2023

Αναφορά 1ης εργαστηριακής Άσκησης

Εργαστήριο VLSI

ΠΑΠΑΔΟΠΟΥΛΟΣ ΣΠΥΡΙΔΩΝ (ΑΜ):03120033
ΕΜΜΑΝΟΥΗΛ ΞΕΝΟΣ (ΑΜ):03120850

Ζήτημα 1ο
Δυαδικός Αποκωδικοποιητής 3 σε 8

Στο ζήτημα αυτό ζητείται να κατασκευάσουμε έναν δυαδικό αποκωδικοποιητή 3 σε 8 περιγράφοντας την αρχιτεκτονική του σε dataflow και behavioral VHDL.

Ξεκινώντας από την behavioral VDHL το δυαδικού αποκωδικοποιητή 3 σε 8:

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

entity dec3to8_b is
    Port (
        enc : in STD_LOGIC_VECTOR(2 downto 0);
        dec : out STD_LOGIC_VECTOR(7 downto 0)
    );
end dec3to8_b;

architecture behavioral of dec3to8_b is
begin
    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 <= (others => '-');
        end case;
    end process;

end behavioral;
```

Αρχικά ορίζουμε το entity dec3to8 που είναι ο 3 σε 8 αποκωδικοποιητής. Στον αποκωδικοποιητή ορίζουμε ως είσοδο ένα vector μεγέθους 3 bit και ένα vector ως έξοδο μεγέθους 8 bit. Στην συνέχεια γράφουμε την behavioral αρχιτεκτονική. Σε αυτή χρησιμοποιούμε ένα μόνο process που έχει ως

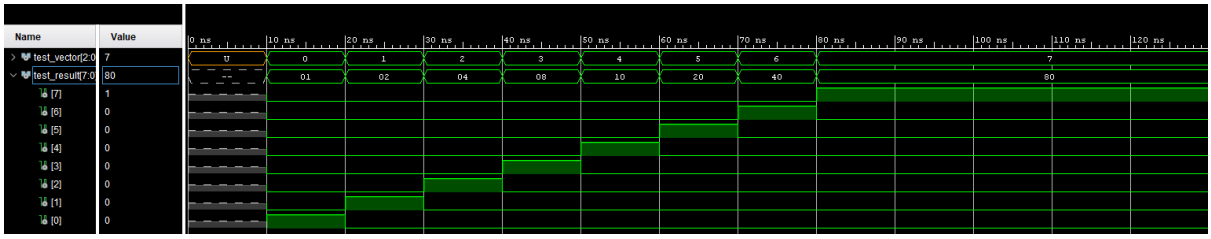
είσοδο την είσοδο του decoder. Σε αυτό το process χρησιμοποιούμε την δομή case όπου για κάθε περίπτωση εισόδου σε δυαδικό σύστημα εξάγουμε την κατάλληλη έξοδο. Στο τέλος ορίζουμε ότι αν η είσοδος δεν είναι κάποια από τις προβλεπόμενες τότε στην έξοδο δεν έχουμε τίποτα (στην προσομοίωση φαίνεται – στην έξοδο).

Για να ελέγξουμε ότι λειτουργεί σωστά η behavioral αρχιτεκτονική που παραθέσαμε παραπάνω χρησιμοποιούμε το ακόλουθο testbench:

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


entity testbenchBehavioral is
end testbenchBehavioral;

architecture Behavioral of testbenchBehavioral is
component dec3to8_b is
        Port (
            enc : in STD_LOGIC_VECTOR(2 downto 0);
            dec : out STD_LOGIC_VECTOR(7 downto 0)
        );
    end component dec3to8_b;

signal test_vector: std_logic_vector(2 downto 0);
signal test_result:std_logic_vector(7 downto 0);

begin
uut: dec3to8_b
    port map(
        enc=>test_vector,
        dec=>test_result);

testing: process
begin
    wait for 10 ns;
    test_vector <= "000";
    wait for 10 ns;
    test_vector <= "001";
    wait for 10 ns;
    test_vector <= "010";
    wait for 10 ns;
    test_vector <= "011";
    wait for 10 ns;
    test_vector <= "100";
    wait for 10 ns;
    test_vector <= "101";
    wait for 10 ns;
    test_vector <= "110";
    wait for 10 ns;
```

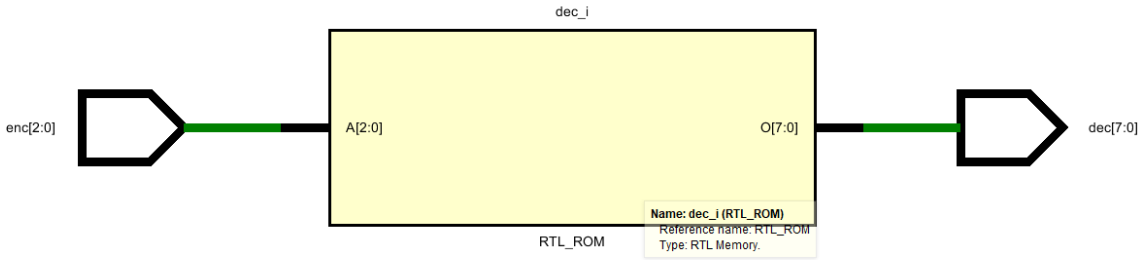
```
    test_vector <= "111";
    wait for 10 ns;
    wait;
end process testing;
end Behavioral;
```

Στο testbench αλλάζουμε ανά 10ns το test_vector που είναι ένα logic vector μεγέθους 3 και αποτελεί την είσοδο του decoder (πράγμα που έχει γίνει μέσω του port map). Κάνοντας το simulation ουσιαστικά αναμένουμε στην αρχή η έξοδος να είναι'—', καθώς δεν έχουμε ορίσει στην αρχή είσοδο, ενώ στην συνέχεια η έξοδος θα πρέπει να έχει την κατάλληλη μορφή σύμφωνα με την είσοδο και να αλλάζει ανά 10ns. Ακολουθεί το αποτέλεσμα της προσομοίωσης:



Βλέπουμε λοιπόν ότι ο decoder λειτουργεί σωστά καθώς στην αρχή η έξοδος είναι πράγματι '—' ενώ μετά ανά 10ns αλλάζει το bit που είναι '1'. Τα νούμερα που βλέπουμε εντός του παλμού στο test_result είναι σε δεκαεξαδικό σύστημα και βλέπουμε και μέσω αυτών ότι η έξοδος είναι η αναμενόμενη.

Στην συνέχεια ακολουθεί το RTL σχηματικό του κυκλώματος:



Βλέπουμε λοιπόν ότι το RTL σχηματικό του decoder με την behavioral αρχιτεκτονική είναι ουσιαστικά μία ROM που παίρνει ένα vector μεγέθους 3 bit και εξάγει ένα vector μεγέθους 7 bit.

Στην συνέχεια ακολουθεί η dataflow αρχιτεκτονική του δυαδικού 3 σε 8 αποκωδικοποιητή:

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

entity dec3to8 is
    Port (
        enc : in STD_LOGIC_VECTOR(2 downto 0);
        dec : out STD_LOGIC_VECTOR(7 downto 0)
    );
end dec3to8;

