FPU: ÎNMULȚIRE ȘI ÎMPĂRȚIRE

Șerban Alexandra

Grupa 30234

Cuprins

1. Introducere

1.1. Context

Proiectul are drept scop proiectarea, implementarea și testarea a 2 operații: înmulțire și împărțire în virgulă flotantă în formatul standard IEEE pe 32 de biți. Numerele flotante cu simplă precizie vor putea fi nu doar implementate ci și vor putea fi folosite în aceste 2 operații, înmulțirea și împărțirea fiind operații mai complexe decât adunarea si scăderea. Înmulțirea durează ca timp de execuție mai puțin decât împărțirea deoarece cea din urmă necesită scaderi iterative ce nu pot fi executate simultan.

Proiectul se va prezenta sub forma unui calculator ce va putea fi integrat și folosit în unitățile de execuție mai exact in unitatea centrală de procesare sau legat ca un dispozitiv periferic conectat la un master. De asemenea, poate fi foarte bine integrat într-un procesor sau microprocesor care pot implementa aceste operații.

1.2. Specificații

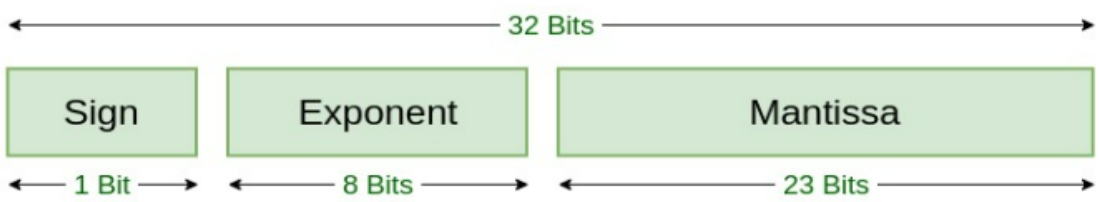
Dispozitivul va fi simulat într-un IDE dezvoltat de vivado și va fi programabil pe o plăcuță de dezvoltare. Numerele vor trebui reprezentate in Standard IEEE, convertite în complement față de 2 cu 16 biți pentru partea întreagă și 16 biți pentru partea fracțională, și de asemenea vor trebui să poată fi înțelese de către utilizator într-un anumit format.

1.3. Obiective

Obiectivele sunt acelea de a reuși proiectarea și implementarea complementului față de 2 în virgulă flotantă și de a afișa rezultatul operației. Trebuie să pot extrage părțile necesare din aceste tipuri de numere normalizându-le astfel încât să găsesc echivalentul acestora pentru ca dispozitivul să poată să realizeze operațiile bazate pe alegerea utilizatorului și mai apoi când operația este finalizată, aceasta să fie afișată pe un ecran cu 7 segmente.

2. Studiu bibliografic

Standardul IEEE pentru aritmetica numerelor flotante tehnic este un standard pentru compunerea numerelor flotante care a abordat multe probleme găsite în diverse implementări care au îngreunat în mod fiabil și au redus portabilitatea. Structura pentru numere flotante este următoarea:



De exemplu pentru numărul 12.345 reprezentarea ar fi:

$$12.345 = \underbrace{12345}_{\text{significand}} \times \underbrace{10^{\overbrace{-3}^{\text{exponent}}}}_{\text{base}}$$

În practică, majoritatea numerelor folosesc baza 2, deși baza 10(decimal floating point) este de asemenea comună.

Din studiul de caz ce l-am realizat asupra operațiilor de înmulțire și împărțire, am ajuns la 4 etape principale în realizarea algoritmului:

- Înmultirea fără semn a mantiselor celor 2 numere: numărul rezultat va trebui să aibă o dimensiune dublă de 48 de biți
- Normalizarea rezultatului (consecință: exponentul poate fi modifcat)
- Adunarea exponenților luând în considerare și bias-ul (pentru un exponent de 8 biți, acesta este de 127)
- Calcularea semnului

Rezultatul final va fi combinația acestor 4 pași.

$X_n=(-1)\times 2^{-7} \Rightarrow X=(-1)^{1}\times 2^{+120}\times(1.0) \Rightarrow X_P=1.01111000.0...0$ (hidden; sign exponent mantisa)

$Y_n=(+1)\times 2^{5} \Rightarrow X=(-1)^{0}\times 2^{+132}\times(1.0) \Rightarrow X_P=1.01111000.0...0$

$Z_P=(-1)^{1\oplus 0}\times 2^{Z_{EP}}\times(1.0)$ $\quad Z_Q=(-1)^{1\oplus 0}\times 2^{Z_{EQ}}\times(1.0)$

$Z_{EP}=+120+132-127$ (bias) $\quad Z_{EQ}=+120-132+127$ (bias)

$X_E=01111000$ $\quad X_E=01111000$

$+Y_E=10000100$ $\quad +Y_{EC2}=01111100$

$+127_{C2}=10000100$ $\quad +127_{SM}=01111111$

$Z_{EP}=01111101=+125$ $\quad Z_{EQ}=01110011=+115$

$Z_{PP}=1.01111101.0...0$ (sign exponent mantisa) $\quad Z_{QP}=1.01110011.0...0$ (sign exponent mantisa)

$Z_{Pn}=(-1)\times 2^{+125-127}=(-1)\times 2^{-2}$ $\quad Z_{Qn}=(-1)\times 2^{+115-127}=(-1)\times 2^{-12}$

Împărțirea se realizează in mod similar înmulțirii cu mici modificări precum: înlocuim adunarea cu scădere și înmulțirea cu împărțire. În unele cazuri, împărțirea întreagă poate fi înlocuită cu o înmulțire reciprocă, dar asta implică stocarea unei valorid e tip LUT pentru toate posibilitățile de reciprocitate și cum operanzii noștri sunt mantise asta inseamnă că va trebui să compunem 2^23 de posibilități, ceea ce este destul de mult. Cum acest lucru ar necesita foarte mult, am decis să implementez împărțirea exact așsa cum se face pe hârtie.

