

Universitatea Tehnică din Cluj-Napoca
Facultatea de Automatică și Calculatoare
Secția Calculatoare
Anul 2024-2025
Semestrul II



Proiectarea Sistemelor Numerice

Proiect: Calculator de buzunar

Profesor coordonator: Dragoș Florin Lișman
Student: Pilug Marcela
Grupa: 30213

Cuprins

Cuprins.....	2
1.Enunțul problemei.....	3
2.Descrierea proiectului.....	4
3.Realizarea schemelor.....	6
4.Componentele proiectului.....	9
5.Justificarea soluției alese.....	13
6.Posibilități de dezvoltare ulterioară.....	14
7.Bibliografie.....	15

1. Enunțul problemei

A10) Să se proiecteze un **calculator de buzunar cu operații aritmetice fundamentale** (adunare, scădere, înmulțire, împărțire). Operațiile de înmulțire și împărțire se vor implementa folosind algoritmi specifici, nu operatorii limbajului. Operanzii sunt reprezentați pe 8 biți cu semn. Operanzii și operatorii vor fi introduși secvențial în formă zecimală. Se vor folosi afișajele cu 7 segmente de pe plăcuțele FPGA. Proiectul va fi realizat de **1 student**.

2.Descrierea proiectului

Pentru implementarea acestui calculator de buzunar a trebuit să îl construiesc mai întâi, în linii mari. Prin urmare, am început prin descrierea acestuia, iar după aceea, realizarea unor scheme.

Funcționarea generală este următoarea:

A) Introducem numerele (operand A și operand B):

- introducem cifrele numerelor una câte una(în baza 2) printr-un set de switch-uri (SW[7:0])

- la fiecare apăsare a butonului BTN0, cifra curentă se salvează și o adăugăm la numărul aflat în construcție

- odată ce primul număr (operand A) este complet, apăsăm BTN1, semnalizând că operandul este gata

- procesul se repetă pentru al doilea număr (operand B)

B) Selectăm operația:

- folosim 4 butoane pentru a selecta operația aritmetică pe care dorim să o aplicăm:

00 = adunare

01 = scădere

10 = înmulțire

11 = împărțire

C) Executăm operația și tratăm erorile dacă este cazul:

- pentru împărțire (OP = 11), dacă operandul B este 0, identificăm o eroare de împărțire la 0 și aprindem LED-ul roșu (ERR_LED). După aceea resetăm automat valorile celor doi operanzi

- pentru celelalte operații (adunare, scădere, înmulțire) verificăm dacă rezultatul produce overflow. Dacă da, aprindem ERR_LED și resetăm operanzii. Dacă nu, afișăm rezultatul final pe un display cu 7 segmente (3 cifre + semn).

D) Construim proiectul pe mai multe fișiere:

-realizăm module separate pentru fiecare componentă:

