Universitatea Tehnica din Cluj-Napoca

Facultatea de Automatica si Calculatoare

Proiect: Unitate Aritmetica si Logica

Student: Bumbu Iulia-Diana

Grupa: 30238

Indrumator: Dragos Lisman

Data: 21/12/2020

Cuprins

1. Rezumat

In contextul familiarizarii cu materia Structura Sistemelor de Calcul, acest proiect a reprezentat o modalitate de aprofundare a cunostintelor acumulate pe parcursul semestrului. Implementarea unei unitati aritmetice si logice mi-a oferit prilejul de a intelege mult mai bine modul in care calculatorul isi prelucreaza datele si informatiile folosindu-se de circuitele electronice. Obiectivul principal al proiectului a fost sa dezvolt o rezolvare in urma unor alegeri pe deplin sustinute de intelgerea conceptelor de baza si diverselor modalitati de rezolvare a aceluiasi tip de operatie. Astfel, folosind limbajul VHDL si mediul de dezvoltare Xilinx ISE Design Suite am propus o unitate aritmetica si logica care manipuleaza numere in binar fara semn si dispune de operatiile de baza ale unui sistem de calcul, printre care adunarea, scaderea, inmultirea, impartirea precum si cele logice. Rezultatele au fost testate atat prin simulari cat si pe placa Basys 3 Artix-7 FPGA. In urma realizarii tuturor functionalitatilor propuse am constatat ca alegerea si intelegerea metodei de rezolvare este extrem de importanta influentand performanta sistemului si corectitudinea rezultatului furnizat.

2. Introducere

2.1 Contextul temei proiectului

Deoarece Unitatea aritmetica si logica este un bloc fundamental din cadrul Unitatii Centrale de Prelucrare, de-a lungul timpului aceasta a suferit multiple schimbari si imbunatatiri, concomitent cu evolutia calculatoarelor. Performanta acesteia este stabilita tinand cont de rapiditatea efectuarii operatiei si de numarul de operatii pe care aceasta este capabila sa il efectueze. Actualmente, sistemele UAL sunt imbunatatite prin utilizarea unor algoritmi si componente hardware mult mai rapide, prin utilizarea tehnicii de pipeline atunci cand operatiile din UAL necesita mai multe perioade de executie dar si prin folosirea unui numar mare de registri de uz general in structura procesorului de unde UAL isi ia datele. Astfel, se reduce numarul de accese la memorie, crescand viteza UAL. In perioada timpurie a anilor 1960 se foloseau unitati de procesare cu multiple unitati de functionare in care fiecare unitate era dedicata unui singur tip de operatie, permitand executia simultana a mai multor operatii si crescand viteza procesorului. Acum, sunt comune sistemele cu multiple UAL, acestea fiind capable sa functioneze simultan si sa efectueze toate tipurile de operatii. [1]

2.2 Domeniul de studiu

Utilizarea unitatii aritmetice si logice tine de domeniul IT, care a cunoscut o continua atentie si evolutie de-a lungul ultimelor decenii deoarece sistemele de calcul se schimba si se imbunatatesc intr-un ritm alert, concomitent cu cererea, un calculator devenind un lucru indispensabil in viata de zi cu zi. Este incontestabila necesitatea ei in cadrul unui sistem de calcul deoarece ea este responsabila de realizarea transformarilor aritmetice si logice pe operanzii primiti, rezultatul fiind returnat dispozitivului de stocare si utilizat in cadrul logicii de functionare a calculatorului.

2.3 Problema de rezolvat si obiective

Scopul acestui proiect este de a implementa o unitate aritmetica si logica care este capabila sa efectueze operatiile aritmetice de baza: adunare, scadere, inmultire si impartire, operatii logice: negare, SI, SAU, SAU-EXCLUSIV, COINCIDENTA si complement fata de 2, precum si shiftare logica la stanga si la dreapta.

Obiectivul proiectului este de a intelege modul de functionare a componentelor, de a alege o solutie optima de implementare a diferitelor operatii si de a aprofunda notiunile acumulate de-a lungul semestrului in cadrul materiei Structura Sistemelor de Calcul.

2.4 Solutia propusa

Am implementat o unitate aritmetica si logica capabila sa efectueze toate operatiile propuse anterior, avand operanzi generic. Astfel, codul poate fi adaptat pentru efectuarea operatiilor cu numere pe un numar de biti ales de noi. Operanzii sunt reprezentati in binar fara semn, interpretarea rezultatului ramanand la latitudinea utilizatorului. Pentru scrierea programului am utilizat limbajul de descriere hardware VHDL, iar pentru testarea pe o placa de dezvoltare FPGA (Field Programmable Gate Array) am utilizer Xilinx ISE. UAL proiectata are ca intrari doi operanzi pe n biti, un opcode de 4 biti pentru alegerea operatiei, precum si semnal de clock, reset si start utilizate in cadrul operatiei de impartire. Ca iesiri exista un semnal term ce indica finalul operatiei de impartire precum si rezultatul pe 2n biti, pentru a fi capabil sa salveze cor=rect rezultatul in cazul operatiei de inmultire sau impartire. Pentru realizarea operatiei de adunare am ales implementarea unui sumator cu propagare succesiva a transportului (“Ripple Carry Adder”) prin conectarea in serie a mai multor sumatoare elementare deoarece il putem utiliza si in cadrul operatiei de scadere daca consideram scazatorul in complement fata de 2. Operatia de inmultire s-a realizat utilizand inmultirea matriceala, iar cea de impartire folosind tehnica de impartire cu refacerea restului partial.

2.5 Sumar al sectiunilor urmatoare

In cele ce urmeaza se vor detalia notiuni teoretice esentiale in dezvoltarea solutiei. Se va descrie partea de proiectare si de implementare a logicii proiectului sustinute de explicatii, scheme si manual de utilizare si se vor furniza rezultatele experimentale obtinute in urma testarii atat in simulator cat si pe placa de dezvoltare Basys 3. In urma interpretarii rezultatelor se vor furniza o serie de concluzii menite sa sintetizeze intreaga esenta a proiectului, precum si scopul sau. Resursele bibliografice utilizate in realizarea proiectului vor fi specificate la final. In anexe se vor regasi schemele de detaliu si codul sursa.

3. Fundamentare teoretica

Unitatea aritmetica si logica reprezinta o componenta esentiala din cadrul unitatii centrale de prelucrare (UCP-poate fi un microprocessor) care se ocupa cu procesarea datelor. UAL este capabila in general sa efectueze doua mari tipuri de procesari: operatii aritmetice sau logice, al caror rezultat este stocat conform logicii de control in register.[2]

Exista mai multe modalitati prin care se pot implementa operatiile de tip aritmetic, astfel am avut posibilitatea de a alege o solutie optima privind atat viteza si cat costul circuitelor.

Adunarea are multiple implementari printre care sumatorul cu propagarea succesiva a transportului, sumatorul cu anticiparea transportului, sumatorul cu selectia transportului, sumatorul cu salvarea transportului, sumatorul serial sau sumatorul zecimal prezentate pe larg in sursa [3]

Inmultirea se poate implementa folosind inmultirea prin deplasare si adunare, tehnica Booth, inmultirea intr-o baza superioara, inmultirea matriceala, arborele Wallace sau circuit de inmultire pipeline. [3]

Impartirea se poate realiza cu sau fara refacerea restului partial utilizand un algoritm conform celui prezentat in sursa [3].

4. Proiectare si implementare

4.1 Metoda experimentala utilizata

Vhdl (VHSIC Hardware Description Language, unde VHSIC = Very High Speed Integrated Circuits) este limbajul ales pentru a modela sistemul, oferind astfel o implementare hardware folosind FPGA (Field Programmable Gate Arrays)- circuit integrat ce are hardware-ul programat de utilizator.

