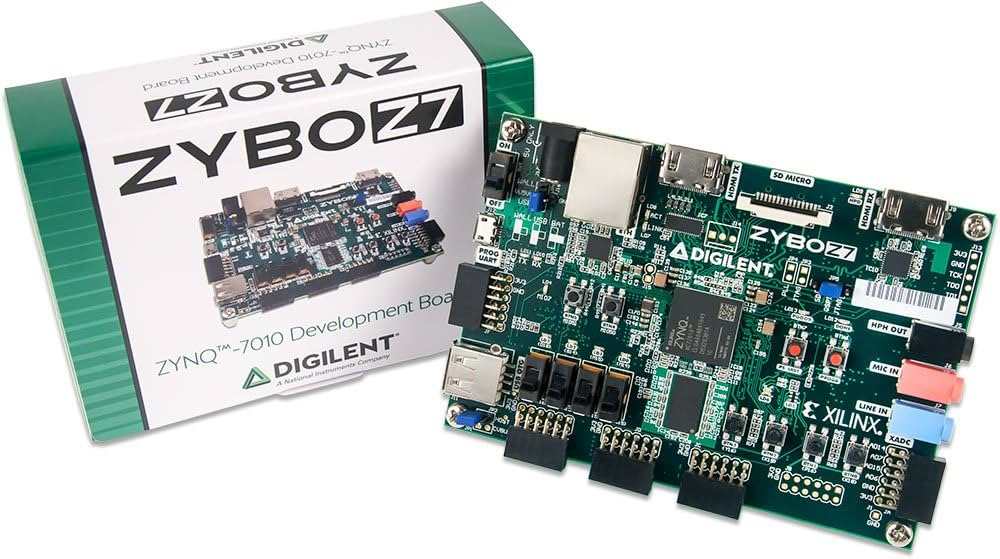
**Ψηφιακά Συστήματα VLSI**

**1η Εργαστηριακή Άσκηση**



* **Ομάδα: 6**
* **Μέλη: Ακύλας Αντώνιος 03121152 & Ιωάννης-Χρυσοβαλάντης Κουμπιάς 03121053**

**Άσκηση 1 (Α.2)**

Σκόπος της πρώτης άσκησης είναι η σχεδίαση ενός αποκωδικοποιητή 3 σε 8.

Αρχικά σχεδιάζουμε τον αποκωδικοποιητή σε Behavioral level κάνοντας χρήσης της δομής case, εξετάζοντας κάθε διαφορετική είσοδο.

-------------------------------------------------------------------

-- DEC 3 TO 8 (BEHAVIORAL)

-------------------------------------------------------------------

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.numeric\_std.all;

entity dec\_3\_to\_8\_behavioral is

port (

i\_enc : in std\_logic\_vector(2 downto 0);

o\_dec : out std\_logic\_vector(7 downto 0)

);

end entity;

architecture behavioral\_arch of dec\_3\_to\_8\_behavioral is

begin

DEC\_MODULE : process(i\_enc)

begin

case i\_enc is

when "000" =>

o\_dec <= X"01";

when "001" =>

o\_dec <= X"02";

when "010" =>

o\_dec <= X"04";

when "011" =>

o\_dec <= X"08";

when "100" =>

o\_dec <= X"10";

when "101" =>

o\_dec <= X"20";

when "110" =>

o\_dec <= X"40";

when "111" =>

o\_dec <= X"80";

when others =>

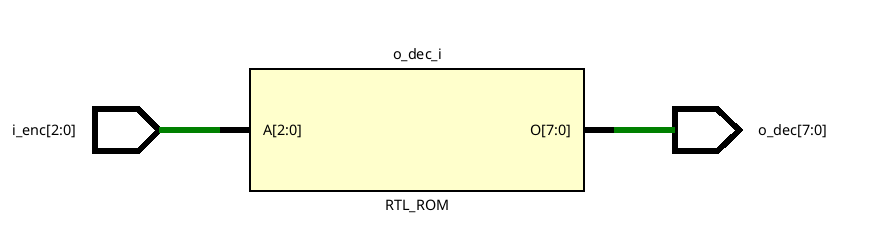
o\_dec <= (others => '-');

end case;

end process;

end behavioral\_arch;

Όπως βλέπουμε με αυτόν τον τρόπο σχεδίασης η “λογική” του κυκλώματος παρουσιάζεται σαν black box.



