Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

1η Σειρά Ασκήσεων

Μάθημα: Ψηφιακά Συστήματα VLSI

Εξάμηνο: 8ο

Ονοματεπώνυμα: Αλεξοπούλου Γεωργία, Γκενάκου Ζωή

Θέμα Α.2: Δυαδικός αποκωδικοποιητής 3 σε 8

Σε αυτό το ερώτημα μας ζητείται η περιγραφή της οντότητας του δυαδικού αποκωδικοποιητή 3 σε 8 και της αρχιτεκτονικής του σε DataFlow και Behavioral VHDL. Επίσης μας ζητείται να γίνει η προσομοίωση και στις δύο αρχιτεκτονικές.

Για την αναπαραγωγή του κώδικα βασιστήκαμε στον παρακάτω πίνακα αληθείας:

input			**output**							
enc(2)	*enc(1)*	*enc(0)*	*dec(7)*	*dec(6)*	*dec(5)*	*dec(4)*	*dec(3)*	*dec(2)*	*dec(1)*	*dec(0)*
0	**0**	**0**	0	0	0	0	0	0	0	**1**
0	**0**	**1**	0	0	0	0	0	0	**1**	0
0	**1**	**0**	0	0	0	0	0	**1**	0	0
0	**1**	**1**	0	0	0	0	**1**	0	0	0
1	**0**	**0**	0	0	0	**1**	0	0	0	0
1	**0**	**1**	0	0	**1**	0	0	0	0	0

1	**1**	**0**	0	**1**	0	0	0	0	0	0
1	**1**	**1**	**1**	0	0	0	0	0	0	0

Ο κώδικας για Behavioral:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity decoder_3_to_8_bhv is
    Port (
        enc : in std_logic_vector (2 downto 0);
        dec : out std_logic_vector (7 downto 0)
    );
end decoder_3_to_8_bhv;

architecture Behavioral of decoder_3_to_8_bhv is

begin
    dec3to8 : process(enc)
    begin
        case enc is
            when "000" => dec <= "00000001";
            when "001" => dec <= "00000010";
            when "010" => dec <= "00000100";
            when "011" => dec <= "00001000";
            when "100" => dec <= "00010000";
            when "101" => dec <= "00100000";
            when "110" => dec <= "01000000";
            when "111" => dec <= "10000000";
            when others => dec <= "00000000";
        end case;
    end process;

end Behavioral;
```

Για την behavioral αρχιτεκτονική χρησιμοποιήθηκε η δομή case, όπως και αναγραφόταν στην υπόδειξη.

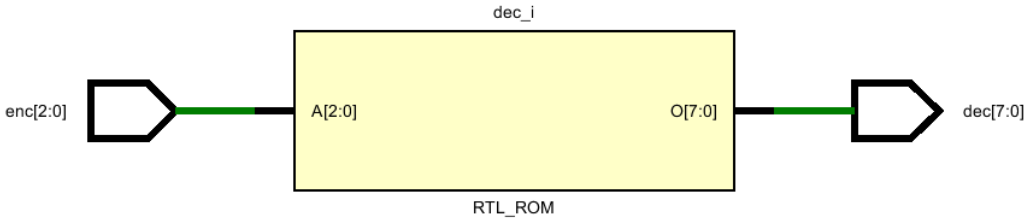
Και ο κώδικας για DataFlow:

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

use IEEE.NUMERIC_STD.ALL;

entity decoder_3_to_8_df is
    Port (
        enc : in std_logic_vector(2 downto 0);
        dec : out std_logic_vector(7 downto 0)
    );

end decoder_3_to_8_df;

architecture Dataflow of decoder_3_to_8_df is

begin
    dec(7) <= enc(2) and enc(1) and enc(0);
    dec(6) <= enc(2) and enc(1) and not enc(0);
    dec(5) <= enc(2) and not enc(1) and enc(0);
    dec(4) <= enc(2) and not enc(1) and not enc(0);
    dec(3) <= not enc(2) and enc(1) and enc(0);
    dec(2) <= not enc(2) and enc(1) and not enc(0);
    dec(1) <= not enc(2) and not enc(1) and enc(0);
    dec(0) <= not enc(2) and not enc(1) and not enc(0);

end Dataflow;
```

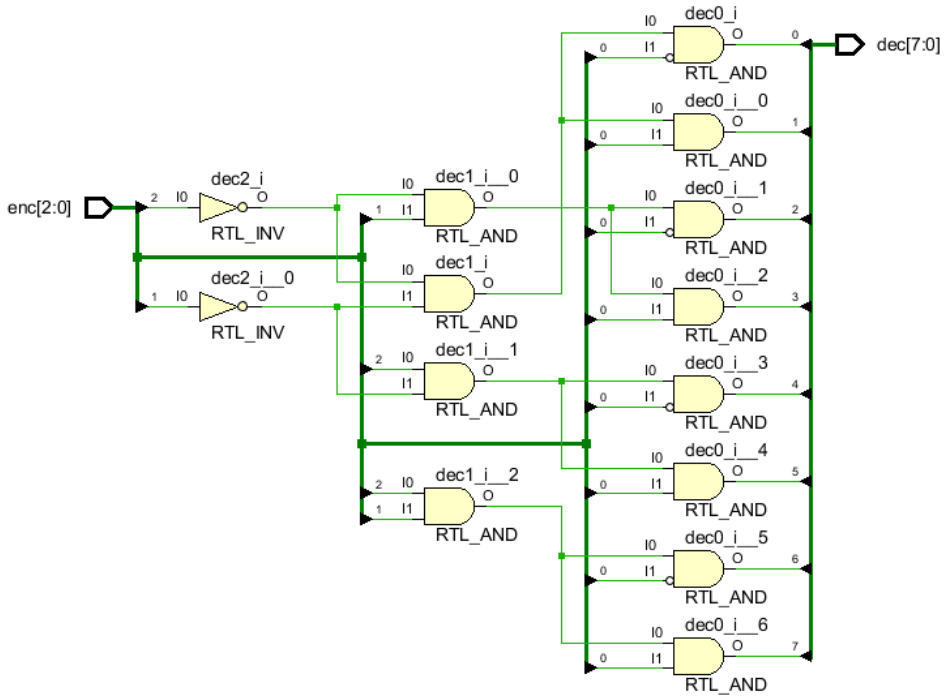
- RTL Analysis

Η ανάλυση RTL περιλαμβάνει την εξέταση της συμπεριφοράς και των χαρακτηριστικών ενός ψηφιακού σχεδίου σε επίπεδο Register-Transfer Level (RTL). Σε επίπεδο RTL, η σχεδίαση περιγράφεται ως προς τους καταχωρητές και τις μεταφορές δεδομένων μεταξύ τους. Η ανάλυση RTL επικεντρώνεται στην επαλήθευση ότι ο σχεδιασμός πληροί τις λειτουργικές απαιτήσεις, τους χρονικούς περιορισμούς και άλλες προδιαγραφές.

Behavioral:



DataFlow:

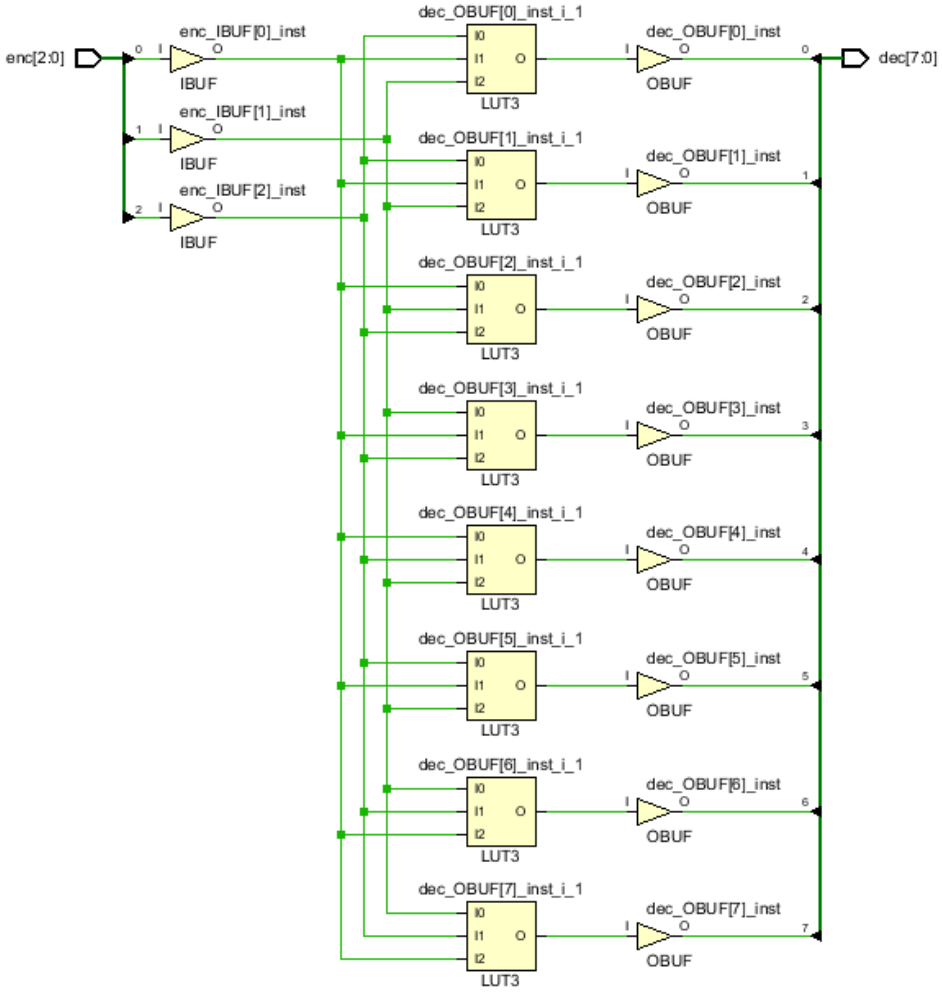


Η RTL ανάλυση στην Behavioral αρχιτεκτονική περιγράφει τη λειτουργικότητα του συστήματος χωρίς να προσδιορίζεται πώς εφαρμόζεται σε χαμηλό επίπεδο. Από την άλλη, η ανάλυση στο DataFlow εστιάζει στον τρόπο ροής δεδομένων μέσω του συστήματος και στον τρόπο εκτέλεσης των λειτουργιών με βάση τη διαθεσιμότητα των δεδομένων. Για αυτό τον λόγο, ενώ και οι δύο αρχιτεκτονικές στοχεύουν

τελικά να περιγράψουν την ίδια λειτουργικότητα του συστήματος, η ανάλυση RTL για κάθε μια θα διαφέρει λόγω των διαφορετικών προσεγγίσεων μοντελοποίησης.

- Synthesis

Σε αυτό το σημείο και οι δύο αρχιτεκτονικές παράγουν τα ίδια αποτελέσματα.



