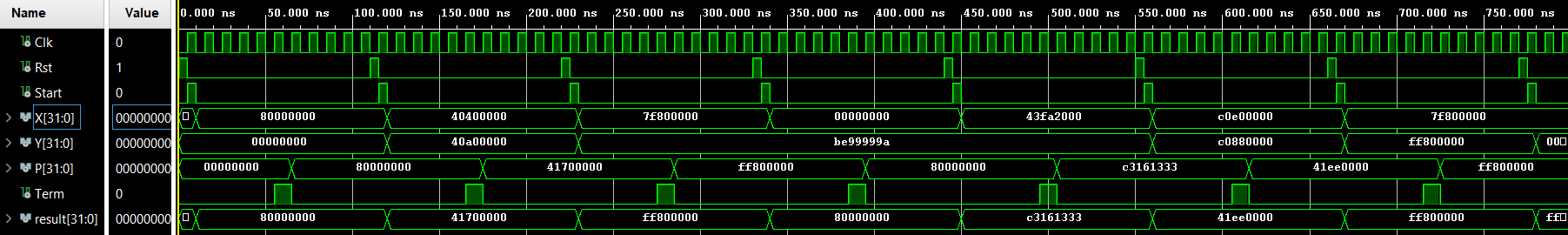
*Figura 5.4. Simularea funcționării normalizatorului în mediul de lucru Vivado*

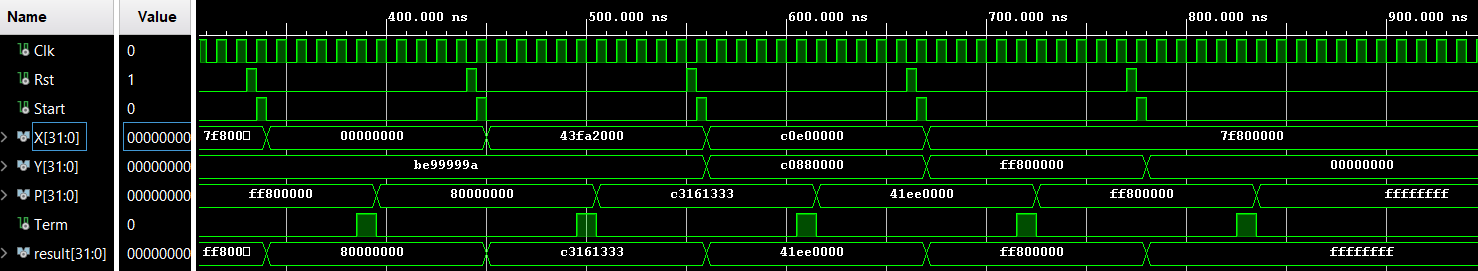
### 5.6 Unitatea de control

### 5.7 Modulul principal

Simularea proiectului a fost realizată în mediul de lucru Vivado. Componenta de simulare testează funcționarea corectă a operației de înmulțire în virgulă mobilă pentru câteva valori, reușind să cuprindem și unele cazuri mai speciale.

Acum vom prezenta câteva calcule mai reprezentative, care demonstrează corectitudinea soluției abordate. Primele două numere pe care le-am înmulțit sunt -0 (x”8000 0000”) cu +0 (x”0000 0000”), rezultatul fiind -0 (x”8000 0000”). Un alt calcul important este cazul în care înmulțim un număr cu infinit (x”7f80 0000”), în cazul nostru am înmulțit cu valoarea -0.3 (x”be99 999a”), care este evident - (x”ff80 0000”). Pentru înmulțirea numărului 500.25 (x”43fa 2000”) cu -0.3 (x”be99 999a”) produsul este: -150.075 (x”c316 1333”). Următoarele două numere importante care au fost înmulțite sunt:  (x” 7f80 0000”) cu  (x”ff80 0000”) având rezultatul (x” 7f80 0000”). Iar pentru cazul de nedeterminare (x”0000 0000”) ne dă o valoare NaN, care este semnalizată cu valoarea x”ffff ffff”.