Pentru o implementarea cât mai eficientă vom folosi half-precision pe 16 biți.

3. Analiza operațiilor

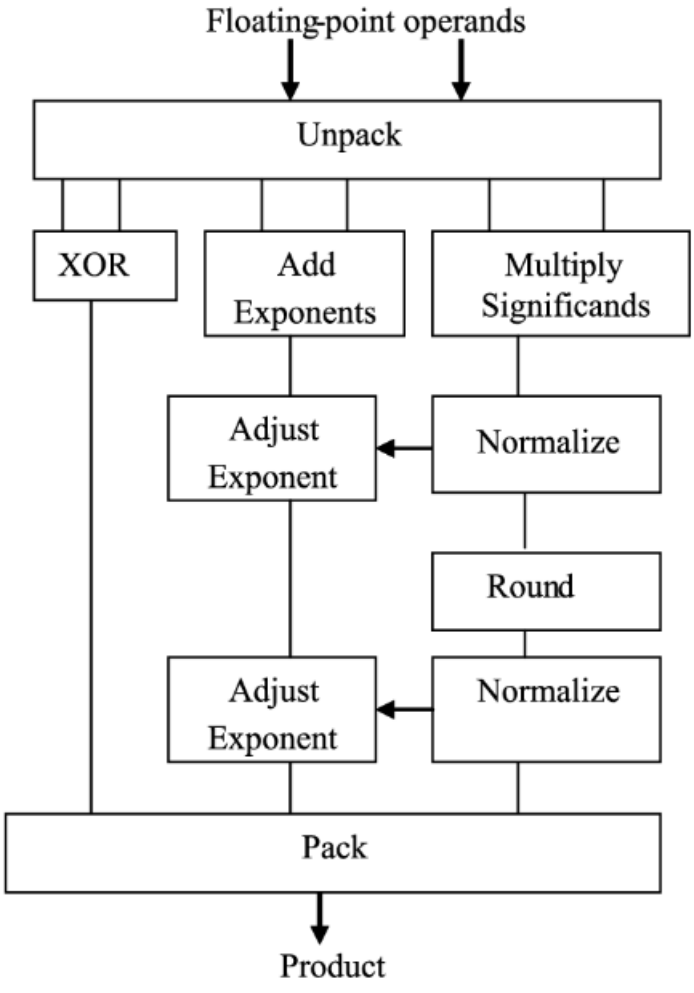
3.1. Înmulțirea

Analizând algoritmul pentru realizarea înmulțirii, putem observa că pentru proiectarea algoritmului avem nevoie de un sumator pe 8 biți și de un multiplicator pentru numere de 24 de biți. Pentru a face cât mai ușoară implementarea acestei operații, avem nevoie ca sumatorul să fie unul cât mai eficient pentru ca și celelalte componente să funcționa cât mai bine.

Primul pas este acela de a aduna exponenții celor 2 numere ținând cont și de bias.

După aceea trebuie să multiplicăm mantisele. Pentru asta trebuie să extragem mantisele operanzilor, concatenăm un ‚1' pentru cel mai semnificativ bit pentru ambele numere, iar mai apoi le trimitem ca input la componenta pentru multiplicare. Dacă primul bit al rezultatului este 1, atunci va trebui să ne împrumutăm cu 23 de biți din rezultatul mantisei, ceea ce înseamnă că va trebui să adăugăm 1 exponentului pentru operația de normalizare. Dacă primul bit este 0, atunci al 2-lea bit va fi mereu 1, deci va trebui să extragem 23 de biti de de la al 3 lea bit.

Acum că avem ambii exponenți si mantisa rezultat, tot ce rămâne de făcut este să calculăm semnul cu ajutorul operației de XOR dintre semnul operanzilor.

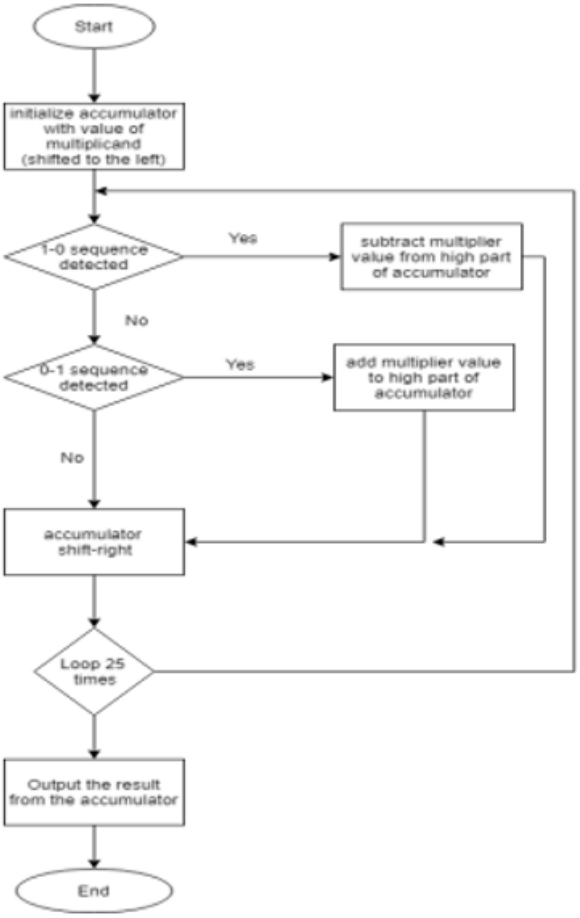


Există și câteva dezavantaje ale acestei abordări precum:

- Overflow la exponent: dacă cei 2 operanzi sunt foarte mari, atunci adunarea exponenților va duce la overflow, rezultând un număr foarte mic
- Underflow la exponent: dacă cei 2 operanzi sunt foarte mici rezultatul poate fi arătat ca un număr eronat foarte mare

Pentru componenta de adunare există diverse soluții precum ripple carry adder care are o implementare ușoară si care funcționează pentru numere cu puțini biți. Ca totul să mearga mult mai rapid am decis să folosesc un sumator mult mai rapid. Am ales să implementez un sumator look-ahead. Acesta mă va ajuta să propag întârzierile pentru împrumut generând carry-ul în avans.

