Unitate aritmetico-logică (UAL)

Student: Ilieșiu Robert-Mircea

Structura sistemelor de calcul

Universitatea Tehnică din Cluj-Napoca

Conținut

Introducere

1.1 Context

Scopul acestui proiect este de a realiza o unitate aritmetico-logică care să efectuieze diferite operații aritmetice cum ar fi: adunare, scădere în complement față de 2, incrementarea și decrementerea unui număr, realizarea de operatii logice (AND, OR, NOT), rotire la stânga sau la dreapta a unui număr,negarea unui număr, înmulțire și împărțire.

1.2 Obiective

Unitatea va fi descrisă în VHDL și va fi inclusă într-un proiect Xilinx Vivado. O unitate de testbench va fi de asemenea creată pentru simularea unității noaste aritmetico-logice (UAL). Vor fi create și câteva componente adiționale cum ar fi: o unitate de afișare pe 7 segmente (seven-segment display unit) pentru afișarea rezultatelor operaților, un registru acumulator pentru un operand de intrare și rezultat, circuite suplimentare pentru realizarea operaților aritmetice mai complexe (înmulțire și împărțire).

Operațile efectuate de unitatea aritmetico-logică (UAL) vor fii cele care urmează:

- adunare
- scădere în complement față de 2
- operații logice (AND, OR, NOT)
- operația de negare a unui număr
- rotire la stânga a unui număr
- rotire la dreapta a unui număr
- operații aritmetice complexe (înmulțire și împărțire)

Studiu bibliografic

2.1 Ce este o unitate aritmetico-logică (UAL) ?

O unitate aritmetico-logică (UAL) face parte din unitatea centrală de execuție (CPU) și realizează operații aritmetice și logice pe instrucțiunile primite de la calculator. Putem spune că o unitate aritmetico-logică este nucleul unde se desfășoara majoritatea calculelor efectuate de calculator.

În anumite procesoare, unitatea aritmetico-logică (UAL) este împărțită în 2 unități: o unitate aritmetică (AU) și o unitate logică (LU).

Acesta contribuie la procesarea informației prin executarea funcților sale principale: operații aritmetice (adunare, scadere, împărțire, înmulțire, incrementare, decrementare), operatii logice (AND, OR, NOT) și comparații (<, >, =<, =>).

2.2 Care este rolul unității aritmetico-logice în CPU ?

ALU primește datele de intrare din registrul procesorului și din memoria RAM. Operațiile efectuate sunt controlate de **unitatea de control** a procesorului, care dictează ce instrucțiuni urmează să fie executate. Rezultatele generate de ALU pot fi fie stocate în registri, fie trimise înapoi către memoria principală pentru utilizări ulterioare.

2.3 Cum implementăm operațile aritmetice?

Baza operaților aritmetice o reprezintă operația de adunare. Cu ajutorul operației de adunare putem implementa cu succes operațile de:

- scădere (implementăm operația de scădere în complement față de 2 și astfel vom folosii operația de adunare)
- înmulțire (cu ajutorul unui registru acumulator și a operației de adunare vom implementa înmulțirea ca fiind o adunare repetată)
- împărțire (pentru împărțire vom folosi un algoritm care restabilește parțial rezultatul)

Adunare

Pentru implementarea operației de adunare ne vom folosi de tabelul de adevăr din figura 2.3.1 după care vom realiza o minimizare pe acest tabel pentru a obține ecuațile necessare pentru a calcula variabila **Sum** și **Carry Out** (figura 2.3.2).

Input			Output	
A	B	Cin	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

figura 2.3.1

După minimizare

$$\Sigma = (A \oplus B) \oplus C_{in}$$

$$C_{out} = AB + (A \oplus B)\, C_{in}$$

figura 2.3.2

Acum după de am obținut ecuațile pentru Sum și Carry Out putem implementa un sumator pe un bit pe care îl vom cascada pentru a putea realiza calcule cu numere pe mai mulți biți.

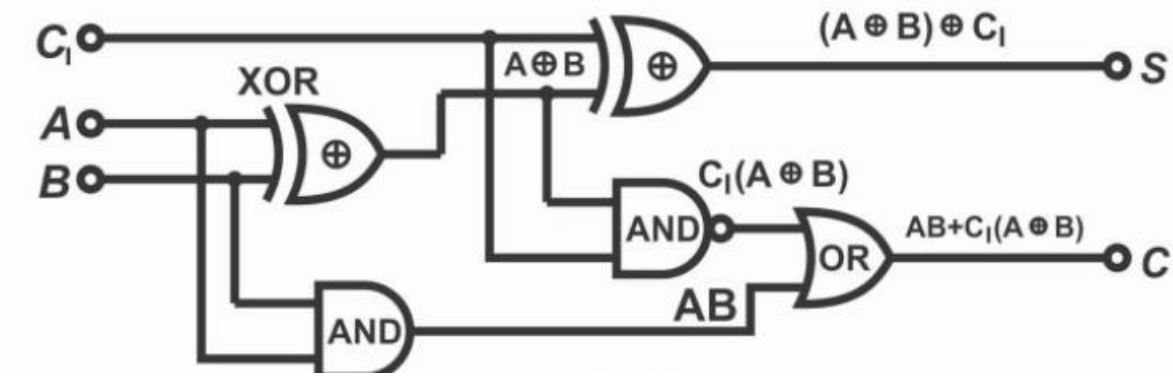


figura 2.3.3 (Sumator pe 1 bit)

Scădere (în complement față de 2)

După cum am menționat și mai sus, adunarea este o operație de bază pentru implementarea al altor operați printre care se numără și scăderea. Astfel vom implementa scăderea în complement față de 2 după următoare ecuație:

$$A - B = A + (-B) = A + not(B) + 1$$

(-B) = numărul binar B în complement față de 1 (pentru obținerea complementului față de 1 se vor nega toți biții numărului iar astfel se va obține complementul față de 1)

Înmulțire

Pentru implementarea înmulțirii ne vom folosi de un algoritm (Multiplication algorithm) pentru care este necesar să creem câteva componente în puls.

Componentele necesare sunt: un registru acumulator, un sumator și de o unitate care va deplasa numerele la stânga și la dreapta cu o poziție.

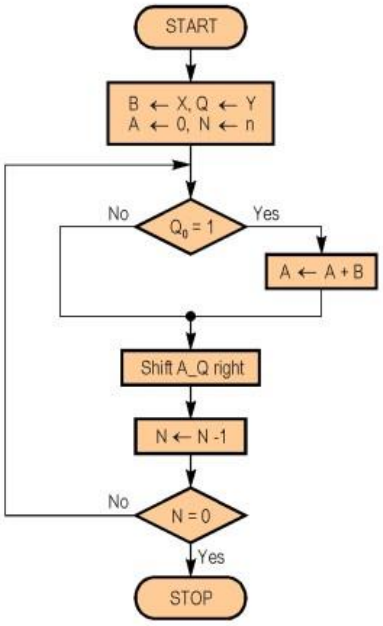


figura 2.3.4 (Multiplication algorithm)

Algoritmul constă în:

- scriem în regiștrii operanzi (B = 0, Q = 0) și inițializăm registrul accumulator (A = 0)
- complementăm numerele negative
- dacă Q0 = 1 atunci A = A + B
- shift numărul A la dreapta
- shift numărul Q la dreapta
- N = N – 1
- dacă N = 0 atunci sari la pasul 3 altfel am ajuns la sfârșit

Împărțire

Pentru împărțre vom folosi la fel ca la înmulțire de un algoritm (figura 2.3.6).

