FACULTATEA DE AUTOMATICA SI CALCULATOARE

CALCULATOARE SI TEHNOLOGIA INFORMATIEI

CIRCUITE DE INMULTIRE ZECIMALA

Student: Badea Ioana Cristiana

Indrumator: Lisman Florin

Grupa: 30237

Data: 10.01.2024

Contents

[1.REZUMAT 3](#_Toc156124785)

[2.INTRODUCERE 4](#_Toc156124786)

[3. FUNDAMENTARE TEORETICA 5](#_Toc156124787)

[3.1 MEDIUL DE DEZVOLTARE XILINX VIVADO DESIGN SUITE 5](#_Toc156124788)

[3.2 LIMBAJUL VHDL 5](#_Toc156124789)

[3.3 FPGA 6](#_Toc156124790)

[3.4 SOLUTII PROPUSE 6](#_Toc156124791)

[3.4.1 METODA ADUNARII REPETATE 6](#_Toc156124792)

[3.4.2 METODA CELOR NOUA MULTIPLII AI DEINMULTITULUI 6](#_Toc156124793)

[4. PROIECTARE SI IMPLEMENTARE 8](#_Toc156124794)

[4.1 Primul inmultitor 8](#_Toc156124795)

[4.2 Al doilea inmultitor 9](#_Toc156124796)

[4.3 Sumator zecimal 11](#_Toc156124797)

[5. REZULTATE EXPERIMENTALE 12](#_Toc156124798)

[6. CONCLUZII 14](#_Toc156124799)

[7. BIBLIOGRAFIE 14](#_Toc156124800)

[8. ANEXE 15](#_Toc156124801)

[Anexa A: inmultitor 15](#_Toc156124802)

[Anexa B: inmultitor1 17](#_Toc156124803)

[Anexa C: control 20](#_Toc156124804)

[Anexa D: control2 23](#_Toc156124805)

[Anexa E: debounce 25](#_Toc156124806)

[Anexa F: reg16 27](#_Toc156124807)

[Anexa G: reg20 28](#_Toc156124808)

[Anexa H: register\_file 30](#_Toc156124809)

[Anexa I: SSD 31](#_Toc156124810)

[Anexa J: sumator 1bit 33](#_Toc156124811)

[Anexa K: sumator 4biti 34](#_Toc156124812)

[Anexa L: sumator zecimal 36](#_Toc156124813)

[Anexa M: sumator zecimal 5 cifre 37](#_Toc156124814)

[Anexa N: sumator zecimal 4/5 cifre 39](#_Toc156124815)

[Anexa 0: proiect 41](#_Toc156124816)

# 1.REZUMAT

În cadrul acestui proiect, am abordat dezvoltarea și implementarea unui circuit de înmulțire zecimală în mediul Vivado. Problema esențială a constat în optimizarea și eficientizarea operațiilor de înmulțire pentru numerele zecimale în cadrul arhitecturilor de calcul moderne.

Obiectivele principale au inclus proiectarea și implementarea unui circuit hardware eficient, utilizând tehnici de optimizare specifice și limbaje de descriere hardware. Am adoptat o abordare iterativă, folosind VHDL pentru descrierea hardware-ului și beneficiind de un set variat de utilitare și metode de simulare pentru validare. Rezultatele au evidențiat o îmbunătățire semnificativă a performanței și eficienței operațiilor de înmulțire zecimală în comparație cu metodele tradiționale.

Concluziile principale au subliniat potențialul remarcabil al abordărilor hardware în optimizarea operațiilor zecimale, oferind o perspectivă valoroasă pentru viitoarele dezvoltări în acest domeniu.

# 2.INTRODUCERE

Proiectul se concentrează pe crearea unui circuit simplu pentru înmulțirea numerelor de 4 cifre folosind platforma Basys3, care, deși nu dispune de suficiente switch-uri și afișaje seven-segment, va necesita multiplexarea datelor cu ajutorul butoanelor disponibile. Utilizatorul va introduce operatorii unu câte unul, va selecta algoritmul de calcul, impulsul de declanșare a înmulțirii și chiar indicele care va controla afișajul într-un anumit moment. Pentru implementare, vom folosi mediu de dezvoltare Vivado și limbajul de descriere hardware VHDL.

Inmulțirea reprezintă una dintre cele mai frecvente operații într-un sistem de calcul și eficiența acestei operații este crucială. Înmulțirea zecimală este mai complexă decât cea binară sau hexazecimală, deoarece valorile zecimale trebuie ajustate în caz de depășiri. Proiectul va implementa două metode de înmulțire diferite: una mai puțin eficientă (metoda adunării repetate) și alta foarte eficientă (metoda celor nouă multipli ai deînmulțitului).

Operatorii vor fi introduși de la switch-uri: primul operand va fi introdus inițial, apoi se va activa un buton pentru a-l încărca. Ulterior, prin apăsarea unui alt buton, se va afișa cel de-al doilea operand, urmând a fi introdus de către utilizator. Un alt buton va iniția operația de înmulțire. Afișajul va prezenta rezultatul în două etape, având în vedere că acesta poate conține până la 8 cifre.

# 3. FUNDAMENTARE TEORETICA

## 3.1 MEDIUL DE DEZVOLTARE XILINX VIVADO DESIGN SUITE

Vivado Design Suite, dezvoltată de Xilinx, reprezintă o aplicație software pentru sinteza și analiza modelelor HDL, oferind un set extins de funcționalități în comparație cu predecesorul său, Xilinx ISE. Lansează în aprilie 2012, Vivado redefinește întregul flux de proiectare, aducând în prim-plan instrumente pentru dezvoltarea sistemelor pe cipuri și sinteza la nivel înalt. De asemenea, introduce sinteza la nivel înalt, permitând conversia codului C în logică programabilă.

Vivado vine cu o suită completă de instrumente de proiectare la nivel de sistem electronic (ESL), facilitând sinteza și verificarea IP-ului algoritmic bazat pe C. Oferă, de asemenea, abordări standardizate pentru împachetarea IP-ului algoritmic și RTL pentru reutilizare, integrând sistemele de diverse tipuri de blocuri de construcție ale sistemului și verificând atât blocurile, cât și sistemele.

