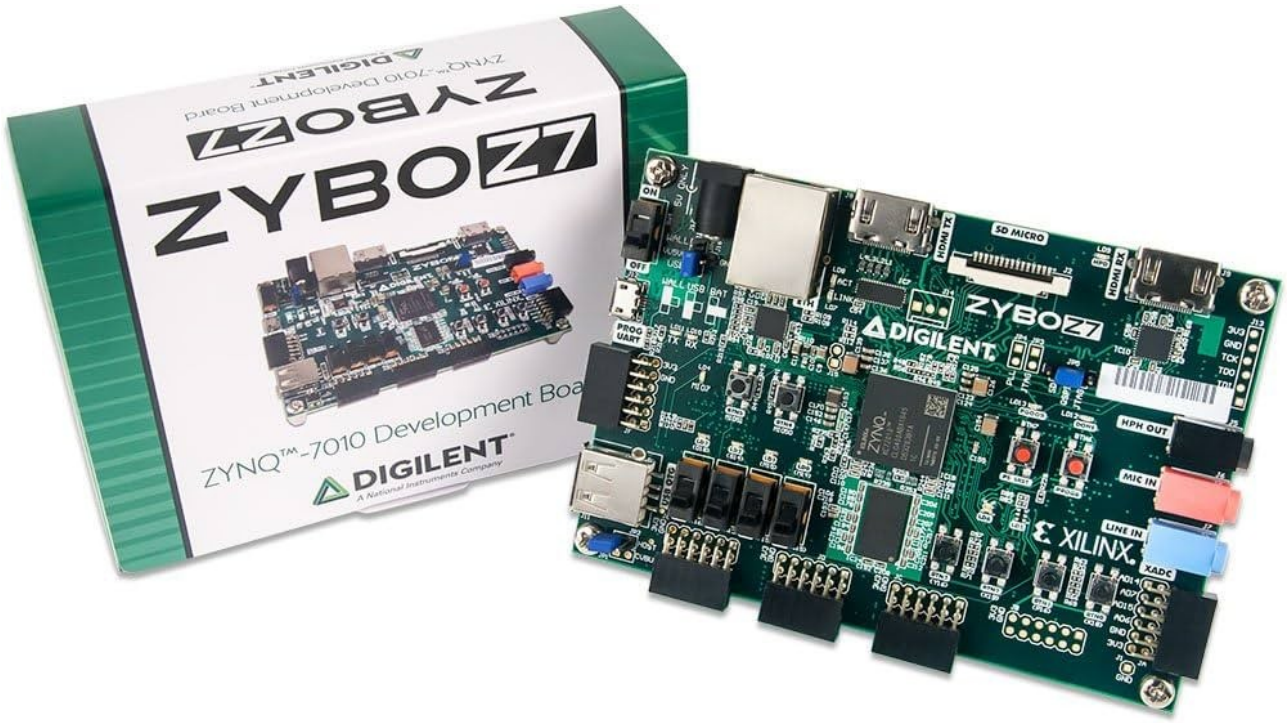
Ψηφιακά Συστήματα VLSI
1η Εργαστηριακή Άσκηση



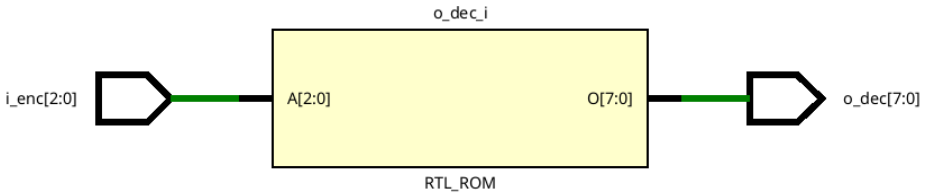
- **Ομάδα: 6**
- **Μέλη: Ακύλας Αντώνιος 03121152 & Ιωάννης-Χρυσοβαλάντης Κουμπιάς 03121053**

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;
```

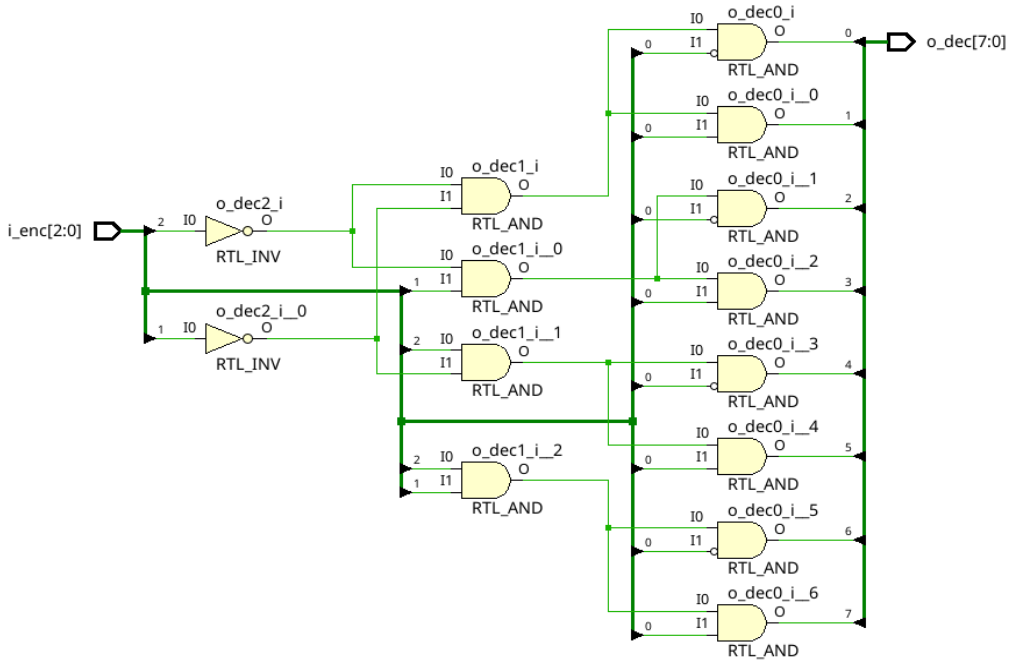
Όπως βλέπουμε με αυτόν τον τρόπο σχεδίασης η "λογική" του κυκλώματος παρουσιάζεται σαν black box.



