

**Facultatea de Automatică și calculatoare**

**Specializarea: Calculatoare și tehnologia informației**

**Disciplina: Proiectarea dispozitivelor numerice**

**CALCULATOR DE BUZUNAR**

Profesor îndrumător : Realizator:

**Turicu Dan Baluș Dan**

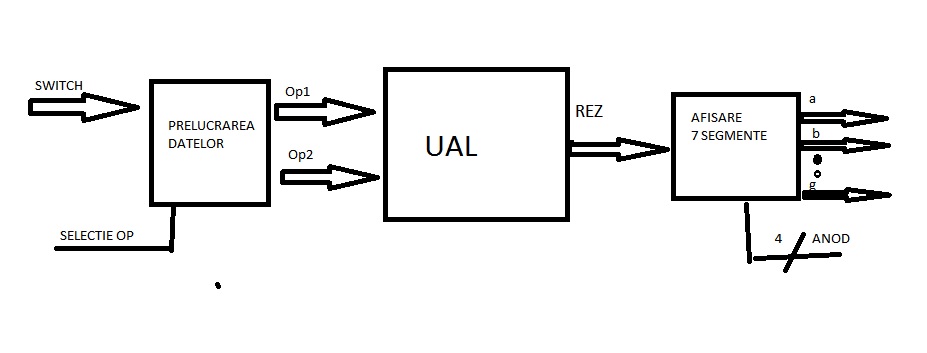
Grupa 30217

**1.Specificația proiectului**

Tema proiectului:

Enunțul: Să se proiecteze un calculator de buzunar cu operații aritmetice fundamentale. Se vor folosi cele patru afișaje cu 7 segmente de pe plăcuțele Digilent Nexys-2 și Spartan3E.

Schema bloc:



Schema bloc este alcătuită din 3 componente:

PRELUCRAREA DATELOR, UNITATEA-ARITMETICO-LOGICA și AFISAREA PE 7 SEGMENTE.

Prelucrarea datelor:

Datele de intrare ale automatului sunt 2 operanzi, reprezentați în întregi cu semn. Cele două numere vor fi convertite în complement față de doi, deoarece operațiile fundamentale ( adunare, scădere, inmulțire) vor fi mai ușor de implementat.

Cei 2 operanzi vor fi reprezentați pe 5 biți (primul, cel mai semnificativ, este pentru semn).

Se va folosi doar reprezentarea pozitiva a lui 0 (+0) și anume 0 0000, nu (-0) respectiv 1 0000.

Rezultatul operațiilor de adunare și scădere nu trebuie să depașească cei 4 biți alocați pentru număr ( va fi cuprins între -15 și +15).

Operația de împărțire se va efectua cu urmatoarele restricții:

-primul operand trebuie sa fie mai mare decât al doilea;

- operanzii trebuie să fie pozitivi.

Pentru a selecta operația pe care dorim să o efectuam vom folosi 2 demultiplexoare, al caror selecție va fi aceași:

00->adunare;

01->scădere;

10-> înmulțire;

11->împărțire.

Unitatea aritmetico-logică

Așadar, cei doi operanzi sunt reprezentați in complement față de 2. Legatura dintre cele 2 componente se va face cu cele 2 demultiplexoare, care corespund celor 2 operanzi.

*Adunarea* și *scăderea* se vor implementa cu un sumator, respectiv scăzător, împreună cu biții de semn.

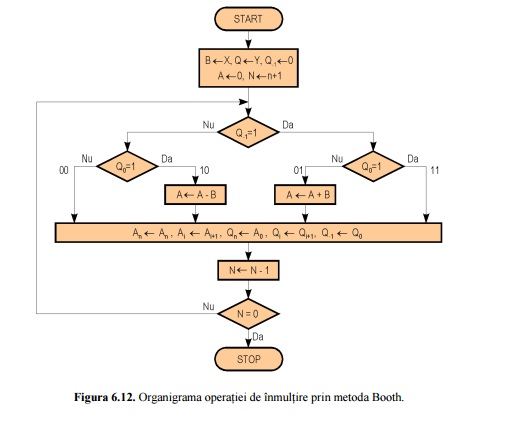
*Înmulțirea* se va efectua cu ajutorul Algoritmului lui Booth:

În aceasta metoda cifrele înmultitorului sunt examinate în perechi, începând cu cea mai putin semnificativa pereche; deînmultitul este adunat sau scazut din produsul partial acumulat, în functie de informatia obtinuta în urma comparatiei bitilor înmultitorului. Algoritmul pentru aceasta metoda poate fi enuntat astfel:

- daca cifrele comparate sunt 00 sau 11 nu se efectueaza nimic si se deplaseaza înmultitorul si produsul partial cu o pozitie spre dreapta;

-daca cifrele comparate sunt 10, se scade deînmulțitul (adica se aduna complementul sau față de doi) din produsul parțial acumulat si se deplaseaza cu o pozitie la dreapta:

-daca cifrele comparate sunt 01, se aduna deînmulțitul la produsul partial acumulat si se deplaseaza cu o pozitie la dreapta.



*Împărțirea* se va realiza prin deplasări la stânga, repetate. Vom afla câtul operației, dar și restul acesteia. Câtul și restul vor fi concatenate în vectorul de biți rezultat.

Rezultatul obținut de una din aceste operații va fi convertit înapoi în întregi cu semn, pentru a fi mai lizibilă afișarea pe plăcuță.

Afișarea pe 7 segmente:

Pentru a putea fi afișat, rezultatul va fi convertit in BCD, folsind algoritmul ”double dabble”. La final procesăm vectorul de intrare și obținem un vector pe 16 biți. Principiul de funcționare: se alege un vector de 16 biți, inițializat cu zero, și valoarea în binar (pe 8 biți) a numărului pe care dorim să-l decodificăm. Se shift-ează câte un bit în stânga, iar în cazul în care avem un număr mai mare decât 4, îl încrimentăm cu 3.

