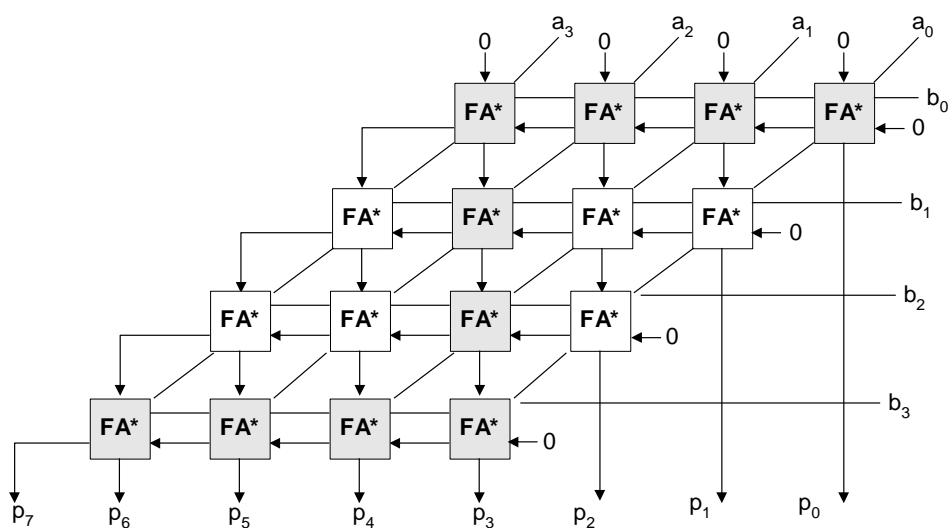


Κ.Ζ. ΠΕΚΜΕΣΤΖΗ
Καθηγήτριας Ε.Μ.Π.

ΨΗΦΙΑΚΑ ΣΥΣΤΗΜΑΤΑ VLSI

ΕΡΓΑΣΤΗΡΙΑΚΕΣ ΑΣΚΗΣΕΙΣ



ΑΘΗΝΑ 2014

Πρόλογος

Συμβολή σε ορισμένα σημεία του περιεχομένου είχαν οι διδάκτορες Γ. Οικονομάκος και Ισ. Σίδερης καθώς και οι μεταπτυχιακοί σπουδαστές Ι. Σιφναίος, Ε. Χανιωτάκης και Κ. Ασφής τους οποίους ευχαριστώ από τη θέση αυτή.

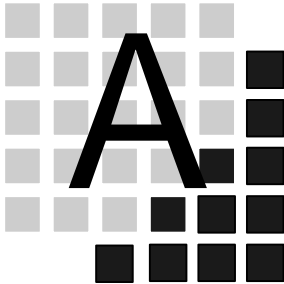
Κ.ΠΕΚΜΕΣΤΖΗ

ΠΕΡΙΕΧΟΜΕΝΑ

ΑΘΗΝΑ 2014.....	1
ΠΕΡΙΕΧΟΜΕΝΑ	2
ΤΟ ΠΕΡΙΒΑΛΛΟΝ ΠΡΟΣΟΜΟΙΩΣΗΣ ACTIVE HDL.....	7
A.1 Δημιουργία νέου design.....	7
A.1.1 ΟΝΟΜΑΣΙΑ ΤΟΥ DESIGN	7
A.1.2 ΕΠΙΛΟΓΗ ΤΩΝ ΠΕΡΙΕΧΟΜΕΝΩΝ ΜΙΑΣ ΣΧΕΔΙΑΣΗΣ	8
A.1.3 ΔΗΜΙΟΥΡΓΙΑ ΑΡΧΕΙΩΝ / ΟΝΤΟΤΗΤΩΝ.	9
A.1.4 ΔΗΜΙΟΥΡΓΙΑ ΕΙΣΟΔΩΝ ΚΑΙ ΕΞΟΔΩΝ ΜΙΑΣ ΟΝΤΟΤΗΤΑΣ.....	11
A.1.5 ΑΠΟΔΟΧΗ ΤΩΝ ΕΠΙΛΟΓΩΝ ΚΑΙ ΔΗΜΙΟΥΡΓΙΑ ΤΗΣ ΣΧΕΔΙΑΣΗΣ	13
A.2 Το περιβάλλον εργασίας.....	13
A.2.1 ΕΙΣΑΓΩΓΗ ΚΩΔΙΚΑ	14
A.2.2 COMPILATION	15
A.2.3 ΠΡΟΣΟΜΟΙΩΣΗ.....	17
A.3 Άσκηση 1.....	20
Θέμα A.1: Δυναδικός κωδικοποιητής προτεραιότητας 4 σε 2	20
Θέμα A.2: Δυναδικός αποκωδικοποιητής 3 σε 8.....	21
Θέμα A.3: Αποπλέκτης 1 σε 4 και πολυπλέκτης 4 σε 1	22
ΤΟ ΠΕΡΙΒΑΛΛΟΝ ΣΥΝΘΕΣΗΣ LEONARDO SPECTRUM.....	25
B.1 Σύνθεση κυκλωμάτων με το Leonardo Spectrum.....	25
B.1.1 ΧΡΗΣΙΜΟΤΗΤΑ ΤΟΥ ΠΡΟΓΡΑΜΜΑΤΟΣ	25
B.1.2 ΠΕΡΙΓΡΑΦΗ ΤΟΥ LEONARDO SPECTRUM	25
Θέμα B.1: Απλός καταχωρητής 1 bit (D Flip-Flop).....	31
Θέμα B.2: Καταχωρητής ολίσθησης των 4 bits με παράλληλη φόρτωση	32
Θέμα B.3: Μετρητής 3 bit με είσοδο ενεργοποίησης και κρατούμενο εξόδου	34
ΕΛΕΓΧΟΣ ΜΟΝΑΔΩΝ – TEST BENCH	41
C.1 Ορισμός του test bench.....	41
C.2 Αυτόματη δημιουργία test bench με το Active HDL.....	43
C.3 Χαρακτηριστικοί τύποι test bench.....	49
C.3.1 TEST BENCH ΜΕ ΟΡΙΣΜΟ ΕΙΣΟΔΟΥ	49
C.3.2 TEST BENCH ΜΕ ΑΡΧΕΙΟ ΕΙΣΟΔΟΥ Η/ΚΑΙ ΕΞΟΔΟΥ	52
C.3.3 TEST BENCH ΠΟΥ ΔΗΜΙΟΥΡΓΕΙ ΕΙΣΟΔΟΥΣ ΚΑΙ ΕΞΟΔΟΥΣ.....	56
C.4 Άσκηση 3	60
ΣΧΕΔΙΑΣΗ ΚΑΙ ΕΞΟΜΟΙΩΣΗ ΑΛΓΟΡΙΘΜΙΚΩΝ ΜΗΧΑΝΩΝ ΚΑΤΑΣΤΑΣΕΩΝ (ΑΜΚ) ..	60
Θέμα C.1: Κύκλωμα διαιτησίας διαδρόμων.....	60
Θέμα C.2: Κύκλωμα υπολογισμού Μέγιστου Κοινού Διαιρέτη (ΜΚΔ) δύο ακεραίων.....	67
ΑΡΙΘΜΗΤΙΚΑ ΚΥΚΛΩΜΑΤΑ - ΑΘΡΟΙΣΤΕΣ.....	77
D.1 Άσκηση 4 - Αθροιστές και αφαιρέτες	77

Θέμα D.1: Ημιαθροιστής	77
Θέμα D.2: Πλήρης αθροιστής	79
Θέμα D.3: Παράλληλος αθροιστής με διάδοση κρατουμένου	81
Θέμα D.4: Παράλληλος συγκριτής	84
ΑΡΙΘΜΗΤΙΚΑ ΚΥΚΛΩΜΑΤΑ - ΠΟΛΛΑΠΛΑΣΙΑΣΤΕΣ	87
E.1 Άσκηση 5 - Πολλαπλασιαστές	87
Θέμα E.1: Πολλαπλασιαστές βασισμένοι σε carry-propagate αθροιστές	87
Θέμα E.2 : Πολλαπλασιαστές βασισμένοι σε carry-save αθροιστές	93
ΑΡΙΘΜΗΤΙΚΑ ΚΥΚΛΩΜΑΤΑ – ΣΕΙΡΙΑΚΑ ΔΕΔΟΜΕΝΑ	95
ΑΣΚΗΣΗ 6 - ΕΠΕΞΕΡΓΑΣΙΑ ΣΕΙΡΙΑΚΩΝ ΔΕΔΟΜΕΝΩΝ	95
Θέμα F.1: Μετατροπή σειριακής εισόδου σε παράλληλη και αντίστροφα	95
Υλοποίηση Εφαρμογών σε XILINX FPGAs	107
ΑΣΚΗΣΗ 7 – ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΔΙΑΤΑΞΕΩΝ FPGA.....	107
Περιγραφή προγραμματιζόμενων διατάξεων FPGA.....	107
XILINX FPGAs.....	107
ΠΕΡΙΓΡΑΦΗ ΑΝΑΠΤΥΞΙΑΚΗΣ ΠΛΑΚΕΤΑΣ SPARTAN3E STARTER KIT.....	111
ΠΕΡΙΒΑΛΛΟΝ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ XILINX ISE 8.1	112
Δημιουργία project.....	114
ΘΕΜΑΤΑ ΆΣΚΗΣΗΣ	123
Μετρητής.....	123
ΧΕΙΡΙΣΜΟΣ LCD ΟΘΟΝΗΣ ΧΑΡΑΚΤΗΡΩΝ	127
ΧΡΗΣΗ ΜΝΗΜΩΝ BLOCKRAM ΤΟΥ FPGA	150

ΠΑΡΑΡΤΗΜΑ –
ΕΡΓΑΣΤΗΡΙΑΚΕΣ ΑΣΚΗΣΕΙΣ



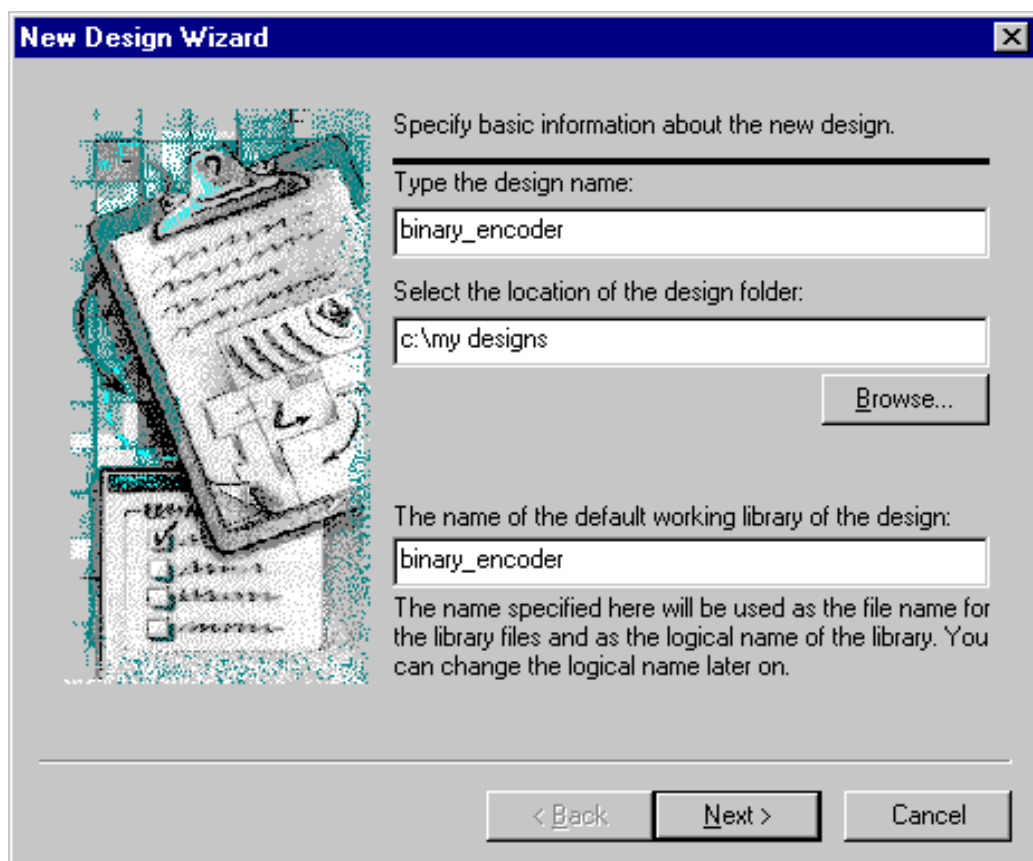
ΤΟ ΠΕΡΙΒΑΛΛΟΝ ΠΡΟΣΟΜΟΙΩΣΗΣ ACTIVE HDL

A.1 Δημιουργία νέου design

Το πρόγραμμα Active HDL χρησιμοποιείται για την προσομοίωση κυκλωμάτων που περιγράφονται με τη βοήθεια της γλώσσας VHDL. Στις παραγράφους που ακολουθούν, θα παρουσιάσουμε την κατασκευή ενός δυαδικού κωδικοποιητή, με σκοπό την καλύτερη κατανόηση της λειτουργίας του εργαλείου. Η κατασκευή ξεκινάει με τη δημιουργία νέας σχεδίασης (design), που γίνεται από τον “New Design Wizard” (Menu: File/New Design).

A.1.1 Ονομασία του design

Στο πρώτο παράθυρο του “New Design Wizard” (σχήμα 1.1) εισάγουμε το όνομα της σχεδίασης (design) και το όνομα της βιβλιοθήκης. Το όνομα της σχεδίασης πρόκειται να χρησιμοποιηθεί ως όνομα του υποκαταλόγου που θα την περιέχει. Ο υποκατάλογος αυτός θα περιέχει όλα τα αρχεία που ανήκουν στη σχεδίαση, γεγονός που μας διευκολύνει στην μεταφορά του σε άλλο υπολογιστή. Εδώ δίνουμε το όνομα `binary_encoder` και προχωρούμε στο επόμενο παράθυρο πατώντας “Next”.



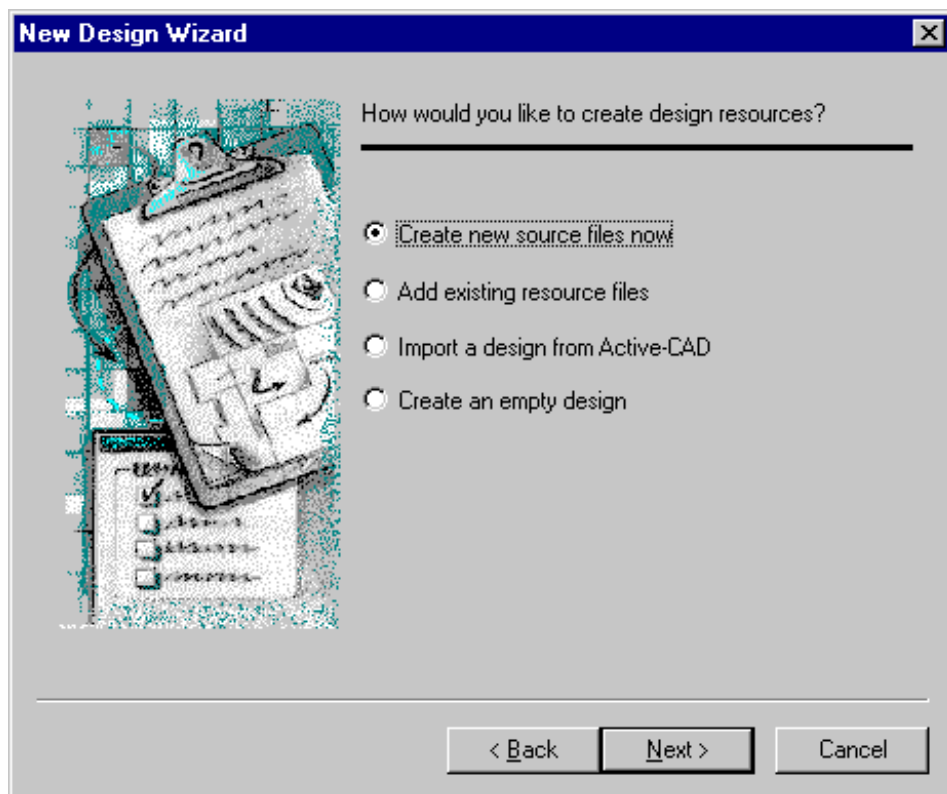
Σχ. 1.1: New Design Wizard – παράθυρο 1

Α.1.2 Επιλογή των περιεχομένων μιας σχεδίασης

Στο επόμενο παράθυρο (σχήμα 1.2), οι δυνατότητες που έχουμε για τα περιεχόμενα της σχεδίασης είναι:

- Δημιουργία νέων αρχείων με τη βοήθεια του wizard
- Προσθήκη έτοιμων αρχείων (η δυνατότητα αυτή υπάρχει και αργότερα)
- Εισαγωγή παλιάς σχεδίασης από το Active – CAD (προηγούμενη έκδοση του προγράμματος)
- Δημιουργία μιας κενής σχεδίασης (empty design).

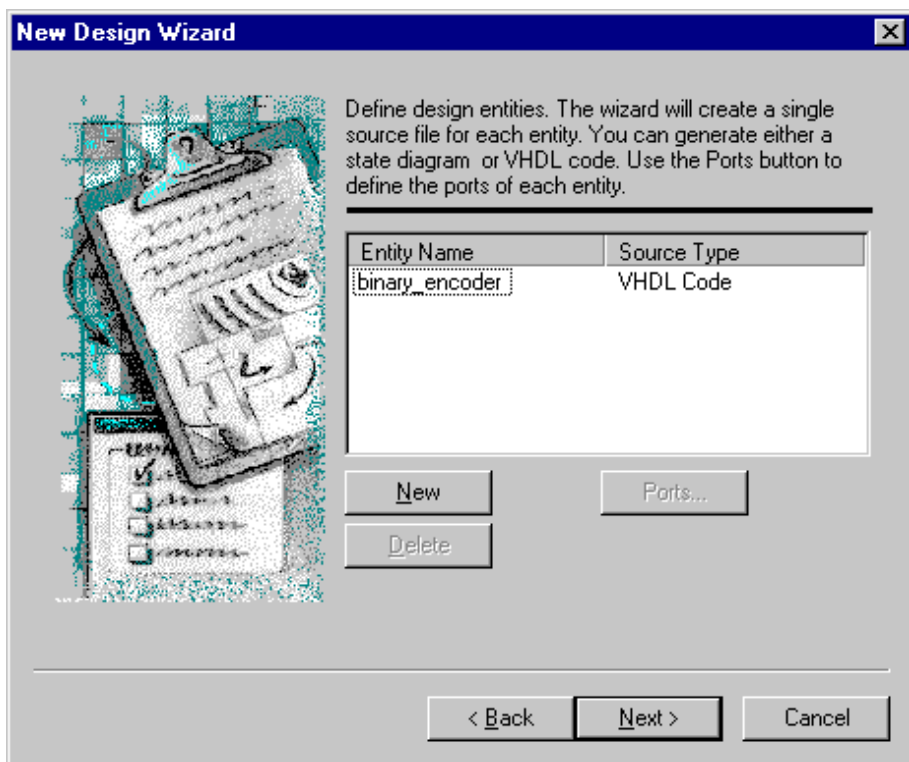
Εδώ επιλέγουμε την πρώτη δυνατότητα για να δούμε τις δυνατότητες του wizard. Προχωρούμε πάλι στο επόμενο παράθυρο με “Next”



Σχ. 1.2: New Design Wizard – παράθυρο 2

A.1.3 Δημιουργία αρχείων / οντοτήτων.

Το τρίτο παράθυρο του New Design Wizard (σχήμα 1.3), μας δίνει τη δυνατότητα να δημιουργήσουμε οντότητες (entities) που θα μας χρειαστούν στη σχεδίαση. Ο wizard γράφει κάθε οντότητα σε ξεχωριστό αρχείο, συνηθισμένη πρακτική που βοηθά στην ευκολότερη διόρθωση και επαναχρησιμοποίηση του κώδικα.



Σχ. 1.3: New Design Wizard – παράθυρο 3

Η δημιουργία νέας οντότητας γίνεται πατώντας το “New” και εισάγοντας το όνομα, που εδώ θα είναι `binary_encoder`. Αμέσως εμφανίζεται το παράθυρο του Ports Wizard, η λειτουργία του οποίου εξηγείται στην επόμενη παράγραφο. Ο τύπος του αρχείου μπορεί να επιλεγεί από τη δεύτερη στήλη και οι δυνατότητες που έχουμε είναι:

- VHDL code (η επιλογή μας)
- State diagram (ειδικά για περιγραφή FSM – Finite State Machine ή Αλγοριθμική Μηχανή Καταστάσεων - σε γραφικό περιβάλλον)
- Block diagram (για δομική σχεδίαση σε γραφικό περιβάλλον)

Αφήνουμε τον τύπο του αρχείου στο VHDL code και προχωρούμε στον καθορισμό εισόδων και εξόδων για την οντότητά μας πατώντας το “Ports”.

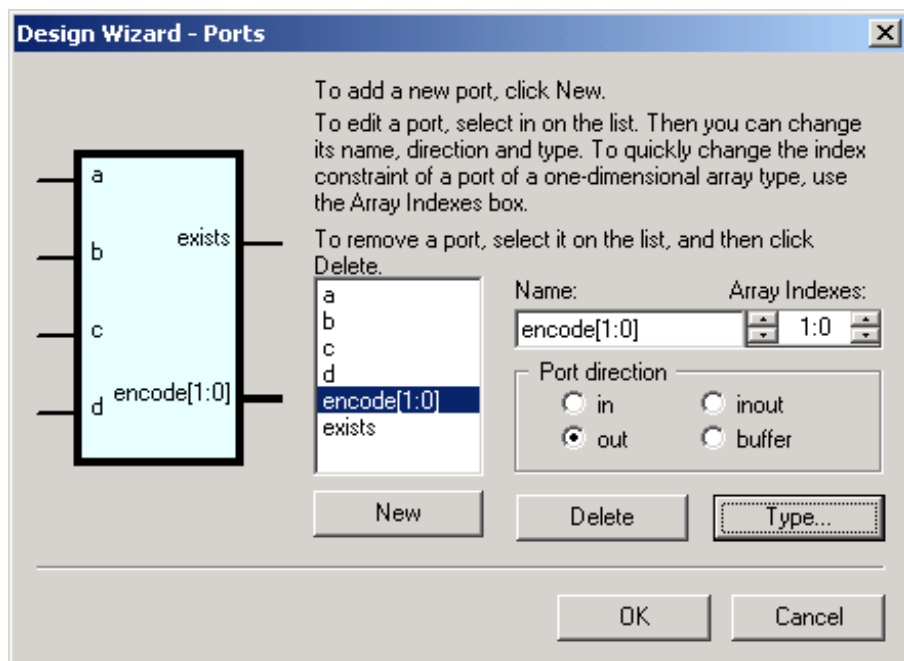
A.1.4 Δημιουργία εισόδων και εξόδων μιας οντότητας

Ο Ports Wizard (σχήμα 1.4) είναι το πρόγραμμα που μας βοηθά στην περιγραφή των εισόδων και των εξόδων που θα έχει μια οντότητα. Πατώντας “New” δημιουργούμε μια νέα θύρα εισόδου τύπου `std_logic`. Η κατεύθυνση της θύρας μπορεί να επιλεγεί από το “Port Direction” ενώ ο τύπος από το “Type”. Αν θέλουμε να κατασκευάσουμε διάδρομο, μπορούμε να αλλάξουμε τους δείκτες που αντιστοιχούν στο αριστερότερο και το δεξιότερο bit από το “Array Indexes”.

Δημιουργούμε με αυτό τον τρόπο τις εισόδους `a`, `b`, `c`, `d` και την έξοδο `exists`, τύπου `std_logic`. Επιπλέον, δημιουργούμε την έξοδο `encode` τύπου `std_logic_vector` με εύρος δεικτών (1 downto 0). Ο κώδικας VHDL που θα παραχθεί από τον ports wizard είναι ο εξής:

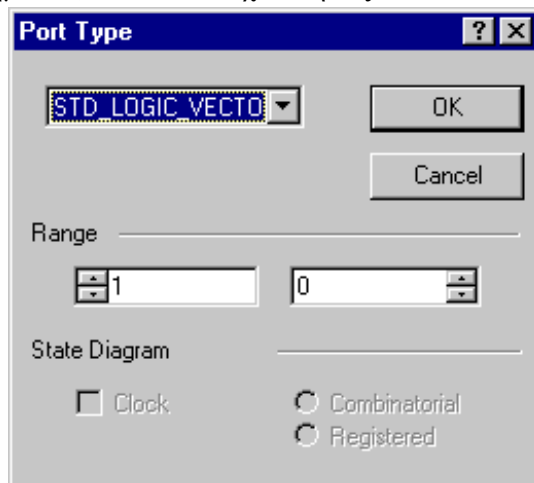
```
port (  
    a : in std_logic;  
    b : in std_logic;  
    c : in std_logic;  
    d : in std_logic;  
    exists : out std_logic;  
    encode : out std_logic_vector(1 downto 0));
```

Αντί να χρησιμοποιήσουμε τον Ports Wizard, μπορούμε να γράψουμε τον κώδικα απευθείας στο αρχείο αργότερα. Αυτό είναι απαραίτητο όταν πρέπει να χρησιμοποιήσουμε διαδρόμους με μεταβλητό εύρος (με χρήση `generics`) ή χρειαζόμαστε τύπο που δεν υποστηρίζεται από τον wizard.



Σχ. 1.4: Ports Wizard

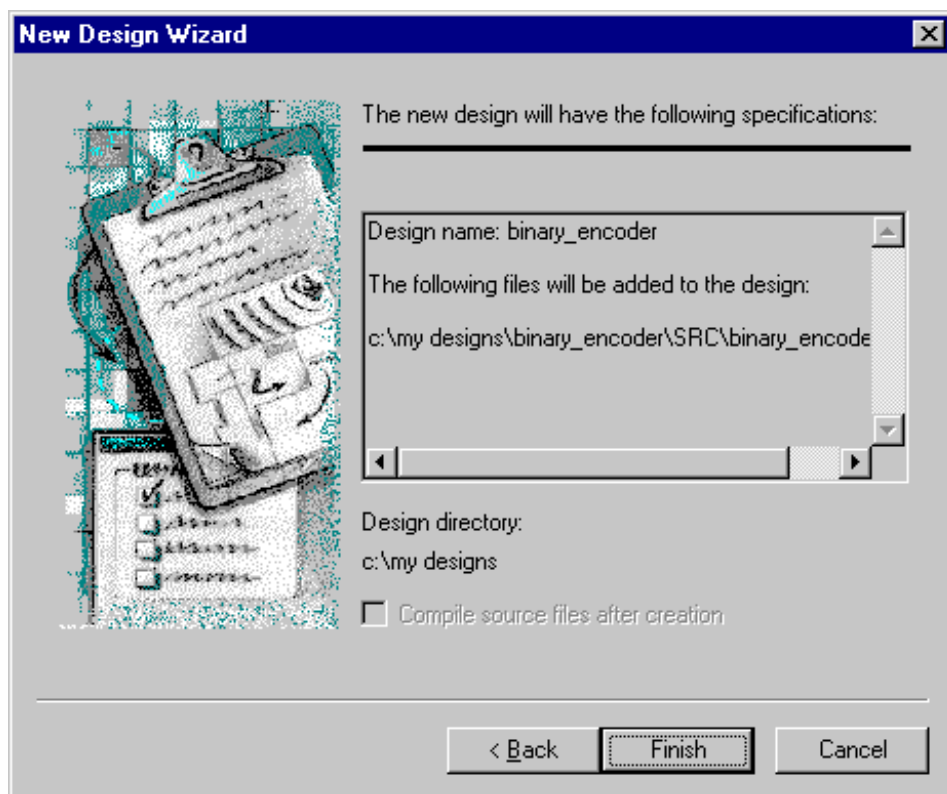
Η επιλογή του τύπου της θύρας γίνεται από το “Type”, με νέο παράθυρο, όπως φαίνεται στο σχήμα 1.5, που αντιστοιχεί στην έξοδο encode.



Σχ. 1.5: Επιλογή τύπου θύρας εισόδου/εξόδου

A.1.5 Αποδοχή των επιλογών και δημιουργία της σχεδίασης

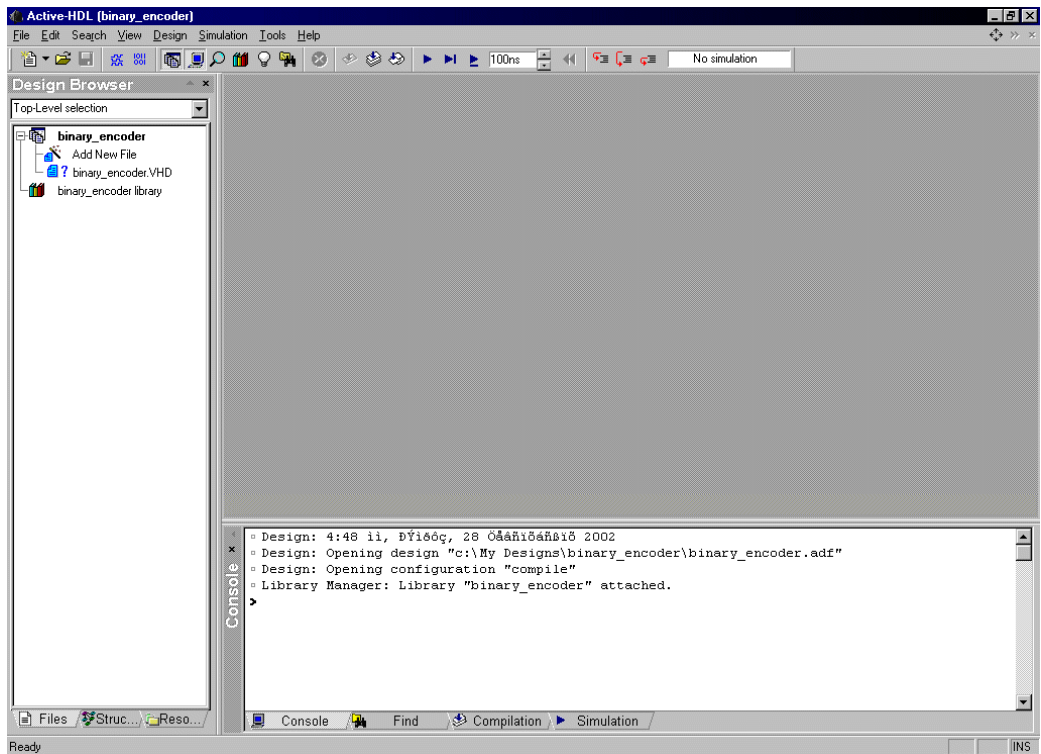
Στο τελευταίο παράθυρο του design wizard (σχήμα 1.6) μπορούμε να αποδεχτούμε τις επιλογές μας με “Finish” ή να γυρίσουμε πίσω και να κάνουμε αλλαγές με “Back”.



Σχ. 1.6: New Design Wizard – παράθυρο 4

A.2 Το περιβάλλον εργασίας

Μόλις ολοκληρώσουμε την δημιουργία της οντότητας `binary_encoder` περνάμε στο περιβάλλον εργασίας της Active HDL, που αποτελείται από πολλά παράθυρα (σχήμα 1.7). Ο Design Browser αριστερά μας δίνει πληροφορίες για την κατάσταση της σχεδίασής μας και η Console στο κάτω μέρος καταγράφει τις εντολές που δίνουμε στα προγράμματα του περιβάλλοντος και τα αποτελέσματα που παίρνουμε.



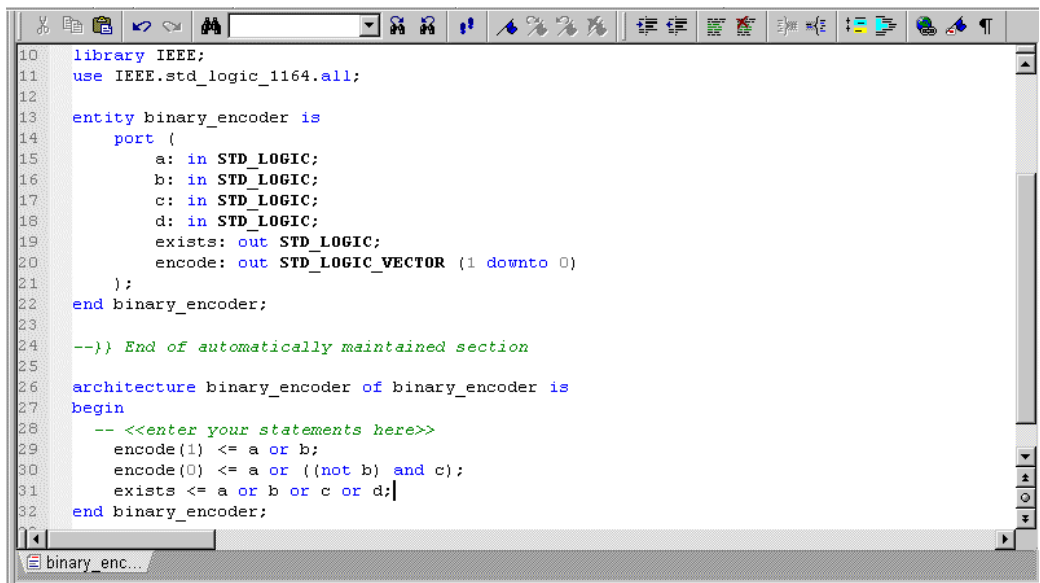
Σχ. 1.7: Το κυρίως περιβάλλον εργασίας του Active HDL

A.2.1 Εισαγωγή κώδικα

Ο Design Browser έχει τρεις όψεις, “File”, “Structure” και “Resources” και η επιλογή της τρέχουσας όψης γίνεται από το κάτω μέρος του παραθύρου. Αρχικά είμαστε σε File view και μπορούμε να δούμε το design μας (binary_encoder) και το αρχείο που κατασκευάσαμε (binary_encoder.vhd) με ένα ερωτηματικό αριστερά του, που υποδηλώνει ότι δεν έχει γίνει ακόμη compilation. Επιλέγοντας το αρχείο μπορούμε να το ανοίξουμε σε ένα νέο παράθυρο τύπου HDL editor. Εισάγουμε στην θέση που υποδεικνύουν τα σχόλια μέσα στην περιγραφή της αρχιτεκτονικής (βλέπε σχήμα 1.8) την παρακάτω περιγραφή του binary_encoder (κωδικοποιητή προτεραιότητας):

```
encode(1) <= a or b;  
encode(0) <= a or ((not b) and c);  
exists <= a or b or c or d;
```

Η λειτουργία του κωδικοποιητή αυτού είναι η εξής: Οι είσοδοι a, b, c, d αντιστοιχίζονται στους αριθμούς 3, 2, 1, 0. Όταν κάποια είσοδος είναι ενεργή (λογικό 1), ο αντίστοιχος αριθμός της εμφανίζεται στην έξοδο encode. Αν δυο ή περισσότεροι είσοδοι είναι ενεργές, τότε η επιλογή του αριθμού που εμφανίζεται στην έξοδο γίνεται με βάση την προτεραιότητά τους. Μέγιστη προτεραιότητα έχει η είσοδος a και ελάχιστη η είσοδος d. Η έξοδος exists ανιχνεύει αν υπάρχει τουλάχιστον μια είσοδος ενεργή και συνεπώς δηλώνει αν η έξοδος encode είναι έγκυρη (ελέγξτε αν η περιγραφή ανταποκρίνεται στην επιθυμητή λειτουργία).

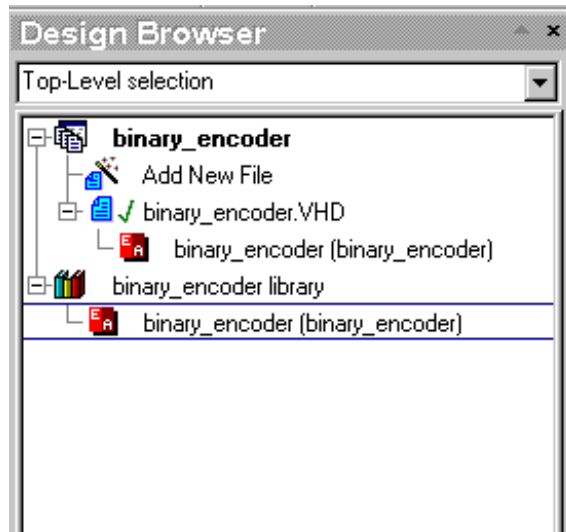


Σχ. 1.8: Προσθήκη εντολών στην αρχιτεκτονική του κωδικοποιητή

A.2.2 Compilation

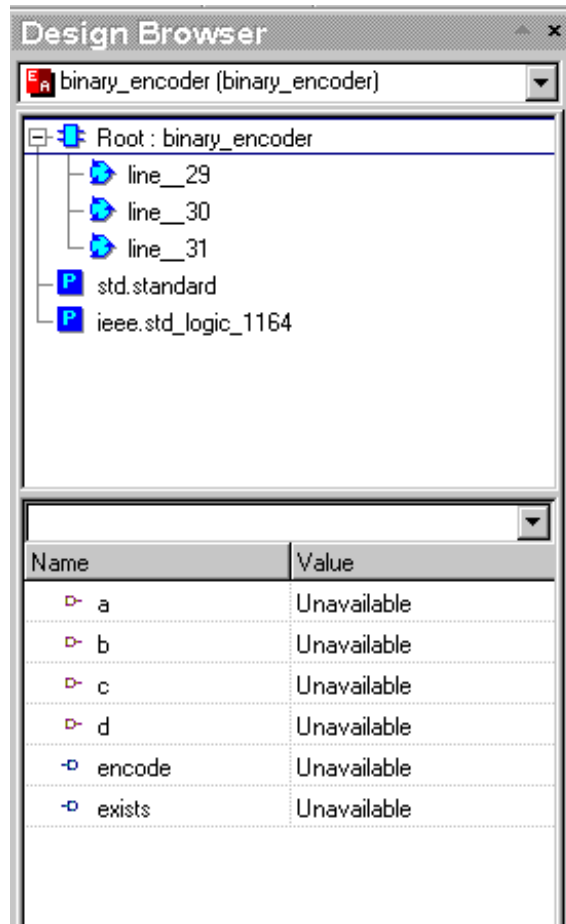
Τώρα είμαστε έτοιμοι να χρησιμοποιήσουμε τον VHDL compiler (Menu : Design/Compile) για να εισάγουμε την οντότητα binary_encoder και την αρχιτεκτονική της στη βιβλιοθήκη. Αφού τρέξει ο compiler, το σύμβολο δίπλα στο

αρχείο μας στον Design Browser μετατρέπεται από ερωτηματικό σε tick (αν υπήρχε κάποιο λάθος θα είχε γίνει X) και στη βιβλιοθήκη του design υπάρχει πλέον ένα ζεύγος οντότητας-αρχιτεκτονικής που αντιστοιχεί στον binary_encoder. Όλα αυτά φαίνονται στο σχήμα 1.9.



Σχ. 1.9: Ο Design Browser μετά από compilation

Περνώντας σε Structure View, μπορούμε να δούμε τη δομή της σχεδίασης. Επιλέγουμε ως top-level τον binary_encoder και έχουμε την εξής εικόνα (σχήμα 1.10): Η σχεδίασή μας αποτελείται από ένα Root block με το όνομα binary_encoder και δυο packages (std.standard και ieee.std_logic_1164). Το block απαρτίζεται από τρεις (3) εντολές, οι οποίες δημιουργήθηκαν από τον κώδικα που εισάγαμε και οδηγούν τις εξόδους του κυκλώματος. Επιλέγοντας το block μπορούμε να δούμε τις εισόδους και τις εξόδους του στο κάτω μέρος του Design Browser.

