Γλώσσα Περιγραφής Υλικού
VHDL

Μιχάλης Ψαράκης

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ - ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ
ΠΜΣ «ΠΡΟΗΓΜΕΝΑ ΣΥΣΤΗΜΑΤΑ ΠΛΗΡΟΦΟΡΙΚΗΣ»

ΠΕΡΙΕΧΟΜΕΝΑ

Κεφάλαιο 1: Θεμελιώδεις Έννοιες

Κεφάλαιο 2: Βαθμωτοί Τύποι Δεδομένων και Πράξεις

Κεφάλαιο 3: Ακολουθιακές Προτάσεις

Κεφάλαιο 4: Σύνθετοι Τύποι Δεδομένων και Πράξεις

Κεφάλαιο 5: Βασικές Δομές Μοντελοποίησης

Κεφάλαιο 1: Θεμελιώδεις Έννοιες

Σε αυτό το εισαγωγικό κεφάλαιο, περιγράφουμε τι εννοούμε μοντελοποίηση ψηφιακού υπολογιστικού συστήματος και βλέπουμε γιατί η μοντελοποίηση και η προσομοίωση είναι ένα σημαντικό μέρος της σχεδιαστικής διαδικασίας. Βλέπουμε πως η *γλώσσα περιγραφής υλικού VHDL* μπορεί να χρησιμοποιηθεί για να μοντελοποιήσει ψηφιακά συστήματα και εισάγουμε κάποιες από τις βασικές έννοιες της γλώσσας. Ολοκληρώνουμε αυτό το κεφάλαιο με μια περιγραφή των βασικών λεξικολογικών και συντακτικών στοιχείων της γλώσσας, για να σχηματίσουμε μια βάση για τις λεπτομερείς περιγραφές των χαρακτηριστικών της γλώσσας που ακολουθούν σε επόμενα κεφάλαια.

1.1 Μοντελοποίηση Ψηφιακών Συστημάτων

Αν πρόκειται να συζητήσουμε το αντικείμενο της μοντελοποίησης ψηφιακών συστημάτων, πρέπει πρώτα να συμφωνήσουμε τι είναι ένα *ψηφιακό σύστημα*. Διαφορετικοί μηχανικοί σχεδίασης (design engineers) θα έδιναν διαφορετικούς ορισμούς, ανάλογα με το υπόβαθρό τους και το πεδίο στο οποίο εργάζονται. Μερικοί μπορεί να θεωρήσουν ότι ένα και μόνο κύκλωμα VLSI (Very Large Scale of Integration) είναι ένα αυτοτελές ψηφιακό σύστημα. Άλλοι μπορεί να το αντιμετωπίζουν ευρύτερα και να σκέφτονται έναν πλήρη υπολογιστή, συσκευασμένο σε ένα κουτί μαζί με περιφερειακούς ελεγκτές και άλλες διασυνδέσεις.

Για τους σκοπούς αυτού του μαθήματος, συμπεριλαμβάνουμε στην έννοια του ψηφιακού συστήματος οποιοδήποτε ψηφιακό κύκλωμα επεξεργάζεται ή αποθηκεύει πληροφορία. Λαμβάνουμε συνεπώς υπόψιν τόσο το σύστημα συνολικά όσο και τα διάφορα μέρη από τα οποία κατασκευάζεται. Έτσι, η συζήτησή μας καλύπτει ένα εύρος συστημάτων από τις πύλες (συστατικά στοιχεία χαμηλού επιπέδου) μέχρι τις λειτουργικές μονάδες υψηλού επιπέδου.

Αν πρόκειται να καλύψουμε όλο αυτό το εύρος των ψηφιακών συστημάτων, πρέπει να γνωρίζουμε την πολυπλοκότητα με την οποία ερχόμαστε αντιμέτωποι. Επειδή δεν είναι εύκολο να κατανοήσουμε τόσο πολύπλοκα συστήματα στο σύνολό τους απαιτείται να βρούμε μεθόδους που να μας διευκολύνουν να σχεδιάζουμε συστατικά και συστήματα που ικανοποιούν συγκεκριμένες προδιαγραφές.

Ο πιο σίγουρος τρόπος για να αντεπεξέλθουμε σε αυτήν την πρόκληση είναι να υιοθετήσουμε μια *συστηματική μεθοδολογία σχεδίασης (systematic methodology of design)*. Αν ξεκινήσουμε με ένα κείμενο προδιαγραφών για το σύστημα, μπορούμε να σχεδιάσουμε μια αφηρημένη δομή που ικανοποιεί τις προδιαγραφές. Μπορούμε στη συνέχεια να αναλύσουμε αυτή τη δομή σε μια συλλογή συστατικών που αλληλεπιδρούν για να εκτελέσουν την ίδια συνάρτηση. Καθένα απ' αυτά τα συστατικά μπορεί με τη σειρά του να αναλυθεί μέχρι να φτάσουμε σε ένα επίπεδο όπου υπάρχουν κάποια διαθέσιμα, θεμελιώδη συστατικά που εκτελούν μια ζητούμενη συνάρτηση. Το αποτέλεσμα αυτής της διαδικασίας είναι ένα ιεραρχικά δομημένο σύστημα, κατασκευασμένο από θεμελιώδη στοιχεία.

Το πλεονέκτημα αυτής της μεθοδολογίας είναι ότι κάθε υποσύστημα μπορεί να σχεδιαστεί ανεξάρτητα από τα άλλα. Όταν χρησιμοποιούμε ένα υποσύστημα, μπορούμε να το σκεφτόμαστε σε μια αφηρημένη μορφή αντί να πρέπει να λαμβάνουμε υπόψιν τη λεπτομερή σύνθεσή του. Έτσι, σε κάθε συγκεκριμένο στάδιο της σχεδιαστικής διαδικασίας, χρειάζεται μόνο να δίνουμε προσοχή στην ποσότητα της πληροφορίας που σχετίζεται με το τρέχον σημείο της σχεδίασης. Με αυτόν τον τρόπο, προστατευόμαστε από το να κατακλυστούμε από τεράστια ποσότητα πληροφορίας.

Χρησιμοποιούμε τον όρο *μοντέλο (model)* εννοώντας τον τρόπο που κατανοούμε ένα σύστημα. Το μοντέλο αναπαριστάνει εκείνη την πληροφορία που είναι σχετική και δημιουργεί μια αφαίρεση από την άσχετη λεπτομέρεια. Ως συνέπεια αυτού μπορεί να υπάρχουν πολλά διαφορετικά μοντέλα του ίδιου συστήματος, αφού διαφορετική πληροφορία μπορεί να θεωρηθεί ως σχετική σε διαφορετικό περιβάλλον. Για παράδειγμα, ένα είδος μοντέλου μπορεί να επικεντρώνεται στην αναπαράσταση της συνάρτησης του συστήματος, ενώ ένα άλλο είδος μπορεί να αναπαριστάνει τον τρόπο με τον οποίο το σύστημα αναλύεται σε υποσυστήματα. Θα επανέλθουμε σ' αυτή την ιδέα με περισσότερες λεπτομέρειες αργότερα.

Υπάρχει ένας αριθμός σημαντικών κινήτρων για τον φορμαλισμό αυτής της ιδέας του μοντέλου:

- Πρώτον, για να σχεδιαστεί ένα ψηφιακό σύστημα, πρέπει αρχικά να προδιαγραφούν οι απαιτήσεις από αυτό το σύστημα. Η δουλειά των μηχανικών είναι να σχεδιάσουν ένα σύστημα που ικανοποιεί αυτές τις προδιαγραφές. Για να το κάνουν αυτό, πρέπει να τους δοθεί μια κατανοητή *περιγραφή των απαιτήσεων (requirements description)*, ελπίζοντας ότι αυτό γίνεται με έναν τρόπο που τους αφήνει την ελευθερία να διερευνήσουν εναλλακτικές υλοποιήσεις και να επιλέξουν την καλύτερη σύμφωνα με κάποια κριτήρια. Ένα από τα προβλήματα που συχνά εμφανίζεται είναι ότι οι απαιτήσεις δεν είναι πλήρεις και εκφράζονται με ασάφεια, και ότι ο πελάτης και οι μηχανικοί σχεδίασης διαφωνούν σ' αυτό που εννοεί το κείμενο των απαιτήσεων. Αυτό το πρόβλημα μπορεί να αποφευχθεί χρησιμοποιώντας ένα τυπικό μοντέλο για την περιγραφή των απαιτήσεων.
- Ένας δεύτερος λόγος για τη χρήση τυπικών μοντέλων είναι για να βοηθήσει το χρήστη να κατανοήσει τη συνάρτηση ενός συστήματος. Ο σχεδιαστής δεν μπορεί πάντα να προβλέψει κάθε πιθανό τρόπο με τον οποίο μπορεί να χρησιμοποιηθεί ένα σύστημα, και δεν είναι επίσης ικανός να απαριθμήσει όλες τις πιθανές

συμπεριφορές του συστήματος. Αν ο σχεδιαστής παρέχει ένα μοντέλο, ο χρήστης μπορεί να το ελέγξει έναντι οποιουδήποτε συνόλου εισόδων και να καθορίσει πως συμπεριφέρεται το σύστημα σ' αυτό το περιβάλλον. Έτσι ένα τυπικό μοντέλο είναι ένα πολύτιμο εργαλείο για την *τεκμηρίωση ενός συστήματος*.

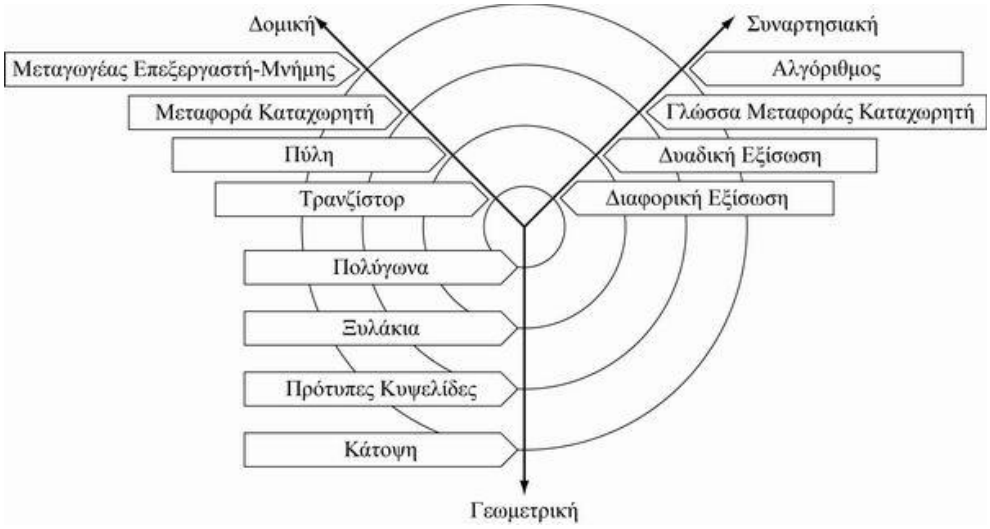
- Ένα τρίτο κίνητρο για τη μοντελοποίηση είναι να καταστήσει δυνατή τη *δοκιμή (testing)* και την *επαλήθευση (verification)* μιας σχεδίασης χρησιμοποιώντας *προσομοίωση (simulation)*. Αν ξεκινήσουμε με ένα μοντέλο απαιτήσεων που ορίζει τη συμπεριφορά ενός συστήματος, μπορούμε να προσομοιώσουμε τη συμπεριφορά χρησιμοποιώντας εισόδους δοκιμής και καταγράφοντας τις παραγόμενες εξόδους του συστήματος. Σύμφωνα με τη σχεδιαστική μας μεθοδολογία, μπορούμε στη συνέχεια να σχεδιάσουμε ένα κύκλωμα από υποσυστήματα, καθένα με το δικό του μοντέλο συμπεριφοράς. Μπορούμε να προσομοιώσουμε αυτό το σύνθετο σύστημα με τις ίδιες εισόδους και να συγκρίνουμε τις εξόδους με αυτές τις προηγούμενης προσομοίωσης. Αν είναι οι ίδιες, γνωρίζουμε ότι το σύνθετο σύστημα ικανοποιεί τις προδιαγραφές για τις περιπτώσεις που δοκιμάστηκαν. Διαφορετικά γνωρίζουμε ότι χρειάζεται κάποια αναθεώρηση της σχεδίασης. Μπορούμε να συνεχίσουμε αυτή τη διαδικασία μέχρι να φτάσουμε στο κατώτερο επίπεδο της σχεδιαστικής μας ιεραρχίας, όπου τα συστατικά είναι πραγματικές συσκευές των οποίων γνωρίζουμε τη συμπεριφορά. Συνεπώς, όταν κατασκευαστεί το πραγματικό σύστημα, οι είσοδοι και έξοδοι δοκιμής από την προσομοίωση μπορούν να χρησιμοποιηθούν για να επαληθεύσουν ότι το φυσικό κύκλωμα λειτουργεί σωστά. Αυτή η προσέγγιση της δοκιμής και της επαλήθευσης υποθέτει βεβαίως ότι οι είσοδοι δοκιμής καλύπτουν όλες τις περιπτώσεις στις οποίες το τελικό κύκλωμα θα χρησιμοποιηθεί. Το θέμα της *κάλυψης δοκιμής (test coverage)* είναι ένα πολύπλοκο πρόβλημα από μόνο του και μια ενεργή ερευνητική περιοχή.
- Ένα τέταρτο κίνητρο για τη μοντελοποίηση είναι να καταστήσει δυνατή την *τυπική επαλήθευση (formal verification)* της ορθότητας μιας σχεδίασης. Η τυπική επαλήθευση απαιτεί μια μαθηματική περιγραφή της απαιτούμενης συνάρτησης του συστήματος. Αυτή η περιγραφή μπορεί να εκφραστεί με χρήση της σημειογραφίας ενός συστήματος τυπικής λογικής (formal logic system), όπως η χρονική λογική (temporal logic). Η τυπική επαλήθευση απαιτεί επίσης ένα μαθηματικό ορισμό της σημασίας της γλώσσας μοντελοποίησης ή της σημειογραφίας που χρησιμοποιείται για την περιγραφή της σχεδίασης. Η διαδικασία της επαλήθευσης περιλαμβάνει την εφαρμογή των κανόνων της συμπερασματολογίας του λογικού συστήματος για να αποδείξει ότι η σχεδίαση συνεπάγεται την απαιτούμενη συνάρτηση. Ενώ η τυπική επαλήθευση δεν είναι ακόμη σε καθημερινή χρήση, είναι μια ενεργή ερευνητική περιοχή. Έχουν παρουσιαστεί σημαντικές εφαρμογές τεχνικών τυπικής επαλήθευσης σε πραγματικά σχεδιαστικά συστήματα, και τα αποτελέσματα ήταν άκρως ενθαρρυντικά.
- Ένα τελευταίο, αλλά εξίσου σημαντικό κίνητρο για τη μοντελοποίηση είναι να καταστήσει δυνατή την *αυτόματη σύνθεση (automatic synthesis)* των κυκλωμάτων. Αν μπορούμε να προδιαγράψουμε τυπικά τη συνάρτηση που απαιτείται από ένα σύστημα, είναι θεωρητικά εφικτό να μεταφράσουμε αυτήν την προδιαγραφή σε ένα κύκλωμα που εκτελεί τη συνάρτηση. Το πλεονέκτημα αυτής της προσέγγισης είναι ότι το ανθρώπινο κόστος της σχεδίασης μειώνεται, και οι μηχανικοί είναι ελεύθεροι να διερευνήσουν εναλλακτικές επιλογές σχεδίασης αντί να είναι υποχρεωμένοι να καταναλώσουν χρόνο μελετώντας τις σχεδιαστικές λεπτομέρειες. Επίσης, υπάρχει μικρότερο περιθώριο για σφάλματα που εισάγονται σε μια σχεδίαση και δεν ανιχνεύονται. Αν αυτοματοποιήσουμε τη μετάφραση από την προδιαγραφή στη σχεδίαση, μπορούμε να είμαστε περισσότερο σίγουροι ότι το παραγόμενο κύκλωμα είναι σωστό.

Ο κοινός παρονομαστής πίσω απ' όλα αυτά τα επιχειρήματα είναι ότι θέλουμε να πετύχουμε τη μέγιστη αξιοπιστία της σχεδιαστικής διαδικασίας με το ελάχιστο κόστος και το μικρότερο χρόνο σχεδίασης. Χρειάζεται να βεβαιώσουμε ότι οι απαιτήσεις προδιαγράφονται και κατανοούνται ξεκάθαρα, ότι τα υποσυστήματα χρησιμοποιούνται σωστά και ότι οι σχεδιάσεις ικανοποιούν τις προδιαγραφές. Ένας βασικός παράγοντας που μπορεί να οδηγήσει σε υπέρβαση του αρχικού κόστους είναι η ανάγκη αναθεώρησης μιας σχεδίασης μετά την κατασκευή της για να διορθωθούν σφάλματα. Αποφεύγοντας τα σφάλματα και παρέχοντας καλύτερα εργαλεία στη σχεδιαστική διαδικασία, τα κόστη και οι καθυστερήσεις μπορούν να περιοριστούν.

1.2 Πεδία και Επίπεδα Μοντελοποίησης

Στην προηγούμενη ενότητα, αναφέραμε ότι μπορεί να υπάρχουν διαφορετικά μοντέλα ενός συστήματος, όπου το καθένα επικεντρώνεται σε διαφορετική πλευρά του συστήματος. Μπορούμε να κατηγοριοποιήσουμε αυτά τα μοντέλα σε τρία διαφορετικά *πεδία (domains)*: *συνάρτησης (function)*, *δομής (structure)* και *γεωμετρίας (geometry)*. Το πεδίο συνάρτησης ασχολείται με τις λειτουργίες που εκτελεί το σύστημα. Υπό μια έννοια, αυτό είναι το πιο αφηρημένο πεδίο περιγραφής, αφού δεν ασχολείται με το πως υλοποιείται η συνάρτηση. Το πεδίο δομής ασχολείται με τον τρόπο που το σύστημα απαρτίζεται από διασυνδεδεμένα υποσυστήματα. Το πεδίο γεωμετρίας ασχολείται με τον τρόπο που διατάσσεται το σύστημα στο φυσικό χώρο.

ΕΙΚΟΝΑ 1-1



Πεδία και επίπεδα αφαίρεσης. Οι ακτινικοί άξονες δείχνουν τα τρία διαφορετικά πεδία μοντελοποίησης. Οι ομόκεντροι δακτύλιοι δείχνουν τα επίπεδα αφαίρεσης, με τα πιο αφηρημένα επίπεδα στο εξωτερικό και τα πιο λεπτομερή επίπεδα προς το κέντρο.

Καθένα απ' αυτά τα πεδία μπορεί να διαιρεθεί σε *επίπεδα αφαίρεσης (levels of abstraction)*. Στο κορυφαίο επίπεδο, θεωρούμε μια γενική άποψη της συνάρτησης, της δομής ή της γεωμετρίας, και στα χαμηλότερα επίπεδα εισάγουμε διαδοχικά μεγαλύτερη λεπτομέρεια. Η Εικόνα 1-1 (την εισήγαγαν για πρώτη φορά οι Gajski και Kuhn) αναπαριστά τα πεδία σε τρεις ανεξάρτητους άξονες και τα επίπεδα της αφαίρεσης με ομόκεντρους κύκλους που διασταυρώνονται με κάθε άξονα.

