**Utilizarea modulului Pmod JSTK**



Student: Indrumator de Proiect:

Popa Alexandru Vasile Dragos Florin Lisman

Grupa 30236

Cuprins

1.Rezumat……………………………………………………………………………….3

2.Introducere…………………………………………………………………………..4

3.Fundamenta Teoretica………………………………………………………….4

4. Proiectare şi implementare……………………………………………….…7

5. Rezultate experimentale………………………………………………………9

6. Concluzii…………………………………………………………………………….10

7. Bibliografie…………………………………………………………………………11

8. Anexe…………………………………………………………………………………12

1. Rezumat

În acest proiect, am explorat capacitatea modulului Pmod JSTK de a controla culoarea unui LED tricolor și afișarea pe un Seven-Segment Display pe placa Nexys 4 FPGA.

Obiectivul a fost integrarea eficientă a modulului Pmod JSTK cu placa Nexys 4, folosind comunicația SPI pentru a permite utilizatorului să modifice culoarea LED-ului tricolor, bazându-se pe mișcarea joystick-ului. Procesul a implicat dezvoltarea unui sistem complex utilizând VHDL, un limbaj de descriere hardware avansat, și implementarea interfeței SPI pentru a interpreta semnalele de intrare de la joystick. Această implementare pe placa Nexys 4 FPGA nu doar că demonstrează capabilitățile extinse ale modulelor Pmod.

Metoda de rezolvare a fost folosind un Finite-State-Machine cu timpi predefiniti ce sa controloeze SPI\_Master-ul(care la randul sau este un finit state machine cu 2 stari(ready,execute)

Rezultatele prezinta capacitatea de a primi si trimite date folosind un SPI\_Master si de a afisa pe tri-color LED si SSD.

Concluziile generale sunt ca aceasta comunicare este extrem de utila dar si dificil de implementat,hazardurile de sincronizare fiind foarte usor de provocat,iar afisarea/stocarea in memorie a datelor primite este cel mai usor implementata pe un stagiu diferit al FSM.

2. Introducere

Proiectul prezinta utilizarea dispozitivului PmodJSTK2 pentru a crea o interfață de introducere a datelor cu ajutorul unui joystick cu două axe și a unui buton de tragere, facilitând schimbarea tri-color ledului si afisarea pozitiei pe SSD.

Pmod JSTK2, un dispozitiv care include un joystick cu două axe și un buton de tragere, este un exemplu perfect al tehnologiei care facilitează o interfație utilizator mai avansată. Această tehnologie se dovedește a fi esențială în controlul și interacțiunea cu diverse proiecte electronice.

Scopul principal al proiectului este de a demonstra cum joystick-ul Pmod JSTK2 poate fi integrat cu placa Nexys 4 FPGA pentru a controla un LED tricolor și a afișa informații pe un display cu șapte segmente.

3. Fundamentare teoretică

Conform: <https://digilent.com/reference/_media/reference/pmod/pmodjstk2/pmodjstk2_rm.pdf>

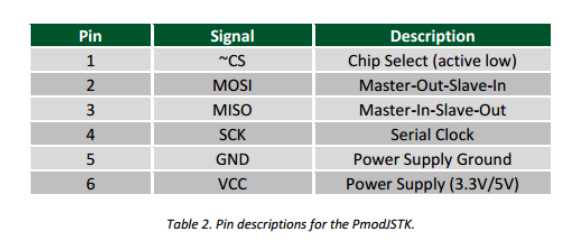
<https://digilent.com/reference/programmable-logic/nexys-4/reference-manual?redirect=1>

  
Acesta include un joystick cu două axe, un buton central, un buton de tragere și un LED RGB programabil capabil să afișeze culori de 24 de biți.

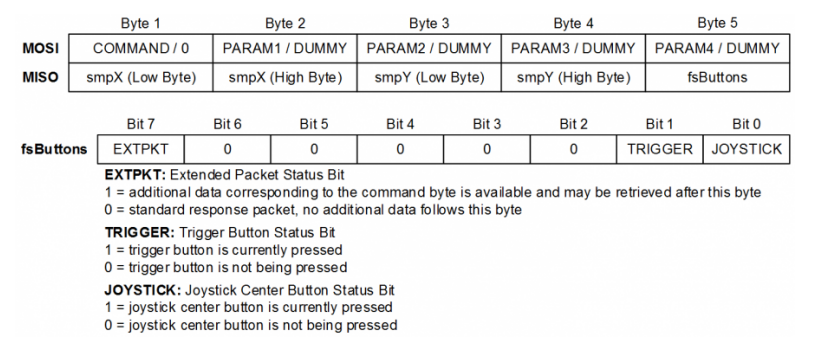
Dispozitivul utilizează două potențiometre orientate orthogonal, manipulate prin mișcarea joystick-ului în direcțiile X și Y. Datele brute măsurate sunt prealuate bit cu bit cu o rată de 0.56 MHz ca o variabilă de 8 biți .