Exemplu:

0000 0000 0000 1111 0011 Inițializare 2432

0000 0000 0001 1110 0110 Șhift-are stânga

0000 0000 0011 1100 1100 Șhift-are stânga

0000 0000 0111 1001 1000 Șhift-are stânga

0000 0000 1010 1001 1000 Adunăm 3 la unități, valoarea trecută -> 7

0000 0001 0101 0011 0000 Șhift-are stânga

0000 0001 1000 0011 0000 Adunăm 3 la unități, valoarea trecută -> 5

0000 0011 0000 0110 0000 Șhift-are stânga

0000 0110 0000 1100 0000 Șhift-are stânga

0000 1001 0000 1100 0000 Adunăm 3 la zeci, valoarea trecută -> 6

0001 0010 0001 1000 0000 Șhift-are stânga

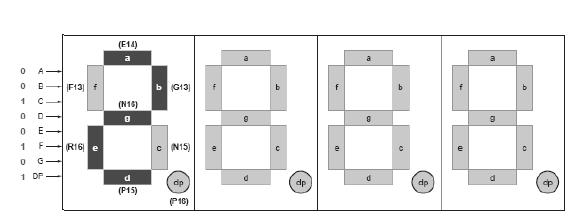
0010 0100 0011 0000 0000 Șhift-are stânga

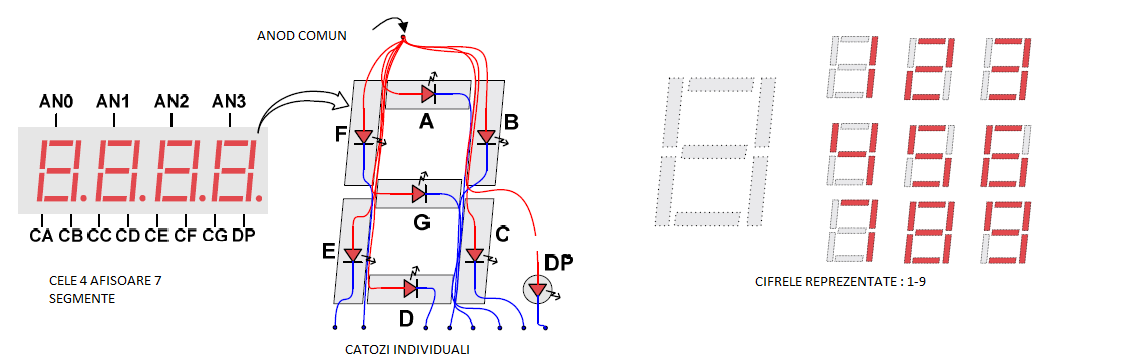
BCD 2 4 3

Afișarea pe 7 segmente:

Toate cele 4 afisoare sunt legate la un anod comun, ceea ce înseamna că pe fiecare afișor se va afișa același lucru. Pentru a afișa informații diferite trebuie sa activam pe rand câte un anod, în timp ce ceilalți se dezactivează. Acest lucre va trebui făcut cu ajutorul unui multiplexor al cărei selecție se va schimba foarte rapid, ca ochiul uman să nu perceapă momentul în care afișorul se stinge. Cu ajutorul selecției se schimbă și anodul, care initial e “1111”, prin conversia în număr întreg și schimbarea poziției **s** a vectorului vectorului, din 1 în 0 sau invers.

OBS: Dacă rezultatul va fi negativ, o să se aprind al doilea led (de la dreapta la stănga)





* afișoarele au cele 7 leduri reprezentând segmentele, iar **dp** reprezeintă punctul decimalș
* fiecare are un anod activ pe 0.

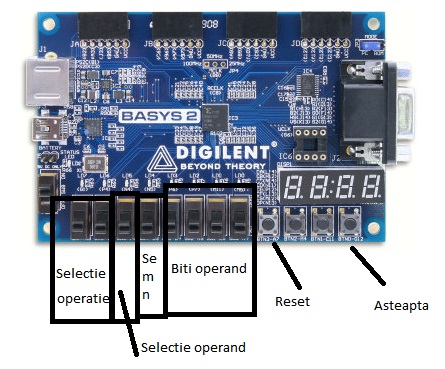
**2.LISTA COMPONENTE**

* Sumator pe 5 biți (pt primul operand);
* Sumator pe 5 biți (pt al doilea operand);
* Scăzător pe 5 biți (pt scădere);
* Demultiplexor 1:4 (pt primul operand);
* Demultiplexor 1:4 (pt al doilea operand);
* 2 x Multiplexor 2:1 (se folosește la tranformarea din întregi cu semn în complement);
* PORȚI;
* Registre de Shift-are pe 10 biți ( se folosesc la înmulțire)ș
* Multiplexoare pe 4 biți;
* Decodificator (pt afișare);
* Numărător( pt afișare);
* Comparatoare pe 4 biți ( se folosesc la conversia din Binar în BCD).

**3.Justificarea soluției alese**

Am ales această soluție deoarece operațiile se efectuaza relativ simplu și este ușor de înțeles. Totodată este și o soluție optimă care permite folosirea operanzilor negativi prin reprezentarea in complement față de 2.