architecture Dataflow of dec3to8 is
    signal enc2_not : std_logic;
```

```
    signal enc1_not : std_logic;
    signal enc0_not : std_logic;
begin
    enc2_not <= not enc(2);
    enc1_not <= not enc(1);
    enc0_not <= not enc(0);
    dec(0) <= enc2_not and enc1_not and enc0_not;
    dec(1) <= enc2_not and enc1_not and enc(0);
    dec(2) <= enc2_not and enc(1) and enc0_not;
    dec(3) <= enc2_not and enc(1) and enc(0);
    dec(4) <= enc(2) and enc1_not and enc0_not;
    dec(5) <= enc(2) and enc1_not and enc(0);
    dec(6) <= enc(2) and enc(1) and enc0_not;
    dec(7) <= enc(2) and enc(1) and enc(0);
end Dataflow
```

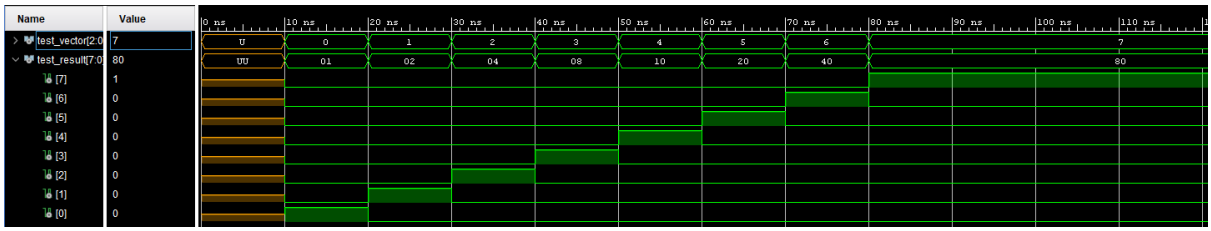
Στην dataflow αρχιτεκτονική ακολουθούμε την λογική ότι για να είναι ενεργό το n οστό bit στην έξοδο πρέπει η δεκαδική αναπαράσταση της εισόδους να είναι ίση με n. Έτσι για παράδειγμα για να είναι ενεργό το 3^{ο} bit η είσοδος πρέπει να είναι ίση με 011, για να είναι ενεργό το 6 bit θα πρέπει η είσοδος να είναι 110. Ένας εύκολος τρόπος να ελέγξουμε αν το n-οστό bit είναι ανοικτό θα πρέπει όπου υπάρχει 1 στην δυαδική αναπαράσταση του αριθμού το αντίστοιχο bit να είναι 1 και όπου υπάρχει 0 το αντίστοιχο bit να είναι 0. Έτσι, συνεχίζοντας τα προηγούμενα δύο παραδείγματά μας για να είναι ενεργό το 3^{ο} bit της εξόδου θα πρέπει το πρώτο bit της εισόδου να είναι 0, ή ισοδύναμα το not του πρώτου bit να είναι 1, και τα άλλα δύο ένα, ενώ για να είναι το 6ο bit της εξόδου ενεργό πρέπει τα δύο MSB να είναι ενεργά το 3^{ο} να είναι 0 ή ισοδύναμα το not του 3^{ου} bit να είναι ενεργό. Με βάση αυτό το σκεπτικό κατασκευάσαμε την dataflow αρχιτεκτονική του 3 σε 8 δυαδικού decoder. Για οικονομία υλικού κατασκευάζουμε εξ αρχής την άρνηση κάθε bit της εισόδου, αντί να το κατασκευάζομαι για κάθε bit της εξόδου.

Στην συνέχεια ακολουθεί το testbench της dataflow αρχιτεκτονικής όπου ελέγχουμε τα ίδια πράγματα με το testbench για την behavioral αρχιτεκτονική.

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

entity testbenchDataflow is
end testbenchDataflow;

architecture Behavioral of testbenchDataflow is
component dec3to8 is
        Port (
            enc : in STD_LOGIC_VECTOR(2 downto 0);
            dec : out STD_LOGIC_VECTOR(7 downto 0)
        );
    end component dec3to8;

signal test_vector: std_logic_vector(2 downto 0);
signal test_result:std_logic_vector(7 downto 0);

begin
```

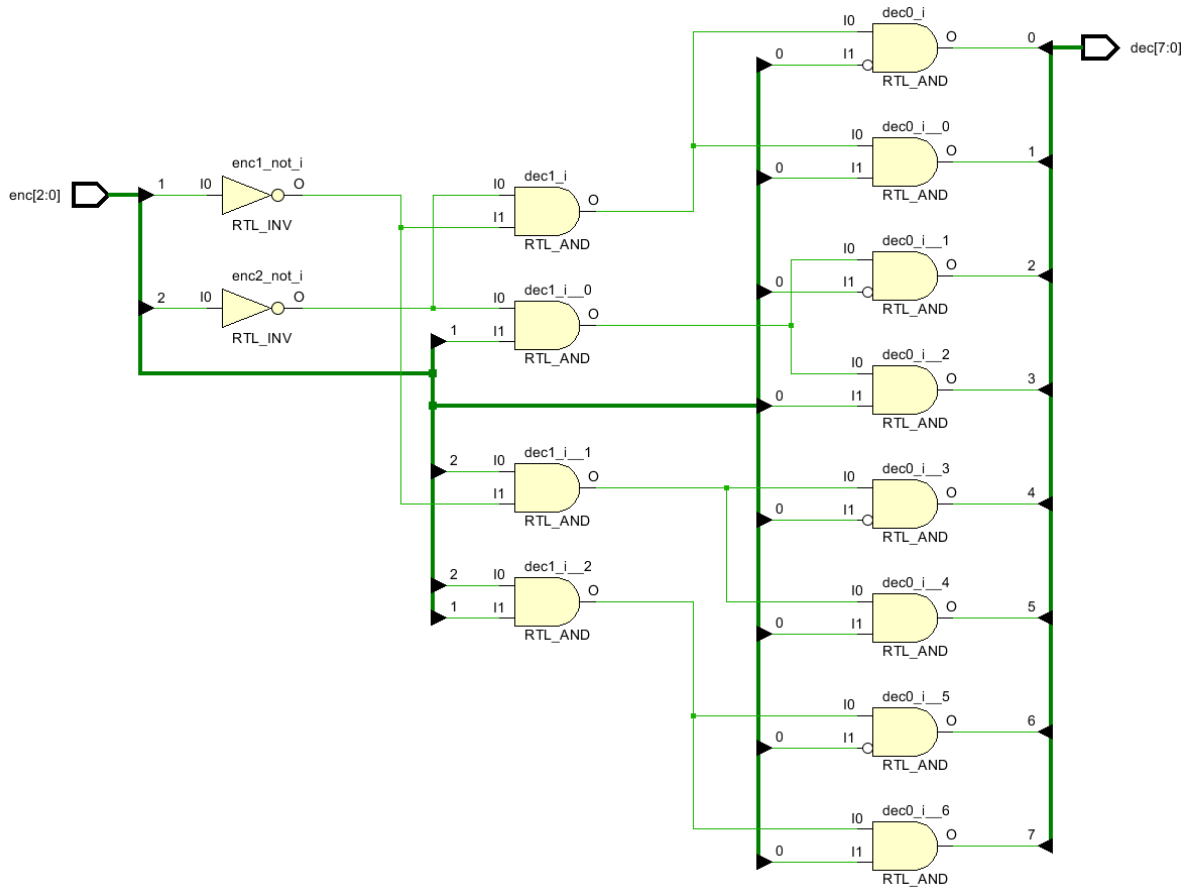
```
uut: dec3to8
    port map(
        enc=>test_vector,
        dec=>test_result);

testing: process
begin
    wait for 10 ns;
    test_vector <= "000";
    wait for 10 ns;
    test_vector <= "001";
    wait for 10 ns;
    test_vector <= "010";
    wait for 10 ns;
    test_vector <= "011";
    wait for 10 ns;
    test_vector <= "100";
    wait for 10 ns;
    test_vector <= "101";
    wait for 10 ns;
    test_vector <= "110";
    wait for 10 ns;
    test_vector <= "111";
    wait for 10 ns;
    wait;
end process testing;
end Behavioral;
```

Το αποτέλεσμα της προσομοίωσης του παραπάνω testbench είναι:



Βλέπουμε λοιπόν ότι όταν δεν έχουμε ορίσει τιμή στην είσοδο και η έξοδος είναι unidentified ενώ με την αύξηση του test_vector, που είναι είσοδος του decoder, ανά 10ns επιφέρει την σωστή αλλαγή στην έξοδο ανά 10ns.

Το RTL σχήμα της dataflow αρχιτεκτονικής είναι:



Βλέπουμε λοιπόν ότι το RTL σχήμα έφτιαξε ακριβώς το σχήμα που του περιγράψαμε, χωρίς καμία αλλαγή, πχ χρησιμοποίηση and 3 θυρών. Έτσι η RTL ανάλυση του dataflow δείχνει ακριβώς την υλοποίηση της dataflow αρχιτεκτονικής.

