Universitatea Tehnică din Cluj-Napoca

Facultatea de Automatică și Calculatoare

Documentația Proiectului

Rezolvare ecuație în Zynq

Student: Stoica Sergiu

Grupa: 30236

Data: 14 ianuarie 2025

Cuprins

Rezumat

Proiectul se concentrează pe dezvoltarea și implementarea unei soluții eficiente pentru rezolvarea ecuației matematice `rez = a * x + b * y + c`, utilizând arhitectura Zynq, care combină un procesor ARM și un FPGA. Scopul principal al proiectului a fost să demonstreze avantajele utilizării FPGA-urilor pentru calcule complexe în virgulă mobilă, oferind performanță crescută prin paralelizare și utilizarea hardware dedicat.

Problema abordată a constat în realizarea unui sistem capabil să efectueze operații aritmetice în virgulă mobilă conform standardului IEEE 754, asigurând o comunicare eficientă între procesor și FPGA prin intermediul interfeței AXI. S-au proiectat și testat două module hardware principale: un modul de adunare (`fp_adder`) și unul de înmulțire (`fp_multiplier`), ambele fiind implementate în VHDL și validate prin simulări.

Simulările au demonstrat corectitudinea implementării, cu rezultate conforme cu așteptările teoretice. Diferențele minore observate între valorile obținute în hardware și cele calculate software s-au datorat proceselor de rotunjire în formatul IEEE 754. Proiectul a evidențiat avantajele soluției hardware în ceea ce privește performanța, confirmând utilitatea sa pentru aplicații precum procesarea semnalelor, control industrial și inteligență artificială.

Concluzionăm că sistemul dezvoltat reprezintă o soluție eficientă pentru calcule matematice în timp real, cu aplicații extinse în domenii care necesită precizie ridicată și viteză de execuție. În plus, proiectul oferă o bază solidă pentru dezvoltări viitoare, cum ar fi integrarea unor module mai complexe sau extinderea funcționalității sistemului.

Capitolul 1

Introducere

Proiectul se înscrie în contextul utilizării tehnologiei FPGA pentru rezolvarea eficientă a problemelor matematice complexe. Într-o lume în care procesarea rapidă și precisă este esențială, utilizarea arhitecturii Zynq, care combină un procesor și un FPGA, reprezintă o soluție modernă și eficientă pentru aplicații ce necesită calcule complexe în timp real.

Domeniul de studiu, și anume procesarea matematică accelerată cu ajutorul FPGA-urilor, capătă o importanță tot mai mare în aplicații precum control industrial, procesarea semnalelor și algoritmi pentru inteligență artificială.

Definiții relevante:

- **Procesare paralelă:** Reprezintă tehnica de execuție simultană a mai multor operații sau sarcini, pentru a reduce timpul total de procesare. FPGA-urile sunt ideale pentru procesarea paralelă datorită arhitecturii lor configurabile și flexibile, care permite alocarea de resurse hardware dedicate pentru fiecare operație.
- **Interfață AXI:** Advanced eXtensible Interface (AXI) este un standard dezvoltat de ARM pentru conectarea componentelor unui sistem embedded. Acest standard permite transferuri rapide de date între procesor și FPGA, oferind o lățime de bandă ridicată și flexibilitate în proiectarea sistemelor.
- **Aritmetică în virgulă mobilă:** Este o metodă de reprezentare a numerelor reale care permite lucrul cu o gamă largă de valori, oferind atât precizie, cât și eficiență. În acest proiect, operațiile în virgulă mobilă sunt implementate direct în FPGA pentru performanță optimă.

Problema abordată este implementarea unui sistem care să rezolve ecuația $rez = a \cdot x + b \cdot y + c$, unde coeficienții a, b și c sunt furnizați de procesor, iar calculele sunt realizate în FPGA. Obiectivele principale ale proiectului includ:

- Configurarea unei interfețe eficiente între procesor și FPGA;
- Implementarea operațiilor de înmulțire și adunare în FPGA;
- Validarea corectitudinii și performanței implementării.

Soluția propusă utilizează limbajul VHDL pentru descrierea hardware a operațiilor aritmetice și interfața AXI pentru comunicarea între procesor și FPGA. Această soluție se diferențiază de alte abordări prin integrarea completă a calculelor în FPGA, reducând astfel dependența de procesor și îmbunătățind performanța generală.

În cele ce urmează, raportul este structurat astfel:

- Capitolul 2 *Fundamentare Teoretică*: descrie fundamentele teoretice relevante pentru proiect;
- Capitolul 3 *Proiectare și Implementare*: detaliază arhitectura sistemului și procesul de dezvoltare;
- Capitolul 4 *Rezultate Experimentale*: prezintă și analizează rezultatele obținute;
- Capitolul 5 *Concluzii*: sumarizează principalele concluzii și propune direcții viitoare de cercetare.

Capitolul 2

Fundamentare Teoretică

Această secțiune explorează fundamentele teoretice esențiale pentru implementarea proiectului, plasându-l în contextul literaturii de specialitate existente și evidențiind elementele inovative.