4.2 Solutia aleasa

Optiuni pe care le-am luat in considerare in cazul alegerii circuitului care sa efectueze adunarea sunt sumatorul cu anticiparea transportului - care reduce timpul necesar pentru formarea semnalelor de transport, sumatorul cu selectia transportului – utilizeaza circuite redundante pentru cresterea vitezei de adunare, sumatorul cu salvarea transportului- - util pentru mai mult de doi operanzi, sumatorul serial-simplu

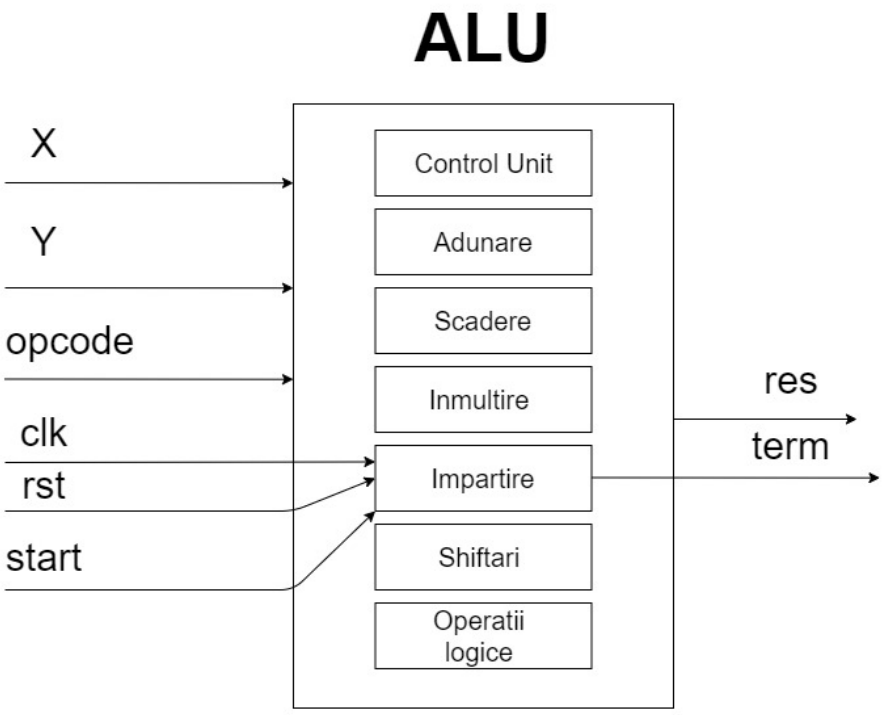
si cu cost redus si cel zecimal – nepotrivit insa deoarece operanzii UAL descrise de mine sunt reprezentati in binar nu in cod BCD. . Desi are viteza mai redusa, am ales metoda cu propagare succesiva a transportului deoarece are cost redus si poate fi folosit ca si scazator economisind astfel componente.

In cazul operatiei de inmultire alternativele au fost inmultirea prin deplasare si adunare care implementeaza metoda de inmultire cunoscuta de la matematica, tehnica Booth care testeaza doi biti ai inmultitorului in fiecare pas, inmultirea intr-o baza superioara care reduce numarul etapelor prin examinarea mai multor bitit ai inmultitorului in fiecare etapa, arborele Wallace care aduna mai rapid produsele partiale precum si tehnica pipelone care permite cresterea vitezei operatiei. Am ales varianta cu inmultire matriceala deoarece utilizarea circuitelor combinationale de inmultire creste viteza si confera o structura uniforma care permite implementarea intr-un circuit VLSI.

Pentru impartire am luat in considereare si impartirea fara refacerea restului partial la care fiecare adunare a impartitorului la restul partial este urmata de o scadere in pasul urmator. Desi am ales varianta cu refacerea restului partial care creste timpul de executie al operatiei, am abordat varianta cu imbunatatiri care reduce dimensiunea registrelor si simplifica circuitele.

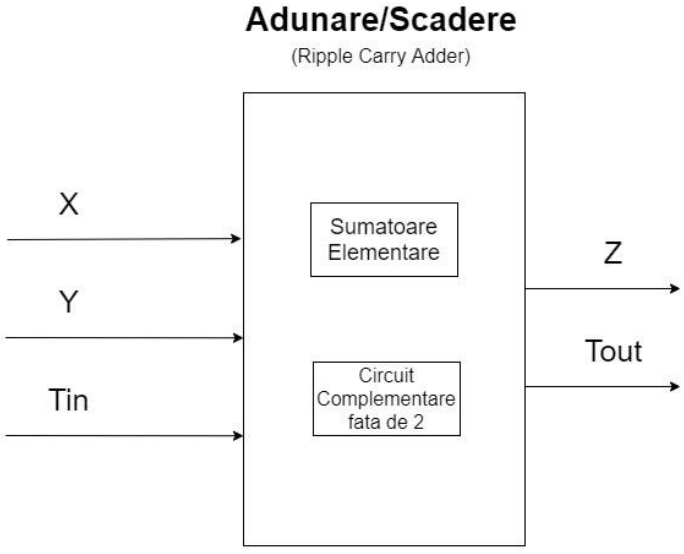
4.3 Scheme bloc si arhitectura generala a sistemului

Arhitectura generala a sistemului este formata dintr-un bloc de control care in functie de opcode selecteaza ce operatie se efectueaza si module specializate pe fiecare tip de operatie. Operatiile aritmetice primesc doi operanzi si genereaza un rezultat. Se observa ca operatioa de impartire primeste suplimentar semnal de clock, reset si de start, generand in plus si un semnal term care simbolizeaza terminarea executiei algoritmului. Operatiile logice sunt reprezentate general in blocul cu numele aferent, insa ele constau in module separate pentru fiecare tip de operatie.



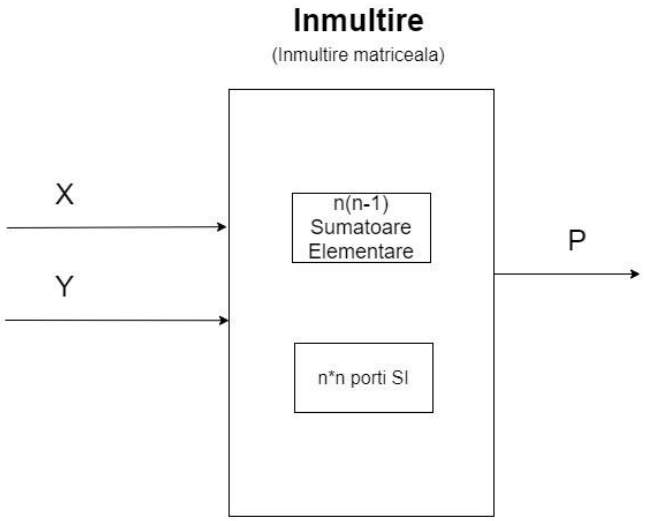
Arhitectura generala de sistem Figura 1

Adunarea si scaderea sunt similare, diferenta fiind ca scaderea primeste ca parametru scazatorul convertit in complement fata de doi. Schema bloc care surprinde intrarile si iesirile corespunzatoare este cea din figura 2. Modul de cascadare al sumatoarelor este surprins in anexa A, insa genralizandu-se ideea de la 4 la n biti.



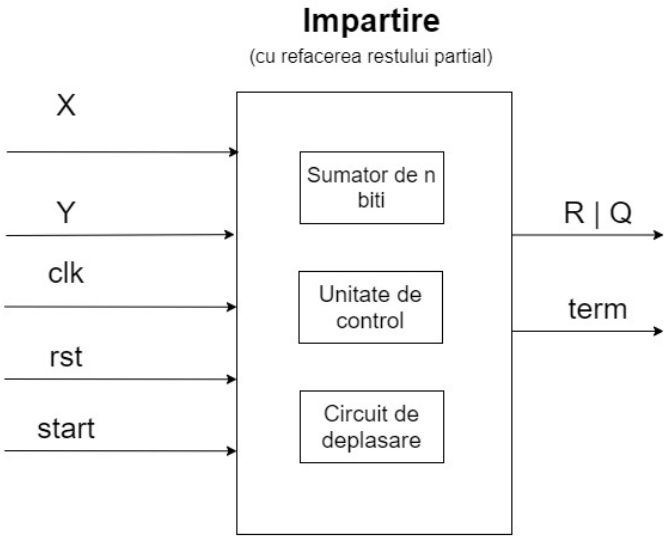
Schema bloc adunare/scadere Figura 2

In cazul operatiei de inmultire schema bloc contine black box-uri pentru sumatoarele elementare si portile si folosite. Schema de detaliu cu toate interconectarile existente se regaseste in anexa B, conceptul aplicat pe 8 biti fiind generalizat la n biti.



Schema bloc inmultire Figura 3

Operatia de impartire se diferentiaza de celelalte deoarece aceasta se efectueaza secvential, sincron cu semnalul de clock primit ca intrare. Inceperea efectuarii calculelor se va face doar dupa ce semnalul de start este actival. La finalul executiei semnalul term se activeaza si rezulatatul este generat sub forma REST concatenate cu CAT. Detalii despre organizarea interna si organigrama operatiei sunt surprinse in anexa C



Schema bloc impartire Figura 4

Operatiile logice au constructie asemanatoare: complement fata de doi primeste un operand si genereaza rezultatul echivelent. AND, OR, XOR si XNOR primesc 2 operanzi si genereaza rezultatul operatiei. In cazul operatiilor de shiftare se primesc 2 operanzi si rezultatul este de forma primul operant shiftat stanga/dreapta cu operand 2 pozitii.