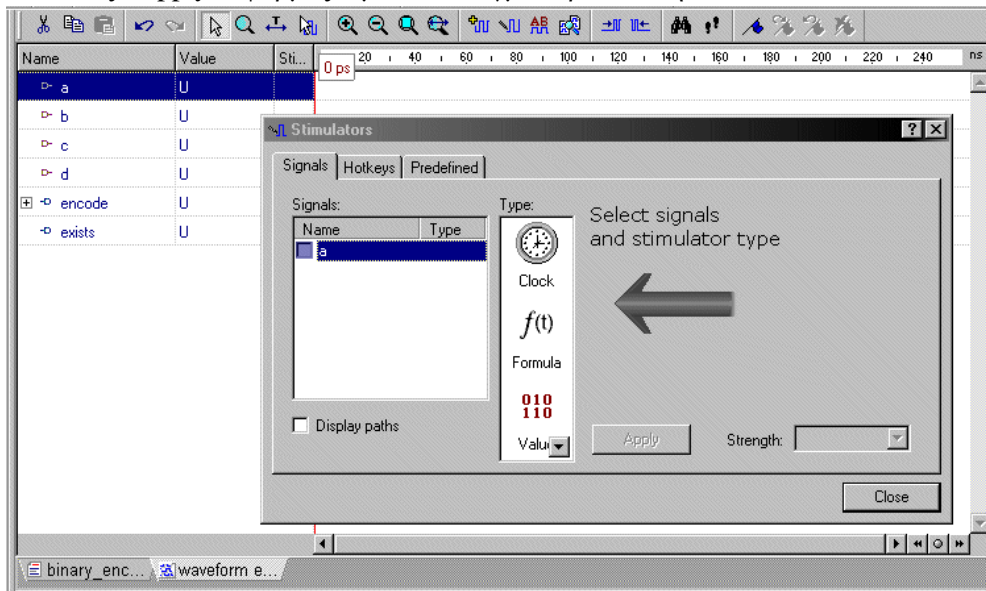


Σχ. 1.10: Structure view του Design Browser μετά από compilation

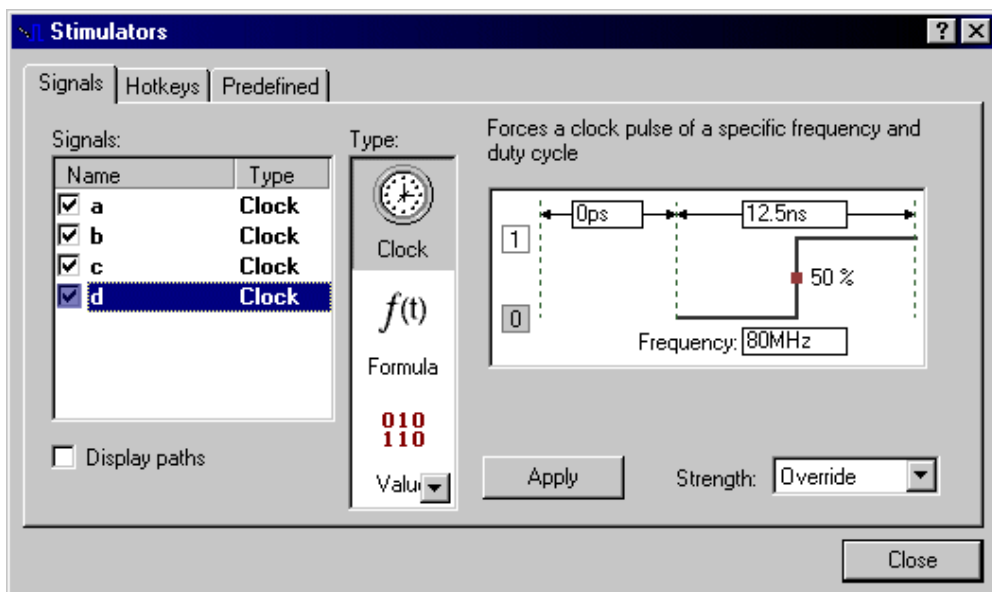
A.2.3 Προσομοίωση

Προσομοίωση του `binary_encoder` γίνεται άμεσα. Δημιουργούμε ένα νέο παράθυρο κυματομορφής (Menu: File/New/Waveform) και εισάγουμε σε αυτό τα σήματα που επιθυμούμε με τη μέθοδο drag and drop (ή Menu : Waveform/Add Signals). Κατόπιν αρχικοποιούμε την προσομοίωση (Menu : Simulation/Initialize Simulation) και προσδιορίζουμε τις κυματομορφές των εισόδων (Menu : Waveform/Stimulators). Ο προσδιορισμός των κυματομορφών γίνεται από ένα παράθυρο όπως αυτό του σχήματος 1.11. Για τις εισόδους a, b, c, d του `binary_encoder` χρησιμοποιούμε κυματομορφές ρολογιού με συχνότητες 10, 20, 40

και 80 MHz, όπως φαίνεται στο σχήμα 1.12. Αφού επιλέξαμε τη συχνότητα, πατώντας “Apply” εφαρμόζουμε το επιλεγμένο ρολόι στην είσοδο.

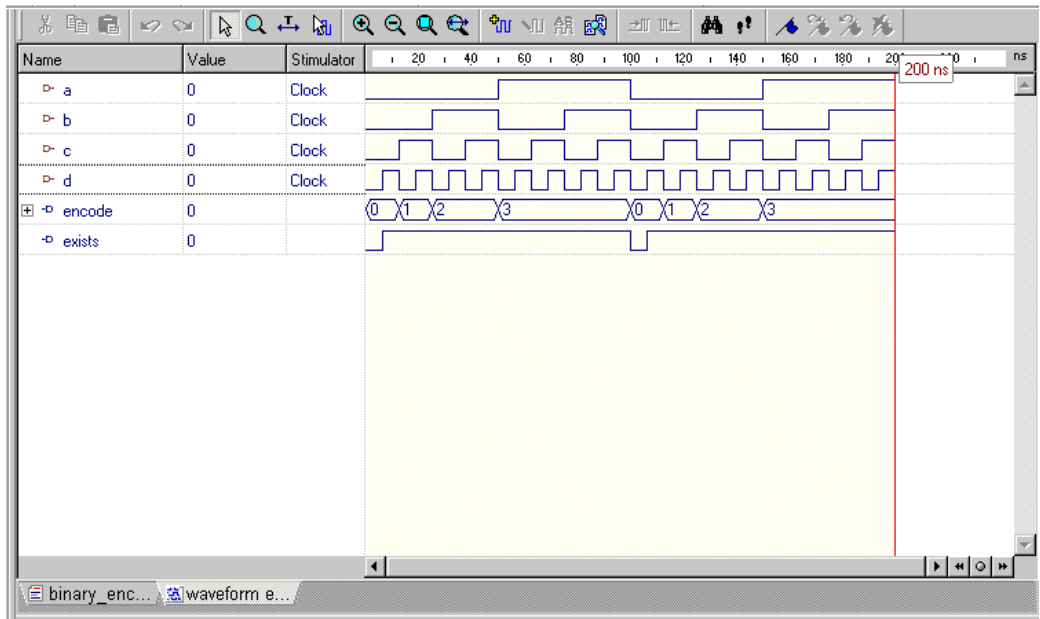


Σχ. 1.11: Προσδιορισμός κυματομορφών εισόδου



Σχ. 1.12: Αντιστοίχιση ρολογιού στην είσοδο d του κωδικοποιητή

Τρέχουμε την προσομοίωση για 200 ns (Menu : Simulation/Run Until/200 ns) και παρατηρούμε ότι η συμπεριφορά του binary_encoder είναι η αναμενόμενη (σχήμα 1.13).

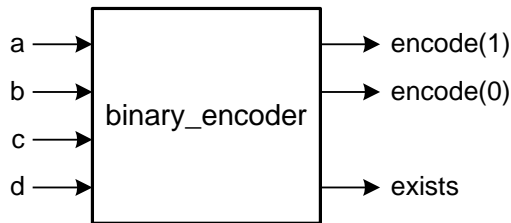


Σχ. 1.13: Προσομοίωση κωδικοποιητή

A.3 Άσκηση 1

Σχεδίαση και προσομοίωση απλών συνδυαστικών λογικών συναρτήσεων σε dataflow και behavioral περιγραφή

Θέμα A.1: Δυαδικός κωδικοποιητής προτεραιότητας 4 σε 2



Σχήμα 1.14 Δυαδικός κωδικοποιητής 4 σε 2

Με τη βοήθεια του Active HDL Tutorial, προσομοιώστε την παρακάτω dataflow αρχιτεκτονική ενός δυαδικού κωδικοποιητή τεσσάρων εισόδων. Κάθε είσοδος αντιστοιχεί σε έναν αριθμό: (a - 3, b - 2, c - 1, d - 0). Όταν μόνο μία είσοδος είναι ενεργή, ο αντίστοιχος αριθμός εμφανίζεται στην έξοδο encode. Αν περισσότερες από μια εισόδους είναι ενεργές, στην έξοδο εμφανίζεται ο μεγαλύτερος αριθμός (κωδικοποίηση κατά προτεραιότητα). Σε αυτές τις περιπτώσεις η έξοδος exists είναι στο λογικό 1. Αν δεν υπάρχουν ενεργές εισοδοί, τότε η έξοδος encode τίθεται σε κατάσταση απομόνωσης ενώ η έξοδος exists είναι 0.

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

entity binary_encoder is

    port (
        a: in std_logic;
        b: in std_logic;
        c: in std_logic;
        d: in std_logic;
        exists: inout std_logic;
        encode: out std_logic_vector(1 downto 0));
end binary_encoder;
```



```

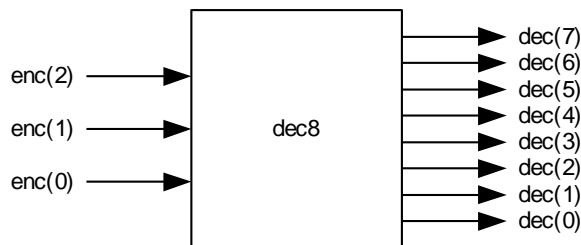
architecture binary_encoder of binary_encoder is
begin
    encode(1) <= a or b when exists='1' else 'Z';
    encode(0) <= a or ((not b) and c) when exists='1'
        else 'Z';
    exists <= a or b or c or d;
end binary_encoder;

```

Ζητούμενο: Να περιγραφεί σε behavioral αρχιτεκτονική και να προσομοιωθεί το ίδιο κύκλωμα.

Υπόδειξη: Χρησιμοποιήστε δομή if σε μια κατάλληλη σειρά ελέγχου των εισόδων για να εξασφαλίσετε την ύπαρξη προτεραιότητας.

Θέμα A.2: Δυαδικός αποκωδικοποιητής 3 σε 8



Σχήμα 1.15 Δυαδικός αποκωδικοποιητής 3 σε 8

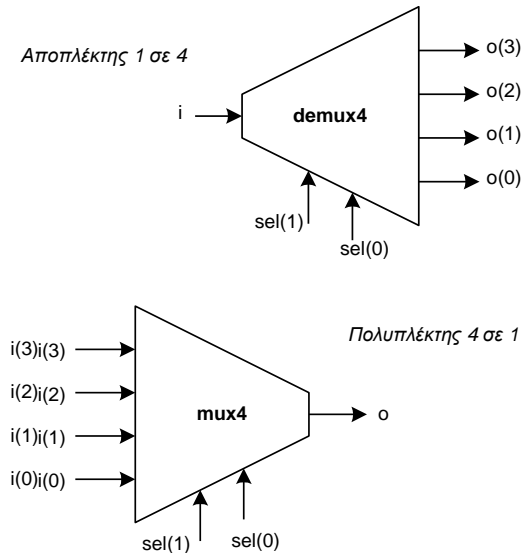
Ένας δυαδικός αποκωδικοποιητής 3 σε 8 είναι ένα συνδυαστικό κύκλωμα που ενεργοποιεί διαφορετική έξοδο (μία από τις 8) για κάθε διαφορετικό συνδυασμό των 3 εισόδων.

Ζητούμενο: Να δοθεί η περιγραφή της οντότητας του δυαδικού αποκωδικοποιητή 3 σε 8 και της αρχιτεκτονικής του σε dataflow και behavioral VHDL. Κατόπιν να γίνει προσομοίωση και στις δύο αρχιτεκτονικές

Υπόδειξη: Χρησιμοποιήστε τον τύπο std_logic_vector τόσο για τις εισόδους όσο και για τις εξόδους. Για την behavioral αρχιτεκτονική χρησιμοποιήστε τη δομή case.

Θέμα Α.3: Αποπλέκτης 1 σε 4 και πολυπλέκτης 4 σε 1

Στο σχήμα 1.16 φαίνονται τα σχηματικά διαγράμματα ενός αποπλέκτη 1 σε 4 και ενός πολυπλέκτη 4 σε 1. Τα σήματα έλεγχου *sel(1)* και *sel(0)* καθορίζουν ποιο από τα τέσσερα θα συνδεθεί με την είσοδο ή την έξοδο αντίστοιχα.



Σχήμα 1.16 Αποπλέκτης 1 σε 4 και πολυπλέκτης 4 σε 1

Ο παρακάτω κώδικας περιγράφει τη λειτουργία ενός αποπλέκτη 1 σε 4 με δύο μορφές behavioral και μία dataflow VHDL.

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

entity demux4 is
    port (
        sel : in std_logic_vector(1 downto 0);
        i   : in std_logic;
        o   : out std_logic_vector(3 downto 0));
end demux4;
```

```

architecture demux4_behavioural of demux4 is
begin

process(i,sel)
begin
    case sel is
        when "00" => o <= (0 => i, 1 => '0', 2 => '0', 3 => '0');
        when "01"
=> o <= (1 => i, others => '0');
        when "10" => o <= "0" & i & "00";
        when "11" => o <= i & "000";
        when others => null;
    end case;
end process;
end demux4_behavioural;

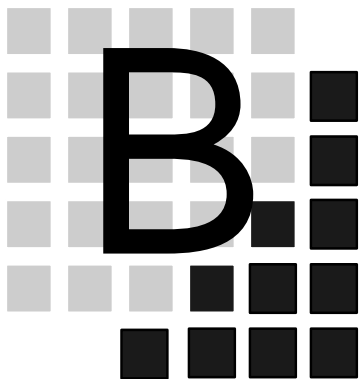
architecture demux4_dataflow of demux4 is
begin
    o(0) <= (not sel(1)) and (not sel(0)) and i;
    o(1) <= (not sel(1)) and sel(0) and i;
    o(2) <= sel(1) and (not sel(0)) and i;
    o(3) <= sel(1) and sel(0) and i;
end demux4_dataflow;

architecture demux4_behavioural_integer of demux4 is
begin
    process(i,sel)
        variable temp : std_logic_vector(3 downto 0);
    begin
        temp := "0000";
        temp(conv_integer(sel)) := i;
        o <= temp;
    end process;
end demux4_behavioural_integer;

```

Ζητούμενο: Να περιγραφεί σε behavioral και dataflow VHDL η αρχιτεκτονική ενός πολυπλέκτη 4 σε 1. Κατόπιν να γίνει προσομοίωση και στις δύο αρχιτεκτονικές. Δώστε και δεύτερη behavioral αρχιτεκτονική.

Υπόδειξη: Για τη 2^η behavioral αρχιτεκτονική, εναλλακτικά της δομής case, μπορεί να χρησιμοποιηθεί η συνάρτηση `conv_integer` με τρόπο ανάλογο με τον παραπάνω κώδικα.



ΤΟ ΠΕΡΙΒΑΛΛΟΝ ΣΥΝΘΕΣΗΣ LEONARDO SPECTRUM

B.1 Σύνθεση κυκλωμάτων με το Leonardo Spectrum

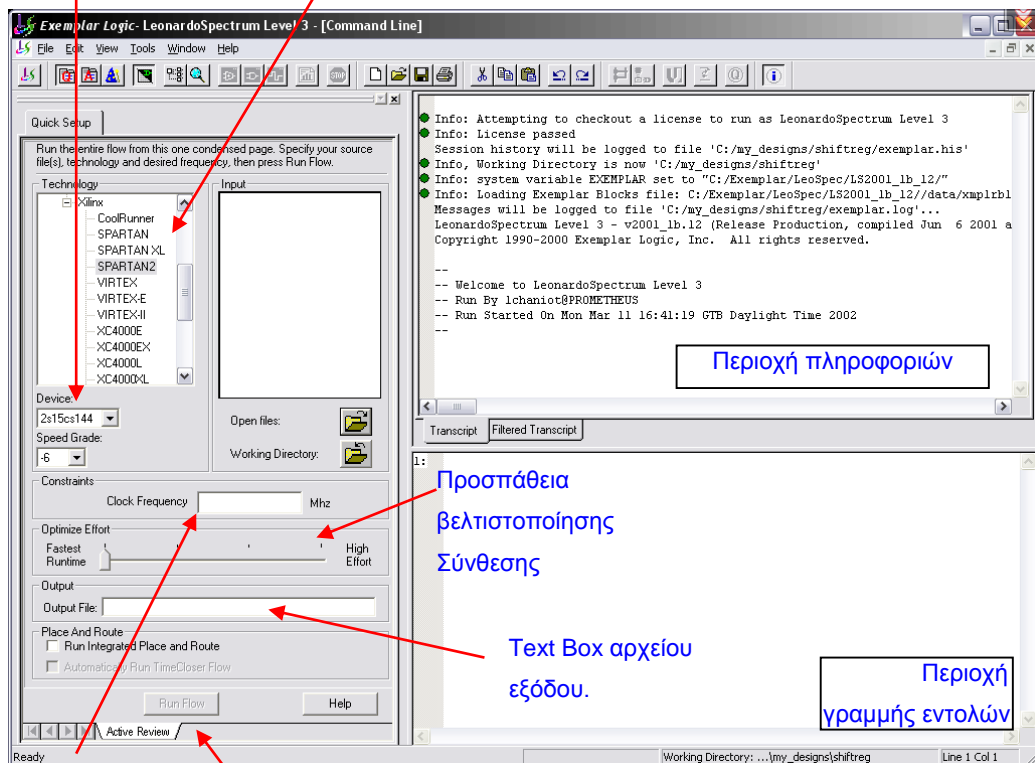
B.1.1 Χρησιμότητα του προγράμματος

Το Leonardo Spectrum είναι ένα πρόγραμμα το οποίο χρησιμεύει στη σύνθεση ενός ολοκληρωμένου ψηφιακού κυκλώματος, δηλαδή στη μετάφραση της περιγραφής του από μια γλώσσα περιγραφής υλικού σε ένα netlist το οποίο θα χρησιμοποιηθεί για την υλοποίηση και κατασκευή του ολοκληρωμένου κυκλώματος. Το netlist αυτό μπορεί να εξαχθεί αυτόματα με τη χρήση προγραμμάτων που λέγονται synthesizers, όπως είναι το Leonardo Spectrum. Η σύνθεση μετατρέπει μια περιγραφή που είναι ανεξάρτητη της τεχνολογίας πάνω στην οποία βασίζεται το τελικό ολοκληρωμένο, σε μια περιγραφή η οποία εξαρτάται από αυτήν. Περιγραφές VHDL οι οποίες φαίνονται να λειτουργούν σωστά στο simulator πολλές φορές δεν μπορούν να περάσουν με επιτυχία το στάδιο της σύνθεσης. Για το λόγο αυτό επιβάλλεται κατά τον σχεδιασμό και την κατασκευή κάθε ολοκληρωμένου κυκλώματος να ελέγχεται και το κύκλωμα που προκύπτει από τη σύνθεση για την ορθότητα της λειτουργίας του.

B.1.2 Περιγραφή του Leonardo Spectrum

Η κεντρική οθόνη του Leonardo Spectrum φαίνεται στο σχήμα 2.1.

Λίστα Συσκευών

Λίστα
ΤεχνολογιώνText Box
περιορισμού
Συχνότητας
Ρολογιού

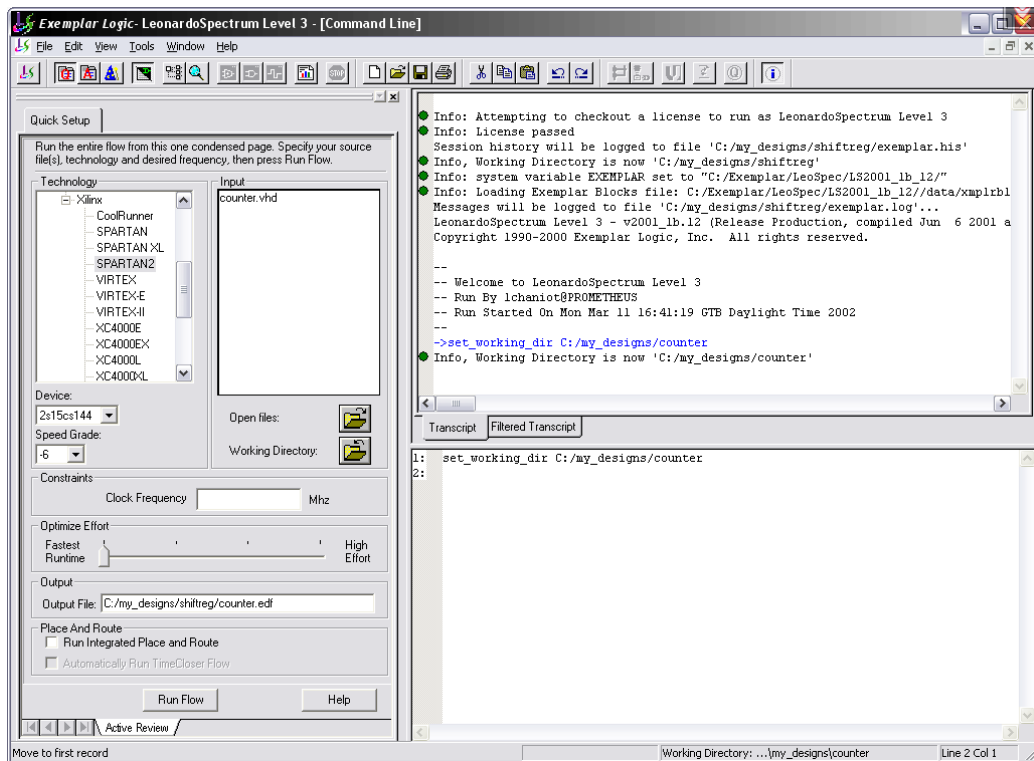
Έναρξη σύνθεσης

Σχ. Π2.1: Η κεντρική οθόνη του Leonardo Spectrum

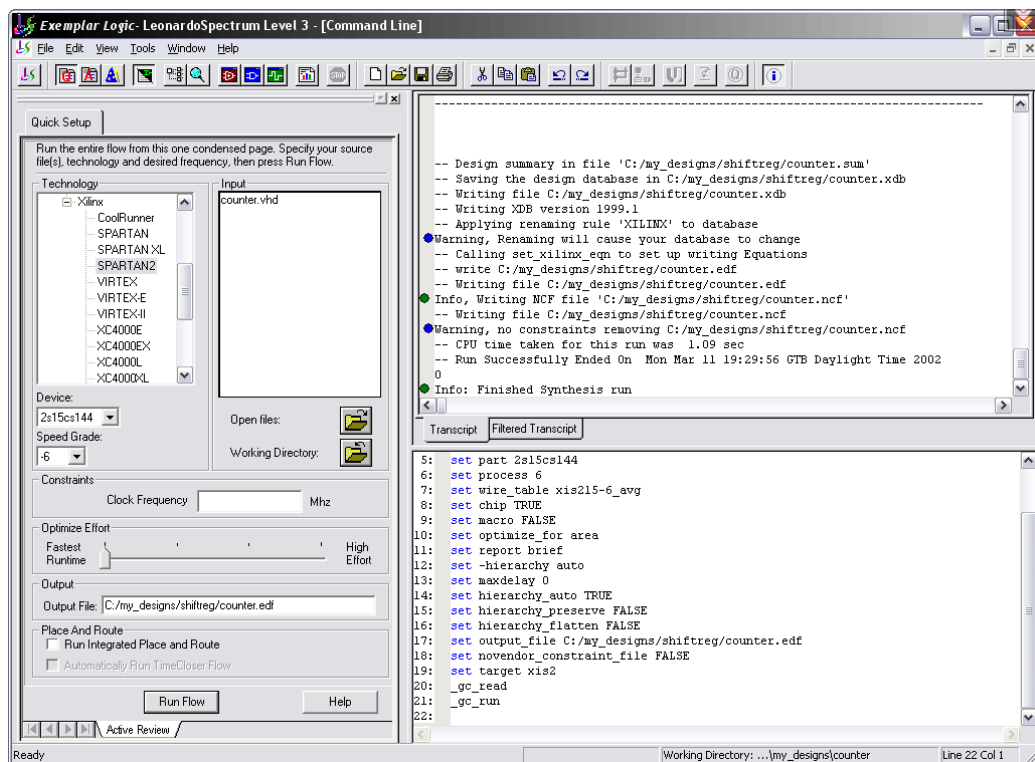
Για να κάνουμε σύνθεση σε μια περιγραφή ενός κυκλώματος σε VHDL χρειάζονται τα εξής βήματα (θεωρούμε ότι το αρχείο με την οντότητα που θέλουμε να συνθέσουμε είναι το counter.vhd):

1. Διαλέγουμε τεχνολογία από τη Λίστα Τεχνολογιών. Για την περίπτωση μας διαλέγουμε FPGA → Xilinx → SPARTAN2
2. Διαλέγουμε το μοντέλο του FPGA από τη Λίστα Συσκευών. Εδώ διαλέγουμε το 2s15cs144.

3. Στη συνέχεια διαλέγουμε το directory εργασίας του προγράμματος, κάνοντας κλικ στο κουμπί Working Directory. Επιλέξτε αυτό στο οποίο βρίσκεται το αρχείο counter.vhd.
4. Κάνοντας κλικ στο κουμπί Open Files διαλέγουμε το αρχείο counter.vhd.
5. Στη συνέχεια μπορούμε να δώσουμε περιορισμό για τη συχνότητα του ρολογιού (π.χ. 50 MHz) στο αντίστοιχο Text Box. Αυτό δεν είναι απαραίτητο βήμα, αλλά συνηθίζεται για την βελτιστοποίηση του αποτελέσματος της σύνθεσης. Για την εισαγωγική αυτή επαφή μας με το εργαλείο το αφήνουμε κενό.
6. Η διαδικασία της σύνθεσης μπορεί να διαρκέσει πολύ σε μια μεγάλη σχεδίαση. Για το λόγο αυτό παρέχεται η δυνατότητα για 4 διαφορετικούς βαθμούς βελτιστοποίησης. Ο χαμηλότερος παρέχει τη γρηγορότερη δυνατή εκτέλεση στην εξαγωγή της σύνθεσης (Fastest Runtime), ενώ ο υψηλότερος την μέγιστη απόδοση ως προς το εξαγόμενο κύκλωμα (High Effort). Και αυτό το αφήνουμε προς το παρόν στην θέση “Fastest Runtime”.
7. Διαλέγουμε στη συνέχεια το όνομα του αρχείου εξόδου. Αυτό το αρχείο εξόδου είναι το netlist που θα χρησιμοποιηθεί από άλλα εργαλεία για να φτιάξει το τελικό κύκλωμα επάνω στο FPGA. Διαλέγουμε το όνομα counter.edf
8. Τώρα είμαστε έτοιμοι να ξεκινήσουμε τη διαδικασία της σύνθεσης. Αυτό γίνεται κάνοντας κλικ στο κουμπί έναρξης “Run Flow”. Το παράθυρο του προγράμματος θα πρέπει να δείχνει περίπου όπως στο σχήμα 2.2. Μετά το πέρας της σύνθεσης, η οθόνη θα δείχνει όπως στο σχήμα 2.3.



Σχ. 2.2: Η κεντρική οθόνη του Leonardo πριν τη σύνθεση

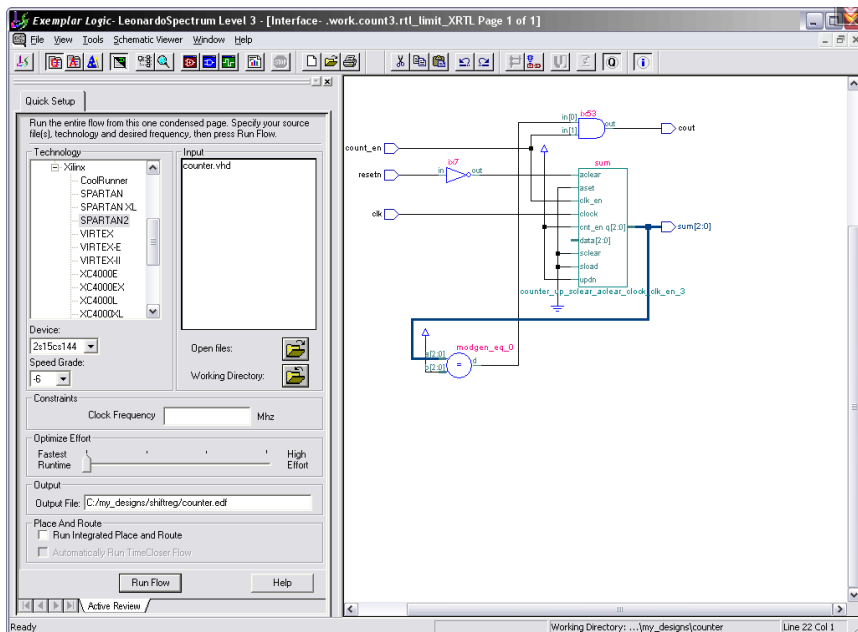


Σχ. 2.3: Η κεντρική οθόνη του Leonardo μετά τη σύνθεση

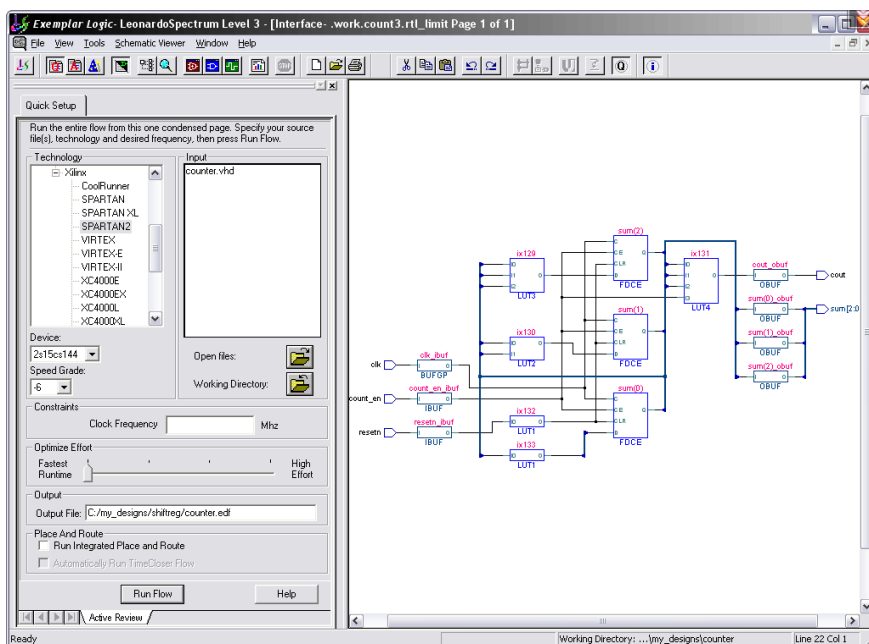
Με το τέλος της σύνθεσης μπορούμε να δούμε στην περιοχή πληροφοριών αναφορές για το μέγεθος και τη συχνότητα λειτουργίας του κυκλώματος. Είναι επίσης δυνατόν να δούμε σε γραφική μορφή το κύκλωμα που προέκυψε από την σύνθεση. Το Leonardo μπορεί να μας δείξει δύο είδη σχηματικών:

1. Το RTL σχηματικό το οποίο δείχνει τη λογική του κυκλώματος χωρίς να κάνει αντιστοίχιση σε συγκεκριμένη τεχνολογία (σχήμα 2.4). Για να δείτε αυτό το σχηματικό πατήστε το κουμπί που φαίνεται στα δεξιά.
2. Το τεχνολογικό σχηματικό το οποίο δείχνει το κύκλωμα με βάση τα πρωτογενή στοιχεία της τεχνολογίας που χρησιμοποιείται για την υλοποίηση, ASIC ή FPGA (σχήμα 2.5). Αυτό το σχηματικό διαφέρει πολύ από το RTL σχηματικό και μπορείτε να το δείτε με το κουμπί που φαίνεται στα δεξιά.

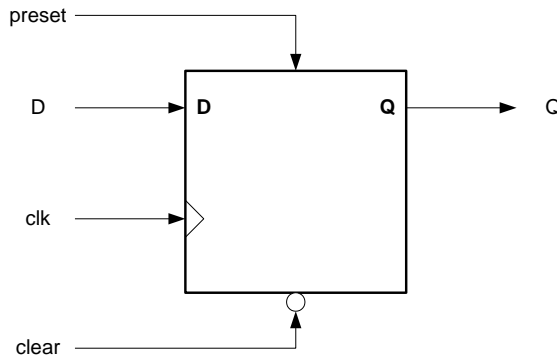




Σχ. 2.4: RTL σχηματικό του counter.vhd



Σχ. 2.5: Τεχνολογικό σχηματικό του counter.vhd

*Σχεδίαση και προσομοίωση κυκλωμάτων καταχωρητών και μετρητών***Θέμα B.1: Απλός καταχωρητής 1 bit (D Flip-Flop)****Σχ. 2.6:** D Flip-Flop με εισόδους preset και clear

Με τη βοήθεια του Active HDL, εξομοιώστε την παρακάτω behavioral αρχιτεκτονική ενός D Flip-Flop με ασύγχρονες εισόδους μηδενισμού και θέσης (Preset). Το D Flip Flop είναι η βασική μονάδα αποθήκευσης δεδομένων και μπορεί να αποθηκεύσει την τιμή ενός bit (καταχωρητής 1 bit).

```

library IEEE;
use IEEE.std_logic_1164.all;

entity dff is
    port (d          : in std_logic;
          preset     : in std_logic;
          clear      : in std_logic;
          clk        : in std_logic;
          q          : out std_logic);
end dff;

architecture bhv_dff of dff is
begin
    process(clk,clear,preset)
    begin
        if clear = '0' then

```

```

-- To clear έχει μεγαλύτερη προτεραιότητα
    q <= '0';
    elsif preset = '1' then
        q <= '1';
    elsif clk'event and clk='1' then
        q <= d;
    end if;
end process;
end bhv_dff;

```

Πρέπει να σημειωθεί εδώ ότι το κύκλωμα που προκύπτει από αυτή την περιγραφή δεν είναι υλοποιήσιμο σε όλες τις τεχνολογίες. Μερικές επιτρέπουν και τις δύο ασύγχρονες εισόδους, ενώ άλλες επιτρέπουν την χρήση μόνο της μιας.

Ζητούμενα:

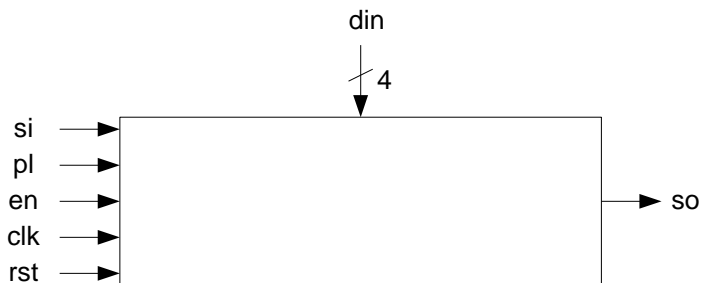
1. Με βάση την περιγραφή του D Flip-Flop να ορίσετε την behavioral περιγραφή ενός T Flip-Flop με ασύγχρονες εισόδους preset και clear.

Υπόδειξη: Το Flip-Flop τύπου T είναι αυτό που σε κάθε παλμό του ρολογιού αλλάζει (toggle) η έξοδός του.

2. Με οδηγό την περιγραφή ενός D Flip-Flop, να περιγράψετε σε behavioral VHDL έναν καταχωρητή 4 bits με είσοδο παράλληλης φόρτωσης.

Υπόδειξη: Να χρησιμοποιήσετε τον τύπο `std_logic_vector` για να περιγράψετε την είσοδο και την έξοδο του καταχωρητή.