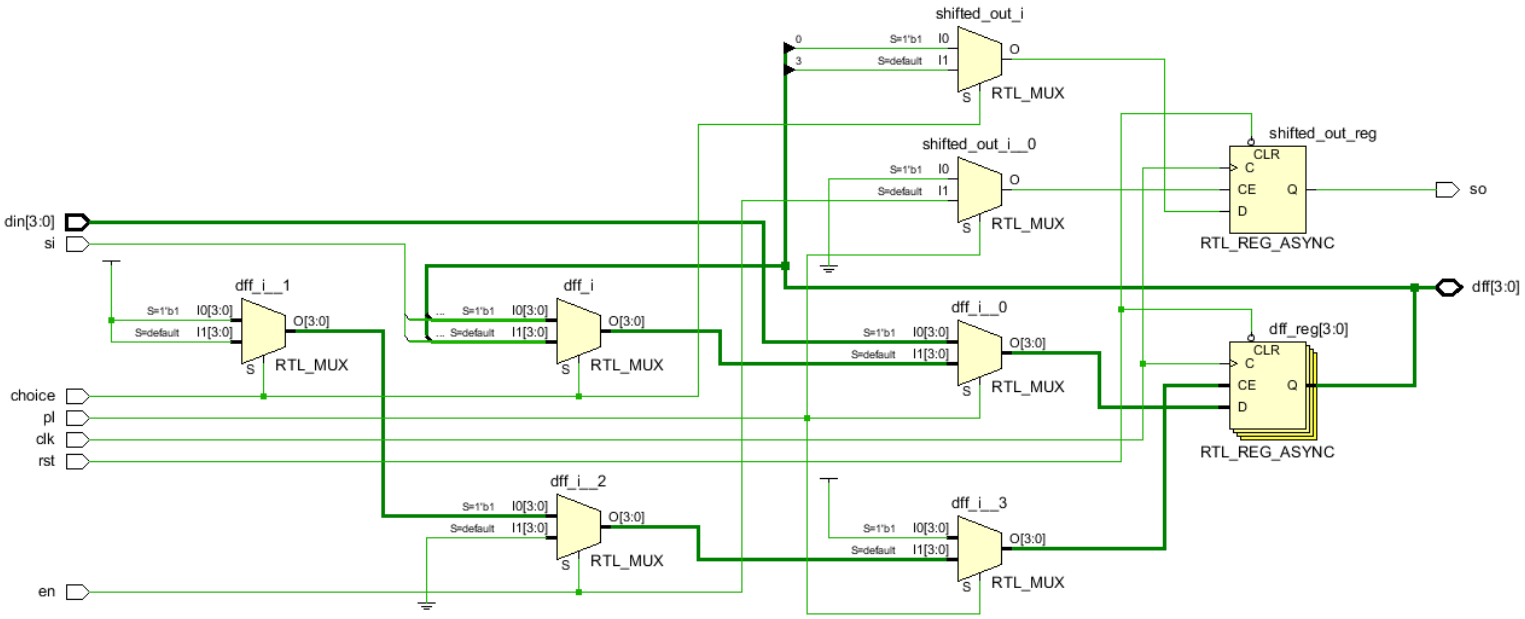
Ζήτημα δεύτερο
Καταχωρητής ολίσθησης των 4 bits με παράλληλη φόρτωση

Στην άσκηση αυτή μας ζητείται να φτιάξουμε ένα καταχωρητή ολίσθησης των 4bit με παράλληλη φόρτωση. Για την υλοποίηση του καταχωρητή αυτού αξιοποιήσαμε τον κώδικα που μας δίνεται στην εκφώνηση της άσκησης. Αρχικά παραθέτουμε τον κώδικα σε VHDL για την υλοποίηση του καταχωρητή ολίσθησης 4-bit με την επιπλέον δυνατότητα επιλογής μεταξύ αριστερής και δεξιά ολίσθησης.

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

entity shift_reg3 is
    port (
        clk,rst,si,en,pl,choice: in std_logic;
        din: in std_logic_vector(3 downto 0);
        dff: inout std_logic_vector(3 downto 0); --change dff to inout in order to see it in the
simulation
```

```
        so: out std_logic);
end shift_reg3;

architecture rtl of shift_reg3 is
    signal shifted_out: std_logic; -- the value of the bit that is now out of the register
begin
    edge: process (clk,rst)
    begin
        if rst='0' then                 -- check if we need to reset
            dff<=(others=>'0');         -- set all reg bits to 0
            shifted_out <= '0';
        elsif clk'event and clk='1' then -- check the rising edge of the clock
                if pl='1' then          -- if pl=1 parallel load in the register
                    dff<=din;
                    elsif en='1' then               -- check if enable=1
                        if choice='1' then          -- else check for choice=1 for right shift
                        shifted_out <= dff(0);
                        dff<=si&dff(3 downto 1);     -- reset the register content correctly
                                                     -- change only the MSB  with si
                        else
                            shifted_out <= dff(3);
                            dff<=dff(2 downto 0)&si;  -- reset the register content correctly
                                                      -- change only the LSB with si
                        end if ;
                end if;
        end if;
    end process;
    so<=shifted_out;    -- set the shifted bit in the output
end rtl;
```

Αρχικά, στον κώδικα μας ορίζουμε το entity τις εισόδους και τις εξόδους. Επιπλέον ορίζουμε και το dff ως inout προκειμένου να μπορούμε να δούμε την τιμή του στο simulation. Στην συνέχεια ορίζουμε την αρχιτεκτονική του καταχωρητή ολίσθησης. Πρώτα ορίζουμε ένα σήμα shifted out στο οποίο θα αποθηκεύσουμε την τιμή που θα περάσουμε στο so, δηλαδή την τιμή που θα φύγει από τον καταχωρητή. Το πρώτο που ελέγχουμε στον κώδικα μας είναι το reset, καθώς αν είναι αυτό 0 πρέπει να κάνουμε reset το register. Αν το reset είναι 1 τότε ελέγχουμε την θετική ακμή του ρολογιού για να πραγματοποιήσουμε μία από τις λειτουργείες του register. Στην περίπτωση της θετικής ακμής του ρολογιού πρώτα ελέγχουμε αν η παράλληλη φόρτωση είναι 1 . Στην περίπτωση αυτή φορτώνουμε την είσοδο στον καταχωρητή με απευθείας ανάθεση. Εφόσον το pl=0 ελέγχουμε αν το enable είναι ενεργό. Αν αυτό είναι ενεργό τότε ελέγχουμε το choice το οποίο αν είναι 1 πραγματοποιούμε μία δεξιά ολίσθηση , ειδάλλως πραγματοποιούμε μία αριστερή ολίσθηση. Τέλος, ανεξάρτητα από το ρολόι αναθέτουμε στο so την τιμή που αντιστοιχεί στην τελευταία έξοδο του καταχωρητή. Αυτή είναι η υλοποίηση του καταχωρητή σε VHDL. Στην συνέχεια παρουσιάζουμε την υλοποίηση που μας δίνει το Vivado για το RTL επίπεδο:



Θα αναλύσουμε στην συνέχεια την ορθότητα της σχεδίασης του RTL design που μας δίνει το Vivado. Αρχικά βλέπουμε ότι όλα τα if υλοποιούνται με την χρήση πολυπλέκτη, ενώ έχουμε και κάποιους registers που κρατάνε τα dff και το so. Βλέπουμε ότι η τιμή του pl πηγαίνει κατευθείαν στον ακροδέκτη επιλογής του πολυπλέκτη, που καθορίζει την είσοδο των D flip-flop που υλοποιούν τον καταχωρητή. Συνεπώς ορθά το pl μέσω ενός πολυπλέκτη καθορίζει την νέα τιμή του καταχωρητή, αν αυτός πρέπει να βρίσκεται σε λειτουργία. Το clk προφανώς πηγαίνει στην είσοδο που αντιστοιχεί στο ρολόι για τα D flip-flop τόσο του καταχωρητή όσο και της εξόδου so. Η τιμή en καθορίζει μέσω του πολυπλέκτη dff_i__2 αν το CE θα είναι 1 ή 0 έμμεσα, καθώς η έξοδος του αντίστοιχου πολυπλέκτη πηγαίνει στην είσοδο ενός πολυπλέκτη με σήμα επιλογής το pl . Αν αυτό είναι 0 τότε θέλουμε σίγουρα το CE να είναι ενεργό για να πραγματοποιηθεί η παράλληλη φόρτωση ειδάλλως το αποτέλεσμα εξαρτάται από την τιμή του en. Επιπλέον το en είναι είσοδος σε έναν πολυπλέκτη όπου εκεί το σήμα επιλογής είναι το 0. Σε αυτή την περίπτωση θέλουμε, εάν το pl είναι 0 να γίνει παράλληλη φόρτωση και να μην είναι ενεργό το CE του flip flop του so(για αυτό έχει την τιμή 0 ο αντίστοιχος ακροδέκτης του πολυπλέκτη), ειδάλλως θέλουμε το CE του so να καθορίζεται από το enable, καθώς αν αυτό είναι θετικό και pl=1 θέλουμε να πραγματοποιήσουμε κάποια λειτουργία ολίσθησης. Ο πολυπλέκτης diff_i παίρνει σαν εισόδους την παλιά τιμή του καταχωρητή μετατοπισμένη κατά μία θέση δεξιά και μία θέση αριστερά, ενώ σαν σήμα επιλογής παίρνει το choice. Συνεπώς, μέσω αυτού του καταχωρητή επιτυγχάνεται η επιλογή ανάμεσα σε δεξιά και αριστερή ολίσθηση. Τέλος, ο πολυπλέκτης shifted_out_i παίρνει το MSB και το LSB της παλιάς τιμής του καταχωρητή και σχηματίζει την είσοδο του flip flop του so με βάση το σήμα choice. Συμπεραίνουμε, λοιπόν ότι αυτός ο πολυπλέκτης υλοποιεί την επιλογή του κατάλληλου κρατουμένου εξόδου. Η υλοποίηση σε RTL που μας έδωσε το Vivado έχει έναν πολυπλέκτη που δεν χρειάζεται τον dff_i__1, του οποίου η έξοδος είναι πάντα ίση με 1 για όλα τα καλώδια. Στην υλοποίηση που μας έδωσε το Vivado επίσης χρησιμοποιήθηκε ένα παραπάνω flip flop για το so, το οποίο έγινε λόγω των δύο διαφορετικών πιθανών τιμών που μπορεί να πάρει το so ανάλογα με την τιμή του choice. Βέβαια σαν βελτιστοποίηση θα μπορούσε αντί για ένα επιπλέον flip flop το κύκλωμα να παίρνει τις εξόδους των dff(0) και dff(3) από έναν πολυπλέκτη και ανάλογα με την τιμή του choice να επιλέγεται η κατάλληλη τιμή για την έξοδο so, όπως γίνεται στο διάγραμμα που μας

δόθηκε. Εντούτοις σε γενικές γραμμές το Vivado αν εξαιρέσουμε τον περιττό πολυπλέκτη που αναφέραμε παραπάνω μας έδωσε ένα αποδοτικό κύκλωμα σε επίπεδο RTL.

Στην συνέχεια της ανάλυσης μας παρουσιάζουμε τον κώδικα μας σε VHDL για την υλοποίηση του testbench της παραπάνω περιγραφής:

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

entity tb_shift_register is
end tb_shift_register;

architecture tb_arch of tb_shift_register is
    -- Constants
    constant CLK_PERIOD : time := 5 ns; -- Clock period
    -- Signals
    component shift_reg3 is
        port (
            clk,rst,si,en,pl,choice: in std_logic;
            din: in std_logic_vector(3 downto 0);
            dff: inout std_logic_vector(3 downto 0);
            so: out std_logic);
    end component;
    -- create all test signals
    signal tb_clk,tb_si,tb_rst,tb_pl,tb_en,tb_choice : std_logic;
    signal tb_din,tb_dff_value : std_logic_vector (3 downto 0);
    signal so_actual : std_logic;
begin
    uut: shift_reg3
    port map(           -- map the entity input and output to the component
        clk => tb_clk,
        si => tb_si,
        rst => tb_rst,
        pl => tb_pl,
        en => tb_en,
        choice => tb_choice,
        din => tb_din,
        so => so_actual,
        dff => tb_dff_value
    );

-- Clock process
    clk_process : process
    begin
        while true loop         -- create a loop for the clock
                                -- in order to make it work non stop
            tb_clk <= '0';
            wait for CLK_PERIOD / 2;
```

```
            tb_clk <= '1';
            wait for CLK_PERIOD / 2;
        end loop;
    end process;


    -- Stimulus process
     stim_process : process
       begin
       -- initialize input
       tb_rst <= '1';
       tb_si <= '0';
       tb_en <= '0';
       tb_pl <= '1';
       tb_choice <= '1';
       tb_din <= "1010";
       wait for CLK_PERIOD;

       -- check reset
       tb_rst <= '0';
       tb_si <= '0';
       tb_en <= '0';
       tb_pl <= '1';
       tb_choice <= '1';
       tb_din <= "1010";
       wait for CLK_PERIOD;

       -- store another result
       tb_rst <= '1';
       tb_si <= '1';
       tb_en <= '1';
       tb_pl <= '1';
       tb_choice <= '0';
       tb_din <= "0101";
       wait for CLK_PERIOD;

       -- check right shift
       tb_rst <= '1';
       tb_si <= '0';
       tb_en <= '1';
       tb_pl <= '0';
       tb_choice <= '1';
       tb_din <= "1010";
       wait for CLK_PERIOD;
       assert so_actual = '1' report "Right shift failed for input 0101"
severity error;

       -- check left shift
```

