ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΙΣΤΩΝ

ΨΗΦΙΑΚΑ VLSI – 8ο ΕΞΑΜΗΝΟ

Μαρκέτος Νικόδημος – ΑΜ: 03117095

Τζομάκα Αφροδίτη – ΑΜ: 03117107

Ομάδα Β2

Θέμα Α2: Δυαδικός αποκωδικοποιητής 3 σε 8

Στο συγκεκριμένο θέμα, καλούμαστε να υλοποιήσουμε την περιγραφή της οντότητας του δυαδικού αποκωδικοποιητή 3-σε-8 τόσο σε behavioural όσο και σε dataflow αρχιτεκτονική.

Για την αναπαραγωγή του κώδικα (behavioural και dataflow) βασιστήκαμε στον αντίστοιχο πίνακα αληθείας, ενώ ο κώδικας φαίνεται παρακάτω:

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

Behavioural Architecture

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

entity dec3to8 is
  port (
        enc : in  std_logic_vector(2 dwonto 0);
        dec : out std_logic_vector(7 downto 0)
        ) ;
end entity;

architecture bh_arch of dec3to8 is

begin

    DEC_LOGIC : process(enc)
    begin
        case enc is
```

Dataflow Architecture

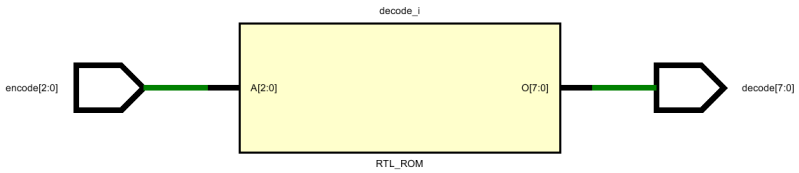
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity dec3to8 is
  port (
        enc : in  std_logic_vector(2 dwonto 0);
        dec : out std_logic_vector(7 downto 0)
        ) ;
end entity;

architecture df_arch of dec3to8 is
begin

    with enc select dec <=
        "00000001" when "000"
        "00000010" when "001"
        "00000100" when "010"
        "00001000" when "011"
```

```
            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 others => dec <= "10000000"
        end case ;
    end process ; -- DEC_LOGIC

end bh_arch ; -- bh_arch
```

```
                    "00010000" when "100"
                    "00100000" when "101"
                    "01000000" when "110"
                    "10000000" when others

end df_arch ; -- df_arch
```

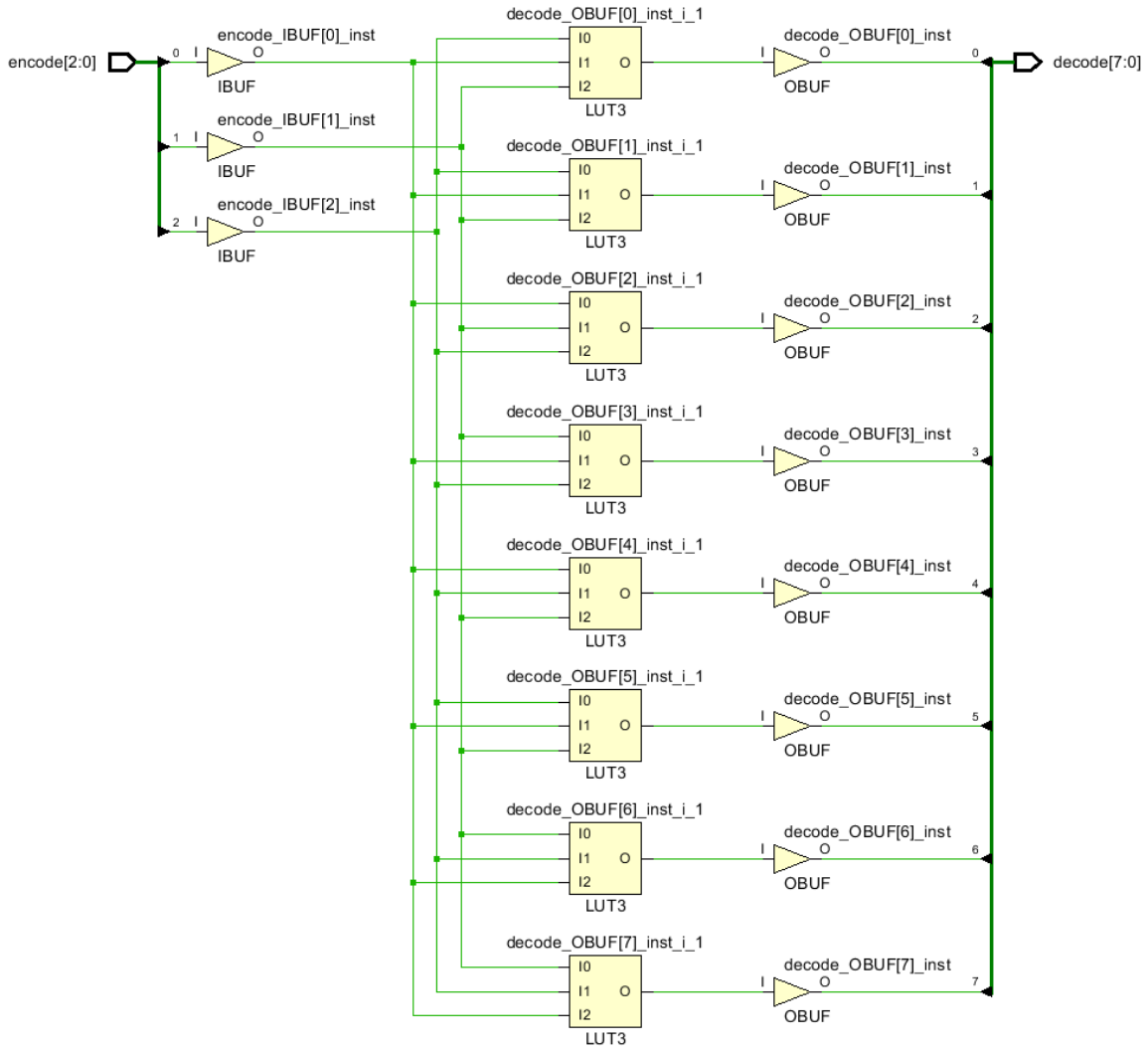
Όσον αφορά στον κώδικα, για την behavioral περιγραφή έγινε χρήση ενός process, το οποίο αντικατοπτρίζει την σειριακή λογική εκτέλεσης των εντολών στην εν λόγω αρχιτεκτονική. Στην dataflow περιγραφή από την άλλη έχουμε παράλληλη εκτέλεση των εντολών (η οποία αντιστοιχεί στην ροή των δεδομένων στο φυσικό κύκλωμα) με την βοήθεια της δομής “with-select”.