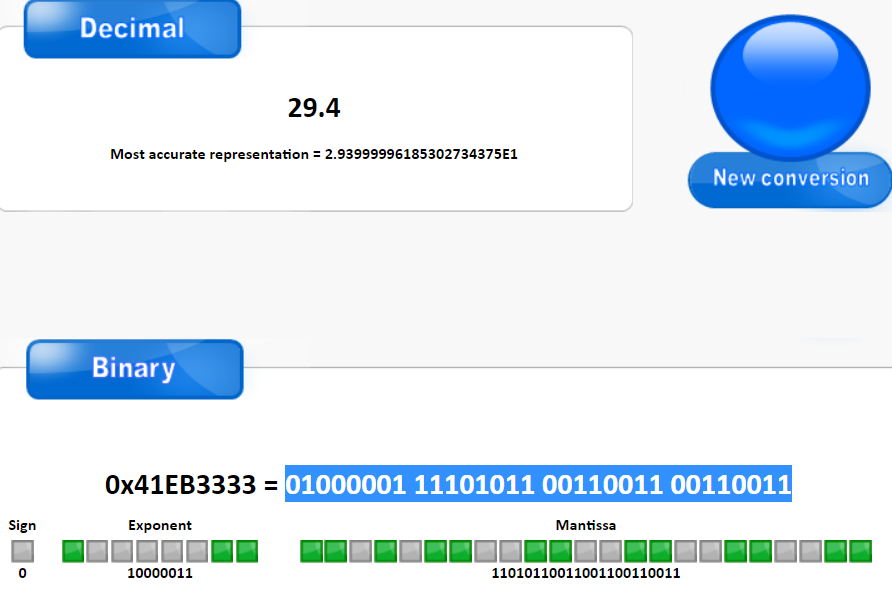
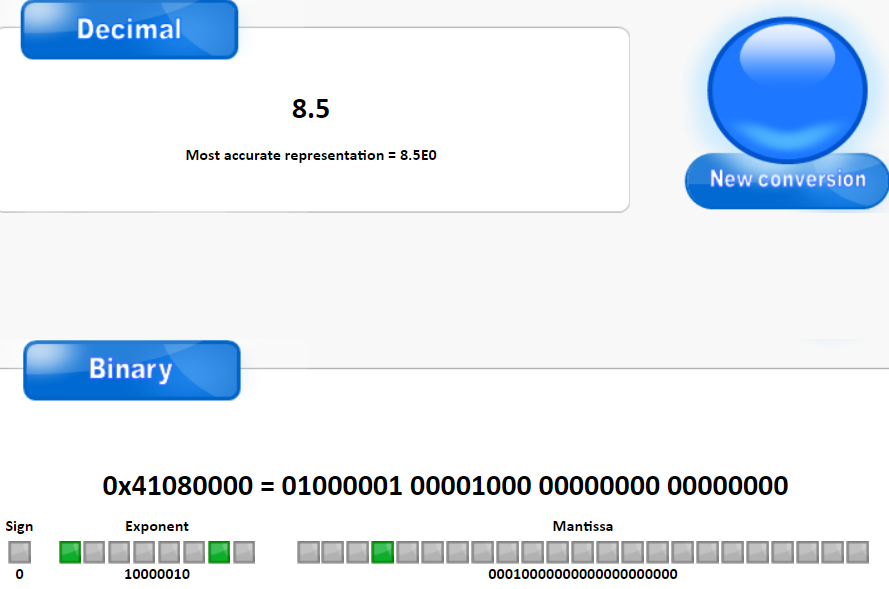


*Figura 5.5. Simularea funcționării înmulțirii în virgulă mobilă în mediul de lucru Vivado*