- Στην συνέχεια γράφουμε ένα test bench κοινό και για τις δύο υλοποιήσεις (διαφέρουν μόνο στο όνομα της αρχιτεκτονικής):

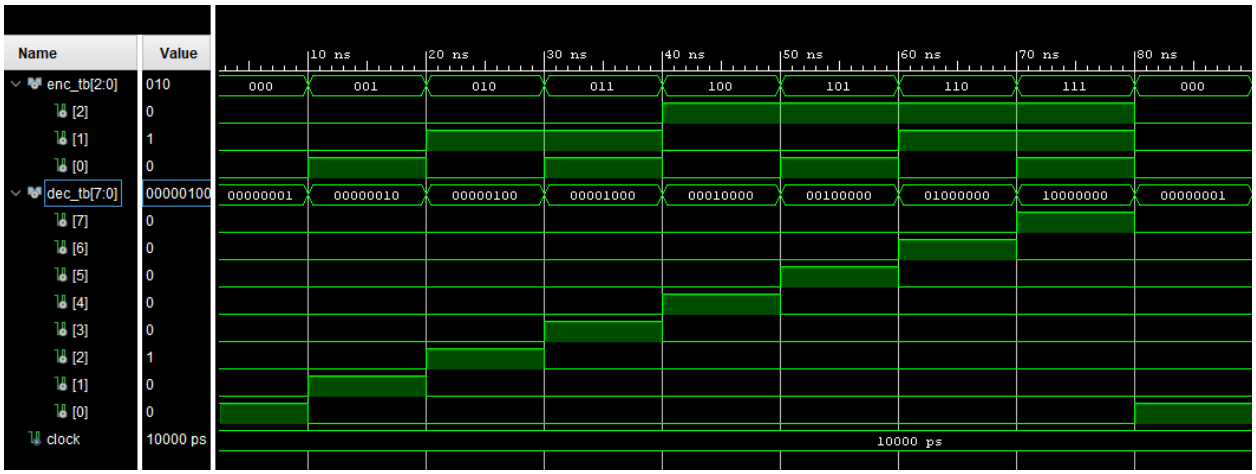
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity decoder_3_to_8_tb is
--  Port ( );
```

```
end decoder_3_to_8_tb;

architecture Behavioral of decoder_3_to_8_tb is

component decoder_3_to_8_bhv is
    Port (
        enc : in std_logic_vector (2 downto 0);
        dec : out std_logic_vector (7 downto 0)
    );
end component;

constant clock : time := 10ns;

signal enc_tb : std_logic_vector(2 downto 0);
signal dec_tb : std_logic_vector(7 downto 0);

begin

    dut : decoder_3_to_8_bhv
        port map (
            enc => enc_tb,
            dec => dec_tb
        );

    simulation : process
    begin
        enc_tb <= "000";
        wait for clock;
        enc_tb <= "001";
        wait for clock;
        enc_tb <= "010";
        wait for clock;
        enc_tb <= "011";
        wait for clock;
        enc_tb <= "100";
        wait for clock;
        enc_tb <= "101";
        wait for clock;
        enc_tb <= "110";
```

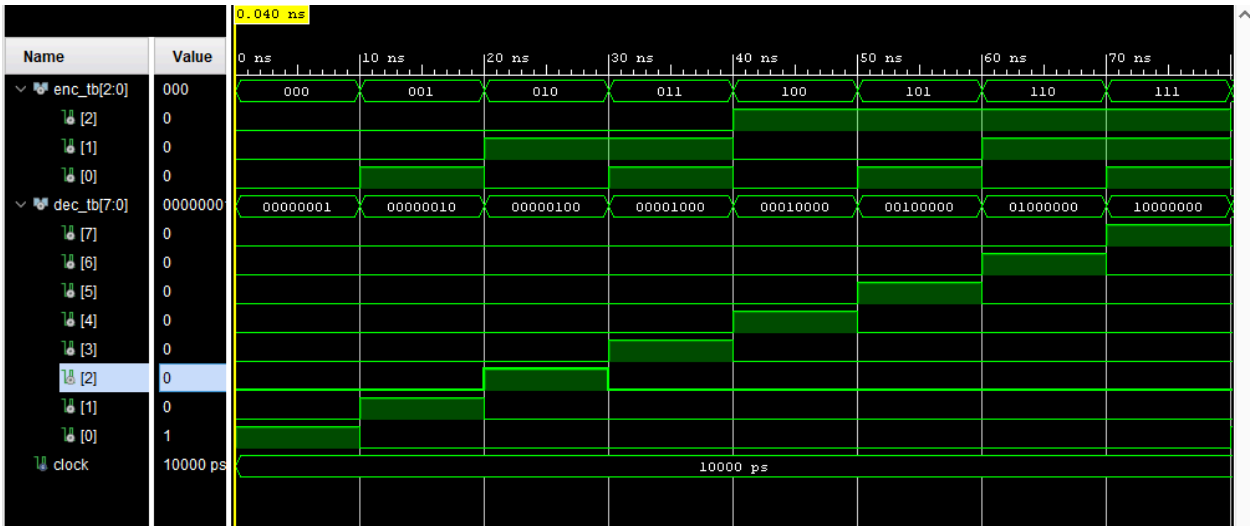
```
        wait for clock;
        enc_tb <= "111";
        wait for clock;

    end process simulation;


end Behavioral;
```

Τα αποτελέσματα του Simulation φαίνονται παρακάτω:

Για το Behavioral:



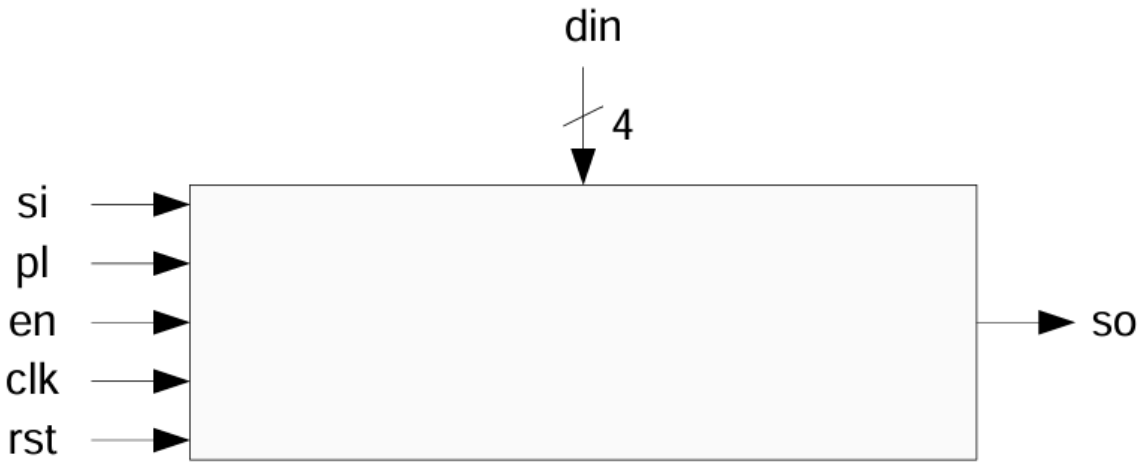
Και για Dataflow Architecture:



Όπως ήταν αναμενόμενο και οι δύο αρχιτεκτονικές αποφέρουν το ίδιο αποτέλεσμα και μπορούμε επίσης να επιβεβαιώσουμε και την ορθή λειτουργία και των δύο υλοποιήσεων.

Θέμα Β.2: Καταχωρητής ολίσθησης των 4 bits με παράλληλη φόρτωση

Στο θέμα Β.2 ζητείται η υλοποίηση ενός 4-bit καταχωρητή ολίσθησης με παράλληλη φόρτωση, με δυνατότητα ολίσθησης και προς τις δύο κατευθύνσεις.
Μας δίνεται ο κώδικας για την αρχιτεκτονική της παρακάτω οντότητας:



Καθώς το παραπάνω σχήμα απεικονίζει έναν 4-bit right shift register with parallel load, θεωρήσαμε σκόπιμο η υλοποίηση της τελικής οντότητας να γίνει σε δύο στάδια, μελετώντας την ανάλυση της αρχιτεκτονικής των επί μέρους οντοτήτων: α) 4-bit right shift register και β) 4-bit left shift register.

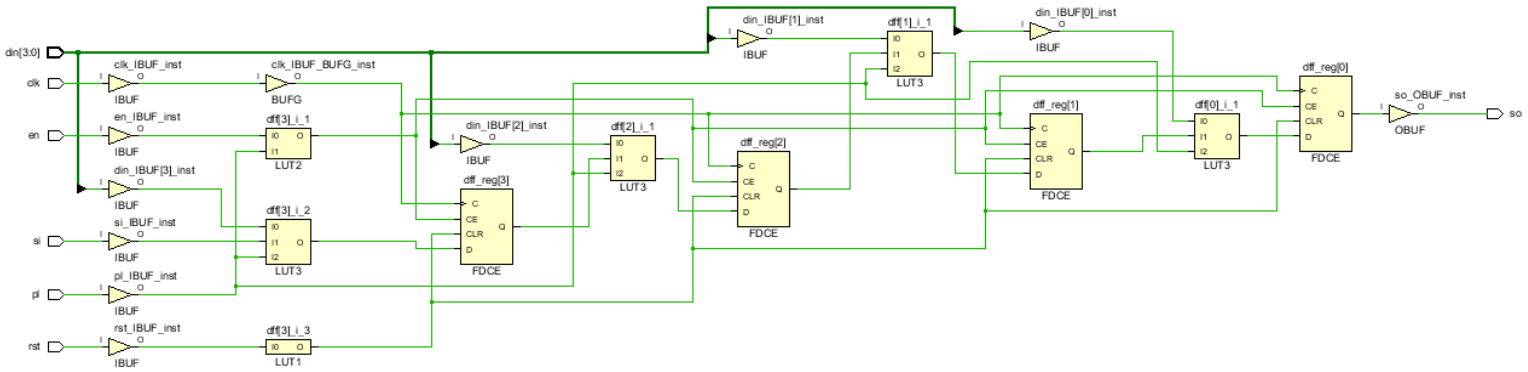
Rshift_sreg:

Ο δοσμένος κώδικας για RTL architecture είναι:

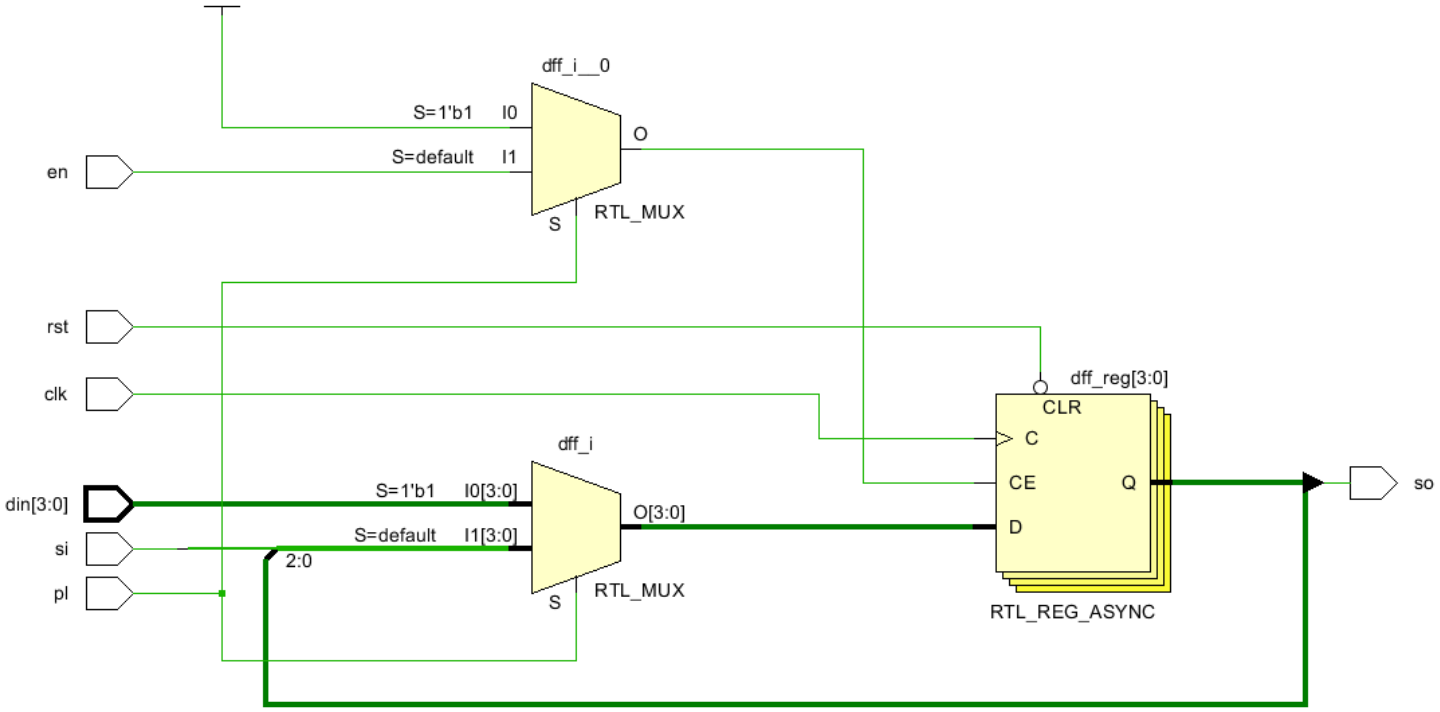
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity rshift_reg3 is
    Port (
        clk, rst, si, en, pl : in std_logic;
        din : in std_logic_vector(3 downto 0);
        so : out std_logic
        );
```

```
end entity;

architecture rtl of rshift_reg3 is
    signal dff : std_logic_vector(3 downto 0);

begin
    edge : process(clk, rst)
    begin
        if rst='0' then
            dff<= (others => '0');
        elsif clk'event and clk='1' then
            if pl='1' then
                dff <= din;
            elsif en='1' then
                dff <= si & dff(3 downto 1);
            end if;
        end if;
    end process;
    so <= dff(0);

end rtl;
```