Acest dispozitiv versatil poate fi folosit pentru controlul unor aspecte diferite ale proiectelor și permite monitorizarea și controlul poziției joystick-ului, butoanelor și culorii LED-ului RGB în timp real.

Descriere Pini:



**Functionarea tranzactiei pe bytes:**



Structura standard a pachetului de date, utilizatorii pot să trimită fie un zero și o serie de 4 dummy bytes pentru a primi cele 5 bytes de date standard, fie pot să trimită un singur byte de comandă, urmat de până la 4 parametri în cele patru bytes următoare, pentru a seta valorile interne, precum calibrarea joystick-ului sau LED-ul RGB încorporat.

Descrierea Componentelor utilizate:

**SPI Master (Master-ul Serial Peripheral Interface)**:

* Este responsabil pentru inițierea și controlul comunicațiilor SPI.
* Funcționează pe baza unui protocol care transmite date între microcontrolere și dispozitive periferice.
* Este esențial pentru sincronizarea și transmiterea datelor între placa Nexys 4 FPGA și modulul Pmod JSTK.

**FSM (Finite State Machine):**

* Administrează fluxul de operațiuni în cadrul sistemului.
* Permite schimbarea stărilor sistemului în funcție de intrări și condiții predefinite.
* Este folosit pentru a coordona procesul de comunicare și manipulare a datelor între diferitele componente ale sistemului.

**Generator SCLK (Serial Clock Generator):**

* Generează semnalul de ceas serial necesar pentru sincronizarea transferurilor SPI.
* Asigură un ritm constant pentru transmiterea datelor.
* Este crucial pentru menținerea integrității și a consistenței comunicării în cadrul sistemului.

**Tri-color LED Decoder:**

* Decodor LED Tricolor: Gestionează controlul unui LED tricolor, permțând afișarea diferitelor culori.
* Semnale de Control: Primește semnale de intrare care determină care dintre cele trei culori (roșu, verde, albastru) va fi activată.

**SSD(Seven Segment Display)**

* Controlorul Display-ului cu Șapte Segmente: Acest modul este proiectat pentru a controla un display cu șapte segmente (SSD).
* Conversia Datelor: Convertește datele de intrare binare în formatul corespunzător pentru afișarea pe SSD.

**Universal Left Shift Register(with parallel load and reset)**

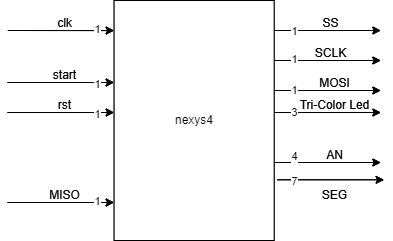
* Registrul de Deplasare la Stânga: Folosit pentru manipularea datelor în cadrul operațiilor de comunicare.
* Operarea cu Biți: Permite deplasarea bitilor de date și stocarea temporară în procesul de transmitere sau primire.

**Nexys4(main component)**

* Integrarea cu Placa Nexys 4 FPGA: Acest modul interconectează componentele hardware și software cu placa Nexys 4.
* Control Centralizat: Funcționează ca un nod central pentru coordonarea diferitelor module și funcționalități.

4. Proiectare şi implementare

Nexys4:



**FSM state types:**

Init-byte\_trans-byte\_pause-output\_res

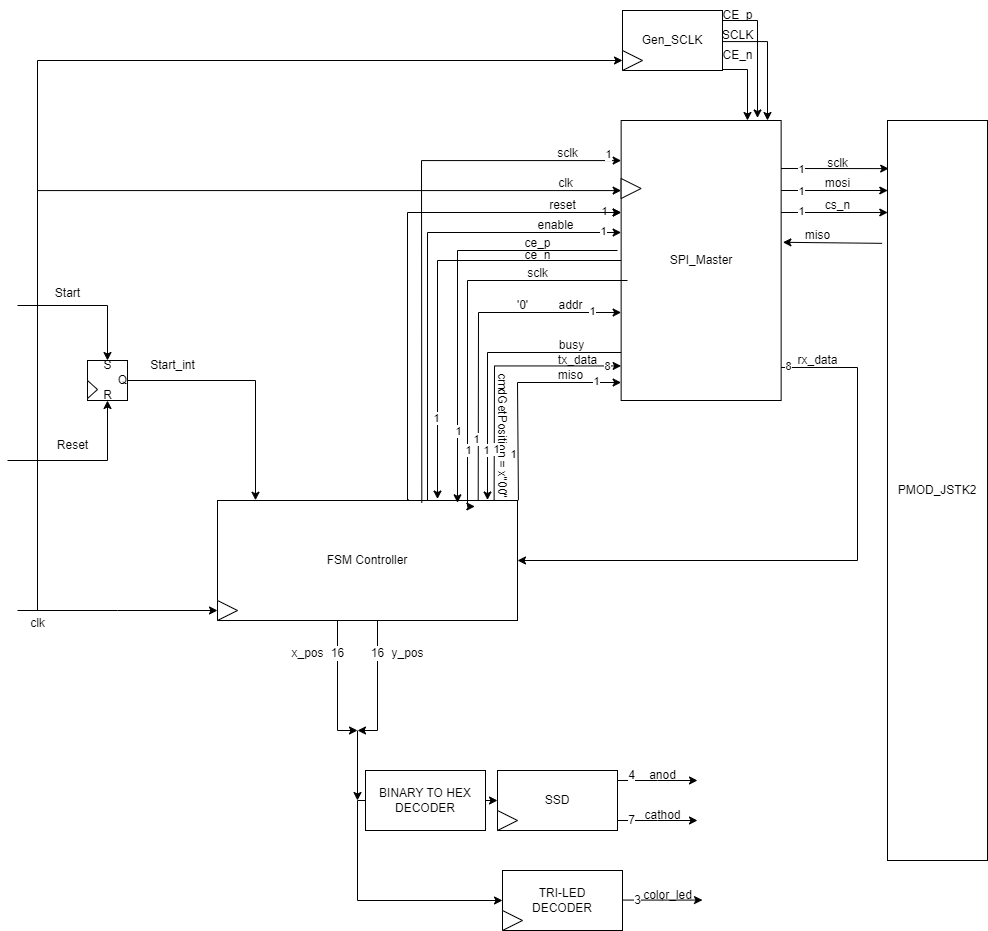
FSM se va folosi de counter si de niste valori predefinite pentru a stabili momentele in care sa preia date si in care sa le prelucreze(sa le salveze in x si y)

