(document version 1.2)

Ιωάννης Α. Καλόμοιρος

Εισαγωγή στη γλώσσα VHDL

Τεχνολογικό Εκπαιδευτικό Ίδρυμα Σερρών,

Τμήμα Πληροφορικής και Επικοινωνιών, 2012

Το σύγγραμμα αυτό προορίζεται αποκλειστικά για χρήση από τους σπουδαστές του Τμήματος Πληροφορικής και Επικοινωνιών του ΤΕΙ Σερρών. Αποτελεί μέρος του διδακτικού υλικού του μαθήματος «Προηγμένα Ψηφιακά Συστήματα», που διδάσκεται στο ΣΤ΄εξάμηνο των σπουδών.

Απαγορεύεται ρητά κάθε ανατύπωση ή διανομή του κειμένου, με οποιοδήποτε μέσο, χωρίς την έγγραφη άδεια του συγγραφέα.

ΤΕΙ Σερρών, Τμήμα Πληροφορικής και Επικοινωνιών, 2012

1 Γενικά χαρακτηριστικά της γλώσσας VHDL

1.1 Εισαγωγή

Η VHDL είναι μία γλώσσα περιγραφής υλικού, που χρησιμοποιείται για την ανάπτυξη ολοκληρωμένων ψηφιακών κυκλωμάτων και συστημάτων. Αρχικά η δημιουργία της αποσκοπούσε στη μοντελοποίηση και στην προσομοίωση κυκλωμάτων, γι' αυτό πολλά χαρακτηριστικά της γλώσσας έχουν σκοπό την προσομοίωση των λειτουργιών. Αργότερα, η γλώσσα χρησιμοποιήθηκε και ως εργαλείο *σύνθεσης*. Κατά τη σύνθεση, ο μεταγλωττιστής συνθέτει ένα πραγματικό κύκλωμα που ανταποκρίνεται με ακρίβεια στη λογική και χρονική περιγραφή, την οποία μοντελοποιεί ο κώδικας.

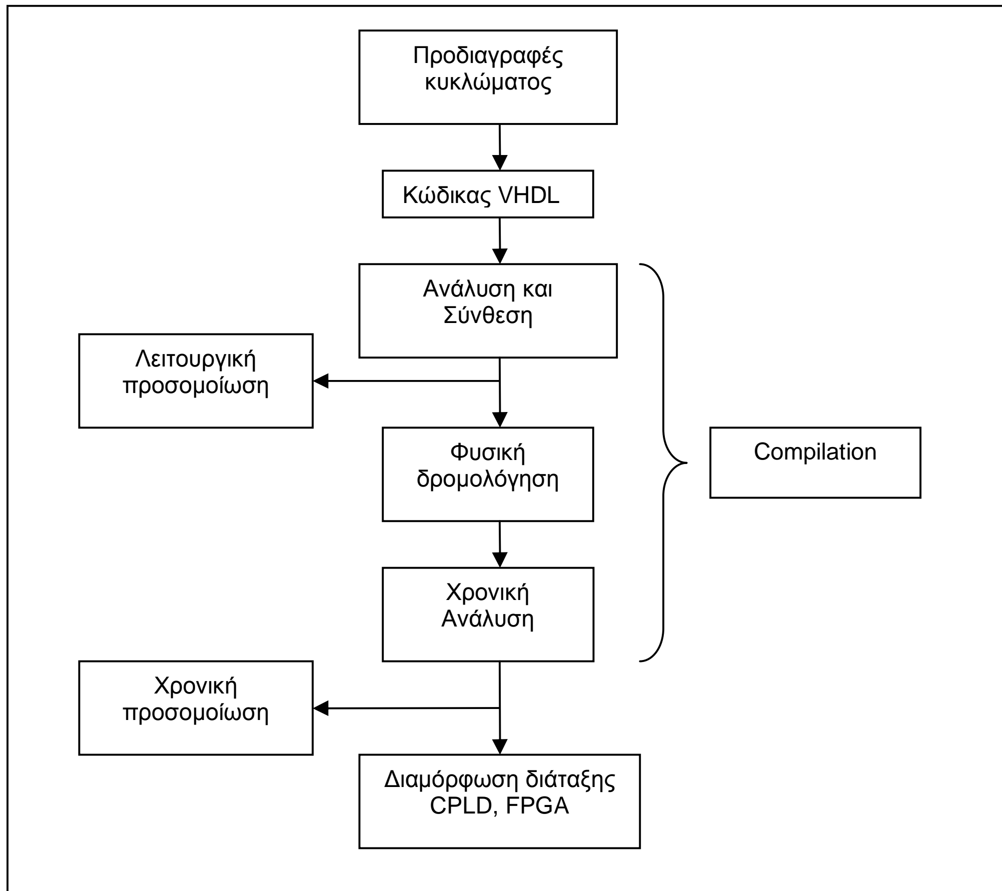
Ο όρος VHDL είναι συντόμευση των λέξεων VHSIC Hardware Description Language, όπου VHSIC σημαίνει Very High Speed Integrated Circuit. Η γλώσσα αυτή αναπτύχθηκε στις αρχές της δεκαετίας του 1980 με χρηματοδότηση από το υπουργείο Άμυνας των ΗΠΑ και έγινε πρότυπη το 1987 από το ινστιστούτο IEEE ως *IEEE 1076*. Αργότερα, δημιουργήθηκε μια βελτιωμένη έκδοσή της, η *IEEE 1164* (1993). Ακολούθησαν και νεότερες αναβαθμίσεις του προτύπου, όπως η *IEEE 1076-2002* και *IEEE 1076-2008*.

Ο κώδικας VHDL αποτελεί τον βασικό τύπο αρχείου που δέχονται ως είσοδο τα λογισμικά ψηφιακής σχεδίασης κυκλωμάτων (Computer Aided Design ή CAD), για τη δημιουργία σύνθετων ολοκληρωμένων κυκλωμάτων. Η γλώσσα VHDL χρησιμοποιείται ευρύτατα για την περιγραφή και υλοποίηση ψηφιακών συστημάτων σε προγραμματιζόμενες λογικές διατάξεις, τύπου CPLDs (Complex Programmable Logic Devices-Σύνθετες προγραμματιζόμενες λογκές διατάξεις) και FPGAs (Field Programmable Gate Arrays-Διατάξεις πυλών προγραμματιζόμενες στο πεδίο). Επίσης, έχει καθιερωθεί σαν ένα πρότυπο (standard) στη σχεδίαση ηλεκτρονικών κυκλωμάτων ASICs (Application Specific Integrated Circuits). Η καθιέρωση της ως πρότυπο μας διαβεβαιώνει ότι και οι επόμενες εκδόσεις εργαλείων σχεδίασης θα υποστηρίζουν το πρότυπο αυτό. Έτσι, ένα κύκλωμα που περιγράφηκε και αναπτύχθηκε με τα σημερινά εργαλεία σχεδίασης θα είναι μεταφέρσιμο μελλοντικά σε νέα εργαλεία σχεδίασης, με ελάχιστες ή καθόλου αλλαγές.

Ας αναφερθεί ότι εκτός από τη γλώσσα VHDL ευρύτατη χρήση και αποδοχή έχει βρει διεθνώς και η γλώσσα περιγραφής υλικού Verilog.

1.2 Ροή σχεδίασης

Στο παρακάτω σχήμα 1.1 φαίνεται μια γενική ροή των βημάτων που ακολουθεί ο σχεδιαστής λογικών κυκλωμάτων όταν εργάζεται με κώδικα VHDL. Οι λειτουργίες που περιγράφονται υλοποιούνται με τη βοήθεια σχεδιαστικών εργαλείων CAD (βλέπε επόμενη παράγραφο). Αρχικά τίθενται οι προδιαγραφές του κυκλώματος, που περιγράφουν με ακρίβεια την επιθυμητή συμπεριφορά. Στη συνέχεια, το κύκλωμα χωρίζεται στα μέρη του ακολουθώντας κανόνες ιεραρχικής δομημένης σχεδίασης. Το κάθε μέρος του κυκλώματος περιγράφεται με κώδικα VHDL, έτσι ώστε τα μέρη αυτά να μπορούν να συνδυαστούν λειτουργικά μεταξύ τους.



Σχήμα 1.1 Βασική ροή εργασιών κατά τη σχεδίαση με τη γλώσσα VHDL

Για τη συγγραφή του κώδικα χρησιμοποιείται συνήθως ο ASCII Editor που ενσωματώνεται στα εργαλεία σχεδίασης. Ο κώδικας αποθηκεύεται με επέκταση *.vhd*. Για μικρές σχεδιάσεις η περιγραφή αποτελείται από ένα αρχείο, ενώ για μεγαλύτερες, από πολλά αρχεία (design units). Το ανώτερο ιεραρχικά αρχείο ονομάζεται «οντότητα ανώτερου επιπέδου» (top-level entity), ενώ τα υπόλοιπα αρχεία συνδέονται με αυτό μέσω κατάλληλων δηλώσεων και αναφορών. Το σύνολο των αρχείων σχεδίασης ενός συστήματος, όπως και αυτά που δημιουργούνται αργότερα για την προσομοίωση και τη διαμόρφωση (configuration), αποτελούν ένα ολοκληρωμένο project μέσα στο περιβάλλον σχεδίασης και αποθηκεύονται σε κοινό φάκελο εργασίας..

Μετά τη συγγραφή του κώδικα ακολουθεί η μεταγλώττιση (compilation). Το πρώτο στάδιο της μεταγλώττισης ονομάζεται *ανάλυση*. Ο αναλυτής είναι ένα εργαλείο που επεξεργάζεται τον κώδικα για συντακτικά λάθη και επιστρέφει σχόλια, που καθοδηγούν το χρήστη στη διόρθωση των λαθών.

Η *σύνθεση* (synthesis) είναι η πιο σημαντική διαδικασία στη συνολική ροή. Κατά τη σύνθεση, ο compiler σχεδιάζει το κύκλωμα που περιγράφει ο κώδικας, σύμφωνα με τους κανόνες της τεχνολογίας της διάταξης την οποία προορίζεται να διαμορφώσει το σχέδιό μας. Άλλο θα είναι το αποτέλεσμα της σύνθεσης αν πρόκειται να διαμορφώσουμε ένα κύκλωμα CPLD και άλλο αν πρόκειται να διαμορφώσουμε ένα FPGA. Ο λόγος είναι ότι τα κυκλώματα αυτά

περιέχουν το καθένα τις δικές τους διαφορετικές βαθμίδες για τη δημιουργία λογικών συναρτήσεων. Ένα CPLD θα πρέπει να υλοποιήσει τις λογικές συναρτήσεις με βάση λογικούς πίνακες πυλών AND και OR (Logic Arrays), ενώ ένα FPGA χρησιμοποιεί τους λεγόμενους πίνακες αναφοράς (Look-up Tables-LUTs). Ως αποτέλεσμα της σύνθεσης, το εργαλείο σχεδίασης μπορεί να εκτελέσει ένα πρώτο στάδιο «λειτουργικής» προσομοίωσης (functional simulation), στο οποίο δεν ενσωματώνονται ακόμη οι χρονικοί περιορισμοί του τελικού κυκλώματος.

Κατά το στάδιο της *φυσικής τοποθέτησης και δρομολόγησης* (place and route), κάθε λογική δομή η οποία έχει παραχθεί από τη σύνθεση βρίσκει τη φυσική της αντιστοίχηση σε μια λογική βαθμίδα μέσα στη διάταξη. Αυτή η πληροφορία θέσης οδηγεί στον υπολογισμό των χρονικών καθυστερήσεων, που παράγονται κατά τις διαδρομές των σημάτων. Έτσι, μετά το στάδιο της φυσικής δρομολόγησης είναι πλέον δυνατό να ακολουθήσει η «χρονική προσομοίωση» (timing simulation) της σχεδίασης.

Εφόσον τα παραπάνω στάδια έχουν υλοποιηθεί με επιτυχία και τα αποτελέσματα της προσομοίωσης αποδεικνύουν ότι το κύκλωμα λειτουργεί σύμφωνα με τις προδιαγραφές, τότε πλέον μπορούμε να διαμορφώσουμε την διάταξη-στόχο, καταφορτώνοντας το αρχείο διαμόρφωσης. Το αρχείο αυτό προκύπτει κατά το τελικό στάδιο της μεταγλώττισης. Μετά τη *διαμόρφωση* (configuration) η διάταξη μπορεί να δεχτεί σήματα εισόδου και να παράγει τα αναμενόμενα σήματα εξόδου. Με τα κατάλληλα εργαλεία σχεδίασης είναι δυνατό να προκύψουν και οι μάσκες που θα χρησιμοποιηθούν για τη δημιουργία ενός ολοκληρωμένου κυκλώματος ASIC.

1.3 Εργαλεία λογισμικού για την ηλεκτρονική σχεδίαση

Τα βασικά εργαλεία (Electronic Design Automation-EDA-tools) που χρησιμοποιούνται στην ηλεκτρονική βιομηχανία για την αυτοματοποίηση της σχεδίασης λογικών κυκλωμάτων είναι τα ακόλουθα:

- Quartus II της Altera, που είναι ολοκληρωμένο εργαλείο σύνθεσης, γραφικής προσομοίωσης και προγραμματισμού
- ISE της Xilinx., που αποτελείται από το XST για σύνθεση και το IDE Simulator για προσομοίωση
- Leonardo Spectrum της Mentor Graphics, για σύνθεση
- ModelSim της Mentor Graphics για προσομοίωση

Στο κείμενο που ακολουθεί γίνεται χρήση του λογισμικού Quartus II v. 9.0 της Altera και του εργαλείου προσομοίωσης ModelSim της εταιρίας Mentor Graphics. Στα παραρτήματα περιέχονται σύντομα βοηθήματα για την καθοδήγηση του αρχάριου στη χρήση των δύο αυτών εργαλείων.

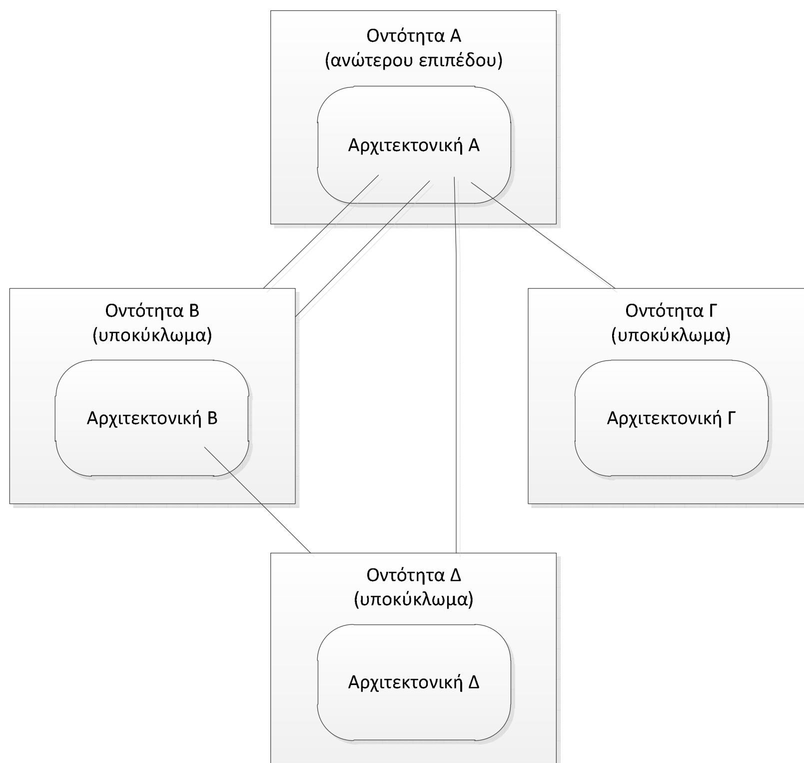
1.4 Χαρακτηριστικά της γλώσσας VHDL

Η VHDL υπακούει στις αρχές των *παράλληλων* γλωσσών και δεν είναι τυχαίο ότι έχει τις ρίζες της στην Ada, που χρησιμοποιείται για να προγραμματίσει παράλληλες διεργασίες. Ταυτόχρονα, υπακούει στις αρχές του *δομημένου προγραμματισμού*, που δίνει τη δυνατότητα της σχεδίασης *ιεραρχικών κυκλωμάτων*. Τέλος, περιέχει τις αρχές του *συγχρονισμού* και του *χρονισμού*, που είναι εγγενείς στα κυκλώματα. Ένα από τα χαρακτηριστικά της VHDL είναι ότι *μοντελοποιεί* με

ακρίβεια τόσο τις λειτουργίες του κυκλώματος, όσο και τους χρόνους κατά τους οποίους οι λειτουργίες παράγουν αποτελέσματα.

Η VHDL διαφέρει από τις συμβατικές γλώσσες κατά το ότι δεν προορίζεται να περιγράψει λειτουργίες που εκτελούνται σειριακά, η μία μετά την άλλη. Κάθε πρόταση ή τμήμα κώδικα περιγράφει λειτουργίες, οι οποίες παράγουν αποτελέσματα σε συγχρονισμό με άλλες λειτουργίες,. Τα αποτελέσματα της προσομοίωσης παράγονται με βάση αυστηρές χρονικές προδιαγραφές σε διάφορα σημεία του κυκλώματος και εν τέλει στις εξόδους. Με την έννοια αυτή, είναι δυνατό ένα τμήμα κώδικα να μπορεί να γραφεί σε οποιοδήποτε σημείο του προγράμματος, αφού παράγει αποτελέσματα ανεξαρτήτως της σειράς του. Στο τέλος, ο compiler θα συνθέσει κυκλώματα που θα υλοποιούν στην πράξη τις σύγχρονες λειτουργίες που περιγράφει ο κώδικας.

Η VHDL μπορεί να περιγράψει τόσο σύγχρονες όσο και ακουλουθιακές λογικές λειτουργίες.Οι διεργασίες που περιγράφονται στη VHDL παράγουν αποτέλεσμα είτε ταυτόχρονα με την εφαρμογή των εισόδων, όπως συμβαίνει στα συνδυαστικά κυκλώματα, είτε σε συγχρονισμό με συμβάντα (π.χ. παλμούς ρολογιού), όπως συμβαίνει στα ακολουθιακά κυκλώματα. Για το λόγο αυτό η σύνταξη του κώδικα μπορεί να γίνει με *σύγχρονες* δομές (concurrent code) ή και με *ακολουθιακές* δομές (sequential code).



Σχήμα 1.2 Ιεραρχική σύνδεση οντοτήτων στη δομημένη σχεδίαση με τη γλώσσα VHDL

Ο ιεραρχικός τρόπος σχεδίασης γίνεται δυνατός στη VHDL εξαιτίας της δομής του κώδικα, που διακρίνει ανάμεσα στην περιγραφή του κυκλώματος ως βαθμίδα (block) και στη λειτουργική του περιγραφή. Τα δύο αυτά μέρη του κώδικα αναφέρονται ως «οντότητα» και «αρχιτεκτονική» (βλέπε επόμενη παράγραφο). Έχοντας σχεδιάσει ένα κύκλωμα σε VHDL, είναι δυνατό να το ενσωματώνουμε σε πιο περίπλοκα κυκλώματα, κάνοντας απλώς αναφορά στην «οντότητά» του, χωρίς να χρειάζεται να επαναλαμβάνουμε την περιγραφή της αρχιτεκτονικής.

Όπως θα περιγραφεί παρακάτω, μπορούμε να παράγουμε «στιγμιότυπα» (instances) ενός κυκλώματος, όσες φορές χρειάζεται, κάνοντας αναφορά σ' αυτό μέσα στην αρχιτεκτονική μιας ανώτερης ιεραρχικής βαθμίδας (σχήμα 1.2). Η δυνατότητα της επανάληψης των στιγμιοτύπων συνιστά ένα χαρακτηριστικό της γλώσσας VHDL που καθιστά την περιγραφή του υλικού (hardware) *επαναχρησιμοποιήσιμη* (reusable). Κάθε κύκλωμα που περιγράφεται μία φορά μπορεί να χρησιμοποιηθεί ξανά σαν βαθμίδα υποκυκλώματος, σε οποιοδήποτε άλλο ψηφιακό σύστημα. Με τον τρόπο αυτό, το hardware στην κωδική του περιγραφή, καθίσταται ευέλικτο όπως και το software, αφού πλέον μπορεί να διαμοιραστεί στους χρήστες σαν λογισμικό ή να χρησιμοποιηθεί για την ανάπτυξη μεγάλων συστημάτων, μέσω βιβλιοθηκών που περιλαμβάνουν βαθμίδες υποκυκλωμάτων. Τέτοιες βιβλιοθήκες υπάρχουν έτοιμες στα μεγάλα εργαλεία ψηφιακής σχεδίασης, όπως το Quartus II, αλλά μπορούν να δημιουργηθούν και από τον κάθε σχεδιαστή, ανάλογα με τις δικές του ανάγκες σχεδίασης.

1.5 Παράσταση αριθμών και χαρακτήρων στη VHDL

Τα διάφορα αντικείμενα δεδομένων (σήματα, μεταβλητές, σταθερές) που θα γνωρίσουμε στην παράγραφο 3, μεταφέρουν αριθμητική πληροφορία. Η βασική αριθμητική πληροφορία που υποστηρίζει η VHDL είναι με τη μορφή *ακεραίων* τιμών και *δυαδικών* τιμών. Για αριθμητικές πράξεις υποστηρίζονται *προσημασμένοι* και μη *προσημασμένοι* αριθμοί. Τέλος, η πληροφορία μπορεί να είναι με τη μορφή *χαρακτήρων*. Οι πραγματικοί αριθμοί (σταθερής και κινητής υποδιαστολής) υποστηρίζονται για τους σκοπούς της προσομοίωσης, όμως έχουν σημαντικούς περιορισμούς στη σύνθεση. Δεν θα τους χρησιμοποιήσουμε σ' αυτό το βιβλίο.

Οι *ακέραιοι* συνήθως παριστάνονται στη VHDL στο δεκαδικό σύστημα. Αν δεν δηλωθεί αλλιώς λαμβάνουν τιμές στην περιοχή $-(2^{31}-1)$ έως $(2^{31}-1)$. Για ειδικούς σκοπούς μπορεί να χρησιμοποιηθούν κι άλλες βάσεις για την αναπαράσταση των δεκαδικών αριθμών, από το 2 μέχρι το 16. Στην περίπτωση αυτή, κατά την αναπαράσταση του αριθμού προηγείται η βάση και ακολουθεί ο αριθμός ανάμεσα σε σύμβολα της δίεσης. Παραδείγματα ακεραίων με αναπαράσταση σε διάφορα αριθμητικά συστήματα είναι τα εξής:

- Στο δεκαδικό σύστημα: 15, 250, 4E2 ($= 4\cdot10^2$)
- Στο δυαδικό σύστημα: 2#1010# (ο δεκαδικός 10)
- Στο δεκαεξαδικό σύστημα: 16#FF# (ο δεκαδικός 255)

Οι *δυαδικές τιμές* παριστάνονται ανάμεσα σε μονά ή διπλά εισαγωγικά, αν είναι τιμές του ενός ή περισσοτέρων bits, αντίστοιχα. Τιμές των πολλών bits μπορούν να παρασταθούν και σε δεκαεξαδική μορφή, κάνοντας χρήση του συμβόλου `X` στην αρχή. Παραδείγματα δίνονται παρακάτω:

- `'0', '1', "01", "1010", b"1100", X"3F7",`

όπου `b` σημαίνει binary και `X` σημαίνει δεκαεξαδικός.

Για τους *προσημασμένους* (signed) και *μη-προσημασμένους* (unsigned) αριθμούς ισχύουν τα γνωστά από τα ψηφιακά κυκλώματα. Οι μη-προσημασμένοι αριθμοί με *N*-bits καλύπτουν την περιοχή τιμών από 0 έως 2^N-1. Ένα σύστημα που λειτουργεί με μη-προσημασμένους αριθμούς θα δέχεται εισόδους και θα παράγει εξόδους σ' αυτή την περιοχή των μη-αρνητικών αριθμών.

Οι προσημασμένοι αριθμοί με *N*-bits βρίσκονται στην περιοχή -2^{N-1} έως $2^{N-1}-1$. Οι αρνητικοί αριθμοί παριστάνονται, ως γνωστό, με το συμπλήρωμα ως προς 2, το οποίο προκύπτει αν λάβουμε το συμπλήρωμα του αριθμού ως προς 1 και προσθέσουμε τη μονάδα. Στους προσημασμένους αριθμούς, το πιο σημαντικό bit παριστάνει το πρόσημο. Ο αριθμός είναι θετικός αν το MSB είναι '0', ενώ είναι αρνητικός αν το MSB είναι '1'.

Οι *χαρακτήρες* του κώδικα ASCII μπορούν να δοθούν ως τιμές σε αντικείμενα της γλώσσας VHDL. Μπορούμε να έχουμε απλούς χαρακτήρες, όπως 'a,', 'A' ή ακολουθίες χαρακτήρων (strings) όπως "flag" ή "SLOW". Ο μεταγλωττιστής θα συνθέσει τους χαρακτήρες αποδίδοντας τιμές από τον πίνακα ASCII.

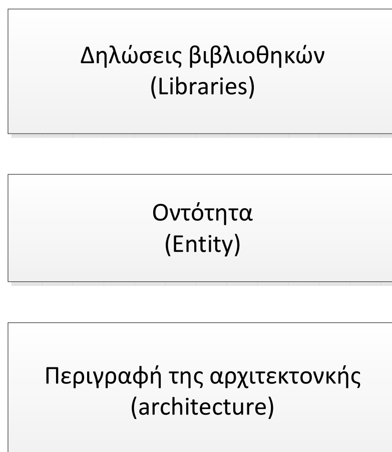
2 Δομή προγράμματος VHDL

Στο κεφάλαιο αυτό περιγράφονται με συμπυκνωμένο τρόπο τα βασικά στοιχεία της γλώσσας VHDL. Ο αρχάριος αναγνώστης καλείται να το μελετήσει προσεχτικά, χωρίς να αποθαρρύνεται από όσα δεν καταλαβαίνει, διότι το κείμενο θα επιστρέφει προοδευτικά στα σημεία αυτά, καθώς θα εξειδικεύεται. Όταν ξεκαθαρίσουν οι βασικές ιδέες, ο αναγνώστης θα πρέπει να επιστρέψει για να ξαναμελετήσει το κεφάλαιο.

Στα παρακάτω θα γίνει αναφορά στους βασικούς κανόνες σύνταξης και στη βασική δομή ενός προγράμματος VHDL. Επίσης, θα γίνει μια πρώτη διάκριση ανάμεσα στις προτάσεις σύγχρονης εκτέλεσης (concurrent statements) και στις προτάσεις ακολουθιακής εκτέλεσης (sequential statements).

2.1 Οντότητα και αρχιτεκτονική

Τα βασικά μέρη ενός κώδικα που περιγράφει κύκλωμα σε VHDL είναι η «οντότητα» (entity) και η «αρχιτεκτονική» (architecture). Στις περισσότερες περιπτώσεις, πριν το τμήμα της οντότητας θα πρέπει να δηλωθούν και κάποια πακέτα βιβλιοθηκών, που περιγράφουν τους τύπους δεδομένων ή υποκυκλώματα που χρησιμοποιεί ο σχεδιαστής. Το σχήμα 2.1 δείχνει την βασική δομή ενός αρχείου VHDL. Παρακάτω περιγράφεται με συντομία αυτή η δομή.
Οι δηλώσεις των *βιβλιοθηκών* (libraries) και των *πακέτων* (packages) επιτρέπουν τη χρήση συγκεκριμένων τμημάτων κώδικα, που έχουν ήδη δημιουργηθεί στο παρελθόν και χρησιμοποιούνται συχνά. Μια *βιβλιοθήκη*, λοιπόν, είναι συλλογή χρήσιμων τμημάτων κώδικα. Τέτοια τμήματα κώδικα είναι δηλώσεις τύπων δεδομένων, υποκυκλώματα, συναρτήσεις (functions), διαδικασίες (procedures). Όταν τοποθετούνται σε μια βιβλιοθήκη, οι κώδικες αυτοί μπορούν να χρησιμοποιηθούν ξανά, σε άλλα σχέδια, κάνοντας απλά μια δήλωση της βιβλιοθήκης στην αρχή του κώδικα. Ένα έτοιμο, τυποποιημένο πακέτο που χρησιμοποιείται πολύ συχνά είναι



Σχήμα 2.1 Βασικά τμήματα ενός αρχείου VHDL

το *std_logic_1164* της βιβλιοθήκης *ieee*, το οποίο περιγράφει τον τύπο δεδομένων `std_logic`. Οι βιβλιοθήκες δηλώνονται με τη λέξη-κλειδί `LIBRARY`, ενώ τα πακέτα εισάγονται με τη λέξη-κλειδί `USE`. Εκτός από τις τυποποιημένες βιβλιοθήκες, ο κάθε χρήστης μπορεί να δημιουργεί τις δικές του. Στην παράγραφο 3.5 θα γίνει εκτενέστερη αναφορά στις βιβλιοθήκες και στα πακέτα.

Η *οντότητα* (entity) περιγράφει το κύκλωμα ως βαθμίδα, με εισόδους και εξόδους. Περιλαμβάνει μόνο τις διασυνδέσεις που έχει το κύκλωμα με άλλες βαθμίδες αλλά αποκρύπτει τη λειτουργία του κυκλώματος. Το αναγνωριστικό όνομα που δίνει ο σχεδιαστής στην οντότητα καθώς και τα σήματα εισόδου και εξόδου είναι καθοριστικά για κάθε υλοποίηση του κυκλώματος αυτού.

Η *αρχιτεκτονική* (architecture) περιλαμβάνει όλες τις λεπτομέρειες της λειτουργίας ενός κυκλώματος. Οι εντολές και οι δηλώσεις που περιλαμβάνονται στο σώμα της αρχιτεκτονικής περιγράφουν με ακρίβεια ποιές λογικές συναρτήσεις θα υλοποιεί το κύκλωμα και τι τιμές θα λαμβάνουν τα σήματα του κυκλώματος σε κάθε χρονική στιγμή.

```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.all;
3 ------------------------------------------------------
4 ENTITY mux IS
5    PORT (x, y : IN std_logic;--input channels
6             s : IN std_logic;--selection line
7             f : OUT std_logic);--output channel
8 END mux;
9 -------------------------------------------------------
10 ARCHITECTURE behaviour OF mux IS
11 BEGIN
12    WITH s SELECT
13          f<= x WHEN '0',
14           y WHEN OTHERS;
15 END behaviour;
```

Κώδικας 2.1 Απλός κώδικας VHDL, που περιγράφει πολυπλέκτη 2:1.

Στον παραπάνω κώδικα 2.1 φαίνεται ένα πλήρες πρόγραμμα σε VHDL. Ο κώδικας περιγράφει την απλή λειτουργία ενός πολυπλέκτη 2:1. Ο αναγνώστης θα πρέπει να μελετήσει τη δομή του κώδικα έστω κι αν του διαφεύγει το νόημα ορισμένων εντολών. Παρακάτω θα αναλυθούν τα τμήματα του κώδικα. Πολλές έννοιες δεν θα γίνουν άμεσα κατανοητές, όμως θα ξεκαθαρίζουν καθώς το κείμενο θα προχωρά σε ειδικότερες αναλύσεις.

2.2 Λεκτικά στοιχεία, αναγνωριστικά και σχόλια

Ο κώδικας VHDL συντάσσεται με μια σειρά από «λεκτικά στοιχεία» (lexical elements), δηλαδή ειδικά αλφαριθμητικά, με τη βοήθεια των οποίων διατυπώνονται δηλώσεις, γίνεται ανάθεση τιμών, ονομάζονται τμήματα κώδικα, συντάσσονται εντολές και ορίζονται τελεστές πράξεων. Ανάμεσα στα λεκτικά στοιχεία διακρίνουμε τις *δεσμευμένες λέξεις* (αλλιώς λέξεις-κλειδιά) και τα *αναγνωριστικά*.

Κάθε βασικό τμήμα κώδικα αρχίζει με μια «δήλωση», που συντάσσεται με τη βοήθεια δεσμευμένων λέξεων. Π.χ. η βιβλιοθήκη δηλώνεται με την κωδική λέξη `LIBRARY`, ενώ η οντότητα και η αρχιτεκτονική δηλώνονται με τις λέξεις `ENTITY` και `ARCHITECTURE`. Άλλες δεσμευμένες λέξεις είναι οι: `signal`, `variable`, `constant`, `downto`, `end`, `generic`, `if`, `is`, `then`, `select`, `with`, `when`, `wait` κ. ά.

Τα σήματα που χρησιμοποιούνται για τη μεταφορά πληροφορίας από και προς το κύκλωμα, όπως και ανάμεσα σε σημεία του ίδιου κυκλώματος, πρέπει να δηλωθούν κι αυτά ακολουθώντας συγκεκριμένες λεκτικές διατυπώσεις. Κάθε οντότητα, σήμα, μεταβλητή κλπ, κατά τη δήλωσή της πρέπει να ονομαστεί, χρησιμοποιώντας ένα *αναγνωριστικό*. Τα αναγνωριστικά είναι ονόματα που αποδίδονται από το χρήστη, ακολουθώντας κανόνες (βλέπε παράγραφο 3.4). Έτσι, τα σήματα που δηλώνονται ως διασυνδέσεις στη δήλωση PORT της οντότητας στον κώδικα 2.1, ονομάζονται χρησιμοποιώντας ένα αναγνωριστικό αλφαριθμητικό για το καθένα:. x, y, s, και f. Όπως θα δούμε, η γλώσσα VHDL είναι ιδιαίτερα αυστηρή ως προς τις συντακτικές διατυπώσεις, καθώς οι μεταγλωττιστές της γλώσσας είναι ιδιαίτερα ανελαστικοί, ώστε να αποφεύγονται παρανοήσεις.

Πολύ σημαντικός στη σύνταξη είναι ο χαρακτήρας τέλους γραμμής (`;`) που σημειώνεται όταν ολοκληρώνεται μια εντολή της VHDL.

Τα τμήματα του κώδικα 2.1 διακρίνονται μεταξύ τους με χρήση διακεκομμένων γραμμών. Ας σημειωθεί πως οτιδήποτε ακολουθεί δύο συνεχόμενες παύλες σε μια γραμμή κώδικα, εκλαμβάνεται από τον μεταγλωττιστή ως σχόλιο και δεν εκτελείται.

2.3 Δήλωση βιβλιοθήκης και πακέτων

Το πρώτο μέρος του κώδικα 2.1 (σειρά 1) είναι η δήλωση της βιβλιοθήκης *ieee* η οποία περιλαμβάνει μια σειρά πακέτων προτυποποιημένων από το ινστιτούτο IEEE. Τα βασικά πακέτα που ανήκουν στη βιβλιοθήκη αυτή είναι:

- *std_logic_1164*, που εισάγει τον τύπο δεδομένων `std_logic` και `std_logic_vector` και επιτρέπει τη χρήση περισσοτέρων τιμών σε ένα σήμα εκτός από τις απλές τιμές `'0'` και `'1'`. Τέτοιες επιπλέον τιμές είναι η απαραίτητη κατάσταση «υψηλής εμπέδησης» (`'Z'`), η «αδιάφορη» κατάσταση (`'-'`), η «άγνωστη» κατάσταση (`'X'`) και άλλες τιμές που είναι χρήσιμες σε ορισμένες περιπτώσεις.
- *numeric_std*, που εισάγει τους τύπους `SIGNED` και `UNSIGNED`, με βάση τον τύπο `std_logic`. Το πακέτο αυτό επιτρέπει την εκτέλεση αριθμητικών πράξεων με προσημασμένους ή μη προσημασμένους αριθμούς.

Στον κώδικα 2.1 γίνεται χρήση του πακέτου *std_logic_1164*, γι' αυτό και στη συνέχεια τα σήματα που δηλώνονται στο μέρος της οντότητας ανήκουν στον τύπο δεδομένων `std_logic`.

Ένα *μη προτυποποιημένο* πακέτο που ανήκει στη βιβλιοθήκη *ieee* είναι το πακέτο *std_logic_arith*. Το πακέτο αυτό είναι shareware και ορίζει τύπους σημάτων `SIGNED` και `UNSIGNED`. Επίσης, επιτρέπει την τέλεση αριθμητικών πράξεων (`+`, `–`, `*`, `/`) ανάμεσα σε σήματα των παραπάνω τύπων. Άρα, για τη χρήση προσημασμένων αριθμών σε σήματα αριθμητικών κυκλωμάτων απαιτείται ο ορισμός ή του πακέτου *numeric_std* ή του πακέτου *std_logic_arith*. Επειδή δεν είναι ισοδύναμα, τα δύο πακέτα δεν πρέπει να χρησιμοποιούνται ταυτόχρονα.

Από το πακέτο *std_logic_arith* προκύπτουν δύο άλλα πακέτα, το *std_logic_signed* και *std_logic_unsigned*. Τα πακέτα αυτά δεν ορίζουν προσημασμένους ή μη τύπους δεδομένων. Χρησιμοποιούνται με τον τύπο δεδομένων `std_logic` και `std_logic_vector`. Απλά, δίνουν τη δυνατότητα χρήσης αριθμητικών τελεστών με σήματα του τύπου `std_logic` και `std_logic_vector`.

Όπως θα δούμε, εκτός από τις προτυποποιημένες βιβλιοθήκες, ο χρήστης μπορεί να ορίσει κι άλλες, δικές του βιβλιοθήκες.

Ας σημειωθεί ότι οι τύποι `integer`, `bit`, `bit_vector` και ορισμένοι άλλοι είναι προκαθορισμένοι στη VHDL και δεν απαιτούν δήλωση βιβλιοθήκης. Λεπτομέρειες για τα πακέτα όπου ορίζονται οι τύποι δεδομένων θα δοθούν στην παράγραφο 3.2. Αναφορά στα πακέτα που ορίζονται από το χρήστη γίνεται στο κεφάλαιο 7.

2.4 Περιγραφή οντότητας

Στο μέρος της οντότητας δηλώνονται υποχρεωτικά το όνομα της οντότητας και οι είσοδοι και οι έξοδοι του κυκλώματος, που χρησιμοποιούνται για τη διασύνδεση με άλλες βαθμίδες. Μετά την κωδική λέξη `ENTITY` ακολουθεί το όνομα της οντότητας, που μπορεί να είναι οποιαδήποτε μη-δεσμευμένη λέξη (βλέπε κανόνες ονοματοδοσίας στη παράγραφο 3.4) και η κωδική λέξη `IS`:

```
ENTITY όνομα_οντότητας IS;
```

Η δήλωση των διασυνδέσεων γίνεται με τη βοήθεια της κωδικής δήλωσης `PORT (…)`, όπου μέσα στην παρένθεση γίνονται οι δηλώσεις των σημάτων διασύνδεσης:

```
PORT(όνομα_σήματος_1 : τρόπος_λειτουργίας τύπος_σήματος_1;
     όνομα_σήματος_2 : τρόπος_λειτουργίας τύπος_σήματος_2;
     ………………………………

     όνομα_σήματος_N : τρόπος_λειτουργίας τύπος_σήματος_N);
```

Στις επόμενες παραγράφους θα γίνει αναφορά στα *αντικείμενα* δεδομένων, όπως είναι τα «**σήματα**», καθώς και στους *τύπους* δεδομένων όπου υπάγεται κάθε σήμα. Ο «`τρόπος_λειτουργίας`» (mode) καθορίζει την κατεύθυνση του σήματος διασύνδεσης και μπορεί να ανήκει σε μια από τις παρακάτω κατηγορίες:

- **`IN`** Χρησιμοποιείται για σήματα που αποτελούν εισόδους στη βαθμίδα
- **`OUT`** Χρησιμοποιείται για σήματα που αποτελούν εξόδους. Η κατάσταση τέτοιων σημάτων μπορεί να αναγνωριστεί μόνον από άλλες βαθμίδες τις οποίες τα σήματα τροφοδοτούν, όχι όμως από το εσωτερικό της οντότητας
- **`INOUT`** Πρόκειται για δικατευθυντήρια σήματα, όπως αυτά που συνδέονται με διαδρόμους δεδομένων δύο κατευθύνσεων (π.χ. σήματα εισόδου/εξόδου σε μνήμη)
- **`BUFFER`** για σήματα εξόδου, που ταυτόχρονα μπορούν να διαβαστούν στο εσωτερικό της οντότητας

Οι παραπάνω κατηγορίες σημάτων διασύνδεσης αποτελούν δεσμευμένες λέξεις.

Η δήλωση της οντότητας κλείνει με την πρόταση τέλους:

```
END [ENTITY] [όνομα_οντότητας];
```

Μέσα στη δήλωση της οντότητας μπορούν να γίνουν προαιρετικά οι λεγόμενες «γενικές δηλώσεις» (generic declarations). Οι γενικές δηλώσεις τοποθετούνται μετά τη δήλωση του ονόματος της οντότητας και πριν τη δήλωση PORT. Θα αναφερθούμε σ' αυτές στην παράγραφο 2.9.

Η οντότητα είναι το «ορατό» μέρος ενός κυκλώματος μέσα σε ένα μεγαλύτερο σύστημα. Δηλαδή, είναι αυτό που «φαίνεται» από άλλα κυκλώματα.

2.5 Αρχιτεκτονική

Η αρχιτεκτονική (architecture) περιέχει μια ακριβή περιγραφή του τρόπου λειτουργίας του κυκλώματος. Ξεκινά με μια δήλωση:

```
ARCHITECTURE όνομα_αρχιτεκτονικής OF όνομα_οντότητας IS
```

όπου `όνομα_αρχιτεκτονικής` είναι ένα αλφαριθμητικό αναγνωριστικό, που ορίζει ο χρήστης για την αρχιτεκτονική και `όνομα_οντότητας` είναι το ίδιο αναγνωριστικό με το οποίο ο σχεδιαστής έχει ονομάσει την οντότητα.

Η περιγραφή της αρχιτεκτονικής αρχίζει πάντα με τη δεσμευμένη λέξη BEGIN. Τα σήματα που ορίζονται στην οντότητα κληρονομούνται στο σώμα της αρχιτεκτονικής μαζί με τους τύπους τους. Ανάμεσα στη δήλωση της αρχιτεκτονικής και στη λέξη BEGIN μπορεί να υπάρχουν επιπλέον δηλώσεις σημάτων εκτός από αυτά που έχουν δηλωθεί ως σήματα εισόδων/εξόδων, τα οποία χρειάζονται για την περιγραφή της αρχιτεκτονικής.

Άρα, η αρχιτεκτονική ενός κυκλώματος σε VHDL περιγράφεται ως εξής:

```
ARCHITECTURE όνομα_αρχιτεκτονικής OF όνομα_οντότητας IS

[Δηλώσεις επιπλέον σημάτων]

BEGIN

Εντολές που περιγράφουν λογικές λειτουργίες και αναθέτουν

τιμές σε σήματα

END [ARCHITECTURE] [όνομα_αρχιτεκτονικής];
```

Ο αγκύλες σημαίνουν ότι το αντίστοιχο τμήμα είναι προαιρετικό.