Ας δούμε αυτήν τη κατηγοριοποίηση με περισσότερη λεπτομέρεια, δείχνοντας πως σε κάθε επίπεδο μπορούμε να δημιουργήσουμε μοντέλα για κάθε πεδίο. Σαν ένα παράδειγμα, θεωρούμε ένα σύστημα μικροελεγκτή σε ένα μόνο ολοκληρωμένο κύκλωμα (single-chip microcontroller) που χρησιμοποιείται για να ελέγχει κάποιο όργανο μετρήσεων (measurement instrument), και το οποίο δέχεται εισόδους δεδομένων και απεικονίζει τις εξόδους σε κάποια οθόνη.

Στο πιο αφηρημένο επίπεδο, η συνάρτηση του συνολικού συστήματος μπορεί να περιγραφεί με όρους ενός αλγορίθμου, παρόμοια με τον αλγόριθμο ενός προγράμματος υπολογιστή. Αυτό το επίπεδο συναρτησιακής μοντελοποίησης συχνά ονομάζεται *μοντελοποίηση συμπεριφοράς (behavioral modeling)*, ένας όρος που θα χρησιμοποιήσουμε αργότερα όταν παρουσιάσουμε αφηρημένες περιγραφές της συνάρτησης ενός συστήματος. Ένας πιθανός αλγόριθμος για τον ελεγκτή του οργάνου φαίνεται στην Εικόνα 1-2. Αυτό το μοντέλο περιγράφει πως ο ελεγκτής επαναληπτικά σαρώνει κάθε είσοδο δεδομένων και απεικονίζει στην οθόνη μια διαβαθμισμένη δεκαδική αναπαράσταση της τιμής της εισόδου.

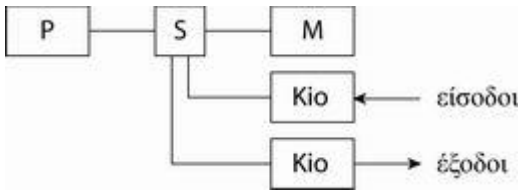
ΕΙΚΟΝΑ 1-2

```
loop
    for each data input loop
        read the value on this input;
        scale the value using the current scale factor for this input;
        convert the scaled value to a decimal string;
        write the string to the display output corresponding to this input;
    end loop;
    wait for 10 ms;
end loop;
```

Ένας αλγόριθμος για τον ελεγκτή οργάνου μέτρησης.

Σ' αυτό το υψηλό επίπεδο αφαίρεσης, η δομή ενός συστήματος μπορεί να περιγραφεί σαν μια διασύνδεση συστατικών όπως επεξεργαστές, μνήμες και συσκευές εισόδου/εξόδου. Αυτό το επίπεδο ονομάζεται μερικές φορές *επίπεδο Μεταγωγής Επεξεργαστή Μνήμης (Processor Memory Switch – PMS)*. Η Εικόνα 1-3 δείχνει ένα δομικό μοντέλο του ελεγκτή του οργάνου που σχεδιάστηκε χρησιμοποιώντας αυτή τη σημειογραφία. Αποτελείται από έναν επεξεργαστή που συνδέεται μέσω ενός μεταγωγέα (switch) με μια μνήμη και με δυο ελεγκτές για τις εισόδους δεδομένων και τις εξόδους απεικόνισης.

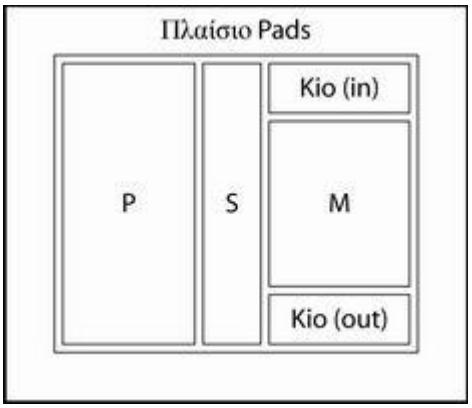
ΕΙΚΟΝΑ 1-3



Ένα μοντέλο PMS για τη δομή του ελεγκτή. Αποτελείται από έναν επεξεργαστή (P), μια μνήμη (M), ένα μεταγωγέα διασύνδεσης (S) και δύο ελεγκτές εισόδου/εξόδου (Kio).

Στο πεδίο γεωμετρίας σ' αυτό το υψηλό επίπεδο αφαίρεσης, ένα σύστημα που θα υλοποιηθεί σαν ένα κύκλωμα VLSI μπορεί να μοντελοποιηθεί χρησιμοποιώντας μια *κάτοψη (floorplan)*. Αυτή δείχνει πως τα στοιχεία που περιγράφονται στο δομικό μοντέλο τοποθετούνται στο ολοκληρωμένο κύκλωμα πυριτίου. Η Εικόνα 1-4 δείχνει μια πιθανή κάτοψη για το ολοκληρωμένο κύκλωμα του ελεγκτή του οργάνου. Υπάρχουν ανάλογες γεωμετρικές περιγραφές για συστήματα που κατασκευάζονται με άλλη τεχνολογία. Για παράδειγμα, ένα σύστημα προσωπικού υπολογιστή μπορεί να μοντελοποιηθεί στο υψηλότερο επίπεδο του πεδίου γεωμετρίας με ένα διάγραμμα συναρμολόγησης που δείχνει τις θέσεις της μητρικής κάρτας (motherboard) και των συνδεόμενων πλακετών επέκτασης (expansion boards) στο κουτί του προσωπικού υπολογιστή.

ΕΙΚΟΝΑ 1-4



Ένα μοντέλο κάτοψης για τη γεωμετρία του ελεγκτή.

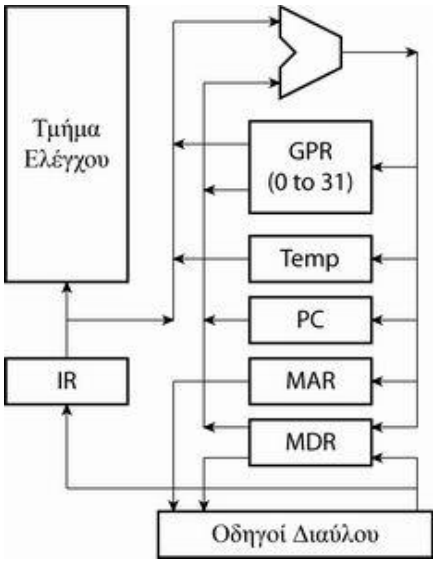
Το επόμενο επίπεδο αφαίρεσης στη μοντελοποίηση, που φαίνεται στον δεύτερο δακτύλιο στην Εικόνα 1-1, περιγράφει το σύστημα με όρους μονάδων αποθήκευσης και μετασχηματισμού δεδομένων. Στο πεδίο δομής, αυτό ονομάζεται συχνά *επίπεδο μεταφοράς καταχωρητή (register-transfer)*, αποτελείται από μία *διαδρομή δεδομένων (data path)* και ένα *τμήμα ελέγχου (control section)*. Η διαδρομή δεδομένων περιέχει καταχωρητές αποθήκευσης δεδομένων, και τα δεδομένα μεταφέρονται μεταξύ τους μέσω μονάδων μετασχηματισμού. Το τμήμα ελέγχου δημιουργεί την ακολουθία των λειτουργιών της διαδρομής δεδομένων. Για παράδειγμα, ένα δομικό μοντέλο επιπέδου μεταφοράς καταχωρητή του επεξεργαστή που είναι ενσωματωμένος μέσα στον ελεγκτή μας φαίνεται στην Εικόνα 1-5.

Στο πεδίο συνάρτησης, χρησιμοποιείται συχνά μια *γλώσσα μεταφοράς καταχωρητή (register-transfer language – RTL)* για την προδιαγραφή των λειτουργιών ενός συστήματος. Η αποθήκευση δεδομένων αναπαριστάνεται χρησιμοποιώντας μεταβλητές καταχωρητών, και οι μετασχηματισμοί αναπαριστάνονται με αριθμητικούς και λογικούς τελεστές. Για παράδειγμα, ένα μοντέλο RTL για τον επεξεργαστή του ελεγκτή του παραδείγματός μας μπορεί να περιλαμβάνει την ακόλουθη περιγραφή:

```
MAR ← PC, memory_read ← 1
PC ← PC + 1
wait until ready = 1
IR ←memory_data
memory_read ← 0
```

Αυτό το τμήμα του μοντέλου περιγράφει τις λειτουργίες που περιλαμβάνονται στην προσκόμιση (fetching) μιας εντολής από τη μνήμη. Τα περιεχόμενα του καταχωρητή PC (program counter) μεταφέρονται στον καταχωρητή διεύθυνσης μνήμης (memory address register) και το σήμα memory_read ενεργοποιείται. Έπειτα, η τιμή του καταχωρητή PC ενημερώνεται (αυξάνεται στην περίπτωση αυτή) και αποθηκεύεται πίσω στον καταχωρητή PC. Όταν η είσοδος ready από τη μνήμη ενεργοποιηθεί, η τιμή στην έξοδο δεδομένων της μνήμης (memory_data) μεταφέρεται στον καταχωρητή εντολής IR (instruction register). Τέλος, το σήμα memory_read απενεργοποιείται.

ΕΙΚΟΝΑ 1-5



Ένα δομικό μοντέλο επιπέδου μεταφοράς καταχωρητή του επεξεργαστή του ελεγκτή. Αποτελείται από ένα αρχείο καταχωρητών γενικού σκοπού (general-purpose register – GPR – file); καταχωρητές για τον μετρητή προγράμματος (program counter – PC), τη διεύθυνση μνήμης (memory address – MAR), τα δεδομένα μνήμης (memory data – MDR), προσωρινές τιμές (Temp) και προσκομισμένες εντολές (instruction register – IR); μια αριθμητική μονάδα; οδηγούς διαύλου (bus drivers) και το τμήμα ελέγχου.

Στο πεδίο γεωμετρίας, το είδος του μοντέλου που χρησιμοποιείται εξαρτάται από το φυσικό μέσο. Στο παράδειγμά μας, πρότυπες κυψελίδες βιβλιοθήκης (standard library cells) μπορεί να χρησιμοποιηθούν για να υλοποιήσουν τους καταχωρητές και τις μονάδες μετασχηματισμού δεδομένων, και αυτές πρέπει να τοποθετηθούν στις περιοχές που τους έχουν αντιστοιχηθεί στην κάτοψη του ολοκληρωμένου κυκλώματος.

Στο τρίτο επίπεδο αφαίρεσης που φαίνεται στην Εικόνα 1-1 βρίσκεται το συμβατικό επίπεδο λογικής. Σ' αυτό το επίπεδο, η δομή μοντελοποιείται χρησιμοποιώντας διασυνδέσεις πυλών, και η συνάρτηση μοντελοποιείται από δυαδικές εξισώσεις (Boolean equations) ή πίνακες αληθείας (truth tables). Στο φυσικό μέσο ενός προσαρμοσμένου ολοκληρωμένου κυκλώματος (custom integrated circuit), η γεωμετρία μπορεί να μοντελοποιηθεί χρησιμοποιώντας μία σημειογραφία εικονικού πλέγματος ή «ξυλάκια» (sticks).

Στο πιο λεπτομερές επίπεδο αφαίρεσης, μπορούμε να μοντελοποιήσουμε τη δομή χρησιμοποιώντας μεμονωμένα τρανζίστορ, τη συνάρτηση χρησιμοποιώντας τις διαφορικές εξισώσεις που συσχετίζουν την τάση και την ένταση του ρεύματος στο κύκλωμα, και τη γεωμετρία χρησιμοποιώντας πολύγωνα για κάθε στρώμα μάσκας ενός ολοκληρωμένου κυκλώματος. Οι περισσότεροι σχεδιαστές δεν χρειάζεται να δουλεύουν σ' αυτό το λεπτομερές επίπεδο, καθώς είναι διαθέσιμα σχεδιαστικά εργαλεία που αυτοματοποιούν τη μετάφραση από ένα υψηλότερο επίπεδο.

1.3 Γλώσσες Μοντελοποίησης

Στην προηγούμενη ενότητα, είδαμε ότι διαφορετικοί τύποι μοντέλων μπορούν να περιγραφούν για την αναπαράσταση των διαφόρων επιπέδων της συνάρτησης, της δομής και της φυσικής διάταξης ενός συστήματος. Υπάρχουν επίσης διαφορετικοί τρόποι έκφρασης αυτών των μοντέλων, ανάλογα με τη χρήση που γίνεται στο μοντέλο.

Σαν ένα παράδειγμα, θεωρείστε τους τρόπους με τους οποίους ένα δομικό μοντέλο μπορεί να εκφραστεί. Μια συνηθισμένη μορφή είναι το σχηματικό διάγραμμα ενός κυκλώματος. Γραφικά σύμβολα χρησιμοποιούνται για την αναπαράσταση υποσυστημάτων, και στιγμιότυπα (instances) αυτών συνδέονται χρησιμοποιώντας γραμμές που αναπαριστάνουν καλώδια. Αυτή η γραφική αναπαράσταση είναι γενικά αυτή που προτιμούν οι σχεδιαστές. Ωστόσο, η ίδια δομική πληροφορία μπορεί να αναπαρασταθεί με κείμενο στη μορφή μιας λίστας δικτυωμάτων (net list).

Όταν μεταφερόμαστε στο πεδίο συνάρτησης, συνήθως βλέπουμε σημειογραφίες κειμένου να χρησιμοποιούνται για μοντελοποίηση. Μερικές απ' αυτές προτίθενται για χρήση σαν γλώσσες προδιαγραφών, για να ικανοποιήσουν την ανάγκη περιγραφής της λειτουργίας ενός συστήματος χωρίς να δείχνουν λεπτομέρειες για το πως θα μπορούσε να υλοποιηθεί. Αυτές οι σημειογραφίες βασίζονται συνήθως σε τυπικούς μαθηματικούς συμβολισμούς, όπως η χρονική λογική ή οι αφηρημένες μηχανές καταστάσεων. Άλλες σημειογραφίες χρησιμοποιούνται για την προσομοίωση του συστήματος για σκοπούς δοκιμής και επαλήθευσης και βασίζονται τυπικά σε συμβατικές γλώσσες προγραμματισμού. Ακόμη άλλες σημειογραφίες προσανατολίζονται προς την σύνθεση υλικού και έχουν συνήθως ένα πιο περιορισμένο σύνολο δυνατοτήτων μοντελοποίησης, αποφεύγοντας έτσι μερικές δομές γλωσσών προγραμματισμού οι οποίες είναι δύσκολο να μεταφραστούν σε υλικό.

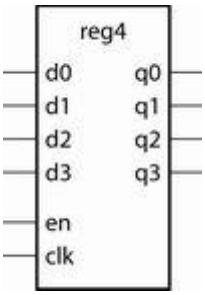
Ο σκοπός αυτών των σημειώσεων είναι να περιγράψουν τη ***γλώσσα μοντελοποίησης VHDL***. Η VHDL παρέχει τις παρακάτω δυνατότητες:

- Επιτρέπει την περιγραφή της δομής και της συνάρτησης σε πλήθος επιπέδων, από το πιο αφηρημένο επίπεδο μέχρι κάτω στο επίπεδο της πύλης.
- Παρέχει έναν μηχανισμό ιδιοτήτων (attributes) που μπορεί να χρησιμοποιηθεί για να ενημερώσει ένα μοντέλο με πληροφορία στο πεδίο γεωμετρίας.
- Προορίζεται επίσης ως μια γλώσσα μοντελοποίησης για προδιαγραφή και προσομοίωση.
- Μπορούμε επίσης να τη χρησιμοποιήσουμε για σύνθεση υλικού αν περιοριστούμε σε ένα υποσύνολο που μπορεί να μεταφραστεί αυτόματα σε υλικό.

1.4 Έννοιες Μοντελοποίησης της VHDL

Στην προηγούμενη ενότητα, είδαμε τρία πεδία μοντελοποίησης (modeling domains): συνάρτησης (function), δομής (structure) και γεωμετρίας (geometry). Σε αυτήν την ενότητα, κοιτάζουμε τις βασικές έννοιες μοντελοποίησης σε κάθε ένα από αυτά τα πεδία και εισάγουμε τα αντίστοιχα στοιχεία της VHDL για την περιγραφή τους. Αυτό θα μας δώσει μια πρώτη ιδέα για τη VHDL και μια βάση πάνω στην οποία θα δουλέψουμε στα επόμενα κεφάλαια. Σαν παράδειγμα, βλέπουμε τρόπους περιγραφής ενός καταχωρητή τεσσάρων ψηφίων, που παρουσιάζεται στην Εικόνα 1-6.

ΕΙΚΟΝΑ 1-6



Μια μονάδα καταχωρητή τεσσάρων ψηφίων. Ο καταχωρητής ονομάζεται reg4 *και έχει έξι εισόδους,* d0, d1, d2, d3, en *και* clk*, και τέσσερις εξόδους,* q0, q1, q2 *και q3.*

Με χρήση ορολογίας της VHDL, καλούμε τη μονάδα reg4 *οντότητα (entity)* σχεδίασης, και τις εισόδους και τις εξόδους *θύρες (ports)*. Η Εικόνα 1-7 παρουσιάζει μια VHDL περιγραφή της διασύνδεσης σε αυτήν την οντότητα. Αυτό είναι ένα παράδειγμα μιας *δήλωσης οντότητας (entity declaration)*. Αυτή εισάγει ένα όνομα για την οντότητα και δημιουργεί έναν κατάλογο των θυρών εισόδου και εξόδου, προσδιορίζοντας ότι μεταφέρουν τιμές bit (‘0’ ή ‘1’) προς και από την οντότητα. Από αυτό βλέπουμε ότι μια δήλωση οντότητας περιγράφει την εξωτερική όψη της οντότητας.

ΕΙΚΟΝΑ 1-7

```
entity reg4 is
    port (  d0, d1, d2, d3, en, clk : in bit;
            q0, q1, q2, q3 : out bit );
end entity reg4;
```

Μια VHDL περιγραφή της οντότητας ενός καταχωρητή τεσσάρων ψηφίων.