**4.Instrucțiuni de utilizare și întreținere**



* Selectie operand=0;
* Asteapta=0;
* Se trece primul operand (Se selectează semnul acestuia);
* Se selectează selecția operației;
* Asteapta=1; --> acesta va opri primul operand din “drumul” său
* Selectie operand=1;
* Se trece al doilea operand (Se selectează semnul acestuia);
* Reset=1 și Selectie operand=0; --> dacă dorim să introducem din nou primul operand și să efectuăm operație;

5.Listă cu intrări, ieșiri, semnale intermediare, variabile

**Intrări și ieșiri:**

* a (primul operand în întregi);
* aB(al doilea operand în întregi);
* zmux(operandul 1 în complement);
* zmuxB(operandul 2 în complement);
* Y0 (ieșirea 0 a demultiplexorului care se activează la adunare pt operandul 1);
* Y1 (ieșirea1 a demultiplexorului care se activează la scădere pt operandul 1);
* Y2 (ieșirea 2 a demultiplexorului care se activează la înmulțire pt operandul 1);
* Y3 (ieșirea3 a demultiplexorului care se activează la împărțire pt operandul 1);
* Y0B, Y1B, Y2B, Y3B( ieșirile pt operandul 2);
* Sel(selecția pt selectarea operației);
* Op1(este zmux în port map);
* Op2(este zmuxB în port map)
* aUAL( este a în port map);
* aUALB(este aB în port map);
* rezultat(cel ce va aduna rezultatul operației selectate);
* rezultati ( rezultatul înmulțirii, care va fi, apoi, asignat lui rezultat);
* rezultatimp (rezultatul împărțirii, care va fi, apoi, asignat lui rezultat= cat&rest);
* clk(clock);
* Z( rezultatul înmulțirii in fișierul mama);
* Catrest(cat&rest);
* Rest(restul împărțirii)ș
* Cat(câtul)ș
* binIN(șirul de biți veniți de la ieșirea numărului în complement):
* nr, astept, switch se folosesc pentru a funcționa pe plăcuță;
* thousands, jundreds, tems, ones(mii, sute, zeci, unități-vor fi afișate pe afișoare-);
* Clr(clear pt afișare);
* A\_to\_go( “funcția” care va afișa pe afișoare, sub formă de leduri);
* Digit(numărul convertit in BCD, împărțit in 4, afișat în cele 4 afișoare);

**Variabile:**

* ssB (semnul operandului 2);
* ss(semnul operandului 1);
* semn(semnul operandului rezultat după operații);
* temp( se folosește pt stocarea șirului inițial primit-binIN- și pt shift-are);
* bcd(vectorul unde va fi stocat codul binar convertit in BCD);
* quotient(câtul-variabilă ce va fi asignată lui cat-):
* remainder(Restul-variabilă ce va fi asignată restului);

**Semnale**

* P1, P1\_SHIFT...........P6,P6\_SHIFT( ajută la shift-area stânga a numerelor în complement)ș
* A(X&”000000”);
* S((not x+”00001”)&”000000”);
* P(“00000”& Y& ‘0’);
* s(selecția care va selecta care dgiit va fi scris);
* aen(catodul, dat sub formă de vector, fiecare 1 “corespunde” unui digit);
* Ximp(zmux în port map)ș
* Yimp(zmuxB în port map);

**6.Posibilități de dezvoltare:**

* extinderea proiectului pe mai mulți biți;
* operanzii, pentru operația de împărțire, sa poata fi și negativi;
* adăugarea de operații suplimentare (radical, logaritm, exponențială, sinus, cosinus etc.);
* efectuarearea operațiilor cu paranteze;
* efectuarea operațiilor cu numere raționale.

**Intructiuni de utilizare**

**Active HDL**

In Active HDL proiectul se foloseste astfel:

**1**. Se deschide programul Active HDL

**2**. Se compileaza cu ajutorul comenzii “Compile” 

**3.** Se simuleaza cu “New waveform”:

**a).** Se face click dreapta pe fereastra nou deschisa

**b).** Se selecteaza “Add signals…”

**c).** Adaugam din lista toate intrarile, iesirile si semnalele intermediare

**d)**. Se dau valori intrarilor facand click dreapta pe lista aparuta si selectand “Simulators” (de acolo se alege “Clock” pentru clk-ul din proiect si “Hotkey” pentru butoane, switch-uri, reset)

**e)**. Dupa ce toate acestea au fost alese se apasa butonul “Run for” din meniu

**XILINX SI IMPLEMENTAREA PE PLACA FPGA**

1. Se deschide programul **XILINX ISE**;

2.Se creeaza un Workspace

3.Se face click pe ”User constraints”  , iar apoi se selectează “Edit constraints”  unde se declară legăturile.

Exemplu din proiect:

NET "clk" loc=B8;

NET "clr" loc=A7;

NET "a\_to\_g(6)" loc=L14;

NET "a\_to\_g(5)" loc=H12;

NET "a\_to\_g(4)" loc=N14;

NET "a\_to\_g(3)" loc=N11;

NET "a\_to\_g(2)" loc=P12;

NET "a\_to\_g(1)" loc=L13;

NET "a\_to\_g(0)" loc=M12;

net "dp" loc=n13;

net "an(0)" loc=F12;

net "an(1)" loc=J12;

net "an(2)" loc=M13;

net "an(3)" loc=K14;

net "sel(1)" loc=N3;

net "sel(0)" loc=E2;

net "switch" loc=F3;

net "nr(4)" loc=G3;

net "nr(3)" loc=B4;

net "nr(2)" loc=K3;

net "nr(1)" loc=L3;

net "nr(0)" loc=P11;

net "semn" loc=P7;

net "astept" loc=G12;

***7.ANEXA COD VHDL:***

**Primul operand(Tansformarea din întregi cu semn în complement față de 2**

\*Se realizeaza acest lucru folosindu-se de semnul operandului; dacă semnul este 0, complementul are aceași formă precum întregul cu semn; dacă semnul este 1, atunci se neagă toți biții, mai puțin bitul de semn, și se adună 1.

Rezultatul este transmis, mai departe, la UAL, unde urmează să se efectueze operațiile aritmetice.

