61

ΑΤΕΙ ΚΑΛΑΜΑΤΑΣ
ΠΑΡΑΡΤΗΜΑ ΣΠΑΡΤΗΣ

ΤΜΗΜΑ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ
της φοιτήτριας:

ΓΕΩΡΓΟΥΔΑΚΗ ΑΛΕΞΑΝΔΡΑ του ΣΤΑΥΡΟΥ

Αριθμός Μητρώου: 2007138

Θέμα:

«ΣΧΕΔΙΑΣΗ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ ΜΕ VHDL: VERY LARGE SCALE INTEGRATION HARDWARE DESCRIPTION LANGUAGE»

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: ΛΙΑΠΕΡΔΟΣ ΙΩΑΝΝΗΣ, ΚΑΘΗΓΗΤΗΣ ΕΦΑΡΜΟΓΩΝ

Σπάρτη, 2011

ΠΕΡΙΕΧΟΜΕΝΑ

ΠΡΟΛΟΓΟΣ

Η παρούσα διπλωματική εργασία έχει ως αντικείμενο την παρουσίαση της γλώσσας περιγραφής υλικού VHDL, με την χρήση της οποίας γίνεται πολύ ευκολότερη η σχεδίαση ψηφιακών συστημάτων.

Στο κεφάλαιο 1 αναφέρεται μια σύντομη ιστορία της εξέλιξης της γλώσσας VHDL και παρουσιάζονται τα χαρακτηριστικά της.

Στο κεφάλαιο 2 περιγράφονται τα βασικά στοιχεία της γλώσσας, όπως οι τύποι δεδομένων, τα αντικείμενα, και τα διάφορα είδη δηλώσεων.

Το κεφάλαιο 3 παρουσιάζει το σχεδιασμό ψηφιακών συσκευών με βάση το μοντέλο συμπεριφοράς. Γίνεται λόγος για τις διάφορες διαδοχικές δηλώσεις που είναι διαθέσιμες σε αυτό το μοντέλο σχεδιασμού και εξηγείται πώς μπορούν να χρησιμοποιηθούν για την μοντελοποίηση της διαδοχική συμπεριφοράς ενός σχεδίου.

Το κεφάλαιο 4 περιγράφει σχεδιασμό ψηφιακών συσκευών με βάση το μοντέλο ροής δεδομένων. Περιγράφει τις δηλώσεις ανάθεσης παράλληλων σημάτων, και παρέχει παραδείγματα για να δείξει πώς η παράλληλη συμπεριφορά ενός σχεδίου μπορεί να σχεδιαστεί με την χρήση αυτών των δηλώσεων.

Το κεφάλαιο 5 παρουσιάζει σχεδιασμό ψηφιακών συσκευών με βάση το δομικό μοντέλο. Σε αυτό το στυλ μοντελοποίησης, ένα σχέδιο εκφράζεται ως ένα σύνολο διασυνδεδεμένων στοιχείων, ενδεχομένως σε μια ιεραρχία.

Στο κεφάλαιο 6 παρουσιάζεται ο σχεδιασμός ψηφιακών συσκευών χρησιμοποιώντας ένα συνδυασμό των τριών μοντέλων που αναφέρθηκαν στα κεφάλαια 3, 4 και 5.

Το κεφάλαιο 7 περιγράφει τα πακέτα και τις βιβλιοθήκες, όπως ορίζονατι από τη γλώσσα.

Σε όλες τις περιγραφές VHDL που εμφανίζονται, οι δεσμευμένες λέξεις εμφανίζονται με έντονα γράμματα. Ένας πλήρης κατάλογος των δεσμευμένων λέξεων, καθώς και ο πηγαίος κώδικας των πακέτων STANDARD και TEXTIO, παρουσιάζονται στο Παράρτημα Α. Τέλος, η πλήρης γραμματική της γλώσσα παρέχεται στο Παράρτημα Β.

Στο σημείο αυτό θέλω να ευχαριστήσω τον επιβλέποντα καθηγητή της διπλωματικής μου εργασίας, κ. Ιωάννη Λιαπέρδο, καθηγητή Εφαρμογών, που με ανέλαβε και με βοήθησε για την ολοκλήρωση της εργασίας μου.

ΚΕΦΑΛΑΙΟ 1 – ΕΙΣΑΓΩΓΗ

Το κεφάλαιο αυτό παρέχει μια σύντομη ιστορία της εξέλιξης της VHDL και περιγράφει τις μεγάλες δυνατότητες που τη διαφοροποιούν από τις άλλες γλώσσες περιγραφής υλικού. Το κεφάλαιο αυτό εξηγεί επίσης εν συντομία την ιδέα μιας οντότητας.

1.1. ΤΙ ΕΙΝΑΙ Η VHDL

Η VHDL είναι μια γλώσσα περιγραφής υλικού που μπορεί να χρησιμοποιηθεί για την ανάπτυξη ολοκληρωμένων ψηφιακών κυκλωμάτων και συστημάτων. Περιέχει στοιχεία που μπορούν να χρησιμοποιηθούν για να περιγράψουν τη συμπεριφορά ή τη δομή ενός ψηφιακού συστήματος, με τη δυνατότητα του καθορισμού ενός ρητού χρονοδιαγράμματος.

Η VHDL είναι μια μεγάλη και λεπτομερής γλώσσα με πολλές πολύπλοκες δομές, που έχουν πολύπλοκες σημειολογικές έννοιες και είναι δύσκολο να την κατανοήσουμε αρχικά. Ωστόσο, είναι δυνατόν να καταλάβουμε γρήγορα ένα υποσύνολο της VHDL το οποίο είναι και απλό και εύκολο στη χρήση και μας δίνει την δυνατότητα να σχεδιάσουμε μοντέλα μεγάλης πολυπλοκότητας.

Συχνά αναφέρεται ότι η VHDL είναι αρκτικόλεξο των λέξεων Very Hard Description Language – Πολύ Δύσκολη Γλώσσα Περιγραφής. Στην πραγματικότητα, είναι το ακρώνυμο των λέξεων VHSIC Hardware Description Language (όπου VHSIC είναι η συντόμευση των λέξεων Very High Speed Integrated Circuits – Ολοκληρωμένα Κυκλώματα Υψηλής Ταχύτητας).

Το ψηφιακό σύστημα και τα υποσυστήματά του μπορούν να περιγραφούν σε οποιοδήποτε επίπεδο αφαιρετικότητας, από το επίπεδο αρχιτεκτονικής μέχρι και το επίπεδο πύλης. Η πολυπλοκότητα του ψηφιακού συστήματος που σχεδιάζεται μπορεί να ποικίλλει από αυτή μιας απλής πύλης μέχρι και αυτήν ενός πλήρους ψηφιακού ηλεκτρονικού συστήματος, ή οτιδήποτε μεταξύ αυτών των δύο. Το ψηφιακό σύστημα μπορεί επίσης να περιγραφεί ιεραρχικά και

υποστηρίζονται οι μεθοδολογίες σχεδιασμού top-down και bottom-up. Το χρονοδιάγραμμα μπορεί επίσης να συμπεριληφθεί στην ίδια περιγραφή.

Ακριβείς έννοιες προσομοίωσης σχετίζονται με όλες τις γλωσσικές δομές, και ως εκ τούτου, τα μοντέλα γραμμένα σε αυτή τη γλώσσα μπορούν να επαληθευτούν με τη χρήση προσομοιωτή VHDL.

1.2. ΙΣΤΟΡΙΑ

Η ανάγκη για τη γλώσσα εμφανίστηκε για πρώτη φορά το 1981 υπό το πρόγραμμα VHSIC. Σε αυτό το πρόγραμμα, συμμετείχαν πολλές αμερικανικές εταιρείες για το σχεδιασμό VHSIC chips για το Υπουργείο Άμυνας (DoD). Εκείνη την εποχή, οι περισσότερες εταιρείες χρησιμοποιούν διαφορετικές γλώσσες περιγραφής υλικού για την περιγραφή και την ανάπτυξη των ολοκληρωμένων κυκλωμάτων τους, με αποτέλεσμα, οι διάφοροι προμηθευτές να μην μπορούν να ανταλλάξουν αποτελεσματικά σχέδια μεταξύ τους. Επίσης, διάφοροι προμηθευτές παρείχαν το Υπουργείου Άμυνας με τις περιγραφές των chips τους σε διάφορες γλώσσες περιγραφής υλικού. Η επαναχρησιμοποίηση ήταν επίσης ένα μεγάλο ζήτημα. Έτσι, προήλθε η ανάγκη για μια τυποποιημένη γλώσσα περιγραφής υλικού για το σχεδιασμό, την τεκμηρίωση, και την επαλήθευση των ψηφιακών συστημάτων.

Μια ομάδα από τις τρεις εταιρείες, η IBM, η Texas Instruments, και η Intermetrics, απονεμήθηκαν πρώτες το συμβόλαιο με το Υπουργείο Άμυνας για την ανάπτυξη μιας έκδοσης της γλώσσας το 1983. Η έκδοση 7.2 της VHDL αναπτύχθηκε και δημοσιεύθηκε στο κοινό το 1985. Υπήρξε μια ισχυρή συμμετοχή της βιομηχανίας σε όλη τη διαδικασία ανάπτυξης της γλώσσας VHDL, ιδίως από τις εταιρείες που ανέπτυσσαν VHSIC chips. Μετά την κυκλοφορία της έκδοσης 7.2, υπήρξε μια αυξανόμενη ανάγκη να καταστεί η γλώσσα ένα ευρέως διαδεδομένο πρότυπο. Κατά συνέπεια, η γλώσσα μεταφέρθηκε στο IEEE[1] για την τυποποίησή της, το 1986. Μετά από μια σημαντική ενίσχυση της γλώσσας, από μια ομάδα εκπροσώπων της βιομηχανίας, των πανεπιστημίων και του Υπουργείου Άμυνας, η γλώσσα τυποποιήθηκε από το IEEE τον Δεκέμβριο του 1987. Αυτή η εκδοχή της γλώσσας είναι τώρα γνωστή ως IEEE Std 1076-1987. Η επίσημη περιγραφή της γλώσσας εμφανίζεται στο εγχειρίδιο αναφοράς του πρότυπου IEEE της γλώσσας VHDL (IEEE Standard VHDL Language Reference Manual) που

[1] **IEEE**: Institute of Electrical and Electronics Engineers (Ινστιτούτο Ηλεκτρολόγων και Ηλεκτρονικών Μηχανικών)

διατίθεται από την IEEE. Έκτοτε η γλώσσα έχει επίσης αναγνωριστεί και ως πρότυπο από το Αμερικανικό Εθνικό Ινστιτούτο Προτύπων (ANSI[2]).

Το Υπουργείο Άμυνας, από τον Σεπτέμβριο του 1988, απαιτεί από όλους τους προμηθευτές των ψηφιακών ολοκληρωμένων κυκλωμάτων συγκεκριμένης εφαρμογής (ASIC[3]) να παρέχουν περιγραφές VHDL των ASICs και των επιμέρους εξαρτημάτων τους, τόσο σε επίπεδο συμπεριφοράς όσο και σε επίπεδο δομής. Οι εγκαταστάσεις δοκιμών (test benches) που χρησιμοποιούνται για την επικύρωση των ASIC chips σε όλα τα επίπεδα της ιεραρχίας της, πρέπει επίσης να παραδοθούν σε VHDL. Αυτό το σύνολο των κυβερνητικών απαιτήσεων περιγράφεται στο στρατιωτικό πρότυπο 454.

1.3. ΧΑΡΑΚΤΗΡΙΣΤΙΚΑ ΤΗΣ VHDL

Παρακάτω αναγράφονται οι μεγάλες δυνατότητες που παρέχει η γλώσσα, μαζί με τα χαρακτηριστικά που τη διαφοροποιούν από τις άλλες γλώσσες περιγραφής υλικού.

- **Η γλώσσα VHDL μπορεί να χρησιμοποιηθεί ως μέσο ανταλλαγής μεταξύ των προμηθευτών chip και των χρηστών εργαλείων CAD[4].** Διάφοροι προμηθευτές chip μπορούν να παρέχουν περιγραφές VHDL των δομικών μονάδων τους στους σχεδιαστές συστημάτων. Οι χρήστες των εργαλείων CAD μπορούν να το χρησιμοποιήσουν για να περιγράψουν τη συμπεριφορά του σχεδίου σε υψηλό επίπεδο αφαιρετικότητας για λειτουργική προσομοίωση.

- **Η γλώσσα μπορεί επίσης να χρησιμοποιηθεί ως μέσο επικοινωνίας μεταξύ των διαφόρων CAD και CAE[5] εργαλείων,** για παράδειγμα, ένα πρόγραμμα καταγραφής σχημάτων μπορεί να χρησιμοποιηθεί για να δημιουργήσει μια περιγραφή VHDL του σχεδίου η οποία στην συνέχεια μπορεί να χρησιμοποιηθεί ως είσοδο σε ένα πρόγραμμα προσομοίωσης.

- **Η γλώσσα υποστηρίζει την ιεραρχία**, δηλαδή, ένα ψηφιακό σύστημα μπορεί να μοντελοποιηθεί ως ένα σύνολο διασυνδεδεμένων δομικών στοιχείων. Κάθε δομικό στοιχείο, με τη σειρά του, μπορεί να μοντελοποιηθεί ως ένα σύνολο διασυνδεδεμένων υπο-στοιχείων.

---

[2] **ANSI**: American National Standards Institute (Αμερικανικό Εθνικό Ινστιτούτο Προτύπων)
[3] **ASIC**: Application-Specific Integrated Circuit (Ολοκληρωμένο Κύκλωμα Συγκεκριμένης Εφαρμογής)
[4] **CAD:** Computer Aided Design
[5] **CAE**: Computer Aided Engineering

- **Η γλώσσα υποστηρίζει ευέλικτες μεθοδολογίες σχεδιασμού**: top-down, bottom-up, ή ένας συνδυασμός αυτών.

- **Η γλώσσα δεν χρειάζεται συγκεκριμένη τεχνολογία,** αλλά είναι σε θέση να υποστηρίζει διάφορες τεχνολογίες υλικού. Για παράδειγμα, μπορεί να ορίσει νέους λογικούς τύπους και νέα συστατικά, καθώς επίσης και να προσδιορίζει σε συγκεκριμένες τεχνολογίες, χαρακτηριστικά. Με το να είναι ανεξάρτητη της τεχνολογίας, το ίδιο μοντέλο συμπεριφοράς μπορεί να συντεθεί σε βιβλιοθήκες διαφόρων κατασκευαστών.

- **Υποστηρίζει τόσο σύγχρονα όσο και ασύγχρονα μοντέλα χρονισμού.**

- **Η γλώσσα είναι διαθέσιμη στο κοινό, αναγνώσιμη από τον άνθρωπο, αναγνώσιμη από μηχανήματα, και πάνω απ 'όλα, δεν είναι ιδιόκτητη.**

- **Πρόκειται για ένα πρότυπο του IEEE και του ANSI**, και ως εκ τούτου, **τα μοντέλα που περιγράφονται με χρήση αυτής της γλώσσας είναι φορητά**.

