Assignment Curs - Implementarea unui Pipeline Aritmetic pentru Adunarea în Virgulă Mobilă

Disciplina: Structura Sistemelor de Calcul

Student: Maria-Magdalena Creț

Grupa 30223

Anul Universitar 2024-2025

Cuprins

1 Introducere

În secțiuniile următoare se va descrie implementarea unui pipeline adder pentru numere în virgulă mobilă folosind standardul IEEE 754 single precision. Implementarea este realizată în VHDL și oferă o soluție eficientă pentru adunarea numerelor în virgulă mobilă cu latență fixă și throughput ridicat.

Formatul de date cu care se va lucra este următorul:

- Total: 32 biți
- Semn: 1 bit (MSB)
- Exponent: 8 biți
- Mantisă: 23 biți

Obiectiele acestui pipeline pentru adunarea cu virgulă mobilă sunt:

- Reducerea timpului de execuție prin procesare paralelă
- Latență fixă de 4 cicluri de ceas
- Frecvență de operare optimizată
- Precizie de calcul conform standardului
- Pipeline balansat între etape
- Scalabilitate

2 Analiza Etapelor Pipelined

Pentru început, se remarcă vis-a-vis de numărul de biți aleși pentru exponent și pentru mantisa. Mai precis 23 pentru mantisă și 8 pentru exponent. Se lucrează cu un Pipeline pe 32 de biți. De precizat este (conform comentariului adăugat în primul fișier de cod - cel cu implementarea) că în formatul IEEE 754 single precision (32 biți), numărul este împărțit în: 1 bit pentru semn (cel mai semnificativ bit - MSB), 8 biți pentru exponent (următorii biți), respectiv 23 biți pentru mantisă (biții rămași). Prin însumarea acestora se ajunge la 32 de biți. Motivele pentru alegerea acestor valori specifice sunt: Pentru exponent (8 biți): permite reprezentarea exponentului în intervalul [-127, 128] și este suficient pentru majoritatea aplicațiilor generale. Iar pentru mantisă (23 biți): oferă aproximativ 7 cifre zecimale de precizie. De asemenea. include un "hidden bit" implicit (care nu este stocat). Precizia efectivă este de 24 biți datorită hidden bit-ului.

Stage 1: Comparare

- **Latență**: 1 ciclu de ceas
- **Operații Critice**:
  - Extracția paralelă a semnelor, exponenților și mantiselor
  - Tratarea bitului ascuns pentru numerele denormalizate

Stage 2: Aliniere

- **Latență**: 1 ciclu de ceas
- **Operații Critice**:
  - Deplasarea mantisei (până la 23 de poziții)
  - De obicei, această etapă reprezintă calea critică din cauza utilizării shifter-ului de tip rotire.
- **Utilizare Resurse**:
  - Comparatoare pentru calcularea diferenței dintre
  - Shifter de tip rotire exponenți

Stage 3: Adunare sau scădere mantise

- **Latență**: 1 ciclu de ceas
- **Operații Critice**:
  - Adunarea mantiselor
  - Deplasare de normalizare cu o poziție
  - Ajustarea exponenților
- **Utilizare Resurse**:
  - Adunător pe 24 de biți
  - Shifter mic pentru normalizare

Stage 4: Normalizare

- **Latență**: 1 ciclu de ceas
- **Operații Critice**:
  - Verificarea depășirii mantisei
  - Normalizare fără depășire
  - Transmiterea semnului și validității

3 Implementare

Implementarea s-a realizat pe baza diagramei de mai jos, respectând etapele de pipeline pentru adunarea cu virgulă mobilă:

3.1 Stage 1: Extracție Componente (Comparare)

Funcționalități:

- Separare paralelă semn/exponent/mantisă
- Adăugare bit implicit la mantisă
- Validare date intrare

Semnale cheie:

```
signal stage1_sign_array : stage1_signs;
signal stage1_exp_array  : stage1_exps;
signal stage1_mant_array : stage1_mants;
```

Se extrag semnul, exponentul și mantisa din `nr1` și `nr2`.

Se adaugă bitul ascuns (*hidden bit*) mantiselor pentru a asigura precizia completă.

Se setează semnalul `stage1_valid` dacă datele de intrare sunt valide. **Cod relevant:**

```
stage1_sign_array(0) <= nr1(31);
stage1_sign_array(1) <= nr2(31);
stage1_exp_array(0) <= unsigned(nr1(30 downto 23));
stage1_exp_array(1) <= unsigned(nr2(30 downto 23));
stage1_mant_array(0) <= '1' & unsigned(nr1(22 downto 0));
stage1_mant_array(1) <= '1' & unsigned(nr2(22 downto 0));
```

3.2 Stage 2: Aliniere

Operații principale:

- Comparare exponenți
- Calcul diferență exponenți
- Shift mantisă număr mai mic

Semnale importante:

```
signal stage2_sign_larger : sign_type;
signal stage2_larger_exp  : exp_type;
signal stage2_larger_mant : mantissa_type;
```

Se compară exponenții numerelor. Numărul cu exponent mai mare devine "mai mare" (*larger*), iar celălalt este ajustat (*smaller*) prin deplasare la dreapta.