Pe de altă parte, Xilinx ISE, întrerupt începând cu ediția lansată în octombrie 2013, a fost principalul instrument software pentru dezvoltarea de firmware încorporat pentru familiile de produse FPGA și CPLD ale Xilinx. Acesta a fost utilizat pentru sinteza, analiza și simularea modelelor HDL, împreună cu instrumente suplimentare precum Embedded Development Kit (EDK), Software Development Kit (SDK) și ChipScope Pro.

Este important de menționat că Xilinx ISE este strâns legat de arhitectura specifică a cipurilor Xilinx și nu poate fi utilizat cu FPGA-uri de la alte companii, având în vedere caracteristicile proprietare ale produselor Xilinx. Deși Xilinx ISE a fost utilizat în principal pentru sinteză și proiectare de circuite, suite-ul Vivado a adus îmbunătățiri semnificative, introducând abordări noi și extinse pentru dezvoltarea și sinteza la nivel înalt a sistemelor pe cipuri.

## 3.2 LIMBAJUL VHDL

Limbajul de descriere VHDL (Very High-Speed Integrated Circuit Hardware Description Language) este un limbaj de programare utilizat pentru a descrie comportamentul și structura circuitelor digitale. Prin VHDL, inginerii pot descrie operațiile logice și comportamentale ale unui circuit digital, precum și interconexiunile dintre diversele componente ale acestuia. Limbajul oferă o modalitate formală de a specifica funcționalitatea unui sistem digital, permițând simularea, sinteza și verificarea proiectelor de circuite.

VHDL permite descrierea structurală (prin conectarea componentelor logice) și descrierea comportamentală (definind funcționalitatea circuitului fără a specifica detalii structurale). De asemenea, oferă abstracții puternice pentru a modela sisteme complexe și permite reutilizarea codului prin definirea entităților și a arhitecturilor.

Un aspect puternic al VHDL constă în capacitatea sa de a specifica și simula sisteme digitale complexe înainte de implementarea lor fizică, facilitând astfel procesul de dezvoltare și debug. Datorită versatilității și capacității sale de a descrie diverse niveluri de abstractizare în proiectarea de circuite digitale, VHDL rămâne un instrument esențial în industria electronică și în proiectarea de sisteme digitale.

## 3.3 FPGA

FPGA (Field-Programmable Gate Array) reprezintă un tip de dispozitiv electronic programabil care poate fi configurat pentru a realiza o varietate de funcții digitale. Acestea sunt alcătuite dintr-o rețea de blocuri logice configurabile și interconexiuni, permițând programatorilor să configureze funcționalitatea dispozitivului pentru a realiza anumite operațiuni sau pentru a implementa diverse circuite digitale.

Prin FPGA, utilizatorii pot defini și configura circuitul la nivel de hardware pentru a îndeplini o gamă largă de cerințe. Ele sunt flexibile și versatilie, având capacitatea de a fi reprogramate și reconfigurate pentru diferite aplicații. Această caracteristică le face utile în prototipare rapidă, dezvoltare rapidă de produse și în aplicații care necesită adaptabilitate și actualizări frecvente. Configurarea unui FPGA implică scrierea unui set de instrucțiuni sau a unui cod care definește interconexiunile și funcționalitatea blocurilor logice ale dispozitivului. Acest cod poate fi scris în limbaje de descriere hardware precum VHDL sau Verilog, iar apoi este sintetizat și implementat pe dispozitivul FPGA.

## 3.4 SOLUTII PROPUSE

## 3.4.1 METODA ADUNARII REPETATE

Această metodă constă în examinarea cifrelor înmulţitorului, începând cu cifra c.m.p.s., şi adunarea repetată a deînmulţitului la produsul parţial. Numărul necesar de adunări este egal cu valoarea cifrei înmulţitorului, valoare determinată printr-o comparaţie. În locul comparaţiei, se poate testa dacă cifra înmulţitorului este 0; în caz contrar, se efectuează o adunare a deînmulţitului la produsul parţial, iar apoi o decrementare a cifrei înmulţitorului. Operaţia se repetă până când cifra înmulţitorului devine 0.

După efectuarea adunărilor specifice unei anumite cifre a înmulţitorului, grupul de registre care păstrează produsul parţial se deplasează la dreapta cu o poziţie. Operaţia continuă până la examinarea tuturor cifrelor înmulţitorului.

Această metodă este cea mai simplă, dar şi cea mai lentă. Numărul de adunări repetate poate fi redus dacă sumatorul zecimal este înlocuit cu un sumator-scăzător zecimal. Dacă cifra înmulţitorului este mai mică sau egală cu 5, se efectuează un număr corespunzător de adunări; dacă cifra este mai mare decât 5, se efectuează un număr de scăderi, şi se adună deînmulţitul la rangul zecimal următor (operaţie echivalentă cu adunarea deînmulţitului multiplicat cu 10). De exemplu, dacă cifra înmulţitorului este 7, se efectuează trei scăderi pentru poziţia rangului zecimal respectiv şi o adunare la rangul zecimal următor.

## 3.4.2 METODA CELOR NOUA MULTIPLII AI DEINMULTITULUI

Această metodă constă în generarea la începutul operaţiei de înmulţire a celor nouă multipli ai deînmulţitului (X) şi memorarea acestora în grupuri de registre speciale. Prin examinarea fiecărei cifre a înmulţitorului (Y), se determină conţinutul cărui grup de 4 registre trebuie adunat la produsul parţial.

Considerăm un asemenea dispozitiv de înmulţire zecimală de tip serie-paralel (Figura 3.1). Elementele componente ale dispozitivului sunt următoarele:

AZ - Grup de registre cu rol de acumulator care păstrează produsul parţial.

QZ - Grup de registre care păstrează înmulţitorul.

BZ1..BZ9 - Grupuri de registre care păstrează multiplii deînmulţitului.

SZ - Sumator zecimal elementar pentru calculul produsului parţial.

D - Bistabil pentru memorarea cifrei de transport de la sumatorul zecimal.

COMP - Circuit comparator, care compară în fiecare etapă a operaţiei cifra curentă a înmulţitorului, QZi, cu indicele registrelor BZi care păstrează multiplii deînmulţitului. Prin activarea unui semnal de ieşire al circuitului comparator, care validează un grup de porţi ŞI, se aplică conţinutul registrului BZi corespunzător la o intrare a sumatorului zecimal.