Παρατηρούμε πως τα schematics των Synthesis και Implementation ταυτίζονται:



Προκύπτει το εξής RTL schema:



Παρατηρούμε πως οι δύο πολυπλέκτες διαχειρίζονται τα σήματα εισόδου, καθώς και τα parallel load και enabling bits, ενώ τα 4 DFFs διαχειρίζονται το right shifting και παράγουν τα σήματα εξόδου, με επιπλέον εισόδους αυτές του ρολογιού και του reset. Αν και το σχηματικό φαίνεται να εκτελεί τη ζητούμενη λειτουργία, θα εξετάσουμε την ακεραιότητα της οντότητας μέσω του testbench που φαίνεται παρακάτω.

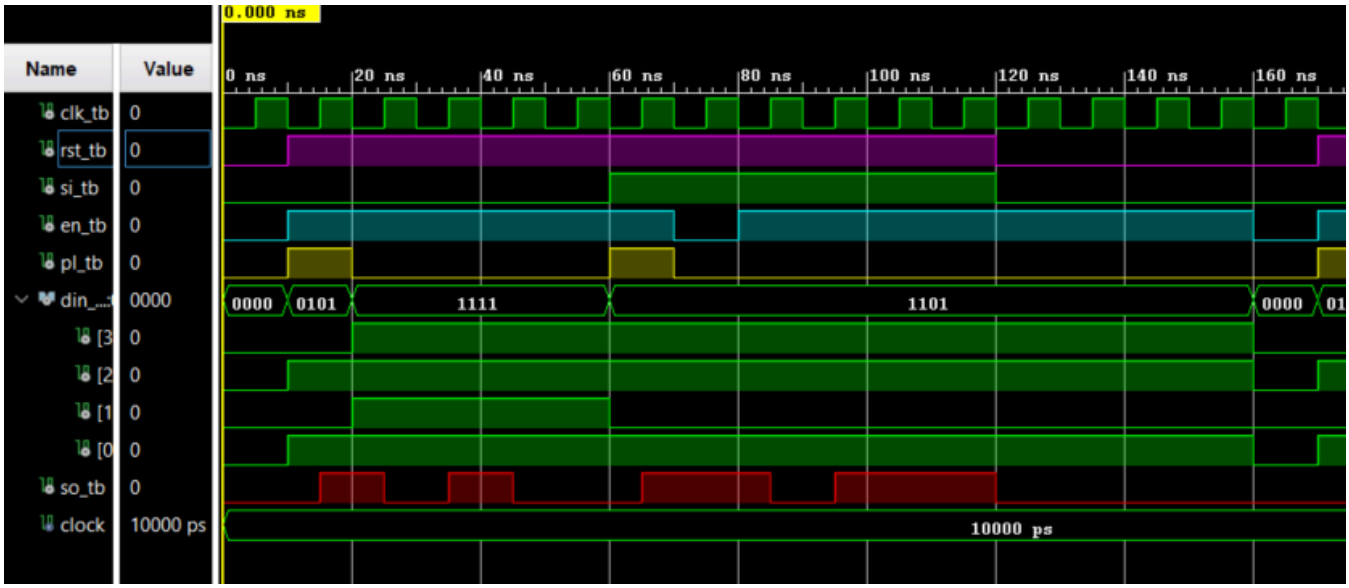
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity rshift_reg3_tb is
--  Port ( );
end rshift_reg3_tb;

architecture bench of rshift_reg3_tb is
    component rshift_reg3 is
        Port (
```

```
            clk, rst, si, en, pl : in std_logic;
            din : in std_logic_vector(3 downto 0);
            so : out std_logic
            );
    end component;

    signal clk_tb, so_tb : std_logic;
    signal rst_tb, si_tb, en_tb, pl_tb : std_logic := '0';
    signal din_tb : std_logic_vector(3 downto 0) := (others => '0');

    constant clock : time := 10 ns;

begin
    dut : rshift_reg3
        port map (
            clk => clk_tb,
            rst => rst_tb,
            si => si_tb,
            en => en_tb,
            pl => pl_tb,
            din => din_tb,
            so => so_tb
        );

    simulation : process
    begin
        -- Initialization
        rst_tb <= '0';
        pl_tb <= '0';
        en_tb <= '0';
        si_tb <= '0';
        din_tb <= "0000";
        wait for clock;

        -- Test parallel load
        rst_tb <= '1';
        pl_tb <= '1';
        en_tb <= '1';
        si_tb <= '0';
```

```
    din_tb <= "0101";
    wait for clock;

    -- Reset parallel load
    -- The new input should not be printed in the output
    rst_tb <= '1';
    pl_tb <= '0';
    en_tb <= '1';
    si_tb <= '0';
    din_tb <= "1111";
    wait for 4*clock;

    -- Test enable bit
    rst_tb <= '1';
    pl_tb <= '1';
    en_tb <= '1';
    si_tb <= '1';
    din_tb <= "1101";
    wait for clock;
    pl_tb <= '0';
    en_tb <= '0';
    wait for clock;
    en_tb <= '1';
    wait for 4*clock;

    -- Test reset
    rst_tb <= '0';
    pl_tb <= '0';
    en_tb <= '1';
    si_tb <= '0';
    wait for 4*clock;

end process;

generate_clock : process
begin
     clk_tb <= '0';
     wait for clock/2;
     clk_tb <= '1';
```

```
        wait for clock/2;
    end process ;

