FPU, single precision (adunare și înmulțire)

Darius-Iulian Stan
UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA |

1. Introducere

1.1. Propunere de proiect

Proiectul are ca și scop proiectarea, implementarea și testarea operațiilor de adunare și înmulțire pentru numerele reale, reprezentate în formatul IEEE 754 (virgulă flotantă), single precision (pe 32 de biți).

1.2. Perspectiva utilizatorului

Din cauza necesității resurselor hardware, pentru ca modul în care utilizatorul interacționează cu proiectul să fie cât mai facil, folosim, ca resursă auxiliară, o placă Basys 3. Utilizatorul poate alege prin intermediul switch-urilor, dintr-o memorie auxiliară, valorile pe care le dorește (dacă sunt în memorie), ele fiind trimise către ALU, iar rezultatul fiind afișat, în hexadecimal, pe cele 4 SSD-uri.

Totodată, dacă utilizatorul are experiență cu IDE-ul Vivado și VHDL, el poate crea o simulare în care să mapeze componenta generală a proiectului, având astfel posibilitatea de a alege orice număr ca input.

1.3. Planificare

Întâlnire 2 :

- Stabilire strategie (diagramă) adunare (luare în calcul valori speciale)
- Stabilire design, restricții ale proiectului general
- Înțelegere a algoritmului de înmulțire
- Redactare a Introducerii, Documentării pentru Documentația proiectului

Întâlnire 3:

- Stabilire strategie înmulțire
- Redactare la secțiunea Analiză a documentației pentru Adunare și Înmulțire (+ Componeta pentru valori speciale).
- Începere redactare pentru secțiunea de Design a coponentei generale + adunare

Întâlnire 4:

- Finalizare secțiunea de Design pentru adunare, înmulțire și componenta valorilor speciale
- Începere a implementarii

Întâlnirea 5 și 6

- Implementare + redactare la secțiunea Implementare

Întâlnire 7:

- Redactare la secțiunea Design și Testare pentru Adunare
- Revizuire documentație + completarea ei.

- Testare adițională a proiectului
- Implementarea resurselor necesare pentru ca proectul să fie utilizabil

2. Studiu Bibliografic

2.1. Transformarea unui număr real din baza 10 în binar

Presupunem că avem numărul real A = x + y , x fiind partea întreagă și y partea fracțională (ex: 85.625 = 85 + 0.625).

Pentru x, algoritmul este unul simplist:

- Scriem x ca sumă de puteri distincte ale lui 2
- De la stânga la dreapta, completăm cu 1 bitul aflat la poziția i dacă 2^i se află ân suma de mai sus, sau cu 0 în caz contrar (i pornește de la 0).

85 = 64 + 16 + 4 + 1
85 = 2^6 + 2^4 + 2^2 + 2^1

85 = 1010101

Pentru y, algoritmul nu o să genereze reprezentarea exactă a numărului pentru fiecare caz, însă, dat fiind că y este relativ mic, aproximarea lui nu duce la erori de calcul considerabile

0.625 = 0.5 + 0.125
0.625 = 2^(-1) + 2^(-3)

- Scriem y ca sumă de puteri ale lui 2 (datorită faptului că y<0, puterile lui 2 vor fi negative)
- De la dreapta la stânga, completăm cu 1 bitul aflat la poziția i dacă 2^(-i) se regăsește în suma, sau cu 0 în caz contrar (i pornește de la 1).

0.625 = 0.101

Astfel, A în binar este echivalent cu x + y în binar

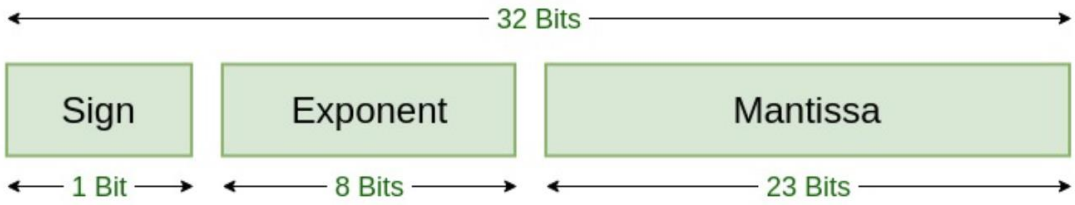
85.625 = 1010101.101

2.2. Formatul IEEE 754, single precision

După ce avem numărul în format binar, trebuie să îl normalizăm astfel încât să putem crea, mai apoi, reprezentarea numărului în formatul IEEE 754 (single precision). Normalizarea constă în aducerea numărului sub forma **1.m * 2 ^ exp**.

85.625 = 1010101.101
85.625 = 1.010101101 * 2^6

Având forma normalizată a numărului, putem să creăm corespondentul lui în formatul IEEE 754, cei 32 de biți fiind repartizați astfel:



a) Sign

Un singur bit (31), reprezentând semnul numărului. Are valoarea 1 dacă numărul este negativ, în cazul în care numărul este pozitiv, are valoarea 0.

b) Exponent

Biții [30]…[23], având ca semificație exponentul numărului. Însă, dat fiind că avem nevoie să reprezentăm și exponenți negativi, Exponent va avea un bias (în cazul nostru 127), astfel că exponent = bias + exp. Bias reprezintă jumătate din numărul de valori ce pot fi reprezentate pe bitii alocați pt exponent (127 = 256 / 2), astfel primele 126 de valori sunt exponenti negativi, iar urmatoarele 128 sunt pozitivi (respectiv 0). Pentru a calcula exponent pe baza exp, în cazul nostru, trebuie doar să adăugăm 127.

c) Mantissa

Biții [22]…[0], reprezintă șirul de biți după virgulă (spre dreapta) din forma normalizată a numărului. Se observă că bitul din stânga virgulului nu este stocat, întrucât prezența lui în forma normalizată este garantată în orice caz (cu excepția lui 0).