- **Η γλώσσα υποστηρίζει τρία βασικά διαφορετικά στυλ περιγραφής: το δομικό (structural), της ροής δεδομένων (dataflow), και της συμπεριφοράς (behavioral)**. Ένα σχέδιο μπορεί επίσης να εκφραστεί με οποιοδήποτε συνδυασμό αυτών των τριών περιγραφικών στυλ.

- **Υποστηρίζει ένα ευρύ φάσμα από τα επίπεδα αφαιρετικότητας**, που κυμαίνονται από αφηρημένες περιγραφές συμπεριφοράς έως και πολύ συγκεκριμένες περιγραφές στο επίπεδο πυλών. Ωστόσο, δεν υποστηρίζει την μοντελοποίηση στο επίπεδο ή κάτω από το επίπεδο του τρανζίστορ.

- **Αυθαίρετα μεγάλα σχέδια μπορούν να μοντελοποιηθούν χρησιμοποιώντας τη γλώσσα και δεν υπάρχουν περιορισμοί που επιβάλλονται από τη γλώσσα για το μέγεθος ενός σχεδίου.**

- **Η γλώσσα έχει στοιχεία που κάνουν την μοντελοποίηση μεγάλης κλίμακας σχεδίων πιο εύκολη**, για παράδειγμα, τα δομικά στοιχεία (components), τις λειτουργίες/συναρτήσεις (functions), τις διαδικασίες (procedures) και τα πακέτα (packages).

- **Δεν είναι αναγκαίο να μάθει κάποιος μια διαφορετική γλώσσα για τον έλεγχο της προσομοίωσης**. Τα προγράμματα δοκιμών (test benches) μπορούν να γραφτούν χρησιμοποιώντας την ίδια γλώσσα, για να ελέγξουν τα άλλα μοντέλα VHDL.

- **Η χρήση γενικεύσεων και χαρακτηριστικών στα μοντέλα** διευκολύνει τις υποσημειώσεις των στατικών πληροφοριών, όπως το χρονοδιάγραμμα, ή πληροφοριών τοποθέτησης.

- **Οι γενικεύσεις και τα χαρακτηριστικά είναι επίσης χρήσιμα για την περιγραφή παραμετρικών σχεδίων.**

- **Ένα μοντέλο δεν μπορεί να περιγράψει μόνο τη λειτουργικότητα του σχεδίου,** αλλά μπορεί επίσης να περιέχει πληροφορίες σχετικά με το ίδιο το σχέδιο όσον αφορά τα οριζόμενα από το χρήστη χαρακτηριστικά, για παράδειγμα, τη συνολική έκταση και την ταχύτητα.

- **Μια κοινή γλώσσα μπορεί να χρησιμοποιείται για να περιγράψει στοιχεία βιβλιοθηκών από διαφορετικούς κατασκευαστές.** Εργαλεία που καταλαβαίνουν VHDL μοντέλα δεν θα έχουν καμία δυσκολία στην ανάγνωση μοντέλων από μια ποικιλία κατασκευαστών εφόσον η γλώσσα είναι ένα πρότυπο.

- **Μοντέλα γραμμένα σε αυτή τη γλώσσα μπορούν να επαληθευτούν μέσω της προσομοίωσης,** δεδομένου ότι οι ακριβείς έννοιες προσομοίωσης ορίζονται για κάθε κατασκεύασμα της γλώσσας.

- **Μοντέλα συμπεριφοράς τα οποία συμμορφώνονται με ένα συγκεκριμένο στυλ συνθετικής περιγραφής είναι δυνατόν να συντεθούν με περιγραφές σε επίπεδο πυλών.**

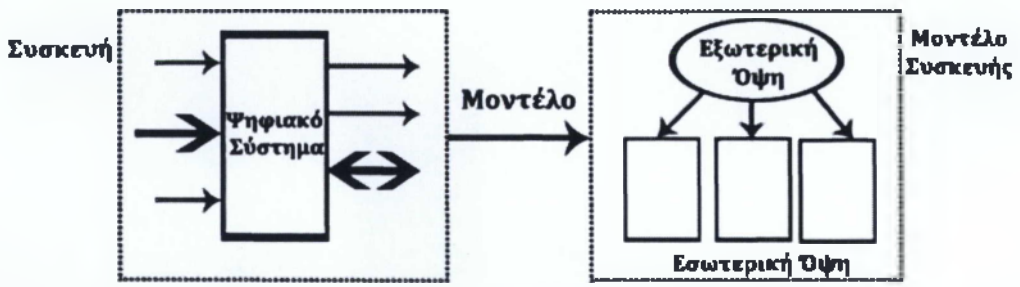
- **Η δυνατότητα καθορισμού νέων τύπων δεδομένων παρέχει τη δυνατότητα της περιγραφής και της προσομοίωσης μιας νέας τεχνικής σχεδιασμού σε πολύ υψηλό επίπεδο αφαιρετικότητας**, χωρίς καμία ανησυχία σχετικά με τις λεπτομέρειες εφαρμογής.

1.4. ΑΦΑΙΡΕΤΙΚΟΤΗΤΑ ΥΛΙΚΟΥ

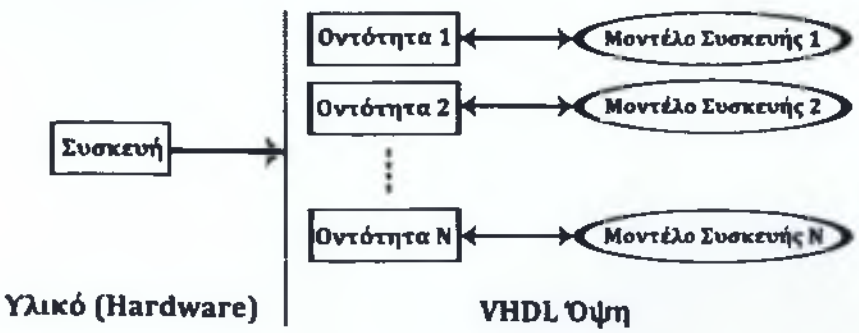
Η VHDL χρησιμοποιείται για να περιγράψει ένα μοντέλο για μια ψηφιακή συσκευή. Το μοντέλο αυτό καθορίζει την εξωτερική όψη της συσκευής και μια ή περισσότερες εσωτερικές όψεις. Η εσωτερική όψη της συσκευής καθορίζει τη λειτουργικότητα ή τη δομή, ενώ η εξωτερική άποψη καθορίζει τη διεπαφή της συσκευής μέσω της οποίας επικοινωνεί με τα άλλα μοντέλα στο περιβάλλον του. Το σχήμα 1.1 δείχνει τη συσκευή (hardware) και το αντίστοιχο μοντέλο λογισμικού.

Η χαρτογράφησης της σχέσης συσκευής προς μοντέλο συσκευής είναι αυστηρά ένα προς πολλά. Δηλαδή, μια συσκευή υλικού μπορεί να έχει πολλά μοντέλα συσκευών. Για παράδειγμα, μια συσκευή η οποία σχεδιάστηκε σε υψηλό επίπεδο αφαιρετικότητας μπορεί να μην έχει ένα ρολόι (clock) ως μία από τις εισόδους της, εφόσον το ρολόι μπορεί να μην χρησιμοποιήθηκε στην περιγραφή. Επίσης, η μεταφορά δεδομένων στη διεπαφή μπορεί να αντιμετωπιστεί από την άποψη ας πούμε, ακέραιων τιμών, αντί για λογικές τιμές.

Στην VHDL, κάθε μοντέλο συσκευής, αντιμετωπίζεται ως μια ξεχωριστή αναπαράσταση μιας μοναδικής συσκευής, που ονομάζεται μια οντότητα. Το σχήμα 1.2 παρουσιάζει την VHDL όψη μίας συσκευής που έχει πολλαπλά μοντέλα συσκευών, όπου κάθε μοντέλο συσκευής αντιπροσωπεύει μία οντότητα. Αν και η οντότητες 1 έως N αντιπροσωπεύουν N διαφορετικές οντότητες από την VHDL όψη, στην πραγματικότητα αντιπροσωπεύουν την ίδια συσκευή.



Σχήμα 1.1 Διαφορά Συσκευής και Μοντέλου συσκευής



Σχήμα 1.2 VHDL όψη συσκευής

Έτσι, η οντότητα είναι μια αφαιρετικότητα του υλικού της πραγματικής συσκευής. Κάθε οντότητα περιγράφεται με ένα μοντέλο που περιέχει μία εξωτερική όψη και μία ή περισσότερες εσωτερικές όψεις. Παράλληλα, μια συσκευή μπορεί να αντιπροσωπεύεται από μια ή περισσότερες οντότητες.

ΚΕΦΑΛΑΙΟ 2 – Βασικά Στοιχεία Γλώσσας

Αυτό το κεφάλαιο περιγράφει τα βασικά στοιχεία της γλώσσας. Αυτά περιλαμβάνουν τα αντικείμενα δεδομένων (data objects) που παίρνουν τιμές ενός συγκεκριμένου τύπου, τα λεκτικά (literals) που αντιπροσωπεύουν σταθερές τιμές, και τους τελεστές (operators) που εφαρμόζονται σε αντικείμενα δεδομένων. Κάθε αντικείμενο δεδομένων ανήκει σε έναν συγκεκριμένο τύπο. Οι διάφορες κατηγορίες τύπων και η σύνταξη για τον καθορισμό των τύπων οι οποίοι είναι ορισμένοι από το χρήστη (user-defined types) αναφέρονται εδώ. Το κεφάλαιο περιγράφει επίσης πώς μπορεί κάποιος να συνδέσει τους τύπους με τα αντικείμενα, χρησιμοποιώντας δηλώσεις αντικειμένων (object declarations).

Είναι σημαντικό να κατανοήσουμε την έννοια των τύπων δεδομένων και των αντικειμένων καθώς η VHDL είναι μια αυστηρά συντεταγμένη (strongly-typed) γλώσσα. Αυτό σημαίνει ότι οι πράξεις και οι αναθέσεις είναι εφικτές στη γλώσσα μόνο εάν ο τύπος των τελεστών και του αποτελέσματος ταιριάζουν, σύμφωνα με ένα σύνολο κανόνων. Ένα παράδειγμα λανθασμένων ενεργειών είναι η πρόσθεση ενός πραγματικού αριθμού με έναν ακέραιο αριθμό. Είναι, επομένως, σημαντικό να γίνει κατανοητό τι είναι οι τύποι και πώς μπορούν να χρησιμοποιηθούν σωστά στη γλώσσα.

2.1. ΑΝΑΓΝΩΡΙΣΤΙΚΑ (IDENTIFIERS)

Στην δήλωση μιας οντότητας, η ίδια η οντότητα και καθεμιά από τις θύρες ονομάζονται με κάποιο αναγνωριστικό (identifier). Ένα αναγνωριστικό στην VHDL αποτελείται από μια ακολουθία ενός ή περισσότερων χαρακτήρων. Ένας χαρακτήρας μπορεί να είναι ένα κεφαλαίο ή ένα μικρό γράμμα (A...Z, α...z), ένα ψηφίο (0...9) ή ο χαρακτήρας υπογράμμισης (_) (underscore). Ο πρώτος χαρακτήρας σε ένα αναγνωριστικό πρέπει να είναι γράμμα και ο τελευταίος χαρακτήρας δεν μπορεί να είναι ο χαρακτήρας υπογράμμισης. Τα πεζά και κεφαλαία γράμματα θεωρούνται πανομοιότυπα όταν χρησιμοποιούνται σε ένα αναγνωριστικό. Για παράδειγμα τα ακόλουθα: Count, COUNT, και CouNT, όλα αναφέρονται στο ίδιο αναγνωριστικό. Επίσης, δύο χαρακτήρες υπογράμμισης δεν μπορούν να εμφανίζονται διαδοχικά.

Τα σχόλια σε μια περιγραφή πρέπει να προηγούνται από δύο διαδοχικές παύλες (-). Να σημειωθεί ότι τα σχόλια εκτείνονται μέχρι το τέλος της γραμμής και μπορούν να εμφανιστούν οπουδήποτε μέσα σε ένα περιγραφή.

Παραδείγματα:

-- Αυτό είναι ένα σχόλιο.

entity AB_1 **is end;** -- Αυτό το σχόλιο ξεκινά μετά από μια δήλωση οντότητας.

Η γλώσσα ορίζει ένα σύνολο δεσμευμένων λέξεων (βλ. παράρτημα Α.1.) Αυτές οι λέξεις, γνωστές επίσης και ως λέξεις κλειδιά, έχουν μια συγκεκριμένη σημασία στη γλώσσα, και ως εκ τούτου, δεν μπορούν να χρησιμοποιηθούν ως αναγνωριστικά.

2.2. ΑΝΤΙΚΕΙΜΕΝΑ ΔΕΔΟΜΕΝΩΝ (DATA OBJECTS)

Ένα αντικείμενο δεδομένων έχει μια τιμή συγκεκριμένου τύπου και δημιουργείται με τη βοήθεια μιας δήλωσης αντικειμένου. Για παράδειγμα:

variable COUNT: INTEGER;

Το παραπάνω παράδειγμα έχει ως αποτέλεσμα τη δημιουργία ενός αντικειμένου δεδομένων που ονομάζεται COUNT το οποίο μπορεί να κρατήσει ακέραιες τιμές. Το αντικείμενο COUNT είναι επίσης δηλωμένο ως μεταβλητή (variable). Κάθε αντικείμενο δεδομένων ανήκει σε μία από τις ακόλουθες τρεις κατηγορίες:

1. *Σταθερές (constants):* Ένα αντικείμενο που ανήκει στην κατηγορίας των σταθερών μπορεί να κρατήσει μια μοναδική τιμή συγκεκριμένου τύπου. Αυτή η τιμή αποδίδεται στο αντικείμενο πριν ξεκινήσει η προσομοίωση και δεν μπορεί να αλλάξει κατά τη διάρκεια της προσομοίωσης.
2. *Μεταβλητές (variables):* Ένα αντικείμενο που ανήκει στην κατηγορίας των μεταβλητών μπορεί επίσης να κρατήσετε μία μόνο τιμή ενός συγκεκριμένου τύπου. Ωστόσο, στην περίπτωση αυτή, διαφορετικές τιμές μπορούν να ανατεθούν στο αντικείμενο σε διαφορετικές χρονικές στιγμές χρησιμοποιώντας μια δήλωση ανάθεσης μεταβλητών (variable assignment statement).
3. *Σήματα (signals):* Ένα αντικείμενο που ανήκει στην κατηγορίας των σημάτων έχει ένα ιστορικό από τιμές, μια τρέχουσα τιμή, και μια σειρά από μελλοντικές τιμές. Οι μελλοντικές τιμές που μπορούν να ανατεθούν σε ένα τέτοιο αντικείμενο χρησιμοποιώντας μια δήλωση ανάθεσης σήματος (signal assignment statement).