1.4.1 Στοιχεία Συμπεριφοράς

Στη VHDL, μια περιγραφή της εσωτερικής υλοποίησης μιας οντότητας καλείται *σώμα αρχιτεκτονικής (architecture body)* της οντότητας. Μπορεί να υπάρχει ένας αριθμός διαφορετικών σωμάτων αρχιτεκτονικής μιας οντότητας, που αντιστοιχούν σε εναλλακτικές υλοποιήσεις που εκτελούν την ίδια λειτουργία. Μπορούμε να γράψουμε ένα σώμα αρχιτεκτονικής *συμπεριφοράς (behavioral)* μιας οντότητας, που περιγράφει τη λειτουργία με έναν αφηρημένο τρόπο. Ένα τέτοιο σώμα αρχιτεκτονικής περιέχει μόνο *προτάσεις διεργασιών (process statements)*, οι οποίες είναι συλλογές ενεργειών που θα εκτελεστούν στη σειρά. Αυτές οι ενέργειες καλούνται *ακολουθιακές προτάσεις (sequential statements)* και μοιάζουν πολύ με τους τύπους των προτάσεων που βλέπουμε σε μια συμβατική γλώσσα προγραμματισμού. Οι τύποι των ενεργειών που μπορούν να πραγματοποιηθούν περιλαμβάνουν υπολογισμούς εκφράσεων, αναθέσεις τιμών σε μεταβλητές, υπό συνθήκη εκτέλεση, επαναλαμβανόμενη εκτέλεση και κλήσεις υποπρογραμμάτων. Επιπλέον, υπάρχει μια ακολουθιακή πρόταση που είναι μοναδική στις γλώσσες μοντελοποίησης

υλικού, η *πρόταση ανάθεσης σήματος (signal assignment)*. Αυτή είναι παρόμοια με την ανάθεση μεταβλητής, εκτός από το γεγονός ότι προκαλεί την ενημέρωση της τιμής ενός σήματος σε κάποια μελλοντική χρονική στιγμή.

Για να διευκρινίσουμε αυτές τις έννοιες, ας δούμε ένα σώμα αρχιτεκτονικής συμπεριφοράς για την οντότητα reg4 που παρουσιάζεται στην Εικόνα 1-8. Σε αυτό το σώμα αρχιτεκτονικής, το κομμάτι μετά την πρώτη λέξη κλειδί **begin** περιλαμβάνει μια πρόταση διεργασίας (process statement), που περιγράφει με ποιο τρόπο συμπεριφέρεται ο καταχωρητής. Ξεκινάει με το όνομα της διεργασίας, storage, και τελειώνει με τη λέξη κλειδί **end process.**

ΕΙΚΟΝΑ 1-8

```
architecture behav of reg4 is
begin
    storage : process is
        variable stored_d0, stored_d1, stored_d2, stored_d3 : bit;
    begin
        if en = '1' and clk = '1' then
            stored_d0 := d0;
            stored_d1 := d1;
            stored_d2 := d2;
            stored_d3 := d3;
        end if;
        q0 <= stored_d0 after 5 ns;
        q1 <= stored_d1 after 5 ns;
        q2 <= stored_d2 after 5 ns;
        q3 <= stored_d3 after 5 ns;
        wait on d0, d1, d2, d3, en, clk;
    end process storage;
end architecture behav;
```

Ένα σώμα αρχιτεκτονικής συμπεριφοράς της οντότητας reg4.

Η πρόταση διεργασίας ορίζει μια ακολουθία ενεργειών οι οποίες λαμβάνουν χώρα όταν το σύστημα προσομοιώνεται. Αυτές οι ενέργειες ελέγχουν με ποιον τρόπο αλλάζουν στο χρόνο οι τιμές στις θύρες της οντότητας. Αυτή η διεργασία μπορεί να τροποποιήσει τις τιμές των θυρών της οντότητας με χρήση προτάσεων ανάθεσης σήματος.

Ο τρόπος με το οποίο δουλεύει αυτή η διεργασία είναι ο ακόλουθος. Όταν η προσομοίωση ξεκινάει, οι τιμές των σημάτων τίθενται στο '0', και η διεργασία ενεργοποιείται. Οι μεταβλητές της διεργασίας (που παρουσιάζονται σε λίστα μετά τη λέξη κλειδί **variable**) αρχικοποιούνται στο '0', και στη συνέχεια οι προτάσεις εκτελούνται με τη σειρά. Η πρώτη πρόταση είναι μια συνθήκη που ελέγχει αν οι τιμές των σημάτων en και clk είναι και οι δύο '1'. Αν είναι, οι προτάσεις μεταξύ των λέξεων κλειδιών **then** και **end if** εκτελούνται, ενημερώνοντας τις μεταβλητές των διεργασιών χρησιμοποιώντας τις τιμές των σημάτων εισόδου. Μετά την υπό συνθήκη πρόταση if, υπάρχουν τέσσερις προτάσεις ανάθεσης σήματος που προκαλούν την ενημέρωση των σημάτων εξόδου 5 ns αργότερα.

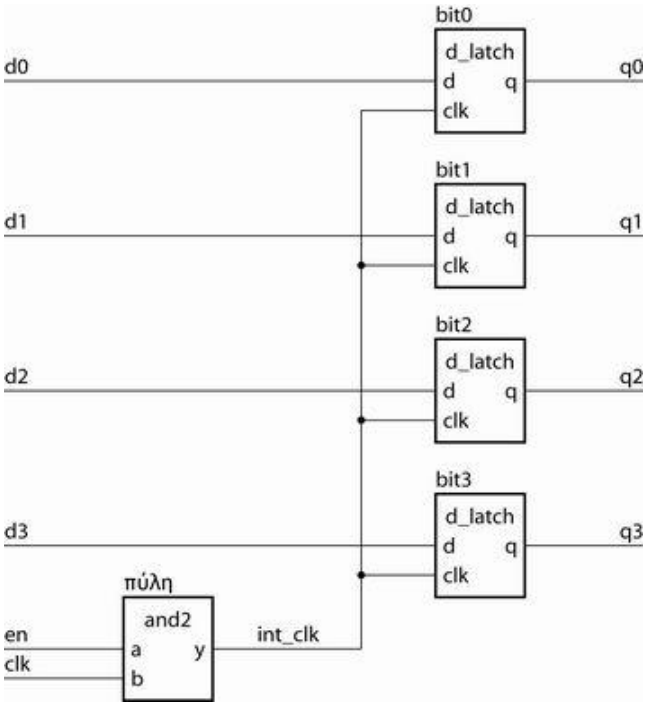
Όταν όλες αυτές οι προτάσεις στη διεργασία έχουν εκτελεστεί, η διεργασία φτάνει στην *πρόταση wait (wait statement)* και *διακόπτεται (suspend)*· αυτό σημαίνει, ότι γίνεται ανενεργή. Μένει ανενεργή μέχρις ότου ένα από τα σήματα στα οποία είναι *ευαίσθητη (sensitive)* να αλλάξει τιμή. Σε αυτήν την περίπτωση, η διεργασία είναι ευαίσθητη στα σήματα d0, d1, d2, d3, en και clk, εφόσον αυτά απαριθμούνται στην πρόταση wait. Όταν ένα από αυτά αλλάξει τιμή, η διεργασία επαναρχίζει. Οι προτάσεις εκτελούνται πάλι, ξεκινώντας από τη λέξη κλειδί **begin**, και ο κύκλος επαναλαμβάνεται. Παρατηρείστε ότι ενώ η διεργασία είναι σταματημένη, οι τιμές στις μεταβλητές της διεργασίας δεν χάνονται. Αυτός είναι ο τρόπος με τον οποίο η διεργασία μπορεί να αναπαραστήσει την κατάσταση ενός συστήματος.

1.4.2 Στοιχεία Δομής

Ένας εναλλακτικός τρόπος περιγραφής της υλοποίησης μιας οντότητας είναι να προσδιορίσουμε πως αυτή συντίθεται από ξεχωριστά υποσυστήματα. Μπορούμε να δώσουμε μια δομική περιγραφή της υλοποίησης της οντότητας. Ένα σώμα αρχιτεκτονικής που αποτελείται μόνο από διασυνδεδεμένα υποσυστήματα καλείται σώμα αρχιτεκτονικής *δομής (structural)*. Η Εικόνα 1-9 παρουσιάζει με ποιον τρόπο η οντότητα reg4 μπορεί να κατασκευαστεί από μανδαλωτές (latches) και πύλες. Αν πρόκειται να περιγράψουμε αυτήν τη δομή σε VHDL, θα χρειαστούμε δηλώσεις οντοτήτων και σώματα αρχιτεκτονικών για τα υποσυστήματα, τα οποία παρουσιάζονται στην Εικόνα 1-10.

Η Εικόνα 1-11 είναι μια VHDL δήλωση ενός σώματος αρχιτεκτονικής που περιγράφει τη δομή που παρουσιάζεται στην Εικόνα 1-9. Η *δήλωση σήματος (signal declaration)*, πριν από τη λέξη κλειδί **begin**, ορίζει τα εσωτερικά σήματα της αρχιτεκτονικής. Σε αυτό το παράδειγμα, το σήμα int_clk δηλώνεται να μεταφέρει μια τιμή bit ('0' ή '1'). Γενικά, τα VHDL σήματα μπορούν να δηλωθούν να μεταφέρουν αυθαίρετα πολύπλοκες τιμές. Εντός του σώματος της αρχιτεκτονικής οι θύρες της οντότητας αντιμετωπίζονται επίσης σαν σήματα.

ΕΙΚΟΝΑ 1-9



Μια σύνθεση δομής της οντότητας reg4.

ΕΙΚΟΝΑ 1-10

```
entity d_latch is
    port (d,clk:in bit; q : out bit );
end d_latch;

architecture basic of d_latch is
begin
    latch_behavior : process is
    begin
        if clk = '1' then
            q <=d after 2 ns;
        end if;
        wait on clk, d;
    end process latch_behavior;
end architecture basic;
-----------------------------------------------------------------------------------------------

entity and2 is
    port (a,b :in bit; y : out bit );
end and2;

architecture basic of and2 is
begin
    and2_behavior : process is
    begin
        y <=a and b after 2 ns;
        wait on a, b;
    end process and2_behavior;
end architecture basic;
```

Δηλώσεις οντότητας και σώματα αρχιτεκτονικής για το D φλιπ φλοπ (D-flip flop) και την πύλη and 2-εισόδων.

Στο δεύτερο μέρος του σώματος της αρχιτεκτονικής, δημιουργείται ένας αριθμός *στιγμιότυπων συστατικών (component instances)*, που αναπαριστούν τα υποσυστήματα από τα οποία αποτελείται η οντότητα reg4. Κάθε στιγμιότυπο συστατικού είναι ένα αντίγραφο της οντότητας που αναπαριστά το υποσύστημα, με χρήση του αντίστοιχου σώματος αρχιτεκτονικής basic. (Το όνομα work αναφέρεται στην τρέχουσα βιβλιοθήκη εργασίας, στην οποία θεωρούμε ότι κρατούνται όλες οι περιγραφές της οντότητας και του σώματος της αρχιτεκτονικής.)

ΕΙΚΟΝΑ 1-11

```
architecture struct of reg4 is
    signal int_clk : bit;
begin
     bit0 : entity work.d_latch(basic)
            port map (d0, int_clk, q0);
     bit1 : entity work.d_latch(basic)
            port map (d1, int_clk, q1);
     bit2 : entity work.d_latch(basic)
            port map (d2, int_clk, q2);
     bit3 : entity work.d_latch(basic)
            port map (d3, int_clk, q3);
     gate : entity work.and2(basic)
            port map (en, clk, int_clk);
end architecture struct;
```

Ένα VHDL σώμα αρχιτεκτονικής δομής της οντότητας reg4.

Η πρόταση *port map* προσδιορίζει τη σύνδεση των θυρών κάθε στιγμιότυπου συστατικού στα σήματα εντός του σώματος αρχιτεκτονικής που τα περικλείει. Για παράδειγμα, το bit0, ένα στιγμιότυπο της οντότητας d_latch, έχει τη θύρα του d συνδεδεμένη στο σήμα d0, τη θύρα του clk συνδεδεμένη στο σήμα int_clk και τη θύρα του q συνδεδεμένη στο σήμα q0.

1.4.3 Μικτά Μοντέλα Δομής και Συμπεριφοράς

Τα μοντέλα δεν είναι απαραίτητο να είναι αμιγώς μοντέλα δομής ή αμιγώς μοντέλα συμπεριφοράς. Συχνά είναι χρήσιμο να προσδιορίσουμε ένα μοντέλο με κάποια τμήματά του να αποτελούνται από διασυνδεδεμένα στιγμιότυπα συστατικών, και άλλα τμήματα να περιγράφονται με χρήση διεργασιών. Χρησιμοποιούμε σήματα σαν μέσα ένωσης στιγμιότυπων συστατικών και διεργασιών. Ένα σήμα μπορεί να σχετίζεται με μια θύρα ενός στιγμιότυπου συστατικού και μπορεί επίσης να οδηγηθεί ή να διαβαστεί σε μια διεργασία.

ΕΙΚΟΝΑ 1-12

```
entity multiplier is
    port (  clk, reset : in bit;
            multiplicand, multiplier : in integer;
            product : out integer );
end entity multiplier;
-------------------------------------------------------------------------------------------------

architecture mixed of multiplier is
    signal partial_product, full_product : integer;
    signal arith_control, result_en, mult_bit, mult_load : bit;
begin -- μικτό
    arith_unit :entity work.shift_adder(behavior)
         port map ( addend => multiplicand, augend => full_product,
                    sum => partial_product,
                    add_control => arith_control);
    result : entity work.reg(behavior)
        port map (  d => partial_product, q => full_product,
                    en => result_en, reset => reset);
    multiplier_sr : entity work.shift_reg(behavior)
        port map (  d => multiplier, q => mult_bit,
                    load => mult_load, clk => clk);
    product <= full_product;
    control_section : process is
        -- δηλώσεις μεταβλητών για το control_section
        -- …
    begin -- control section
        -- ακολουθιακές προτάσεις ανάθεσης τιμών στα σήματα ελέγχου
        -- …
        wait on clk, reset;
    end process control_section;
end architecture mixed;
```

Ένα περίγραμμα ενός μικτού μοντέλου δομής και συμπεριφοράς ενός πολλαπλασιαστή.

Μπορούμε να γράψουμε ένα τέτοιο υβριδικό μοντέλο εισάγοντας τόσο στιγμιότυπα συστατικών όσο και προτάσεις διεργασιών στο σώμα μιας αρχιτεκτονικής. Αυτές οι προτάσεις συνολικά καλούνται *ταυτόχρονες προτάσεις (concurrent statements)*, εφόσον οι αντίστοιχες διεργασίες εκτελούνται όλες ταυτόχρονα όταν προσομοιώνεται το μοντέλο. Ένα περίγραμμα ενός τέτοιου μοντέλου παρουσιάζεται στην Εικόνα 1-12. Αυτό το μοντέλο περιγράφει έναν πολλαπλασιαστή που αποτελείται από μια διαδρομή δεδομένων και έναν τομέα ελέγχου. Η διαδρομή δεδομένων περιγράφεται δομικά, με χρήση ενός αριθμού στιγμιότυπων συστατικών. Ο τομέας ελέγχου περιγράφεται με συμπεριφορά, με χρήση μιας διεργασίας που αναθέτει τιμές στα σήματα ελέγχου της διαδρομής δεδομένων.

1.4.4 Πάγκοι Ελέγχου

Στην εισαγωγική μας ανάλυση, αναφέραμε τη δοκιμή μέσω προσομοίωσης σαν ένα κίνητρο για την μοντελοποίηση. Συχνά δοκιμάζουμε ένα VHDL μοντέλο με χρήση ενός μοντέλου που το εσωκλείει και που καλείται *πάγκος ελέγχου (test bench)*. Το όνομα προέρχεται από την αναλογία με έναν πραγματικό πάγκο ελέγχου υλικού, στον οποίο μια συσκευή υπό έλεγχο διεγείρεται με γεννήτριες σήματος (signal generators) και παρατηρείται με ανιχνευτές σημάτων (signal probes). Ένας πάγκος ελέγχου στη VHDL αποτελείται από ένα σώμα αρχιτεκτονικής που περιέχει ένα στιγμιότυπο του συστατικού υπό έλεγχο και διεργασίες που παράγουν ακολουθίες τιμών σε σήματα συνδεδεμένα στις θύρες εισόδου του συστατικού. Το σώμα της αρχιτεκτονικής μπορεί επίσης να περιέχει διεργασίες που ελέγχουν ότι το στιγμιότυπο του συστατικού παράγει τις αναμενόμενες τιμές στα σήματα εξόδου του. Εναλλακτικά, μπορούμε να χρησιμοποιήσουμε τις δυνατότητες παρακολούθησης που μας παρέχει ένας προσομοιωτής για την παρατήρηση των εξόδων.

Ένα μοντέλο πάγκου έλεγχου για τη δοκιμή της συμπεριφοράς του καταχωρητή reg4 παρουσιάζεται στην Εικόνα 1-13. Η δήλωση της οντότητας δεν έχει λίστα θυρών, εφόσον ο πάγκος ελέγχου είναι πλήρως αυτόνομος. Το σώμα αρχιτεκτονικής περιέχει σήματα που συνδέονται στις θύρες εισόδου και εξόδου του στιγμιότυπου του συστατικού dut, της συσκευής υπό έλεγχο (device under test). Η διεργασία με την ετικέτα stimulus παρέχει μια ακολουθία τιμών ελέγχου στα σήματα εισόδου εκτελώντας προτάσεις ανάθεσης σήματος μαζί με προτάσεις wait. Κάθε πρόταση wait προσδιορίζει μια παύση 20 ns κατά τη διάρκεια της οποίας ο καταχωρητής καθορίζει τις τιμές εξόδου. Μπορούμε να χρησιμοποιήσουμε έναν προσομοιωτή για να παρατηρήσουμε τις τιμές στα σήματα q0 έως q3 για την επαλήθευση της σωστής λειτουργίας του καταχωρητή. Όταν όλες οι τιμές διέγερσης έχουν εφαρμοστεί, η διαδικασία διέγερσης αναστέλλεται επ'αόριστον ολοκληρώνοντας με αυτόν τον τρόπο την προσομοίωση.

ΕΙΚΟΝΑ 1-13

```
entity test_bench is
end entity test_bench;
architecture test_reg4 of test_bench is
    signal d0, d1, d2, d3, en, clk, q0, q1, q2, q3 : bit;
begin
    dut : entity work.reg4(behav)
        port map ( d0, d1, d2, d3, en, clk, q0, q1, q2, q3 );
    stimulus : process is
    begin
        d0 <= '1'; d1 <= '1'; d2 <= '1'; d3 <= '1';
        en <= '0'; clk <= '0';
        wait for 20 ns;
        en <= '1'; wait for 20 ns;
        clk <= '1'; wait for 20 ns;
        d0 <= '0'; d1 <= '0'; d2 <= '0'; d3 <= '0'; wait for 20 ns;
        en <= '0'; wait for 20 ns;
        …
        wait;
    end process stimulus;
end architecture test_reg4;
```

Ένας VHDL πάγκος ελέγχου για το μοντέλο καταχωρητή reg4.

1.4.5 Ανάλυση, Ανάπτυξη και Εκτέλεση

Ένας από τους κύριους λόγους για τη συγγραφή του μοντέλου ενός συστήματος είναι να έχουμε τη δυνατότητα να το προσομοιώσουμε. Αυτό περιλαμβάνει τρία στάδια: την *ανάλυση (analysis)*, την *ανάπτυξη (elaboration)* και την *εκτέλεση (execution)*. Η ανάλυση και η ανάπτυξη απαιτούνται επίσης για την προετοιμασία του μοντέλου για άλλες χρήσεις, όπως η λογική σύνθεση (logic synthesis).