Înmulțirea va fi puțin mai complexă. Există o mulțime de moduri în care poate fi proiectată operația de înmulțire. Un design simplu ar dura prea mult având de multiplicat numere pe 24 de biți. O altă metodă eficientă este utilizarea algoritmului Wallace Tree care este optimizat, dar care ar fi prea costisitor pentru numere atât de mari. De aceea, voi alege să implementez algoritmul Booth care funcționează pentru numere cu semn. Cum numerele noastre sunt unele fără semn, vom concatena un ‘0’ la mantisa noastră pentru a fi mereu numere pozitive. Dacă vor exista schimbări de semn algoritmul va face o adunare sau o scădere in afara shiftării.

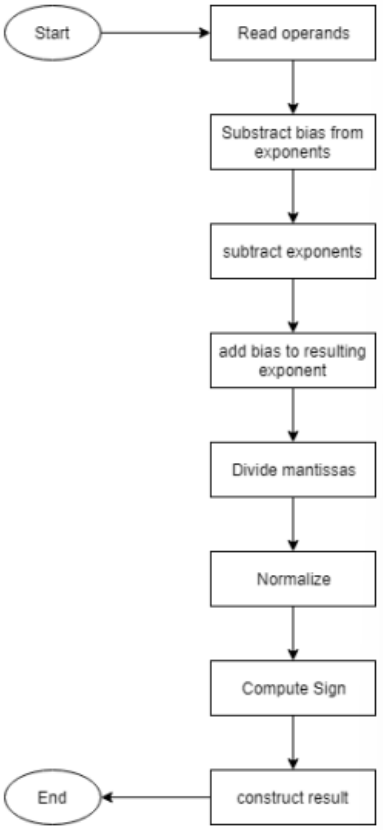


3.2. Împărțirea

Din analiza numerelor flotante putem observa că algoritmul pentru această operație este aproape identic cu cel de înmulțire dacă privim dintr-o perspectivă de nivel inalt. La această operație avem de făcut aproximativ aceleași operații doar că vom elimina bias-ul de la exponent, se realizează operația de scădere de la operația anterioară si adunăm bias-ul la forma exponentului rezultatului final.

Pasul următor constă în crearea mantisei. Selectăm mantisa de la operand îi adăugăm un 1(zecimal imaginar) și împărțim mantisele. Apoi urmează să normalizăm dacă este cazul și într-un final avem mantisa rezultat.

Pasul final constă în alcătuirea semnului rezultatului. Acest lucru se va face cu un simplu XOR dintre cele 2 numere inițiale.



Precum am întâmpinat probleme la înmulțire, există câteva probleme și la această operație de împărțire:

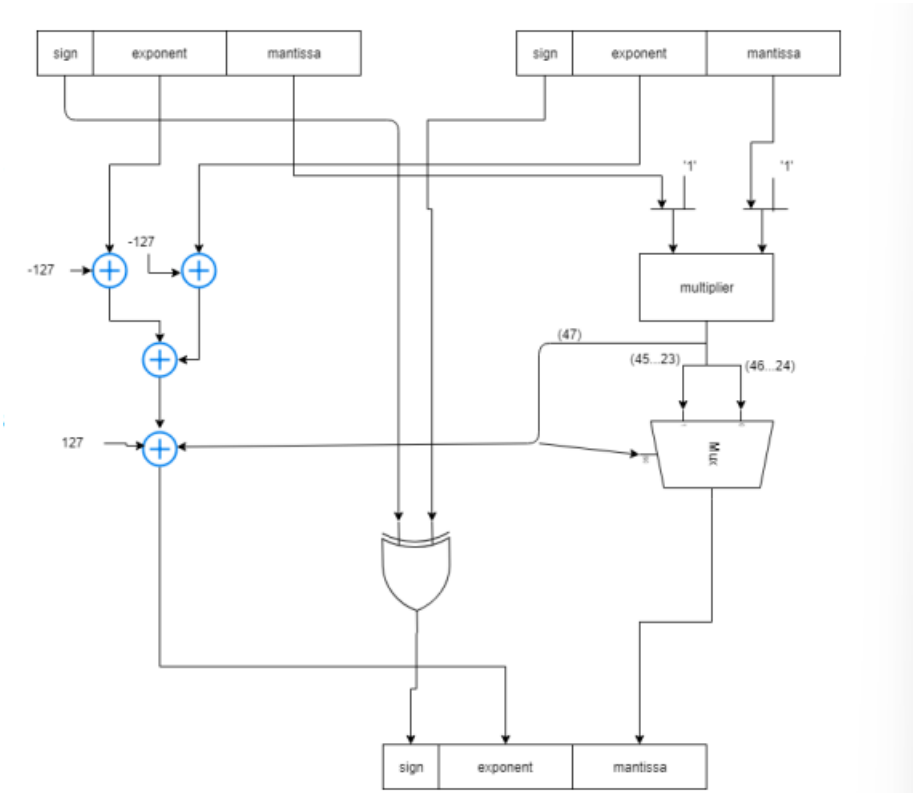
- Împărțirea cu 0 va duce la o eroare
- Overflow la exponent: dacă împărțim un număr foarte mare cu unul foarte mic, rezultatul scaderii exponentului poate duce la overflow și astfel să fie arătat un număr foarte mic în loc de unul foarte mare
- Underflow la exponent: aceeși operație doar că invers- împărțim un număr foarte mic cu unul foarte mare

Acesta este un algoritm dintr-o perspectivă de nivel înalt. Urmează să definim modul în care operațiile intermediare funcționează, dar cum scăderea este doar o operație de adunare putem să omitem prezentarea uneia dintre ele și ne vom axa pe cum se realizează împărțirea.