Ανάμεσα στο BEGIN και στο END της αρχιτεκτονικής υπάρχουν οι εντολές που υλοποιούν τη λογική του κυκλώματος. Μια πρώτη ανάγνωση του παραδείγματος που παρουσιάζεται στον κώδικα 2.1 θα δώσει μια διαισθητική εικόνα σχετικά με το τι επιτελείται στον κώδικα. Το παράδειγμα υλοποιεί έναν πολυπλέκτη 2:1. Προφανώς, ο πολυπλέκτης εξάγει στην έξοδο το κανάλι x αν το σήμα επιλογής s είναι μηδέν ('0'), ενώ εξάγει το κανάλι y αν το σήμα επιλογής είναι '1'. Η ανάθεση τιμής στο σήμα εξόδου γίνεται στον κώδικα 2.1 με την εντολή SELECT, που θα μελετηθεί πιο αναλυτικά σε επόμενη παράγραφο. Εδώ, αρκεί να ειπωθεί ότι η εντολή SELECT πρέπει να καλύπτει όλες τις πιθανές τιμές του σήματος εισόδου s, ώστε να παρέχει έναν πλήρη πίνακα αληθείας. Επειδή τα σήματα τύπου std_logic δεν περιλαμβάνουν μόνον τις

τιμές `'0'` και `'1'`, αλλά υποστηρίζουν συνολικά εννέα χαρακτήρες (όπως `'Z'`, `'-'`, `'X'` κ.ά.) θα πρέπει να καλυφθεί ο πίνακας αληθείας για όλες αυτές τις περιπτώσεις. Αυτός είναι ο λόγος, που στη σειρά 14 του κώδικα 2.1 η εντολή `SELECT` κλείνει τις περιπτώσεις ανάθεσης τιμών με τη διατύπωση `WHEN OTHERS;` Δηλαδή, στην έξοδο `f` θα ανατεθεί η τιμή του καναλιού `y` σε κάθε περίπτωση που το σήμα `s` δεν έχει τιμή `'0'`.

2.6 Το παράδειγμα ενός απαριθμητή

Στον κώδικα 2.2 παρουσιάζεται πρόγραμμα VHDL για έναν απαριθμητή δεκαέξι καταστάσεων (ή αλλιώς απαριθμητής mod16). Ο απαριθμητής είναι ένα ακουθιακό κύκλωμα, που αυξάνει την έξοδό του κατά ένα κάθε φορά που δέχεται ένα μέτωπο παλμού ρολογιού. Για το συγχρονισμό διεργασιών με μεταβάσεις σημάτων (όπως ο παλμός clock) η γλώσσα VHDL χρησιμοποιεί μια δομή που ονομάζεται `PROCESS` (διεργασία). Οι εντολές που περιλαμβάνονται στην `PROCESS`, όπως για παράδειγμα η `IF` στον κωδ. 2.2, εκτελούνται σειριακά, αλλά οι αναθέσεις των σημάτων γίνονται μόνον στο τέλος, με σύγχρονο τρόπο.

Όπως αναφέρθηκε στην προηγούμενη παράγραφο, για την περιγραφή της λειτουργίας του κυκλώματος είναι δυνατό να απαιτείται η χρήση ενδιάμεσων σημάτων, εκτός από αυτά που έχουν δηλωθεί ως σήματα εισόδων/εξόδων για τη βαθμίδα. Τέτοιες δηλώσεις γίνονται συνήθως μέσα στην αρχιτεκτονική, πριν τη δήλωση `BEGIN`. Στον κώδικα 2.2 παρατηρούμε ότι αμέσως μετά τη δήλωση της αρχιτεκτονικής δηλώνεται ένα βοηθητικό σήμα `m` τύπου `std_logic_vector`, το οποίο στη συνέχεια χρησιμοποιείται μέσα στο σώμα της αρχιτεκτονικής προκειμένου να περιγραφεί η λειτουργία του απαριθμητή. Το σήμα αυτό δεν ανήκει στα σήματα διασύνδεσης εισόδου/εξόδου. Μετά το πέρας του τμήματος κώδικα που περιλαμβάνεται στην `PROCESS` το σήμα `m` αποδίδεται στην έξοδο `q` προκειμένου να φανεί στους ακροδέκτες.

2.7 Σύγχρονες και ακολουθιακές προτάσεις

Η εντολή `SELECT` του κώδικα 2.1 αποτελεί μια «*σύγχρονη*» (concurrent) εντολή. Η λειτουργία που περιγράφει η `SELECT` θα υλοποιηθεί άμεσα, σε όποιο σημείο του κώδικα κι αν βρίσκεται η εντολή, όπως συμβαίνει στα συνδυαστικά κυκλώματα. Κατά την προσομοίωση, οι αναθέσεις τιμών θα ανανεωθούν ταυτόχρονα με όλες τις άλλες σύγχρονες εντολές που μπορεί να υπάρχουν σε έναν κώδικα. Υπενθυμίζουμε ότι η VHDL είναι μια παράλληλη (σύγχρονη) γλώσσα προσομοίωσης λογικών κυκλωμάτων και οι διεργασίες που περιγράφει εκτελούνται σε συγχρονισμό μεταξύ τους, ανεξάρτητα από τη θέση των εντολών στον κώδικα. Δηλαδή, αν μια λειτουργία που περιγράφεται *στο τέλος* του κώδικα ενημερώνει ένα σήμα που χρησιμοποιείται από εντολές *στην αρχή* του κώδικα, τότε ο προσομοιωτής *επιστρέφει* στην αρχή και ενημερώνει τα αποτελέσματα των εντολών, σε συγχρονισμό με το σήμα που μόλις άλλαξε. Οι αλλαγές των σημάτων σταματούν, όταν σταθεροποιηθούν όλα τα σήματα. Όταν εκτελεστούν όλες οι προβλεπόμενες διεργασίες, τότε ολοκληρώνεται ένας κύκλος προσομοίωσης. Όταν αλλάξουν οι τιμές κάποιων σημάτων, μετά από μια προγραμματισμένη χρονική καθυστέρηση, τότε οι διεργασίες του κώδικα σαρώνονται ξανά και γίνεται η ανάθεση των νέων τιμών στον επόμενο

```
1 LIBRARY  ieee;
2 USE ieee.std_logic_1164.all;
3 USE ieee.std_logic_unsigned.all;
4 --------------------------------------------------------
5 ENTITY counter1 IS
6     PORT(clk :  IN std_logic;
7           q  :  OUT std_logic_vector (3 DOWNTO 0));
8 END counter1;
9 --------------------------------------------------------
10 ARCHITECTURE behaviour OF counter1 IS
11 SIGNAL m : std_logic_vector (3 DOWNTO 0);
12 BEGIN
13   PROCESS(clk)
14   BEGIN
15          IF clk 'event AND clk='1' THEN
16            m<= m+1;
17          ELSE
18            m<=m;
19          END IF;
20   END process;
21   q<=m;
22 END behaviour;
```

Κώδικας 2.2 Κώδικας VHDL για την περιγραφή ενός απαριθμητή mod16

κύκλο προσομοίωσης. Μετά τη ***σύνθεση*** (βλέπε παράγραφο 1.2), ο συγχρονισμός αυτός προφανώς ισχύει και στο φυσικό κύκλωμα, όπως και στο επίπεδο της προσομοίωσης. Η λειτουργία του προσομοιωτή, προσομοιώνει με όση χρονική ακρίβεια είναι δυνατό τις λειτουργίες του φυσικού ψηφιακού κυκλώματος, σε κάθε κύκλο προσομοίωσης.

Όπως είναι γνωστό, υπάρχουν λειτουργίες στα ψηφιακά κυκλώματα, που δεν επιτελούνται διαρκώς, παρά μόνον σε συγχρονισμό με συγκεκριμένα «συμβάντα», όπως είναι τα μέτωπα των παλμών ρολογιού. Τέτοιες είναι οι λειτουργίες των ακολουθιακών κυκλωμάτων. Στις λειτουργίες αυτές οι αναθέσεις των εξόδων δεν γίνονται άμεσα (όπως στα συνδυαστικά κυκλώματα) αλλά μόνο κατά τις μεταβάσεις σημάτων. Επίσης, λαμβάνουν υπόψη και προηγούμενες καταστάσεις. Δηλαδή, οι αναθέσεις γίνονται «υπό συνθήκη», με βάση και τη μνήμη προηγούμενων καταστάσεων.

Έτσι, υπάρχει η ανάγκη να εισαχθούν δομές κώδικα που δεν ανανεώνουν άμεσα τις τιμές των σημάτων, αλλά μόνον σε συγκεκριμένες και διατεταγμένες χρονικές στιγμές, επιτρέποντας και φαινόμενα μνήμης ανάμεσα σε διαδοχικές μεταβάσεις σημάτων. Τέτοια δομή είναι η PROCESS, που υπάρχει στην αρχιτεκτονική του κώδικα 2.2. Η PROCESS ανανεώνει τις αναθέσεις τιμών σε σήματα που αναφέρονται στο εσωτερικό της μόνον όταν αλλάξουν οι τιμές των σημάτων που περιλαμβάνονται στη «λίστα ευαισθησίας» της. Η λίστα αυτή βρίσκεται σε

παρένθεση αμέσως δίπλα στην δήλωση PROCESS. Στον κωδ. 2.2 η λίστα ευαισθησίας περιλαμβάνει το σήμα clk. Κατά την προσομοίωση του κυκλώματος ο μεταγλωττιστής θα εκτελέσει τις εντολές στο εσωτερικό της δομής PROCESS *σειριακά*, αλλά θα αποδώσει στα σήματα *μόνον* τις τελικές τιμές τους, αφού ολοκληρωθεί η σειριακή εκτέλεση της διεργασίας. Από τη σκοπιά του χρήστη, η συνολική διεργασία PROCESS αναθέτει τιμές στα σήματα με «σύγχρονο» τρόπο, όπως και κάθε άλλη σύγχρονη εντολή. Απλά, οι αναθέσεις αυτές ανανεώνονται μόνον κατά τις μεταβάσεις των σημάτων που περιλαμβάνονται στη λίστα ευαισθησίας, όπως πρέπει να συμβαίνει σε ένα ακολουθιακό κύκλωμα. Όταν γίνει η σύνθεση του κυκλώματος, τότε το εργαλείο σχεδίασης μεταφράζει την PROCESS σε ένα ακολουθιακό κύκλωμα με Flip-Flops, που λειτουργούν ως καταχωρητές και διατηρούν τις τιμές των σημάτων, μέχρι να συμβούν οι επόμενες μεταβάσεις.

Τα παραπάνω εισάγουν τις βασικές ιδέες που συνδέονται με τις σύγχρονες και τις ακολουθιακές εντολές, αλλά θα μελετηθούν περισσότερο σε επόμενες παραγράφους. Εκεί θα γίνει αναφορά και σε άλλες σύγχρονες προτάσεις εκτός από τη SELECT, καθώς και στις ακολουθιακές προτάσεις, όπως η IF, η CASE κ. ά. Στο παράδειγμα του κώδικα 2.2, η IF αναθέτει στο m την τιμή m+1, αν συμβεί μετάβαση του παλμού ρολογιού σε '1' (θετικό μέτωπο), αλλιώς διατηρεί την προηγούμενη τιμή (δηλαδή εισάγει έναν καταχωρητή). Η τελική τιμή του m εμφανίζεται μόνον μετά το πέρας της διεργασίας.

2.8 Λίγα λόγια για τους τύπους δεδομένων

Πολύ σημαντική για την ανάπτυξη της αρχιτεκτονικής είναι η κατανόηση του ρόλου των τύπων δεδομένων στη VHDL. Στο θέμα αυτό θα γίνει αναφορά κυρίως στη παράγραφο 3 Μια εισαγωγική νύξη μπορεί να γίνει με τη βοήθεια του παραδείγματος που παρουσιάζεται στον κώδικα 2.2, που περιγράφει τον απαριθμητή. Όπως φαίνεται στο μέρος της οντότητας, ο απαριθμητής έχει είσοδο παλμών ρολογιού (clk). Η είσοδος αυτή είναι σήμα του τύπου std_logic, που όπως θα δούμε ορίζει ένα σύνολο εννέα τιμών, τις οποίες μπορεί να λάβει ένα σήμα. Οι κυριότερες είναι οι '1', '0', 'Z', που η κάθε μια ορίζεται ως ένας χαρακτήρας. Η έξοδος q ορίζεται ως ένα σήμα τύπου std_logic_vector, δηλαδή ως ένας πίνακας χαρακτήρων τύπου std_logic. Το (3 downto 0) στη δήλωση του τύπου σημαίνει ότι το σήμα έχει 4 bit. Τα στοιχεία του πίνακα είναι τα q(3), q(2), q(1), q(0), με πρώτο εξ' αριστερών το πιο σημαντικό bit. Ο ίδιος τύπος έχει προβλεφθεί και για το βοηθητικό σήμα m. Η χρήση των τύπων αυτών γίνεται δυνατή από τη χρήση του πακέτου *std_logic_1164* της βιβλιοθήκης *ieee* (σειρές 1, 2).

Ο αναγνώστης μπορεί να παρατηρήσει ότι η λειτουργία της απαρίθμησης εξασφαλίζεται από την πρόσθεση της μονάδας στο σήμα m, σε κάθε θετικό μέτωπο του παλμού ρολογιού (σειρά 16). Όμως, το πρόβλημα είναι ότι ο τύπος δεδομένων std_logic_vector που ορίζεται στο πακέτο *std_logic_1164* δεν υποστηρίζει τη χρήση αριθμητικών τελεστών, όπως ο «+». Για να επιτελεστεί η πρόσθεση χρειάζεται η χρήση κατάλληλου πακέτου, που να επεκτείνει τη χρήση του αριθμητικού τελεστή «+» σε σήματα τύπου std_logic_vector. Τέτοιο πακέτο είναι το *std_logic_unsigned*, το οποίο χρησιμοποιούμε με τη USE στη σειρά 3. Το πακέτο αυτό επιτρέπει την τέλεση μη-προσημασμένων αριθμητικών πράξεων για σήματα του τύπου std_logic και std_logic_vector (βλέπε και σχετικά σχόλια και στην παράγραφο 2.2).

Ας σημειωθεί ότι αν κάναμε χρήση του προκαθορισμένου τύπου `integer` για τα σήματα `m` και `q`, τότε θα μπορούσαμε να κάνουμε χρήση των σημάτων και του τελεστή «+» χωρίς δηλώσεις βιβλιοθηκών και πακέτων. Μια εκδοχή του κώδικα 2.2 με χρήση μόνον προκαθορισμένων τύπων σημάτων (`bit` για το `clk` και `integer` για τα `q` και `m`) φαίνεται στο παράδειγμα 3.1 (κώδικας 3.1), της παραγράφου 3.2.1.

Τέλος, ας σημειωθεί, ότι η ανάθεση της τιμής του `m` στο `q` (σειρά 21 στον κωδ. 2.2) μπορεί να γίνει άμεσα επειδή τα δύο σήματα `m` και `q` είναι ίδιου τύπου (`std_logic_vector`). Σε διαφορετική περίπτωση, η ανάθεση θα έπρεπε να γίνει μέσω μιας συναρτήσεως μετατροπής τύπου. Ο τελεστής `<=` , όταν χρησιμοποιείται σε σήματα, όπως στις γραμμές 13, 15 και 18, είναι ο τελεστής ανάθεσης (ή αντιστοίχησης). Προφανώς, αυτή η χρήση δεν έχει σχέση με τον τελεστή «μικρότερο ή ίσο».

2.9 Γενικές δηλώσεις (GENERIC declarations)

Οι γενικές δηλώσεις αφορούν σε σταθερές που ορίζονται με γενικό τρόπο, ώστε να μπορούν να τροποποιηθούν. Με τον τρόπο αυτό μια σχεδίαση μπορεί να προσαρμοστεί ανάλογα με τις ανάγκες κάθε συγκεκριμένης υλοποίησης. Ο σκοπός της χρήσης των γενικών δηλώσεων είναι να παραμετροποιηθεί το κύκλωμα ώστε να μπορεί να επαναχρησιμοποιηθεί σε περισσότερες περιπτώσεις.

Οι γενικές δηλώσεις σταθερών γίνονται με τη δεσμευμένη λέξη `GENERIC` και τοποθετούνται μέσα στο τμήμα της οντότητας, αμέσως πριν τη δήλωση PORT. Ας σημειωθεί ότι είναι η μόνη δήλωση που επιτρέπεται να τοποθετηθεί πριν από τα σήματα θυρών. Με τον τρόπο αυτό, οι γενικές σταθερές (generic constants) είναι πραγματικά καθολικές (global) και μπορούν να επηρεάσουν ακόμη και τις προδιαγραφές των θυρών.

Για παράδειγμα, ας υποθέσουμε ότι θέλουμε να περιγράψουμε έναν γενικευμένο δυαδικό αποκωδικοποιητή. Αυτός δέχεται Ν εισόδους και παράγει 2^N εξόδους. Ο κώδικας 2.3 περιγράφει τον δυαδικό αποκωδικοποιητή για οποιονδήποτε αριθμό εισόδων/εξόδων και μπορεί να χρησιμοποιηθεί σε όποιο σύστημα χρειάζεται ένας δυαδικός αποκωδικοποιητής, αλλάζοντας απλώς τη σταθερά Ν. Στη συγκεκριμένη υλοποίηση η τιμής της σταθεράς είναι Ν=3, οπότε ο αποκωδικοποιητής είναι 3:8. Ο κώδικας περιέχει πολλά στοιχεία που δεν αναπτύχθηκαν προς το παρόν, αλλά μεταδίδει την κεντρική ιδέα. Ορισμένα σημεία προσοχής είναι τα εξής:

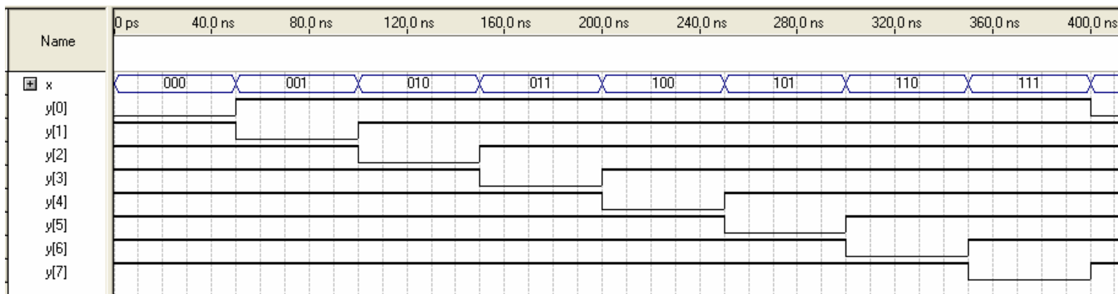
Α. Για τα σήματα εισόδων/εξόδων χρησιμοποιείται ο τύπος δεδομένων `std_logic_vector`, που είναι το πρότυπο της βιομηχανίας. Ο αριθμός των bits στην είσοδο/έξοδο είναι γενικός και καθορίζεται από την τιμή Ν. Επειδή στη γραμμή 14 η είσοδος `x` παίρνει μέρος σε μια σύγκριση με έναν ακέραιο (i), προστίθεται η χρήση και του πακέτου *std_logic_unsigned*. Το πακέτο αυτό επεκτείνει τη δράση του τελεστή σύγκρισης και επιτρέπει μη προσημασμένες συγκρίσεις ακεραίων με τον τύπο `std_logic_vector`.

Β. Η βασική δομή που χρησιμοποιείται στο σώμα της αρχιτεκτονικής είναι η `FOR…GENERATE`. Πρόκειται για μια πολύ ευέλικτη δομή επανάληψης, που επιτρέπει να λαμβάνουν τιμές σήματα τύπου πίνακα, όπως είναι ο τύπος `std_logic_vector`. Θα αναφερθούμε αναλυτικότερα στην εντολή αυτή στο κεφάλαιο 5.

Στο σχήμα 2.2 παρουσιάζεται το αποτέλεσμα της προσομοίωσης του κυκλώματος στο περιβάλλον του λογισμικού Quartus II.

```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.all;
3 USE ieee.std_logic_unsigned.all;
4 ---------------------------------
5 ENTITY generic_decoder IS
6     GENERIC(N: NATURAL :=3);
7     PORT(x : IN std_logic_vector(N-1 downto 0);
8           y : OUT std_logic_vector(2**N-1 downto 0));
9 END generic_decoder;
10 ---------------------------------
11 ARCHITECTURE gen_dec OF generic_decoder IS
12 BEGIN
13    rep: FOR i IN 0 TO 2**N-1 GENERATE
14          y(i) <= '0' WHEN i=x ELSE
15                   '1' ;
16    END GENERATE;
17 END gen_dec;
```

Κώδικας 2.3 Κώδικας γενικευμένου δυαδικού αποκωδικοποιητή με χρήση της δήλωσης GENERIC



Σχήμα 2.2 Αποτέλεσμα της προσομοίωσης του κώδικα 2.3 για τον δυαδικό αποκωκδικοποιητή, με N=3.

2.10 Εξοικείωση με το λογισμικό Quartus II

Στο σημείο αυτό ο αναγνώστης ενθαρρύνεται να μελετήσει το Παράρτημα Α, προκειμένου να αποκτήσει μια βασική εξοικείωση με το λογισμικό Quartus II. Το λογισμικό αυτό αποτελεί εργαλείο σχεδίασης και σύνθεσης της εταιρίας Altera. Ακολουθώντας τα βήματα εκμάθησης του παραρτήματος, ο αναγνώστης θα καταστεί ικανός να επαναλάβει τις σχεδιάσεις που περιγράφονται στα επόμενα και να δημιουργήσει τις δικές του.

3 Αντικείμενα και τύποι δεδομένων στη VHDL

Στην παράγραφο αυτή θα γίνει αναφορά στα βασικά αντικείμενα δεδομένων που χρησιμοποιούνται στη σύνταξη του κώδικα. Επίσης, θα αναφερθούν οι βασικοί τύποι δεδομένων και οι σχετικές βιβλιοθήκες και πακέτα, όπου ορίζονται αυτοί οι τύποι.

3.1 Αντικείμενα δεδομένων (objects)

Η γλώσσα VHDL διαχειρίζεται τρία «αντικείμενα δεδομένων» (data objects) τα οποία μπορούν να μεταφέρουν πληροφορία μέσα στο σύστημα και να αποδίδουν τιμές σε διάφορα σημεία. Σε κάθε αντικείμενο δίνουμε ένα όνομα και το ορίζουμε σύμφωνα με κάποιο «τύπο δεδομένων» (data type). Επίσης, κάθε αντικείμενο έχει σε κάθε στιγμή κάποια τιμή. Τα αντικείμενα δεδομένων που χρησιμοποιούνται είναι:

Α. Σήματα (`SIGNALS`)
Β. Μεταβλητές (`VARIABLES`)
Γ. Σταθερές (`CONSTANTS`)

Υπάρχει ένα ακόμη αντικείμενο, τα αρχεία (FILES) τα οποία προορίζονται για προσομοίωση και όχι για σύνθεση, οπότε θα περιγραφούν με συντομία στην αντίστοιχη παράγραφο. Τα άλλα τρία αντικείμενα θα περιγραφούν στις επόμενες παραγράφους.

Τα **σήματα** είναι τα πλέον σημαντικά, καθώς αποδίδουν τιμές στα καλώδια του κυκλώματος και αντιπροσωπεύουν τις διασυνδέσεις ανάμεσα σε μονάδες του κυκλώματος. Μπορούν να χρησιμοποιηθούν τόσο σε τμήματα κώδικα που περιλαμβάνουν σύγχρονες εντολές, όσο και σε ακολουθιακά τμήματα κώδικα.

Οι **μεταβλητές** χρησιμοποιούνται για την προσωρινή αποθήκευση τιμών που προκύπτουν από την τέλεση αριθμητικών πράξεων. Μια μεταβλητή δηλώνεται και χρησιμοποιείται μόνο σε τμήματα του κώδικα που περιλαμβάνουν «ακολουθιακές εντολές». Ένα τέτοιο τμήμα κώδικα δηλώνεται συνήθως ως ΔΙΕΡΓΑΣΙΑ (`PROCESS`) και χρησιμοποιείται κυρίως για την περιγραφή ακολουθιακών κυκλωμάτων, όπως καταχωρητές, απαριθμητές κλπ. Στην πράξη, όμως, μπορεί να χρησιμοποιηθεί και για την περιγραφή συνδυαστικών κυκλωμάτων, όπως πολυπλέκτες ή αποκωδικοποιητές. Ο ακολουθιακός κώδικας εκτελείται με βάση μια ακολουθία συμβάντων, όπως οι παλμοί ρολογιού ή οι μεταβάσεις άλλων σημάτων. Εκτός από τις διεργασίες, άλλα τμήματα κώδικα που περιγράφονται με ακολουθιακές εντολές είναι τα λεγόμενα υποπρογράμματα (`FUNCTIONS` και `PROCEDURES`).
Άρα, μια μεταβλητή δηλώνεται και χρησιμοποιείται μόνον μέσα σε `PROCESS` ή σε υποπρόγραμμα. Σ’ αυτά τα τμήματα κώδικα, οι μεταβλητές είναι ιδιαίτερα χρήσιμες, καθώς παρέχουν έναν μηχανισμό αποθήκευσης τιμών, που είναι ήδη γνωστός από άλλες γλώσσες προγραμματισμού, όπως η C ή η Pascal. Για να επηρεάσει το κύκλωμα, η τιμή μιας μεταβλητής θα πρέπει εν τέλει να αποδοθεί σε κάποιο σήμα. Όπως θα εξηγηθεί παρακάτω, υπάρχει διαφορά στη χρήση σημάτων και μεταβλητών, που πρέπει να γίνεται απόλυτα σεβαστή.

Τέλος, οι **σταθερές** λαμβάνουν τιμή κατά τη δήλωσή τους και η τιμή αυτή παραμένει στη συνέχεια σταθερή. Άρα, η σταθερά δεν αντιπροσωπεύει κάποιο καλώδιο του συστήματος, απλά μεταφέρει μια συγκεκριμένη αριθμητική τιμή, όπου χρειάζεται.

3.1.1 Δήλωση (declaration) αντικειμένων (σταθερών, σημάτων, μεταβλητών)

Προκειμένου να χρησιμοποιηθούν μέσα στον κώδικα τα παραπάνω αντικείμενα δεδομένων, πρέπει πρώτα να γίνει η «δήλωσή» τους (declaration). Όταν δηλώνεται ένα αντικείμενο, ορίζεται το όνομα με το οποίο θα το χρησιμοποιούμε μέσα στον κώδικα και ο τύπος του (σύμφωνα με τους τύπους δεδομένων που υποστηρίζονται από τις βιβλιοθήκες της VHDL ή σύμφωνα με κάποιον τύπο ορισμένο από τον χρήστη). Αν το αντικείμενο είναι σταθερά, ορίζεται η τιμή της. Προαιρετικά, μπορεί να οριστεί αρχική τιμή και για ένα σήμα ή μεταβλητή.

Στα αντικείμενα δεδομένων (σταθερές, σήματα, μεταβλητές) δίνει ο χρήστης ότι όνομα θέλει. Το όνομα που θα δώσει θα πρέπει να υπακούει σε κάποιους κανόνες. Στην VHDL τα ονόματα μπορούν να έχουν οποιονδήποτε αλφαριθμητικό χαρακτήρα πεζό η κεφαλαίο καθώς και τον χαρακτήρα '_' (A-Z, a-z, _ , 0-9). Οι περιορισμοί που υπάρχουν στην απόδοση ονομάτων είναι οι εξής :

- Το όνομα δεν μπορεί να έχει δύο συνεχόμενους χαρακτήρες '_' (π.χ. `ono__ma`).
- Το όνομα του αντικειμένου δεν μπορεί να ξεκινάει ή να τελειώνει σε '_' (π.χ. `_name` ή `name_`).
- Δεν μπορεί να ξεκινάει με αριθμό (π.χ. `1name`).
- Δεν μπορεί να ταυτίζεται με λέξη-κλειδί της γλώσσας VHDL (π.χ. `ENTITY`, `ARCHITECTURE`, `FILE`, κ.λπ.)

Στη VHDL δεν υπάρχει διαχωρισμός μεταξύ πεζών και κεφαλαίων γραμμάτων, δηλαδή το '`K`' είναι το ίδιο με το '`k`'.

Δήλωση σταθερών

Η μορφή δήλωσης μιας σταθεράς είναι:

```
CONSTANT όνομα σταθεράς : τύπος := τιμή σταθεράς;
```

Δήλωση σημάτων

Τα σήματα μπορούμε να τα ορίσουμε σε τρεις περιοχές μέσα στον κώδικα. Στην περιοχή δηλώσεων της οντότητας (entity), στο τμήμα δήλωσης της αρχιτεκτονικής και στο τμήμα δηλώσεων ενός πακέτου (package). Η μορφή ορισμού ενός σήματος είναι :

```
SIGNAL όνομα σήματος  :  τύπος σήματος
```

Ο τύπος ενός σήματος προσδιορίζει τις τιμές που μπορεί να λάβει το σήμα στην συγκεκριμένη περιγραφή, σύμφωνα με όσα αναφέρθηκαν στην προηγούμενη παράγραφο.

Δήλωση μεταβλητών

Τα αντικείμενα δεδομένων τύπου μεταβλητή (VARIABLE) χρησιμοποιούνται για την προσωρινή αποθήκευση τιμών που προκύπτουν από την τέλεση αριθμητικών πράξεων καθώς και για τις

μεταβλητές ενός δείκτη μέσα σε βρόγχους. Η δήλωση μιας μεταβλητής γίνεται με τον εξής τρόπο:

```
VARIABLE  όνομα μεταβλητής : τύπος μεταβλητής;
```

3.2 Τύποι δεδομένων (data types) ορισμένοι μέσω προτύπων

Κάθε αντικείμενο δεδομένων στη VHDL πρέπει να έχει έναν «τύπο». Ο τύπος ενός σήματος προσδιορίζει τις τιμές που μπορεί να λάβει το σήμα, καθώς και τις πράξεις που υποστηρίζει. Δεν είναι δυνατό να γράψει κανείς κώδικα VHDL χωρίς να καταλαβαίνει επαρκώς ποιοί είναι οι επιτρεπτοί τύποι και πως χρησιμοποιούνται.

Ο κάθε τύπος δεδομένων υποστηρίζει τη χρήση κάποιων τελεστών, ενώ δεν υποστηρίζει τη χρήση κάποιων άλλων. Στους τελεστές της γλώσσας VHDL θα αναφερθούμε σε επόμενη παράγραφο. Εν ολίγοις, οι τελεστές ανήκουν σε μια από τις εξής κατηγορίες:

- Λογικοί τελεστές (`NOT, AND, NAND, OR, NOR, XOR, XNOR`).
- Αριθμητικοί τελεστές (`+, -, *, /, **, ABS, REM, MOD`)
- Σχεσιακοί τελεστές: (`=, /=, >, <, >=, <=`).
- Τελεστές ολίσθησης: (`SLL, SRL`, κλπ).
- Τελεστής συνένωσης: (`&`).

Στη VHDL διακρίνουμε τους προκαθορισμένους τύπους δεδομένων (predefined data types) και αυτούς που δημιουργούν οι χρήστες (user defined data types).

Οι προκαθορισμένοι τύποι είναι προτυποποιημένοι και οι βιβλιοθήκες τους περιλαμβάνονται στα εργαλεία σχεδίασης, οπότε η χρήση τους μπορεί να γίνει με απλή αναφορά σε έτοιμα πακέτα, που περιγράφουν αυτούς τους τύπους.

3.2.1 Βασικοί προκαθορισμένοι τύποι (πακέτο *standard*)

Κάποιοι βασικοί τύποι δεδομένων ανήκουν στην αρχική προτυποποίηση της γλώσσας (πακέτο *standard* της βιβλιοθήκης *std*) και λειτουργούν χωρίς να χρειάζεται αναφορά σε βιβλιοθήκες.

Οι βασικοί αυτοί τύποι είναι οι εξής:

- **BIT** Τα σήματα αυτού του τύπου μπορούν να πάρουν τις τιμές ‘0’ ή ‘1’. Ο τύπος αυτός υποστηρίζει λογικούς και σχεσιακούς τελεστές. Θεωρείται βαθμωτός (scalar) τύπος, καθώς αποτελείται από ένα bit. Παράδειγμα δήλωσης σήματος τύπου bit:

```
signal flag : bit;
```

- **BIT_VECTOR** Τα σήματα μπορούν να λάβουν μια σειρά τιμών ‘0’ ή ‘1’. Δηλαδή, ο τύπος ορίζεται ως ένας μονοδιάστατος πίνακας (array) με στοιχεία `BIT`. Η αλληλουχία των bits “10101100” αποτελεί ένα διάνυσμα (vector) με οκτώ στοιχεία. Ο τύπος `bit_vector` υποστηρίζει λογικούς και σχεσιακούς τελεστές, καθώς και τελεστές ολίσθησης και συνένωσης. Ένας τρόπος με τον οποίο ορίζεται σήμα αυτού του τύπου δίνεται στο παράδειγμα:

```
signal data_bus : bit_vector (7 downto 0);
```

- **INTEGER** Ένα σήμα αυτού του τύπου μεταφέρει ακέραιες τιμές. Από τη σκοπιά του χρήστη είναι βαθμωτός τύπος, καθώς διαχειρίζεται μία μόνο τιμή. Από την άλλη μεριά, σε επίπεδο

σήματος περιλαμβάνει ένα σύνολο bits. Γενικά, έχει μήκος 32 bits και εύρος τιμών από $-(2^{31}-1)$ ως $(2^{31}-1)$. Το εύρος τιμών μπορεί να αλλάξει με την χρήση της λέξης `RANGE`. Παράδειγμα ορισμού ενός σήματος με ακέραιη τιμή είναι το εξής:

```
signal m : integer range -32 to 32;
```

Μια μεταβλητή τύπου integer με αρχική τιμή 0 και με όρια τιμών -128 έως 128 δηλώνεται ως εξής:

```
variable count : integer range -128 to 128 :=0;
```

Όταν γράφουμε κώδικα που θα χρησιμοποιηθεί για σύνθεση είναι σημαντικό να προσδιορίσουμε τα όρια στα οποία λαμβάνουν τιμές οι ακέραιοι, διότι σε διαφορετική περίπτωση ο compiler θα τους υλοποιήσει με εύρος 32 bit. Ο τύπος `integer` υποστηρίζει αριθμητικές πράξεις και πράξεις σύγκρισης,

- **NATURAL** Είναι υποτύπος του integer και περιλαμβάνει μη αρνητικούς ακεραίους (από 0 μέχρι και το άνω όριο των ακεραίων). Υποστηρίζει τις ίδιες πράξεις με τον τύπο `integer`. Παράδειγμα δήλωσης σήματος τύπου `natural`:

```
signal f : natural range 0 to 15;
```

- **POSITIVE** Είναι υποτύπος του `integer` και περιλαμβάνει μόνον θετικούς ακεραίους.
- **BOOLEAN** Ένα σήμα αυτού του τύπου μπορεί να πάρει την τιμή `TRUE` ή `FALSE`. Είναι βαθμωτός τύπος και υποστηρίζει λογικούς και σχεσιακούς τελεστές.
- **CHARACTER** Σήματα αυτού του τύπου μπορούν να πάρουν τιμές από ένα σύνολο 256 συμβόλων των 8-bit. Τα σύμβολα αντιπροσωπεύουν το σύνολο χαρακτήρων ISO 8859-1, ενώ τα πρώτα 128 σύμβολα ανήκουν στον κοινό κώδικα ASCII. Αν και κάθε χαρακτήρας έχει εύρος 8 bits, ο τύπος διαχειρίζεται κάθε φορά ένα χαρακτήρα, οπότε είναι βαθμωτός. Υποστηρίζει σχεσιακούς τελεστές. Ο παράγωγος τύπος **STRING** ορίζεται ως πίνακας χαρακτήρων και υποστηρίζει σχεσιακούς τελεστές και τη συνένωση (concatenation '&').
- **TIME** Αυτός ο τύπος δεδομένων προορίζεται μόνο για προσομοίωση (όχι για σύνθεση). Με τη βοήθειά του ορίζονται ακέραιοι, που παριστάνουν χρονικές στιγμές στον χρόνο προσομοίωσης. Υποστηρίζει τους αριθμητικούς και σχεσιακούς τελεστές.

Ορισμένοι άλλοι βασικοί τύποι είναι εξειδικευμένοι και δεν θα αναφερθούμε σ' αυτούς.

Παράδειγμα 3.1 Απαριθμητής με προκαθορισμένους τύπους

Στον κώδικα 3.1 περιγράφεται ένας απαριθμητής, όπου τα σήματα ανήκουν στους βασικούς τύπους που περιγράψαμε παραπάνω. Το σήμα ρολογιού είναι τύπου `bit`, ενώ το σήμα εξόδου `q` είναι τύπου `integer`, όπως και το βοηθητικό σήμα `m`. Προφανώς, τα δύο αυτά σήματα θα μπορούσαν να δηλωθούν με τύπο `natural`, αφού η περιοχή τιμών τους συμφωνεί με αυτό.

Στη σειρά 12, το σήμα `clk` αποτελεί το όρισμα ενός σχεσιακού τελεστή, κάτι που όπως αναφέρθηκε υποστηρίζεται από τον τύπο `bit`. Στη σειρά 13 το σήμα `m` χρησιμοποιείται σε μια πρόσθεση, κάτι που υποστηρίζεται από τον τύπο `integer`.

```
1 --------------------------------------------------
2 ENTITY counter1 IS
3    PORT(clk      :     IN bit;
4          q       :     OUT INTEGER RANGE 0 TO 15);
5 END counter1;
6 --------------------------------------------------
7 ARCHITECTURE behaviour OF counter1 IS
8 SIGNAL m : INTEGER RANGE 0 TO 15;
9 BEGIN
10  PROCESS(clk)
11  BEGIN
12         IF clk 'event AND clk='1'  THEN
13          m<= m+1;
14         ELSE
15          m<=m;
16         END IF;
17  END process;
18 q<=m;
19 END behaviour;
```

Κώδικας 3.1 Ο κώδικας του απαριθμητή (κωδ. 2.2) με χρήση των προκαθορισμένων τύπων BIT και INTEGER.

Στη σειρά 18 γίνεται απ' ευθείας ανάθεση του σήματος m στο σήμα q, κάτι που είναι εφικτό επειδή τα δύο σήματα είναι ίδιου τύπου. Σε άλλη περίπτωση είναι πιθανό ότι η ανάθεση θα έπρεπε να γίνει μέσω μιας συνάρτησης μετατροπής τύπου.

Παράδειγμα 3.2

Να σχεδιαστεί ένας δυαδικός αποκωδικοποιητής 2:4 με σήματα τύπου bit_vector και να προσομοιωθεί η λειτουργία του με το λογισμικό Quartus II.

Λύση: Το πρόγραμμα VHDL για τον αποκωδικοποιητή 2:4 παρουσιάζεται στον παρακάτω κώδικα 3.2. Η είσοδος 2-bit έχει δηλωθεί στην οντότητα ως σήμα bit_vector, ενώ η έξοδος y είναι κι αυτή ένα διάνυσμα 4-bit του ίδιου τύπου. Ο τύπος ανήκει στο προκαθορισμένο πακέτο standard της βιβλιοθήκης std, οπότε δεν απαιτείται η δήλωση βιβλιοθηκών.
Στο σώμα της αρχιτεκτονικής περιγράφεται η λειτουργία του κυκλώματος με τη βοήθεια της σύγχρονης πρότασης αντιστοίχησης:

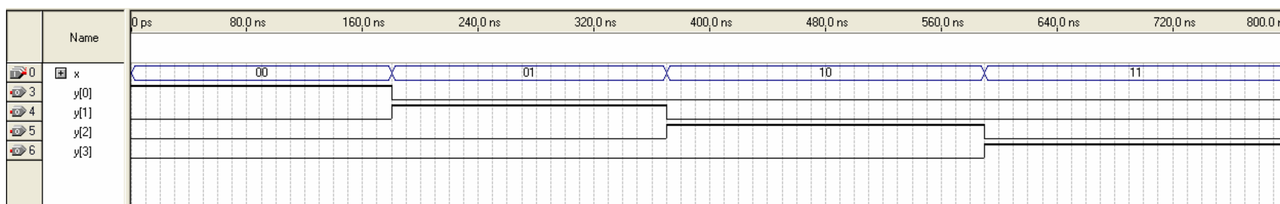
```
Έκφραση_ανάθεσης WHEN <συνθήκη>  ELSE
   Ανάθεση_τιμής WHEN <συνθήκη>  ELSE
        ……. ;
```

Ο κώδικας μεταγλωττίζεται στο περιβάλλον του Quartus II. Μετά την Ανάλυση και Σύνθεση δημιουργούμε το αρχείο εισόδου για τον προσομοιωτή (vector waveform file), κάνοντας χρήση

του επεξεργαστή κυματομορφών (waveform editor) του λογισμικού Quartus II. Εκτελούμε λειτουργική προσομοίωση, οπότε το λογισμικό εμφανίζει την αναφορά που παρουσιάζεται στο σχήμα 3.1. Στην κορυφή του διαγράμματος βρίσκεται το σήμα εισόδου `x`, το οποίο λαμβάνει τις τέσσερις δυνατές τιμές `00`, `01`, `10`, `11`. Για κάθε μια από τις τιμές αυτές ενεργοποιείται το αντίστοιχο bit της εξόδου `y`.
Ο αναγνώστης μπορεί να συγκρίνει την παραπάνω υλοποίηση με αυτή του γενικού αποκωδικοποιητή της παραγράφου 2.9.

```
1 --decoder 2:4
2 ENTITY decoder1 IS
3    PORT(x : IN bit_vector(1 downto 0);
4         y : OUT bit_vector(3 downto 0));
5 END decoder1;
6 -----------------------------------------
7 ARCHITECTURE decoder2_1 OF decoder1 IS
8 BEGIN
9    y<="0001" WHEN x="00" ELSE
10        "0010" WHEN x="01" ELSE
11        "0100" WHEN x="10" ELSE
12        "1000";
13 END decoder2_1;
```

Κώδικας 3.2 Δυαδικός αποκωδικοποιητής



Σχήμα 3.1 Λειτουργική προσομοίωση του κώδικα του σχήματος 3.2.