**SPI Master** :

* Va tot trece prin starile ready&execute pentru fiecare bit(de 4 ori)
* Va ramane pe execute pentru 8 sclk
* Va primi un semnal de reset(pentru a sterge tot si a reinitializa toate semnalele) in timpul byte\_pause, x si y fiind preluate pe ultimul clk din byte\_trans

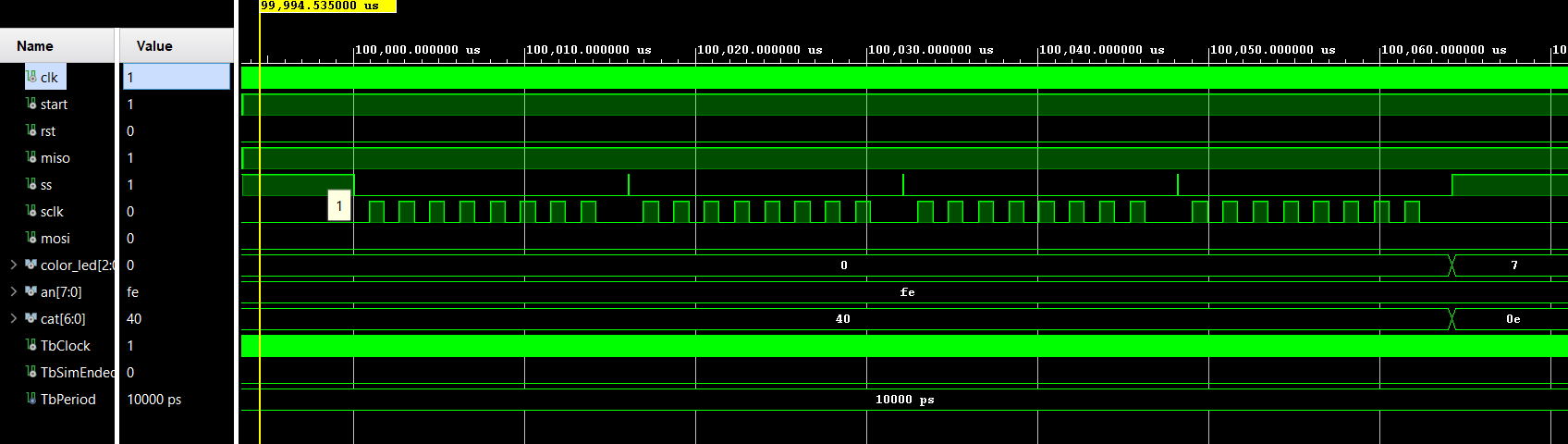
**Probleme Intampinate**:

Pe langa probleme generate de vivado(clk ibuf incorrect placement(a trebuit sa sterg chk\_pointul de fiecare data cand generam bit stream-ul) am intampinat o serie de probleme cu sincronizarea sclk si semnalelor ce\_p si ce\_n dar si in spi\_master am avut probleme cu generarea byte-ului deoarece eu foloseam sclk ca si semnal pentru state machine deci aveam nevoie de 9 sclk-uri pentru a primi un byte dar am reparat-o folosind semnalul ce\_p si ce\_n.