Το σχηματικό που προκύπτει σε Register-Transfer Level από τις παραπάνω περιγραφές με χρήση του Elaborated Design του Vivado είναι:



Παρατηρούμε, λοιπόν, ότι και στις δύο περιπτώσεις (behavioral – dataflow) επιλέχθηκε η χρήση μίας μνήμης ROM για την υλοποίηση του αποκωδικοποιητή.

Σε επίπεδο σύνθεσης, ωστόσο, το σχηματικό που παράγει το Vivado για την πλακέτα Zybo είναι το παρακάτω:



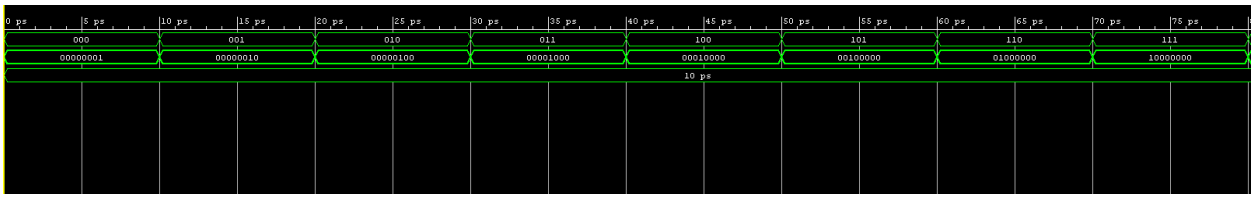
Για να επαληθεύσουμε την ορθή λειτουργία του κυκλώματος, το προσομοιώνουμε με το παρακάτω Testbench.

Dec3to8 Testbench

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

entity dec3to8_tb is
end dec3to8_tb;

architecture bench of dec3to8_tb is

    component dec3to8 is
        port (
            enc : in std_logic_vector(2 downto 0);
            dec : out std_logic_vector(7 downto 0)
             );
    end component;
```