În etapa de iniţializare se încarcă deînmulţitul şi înmulţitorul în registrele BZ1, respectiv QZ, se iniţializează acumulatorul AZ şi bistabilul D cu 0, şi se formează multiplii deînmulţitului. Pentru aceasta se utilizează circuite combinaţionale sau sumatorul zecimal. În ultimul caz, după încărcarea deînmulţitului în registrul BZ1, se adună acest registru cu registrul acumulator AZ (care a fost iniţializat cu 0), şi rezultatul se transferă în AZ. Se adună apoi conţinutul registrelor BZ1 şi AZ, iar rezultatul (2X) se transferă în BZ2. Se procedează similar pentru obţinerea celorlalţi multipli ai deînmulţitului.

Pentru înmulţire se efectuează următoarele operaţii:

1. Se testează cifra c.m.p.s. a înmulţitorului, QZ0.

2. Se adună la produsul parţial multiplul corespunzător al deînmulţitului, în funcţie de valoarea cifrei testate.

3. Se deplasează produsul parţial la dreapta cu o poziţie.

4. Se repetă etapele 1 - 3 până când se testează toate cifrele înmulţitorului.

5. Se stabileşte semnul rezultatului pe baza semnului operanzilor.

# 4. PROIECTARE SI IMPLEMENTARE

## 4.1 Primul inmultitor

Acest circuit de multiplicare utilizează metoda adunărilor repetate pentru a efectua operația de înmulțire. În primul rând, se încarcă cei doi operanzi în două registre, RegB și RegA. Aceste două registre reprezintă multiplicatorul (RegB) și multiplicandul (RegA).

Odată ce operanzii sunt încărcați, circuitul începe să efectueze operația de înmulțire prin adunarea repetată a produsului parțial cu deinmulțitorul atunci când controlerul activează semnalul de adunare. Acest lucru înseamnă că, pe măsură ce controlerul emite semnale de adunare, se adună produsul parțial cu deinmulțitorul și rezultatul este stocat într-un registru special sau într-un registru dedicat pentru rezultatul final.

Următorul pas implică shiftarea înregistrărilor RegA și RegB, pregătindu-le pentru următoarea iterație a procesului de înmulțire. Această shiftare reprezintă procesul de deplasare a biților către poziții mai mari, similar cu deplasarea decimală în înmulțirea manuală.

Repetăm acești pași de adunare, shiftare și verificare a numărului de iterații necesare în funcție de numărul de cifre al operandului mai mare (de obicei, operandul cu mai multe biți).

În final, obținem rezultatul final al înmulțirii, care este stocat în registrele corespunzătoare (RegA și RegB). Acest circuit implementează eficient operația de înmulțire, folosind adunări repetate și shiftări pentru a obține rezultatul dorit.

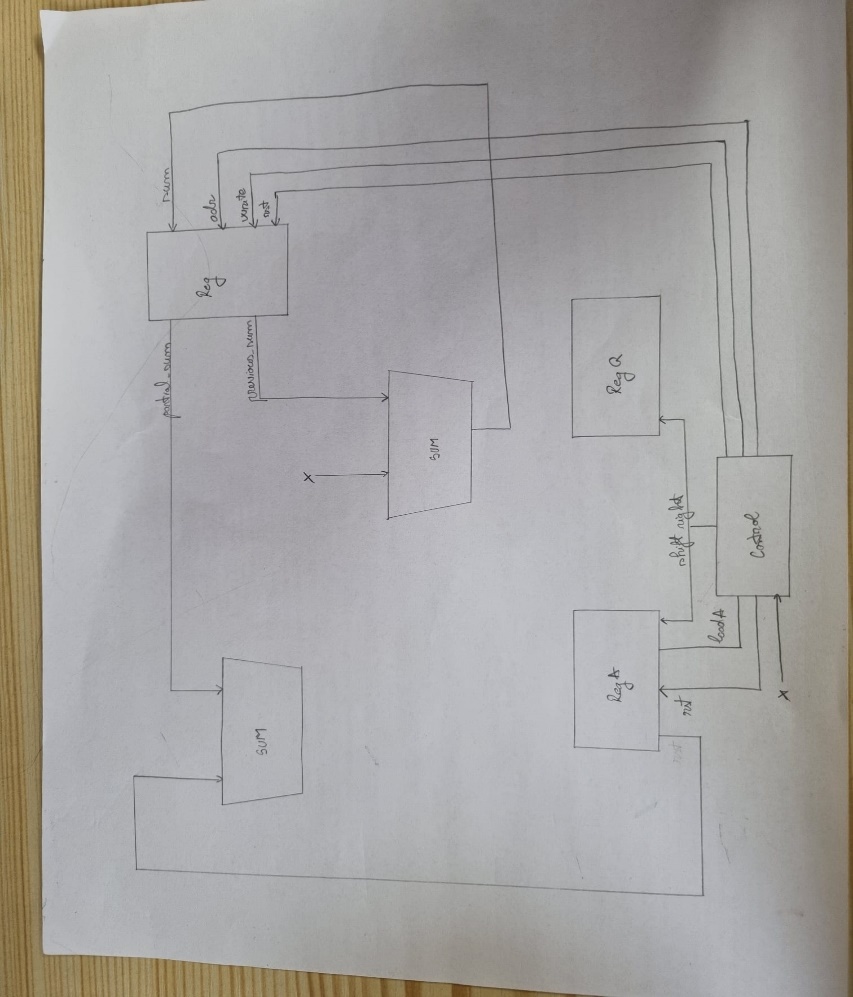


Fig. 1

A diagram with writing on paper

Description automatically generated

Fig. 2

## 4.2 Al doilea inmultitor

Acest al doilea multiplicator are o structură similară cu primul, cu excepția faptului că folosește un bloc de registre pentru a stoca produsul parțial al înmulțirii. Procesul începe prin resetarea blocului de registre. Selectia, un semnal de control, primește valoarea 1, indicând că începem să salvăm multiplii deinmulțitului, iar la indexul 0, evident, registrele trebuie să aibă valoarea 0.

Blocul de registre are două ieșiri: una care conține valoarea la indexul curent și alta care conține valoarea de la indexul precedent, pentru a putea fi utilizată în calculul tuturor sumelor. Această configurație permite stocarea succesivă a multiplilor deinmulțitului în blocul de registre.

