Facultatea de Automatica și Calculatoare

Proiectarea unei unitati aritmetico-logice in virgula flotanta pentru adunare si inmultire

Îndrumător

Student: Ilies Oana-Elena

Grupa: 30232

15.10.2023

Cuprins

Cuprins

1. Introducere

ALU (Unitatea Aritmetică și Logică) este un circuit digital care efectuează operații aritmetice și logice. Acest circuit reprezintă blocul de bază al CPU-ului și GPU-ului, chiar și pentru cei mai simpli microprocesoare.

Utilizarea aritmeticii cu virgulă mobilă în sisteme reconfigurabile performante a fost limitată din cauza cerințelor de spațiu. Abordarea tradițională cu aritmetică cu virgulă fixă implică analize complexe ale gamei dinamice și verificări de precizie. Aceasta poate eșua în cazurile în care gama dinamică nu poate fi restrânsă suficient. În astfel de situații, soluția cu virgulă mobilă este necesară, deoarece permite ajustarea dinamică a gamei prin intermediul unui exponent.

A.Definirea Proiectului

Descrierea Proiectului: Scopul acestui proiect este să proiectăm o unitate aritmetico-logică în virgulă flotantă, o componentă esențială pentru operații numerice complexe în calculatoare. Această unitate va fi dezvoltată pentru a funcționa într-un mediu FPGA (Field-Programmable Gate Array), folosind platforma Vivado.

Motivația Proiectului: Într-o eră în care cerințele pentru putere de calcul cresc exponențial, eficiența în manipularea numerelor cu virgulă mobilă devine critică. Avantajul major al utilizării unui mediu FPGA este flexibilitatea sa și capacitatea de a fi reprogramat pentru a îndeplini cerințele specifice ale unei aplicații.

Prin acest proiect, ne propunem să aducem o contribuție semnificativă în dezvoltarea de soluții de calcul numeric avansat, cu un accent deosebit pe performanță și optimizare. Implementarea în FPGA asigură o platformă care poate fi adaptată pentru o gamă largă de aplicații, de la procesarea semnalelor în timp real până la simulări științifice complexe.

B. Obiective

Obiectiv General: Scopul acestui proiect este proiectarea și implementarea unei unități aritmetico-logice eficiente în virgulă flotantă în mediul de dezvoltare Vivado.

Obiective Specifice:

1. Reprezentarea corectă a virgulei flotante în FPGA
2. Implementarea precisă a reprezentării numerelor în formatul virgulă flotantă, asigurând acuratețea în operații.
3. Adunarea eficientă în virgulă flotantă si dezvoltarea unui algoritm de adunare pentru precizie și performanță optimă.
4. Înmulțirea corectă și eficientă în virgulă flotantă si implementarea unui algoritm de înmulțire pentru precizie și optimizarea resurselor hardware.
5. Verificarea funcționalității unității aritmetico-logice
6. Crearea unui set exhaustiv de teste pentru verificarea corectitudinii operației în diverse scenarii.
7. Identificarea și implementarea îmbunătățirilor pentru maximizarea eficienței și performanței unității aritmetico-logice.

2. Studiu Bibliografic

Standardul IEEE reprezintă numerele în virgulă mobilă cu precizie simplă folosind 1 bit pentru semnul numărului, 8 biți pentru exponent și 23 de biți pentru mantisă. Pornind de la aceasta, numărul este format folosind următoarea formulă:

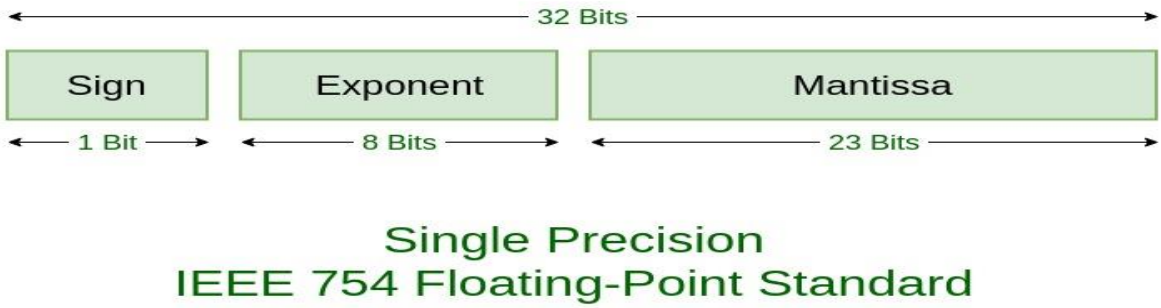


Figura 2.1

Semnul (S): Un singur bit care indică dacă numărul este pozitiv sau negativ. Dacă bitul semn este 0, numărul este considerat pozitiv. În caz contrar, dacă este 1, numărul este negativ.

Exponentul (E): Un grup de biți care reprezintă exponentul. Acesta permite reprezentarea unui interval larg de valori, de la foarte mici la foarte mari.
Mantisa (M): Un grup de biți care reprezintă fracția sau mantisa numărului.

Formatul standard IEEE 754 permite o precizie variabilă, în funcție de cât de mulți biți sunt alocati mantisei. Există formate cu precizie mai mică (ex. single-precision cu 32 de biți) și formate cu precizie mai mare (ex. double-precision cu 64 de biți).

Acest standard este folosit în majoritatea calculatoarelor și a devenit un aspect esențial în calculul științific și în industria de calculatoare. Este important să se înțeleagă formatul IEEE 754 pentru a evita problemele de precizie și pentru a asigura că rezultatele obținute în calcule sunt corecte.

Opera-ție	Rezultat
$x + y$	$(M_x B^{E_x - E_y} + M_y) \times B^{E_y}, E_x \le E_y$
$x - y$	$(M_x B^{E_x - E_y} - M_y) \times B^{E_y}, E_x \le E_y$
$x \times y$	$(M_x \times M_y) \times B^{E_x + E_y}$
$x \div y$	$(M_x \div M_y) \times B^{E_x - E_y}$

Tabel 2.1:Operatii in virgula mobila

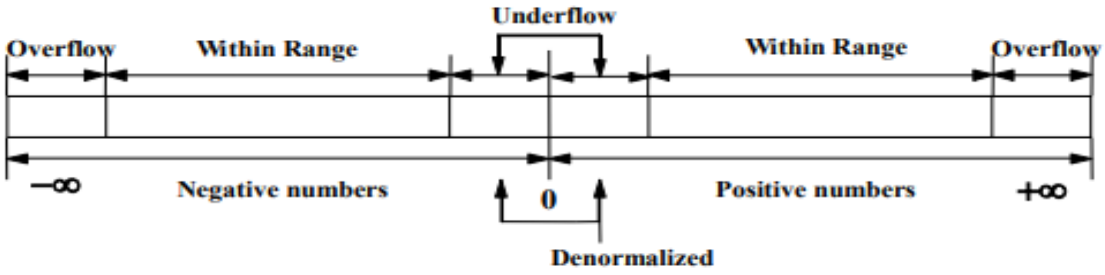


Figura 2.2 Range of floating point numbers

Valoarea maxima a exponentului: +127

Valoarea minima a exponentului: -126

Exceptii in IEEE 754:

Overflow: Rezultatul poate fi +-inf sau valoarea maxima implicita

Underflow: Rezultatul poate fi 0 sau anormal

Invalid: Rezultatul poate fi Nan

Inexact: Rotunjirea specificată de sistem poate fi necesară

Rotunjirea:

Significand	Rounded Result	Error	Significand	Rounded Result	Error
X0.00	X0.	0	X1.00	X1.	0
X0.01	X0.	- 1/4	X1.01	X1.	- 1/4
X0.10	X0.	- 1/2	X1.10	X1. + 1	+ 1/2
X0.11	X1.	+ 1/4	X1.11	X1. + 1	+ 1/4

Tabel 2.2 Rotunjirea

Sumatorul elementar

Sumatorul este blocul operațional de bază implicat în majoritatea operațiilor complexe ale majorității circuitelor, fiind utilizat în special în structura internă a UAL- Unității Aritmetico-logice. Acestea fiind spuse, optimizarea sumatoarelor se traduce în optimizarea întregului sistem, deoarece viteza de operare a acestora va influența direct proporțional viteza de calcul a circuitului final.

Sumatorul elementar este un bloc combinațional ce primește trei intrări și are ca rezultat două ieșiri. Intrările sunt compuse din cei doi biți de adunat x_i, respectiv y_i, și bitul de transport de la nivelul precedent - bitul din poziția mai puțin semnificativă - notat cu T_i. Ieșirile sunt reprezentate de

bitul suma, S_i, respectiv bitul generat de transport către următorul bit din poziție mai semnificativă, T_{i+1}.

Expresiile booleene ale ieșirilor sunt după cum urmează:

$$S_i = x_i \otimes y_i \otimes T_i$$

$$T_{i+1} = x_i y_i + (x_i + y_i) T_i \ .$$

Operația "sau exclusiv" ($\otimes$) are următoarea formulă de calcul:

$$x_i \otimes y_i = x_i \underline{y_i} + \underline{x_i} y_i .$$

Tabelul 3.2. Reprezintă tabelul de adevăr a operației de adunare cu ajutorul sumatorului elementar. Schema bloc a sumatorului elementar este dată de figura 3.2.

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 2.3. Tabelul de adevăr al sumatorului elementar

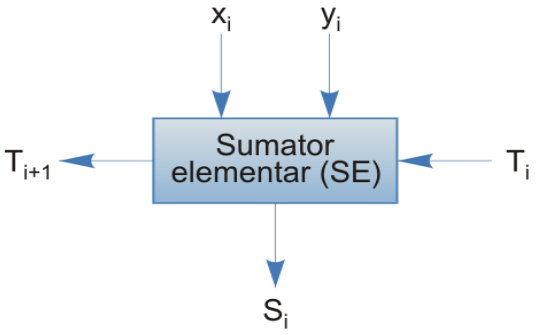


Figura 2.4 Simbolul sumatorului elementar

Un sumator particular este *semisumatorul elementar* care primește ca și intrări cei doi biți de adunat (se omite bitul de transport) și generează la ieșire un bit de sumă și unul de transport.

În mod similar, *semisumatorul elementar* are ca și intrări doi biți - scăzătorul și descăzutul, iar la ieșire este generat un bit diferență și unul de împrumut. Scăzătorul elementar are în plus ca și intrare bitul de împrumut de la bitul mai puțin semnificativ.

Sumatorul cu anticiparea transportului

În sumatorul cu transport anticipat (Carry Look-Ahead), fiecare adunare pe bit elimină dependența 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.

Fiecare etaj va avea un bloc de transport, conținând o logică combinațională ce va calcula transportul de intrare pentru acel etaj în funcție de transportul inițial și toți biții din etajele precedente (deci independent de alte semnale de transport).

