ADUNARE ȘI SCĂDERE ÎN VIRGULĂ MOBILĂ

Proiect VHDL

POPESC ARIADNA-IOANA
GRUPA 30238
Catedra de Calculatoare
Îndrumător: LIȘMAN DRAGOȘ FLORIN
14.01.2024

UNIVERSITATEA TEHNICĂ CLUJ-NAPOCA

Cuprins

1. Rezumat

Proiectul propus se concentrează pe dezvoltarea unui sumator/scăzător pentru numere în virgulă mobilă conform standardului IEEE 754. Alegerea s-a îndreptat către implementarea în limbajul VHDL, având formatul de precizie simplă (32 de biți) ca focalizare.

În vederea evaluării și testării proiectului, am creat un test bench care a fost folosit pentru a efectua diverse operații asupra a 17 seturi distincte de numere. Am evidențiat rezultatele obținute într-un mod explicit, inclusiv stările intermediare, prin intermediul unei diagrame de undă. Această abordare a avut rolul de a ilustra procesul prin care trec aceste numere în funcție de formatul ales. Însă am ales și testarea fizică pe placa FPGA Basys3, fiind ușor de urmărit.

În concluzie, am reușit să implementăm cu succes un sumator/scăzător pentru numerele în virgulă mobilă. Proiectul a fost supus cu succes testelor utilizând diverse exemple, iar rezultatele corespund obiectivelor inițiale stabilite.

2. Introducere

Adunarea se afirmă ca fiind operația predominantă în funcționarea sistemelor de calcul, cu posibilitatea ca și operații complexe să fie simplificate, reducându-se la o serie de adunări efectuate de către Unitatea Aritmetică Logică (UAL).

Diversitatea modalităților de implementare a operației de adunare aduce cu sine avantaje și dezavantaje legate de viteza de execuție a operațiilor și resursele hardware necesare. Informații detaliate privind aceste aspecte vor fi dezvăluite în secțiunea de Fundamentare Teoretică.

În capitolul de Proiectare și Implementare, am expus modul în care am concretizat implementarea proiectului și scopurile pentru care am ales fiecare modul în parte. În secțiunea de Rezultate Experimentale, am generat exemple specifice pentru a testa operațiile pe diferite valori. Concluziile obținute din acest proiect sunt prezentate în capitolul de Concluzii, iar referințele folosite pentru fundamentarea teoretică sunt enumerate în capitolul de Bibliografie.

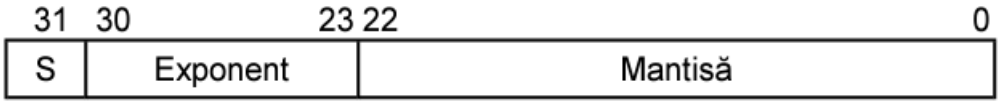
3. Fundamentare teoretică

3.1. Reprezentarea numerelor în VM

În general, un număr N se poate reprezenta în virgulă mobilă (VM) în forma următoare: N = ± M · B ±E (2.25) Un număr reprezentat în VM are două componente. Prima componentă este mantisa (M), care indică valoarea exactă a numărului într-un anumit domeniu, fiind reprezentată de obicei ca un număr fracționar cu semn. A doua componentă este exponentul (E), care indică ordinul de mărime al numărului. În expresia de jos, B este baza exponentului.

$$N = \pm M \cdot B \pm E$$

Această reprezentare poate fi memorată într-un cuvânt binar cu trei câmpuri: semnul, mantisa şi exponentul. De exemplu, presupunând un cuvânt de 32 de biţi, o asignare posibilă a biţilor la fiecare



câmp poate fi următoarea:

Aceasta este o reprezentare în mărime și semn, deoarece semnul are un câmp separat față de restul numărului. Câmpul de semn constă dintr-un bit care indică semnul numărului, 0 pentru un număr pozitiv și 1 pentru un număr negativ. Nu există un câmp rezervat pentru baza B, deoarece această bază este implicită și ea nu trebuie memorată, fiind aceeași pentru toate numerele.

De obicei, câmpul rezervat exponentului nu conține exponentul real, ci o valoare numită caracteristică, care se obține prin adunarea unui deplasament la exponent, astfel încât să rezulte întotdeauna o valoare pozitivă. Astfel, nu este necesar să se rezerve un câmp separat pentru semnul exponentului. Caracteristica C este deci exponentul deplasat:

C = E + deplasament (deplasament = 127)

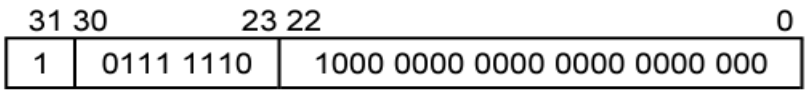
Pentru simplificarea operațiilor cu numere în virgulă mobilă și pentru a crește precizia acestora, se utilizează reprezentarea sub formă normalizată. Un număr în virgulă mobilă este considerat normalizat atunci când are 1 la stânga virgulei. Datorită faptului că bitul din fața virgulei al unui număr normalizat în virgulă mobilă este întotdeauna 1, acest bit nu este de obicei memorat, fiind un bit ascuns la dreapta virgulei binare.

$+1.11 \cdot 2^0$

0	1000 0000	1100 0000 0000 0000 0000 000

Numărul normalizat 1.75 va avea următoarea formă:

Sau numărul –0,75 poate fi scris ca –3/4 sau –0,11 în binar. Reprezentarea numărului în precizie simplă este deci:



3.2. Reprezentarea numerelor în formatul IEEE 754

Standardul IEEE 754 definește următoarele formate sau precizii: precizie simplă, precizie simplă extinsă, precizie dublă şi precizie dublă extinsă. Parametrii principali ai acestor formate sunt prezentaţi în Tabelul 2.7. Standardul nu precizează ca obligatorie implementarea tuturor formatelor, dar recomandă implementarea combinației formatelor cu precizie simplă și precizie simplă extinsă, sau a formatelor cu precizie simplă, precizie dublă și precizie dublă extinsă.