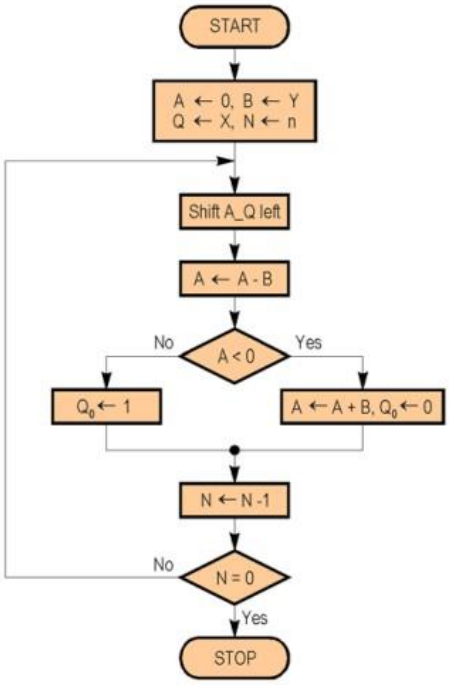


figura 2.3.5 (Division Algorithm)

Algoritmul consta în:

- se încarcă operanzi A, Q și B
- Shift Q left și Shift A left
- dacă A<0 atunci A = A + B, Q0 = 0 altfel Q0 = 1
- N = N -1
- dacă N = 0 atunci am ajuns la sfărșit altfel sari la pasul 2

2.4 Cum implementăm operațile logice?

Toate operațile logice (AND, OR, XOR, NOT) se bazează pe un tabel de adevăr reprezantat pe 1 bit. De asemenea aceste operații sunt implementate în VHDL pe baza tabelelor de adevăr corespondente lor.

Analiză și design

3.1 Propunerea de proiect

Proiectul final va cuprinde următoarele caracteristici descrise mai jos:

1. Implementarea operaților de: adunare, scădere, incrementare, decrementare, ȘI, SAU, NU logic, negare, rotație stânga și dreapta, înmulțire și împărțire
2. Implementarea unui registru acumulator pentru operanzi de intrare și rezultat
3. Implementarea de multiplexoare pentru selectarea operaților în funcție de codul operației
4. Implementarea unei unități de control care gestionează execuția instrucțiunilor

3.2 Analiza și design-ul proiectului

3.2.1 Soluția pentru operația de adunare

După cum spuneam și la punctul 2, operația de adunare o vom implementa minimizând tabelul de adevăr din figura 2.3.1 și obținund astfel operațile din figura 2.3.2.

Având operațile vom implementa un sumator pe 1 bit cu ajutorul căruia putem implementa un sumator pe 4 biți (4 BIT FULL ADDER) cascadând 4 sumatoare pe 1 bit. Astfel putem implementa operația de adunare pe n biți folosindune de schema din figura 3.2.1.

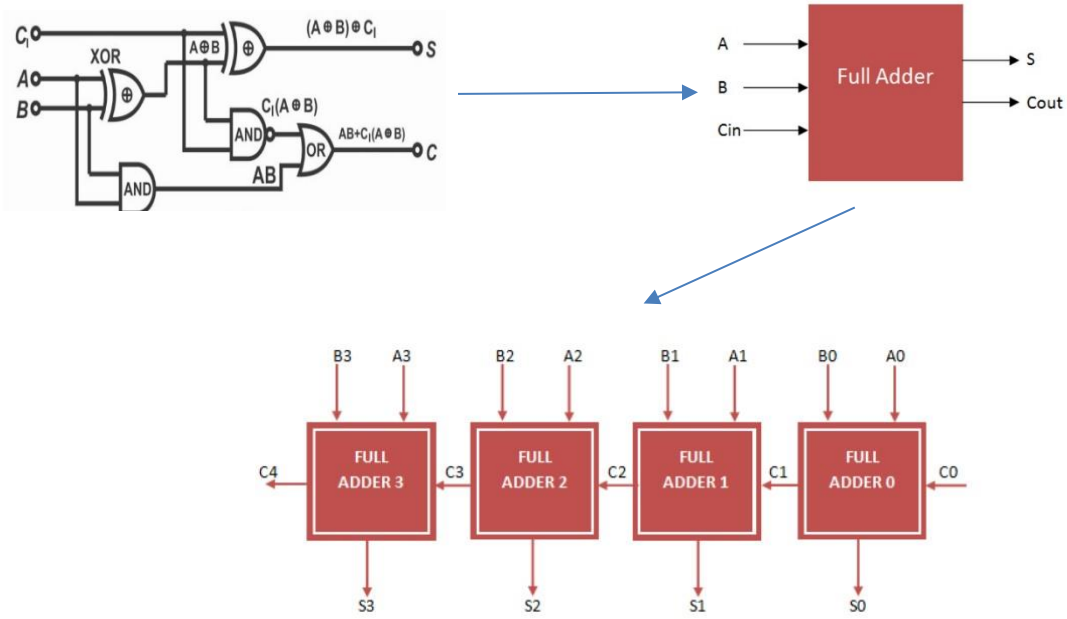


figura 3.2.1 (sumator pe 4 biți)

3.2.2 Soluția pentru operația de scădere

Pentru a implementa operația de cădere vom folosi scadărea în complement față de 2. Scăderea în complement față de 2 o implementăm aflând complementul față de 1 a operandului B(negăm toți biți lui B și obținem complementul față de 1) după care adunăm un 1 la operația de adunare dintre A și B.

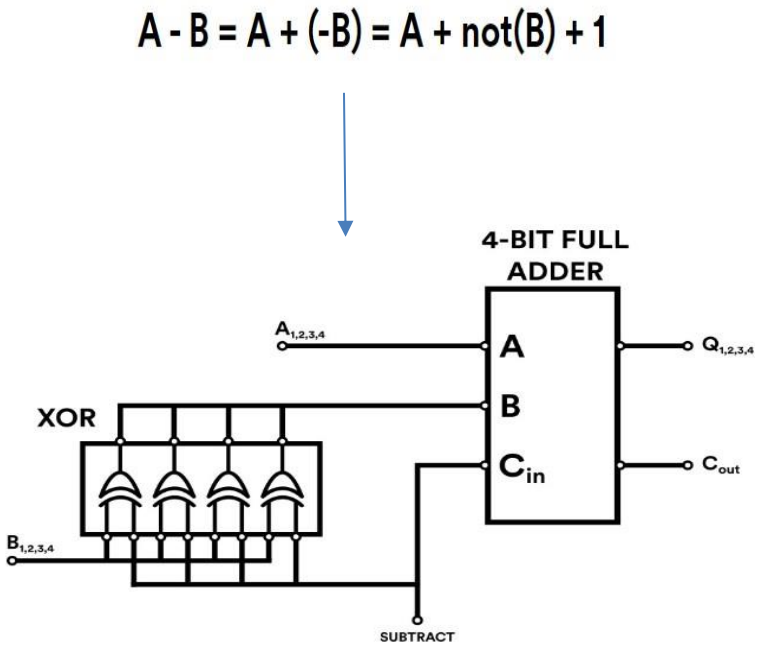


figura 3.2.2 (scăzător pe 4 biți)

3.2.3 Soluția pentru operațile de incrementare și decrementare

Pentru a implementa operațile de incrementare și decrementare ne vom folosi de sumatorul și scăzătorul implementate anterior pentru operațile de adunare și scădere.

3.2.4 Soluția pentru operație logice NU, ȘI, SAU, XOR

Pentru implementare operaților logice, nu trebuie să creeăm componente noi deoarece aceste operații sunt deja implementate în VHDL pe baza tabelelor de adevăr din figura de mai jos.

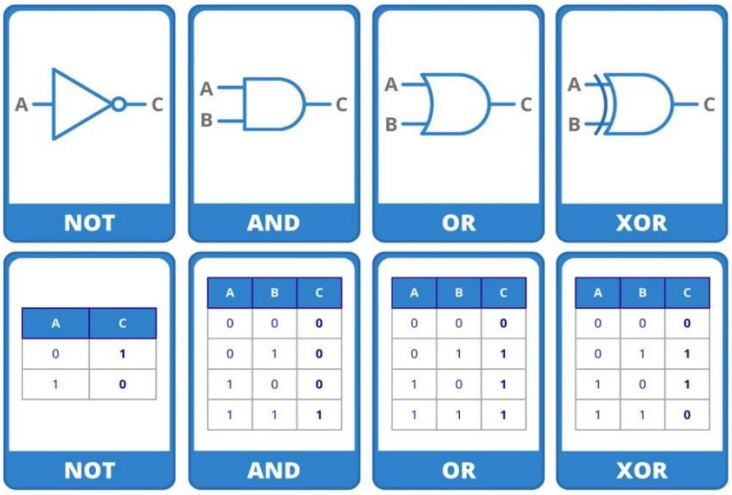


figura 3.2.3 (operațile NOT, AND, OR, XOR))

Am adăugat în plus operația logică XOR deoarece cu ajutorul ei vom implementa operația de complement față de 1 a unui număr, necesară pentru realizarea operație de scădere dintre 2 numere(figura 3.2.2).