end bench;
```

Τα αποτελέσματα της προσομοίωσης φαίνονται παρακάτω:



Βλέπουμε πως πράγματι η λειτουργία του κώδικά μας είναι ορθή. Πιο συγκεκριμένα, βλέπουμε πως οι αλλαγές των control bits γίνονται με τη θετική ακμή του ρολογιού (`clk='1'`). Ξεκινάμε αρχικοποιώντας τις τιμές των control bits . Όταν τα reset, parallel load και enabling bits γίνονται triggered (`rst_tb='1', pl_tb='1', en_tb='1'`), τότε φορτώνεται η είσοδος `din` στο περιεχόμενο του καταχωρητή `sreg`, εξ' ου και η σειριακή έξοδος `so='1 0 1 0'`. Όταν το η είσοδος `din` μεταβάλλεται (`din='1111'`) και το parallel bit μηδενίζεται (βλ. κίτρινο χρώμα), τότε η είσοδος δεν φορτώνεται στον καταχωρητή `sreg`, Γι' αυτό και η έξοδος `so` που φαίνεται με κόκκινο χρώμα αντιστοιχεί στην πρώτη είσοδο, `din='0101'`. Έτσι, αποδεικνύεται η λειτουργικότητα της παράλληλης φόρτωσης.

Στη συνέχεια, εξετάζουμε τη λειτουργικότητα του enabling bit, θέτοντας την τιμή του ίση με 1 (βλ. γαλάζιο χρώμα). Έχοντας φορτώσει την καινούρια είσοδο `din='1101'`, Όσο το enabling bit είναι triggered, η ολίσθηση είναι ενεργοποιημένη, γι' αυτό και η έξοδος αρχικά είναι ίση με 1 (`so='1'`). Μηδενίζουμε το `en_tb` και παρατηρούμε πως η ολίσθηση "παγώνει" και η έξοδος παραμένει ίση με 1. Πράγματι, η ολίσθηση ελέγχεται επιτυχώς από το enabling bit.

Τέλος, αξίζει να ελέγξουμε την ορθότητα της λειτουργίας reset. Θέτοντας την τιμή `rst_tb='0'`, το περιεχόμενο του καταχωρητή `sreg` μηδενίζεται, εξ' ου και η μηδενική σειριακή έξοδος (`so='0'`).

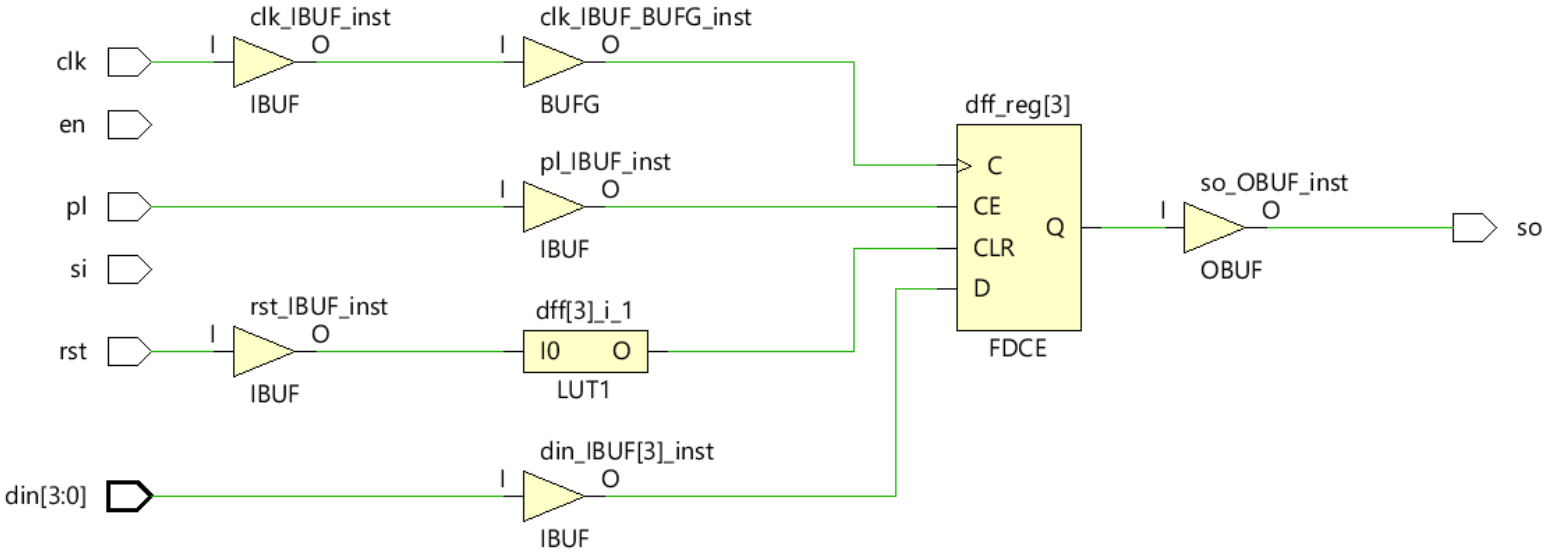
Lshift_sreg:

Στο σημείο αυτό, έχουμε τροποποιήσει τον δοσμένο κώδικα, ώστε να εκτελεί αριστερή ολίσθηση.

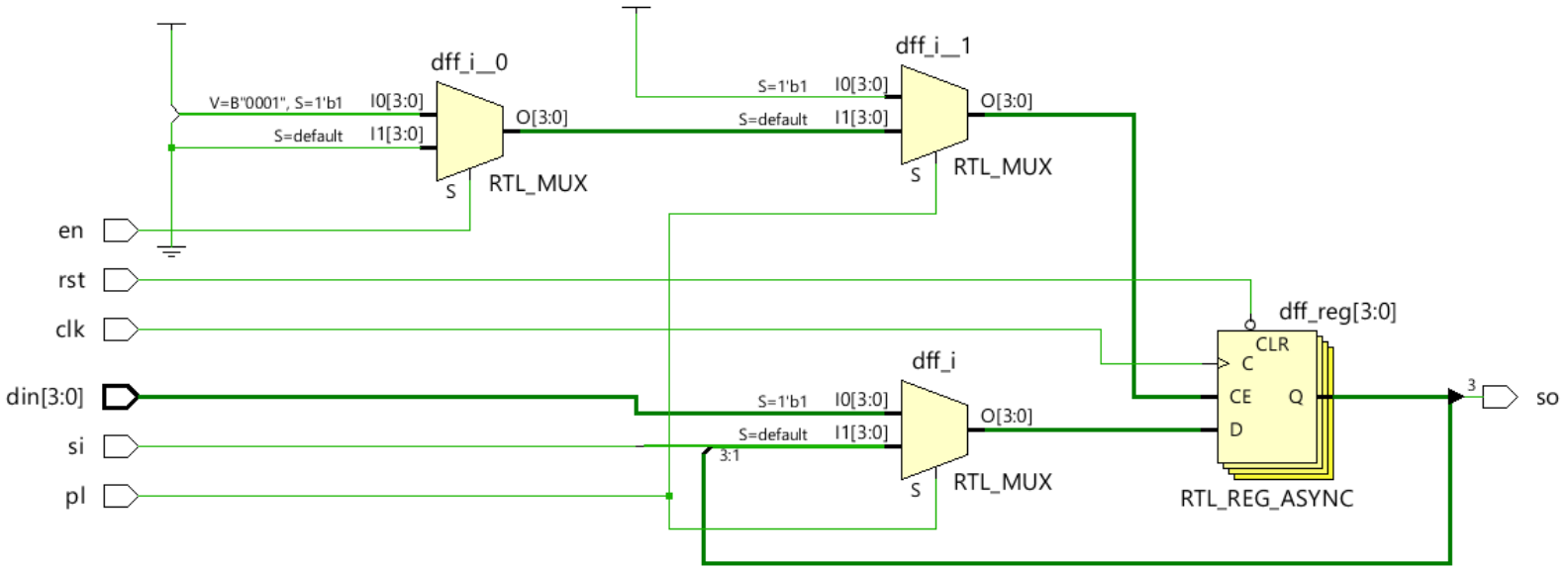
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity lshift_reg3 is
    Port (
        clk, rst, si, en, pl : in std_logic;
        din : in std_logic_vector(3 downto 0);
        so : out std_logic
        );
end entity;

architecture rtl of lshift_reg3 is
    signal dff : std_logic_vector(3 downto 0);

begin
    edge : process(clk, rst)
    begin
        if rst='0' then
            dff<= (others => '0');
        elsif clk'event and clk='1' then
            if pl='1' then
                dff <= din;
            elsif en='1' then
                dff <= dff(2 downto 1) & si; – Shifting the 3 LSBs of
the DFF and concatenating them with the input ‘si’ shifts the
contents of the register to the left by one position
            end if;
        end if;
    end process;
    so <= dff(3); – Ths MSB is now assigned as the serial output

end rtl;
```

Οι διαφορές σε σχέση με τον κώδικα rshift_sreg έχουν σχολιαστεί παραπάνω.

Το synthesis schema:



Προκύπτει το εξής RTL schema:



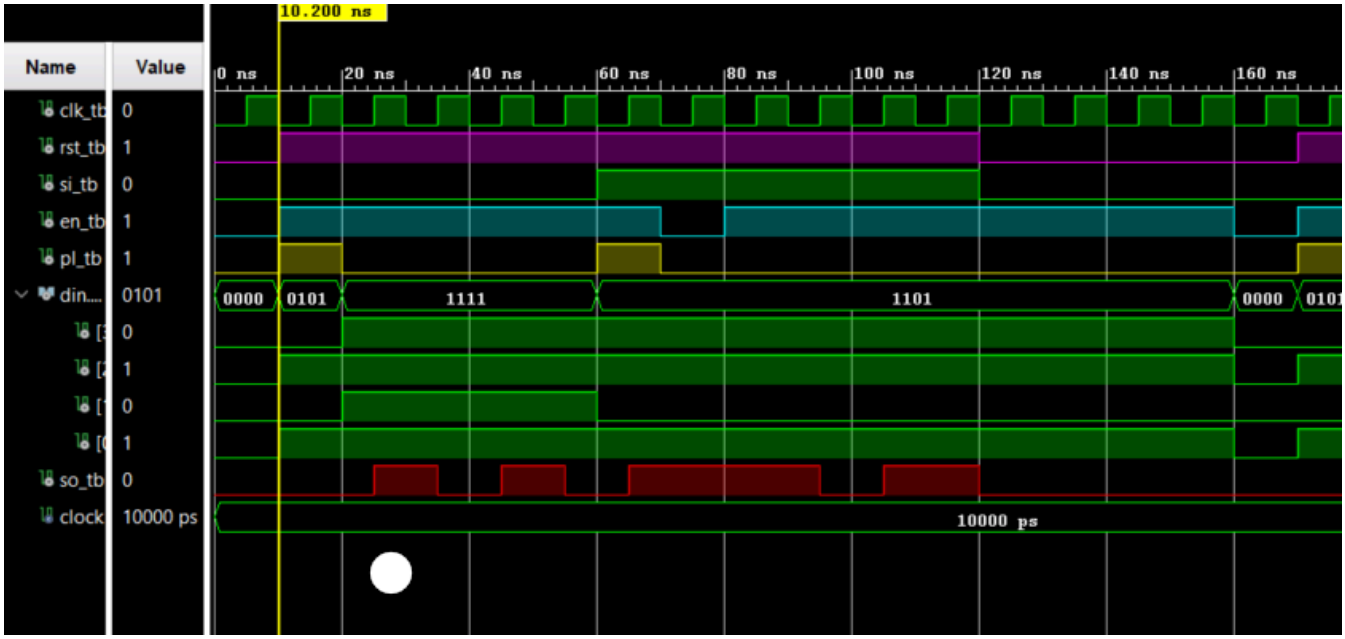
Όπως και προηγουμένως, οι δύο πολυπλέκτες διαχειρίζονται τα σήματα εισόδου, καθώς και τα parallel load και enabling bits, ενώ τα 4 DFFs διαχειρίζονται το left shifting και παράγουν τα σήματα εξόδου, με επιπλέον εισόδους αυτές του ρολογιού και του reset. Αν και το σχηματικό φαίνεται να εκτελεί τη ζητούμενη λειτουργία, θα εξετάσουμε την ακεραιότητα της οντότητας μέσω του testbench που φαίνεται παρακάτω.