a) formare_nr = pentru formarea numărului din cifre introduse una după cealaltă

b) adder = pentru adunare

c) subtractor = pentru scădere

d) multiplier = pentru înmulțire, dar pe baza adunării și a shiftării

e) division = pentru împărțire, dar pe baza scăderii și a shiftării

f) operation_selector = pentru alegerea operației care va avea loc

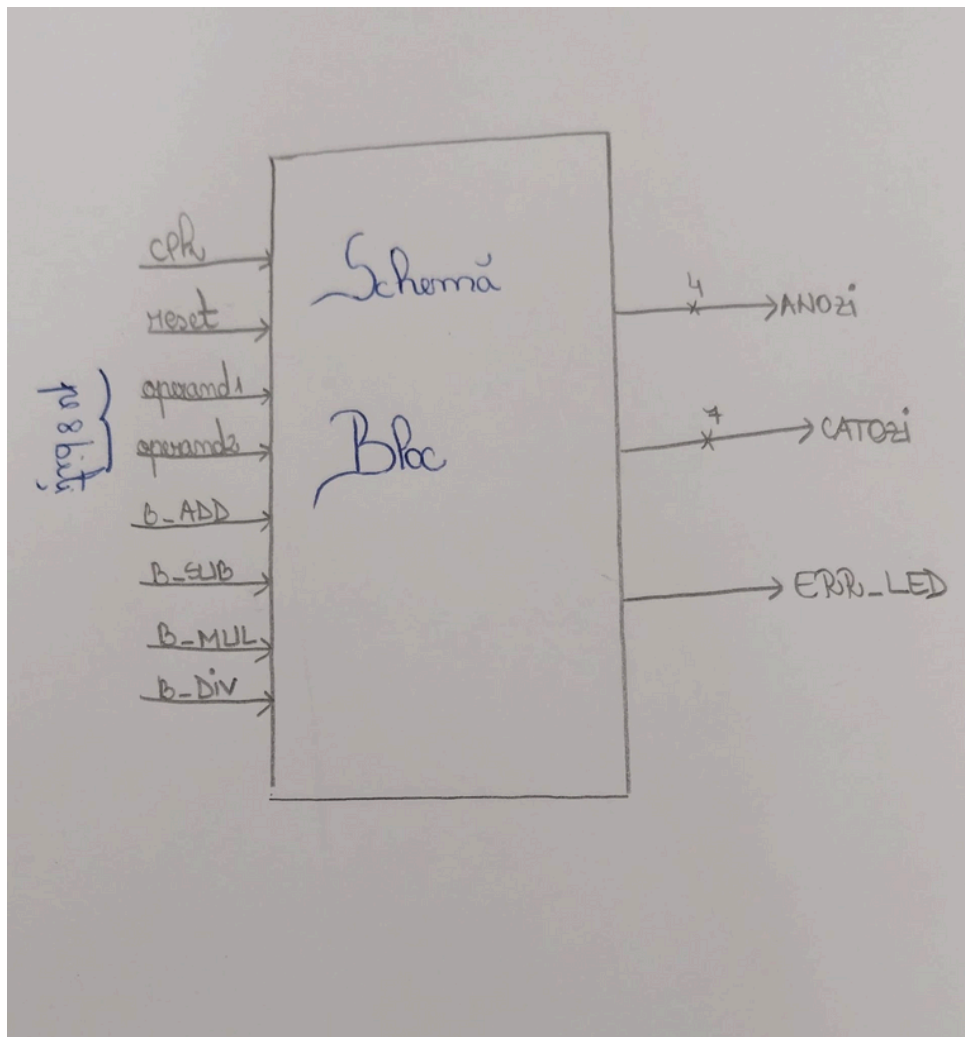
g) BCD_to_7segm = pentru conversia rezultatului din baza 10 în cele 7 segmente

h) afisare_display = pentru afișarea pe 7 segmente a rezultatului final

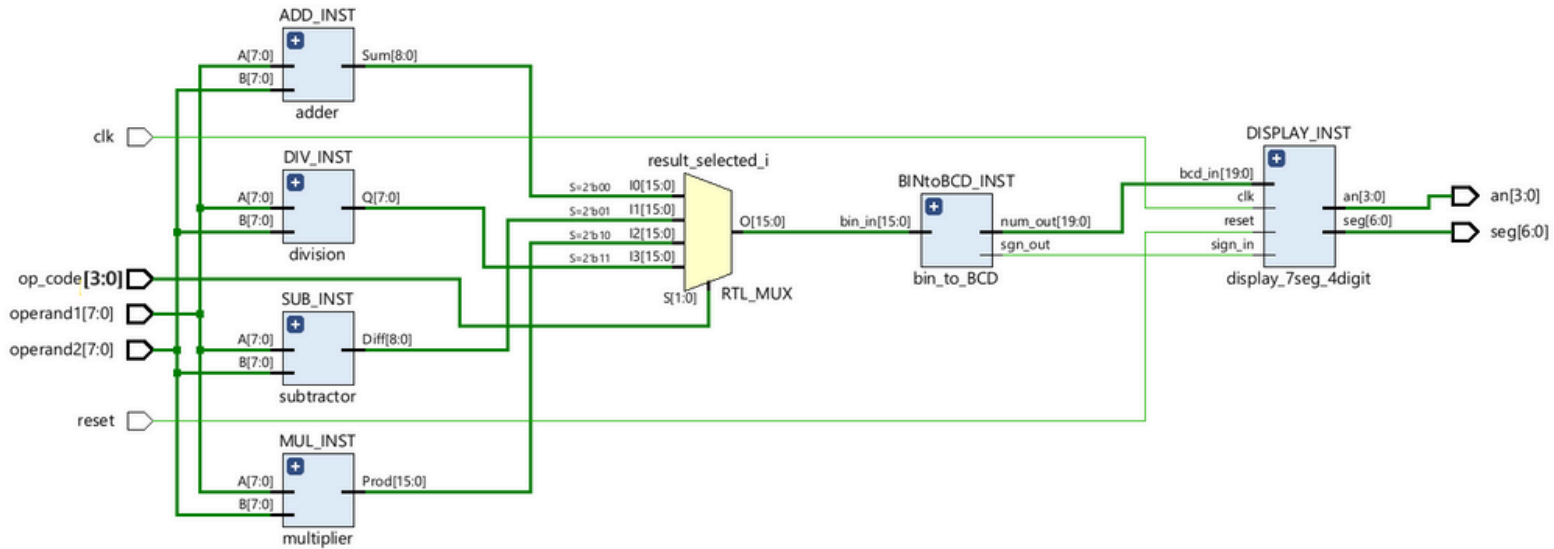
i) top_module = fișierul principal unde “apelăm” fiecare modul în parte pentru controlul și gestionarea datelor