Έπειτα επαναλαμβάνουμε την ίδια διαδικασία σε Dataflow level. Εδώ έχουμε δύο επιλογές. Η πρώτη είναι να κάνουμε χρήση της εντολής select η οποία έχει μεγάλη ομοιότητα με την δομή case. Η δεύτερη επιλογή είναι να δουλέψουμε σε επίπεδο λογικών συναρτήσεων και να υλοποιήσουμε κάθε έξοδο του αποκωδικοποιητή “χειροκίνητα”. Παρακάτω βλέπουμε και τις δύο μεθόδους (η μία σε μορφή σχολίων):

--3 to 8 decoder | dataflow design

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.NUMERIC\_STD.ALL;

entity dec\_3\_to\_8\_dataflow is

port(

i\_enc : in std\_logic\_vector (2 downto 0);

o\_dec : out std\_logic\_vector (7 downto 0)

);

end entity;

architecture dataflow\_arch of dec\_3\_to\_8\_dataflow is

begin

o\_dec(0) <= (not i\_enc(2)) and (not i\_enc(1)) and (not i\_enc(0)) ;

o\_dec(1) <= (not i\_enc(2)) and (not i\_enc(1)) and i\_enc(0) ;

o\_dec(2) <= (not i\_enc(2)) and i\_enc(1) and (not i\_enc(0)) ;

o\_dec(3) <= (not i\_enc(2)) and i\_enc(1) and i\_enc(0) ;

o\_dec(4) <= i\_enc(2) and (not i\_enc(1)) and (not i\_enc(0)) ;

o\_dec(5) <= i\_enc(2) and (not i\_enc(1)) and i\_enc(0) ;

o\_dec(6) <= i\_enc(2) and i\_enc(1) and (not i\_enc(0)) ;

o\_dec(7) <= i\_enc(2) and i\_enc(1) and i\_enc(0) ;

-- with i\_enc select o\_dec <=

-- X"01" when "000",

-- X"02" when "001",

-- X"04" when "010",

-- X"08" when "011",

-- X"10" when "100",

-- X"20" when "101",

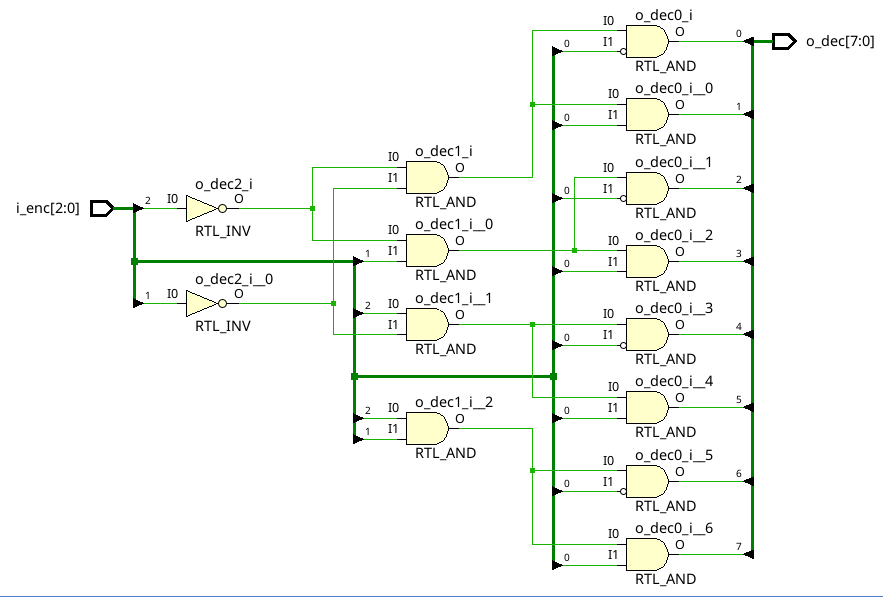
-- X"40" when "110",

-- X"80" when "111",

-- (others => '-') when others;

end architecture;

Όπως βλέπουμε παρακάτω στην υλοποίηση σε επίπεδο λογικών συναρτήσεων μπορούμε να δούμε αναλυτικά το κύκλωμα που παράγεται. Ωστόσο στην υλοποίηση με την εντολή select το διάγραμμα που παίρνουμε είναι και πάλι ένα “μαύρο κουτί”.