3.2.2 Τύποι `standard logic` (πακέτο *std_logic_1164*)

Οι τύποι `std_logic` και `std_logic_vector` αποτελούν τα πρότυπα της ηλεκτρονικής βιομηχανίας. Όταν ένας σχεδιαστής αναπτύσσει ένα σύστημα στο πλαίσιο μιας συνεργασίας ή έχει πρόθεση να διαμοιράσει τμήματα του κώδικα, είναι σκόπιμο να σέβεται αυτό το πρότυπο, τουλάχιστο ως προς τα σήματα διασυνδέσεων με άλλα κυκλώματα. Οι τύποι `std_logic` και `std_logic_vector` αποτελούν τη βάση για τις περισσότερες σχεδιάσεις που παρουσιάζονται σ' αυτό το βιβλίο.

Οι τύποι αυτοί ορίζονται στο πακέτο *std_logic_1164*, που προστέθηκε στο πρότυπο IEEE 1174 της VHDL το 1993. Η δήλωση της βιβλιοθήκης *ieee* και του πακέτου *std_logic_1164* γίνεται στην περιοχή δηλώσεων βιβίοθηκών, ως εξής:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
```

Σύμφωνα με το πακέτο που ορίζει τον τύπο `std_logic`, ένα σήμα αυτού του τύπου μπορεί να πάρει τις τιμές '0' και '1', όπως κι ένα σήμα BIT και επιπλέον μπορεί να πάρει κι άλλες επτά τιμές, όπως την τιμή 'Z' που αντιπροσωπεύει την κατάσταση υψηλής εμπέδησης (high-Z) και την αδιάφορη κατάσταση ('-'). Έτσι, τα σήματα `std_logic` μπορούν να χρησιμοποιηθούν για τη σύνθεση απομονωτών τριών καταστάσεων (tri-state buffers), οι οποίοι είναι απαραίτητοι για την οδήγηση διαδρόμων. Επίσης, η συμπερίληψη της αδιάφορης κατάστασης '-' βοηθά στην καλύτερη περιγραφή των λειτουργιών μέσω πινάκων αληθείας και οδηγεί σε καλύτερη βελτιστοποίηση των κυκλωμάτων.

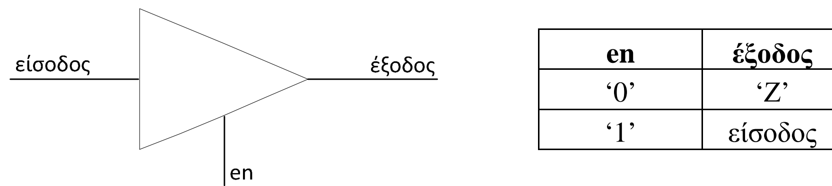
Όλες οι τιμές που λαμβάνει ένα σήμα *std_logic* είναι οι εξής:

```
'0' Λογικό μηδέν
'1' Λογικό ένα
'Z' Υψηλή εμπέδηση
'-' Αδιάφορη κατάσταση
'U' Μη αρχικοποιημένη κατάσταση (Uninitialized)
'X' Άγνωστη κατάσταση
'L' Ασθενής χαμηλή (Weak low)
'H' Ασθενής υψηλή (Weak high)
'W' Ασθενής άγνωστη (Weak unknown)
```

Από τις παραπάνω τιμές οι τέσσερις πρώτες μπορούν να λάβουν μέρος στη σύνθεση, ενώ από τις υπόλοιπες η `'L'` συντίθεται ως `'0'` και η `'H'` ως `'1'`. Οι υπόλοιπες συντίθενται ως αδιάφορες καταστάσεις.

Τα σήματα του τύπου `std_logic_vector` μπορούν να πάρουν τις ίδιες τιμές με τα σήματα `std_logic`, με την διαφορά ότι αφού είναι vector θα αποτελούν έναν πίνακα από τις παραπάνω τιμές και όχι απλά μία από αυτές (π.χ. "0101001100111010").

Το πακέτο std_logic_1164 ορίζει κυρίως τη χρήση λογικών και σχεσιακών τελεστών για τους τύπους `std_logic` και `std_logic_vector`. Για να είναι δυνατή η χρήση αριθμητικών με τους τύπους αυτούς πρέπει να δηλωθεί επιπλέον ένα από τα πακέτα *std_logic_unsigned* ή *std_logic_signed.* Στην περίπτωση αυτή τα αριθμητικά κυκλώματα που προκύπτουν από τη σύνθεση τελούν μη προσημασμένες ή προσημασμένες πράξεις αντίστοιχα. Επίσης, η χρήση αυτών των πακέτων επεκτείνει τη χρήση των σχεσιακών τελεστών για συγκρίσεις σημάτων τύπου `std_logic_vector` με ακεραίους (βλέπε παράδειγμα 2.4).
Ο τύπος `std_logic` αποτελεί περίπτωση του τύπου `std_ulogic`. Οι τιμές που λαμβάνουν τα σήματα του τύπου `std_ulogic` είναι ίδιες με τις τιμές των σημάτων τύπου `std_logic`, με την διαφορά ότι τα σήματα του τύπου `std_ulogic` δεν επιτρέπουν την οδήγηση ενός διαδρόμου από πολλές πηγές σήματος ταυτόχρονα. Το U στο όνομα του τύπου αντιστοιχεί στη λέξη "Unresolved" (χωρίς επίλυση διαμάχης). Ο τύπος `std_logic`, επιλύει τη διαμάχη που προκύπτει όταν πολλές πηγές οδηγούν το ίδιο σήμα. Ας υποθέσουμε, για παράδειγμα, ότι δύο απομονωτές τριών καταστάσεων οδηγούν το ίδιο bit ενός σήματος `f` και ο ένας παράγει έξοδο `'0'`ενώ ο άλλος `'Z'`. Ο τύπος `std_logic` στηρίζεται σε μια «συνάρτηση επίλυσης διαμάχης»



en	έξοδος
‘0’	‘Z’
‘1’	είσοδος

Σχήμα 3.2 Απομονωτής τριών καταστάσεων και πίνακας αληθείας

(ή συνάρτηση διάκρισης-resolution function) προκειμένου να αποδώσει τιμή στο bit του σήματος `f`. Προφανώς, η κεντρική ιδέα είναι είναι ότι η ασθενέστερη τιμή που οδηγεί το δίαυλο είναι η `'Z'`, καθώς ανάμεσα στο `'0'` και το `'Z'` ή ανάμεσα στο `'1'` και στο `'Z'` πάντοτε επικρατεί το `'0'` ή το `'1'` αντίστοιχα.

Παράδειγμα 3.2 Να σχεδιαστεί ένας απομονωτής τριών καταστάσεων με κώδικα VHDL.

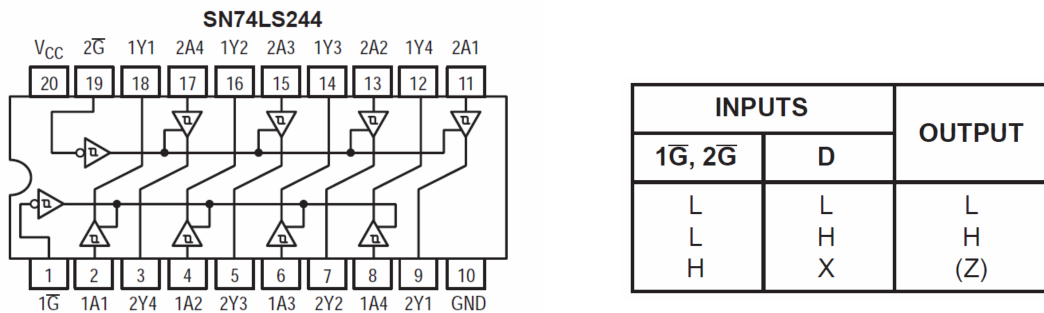
Λύση: Όπως είναι γνωστό, ο απομονωτής τριών καταστάσεων είναι μια πύλη με είσοδο `x` και έξοδο `f`, η οποία έχει και έναν τρίτο ακροδέκτη en που ονομάζεται ακροδέκτης ενεργοποίησης. Αν ο ακροδέκτης en είναι ενεργός, τότε η είσοδος οδηγείται στην έξοδο, ενώ αν είναι ανενεργός η έξοδος αποκτά την «τρίτη κατάσταση», αυτή της υψηλής εμπέδησης (High-Z) (βλέπε και πίνακα αληθείας στο σχήμα 3.2). Όπως αναφέρθηκε, τα σήματα std_logic και std_logic_vector υποστηρίζουν ανάμεσα στις τιμές που μπορούν να λάβουν την κατάσταση `'Z'` της υψηλής εμπέδηση. Άρα, μια απλή υλοποίηση του ζητούμενου κώδικα φαίνεται στον κώδικα 3.3.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------
ENTITY tri_state_1 IS
      PORT(x : IN std_logic;
          en : IN std_logic;
           f : OUT std_logic);
END tri_state_1;
-------------------------------
ARCHITECTURE tri OF tri_state_1 IS
      BEGIN
         f <= x WHEN en ='1' ELSE 'Z';
END tri;
```

Κώδικας 3.3 Κώδικας για απομονωτή τριών καταστάσεων

Παράδειγμα 3.3 Να σχεδιαστεί μια βαθμίδα απομόνωσης τριών καταστάσεων που να υλοποιεί τη λειτουργία του ολοκληρωμένου κυκλώματος 74LS244.

Λύση: Το κύκλωμα 74LS244 φαίνεται στο σχήμα 3.3. Είναι μια οκταπλή βαθμίδα απομόνωσης 4+4-bit. Οι απομονωτές τριών καταστάσεων έχουν ανά τέσσερις κοινό τον ακρο-



INPUTS		OUTPUT
$1\overline{G}$, $2\overline{G}$	D	
L	L	L
L	H	H
H	X	(Z)

Σχήμα 3.3 Το κύκλωμα 74LS244 και ο πίνακας αληθείας του

δέκτη ενεργοποίησης. Συνολικά το σύστημα περιέχει δύο $(1\overline{G}, 2\overline{G})$. Όπως φαίνεται στο σχήμα, οι απομονωτές ενεργοποιούνται με LOW στους ακροδέκτες enable (active low).

Όπως φαίνεται στον πίνακα αληθείας, το κύκλωμα περνάει την είσοδο στην έξοδο όταν ενεργοποιείται με LOW ο αντίστοιχος ακροδέκτης ενεργοποίησης. Με την ευκαιρία πρέπει να σχολιαστεί ότι το X στον πίνακα αληθείας σημαίνει ως γνωστό την αδιάφορη κατάσταση, αν και στον συμβολισμό που (ατυχώς) υιοθετήθηκε στη VHDL ο χαρακτήρας 'X' σημαίνει την άγνωστη κατάσταση. Η αδιάφορη κατάσταση στη VHDL συμβολίζεται με τον χαρακτήρα '-'. Ο κώδικας VHDL που περιγράφει το παραπάνω κύκλωμα είναι ο 3.4.

Οι δηλώσεις των σημάτων διασύνδεσης ανταποκρίνονται ακριβώς στις εισόδους και τις εξόδους του σχ. 3.3. Το κύκλωμα έχει δύο εισόδους 4-bit και δύο αντίστοιχες εξόδους. Στις εισόδους δηλώνονται και τα σήματα ενεργοποίησης. Στο μέρος της αρχιτεκτονικής χρησιμοποιούμε δύο σύγχρονες αναθέσεις για τα σήματα των εξόδων, με την εντολή WHEN...ELSE. Η σύνταξη

```
Y_1 <= (OTHERS=>'Z') WHEN G_1='1' ELSE A_1;
```

σημαίνει ότι τα bit της εξοδου θα βρίσκονται σε κατάσταση υψηλής εμπέδησης (OTHERS=>'Z') όταν η είσοδος ενεργοποίησης είναι HIGH, αλλιώς λαμβάνουν τις τιμές της εισόδου A_1. Εδώ, σημειώνουμε ότι ο τελεστής => χρησιμοποιείται για να αποδώσει τιμές στα στοιχεία ενός πίνακα. Όταν συντάσσεται με την κωδική λέξη OTHERS όπως στις γραμμές 15 και 16 του κωδ. 3.4, τότε αποδίδει την ίδια τιμή ('Z') σε όλα τα στοιχεία του πίνακα. Η έκφραση

```
Y_1<=(0=>'1', 1=>'1', OTHERS=>'0')
```

θα σήμαινε ότι τα 4 bits της εξόδου Y_1 θα λάβουν τιμές "0011".

```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.all;
3 ---------------------------------------------
4 ENTITY tri_244 IS
5      PORT(A_1 : IN std_logic_vector(3 downto 0);
6            A_2 : IN std_logic_vector(3 downto 0);
7            G_1 : IN std_logic;
8            G_2 : IN std_logic;
```

```
9           Y_1 : OUT std_logic_vector(3 downto 0);
10          Y_2 : OUT std_logic_vector(3 downto 0));
11 END tri_244;
12 ------------------------------------------------
13 ARCHITECTURE buffers OF tri_244 IS
14 BEGIN
15   Y_1 <=(OTHERS=>'Z') WHEN G_1='1' ELSE A_1;
16   Y_2 <=(OTHERS=>'Z') WHEN G_2='1' ELSE A_2;
17 END buffers;
```

Κώδικας 3.4 Σχεδίαση του 74LS244 σε κώδικα VHDL

Άσκηση 3.1 Το ολοκληρωμένο κύκλωμα 74LS245 είναι ένας οκταπλός πομποδέκτης διαύλου (bus tranceiver) που χρησιμοποιείται για να συνδέσει μεταξύ τους δύο δικατευθυντήριους διαύλους δεδομένων A, B με εύρος 8-bit. Ένας ακροδέκτης (DIR) ρυθμίζει αν ο A εκπέμπει και ο B λαμβάνει ή το αντίθετο. Να βρείτε στο διαδίκτυο τα φύλλα δεδομένων του κυκλώματος και να σχεδιάσετε το κύκλωμα σε VHDL. *Σημείωση:* οι δίαυλοι A και B πρέπει να οριστούν με τρόπο λειτουργίας (mode) INOUT, αφού προορίζονται να λαμβάνουν και να στέλνουν δεδομένα.

3.2.3 Χρήση των πακέτων *std_logic_(un)signed* με τον τύπο std_logic

Όπως αναφέρθηκε στην παράγραφο 3.2.2 ο τύπος δεδομένων `std_logic/ std_logic_vector` επιτρέπει τη χρήση λογικών τελεστών (`AND`, `OR` κλπ), και σχεσιακών τελεστών (`>=`, `<` κλπ, υπό περιορισμούς), όχι όμως τη χρήση αριθμητικών τελεστών (`+`, `-`, `*`, `/`). Τι γίνεται, όμως, όταν η περιγραφή ενός κυκλώματος απαιτεί μη προσημασμένη ή προσημασμένη πρόσθεση; Σε ορισμένες περιπτώσεις, τέτοια κυκλώματα μπορούν να υλοποιηθούν με την αριθμητική πληροφορία σε μορφή ακεραίου, οπότε ο τύπος των σημάτων δηλώνεται ως `integer`. Από την άλλη μεριά, σε βιομηχανικές σχεδιάσεις χρησιμοποιείται ως πρότυπο ο τύπος δεδομένων `std_logic` και `std_logic_vector`, ειδικά για σήματα που οδηγούν διαύλους.

Η δυνατότητα της χρήσης αριθμητικών τελεστών με σήματα `std_logic/ std_logic_vector` εξασφαλίζεται με τη χρήση ενός από τα πακέτα *std_logic_unsigned* και *std_logic_signed.* Τότε ο compiler θεωρεί τα σήματα `std_logic` και `std_logic_vector` ως μη-προσημασμένα ή προσημασμένα, αντίστοιχα, και συνθέτει τα αντίστοιχα αριθμητικά κυκλώματα.

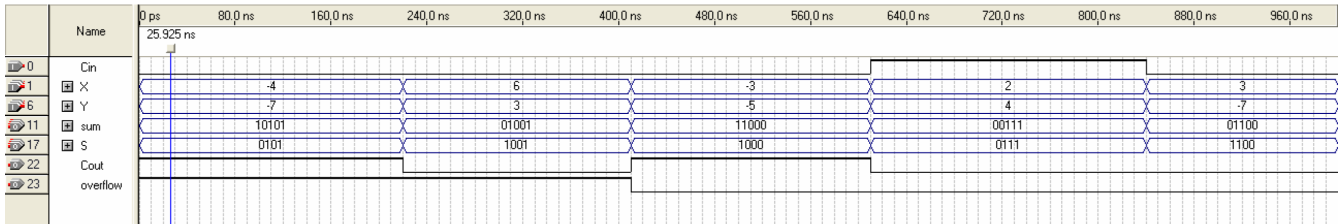
Παράδειγμα 3.4 Να σχεδιαστεί αθροιστής 4-bit που να εκτελεί προσημασμένη/μη-προσημασμένη πρόσθεση.

Λύση: Ο κώδικας 3.5 παρουσιάζει το σχεδιασμό ενός αθροιστή, όπου τα σήματα εισόδου `X` και `Y` και η έξοδος `S` θεωρούνται προσημασμένοι αριθμοί 4-bit, με χρήση του τύπου `std_logic_vector` και του πακέτου *std_logic_signed.* Αντίστοιχα, αν γίνει χρήση του πακέτου *std_logic_unsigned* τα σήματα `X` και `Y` θα θεωρούνται μη προσημασμένοι αριθμοί. Υπενθυμίζουμε ότι σε κάθε μια από αυτές τις δύο περιπτώσεις ισχύει διαφορετικό εύρος τιμών. Το σήμα `Cin` είναι το κρατούμενο εισόδου. Το κύκλωμα παράγει το κρατούμενο εξόδου και το

σήμα αριθμητικής υπερχείλισης (overflow) το οποίο σηματοδοτεί ότι το αποτέλεσμα βρίσκεται έξω από τα αποδεκτά όρια (-8 έως 7 όταν αριθμός bit N=4). Για την ορθή διαχείριση του αποτελέσματος ορίζουμε ένα ενδιάμεσο σήμα `sum` με N+1 bits το οποίο περιέχει το συνολικό άθροισμα, όπως υπολογίζεται στη γραμμή 16. Όπως παρατηρούμε, τα σήματα `X` και `Y` επεκτείνονται με τη βοήθεια του τελεστή συνένωσης (&), ώστε να τηρείται η προαπαιτούμενη συνθήκη ότι ο μεγαλύτερος από τους δύο προσθετέους έχει αριθμό bits ίσο με το αποτέλεσμα `sum`. Το αποτέλεσμα `sum` ορίστηκε με N+1 bits (δηλαδή εδώ με 5 bits). Στη συνέχεια, τα 4 πρώτα bits αποδίδονται στην έξοδο S που αντιπροσωπεύει το τελικό άθροισμα με τη μορφή προσημασμένου αριθμού. Το πλέον σημαντικό bit του `sum` αντιπροσωπεύει το τελικό κρατούμενο. Σε περίπτωση που το άθροισμα είναι εκτός της προβλεπόμενης περιοχής το σήμα `overflow` που ορίζεται στη γραμμή 19 γίνεται λογικό 1. Η προσομοίωση της λειτουργίας του κώδικα φαίνεται στο σχήμα 3.4. Υπενθυμίζεται ότι το αρνητικό αποτέλεσμα παριστάνεται με το συμπλήρωμα ως προς 2. Με αντίστοιχο τρόπο σχεδιάζεται αθροιστής περισσότερων bits.

```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.all;
3 USE ieee.std_logic_signed.all;
4 ------------------------------
5 ENTITY arithm1 IS
6 PORT(X, Y : IN std_logic_vector(3 downto 0);
7      Cin : IN std_logic;
8        S : OUT std_logic_vector(3 downto 0);
9        sum : BUFFER std_logic_vector(4 downto 0);
10       Cout : OUT std_logic;
11       overflow : OUT std_logic);
12 END arithm1;
13 -------------------------------
14 ARCHITECTURE addition OF arithm1 IS
15 BEGIN
16          sum<=('0'& X) + ('0'& Y) + Cin;
17          S<=sum(3 downto 0);
18          Cout<=sum(4);
19          overflow<=sum(4) xor X(3) xor Y(3) xor sum(3);
20 END addition;
```

Κώδικας 3.5 Κύκλωμα αθροιστή 4-bit με σήματα `std_logic_vector` και το πακέτο *std_logic_signed*



Σχήμα 3.4 Προσομοίωση του αθροιστή στο λογισμικό Quartus II

3.2.4 Τύποι προσημασμένων και μη προσημασμένων αριθμών (`SIGNED/UNSIGNED`)

Οι τύποι SIGNED και UNSIGNED ορίζονται σε δύο διαφορετικά πακέτα της βιβλιοθήκης ieee. Το ένα είναι shareware της εταιρίας Synopsis και ονομάζεται *std_logic_arith*. Το άλλο είναι επίσημα προτυποποιημένο πακέτο και ονομάζεται *numeric_std*.

Το πακέτο *std_logic_arith* είναι η βάση για τον ορισμό των δύο πακέτων *std_logic_signed* και *std_logic_unsigned* που αναφέρθηκαν στην προηγούμενη παράγραφο. Όταν αντί γι' αυτά χρησιμοποιούμε το πακέτο *std_logic_arith*, θα πρέπει να μετατρέψουμε τα σήματα τύπου *std_logic_vector* σε `SIGNED` ή `UNSIGNED` και στη συνέχεια να εκτελέσουμε τις αριθμητικές πράξεις +, –, *, / κλπ. με βάση αυτά τα σήματα. Για παράδειγμα:

```
signal X_sgn, Y_sgn : SIGNED(7 downto 0);
signal sum, sub : SIGNED(8 downto 0);
sum<=X_sgn+Y_sgn;
sub<=X_sgn-Y_sgn;
```

Τα σήματα που θα χρησιμοποιηθούν ως ορίσματα αριθμητικών πράξεων μπορούν να οριστούν ως προσημασμένοι `SIGNED` ή μη-προσημασμένοι `UNSIGNED` στη δήλωση θυρών στην οντότητα. Εναλλακτικά, τα σήματα θυρών μπορούν να οριστούν ως τύπου `std_logic_vector` και η μετατροπή τους να γίνει στο σώμα της αρχιτεκτονικής.

Παράδειγμα 3.5 Να σχεδιαστεί ένα κύλωμα που εκτελεί πρόσθεση (ή αφαίρεση) προσημασμένων αριθμών, με βάση το πακέτο *std_logic_arith*.

Λύση: Στον κώδικα 3.6 περιγράφεται ένα κύλωμα που εκτελεί πρόσθεση (ή αφαίρεση) προσημασμένων αριθμών, με βάση το πακέτο *std_logic_arith*. Η επέκταση των σημάτων με τον τελεστή συνένωσης σχολιάστηκε στην προηγούμενη παράγραφο. Στο σχήμα 3.5 παρουσιάζεται το αποτέλεσμα της προσομοίωσης όταν ο κώδικας αυτός χρησιμοποιείται για να εκτελέσει την πράξη της αφαίρεσης προσημασμένων αριθμών (όπως στη γραμμή 20).

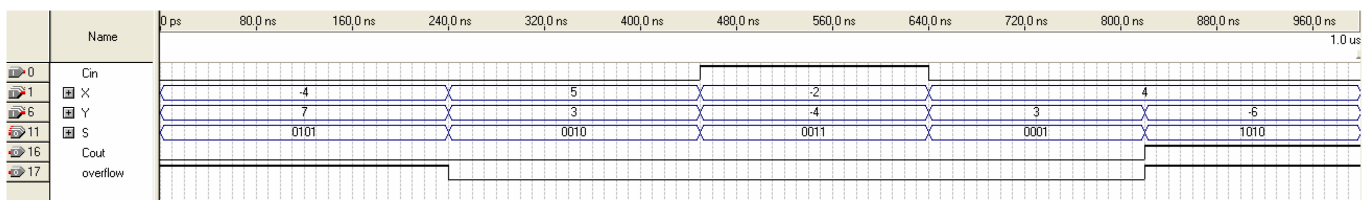
```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.all;
3 USE ieee.std_logic_arith.all;
4 ------------------------------
5 ENTITY signed_arith IS
6 PORT(X,Y :IN std_logic_vector(3 downto 0);
7      Cin :IN std_logic;
```

```
8       S :OUT std_logic_vector(3 downto 0);
9       Cout :OUT std_logic;
10      Overflow :OUT std_logic);
11 END signed_arith;
12 -------------------------------
13 ARCHITECTURE addition OF signed_arith IS
14 signal X_sgn, Y_sgn, S_sgn : SIGNED(3 downto 0);
15 signal sum_sgn : SIGNED(4 downto 0);
16 BEGIN
17    X_sgn<=SIGNED(X);
18    Y_sgn<=SIGNED(Y);
19    sum_sgn<=('0'& X_sgn) + ('0'& Y_sgn) + Cin;
20   --sum_sgn<=('0'& X_sgn) - ('0'& Y_sgn) + Cin;--subtraction
21    S_sgn<=sum_sgn(3 downto 0);
22    Cout<=sum_sgn(4);
23    overflow<=sum_sgn(4) xor X_sgn(3) xor Y_sgn(3) xor sum_sgn(3);
24    S<=std_logic_vector(S_sgn);
25 END addition;
----------------------------------
```

Κώδικας 3.6 Αθροιστής 4-bits με χρήση σημάτων `signed`, με βάση το πακέτο *std_logic_arith*.

Το πακέτο *numeric_std* χρησιμοποιείται κι αυτό για να ορίσει (μη) προσημασμένους αριθμούς με τους τύπους `SIGNED/UNSIGNED`. Άρα είναι αντίστοιχο με το *std_logic_arith*. Πάντως, τα δύο πακέτα δεν είναι ισοδύναμα, άρα δεν μπορούν να χρησιμοποιηθούν μαζί. Το πακέτο *numeric_std* επιτρέπει επιπλέον τη χρήση λογικών τελεστών με σήματα `(UN)SIGNED`, κάτι που δεν συμβαίνει με το *std_logic_arith*.

Ένας γενικός κανόνας είναι να χρησιμοποιούμε εισόδους/εξόδους με τύπο `std_logic/ std_logic_vector` και αν χρειάζεται να εκτελέσουμε αριθμητικές πράξεις να μετατρέπουμε τους τύπους των σημάτων στο εσωτερικό του κυκλώματος σε τύπους `SIGNED` και `UNSIGNED`.



Σχήμα 3.5 Προσομοίωση της αφαίρεσης προσημασμένων αριθμών

Παράδειγμα 3.6 Στον κώδικα 3.7 παρουσιάζεται ένας γενικός αθροιστής και αφαιρέτης *N*-bits, με χρήση του πακέτου *numeric_std* και εισόδους/εξόδους τύπου `std_logic_vector`. Ας προσεχθεί η χρήση της δήλωσης `GENERIC` για τη δήλωση του αριθμού των bits.

```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.all;
3 USE ieee.numeric_std.all;
4 --------------------------------------------------------
5 ENTITY signed_numeric IS
6     GENERIC(N : INTEGER :=4);
7 PORT(X,Y :IN std_logic_vector(N-1 downto 0);
8       Cin :IN std_logic;
9       Add, Sub :OUT std_logic_vector(N-1 downto 0);
10      Cout_sum, Cout_sub: OUT std_logic;
11      overflow_sum, overflow_sub : OUT std_logic);
12 END signed_numeric;
13 --------------------------------------------------------
14 ARCHITECTURE sum_sub OF signed_numeric IS
15 signal X_sgn, Y_sgn : SIGNED(N-1 downto 0);
16 signal sum_sgn, sub_sgn : SIGNED(N downto 0);
17 BEGIN
18     X_sgn<=SIGNED(X);
19     Y_sgn<=SIGNED(Y);
20     sum_sgn<=('0'& X_sgn) + ('0'& Y_sgn) + ('0' & Cin);
21     sub_sgn<=('0'& X_sgn) - ('0'& Y_sgn) + ('0' & Cin);
22     Add<=std_logic_vector(sum_sgn(N-1 downto 0));
23     Sub<=std_logic_vector(sub_sgn(N-1 downto 0));
24     Cout_sum<=sum_sgn(N);
25     overflow_sum<=sum_sgn(N)   xor   X_sgn(N-1)   xor   Y_sgn(N-1)   xor
sum_sgn(N-1);
26     Cout_sub<=sub_sgn(N);
27     overflow_sub<=sub_sgn(N)   xor   X_sgn(N-1)   xor   Y_sgn(N-1)   xor
sub_sgn(N-1);
28 END sum_sub;
```

Κώδικας 3.7 Γενικός αθροιστής και αφαιρέτης προσημασμένων αριθμών *N* bits με βάση το αριθμητικό πακέτο *numeric_std*.

Στα παραπάνω παραδείγματα φαίνεται ο τρόπος δήλωσης των πακέτων στο τμήμα βιβλιοθηκών του κώδικα.

Ανάμεσα στους τύπους `SIGNED` και `UNSIGNED` μπορούν να γίνουν άμεσες μετατροπές (type casting), με χρήση συναρτήσεων μετατροπής: Έτσι, αν το σήμα `X` είναι `std_logic_vector` μπορεί να μετατραπεί σε μη προσημασμένο ή σε προσημασμένο, αντίστοιχα, με τις συναρτήσεις:

```
X_unsgn<=UNSIGNED(X);
X_sgn<=SIGNED(X);
```

Ένα προσημασμένο σήμα μπορεί να μετατραπεί σε `std_logic_vector` με τη συνάρτηση:

```
Y<=std_logic_vector(X_sgn);
```

Η χρήση αυτών των συναρτήσεων φαίνεται στους κώδικες 3.6 και 3.7.

3.3 Τύποι που ορίζονται από τον χρήστη (user-defined types)

Εκτός από τους προκαθορισμένους και προτυποποιημένους τύπους δεδομένων που παρουσιάστηκαν παραπάνω, ο κάθε χρήστης μπορεί να δημιουργήσει τους δικούς του τύπους δεδομένων, ώστε να χειριστεί πιο εύκολα τη σχεδίαση ενός συστήμαος. Η δήλωση του τύπου δεδομένων μπορεί να γίνει στο σώμα δηλώσεων της αρχιτεκτονικής ή σε ξεχωριστό πακέτο (package), όπως είναι και το πιο βολικό σε μεγάλες σχεδιάσεις. Η δήλωση ενός τύπου δεδομένων από το χρήστη γίνεται με την εξής γενική μορφή:

```
TYPE όνομα_τύπου IS περιγραφή τύπου;
```

Οι τύποι δεδομένων που ορίζει ο χρήστης εμπίπτουν γενικά στις ιδιες κατηγορίες, όπως οι προκαθορισμένοι. Μία κατηγορία τύπων είναι οι «βαθμωτοί» (scalar), όπου τα αντικείμενα δεδομένων λαμβάνουν μία τιμή από ένα σύνολο (ας θυμηθούμε τον τύπο `BIT`), ενώ μία άλλη κατηγορία είναι οι «σύνθετοι» (composite), που περιλαμβάνουν πολλές τιμές (όπως ο προκαθορισμένος τύπος `BIT_VECTOR`). Μια μορφή σύνθετου τύπου είναι οι πίνακες (`ARRAYS`), οι οποίοι μπορεί να είναι μονοδιάστατοι, διδιάστατοι ή τρισδιάστατοι. Τα στοιχεία τους προσπελάζονται με τη βοήθεια δεικτών, όπως στις συμβατικές γλώσσες. Κάθε σήμα `q` τύπου `std_logic_vector` είναι πίνακας, με στοιχεία π.χ. `q(7)`, `q(6)`, …, `q(0)`. Επίσης, ένας σύνθετος τύπος μπορεί να είναι RECORD (εγγραφή), οπότε περιλαμβάνει συλλογές δεδομένων, που μπορεί να ανήκουν και σε διαφορετικούς τύπους. Δεν θα ασχοληθούμε με records σ' αυτό το βιβλίο. Οι πίνακες, όμως, αποδεικνύονται ιδιαίτερα χρήσιμοι, καθώς επιτρέπουν μεγάλη ευελιξία στη σύνταξη του κώδικα.

3.3.1 Ακέραιοι τύποι ορισμένοι από τον χρήστη

Αν ο σχεδιαστής ενός συστήματος πρέπει να διαχειριστεί ένα συγκεκριμένο σύνολο ακέραιων τιμών, τότε μπορεί να ορίσει μια ειδική κατηγορία ακεραίων που να εξυπηρετεί το σκοπό του και στη συνέχεια να αναφέρεται σ' αυτήν, με το όνομά της. Ογενικός τρόπος ορισμού ακέραιων τύπων είναι:

```
TYPE όνομα_τύπου IS RANGE αρχική_τιμή TO τελική_τιμή;
```

Για παράδειγμα:

```
TYPE temperature IS RANGE 0 TO 273;
```

Στη συνέχεια, για να ορίσουμε ένα σήμα ως τύπου `temperature` θα πρέπει να το δηλώσουμε ως τέτοιο:

```
SIGNAL s : temperature;
```

3.3.2 Τύποι απαρίθμησης (enumeration)

Ο τύπος απαρίθμησης περιλαμβάνει ένα σύνολο τιμών, με τη μορφή συμβόλων, σε μια ορισμένη σειρά. Το σύνολο πρέπει να περιλαμβάνεται («απαριθμείται») σε μια λίστα:

```
TYPE όνομα_τύπου IS (λίστα τιμών)
```

Ένας προκαθορισμένος τύπος απαρίθμησης είναι ο `std_ulogic`, που ορίζεται ως εξής:

```
TYPE STD_ULOGIC IS ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
```

Σημειώνεται ότι τα σύμβολα τοποθετούνται σε μονά εισαγωγικά επειδή στον συγκεκριμένο τύπο ορίζονται ως χαρακτήρες (αφού τα 0 και 1 είναι ήδη δεσμευμένα ως αριθμοί).
Ο χρήστης μπορεί να ορίσει δικούς του τέτοιους τύπους για να διαχειριστεί ιδαίτερα προβλήματα. Για παράδειγμα σε μια μηχανή πεπερασμένων καταστάσεων μπορεί να χρειάζεται να διαχειριστούμε τέσσερις καταστάσεις A, B, C, D. Ορίζουμε, τότε, έναν τύπο δεδομένων που να περιλαμβάνει αυτές τις συμβολικές τιμές:

```
TYPE fsm_states IS (A, B, C, D, E);
TYPE my_symbols IS (first, second, third);
```

Στη συνέχεια, κάθε σήμα τύπου `fsm_states` ή `my_symbols` μπορεί να πάρει ως τιμή μόνον μια από τις τιμές της λίστας. Στο επίπεδο της προσομοίωσης και της σύνθεσης ο compiler κωδικοποιεί τις παραπάνω καταστάσεις σε μια κατάλληλη δυαδική ακολουθία.

3.3.3 Υποτύποι δεδομένων

Οι υποτύποι ορίζουν μια ειδική υποκατηγορία ενός τύπου δεδομένων. Για παράδειγμα, μπορούμε να ορίσουμε ως υποτύπο του τύπου `SIGNED`, αυτόν που περιγράφει σήματα που λαμβάνουν προσημασμένες τιμές με οκτώ bits:

```
SUBTYPE sample IS SIGNED(7 downto 0);
```

Στη συνέχεια μπορούμε να ορίσουμε τον τύπο ενός σήματος ότι ανήκει στον υποτύπο sample:

```
SIGNAL normal : sample;
```

Συνηθισμένοι υποτύποι είναι αυτοί του τύπου INTEGER:

```
SUBTYPE my_int IS INTEGER RANGE -256 TO 255;
```

3.3.4 Τύποι πινάκων

Ένας πίνακας μπορεί να είναι μονοδιάστατος ή περισσότερων διαστάσεων και δηλώνεται γενικά ως εξής:

```
TYPE όνομα_τύπου IS ARRAY (περιοχή δεικτών) OF τύπος_στοιχείων;
```

Για παράδειγμα ένας πίνακας ακεραίων με πέντε στοιχεία μπορεί να οριστεί ως εξής:

```
TYPE int_matrix IS ARRAY(1 TO 5) OF INTEGER;
```

Μια σταθερά (constant) αυτού του τύπου θα οριστεί ως εξής:

```
CONSTANT c1: int_matrix(1 TO 5) :=(8, -2, 10, -20, 15);
```

Προφανώς, θα ισχύει ότι `c1(1)=8, c1(2)=-2` κλπ.
Προκειμένου να αφήσουμε ελεύθερη την περιοχή δεικτών να καταλάβει όλο το εύρος των ακεραίων από 0 έως το άνω όριο, γράφουμε τη δήλωση:

```
TYPE my_type IS ARRAY(NATURAL RANGE <>) OF INTEGER;
```

Ο `τύπος_στοιχείων` μπορεί να είναι οποιοσδήποτε τύπος δεδομένων. Για παράδειγμα, μπορεί να είναι `std_logic_vector` 8-bit. Έστω, λοιπόν, ότι θέλουμε να δηλώσουμε έναν τύπο πίνακα με όνομα `OneD` ο οποίος περιγράφει γενικά μονοδιάστατους πίνακες, με στοιχεία `std_logic_vector` 8-bit. Τότε, γράφουμε:

```
TYPE OneD IS ARRAY(NATURAL RANGE <>) OF std_logic_vector(7 downto 0);
```

Στην περίπτωση αυτή, ένα σήμα `delay` με τύπο δεδομένων `OneD` και τέσσερα στοιχεία θα δηλωθεί ως εξής:

```
SIGNAL delay : OneD(0 TO 3);
```

Είναι προφανές ότι στην πραγματικότητα το «σήμα» `delay` είναι ένας πίνακας τεσσάρων σημάτων `delay(0)` έως `delay(3)`. Επειδή κάθε στοιχείο του πίνακα είναι μονοδιάστατο (1D) σήμα 8-bit, προφανώς ο πίνακας delay του τύπου `OneD` μπορεί να θεωρηθεί ότι σε επίπεδο bits είναι μια δομή 1Dx1D.

Είναι δυνατό να οριστεί ένας πίνακας με στοιχεία που ανήκουν σε τύπο δεδομένων που ορίστηκε από τον χρήστη. Έτσι, ο τύπος `sample` που ορίστηκε στην παράγραφο Π3.3.3 μπορεί να χρησιμοποιηθεί για να δημιουργηθεί ένας τύπος πίνακα:

```
SUBTYPE sample IS SIGNED(7 downto 0);

TYPE OneD IS ARRAY(NATURAL RANGE <>) OF sample;
```

Τυπικό παράδειγμα συστήματος που διαχειρίζεται δεδομένα σε μορφή μονοδιάστατου πίνακα είναι η επεξεργασία ηχητικού σήματος, που ρέει μέσα στο ψηφιακό σύστημα.
Τέλος, πολύ χρήσιμοι σε σύνθετα προβλήματα είναι οι τύποι που περιγράφουν διδιάστατους πίνακες στοιχείων. Για παράδειγμα, αν θέλουμε να δηλώσουμε έναν τύπο δεδομένων με όνομα `TwoD` ο οποίος περιγράφει γενικά διδιάστατους πίνακες με στοιχεία 8-bit `SIGNED`, γράφουμε:

```
TYPE TwoD IS ARRAY(NATURAL RANGE <>, NATURAL RANGE <>) OF sample;
```

Στη συνέχεια, ορίζουμε έναν πίνακα σημάτων του τύπου `TwoD`:

```
SIGNAL conv : TwoD(0 to 2, 0 to 2);
```

Δηλαδή, ο τύπος `TwoD` ορίζεται με δείκτες σε όλη την περιοχή `natural`, αλλά το σήμα `conv` δηλώνεται ως τύπος `TwoD` με εννέα στοιχεία: `conv(0,0)`, `conv(0,1)`, `conv(0,2)`, `conv(1,0)`, `conv(1,1)`, …, `conv(2,2)`.
Τυπικό παράδειγμα συστήματος που διαχειρίζεται δεδομένα σε μορφή διδιάστατου πίνακα είναι η επεξεργασία σήματος εικόνας, όταν οι τιμές των εικονοστοιχείων ρέουν μέσα στο ψηφιακό σύστημα.

3.3.5 Μετατροπή τύπων

Όπως έχει αναφερθεί, η VHDL διακρίνει με αυστηρότητα τους τύπους δεδομένων. Ο μεταγλωττιστής έχει ελάχιστη έως καθόλου ευελιξία όταν η σύνταξη του προγράμματος αναμιγνύει διαφορετικούς τύπους δεδομένων, οπότε παράγει μηνύματα σφάλματος. Έτσι, αν προσπαθήσουμε να αναθέσουμε μια τιμή τύπου `integer` σε ένα σήμα με τύπο `std_logic_vector`, τότε ο compiler θα παράγει μήνυμα σφάλματος, ακόμη κι αν οι τιμές βρίσκονται σε συμβατά μεταξύ τους όρια. Έστω ότι γίνονται οι παρακάτω δηλώσεις σημάτων

```
SIGNAL m :integer RANGE 0 to 255 :=128;
SIGNAL y :std_logic_vector(7 downto 0);
```

Τότε, η παρακάτω ανάθεση της τιμής του `m` στο `y`

```
y<=m;
```

θα παράγει στο Quartus II το εξής μήνυμα σφάλματος

```
Error (10476): type of identifier "m" does not agree with its usage as
"std_logic_vector" type
```

Σ' αυτές τις περιπτώσεις ο προγραμματιστής θα πρέπει να ελέγχει αν έχει ορίσει τα κατάλληλα πακέτα περιγραφής τύπων, καθώς και πιθανά σφάλματα στις αναθέσεις τιμών ανάμεσα σε αντικείμενα ασύμβατων μεταξύ τους τύπων.
Τα διάφορα πακέτα μας δίνουν τη δυνατότητα να μετατρέπουμε τύπους δεδομένων από τη μια μορφή στην άλλη. Έτσι, η παραπάνω λανθασμένη ανάθεση μπορεί να διορθωθεί, αν μετατρέψουμε πρώτα το σήμα `integer` σε σήμα `std_logic_vector`:

```
y<=conv_std_logic_vector(m,8);
```

Η παραπάνω συνάρτηση

```
conv_std_logic_vector(<signal>, <number_of_bits>)
```

περιέχεται στο πακέτο *std_logic_arith*. Άρα, για να γίνει η μετατροπή πρέπει να έχει οριστεί το πακέτο στην περιοχή δηλώσεων βιβλιοθήκης, με μια δήλωση `USE`. Το όρισμα `<number_of_bits>` είναι απαραίτητο όταν το εύρος ενός σήματος δεν προκύπτει με αυτονόητο τρόπο.
Η μετατροπή τύπου δεδομένων γενικά ανήκει σε μια από τις εξής περιπτώσεις:

Αυτόματη μετατροπή τύπου

Οι τύποι `std_logic` και `std_logic_vector` έχουν τον ίδιο «βασικό τύπο» (`std_logic`). Έτσι, ένα στοιχείο σήματος `std_logic_vector` μπορεί να αποδοθεί ευθέως σε σήμα `std_logic`:

```
SIGNAL s1 : std_logic_vector(7 downto 0);
SIGNAL s2 : std_logic;
-----------------------------
s2<=s1(7);
```

Αντίστοιχα ισχύουν για τα σήματα `bit` και `bit_vector`.