**Schematic**:

5. Rezultate experimentale

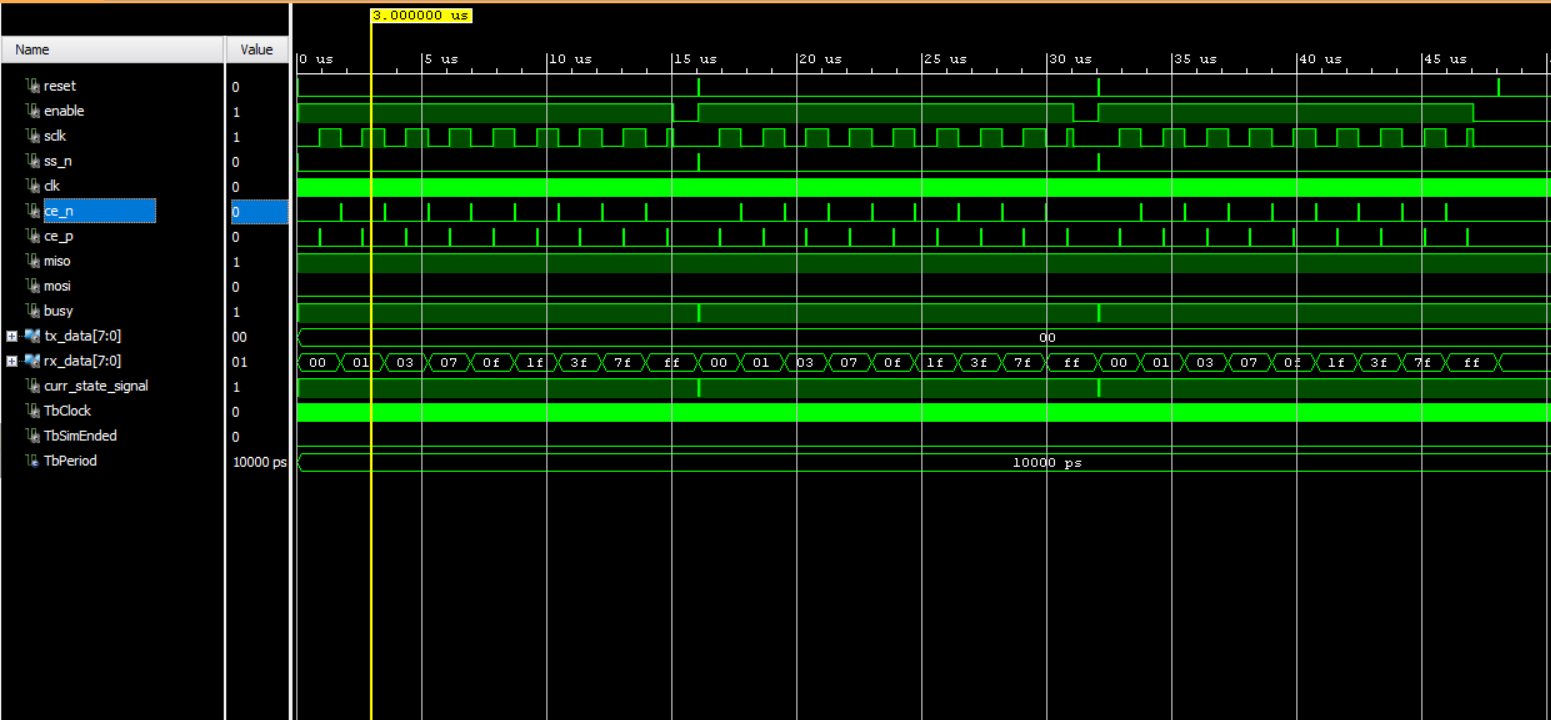
Simulare nexys4 cu toate componentele:



Simulare FSM cu date de iesire additionale:



Simulare spi\_master cu date de iesire additionale:



Din pacate nu am reusit sa fac o testare documentate si sigura pe placuta, dar sunt partial sigur ca ar trebuii sa functionezem, am pus constrangere si am rulat programul dar nu am verificat daca datele afisate pe SSD sunt corect(sau tri color ledurile)

6. Concluzii

În acest proiect, am abordat cu succes problema controlului unui LED tricolor și afișarea informațiilor pe un display cu șapte segmente folosind modulul Pmod JSTK pe placa Nexys 4 FPGA. Am atins obiectivele principale ale proiectului, care erau de a integra eficient modulul Pmod JSTK și de a permite utilizatorului să controleze LED-ul și să afișeze informații bazate pe mișcarea joystick-ului.

Contributii Personale : Dezvoltarea unui sistem complex utilizând limbajul VHDL și implementarea interfeței SPI pentru interpretarea semnalelor de la joystick.

Avantajele proiectului includ capacitatea de a controla un LED tricolor și de a afișa date pe un display cu șapte segmente, oferind o interfață utilizator avansată. Cu toate acestea, dezavantajele pot include dificultatea implementării și necesitatea de a evita hazardurile de sincronizare în timpul comunicării.

Posibile devoltarii include folosirea continua a joystick-ului si o implementare mai failsafe pentru receptarea datelor.