Στο πρώτο στάδιο, το στάδιο της ανάλυσης, η VHDL περιγραφή του συστήματος ελέγχεται για διάφορους τύπους σφαλμάτων. Όπως οι περισσότερες γλώσσες προγραμματισμού, η VHDL έχει αυστηρά ορισμένο *συντακτικό (syntax)* και *σημειολογία (semantics)*. Το συντακτικό είναι το σύνολο των γραμματικών κανόνων που ελέγχουν τον τρόπο

γραφής ενός μοντέλου. Οι κανόνες της σημειολογίας ελέγχουν τη σημασία ενός προγράμματος. Για παράδειγμα, είναι λογικό να πραγματοποιήσουμε μια λειτουργία πρόσθεσης σε δύο αριθμούς αλλά όχι σε δυο διαδικασίες.

Κατά τη διάρκεια της φάσης της ανάλυσης, η VHDL περιγραφή εξετάζεται και εντοπίζονται συντακτικά και στατικά σημειολογικά σφάλματα. Το συνολικό μοντέλο ενός συστήματος δεν χρειάζεται να αναλυθεί αμέσως. Αντίθετα, είναι πιθανό να αναλυθούν οι *μονάδες σχεδίασης (design units)*, όπως οι δηλώσεις οντότητας και σώματος αρχιτεκτονικής, ξεχωριστά. Αν ο αναλυτής δεν βρει σφάλματα σε μια μονάδα σχεδίασης, δημιουργεί μια ενδιάμεση αναπαράσταση της μονάδας και την αποθηκεύει σε μια βιβλιοθήκη. Ο ακριβής μηχανισμός διαφέρει μεταξύ διαφορετικών εργαλείων VHDL.

Το δεύτερο στάδιο στην προσομοίωση ενός μοντέλου, η ανάπτυξη, είναι η πράξη της εργασίας μέσω της ιεραρχίας της σχεδίασης και της δημιουργίας όλων των αντικειμένων που ορίζονται στις δηλώσεις. Το τελικό αποτέλεσμα της ανάπτυξης της σχεδίασης είναι μια συλλογή σημάτων και διεργασιών, με κάθε διεργασία πιθανώς να περιέχει μεταβλητές. Ένα μοντέλο πρέπει να μπορεί να αναχθεί σε μια συλλογή από σήματα και διεργασίες ώστε να προσομοιωθεί.

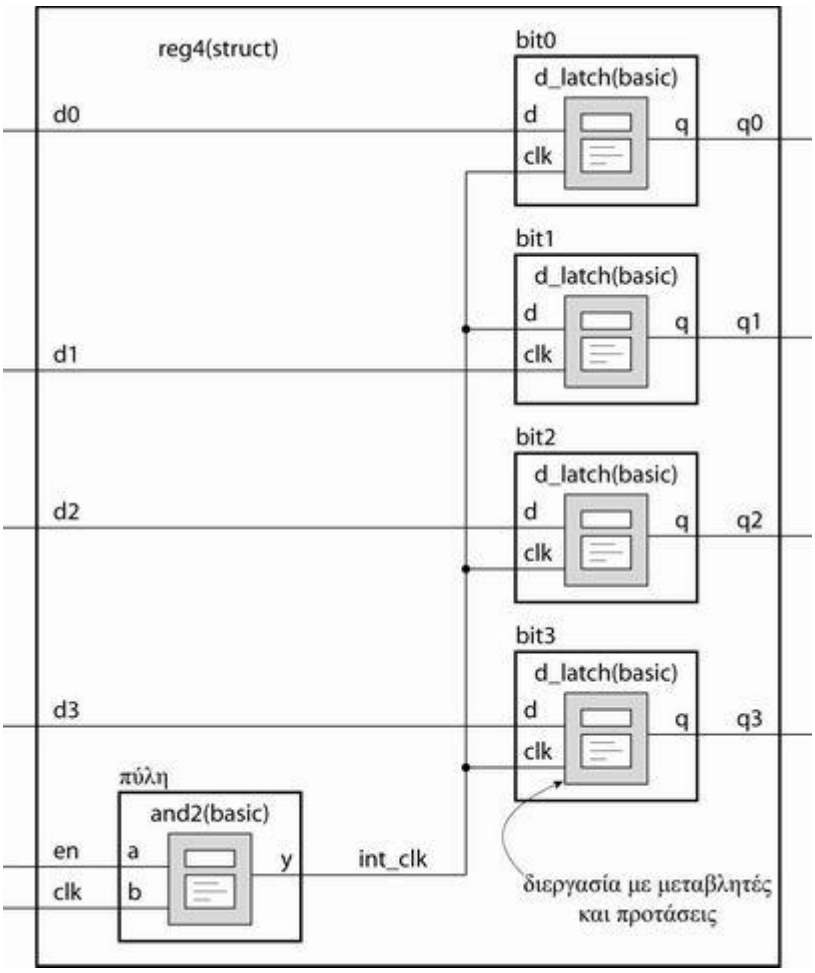
Μπορούμε να δούμε με ποιο τρόπο η ανάπτυξη πετυχαίνει αυτήν την αναγωγή ξεκινώντας από το υψηλότερο επίπεδο ενός μοντέλου, την οντότητα, και επιλέγοντας μια αρχιτεκτονική της υπό προσομοίωση οντότητας. Η αρχιτεκτονική περιλαμβάνει σήματα, διεργασίες και στιγμιότυπα συστατικών. Κάθε στιγμιότυπο συστατικού είναι ένα αντίγραφο μιας οντότητας και μιας αρχιτεκτονικής που περιλαμβάνει επίσης σήματα, διεργασίες και στιγμιότυπα συστατικών. Στιγμιότυπα αυτών των σημάτων και των διεργασιών δημιουργούνται, σε αντιστοιχία με το στιγμιότυπο του συστατικού, και στη συνέχεια η λειτουργία ανάπτυξης επαναλαμβάνεται για τα στιγμιότυπα των υποσυστατικών. Τελικά, καταλήγουμε σε ένα στιγμιότυπο συστατικού το οποίο είναι αντίγραφο μιας οντότητας με καθαρά αρχιτεκτονική συμπεριφοράς, που περιέχει μόνο διεργασίες. Αυτό αντιστοιχεί σε ένα πρωτογενές συστατικό για το επίπεδο της σχεδίασης που προσομοιώνεται. Η Εικόνα 1-14 παρουσιάζει με ποιο τρόπο προχωράει η ανάπτυξη για το σώμα αρχιτεκτονικής δομής της οντότητας reg4. Όπως δημιουργείται κάθε στιγμιότυπο μιας διεργασίας, οι μεταβλητές του δημιουργούνται και τους δίνονται αρχικές τιμές. Μπορούμε να θεωρήσουμε κάθε στιγμιότυπο διεργασίας ότι αντιστοιχεί σε ένα στιγμιότυπο ενός συστατικού.

Το τρίτο στάδιο μιας προσομοίωσης είναι η εκτέλεση του μοντέλου. Το πέρασμα του χρόνου προσομοιώνεται σε διακριτά βήματα, ανάλογα με το πότε συμβαίνουν γεγονότα (events). Ως εκ τούτου χρησιμοποιείται ο όρος *προσομοίωση διακριτών γεγονότων (discrete event simulation)*. Σε κάποια χρονική στιγμή της προσομοίωσης, μια διεργασία μπορεί να διεγείρεται από την αλλαγή της τιμής σε ένα σήμα στο οποίο είναι ευαίσθητη. Η διεργασία ξαναρχίζει και μπορεί να χρονοπρογραμματίσει (schedule) την απόδοση νέων τιμών σε σήματα σε κάποιο μεταγενέστερο χρόνο της προσομοίωσης. Αυτό καλείται *χρονοπρογραμματισμός μιας συναλλαγής (scheduling a transaction)* σε αυτό το σήμα. Αν η νέα τιμή είναι διαφορετική από την προηγούμενη τιμή στο σήμα, ένα γεγονός συμβαίνει, και οι άλλες διεργασίες που είναι ευαίσθητες στο σήμα μπορούν να ξαναξεκινήσουν.

Η προσομοίωση ξεκινάει με μια *φάση αρχικοποίησης (initialization phase)*, ακολουθούμενη από την επαναληπτική εκτέλεση ενός *κύκλου προσομοίωσης (simulation cycle)*. Κατά τη διάρκεια της φάσης αρχικοποίησης, σε κάθε σήμα ανατίθεται μια αρχική τιμή, ανάλογα με τον τύπο του. Ο χρόνος προσομοίωσης τίθεται στο μηδέν, και στη συνέχεια κάθε στιγμιότυπο διεργασίας ενεργοποιείται και οι ακολουθιακές προτάσεις του εκτελούνται. Συνήθως, μια διεργασία θα περιλαμβάνει μια πρόταση ανάθεσης σήματος για να χρονοπρογραμματίσει μια συναλλαγή σε ένα σήμα σε κάποιο μεταγενέστερο χρόνο της προσομοίωσης. Η εκτέλεση μιας διεργασίας συνεχίζει μέχρι να φτάσει σε μια πρόταση wait, που προκαλεί τη διακοπή της διεργασίας.

Κατά τη διάρκεια του κύκλου προσομοίωσης, ο χρόνος προσομοίωσης πρώτα προωθείται στην επόμενη χρονική στιγμή στην οποία έχει χρονοπρογραμματιστεί μια συναλλαγή σε ένα σήμα. Δεύτερον, πραγματοποιούνται όλες οι συναλλαγές που είναι χρονοπρογραμματισμένες για εκείνη τη χρονική στιγμή. Αυτό μπορεί να προκαλέσει να συμβούν κάποια γεγονότα σε κάποια σήματα. Τρίτον, όλες οι διεργασίες που είναι ευαίσθητες σε αυτά τα γεγονότα ξαναρχίζουν και τους επιτρέπεται να συνεχίσουν μέχρι να φτάσουν σε μια πρόταση wait και να σταματήσουν. Πάλι, οι διεργασίες συχνά εκτελούν αναθέσεις σημάτων για να χρονοπρογραμματίσουν παραπέρα συναλλαγές σε σήματα. Όταν όλες οι διεργασίες έχουν σταματήσει πάλι, ο κύκλος προσομοίωσης επαναλαμβάνεται. Όταν η προσομοίωση φτάσει στο στάδιο όπου δεν υπάρχουν επιπλέον χρονοπρογραμματισμένες συναλλαγές, σταματάει, εφόσον η προσομοίωση έχει τότε ολοκληρωθεί.

ΕΙΚΟΝΑ 1-14



Η ανάπτυξη της οντότητας reg4 *με χρήση του σώματος αρχιτεκτονικής δομής. Κάθε στιγμιότυπο των οντοτήτων* d_latch *και* and2 *αντικαθίσταται με τα περιεχόμενα της αντίστοιχης αρχιτεκτονικής* basic. *Το καθένα από αυτά αποτελείται από μια διεργασία με τις μεταβλητές και τις προτάσεις της.*

1.5 Μαθαίνοντας μια Νέα Γλώσσα: Λεξικολογικά Στοιχεία και Συντακτικό

Όταν μαθαίνουμε μια νέα φυσική γλώσσα, όπως Ελληνικά, Κινέζικα ή Αγγλικά, αρχίζουμε με την εκμάθηση του αλφάβητου των συμβόλων που χρησιμοποιούνται στη γλώσσα, και στη συνέχεια μαθαίνουμε πώς να σχηματίζουμε λέξεις με αυτά τα σύμβολα. Έπειτα, μαθαίνουμε πως να τοποθετούμε τις λέξεις μαζί ώστε να σχηματίζουμε προτάσεις και την έννοια αυτών των προτάσεων. Φθάνουμε σε ευχέρεια λόγου σε μια γλώσσα όταν μπορούμε εύκολα να εκφράσουμε ότι θέλουμε να πούμε χρησιμοποιώντας σωστά σχηματισμένες προτάσεις.

Οι ίδιες ιδέες ισχύουν όταν πρέπει να μάθουμε μια νέα γλώσσα ειδικής-χρήσης, όπως η VHDL, για την περιγραφή ψηφιακών συστημάτων. Μπορούμε να δανειστούμε μερικούς όρους από τη γλωσσική θεωρία για να περιγράψουμε τι απαιτείται να μάθουμε. Πρώτον, χρειάζεται να μάθουμε το αλφάβητο με το οποίο γράφεται η γλώσσα. Το αλφάβητο της VHDL αποτελείται από όλους τους χαρακτήρες του συνόλου χαρακτήρων 8-bit ISO 8859 Latin-1. Αυτό περιλαμβάνει τα κεφαλαία και πεζά γράμματα (συμπεριλαμβανομένων των γραμμάτων με διακριτικά σημάδια, όπως ‘à’, ‘ä’ και ούτω καθεξής), τα ψηφία 0 έως 9, τους χαρακτήρες στίξης και άλλους ειδικούς χαρακτήρες. Δεύτερον, χρειάζεται να μάθουμε τα *λεξικολογικά στοιχεία (lexical elements)* της γλώσσας. Στη VHDL, αυτά είναι τα αναγνωριστικά (identifiers), οι δεσμευμένες λέξεις (reserved words), τα ειδικά σύμβολα (special symbols) και τα κυριολεκτικά (literals). Τρίτον, χρειάζεται να μάθουμε τη *συντακτικό (syntax)* της γλώσσας. Αυτό είναι η γραμματική που καθορίζει ποιοι συνδυασμοί λεξικολογικών στοιχείων συγκροτούν νόμιμες VHDL περιγραφές. Τέταρτον, χρειάζεται να μάθουμε τη *σημασιολογία (semantics)*, ή την έννοια, των VHDL περιγραφών. Η σημασιολογία είναι αυτή που επιτρέπει σε μια συλλογή συμβόλων να περιγράψει μια ψηφιακή σχεδίαση. Πέμπτον, χρειάζεται να μάθουμε πώς να αναπτύξουμε τις δικές μας VHDL περιγραφές για να περιγράψουμε μια σχεδίαση για την οποία εργαζόμαστε. Αυτό

είναι το δημιουργικό μέρος της μοντελοποίησης, και η ευχέρεια σε αυτό το μέρος θα ενισχύσει σημαντικά τις δεξιότητές μας στη σχεδίαση.

Στο υπόλοιπο αυτού του κεφαλαίου, περιγράφουμε τα λεξικολογικά στοιχεία που χρησιμοποιούνται στη VHDL και εισάγουμε τη σημειογραφία (notation) που χρησιμοποιούμε για να περιγράψουμε τους συντακτικούς κανόνες (syntax rules). Κατόπιν στα επόμενα κεφάλαια, εισάγουμε τις διαφορετικές δυνατότητες που μας παρέχει η γλώσσα. Για κάθε μία από αυτές, παρουσιάζουμε τους συντακτικούς κανόνες, περιγράφουμε την αντίστοιχη σημασιολογία και δίνουμε παραδείγματα για το πως χρησιμοποιείται για να μοντελοποιήσει συγκεκριμένα μέρη ενός ψηφιακού συστήματος.

VHDL-87

Η VHDL-87 χρησιμοποιεί το σύνολο χαρακτήρων ASCII, και όχι το πλήρες σύνολο χαρακτήρων ISO. Το σύνολο χαρακτήρων ASCII είναι ένα υποσύνολο του συνόλου χαρακτήρων ISO, που αποτελείται ακριβώς από τους πρώτους 128 χαρακτήρες. Αυτό περιλαμβάνει όλα τα ατόνιστα γράμματα, αλλά αποκλείει γράμματα με διακριτικά σημάδια.

1.5.1 Λεξικολογικά Στοιχεία

Στην επόμενη ενότητα, συζητάμε τα λεξικολογικά στοιχεία της VHDL: *σχόλια (comments), αναγνωριστικά (identifiers), δεσμευμένες λέξεις (reserved words), ειδικά σύμβολα (special symbols), αριθμούς (numbers), χαρακτήρες (characters), αλφαριθμητικά (strings),* και *ψηφιοσειρές (bit strings)*.

Σχόλια

Όταν γράφουμε ένα μοντέλο υλικού στη VHDL, είναι σημαντικό να προσθέτουμε σχόλια στον κώδικα. Ο λόγος για αυτό είναι να βοηθήσουμε τους αναγνώστες να καταλάβουν τη δομή και τη λογική που κρύβεται πίσω από το μοντέλο. Είναι σημαντικό να συνειδητοποιήσουμε ότι αν και γράφουμε ένα μοντέλο μόνο μια φορά, είναι πιθανό στη συνέχεια να διαβαστεί και να τροποποιηθεί πολλές φορές, και από τον ίδιο το συγγραφέα και από άλλους μηχανικούς. Επομένως, οποιαδήποτε βοήθεια μπορούμε να δώσουμε ώστε να κατανοήσει κάποιος το μοντέλο αξίζει τον κόπο.

Ένα μοντέλο VHDL αποτελείται από έναν αριθμό γραμμών κειμένου. Ένα σχόλιο μπορεί να προστεθεί σε μια γραμμή γράφοντας δύο παύλες μαζί, ακολουθούμενες από το κείμενο του σχολίου. Για παράδειγμα:

```
… μια γραμμή περιγραφής VHDL … -- ένα περιγραφικό σχόλιο
```

Το σχόλιο επεκτείνεται από τις δύο παύλες έως το τέλος της γραμμής και μπορεί να περιλαμβάνει οποιοδήποτε κείμενο επιθυμούμε, εφόσον δεν είναι τυπικά μέρος του μοντέλου VHDL. Ο κώδικας ενός μοντέλου μπορεί να περιλαμβάνει κενές γραμμές και γραμμές που περιέχουν μόνο σχόλια, αρχίζοντας από δύο παύλες. Μπορούμε να γράψουμε μακρά σχόλια σε διαδοχικές γραμμές, με την κάθε μια να αρχίζει με δύο παύλες, για παράδειγμα:

```
-- Ο παρακάτω κώδικας μοντελοποιεί
--  το τμήμα ελέγχου του συστήματος
… κάποιος VHDL κώδικας …
```

Αναγνωριστικά

Τα αναγνωριστικά χρησιμοποιούνται για να ονομάσουν στοιχεία σε ένα μοντέλο VHDL. Είναι ορθή πρακτική να χρησιμοποιούμε ονόματα που υποδηλώνουν το σκοπό του στοιχείου, έτσι η VHDL επιτρέπει τα ονόματα να είναι αυθαίρετα μακριά. Εντούτοις, υπάρχουν μερικοί κανόνες για το πως πρέπει να σχηματίζονται τα αναγνωριστικά. Ένα βασικό αναγνωριστικό

- επιτρέπεται να περιέχει μόνο γράμματα της αλφαβήτου (‘A’ έως ‘Z’ και ‘a’ έως ‘z’), δεκαδικά ψηφία (‘0’ έως ‘9’) και το χαρακτήρα υπογράμμισης (`_")
- πρέπει να αρχίζει με ένα γράμμα της αλφαβήτου
- δεν επιτρέπεται να τελειώνει με ένα χαρακτήρα υπογράμμισης και
- δεν επιτρέπεται να περιλαμβάνει δύο διαδοχικούς χαρακτήρες υπογράμμισης.