3. Realizarea schemelor

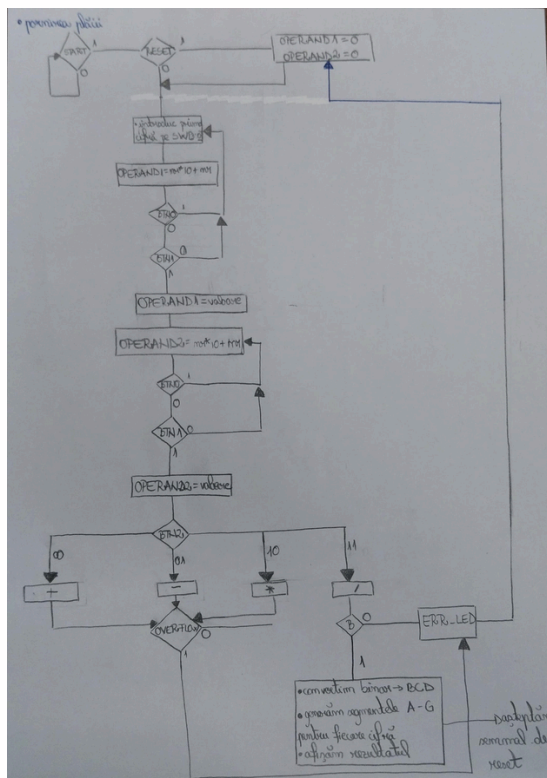
A) Schema bloc



B) Schema cu unitatea de comandă și execuție



C) Schema organigramei

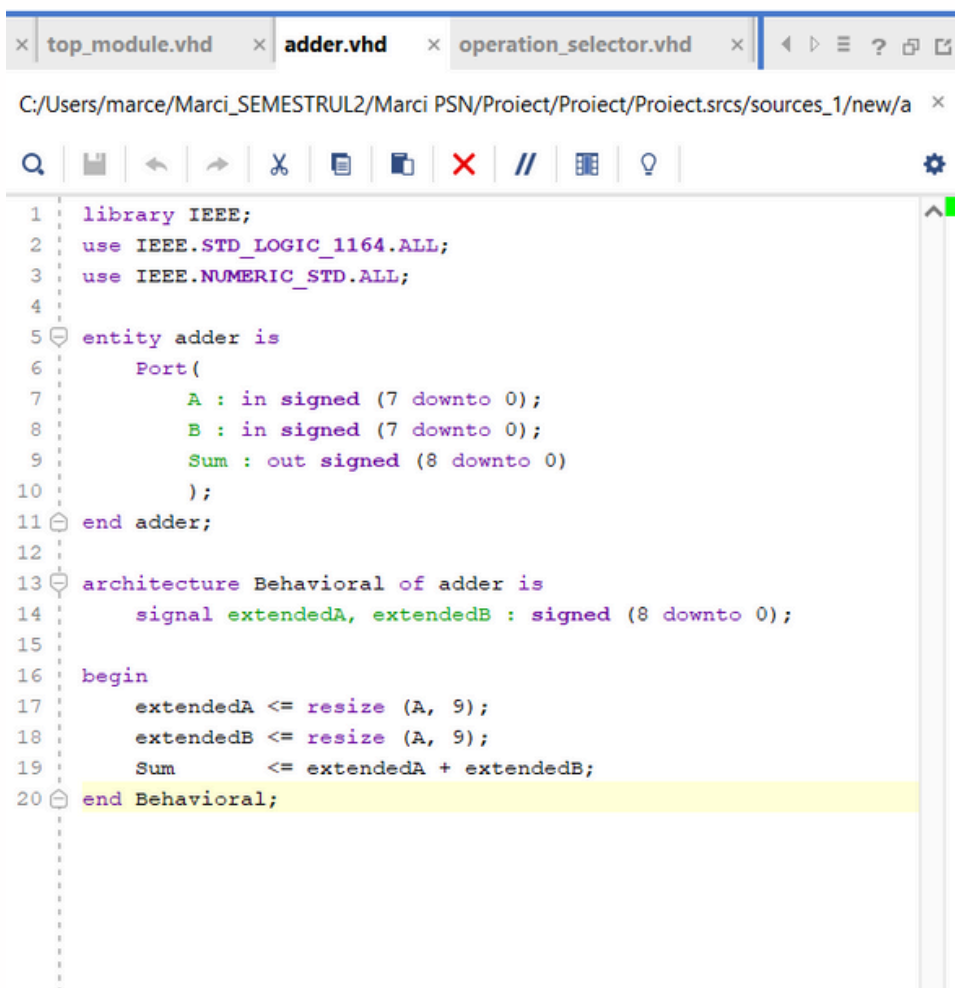


4. Componentele proiectului

Am creat mai multe module VHDL. Acestea rezolvă fiecare în parte câte o problemă mică din proiect, iar în final le-am pus împreună într-un modul mai mare. În cele ce urmează voi prezenta pe scurt fiecare componentă.

4.1. Adder

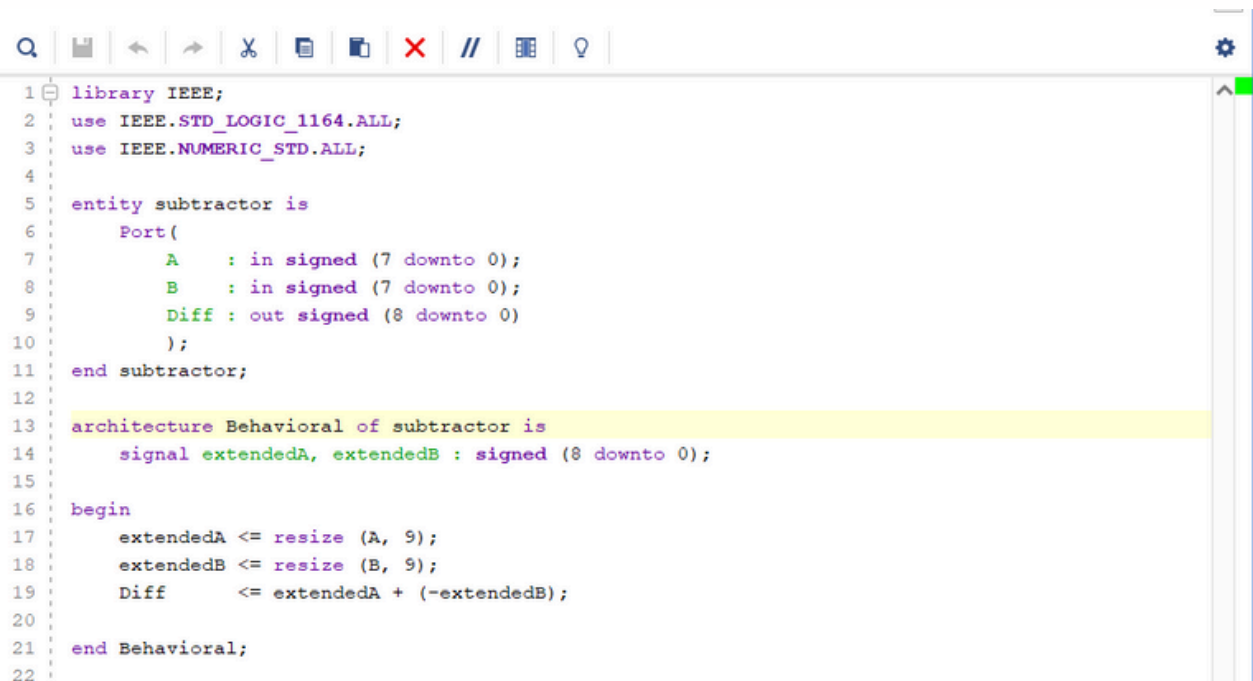
- un sumator care adună două numere pe 8 biți semnați (A și B) și produce un rezultat pe 9 biți (Sum) pentru a evita depășirea. Semnalele A și B sunt extinse la 9 biți folosind funcția `resize`.



```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity adder is
6      Port(
7          A : in signed (7 downto 0);
8          B : in signed (7 downto 0);
9          Sum : out signed (8 downto 0)
10     );
11 end adder;
12
13 architecture Behavioral of adder is
14     signal extendedA, extendedB : signed (8 downto 0);
15
16     begin
17         extendedA <= resize (A, 9);
18         extendedB <= resize (A, 9);
19         Sum <= extendedA + extendedB;
20 end Behavioral;
```