Στη συνέχεια καλούμαστε να επιβεβαιώσουμε την ορθή λειτουγρία των προγραμμάτων μέσω προσομοίωσης. Για τον λόγο αυτό παράγουμε ένα test bench το οποίο δοκιμάζει κάθε πιθανή είσοδο του αποκωδικοποιητή.

--Test Bench for dec\_3\_to\_8\_behavioral

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.NUMERIC\_STD.ALL;

entity dec\_3\_to\_8\_behavioral\_tb is

end entity;

architecture tb of dec\_3\_to\_8\_behavioral\_tb is

---------------------------------------------------------------

-- COMPONENT

---------------------------------------------------------------

component dec\_3\_to\_8\_behavioral is

port (

i\_enc : in std\_logic\_vector(2 downto 0);

o\_dec : out std\_logic\_vector(7 downto 0)

);

end component;

---------------------------------------------------------------

-- SIGNALS

---------------------------------------------------------------

signal i\_enc : std\_logic\_vector(2 downto 0) := (others => '0');

signal o\_dec : std\_logic\_vector(7 downto 0) := (others => '0');

---------------------------------------------------------------

-- CONSTANTS

---------------------------------------------------------------

constant TIME\_DELAY : time := 10 ns;

begin

DUT : dec\_3\_to\_8\_behavioral

port map (

i\_enc => i\_enc,

o\_dec => o\_dec

);

STIMULUS : process

begin

---------------------------------------------------------------

-- INITIALIZE SIGNALS

i\_enc <= (others => '0');

wait for (1 \* TIME\_DELAY);

---------------------------------------------------------------

-- EXAMPLE INPUTS

for i in 1 to 7 loop

i\_enc <= std\_logic\_vector(to\_unsigned(i,3));

wait for (1 \* TIME\_DELAY);

end loop;

i\_enc <= (others => '0');

wait for (1 \* TIME\_DELAY);

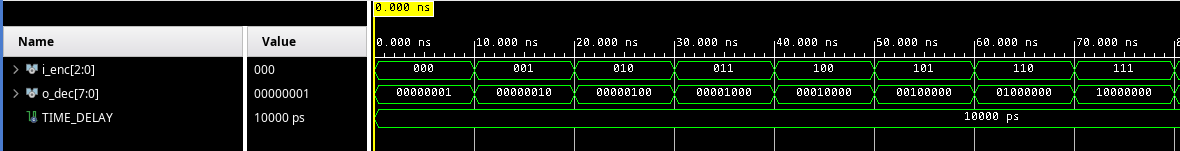
wait;

end process;

end architecture;

Είναι προφανές ότι ο ίδιος κώδικας μπορεί να χρησιμοποιηθεί και για τον έλεγχο της dataflow σχεδίασης αλλάζοντας απλά τον τύπο του component.

Σε κάθε περίπτωση τρέχουμε την προσομοίωση και μπορούμε εύκολα να επιβεβαιώσουμε την ορθή λειτουργία του κυκλώματος.



**Άσκηση 2 (Β.2)**

Σε αυτήν την άσκηση καλούμαστε να σχεδιάσουμε έναν καταχωρητή αριστερή και δεξιάς ολίσθησης βασισμένοι σε δεδομένο κώδικα. Στην πραγματικότητα αρκεί να χρησιμοποιήσουμε μία επιπλέον είσοδο (την lr στην περίπτωσή μας) για να ελέγξουμε πως θα μετακινηθεί το περιεχόμενο του καταχωρητή, πως θα εισαχθούν τα δεδομένα από την σειριακή είσοδο si και ποιο bit θα εξαχθεί στη σειριακή έξοδο so.