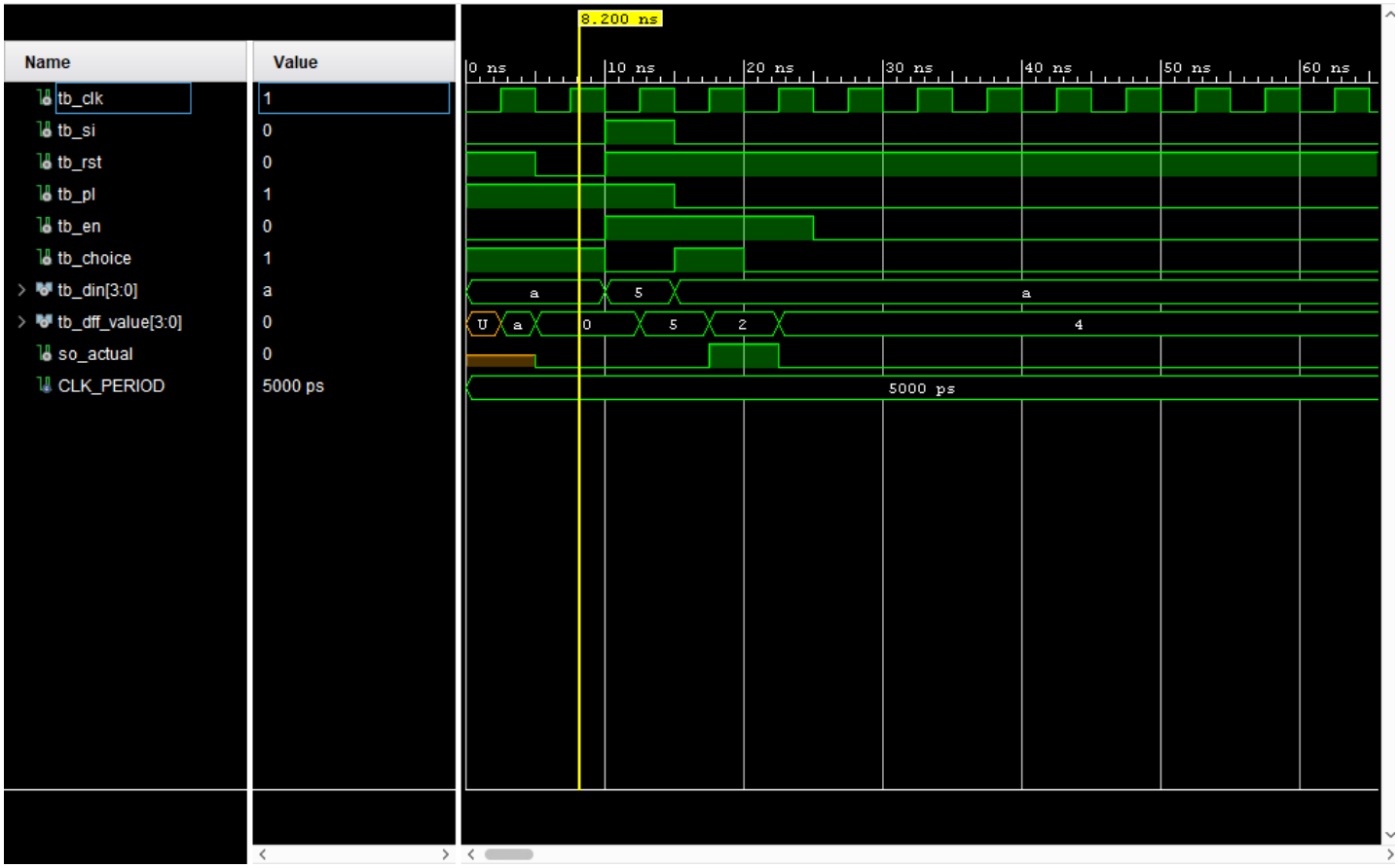
```
        tb_rst <= '1';
        tb_si <= '0';
        tb_en <= '1';
        tb_pl <= '0';
        tb_choice <= '0';
        tb_din <= "1010";
        wait for CLK_PERIOD;
        assert so_actual = '0' report "Right shift failed for input 0010"
severity error;

        --check enable
        tb_rst <= '1';
        tb_si <= '0';
        tb_en <= '0';
        tb_pl <= '0';
        tb_choice <= '0';
        tb_din <= "1010";
        wait for CLK_PERIOD;
        wait;
    end process;
end tb_arch;
```

Για τον παραπάνω κώδικα ακολουθούμε την τυπική δομή για την υλοποίηση ενός testbench. Αρχικά ορίζουμε το entity του testbench αυτού(απαραίτητο για την δημιουργία του instance του testbench). Στην συνέχεια προχωράμε στον ορισμό της αρχιτεκτονικής του testbench αυτού. Ορίζουμε το component που θα εξετάσουμε(ορίζοντας τις εισόδους και τις εξόδους του), καθώς και τα σήματα του testbench που θα χρησιμοποιηθούν για την παρακολούθηση των αποτελεσμάτων. Ακολούθως, ορίζουμε το uut (unit under test) ορίζοντας την αντιστοίχιση μεταξύ των σημάτων του testbench και των σημάτων του component που θέλουμε να εξετάσουμε. Έπειτα ορίζουμε μία διαδικασία για το ρολόι (την περίοδο του ρολογιού την ορίσαμε στην αρχή του testbench) η οποία θα τρέχει ένα while true loop προκειμένου να τρέχει συνεχώς και για πάντα το ρολόι. Σε χρόνο μίας ημιπεριόδου θέτουμε το ρολόι ίσο με το 0 και στην υπόλοιπη ίσο με 1 . Για το clock είναι απαραίτητο να φτιάξουμε ξεχωριστή διαδικασία, καθώς οι διαδικασίες εκτελούνται παράλληλα και πλήρως, δηλαδή δύο διαδικασίες μπορούν να εκτελούνται παράλληλα, αλλά μία διαδικασία εκτελείται σειριακά μέχρι να τελειώσει. Εμείς θέλουμε το ρολόι να λειτουργεί συνεχώς όσο εμείς εφαρμόζουμε διάφορες τιμές για τις εισόδους του uut και για αυτό φτιάχνουμε για το ρολόι ένα ξεχωριστό process. Τέλος, εφαρμόζουμε τα διάφορα σενάρια που θέλουμε να ελέγξουμε και πραγματοποιούμε ένα wait ίσο με μία περίοδο για να δούμε τα αποτελέσματα. Εμείς κατά σειρά εκτελούμε τους ακόλουθους ελέγχους:

1. Αρχικοποιούμε την είσοδο στην τιμή 1010 ελέγχοντας την παράλληλη φόρτωση του shift reg
2. Ελέγχουμε την ορθή λειτουργία του reset
3. Εκτελούμε πάλι παράλληλη φόρτωση τώρα το 0101
4. Πραγματοποιούμε right shift ελέγχοντας με assert ότι τελικά so=1
5. Πραγματοποιούμε left shift ελέγχοντας με assert ότι τελικά so=0

6. Ελέγχουμε ότι αν το enable είναι 0 η κατάσταση του register δεν αλλάζει, σε περίπτωση που αν το en=1 θα πραγματοποιούνταν κάποια ολίσθηση.

Πραγματοποιώντας αυτούς τους ελέγχους και βλέποντας τα αποτελέσματα της προσομοίωσης μπορούμε να βεβαιωθούμε ότι η υλοποίηση μας για τον καταχωρητή ολίσθησης είναι ορθή. Στην συνέχεια παρουσιάζουμε τα αποτελέσματα της προσομοίωσης:



Από τα αποτελέσματα της προσομοίωσης παραπάνω βλέπουμε ότι ο shift register λειτουργεί ορθά σύμφωνα με τα testcases που περιγράψαμε παραπάνω και για αυτό συμπεραίνουμε ότι η υλοποίηση του είναι ορθή

Ζήτημα τρίτο
Μετρητής 3 bit με είσοδο ενεργοποίησης και κρατούμενο εξόδου

Στην άσκηση αυτή μας δίνεται η υλοποίηση ενός μετρητή 3 bit σε VHDL και μας ζητείται να κάνουμε δύο τροποποιήσεις και να ελέγξουμε την λειτουργία τους καθώς και την RRTL υλοποίηση τους.

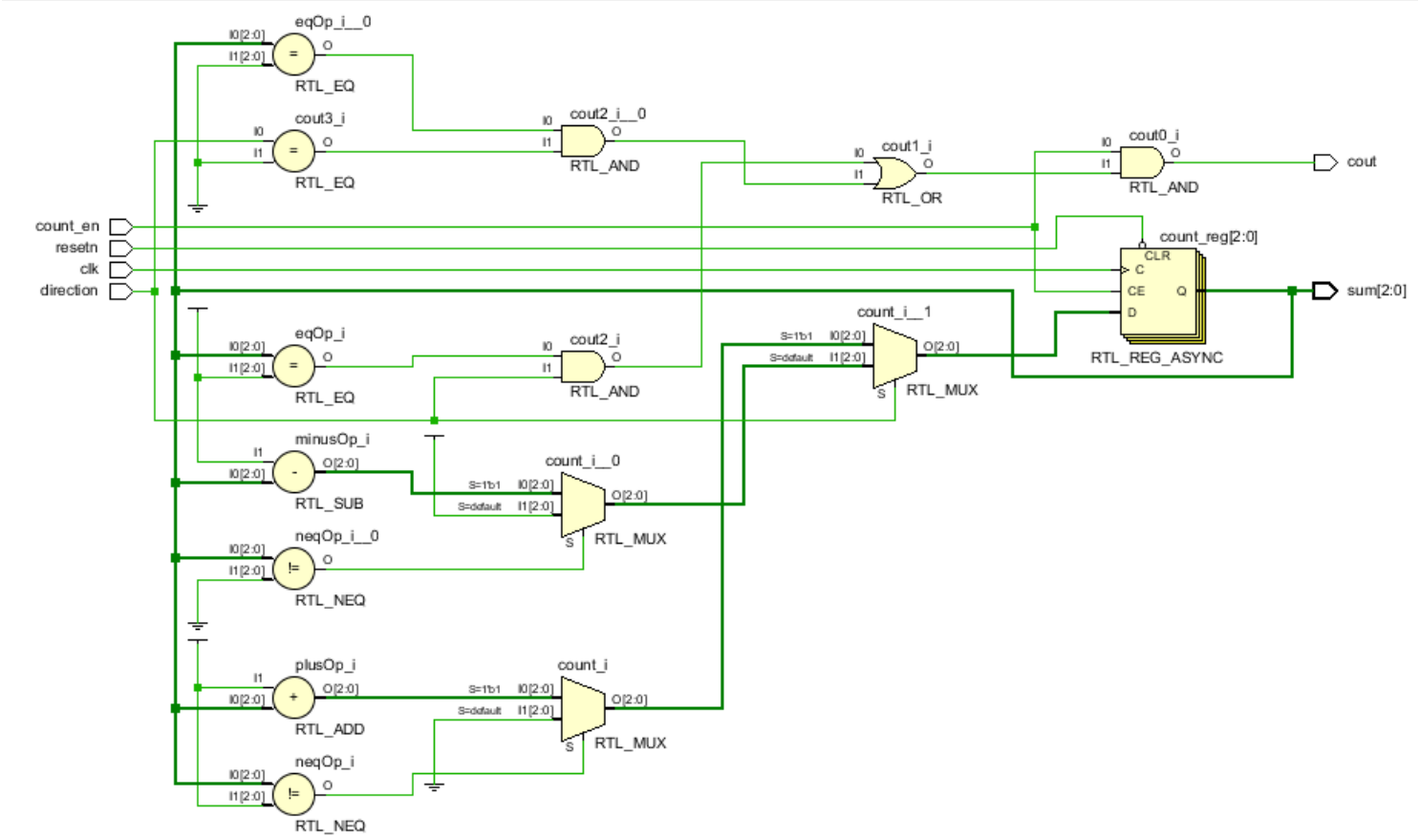
1) Μετρητής up/down τριών bit:

Στο μέρος αυτό μας ζητείται να τροποποιήσουμε τον μετρητή που μας δίνεται προσθέτοντας την δυνατότητα για μέτρηση up/down. Αρχικά παραθέτουμε τον κώδικα σε VHDL που αφορά την υλοποίηση του προαναφερθέντος κυκλώματος:

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

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

architecture rtl_limit of count3 is
    signal count : std_logic_vector(2 downto 0);
begin
    process(clk, resetn)
        begin
        if resetn='0' then
        -- Ασύγχρονος μηδενισμός
            count <= (others=>'0');
            elsif clk'event and clk='1' then
            if count_en = '1' then
            -- Μέτρηση μόνο αν count_en='1'
                if direction='1' then
                    if count/=7 then
                    -- Αυξάνουμε το μετρητή μόνο αν
                    -- δεν είναι 7
                        count <= count+1;
                    else
                    -- Αλλιώς τον μηδενίζουμε
                    count<=(others=>'0');
                    end if;
                else
                    if count/=0 then
                    -- Αυξάνουμε το μετρητή μόνο αν
                    -- δεν είναι 7
                        count <= count-1;
                    else
                    -- Αλλιώς τον μηδενίζουμε
                    count<=(others=>'1');
                    end if;
                end if;
            end if;
        end if;
    end process;
    sum<= count;
    cout <= '1' when count=7 and count_en='1' else '0';
end rtl_limit;
```

Αρχικά, στον παραπάνω κώδικα(αφού προσθέσουμε τις απαραίτητες βιβλιοθήκες) ορίζουμε το entity όπως ακριβώς μας δόθηκε μόνο που τώρα προσθέτουμε μία ακόμα είσοδο που ονομάζουμε direction και θα καθορίζει την φορά της μέτρησης. Αν το direction είναι ίσο με 1 τότε αυξάνουμε τον μετρητή κατά 1 ειδάλλως τον μειώνουμε κατά 1. Στην συνέχεια ορίζουμε την αρχιτεκτονική του μετρητή μας, ορίζοντας πρώτα ένα σήμα count το οποίο χρησιμεύει ώστε να μπορεί να πραγματοποιηθεί πρόσθεση σε αυτό και ανάθεση τιμής σε αυτό (θα μπορούσαμε να κάνουμε και το sum inout σήμα, αλλά στην εκφώνηση ο κώδικας μας δόθηκε έτσι οπότε επιλέξαμε να μην τροποποιήσουμε σημεία που δεν ήταν απαραίτητο). Ακολούθως ορίζουμε το process μας και στο netlist προφανώς βάζουμε το clk και το resetn. Πρώτα στο process ελέγχουμε αν το resetn είναι 0, καθώς σε αυτή την περίπτωση η έξοδος μηδενίζεται, ειδάλλως σε κάθε ακμή του ρολογιού πραγματοποιούμε μία αλλαγή του count, εφόσον το enable είναι 1. Τον έλεγχο για το reset τον πραγματοποιούμε με το πρώτο if , ενώ με το ακόλουθο elsif πραγματοποιούμε τον έλεγχο για την θετική ακμή του ρολογιού. Εντός του elsif αυτού ελέγχουμε εάν το enable είναι ενεργό. Αν είναι ελέγχουμε την τιμή του direction για να πραγματοποιήσουμε την αντίστοιχη αύξηση ή μείωση του count. Στην περίπτωση της αύξησης ελέγχουμε την τιμή του count , ώστε αν είναι 7 να επανέλθει στην τιμή 0. Αν πραγματοποιούμε μείωση του μετρητή ελέγχουμε αν αυτός είναι 0 ώστε να τον επαναφέρουμε στην τιμή 7. Σε αυτό το σημείο βγαίνουμε από το process και σε κάθε κύκλο αναθέτουμε την τρέχουσα τιμή του count στην έξοδο του sum και ελέγχουμε εάν το count είναι 7 ,το count_en=1 και το direction ίσο με 1 ώστε να θέσουμε το κρατούμενο εξόδου ίσο με 1 και αντίστοιχα ελέγχουμε εάν το count=0, το count_en=1 και το direction=0 για να θέσουμε το κρατούμενο εξόδου ίσο με 1. Σε κάθε άλλη περίπτωση θέτουμε το κρατούμενο εξόδου ίσο με 0.

Στην συνέχεια παρουσιάζουμε το σχηματικό σε RTL που μας έδωσε το Vivado με βάση την παραπάνω VHDL:



Το παραπάνω διάγραμμα αποτελεί την υλοποίηση σε RT επίπεδο από το Vivado με βάση τον κώδικα μας. Προφανώς έχει 3 flip-flop(καταχωρητής 3 bit) όπου διατηρείται η τρέχουσα τιμή του count και συνεπώς από την έξοδο τους λαμβάνεται το sum. Το ρολόι αντιστοιχεί στην είσοδο του ρολογιού το reset στο αντίστοιχο reset και το enable στο αντίστοιχο CE. Αναφορικά με του cout. Οι δύο συγκριτές στην κορυφή συγκρίνουν την τιμή του direction, αν είναι ίση με το 0 και τη τιμή του count αν είναι ίση με το 0. Στην συνέχεια πραγματοποιείται μία σύζευξη AND αυτών των δύο τιμών (όπως ακριβώς περιγράψαμε εντός του when στην τελευταία γραμμή του κώδικα μας). Αντίστοιχα έχουμε έναν συγκριτή που συγκρίνει το sum με το 7 και η έξοδος του περνά σε μία and της οποίας η άλλης είσοδος είναι το direction (έτσι ελέγχουμε εάν το count είναι 7 και το direction 1 όπως αναφέραμε στον κώδικα μας). Τις εξόδους των δύο παραπάνω and τις κάνουμε εισόδους σε μία or, όπως και στον κώδικα μας είχαμε τον τελεστή or εντός του when. Η έξοδος αυτή περνάει από μία πύλη AND της οποίας η άλλη είσοδος είναι το enable και με βάση τον ορισμό του κρατούμενου εξόδου που δόθηκε(φαίνεται εντός της συνθήκης του when) , όταν όλες οι προϋποθέσεις καλύπτονται το cout=1. Στην συνέχεια εξηγούμε το υπόλοιπο κύκλωμα που υλοποιεί την είσοδο των D flip-flop που αλλάζουν το count. Το κυκλάκι με το – αντιστοιχεί σε ένα λογικό κύκλωμα που υλοποιεί την αφαίρεση του 1 από το count, ενώ με το ακριβώς από κάτω != ελέγχουμε εάν το count είναι ίσο με το 0. Από την σύγκριση αυτή επιλέγουμε αν η επόμενη τιμή του count σε περίπτωση αφαίρεσης θα είναι το 7 ή το count-1(καθώς όπως φαίνεται από τον πολυπλέκτη αν count!=0 τότε επιλέγεται μέσω του πολυπλέκτη η πάνω είσοδος και συνεπώς η νέα τιμή του count θα είναι count-1, αν έχει επιλεγεί αφαίρεση και enable=1). Ακριβώς με την ίδια λογική έχουμε έναν αθροιστή, που προσθέτει 1 στο count και έναν συγκριτή του count με το 7 που καθορίζει μέσω ενός πολυπλέκτη, εάν η η επόμενη τιμή του count θα είναι count+1 ή 0, ανάλογα με το αν το count=7 ή όχι. Τέλος, οι δύο έξοδοι των πολυπλέκτων αυτών καθορίζουν την τιμή που θα πρέπει να πάρει το count αν πραγματοποιηθεί μία αφαίρεση ή μία πρόσθεση αντίστοιχα. Για αυτό γίνονται είσοδοι ενός ακόμα πολυπλέκτη που έχει σαν σήμα επιλογής το direction και καθορίζεται εάν η είσοδος των D flip-flop που ορίζουν το count θα προέρχονται από την πρόσθεση ή την αφαίρεση του 1 από την τρέχουσα τιμή του count. Από την παραπάνω ανάλυση μπορούμε εύκολα να συμπεράνουμε ότι το κύκλωμα μας λειτουργεί ορθά και είναι σχετικά αποδοτικά υλοποιημένο.

Στην συνέχεια θα παρουσιάσουμε την υλοποίηση του testbench, το οποίο φτιάξαμε για να ελέγξουμε την ορθή λειτουργία του μετρητή μας. Παραθέτουμε ακολούθως τον κώδικα για το Testbench:

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

entity tb_count3 is
end tb_count3;

architecture Behavioral of tb_count3 is
    constant CLK_PERIOD : time := 5 ns; -- Clock period

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