Expresiile booleene pentru fiecare etaj de transport sunt date după cum urmează:

$$T_{i+1} = x_i y_i + (x_i + y_i) T_i$$

Semnalele P și G menționate anterior vin în ajutorul simplificării expresiei, referindu-se la *propagarea*, respectiv *generarea* transportului la ieșire. Acestea sunt definite astfel:

$$g_i = x_i y_i$$

$$p_i = x_i + y_i$$

Funcția de *generare* este denumită astfel deoarece etajul *i* va produce un transport ($T_{i+1} = 1$) dacă ambii operanzi vor fi 1 ($x_i y_i = 1$), independent de T_i. Alternativ, *propagarea* se referă la prezența unui transport ($T_{i+1} = 1$) dacă având ca și intrare $T_i = 1$, unul din cei doi operanzi este setat ($x_i + y_i = 1$). Noua expresie devine:

$$T_{i+1} = g_i + p_i T_i$$

Schema bloc a unui sumator de 4 biți cu anticiparea transportului care utilizează un generator de transport anticipat:

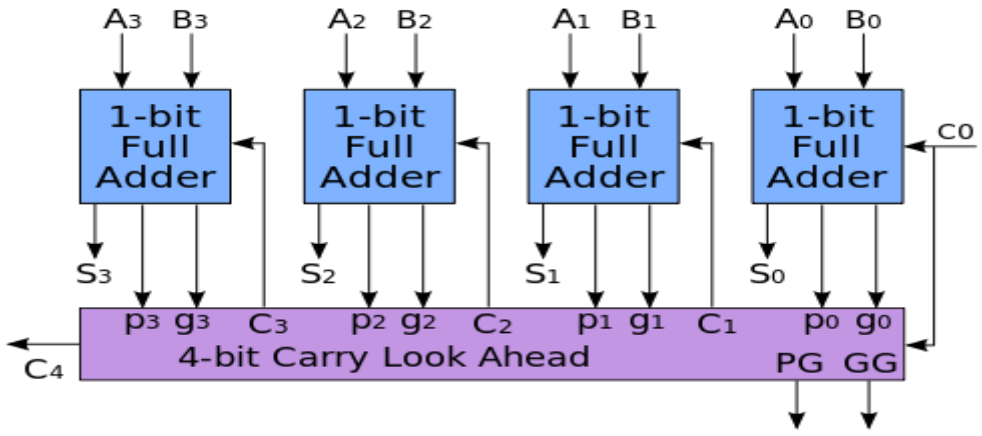


Figura 2.5. Sumator de 4 biți cu anticiparea transportului

Următoarele ecuații definesc ieșirile semnalelor de transport în cazul sumatorului pe 4 biți:

$$T_1 = g_0 + p_0 T_0$$

$$T_2 = g_1 + p_1 g_0 + p_1 p_0 T_0$$

$$T_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 T_0$$

$$T_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 T_0$$

Sumatorul cu propagarea succesivă a transportului

Acest sumator este format dintr-o serie de sumatoare elementare conectate în cascadă și se realizează adunarea bit cu bit a două numere. Se utilizează câte un sumator pentru fiecare pereche de biți de adunat de la fiecare poziție. Spre exemplu adunarea a două numere reprezentate pe patru biți, $x_3x_2x_1x_0$ și $y_3y_2y_1y_0$va utiliza patru sumatoare elementare; primul sumator va aduna biții x_0, respectiv y_0și va genera bitul cel mai puțin semnificativ al sumei, S_0 și un transport T_0spre bitul următor, mai semnificativ (în cazul acesta, bitul de la poziția 1). Asftel ieșirea de transport a primului sumator este conectată la intrarea de transport a celui de-al doilea, realizându-se conectarea în serie a sumatoarelor. Primul sumator, deoarece are intrarea de transport setată pe 0, va putea fi implementat ca un semisumator. Ieșirea de transport a ultimului sumator (T_3), în cazul în care va fi setată, va reprezenta cel mai semnificativ bit (de pe poziția 4, S_4). Schema bloc a acestui tip de sumator este redată în figura 3.3.

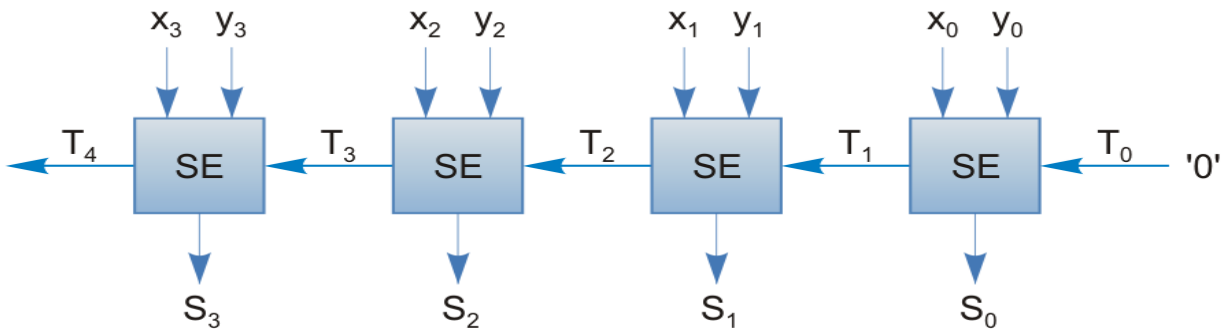


Figura 2.6. Sumator de 4 biți cu propagarea succesivă a transportului

Acest tip de sumator este avantajos ca și simplitate și datorită faptului ca nu necesită logică combinațională suplimentară, având un cost redus. Dezavantajul major, însă, este dat de viteza foarte redusă a acestuia. Această latență se datorează faptului că sumatorul de la fiecare etaj va trebui să aștepte după transportul de intrare propagat de etajul anterior, acest lucru fiind foarte costisitor pentru timpul de execuție.

Acest sumator se poate utiliza și cu funcționalitatea unui scăzător, în cazul în care se vor aduna unul din termeni (cel pozitiv) cu cel de-al doilea (negativ) reprezentat în complement față de doi. Transportul din poziția cea mai semnificativă a rezultatului se neglijează.

Înmulțirea matricială

Circuitele de înmulțire matriceală se deosebesc de circuitele secvențiale prin utilizarea unei logici combinaționale suplimentare ce facilitează calculul produsului într-un pas. Elementele combinaționale folosite sunt elementare, ele realizând însumarea și deplasarea a doi biți sau a unui grup restrâns de biți.

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

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

Ecuația de mai sus se poate rescrie astfel:

$$P = \sum_{i=0}^{n-1} 2^i * \left(\sum_{j=0}^{n-1} x_i * y_j * 2^j \right)$$

Pentru a realiza înmulțirea prin formula dată mai sus avem nevoie de un bloc elementar pentru a calcula produsul $x_i * y_j$. În acest scop vom folosi porți ȘI, deoarece înmulțirea aritmetică a doi biți coincide cu înmulțirea lor logică. Vom avea, în acest fel, un număr de $n \times n$ porti ȘI, ce vor calcula în mod concurent toate produsele elementare dintre biții înmulțitorului și deînmulțitului. Operația de adunare se va realiza utilizând un număr de $n * (n-1)$ sumatoare elementare, dispuse matriceal, asemănător unui sumator bidimensional cu propagare succesivă a transportului. Dintre acestea, un număr de n sumatoare pot fi înlocuite cu semisumatoare elementare. Deplasarea biților reprezentată de factorii 2^i și 2^j ai înmulțirii, se realizează prin dispunerea fizică în mod deplasat a nivelurilor de sumatoare, pe direcțiile x și y.

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 2.7 Înmulțirea matriceală pe 4 biți

Biții produsului final se pot calcula după următoarele formule:

$$P_0 = x_0y_0$$

$$P_1 = x_1y_0 + x_0y_1$$

$$P_2 = x_2y_0 + x_1y_1 + x_0y_2$$

$$P_3 = x_3y_0 + x_2y_1 + x_1y_2 + x_0y_3$$

$$P_4 = x_3y_1 + x_2y_2 + x_1y_3$$

$$P_5 = x_3y_2 + x_2y_3$$

$$P_6 = x_3y_3$$

Figura următoare reprezintă o modalitate de calcul a produselor parțiale. Blocurile ce conțin termeni de forma $x_iy_j$sunt porți ȘI. Fiecare nivel logic conține sumatoare elementare legate în serie ca și în cazul sumatorului cu propagare succesivă a transportului. Transportul de pe un nivel este propagat la următorul nivel și în cele din urmă, dacă valoarea finală a transportului este 1, acest lucru se va traduce prin setarea celui mai semnificativ bit al produsului final. Rândul de sumatoare așezate la început calculează o sumă a primelor două produsuri parțiale, urmând ca mai apoi fiecare etaj de sumatoare să adauge un produs parțial la suma anterior calculată.

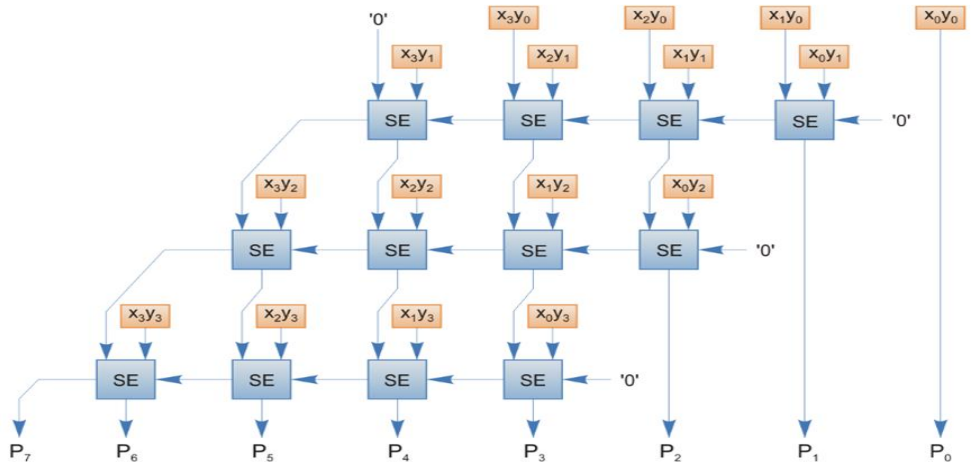


Figura 2.8. Matrice de sumatoare elementare pentru înmulțirea a două numere fără semn de câte patru biți

Adunarea:

Adunarea în VM este mai complexe decât înmulţirea şi împărţirea.
Aceasta deoarece pentru adunarea a celor două numere, trebuie să
se realizeze egalizarea exponenţilor acestora. Aceasta implică compararea mărimii exponenţilor şi apoi alinierea mantisei numărului cu exponentul mai mic. Algoritmul pentru adunare are patru etape principale:

Asigurarea aceloraşi exponenţi: Pentru a aduna A și B, exponenții lor (notati EA și EB) trebuie să fie egali. Dacă EA este diferit de EB, se folosește un substractor pe 8 biți pentru a verifica care dintre cele două numere are exponența mai mare. Dacă rezultatul este 0, înseamnă că A are un exponență mai mare decât B și B trebuie să fie deplasat spre dreapta.

Calcularea diferenței de exponenți: Diferența dintre exponenții (EA - EB) oferă informația necesară pentru a ști cât de mult trebuie deplasată mantisa numărului cu exponența mai mică.