Μερικά παραδείγματα έγκυρων βασικών αναγνωριστικών είναι

```
A   X0   counter   Next_Value   generate_read_cycle
```

Μερικά παραδείγματα μη έγκυρων βασικών αναγνωριστικών είναι

```
last@value    -- περιέχει ένα μη νόμιμο χαρακτήρα για ένα αναγνωριστικό
5bit_counter  -- αρχίζει με ένα μη-αλφαβητικό χαρακτήρα
_A0           -- αρχίζει με ένα χαρακτήρα υπογράμμισης
A0_           -- τελειώνει με ένα χαρακτήρα υπογράμμισης
```

```
clock__pulse  -- περιέχει δύο διαδοχικούς χαρακτήρες υπογράμμισης
```

Σημειώστε ότι η διάκριση πεζών/κεφαλαίων δεν θεωρείται σημαντική, έτσι τα αναγνωριστικά cat και Cat είναι ίδια. Οι χαρακτήρες υπογράμμισης στα αναγνωριστικά είναι σημαντικοί, έτσι τα αναγνωριστικά This_Name και ThisName είναι διαφορετικά.

Εκτός από τα βασικά αναγνωριστικά, η VHDL επιτρέπει *εκτεταμένα αναγνωριστικά (extended identifiers)*, τα οποία μπορούν να περιέχουν οποιαδήποτε ακολουθία χαρακτήρων. Τα εκτεταμένα αναγνωριστικά έχουν συμπεριληφθεί ώστε να επιτρέπουν την επικοινωνία μεταξύ εργαλείων σχεδίασης με τη βοήθεια υπολογιστή (computer-aided design tools) για την επεξεργασία περιγραφών VHDL και άλλων εργαλείων που χρησιμοποιούν διαφορετικούς κανόνες για τα αναγνωριστικά. Ένα εκτεταμένο αναγνωριστικό γράφεται εσωκλείοντας τους χαρακτήρες του αναγνωριστικού μεταξύ των χαρακτήρων '\'. Για παράδειγμα:

```
\data bus\   \global.clock\   \923\   \d#1\   \start__\
```

Εάν χρειαστεί να συμπεριλάβουμε ένα χαρακτήρα '\' σε ένα εκτεταμένο αναγνωριστικό, το κάνουμε με το διπλασιασμό του χαρακτήρα, για παράδειγμα:

```
\A:\\name\          -- περιέχει ένα '\' μεταξύ του ':' και του 'n'
```

Σημειώστε ότι η διάκριση πεζών/κεφαλαίων είναι σημαντική στα εκτεταμένα αναγνωριστικά και ότι όλα τα εκτεταμένα αναγνωριστικά είναι ξεχωριστά από όλα τα βασικά αναγνωριστικά. Έτσι τα παρακάτω είναι όλα διαφορετικά αναγνωριστικά:

```
name   \name\   \Name\   \NAME\
```

VHDL-87

Η VHDL-87 επιτρέπει μόνο βασικά αναγνωριστικά, και όχι εκτεταμένα αναγνωριστικά. Οι κανόνες για το σχηματισμό βασικών αναγνωριστικών είναι οι ίδιοι με εκείνους της VHDL-93 και της VHDL-2001.

Δεσμευμένες Λέξεις

Μερικά αναγνωριστικά, που αποκαλούνται δεσμευμένες λέξεις (reserved words) ή λέξεις-κλειδιά (keywords), είναι δεσμευμένες για ειδική χρήση στη VHDL. Χρησιμοποιούνται για να υποδηλώσουν συγκεκριμένες δομές που διαμορφώνουν ένα μοντέλο, έτσι δεν μπορούμε να τις χρησιμοποιήσουμε ως αναγνωριστικά για στοιχεία που ορίζουμε. Ο πλήρης κατάλογος δεσμευμένων λέξεων παρουσιάζεται στην Εικόνα 1-15. Συχνά, όταν στοιχειοθετείται ένα πρόγραμμα VHDL, οι δεσμευμένες λέξεις τυπώνονται σε έντονη μορφή (boldface). Αυτή η σύμβαση ακολουθείται και σε αυτό το βιβλίο.

ΕΙΚΟΝΑ 1-15

abs	**disconnect**	**label**	**package**	**sla**
access	**downto**	**library**	**port**	**sll**
after	**else**	**linkage**	**postponed**	**sra**
alias	**elsif**	**literal**	**procedure**	**srl**
all	**end**	**loop**	**process**	**subtype**
and	**entity**	**map**	**protected**	**then**
architecture	**exit**	**mod**	**pure**	**to**
array	**file**	**nand**	**range**	**transport**
assert	**for**	**new**	**record**	**type**
attribute	**function**	**next**	**register**	**unaffected**
begin	**generate**	**nor**	**reject**	**units**
block	**generic**	**not**	**rem**	**until**
body	**group**	**null**	**report**	**use**
buffer	**guarded**	**of**	**return**	**variable**
bus	**if**	**on**	**rol**	**wait**
case	**impure**	**open**	**ror**	**when**
component	**in**	**or**	**select**	**while**
configuration	**inertial**	**others**	**severity**	**with**
constant	**inout**	**out**	**shared**	**xnor**
	is		**signal**	**xor**

Δεσμευμένες λέξεις της VHDL.

VHDL-87

Τα παρακάτω αναγνωριστικά δεν χρησιμοποιούνται ως δεσμευμένες λέξεις στη VHDL-87. Οπότε μπορούν να χρησιμοποιηθούν ως αναγνωριστικά για άλλους σκοπούς, αν και αυτό δεν ενδείκνυται, επειδή αυτό μπορεί να προκαλέσει δυσκολίες στη μεταφορά των μοντέλων στη VHDL-93 και στη VHDL-2001.

group	**postponed**	**ror**	**sra**
impure	**protected**	**shared**	**srl**
inertial	**pure**	**sla**	**unaffected**
literal	**reject**	**sll**	**xnor**
	rol		

VHDL-93

Το αναγνωριστικό **protected** δεν χρησιμοποιείται ως δεσμευμένη λέξη στη VHDL-93.

Ειδικά Σύμβολα

Η VHDL χρησιμοποιεί έναν αριθμό από ειδικά σύμβολα για να υποδηλώσει τελεστές, για να οριοθετήσει μέρη των δομών της γλώσσας και ως σημεία στίξης. Μερικά από αυτά τα ειδικά σύμβολα αποτελούνται από ένα μόνο χαρακτήρα. Αυτά είναι

```
" # & ' ( ) * + – , . / : ; < = > [ ] |
```

Άλλα ειδικά σύμβολα αποτελούνται από ζευγάρια χαρακτήρων. Οι δύο χαρακτήρες πρέπει να πληκτρολογηθούν ο ένας δίπλα στον άλλο, χωρίς να παρεμβάλλεται κενό διάστημα. Αυτά τα σύμβολα είναι

```
=>  **  :=  /=  >=  <=  <>
```

Αριθμοί

Υπάρχουν δύο μορφές αριθμών που μπορούν να γραφτούν σε κώδικα VHDL: *κυριολεκτικά ακέραιων αριθμών (integer literals)* και *κυριολεκτικά πραγματικών αριθμών (real literals)*. Ένα κυριολεκτικό ακέραιου αριθμού απλά αναπαριστά έναν ολόκληρο αριθμό και αποτελείται από ψηφία χωρίς υποδιαστολή. Τα κυριολεκτικά πραγματικών αριθμών, από την άλλη μεριά, μπορούν να αναπαραστήσουν κλασματικούς αριθμούς. Περιλαμβάνουν πάντα μια υποδιαστολή, από την οποία προηγείται ένα τουλάχιστον ψηφίο και ακολουθεί ένα τουλάχιστον ψηφίο. Τα κυριολεκτικά πραγματικών αριθμών αναπαριστούν μια προσέγγιση των πραγματικών αριθμών.

Μερικά παραδείγματα κυριολεκτικών δεκαδικών ακέραιων αριθμών είναι

```
23  0  146
```

Σημειώστε ότι το –10, για παράδειγμα, δεν είναι ένα ακέραιο κυριολεκτικό. Στην πραγματικότητα είναι ένας συνδυασμός ενός τελεστή άρνησης και του ακέραιου κυριολεκτικού 10.

Μερικά παραδείγματα κυριολεκτικών πραγματικών αριθμών είναι

```
23.1  0.0  3.14159
```

Και τα ακέραια και τα πραγματικά κυριολεκτικά μπορούν επίσης να χρησιμοποιήσουν την εκθετική σημειογραφία, στην οποία ο αριθμός ακολουθείται από το γράμμα ‘E’ ή ‘e’, και μια τιμή εκθέτη. Αυτό υποδεικνύει μια δύναμη του 10 με την οποία ο αριθμός πολλαπλασιάζεται. Για κυριολεκτικά ακέραιων αριθμών, ο εκθέτης δεν πρέπει να είναι αρνητικός, ενώ για τα κυριολεκτικά πραγματικών αριθμών, μπορεί να είναι είτε θετικός είτε αρνητικός. Μερικά παραδείγματα κυριολεκτικών ακέραιων αριθμών που χρησιμοποιούν την εκθετική σημειογραφία είναι

```
46E5  1E+12  19e00
```

Μερικά παραδείγματα κυριολεκτικών πραγματικών αριθμών που χρησιμοποιούν την εκθετική σημειογραφία είναι

```
1.234E09  98.6E+21  34.0e–08
```

Τα κυριολεκτικά ακέραιων και πραγματικών αριθμών μπορούν επίσης να παρασταθούν σε μια βάση διαφορετική από τη βάση του 10. Στην πραγματικότητα, η βάση μπορεί να είναι οποιοσδήποτε ακέραιος αριθμός μεταξύ 2 και 16. Για να το κάνουμε αυτό, περιβάλλουμε τον αριθμό με χαρακτήρες δίεσης (‘#’), των οποίων προηγείται η βάση. Για βάσεις μεγαλύτερες από 10, τα γράμματα ‘A’ μέχρι ‘F’ (ή ‘a’ μέχρι ‘f’) χρησιμοποιούνται για να αναπαραστήσουν τα ψηφία 10 μέχρι 15. Για παράδειγμα, ακολουθούν διάφοροι τρόποι για να γράψουμε την τιμή 253:

```
2#11111101#  16#FD#  16#0fd#  8#0375#
```

Όμοια, η τιμή 0.5 μπορεί να αναπαρασταθεί ως εξής

```
2#0.100#   8#0.4#   12#0.6#
```

Σημειώστε ότι σε όλες αυτές τις περιπτώσεις, η ίδια η βάση εκφράζεται *σε δεκαδική μορφή*.

Τα κυριολεκτικά με βάση μπορούν επίσης να χρησιμοποιήσουν εκθετική σημειογραφία. Σε αυτήν την περίπτωση, ο εκθέτης, που εκφράζεται σε δεκαδική μορφή, επισυνάπτεται στον αριθμό με τη βάση μετά από τον τελικό χαρακτήρα δίεσης. Ο εκθέτης αναπαριστά τη δύναμη της βάσης με την οποία ο αριθμός πολλαπλασιάζεται. Για παράδειγμα, ο αριθμός 1024 θα μπορούσε να αναπαρασταθεί από τα κυριολεκτικά ακέραιων αριθμών:

```
2#1#E10   16#4#E2   10#1024#E+00
```

Τέλος, σαν μια βοήθεια στην αναγνωσιμότητα των μακριών αριθμών, μπορούμε να συμπεριλάβουμε χαρακτήρες υπογράμμισης (underline characters) ως διαχωριστές μεταξύ των ψηφίων. Οι κανόνες για να συμπεριλάβουμε χαρακτήρες υπογράμμισης είναι παρόμοιοι με εκείνους για τα αναγνωριστικά - δηλαδή δεν επιτρέπεται να εμφανιστούν στην αρχή ή στο τέλος ενός αριθμού, ούτε επιτρέπεται να εμφανιστούν δύο στη σειρά. Μερικά παραδείγματα είναι

```
123_456   3.141_592_6   2#1111_1100_0000_0000#
```

Χαρακτήρες

Ένα κυριολεκτικό χαρακτήρα μπορεί να γραφτεί σε κώδικα VHDL εσωκλείοντάς το σε απλά εισαγωγικά. Οποιοιδήποτε από τους εκτυπώσιμους χαρακτήρες στο πρότυπο σύνολο χαρακτήρων (συμπεριλαμβανομένου του κενού διαστήματος) μπορούν να γραφτούν κατ' αυτό τον τρόπο. Μερικά παραδείγματα είναι

```
'A'  -- κεφαλαίο γράμμα
'z'  -- πεζό γράμμα
','  -- ο χαρακτήρας στίξης κόμμα
'"'  -- ο χαρακτήρας στίξης μονά εισαγωγικά
' '  -- ο χαρακτήρας κενού διαστήματος
```

Αλφαριθμητικά

Ένα αλφαριθμητικό κυριολεκτικό αναπαριστά μια ακολουθία χαρακτήρων και γράφεται εσωκλείοντας τους χαρακτήρες σε διπλά εισαγωγικά. Το αλφαριθμητικό μπορεί να περιλαμβάνει οποιοδήποτε αριθμό χαρακτήρων (συμπεριλαμβανομένου του μηδενός), αλλά πρέπει να χωράει ολόκληρο σε μια γραμμή. Μερικά παραδείγματα είναι

```
"A string"
"We can include any printing characters (e.g., &%@^*) in a string!!"
"00001111ZZZZ"
""   -- κενό αλφαριθμητικό
```

Εάν χρειαστεί να συμπεριλάβουμε ένα χαρακτήρα διπλών εισαγωγικών σε ένα αλφαριθμητικό, γράφουμε δύο χαρακτήρες διπλών εισαγωγικών μαζί. Το ζευγάρι ερμηνεύεται ως ένας μονός χαρακτήρας στο αλφαριθμητικό. Για παράδειγμα:

```
"A string in a string: ""A string"". "
```

Εάν χρειαστεί να γράψουμε ένα αλφαριθμητικό που είναι μακρύτερο από αυτό που μπορεί να χωρέσει σε μια γραμμή, μπορούμε να χρησιμοποιήσουμε τον τελεστή συνένωσης ("&") για να ενώσουμε δύο υπό-αλφαριθμητικά μαζί. Για παράδειγμα:

```
"If a string will not fit on one line, "
& "then we can break it into parts on separate lines."
```

Ψηφιοσειρές

Η VHDL περιλαμβάνει τιμές που αναπαριστούν δυαδικά ψηφία (bits), τα οποία μπορούν να είναι είτε ‘0’ είτε ‘1’. Ένα κυριολεκτικό ψηφιοσειράς αναπαριστά μια ακολουθία αυτών των τιμών δυαδικών ψηφίων. Αναπαριστάται από μια ακολουθία ψηφίων, που εσωκλείονται σε διπλά εισαγωγικά και των οποίων προηγείται ένας χαρακτήρας που καθορίζει τη βάση των ψηφίων. Ο καθοριστής βάσης μπορεί να είναι ένας από τους παρακάτω:

- B για δυαδική βάση,
- O για οκταδική (βάση 8) και
- X για δεκαεξαδική (βάση 16).

Για παράδειγμα, μερικά κυριολεκτικά ψηφιοσειρών που καθορίζονται στο δυαδικό σύστημα είναι

```
B"0100011"   B"10"   b"1111_0010_0001"   B""
```

Παρατηρήστε ότι μπορούμε να συμπεριλάβουμε χαρακτήρες υπογράμμισης στα κυριολεκτικά ψηφιοσειρών για να ξεχωρίσουμε γειτονικά ψηφία. Οι χαρακτήρες υπογράμμισης δεν επηρεάζουν την έννοια του κυριολεκτικού - απλά καθιστούν το κυριολεκτικό πιο αναγνώσιμο. Ο καθοριστής βάσης μπορεί να είναι κεφαλαίος ή πεζός χαρακτήρας. Το τελευταίο από τα παραπάνω παραδείγματα δείχνει μια κενή ψηφιοσειρά.

Εάν ο καθοριστής βάσης είναι οκταδικός, μπορούν να χρησιμοποιηθούν τα ψηφία ‘0’ μέχρι ‘7’. Κάθε ψηφίο αναπαριστά ακριβώς τρία δυαδικά ψηφία (bit) στην ακολουθία. Μερικά παραδείγματα είναι

```
O"372"    -- ισοδύναμο με το B"011_111_010"
o"00"   -- ισοδύναμο με το B"000_000"
```

Εάν ο καθοριστής βάσης είναι δεκαεξαδικός, μπορούν να χρησιμοποιηθούν τα ψηφία ‘0’ μέχρι ‘9’ και ‘A’ μέχρι ‘F’ ή ‘a’ μέχρι ‘f’ (που αναπαριστούν τα ψηφία 10 μέχρι 15). Στο δεκαεξαδικό, κάθε ψηφίο αναπαριστά ακριβώς τέσσερα δυαδικά ψηφία (bit). Μερικά παραδείγματα είναι

```
X"FA"  -- ισοδύναμο με το B"1111_1010"
x"0d"   -- ισοδύναμο με το B"0000_1101"
```

Παρατηρήστε ότι το O"372" δεν είναι το ίδιο με το X"FA", αφού το πρώτο είναι μια ακολουθία εννέα bit, ενώ το τελευταίο είναι μια ακολουθία οκτώ bit.

Περιγραφές Συντακτικού

Στο υπόλοιπο αυτού του βιβλίου, περιγράφουμε τους κανόνες σύνταξης χρησιμοποιώντας μια σημειογραφία που βασίζεται στην εκτεταμένη μορφή Backus-Naur (Extended Backus-Naur Form, EBNF). Αυτοί οι κανόνες καθορίζουν πώς μπορούμε να συνδυάσουμε λεξικολογικά στοιχεία ώστε να σχηματίσουμε έγκυρες περιγραφές VHDL. Είναι χρήσιμο να έχουμε μια καλή γνώση της λειτουργίας των συντακτικών κανόνων, δεδομένου ότι οι αναλυτές VHDL αναμένουν έγκυρες περιγραφές VHDL ως είσοδο. Τα μηνύματα λάθους που σε διαφορετική περίπτωση παράγουν μπορεί σε μερικές περιπτώσεις να μας φανούν αινιγματικά εάν δεν είμαστε γνώστες των συντακτικών κανόνων.

Η ιδέα πίσω από τη σημειογραφία EBNF είναι να διαιρεθεί η γλώσσα σε *συντακτικές κατηγορίες (syntactic categories)*. Για κάθε συντακτική κατηγορία γράφουμε έναν κανόνα που περιγράφει πώς να χτίσουμε μια φράση VHDL αυτής της κατηγορίας συνδυάζοντας λεξικολογικά στοιχεία και φράσεις άλλων κατηγοριών. Αυτοί οι κανόνες είναι ανάλογοι με τους κανόνες της Αγγλικής γραμματικής. Για παράδειγμα, υπάρχουν κανόνες που περιγράφουν μια πρόταση από την άποψη ενός υποκειμένου και ενός κατηγορήματος, και αυτό περιγράφει ένα κατηγόρημα από την άποψη ενός ρήματος και μιας φράσης αντικειμένου. Στους κανόνες για την Αγγλική γραμματική, η "πρόταση", το "υποκείμενο", το "κατηγόρημα", και τα λοιπά, είναι συντακτικές κατηγορίες.