85.625 = 1.010101101 * 2^6
Sign = 0
Exp = 127 + 6 = 133 = 10000101
Mantissa = 010101101
⇨ 85.625 = 0 10000101 00000000000000010101101

2.3. Valori speciale în IEEE 754

2.3.1. **Zero** – o valoare specială, cu două reprezentări chiar, mantissa și exponentul au toți biții 0, iar bitul de semn poate fi 1 sau 0, deci “există” +0 și -0, ele fiind valori distincte, chiar dacă sunt egale.

2.3.2. **Infinity** – valorile infinit și minus infinit sunt reprezentate cu biții exponentului 1 și ai mantissei 0, sign facând diferența dintre cele două valori.

2.3.3. **Denormalised** – pentru numerele extrem de mici (2 la puterea -128 și mai mici), când biții exponentuli nu mai pot reprezenta exponentul în sine (valoarea minimă este 0 (127 – 127), avem cazul numerelor denormalizate, ele fiind reprezentate prin expoent = 0 și mantissa diferită de 0 (dacă e 0, avem cazul 0 la puterea 0).

2.3.4. **NAN** – Not a number – folosită pentru a reprezenta un număr care este o eroare, biții exponentului sunt 1 iar semnul e 0, sau matissa diferită de 0.

EXPONENT	MANTISA	VALUE
0	0	exact 0
255	0	Infinity
0	not 0	denormalised
255	not 0	Not a number (NAN)

2.4. Adunarea și înmulțirea numerelor în virgulă mobilă (forma normalizată)

Pentru a putea proiecta adunarea și înmulțirea pentru formatul IEEE 754, trebuie să ne raportăm la operațiile respective în cazul formei normalizate a numerelor, dat fiind că IEEE 754 este conceput pe baza ei.

2.4.1. Adunarea

Pentru adunarea a două numere, A și B, ne raportăm la adunarea lor în forma normalizată. Astfel, separând atributele numerelor (sign, exponent, mantissa), avem următorul algoritm:

- Când sign(A) = sign (B)
- După ce am aflat exponenții, vom scrie (mantissa, exponent) la exponent comun (cel mai mic), astfel încât să putem aduna mantissele
- Adunăm mantissele, părțile întregi spearat de părțile flotante, la final urmând să adunăm eventuala parte reală obținută prin adunarea flotantelor, la adunarea întregilor.
- Rotunjim mantissa și exponentul astfel încât să respecte formatul ales.

- Când sign(A) != sign (B)
- Algoritmul este asemănător, însă scădem din modului celui mai mare modului celui mai mic, urmând preluăm semnul numărului cu modului mai mare.

Ca observație, În adunare vom ține cont de potențialele “Valori speciale” (cum ar fi infinit) sau de posibila apariíe a overflow-ului sau underflow-ului. Mai multe detalii vor fi scrise la **Implementare și Design.**

2.4.2. Înmulțirea

La înmulțirea a două numere A, B în forma normalizată, după ce separăm atributele, executăm următorii pași:

- Semnul va fi decis astfel: dacă numerele au același semn, atunci semnul rezultatului va fi +, dacă au semne diferite, semnul va fi –.

- Exponentul rezultatului va fi suma exponenților numerelor
- Matissa rezultatului are un proces mai complex:
    - Shift-ăm la dreapta matissa cu numărul mai mic de biți, astfel încât ambele să aibă același număr de biți.
    - Înmulțim matissele, conform desparțirii parantezelor următoare:
    - Datorită faptului că al treilea termen are partea întreagă 0, ea nu are urmări în rezultat.

(1 + 0.matissa1) (1 + 0.matissa2) =

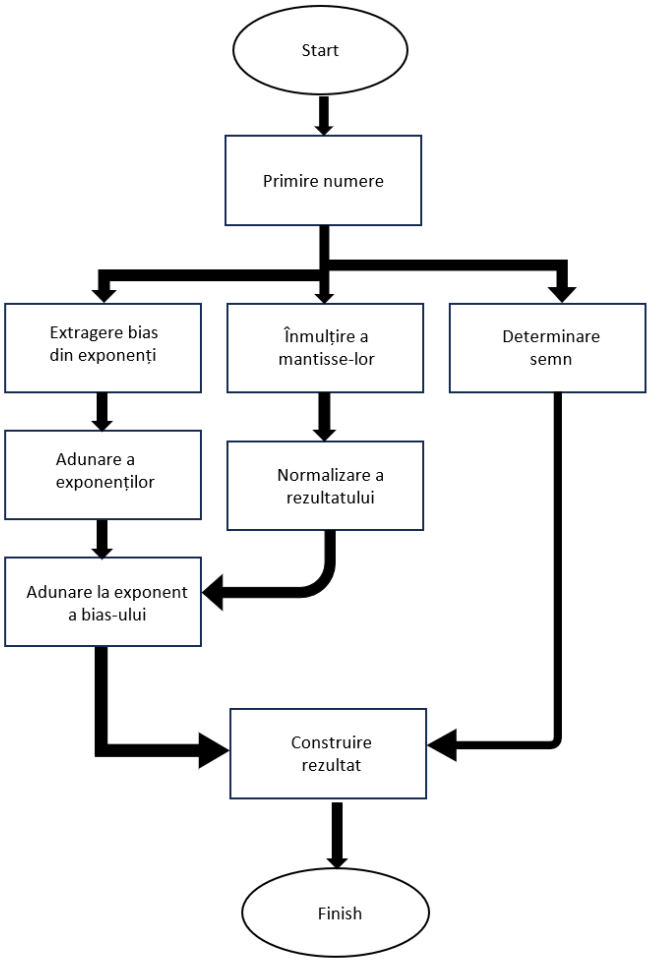
1 + 0.matissa1 + 0.matissa2 + 0.matissa1*0.matissa2

- După ce calculăm al patrulea termen, adunăm la el celelalte numere fracționale, iar dacă rezultatul are mai mulți biți decât matissele inițiale, cei mai semificativi reprezintă un surplus, partea întreagă, care urmează să se adune cu primul terme din chenar (1).
- Astfel, dacă nu ne aflăm în cazul de la punctul anterior, am obținut rezultatul, altfel, va trebui să aducem în forma normalizată rezultatul obținut, asta însemnand că obținem un alt exponent, care va fi adunat cu rezultatul obținut din adunarea celor doi exponenți inițiali.