Deplasarea mantisei: Mantisa numărului cu exponența mai mică este deplasată spre dreapta de numărul de poziții determinat de diferența de exponenți. Astfel, ambele numere A și B ajung să aibă aceeași exponență.

Calculul mantisei finale: Mantisa finală este determinată ca fiind maximul dintre mantiselor inițiale ale celor două numere și este incrementată în funcție de transportul (carry) rezultat în urma operației de adunare. În funcție de valoarea acestui transport, mantisa finală poate fi deplasată la dreapta cu 1 sau rămâne neschimbată.

Calcularea exponentului final şi a mantisei: Exponentul final și mantisa finală sunt astfel calculate, luând în considerare acești pași anteriori și aplicând regula specifică pentru obținerea rezultatului corect al adunării numerelor A și B în formatul specific utilizat.

Exemplu: Să se adune numerele 0,510 şi –0,437510 în bina. Se presupune că precizia mantisei este de 4 biţi, exponentul real este cuprins între
–126 şi 127, iar deplasamentul exponentului este 127.

Numerele scrise în binar utilizând notaţia ştiinţifică sunt:
0,510 = 0,1 = 0,1 × 20
= 1,000 × 2–1
–0,437510 = –0,0111 = –0,0111 × 20
= –1,110 × 2–2
Etapele algoritmului sunt următoarele:
1. Se deplasează la dreapta mantisa numărului mai mic (–1,11 × 2–2) şi se incrementează exponentul până când acesta devine egal cu exponentul numărului
mai mare:

–1,110 × 2^ –2 = –0,111 × 2^ –1

2.Se adună mantisele:

1,0 × 2^ –1 + (–0,111 × 2^ –1) = 0,001 × 2^ –1
3.Se normalizează suma, verificând dacă apare depăşire superioară sau inferioară:

0,001 × 2^ –1= 0,010 × 2^ –2 = 0,100 × 2^ –3 = 1,000 × 2^ –4

Deoarece –126 ≤ –4 ≤ 127, nu există depăşire superioară sau inferioară (exponentul deplasat este –4 + 127 = 123, care este cuprins între 0 şi 255).

4. Se rotunjeşte suma:

1,000 × 2^ –4

Suma se poate exprima pe 4 biţi, astfel încât rotunjirea nu este necesară. Suma este deci:

1,000 × 2^ –4 = 0,0001 = 1/1610 = 0,062510

Aceasta este suma corectă dintre 0,510 şi –0,437510.

Înmulțirea:

În contrast cu operația de adunare, înmulțirea implica manipularea exponentului si a mantisei pentru a obtine un rezultat.Algoritmul pentru inmultire:

Verificarea dacă unul dintre numere este 0 sau nu: Se verifică dacă una dintre mantisele numerelor este 0. Dacă una dintre mantise este 0, rezultatul înmulțirii va fi 0 indiferent de ceilalți pași. Deci, în această situație, rezultatul este direct 0.
Adunarea exponentilor și scăderea cu 127:
Exponenții celor două numere sunt adunați, iar apoi se scade 127. În formatul virgulă mobilă, exponenții sunt stocați ca numere întregi și sunt folosiți pentru a reprezenta cât de mare sau mic este numărul. Prin scăderea cu 127, se normalizează valorile exponențiale într-un interval specific.
Înmulțirea mantiselor: Mantisele celor două numere sunt înmulțite împreună. Mantisa reprezintă partea precisă a numărului în virgulă mobilă.
Normalizarea produsului: După înmulțirea mantiselor, produsul obținut trebuie să fie normalizat. Normalizarea implică ajustarea mantisei și a exponențialului pentru a asigura că rezultatul final respectă formatul specific al virgulei mobile. Se pot face ajustări la mantisă și exponențial pentru a obține un rezultat corect.

Exemplu: Să se inumultească numerele 0,510 şi –0,437510 în binar. Se presupune că precizia mantisei este de 4 biţi, exponentul real este cuprins între
–126 şi 127, iar deplasamentul exponentului este 127.

Numerele scrise în binar utilizând notaţia ştiinţifică sunt:
0,510 = 0,1 = 0,1 × 20
= 1,000 × 2^ –1
–0,437510 = –0,0111 = –0,0111 × 20
= –1,110 × 2^ –2

1.Adăugarea exponenților:
2^ −1+2^ −1=2^ −12−1+2^ −1=2^ −1.
2.Înmulțirea mantiselor:
1.000×−0.111=−0.1111.000×−0.111=−0.111.

3.Normalizarea rezultatului
În acest caz, rezultatul este
−0.111×2^ −1−0.111×2^ −1 = −0.0111×20−0.0111×20.

4.Rotunjirea rezultatului
Astfel, înmulțirea numerelor 0.510 și -0.4375 în formatul specificat dă ca rezultat

−0.0111×20=−0.0111=−7/8≈−0.875−0.0111×20=−0.0111=−7/8≈−0.875

3. Analiza

3.1 VHDL

VHDL, prescurtarea de la 'VHSIC Hardware Description Language' (Very High-Speed Integrated Circuit Hardware Description Language), reprezintă un limbaj specificat pentru a descrie și modela comportamentul, arhitectura și funcționarea circuitelor electronice digitale. Este un instrument esențial în proiectarea sistemelor logice combinatorii și secvențiale. Împreună cu Verilog, este unul dintre cele mai prevalente limbaje utilizate în proiectarea și dezvoltarea sistemelor electronice digitale moderne. Este amplu folosit în domeniul microprocesoarelor, telecomunicațiilor, industriei auto și în alte domenii. VHDL este esențial în dezvoltarea asistată de calculator (CAD) pentru circuite integrate și pentru configurarea FPGA-urilor.

3.2 Tehnologii utilizate

3.2.1 Basys 3

Basys 3 este una dintre cele mai bune plăci de pe piață pentru a începe cu FPGA. Este o placă de dezvoltare entry-level construită în jurul unui FPGA Xilinx Artix-7.

Fiind o platformă de dezvoltare a circuitelor digitale completă și gata de utilizat, include suficiente comutatoare, LED-uri și alte dispozitive I/O pentru a permite finalizarea unui număr mare de modele fără a fi nevoie de niciun hardware suplimentar. Există, de asemenea, destui pini I/O FPGA neangajați pentru a permite extinderea design-urilor folosind Pmod-uri Digilent sau alte plăci și circuite personalizate, și toate acestea la un preț convenabil pentru studenți.

Basys 3 este proiectat exclusiv pentru Vivado Design Suite de la Xilinx, iar ediția WebPACK este disponibilă ca descărcare gratuită de la Xilinx.

Ghidurile și demonstrațiile sunt disponibile pentru a ajuta utilizatorii să înceapă rapid cu Basys 3. Acestea pot fi găsite prin fila Materiale de asistență.

Figura 3.1. Placa de dezvoltare Basys3

3.3 Algoritmul de adunare

Algoritmul de adunare în virgulă mobilă este un proces complex care necesită proiectarea și implementarea mai multor componente cheie. În centrul acestui proces se află adunătorul, care este esențial pentru realizarea operațiilor de bază. Prin urmare, este logic să ne concentrăm pe proiectarea unui adunător eficient, deoarece întregul sistem va beneficia de performanța acestuia.
Primul pas în algoritmul de adunare este asigurarea aceloraşi exponenți. Acest lucru implică utilizarea unui scăzător pe 8 biți pentru a efectua operația de scădere între exponenții numerelor A și B, rezultând diferența (EA - EB). Următorul pas este deplasarea mantisei numărului cu exponența mai mică spre dreapta cu numărul de poziții determinat de diferența de exponenți. Acest lucru necesită o componentă de shiftare pe 24 de biți. După aceea, avem nevoie de o componentă de adunare pe 24 de biți pentru a aduna mantisele ajustate după deplasare și eventual complementare (dacă rezultatul scăderii exponenților este negativ). După scăderea exponenților, trebuie să verificăm dacă rezultatul este negativ. Dacă este negativ, trebuie să facem complementul față de 2 al exponenței sau al mantisei corespunzătoare, în funcție de caz. În final, după ajustarea mantiselor și a exponenților conform regulilor specificate, trebuie să calculăm exponenții și mantisele finale care vor oferi rezultatul corect al adunării numerelor A și B în formatul specific utilizat. Aceste componente sunt integrate într-un sistem mai mare care gestionează adunarea numerelor în conformitate cu regulile și condițiile specificate. Fiecare componentă are rolul său specific în efectuarea operațiilor necesare pentru adunare și ajustarea numerelor pentru a obține rezultatul corect conform regulilor date.

Pseudocod:
1. Încărcarea Operandelor:
Acest pas presupune introducerea numerelor în unitatea aritmetico-logică. Să luăm un exemplu:
X = 1.010 (1.25 în decimal)
Y = 0.110 (0.75 în decimal)

2.Comparatii Exponenti:
Există cinci cazuri posibile de comparație a exponenților:
Cazul 1: Exponenții sunt egali (ex = ey)
Exemplu: ex = ey = 3
Rezultat: Adunăm mantisele și copiem exponentul.
Cazul 2: Exponența lui X este mai mare și diferența dintre exponenți este mai mică decât numărul de biți în mantisă (ex > ey and (ex – ey) < numărul de biți în mantisă)
Exemplu: ex = 4, ey = 2
Rezultat: Aliniem mantisa lui Y cu X prin shiftare la dreapta cu 2 poziții.
Cazul 3: Exponența lui X este mai mare și diferența dintre exponenți este mai mare sau egală cu numărul de biți în mantisă (ex > ey and (ex – ey) ≥ numărul de biți în mantisă)
Exemplu: ex = 4, ey = 1
Rezultat: X este copiat în rezultat, deoarece Y este prea mic.
Cazul 4: Exponența lui Y este mai mare și diferența dintre exponențe este mai mică decât numărul de biți în mantisă (ex < ey and (ey – ex) < numărul de biți în mantisă)
Exemplu: ex = 2, ey = 4
Rezultat: Aliniem mantisa lui X cu Y prin shiftare la dreapta cu 2 poziții.
Cazul 5: Exponența lui Y este mai mare și diferența dintre exponențe este mai mare sau egală cu numărul de biți în mantisă (ex < ey and (ey – ex) ≥ numărul de biți în mantisă)
Exemplu: ex = 1, ey = 4