Τα σήματα μπορούν να θεωρηθούν ως σύρματα σε ένα κύκλωμα, ενώ οι μεταβλητές και οι σταθερές είναι ανάλογες με τις αντίστοιχές έννοιες σε μια

υψηλού επιπέδου γλώσσα προγραμματισμού όπως η C ή Pascal. Τα σήματα χρησιμοποιούνται συνήθως για των σχεδιασμό καλωδίων και flip-flops, ενώ οι μεταβλητές και οι σταθερές συνήθως χρησιμοποιούνται για να περιγράψουν τη συμπεριφορά του κυκλώματος.

Μια δήλωση αντικειμένου χρησιμοποιείται για να δηλωθεί ένα αντικείμενο, ο τύπος του, η κατηγορία του, και προαιρετικά να του ανατεθεί μιας τιμή.

Παρακάτω ακολουθούν μερικά παραδείγματα των δηλώσεων αντικείμενών διαφόρων τύπων και κατηγοριών.

2.2.1. ΔΗΛΩΣΕΙΣ ΣΤΑΘΕΡΩΝ (CONSTANT DECLARATIONS)

Η δήλωση σταθερών στην VHDL γίνεται ως εξής:

constant όνομα_αντικειμένου: τύπος: = τιμή;

Παραδείγματα:

constant RISE_TIME: TIME: = 10ns;
constant BUS_WIDTH: INTEGER: = 8;

Η πρώτη δήλωση δηλώνει το αντικείμενο RISE_TIME στο οποίο μπορεί να ανατεθεί τιμή τύπου TIME (ενός προκαθορισμένου τύπου στη γλώσσα VHDL) και η τιμή που αποδόθηκε στο αντικείμενο κατά την έναρξη της προσομοίωσης είναι 10ns. Η δεύτερη δήλωση δηλώνει μια σταθερά BUS_WIDTH τύπου integer (ακέραιου) με αξία 8.

Ένα παράδειγμα μιας άλλης μορφής δήλωσης σταθερών είναι το εξής:

constant NO_OF_INPUTS: INTEGER;

Σε αυτή την περίπτωση η τιμή της σταθεράς δεν έχει καθορισθεί. Μια τέτοια σταθερά ονομάζεται «αναβαλλόμενη σταθερά» και μπορεί να εμφανιστεί μόνο μέσα σε μια δήλωση πακέτο (package declaration). Η πλήρης δήλωση της σταθεράς με την τιμή της πρέπει να αναγράφονται στο αντίστοιχο κομμάτι του κώδικα του πακέτου (package body).

2.2.2. ΔΗΛΩΣΕΙΣ ΜΕΤΑΒΛΗΤΩΝ (VARIABLE DECLARATIONS)

Η δήλωση μεταβλητών στην VHDL γίνεται ως εξής:

variable όνομα_αντικειμένου: τύπος: [= αρχική τιμή];

Παραδείγματα:

variable CTRL_STATUS: BIT_VECTOR(10 **downto** 0);
variable SUM: INTEGER **range** 0 **to** 100: = 10;
variable FOUND, DONE: BOOLEAN;

Η πρώτη δήλωση προσδιορίζει μια μεταβλητή CTRL_STATUS ως ένας πίνακας από έντεκα στοιχεία τύπου BIT. Η αρχική τιμή για όλα τα στοιχεία του πίνακα του αντικειμένου CTRL_STATUS είναι '0'.

Στη δεύτερη δήλωση, έχει οριστεί μια ρητή αρχική τιμή στη μεταβλητή SUM. Στην αρχή της προσομοίωση, η μεταβλητή SUM θα έχει αρχική τιμή 10. Εάν δεν έχει οριστεί αρχική τιμή για τη μεταβλητή, μια προκαθορισμένη (default) τιμή χρησιμοποιείται ως αρχική τιμή.

Στην τρίτη δήλωση, η αρχική τιμή που αποδίδεται στις μεταβλητές FOUND και DONE στην αρχή της προσομοίωσης είναι η FALSE (FALSE είναι η πιο αριστερή αξία του προκαθορισμένου τύπου, Boolean).

2.2.3. ΔΗΛΩΣΕΙΣ ΣΗΜΑΤΩΝ (SIGNAL DECLARATIONS)

Η δήλωση σημάτων στην VHDL γίνεται ως εξής:

signal όνομα_αντικειμένου: τύπος: [= αρχική τιμή];

Παραδείγματα:

signal CLOCK: BIT;
signal GATE_DELAY: TIME := 10 ns;

Η ερμηνεία αυτών των δηλώσεων σημάτων είναι πολύ παρόμοια με εκείνη των δηλώσεων των μεταβλητών. Η πρώτη δήλωση δηλώνει το σήμα CLOCK τύπου BIT και του δίνει μια αρχική τιμή '0' ('0' είναι η πιο αριστερή τιμή του τύπου BIT). Η δεύτερη δήλωση δηλώνει ένα σήμα GATE_DELAY τύπου TIME που έχει μια αρχική τιμή 10ns.

2.2.3.1. ΤΥΠΟΙ ΣΗΜΑΤΩΝ (SIGNAL TYPES)

Παρακάτω αναλύονται οι ακόλουθοι εννέα τύποι σήματος: **BIT, BIT_VECTOR, STD_LOGIC, STD_LOGIC_VECTOR, SIGNED, UNSIGNED, INTEGER, ENUMERATION** και **BOOLEAN.**

2.2.3.1.1. BIT - BIT_VECTOR

Ένα αντικείμενο τύπου **BIT** μπορεί να έχει την τιμή 0 ή 1 ενώ ένα αντικείμενο τύπου **BIT_VECTOR** είναι στην ουσία ένας πίνακας δυαδικών ψηφίων 0 και 1.

Παραδείγματα:

```
signal CLOCK: BIT;
signal GND: BIT: = '0';
signal DATABUS: BIT_VECTOR (15 downto 0);
signal ADDRBUS: BIT_VECTOR (0 to 31);
```

Η χρήση του **downto** σε ένα πίνακα υποδηλώνει ότι το πιο σημαντικό bit ορίζεται με τη βοήθεια του υψηλότερου δείκτη και το λιγότερο σημαντικό bit ορίζεται με τη βοήθεια του χαμηλότερου δείκτη. Η χρήση του **to** σε ένα πίνακα υποδηλώνει ότι το πιο σημαντικό bit ορίζεται με τη βοήθεια του χαμηλότερου δείκτη και το λιγότερο σημαντικό bit ορίζεται με τη βοήθεια του υψηλότερου δείκτη.

Παράδειγμα:

(i) **signal** DATABUS: BIT_VECTOR (4 **downto** 1);
(ii) **signal** DATABUS: BIT_VECTOR (1 **to** 4);

Έστω ότι DATABUS <= "1000" τότε στην περίπτωση (i) η θέση DATABUS(1) έχει την τιμή 0 ενώ στην περίπτωση (ii) η θέση DATABUS(1) έχει την τιμή 1 και η θέση DATABUS(4) θα έχει την τιμή 0.

2.2.3.1.2. STD_LOGIC – STD_LOGIC_VECTOR

Ένα αντικείμενο τύπου **STD_LOGIC** μπορεί να έχει την τιμή 0, 1, Z, -, L, H, U, X και W. Είναι πλέον γνωστό το τι αντιπροσωπεύουν οι τιμές 0 και 1. Οι υπόλοιπες τιμές έχουν τις ακόλουθες έννοιες:

- Z: κατάσταση υψηλής σύνθετης αντίστασης
- - : αδιάφορη λογική κατάσταση
- L: αδιάφορη λογική κατάσταση
- H: ασθενής τιμή 1
- U: μη-μηδενισμένη τιμή
- X: άγνωστη τιμή
- W: άγνωστη ασθενής τιμή

Ένα αντικείμενο τύπου **STD_LOGIC_VECTOR** είναι στην ουσία ένας πίνακας αντικειμένων τύπου **STD_LOGIC.**

Παραδείγματα:

```
signal x: STD_LOGIC;
signal y: STD_LOGIC_VECTOR (15 downto 0);
signal s: STD_LOGIC_VECTOR (0 to 31);
```

Η χρήση των εν λόγω τύπων αντικειμένων επιβάλλει την εισαγωγή στην αρχή ενός προγράμματος των εντολών:

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

2.2.3.1.3. SIGNED – UNSIGNED

Ένα αντικείμενο τύπου **STD_LOGIC_VECTOR** καθορίζεται ως **SIGNED** όταν έχουμε να χειριστούμε προσημασμένους αριθμούς ενώ ένα αντικείμενο τύπου **STD_LOGIC_VECTOR** καθορίζεται ως **UNSIGNED** όταν έχουμε να χειριστούμε μη - προσημασμένους αριθμούς.

Η χρήση των παραπάνω τύπων (SIGNED & UNSIGNED) επιβάλλει την εισαγωγή στην αρχή ενός προγράμματος του πακέτου *std_logic_signed* όταν αναφερόμαστε σε προσημασμένους αριθμούς και το πακέτο *std_logic_unsigned* όταν αναφερόμαστε σε μη-προσημασμένους αριθμούς.

Παραδείγματα:

signal y: SIGNED STD_LOGIC_VECTOR (15 **downto** 0);
signal s: UNSIGNED STD_LOGIC_VECTOR (0 **to** 31);

2.2.3.1.4. INTEGER

Ένα σήμα τύπου **INTEGER** αντιπροσωπεύει ένα δυαδικό αριθμό, έχει μήκος 32 bits έχοντας εύρος τιμών από - $(2^{31} -1)$ έως $(2^{31} - 1)$. Το εύρος μπορεί να καθοριστεί από τον σχεδιαστή - μηχανικό με τη χρήση της λέξης κλειδί **RANGE**.

Παράδειγμα:

signal y: INTEGER **RANGE** - 32768 TO 32767

2.2.3.1.5. ENUMERATION

Ένα σήμα τύπου **ENUMERATION** είναι ένα σήμα για το οποίο οι δυνατές τιμές που μπορεί να πάρει καθορίζονται από τον χρήστη. Τα εν λόγω σήματα χρησιμοποιούνται κυρίως σε μηχανές πεπερασμένων καταστάσεων (finite state machines).

Η γενική μορφή χρήσης ενός τέτοιου τύπου είναι:

type όνομα_τύπου_enumeration **is** (name_A, name_B ...);

Παράδειγμα:

type State_type **is** (stateA, stateB, stateC);
signal y: State_type;

Έτσι έχουμε ένα σήμα y του οποίου οι δυνατές τιμές είναι οι stateA, stateB και stateC.

2.2.3.1.6. BOOLEAN

Ένα σήμα τύπου **BOOLEAN** είναι ένα σήμα το οποίο μπορεί να έχει την τιμή **TRUE** (ισοδύναμη με την τιμή 1) και την τιμή **FALSE** (ισοδύναμη με την τιμή 0).

Παράδειγμα:

```
signal y: BOOLEAN;
```

2.2.4. ΑΛΛΟΙ ΤΡΟΠΟΙ ΔΗΛΩΣΗΣ ΑΝΤΙΚΕΙΜΕΝΩΝ

Σε μια περιγραφή VHDL δεν δημιουργούνται όλα τα αντικείμενα με τη χρήση δηλώσεων αντικειμένων. Αυτά τα αντικείμενα δηλώνονται ως:

1. *Θύρες μιας οντότητας.* Όλες οι θύρες είναι αντικείμενα σήματος.
2. *Γενικεύσεις μιας οντότητας.* Αυτά είναι αντικείμενα σταθερών.
3. *Τυπικές παράμετροι λειτουργιών και διαδικασιών.* Οι παράμετροι λειτουργιών είναι αντικείμενα σταθερών ή αντικείμενα σήματος, ενώ οι παράμετροι διαδικασιών μπορούν να ανήκουν σε οποιαδήποτε κατηγορία αντικειμένων.
4. *Ένα αρχείο που δηλώνεται από μια δήλωση αρχείων* (file declaration).

Υπάρχουν δύο άλλοι τύποι αντικειμένων που δηλώνεται εμμέσως. Αυτοί είναι οι δείκτες μιας for. . .loop δήλωσης (for...loop statement), καθώς και οι δηλώσεις παραγωγής (generate statements).

Παράδειγμα:

```
for COUNT in 1 to 10 loop
SUM := SUM + COUNT;
end loop;
```

Στη δήλωση αυτή, το αντικείμενο COUNT έχει μια έμμεση δήλωση τύπου integer με εύρος 1 έως 10, και ως εκ τούτου, δεν χρειάζεται να δηλωθεί ρητά. Το αντικείμενο COUNT δημιουργείται όταν το πρόγραμμα μπαίνει στο βρόχο για πρώτη φορά και παύει να υπάρχει μόλις βγει από το βρόχο.

2.3. ΤΥΠΟΙ ΔΕΔΟΜΕΝΩΝ

Κάθε αντικείμενο δεδομένων στην VHDL μπορεί να κρατήσει μια τιμή που ανήκει σε ένα σύνολο τιμών. Αυτό το σύνολο τιμών προσδιορίζεται με την χρήση μιας δήλωσης τύπου. Ένας τύπος είναι ένα όνομα το οποίο συνδέεται με ένα σύνολο τιμών και μια σειρά πράξεων. Ορισμένοι τύποι, και οι πράξεις που μπορούν να εκτελεστούν με αντικείμενα αυτών των τύπων, είναι προκαθορισμένοι στη γλώσσα.

Για παράδειγμα, όπως αναφέρθηκε και παραπάνω, ο INTEGER είναι ένας προκαθορισμένος τύπος με ένα σύνολο ακεραίων τιμών που ανήκει σε ένα συγκεκριμένο εύρος τιμών που παρέχει στο σύστημα η VHDL. Το κατώτατο εύρος τιμών που μπορεί να πάρει είναι από - (2^{31} -1) έως (2^{31} - 1). Μερικοί από τους επιτρεπόμενους και συχνά χρησιμοποιούμενοι προκαθορισμένοι τελεστές είναι το (+), για την προσθήκη, το (-), για την αφαίρεση, το (/), για την διαίρεση, και το (*), για τον πολλαπλασιασμό.

Ένας άλλος προκαθορισμένος τύπος είναι ο BOOLEAN. Παίρνει τις τιμές FALSE και TRUE, και μερικοί από τους προκαθορισμένους τελεστές του είναι οι AND, OR, NOR, NAND, και NOT.

Οι δηλώσεις για την προκαθορισμένους τύπους της γλώσσας περιέχονται στο πακέτο STANDARD (βλ. Παράρτημα Α). Οι τελεστές για αυτούς τους τύπους είναι προκαθορισμένοι στη γλώσσα.