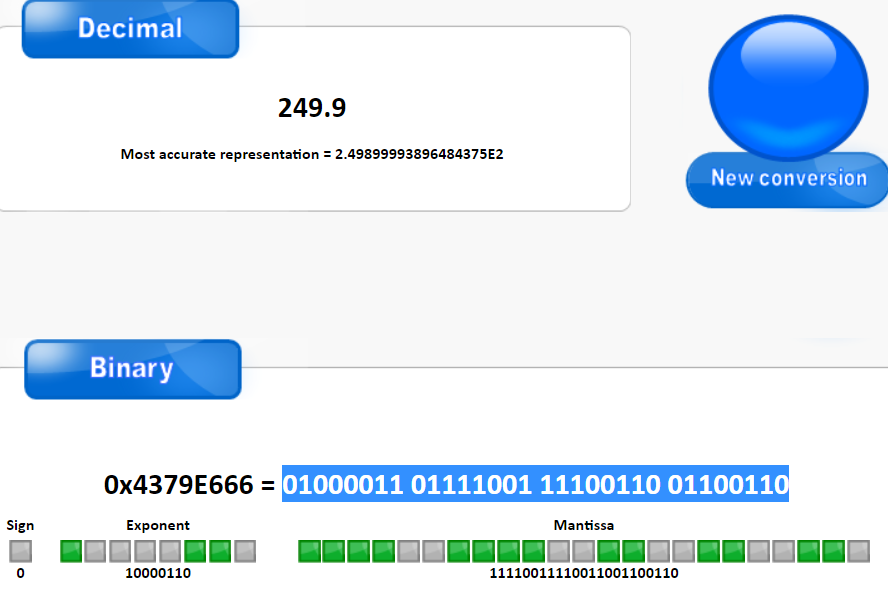
# **6. Manual de utilizare**

Pentru realizarea înmulțirii a două numere în virgula mobilă trebuie să urmăm următorii pași:

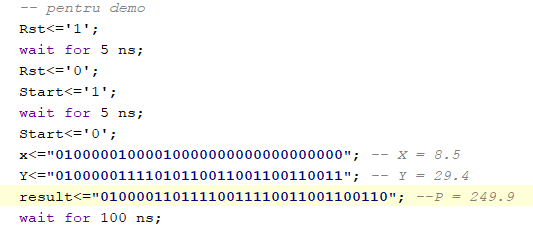
* intrăm pe site-ul [Float (IEEE754 Single precision 32-bit) Converter (binaryconvert.com)](https://www.binaryconvert.com/convert_float.html) pentru a converti numerele din zecimal în numere în virgulă mobilă (sau oricare alt site care convertește numerele din zecimal în virgulă mobilă)
* cu ajutorul unui calculator realizăm produsul celor două numere în zecimal, iar rezultatul îl convertim în virgulă mobilă pentru a demonstra corectitudinea operației
* toate cele trei valori din virgula mobilă, și anumele cele două numere pe care dorim să le înmulțim cât și rezultatul acestora, le introducem în simulatorul Vivado, iar apoi simulăm programul și putem observa rezultatul (tot acolo în simulare vor fi mai multe operații de înmulțire cu diferite valori, dar dacă cineva dorește, poate să adauge alte numere, urmând pașii anteriori)



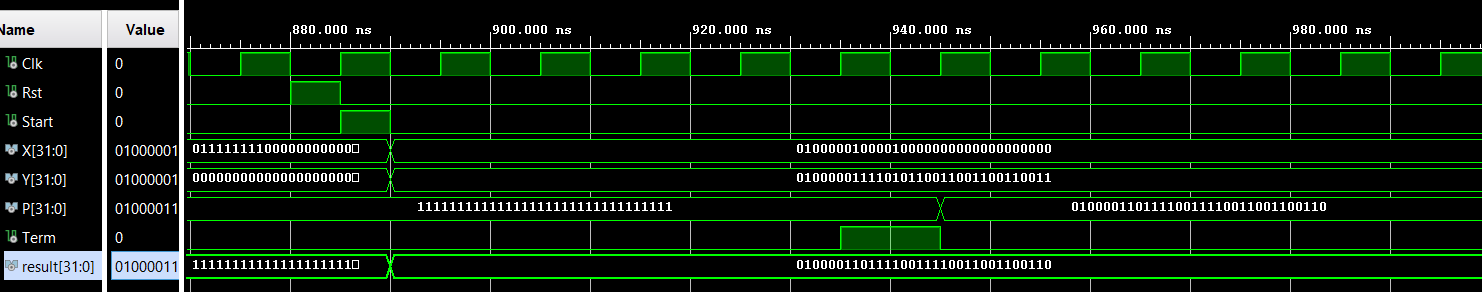
Figură 6.. Primul număr convertit în virgulă mobilă Figură 6.2. Al doilea număr convertit în virgulă mobiă



Figură 6.3. Rezultatul în virgulă mobilă



Figură 6.4. Valorile introduse în simulatorul Vivado



Figură 6.5. Rezultatul simulării

# **7. Concluzii**

În urma acestui proiect s-a realizat un modul de înmulțire în virgulă mobilă a numerelor reprezentate pe 32 de biți. Numerele sunt reprezentate în precizie simplă, ceea ce facilitează rezolvarea operației de înmulțire, deoarece fiecare componentă a numărului are o cale distinctă față de celălalte.

Astfel, am reușit să rezolvăm problema fără prea multe bătăi de cap, totodată am fixat foarte bine primele capitole ale acestei materii. Un alt punct forte al acestui proiect a fost faptul că testarea s-a putut realiza în simulator, astfel am învățat cum se face corect o simulare.