3.Adunati Mantisele:
Continuând cu exemplele de mai sus, adunăm mantisele ajustate în funcție de rezultatul comparației exponenților.
4.Calculul Semnului:
Acest pas determină semnul final al rezultatului. Dacă unul dintre numerele inițiale este negativ, semnul final va fi negativ.
Combinând rezultatele acestor pași, obținem rezultatul final al operației de adunare. În exemplul nostru, rezultatul final ar fi un număr în format virgulă mobilă calculat folosind acești pași
Pseudocod+explicatii:

```
-- 1. Încărcarea Operandelor:
X <= valoare_numerica_X;
Y <= valoare_numerica_Y;

-- 2. Compararea Exponenților:
if ex = ey then
   -- Adunăm  mantisele
   mantisa_rezultat <= mantisa_X + mantisa_Y;
   -- Copiem exponentul
   exponent_rezultat <= ex;
elsif ex > ey and (ex - ey) < lungime_mantisa then
   -- Aliniem mantisa lui Y cu X prin shiftare la dreapta cu 2 poziții
   mantisa_rezultat <= mantisa_X + (mantisa_Y >> (ex - ey));
   -- Copiem exponentul
   exponent_rezultat <= ex;
elsif ex > ey and (ex - ey) >= lungime_mantisa then
   -- Copiem X în rezultat, deoarece Y este prea mic.
   mantisa_rezultat <= mantisa_X;
   -- Copiem exponentul
   exponent_rezultat <= ex;
elsif ex < ey and (ey - ex) < lungime_mantisa then
   -- Aliniem mantisa lui X cu Y prin shiftare la dreapta cu 2 poziții
   mantisa_rezultat <= (mantisa_X >> (ey - ex)) + mantisa_Y;
   -- Copiem exponentul
   exponent_rezultat <= ey;
else
   -- Copiem Y în rezultat
   mantisa_rezultat <= mantisa_Y;
   -- Copiem exponentul
   exponent_rezultat <= ey;
end if;

-- 3. Adunarea Mantiselor:
mantisare_finala <= mantisa_rezultat;

-- 4. Calculul Semnului:
if semn_X = '1' or semn_Y = '1' then
   semn_final <= '1'; -- Semnul final va fi negativ
```

```
else
    semn_final <= '0'; -- Semnul final va fi pozitiv
end if;

-- 5. Obținerea Rezultatului Final:
rezultat_final <= semn_final & exponent_rezultat & mantisare_finala;
```

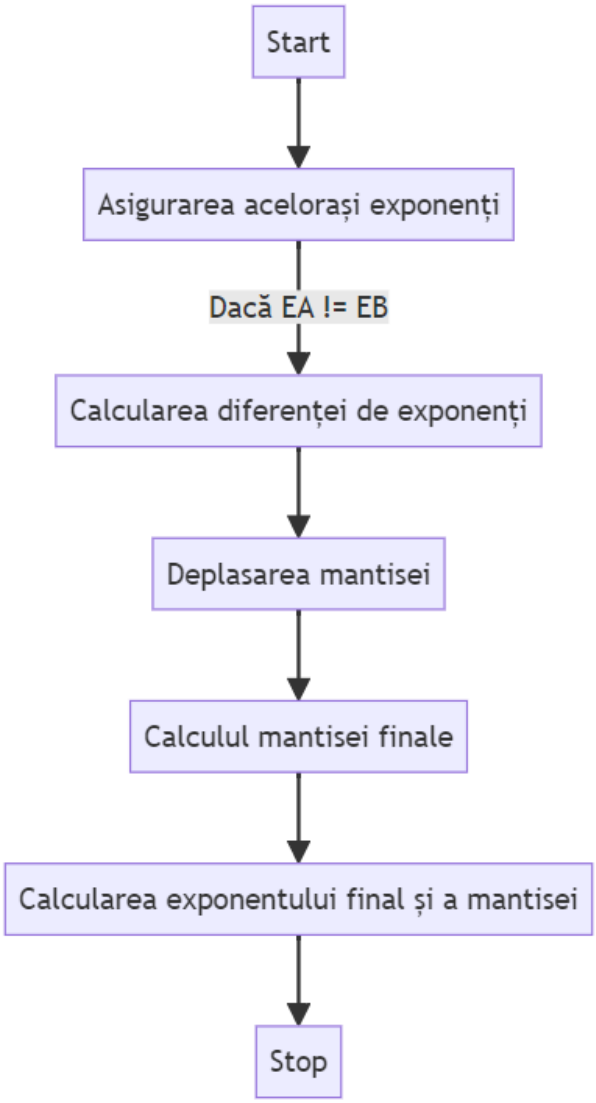


Figura 3.2 Diagrama flux de date pentru adunare

3.4 Algoritmul de Inmultire

Înmulțirea în virgulă mobilă implică mai multe componente esențiale. Adunătorul este nucleul acestui proces, deci accentul este pe proiectarea unui adunător eficient.

Primul pas este calcularea exponentului rezultatului prin adunarea celor doi exponenti si scaderea acestora cu 127. Apoi, mantisele operanzilor sunt înmulțite, pentru inmultire avem nevoie de o compoenta de inmultire ce primeste mantisele pe 24 de biti si genereaza un rezultat pe 48 de biti. Dacă primul bit al rezultatului este 1, luăm cele 23 de biți pentru mantisa rezultatului începând de la al doilea bit și normalizăm rezultatul. Dacă primul bit este zero, luăm cei 23 de biți începând de la al treilea bit înainte. În final, calculăm semnul ca un XOR între semnele operanzilor.
Pentru adunător, am ales să implementez adunătorul cu propagare anticipată a transportului, unul pe 24 de biti, chiar daca mantisa este pe 23 de biti avem nevoie de o compoenta de 24 de biti pentru acel bit “ascuns”, care nu este luat in considerare in momemtul creeari rezultatului in virgule flotanta, dar este essential in calcul. Această descriere oferă o înțelegere a procesului de înmulțire și ajustare a numerelor în conformitate cu regulile specificate pentru exponenți și mantise.

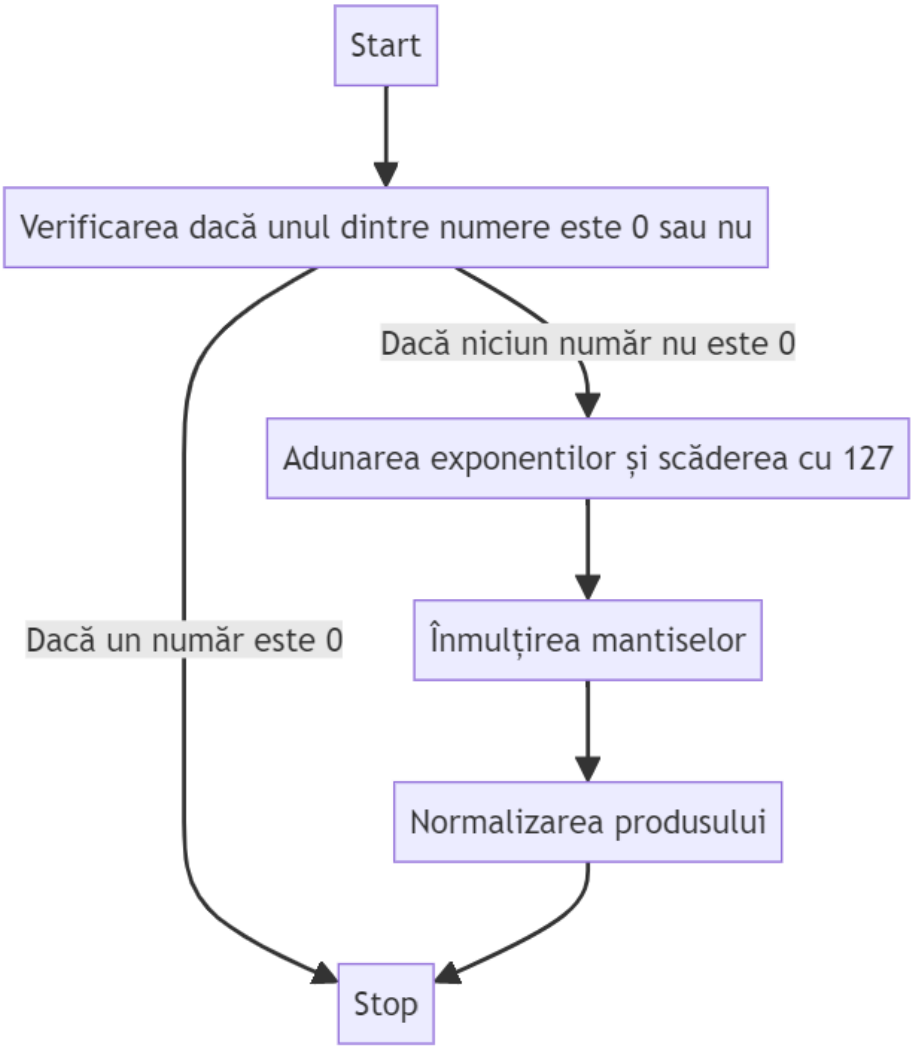


Figura 3.3 Diagrama flux de date pentru inmultire

4 Proiectare

Adunarea

Operația de adunare este operația aritmetică cea mai frecvent utilizată în orice sistem de calcul. Dacă există funcții aritmetice mai complexe, acestea sunt reduse la o serie de adunări. Rețineți că prin creșterea vitezei de adunare, se crește și viteza ALU-ului, însă viteza și costul adăugătoarelor sunt direct proporționale cu complexitatea lor.

Proiectarea operației de adunare în virgulă mobilă nu se oprește doar la manipularea semnelor, exponentelor și mantișelor, ci necesită și gestionarea corectă a cazurilor speciale pentru a asigura comportamentul corect al unității de calcul în orice condiții.

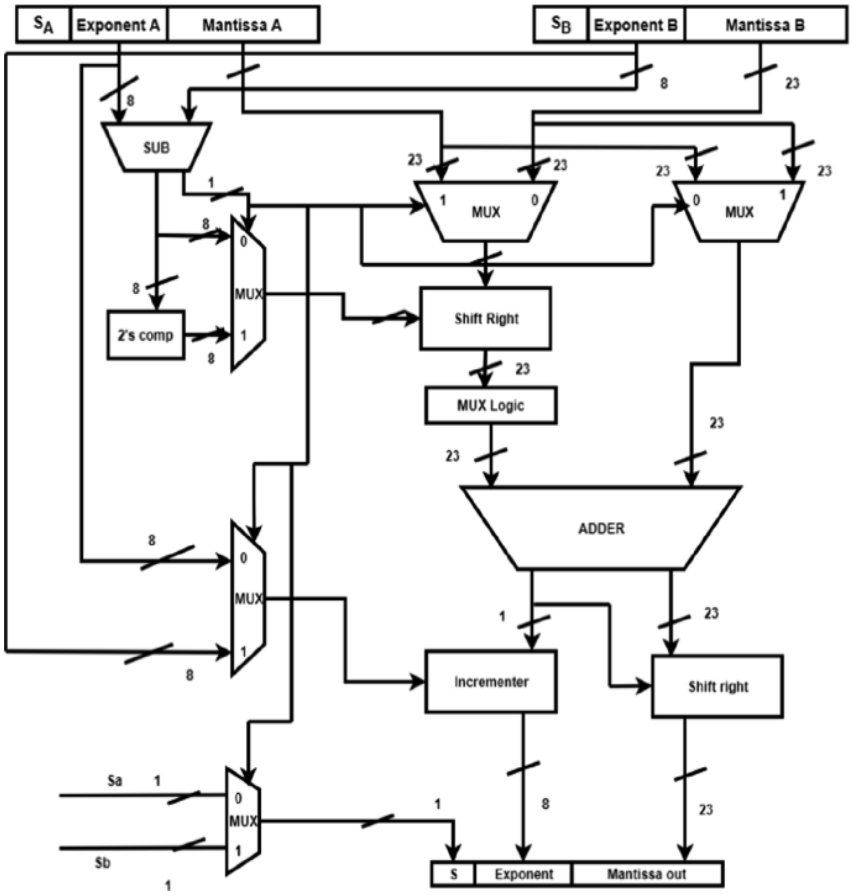


Figura 4.1 Floating Point Adder

Pentru Shiftare avem nevoie de o componenta specifica, Barrel Shifter pe 24 de biți este un circuit digital care poate deplasa un cuvânt de date de 24 de biți cu un număr specificat de biți într-un singur ciclu de ceas, fără a utiliza logica secvențială, doar logica combinatorie pură1.