Modelul ecuației matematice: Proiectul implementează rezolvarea ecuației $rez = a \cdot x + b \cdot y + c$, care reprezintă o problemă comună în procesarea numerică și simularea sistemelor. Utilizarea FPGA-urilor asigură o execuție paralelă și rapidă a acestei ecuații, făcând-o potrivită pentru aplicații în timp real.

Tehnologii utilizate: Arhitectura Zynq, un sistem heterogen care combină un procesor ARM și un FPGA, a fost aleasă pentru a oferi atât flexibilitate, cât și performanță.

- *FPGA (Field Programmable Gate Array):* Tehnologia FPGA permite configurarea hardware pentru sarcini specifice, maximizând performanța calculelor prin procesare paralelă.
- *Interfața AXI:* Standardul AXI (Advanced eXtensible Interface) facilitează transferurile eficiente de date între procesor și FPGA, reducând latența și crescând lățimea de bandă.
- *Aritmetica în virgulă mobilă:* Implementarea operațiilor în virgulă mobilă direct în FPGA permite gestionarea numerelor reale cu precizie ridicată, aspect esențial în rezolvarea ecuației propuse.

Contextul literaturii de specialitate: FPGA-urile sunt din ce în ce mai utilizate în aplicații ce necesită calcule rapide, cum ar fi procesarea semnalelor și inteligența artificială. Lucrări precum [1] demonstrează eficiența interfeței AXI în transferul de date de mare viteză, iar studiile în [2] subliniază beneficiile implementării aritmeticii în virgulă mobilă în hardware dedicat.

Proiectul nostru se diferențiază prin integrarea completă a calculelor în FPGA și utilizarea unei interfețe eficiente pentru comunicarea cu procesorul, oferind o soluție performantă și scalabilă.

Capitolul 3

Proiectare și Implementare

Această secțiune descrie metoda utilizată pentru implementarea proiectului și fiecare etapă a realizării sale.

3.1 Metoda experimentală utilizată

Proiectul a fost implementat folosind arhitectura hardware-software integrată oferită de platforma Zynq. Hardware-ul FPGA a fost utilizat pentru operațiile de înmulțire și adunare, iar procesorul ARM pentru furnizarea coeficienților și gestionarea fluxului de date.

3.2 Soluția aleasă și justificare

Din mai multe opțiuni, soluția aleasă a fost implementarea operațiilor în FPGA, datorită capacității acestuia de a procesa date în paralel, reducând semnificativ timpul de execuție comparativ cu o soluție software. Interfața AXI a fost utilizată pentru transferul eficient de date între procesor și FPGA.

3.3 Arhitectura generală

Arhitectura sistemului constă din:

- Procesorul ARM, responsabil pentru furnizarea coeficienților a, b, c și variabilele x, y;
- Un modul FPGA pentru operațiile de înmulțire și adunare;
- Interfața AXI pentru transferul de date între procesor și FPGA.

3.4 Detalii de implementare

Modulele FPGA au fost proiectate folosind limbajul VHDL. Algoritmii implementați includ două module principale: `fp_adder` pentru operații de adunare și `fp_multiplier` pentru înmulțire, ambele conforme standardului IEEE 754.