Avantajul acestui proiect este faptul că, cuprinde o varietate largă de operații utile, folosind diferiți algoritmi clasici, fiind ușor de utilizat fără a necesita diferite module hardware în plus. Unul dintre dezavantajele evidente al acestei implementări este faptul că suntem limitați la un număr fix de biți pentru reprezentare și prelucrare.

Ca și dezvoltări ulterioare ale implementării proiectului ar fi foarte multe, însă noi dorim să propunem o aplicație de frontend pentru a ușura introducerea numerelor și testarea modulului și de asemenea modificarea operației, cum ar fi adunarea.

# **Anexe**

## **Comparator**

entity comparator is

generic(n:natural);

Port(

X: in std\_logic\_vector(n-1 downto 0);

Y: in std\_logic\_vector(n-1 downto 0);

Less: out std\_logic;

Equal: out std\_logic;

More: out std\_logic

);

end comparator;

architecture Behavioral of comparator is

begin

Equal<='1' when X=Y else '0';

Less<='1' when X<Y else '0';

More<='1' when X>Y else '0';

end Behavioral;

## **Registru de n biți cu resetare sincronă**

entity FDN is

generic(n:natural);

port(Clk:in std\_logic;

D:in std\_logic\_vector(n-1 downto 0);

Rst:in std\_logic;

CE:in std\_logic;

Q:out std\_logic\_vector(n-1 downto 0));

end FDN;

architecture Behavioral of FDN is

begin

process(Clk,Rst)

begin

if(rising\_edge(Clk)) then

if(Rst='1') then

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

else

if(CE='1') then

Q<=D;

end if;

end if;

end if;

end process;

end Behavioral;

## **Înmulțitor**

entity inm is

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

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

P : out STD\_LOGIC\_VECTOR (47 downto 0));

end inm;

architecture Behavioral of inm is

type PP\_array is array(0 to 23) of std\_logic\_vector(23 downto 0);

type S\_array is array(0 to 23) of std\_logic\_vector(24 downto 0);

type T\_array is array(0 to 22) of std\_logic\_vector(24 downto 0);

signal PP:PP\_array;

signal S:S\_array;

signal T:T\_array;

component sum\_el is

Port ( X : in STD\_LOGIC;

Y : in STD\_LOGIC;

Tin : in STD\_LOGIC;

S : out STD\_LOGIC;

Tout : out STD\_LOGIC);

end component;

begin

generarePP1: for index1 in 0 to 23 generate

generarePP2: for index2 in 0 to 23 generate

PP(index1)(index2)<=X(index1) and Y(index2);

end generate;

end generate;

generareT: for index in 0 to 22 generate

T(index)(0)<='0';

end generate;

generareS: for index in 0 to 23 generate

S(0)(index)<=PP(0)(index);

end generate;

S(0)(24)<='0';

generareSE1:for index1 in 0 to 22 generate

generareSE2:for index2 in 0 to 23 generate

sum: sum\_el port map(X=>S(index1)(index2+1),

Y=>PP(index1+1)(index2),

Tin=>T(index1)(index2),

Tout=>T(index1)(index2+1),

S=>S(index1+1)(index2));

end generate;

S(index1+1)(24)<=T(index1)(24);

end generate;

generareP1:for index in 0 to 22 generate

P(index)<=S(index)(0);

end generate;

generareP2: for index in 0 to 24 generate

P(index+23)<=S(23)(index);

end generate;

end Behavioral;

## **Normalizare**

entity normalizare\_rotunjire is

Port ( Clk : in STD\_LOGIC;

Rst : in STD\_LOGIC;

CE : in STD\_LOGIC;

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

Mantisa : in STD\_LOGIC\_VECTOR (47 downto 0);