```
    signal tb_count_en,tb_direction,tb_cout,tb_clk,tb_resetn :  std_logic;
    signal tb_sum :  std_logic_vector(2 downto 0);
begin
    uut: count3
        port map(
            clk => tb_clk,
            resetn => tb_resetn,
            count_en => tb_count_en,
            direction => tb_direction,
            cout => tb_cout,
            sum => tb_sum
        );
-- Clock process
    clk_process : process
    begin
        while true loop
            tb_clk <= '0';
            wait for CLK_PERIOD / 2;
            tb_clk <= '1';
            wait for CLK_PERIOD / 2;
        end loop;
    end process;
  -- Stimulus process
       stim_process : process
         begin
         -- initialize input
         tb_resetn <= '0';
         tb_count_en <= '0';
         tb_direction <= '0';
         wait for CLK_PERIOD;
         -- count up for 10 cycles
         tb_resetn <= '1';
         tb_count_en <= '1';
         tb_direction <= '1';
         wait for CLK_PERIOD*10;
         -- count down for 10 cycles
         tb_resetn <= '1';
         tb_count_en <= '1';
         tb_direction <= '0';
         wait for CLK_PERIOD*10;
         -- disable enable for 1 cycle
         tb_resetn <= '1';
         tb_count_en <= '0';
         tb_direction <= '0';
         wait;
       end process;
end Behavioral;
```

Η λογική της υλοποίησης είναι ακριβώς ίδια με προηγουμένως. Αρχικά ορίζουμε το entity του testbench και στην συνέχεια την αρχιτεκτονική του. Εντός της αρχιτεκτονικής ορίζουμε την περίοδο του ρολογιού καθώς ,το component που θα εξετάσουμε και τα σήματα του testbench που θα αντιστοιχίσουμε με τις εισόδους και εξόδους που θα εξετάσουμε. Για να δημιουργήσουμε το Instance αυτής της αντιστοιχίας , ορίζουμε το uut(unit under test) όπως φαίνεται στον κώδικά μας. Εν συνεχεία, ορίζουμε ένα process για το ρολόι, όπως κάναμε και στην άσκηση 2, ώστε το ρολόι να τρέχει με την δική του περιοδικότητα σε όλη την διάρκεια της προσομοίωσης. Τέλος ορίζουμε τα test cases που θέλουμε να τρέξουμε. Αυτά όπως φαίνονται παραπάνω είναι τα ακόλουθα:

1. Έλεγχος reset , παράλληλα αρχικοποιούμε και την είσοδο και έξοδο.
2. Πραγματοποιούμε μία μέτρηση προς τα άνω για 10 κύκλους, για αυτό στο wait έχουμε την τιμή 10*CLK_PERIOD
3. Πραγματοποιούμε μία μέτρηση προς τα κάτω για 10 κύκλους
4. Ελέγχουμε ότι για enable=0 και reset=1 δεν αλλάζει η κατάσταση της εξόδου

Παρακάτω φαίνονται τα αποτελέσματα της εκτέλεσης μίας προσομοίωσης για αυτό το testbench, όπου μπορούμε να επιβεβαιώσουμε την ορθή λειτουργία της περιγραφής μας.

Παραπάνω παρατηρούμε ότι η έξοδος, τόσο το sum όσο και το cout παίρνουν την αναμενόμενη τιμή για κάθε ένα από τα τεστ που τρέξαμε(βλέπουμε ορθά τόσο το κρατούμενο εξόδου όσο και την αλλαγή της μέτρησης από 0 σε 7 στην κάτω μέτρηση και από 7 σε 1 στην άνω μέτρηση). Συμπεραίνουμε λοιπόν ότι η περιγραφή μας είναι ορθή για το παρόν πρόβλημα.

2) Μετρητής με όριο μέτρησης:

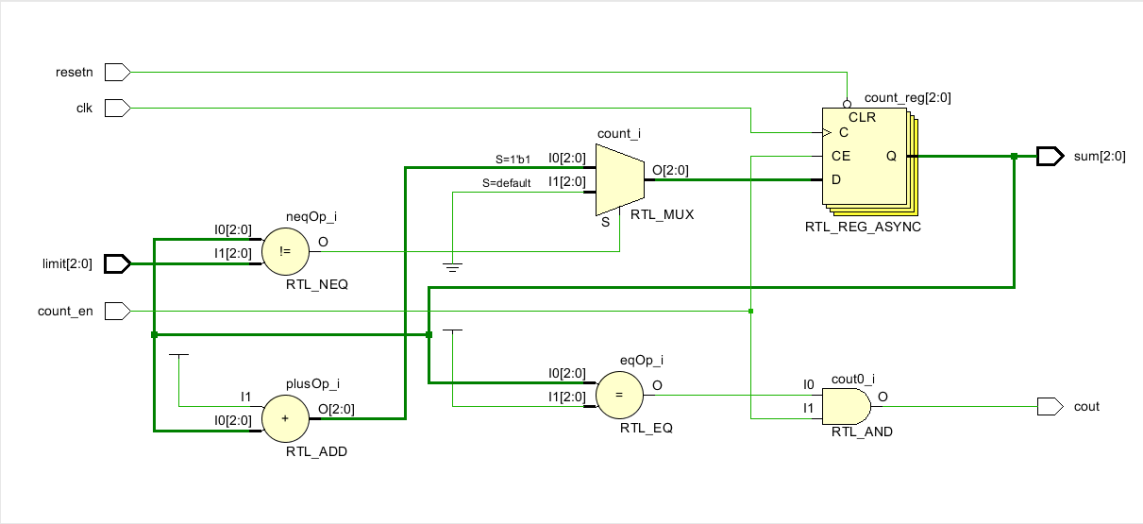
Στο ερώτημα αυτό θα χρησιμοποιήσουμε τον κώδικα που μας δόθηκε στην εκφώνηση και θα πραγματοποιήσουμε μία μικρή τροποποίηση προκειμένου να έχει μεταβλητό όριο μέτρησης. Αυτό όπως θα δούμε στην συνέχεια είναι πάρα πολύ απλό. Παραθέτουμε για αρχή τον κώδικα μας σε VHDL:

```
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;
limit : in std_logic_vector(2 downto 0);
sum : out std_logic_vector(2 downto 0);
cout : out std_logic);
end;


architecture rtl_limit of count3 is
    signal count : std_logic_vector(2 downto 0);
    begin
    process(clk, resetn)
        begin
        if resetn='0' then
        -- Ασύγχρονος μηδενισμός
            count <= (others=>'0');
            elsif clk'event and clk='1' then
            if count_en = '1' then
            -- Μέτρηση μόνο αν count_en=’1’
                if count/=limit then
                -- Αυξάνουμε το μετρητή μόνο αν
                -- δεν είναι 7
                     count <= count+1;
                else
                -- Αλλιώς τον μηδενίζουμε
                     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 rtl_limit ;
```

Η δομή του κώδικα είναι ίδια με προηγουμένως, δηλαδή αρχικά ορίζουμε το entity με τις εισόδους και τις εξόδους του και στην συνέχεια την αρχιτεκτονική του. Ο κώδικας είναι σχεδόν ίδιος με αυτόν που μας δίνεται στην εκφώνηση και αναλύσαμε προηγουμένως με μία μικρή τροποποίηση. Αρχικά ορίσαμε ένα νέο logic vector σαν είσοδο 3 bit το οποίο το ονομάσαμε limit, αυτή η είσοδος θα είναι το όριο της μέτρησης μας. Για να δείξουμε ότι αυτό θα είναι το όριο της μέτρησης μας κάτω από το «if count_en=1 then», αλλάξαμε το if μας προκειμένου πλέον η μέτρηση να μην σταματάει στο 7 αλλά στο όριο που θα έχει θέσει ο χρήστης. Οπότε στο if αυτό πλέον αντί να ελέγχουμε ότι το count!=7 ελέγχουμε ότι το count!=limit. Όταν το count=limit , τότε η μέτρηση ξεκινάει πάλι από το 0. Τέλος να σημειώσουμε ότι ο έλεγχος για το κρατούμενο εξόδου δεν αλλάζει και αυτό γιατί σε έναν 3 bit αθροιστή που προσθέτουμε πάντα το 1 σε κάποιον 3 Bit αριθμό overflow θα έχουμε μόνο όταν προσθέτουμε το 1 στο 7. Συνεπώς βλέπουμε πως με μία πολύ μικρή αλλαγή στον κώδικα μας επιτρέψαμε την ύπαρξη μεταβλητού ορίου στην μέτρηση.

Στην συνέχεια παρουσιάζουμε το σχεδιάγραμμα RTL που μας έδωσε το Vivado για τον κώδικα που αναλύσαμε παραπάνω:



Βλέπουμε ότι το παραπάνω διάγραμμα είναι πολύ απλό και σε λογική ακολουθεί αυτό του πρώτου μέρους της άσκησης. Προφανώς έχουμε έναν καταχωρητή τριών bit , που αποτελείται από 3 D flip flop για την αποθήκευση της τρέχουσας τιμής του μετρητή. Από αυτό τον καταχωρητή λαμβάνουμε και την έξοδο μας. Εννοείται ότι το clk αντιστοιχεί στο C των flip flop, το resetn στο CLR των flip flop και το count_en στο CE των flip flop. Αναφορικά με το κρατούμενο εξόδου, έχουμε έναν συγκριτή του count με το 7, η έξοδος του οποίου γίνεται είσοδος σε μία πύλη and της οποίας η άλλη είσοδος είναι το enable. Αυτό διασφαλίζει ότι το cout θα είναι 1 μόνο εάν το count=7 και το enable=1. Επιπλέον έχουμε έναν αθροιστή που προσθέτει πάντα το 1 στην τρέχουσα τιμή του count. Το αποτέλεσμα της άθροισης αυτής πηγαίνει στην είσοδο ενός πολυπλέκτη, του οποίου η άλλη είσοδος είναι το 0. Η επιλογή ανάμεσα στις δύο αυτές εισόδους για τον καθορισμό της εξόδου του πολυπλέκτη γίνεται από την έξοδο ενός συγκριτή, που συγκρίνει το limit με το τρέχω count. Με αυτό τον τρόπο διασφαλίζεται το σωστό όριο στην μέτρηση και η ορθή είσοδος για τα flip flop που καθορίζουν το count. Βλέπουμε ότι το κύκλωμα μας είναι πολύ απλό και με μία εποπτική ανάλυση μόνο καταφέραμε να αποδείξουμε την ορθότητά του.

Τέλος, όπως σε κάθε ερώτημα θα υλοποιήσουμε και ένα testbench προκειμένου να αποδείξουμε και πειραματικά την ορθή λειτουργία της περιγραφής που κάναμε. Ακολούθως παραθέτουμε το testbench για το παραπάνω component:

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

entity tb_count3 is
end tb_count3;

architecture Behavioral of tb_count3 is
    constant CLK_PERIOD : time := 5 ns; -- Clock period

    component count3 is
    port( clk,
    resetn,
    count_en : in std_logic;
```