Ultimul lucru legat de operația de împărțire este acela că nu am găsit nicio implementare satisfăcătoare așa că am decis să realizez o implementare proprie în care împărțirea se realizează ca pe hârtie, unde se testează daca deînmulțitul încape în împărțitor, dacă da shiftăm un 1 la rezultat și apoi scădem împărțitorul de la deîmpărțit, dacă nu shiftăm un 0 rezultatului, iar pentru pasul următor vom lua în considerare un bit suplimentar până când depășim precizia semnelor noastre de 23 de biți.

3.3. Design

Părțile principale pentru algoritmul de multiplicare vor fi alcătuite dintr-o serie de sumatoare. Multiplicatorul de 24 de biți va fi folosit pentru a înmulți mantisele, la care vom adăuga un 1 deoarece rezultatul celui mai semnificativ bit este pe 48 de biți. Din acest rezultat, daca primul bit e 1, va trebui sa normalizăm luând următorii 23 de biți de la rezultatul produsului ca mantisa rezultată și vom adăuga unu la rezultatul exponentului. Dacă primul bit e 0 atunci rezultatul nu va trebui normalizat și vom putea lua următorii 23 de biți după al 2-lea bit(care va fi 1).



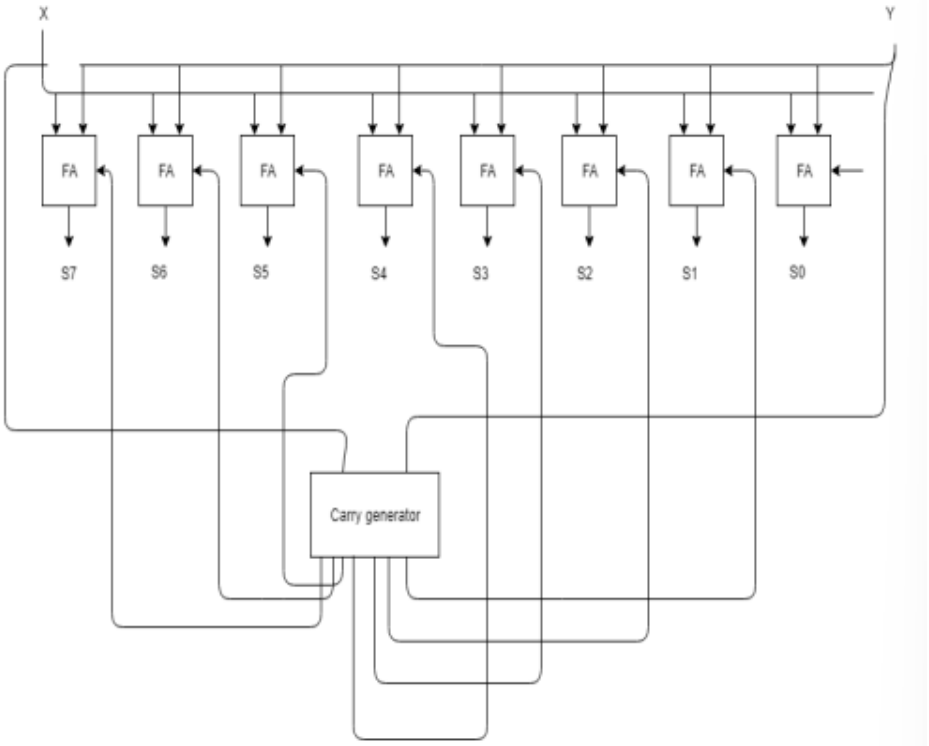
Exponenții conțin biasul. Pentru a putea avea exponentul actual va trebui sa extragem biasul (127) de la exponentul operanzilor, adunăm rezultatele și la final adăugăm biasul împreună cu bitul de normalizare de la multiplicare. Dacă avem împrumut(carry)

înseamnă ca a avut loc overflow. Semnul rezultatului este un simplu xor între semnele operanzilor.

În cazul împărțirii ideea este asemănătoare. Pentru exponenți avem de făcut scădere în loc de adunare, dar putem păstra componentele și doar să obținem complementul față de 2 al celui de al 2 lea operand. Mai apoi în loc de multiplicator ne va trebui o împărțire cu puncte fixe care va deveni următoarea provocare în pasul final de design al componentelor.

Pentru componenta de adunare vom avea nevoie de un sumator pe 8 biți. Pentru a face algoritmul mult mai eficient am ales să implementez sumator carry look-ahead care va trece peste problema de propagare a întârzierii împrumutului. Pentru acesta se va folosi un sumator complet clasic implementat pe baza tabelului de adevăr și o componentă pentru transport. Câteva semnale importante:

- Un semnal compus generat cu formula g(i) = X(i) and Y(i)
- Un semnal compus propagat compus cu formula p(i) = X(i) or Y(i)
- Împrumutul pentru următorul sumator compus astfel: C(i+1) = g(i) or (p(i) and C(i))