library IEEE;

use IEEE.std\_logic\_1164.all;

entity shift\_reg3 is

port(

clk,rst,si,en,pl,lr: in std\_logic; --lr for left/right

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 --parallel load

dff<=din;

elsif en='1' then --shift

if lr='1' then --lr = 1 -> right

dff<=si&dff(3 downto 1); --right shift input

elsif lr='0' then --lr = 0 -> left

dff<=dff(2 downto 0)&si; --left shift input

end if;

end if;

end if;

end process;

so <=

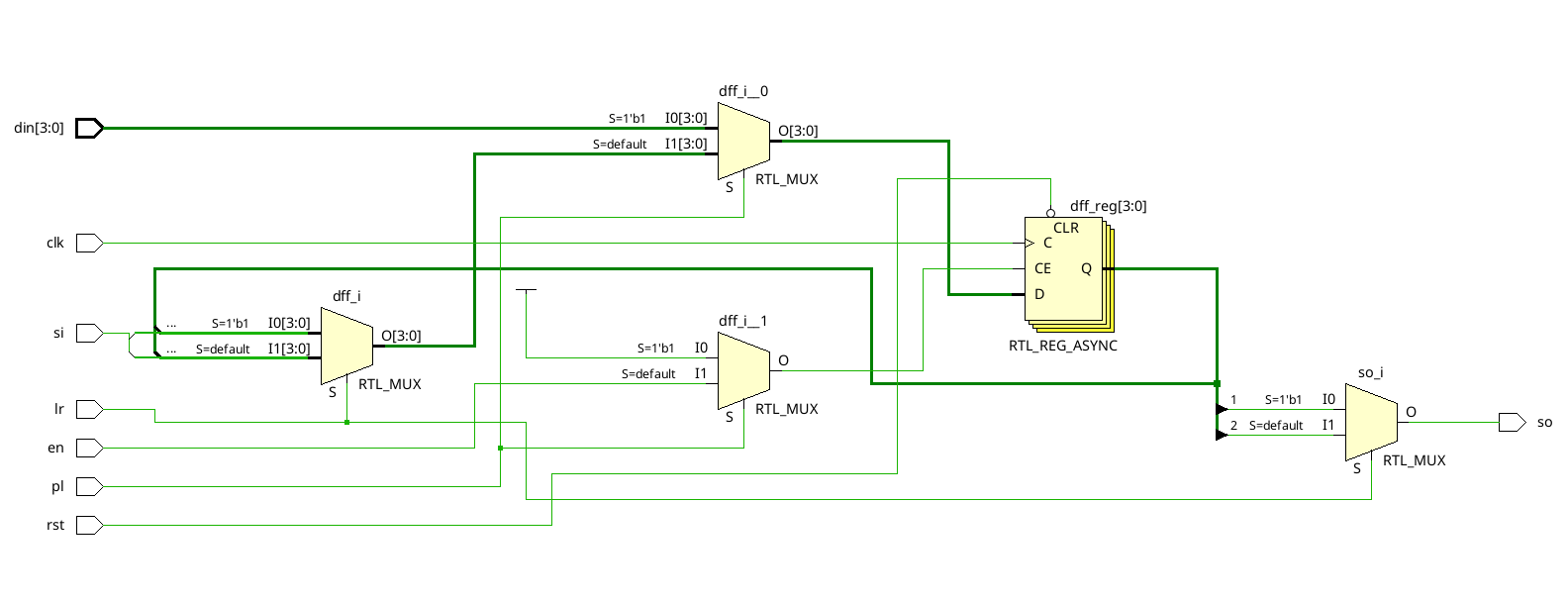
dff(0) when lr='1' else

dff(3) when lr='0';

end rtl;

Για την επιλογή της ολίσθησης των δεδομένων αρκεί μία δομή if/else εσωτεικά της διεργασίας ενώ για την επιλογή της εξόδου χρησιμοποιούμε μία δομή when/else.

Παρακάτω βλέπουμε το κύκλωμα το οποίο παράγεται κατά την προσομοίωση:



Συγκρίνοντάς το με το κύκλωμα δεξιάς ολίσθησης, βλέπουμε ότι υπάρχουν δύο παραπάνω πολυπλέκτες. Όπως ήταν αναμενόμενο ο ένα πολυπλέκτης ελέγχει το σήμα εξόδου so ενώ ο άλλος (στην πραγματικότητα αποτελείται από 4 πολυπλέκτες) καθορίζει την ολίσθηση των δεδομένων.

Για τον έλεγχο λειτουργίας του κυκλώματος σχεδιάζουμε ένα test bench όπου εκτελεί την παρακάτω ρουτίνα.