Στη σημειογραφία EBNF, γράφουμε έναν κανόνα με τη συντακτική κατηγορία που καθορίζουμε στα αριστερά ενός σημαδιού "⇐" (που διαβάζεται ως "ορίζεται να είναι"), και ένα γλωσσικό πρότυπο στα δεξιά. Το απλούστερο είδος προτύπου είναι μια συλλογή στοιχείων στη σειρά, για παράδειγμα:

```
variable_assignment ⇐ target := expression ;
```

Αυτός ο κανόνας υποδεικνύει ότι μια φράση VHDL της κατηγορίας "variable_assignment" ορίζεται να είναι μια φράση της κατηγορίας "target", που ακολουθείται από το σύμβολο “:=”, που ακολουθείται από μια φράση της κατηγορίας “expression”, που ακολουθείται από το σύμβολο “;”. Για να διαπιστώσουμε εάν η φράση VHDL

```
d0 := 25 + 6;
```

είναι συντακτικά έγκυρη, θα έπρεπε να ελέγξουμε τους κανόνες για “target” και “expression”. Όπως συμβαίνει, τα “d0” και “25+6” είναι έγκυρες υποφράσεις, έτσι ολόκληρη η φράση συμμορφώνεται με το πρότυπο του κανόνα και έτσι είναι μια έγκυρη ανάθεση μεταβλητής (variable assignment). Από την άλλη μεριά, η πρόταση

```
25 fred := x if := .
```

δεν μπορεί ενδεχομένως να είναι μια έγκυρη ανάθεση μεταβλητής, δεδομένου ότι δεν ταιριάζει με το πρότυπο στη δεξιά πλευρά του κανόνα.

Το επόμενο είδος κανόνα που εξετάζουμε είναι ένα που επιτρέπει τη χρήση ενός προαιρετικού συστατικού σε μια φράση. Υποδεικνύουμε το προαιρετικό μέρος εσωκλείοντάς το μεταξύ των συμβόλων “[” και “]”. Για παράδειγμα:

```
function_call ⇐ name [ ( association_list ) ]
```

Αυτό υποδεικνύει ότι μια κλήση συνάρτησης (function call) αποτελείται από ένα όνομα (name) που μπορεί να ακολουθηθεί από μια λίστα συσχέτισης (association list) σε παρενθέσεις.

Σε πολλούς κανόνες, χρειάζεται να καθορίσουμε ότι μια φράση είναι προαιρετική, αλλά εάν είναι παρούσα, μπορεί να επαναληφθεί όσες φορές απαιτείται. Για παράδειγμα, σε αυτόν τον απλουστευμένο κανόνα για μια πρόταση διεργασίας (process statement):

process_statement ⇐

process is
{ process_declarative_item }

begin
{ sequential_statement }

end process ;

τα σγουρά άγκιστρα καθορίζουν ότι μια διεργασία μπορεί να περιλαμβάνει κανένα ή περισσότερα δηλωτικά στοιχεία της διεργασίας (process declarative items) και καμία ή περισσότερες ακολουθιακές προτάσεις (sequential statements). Μια περίπτωση που προκύπτει συχνά στους κανόνες της VHDL είναι ένα πρότυπο που αποτελείται από κάποια κατηγορία που ακολουθείται από καμία ή περισσότερες επαναλήψεις αυτής της κατηγορίας. Σε αυτήν την περίπτωση, χρησιμοποιούμε τελείες μέσα στα άγκιστρα για να αναπαραστήσουμε την κατηγορία που επαναλαμβάνεται, παρά να το καταγράψουμε πάλι πλήρως. Για παράδειγμα, ο κανόνας

case_statement ⇐

case expression **is**

case_statement_alternative

{ … }

end case ;

υποδεικνύει ότι μια πρόταση case (case statement) πρέπει να περιέχει τουλάχιστον μια εναλλακτική πρόταση case (case statement alternative), αλλά μπορεί να περιέχει και έναν αυθαίρετο αριθμό πρόσθετων εναλλακτικών προτάσεων case, όσες απαιτούνται. Εάν υπάρχει μια ακολουθία κατηγοριών και συμβόλων που προηγούνται των αγκίστρων, οι τελείες αναπαριστούν μόνο το τελευταίο στοιχείο της ακολουθίας. Κατά συνέπεια, στο παραπάνω παράδειγμα, οι τελείες αναπαριστούν μόνο την εναλλακτική πρόταση case (case statement alternative), και όχι την ακολουθία "**case** expression **is** case_statement_alternative".

Χρησιμοποιούμε επίσης τη σημειογραφία με τις τελείες στις περιπτώσεις όπου απαιτείται ένας κατάλογος μιας ή περισσότερων επαναλήψεων μιας φράσης, αλλά κάποιο σύμβολο οριοθέτησης χρειάζεται μεταξύ των επαναλήψεων. Για παράδειγμα, ο κανόνας

identifier_list ⇐ identifier { , … }

καθορίζει ότι μια λίστα αναγνωριστικών (identifier list) αποτελείται από ένα ή περισσότερα αναγνωριστικά (identifier), και ότι εάν υπάρχουν περισσότερα του ενός, χωρίζονται με κόμματα. Σημειώστε ότι οι τελείες αναπαριστούν πάντα μια επανάληψη της κατηγορίας που προηγείται του αριστερού άγκιστρου. Κατά συνέπεια, στον παραπάνω κανόνα, είναι το αναγνωριστικό που επαναλαμβάνεται, και όχι το κόμμα.

Πολλοί συντακτικοί κανόνες επιτρέπουν σε μια κατηγορία να απαρτίζεται από μια επιλογή από έναν αριθμό εναλλακτικών επιλογών. Ένας τρόπος για να το αναπαραστήσουμε αυτό είναι να έχουμε διάφορους ξεχωριστούς κανόνες για την κατηγορία, έναν για κάθε εναλλακτική επιλογή. Εντούτοις, είναι συχνά πιο βολικό να συνδυάζουμε εναλλακτικές επιλογές χρησιμοποιώντας το σύμβολο "|". Για παράδειγμα, ο κανόνας

mode ⇐ **in** I **out** I **inout**

καθορίζει ότι η κατηγορία "mode" μπορεί να σχηματιστεί από μια φράση που αποτελείται από μια δεσμευμένη λέξη που επιλέγεται από τις εναλλακτικές επιλογές δεσμευμένων λέξεων που απαριθμούνται.

Η τελευταία σημειογραφία που χρησιμοποιούμε στους συντακτικούς μας κανόνες είναι μια ομαδοποίηση με παρενθέσεις, χρησιμοποιώντας τα σύμβολα "(" και ")". Αυτά απλά εξυπηρετούν την ομαδοποίηση τμήματος ενός προτύπου, έτσι ώστε να μπορούμε να αποφύγουμε οποιαδήποτε ασάφεια που διαφορετικά θα μπορούσε να προκύψει. Για παράδειγμα, η χρησιμοποίηση παρενθέσεων στον κανόνα

term ⇐ factor { (* I / I **mod** I **rem**) factor }

καθιστά σαφές ότι ένας παράγοντας (factor) μπορεί να ακολουθηθεί από ένα από τα σύμβολα τελεστών, και έπειτα από έναν άλλο παράγοντα. Χωρίς τις παρενθέσεις, ο κανόνας θα ήταν

term ⇐ factor { * I / I **mod** I **rem** factor }

υποδεικνύοντας ότι ένας παράγοντας μπορεί να ακολουθηθεί από έναν από τους τελεστές "*", "/" ή **mod** μόνο, ή από τον τελεστή **rem** και έπειτα από έναν άλλο παράγοντα. Αυτό βέβαια δεν είναι αυτό που είχαμε κατά νου. Ο λόγος για αυτήν την ανακριβή ερμηνεία είναι ότι υπάρχει μια *προτεραιότητα (precedence)*, ή σειρά προτεραιότητας, στη σημειογραφία EBNF που χρησιμοποιούμε. Ελλείψει των παρενθέσεων, μια ακολουθία συστατικών προτύπου που

ακολουθεί το ένα το άλλο θεωρείται ως μια ομάδα με υψηλότερη προτεραιότητα από συστατικά που χωρίζονται από σύμβολα "|".

Αυτή η σημειογραφία EBNF είναι επαρκής για να περιγράψει την πλήρη γραμματική της VHDL. Εντούτοις, υπάρχουν συχνά επιπλέον περιορισμοί σε μια περιγραφή VHDL που αφορούν την έννοια των λεξικολογικών στοιχείων που χρησιμοποιούνται. Για παράδειγμα, μια περιγραφή που καθορίζει τη σύνδεση ενός σήματος σε ένα ονοματισμένο αντικείμενο που προσδιορίζει ένα συστατικό και όχι μια θύρα δεν είναι σωστή, ακόμα κι αν πιθανώς συμμορφώνεται με τους συντακτικούς κανόνες. Για να αποφευχθούν τέτοια προβλήματα, πολλοί κανόνες περιλαμβάνουν πρόσθετες πληροφορίες σχετικά με την έννοια ενός χαρακτηριστικού γνωρίσματος της γλώσσας. Για παράδειγμα, ο κανόνας που παρουσιάστηκε παραπάνω και περιγράφει πως σχηματίζεται μια κλήση συνάρτησης (function call) επαυξάνεται με αυτόν τον τρόπο:

function_call ⇐ *function*_name [(*parameter*_association_list)]

Το πρόθεμα που είναι γραμμένο με πλάγια στοιχεία σε μια συντακτική κατηγορία στο πρότυπο παρέχει απλά τις σημασιολογικές πληροφορίες. Αυτός ο κανόνας υποδεικνύει ότι το όνομα δεν μπορεί να είναι απλά οποιοδήποτε όνομα, αλλά πρέπει να είναι το όνομα μιας συνάρτησης. Ομοίως, η λίστα συσχέτισης πρέπει να περιγράφει τις παραμέτρους που παρέχονται στη συνάρτηση. Οι σημασιολογικές πληροφορίες είναι για το δικό μας όφελος ως σχεδιαστές που διαβάζουν τον κανόνα, για να μας βοηθήσουν να κατανοήσουμε την επιδιωκόμενη σημασιολογία. Όσον αφορά τη σύνταξη, ο κανόνας είναι ισοδύναμος με τον αρχικό κανόνα χωρίς τα πλάγια τμήματα.

Στα επόμενα κεφάλαια, θα εισαγάγουμε κάθε νέο χαρακτηριστικό γνώρισμα της VHDL περιγράφοντας τη σύνταξή του με τη χρήση κανόνων EBNF, και έπειτα θα περιγράφουμε την έννοια και τη χρήση του χαρακτηριστικού μέσω παραδειγμάτων. Σε πολλές περιπτώσεις, θα αρχίσουμε με μια απλουστευμένη έκδοση της σύνταξης ώστε να κάνουμε ευκολότερη την εκμάθηση της περιγραφής και θα επιστρέψουμε με πλήρης στοιχεία για τη σύνταξη σε επόμενο κεφάλαιο.

Κεφάλαιο 2: Βαθμωτοί Τύποι Δεδομένων και Πράξεις

Η έννοια του τύπου είναι πολύ σημαντική όταν περιγράφουμε αντικείμενα δεδομένων σε ένα VHDL μοντέλο. Ο τύπος ενός αντικειμένου ορίζει το σύνολο των τιμών που μπορεί να λάβει το αντικείμενο, καθώς επίσης και το σύνολο των πράξεων που μπορούν να εκτελεσθούν σε αυτές τις τιμές. Ένας βαθμωτός τύπος (scalar type) αποτελείται από ενιαίες, αδιαίρετες τιμές. Σε αυτό το κεφάλαιο εξετάζουμε τους βασικούς βαθμωτούς τύπους που παρέχει η VHDL και βλέπουμε πως μπορούν να χρησιμοποιηθούν για να ορίσουν τα αντικείμενα που περιγράφουν την εσωτερική κατάσταση μιας λειτουργικής μονάδας.

2.1 Σταθερές και Μεταβλητές

Ένα αντικείμενο (object) σε ένα VHDL μοντέλο ορίζεται από ένα όνομα και μία τιμή η οποία ανήκει σε έναν προκαθορισμένο τύπο. Υπάρχουν τέσσερις κατηγορίες αντικειμένων: σταθερές (constants), μεταβλητές (variables), σήματα (signals) και αρχεία (files). Σε αυτό το κεφάλαιο, εξετάζουμε τις σταθερές και τις μεταβλητές, ενώ με τα σήματα θα ασχοληθούμε εκτενώς στο Κεφάλαιο 5. Οι σταθερές και οι μεταβλητές είναι αντικείμενα στα οποία αποθηκεύονται δεδομένα για να χρησιμοποιηθούν από ένα μοντέλο. Η διαφορά μεταξύ τους είναι ότι η τιμή μιας σταθεράς δεν μπορεί να αλλάξει, ενώ η τιμή μιας μεταβλητής μπορεί να αλλάξει όσες φορές απαιτείται κατά τη διάρκεια εκτέλεσης του μοντέλου. Η τιμή μιας μεταβλητής αλλάζει με τη χρήση πρότασης ανάθεσης μεταβλητής (variable assignment statement).

2.1.1 Δηλώσεις Σταθερών και Μεταβλητών

Τόσο οι σταθερές όσο και οι μεταβλητές πρέπει να δηλωθούν προτού χρησιμοποιηθούν σε ένα μοντέλο. Μια *δήλωση (declaration)* εισάγει απλά το όνομα του αντικειμένου, ορίζει τον τύπο του και του δίνει (προαιρετικά) μια αρχική τιμή. Ο συντακτικός κανόνας για μια δήλωση σταθεράς είναι

constant_declaration ⇐

constant identifier { , … } : subtype_indication [:= expression] ;

Τα αναγνωριστικά (identifiers) του συντακτικού κανόνα είναι τα ονόματα των σταθερών που δηλώνονται (μια σταθερά ανά όνομα) και η ένδειξη υποτύπου (subtype_indication) που ορίζει τον τύπο όλων των σταθερών. Στις επόμενες ενότητες αυτού του κεφαλαίου εξετάζουμε λεπτομερώς τους τρόπους ορισμού του τύπου. Το προαιρετικό μέρος του συντακτικού κανόνα είναι μια παράσταση (expression) που καθορίζει την τιμή που ανατίθεται σε κάθε σταθερά. Αυτό το μέρος μπορεί να παραλειφθεί μόνο σε ορισμένες ειδικές περιπτώσεις με τις οποίες δεν θα ασχοληθούμε. Ακολουθούν μερικά παραδείγματα δήλωσης σταθερών:

```
constant number_of_bytes : integer := 4;
constant number_of_bits : integer := 8 * number_of_bytes;
constant e : real := 2.718281828;
constant prop_delay : time := 3 ns;
constant size_limit, count_limit : integer := 255;
```

Ο λόγος που χρησιμοποιούμε σταθερές είναι για να αποδώσουμε ένα όνομα και ένα ρητά καθορισμένο τύπο σε μία τιμή, και στη συνέχεια να χρησιμοποιούμε τη σταθερά στην περιγραφή του μοντέλου αντί να γράφουμε απευθείας την κυριολεκτική της τιμή (literal value). Αυτό καθιστά το μοντέλο περισσότερο ευανάγνωστο, δεδομένου ότι το όνομα και ο τύπος της σταθεράς εμπεριέχουν περισσότερες πληροφορίες για τη χρήση για την οποία προορίζεται το αντικείμενο από ότι η κυριολεκτική τιμή του. Επιπλέον, εάν πρέπει να αλλάξουμε την τιμή ενός αντικειμένου καθώς το μοντέλο εξελίσσεται, αρκεί μόνο να ενημερώσουμε τη δήλωση της σταθεράς. Αυτό είναι ευκολότερο και πιο αξιόπιστο από το να προσπαθεί κάποιος να εντοπίσει και να ενημερώσει όλες τις εμφανίσεις της συγκεκριμένης τιμής σε όλο το μοντέλο. Επομένως, η χρήση σταθερών στην περιγραφή των μοντέλων αποτελεί ορθή σχεδιαστική πρακτική.

Η μορφή μιας δήλωσης μεταβλητής είναι παρόμοια με μια δήλωση σταθεράς. Ο συντακτικός κανόνας είναι

variable_declaration ⇐

variable identifier { , … } : subtype_indication [:= expression] ;

Και στη δήλωση μιας μεταβλητής η αρχικοποίηση είναι προαιρετική. Εάν παραλείψουμε την παράσταση αρχικοποίησης από τη δήλωση μιας μεταβλητής, η εξ'ορισμού (default) αρχική τιμή της μεταβλητής εξαρτάται από τον τύπο της. Για τους βαθμωτούς τύπους, η εξ'ορισμού αρχική τιμή είναι η αριστερότερη τιμή του τύπου. Για παράδειγμα, για τους ακέραιους αριθμούς είναι ο μικρότερος ακέραιος αριθμός που μπορεί να αναπαρασταθεί από το σύστημα. Ακολουθούν μερικά παραδείγματα δήλωσης μεταβλητών

```
variable index : integer := 0;
variable sum, average, largest : real;
variable start, finish : time := 0 ns;
```

Εάν σε μία δήλωση εισάγουμε περισσότερα ονόματα μεταβλητών, είναι το ίδιο με το να γράψουμε ξεχωριστές δηλώσεις, μία για κάθε μεταβλητή. Για παράδειγμα, η τελευταία από τις παραπάνω δηλώσεις ισοδυναμεί με τις δύο ακόλουθες δηλώσεις

```
variable start : time := 0 ns;
variable finish : time := 0 ns;
```

Οι δηλώσεις σταθερών και μεταβλητών μπορούν να εμφανιστούν σε διαφορετικά μέρη ενός VHDL μοντέλου, συμπεριλαμβανομένου και του τμήματος δήλωσης των διεργασιών. Σε αυτήν την περίπτωση, το δηλωμένο αντικείμενο, σταθερά ή μεταβλητή, είναι ορατό μόνο εντός της διεργασίας. Ένας περιορισμός στο μέρος όπου μπορεί να δηλωθεί μια μεταβλητή είναι ότι δεν μπορεί να δηλωθεί σε σημείο του VHDL μοντέλου από όπου μπορεί να είναι προσπελάσιμη από περισσότερες από μια διεργασίες. Αυτός ο περιορισμός αποσκοπεί στο να αποτρέψει παράξενα αποτελέσματα που πιθανώς να συμβούν εάν οι διεργασίες πρόκειται να τροποποιήσουν την μεταβλητή με απροσδιόριστη σειρά. Η εξαίρεση σε αυτόν τον κανόνα είναι όταν μια μεταβλητή δηλώνεται σαν *κοινόχρηστη* μεταβλητή (*shared* variable).

ΠΑΡΑΔΕΙΓΜΑ

Η Εικόνα 2-1 περιγράφει ένα σώμα αρχιτεκτονικής που δείχνει πώς εισάγουμε δηλώσεις σταθερών και μεταβλητών σε ένα VHDL μοντέλο.