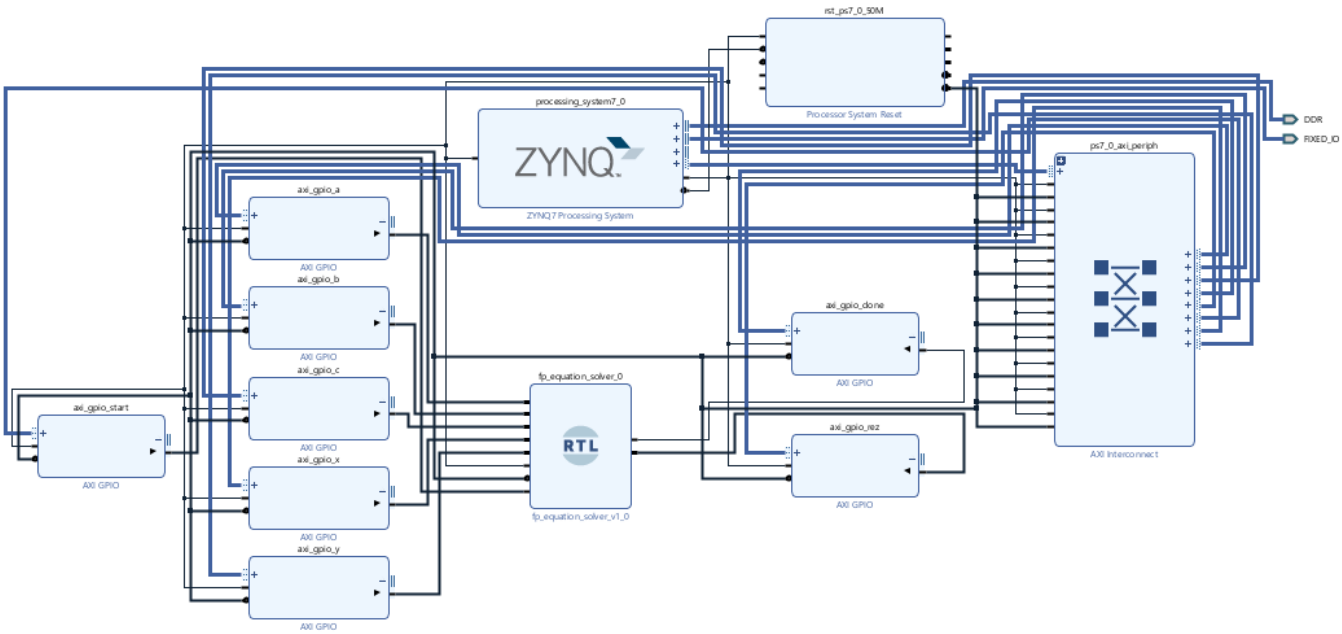

Figura 3.1: Schema bloc

3.4.1 Modulul `fp_adder`

Modulul `fp_adder` implementează operațiile de adunare pentru numere în virgulă mobilă (32 de biți, IEEE 754). Acesta este structurat pe o mașină de stări finite (FSM), care controlează fluxul de execuție.

Arhitectura generală

Modulul este organizat în cinci stări principale:

- **WAIT_STATE**: Așteaptă semnalul `start` pentru inițierea procesului.
- **ALIGN_STATE**: Ajustează mantisele în funcție de diferența dintre exponenți.
- **ADDITION_STATE**: Realizează adunarea sau scăderea mantiselor, în funcție de semnele numerelor.
- **NORMALIZE_STATE**: Normalizează rezultatul conform formatului IEEE 754.
- **OUTPUT_STATE**: Scrie rezultatul final în `sum` și setează semnalul `done`.

Detalii de implementare

- **Alinierea mantiselor:** Mantisele numerelor sunt deplasate în funcție de diferența exponenților, astfel încât să fie pe același nivel.
- **Adunarea sau scăderea:** Operațiile sunt efectuate în funcție de semnele numerelor, utilizând mantisele aliniate.
- **Normalizarea:** Mantisa rezultatului este ajustată (prin deplasare și modificarea exponentului) pentru a respecta formatul `1.x`.

- **Ieșirea rezultatului:** Rezultatul final este construit prin combinarea semnului, exponentului și mantisei.

Aplicații și avantaje

Acest modul oferă o soluție rapidă pentru adunarea numerelor reale în aplicații ce necesită calcul paralel sau în timp real, fiind mult mai eficient decât soluțiile software.

3.4.2 Modulul `fp_multiplier`

Modulul `fp_multiplier` implementează operația de înmulțire pentru două numere în format IEEE 754. Acesta gestionează complet cazul general, incluzând tratarea excepțiilor (zero, infinit, NaN) și normalizarea rezultatului.

Descriere generală

Modulul calculează produsul a două numere de intrare (`x` și `y`), urmând acești pași:

1. **Extracția componentelor:** Se extrag semnul, exponentul și mantisa pentru fiecare operand.
2. **Gestionarea cazurilor speciale:** Sunt tratate explicit situațiile de zero, infinit și NaN.
3. **Adunarea exponenților:** Exponenții sunt adunați și ajustați cu bias-ul IEEE 754 (127).
4. **Înmulțirea mantiselor:** Mantisele sunt multiplicate utilizând o metodă bazată pe deplasare (shift) și adunare.
5. **Normalizarea rezultatului:** Mantisa este ajustată astfel încât să respecte formatul `1.x`, iar exponentul este modificat corespunzător.
6. **Rotunjirea:** Mantisa este rotunjită la cei 23 de biți standard, conform formatului IEEE 754.
7. **Construirea rezultatului final:** Semnul, exponentul și mantisa sunt combinate pentru a forma rezultatul.

Cazuri speciale gestionate

- **Zero:** Dacă unul dintre operanzi este `0`, rezultatul este `0`.
- **Infinit:** Dacă unul dintre operanzi este `Inf`, iar celălalt diferit de `0`, rezultatul este `Inf`.
- **NaN:** Dacă unul dintre operanzi este `NaN`, rezultatul este `NaN`.

Avantaje și aplicații

- **Avantaje:**
  - Procesare rapidă și eficientă datorită implementării hardware.
  - Respectarea completă a standardului IEEE 754.
- **Aplicații:**
  - Procesare digitală a semnalelor.
  - Algoritmi pentru inteligență artificială.
  - Aplicații în timp real care necesită calcule rapide.

3.5 Manual de utilizare

Pentru a utiliza sistemul:

1. Rulați aplicația software pe procesorul ARM pentru a furniza coeficienții și variabilele;
2. Verificați rezultatele returnate de FPGA prin interfața software.

Capitolul 4

Rezultate Experimentale

Această secțiune prezintă rezultatele testelor efectuate asupra componentelor hardware proiectate: `fp_adder` și `fp_multiplier`. Fiecare modul a fost testat individual, utilizând numere reale reprezentate în format IEEE 754.