Diferența dintre exponenți (`exp_diff`) este utilizată pentru alinierea mantiselor.

Cod relevant:

```
if stage1_exp_array(0) >= stage1_exp_array(1) then
    stage2_larger_exp <= stage1_exp_array(0);
    stage2_larger_mant <= stage1_mant_array(0);
    stage2_sign_larger <= stage1_sign_array(0);
    exp_diff := to_integer(stage1_exp_array(0) - stage1_exp_array(1));
else
    stage2_larger_exp <= stage1_exp_array(1);
    stage2_larger_mant <= stage1_mant_array(1);
    stage2_sign_larger <= stage1_sign_array(1);
    exp_diff := to_integer(stage1_exp_array(1) - stage1_exp_array(0));
end if;

stage2_smaller_mant <= shift_right(resize(stage1_mant_array(1), 2*
   MANTISSA_BITS+2), exp_diff);
```

3.3 Stage 3: Adunare și scădere mantise

În această etapă, mantisele celor două numere de intrare sunt ajustate astfel încât să fie aliniate, iar operația de adunare sau scădere este realizată pe baza semnului fiecărui număr.

Funcționalități:

- Adunare/scădere mantise aliniate

Semnale critice:

```
signal stage3_sign : sign_type;
signal stage3_exp  : exp_type;
signal stage3_sum  : sum_type;
```

Mantisele sunt fie adunate, fie scăzute, în funcție de semn.

Exponentul final este cel mai mare exponent.

- **Alinierea mantisei mai mici:** Dacă există o diferență între exponenți, mantisa mai mică este deplasată la dreapta pentru a alinia pozițiile zecimale, păstrând astfel precizia calculului. Această operație este implementată folosind un `shift_right`.
- **Operația matematică (adunare sau scădere):**
  - Dacă semnele celor două numere sunt identice (`stage1_sign_array` și `nr2(31)`), se efectuează adunarea mantiselor.
  - Dacă semnele sunt diferite, se realizează scăderea mantiselor, iar semnul rezultatului final este păstrat din mantisa mai mare.
- **Salvarea exponentului:** După efectuarea operației, exponentul mai mare este propagat înainte, deoarece acesta corespunde poziției zecimale aliniate.
- **Semnalizarea validității etapei:** Semnalul de validitate (`stage3_valid`) este actualizat pe baza valorii semnalului anterior de validitate (`stage2_valid`).

Cod relevant:

```
1     if stage1_sign_array = nr2(31) then
2                 -- Semne egale, adunare
3                 stage3_sum <= resize(stage2_larger_mant, MANTISSA_BITS
                     +3) + resize(smaller_mant_shifted, MANTISSA_BITS+3);
4                 stage3_sign <= stage1_sign_array;
5      else
6                 -- Semne diferite, scadere
7                 stage3_sum <= resize(stage2_larger_mant, MANTISSA_BITS
                     +3) - resize(smaller_mant_shifted, MANTISSA_BITS+3);
8                 -- Semnul se pastreaza de la mantisa mai mare
9                 stage3_sign <= stage2_larger_mant(MANTISSA_BITS);
10    end if;
11
12    stage3_exp <= stage2_larger_exp;
13    stage3_valid <= stage2_valid;
```

3.4 Stage 4: Normalizare

Etapa 4 din pipeline-ul adderului pentru numere în virgulă mobilă are rolul de a normaliza rezultatul intermediar astfel încât să respecte standardul IEEE 754. În această etapă se ajustează mantisa și exponentul pentru a obține o reprezentare corectă.

Funcționalități:

- **Verificarea depășirii mantisei:** Dacă rezultatul adunării/scăderii mantiselor din etapa 3 are bitul cel mai semnificativ (`MANTISSA_BITS+2`) setat la `1`, mantisa este mutată spre dreapta, iar exponentul este incrementat cu `1`.
- **Normalizare fără depășire:** Dacă bitul cel mai semnificativ nu este `1`, mantisa și exponentul rămân nemodificate.
- **Transmiterea semnului și validității:** Semnul rezultatului este transmis mai departe din etapa 3, iar semnalul de validitate este activat dacă datele sunt valide.

Cod relevant:

```
process(clk, reset)
begin
    if reset = '1' then
        stage4_valid <= '0';
    elsif rising_edge(clk) then
        if stage3_sum(MANTISSA_BITS+2) = '1' then
            stage4_mant_norm <= stage3_sum(MANTISSA_BITS+1 downto 1);
               -- Mutare mantisa
            stage4_exp_norm <= stage3_exp + 1;
               -- Cre tere exponent
        else
            stage4_mant_norm <= stage3_sum(MANTISSA_BITS downto 0);
               -- Mantisa nemodificat
            stage4_exp_norm <= stage3_exp;
               -- Exponent nemodificat
        end if;

        stage4_sign <= stage3_sign;    -- Semnul rezultatului
        stage4_valid <= stage3_valid; -- Semnalul de validitate
    end if;
end process;
```

La finalul acestei etape:

- Mantisa este normalizată, astfel încât primul bit semnificativ să fie `1`.
- Exponentul este ajustat pentru a păstra valoarea corectă.
- Semnul și semnalul de validitate sunt propagate mai departe.