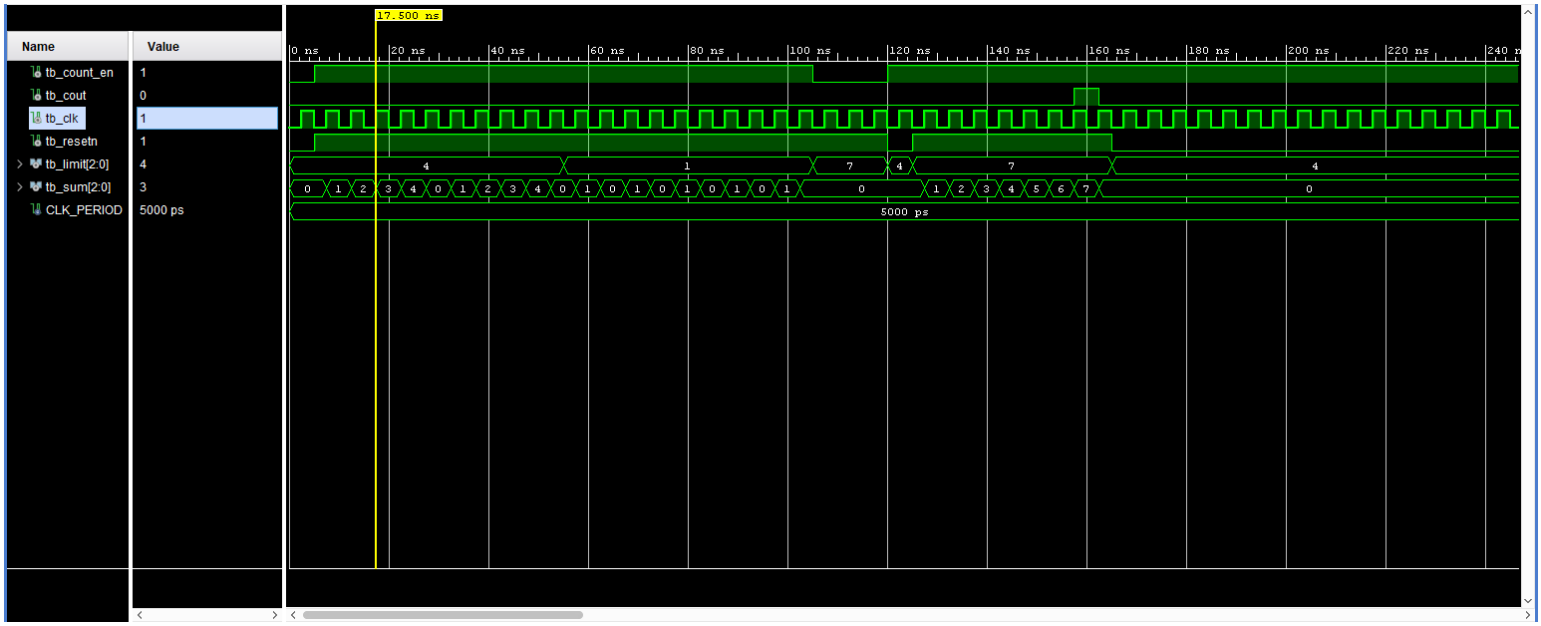
```
    limit : in std_logic_vector(2 downto 0);
    sum : out std_logic_vector(2 downto 0);
    cout : out std_logic);
    end component;
    signal tb_count_en,tb_cout,tb_clk,tb_resetn :  std_logic;
    signal tb_limit :  std_logic_vector(2 downto 0);
    signal tb_sum :  std_logic_vector(2 downto 0);
begin
    uut: count3
        port map(
            clk => tb_clk,
            resetn => tb_resetn,
            count_en => tb_count_en,
            limit => tb_limit,
            cout => tb_cout,
            sum => tb_sum
        );
-- Clock process
    clk_process : process
    begin
        while true loop
            tb_clk <= '0';
            wait for CLK_PERIOD / 2;
            tb_clk <= '1';
            wait for CLK_PERIOD / 2;
        end loop;
    end process;
  -- Stimulus process
      stim_process : process
        begin
        -- initialize input
        tb_resetn <= '0';
        tb_count_en <= '0';
        tb_limit <= "100";
        wait for CLK_PERIOD;
        -- count up for 10 cycles
        tb_resetn <= '1';
        tb_count_en <= '1';
        tb_limit <= "100"; -- limit 4
        wait for CLK_PERIOD*10;
        -- count up for 10 cycles
        tb_resetn <= '1';
        tb_count_en <= '1';
        tb_limit <= "001"; -- limit 1
        wait for CLK_PERIOD*10;
        -- disable enable for 3 cycles
        tb_resetn <= '1';
        tb_count_en <= '0';
```

```
        tb_limit <= "111"; -- limit 7
        wait for CLK_PERIOD*3;
        -- initialize input
        tb_resetn <= '0';
        tb_count_en <= '1';
        tb_limit <= "100"; -- reset
        wait for CLK_PERIOD;
        -- disable enable for 1 cycle
        tb_resetn <= '1';
        tb_count_en <= '1';
        tb_limit <= "111"; -- limit 7
        wait for CLK_PERIOD*8;
        -- initialize input
        tb_resetn <= '0';
        tb_count_en <= '1';
        tb_limit <= "100"; -- reset
        wait;
      end process;
end Behavioral;
```

Ομοίως με όλα τα testbenches που παρουσιάσαμε ξεκινάμε ορίζοντας ένα κενό από εισόδους και εξόδους entity, που θα είναι το testbench μας και θα περικλείει το component του οποίου την λειτουργία θέλουμε να ελέγξουμε. Στην συνέχεια ορίζουμε την αρχιτεκτονική ορίζοντας την περίοδο του ρολογιού, το component που θα ελέγξουμε και τα σήματα του Tb τα οποία θα αντιστοιχίσουμε στις εισόδους και στις εξόδους του component . Ακολούθως δημιουργούμε ένα instance που το ονομάζουμε uut(unit under test) και αντιστοιχίζουμε τα σήματα του tb με τις αντίστοιχες εισόδους και εξόδους του component. Εν συνεχεία ορίζουμε ένα process για το ρολόι ,ώστε αυτό να λειτουργεί ανεξάρτητα από κάθε άλλο process περιοδικά για όλη την περίοδο που θα ελέγξουμε το κύκλωμα μας. Τέλος , γράφουμε τα test case μας, τα οποία είναι τα ακόλουθα:

1. Αρχικοποιούμε την είσοδο και την έξοδο πραγματοποιώντας ένα reset. Έτσι ελέγχουμε και την λειτουργία αυτή
2. Πραγματοποιούμε με όριο το 4 μέτρηση προς τα άνω για 10 κύκλους
3. Πραγματοποιούμε με όριο το 1 μέτρηση προς τα άνω για 10 κύκλους
4. Απενεργοποιούμε το enable για 3 κύκλους
5. Επαναφέρουμε την είσοδο με το reset
6. Μετράμε προς τα πάνω με όριο το 7 για 8 κύκλους
7. Επαναφέρουμε την είσοδο με το reset

Παρουσιάζουμε τα αποτελέσματα της προσομοίωσης για τα παραπάνω test cases:



Η ορθότητα της περιγραφής μας επιβεβαιώνεται και από τα αποτελέσματα της προσομοίωσης που φαίνονται παραπάνω, τα οποία συμβαδίζουν με τα αναμενόμενα που αναφέραμε παραπάνω. Συνοψίζοντας, βλέπουμε ότι για την μέτρηση με όριο απαιτούνται ελάχιστες τροποποιήσεις στον αρχικό κώδικα. Επίσης το RTL design που μας δίνει το Vivado σε αυτή την περίπτωση είναι πολύ αποδοτικό, από άποψης υλικού.

Συμπερασματικά, και για τα δύο ζητούμενα υλοποιήσαμε τις αντίστοιχες περιγραφές σε VHDL και επιβεβαιώσαμε την λειτουργία τους τόσο με τα RTL όσο και με το testbench. Αναφορικά με τα RTL το Vivado μας έδωσε μία ιδιαιτέρως αποδοτική υλοποίηση από πλευράς υλικού για τον μετρητή με όριο, η οποία είναι πολύ κοντά σε αυτή της εκφώνησης(εξαρτάται από την υλοποίηση του αθροιστή και των συγκριτών), ενώ για την περίπτωση του μετρητή με δυνατότητα επιλογής άνω ή κάτω μέτρησης το υλικό που προστίθεται σε σχέση με το σχηματικό του απλού μετρητή που δίνεται στην εκφώνηση είναι αρκετό λόγω των νέων ελέγχων που απαιτούνται (με τα αντίστοιχα if και την συνθήκη στο when) για την κατεύθυνσης της μέτρησης. Αξίζει βέβαια να σημειώσουμε ότι το Vivado δεν μας έδωσε την υλοποίηση της εκφώνησης για τον απλό μετρητή, εικάζουμε ότι το Vivado χρησιμοποιεί πιο συχνά πολυπλέκτες στα if και αθροιστές με ένα υψηλότερο επίπεδο αφαίρεσης (χρησιμοποιεί ένα κύκλο με ένα + για να τους συμβολίσει αντί για όλο το λογικό κύκλωμα) σε σχέση με το λογισμικό που χρησιμοποιήθηκε για την δημιουργία του RTL της εκφώνησης της άσκησης.