7.Bibliografie

* *https://digilent.com/reference/\_media/reference/pmod/pmodjstk/pmodjstk\_sch.pdf -- pmod schematic*
* *https://digilent.com/reference/learn/programmable-logic/tutorials/pmod-ips/start -- pmod basics*
* *https://forum.digikey.com/t/joystick-jstk2-pmod-controller-vhdl --pmod jstk basic & pin descriptions*
* *https://shop.trenz-electronic.de/en/Products/Digilent/Peripheral-Modules/ -- basic tutorial pmod*
* *https://github.com/CCChanhan/Pmod\_JSTK\_and\_bluetooth -- proiect pmod jstk bluetooth*
* *https://web.itu.edu.tr/~yalcinmust/EHB432/2017/Project\_S\_Report.pdf -- raport VGA folosind pmod jstk*
* *https://www.electrokit.com/uploads/productfile/41010/PmodJSTK\_rm\_RevC.pdf --alt manual -- jstk vechi*
* *https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi/all -- SPI*

8.Anexa pentru codul sursa:

SCLK: (generatorul de frecventa pentru SCLK si semnalele ce\_n si ce\_p, componenta esentiala pentru comunicare)

entity SCLK\_Gen is

Port (

clk : in STD\_LOGIC;

reset : in STD\_LOGIC;

enable : in STD\_LOGIC;

sclk : out STD\_LOGIC;

CE\_p : out STD\_LOGIC;

CE\_n : out STD\_LOGIC

);

end SCLK\_Gen;

architecture Behavioral of SCLK\_Gen is

constant SYS\_CLK\_FREQUENCY : INTEGER := 100\_000\_000;

constant SCLK\_TARGET\_FREQUENCY : INTEGER := 565\_000;

signal counter : INTEGER := 0;

signal clock\_period : INTEGER := integer(SYS\_CLK\_FREQUENCY / SCLK\_TARGET\_FREQUENCY);

signal CE\_p\_internal : STD\_LOGIC := '0';

signal CE\_n\_internal : STD\_LOGIC := '0';

signal sclk\_internal : STD\_LOGIC := '0';

begin

process(clk, reset)

begin

if reset = '1' then

counter <= 0;

CE\_p\_internal <= '0';

CE\_n\_internal <= '0';

sclk\_internal <= '0';

elsif rising\_edge(clk) then

if enable = '1' then

counter <= counter + 1;

-- sclk and CE\_p go high at the same time

if counter = (clock\_period / 2) - 1 then

sclk\_internal <= '1'; -- sclk goes high

CE\_p\_internal <= '1'; -- CE\_p goes high

elsif counter = (clock\_period / 2) then

CE\_p\_internal <= '0'; -- CE\_p goes l

-- sclk and CE\_n go low at the same time

elsif counter = clock\_period - 1 then

sclk\_internal <= '0'; -- sclk goes low

CE\_n\_internal <= '1'; -- CE\_n goes high

elsif counter = clock\_period then

counter <= 0; -- Reset the counter for the next cycle

CE\_p\_internal <= '0'; -- Reset CE\_p

CE\_n\_internal <= '0'; -- Reset CE\_n

end if;

else

counter <= 0;

CE\_p\_internal <= '0';

CE\_n\_internal <= '0';

sclk\_internal <= '0';

end if;

end if;

end process;

-- Output assignments

sclk <= sclk\_internal;

CE\_p <= CE\_p\_internal;

CE\_n <= CE\_n\_internal;

end Behavioral;

Left\_Shift\_Register : (componenta esentiala pentru stocarea semnalelor ce trebuie transmise si receptate)

entity left\_shift\_reg is

Port ( Sin : in STD\_LOGIC;

Sout : out STD\_LOGIC:= '0';

D : in STD\_LOGIC\_VECTOR (7 downto 0);

Q : out STD\_LOGIC\_VECTOR (7 downto 0);

clk : in STD\_LOGIC;

load : in STD\_LOGIC;

left\_shift : in STD\_LOGIC);

end left\_shift\_reg;

architecture Behavioral of left\_shift\_reg is

signal value\_stored : std\_logic\_vector(7 downto 0) := "00000000";

begin

process(clk)

begin

if rising\_edge(clk) then

if load = '1' then

value\_stored <= D;

elsif left\_shift = '1' then

value\_stored <= value\_stored(6 downto 0) & Sin;

Sout <= value\_stored(7);

end if;

end if;

end process;

Q <= value\_stored;

end Behavioral;