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

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

OVF : out STD\_LOGIC);

end normalizare\_rotunjire;

architecture Behavioral of normalizare\_rotunjire is

signal exponent\_n:std\_logic\_vector(7 downto 0):=x"00";

signal mantisa\_n:std\_logic\_vector(22 downto 0):=(others=>'0');

signal grs\_aux: std\_logic := '0';

signal dep:std\_logic:='0';

begin

process(Mantisa,Clk,CE,Rst,Exp)

begin

if(rising\_edge(Clk)) then

if(Rst='1') then

exponent\_n<=(others =>'0');

mantisa\_n<=(others =>'0');

grs\_aux<='0';

dep<='0';

elsif(CE='1') then

if(Exp=x"FF") then

dep<='1';

exponent\_n<=Exp;

mantisa\_n<=(others=>'0');

else

case(Mantisa(47 downto 46)) is

when "01"=> mantisa\_n<=Mantisa(45 downto 23);

exponent\_n<=Exp;

for i in 22 downto 0 loop

grs\_aux <= grs\_aux or Mantisa(i);

end loop;

when "10"=> mantisa\_n<='0' & Mantisa(45 downto 24);

exponent\_n<=Exp+1;

for i in 23 downto 0 loop

grs\_aux <= grs\_aux or Mantisa(i);

end loop;

when "11"=> mantisa\_n<='1' & Mantisa(45 downto 24);

exponent\_n<=Exp+1;

for i in 23 downto 0 loop

grs\_aux <= grs\_aux or Mantisa(i);

end loop;

when others=> mantisa\_n<=Mantisa(45 downto 23);

exponent\_n<=Exp;

grs\_aux<='0';

end case;

if(Exp=x"FF") then

dep<='1';

exponent\_n<=Exp;

mantisa\_n<=(others=>'0');

end if;

end if;

end if;

end if;

end process;

OVF<=dep;

MantisaOut<=(mantisa\_n + 1) when grs\_aux='1' else mantisa\_n;

ExpOut<=exponent\_n;

end Behavioral;

## **Sumator elementar**

entity sum\_el is

Port ( X : in STD\_LOGIC;

Y : in STD\_LOGIC;

Tin : in STD\_LOGIC;

S : out STD\_LOGIC;

Tout : out STD\_LOGIC);

end sum\_el;

architecture Behavioral of sum\_el is

begin

S<=X xor Y xor Tin;

Tout<=(X and Y) or (X and Tin) or (Y and Tin);

end Behavioral;

## **Sumator pe 2 biți**

entity sum2 is

Port ( X : in STD\_LOGIC\_VECTOR (1 downto 0);

Y : in STD\_LOGIC\_VECTOR (1 downto 0);

Tin : in STD\_LOGIC;

S : out STD\_LOGIC\_VECTOR (1 downto 0);

P : out STD\_LOGIC;

G : out STD\_LOGIC);

end sum2;

architecture Behavioral of sum2 is

signal g0,g1,p0,p1:std\_logic;

signal T1:std\_logic:='0';

begin

S(0) <= X(0) xor Y(0) xor Tin;

T1<=g0 or (p0 and Tin);

S(1) <= X(1) xor Y(1) xor T1;

g0 <= X(0) and Y(0);

g1 <= X(1) and Y(1);

p0 <= X(0) or Y(0);

p1 <= X(1) or Y(1);

G <= g1 or (p1 and g0);

P <= p1 and p0;

end Behavioral;

## **Sumator pe 10 biți**

entity sum8 is

Port ( X : in STD\_LOGIC\_VECTOR (9 downto 0);

Y : in STD\_LOGIC\_VECTOR (9 downto 0);

Tin : in STD\_LOGIC;

S : out STD\_LOGIC\_VECTOR (9 downto 0);

Tout : out STD\_LOGIC);

end sum8;

architecture Behavioral of sum8 is

signal G0,G2,G4,G6,G8,P0,P2,P4,P6,P8:std\_logic:='0';

signal T2,T4,T6,T8:std\_logic:='0';

begin

sum2\_0: entity Work.sum2 port map(

X => X(1 downto 0),

Y => Y(1 downto 0),

Tin => Tin,

S => S(1 downto 0),

P => P0,

G => G0);

T2<= G0 or (P0 and Tin);

sum2\_2: entity WORK.sum2 port map(

X => X(3 downto 2),

Y => Y(3 downto 2),

Tin => T2,

S => S(3 downto 2),

P => P2,

G => G2);

T4 <= G2 or (P2 and T2);

sum2\_4: entity WORK.sum2 port map(

X => X(5 downto 4),

Y => Y(5 downto 4),

Tin => T4,

S => S(5 downto 4),

P => P4,

G => G4);

T6 <= G4 or (P4 and T4);

sum2\_6: entity WORK.sum2 port map(

X => X(7 downto 6),

Y => Y(7 downto 6),

Tin => T6,

S => S(7 downto 6),

P => P6,

G => G6);