Rezultatul poate fi combinat într-o reprezentare finală conform formatului IEEE 754.

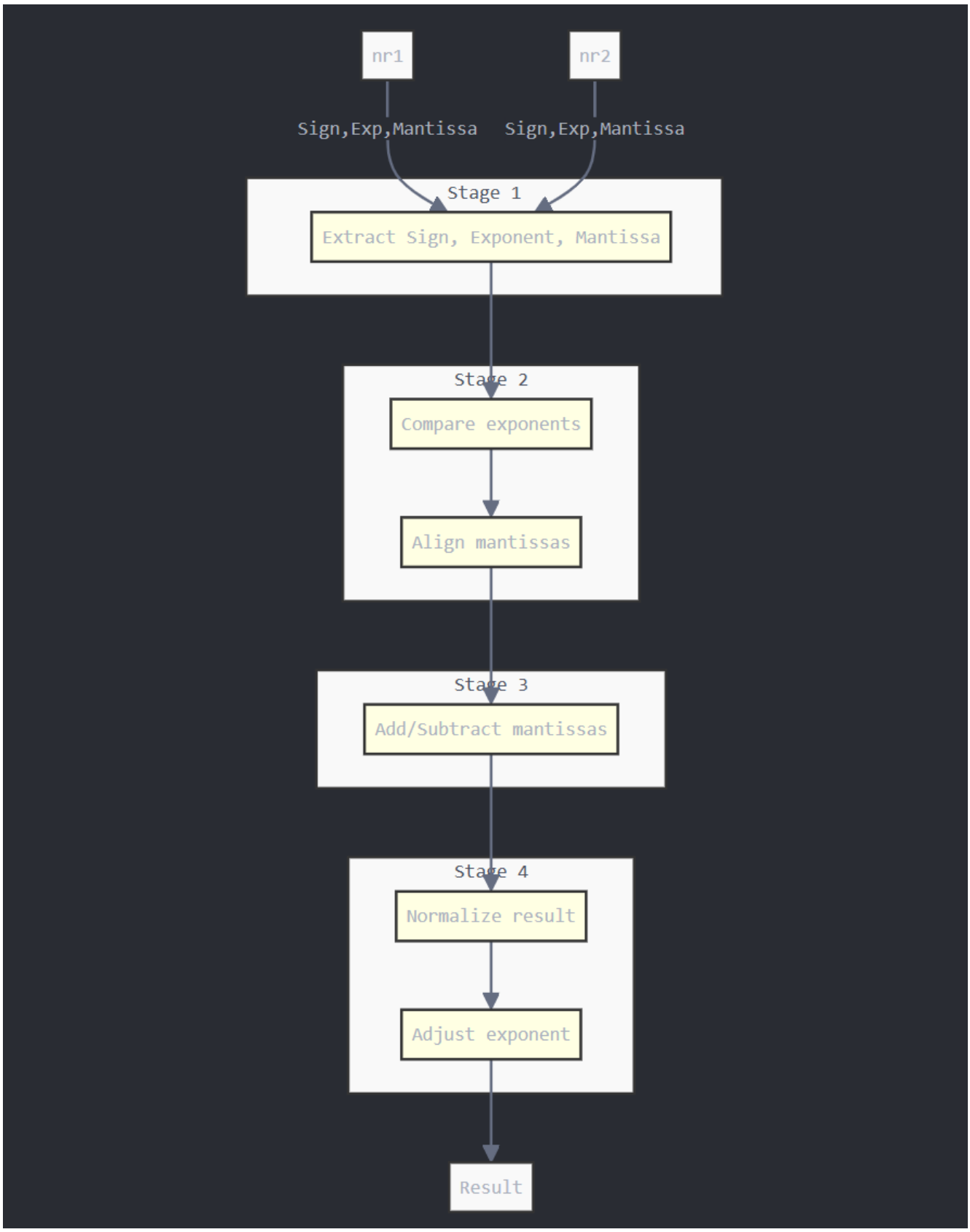

Figura 1: Implementare Floating Adder Pipeline - stages

3.5 Aspecte Tehnice Detaliate

3.5.1 Managementul Datelor

Tipuri de date specializate:

```
subtype mantissa_type is unsigned(MANTISSA_BITS downto 0);
subtype large_mantissa_type is unsigned(2*MANTISSA_BITS+1 downto 0);
subtype sum_type is unsigned(MANTISSA_BITS+2 downto 0);
```

3.5.2 Control Flow

- Reset sincron
- Validare input/output
- Propagare semnale valid între etape

3.5.3 Timing și Sincronizare

- Clock-uri: Rising edge triggering
- Latențe:
  - Stage 1: 1 ciclu
  - Stage 2: 1 ciclu
  - Stage 3: 1 ciclu
  - Stage 4: 1 ciclu
- Throughput: 1 rezultat/ciclu după latență inițială

4 Performanță

Metrici de performanță:

1. **Latența:** este fixă pentru 4 cicluri de la intrare la ieșire. Totodată nu există variații de latență dependente de date. Semnalul `valid_out` indică disponibilitatea rezultatului.
2. **Throughput:** Un rezultat la fiecare ciclu de ceas după o latență inițială de 4 cicluri. Debit efectiv limitat de rata datelor de intrare și de semnalul `valid_in`.
3. Logica combinatorie este determinată de shifter-ul cu rotație din Etapa 2.
4. **Frecvența Ceasului:** Calea critică se află în Etapa 2, adică în etapa de aliniere. Frecvența va depinde de dispozitivul specific și de optimizare.