Evident, în IEEE 754, procesul este unul mult mai complex, unde va treui să ținem cont și de numerele speciale și de posibilitatea de underflow și overflow a rezultatului, mai multe detalii vor fi expuse în secțiunea **Implementare și Design**.

3. Analiză

3.1. Înmulțirea



După cum observăm în imaginea din stânga, componentele principale vor fi un sumator pe 8 biți pentru exponent și un multiplicator pe 24 biți pentru mantisse.

Bineînțeles, procesul de normalizare a rezultatului are consecințe asupra exponentului final, în cazul în care avem transport în partea întreaga, așadar sumatorul pentru exponenți trebuie să primească trei inputuri.

Așadar, sumatorul ce primește trei inputuri va fi de tipul CarrySaveAdder, datorită vitezei cu care lucrează și facilității de a aduna trei numere simultan, ales în detrimentul CarryLookAhead-ului.

Pentru operația de înmulțire, vom implementa WallaceTreeMultiply-er, folosind ca și componente auxiliare CarrySaveAdder-e (care primesc ca și inputuri numere pe 24 de

biți) și, dat fiind că avem 24 astfel de adunări, propun ca CarrySaveAdder-ul să aibă patru inputuri.

Determinarea semnului o vom face folosind o simplă poarte Xor.

Referitor la valorile speciale din IEEE 754, există două posibilități în care să apară în procesul de înmulțire

- Când un operand (primit ca și input) este o valoare specială: Componenta specială care identifică valorile speciale va semnala că avem de-a face cu o valoare specială, astfel că va trimite rezultatul corespunzător fără să mai trebuiască să se execute prcesul de înmulțire. Valorile trimise de componentă se regăsesc în tabelul din secțiunea 2.
- Când rezultatul este (sau trebuie să fie) o valoare specială. Overflow-ul și Underflow-ul vor fi semnalate de unitatea de înmulțire, iar, rezultatul, împreună cu semnalele de over, respectiv underflow, vor fi trimise către componenta de tratare a valorilor speciale, urmând ca ea să decidă cum trebuie, de fapt, să arate rezultatul.

3.2. Adunarea

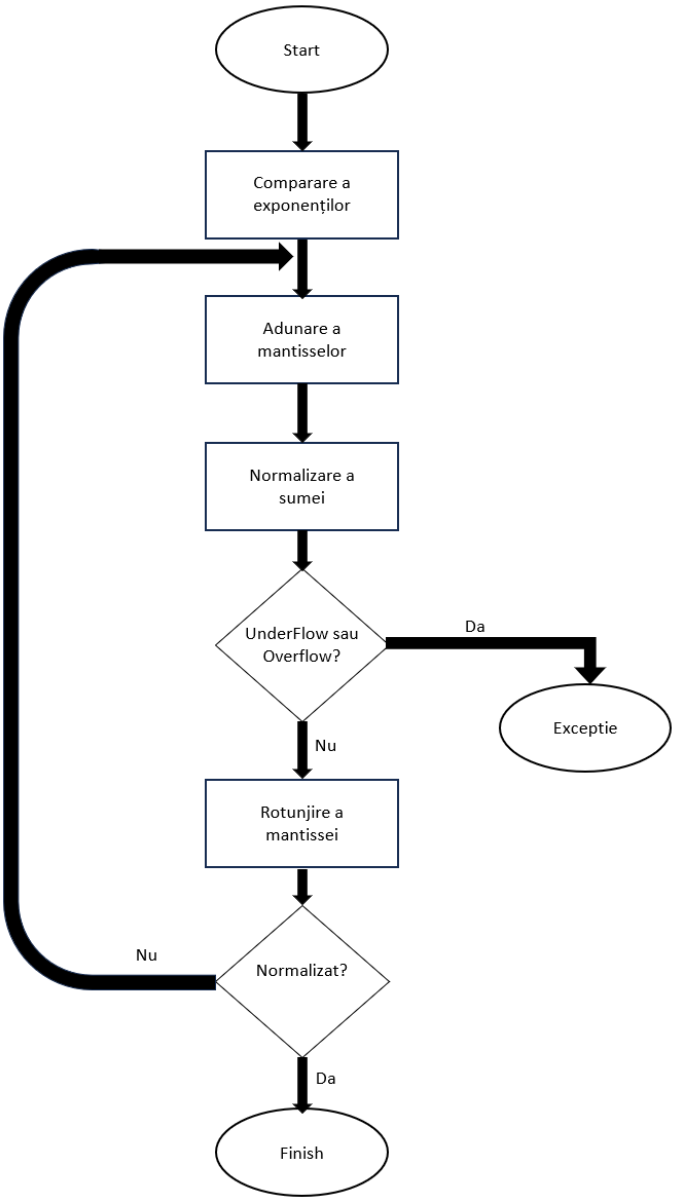
După cum observăm în imaginea din dreapta, adunarea în IEEE 754 este extrem de asemănătoare cu adunarea în forma normalizată.

Avem un sumator pe 24 de biți pentru adunarea mantisselor, alegem să fie CarryLookAhead, datorită vitezei de lucru.

Compararea exponenților (și automat modificarea uneia dintre mantisse) o vom face printr-un scăzător. Dacă diferența este 0, atunci nu modificăm mantissele, în caz contrar vom modifica mantissele astfel încât să avem exponet comun.

Normalizarea sumei matiselor, evident, aduce problemă underflow-ului sau overflow-ului, în cazul în care se va trimite un semnal către componenta responsabilă cu detectarea și tratarea valorilor speciale.

Rotunjirea mantissei constă, dacă e cazul, în aprximarea ei la cel mai apropiat număr care poate fi reprezentat pe 24 de biți.