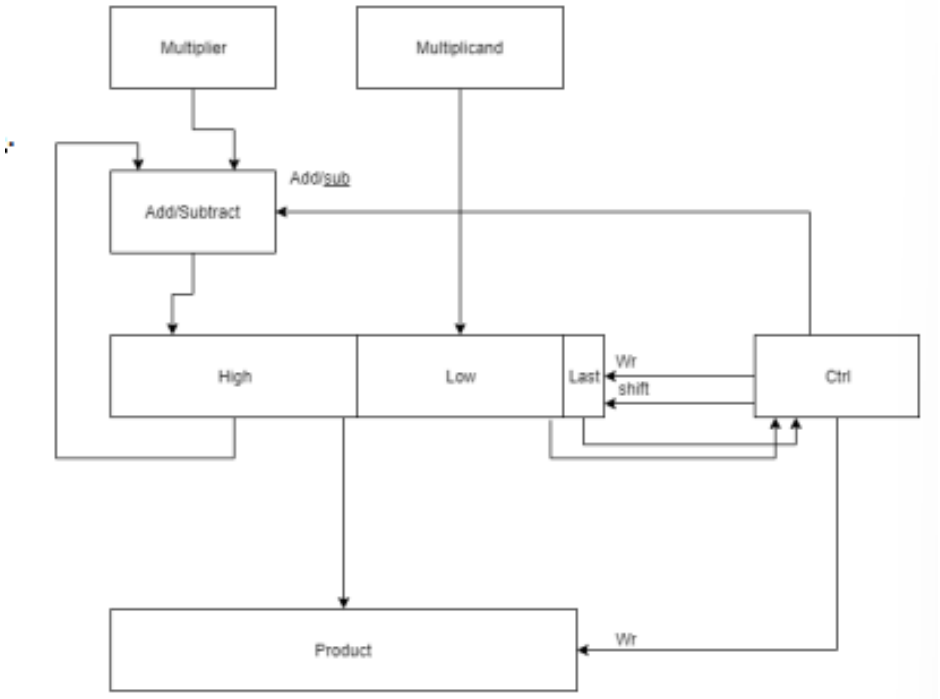
Ce mai rămâne de făcut este designul componentei de înmuțire. Algoritmul folosit a fost descris în partea de analiză în organigramă. Vom proiecta o componentă pentru înmulțirea a 2 numere de 25 de biți.

Pentru aceasta vom avea nevoie de un acumulator pentru a reține rezultatul operației de shiftare și acre va reține și rezultatele intermediare. Acesta va avea dimensiunea necesară standard de MxN, unde M și N sunt dimensiunile operanzilor, în

ambele cazuri 25 de biți, și un bit pentru a stoca ultimul bit shiftat pentru a verifica dacă a avut loc vreo schimbare din 0 în 1 sau invers.

Primul bit este inițial 0, partea cea mai slabă a acumulatorului este inițializată cu deînmulțitul iar mai apoi putem da startul buclei principale care se va executa de 25 de ori.

La fiecare iterare verificăm dacă a avut loc vreo schimbare de biți din 1 în 0, iar dacă a avit loc atunci stocăm în high part al acumulatorului diferența dintre el însuși și multiplicator. Dacă detectăm cealaltă schimbare atunci se efctuează adunarea in locul scăderii. Urmează o shiftare aritmetică la stânga (păstrăm semnul) indiferent dacă o detecție de schimbare a fost făcută. După ce toate iterațiile sunt gata, ce rămâne de făcut este să scriem părțile high și low ale acumulatorului produsului final.



Pentru împărțire implementarea va fi aproape identică cu cea din prima figură al acestui capitol, singura diferență va fi aceea că exponenții vor fi trecuți în complement pentru a putea fi extrase valorile lor negative iar mai apoi vor fi încărcați în sumatoare, iar operațiile de adunare vor fi tranformate în operații de scădere.

O altă diferență va fi componneta de multiplicare care va fi înlocuită cu o componentă de împărțire care de data aceasta va scoate direct matisa rezultatului și de asemenea valoarea normalizată care ne va spune de câte ori va trebui să reducem exponentul pentru ca rezultatul să fie normalizat, iar această valoare va fi scăzută din valoarea finală a exponentului.

Proiectarea componentei de împărțire a fost partea cea mai grea din acest proiect. Ideea este de a clona metoda de împărțire pe care o folosim și pe hârtie atunci când facem această operație. Începem prin a găsi în numitor cel mai semnificativ ‘1’ și știm că în acea poziție va fi punctul fracționar din rezultat și luăm partea high până la această poziție de la numitor. Cu aceasta incepem să impunem această parte a numitorului pe numărător

începând cu cel mai semnificativ bit. Verificăm dacă numitorul se potrivește în această parte a numărătorului (adică acea parte a numărătorului este mai mare decât numitorul nostru actual). Dacă este adevărat atunci shiftăm un 1 la rezultat, scădem din numărător numitorul și punem rezultatul înapoi în numărător la poziția la care s-a realizat scăderea. Dacă numărătorul nu este mai mare decât numitorul, atunci nu putem scădea și doar deplasăm un 0 la rezultat. Apoi ne mutăm cu o poziție la dreapta în numărător și repetăm pașii. Numărătorul a fost dublat în mărime și concatenat la stânga cu 0 pentru a obține o precizie mai mare. La final, căutăm cel mai smenificativ bit cu valoarea 1 și pentru normalizare trebuie să punem virgula mobilă după acets punct și diferența între punctul inițial găsit și puncut din rezultat reprezintă valoarea de normalizare pe care trebuie să o scădem din exponentul final.

4. Implementare

Implementarea se va face conform primei scheme din capitolul de Design. Fiecare componentă va fi făcută in mod separat in programul Vivado urmând ca mai apoi să fie mapate într-un program principal și testate cu ajutorul modului de simulare.

Sumatorul complet pe 8 biți va fi realizat cu ajutorul unui carry generator integrat în codul acestuia. Carry generaterul a fost realizat cu ajutorul unor loopuri în care am generat semnale pentru fiecare poziție. Se putea face și altfel: implementând un generator de un bit și mai apoi o cascadare, dar acest lucru ar fi necesitat un efort suplimentar.