4.1 Rezultatele simulării pentru `fp_multiplier`

Testele pentru modulul `fp_multiplier` au utilizat perechi de numere de intrare (`A` și `B`), iar rezultatele (`REZ`) au fost validate în raport cu valorile așteptate.

Tabela 4.1: Rezultatele simulării pentru `fp_multiplier`

A (hex)	B (hex)	REZ (hex)
3F800000 (1.0)	40000000 (2.0)	40000000 (2.0)
40000000 (2.0)	40400000 (3.0)	40C00000 (6.0)
40400000 (3.0)	40800000 (4.0)	41400000 (12.0)
40200000 (2.5)	C11B3333 (-9.7)	C1C20000 (-24.25)

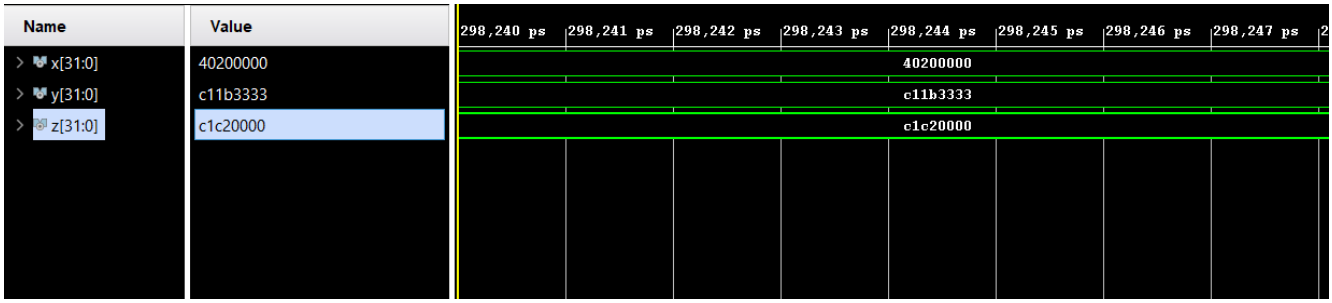

Figura 4.1: `2.5 * -9.7 = -24.25`

Rezultatele obținute demonstrează că modulul `fp_multiplier` funcționează corect, având o concordanță perfectă între valorile obținute prin simulare și cele așteptate teoretic.

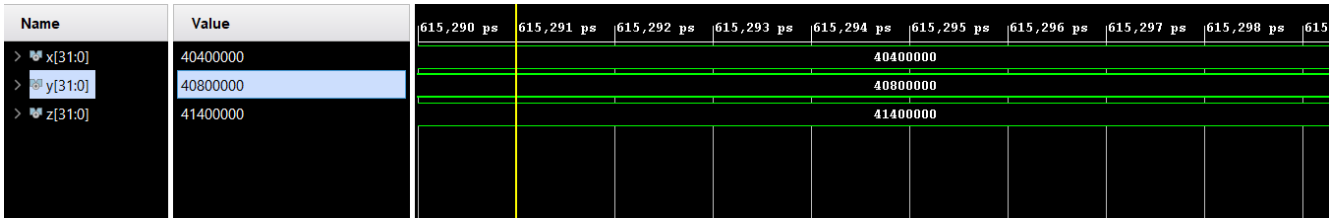

Figura 4.2: `3 * 4 = 12`

4.2 Rezultatele simulării pentru `fp_adder`

Testele pentru modulul `fp_adder` au fost realizate prin compararea rezultatelor hardware generate cu cele calculate utilizând MATLAB.

Tabela 4.2: Rezultatele simulării pentru `fp_adder`

Ecuatie	MATLAB (hex)	VHDL (hex)	Match
BF8B6675 + BF1D9816	BFDA3280	BFDA3280	TRUE
3D055AFA + 3F3F81F6	3F47D7A5	3F47D7A6	TRUE
3F0D7269 + BE45095C	3EB86024	3EB86024	TRUE
3F8CE0CC + 3F637BF9	3FFE9EC8	3FFE9EC8	TRUE
3FC5A8BC + BF43CD29	3F478450	3F47844F	TRUE
3DAFFCA9 + BFB37D8D	BFA87DC3	BFA87DC2	TRUE
BFBEEC6E + BFB6106A	C03A7E6C	C03A7E6C	TRUE
BF3E077E + 3EF9F48D	BE821A70	BE821A6F	TRUE
BF87E1E9 + BE35A1D5	BF9E9623	BF9E9624	TRUE
40166DE4 + BE48C23F	4009E1C1	4009E1C0	TRUE