Library IEEE;

use IEEE.STD\_LOGIC\_1164.all;

use IEEE.STD\_Logic\_unsigned.all;

entity Sumatorplusmux is

port( a:inout STD\_LOGIC\_VECTOR(4 downto 0);

zmux: inout STD\_LOGIC\_VECTOR(4 downto 0) --iesire multiplexor la transformare intregi cu semn in complement (inout->pt a fi compatibil cu port map-ul

);

end Sumatorplusmux;

architecture sumatorsimux of Sumatorplusmux is

begin

process(a)

variable temp: STD\_LOGIC\_VECTOR(4 downto 0);

variable ss: STD\_LOGIC;

begin

ss:=a(4); --salvez bitul de semn

if(ss= '0') then

zmux<= a --daca bitul de semn e 0, numarul in complement ramane la fel

elsif(ss='1') then

zmux<=('1'& not a(3)&not a(2)&not a(1)&not a(0))+"00001";

--daca bitul de semn e 1, se aplica transformarea

end if;

end process;

end sumatorsimux;

**Al doilea operand(Tansformarea din întregi cu semn în complement față de 2)**

\*la fel ca la primul operand

Library IEEE;

use IEEE.STD\_LOGIC\_1164.all;

use IEEE.STD\_Logic\_unsigned.all;

entity SumatorplusmuxB is

port( aB:in STD\_LOGIC\_VECTOR(4 downto 0);

zmuxB: inout STD\_LOGIC\_VECTOR(4 downto 0) --iesire multiplexor la transformare intregi cu semn in complement PENTRU OPERANDUL B

);

end SumatorplusmuxB;

architecture sumatorsimuxB of SumatorplusmuxB is

begin

process(aB)

variable tempB: STD\_LOGIC\_VECTOR(4 downto 0);

variable ssB: STD\_LOGIC;

begin

ssB:=aB(4); --salvez bitul de semn PENTRU OPERANDUL B

if(ssB= '0') then

zmuxB<= aB; --daca bitul de semn e 0, numarul in complement ramane la fel

elsif(ssB='1') then

zmuxB<=('1'& not aB(3)&not aB(2)&not aB(1)&not aB(0))+"00001";

--daca bitul de semn e 1, se aplica transformarea

end if;

end process;

end sumatorsimuxB;

**UAL**

\*este fișierul care înmagazinează toate operațiile aritmetice, rezultatul fiind, apoi, trimis în complement\_in\_intreg pentru conversia a doua

Library IEEE;

use IEEE.STD\_LOGIC\_1164.all;

use IEEE.STD\_Logic\_unsigned.all;

entity ual is

port( Y0: inout STD\_LOGIC\_VECTOR(4 downto 0); --| operatia adunare/ scadere/ inmultire/ impartire

Y1: inout STD\_LOGIC\_VECTOR(4 downto 0); --| adunare --> 00

Y2: inout STD\_LOGIC\_VECTOR(4 downto 0); --| scadere --> 01

Y3: inout STD\_LOGIC\_VECTOR(4 downto 0); --| inmultire --> 10

Y0B: inout STD\_LOGIC\_VECTOR(4 downto 0); --| impartire --> 11

Y1B: inout STD\_LOGIC\_VECTOR(4 downto 0); --|

Y2B: inout STD\_LOGIC\_VECTOR(4 downto 0);

--| 4 downto 0 deoarece nu voi mai stoca bitul de semn in vector

Y3B: inout STD\_LOGIC\_VECTOR(4 downto 0);

sel: inout STD\_LOGIC\_VECTOR(1 downto 0);

Op1: inout STD\_LOGIC\_VECTOR(4 downto 0); --| pt PORT MAP (primul operand)

Op2: inout STD\_LOGIC\_VECTOR(4 downto 0); --| pt PORT MAP (al doilea operand)

Opi1: inout STD\_LOGIC\_VECTOR(4 downto 0);

-- pt PORT MAP (primul operand al inmultirii)

Opi2: inout STD\_LOGIC\_VECTOR(4 downto 0);

-- pt PORT MAP (al doilea operand al imultirii)

aUAL:inout STD\_LOGIC\_VECTOR(4 downto 0);

aUALB:inout STD\_LOGIC\_VECTOR(4 downto 0);

--nr:inout std\_logic\_vector (4 downto 0);

--switch: in std\_logic;

-- astept: in std\_logic;

rezultat: inout STD\_LOGIC\_VECTOR(9 downto 0);

-->aici va fi stocat rezultatul dintre operatia Op1 cu Op2

rezultati: inout STD\_LOGIC\_VECTOR(9 downto 0);

rezultatimp: inout STD\_LOGIC\_VECTOR(9 downto 0);

clk: inout std\_logic

);

end ual;

architecture unitarlog of ual is

component Sumatorplusmux --componenta pt operandul A

port(a:inout STD\_LOGIC\_VECTOR(4 downto 0);

zmux: inout STD\_LOGIC\_VECTOR(4 downto 0)

);

end component;

component SumatorplusmuxB --componenta pt operandul B

port(aB:inout STD\_LOGIC\_VECTOR(4 downto 0);

zmuxB: inout STD\_LOGIC\_VECTOR(4 downto 0)

);

end component;

component Booth\_Con\_RCI is

Port (

Z : out STD\_LOGIC\_VECTOR(9 DOWNTO 0);

ai:inout STD\_LOGIC\_VECTOR(4 downto 0);

aiB:inout STD\_LOGIC\_VECTOR(4 downto 0));

end component;

component impartire is

Port ( catrest:inout std\_logic\_vector(9 downto 0);

rest: inout std\_logic\_vector(4 downto 0);

cat: inout std\_logic\_vector(4 downto 0);

ai: inout STD\_LOGIC\_VECTOR(4 downto 0);

aiB: inout STD\_LOGIC\_VECTOR(4 downto 0));

end component;

--component introducere\_numere

--port(

-- nr:inout std\_logic\_vector (4 downto 0);

-- switch: in std\_logic;

-- astept: in std\_logic;

-- aUAL:inout STD\_LOGIC\_VECTOR(4 downto 0);

-- aUALB:inout STD\_LOGIC\_VECTOR(4 downto 0)

-- );

--end component;

begin

G1: Sumatorplusmux port map(a=>aUAL,zmux=>Op1);

G2: SumatorplusmuxB port map(aB=>aUALb,zmuxB=>Op2);

G3: Booth\_Con\_RCI port map(ai=>aual,aib=>aualb,Z=>rezultati);