4.2. Subtractor

- am creat un modul care face scăderea a două numere întregi pe 8 biți cu semn. Am extins ambele numere la 9 biți ca să pot păstra semnul și eventualul overflow. Apoi, am făcut scăderea prin adunarea lui A cu minus B. La final, rezultatul îl trimit pe ieșirea Diff.



```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity subtractor is
6     Port(
7         A      : in signed (7 downto 0);
8         B      : in signed (7 downto 0);
9         Diff   : out signed (8 downto 0)
10    );
11 end subtractor;
12
13 architecture Behavioral of subtractor is
14     signal extendedA, extendedB : signed (8 downto 0);
15
16 begin
17     extendedA <= resize (A, 9);
18     extendedB <= resize (B, 9);
19     Diff      <= extendedA + (-extendedB);
20
21 end Behavioral;
```

4.3. Multiplier

- am făcut un multiplicator fără să folosesc operatorul *, doar cu shift și adunări. Am calculat semnul rezultatului cu XOR pe bitul de semn al lui A și B. Apoi am luat valorile absolute și am adunat pe rând shiftări ale lui A, acolo unde B are biți de 1. La final, dacă semnul rezultatului e negativ, am pus minus în față.

```

r.vhd x operation_selector.vhd x formare_nr.vhd x BIN_TO_BCD.vhd x af
sers/marce/Marci_SEMESTRUL2/Marci PSN/Proiect/Proiect/Proiect.srcs/sources_1/new/multiplie

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

entity multiplier is
    Port(
        A : in signed (7 downto 0);
        B : in signed (7 downto 0);
        Prod : out signed (15 downto 0)
    );
end multiplier;

architecture Behavioral of multiplier is
begin
    process (A, B)
        variable A_abs : signed (15 downto 0);
        variable B_abs : signed (7 downto 0);
        variable rez_temp : signed (15 downto 0);
        variable sign_rez : std_logic;
    begin
        sign_rez := A(7) xor B(7); --determinam semnul rezultatului
        if A(7) = '1' then
            A_abs := resize (~A, 16);
        else
            A_abs := resize (A, 16);
        end if;

        if B(7) = '1' then
            B_abs := -B;
        else
            B_abs := B;
        end if;

        rez_temp := (others => '0');

        for i in 0 to 7 loop
            if B_abs(i) = '1' then
                rez_temp := rez_temp + shift_left(A_abs, i);
            end if;
        end loop;

        if sign_rez = '1' then
            Prod <= -rez_temp;
        else
            Prod <= rez_temp;
        end if;
    end process;
end Behavioral;

```

4.4. Division

- am făcut un modul care împarte două numere pe 8 biți cu semn, fără să folosesc operatorul /. Mai întâi, am determinat semnele pentru cât și rest. Apoi am luat valorile absolute și le-am extins la 9 biți. Am folosit un algoritm simplificat de împărțire cu scăderi repetate și shiftări. La final, am pus semnul înapoi pe rezultat.

4.5. Convertor din binar în BCD

- am făcut un modul care transformă un număr binar semnat pe 16 biți în format zecimal (BCD) pe 5 cifre. Mai întâi, convertesc binarul la întreg și verific semnul, pe care îl trimit separat. Dacă numărul e negativ, îl fac pozitiv. Apoi calculez fiecare cifră zecimală (zeci de mii, mii, sute, zeci, unități) prin împărțiri și modulo. La final, pun fiecare cifră în poziția ei din ieșirea num_out, ca să poată fi afișată corect pe 7 segmente.