T8 <= G6 or (P6 and T6);

sum2\_8: entity WORK.sum2 port map(

X => X(9 downto 8),

Y => Y(9 downto 8),

Tin => T8,

S => S(9 downto 8),

P => P8,

G => G8);

Tout <= G8 or (P8 and T8);

end Behavioral;

## **Unitatea de control**

entity UC is

Port ( Clk : in STD\_LOGIC;

Rst : in STD\_LOGIC;

Start : in STD\_LOGIC;

zeroX : in STD\_LOGIC;

zeroY : in STD\_LOGIC;

Load : out STD\_LOGIC;

CE\_sum : out STD\_LOGIC;

CE\_inm : out STD\_LOGIC;

CE\_norm : out STD\_LOGIC;

zero : out STD\_LOGIC;

Term : out STD\_LOGIC);

end UC;

architecture Behavioral of UC is

type TIP\_STARE is (idle, initializare, adunare, verificare1, inmultire, normalizare\_rotunjire,stop);

signal stare: TIP\_STARE;

begin

process(Clk)

begin

if(rising\_edge(Clk)) then

if(Rst='1') then

stare<=idle;

else

case stare is

when idle=>

if(Start='1') then

stare<=initializare;

else

stare<=idle;

end if;

when initializare=>

stare<=adunare;

when adunare=>

stare<=verificare1;

when verificare1=>

if(zeroX='1' or zeroY='1') then

zero<='1';

else

zero<='0';

end if;

stare<=inmultire;

when inmultire=>

stare<=normalizare\_rotunjire;

when normalizare\_rotunjire=>

stare<=stop;

when stop=>

stare<=idle;

end case;

end if;

end if;

end process;

process(stare)

begin

case stare is

when idle=>

Load<='0';

CE\_sum<='0';

CE\_inm<='0';

CE\_norm<='0';

Term<='0';

when initializare=>

Load<='1';

CE\_sum<='0';

CE\_inm<='0';

CE\_norm<='0';

Term<='0';

when adunare=>

Load<='0';

CE\_sum<='1';

CE\_inm<='0';

CE\_norm<='0';

Term<='0';

when verificare1=>

Load<='0';

CE\_sum<='0';

CE\_inm<='0';

CE\_norm<='0';

Term<='0';

when inmultire=>

Load<='0';

CE\_sum<='0';

CE\_inm<='1';

CE\_norm<='0';

Term<='0';

when normalizare\_rotunjire=>

Load<='0';

CE\_sum<='0';

CE\_inm<='0';

CE\_norm<='1';

Term<='0';

when stop=>

Load<='0';

CE\_sum<='0';

CE\_inm<='0';

CE\_norm<='0';

Term<='1';

end case;

end process;

end Behavioral;

## **Sumator**

entity sumator is

Port ( X : in STD\_LOGIC\_VECTOR (9 downto 0);

Y : in STD\_LOGIC\_VECTOR (9 downto 0);

Tin : in STD\_LOGIC;

S : out STD\_LOGIC\_VECTOR (9 downto 0);

Tout : out STD\_LOGIC);

end sumator;

architecture Behavioral of sumator is

signal S\_aux:std\_logic\_vector(9 downto 0);

signal Tout\_aux:std\_logic;

begin

sum: entity WORK.sum8

port map(X=>X, Y=>Y, Tin=>Tin, S=>S\_aux, Tout=>Tout\_aux);

sub: entity WORK.sum8

port map(S\_aux, Y=>"1110000001", Tin=>'0', S=>S, Tout=>Tout);

end Behavioral;

## **Prelucrare rezultat**

entity prelucrareRezultat is

Port ( Clk : in STD\_LOGIC;

Rst : in STD\_LOGIC;

CE : in STD\_LOGIC;

numar : in STD\_LOGIC\_VECTOR (30 downto 0);

semn : in STD\_LOGIC;

zero : in STD\_LOGIC;

zeroX : in STD\_LOGIC;

zeroY : in STD\_LOGIC;

dep\_inf : in STD\_LOGIC;

dep\_sup : in STD\_LOGIC;

infinit : in STD\_LOGIC;