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".



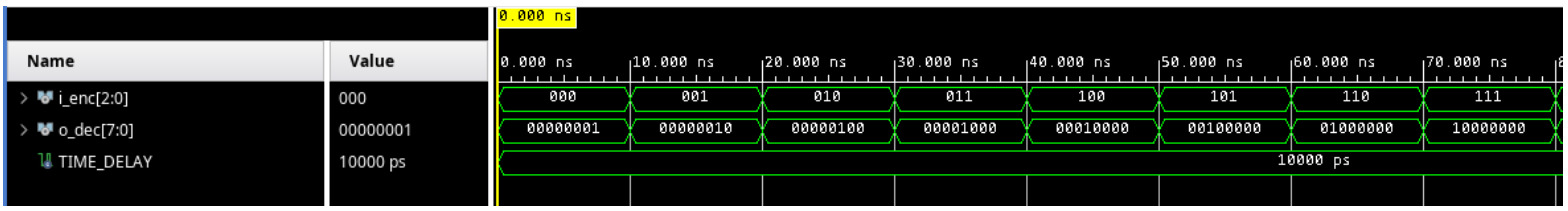
Στη συνέχεια καλούμαστε να επιβεβαιώσουμε την ορθή λειτουργία των προγραμμάτων μέσω προσομοίωσης. Για τον λόγο αυτό παράγουμε ένα test bench το οποίο δοκιμάζει κάθε πιθανή είσοδο του αποκωδικοποιητή.

```
--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.