3.2.5 Soluția pentru operația de rotație la stânga și dreapta

Pentru implementarea operației de rotire(stânga și dreapta), vom trata operandul ca un buffer circular de biți iar astfel cel mai semnificativ respectiv, cel mai nesemnificativ bit vor putea fi rotiți ușor.

Această operație este denumită și "rotație fără transport", biții sunt "rotați" ca și cum capetele stânga și dreapta ale registrului ar fi unite.

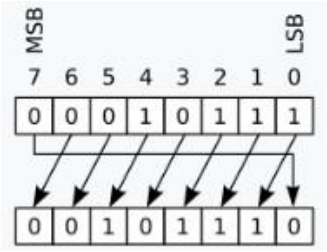


figura 3.2.4 (rotire la stânga)

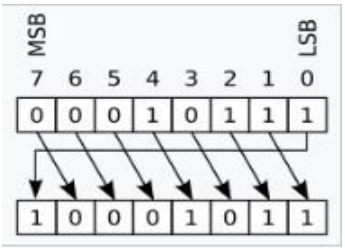


figura 3.2.5 (rotire la dreapta)

3.2.6 Soluția pentru operația de înmulțire

Pentru implementarea înmulțiri vom utiliza algoritmul din figura 2.3.4.

După cum am explicat și mai sus la punctul 2, algoritmul este similar cu înmulțirea realizată pe hârtie. Această metodă adună multiplicandul **X** cu el însuși de **Y** ori, unde **Y** reprezintă factorul de înmulțire.

În înmulțirea pe hârtie, algoritmul presupune parcurgerea cifrelor factorului de înmulțire una câte una, de la dreapta la stânga, înmulțirea multiplicandului cu o singură cifră a factorului de înmulțire și plasarea produsului intermediar în pozițiile corespunzătoare, la stânga rezultatelor anterioare. (avem mai jos un exemplu pentru mai multă claritate)

```
Multiplicand          1000 ×
Multiplier            1001
                      1000
                     0000
                    0000
                   1000
Product            1001000
```

În cazul înmulțirii binare, deoarece cifrele sunt 0 și 1, fiecare pas al înmulțirii este simplu. Dacă cifra factorului de înmulțire este 1, o copie a multiplicandului este plasată în pozițiile corespunzătoare; dacă cifra factorului de înmulțire este 0, un șir de cifre de 0 este plasat în pozițiile corespunzătoare.

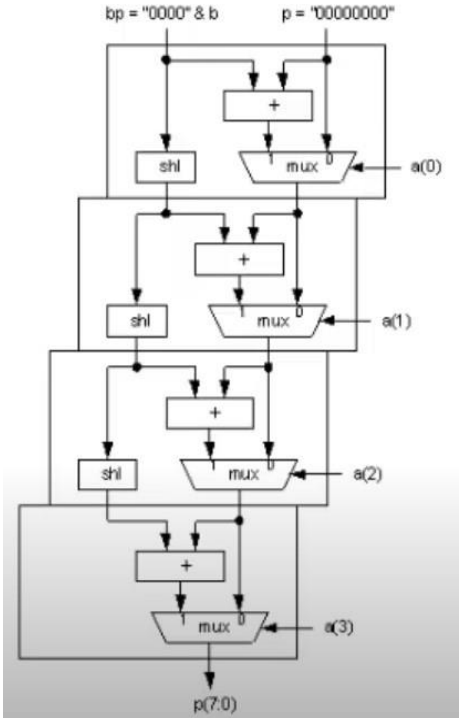


figura 3.2.6 (schema algoritmului de înmulțire)

Algoritmul folosește adunarea repetată și shiftarea pentru a realiza înmulțirea, urmând o metodă eficientă similară cu înmulțirea manuală binară. În exemplul dat mai jos, procesul este extins la 32 de biți pentru fiecare operand, iar produsul final este stocat pe 64 de biți pentru a preveni depășirile de domeniu.

Exemplu pas cu pas pentru algoritmul de înmulțire:

multiplicand = “0011” și multiplier = “0101” iar rezultatul este PV = **X”0000 0111”**

Pasul	**PV**	**BP**	**Bitul curent**	**Operația**
1	0000 0000	0000 0101	0	Registrul pentru **valoarea parțială** a produsului Extindem multiplier la 8 biți
2	0000 0101	0000 1010	multiplicand[0] = 1	PV = PV + BP Deplasare stânga BP cu o poziție
3	0000 1111	0001 0100	multiplicand[1] = 1	PV = PV + BP Deplasare stânga BP cu o poziție
4	0000 1111	0010 1000	multiplicand[2] = 0	Nu adăugăm nimic la PV (bitul curent este 0) Deplasare stânga BP cu o poziție
5	**0000 1111**	0101 0000	multiplicand[3] = 0	Nu adăugăm nimic la PV (bitul curent este 0) Deplasare stânga BP cu o poziție

3.2.7 Soluția pentru operația de împărțire

Pentru implementarea împărțire vom utiliza algoritmul din figura 2.3.5.

Numerele binare conțin doar cifrele 0 și 1, astfel încât diviziunea binară este limitată la aceste două valori. Operația de împărțire constă într-o serie de scăderi ale împărțitorului din restul parțial, care sunt executate doar dacă împărțitorul este mai mic decât restul parțial; în acest caz, cifra corespunzătoare din cât este 1; altfel, cifra corespunzătoare din cât este 0.

```
 1001010 : 1000 = 1001      Quotient
-1000
    10                      Partial remainders
    101
    1010
   -1000
     10                     Remainder
```

Deplasarea restului parțial la stânga, produce aceeași aliniere și simplifică hardware-ul necesar pentru ALU și registrul împărțitorului.

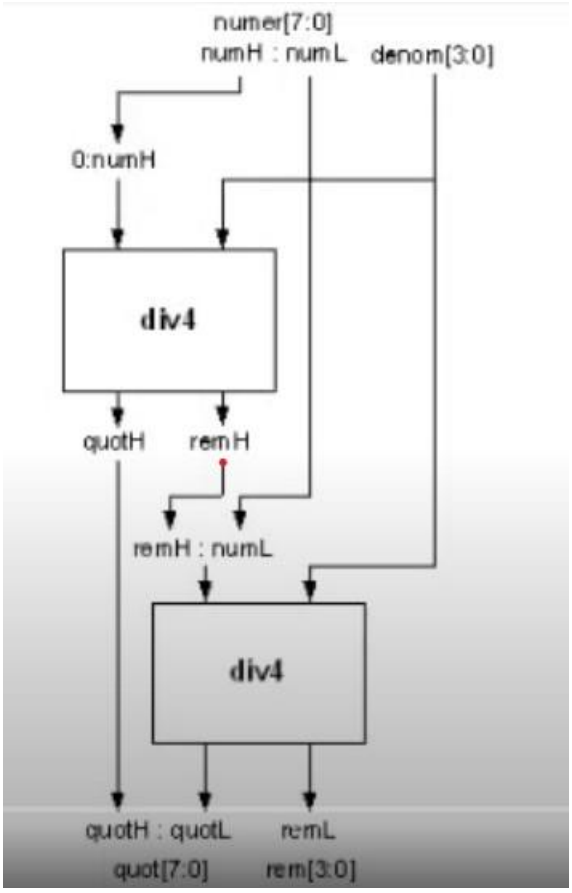


figura 3.2.7 (schema algoritmului de împărțire)

Acest algoritm reproduce pas cu pas împărțirea binară folosind o combinație de **shift și scădere**. Este eficient pentru implementări hardware și respectă formatul fix pe 32 de biți al datelor. Mai jos avem un exemplu pentru mai multă claritate.

numerator = “1101”, denominator = “0011”, rezultat : quotient = ”0100” și remainder = “0001”

Pas	**d**	**n1**	**n2**	**Bitul curent**	**Operația**
1	0000011	0000 0000	1101	0	Extindem **d**enominator cu un bit de 0 în față n1 = 0000 0000 (restul curent inițial) n2 = numerator (numeratorul original)
2	0000011	0000 0001	1010	numerator[3] = 1	Extindem n1 = 00000000 << 1 + 1 Verificăm n1 >= d Shiftăm n2 și lăsăm bitul curent din cât la 0
3	0000011	0000 0011 0000 0000	0010	numerator[2] = 1	Extindem n1 = 00000000 << 1 + 1 Verificăm n1 >= d Actualizăm restul Setăm bitul corespunzător din n2 la 1
4	0000011	0000 0000	0010	numerator[1] = 0	Extindem n1 = 00000000 << 1 + 0 Verificăm n1 >= d Lăsăm bitul curent din cât la 0
5	0000011	0000 0001 (remainder)	0100 (quotient)	numerator[0] = 1	Extindem n1 = 00000000 << 1 + 0 Verificăm n1 >= d Lăsăm bitul curent din cât la 0

Implementarea

Implementarea proiectului începe prin a ne asigura ca toate operațile funcționează după așteptările dorite. Astfel am implementat fiecare componenta individual și am testat fiecare funcționalitate în parte.