NAN : in STD\_LOGIC;

rezultat : out STD\_LOGIC\_VECTOR (31 downto 0));

end prelucrareRezultat;

architecture Behavioral of prelucrareRezultat is

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

constant valoarea\_NAN:std\_logic\_vector(31 downto 0):=(others=>'1');

begin

process(Clk)

begin

if(rising\_edge(Clk)) then

if(Rst='1') then

rezultat\_aux<=(others=>'0');

end if;

if(CE='1') then

if(NAN='1' and (zeroX and zeroY)='0') then

rezultat\_aux<=valoarea\_NAN;

elsif(zero='1') then

rezultat\_aux(30 downto 0)<=(others=>'0');

rezultat\_aux(31)<=semn;

elsif(dep\_inf='1' or dep\_sup='1' or infinit='1') then

rezultat\_aux<=semn & "1111111100000000000000000000000";

else

rezultat\_aux<=semn & numar;

end if;

end if;

end if;

end process;

rezultat<=rezultat\_aux;

end Behavioral;

## **Modulul principal**

entity inmultitor\_VM is

Port ( Clk : in STD\_LOGIC;

Rst : in STD\_LOGIC;

Start : in STD\_LOGIC;

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

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

P : out STD\_LOGIC\_VECTOR (31 downto 0);

Term : out STD\_LOGIC);

end inmultitor\_VM;

architecture Behavioral of inmultitor\_VM is

signal Load, CE\_sum, CE\_inm, CE\_norm:std\_logic:='0';

signal X\_aux, Y\_aux:std\_logic\_vector(31 downto 0):=(others=>'0');

signal LessX, LessY, EqualX, EqualY, MoreX, MoreY:std\_logic:='0';

signal LessE1, LessE2, EqualE1, EqualE2, MoreE1, MoreE2:std\_logic:='0';

signal LessInf1, LessInf2, EqualInf1, EqualInf2, MoreInf1, MoreInf2:std\_logic:='0';

signal S\_exp:std\_logic\_vector(9 downto 0):=(others=>'0');

signal Tout\_exp:std\_logic:='0';

signal ExpX\_aux, ExpY\_aux:std\_logic\_vector(9 downto 0):=(others=>'0');

signal MantisaX\_aux, MantisaY\_aux:std\_logic\_vector(23 downto 0):=(others=>'0');

signal P\_Mantisa:std\_logic\_vector(47 downto 0):=(others=>'0');

signal Exp\_aux, Exp\_norm:std\_logic\_vector(7 downto 0):=(others=>'0');

signal Mantisa\_aux:std\_logic\_vector(47 downto 0):=(others=>'0');

signal Mantisa\_norm:std\_logic\_vector(22 downto 0):=(others=>'0');

signal OVF\_aux:std\_logic:='0';

signal depi\_aux,deps\_aux,dep\_inf,dep\_sup:std\_logic:='0';

signal semn:std\_logic:='0';

signal numar:std\_logic\_vector(30 downto 0):=(others=>'0');

signal Term\_aux:std\_logic:='0';

signal zero:std\_logic:='0';

signal infinit:std\_logic:='0';

signal NAN:std\_logic:='0';

constant valoarea\_zero:std\_logic\_vector(30 downto 0):=(others=>'0');

constant valoarea\_infinit:std\_logic\_vector(30 downto 0):="1111111100000000000000000000000";

begin

regX: entity WORK.FDN

generic map(n=>32)

port map(Clk=>Clk, D=>X, Rst=>Rst, CE=>Load, Q=>X\_aux);

regY: entity WORK.FDN

generic map(n=>32)

port map(Clk=>Clk, D=>Y, Rst=>Rst, CE=>Load, Q=>Y\_aux);

zero\_X: entity WORK.comparator

generic map(n=>31)

port map(X=>X\_aux(30 downto 0), Y=>valoarea\_zero, Less=>LessX,

Equal=>EqualX, More=>MoreX);

zero\_Y: entity WORK.comparator

generic map(n=>31)

port map(X=>Y\_aux(30 downto 0), Y=>valoarea\_zero, Less=>LessY,

Equal=>EqualY, More=>MoreY);

infinitX: entity WORK.comparator

generic map(n=>31)

port map(X=>X\_aux(30 downto 0), Y=>valoarea\_infinit, Less=>LessInf1,

Equal=>EqualInf1, More=>MoreInf1);

infinitY: entity WORK.comparator

generic map(n=>31)

port map(X=>Y\_aux(30 downto 0), Y=>valoarea\_infinit, Less=>LessInf2,

Equal=>EqualInf2, More=>MoreInf2);

infinit<=EqualInf1 or EqualInf2;

ExpX\_aux<="00" & X\_aux(30 downto 23);