5 Simulare

5.1 Structura Testbench

```
entity fp_adder_pipeline_tb is
end fp_adder_pipeline_tb;
```

5.2 Semnale de Test

- Clock: Perioadă de 20ns
- Reset: Activ high
- Semnale de date: nr1, nr2 (32 biți)
- Semnale de control: valid_input, valid_outup

5.3 Cazuri de Test

Test 1: Adunare Numere Pozitiv-Negativ

nr1 = 11000000101000000000000000000000 (5.0)

nr2 = 01000000101000000000000000000000 (-5.0)

Rezultat așteptat: 00000000000000000000000000000000 (0.0)

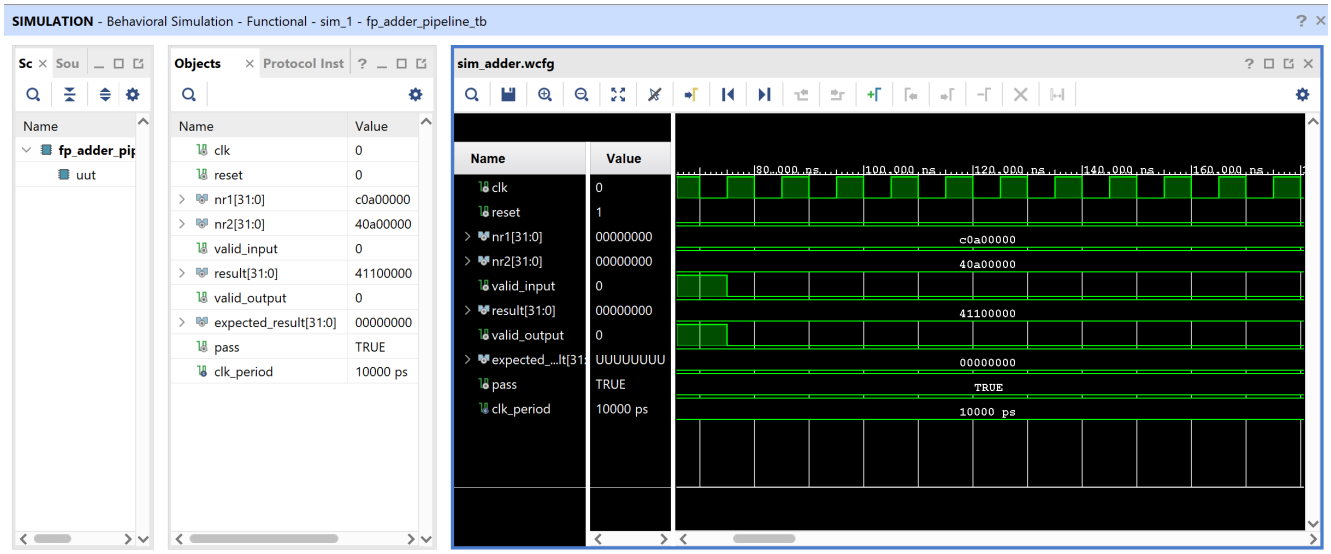

Figura 2: Testare Cazul 1

Test 2: Adunare cu Overflow

nr1 = 01111111000000000000000000000000 Float Max

nr2 = 00111111100000000000000000000000 (1.0)