3.3. Tratarea valorilor speciale

În tabelle alăturat sunt prezentate valorile rezultate pentru fiecare combinație de valori speciale, atât la adunare, cât și la înmulțire

+	0	+inf	-inf	NaN	Normalised	Denormalised
0	0	+inf	-inf	NaN	Normalised	Denormalised
+inf	+inf	+inf	NaN	NaN	+inf	+ inf
-inf	-inf	NaN	-inf	NaN	-inf	-inf
NaN	NaN	NaN	NaN	NaN	NaN	NaN
Normalised	Normalised	+inf	-inf	NaN	Normalised	Normalised(?)
Denormalised	Denormalised	+inf	-inf	NaN	Denormalised	Denormalised
*	0	+inf	-inf	NaN	Normalised	Denormalised
0	0	NaN	NaN	NaN	0	0
+inf	NaN	+inf	-inf	NaN	+inf	+inf
-inf	NaN	-inf	+inf	NaN	-inf	-inf
NaN	NaN	NaN	NaN	NaN	NaN	NaN
Normalised	0	+inf	-inf	NaN	Normalised(or not)	Depending
Denormalised	0	+inf	-inf	NaN	Depending	Denormalised

Componenta ce se ocupă cu tratarea și identificarea valorilor speciale va trata fiecare caz prezentat în cele două tabele, având ca output formatul valorii speciale sau, dacă totuși valoarea rezultată poate fi un număr reprezentabil în IEEE 754, va trimite un semnal care să activeze coponenta de adunare (sau înmulțire).

4. Design

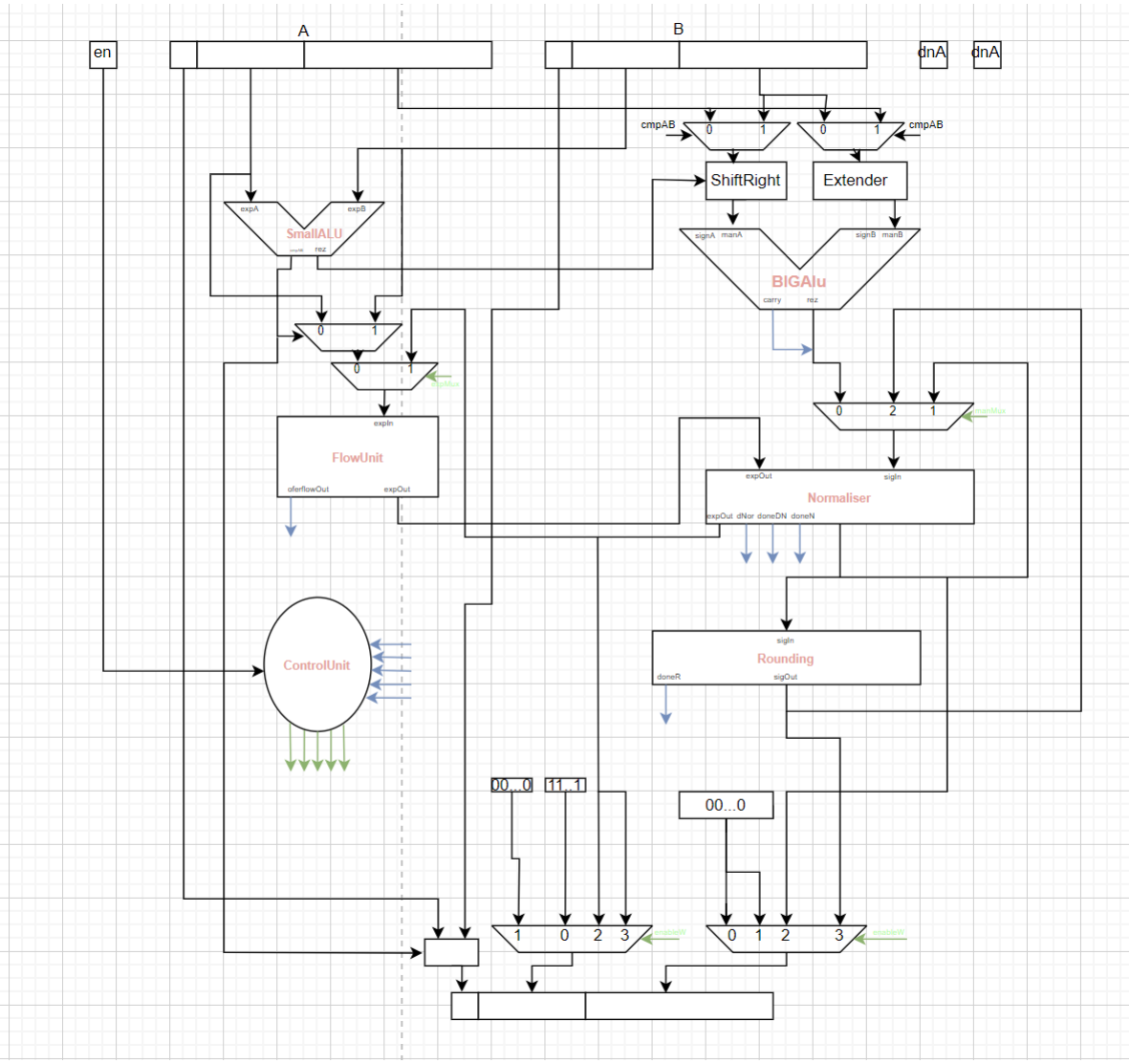
4.1. Componenta generală (principală)

Componenta generală va avea ca input A, B – numerele pe 32 de biți (format IEEE 754) - și op – un bit, 1 semificând operația de înmulțire și 0 de adunare. Ca și output are rezultatul operației pe A și B, pe 32 biți.

Ea va conține componenta responsabilă cu adunarea, cea responsabilă cu înmulțirea, precum și cea care trateaza valorile speciale. În plus, ea va conține toate componentele ce fac legătura între cele trei componente mari.