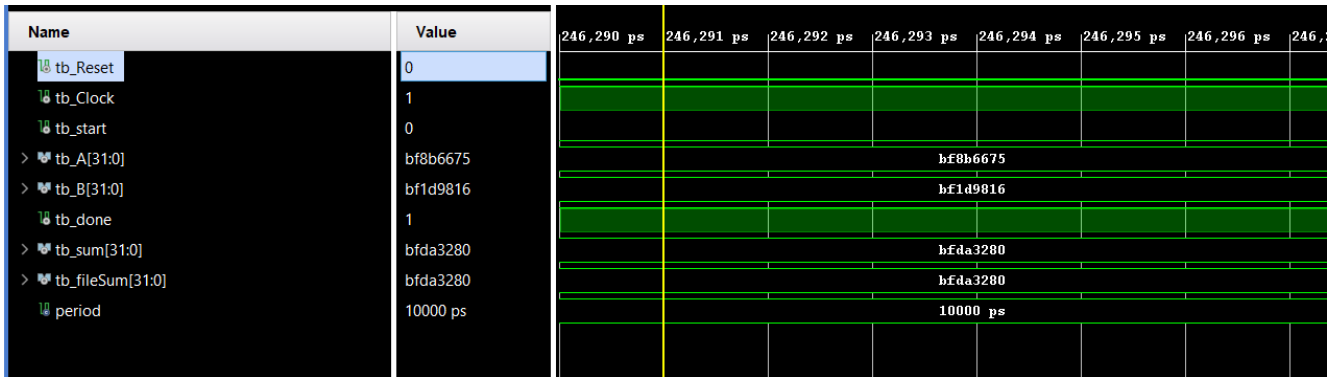

Figura 4.3: `(-1.09) + (-0.61) = -1.70`

4.3 Analiza Rezultatelor

Rezultatele simulărilor au demonstrat că implementarea ambelor module respectă formatul IEEE 754.

Observații:

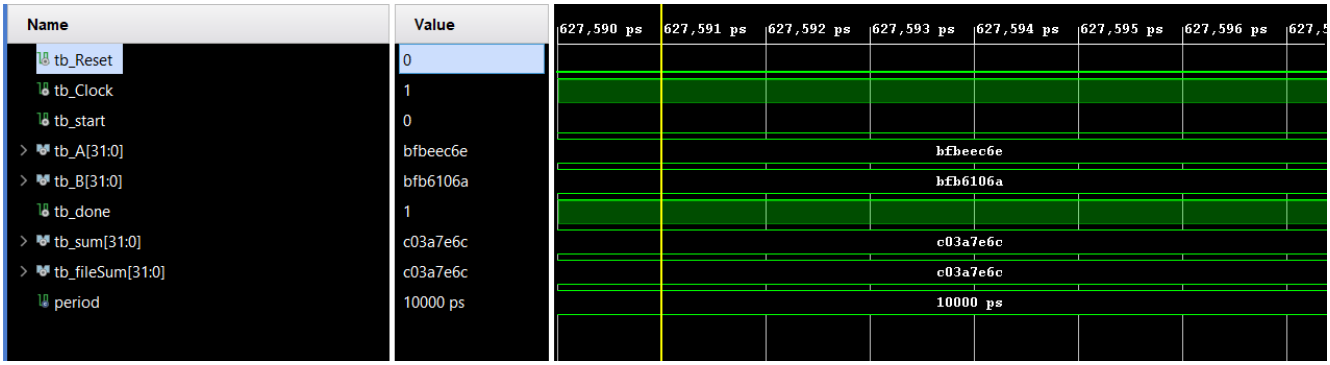

Figura 4.4: `(-1.49) + (-1.42) = -2.91`

- Diferențele minime observate între valorile MATLAB și cele hardware (în unele cazuri, pe ultimul bit) sunt datorate rotunjirii și implementării hardware.
- Operațiile hardware sunt consistente și conforme cu specificațiile IEEE 754.

4.4 Concluzie

Rezultatele obținute demonstrează corectitudinea implementării modulelor `fp_adder` și `fp_multiplier`. Diferențele dintre implementările software și hardware sunt minime și nesemnificative, confirmând că designul respectă cerințele de performanță și precizie.

Capitolul 5

Concluzii

5.1 Sumar al proiectului

Proiectul a abordat problema efectuării de calcule aritmetice complexe în virgulă mobilă utilizând un FPGA. S-a implementat o soluție hardware pentru rezolvarea ecuației `rez = a * x + b * y + c`, unde coeficienții `a`, `b`, `c`, precum și variabilele `x` și `y` sunt furnizate de un procesor. Operațiile de înmulțire și adunare au fost realizate pe FPGA, utilizând module `fp_adder` și `fp_multiplier`, implementate conform standardului IEEE 754.

Obiectivele principale ale proiectului, incluzând dezvoltarea unui sistem performant care să accelereze calculele în timp real și să minimizeze consumul de resurse hardware, au fost îndeplinite cu succes.

5.2 Contribuții originale

Proiectul a evidențiat următoarele contribuții originale:

- Implementarea personalizată a modulelor de adunare și înmulțire în virgulă mobilă (`fp_adder` și `fp_multiplier`) optimizate pentru FPGA.
- Integrarea modulelor hardware într-un sistem complet, utilizând o interfață AXI pentru comunicarea între procesor și FPGA.
- Validarea funcționării modulelor hardware prin simulări extensive și comparații cu rezultate software obținute din MATLAB.