ΕΙΚΟΝΑ 2-1

```
architecture sample of ent is
    constant pi : real := 3.14159;
begin
    process is
        variable counter : integer;
    begin
        …  -- statements using pi and counter
    end process;
end architecture sample;
```

Ένα σώμα αρχιτεκτονικής που δείχνει τις δηλώσεις μίας σταθεράς pi *και μίας μεταβλητής* counter.

2.1.2 Ανάθεση Μεταβλητής

Εφόσον μια μεταβλητή έχει δηλωθεί, η τιμή της μπορεί να τροποποιηθεί από μια πρόταση ανάθεσης. Η σύνταξη μιας πρότασης ανάθεσης μεταβλητής (variable assignment statement) δίνεται από τον κανόνα:

variable_assignment_statement ⇐ [label :] name := expression ;

Η προαιρετική ετικέτα έχει το νόημα του προσδιορισμού της πρότασης ανάθεσης. Το όνομα σε μια πρόταση ανάθεσης μεταβλητής προσδιορίζει τη μεταβλητή που πρόκειται να τροποποιηθεί, και η παράσταση αξιολογείται για να υπολογιστεί η νέα τιμή. Ο τύπος αυτής της τιμής πρέπει να ταιριάζει με τον τύπο της μεταβλητής. Όλες οι λεπτομέρειες για το πώς συντάσσεται μια παράσταση καλύπτονται στο υπόλοιπο αυτού του κεφαλαίου. Προς το παρόν, απλά θεωρήστε τις παραστάσεις ως συνηθισμένους συνδυασμούς αναγνωριστικών και κυριολεκτικών με τελεστές. Εδώ είναι μερικά παραδείγματα προτάσεων ανάθεσης:

```
program_counter := 0;
index := index + 1;
```

Η πρώτη ανάθεση θέτει την τιμή της μεταβλητής program_counter στο μηδέν, αντικαθιστώντας οποιαδήποτε προηγούμενη τιμή. Το δεύτερο παράδειγμα αυξάνει την τιμή της μεταβλητής index κατά ένα.

Είναι σημαντικό να σημειώσουμε τη διαφορά μεταξύ μιας πρότασης ανάθεσης μεταβλητής, που παρουσιάζεται εδώ, και μιας πρότασης ανάθεσης σήματος. Μια ανάθεση μεταβλητής ενημερώνει αμέσως τη μεταβλητή με τη νέα τιμή. Από την άλλη μεριά, μια ανάθεση σήματος χρονοπρογραμματίζει την εφαρμογή της νέας τιμής στο σήμα σε κάποια μελλοντική χρονική στιγμή. Θα επιστρέψουμε στις αναθέσεις σημάτων στο Κεφάλαιο 5. Λόγω της σημαντικής διαφοράς μεταξύ των δύο ειδών ανάθεσης, η VHDL χρησιμοποιεί ξεχωριστά σύμβολα: ": =" για την ανάθεση μεταβλητής και "<=" για την ανάθεση σήματος.

VHDL-87

Οι προτάσεις ανάθεσης μεταβλητών δεν επιτρέπεται να έχουν ετικέτες στη VHDL-87.

2.2 Βαθμωτοί Τύποι

Η έννοια του *τύπου* είναι πολύ σημαντική στη VHDL. Λέμε ότι η VHDL είναι μια γλώσσα με *ισχυρούς τύπους (strongly typed)*, που σημαίνει ότι κάθε αντικείμενο μπορεί μόνο να λάβει τιμές του τύπου που έχει δηλωθεί. Επιπλέον, ο ορισμός κάθε πράξης περιλαμβάνει τους τύπους τιμών στους οποίους η πράξη μπορεί να εφαρμοστεί. Ο στόχος της αυστηρής τυποποίησης είναι να συμβάλλει στην ανίχνευση λαθών σε ένα αρχικό στάδιο της διαδικασίας σχεδίασης, και συγκεκριμένα, όταν αναλύεται ένα μοντέλο.

Σε αυτήν την ενότητα, δείχνουμε πώς δηλώνεται ένας νέος τύπος. Στη συνέχεια δείχνουμε πώς ορίζουμε διαφορετικούς *βαθμωτούς* τύπους (*scalar* types). Βαθμωτός τύπος είναι ένας τύπος του οποίοι οι τιμές είναι αδιαίρετες. Στο Κεφάλαιο 4 θα δείξουμε πώς δηλώνουμε τύπους των οποίων οι τιμές απαρτίζονται από συλλογές στοιχειωδών τιμών.

2.2.1 Δηλώσεις Τύπων

Εισάγουμε νέους τύπους σε ένα VHDL μοντέλο χρησιμοποιώντας δηλώσεις τύπων. Η δήλωση ονομάζει έναν τύπο και ορίζει ποιες τιμές μπορούν να αποθηκευτούν στα αντικείμενα αυτού του τύπου. Ο συντακτικός κανόνας για μια δήλωση τύπου είναι

type_declaration ⇐ **type** identifier **is** type_definition ;

Ένα σημαντικό σημείο που πρέπει να τονιστεί είναι ότι εάν δύο τύποι δηλωθούν χωριστά με ταυτόσημους ορισμούς, είναι εντούτοις ξεχωριστοί και ασύμβατοι τύποι. Για παράδειγμα, εάν έχουμε δύο δηλώσεις τύπων:

```
type apples is range 0 to 100;
type oranges is range 0 to 100;
```

δεν μπορούμε να αναθέσουμε μια τιμή του τύπου apples σε μια μεταβλητή του τύπου oranges, δεδομένου ότι ανήκουν σε διαφορετικούς τύπους.

Μια σημαντική χρήση των τύπων είναι να οριστούν οι επιτρεπόμενες τιμές για τις θύρες μιας οντότητας. Στα παραδείγματα στο Κεφάλαιο 1, είδαμε να χρησιμοποιείται το όνομα τύπου bit για να καθορίσει ότι οι θύρες μπορούν να πάρουν μόνο τις τιμές '0' και '1'. Εάν ορίσουμε δικούς μας τύπους για τις θύρες, τα ονόματα των τύπων πρέπει να δηλωθούν σε ένα *πακέτο* (*package*), έτσι ώστε να είναι ορατοί στη δήλωση της οντότητας. Για παράδειγμα, υποθέστε ότι επιθυμούμε να ορίσουμε μια οντότητα αθροιστή που προσθέτει μικρούς ακέραιους αριθμούς που ανήκουν στο εύρος 0 έως 255. Γράφουμε ένα πακέτο που περιέχει τη δήλωση τύπου, ως εξής:

```
package int_types is
    type small_int is range 0 to 255;
end package int_types;
```

Αυτό ορίζει ένα πακέτο με το όνομα int_types, το οποίο παρέχει τον τύπο που ονομάζεται small_int. Το πακέτο είναι μια χωριστή μονάδα σχεδίασης και αναλύεται προτού οποιαδήποτε δήλωση οντότητας επιχειρήσει να χρησιμοποιήσει έναν τύπο που έχει δηλωθεί στο πακέτο. Μπορούμε να χρησιμοποιήσουμε τον τύπο εάν πριν τη δήλωσης της οντότητας προηγηθεί μια *φράση χρήσης* (*use clause*), για παράδειγμα:

```
use work.int_types.all;
entity small_adder is
    port ( a, b : in small_int; s : out small_int );
end entity small_adder;
```

2.2.2 Ακέραιοι Τύποι

Στη VHDL, οι ακέραιοι τύποι (integer types) έχουν τιμές που είναι ακέραιοι αριθμοί. Ένα παράδειγμα ενός ακέραιου τύπου είναι ο προκαθορισμένος τύπος integer, ο οποίος περιλαμβάνει όλους τους ακέραιους αριθμούς που μπορούν να αναπαρασταθούν σε ένα συγκεκριμένο υπολογιστή (host computer). Το πρότυπο της γλώσσας απαιτεί ο τύπος integer να περιλαμβάνει τουλάχιστον τους αριθμούς –2.147.483.647 έως +2.147.483.647 ($-2^{31} + 1$ έως $+2^{31} - 1$), αλλά κάποιες υλοποιήσεις της VHDL μπορούν να επεκτείνουν αυτό το σύνολο τιμών.

Μπορούμε να ορίσουμε ένα νέο ακέραιο τύπο χρησιμοποιώντας έναν ορισμό τύπου με περιορισμό εύρους. Ο απλουστευμένος συντακτικός κανόνας για τον ορισμό ενός ακέραιου τύπου είναι

integer_type_definition ⇐

range simple_expression (**to** I **downto**) simple_expression

ο οποίος ορίζει το σύνολο των ακέραιων αριθμών μεταξύ (και συμπεριλαμβανομένων) των τιμών που δίνονται από τις δύο παραστάσεις. Οι αξιολογήσεις των παραστάσεων πρέπει να δίνουν ακέραιες τιμές. Κάνοντας χρήση της λέξης κλειδί **to**, καθορίζουμε μια *αύξουσα σειρά*, στην οποία οι τιμές διατάσσονται από το μικρότερο στα αριστερά προς το μεγαλύτερο στα δεξιά. Από την άλλη, η χρησιμοποίηση της λέξης κλειδί **downto** ορίζει μια *φθίνουσα σειρά*, στην οποία

οι τιμές διατάσσονται από τα αριστερά προς τα δεξιά από το μεγαλύτερο στο μικρότερο. Οι λόγοι για το διαχωρισμό μεταξύ αύξουσας και φθίνουσας σειράς θα αποσαφηνιστούν αργότερα.

ΠΑΡΑΔΕΙΓΜΑ

Εδώ είναι δύο δηλώσεις ακέραιων τύπων:

```
type day_of_month is range 0 to 31;
type year is range 0 to 2100;
```

Αυτοί οι δύο τύποι είναι απολύτως διακριτοί, παρά το γεγονός ότι περιλαμβάνουν μερικές κοινές τιμές. Κατά συνέπεια εάν δηλώσουμε μεταβλητές αυτών των τύπων:

```
variable today : day_of_month := 9;
variable start_year : year := 1987;
```

θα ήταν παράνομο να γίνει η ανάθεση

```
start_year := today;
```

Παρά το γεγονός ότι ο αριθμός 9 είναι μέλος του τύπου year, θεματικά αντιμετωπίζεται όντας μέλος του τύπου day_of_month, ο οποίος είναι ασύμβατος με τον τύπο year. Αυτός ο κανόνας τύπων μας βοηθά να αποφύγουμε ακούσια ανάμιξη αριθμών που αντιπροσωπεύουν διαφορετικά είδη πραγμάτων.

Εάν επιθυμούμε να χρησιμοποιήσουμε μια αριθμητική παράσταση για να προσδιορίσουμε τα όρια του εύρους, οι τιμές που εμφανίζονται στην παράσταση πρέπει να είναι *τοπικά στατικές* (*locally static*) - που σημαίνει ότι πρέπει είναι γνωστές όταν αναλύεται το μοντέλο. Για παράδειγμα, μπορούμε να χρησιμοποιήσουμε σταθερές τιμές σε μια παράσταση ως τμήμα καθορισμού ενός εύρους:

```
constant number_of_bits : integer := 32;
type bit_index is range 0 to number_of_bits – 1;
```

Οι πράξεις που μπορούν να εκτελεσθούν στις τιμές των ακέραιων τύπων περιλαμβάνουν τις γνωστές αριθμητικές πράξεις:

+	πρόσθεση, ή συνάρτηση ταυτότητας
-	αφαίρεση, ή αλλαγή προσήμου
*	πολλαπλασιασμός
/	διαίρεση
mod	modulo
rem	υπόλοιπο διαίρεσης
abs	απόλυτη τιμή
**	ύψωση σε δύναμη

Το αποτέλεσμα μιας πράξης είναι ένας ακέραιος αριθμός του ίδιου τύπου με τον ή τους τελεστέους. Για τους δυαδικούς τελεστές (εκείνους που παίρνουν δύο τελεστέους), οι τελεστέοι πρέπει να είναι του ίδιου τύπου. Ο δεξιός τελεστέος του τελεστή ύψωσης σε δύναμη πρέπει να είναι ένας μη αρνητικός ακέραιος αριθμός. Οι τελεστές ταυτότητας και αλλαγής προσήμου είναι μοναδιαίοι, που σημαίνει ότι παίρνουν μόνο έναν απλό, δεξί τελεστέο. Το αποτέλεσμα του τελεστή ταυτότητας είναι ο ίδιος ο τελεστέος του αμετάβλητος, ενώ ο τελεστής αλλαγής προσήμου παράγει το αποτέλεσμα της πράξης μηδέν μείον τον τελεστέο. Έτσι, για παράδειγμα, όλα τα ακόλουθα παράγουν το ίδιο αποτέλεσμα:

```
A + (–B),   A – (+B),   A – B
```

Ο τελεστής διαίρεσης παράγει έναν ακέραιο αριθμό που είναι το αποτέλεσμα της διαίρεσης, με οποιοδήποτε κλασματικό μέρος να αποκόπτεται προς το μηδέν. Ο τελεστής υπολοίπου ορίζεται έτσι ώστε η σχέση

```
A = (A / B) * B + (A rem B)
```

να ικανοποιείται. Το αποτέλεσμα της πράξης A **rem** B είναι το υπόλοιπο της διαίρεσης του A δια B. Έχει το ίδιο πρόσημο με το A και απόλυτη τιμή μικρότερη από την απόλυτη τιμή του B. Για παράδειγμα:

```
5 rem 3 = 2,   (–5) rem 3 = –2   5 rem (–3) = 2,   (–5) rem (–3) = –2
```

Σημειώστε ότι σε αυτές τις παραστάσεις, οι παρενθέσεις απαιτούνται από τη γραμματική της VHDL. Οι δύο τελεστές, υπολοίπου διαίρεσης και αλλαγής προσήμου, δεν μπορούν να γραφτούν δίπλα-δίπλα. Ο τελεστής modulo προσαρμόζεται στο μαθηματικό ορισμό που ικανοποιεί τη σχέση

A = B* N + (A **mod** B) – για κάποιο ακέραιο N

Το αποτέλεσμα της πράξης A **mod** B έχει το ίδιο πρόσημο με το B και απόλυτη τιμή μικρότερη από την απόλυτη τιμή του B. Για παράδειγμα:

5 **mod** 3 = 2, (–5) **mod** 3 = 1 5 **mod** (–3) = –1, (–5) **mod** (–3) = –2

Όταν μια μεταβλητή δηλώνεται ως ακέραιος τύπος, η εξ'ορισμού αρχική τιμή της είναι η αριστερότερη τιμή του εύρους του τύπου. Για τις αύξουσες σειρές, θα είναι η μικρότερη τιμή, και για τις φθίνουσες σειρές, θα είναι η μεγαλύτερη. Εάν έχουμε τις παρακάτω δηλώσεις:

```
type set_index_range is range 21 downto 11;
type mode_pos_range is range 5 to 7;
variable set_index : set_index_range;
variable mode_pos : mode_pos_range;
```

η αρχική τιμή του set_index είναι 21, και αυτή του mode_pos είναι 5. Η αρχική τιμή μιας μεταβλητής τύπου integer είναι –2.147.483.647 ή μικρότερη, δεδομένου ότι αυτός ο τύπος ορίζεται ως αύξουσα σειρά που πρέπει να περιλαμβάνει την τιμή –2.147.483.647.

2.2.3 Τύποι Κινητής Υποδιαστολής

Οι τύποι κινητής υποδιαστολής (floating-point types) στη VHDL χρησιμοποιούνται για να αναπαραστήσουν τους πραγματικούς αριθμούς. Μιλώντας από μαθηματική σκοπιά, υπάρχει ένας άπειρος αριθμός πραγματικών αριθμών μέσα σε οποιοδήποτε διάστημα, έτσι δεν είναι δυνατό να αναπαραστήσουμε ακριβώς τους πραγματικούς αριθμούς σε έναν υπολογιστή. Ως εκ τούτου οι τύποι κινητής υποδιαστολής είναι απλά μια προσέγγιση των πραγματικών αριθμών. Ο όρος "κινητή υποδιαστολή" αναφέρεται στο γεγονός ότι οι πραγματικοί αριθμοί αναπαριστώνται χρησιμοποιώντας ένα κλασματικό τμήμα (mantissa part) και ένα εκθετικό τμήμα (exponent part). Αυτό είναι παρόμοιο με τον τρόπο με τον οποίο αναπαριστούμε τους αριθμούς στην επιστημονική σημειογραφία.

Οι τύποι κινητής υποδιαστολής στη VHDL είναι συμβατοί με τα πρότυπα του IEEE 754 ή 854 για υπολογισμό κινητής υποδιαστολής και αναπαριστώνται χρησιμοποιώντας τουλάχιστον 64 bit. Αυτό δίνει ακρίβεια περίπου 15 δεκαδικών ψηφίων, και εύρος περίπου –1,8E+308 έως +1,8E+308. Μια υλοποίηση μπορεί να επιλέξει να χρησιμοποιήσει μια μεγαλύτερη αναπαράσταση, που παρέχει αντίστοιχα και μεγαλύτερη ακρίβεια ή εύρος. Υπάρχει ένας προκαθορισμένος τύπος κινητής υποδιαστολής με το όνομα real, ο οποίος περιλαμβάνει το μέγιστο εύρος που επιτρέπεται από την αναπαράσταση κινητής υποδιαστολής της υλοποίησης. Στις περισσότερες υλοποιήσεις, αυτό θα είναι το εύρος της IEEE αναπαράστασης 64-bit διπλής ακρίβειας.

Ορίζουμε ένα νέο τύπο κινητής υποδιαστολής χρησιμοποιώντας έναν ορισμό τύπου με περιορισμό εύρους. Ο απλουστευμένος συντακτικός κανόνας για έναν ορισμό τύπου κινητής υποδιαστολής είναι

floating_type_definition ⇐

range simple_expression (**to** I **downto**) simple_expression

Αυτό είναι παρόμοιο με τον τρόπο με τον οποίο δηλώνεται ένας ακέραιος τύπος, εκτός του ότι τα όρια πρέπει να αξιολογηθούν ως αριθμοί κινητής υποδιαστολής. Μερικά παραδείγματα δηλώσεων τύπων κινητής υποδιαστολής είναι

```
type input_level is range –10.0 to +10.0;
type probability is range 0.0 to 1.0;
```

Οι πράξεις που μπορούν να εκτελεσθούν σε τιμές κινητής υποδιαστολής περιλαμβάνουν τις αριθμητικές πράξεις: πρόσθεση και ταυτότητα ("+"), αφαίρεση και αλλαγή προσήμου ("–"), πολλαπλασιασμό ("*"), διαίρεση ("/"), απόλυτη τιμή (**abs**) και ύψωση σε δύναμη ("**"). Το αποτέλεσμα μιας πράξης είναι του ίδιου τύπου κινητής υποδιαστολής με τον ή τους τελεστέους. Για τους δυαδικούς τελεστές (εκείνους που παίρνουν δύο τελεστέους), οι τελεστέοι πρέπει να είναι του ίδιου τύπου. Η εξαίρεση είναι ότι ο δεξιός τελεστέος του τελεστή ύψωσης σε δύναμη πρέπει να είναι ακέραιος. Οι τελεστές ταυτότητας και αλλαγής προσήμου είναι μοναδιαίοι (που σημαίνει ότι παίρνουν μόνο έναν απλό, δεξί τελεστέο).