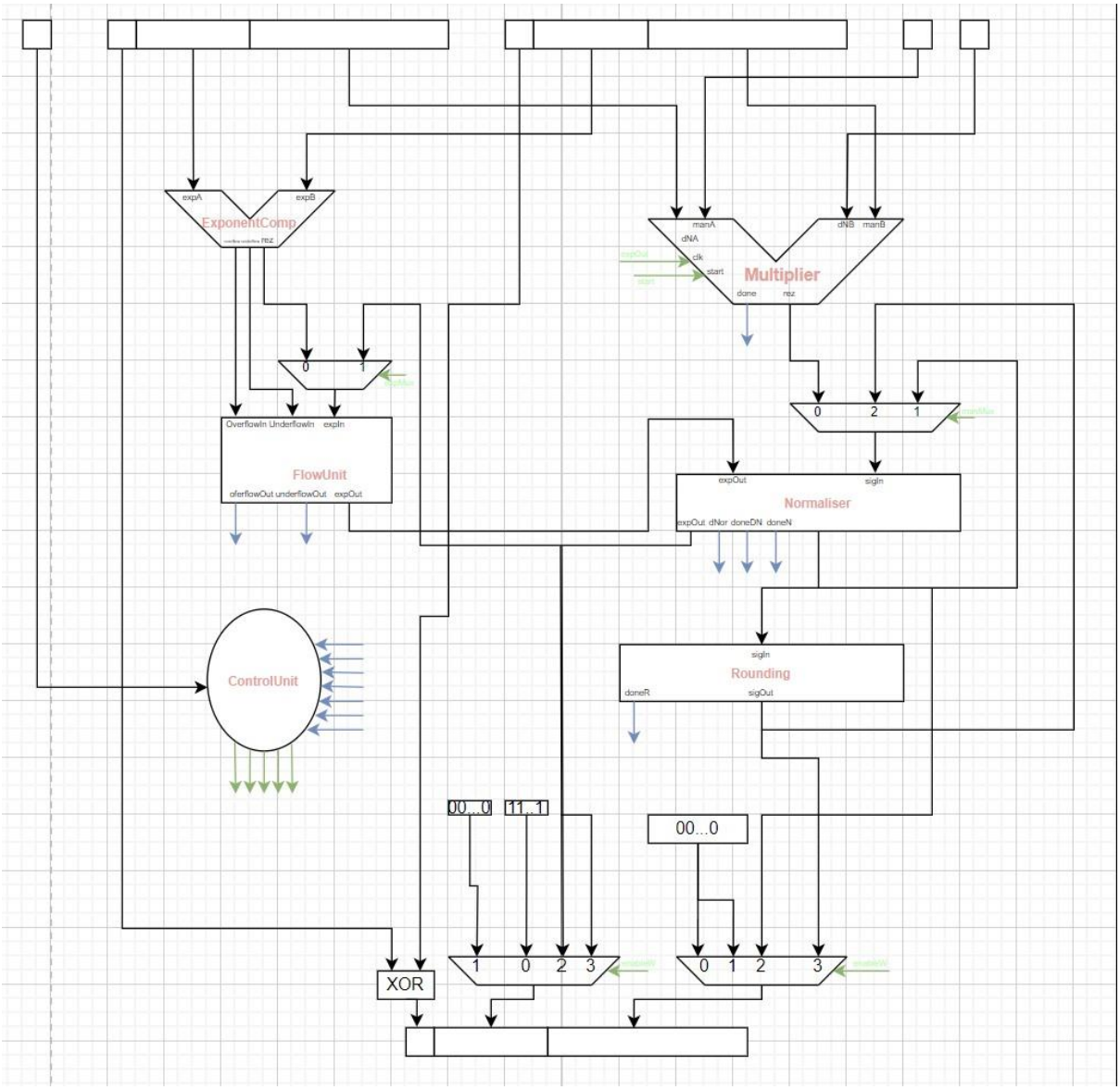
4.2. Compoenta pentru Adunare

Are două inputuri pe 32 de biți, A și B, reprezentând numerele ce vrem să le adunăm. Ca și output va avea rezultatul adunării (pe 32 de biți), dar și două intrari de cate un bit, reprezentând dacă numerele sunt în forma normalizată sau nu. Totodată, are un input ca enable, execuția componentei începând doar când el este setat pe 1.



4.3. Componenta pentru Înmulțire

Avem două inputuri pe 32 de biți, reprezentând cele două numere de înmulțit. Ca și output avem rezultatul pe 32 de biți. Totodată, are un input ca enable, execuția componentei începând doar când el este setat pe 1, si doua inputuri, reprezentand daca A si/sau B sunt numere normalizate sau denormalizate.



4.4. Componenta pentru valorile speciale

Are ca input cele două numere și operația ce se execută, dacă avem de-a face cu valori speciale, cu rezultat ce nu necesită calcul (predefinit), o să trimită un unenable către componentele de adunare și înmulțire, astfel încât să nu mai efectueze operațiile pe numerele date și, pe portul de ieșire de 32 de biți, valoarea specială rezultată. În caz contrar, trimite un enable (dacă avem Denormalised cu Normalised). Totodată, are și un port de intrare pe doi biți, care cere, prin codificarea internă, o valoare specială (ea va fi relevată prin portul de 32 de biți menționat mai sus)

5. Implementare

5.1. Componenta pentru adunare

După cum observăm în design-ul pentru Adunare, avem mai multe componente, după cum urmează:

- SmallALU – responsabil cu diferența exponenților. Exponenții sunt primiți pe porturi de 8 biți, dat fiind că au valori între 1 și 254, amândoi sunt pozitivi, așadar, pentru a putea efectua scăderea (A – B), concatenăm un '0' înaintea fiecăruia, iar numerele sunt deja în complement față de doi. Pentru că scădem pe B din A, iar B este pozitiv, va trebui să-i aflăm opusul (-B), așadar folosim componenta Opp2C, care ne dă opusul numărului în complement față de doi. Efectuăm scăderea cu ajutorul unui CarryLookAheadAdder pe 9 biți. Rezultatul primit este în complement față de 2, pe 9 biți, dacă cel mai seminificativ bit este 1, atunci rezultatul este negativ, deci efectuăm Opp2C pentru el, expDiff primind valoarea negată, iar cmpAB, semnaleaază că B este mai mare decât A, în caz contrar, expDiff primește chiar rezultatul obținut, iar cmpAB semnalează ca A este mai mare decât B.

```
begin
    expB9 <= '0' & expB;
    expA9 <= '0' & expA;
    negare: Opp2C port map(expB9,mexpB9);

    scadere: carryLookAheadAdder9Bits port map('0',expA9,mexpB9,diff9,overflow);
    aducereinBinar: Opp2C port map(diff9, mdiff9);
    bs <= diff9(8);
    cmpAB <= not diff9(8);
    expDiff(0) <= ((not bs) and diff9(0)) or (bs and mdiff9(0));
    expDiff(1) <= ((not bs) and diff9(1)) or (bs and mdiff9(1));
    expDiff(2) <= ((not bs) and diff9(2)) or (bs and mdiff9(2));
    expDiff(3) <= ((not bs) and diff9(3)) or (bs and mdiff9(3));
    expDiff(4) <= ((not bs) and diff9(4)) or (bs and mdiff9(4));
    expDiff(5) <= ((not bs) and diff9(5)) or (bs and mdiff9(5));
    expDiff(6) <= ((not bs) and diff9(6)) or (bs and mdiff9(6));
    expDiff(7) <= ((not bs) and diff9(7)) or (bs and mdiff9(7));
    --- cmpAB = 1 = True, A > B
    --- expDiff - valoarea A-B, in modul
```