```
    signal clock : std_logic;
    signal enc : std_logic_vector(2 downto 0) := (others => '0');
    signal dec : std_logic_vector(7 downto 0) := (others => '0');

    constant CLOCK_PERIOD : time := 10 ns;
begin
    uut : dec3to8
        port map (
                  enc   => enc,
                  dec   => dec
                 );

    stimulus : process
    begin
        enc <= (others => '0');
        wait for CLOCK_PERIOD;
        for i in 1 to 7 loop
            enc <= std_logic_vector(to_unsigned(i,3));
            wait for CLOCK_PERIOD;
        end loop ;
        enc <= (others => '0');
        wait;
    end process ; --   stimulus

end bench ; -- benc
```

Και όπως βλέπουμε από το αποτέλεσμα της προσομοίωσης ο κώδικας μας λειτουργεί όπως θα έπρεπε.



Θέμα Β2: Καταχωρητής ολίσθησης 4 bits με παράλληλη φόρτωση

Στο συγκεκριμένο θέμα, καλούμαστε να υλοποιήσουμε την περιγραφή της οντότητας ενός σύγχρονου καταχωρητή ολίσθησης των 4 bits, με ασύγχρονο reset και δυνατότητες παράλληλης φόρτωσης και ολίσθησης και προς τις δύο κατευθύνσεις.

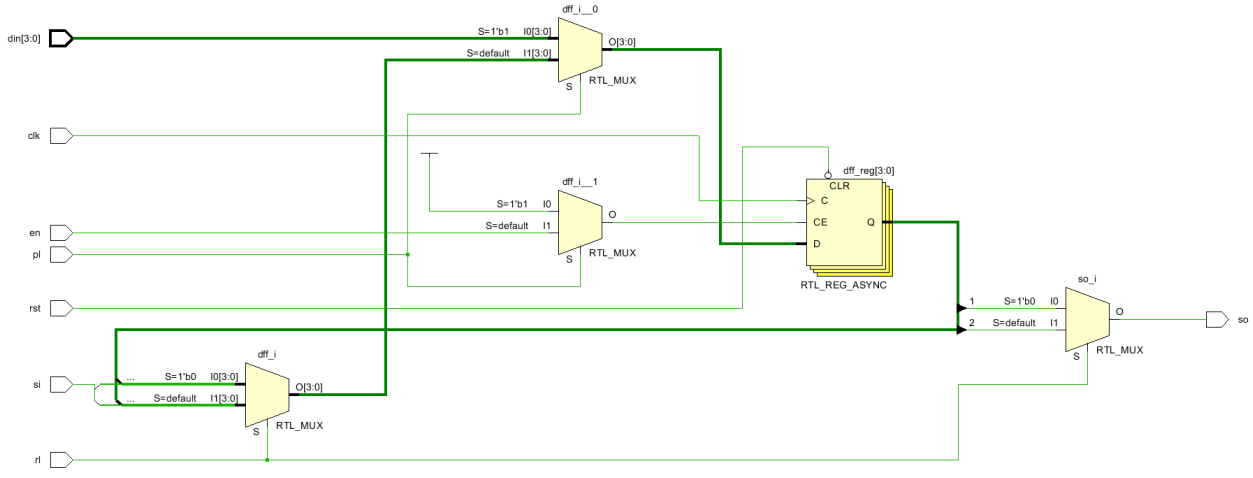
Behavioural Architecture of shift register

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

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

```
        );
end shift_reg;