Rezultat așteptat: 01111111000000000000000000000000 Float Max

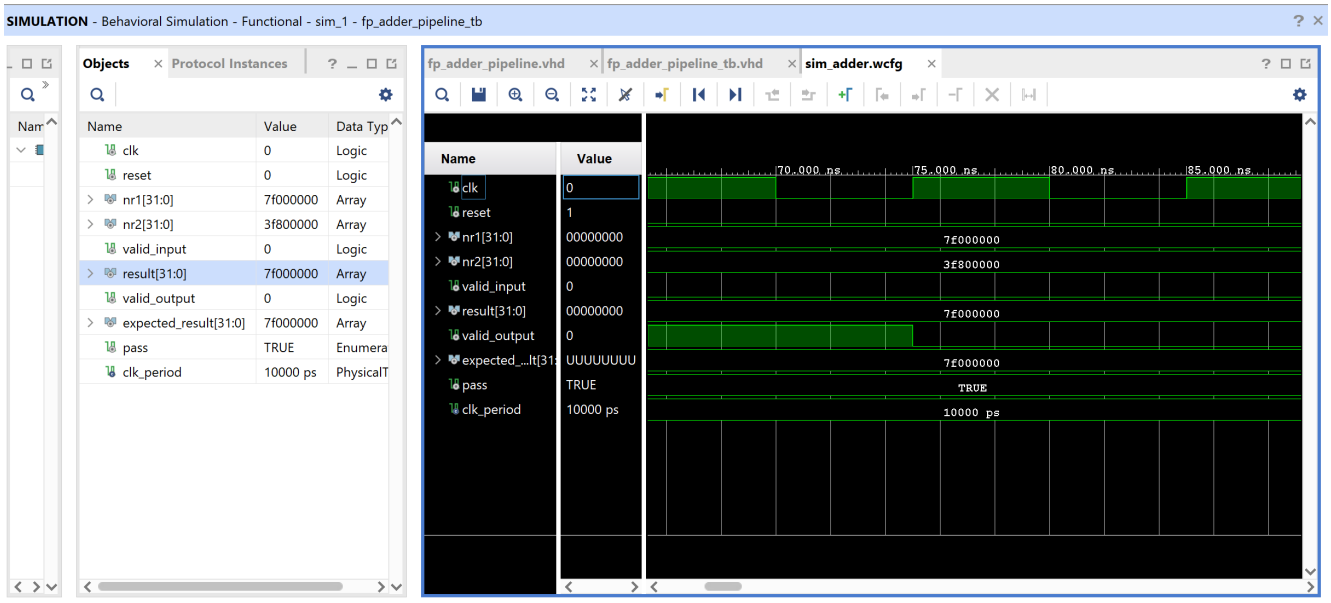

Figura 3: Testare Cazul 2

Test 3: Adunare cu numere cu zecimale

nr1 = 01000000100110011001100110011010 (4.6)

nr2 = 01000000010011001100110011001101 (3.2)

Rezultat așteptat: 01000001000001100110011001100110 (7.8)

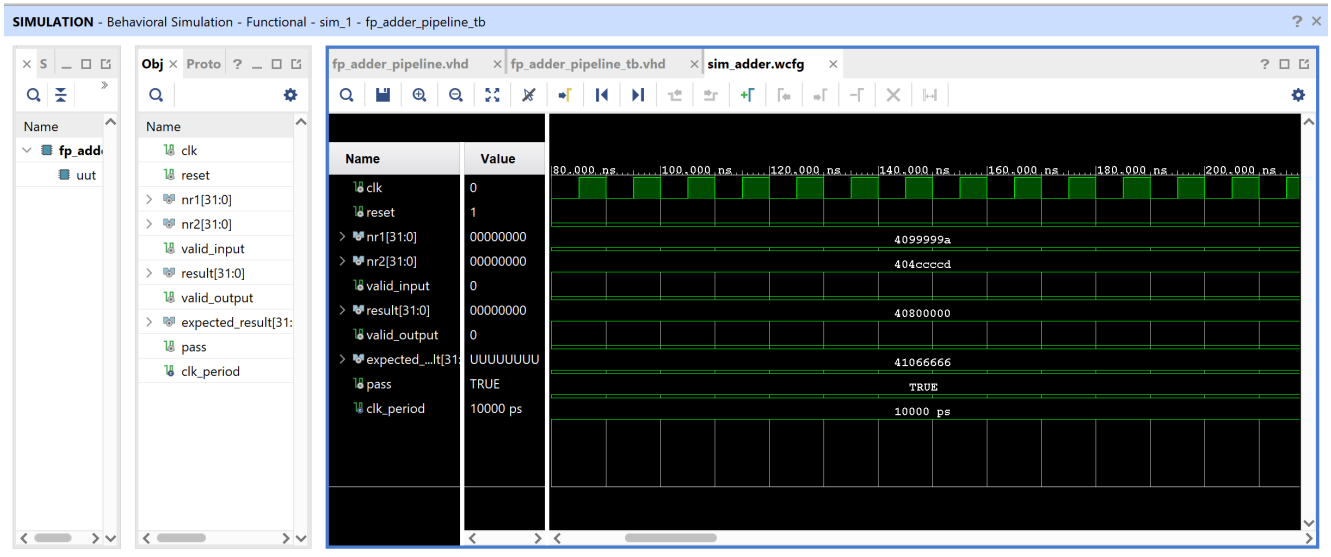

Figura 4: Testare Cazul 2

Test 4: Adunare numere simple

nr1 = 01000001010000000000000000000000 (12.0)

nr2 = 01000000101000000000000000000000 (5.0)

Rezultat așteptat: 01000001100100000000000000000000 (17.0)