4.4 Algoritmi implementati

In cazul impartirii prin refacerea restului partial in fiecare etapa a operatiei se efectueaza o scadere a impartitorului din restul partial. Daca rezultatul este negative, deci restul partial mai mic decat impartitorul se reface restul partial la valoarea anterioara prin adunarea impartitorului la restul partial. Daca se obtine un numar pozitiv cifra corespunzatoare catului este 1. Procesul se repeat de n ori daca se doreste obtinerea unui cat de n cifre. Procesul de impartire se opreste daca restul partial devine zero. Mai multe detalii despre pasii acestui algoritm se deduc din organigrama prezenta in anexa C.

4.5 Detalii de implementare

Unitatea aritmetica si logica implementata are ca porturi de I/O:
Opcode – intrare, 4 biti: semnal de comanda
X – intrare, n biti: primul operand
Y – intrare, n biti: al doilea operand
Clk – intrare, 1 bit: semnal de clock
Rst – intrare, 1 bit: semnal de reset
Start – intrare, 1 bit: semnal de start pentru impartire
Term – iesire, 1 bit: semnalizeaza terminarea impartirii
Rez – iesire, 2n biti: rezultatul operatiei

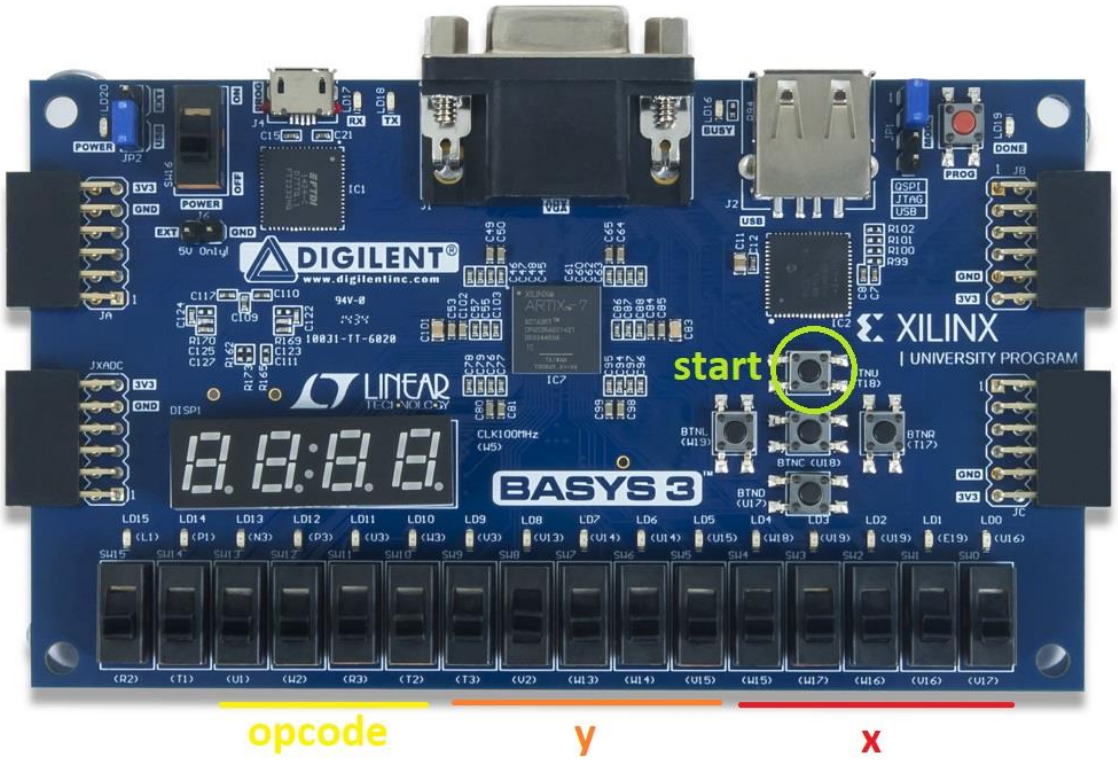
Semnalul de comanda opcode seleteaza ce operatie se efectueaza astfel:
0000 = adunare
0001 = scadere
0010 = inmultire
0011 = impartire
0100 = SI
0101 = SAU
0110 = SAU EXCLUSIV
0111 = COINCIDENTA
1000 = shiftare la stanga

1001 = shiftare la dreapta
1010 = complement fata de doi

Unitatea de control este simbolizata prin procesul “control” din cadrul sursei alu.vhd din anexa D, in care se selecteaza operatia dorita conform opcode-ului primit. Sursele celorlalte operatii sunt specificate in anexa D, fiind vizibile toate detaliile de implementare.

4.6 Manual de utilizare

In cazul testarii pe placa se utilizeaza modulul modul_alu.vhd din anexa D care are in plus conectarea iesirilor la un 7 Segment Display pentru vizualizarea rezultatelor. In urma realizarii sintezei, implementarii, generarea bitstreamului si programarea placii, functionarea UAL poate fi testat pe FPGA-ul dorit. In cazul meu deoarece detin un Basys 3 am ales sa testez operatii cu numere pe 5 biti, fiind limitata de numarul de switch-uri de pe placa. Astfel, primele 5 switchuri corespund primului operand, urmatoarele 5 celui de al doilea si urmatorii 4 opcode-ului. Daca se doreste efectuarea operatiei de impartire pentru incepere trebuie apasat butonul T18. Odata aleasa operatia si operanzii rezultatul va fi disponibil pe display.



Placa utilizata Figura 5

5. Rezultate experimentale

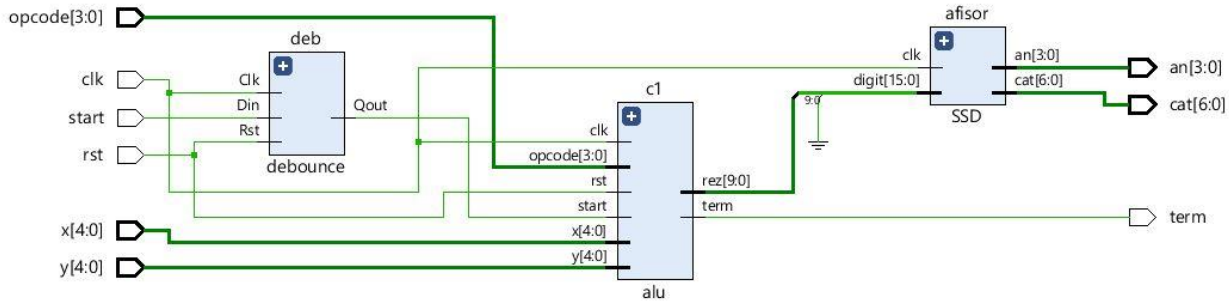
5.1 Instrumente de proiectare utilizate

Limbajul de descriere hardware utilizat este VHDL. Pentru sinteza, analiza si simularea proiectului am utilizat mediul Vivado Design Suite 2020.1 descarcat pe un PC cu sistem de operare Windows 10.

5.2 Circuitul utilizat pentru implementare

FPGA-ul utilizat in testare este Basys 3 Xilinx Artix-7 (XC7A35T-1CPG236C) care contine 33280 celule logice in 5200 de slice-uri(fiecare slice continue patru 6-input LUT si 8 bistabile), 90 slice-uri DSP si 1800 Kbits fast block RAM. Viteza clock-ului intern este de 450 MHz

Circuitul rezultat in urma implementarii este prezentat in figura urmatoare:



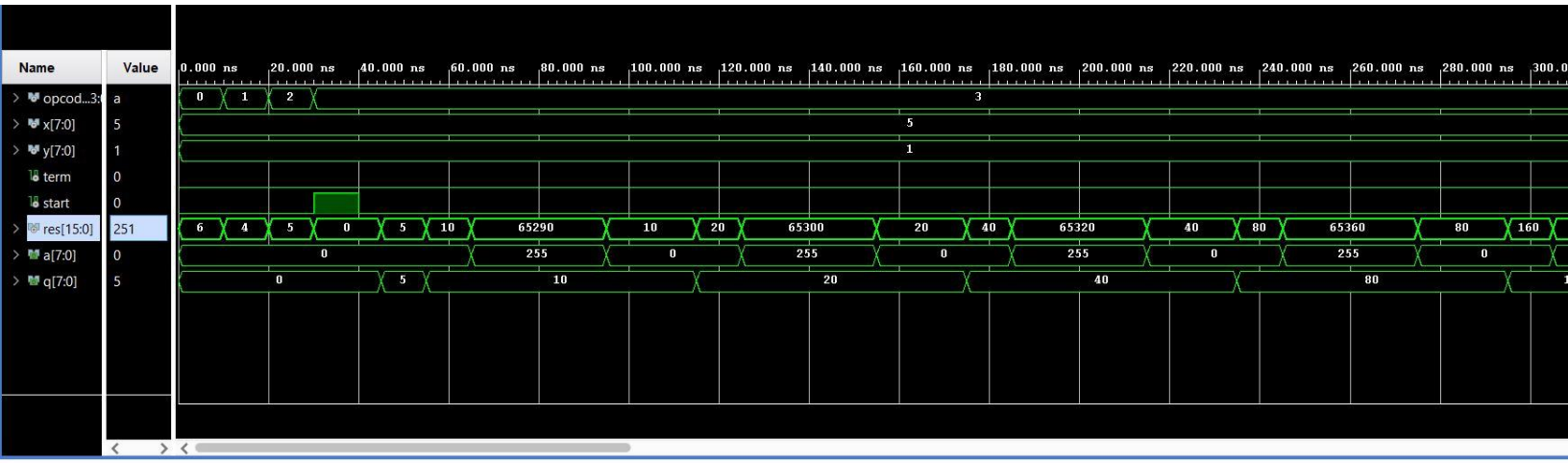
Circuit schematic Figura 6

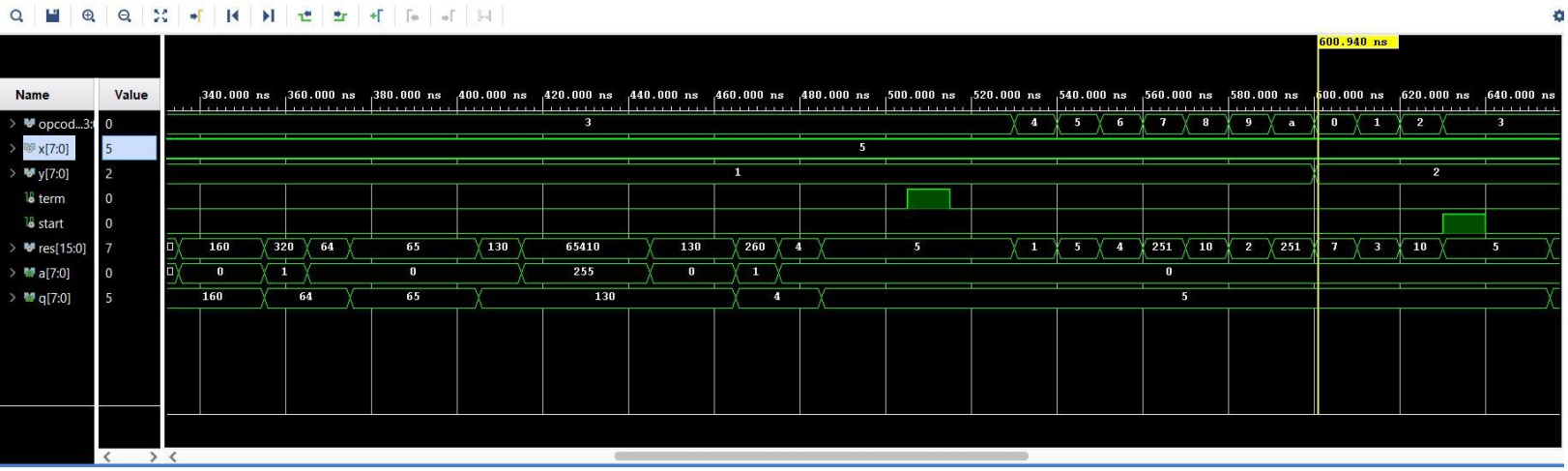
5.3 Procedura de test utilizata

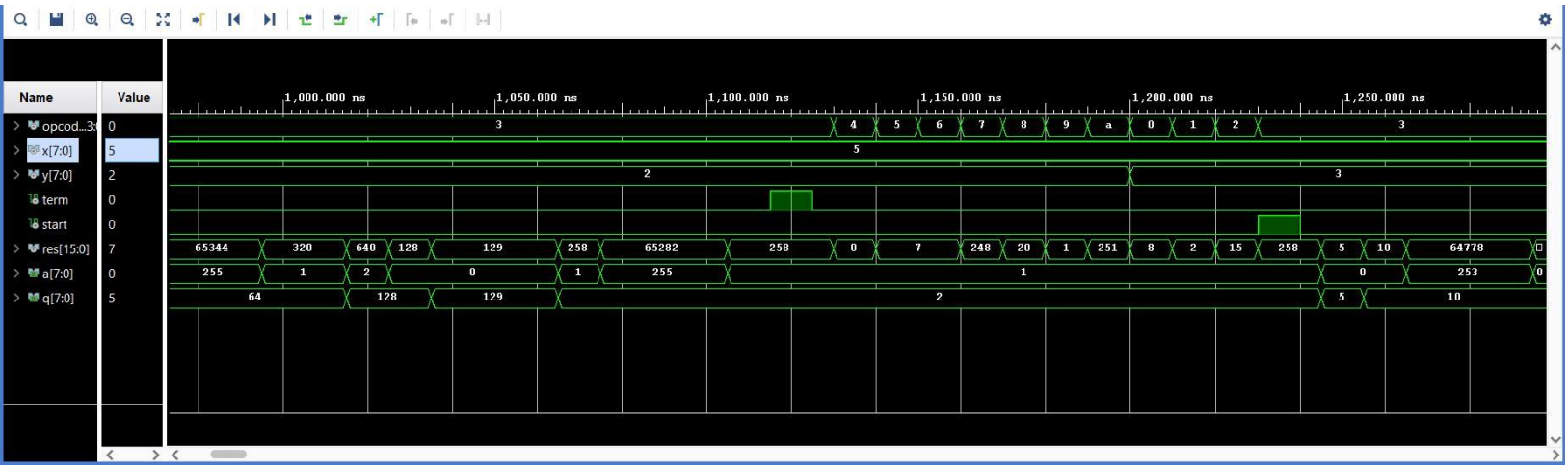
Unitatea aritmetica si logica propusa a fost testata atat prin simulare in mediul Vivado cat si pe placa. Simularea a fost facuta cu operanzi de 8 biti si au fost testate valori apropiate de valorile minime de reprezentare (ca exemplu 5 si 1, 2, 3), medii (ca exemplu 150 si 63, 64, 65) si apropiate de valorile maxime (ca exemplu 255 si 251, 252, 253). Pentru fiecare set de operanzi au fost realizate toate operatiile implementate conform codului din anexa E.
Testarea pe placa a fost facuta conform video-ului pus la adresa https://drive.google.com/file/d/1d2W6ZaJyau_MlPFcuYznyQkJhSLbJaKs/view?usp=sharing