architecture rtl of shift_reg 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
                    case rl is
                        when '0' =>-- rl=0 -> right // rl=1 -> left
                            dff <= si&dff(3 downto 1);
                        when others =>
                            dff <= dff(2 downto 0)&si;
                    end case ;
                end if;
            end if;
    end process;
    with rl select so <=
        dff(0) when '0',
        dff(3) when others;
end rtl;
```

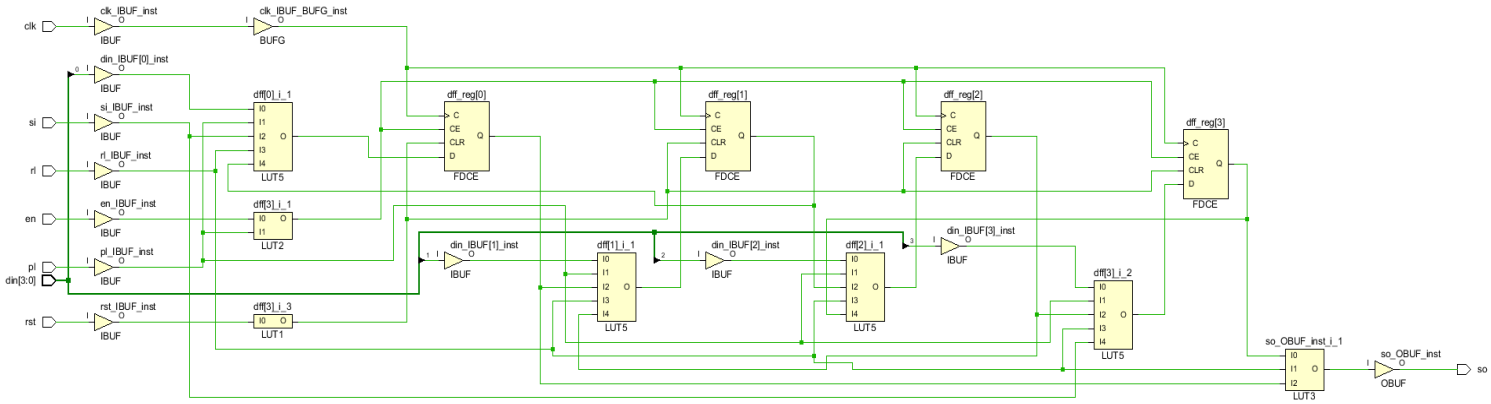
Το σήμα εισόδου rst είναι ασύγχρονο και ακολουθεί αρνητική λογική. Τα σήματα en και pl είναι σύγχρονα, με το σήμα pl να έχει μεγαλύτερη προτεραιότητα από το en. Για rl ίσο με λογικό 0 και λειτουργία ολίσθησης (rst = '1' && pl = '0' && en = '1') έχουμε ολίσθηση προς τα δεξιά, ενώ για rl ίσο με λογικό 1 και αντίστοιχη λειτουργία έχουμε ολίσθηση προς τα αριστερά. Το σήμα dff χρησιμοποιείται ως register 4 bit, στο οποίο κάθε φορά είναι φορτωμένη η 4-bit λέξη μας. Δεδομένου ότι, όπως εξηγήσαμε και στο προηγούμενο θέμα, το process θα τρέξει παράλληλα με την δομή with, είναι επιθυμητό η εκτέλεση οποιουδήποτε testbench να εκκινεί με ένα ασύγχρονο reset, ώστε να μην βρεθούμε σε κάποια απροσδιόριστη κατάσταση στο σήμα εξόδου so.

Ας δούμε τώρα το σχηματικό της περιγραφής σε Register-Transfer Level που προκύπτει από το Vivado.



Φαίνεται πως λειτουργεί όπως θα θέλαμε, μέσω των mux οι οποίοι διαχειρίζονται κατάλληλα τα σήματα εισόδου και εξόδου αλλά και των τεσσάρων καταχωρητών για το shifting, ωστόσο η πλήρης ορθότητα θα εξεταστεί μέσω του testbench που ακολουθεί αργότερα.

Σε επίπεδο σύνθεσης, το σχηματικό που παράγει το Vivado για την πλακέτα Zybo είναι το παρακάτω



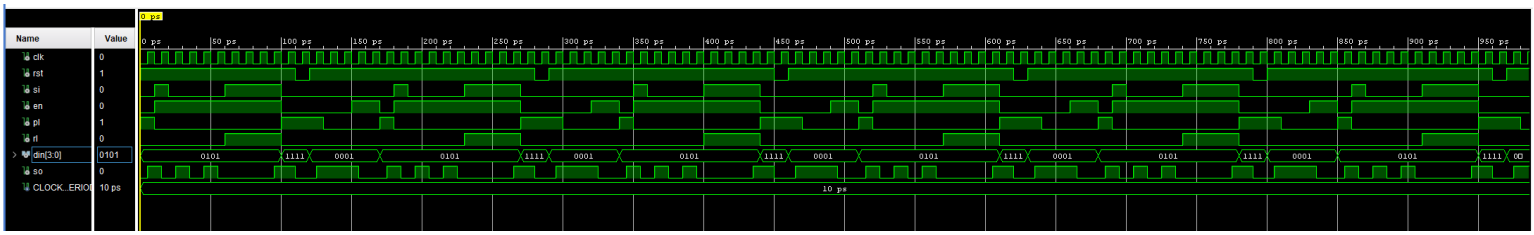
Εδώ βλέπουμε, πέρα από τα Lookup Tables και τους buffer σε είσοδο και έξοδο, την εισαγωγή μονάδων FDCE, οι οποίες υλοποιούν τα flip-flops του register, καθώς και ενός Global Clock Buffer, ο οποίος χρησιμοποιείται, ώστε να μην υπάρχει καθυστέρηση μεταξύ των σημάτων που λαμβάνουν από το ρολόι τα διάφορα FDCE, ώστε και οι 4 μονάδες να λειτουργούν ταυτόχρονα και συγχρονισμένα.

Για την επαλήθευση και πάλι της ορθής λειτουργείας τρέχουμε το παρακάτω Testbench το οποίο περιέχει σχόλια για τις λειτουργίες που ελέγχει.

Shifter Testbench

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

entity shift_reg_tb is
end shift_reg_tb;

architecture bench of shift_reg_tb is

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

    signal clk : std_logic;
    signal rst : std_logic := '0';
    signal si  : std_logic := '0';
    signal en  : std_logic := '0';
    signal pl  : std_logic := '0';
    signal rl  : std_logic := '0';
    signal din : std_logic_vector(3 downto 0) := (others => '0');

    signal so : std_logic;

    constant CLOCK_PERIOD : time := 10 ps;

begin
    utt: shift_reg
        port map (
            clk => clk,
```

```
            rst => rst,
            si  => si,
            en  => en,
            pl  => pl,
            rl  => rl,
            din => din,
            so  => so
        );

        stimulus : process
        begin
            -- Test Parallel Loading
            rst <= '1';
            en <= '0';
            si <= '0';
            pl <= '1';
            rl <= '0';
            din <= "0101";
            wait for CLOCK_PERIOD;      -- sreg => 0101 & so =>1

            -- Test shift with rl '0' == right
            pl <= '0';
            en <= '1';
            rl <= '0';
            si <= '1';
            wait for CLOCK_PERIOD;      -- sreg => 1010 & so => 0
            si <= '0';
            wait for 4*CLOCK_PERIOD;    -- sreg => 0101 -> 0010 -> 0001 -> 0000
                                        -- so => 1 -> 0 -> 1 -> 0

            -- Test shift with rl '1' == left
            pl <= '0';
            en <= '1';
            rl <= '1';
            si <= '1';
            wait for 4*CLOCK_PERIOD;    -- sreg => 0001 -> 0011 -> 0111 -> 1111
                                        -- so => 0 -> 0 -> 0 -> 1
            -- Test reset
            si <= '0';
            rl <= '0';
            rst <= '0';
            wait for CLOCK_PERIOD;      -- shifting occurs but so is 0 therefore reset is working
                                        --(sreg was 1111 so if reset didn't work so should be 1)

            -- Test enable
            rst <= '1';
            en <= '0';
            si <= '0';
            pl <= '1';
            rl <= '0';
            din <= "0001";
            wait for CLOCK_PERIOD;      -- sreg => 0001 & so => 1
            pl <= '0';
            wait for 2*CLOCK_PERIOD;    -- we must see that so is still 1
                                        -- so shifting does not occur without enable
            en <= '1';
            wait for 2*CLOCK_PERIOD;    -- now so => 0 due to right shifting
        end process ; -- stimulus

        generate_clock : process
        begin
            clk <= '0';
            wait for CLOCK_PERIOD/2;
            clk <= '1';
            wait for CLOCK_PERIOD/2;
        end process ; -- generate_clock
end bench ; -- bench
```