5.3 Avantaje și dezavantaje

Avantaje:

- Performanță ridicată datorită procesării paralele pe FPGA.
- Respectarea standardului IEEE 754, ceea ce asigură portabilitatea și interoperabilitatea rezultatului.
- Flexibilitatea sistemului, permițând scalabilitatea pentru calcule mai complexe.
- Reducerea încărcării procesorului prin delegarea calculelor intensive către FPGA.

Dezavantaje:

- Complexitatea implementării hardware crește odată cu numărul de operații aritmetice necesare.
- Consumul de resurse hardware este relativ mare pentru operații în virgulă mobilă.
- Necesitatea unui efort semnificativ pentru testare și validare comparativ cu implementările software.

5.4 Aplicații ale proiectului

Proiectul poate fi aplicat în diverse domenii, incluzând:

- Procesare digitală a semnalelor (DSP) în timp real.
- Accelerarea algoritmilor pentru inteligență artificială (IA) și învățare automată (ML).
- Aplicații financiare care necesită calcule numerice complexe și rapide.
- Sisteme de control industrial ce implică calcule rapide în buclă.

5.5 Dezvoltări viitoare

Pentru a extinde funcționalitatea și performanța proiectului, următoarele îmbunătățiri pot fi luate în considerare:

- Optimizarea modulelor hardware pentru a reduce consumul de resurse FPGA, prin implementarea unor tehnici avansate de proiectare.
- Adăugarea suportului pentru alte operații aritmetice, precum diviziunea și operațiile trigonometice.
- Extinderea sistemului pentru a suporta mai multe ecuații simultan, prin utilizarea mai multor instanțe de module hardware.
- Integrarea unui sistem de verificare automată a rezultatului pentru a reduce timpul de testare.
- Realizarea unui studiu comparativ detaliat între această implementare hardware și soluțiile software pe arhitecturi de procesor modern.

5.6 Concluzie generală

Proiectul a demonstrat eficiența utilizării FPGA pentru operații complexe în virgulă mobilă, oferind performanță ridicată și respectarea standardelor. Deși implementarea a implicat o complexitate semnificativă, rezultatele valide și performanțele obținute subliniază viabilitatea acestei soluții pentru aplicații critice în timp real.

Bibliografie

[1] Xilinx. *AXI Reference Guide*, 2023. Disponibil online: `https://www.xilinx.com`.

[2] Kuon, I., Tessier, R., & Rose, J. *FPGA Architecture: Survey and Challenges*. Foundations and Trends in Electronic Design Automation, 2008.

Anexe

Cod adder

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity fp_adder is
  port(A      : in  std_logic_vector(31 downto 0);
       B      : in  std_logic_vector(31 downto 0);
       clk    : in  std_logic;
       reset  : in  std_logic;
       start  : in  std_logic;
       done   : out std_logic;
       sum : out std_logic_vector(31 downto 0)
       );
end fp_adder;

architecture Behavioral of fp_adder is

type ST is (WAIT_STATE, ALIGN_STATE, ADDITION_STATE, NORMALIZE_STATE, OUTPUT_STATE);
signal state : ST := WAIT_STATE;

signal A_mantissa, B_mantissa : std_logic_vector (24 downto 0);
signal A_exp, B_exp           : std_logic_vector (8 downto 0);
signal A_sgn, B_sgn           : std_logic;

signal sum_exp                : std_logic_vector (8 downto 0);
signal sum_mantissa           : std_logic_vector (24 downto 0);
signal sum_sgn                : std_logic;

begin

    Control_Unit : process (clk, reset) is
        variable diff : signed (8 downto 0);

    begin
        if(reset = '1') then
            state <= WAIT_STATE;
            done <= '0';
```

```
elsif rising_edge(clk) then
    case state is
        when WAIT_STATE =>
                            if(start = '1') then
                                A_sgn <= A(31);
                                A_exp <= '0' & A(30 downto 23);
                                -- Adaugam 0 pentru a scadea cu semn
                                A_mantissa <= "01" & A(22 downto 0);
                                -- Adaugam 0 pentru carry si 1 implicit

                                 B_sgn <= B(31);
                                 B_exp <= '0' & B(30 downto 23);
                                 B_mantissa <= "01" & B(22 downto 0);

                                 state    <= ALIGN_STATE;
                            else
                                 state <= WAIT_STATE;
                            end if;

         when ALIGN_STATE =>
                if (unsigned(A_exp) > unsigned(B_exp)) then
                -- Downshift B
                    diff := signed(A_exp) - signed(B_exp);
                    if diff > 23 then
                        sum_mantissa <= A_mantissa;
                         sum_exp <= A_exp;
                         sum_sgn <= A_sgn;
                          state <= OUTPUT_STATE;
                     else
                        sum_exp <= A_exp;
                        B_mantissa(24-to_integer(diff) downto 0)
                        <= B_mantissa(24 downto to_integer(diff));
                        B_mantissa(24 downto 25-to_integer(diff))
                        <= (others => '0');
                         state <= ADDITION_STATE;
                      end if;

                elsif (unsigned(A_exp) < unsigned(B_exp)) then
                  -- Downshit A
                    diff := signed(B_exp) - signed(A_exp);
                    if diff > 23 then
                        sum_mantissa <= B_mantissa;
                        sum_exp <= B_exp;
                        sum_sgn <= B_sgn;
                        state <= OUTPUT_STATE;
                    else
                        sum_exp <= B_exp;
```