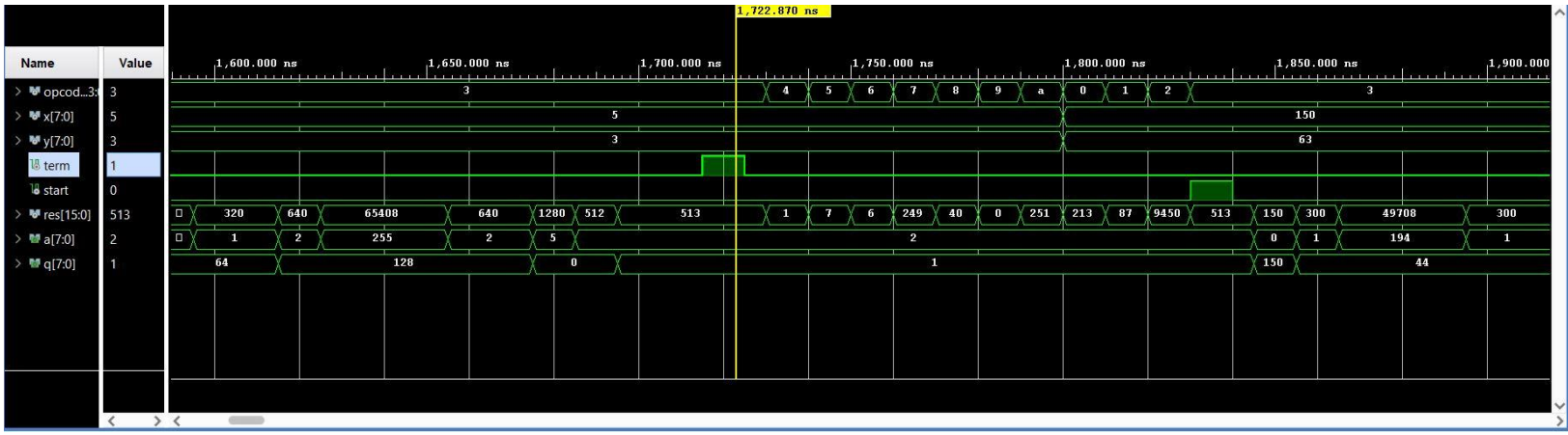
5.4 Rezultatele simularii

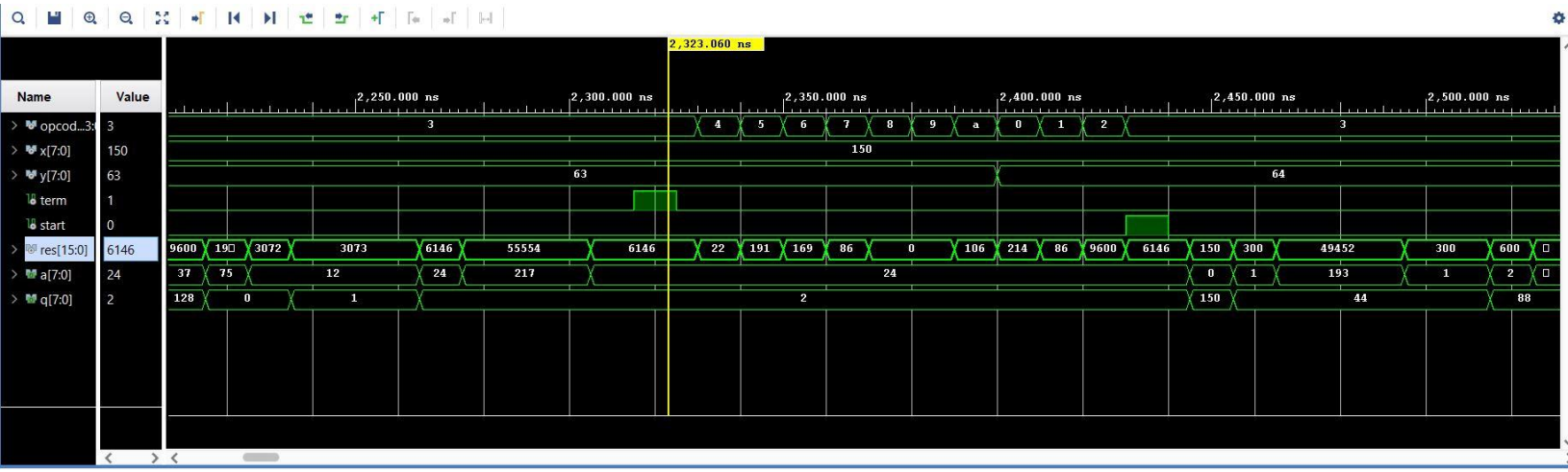
In cadrul capturilor de ecran urmatoare se pot observa operanzii, rezultatul, precum si semnalul A si Q din cadrul impartirii pentru fiecare tip de operatie pe valorile mentionate anterior.