	Precizie simplă	Precizie simplă extinsă	Precizie dublă	Precizie dublă extinsă
Biți ai mantisei	24	≥ 32	53	≥ 64
Exponent real maxim	127	≥ 1023	1023	≥ 16383
Exponent real minim	-126	≤ -1022	-1022	≤ -16382
Deplasament exponent	127	Nespecificat	1023	Nespecificat

Tabelul 2.7. Parametrii formatelor definite de standardul IEEE 754.

Pentru toate formatele, baza implicită este 2. Formatele cu precizie simplă, precizie dublă şi precizie dublă extinsă sunt prezentate în Figura 2.2. Coprocesoarele matematice şi unităţile de calcul în virgulă mobilă ale procesoarelor implementează de obicei aceste formate.

3.3. Adunarea și scăderea în virgulă mobilă

Adunarea și scăderea în VM sunt mai complexe decât înmulțirea și împărțirea. Aceasta deoarece pentru adunarea sau scăderea corectă a celor două numere, trebuie să se realizeze egalizarea exponenților acestora. Aceasta implică compararea mărimii exponenților și apoi alinierea mantisei numărului cu exponentul mai mic. Algoritmul pentru adunare şi scădere are patru etape principale:

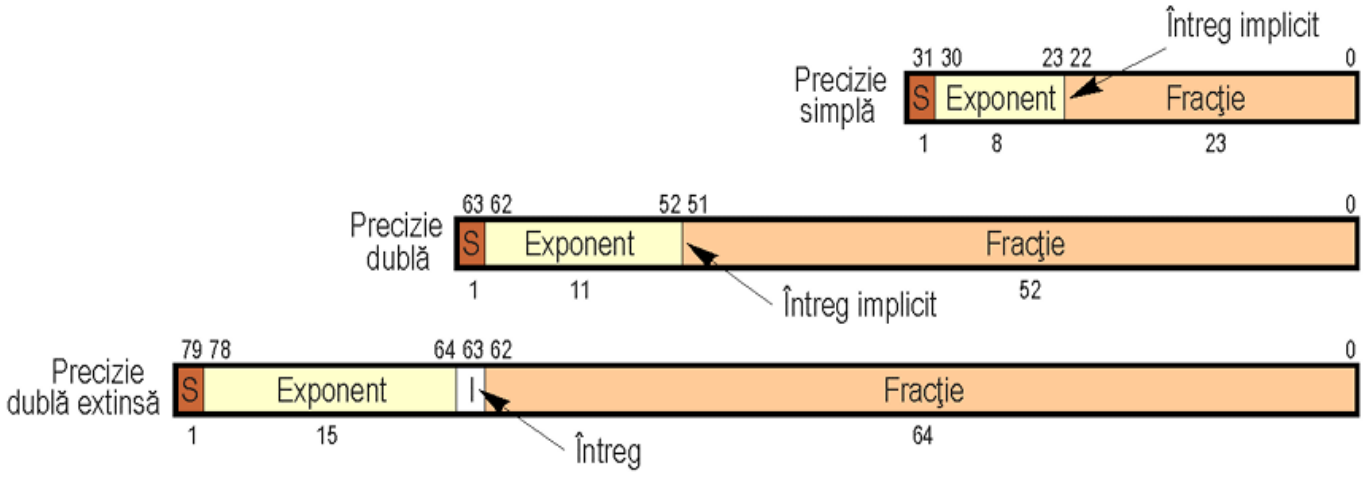


Figura 2.2. Formatele cu precizie simplă, precizie dublă şi precizie dublă extinsă definite de standardul IEEE 754.

1. Alinierea mantiselor

În această etapă, se efectuează o comparație între exponentii celor două numere, iar mantisa asociată cu exponentul mai mic este deplasată spre dreapta. Pentru a determina diferența dintre exponenți, se realizează o scădere. În cazul în care rezultatul este negativ, exponentul al doilea este considerat mai mare; în caz contrar, primul exponent este mai mare, indicând necesitatea deplasării către dreapta cu un număr de biți egal cu această diferență.

2. Adunarea sau scăderea mantiselor

În această fază, se efectuează operația de adunare sau scădere a mantiselor, în funcție de alegerea utilizatorului. Alegerea va fi dată de un switch de pe plăcuța Basys3, explicația fiind enunțată mai jos în partea de implementare.

3. Normalizarea rezultatului

După operația de adunare a mantiselor, se verifică dacă rezultatul este normalizat sau nu. În cazul în care nu este normalizat, se procedează la deplasarea către stânga până când primul bit înaintea virgulei devine egal cu 1. De asemenea aici se va introduce acel adaos de 127, care este adăugat pentru a transla exponentul într-un interval care începe de la zero. Acest lucru este crucial pentru a gestiona numerele atât în partea pozitivă, cât și în cea negativă a spectrului, oferind un interval simetric pentru exponent.

Forma normalizată a unui număr se realizează pentru a simplifica operațiile cu nume în virgulă mobilă și pentru a crește precizia acestora.

4. Rotunjirea rezultatului

În cadrul proiectului am ales opțiunea de trunchiere pentru partea de rotunjire a rezultatului final. Trunchierea este o operație simplă și eficientă, care poate fi implementată cu resurse minime de hardware. Aceasta poate contribui la o execuție mai rapidă a algoritmului, ceea ce este crucial în implementările embedded sau în aplicații cu resurse limitate. Alte metode de rotunjire, cum ar fi rotunjirea la cel mai apropiat sau rotunjirea către zero, implică calcule suplimentare și pot necesita hardware adițional pentru a fi implementate eficient. Prin comparație, trunchierea necesită doar eliminarea zecimalelor fără a introduce operații complexe.

4. Proiectare și implementare

4.1.Proiectarea proiectului

Organigrama pe care m-am bazat pentru acest proiect este cea din cursul de AC(** în bibliografie).