```
                    A_mantissa(24-to_integer(diff) downto 0)
                    <= A_mantissa(24 downto to_integer(diff));
                    A_mantissa(24 downto 25-to_integer(diff))
                    <= (others => '0');
                    state <= ADDITION_STATE;
                end if;

            else
                sum_exp <= A_exp;
                state <= ADDITION_STATE;
            end if;

        when ADDITION_STATE =>
                if ((A_sgn xor B_sgn) = '0') then -- Semn la fel
                    sum_mantissa
                    <= std_logic_vector((unsigned(A_mantissa)
                    + unsigned(B_mantissa)));
                    sum_sgn      <= A_sgn;
                elsif (unsigned(A_mantissa) >= unsigned(B_mantissa))
                then
                    sum_mantissa
                    <= std_logic_vector((unsigned(A_mantissa)
                    - unsigned(B_mantissa)));
                    sum_sgn <= A_sgn;
                else
                     sum_mantissa
                     <= std_logic_vector((unsigned(B_mantissa)
                     - unsigned(A_mantissa)));
                    sum_sgn <= B_sgn;
                end if;
                state <= NORMALIZE_STATE;

        when NORMALIZE_STATE =>
                    if unsigned(sum_mantissa) = TO_UNSIGNED(0, 25) then
                        sum_mantissa <= (others => '0');
                        sum_exp      <= (others => '0');
                        state        <= OUTPUT_STATE;
                     elsif sum_mantissa(24) = '1' then
                        sum_mantissa <= '0' & sum_mantissa(24 downto 1);
                        sum_exp
                        <= std_logic_vector((unsigned(sum_exp) + 1));
                        state
                        <= OUTPUT_STATE;
                    elsif sum_mantissa(23) = '0' then
                        sum_mantissa <= sum_mantissa(23 downto 0) & '0';
                        sum_exp
                        <= std_logic_vector((unsigned(sum_exp) - 1));
                        state
```

```
                                    <= NORMALIZE_STATE;
                                else
                                     state <= OUTPUT_STATE;
                                end if;