Όπως και αναφέραμε και στην εξέταση με την παραπάνω προσομοίωση αποδεικνύουμε την ορθή λειτουργεία του κυκλώματος.

Θέμα Β3: Μετρητές 3 bit

Στο συγκεκριμένο θέμα, καλούμαστε να υλοποιήσουμε την περιγραφή της οντότητας

i) ενός μετρητή up/down των 3 bits

ii) ενός μετρητή up των 3 bits με παράλληλη είσοδο modulo, 3 bits

i)

3-bit Up/Down Counter

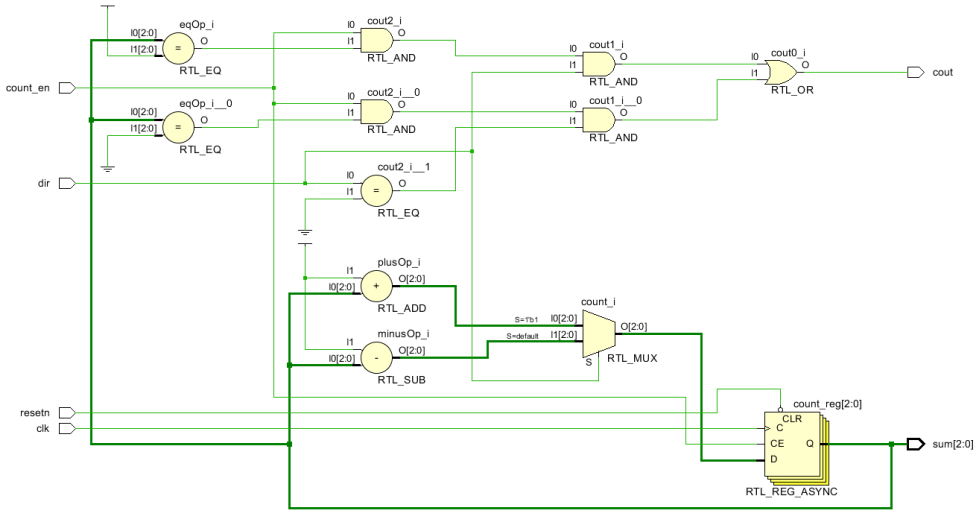
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity counter is
port(   clk, resetn, dir , count_en : in std_logic;
        sum : out std_logic_vector(2 downto 0);
        cout : out std_logic
    );
end;