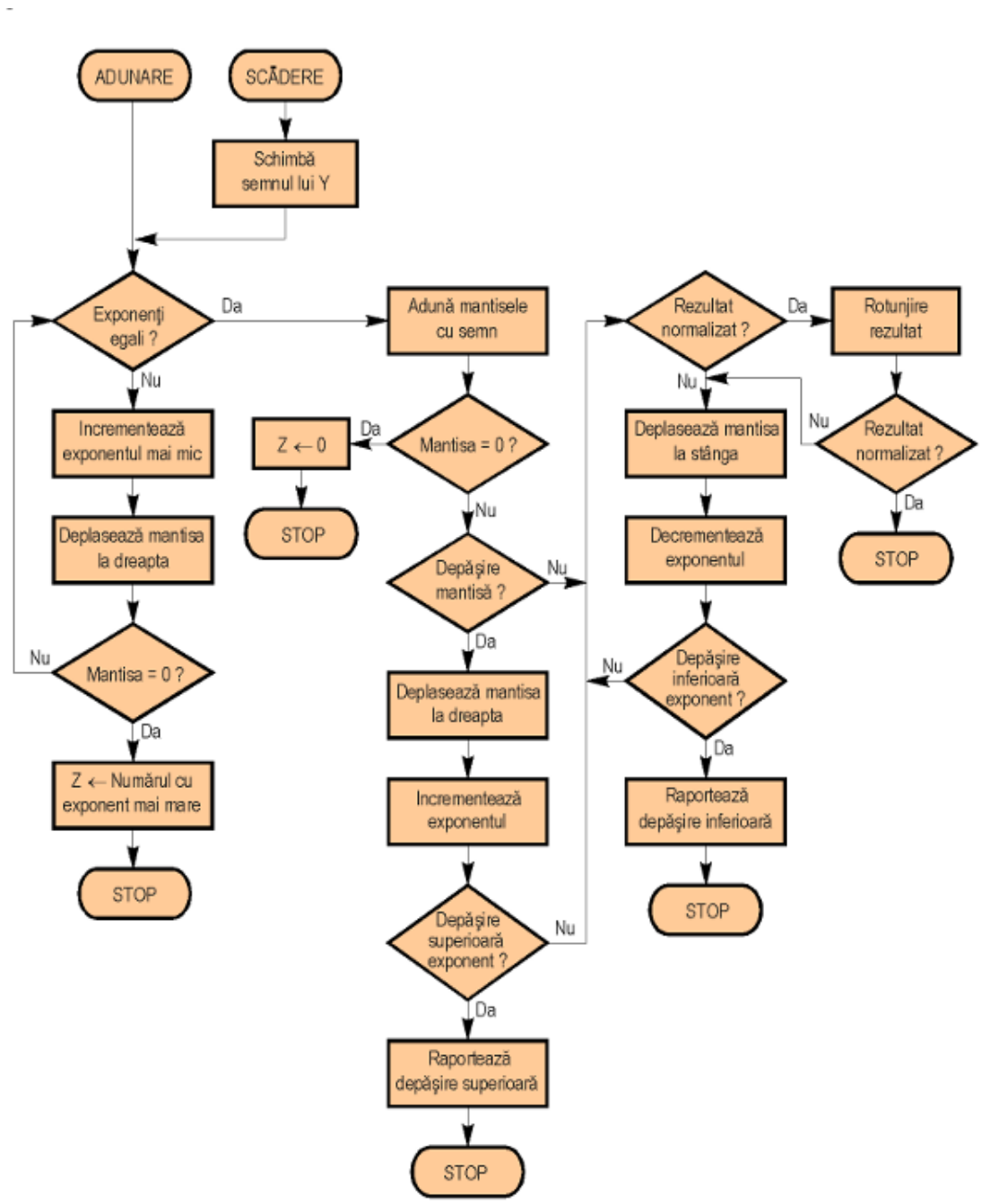
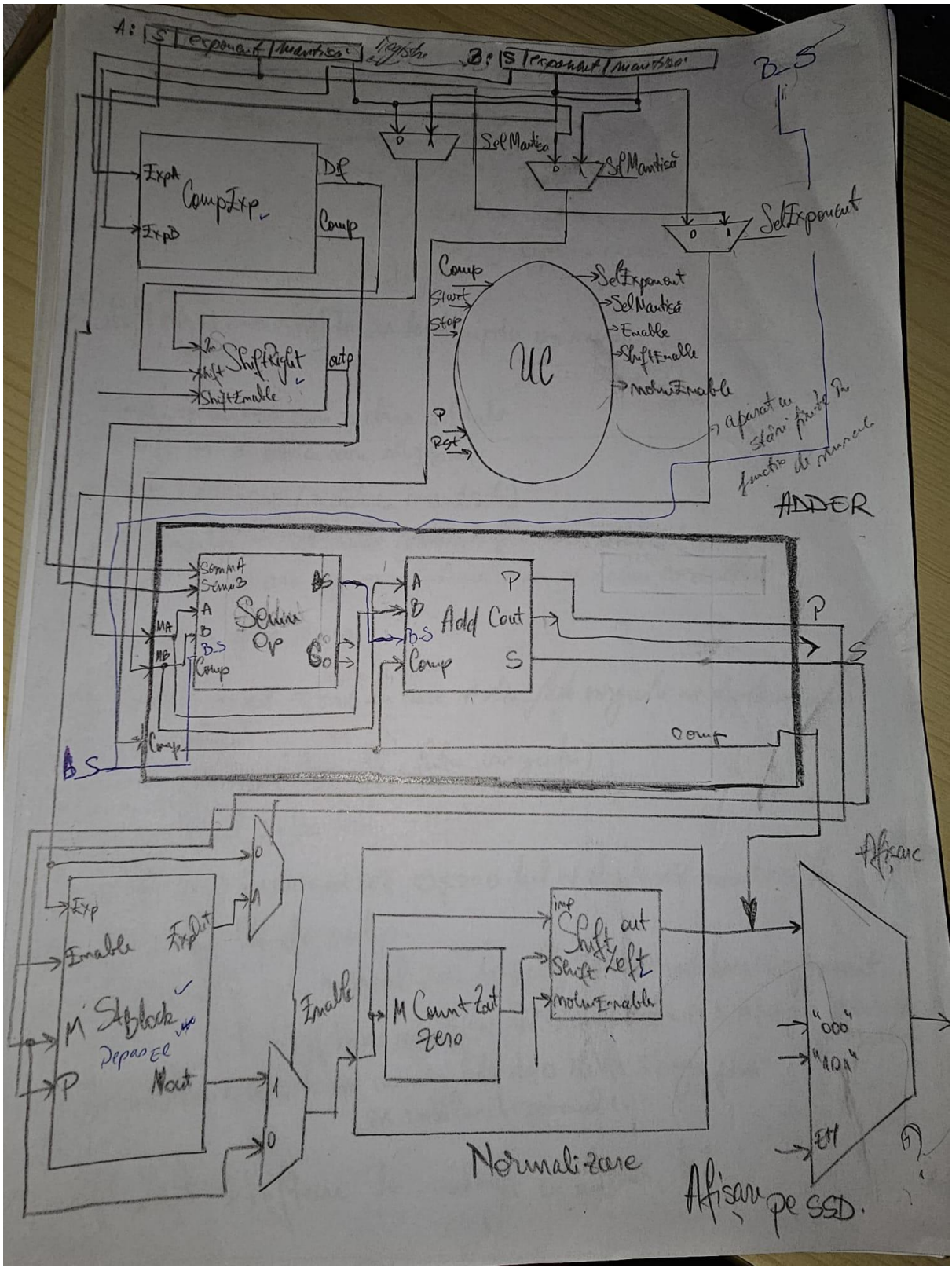