SPI Master(componenta de controleaza comunicarea bitilor si a semnalelor busy/ss\_n)

entity SPI\_Master is

Port (

reset : in STD\_LOGIC;

enable : in STD\_LOGIC;

sclk : out STD\_LOGIC;

ss\_n : out STD\_LOGIC;

clk : in STD\_LOGIC;

ce\_n : out STD\_LOGIC;

ce\_p : out STD\_LOGIC;

miso : in STD\_LOGIC;

mosi : out STD\_LOGIC := '0';

busy : out STD\_LOGIC;

tx\_data : in std\_logic\_vector(7 downto 0);

rx\_data : out std\_logic\_vector(7 downto 0);

curr\_state\_signal : out STD\_LOGIC

);

end SPI\_Master;

architecture Behavioral of SPI\_Master is

component left\_shift\_reg is

Port ( Sin : in STD\_LOGIC;

Sout : out STD\_LOGIC:= '0';

D : in STD\_LOGIC\_VECTOR (7 downto 0);

Q : out STD\_LOGIC\_VECTOR (7 downto 0);

clk : in STD\_LOGIC;

load : in STD\_LOGIC;

left\_shift : in STD\_LOGIC);

end component;

component SCLK\_Gen is

Port (

clk : in STD\_LOGIC;

reset : in STD\_LOGIC;

enable : in STD\_LOGIC; -- Add enable

sclk : out STD\_LOGIC;

CE\_p : out STD\_LOGIC;

CE\_n : out STD\_LOGIC

);

end component;

-- SPI MASTER

type machine is (ready, execute);

signal curr\_state, next\_state : machine := ready; -- Initialize the current state

signal Sin\_temp : std\_logic;

signal sclk\_signal : std\_logic := '1';

signal LdTxRx, ShTxRx : std\_logic;

signal counter : integer range 0 to 8 := 0;

signal busy\_internal : std\_logic := '0';

signal enable\_signal : std\_logic := '0';

-- SCLK\_GEN

constant SYS\_CLK\_FREQUENCY : INTEGER := 100\_000\_000; -- system clock frequency (100 MHz)

constant SCLK\_TARGET\_FREQUENCY : INTEGER := 1\_000\_000; -- SCLK frequency (4 MHz)

signal counter\_sclk : INTEGER := 0;

signal clock\_period : INTEGER := integer(SYS\_CLK\_FREQUENCY / (SCLK\_TARGET\_FREQUENCY));

signal CE\_p\_internal : STD\_LOGIC := '0';

signal CE\_n\_internal : STD\_LOGIC := '0';

signal sclk\_internal : STD\_LOGIC := '0';

-- LEFT SHIFT REG

signal value\_stored : std\_logic\_vector(7 downto 0) := "00000000";

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

signal reset\_signal,reset\_clk : std\_logic := '0';

signal rx\_data\_signal : std\_logic\_vector(7 downto 0) := "00000000";

begin

STATE\_LOGIC : process(clk, reset,enable)

begin

if reset = '1' or enable = '0' then

counter <= 0;

LdTxRx <= '1';

reset\_signal <= '1';

next\_state <= ready;

elsif rising\_edge(clk) then

curr\_state <= next\_state;

case curr\_state is

when ready =>

busy\_internal <= '0';

ShTxRx <= '0';

LdTxRx <= '1';

if enable = '1' and rx\_data\_signal = "00000000" and counter = 0 then

enable\_signal <= '1';

next\_state <= execute;

end if;

when execute =>

busy\_internal <= '1';

ShTxRx <= '1';

LdTxRx <= '0';

if ce\_p\_internal = '1' and counter < 7 then

counter <= counter + 1;

elsif ce\_n\_internal = '1' and counter = 7 then

counter <= 0;

ShTxRx <= '0';

elsif enable = '0' then

next\_state <= ready;

end if;

end case;

end if;

end process;

process(ce\_p\_internal)

begin

if rising\_edge(ce\_p\_internal) then

Sin <= miso;

end if;

end process;

ss\_n <= not busy\_internal;

curr\_state\_signal <= '0' when curr\_state = ready else '1';

busy <= busy\_internal;

sclk <= sclk\_signal;

ce\_p <= CE\_p\_internal;

ce\_n <= CE\_n\_internal;

--SCLK

clock\_gen: SCLK\_Gen port map (

enable => enable,

clk => clk,

reset => reset, -- Reset signal

sclk => sclk\_signal, -- connect to sclk of SPI\_Master

CE\_p => CE\_p\_internal, -- connect to ce\_p of SPI\_Master

CE\_n => CE\_n\_internal -- connect to ce\_n of SPI\_Master

);

--LEFT SHIFT

reset\_clk <= ce\_n\_internal or reset;

left\_shift\_component: left\_shift\_reg port map (

Sin => Sin,

Sout => mosi,

D => tx\_data,

Q => rx\_data\_signal,

clk => reset\_clk,

load => LdTxRx,

left\_shift => ShTxRx

);

rx\_data<= rx\_data\_signal;

end Behavioral;