architecture beh_counter of counter is
    signal count : std_logic_vector(2 downto 0);
    begin
    process(clk, resetn)
    begin
        if resetn='0' then
            count <= (others=>'0');
        elsif (rising_edge(clk) and count_en = '1' ) then
            if (dir ='1') then
                count <= count+1;
            else
                count <= count-1;
            end if;
        end if;
    end process;
    sum <= count;
    cout <= '1' when ((count_en = '1' and count="111" and dir='1')
         or (count_en = '1' and count="000" and dir='0')) else '0';
end;
```

TestBench

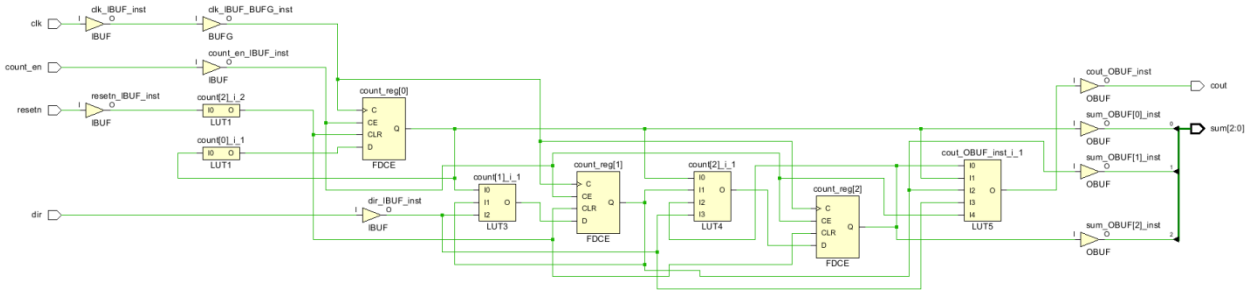
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity counter_tb is
end counter_tb;

architecture bench of counter_tb is
component counter
port(
        clk, resetn, dir , count_en : in std_logic;
        sum : out std_logic_vector(2 downto 0);
        cout : out std_logic);
end component;
  --Input Signals
    signal clk : std_logic := '0';
    signal dir : std_logic := '1';
    signal count_en: std_logic := '0';
    signal resetn: std_logic := '0';

    --Output Signals
    signal cout: std_logic;
    signal sum: std_logic_vector(2 downto 0);

    --Clock
    constant CLK_PERIOD : time := 10ps;
begin
UUT: counter port map(
        clk => clk, dir => dir,
        count_en => count_en, resetn => resetn,
        cout => cout, sum => sum
        );

    clk_proc: process
    begin
        clk <= '0';
        wait for CLK_PERIOD / 2;
        clk <= '1';
        wait for CLK_PERIOD / 2;
    end process;

     stimulus: process
        begin
        -- Test count up with upper limit
        count_en <= '1';
        resetn <= '1';
        dir <= '1';
        wait for 12*CLK_PERIOD;     -- output must be 001
        dir <= '0';
        wait for 8*CLK_PERIOD;     -- output must be 001
        count_en <= '0';
        wait for 3*CLK_PERIOD;     -- output must be 2
        resetn <= '0';             -- output must be zero
        wait for 3*CLK_PERIOD;
    end process;
end bench;
```

Για την ανάπτυξη του κώδικα χρησιμοποιήθηκε ο ήδη δοσμένος από την εκφώνηση με την προσθήκη ενός σήματος κατεύθυνσης το οποίο μέσω ενός if ελέγχει το αν θα έχουμε μέτρηση προς τα κάτω ή προς τα πάνω.

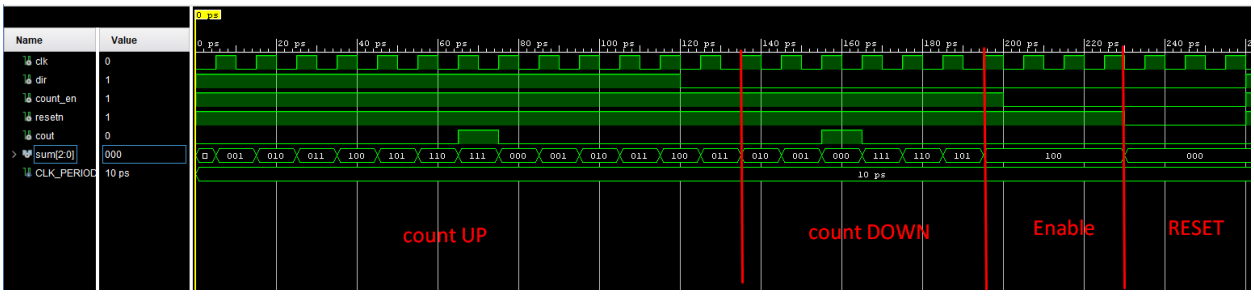
Το σχηματικό που προκύπτει σε επίπεδο RTL φαίνεται παρακάτω:



Ξανά, εποπτικά μπορούμε να πούμε ότι το κύκλωμα θα έχει την επιθυμητή απόδοση (πράγμα που θα επαληθεύσει το testbench παρακάτω.
Το σχηματικό της σύνθεσης:



Πράγματι, τελικά, μέσω της προσομοίωσης βλέπουμε ότι όλα έχουν γίνει σωστά.