Figura 6.17. Adunarea și scăderea în virgulă mobilă ($z \leftarrow x \pm y$).

Schema bloc a aplicației:



5. Implementarea proiectului

Am împărțit codul în mai multe module:

3.2. CompExp

Acest modul are rolul de a realiza compararea a doi exponenți.

Modulul returnează diferența dintre cei doi exponenți și o valoare numită "Comp". Variabila "Comp" are doi biți și poate avea valorile:

- "00" atunci când expA > expB,
- "01" atunci când expA < expB, și
- "10" în cazul în care exponenții sunt egali.

```
C<="00" when (ExpA>ExpB) else
   "01" when (ExpA<ExpB) else
   "10" when (ExpA=ExpB) else
   "11";

Comp<=C;
```

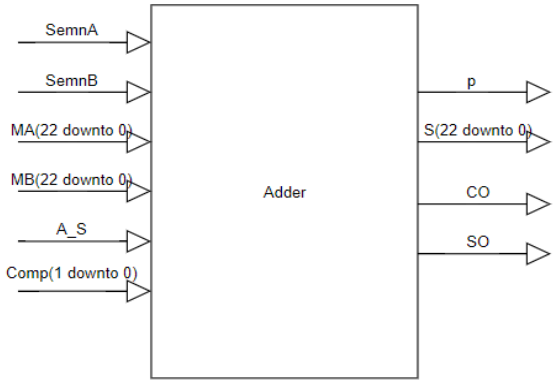
3.3. ShiftRight

Modulul realizează o operațiune de deplasare la dreapta a unui număr cu 23 de biți, utilizând o construcție de tip "case" în funcție de parametrul "shift1". Primul bit adăugat în față este 1, în timp ce restul sunt 0, deoarece înaintea virgulei avem deja un 1. Intrarea "inp" reprezintă mantisa care trebuie supusă operației de deplasare, iar "shift1" indică numărul de biți cu care se dorește efectuarea deplasării.

```
57  process(shift,inp)
58  begin
59      case shift is
60      when "00000" => s5<= inp(22 downto 0);
61      when "00001" => s5<='1' & inp(22 downto 1);
62      when "00010" => s5<="01" & inp(22 downto 2);
63      when "00011" => s5<="001" & inp(22 downto 3);
64      when "00100" => s5<="0001" & inp(22 downto 4);
65      when "00101" => s5<="00001" & inp(22 downto 5);
66      when "00110" => s5<="000001" & inp(22 downto 6);
67      when "00111" => s5<="0000001" & inp(22 downto 7);
68      when "01000" => s5<="00000001" & inp(22 downto 8);
69      when "01001" => s5<="000000001" & inp(22 downto 9);
70      when "01010" => s5<="0000000001" & inp(22 downto 10);
71      when "01011" => s5<="00000000001" & inp(22 downto 11);
72      when "01100" => s5<="000000000001" & inp(22 downto 12);
73      when "01101" => s5<="0000000000001" & inp(22 downto 13);
74      when "01110" => s5<="00000000000001" & inp(22 downto 14);
75      when "01111" => s5<="000000000000001" & inp(22 downto 15);
76      when "10000" => s5<="0000000000000001" & inp(22 downto 16);
77      when "10001" => s5<="00000000000000001" & inp(22 downto 17);
78      when "10010" => s5<="000000000000000001" & inp(22 downto 18);
79      when "10011" => s5<="0000000000000000001" & inp(22 downto 19);
80      when "10100" => s5<="00000000000000000001" & inp(22 downto 20);
81      when "10101" => s5<="000000000000000000001" & inp(22 downto 21);
82      when "10110" => s5<="0000000000000000000001" & inp(22);
83      when others =>  s5<="00000000000000000000001";
84      end case;
85  end process;
```

3.4. Adder

Acest modul efectuează operații de adunare/scădere între mantise și determină semnul rezultatului numeric.