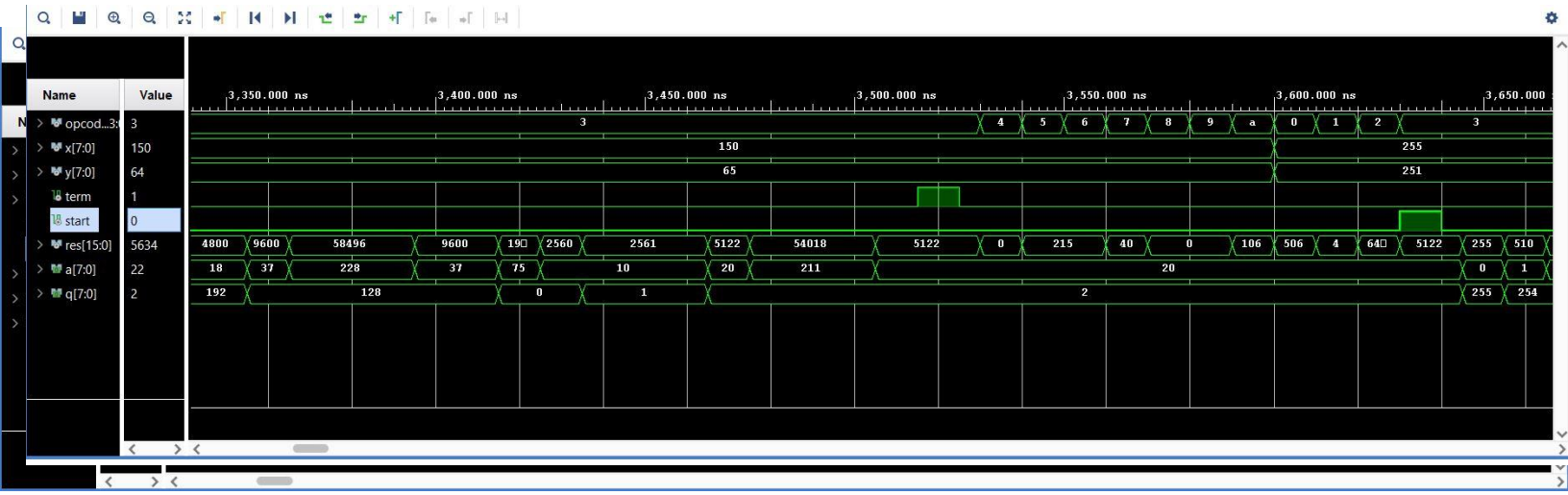


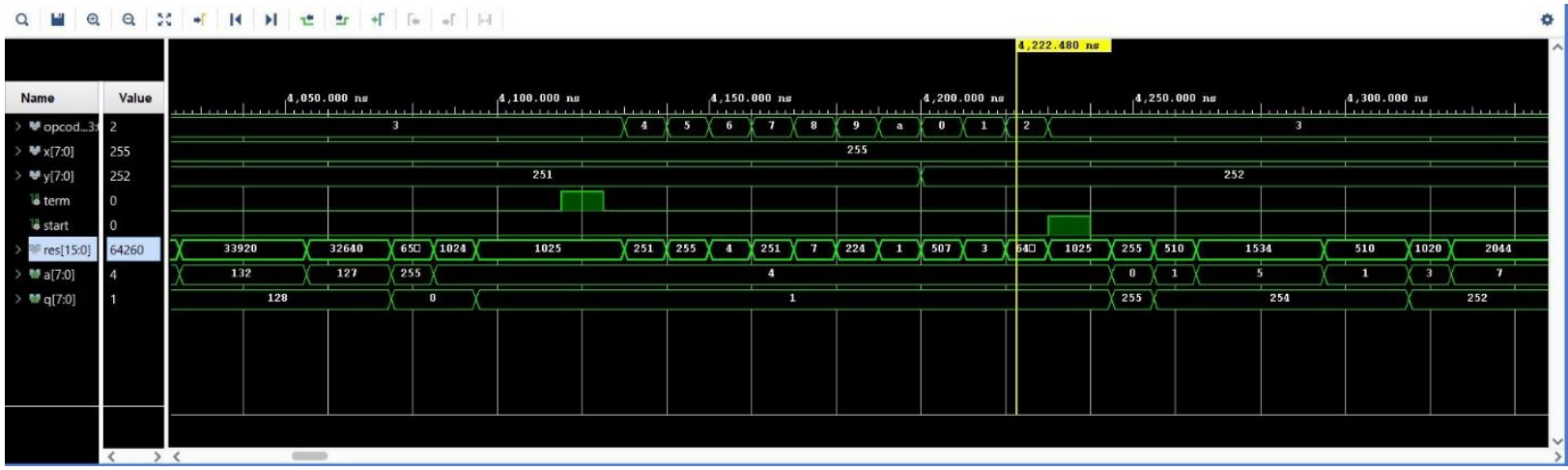


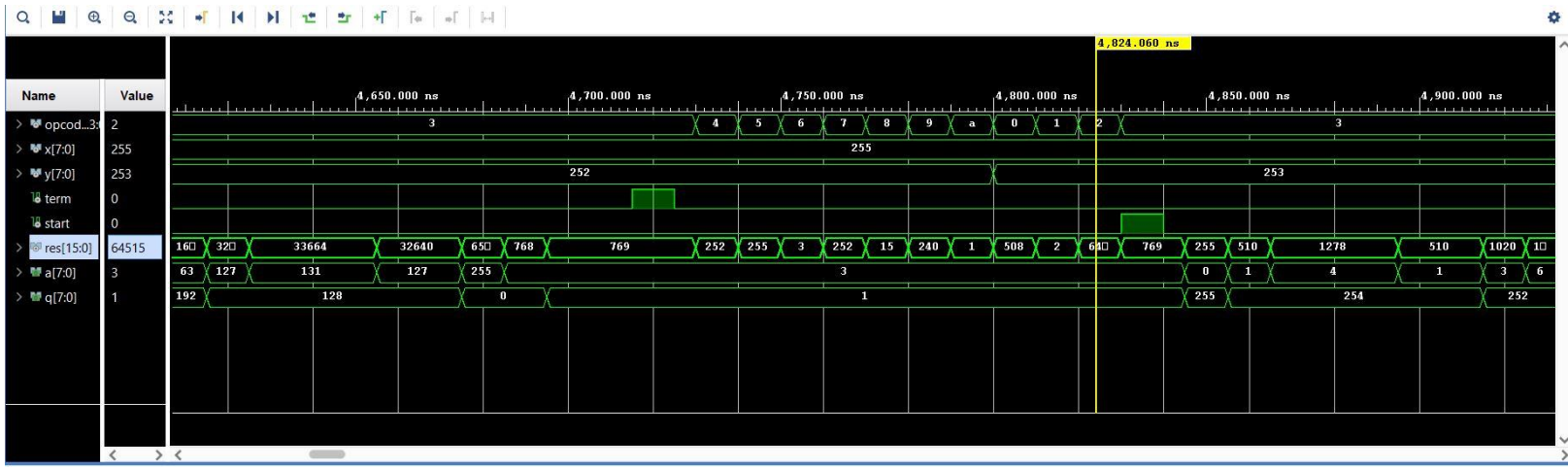

Name
Value
opcod...
x[7:0]
y[7:0]
term
start
res[15:0]
a[7:0]
q[7:0]
1,000.000 ns
1,050.000 ns
1,100.000 ns
1,150.000 ns
1,200.000 ns
1,250.000 ns

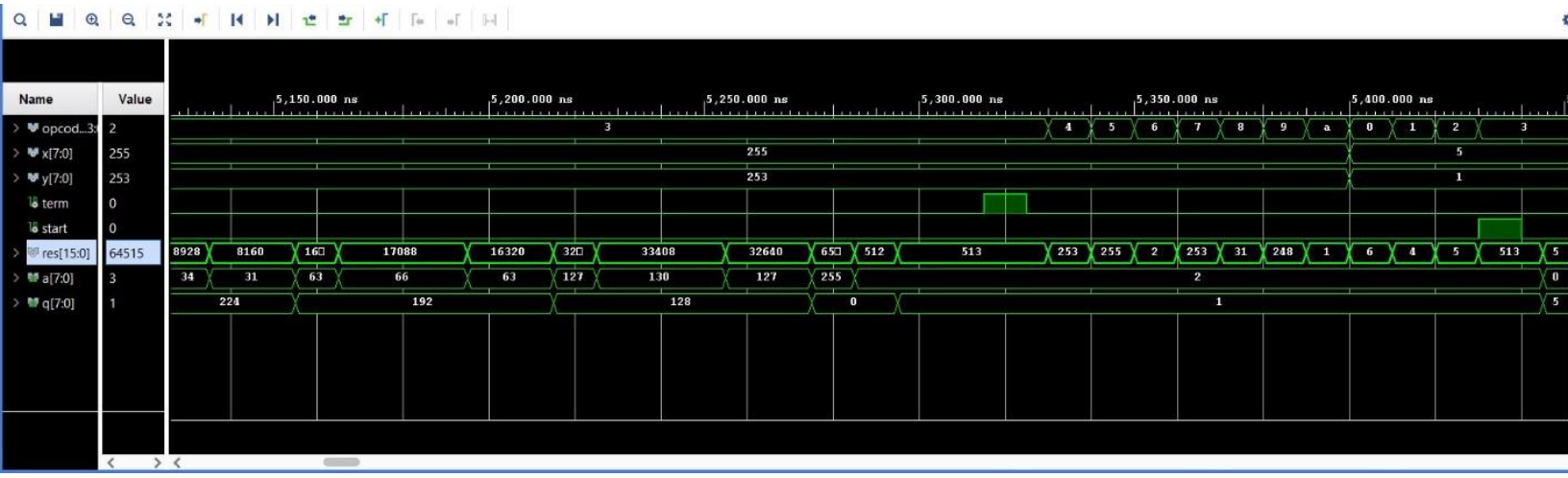

Name
Value
opcod...
x[7:0]
y[7:0]
term
start
res[15:0]
a[7:0]
q[7:0]
1,722.870 ns
1,600.000 ns
1,650.000 ns
1,700.000 ns
1,750.000 ns
1,800.000 ns
1,850.000 ns
1,900.000


Name
Value
opcod...
x[7:0]
y[7:0]
term
start
res[15:0]
a[7:0]
q[7:0]
2,323.060 ns
2,250.000 ns
2,300.000 ns
2,350.000 ns
2,400.000 ns
2,450.000 ns
2,500.000 ns


Name
Value
opcod...3:
x[7:0]
y[7:0]
term
start
res[15:0]
a[7:0]
q[7:0]
3
150
64
1
0
5634
22
2
3,350.000 ns
3,400.000 ns
3,450.000 ns
3,500.000 ns
3,550.000 ns
3,600.000 ns
3,650.000 ns


Name
Value
opcod...3:
x[7:0]
y[7:0]
term
start
res[15:0]
a[7:0]
q[7:0]
2
255
252
0
0
64260
4
1
4,222.480 ns
4,050.000 ns
4,100.000 ns
4,150.000 ns
4,200.000 ns
4,250.000 ns
4,300.000 ns





6. Concluzii

Asadar, acest proiect a solutionat implementarea unei UAL capabile sa efectueze atat operatii aritmetice cat si logice pe operanzi generici in reprezentare binara, fara semn. Avantajul consta in alegerea modului in care implementam operatiile, majoritatea fiind combinationale, exceptand impartirea. Ca dezavantaj se poate preciza limitarea UAL la reprezentare binara, aceasta putand fi adoptata si pentru numere cu mantisa si semn sau in reprezentare zecimala. Proiectul se poate simula si pentru numere de dimensiune diferita fata ce cele prezentate, prin schimbarea genericului n. Ca dezvoltari ulterioare se pot adauga mai multe operatii precum ridicare la putere, rotiri sau shiftari pe un numar predefinit de biti.

Conclusiv, pot afirma ca actualul proiect mi-a oferit oportunitatea de a sedimenta mult mai bine cunostintele dobandite de-a lungul semestrului la materia SSC si mi-a oferit ocazia de a aprofunda scrierea in limbajul VHDL.

Bibliografie

[1] https://steemit.com/technology/@kiddady/summary-of-the-arithmetic-logic-unit-or-the-enhancement-of-the-alu-subsystem
[2] Electronics explained-The new systems approach to learning electronics 2010, Chapter 6: How Microcomputers Work: The Brains of every Electronic Product Today
[3] https://users.utcluj.ro/~baruch/ro/pages/cursuri/structura-sistemelor-de-calcul/curs.php

Anexe

Anexa A
Schema de detaliu a sumatorului cu propagarea transportului

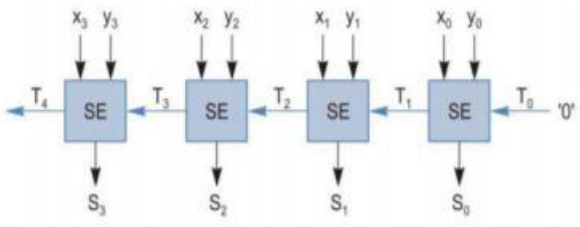


Figura 4.1. Schema bloc a unui sumator de patru biți cu propagarea succesivă a transportului.