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



Σχ. 2.7: Καταχωρητής δεξιάς ολίσθησης των 4 bits

Ο καταχωρητής ολίσθησης (σχήμα 2.7) είναι ένα κύκλωμα το οποίο δέχεται μια παράλληλη είσοδο `din` (Data in) η οποία φορτώνεται μέσω του σύγχρονου

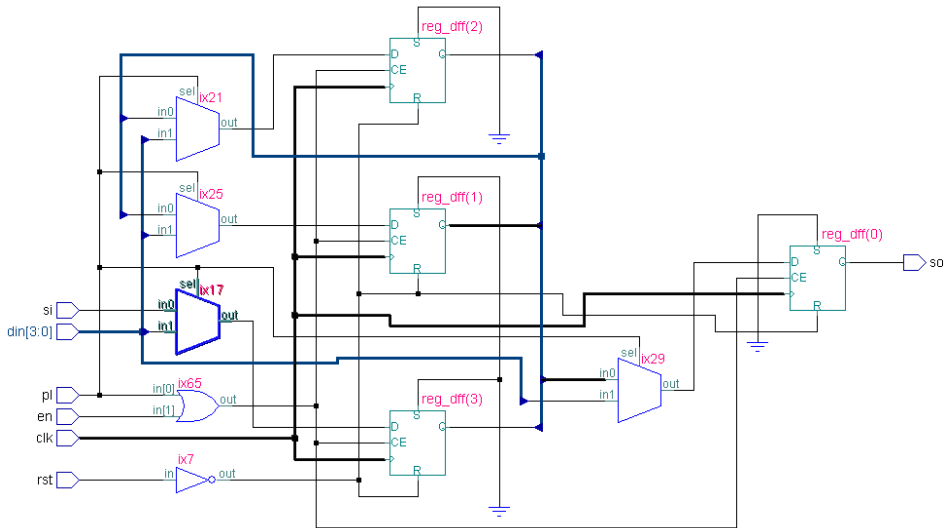
σήματος ενεργοποίησης παράλληλης φόρτωσης pl (Parallel Load). Η ενεργοποίηση της ολίσθησης γίνεται μέσω του σήματος en (Enable). Ο καταχωρητής έχει και μια σειριακή είσοδο si (Serial Input), καθώς και μια σειριακή έξοδο so (Serial Output). Κατά την δεξιά ολίσθηση το περιεχόμενο του καταχωρητή ολισθαίνει κατά μια θέση προς το LSB (bit 0). Το LSB βγαίνει στην έξοδο so και η είσοδος si περνά στο MSB του καταχωρητή. Αυτό γίνεται σε κάθε θετικό παλμό του ρολογιού clk. Η ασύγχρονη είσοδος rst (reset) μηδενίζει τα flip-flops του καταχωρητή ολίσθησης. Η περιγραφή του καταχωρητή ολίσθησης σε behavioral VHDL είναι η εξής:

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

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

architecture rtl of rshift_reg3 is
    signal dff: std_logic_vector(3 downto 0);
begin
    edge: process (clk,rst)
    begin
        if rst='0' then
            dff<=(others=>'0');
        elsif clk'event and clk='1' then
            if pl='1' then
                dff<=din;
            elsif en='1' then
                dff<=si&dff(3 downto 1);
            end if;
        end if;
    end process;
    so <= dff(0);
end rtl;
```

Με τη βοήθεια του Active-HDL εξομοιώστε την αρχιτεκτονική του παραδείγματος. Το κύκλωμα που προκύπτει από τον synthesizer για την περιγραφή αυτή είναι αυτό που φαίνεται στο σχήμα 2.8. Φαίνονται τα Flip-Flops που αποτελούν τον καταχωρητή, μαζί με το συνδυαστικό κύκλωμα που καθορίζει τη λειτουργία του shift register. Ελέγξτε την ορθότητα λειτουργίας του κυκλώματος αυτού και αν θα μπορούσε να σχεδιαστεί “με το χέρι” σε απλούστερη μορφή.



Σχ. 2.8: Καταχωρητής ολίσθησης των 4 bits – Κύκλωμα που προκύπτει από τον synthesizer

Ζητούμενο: Να περιγραφεί η οντότητα του καταχωρητή ολίσθησης με μια επιπλέον είσοδο (std_logic) η οποία θα επιλέγει ανάμεσα σε αριστερή και δεξιά ολίσθηση. Υπενθυμίζεται ότι στην αριστερή ολίσθηση η έξοδος είναι το MSB του καταχωρητή και η σειριακή είσοδος γίνεται από το LSB. Για την περιγραφή την οποία θα φτιάξετε να ελέγξετε και το κύκλωμα που προκύπτει από τον synthesizer, παρατηρώντας τις διαφορές που έχει από το κύκλωμα που δίνεται στο σχήμα 2.8

Υπόδειξη: Χρησιμοποιήστε την περιγραφή του καταχωρητή δεξιάς ολίσθησης, με έναν επιπλέον έλεγχο (μέσω εντολής case) για την επιλογή αριστερής ή δεξιάς ολίσθηση. Εναλλακτικά, δώστε μια δεύτερη λύση με χρήση της εντολής if.

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

Ο μετρητής που θα παρουσιάσουμε στα επόμενα είναι μονής κατεύθυνσης (δηλαδή μετράει μόνο προς τα πάνω) και διαθέτει ασύγχρονη είσοδο μηδενισμού

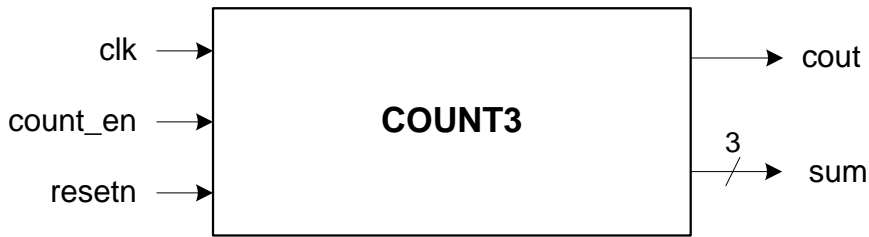
και σύγχρονη είσοδο ενεργοποίησης. Η είσοδος μηδενισμού είναι ενεργή σε λογικό '0', ενώ η είσοδος ενεργοποίησης είναι ενεργή σε λογικό '1'.

Εάν χρησιμοποιηθεί το package `ieee.std_logic_unsigned`, αρκεί να χρησιμοποιηθούν οι συναρτήσεις "+" ή "-" όταν πρέπει να αυξηθεί ή να μειωθεί η τιμή του μετρητή (στο παράδειγμα που ακολουθεί η τιμή του μετρητή μόνο αυξάνει). Εάν ο μετρητής είναι δηλωμένος σαν `std_logic_vector` η τιμή του θα αλλάξει αυτόματα όταν όλα τα bits έχουν την τιμή '1' (στην περίπτωση του "+") και η τιμή του μετρητή θα μηδενιστεί. Εάν ο μετρητής πρέπει να σταματήσει σε μια τιμή, π.χ. "101" η τιμή αυτή πρέπει να ελεγχθεί πριν την πρόσθεση του +1. Ακολουθούν δύο αρχιτεκτονικές, μια με έλεγχο ορίου και μια χωρίς. Δοκιμάστε με τη βοήθεια του Active HDL τις δυο αυτές αρχιτεκτονικές.

Στην περίπτωση που δεν χρησιμοποιηθεί το package `ieee.std_logic_unsigned` για να γίνει η πρόσθεση ή η αφαίρεση ενός αριθμού σε ένα `std_logic_vector` θα πρέπει να γραφούν οι ακόλουθες γραμμές σε VHDL:

```
signal count: std_logic_vector (2 downto 0);
signal count_int: integer range (0 to 7);
...
count_int <= conv_integer (count);
count_int <= count_int+1;
count <= conv_std_logic_vector(count,3);
...
```

Οι γραμμές αυτές μετατρέπουν το `std_logic_vector` σε ακέραιο αριθμό, ώστε να είναι δυνατή η πρόσθεση του 1. Αφού γίνει η πρόσθεση, μετατρέπεται πάλι ο ακέραιος (με τη νέα τιμή) σε `std_logic_vector`.



Σχ. 2.9: Μετρητής 3 bit

```

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

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

architecture rtl_nolimit of count3 is
    signal count: std_logic_vector(2 downto 0);
begin
    process(clk, resetn)
    begin
        if resetn='0' then
            -- Κώδικας για την περίπτωση του reset (ενεργό χαμηλά)
            count <= (others=>'0');
        elsif clk'event and clk='1' then
            if count_en='1' then
                -- Μέτρηση μόνο αν count_en = 1
                count<=count+1;
            end if;
        end if;
    end process;
    -- Ανάθεση τιμών στα σήματα εξόδου

```



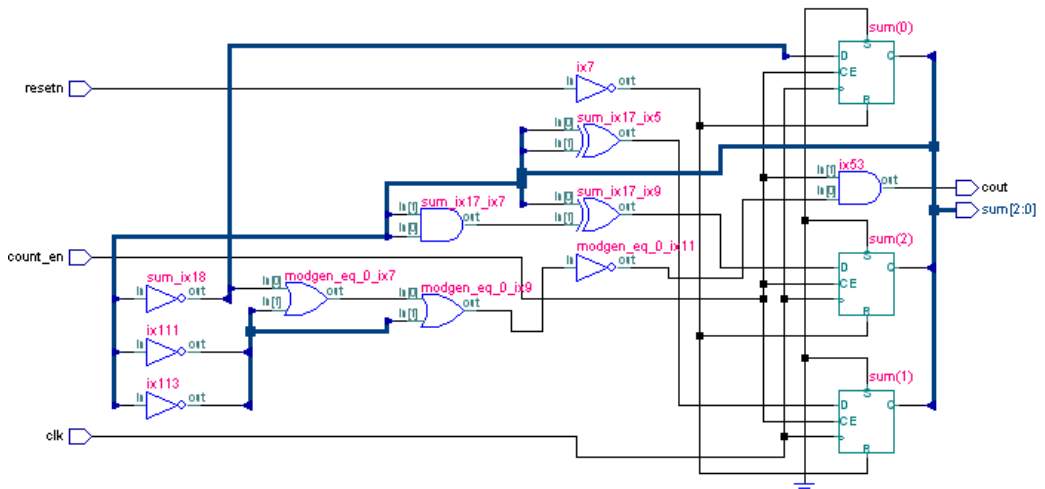
```

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

architecture rtl_limit of count3 is
signal count : std_logic_vector(2 downto 0);
begin
    process(clk, resetn)
    begin
        if resetn='0' then
            -- 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/=7 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;

```

Το όριο του μετρητή στη δεύτερη αρχιτεκτονική είναι το 7, πράγμα που σημαίνει ότι το κύκλωμα που θα προκύψει από τον synthesizer θα είναι το ίδιο και στις δύο περιπτώσεις. Το σχήμα 2.10 δείχνει το κύκλωμα αυτό. Ελέγξτε την ορθότητα λειτουργίας του κυκλώματος αυτού και αν θα μπορούσε να σχεδιαστεί “με το χέρι” σε απλούστερη μορφή.



Σχ. 2.10: Κύκλωμα μετρητή 3 bits από τον synthesizer.

Στην περίπτωση που δεν χρησιμοποιηθεί το package `ieee.std_logic_unsigned` για να γίνει η πρόσθεση ή η αφαίρεση ενός αριθμού σε ένα `std_logic_vector` θα πρέπει να γραφούν οι ακόλουθες γραμμές σε VHDL, ώστε να γίνουν οι πράξεις με ακραίους αριθμούς:

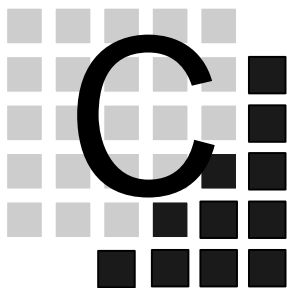
```
use ieee.std_logic_arith.all;
...
signal count: std_logic_vector (2 downto 0);
signal count_int: integer range (0 to 7);
...
count_int <= conv_integer (count);
count_int <= count_int+1;
count <= conv_std_logic_vector(count,3);
...
```

Οι γραμμές αυτές μετατρέπουν το `std_logic_vector` σε ακέραιο αριθμό, ώστε να είναι δυνατή η πρόσθεση του 1. Αφού γίνει η πρόσθεση, μετατρέπεται πάλι ο ακέραιος (με τη νέα τιμή) σε `std_logic_vector`. Η πρακτική αυτή δεν χρησιμοποιείται συνήθως για την πρόσθεση και την αφαίρεση, διότι οδηγεί σε

μακροσκελείς περιγραφές επιρρεπείς στα λάθη και δεν προσφέρει κάτι παραπάνω από την περιγραφή που προκύπτει με τη χρήση του `std_logic_unsigned`.

Ζητούμενα:

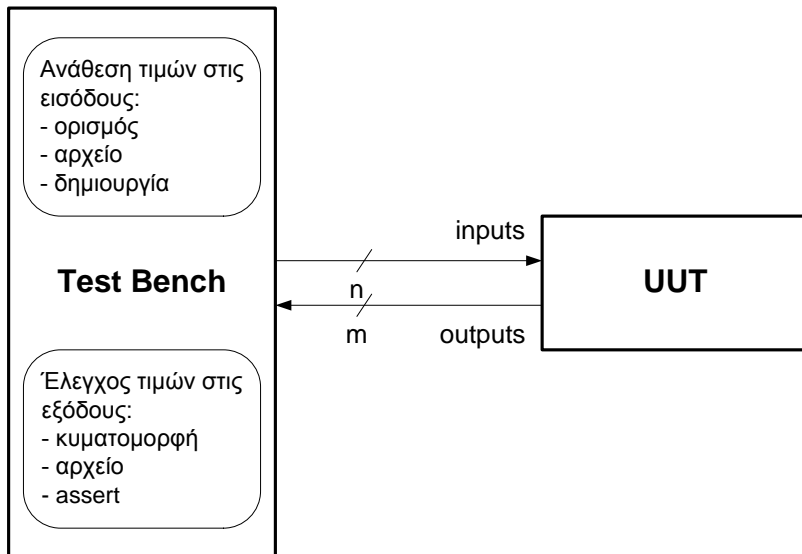
1. Βασιζόμενοι στην περιγραφή του μετρητή των 3 bits, να περιγράψετε έναν μετρητή up/down των 3 bits.
Υπόδειξη: Χρησιμοποιήστε μια είσοδο επιλογής κατεύθυνσης: 1 για μέτρηση προς τα πάνω, 0 για μέτρηση προς τα κάτω και την εντολή case.
2. Βασιζόμενοι στην περιγραφή του μετρητή των 3 bits να περιγράψετε έναν up counter των 3 bits με παράλληλη είσοδο modulo (όριο μέτρησης) των 3 bits.
Υπόδειξη: Χρησιμοποιήστε μια είσοδο `std_logic_vector` την οποία θα συγκρίνετε με την τιμή του μετρητή.
3. Ελέγξτε την ορθότητα λειτουργίας του κυκλώματος που δίνει ο synthesizer για τα ζητούμενα 1 και 2 και αν θα μπορούσε να σχεδιαστεί “με το χέρι” σε απλούστερη μορφή.



ΕΛΕΓΧΟΣ ΜΟΝΑΔΩΝ – TEST BENCH

C.1 Ορισμός του test bench

Η πρώτη εφαρμογή σχεδίασης κυκλωμάτων στην οποία χρησιμοποιήθηκε η VHDL ήταν η προσομοίωση. Η προσομοίωση για να λειτουργήσει απαιτεί, εκτός από την περιγραφή του κυκλώματος, έναν μηχανισμό ανάθεσης τιμών στα σήματα εισόδου και έναν αντίστοιχο μηχανισμό ελέγχου τιμών στα σήματα εξόδου. Για μικρά κυκλώματα, το περιβάλλον εργασίας του κάθε προγράμματος προσομοίωσης ικανοποιεί αυτή την απαίτηση με διάφορα στοιχεία όπως επιλογές από μενού, κουμπιά, λίστες επιλογής και πλαίσια κειμένου, με τα οποία επιλέγονται είσοδοι και ορίζονται αντίστοιχες τιμές, και ένα παράθυρο παρατήρησης των κυματομορφών εξόδου. Για μεγαλύτερα κυκλώματα και για διαδικασίες που απαιτούν μεγάλο βαθμό αυτοματισμού (όπως συμβαίνει στη βιομηχανία), ο έλεγχος μιας μονάδας VHDL γίνεται με την δημιουργία μιας άλλης μονάδας, χωρίς εξωτερικές εισόδους και εξόδους, η οποία συνδέεται με όλες τις εισόδους και τις εξόδους της μονάδας που θέλουμε να ελέγξουμε και καθορίζει τις τιμές τους ή/και παρατηρεί και καταγράφει τις αποκρίσεις τους. Η μονάδα που κάνει τον έλεγχο λέγεται test bench και η μονάδα που ελέγχεται unit (ή entity) under test ή σε συντομογραφία UUT. Σχηματικά, η διαδικασία αυτή μπορεί να παρασταθεί όπως στο σχήμα 3.1. Οι διαφορετικοί τρόποι ανάθεσης τιμών στις εισόδους και ελέγχου τιμών στις εξόδους, όπως καταγράφονται στο σχήμα, θα παρουσιαστούν αναλυτικά στη συνέχεια.



Σχ. 3.1: Σχηματική παράσταση της λειτουργίας του test bench.

Ο γενικός σκελετός ενός test bench είναι ο ακόλουθος. Στη συνέχεια αυτής της εισαγωγής θα δοθούν χαρακτηριστικά παραδείγματα.

```
entity test_bench is
```

```
end test_bench;
```

```
architecture behv of test_bench is
```

```
    component UUT
```

```
        port (...);
```

```
    end component;
```

```
    Δηλώσεις τοπικών σημάτων και μεταβλητών;
```

```
begin
```

```
    UUT_INST: UUT port map (...);
```

```
    Εντολές δημιουργίας κυματομορφών εισόδου;
```

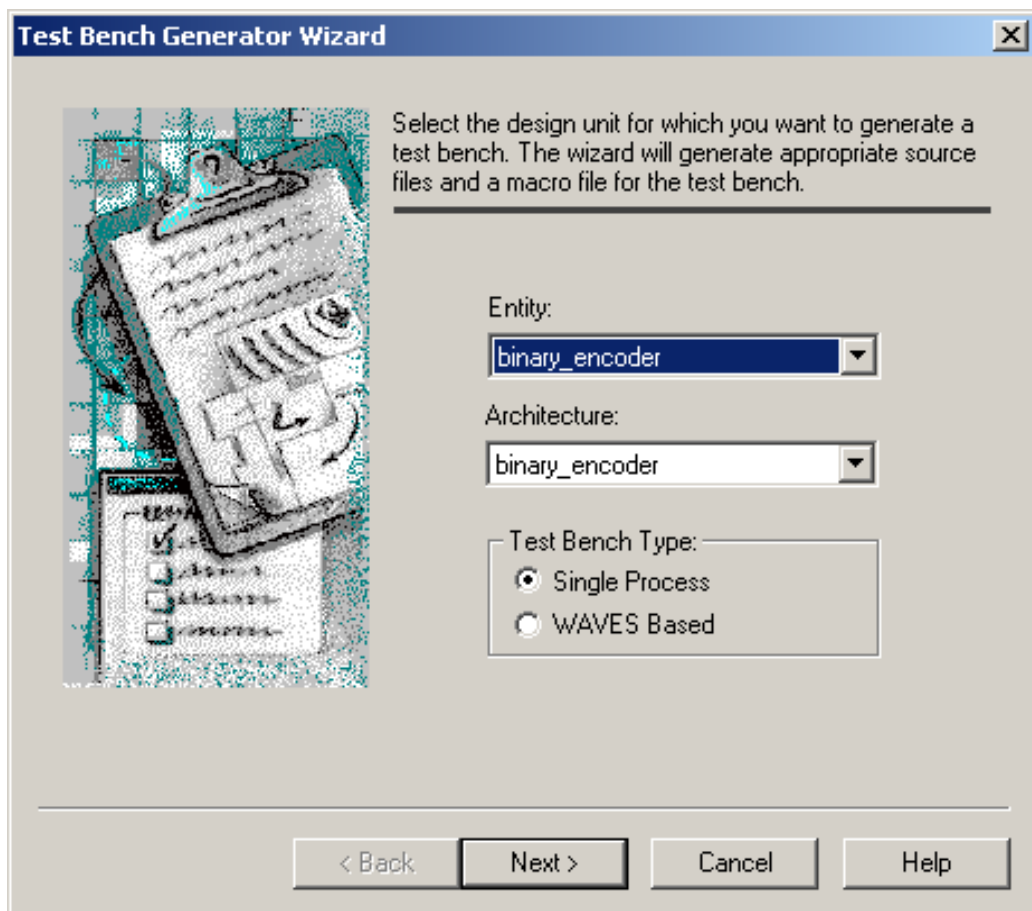
```
    Εντολές σύνδεσης κυματομορφών εισόδου με τη μονάδα UUT_INST;
```

```
    Εντολές παρατήρησης και καταγραφής κυματομορφών εξόδου;
```

```
end behv;
```

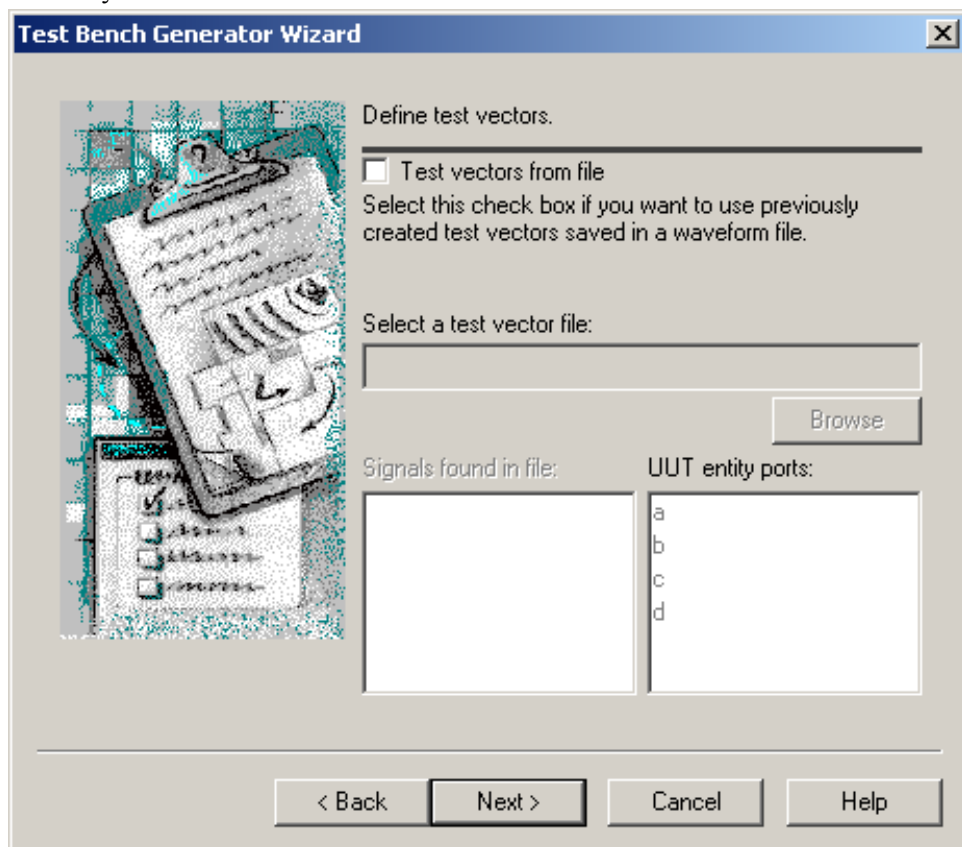
C.2 Αυτόματη δημιουργία test bench με το Active HDL

Στο περιβάλλον Active HDL, η δημιουργία test bench μπορεί να γίνει αυτόματα με τον “Test Bench Generator Wizard” (Menu : Tools/Generate Test Bench). Ας συνεχίσουμε το παράδειγμα του κωδικοποιητή (binary_encoder) της εισαγωγή. Στο πρώτο παράθυρο του “Test Bench Generator Wizard” (σχήμα 3.2) επιλέγουμε το όνομα της οντότητας και της αρχιτεκτονικής για τις οποίες θα δημιουργηθεί το test bench, καθώς και αν θα είναι απλό ή θα ακολουθεί το πρότυπο WAVES του IEEE. Για λόγους απλότητας επιλέγουμε single και προχωράμε στο επόμενο παράθυρο πατώντας “Next”.



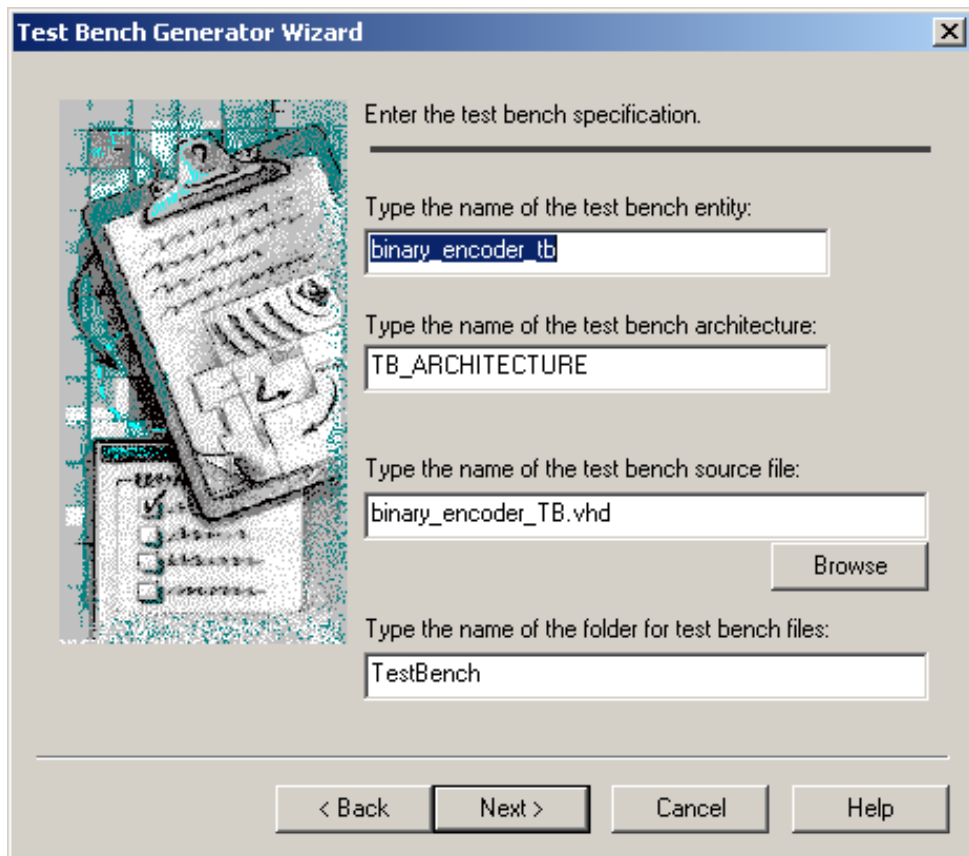
Σχ. 3.2: Test Bench Generator Wizard – παράθυρο 1

Στο επόμενο παράθυρο (σχήμα 3.3), ο “Test Bench Generator Wizard” μας επιτρέπει, αν θέλουμε, να δημιουργήσουμε αυτόματα σήματα για τις κυματομορφές εισόδου, σύμφωνα με αρχείο κυματομορφών που έχουμε αποθηκεύσει από προηγούμενη προσομοίωση. Δεν επιλέγουμε το αντίστοιχο πεδίο, και προχωράμε πατώντας “Next”.



Σχ. 3.3: Test Bench Generator Wizard – παράθυρο 2

Το επόμενο παράθυρο (σχήμα 3.4) μας επιτρέπει να επιλέξουμε το όνομα της οντότητας, της αρχιτεκτονικής, του αρχείου και του υποκαταλόγου του test bench. Αφήνουμε τις προεπιλεγμένες τιμές και προχωράμε πατώντας “Next”.



Test Bench Generator Wizard

Enter the test bench specification.

Type the name of the test bench entity:

Type the name of the test bench architecture:

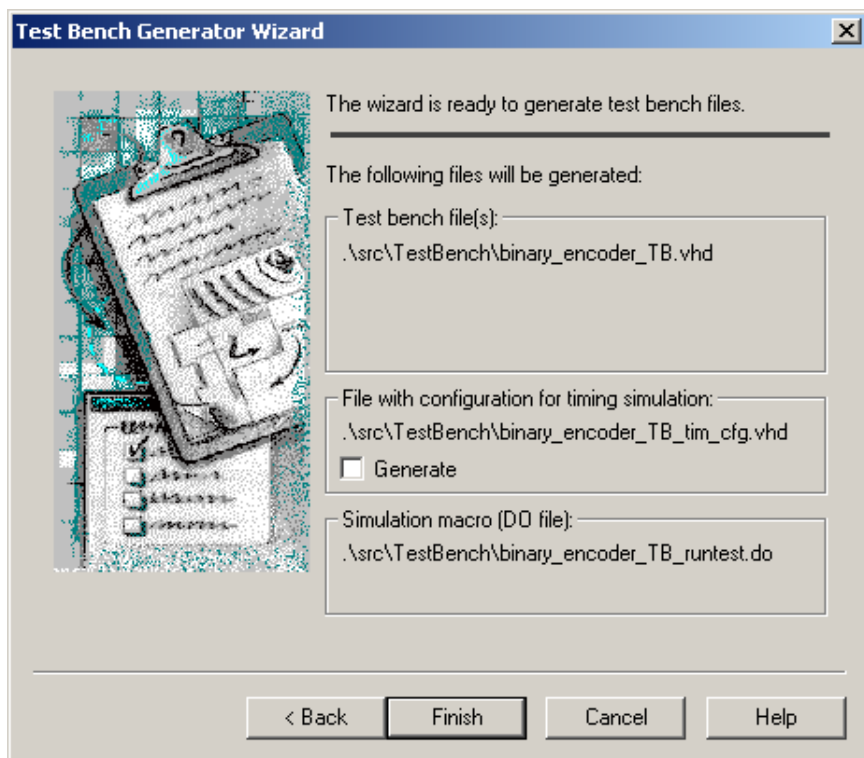
Type the name of the test bench source file:

Type the name of the folder for test bench files:

< Back Next > Cancel Help

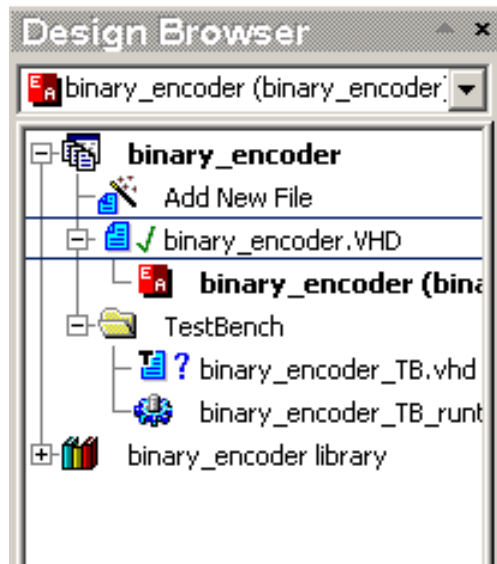
Σχ. 3.4: Test Bench Generator Wizard – παράθυρο 3

Το επόμενο (σχήμα 3.5) είναι το τελευταία παράθυρο του “Test Bench Generator Wizard”. Συνοψίζει τις μέχρι τώρα επιλογές μας και μας επιτρέπει να το συνδυάσουμε με ένα αρχείο χρονικών περιορισμών και σχέσεων μεταξύ των σημάτων του κυκλώματος, για προσομοίωση και έλεγχο της χρονικής αλληλουχίας τους. Για απλά test bench, που ελέγχουν μόνο τη λειτουργία του κυκλώματος, δεν επιλέγουμε το αντίστοιχο πεδίο και τερματίζουμε τη διαδικασία πατώντας “Finish”.



Σχ. 3.5: Test Bench Generator Wizard – παράθυρο 4

Το αποτέλεσμα των ενεργειών αυτών είναι να γίνει εισαγωγή των αντίστοιχων αρχείων στον Design Browser, όπως φαίνεται στο σχήμα 3.6. Στη συνέχεια προσθέτουμε τις κατάλληλες εντολές στα αρχεία και κάνουμε προσομοίωση, όπως στις προηγούμενες περιπτώσεις.



Σχ. 3.6: Design browser μετά τη δημιουργία test bench

Ο σκελετός του κώδικα του test bench που δημιουργήθηκε είναι ο ακόλουθος:

```

_*****
--* This file is automatically generated test bench template *
--* By ACTIVE-VHDL <TBgen v1.10>. Copyright (C) ALDEC Inc. *
--*                                     *
--* This file was generated on:          8:28 pm, 9/3/2002 *
--* Tested entity name:                  binary_encoder *
--* File name contains tested entity:    *
--* $DSN\src\binary_encoder.VHD         *
_*****

library ieee;
use ieee.std_logic_1164.all;

-- Add your library and
-- packages declaration here ...
entity binary_encoder_tb is
end binary_encoder_tb;

```

```
architecture TB_ARCHITECTURE of binary_encoder_tb is
    -- Component declaration of the tested unit
    component binary_encoder
    port(
        a : in std_logic;
        b : in std_logic;
        c : in std_logic;
        d : in std_logic;
        exists : out std_logic;
        encode : out std_logic_vector(1 downto 0) );
    end component;

    -- Stimulus signals – signals mapped
    -- to the input and inout ports of tested entity
    signal a : std_logic;
    signal b : std_logic;
    signal c : std_logic;
    signal d : std_logic;
    -- Observed signals - signals mapped
    -- to the output ports of tested entity
    signal exists : std_logic;
    signal encode : std_logic_vector(1 downto 0);

    -- Add your code here ...
begin

    -- Unit Under Test port map
    UUT : binary_encoder
        port map
            (a => a,
             b => b,
             c => c,
             d => d,
             exists => exists,
             encode => encode );

    -- Add your stimulus here ...
```

```

end TB_ARCHITECTURE;

configuration TESTBENCH_FOR_binary_encoder of binary_encoder_tb is
    for TB_ARCHITECTURE
        for UUT : binary_encoder
            use entity work.binary_encoder(binary_encoder);
        end for;
    end for;
end TESTBENCH_FOR_binary_encoder;

```

Για να γίνει πλήρως λειτουργικό, απαιτείται η προσθήκη κώδικα για την ανάθεση τιμών στα σήματα εισόδου και τον έλεγχο τιμών στα σήματα εξόδου. Τέλος, γίνεται χρήση και της δομής configuration, χωρίς να είναι απαραίτητη.

C.3 Χαρακτηριστικοί τύποι test bench

C.3.1 Test bench με ορισμό εισόδου

Τα test bench της κατηγορίας αυτής είναι τα πιο απλά. Αποτελούνται από εντολές ανάθεσης που καθορίζουν τις τιμές των σημάτων εισόδου. Για παράδειγμα, η παρακάτω διαδικασία ορίζει ένα ρολόι με περίοδο 20 ns.

```

process
begin
    wait for 10 ns;
    clk<='0';
    wait for 10 ns;
    clk<='1';
end process;

```

Παρόμοια, η παρακάτω εντολή ανάθεσης ταυτόχρονης εκτέλεσης ορίζει έναν παλμό αρχικοποίησης.

```

rst<='U' after 5 ns, '1' after 60 ns,
'0' after 75 ns;

```

Εφόσον απαιτείται, μπορούν να χρησιμοποιηθούν και οποιεσδήποτε συνθετότερες δομές της γλώσσας.

Για το παράδειγμα του `binary_encoder`, προσθέτοντας αρχικές τιμές στα σήματα `a`, `b`, `c` και `d` και κατάλληλες εντολές ανάθεσης, μπορούμε να τους αντιστοιχίσουμε τετραγωνικές κυματομορφές περιόδου 50 ns, 10 ns, 200 ns και 400 ns αντίστοιχα, όπως φαίνεται παρακάτω:

```
--*****
--* This file is automatically generated test bench template *
--* By ACTIVE-VHDL <TBgen v1.10>. Copyright (C) ALDEC Inc. *
--*                                     *
--* This file was generated on:          8:28 pm, 9/3/2002 *
--* Tested entity name:                  binary_encoder *
--* File name contains tested entity:    *
--* $DSN\src\binary_encoder.VHD        *
--*****
```

```
library ieee;
```

```
use ieee.std_logic_1164.all;
```

```
-- Add your library and
```

```
-- packages declaration here ...
```

```
entity binary_encoder_tb is
```

```
end binary_encoder_tb;
```

```
architecture TB_ARCHITECTURE of binary_encoder_tb is
```

```
-- Component declaration of the tested unit
```

```
component binary_encoder
```

```
port(
```

```
    a : in std_logic;
```

```
    b : in std_logic;
```

```
    c : in std_logic;
```

```
    d : in std_logic;
```

```
    exists : out std_logic;
```

```
    encode : out std_logic_vector(1 downto 0) );
```

```
end component;
```

```
-- Stimulus signals - signals mapped
```

```
-- to the input and inout ports of tested entity
```

```
signal a : std_logic:=’0’;
```

```
signal b : std_logic:=’0’;
```

```

    signal c : std_logic:=’0’;
    signal d : std_logic:=’0’;
    -- Observed signals - signals mapped
    -- to the output ports of tested entity
    signal exists : std_logic;
    signal encode : std_logic_vector(1 downto 0);

    -- Add your code here ...

begin
    -- Unit Under Test port map
    UUT : binary_encoder
        port map
            (a => a,
             b => b,
             c => c,
             d => d,
             exists => exists,
             encode => encode );

    -- Add your stimulus here ...

    a<=not a after 25 ns;
    b<=not b after 50 ns;
    c<=not c after 100 ns;
    d<=not d after 200 ns;
end TB_ARCHITECTURE;

configuration TESTBENCH_FOR_binary_encoder of binary_encoder_tb is
    for TB_ARCHITECTURE
        for UUT : binary_encoder
            use entity work.binary_encoder(binary_encoder);
            end for;
        end for;
    end TESTBENCH_FOR_binary_encoder;

```

C.3.2 Test bench με αρχείο εισόδου ή/και εξόδου

Τα test bench αυτού του τύπου βασίζονται στις συναρτήσεις χειρισμού αρχείων που περιγράφει και υλοποιεί το πακέτο TEXTIO της βιβλιοθήκης STD. Αρχικά λοιπόν πρέπει να αναφερθούμε στο πακέτο (και όχι στη βιβλιοθήκη, γιατί η βιβλιοθήκη STD θεωρείται προσπελάσιμη χωρίς να δηλωθεί, από όποια μονάδα τη χρειάζεται) με την εντολή:

```
use STD.TEXTIO.all;
```

Στη συνέχεια, στην αρχή της αρχιτεκτονικής του test bench, δηλώνουμε τα αρχεία που θα χρησιμοποιήσουμε για είσοδο δεδομένων ή/και για αποθήκευση αποτελεσμάτων με την εντολή:

```
file VECTORS: TEXT open READ_MODE is “inputs.txt” ή  
file VECTORS: TEXT open WRITE_MODE is “outputs.txt”
```

Η λειτουργία του test bench ορίζεται από μία διεργασία, η οποία με ένα βρόχο διαβάζει συνεχώς εισόδους, τις αναθέτει στα σήματα εισόδου της μονάδας UUT και αποθηκεύει ή συγκρίνει τις τιμές των εξόδων που προκύπτουν. Ο σκελετός ενός τέτοιου test bench είναι ο ακόλουθος:

```
use STD.TEXTIO.all;  
  
entity test_bench is  
end test_bench;  
  
architecture behv of test_bench is  
    file IN_VECTORS: TEXT open READ_MODE is “inputs.txt”;  
    file OUT_VECTORS: TEXT open WRITE_MODE is “outputs.txt”;  
    component UUT  
        port (...);  
    end component;  
    Δηλώσεις τοπικών σημάτων και μεταβλητών;  
begin  
    process  
        variable IN_BUF: LINE;  
        variable OUT_BUF: LINE;
```



```

        Δηλώσεις μεταβλητών ιδίου τύπου με τις εισόδους
        του UUT, οι οποίες θα πάρουν τιμές από τη
        γραμμή εισόδου BUF με κλήσεις της συνάρτησης
        READ (βλέπε παρακάτω)
begin
        while not ENDFILE(IN_VECTORS) loop
            READLINE(IN_VECTORS,IN_BUF);
            -- Διάβασε από τη γραμμή IN_BUF τις τιμές
            -- των επιθυμητών εισόδων και αποθήκευσέ
            -- τις σε τοπικές μεταβλητές
            READ(IN_BUF,var1);
            READ(IN_BUF,var2);
            ...
            -- Ανέθεσε τις τοπικές μεταβλητές στα
            -- σήματα εισόδου
            input1<=var1;
            input2<=var2;
            ...
            -- Περίμενε να σταθεροποιηθούν τα σήματα
            wait for ACTIVATION_PERIOD;
            -- Η εντολή WRITE, σε αντίθεση με την
            -- READ, δέχεται για παραμέτρους και
            -- σήματα (και όχι μόνο μεταβλητές)
            WRITE(OUT_BUF,STRING'("Output1= "));
            WRITE(OUT_BUF,output1);
            ...
            WRITELINE(OUT_VECTORS,OUT_BUF);
        end loop;
        wait;
    end process;
    UUT_INST: UUT port map (...);
end behv;

```

Για το παράδειγμα του `binary_encoder`, το παρακάτω test bench διαβάζει εισόδους από το αρχείο `inputs.txt` και αποθηκεύει τις αντίστοιχες εξόδους στο αρχείο `outputs.txt`. Το αρχείο `inputs.txt` περιέχει σειρές από 0 και 1, είτε συνεχόμενα είτε χωρισμένα με κενά. Το αρχείο `outputs.txt` μορφοποιείται όπως θέλουμε, με τις εντολές `WRITE`, στην κεντρική διεργασία του test bench. Οι εντολές `READ` και `WRITE` χειρίζονται προκαθορισμένους τύπους, οπότε είναι απαραίτητη η χρήση των

εντολών μετατροπής to_stdlogic, to_bit και to_bitvector. Ακόμα η καθυστέρηση των 5 ns (η συγκεκριμένη τιμή είναι αυθαίρετη) είναι απαραίτητη για να σταθεροποιηθούν τα σήματα εξόδου πριν γίνει δειγματοληψία και αποθήκευση των εξόδων.