De asemenea conține alte 2 module:

- **Signout**

Acest modul determină semnul operației, semnul numărului care are exponentul mai mic și semnul rezultatului numeric.

```
architecture Behavioral of SignOut is
    signal SB_aux :std_logic;
    signal Aaux,Baux : std_logic_vector(22 downto 0);
begin
    SB_aux <= SemnB xor A_S;

    SO <=SemnA when Comp="00" else
          SB_aux when Comp="01" else
          SemnA when (A<B) else
          SB_aux;
    AS<='1' when SemnA /=SB_aux else
        '0';
```

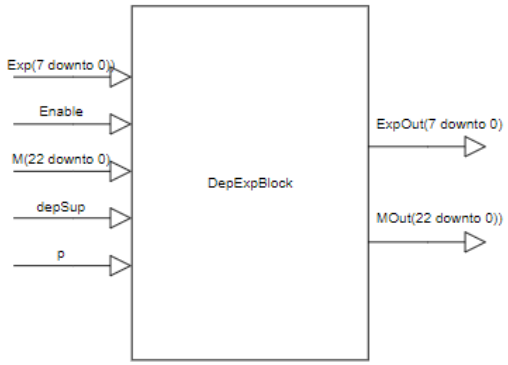
- **FractionAdd**

Acest modul efectuează operația propriu-zisă, folosind sumatoare elementare.

```
nlaux<='1' when Comp/="01" else '0';
n2<='1' when Comp/="00" else '0';

compl: for i in 0 to 22 generate
    Bl(i)<=B(i) xor A_S;
    nl<=nlAux xor A_S;
    sumator_0: if i=0 generate
              sumator_0comp: SumatorElementar port map(Tin=>A_S,Yi=>Bl(i),Xi=>A(i),tout=>aux(i),S=>S_aux(i));
              end generate;
    sumator_i: if (i>0 and i<23) generate
              sumator_icomp: SumatorElementar port map(Tin=>aux(i-1),Yi=>Bl(i),Xi=>A(i),tout=>aux(i),S=>S_aux(i))
            end generate;
end generate;

comp23:for i in 23 to 23 generate
    sumator_23comp: SumatorElementar port map(Tin=>aux(22),Yi=>nl,Xi=>n2,tout=>depNorm,S=>norm);
end generate;

p<=norm;
S<=S_aux;
Cout<=depNorm;
```

3.5. DepExpBlock

Acest modul este folosit pentru cazul în care acel 1 din fața virgulei se deplasează la stânga (ne dăm seama cu ajutorul biților p și Cout, ieșirile de la Adder, unde p= primul bit din fața virgulei, Cout= al doilea bit din fața virgulei). Modulul constă în incrementarea exponentului și în deplasarea mantisei la dreapta cu un bit de valoare p.



3.6. BlockNorm

Acest modul realizează normalizarea rezultatului prin decrementarea exponentului și prin shiftarea mantisei la stânga pana ajunge 1 înaintea virgulei. Conține următoarele 2 module:

- **CountLeadingZero**

  Acest modul numără biții de zero de la stânga mantisei până la întâlnirea primului bit de 1.

```
ZeroVector<="00000000000000000000000";

aux <= X"17" when M(22 downto 0)=ZeroVector(22 downto 0) else
    X"16" when M(22 downto 1)=ZeroVector(22 downto 1) else
    X"15" when M(22 downto 2)=ZeroVector(22 downto 2) else
    X"14" when M(22 downto 3)=ZeroVector(22 downto 3) else
    X"13" when M(22 downto 4)=ZeroVector(22 downto 4) else
    X"12" when M(22 downto 5)=ZeroVector(22 downto 5) else
    X"11" when M(22 downto 6)=ZeroVector(22 downto 6) else
    X"10" when M(22 downto 7)=ZeroVector(22 downto 7) else
    X"0F" when M(22 downto 8)=ZeroVector(22 downto 8) else
    X"0E" when M(22 downto 9)=ZeroVector(22 downto 9) else
    X"0D" when M(22 downto 10)=ZeroVector(22 downto 10) else
    X"0C" when M(22 downto 11)=ZeroVector(22 downto 11) else
    X"0B" when M(22 downto 12)=ZeroVector(22 downto 12) else
    X"0A" when M(22 downto 13)=ZeroVector(22 downto 13) else
    X"09" when M(22 downto 14)=ZeroVector(22 downto 14) else
    X"08" when M(22 downto 15)=ZeroVector(22 downto 15) else
    X"07" when M(22 downto 16)=ZeroVector(22 downto 16) else
    X"06" when M(22 downto 17)=ZeroVector(22 downto 17) else
    X"05" when M(22 downto 18)=ZeroVector(22 downto 18) else
    X"04" when M(22 downto 19)=ZeroVector(22 downto 19) else
    X"03" when M(22 downto 20)=ZeroVector(22 downto 20) else
    X"02" when M(22 downto 21)=ZeroVector(22 downto 21) else
    X"01" when M(22 downto 22)=ZeroVector(22 downto 22) else
    X"00";

    ZCount<=aux(4 downto 0);
```