4.1 Implementarea operației de adunare

Pentru implementarea operației de adunare, am folosit soluția propusă de la punctul 3.2.1.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity FullAdderOn1Bit is
5    Port (A : in STD_LOGIC;
6          B : in STD_LOGIC;
7          Cin : in STD_LOGIC;
8          S : out STD_LOGIC;
9          Cout : out STD_LOGIC);
10 end FullAdderOn1Bit;
11
12 architecture Behavioral of FullAdderOn1Bit is
13
14 signal P : STD_LOGIC;
15 signal Pxor : STD_LOGIC;
16
17 begin
18     S <= A xor B xor Cin;
19     P <= A and B;
20     Pxor <= A xor B;
21     Cout <= P or (Pxor and Cin);
22 end Behavioral;
```

După cum se poate observa mai sus, am implementat un sumator pe 1 bit folosind minimizarea tabelului de adevăr prezentat în secțiunea 2.3. Astfel am obținut 2 ecuații de bază pentru implementarea adunării a 2 numere pe 1 bit:

- **Sum = A xor B xor Cin**
- **Cout = A and B + (A xor B) and Cin**

Iar pentru a obține un sumator pe 32 de biți, am cascadat acest sumator pe 1 bit obținund astfel implementarea care urmează:

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

entity FullAdderOn32Bit is
  Port (A : in STD_LOGIC_VECTOR(31 downto 0);
        B : in STD_LOGIC_VECTOR(31 downto 0);
        Sum : out STD_LOGIC_VECTOR(31 downto 0);
        Cout : out STD_LOGIC);
end FullAdderOn32Bit;

architecture Behavioral of FullAdderOn32Bit is

component FullAdderOn1Bit is
  Port (A : in STD_LOGIC;
        B : in STD_LOGIC;
        Cin : in STD_LOGIC;
        S : out STD_LOGIC;
        Cout : out STD_LOGIC);
end component;

signal carry : STD_LOGIC_VECTOR(31 downto 0);

begin
    ForFirstBit : FullAdderOn1Bit port map(A => A(0), B => B(0), Cin => '0', S => Sum(0), Cout => carry(0));

    For1To30Bits : for i in 1 to 30 generate
        For1Bit : FullAdderOn1Bit port map(A => A(i), B => B(i), Cin => carry(i-1), S => Sum(i), Cout => carry(i));
    end generate For1To30Bits;

    ForLastBit : FullAdderOn1Bit port map(A => A(31), B => B(31), Cin => carry(30), S => Sum(31), Cout => Cout);
end Behavioral;
```

Am adunat cele 2 numere bit cu bit iar la început semnalul de **Cin** este 0.

La pasul 2, semnalul de **Cin** este de fapt valoarea pe care am avut-o anterior pe semnalul de **Cout** iar astfel vom obține cascadarea menționată la figura 3.2.1.

La ultimul pas, semnalul de **Cin** se va baza pe exact același principiu ca la pasul 2 iar semnalul **Cout** va rezulta în urma operație de adunare între ultimi biți a celor 2 numere.

4.2 Implementarea operației de scădere

Pentru implementarea operației de scădere, am folosit soluția propusă la punctul 3.2.2, care consta în realizarea acestei operații în complement față de 2.

Scăderea în complement față de 2, se bazează în principal pe operația de adunare și ecuația prezentată în figura 3.2.2.

Implementarea este după cum urmează:

```
entity FullSubtractorOn32Bit is
    Port (A : in STD_LOGIC_VECTOR(31 downto 0);
          B : in STD_LOGIC_VECTOR(31 downto 0);
          Dif : out STD_LOGIC_VECTOR(31 downto 0);
          Borrow : out STD_LOGIC);
end FullSubtractorOn32Bit;

architecture Behavioral of FullSubtractorOn32Bit is

component FullAdderOn1Bit is
  Port (A : in STD_LOGIC;
        B : in STD_LOGIC;
        Cin : in STD_LOGIC;
        S : out STD_LOGIC;
        Cout : out STD_LOGIC);
end component;

signal notB : STD_LOGIC_VECTOR(31 downto 0);
signal difference : STD_LOGIC_VECTOR(31 downto 0);
signal bor : STD_LOGIC_VECTOR(31 downto 0);

begin
    notB <= not(B);
    ForFirstBit : FullAdderOn1Bit port map(A => A(0), B => notB(0), Cin => '1', S => difference(0), Cout => bor(0));
    For1To30Bits : for i in 1 to 30 generate
        For1Bit : FullAdderOn1Bit port map(A => A(i), B => notB(i), Cin => bor(i-1), S => difference(i), Cout => bor(i));
    end generate For1To30Bits;
    ForLastBit : FullAdderOn1Bit port map(A => A(31), B => notB(31), Cin => bor(30), S => difference(31), Cout => Borrow);
    Dif <= difference;
end Behavioral;
```

Principiul se bazează exact ca cel de la adunare, iar singurele modificări ar fi adunare operandului **A** cu complementul față de 1 a operandului **B** (A + not(B), unde not(B) este complementul față de 1 a numărului B) și setarea la început a semnalul **Cin** ca fii 1 pentru a putea aduna un 1 la sfârșitul operației(A – B = A + not(B) + 1).

4.3 Implementarea unității aritmetico-logice

Unitate aritmetico-logică va realiza 12 operații, conform tablelului de mai jos:

Semnalul de selecție (SEL)	Operația executată
0000	adunare (A + B)
0001	scădere (A - B)
0010	negare (not(A))
0011	și (A AND B)
0100	sau (A OR B)
0101	sau exclusiv (A XOR B)
0110	incrementare (inc A)
0111	decrementare (dec A)
1000	rotire la stânga
1001	rotire la dreapta
1010	înmulțire (A * B)
1011/1100	împărțire (A / B)

În implementarea unității aritmetico-logice, am comasat câteva operații (AND, OR, NOT, XOR, incrementarea, decrementare și rotire la sânga sau dreapta) deoarece aceste operații sunt deja implementate în VHDL(AND, OR, NOT, XOR), fie am folosit componente care erau deja implementate(sumator, scăzător) sau operațile nu necesitau componente noi, implementare lor fiind una ușoară.

Pentru implemenetarea unității aritmetico-logice am folosit **with var select** pentru ca rezultatele operaților să fie sincronizare cu semnalul de ceas, ca să evităm întârzierile semnalului **ALUOut**.

Pentru operațile mai complexe(adunare, scădere în complement față de 2) am mapat rezultatul intr-un semnal local, după care am atribuit valoarea acestuia semnalului **ALUOut** pentru a evita diferite erori de sintaxă VHDL.

Exact după cum spunem și mai sus, găsim to aici implementarea operaților:

- logice(AND, OR, NOT, XOR) – acestea fiind deja implementate în VHDL pe baza tabelelor de adevăr(vezi figura 3.2.3)
- incrementare și decrementare – aceste 2 operații vor incrementa respectiv decrementa doar operandul **A**, iar acestea sunt implementate pe baza operaților de adunare și scădere
- rotire la stânga și dreapta – aceste operații sunt implementate pe baza figurilor 3.2.4 și 3.2.5 și a explicaților de la punctul 3.2.5, iar acestea vor rotii la stânga/dreapta doar operandul **A**
- operații aritmetice complexe(înmulțire și împărțire) – aceste operații sunt implementate pe baza figurilor 3.2.6, 3.2.7 și a exemplelor din secțiunile 3.2.6 și 3.2.7.