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

entity lshift_reg3_tb is
--  Port ( );
end lshift_reg3_tb;

architecture bench of lshift_reg3_tb is
    component lshift_reg3 is
        Port (
            clk, rst, si, en, pl : in std_logic;
            din : in std_logic_vector(3 downto 0);
            so : out std_logic
            );
    end component;

    signal clk_tb, so_tb : std_logic;
    signal rst_tb, si_tb, en_tb, pl_tb : std_logic := '0';
    signal din_tb : std_logic_vector(3 downto 0) := (others => '0');

    constant clock : time := 10 ns;

begin
    dut : lshift_reg3
        port map (
            clk => clk_tb,
            rst => rst_tb,
            si => si_tb,
            en => en_tb,
            pl => pl_tb,
            din => din_tb,
            so => so_tb
        );

    simulation : process
    begin
        -- Initialization
        rst_tb <= '0';
```

```
pl_tb <= '0';
en_tb <= '0';
si_tb <= '0';
din_tb <= "0000";
wait for clock;

-- Test parallel load
rst_tb <= '1';
pl_tb <= '1';
en_tb <= '1';
si_tb <= '0';
din_tb <= "0101";
wait for clock;

-- Reset parallel load
-- The new input should not be printed in the output
rst_tb <= '1';
pl_tb <= '0';
en_tb <= '1';
si_tb <= '0';
din_tb <= "1111";
wait for 4*clock;

-- Test enable bit
rst_tb <= '1';
pl_tb <= '1';
en_tb <= '1';
si_tb <= '1';
din_tb <= "1101";
wait for clock;
pl_tb <= '0';
en_tb <= '0';
wait for clock;
en_tb <= '1';
wait for 4*clock;

-- Test reset
rst_tb <= '0';
pl_tb <= '0';
```

```
        en_tb <= '1';
        si_tb <= '0';
        wait for 4*clock;

    end process;

    generate_clock : process
        begin
            clk_tb <= '0';
            wait for clock/2;
            clk_tb <= '1';
            wait for clock/2;
    end process ;

end bench;
```

Τα αποτελέσματα της προσομοίωσης φαίνονται παρακάτω:



Όμοια με προηγουμένως, επιβεβαιώνεται η ορθή λειτουργία του κώδικα και για την αριστερή ολίσθηση.

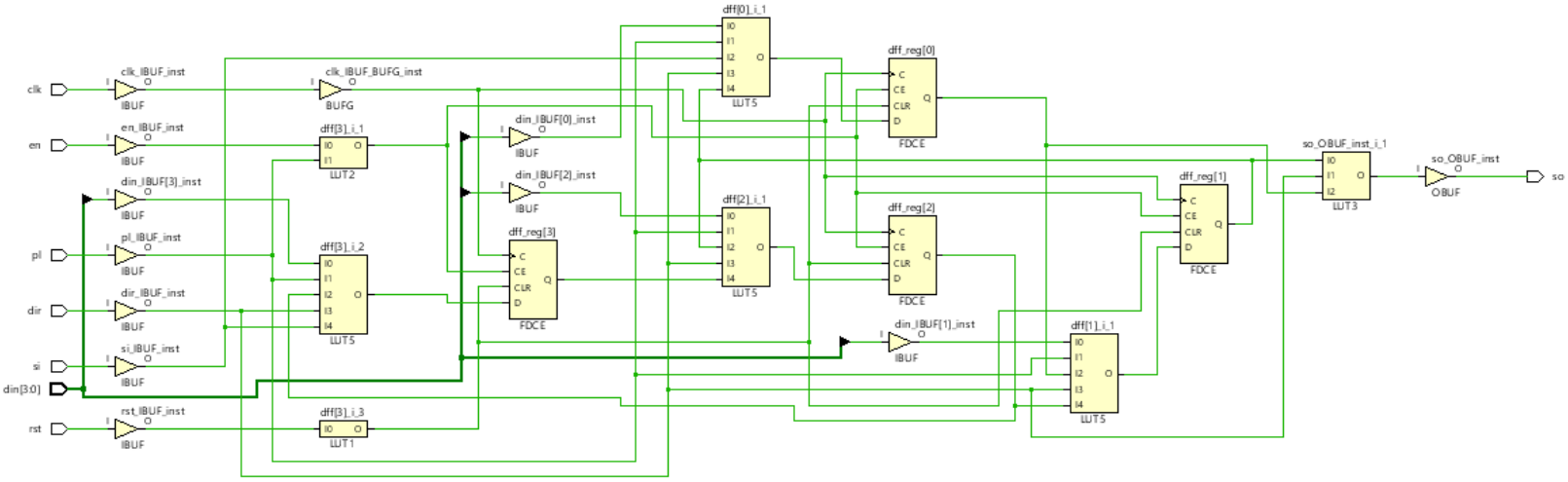
Shift_sreg:

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

```
entity shift_reg3 is
    Port (
        clk, rst, si, en, pl, dir : in std_logic;
        din : in std_logic_vector(3 downto 0);
        so : out std_logic
        );
end entity;

architecture rtl of shift_reg3 is
    signal dff : std_logic_vector(3 downto 0);

     begin
          edge : process(clk, rst)
          begin
               if rst='0' then
                    dff<= (others => '0');
               elsif clk'event and clk='1' then
                    if pl='1' then
                         dff <= din;
                    elsif en='1' then
                         if dir='0' then -- right shift register
                              dff <= si & dff(3 downto 1);
                         elsif dir='1' then -- left shift register
                              dff <= dff(2 downto 0) & si;
                         end if;
                    end if;
               end if;
          end process;

     with dir select so <= dff(0) when '0',
                           dff(3) when others;


end rtl;
```

Έχουμε ενοποιήσει τους δύο παραπάνω κώδικες, προσθέτοντας το control bit dir, το οποίο ελέγχει την κατεύθυνση ολίσθησης του καταχωρητή. Όταν `dir='0'`, τότε έχουμε δεξιά ολίσθηση, ενώ όταν `dir='1'`, έχουμε αριστερή ολίσθηση.

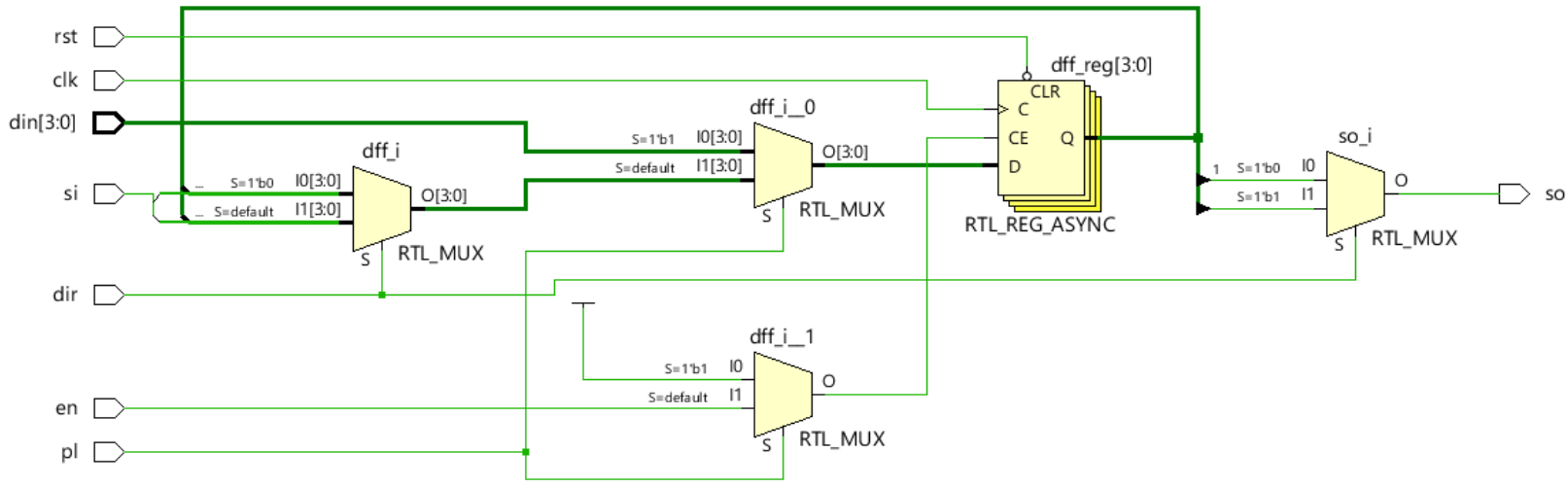
To synthesis schema:



Το κύκλωμα που προκύπτει από τον synthesizer, σε αυτή την περίπτωση, είναι το παραπάνω. Αξίζει να σημειώσουμε πως, καθώς έχουμε ένα επιπλέον control bit (`dir`), έχουμε και μια επιπλέον είσοδο στα LUTs.

Στην άσκηση αυτή καλούμαστε να απαντήσουμε αν θα μπορούσε το παραπάνω να σχεδιαστεί "με το χέρι" σε απλούστερη μορφή. Στην περίπτωση αυτή όμως, παρατηρούμε ότι δεν υπάρχουν πλεονασμοί, κομμάτια που επαναλαμβάνονται ή δεν χρησιμοποιούνται για παράδειγμα οπότε μπορούμε να πούμε ότι η παραπάνω υλοποίηση δεν μπορεί να απλουστευθεί παραπάνω.

To RTL schema:



Παρατηρούμε πως το σχήμα RTL διατηρεί τα χαρακτηριστικά των δύο προηγούμενων περιπτώσεων (οι δύο πολυπλέκτες διαχειρίζονται τα σήματα εισόδου,τα parallel load και enabling bits, ενώ τα 4 DFFs διαχειρίζονται το shifting και παράγουν τα σήματα εξόδου). Η ειδοποιός διαφορά έγκειται στην ύπαρξη ενός επιπλέον control bit, το direction bit. Όταν το bit είναι ίσο με μηδέν, τότε η φορά της ολίσθησης είναι προς τα δεξιά, ενώ όταν το control bit είναι triggered (ίσο με 1), η ολίσθηση έχει φορά προς τα αριστερά.

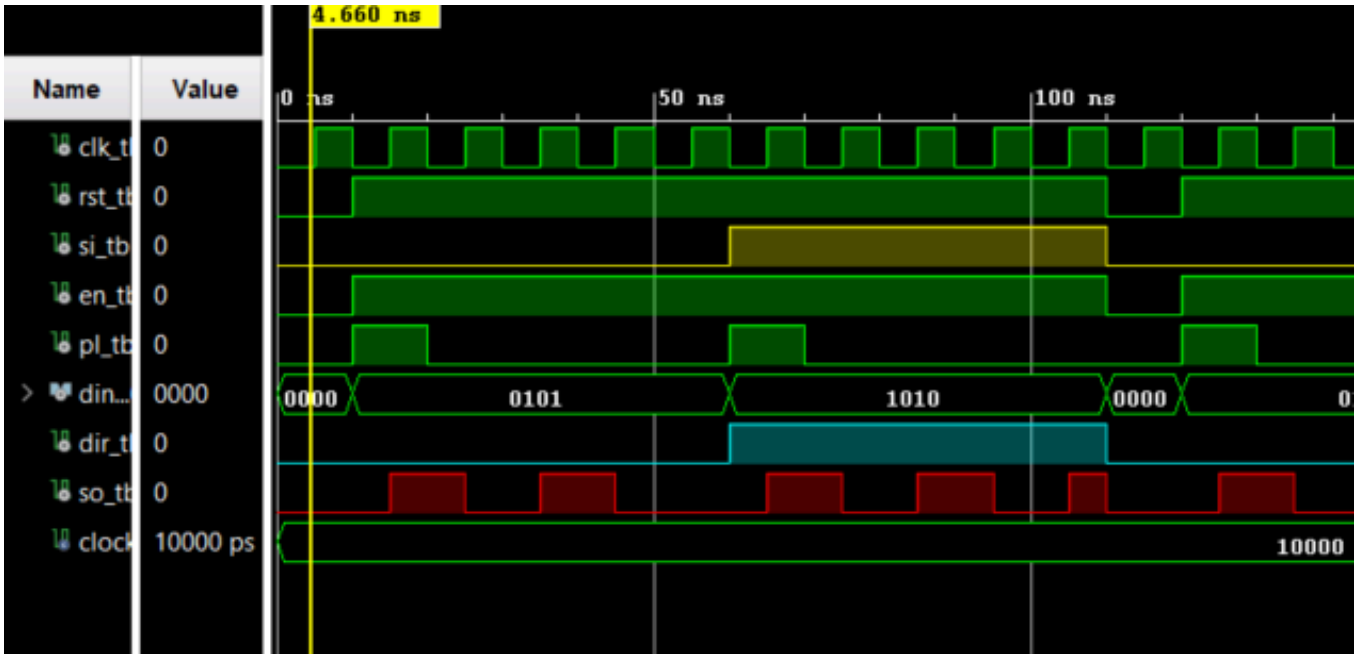
Ο κώδικας που χρησιμοποιήθηκε για το TestBench:

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

entity shift_reg3_tb is
--  Port ( );
end shift_reg3_tb;

architecture bench of shift_reg3_tb is
    component shift_reg3 is
        Port (
            clk, rst, si, en, pl, dir : in std_logic;
            din : in std_logic_vector(3 downto 0);
            so : out std_logic
            );
    end component;

    signal clk_tb, so_tb : std_logic;
    signal rst_tb, si_tb, en_tb, pl_tb, dir_tb : std_logic := '0';
    signal din_tb : std_logic_vector(3 downto 0) := (others => '0');

    constant clock : time := 10 ns;

begin
    dut : shift_reg3
        port map (
            clk => clk_tb,
            rst => rst_tb,
            si => si_tb,
            en => en_tb,
            pl => pl_tb,
```

```
        dir => dir_tb,
        din => din_tb,
        so => so_tb
    );

simulation : process
begin
    -- Initialization
    rst_tb <= '0';
    pl_tb <= '0';
    en_tb <= '0';
    si_tb <= '0';
    dir_tb <= '0';
    din_tb <= "0000";
    wait for clock;

    -- Test right shift
    rst_tb <= '1';
    pl_tb <= '1';
    en_tb <= '1';
    si_tb <= '0';
    dir_tb <= '0';
    din_tb <= "0101";
    wait for clock;
    pl_tb <= '0';
    wait for 4*clock;

    -- Test left shift
    rst_tb <= '1';
    pl_tb <= '1';
    en_tb <= '1';
    si_tb <= '1';
    dir_tb <= '1';
    din_tb <= "1010";
    wait for clock;
    pl_tb <= '0';
    wait for 4*clock;
```

```
    end process;

    generate_clock : process
    begin
        clk_tb <= '0';
        wait for clock/2;
        clk_tb <= '1';
        wait for clock/2;
    end process ;

end bench;
```