4.6. Selectarea operației care va avea loc

- am făcut un modul care alege operația dorită (adunare, scădere, înmulțire sau împărțire) între două numere semnate pe 8 biți, în funcție de codul de control OP. Am instanțiat patru componente (adder, subtractor, multiplier și division) și le-am conectat la semnalele de intrare. Într-un proces, aleg operația în funcție de OP și trimit rezultatul în ieșirea result. Așadar, fiecare operație are ieșirea dimensionată potrivit valorilor maxime posibile, iar în modulul operation_selector am redimensionat (resize) rezultatele ca să le pun toate într-un semnal comun de 16 biți, pentru afișare și control unificat.

4.7. Formarea operanzilor

- acest cod formează un număr de 8 biți semnat, primind câte un bit pe rând la fiecare impuls de ceas. Eu folosesc semnalul load_bit pentru a ști când să preiau bitul nou (bit_in) și îl inserez printr-un shift stânga în num_temp. Contorul cnt numără câți biți au fost primiți. Când s-au primit toți cei 8 biți, semnalul ready devine '1', semnalând că numărul este complet. La resetare, totul se golește și pornește de la zero.

4.8. Afișarea rezultatului pe placă

- acest cod controlează un afișaj cu 4 cifre pe 7 segmente, afișând un număr BCD și semnul lui. Eu folosesc un contor de „refresh” pentru a schimba rapid între cifrele afișate, astfel încât toate să pară aprinse simultan. Din bcd_in aleg cifra potrivită în funcție de care digit e activ și trimit modelul corect la seg. Dacă cifra e a patra și numărul e negativ (sign_in = '1'), afișez semnul minus. Cifrele sunt aprinse pe rând prin semnalul an, care activează doar un digit la un moment dat.

4.9. Modulul principal al calculatorului

- acest cod e partea de top a calculatorului meu digital, adică face legătura între toate modulele individuale. Primește doi operanzi, un cod de operație și, în funcție de el, afișează rezultatul dintre adunare, scădere, înmulțire sau împărțire. Apoi convertește rezultatul binar semnat în BCD ca să-l poată afișa. Folosesc modulul bin_to_BCD pentru conversie și display_7seg_4digit pentru afișare pe 4 cifre cu semn. Totul e coordonat de semnalul de ceas și de reset, ca să meargă sincron. Practic, pun cap la cap toate componentele calculatorului și trimit rezultatul pe display.

5. Justificarea soluției alese

Am ales să introduc operanzii secvențial, pe cifre binare, folosind butoanele BTN0 pentru salvarea fiecărei cifre și BTN1 pentru a marca finalizarea introducerii fiecărui număr. Această metodă permite o intrare clară și controlată, adaptată interfeței simple de pe FPGA.

Pentru selecția operației, am folosit butoanele BTN2 și BTN3, oferind o modalitate rapidă și intuitivă de a alege între adunare, scădere, înmulțire și împărțire.

Operațiile de înmulțire și împărțire sunt implementate cu algoritmi specifici, nu folosind operatorii nativi, pentru a respecta cerințele proiectului și pentru a înțelege în profunzime funcționarea acestor operații la nivel hardware.

Am prevăzut verificări pentru cazuri speciale: împărțirea la zero declanșează aprinderea unui LED roșu și resetarea operanzilor, iar depășirea intervalului (overflow) la celelalte operații conduce la resetarea operanzilor pentru a preveni afișarea unui rezultat incorrect. Structura proiectului este modulară, cu componente separate pentru adunare, scădere, înmulțire și împărțire, ceea ce facilitează testarea și reutilizarea modulelor. De exemplu, modulul adder este reutilizat în implementarea înmulțirii, iar subtractorul în împărțire, pentru eficiență și claritate.

6. Posibilități de dezvoltare ulterioare

- Extinderea operandului la 16 sau 32 biți pentru numere mai mari.
- Adăugarea unor operații noi: modul, ridicare la putere, operații logice.
- Implementarea conversiei dintr-o bază în alta.
- Optimizarea algoritmilor de înmulțire și împărțire.
- Adăugarea unui control FSM pentru gestionarea stărilor calculatorului.
- Extinderea afișajului cu mai multe cifre și implementarea unui sistem de intrare mai flexibil (reset, ștergere).

7. Bibliografie

"VHDL: Programming by Example" – Douglas L. Perry

Cursurile de la facultate

Teoria de la laboratoare

Articole online și tutoriale despre algoritmi aritmetici fără operatori standard