* Αρχικοποιεί τον καταχωρητή θέτοντας το rst = 0
* Φορτώνει παράλληλα το σήμα ‘1111’
* Εκτελεί 3 δεξιές ολισθήσεις
* Εκτελεί 3 αριστερές ολισθήσεις
* Θέτει την είσοδο si σε 1 και εκτελεί άλλες 4 δεξιές ολισθήσεις

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.NUMERIC\_STD.ALL;

entity shift\_reg3\_tb is

end entity;

architecture tb of shift\_reg3\_tb is

---------------------------------------------------------------

-- COMPONENT

---------------------------------------------------------------

component shift\_reg3 is

port (

clk,rst,si,en,pl,lr: in std\_logic; --lr for left/right

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

so: out std\_logic

);

end component;

---------------------------------------------------------------

-- SIGNALS

---------------------------------------------------------------

signal t\_din : std\_logic\_vector(3 downto 0) := (others => '1');

signal t\_rst : std\_logic := '1';

signal t\_en : std\_logic := '0';

signal t\_si : std\_logic := '0';

signal t\_pl : std\_logic := '0';

signal t\_lr : std\_logic := '1';

signal t\_so : std\_logic;

signal t\_clk : std\_logic := '0';

---------------------------------------------------------------

-- CONSTANTS

---------------------------------------------------------------

constant CLOCK\_PERIOD : time := 100 ns;

begin

DUT : shift\_reg3

port map (

din => t\_din,

rst => t\_rst,

en => t\_en,

si => t\_si,

pl => t\_pl,

lr => t\_lr,

so => t\_so,

clk => t\_clk

);

STIMULUS : process

begin

---------------------------------------------------------------

-- EXAMPLE INPUTS

t\_rst <= '0'; --reset

wait for (1 \* CLOCK\_PERIOD);

t\_rst <= '1';

wait for (2 \* CLOCK\_PERIOD);

t\_pl <= '1'; --load

wait for (1 \* CLOCK\_PERIOD);

t\_pl <='0';

wait for (2 \* CLOCK\_PERIOD);

t\_en <= '1'; --shift 3 times right

wait for (3 \* CLOCK\_PERIOD);

t\_lr <= '0'; --shift 3 times left

wait for (3 \* CLOCK\_PERIOD);

t\_en <= '1';

t\_si <= '1';

wait for (4 \* CLOCK\_PERIOD);

wait;

end process;

GEN\_CLK : process

begin

t\_clk <= '0';

wait for (CLOCK\_PERIOD / 2);

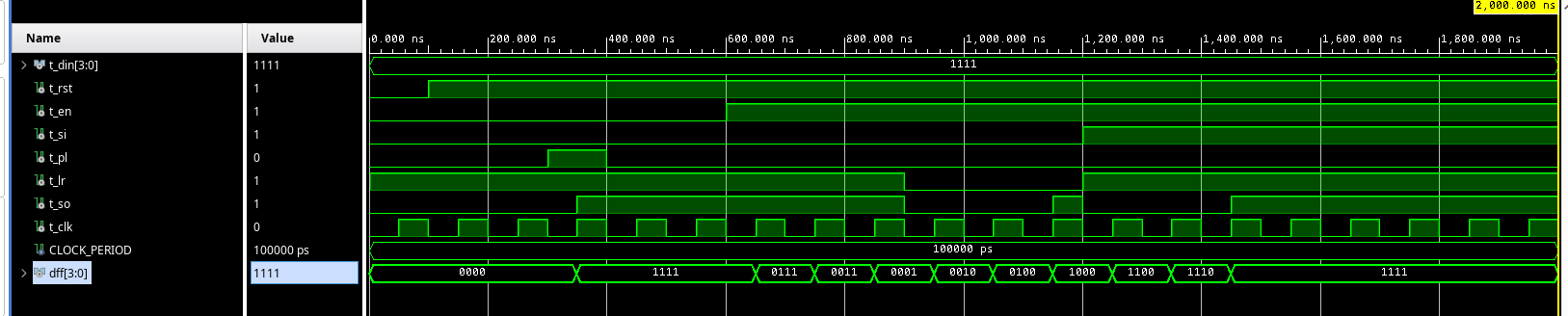
t\_clk <= '1';

wait for (CLOCK\_PERIOD / 2);

end process;

end architecture;

Παρακάτω βλέπουμε το αποτέλεσμα της προσομοίωσης που μας επιβεβαιώνει ότι ο σχεδιασμός είναι σωστός.