- **ShiftLeft**

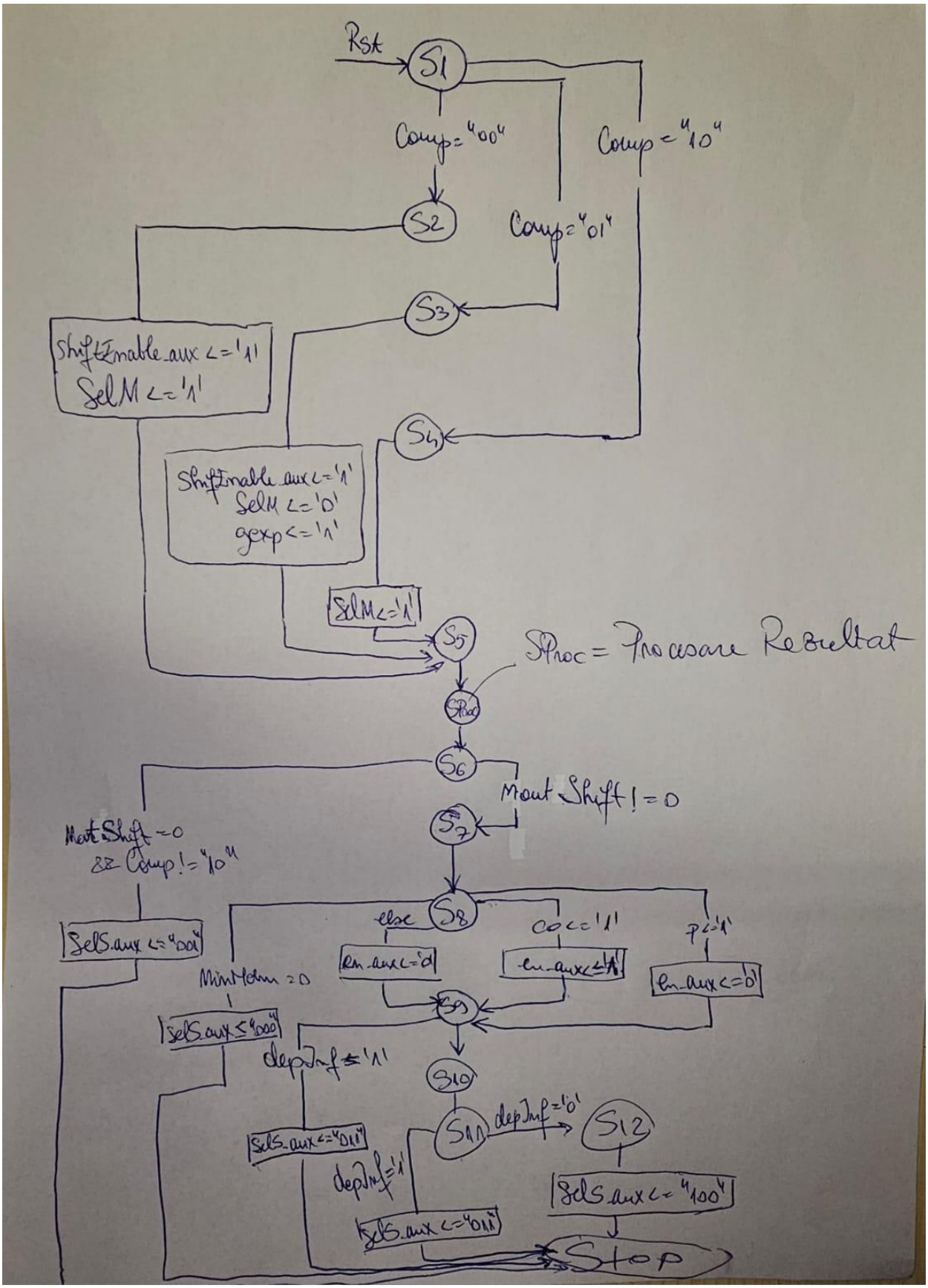
Acest modul realizează operația de deplasare la stânga a mantisei cu un număr specificat de biți.

```
process(normEnable,inp)
begin
    if normEnable='1' then
        case shift1 is
              when "00000" => outp_aux<= inp(22 downto 0);
              when "00001" => outp_aux<= inp(21 downto 0) & '0';
              when "00010" => outp_aux<= inp(20 downto 0) & "00";
              when "00011" => outp_aux<= inp(19 downto 0) & "000";
              when "00100" => outp_aux<= inp(18 downto 0) & "0000";
              when "00101" => outp_aux<= inp(17 downto 0) & "00000";
              when "00110" => outp_aux<= inp(16 downto 0) & "000000";
              when "00111" => outp_aux<= inp(15 downto 0) & "0000000";
              when "01000" => outp_aux<= inp(14 downto 0) & "00000000";
              when "01001" => outp_aux<= inp(13 downto 0) & "000000000";
              when "01010" => outp_aux<= inp(12 downto 0) & "0000000000";
              when "01011" => outp_aux<= inp(11 downto 0) & "00000000000";
              when "01100" => outp_aux<= inp(10 downto 0) & "000000000000";
              when "01101" => outp_aux<= inp(9 downto 0) & "0000000000000";
              when "01110" => outp_aux<= inp(8 downto 0) & "00000000000000";
              when "01111" => outp_aux<= inp(7 downto 0) & "000000000000000";
              when "10000" => outp_aux<= inp(6 downto 0) & "0000000000000000";
              when "10001" => outp_aux<=inp(5 downto 0) & "00000000000000000";
              when "10010" => outp_aux<=inp(4 downto 0) & "000000000000000000";
              when "10011" => outp_aux<=inp(3 downto 0) & "0000000000000000000";
              when "10100" => outp_aux<=inp(2 downto 0) & "00000000000000000000";
              when "10101" => outp_aux<=inp(1 downto 0) & "000000000000000000000";
              when "10110" => outp_aux<=inp(0) & "0000000000000000000000";
              when others =>  outp_aux<="00000000000000000000000";
              end case;
```

3.7. UC

Acest modul implementează un aparat cu stări finite conform organigramei prezentate mai sus. Tabelul cu fiecare semnal în funcție de fiecare stare și diagrama de stări se găsesc mai jos:

Stări	SelExp	SelMantisa	enableLoad	enable	SelS	Term	shiftEnable	normEnable	EnableMoutShift
S1: Initializare + comparare exponenti	gexp	SelM	0	enable_aux	110	0	shiftEnable_aux	0	0
S2: Tratare expA > expB	gexp	SelM	1	enable_aux	110	0	shiftEnable_aux	0	0
S3: Tratare expA < expB	gexp	SelM	1	enable_aux	110	0	shiftEnable_aux	0	0
S4: Tratare expA = expB	gexp	SelM	1	enable_aux	110	0	shiftEnable_aux	0	0
S5: Shiftare dreapta	gexp	SelM	1	enable_aux	110	0	shiftEnable_aux	0	0
Sproc: Procesare rezoltat shiftare	gexp	SelM	1	enable_aux	110	0	shiftEnable_aux	0	1
S6: Decizie in functie de razultatul shiftarii	gexp	SelM	0	enable_aux	110	0	shiftEnable_aux	0	0
S7: Adunare/Scadere	gexp	SelM	0	enable_aux	110	0	shiftEnable_aux	0	0
S8: Decizie in functie de rezultatul adunarii	gexp	SelM	0	enable_aux	110	0	shiftEnable_aux	0	0
S9: Verificare depasire superioara	gexp	SelM	0	enable_aux	110	0	shiftEnable_aux	0	0
S10: Normalizare	gexp	SelM	0	enable_aux	110	0	shiftEnable_aux	1	0
S11: Verificare depasire inferioara	gexp	SelM	0	enable_aux	110	0	shiftEnable_aux	1	0
S12: Procesare rezultat final	gexp	SelM	0	enable_aux	110	0	shiftEnable_aux	1	0
Sop: STOP	gexp	SelM	0	enable_aux	Sels_aux	1	shiftEnable_aux	1	0