Un sumator primește un operand ce conține deinmulțitul și un alt operand ce conține valoarea multiplului recent calculat. Astfel, se obține o nouă sumă care este salvată la indexul curent al blocului de registre. Se incrementează apoi indexul și se continuă acești pași până când se obțin toți cei 9 multiplii ai deinmulțitului.

Prin utilizarea acestui bloc de registre și a sumatorului, se eficientizează procesul de înmulțire, permițând stocarea și actualizarea continuă a produsului parțial înregistrat. Această abordare poate fi utilă în implementarea unui multiplicator mai complex și mai eficient.

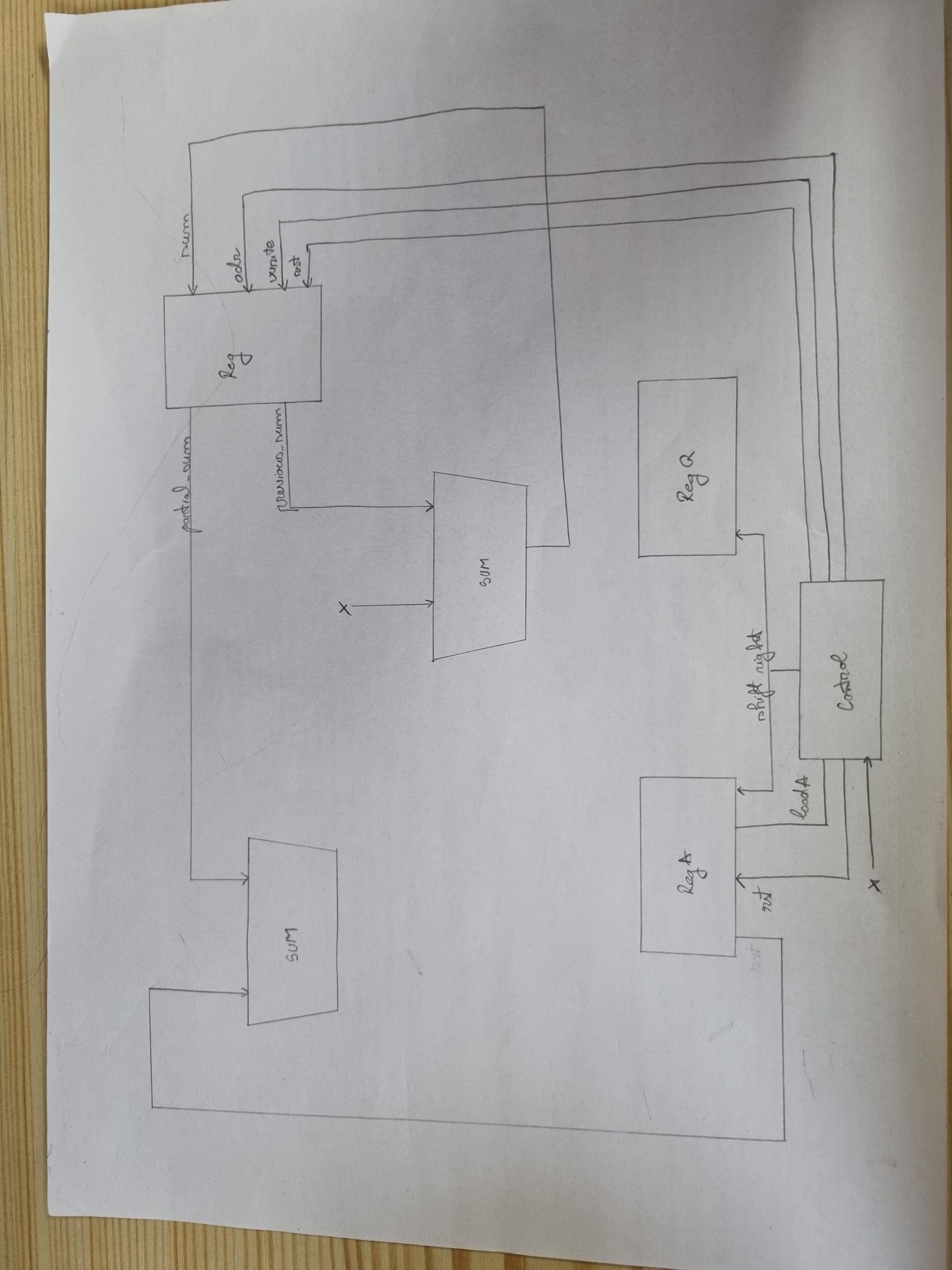


Fig. 3

A diagram of a flowchart

Description automatically generated

Fig. 4

## 4.3 Sumator zecimal

Un circuit BCD adder este un dispozitiv utilizat pentru adunarea a două numere reprezentate în Binary Coded Decimal (BCD). BCD reprezintă un sistem de codificare a numerelor în care fiecare cifră a unui număr este reprezentată independent prin intermediul unui grup de 4 biți. Un adder BCD adună două astfel de cifre BCD în paralel și generează o sumă în format BCD. În cazul în care suma depășește 9 sau apare un carry către cifra următoare, se aplică o corecție specifică - se adaugă valoarea 6 la rezultat. Acest proces asigură că rezultatul final este reprezentat în mod corespunzător în sistemul BCD și că orice transport este gestionat în mod adecvat.



Fig. 5

# 5. REZULTATE EXPERIMENTALE

Pentru testarea dispozitivului de inmultirea am folosit simulatorul Vivado. In urma aplicarii mai multor operanzi s-a constatat ca rezultatul de la iesirea inmultitorului este identic cu rezultatul aplicat. Cei doi operanzi sunt 72 si 73, iar rezultatul este 5256 in prima faza, iar apoi 73 si 77 cu rezultatul 5621.

A screenshot of a computer

Description automatically generated

Fig. 6

A screenshot of a computer

Description automatically generated

Fig. 7

A screenshot of a computer

Description automatically generated

Fig. 8

A screenshot of a computer

Description automatically generated

Fig. 9

# 6. CONCLUZII

Circuitul de înmulțire pe care l-am implementat este potrivit pentru o gamă variată de dispozitive numerice datorită complexității relativ reduse pe care o prezintă. În procesul de implementare a acestui proiect, am adunat o serie de cunoștințe vaste în VHDL și în implementarea de automate, învățând cum să transpun un algoritm abstract într-un circuit practic. Această experiență mi-a permis să pătrund mai adânc în lumea algoritmilor matematici implicați în operația de înmulțire și să înțeleg complexitatea asociată acestora. Am descoperit că atunci când se multiplică două numere, fiecare având lungimea de n biți, rezultatul va avea o lungime de 2 \* n biți.