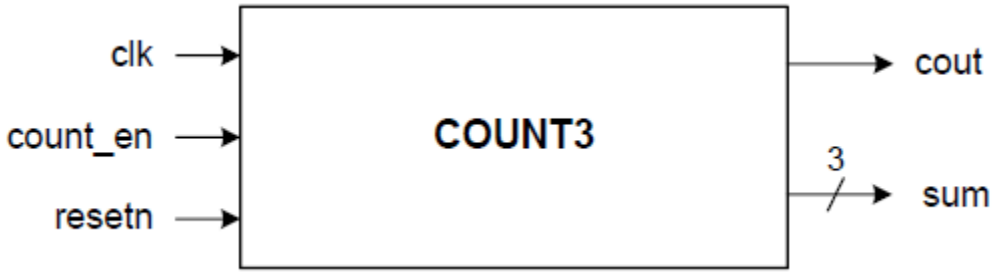
Από το Simulation προκύπτει:



Βλέπουμε πως πράγματι η λειτουργία του κώδικά μας είναι ορθή. Πιο συγκεκριμένα, κι έχοντας εξασφαλίσει τη σωστή λειτουργία των υπόλοιπων control bits παραπάνω, βλέπουμε πως όταν `dir='0'`, ο καταχωρητής πραγματοποιεί δεξιά ολίσθηση. Πιο συγκεκριμένα, για input `din='0101'` έχουμε σειριακή έξοδο `so = '1 0 1 0'`. Με την ολοκλήρωση του 6ου παλμού του ρολογιού, γίνεται trigger του direction bit, επομένως για `dir='1'`, ο καταχωρητής πραγματοποιεί αριστερή ολίσθηση. Πιο συγκεκριμένα, για input `din='1010'` έχουμε σειριακή έξοδο `so = '0 1 1 0 1 1'`. Ο επιπλέον άσσος οφείλεται στην αλλαγή του σειριακού input `si='1'`.

Θέμα Β.3: Μετρητής 3 bit με είσοδο ενεργοποίησης και κρατούμενο εξόδου

Στο ερώτημα αυτό μας δίνεται ένας μετρητής 3 bit, με είσοδο ενεργοποίησης και κρατούμενο εξόδου. Ο συγκεκριμένος μετρητής διαθέτει ασύγχρονη είσοδο μηδενισμού και σύγχρονη είσοδο ενεργοποίησης. Η είσοδος μηδενισμού είναι ενεργή σε λογικό ‘0’, ενώ η είσοδος ενεργοποίησης είναι ενεργή σε λογικό ‘1’.

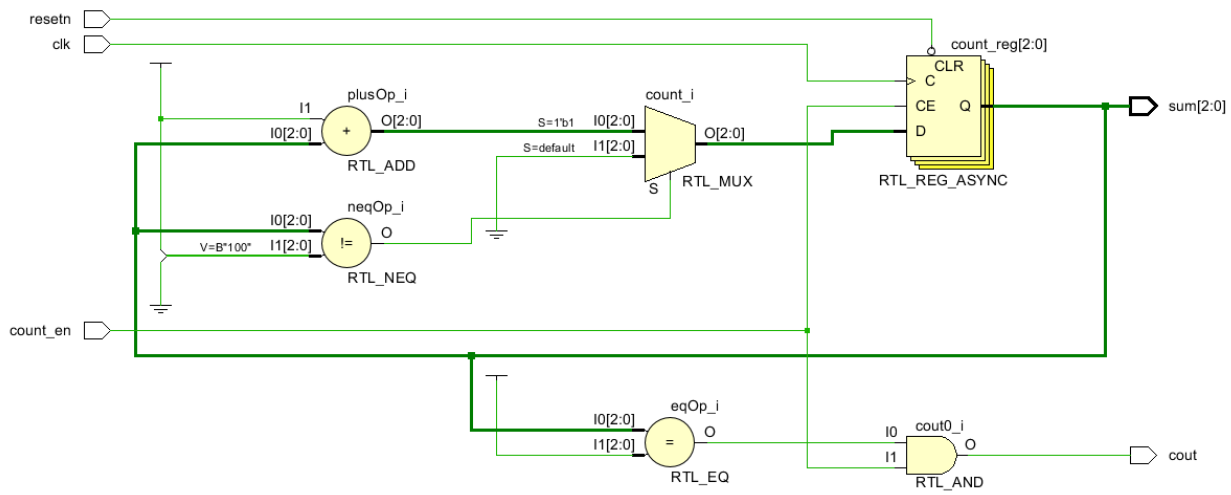


Ο κώδικας που μας δίνεται για τον μετρητή είναι ο παρακάτω:

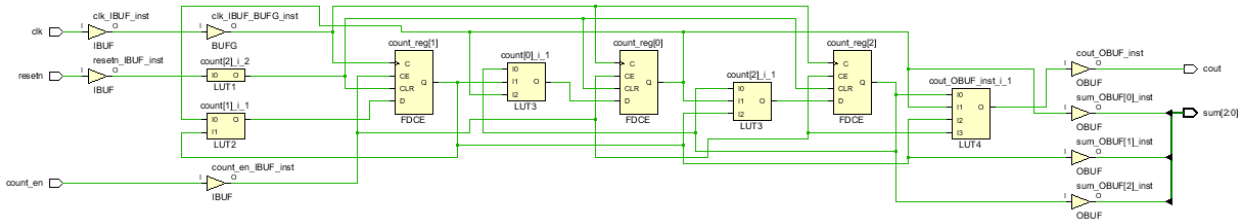
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity count3 is
    port(
        clk, resetn, count_en : in std_logic;
        sum : out std_logic_vector(2 downto 0);
        cout : out std_logic
    );
end;

architecture rtl_nolimit of count3 is
signal count: std_logic_vector(2 downto 0);
begin
    process(clk, resetn)
    begin
        if resetn='0' then
            count <= (others=>'0');
        elsif clk'event and clk='1' then
            if count_en='1' then
                count<=count+1;
            end if;
```