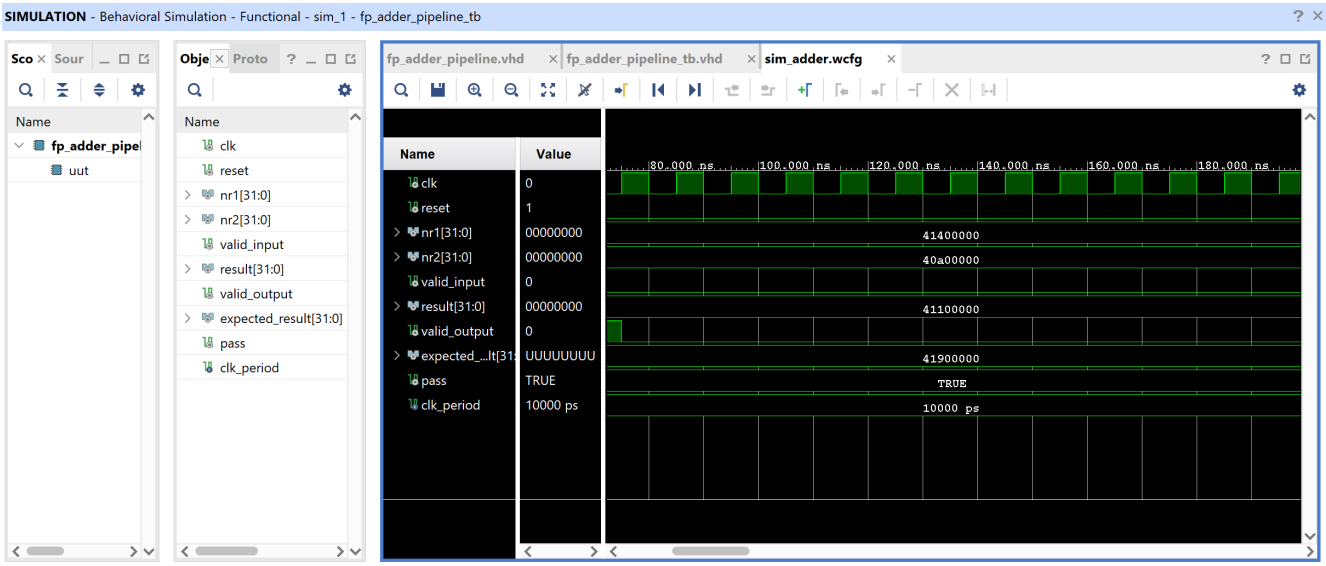

Figura 5: Testare Cazul 4

Explicație Cod

Testbench-ul include:

- Generare semnal clock cu perioadă 20ns
- Secvență de reset la început (2 perioade clock)
- Activare valid_input după reset

Testare secvențială a trei cazuri distincte:

- Adunare numere pozitive
- Operații cu numere de semne diferite
- Verificare comportament pentru rezultate negative

6 Concluzie

Implementarea unui *Floating Point Adder Pipeline* constă într-un proces secvențial care realizează adunarea sau scăderea a două numere în virgulă mobilă, respectând standardele IEEE 754. Arhitectura pipeline este împărțită în patru etape esențiale, fiecare contribuind la procesarea parțială a datelor pentru a optimiza latența și a permite paralelismul. Această implementare pipeline aduce numeroase avantaje:

- **Performanță crescută:** Divizarea operațiilor în etape permite suprapunerea lor, ceea ce reduce latența generală.
- **Modularitate:** Fiecare etapă este responsabilă de un pas bine definit în procesul de calcul, permițând reutilizarea și îmbunătățirea separată a fiecărui modul.
- **Precizie:** Alinierea corectă a mantiselor, semnul rezultatului și normalizarea sunt asigurate, respectând cerințele standardului IEEE 754.
- **Flexibilitate:** Arhitectura poate fi extinsă pentru a sprijini alte operații în virgulă mobilă, cum ar fi înmulțirea sau împărțirea.

Posibile optimizări

Modificarea Etapelor

Etapa 2 ar putea fi împărțită în două etape:

- Compararea exponenților și calcularea cantității de deplasare
- Operația efectivă de deplasare

Acest lucru ar crește latența la 4 cicluri, dar ar îmbunătăți frecvența maximă.

Precizie

Implementarea actuală menține precizia completă.

Se poate reduce dimensiunea shifter-ului pentru aplicații care tolerează o precizie mai mică.

Gestionarea Cazurilor Speciale

Adăugarea detectării timpurii pentru intrări zero (când se realizează adunare cu 0, acest lucru să poată fi detectat mai rapid și optimizat ca să nu mai treacă prin toate etapele).

Implementarea unei căi rapide pentru exponenți egali.

Bibliografie

[1] https://www.javatpoint.com/arithmetic-pipeline

[2] https://www.researchgate.net/publication/3044180$_{A}n_{I}EEE_{c}ompliant_{f}loating$ – $point_{a}dder_{t}hat_{c}onforms_{w}ith_{t}he_{p}ipeline_{p}acket - forwarding_{p}aradigm$

[3] https://www.philadelphia.edu.jo/academics/kaubaidy/uploads/ACA-Lect5.pdf

Anexe

Mai jos s-au anexat codurile modificate pentru implemetare, respectiv pentru simulare.