**Άσκηση 3(Β.3)**

**1.** Σκοπός αυτού του ερωτήματος είναι να τροποποιήσουμε τον δεδομένο κώδικα περιγραφής ενός μετρητή προσθέτωντας μία είσοδο ελέγχου για μέτρηση προς τα πάνω ή προς τα κάτω.

Κατά πλήρη αντιστοιχία με την προηγούμενη άσκηση αρκεί να προσθέσουμε μία δομή case για την αλλαγή των περιεχόμενων του μετρητή (αύξηση ή αφαίρεση) και να τροποποιήσουμε κατάλληλα το cout για το ενδεχόμενο όπου το αποτέλεσμα γίνεται αρνητικό.

Στο παράδειγμα που θα εξετάσουμε χρησιμοποιούμε τον κώδικα για την αρχιτεκτονική χωρίς όριο, αλλά η ίδια ακριβώς υλοποίηση μπορεί να εφαρμοστεί και στην περίπτωση της αρχιτεκτονικής με όριο (όπως φαίνεται στα σχόλια).

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;

ud :in std\_logic; --up/down control

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

cout :out std\_logic

);

end entity;

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

case ud is

when '1'=> count<=count+1;

when '0'=> count<=count-1;

when others => count<=(others=>'-');

end case;

end if;

end if;

end process;

sum<=count;

cout<='1' when (count=7 and count\_en='1' and ud='1') or (count=0 and count\_en='1' and ud='0')

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

-- case ud is

-- when '1'=>

-- if count/=3 then

-- count<=count+1;

-- else

-- count<=(others=>'0');

-- end if;

-- when '0'=>

-- if count/=3 then

-- count<=count-1;

-- else

-- count<=(others=>'1');

-- end if;

-- when others => count<=(others=>'-');

-- end case;

-- end if;

-- end if;

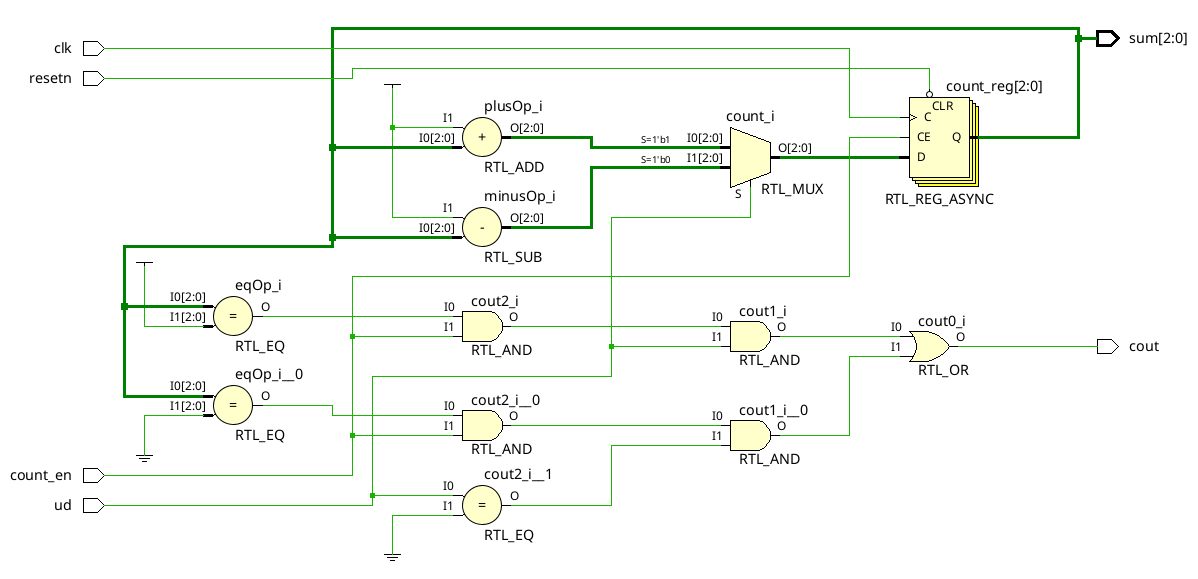
-- end process;

-- sum<=count;

-- cout<='1' when (count=7 and count\_en='1' and ud='1') or (count=0 and count\_en='1' and ud='0')

-- else '0';