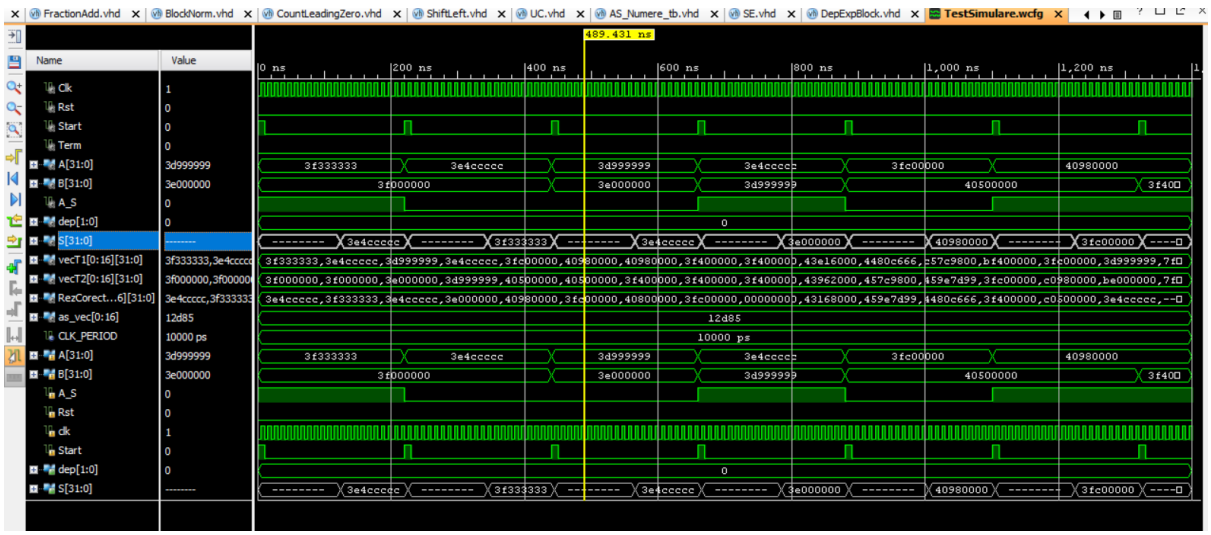
3.8. Main

Acest modul este modulul principal, care conectează toate modulele prezentate mai sus.

4. Rezultate experimentale

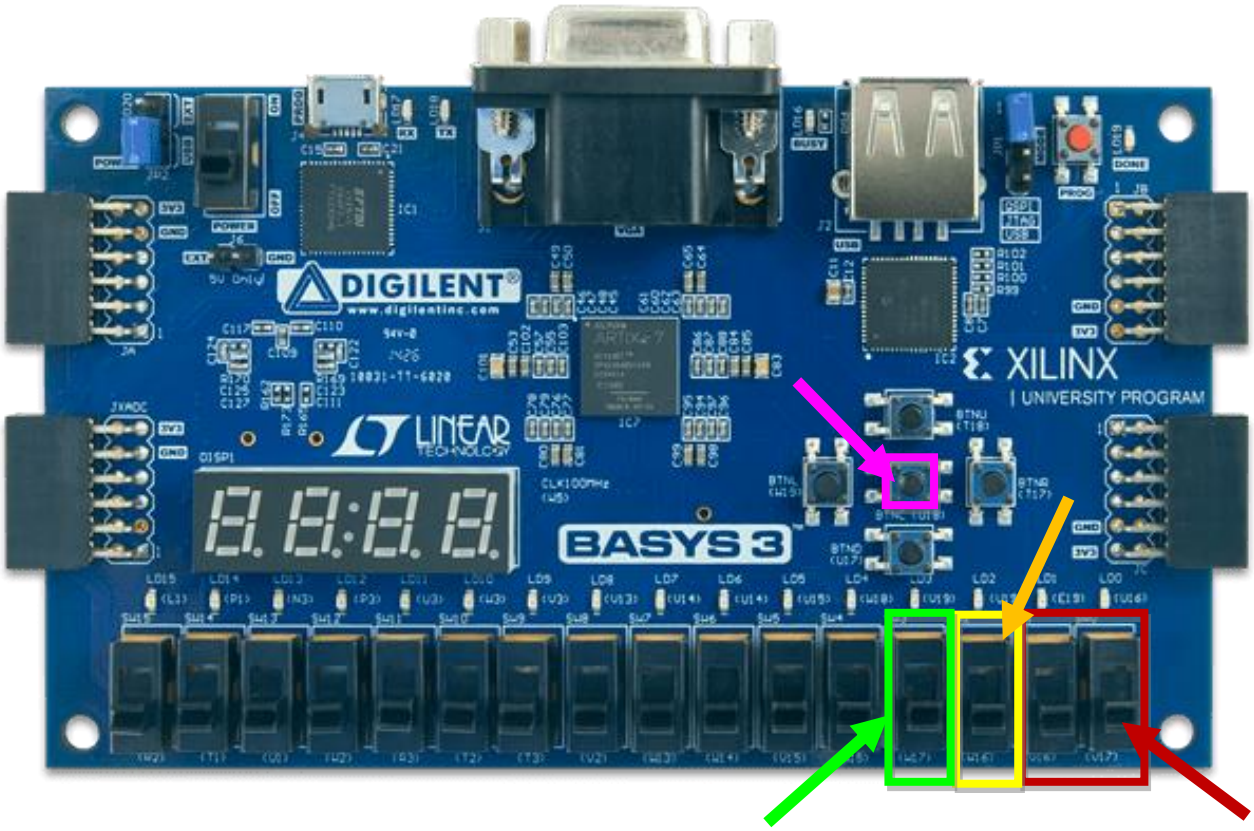
Pentru partea de testare am făcut un testbench pentru a simula și în Vivado, însă am făcut și testarea pentru placă. În partea de simulare am creat un testbench pentru a testa, în care am inserat mai multe semnale pentru a verifica apoi în funcție de fiecare element din vector.

```
signal Clk :  std_logic;
signal Rst :  std_logic;
signal Start :  std_logic;
signal Term :std_logic:='0';
signal A : STD_LOGIC_VECTOR (31 downto 0);
signal B:  STD_LOGIC_VECTOR (31 downto 0);
signal A_S: std_logic;
signal dep: std_logic_vector(1 downto 0);
signal S :  STD_LOGIC_VECTOR (31 downto 0);
CONSTANT CLK_PERIOD : TIME := 10 ns;

type vecType is array(0 to 16) of std_logic_vector(31 downto 0);
signal vecT1: vecType:=(x"3F333333",x"3E4CCCCC",x"3D999999",x"3E4CCCCC",x"3FC00000",x"40980000",x"40980000",x"3F400000",
                        x"3F400000",x"43E16000",x"4480C666",x"C57C9800",x"BF400000",x"3FC00000",x"3D999999",x"7F800000",x"00400000");
signal vecT2: vecType:=(x"3F000000",x"3F000000",x"3E000000",x"3D999999",x"40500000",x"40500000",x"3F400000",x"3F400000",
                        x"3F400000",x"43962000",x"457C9800",x"459E7D99",x"3FC00000",x"C0980000",x"BE000000",x"7F800000",x"00000000");

signal RezCorect:vecType:=(x"3E4CCCCC",x"3F333333",x"3E4CCCCC",x"3E000000",x"40980000",x"3FC00000",x"40800000",x"3FC00000",
                        x"00000000",x"43168000",x"459E7D99",x"4480C666",x"3F400000",x"C0500000",x"3E4CCCCC","----------------------------

signal as_vec: std_logic_vector(0 to 16) :=('1','0','0','1','0','1','1','0',
                                            '1','1','0','0','0','0','1','0','1');
```