Componenta de înmulțire se face într-o manieră comportamentală cu un contor pentru lungimea de biți ai intrătilor. Există un acumulator variabil care are lungimea ieșirii +1 bit suplimentar pentru a stoca valoarea deplasată. Apoi la fiecare pas verificăm dacă există o schimbare de la 0 la 1 sau invers în ultimii 2 biți ai acumulatorului. Dacă există, adunăm sau scădem la partea high depinzând de schimbare. Indiferent dacă a existat o schimbare sau nu schimbăm tot acumulatorul. Au fost folosite niște variabile suplimentare pentru a stoca și face operațiuni.

5. Teste și validări

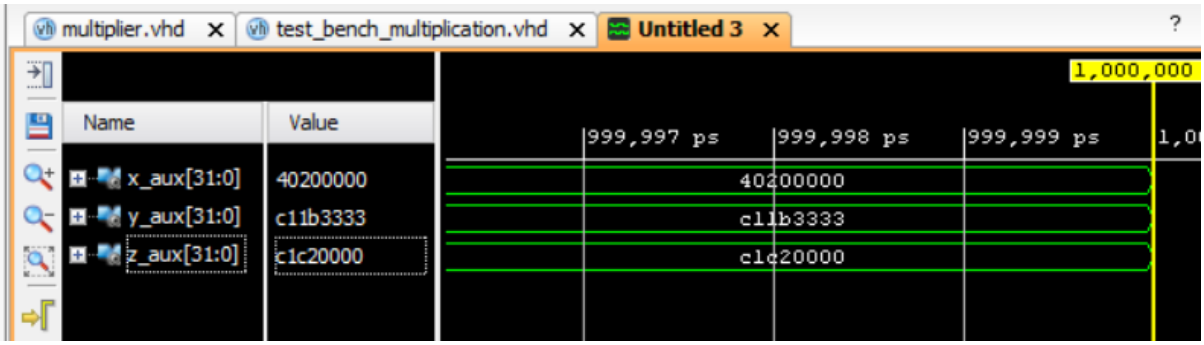
Pentru acest test am ales 2 numere mici:

x_aux <= "01000000001000000000000000000000"; --2.5

y_aux <= "11000001000110110011001100110011"; --(-9.7)

--z = (2.5) * (-9.7) = -24.25

-- z will be "11000001110000100000000000000000" ➔ C1C20000

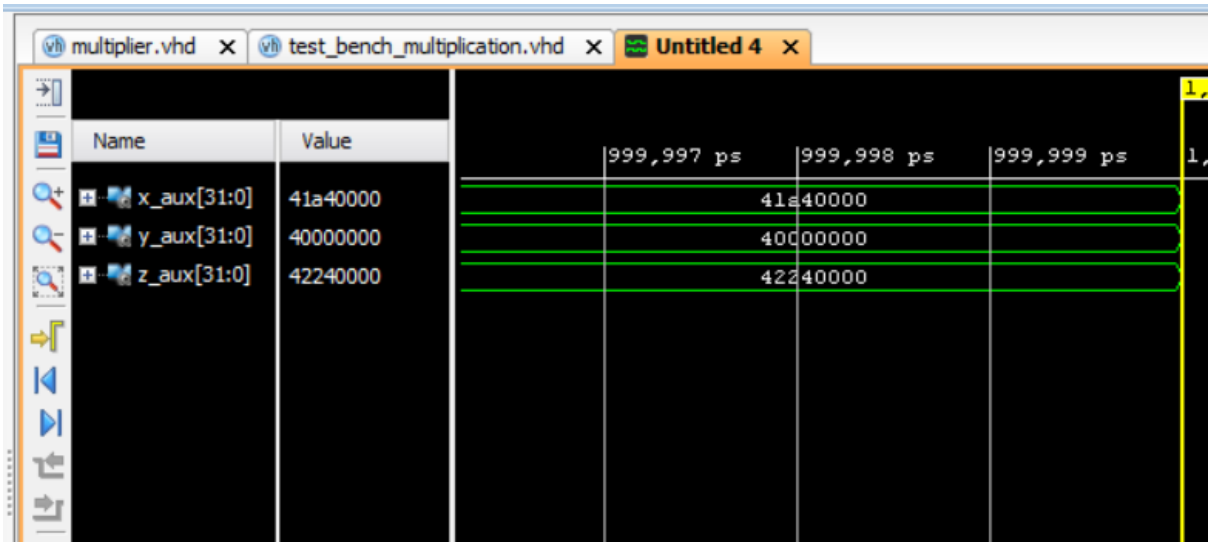


Alt exemplu rulat:

$$A = 20.5_{10} = 0.10000011.01001000000000000000000_2 = 41a40000_{16}$$

$$B = 2_{10} = 0.10000000.00000000000000000000000_2 = 40000000_{16}$$

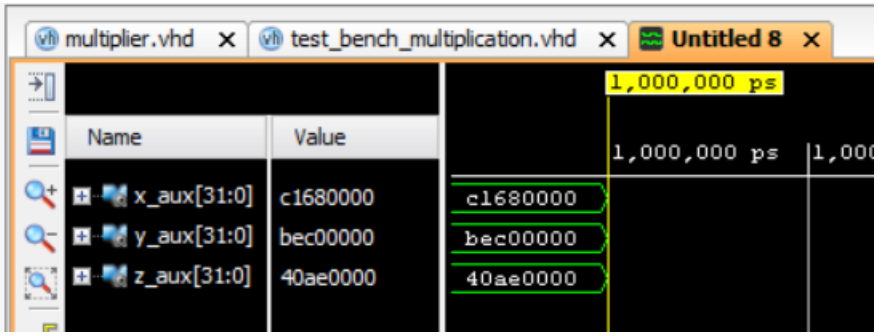
$$Q = A * B = 41.00_{10} = 0.10000100.01001000000000000000000_2 = 42240000_{16}$$