Codul corespunzător implementării este următorul:

```
------------------------------------------------------------------
----- Universitatea Tehnica din Cluj-Napoca
----- Facultatea de Automatica si Calculatoare
----- 2024-2025
----- Assignment Curs - testbench pentru adunarea in virgula mobila
----- Student: Cret Maria Magdalena
----- Grupa: 30233 Semigrupa 1
----- Fisier Implementare Floating Adder Pipeline ------
------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity fp_adder_pipeline is
    port (
        clk         : in  std_logic;
        reset       : in  std_logic;
        nr1         : in  std_logic_vector(31 downto 0);
        nr2         : in  std_logic_vector(31 downto 0);
        valid_input: in  std_logic;
        result      : out std_logic_vector(31 downto 0);
        valid_output: out std_logic
    );
end fp_adder_pipeline;

architecture pipeline of fp_adder_pipeline is

    constant EXPONENT_BITS : integer := 8;
    constant MANTISSA_BITS : integer := 23;

    subtype sign_type is std_logic;
    subtype exp_type is unsigned(EXPONENT_BITS-1 downto 0);
    subtype mantissa_type is unsigned(MANTISSA_BITS downto 0);
    subtype sum_type is unsigned(MANTISSA_BITS+2 downto 0);

    -- Semnale pentru etape
    signal stage1_sign_array : sign_type;
    signal stage1_exp_array  : exp_type;
    signal stage1_mant_array : mantissa_type;
    signal stage1_valid      : std_logic;

    signal stage2_larger_exp : exp_type;
    signal stage2_larger_mant: mantissa_type;
    signal stage2_smaller_mant: mantissa_type;
    signal stage2_exp_diff   : integer;
    signal stage2_valid      : std_logic;

```

```
    signal stage3_sign       : sign_type;
    signal stage3_sum        : sum_type;
    signal stage3_exp        : exp_type;
    signal stage3_valid      : std_logic;

    signal stage4_mant_norm  : mantissa_type;
    signal stage4_exp_norm   : exp_type;
    signal stage4_sign       : sign_type;
    signal stage4_valid      : std_logic;

begin

    -- Etapa 1: Extrage semnul, exponentul si mantisa
    process(clk, reset)
    begin
        if reset = '1' then
            stage1_valid <= '0';
        elsif rising_edge(clk) then
            stage1_sign_array <= nr1(31);
            stage1_exp_array <= unsigned(nr1(30 downto 23));
            stage1_mant_array <= "1" & unsigned(nr1(22 downto 0)); --
               Adaugarea bitului ascuns
            stage1_valid <= valid_input;
        end if;
    end process;

    -- Etapa 2: Alinierea mantiselor
    process(clk, reset)
    begin
        if reset = '1' then
            stage2_valid <= '0';
        elsif rising_edge(clk) then
            if stage1_exp_array >= unsigned(nr2(30 downto 23)) then
                stage2_larger_exp <= stage1_exp_array;
                stage2_larger_mant <= stage1_mant_array;
                stage2_smaller_mant <= "1" & unsigned(nr2(22 downto 0));
                stage2_exp_diff <= to_integer(stage1_exp_array -
                   unsigned(nr2(30 downto 23)));
            else
                stage2_larger_exp <= unsigned(nr2(30 downto 23));
                stage2_larger_mant <= "1" & unsigned(nr2(22 downto 0));
                stage2_smaller_mant <= stage1_mant_array;
                stage2_exp_diff <= to_integer(unsigned(nr2(30 downto 23)
                   ) - stage1_exp_array);
            end if;
            stage2_valid <= stage1_valid;
        end if;
    end process;

    -- Etapa 3: Adunare sau scadere mantise
    process(clk, reset)
    variable smaller_mant_shifted : mantissa_type;
    begin
        if reset = '1' then
```

```
100             stage3_valid <= '0';
101         elsif rising_edge(clk) then
102             if stage2_exp_diff > 0 then
103                 smaller_mant_shifted := shift_right(stage2_smaller_mant,
                        stage2_exp_diff);
104             else
105                 smaller_mant_shifted := stage2_smaller_mant;
106             end if;
107 
108             if stage1_sign_array = nr2(31) then
109                 -- Semne egale, adunare
110                 stage3_sum <= resize(stage2_larger_mant, MANTISSA_BITS
                        +3) + resize(smaller_mant_shifted, MANTISSA_BITS+3);
111                 stage3_sign <= stage1_sign_array;
112             else
113                 -- Semne diferite, scadere
114                 stage3_sum <= resize(stage2_larger_mant, MANTISSA_BITS
                        +3) - resize(smaller_mant_shifted, MANTISSA_BITS+3);
115                 -- Semnul se pastreaza de la mantisa mai mare
116                 stage3_sign <= stage2_larger_mant(MANTISSA_BITS);
117             end if;
118 
119             stage3_exp <= stage2_larger_exp;
120             stage3_valid <= stage2_valid;
121         end if;
122     end process;
123 
124     -- Etapa 4: Normalizare
125     process(clk, reset)
126     begin
127         if reset = '1' then
128             stage4_valid <= '0';
129         elsif rising_edge(clk) then
130             if stage3_sum(MANTISSA_BITS+2) = '1' then
131                 stage4_mant_norm <= stage3_sum(MANTISSA_BITS+1 downto 1)
                        ;  -- Mutare mantisa pentru normalizare
132                 stage4_exp_norm <= stage3_exp + 1;
133             else
134                 stage4_mant_norm <= stage3_sum(MANTISSA_BITS downto 0);
135                 stage4_exp_norm <= stage3_exp;
136             end if;
137 
138             stage4_sign <= stage3_sign;
139             stage4_valid <= stage3_valid;
140         end if;
141     end process;
142 
143     -- Rezultat final
144     process(clk, reset)
145     begin
146         if reset = '1' then
147             result <= (others => '0');
148             valid_output <= '0';
149         elsif rising_edge(clk) then
```

```
            if stage4_valid = '1' then
                result <= stage4_sign & std_logic_vector(stage4_exp_norm
                    ) &
                          std_logic_vector(stage4_mant_norm(
                              MANTISSA_BITS-1 downto 0));
                valid_output <= '1';
            else
                valid_output <= '0';
            end if;
        end if;
    end process;

end pipeline;
```