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

entity ALU is
    Port (A : in STD_LOGIC_VECTOR(31 downto 0);
          B : in STD_LOGIC_VECTOR(31 downto 0);
          SEL : in STD_LOGIC_VECTOR(3 downto 0);
          ALUOut : out STD_LOGIC_VECTOR(63 downto 0));
end ALU;

architecture Behavioral of ALU is
component FullAdderOn32Bit is
   Port (A : in STD_LOGIC_VECTOR(31 downto 0);
         B : in STD_LOGIC_VECTOR(31 downto 0);
         Sum : out STD_LOGIC_VECTOR(31 downto 0);
         Cout : out STD_LOGIC);
end component;

component FullSubtractorOn32Bit is
    Port (A : in STD_LOGIC_VECTOR(31 downto 0);
          B : in STD_LOGIC_VECTOR(31 downto 0);
          Dif : out STD_LOGIC_VECTOR(31 downto 0);
          Borrow : out STD_LOGIC);
end component;

component MultiplierOn32Bit is
    Port ( multiplicand : in STD_LOGIC_VECTOR(31 downto 0);
           multiplier : in STD_LOGIC_VECTOR(31 downto 0);
           product : out STD_LOGIC_VECTOR(63 downto 0)
           );
end component;


component DividerOn32Bit is
    port( numerator : in STD_LOGIC_VECTOR(31 downto 0);
          denominator : in STD_LOGIC_VECTOR(31 downto 0);
          quotient : out STD_LOGIC_VECTOR(31 downto 0);
          remainder : out STD_LOGIC_VECTOR(31 downto 0));
end component;

signal MultResult : STD_LOGIC_VECTOR(63 downto 0);

signal IntermediateALUOutSum : STD_LOGIC_VECTOR(31 downto 0);
signal IntermediateALUOutSub : STD_LOGIC_VECTOR(31 downto 0);
signal Inc : STD_LOGIC_VECTOR(31 downto 0);
signal Dec : STD_LOGIC_VECTOR(31 downto 0);
signal Quotient : STD_LOGIC_VECTOR(31 downto 0);
signal Remainder : STD_LOGIC_VECTOR(31 downto 0);

signal CarryOut : STD_LOGIC;
signal Borrow : STD_LOGIC;

begin
    Addition : FullAdderOn32Bit port map(A => A, B => B, Sum => IntermediateALUOutSum, Cout => CarryOut);
    Subtraction : FullSubtractorOn32Bit port map(A => A, B => B, Dif => IntermediateALUOutSub, Borrow => Borrow);
    Incrementation : FullAdderOn32Bit port map(A => A, B => X"00000001", Sum => Inc, Cout => CarryOut);
    Decrementation : FullSubtractorOn32Bit port map(A => A, B => X"00000001", Dif => Dec, Borrow => Borrow);
    Multiplication : MultiplierOn32Bit port map(multiplicand => A, multiplier => B, product => MultResult);
    Division : DividerOn32Bit port map(numerator => A, denominator => B, quotient => Quotient, remainder => Remainder);

    with SEL select
        ALUOut <= X"00000000" & IntermediateALUOutSum when "0000",
                  X"00000000" & IntermediateALUOutSub when "0001",
                  X"00000000" & (not(A)) when "0010",
                  X"00000000" & (A and B) when "0011",
                  X"00000000" & (A OR B) when "0100",
                  X"00000000" & (A XOR B) when "0101",
                  X"00000000" & Inc when "0110",
                  X"00000000" & Dec when "0111",
                  X"00000000" & (A(30 downto 0) & A(31)) when "1000",
                  X"00000000" & (A(0) & A(31 downto 1)) when "1001",
                  MultResult when "1010",
                  X"00000000" & Quotient when "1011",
                  X"00000000" & Remainder when "1100",
                  X"0000000000000000" when others;
end Behavioral;
```

4.4 Implementarea algoritmului de înmulțire

Arhitectura implementează **algoritmul de multiplicare binară** folosind o buclă de 32 de pași pentru a calcula produsul bit cu bit.

Variabilele utilizate în proces:

- **pv (Partial Value):** Un registru pe 64 de biți pentru acumularea parțială a rezultatului în timpul calculelor.
- **bp (Bit Product):** O copie extinsă pe 64 de biți a multiplier, utilizată pentru efectuarea operațiilor de adunare și shiftare.

Algoritmul pas cu pas:

1. Inițializare:
    - pv este setat la 0 (rezultatul parțial inițial)
    - bp este obținut prin concatenarea unui vector de 32 de biți de 0 în fața valorii multiplier
2. Buclă pentru fiecare bit din multiplicand (de la cel mai puțin semnificativ la cel mai semnificativ):
    - Se verifică dacă bitul curent al multiplicand este 1: dacă da, se adaugă bp la pv
    - bp este shiftat la stânga cu un bit (echivalent cu multiplicarea cu 2)
3. La finalul buclei:
    - Operația de **shiftare la stânga** a lui bp corespunde multiplicării cu puteri ale lui 2, în funcție de poziția bitului curent

Caracteristici:

- Algoritmul este simplu, dar eficient pentru hardware, deoarece utilizează doar operații de **adunare** și **shiftare**.
- Rezultatul este stocat pe 64 de biți, astfel încât să poată gestiona toate produsele posibile rezultate din înmulțirea a doi operanzi pe 32 de biți.
- Este potrivit pentru implementări hardware pe FPGA sau ASIC.

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

entity MultiplierOn32Bit is
    Port ( multiplicand : in STD_LOGIC_VECTOR(31 downto 0);
           multiplier : in STD_LOGIC_VECTOR(31 downto 0);
           product : out STD_LOGIC_VECTOR(63 downto 0)
           );
end MultiplierOn32Bit;

architecture Behavioral of MultiplierOn32Bit is

begin
    process(multiplicand, multiplier)
    variable pv, bp : STD_LOGIC_VECTOR(63 downto 0);
    begin
        pv := (others => '0');
        bp := X"00000000" & multiplier;
        for i in 0 to 31 loop
            if multiplicand(i) = '1' then
                pv := pv + bp;
            end if;

            bp := bp(62 downto 0) & '0';
        end loop;

        product <= pv;
    end process;
end Behavioral;
```

4.5 Implementarea algoritmului de împărțire

Arhitectura implementează logica de divizare folosind o procedură numită div32. Aceasta efectuează împărțirea bit cu bit, iterând prin fiecare bit al deimpartului (numerator).

Algoritmul pas cu pas:

1. Inițializare:
    - **d (denominator extins):** Împărțitorul (denom) este extins cu un bit suplimentar la stânga ('0' & denom), pentru a permite comparații între numere de lungimi diferite
    - **n1 (restul parțial):** Inițial, este setat la zero (toate biturile 0)
    - **n2 (deimpartul):** Inițial, este egal cu numer
2. Buclă de iterare (de la bitul cel mai semnificativ la cel mai puțin semnificativ):
    - Actualizarea restului temporar (n1): shift n1 la stânga și adăugăm bitul cel mai semnificativ din n2
    - Shift n2 la stânga și adăugăm un bit 0 în partea cea mai puțin semnificativă
    - Dacă n1 >= d, înseamnă că împărțitorul poate fi scăzut din restul current
    - Actualizăm n1 prin scădere (n1 := n1 - d) și setăm bitul cel mai puțin semnificativ din n2 la 1
3. La finalul buclei:
    - n2 conține câtul (quotient) și n1 conține restul (remainder)

Caracteristici:

- **Algoritm iterativ:** Procesul se desfășoară în 32 de pași, calculând câte un bit al câtului pe rând.
- **Precizie pe 32 de biți:** Atât câtul, cât și restul sunt calculate exact, fără pierdere de informație.
- **Generalitate:** Funcționează pentru orice numere binare pe 32 de biți (pozitive).