ExpY\_aux<="00" & Y\_aux(30 downto 23);

sum\_exp: entity WORK.sumator

port map(X=>ExpX\_aux, Y=>ExpY\_aux, Tin=>'0', S=>S\_exp, Tout=>Tout\_exp);

depasire\_inf: entity WORK.comparator

generic map(n=>9)

port map(X=>S\_exp(8 downto 0), Y=>"000000001", Less=>LessE1,

Equal=>EqualE1, More=>MoreE1);

depasire\_sup: entity WORK.comparator

generic map(n=>9)

port map(X=>S\_exp(8 downto 0), Y=>"011111110", Less=>LessE2,

Equal=>EqualE2, More=>MoreE2);

depi\_aux<=LessE1 or S\_exp(9);

deps\_aux<=MoreE2 and (not S\_exp(9));

reg\_depi: entity WORK.FD

port map(Clk=>Clk, D=>depi\_aux, CE=>CE\_sum, Rst=>Rst, Q=>dep\_inf);

reg\_deps: entity WORK.FD

port map(Clk=>Clk, D=>deps\_aux, CE=>CE\_sum, Rst=>Rst, Q=>dep\_sup);

MantisaX\_aux<='1' & X\_aux(22 downto 0);

MantisaY\_aux<='1' & Y\_aux(22 downto 0);

regExp: entity WORK.FDN

generic map(n=>8)

port map(Clk=>Clk, D=>S\_exp(7 downto 0), Rst=>Rst, CE=>CE\_sum, Q=>Exp\_aux);

inmultire: entity WORK.inm

port map(X=>MantisaX\_aux, Y=>MantisaY\_aux, P=>P\_Mantisa);

regMantisa: entity WORK.FDN

generic map(n=>48)

port map(Clk=>Clk, D=>P\_Mantisa, Rst=>Rst, CE=>CE\_inm, Q=>Mantisa\_aux);

norm: entity WORK.normalizare\_rotunjire

port map(Clk=>Clk, Rst=>Rst, CE=>CE\_norm, Exp=>Exp\_aux, Mantisa=>Mantisa\_aux,

ExpOut=>Exp\_norm, MantisaOut=>Mantisa\_norm, OVF=>OVF\_aux);

semn<=X\_aux(31) xor Y\_aux(31);

numar<=Exp\_norm & Mantisa\_norm;

NAN<='1' when ((EqualX and EqualInf2) or (EqualY and EqualInf1))='1' else '0';

rezultat: entity WORK.prelucrareRezultat

port map(Clk=>Clk, Rst=>Rst, CE=>term\_aux, numar=>numar, semn=>semn, zero=>zero, zeroX=>EqualX,

zeroY=>EqualY, dep\_inf=>dep\_inf, dep\_sup=>dep\_sup, infinit=>infinit,NAN=>NAN, rezultat=>P);

UC1: entity WORK.UC

port map(Clk=>Clk, Rst=>Rst, Start=>Start, zeroX=>EqualX,

zeroY=>EqualY, Load=>Load, CE\_sum=>CE\_sum, CE\_inm=>CE\_inm,

CE\_norm=>CE\_norm, zero=>zero, Term=>term\_aux);

Term<=term\_aux;

end Behavioral;

# **Bibliografie**

|  |  |
| --- | --- |
| [1] | computerplusro, "COMPUTERPLUS.RO," [Online]. Available: https://computerplusro.wordpress.com/. [Accessed 11 11 2020]. |
| [2] | Z. F. Baruch, Structura sistemelor de calcul cu aplicații, Cluj-Napoca: Editura TODESCO, 2001. |
| [3] | 20 01 2011. [Online]. Available: http://andrei.clubcisco.ro/cursuri/f/f-sym/3cn2/VHDL.pdf. [Accessed 11 11 2020]. |
| [4] | Wikipedia, "Wikipedia," 5 12 2018. [Online]. Available: https://ro.wikipedia.org/wiki/FPGA. [Accessed 11 11 2020]. |
| [5] | Z. F. Baruch, "Reprezentarea numerelor în virgulă mobilă," in *Arhitectura calculatoarelor*, Cluj-Napoca, Editura TODESCO, 200, p. 286. |
| [6] | 17 10 2020. [Online]. Available: http://users.utcluj.ro/~baruch/ssc/labor/Aritm-Combinationala.pdf. [Accessed 11 11 2020]. |