Η γλώσσα παρέχει επίσης τη δυνατότητα να ορίσουμε νέους τύπους χρησιμοποιώντας τις δηλώσεις τύπων, αλλά και να καθοριστεί μια σειρά από πράξεις σε αυτούς τους τύπους με τη χρήση συναρτήσεων που επιστρέφουν τιμές του νέου αυτού τύπου. Όλα τα πιθανά είδη που μπορεί να υπάρξουν στη γλώσσα μπορούν να ταξινομηθούν στις εξής τέσσερις κύριες κατηγορίες:

1. *Κλιμακωτοί τύπους (scalar types):* Οι τιμές που ανήκουν σε αυτούς τους τύπους εμφανίζονται σε μια διαδοχική σειρά.
2. *Σύνθετες μορφές (composite types):* Αυτοί αποτελούνται από στοιχεία ενός μοναδικού τύπου (ένας τύπος πίνακα – array type) ή τα στοιχεία των διαφόρων τύπων (ένα είδος εγγραφής – record type).
3. *Τύποι πρόσβασης (access types):* Αυτοί παρέχουν πρόσβαση σε αντικείμενα ενός συγκεκριμένου τύπου (μέσω της χρήσης δεικτών).
4. *Τύποι αρχείων (file types):* Αυτοί παρέχουν πρόσβαση σε αντικείμενα που περιέχουν μια ακολουθία τιμών ενός συγκεκριμένου τύπου.

Είναι δυνατό να καθοριστούν περιορισμένοι τύποι, ονομάζονται υποκατηγορίες (subtypes), από άλλους προκαθορισμένα ή οριζόμενους από το χρήστη τύπους.

2.4. ΠΡΑΞΕΙΣ ΚΑΙ ΤΕΛΕΣΤΕΣ

Οι προκαθορισμένες τελεστές της γλώσσας κατατάσσονται στις ακόλουθες κατηγορίες:

1. Λογικοί τελεστές
2. Σχεσιακοί τελεστές
3. Αριθμητικοί τελεστές

Οι τελεστές έχουν αυξανόμενη προτεραιότητα πράξεων που πηγαίνει από την κατηγορία (1) και καταλήγει στην κατηγορία (5). Οι τελεστές που ανήκουν στην ίδια κατηγορία έχουν την ίδια προτεραιότητα και η αξιολόγηση γίνεται από τα αριστερά προς τα δεξιά. Οι παρενθέσεις μπορούν να χρησιμοποιηθούν για να παρακαμφθεί η από αριστερά προς τα δεξιά αξιολόγηση.

2.4.1. ΛΟΓΙΚΕΣ ΤΕΛΕΣΤΕΣ

Η VHDL υποστηρίζει λογικές πράξεις για τους προκαθορισμένους τύπους **BIT, BOOLEAN** αλλά και για τους μονοδιάστατους πίνακες **BIT_VECTOR.** Τα αποτελέσματα των λογικών αυτών πράξεων είναι του ιδίου τύπου και μεγέθους με τους τελεσταίους. Πράξεις μεταξύ τύπων διαφορετικού μεγέθους απαγορεύονται.

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

Η πράξη not έχει την μεγαλύτερη προτεραιότητα ενώ όλες οι άλλες λογικές πράξεις, έχουν την ίδια προτεραιότητα. Η πράξη not όμως δεν χρειάζεται παρένθεση.

2.4.2. ΣΧΕΣΙΑΚΟΙ ΤΕΛΕΣΤΕΣ

Οι σχεσιακές πράξεις ελέγχουν για ισότητα, ανισότητα και γενικά κάνουν συγκρίσεις μεταξύ τελεστών. Όλες οι σχεσιακές πράξεις έχουν την ίδια προτεραιότητα και μεγαλύτερη από τις λογικές πράξεις. Το δε αποτέλεσμα είναι πάντα τύπου Boolean (True / False).

Τελεστής	**Λειτουργία**
=	Ισότητα
/=	Όχι ίσο με ...

<	Μικρότερο από ...
<=	Μικρότερο ή ίσο από ...
>	Μεγαλύτερο από ...
>=	Μεγαλύτερο ή ίσο από ...

2.4.3. ΑΡΙΘΜΗΤΙΚΟΙ ΤΕΛΕΣΤΕΣ

Οι αριθμητικές πράξεις θεωρούνται γνωστές οπότε ακολουθεί μια σύντομη αναφορά των τελεστών και των λειτουργιών τους.

Τελεστής	**Λειτουργία**
+	Πρόσθεση
-	Αφαίρεση
&	Συνένωση
*	Πολλαπλασιασμός
/	Διαίρεση
mod	modulus
rem	Υπόλοιπο
abs	Απόλυτο
**	Ύψωση σε δύναμη

ΚΕΦΑΛΑΙΟ 3 – ΜΟΝΤΕΛΟ ΣΥΜΠΕΡΙΦΟΡΑΣ (BEHAVIORAL MODEL)

Το κεφάλαιο αυτό παρουσιάζει το συμπεριφορικό μοντέλο σχεδιασμού. Σε αυτό το στυλ μοντελοποίησης, η συμπεριφορά της οντότητας εκφράζεται χρησιμοποιώντας διαδοχικά εκτελέσιμο, διαδικαστικού τύπου κώδικα, ο οποίος είναι παρόμοιος στη σύνταξη και στη σημασιολογία με εκείνον μιας υψηλού επιπέδου γλώσσας προγραμματισμού όπως η C ή η Pascal. Ο κύριος μηχανισμός που χρησιμοποιείται στο σχεδιασμό της διαδικαστικού τύπου συμπεριφοράς μιας οντότητας είναι η δήλωση διαδικασιών.

Αυτό το κεφάλαιο περιγράφει την δήλωση διαδικασιών (process statement) και τα διάφορα είδη των διαδοχικών δηλώσεων (sequential statements) που μπορούν να χρησιμοποιηθούν μέσα σε μια δήλωση διαδικασίας για την περιγραφή αυτής της συμπεριφοράς.

Ανεξάρτητα από το στυλ μοντελοποίησης που χρησιμοποιείται, κάθε οντότητα αντιπροσωπεύεται με την χρήση μιας δήλωσης οντότητας (entity declaration) και τουλάχιστον ένα 'σώμα' αρχιτεκτονικής (architecture body).

3.1. ΔΗΛΩΣΗ ΟΝΤΟΤΗΤΑΣ (ENTITY DECLARATION)

Μια δήλωση οντότητας περιγράφει την εξωτερική διασύνδεση της οντότητας. Προσδιορίζει το όνομα της οντότητας, τα ονόματα των θυρών διασύνδεσης, τη λειτουργία τους (δηλαδή, κατεύθυνση), καθώς και το είδος των θυρών. Η σύνταξη για μια δήλωση οντότητας είναι:

entity *entity-name* **is**
[**generic** (*list-of-generics-and-their-types*) ;]
[**port** (*list-of-interface-port-names-and-their-types*) ;]
[*entity-item-declarations*]

[**begin**
entity-statements]
end [*entity-name*];

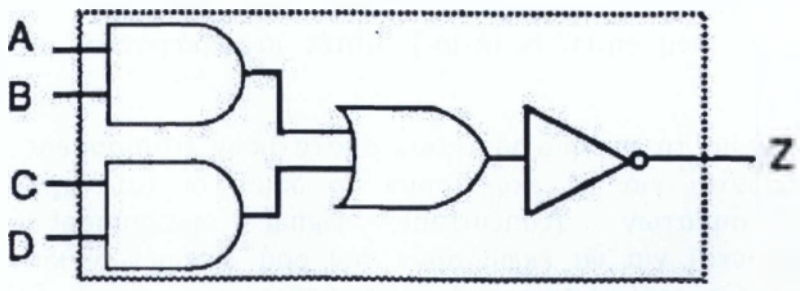
Το *entity-name* είναι το όνομα της οντότητας και οι θύρες διασύνδεσης (*interface-ports)* είναι τα σήματα μέσω των οποίων η οντότητα μεταβιβάζει τις πληροφορίες από και προς το εξωτερικό περιβάλλον της. Κάθε θύρα διασύνδεσης μπορεί να έχει μία από τις παρακάτω λειτουργίες:

1. in: η αξία μιας θύρας εισόδου μπορεί να διαβαστεί μόνο στο εσωτερικό του μοντέλου της οντότητας.
2. out: η αξία μιας θύρας εξόδου δεν μπορεί παρά να ενημερωθεί εντός του μοντέλου οντότητας και δεν μπορεί να διαβαστεί.
3. inout: η αξία μιας αμφίδρομης θύρας μπορεί να διαβαστεί και να ενημερωθεί εντός του μοντέλου οντότητας.
4. buffer: η αξία μιας θύρας buffer μπορεί να διαβαστεί και να ενημερωθεί εντός του μοντέλου οντότητας. Ωστόσο, διαφέρει από την inout λειτουργία στο ότι δεν μπορεί να έχει περισσότερες από μια πηγή και ότι το μόνο είδος σήματος που μπορεί να συνδέεται με αυτό μπορεί να είναι μια άλλη θύρα buffer ή ένα σήμα με το πολύ μια πηγή.

Παράδειγμα:

```
entity AOI is
        port (A, B, C, D: in BIT; Z: out BIT);
end AOI;
```

Η δήλωση της οντότητας προσδιορίζει ότι το όνομα της οντότητας είναι AOI και ότι έχει τέσσερα σήματα εισόδου τύπου BIT και ένα σήμα εξόδου τύπου BIT. Σημειώστε ότι δεν προσδιορίζει τη σύνθεση ή τη λειτουργία της οντότητας.



Σχήμα 3.1 And-Or-Invert Circuit

3.2. ΚΟΜΜΑΤΙ ΑΡΧΙΤΕΚΤΟΝΙΚΗΣ (ARCHITECTURE BODY)

Ένα κομμάτι αρχιτεκτονικής περιγράφει την εσωτερική όψη μιας οντότητας, δηλαδή τη λειτουργικότητα ή τη δομή της οντότητας. Η σύνταξη ενός 'σώματος' αρχιτεκτονικής είναι:

architecture *architecture-name* **of** *entity-name* **is**
    [*architecture-item-declarations*]
begin
    concurrent-statements; these are —>
        process-statement
        block-statement
        concurrent-procedure-call
        concurrent-assertion-statement
        concurrent-signal-assignment-statement
        component-instantiation-statement
        generate-statement
end [*architecture-name*] ;

Οι ταυτόχρονες δηλώσεις (concurrent-statements) περιγράφουν την εσωτερική σύνθεση της οντότητας. Όλες οι ταυτόχρονες δηλώσεις εκτελούνται παράλληλα, και ως εκ τούτου, η σειρά με την οποία εμφανίζονται μέσα στο σώμα αρχιτεκτονικής δεν έχει καμία επίπτωση στην επιθυμητή συμπεριφορά της οντότητας.

Η εσωτερική σύνθεση της οντότητας μπορεί να εκφραστεί από την άποψη της δομής (structure), της ροής δεδομένων (dataflow) και της διαδοχικής συμπεριφοράς (sequential behavior). Αυτές περιγράφονται με ταυτόχρονες δηλώσεις.

Για παράδειγμα, τα στιγμιότυπα των συστατικών (component-instantiation) χρησιμοποιούνται για να εκφράσουν τη δομή, οι ταυτόχρονες δηλώσεις ανάθεσης σημάτων (concurrent signal assignment statements) χρησιμοποιούνται για να εκφράσουν την ροή δεδομένων και οι δηλώσεις διαδικασιών (process statements) χρησιμοποιούνται για να εκφράσουν την διαδοχική συμπεριφορά. Κάθε παράλληλη δήλωση είναι ένα διαφορετικό στοιχείο που λειτουργεί παράλληλα όπως οι πύλη ενός σχεδίου λειτουργούν παράλληλα.

Παραδείγματα:

```
architecture AOI_CONCURRENT of AOI is
begin
        Z <= not ( (A and B) or (C and D) );
end AOI_CONCURRENT;


architecture AOI_SEQUENTIAL of AOI is
      begin
            process (A, B, C, D)
                 variable TEMPI ,TEMP2: BIT;
            begin
               TEMP1 := A and B;                -- statement 1
               TEMP2:=C and D;                  -- statement 2
               TEMP1 := TEMP1 or TEMP2          -- statement 3
               Z<= not TEMP1;                   -- statement 4
            end process;
       end AOI_SEQUENTIAL;
```

Το πρώτο κομμάτι αρχιτεκτονικής, AOI_CONCURRENT, περιγράφει την οντότητα AOI χρησιμοποιώντας τη σχεδίαση διαγράμματος ροής δεδομένων (dataflow style of modeling). Το δεύτερο κομμάτι αρχιτεκτονικής, AOI_SEQUENTIAL, χρησιμοποιεί το τον συμπεριφορικό τρόπο σχεδίασης (behavioral modeling style).

Στο κεφάλαιο αυτό, περιγράφεται μια οντότητα που χρησιμοποιεί τη συμπεριφορική μοντελοποίηση. Ο κύριος μηχανισμός που χρησιμοποιείται για την περιγραφή της λειτουργικότητας μιας οντότητας σε αυτό το ύφος μοντελοποίησης είναι μια δήλωση διαδικασίας (process statement), η οποία είναι μια ταυτόχρονη δήλωση.

3.3. ΔΗΛΩΣΕΙΣ ΔΙΑΔΙΚΑΣΙΩΝ (PROCESS STATEMENTS)

Μια δήλωση διαδικασιών περιλαμβάνει διαδοχικές δηλώσεις που περιγράφουν τη λειτουργικότητα ενός τμήματος μιας οντότητας. Η σύνταξη της δήλωσης της διαδικασίας είναι:

```
[ process-label: ] process [ ( sensitivity-list ) ]
        [process-item-declarations]
begin
```

sequential-statements; these are ->
variable-assignment-statement
signal-assignment-statement
wait-statement
if-statement
case-statement
loop-statement
null-statement
exit-statement
next-statement
assertion-statement
procedure-call-statement
return-statement.

end process [*process-label*];

Ένα σύνολο σημάτων στα οποία η διαδικασία είναι ευαίσθητη, ορίζονται από τη λίστα ευαισθησίας. Με άλλα λόγια, κάθε φορά που συμβαίνει κάποιο γεγονός σε οποιαδήποτε από τα σήματα της λίστας ευαισθησίας, οι διαδοχικές δηλώσεις που βρίσκονται μέσα στην διαδικασία εκτελούνται σε διαδοχική σειρά, δηλαδή, με τη σειρά με την οποία εμφανίζονται. Η διαδικασία αναστέλλεται μετά την εκτέλεση της τελευταίας διαδοχικής δήλωσης και περιμένει ένα άλλο γεγονός να συμβεί σε ένα σήμα της λίστας ευαισθησίας. Στοιχεία που δηλώνονται στο κομμάτι δηλώσεων στοιχείων είναι διαθέσιμα για χρήση μόνο εντός της διαδικασίας.