Μορφοποίηση τύπου (type casting)

Έστω οι δηλώσεις σημάτων:

```
SIGNAL m1 : std_logic_vector(3 downto 0);
SIGNAL f1 : SIGNED(3 downto 0);
----------------------------------------------
```

Οι τύποι `std_logic_vector` και `SIGNED/UNSIGNED` έχουν τον ίδιο βασικό τύπο (`std_logic`) και δεικτοδότηση (`natural`), οπότε ισχύουν οι παρακάτω μορφοποιήσεις:

```
m1<=std_logic_vector(f1);
f1<=SIGNED(m1);
```

Συναρτήσεις μετατροπής τύπου

Τα πακέτα που περιγράφουν τους τύπους δεδομένων υποστηρίζουν συναρτήσεις μετατροπής τύπων, σύμφωνα με τον πίνακα 3.1. Προφανώς, για να λειτουργήσουν οι συναρτήσεις μετατροπής πρέπει να έχουν δηλωθεί τα αντίστοιχα πακέτα.

From	To	Type conversion function	Package of origin
INTEGER	STD_LOGIC_VECTOR	conv_std_logic_vector(a, cs)	std_logic_arith
	UNSIGNED	to_unsigned(a, cs) conv_unsigned(a, cs)	numeric_std std_logic_arith
	SIGNED	to_signed(a, cs) conv_signed(a, cs)	numeric_std std_logic_arith
	UFIXED	to_ufixed(a, cs)	fixed_generic_pkg
	SFIXED	to_sfixed(a, cs)	fixed_generic_pkg
	FLOAT	to_float(a, cs)	float_generic_pkg
BIT_VECTOR	STD_LOGIC_VECTOR	to_stdlogicvector(a, cs)	std_logic_1164
STD_LOGIC_VECTOR	INTEGER	conv_integer(a, cs) conv_integer(a, cs) to_integer(a, cs)	std_logic_signed std_logic_unsigned numeric_std_unsigned
	BIT_VECTOR	to_bitvector(a, cs)	std_logic_1164
	UNSIGNED	unsigned(a) (*) unsigned(a) (*)	numeric_std std_logic_arith
	SIGNED	signed(a) (*) signed(a) (*)	numeric_std std_logic_arith
	UFIXED	to_ufixed(a, cs)	fixed_generic_pkg
	SFIXED	to_sfixed(a, cs)	fixed_generic_pkg
	FLOAT	to_float(a, cs)	float_generic_pkg
UNSIGNED and SIGNED	INTEGER	to_integer(a, cs) conv_integer(a, cs)	numeric_std std_logic_arith
	STD_LOGIC_VECTOR	std_logic_vector(a) (*) std_logic_vector(a) (*) conv_std_logic_vector(a, cs)	numeric_std std_logic_arith std_logic_arith
	UNSIGNED	conv_unsigned(a, cs)	std_logic_arith
	SIGNED	conv_signed(a, cs)	std_logic_arith
	UFIXED (unsigned only)	to_ufixed(a, cs)	fixed_generic_pkg
	SFIXED (signed only)	to_sfixed(a, cs)	fixed_generic_pkg

Πίνακας 3.1 Συναρτήσεις μετατροπής και μορφοποίησης τύπων δεδομένων

4 Τελεστές, πράξεις και χαρακτηριστικά

Στη VHDL μπορούμε να κάνουμε αριθμητικές, λογικές και σχεσιακές πράξεις χρησιμοποιώντας τους αντίστοιχους τελεστές. Γενικά, υποστηρίζονται τα εξής είδη τελεστών
Τελεστές ανάθεσης (ή αντιστοίχησης), λογικοί τελεστές, αριθμητικοί τελεστές, τελεστές σύγκρισης (σχεσιακοί τελεστές), τελεστές ολίσθησης, ο τελεστής συνένωσης και ορισμένοι άλλοι.

4.1 Τελεστές ανάθεσης

Οι τελεστές αυτοί αποδίδουν τιμές σε σήματα, μεταβλητές και σταθερές. Προφανώς, τους έχουμε συναντήσει σε όλα τα παραδείγματα που δώσαμε μέχρι τώρα, στις προηγούμενες παραγράφους. Χάριν πληρότητας, αναφέρουμε συνοπτικά στο σημείο αυτό τη χρήση των τελεστών ανάθεσης. Έστω ότι έχουμε δηλώσει τα παρακάτω αντικείμενα δεδομένων

```
SIGNAL s1, s2, s3 : std_logic_vector(3 downto 0);
VARIABLE v1, v2 : unsigned(3 downto 0);
VARIABLE v3 : bit_vector(7 downto 0);
```

Οι τελεστές ανάθεσης χρησιμοποιούνται όπως αναφέρεται στον παρακάτω πίνακα

Τελεστής	**Λειτουργία**	**Παράδειγμα**
`<=`	Αναθέτει τιμές σε σήματα	`<όνομα_σήματος> <=<έκφραση>` `s1<=s2+s3;`

`:=`	Αναθέτει τιμές σε μεταβλητές
	Δίνει αρχικές τιμές σε σήματα, κατά τη δήλωσή τους
		`<όνομα_μεταβλητής> :=<έκφραση>`
		`v1 := s1 OR s2;`
		`v2 := "1010";`

`=>`	Δίνει τιμές σε στοιχεία πινάκων	`v3:=(0=>'1', 1=>'0', OTHERS=>'0');`

Πίνακας 4.1 Τελεστές ανάθεσης και η λειτουργία τους

Η ανάθεση της τελευταίας γραμμής του πίνακα αποδίδει στη μεταβλητή `v3` την τιμή `"0001"`.

4.2 Λογικοί τελεστές

Η VHDL όπως είπαμε και παραπάνω υποστηρίζει τις λογικές πράξεις. Τα αποτελέσματα των λογικών πράξεων είναι του ιδίου τύπου και μεγέθους με τους τελεστέους. Πράξεις μεταξύ τύπων διαφορετικού μεγέθους παράγουν σφάλμα.

Τελεστής	Λειτουργία
Not	Αντιστροφή
And	Και
Nand	Όχι Και
Or	Ή
Nor	Ούτε
Xor	Αποκλειστικό Ή
Xnor	Αποκλειστικό Ούτε

Πίνακας 4.2 Τελεστές Λογικών Πράξεων

Όλες οι λογικές πράξεις, έχουν την ίδια προτεραιότητα, με εξαίρεση τον τελεστή Not που έχει μεγαλύτερη προτεραιότητα από τους υπόλοιπους. Για τον λόγο αυτό όταν σε μία δήλωση έχουμε πολλές λογικές πράξεις, είναι καλό να χρησιμοποιούμε παρενθέσεις για να δηλώνουμε την προτεραιότητα της κάθε μιας.

4.3 Σχεσιακοί τελεστές

Οι σχεσιακές πράξεις κάνουν συγκρίσεις μεταξύ σημάτων ή μεταβλητών, και ελέγχουν αν υπάρχει ισότητα ή ανισότητα μεταξύ τους. Όλες οι σχεσιακές πράξεις έχουν την ίδια προτεραιότητα μεταξύ τους και μεγαλύτερη από τις λογικές πράξεις. Το αποτέλεσμα που προκύπτει από μια τέτοια πράξη είναι πάντα τύπου Boolean, δηλαδή TRUE ή FALSE.

Τελεστής	Λειτουργία
=	Ισότητα
/=	Διάφορο
<	Μικρότερο
<=	Μικρότερο ή ίσο
>	Μεγαλύτερο
>=	Μεγαλύτερο ή ίσο

Πίνακας 4.3 Τελεστές Σχεσιακών Πράξεων

Παράδειγμα 4.1: Να υλοποιηθεί ένας **συγκριτής μεγέθους** δύο μη-προσημασμένων αριθμών a, b με εύρος 4-bit. Ο συγκριτής να παράγει λογικό 1 στην έξοδο, όταν a>b.

Λύση 1: Μία λύση είναι να χρησιμοποιήσουμε το πακέτο *numeric_std* προκειμένου να μετατρέψουμε τις εισόδους τύπου `std_logic/std_logic_vector` σε `UNSIGNED`, με τη βοήθεια των ενδιάμεσων σημάτων `a_uns`, `b_uns`.

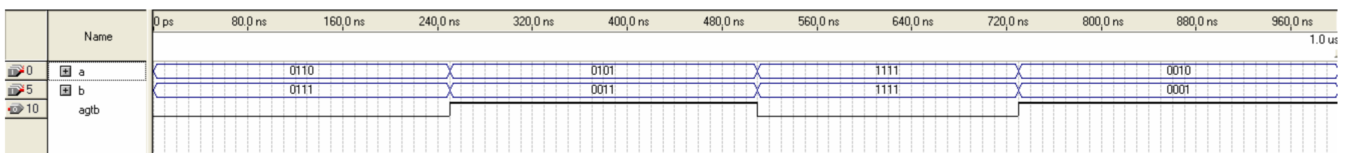
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
------------------------------
```

```
ENTITY comparator1 IS
      PORT(a, b : IN std_logic_vector(3 downto 0);
           agtb : OUT std_logic);
END comparator1;
------------------------------
ARCHITECTURE compare OF comparator1 IS
SIGNAL a_uns, b_uns : UNSIGNED(3 downto 0);
BEGIN
      a_uns<=UNSIGNED(a);
      b_uns<=UNSIGNED(b);
      agtb<='1' WHEN a>b ELSE '0';
END compare;
```

Λύση 2: Μια ισοδύναμη λύση είναι να χρησιμοποιήσουμε το πακέτο *std_logic_unsigned*, που επιτρέπει την άμεση χρήση σημάτων του τύπου `std_logic/std_logic_vector` σε σχεσιακές πράξεις. Αντίστοιχα, μπορούμε να σχεδιάσουμε έναν συγκριτή προσημασμένων αριθμών με χρήση του πακέτου *std_logic_signed*.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
------------------------------
ENTITY comparator1 IS
      PORT(a, b : IN std_logic_vector(3 downto 0);
           agtb : OUT std_logic);
END comparator1;
------------------------------
ARCHITECTURE compare OF comparator1 IS
BEGIN
      agtb<='1' WHEN a>b ELSE '0';
END compare;
------------------------------
```

Κώδικας 4.1 Συγκριτές προσημασμένων αριθμών, με χρήση αριθμητικών πακέτων



Σχήμα 4.1 Προσομοίωση του κώδικα που περιγράφει τον συγκριτή αριθμών 4-bit.

Στο παραπάνω σχήμα 4.1 παρουσιάζεται το αποτέλεσμα της προσομοίωσης του κώδικα στο περιβάλλον Quartus II. Η λύση 1 είναι ισοδύναμη με τη λύση 2.

4.4 Τελεστές αριθμητικών πράξεων

Οι τελεστές αριθμητικών πράξεων παρατίθενται στον πίνακα 4.4, μαζί με τη λειτουργία τους. Στην τρίτη στήλη φαίνεται η σύνταξη της πράξης. Στην τελευταία στήλη δίνεται μια εξήγηση της πράξης, όπου χρειάζεται.

Τελεστής	Πράξη	Σύνταξη	Εξήγηση
`+`	Πρόσθεση	`result<=x + y;`	
`-`	Αφαίρεση	`result<=x-y;`	
`*`	Πολλαπλασιασμός	`result<=x * y;`	
`/`	Διαίρεση	`result<=x / y;`	
`**`	Ύψωση σε δύναμη	`result<=x **2;`	
`ABS`	Απόλυτη τιμή	`result<=ABS x;`	Απόλυτη τιμή
`REM`	Υπόλοιπο (remainder)	`result<=x REM y;`	Υπόλοιπο διαίρεσης x/y
`MOD`	Modulo	`result<=x MOD y;`	x REM y +a*y (a=1 αν x και y έχουν ίδιο πρόσημο, αλλιώς 0)

Πίνακας 4.4 Τελεστές αριθμητικών πράξεων

Ο παραπάνω τελεστές είναι πάντα συνθέσιμοι και οδηγούν στα αντίστοιχα αριθμητικά κυκλώματα, όταν οι τύποι δεδομένων που χρησιμοποιούμε είναι οι προκαθορισμένοι `integer`, `natural`, `positive`. Όταν οι τύποι δεδομένων είναι `(un)signed` τότε είναι απαραίτητο να κάνουμε χρήση του αριθμητικού πακέτου *numeric_std* ή *std_logic_arith.* Για να χρησιμοποιήσουμε τους αριθμητικούς τελεστές με τον τύπο δεδομένων `std_logic_vector` πρέπει να κάνουμε χρήση του πακέτου *std_logic_unsigned* ή *std_logic_signed.*
Τα σχετικά με τους αριθμητικούς τελεστές και ιδιαίτερα για την πρόσθεση αναπτύχθηκαν και στις παραγράφους 3.2.3 και 3.2.4. Στο παρακάτω παράδειγμα περιγράφεται ο σχεδιασμός ενός πολλαπλασιαστή.

Παράδειγμα 4.2 Να περιγραφεί σε VHDL ένα κύκλωμα προσημασμένου πολλαπλασιασμού, όπου οι είσοδοι/έξοδοι υπακούουν στο πρότυπο `std_logic_vector`.

Λύση: Προκειμένου να υλοποιηθεί ο προσημασμένος πολλαπλασιασμός ανάμεσα σε δύο ορίσματα `x`, `y` με εύρος *N* bits, πρέπει να προβλεφθεί ότι το γινόμενο θα έχει εύρος ίσο με το άθροισμα των bits των ορισμάτων. Για *N*=8, η περιοχή των τιμών των δύο συντελεστών είναι από -128 έως +127. Το γινόμενο θα έχει εύρος 16 bits. Τα σήματα μετατρέπονται σε `SIGNED` κάνοντας χρήση των αντίστοιχων συναρτήσεων μετατροπής και το αποτέλεσμα του πολλαπλασιασμού επιστρέφει στην έξοδο `product` αφού μετατραπεί σε `std_logic_vector`.

```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.all;
3 USE ieee.numeric_std.all;
4 ENTITY multi IS --signed multiplication
5     PORT(x, y     : IN  std_logic_vector(7 downto 0);
6          product : OUT std_logic_vector(15 downto 0));
7 END multi;
8  ARCHITECTURE sign_prod OF multi IS
9  SIGNAL a_s, b_s : SIGNED(7 downto 0);
10 SIGNAL prod_s : SIGNED(15 downto 0);
11 BEGIN
12    a_s<=SIGNED(x);
13    b_s<=SIGNED(y);
14    prod_s<=a_s*b_s;
15    product<=std_logic_vector(prod_s);
16 END sign_prod;
```

Κώδικας 4.2 Κώδικας για τον πολλαπλασιασμό προσημασμένων σημάτων με χρήση του αριθμητικού πακέτου *numeric_std*.

Αντίστοιχα ισχύουν για τη διαίρεση, όπου όμως το αποτέλεσμα πρέπει να έχει ίσο αριθμό bits με τον διαιρετέο. Ο προσημασμένος πολλαπλασιασμός μπορεί να υλοποιηθεί και με το πακέτο *std_logic_arith*, αντί του *numeric_std*.

4.5 Τελεστές ολίσθησης

Οι τελεστές ολίσθησης χρησιμοποιούνται από τη VHDL για την ολίσθηση των bits δεδομένων τύπου vector. Υποστηρίζονται από τον προκαθορισμένο τύπο `bit_vector`. Αν γίνει χρήση του πακέτου *numeric_std*, τότε οι τελεστές αυτοί υποστηρίζονται και από τους τύπους `(un)signed`. Οι τελεστές και οι λειτουργίες τους παρουσιάζονται στον πίνακα 4.5, όπου θεωρούμε πως έγιναν οι παρακάτω δηλώσεις σημάτων:

Τελεστής	**Σύνταξη**	**Λειτουργία**	**Αποτέλεσμα (για n=2)**
SLL	`y<=x SLL n`	Λογική ολίσθηση αριστερά κατά n θέσεις. Οι θέσεις στα δεξιά γεμίζουν με ‘0’.	`y=0100`
SRL	`y<=x SRL n`	Λογική ολίσθηση δεξιά κατά n θέσεις. Οι θέσεις στα αριστερά γεμίζουν με ‘0’.	`y=0011`
SLA	`y<=x SLA n`	Ολίσθηση προς τα αριστερά. Οι θέσεις στα δεξιά γεμίζουν με το δεξιότερο bit.	`y=0111`
SRA	`y<=x SRA n`	Ολίσθηση προς τα δεξιά. Οι θέσεις στα αριστερά γεμίζουν με το αριστερότερο bit.	`y=1111`
ROL	`y<=x ROL n`	Περιστροφική ολίσθηση προς τα αριστερά.	`y=0111`
ROR	`y<=x ROR n`	Περιστροφική ολίσθηση προς τα δεξιά.	`y=0111`

Πίνακας 4.5 Τελεστές ολίσθησης και λειτουργία τους

```
SIGNAL x, y : bit_vector(3 downto 0);
x<= "1101";
```

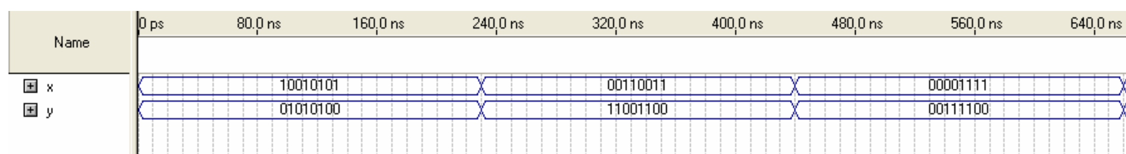
Το n στη σύνταξη των εντολών σημαίνει τον αριθμό των θέσεων που ολισθαίνει ο αριθμός.

Παράδειγμα 4.3 Ο κώδικας 4.3 παρουσιάζει μια χρήση των τελεστών ολίσθησης. Συγκεκριμένα γίνεται χρήση της εντολής `y<=x sll 2,` σε σήματα τύπου `unsigned`. Τα σήματα εισόδου/εξόδου στη δήλωση `PORT` είναι του τύπου `std_logic_vector`, ώστε να τηρείται το πρότυπο της βιομηχανίας.

Το αριθμητικό πακέτο που δηλώνεται είναι το *numeric_std*, με τη βοήθεια του οποίου ο τύπος `std_logic_vector` των σημάτων διασύνδεσης μετατρέπεται σε `unsigned`. Στο σχήμα 4.2 φαίνεται η προσομοίωση της ολίσθησης αριστερά κατά δύο θέσεις.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
-----------------------------
ENTITY shift_op IS
     PORT(x : IN std_logic_vector(7 downto 0);
          y : OUT std_logic_vector(7 downto 0));
END shift_op;
-----------------------------
ARCHITECTURE shift_test OF shift_op IS
SIGNAL x1, y1 : UNSIGNED(7 downto 0);
BEGIN
          x1<= UNSIGNED(x);
          y1<=x1 sll 2;
          y<=std_logic_vector(y1);
END shift_test;
```

Κώδικας 4.3 Χρήση των τελεστών ολίσθησης με σήματα εισόδου/εξόδου τύπου `std_logic_vector`



Σχήμα 4.2 Το αποτέλεσμα της ολίσθησης αριστερά κατά δύο θέσεις, σε περιβάλλον προσομοίωσης

4.6 Τελεστής συνένωσης

Τη χρήση του τελεστή συνένωσης (&) την είδαμε αρκετές φορές σε προηγούμενα παραδείγματα. Ο τελεστής αυτός ομαδοποιεί τιμές και μ' αυτό τον τρόπο επεκτείνει το εύρος ενός σήματος ή επιτελεί πράξεις ολίσθησης. Έστω δύο σήματα x, y:

```
SIGNAL x,y : std_logic_vector(3 downto 0);
```

Τότε, η πράξη

```
y<= x(1 downto 0) & "00";
```

ισοδυναμεί με την ολίσθηση κατά δύο θέσεις αριστερά.

4.7 Χαρακτηριστικά (attributes)

Η VHDL αποδίδει στους τύπους δεδομένων και στα αντικείμενα δεδομένων ορισμένα χαρακτηριστικά. Ένα χαρακτηριστικό μπορεί να είναι το κάτω ή το πάνω όριο της περιοχής ενός ακεραίου τύπου, ή η περιοχή δεικτών ενός τύπου πίνακα.
Διακρίνουμε χαρακτηριστικά *βαθμωτών τύπων*, χαρακτηριστικά *τύπων πινάκων* και χαρακτηριστικά *σημάτων*.

Χαρακτηριστικά βαθμωτών τύπων

Οι ακέραιοι τύποι είναι βαθμωτοί. Έστω ότι ένας τύπος integer έχει δηλωθεί ως εξής:

```
TYPE temp IS RANGE 0 to 273;
```

Τότε, ορίζονται διάφορα χαρακτηριστικά του τύπου temp, που αναφέρονται ως:

<όνομα_τύπου>'<όνομα_χαρακτηριστικού> , όπου η απόστροφος διαβάζεται "τικ".

Για παράδειγμα, το κάτω όριο της περιοχής τιμών του τύπου temp αναφέρεται ως temp'LOW και η τιμή του βέβαια είναι μηδέν. Η έκφραση:
Result<=temp'LOW αναθέτει στο σήμα result την τιμή 0. *Ο τύπος του σήματος result πρέπει να είναι temp.*
Αντίστοιχα, η έκφραση:
result<=temp'HIGH, αναθέτει στο σήμα result την τιμή 273.
Το χαρακτηριστικό Temp'ASCENDING επιστρέφει TRUE αν η περιοχή τιμών του τύπου temp είναι αύξουσα.

Χαρακτηριστικά τύπων πίνακα (array)

Έστω, τώρα, ότι έχει δηλωθεί από τον χρήστη ένας τύπος array δύο διαστάσεων, ως εξής:

```
TYPE TwoD IS ARRAY(0 to 10, 5 downto 0) OF std_logic_vector(3 downto 0);
```

Μπορούν να αναζητηθούν οι τιμές χαρακτηριστικών του τύπου TwoD, όπως:
TwoD'LEFT(N), που σημαίνει «αριστερό όριο της νιοστής περιοχής δεικτών του τύπου πίνακα TwoD. Αν N=2, τότε η έκφραση y<= TwoD'LEFT(2) επιστρέφει στο σήμα y την τιμή 5. Το σήμα y πρέπει να είναι ίδιου τύπου όπως και η περιοχή τιμών, δηλαδή integer.

Αντίστοιχα ορίζονται τα χαρακτηριστικά `TwoD'RIGHT(N)`, `TwoD'LOW(N)` κι έχουν ακέραιη τιμή 10 και 0 αντίστοιχα, για N=1. Το χαρακτηριστικό `TwoD'HIGH(2)` έχει τιμή 5. Ας σημειωθεί ότι σε μονοδιάστατο πίνακα η παρένθεση `(N)` μπορεί να παραληφθεί.
Τα παραπάνω χαρακτηριστικά μπορούν να πάρουν μέρος σε γενικευμένες δομές επανάληψης, όπως

```
FOR i IN x'LEFT to x'RIGHT LOOP…
```

Ας σημειωθεί ότι τα χαρακτηριστικά που ορίζονται για έναν βαθμωτό τύπο ή για τύπο πίνακα, μπορούν να οριστούν και για κάθε σήμα που ανήκει σ' αυτόν τύπο. Έτσι, στην παραπάνω έκφραση το `x` μπορεί να είναι ένα σήμα τύπου πίνακα, με περιοχή δεικτών από `x'LEFT` έως `x'RIGHT`. Εναλλακτικά, η παραπάνω επαναληπτική δομή μπορεί να γραφεί με τη βοήθεια του χαρακτηριστικού `x'RANGE` ως εξής:

```
FOR i IN x'RANGE LOOP…
```

Χαρακτηριστικά σημάτων

Το βασικότερο χαρακτηριστικό των σημάτων και αυτό που κυρίως χρησιμοποιούμε σ'αυτό το βιβλίο είναι το `'EVENT`. Σημαίνει τη μετάβαση του σήματος και λαμβάνει τιμές TRUE ή FALSE:

```
SIGNAL s : std_logic;
```

Το χαρακτηριστικό `s'EVENT` λαμβάνει την τιμή TRUE αν κατά τη διάρκεια του παρόντος κύκλου έγινε μετάβαση του σήματος `s` από μια κατάσταση σε άλλη (π.χ. από `'0'` σε `'1'`). Χρησιμοποιείται κατά κόρο σε ακολουθιακά συστήματα για να σημάνει τη μετάβαση σημάτων ρολογιού. Για παράδειγμα, έστω ότι σε ένα flip-flop έχει οριστεί σήμα `clk` τύπου `std_logic`. Τότε, η διατύπωση

```
IF clk'EVENT AND clk='1' THEN
….
```

σημαίνει ότι η συνθήκη της `IF` αληθεύει αν έχει συμβεί μετάβαση ρολογιού (`clk'EVENT=TRUE`) ΚΑΙ η μετάβαση αυτή είναι απο το `'0'` στο `'1'`. Με άλλα λόγια η συνθήκη αληθεύει σε κάθε θετικό μέτωπο του παλμού ρολογιού.

5 Σύγχρονες προτάσεις

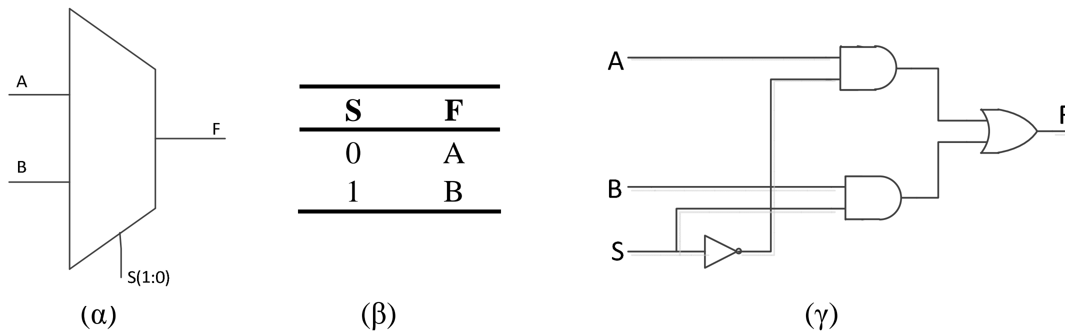
5.1 Είδη εντολών στη VHDL

Όπως έχει αναφερθεί, η VHDL στηρίζεται στις αρχές των παράλληλων γλωσσών, αφού τα ψηφιακά κυκλώματα που προσομοιώνει και περιγράφει λειτουργούν ως αυστηρά συστήματα πραγματικού χρόνου. Ως γνωστό, τα ψηφιακά κυκλώματα χωρίζονται στα συνδυαστικά (combinational) και στα ακολουθιακά (sequential) κυκλώματα. Ένα συνδυαστικό κύκλωμα μεταβάλει τα σήματα εξόδου παράλληλα με τις εισόδους, με άμεσο τρόπο, χωρίς φαινόμενα μνήμης. Ένα ακολουθιακό κύκλωμα περιέχει βρόχους ανάδρασης, ώστε οι τιμές των εξόδων επηρεάζονται από προηγούμενες καταστάσεις του συστήματος. Οι αλλαγές των εξόδων γίνονται σε διατεταγμένα βήματα, με τη βοήθεια μια ακολουθίας παλμών, που ελέγχει πότε γίνονται οι αλλαγές των καταστάσεων.

Ένα ψηφιακό σύστημα περιλαμβάνει και τα δύο είδη κυκλωμάτων, δηλαδή είναι ένας συνδυασμός από συνδυαστικά και ακουθιακά κυκλώματα. Η περιγραφή και προσομοίωση ενός κυκλώματος σε VHDL γίνεται με δύο τύπους εντολών, τις *σύγχρονες* (concurrent) και τις *ακολουθιακές* (sequential). Ήδη, στα παραδείγματα των προηγούμενων παραγράφων εμφανίστηκε η χρήση και των δύο τύπων προτάσεων.

Οι αντιστοιχήσεις τιμών που δεν γίνονται με άμεσο τρόπο, αλλά συγχρονίζονται με μεταβάσεις άλλων σημάτων, ενωματώνονται μέσα σε μια δομή «διεργασίας» (`PROCESS`). Μια διεργασία συνοδεύεται από μια «λίστα ευαισθησίας», όπου περιλαμβάνονται τα σήματα που οι μεταβάσεις τους σκανδαλίζουν την εκτέλεση της διεργασίας. Αν δεν υπάρξει μετάβαση, η διεργασία δεν εκτελείται. Ο κώδικας που περιέχεται στην `PROCESS` εκτελείται σειριακά, αλλά τα τελικά αποτελέσματα εμφανίζονται στα σήματα με σύγχρονο τρόπο, μόνον μετά το πέρας της εκτέλεσης της διεργασίας. Επειδή η `PROCESS` συνολικά είναι σύγχρονη, μπορεί να χρησιμοποιηθεί για την περιγραφή και συνδυαστικών και ακολουθιακών κυκλωμάτων. Ωστόσο, χωρίς `PROCESS`, δεν μπορούμε να περιγράψουμε σύνθετα ακολουθιακά κυκλώματα, όπως απαριθμητές, καταχωρητές κλπ. Εκτός από τον κώδικα της διεργασίας, με ακολουθιακό τρόπο εκτελείται και ο κώδικας που ανήκει στα λεγόμενα *υποπρογράμματα*. Υποπρογράμματα θεωρούνται οι συναρτήσεις (`FUNCTION`) και οι διαδικασίες (`PROCEDURE`).

Κώδικας που βρίσκεται έξω από διεργασίες, συναρτήσεις και διαδικασίες θεωρείται καθαρά σύγχρονος και εκτελείται άμεσα, ανεξάρτητα από τη θέση της εντολής μέσα στον κώδικα και χωρίς προϋποθέσεις μετάβασης κάποιων σημάτων. Σε κάθε κύκλο *προσομοίωσης* ο προσομοιωτής επιστρέφει σε όλες τις γραμμές του κώδικα και μεταδίδει τις αλλαγές, ενημερώνοντας τα αποτελέσματα, μέχρι να σταθεροποιηθούν τα σήματα. Το ίδιο επαναλαμβάνεται σε κάθε κύκλο προσομοίωσης. Κατά τη *σύνθεση* του σύγχρονου κώδικα, ο compiler συνθέτει συνδυαστικά κυκλώματα που ανανεώνουν άμεσα τις εξόδους, ως αποτέλεσμα του συνδυασμού των εισόδων.



S	F
0	A
1	B

Σχήμα 5.1 (α) Το κυκλωματικό σύμβολο του πολυπλέκτη 2:1, (β) ο πίνακας αληθείας και (γ) το κύκλωμα με πύλες.

Οι προτάσεις `SELECT`, `WHEN` και `GENERATE` θεωρούνται καθαρά σύγχρονες και τοποθετούνται αποκλειστικά έξω από διεργασίες και υποπρογράμματα. Αντίστοιχα,οι προτάσεις `IF`, `WAIT`, `LOOP` και `CASE` θεωρούνται καθαρά ακολουθιακές και τοποθετούνται μόνον σε διεργασίες και υποπρογράμματα. Στα επόμενα θα εξετάσουμε τις σύγχρονες προτάσεις και στη συνέχεια θα μελετηθούν οι ακολουθιακές.

5.2 Σύγχρονες αναθέσεις σημάτων-ένας απλός πολυπλέκτης

Όπως είναι γνωστό, ένας πολυπλέκτης δύο εισόδων και μιας εξόδου (2 προς 1) είναι συνδυαστικό κύκλωμα που εμφανίζει στην έξοδο τη μία από τις δύο εισόδους, με τη βοήθεια μιας γραμμής επιλογής. Το κυκλωματικό σύμβολο και ο πίνακας αληθείας φαίνονται στο σχήμα 5.1. (α) και (β). Στο σχήμα 5.1.(γ) φαίνεται το κύκλωμα του πολυπλέκτη με πύλες.

Ο πιο άμεσος τρόπος για τη σχεδίαση με VHDL ενός απλού κυκλώματος, όπως το παραπάνω, είναι η χρήση λογικών τελεστών. Αυτή η λύση παρουσιάζεται στο παρακάτω παράδειγμα.

Παράδειγμα 5.1 Να σχεδιαστεί σε VHDL ένας πολυπλέκτης 2:1, κάνοντας χρήση των λογικών τελεστών.

Λύση: Τα σήματα εισόδου/εξόδου ορίζονται ως τύπου `std_logic`, αφού μεταφέρουν πληροφορία ενός bit. Οι λογικοί τελεστές αναπαράγουν τη λογική δομή του κυκλώματος που παρουσιάζεται στο σχήμα 5.2 (γ). Οι αντιστοιχήσεις που φαίνονται στις γραμμές 12, 13 και 14 είναι σύγχρονες. Η ανάθεση των σημάτων γίνεται άμεσα σε κάθε κύκλο προσομοίωσης.

```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.all;
3 ----------------------------------------------------
4 ENTITY mux2_1 IS
5 PORT (A,B,s : IN std_logic;
6           F : OUT std_logic);
7 END mux2_1;
```

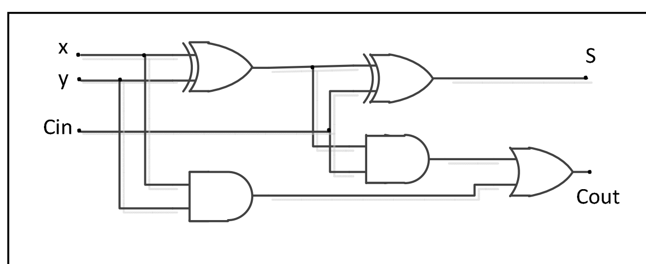
```
8 ------------------------------------------------------
9 ARCHITECTURE structural OF mux2_1 IS
10 SIGNAL m1, m2 : std_logic;
11 BEGIN
12      m1<= A AND NOT(s);
13      m2<= B AND s;
14      F<= m1 OR m2;
15 END structural;
```

Κώδικας 5.1 Σχεδίαση πολυπλέκτη 2:1 με λογικούς τελεστές

5.3 Ο πλήρης αθροιστής (full adder)

Ο πλήρης αθροιστής (Full Adder) είναι ένα συνδυαστικό κύκλωμα που εκτελεί την πρόσθεση δύο δυαδικών ψηφίων λαμβάνοντας υπόψη και την ύπαρξη κρατούμενου προηγούμενης τάξης. Ο πίνακας αληθείας του πλήρη αθροιστή φαίνεται στον σχήμα 5.2, μαζί με το κυκλωματικό διάγραμμα. Στον πίνακα αυτό με Cin συμβολίζεται το κρατούμενο εισόδου και με Cout συμβολίζεται το κρατούμενο εξόδου.
Όπως σχεδιάστηκε ο πολυπλέκτης, στην προηγούμενη παράγραφο, έτσι και ο πλήρης αθροιστής μπορεί να περιγραφεί με βάση την κυκλωματική δομή, με τη βοήθεια λογικών τελεστών που αναπαράγουν το δίκτυο των πυλών. Το βασικό αυτό κύκλωμα, όπως θα δούμε, μπορεί στη συνέχεια να χρησιμοποιηθεί για να δημιουργηθούν μεγαλύτερα κυκλώματα, που επιτελούν την πράξη της πρόσθεσης για αριθμούς πολλών bits. Ο κώδικας που περιγράφει το κύκλωμα είναι ο 5.2.

x	y	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



Σχήμα 5.2 Πίνακας αληθείας του πλήρους αθροιστή και κυκλωματικό διάγραμμα

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------
ENTITY fulladder IS
PORT(Cin,x,y :IN STD_LOGIC;
      s, Cout :OUT STD_LOGIC);
END fulladder;
-----------------------------
ARCHITECTURE structural OF fulladder IS
BEGIN
      s<=x XOR y XOR Cin;
      Cout<=(x AND y) OR (Cin AND x) OR (Cin AND y);
END Structural;
```

Κώδικας 5.2 Κώδικας για τη σχεδίαση του πλήρη αθροιστή αριθμών 1-bit.

5.3 Η εντολή SELECT

Μια βασική σύγχρονη εντολή είναι η SELECT, που συντάσσεται ως εξής:

```
      WITH αναγνωριστικό SELECT
      Έκφραση_ανάθεσης_τιμής WHEN τιμή
               Ανάθεση_τιμής WHEN τιμή
               Ανάθεση_τιμής WHEN τιμή
                ....;
```

Το «αναγνωριστικό» μπορεί να είναι το όνομα ενός σήματος. Επειδή η εντολή SELECT απαιτεί να καλύπτονται όλες οι δυνατές τιμές που λαμβάνει το «αναγνωριστικό», συχνά η παραπάνω δομή κώδικα κλείνει με την έκφραση WHEN OTHERS; στην οποία περιλαμβάνονται όλες οι δυνατές τιμές του αναγνωριστικού που δεν έχουν αναφερθεί παραπάνω. Η εντολή SELECT προφανώς υλοποιεί κυκλώματα με εισόδους επιλογής, όπως οι πολυπλέκτες, οι αποκωδικοποιητές και οι κωδικοποιητές.

Παράδειγμα 5.2 Να σχεδιαστεί ο πολυπλέκτης του σχήματος 5.1 με την εντολή SELECT.

Λύση: Το πρόγραμμα του πολυπλέκτη παρουσιάζεται στον κώδικα 5.3. Όταν η SELECT χρησιμοποιείται για την υλοποίηση πολυπλέκτη, τότε το «αναγνωριστικό» στη δομή SELECT είναι το σήμα επιλογής.

```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.all;
3 ---------------------------------------------------
4 ENTITY mux2_1 IS
```

```
5 PORT (A,B,s : IN std_logic;
6           F : OUT std_logic);
7 END mux2_1;
8 ------------------------------------------------------
9 ARCHITECTURE structural OF mux2_1 IS
10 SIGNAL m1, m2 : std_logic;
11 BEGIN
12 WITH s SELECT
13      F<= A WHEN '0',
14          B WHEN OTHERS;
15 END structural;
```

Κώδικας 5.3 Απλός πολυπλέκτης 2:1 με την εντολή `SELECT`

Παράδειγμα 5.3 Να σχεδιαστεί πολυπλέκτης δύο καναλιών, όπου το κάθε κανάλι έχει εύρος 8-bit.

Λύση: Ουσιαστικά πρόκειται για μια επανάληψη του πολυπλέκτη του σχήματος 5.1 οκτώ φορές, με κοινό το σήμα επιλογής s. Τα κανάλια A, B και F, γίνονται A(7…0), B(7...0) και F(7...0).

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------------------------
ENTITY part2 IS
PORT (A, B : IN std_logic_vector (7 downto 0);
         s : IN std_logic;
         F : OUT std_logic_vector(7 downto 0));
END part2;
----------------------------------------------------------
ARCHITECTURE behaviour OF part2 IS
BEGIN
 WITH s SELECT
    F<= A WHEN '0',
        B WHEN OTHERS;
END behaviour;
```

Κώδικας 5.4 Πολυπλέκτης δύο καναλιών, με εύρος καναλιού 8-bit.

Παράδειγμα 5.4 Να σχεδιαστεί πολυπλέκτης τεσσάρων καναλιών, όπου το κάθε κανάλι έχει εύρος 8-bit.

Λύση: Η φιλοσοφία σχεδίασης είναι ίδια με τις παραπάνω περιπτώσεις, αλλά το σήμα επιλογής s έχει πλέον εύρος δυο bit και η δομή `SELECT` πρέπει να λάβει υπόψη όλες τις τιμές του. Ο κώδικας φαίνεται στη λίστα 5.5.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------------------------
ENTITY part2 IS
PORT (A, B, C, D : IN std_logic_vector (7 downto 0);
         s : IN std_logic_vector(1 downto 0);
         F : OUT std_logic_vector (7 DOWNTO 0));
END part2;
----------------------------------------------------------
ARCHITECTURE behaviour OF part2 IS
BEGIN
 WITH s SELECT
    F<= A WHEN "00",
        B WHEN "01",
        C WHEN "10",
        D WHEN  OTHERS;
END behaviour;
```

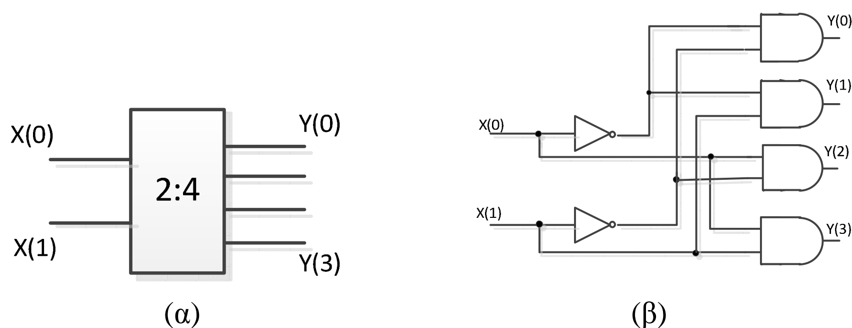
Κώδικας 5.5 Πολυπλέκτης τεσσάρων καναλιών, με εύρος καναλιού 8-bit.

5.3 Η εντολή `SELECT` σε κυκλώματα αποκωδικοποιητών

Στην παράγραφο αυτή παρουσιάζονται κυκλώματα αποκωδικοποιητών που σχεδιάζονται με χρήση της σύγχρονης εντολής `SELECT`. Ο δυαδικός αποκωδικοποιητής είναι ένα ψηφιακό κύκλωμα με n εισόδους και 2^n εξόδους, το οποίο για κάθε συνδυασμό των εισόδων ενεργοποιεί την έξοδο που αντιστοιχεί στο δυαδικό συνδυασμό της εισόδου. Έτσι, σε έναν αποκωδικοποιητή 2:4, οι δυνατοί συνδυασμοί των εισόδων είναι 00, 01, 10, 11 και οι έξοδοι απαριθμούνται 0, 1, 2, 3. Για τιμή της εισόδου 10 ενεργοποιείται η τρίτη στη σειρά έξοδος (2).

Παράδειγμα 5.6 Να σχεδιαστεί δυαδικός αποκωδικοποιητής 2:4, με την εντολή `SELECT`

Λύση: Στο σχήμα 5.3 παρουσιάζεται το διάγραμμα βαθμίδας ενός δυαδικού αποκωδικοποιητή 2:4 (α), το κυκλωματικό διάγραμμα (β) και ο πίνακας αληθείας (γ).
Οι είσοδοι και οι έξοδοι ομαδοποιούνται με τη μορφή πίνακα `std_logic_vector`, κάτι που διευκολύνει την περιγραφή της συμπεριφοράς με την εντολή SELECT. Ο κώδικας 5.6 παρουσιάζει το πρόγραμμα και το σχήμα 5.4 παρουσιάζει την προσομοίωση του δυαδικού αποκωδικοποιητή 2:4.