În timpul procesului de testare, odată cu încărcarea circuitului pe o placă fizică, am înțeles că conceptele teoretice nu se potrivesc mereu cu realitatea practică. Există o discrepanță între proiectarea unui circuit care funcționează perfect într-o simulare virtuală și realizarea unuia care să se comporte perfect într-un mediu fizic.

Pentru a aduce îmbunătățiri ulterioare, am identificat mai multe direcții de dezvoltare. Acestea ar putea include extinderea numărului de cifre pe care le pot accepta operanzii sau realizarea înmulțirii între numere semnate. O altă direcție ar putea fi optimizarea algoritmului prin schimbul operanzilor, astfel încât să se efectueze cât mai puține operații posibile în procesul de înmulțire.

Un concept interesant ar fi să implementăm un contor care să înregistreze numărul de cicluri de ceas necesare pentru a finaliza operația de înmulțire. Astfel, am putea afișa durata totală a operației și intervalul de timp în care aceasta a avut loc, oferindu-ne o perspectivă asupra performanței circuitului.

Aceste direcții de dezvoltare ar putea contribui la îmbunătățirea și extinderea circuitului de înmulțire existent, oferindu-ne o mai mare flexibilitate și eficiență în utilizarea acestuia într-o varietate de scenarii și aplicații practice.

# 7. BIBLIOGRAFIE

[1] Baruch, F. , Zoltan, “Structura sistemelor de calcul – Dispozitive de inmultire si impartire zecimala”

[2] Baruch, F. , Zoltan, “Structura sistemelor de calcul – Circuite aritmetice secventiale”

[3] Baruch, F. , Zoltan, “Structura sistemelor de calcul – Circuite aritmetice combinationale”

# 8. ANEXE

## Anexa A: inmultitor

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.std\_logic\_unsigned.all;

entity inmultitor is

port(

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

y: in std\_logic\_vector(15 downto 0);

start: in std\_logic;

clk: in std\_logic;

rez: out std\_logic\_vector(31 downto 0);

finishFlag: out std\_logic

);

end entity;

architecture behaviour of inmultitor is

signal regQ, regB: std\_logic\_vector(15 downto 0);

signal rstA, loadB, loadQ, shiftAQ, loadSum, tout: std\_logic;

signal regA, sum: std\_logic\_vector(19 downto 0);

begin

rez <= regA(15 downto 0) & regQ(15 downto 0);

control: entity work.control

port map(

qLast => regQ(3 downto 0),

startM => start,

clk => clk,

rstA => rstA,

loadB => loadB,

loadQ => loadQ,

shiftAQ => shiftAQ,

loadSum => loadSum,

finishFlag => finishFlag

);

reA: entity work.reg20

port map(

s => sum,

load => loadSum,

si => "0000",

shift => shiftAQ,

rst => rstA,

clk => clk,

rez => regA

);

reQ: entity work.reg16

port map(

s => y,

load => loadQ,

si => regA(3 downto 0),

shift => shiftAQ,

rst => '0',

clk => clk,

rez => regQ

);

reB: entity work.reg16

port map(

s => x,

load => loadB,

si => "0000",

shift => '0',

rst => '0',

clk => clk,

rez => regB

);

alu: entity work.sumator\_zecimal4\_5

port map(

x => regB,

y => regA,

tin => '0',

s => sum(19 downto 0),

tout => tout

);

end architecture;

## Anexa B: inmultitor1

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.std\_logic\_unsigned.all;

entity inmultitor1 is

port(

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

y: in std\_logic\_vector(15 downto 0);

start: in std\_logic;

clk: in std\_logic;

rez: out std\_logic\_vector(31 downto 0);

finishFlag: out std\_logic

);

end entity;

architecture behaviour of inmultitor1 is

signal tout, tout1: std\_logic;

signal regQ: std\_logic\_vector(15 downto 0);

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

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

signal savedSum, savedAntSum: std\_logic\_vector(19 downto 0);

signal loadSum, shiftAQ, rstA, loadB, loadQ, writeReg, rstSum: std\_logic;

signal sum, sum1: std\_logic\_vector(19 downto 0);

begin

rez <= regA(15 downto 0) & regQ(15 downto 0);

sum\_register: entity work.sumator\_zecimal4\_5

port map(

x => x,

y => savedAntSum,

tin => '0',

s => sum1,

tout => tout

);

reg\_file: entity work.register\_file

port map(

rst => rstSum,

clk => clk,

x => sum1,

w => writeReg,

sel => sel,

zant => savedAntSum,

z => savedSum

);

control: entity work.control2

port map(

qLast => y,

startM => start,

clk => clk,

rstA => rstA,

loadB => loadB,

loadQ => loadQ,

shiftAQ => shiftAQ,

loadSum => loadSum,

finishFlag => finishFlag,

writeReg => writeReg,

sel => sel,

rstSum => rstSum

);

reA: entity work.reg20

port map(

s => sum,

load => loadSum,

si => "0000",

shift => shiftAQ,

rst => rstA,

clk => clk,

rez => regA

);

reQ: entity work.reg16

port map(

s => y,

load => loadQ,

si => regA(3 downto 0),

shift => shiftAQ,

rst => '0',

clk => clk,

rez => regQ

);

alu: entity work.sumator\_zecimal\_5

port map(

x => savedSum,

y => regA,

tin => '0',

s => sum(19 downto 0),

tout => tout1

);

end architecture;

## Anexa C: control

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.std\_logic\_unsigned.all;

entity control is

port(

qLast: in std\_logic\_vector(3 downto 0);

startM: in std\_logic;

clk: in std\_logic;

rstA: out std\_logic;

loadB: out std\_logic;

loadQ: out std\_logic;

shiftAQ: out std\_logic;

loadSum: out std\_logic;

finishFlag: out std\_logic

);

end entity;

architecture behaviour of control is

type tip\_stare is (idle, start, loadLast, testLast, decrementLast, testN, shift, finish);

signal stare: tip\_stare := idle;

signal n:integer;