Acesta este adesea utilizat pentru a efectua operații de shiftare și rotire în microprocesoare moderne1. Un Barrel Shifter pe 24 de biți poate fi implementat ca o secvență de multiplexoare unde ieșirea unui multiplexor este conectată la intrarea următorului multiplexor într-un mod care depinde de distanța de shiftare1.

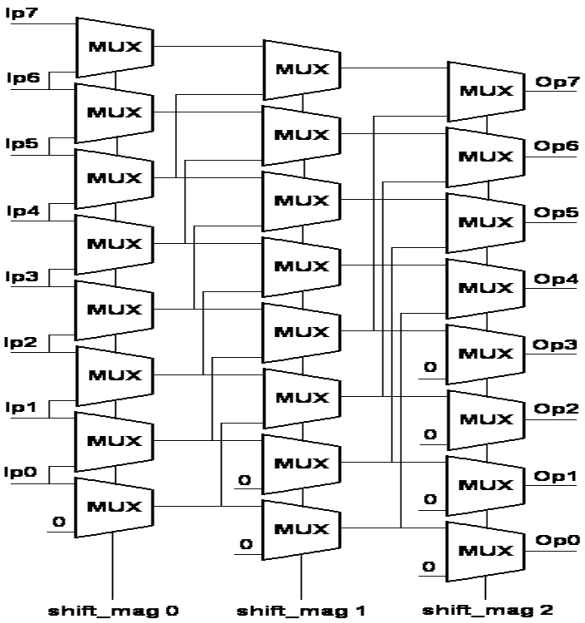


Figura 4.2 Barrel Shifter 8 bits

Inmultirea

Înmulțirea în virgulă mobilă este o operație complexă folosită pentru a multiplica două numere reprezentate în formatul virgulă mobilă, cum ar fi cele din standardul IEEE 754 utilizat pentru reprezentarea numerelor în computer. Procesul implică înmulțirea mantiselor numerelor, adunarea exponentelor și corectarea rezultatului prin ajustarea virgulei și normalizarea rezultatului în formatul standardizat. Această operație necesită gestionarea exponenților și a mantiselor pentru a asigura precizia și acuratețea rezultatului final, având grijă de condiții precum depășirea sau subfluxul, precum și gestionarea bitului de sem

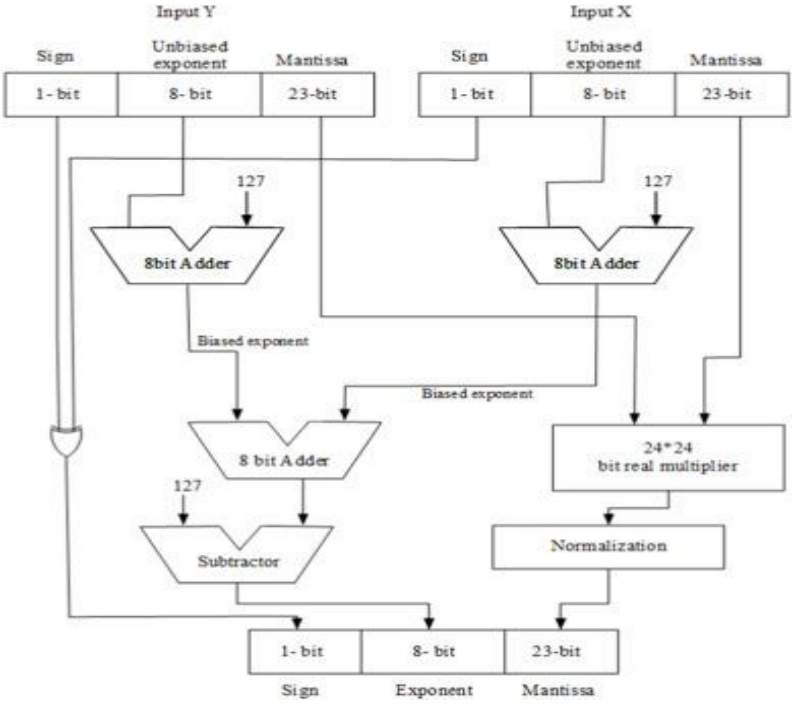


Figura 4.3 Floating Point Multiplier

Principalele componente ale algoritmului de înmulțire vor fi multiplicatorul și seria de adăugătoare utilizate. Multiplicatorul de 24 de biți va fi folosit pentru a înmulți mantisele, la care vom adăuga un 1 ca cel mai semnificativ bit, rezultatul fiind de 48 de biți. Din acest rezultat, dacă primul bit este 1, atunci trebuie să normalizăm, și o vom face luând următorii 23 de biți din rezultatul produsului ca mantisă a rezultatului, și adăugând unu la exponentul rezultatului. Dacă primul bit este 0, atunci rezultatul nu are nevoie de normalizare și putem lua următorii 23 de biți după cel de-al doilea bit (care va fi unu).

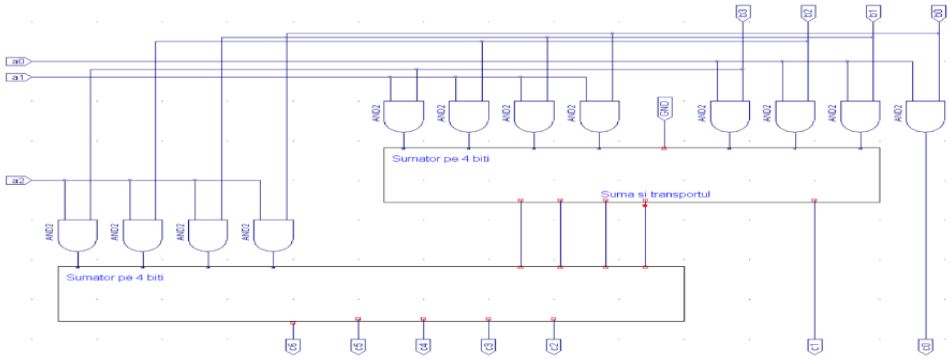


Figura 4.4 Matrice de multiplicare

Exponenții sunt afectați de o anumită diferență (bias), pentru a obține exponentul real vom scădea această diferență (adica 127) din exponenții operanzi, vom aduna rezultatele și apoi vom adăuga înapoi diferența plus bitul de normalizare din înmulțire. Un carry în acest stadiu înseamnă că există o depășire. Semnul rezultatului este pur și simplu XOR-ul semnelor operanzilor.

Pentru componenta de adunare avem nevoie de un adunător de 8 biți. Pentru o creștere mică în viteză, am ales adunătorul "*carry look-ahead*" care depășește problema întârzierii de propagare a carry-ului. Pentru aceasta, folosim adunătoare complete clasice implementate pe baza tabelei de adevăr și o componentă de generare a carry-ului. Sunt folosite unele semnale:

5 Implementare

5.1 Adunare

În cadrul acestei implementări, exponenții și mantisele sunt extrase din intrările X și Y și supuse unui proces complex de manipulare pentru a realiza operația corectă de adunare în virgulă flotantă. Se folosesc componente precum Substract_Result pentru a calcula diferența între exponenți și pentru a determina necesitatea ajustării mantiselor în funcție de această diferență.

Module precum Mux_24, Adder_24Bit, ControlledIncrementor și BarrelShifter sunt integrate în arhitectura acestei componente pentru a gestiona cu precizie exponenții și mantisele, asigurând corectitudinea operației de adunare.

În final, Suma conține rezultatul adunării, respectând formatul specific al numerelor în virgulă flotantă, inclusiv semnul, exponenții calculați și mantisa rezultată din operația de adunare

În implementarea algoritmului de aduanre în virgulă mobilă, am integrat o serie de componente esențiale pentru a realiza operația de adunare a mantiselor pe 24 de biți. Am folosit atât componente de bază precum full adders și half adders pentru a construi adunarea pe 24 de biți a mantiselor numerelor, cât și un sumator cu propagare în cascada succesivă a transportului (cunoscut și sub numele de ripple carry adder) pentru a gestiona corect transportul dintre biții adunați. Full adder-ul combină doi biți de intrare și un carry-in (transport anterior) pentru a genera suma și carry-out-ul (transportul următor), esențial pentru adunarea precisă a mantiselor pe 24 de biți. Apoi, utilizarea unui sumator cu propagare în cascada succesivă a

transportului ne-a permis să gestionăm operația de adunare a celor 24 de biți, având grijă de corectitudinea transportului între fiecare pereche de biți adunați. Astfel, aceste componente sunt fundamentale în procesul de adunare în virgulă mobilă, asigurând precizia și corectitudinea rezultatului final.

```
entity HalfAdder is
    Port (
        A : in  STD_LOGIC;
        B : in  STD_LOGIC;
        S : out STD_LOGIC;
        C : out STD_LOGIC
    );
end HalfAdder;

architecture Behavioral of HalfAdder is

begin

    S <= A xor B;
    C <= A and B;

end Behavioral;
```

Figura 5.1.1 Half Adder

```
entity FullAdder is
    Port (
        A : in  STD_LOGIC;
        B : in  STD_LOGIC;
        Cin : in  STD_LOGIC;
        S : out STD_LOGIC;
        Cout : out STD_LOGIC
    );
end FullAdder;

architecture Behavioral of FullAdder is

signal w1, w2, w3 : STD_LOGIC;

begin

     w1 <= A xor B;
     S <= w1 xor Cin;
     w2 <= w1 and Cin;
     w3 <= A and B;
     Cout <= w3 or w2;

end Behavioral;
```

Figura 5.1.2 Full Adder

Scăderea exponentelor necesită utilizarea unui half subtractor și a unui full subtractor pentru manipularea acestor valori. În cadrul unui ripple-carry subtractor pe 8 biți, structura logica implicată presupune aplicarea operației de scădere bit cu bit între perechile de biți corespondenți din cele două numere, folosind un half subtractor pentru primul bit și un full subtractor pentru restul. Aceste componente sunt esențiale pentru generarea rezultatului și a transportului de la un bit la altul, însă această abordare poate conduce la întârzieri în operațiile de scădere pentru numere mai mari, deoarece transportul trebuie propagat

secvențial de-a lungul bitilor, făcându-l potrivit pentru operații pe 8 biți, dar potențial ineficient pentru numere mai lungi.

```
H1: HalfSubtractor port map  (A=>A(0),B=> B(0),D=> D(0),Bout=> aux(0));
F1: FullSubstractor port map (A=>A(1), B=>B(1), C=>aux(0), D=>D(1), Bout=>aux(1));
F2: FullSubstractor port map (A=>A(2), B=>B(2), C=>aux(1), D=>D(2), Bout=>aux(2));
F3: FullSubstractor port map (A=>A(3), B=>B(3), C=>aux(2), D=>D(3), Bout=>aux(3));
F4: FullSubstractor port map (A=>A(4), B=>B(4), C=>aux(3), D=>D(4), Bout=>aux(4));
F5: FullSubstractor port map (A=>A(5), B=>B(5), C=>aux(4), D=>D(5), Bout=>aux(5));
F6: FullSubstractor port map (A=>A(6), B=>B(6), C=>aux(5), D=>D(6), Bout=>aux(6));
F7: FullSubstractor port map (A=>A(7), B=>B(7), C=>aux(6), D=>D(7), Bout=>aux(7));
```