- ShifterRight – primește un șir pe 8 biți (reprezentând, de fapt, numărul de poziții cu care trebuie shiftat un număr) și un al număr pe 23 de biți, reprezentând numărul ce trebuie shiftat. Totodată, primește și un bit care semnaleaza dacă numărul cuc are lucrăm e denormalizat sau normalizat. Componenta este una asincronă, întrucât pentru fiecare bit din numărul reprezentând cantitatea de shiftat avem un Shifter auxiliar, care shiftează inputul cu numărul de poziții corespunzator seminifcativității

bitului respectiv (de ex, pentru bitul 3, cu valoarea 1, shiftam 8 poziții).

```
s1: Shift1Bit port map(nrBits(0), outA0, outA1);
s2: Shift2Bits port map(nrBits(1), outA1, outA2);
s3: Shift4Bits port map(nrBits(2), outA2, outA3);
s4: Shift8Bits port map(nrBits(3), outA3, outA4);
s5: Shift16Bits port map(nrBits(4), outA4, outA5);
s6: Shift32Bits port map(nrBits(5), outA5, outA6);
s7: Shift64Bits port map(nrBits(6), outA6, outA7);
s8: Shift128Bits port map(nrBits(7), outA7, outA8);
outA <= outA8;
```

- Extender – asemănător cu ShifterRight, însă nu shiftează, doar extinde celălalt număr de 23 de biți cu ‘0’ în partea stângă, pentru ca numerele primite ca output din ShifterRight și Extender să aibă aceeași dimensiune.

- BigALU – responsabil cu adunarea mantinselor. După ce am shiftat numărul mai mic și am extins numărul mai mare, pentru a efectua adunarea mantinselor, folosim Opp2CBig pentru a afla corespondentul în complement față de doi a fiecărui număr (avem ca și intrari signA și signB, pentru a putea alege corespondentul negativ sau pozitiv, după caz). Efectuam adunarea folosind 3 CarryLookAheadAdder pe 93 de biți, iar cel mai seminifativ bit va fi xor între cei mai semnificativi biți din A și B, cât și carryOut-ul de la ultima adunare. Acest bit semnalează dacă avem overflow sau underflow, în funcție de el creăm rezultatul dorit (în modul) cât și carry-ul, dacă este cazul).

```
manAext <= "0" & manA;
manBext <= "0" & manB;
n1: Opp2CBig port map(manAext, minusmanAext);
n2: Opp2CBig port map(manBext, minusmanBext);
-------------------------------------------------------------------------------------------

gen_for_loop: for i in 0 to 279 generate
        Bext(i) <= ((not signB) and manBext(i))or (signB and minusmanBext(i));
        Aext(i) <= ((not signA) and manAext(i))or (signA and minusmanAext(i));
end generate;
-------------------------------------------------------------------------------------------
c1: CarryLookAheadAdder93bits port map('0',Aext(92 downto 0),Bext(92 downto 0),partial1,cout1);
c2: CarryLookAheadAdder93bits port map(cout1,Aext(185 downto 93),Bext(185 downto 93),partial2,cout2);
c3: CarryLookAheadAdder93bits port map(cout2,Aext(278 downto 186),Bext(278 downto 186),partial3,cout3);
leftbit <= Aext(279) xor Bext(279) xor cout3;
-- asta e ultimul bit care basically e suma dintre ultimii biti ai lui a si b si carry ul trecut.
--dat fiind ca lucram in comp fara de 2 pe 279 de nr, nu avem cazul de overflow.
--daca rez e negativ, aflat pozitivul:)
GivenResult <= leftbit & partial3 & partial2 & partial1; -- aici am suma in comp. fata de 2
n3: Opp2CBig port map(GivenResult, OppGivenResult); -- opusul lui given result
CarryOut <= ((not signA) and (not signB) and leftbit) or
            (signA and signB and (not leftbit));

gen_for_loop2: for i in 0 to 278 generate
    manRez(i) <= ((not signA) and (not signB) and GivenResult(i)) or
                 ((signA) and (signB) and OppGivenResult(i)) or
                 ((signA xor signB) and (not leftbit) and GivenResult(i)) or
                 ((signA xor signB) and leftbit and OppGivenResult(i));
end generate;
```

- Normaliser – primește un număr pe 280 de biți, reprezentând mantinssa rezultatului, dacă aceasta începe cu “00” – numărul e denormalizat, activăm dNor, dacă exponentul e 0, activăm și doneDN, semnalizând ca avem rezultatul pregătit. Dacă cei mai seminificativi biți sunt “01”, numărul e deja normalizat, așadar trebuie doar să-l

trimitem spre rotunjire. Dacă aceștia sunt “10” sau “11”, incrementam exponentul și shiftam la dreapta mantissa, apoi trimitem numărul obținut spre rotunjire.