```
        end if;
    end process;

    sum <= count;
    cout <= '1' when count=7 and count_en='1' else '0';
end rtl_nolimit;

architecture rtl_limit of count3 is
signal count : std_logic_vector(2 downto 0);
begin
    process(clk, resetn)
    begin
        if resetn='0' then
        count <= (others=>'0');
        elsif clk'event and clk='1' then
            if count_en = '1' then
                if count/=7 then
                    count <= count+1;
                else
                    count<=(others=>'0');
                end if;
            end if;
        end if;
    end process;
    sum<= count;
    cout <= '1' when count=7 and count_en='1' else '0';
end;
```

Από την RTL ανάλυση προκύπτει το παρακάτω διάγραμμα:



Από τον Synthesizer προκύπτει το παρακάτω διάγραμμα:



Το διάγραμμα που προκύπτει είναι το ίδιο και για τις δύο αρχιτεκτονικές καθώς το όριο της δεύτερης αρχιτεκτονικής είναι το 7.

Στην άσκηση αυτή καλούμαστε να απαντήσουμε αν θα μπορούσε το παραπάνω να σχεδιαστεί “με το χέρι” σε απλούστερη μορφή. Στην περίπτωση αυτή όμως, παρατηρούμε ότι δεν υπάρχουν πλεονασμοί, κομμάτια που επαναλαμβάνονται ή δεν χρησιμοποιούνται για παράδειγμα οπότε μπορούμε να πούμε ότι η παραπάνω υλοποίηση δεν μπορεί να απλουστευθεί παραπάνω.

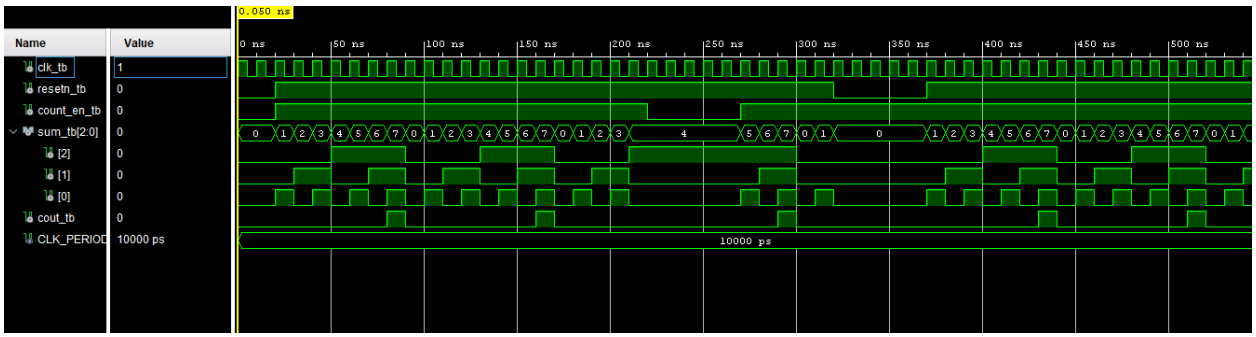
Χρησιμοποιούμε το παρακάτω TestBench για να επιβεβαιώσουμε την ορθή λειτουργία της υλοποίησης μας.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity count3_tb_rtl_nolimit is
end count3_tb_rtl_nolimit;
```

```
architecture rtl_nolimit of count3_tb_rtl_nolimit is
    component count3
        port(
            clk, resetn, count_en : in std_logic;
            sum : out std_logic_vector(2 downto 0);
            cout : out std_logic
        );
    end component;

    signal clk_tb : std_logic := '0';
    signal resetn_tb : std_logic := '1';
    signal count_en_tb : std_logic := '0';
    signal sum_tb : std_logic_vector(2 downto 0);
    signal cout_tb : std_logic;

    constant CLK_PERIOD : time := 10 ns;
begin
    UUT_nolimit: count3
    port map (
        clk => clk_tb,
        resetn => resetn_tb,
        count_en => count_en_tb,
        sum => sum_tb,
        cout => cout_tb
    );

    clk_process: process
    begin
        while now < 1000 ns loop
            clk_tb <= not clk_tb;
            wait for CLK_PERIOD / 2;
        end loop;
        wait;
    end process;

    stimulus_process: process
    begin
        resetn_tb <= '0';
```

```
        count_en_tb <= '0';

        wait for 20 ns;

        resetn_tb <= '1';
        count_en_tb <= '1';

        wait for 200 ns;

        count_en_tb <= '0';

        wait for 50 ns;

        count_en_tb <= '1';

        wait for 50 ns;

        resetn_tb <= '0';

        wait for 50 ns;

        resetn_tb <= '1';

        wait;
    end process;
end rtl_nolimit;
```

Η προσομοίωση προκύπτει ως εξής:



Βλέπουμε το enable παίρνει την λογική τιμή ‘1’ και το reset έχει την λογική τιμή ‘1’, ξεκινάει η μετρηση μέχρι το 7, όπου το carry out γίνεται ‘1’ και η μέτρηση ξεκινάει από την αρχή. Όταν το enable πάρει την λογική τιμή ‘0’, η μέτρηση σταματάει και ξεκινάει πάλι όταν το enable πάρει πάλι την λογική τιμή ‘1’. Αντίστοιχα, όταν το reset πάρει την λογική τιμή ‘0’, όπου γίνεται ενεργή η είσοδος μηδενισμού, και όσο έχει αυτή την τιμή, η μέτρηση ξεκινάει πάλι από το 0. Όταν ξαναπάρει την λογική τιμή ‘1’, ξεκινάει για άλλη μια φορά η μέτρηση.

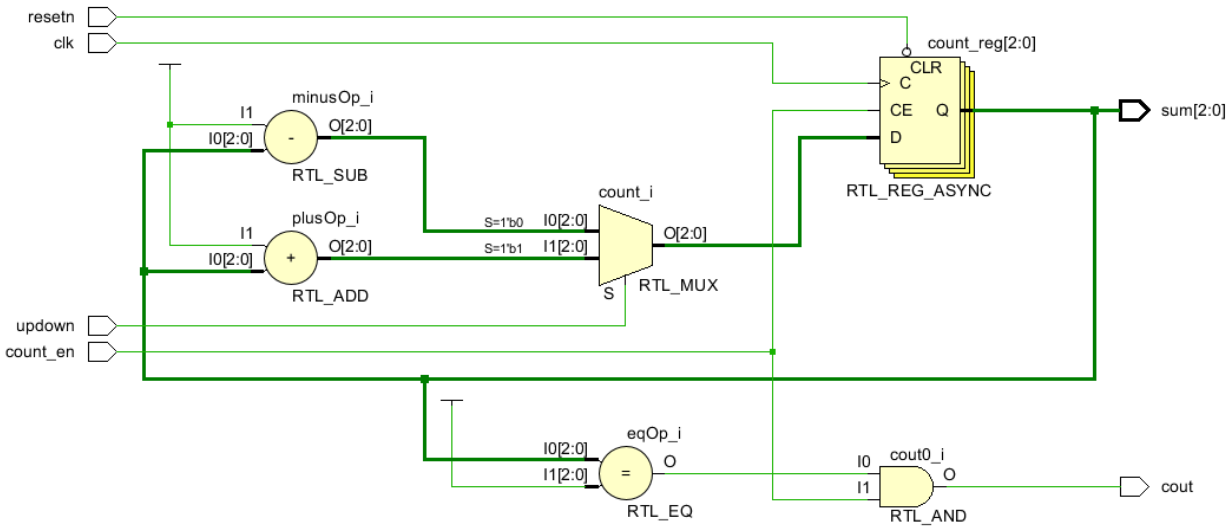
1. Βασιζόμενοι στην περιγραφή του μετρητή των 3 bits, μας ζητείται να περιγράψουμε έναν μετρητή up/down των 3 bits. Για την υλοποίηση θα χρησιμοποιήσουμε μία είσοδο επιλογής κατεύθυνσης, όπου όταν παίρνει την τιμή ‘1’ η μέτρηση θα γίνεται προς τα πάνω και όταν παίρνει την τιμή ‘0’ η μέτρηση θα γίνεται προς τα κάτω. Για αυτή την λειτουργία χρησιμοποιούμε την εντολή case.

Ο κώδικας VHDL φαίνεται παρακάτω:

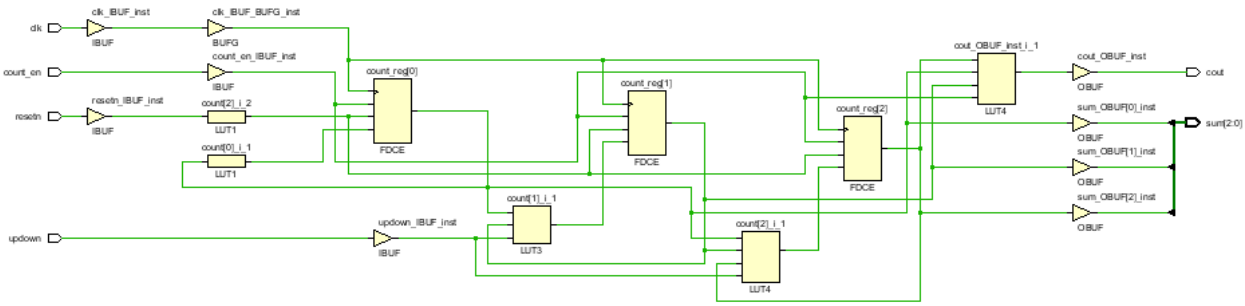
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity count3_updown is
    port(
        clk,
        resetn,
        updown : in std_logic;
        count_en : in std_logic;
        sum : out std_logic_vector(2 downto 0);
        cout : out std_logic
    );
end;

architecture rtl_nolimit of count3_updown is
signal count: std_logic_vector(2 downto 0);
begin
    process(clk, resetn)
    begin
        if resetn='0' then
            count <= (others=>'0');
        elsif clk'event and clk='1' then
            if count_en='1' then
                case updown is
```

```
                    when '0' => count<=count-1;
                    when '1' => count<=count+1;
                end case;
            end if;
        end if;
    end process;

    sum <= count;
    cout <= '1' when count=7 and count_en='1' else '0';

end;
```

Από την RTL ανάλυση προκύπτει το εξής schematic:



Και από τον Synthesizer:

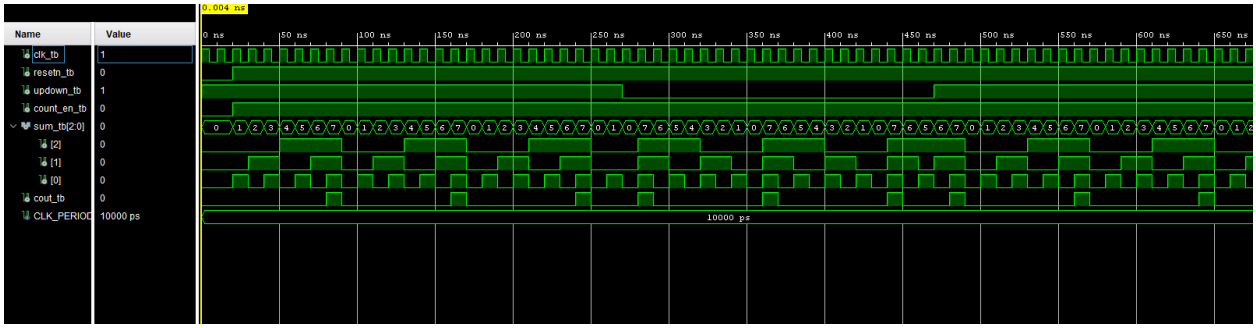


Για το TestBench χρησιμοποιήσαμε τον παρακάτω κώδικα:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity count3_updown_tb is
end count3_updown_tb;

architecture sim of count3_updown_tb is
    component count3_updown
        port(
            clk, resetn, updown, count_en : in std_logic;
            sum : out std_logic_vector(2 downto 0);
            cout : out std_logic
        );
    end component;

    signal clk_tb : std_logic := '0';
    signal resetn_tb : std_logic := '1';
    signal updown_tb : std_logic := '1';
    signal count_en_tb : std_logic := '0';
    signal sum_tb : std_logic_vector(2 downto 0);
    signal cout_tb : std_logic;

    constant clk_period : time := 10 ns;
begin
    dut: count3_updown
    port map (
        clk => clk_tb,
        resetn => resetn_tb,
        updown => updown_tb,
        count_en => count_en_tb,
        sum => sum_tb,
        cout => cout_tb
    );

    clk_process: process
    begin
        while now < 1000 ns loop
```

```
            clk_tb <= not clk_tb;
            wait for clk_period / 2;
        end loop;
        wait;
    end process;

    stimulus_process: process
    begin
        resetn_tb <= '0';
        count_en_tb <= '0';

        wait for 20 ns;

        resetn_tb <= '1';
        count_en_tb <= '1';

        wait for 200 ns;

        updown_tb <= '1';

        wait for 50 ns;

        updown_tb <= '0';

        wait for 200 ns;

        updown_tb <= '1';

        wait;
    end process;
end sim;
```