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

entity DividerOn32Bit is
    port( numerator : in STD_LOGIC_VECTOR(31 downto 0);
          denominator : in STD_LOGIC_VECTOR(31 downto 0);
          quotient : out STD_LOGIC_VECTOR(31 downto 0);
          remainder : out STD_LOGIC_VECTOR(31 downto 0));
end DividerOn32Bit;

architecture Behavioral of DividerOn32Bit is
procedure div32(numer : in STD_LOGIC_VECTOR(31 downto 0);
                denom : in STD_LOGIC_VECTOR(31 downto 0);
                quotient : out STD_LOGIC_VECTOR(31 downto 0);
                remainder : out STD_LOGIC_VECTOR(31 downto 0)) is
    variable d, n1 : STD_LOGIC_VECTOR(32 downto 0);
    variable n2 : STD_LOGIC_VECTOR(31 downto 0);
```

```
architecture Behavioral of DividerOn32Bit is
procedure div32(numer : in STD_LOGIC_VECTOR(31 downto 0);
                denom : in STD_LOGIC_VECTOR(31 downto 0);
                quotient : out STD_LOGIC_VECTOR(31 downto 0);
                remainder : out STD_LOGIC_VECTOR(31 downto 0)) is
    variable d, n1 : STD_LOGIC_VECTOR(32 downto 0);
    variable n2 : STD_LOGIC_VECTOR(31 downto 0);
begin
    d := '0' & denom;
    n1 := (others => '0');
    n2 := numer;

    if n2 = X"00000000" then
        quotient := n2;
        remainder := denom;
    else
        for i in 0 to 31 loop
            n1 := n1(31 downto 0) & n2(31);
            n2 := n2(30 downto 0) & '0';

            if n1 >= d then
                n1 := n1 - d;
                n2(0) := '1';
            end if;
        end loop;

        quotient := n2;
        remainder := n1(31 downto 0);
    end if;
end procedure;
```

```
begin
    process(numerator, denominator)
        variable remH, remL, quotH, quotL : STD_LOGIC_VECTOR(31 downto 0);
    begin
        div32(numerator, denominator, quotH, remH);

        quotient <= quotH;
        remainder <= remH;
    end process;
end Behavioral;
```

Testare și validare

5.1 Testarea pe placă

Pentru testarea Unități Aritmetico Logice am creat cateva componente suplimentare pentru a putea valida cat se poate de corect rezultatele primite.

5.1.1 Debouncer

Debouncing-ul este necesar pentru a elimina fluctuațiile electrice sau "zgomotul" care apare atunci când un buton este apăsat sau eliberat. Aceste fluctuații pot determina un semnal neclar (mai multe tranziții între 1 și 0 în loc de o singură tranziție curată).

Scop: Filtrează fluctuațiile rapide ale semnalului de intrare al butonului (btn_in) și produce un semnal stabilizat (btn_out), care schimbă starea numai după ce semnalul de intrare rămâne constant pentru un timp definit.

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

entity Debouncer is
    Port (
        btn_in : in STD_LOGIC;
        clk : in STD_LOGIC;
        btn_out : out STD_LOGIC
    );
end Debouncer;

architecture Behavioral of Debouncer is
    signal counter : integer := 0;
    signal stable_state : STD_LOGIC := '0';
begin
    process(clk)
    begin
        if rising_edge(clk) then
            if btn_in = stable_state then
                counter <= 0;
            else
                counter <= counter + 1;
                if counter > 10000 then -- Adapteaz? la frecven?a ceasului
                    stable_state <= btn_in;
                    counter <= 0;
                end if;
            end if;
        end if;
    end process;
    btn_out <= stable_state;
end Behavioral;
```

5.1.2 Unitate pentru generarea de semnale monoplus

MPG-ul este un **circuit digital pentru detectarea tranzițiilor unui buton** (btn) și generarea unui semnal de ieșire (enable) sincronizat cu un semnal de ceas (clk). Acest circuit folosește o combinație de contorizare și sincronizare pentru a detecta evenimente asociate cu butonul.

Scop: Detectarea și generarea unui semnal (enable) pentru tranziția de la 0 la 1 a unui semnal de buton (btn).

```
entity MPG is
    Port ( enable : out STD_LOGIC;
           btn : in STD_LOGIC;
           clk : in STD_LOGIC);
end MPG;
architecture Behavioral of MPG is
signal cnt_int : STD_LOGIC_VECTOR(17 downto 0) := (others => '0');
signal Q1, Q2, Q3 : STD_LOGIC;
begin
    enable <= Q2 and (not Q3);
    process(clk)
    begin
        if clk='1' and clk'event then
            cnt_int <= cnt_int + 1;
        end if;
    end process;
    process(clk)
    begin
        if clk'event and clk='1' then
            if cnt_int(17 downto 0) = "11111111111111111" then
                Q1 <= btn;
            end if;
        end if;
    end process;
    process(clk)
    begin
        if clk'event and clk='1' then
            Q2 <= Q1;
            Q3 <= Q2;
        end if;
    end process;
end Behavioral;
```

5.1.3 ROM(Read Only Memory) pentru gestionare operanzilor

Memora ROM(Read Only Memory) este o memorie pe care o putem accesa prin intermediul unui semnal de ceas (clk), unei adrese de intrare (Address) și unui semnal de activare a citirii (Read_En). Rezultatul citirii este trimis la ieșire (Data_out). Deasemenea din memoria ROM putem doar sa citim operanzi pe care îi vom folosii în operațile implementate în Unitatea Aritmetica Logică (UAL).

Scop: Este un ROM (memorie doar pentru citire) de 32 locații x 32 biți, cu acces pe baza unei adrese pe 4 biți.

```
entity ROM_Memory is
    Port (clk : in STD_LOGIC;
          Read_En : in STD_LOGIC;
          Address : in STD_LOGIC_VECTOR(3 downto 0);
          Data_out : out STD_LOGIC_VECTOR(31 downto 0));
end ROM_Memory;

architecture Behavioral of ROM_Memory is
type memROM is array (0 to 31) of STD_LOGIC_VECTOR(31 downto 0);
signal rom : memROM := (X"00000001", -- 1
                        X"00000005", -- 5
                        X"00000000", -- 0
                        X"00000036", -- 54
                        X"0000002E",  -- 46
                        X"000000FF", -- 510
                        X"00000008", -- 8
                        X"0000004B", -- 75
                        X"10101010", -- 269488144
                        X"10000001", -- 268435457
                        others => X"00000000");

signal Data_reg : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
begin
    process(clk)
    begin
        if rising_edge(clk) then
            if Read_En = '1' then
                Data_reg <= rom(conv_integer(Address));
            end if;
        end if;
    end process;

    Data_out <= Data_reg;
end Behavioral;
```

5.1.4 Gestionarea adresei de memorie(Memory Address Management)

Componenta **gestionează adrese de memorie**, folosind două bănci de memorie (A și B). Modulul alternează între actualizarea adreselor celor două bănci pe baza unui semnal de ceas (clk) și a unui semnal de activare (en).

Scop: Util pentru circuite care necesită scriere/citire alternantă între două bănci de memorie, cum ar fi în aplicații de procesare paralelă sau pentru buffering.

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

entity MemoryAddressManagement is
    Port (clk : in STD_LOGIC;
          en : in STD_LOGIC;
          reset : in STD_LOGIC;
          enable_A : out STD_LOGIC;
          enable_B : out STD_LOGIC;
          Address_A : out STD_LOGIC_VECTOR(3 downto 0);
          Address_B : out STD_LOGIC_VECTOR(3 downto 0)
          );
end MemoryAddressManagement;

architecture Behavioral of MemoryAddressManagement is
signal addressA : STD_LOGIC_VECTOR(3 downto 0) := (others => '0');
signal addressB : STD_LOGIC_VECTOR(3 downto 0) := (others => '0');
signal toggle : STD_LOGIC := '0';
```

```
begin
    process(clk, reset)
    begin
        if reset = '1' then
            toggle <= '0';
            addressA <= (others => '0');
            addressB <= (others => '0');
        elsif rising_edge(clk) then
            if en = '1' then
                toggle <= not toggle;
                if toggle = '0' then
                    addressA <= std_logic_vector(unsigned(addressA) + 1);
                else
                    addressB <= std_logic_vector(unsigned(addressB) + 1);
                end if;
            end if;
        end if;
    end process;
```

```
    process(toggle, en)
    begin
        if en = '1' then
            if toggle = '0' then
                enable_A <= '1';
                enable_B <= '0';
            else
                enable_A <= '0';
                enable_B <= '1';
            end if;
        else
            enable_A <= '0';
            enable_B <= '0';
        end if;
    end process;