```

--*****
--* This file is automatically generated test bench template *
--* By ACTIVE-VHDL <TBgen v1.10>. Copyright (C) ALDEC Inc. *
--*                                     *
--* This file was generated on:          8:28 pm, 9/3/2002 *
--* Tested entity name:                  binary_encoder *
--* File name contains tested entity:    *
--* $DSN\src\binary_encoder.VHD          *
--*****

library ieee;
use ieee.std_logic_1164.all;
use STD.TEXTIO.all;

-- Add your library and
-- packages declaration here ...

entity binary_encoder_tb is
end binary_encoder_tb;

architecture TB_ARCHITECTURE of binary_encoder_tb is
    file IN_VECTORS: TEXT open READ_MODE is "inputs.txt";
    file OUT_VECTORS: TEXT open WRITE_MODE is "outputs.txt";
    -- Component declaration of the tested unit
    component binary_encoder
    port(
        a : in std_logic;
        b : in std_logic;
        c : in std_logic;
        d : in std_logic;
        exists : out std_logic;
        encode : out std_logic_vector(1 downto 0) );
    end component;

```

```

-- Stimulus signals - signals mapped
-- to the input and inout ports of tested entity
signal a : std_logic;
signal b : std_logic;
signal c : std_logic;
signal d : std_logic;
-- Observed signals - signals mapped
-- to the output ports of tested entity
signal exists : std_logic;
signal encode : std_logic_vector(1 downto 0);

-- Add your code here ...

begin

-- Unit Under Test port map
UUT : binary_encoder
    port map
        (a => a,
         b => b,
         c => c,
         d => d,
         exists => exists,
         encode => encode );

-- Add your stimulus here ...
process
    variable IN_BUF: LINE;
    variable OUT_BUF: LINE;
    variable a_var,b_var,c_var,d_var : bit;
begin
    while not ENDFILE(IN_VECTORS) loop
        READLINE(IN_VECTORS,IN_BUF);
        READ(IN_BUF,a_var);
        READ(IN_BUF,b_var);
        READ(IN_BUF,c_var);
        READ(IN_BUF,d_var);
        a<=to_stdulogic(a_var);
        b<=to_stdulogic(b_var);
        c<=to_stdulogic(c_var);

```

```

        d<=to_stdlogic(d_var);
        wait for 5 ns;
        WRITE(OUT_BUF,STRING'("Exists= "));
        WRITE(OUT_BUF,to_bit(exists));
        WRITE(OUT_BUF,STRING'(", Encode= "));
        WRITE(OUT_BUF,to_bitvector(encode));
        WRITELINE(OUT_VECTORS,OUT_BUF);

    end loop;
    wait;
end process;
end TB_ARCHITECTURE;

configuration TESTBENCH_FOR_binary_encoder of binary_encoder_tb is
    for TB_ARCHITECTURE
        for UUT : binary_encoder
            use entity work.binary_encoder(binary_encoder);
        end for;
    end for;
end TESTBENCH_FOR_binary_encoder;

```

C.3.3 Test bench που δημιουργεί εισόδους και εξόδους

Μια τρίτη κατηγορία test bench είναι αυτή που περιγράφει αλγοριθμικά τη λειτουργία του κυκλώματος UUT και συγκρίνει τα αποτελέσματα. Τυχόν ασυμφωνίες ελέγχονται και αναφέρονται με την εντολή assert. Ο σκελετός ενός τέτοιου test bench είναι ο ακόλουθος:

```

entity test_bench is
end test_bench;

architecture behv of test_bench is
    component UUT
        port (...);
    end component;
    Δηλώσεις τοπικών σημάτων και μεταβλητών (in1,in2,out1);
begin
    process
        function behavior (in1, in2: integer) return integer is
            begin
                return συνάρτηση των in1 και in2;
            end behavior;
    end process;
end architecture behv;

```

```

        end behavior;
    begin
        for in1 in x1 to y1 loop
            for in2 in x2 to y2 loop
                assert(out1=behavior(in1,in2))
                    report "Simulation error!";
            end loop;
        end loop;
    wait;
end process;
UUT_INST: UUT port map (in1,in2,out1);
end behv;

```

Για το παράδειγμα του `binary_encoder`, το παρακάτω test bench διαβάζει δημιουργεί όλους τους δυνατούς συνδυασμούς των εισόδων `a`, `b`, `c`, `d`, χρησιμοποιώντας φωλιασμένους βρόχους, και ελέγχει τις σωστές εξόδους χρησιμοποιώντας δύο συναρτήσεις, που υλοποιούν την επιθυμητή τους συμπεριφορά. Ο έλεγχος γίνεται με την εντολή `assert` και τα αντίστοιχα διαγνωστικά μηνύματα δίνονται με την εντολή `report`. Παρατηρείστε ότι οι εντολές βρόχου δημιουργούν μεταβλητές ελέγχου ροής (`a1`, `b1`, `c1`, `d1`) χωρίς να προηγείται σχετική δήλωση. Ακόμα, χρησιμοποιείται πάλι μια αυθαίρετη καθυστέρηση ενώ η τελική εντολή `wait` αποτρέπει την συνεχή λειτουργία του test bench.

```

--*****
--* This file is automatically generated test bench template *
--* By ACTIVE-VHDL <TBgen v1.10>. Copyright (C) ALDEC Inc. *
--*
--* This file was generated on:      8:28 pm, 9/3/2002 *
--* Tested entity name:             binary_encoder *
--* File name contains tested entity:
--* $DSN\src\binary_encoder.VHD
--*****

library ieee;
use ieee.std_logic_1164.all;

-- Add your library and
-- packages declaration here ...

```

```
entity binary_encoder_tb is
end binary_encoder_tb;
architecture TB_ARCHITECTURE of binary_encoder_tb is
    -- Component declaration of the tested unit
    component binary_encoder
    port(
        a : in std_logic;
        b : in std_logic;
        c : in std_logic;
        d : in std_logic;
        exists : out std_logic;
        encode : out std_logic_vector(1 downto 0) );
    end component;

    -- Stimulus signals - signals mapped
    -- to the input and inout ports of tested entity
    signal a : std_logic;
    signal b : std_logic;
    signal c : std_logic;
    signal d : std_logic;

    -- Observed signals -signals mapped
    -- to the output ports of tested entity
    signal exists : std_logic;
    signal encode : std_logic_vector(1 downto 0);

    -- Add your code here ...

begin
    -- Unit Under Test port map
    UUT : binary_encoder
        port map
            (a => a,
             b => b,
             c => c,
             d => d,
             exists => exists,
             encode => encode );

    -- Add your stimulus here ...
```

```
process
    function exists_behavior (a,b,c,d: std_logic)
    return std_logic is
    begin
        if (a='1') or (b='1') or (c='1') or (d='1') then
            return '1';
        else
            return '0';
        end if;
    end exists_behavior;
    function encode_behavior (a,b,c,d: std_logic)
    return std_logic_vector is
    begin
        if (a='1') then
            return "11";
        elsif (b='1') then
            return "10";
        elsif (c='1') then
            return "01";
        else
            return "00";
        end if;
    end encode_behavior;
begin
    for a1 in std_logic('0') to std_logic('1') loop
        for b1 in std_logic('0') to std_logic('1') loop
            for c1 in std_logic('0') to std_logic('1') loop
                for d1 in std_logic('0') to std_logic('1') loop
                    a<=a1;
                    b<=b1;
                    c<=c1;
                    d<=d1;
                    wait for 5 ns;
                    assert(exists=exists_behavior(a,b,c,d))
                        report "Error on signal exists!";
                    assert(encode=encode_behavior(a,b,c,d))
                        report "Error on signal encode!";
                end loop;
            end loop;
        end loop;
    end loop;
```

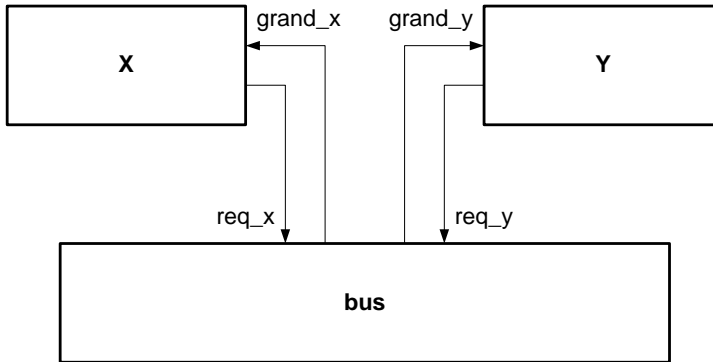
```
        end loop;  
    end loop;  
    wait;  
end process;  
end TB_ARCHITECTURE;  
  
configuration TESTBENCH_FOR_binary_encoder of binary_encoder_tb is  
    for TB_ARCHITECTURE  
        for UUT : binary_encoder  
            use entity work.binary_encoder(binary_encoder);  
        end for;  
    end for;  
end TESTBENCH_FOR_binary_encoder;
```

C.4 Άσκηση 3

Σχεδίαση και εξομοίωση Αλγοριθμικών Μηχανών Καταστάσεων (AMK)

Θέμα C.1: Κύκλωμα διαιτησίας διαδρόμου

Οι Ακολουθιακές Μηχανές Καταστάσεων (AMK) είναι ακολουθιακά κυκλώματα που αποτελούνται από ένα σύνολο διακριτών καταστάσεων και από ένα σύνολο μεταβάσεων. Η κάθε μετάβαση συνοδεύεται από έναν ή περισσότερους συνδυασμούς συνθηκών, σχετικά με τις τιμές των εισόδων, οι οποίες όταν ισχύουν ενεργοποιούν τη μετάβαση. Το κύκλωμα ξεκινάει από την αρχική του κατάσταση και με κάθε παλμό του ρολογιού, ανάλογα με τις τιμές των συνθηκών και την τρέχουσα κατάσταση, εκτελεί συνεχώς μεταβάσεις. Σε περίπτωση που υπάρχει τελική κατάσταση, το κύκλωμα σταματάει τη λειτουργία του όταν φτάσει σε αυτήν. Η διαδοχή των καταστάσεων παράγει και τιμές εξόδου. Εάν οι τιμές των εξόδων καθορίζονται μόνο από την τρέχουσα κατάσταση η αλγοριθμική μηχανή ονομάζεται Moore ενώ αν καθορίζονται από την τρέχουσα κατάσταση και από τις τιμές των εισόδων ονομάζεται Mealy.



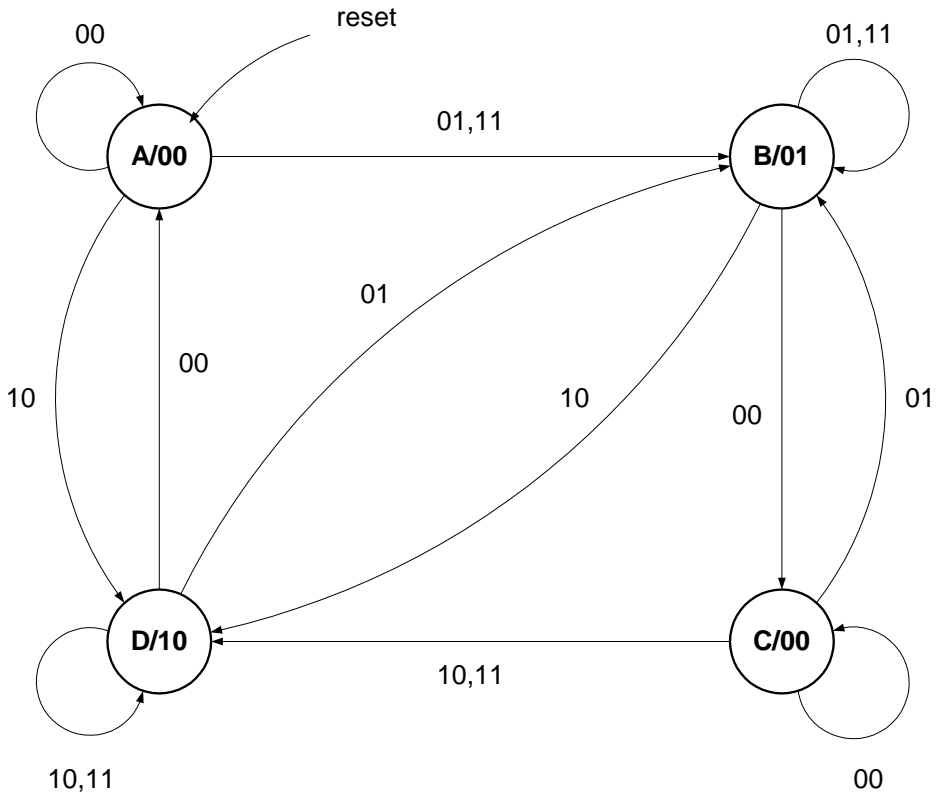
Σχ. 3.7: Κύκλωμα διαιτησίας διαδρόμου

Για παράδειγμα, στο σχήμα 3.7 εικονίζεται ένα παράδειγμα AMK, ένα κύκλωμα διαιτησίας διαδρόμου. Θεωρούμε ότι δύο μονάδες, X και Y, συνδέονται σε κοινό διάδρομο. Κάθε μια έχει ένα σήμα εξόδου req, το οποίο θέτει σε λογικό '1' όταν θέλει να χρησιμοποιήσει το διάδρομο και ένα σήμα εισόδου grand, το οποίο πρέπει να τεθεί σε λογικό '1' για να αρχίσει η μονάδα να χρησιμοποιεί το διάδρομο. Κάθε μονάδα, για να χρησιμοποιήσει το διάδρομο, θέτει το σήμα req της σε λογικό '1', περιμένει από το διάδρομο να της απαντήσει θέτοντας το αντίστοιχο grand σε λογικό '1', κάνει χρήση του διαδρόμου για όσο χρόνο χρειάζεται, και όταν τελειώσει θέτει το req σε λογικό '0'. Ο διάδρομος, όταν είναι ελεύθερος και δεχθεί αίτηση από μια από τις μονάδες, γίνεται διαθέσιμος θέτοντας το αντίστοιχο grand '1'. Αν δεχθεί ταυτόχρονα αίτηση και από τις δύο μονάδες επιτρέπει τη χρήση σε αυτήν που έχει περισσότερο χρόνο να την χρησιμοποιήσει. Για τη λειτουργία αυτή απαιτούνται τέσσερις καταστάσεις, οι ακόλουθες:

- A – ο διάδρομος είναι ελεύθερος και η τελευταία μονάδα που τον χρησιμοποίησε ήταν η Y, ή αρχική κατάσταση.
- B – ο διάδρομος χρησιμοποιείται από τη μονάδα X.
- C – ο διάδρομος είναι ελεύθερος και η τελευταία μονάδα που τον χρησιμοποίησε ήταν η X.
- D – ο διάδρομος χρησιμοποιείται από τη μονάδα Y.

Με βάση αυτά, η λειτουργία του κυκλώματος διαιτησίας περιγράφεται με το διάγραμμα του σχήματος 3.8, όπου οι δυαδικές τιμές που βρίσκονται μέσα στις καταστάσεις είναι οι τιμές εξόδου (μηχανή τύπου Moore), και οι τιμές που βρίσκονται πάνω στις μεταβάσεις είναι οι συνθήκες εισόδου. Και στις δύο

περιπτώσεις το λιγότερο σημαντικό bit αφορά τη μονάδα X και το περισσότερο τη μονάδα Y.



Σχ. 3.8: AMK κυκλώματος διαίτησίας διαδρόμου

Η παραπάνω AMK περιγράφεται σε VHDL με την ακόλουθη αρχιτεκτονική, που αποτελείται από δύο διεργασίες, μια που αντιστοιχεί στο συνδυαστικό κύκλωμα υπολογισμού της επόμενης κατάστασης (process comb) και μια που αντιστοιχεί στο ακολουθιακό κύκλωμα αποθήκευσης της τρέχουσας κατάστασης (process seq), και δύο εντολές ανάθεσης ταυτόχρονης εκτέλεσης, μία για τον υπολογισμό κάθε εξόδου. Με τη βοήθεια του Active HDL, προσομοιώστε την αρχιτεκτονική και παρατηρήστε την ορθή εναλλαγή των καταστάσεων.

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

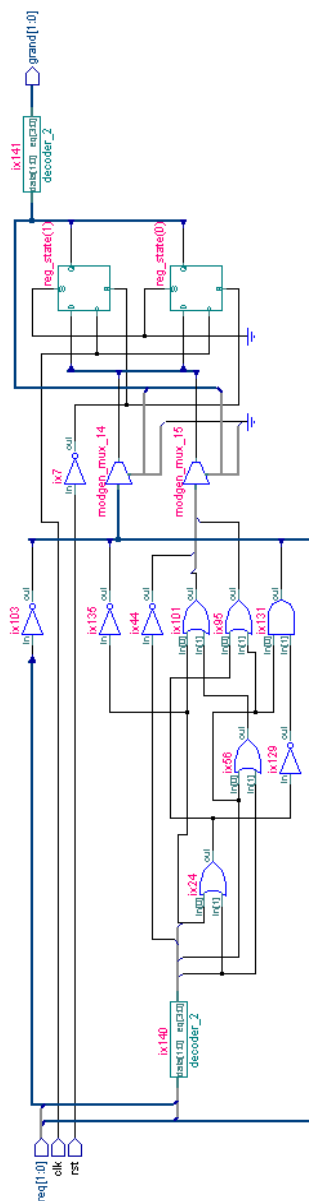
entity bus_arbiter1 is
    port (
        clk,rst: in std_logic;
        req: in std_logic_vector(1 downto 0);
        grand: out std_logic_vector(1 downto 0));
end bus_arbiter1;

architecture rtl of bus_arbiter1 is
    type STATES is (A,B,C,D);
    signal state,next_state: STATES;
begin
    seq: process (clk,rst)
    begin
        if (rst='0') then
            state<=A;
        elsif (clk'event and clk='1') then
            state<=next_state;
        end if;
    end process;

    cmb: process (req,state)
    begin
        next_state<=state;
        case state is
            when A =>
                if (req="01" or req="11") then
                    next_state<=B;
                elsif (req="10") then
                    next_state<=D;
                end if;
            when B =>
                if (req="00") then
                    next_state<=C;
                elsif (req="10") then
                    next_state<=D;
                end if;
        end case;
    end process;
end rtl;
```

```
        when C =>
            if (req="01") then
                next_state<=B;
            elsif (req="10" or req="11") then
                next_state<=D;
            end if;
        when D =>
            if (req="01") then
                next_state<=B;
            elsif (req="00") then
                next_state<=A;
            end if;
        when others => null;
    end case;
end process;
grand(0)<='1' when state=B else '0';
grand(1)<='1' when state=D else '0';
end rtl;
```

Το κύκλωμα που προκύπτει από τον synthesizer για την περιγραφή αυτή είναι αυτό που φαίνεται στο σχήμα 3.9. Φαίνονται τα 2 flip-flops που χρησιμεύουν για την αποθήκευση των δύο bit κατάστασης μαζί με το συνδυαστικό κύκλωμα που καθορίζει την επόμενη κατάσταση και τις εξόδους.



Σχ. 3.9: Διατητής διαδρόμου – κύκλωμα που προκύπτει από τον synthesizer

Εναλλακτικά, το ίδιο κύκλωμα μπορεί να περιγραφεί με την ακόλουθη αρχιτεκτονική, στην οποία το συνδυαστικό και το ακολουθιακό κομμάτι της AMK ενώνονται σε μία διεργασία (process fsm), ενώ ο υπολογισμός των εξόδων παραμένει ίδιος. Με τη βοήθεια του Active HDL, προσομοιώστε την αρχιτεκτονική και παρατηρήστε αν είναι λειτουργικά ισοδύναμη με την πρώτη.

```
library IEEE;

use IEEE.std_logic_1164.all;

entity bus_arbiter2 is
    port (
        clk,rst: in std_logic;
        req: in std_logic_vector(1 downto 0);
        grand: out std_logic_vector(1 downto 0));
end bus_arbiter2;

architecture rtl of bus_arbiter2 is
    type STATES is (A,B,C,D);
    signal state: STATES;
begin
    fsm: process (clk,rst,req,state)
    begin
        if (rst='0') then
            state<=A;
        elsif (clk'event and clk='1') then
            case state is
                when A =>
                    if (req="01" or req="11") then
                        state<=B;
                    elsif (req="10") then
                        state<=D;
                    end if;
                when B =>
                    if (req="00") then
                        state<=C;
                    elsif (req="10") then
                        state<=D;
                    end if;
            end case;
        end if;
    end process;
end rtl;
```

```

                                end if;
                        when C =>
                                if (req="01") then
                                        state<=B;
                                elsif (req="10" or req="11") then
                                        state<=D;
                                end if;
                        when D =>
                                if (req="01") then
                                        state<=B;
                                elsif (req="00") then
                                        state<=A;
                                end if;
                        when others => null;
                end case;
        end if;
end process;
grand(0)<='1' when state=B else '0';
grand(1)<='1' when state=D else '0';
end rtl;

```

Ζητούμενα:

1. Να κάνετε προσομοίωση και σύνθεση των δύο περιγραφών του κυκλώματος διαιτησίας διαδρόμου. Συγκρίνετε τα αποτελέσματα.

Παρατήρηση: Να σχολιάσετε τη λίστα ευαισθησίας της δεύτερης αρχιτεκτονικής σε συσχετισμό με αυτήν της πρώτης. Είναι απαραίτητο να περιλαμβάνει όλα τα σήματα εισόδου;

Θέμα C.2: Κύκλωμα υπολογισμού Μέγιστου Κοινού Διαιρέτη (ΜΚΔ) δύο ακεραίων

Οι αλγοριθμικές μηχανές καταστάσεων χρησιμοποιούνται πολύ συχνά για να υλοποιήσουν τη μονάδα ελέγχου μικροϋπολογιστών και γενικότερα υπολογιστικών κυκλωμάτων. Για παράδειγμα, ας υποθέσουμε ότι έχουμε να υλοποιήσουμε με κύκλωμα τον αλγόριθμο υπολογισμού του Μέγιστου Κοινού Διαιρέτη (ΜΚΔ) δύο ακεραίων. Αν x και y είναι οι ακέραιοι, τότε ένας απλός αλγόριθμος σε γλώσσα C, που βασίζεται στον αλγόριθμο του Ευκλείδη και περιλαμβάνει μόνο αφαιρέσεις, είναι ο ακόλουθος:

```
int gcd(int x, int y)
{
    while (x!=y)
    {
        if (x>y)
            x=x-y;
        else if (y>x)
            y=y-x;
        }
    return x;
}
```

Για την καλύτερη κατανόηση του αλγορίθμου δίνονται τα ακόλουθα παραδείγματα:

Παράδειγμα 1: ΜΚΔ των αριθμών 117 και 39

Βήμα 1: $X=117, Y=39, X>Y \Rightarrow X=X-Y=117-39=78$

Βήμα 2: $X=78, Y=39, X>Y \Rightarrow X=X-Y=78-39=39$

Βήμα 3: $X=39, Y=39, X=Y \Rightarrow$ ο ΜΚΔ των 117 και 39 είναι ο 39

Παράδειγμα 2: ΜΚΔ των αριθμών 95 και 25

Βήμα 1: $X=95, Y=25, X>Y \Rightarrow X=X-Y=95-25=70$

Βήμα 2: $X=70, Y=25, X>Y \Rightarrow X=X-Y=70-25=45$

Βήμα 3: $X=45, Y=25, X>Y \Rightarrow X=X-Y=45-25=20$

Βήμα 4: $X=20, Y=25, X<Y \Rightarrow Y=Y-X=25-20=5$

Βήμα 5: $X=20, Y=5, X>Y \Rightarrow X=X-Y=20-5=15$

Βήμα 6: $X=15, Y=5, X>Y \Rightarrow X=X-Y=15-5=10$

Βήμα 7: $X=10, Y=5, X>Y \Rightarrow X=X-Y=10-5=5$

Βήμα 8: $X=5, Y=5, X=Y \Rightarrow$ ο ΜΚΔ των 95 και 25 είναι ο 5

Παράδειγμα 3: ΜΚΔ των αριθμών 17 και 10

Βήμα 1: $X=17, Y=10, X>Y \Rightarrow X=X-Y=17-10=7$

Βήμα 2: $X=7, Y=10, X<Y \Rightarrow Y=Y-X=10-7=3$

Βήμα 3: $X=7, Y=3, X>Y \Rightarrow X=X-Y=7-3=4$

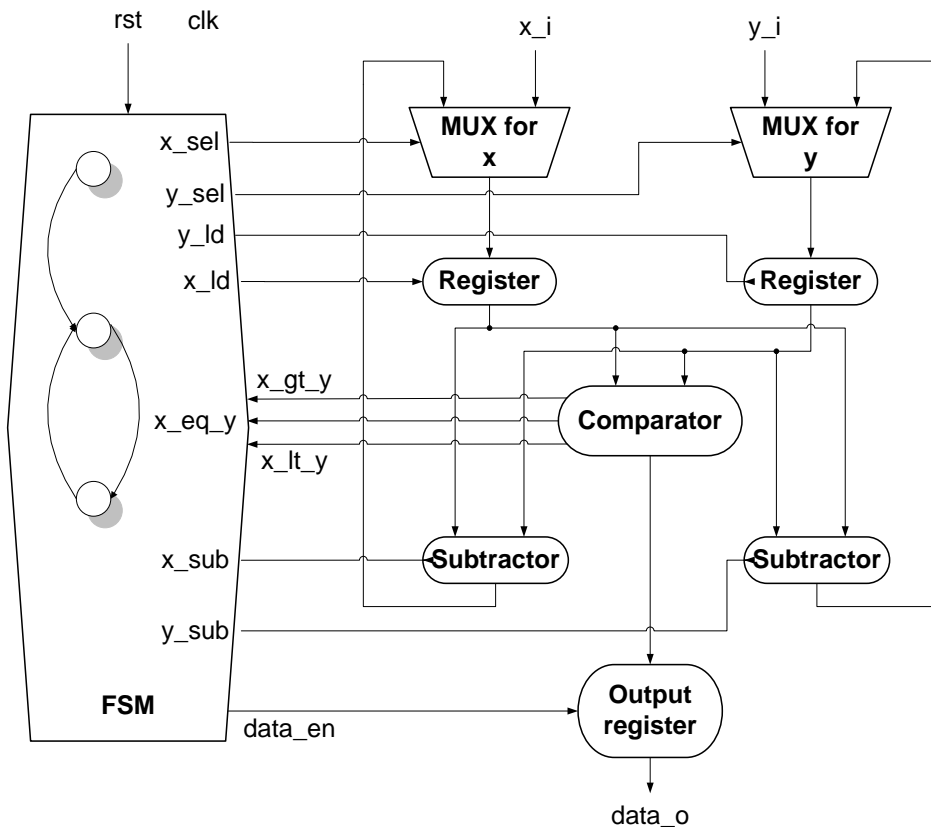
Βήμα 4: $X=4, Y=3, X>Y \Rightarrow X=X-Y=4-3=1$

Βήμα 5: $X=1, Y=3, X<Y \Rightarrow Y=Y-X=3-1=2$

Βήμα 5: $X=1, Y=2, X<Y \Rightarrow Y=Y-X=2-1=1$

Βήμα 6: $X=1, Y=1, X=Y \Rightarrow$ ο ΜΚΔ των 17 και 10 είναι ο 1 ή δεν υπάρχει ΜΚΔ

Η υλοποίηση ενός τέτοιου κυκλώματος μπορεί να γίνει με πολλούς τρόπους. Για λόγους απλότητας, ας υποθέσουμε ότι η αρχιτεκτονική υλοποίησης είναι δεδομένη και παρουσιάζεται στο σχήμα 3.10. Στη δεξιά πλευρά του σχήματος, εικονίζονται οι δομικές μονάδες που αποτελούν τον τμήμα της αρχιτεκτονικής που είναι υπεύθυνο για την εκτέλεση αριθμητικών πράξεων (μονοπάτι δεδομένων), ενώ στο αριστερό εικονίζεται μία ΑΜΚ, που είναι υπεύθυνη για το συγχρονισμό των διαφόρων μονάδων και την κυκλωματική υλοποίηση του βρόχου while (μονοπάτι ελέγχου).



Σχ. 3.10: Αρχιτεκτονική κυκλώματος υπολογισμού ΜΚΔ

Το μονοπάτι δεδομένων αποτελείται από πολυπλέκτες εισόδου, οι οποίοι επιλέγουν είτε τις αρχικές εισόδους είτε το αποτέλεσμα των διαδοχικών μεταξύ τους αφαιρέσεων, καταχωρητές που κρατάνε σταθερά τα δεδομένα εισόδου μέχρι να τελειώσει και η μεταξύ τους σύγκριση αλλά και οι αφαιρέσεις που προκύπτουν, συγκριτή και δύο αφαιρέτες που εκτελούν τις αριθμητικές πράξεις και τέλος, έναν καταχωρητή εξόδου που κρατάει το τελικό αποτέλεσμα του υπολογισμού. Απλές, behavioral περιγραφές όλων αυτών των στοιχείων σε VHDL είναι οι ακόλουθες:

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

entity mux8_2x1 is
    port (
        sel: in std_logic;
        inp_a,inp_b: in std_logic_vector(7 downto 0);
        mout: out std_logic_vector(7 downto 0));
end mux8_2x1;

architecture rtl of mux8_2x1 is
begin
    seq: process (sel,inp_a,inp_b)
    begin
        case sel is
            when '0' => mout<=inp_a;
            when others => mout<=inp_b;
        end case;
    end process;
end rtl;

library IEEE;
use IEEE.std_logic_1164.all;

entity reg8 is
    port (
        en,clk: in std_logic;
        inp: in std_logic_vector(7 downto 0);
        outp: out std_logic_vector(7 downto 0));
```

```
end reg8;

architecture rtl of reg8 is
begin
    seq: process (en,clk,inp)
    begin
        if en='1' then
            if (clk'event and clk='1') then
                outp<=inp;
            end if;
        end if;
    end process;
end rtl;

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

entity cmp8 is
    port (
        inp_a,inp_b: in std_logic_vector(7 downto 0);
        a_gt_b,a_eq_b,a_lt_b: out std_logic;
        outp: out std_logic_vector(7 downto 0));
end cmp8;

architecture behv of cmp8 is
begin
    cmb: process(inp_a,inp_b)
        variable a,b: integer;
    begin
        a:=conv_integer(inp_a);
        b:=conv_integer(inp_b);
        if (a>b) then
            a_gt_b<='1';
            a_eq_b<='0';
            a_lt_b<='0';
        elsif (a=b) then
            a_gt_b<='0';
            a_eq_b<='1';
```

```

                                a_lt_b<='0';
                                else
                                a_gt_b<='0';
                                a_eq_b<='0';
                                a_lt_b<='1';
                                end if;
                                end process;
                                outp<=inp_a;
end behv;

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

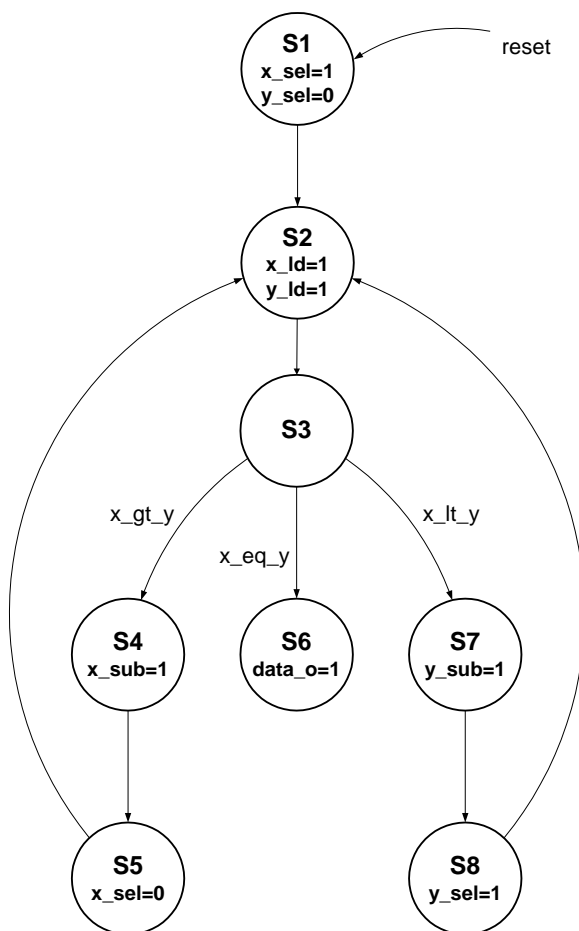
entity sub8 is
    port (
        en: in std_logic;
        inp_a,inp_b: in std_logic_vector(7 downto 0);
        outp: out std_logic_vector(7 downto 0));
end sub8;

architecture behv of sub8 is
begin
    cmb: process(en,inp_a,inp_b)
        variable a,b: integer;
    begin
        if (en='1') then
            a:=conv_integer(inp_a);
            b:=conv_integer(inp_b);
            outp<=conv_std_logic_vector(a-b,8);
        end if;
    end process;
end behv;

```

Παρατήρηση: Οι παραπάνω περιγραφές γενικά δεν δίνουν (βέλτιστο) αποτέλεσμα αν περάσουν από το synthesizer. Στο ίδιο κύκλωμα θα επανέλθουμε και σε επόμενη άσκηση, οπότε θα γραφούν και περιγραφές για σύνθεση.

Το μονοπάτι ελέγχου είναι μια AMK που δέχεται εισόδους από το συγκριτή (τα σήματα x_gt_y , x_eq_y και x_lt_y), και παρέχει εξόδους ελέγχου για όλες τις υπόλοιπες μονάδες (x_sel και y_sel για τους πολυπλέκτες, x_ld και y_ld για τους καταχωρητές εισόδου, x_sub και y_sub για τους αφαιρέτες και $data_en$ για τον καταχωρητή εξόδου). Η λειτουργία της AMK του μονοπατιού ελέγχου περιγράφεται ενδεικτικά με το διάγραμμα του σχήματος 3.11, χρησιμοποιώντας τον ίδιο συμβολισμό με την περίπτωση του θέματος 3.1.



Σχ. 3.11: AMK μονοπατιού ελέγχου κυκλώματος υπολογισμού ΜΚΔ

Το μονοπάτι δεδομένων και το μονοπάτι ελέγχου επικοινωνούν όπως φαίνεται στο σχήμα 3.10. Η δομή τους και η μεταξύ τους διασύνδεση μπορεί να εκφραστεί με μια περιγραφή δομής σε VHDL, όπως η ακόλουθη:

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

entity gcd_calc is
    port (clk,rst: in std_logic;
          x_i,y_i: in std_logic_vector(7 downto 0);
          data_o: out std_logic_vector(7 downto 0));
end gcd_calc;

architecture struct of gcd_calc is
    component mux8_2x1
        port (sel: in std_logic;
              inp_a,inp_b: in std_logic_vector(7 downto 0);
              mout: out std_logic_vector(7 downto 0));
    end component;
    component reg8
        port (en,clk: in std_logic;
              inp: in std_logic_vector(7 downto 0);
              outp: out std_logic_vector(7 downto 0));
    end component;
    component cmp8
        port (inp_a,inp_b: in std_logic_vector(7 downto 0);
              a_gt_b,a_eq_b,a_lt_b: out std_logic;
              outp: out std_logic_vector(7 downto 0));
    end component;
    component sub8
        port (en: in std_logic;
              inp_a,inp_b: in std_logic_vector(7 downto 0);
              outp: out std_logic_vector(7 downto 0));
    end component;
    component fsm
        port (clk,rst: in std_logic; gt,eq,lt: in std_logic;
              sel,ld,sub: out std_logic_vector(1 downto 0);
              out_en: out std_logic);
    end component;
```

```

signal muxx_o,regx_o,subx_o: std_logic_vector(7 downto 0);
signal muxy_o,regy_o,suby_o: std_logic_vector(7 downto 0);
signal cmp_o: std_logic_vector(7 downto 0);
signal x_sel,y_sel,x_ld,y_ld,x_sub,y_sub: std_logic;
      signal x_gt_y,x_eq_y,x_lt_y,data_en: std_logic;

begin
    mux_x: mux8_2x1 port map (x_sel,subx_o,x_i,muxx_o);
    mux_y: mux8_2x1 port map (y_sel,y_i,suby_o,muxy_o);
    reg_x: reg8 port map (x_ld,clk,muxx_o,regx_o);
    reg_y: reg8 port map (y_ld,clk,muxy_o,regy_o);
    cmp: cmp8 port map
      (regx_o,regy_o,x_gt_y,x_eq_y,x_lt_y,cmp_o);
    sub_x: sub8 port map (x_sub,regx_o,regy_o,subx_o);
    sub_y: sub8 port map (y_sub,regy_o,regx_o,suby_o);
    reg_out: reg8 port map (data_en,clk,cmp_o,data_o);
    ctrl: fsm port map (clk,rst,x_gt_y,x_eq_y,x_lt_y,
      sel(0)=>x_sel,sel(1)=>y_sel,
      ld(0)=>x_ld,ld(1)=>y_ld,
      sub(0)=>x_sub,sub(1)=>y_sub,out_en=>data_en);

end struct;

```

Ζητούμενα:

1. Βασιζόμενοι στην περιγραφή της AMK του θέματος 3.1 και στο σχήμα 3.11, περιγράψτε την AMK που αντιστοιχεί στο μονοπάτι ελέγχου του κυκλώματος. Είναι οι τιμές των συνθηκών μετάβασης και των εξόδων πλήρεις έτσι ώστε να λειτουργεί το κύκλωμα σωστά; Αν όχι, κάντε τις απαιτούμενες προσθήκες. Αν κρίνετε σκόπιμο, μπορείτε να κάνετε αλλαγές και στις δομικές μονάδες που έχουν δοθεί (πολυπλέκτη, καταχωρητή, συγκριτή, αφαιρέτη). Συνδέστε τη μονάδα που θα κατασκευάσετε με τον υπόλοιπο κώδικα και κάντε από κοινού προσομοίωση.

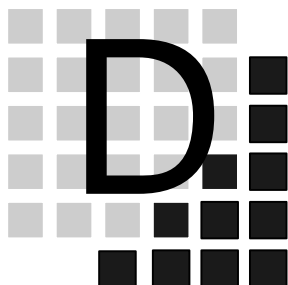
Υπόδειξη: Οι είσοδοι και οι εξόδοι της ζητούμενης AMK φαίνονται από τη δήλωση του αντίστοιχου component στην περιγραφή δομής ολοκλήρου του κυκλώματος. Για αρχιτεκτονική, αντιγράψτε μια από αυτές που δόθηκαν στο θέμα 3.1. Δώστε προσοχή στο σωστό χρονισμό των εξόδων. Αν χρειαστείτε να διαβάσετε την τιμή θύρας εξόδου (πράγμα που δεν επιτρέπεται στη VHDL), χρησιμοποιήστε ένα εσωτερικό σήμα, χειριστείτε το ελεύθερα και σαν είσοδο και σαν έξοδο, και συνδέστε το στην αντίστοιχη θύρα εξόδου.

2. Κάντε έλεγχο του κυκλώματος υπολογισμού ΜΚΔ με test bench που θα διαβάξει τιμές εισόδων και επιθυμητής εξόδου από αρχείο κειμένου.

Υπόδειξη: Χρησιμοποιήστε το πακέτο STD.TEXTIO και τις συναρτήσεις endfile, readline και read που παρέχει.

3. Κάντε λεπτομερή έλεγχο του κυκλώματος υπολογισμού ΜΚΔ με testbench που θα δημιουργεί τιμές εισόδων και επιθυμητής εξόδου αυτόματα.

Υπόδειξη: Χρησιμοποιήστε loops που θα διατρέχουν μια περιοχή ακεραίων και μια συνάρτηση VHDL που θα υπολογίζει το ΜΚΔ τους με βάση τον αλγόριθμο (σε C) που δόθηκε αρχικά. Συγκρίνετέ τον με την έξοδο του κυκλώματος. Εάν σας φαίνεται χρήσιμο, αλλάξτε την εξωτερική περιγραφή του κυκλώματος ώστε το σήμα ένδειξης έγκυρου αποτελέσματος να γίνει έξοδος.