(α) (β)

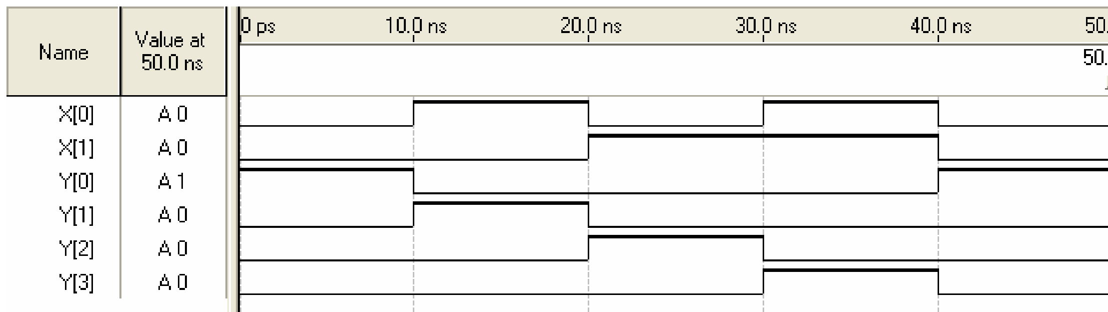
X_1	X_0	Y_0	Y_1	Y_2	Y_3
0	0	**1**	0	0	0
0	1	0	**1**	0	0
1	0	0	0	**1**	0
1	1	0	0	0	**1**

(γ)

Σχήμα 5.3 (α) Διάγραμμα βαθμίδας αποκωδικοποιητή 2:4, (β) κυκλωματικό διάγραμμα και (γ) πίνακας αληθείας.

```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.all;
3 ---------------------------------------------
4 ENTITY Decoder_2to4 IS
5 PORT(X :IN std_logic_vector (1 DOWNTO 0);
6      Y :OUT std_logic_vector (3 DOWNTO 0));
7 END Decoder_2to4;
8 ---------------------------------------------
9 ARCHITECTURE behaviour OF Decoder_2to4 IS
10 BEGIN
11 WITH X SELECT
12   Y<= "0001" WHEN "00",
13        "0010" WHEN "01",
14        "0100" WHEN "10",
15        "1000" WHEN  OTHERS;
16 END behaviour;
```

Κώδικας 5.6 Κώδικας για τη σχεδίαση δυαδικού αποκωδικοποιητή 2:4.

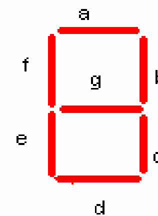


Σχήμα 5.4 Προσομοίωση του δυαδικού αποκωδικοποιητή 2:4.

5.4 Κωδικοποιητής BCD σε απεικόνιση επτά τομέων (SSD)

Ένας αποκωδικοποιητής κώδικα BCD σε ενδείκτη επτά τομέων (seven segment display ή SSD), μετατρέπει ένα δεκαδικό ψηφίο σε σήματα που οδηγούν τις διόδους φωτοεκπομπής (LEDs) του ενδείκτη. Οι δίοδοι αυτοί συμβολίζονται με τα γράμματα a ως g. O μετατροπέας αυτός αντιστοιχίζει κάθε συνδυασμό των εισόδων στον αντίστοιχο αριθμό του δεκαδικού συστήματος. Δηλαδή, θέτει σε λειτουργία τα απαραίτητα LEDs ώστε να σχηματιστεί ο αριθμός. Ο πίνακας αληθείας του μετατροπέα καθώς και ο ενδείκτης επτά τομέων φαίνονται στο σχήμα 5.5 που ακολουθεί. Εδώ, η απεικόνιση SSD είναι κοινής ανόδου, ώστε κάθε LED ανάβει με '0' στην κάθοδο.

X_3	X_2	X_1	X_0	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1



Σχήμα 5.5 Πίνακας αληθείας της απεικόνισης επτά τομέων και αρίθμηση των διόδων

Η περιγραφή του κυκλώματος του μετατροπέα BCD to SSD γίνεται με την ανάλυση της συμπεριφοράς του. Η περιγραφή του κυκλώματος με βάση την κυκλωματική δομή του είναι πολύπλοκη και θα μας υποχρέωνε στη συγγραφή πολλών γραμμών κώδικα, με μεγάλη πιθανότητα λάθους. Η δυνατότητα περιγραφής του κυκλώματος με βάση τη συμπεριφορά, έστω κι αν δεν γνωρίζουμε την εσωτερική δομή, μας διευκολύνει και αποτελεί σημαντικό πλεονέκτημα της γλώσσας VHDL. Στον κώδικα 5.7 που ακολουθεί, ο μετατροπέας αποτελείται από μία είσοδο `c` η οποία έχει μέγεθος 4 bits, και μία έξοδο `ex1` μεγέθους 7 bits. Το κάθε bit της εξόδου αντιστοιχεί σε ένα led του ενδείκτη επτά τομέων. Ο μετατροπέας αυτός μπορεί να απεικονίσει

τιμές από το 0 ως το 9, αφού θέλουμε την έξοδο σε δεκαδική μορφή. Στην είσοδο η τιμές που δίνουμε είναι από 0000 ως 1001 για να πάρουμε τα επιθυμητά αποτελέσματα ενώ για οποιαδήποτε άλλη τιμή στην είσοδο, στην απεικόνιση επτά τομέων παίρνουμε μια παύλα (-).

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
------------------------------------------------
ENTITY seg_7 IS
PORT(c  : IN std_logic_vector (3 DOWNTO 0);
     ex1: OUT std_logic_vector(6 DOWNTO 0));
END seg_7;
------------------------------------------------
ARCHITECTURE behaviour OF seg_7 IS
BEGIN
   WITH c SELECT
     ex1<= "1111110" WHEN "0000",
           "0000110" WHEN "0001",
           "1101101" WHEN "0010",
           "1111001" WHEN "0011",
           "0110011" WHEN "0100",
           "1011011" WHEN "0101",
           "1011111" WHEN "0110",
           "1110000" WHEN "0111",
           "1111111" WHEN "1000",
           "1111011" WHEN "1001",
           "0000001" WHEN OTHERS;
 END behaviour;
```

Κώδικας 5.7 Αποκωδικοποιητής BCD σε απεικόνιση επτά τομέων.

5.5 Η εντολή WHEN...ELSE

Η σύγχρονη εντολή WHEN...ELSE πραγματοποιεί ανάθεση τιμής σε σήμα υπό συνθήκη και μοιάζει με την ακολουθιακή εντολή IF. Η σύνταξη της εντολής είναι ως εξής:

```
Έκφραση_ανάθεσης WHEN συνθήκη ELSE
         τιμή_ανάθεσης WHEN συνθήκη ELSE
           ...
          τμή ανάθεσης;
```

Η παρακάτω ανάθεση υπό συνθήκη υλοποιεί μια δομή πολυπλέκτη

```
F<=x WHEN s= '0' ELSE y;
```

Παράδειγμα 5.7 Στο παράδειγμα αυτό σχεδιάζεται ένας πολυπλέκτης τεσσάρων καναλιών με τη βοήθεια της εντολής `WHEN…ELSE`.

Λύση: Στον παρακάτω κώδικα οι είσοδοι και η έξοδος ορίζονται με τύπο `std_logic_vector`. Τα κανάλια έχουν εύρος 4-bit. Η ίδια περιγραφή μπορεί να γίνει για οποιοδήποτε εύρος καναλιού, ή για εύρος ποου ρυθμίζεται με τη δήλωση `GENERIC`.

```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.all;
3 ----------------------------------------------------------
4 ENTITY mux IS
5 PORT (A, B, C, D : IN std_logic_vector (3 downto 0);
6          s : IN std_logic_vector(1 downto 0);
7          F : OUT std_logic_vector (3 DOWNTO 0));
8 END mux;
9 ----------------------------------------------------------
10 ARCHITECTURE behaviour OF mux IS
11 BEGIN
12     F<= A WHEN s="00" ELSE
13         B WHEN s="01" ELSE
14         C WHEN s="10" ELSE
15         D;
16 END behaviour;
```

Κώδικας 5.8 Σχεδίαση πολυπλέκτη με την εντολή `WHEN…ELSE`

5.6 Διαφορές ανάμεσα στις εντολές SELECT και WHEN

Υπάρχουν κάποιες διαφορές στη σύγχρονη `WHEN` και στη σύγχρονη εντολή `SELECT`. Στη `SELECT`, οι συνθήκες ανάθεσης είναι αμοιβαία αποκλειστικές, δηλαδή εκεί δεν μπορεί να γίνουν δύο διαφορετικές αναθέσεις με συνθήκες που συναληθεύουν. Στη `WHEN…ELSE`, όμως, οι συνθήκες έχουν σειρά προτεραιότητας τη σειρά με την οποία εμφανίζονται στον κώδικα και έτσι δεν είναι απαραίτητο η μία συνθήκη να αποκλείει την άλλη. Αυτό φαίνεται στο παράδειγμα 5.9. Στη γραμμή 15 του σχήματος 5.13 καλύπτονται όλες οι πιθανές περιπτώσεις τιμών που μπορεί να πάρει η γραμμή επιλογής s του πολυπλέκτη, με χρήση της εντολής `WHEN`. Πάντως, πρέπει να σημειωθεί ότι η σύνταξη της εντολής `WHEN` μπορεί να γίνει ακόμη κι αν δεν καλύπτονται όλες οι περιπτώσεις του πίνακα αληθείας. Αυτό είναι μια διαφορά από την εντολή `SELECT`, που απαιτεί να καλύπτονται όλες οι περιπτώσεις (σ' αυτό βοηθά η σύνταξη `WHEN OTHERS)`.

5.7 Σχεδίαση συγκριτών με την εντολή WHEN

Τυπικό παράδειγμα κώδικα με την εντολή WHEN είναι η περιγραφή συγκριτών.

Παράδειγμα 5.8 Να σχεδιαστεί συγκριτής δύο μη προσημασμένων αριθμών 4-bit. Οι τρεις έξοδοι `AgtB`, `AltB` και `AeqB` είναι `std_logic` και λαμβάνουν τιμή '1' όταν πληρούνται αντίστοιχα οι συνθήκες: A>B, A<B και A=B. Αλλιώς η έξοδος είναι 0.

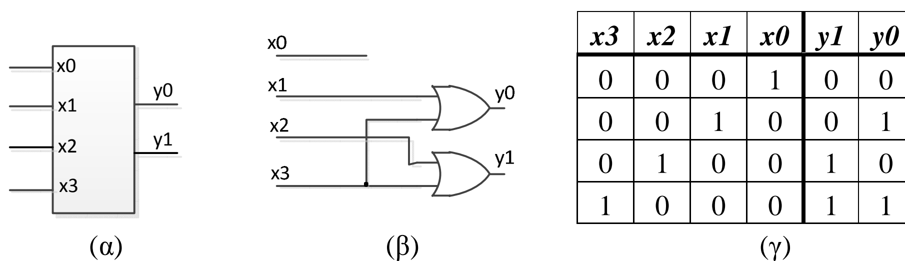
Λύση: Το κύκλωμα σχεδιάζεται με χρήση τριών εντολών WHEN…ELSE.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
------------------------------------------------
ENTITY compare IS
      PORT(A,B:IN STD_LOGIC_VECTOR(3 DOWNTO 0);
            AeqB, AgtB, AltB: OUT STD_LOGIC);
END compare;
------------------------------------------------
ARCHITECTURE behaviour OF compare IS
BEGIN
      AeqB<='1' WHEN A=B ELSE '0';
      AgtB<='1' WHEN A>B ELSE '0';
      AltB<='1' WHEN A<B ELSE '0';
END behaviour;
```

Κώδικας 5.9 Σχεδίαση συγκριτή μεγέθους αριθμών 4-bit.

5.8 Σχεδίαση κωδικοποιητών

Ο κωδικοποιητής είναι ένα κύκλωμα που επιτελεί την αντίστροφη λειτουργία από αυτή των κωδικοποιητών. Ένας δυαδικός κωδικοποιητής λαμβάνει είσοδο 2^n bits και την κωδικοποιεί σε n bits στην έξοδο του κυκλώματος. Αν η είσοδος έχει μία γραμμή ενεργή, τότε στην έξοδο κωδικοποιείται ο δυαδικός αριθμός που αντιστοιχεί στην ενεργή είσοδο. Στο σχήμα 5.6 παρουσιάζεται το διάγραμμα βαθμίδας για έναν κωδικοποιητή 4:2, το κύκλωμα με πύλες και ο αντίστοιχος πίνακας αληθείας. Η παραπάνω περιγραφή μπορεί να επεκταθεί για κωδικοποιητές 8:3. Η περιγραφή των κωδικοποιητών σε VHDL γίνεται με τις σύγχρονες εντολές που παρουσιάστηκαν στις προηγούμενες παραγράφους.



x3	*x2*	*x1*	*x0*	*y1*	*y0*
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Σχήμα 5.6 Δυαδικός κωδικοποιητής 4:2. (α) διάγραμμα βαθμίδων (β) κυκλωματικό διάγραμμα και (γ) πίνακας αληθείας.

x3	*x2*	*x1*	*x0*	*y1*	*y0*	*z*
0	0	0	0	x	x	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

Σχήμα 5.7 Πίνακας αληθείας κωδικοποιητή προτεραιότητας τεσσάρων εισόδων

Μια κατηγορία κωδικοποιητών είναι οι κωδικοποιητές προτεραιότητας. Σ' αυτούς λαμβάνεται υπόψη η προτεραιότητα των εισόδων. Έτσι, αν στο σήμα εισόδου δύο ή περισσότερες είσοδοι είναι μονάδα, τότε στην έξοδο κωδικοποιείται η είσοδος με την υψηλότερη προτεραιότητα. Σε έναν κωδικοποιητή προτεραιότητας 4:2 θεωρούμε ότι η είσοδος x0 έχει το μικρότερο βαθμό προτεραιότητας και η είσοδος x3 έχει τον μεγαλύτερο βαθμό προτεραιότητας. Έτσι, η κωδικοποίηση που εμφανίζεται στην έξοδο υπακούει στον πίνακα αληθείας του σχήματος 5.16. Εκεί φαίνεται ότι στον κωδικοποιητή προτεραιότητας υπάρχει μία ακόμη έξοδος, η *z*, η οποία γίνεται μηδέν όταν όλες οι είσοδοι είναι μηδέν, ενώ είναι μονάδα όταν τουλάχιστο μία είσοδος είναι μονάδα. Δηλαδή, η *z* λειτουργεί ως δείκτης εγκυρότητας των εισόδων. Αν όλες οι είσδοι είναι μηδέν, τότε η έξοδος δεν έχει νόημα, οπότε συμβολίζεται με την αδιάφορη κατάσταση (x). Οι αδιάφορες τιμές (x) στο αριστερό μέρος του πίνακα σημαίνουν προφανώς ότι αν η είσοδος μεγαλύτερης προτεραιότητας είναι μονάδα, τότε η τιμή της εξόδου y δεν αλλάζει, όποια τιμή κι αν έχουν οι είσοδοι μικρότερης προτεραιότητας. Έτσι, ο πίνακας αληθείας συμπτύσσεται, αφού δεν χρειάζεται να παραθέσουμε όλες τις περιπτώσεις τιμών που αντιστοιχούν στην αδιάφορη τιμή x.

Παράδειγμα 5.9 Να σχεδιαστεί κωδικοποιητής προτεραιότητας με χρήση της εντολής `WHEN…ELSE`

Λύση: Εδώ, γίνεται χρήση της προτεραιότητας που έχουν οι συνθήκες που ορίζονται από διαδοχικές εντολές `WHEN`. Οι συνθήκες που εμφανίζονται στις εντολές `WHEN` δεν αποκλείουν η μία την άλλη, όμως αν ισχύει η πρώτη, αυτό έχει προτεραιότητα έναντι της δεύτερης κ.ο.κ. Αυτός είναι ο λόγος που ο πίνακας αληθείας περιγράφεται με τις εντολές `WHEN` από κάτω προς τα πάνω, δηλαδή από την περίπτωση όπου η πιο σημαντική είσοδος είναι μονάδα. Ο παρακάτω κώδικας υλοποιεί τον κωδικοποιητή προτεραιότητας, χωρίς να συμπεριλαμβάνονται οι αδιάφορες καταστάσεις.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------
ENTITY priority_coder IS
      PORT(X : IN std_logic_vector(3 downto 0);
            Y : OUT std_logic_vector(1 downto 0);
```

```
          z : OUT std_logic);
END priority_coder;
------------------------------
ARCHITECTURE behaviour OF priority_coder IS
BEGIN
          Y<="11" WHEN X(3)='1' ELSE
             "10" WHEN X(2)='1' ELSE
             "01" WHEN X(1)='1' ELSE
             "00";
          z<='0' WHEN X="0000" ELSE '1';
END behaviour;
```

Κώδικας 5.10 Περιγραφή του κωδικοποιητή προτεραιότητας

5.9 Υλοποίηση αριθμητικής και λογικής μονάδας

Άσκηση 5.1 Να σχεδιάσετε σε VHDL μια αριθμητική και λογική μονάδα (ALU), κάνοντας χρήση των αριθμητικών και λογικών τελεστών. Η ALU θα δέχεται είσοδο δύο αριθμών A, B με εύρος 4-bit, είσοδο κρατουμένου Cin καθώς και έναν δυδικό κώδικα S για την επιλογή της πράξης (opcode), με εύρος 3-bit. Θα παράγει έξοδο F εύρους 4-bit και έξοδο κρατουμένου Cout. Θα υλοποιεί τις αριθμητικές και λογικές πράξεις που εμφανίζονται στον παρακάτω πίνακα:

S (opcode)	Λειτουργία	Περιγραφή
000	F=A+B+Cin	Άθροισμα με κρατούμενο
001	F=A-B-Cin	Αφαίρεση με κρατούμενο
010	F=A+1	Αύξησε το A
011	F=A-1	Μείωσε το A
100	F=A AND B	Λογικό AND
101	F=A OR B	Λογικό OR
110	F=A XOR B	Λογικό XOR
111	F=NOT A	Συμπλήρωμα του A

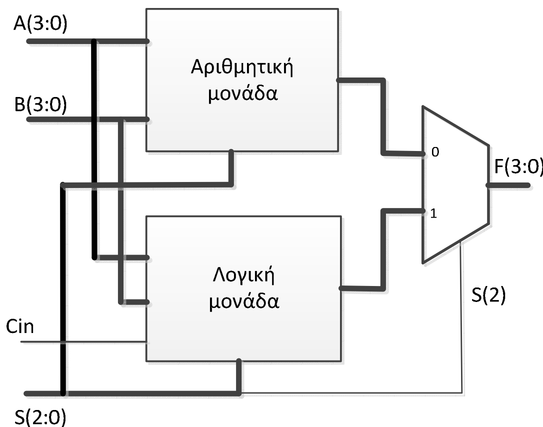
Πίνακας 5.1 Αριθμητικές και λογικές λειτουργίες που πρέπει να επιτελεί η ALU

Η ALU θα πρέπει επιπλέον να υπακούει στις παρακάτω προδιαγραφές

α. Οι αριθμητικές πράξεις πρέπει να είναι μη προσημασμένες

β. Τα σήματα εισόδου/εξόδου θα πρέπει να υπακούν στον τύπο `std_logic_vector`

Τέλος, ο κώδικας μπορεί να δομηθεί έτσι ώστε να υλοποιεί το παρακάτω διάγραμμα βαθμίδων (σχ. 5.8). Οι κανόνες για τον σχεδιασμό αριθμητικών και λογικών κυκλωμάτων έχουν αναπτυχθεί στις προηγούμενες παραγράφους.



Σχήμα Π.5.8 Διάγραμμα της ALU

5.10 Η εντολή FOR…GENERATE

Η σύγχρονη εντολή `GENERATE` δημιουργεί έναν βρόχο όπου επαναλαμβάνεται ένα τμήμα κώδικα, με τη βοήθεια ενός δείκτη. Ο κώδικας μπορεί να περιλαμβάνει απλές αναθέσεις σημάτων, αριθμητικές πράξεις ή και στιγμιότυπα κυκλωμάτων. Αποτελεί, λοιπόν, ένα δυναμικό τρόπο δημιουργίας μεγαλύτερων κυκλωμάτων από απλούστερα και βοηθά στη γραφή συμπυκνωμένου κώδικα. Η εντολή `GENERATE` εμφανίζεται με δύο τρόπους. Ο ένας είναι υπό συνθήκη (`IF…GENERATE`) και ο άλλος χωρίς συνθήκη (`FOR…GENERATE`). Θα εξεταστεί δεύτερος τρόπος, που είναι ο πλέον συνηθισμένος. Η διατύπωση της εντολής είναι ως εξής:

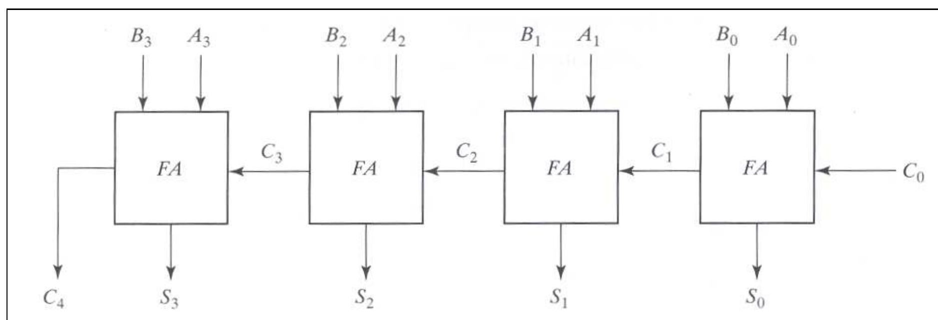
```
Ετικέτα: FOR αναγνωριστικό IN περιοχή τιμών GENERATE
            Τμήμα σύγχρονων εντολών
END GENERATE;
```

Στην παραπάνω διατύπωση, η ετικέτα είναι υποχρεωτική.
Για παράδειγμα, το παρακάτω τμήμα κώδικα αντιστρέφει τη σειρά των bits σε ένα σήμα x με εύρος οκτώ bits και αναθέτει την τιμή σε ένα σήμα y.

```
SIGNAL x,y,z : std_logic_vector(7 downto 0);
-----------------------------------------
My_label: FOR i IN 0 TO 7 GENERATE
             y(i)<=x(7-i);
END GENERATE;
```

Ο παρακάτω κώδικας αποδίδει λογικές τιμές στα bits ενός σήματος `z`, εκτελώντας λογικές πράξεις ανάμεσα στα bits των σημάτων `x` και `y`. Ας προσεχθεί η χρήση του χαρακτηριστικού (attribute) `x'RANGE` στη δήλωση της περιοχής τιμών του δείκτη.

```
Gen_0: FOR I IN x'RANGE GENERATE
          z(i)<= '1' WHEN x(i) OR y(i) = '1' ELSE '0';
END GENERATE;
```



Σχήμα 5.9 Κύκλωμα αθροιστή 4-bit με επανάληψη της δομής του πλήρη αθροιστή 1-bit και διάδοση του κρατούμενου.

Είναι σημαντικό να μην δημιουργούνται βρόχοι που αναθέτουν σε ένα σήμα παραπάνω από μία τιμές. Έτσι, ο παρακάτω κώδικας είναι λάθος, καθώς αναθέτει στο z πολλαπλές τιμές, ανάλογα με την ισχύ ή μη της συνθήκης:

```
Wrong_code: FOR i IN x'RANGE GENERATE
           z<= "10101010" WHEN x(i) OR y(i) = '1' ELSE "01010101";
END GENERATE;
```

Παράδειγμα 5.10 Να σχεδιαστεί ένας αθροιστής 4-bit, κάνοντας χρήση της οντότητας fulladder (Κώδικας 5.2, παράγραφος 5.3) και της εντολής FOR…GENERATE.

Λύση: Ένα ψηφιακό σύστημα αποτελείται από συνδεμένα μεταξύ τους στοιχεία ή υποκυκλώματα. Μια ανώτερη ιεραρχική οντότητα μπορεί να δημιουργεί ένα ή περισσότερα στιγμιότυπα από τέτοια στοιχεία και να τα συνδέει μεταξύ τους (βλέπε και σχ. 1.2, παράγραφος 1.3). Το κύκλωμα του πλήρη αθροιστή που προσθέτει αριθμούς πολλών bits είναι τυπικό παράδειγμα μιας τέτοιας σχεδίασης. Το σχήμα 5.9 παρουσιάζει το διάγραμμα του αθροιστή 4-bit, που αποτελείται από μια επανάληψη της βασικής δομής του πλήρη αθροιστή 1-bit, με κατάλληλη αντιστοίχηση των σημάτων εισόδου/εξόδου και μεταφορά του κρατουμένου ανάμεσα στις βαθμίδες. Στη VHDL η επανάληψη ενός κυκλώματος επιτυγχάνεται με μια FOR…GENERATE. Για κάθε τιμή του δείκτη i δημιουργείται ένα στιγμιότυπο (instance) του πλήρη αθροιστή 1-bit και επιτυγχάνεται η κατάλληλη αντιστοίχηση των σημάτων εισόδου/εξόδου. Ο αντίστοιχος κώδικας (οντότητα adder4) είναι ο 5.11.

Για την χρησιμοποίηση του κώδικα του πλήρη αθροιστή 1-bit πρέπει να δηλωθεί η οντότητά του ως στοιχείο (COMPONENT) μέσα στο σώμα της αρχιτεκτονικής. Αυτό γίνεται στην περιοχή των δηλώσεων (declarative part). Οι δηλώσεις των σημάτων εισόδου/εξόδου πρέπει να είναι ακριβώς αντίστοιχες με αυτές της αρχικής οντότητας fulladder. Σε κάθε δημιουργία στιγμιοτύπου fulladder με το βρόχο FOR…GENERATE περιλαμβάνεται μια πρόταση PORT MAP, που επιτελεί την αντιστοίχηση των σημάτων διασύνδεσης της οντότητας fulladder με τα σήματα της ανώτερης οντότητας adder4 (αντιστοίχηση θέσης). Η δομική σχεδίαση ιεραρχικών κυκλωμάτων με τη χρήση υποκυκλωμάτων θα εξεταστεί αναλυτικά στο κεφάλαιο 7.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
------------------------------------------------------
ENTITY adder4 IS
      PORT(Cin, x, y :IN std_logic_vector(3 downto 0);
            s,Cout :OUT std_logic_vector(3 downto 0));
END adder4;
----------------------------------------------------
ARCHITECTURE Structure OF adder4 IS
      SIGNAL c: std_logic_vector(0 to 4);

      COMPONENT fulladder
      PORT(Cin,x,y: IN std_logic;
            s, Cout :OUT std_logic);
      END COMPONENT;

BEGIN
      C(0)<=Cin
      Gen_label: FOR i IN 0 to 3 GENERATE
      Stage_i: fulladder PORT MAP(C(i),x(i),y(i),s(i),c(i+1));
      END GENERATE;
      Cout<=c(4);
END Structure;
```

Κώδικας 5.11 Κώδικας αθροιστή 4-bit με δήλωση δομικού στοιχείου (COMPONENT) και χρήση της εντολής FOR...GENERATE

Είναι προφανές ότι αθροιστές με 8 ή και περισσότερα bits μπορούν να σχεδιαστούν στη VHDL απλά αλλάζοντας το άνω όριο της περιοχής ορισμού του δείκτη i, καθώς και τον αριθμό των στοιχείων του σήματος c, που μεταφέρει τα κρατούμενα. Ο κώδικας μπορεί να γενικευτεί για οποιονδήποτε αριθμό bits με χρήση της δήλωσης GENERIC. Ας σημειωθεί ότι για να αναλυθεί χωρίς λάθη ο κώδικας του σχήματος 5.11 πρέπει η οντότητα του υποκυκλώματος fulladder να είναι αποθηκευμένη στον ίδιο φάκελο με την οντότητα adder4.

6 Ακολουθιακές προτάσεις

6.1 Χαρακτηριστικά ακολουθιακού κώδικα

Όπως αναφέρθηκε στην παράγραφο 5.1, οι ακολουθιακές προτάσεις στη σύνταξη του κώδικα της VHDL μπορούν να βρίσκονται μέσα σε διεργασίες (`PROCESS`) ή μέσα σε υποπρογράμματα (`FUNCTIONS` και `PROCEDURES`). Οι ακολουθιακές αυτές προτάσεις είναι οι `IF`, `WAIT`, `LOOP` και `CASE`. Αν οι προτάσεις αυτές γραφούν έξω από διεργασίες ή υποπρογράμματα, η ανάλυση του κώδικα θα παράγει σφάλμα και θα σταματήσει. Παρομοίως, θα παραχθεί σφάλμα αν γραφούν οι σύγχρονες προτάσεις `WHEN`, `SELECT`, `GENERATE` μέσα σε διεργασίες ή υποπρογράμματα.

Οι διεργασίες αποτελούν τον βασικό τρόπο με τον οποίο συντάσσεται ακολουθιακός κώδικας. Μπορούν να χρησιμοποιηθούν τόσο για την περιγραφή ακολουθιακών κυκλωμάτων όσο και για την περιγραφή συνδυαστικών κυκλωμάτων. Η δυνατότητα της περιγραφής συνδυαστικών κυκλωμάτων με ακολουθιακό κώδικα οφείλεται στο ότι η διεργασία σαν σύνολο περιγράφει μια σύγχρονη λειτουργία ανάθεσης τιμών, όταν συμβεί μετάβαση στα σήματα της λίστας ευαισθησίας.

Ένα άλλο χαρακτηριστικό των ακολουθιακών τμημάτων κώδικα είναι η δυνατότητα να ορίσουμε μέσα σ’ αυτά εκτός από σήματα (signals) και μεταβλητές (variables). Όπως είδαμε και στο κεφάλαιο 5, όπου μελετήσαμε σύγχρονο κώδικα, δεν μπορούμε να ορίσουμε μεταβλητές έξω από διεργασίες ή υποπρογράμματα. Εκεί, η μεταφορά της πληροφορίας γίνεται αποκλειστικά μέσω σημάτων ή σταθερών. Ωστόσο, οι μεταβλητές αποτελούν πολύ χρήσιμα αντικείμενα δεδομένων. Στην επόμενη παράγραφο θα δούμε συνοπτικά τις βασικές διαφορές στη χρήση σημάτων και μεταβλητών.

6.2 Σήματα και μεταβλητές

Όπως είχε αναλυθεί στο κεφάλαιο 3, τα σήματα αντιπροσωπεύουν τις διασυνδέσεις ανάμεσα σε σημεία του κυκλώματος. Μεταφέρουν πληροφορία από τον εξωτερικό κόσμο στο κύκλωμα ή από το ένα σημείο του κυκλώματος στο άλλο. Δηλώνονται κυρίως στο τμήμα δηλώσεων της οντότητας και της αρχιτεκτονικής. Επίσης, μπορούν να δηλωθούν στο τμήμα δηλώσεων των πακέτων (`PACKAGE`) που θα μελετηθούν σε άλλο κεφάλαιο. Τέλος, σημειώνουμε ότι προβλέπεται τμήμα δηλώσεων σημάτων και μέσα στην εντολή `GENERATE`, κάτι που δεν χρησιμοποιούμε σ’ αυτό το βιβλίο. Ένα σήμα δεν μπορεί να δηλωθεί μέσα σε μια διεργασία, ή άλλο ακολουθιακό τμήμα κώδικα. Όμως, ένα σήμα που έχει δηλωθεί στα παραπάνω προβλεπόμενα μέρη, είναι ορατό σε όλο τον κώδικα και μπορεί να χρησιμοποιηθεί και μέσα στις διεργασίες. Δηλαδή, οι δηλώσεις των σημάτων είναι καθολικές (global), όπως και η χρήση τους.

Όταν ένα σήμα χρησιμοποιείται μέσα σε ακολουθιακό τμήμα κώδικα, π.χ. μέσα σε μια διεργασία, τότε η τιμή του δεν ανανεώνεται αμέσως μόλις συμβεί μια ανάθεση (π.χ. `z<="0101"`), αλλά μόνον μετά το πέρας της διεργασίας. Αυτό σημαίνει ότι αν μέσα στο σώμα της διεργασίας συμβούν παραπάνω από μία αναθέσεις τιμών σε ένα σήμα, τότε στο τέλος το

σήμα αυτό λαμβάνει την τιμή που προκύπτει από την τελευταία ανάθεση. Γενικά, ένα σήμα μπορεί να λαμβάνει τιμή μία φορά μέσα στον κώδικα.

Μία μεταβλητή μπορεί να δηλώνεται και να χρησιμοποιείται μόνον μέσα σε διεργασία ή υποπρόγραμμα. Υπάρχει μια απόκλιση από αυτόν τον κανόνα, που αφορά στις κοινές μεταβλητές (shared variables), αλλά δεν θα ασχοληθούμε με αυτή την περίπτωση. Έτσι, οι μεταβλητές είναι τόσο ως προς τη δήλωσή τους όσο και ως προς τη χρήση τους «τοπικές» (local).

Η περιγραφή της δήλωσης μιας μεταβλητής δόθηκε στην παράγραφο 3.1.1 και 3.1.2. Η ανάθεση τιμής σε μια μεταβλητή γίνεται με το σύμβολο := . Παράδειγμα δήλωσης και ανάθεσης τιμής σε μεταβλητή δίνεται παρακάτω.

```
VARIABLE q : INTEGER
q:=q+1;
```

Μια μεταβλητή ανανεώνει την τιμή της άμεσα με την ανάθεση, έτσι ώστε μπορεί να χρησιμοποιηθεί με τη νέα τιμή στην επόμενη γραμμή του κώδικα. Με τον τρόπο αυτό, μια μεταβλητή μπορεί να αλλάζει τιμή σωρρευτικά, όπως στην ανάθεση παραπάνω, κάτι που δεν μπορεί να γίνει με τα σήματα, αφού αυτά λαμβάνουν μία τιμή σ' όλον τον κώδικα.

Όταν ένα σήμα ή μια μεταβλητή αποκτά τιμή κατά τη μετάβαση ενός άλλου σήματος, τότε το εργαλείο σύνθεσης συμπεραίνει ότι το σήμα είναι είσοδος σε ένα Flip-Flop.

6.3 Διεργασίες (PROCESS)

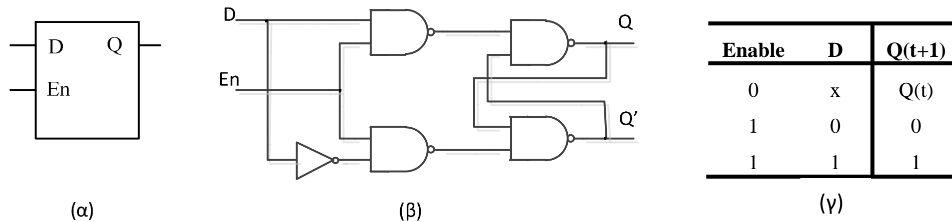
Το πιο κοινό τμήμα ακολουθιακού κώδικα είναι η διεργασία. Η δήλωση μιας διεργασίας γίνεται όπως παρακάτω:

```
PROCESS [(λίστα ευαισθησίας)] [IS]
[τμήμα δηλώσεων]
BEGIN
Ακολουθιακές προτάσεις
END PROCESS;
```

Μια διεργασία συνοδεύεται τις περισσότερες φορές από μια λίστα ευαισθησίας. Αυτή, περιλαμβάνει τα σήματα, που η αλλαγή της κατάστασής τους έχει ως αποτέλεσμα την εκτέλεση της διεργασίας. Πολύ συνηθισμένο σήμα στη λίστα ευαισθησίας είναι το clock. Η μόνη περίπτωση που η διεργασία δεν έχει λίστα ευαισθησίας είναι όταν περιλαμβάνει μια εντολή `WAIT`. Τότε, το κριτήριο για την εκτέλεση της διεργασίας είναι να εκπληρώνεται η συνθήκη της `WAIT`.

Στο τμήμα δηλώσεων της διεργασίας μπορεί να περιλαμβάνεται σχεδόν οτιδήποτε, εκτός από σήματα. Εκεί, δηλώνονται σταθερές, μεταβλητές, τύποι δεδομένων, υποπρογράμματα κλπ.

Όταν αλλάξει η τιμή ενός σήματος από αυτά που περιλαμβάνονται στη λίστα ευαισθησίας, αρχίζει η εκτέλεση της διεργασίας. Οι ακολουθιακές εντολές εκτελούνται σειριακά, η μία μετά την άλλη. Στο τέλος της διεργασίας γίνονται οι αναθέσεις των τιμών στα σήματα. Όλη η διεργασία από τη σκοπιά του χρήστη εκτελείται σαν μια σύγχρονη εντολή.



Enable	D	Q(t+1)
0	x	Q(t)
1	0	0
1	1	1

Σχήμα 6.1 Μάνδαλο τύπου D: (α) Διάγραμμα βαθμίδας, (β) κυκλωματικό διάγραμμα, (γ) πίνακας αληθείας.