Το κομμάτι αρχιτεκτονικής, AOI_SEQUENTIAL, που παρουσιάστηκε νωρίτερα, περιλαμβάνει μία δήλωση διαδικασίας. Η δήλωση αυτή έχει τέσσερα σήματα στη λίστα ευαισθησίας του και έχει μια δήλωση μεταβλητής. Αν λάβει χώρα ένα γεγονός σε οποιαδήποτε από τα σήματα, Α, Β, Γ, ή Δ, η διαδικασία εκτελείται. Αυτό επιτυγχάνεται με την εκτέλεση της δήλωσης 1, πρώτα, στην συνέχεια με την δήλωση 2, ακολουθούμενη από τη δήλωση 3, και στη τέλος τη δήλωση 4. Μετά από αυτό, η διαδικασία αναστέλλεται (η προσομοίωση δεν σταματά) και περιμένει για ένα άλλο γεγονός να συμβεί σε ένα σήμα της λίστας ευαισθησίας

3.4. ΔΗΛΩΣΕΙΣ ΑΝΑΘΕΣΗΣ ΜΕΤΑΒΛΗΤΗΣ (VARIABLE ASSIGNEMENT STATEMENTS)

Οι μεταβλητές μπορούν να δηλωθούν και να χρησιμοποιηθούν μέσα σε μια δήλωση διαδικασίας. Της αποδίδεται μια τιμή χρησιμοποιώντας τη δήλωση ανάθεσης μεταβλητής, η οποία έχει συνήθως τη παρακάτω μορφή:

variable-object := expression;

Η έκφραση υπολογίζεται όταν η δήλωση εκτελείται και η υπολογιζόμενη αξία εκχωρείται στη μεταβλητή στιγμιαία, δηλαδή, στην τρέχουσα χρονική στιγμή προσομοίωσης.
Οι μεταβλητές δημιουργούνται κατά τη στιγμή της επεξεργασίας και διατηρούν τις τιμές τους καθ' όλη την διάρκεια της προσομοίωσης. Αυτό οφείλεται στο γεγονός ότι μια διαδικασία δεν τερματίζει ποτέ. Είναι είτε σε ενεργή κατάσταση, δηλαδή, εκτελείται, ή σε κατάσταση αναστολής, δηλαδή, περιμένει να συμβεί ένα συγκεκριμένο γεγονός.

Παράδειγμα:

```
process (A)
        variable EVENTS_ON_A: INTEGER := 0;
begin
        EVENTS_ON_A := EVENTS_ON_A+1;
end process;
```

Στην έναρξη της προσομοίωσης, η διαδικασία εκτελείται μία φορά. Η μεταβλητή EVENTS_ON_A αρχικοποιείται σε 0 και στη συνέχεια αυξάνεται κατά 1. Στην συνέχεια, κάθε φορά που εμφανίζεται ένα συμβάν με το σήμα A, η διαδικασία ενεργοποιείται και η δήλωση ανάθεσης μεταβλητής εκτελείται. Αυτό κάνει την μεταβλητή EVENTS_ON_A να αυξάνεται. Στο τέλος της προσομοίωσης, η μεταβλητή EVENTS_ON_A περιλαμβάνει το συνολικό αριθμό των συμβάντων που έλαβαν χώρα στο σήμα A συν ένα.

3.4. ΔΗΛΩΣΕΙΣ ΑΝΑΘΕΣΗΣ ΣΗΜΑΤΟΣ (SIGNAL ASSIGNEMENT STATEMENTS)

Στα σήματα αποδίδεται μια τιμή χρησιμοποιώντας τη δήλωση ανάθεσης σήματος, η οποία έχει συνήθως τη παρακάτω μορφή:

signal-object **<=** *expression* [**after** *delay-value*];

Μια δήλωση ανάθεσης σήματος μπορεί να εμφανιστεί μέσα σε μια διαδικασία ή εκτός της διαδικασίας. Εάν συμβεί έξω από μια διαδικασία, θεωρείται ως μια παράλληλη δήλωση ανάθεσης του σήματος. Όταν η δήλωση ανάθεσης σήματος εμφανίζεται μέσα σε μια διαδικασία, θεωρείται ως διαδοχική δήλωση ανάθεσης σήματος και εκτελείται σε σειριακά σε σχέση με τις άλλες διαδοχικές δηλώσεις που εμφανίζονται σε αυτή τη διαδικασία.

Όταν εκτελείται η δήλωση ανάθεσης σήματος, η τιμή της έκφρασης υπολογίζεται και ανατίθεται στο σήμα μετά την καθορισμένη καθυστέρηση. Είναι σημαντικό να σημειωθεί ότι η έκφραση υπολογίζεται κατά τη στιγμή που εκτελείται η δήλωση και όχι μετά την καθορισμένη καθυστέρηση. Εάν δεν έχει

καθοριστεί κάποια χρονική καθυστέρηση, θεωρούμε ότι υπάρχει μια default δέλτα καθυστέρηση.

Παράδειγμα:

```
COUNTER <= COUNTER+ "0010"; - Assign after a delta delay.
PAR <= PAR xor DIN after 12 ns;
Z <= (A0 and A1) or (B0 and B1) or (C0 and C1) after 6 ns;
```

3.5. ΔΕΛΤΑ ΚΑΘΥΣΤΕΡΗΣΗ (DELTA DELAY)

Μια καθυστέρηση δέλτα είναι μια πολύ μικρή καθυστέρηση (απειροελάχιστα μικρή). Δεν αντιστοιχεί σε καμία πραγματική καθυστέρηση και κατά την διάρκεια της προσομοίωσης ο χρόνος δεν προχωράει. Η καθυστέρηση αυτή χρησιμοποιείται σε μοντέλα όπου ένα ελάχιστο χρονικό διάστημα απαιτείται για να συμβεί κάποια αλλαγή.

Μια δέλτα καθυστέρηση επιτρέπει για τη διάταξη των γεγονότων που συμβαίνουν την ίδια στιγμή κατά τη διάρκεια μιας προσομοίωσης. Κάθε μονάδα του χρόνου προσομοίωσης μπορεί να θεωρηθεί ότι αποτελείται από έναν άπειρο αριθμό δέλτα καθυστερήσεων. Ως εκ τούτου, ένα γεγονός συμβαίνει πάντα σε πραγματικό χρόνο προσομοίωσης συν ένα ακέραιο πολλαπλάσιο δέλτα καθυστερήσεων. Για παράδειγμα, τα γεγονότα μπορούν να λάβουν χώρα σε 15ns, 15ns+(1)A, 15ns+2A, 15ns+3A, 22ns, 22ns+A, 27ns, 27ns+A, κτλ.

ΚΕΦΑΛΑΙΟ 4 – ΜΟΝΤΕΛΟ ΡΟΗΣ ΔΕΔΟΜΕΝΩΝ (DATAFLOW MODEL)

Το κεφάλαιο αυτό παρουσιάζει τεχνικές για τον σχεδιασμό του διαγράμματος της ροής δεδομένων μιας οντότητας. Ένα μοντέλο ροής δεδομένων καθορίζει τη λειτουργία της οντότητας, χωρίς όμως να καθορίζεται ρητά η δομή της. Η λειτουργικότητα αυτή δείχνει τη ροή των πληροφοριών που περνάει μέσω της οντότητας, η οποία εκφράζεται κυρίως με την χρήση δηλώσεων ανάθεσης παράλληλων σημάτων. Σε αντίθεση με το σχεδιασμό με βάση το μοντέλο συμπεριφοράς, που περιγράφονται στο προηγούμενο κεφάλαιο, στο οποίο η λειτουργικότητα της οντότητας εκφράζεται χρησιμοποιώντας διαδικαστικού τύπου δηλώσεις που εκτελούνται διαδοχικά.

4.1. ΔΗΛΩΣΗ ΑΝΑΘΕΣΗΣ ΠΑΡΑΛΛΗΛΩΝ ΣΗΜΑΤΩΝ (CONCURRENT SIGNAL ASSIGNMENT STATEMENT)

Ένας από τους κύριους μηχανισμούς για την μοντελοποίηση της συμπεριφοράς της ροής δεδομένων μιας οντότητας, είναι η χρήση δηλώσεων ανάθεσης παράλληλων σημάτων.

Παράδειγμα 1:

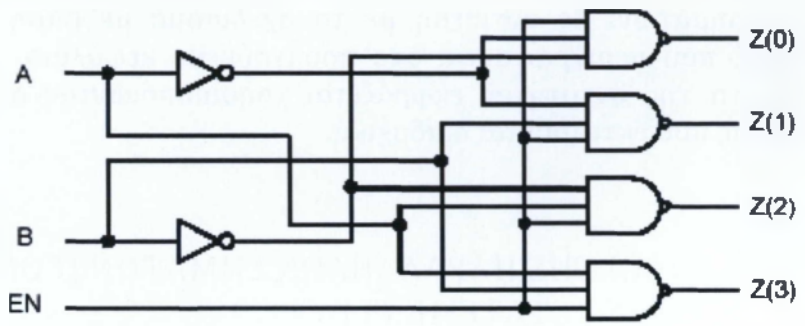
Σχήμα 4.1. Πύλη OR 2 εισόδων

```
entity OR2 is
        port (signal A, B: in BIT; signal Z: out BIT);
end OR2;
architecture OR2 of OR2 is
```

```
begin
        Z <= A or B after 9 ns;
end OR2;
```

Το σώμα αρχιτεκτονικής περιέχει μία μόνο δήλωση ανάθεσης παράλληλου σήματος που αντιπροσωπεύει τη ροής δεδομένων της πύλης OR. Η σημασιολογική ερμηνεία αυτής της δήλωσης είναι ότι κάθε φορά που υπάρχει ένα γεγονός (μια αλλαγή της αξίας), είτε στο σήμα A ή στο B (A και B είναι σήματα στην έκφραση για το Z), η έκφραση στα δεξιά αξιολογείται και η αξία της εμφανίζεται στο σήμα Z μετά από μια καθυστέρηση 9ns. Τα σήματα στην έκφραση, A και B, αποτελούν τη «λίστα ευαισθησίας» για τη δήλωση ανάθεσης του σήματος.

Παράδειγμα 2:



Σχήμα 4.2. Κύκλωμα 2 σε 4 decoder

Η παραπάνω οντότητα έχει τρείς εισόδους (A, B, ENABLE) και τέσσερις εξόδους (Z(0), Z(1), Z(2), Z(3)). Η πύλη Z είναι τύπου bit_vector, δηλαδή ένας μονοδιάστατος πίνακας από bits του οποίου το μέγεθος δηλώνεται από τον χρήστη. Στην προκειμένη περίπτωση το μέγεθος του είναι 4 (0 to 3) ή αλλιώς (3 downto 0).

Με βάση το κύκλωμα 2 to 4 decoder, το κομμάτι αρχιτεκτονικής (architecture body) της οντότητας (entity) DECODER2x4 με τη χρήση της dataflow σχεδίασης είναι:

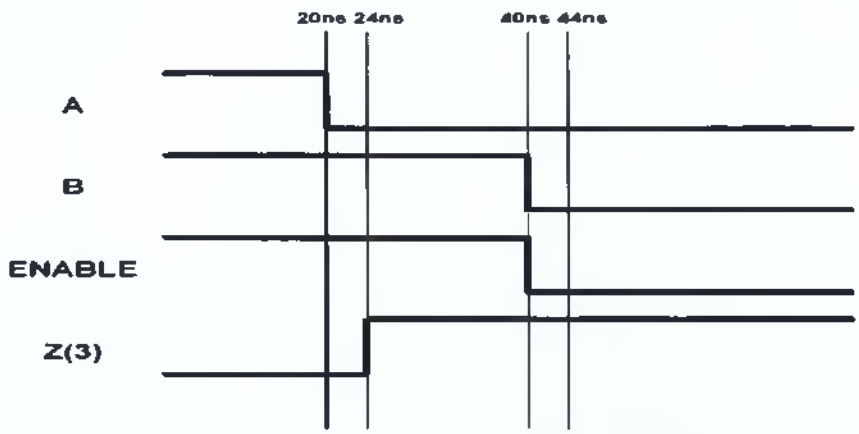
```
architecture DEC_DATAFLOW of DECODER2x4 is
        signal ABAR, BBAR: BIT;
begin
```

```
    Z(3) <= not (A and B and ENABLE);          - statement 1
    Z(0) <= not (ABAR and BBAR and ENABLE);    - statement 2
    BBAR <= not B;                             - statement 3
    Z(2) <= not (A and BBAR and ENABLE);       - statement 4
    ABAR <= not A;                             - statement 5
    Z(1) <= not (ABAR and B and ENABLE);       - statement 6
end DEC_DATAFLOW;
```

Το παραπάνω κομμάτι αρχιτεκτονικής (architecture body), αποτελείται από το κομμάτι ορισμών (declarative part) όπου απλά δηλώνονται δύο σήματα (ABAR, BBAR) και από το κομμάτι των δηλώσεων όπου και βρίσκονται 6 δηλώσεις (παράλληλες). Αξίζει σε αυτό το σημείο να τονίσουμε ότι στην VHDL μπορούμε να κάνουμε χρήση χρονικών καθυστερήσεων. Έτσι στην προκειμένη περίπτωση θα μπορούσαμε για παράδειγμα να γράψουμε για την 1η δήλωση:

```
Z(3)<= not (A and B and ENABLE) after 4ns;      - statement 1
```

Σε αυτήν την περίπτωση αν υποθέσουμε ότι ένα από τα τρία σήματα, στο δεξί μέρος της δήλωσης, αλλάξει κατάσταση την χρονική στιγμή T, τότε η νέα τιμή του Z(3) θα εμφανιστεί την χρονική τιμή T+4ns όπως φαίνεται στο παρακάτω σχήμα. Και για να το αποσαφηνίσουμε ακόμα περισσότερο, αν τη χρονική τιμή T′ όπου T<T′<T+4ns υπάρξει νέα αλλαγή καταστάσεων πάλι σε ένα από τα τρία σήματα, αυτή θα φανεί στο σήμα Z(3) μετά από χρόνο T′+4ns.



Σχήμα 4.3. Κυματομορφή Z(3) σε σχέση με τα σήματα εισόδου

Επανερχόμενοι στον αρχικό κώδικα όπου δεν ήταν δηλωμένη καμία χρονική καθυστέρηση, υποθέτουμε ότι αυτή είναι μηδενική (0ns) ή απείρως μικρή (δηλαδή έχουμε μια καθυστέρηση δέλτα (Δ)).