ΑΡΙΘΜΗΤΙΚΑ ΚΥΚΛΩΜΑΤΑ - ΑΘΡΟΙΣΤΕΣ

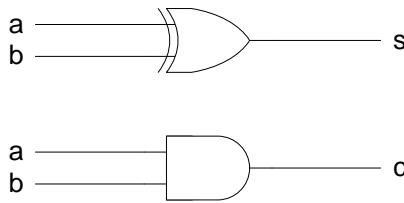
D.1 Άσκηση 4 - Αθροιστές και αφαιρέτες

Θέμα D.1: Ημιαθροιστής

Ο ημιαθροιστής είναι ένα απλό ψηφιακό αριθμητικό κύκλωμα, που δέχεται δύο εισόδους και παράγει δύο εξόδους, που αντιστοιχούν στο άθροισμα και το κρατούμενο των εισόδων. Αν a , b είναι οι είσοδοι του ημιαθροιστή, s το άθροισμα και c το κρατούμενο, ο πίνακας αλήθειας του κυκλώματος είναι ο ακόλουθος:

a	b	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Το αντίστοιχο λογικό κύκλωμα εικονίζεται στο σχήμα 4.1



Σχήμα 4.1 Ημιαθροιστής

Κάθε μια από τις παραπάνω περιγραφές (πίνακας αλήθειας και λογικό κύκλωμα) μπορεί να αποτελέσει τη βάση για την κατασκευή dataflow περιγραφών VHDL. Αυτές είναι οι ακόλουθες:

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

entity h_adder is
    port (
        a,b: in std_logic;
        s,c: out std_logic);
end h_adder;

architecture truth_table of h_adder is
begin
    s<='1' when ((a='1' and b='0') or
                (a='0' and b='1'))
        else '0';
    c<='1' when (a='1' and b='1')
        else '0';
end truth_table;

architecture netlist of h_adder is
begin
    s<=a xor b;
    c<=a and b;
end netlist;
```

Ζητούμενα:

1. Να κάνετε προσομοίωση και σύνθεση και των δύο αρχιτεκτονικών του ημιαθροιστή. Μπορείτε με κάποιο τρόπο (report, RTL σχηματικό,

τεχνολογικό σχηματικό) να αποδείξετε ότι παρότι χρησιμοποιούνται διαφορετικές γλωσσικές δομές, η τελική υλοποίηση είναι η ίδια;

Υπόδειξη: Δοκιμάστε διαφορετικές τεχνολογίες υλοποίησης και περισσότερη ή λιγότερη προσπάθεια υλοποίησης στον synthesizer.

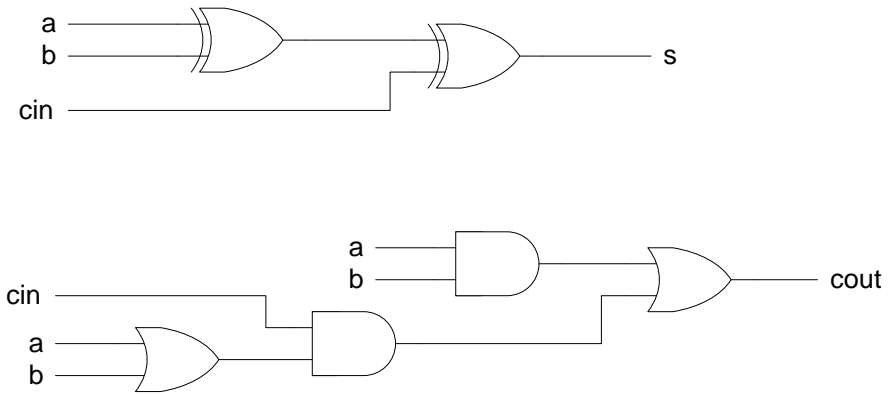
2. Ακολουθείστε τα ίδια βήματα για την περίπτωση του ημιαφαιρέτη.

Θέμα D.2: Πλήρης αθροιστής

Ο πλήρης αθροιστής είναι μια επέκταση του ημιαθροιστή, η οποία λαμβάνει υπόψη μια παραπάνω είσοδο κρατουμένου, που προέρχεται συνήθως από την προηγούμενη βαθμίδα (σε περίπτωση αθροίσματος αριθμών με περισσότερα του 1 bit, όπως θα παρουσιαστεί στο επόμενο θέμα). Αν a , b είναι οι κύριες είσοδοι του πλήρη αθροιστή, cin είναι το κρατούμενο εισόδου, s το άθροισμα και $cout$ το κρατούμενο εξόδου, ο πίνακας αλήθειας του κυκλώματος είναι ο ακόλουθος:

a	b	cin	s	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Το αντίστοιχο λογικό κύκλωμα εικονίζεται στο σχήμα 4.2



Σχήμα 4.2 Πλήρης αθροιστής

Όπως και στην περίπτωση του ημιαθροιστή, για κάθε μια από τις παραπάνω περιγραφές (πίνακας αλήθειας και λογικό κύκλωμα) μπορεί να κατασκευαστεί η αντίστοιχη dataflow VHDL περιγραφή. Επειδή όμως η άθροιση δηλώνεται άμεσα και από το σύμβολο '+', μπορούμε εναλλακτικά να περιγράψουμε τον πλήρη αθροιστή με μία μόνο εντολή, που θα περιλαμβάνει το σύμβολο της άθροισης, και να αφήσουμε το synthesizer να «συμπεράνει» και να χρησιμοποιήσει την αντίστοιχη μονάδα υλικού από τις βιβλιοθήκες που διαθέτει. Οι τρεις εναλλακτικές περιγραφές δίνονται παρακάτω:

```

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

entity f_adder is
    port (
        a,b,cin: in std_logic;
        s,cout: out std_logic);
end f_adder;

architecture truth_table of f_adder is
begin
    s<='1' when ((a='0' and b='0' and cin='1') or
                (a='0' and b='1' and cin='0') or
  
```

```

                                (a='1' and b='0' and cin='0') or
                                (a='1' and b='1' and cin='1'))
        else '0';
    cout<='1' when ((a='0' and b='1' and cin='1') or
                    (a='1' and b='0' and cin='1') or
                    (a='1' and b='1' and cin='0') or
                    (a='1' and b='1' and cin='1'))
        else '0';
end truth_table;

architecture netlist of f_adder is
begin
    s<=a xor b xor cin;
    cout<=((a or b) and cin) or (a and b);
end netlist;

architecture inference of f_adder is
    signal a_vec,b_vec,cin_vec,out_vec: std_logic_vector(1 downto 0);
begin
    a_vec<=('0',a);
    b_vec<=('0',b);
    cin_vec<=('0',cin);
    (cout,s)<=out_vec;
    out_vec<=a_vec+b_vec+cin_vec;
end inference;

```

Ζητούμενα:

1. Να κάνετε προσομοίωση και σύνθεση των αρχιτεκτονικών του πλήρη αθροιστή. Ποια είναι η περισσότερο αποδοτική υλοποίηση;

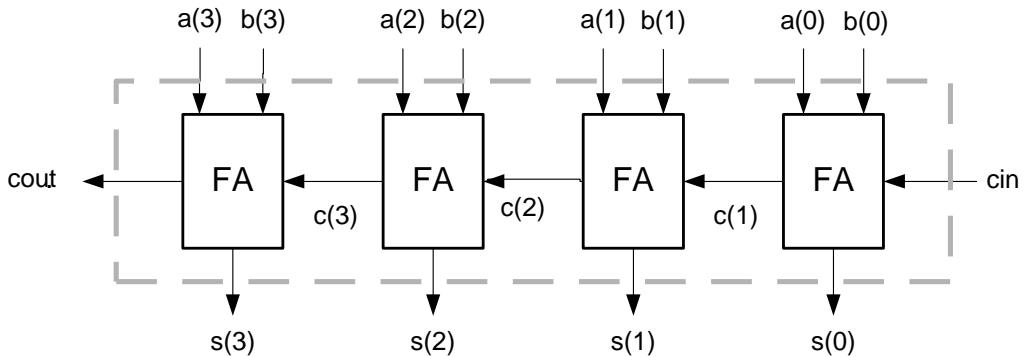
Υπόδειξη: Δοκιμάστε περισσότερη ή λιγότερη προσπάθεια υλοποίησης και αξιολογείτε την δυσκολία δημιουργίας της περιγραφής.

2. Ακολουθείστε τα ίδια βήματα για την περίπτωση του πλήρη αφαιρέτη.

Θέμα D.3: Παράλληλος αθροιστής με διάδοση κρατουμένου

Ο παράλληλος αθροιστής με διάδοση κρατουμένου είναι ένα κύκλωμα που υλοποιεί την πράξη της πρόσθεσης μεταξύ δυαδικών αριθμών των N bit. Όπως εικονίζεται και στο σχήμα 4.3 (για N=4), αποτελείται από N πλήρεις αθροιστές. Η

έξοδος *cout* μιας βαθμίδας συνδέεται με την είσοδο *cin* την επόμενη. Το μειονέκτημα της συγκεκριμένης αρχιτεκτονικής είναι ότι ενώ τα δεδομένα εισόδου είναι ταυτόχρονα διαθέσιμα σε όλες τις βαθμίδες, κάθε μια περιμένει να τελειώσει η προηγούμενη για να λειτουργήσει, ώστε να είναι διαθέσιμο και το *cin*.



Σχήμα 4.3 Παράλληλος αθροιστής των τεσσάρων (4) bit με διάδοση κρατουμένου

Η περιγραφή της δομής ενός τέτοιου αθροιστή γίνεται εύκολα σε VHDL, χρησιμοποιώντας τον πλήρη αθροιστή της προηγούμενης παραγράφου, και την εντολή *generate*. Εναλλακτικά, μπορεί να δοθεί περιγραφή ροής δεδομένων, στην οποία να δίνονται όλες οι λογικές πύλες των επιμέρους αθροιστών σε συνεπτυγμένη μορφή, χρησιμοποιώντας διανύσματα *std_logic_vector*. Ο αθροιστής μπορεί να οριστεί παραμετρικός ως προς το εύρος του, χρησιμοποιώντας τη δομή *generic* της γλώσσας. Οι σχετικές περιγραφές δίνονται παρακάτω:

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

entity adder_n is
    generic (N: integer:=8);
    port (
        a,b: in std_logic_vector(0 to N-1);
        cin: in std_logic;
        s: out std_logic_vector(0 to N-1);
        cout: out std_logic);
end adder_n;
```

```

architecture struct of adder_n is
    component f_adder
        port (
            a,b,cin: in std_logic;
            s,cout: out std_logic);
    end component;
    signal c: std_logic_vector(0 to N);
    begin
        c(0)<=cin;
        cout<=c(N);
        adders: for k in 0 to N-1 generate
            A1: f_adder port map(a(k),b(k),c(k),s(k),c(k+1));
        end generate adders;
    end struct;

architecture rtl of adder_n is
    signal c: std_logic_vector(1 to N);
    begin
        s<=(a xor b) xor (cin&c(1 to N-1));
        c<=((a or b) and (cin&c(1 to N-1))) or (a and b);
        cout<=c(N);
    end rtl;

```

Ζητούμενα:

1. Να κάνετε προσομοίωση και σύνθεση των παραπάνω αρχιτεκτονικών του αθροιστή N bits, καθώς και μιας δικής σας αρχιτεκτονικής, που να χρησιμοποιεί το σύμβολο της πράξης ('+'), και να αφήνει το synthesizer να κατασκευάσει αυτόματα το κύκλωμα (βλ. αρχιτεκτονική inference στο παράδειγμα του πλήρη αθροιστή). Για τη δομική περιγραφή, μην ξεχάσετε να συμπεριλάβετε και το αρχείο του πλήρη αθροιστή. Ποια είναι η περισσότερη αποδοτική υλοποίηση;

ΠΡΟΣΟΧΗ !!! Επειδή τα όρια των εισόδων έχουν δοθεί (0 **to** N-1), θα χρειαστεί να ενεργοποιήσετε την επιλογή "Reverse order" στα stimulators και στα properties των σημάτων, στο παράθυρο waveform του Active-HDL, για να παρατηρήσετε σωστά αποτελέσματα.

2. Ακολουθείστε τα ίδια βήματα για την περίπτωση του παράλληλου αφαιρέτη με διάδοση δανεικού.

3. Αφού διαβάσετε τη θεωρία του βιβλίου και μελετήσετε το σχήμα 1.24, σελ. 39, να κατασκευάσετε αθροιστή 4 bit με πρόβλεψη κρατουμένου. Κάντε προσομοίωση και σύνθεση και συγκρίνετε τα αποτελέσματα με έναν αθροιστή 4 bit με διάδοση κρατουμένου. Παρατηρείστε το αποτέλεσμα της σύνθεσης χρησιμοποιώντας ή όχι κρατούμενο εισόδου.

Θέμα D.4: Παράλληλος συγκριτής

Ο συγκριτής είναι ένα απλό συνδυαστικό κύκλωμα που πολλές φορές περιγράφεται μαζί με άλλα αριθμητικά κυκλώματα αν και ουσιαστικά δεν έχει να κάνει κάποια συγκεκριμένη αριθμητική πράξη. Η συνήθης λειτουργία του είναι να δέχεται δύο ορίσματα και να παράγει εξόδους (μία ή περισσότερες) που περιγράφουν σχέσεις μεταξύ των ορισμάτων. Η πιο γενική μορφή συγκριτή δέχεται δύο δυαδικούς αριθμούς N bit a και b και παράγει 3 εξόδους (λογικά σήματα του 1 bit), για κάθε μια από τις περιπτώσεις $a > b$, $a < b$ και $a = b$. Δύο αντιπροσωπευτικές περιγραφές παραμετρικού παράλληλου συγκριτή σε VHDL δίνονται παρακάτω:

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

entity comparator_n is
    generic (N: integer:=8);
    port (
        a,b: in std_logic_vector (N-1 downto 0);
        eq,gt,lt: out std_logic);
end comparator_n;

architecture behv of comparator_n is
begin
    p0: process(a,b)
    begin
        eq<='1';
        gt<='0';
        lt<='0';
        for j in a'range loop
            if (a(j) and (not b(j)))='1' then
                gt<='1';
```



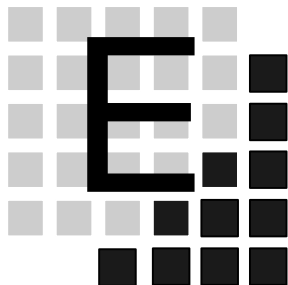
```
        eq<='0';
        exit;
    elsif ((not a(j)) and b(j)) = '1' then
        lt<='1';
        eq<='0';
        exit;
    end if;
end loop;
end process;
end behv;

architecture rtl of comparator_n is
begin
    eq<='1' when (a=b) else '0';
    gt<='1' when (a>b) else '0';
    lt<='1' when (a<b) else '0';
end rtl;
```

Ζητούμενα:

1. Δώστε ακόμα μια περιγραφή RTL για το κύκλωμα του συγκριτή, αναλυτική σε επίπεδο bit .
2. Να κάνετε προσομοίωση και σύνθεση των παραπάνω αρχιτεκτονικών παράλληλου συγκριτή. Ποια είναι η περισσότερη αποδοτική υλοποίηση;
3. Αντικαταστήστε τα πιο αποδοτικά κυκλώματα που κατασκευάσατε σε αυτή την άσκηση με τις αντίστοιχες μονάδες του κυκλώματος υπολογισμού μέγιστου κοινού διαιρέτη (θέμα 3.2). Να κάνετε σύνθεση του παλιού και του νέου κυκλώματος και να σχολιάσετε τα αποτελέσματα.

Υπόδειξη: Προσέξτε ώστε τα κυκλώματα που θα αντικαταστήσετε να έχουν ακριβώς την ίδια συμπεριφορά.

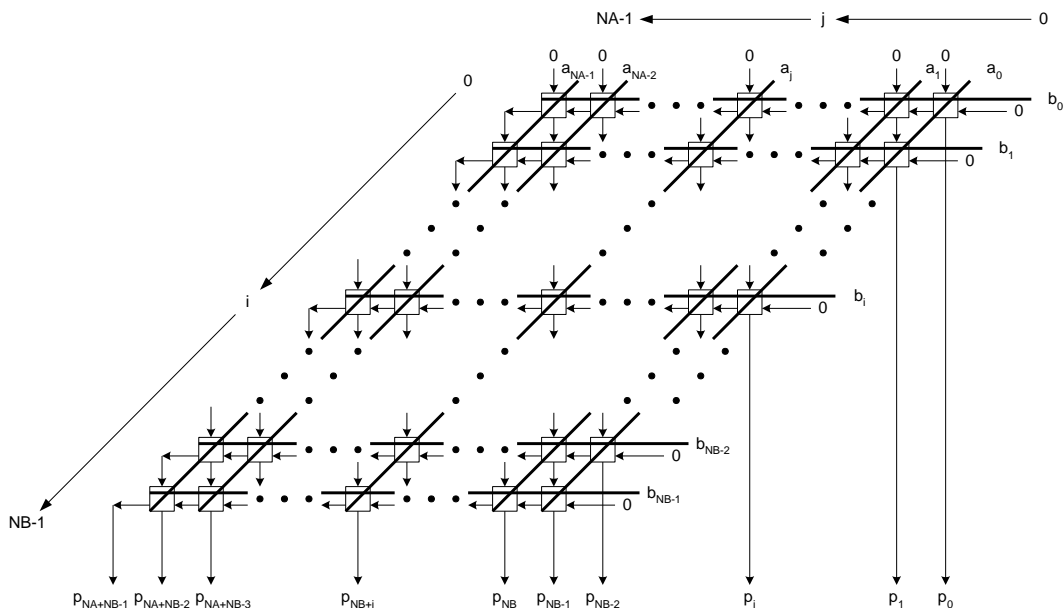


ΑΡΙΘΜΗΤΙΚΑ ΚΥΚΛΩΜΑΤΑ - ΠΟΛΛΑΠΛΑΣΙΑΣΤΕΣ

Ε.1 Άσκηση 5 - Πολλαπλασιαστές

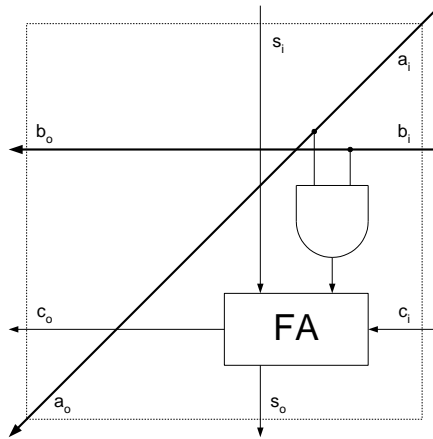
Θέμα Ε.1: Πολλαπλασιαστές βασισμένοι σε carry-propagate αθροιστές

Στο σχήμα φαίνεται η δομή ενός παράλληλου πολλαπλασιαστή απρόσημων δυαδικών ακεραίων. Η άθροιση των μερικών γινομένων γίνεται από αθροιστές διάδοσης κρατουμένου. Ο πολλαπλασιαστέος a έχει ακρίβεια NA bits και ο πολλαπλασιαστής b NB bits.



Σχήμα 5.1 Παράλληλος πολλαπλασιαστής απρόσημων ακεραίων με carry-propagate αθροιστές

Τα κύτταρα που απαρτίζουν τον πολλαπλασιαστή αποτελούνται από ένα πλήρη αθροιστή και μια πύλη AND. Στο κύτταρο j του i αθροιστή, η πύλη AND παράγει το j ψηφίο του μερικού γινομένου i ($a_j * b_i$). Ο πλήρης αθροιστής το προσθέτει με το αντίστοιχης τάξης ψηφίο του μερικού αθροίσματος (s_i) και το κρατούμενο που διαδίδεται από το προηγούμενο κύτταρο (c_i). Το κύτταρο φαίνεται αναλυτικά στο σχήμα 5.2



Σχήμα 5.2 Το κύτταρο του πολλαπλασιαστή

Ακολουθεί η περιγραφή του κυκλώματος σε VHDL και ενός testbench που ελέγχει την λειτουργία του.

```
-- Multiplier cell
--
-- (co,so) <= (a*b) + si + ci
-- (ao,bo) <= (ai,bi)
--
library IEEE;
use IEEE.std_logic_1164.all;

entity mul_cell is
    port
    (
        ai : in  std_logic;
        bi : in  std_logic;
        si : in  std_logic;
```

```
        ci : in std_logic;
        ao : out std_logic;
        bo : out std_logic;
        so : out std_logic;
        co : out std_logic

    );
end mul_cell;

architecture dataflow of mul_cell is
    signal ab : std_logic;
begin
    ab <= ai and bi;
    so <= ab xor si xor ci;
    co <= (ab and si) or (ab and ci) or (si and ci);
    ao <= ai;
    bo <= bi;
end dataflow;

-- Unsigned Parallel Carry-Propagate Multiplier
--
-- p <= a * b;
--
library IEEE;
use IEEE.std_logic_1164.all;
entity mul_u_cp is
    generic
    (
        NA : positive := 6;
        NB : positive := 4
    );
    port
    (
        a : in std_logic_vector(NA-1 downto 0);
        b : in std_logic_vector(NB-1 downto 0);
        p : out std_logic_vector(NA+NB-1 downto 0)
    );
end mul_u_cp;

architecture structural of mul_u_cp is
```

```

subtype a_word is std_logic_vector(NA-1 downto 0);
type a_word_array is array(natural range <>) of a_word;
signal ai,ao,bi,bo,si,so,ci,co :
                                a_word_array(NB-1 downto 0);

component mul_cell
    port
    (
        ai : in std_logic;
        bi : in std_logic;
        si : in std_logic;
        ci : in std_logic;
        ao : out std_logic;
        bo : out std_logic;
        so : out std_logic;
        co : out std_logic
    );
end component;

begin

    -- cell generation
gcb: for i in 0 to NB-1 generate
gca:     for j in 0 to NA-1 generate
gc:         mul_cell port map
            (
                ai => ai(i)(j),
                bi => bi(i)(j),
                si => si(i)(j),
                ci => ci(i)(j),
                ao => ao(i)(j),
                bo => bo(i)(j),
                so => so(i)(j),
                co => co(i)(j)
            );
        end generate;
    end generate;

    -- intermediate wires generation
gasw: for i in 1 to NB-1 generate
        ai(i) <= ao(i-1);

```

```

        si(i) <= co(i-1)(NA-1) & so(i-1)(NA-1 downto 1);
    end generate;
gbciw: for i in 0 to NB-1 generate
gbcjw:   for j in 1 to NA-1 generate
        bi(i)(j) <= bo(i)(j-1);
        ci(i)(j) <= co(i)(j-1);
    end generate;
end generate;

-- input connections
gai:    ai(0) <= a;
gsi:    si(0) <= (others => '0');
gbi:    for i in 0 to NB-1 generate
        bi(i)(0) <= b(i);
    end generate;
gci:    for i in 0 to NB-1 generate
        ci(i)(0) <= '0';
    end generate;

-- output connections
gpa:    p(NA+NB-1 downto NB) <=
        co(NB-1)(NA-1) & so(NB-1)(NA-1 downto 1);
gpb:    for i in 0 to NB-1 generate
        p(i) <= so(i)(0);
    end generate;

end structural;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use std.textio.all;

entity tb_mul_u_cp is
    generic
    (
        NA : positive := 8;
        NB : positive := 4
    );

```

```

end tb_mul_u_cp;

architecture tb_arch of tb_mul_u_cp is
    component mul_u_cp
        generic
            (
                NA : positive := 6;
                NB : positive := 4
            );
        port
            (
                a : in std_logic_vector(NA-1 downto 0);
                b : in std_logic_vector(NB-1 downto 0);
                p : out std_logic_vector(NA+NB-1 downto 0)
            );
    end component;

    signal a : std_logic_vector(NA-1 downto 0);
    signal b : std_logic_vector(NB-1 downto 0);
    signal p_cp : std_logic_vector(NA+NB-1 downto 0);
    constant TIME_STEP : time := 1 ns;

begin
str: mul_u_cp generic map(NA,NB) port map (a,b,p_cp);
    process
        function reportline(a,b,p:in std_logic_vector)
            return string is
                variable report_line : line;
                variable return_str : string(1 to 160);

            begin
                write(report_line,conv_integer(a));
                write(report_line,string("(" * "));
                write(report_line,conv_integer(b));
                write(report_line,string(" = "));
                write(report_line,conv_integer(a)*conv_integer(b));
                write(report_line,string("(" /= "));
                write(report_line,conv_integer(p));
                for i in report_line'range loop
                    return_str(i) := report_line(i);
                end loop;
            end function;
    end process;

```



```

        return return_str;
    end reportline;

begin
    -- create new test vector
    if a(0) = 'U' then
        -- initialize a and b if uninitialized
        a <= (others => '0');
        b <= (others => '0');
    else
        if (a = 2**NA-1) then
            assert (b /= 2**NB-1)
                -- end if b is maximum
            report "End of simulation"
            severity failure;
            -- increase b if a is maximum
            b <= b + 1;
        end if;
        a <= a + 1;      -- increase a
    end if;
    wait for TIME_STEP;
    -- test the new vector
    assert (p_cp = a*b)
        report reportline(a,b,p_cp)
        severity error;
    end process;
end tb_arch;

```

Ζητούμενο:

Να μετατραπεί το κύκλωμα σε συστολικό αλλάζοντας το mul_cell ώστε να περιέχει τις απαιτούμενες καθυστερήσεις (D flip-flops). Ελέγξτε το συστολικό κύκλωμα με νέο testbench.

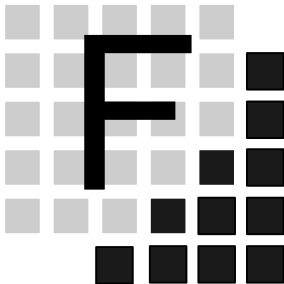
Υπόδειξη : θα χρειαστεί να εισαχθούν επιπλέον μονάδες καθυστέρησης σε σήματα στα όρια του κυκλώματος.

Θέμα Ε.2 : Πολλαπλασιαστές βασισμένοι σε carry-save αθροιστές

Ένας παράλληλος πολλαπλασιαστής μπορεί να υλοποιηθεί και με carry-save αθροιστές όπως στο σχήμα 1.37 του βιβλίου.

Ζητούμενα:

1. Περιγράψτε σε structural VHDL έναν παράλληλο πολλαπλασιαστή απρόσημων ακεραίων με carry-save αθροιστές. Χρησιμοποιήστε το ίδιο κύτταρο που χρησιμοποιήθηκε στο 1^ο θέμα της άσκησης αυτής. Ελέγξτε το κύκλωμα με παρόμοιο testbench.
2. Να περιγραφεί ο παράλληλος πολλαπλασιαστής σε dataflow VHDL χρησιμοποιώντας τον τελεστή '*'. Συγκρίνετε τα αποτελέσματα της σύνθεσης του carry-propagate πολλαπλασιαστή, του carry-save πολλαπλασιαστή και του πολλαπλασιαστή που υλοποιήθηκε αυτόματα από τον τελεστή '*' με παραμέτρους $NA = 4$ και $NB = 3$.

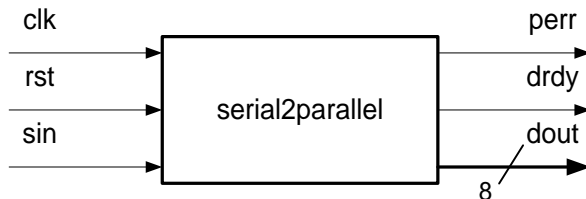


ΑΡΙΘΜΗΤΙΚΑ ΚΥΚΛΩΜΑΤΑ – ΣΕΙΡΙΑΚΑ ΔΕΔΟΜΕΝΑ

ΑΣΚΗΣΗ 6 - ΕΠΕΞΕΡΓΑΣΙΑ ΣΕΙΡΙΑΚΩΝ ΔΕΔΟΜΕΝΩΝ

Θέμα F.1: Μετατροπή σειριακής εισόδου σε παράλληλη και αντίστροφα

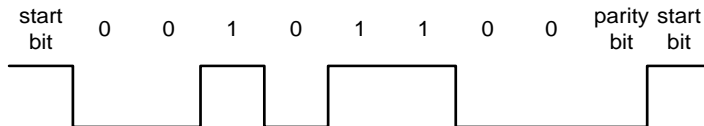
Ένα πρόβλημα που συχνά παρουσιάζεται στη σχεδίαση ψηφιακών συστημάτων είναι η μετατροπή σειριακών δεδομένων σε παράλληλα και αντίστροφα. Το πρόβλημα προκύπτει διότι η μεταφορά σειριακών δεδομένων γίνεται πιο εύκολα, με λιγότερο αριθμό αγωγών και ακροδεκτών εισόδου/εξόδου, ενώ η περαιτέρω επεξεργασία απαιτεί τις εισόδους της σε παράλληλη μορφή. Ένα χαρακτηριστικό παράδειγμα μετατροπέα σειριακών δεδομένων σε παράλληλα δίνεται στο σχήμα 6.1 παρακάτω.



Σχ. 6.1: Κύκλωμα μετατροπής σειριακής εισόδου σε παράλληλη

Το κύκλωμα λειτουργεί ως εξής: Η σειριακή είσοδος *sin*, σε κάθε παλμό του ρολογιού, στέλνει δεδομένα των 8 bit, που συνοδεύονται στην αρχή από ένα start bit, που είναι πάντα '1', και στο τέλος από ένα bit περιττής ισοτιμίας (parity bit) που είναι '1' αν η λέξη περιέχει άρτιο αριθμό μονάδων και '0' αν περιέχει περιττό

αριθμό. Συνολικά, για κάθε λέξη απαιτούνται 10 bit, όπως φαίνεται στο σχήμα 6.2 παρακάτω. Αφού διαβαστούν και τα 10 bit εισόδου, η παράλληλη έξοδος των 8 bit *dout* περιέχει τα bit δεδομένου, η έξοδος *drdy* γίνεται '1', και η έξοδος *perr* γίνεται '1' αν εντοπισθεί λάθος στο bit ισοτιμίας και '0' σε κάθε άλλη περίπτωση.



Σχ. 6.2: Χρονισμός σειριακών δεδομένων εισόδου για το κύκλωμα του σχήματος 6.1

Μια περιγραφή του κυκλώματος σε VHDL είναι η ακόλουθη:

```
library IEEE;

use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity serial2parallel is
    port (clk,rst,sin: in std_logic;
          perr,drdy: out std_logic;
          dout: out std_logic_vector(7 downto 0));
end serial2parallel;

architecture rtl of serial2parallel is

    function reduce_xnor(din: in std_logic_vector)
        return std_logic is
            variable result: std_logic;
        begin
            result:='0';
            for i in din'range loop
                result:=result xor din(i);
            end loop;
            return not(result);
        end reduce_xnor;
```

```

signal reg: std_logic_vector(7 downto 0);
signal cnt: std_logic_vector(2 downto 0);
signal cnt7_ff,sen,normal: std_logic;
signal perr_ff,pen,cen: std_logic;

begin
  regs: process
  begin
    wait until (clk'event and clk='1');
    if (rst='0') then
      reg<=(reg'range=>'0');
      cnt<=(cnt'range=>'0');
      normal<='1';
      perr_ff<='0';
      drdy<='0';
      pen<='0';
      cnt7_ff<='0';
    else
      if (sen='1') then
        reg<=reg(6 downto 0)&sin;
      end if;
      if (perr_ff='1') then
        normal<='0';
      end if;
      if (cen='1') then
        cnt<=cnt+1;
      end if;
      if (pen='1') then
        perr_ff<=reduce_xnor(reg) xor sin;
      end if;
      drdy<=pen;
      cnt7_ff<=cnt(2) and cnt(1) and cnt(0);
      pen<=cnt7_ff;
    end if;
  end process;
  sen<=normal and (cnt7_ff or cnt(2) or cnt(1) or cnt(0));
  cen<='1' when (normal='1') and (pen='0') and
    (cnt7_ff='0') and (perr_ff='0') and
    ((cnt="000" and sin='1') or (cnt/="000"))

```

```

        else '0';
        perr<=perr_ff;
        dout<=reg;
    end rtl;

```

Η λειτουργία της περιγραφής είναι η ακόλουθη: Ο μετρητής *cnt* μετράει τα 8 bit δεδομένων. Κατά τη διάρκεια της μέτρησης και με κάθε παλμό του ρολογιού, ο καταχωρητής *reg* ολισθαίνει μια θέση προς τα αριστερά και αποθηκεύει στη θέση του λιγότερο σημαντικού bit, την τρέχουσα τιμή της σειριακής εισόδου. Θεωρείται ότι τα δεδομένα στη σειριακή είσοδο ξεκινάνε από το περισσότερο σημαντικό bit (MSB) της λέξης. Με το τέλος της μέτρησης του μετρητή, ελέγχεται το bit ισοτιμίας, κάνοντας xor μεταξύ της τιμής που έχει υπολογιστεί με τη συνάρτηση *reduce_xnor* και αυτής που ακολουθεί τα bit δεδομένου στη σειριακή είσοδο. Σε περίπτωση λάθους, η έξοδος *perr* γίνεται '1' και το κύκλωμα μπαίνει σε κατάσταση αναμονής (*normal*='0'), από όπου για να συνεχίσει απαιτείται ένας νέος παλμός αρχικοποίησης (*rst*). Σε κάθε άλλη περίπτωση η έξοδος *pen* γίνεται '1' και υποδηλώνει ότι η έξοδος *dout* περιέχει έγκυρα αποτελέσματα.

Για τον έλεγχο της μονάδας, μπορεί να χρησιμοποιηθεί το ακόλουθο testbench:

```

--*****

--* This file is automatically generated test bench template *
--* By ACTIVE-VHDL <TBgen v1.10>. Copyright (C) ALDEC Inc. *
--*
--* This file was generated on:      12:23 pm, 17/6/2002 *
--* Tested entity name:              serial2parallel *
--* File name contains tested entity: $DSN\src\s2p.vhd *
--*****

library ieee;
use ieee.STD_LOGIC_UNSIGNED.all;
use ieee.std_logic_1164.all;

-- Add your library and packages declaration here ...

```

```
entity serial2parallel_tb is
end serial2parallel_tb;

architecture TB_ARCHITECTURE of serial2parallel_tb is
    -- Component declaration of the tested unit
    component serial2parallel
    port(
        clk : in std_logic;
        rst : in std_logic;
        sin : in std_logic;
        perr : out std_logic;
        drdy : out std_logic;
        dout : out std_logic_vector(7 downto 0) );
    end component;

    constant DELAY1 : time := 50ns;
    constant DELAY2 : time := 25ns;
    -- Stimulus signals - signals mapped to the input and
    -- inout ports of tested entity
    signal clk : std_logic;
    signal rst : std_logic := '0';
    signal sin : std_logic := '0';
    -- Observed signals - signals mapped to the output
    -- ports of tested entity
    signal perr : std_logic;
    signal drdy : std_logic;
    signal dout : std_logic_vector(7 downto 0);

    begin
        -- Add your code here ...
        clkgen : process
        begin
            clk<='0'; wait for DELAY1;
            clk<='1'; wait for DELAY1;
            -- Normally, keep rst high negative logic)

            rst<='1' after 10*DELAY1;
            -- However, send new rst low pulse
```

```

        -- if error has been found

        if (perr='1') then
            rst<='0' after DELAY2;
        end if;
    end process;

dgen : process
    variable RAND : std_logic_vector(4 downto 0) := "01001";
begin
    wait until (clk'event) and (clk='1');
    RAND:=RAND(3 downto 0)&(RAND(4) xor RAND(2));
    sin<=RAND(3) after DELAY2;
end process;

-- Unit Under Test port map
UUT : serial2parallel
    port map
        (clk => clk,
         rst => rst,
         sin => sin,
         perr => perr,
         drdy => drdy,
         dout => dout );

-- Add your stimulus here ...