G12: impartire port map(ai=>aual,aib=>aualb,catrest=>rezultatimp);

--G9: introducere\_numere port map ( nr=>nr, switch=>switch, astept=>astept, aual=>aual, aualb=>aualb);

process(Op1,Op2, sel,rezultat,clk)

variable semn: STD\_LOGIC; --semnul operatiei (avem nevoie sa stim daca mai convertim rezultatul la sfarsit)

begin

if(sel ="00") then

Y0<=Op1;

Y0B<=Op2;

rezultat<="00000"&(Y0+Y0B); --semnul e lasat in vector

semn:=rezultat(4); --pun semnul intr o variabila deoarece imi e mai usor cand transform in intregi cu semn

rezultat<=semn&semn&semn&semn&semn&(Y0+Y0B);

elsif(sel ="01") then

Y1<=Op1;

Y1B<=Op2;

rezultat<="00000"&(Y1-Y1B); --semnul e lasat in vector

semn:=rezultat(4); -- -,,-

rezultat<=semn&semn&semn&semn&semn&(Y1-Y1B);

elsif(sel="10") then

semn:=(Op1(4) xor Op2(4));

Y2<=Op1;

Y2B<=Op2;

rezultat<=rezultati;

elsif(sel="11") then

semn:=(Op1(4) xor Op2(4));

Y3<=Op1;

Y3B<=Op2;

rezultat<=rezultatimp;

end if;

end process;

end unitarlog;

**ÎNMULȚIRE**

\*Algoritmul de înmulțire a lui Booth care n permite să înmulțim și numere negative reprezentate în complement față de 2.

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.STD\_LOGIC\_ARITH.ALL;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

entity Booth\_Con\_RCI is

Port (

Z : out STD\_LOGIC\_VECTOR(9 DOWNTO 0);

ai: inout STD\_LOGIC\_VECTOR(4 downto 0);

aiB: inout STD\_LOGIC\_VECTOR(4 downto 0)

);

end Booth\_Con\_RCI;

architecture Behavioral of Booth\_Con\_RCI is

SIGNAL A:STD\_LOGIC\_VECTOR(10 DOWNTO 0); --|

SIGNAL S:STD\_LOGIC\_VECTOR(10 DOWNTO 0); --|

SIGNAL P:STD\_LOGIC\_VECTOR(10 DOWNTO 0); --|

SIGNAL P1,P1\_SHIFT:STD\_LOGIC\_VECTOR(10 DOWNTO 0); --| 5+5+1(pt shift-are) biti

SIGNAL P2,P2\_SHIFT:STD\_LOGIC\_VECTOR(10 DOWNTO 0); --|

SIGNAL P3,P3\_SHIFT:STD\_LOGIC\_VECTOR(10 DOWNTO 0); --|

SIGNAL P4,P4\_SHIFT:STD\_LOGIC\_VECTOR(10 DOWNTO 0); --|

SIGNAL P5,P5\_SHIFT:STD\_LOGIC\_VECTOR(10 DOWNTO 0); --|

SIGNAL P6,P6\_SHIFT:STD\_LOGIC\_VECTOR(10 DOWNTO 0);

component Sumatorplusmux --componenta pt operandul A

port(a:inout STD\_LOGIC\_VECTOR(4 downto 0);

zmux: inout STD\_LOGIC\_VECTOR(4 downto 0)

);

end component;

component SumatorplusmuxB --componenta pt operandul B

port(aB:inout STD\_LOGIC\_VECTOR(4 downto 0);

zmuxB: inout STD\_LOGIC\_VECTOR(4 downto 0)

);

end component;

signal X: STD\_LOGIC\_VECTOR(4 downto 0);

signal Y: STD\_LOGIC\_VECTOR(4 downto 0);

begin

G4: Sumatorplusmux port map(a=>ai, zmux=>X);

G5: SumatorplusmuxB port map(aB=>aiB,zmuxB=>Y);

A(10 DOWNTO 6)<=X; --A il variable contine pe X, la inceput

A(5 DOWNTO 0)<=(OTHERS=>'0'); --se pune 0 in rest

S(10 DOWNTO 6)<=NOT X + "00001";

--se calculeaza complementul numarului (ajuta la calcularea produsului) si se copiaza la inceputul lui S

S(5 DOWNTO 0)<=(OTHERS=>'0'); -- restul bitilor vectorului S vor fi 0

P(10 DOWNTO 6)<=(OTHERS=>'0'); --punem 0 pe primele pozitii ale lui P

P(5 DOWNTO 1)<=Y; P(0)<='0'; --P il variable contine pe Y, la sfarsit & 0

P1<=P WHEN (P(1 DOWNTO 0)="00" OR P(1 DOWNTO 0)="11")

--daca ultimele 2 cifre ale vectorului sunt 00 sau 11, in P1 se copiaza P

ELSE P+A WHEN P(1 DOWNTO 0)="01"

--daca ultimele 2 cifre ale vectorului sunt 01, in P1 se copiaza P+A

ELSE P+S WHEN P(1 DOWNTO 0)="10";

--daca ultimele 2 cifre ale vectorului sunt 10, in P1 se copiaza P+S

P1\_SHIFT(9 DOWNTO 0)<=P1(10 DOWNTO 1);

--se aplica shift-area la dreapta

P1\_SHIFT(10)<=P1(10);

--dupa shift-are, pe prima pozitie se pune bitul cel mai semnificativ a lui P1

P2<=P1\_SHIFT WHEN (P1\_SHIFT(1 DOWNTO 0)="00" OR P1\_SHIFT(1 DOWNTO 0)="11") -- se aplica aceasi metoda ca mai sus

ELSE P1\_SHIFT+A WHEN P1\_SHIFT(1 DOWNTO 0)="01"

ELSE P1\_SHIFT+S WHEN P1\_SHIFT(1 DOWNTO 0)="10";

P2\_SHIFT(9 DOWNTO 0)<=P2(10 DOWNTO 1);