Anexa B

Schema de detaliu a inmultirii matriceale

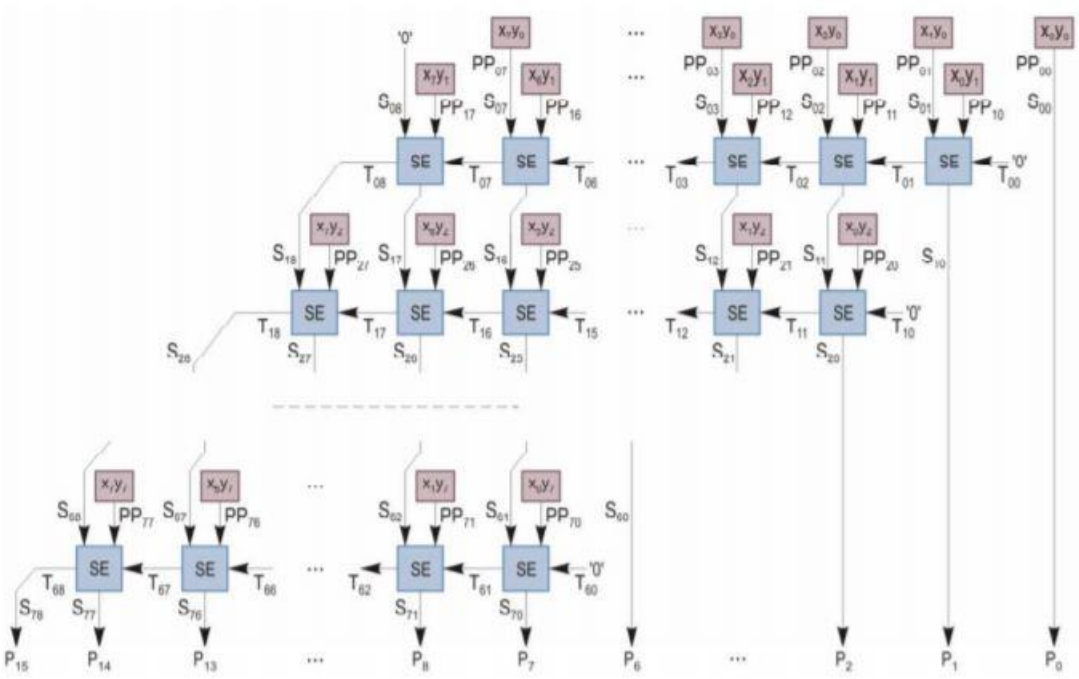


Figura 4.15. Denumirea semnalelor utilizate de circuitul de înmulțire matriceală pentru numere de câte opt biți.

Anexa C

Schema de detaliu a impartirii cu refacerea restului si organigrama

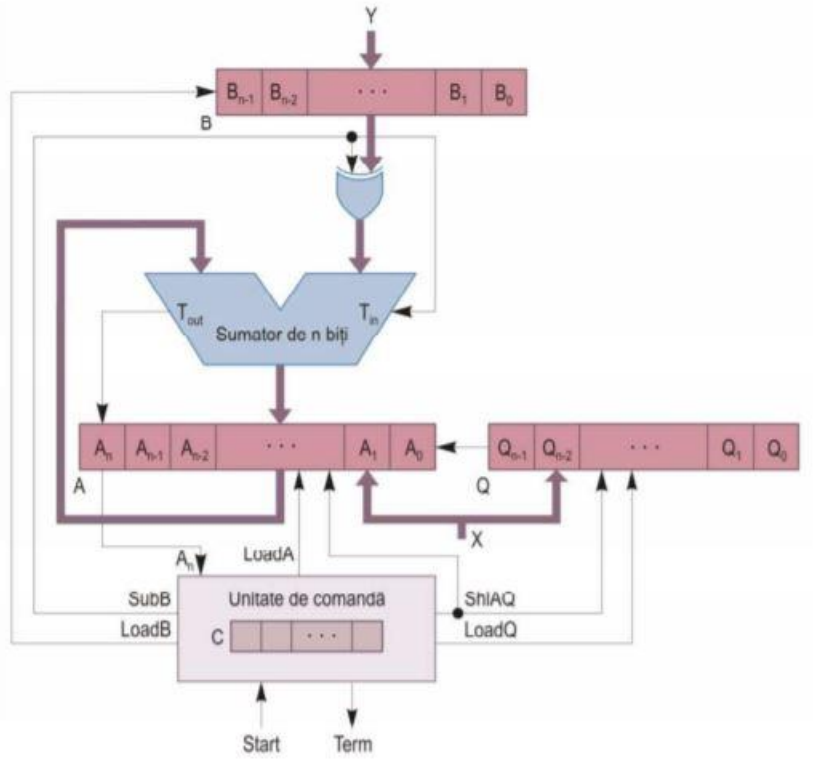


Figura 5.5. Schema bloc a unui circuit de împărțire prin metoda refacerii restului parțial pentru numere fără semn.

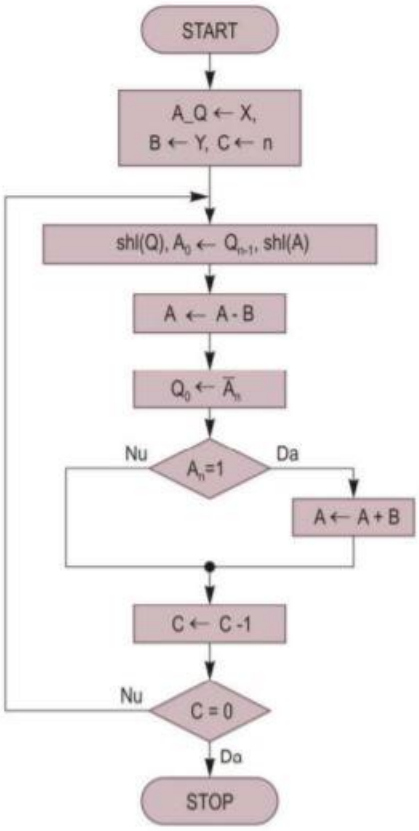


Figura 5.6. Organigrama operației de împărțire prin metoda refacerii restului parțial pentru numere fără semn.

Anexa D
Codul sursa al operatiilor aritmetice si UAL

Modul ALU

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

entity modul_alu is
 Port ( opcode: in std_logic_vector(3 downto 0);
 x, y: in std_logic_vector(4 downto 0);
 clk, rst, start: in std_logic;
 term: out std_logic;
 cat: out std_logic_vector(6 downto 0);
 an: out std_logic_vector(3 downto 0));
end modul_alu;
```

```
architecture Behavioral of modul_alu is
signal res: std_logic_vector(9 downto 0);
signal sig: std_logic_vector(15 downto 0);
signal auxStart: std_logic;

begin
afisor: entity WORK.SSD port map(clk, sig, cat, an);
deb: entity WORK.debounce port map(clk, Rst, start, auxStart);
c1: entity WORK.alu generic map (n => 5)  port map(opcode, x, y, clk, rst, auxStart,
term, res);

sig <= "000000" & res;

end Behavioral;
```

ALU

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.ALL;
entity alu is
generic( n: natural);
  Port ( opcode: in std_logic_vector(3 downto 0);
 x, y: in std_logic_vector(n-1 downto 0);
 clk, rst, start: in std_logic;
 term: out std_logic;
 rez: out  std_logic_vector(2*n-1 downto 0));
end alu;

architecture Behavioral of alu is

signal sum,dif, compl, a, q, zero, andRes, orRes, xorRes, xnorRes,shlRes, shrRes  :
std_logic_vector(n-1 downto 0) := (others => '0');
signal p: std_logic_vector(2*n-1 downto 0) := (others => '0');
signal tout: std_logic := '0';

begin

adunare: entity WORK.adunare  generic map (n => n) port map('0', x, y, sum, tout);
```

```
scadere: entity WORK.scadere  generic map (n => n) port map(x, y, dif);
produs: entity WORK.InmultireMatriceala generic map (n => n) port map(x, y, p);
impartire: entity WORK.impartire generic map (n => n) port map(clk, rst, start, x, y, a ,q,
term);
complement: entity WORK.complement2 generic map (n => n) port map(x, compl);
si: entity WORK.andLogic generic map (n => n) port map(x,y, andRes);
sau: entity WORK.orLogic generic map (n => n) port map(x,y, orRes);
sau_exclusiv: entity WORK.xorLogic generic map (n => n) port map(x,y, xorRes);
coincidenta: entity WORK.xnorLogic generic map (n => n) port map(x,y, xnorRes);
shiftLeft: entity WORK.shiftLeftLogic generic map (n => n) port map(x,y, shlRes);
shiftRight: entity WORK.shiftRightLogic generic map (n => n) port map(x,y, shrRes);

control:process (opcode, sum, dif, p, a, q, compl, andRes, orRes, xorRes, xnorRes,
shlRes, shrRes)
begin
case opcode is
                        when "0000" => rez <= zero(n-1 downto 1)& tout & sum;
                        when "0001" => rez <= zero & dif;
                        when "0010" => rez <= p;
                        when "0011" => rez <= a & q;
                        when "0100" => rez <= zero & andRes;
                        when "0101" => rez <= zero & orRes;
                        when "0110" => rez <= zero & xorRes;
                        when "0111" => rez <= zero & xnorRes;
                        when "1000" => rez <=  zero & shlRes;
                        when "1001" => rez <= zero & shrRes;
                        when "1010" => rez <= zero & compl;
                        when others => rez <= zero & zero;

end case;
end process;
end Behavioral;
```