- Rounding – dacă cel mai semificativ bit din inputul de 280 de biți care nu intră în mantissa finala (adică 254).
- FlowUnit – primim exponentul cu care lucrăm, în primă instanță ori exponentul lui A, ori exponentul lui B, în funcție de cmpAB din SmallALU, urmând să primim exponentul rezultat din Normaliser.
- ControlUnit

```
process(current_state, next_state, doneR, dNor, doneN, overflow)
begin
    case current_state is
    when State1 => expMux <= '0';
                   manMux <= "00";
                   next_state <= State2;
                   stateX <= "0001";
                   done <= '0';

    when State2 => if overflow = '1' then next_state <= FinalState1;
                   elsif doneDN = '1' then next_state <= FinalState3;
                   elsif dNor = '1' then next_state <= State3;
                   elsif dNor = '0' and doneR = '1' then next_state <= FinalState4;
                   elsif dNor = '0' and doneR = '0' then next_state <= State4;
                   end if;
                   stateX <= "0010";
                   done <= '0';

    when State3 => expMux <= '1';
                   manMux <= "01";
                   next_state <= State2;
                   stateX <= "0011";
                   done <= '0';

    when State4 => expMux <= '1';
                   manMux <= "10";
                   stateX <= "0100";
                   done <= '0';
                   next_state <= State2;

    when FinalState1 => enableW <= "00"; done <= '1'; stateX <= "1001";
    when FinalState3 => enableW <= "10"; done <= '1'; stateX <= "1011";
    when FinalState4 => enableW <= "11"; done <= '1'; stateX <= "1100";
    when others => next_state <= nonState; done <= '0';
    end case;
end process;
```

5.2. Componenta pentru înmulțire

Este construită după același raționament precum componenta pentru adunare, după cum urmează:

- ExponentComp: echivalentul lui SmallALU, însă, calculăm exponentul final pe care să îl aibă rezultatul: expA + expB – bias (bias fiind 127).

```
extA <= "0" & expA;
extB <= "0" & expB;
```

```
Aminnus127: CarryLookAheadAdder9bits port map('0',extA, bias,extRez1,carryOut1);
plusB: CarryLookAheadAdder9bits port map('0',extB, extRez1,extRez2, carryOut2);
```

```
overflow <= ((extRez2(8)) and (not extRez1(8))) or ((not extRez2(8)) and
            extRez2(7) and extRez2(6) and extRez2(5) and extRez2(4) and extRez2(3) and extRez2(2) and extRez2(1) and extRez2(0));
underflow <= (extRez2(8) and extRez1(8));
```

```
rez <= extRez2(7 downto 0);
```

Evident, există și cazul în care exponentul rezultat este mai mare decât 254 sau mai mic decât 0, așa ca tratăm cazurile de Underflow sau Overflow.

- Multiplier – echivalentul BigALU, însă este responsabil cu înmulțirea mantisselor. El este, de fapt, un simplu Shift and Add multiplicator.

```
extA <= ( dNA) & manA;
extB <= ( dNB) & manB;
```

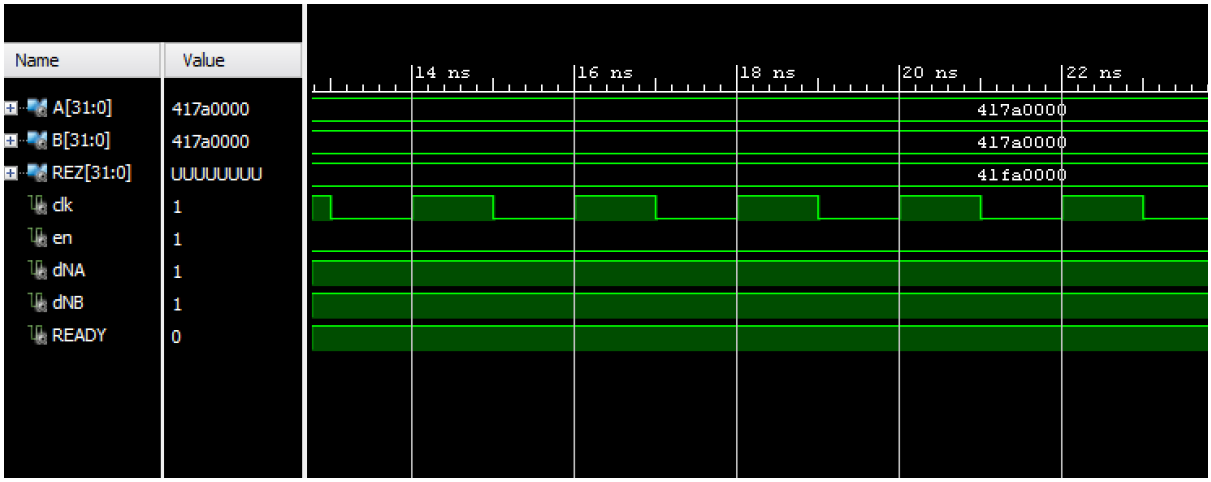
```
extA <= ( dNA) & manA;
extB <= ( dNB) & manB;
```

```
unitcontrol: ControlUnitM port map(en => start, clk => clk, Q0 => A_Q(0), Add => add, Write => write, Shift => shift, Done => d);
sumator: Sumator24Bits port map(en => add, A=>A_Q(47 downto 24),B=>M, carryIn => '0', rez => prez, carryOut => cout);
ccnntt: Counter port map(clk => clk, en => shift, rst => (start), cnt => cnt, done => d);
process(start,shift,add,write,d, clk)
    begin
        if start = '1' and rising_edge(clk) then
            A_Q <= "000000000000000000000000"& extB;
            M <= extA;
        else
            if add = '1' then
                pprez <= prez;
                pcout <= cout;
            elsif write = '1' then
                A_Q(47 downto 24) <= pprez;
            elsif shift = '1' then
                    ppprez <= pcout & A_Q(47 downto 1);
            else pcout <= '0';
            end if;

            if rising_edge(clk) then
                if shift = '1' then
                    A_Q <= ppprez;
                end if;
            end if;

        end if;

            rez <= A_Q;

    end process;
```

- Normaliser și Rounding sunt asemănătoare cu cele de la adunare, cu excepția că inputurile și outpurile reprezentând mantissele sunt pe 48 de biți, nu 280.
- FlowUnit este și el asemănător cu cel de la adunare, însă aici emite semnale pentu overflow sau underflow, primite de la ExponentComp sau găsite de el (dacă exponentul primit din Normaliser este 255 – 11111111).
- ControlUnit – asemanatoare cu cea de la adunare, însa acoperă și cazurile în care avem underflow.
- Semnul rezultatului este un simplu xor intre semnele numerelor initiale.