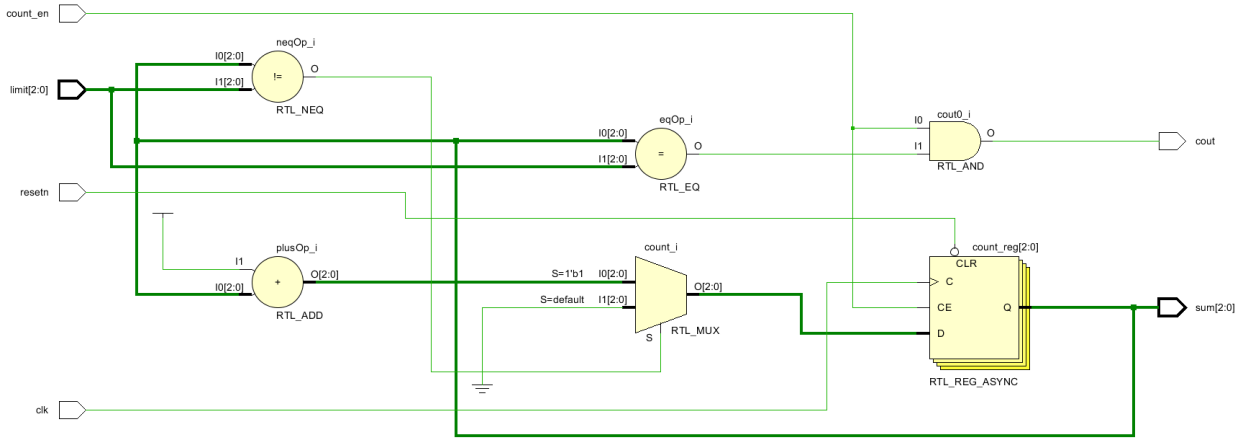
ii) Για τον κώδικα αυτού του ερωτήματος, βασιστήκαμε πάλι σε εκείνον της εκφώνησης προσθέτοντας το κομμάτι εκείνο που ελέγχει αν ο μετρητής έφτασε στο δοθέν από την είσοδο όριο (limit) και στην περίπτωση αυτήν να τον μηδενίζει. Ο κώδικας, τα σχηματικά από την RTL ανάλυση και την σύνθεση καθώς και η τελική προσομοίωση του Testbench φαίνονται παρακάτω.