P2\_SHIFT(10)<=P2(10);

P3<=P2\_SHIFT WHEN (P2\_SHIFT(1 DOWNTO 0)="00" OR P2\_SHIFT(1 DOWNTO 0)="11")

ELSE P2\_SHIFT+A WHEN P2\_SHIFT(1 DOWNTO 0)="01"

ELSE P2\_SHIFT+S WHEN P2\_SHIFT(1 DOWNTO 0)="10";

P3\_SHIFT(9 DOWNTO 0)<=P3(10 DOWNTO 1);

P3\_SHIFT(10)<=P3(10);

--P2<=P1\_SHIFT P2;

P4<=P3\_SHIFT WHEN (P3\_SHIFT(1 DOWNTO 0)="00" OR P3\_SHIFT(1 DOWNTO 0)="11")

ELSE P3\_SHIFT+A WHEN P3\_SHIFT(1 DOWNTO 0)="01"

ELSE P3\_SHIFT+S WHEN P3\_SHIFT(1 DOWNTO 0)="10";

P4\_SHIFT(9 DOWNTO 0)<=P4(10 DOWNTO 1);

P4\_SHIFT(10)<=P4(10);

P5<=P4\_SHIFT WHEN (P4\_SHIFT(1 DOWNTO 0)="00" OR P4\_SHIFT(1 DOWNTO 0)="11")

ELSE P4\_SHIFT+A WHEN P4\_SHIFT(1 DOWNTO 0)="01"

ELSE P4\_SHIFT+S WHEN P4\_SHIFT(1 DOWNTO 0)="10";

P5\_SHIFT(9 DOWNTO 0)<=P5(10 DOWNTO 1);

P5\_SHIFT(10)<=P5(10);

Z<=P5\_SHIFT(10 DOWNTO 1);

--in rezultatul final se variable copia ultima shift-are,dar fara ultimul bit din dreapta

end Behavioral;

**COMPLEMENT ÎN ÎNTREG**

\*Convertim înapoi în forma originală deoarece este mai ușor de implementat afișarea pe 7 segmente

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.STD\_LOGIC\_ARITH.ALL;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

entity complementintreg is

port( A: inout STD\_LOGIC\_VECTOR(9 downto 0); --numarul in complement ce va fi convertit

Y: inout STD\_LOGIC\_VECTOR(9 downto 0);

aUAL: inout STD\_LOGIC\_VECTOR(4 downto 0);

aUALB: inout STD\_LOGIC\_VECTOR(4 downto 0);

sel: inout STD\_LOGIC\_VECTOR(1 downto 0);

semn: inout STD\_LOGIC

--astept: in std\_logic; --pentru semn

--nr:inout std\_logic\_vector (4 downto 0);

--switch: in std\_logic

);

end complementintreg;

architecture complementininintreg of complementintreg is

component ual

port(

Y0: inout STD\_LOGIC\_VECTOR(4 downto 0);

Y1: inout STD\_LOGIC\_VECTOR(4 downto 0);

Y2: inout STD\_LOGIC\_VECTOR(4 downto 0);

Y3: inout STD\_LOGIC\_VECTOR(4 downto 0);

Y0B: inout STD\_LOGIC\_VECTOR(4 downto 0);

Y1B: inout STD\_LOGIC\_VECTOR(4 downto 0);

Y2B: inout STD\_LOGIC\_VECTOR(4 downto 0);

Y3B: inout STD\_LOGIC\_VECTOR(4 downto 0);

sel: inout STD\_LOGIC\_VECTOR(1 downto 0);

Op1: inout STD\_LOGIC\_VECTOR(4 downto 0);

Op2: inout STD\_LOGIC\_VECTOR(4 downto 0);

Opi1: inout STD\_LOGIC\_VECTOR(4 downto 0);

Opi2: inout STD\_LOGIC\_VECTOR(4 downto 0);

aUAL:inout STD\_LOGIC\_VECTOR(4 downto 0);

aUALB:inout STD\_LOGIC\_VECTOR(4 downto 0);

--nr:inout std\_logic\_vector (4 downto 0);

--switch: in std\_logic;

--astept: in std\_logic;

rezultat: inout STD\_LOGIC\_VECTOR(9 downto 0);

-->aici va fi stocat rezultatul dintre operatia Op1 cu Op2

rezultati: inout STD\_LOGIC\_VECTOR(9 downto 0);

rezultatimp: inout std\_logic\_vector(9 downto 0);

clk: inout std\_logic

);

end component;

begin

G7: ual port map(rezultat=>A, aual=>aual, aualb=>aualb,sel=>sel);

semn<=a(4);

process(A)

begin

if(semn='0')then

Y<=A; --pt 0 vom folosi doar reprezentarea cu plus(+0): 0 0000, nu(-0) 1 0000

elsif(semn='1') then

Y<=('0' & not A(8)& not A(7)& not A(6)& not A(5)& not A(4)& not A(3)& not A(2)& not A(1)& not A(0))+"0000000001";

end if;

end process;

end architecture;

**BINAR ÎN BCD**

\*Aici se convertește cuvântul în binar în BCD prin shift-ări stânga și comparări cu 0100, adunându-se de fiecare dată 0011 când vectorul, împarțit în vectori mai mici de 4 biți, este mai mare.