Figura 5.1.3 Ripple Cary Substractor 8 bits

Pentru verificarea exponentilor am folosit "Substract_Result", care funcționează ca un modul de scădere pe 8 biți între două numere reprezentate în formatul standard IEEE 754. În cadrul acestei entități, sunt integrate două componente interne. Prima, "Subtractor_8Bit", execută operația de scădere între cele două intrări primite și furnizează rezultatul scăderii împreună cu un semnal de ieșire (BOut) care indică dacă rezultatul este negativ sau pozitiv. A doua componentă, "Complement2s", calculează complementul la 2 al valorii pentru a obține rezultatul corect al operației de scădere. Funcționând în mod comportamental, acest modul utilizează aceste componente interne pentru a efectua scăderea între cele două numere de intrare și furnizează rezultatul scăderii pe ieșirea "Res", împreună cu un semnal (BOut) care indică semnul rezultatului, în funcție de valorile primite. Astfel, codul implementează operația de scădere pe 8 biți și furnizează rezultatul, indicând semnul acestuia, bazându-se pe intrările primite.

```
entity Substract_Result is
    Port (
        A : in std_logic_vector(7 downto 0);
        B : in std_logic_vector(7 downto 0);
        Res : out std_logic_vector(7 downto 0);
        BOut : out std_logic
    );
end Substract_Result;

architecture Behavioral of Substract_Result is

signal d, d1 : std_logic_vector(7 downto 0);
signal b_internal: std_logic;

    component Subtractor_8Bit is
        Port (
        A : in STD_LOGIC_VECTOR(7 downto 0);
        B : in STD_LOGIC_VECTOR(7 downto 0);
        D : out STD_LOGIC_VECTOR(7 downto 0);
        Bout : out STD_LOGIC
        );
    end component;

    component Complement2s is
        Port (
          A : in STD_LOGIC_VECTOR(7 downto 0);
          Output : out STD_LOGIC_VECTOR(7 downto 0)
        );
    end component;
begin

Subtractor_Instance: Subtractor_8Bit
        port map (
            A => A,
            B => B,
            D => d,
            BOut => b_internal );

Complement2s_Instance: Complement2s
        port map (
            A => d,
            Output => d1);


Res<=d when b_internal='1' else d1;
BOut <= b_internal;
end Behavioral;
```

Figura 5.1.4 Verificarea exponentilor

Barrel Shifter ajustează biții mantiselor, mișcându-i către stânga sau dreapta în funcție de diferența exponenților numerelor. Această shiftare precisă aliniază mantisele, esențială pentru adunarea și alte operații matematice în virgulă flotantă. Asigură integritatea formatului și precizia în operațiile numerice complexe, asigurând rezultate exacte.

```
entity BarrelShifter is
    Port (
        input : in STD_LOGIC_VECTOR(23 downto 0);
        output : out STD_LOGIC_VECTOR(23 downto 0);
        Shift : in STD_LOGIC_VECTOR(4 downto 0)
    );
end BarrelShifter;

architecture Behavioral of BarrelShifter is
    component Mux is
        Port (
            In0 : in STD_LOGIC;
            In1 : in STD_LOGIC;
            S : in STD_LOGIC;
            Res : out STD_LOGIC
        );
    end component;
    signal a, a1, a2, a3, a4 : STD_LOGIC_VECTOR(23 downto 0);

begin

    Mux_0: Mux port map (In0=>input(0), In1=>input(1), S=>Shift(0), Res=>a(0));
    Mux_1: Mux port map (In0=>input(1), In1=>input(2), S=>Shift(0), Res=>a(1));
    Mux_2: Mux port map (In0=>input(2), In1=>input(3), S=>Shift(0), Res=>a(2));
    Mux_3: Mux port map (In0=>input(3), In1=>input(4), S=>Shift(0), Res=>a(3));
    Mux_4: Mux port map (In0=>input(4), In1=>input(5), S=>Shift(0), Res=>a(4));
```

Figura 5.1.5 Barrel Shifter

Pentru optinerea rezultatului corect final am implementat **ControlledIncrementor,** servește pentru a realiza un increment controlat, adăugând valoarea 1 la un număr reprezentat în formatul virgulă mobile. Este utilizată în operația de adunare în virgulă mobilă pentru a ajusta și actualiza corect exponentul rezultatului în urma operațiilor de adunare a mantiselor numerelor reprezentate în acest format.

5.2 Inmultirea

Algoritmul de inmultire in virgule flotanta reprezintă un algoritm complex pentru înmulțirea a două numere în format virgulă mobilă. Structura modulară și componentele definite separat sunt utilizate pentru a manipula și realiza operațiile necesare. Procesul începe cu preluarea și organizarea exponenților și mantiselor numerelor de intrare (A și B) Aceste valori sunt apoi direcționate către diverse componente specializate precum Adunare_Exponenti, InmultireMantise, Exponenti, Mantise, Normalizare, etc. Fiecare componentă are rolul său specific în manipularea datelor, de exemplu, adunarea exponenților, înmulțirea mantiselor, normalizarea rezultatului, precum și alte operații critice specifice virgulei mobile. Rezultatul final al operației de înmulțire este format din exponențiala rezultantă (exp7), mantisa rezultantă (mant6) și semnul rezultatului (Produs(31)), respectând rigorile și convențiile standardului virgulă mobilă. Astfel, această arhitectură reprezintă un algoritm sofisticat pentru înmulțirea eficientă a numerelor în format virgulă mobilă.

Pentru aceasta implementare am optat pentru o metoda diferita de implementare, am facut component diferite pentru etape diferite de implementare(pentru a reduce numarul de port map-uri) .

"MatrixMultiplication", implementează înmulțirea mantiselor a două numere folosind operația de multiplicare într-o manieră asemănătoare cu o operație de multiplicare a matricelor. În acest caz, valorile din vectorii de intrare **a** și **b** sunt utilizate pentru a calcula înmulțirea mantiselor.

Procesul este realizat prin utilizarea a numeroase sumatoare (reprezentate aici ca Adder_24Bit) care sunt conectate secvențial pentru a realiza operația de multiplicare a fiecărui bit în parte din mantisele numerelor date. Fiecare sumator calculează produsul dintre un anumit bit al mantisei lui **a** și întreaga mantisă a lui **b**, rezultând o parte din produsul final.

Sumatoarele sunt configurate astfel încât să înmulțească fiecare bit al mantisei lui **a** cu toate cele 24 de biți ai mantisei lui **b**, rezultând în final un vector de ieșire **r** de 48 de biți, care conține rezultatul înmulțirii mantiselor. Acest rezultat este obținut prin conectarea succesivă și înmulțirea fiecărui bit în parte și a rezultatelor intermediare obținute.

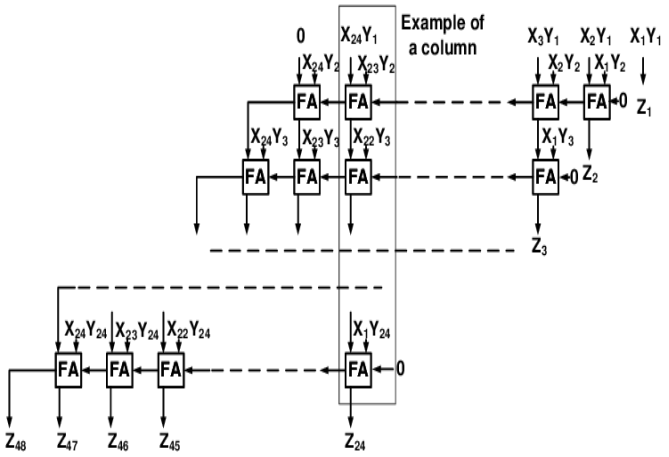


Figura 5.2.1 Matrix Multiplication

"InmultireMantise," efectuează operația de înmulțire a mantiselor a două numere. Prin intermediul unei componente interne, "MatrixMultiplication," intrările mantiselor sunt transmise și înmulțite pentru a genera rezultatul final al înmulțirii.

Modulul primește o mantisă de intrare de 48 de biți și un vector de control "zero" de 2 biți. Pentru a efectua înmulțirea mantiselor, primele 23 de biți din vectorul de intrare sunt utilizați pentru a forma mantisele a și b, cu cel de-al 24-lea bit setat la 1 în ambele cazuri. Această acțiune realizează extinderea semnului, făcând înmulțirea adecvată pentru virgula mobilă.

Operația de înmulțire efectivă a mantiselor se face prin intermediul componentei "MatrixMultiplication," care primește mantisele a și b și returnează rezultatul într-un vector de 48 de biți, stocat în semnalul r.

Rezultatul operației de înmulțire este transmis către ieșirea "mant_out." Dacă mantisele de intrare sunt zero sau, mai precis, dacă toți cei 23 de biți ai mantiselor sunt zero, ieșirea este setată la zero. În caz contrar, semnul rezultatului este calculat prin operația XOR între semnul mantisei de intrare și cel al rezultatului înmulțirii, iar rezultatul înmulțirii este reprezentat de ultimii 23 de biți din rezultatul obținut prin înmulțirea mantiselor.

"Adunare_Exponenti" care efectuează adunarea a două exponenți reprezentați sub formă de vectori de 16 biți. El primește exponenții în intrare și generează rezultatul adunării într-un vector de 8 biți, stocat în ieșirea "r".

Adunarea este realizată pas cu pas, fiecare bit fiind adunat cu atenție, iar carry-ul (depășirea) generat este reținut și folosit în următoarea operație de adunare pentru biți.

Mai întâi, exponenții sunt separați în două vectori, "a" și "b". Apoi, se efectuează adunarea binară clasică pentru fiecare bit al acestor vectori, iar rezultatele sunt stocate în "raux" (rezultatul auxiliar). În paralel, se calculează și carry-ul pentru fiecare operație de adunare și se păstrează într-un vector "c".

Finalizând adunarea pentru fiecare bit al exponenților, se generează rezultatul final al adunării în "r", exceptând situația în care unul dintre exponenți este zero. Dacă unul dintre exponenți este zero, rezultatul adunării este setat la valoarea default a unui exponent nenormalizat, "01111111".

De asemenea, modulul generează un vector de ieșire "zero" de 2 biți pentru a indica anumite condiții speciale întâlnite în adunarea exponenților. Aceste condiții includ cazurile în care ambii exponenți sunt zero, unul dintre ei este zero sau ambii sunt nenuli. Aceste informații pot fi utilizate ulterior în procesarea sau evaluarea datelor în funcție de cazul întâlnit

Figura 5.2.2 Adunare Exponenti

"Check_exponent", primește un exponent de intrare reprezentat printr-un vector de 8 biți și furnizează un exponent de ieșire, tot printr-un vector de 8 biți. Scopul principal al acestei componente este de a verifica și normaliza exponentul de intrare.