6.4 Η εντολή `IF`

Η εντολή Αν είναι εντολή διακλάδωσης υπό συνθήκη. Μπορεί να γραφεί μόνον μέσα σε ακολουθιακό τμήμα κώδικα (συνήθως διεργασία-`PROCESS`). Ελέγχει αν μια συνθήκη είναι αληθής, οπότε εκτελεί το μέρος του κώδικα που ακολουθεί, αλλιώς ελέγχει μια νέα σειρά συνθηκών και εκτελεί το αντίστοιχο μέρος του κώδικα. Η πλήρης περιγραφή της `IF` έχει ως εξής:

```
IF συνθήκη THEN
      Προτάσεις ανάθεσης;
ELSIF συνθήκη  THEN
      Προτάσεις ανάθεσης
ELSE
      Προτάσεις ανάθεσης;
END IF;
```

Σε περίπλοκες περιπτώσεις μπορεί να υπάρχουν πολλαπλές `ELSIF`. Η `IF` μπορεί να διατυπωθεί και απλά:

```
IF συνθήκη THEN
      προτάσεις ανάθεσης;
END IF;
```

Παράδειγμα 6.1 Το μάνδαλο τύπου D (D-latch) είναι το απλούστερο κύτταρο αποθήκευσης της τιμής ενός bit. Έχει μια είσοδο D, μια έξοδο Q και δέχεται ένα σήμα ενεργοποίησης En. Όταν En=1, τότε ό,τι υπάρχει στην είσοδο οδηγείται στην έξοδο, ενώ όταν En=0, τότε η έξοδος παραμένει «κλειδωμένη» στην προηγούμενη κατάσταση. Δηλαδή, το μάνδαλο διατηρεί τη μνήμη της προηγούμενης εξόδου. Στο σχήμα 6.1 παρουσιάζονται το διάγραμμα βαθμίδας του μάνδαλου το κυκλωματικό διάγραμμα και ο πίνακας αληθείας. Ο παρακάτω κώδικας περιγράφει το μάνδαλο τύπου D σε VHDL με τη βοήθεια της εντολής `IF`.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
--------------------------------------------------------
```

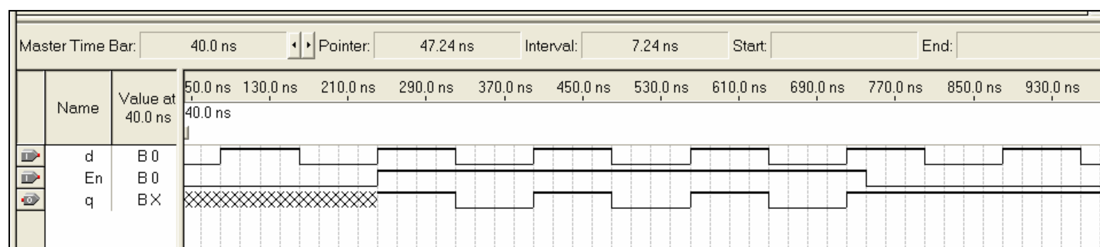
```
ENTITY dlatch IS
   PORT (d,En: IN std_logic;
            q: OUT std_logic);
END dlatch;
----------------------------------------------------------
ARCHITECTURE behaviour OF dlatch IS
BEGIN
   PROCESS (d, En)
   BEGIN
      IF En='1' THEN
       q<=d;
      END IF;
   END process;
END behaviour;
```

Κώδικας 6.1 Κώδικας που περιγράφει μάνδαλο τύπου D.

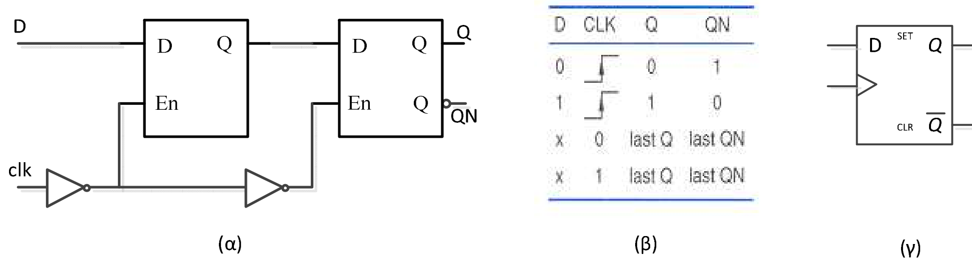
Στον παραπάνω κώδικα 6.1 παρατηρούμε ότι η εντολή IF διατυπώνεται στην απλούστερη εκδοχή της, χωρίς ELSIF ή ELSE. Ο μεταγλωττιστής θα μεταφράσει αυτή την κατάσταση ως εξής: Αν ισχύει η συνθήκη της IF τότε η έξοδος ανανεώνεται και λαμβάνει την τιμή της εισόδου d. Όταν δεν ισχύει η συνθήκη, τότε η έξοδος q θα παραμείνει αμετάβλητη. Άρα, αυτόματα, κατά τη σύνθεση, ο μεταγλωττιστής θα εισάγει ένα στοιχείο καταχώρησης για την υλοποίηση του παραπάνω κώδικα. Στο σχήμα 6.2 παρουσιάζεται η προσομοίωση του παραπάνω κώδικα. Όταν η είσοδος En είναι ίση με `'1'`, η έξοδος q παίρνει την τιμή που έχει η είσοδος d, ενώ όταν έχει την τιμή `'0'` (En=0) η έξοδος q διατηρεί την τελευταία τιμή που είχε, προτού η είσοδος ενεργοποίησης γίνει `'0'`.

6.5 Flip Flop τύπου D

Τα Flip Flop είναι κυκλώματα τα οποία αποτελούνται από μία είσοδο δεδομένων D, μία είσοδο ρολογιού clk και δύο εξόδους Q και QN. Τόσο τα Flip Flop όσο και οι μανδαλωτές που περιγράφηκαν προηγουμένως, αποτελούν τα πιο μικρά στοιχεία αποθήκευσης, χωρητικότητας 1 bit. Υπάρχουν δύο είδη Flip-Flops. Το πρώτο είδος είναι αυτά που η έξοδος τους επηρεάζεται από την είσοδο κατά το θετικό μέτωπο του παλμού του ρολογιού, δηλαδή η ανανέωση της εξόδου γίνεται την στιγμή της μετάβασης του ρολογιού από την κατάσταση Low στην



Σχήμα 6.2 Προσομοίωση του μάνδαλου τύπου D στο Quartus II.

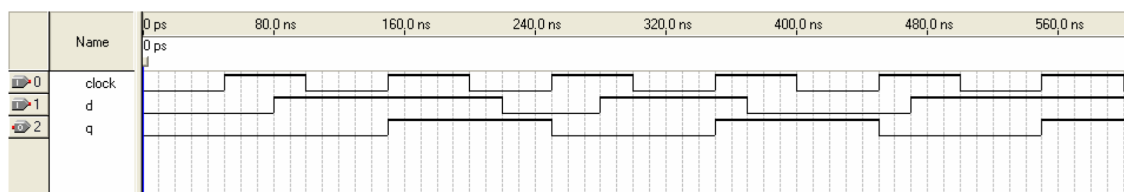


Σχήμα 6.3 α) Κύκλωμα D Flip Flop β) Πίνακας Αληθείας και γ) Σύμβολο D Flip Flop

κατάσταση High. Το δεύτερο είδος FF είναι αυτά που η έξοδος τους επηρεάζεται από την είσοδο κατά την αρνητική ακμή του ρολογιού, δηλαδή, κατά την μετάβαση του ρολογιού από την High κατάσταση στην Low. Στο σχήμα 6.3 βλέπουμε το κύκλωμα ενός D Flip Flop το οποίο διεγείρεται με θετικό μέτωπο, τον πίνακα αληθείας του και το κυκλωματικό σύμβολο. Όπως βλέπουμε, ένα D Flip Flop αποτελείται από δύο σύγχρονα μάνδαλα τύπου D (D-latches), από τα οποία το πρώτο λέγεται master και το δεύτερο slave.

Παράδειγμα 6.2 Να περιγραφεί σε VHDL η λειτουργία ενός D Flip-Flop

Λύση: Όπως αναφέραμε και πριν, to D Flip-Flop αποτελείται από δύο μάνδαλα τύπου D συνδεδεμένα στη σειρά. Ένας τρόπος περιγραφής του, λοιπόν, είναι να το περιγράψουμε δομικά, με χρήση στιγμιοτύπων, όπως κάναμε και στη περίπτωση του πλήρη αθροιστή. Όπως βλέπουμε και στον κώδικα 6.2, έχουμε περιγράψει το κύκλωμα πιο απλά, με βάση την συμπεριφορά του. Καταρχάς, ορίζουμε την οντότητα που θα περιέχει το D Flip Flop και της δίνουμε όνομα **dff_pos**. Η οντότητα περιέχει τις δύο εισόδους του κυκλώματος, `d` και `clock`, και την έξοδο `q`. Στη συνέχεια, χρησιμοποιούμε μία διεργασία (`process`). Στην λίστα ευαισθησίας της περιέχεται μόνο το σήμα του ρολογιού, αφού η έξοδος του FF αλλάζει μόνον όταν αλλάζει αυτό. Η πρώτη εντολή που συναντούμε μέσα στην διεργασία είναι μία `IF` με την εξής συνθήκη:
`IF clock'event and clock='1'THEN`.
Αυτή η γραμμή του κώδικα κάνει χρήση του χαρακτηριστικού (attribute) event και περιγράφει τη συνθήκη: «**αν το ρολόι μεταβαίνει στην κατάσταση '1' από την κατάσταση '0'**». Στην περίπτωση αυτή, δηλαδή κατά το θετικό μέτωπο ωρολογιακών παλμών, θα έχουμε στην έξοδο την τιμή που έχει εκείνη την στιγμή η είσοδος: **`then q<=d;`**. Στη συνέχεια του προγράμματος



Σχήμα 6.4 Προσομοίωση του D Flip Flop

```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.all;
3 ---------------------------------------------
4 ENTITY dff_pos IS
5   PORT (d, clock: IN    std_logic;
6               q : OUT std_logic);
7 END dff_pos;
8 ---------------------------------------------
9  ARCHITECTURE behaviour OF dff_pos IS
10 BEGIN
11  PROCESS (clock)
12    BEGIN
13         IF clock'event AND clock='1' THEN
14         q<=d;
15         END IF;
16  END process;
17 END behaviour;
```

Κώδικας 6.2 Κώδικας για την περιγραφή ενός D Flip-Flop που διεγείρεται με θετικό μέτωπο παλμού ρολογιού.

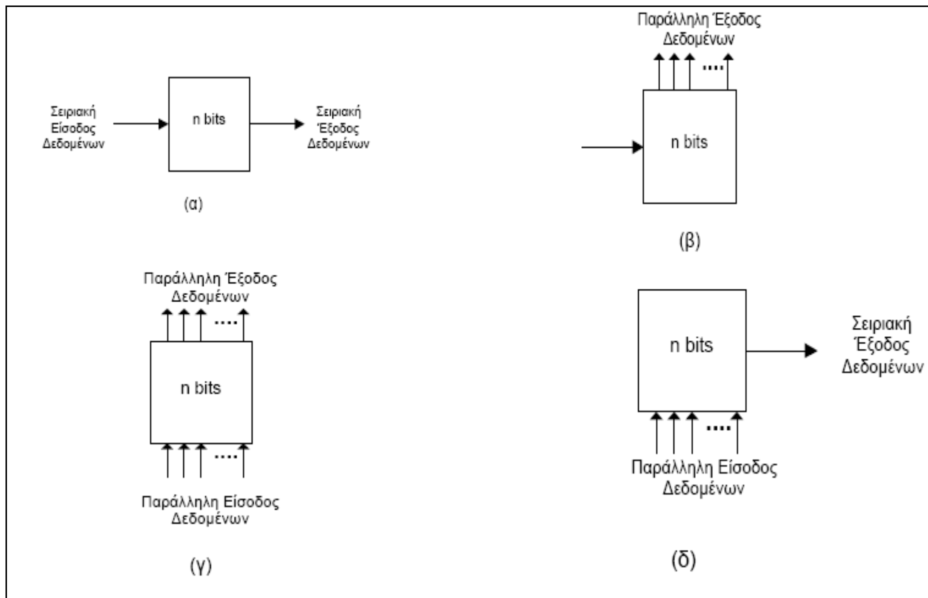
παρατηρούμε ότι κι εδώ παραλείπεται η χρήση της εντολής ELSE, αφού και στο D Flip Flop επιθυμούμε την διατήρηση της τιμής της εξόδου μέχρι το επόμενο θετικό μέτωπο παλμών, οπότε η έξοδος θα ανανεωθεί.
Στην προσομοίωση του σχήματος 6.4 φαίνεται η λειτουργία του D Flip-Flop. Όπως παρατηρούμε, όταν το ρολόι μεταβαίνει από την κατάσταση Low στην κατάσταση High, η έξοδος αλλάζει τιμή και παίρνει εκείνη της εισόδου κατά την συγκεκριμένη χρονική στιγμή. Σε όλες τις άλλες περιπτώσεις δεν συμβαίνει καμιά αλλαγή, δηλαδή η έξοδος κρατάει την τιμή που είχε πάρει κατά το τελευταίο θετικό μέτωπο του ωρολογιακού παλμού.

6.6 Καταχωρητές

Σ' αυτή την παράγραφο περιγράφεται η υλοποίηση καταχωρητών. Ως καταχωρητές θεωρούμε ένα σύνολο από Flip Flops, τα οποία είναι κατάλληλα συνδεδεμένα μεταξύ τους και δέχονται ένα κοινό ωρολογιακό σήμα. Σε κάθε Flip Flop μπορεί να αποθηκευτεί 1 bit πληροφορίας. Έτσι, σ' έναν καταχωρητή ο οποίος αποτελείται από n Flip Flops μπορούμε να αποθηκευτούν n bits πληροφορίας. Άρα, για την δημιουργία ενός καταχωρητή χωρητικότητας 8 bits θα πρέπει να χρησιμοποιηθούν οκτώ Flip-Flops.

Η εισαγωγή των δεδομένων σ' έναν καταχωρητή μπορεί να γίνει είτε σειριακά, είτε παράλληλα. Στην πρώτη περίπτωση σε κάθε ωρολογιακό παλμό εισάγεται ένα bit, ενώ στην δεύτερη περίπτωση εισάγονται και τα n bits με έναν ωρολογιακό παλμό. Το ίδιο ισχύει και για την εξαγωγή των δεδομένων. Τα δεδομένα μπορούν να εξαχθούν ή παράλληλα ή σειριακά.

Με βάση τους παραπάνω τρόπους εισαγωγής και εξαγωγής των δεδομένων, οι καταχωρητές χωρίζονται στις εξής τέσσερις κατηγορίες:



Σχήμα 6.5 Τύποι καταχωρητών: α) σειριακής εισόδου-σειριακής εξόδου, β) σειριακής εισόδου-παράλληλης εξόδου, γ) παράλληλης εισόδου-παράλληλης εξόδου, δ) παράλληλης εισόδου-σειριακής εξόδου.

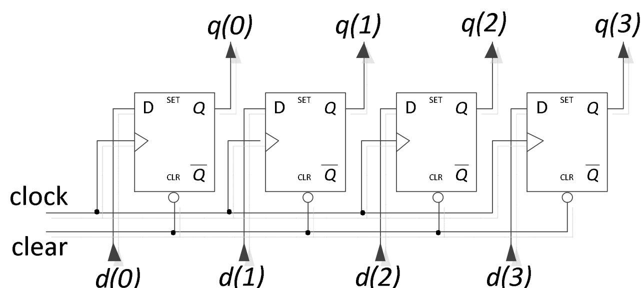
- Καταχωρητές σειριακής εισόδου-σειριακής εξόδου
- Καταχωρητές σειριακής εισόδου-παράλληλης εξόδου
- Καταχωρητές παράλληλης εισόδου-σειριακής εξόδου
- Καταχωρητές παράλληλης εισόδου-παράλληλης εξόδου

Οι τέσσερις τύποι των παραπάνω καταχωρητών φαίνονται στο σχήμα 6.5.

6.7 Καταχωρητές παράλληλης φόρτωσης

Παράδειγμα 6.3 Να σχεδιαστεί ένας καταχωρητής παράλληλης εισόδου - παράλληλης εξόδου με ασύγχρονη είσοδο clear. Ο καταχωρητής να σχεδιαστεί με δήλωση γενικής σταθερής (GENERIC) ώστε να μπορεί να επεκταθεί σε οποιοδήποτε εύρος bits.

Λύση: Ένας τέτοιος καταχωρητής χωρητικότητας 4 bits φαίνεται στο σχήμα 6.6. Η επέκταση του καταχωρητή σε οποιοδήποτε εύρος bits γίνεται μεταβάλλοντας τον αριθμό των Flip-Flops.
Στον κώδικα 6.3, που ακολουθεί, παρουσιάζεται η περιγραφή της λειτουργίας ενός καταχωρητή παράλληλης εισόδου και παράλληλης εξόδου.



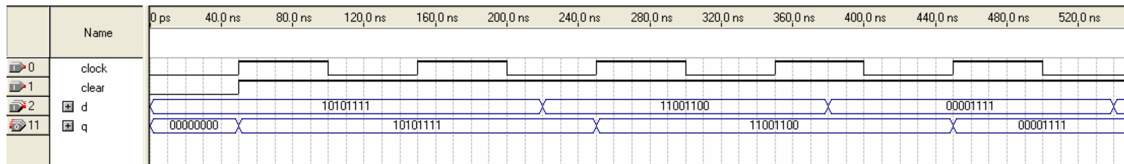
Σχήμα 6.6 Καταχωρητής παράλληλης εισόδου-παράλληλης εξόδου με εύρος 4-bit.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------------
ENTITY reg IS
      GENERIC(N : NATURAL :=8);
      PORT (d: IN std_logic_vector(N-1 DOWNTO 0);
            clock, clear: IN std_logic;
            q: OUT std_logic_vector(N-1 DOWNTO 0));
 END reg;
---------------------------------------------
ARCHITECTURE behaviour OF reg IS
BEGIN
      PROCESS (clear, clock)
      BEGIN
        IF clear='0' THEN
            q<=(OTHERS=>'0');
        ELSIF clock'event AND clock='1' THEN
            q<=d;
        END IF;
      END process;
END behaviour;
```

Κώδικας 6.3 Γενικός καταχωρητής παράλληλης φόρτωσης, με ασύγχρονη είσοδο clear.

Το όνομα της οντότητας του κυκλώματος είναι **reg**. Αποτελείται από τρεις εισόδους εκ των οποίων οι δύο (clock και clear) έχουν εύρος 1 bit και η τρίτη, αυτή των δεδομένων (d), έχει εύρος 8 bits. Η είσοδος και η έξοδος έχουν *N* bits, όπου η σταθερά *N* ορίζεται με γενικό τρόπο μέσω της δήλωσης GENERIC. Η τιμή της *N* εδώ είναι *N*=8. Η είσοδος clear λειτουργεί σαν μια ασύγχρονη είσοδος μηδενισμού.

Στο μέρος της αρχιτεκτονικής έχουμε μια διαδικασία (process), όπως σε όλα τα ακολουθιακά κυκλώματα. Στην λίστα ευαισθησίας περιέχονται οι είσοδοι clock και clear. Αν η είσοδος clear είναι ίση με το ‘0’ τότε ο καταχωρητής μηδενίζει την έξοδο, ανεξάρτητα από την τιμή της εισόδου. Αν η είσοδος clear είναι ‘1’ τότε ο καταχωρητής λειτουργεί κανονικά και η τιμή της



Σχήμα 6.7 Προσομοίωση Καταχωρητή 8 bits

εξόδου εξαρτάται πλέον από το ρολόι του κυκλώματος. Δηλαδή, κατά το θετικό μέτωπο ωρολογιακών παλμών θα λάβουμε στην έξοδο την τιμή που έχει εκείνη την στιγμή η είσοδος (`q<=d;`). Στη συνέχεια του κώδικα παρατηρούμε ότι παραλείπεται η χρήση της εντολής `ELSE`, αφού επιθυμούμε την διατήρηση της τιμής της εξόδου μέχρι το επόμενο θετικό μέτωπο παλμών, οπότε η έξοδος θα ανανεωθεί. Η λογική ανανέωσης της εξόδου σε έναν καταχωρητή είναι ίδια με αυτή που περιγράψαμε στο παράδειγμα 6.2 για το απλό Flip Flop.

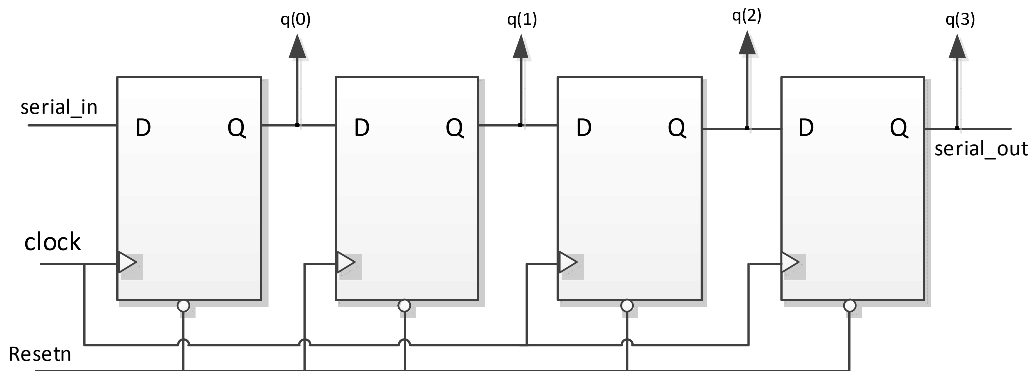
Στο Σχήμα 6.7 φαίνεται η προσομοίωση της λειτουργίας του καταχωρητή, οποία είναι σύμφωνη με τις θεωρητικές απαιτήσεις μας. Δηλαδή, όταν η είσοδος clear είναι ίση με ‘0’ στην έξοδο έχουμε πάντα ‘0’. Αντίθετα, όταν η είσοδος reset παίρνει την τιμή ‘1’ και έχουμε θετικό μέτωπο ωρολογιακών παλμών, η έξοδος παίρνει την τιμή που έχει η είσοδος την συγκεκριμένη χρονική στιγμή.

6.8 Καταχωρητές σειριακής ολίσθησης

Το δεύτερο είδος καταχωρητών που παρουσιάζονται στο σχήμα 6.5 είναι οι σειριακοί καταχωρητές. Αυτοί λαμβάνουν τα δεδομένα σειριακά και τα μεταθέτουν προ τα δεξιά ή προς τα αριστερά, σε κάθε παλμό ρολογιού. Τα δεδομένα μπορούν να λαμβάνονται σειριακά στην έξοδο μετά από μια καθυστέρηση Ν παλμών, οπότε οι κταχωρητές αναφέρονται ως σειριακής εισόδου, σειριακής εξόδου (Serial-In-Serial-Out ή SISO). Τέτοιοι είναι οι καταχωρητές που λειτουργούν ως γραμμές καθυστέρησης. Ο σκοπός τους είναι κυρίως να συγχρονίζουν τη διοχέτευση διαφορετικών δεδομένων σε μια μονάδα παράλληλης επεξεργασίας.

Εναλλακτικά, ένας καταχωρητής μπορεί να δέχεται σειριακά τα δεδομένα και να τα εμφανίζει παράλληλα σε μια έξοδο, μετά απο Ν παλμούς. Αυτοί οι καταχωρητές αναφέρονται ως σειριακής εισόδου-παράλληλης εξόδου (Serial-InParallel-Out ή SIPO). Τέτοιοι είναι οι καταχωρητές που λειτουργούν ως μετατροπείς σεριακών δεδομένων σε παράλληλα και βρίσκουν ευρύτατη εφαρμογή στη μετάδοση δεδομένων.

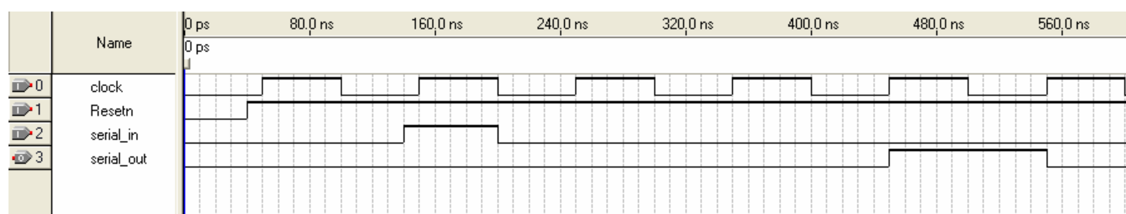
Παράδειγμα 6.4 Να σχεδιαστεί καταχωρητής ολίσθησης σειριακής εισόδου-σειριακής εξόδου, με τέσσερα Flip-Flops στη σειρά. Πως μπορούμε να επεκτείνουμε τον καταχωρητή ώστε να διαθέτει και έξοδο παράλληλης εμφάνισης των δεδομένων;



Σχήμα 6.8 Καταχωρητής ολίσθησης με τέσσερα Flip-Flops

Στο σχήμα 6.8 παρουσιάζεται ένας τυπικός καταχωρητής ολίσθησης. Κάθε bit στην είσοδο serial_in εμφανίζεται διαδοχικά στις εξόδους q(0), q(1), q(2), q(3), και συνεπώς θα παρουσιαστεί στην έξοδο serial_out μετά από τέσσερις παλμούς ρολογιού. Για τη σχεδίαση του καταχωρητή σε VHDL θα χρησιμοποιηθεί μια διεργασία με λίστα ευαισθησίας τα σήματα clock και Resetn. Ο μηδενισμός του καταχωρητή όταν Resetn=0 γίνεται στις γραμμές 16, 17.

Όταν Resetn=1, σε κάθε παλμό clock η είσοδος κάθε Flip-Flop οδηγείται στην έξοδο, όπως φαίνεται στις γραμμές 19 έως 22. Εδώ, υπενθυμίζεται ότι το σήμα q δεν αλλάζει τιμή μέσα στη διεργασία, αλλά μόνον μετά το πέρας της. Έτσι, κάθε επόμενη γραμμή δεν επηρεάζεται από τις μεταβολές των σημάτων που περιγράφονται στην προηγούμενη. Για παράδειγμα, το σήμα q(1) λαμβάνει την τιμή του q(0) στη γραμμή 20, το σήμα q(2) λαμβάνει την τιμή του q(1) στη γραμμή 21 και στη συνέχεια, το q(3) λαμβάνει την τιμή του q(2) στη γραμμή 22. Αυτό ΔΕΝ σημαίνει ότι το q(3) τελικά λαμβάνει την τιμή του q(0) μέσα στον ίδιο κύκλο του ρολογιού, διότι οι ανανεώσεις των σημάτων δεν εκτελούνται σε κάθε γραμμή κώδικα, παρά μόνον στο πέρας της διεργασίας. Άρα στην έξοδο κάθε Flip-Flop ολισθαίνει η τιμή της εισόδου, όπως καθορίστηκε κατά τον προηγούμενο κύκλο ρολογιού. Η έξοδος serial_out λαμβάνει την τιμή της εισόδου του τελευταίου Flip-Flop, δηλαδή της τιμή q(3).



Σχήμα 6.9 Αναφορά προσομοίωσης του καταχωρητή ολίσθησης με 4 FFs. Το λογικό 1 της εισόδου εμφανίζεται στην έξοδο μετά από τέσσερις παλμούς ρολογιού.

Η ιδιότητα των σημάτων να διατηρούν τις τιμές τους μέσα στη διεργασία και να τις ανανεώνουν μόνον κατά το πέρας, δεν ισχύει για τις μεταβλητές. Στην επόμενη παράγραφο θα συζητηθεί η χρήση των μεταβλητών, με αφορμή τον καταχωρητή ολίσθησης.

Ο παρακάτω κώδικας 6.4 περιγράφει τον καταχωρητή σειριακής εισόδου-σειριακής εξόδου με τέσσερα Flip-Flops. Προκειμένου να λάβουμε ως έξοδο και το σήμα q(0:3) αρκεί να το συμπεριλάβουμε στις διασυνδέσεις της δήλωσης PORT. Με αυτό τον τρόπο το κύκλωμα λειτουργεί και ως καταχωρητής σειριακής εισόδου-παράλληλης εξόδου.

```
1 -----------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 --------------------------------
5 ENTITY shift_reg4 IS
6         PORT(serial_in: IN std_logic;
7          clock, Resetn: IN std_logic;
8             serial_out: OUT std_logic);
9 END shift_reg4;
10---------------------------------
11 ARCHITECTURE behaviour OF shift_reg4 IS
12 SIGNAL q : std_logic_vector(0 to 3);
13 BEGIN
14   PROCESS(clock, Resetn)
15    BEGIN
16         IF Resetn='0' THEN
17              q<=(OTHERS=>'0');
18         ELSIF clock'EVENT AND clock='1' THEN
19              q(0)<=serial_in;
20              q(1)<=q(0);
21              q(2)<=q(1);
22              q(3)<=q(2);
23         END IF;
24    serial_out<=q(3);
25    END PROCESS;
26 END behaviour;
```

Κώδικας 6.4 Καραχωρητής ολίσθησης με τέσσερα Flip-Flops

Η προσομοίωση του παραπάνω κώδικα παρουσιάζεται στο σχήμα 6.9. Είναι χαρακτηριστικό ότι καθε bit που τοποθετείται στην είσοδο serial_in ολισθαίνει και εμφανίζεται στην έξοδο μετά από τέσσερις παλμούς ρολογιού.

6.9 Σήματα και μεταβλητές στον καταχωρητή ολίσθησης

Τι θα γίνει αν θελήσουμε να περιγράψουμε τον καταχωρητή ολίσθησης χρησιμοποιώντας μεταβλητές αντί για σήματα, στη διεργασία; Παρακάτω παραθέτουμε μια παραλλαγή της αχριτεκτονικής του κώδικα 6.4, όπου αντικαθίστανται τα σήματα με μεταβλητές. Εφιστάται η προσοχή του αναγνώστη, καθώς θα διαπιστωθεί ότι ο κώδικα αυτός είναι λανθασμένος.

```
1 ARCHITECTURE wrong_code OF shift_reg4 IS
2 BEGIN
3    PROCESS (clock, Resetn)
4     VARIABLE q : std_logic_vector(0 to 3);
5     BEGIN
6          IF Resetn='0' THEN
7               q:=(OTHERS=>'0');
8          ELSIF clock'EVENT AND clock='1' THEN
9               q(0):=serial_in;
10              q(1):=q(0);
11              q(2):=q(1);
12              q(3):=q(2);
13         END IF;
14    serial_out<=q(3);
15    END PROCESS;
16 END wrong_code;
```

Κώδικας 6.5 Λανθασμένος κώδικας για τον καταχωρητή ολίσθησης με χρήση μεταβλητών

Όπως εξηγήθηκε στην παράγραφο 6.2 μια μεταβλητή ανανεώνει την τιμή της άμεσα, μόλις εκτελεστεί με τη σειρά της η εντολή που αναθέτει τιμή στην μεταβλητή. Έτσι, η εκτέλεση των εντολών 9 έως 12 στον παραπάνω κώδικα θα έχει ως αποτέλεσμα ότι μετά την εκτέλεση και της εντολής 12 η μεταβλητή q(3) θα πάρει την τιμή serial_in, μέσα στον ίδιο κύκλο εκτέλεσης της διεργασίας. Ο compiler θα αντικαταστήσει τα τέσσερα Flip-Flops με ένα, καθώς αυτό αρκεί προκειμένου να ανεωθεί η έξοδος q(3) με την τιμή serial_in σε έναν κύκλο ρολογιού.

Ο σωστός τρόπος για να δημιουργήσουμε τον καταχωρητή ολίσθησης με τη βοήθεια μεταβλητής είναι να κάνουμε τις αναθέσεις από την έξοδο προς την είσοδο, αντιστρέφοντας τη σειρά των εντολών 9 έως 12. Ο κώδικας 6.6 παρουσιάζει τη σωστή σειρά ανάθεσης τιμών στη μεταβλητή q για τον καταχωρητή ολίσθησης. Κάνοντας τις αναθέσεις από την έξοδο q(3) προς την q(0) αποφεύγουμε να χρησιμοποιούμε σε κάθε επόμενη γραμμή μεταβλητή που η τιμή της αλλάζει στην αμέσως προηγούμενη. Έτσι, οι τιμές q που χρησιμοποιούμε σε κάθε γραμμή είναι αυτές που ανατέθηκαν στον προηγούμενο κύκλο ρολογιού, όπως είναι και το ζητούμενο. Το παράδειγμα αυτό αναδεικνύει την προσοχή που απαιτείται κατά τη χρήση μεταβλητών στις διεργασίες.

```
1  ARCHITECTURE wrong_code OF shift_reg4 IS
2  BEGIN
3    PROCESS(clock, Resetn)
```

```
4     VARIABLE q : std_logic_vector(0 to 3);
5     BEGIN
6           IF Resetn='0' THEN
7                 q:=(OTHERS=>'0');
8           ELSIF clock'EVENT AND clock='1' THEN
9                 q(3):=q(2);
10                q(2):=q(1);
11                q(1):=q(0);
12                q(0):=serial_in;
13          END IF;
14    serial_out<=q(3);
15    END PROCESS;
16 END wrong_code;
```

Κώδικας 6.6 Ορθός κώδικας για τον καταχωρητή ολίσθησης με χρήση μεταβλητών

6.10 Απαριθμητές

Σ' αυτή την ενότητα θα παρουσιάσουμε ένα κύκλωμα απαριθμητή. Απαριθμητές ονομάζονται τα κυκλώματα που μπορούν να αυξήσουν ή να μειώσουν την τιμή της εξόδου τους κατά ένα, όταν στην είσοδο ρολογιού δέχονται παλμό clock. Τα κυκλώματα των απαριθμητών μπορούν να χρησιμοποιηθούν α) για να μετρούν πόσες φορές εμφανίστηκε κάποιο γεγονός, για παράδειγμα πόσες φορές ένα περιφερειακό διακόπτει τον επεξεργαστή β) για την απαρίθμηση διαδοχικών εργασιών σε ένα σύστημα, όπως για παράδειγμα για την απαρίθμηση και τον έλεγχο των φάσεων εκτέλεσης εντολής σε μικροεπεξεργαστή γ) για την παραγωγή χρονικών καθυστερήσεων, όπως σε κυκλώματα χρονιστών.

Η κατασκευή τέτοιων κυκλωμάτων γίνεται συνήθως με τη χρήση καταχωρητών, και με Flip-Flops διαφόρων τύπων.

Παράδειγμα 6.4 Γενικός απαριθμητής με είσοδο παράλληλης φόρτωσης, είσοδο μηδενισμού και είσοδο enable.

Ο απαριθμητής που θα σχεδιάσουμε εδώ, λειτουργεί καταμετρώντας προς τα πάνω (up counter). Είναι σχεδιασμένος ως γενικός απαριθμητής mod N, με τη βοήθεια της δήλωσης `GENERIC`. Για N=4 ο απαριθμητής λειτουργεί με 4 bits. Διαθέτει είσοδο μηδενισμού (**`Resetn`**) και είσοδο ενεργοποίησης (**`en`**). Επίσης, διαθέτει είσοδο παράλληλης φόρτωσης **`A`** και ακροδέκτη για την ενεργοποίηση της παράλληλης φόρτωσης **`Ld`**.

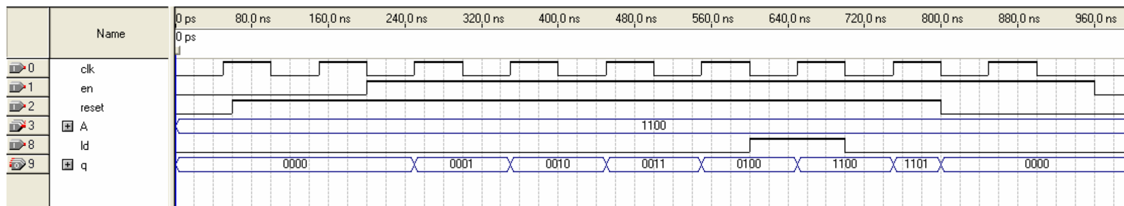
Όταν N=4, η έξοδος έχει εύρος 4 bits σημαίνει ότι μπορεί να πάρει μέχρι και την τιμή "1111", που στο δεκαδικό σύστημα είναι ο αριθμός 15. Δηλαδή καταμετρά συνολικά δεκαέξι καταστάσεις.

Οι είσοδοι **`resetn`** και **`en`** είναι οι είσοδοι μηδενισμού και ενεργοποίησης. Εάν η είσοδος **`resetn`** βρίσκεται σε λογική κατάσταση μηδέν, η έξοδος του κυκλώματος (**`q`**) θα μηδενίζεται.

```
1 LIBRARY  ieee;
2 USE ieee.std_logic_1164.all;
3 USE ieee.std_logic_unsigned.all;
4 ----------------------------------------------------------------
5 ENTITY counter IS
6     GENERIC(N : NATURAL :=8);
7     PORT(clk, ld, resetn, en : IN std_logic;
8           A: IN std_logic_vector (N-1 DOWNTO 0);
9           q: OUT std_logic_vector (N-1 DOWNTO 0));
10          END counter;
11----------------------------------------------------------------
12 ARCHITECTURE behaviour OF counter IS
13 SIGNAL m : std_logic_vector (3 DOWNTO 0);
14 BEGIN
15   PROCESS(clk, resetn)
16   BEGIN
17          IF resent='0' THEN
18               m<=(OTHERS=>'0');
19          ELSIF clk'event AND clk='1' THEN
20               IF en='1' THEN
21                    IF ld='1' THEN
22                      m<=A;
23                    ELSE
24                      m<= m+1;
25                    END IF;
26               ELSE
27                    m<=m;
28               END IF;
29          END IF;
30  END process;
  q<=m;
END behaviour;
```

Κώδικας 6.7 Απαριθμητής με είσοδο μηδενισμού, ενεργοποίησης και με δυνατότητα αρχικής φόρτωσης δεδομένων

Αν η είσοδος **resetn** είναι ίση με ένα και η είσοδος ενεργοποίησης **en** είναι και αυτή ένα, τότε η τιμή της εξόδου του μετρητή θα αυξάνεται κατά ένα σε κάθε θετικό ωρολογιακό μέτωπο, αλλιώς (αν **en**=0) η τιμή της εξόδου θα παραμένει αμετάβλητη. Αν η είσοδος **Ld** λάβει λογικό '1', στον επόμενο παλμό ρολογιού θα μεταφερθεί η είσοδος **A** στην έξοδο **q**, οπότε ο απαριθμητής θα συνεχίσει την απαρίθμηση από την τιμή A.



Σχήμα 6.10 Προσομοίωση Απαριθμητή 4 bits, με ασύγχρονη είσοδο `Resetn` και εισόδους Enable (`En`) και Load (`Ld`)

Οι παραπάνω περιπτώσεις των εισόδων περιγράφονται με εντολές `IF` μέσα σε μια διεργασία (`process`). Η λίστα ευαισθησίας περιέχει την είσοδο **`clock`** και την είσοδο μηδενισμού **`Resetn`**. Οποιαδήποτε αλλαγή στις τιμές αυτών των δύο εισόδων προκαλεί την αλλαγή της κατάστασης της εξόδου.

Στη προσομοίωση του σχήματος 6.10, φαίνεται η αύξηση της τιμής της εξόδου κατά ένα, σε κάθε θετικό ωρολογιακό παλμό όταν η είσοδος **resetn** ισούται με ένα και η είσοδος **en** επίσης ισούται με ένα. Επίσης, φαίνεται η φόρτωση της αρχικής τιμής `A="1100"` όταν η είσοδος `Ld='1'`.

6.11 Σήματα και μεταβλητές στους απαριθμητές

Παράδειγμα 6.5 Να σχεδιαστεί απαριθμητής με χρήση μεταβλητής, ο οποίος να απαριθμεί από το 0 έως και το 12 (mod 13). Στη συνέχεια να κάνετε τις ίδιες αναθέσεις και συγκρίσεις χρησιμοποιώντας σήματα αντί για μεταβλητές.Τι διαφορές θα προκύψουν;

Λύση: Αρχικά σχεδιάζουμε το κύκλωμα με χρήση μεταβλητής `m`, που τη χρησιμοποιούμε για την απαρίθμηση και την ορίζουμε ως ακέραιη στο τμήμα της διεργασίας. Σε κάθε θετικό μέτωπο ρολογιού η τιμή της μεταβλητής αυξάνει κατά ένα και αμέσως μετά συγκρίνεται με τον ακέραιο 13. Σε προηγούμενα παραδείγματα, όπως στους κώδικες 2.2 και 3.1, στην εντολή `IF` που περιγράφει τον απαριθμητή υπήρχε για λόγους πληρότητας και η πρόταση `ELSE`, που αναθέτει στο `m` την τελευταία κατάσταση (`m<=m;`), αν το συμβάν αντιστοιχεί σε αρνητική και όχι σε θετική μετάβαση του ρολογιού. Η `ELSE` μπορεί να παραληφθεί, καθώς ο compiler θα διατηρήσει αυτόματα την ίδια τιμή της `m` αν δεν εκτελεστεί η `IF`.

Αν `m=13`, τότε η μεταβλητή μηδενίζεται. Αυτό σημαίνει ότι ο απαριθμητής θα μετρήσει μέχρι το 12 και θα επιστρέψει στο μηδέν. Τέλος, η ανάθεση της μεταβλητής στο σήμα `q` πρέπει να γίνει μέσα στη διεργασία, καθώς η μεταβλητή `m` δεν είναι ορατή έξω από την `PROCESS`.

Υπενθυμίζεται ότι η τιμή μιας μεταβλητής ανανεώνεται αμέσως με την ανάθεση, ώστε η νέα τιμή είναι διαθέσιμη στην επόμενη γραμμή κώδικα. Αυτό σημαίνει ότι αν στη γραμμή 16 η εντολή `m:=m+1` αναθέσει στην `m` την τιμή 13, η τιμή αυτή είναι διαθέσιμη για την επόμενη σύγκριση, στη γραμμή 17. Έτσι, μόλις λάβει η μεταβλητή την τιμή 13, η έξοδος του απαριθμητή θα μηδενιστεί.

```
1 LIBRARY  ieee;
2 USE ieee.std_logic_1164.all;
3 USE ieee.std_logic_unsigned.all;
```

```
4 --------------------------------------------------------
5 ENTITY counter1 IS
6     PORT(clk :  IN std_logic;
7           q  :  OUT std_logic_vector (3 DOWNTO 0));
8 END counter1;
9 --------------------------------------------------------
10 ARCHITECTURE behaviour OF counter1 IS
11 BEGIN
12    PROCESS(clk)
13    VARIABLE m : INTEGER RANGE 0 TO 13;
14    BEGIN
15          IF clk 'event AND clk='1' THEN
16            m:= m+1;
17                IF (m=13)THEN
18                m:=0;
19                END IF;
20          END IF;
20    q<=m;
20    END process;
22 END behaviour;
```

Κώδικας 6.8 Απλός απαριθμητής mod 13, με χρήση μεταβλητής

Στη συνέχεια, γράφουμε τον αντίστοιχο κώδικα χρησιμοποιώντας για την απαρίθμηση ένα ακέραιο σήμα m, που το ορίζουμε κατά τα γνωστά στην περιοχή δηλώσεων της αρχιτεκτονικής. Κατά τα άλλα, ο κώδικας παραμένει ίδιος. Ένα σημείο προσοχής είναι ότι τώρα πλέον μπορούμε να κάνουμε την τελική ανάθεση της γραμμής 22 έξω από τη διεργασία, αφού τα σήματα είναι ορατά σε όλο τον κώδικα.

```
1 LIBRARY  ieee;
2 USE ieee.std_logic_1164.all;
3 USE ieee.std_logic_unsigned.all;
4 --------------------------------------------------------
5 ENTITY counter1 IS
6     PORT(clk :  IN std_logic;
7           q  :  OUT std_logic_vector (3 DOWNTO 0));
8 END counter1;
9 --------------------------------------------------------
10 ARCHITECTURE behaviour OF counter1 IS
11 SIGNAL m : INTEGER RANGE 0 TO 13;
12 BEGIN
13    PROCESS(clk)
14    BEGIN
15          IF clk 'event AND clk='1' THEN
```

```
16            m<= m+1;
17                IF (m=13)THEN
18                m<=0;
19                END IF;
20          END IF;
21    END process;
22    q<=m;
23 END behaviour;
```

Κώδικας 6.9 Χρήση σήματος για την απαρίθμηση. Ο απαριθμητής λειτουργεί ως mod 14.

Αν και η λογική του κώδικα δεν διαφέρει, ωστόσο κατά την εκτέλεση θα προκύψουν διαφορές. Στην πραγματικότητα, ο απαριθμητής του κώδικα 6.9 θα απαριθμεί μέχρι και το 13 πριν μηδενιστεί και όχι μέχρι και το 12. Ο λόγος είναι ότι το σήμα `m` δεν αλλάζει τιμή μέσα στη διεργασία, αλλά μόνον μετά το πέρας της. Έτσι, όταν η ανάθεση `m<=m+1;` αναθέσει την τιμή 13 στο σήμα (γραμμή 16), αυτό δεν θα αλλάξει αμέσως τιμή, οπότε η σύγκριση της γραμμής 17, που ακολουθεί, δεν θα αποδώσει τιμή TRUE. Για το λόγο αυτό το `m` δεν θα μηδενιστεί. Μόνον μετά το πέρας της διεργασίας το `m` θα λάβει την τιμή 13, οπότε ο απαριθμητής θα μηδενιστεί κατά τον επόμενο κύκλο ρολογιού (δηλαδή κατά τον δέκατο τέταρτο κύκλο). Έτσι, όταν για την απαρίθμηση χρησιμοποιούμε σήματα, θα πρέπει να προβλέψουμε ότι για να μηδενιστεί ο απαριθμητής στον δέκατο τρίτο παλμό η σύγκριση πρέπει να γίνει με το 12.

Το παραπάνω παράδειγμα δείχνει με ποιό τρόπο λειτουργούν τα σήματα μέσα σε διεργασίες. Όταν κάνουμε ανάθεση σε ένα σήμα `m` μέσα σε μια διεργασία, τότε «προγραμματίζουμε» μια μεταβολή στην τιμή του σήματος, η οποία θα έχει ισχύ για τη διεργασία την επόμενη φορά που θα εκτελεστεί η διεργασία. Αν χρησιμοποιούμε το ίδιο σήμα `m` σε σύγχρονες εντολές εκτός της διεργασίας, π.χ. σε μια `WHEN`, τότε το σήμα θα έχει εκτός διεργασίας την τιμή που του ανατέθηκε μέσα στη διεργασία, αμέσως μετά το πέρας της διεργασίας.

Η χρήση των μεταβλητών μέσα σε διεργασίες εξασφαλίζει ότι οι τιμές που αυτές αναλαμβάνουν μέσα σε μια διεργασία είναι διαθέσιμες αμέσως στην επόμενη εντολή. Γενικά, χρειάζεται πολλή προσοχή στα παραπάνω σημεία, είτε εργάζεται κανείς με σήματα είτε με μεταβλητές. Είδαμε ότι στην περίπτωση του ολισθητή η χρήση μεταβλητών μπορεί να οδηγήσει σε λάθος αποτέλεσμα, καθώς αυτές ανανεώνονται όλες μέσα στον ίδιο κύκλο εκτέλεσης της διεργασίας (κώδικας 6.5), ενώ προβλήματα μπορεί να προκύψουν και με τη χρήση των σημάτων, όπως στην περίπτωση του απαριθμητή (κώδικας 6.9).