Για να κατανοήσουμε γενικά τον κώδικα dataflow, ας υποθέσουμε ότι η είσοδος Β αλλάζει κατάσταση την χρονική στιγμή Τ. Αυτό θα έχει ως αποτέλεσμα να "ενεργοποιηθούν" οι δηλώσεις 1,3,6. Οι νέες τιμές των Z(3), BBAR και Z(1) θα εμφανιστούν μετά από χρόνο Τ+Δ. Αφού όμως αλλάζει η τιμή του σήματος BBAR την χρονική στιγμή Τ+Δ τότε θα "ενεργοποιηθεί" και η 2η και η 4η δήλωση και οι νέες τιμές των Z(0) και Z(2) θα εμφανιστούν την χρονική στιγμή Τ+2Δ.

Συνοψίζοντας για την σχεδίαση dataflow, μπορούμε να πούμε ότι αυτή δεν είναι ιδανική για την περιγραφή ακολουθιακών κυκλωμάτων. Αντίθετα όπως υποδηλώνει και το όνομα του, αυτός ο τρόπος σχεδίασης είναι κατάλληλος για περιγραφή κυκλωμάτων που υπάρχει "ροή δεδομένων".

ΚΕΦΑΛΑΙΟ 5 – ΜΟΝΤΕΛΟ ΔΟΜΗΣ (STRUCTURAL MODEL)

Αυτό το κεφάλαιο περιγράφει το δομικό τρόπο σχεδίασης. Μια οντότητα μοντελοποιείται ως ένα σύνολο στοιχείων που συνδέονται με σήματα. Η συμπεριφορά της οντότητας δεν είναι σαφής από το μοντέλο αυτό. Ο κύριος μηχανισμός που χρησιμοποιείται για να περιγραφεί ένα τέτοιο μοντέλο μιας οντότητας είναι η δήλωση δημιουργίας στιγμιότυπου ενός συστατικού (component instantiation statement).

5.1. ΜΟΝΤΕΛΟ ΔΟΜΗΣ (STRUCTURAL MODEL)

Όπως αναφέρθηκε παραπάνω ο δομικός τρόπος σχεδίασης περιγράφει το κύκλωμα κυρίως με βάση τα στοιχεία (components) που το απαρτίζουν. Αυτά μπορεί να είναι στοιχεία από διάφορες βιβλιοθήκες ή και από τον ίδιο το χρήστη από μία άλλη σχεδίαση του. Το αν ο σχεδιαστής θα χρησιμοποιήσει στοιχεία από "έτοιμες" βιβλιοθήκες ή θα προτιμήσει δικής του κατασκευής στοιχεία (generic/user-defined components) είναι ένα σημαντικό θέμα το οποίο επιλύεται με βάση την εμπειρία του αλλά και με βάση τις απαιτήσεις της σχεδίασης του.

Στο δομικό τρόπο σχεδίασης, το κομμάτι της αρχιτεκτονικής (architecture body) είναι αυτό που ορίζει ποια στοιχεία (components) εμπεριέχονται στη συγκεκριμένη σχεδίαση καθώς και το πώς ενώνονται μεταξύ τους. Τα κύρια στοιχεία της VHDL που χρησιμοποιούνται σε αυτό τον τρόπο σχεδίασης είναι:

- Component declaration and instantiation (Δήλωση στοιχείων)
- Port mapping and signal interface lists (Αναφορά στον τρόπο σύνδεσης του κάθε στοιχείου της σχεδίασης με τα υπόλοιπα. Δηλαδή η έξοδος του στοιχείου σε ποια είσοδο άλλου στοιχείου πηγαίνει και με τη χρήση ποιου σήματος)
- Libraries and packages (βιβλιοθήκες και πακέτα)
- Signals (for interconnections)

Παράδειγμα:

```
architecture DEC_STRUCT of DECODER2x4 is
  component INV
    port(PIN: in BIT; POUT: out BIT);
```

```
  end component;
  component NAND3
    port(DO, DI, D2: in BIT; DZ: out BIT);
  end component;
  signal ABAR,BBAR: BIT;
begin
  V0: INV port map (A, ABAR);
  V1: INV port map (A, BBAR);
  N0: NAND3 port map (ENABLE, ABAR, BBAR, Z(0));
  NI: NAND3 port map (ABAR, B, ENABLE, Z(l));
  N2: NAND3 port map (A, BBAR, ENABLE, Z(2));
  N3: NAND3 port map (A, B, ENABLE, Z(3));
end DEC_STRUCT;
```

Σε αυτό το παράδειγμα το όνομα του κομματιού της αρχιτεκτονικής (architecture body) είναι DEC_STRUCT. Η οντότητα (entity) DECODER2x4 ορίζει τις εισόδους και τις εξόδους γι' αυτό το κομμάτι αρχιτεκτονικής. Το κομμάτι αρχιτεκτονικής χωρίζεται σε δύο μέρη: το κομμάτι των ορισμών (declarative part) πριν τη λέξη-κλειδί begin και το κομμάτι των δηλώσεων (statement part) μετά τη λέξη-κλειδί begin.

Στο κομμάτι των ορισμών υπάρχει ο ορισμός δύο στοιχείων (components), της πύλης not (INV) και της πύλης nand 3-εισόδων (NAND3). Αυτοί οι ορισμοί δείχνουν ουσιαστικά τις εισόδους και τις εξόδους κάθε στοιχείου. Τα στοιχεία αυτά μπορεί να είναι στοιχεία βιβλιοθηκών ή στοιχεία δημιουργημένα από τον σχεδιαστή.

Επίσης στο κομμάτι των ορισμών υπάρχει και ο ορισμός δύο σημάτων (signals) του ABAR και του BBAR. Αυτά τα δύο σήματα είναι κατ' ουσία δύο σύρματα τα οποία χρησιμοποιούνται για την ένωση των στοιχείων για τη δημιουργία του 2-σε-4 decoder. Η εμβέλεια αυτών των σημάτων βρίσκεται μόνο μέσα στο κομμάτι αρχιτεκτονικής και κατά συνέπεια δεν είναι ορατά έξω απ' αυτό. Σε αντίθεση με τις πύλες της οντότητας οι οποίες είναι ορατές σε όλα τα μέρη του κώδικα.

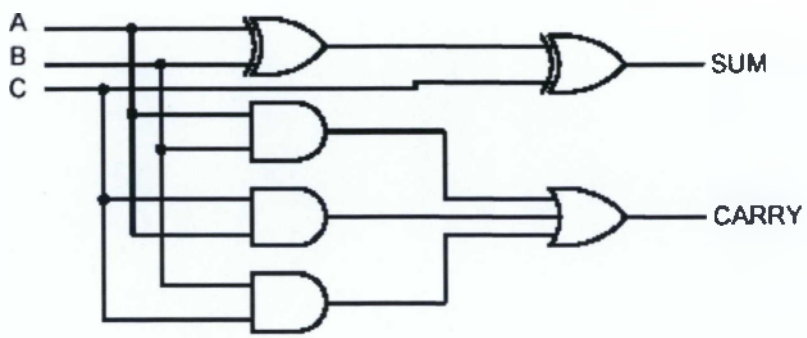
Τα ορισμένα στοιχεία κατόπιν "αποτυπώνονται" στο κομμάτι των δηλώσεων (instantiated). Οι ταυτότητες (labels) αυτών των αποτυπώσεων (στιγμιότυπων) στην συγκεκριμένη περίπτωση είναι οι V0, V1, N0, N1, N2, N3. Η ταυτότητα V0 (μιας πύλης NOT) αν κοιτάξει κανείς τον κώδικα, δείχνει ότι δέχεται, το συγκεκριμένο στοιχείο, ως είσοδο μια είσοδο της σχεδίασης (την A) και έχει ως έξοδο το σήμα ABAR. Ένα σύρμα δηλαδή που αργότερα πηγαίνει ως είσοδο σε δύο πύλες NAND3 και ειδικότερα στα στοιχεία N2 και N3. Το ίδιο συμβαίνει και με τα υπόλοιπα στοιχεία.

Η αποτύπωση (instantiation) γενικά είναι μια παράλληλη δήλωση. Γι' αυτό το λόγο δεν παίζει ρόλο η σειρά των δηλώσεων. Ο δομικός τρόπος σχεδίασης γενικά δείχνει απλά το πώς συνδέονται το διάφορα στοιχεία μεταξύ τους. Τα στοιχεία με αυτόν τον τρόπο σχεδίασης αποτελούν "μαύρα κουτιά", δηλ. δεν γίνεται καμία αναφορά για τη συμπεριφορά ή τη λειτουργία τους. Γι' αυτό το λόγο αλλά και για άλλους, ο δομικός τρόπος σχεδίασης χρησιμοποιείται συχνά στο υψηλότερο επίπεδο σχεδίασης έτσι ώστε να γίνεται ένας διαυγής καθορισμός των επιμέρους στοιχείων της τελικής σχεδίασης αλλά και για να έχει η σχεδίαση μια καθαρά ιεραρχική δομή.

ΚΕΦΑΛΑΙΟ 6 – ΣΥΝΔΙΑΣΜΟΣ ΤΡΟΠΩΝ ΣΧΕΔΙΑΣΗΣ (MIXED STYLE OF MODELING)

6.1. ΣΥΝΔΙΑΣΜΟΣ ΤΡΟΠΩΝ ΣΧΕΔΙΑΣΗΣ (MIXED STYLE OF MODELING)

Είναι δυνατό μία σχεδίαση να περιέχει και τους τρεις προαναφερθέντες τρόπους σχεδίασης. Δηλαδή να γίνεται χρήση components (Structural Modeling), παράλληλες δηλώσεις (Dataflow Modeling) καθώς και processes (Behavioral Modeling). Το πιο συνηθισμένο όμως είναι να συνυπάρχει ο dataflow τρόπος σχεδίασης με έναν από τους άλλους δύο. Παρακάτω ακολουθεί η περιγραφή ενός αθροιστή (1-bit) και με τους τρεις τρόπους σχεδίασης:



Σχήμα 6.1 : Πλήρης αθροιστής 1-bit

```
entity FULL_ADDER is
  port(A : in bit; B : in bit; CIN : in bit; SUM : out bit; COUT : out bit);
end FULL_ADDER;
architecture FA_MIXED of FULL_ADDER is
```

```
  component XOR2
    port (Pl, P2 : in bit; PZ: out bit);
  end component;
  signal Sl : bit;
begin
  XI: XOR2 port map (A, B, S1);                    -- structure
  process (A, B, CIN)                              -- behavior
    variable T1,T2,T3 : bit;
  begin
    Tl := A and B;
    T2 := B and CIN;
    T3 := A and CIN;
    COUT<= Tl or T2 or T3;
  end process;
  SUM <= Sl xor CIN;                               -- dataflow
end FA_MIXED;
```

ΚΕΦΑΛΑΙΟ 7 – ΠΑΚΕΤΑ & ΒΙΒΛΙΟΘΗΚΕΣ (PACKAGES & LIBRARIES)

7.1. ΠΑΚΕΤΑ (PACKAGES)

Ο ορισμός πακέτων χρησιμοποιείται για την αποθήκευση ενός συνόλου κοινών στοιχείων (components), διαδικασιών (procedures), συναρτήσεων (functions), types, subtypes κ.ά. τα οποία χρησιμοποιούνται από τον σχεδιαστή συχνά. Η δήλωση πακέτων (package declaration) καθώς και τα ίδια τα πακέτα (package body) είναι σχεδιαστικές μονάδες (design units), κατά συνέπεια μπορούν να κάνουν χρήση των δεδομένων άλλων πακέτων.

Ένα παράδειγμα πακέτου προσβάσιμο σε όλους τους χρήστες είναι το πακέτο STANDARD (βλ. παράρτημα Α.2.). Αυτό δεν μπορεί να τροποποιηθεί από τον χρήστη και μαζί με το πακέτο TEXTIO (βλ. παράρτημα Α.3.), βρίσκονται μέσα στη βιβλιοθήκη STD. Η ειδική χρησιμότητα των πακέτων δεν φαίνεται εκ πρώτης όψεως. Όμως αξίζει να αναφερθεί ότι μερικά εξειδικευμένα πακέτα που προσφέρουν οι εταιρίες κατασκευής ASICs και FPGAs[6], μπορούν να βοηθήσουν τον σχεδιαστή να υλοποιήσει ένα σύστημα βασισμένος σε μια συγκεκριμένη τεχνολογία.

7.1. 1. ΔΗΛΩΣΗ ΠΑΚΕΤΟΥ (PACKAGE DECLARATION)

Η σύνταξη της δήλωσης πακέτο είναι η εξής:

package *package-name* **is**
package-item-declarations "> These may be:
- subprogram declarations ~ type declarations
- subtype declarations

[6] **FPGA:** Field Programmable Gate Arrays

- constant declarations
- signal declarations
- file declarations
- alias declarations
- component declarations
- attribute declarations
- attribute specifications
- disconnection specifications
- use clauses

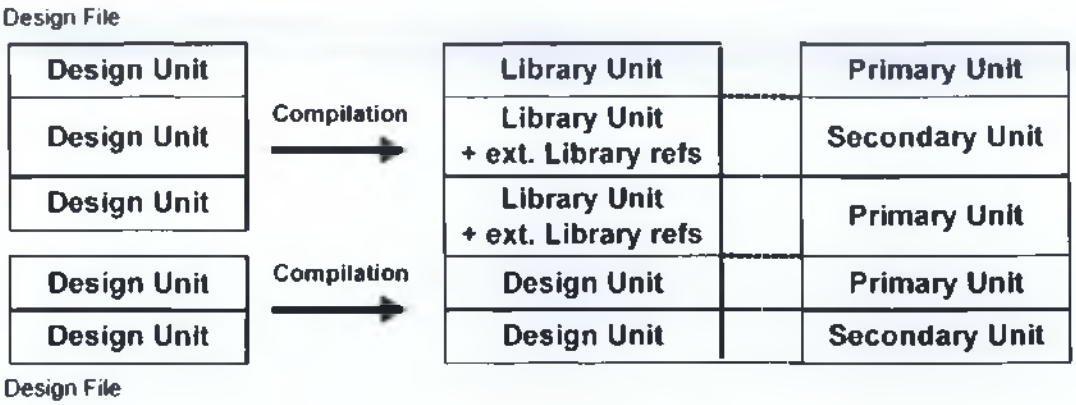
end [*package-name*] ;

Παράδειγμα:

```
package SYNTH_PACK is
        constant LOW2HIGH: TIME := 20ns:
        type ALU_OP is (ADD, SUB, MUL, DIV, EQL);
        attribute PIPELINE: BOOLEAN;
        type MVL is ('U', '0', '1', 'Z');
        type MVL_VECTOR is array (NATURAL range <>) of MVL;
        subtype MY_ALU_OP is ALU_OP range ADD to DIV;
        component NAND2
                port (A, B: in MVL; C: out MVL);
        end component;
end SYNTH_PACK;
```

7.2. ΒΙΝΑΙΟΘΗΚΕΣ (LIBRARIES)

Σχεδόν καμία σχεδίαση που κάνουμε δεν είναι απόλυτα αυτόνομη. Πάντα θα γίνεται αναφορά σε κάποιο block το οποίο αναπαριστά ένα δομικό στοιχείο (component) ή έναν ορισμό. Αυτά μπορεί να είναι αποθηκευμένα στην ίδια βιβλιοθήκη με το παρόν σχέδιο ή σε κάποια ευρύτερη (global) βιβλιοθήκη. Επίσης κάθε βιβλιοθήκη χωρίζεται σε επιμέρους τμήματα, ανάλογα με τα είδη αναφορών που έχει. Η δομή μίας βιβλιοθήκης φαίνεται στο παρακάτω σχήμα:



Σχήμα 6.1. Δομή Βιβλιοθήκης

Ένα design file[7] περιέχει τον VHDL κώδικα ενός στοιχείου (element) του σχεδίου. Πολλά design files μπορεί να χρησιμοποιηθούν για την εφαρμογή πολλών στοιχείων του ιδίου σχεδίου. Κάθε τέτοιο αρχείο μπορεί και να περιέχει κομμάτι(α) της δομής ενός στοιχείου κυκλώματος. Αυτά τα κομμάτια (units) χωρίζονται σε:

- Πρωτεύοντα (primary units). Περιέχουν τον ορισμό μιας οντότητας (entity) ή ενός πακέτου (package).
- Δευτερεύοντα (secondary units). Περιέχουν τον ορισμό ενός κομματιού αρχιτεκτονικής (architecture body) ή ενός πακέτου (package).