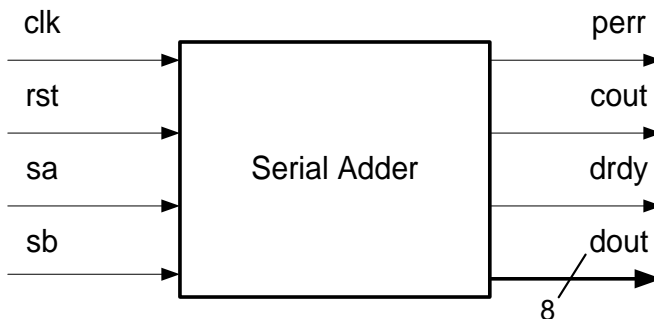
end TB_ARCHITECTURE;

configuration TESTBENCH_FOR_serial2parallel of
    serial2parallel_tb is
        for TB_ARCHITECTURE
            for UUT : serial2parallel
                use entity work.serial2parallel(rtl);
            end for;
        end for;
    end TESTBENCH_FOR_serial2parallel;

```


Ζητούμενα:

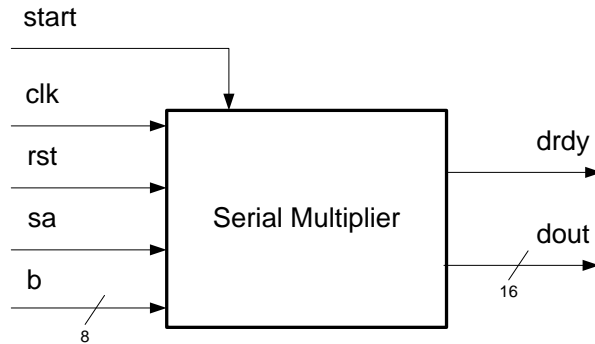
1. Κάντε προσομοίωση του παραπάνω κυκλώματος και του testbench.
2. Περιγράψτε ένα κύκλωμα που να κάνει την αντίστροφη λειτουργία, δηλαδή μετατροπή δεδομένων από παράλληλη σε σειριακή μορφή (μαζί με τα αντίστοιχα start και parity bits). Κάντε προσομοίωση.
3. Φτιάξτε ένα αθροιστή, με δύο εισόδους που παρέχονται σε σειριακή μορφή και έξοδο σε παράλληλη, όπως εικονίζεται στο σχήμα 6.3 παρακάτω, με δύο τρόπους.
 - a. Κάνοντας τις σειριακές εισόδους παράλληλες και χρησιμοποιώντας ένα παράλληλο αθροιστή και
 - b. Χρησιμοποιώντας έναν σειριακό αθροιστή (πλήρη αθροιστή και ανατροφοδότηση κρατουμένου). Το αποτέλεσμα προκύπτει σε σειριακή μορφή και θα πρέπει να μετατραπεί σε παράλληλη.



Σχ. 6.3: Σειριακός αθροιστής δύο σειριακών εισόδων και παράλληλης εξόδου

Στη συνέχεια, συγκρίνετε τα αποτελέσματα με προσομοίωση. Για διευκόλυνσή σας, θεωρίστε ότι η σειριακή είσοδος μεταφέρει τα δεδομένα ξεκινώντας από το λιγότερο σημαντικό bit (LSB), αντίθετα από την περίπτωση του μετατροπέα που παρουσιάστηκε παραπάνω.

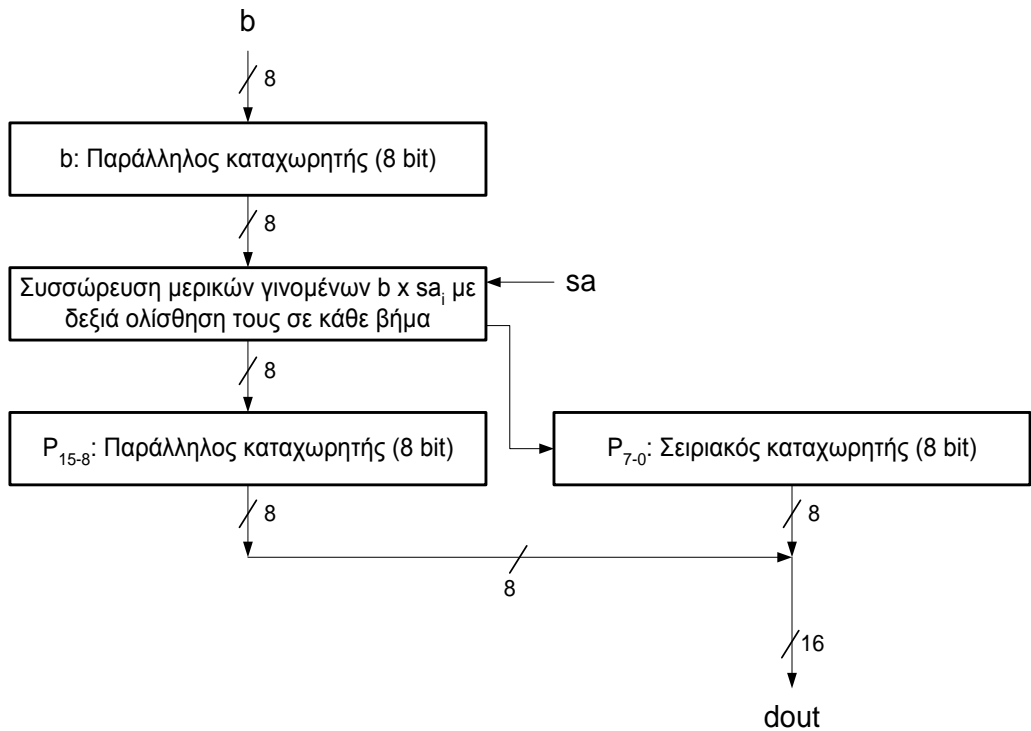
4. Φτιάξτε και προσομοιώστε (με χαρακτηριστικά παραδείγματα) πολλαπλασιαστή θετικών αριθμών 8 bit, με μία σειριακή και μία παράλληλη είσοδο και έξοδο σε παράλληλη μορφή με πλήρη ακρίβεια (16 bit), όπως εικονίζεται στο σχήμα 6.4 παρακάτω:



Σχ. 6.4: Πολλαπλασιαστής μιας σειριακής και μιας παράλληλης εισόδου, και παράλληλης εξόδου

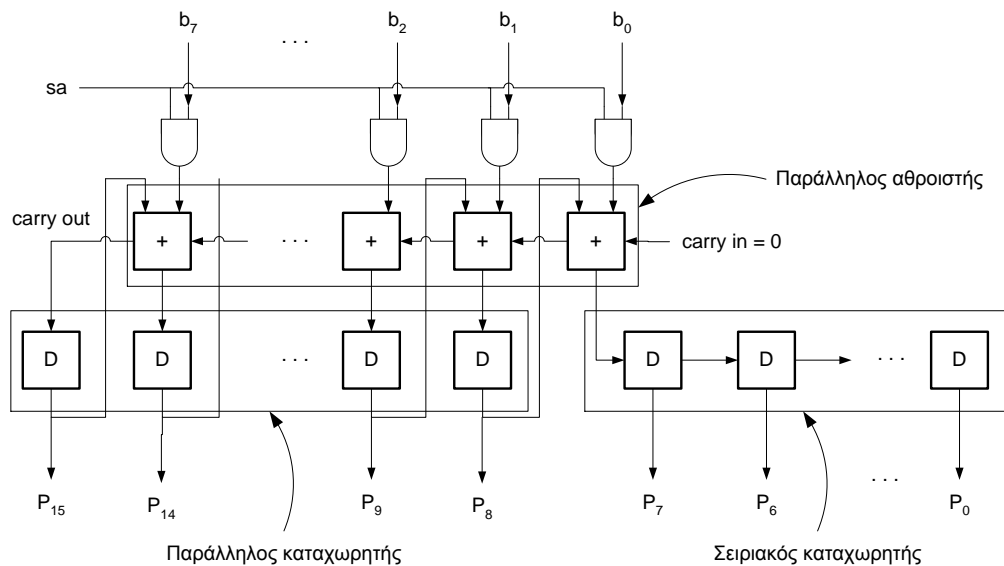
Προτείνεται η υλοποίηση της αρχιτεκτονικής του σχήματος 6.5 σε behavioral VHDL, με έναν καταχωρητή για το σήμα b , έναν αθροιστή για τη δημιουργία των μερικών γινομένων, έναν καταχωρητή για την συσσώρευση των μερικών γινομένων, έναν σειριακό καταχωρητή και έναν μετρητή από 0 μέχρι 7. Θεωρείστε ότι η σειριακή είσοδος έρχεται ξεκινώντας από το LSB, τη στιγμή που ο μετρητής έχει τιμή 0 και το αποτέλεσμα είναι έτοιμο μετά το τέλος και της 7^{ης} περιόδου.

Σημείωση: Μην παρασυρθείτε από το σχηματικό και κάνετε την υλοποίηση σε structural μορφή. Με βάση τα blocks των κυκλωμάτων του Σχ. 6.6 δώστε μια behavioral περιγραφή.

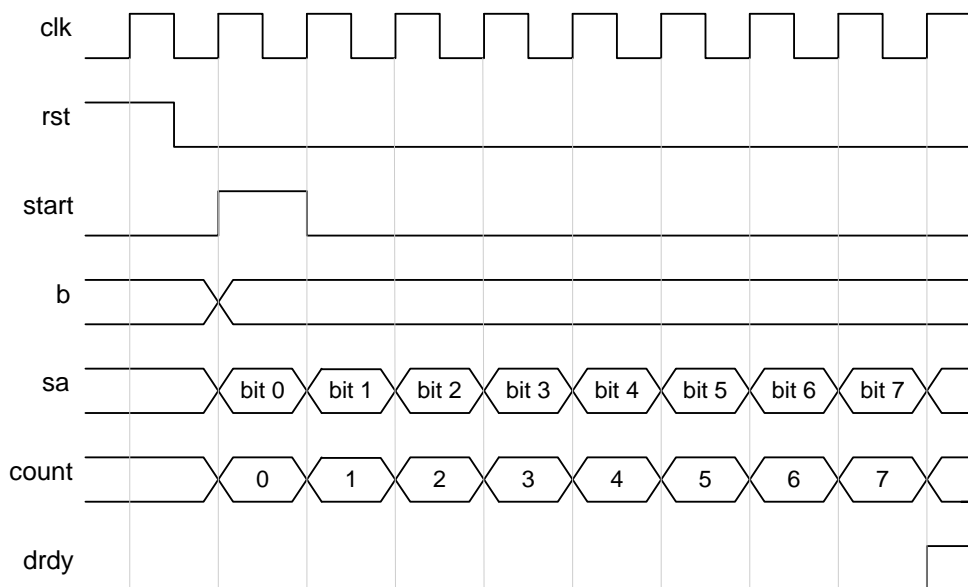


Σχ. 6.5: Διάγραμμα δομικών μονάδων πολλαπλασιαστή

Περισσότερες κυκλωματικές λεπτομέρειες για την αρχιτεκτονική του σχήματος 6.5 δίνονται στο σχήμα 6.6 ενώ το διάγραμμα χρονισμού του κυκλώματος δίνεται στο σχήμα 6.7.



Σχ. 6.6: Κυκλωματικό διάγραμμα πολλαπλασιαστή



Σχ. 6.7: Διάγραμμα χρονισμού πολλαπλασιαστή

Τέλος, προτείνεται να χρησιμοποιήσετε τον ακόλουθο σκελετό για τη μονάδα που βρίσκεται στο υψηλότερο επίπεδο της ιεραρχίας του κυκλώματος:

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

entity sermult is
    port (start,clk,rst: in std_logic;
          sa: in std_logic;
          b: in std_logic_vector(7 downto 0);
          drdy: out std_logic;
          dout: out std_logic_vector(15 downto 0)
    );
end sermult;

architecture rtl of sermult is

    component fulladder8
    port (
        a,b: in std_logic_vector(7 downto 0);
        cin: in std_logic;
        sum: out std_logic_vector(7 downto 0);
        cout: out std_logic);
    end component;

    -- internal signal declarations

    begin
        and_gates: for k in 0 to 7 generate
            -- behavioral implementation of AND gates
            end generate;

        fa8: fulladder8 port map (a=>,b=>,cin=>,sum=>,cout=>);

count: process (start,clk,rst)
    begin
        if (rst='1') then
            -- counter reset
        elsif (start='1') then
```

```
-- counter start to count
elsif (clk'event and clk='1') then
    -- counter increase
end if;
end process;

b_reg: process (clk,rst)
    begin
        if (rst='1') then
            -- b register reset
        elsif (clk'event and clk='1') then
            -- b register implementation and use;
        end if;
    end process;

par_reg: process (clk,rst)
    begin
        if (rst='1') then
            -- parallel register reset
        elsif (clk'event and clk='1') then
            -- parallel register implementation and use
        end if;
    end process;

shift_reg: process (clk,rst)
    begin
        if (rst='1') then
            -- shift register reset
        elsif (clk'event and clk='1') then
            -- shift register implementation and use
        end if;
    end process;
end rtl;
```



Υλοποίηση Εφαρμογών σε XILINX FPGAs

ΑΣΚΗΣΗ 7 – ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΔΙΑΤΑΞΕΩΝ FPGA

Περιγραφή προγραμματιζόμενων διατάξεων FPGA

Τα FPGAs (Field Programmable Gate Arrays) είναι ψηφιακά ολοκληρωμένα κυκλώματα που περιέχουν προγραμματιζόμενα μπλοκ λογικής τα οποία διασυνδέονται με προγραμματιζόμενες συνδέσεις. Ο όρος field στην ονομασία δηλώνει προγραμματισμό στο χώρο λειτουργίας του, εν αντιθέσει με άλλα ολοκληρωμένα, όπου η λειτουργικότητα τους δεν αλλάζει μετά την κατασκευή.

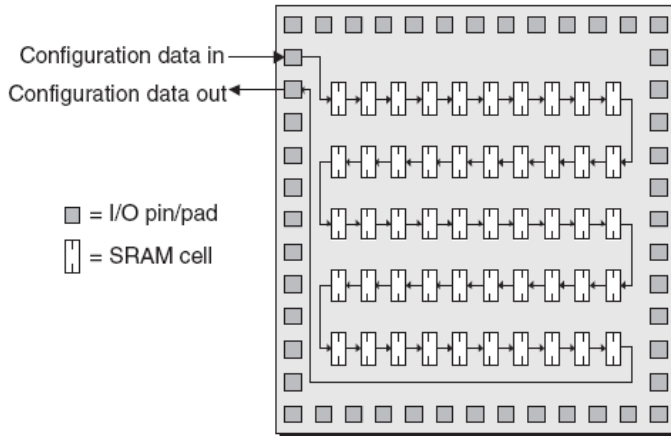
Ακριβώς αυτό το στοιχείο προσδίδει ιδιαίτερη ευελιξία, και τα κάνει μια πολύ καλή πλατφόρμα για γρήγορη υλοποίηση σχεδιάσεων, με εύκολη διόρθωση λαθών και μικρό κόστος, εν αντιθέσει με τις ASIC σχεδιάσεις που απαιτούν πολύ μεγάλο χρόνο υλοποίησης και μεγάλο κόστος.

Τα FPGAs μπορούν να χρησιμοποιηθούν για την υλοποίηση σχεδόν κάθε εφαρμογής, όπως ψηφιακές επικοινωνίες, Σχήμα, βίντεο, ψηφιακή επεξεργασία σήματος. Μπορούν να υλοποιηθούν πάνω τους συστήματα σε ψηφίδα (System on a Chip – SoC), και να γίνει έτσι συνδυασμένη υλοποίηση εφαρμογών σε υλικό και λογισμικό.

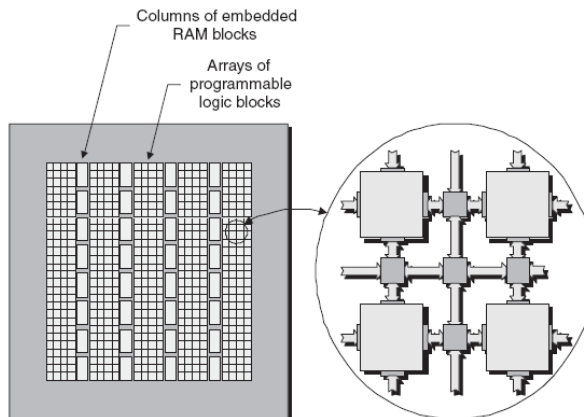
XILINX FPGAs

Υπάρχουν διάφορες κατηγορίες FPGAs ανάλογα με τον τύπο προγραμματισμού των κυττάρων τους (antifuse, SRAM). Στην άσκηση αυτή θα χρησιμοποιήσουμε FPGAs της XILINX, που ο προγραμματισμός τους βασίζεται σε SRAM κύτταρα (ο προγραμματισμός χάνεται με το που κλείσουμε το ρεύμα). Τα

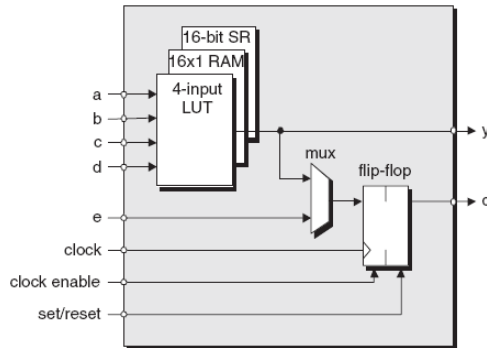
SRAM bit βρίσκονται κατανεμημένα σε όλο το FPGA και προγραμματίζουν τα διάφορα μπλοκ και τις διασυνδέσεις. Ενώνονται μεταξύ τους σαν μία μεγάλη αλυσίδα, και σχηματίζουν έναν καταχωρητή ολίσθησης (όπως στο σχήμα A7.1). Κατά το προγραμματισμό, μία αλυσίδα από bits ολισθαίνει από την είσοδο του καταχωρητή ολίσθησης μέχρι την έξοδο. Στην πραγματικότητα μπορεί να υπάρχουν περισσότερες ανεξάρτητες αλυσίδες, αλλά η αρχή λειτουργίας είναι η ίδια.



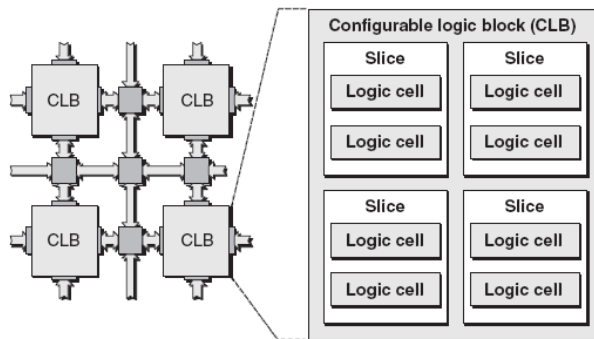
Σχήμα A7.1 SRAM bit προγραμματισμού σαν ένας μεγάλος καταχωρητής ολίσθησης



Σχήμα A7.2 Διάταξη FPGA: μπλοκ λογικής, προγραμματιζόμενες συνδέσεις & ενσωματωμένες RAM



Σχήμα Α7.3 Απλοποιημένη δομή Λογικού Κυττάρου (LC)



Σχήμα Α7.4 Μπλοκ Λογικής (CLB) – Slice – Λογικό Κύτταρο (LC)

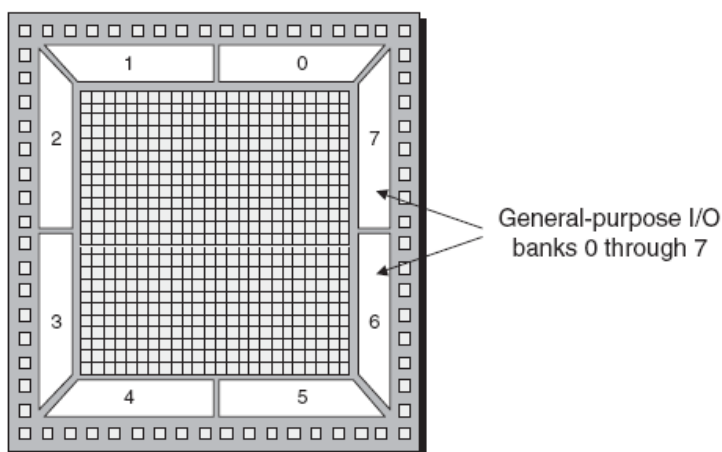
Στην Σχήμα Α7.2 φαίνεται η μορφή της διάταξης του FPGA, που αποτελείται από προγραμματιζόμενα μπλοκ λογικής (CLBs) και προγραμματιζόμενες διασυνδέσεις. Ένα CLB αποτελείται από 4 slices (Σχήμα Α7.4), και κάθε slice από 2 λογικά κύτταρα, που είναι η στοιχειώδης μονάδα προγραμματισμού. Η βασική μορφή των λογικών κυττάρων φαίνεται στην Σχήμα Α7.3. Αποτελείται από μια LUT (Look Up Table) 4 εισόδων, έναν πολυπλέκτη και ένα φλιπ-φλοπ (που μπορεί να λειτουργήσει σαν θετικά/αρνητικά ακμοपुरοδόττο φλιπ-φλοπ ή μανταλωτής). Η LUT (4 εισόδων) μπορεί να υλοποιήσει είτε μία συνδυαστική συνάρτηση 4 εισόδων, ή έναν καταχωρητή ολίσθησης 16 θέσεων, ή να χρησιμοποιηθεί ως κατανεμημένη μνήμη (16bit). Είναι υλοποιημένη σαν μία στήλη από 16 SRAM bit, καθένα από τα οποία αποτελεί είσοδο ενός 16x1 πολυπλέκτη, η είσοδος επιλογής του οποίου είναι οι 4 εισοδοί της LUT. Το κύτταρο είναι λίγο πιο

πολύπλοκο στην πραγματικότητα. Υπάρχει επίσης και λογική κρατούμενου. Τα λογικά κύτταρα μέσα σε ένα slice επικοινωνούν γρήγορα, τα slices μέσα σε ένα CLB επικοινωνούν λίγο πιο αργά, και τέλος τα CLB μεταξύ τους μπορεί να επικοινωνούν με μεγάλες αργές διασυνδέσεις αν βρίσκονται τελείως απομακρυσμένα.

Εκτός από τα CLBs τα FPGA της XILINX περιέχουν και hardwired μνήμες (blockRAM) και πολλαπλασιαστές (18×18bit). Τα στοιχεία αυτά μπορούν να διασυνδεθούν με τα CLBs, κι έτσι να χρησιμοποιούνται στη σχεδίαση.

Τα FPGA περιέχουν προγραμματιζόμενα μπλοκ εισόδου/εξόδου (IOB) στα άκρα. Μάλιστα είναι οργανωμένα σε banks, και κάθε bank μπορεί να χρησιμοποιεί συγκεκριμένο πρότυπο IO (πχ. TTL κλπ). Ο προγραμματισμός έγκειται στον ορισμό της κατεύθυνσης (είσοδος, έξοδος ή είσοδος & έξοδος), αν θα υποστηρίζεται τρικατάστατη λογική, ορισμός pull up/ pull down αντιστάσεων κα.

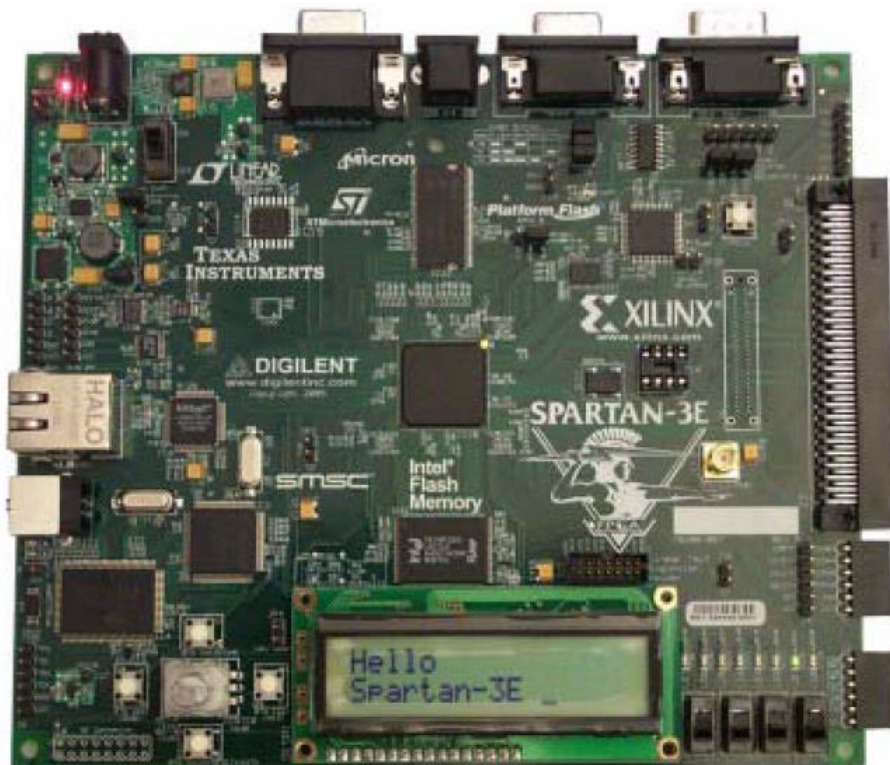
Υπάρχουν επίσης και ψηφιακοί διαχειριστές ρολογιού (Digital Clock Managers - DCMs) που μπορούν να προγραμματιστούν ώστε να παράγουν διάφορες συχνότητες ρολογιού από το ρολόι εισόδου (από τον εξωτερικό κρύσταλλο). Μπορούν να δημιουργηθούν πολλά ρολόγια με διαφορετικές συχνότητες και/ή διαφορά φάσης.



Σχήμα A7.5 Προγραμματιζόμενα μπλοκ εισόδου/εξόδου IOBs οργανωμένα σε banks

Περιγραφή αναπτυξιακής πλακέτας SPARTAN3E STARTER KIT

Στην εργαστηριακή άσκηση θα χρησιμοποιηθεί η αναπτυξιακή πλάκετα SPARTAN3E STARTER KIT της XILINX, που φαίνεται στο Σχήμα A7.6. Η πλακέτα περιέχει το FPGA SPARTAN3E XC3S500E σε package FG320.



Σχήμα A7.6 Αναπτυξιακή Πλακέτα SPARTAN-3E STARTER KIT

Το συγκεκριμένο μοντέλο (XC3S500) περιέχει 10476 λογικά κύτταρα (LCs), 360K bits blockRAM, 72K bits κατανεμημένης μνήμης, 20 hardwired πολλαπλασιαστές 18×18, 4 DCMs. Το ισοδύναμο σε πύλες είναι 500K. Το συγκεκριμένο package (FG320) έχει 320 pins και οι διαθέσιμοι στο χρήστη ακροδέκτες είναι 232.

Η πλακέτα περιέχει 64MB DDR SDRAM, 16MB παράλληλης NOR Flash, 16MB σειριακής SPI Flash, 4MB ROM για αποθήκευση bitstreams προγραμματισμού και ένα CPLD. Περιέχει επίσης οθόνη LCD 2×16 χαρακτήρων, 4

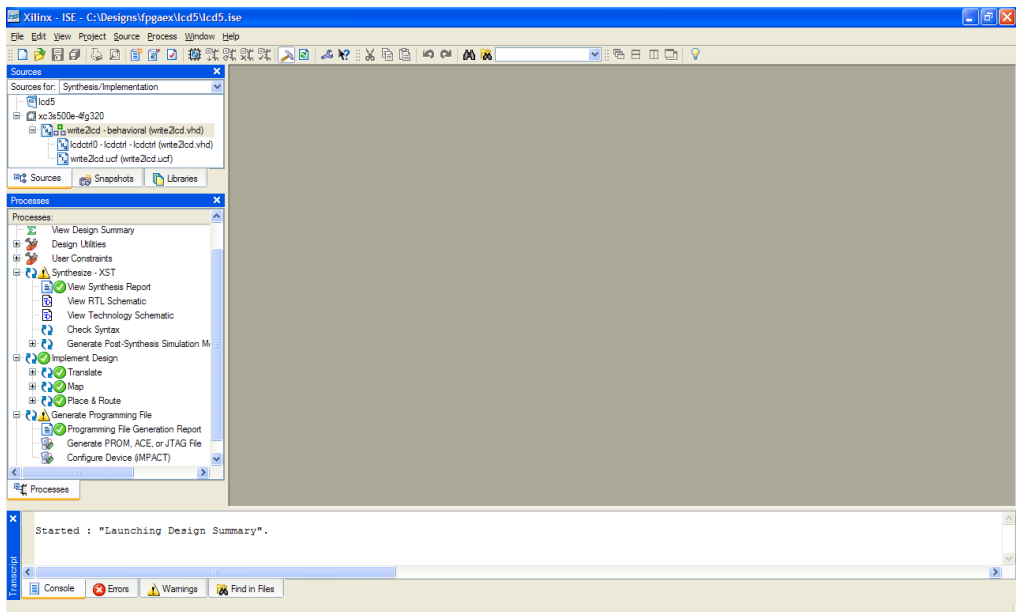
slide switches, 4 πλήκτρα πίεσης (push buttons), 1 περιστροφικό διακόπτη με πλήκτρο πίεσης, 8 LEDs, 1 VGA θύρα, 2 σειριακές θύρες, 1 PS/2 (για πληκτρολόγιο/ποντίκι), 1 10/100 Ethernet PHY, μετατροπείς από αναλογικό σε ψηφιακό και αντίστροφα (ADC, DAC).

Υπάρχουν επιπλέον θύρες επέκτασης για σύνδεση με εξωτερικό κύκλωμα. Ο κρύσταλλος είναι 50MHz, ενώ υπάρχουν είσοδοι για εξωτερικό ρολόι.

Σημείωση: Για περισσότερες πληροφορίες σχετικά με την αναπτυξιακή πλακέτα ανατρέχετε στο manual της συσκευής.

Περιβάλλον προγραμματισμού XILINX ISE 8.1

Για την υλοποίηση του αρχείου προγραμματισμού μίας σχεδίασης & τον προγραμματισμό του FPGA χρησιμοποιούμε το συνοδευτικό λογισμικό XILINX ISE. Ένα στιγμιότυπο χρήσης του προγράμματος φαίνεται στην Σχήμα A7.7. Στο κύριο μέρος ανοίγουν τα διάφορα αρχεία και οι αναφορές. Στο κάτω μέρος βγαίνουν τα logs (εκεί εμφανίζονται τα διάφορα λάθη και τα warnings). Πάνω αριστερά υπάρχει μία δενδρική αναπαράσταση των πηγαίων αρχείων. Κάνοντας δεξιά κλικ μπορούμε να προσθέσουμε νέα αρχεία ή να ανοίξουμε ήδη υπάρχοντα. Η δεντρική αναπαράσταση αντιστοιχεί στην ιεραρχία οντοτήτων της σχεδίασης.



Σχήμα A7.7 Περιβάλλον Εργασίας XILINX ISE 8.1

Ακριβώς από κάτω υπάρχει μία δεντρική αναπαράσταση των διεργασιών που υποστηρίζονται. Οι διαθέσιμες διεργασίες αλλάζουν ανάλογα με το τι πηγαία αρχεία επιλέγουμε στο πάνω μέρος. Οι διεργασίες που αφορούν την υλοποίηση, εμφανίζονται όταν επιλεγεί η top level οντότητα.

Οι κύριες διεργασίες-στάδια υλοποίησης είναι κατά σειρά που εκτελούνται: η *σύνθεση* (*primitives netlist*), η *υλοποίηση* (*translate, map, place & route*) και η *παραγωγή αρχείου προγραμματισμού*. Κάθε στάδιο βγάζει και αντίστοιχη αναφορά. Οι αναφορές αυτές περιέχουν χρήσιμες πληροφορίες και πρέπει να μελετούνται.

Κατά τη *σύνθεση* παράγεται ένα netlist από *primitives* του συγκεκριμένου FPGA. Τα primitives δεν είναι αναγκαίο να είναι υπαρκτές οντότητες του FPGA, αλλά μπορεί να είναι λογικές οντότητες που μπορούν να αντιστοιχηθούν αποδοτικά σε στοιχεία του FPGA. Η αναφορά σύνθεσης περιέχει μία εκτίμηση της επιφάνειας σε αριθμό slices, μία αναλυτική καταγραφή του αριθμού και των τύπων των primitives (LUT1, LUT2 κλπ – στο μέρος Low Level Synthesis), μία εκτίμηση της μέγιστης συνδυαστικής καθυστέρησης (κρίσιμο μονοπάτι). Περιέχει επίσης επισημάνσεις για το τι «κατάλαβε» ο synthesizer για συγκεκριμένα τμήματα του κώδικα μας. Ωστόσο τόσο η εκτίμηση επιφάνειας, όσο και η εκτίμηση καθυστέρησης δεν είναι ακριβείς. Η πρώτη για το ότι η ομαδοποίηση primitives σε CLBs γίνεται σε επόμενο στάδιο, και η δεύτερη γιατί εξαρτάται σε μεγάλο βαθμό από τις διασυνδέσεις, οι οποίες γίνονται γνωστές μετά το στάδιο place & route. (Αξίζει να σημειωθεί ότι η σύνθεση μπορεί να γίνει και από εξωτερικό εργαλείο, πχ. από το Leonardo Spectrum.)

Ακολουθεί η *υλοποίηση*, με πρώτο στάδιο αυτό της *μετάφρασης* (*translate*). Κατά το στάδιο αυτό διάφορα netlist που αποτελούν το design (και βρίσκονται συνήθως στη μορφή EDIF) συνενώνονται και μετατρέπονται σε ένα netlist τύπου XILINX. (εδώ να σημειωθεί ότι μπορεί στη σχεδίαση να έχουμε και EDIF αρχεία (εκτός από VHDL), π.χ. από την χρήση κάποιου έτοιμου core. Το EDIF είναι καθιερωμένο πρότυπο netlist).

Γίνεται επίσης ανάγνωση των *περιορισμών χρήση* (αρχείο *.ucf*). Οι περιορισμοί αυτοί αφορούν είτε αντιστοιχίσεις pins του FPGA με πόρτες του design, είτε χρονικοί περιορισμοί (όπως δηλώσεις ρολογιών & αντίστοιχης συχνότητας, ορισμός απαιτήσεων για μέγιστη συνδυαστική καθυστέρηση από σήμα σε σήμα, ορισμός λανθασμένων μονοπατιών και μονοπατιών πολλών κύκλων για σωστή οδήγηση του εργαλείου που κάνει ανάλυση στατικού χρονισμού), ή χωρικοί

περιορισμοί (οι οποίοι περιλαμβάνουν περιορισμό χώρου συγκεκριμένων υπομονάδων, αλλά και εντοπισμό σε συγκεκριμένο τμήμα του FPGA). Οι περιορισμοί αυτοί γράφονται είτε σε μορφή κειμένου ή με τη βοήθεια αντίστοιχων εργαλείων που εμφανίζονται στο δέντρο διεργασιών (κάτω από το υποδέντρο user constraints). Πιθανά λάθη στο αρχείο περιορισμών αναφέρονται εδώ, αλλά οι περιορισμοί λαμβάνονται υπόψη στα επόμενα στάδια κυρίως.

Κατά το επόμενο στάδιο, το στάδιο *map*, γίνεται ένα επιπλέον reduction στο netlist, και ακολούθως γίνεται χωρισμός σε CLBs. Το στάδιο αυτό μπορεί να αποτύχει αν υπάρχουν δομές στο κώδικα που δεν απεικονίζονται «καλά» στο FPGA, παρότι το στάδιο της σύνθεσης μπορεί να είχε επιτυχία.

Το τελευταίο υποστάδιο της υλοποίησης είναι το *place & route*, το οποίο κάνει και το μεγαλύτερο χρόνο να εκτελεστεί. Κατά το στάδιο αυτό δοκιμάζονται διάφορες τοποθετήσεις των CLBs στο FPGA, και για κάθε τέτοια γίνεται routing των διασυνδέσεων. Η διαδικασία απαιτεί πολλά βήματα του αλγορίθμου καθώς υπάρχουν τοποθετήσεις για τις οποίες δεν μπορεί να επιτευχθεί routing, καθώς επίσης και τοποθετήσεις στις οποίες δεν υπάρχει routing ώστε να πληρούνται οι χρονικοί περιορισμοί. Ο αλγόριθμος λαμβάνει υπόψη του και τους περιορισμούς χρόνιου και χώρου που έχει δώσει ο χρήστης. Γενικά, η τελική κατάληψη χώρου και η καθυστέρηση κρίσιμου μονοπατιού είναι μεγαλύτερες από τις εκτιμήσεις της σύνθεσης.

Το στάδιο *παραγωγής αρχείου προγραμματισμού*, απλά δημιουργεί το *bitstream* προγραμματισμού για τη τοποθέτηση-δρομολόγηση που τελικά επιλέχθηκε. Τρέχοντας τη διεργασία *impact*, μπορούμε να μεταφέρουμε το αρχείο αυτό στο FPGA (θα περνάει bit προς bit από ένα pin στην αλυσίδα των SRAM bits του FPGA).

Δημιουργία project

Επιλέγουμε File->New Project από το μενού και στο παράθυρο που εμφανίζεται (Σχήμα A7.8) δίνουμε ένα όνομα για το project και πατάμε Next. Στο ακόλουθο παράθυρο (Σχήμα A7.9) δίνουμε τα χαρακτηριστικά του FPGA και τον τύπο των πηγαίων αρχείων.

Προσθέτουμε ένα αρχείο VHDL κάνοντας δεξί κλικ στο device xc3s500e-4fg320 στο τμήμα πηγαίων αρχείων (πάνω αριστερά) (Σχήμα A7.10). Δίνουμε όνομα οντότητας και αρχιτεκτονικής, και παράγεται αυτόματα ένα αρχείο (Σχήμα A7.11). Τροποποιούμε την δήλωση της οντότητας ώστε να συμπεριλάβουμε τις πόρτες της σχεδίασης, και συμπληρώνουμε την αρχιτεκτονική.

```
entity fpgademo is
port(clk: in std_logic;
      button: in std_logic;
      switch: in std_logic;
      led: out std_logic);
end fpgademo;

architecture fpgademo of fpgademo is
signal counter: natural range 0 to 2**17-1; --0 every 2.62ms
begin
process(clk, button)
begin
    if button='1' then
        counter<=0;
    elsif clk'event and clk='1' then
        counter<=counter+1;
        if counter=2**17-1 then
            led<=switch;
        end if;
    end if;
end process;

end fpgademo;
```

Αν δεν έχει εμφανιστεί το αρχείο source στη δεντρική αναπαράσταση πάνω αριστερά στο τμήμα των πηγαίων αρχείων, επιλέξτε sources for: Behavioral Simulation, οπότε εμφανίζεται. Κατόπιν, κάντε δεξί κλικ πάνω του και επιλέξτε Properties. Στο pull down μενού του παραθύρου που αναδύεται (Σχ. A7.12), επιλέξτε Synthesis/Imp + Simulation (ή και Synthesis/Implementation Only). Επιλέξτε ξανά Sources for: Synthesis/Implementation. Πλέον εμφανίζεται και το αρχείο source, και οι διεργασίες που αφορούν τη σύνθεση/υλοποίηση.

Τρέξτε τη διεργασία User Constraints/Edit Constraints (Text) (Σχ. A7.13). Στην προτροπή να δημιουργηθεί νέο αρχείο, απαντήστε θετικά. Στη συνέχεια προσθέστε τους περιορισμούς:

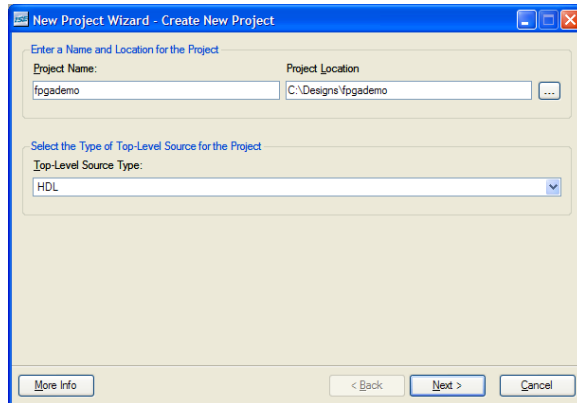
```
NET "button" LOC = "K17" | IOSTANDARD = LVTTTL | PULLDOWN ;
NET "switch" LOC = "L13" | IOSTANDARD = LVTTTL | PULLUP ; #switch 0
NET "led" LOC = "F12" | IOSTANDARD = LVTTTL | SLEW = SLOW | DRIVE =
8 ; #led 0
```

```
NET "clk" LOC = "C9" | IOSTANDARD = LVCMOS33 ;
NET "clk" PERIOD = 20.0ns HIGH 40%;
```

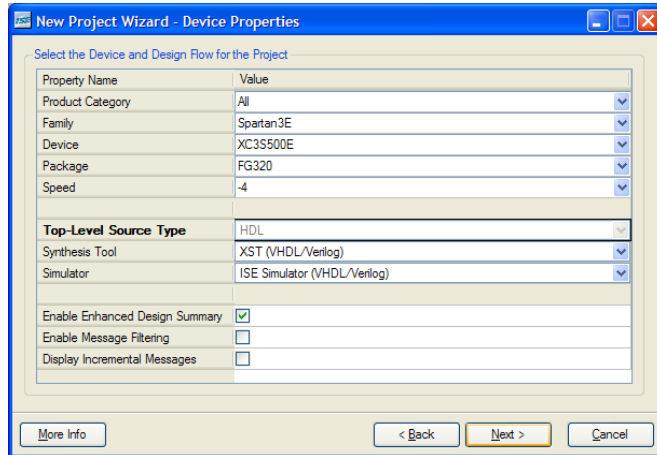
Αντιστοιχούμε πόρτες της toplevel οντότητας με pins του FPGA (θα μπορούσε να γίνει και με γραφικό περιβάλλον – Assign Package Pins). Ορίζουμε επίσης και το ρολόι και τη συχνότητα του.

Στο υποδέντρο Synthesis XST εκτελούμε Check Syntax (Σχήμα A7.14), διορθώνουμε τυχόν λάθη και εκτελούμε σύνθεση. Στη συνέχεια εκτελούμε τα επόμενα βήματα ή τρέχουμε κατευθείαν το τελευταίο, οπότε τρέχουν και όλα τα υπόλοιπα (Σχήμα A7.15). Αν δεν υπάρξει λάθος, είμαστε σε θέση να προγραμματίσουμε το FPGA. Συνδέουμε αρχικά το board με usb καλώδιο στο PC και ανοίγουμε την τροφοδοσία. Την πρώτη φορά που το κάνουμε μπορεί να εγκατασταθεί ο driver. Ακολούθως εκτελούμε Configure Device (iMPACT) (Σχήμα A7.16) στο υποδέντρο Generate Programming File (προσέχουμε να μην έχουμε πολλές υποστάσεις του iMPACT ανοιχτές, γιατί μπορεί να δεσμευτεί η θύρα και να μην επιτυγχάνεται σύνδεση). Επιλέγουμε Configure Devices using Boundary-Scan chain (JTAG), και πατάμε Finish (Σχήμα A7.17). Θα πρέπει να έχει εμφανιστεί ένα μπλε παραλληλόγραμμο που γράφει “identify succeeded”. Στο παράθυρο που μας προτρέπει να αναθέσουμε αρχείο προγραμματισμού πατάμε cancel all (Σχήμα A7.18). Εμφανίζεται μία αλυσίδα με 3 μονάδες προγραμματισμού (Σχήμα 19). Η πρώτη είναι το FPGA, οπότε την επιλέγουμε και με δεξί κλικ επιλέγουμε Assign New Configuration File. Διαλέγουμε το αρχείο προγραμματισμού .bit που έχει δημιουργηθεί στο φάκελο του project (fpgademo.bit) (Σχήμα A7.20). Πατάμε ok στην προειδοποίηση για το jtagclk. Ακολούθως κάνουμε και πάλι δεξί κλικ και επιλέγουμε Program. Πατάμε ok στο επόμενο παράθυρο (Σχήμα 21). Αν όλα έχουν πάει καλά θα εμφανίζεται το μήνυμα Program Succeeded (Σχήμα A7.22). Ανεβοκατεβάστε το slide switch 0 (αριστερότερο) στο board και παρατηρήστε το

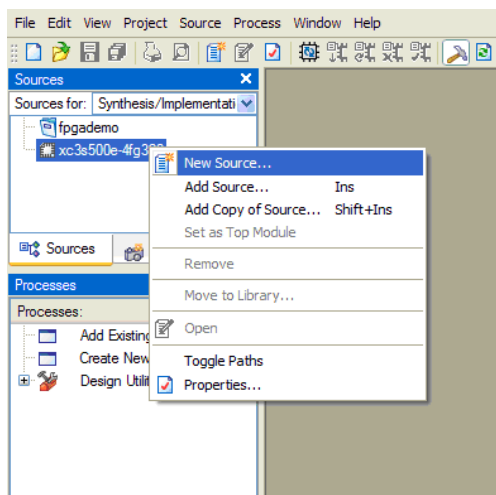
led που ανάβει (led 0 – αριστερότερο). Πατήστε το πλήκτρο πίεσης btn_south (κάτω από τον περιστροφικό διακόπτη) που χρησιμοποιείται σαν reset.