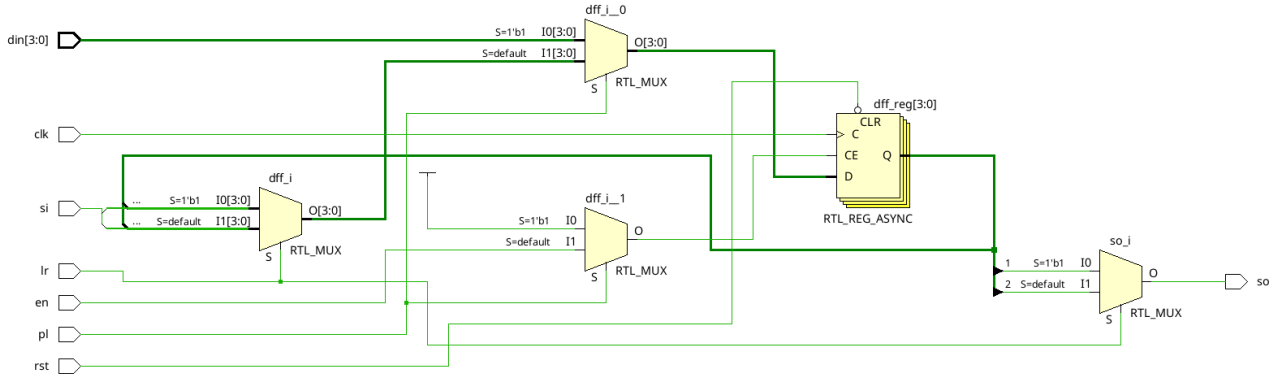
Άσκηση 2 (Β.2)

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.

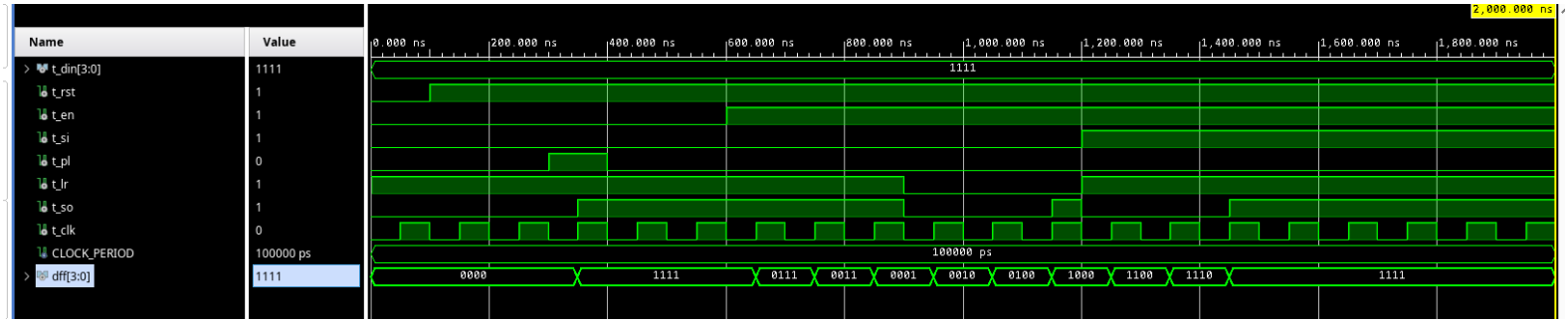
- Αρχικοποιεί τον καταχωρητή θέτοντας το rst = 0
- Φορτώνει παράλληλα το σήμα ‘1111’
- Εκτελεί 3 δεξιές ολισθήσεις
- Εκτελεί 3 αριστερές ολισθήσεις
- Θέτει την είσοδο si σε 1 και εκτελεί άλλες 4 δεξιές ολισθήσεις

```
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(B.3)

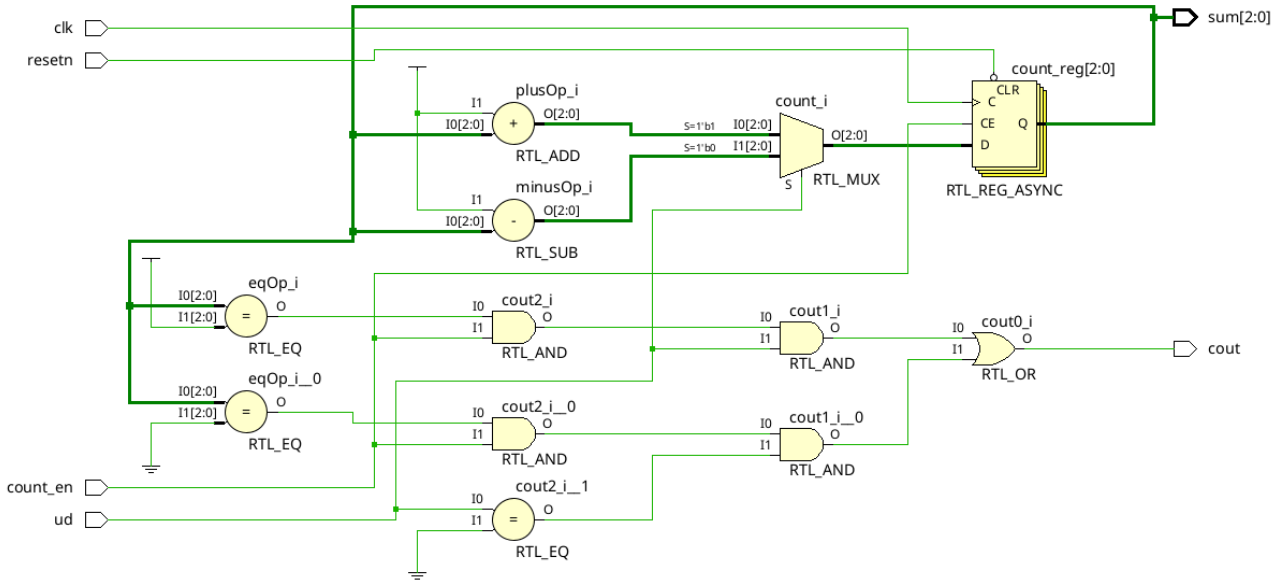