6. Testare

6.1. Componenta pentru adunare

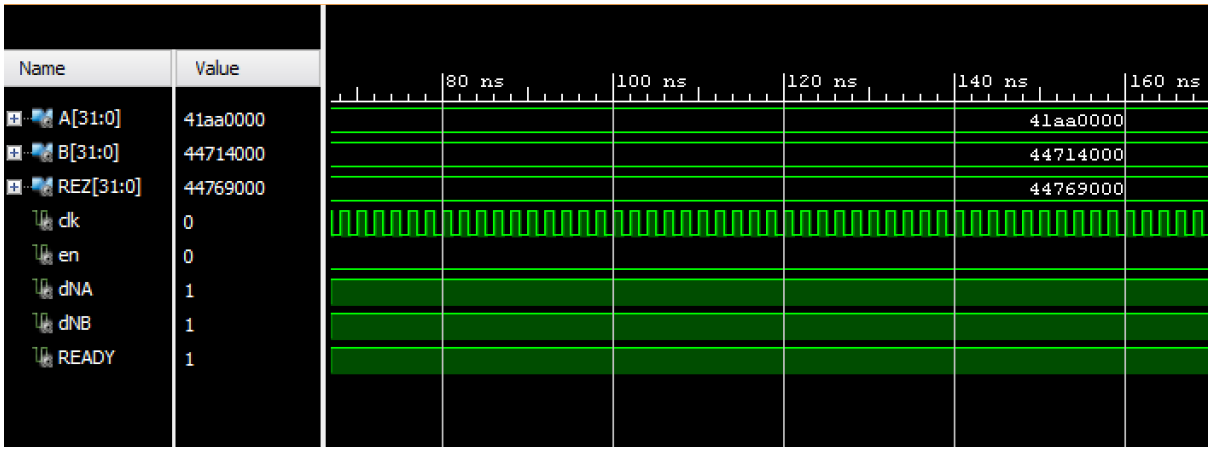


A = 01000001011110100000000000000000 = 15.625

B = 01000001011110100000000000000000 = 15.625

A + B = 31.25

rezultat primit => 41fa0000 = 01000001111110100000000000000000 = 31.25



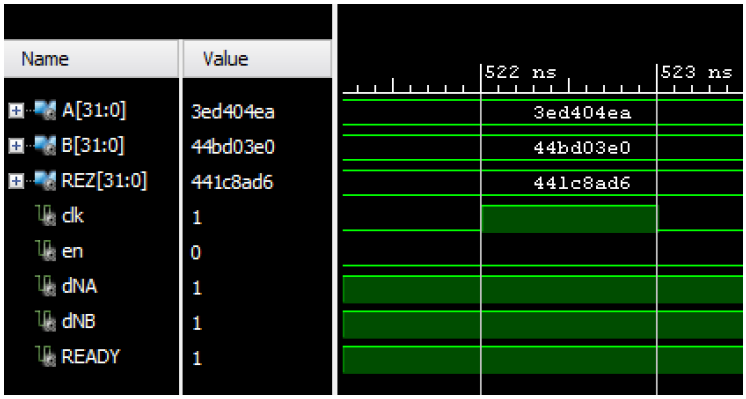
A = 01000001101010100000000000000000 = 21.25

B = 01000100011100010100000000000000 = 965.0

A + B = 986.25

rezultat primit => 44769000 = 01000100011101101001000000000000 = 986.25

6.2. Componenta pentru inmultire

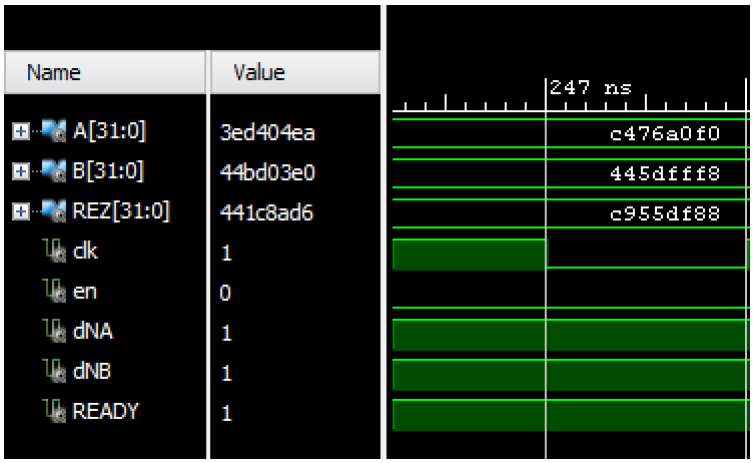


A = 0,41409999132156372070312 5 = 00111110110101000000010011101010

B = 1512.12109375 = 01000100101111010000001111100000

A * B = 626.1693298006641

rezultat primit => 441c8ad6 = 01000100000111001000101011010110 = 626.169311523



A = -986.514648438 = 11000100011101101010000011110000

B = 887.999511719 = 01000100010111011111111111111000

A * B = -876024.5261165849

rezultat primit => c955df88 = 11001001010101011101111110001000 = -876024.5

7. Bibliografie

7.1. „Single-Precision, Double-Precision, & Multi-Precision Computing: Main Differences” – AMD Xilinx, disponibil online: Single-Precision, Double-Precision, & Multi-Precision Computing: Main Differences (xilinx.com)

7.2. „IEEE Standard 754 for Binary Floating-Point Arithmetic” – Lectures Notes, Prof. W. Kahan Elect.Eng & Computer Science University of California, 1997

7.3. „Computer Organisation and Design”, ediția a cincea, David A. Patterson and John L. Hennessy

7.4. „IEEE Standard 754 Floating Point Numbers” – GeeksForGeeks, disponibil online: IEEE Standard 754 Floating Point Numbers - GeeksforGeeks