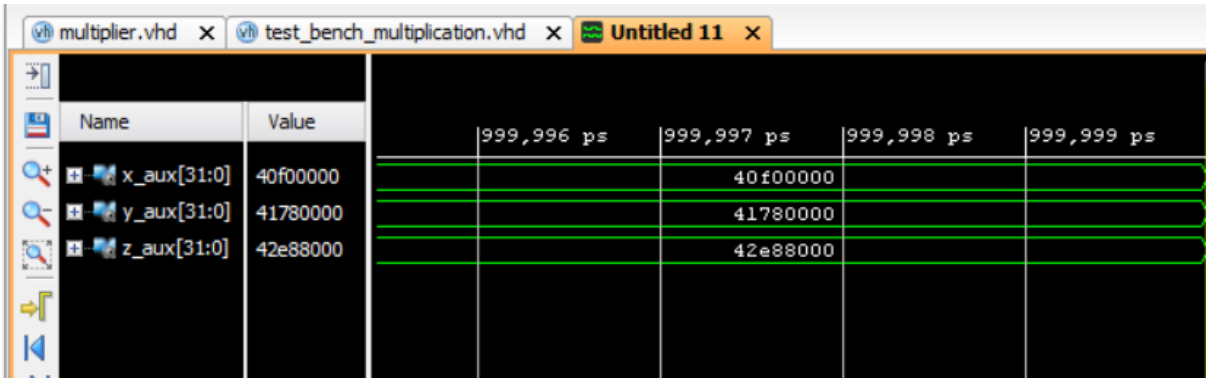
$$A = -14.5_{10} = 1\ 10000010\ 11010000000000000000000_2 = C1680000_{16}$$

$$B = -0.375_{10} = 1\ 01111101\ 10000000000000000000000_2 = BEC00000_{16}$$

$$Q = A * B = 5.4375_{10} = 0\ 10000001\ 01011100000000000000000_2 = 40AE0000_{16}$$



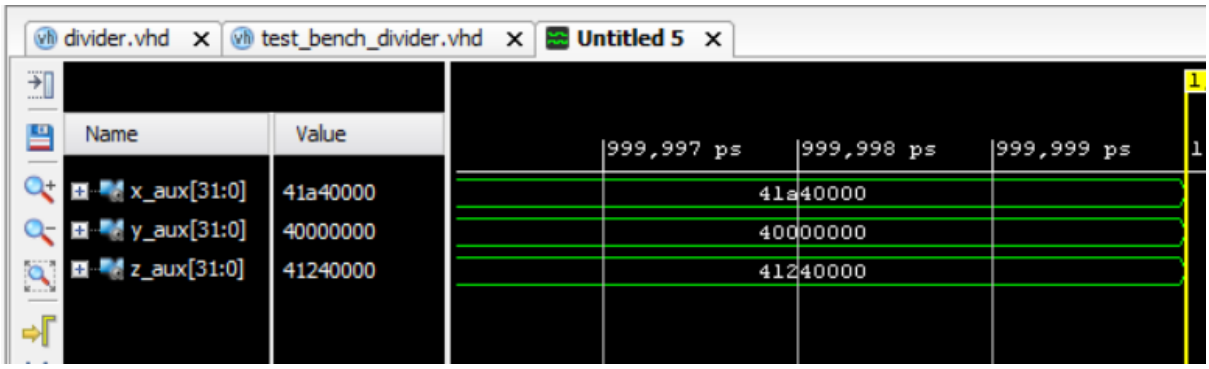
$$A = 7.5_{10} = 0\ 10000001\ 11100000000000000000000_2 = 40F00000_{16}$$
$$B = 15.5_{10} = 0\ 10000010\ 11110000000000000000000_2 = 41780000_{16}$$
$$Q = A * B = 116.25_{10} = 0\ 10000101\ 11010001000000000000000_2 = 42E88000_{16}$$



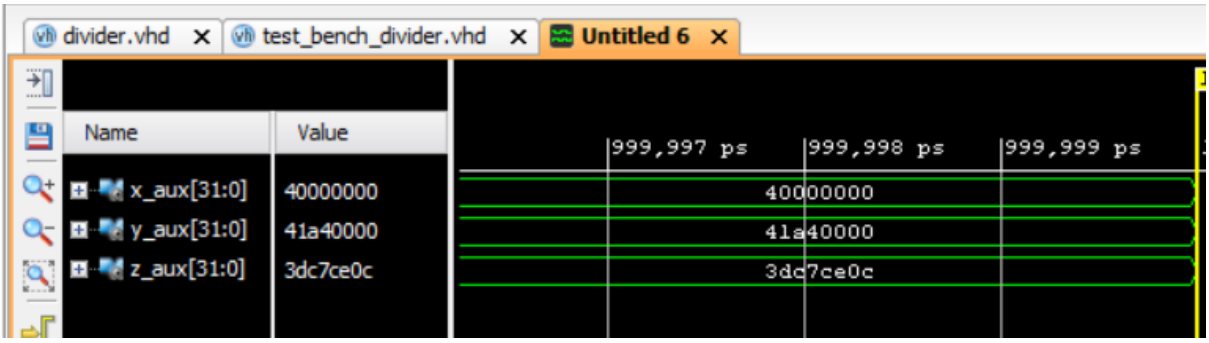
Și pentru împărțire:

$$A = 20.5_{10} = 0.10000011.01001000000000000000000_2 = 41a40000_{16}$$
$$B = 2_{10} = 0.10000000.00000000000000000000000_2 = 40000000_{16}$$