FSM ☹componenta ce controleaza semnalele de pornire si vectorii de semnale comunicati folosind constante de timp si numaratoare si masini de stari)

entity FSM is

Port ( start : in STD\_LOGIC;

reset : in STD\_LOGIC;

clk : in STD\_LOGIC;

sclk : out STD\_LOGIC;

miso : in STD\_LOGIC;

mosi : out STD\_LOGIC;

x : out STD\_LOGIC\_VECTOR(15 downto 0):=(others => '0');

y : out STD\_LOGIC\_VECTOR(15 downto 0):=(others => '0');

ss\_n : out STD\_LOGIC);

end FSM;

architecture Behavioral of FSM is

component SPI\_Master is

Port (

reset : in STD\_LOGIC;

enable : in STD\_LOGIC;

sclk : out STD\_LOGIC;

ss\_n : out STD\_LOGIC;

clk : in STD\_LOGIC;

ce\_n : out STD\_LOGIC;

ce\_p : out STD\_LOGIC;

miso : in STD\_LOGIC;

mosi : out STD\_LOGIC := '0';

busy : out STD\_LOGIC;

tx\_data : in std\_logic\_vector(7 downto 0);

rx\_data : out std\_logic\_vector(7 downto 0)

);

end component;

type state\_machine is (init, byte\_trans, byte\_pause, output\_res);

signal curr\_state : state\_machine := init;

signal next\_state : state\_machine := init;

signal byte\_counter : integer := 4;

signal x\_out,y\_out : std\_logic\_vector(15 downto 0);

signal byte\_recieved : std\_logic\_vector(7 downto 0);

signal enable,output : std\_logic;

signal busy\_signal,ss\_n\_signal,sclk\_signal,reset\_signal : std\_logic;

-- signals for timing

signal counter : integer := 0;

constant STARTUP\_TIME : Integer := 10\_000\_000;

constant BYTE\_TIME : Integer := 1\_500;

constant PAUSE\_TIME : INTEGER := 100;

begin

SPI\_Instance: SPI\_Master

port map (

reset => reset\_signal,

enable => enable,

sclk => sclk\_signal,

ss\_n => ss\_n\_signal,

clk => clk,

miso => miso,

mosi => mosi,

busy => busy\_signal,

tx\_data => "00000000",

rx\_data => byte\_recieved

);

sclk <= sclk\_signal;

ss\_n <= ss\_n\_signal or output;

process(clk, reset,start)

begin

if reset = '1' then

-- Reset all relevant signals

reset\_signal <= '1';

counter <= 0;

enable <= '0';

output <= '0';

curr\_state <= init;

byte\_counter <= 4;

x\_out <= (others => '0');

y\_out <= (others => '0');

elsif rising\_edge(clk) and start = '1' then

curr\_state <= next\_state;

case curr\_state is

when init =>

if counter >= STARTUP\_TIME then

counter <= 0;

next\_state <= byte\_trans;

else

counter <= counter + 1;

end if;

when byte\_trans =>

if counter > BYTE\_TIME then

counter <= 0;

case byte\_counter is

when 4 => x\_out(15 downto 8) <= byte\_recieved;

when 3 => x\_out(7 downto 0) <= byte\_recieved;

when 2 => y\_out(15 downto 8) <= byte\_recieved;

when 1 => y\_out(7 downto 0) <= byte\_recieved;

when others => next\_state <= init;

end case;

reset\_signal <= '1';

next\_state <= byte\_pause;

else

reset\_signal <= '0';

enable <= '1';

counter <= counter + 1;

end if;

when byte\_pause =>

enable <= '0';

if counter > PAUSE\_TIME then

counter <= 0;

byte\_counter <= byte\_counter - 1;

if byte\_counter = 1 then

next\_state <= output\_res;

else

next\_state <= byte\_trans;

end if;

else

counter <= counter + 1;

end if;

when output\_res =>

output <= '1';

x <= x\_out;

y <= y\_out;

end case;

end if;

end process;

end Behavioral;

Tri-Color LED Decoder : (componenta ce decodifica ultimii biti de la x si y pentru a afisa unul din 3 culor RGB)

entity tri\_color\_decoder is

Port (

clk : in STD\_LOGIC;

data\_in : in STD\_LOGIC\_VECTOR (31 downto 0);

rgb\_led\_code : out STD\_LOGIC\_VECTOR (2 downto 0)

);

end tri\_color\_decoder;

architecture Behavioral of tri\_color\_decoder is

signal counter : std\_logic\_vector(16 downto 0) := (others => '0');

signal selection : STD\_LOGIC\_VECTOR(2 downto 0);

begin

process(clk)

begin

if rising\_edge(clk) then

counter <= counter + 1;

end if;

end process;

-- Clock divider

selection <= counter(16 downto 14);

process(data\_in)

begin

-- Default values

rgb\_led\_code <= "000";

-- Check x direction (bit 31)

if data\_in(31) = '1' then

rgb\_led\_code(0) <= '1'; -- Red LED

end if;

-- Check y direction (bit 15)

if data\_in(15) = '1' then

rgb\_led\_code(1) <= '1'; -- Green LED

end if;