1. Σκοπός αυτού του ερωτήματος είναι να τροποποιήσουμε τον δεδομένο κώδικα περιγραφής ενός μετρητή προσθέτωντας μία είσοδο ελέγχου για μέτρηση προς τα πάνω ή προς τα κάτω.
Κατά πλήρη αντιστοιχία με την προηγούμενη άσκηση αρκεί να προσθέσουμε μία δομή case για την αλλαγή των περιεχόμενων του μετρητή (αύξηση 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')
```

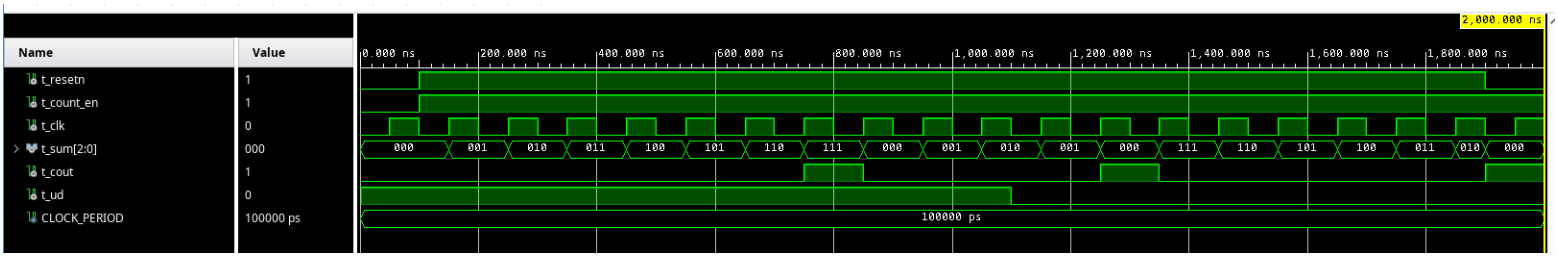
```
--              else '0';

--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.