Τα πρωτεύοντα κομμάτια πρέπει να «περάσουν» το compilation και να αποθηκευτούν σε μια βιβλιοθήκη πριν ακολουθήσουν τα δευτερεύοντα με τα οποία συνδέονται. Το σύνολο αυτών των μονάδων απαρτίζουν τα library units.

7.3. ΠΡΟΣΒΑΣΗ ΣΕ ΒΙΒΛΙΟΘΗΚΕΣ ΚΑΙ ΠΑΚΕΤΑ

Πρώτα απ' όλα να σημειωθεί ότι υπάρχουν design units μέσα σε βιβλιοθήκες που είναι "ορατά" σε κάθε σχεδίαση. Αυτά βρίσκονται στο STANDARD πακέτο στην STD βιβλιοθήκη καθώς και σε όλα τα πακέτα της WORK βιβλιοθήκης. Αν αυτά δεν ήταν ορατά στον χρήστη τότε κάθε σχεδίαση του θα έπρεπε να προαναφέρει τις ακόλουθες δηλώσεις:

[7] **design file:** είναι ένα ASCII αρχείο που περιέχει τον πηγαίο κώδικα της VHDL. Μπορεί να περιέχει μία ή περισσότερες μονάδες σχεδιασμού (πχ. δήλωση οντότητας, δήλωση πακέτου, κ.ά.)

library STD, WORK; - δήλωση βιβλιοθηκών

use STD.STANDARD.**all** - δήλωση όλων των στοιχείων ενός πακέτου

Η πρώτη δήλωση ονομάζεται library clause και κάθε βιβλιοθήκη μπορεί να γίνει "ορατή" στη σχεδίαση του χρήστη με αυτόν τον τρόπο. Αντίστοιχα η δεύτερη δήλωση ονομάζεται use clause και καλεί συγκεκριμένα πακέτα ή στοιχεία πακέτων από μία βιβλιοθήκη.

Αν ο χρήστης θέλει να έχει πρόσβαση σε ένα αντικείμενο από ένα πακέτο, δεν είναι αναγκαίο να «φορτώσει» όλο το πακέτο αλλά μόνο το επιθυμητό αντικείμενο. Για παράδειγμα, ας υποθέσουμε ότι το αντικείμενο είναι μία διαδικασία (procedure) ή ένα στοιχείο (component) με το όνομα DECODE και βρίσκεται στο πακέτο MY_PROCS μέσα στη βιβλιοθήκη MY_LIB. Τότε η πρόσβαση γίνεται ως εξής:

```
MY_LIB.MY_PROCS.DECODE;
```

ΠΑΡΑΡΤΗΜΑ Α

Αυτό το παράρτημα περιγράφει το προκαθορισμένο περιβάλλον της γλώσσας.

Α.1. ΔΕΣΜΕΥΜΕΝΕΣ ΛΕΞΕΙΣ

Τα παρακάτω αναγνωριστικά είναι δεσμευμένες λέξεις στη γλώσσα (γνωστά και ως λέξεις-κλειδιά), και ως εκ τούτου, δεν μπορούν να χρησιμοποιηθούν ως αναγνωριστικά σε μια περιγραφή VHDL.

Abs	**access**	**after**	**alias**
all	**and**	**architecture**	**array**
assert	**attribute**		
begin	**block**	**body**	**buffer**
bus			
case	**component**	**configuration**	**constant**
disconnect	**downto**		
else	**elsif**	**end**	**entity**
exit			
file	**for**	**function**	
generate	**generic**	**guarded**	
if	**in**	**inout**	**is**
label	**library**	**linkage**	**loop**
map	**mod**		
nand	**new**	**next**	**nor**

not	**null**		
of	**on**	**open**	**or**
others	**out**		
package	**port**	**procedure**	**process**
range	**record**	**register**	**rem**
report	**return**		
select	**severity**	**signal**	**subtype**
then	**to**	**transport**	**type**
units	**until**	**use**	
variable			
wait	**when**	**while**	**with**
xor			

Α.2. ΠΑΚΕΤΟ STANDARD (PACKAGE STANDARD)

Το πακέτο **STANDARD** είναι ένα προκαθορισμένο πακέτο που περιέχει τους ορισμούς για την προκαθορισμένους τύπους και τις λειτουργίες της γλώσσας. Ακολουθεί ένα κομμάτι του πηγαίου κώδικα του πακέτου.

```
package STANDARD is
    -- Predefined enumeration types:
    type BOOLEAN is (FALSE, TRUE);
    type BIT is ('0', '1');
    type CHARACTER is (
      NUL,  SOH,  STX,  ETX,  EOT,  ENQ,  ACK,
      BEL,  BS,   HT,   LF,   VT,   FF,   CR,
      SO,   SI,   DLE,  DCI,  DC2,  DC3,  DC4,
      NAK,  SYN,  ETB,  CAN,  EM,   SUB,  ESC,
      FSP,  GSP,  RSP,  USP,
      ' ',  '!',  '"',  '#',  '$',  '%',  '&',
      ''',  '(',  ')',  '*',  '+',  ',',  '-',
      '.',  '/',  '0',  '1',  '2',  '3',  '4',
      '5',  '6',  '7',  '8',  '9',  ':',  ';',
```

```
    '<',  '=',  '>',  '?',
    '@',  'A',  'B',  'C',  'D',  'E',  'F',
    'G',  'H',  'I',  'J',  'K',  'L',  'M',
    'N',  'O',  'P',  'Q',  'R',  'S',  'T',
    'U',  'V',  'W',  'X',  'Y',  'Z',  '[',
    '\',  ']',  '^',  '_',
    '`',  'a',  'b',  'c',  'd',  'e',  'f',
    'g',  'h',  'i',  'j',  'k',  'l',  'm',
    'n',  'o',  'p',  'q',  'r',  's',  't',
    'u',  'v',  'w',  'x',  'y',  'z',  '{',
    '|',  '}',  '~',  DEL
);
```

```
type SEVERITY_LEVEL is (NOTE, WARNING,
                ERROR, FAILURE);
                -- Predefined numeric types:
type INTEGER is range implementation_defined;
type REAL is range implementation_defined;
                - Predefined type TIME:
type TIME is range implementation_defined
                        units
fs; -- femtosecond
ps = 1000 fs; -- picosecond
ns = 1000 ps; -- nanosecond
us = 1000 ns; -- microsecond
ms = 1000 us; -- minisecond
sec = 1000 ms; -- seconds
min = 60 secs; -- minutes
hr = 60 min; -- hours
                        end units;
-- Function that returns the current simulation time:
                        function NOW return TIME;
                        -- Predefined numeric subtypes:
subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;
subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;
                        - Predefined array types:
type STRING is array (POSITIVE range <>) of CHARACTER;
type BIT_VECTOR is array (NATURAL range <>) of BIT;
end STANDARD;
```

Α.3. ΠΑΚΕΤΟ TEXTIO (PACKAGE TEXTIO)

Το πακέτο TEXTIO περιέχει δηλώσεις των τύπων και των υποπρογραμμάτων που υποστηρίζουν τη μορφοποίηση ASCII λειτουργιών εισόδου / εξόδου. Ακολουθεί ένα κομμάτι του πηγαίου κώδικα του πακέτου.

```
package TEXTIO is
-- Type definitions for text I/O:
type LINE is access STRING;          -- A line is a pointer to a
                                     -- STRING value.
type TEXT is file of STRING;         -- A file of variable- length
                                     -- ASCII records.

type SIDE is (RIGHT, LEFT);          -- For justifying output data within
                                     -- fields.
subtype WIDTH is NATURAL;            -- For specifying widths of output
                                     -- fields.

-- Standard text files:
file INPUT: TEXT is in "STD_INPUT";
file OUTPUT: TEXT is out "STD_OUTPUT":

-- Input routines for standard types:
procedure READLINE (F: in TEXT; L: out LINE);

procedure READ (L: inout LINE; VALUE: out BIT;
                GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out BIT);

procedure READ (L: inout LINE; VALUE: out BIT_VECTOR;
                GOOD; out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out BIT_VECTOR);

procedure READ (L; inout LINE; VALUE: out BOOLEAN;
                GOOD: out BOOLEAN);
procedure READ (L; inout LINE; VALUE: out BOOLEAN):

procedure READ (L: inout LINE; VALUE: out CHARACTER;
                GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out CHARACTER);

procedure READ (L: inout LINE; VALUE: out INTEGER;
                GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out INTEGER);
```

```
procedure READ (L: inout LINE; VALUE: out REAL;
                GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out REAL);

procedure READ (L: inout LINE; VALUE: out STRING;
                GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out STRING);

procedure READ (L: inout LINE; VALUE: out TIME;
                GOOD: out BOOLEAN);
procedure READ (L: inout LINE; VALUE: out TIME);

-- Output routines for standard types;
procedure WRITELINE (F: out TEXT; L: in LINE);

procedure WRITE (L: inout LINE; VALUE: in BIT;
      JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE; VALUE: in BIT_VECTOR;
      JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE; VALUE: in BOOLEAN:
      JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE; VALUE: in CHARACTER;
      JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE; VALUE: in INTEGER;
      JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE; VALUE: in REAL;
      JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0;
DIGITS: in NATURAL := 0);
procedure WRITE (L: inout LINE; VALUE: in STRING;
      JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE; VALUE: in TIME;
      JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0;

UNIT; in TIME := ns);

-- File position predicates:
function ENDLINE (L: in LINE) return BOOLEAN;
-- function ENDFILE (F: in TEXT) return BOOLEAN;
end TEXTIO;
```

ΠΑΡΑΡΤΗΜΑ Β

Το παράρτημα αυτό παρουσιάζει την πλήρη σύνταξη της γλώσσας VHDL.

Β.1. ΣΥΜΒΑΣΕΙΣ

Οι ακόλουθες συμβάσεις έχουν χρησιμοποιηθεί για την περιγραφή της σύνταξης της γλώσσας.

1. Οι κανόνες σύνταξης είναι οργανωμένοι με αλφαβητική σειρά.
2. Οι δεσμευμένες λέξεις είναι γραμμένες με έντονα γράμματα.
3. Ένα όνομα με πλάγιους χαρακτήρες πρόθεμα σε ένα μη ορισμένο όνομα αντιπροσωπεύει την σημασιολογική έννοια που σχετίζεται με αυτό το όνομα.
4. Η κάθετη μπάρα (|) χωρίζει εναλλακτικά στοιχεία, εκτός αν εμφανίζεται αμέσως μετά το άνοιγμα άγκιστρου στην οποία περίπτωση έχει τη κανονική του σημασία.
5. Αγκύλες ([...]) Υποδηλώνουν τα προαιρετικά στοιχεία.
6. Άγκιστρα ({...}) Προσδιορίζουν ένα στοιχείο που επαναλαμβάνεται μηδέν ή περισσότερες φορές.

Β.2. ΣΥΝΤΑΞΗ

Abstract_literal ::= decimal_literal | based_literal

access_type_definition ::= **access** subtype_indication

actual_designator ::=
    expression
    | *signal*_name
    | *variable*_name
    | **open**

actual_parameter_part := parameter_association_list

actual_part ::= actual_designator I function_name (actual_designator)

adding_operator ::= + | - | &

aggregate ::= (element_association { , element_association })

alias_declaration ::= **alias** identifier: subtype_indication **is** name;

allocator ::= **new** subtype_indication | **new** qualified_expression

architecture_body **::=**
    architecture identifier **of** *entity*_name **is**
        architecture_declarative_part
    begin
        architecture_statement_part
    end [*architecture*_simple_name] ;

architecture_declarative_part ::= { block_declarative_item}

architecture_statement_part ::= { concurrent_statement}

array_type_definition ::=
    unconstrained_array_definition | constrained_array_definition

assertion_statement ::=
    assert condition
        [**report** expression]
        [**severity** expression] ;

association_element ::= [format_part =>] actual_part

association_list ::= association_element { , association_element}

attribute_declaration ::= **attribute** identifier: type_mark;

attribute_designator ::= *attribute*_simple_name

attribute_name ::= prefix ' attribute_designator [(static_expression)]

attribute_specification ::=
    attribute attribute_designator **of** entity_specification **is** expression ;

base ::= integer

base_specifier ::= B | O | X

```
base_unit_declaration ::= identifier;

based_integer ::= extended_digit { [ UNDERLINE ] extended_digit )

based_literal ::= base # based_integer [. Based_integer ] # [ exponent ]

basic_character ::= basic_graphic_character | FORMAT_EFFECTOR

basic_graphic_character :=
        UPPER_CASE_LETTER
        | DIGIT
        | SPECIAL_CHARACTER
        | SPACE_CHARACTER

binding_indication ::=
        entity_aspect [ generic_map_aspect ] [ port_map_aspect ]

bit_string_literal ::= base_specifier " bit_value "

bit_value ::= extended_digit { [ UNDERLINE ] extended_digit }

block_configuration ::=
        for block_specification
                { use_clause}
                { configuration_item}
        end for;

block_declarative_item ::=
        subprogram_declaration
        | subprogram_body
        | type_declaration
        | subtype_declaration
        | constant_declaration
        | signal_declaration
        | file_declaration
        | alias_declaration
        | component_declaration
        | attribute_declaration
        | attribute_specification
        | configuration_specification
        | disconnection_specification
        | use_clause

block_declarative_part ::= { block_declarative_item}
```

```
block_header ::=
        [ generic_clause [ generic_map_aspect ; ] ]
        [ port_clause [ port_map_aspect ; ] ]

block_specification ::=
        architecture_name
        | block_staten1ent_label
        | generate_statement_label [ ( index_specification ) ]

block_statement ::=
        block_label:
                block [ ( guard_expression ) ]
                        block_header
                        block_declarative_part
                begin
                        block_statement_part
                end block [ block_label ] ;

block_statement_part ::= { concurrent_statement}
case_statement ::=
        case expression is
                case_statement_alternative
                { case_statement_alternative}
        end case;

case_statement_alternative ::= when choices => sequence_of_statements

character_literal ::= ' graphic_character '
choice ::=
        simple_expression
        | discrete_range
        | element_simple_name
        | others

choices ::= choice { | choice )

component_configuration ::=
        for component_specification
                [ use binding_indication ; ]
                [ block_configuration ]
        end for;

component_declaration ::=
        component identifier
                [ local_generic_clause ]
```