Componenta folosește un proces în care verifică dacă exponentul de intrare este diferit de "UUUUUUUU" (care reprezintă o valoare nedeterminată). În cazul în care exponentul de intrare nu este o valoare nedeterminată, componenta procedează la normalizarea acestuia pentru a îl aduce la forma corectă pentru reprezentarea unui exponent în virgulă mobilă.

```
entity check_exponent is
 Port (
    exp_in: in std_logic_vector(7 downto 0);
    exp_out: out std_logic_vector(7 downto 0) );
end check_exponent;

architecture Behavioral of check_exponent is

begin
process(exp_in) begin
        if(exp_in /= "UUUUUUUU") then
            exp_out <= std_logic_vector(unsigned(exp_in)-127);
        end if;
    end process;

end Behavioral;
```

Figura 5.2.3 check exponent

"Normalizare", primește o mantisă de intrare reprezentată printr-un vector de 24 de biți și furnizează o mantisă de ieșire, tot printr-un vector de 24 de biți, împreună cu un ultim bit care indică starea de normalizare.

Primul lucru pe care îl face modulul este să stabilească valoarea ultimului bit din mantisă pe baza celui de-al 23-lea bit al mantisei de intrare.

În continuare, procesul din acest modul tratează două cazuri:

1. Dacă bitul de ordinul 23 al mantisei este '0', indicând că mantisa este normalizată (adică primul bit diferit de zero într-o reprezentare în virgulă mobilă este poziția 23), atunci mantisa de ieșire va păstra exact aceeași structură ca mantisa de intrare. Se mută bitul 23, se păstrează biții de la 22 la 2 și ultimii doi biți și sunt setați la '0', conform convențiilor virgulei mobile.
2. Dacă bitul de ordinul 23 al mantisei este '1', ceea ce indică o mantisă care necesită normalizare, se modifică mantisa de ieșire astfel încât să fie reprezentată în forma normalizată. În acest caz, bitul 23

este copiat în mod direct în mantisa rezultatului (Biții de la 21 la 0 din mantisa de intrare sunt mutați în mantisa de ieșire , iar ultimul bit și cel mai puțin semnificativ bit sunt setate la '0'.

Practic, acest modul asigură că mantisa rezultatului este într-o formă normalizată pentru a respecta regulile și convențiile reprezentării în virgulă mobilă. Ultimul bit furnizează informații despre starea de normalizare a mantisei de ieșire.

"Shift_Exponent", primește un exponent reprezentat printr-un vector de 8 biți și un bit care indică starea finală a normalizării .Iar ca ieșire, furnizează un nou exponent reprezentat tot printr-un vector de 8 biți.

Procesul din acest modul examinează starea de normalizare . Dacă aceasta este '1', indicând că normalizarea mantisei a necesitat o schimbare în exponent, modulul realizează o operație de incrementare a exponentului de intrare.

Practic, dacă starea de normalizare este setată la '1', modulul incrementează valoarea reprezentată de exponentul de intrare cu 1 prin conversia valorii la întreg și apoi returnarea acestei valori incrementate în forma sa originală, ca un vector de 8 biți.

În caz contrar, când starea de normalizare este '0', adică nu este necesară nicio modificare a exponentului, modulul returnează pur și simplu valoarea existentă a exponentului de intrare prin ieșire.

```
entity Shift_Exponent is
 Port (
    exp_in: in std_logic_vector(7 downto 0);
    last: in std_logic;
    exp_out: out std_logic_vector(7 downto 0)
 );
end Shift_Exponent;

architecture Behavioral of Shift_Exponent is

begin
process(exp_in,last) begin
    if(last = '1') then
        exp_out <= std_logic_vector(unsigned(exp_in) +1);
    else
        exp_out <= exp_in;
    end if;
    end process;

end Behavioral;
```

Figura 5.2.4 Shift exponent

Pentru a putea utiliza cele 2 operatii impreuna, am implementat o componenta speciala “FloatingPointALU” care primeste un flag special pentru a indica ce operatie urmeaza sa fie realizata, aceasta implementare este folosita pentru a efectua o simulare mai complexa a operatiilor prezentate.

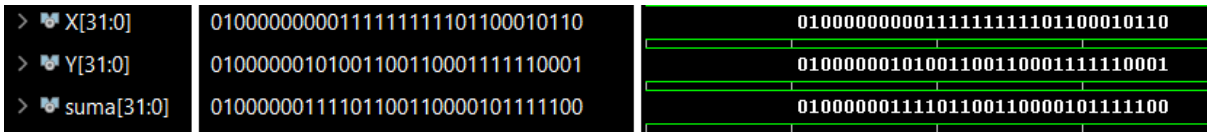
6 Testare

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.

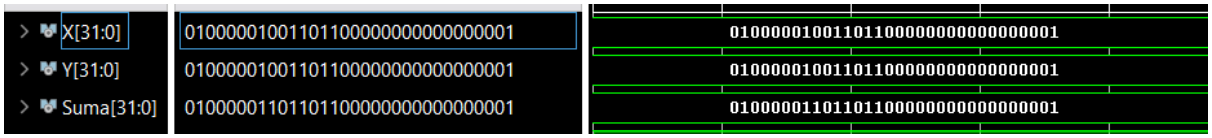
Adunare

Datorita implementarii se va putea efectua operatii pe numere cu acelasi semn deoarece, practice, adunarea unor numere cu semn diferit ar reprezenta o scadere.

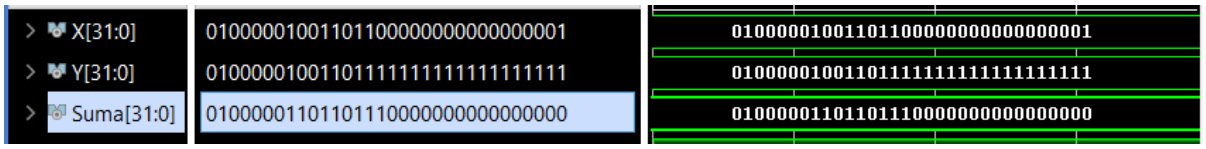
Test1: X=2.5, Y=5.2 => Suma=7.7



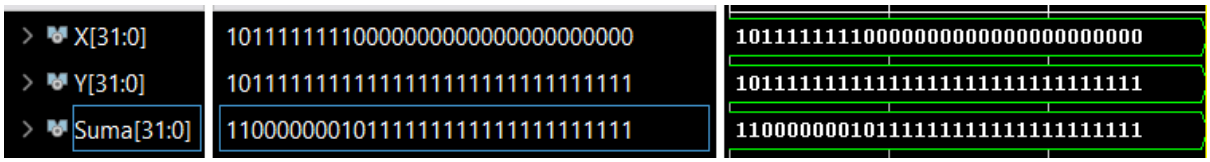
Test2: X=11.375, Y=11.375 =>Suma= 22.75



Test3: X= 11.375, Y= 11.5=> Suma= 22.875



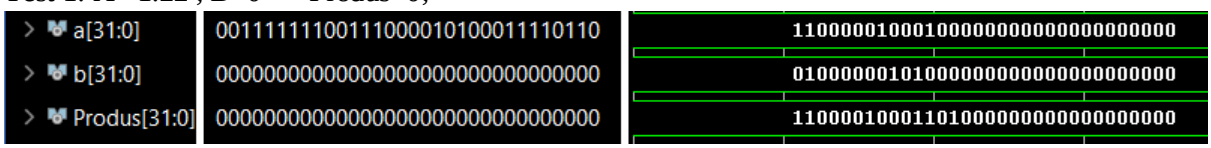
Test4: X= -1.5, Y= -1.999 => Suma= -3.4999



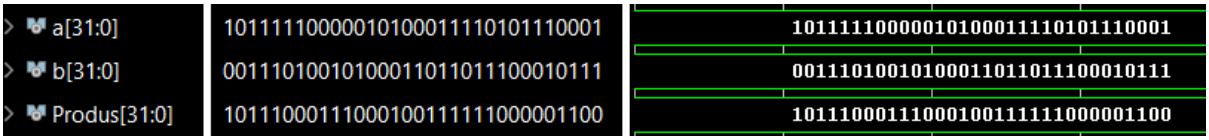
6.1 Inmultire

Implementarea inmultiri difera de cea de adunare si astfel se potate opera cu numere cu semen diferite.

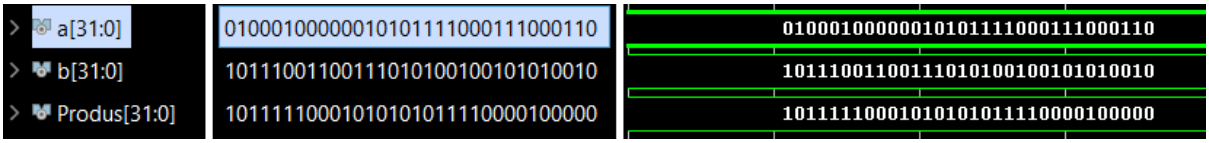
Test 1: A= 1.22 , B=0 => Produs=0;



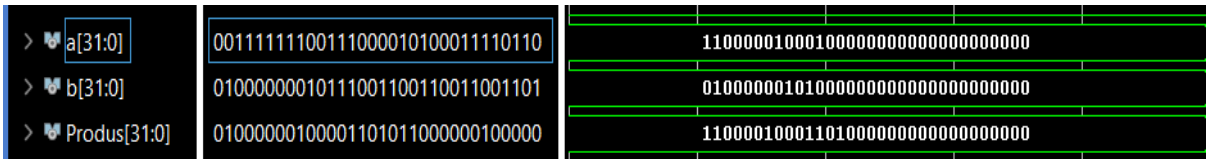
Test2: A= -0.135, B= 0.0008 => Produs= -0.00010799



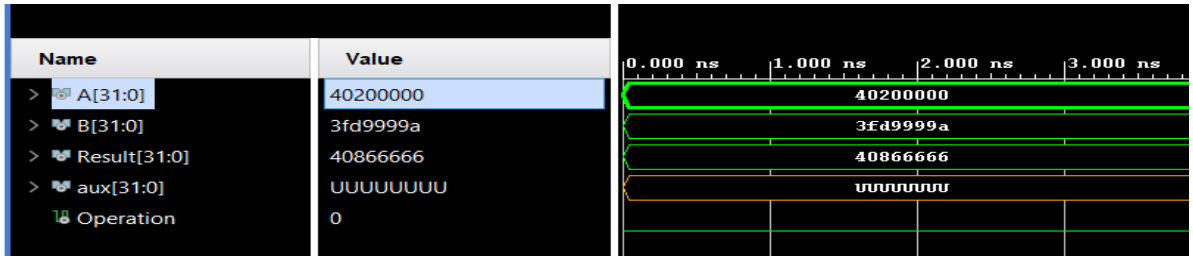
Test3: A= 555.7777, B= -0.0003 => Produs=-0.1667