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

begin

proc2:process(stare, qLast)

begin

case stare is

when idle => rstA <= '0'; loadB <= '0'; loadQ <= '0'; shiftAQ <= '0'; loadSum <= '0'; finishFlag<= '1';

when start => rstA <= '1'; loadB <= '1'; loadQ <= '1'; shiftAQ <= '0'; loadSum <= '0'; finishFlag <= '0'; n <= 3;

when loadLast => rstA <= '0'; loadB <= '0'; loadQ <= '0'; shiftAQ <= '0'; loadSum <= '0'; qL <= qLast;

when testLast => rstA <= '0'; loadB <= '0'; loadQ <= '0'; shiftAQ <= '0'; loadSum <= '0';

when decrementLast => rstA <= '0'; loadB <= '0'; loadQ <= '0'; shiftAQ<= '0'; loadSum <= '1'; qL <= qL - 1;

when testN => rstA <= '0'; loadB <= '0'; loadQ <= '0'; shiftAQ<= '0'; loadSum <= '0';

when shift => rstA <= '0'; loadB <= '0'; loadQ <= '0'; shiftAQ<= '1'; loadSum <= '0'; n <= n - 1;

when finish => rstA <= '0'; loadB <= '0'; loadQ <= '0'; shiftAQ<= '1'; loadSum <= '0'; finishFlag <= '1';

end case;

end process;

proc1: process(clk, qLast)

begin

if(rising\_edge(clk)) then

case stare is

when idle => if (startM = '1') then

stare <= start;

end if;

when start => stare <= loadLast;

when loadLast => stare <= testLast;

when testLast =>

if(qL = "0000") then

stare <= testN;

else

stare <= decrementLast;

end if;

when decrementLast => stare <= testLast;

when testN => if (n = 0) then

stare <= finish;

else

stare <= shift;

end if;

when shift => stare <= loadLast;

when finish => stare <= idle;

end case;

end if;

end process;

end architecture;

## Anexa D: control2

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.std\_logic\_unsigned.all;

entity control2 is

port(

qLast: in std\_logic\_vector(15 downto 0);

startM: in std\_logic;

clk: in std\_logic;

rstA: out std\_logic;

loadB: out std\_logic;

loadQ: out std\_logic;

shiftAQ: out std\_logic;

loadSum: out std\_logic;

finishFlag: out std\_logic;

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

writeReg: out std\_logic;

rstSum: out std\_logic

);

end entity;

architecture behaviour of control2 is

type tip\_stare is (idle, start, loadReg, loadLast, sum, testN, shift, finish);

signal stare: tip\_stare := idle;

signal n:integer;

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

signaL qLast1: std\_logic\_vector(15 downto 0);

signal count: std\_logic\_vector(3 downto 0):= "0001";

begin

proc2:process(clk)

begin

if(rising\_edge(clk)) then

case stare is

when idle => rstA <= '0'; loadB <= '0'; loadQ <= '0'; shiftAQ <= '0'; loadSum <= '0'; finishFlag<= '1';

when start => rstA <= '1'; loadB <= '1'; loadQ <= '1'; shiftAQ <= '0'; loadSum <= '0'; finishFlag <= '0'; n <= 4; qLast1 <= qLast; count <= "0001"; rstSum <= '0';

when loadReg => rstA <= '0'; loadB <= '0'; loadQ <= '0'; shiftAQ <= '0'; loadSum <= '0'; count <= count + 1; writeReg <= '1';

when loadLast => rstA <= '0'; loadB <= '0'; loadQ <= '0'; shiftAQ <= '0'; loadSum <= '0'; qL <= qLast1(3 downto 0); writeReg <= '0';

when sum => rstA <= '0'; loadB <= '0'; loadQ <= '0'; shiftAQ<= '0'; loadSum <= '1'; count <= ql;

when testN => rstA <= '0'; loadB <= '0'; loadQ <= '0'; shiftAQ<= '0'; loadSum <= '0';

when shift => rstA <= '0'; loadB <= '0'; loadQ <= '0'; shiftAQ<= '1'; loadSum <= '0'; n <= n - 1; qLast1 <= "0000" & qLast1(15 downto 4);

when finish => rstA <= '0'; loadB <= '0'; loadQ <= '0'; shiftAQ<= '1'; loadSum <= '0'; finishFlag <= '1';

end case;

sel <= count;

end if;

end process;

proc1: process(clk)

begin

if(rising\_edge(clk)) then

case stare is

when idle => if (startM = '1') then

stare <= start;

end if;

when start => stare <= loadReg;

when loadReg => if(count = "1001")then

stare <= loadLast;

else

stare <= loadReg;

end if;

when loadLast => stare <= sum;

when sum => stare <= testN;

when testN => if (n = 0) then

stare <= finish;

else

stare <= shift;

end if;

when shift => stare <= loadLast;

when finish => stare <= idle;

end case;

end if;

end process;

end architecture;

## Anexa E: debounce

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use ieee.std\_logic\_unsigned.all;

entity debounce is

Port ( Clk : in STD\_LOGIC;

Rst : in STD\_LOGIC;

D\_in : in STD\_LOGIC;

Q\_out : out STD\_LOGIC);

end debounce;

architecture Behavioral of debounce is

signal Q1, Q2, Q3 : std\_logic;

signal nr:std\_logic\_vector(15 downto 0):=x"0000";

begin

q\_out<=q2 and not q3;

process(clk)

begin

if(rising\_edge(clk))then

nr<=nr+'1';

end if;

end process;

process(clk)

begin

if(rising\_edge(clk))then

if(nr=x"ffff")then

q1<=d\_in;

end if;

end if;

end process;

process(clk)

begin

if(rising\_edge(clk))then

q2<=q1;

q3<=q2;

end if;

end process;

end Behavioral;

## Anexa F: reg16

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.std\_logic\_unsigned.all;

entity reg16 is

port(

s: in std\_logic\_vector(15 downto 0);

load: in std\_logic;

si: in std\_logic\_vector(3 downto 0);

shift: in std\_logic;

rst: in std\_logic;

clk: in std\_logic;

rez: out std\_logic\_vector(15 downto 0)