[*local*_port_clause]
end component;

component_instantiation_statement ::=
instantiation_label:
component_name [generic_map_aspect] [port_map_aspect];

component_specification ::= instantiation_list: *component_name*

composite_type_definition ::= array_type_definition | record_type_definition

concurrent_assertion_statement ::= [label :] assertion_statement

concurrent_procedure_call ::= [label :] procedureLcall_statement

concurrent_signal_assignment_statement ::=
[label :] conditional_signaLassignment
| [label :] selected_signal_assignment

concurrent_statement ::=
block_statement
| process_statement
| concurrent_procedure_call
| concurrent_assertion_statement
| concurrent_signal_assignment_statement
| component_instantiation_statement
| generate_statement

condition ::= boolean_expression

condition_clause ::= **until** condition

conditional_signal_assignment ::= target <= options conditional_waveforms ;

conditional_waveforms ::= { waveform when condition else} waveform
configuration_declaration ::=
configuration identifier **of** *entity*_name **is**
configuration_declarative_part
block_configuration
end [*configuration*_simple_name];

configuration_declarative_item ::= use_clause | attribute_specification

configuration_declarative_part ::= { configuration_declarative_item}

configuration_item ::= block_configuration | component_configuration

configuration_specification ::=
    for component_specification **use** binding_indication ;

constant_declaration ::=
    constant identifier_list: subtype_indication [:= expression] ;

constrained_array_definition ::=
    array index_constraint **of** *element*_subtype_indication

constraint ::= range_constraint | index_constraint

context_clause ::= { context_Item }

context_item ::= library_clause | use_clause

decimal_literal ::= integer [. integer] [exponent]

declaration ::=
    type_declaration
    | subtype_declaration
    | object_declaration
    | file_declaration
    | interface_declaration
    | alias_declaration
    | attribute_declaration
    | component_declaration
    | entity_declaration
    | configuration_declaration
    | subprogram_declaration
    | package_declaration

design_file ::= design_unit { design_unit}

design_unit ::= context_clause library_unit

designator ::= identifier | operator_symbol

direction ::= **to** | **downto**

disconnection_specification ::=
    disconnect guarded_signal_specification **after** *time*_expression ;

discrete_range ::= discrete_subtype_indication | range

element_association ::= [choices =>] expression

element_declaration ::= identifier_list: element_subtype_definition ;

element_subtype_definition ::= subtype_indication

entity_aspect ::=
    entity *entity*_name [(*arcitecture*_identifier)]
    | **configuration** *configuration*_name
    | **open**

entity_class ::=
    entity | architecture | configuration | procedure
    | function | package | type | subtype | constant | signal
    | variable | component | label

entity_declaration ::=
    entity identifier **is**
        entity_header
        entity_declarative_part
    [**begin**
        entity_statement_part]
    end [*entify*_simple_name];

entity_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | file_declaration
    | alias_declaration
    | attribute_declaration
    | attribLrte_specification
    | disconnection_specification
    | use_clause

entity_declarative_part ::= { entity_declarative_item}

entity_designator ::= simple_name | operator_symbol

entity_header ::= [*fomal*_generic_clause] [*formal*_port_clause]

entity_name_list ::=
entity_designator { , entity_designator}
| **others**
| **all**

entity_specification ::= entity_name_list: entity_class

entity_statement ::=
concurrent_assertion_statement
| *passive*_concurrent_procedure_call
| *passive*_process_statement

entity_statement_part ::= { entity_statement}

enumeration_literal ::= identifier | character_literal

enumeration_type_definition ::= (enumeration_literal { , enumeration_literal})

exit_statement ::= **exit** [*loop*_label] [**when** condition] ;

exponent ::= E [+] integer | E - integer

expression ::=
relation { **and** relation }
| relation { **or** relation }
| relation { **xor** relation }
| relation [**nand** relation]
| relation [**nor** relation]

extended_digit ::= DIGIT | letter

factor ::=
primary [** primary]
| **abs** primary
| **not** primary

file_declaration ::=
file identifier: subtype_indication **is** [mode] file_logical_name;

file_logical_name ::= *string*_expression

file_type_definition ::= **file of** type_mark

floating_type_definition ::= range_constraint

```
formal_designator ::= generic_name | port_name | parameter_name

formal_parameter_list ::= parameter_interface_list

formal_part ::= formal_designator | function_name ( formal_designator )

full_type_declaration ::= type identifier is type_definition ;

function_call ::= function_name [ ( actual_parameter_part ) ]

generate_statement ::=
        generate_label:
                generation_scheme generate
                        { concurrent_statement}
                end generate [ generate_label ] ;

generation_scheme ::=
        for generate_parameter_specification
        | if condition

generic_clause ::= generic ( generic_list ) ;

generic_list ::= generic_interface_list

generic_map_aspect ::= generic map ( genenc_association_list )

graphic_character ::=
        basic_graphic_character
        | LOWER_CASE_LETTER
        | OTHER_SPECIAL_CHARACTER

guarded_signal_specification ::= guarde_signal_list : type_mark

identifier ::= letter { [ UNDERLINE ] letter_or_digit}

identifier_list ::= identifier { , identifier}

if_statement ::=
        if condition then
                sequence_of_statements
        { elsif condition then
                sequence_of_statements}
        [ else
                sequence_of_statements ]
        end if;
```

incomplete_type_declaration ::= **type** identifier;

index_constraint ::= (discrete_range {, discrete_range })

index_specification ::= discrete_range | *static*_expression

index_subtype_definition ::= type_mark **range <>**

indexed_name ::= prefix (expression { , expression })

instantiation_list ::=
    instantiation_*label* {, *instantiation*_label}
    | **others**
    | **all**

integer ::= DIGIT { [UNDERLINE] DIGIT}

integer_type_definition ::= range_constraint

interface_constant_declaration ::=
    [**constant**] identifier_list: [**in**] subtype_indication
    [:= *static*_expression]

interface_declaration ::=
    interface_constant_declaration
    | interface_signal_dsclaration
    | interface_variable_declaration

interface_element ::= interface_declaration

interface_list ::= interface_element {, interface_element}

interface_signal_declaration ::=
    [**signal**] identifier_list : [mode] subtype_indication [**bus**]
    [:= static_expression]

interface_variable_declaration ::=
    [**variable**] identifier_list: [mode] subtype_indication
    [:= static_expression]

iteration_scheme ::=
    while condition
    | **for** *loop*_parameter_specification

label ::= identifier

letter ::= UPPER_CASE_LETTER | LOWER_CASE_LETTER

letter_or_digit ::= letter | digit

library_clause ::= **library** logical_name_list ;

library_unit ::= primary_unit | secondary_unit

literal ::=
    numeric_literal
    | enumeration_literal
    | string_literal
    | bit_string_literal
    | **null**

logical_name ::= identifier

logical_name_list ::= logical_name { , logical_name }

logical_operator ::= **and** | **or** | **nand** | **nor** | **xor**

loop_statement ::=
    [toop_label:]
    [iteration_scheme] **loop**
        sequence_of_statements
    end loop [*loop*_label] ;

miscellaneous_operator ::= ** | **abs | not** logical_name ::= identifier

mode ::= **in | out | inout | buffer | linkage**

multiplying_operator ::= * | / | **mod** | **rem**

name ::=
    simple_name
    | operator_symbol
    | selected_name
    | indexed_name
    | slice_name
    | attribute_name

next_statement ::= **next** [*loop*_label] [**when** condition];

null_statement ::= **null;**

numeric_literal ::= abstract_literal | physical_literal

object_declaration ::=
    constant_declaration
    | signal_declaration
    | variable_declaration

operator_symbol ::= string_literal

options ::= [**guarded**] [**transport**]

package_body ::=
    package body *pactege*_simple_name **is**
        package_body_declarative_part
    end [*package*_simple_name];

package_body_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | file_declaration
    | alias_declaration
    | use_clause

package_body_declarative_part ::= {
    package_body_declarative_item}

package_declaration ::=
    package identifier **is**
        package_declarative_part
    end [*package*_simple_name];

package_declarative_item ::=
    subprogram_declaration
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | file_declaration
    | alias_declaration

```
        | component_declaration
        | attribute_declaration
        | attribLite_specification
        | disconnection_specification
        | use_clause

package_declarative_part ::= { package_declarative_item )

parameter_specification ::= identifier in discrete_range

physical_literal ::= [ abstract_literal ] unit_name

physical_type_definition ::=
        range_constraint
                units
                        base_unit_declaration
                        { secondary_unit_declaration}
                end units

port_clause ::= port ( port_list ) ;

port_list ::= port_interface_list

port_map_aspect ::= port map ( port_association_list )

prefix ::= name | function_call

primary ::=
        name
        | literal
        | aggregate
        | function_call
        | qualified_expression
        | type_conversion
        | allocator
        | ( expression )

primary_unit ::=
        entity_declaration
        | configuration_declaration
        | package_declaration

procedure_call_statement ::= procedure_name [ ( actual_parameter_part ) ] ;

process_declarative_item ::=
```

```
        subprogram_declaration
        | subprogram_body
        | type_declaration
        | subtype_declaration
        | constant_declaration
        | variable_declaration
        | file_declaration
        | alias_declaration
        | attribute_declaration
        | attribute_specification
        | use_clause

process_declarative_part ::= { process_declarative_item}

process_statement ::=
        [ process_label: ]
                process [ ( sensitivity_list ) ]
                        process_declarative_part
                begin
                        process_statement_part
                end process [ process_label ];

process_statement_part ::= { sequential_statement}

qualified_expression ::=
        type_mark ' ( expression )
        | type_mark ' aggregate

range ::=
        range_attribute_name
        | simple_expression direction simple_expression

range_constraint ::= range range

record_type_definition ::=
        record
                element_declaration
                { element_declaration}
        end record

relation ::= simple_expression [ relational_operator simple_expression ]

relational_operator ::= = | /= | < | <= | > | >=

return_statement ::= return [ expression ] ;
```

```
scalar_type_definition ::=
        enumeration_type_definition
        | integer_type_definition
        | floating_type_definition
        | physical_type_definition

secondary_unit ::= architecture_body | package_body

secondary_unit_declaration ::= identifier = physical_literal;

selected_name ::= prefix. suffix

selected_signal_assignment ::=
        with expression select
                target <= options selected_waveforms ;

selected_waveforms ::= { waveform when choices ,} waveform when choices

sensitivity_clause ::= on sensitivity_list

sensitivity_list ::= signal_name {, signal_name }

sequence_of_statements ::= { sequential_statement}

sequential_statement ::=
        wait_statement
        | assertion_statement
        | signal_assignment_statement
        | variable_assignment_statement
        | procedure_call_statement
        | if_statement
        | case_statement
        | loop_statement
        | next_statement
        | exit_statement
        | retum_statement
        | null_statement

sign ::=+ | -

signal_assignment_statement ::= target <= [ transport ] waveform ;

signal_declaration ::=
        signal identifier_list : subtype_indication [ signal_kind ] [ := expression ];
```

signal_kind ::= **register | bus**

signal_list ::=
    *signal*_narne {, *signal*_name }
    | **others**
    | **all**

simple_expression ::= [sign] term { adding_operator term}

simple_name ::= identifier

slice_name ::= prefix (discrete_range)

string_literal ::= " { graphic_character} "

subprogram_body ::=
    subprogram_specification **is**
        subprogram_declarative_part
    begin
        subprogram_statement_part
    end [designator] ;

subprogram_declaration ::= subprogram_specification;

subprogram_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | file_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | use_clause

subprogram_declarative_part ::= { subprogram_declarative_item}

subprogram_specification ::=
    procedure designator [(formal_parameter_list)]
    | **function** designator [(formal_parameter_list)] **return** type_mark

subprogram_statement_part ::= { sequential_statement}

subtype_declaration ::= **subtype** identifier **is** subtype_indication ;

subtype_Indication ::= [*resolution_function*_name] type_mark [constraint]

suffix ::=
    simple_name
    | character_literal
    | operator_syrnbol
    | **all**

target ::= name | aggregate

term ::= factor { multiplying_operator factor}

timeout_clause := **for** *time*_expression

type_conversion ::= type_mark (expression)

type_declaration ::= full_type_declaration | incomplete_type_declaration

type_definition ::=
    scalar_type_definition
    | composite_type_definition
    | access_type_definition
    | file_type_definition

type_mark ::= *type*_name | *subtype*_name

unconstrained_array_definition ::=
    array (index_subtype_definition {, index_subtype_definition })
    of *element*_subtype_indication

use_clause ::= **use** selected_name {, selected_name } ;

variable_assignment_statement ::= target := expression ;

variable_declaration ::=
    variable identifier_list : subtype_indication [:= expression] ;

wait_statement ::=
    wait [sensitivity_clause] [condition_clause] [timeout_clause] ;

waveform ::= waveform_element {, waveform_element}

waveform_element ::=
    *value*_expression [**after** *time*_expression]
    | **null** [**after** *time*_expression

ΠΗΓΕΣ & ΒΙΒΛΙΟΓΡΑΦΙΑ

ΠΗΓΕΣ

1. http://el.wikipedia.org/wiki/VHDL#.CE.A0.CE.BB.CE.B5.CE.BF.CE.BD.CE.B5.CE.BA.CF.84.CE.AE.CE.BC.CE.B1.CF.84.CE.B1
2. http://en.wikipedia.org/wiki/Vhdl
3. http://homes.dsi.unimi.it/~pedersini/AD/VHDLReference_IEEE1076.pdf

ΒΙΒΛΙΟΓΡΑΦΙΑ

1. IEEE Standard VHDL Language Reference Manual, Std 1076-1987
2. IEEE Standard 1076 VHDL Tutorial
3. Barton, D., A first course in VHDL
4. Armstrong, J. R., Chip-level Modeling with VHDL
5. Charles H. Roth, Jr., Digital Systems Design Using VHDL
6. Volnei A. Pedroni, Circuit Design with VHDL