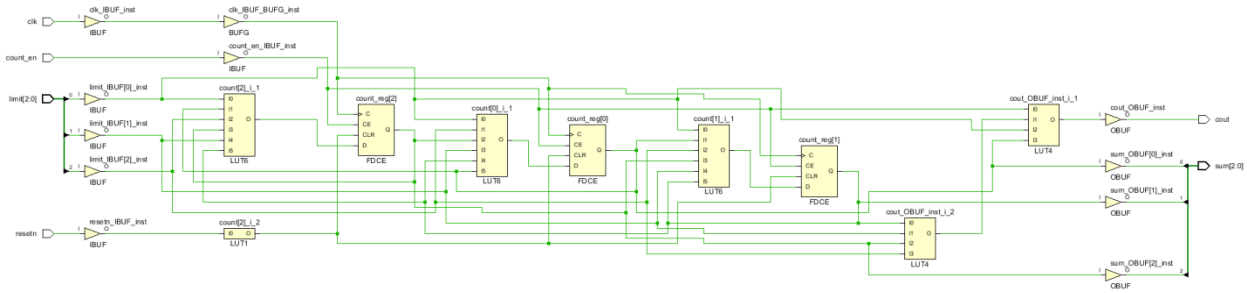
3-bit Up Limit Counter

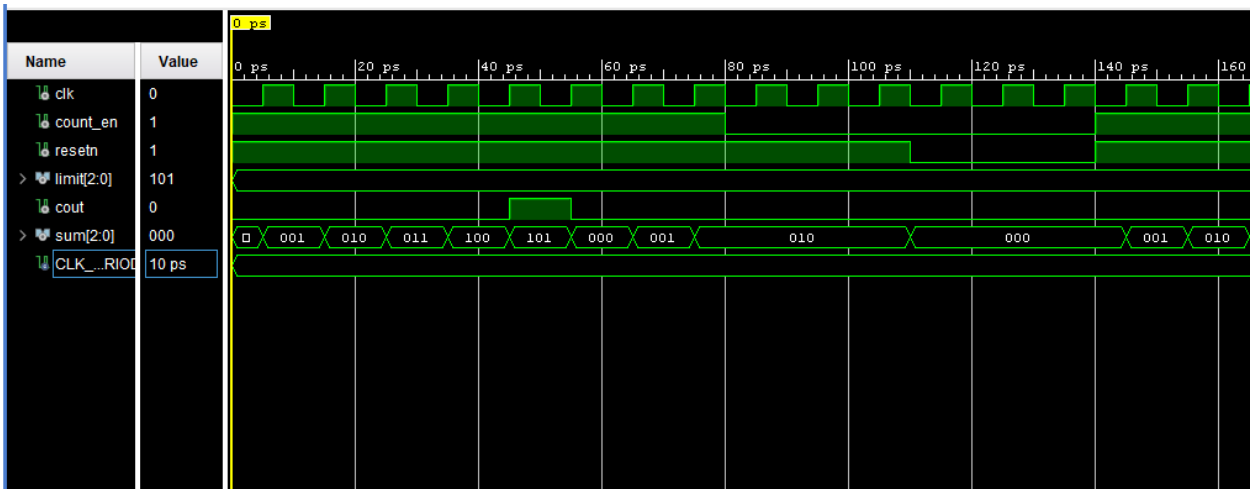
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity counter_lmt is
port(   clk, resetn, count_en : in std_logic;
        limit : in std_logic_vector(2 downto 0);
        sum : out std_logic_vector(2 downto 0);
        cout : out std_logic
    );
end;

architecture beh_counter of counter_lmt is
    signal count : std_logic_vector(2 downto 0);
    begin
    process(clk, resetn)
    begin
        if resetn='0' then
            count <= (others=>'0');
        elsif (rising_edge(clk) and count_en = '1' ) then
            if (count /= limit) then
                count <= count+1;
            else
                count <= "000";
            end if;
        end if;
    end process;
    sum <= count;
    cout <= '1' when (count_en = '1' and count = limit)
             else '0';
    end;
```

Testbench

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

entity counter_lmt_tb is
end counter_lmt_tb;

architecture bench of counter_lmt_tb is
component counter_lmt
port(
        clk, resetn, count_en : in std_logic;
        limit : in std_logic_vector(2 downto 0);
        sum : out std_logic_vector(2 downto 0);
        cout : out std_logic);
end component;
  --Input Signals
    signal clk : std_logic := '0';
    signal count_en: std_logic := '0';
    signal resetn: std_logic := '0';
    signal limit: std_logic_vector(2 downto 0) := (others => '0');

    --Output Signals
    signal cout: std_logic;
    signal sum: std_logic_vector(2 downto 0);

    --Clock
    constant CLK_PERIOD : time := 10ps;
begin
UUT: counter_lmt port map(
        clk => clk,
        count_en => count_en,
        resetn => resetn,
        limit => limit,
        cout => cout,
        sum => sum
        );

    clk_proc: process
    begin
        clk <= '0';
        wait for CLK_PERIOD / 2;
        clk <= '1';
        wait for CLK_PERIOD / 2;
    end process;

     stimulus: process
        begin
        -- Test count up with upper limit
        count_en <= '1';
        resetn <= '1';
        limit <= "101";
        wait for 8*CLK_PERIOD;    -- output must be 001

        count_en <= '0';
        wait for 3*CLK_PERIOD;    -- output must be 2

        resetn <= '0';            -- output must be zero
        wait for 3*CLK_PERIOD;

    end process;
end bench;
```


count_en
limit[2:0]
resetn
clk
neqOp_i
I0[2:0]
I1[2:0]
!=
O
RTL_NEQ
eqOp_i
I0[2:0]
I1[2:0]
=
O
RTL_EQ
cout0_i
I0
I1
O
RTL_AND
cout
plusOp_i
I1
I0[2:0]
+
O[2:0]
RTL_ADD
count_i
S=1'b1
I0[2:0]
S=default
I1[2:0]
O[2:0]
S
RTL_MUX
count_reg[2:0]
CLR
C
CE
D
Q
RTL_REG_ASYNC
sum[2:0]




Name
Value
clk
0
count_en
1
resetn
1
limit[2:0]
101
cout
0
sum[2:0]
000
CLK_...RIOD
10 ps
0 ps
20 ps
40 ps
60 ps
80 ps
100 ps
120 ps
140 ps
160
001
010
011
100
101
000
001
010
000
001
010