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.numeric\_std.all;

entity bin2bcd\_12bit is

Port ( binIN : inout STD\_LOGIC\_VECTOR (11 downto 0);

semn:inout std\_logic;

aUAL: inout STD\_LOGIC\_VECTOR(4 downto 0);

aUALB: inout STD\_LOGIC\_VECTOR(4 downto 0);

ones : inout STD\_LOGIC\_VECTOR (3 downto 0);

tens : inout STD\_LOGIC\_VECTOR (3 downto 0);

hundreds : inout STD\_LOGIC\_VECTOR (3 downto 0);

sel: inout STD\_LOGIC\_VECTOR(1 downto 0);

--nr:inout std\_logic\_vector (4 downto 0);

--switch: in std\_logic;

--astept: in std\_logic;

thousands : inout STD\_LOGIC\_VECTOR (3 downto 0)

);

end bin2bcd\_12bit;

architecture Behavioral of bin2bcd\_12bit is

component complementintreg

port( A: inout STD\_LOGIC\_VECTOR(9 downto 0);

--numarul in complement ce variable fi convertit

Y: inout STD\_LOGIC\_VECTOR(9 downto 0); --numarul convertit

semn: inout STD\_LOGIC;

sel: inout STD\_LOGIC\_VECTOR(1 downto 0); --pentru semn

aUAL: inout STD\_LOGIC\_VECTOR(4 downto 0);

aUALB: inout STD\_LOGIC\_VECTOR(4 downto 0)

--nr:inout std\_logic\_vector (4 downto 0);

--astept: in std\_logic;

--switch: in std\_logic

);

end component;

begin

G8: complementintreg port map ( y=>binIN(9 downto 0), semn=>semn, aual=>aual, aualb=>aualb,sel=>sel);

binIN(11 downto 10)<="00";

bcd1: process(binIN)

variable temp : STD\_LOGIC\_VECTOR (11 downto 0); -- temporary variable

-- variabilele ce sinconizeaza output-ul lui BCD

-- thousands = bcd(15 downto 12)

-- hundreds = bcd(11 downto 8)

-- tens = bcd(7 downto 4)

-- units = bcd(3 downto 0)

variable bcd : UNSIGNED (15 downto 0) := (others => '0');

begin

bcd := (others => '0'); -- punem 0 in bcd

temp(11 downto 0) := binIN; -- citesc imput-ul in variabila temporala

for i in 0 to 11 loop

if bcd(3 downto 0) > 4 then

bcd(3 downto 0) := bcd(3 downto 0) + 3;

end if;

if bcd(7 downto 4) > 4 then

bcd(7 downto 4) := bcd(7 downto 4) + 3;

end if;

if bcd(11 downto 8) > 4 then

bcd(11 downto 8) := bcd(11 downto 8) + 3;

end if;

-- thousands nu pot fi mai mari decat 4 pet nr pe 12 biti, deci nu vom face nimic

bcd := bcd(14 downto 0) & temp(11);

--shift-am bcd la stanga cu 1 bit, copiem cel mai semnificativ bit a variabilei temp in cel mai nesemnificativ bit a llu bcd

temp := temp(10 downto 0) & '0'; --shift-am in stanga cu 1 bit

end loop;

ones <= STD\_LOGIC\_VECTOR(bcd(3 downto 0)); --iesirile

tens <= STD\_LOGIC\_VECTOR(bcd(7 downto 4));

hundreds <= STD\_LOGIC\_VECTOR(bcd(11 downto 8));

thousands <= STD\_LOGIC\_VECTOR(bcd(15 downto 12));

end process bcd1;

end Behavioral;

**AFIȘAREA PE 7 SEGMENTE**

\*afișează pe cele 4 afișoare codul convertit în binar\_in\_bcd

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.std\_logic\_unsigned.all;

entity x7seg is

port(x: inout std\_logic\_vector(15 downto 0);

clk: in std\_logic;

clr: in std\_logic;

a\_to\_g: out std\_logic\_vector (6 downto 0);

semn: inout std\_logic;

an: out std\_logic\_vector (3 downto 0);

sel: inout STD\_LOGIC\_VECTOR(1 downto 0);

aUAL: inout STD\_LOGIC\_VECTOR(4 downto 0);

aUALB: inout STD\_LOGIC\_VECTOR(4 downto 0);

--nr:inout std\_logic\_vector (4 downto 0);

--switch: in std\_logic;

--astept: in std\_logic;

dp: out std\_logic);

end x7seg;

architecture x7seg of x7seg is

component bin2bcd\_12bit

Port ( binIN : inout STD\_LOGIC\_VECTOR (11 downto 0);

semn:inout std\_logic;

ones : inout STD\_LOGIC\_VECTOR (3 downto 0);

tens : inout STD\_LOGIC\_VECTOR (3 downto 0);

aUAL: inout STD\_LOGIC\_VECTOR(4 downto 0);

aUALB: inout STD\_LOGIC\_VECTOR(4 downto 0);

sel: inout STD\_LOGIC\_VECTOR(1 downto 0);

--nr:inout std\_logic\_vector (4 downto 0);

--switch: in std\_logic;

--astept: in std\_logic;

hundreds : inout STD\_LOGIC\_VECTOR (3 downto 0);

thousands : inout STD\_LOGIC\_VECTOR (3 downto 0)

);

end component;

signal s: std\_logic\_vector (1 downto 0);

signal digit: std\_logic\_vector (3 downto 0);

signal aen: std\_logic\_vector (3 downto 0);

signal clkdiv: std\_logic\_vector (19 downto 0);

begin

G8: bin2bcd\_12bit port map (binin=>open, semn=>semn, ones=>x(3 downto 0), tens=>x(7 downto 4),

hundreds => x(11 downto 8), thousands=>x(15 downto 12), aual=>aual, aualb=>aualb,sel=>sel);

s <= clkdiv(19 downto 18); --selectia pe 2 biti

aen <= "1111"; --anodul

dp <= '1';

-- mux

process(s, x)

begin

case s is

when "00" => digit <= x(3 downto 0);

when "01" => digit <= x(7 downto 4);

when "10" => digit <= x(11 downto 8);

when others => digit <= x(15 downto 12);

end case;

end process;

process(digit)

begin

case digit is

when "0000" => a\_to\_g <= "0000001";

when "0001" => a\_to\_g <= "1001111";

when "0010" => a\_to\_g <= "0010010";

when "0011" => a\_to\_g <= "0000110";

when "0100" => a\_to\_g <= "1001100";

when "0101" => a\_to\_g <= "0100100";

when "0110" => a\_to\_g <= "0100000";

when "0111" => a\_to\_g <= "0001101";

when "1000" => a\_to\_g <= "0000000";

when "1001" => a\_to\_g <= "0000100";

when others => a\_to\_g <= "0111000";

end case;

end process;

process (s, aen) --selecteaza cifra : ones/tens etc

begin

an <= "1111";

if aen(conv\_integer(s)) = '1' then

an(conv\_integer(s)) <= '0';

end if;

end process;

process(clk, clr) --divizor de clock

begin

if clr = '1' then

clkdiv <= (others => '0');

elsif clk'event and clk = '1' then

clkdiv <= clkdiv + 1;

end if;

end process;

end x7seg;