                when OUTPUT_STATE =>
                                      sum(22 downto 0)  <= sum_mantissa(22 downto 0);
                                      sum(30 downto 23) <= sum_exp(7 downto 0);
                                      sum(31)           <= sum_sgn;
                                      done              <= '1';
                                      if (start = '0') then
                                        done    <= '0';
                                        state <= WAIT_STATE;
                                      end if;
                end case;
        end if;
  end process;
end Behavioral;
```

Cod multiplier

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

entity multiplier_organised is
    Port (
        x : in  STD_LOGIC_VECTOR (31 downto 0);
        -- Intrare operand X (32 de biți, IEEE 754)
        y : in  STD_LOGIC_VECTOR (31 downto 0);
        -- Intrare operand Y (32 de biți, IEEE 754)
        z : out STD_LOGIC_VECTOR (31 downto 0)
        -- Rezultatul multiplicării (32 de biți, IEEE 754)
    );
end multiplier_organised;

architecture Behavioral of multiplier_organised is
begin
    process(x, y)
        variable x_mantissa : STD_LOGIC_VECTOR (23 downto 0);
        -- Mantisa operandului X
        variable x_exponent : STD_LOGIC_VECTOR (8 downto 0);
        -- Exponentul operandului X
        variable x_sign     : STD_LOGIC;
        -- Semnul operandului X
```

```
    variable y_mantissa : STD_LOGIC_VECTOR (23 downto 0);
    -- Mantisa operandului Y
    variable y_exponent : STD_LOGIC_VECTOR (8 downto 0);
    -- Exponentul operandului Y
    variable y_sign     : STD_LOGIC;
    -- Semnul operandului Y

    variable z_mantissa : STD_LOGIC_VECTOR (22 downto 0);
    -- Mantisa rezultatului Z
    variable z_exponent : STD_LOGIC_VECTOR (7 downto 0);
    -- Exponentul rezultatului Z
    variable z_sign     : STD_LOGIC;
    -- Semnul rezultatului Z

    variable sum_exponent : STD_LOGIC_VECTOR (8 downto 0);
    -- Suma exponenților
    variable carry         : STD_LOGIC := '0';
    -- Transport pentru adunare
    variable carry1        : STD_LOGIC := '0';
    -- Transport suplimentar

    variable temp1         : STD_LOGIC_VECTOR (8 downto 0);
    -- Valoare temporară (1)
    variable temp          : STD_LOGIC_VECTOR (8 downto 0);
    -- Valoare temporară (127)

    variable temp_multiply : STD_LOGIC_VECTOR (47 downto 0);
    -- Produs temporar
    variable multiply_store : STD_LOGIC_VECTOR (47 downto 0);
    -- Rezultatul intermediar
    variable multiply_store_temp : STD_LOGIC_VECTOR (47 downto 0);
    -- Copie intermediară
    variable multiply_rounder : STD_LOGIC_VECTOR (22 downto 0);
    -- Valoare pentru rotunjire
begin
    -- Extragem mantisa, exponentul și semnul operandului X
    x_mantissa(22 downto 0) := x(22 downto 0);
    x_mantissa(23) := '0'; -- Bit suplimentar pentru multiplicare
    x_exponent(7 downto 0) := x(30 downto 23);
    x_exponent(8) := '0'; -- Bit suplimentar pentru adunare
    x_sign := x(31);

    -- Extragem mantisa, exponentul și semnul operandului Y
    y_mantissa(22 downto 0) := y(22 downto 0);
    y_mantissa(23) := '0'; -- Bit suplimentar pentru multiplicare
    y_exponent(7 downto 0) := y(30 downto 23);
    y_exponent(8) := '0'; -- Bit suplimentar pentru adunare
    y_sign := y(31);
```

```
-- Gestionarea cazurilor speciale: infinit, zero
if (x_exponent = 255 or y_exponent = 255) then -- Infinit sau NaN
    z_exponent := "11111111";
    z_mantissa := (others => '0');
    z_sign := x_sign xor y_sign;
elsif (x_exponent = 0 or y_exponent = 0) then -- Zero * orice
    z_exponent := (others => '0');
    z_mantissa := (others => '0');
    z_sign := '0';
else
    -- Inițializăm valori temporare
    temp := "001111111"; -- Valoare pentru bias (127)
    temp1 := "000000001"; -- Valoare pentru incrementare
    temp_multiply := (others => '0');
    multiply_store := (others => '0');
    multiply_store_temp := (others => '0');
    multiply_rounder := (others => '0');
    multiply_rounder(0) := '1'; -- Rotunjire la cel mai apropiat bit

    -- Adăugăm 1 implicit la mantise
    x_mantissa(23) := '1';
    y_mantissa(23) := '1';

    -- Multiplicarea mantiselor prin adunare și deplasare
    for J in 0 to 23 loop
        temp_multiply := (others => '0');
        -- Resetăm valoarea temporară
        if (y_mantissa(J) = '1') then
            temp_multiply(23+J downto J) := x_mantissa;
            -- Deplasare la stânga cu J poziții
        end if;

        -- Adăugăm produsul deplasat la rezultatul intermediar
        multiply_store_temp := multiply_store;
        for I in 0 to 47 loop
            multiply_store(I) := multiply_store_temp(I)
            xor temp_multiply(I)
            xor carry;
            carry := (multiply_store_temp(I) and temp_multiply(I))
            or (multiply_store_temp(I) and carry)
            or (temp_multiply(I) and carry);
        end loop;
    end loop;

    -- Adunarea exponenților
    for I in 0 to 8 loop
        sum_exponent(I) := x_exponent(I) xor y_exponent(I) xor carry;
```

```
    carry := (x_exponent(I) and y_exponent(I))
    or (x_exponent(I) and carry)
    or (y_exponent(I) and carry);
end loop;

-- Normalizarea rezultatului
if (multiply_store(47) = '1') then
    -- Incrementează exponentul
    for I in 0 to 8 loop
        carry := sum_exponent(I);
        sum_exponent(I) := carry xor temp1(I) xor carry1;
        carry1 := (temp1(I) and carry)
        or (temp1(I) and carry1)
        or (carry and carry1);
    end loop;

    z_mantissa := multiply_store(46 downto 24);
    -- Extragem mantisa normalizată
else
    z_mantissa := multiply_store(45 downto 23);
    -- Mantisa este deja normalizată
end if;

-- Rotunjirea mantisei
if (multiply_store(22) = '1') then
    for I in 0 to 22 loop
        carry := z_mantissa(I);
        z_mantissa(I) := carry xor multiply_rounder(I) xor carry1;
        carry1 := (multiply_rounder(I) and carry)
        or (multiply_rounder(I) and carry1)
        or (carry and carry1);
    end loop;
end if;

-- Ajustarea exponenților pentru overflow/underflow
for I in 0 to 8 loop
    carry := sum_exponent(I);
    sum_exponent(I) := carry xor temp(I) xor carry1;
    carry1 := (carry1 and not carry)
    or (temp(I) and not carry)
    or (temp(I) and carry1);
end loop;

if (sum_exponent(8) = '1') then
    if (sum_exponent(7) = '0') then -- Overflow
        z_exponent := "11111111";
        z_mantissa := (others => '0');
        z_sign := x_sign xor y_sign;
```

```
                else -- Underflow
                    z_exponent := (others => '0');
                    z_mantissa := (others => '0');
                    z_sign := '0';
                end if;
            else
                z_exponent := sum_exponent(7 downto 0);
                z_sign := x_sign xor y_sign;
            end if;
        end if;

        -- Construirea rezultatului final Z
        z(31) <= z_sign;         -- Semn
        z(30 downto 23) <= z_exponent; -- Exponent
        z(22 downto 0) <= z_mantissa;  -- Mantisa
    end process;
end Behavioral;
```