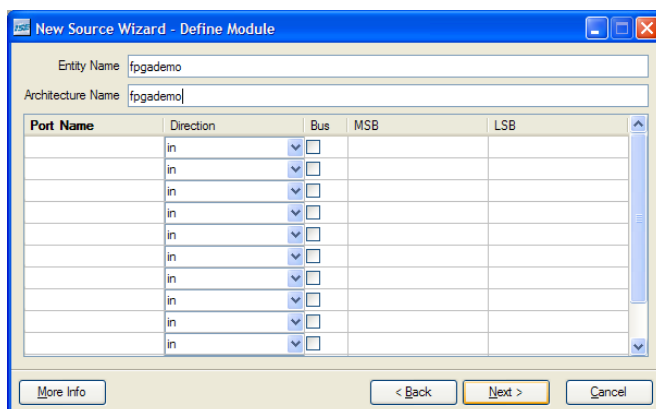
Σχήμα A7.8 Ορισμός ονόματος design



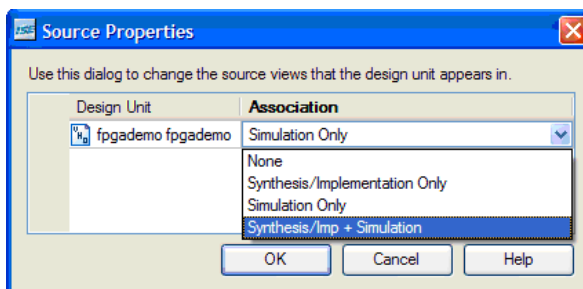
Σχήμα A7.9 Ορισμός χαρακτηριστικών FPGA



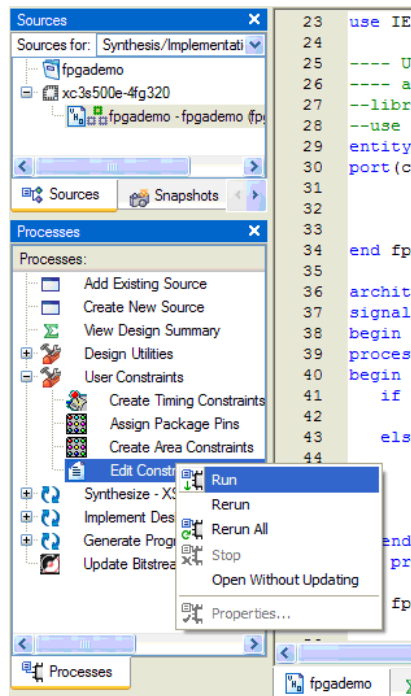
Σχήμα Α7.10 Προσθήκη source



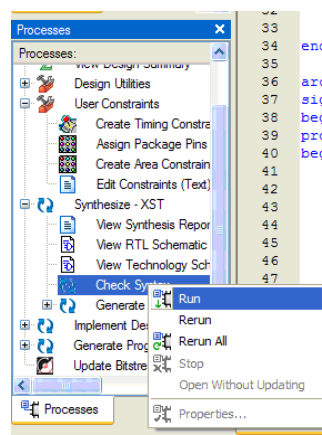
Σχήμα Α7.11 Προσθήκη VHDL Module



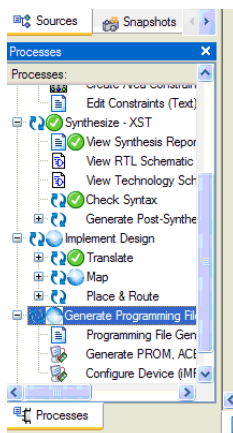
Σχήμα Α7.12 Επισήμανση ότι πρόκειται να ακολουθήσουμε Synthesis flow



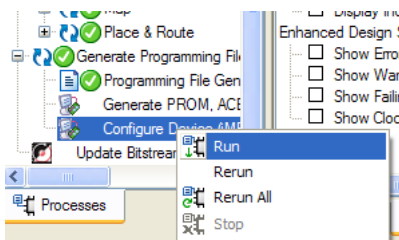
Σχήμα A7.13 Δημιουργία αρχείου περιορισμών χρήστη (ucf)



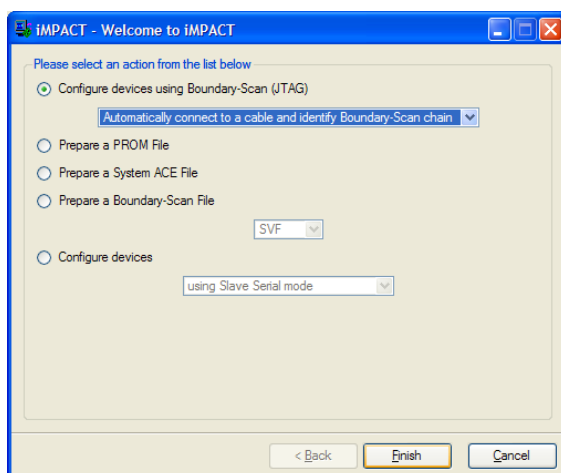
Σχήμα A7.14 Έλεγχος συντακτικών λαθών VHDL



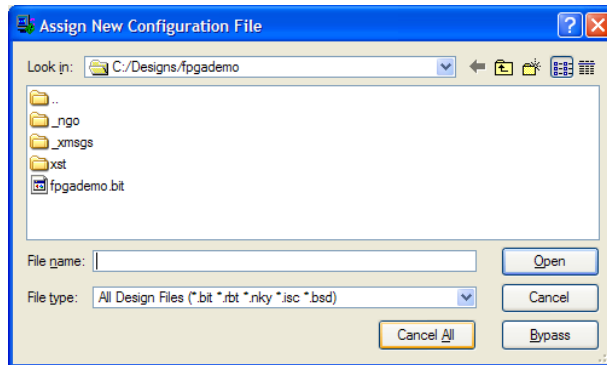
Σχήμα A7.15 Τρέξιμο όλων των προηγούμενων σταδίων, εκτελώντας «παραγωγή αρχείου προγραμματισμού»



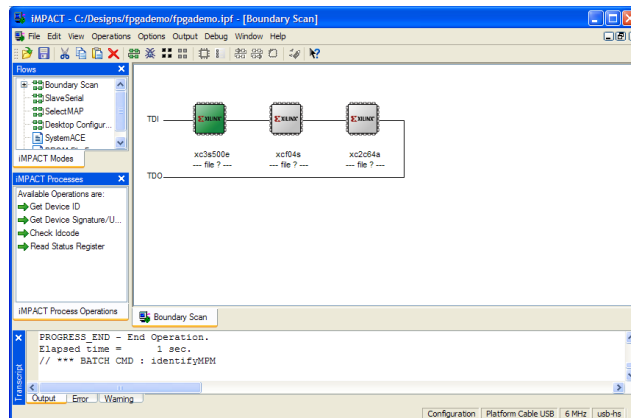
Σχήμα A7.16 Τρέξιμο iMPACT για προγραμματισμό



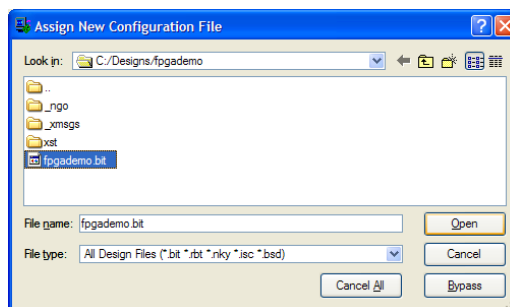
Σχήμα A7.17 Επιλογή αυτόματης ανίχνευσης αλυσίδας



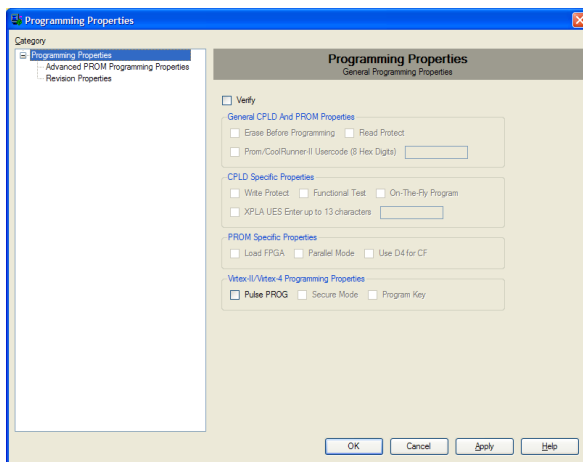
Σχήμα A7.18 Αγνόηση ανάθεσης αρχείου προγραμματισμού



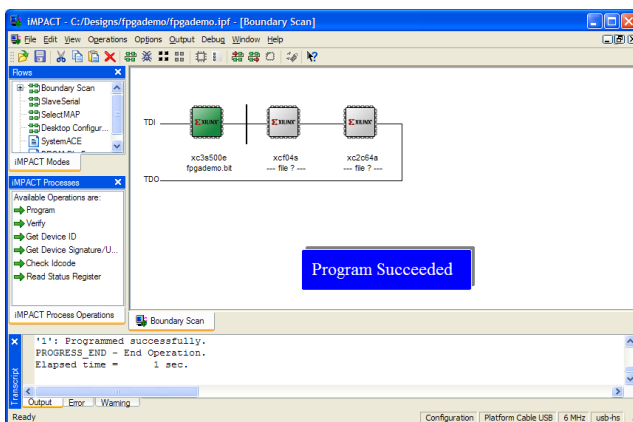
Σχήμα A7.18 Αλυσίδα από Προγραμματιζόμενες Διατάξεις (η πρώτη είναι το FPGA)



Σχήμα A7.20 Ανάθεση αρχείου προγραμματισμού στο FPGA

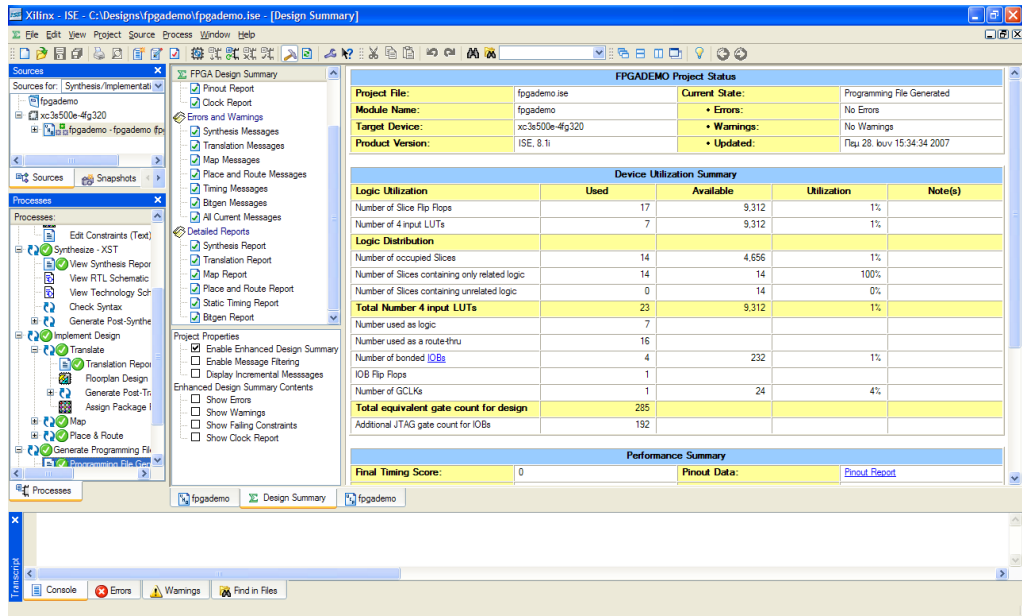


Σχήμα A7.21 Προγραμματισμός FPGA



Σχήμα A7.192 Επιτυχής προγραμματισμός FPGA

Αναφορές για τα διάφορα στάδια της σύνθεσης/υλοποίησης μπορούμε να δούμε επιλέγοντας συγκεκριμένες αναφορές στη σύννοψη που εμφανίζεται μετά το τρέξιμο της σύνθεσης/υλοποίησης. Στα αριστερά μπορούμε να διαλέξουμε αναλυτικές αναφορές για τα διάφορα στάδια.



Σχήμα Α7.23 Συνοπτική Αναφορά Σχεδίασης & Επιμέρους Αναλυτικές Αναφορές Σταδίων

Παρατηρούμε ότι χρησιμοποιήθηκαν 14 slices (1 slice έχει 2 λογικά κύτταρα), που αντιστοιχεί σε 1% χρησιμοποίηση. Στην place & route αναφορά βλέπουμε καθυστέρηση κρίσιμου μονοπατιού 5.985ns που αντιστοιχεί σε συχνότητα 167MHz.

Θέματα Άσκησης

Μετρητής

Στο παράδειγμα αυτό υλοποιούμε έναν απλό up/down μετρητή 8bit στα LEDs του FPGA. Όταν το slide switch 0 είναι 1 επιτρέπεται η μέτρηση, αλλιώς ο μετρητής σταματάει. Το slide switch 1 ρυθμίζει την κατεύθυνση μέτρησης (0: πάνω, 1: κάτω).

Η μέτρηση ανανεώνεται με συχνότητα 1.49Hz.

Το πλήκτρο πίεσης *btn_south* χρησιμοποιείται σαν reset. Εδώ να σημειωθεί ότι κατά το προγραμματισμό του FPGA όλοι οι καταχωρητές παίρνουν μηδενικές τιμές, κι έτσι δεν χρειάζεται αρχικό πάτημα του reset.

Η αντιστοίχιση των θυρών της οντότητας με τα pins του FPGA γίνεται με τη χρήση αρχείου περιορισμών χρήστη, όπως φαίνεται παρακάτω. Στο ίδιο αρχείο δηλώνεται το ρολόι του συστήματος και η επιθυμητή συχνότητα λειτουργίας.

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

entity counter is
port(clk: in std_logic; --clock
      rst: in std_logic; --reset
      switches: in std_logic_vector(3 downto 0);
      leds: out std_logic_vector(7 downto 0));
end counter;

architecture counter of counter is
signal counter1Hz49: std_logic_vector(22 downto 0);--50MHz/2**23=1.49Hz
signal direction: std_logic;--0: up, 1: down
signal enable: std_logic; --counter enable
signal counter: std_logic_vector(7 downto 0);
begin
process(clk, rst)
begin
    if rst='1' then
        counter1Hz49<=(others=>'0');
    elsif clk'event and clk='1' then
        counter1Hz49<=counter1Hz49+1;
    end if;
end process;

process(clk, rst)
begin
    if rst='1' then
        counter<=(others =>'0');
        direction<='0';
    elsif clk'event and clk='1' then
        enable<=switches(0);
        direction<=switches(1);
    end if;
end process;
```



```

        if counter1Hz49=0 and enable='1' then
            if direction='0' then
                counter<=counter+1;
            else
                counter<=counter-1;
            end if;
        end if;
    end if;

end process;

leds<=counter;

end counter;

UCF αρχείο

#btn_south
NET "rst" LOC = "K17" | IOSTANDARD = LVTTTL | PULLDOWN ;

NET "LEDS<7>" LOC = "F9" | IOSTANDARD = LVTTTL | SLEW = SLOW |
DRIVE = 8 ;
NET "LEDS<6>" LOC = "E9" | IOSTANDARD = LVTTTL | SLEW = SLOW |
DRIVE = 8 ;
NET "LEDS<5>" LOC = "D11" | IOSTANDARD = LVTTTL | SLEW = SLOW |
DRIVE = 8 ;
NET "LEDS<4>" LOC = "C11" | IOSTANDARD = LVTTTL | SLEW = SLOW |
DRIVE = 8 ;
NET "LEDS<3>" LOC = "F11" | IOSTANDARD = LVTTTL | SLEW = SLOW |
DRIVE = 8 ;
NET "LEDS<2>" LOC = "E11" | IOSTANDARD = LVTTTL | SLEW = SLOW |
DRIVE = 8 ;
NET "LEDS<1>" LOC = "E12" | IOSTANDARD = LVTTTL | SLEW = SLOW |
DRIVE = 8 ;
NET "LEDS<0>" LOC = "F12" | IOSTANDARD = LVTTTL | SLEW = SLOW |
DRIVE = 8 ;

NET "SWITCHES<0>" LOC = "L13" | IOSTANDARD = LVTTTL | PULLUP ;
NET "SWITCHES<1>" LOC = "L14" | IOSTANDARD = LVTTTL | PULLUP ;
NET "SWITCHES<2>" LOC = "H18" | IOSTANDARD = LVTTTL | PULLUP ;

```

```
NET "SWITCHES<3>" LOC = "N17" | IOSTANDARD = LVTTTL | PULLUP ;
```

```
NET "CLK" LOC = "C9" | IOSTANDARD = LVCMOS33 ;
```

```
NET "CLK" PERIOD = 20.0ns HIGH 40%;
```

Κάντε σύνθεση, υλοποίηση & παραγωγή αρχείου προγραμματισμού. Εκτελώντας το τελευταίο εκτελούνται και τα προηγούμενα (προαπαιτούμενα βήματα). Ανοίξτε το impact (με δεξί κλικ στο Configure Device (impact)) και αναθέστε στην πρώτη μονάδα της αλυσίδας που εμφανίζεται, το αρχείο προγραμματισμού που έχει παραχθεί. Κάντε δεξί κλικ πάνω στη μονάδα και πατήστε program. Το FPGA προγραμματίζεται και στην οθόνη του impact εμφανίζεται μήνυμα επιτυχίας. Ελέγξτε τη λειτουργικότητα του design.

Ζητούμενο:

Υλοποιείτε έναν up counter 8bit με modulo στα LEDs του FPGA (με συχνότητα 1.49Hz όπως και πριν). Ο μετρητής θα μετράει εφόσον το slide switch επίτρησης 0 είναι '1' (πάνω). Το 8bit modulo θα τίθεται ως εξής. Όταν το slide switch επίτρησης είναι '0' (κάτω), πάτημα του *btn_west* θα θέτει τα bits modulo(7 downto 5) με τις τιμές των 3 αριστερότερων slide switches(3 downto 1), πάτημα του *btn_east* θα θέτει τα bits modulo(2 downto 0), ενώ με πάτημα του *btn_north* θα τίθενται τα bits modulo(4 downto 3) με τις τιμές των switches(3 downto 2).

Υπόδειξη: Θα πρέπει να προσθέσετε τις εισόδους *btn_west*, *btn_east* και *btn_north* στη δήλωση της οντότητας, και να προσθέσετε τις ακόλουθες 3 γραμμές στο αρχείο περιορισμών χρήστη (ucf).

```
NET "BTN_EAST" LOC = "H13" | IOSTANDARD = LVTTTL | PULLDOWN ;
```

```
NET "BTN_NORTH" LOC = "V4" | IOSTANDARD = LVTTTL | PULLDOWN ;
```

```
NET "BTN_WEST" LOC = "D18" | IOSTANDARD = LVTTTL | PULLDOWN ;
```

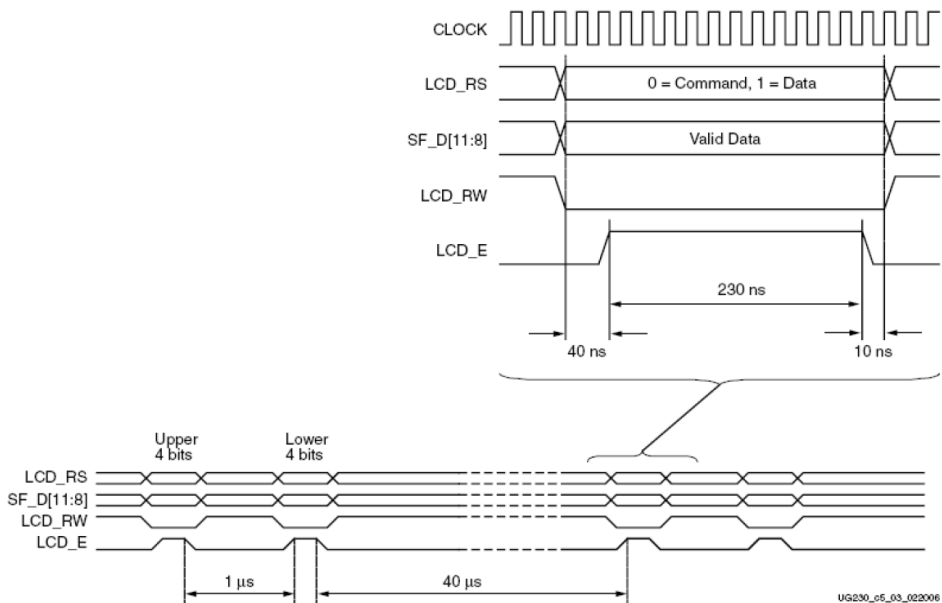
Σε περίπτωση που δεν υπάρχει αρχείο ucf, πατήστε “Edit User Constraints” στο υποδέντρο “User Constraints”. Η λέξη PULLDOWN στους περιορισμούς σημαίνει τοποθέτηση pull down αντίστασης, έτσι ώστε όταν δεν είναι πατημένο το πλήκτρο πίεσης να παρέχει λογική έξοδο 0 (που αποτελεί είσοδο στο FPGA).

Χειρισμός LCD οθόνης χαρακτήρων

Παράδειγμα Α - Αρχικοποίηση LCD & Εκτύπωση μηνύματος

Ο παρακάτω κώδικας αρχικοποιεί την LCD οθόνη και γράφει ένα μήνυμα. Αρχικά εκτελούμε μία σειρά εντολών που προτείνονται από τον κατασκευαστή. Για την μέτρηση των χρόνων χρησιμοποιείται ο μετρητής `lcd_counter2`. Κάθε κύκλος αντιστοιχεί σε 20ns (50MHz ρολόι), οπότε με βάση την τιμή του `lcd_counter2` μετράμε το χρόνο. Όταν ολοκληρωθεί η αρχικοποίηση, τίθεται το σήμα `lcd_ready`. Η LCD οθόνη μπορεί να δεχθεί εντολές. Εκτελούμε τις εντολές που έχουμε αποθηκευμένες στον πίνακα `lcd_initcom`. Και πάλι χρησιμοποιείται ένας μετρητής (ο `lcd_counter`) για τη μέτρηση του χρόνου. Οι εντολές στέλνονται σε 2 μέρη (MSB, LSB) τα οποία απέχουν τουλάχιστον 1μs (50 κύκλους), όπως απαιτείται από τον ελεγκτή της LCD (βλέπε Σχήμα A7.24). Δύο διαδοχικές εντολές πρέπει να απέχουν τουλάχιστον 40μs (2000 κύκλοι), με εξαίρεση την εντολή `clear` που απαιτεί 1.64ms (82000 κύκλοι) για την εκτέλεσή της.

Μετά την εκτέλεση της κάθε εντολής τίθεται το σήμα `next_command` και επεξεργαζόμαστε την επόμενη. Όταν φτάσουμε στο τέλος, τίθεται το σήμα `stop` και σταματάμε.



Σχήμα A7.24 Χρονισμός αποστολής εντολής στον ελεγκτή της LCD. Μία εντολή στέλνεται σε 2 μέρη (upper 4bit & lower 4bit) που απέχουν χρονικά τουλάχιστον 1μs. Δύο διαδοχικές εντολές απέχουν τουλάχιστον 40μs.

Η LCD οθόνη συνδέεται σε συγκεκριμένους ακροδέκτες του FPGA. Μέσω ενός αρχείου περιορισμών χρήστη (ucf αρχείο) αντιστοιχούμε πόρτες της οντότητας μας με pins του FPGA. Επιπλέον ορίζουμε και τα πρότυπα λογικής των ακροδεκτών (πχ TTL, κλπ), καθώς και άλλες πληροφορίες σχετικές με την οδήγηση. Το ucf αρχείο του παραδείγματος παρατίθεται παρακάτω.

Η διεπαφή με τον ελεγκτή της LCD συνιστάται από τα σήματα LCD_RS, SF_D(11 downto 8), LCD_RW και LCD_E. Ο τρόπος με τον οποίο στέλνεται μία εντολή φαίνεται στο σχήμα (και στον κώδικα, όπου υλοποιούνται οι παλμοί με τη βοήθεια του μετρητή *lcd_counter*).

Οι εντολές αφορούν είτε εντολή οθόνης (*lcd_rs='1'*) ή εγγραφή δεδομένων (*lcd_rs='0'*). Η εγγραφή γίνεται με κατέβασμα του *lcd_rw*. Οι κωδικοποιήσεις των χαρακτήρων υπάρχουν στο σχήμα 5-4 του manual (σελ. 45). Ο πίνακας *lcd_initcom* με τις εντολές περιέχει στα δύο MSB τις αντίστοιχες τιμές *lcd_rs*, *lcd_rw*. Επειδή στο παράδειγμα δε χειριζόμαστε ανάγνωση από την LCD, αγνοούμε τη τιμή για το σήμα *lcd_rw*, θέτοντας το σε κάθε εντολή με '0'.

Γενικά την οθόνη μπορούμε να τη δούμε σαν array 2×40 χαρακτήρων με ορατό πλαίσιο 2×16. Για να κάνουμε εγγραφή σε συγκεκριμένη θέση του array θέτουμε πρώτα τη διεύθυνση εγγραφής με ειδική εντολή (Set DDRAM address). Οι διευθύνσεις φαίνονται στο Σχήμα A7.25. Για να μη γράφουμε συνέχεια διευθύνσεις υποστηρίζεται αυτόματη αύξηση διεύθυνσης μετά από κάθε εγγραφή (επιλέγεται με την εντολή entry mode set). Υπάρχει επίσης εντολή (Cursor and Display Shift) που μετατοπίζει το ορατό πλαίσιο αριστερά ή δεξιά (και μάλιστα κυκλικά).

Character Display Addresses																Undisplayed Addresses			
1	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	...	27
2	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	50	...	67
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	...	40

Σχήμα A7.25 Εσωτερική μνήμη DDRAM (Display Data RAM) ελεγκτή LCD. Διευθύνσεις εγγραφής 00-27hex, 40-67hex.

Οι εντολές που υποστηρίζει ο ελεγκτής της LCD φαίνονται στον πίνακα 5-3 (σελ. 46-47 του manual). Οι εντολές αυτές συμπεριλαμβάνουν εντολές καθορισμού λειτουργίας, εντολές καθαρισμού και μετακίνησης οθόνης, εντολές

καθορισμού νέας διεύθυνσης, εντολές εγγραφής δεδομένων στις μνήμες του ελεγκτή και εντολές ανάγνωσης.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity write2lcd is
port(button, clk: in std_logic; sf_d: out std_logic_vector(11 downto 8);
      lcd_e, lcd_rs, lcd_rw, led: out std_logic);
end write2lcd;

architecture behavioral of write2lcd is

subtype lcd_command is std_logic_vector(9 downto 0);
type lcd_command_vector is array (natural range<>) of lcd_command;
constant lcd_initcom : lcd_command_vector(0 to 29) :=
(
    0 => "0000101000", -- Function set
    1 => "0000000100", --"0000000100", -- Entry mode set (auto increment,
no shift)
    2 => "0000001100", -- Display on
    3 => "0000000001", -- Clear display
    4 => "0010000000", -- Set DD RAM address 0
    5 => "1001110111", -- Write 'w'
    6 => "1001110111", -- Write 'w'
    7 => "1001110111", -- Write 'w'
    8 => "1000101110", -- Write '.'
    9 => "1001101101", -- Write 'm'
    10 => "1001101001", --Write 'i'
    11 => "1001100011", -- Write 'c'
    12 => "1001110010", -- Write 'r'
    13 => "1001101111", -- Write 'o'
    14 => "1001101100", --Write 'l'
    15 => "1001100001", -- Write 'a'
    16 => "1001100010", -- Write 'b'
    17 => "1000101110", -- Write '.'
    18 => "1001100101", -- Write 'e'

```

```

19 => "1001100011", -- Write 'c'
20 => "1001100101", --Write 'e'
21 => "1000101110", -- Write '.'
22 => "1001101110", -- Write 'n'
23 => "1001110100", -- Write 't'
24 => "1001110101", --Write 'u'
25 => "1001100101", -- Write 'a'
26 => "1000101110", -- Write '.'
27 => "1001100111", -- Write 'g'
28 => "1001110010", -- Write 'r'
29 => "0000011000", --shifts the entire display left
others => "0000000000");

```

```

signal lcd_ready : std_logic; -- LCD init done flag
signal next_command: std_logic;
signal command_index: natural range 0 to 40;
signal stop: std_logic;
signal command: lcd_command;
signal lcd_counter : natural range 0 to 2**17-1;
signal lcd_counter2 : natural range 0 to 2**20-1;

```

```
begin
```

```
led<=lcd_ready;
```

```
process(clk)
```

```
begin
```

```

    if clk'event and clk='1' then
        if button='1' then
            --lcd initialization
            lcd_ready<='0';
            lcd_counter<=0;
            command_index<=0;
            command<=lcd_initcom(0);
            next_command<='0';
            stop<='0';
            lcd_counter2<=0;
        elsif lcd_ready='1' then
            if next_command='1' then
                lcd_counter<=0;

```

```

        command<=lcd_initcom(command_index);
    else
        lcd_counter<=(lcd_counter+1) mod 2**25;
    end if;
    if stop='0' then
        case lcd_counter is
            when
                                0
=>lcd_rs<=command(9);sf_d<=command(7 downto 4);
                lcd_rw<='0';next_command<='0';
                when 2 =>lcd_e<='1';
                when 14 =>lcd_e<='0';
                when 15 =>lcd_rw<='1';
                when 66 =>--16+50
                                --second part of
command
        lcd_rs<=command(9);sf_d<=command(3 downto 0);
                lcd_rw<='0';
                when 68 =>lcd_e<='1';
                when 80 =>lcd_e<='0';
                when 81 => lcd_rw<='1';
                when 2**17-1=>
                                --for clear command (need for 17bit counter)
                                next_command<='1';
                                if
command_index=(lcd_initcom'high) then
                                stop<='1';
                                else
                                stop<='0';

        command_index<=command_index+1;

                                end if;
                                when others=>null;
                                end case;
                                end if;
                                else
        lcd_counter2 <= (lcd_counter2 + 1) mod 2**20;
        case(lcd_counter2) is

```

```

configuration
    when 750000 =>--15ms after fpga
        sf_d<="0011";lcd_e<='1';

        when 750012 =>lcd_e<='0';
        when 955012 =>lcd_e<='1';--sf_d<="0011"
        when 955024 =>lcd_e<='0';
        when 960024 =>lcd_e<='1';--sf_d<="0011";
        when 960036 =>lcd_e<='0';
        when 962036 =>sf_d<="0010";lcd_e<='1';
        when 962048 =>lcd_e<='1';
        when 964048 =>--poweron initialization

completed
        lcd_ready<='1';
        when others =>null;
    end case;
end if;
end if;
end process;
end behavioral;

```

UCF αρχείο

```

NET "LCD_E" LOC = "M18" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;
NET "LCD_RS" LOC = "L18" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;
NET "LCD_RW" LOC = "L17" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;
NET "SF_D<8>" LOC = "R15" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;
NET "SF_D<9>" LOC = "R16" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;
NET "SF_D<10>" LOC = "P17" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;
NET "SF_D<11>" LOC = "M15" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;

NET "CLK" LOC = "C9" | IOSTANDARD = LVCMOS33 ;

```



```

NET "CLK" PERIOD = 20.0ns HIGH 40%;

#btn_south
NET "BUTTON" LOC = "K17" | IOSTANDARD = LVTTTL | PULLDOWN ;

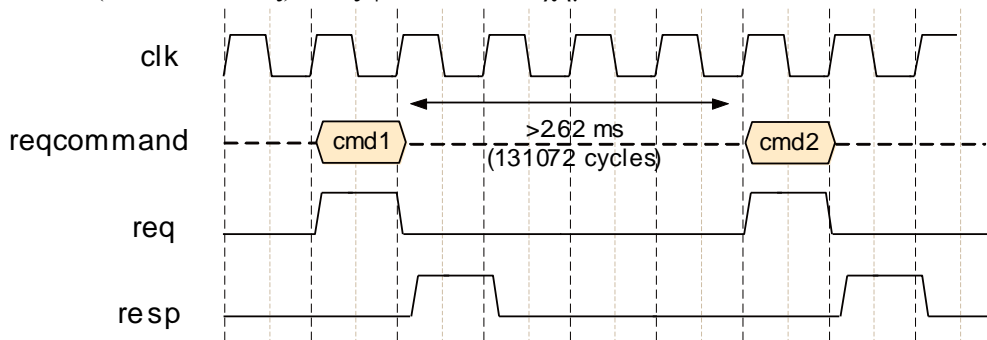
#led<0>
NET "LED" LOC = "F12" | IOSTANDARD = LVTTTL | SLEW = SLOW | DRIVE
= 8 ;

```

Σημείωση: Για καλύτερη κατανόηση του κώδικα, φανταστείτε ότι κάθε σήμα στο οποίο ανατίθεται τιμή συμπεριφέρεται σαν flip flop, και ότι οι τιμές οι οποίες ανατίθενται είναι διαθέσιμες στην επόμενη θετική ακμή ρολογιού.

Παράδειγμα Β - Υλοποίηση Απλοποιημένου Ελεγκτή LCD & Εφαρμογή Περιστροφής Μηνύματος

Για διευκόλυνση στην υλοποίηση εφαρμογών που χρησιμοποιούν την οθόνη LCD, ορίζουμε έναν ελεγκτή για το χειρισμό της και στέλνουμε εντολές σ' αυτόν με μία απλοποιημένη διεπαφή. Συγκεκριμένα, στέλνουμε την εντολή (10bit) μαζί με έναν παλμό *req* (που διαρκεί τουλάχιστον ένα κύκλο, και το πολύ 131072 κύκλους ή 2.62ms). Ο ελεγκτής αποκρίνεται με ένα παλμό ενός κύκλου *resp* για την αποδοχή της εντολής. Δυο διαδοχικές εντολές πρέπει να απέχουν τουλάχιστον 2.62ms (2*17 κύκλους) όπως φαίνεται στο Σχήμα Α7.26.



Σχήμα Α7.26 Επικοινωνία με τον απλοποιημένο ελεγκτή lcdctrl

Ακολουθεί η αρχιτεκτονική του ελεγκτή.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

```

```

use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity lcdctrl is
port(rst, clk: in std_logic;
      --control interface
      req: in std_logic;
      reqcommand: in std_logic_vector(9 downto 0);
      resp: out std_logic;
      lcd_ready_o: out std_logic;
      --interface with the lcd -fpga pins
      sf_d: out std_logic_vector(11 downto 8);
      lcd_e, lcd_rs, lcd_rw: out std_logic);
end lcdctrl;

architecture lcdctrl of lcdctrl is
subtype lcd_command is std_logic_vector(9 downto 0);
type lcd_command_vector is array (natural range<>) of lcd_command;
constant lcd_initcom : lcd_command_vector(0 to 29) :=
(
    0 => "0000101000", -- Function set
    1 => "0000000100",--"0000000100", -- Entry mode set (auto increment,
no shift)

    2 => "0000001100", -- Display on
    3 => "0000000001", -- Clear display
    4 => "0010000000", -- Set DD RAM address 0
    5 => "1001110111", -- Write 'w'
    6 => "1001110111", -- Write 'w'
    7 => "1001110111", -- Write 'w'
    8 => "1000101110", -- Write '.'
    9 => "1001101101", -- Write 'm'
    10 => "1001101001", --Write 'i'
    11 => "1001100011", -- Write 'c'
    12 => "1001110010", -- Write 'r'
    13 => "1001101111", -- Write 'o'
    14 => "1001101100", --Write 'l'
    15 => "1001100001", -- Write 'a'
    16 => "1001100010", -- Write 'b'
    17 => "1000101110", -- Write '.'

```

```

18 => "1001100101", -- Write 'e'
19 => "1001100011", -- Write 'c'
20 => "1001100101", --Write 'e'
21 => "1000101110", -- Write '.'
22 => "1001101110", -- Write 'n'
23 => "1001110100", -- Write 't'
24 => "1001110101", --Write 'u'
25 => "1001100001", -- Write 'a'
26 => "1000101110", -- Write '.'
27 => "1001100111", -- Write 'g'
28 => "1001110010", -- Write 'r'
29 => "0000011000", --shifts the entire display left
others => "0000000000");

```

```

signal lcd_ready : std_logic; -- LCD init done flag
signal lcd_init : std_logic; -- LCD poweron init done flag
signal next_command: std_logic;
signal command_index: natural range 0 to 40;
signal stop: std_logic;
signal command: lcd_command;
signal lcd_counter : natural range 0 to 2**17-1;
signal lcd_counter2 : natural range 0 to 2**20-1;

```

```

signal process_command : std_logic;

```

```

begin
lcd_ready_o<=lcd_ready;
process(clk)
begin
    if clk'event and clk='1' then
        if rst='0' then
            --lcd initialization
            lcd_ready<='0';
            lcd_init<='0';
            lcd_counter<=0;
            process_command<='0';
            command_index<=0;
            command<=lcd_initcom(0);
            next_command<='0';

```

```

        stop<='0';
        lcd_counter2<=0;
    elsif lcd_init='1' and lcd_ready='0' then
        if next_command='1' then
            lcd_counter<=0;
            command<=lcd_initcom(command_index);
        else
            lcd_counter<=(lcd_counter+1) mod 2**17;
        end if;
        if stop='0' then
            case lcd_counter is
                when
                    0
                    =>lcd_rs<=command(9);sf_d<=command(7 downto 4);
                        lcd_rw<='0';next_command
                        <='0';
                    when 2 =>lcd_e<='1';
                    when 14 =>lcd_e<='0';
                    when 15 =>lcd_rw<='1';
                    when 66 =>--16+50
                        --second part of command

            lcd_rs<=command(9);sf_d<=command(3 downto 0);
                lcd_rw<='0';
                when 68 =>lcd_e<='1';
                when 80 =>lcd_e<='0';
                when 81 => lcd_rw<='1';
                when 2**17-1=>--2**23-1 =>
                    --for clear command (need for 17bit counter)
                    if
command_index=(lcd_initcom'high) then
                    stop<='1';

next_command<='0';

process_command<='0';

                                lcd_ready<='1';
                                lcd_counter<=0;
                        else
                                stop<='0';

```

```

next_command<='1';

command_index<=command_index+1;

                                end if;
                                when others=>null;
                                end case;
                                end if;
elseif lcd_ready='1' then
    --take new command
    if req='1' and process_command='0' then
        command<=reqcommand;
        process_command<='1';
        resp<='1';
    end if;

    if process_command='1' then
        lcd_counter<=(lcd_counter+1) mod 2**17;
        case lcd_counter is
            when 0 =>resp<='0';
            when                                     1
=>lcd_rs<=command(9);sf_d<=command(7 downto 4);

                                lcd_rw<='0';
                                when 3 =>lcd_e<='1';
                                when 15 =>lcd_e<='0';
                                when 16 =>lcd_rw<='1';
                                when 67 =>--16+50
                                --second part of command

                                lcd_rs<=command(9);sf_d<=command(3 downto 0);
                                lcd_rw<='0';
                                when 69 =>lcd_e<='1';
                                when 81 =>lcd_e<='0';
                                when 82 => lcd_rw<='1';
                                when          2**17-1          =>
process_command<='0';

                                when others=>null;
                                end case;
                                end if;

```

```

else
    lcd_counter2 <= (lcd_counter2 + 1) mod 2**20;
    case(lcd_counter2) is
        when 750000 =>--15ms after fpga
configuration
            sf_d<="0011";lcd_e<='1';
            when 750012 =>lcd_e<='0';
            when 955012 =>lcd_e<='1';--sf_d<="0011"
            when 955024 =>lcd_e<='0';
            when 960024 =>lcd_e<='1';--sf_d<="0011";
            when 960036 =>lcd_e<='0';
            when 962036 =>sf_d<="0010";lcd_e<='1';
            when 962048 =>lcd_e<='1';
            when 964048 =>--poweron initialization
completed
                lcd_init<='1';
                when others =>null;
            end case;
        end if;
    end if;
end process;
end lcdctrl;

```

Παρακάτω φαίνεται ένα παράδειγμα που χρησιμοποιεί τον ελεγκτή αυτό. Αρχικοποιείται η LCD με το μήνυμα “www.microlab.ece.ntua.gr” και κάθε φορά που πατάμε τα πλήκτρα πίεσης *btn_west* και *btn_east* μετατοπίζουμε το μήνυμα αριστερά ή δεξιά αντίστοιχα (και μάλιστα κυκλικά).

Χρησιμοποιούμε τον μετρητή counter για να σαρώνουμε τα πλήκτρα πίεσης κάθε 2.62ms (2**17 κύκλους). Αν πατηθεί κάποιο εκ των *btn_west* και *btn_east* στέλνουμε εντολή μετατόπισης στον ελεγκτή *lcdctrl*, και ξανασαρώνουμε το πληκτρολόγιο μετά από 0.33s (2**24 κύκλους).

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

```

```

entity write2lcd is

```

```
port(button, clk, btn_east, btn_west: in std_logic; sf_d: out std_logic_vector(11
downto 8);
```

```
    lcd_e, lcd_rs, lcd_rw, led: out std_logic);
end write2lcd;
```

architecture behavioral of write2lcd is

```
component lcdctrl
```

```
port(rst, clk: in std_logic;
    --control interface
    req: in std_logic;
    reqcommand: in std_logic_vector(9 downto 0);
    resp: out std_logic;
    lcd_ready_o: out std_logic;
    --interface with the lcd -fpga pins
    sf_d: out std_logic_vector(11 downto 8);
    lcd_e, lcd_rs, lcd_rw: out std_logic);
end component;
```

```
subtype lcd_command is std_logic_vector(9 downto 0);
```

```
signal req, resp, lcd_ready: std_logic;
```

```
signal reqcommand: lcd_command;
```

```
signal rst: std_logic;
```

```
signal lcd_counter: natural range 0 to 2**24-1; --counter for lcd
```

```
signal counter: natural range 0 to 2**17-1; --counter for io polling
```

```
signal lcdwait: std_logic;
```

```
begin
```

```
    lcdctrl0: lcdctrl port map(rst => rst, clk=>clk,
```

```
        --control interface
```

```
        req => req, reqcommand => reqcommand,
```

```
        resp => resp, lcd_ready_o => lcd_ready,
```

```
        --interface with the lcd -fpga pins
```

```
        sf_d => sf_d, lcd_e => lcd_e, lcd_rs => lcd_rs, lcd_rw => lcd_rw);
```

```
    rst<=not button;
```

```
    led<=lcd_ready;
```

```
    process(clk)
```

```

begin
    if clk'event and clk='1' then
        if rst='0' then
            counter<=0;
            lcd_counter<=0;
            lcdwait<='1';
            req<='0';
        else
            counter<=counter+1;

            if lcdwait='1' then
                lcd_counter<=lcd_counter+1;
                if lcd_counter=2**24-1 then
                    lcdwait<='0';
                end if;
            end if;

            if counter=2**17-1 and lcdwait='0' then
                --check inputs
                if btn_east='1' then
                    req<='1';
                    reqcommand<="0000011000";    --
shifts the entire display left

                    lcdwait<='1';
                elsif btn_west='1' then
                    req<='1';
                    reqcommand<="0000011100";    --
shifts the entire display right

                    lcdwait<='1';
                end if;
            else
                req<='0';
            end if;
        end if;
    end if;
end process;

end behavioral;

```


UCF αρχείο

```

NET "LCD_E" LOC = "M18" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;
NET "LCD_RS" LOC = "L18" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;
NET "LCD_RW" LOC = "L17" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;

NET "SF_D<8>" LOC = "R15" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;
NET "SF_D<9>" LOC = "R16" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;
NET "SF_D<10>" LOC = "P17" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;
NET "SF_D<11>" LOC = "M15" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;

NET "CLK" LOC = "C9" | IOSTANDARD = LVCMOS33 ;
NET "CLK" PERIOD = 20.0ns HIGH 40%;

NET "BUTTON" LOC = "K17" | IOSTANDARD = LVTTL | PULLDOWN ;
#btn_south
NET "BTN_EAST" LOC = "H13" | IOSTANDARD = LVTTL | PULLDOWN ;
#NET "BTN_NORTH" LOC = "V4" | IOSTANDARD = LVTTL | PULLDOWN ;
NET "BTN_WEST" LOC = "D18" | IOSTANDARD = LVTTL | PULLDOWN ;

#led<0>
NET "LED" LOC = "F12" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE
= 8 ;

```

Παράδειγμα Γ - Πρόσθεση Ακεραίων & Εκτύπωση στην LCD

Στο ακόλουθο παράδειγμα κάνουμε πρόσθεση 2 διψήφιων δεκαεξαδικών που εισάγονται μέσω των slide switches. Κάθε φορά που πατάμε το πλήκτρο *btn_east* παίρνουμε την τιμή των 4 slide switches σαν ένα δεκαεξαδικό ψηφίο. Τα πρώτα 2 πατήματα αντιστοιχούν στον πρώτο διψήφιο, τα δύο επόμενα στο δεύτερο,

ενώ με ένα πέμπτο πάτημα εμφανίζεται στην οθόνη το άθροισμα (σε 3 δεκαεξαδικά ψηφία).