    Address_A <= addressA;
    Address_B <= addressB;
end Behavioral;
```

5.1.5 Componenta test_env (test environment = mediu de testare)

Unitatea test_env este un mediu de testare (testbench) implementat în VHDL care integrează mai multe componente pentru a simula și demonstra funcționalitatea unui sistem digital complex. Aceasta combină module precum debouncer-ul, generatorul de semnale sincronizate, managementul memoriei, o unitate aritmetică și logică (ALU), și un afișaj cu 7 segmente. Scopul acestei unități este de a permite testarea interacțiunii dintre componente și verificarea funcționalității lor într-un sistem integrat.

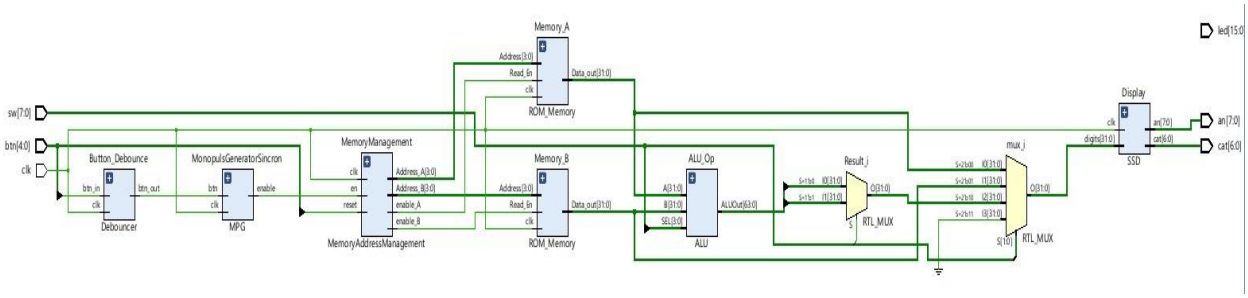
Importanța unității:

1) **Integrarea componentelor:**
   a) test_env este un exemplu de integrare funcțională a mai multor componente digitale.
   b) Permite testarea și verificarea funcționării fiecărei componente într-un sistem real.
2) **Flexibilitate:**
   a) Comutatoarele și butoanele oferă utilizatorului control asupra operațiilor, ceea ce facilitează testarea diferitelor scenarii.
3) **Modularitate:**
   a) Fiecare componentă este implementată separat și poate fi reutilizată sau înlocuită, dacă este necesar.
4) **Educațional:**
   a) Este un mediu ideal pentru învățarea conceptelor legate de proiectarea digitală, managementul memoriei și afișaj.

Scop:

1) **Simulare și testare:**
   a) Validarea funcționalității componentelor integrate și verificarea corectitudinii comunicației între ele.
2) **Demonstrativ:**
   a) Afișajul cu 7 segmente permite vizualizarea rezultatului operațiilor, fiind util în demonstrații sau prototipuri.
3) **Aplicații practice:**
   a) Poate fi folosit ca bază pentru implementarea unui sistem mai complex, cum ar fi un procesor simplificat sau un controler.

Schema RTL a Unități Aritmetico Logice




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

entity test_env is
    Port( sw : in STD_LOGIC_VECTOR(7 downto 0);
          btn : in STD_LOGIC_VECTOR(4 downto 0);
          clk : in STD_LOGIC;
          cat : out STD_LOGIC_VECTOR(6 downto 0);
          an : out STD_LOGIC_VECTOR(7 downto 0);
          led : out STD_LOGIC_VECTOR (15 downto 0));
end test_env;

architecture Behavioral of test_env is

component Debouncer is
    Port (
        btn_in : in STD_LOGIC;
        clk : in STD_LOGIC;
        btn_out : out STD_LOGIC
    );
end component;

component MPG is
    Port ( enable : out STD_LOGIC;
           btn : in STD_LOGIC;
           clk : in STD_LOGIC);
end component;
```

```
component SSD is
    Port ( clk : in STD_LOGIC;
           digits : in STD_LOGIC_VECTOR(31 downto 0);
           an : out STD_LOGIC_VECTOR(7 downto 0);
           cat : out STD_LOGIC_VECTOR(6 downto 0));
end component;

component ROM_Memory is
    Port (clk : in STD_LOGIC;
          Read_En : in STD_LOGIC;
          Address : in STD_LOGIC_VECTOR(3 downto 0);
          Data_out : out STD_LOGIC_VECTOR(31 downto 0));
end component;

component MemoryAddressManagement is
    Port (clk : in STD_LOGIC;
          en : in STD_LOGIC;
          reset : in STD_LOGIC;
          enable_A : out STD_LOGIC;
          enable_B : out STD_LOGIC;
          Address_A : out STD_LOGIC_VECTOR(3 downto 0);
          Address_B : out STD_LOGIC_VECTOR(3 downto 0)
          );
end component;

component ALU is
    Port (A : in STD_LOGIC_VECTOR(31 downto 0);
          B : in STD_LOGIC_VECTOR(31 downto 0);
          SEL : in STD_LOGIC_VECTOR(3 downto 0);
          ALUOut : out STD_LOGIC_VECTOR(63 downto 0));
end component;
```

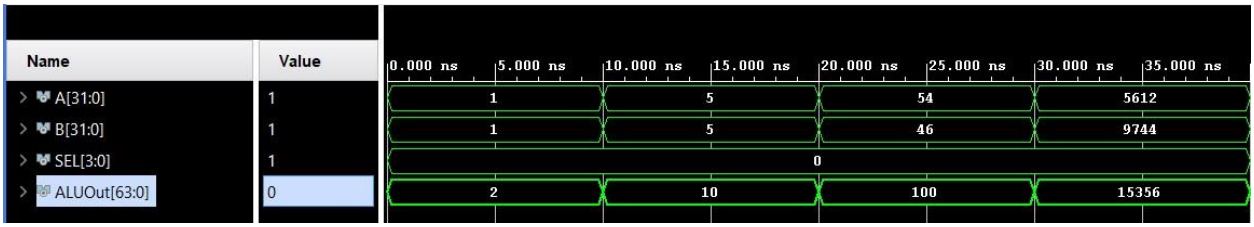
```
signal debounced_btn : STD_LOGIC;
signal en : STD_LOGIC;
signal enable_A : STD_LOGIC;
signal enable_B : STD_LOGIC;
signal Address_A : STD_LOGIC_VECTOR(3 downto 0);
signal Address_B : STD_LOGIC_VECTOR(3 downto 0);
signal DataOut_A : STD_LOGIC_VECTOR(31 downto 0);
signal DataOut_B : STD_LOGIC_VECTOR(31 downto 0);
signal mux : STD_LOGIC_VECTOR(31 downto 0);
signal Result : STD_LOGIC_VECTOR(31 downto 0);

signal DataALUOut : STD_LOGIC_VECTOR(63 downto 0);


begin
    Button_Debounce: Debouncer port map(btn(0), clk, debounced_btn);

    MonopulsGeneratorSincron : MPG port map(en, debounced_btn, clk);

    MemoryManagement : MemoryAddressManagement port map(clk, en, btn(1), enable_A, enable_B, Address_A, Address_B);

    Memory_A : ROM_Memory port map(clk, enable_A, Address_A, DataOut_A);

    Memory_B : ROM_Memory port map(clk, enable_B, Address_B, DataOut_B);

    ALU_Op : ALU port map(DataOut_A, DataOut_B, sw(3 downto 0), DataALUOut);

    with sw(4) SELECT
        Result <= DataALUOut(31 downto 0) when '0',
                  DataALUOut(63 downto 32) when '1';

    with sw(6 downto 5) SELECT
        mux <= DataOut_A when "00",
               DataOut_B when "01",
               Result when "10",
               X"00000000" when "11";

    Display : SSD port map(clk, mux, an, cat);
end Behavioral;
```

5.2 Testare prin simularea

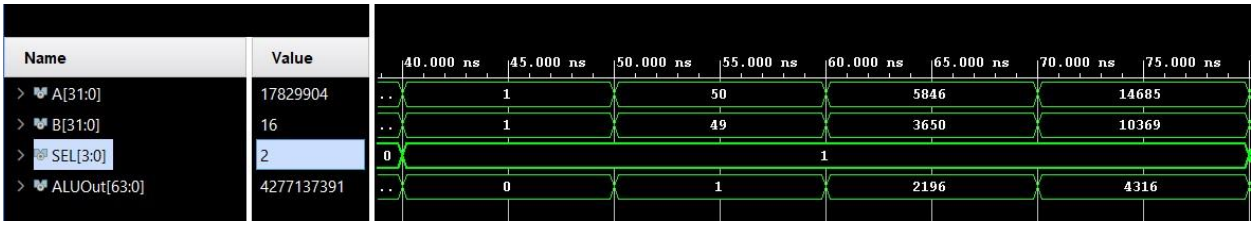
Simulările în VHDL sunt esențiale pentru orice proiect de design digital, având scopul de a verifica corectitudinea funcțională, de a optimiza performanța și de a preveni erorile costisitoare care pot apărea în timpul implementării hardware. Ele permit testarea în condiții variate, oferind inginerilor posibilitatea de a analiza și îmbunătăți designul într-un mediu controlat, reducând semnificativ riscurile și costurile asociate cu dezvoltarea hardware.