Codul corespunzător simulării este următorul:

```
----------------------------------------------------------------
----- Universitatea Tehnica din Cluj-Napoca
----- Facultatea de Automatica si Calculatoare
----- 2024-2025
----- Assignment Curs - testbench pentru adunarea in virgula mobila
----- Student: Cret Maria Magdalena
----- Grupa: 30233 Semigrupa 1
----- Fisier Testbench ------
----------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity fp_adder_pipeline_tb is
end fp_adder_pipeline_tb;

architecture testbench of fp_adder_pipeline_tb is

    -- semnale pentru instanta
    signal clk           : std_logic := '0';
    signal reset         : std_logic := '0';
    signal nr1           : std_logic_vector(31 downto 0) := (others =>
        '0');
    signal nr2           : std_logic_vector(31 downto 0) := (others =>
        '0');
    signal valid_input   : std_logic := '0';
    signal result        : std_logic_vector(31 downto 0);
    signal valid_output  : std_logic;

    -- semnale pentru verificare
    signal expected_result : std_logic_vector(31 downto 0);
    signal pass            : boolean := true;

    constant clk_period : time := 10 ns;

begin
```

```
36
37      -- Instantierea unitatii testate
38      uut: entity work.fp_adder_pipeline
39          port map (
40              clk => clk,
41              reset => reset,
42              nr1 => nr1,
43              nr2 => nr2,
44              valid_input => valid_input,
45              result => result,
46              valid_output => valid_output
47          );
48
49      -- Generare clock
50      clk_process: process
51      begin
52          while true loop
53              clk <= '0';
54              wait for clk_period / 2;
55              clk <= '1';
56              wait for clk_period / 2;
57          end loop;
58      end process;
59
60      -- Proces de testare
61      stimulus_process: process
62      begin
63          -- Reset
64          reset <= '1';
65          wait for 2 * clk_period;
66          reset <= '0';
67
68          -- Test 1: Adunare simpla
69          nr1 <= "01000001010000000000000000000000"; -- 12.0 -- HEXA: 0
                x41400000
70          nr2 <= "01000000101000000000000000000000"; -- 5.0  -- HEXA: 0
                x40A00000
71          expected_result <= "01000001100100000000000000000000"; -- 17.0
                -- HEXA: 0x43100000
72          valid_input <= '1';
73          wait for clk_period;
74          valid_input <= '0';
75
76          wait until valid_output = '1';
77          assert result = expected_result report "Test 1 failed" severity
                error;
78
79          -- Test 2: Adunare cu numere negative
80          nr1 <= "11000000101000000000000000000000"; -- -5.0 --  HEXA:
81          nr2 <= "01000000101000000000000000000000"; -- 5.0  -- HEXA:
82          expected_result <= "00000000000000000000000000000000"; -- 0.0
83          valid_input <= '1';
84          wait for clk_period;
85          valid_input <= '0';
```

```
86
87          wait until valid_output = '1';
88          assert result = expected_result report "Test 2 failed" severity
               error;
89
90          -- Test 3: Adunare cu overflow
91          nr1 <= "01111111000000000000000000000000"; -- Float Max
92          nr2 <= "00111111100000000000000000000000"; -- 1.0
93          expected_result <= "01111111000000000000000000000000"; -- Float
               Max (overflow)
94          valid_input <= '1';
95          wait for clk_period;
96          valid_input <= '0';
97
98          wait until valid_output = '1';
99          assert result = expected_result report "Test 3 failed" severity
               error;
100
101         -- Test 4: Adunare cu numere cu zecimale
102         nr1 <= "01000000100110011001100110011010"; -- 4.6
103         nr2 <= "01000000010011001100110011001101"; -- 3.2
104         expected_result <= "01000001000001100110011001100110"; -- 7.8
105         valid_input <= '1';
106         wait for clk_period;
107         valid_input <= '0';
108
109         wait until valid_output = '1';
110         assert result = expected_result report "Test 4 failed" severity
               error;
111
112         report "All tests passed" severity note;
113         wait;
114     end process;
115
116 end testbench;
```