Cât despre partea de testare pe placă, aceasta se va face pe o placă FPGA Basys3. În modulul principal am făcut un MUX cu un switch pentru afișarea de 16b și 16b, unde fiecare număr va fi afișat în jumătate(primii 8 octeți și următorii 8). Această afișare va fi determinată de un switch care funcționează ca selecție în cel de-al doilea MUX, acest switch fiind al 3-lea de pe placa(**W16**), iar cele 2 pentru prima și a doua jumătate a numărului fiind primul și al doilea(**V17** și **V16**). Valorile celor două numere sunt hardcodate direct pe intrările A și B. Cât despre testarea pentru operație: pe switch-ul al 5-lea(sw(15) în cod și **W17** pe placă) se va decide, dacă este pe 0 va fi adunare, iar dacă este pe 1 va fi scădere. De asemenea, rezultatul va fi afișat când pe switch-urile V16 si V17 va fi selectat 1 și 0. Însă pentru a testa cu succes, va fi nevoie să se selecteze switch-ul operației, iar apoi butonul de Rst(care este pe **U18**) și doar după se va afișa bine valoarea operației.

```
A <= x"40f00000"; -- +7.5
-- A <= x"c0f00000"; -7.5
B <= x"3fa66666"; -- +1.3
-- B <= x"bfa66666"; -1.3
A_S<=sw(3);

MUX1:process(sw(1 downto 0))
     begin
      case sw(1 downto 0) is
          when "00"=> iesire <= A;          --a
          when "01"=> iesire <= B;          --b
          when "10"=> iesire <= Rez;
          when others=> iesire <= "00000000000000000000000000000000";
      end case;
end process;

MUX2:process(sw(2))
     begin
        case sw(2) is
            when '0' => afis <= iesire(31 downto 16);
            when others => afis <= iesire(15  downto 0);
        end case;
end process;
```



5. Concluzii

În cadrul acestui proiect, am realizat implementarea operațiilor de adunare și scădere a numerelor în virgulă mobilă, adunând astfel informații valoroase despre Standardul IEEE 754 și despre modalitatea în care aceste operații sunt efectuate. Pe parcursul acestui demers, mi-am consolidat abilitățile de a scrie cod în limbajul VHDL.

Pentru dezvoltări ulterioare, proiectul ar putea fi integrat într-o aplicație extinsă, cum ar fi un calculator specializat pentru manipularea și efectuarea de operații pe numere exprimate în binar. Aceasta ar oferi o platformă pentru utilizarea eficientă a operațiilor implementate, contribuind la extinderea funcționalităților unei aplicații mai ample.

6. Bibliografie

https://digitalsystemdesign.in/floating-point-addition-and-subtraction/

https://www.doc.ic.ac.uk/~eedwards/compsys/float/

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

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

https://www.xilinx.com/content/dam/xilinx/support/documents/application_notes/xap p599-floating-point-vivado-hls.pdf

http://www.ijirset.com/upload/2017/december/23_IEEE%20754_IJ61212604%20_1_.pdf

https://web.cecs.pdx.edu/~mperkows/CLASS_VHDL/VHDL_CLASS_2001/Floating/projreport.html

https://www.h-schmidt.net/FloatConverter/IEEE754.html

https://iremi.univ-reunion.fr/IMG/pdf/ieee-754-2008.pdf