);

end entity;

architecture behavour of reg16 is

signal content: std\_logic\_vector(15 downto 0);

begin

process(clk, rst)

begin

if(rst = '1') then

content <= x"0000";

else

if(rising\_edge(clk)) then

if(shift = '1') then

content(3 downto 0) <= content(7 downto 4);

content(7 downto 4) <= content(11 downto 8);

content(11 downto 7) <= content(15 downto 11);

content(15 downto 12)<= si;

elsif(load = '1') then

content <= s;

end if;

end if;

end if;

rez <= content;

end process;

end architecture;

## Anexa G: reg20

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.std\_logic\_unsigned.all;

entity reg20 is

port(

s: in std\_logic\_vector(19 downto 0);

load: in std\_logic;

si: in std\_logic\_vector(3 downto 0);

shift: in std\_logic;

rst: in std\_logic;

clk: in std\_logic;

rez: out std\_logic\_vector(19 downto 0)

);

end entity;

architecture behavour of reg20 is

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

begin

process(clk, rst)

begin

if(rst = '1') then

content <= x"00000";

else

if(rising\_edge(clk)) then

if(shift = '1') then

content(3 downto 0) <= content(7 downto 4);

content(7 downto 4) <= content(11 downto 8);

content(11 downto 7) <= content(15 downto 11);

content(15 downto 12)<= content(19 downto 16);

content(19 downto 16) <= si;

elsif(load = '1') then

content <= s;

end if;

end if;

end if;

rez <= content;

end process;

end architecture;

## Anexa H: register\_file

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.std\_logic\_unsigned.all;

entity register\_file is

port(

rst: in std\_logic;

clk: in std\_logic;

x: in std\_logic\_vector(19 downto 0);

w: in std\_logic;

sel: in std\_logic\_vector(3 downto 0);

zant: out std\_logic\_vector(19 downto 0);

z: out std\_logic\_vector(19 downto 0)

);

end entity;

architecture behavioral of register\_file is

type reg\_type is array(0 to 15) of std\_logic\_vector(19 downto 0);

signal sume: reg\_type :=(others => x"00000");

begin

process(clk)

begin

if(rst = '1') then

sume <= (others => x"00000");

elsif(rising\_edge(clk)) then

if(w = '1') then

sume(conv\_integer(sel)) <= x;

end if;

end if;

zant <= sume(conv\_integer(sel - 1));

z <= sume(conv\_integer(sel));

end process;

end architecture;

## Anexa I: SSD

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

use IEEE.STD\_LOGIC\_ARITH.ALL;

entity SSD is

Port ( clk : in STD\_LOGIC;

digit: in STD\_LOGIC\_VECTOR(15 downto 0);

cat : out STD\_LOGIC\_VECTOR (6 downto 0);

an : out STD\_LOGIC\_VECTOR (3 downto 0));

end SSD;

architecture Behavioral of SSD is

signal temp: std\_logic\_vector(15 downto 0):="0000000000000000";

signal mux1: std\_logic\_vector(3 downto 0);

begin

process(clk)

begin

if(rising\_edge(clk)) then

temp <= temp+1;

end if;

end process;

process(temp,digit)

begin

case temp(15 downto 14) is

when "00"=> mux1 <=digit(3 downto 0);

when "01"=> mux1 <=digit(7 downto 4);

when "10"=> mux1 <=digit(11 downto 8);

when "11"=> mux1 <=digit(15 downto 12);

when others =>

end case;

end process;

process(temp)

begin

case temp(15 downto 14) is

when "00"=> an<="1110";

when "01"=> an<="1101";

when "10"=> an<="1011";

when "11"=> an<="0111";

when others =>

end case;

end process;

with mux1 select

cat<= "1111001" when "0001", --1

"0100100" when "0010", --2

"0110000" when "0011", --3

"0011001" when "0100", --4

"0010010" when "0101", --5

"0000010" when "0110", --6

"1111000" when "0111", --7

"0000000" when "1000", --8

"0010000" when "1001", --9

"0001000" when "1010", --A

"0000011" when "1011", --b

"1000110" when "1100", --C

"0100001" when "1101", --d

"0000110" when "1110", --E

"0001110" when "1111", --F

"1000000" when others; --0

end Behavioral;

## Anexa J: sumator 1bit

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity sumator\_1bit is

Port (

x: in std\_logic;

y: in std\_logic;

tin: in std\_logic;

s: out std\_logic;

g: out std\_logic;

p: out std\_logic

);

end sumator\_1bit;

architecture Behavioral of sumator\_1bit is

begin

s <= x xor y xor tin;

g <= x and y;

p <= x or y;

end Behavioral;

## Anexa K: sumator 4biti

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity sumator\_4biti is

Port (

x: in std\_logic\_vector(3 downto 0);

y: in std\_logic\_vector(3 downto 0);

tin: in std\_logic;

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

tout: out std\_logic

);

end sumator\_4biti;

architecture Behavioral of sumator\_4biti is

signal g, p, t: std\_logic\_vector(3 downto 0);

begin

s0: entity work.sumator\_1bit

port map(

x => x(0),

y => y(0),

tin => t(0),

s => s(0),

g => g(0),

p => p(0)

);

s1: entity work.sumator\_1bit

port map(

x => x(1),

y => y(1),

tin => t(1),

s => s(1),

g => g(1),

p => p(1)

);

s2: entity work.sumator\_1bit

port map(

x => x(2),

y => y(2),

tin => t(2),

s => s(2),

g => g(2),

p => p(2)

);

s3: entity work.sumator\_1bit

port map(

x => x(3),

y => y(3),

tin => t(3),

s => s(3),

g => g(3),

p => p(3)

);

t(0) <= tin;

t(1) <= g(0) or (p(0) and t(0));

t(2) <= g(1) or (p(1) and t(1));

t(3) <= g(2) or (p(2) and t(2));

tout <= g(3) or (p(3) and t(3));

end Behavioral;

## Anexa L: sumator zecimal

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity sumator\_zecimal is

Port (

x: in std\_logic\_vector(3 downto 0);

y: in std\_logic\_vector(3 downto 0);

tin: in std\_logic;