Operatia de adunare

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

entity adunare is
generic( n: natural);
  Port ( tin: in std_logic;
  x, y: in std_logic_vector(n-1 downto 0);
  s: out  std_logic_vector(n-1 downto 0);
  tout: out std_logic);
end adunare;

architecture Behavioral of adunare is

component SumatorElementar is
  Port ( x, y, tin: in std_logic;
  s, tout: out std_logic );
end component;

signal aux: std_logic_vector(n downto 0) := (others => '0');
begin
aux(0) <= tin;
tout <= aux(n);

Suma: for i in 0 to n-1 generate
        c: SumatorElementar port map(x(i), y(i), aux(i),s(i), aux(i+1));
end generate;
end Behavioral;
```

Operatia de scadere

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity scadere is
generic( n: natural);
  Port (
 x, y: in std_logic_vector(n-1 downto 0);
 s: out  std_logic_vector(n-1 downto 0));
end scadere;
```

```
architecture Behavioral of scadere is
component complement2 is
generic( n: natural);
  Port (
  x: in std_logic_vector(n-1 downto 0);
  y: out  std_logic_vector(n-1 downto 0));
end component;

component adunare is
generic( n: natural);
  Port ( tin: in std_logic;
  x, y: in std_logic_vector(n-1 downto 0);
  s: out  std_logic_vector(n-1 downto 0);
  tout: out std_logic);
end component;

signal auxy: std_logic_vector(n-1 downto 0);
signal tout: std_logic;
begin

c1: component complement2 generic map (n => n) port map(y, auxy);
c2: component adunare generic map (n => n) port map('0', x, auxy, s, tout);


end Behavioral;
```

Operatia de inmultire

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity InmultireMatriceala is
  generic( n: natural);
  Port (x, y: in std_logic_vector(n-1 downto 0);
  p: out std_logic_vector(2*n-1 downto 0));
end InmultireMatriceala;

architecture Behavioral of InmultireMatriceala is
component SumatorElementar is
  Port ( x, y, tin: in std_logic;
```

```
  s, tout: out std_logic );
end component;

type matrixnxn is array ( n-1 downto 0 ) of std_logic_vector( n-1 downto 0);
type matrixnxnp1 is array ( n-1 downto 0 ) of std_logic_vector( n downto 0);
type matrixn_1xnp1 is array ( n-2 downto 0 ) of std_logic_vector( n downto 0);
signal pp: matrixnxn:= (others =>  ( others => '0'));
signal s: matrixnxnp1:= (others =>  ( others => '0'));
signal t: matrixn_1xnp1:= (others =>  ( others => '0'));

begin


PPx: for i in 0 to n-1 generate
      PPy: for j in 0 to n-1 generate
          pp(i)(j) <= x(j) and y(i);
       end generate;
end generate;

s(0)(n) <= '0';

S0: for i in 0 to n-1 generate
          s(0)(i) <= x(i) and y(0);
end generate;

t0: for i in 0 to n-2 generate
          t(i)(0) <= '0';
end generate;

t8: for i in 0 to n-2 generate
          s(i+1)(n) <= t(i)(n);
end generate;

l0: for j in 0 to n-2 generate
       l1: for i in 0 to n-1 generate
          l2: SumatorElementar port map(s(j)(i+1),pp(j+1)(i), t(j)(i), s(j+1)(i), t(j)(i+1));
       end generate;
end generate;

p0: for i in 0 to n-1 generate
```

```
        p(i) <= s(i)(0);
end generate;

p1: for i in n to 2*n-1 generate
        p(i) <= s(n-1)(i-n+1);
end generate;
```

Operatia de impartire

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

entity impartire is
  generic( n: natural);
  Port ( clk, rst, start: in std_logic;
  x, y: in std_logic_vector(n-1 downto 0);
  a, q: out  std_logic_vector(n-1 downto 0);
  term: out std_logic);
end impartire;

architecture Behavioral of impartire is

signal ldA, ldB, ldQ,ldQi, subB, shrAQ, rstA, rstQ: std_logic;
signal tout, ovf, An, q0, loadQ: std_logic;
signal sigB, sigQ, sum, auxB, auxQ: std_logic_vector(n-1 downto 0) := (others => '0');
signal sigA,auxA, zero: std_logic_vector(n downto 0) := (others => '0');
begin

a <= sigA(n-1 downto 0);
q <= sigQ;
an <= tout;
q0 <= not(An);

regB: entity work.FDN generic map (n => n)
     port map(clk, '0', ldB, y, sigB);

xorB: for i in 0 to n-1 generate
   auxB(i) <= subB xor sigB(i);
end generate;
```

```
--comp: entity WORK.complement2 generic map (n => n) port map(sigB, auxB);

sumator: entity work.ADDN generic map (n => n)
   port map(subB, auxB, sigA(n-1 downto 0), sum, tout, ovf);

auxA <= tout & sum;

shiftA: entity work.SLLN generic map (n => n+1)
   port map(clk, rstA, shrAQ, sigQ(n-1), ldA, auxA, sigA);

shiftQ: entity work.SLLN generic map (n => n)
   port map(clk, rstQ, shrAQ, '0', loadQ, auxQ, sigQ);

auxQ <= sigQ(n-1 downto 1) & q0 when ldQi = '0' else x;
loadQ <= ldQi or ldQ;

comanda: entity work.comanda_impartire generic map (n => n)
   port map(clk, rst, start, An, term, ldA, ldB, ldQ, subB, shrAQ, ldQi, rstA, rstQ);
end Behavioral;
```

Anexa E
Simularea UAL

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

entity alu_sim is
--  Port ( );
end alu_sim;

architecture Behavioral of alu_sim is

signal x, y: std_logic_vector(7 downto 0);
signal res: std_logic_vector(15 downto 0);
signal opcode: std_logic_vector(3 downto 0);
signal clk, rst, start, term: std_logic;

begin
```

```
c1: entity WORK.alu generic map (n => 8)  port map(opcode, x, y, clk, rst,start, term,
res);

gen_clk: process
begin
   clk <= '0';
   wait for 5 ns;
   clk <= '1';
   wait for 5 ns;
end process gen_clk;

rst <= '1','0' after 1 ns;
--start <= '1', '0' after 60 ns;
gen_start: process
begin
   start <= '0';
   wait for 30 ns;
   start <= '1';
   wait for 10 ns;
   start <= '0';
   wait for 560 ns;

end process gen_start;

--simulare pentru valori mici
process
begin

x <= std_logic_vector(conv_std_logic_vector(5,8));
for j in 1 to 3 loop
   y <= std_logic_vector(conv_std_logic_vector(j,8));

   opcode <= "0000";
   wait for 10 ns;

   opcode <= "0001";
   wait for 10 ns;
```

```
    opcode <= "0010";
    wait for 10 ns;

    opcode <= "0011";
    wait for 500 ns;

    opcode <= "0100";
    wait for 10 ns;

    opcode <= "0101";
    wait for 10 ns;

    opcode <= "0110";
    wait for 10 ns;

    opcode <= "0111";
    wait for 10 ns;

    opcode <= "1000";
    wait for 10 ns;

    opcode <= "1001";
    wait for 10 ns;

    opcode <= "1010";
    wait for 10 ns;
end loop;


--simulare pentru valori intermediare


x <= std_logic_vector(conv_std_logic_vector(150,8));
for j in 63 to 65 loop
    y <= std_logic_vector(conv_std_logic_vector(j,8));

    opcode <= "0000";
    wait for 10 ns;

    opcode <= "0001";
```

```
    wait for 10 ns;

    opcode <= "0010";
    wait for 10 ns;

    opcode <= "0011";
    wait for 500 ns;

    opcode <= "0100";
    wait for 10 ns;

    opcode <= "0101";
    wait for 10 ns;

    opcode <= "0110";
    wait for 10 ns;

    opcode <= "0111";
    wait for 10 ns;

    opcode <= "1000";
    wait for 10 ns;

    opcode <= "1001";
    wait for 10 ns;

    opcode <= "1010";
    wait for 10 ns;
end loop;

--simulare pentru valori mari

x <= std_logic_vector(conv_std_logic_vector(255,8));
for j in 251 to 253 loop
    y <= std_logic_vector(conv_std_logic_vector(j,8));

    opcode <= "0000";
    wait for 10 ns;

    opcode <= "0001";
```

```
    wait for 10 ns;

    opcode <= "0010";
    wait for 10 ns;

    opcode <= "0011";
    wait for 500 ns;

    opcode <= "0100";
    wait for 10 ns;

    opcode <= "0101";
    wait for 10 ns;

    opcode <= "0110";
    wait for 10 ns;

    opcode <= "0111";
    wait for 10 ns;

    opcode <= "1000";
    wait for 10 ns;

    opcode <= "1001";
    wait for 10 ns;

    opcode <= "1010";
    wait for 10 ns;
end loop;
end process;

end Behavioral;
```