6.12 Η εντολή `WAIT`

Πρόκειται για άλλη μια ακολουθιακή εντολή, που χρησιμοποιείται αποκλειστικά μέσα σε διεργασία ή υποπρογράμματα. Η εντολή αυτή αποτελεί έναν εναλλακτικό τρόπο για να καθοριστεί πότε μια διεργασία θα ανταποκριθεί σε αλλαγές των σημάτων. Όπως αναφέρθηκε, όταν υπάρχει η `WAIT` η διεργασία ***δεν*** περιλαμβάνει λίστα ευαισθησίας. Μπορεί να συνοδεύεται από μια συνθήκη (`wait until`), από λίστα σημάτων (`wait on`) ή από μια χρονική έκφραση (`wait for`). Οι τρεις τρόποι σύνταξης περιγράφονται παρακάτω:

```
[Ετικέτα:] WAIT UNTIL συνθήκη;
[Ετικέτα:] WAIT ON λίστα_ευαισθησίας;
[Ετικέτα:] WAIT FOR χρονική_έκφραση;
```

Η ετικέτα δεν είναι υποχρεωτική. Παραδείγματα χρήσης της `WAIT UNTIL` παρουσιάζονται παρακάτω.

```
SIGNAL clock, z : bit;
WAIT UNTIL not clock;
```

Με την παραπάνω εντολή η διεργασία βρίσκεται σε αναμονή και δεν εκτελείται, όσο το σήμα clock είναι `'1'`. Η εκτέλεση συνεχίζεται όταν clock= `'0'`. Ένα άλλο παράδειγμα είναι το εξής:

```
WAIT UNTIL clock'event AND clock = '1';
      z<= '1';
```

Η παραπάνω χρήση της `WAIT` είναι ισοδύναμη με την εντολή

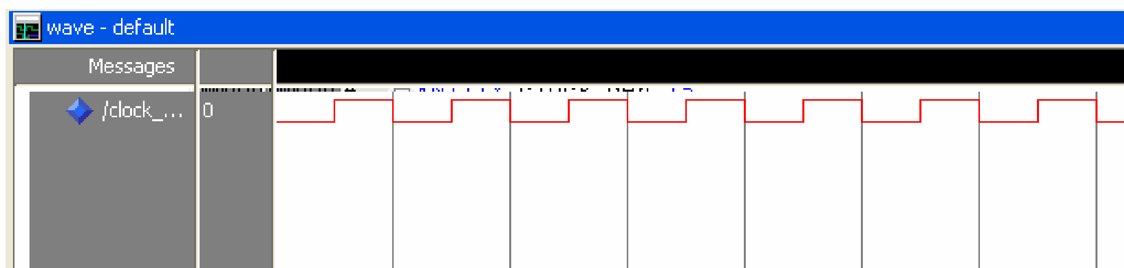
```
IF clock'event AND clock= '1' THEN
      z<= '1';
```

Τέλος, η εντολή `WAIT FOR` χρησιμοποιείται σε προσομοίωση κυκλωμάτων, όπου ιδιαίτερο ρόλο παίζει ο χρόνος.

Παράδειγμα 6.6 Η διεργασία που περιγράφεται παρακάτω υλοποιεί μια γεννήτρια ρολογιού, με περίοδο 20 ns. Ο κώδικας προσομοιώνεται στο λογισμικό προσομοίωσης ModelSim.

```
PROCESS
BEGIN
      Clock<= '1' after 10 ns, '0' after 20 ns;
      WAIT FOR 20 ns;
END PROCESS;
```

Η εντολή `after` συντάσσεται με σταθερές τύπου `TIME` ή άλλες χρονικές εκφράσεις και χρησιμοποιείται για τη δημιουργία προγραμμάτων προσομοίωσης (test benches). Το αποτέλεσμα της προσομοίωσης με το λογισμικό ModelSim φαίνεται στο σχήμα 6.11.



Σχήμα 6.11 Παραγωγή του σήματος ρολογιού με βάση τη διεργασία του παραδείγματος 6.6. Η προσομοίωση έγινε στο περιβάλλον ModelSim.

Παράδειγμα 6.7 Να γραφεί η διεργασία ενός D Flip-Flop με την εντολή `WAIT UNTIL`.

Λύση:
Η εντολή `WAIT UNTIL` συντάσσεται με συνθήκη:

```
PROCESS
    BEGIN
      WAIT UNTIL clock'event AND clock='1';
      q<=d;
END PROCESS;
```

Παράδειγμα 6.7 Να περιγράψετε τον πλήρη αθροιστή 1-bit χρησιμοποιώντας κώδικα συμπεριφοράς με διεργασία (`PROCESS`) και εντολή `WAIT ON`.

Λύση: Ο παρακάτω κώδικας υλοποιεί τον πλήρη αθροιστή με μια διεργασία. Χρησιμοποιεί ως λίστα ευαισθησίας τα σήματα της `WAIT ON`, ενώ δεν έχει λίστα ευαισθησίας στη διεργασία:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------
ENTITY fulladder IS
PORT(Cin,x,y :IN STD_LOGIC;
        s, Cout :OUT STD_LOGIC);
END fulladder;
-----------------------------
ARCHITECTURE adder OF fulladder IS
BEGIN
      PROCESS
        BEGIN
        s<=x XOR y XOR Cin;
        Cout<=(x AND y) OR (Cin AND x) OR (Cin AND y);
        WAIT ON x, y;
      END PROCESS;
END adder;
```

Κώδικας 6.10 Ο πλήρης αθροιστής με διεργασία και `WAIT ON`

6.13 Εντολή LOOP

Η εντολή `LOOP` δημιουργεί βρόχους επανάληψης, όπου υλοποιούνται πολλά στιγμιότυπα (instances) ενός κυκλώματος. Έτσι, για παράδειγμα, με μια `LOOP` μπορούμε να επαναλάβουμε ένα D Flip-Flop όσες φορές χρειάζεται για τη δημιουργία ενός καταχωρητή. Σε έναν απαριθμητή η Loop μπορεί να χρησιμοποιηθεί για την παραγωγή των διαδοχικών αυξήσεων της εξόδου, μέχρι να συμπληρωθεί ο απαραίτητος αριθμός καταστάσεων. Σε όλες τις περιπτώσεις, η `LOOP` πρέπει να βρίσκεται μέσα σε διεργασία ή υποπρόγραμμα. Η αντίστοιχη σύγχρονη εντολή είναι η `GENERATE`. Οι μορφές με τις οποίες συναντούμε τη `LOOP` είναι οι εξής:

Α. LOOP χωρίς συνθήκη (infinite loop-ατέρμων βρόχος):

```
[ετικέτα_βρόχου:] LOOP
ακολουθιακός κώδικας
END LOOP [ετικέτα_βρόχου];
```

Στο συμβατικό προγραμματισμό, ένας βρόχος δεν πρέπει να επαναλαμβάνεται στο άπειρο, χωρίς συνθήκη. Όμως, πολλά κυκλώματα λειτουργούν ακριβώς μ' αυτό τον τρόπο. Όπως θα δούμε, συχνά προβλέπεται και κάποια συνθήκη εξόδου από τον ατέρμονα βρόχο. Το παρακάτω παράδειγμα υλοποιεί έναν απαριθμητή mod 16 με έναν ατέρμονα βρόχο.

Παράδειγμα 6.8 Μοντελοποίηση απαριθμητή mod 16 με εντολή LOOP.

Ο παρακάτω κώδικας εισάγει έναν ατέρμονα βρόχο LOOP μέσα σε μια διεργασία, με σκοπό την αύξηση της εξόδου σε κάθε θετικό μέτωπο ρολογιού, ξεκινώντας από την τιμή μηδέν. Η επανάληψη διακόπτεται κάθε φορά στην αρχή, αναμένοντας την εκπλήρωση της συνθήκης WAIT. Στο θετικό μέτωπο του σήματος clk η LOOP συνεχίζει, οπότε ανανεώνει την τιμή της μεταβλητής m και την έξοδο q. Ας προσεχθεί ό μηδενισμός της m μόλις αυτή αυξηθεί και γίνει 16. Αυτό πετυχαίνεται με το υπόλοιπο της διαίρεσης (mod) με το 16, που γίνεται μηδέν μονον όταν m=16. Τέλος, η διεργασία προφανώς δεν έχει λίστα ευαισθησίας, αφού περιέχεται εντολή WAIT.

Κάθε φορά που αναστέλλεται η διεργασία εξαιτίας της εντολής WAIT, αποδίδονται και οι τιμές των σημάτων. Ο παρακάτω κώδικας μοντελοποιεί τη λειτουργία του απαριθμητή, αλλά δεν παρέχει κάποιο μηχανισμό για την έξοδο από τον ατέρμονα βρόχο.

```
------------------------------------
ENTITY loop_counter IS
      PORT(clk : IN bit;
              q : OUT natural);
END loop_counter;
------------------------------------
ARCHITECTURE behaviour OF loop_counter IS
BEGIN
      PROCESS
      VARIABLE m : natural :=0;
      BEGIN
        q<=m;
        LOOP
            wait until clk'event AND clk ='1';
            m:=(m+1) mod 16;
            q<=(m);
        END LOOP;
      END PROCESS;
END behaviour;
```

Κώδικας 6.11 Μοντελοποίηση απαριθμητή με ατέρομονα βρόχο (LOOP)

B. `LOOP` με εντολή `EXIT`

Η `exit` διακόπτει την εκτέλεση του βρόχου και μεταφέρει τη σειρά εκτέλεσης των εντολών έξω από το βρόχο. Μπορεί να διατυπωθεί απλά, χωρίς συνθήκη (unconditional) ή με συνθήκη. Η γενική διατύπωση έχει ως εξής:

```
[ετικέτα:] exit [ετικέτα_βρόχου] [when συνθήκη]
```

Ως γνωστό, οι αγγύλες σημαίνουν προαιρετική σύνταξη. Ένα γενικό παράδειγμα διατύπωσης είναι το εξής:

```
PROCESS
VARIABLE m : natural;
BEGIN
      LOOP
            m:=m+1;
            exit when m>10;
      END LOOP;
      --O έλεγχος της εκτέλεσης μεταφέρεται σ’ αυτό το σημείο,
      --όταν η συνθήκη m>10 γίνει αληθής
END PROCESS;
```

Παράδειγμα 6.9 Ο κώδικας 6.12 περιγράφει το μοντέλο απαριθμητή με εμφωλευμένους βρόχους (`LOOP`). Ο απαριθμητής χρησιμοποιεί είσοδο ενεργοποίησης και ασύγχρονη είσοδο μηδενισμού (reset). Το μοντέλο κάνει χρήση της εντολής `Exit` υπό συνθήκη, η οποία μεταφέρει τον έλεγχο στις εντολές μετά το τέλος του βρόχου. Σε εμφωλευμένους βρόχους, η ετικέτα του βρόχου μπορεί να χρησιμοποιηθεί στην εντολή `exit` για να μεταφέρει τον έλεγχο έξω από τον εξωτερικό βρόχο. Ο εσωτερικός βρόχος του κώδικα 6.12 λειτουργεί όπως αυτός του παραδείγματος 6.8, όμως εδώ προβλέπεται η έξοδος από τον βρόχο όταν το κύκλωμα λάβει είσοδο `resetn` σε λογικό μηδέν. Όταν η είσοδος ενεργοποίησης `en` γίνει μηδέν, η εκτέλεση εξέρχεται από τον εξωτερικό βρόχο, η έξοδος λαμβάνει την τελευταία τιμή της απαρίθμησης και η διεργασία αναστέλλεται, έως ότου το σήμα `en` γίνει πάλι μονάδα. Κάθε φορά που η πρόταση `WAIT` διακόπτει τη διεργασία στη μέση της εκτέλεσης του βρόχου, αποδίδονται οι τιμές των σημάτων και η έξοδος `q` ανανεώνεται.

```
------------------------------------
ENTITY loop_counter2 IS
  PORT(clk, resetn, en : IN bit;
            q : OUT natural);
END loop_counter2;
------------------------------------
ARCHITECTURE behaviour OF loop_counter2 IS
BEGIN
one:  PROCESS
VARIABLE m : natural :=0;
      BEGIN
```

```
        q<=m;
outer:    LOOP
   inner:    LOOP
               wait until (clk'event AND clk ='1') OR resetn='0';
               exit inner when resetn='0' ;
               exit outer when en='0';
               m:=(m+1) mod 16;
               q<=m;
             END LOOP inner;
             m:=0; --resetn='0'
             q<=m;
             wait until resetn='1';
           END LOOP outer;
        wait until en='1'; --en='0'
    END PROCESS one;
END behaviour;
```

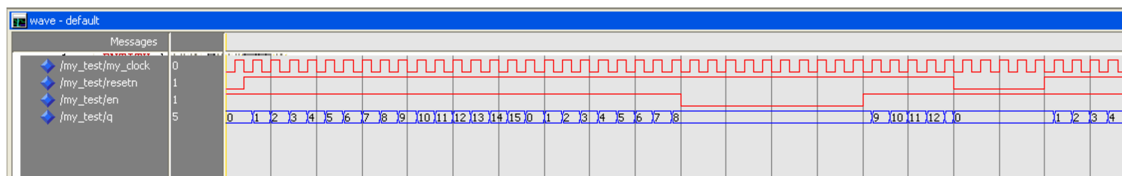
Κώδικας 6.12 Μοντέλο απαριθμητή με εμφωλευμένους βρόχους και εισόδους μηδενισμού και ενεργοποίησης.

Στο παραπάνω σχήμα φαίνεται το αποτέλεσμα της προσομοίωσης στο λογισμικό ModelSim. Η έξοδος `q` του απαριθμητή αυξάνεται διαδοχικά μέχρι και το 15. Όταν η είσοδος `resetn` λάβει λογικό μηδέν, η έξοδος μηδενίζεται.

Γ. LOOP με FOR

Πρόκειται για την πιο διαδεδομένη χρήση της `LOOP`. Η σύνταξη της εντολής είναι ως εξής:

```
[ετικέτα:] FOR αναγνωριστικό IN περιοχή_τιμών LOOP
ακολουθιακός κώδικας
END LOOP [ετικέτα];
```



Σχήμα 6.12 Το αποτέλεσμα της προσομοίωσης στο λογισμικό ModelSim

Παράδειγμα 6.10 Ο παρακάτω κώδικας είναι ένας μετρητής μονάδων, δηλαδή, λαμβάνει είσοδο ένα αριθμό με εύρος 8-bit και μετρά τα bits που είναι μονάδα.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
--------------------------------
ENTITY count_ones IS
      PORT(input_data : IN std_logic_vector(7 downto 0);
             numb_ones : OUT std_logic_vector(3 downto 0));
END count_ones;
ARCHITECTURE behavior OF count_ones IS
BEGIN
      PROCESS (input_data)
       VARIABLE count : std_logic_vector(3 downto 0) ;
        BEGIN
        count:="0000";
        FOR i IN 0 TO 7 LOOP
         count:=count+input_data(i);
        END LOOP;
       numb_ones<=count;
      END PROCESS;
END behavior;
```

Κώδικας 6.13 Κύκλωμα που απαριθμεί πόσες μονάδες υπάρχουν σε έναν αριθμό με εύρος 8-bit

Στον παραπάνω κώδικα 6.13, ας προσεχθεί ο ρόλος της μεταβλητής `count`. Η μεταβλητή συσσωρεύει τον αριθμό των μονάδων στον δυαδικό αριθμό `input_data`. Αν η `count` είχε δηλωθεί ως σήμα, τότε δεν θα μπορούσε να παίξει το ρόλο του συσσωρευτή, αφού δεν θα ανανέωνε την τιμή του παρά μόνον μετά την έξοδο από την διεργασία.
Ας σημειωθεί ότι ακριβώς με την ίδια λογική μπορεί να σχεδιαστεί ένα κύκλωμα που να μετρά τα bits που έχουν τιμή `'0'`.

Παράδειγμα 6.11 Αθροιστής μη προσημασμένων αριθμών 8-bit.

Η κεντρική ιδέα ενός απλού αθροιστή δύο αριθμών πολλών bits δόθηκε στο σχήμα 5.9. Εκεί, φαίνεται ότι η άθροιση διαδοχικών bits γίνεται με την επανάληψη του απλού πλήρη αθροιστή 1-bit. Ο πλήρης αθροιστής είναι το συνδυαστικό κύκλωμα που παρουσιάστηκε στο Σχήμα 5.2, και περιγράφεται σε VHDL στον κώδικα 5.2. Στον κώδικα 5.11 παρουσιάστηκε η δομική σχεδίαση ενός αθροιστή πολλών bits με επανάληψη του βασικού δομικού στοιχείου fulladder και χρήση της σύγχρονης εντολής `FOR…GENERATE`. Ο παρακάτω κώδικας 6.13 χρησιμοποιεί μια περιγραφή συμπεριφοράς (behavioral description) με βάση την εντολή `FOR…LOOP`, για να περιγράψει τον αθροιστή 8-bit.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------
ENTITY adder_8_bit IS
      PORT(x, y : IN std_logic_vector(7 downto 0);
              Cin : IN std_logic;
              Sum : OUT std_logic_vector(7 downto 0);
             Cout : OUT std_logic);
END adder_8_bit;
------------------------------------
ARCHITECTURE behaviour OF adder_8_bit IS
BEGIN
  PROCESS(x, y, Cin)
  VARIABLE S : std_logic_vector(7 downto 0);
  VARIABLE C : std_logic_vector(8 downto 0);
  BEGIN
      C(0):=Cin;
        FOR i IN 0 TO 7 LOOP
          S(i):=x(i) XOR y(i) XOR C(i);
          C(i+1):=(x(i) AND y(i)) OR (C(i) AND x(i)) OR (C(i) AND y(i));
        END LOOP;
        Sum<=S;
        Cout<=C(8);
  END PROCESS;
END Behaviour;
```

Κώδικας 6.13 Κώδικας συμπεριφοράς για τον αθροιστή 8-bit, με την εντολή FOR...LOOP

Παράδειγμα 6.12 Γενικός καταχωρητής ολίσθησης N-bit με την εντολή FOR...LOOP
Ο καταχωρητής ολίσθησης με τέσσερα Flip-Flops φαίνεται στο σχήμα 6.8 και περιγράφεται στο παράδειγμα 6.4. Στο σημείο αυτό περιγράφουμε έναν γενικό καταχωρητή ολίσθησης με N Flip-Flops, κάνοντας χρήση της εντολης FOR…LOOP. Έτσι, οι διαδοχικές ολισθήσεις της εισόδου στις εξόδους των Flip-Flops συμπυκνώνονται στη LOOP των γραμμών 20 έως 22. Όταν ολοκληρωθεί η ολίσθηση, η τελική ανάθεση της εξόδου γίνεται στη γραμμή 24.

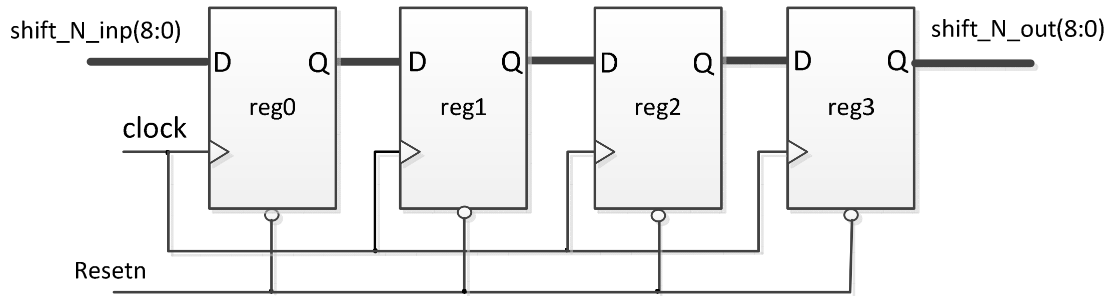
```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.all;
3 --------------------------------
4 ENTITY shift_reg_gen IS
5    GENERIC (N : NATURAL :=4);--N FFs
6          PORT(serial_in: IN std_logic;
7           clock, Resetn: IN std_logic;
8              serial_out: OUT std_logic);
```

```
9 END shift_reg_gen;
10 ---------------------------------
11 ARCHITECTURE behaviour OF shift_reg_gen IS
12 SIGNAL q : std_logic_vector(0 to N-1);
13 BEGIN
14   PROCESS(clock, Resetn)
15    BEGIN
16          IF Resetn='0' THEN
17               q<=(OTHERS=>'0');
18          ELSIF clock'EVENT AND clock='1' THEN
19               q(0)<=serial_in;
20               FOR i IN 0 TO N-2 LOOP
21               q(i+1)<=q(i);
22               END LOOP;
23          END IF;
24    serial_out<=q(N-1);
25    END PROCESS;
26 END behaviour;
```

Κώδικας 6.14 Γενικός καταχωρητής ολίσθησης N bits, με την εντολή `FOR...LOOP`

Παράδειγμα 6.13 Να περιγραφεί για γραμμή καθυστέρησης με *N* καταχωρητές στη σειρά, όπως στο σχ. 6.11. Η γραμμή καθυστέρησης αποτελείται από καταχωρητές ολίσθησης, που ο ρόλος του είναι να καθυστερούν την εμφάνιση στην έξοδο όλων των bits μιας λέξης δεδομένων και όχι απλώς ενός bit. Ο κώδικας να σχεδιαστεί με τη βοήθεια πίνακα (array) και με τη βοήθεια γενικής σταθερής *N*, ώστε να μπορεί να χρησιμοποιηθεί για οποιοδήποτε βάθος γραμμής καθυστέρησης. Η γραμμή να δέχεται είσοδο προσημασμένων αριθμών με εύρος 9-bit.

Λύση: Επειδή το κύκλωμα είναι προφανώς ακολουθιακό υλοποιείται με δομή διεργασίας (`PROCESS`). Επίσης, ο κώδικας μέσα στη διεργασία γράφεται με τη βοήθεια της μεταβλητής q. Ο μονοδιάστατος πίνακας q δηλώνεται με τη μορφή μεταβλητής μέσα στη διεργασία και αποθηκεύει τις τιμές των εξόδων των καταχωρητών. Οι καταχωρητές σχεδιάζονται με ασύγχρονη είσοδο Reset. Η δομή επανάληψης `gen0` μηδενίζει όλες τις εξόδους των καταχωρητών όταν το σήμα Resetn πάρει λογικό 0. Η δομή `gen1` μεταφέρει αλυσιδωτά, με διαδικασία σειριακής ολίσθησης σε κάθε έξοδο την έξοδο του προηγούμενου καταχωρητή (`q(i):=q(i-1);`). Αυτό συμβαίνει σε κάθε θετικό μέτωπο των παλμών ρολογιού. Για *N*=4, η είσοδος `shift_N_inp` που υπάρχει τώρα στον πρώτο καταχωρητή θα εμφανιστεί στην έξοδο `shift_N_out` μετά από 4 παλμούς ρολογιού. Ουσιαστικά, η δομή gen1 καθοδηγεί τον compiler να δημιουργήσει μια συστοιχία καταχωρητών, όπου ο καθένας λαμβάνει είσοδο από την έξοδο του προηγούμενου. Ας προσεχθεί ο τελεστής ανάθεσης σε μεταβλητές (`:=`). Η τιμή `q(N)` της μεταβλητής, που αποθηκεύει την τιμή της εξόδου του τελευταίου προς τα δεξιά καταχωρητή, ανατίθεται στην τελι-



Σχήμα 6.11 Γραμμή καθυστέρησης με 4 καταχωρητές στη σειρά. Κάθε καταχωρητής μπορεί να καταχωρήσει δεδομένο με εύρος *M* bits.

κή έξοδο `shift_N_out`, στο τέλος της διεργασίας. Στην παράγραφο 6.9 εξηγήθηκε αναλυτικά η ανάγκη ώστε η σειρά ανάθεσης των μεταβλητών q(i) να οδεύει από το τέλος προς την αρχή. Ο τύπος δεδομένων είναι `SIGNED` με εύρος 9 bits, ώστε το σύστημα να διαχειρίζεται δυαδικούς στην δεκαδική περιοχή τιμών -256 έως +255. Προφανώς, ο αριθμός των bits των δεδομένων που καταχωρούνται και διαδίδονται σειριακά, μπορεί να γενικευτεί με τη βοήθεια μιας ακόμη γενικής σταθεράς.

```
1 -------------------------------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 USE ieee.numeric_std.all;
5 -------------------------------
6 ENTITY delay_line IS --Delay line N-positions-deep
   7  GENERIC(N: INTEGER :=4); -- four positions-deep delay line       8
            PORT(shift_N_inp: IN SIGNED(8 downto 0);
9                 clock, Resetn: IN std_logic;
10                shift_N_out: OUT SIGNED(8 downto 0));
11 END delay_line;
12 ---------------------------------
13 ARCHITECTURE delay OF delay_line IS
14 SUBTYPE sample IS SIGNED(8 downto 0);
15 TYPE OneD IS ARRAY(NATURAL RANGE <>) OF sample;
16 BEGIN
17   PROCESS(clock, Resetn)
18      VARIABLE q : OneD(0 TO N);--Array q holds the outputs of FFs
19    BEGIN
20    IF Resetn='0' THEN
21    gen0: FOR i IN 1 TO N LOOP
22          q(i):=(OTHERS=>'0');
23          END LOOP;
```

```
24          ELSIF clock'EVENT AND clock='1' THEN
25          q(0):=shift_N_inp;
26    gen1: FOR i IN N DOWNTO 1 LOOP --see paragraph 6.9
27          q(i):=q(i-1);
28          END LOOP;
29          END IF;
30    shift_N_out<=q(N);
31    END PROCESS;
32 END delay;
```

Κώδικας 6.15 Γενική γραμμή καθυστέρησης (delay-line) με *N* Flip-Flops (εδώ *N*=4), σχεδιασμένη με τη βοήθεια μονοδιάστατου πίνακα και εντολή FOR...LOOP

Δ. LOOP με WHILE

Η εντολή αυτή συντάσσεται ως εξής:

```
[ετικέτα:] WHILE συνθήκη LOOP
ακολουθιακός κώδικας
END LOOP [ετικέτα];
```

6.14 Η εντολή CASE

Η εντολή CASE έχει παρόμοια λειτουργία με την IF, αφού κι αυτή επιλέγει ανάμεσα σε διαφορετικές ομάδες ακολουθιακών εντολών με βάση την ισχύ μιας συνθήκης. Η διαφορά είναι ότι η IF εξετάζει διαδοχικά την ισχύ μιας σειράς λογικών συνθηκών, προκειμένου να επιλέξει την ομάδα εντολών που θα εκτελέσει. Η CASE επιλέγει με βάση την τιμή που λαμβάνει μια μοναδική έκφραση. Η σύνταξή της είναι ως εξής:

```
[ετικέτα:] CASE έκφραση IS
      WHEN τιμή => αναθέσεις;
      WHEN τιμή => αναθέσεις;
      ....
END CASE;
```

Ένα παράδειγμα είναι το σύστημα ελέγχου ενός επεξεργαστή, που κάνει διαφορετικές αναθέσεις στα σήματα ελέγχου x, y, ανάλογα με την τιμή του κωδικού εντολής. Έστω, λοιπόν, ότι σε μια απλή περίπτωση ο κωδικός εντολής opcode μπορεί να λάβει τις τιμές “00”, “01”, “10”, “11”. Έτσι, η CASE θα διατυπωθεί ως εξής:

```
CASE opcode IS
      WHEN "00"=>
            x<='0'; y<='1';
      WHEN "01"=>
            x<='1'; y<='0';
      WHEN "10"=>
            x<='0'; y<='1';
```

```
    WHEN "11" =>
         x<='1'; y<='1';
END CASE;
```

Προκειμένου να ληφθούν υπόψη όλες οι δυνατές τιμές της έκφρασης επιλογής, είναι δυνατό η τελευταία WHEN να είναι διατυπωμένη με την κωδική έκφραση OTHERS:

```
    WHEN OTHERS => αναθέσεις;
```

6.15 Η εντολή CASE στις μηχανές πεπερασμένων καταστάσεων

Η εντολή CASE είναι πολύ χρήσιμη στην περιγραφή μηχανών πεπερασμένων καταστάσεων (Finite State Machines). Μια μηχανή πεπερασμένων καταστάσεων είναι ένα ακολουθιακό κύκλωμα που υλοποιείται με τη βοήθεια συνδυαστικής λογικής και ενός ή περισσοτέρων Flip-Flops. Σε κάθε διαδοχικό παλμό ρολογιού το κύκλωμα βρίσκεται σε μια από τις δυνατές καταστάσεις του. Η μετάβαση στην επόμενη κατάσταση συμβαίνει με βάση την τρέχουσα κατάσταση και την τιμή του σήματος εισόδου. Τα σήματα εξόδου εξαρτώνται από την τρέχουσα κατάσταση (μηχανή του Moore) ή από την τρέχουσα κατάσταση και την είσοδο (μηχανή του Mealy). Τυπικά, μια τέτοια μηχανή καταστάσεων μπορεί να περιγραφεί από μια διεργασία (PROCESS) η οποία περιλαμβάνει το σήμα ρολογιού στη λίστα ευαισθησίας. Επίσης, περιλαμβάνει μια CASE που αναθέτει την επόμενη κατάσταση (y_next) με βάση την τρέχουσα κατάσταση (y_present) και την τιμή της εισόδου, για κάθε υποπερίπτωση της CASE. Τα σήματα εξόδου παράγονται με τη βοήθεια σύγχρονων (concurrent) εντολών, με βάση την τρέχουσα κατάσταση (μηχανή του Moore) ή την τρέχουσα κατάσταση και την τιμή της εισόδου (μηχανή του Mealy). Περισσότερα για τις μηχανές καταστάσεων θα αναπτυχθούν σε μεταγενέστερη επέκταση του βοηθήματος.

Άσκηση 6.1 Να σχεδιάσετε έναν πολυπλέκτη 4:1 με κανάλια εύρους 4-bit, κάνοντας χρήση της εντολής CASE.

Άσκηση 6.2 Να σχεδιάσετε έναν κωδικοποιητή BCD σε οθόνη απεικόνισης επτά τομέων, όπως το κύκλωμα που περιγράφηκε στην παράγραφο 5.4, κάνοντας χρήση της εντολής CASE.

7 Δομική σχεδίαση και πακέτα

7.1 Δομικά στοιχεία (components)

Η VHDL είναι μια γλώσσα που επιτρέπει την ιεραρχική σχεδίαση κυκλωμάτων. Έτσι, ο χρήστης μπορεί να περιγράψει ένα σύνθετο κύκλωμα με βάση τα υποκυκλώματα που το αποτελούν. Τα υποκυκλώματα που συμπεριλαμβάνονται σε μια ανώτερη οντότητα (top-level entity) ονομάζονται δομικά **στοιχεία** (components). Τα στοιχεία είναι μικρότερα κυκλώματα, που έχουν ήδη περιγραφεί σε VHDL. Τέτοια αρχεία μπορεί να είναι έτοιμοι κώδικες που περιλαμβάνονται στο λογισμικό σχεδίασης ή κώδικες που σχεδιάζει ο ίδιος ο χρήστης.

Το σχήμα 1.2 (παράγραφος 1.3) παρουσιάζει τη βασική ιδέα της δομικής σχεδίασης. Όπως φαίνεται στο σχήμα, μια ανώτερη οντότητα δομείται με τη βοήθεια στοιχείων, των οποίων ο κώδικας περιλαμβάνεται στη βιβλιοθήκη εργασίας. Η κάθε μια από τις κατώτερες οντότητες μπορεί να περιλαμβάνει στη σχεδίασή της άλλες οντότητες. Με τον τρόπο αυτό μπορούν να περιγραφούν μεγάλα ψηφιακά συστήματα.

Τα κύρια χαρακτηριστικά της σχεδίασης μιας ανώτερης ιεραρχικής οντότητας με υποκυκλώματα είναι η δήλωση των δομικών στοιχείων που περιέχονται στο κύκλωμα, καθώς και η περιγραφή του τρόπου με τον οποίο τα στοιχεία συνδέονται μεταξύ τους.

Με την δήλωση του κάθε υποκυκλώματος καθορίζουμε το όνομα του, καθώς και τα ονόματα των εισόδων και των εξόδων του. Η δήλωση των δομικών στοιχείων μπορεί να γίνει στην περιοχή δηλώσεων της αρχιτεκτονικής ή σ' ένα πακέτο (PACKAGE). Η γενική μορφή μιας τέτοιας δήλωσηςείναι ως εξής:

```
COMPONENT όνομα συνιστώσας
      [GENERIC (όνομα παραμέτρου : integer := τιμή; ]
              PORT (όνομα ακροδέκτη : mode τύπος;
                     όνομα ακροδέκτη : mode τύπος );
END COMPONENT;
```

Παρακάτω δίνεται ένα παράδειγμα δήλωσης ενός απαριθμητή (οντότητα upcount) ως δομικού στοιχείου:

```
COMPONENT upcount
PORT(clock, Resetn, E: IN     std_logic;
                    Q: OUT std_logic_vector (3 DOWNTO 0));
END COMPONENT;
```

Στη συνέχεια, στο κύριο σώμα της αρχιτεκτονικής, πρέπει να δημιουργήσουμε στιγμιότυπα για τα υποκυκλώματα που έχουμε δηλώσει, και να περιγράψουμε τις συνδέσεις τους. Η δημιουργία των στιγμιοτύπων γίνεται πάντα μετά τη δήλωση ενός δομικού στοιχείου. Τα στιγμιότυπα είναι της μορφής:

```
Όνομα_στιγμιότυπου: όνομα στοιχείου PORT MAP (ονόματα σημάτων);
```

Στη δήλωση PORT MAP δημιουργούμε τις διασυνδέσεις ανάμεσα στις εισόδους/εξόδους κάθε υποκυκλώματος και στα σήματα της ανώτερης οντότητας την οποία περιγράφουμε.

Η ετικέτα «όνομα_στιγμιότυπου» είναι υποχρεωτική. Η ονομασία μπορεί να είναι οποιαδήποτε, αρκεί να υπακούει στους κανόνες ονοματοδοσίας των αναγνωριστικών.

Παράδειγμα δημιουργίας στιγμιοτύπου του απαριθμητή upcount, που αναφέρθηκε παραπάνω είναι το εξής:

```
stage1: upcount PORT MAP (clk, Rstn, En, QS);
```

Εδώ, τα σήματα clk, Rstn, En, QS ανήκουν στην οντότητα που χρησιμοποιεί τον upcount ως δομικό στοιχείο και διασυνδέονται ένα προς ένα με τα σήματα που αναφέρονται στη δήλωση PORT του στοιχείου upcount (clock, Resetn, E, Q).

Παράδειγμα 7.1 Θα σχεδιαστεί ένα κύκλωμα, που στηρίζεται σε δύο υποκυκλώματα. Το πρώτο είναι ένας αύξων απαριθμητής (upcounter), ο οποίος απαριθμεί από το 0 έως και το 9, παρόμοιος με αυτόν που περιγράφεται στον κώδικα 6.9. Το δεύτερο είναι ένας κωδικοποιητής BCD σε απεικόνιση επτά τομέων, όπως στο παράδειγμα 5.4. Σχεδιάζεται ένα ιεραρχικό κύκλωμα που δημιουργεί στιγμιότυπα από τα δύο αυτά δομικά στοιχεία και εξάγει σε ενδείκτη επτά τομέων τη δεκαδική τιμή που απαριθμεί ο counter.
Πρώτα, πρέπει να δημιουργηθούν οι παρακάτω κώδικες και να αποθηκευτούν στον ίδιο φάκελο εργασίας.

Ο απαριθμητής

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
-----------------------------------------------------------------
ENTITY upcount IS
PORT(clock, Resetn, E: IN     std_logic;
                       Q: OUT std_logic_vector (3 DOWNTO 0));
END upcount;
ARCHITECTURE behaviour OF upcount IS
SIGNAL Count: STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN
  PROCESS(Clock, Resetn)
  BEGIN
  IF Resetn='0' THEN
           Count<="0000";
  ELSIF Clock'EVENT AND Clock='1' THEN
      IF E='1' THEN
           IF Count<9 THEN
                Count<=Count+1;
           ELSE
                Count<="0000";
           END IF;
      ELSE
           Count<=Count;
      END IF;
```

```
  END IF;
  END PROCESS;
  Q<=Count;
END behaviour;
```

Κώδικας 7.1 Απαριθμητής mod 10

Ο αποκωδικοποιητής BCD-to-7 segment

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------------------------------
ENTITY seg_7 IS
  PORT(c : IN      std_logic_vector (3 DOWNTO 0);
      ex1: OUT std_logic_vector (6 DOWNTO 0));
END seg_7;
---------------------------------------------------------------
ARCHITECTURE behaviour OF seg_7 IS
BEGIN
WITH c SELECT
         ex1<= "1111110" WHEN "0000", --0
               "0000110" WHEN "0001", --1
               "1101101" WHEN "0010", --2
               "1111001" WHEN "0011", --3
               "0110011" WHEN "0100", --4
               "1011011" WHEN "0101", --5
               "1011111" WHEN "0110", --6
               "1110000" WHEN "0111", --7
               "1111111" WHEN "1000", --8
               "1111011" WHEN "1001", --9
               "0000001" WHEN OTHERS;
END behaviour;
```

Κώδικας 7.2 Μετατροπέας BCD σε απεικόνιση επτά τομέων

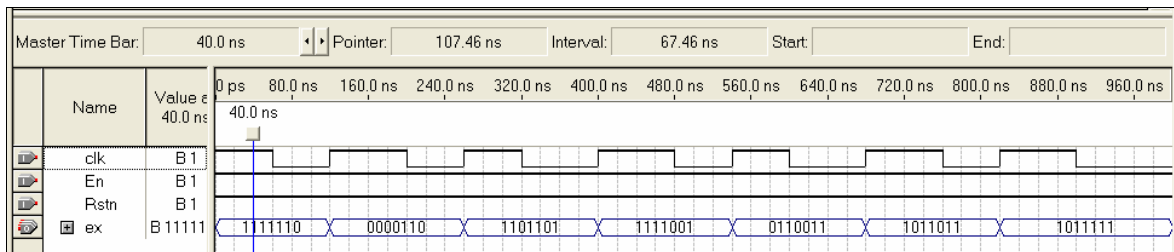
Ιεραρχικός κώδικας VHDL που ενσωματώνει τον απαριθμητή και τον αποκωδικοποιητή BCD-to-SSD

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------------------------------
ENTITY count_sev_segm IS
PORT(clk, Rstn, En: IN     std_logic;
                ex: OUT std_logic_vector (6 DOWNTO 0));
END count_sev_segm;
```

```
------------------------------------------------------------------
ARCHITECTURE structure OF count_sev_segm IS
      SIGNAL QS: std_logic_vector (3 DOWNTO 0);
      SIGNAL ex2: std_logic_vector (6 DOWNTO 0);
      ----------------------
      COMPONENT upcount
      PORT(clock, Resetn, E : IN std_logic;
                          Q : OUT std_logic_vector (3 DOWNTO 0));
      END COMPONENT;
      ----------------------
      COMPONENT seg_7
      PORT(c : IN std_logic_vector (3 DOWNTO 0);
          ex1: OUT std_logic_vector(6 DOWNTO 0));
      END COMPONENT;
      --------------------
BEGIN
      stage1:upcount PORT MAP(clk,Rstn,En,QS);
      stage2:seg_7 PORT MAP(QS,ex2);
      ex(6 DOWNTO 0)<=ex2(6 DOWNTO 0);
END structure;
```

Κώδικας 7.3 Δομική σχεδίαση του συνολικού κυκλώματος

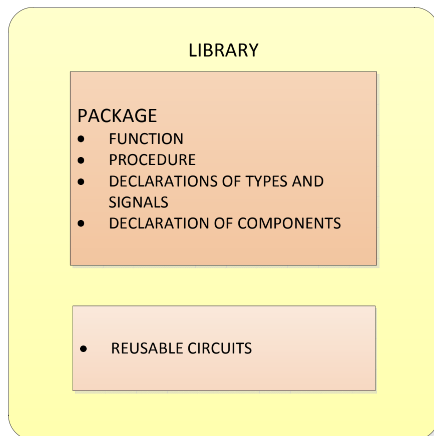
Η λειτουργική προσομοίωση του ιεραρχικού σχεδίου στο περιβάλλον Quartus II, φαίνεται στο παρακάτω σχήμα.



Σχήμα 7.1 Η προσομοίωση του κυκλώματος που περιγράφει ο κώδικας 7.2. Σε κάθε παλμό clock η έξοδος στην απεικόνιση επτά τομέων αυξάνεται κατά 1.

7.2 Βιβλιοθήκες και πακέτα

Οι βιβλιοθήκες λειτουργούν ως συλλογές, μέσα στις οποίες αποθηκεύονται κώδικες σε γλώσσα VHDL. Ο σκοπός μιας βιβλιοθήκης είναι να συγκεντρώνει σχεδιάσεις που θα χρησιμοποιηθούν ξανά από τους ίδιους ή άλλους χρήστες. Η δυνατότητα αυτή αναφέρεται ως επαναχρησιμοποίηση του κώδικα (code reusability). Μια βιβλιοθήκη είναι οργανωμένη σε έναν φάκελο αρχείων, όπου συγκεντρώνονται όλοι οι κώδικες της συλλογής.



Σχ. Π.7.2 Βασικά μέρη μιας βιβλιοθήκης

Όπως φαίνεται στο σχήμα 7.1, μέρος μιας βιβλιοθήκης είναι τα πακέτα. Τα πακέτα, είναι τμήματα κώδικα, που παρέχουν έναν τρόπο οργάνωσης των δεδομένων, σε μια μεγάλη ψηφιακή σχεδίαση. Ένα ψηφιακό σύστημα αποτελείται από έναν αριθμό δομικών στοιχείων, τα οποία συνδέονται ιεραρχικά, όπως περιγράφηκε στο προηγούμενο κεφάλαιο. Τα πακέτα περιλαμβάνουν τις βασικές δηλώσεις τύπων, σημάτων, μεταβλητών και σταθερών που είναι κοινές σε πολλές σχεδιάσεις, που περιέχονται στο σύστημα. Μέσα στα πακέτα μπορούν να συμπεριληφθούν επίσης, δηλώσεις δομικών στοιχείων, συναρτήσεις και υποπρογράμματα. Έτσι, οι δηλώσεις αυτές δεν διασκορπίζονται στους επιμέρους κώδικες που αποτελούν ένα σύστημα, αλλά είναι συγκεντρωμένοι με ιεραρχικό τρόπο, σε ένα πακέτο.

Τα πακέτα μπορούμε να τα χωρίσουμε σε δύο μέρη, στην δήλωση του πακέτου και στο σώμα του πακέτου, του οποίου η χρήση είναι προαιρετική. Η γενική μορφή ενός πακέτου είναι:

```
PACKAGE όνομα_πακέτου IS
        [Δηλώσεις σημάτων]
        [Δηλώσεις δομικών στοιχείων]
END όνομα πακέτου ;
-------------------------------------------
PACKAGE BODY όνομα_πακέτου IS
   Κώδικας υποπρογράμματος (function ή procedure)
END PACKAGE BODY [όνομα_πακέτου]
```

Παράδειγμα 7.2 Έστω η ιεραρχική σχεδίαση μια γραμμής καθυστέρησης (delay line) με βάθος `NF`=4. Το κύκλωμα της γραμμής έχει δοθεί στο σχήμα 6.11, όπου φαίνεται ότι τα δεδομένα ολισθαίνουν σε διαδοχικούς καταχωρητές. Στις παρακάτω γραμμές παρουσιάζεται η δομική περιγραφή της γραμμής καθυστέρησης, με τη βοήθεια ενός πακέτου που ορίζεται από τον χρήστη και ενός δομικού στοιχείου.