Κατά την εκκίνηση, το σήμα *init_done* είναι 0, κι έτσι θα εκτελείται το τμήμα αρχικοποίησης. Χρησιμοποιούμε το σήμα *phase*, για να χωρίζουμε τη διαδικασία σε φάσεις. Η φάση αυξάνεται σε κάθε ακμή ρολογιού, εκτός κι αν της αναθέτουμε την ίδια τιμή, οπότε παραμένουμε στην ίδια. Με τον τρόπο αυτό, με έναν δεύτερο μετρητή, μπορούμε να περιμένουμε μέχρι να ικανοποιηθεί μία συνθήκη. Στην τελευταία φάση τίθεται το σήμα *init_done*, οπότε τελειώνει η διαδικασία.

Δειγματοληπτούμε τα πλήκτρα πίεσης κάθε 2.62ms ($2^{**}17$ κύκλους), και διατηρούμε τον αριθμό ψηφίου στο σήμα *digit*. Αν πατηθεί πλήκτρο πίεσης τίθεται το σήμα *key_pressed*. Αν έχουμε ξεπεράσει το τελευταίο ψηφίο (*digit=4*), εκτελούμε υπολογισμό (*compute='1'*). Στην επόμενη ακμή ρολογιού ξεκινάει η εκτύπωση ψηφίου ή ο υπολογισμός και η εκτύπωση του αποτελέσματος. Και πάλι οργανώνουμε τις διαδικασίες σε φάσεις.

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

entity write2lcd is
port(button, clk, btn_east: in std_logic;
      switches: in std_logic_vector(3 downto 0);
      sf_d: out std_logic_vector(11 downto 8);
      lcd_e, lcd_rs, lcd_rw: out std_logic;
      led: out std_logic);
end write2lcd;

architecture behavioral of write2lcd is

component lcdctrl
port(rst, clk: in std_logic;
     --control interface
     req: in std_logic;
     reqcommand: in std_logic_vector(9 downto 0);
```

```

        resp: out std_logic;
        lcd_ready_o: out std_logic;
        --interface with the lcd -fpga pins
        sf_d: out std_logic_vector(11 downto 8);
        lcd_e, lcd_rs, lcd_rw: out std_logic);
end component;

subtype lcd_command is std_logic_vector(9 downto 0);
signal req, resp, lcd_ready: std_logic;
signal reqcommand: lcd_command;
signal rst: std_logic;

signal counter: natural range 0 to 2**17-1; --counter for io polling
signal lcd_counter: natural range 0 to 2**24-1;
signal compute, keypressed, init_done: std_logic;
signal phase: natural range 0 to 16;
signal digit: natural range 0 to 4;
type digitarray is array(natural range <>) of std_logic_vector(3 downto 0);
signal data: digitarray(0 to 3);
signal result: std_logic_vector(8 downto 0);
begin

lcdctrl0: lcdctrl port map(rst=> rst, clk=>clk,
        --control interface
        req=>req, reqcommand=> reqcommand,
        resp=>resp, lcd_ready_o=>lcd_ready,
        --interface with the lcd -fpga pins
        sf_d=> sf_d, lcd_e=> lcd_e, lcd_rs=> lcd_rs, lcd_rw=>lcd_rw);

rst<=not button;
led<=lcd_ready;
process(clk)
function encode(p: in std_logic_vector(3 downto 0)) return std_logic_vector is
    variable a: std_logic_vector(7 downto 0);
begin
    case p is
        when "0000" => a:="00110000";--0
        when "0001" => a:="00110001";--1

```

```

when "0010" => a:="00110010";--2
when "0011" => a:="00110011";--3
when "0100" => a:="00110100";--4
when "0101" => a:="00110101";--5
when "0110" => a:="00110110";--6
when "0111" => a:="00110111";--7
when "1000" => a:="00111000";--8
when "1001" => a:="00111001";--9
when "1010" => a:="01000001";--a
when "1011" => a:="01000010";--b
when "1100" => a:="01000011";--c
when "1101" => a:="01000100";--d
when "1110" => a:="01000101";--e
when "1111" => a:="01000110";--f
when others => null;

end case;
return a;
end encode;
begin
  if clk'event and clk='1' then
    if rst='0' then
      counter<=0;
      req<='0';
      digit<=0;
      compute<='0';
      keypressed<='0';
      init_done<='0';
      phase<=0;
    else
      counter<=counter+1;

      if init_done='0' then
        --some display init
        phase<=phase+1;
        case phase is
          when 0 => phase<=0;
            if lcd_ready='1' then --wait
for lcd initialization
                                phase<=1;

```

```

end if;
when 1 => phase<=1;
lcd_counter<=(lcd_counter+1) mod 2**24;
--just to see the init msg
if lcd_counter=2**24-1 then
    phase<=2;
end if;
when 2 =>
reqcommand<="0000000001";--clear display
req<='1';
when 3 => req<='0'; lcd_counter<=0;
when 4 => phase<=4;
lcd_counter<=(lcd_counter+1) mod 2**18;
if lcd_counter=2**18-1 then
    phase<=5;
end if;
when 5 =>
reqcommand<="0010000000"; -- Set DD RAM address 0
req<='1';
when 6 => req<='0'; lcd_counter<=0;
when 7 => phase<=7;
lcd_counter<=(lcd_counter+1) mod 2**18;
if lcd_counter=2**18-1 then
    init_done<='1';
end if;
when others =>null;
end case;
end if;

if keypressed='1' and compute='0' then
    phase<=phase+1;
    case phase is
        when 0 =>
reqcommand<="10"&encode(data(digit));
--write data(digit);
req<='1';
when 1 => req<='0'; digit<=digit+1;
lcd_counter<=0;

```

```

when      2      =>      phase<=2;

lcd_counter<=lcd_counter+1;

                                if lcd_counter=2**24-1 then
                                    keypressed<='0';
                                end if;
                                when others =>null;
                                end case;
                                end if;

                                if compute='1' then
                                    phase<=phase+1;
                                    case phase is
                                        when
                                                    0
=>result<=("0"&data(0)&data(1))+("0"&data(2)&data(3));

                                when
                                                    1      =>
reqcommand<="10"&encode("000"&result(8));

                                req<='1';digit<=0;

                                when 2 => req<='0'; lcd_counter<=0;
                                when      3      =>      phase<=3;

                                if lcd_counter=2**18-1 then
                                    phase<=4;
                                end if;
                                when
                                                    4      =>
reqcommand<="10"&encode(result(7 downto 4));

                                req<='1';
                                when 5 => req<='0'; lcd_counter<=0;
                                when      6      =>      phase<=6;

                                if lcd_counter=2**18-1 then
                                    phase<=7;
                                end if;
                                when
                                                    7      =>
reqcommand<="10"&encode(result(3 downto 0));

                                req<='1';
                                when 8 => req<='0'; lcd_counter<=0;

```

```

                                when      9      =>      phase<=9;
lcd_counter<=(lcd_counter+1) mod 2**18;
                                if lcd_counter=2**18-1 then
                                    phase<=10;
                                end if;
                                when      10      =>      phase<=10;
lcd_counter<=(lcd_counter+1) mod 2**24;
                                if lcd_counter=2**24-1 then
                                    compute<='0';
                                    keypressed<='0';
                                end if;
                                when others =>null;
                                end case;
                                end if;

                                if counter=2**17-1 and keypressed='0' and init_done='1'
then
                                --check inputs
                                if btn_east='1' then
                                    if digit/=4 then
                                        data(digit)<=switches;
                                    end if;
                                    if digit=4 then
                                        compute<='1';
                                    end if;
                                    --digit<=digit+1;
                                    keypressed<='1';
                                    phase<=0;
                                end if;
                                end if;
                                end if;
                                end process;

                                end behavioral;

```

```

NET "LCD_E" LOC = "M18" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;
NET "LCD_RS" LOC = "L18" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;
NET "LCD_RW" LOC = "L17" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;
NET "SF_D<8>" LOC = "R15" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;
NET "SF_D<9>" LOC = "R16" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;
NET "SF_D<10>" LOC = "P17" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;
NET "SF_D<11>" LOC = "M15" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;

```

```

NET "CLK" LOC = "C9" | IOSTANDARD = LVCMOS33 ;
NET "CLK" PERIOD = 20.0ns HIGH 40% ;

```

```

NET "BUTTON" LOC = "K17" | IOSTANDARD = LVTTL | PULLDOWN ;
#btn_south
NET "BTN_EAST" LOC = "H13" | IOSTANDARD = LVTTL | PULLDOWN ;
#NET "BTN_NORTH" LOC = "V4" | IOSTANDARD = LVTTL | PULLDOWN ;
#NET "BTN_WEST" LOC = "D18" | IOSTANDARD = LVTTL | PULLDOWN ;

#led<0>
NET "LED" LOC = "F12" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE
= 8 ;

```

```

NET "SWITCHES<0>" LOC = "L13" | IOSTANDARD = LVTTL | PULLUP ;
NET "SWITCHES<1>" LOC = "L14" | IOSTANDARD = LVTTL | PULLUP ;
NET "SWITCHES<2>" LOC = "H18" | IOSTANDARD = LVTTL | PULLUP ;
NET "SWITCHES<3>" LOC = "N17" | IOSTANDARD = LVTTL | PULLUP ;

```


Ζητούμενα (κάντε 3 από τα 5 ερωτήματα):

α) Τροποποιήστε το παράδειγμα 2Α, ώστε να τυπώσετε ένα δικό σας μήνυμα. Συμβουλευτείτε το σχήμα 5-4 του manual (σελ 45) για τις κωδικοποιήσεις των χαρακτήρων.

β) Υλοποιήστε έναν 8bit up counter με συχνότητα περίπου 1Hz. Η μέτρηση θα φαίνεται στους 2 πρώτους χαρακτήρες της LCD οθόνης σε μορφή 2 δεκαεξαδικών ψηφίων. Με reset (*btn_south*), η μέτρηση θα μηδενίζεται.

Υπόδειξη: Κάντε χρήση του απλοποιημένου ελεγκτή του παραδείγματος Β. Ορίστε έναν μετρητή που θα γίνεται 0 κάθε περίπου 1 δευτερόλεπτο, και έναν 8 bit μετρητή *counter* που θα μετράει τα δευτερόλεπτα. Κάθε φορά που ο πρώτος παίρνει τη μέγιστη τιμή, ανανεώστε τον μετρητή *counter* και εκδώστε εντολές για ανανέωση της μέτρησης στην LCD οθόνη. Χρησιμοποιήστε φάσεις, όπως στα παραδείγματα Β, Γ.

γ) Τροποποιήστε το παράδειγμα Γ, ώστε να εκτελεί πολλαπλασιασμό, αντί για πρόσθεση. Χρησιμοποιείτε τον τελεστή * και αφήστε τον synthesizer να επιλέξει πολλαπλασιαστή. Προσθέστε ένα επιπλέον ψηφίο για το αποτέλεσμα. Παρατηρήστε το synthesis report. Πως υλοποιήθηκε ο πολλαπλασιαστής; Πόση είναι η εκτιμώμενη συχνότητα λειτουργίας;

δ) Τροποποιήστε το παράδειγμα Γ, ώστε κάθε φορά που πατάμε το πλήκτρο *btn_north* να καθαρίζει η οθόνη και να ξεκινάμε είσοδο από τον πρώτο χαρακτήρα της πρώτης γραμμής. Επιπλέον, μετά τα δύο πρώτα ψηφία να εμφανίζεται το σύμβολο +, και μετά και τα δύο επόμενα το = (πχ AF+98=147).

Υπόδειξη: Στη διαδικασία προβολής ψηφίου, προσθέστε φάσεις, έτσι ώστε αν *digit=1* να δίνεται και εντολή εκτύπωσης του συμβόλου +. Επιπλέον, στη διαδικασία υπολογισμού (*compute='1'*) πριν την εκτύπωση των ψηφίων του αποτελέσματος εκτυπώστε ένα =, δίνοντας αντίστοιχη εντολή στην LCD. Για την έναρξη νέας εισόδου, προσθέστε ένα σήμα-σημαία (πχ *restart*) που θα τίθεται κάθε φορά που θα πατιέται το πλήκτρο *btn_north*. Δημιουργήστε μία αντίστοιχη διαδικασία που θα εκτελείται όταν ισχύει *restart='1'* και εκτελέστε τις απαιτούμενες ενέργειες με τη χρήση φάσεων. Οι ενέργειες που απαιτούνται είναι μία εντολή

καθαρισμού οθόνης και μία εντολή εγγραφής διεύθυνσης (εγγραφή διεύθυνσης 0). Μη ξεχάσετε να αρχικοποιήσετε τον καταχωρητή *digit* στο 0.

ε) Τροποποιήστε το παράδειγμα Γ, ώστε αντί για πρόσθεση να υπολογίζει τον μέγιστο κοινό διαιρέτη των δύο 8-bit αριθμών. Χρησιμοποιείτε το κύκλωμα που σχεδιάσατε στην Άσκηση 3.

Υπόδειξη: Προσθέστε στην οντότητα *gcd_calc* της άσκησης 3 το σήμα *data_en* της FSM που γίνεται '1' όταν τελειώσει ο υπολογισμός. Δημιουργήστε μία υπόσταση (instantiation) της *gcd_calc* στην top level οντότητα της σχεδίασης του παραδείγματος Γ. Στην είσοδο *clk* συνδέστε το ρολόι του συστήματος *clk* (50MHz), ενώ στην *rst* συνδέστε ένα σήμα *gcd_rst*. Στις εισόδους *x_i*, *y_i*, και *data_o* συνδέστε αντίστοιχα σήματα. Στη διαδικασία *compute* προσθέστε φάσεις ώστε αρχικά να θέσετε τα σήματα *x_i*, *y_i*, ακολούθως να κάνετε *gcd_rst*='0' για έναν κύκλο, και στη συνέχεια να περιμένετε μέχρι το σήμα *data_en* να γίνει '1'. Η αναμονή μπορεί να γίνει περιμένοντας στην ίδια φάση, και μόνο όταν γίνει το *data_en* '1' να προχωράμε στην επόμενη (when *K* => *phase* <= *k*; if cond then *phase* <= *K*+1). Ανάλογα με την υλοποίηση του *gcdcalc*, προωθήστε το αποτέλεσμα (*data_o*) στον επόμενο ή μεθεπόμενο κύκλο. Στις ακόλουθες φάσεις εκτυπώστε το αποτέλεσμα (2 hex ψηφία) στην LCD. Υποθέστε ότι ο χρήστης δε δίνει μηδενικές εισόδους για τους ακέραιους, και ότι ο υπολογισμός του ΜΚΔ διαρκεί πολύ μικρό χρόνο σε σχέση με τη συχνότητα αποστολής εντολών στην LCD.

Χρήση μνήμων blockRAM του FPGA

Στο ακόλουθο παράδειγμα κάνουμε χρήση μίας εσωτερικής μνήμης blockRAM 256*16bit (4K) του FPGA. Χρησιμοποιούμε τα slide switches και τα πλήκτρα πίεσης και την LCD οθόνη για την ανάγνωση/εγγραφή. Συγκεκριμένα, δίνουμε την 8bit διεύθυνση με δύο πατήματα του πλήκτρου πίεσης *btn_east* (δειγματολειτουργούντα οι αντίστοιχες τιμές των slide switches). Επόμενο πάτημα του πλήκτρου, εμφανίζει τα περιεχόμενα της μνήμης στη θέση αυτή (4 hex ψηφία). Στη συνέχεια, με πάτημα του πλήκτρου *btn_west* σχηματίζεται μία νέα τιμή προς αποθήκευση. Με 4 πατήματα, ανανεώνεται η τιμή προς αποθήκευση, ενώ με ένα πέμπτο αποθηκεύεται στην μνήμη. Με reset (*btn_south*) οι τιμές στην blockRAM διατηρούνται. Δοκιμάστε να δείτε το περιεχόμενο της θέσης που τροποποιήσατε.

Η blockRAM περιγράφεται όπως φαίνεται στο κώδικα. Πρόκειται για σύγχρονη RAM με μία πόρτα εγγραφής/ανάγνωσης και μία πόρτα ανάγνωσης. Από τον τρόπο που την περιγράφουμε ο synthesizer καταλαβαίνει ότι πρέπει να χρησιμοποιήσει blockRAM. Θα μπορούσαμε να κάνουμε instantiation της μνήμης που θέλαμε από τη βιβλιοθήκη UNISIM της XILINX. Επίσης θα μπορούσαμε να δημιουργήσουμε περιγραφή μνήμης από το εργαλείο XILINX core generator.

Ως προς το χειρισμό του πληκτρολογίου και την εκτέλεση των λειτουργιών κάνουμε ότι και στο προηγούμενο παράδειγμα. Όταν γίνει αίτηση για ανάγνωση, γράφουμε τον καταχωρητή διεύθυνσης *ram_addr_rd* με τη διεύθυνση θέσης προς ανάγνωση, και 1 κύκλο μετά το περιεχόμενο βρίσκεται στον καταχωρητή *ram_data_out*. Αντίστοιχα στην εγγραφή, γράφουμε τον καταχωρητή διεύθυνσης *ram_addr_wr* και το καταχωρητή δεδομένων *ram_data_in* και μετά από ένα κύκλο εγγράφεται η μνήμη.

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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity write2lcd is
port(button, clk, btn_east, btn_west: in std_logic;
      switches: in std_logic_vector(3 downto 0);
      sf_d: out std_logic_vector(11 downto 8);
      lcd_e, lcd_rs, lcd_rw: out std_logic;
      led: out std_logic);
end write2lcd;

architecture behavioral of write2lcd is

component lcdctrl
port(rst, clk: in std_logic;
     --control interface
```

```

    req: in std_logic;
    reqcommand: in std_logic_vector(9 downto 0);
    resp: out std_logic;
    lcd_ready_o: out std_logic;
    --interface with the lcd -fpga pins
    sf_d: out std_logic_vector(11 downto 8);
    lcd_e, lcd_rs, lcd_rw: out std_logic);
end component;

subtype lcd_command is std_logic_vector(9 downto 0);

type word_array is array(natural range <>) of std_logic_vector(15 downto 0);
signal ram_data: word_array(0 to 255);
signal ram_out: std_logic_vector(7 downto 0);
signal ram_data_in: std_logic_vector(15 downto 0);
signal ram_data_out: std_logic_vector(15 downto 0);
signal ram_we: std_logic;
signal ram_addr_rd: std_logic_vector(7 downto 0);
signal ram_addr_wr: std_logic_vector(7 downto 0);

signal req, resp, lcd_ready: std_logic;
signal reqcommand: lcd_command;
signal rst: std_logic;

signal counter: natural range 0 to 2**17-1; counter for io polling
signal lcd_counter: natural range 0 to 2**24-1;
signal keypressed, init_done, display_addr_digit,
        display_data_digit, display_data, perform_write: std_logic;
signal phase: natural range 0 to 16;
signal digit: natural range 0 to 6;

begin

process (clk)
begin
    if (clk'event and clk = '1') then
        if (ram_we = '1') then

```

```

        ram_data(conv_integer(ram_addr_wr)) <= ram_data_in;
    end if;
    ram_data_out <= ram_data(conv_integer(ram_addr_rd));
end if;
end process;

```

```

lcdctrl0: lcdctrl port map(rst => rst, clk=>clk,
    --control interface
    req =>req, reqcommand => reqcommand,
    resp =>resp, lcd_ready_o =>lcd_ready,
    --interface with the lcd -fpga pins
    sf_d => sf_d, lcd_e => lcd_e, lcd_rs => lcd_rs, lcd_rw =>lcd_rw);

```

```

rst<=not button;
led<=lcd_ready;
process(clk)
function encode(p: in std_logic_vector(3 downto 0)) return std_logic_vector is
    variable a: std_logic_vector(7 downto 0);
begin
    case p is
        when "0000" => a:="00110000";--0
        when "0001" => a:="00110001";--1
        when "0010" => a:="00110010";--2
        when "0011" => a:="00110011";--3
        when "0100" => a:="00110100";--4
        when "0101" => a:="00110101";--5
        when "0110" => a:="00110110";--6
        when "0111" => a:="00110111";--7
        when "1000" => a:="00111000";--8
        when "1001" => a:="00111001";--9
        when "1010" => a:="01000001";--a
        when "1011" => a:="01000010";--b
        when "1100" => a:="01000011";--c
        when "1101" => a:="01000100";--d
        when "1110" => a:="01000101";--e
        when "1111" => a:="01000110";--f
        when others => null;
    end case;
end process;

```

```

        return a;
    end encode;
    variable tmp_digit: std_logic_vector(3 downto 0);
    begin
        if clk'event and clk='1' then
            if rst='0' then
                counter<=0;
                req<='0';
                digit<=0;
                keypressed<='0';
                init_done<='0';
                display_addr_digit<='0';
                display_data_digit<='0';
                display_data<='0';
                perform_write<='0';
                phase<=0;
            else
                counter<=counter+1;

                if init_done='0' then
                    --some display init
                    phase<=phase+1;
                    case phase is
                        when 0 => phase<=0;
                            if lcd_ready='1' then --wait
                                phase<=1;
                            end if;
                        when 1 => phase<=1;

                            --just to see the init msg
                            if lcd_counter=2**24-1 then
                                phase<=2;
                            end if;
                        when 2 =>

                            req<='1';
                        when 3 => req<='0'; lcd_counter<=0;
                    end case;
                end if;
            end if;

            for lcd initialization
                lcd_counter<=(lcd_counter+1) mod 2**24;
            reqcommand<="0000000001";--clear display
        end if;
    end;

```

```

when 4 => phase<=4;
lcd_counter<=(lcd_counter+1) mod 2**18;
if lcd_counter=2**18-1 then
    phase<=5;
end if;
when 5 =>
reqcommand<="0010000000"; -- Set DD RAM address 0
    req<='1';
when 6 => req<='0'; lcd_counter<=0;
when 7 => phase<=7;
lcd_counter<=(lcd_counter+1) mod 2**18;
if lcd_counter=2**18-1 then
    init_done<='1';
end if;
when others =>null;
end case;
end if;

--display address digit
if keypressed='1' and display_addr_digit='1' then
    phase<=phase+1;
    case phase is
        when 0 =>
            if digit=0 then

tmp_digit:=ram_addr_rd(7 downto 4);

            else

tmp_digit:=ram_addr_rd(3 downto 0);

            end if;

reqcommand<="10"&encode(tmp_digit);
--write data(digit);
    req<='1';
    when 1 => req<='0'; digit<=digit+1;

lcd_counter<=0;
    when 2 => phase<=2;

lcd_counter<=lcd_counter+1;
if lcd_counter=2**24-1 then

```

```

keypressed<='0';

display_addr_digit<='0';

                                end if;
                                when others =>null;
                                end case;
                                end if;

--display data digits
if keypressed='1' and display_data='1' then
    phase<=phase+1;
    case phase is
        when 0 =>

                                when 1 =>
                                reqcommand<="10"&encode(ram_data_out(15 downto 12));
                                req<='1';
                                when 2 => req<='0'; lcd_counter<=0;
                                when      3      =>      phase<=3;

lcd_counter<=(lcd_counter+1) mod 2**18;

                                if lcd_counter=2**18-1 then
                                phase<=4;
                                end if;
                                when      4      =>

reqcommand<="10"&encode(ram_data_out(11 downto 8));

                                req<='1';
                                when 5 => req<='0'; lcd_counter<=0;
                                when      6      =>      phase<=6;

lcd_counter<=(lcd_counter+1) mod 2**18;

                                if lcd_counter=2**18-1 then
                                phase<=7;
                                end if;
                                when      7      =>

reqcommand<="10"&encode(ram_data_out(7 downto 4));

                                req<='1';
                                when 8 => req<='0'; lcd_counter<=0;
                                when      9      =>      phase<=9;

lcd_counter<=(lcd_counter+1) mod 2**18;

```



```

        if lcd_counter=2**18-1 then
            phase<=10;
        end if;
    when 10 =>
        reqcommand<="10"&encode(ram_data_out(3 downto 0));
        req<='1';
        when 11 => req<='0'; lcd_counter<=0;
        when 12 => phase<=12;

        lcd_counter<=(lcd_counter+1) mod 2**18;

        if lcd_counter=2**18-1 then
            phase<=13;
        end if;
    when 13 =>
        reqcommand<="0010000010";
        -- Set DD RAM address 2 (1st data
        digit)
        req<='1';
        when 14 => req<='0'; lcd_counter<=0;
        when 15 => phase<=15;

        lcd_counter<=(lcd_counter+1) mod 2**18;

        if lcd_counter=2**18-1 then
            phase<=16;
        end if;
        when 16 => phase<=16;
        lcd_counter<=(lcd_counter+
        1) mod 2**24;
        --for io
        if

        lcd_counter=2**24-1 then

            display_data<='0';

            keypressed<='0';

            end if;
        when others =>null;
    end case;
end if;

```

```

--display data digit
if keypressed='1' and display_data_digit='1' then
    phase<=phase+1;
    case phase is
        when 0 =>
            if digit=2 then

tmp_digit:=ram_data_in(15 downto 12);

                                                    elsif digit=3 then

tmp_digit:=ram_data_in(11 downto 8);

                                                    elsif digit=4 then

tmp_digit:=ram_data_in(7 downto 4);

                                                    elsif digit=5 then

tmp_digit:=ram_data_in(3 downto 0);

                                                    end if;

reqcommand<="10"&encode(tmp_digit);

                                                    --write data(digit);
                                                    req<='1';
when 1 => req<='0'; digit<=digit+1;

lcd_counter<=0;

when 2 => phase<=2;

lcd_counter<=lcd_counter+1;

                                                    if lcd_counter=2**24-1 then
                                                        keypressed<='0';

display_data_digit<='0';

                                                    end if;
                                                    when others =>null;
    end case;
end if;

if keypressed='1' and perform_write='1' then
    phase<=phase+1;
    case phase is

```

```

ram_addr_wr<=ram_addr_rd;

when 0 =>

when 1 => ram_we<='1';
when 2 => ram_we<='0';
when 3 => phase<=3;

lcd_counter<=lcd_counter+1;

if lcd_counter=2**24-1 then
    keypressed<='0';
    perform_write<='0';
end if;
when others =>null;
end case;
end if;

if counter=2**17-1 and keypressed='0' and init_done='1'
then
    --check inputs
    if btn_east='1' then
        if digit=0 then
            ram_addr_rd(7 downto
4)<=switches;

            display_addr_digit<='1';
        elsif digit=1 then
            ram_addr_rd(3 downto
0)<=switches;

            display_addr_digit<='1';
        elsif digit=2 then
            display_data<='1';
            display_addr_digit<='0';
        end if;
        keypressed<='1';
        phase<=0;
    elsif btn_west='1' then
        if digit=2 then
            ram_data_in(15 downto
12)<=switches;

            display_data_digit<='1';
        elsif digit=3 then

```

```

ram_data_in(11 downto
8)<=switches;

display_data_digit<='1';
elsif digit=4 then
ram_data_in(7 downto
4)<=switches;

display_data_digit<='1';
elsif digit=5 then
ram_data_in(3 downto
0)<=switches;

display_data_digit<='1';
elsif digit=6 then
perform_write<='1';
end if;
keypressed<='1';
phase<=0;
end if;
end if;
end if;
end process;
end behavioral;

```

UCF αρχείο

```

NET "LCD_E" LOC = "M18" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;
NET "LCD_RS" LOC = "L18" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;
NET "LCD_RW" LOC = "L17" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;
NET "SF_D<8>" LOC = "R15" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;
NET "SF_D<9>" LOC = "R16" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;
NET "SF_D<10>" LOC = "P17" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;
NET "SF_D<11>" LOC = "M15" | IOSTANDARD = LVCMOS33 | DRIVE = 4 |
SLEW = SLOW ;

```

```

NET "CLK" LOC = "C9" | IOSTANDARD = LVCMOS33 ;
NET "CLK" PERIOD = 20.0ns HIGH 40%;

NET "BUTTON" LOC = "K17" | IOSTANDARD = LVTTTL | PULLDOWN ;
#btn_south
NET "BTN_EAST" LOC = "H13" | IOSTANDARD = LVTTTL | PULLDOWN ;
#NET "BTN_NORTH" LOC = "V4" | IOSTANDARD = LVTTTL | PULLDOWN ;
NET "BTN_WEST" LOC = "D18" | IOSTANDARD = LVTTTL | PULLDOWN ;

#led<0>
NET "LED" LOC = "F12" | IOSTANDARD = LVTTTL | SLEW = SLOW | DRIVE
= 8 ;

NET "SWITCHES<0>" LOC = "L13" | IOSTANDARD = LVTTTL | PULLUP ;
NET "SWITCHES<1>" LOC = "L14" | IOSTANDARD = LVTTTL | PULLUP ;
NET "SWITCHES<2>" LOC = "H18" | IOSTANDARD = LVTTTL | PULLUP ;
NET "SWITCHES<3>" LOC = "N17" | IOSTANDARD = LVTTTL | PULLUP ;

```

Μετά τη σύνθεση του παραπάνω κυκλώματος, παρατηρήστε στο synthesis report τι μνήμη αναγνώρισε ο synthesizer από την περιγραφή μας. Πόσα slices χρησιμοποιούνται; Σε τι συχνότητα εκτιμάται ότι θα λειτουργεί το κύκλωμα. Παρατηρήστε τώρα το place and route report. Πόσα slices χρησιμοποιούνται; Σε τι συχνότητα λειτουργεί το κύκλωμα; Γιατί διαφέρουν τα νούμερα αυτά από το synthesis report;

Ζητούμενα:

α) Τροποποιήστε το παράδειγμα, ώστε κάθε φορά που πατάμε το πλήκτρο *btn_north*, να καθαρίζει η οθόνη και να δεχόμαστε νέα είσοδο διεύθυνσης. Προαιρετικά, εκτυπώνετε και τους χαρακτήρες '[', ']', '=' στην έξοδο. (πχ. [1F]=02AC).

β) (*προαιρετικό*) «Τροφοδότηση Συστολικού Πολλαπλασιαστή από blockRAM»

Στο ερώτημα αυτό ζητείται να γίνει ενσωμάτωση & τροφοδότηση του συστολικού πολλαπλασιαστή της Άσκησης 5 (με παραμέτρους $NA=NB=8$). Για το σκοπό αυτό θα τροποποιηθεί κατάλληλα το παράδειγμα Α. Συγκεκριμένα, θα προστεθεί άλλη μία blockRAM ίδιων διαστάσεων ($256 \times 16\text{bit}$). Κάθε γραμμή της

πρώτης θα περιέχει στο MS byte τον πολλαπλασιαστή και στο LS byte τον πολλαπλασιαστέο. Η αντίστοιχη γραμμή της δεύτερης θα πρέπει να περιέχει το γινόμενο τους, όταν τελειώσει ο υπολογισμός. Το κύκλωμα θα επιτρέπει ανάγνωση/εγγραφή της πρώτης blockRAM, όπως και στο παράδειγμα Α. Επιπλέον, με το που πατηθεί το πλήκτρο πίεσης *ROT_CENTER* (πιέζοντας προς τα κάτω τον περιστροφικό διακόπτη), θα ξεκινάει η τροφοδότηση του πολλαπλασιαστή με 1 ζεύγος τελεστών ανά κύκλο (συχνότητα 50MHz). Όταν τελειώσει ο υπολογισμός (μετά από περίπου 256+22 κύκλους), η δεύτερη blockRAM θα περιέχει τα γινόμενα. Κατά την ένδειξη του περιεχομένου συγκεκριμένης διεύθυνσης της πρώτης blockRAM στην LCD, θα εκτυπώνουμε και το περιεχόμενο της αντίστοιχης θέσης της δεύτερης blockRAM (που θα περιέχει το γινόμενο, αν έχει προηγηθεί υπολογισμός). (Για έλεγχο της λειτουργίας, είναι βολικό να έχει ενσωματωθεί και η λειτουργικότητα του ερωτήματος α)

Υπόδειξη: Στο παράδειγμα Α, η δίπορτη μνήμη χρησιμοποιείται για εγγραφή ή για ανάγνωση μίας θέσης μνήμης. Η λειτουργία αυτή θα μπορούσε να γίνει και με μία πόρτα εγγραφής/ανάγνωσης (στην περιγραφή απλά το σήμα διεύθυνσης ανάγνωσης θα ταυτιζόταν με το σήμα διεύθυνσης εγγραφής). Στην περίπτωση μας, θα χρησιμοποιηθεί δίπορτη blockRAM με μία πόρτα εγγραφής/ανάγνωσης και μία πόρτα μόνο ανάγνωσης για την πρώτη μνήμη. Η πόρτα εγγραφής/ανάγνωσης θα χρησιμοποιηθεί για την είσοδο από πληκτρολόγιο/έξοδο στην LCD, ενώ η πόρτα μόνο ανάγνωσης θα χρησιμοποιηθεί για διάσχιση της blockRAM (256 θέσεις) και τροφοδότηση του πολλαπλασιαστή. Ακολουθεί η περιγραφή μίας τέτοιας μνήμης:

```
process (clk)
begin
    if (clk'event and clk = '1') then
        if (ram_we = '1') then
            ram_data(conv_integer(ram_addr1)) <= ram_data_in1;
        end if;
        ram_data_out1 <= ram_data(conv_integer(ram_addr1));
        ram_data_out2 <= ram_data(conv_integer(ram_addr2));
    end if;
end process;
```

Η πόρτα 1 είναι ανάγνωσης/εγγραφής, ενώ η 2 μόνο ανάγνωσης. Για τη δεύτερη blockRAM απαιτείται μία αντίστοιχη μνήμη, μόνο που η πόρτα 1 (ανάγνωσης/εγγραφής) θα χρησιμοποιείται από το κύκλωμα ελέγχου του

πολλαπλασιαστή, ενώ η πόρτα 2 (μόνο ανάγνωσης) για την εκτύπωση στην LCD οθόνη.

Το κύκλωμα ελέγχου του πολλαπλασιαστή (μπορεί να υλοποιηθεί σαν μία μηχανή καταστάσεων) θα περιέχει δύο 8bit δείκτες στις μνήμες A και B (*pa* και *pb* αντίστοιχα). Σε κάθε κύκλο ο δείκτης *pa* θα χρησιμοποιείται ως διεύθυνση στη μνήμη A, και η 16-bit λέξη που θα εξάγεται θα τροφοδοτεί τον πολλαπλασιαστή. Ταυτόχρονα, ο *pa* θα αυξάνεται κατά 1, ώστε να δείξει στην επόμενη θέση. Θα σταματάει όταν φτάσει στην τελευταία θέση (255). Ο δείκτης *pb*, θα ξεκινάει από τη διεύθυνση 0, όταν εξαχθεί το πρώτο δεδομένο (δηλ. μετά το latency του πολλαπλασιαστή), και θα χρησιμοποιείται σαν διεύθυνση εγγραφής των γινομένων που εξάγονται 1 ανά κύκλο. Θα αυξάνεται κι αυτός κατά 1 σε κάθε κύκλο, μέχρι να φτάσει στη θέση 255. Όταν γίνει αυτό, ένα σήμα *done* μπορεί να γίνει '1', ώστε να αποτελεί την ένδειξη τέλους του υπολογισμού.

Πριν προχωρήσετε στην υλοποίηση του κυκλώματος ελέγχου του πολλαπλασιαστή, κάντε λειτουργική εξομοίωση στο περιβάλλον της ActiveHDL. Θεωρήστε αρχικοποιημένες μνήμες, και βεβαιωθείτε για τη σωστή λειτουργία του υποσυστήματος. Ακολουθώ, ενσωματώστε τη μονάδα σας στην υπόλοιπη σχεδίαση, και κάντε υλοποίηση στο FPGA.

Μπορείτε να κάνετε σύνθεση μόνο του υποσυστήματος αυτού, ώστε να δείτε σε τι συχνότητα θα μπορούσε να δουλέψει. Δείτε αν λειτουργεί πιο γρήγορα, από ότι μαζί με την υπόλοιπη σχεδίαση.