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

tout: out std\_logic

);

end sumator\_zecimal;

architecture Behavioral of sumator\_zecimal is

signal t,zero, t1:std\_logic;

signal sint, xint: std\_logic\_vector(3 downto 0);

begin

sumator1: entity work.sumator\_4biti

port map(

x => x,

y => y,

tin => tin,

s => sint,

tout => t

);

xint <= '0' & t1 & t1 & '0';

sumator2: entity work.sumator\_4biti

port map(

x => xint,

y => sint,

tin => '0',

s => s,

tout => zero

);

t1 <= t or (sint(3) and sint(2)) or (sint(3) and sint(1));

tout <= t1;

end Behavioral;

## Anexa M: sumator zecimal 5 cifre

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.std\_logic\_unsigned.all;

entity sumator\_zecimal\_5 is

port(

x: in std\_logic\_vector(19 downto 0);

y: in std\_logic\_vector(19 downto 0);

tin: in std\_logic;

s: out std\_logic\_vector(19 downto 0);

tout: out std\_logic

);

end sumator\_zecimal\_5;

architecture behaviour of sumator\_zecimal\_5 is

signal t1, t2, t3, t4: std\_logic;

begin

s0: entity work.sumator\_zecimal

port map(

x => x(3 downto 0),

y => y(3 downto 0),

tin => tin,

s => s(3 downto 0),

tout => t1

);

s1: entity work.sumator\_zecimal

port map(

x => x(7 downto 4),

y => y(7 downto 4),

tin => t1,

s => s(7 downto 4),

tout => t2

);

s2: entity work.sumator\_zecimal

port map(

x => x(11 downto 8),

y => y(11 downto 8),

tin => t2,

s => s(11 downto 8),

tout => t3

);

s3: entity work.sumator\_zecimal

port map(

x => x(15 downto 12),

y => y(15 downto 12),

tin => t3,

s => s(15 downto 12),

tout => t4

);

s4: entity work.sumator\_zecimal

port map(

x => x(19 downto 16),

y => y(19 downto 16),

tin => t4,

s => s(19 downto 16),

tout => tout

);

end architecture;

## Anexa N: sumator zecimal 4/5 cifre

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.std\_logic\_unsigned.all;

entity sumator\_zecimal4\_5 is

port(

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

y: in std\_logic\_vector(19 downto 0);

tin: in std\_logic;

s: out std\_logic\_vector(19 downto 0);

tout: out std\_logic

);

end sumator\_zecimal4\_5;

architecture behaviour of sumator\_zecimal4\_5 is

--type numar4 is array(3 downto 0) of std\_logic\_vector(3 downto 0);

--signal numar: numar4;

signal t1, t2, t3, t4: std\_logic;

begin

s0: entity work.sumator\_zecimal

port map(

x => x(3 downto 0),

y => y(3 downto 0),

tin => tin,

s => s(3 downto 0),

tout => t1

);

s1: entity work.sumator\_zecimal

port map(

x => x(7 downto 4),

y => y(7 downto 4),

tin => t1,

s => s(7 downto 4),

tout => t2

);

s2: entity work.sumator\_zecimal

port map(

x => x(11 downto 8),

y => y(11 downto 8),

tin => t2,

s => s(11 downto 8),

tout => t3

);

s3: entity work.sumator\_zecimal

port map(

x => x(15 downto 12),

y => y(15 downto 12),

tin => t3,

s => s(15 downto 12),

tout => t4

);

s4: entity work.sumator\_zecimal

port map(

x => "0000",

y => y(19 downto 16),

tin => t4,

s => s(19 downto 16),

tout => tout

);

end architecture;

## Anexa 0: proiect

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use ieee.std\_logic\_unsigned.all;

entity project is

Port (

sw: in std\_logic\_vector(15 downto 0);

button1: in std\_logic;--schimbare date afisor

button2: in std\_logic;--load input

button3: in std\_logic;--alegere metoda

button4: in std\_logic;-- start

clk: in std\_logic;

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

cat: out std\_logic\_vector(6 downto 0);

finishFlag: out std\_logic

);

end project;

architecture Behavioral of project is

signal digit: std\_logic\_vector(15 downto 0);

signal rez1, rez2, rez: std\_logic\_vector(31 downto 0):= x"ffffeeee";

signal x, y: std\_logic\_vector(15 downto 0):= x"0000";

signal start, finishFlag1, finishFlag2, en1, en2, en3: std\_logic:= '0';

signal count1: std\_logic\_vector(1 downto 0):= "00";--date iesire afisor

signal count2: std\_logic:= '0';--date intrare

signal count3: std\_logic:= '0';

begin

debounce1: entity work.debounce

port map(

clk => clk,

rst => '0',

d\_in => button1,

q\_out => en1

);

debounce2: entity work.debounce

port map(

clk => clk,

rst => '0',

d\_in => button2,

q\_out => en2

);

debounce3: entity work.debounce

port map(

clk => clk,

rst => '0',

d\_in => button3,

q\_out => en3

);

debounce4: entity work.debounce

port map(

clk => clk,

rst => '0',

d\_in => button4,

q\_out => start

);

process(clk, en1)

begin

if(rising\_edge(clk)) then

if(en1 = '1') then

count1 <= count1 + '1';

end if;

end if;

end process;

with count1 select

digit <= x when "00",

y when "01",

rez(15 downto 0) when "10",

rez(31 downto 16) when "11",

x"0000" when others;

process(clk, en2)

begin

if(rising\_edge(clk)) then

if(en2 = '1') then

count2 <= count2 xor '1';

end if;

end if;

if (count2 = '0') then

x <= sw;

else

y <= sw;

end if;

end process;

process(clk, en3)

begin

if(rising\_edge(clk)) then

if(en3 = '1') then

count3 <= count3 xor '1';

end if;

end if;

if(count3 = '0') then

rez <= rez1;

finishFlag <= finishFlag1;

else

rez <= rez2;

finishFlag <= finishFlag2;

end if;

end process;

ssd: entity work.SSD

port map(

clk => clk,

digit => digit,

cat => cat,

an => an

);

inm1: entity work.inmultitor

port map(

x => x,

y => y,

start => start,

clk => clk,

rez => rez2,

finishFlag => finishFlag1

);

inm2: entity work.inmultitor1

port map(

x => x,

y => y,

start => start,

clk => clk,

rez => rez1,

finishFlag => finishFlag2

);

end Behavioral;

## 