Στο πακέτο γίνεται η δήλωση δύο σταθερών `N`, `NF`, ενός υποτύπου `sample` και ενός σύνθετου τύπου πίνακα. Προφανώς, οι δηλώσεις αυτές δεν θα επαναληφθούν στα επί μέρους τμήματα κώδικα, όπου θα χρησιμοποιηθούν οι τύποι και οι σταθερές. Τέλος, γίνεται η δήλωση των δομικών στοιχείων που θα περιλαμβάνει η σχεδίαση. Προφανώς, η δήλωση αυτή δεν θα επαναληφθεί στο αντίστοιχο τμήμα της αρχιτεκτονικής της ανώτερης ιεραρχικής οντότητας.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
------------------------------
PACKAGE my_package IS
--CONSTANT DECLARATIONS----------------------------------------
CONSTANT N: INTEGER RANGE 0 TO 32 :=8;--Number of bits in samples
CONSTANT NF: NATURAL:=4;--Number of regs in delay line
--TYPE DECLARATIONS--------------------------------------------
SUBTYPE sample IS SIGNED(8 downto 0);
TYPE OneD IS ARRAY(NATURAL RANGE <>) OF sample;
--COMPONENT DECLARATIONS---------------------------------------
COMPONENT reg IS --DFF
      PORT (d_in: IN sample;
      clock, Resetn : IN std_logic;
               q_out: OUT sample);
END COMPONENT;
END PACKAGE;
```

Κώδικας 7.4 Το πακέτο που ορίζεται από το χρήστη για τις δηλώσεις των τύπων και του δομικού στοιχείου, που χρειάζονται για τη σχεδίαση της γραμμής καθυστέρησης.

Ο καταχωρητής `reg` αποτελεί το δομικό στοιχείο της σχεδίασης. Έχει είσοδο και έξοδο τύπου `sample`. Τα διαδοχικά στιγμιότυπά του καταχωρητή υλοποιούν τη γραμμή. Ο κώδικάς του δίνεται παρακάτω και πρέπει να είναι αποθηκευμένος στο φάκελο εργασίας του project. Προφανώς, είναι απαραίτητο να γίνει χρήση (`USE`) του παραπάνω πακέτου `my_package`.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.my_package.all;
---------------------------------------------
ENTITY reg IS
      PORT (d_in: IN sample;
       clock, Resetn: IN std_logic;
           q_out: OUT sample);
 END reg;
---------------------------------------------
ARCHITECTURE behaviour OF reg IS
BEGIN
      PROCESS (Resetn, clock)
      BEGIN
        IF Resetn='0' THEN
           q_out<=(OTHERS=>'0');
        ELSIF clock'event AND clock='1' THEN
           q_out<=d_in;
        END IF;
```

```
END process;
END behaviour;
```

Κώδικας 7.5 Το δομικό στοιχείο reg της γραμμής καθυστέρησης

Τέλος, δίνεται η ανώτερη ιεραρχική οντότητα για τη γραμμή καθυστέρησης, η οποία περιγράφει τη γραμμή μόνο με σύγχρονες εντολές, επαναλαμβάνοντας `NF` (εδώ `NF=4`) φορές τον καταχωρητή `reg`, με τη βοήθεια εντολής `GENERATE`. Ας προσεχθεί ο ρόλος του μονοδιάστατου πίνακα `q`, που αποτελείται από στοιχεία τύπου `sample`. Ο πίνακας αυτός περιέχει τις διαδοχικές εξόδους των καταχωρητών, αφού σε κάθε στιγμιότυπο `i` που δημιουργείται μέσα στη `FOR...GENERATE` το στοιχείο `q(i)` του πίνακα αντιστοιχίζεται στην έξοδο `q_out` του καταχωρητή. Κάθε προηγούμενη έξοδος γίνεται είσοδος στον επόμενο καταχωρητή, ώστε να δημιουργηθεί η σειριακή γραμμή καθυστέρησης.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.my_package.all;
-----------------------------------------------------
ENTITY delay_line IS --Delay line NF-positions-deep, top level
      PORT(shift_in : IN sample;
            clock, Resetn: IN std_logic;
                shift_out: OUT sample);
END delay_line;
ARCHITECTURE delay OF delay_line IS
SIGNAL q : OneD(0 TO NF);--Array q holds the outputs of regs
BEGIN
            q(0)<=shift_in;
      gen1: FOR i IN 1 TO NF GENERATE
      inst: reg  PORT MAP (q(i-1), clock, Resetn, q(i));
            END GENERATE;
            shift_out<=q(NF);
 END delay;
```

Κώδικας 7.6 Η ανώτερη οντότητα (top-level entity) της γραμμής καθυστέρησης

Σε βιβλιοθήκες που ορίζονται από τον χρήστη, ο φάκελος αποθήκευσης του κώδικα είναι συνήθως ο φάκελος εργασίας. Μια τέτοια βιβλιοθήκη εργασίας ονομάζεται `work` και μπορεί προαιρετικά να δηλωθεί ως εξής:

```
library work;
```

Στην πράξη, η βιβλιοθήκη `work` δεν χρειάζεται να δηλωθεί ως ξεχωριστή βιβλιοθήκη, αφού ο φάκελος εργασίας είναι αυτόματα ορατός. Το πακέτο του χρήστη δηλώνεται ως εξής:

```
USE όνομα_βιβλιοθήκης.όνομα_πακέτου.all ;
```

7.3 Η δήλωση GENERIC MAP

Όπως είδαμε στην παράγραφο 2.9, είναι δυνατό να γενικεύσουμε τον κώδικα που περιγράφει ένα ψηφιακό κύκλωμα με τη βοήθεια της δήλωσης GENERIC. Η δήλωση αυτή ορίζει ορισμένες σταθερές ως γενικές σταθερές και χρησιμοποιείται για τη γενίκευση του αριθμού των bits των δεδομένων και εν γένει για την παραμετροποίηση του κυκλώματος. Διάφορα παραδείγματα σε όλη την έκταση του βοηθήματος επιδεικνύουν τη χρήση της δήλωσης GENERIC.

Έστω ότι έχουμε περιγράψει ένα δομικό στοιχείο (COMPONENT) με τη γενίκευση GENERIC. Τότε είναι δυνατό να μεταφέρουμε τιμές στις γενικευμένες σταθερές από μια ανώτερη ιεραρχική οντότητα, κατά τη δημιουργία στιγμιοτύπων του δομικού στοιχείου. Η δήλωση με την οποία μεταφέρονται οι τιμές είναι η GENERIC MAP. Με τον τρόπο αυτό, το δομικό στοιχείο μπορεί να χρησιμοποιηθεί με πραγματικά γενικό τρόπο, αφού κάθε στιγμιότυπο μπορεί να είναι προσαρμοσμένο σε διαφορετικές ανάγκες.

Παράδειγμα 7.3 Να περιγράψετε τον καταχωρητή του κώδικα 7.5 με τη δήλωση GENERIC. Στη συνέχεια να δημιουργήσετε ένα στιγμιότυπο του καταχωρητή, με εύρος 16 bits, κάνοντας χρήση της δήλωσης GENERIC MAP

Λύση: Η γενική περιγραφή του δομικού στοιχείου δίνεται στον παρακάτω κώδικα.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------------
ENTITY reg_gen IS
GENERIC(N : Integer RANGE 1 TO 32 :=8);
PORT (d_in: IN std_logic_vector(N-1 downto 0);
      clock, Resetn: IN std_logic;
           q_out: OUT std_logic_vector(N-1 downto 0));
END reg_gen;
---------------------------------------------
ARCHITECTURE behaviour OF reg_gen IS
BEGIN
     PROCESS (Resetn, clock)
     BEGIN
       IF Resetn='0' THEN
          q_out<=(OTHERS=>'0');
       ELSIF clock'event AND clock='1' THEN
          q_out<=d_in;
       END IF;
END process;
END behaviour;
```

Κώδικας 7.7 Γενική περιγραφή καταχωρητή με τη δήλωση GENERIC

Στη συνέχεια, δημιουργούμε ένα στιγμιότυπο του παραπάνω καταχωρητή, μεταφέροντας τιμή στην γενική σταθερή N μέσω της δήλωσης `GENERIC MAP`:

```
--DECLARATION OF COMPONENT--------------------------
COMPONENT reg_gen IS --DFF
      PORT (d_in: IN std_logic_vector(N-1 downto 0);
      clock, Resetn : IN std_logic;
                q_out: OUT std_logic_vector(N-1 downto 0));
END COMPONENT;
----------------------------------------------------
--INSTANTIATION OF COMPONENT------------------------
inst1: reg_gen GENERIC MAP (16) PORT MAP(d, clk, Rstn, q);
----------------------------------------------------
```

όπου `d`, `clk`, `Rstn` και `q` τα σήματα της ανώτερης οντότητας που αντιστοιχίζονται στα σήματα διασύνδεσης του δομικού στοιχείου. Όπως έχει αναφερθεί, η αντιστοίχηση αυτή στηρίζεται στη θέση όπου αναγράφεται το κάθε σήμα μέσα στη δήλωση `PORT MAP` και ονομάζεται αντιστοίχηση θέσης.

Ας υποθέσουμε ότι σε ένα δομικό στοιχείο έχουμε ορίσει δύο γενικές σταθερές N, K. Τότε, κατά τη δημιουργία στιγμιοτύπου θα μπορούμε να μεταφέρουμε τιμές και στις δύο σταθερές:

```
--INSTANTIATION OF COMPONENT------------------------
inst1: reg_gen GENERIC MAP (N=>16, K=>8)
        PORT MAP(d_in=>d, q_out=>q, clock=>clk, Resetn=>Rstn);
----------------------------------------------------
```

Παραπάνω, χρησιμοποιήσαμε τον τελεστή => για να αντιστοιχίσουμε ανάμεσα σε σήματα και τιμές. Με τον τρόπο αυτό δεν είναι υποχρεωτικό να τηρείται η αντιστοίχηση θέσης.

Σε επόμενη επέκταση του βοηθήματος θα περιληφθούν τα παρακάτω θέματα

8. Υποπρογράμματα
9. Προσομοίωση
10. Σχεδίαση ενός επεξεργαστή
11. Ψηφιακή Επεξεργασία σήματος με VHDL
12. Ασκήσεις

ΒΙΒΛΙΟΓΡΑΦΙΑ

1. Volnei A. Pedroni, Circuit Design and Simulation with VHDL, Second Edition, The MIT Press, 2010.
2. Peter J. Ashenden, Jim Lewis, The designer's Guide to VHDL, Morgan Kaufmann.
3. Stephen Brown, Zvonko Vranesic, Σχεδίαση Ψηφιακών Συστημάτων με τη γλώσσα VHDL, Εκδόσεις Τζιόλα, 2001.
4. John Wakerly, Ψηφιακή Σχεδίαση, Αρχές και Πρακτικές, Εκδόσεις Κλειδάριθμος, 2002.

Παράρτημα Α: Εισαγωγή στο λογισμικό Quartus II

Π.1 Εισαγωγή, Σύνθεση και Προσομοίωση του Ψηφιακού Κυκλώματος

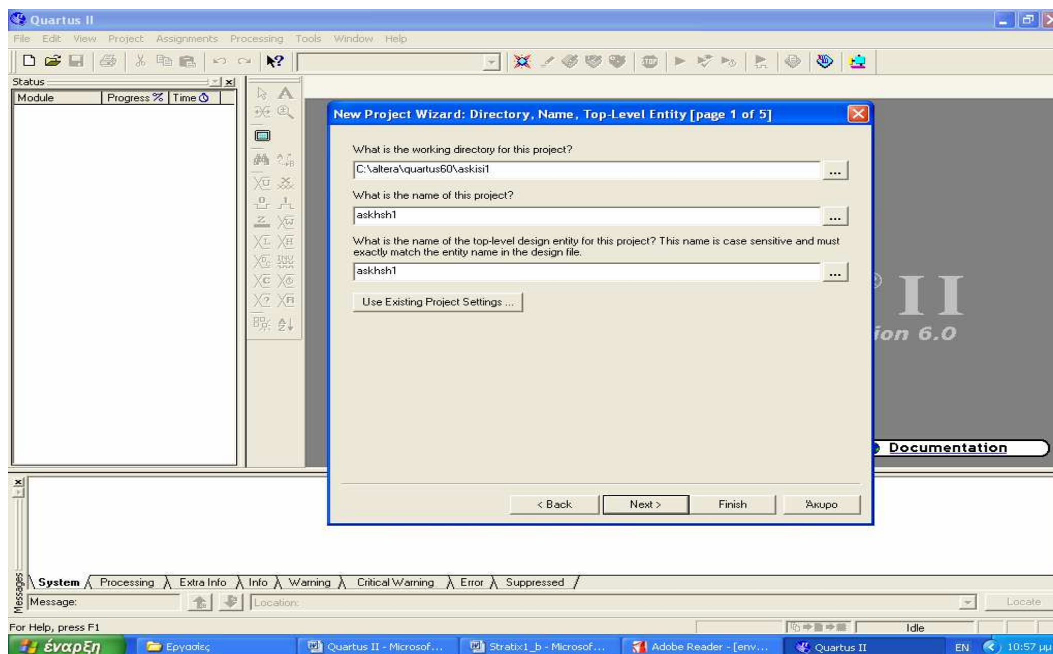
Σ' αυτή την ενότητα θα περιγράψουμε με ένα παράδειγμα την χρήση και τις βασικές λειτουργίες του εργαλείου σχεδίασης Quartus II, της εταιρίας Altera. Τα βήματα που ακολουθούμε ξεκινούν από τον τρόπο σχεδίασης ενός ψηφιακού κυκλώματος (μέσω σχηματικού διαγράμματος) και φτάνουν στον προγραμματισμό ενός FPGA, ώστε να εκτελεί την λειτουργία του κυκλώματος που σχεδιάσαμε. Στο παράδειγμα αυτό θα υλοποιήσουμε τη συνάρτηση $\mathbf{F=X_1X_2+X_1'X_3+X_2'X_3'}$.

Π.2 Ορισμός του σχεδίου (Project)

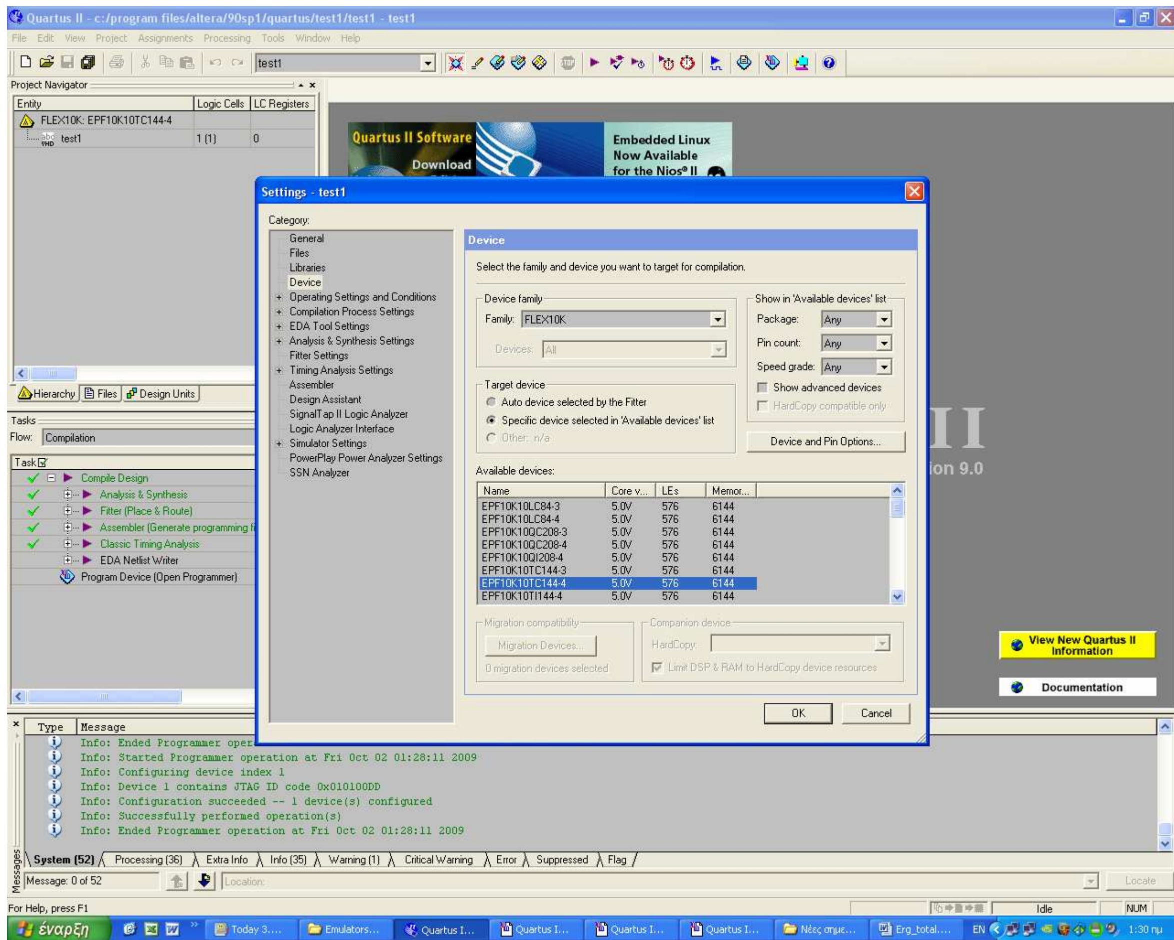
Ανοίγοντας το σχεδιαστικό περιβάλλον του Quartus II μας ζητείται η ονομασία του project. Η έννοια του project περιλαμβάνει το σύνολο των αρχείων που δημιουργούμε εμείς (το αρχικό σχέδιο και τις κυματομορφές εισόδου για την προσομοίωση), καθώς και τα αρχεία που δημιουργεί το πρόγραμμα για να εκτελέσει τις διάφορες λειτουργίες του. Με λίγα λόγια, σαν project θεωρούμε το σύνολο των αρχείων που δημιουργούνται για μια εφαρμογή. Η δημιουργία ενός project γίνεται από το **Menu File/New Project Wizard**.

Στη συνέχεια, ανοίγει ένα παράθυρο στο οποίο δηλώνουμε το όνομα του Project, καθώς και το όνομα του φακέλου που θα το περιέχει (σχήμα Π.4). Δημιουργείστε έναν διαφορετικό φάκελο, για κάθε ξεχωριστή εφαρμογή. Ποτέ μην αποθηκεύετε στον ίδιο φάκελο δύο διαφορετικά σχέδια (projects). ***Μια καλή συμβουλή είναι να χρησιμοποιείτε το ίδιο όνομα για τα τρία πεδία της καρτέλας του σχ. Π.4***.

Πατώντας το NEXT στην ίδια καρτέλα, το πρόγραμμα μας ζητάει να επιλέξουμε από λίστα την συγκεκριμένη διάταξη FPGA που θα χρησιμοποιήσουμε. Στο παράδειγμά μας, το όνομα του Project είναι askhsh1 και το FPGA που θα χρησιμοποιήσουμε ανήκει στην οικογένεια **FLEX10K** και είναι το **EPF 10K10TC144-4** (βλέπε σχ. Π.5).



Σχήμα Π.4 Δημιουργία Φακέλου και Ονομασία του Project



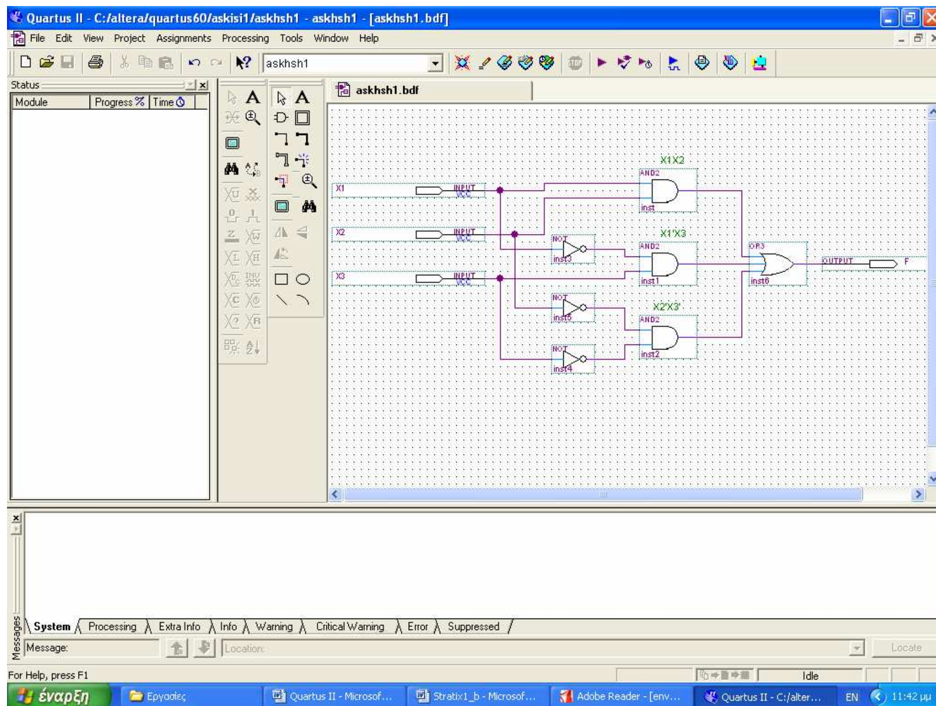
Σχήμα Π.5 Επιλογή του FPGA που θα προγραμματίσουμε

Π.3 Εισαγωγή σχηματικού αρχείου

Αφού έχουμε δώσει όνομα στο Project μπορούμε να συνεχίσουμε κάνοντας την εισαγωγή του κυκλώματος μας με όποιον τρόπο επιθυμούμε. Η επιλογή αυτή γίνεται από την καρτέλα **New** στο **Menu File**. Εκεί έχουμε να επιλέξουμε ανάμεσα σε διαφορετικούς τρόπους δημιουργίας του κυκλώματος μας. Αν θέλουμε να το περιγράψουμε σχηματικά διαλέγουμε την επιλογή **Block Diagram/Schematic File**, ενώ αν θέλουμε να το περιγράψουμε με κάποια γλώσσα περιγραφής υλικού διαλέγουμε μία από τις ακόλουθες επιλογές: **Verilog HDL File** ή **VHDL File**.

Στην περίπτωση που το περιγράφουμε σχηματικά, το Quartus II μας παρέχει ένα μεγάλο αριθμό από πύλες, από ακροδέκτες εισόδου/εξόδου καθώς και από άλλα κυκλώματα όπως flip-flop ώστε να κάνει την σχεδίαση του κυκλώματος μας πιο εύκολη. Όλα αυτά υπάρχουν στο **Menu Edit/Insert Symbol**. Εναλλακτικά, οι βιβλιοθήκες ανοίγουν πιέζοντας το πλήκτρο Symbol Tool, στα αριστερά του επεξεργαστή. Οι βασικές πύλες βρίσκονται στη βιβλιοθήκη **Primitives/logic**, ενώ οι ακροδέκτες I/O βρίσκονται στη βιβλιοθήκη **Primitives/Pins**. Στο παρακάτω σχήμα Π.6 φαίνεται το κύκλωμα μας. Τα ονόματα των ακροδεκτών μπορούμε να τα ορίσουμε κατάλληλα κάνοντας διπλό κλικ πάνω στους ακροδέκτες.

Αφού σχεδιάσουμε το κύκλωμα, πρέπει να σώσουμε το αρχείο με το ίδιο όνομα που είχαμε δώσει στο Project, δηλαδή askhsh1.

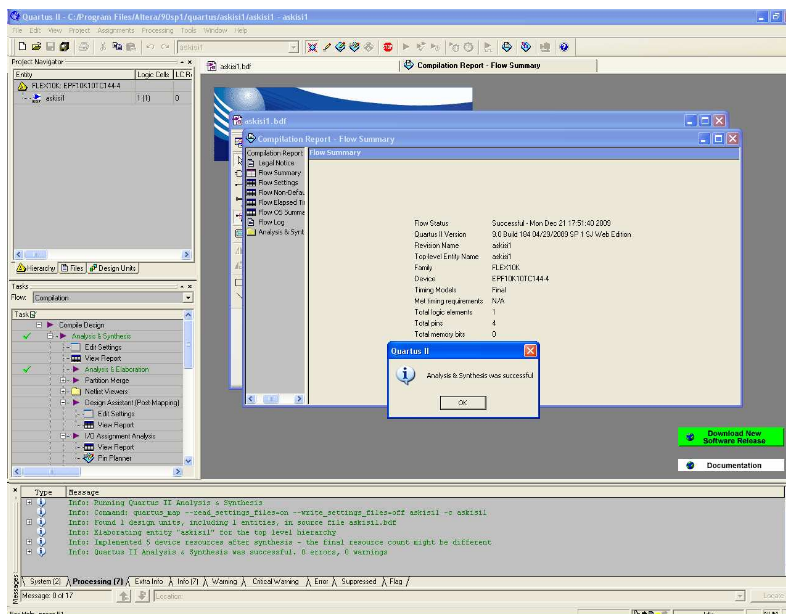


Σχήμα Π.6 Το παραπάνω κύκλωμα υλοποιεί τη Συνάρτηση $F=X_1X_2+X_2'X_3+X_1'X_3'$

Π.4 Ανάλυση και σύνθεση

Αφού έχουμε τελειώσει με την περιγραφή του κυκλώματος μας και το έχουμε αποθηκεύσει, ακολουθεί η διαδικασία της Ανάλυσης και Σύνθεσης. Η διαδικασία αυτή ξεκινά από το **Menu Processing/Start/Start Analysis & Synthesis**.

Μετά το τέλος της διαδικασίας, εμφανίζεται μια αναφορά, που μας ενημερώνει για τις απαιτήσεις σε λογικά στοιχεία (LEs), bits μνήμης και ακροδέκτες, που έχει η εφαρμογή μας, όπως φαίνεται στο σχήμα. Π.7.



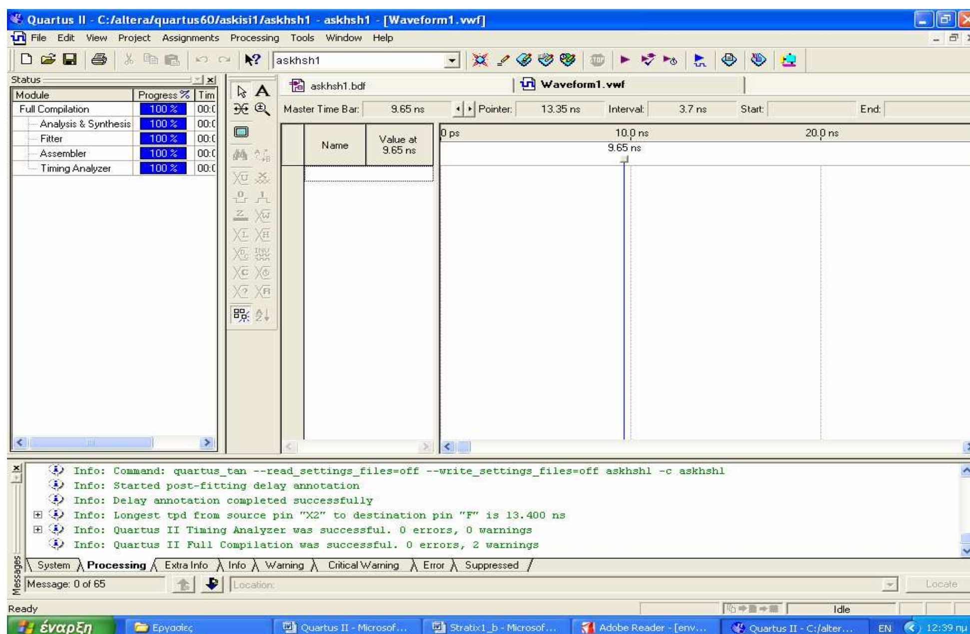
Σχήμα Π.7 Η Διαδικασία της Ανάλυσης & Σύνθεσης (Analysis & Synthesis)

Π.5 Η διαδικασία της προσομοίωσης

Όταν τελειώσει η διαδικασία της μετάφρασης επιτυχώς, ακολουθεί η διαδικασία της προσομοίωσης. Στην προσομοίωση θέλουμε να διαπιστώσουμε εάν το κύκλωμα που σχεδιάσαμε επαληθεύει τον επιθυμητό πίνακα αληθείας, ο οποίος για το συγκεκριμένο κύκλωμα είναι ο ακόλουθος:

X1	X2	X3	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Η διαδικασία αυτή γίνεται μέσα από τον Waveform Editor, με τον οποίο εισάγουμε τις κατάλληλες κυματομορφές εισόδου (waveform vectors) που είναι απαραίτητες για την προσομοίωση. Ο Waveform Editor καλείται από την εισαγωγική καρτέλα, επιλέγοντας **Menu File/New/Other Files/Vector Waveform File**. Στην επιφάνεια εργασίας του Quartus εμφανίζεται το παράθυρο που φαίνεται στο σχήμα Π.8. Κάνοντας διπλό κλικ κάτω από Name, εμφανίζεται το παράθυρο **Insert Node or Bus**. Στο παράθυρο αυτό πατούμε στο κουμπί **Node Finder** και στη συνέχεια επιλέγουμε **List** (προσέχοντας να έχουμε βάλει στην επιλογή Filter: Pins All). Επιλέγουμε τις εισόδους και εξόδους του κυκλώματος, των οποίων τη λειτουργία επιθυμούμε να προσομοιώσουμε.

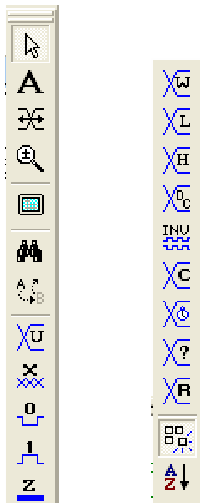


Σχήμα Π.8 Παράθυρο επεξεργαστή κυματομορφών πριν την εισαγωγή εισόδων/εξόδων

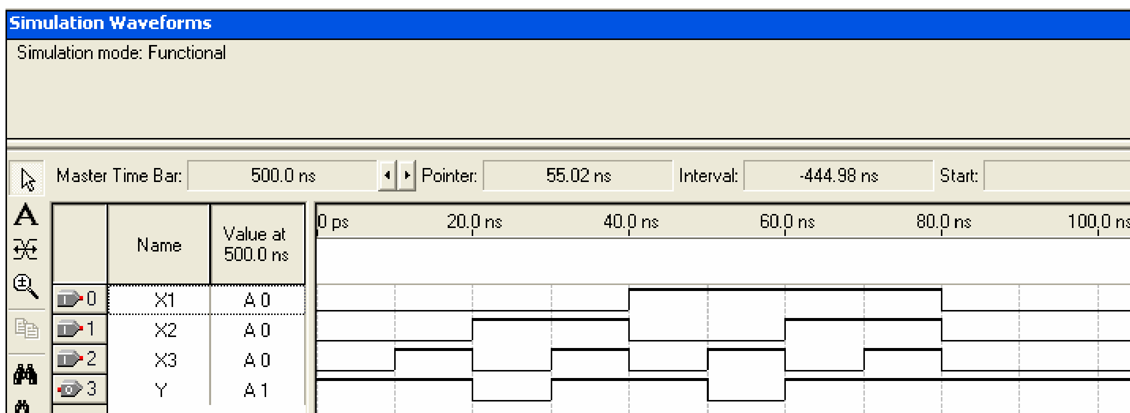
Μετά την ολοκλήρωση της παραπάνω διαδικασίας, στο παράθυρο του επεξεργαστή έχουν εμφανιστεί τα ονόματα των εισόδων και των εξόδων του κυκλώματος. Πριν θέσουμε τιμές στις εισόδους μας, θα πρέπει να επιλέξουμε **Menu Processing/Generate Functional Simulation Netlist** ώστε να δημιουργηθεί ένα αρχείο λειτουργικής περιγραφής του κυκλώματος. Επίσης από το **Menu Assignments/Settings/Simulator Settings** μπορούμε να διαλέξουμε αν η προσομοίωση θα είναι Timing ή Functional (χρονική ή λειτουργική). Επιλέγουμε **Functional**. Στην ίδια καρτέλα θέτουμε τη συνολική διάρκεια της προσομοίωσης στο 1μs. (**End Simulation at 1μs**).

Κατόπιν, χρησιμοποιώντας τα εργαλεία του Editor (σχ. Π.9), μπορούμε να δημιουργήσουμε τις κατάλληλες κυματομορφές εισόδου, θέτοντας τις τιμές των εισόδων σε 1 ή 0, για συγκεκριμένα χρονικά διαστήματα. Ακολουθήστε το υπόδειγμα του σχ. Π.10 για τις εισόδους. Αποθηκεύουμε το αρχείο των διανυσμάτων εισόδου (Waveform vector file) με όνομα askisi1.wvf.

Τέλος, πηγαίνοντας στο **Menu Processing/Start Simulation** εκτελούμε την προσομοίωση. Στο τέλος της προσομοίωσης, εμφανίζονται οι τιμές των εξόδων για τις τιμές των εισόδων που σχεδιάσαμε στα παραπάνω βήματα.



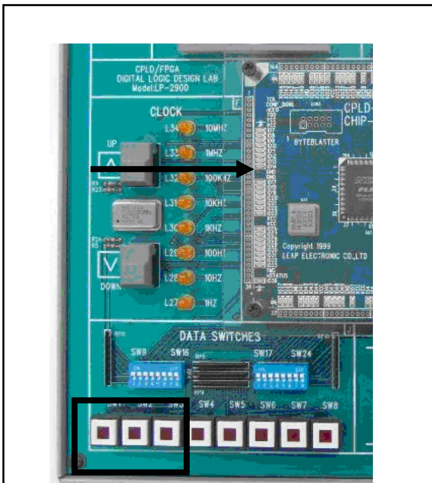
Σχήμα Π.9 Εργαλεία του Waveform Editor. Αφού επιλέξουμε ένα χρονικό διάστημα της εισόδου που επιθυμούμε να επεξεργαστούμε, θέτουμε την κυματομορφή εισόδου σε 1 ή 0 με τα αντίστοιχα εργαλεία.



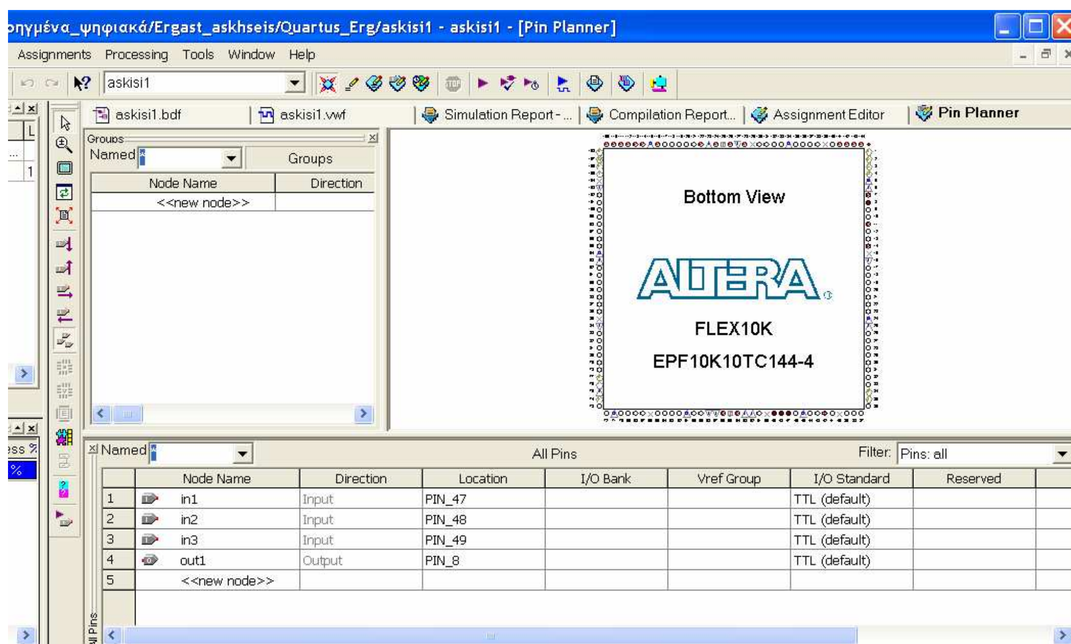
Σχήμα Π.10 Το παράθυρο της προσομοίωσης με τιμές εισόδων και εξόδων

Π.6 Ορισμός ακροδεκτών (pin assignments)

Το επόμενο βήμα είναι να αντιστοιχίσουμε τις εισόδους και τις εξόδους του κυκλώματός μας με συγκεκριμένους ακροδέκτες της διάταξης που πρόκειται να προγραμματίσουμε. Έχουμε ήδη ορίσει ότι η διάταξή μας είναι ένα FPGA που ανήκει στην οικογένεια FLEX10K της εταιρίας ALTERA και συγκεκριμένα η διάταξη EPF10K10TC144-4 (βλέπε στο **Menu Assignments/Device**). Προκειμένου να ορίσουμε σε ποιους ακροδέκτες του τσιπ θα συνδεθούν οι τρεις είσοδοι και η μία έξοδος του κυκλώματος που σχεδιάσαμε κάνουμε τις εξής επιλογές:



Επιλέγουμε **Menu Assignments/Pins**. Ανοίγει το παράθυρο του σχ. Π.11. Στο πεδίο Node Name εμφανίζονται τα ονόματα που έχουμε δώσει στους ακροδέκτες μας στο σχηματικό διάγραμμα. Διπλοπατώντας πάνω στο πεδίο Location εμφανίζεται η αρίθμηση όλων των ακροδεκτών της συγκεκριμένης διάταξης. Εκεί, κάνουμε τις επιλογές που φαίνονται στο σχήμα Π.11, δηλαδή οι είσοδοι in1, in2, in3 αντιστοιχίζονται στα pin 47, pin 48, pin 49 που αντιστοιχούν στους διακόπτες τύπου push-button SW1, SW2 και SW3, αντίστοιχα. Η έξοδος



Σχήμα Π.11 Παράθυρο Ορισμού Ακροδεκτών για το FLEX10K. Ορίστε τους ακροδέκτες με τον τρόπο που φαίνεται στο κάτω μέρος της εικόνας.

out1 αντιστοιχίζεται στο pin 8 που αντιστοιχεί σε ένα SMD led πάνω στην πλακέτα (Chip-Board) του FPGA. Οι επιλογές αυτές προκύπτουν από τα σχηματικά διαγράμματα του κυκλώματος που πρόκειται να προγραμματίσουμε.

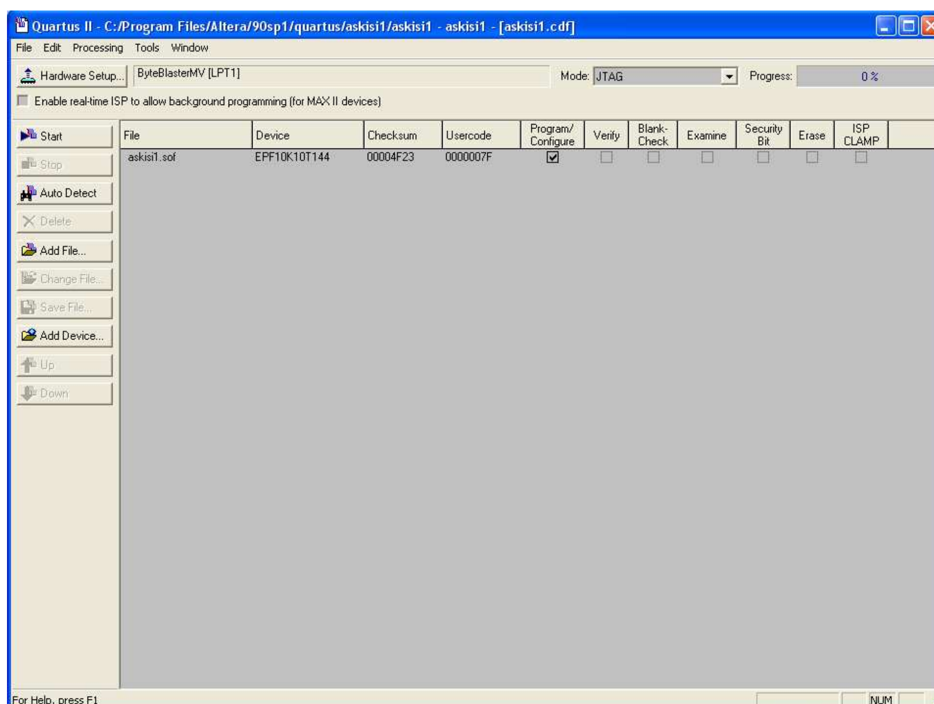
Εναλλακτικά, η αντιστοίχιση ενός pin του FPGA με μια είσοδο ή έξοδο μπορεί να γίνει διπλοπατώντας στο pin που θέλουμε να αντιστοιχίσουμε πάνω στην απεικόνιση του ολοκληρωμένου κυκλώματος, στο πάνω δεξιά πλαίσιο στο παράθυρο του ορισμού ακροδεκτών και συμπληρώνοντας τα κατάλληλα στοιχεία στη φόρμα που θα εμφανιστεί.

Π.7 Προγραμματισμός (διαμόρφωση) του κυκλώματος

Αφού ολοκληρωθεί ο ορισμός ακροδεκτών και πριν πραγματοποιηθεί η διαμόρφωση του FPGA θα πρέπει να γίνει Μετάφραση (Compilation) ώστε να δημιουργηθεί το αρχείο *.sof το οποίο περιέχει όποιες πληροφορίες χρειάζονται για την διαμόρφωση του FPGA. Η Μετάφραση γίνεται από το Menu **Processing/Start Compilation**. Υπενθυμίζουμε ότι η μετάφραση περιλαμβάνει την Ανάλυση & Σύνθεση, τη Δρομολόγηση, την Χρονική Ανάλυση και την παραγωγή των αρχείων προγραμματισμού του FPGA.

Μετά την επιτυχή ολοκλήρωση της Μετάφρασης μπορεί να γίνει η διαμόρφωση του FPGA με τον Programmer, ο οποίος βρίσκεται στο **Menu Tools/Programmer**. Το παράθυρο που εμφανίζεται είναι αυτό που ακολουθεί.

Την πρώτη φορά που θα χρησιμοποιήσουμε τον programmer θα πρέπει να κάνουμε τις κατάλληλες επιλογές του υλικού (Hardware Setup) για την επιλογή του κατάλληλου κυκλώματος προγραμματισμού. Συνήθεις επιλογές είναι ο οδηγός **BYTE-BLASTER** για προγραμματισμό μέσω της παράλληλης θύρας ή ο **USB-BLASTER** για προγραμματισμό μέσω της θύρας USB. Το αναπτυξιακό **LP-2900** που χρησιμοποιείται στο εργαστήριο χρησιμοποιεί τον οδηγό BYTE-BLASTER.



Σχήμα Π.12 Το παράθυρο του Programmer

Αν δεν είναι ήδη επιλεγμένο θα πρέπει να επιλέξουμε το τελικό αρχείο διαμόρφωσης (.sof) που δημιουργήθηκε κατά το τελικό στάδιο της μετάφρασης. Κατόπιν, επιλέγοντας Start αρχίζει η διαδικασία διαμόρφωσης, που συνήθως διαρκεί μερικά δευτερόλεπτα. Η πρόοδος της διαδικασίας εμφανίζεται στην μπλε μπάρα στα δεξιά της οθόνης.