Το αποτέλεσμα της προσομοίωσης είναι το εξής:



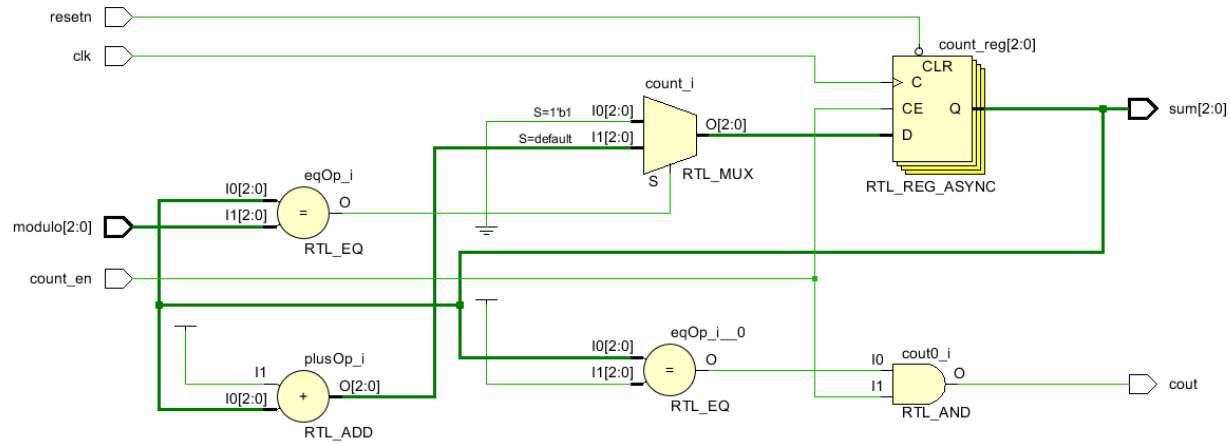
Παρατηρούμε ότι όντως όταν το updown γίνεται ‘0’, όντως η μέτρηση αντιστρέφεται και μετράμε προς τα κάτω.

2. Βασιζόμενοι στην περιγραφή του μετρητή 3 bits μας ζητείται να περιγράψουμε έναν up counter με παράλληλη είσοδο modulo (όριο μέτρησης) των 3 bits. Η λειτουργία modulo ορίζει τη μέγιστη τιμή μέτρησης που μπορεί να φτάσει ο μετρητής πριν την επαναφορά στο μηδέν. Δημιουργεί ουσιαστικά μια κυκλική συμπεριφορά, όπου ο μετρητής μετράει από το 0 μέχρι την τιμή του modulo και στη συνέχεια επανέρχεται στο 0. Σε αυτήν την περίπτωση, η τιμή modulo θα καθορίζεται από το σήμα εισόδου “modulo”, επιτρέποντας στον μετρητή να έχει ένα ρυθμιζόμενο όριο μέτρησης .

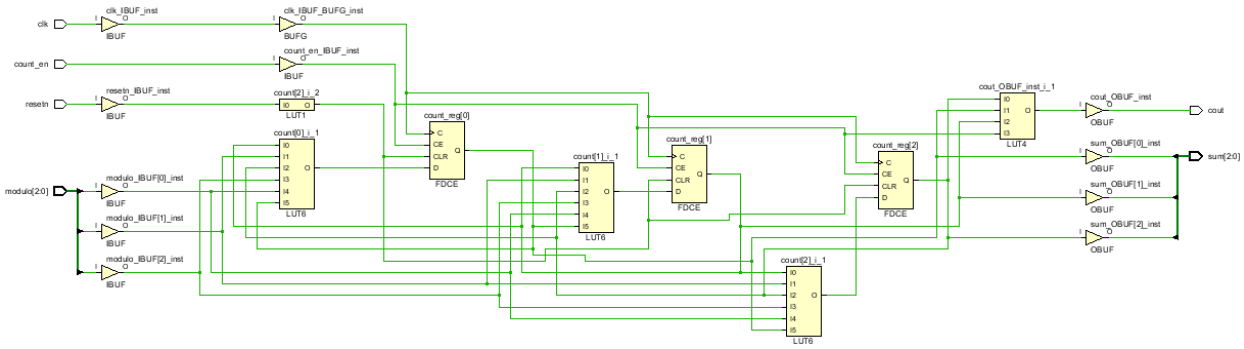
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity count3_mod is
    port(
        clk : in std_logic;
        resetn : in std_logic;
        modulo : in std_logic_vector(2 downto 0);
        count_en : in std_logic;
        sum : out std_logic_vector(2 downto 0);
        cout : out std_logic);
end;

architecture rtl_limit of count3_mod is
    signal count : std_logic_vector(2 downto 0);
begin
    process(clk, resetn)
    begin
```

```
        if resetn = '0' then
            count <= (others => '0');
        elsif clk'event and clk = '1' then
            if count_en = '1' then
                if count = modulo then
                    count <= (others => '0');
                else
                    count <= count + 1;
                end if;
            end if;
        end if;
    end process;

    sum <= count;
    cout <= '1' when count = "111" and count_en = '1' else '0';
end architecture rtl_limit;
```

Από την RTL ανάλυση προκύπτει το εξής:



Από το Synthesis:



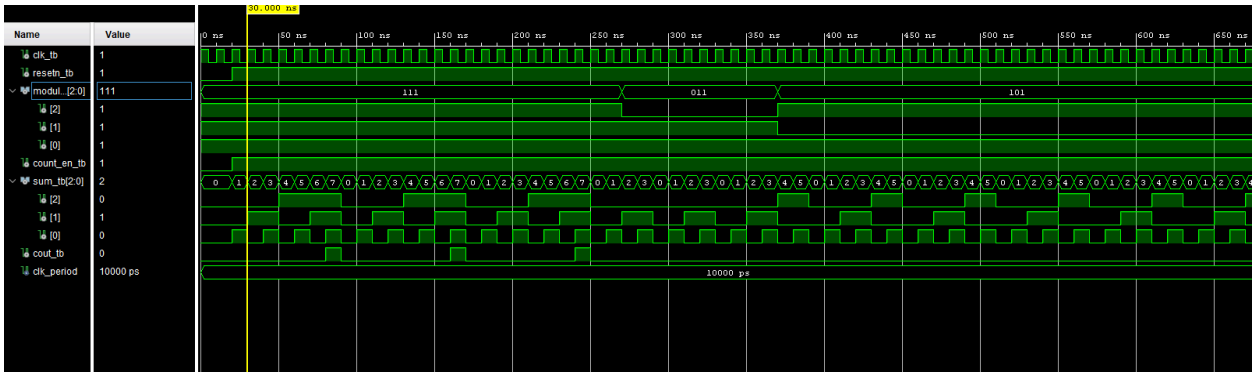
Ο κώδικας που χρησιμοποιήθηκε για το TestBench:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity count3_mod_tb is
end count3_mod_tb;

architecture sim of count3_mod_tb is
    constant clk_period : time := 10 ns;

    signal clk_tb : std_logic := '0';
    signal resetn_tb : std_logic := '1';
    signal modulo_tb : std_logic_vector(2 downto 0) := "111";
    signal count_en_tb : std_logic := '0';
    signal sum_tb : std_logic_vector(2 downto 0);
    signal cout_tb : std_logic;

    component count3_mod
        port(
            clk : in std_logic;
            resetn : in std_logic;
            modulo : in std_logic_vector(2 downto 0);
            count_en : in std_logic;
            sum : out std_logic_vector(2 downto 0);
            cout : out std_logic
        );
    end component;
```

```
begin
    dut: count3_mod
        port map (
            clk => clk_tb,
            resetn => resetn_tb,
            modulo => modulo_tb,
            count_en => count_en_tb,
            sum => sum_tb,
            cout => cout_tb
        );

    clk_process: process
    begin
        while now < 1000 ns loop
            clk_tb <= not clk_tb;
            wait for clk_period / 2;
        end loop;
        wait;
    end process;

    stimulus_process: process
    begin
        resetn_tb <= '0';
        count_en_tb <= '0';

        wait for 20 ns;

        resetn_tb <= '1';
        count_en_tb <= '1';

        wait for 200 ns;

        count_en_tb <= '1';
        wait for 50 ns;

        modulo_tb <= "011";
        wait for 50 ns;

        count_en_tb <= '1';
```

```
        wait for 50 ns;

        modulo_tb <= "101";
        wait for 50 ns;

        count_en_tb <= '1';
        wait for 50 ns;

        wait;
    end process;
end sim;
```

Και το Simulation βγαίνει ως εξής:



Πάλι μπορούμε να επιβεβαιώσουμε την ορθή λειτουργία του προγράμματος. Αρχικά η τιμή του "`modulo`" είναι 111 και βλέπουμε ότι ο μετρητής μας, μετράει κανονικά μέχρι το 7. Στην συνέχει το "`modulo`" αλλάζει σε 011 και 101 και ο μετρητής μας σε αυτές τις περιπτώσεις μετράει μέχρι το 3 και το 5 αντίστοιχα. Δηλαδή βλέπουμε ότι όντως το "`modulo`" λειτουργεί σαν άνω φράγμα στην μέτρησή μας.

Στο τέλος αφού έχουμε ελέγξει την ορθότητα λειτουργίας του κυκλώματος που δίνει ο Synthesizer και από τις προσομοιώσεις, μας ζητείται να απαντήσουμε για τι άν θα μπορούσε το κύκλωμα σε κάθε ζήτημα να σχεδιαστεί "με το χέρι" σε απλούστερη μορφή. Πάλι η απάντηση είναι ότι στις συγκεκριμένες υλοποιήσεις δεν υπάρχουν αλλαγές που θα μπορούσαμε να κάνουμε προκειμένου να απλοποιήσουμε παραπάνω το κύκλωμα. Το Vivado, ούτως ή άλλως, έχει αλγορίθμους που βρίσκουν τις πιο απλές λύσεις για την υλοποίηση των κυκλωμάτων και την