Test3: A=1,22 , B= 3.45 => Produs =4.2



In final am test si componeta “FloatingPointALU” care cuprinde cele doua operatii pentru a putea lucra cu ambele operatii:
Am inceput cu o adunare

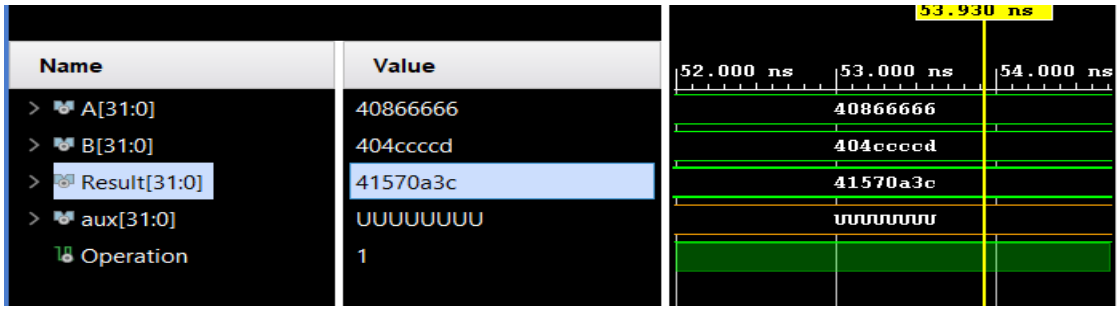


Operatia1: 2.5 + 1.7 = 4.2

```
A<="01000000001000000000000000000000"; --2.5
B<="00111111110110011001100110011010"; --1.7
Operation <= '0';
wait for 50 ns;


A<=Result; --4.2
B<="01000000010011001100110011001101"; --3.2
Operation <= '1';
wait for 50 ns;

A<=Result; --13.44
B<="01000000000011001100110011001101";--2.2
Operation <= '1';
wait for 50 ns;

A<=Result; --29.568
B<="01000000000011001100110011001101";--2.2
Operation <= '1';
aux<=Result;
wait for 50 ns;
```

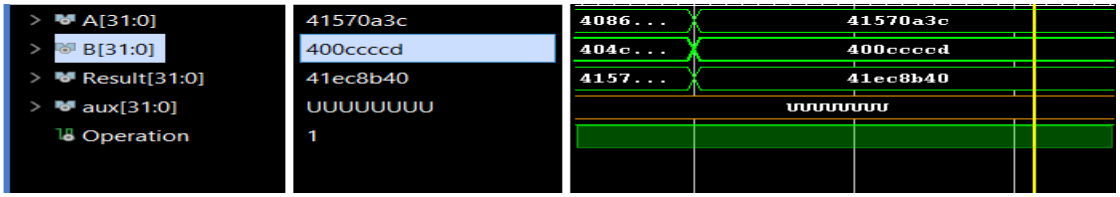
Fig 6.1. Cod testare ALU

Dupa cum se poate observa, urmatoarea operatie este inmultirea intre rezultatul anterior si 4.3.



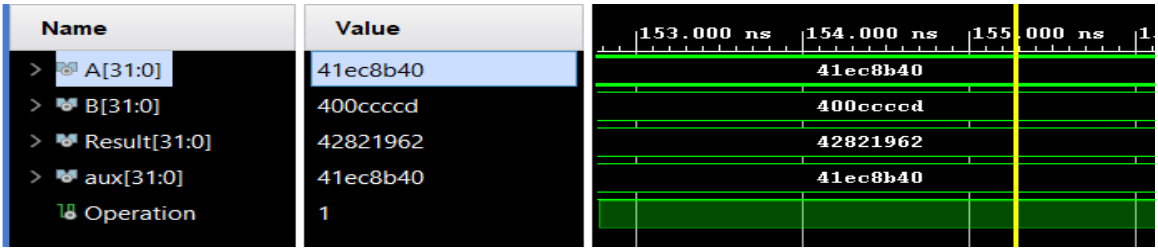
Operatie 2: 4.2 * 3.2 =13.4399

Am continua cu inmultirea



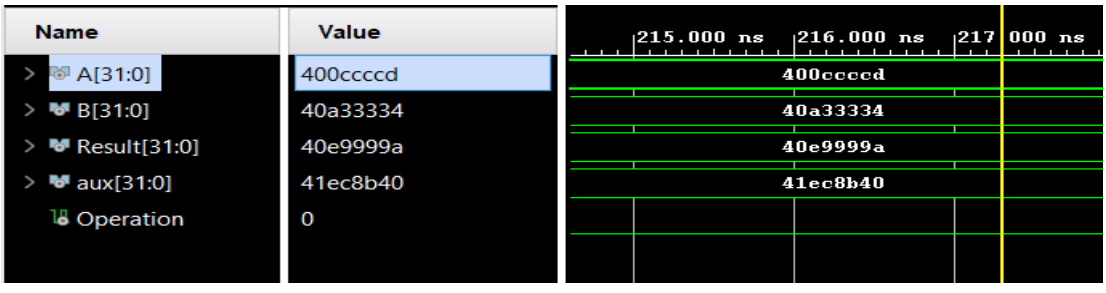
Operatie 3: 13.4399 * 2.2 = 29.567

Am stocat in variabila aux rezultatul de la operatia 3



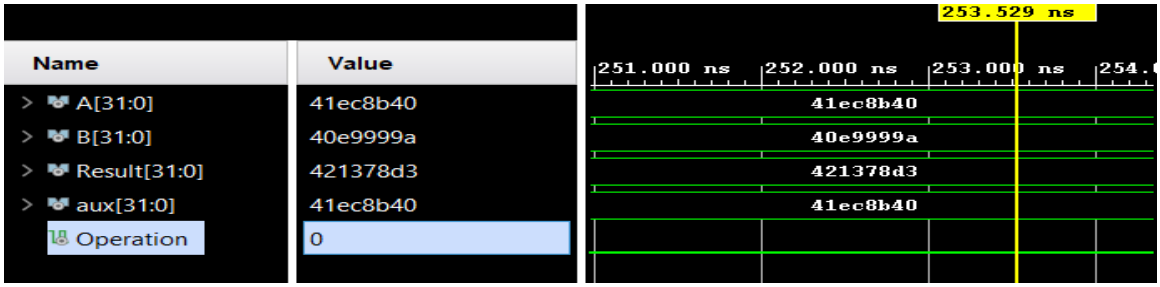
Operatia4: 29.567 * 2.2 = 65.0495

Dupa care am facut o adunare oarecare, fara a tine cont de rezultatul anterior, am pastrat variabila aux



Operatia 5: 2.2 + 5.1 = 7.3

Am pastrat rezultatul anterior de la operatia 5 in variabila A pe care l am adunat cu variabila aux



Opeatia 6: 7.3 + 29.567 = 36.867

Ultima operatie consta in inmultirea rezultatului de la operatia 7 cu o valoare

Operatia 7: 36.867 * 3.25 = 119.82

7 Concluzii

Implementarea algoritmilor de adunare și înmulțire în virgulă mobilă este fundamentală pentru o varietate de domenii, inclusiv calculul numeric, grafică computerizată, inginerie și multe altele. Abordarea pentru realizarea acestor operații necesită atenție deosebită și precizie în manipularea detaliilor specifice reprezentării numerelor în formatul virgulă mobilă conform standardului IEEE 754.

Manipularea precisă a exponenților și mantiselor este esențială pentru a asigura corectitudinea și precizia operațiilor numerice. Respectarea standardelor impuse de IEEE 754 asigură consistența rezultatelor și compatibilitatea între diferite platforme și implementări.

Interacțiunea corectă și integrarea eficientă a componentelor cheie, precum adunătoarele, înmulțitorii, verificarea și ajustarea exponenților, normalizarea și verificarea mantiselor, sunt cruciale pentru obținerea rezultatelor precise și fiabile. Componentele trebuie să funcționeze împreună în armonie, respectând fiecare etapă a

În concluzie, implementarea algoritmilor de adunare și înmulțire în virgulă mobilă necesită nu doar expertiză tehnică, ci și o atenție deosebită la detalii. Respectarea standardelor, integrarea eficientă a componentelor cheie și testarea riguroasă sunt pilonii esențiali pentru a asigura precizia, corectitudinea și fiabilitatea operațiilor numerice în virgulă mobilă.

8 Dezvoltari ulterioare

Extinderea funcționalității pentru a include operațiile de scădere și împărțire în virgulă mobilă necesită o abordare detaliată și adaptare precisă a algoritmilor în conformitate cu specificațiile IEEE 754. Aceste operații sunt complexe, necesitând ajustări considerabile pentru manipularea precisă a mantiselor și a exponenților.

Implementarea algoritmului de scădere în virgulă mobilă necesită gestionarea cu grijă a situațiilor speciale și a pierderii de cifre semnificative în mantisă. Împărțirea, pe de altă parte, necesită un algoritm specializat pentru a gestiona corect împărțirea precisă a numerelor.

În ceea ce privește adaptarea pe placa de dezvoltare Bays3, trebuie să abordăm interacțiunea între software și hardware pentru a obține performanță și compatibilitate optimizate. Acest proces necesită o înțelegere profundă a arhitecturii hardware-ului Bays3 și optimizări specifice pentru a gestiona eficient operațiile numerice complexe.

Eșecurile anterioare în adaptarea pe Bays3 reprezintă o sursă valoroasă de învățare, oferind indicii despre provocările tehnice întâmpinate. Astfel, dezvoltarea ulterioară necesită o analiză detaliată a eșecurilor anterioare pentru a identifica și a remedia în mod specific problemele întâmpinate în integrarea hardware-ului și a software-ului.

Bibliografie

[1]. "Single-precision floating-point format," [Online]. Available:

https://en.wikipedia.org/wiki/Singleprecision_floating-point_format.

[2]. E. Sanchez, "A Floating-Point Multiplier," [Online]. Available:

http://lslwww.epfl.ch/pages/teaching/cours_lsl/sl_info/FPMultiplier.pdf.

[3]. P. K. B. S. R.P.P. Singh, "Performance Analysis of fast adders," [Online]. Available:

https://www.researchgate.net/publication/220849794_Performance_Analysis_of_Fast_Adders_U

sing_VHDL.

[4]. "rf wireless-wordl," [Online]. Available: http://www.rfwireless-world.com/Tutorials/floating-pointtutorial.html.

[5]. "floating point arithmetic on division," [Online]. Available:

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

[6]. "Adunarea şi scăderea în virgulă mobilă" [Online]. Available:

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

[7]. Gustavo Sutter, Diego Sanchez-Roman, "Floating Point FPGA Cores: Multiplication and Addition" [Online]. Available:

https://www.researchgate.net/publication/260813505_Floating_Point_FPGA_Cores_Multiplication_and_Addition

[8]. Concordia University "Floating Point Adders and multipliers " [Online]. Available:

http://users.encs.concordia.ca/~asim/COEN_6501/Lecture_Notes/L4_Slides.pdf

[9] Baruch Zoltan Francisc, "Structura sistemelor de calcul", Editura U. T. PRES, Cluj-Napoca, 2002.