**ÎMPĂRȚIRE**

\*face împărțirea celor 2 numere pozitive

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity impartire is

port(

catrest:inout std\_logic\_vector(9 downto 0);

rest: inout std\_logic\_vector(4 downto 0);

cat: inout std\_logic\_vector(4 downto 0);

ai: inout STD\_LOGIC\_VECTOR(4 downto 0);

aiB: inout STD\_LOGIC\_VECTOR(4 downto 0)

);

end impartire;

architecture imp of impartire is

component Sumatorplusmux --componenta pt operandul A

port(a:inout STD\_LOGIC\_VECTOR(4 downto 0);

zmux: inout STD\_LOGIC\_VECTOR(4 downto 0)

);

end component;

component SumatorplusmuxB --componenta pt operandul B

port(aB:inout STD\_LOGIC\_VECTOR(4 downto 0);

zmuxB: inout STD\_LOGIC\_VECTOR(4 downto 0)

);

end component;

signal Ximp: STD\_LOGIC\_VECTOR(4 downto 0);

signal Yimp: STD\_LOGIC\_VECTOR(4 downto 0);

begin

G10: Sumatorplusmux port map(a=>ai, zmux=>Ximp);

G11: SumatorplusmuxB port map(aB=>aiB,zmuxB=>Yimp);

process(Ximp,Yimp)

variable quotient: unsigned (4 downto 0);

variable remainder: unsigned (4 downto 0);

begin

quotient := (others => '0'); -- punem in cat si rest "0000"

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

for i in 4 downto 0 loop

remainder := remainder (3 downto 0) & '0'; -- shift-am la stanga cu 1 bit

remainder(0) := Ximp(i);

if remainder >= unsigned(Yimp) then

remainder := remainder - unsigned(Yimp);

quotient(i) := '1';

end if;

end loop;

cat <= std\_logic\_vector(quotient);

rest <= std\_logic\_vector(remainder);

end process;

catrest<=cat&rest;

end imp;

**Introducere numere**

\*face ca primul operand sa îl „aștepte” pe cel de al doilea

library ieee;

use IEEE.STD\_LOGIC\_1164.ALL;

entity introducere\_numere is

port(

nr:inout std\_logic\_vector (4 downto 0);

switch: in std\_logic;

aUAL:inout STD\_LOGIC\_VECTOR(4 downto 0);

astept: in std\_logic;

aUALB:inout STD\_LOGIC\_VECTOR(4 downto 0)

);

end introducere\_numere;

architecture Behavioral of introducere\_numere is

begin

process(switch)

begin

if switch='0' then

aual<=nr;

else

if astept='1' then

aualb<=nr;

end if;

end if;

end process;

end Behavioral;

BIBLIOGRAFIE

https://en.wikipedia.org/wiki/Two%27s\_complement#Sign\_extension

https://ro.wikipedia.org/wiki/Sistem\_binar#Sc.C4.83derea\_.C3.AEn\_binar

https://www.doulos.com/knowhow/vhdl\_designers\_guide/components\_and\_port\_maps/

https://www.youtube.com/watch?v=jcUIETi-VtM

https://www.youtube.com/watch?v=Iw5J7UsA\_kc

https://www.youtube.com/watch?v=iF1BTl1Cf6U

http://www.slideshare.net/naseermna/seminar-on-digital-multiplierbooth-multiplier-using-vhdl

http://stud.usv.ro/~nitco/anII/ASC/lab1\_2.pdf

http://www.dannicula.ro/hdl/lab/arith.html

http://www.acs.pub.ro/~cpop/CN1\_2CB/Laborator3-CN1.pdf

https://en.wikipedia.org/wiki/Booth%27s\_multiplication\_algorithm

http://www.acsa.upt.ro/teaching/AC/Lucrarea\_8.pdf

http://users.utcluj.ro/~baruch/book\_ac/AC-Inmultire-Booth.pdf

https://cseweb.ucsd.edu/classes/wi06/cse140/LN07.pdf

http://www.slideshare.net/naseermna/seminar-on-digital-multiplierbooth-multiplier-using-vhdl

http://embdev.net/topic/263093

http://www.vlsibank.com/sessionspage.asp?titl\_id=20025

http://www.cs.utah.edu/~rajeev/cs3810/slides/3810-08.pdf

http://www.slideshare.net/muhammadzawawi1/csc1401-lecture03-computer-arithmetic-arithmetic-and-logic-unit-alu?qid=c7373020-8b3b-4b2d-9a19-2c873e6cbd65&v=&b=&from\_search=4

http://hpds.ee.kuas.edu.tw/download/advanced\_CA/mathematics.pdf

http://stud.usv.ro/~nitco/anII/ASC/lab1\_2.pdf

-http://vhdlguru.blogspot.ro/2010/04/8-bit-binary-to-bcd-converter-double.html

https://en.wikipedia.org/wiki/Double\_dabble

http://www.ics.uci.edu/~jmoorkan/vhdlref/function.html

http://users.utcluj.ro/~baruch/ssc/labor/Aritm-Secventiala.pdf

http://stackoverflow.com/questions/12133810/non-restoring-division-algorithm

http://jjmk.dk/MMMI/Lessons/06\_Arithmetics/No4\_Division/Index.htm

http://vhdlguru.blogspot.ro/2010/03/vhdl-function-for-division-two-signed.html