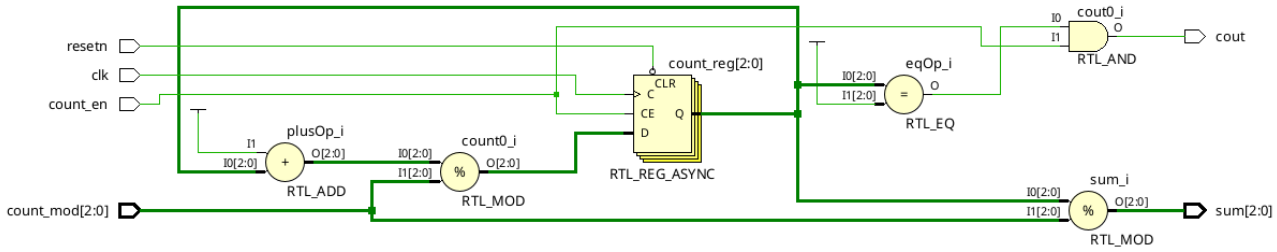


2. Τέλος τροποποιούμε ξανά τον μετρητή προσθέτωντας μία παράλληλη είσοδο η οποία περιορίζει το εύρος μέτρησης μέσω της modulo.
Αξίζει να σημειωθεί ότι η VHDL περιλαμβάνει στη βιβλιοθήκη της υλοποιημένη την πράξη modulo για ακέραιους αριθμούς οπότε -δίνοντας ιδιαίτερη προσοχή στις μετατροπές τύπων- μπορούμε 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:

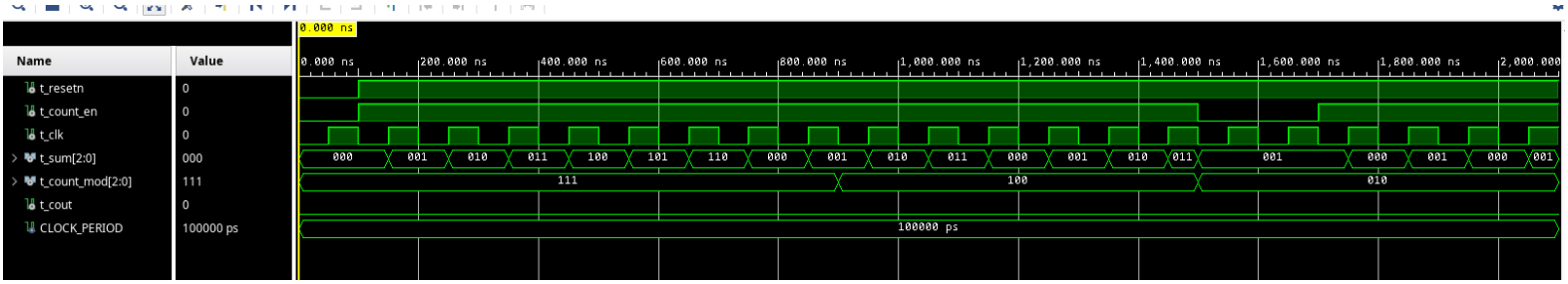
- Αρχικοποιεί τον μετρητή
- Μετράει για 8 κύκλους ρολογιού έχοντας όριο το "111" (7)
- Μετράει για 6 κύκλους ρολογιού έχοντας όριο το "100" (4)
- Σταματάει για 2 κύκλους ρολογιού αλλάζοντας ταυτόχρονα το όριο σε "010" (2)
- Μετράει για 4 κύκλους ρολογιού έχοντας όριο το "010" (2)

```
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.