$$Q = A/B = 10.25_{10} = 0.10000010.01001000000000000000000_2 = 41240000_{16}$$



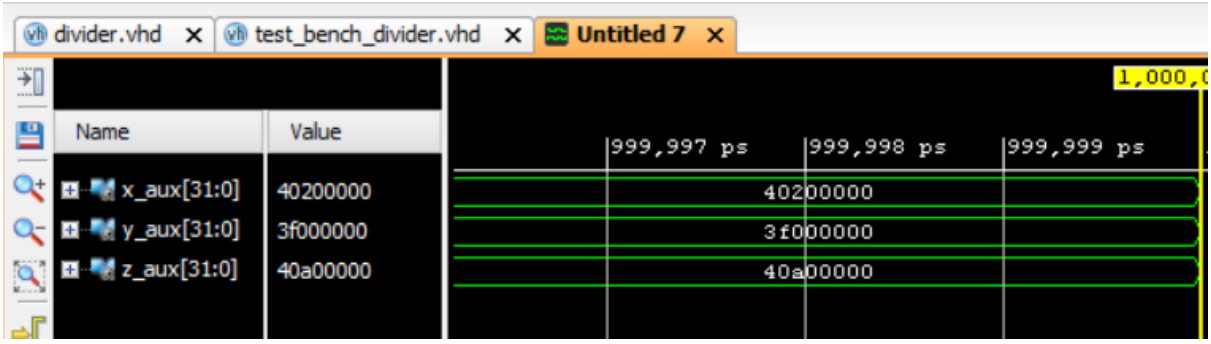
$$Q = B/A = 0.0975_{10} = 0.01111011.10001111100111000001100_2 = 3dc7ce0c_{16}$$



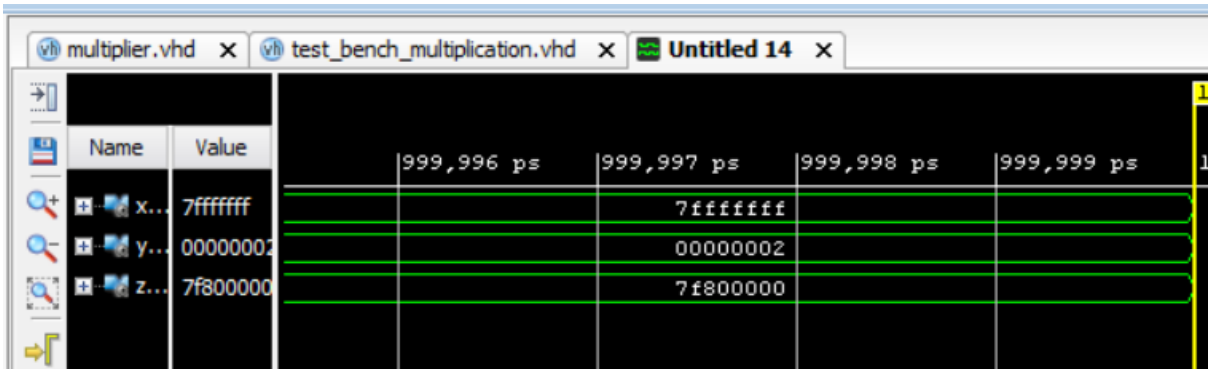
$$A = 2.5_{10} = 0.10000000.01000000000000000000000_2 = 40200000_{16}$$

$$B = 0.5_{10} = 0.01111110.00000000000000000000000_2 = 3f000000_{16}$$

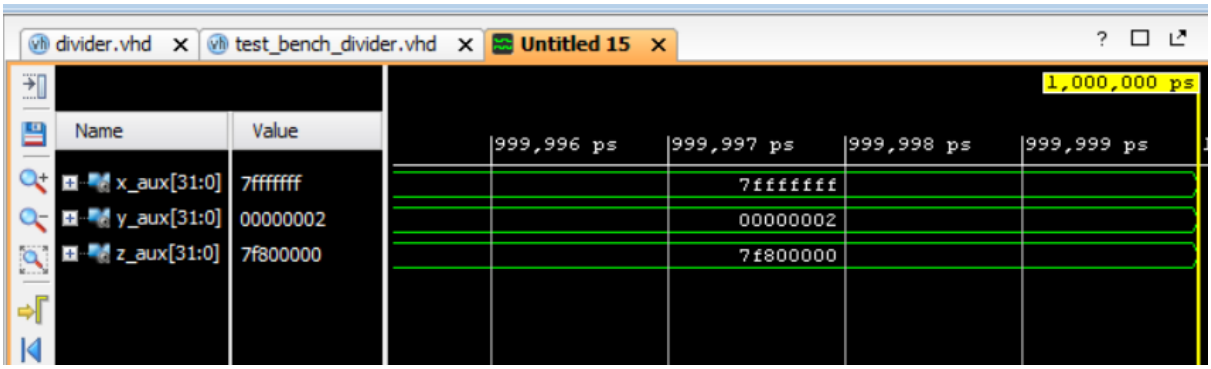
$$Q = A/B = 5_{10} = 0.10000001.01000000000000000000000_2 = 40a00000_{16}$$



Caz de overflow pentru inmultire:



Caz de underflow pentru impartire:



6. Concluzii

Scopul acestui proiect a fost implementarea si testarea inmultii si impartirii a 2 numere in formatul IEEE pe 32 de biti. Diferenta dintre implementarea obisnuita a inmultirii si impartire este aceea ca eu am ales implementarea acestora in virgula flotanta cu simpla precizie.

Mi-a placut la acest proiect faptul ca m-a pus la incercare pentru a gasi o solutie cat mai eficienta si usor de implementat. Cred ca cel mai greu task a fost implementarea impartirii mantiselor.

In final, mi s-a parut un proiect interesant si din care am avut de invatat mai bine cum functioneaza componentele in limbaj VHDL.

7. Bibliografie

https://binary-system.base-conversion.ro/convert-real-numbers-from-decimal-system-to-32bit-single-precision-IEEE754-binary-floating-point.php

https://en.wikipedia.org/wiki/Floating-point_arithmetic