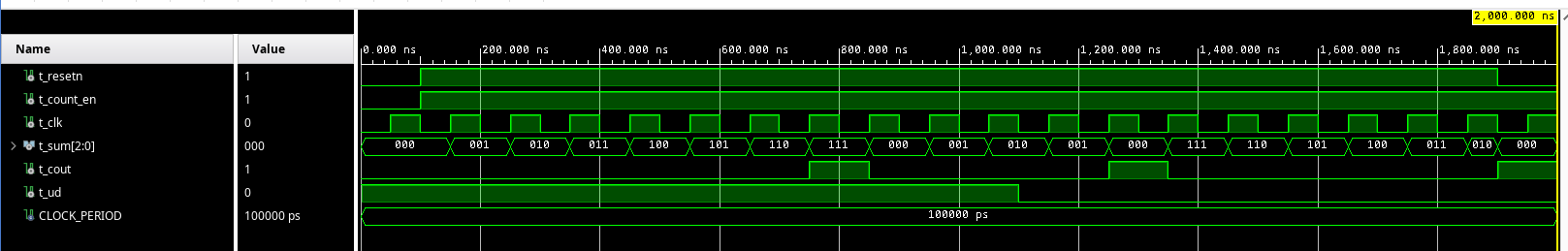
--end rtl\_limit;



Για την δοκιμή του κυκλώματος κατασκευάζουμε ένα απλό test bench το οποίο:

* Αρχικοποιεί τον μετρητή
* Μετράει προς τα πάνω για 10 κύκλους ρολογιού
* Μετράει προς τα κάτω για 8 κύκλους ρολογιού
* Μηδενίζει την τιμή του

Το αποτέλεσμα της προσομοίωσης φαίνεται παρακάτω και επιβεβαιώνει την ορθή λειτουργία του κυκλώματος.



**2.** Τέλος τροποποιούμε ξανά τον μετρητή προσθέτωντας μία παράλληλη είσοδο η οποία περιορίζει το εύρος μέτρησης μέσω της modulo.

Αξίζει να σημειωθεί ότι η VHDL περιλαμβάνει στη βιβλιοθήκη της υλοποιημένη την πράξη modulo για ακέραιους αριθμούς οπότε -δίνοντας ιδιαίτερη προσοχή στις μετατροπές τύπων- μπορούμε να πραγματοποιήσουμε την ίδια πράξη και για σήματα της μορφής *std\_logic\_vector*.

library IEEE;

use IEEE.std\_logic\_1164.all;

use IEEE.std\_logic\_unsigned.all;

use IEEE.numeric\_std.ALL;

entity count3\_mod is

Port (

clk,

resetn,

count\_en :in std\_logic;

count\_mod :in std\_logic\_vector (2 downto 0);

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

cout :out std\_logic

);

end entity;

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

count <= std\_logic\_vector(to\_unsigned(to\_integer(unsigned(count+1)) mod to\_integer(unsigned(count\_mod)), 3));

end if;

end if;

end process;

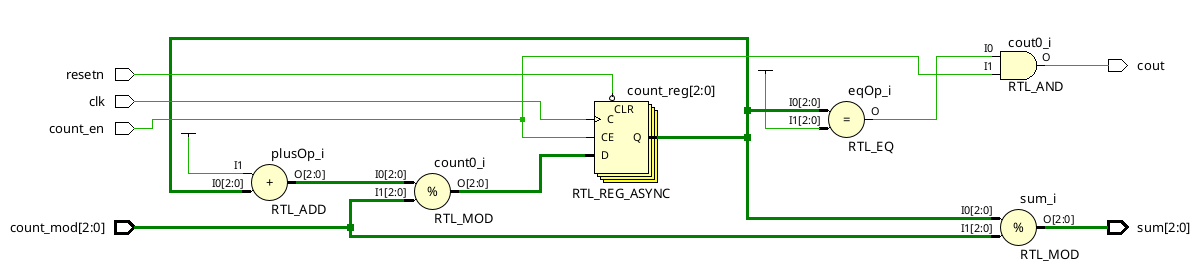
sum <= std\_logic\_vector(to\_unsigned(to\_integer(unsigned(count)) mod to\_integer(unsigned(count\_mod)), 3));

cout<='1' when count=7 and count\_en='1'

else '0';

end rtl\_limit;

Παρακάτω βλέπουμε το διάγραμμα του κυκλώματος:



Για τον έλεγχο του κυκλώματος φτιάχνουμε ένα test bench το οποίο:

* Αρχικοποιεί τον μετρητή
* Μετράει για 8 κύκλους ρολογιόυ έχοντας όριο το "111" (7)
* Μετράει για 6 κύκλους ρολογιόυ έχοντας όριο το "100" (4)
* Σταματάει για 2 κύκλους ρολογιόυ αλλάζοντας ταυτόχρονα το όριο σε "010" (2)
* Μετράει για 4 κύκλους ρολογιόυ έχοντας όριο το "010" (2)

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.NUMERIC\_STD.ALL;

entity count3\_tb is

end entity;

architecture tb of count3\_tb is

---------------------------------------------------------------

-- COMPONENT

---------------------------------------------------------------

component count3 is

port (

clk,

resetn,

count\_en :in std\_logic;

ud :in std\_logic;

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

cout :out std\_logic

);

end component;

---------------------------------------------------------------

-- SIGNALS

---------------------------------------------------------------

signal t\_resetn : std\_logic := '1';

signal t\_count\_en : std\_logic := '0';

signal t\_clk : std\_logic := '0';

signal t\_sum : std\_logic\_vector (2 downto 0) := (others=>'0');

signal t\_cout : std\_logic := '0';

signal t\_ud : std\_logic := '1';

---------------------------------------------------------------

-- CONSTANTS

---------------------------------------------------------------

constant CLOCK\_PERIOD : time := 100 ns;

begin

DUT : count3

port map (

clk => t\_clk,

resetn => t\_resetn,

count\_en => t\_count\_en,

sum => t\_sum,

cout => t\_cout,

ud => t\_ud

);

STIMULUS : process

begin

---------------------------------------------------------------

-- EXAMPLE INPUTS

t\_resetn <= '0';

wait for (1 \* CLOCK\_PERIOD);

t\_resetn <= '1';

t\_count\_en <= '1';

wait for (10 \* CLOCK\_PERIOD);

t\_ud <= '0';

wait for (8 \* CLOCK\_PERIOD);

t\_resetn <= '0';

wait for (1 \* CLOCK\_PERIOD);

t\_resetn <= '1';

wait;

end process;

GEN\_CLK : process

begin

t\_clk <= '0';

wait for (CLOCK\_PERIOD / 2);

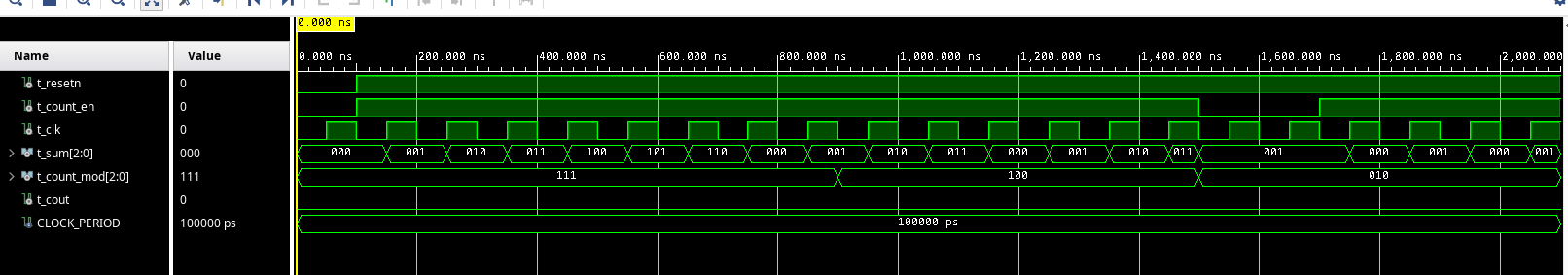
t\_clk <= '1';

wait for (CLOCK\_PERIOD / 2);

end process;

end architecture;

Όπως βλέπουμε στην παρακάτω προσομοίωση το κύκλωμα λειτουργεί όπως προβλέπεται.



Σε καθε περίπτωση βλέπουμε ότι η πολυπλοκότητα των κυκλώμάτων γίνεται όλο και μεγαλύτερη με αποτέλεσμα να είναι αρκετά δύσκολος ο σχεδιασμός "με το χέρι". Έτσι, βλέπουμε τη σπουδαιότητα των εργαλείων σχεδίασης που χρησιμοποιούμε.