Astfel am simulat fiecare operație implementată de Uniatea Aritmetico Logică pentru a fii sigur de funcționalitate corectă a proiectului.

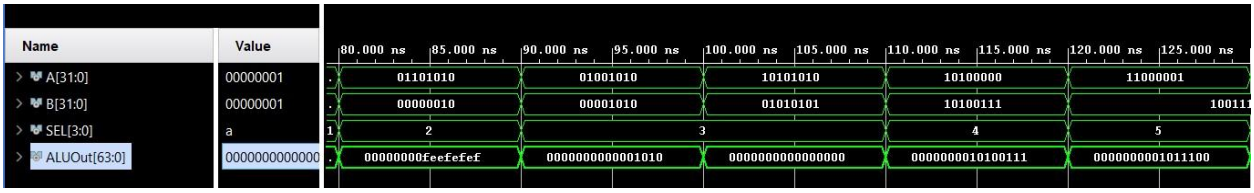
5.2.1 Operația de adunare(SEL = “0000” = 0)



5.2.2 Operația de scădere(SEL = “0001” = 1)

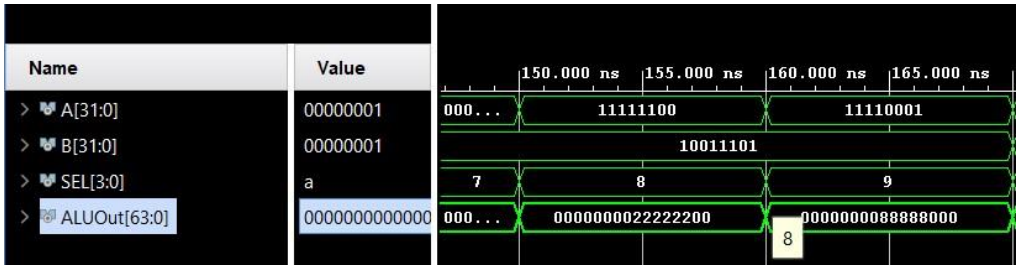


5.2.3 Operațile logice(SEL = “0010” = 2, “0011” = 3, “0100” = 4, “0101” = 5)

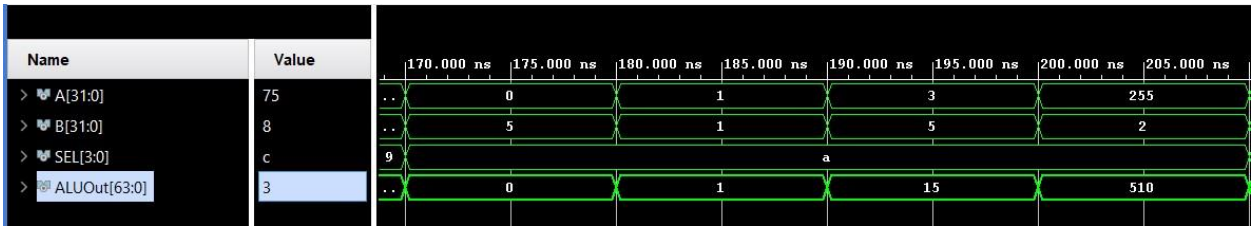


5.2.4 Operațile de incrementare și decrementare(SEL = “0110” = 6, “0111” = 7)

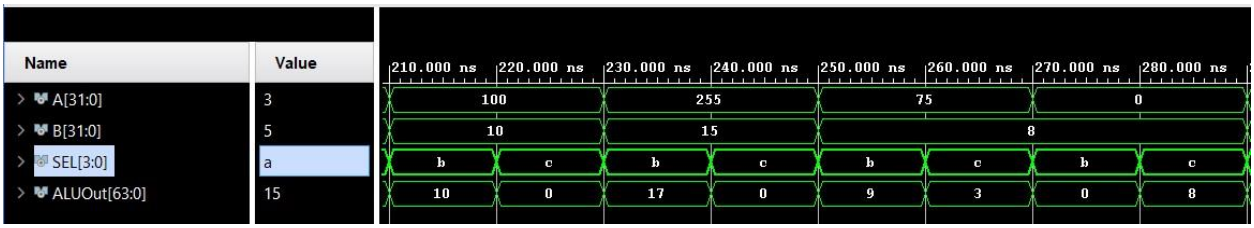
5.2.5 Operațile de rotire la stânga și dreapta(SEL = “1000” = 8, “1001” = 9)



5.2.6 Operația de înmulțire(SEL = “1010” = A = 10)



5.2.7 Operația de împărțire (SEL = “1011” = B = 11 (rest), “1100” = C = 12 (cât))



Concluzii

Unitatea Aritmetico-Logică (UAL) este un element central în orice sistem digital sau procesor, având un rol esențial în executarea operațiunilor matematice și logice.

UAL-ul este responsabil pentru realizarea operațiunilor fundamentale în orice sistem digital, cum ar fi adunarea, scăderea, multiplicarea, împărțirea, operațiile logice (AND, OR, XOR) și compararea. Aceste operațiuni sunt esențiale pentru calculul numeric, procesarea semnalelor și luarea deciziilor logice într-un procesor sau microcontroler. Fără UAL, sistemele de calcul ar fi incapabile să efectueze cele mai simple funcții aritmetice sau logice.

Eficiența și performanța unui procesor sunt în mare măsură dependente de designul și capacitatea UAL-ului. O UAL bine optimizată poate spori semnificativ viteza de procesare a datelor, iar un procesor care dispune de un UAL puternic poate executa mai rapid algoritmi complecși, ceea ce este esențial în aplicații precum criptografia, procesarea semnalului, și simulările științifice. În schimb, o UAL mai lentă sau mai puțin eficientă poate constitui un factor limitativ al performanței.

UAL-urile sunt extrem de versatibile și pot fi utilizate într-o gamă largă de aplicații. Acestea nu sunt doar parte integrantă a microprocesoarelor, ci și a sistemelor de control, procesorilor grafici (GPU-uri), circuitelor de procesare a semnalelor digitale (DSP) și multor alte tipuri de sisteme de calcul. Aceste aplicații variază de la calculul numeric simplu până la realizarea unor funcții logice complexe necesare în sistemele inteligente și în AI (Inteligența Artificială).

În majoritatea arhitecturilor de calcul, instrucțiunile procesorului sunt implementate prin operații aritmetico-logice. UAL-ul interpretează și execută instrucțiuni de tip aritmetic (cum ar fi adunarea și scăderea) și de tip logic (cum ar fi operațiile pe biți), având un rol crucial în interpretarea corectă a programelor și a algoritmilor implementați în software.

Bibliografie

1. Cartea Arhitectura Calculatoarelor (Prof. Baruch Zoltan Francisc)
2. Cartea De la bit la procesor (Prof. Florin Oniga)
3. https://www.geeksforgeeks.org/multiplication-algorithm-in-signed-magnitude-representation/
4. https://users.utcluj.ro/~baruch/book_ssce/SSCE-Basic-Division.pdf
5. https://users.utcluj.ro/~baruch/book_ssce/SSCE-Shift-Mult.pdf
6. https://users.utcluj.ro/~baruch/ro/
7. https://users.utcluj.ro/~vcristian/AC.html