-- Check diagonal directions (bit 31 and bit 15)

if data\_in(31) = '1' and data\_in(15) = '1' then

rgb\_led\_code(2) <= '1'; -- Blue LED

end if;

end process;

end Behavioral;

SSD(componenta ce afiseaza pe seven segment display numerele)

entity ssd is

Port

(

signal clk:in std\_logic;

signal data:in std\_logic\_vector(31 downto 0);

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

signal an:out std\_logic\_vector(7 downto 0)

);

end ssd;

architecture Behavioral of ssd is

signal counter: std\_logic\_vector(16 downto 0);

signal selection:std\_logic\_vector(2 downto 0);

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

begin

process(clk)

begin

if clk='1' and clk'event then

counter<=counter+1;

end if;

end process;

selection<=counter(16 downto 14);

process(selection,data)

begin

case selection is

when "000" => an<="01111111";decoder<=data(31 downto 28);

when "001" => an<="10111111";decoder<=data(27 downto 24);

when "010" => an<="11011111";decoder<=data(23 downto 20);

when "011" => an<="11101111";decoder<=data(19 downto 16);

when "100" => an<="11110111";decoder<=data(15 downto 12);

when "101" => an<="11111011";decoder<=data(11 downto 8);

when "110" => an<="11111101";decoder<=data(7 downto 4);

when others =>an<="11111110";decoder<=data(3 downto 0);

end case;

end process;

process(decoder)

begin

case decoder is

when "0000" =>cat<="1000000"; -- 0

when "0001" =>cat<="1111001"; -- 1

when "0010" =>cat<="0100100"; -- 2

when "0011" =>cat<="0110000"; -- 3

when "0100" =>cat<="0011001"; -- 4

when "0101" =>cat<="0010010"; -- 5

when "0110" =>cat<="0000010"; -- 6

when "0111" =>cat<="1111000"; -- 7

when "1000" =>cat<="0000000"; -- 8

when "1001" =>cat<="0010000"; -- 9

when "1010" =>cat<="0001000"; -- A

when "1011" =>cat<="0000011"; -- B

when "1100" =>cat<="1000110"; -- C

when "1101" =>cat<="0100001"; -- D

when "1110" =>cat<="0000110"; -- E

when others =>cat<="0001110"; -- F

end case;

end process;

end Behavioral;

nexys4:

entity nexys4 is

Port ( clk : in STD\_LOGIC;

start : in STD\_LOGIC;

rst : in STD\_LOGIC;

miso : in STD\_LOGIC;

ss : out STD\_LOGIC;

sclk : out STD\_LOGIC;

mosi : out STD\_LOGIC;

color\_led : out std\_logic\_vector (2 downto 0);

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

an : out std\_logic\_vector (7 downto 0);

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

end nexys4;

architecture Behavioral of nexys4 is

component FSM is

Port ( start : in STD\_LOGIC;

reset : in STD\_LOGIC;

clk : in STD\_LOGIC;

sclk : out STD\_LOGIC;

miso : in STD\_LOGIC;

mosi : out STD\_LOGIC;

x : out STD\_LOGIC\_VECTOR(15 downto 0):=(others => '0');

y : out STD\_LOGIC\_VECTOR(15 downto 0):=(others => '0');

ss\_n : out STD\_LOGIC);

end component;

component ssd is

Port (

clk: in std\_logic;

data: in std\_logic\_vector(31 downto 0);

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

an: out std\_logic\_vector(7 downto 0)

);

end component;

component tri\_color\_decoder is

Port (

clk : in STD\_LOGIC;

data\_in : in STD\_LOGIC\_VECTOR(31 downto 0);

rgb\_led\_code : out STD\_LOGIC\_VECTOR(2 downto 0)

);

end component;

signal sclk\_internal : STD\_LOGIC := '0'; -- Internal signal for sclk

signal x\_output, y\_output : STD\_LOGIC\_VECTOR(15 downto 0);

signal data\_output : std\_logic\_vector (31 downto 0);

signal rgb\_led\_output : STD\_LOGIC\_VECTOR(2 downto 0);

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

signal an\_output : STD\_LOGIC\_VECTOR(7 downto 0);

signal start\_signal,rst\_signal : std\_logic:= '0';

begin

FSM\_Instance: FSM

port map (

start => start\_signal,

reset => rst\_signal,

clk => clk,

sclk => sclk\_internal,

miso => miso,

mosi => mosi,

x => x\_output,

y => y\_output,

ss\_n => ss

);

data\_output <= x\_output & y\_output;

data <= data\_output;

ssd\_Instance: ssd

port map (

clk => clk,

data => data\_output,

cat => cat\_output,

an => an\_output

);

tri\_color\_decoder\_Instance: tri\_color\_decoder

port map (

clk => clk,

data\_in => data\_output,

rgb\_led\_code => rgb\_led\_output

);

start\_signal <= start;

rst\_signal <= rst;

sclk<= sclk\_internal;

cat <= cat\_output;

an <= an\_output;

color\_led <= rgb\_led\_output;

end Behavioral;