Οι μεταβλητές που δηλώνονται να είναι τύπου κινητής υποδιαστολής έχουν μια εξ'ορισμού αρχική τιμή που είναι η αριστερότερη τιμή του εύρους του τύπου. Έτσι εάν δηλώσουμε μια μεταβλητή να είναι του τύπου input_level που ορίστηκε προηγουμένως:

```
variable input_A : input_level;
```

η αρχική της τιμή είναι -10,0.

VHDL-87 και VHDL-93

Στη VHDL-87 και τη VHDL-93, η ακρίβεια των τύπων κινητής υποδιαστολής είναι εγγυημένο ότι θα είναι τουλάχιστον έξι δεκαδικά ψηφία, και το εύρος τους τουλάχιστον από –1,0E+38 έως +1,0E+38. Αυτό αντιστοιχεί στην IEEE αναπαράσταση απλής ακρίβειας 32-bit. Στις υλοποιήσεις της γλώσσας επιτρέπεται να χρησιμοποιήσουν μεγαλύτερες αναπαραστάσεις. Ο προκαθορισμένος τύπος real είναι εγγυημένο ότι θα έχει ακρίβεια τουλάχιστον έξι ψηφία και εύρος τουλάχιστον από –1,0E+38 έως +1,0E+38, ανεξάρτητα από το μέγεθος της αναπαράστασης που έχει επιλεγεί από την υλοποίηση.

2.2.4 Φυσικοί Τύποι

Οι εναπομείναντες αριθμητικοί τύποι στη VHDL είναι οι φυσικοί τύποι. Χρησιμοποιούνται για να αναπαραστήσουν φυσικές ποσότητες από τον πραγματικό κόσμο, όπως το μήκος, η μάζα, ο χρόνος και το ρεύμα. Ο ορισμός ενός φυσικού τύπου περιλαμβάνει την *πρωτεύουσα μονάδα (primary unit)* μέτρησης και μπορεί επίσης να περιλαμβάνει μερικές *δευτερεύουσες μονάδες (secondary units)*, οι οποίες είναι ακέραια πολλαπλάσια της πρωτεύουσας μονάδας. Ο απλουστευμένος συντακτικός κανόνας για τον ορισμό ενός φυσικού τύπου είναι

physical_type_definition ⇐

range simple_expression (**to** I **downto**) simple_expression

units
identifier ;
{ identifier = physical_literal ; }
end units [identifier]

physical_literal ⇐ [decimal_literal I based_literal] *unit*_name

Ο ορισμός φυσικού τύπου είναι όμοιος με τον ορισμό ακέραιου τύπου, με τη διαφορά ότι έχει προστεθεί το τμήμα ορισμού μονάδων. Η πρωτεύουσα μονάδα (το πρώτο αναγνωριστικό μετά από τη λέξη κλειδί **units**) είναι η μικρότερη μονάδα που αναπαριστάται. Μπορούμε έπειτα να ορίσουμε διάφορες δευτερεύουσες μονάδες, όπως θα δούμε αμέσως μετά. Το εύρος καθορίζει τα πολλαπλάσια της πρωτεύουσας μονάδας που περιλαμβάνονται στον τύπο. Εάν το αναγνωριστικό περιλαμβάνεται στο τέλος του τμήματος ορισμού μονάδων, πρέπει να επαναληφθεί το όνομα του τύπου που ορίζεται.

ΠΑΡΑΔΕΙΓΜΑ

Εδώ είναι μια δήλωση ενός φυσικού τύπου που αναπαριστά την ηλεκτρική αντίσταση:

```
type resistance is range 0 to 1E9
    units
        ohm;
    end units resistance;
```

Οι κυριολεκτικές τιμές αυτού του τύπου γράφονται σαν αριθμητικά κυριολεκτικά ακολουθούμενα από το όνομα της μονάδας, για παράδειγμα:

```
5 ohm   22 ohm   471_000 ohm
```

Παρατηρήστε ότι πρέπει να εισάγουμε ένα κενό διάστημα πριν από το όνομα της μονάδας. Επίσης, εάν ο αριθμός είναι το κυριολεκτικό 1, μπορεί να παραλειφθεί, αφήνοντας απλά το όνομα της μονάδας. Έτσι τα ακόλουθα δύο κυριολεκτικά αντιπροσωπεύουν την ίδια τιμή:

```
ohm   1 ohm
```

Σημειώστε ότι τιμές όπως –5 ohm και 1E16 ohm δεν συμπεριλαμβάνονται στον τύπο resistance, δεδομένου ότι οι τιμές –5 και 1E16 βρίσκονται εκτός του εύρους του τύπου.

Τώρα που έχουμε δει πώς γράφουμε φυσικά κυριολεκτικά, μπορούμε να εξετάσουμε πώς καθορίζουμε τις δευτερεύουσες μονάδες σε μια δήλωση φυσικού τύπου. Αυτό το κάνουμε προσδιορίζοντας πόσες πρωτεύουσες μονάδες συνιστούν μια δευτερεύουσα μονάδα. Η δήλωσή μας για τον τύπο resistance μπορεί τώρα να επεκταθεί:

```
type resistance is range 0 to 1E9
    units
        ohm;
        kohm = 1000 ohm;
        Mohm = 1000 kohm;
    end units resistance;
```

Παρατηρήστε ότι μόλις οριστεί μια δευτερεύουσα μονάδα, μπορεί να χρησιμοποιηθεί για να καθοριστούν περαιτέρω δευτερεύουσες μονάδες. Φυσικά, οι δευτερεύουσες μονάδες δεν είναι απαραίτητο να είναι δυνάμεις του 10πλάσιου της πρωτεύουσας μονάδας; εντούτοις, ο πολλαπλασιαστής πρέπει να είναι ακέραιος αριθμός. Για παράδειγμα, ένας φυσικός τύπος για το μήκος θα μπορούσε να δηλωθεί ως εξής

```
type length is range 0 to 1E9
    units
        um;                 -- primary unit: micron
        mm = 1000 um;       -- metric units
        m = 1000 mm;
        inch = 25400 um;    -- English units
        foot = 12 inch;
    end units length;
```

Μπορούμε να γράψουμε φυσικά κυριολεκτικά αυτού του τύπου χρησιμοποιώντας τις δευτερεύουσες μονάδες, για παράδειγμα:

```
23 mm   2 foot   9 inch
```

Όταν γράφουμε φυσικά κυριολεκτικά, μπορούμε να γράψουμε μη-ακέραια πολλαπλάσια της πρωτεύουσας ή των δευτερευουσών μονάδων. Εάν η τιμή που γράφουμε δεν είναι ακριβές πολλαπλάσιο της πρωτεύουσας μονάδας, στρογγυλεύεται προς τα κάτω στο κοντινότερο πολλαπλάσιο. Για παράδειγμα, θα μπορούσαμε να γράψουμε τα ακόλουθα κυριολεκτικά του τύπου length, κάθε ένα από τα οποία αναπαριστά την ίδια τιμή:

```
0.1 inch   2.54 mm   2.540528 mm
```

Τα δύο τελευταία στρογγυλεύονται προς τα κάτω στην τιμή 2540 um, δεδομένου ότι η πρωτεύουσα μονάδα για τον τύπο length είναι το um. Εάν γράψουμε το φυσικό κυριολεκτικό 6.8 um, αυτό στρογγυλεύεται προς τα κάτω στην τιμή 6 um.

Πολλοί από τους αριθμητικούς τελεστές μπορούν να εφαρμοστούν στους φυσικούς τύπους, αλλά με μερικούς περιορισμούς. Οι τελεστές πρόσθεσης, αφαίρεσης, ταυτότητας και αλλαγής προσήμου μπορούν να εφαρμοστούν στις τιμές των φυσικών τύπων, οπότε σ'αυτή την περίπτωση παράγουν αποτελέσματα του ίδιου τύπου με τον ή τους τελεστέους. Μια τιμή ενός φυσικού τύπου μπορεί να πολλαπλασιαστεί με έναν αριθμό τύπου integer ή real για να παραγάγει μια τιμή του ίδιου φυσικού τύπου, για παράδειγμα:

```
5 mm * 6 = 30mm
```

Μια τιμή ενός φυσικού τύπου μπορεί να διαιρεθεί με έναν αριθμό τύπου integer ή real για να παραγάγει μια τιμή του ίδιου φυσικού τύπου. Επιπλέον, δύο τιμές του ίδιου φυσικού τύπου μπορούν να διαιρεθούν για να παραγάγουν έναν ακέραιο αριθμό, για παράδειγμα:

```
18 kohm / 2.0 = 9 kohm,   33 kohm / 22 ohm = 1500
```

Τέλος, ο τελεστής **abs** μπορεί να εφαρμοστεί σε μια τιμή ενός φυσικού τύπου για να παραγάγει μια τιμή του ίδιου τύπου, για παράδειγμα:

abs 2 foot = 2 foot,   **abs** (–2 foot) = 2 foot

Οι περιορισμοί έχουν νόημα όταν θεωρούμε ότι οι φυσικοί τύποι αναπαριστούν πραγματικές φυσικές ποσότητες, και η αριθμητική πρέπει να γίνει ώστε να παράγει αποτελέσματα των σωστών διαστάσεων. Δεν έχει νόημα να πολλαπλασιάσουμε δύο μήκη για να παράγουμε ένα μήκος - το αποτέλεσμα πρέπει λογικά να είναι εμβαδό. Έτσι η VHDL δεν επιτρέπει τον απ'ευθείας πολλαπλασιασμό δύο φυσικών τύπων. Αντ' αυτού, πρέπει να μετατρέψουμε τις τιμές σε αφηρημένους ακέραιους αριθμούς για να κάνουμε τον υπολογισμό, κατόπιν να μετατρέψουμε ξανά το αποτέλεσμα στον τελικό φυσικό τύπο. (Δείτε τη συζήτηση για τις ιδιότητες 'pos και 'val στην Ενότητα 2.4.)

Η μεταβλητή που δηλώνεται να είναι φυσικού τύπου έχει μια εξ'ορισμού αρχική τιμή που είναι η αριστερότερη τιμή του εύρους του τύπου. Για παράδειγμα, οι εξ'ορισμού αρχικές τιμές για τους τύπους που δηλώθηκαν παραπάνω είναι 0 ohm για τον τύπο resistance και 0 um για τον τύπο length.

VHDL-87

Ένας ορισμός φυσικού τύπου στη VHDL-87 δεν επιτρέπεται να επαναλάβει το όνομα του τύπου μετά από τις λέξεις κλειδιά **end units**.

Time

Ο προκαθορισμένος φυσικός τύπος time είναι πολύ σημαντικός στη VHDL, δεδομένου ότι χρησιμοποιείται εκτενώς για να καθορίσει καθυστερήσεις (delays). Ο ορισμός του είναι

```
type time is range implementation defined
    units
        fs;
        ps = 1000 fs;
        ns = 1000 ps;
        us = 1000 ns;
        ms = 1000 us;
        sec = 1000 ms;
        min = 60 sec;
        hr = 60 min;
    end units;
```

Εξ ορισμού, η πρωτεύουσα μονάδα fs είναι το *όριο ανάλυσης (resolution unit)* που χρησιμοποιείται όταν προσομοιώνεται ένα μοντέλο. Τιμές χρόνου μικρότερες από το όριο ανάλυσης στρογγυλεύονται προς τα κάτω σε μηδενικές μονάδες. Ένας προσομοιωτής (simulator) μπορεί να μας επιτρέψει να επιλέξουμε μια δευτερεύουσα μονάδα τύπου time ως όριο ανάλυσης. Σε αυτήν την περίπτωση, η μονάδα όλων των φυσικών κυριολεκτικών τύπου time του μοντέλου δεν πρέπει να είναι μικρότερη από το όριο ανάλυσης. Όταν το μοντέλο εκτελείται, το όριο ανάλυσης χρησιμοποιείται για να καθορίσει την ακρίβεια με την οποία οι χρονικές τιμές αναπαριστώνται. Ο λόγος που επιτρέπεται μειωμένη ακρίβεια κατ' αυτό τον τρόπο είναι για να καταστεί δυνατή η αναπαράσταση μεγαλύτερου εύρους χρονικών τιμών. Αυτό μπορεί να επιτρέψει την προσομοίωση ενός μοντέλου για μεγαλύτερη χρονική περίοδο.

2.2.5 Τύποι Απαρίθμησης

Συχνά όταν γράφουμε μοντέλα υλικού σε ένα αφηρημένο επίπεδο, είναι χρήσιμο να χρησιμοποιήσουμε ένα σύνολο ονομάτων για τις κωδικοποιημένες τιμές μερικών σημάτων, παρά να καταφύγουμε αμέσως σε μια κωδικοποίηση σε επίπεδο bit. Οι *τύποι απαρίθμησης (enumeration types)* της VHDL μας επιτρέπουν να κάνουμε κάτι τέτοιο. Για παράδειγμα, υποθέστε ότι μοντελοποιούμε έναν επεξεργαστή, και θέλουμε να ορίσουμε ονόματα για τους κωδικούς λειτουργίας της αριθμητικής μονάδας. Μια κατάλληλη δήλωση τύπου είναι

```
type alu_function is (disable, pass, add, subtract, multiply, divide);
```

Ένας τέτοιος τύπος καλείται μία *απαρίθμηση (enumeration)*, επειδή οι κυριολεκτικές τιμές που χρησιμοποιούνται απαριθμούνται σε έναν κατάλογο. Ο συντακτικός κανόνας για τους ορισμούς τύπων απαρίθμησης γενικά είναι

```
enumeration_type_definition ⇐ ( ( identifier I character_literal ) { , … } )
```

Πρέπει να υπάρχει τουλάχιστον μια τιμή στον τύπο, και κάθε τιμή μπορεί να είναι είτε ένα αναγνωριστικό, όπως στο παραπάνω παράδειγμα, είτε ένα κυριολεκτικό χαρακτήρα. Ένα παράδειγμα της τελευταίας περίπτωσης είναι

```
type octal_digit is ('0', '1', '2', '3', '4', '5', '6', '7');
```

Δοθέντος των δύο παραπάνω δηλώσεων τύπων, θα μπορούσαμε να δηλώσουμε τις μεταβλητές:

```
variable alu_op : alu_function;
variable last_digit : octal_digit := '0';
```

και να κάνουμε αναθέσεις σε αυτές:

```
alu_op := subtract;
last_digit := '7';
```

Διαφορετικοί τύποι απαρίθμησης μπορούν να περιλαμβάνουν το ίδιο αναγνωριστικό ως ένα κυριολεκτικό (αποκαλείται *υπερφόρτωση - overloading*), έτσι το πλαίσιο χρήσης πρέπει να καταστήσει σαφές σε ποιον τύπο αναφέρεται. Για να το εξηγήσουμε αυτό, θεωρήστε τις ακόλουθες δηλώσεις:

```
type logic_level is (unknown, low, undriven, high);
variable control : logic_level;
type water_level is (dangerously_low, low, ok);
variable water_sensor : water_level;
```

Εδώ το κυριολεκτικό low υπερφορτώνεται, επειδή είναι μέρος και των δύο τύπων. Εντούτοις, οι αναθέσεις

```
control := low;
water_sensor := low;
```

είναι και ο δύο αποδεκτές, εφόσον οι τύποι των μεταβλητών είναι επαρκείς για να καθορίσουν σε ποιο low αναφέρονται.

Όταν δηλώνεται μια μεταβλητή ενός τύπου απαρίθμησης, η εξ'ορισμού αρχική τιμή είναι το αριστερότερο στοιχείο της λίστας απαρίθμησης. Έτσι η εξ'ορισμού αρχική τιμή για τον τύπο logic_level είναι unknown, και για τον τύπο water_level είναι dangerously_low.

Υπάρχουν τρεις προκαθορισμένοι τύποι απαρίθμησης που ορίζονται ως εξής

```
type severity_level is (note, warning, error, failure);
type file_open_status is (open_ok, status_error, name_error, mode_error);
type file_open_kind is (read_mode, write_mode, append_mode);
```

Ο τύπος severity_level χρησιμοποιείται στις προτάσεις ισχυρισμού (assertion statements), τις οποίες θα συζητήσουμε στο Κεφάλαιο 3, και οι τύποι file_open_status και file_open_kind χρησιμοποιούνται για τις λειτουργίες αρχείων. Στο υπόλοιπο αυτής της ενότητας, εξετάζουμε τους άλλους προκαθορισμένους τύπους απαρίθμησης και τις πράξεις που μπορούν να εφαρμοστούν σε αυτούς.

VHDL-87

Οι τύποι file_open_status και file_open_kind δεν προκαθορίζονται στη VHDL-87.

Characters

Στο Κεφάλαιο 1 είδαμε πώς γράφουμε κυριολεκτικές τιμές χαρακτήρα. Αυτές οι τιμές είναι μέλη του προκαθορισμένου τύπου απαρίθμησης character, ο οποίος περιλαμβάνει όλους τους χαρακτήρες του συνόλου χαρακτήρων 8-bit ISO 8859 Latin-1. Ο ορισμός του τύπου παρουσιάζεται στην Εικόνα 2-2. Παρατηρήστε ότι αυτός ο τύπος είναι ένα παράδειγμα ενός τύπου απαρίθμησης που περιέχει ένα μίγμα αναγνωριστικών και κυριολεκτικών χαρακτήρα ως στοιχεία.

ΕΙΚΟΝΑ 2-2

```
type character is (
  nul,  soh,  stx,  etx,  eot,  enq,  ack,  bel,
  bs,   ht,   lf,   vt,   ff,   cr,   so,   si,
  dle,  dc1,  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,
  c128, c129, c130, c131, c132, c133, c134, c135,
  c136, c137, c138, c139, c140, c141, c142, c143,
  c144, c145, c146, c147, c148, c149, c150, c151,
  c152, c153, c154, c155, c156, c157, c158, c159,
  ' ',  '¡',  '¢',  '£',  '¤',  '¥',  '¦',  '§',
  '¨',  '©',  'ª',  '«',  '¬',  '-',  '®',  '¯',
  '°',  '±',  '²',  '³',  '´',  'µ',  '¶',  '·',
  '¸',  '¹',  'º',  '»',  '¼',  '½',  '¾',  '¿',
  'À',  'Á',  'Â',  'Ã',  'Ä',  'Å',  'Æ',  'Ç',
  'È',  'É',  'Ê',  'Ë',  'Ì',  'Í',  'Î',  'Ï',
  'Ð',  'Ñ',  'Ò',  'Ó',  'Ô',  'Õ',  'Ö',  '×',
  'Ø',  'Ù',  'Ú',  'Û',  'Ü',  'Ý',  'Þ',  'ß',
  'à',  'á',  'â',  'ã',  'ä',  'å',  'æ',  'ç',
  'è',  'é',  'ê',  'ë',  'ì',  'í',  'î',  'ï',
  'ð',  'ñ',  'ò',  'ó',  'ô',  'õ',  'ö',  '÷',
  'ø',  'ù',  'ú',  'û',  'ü',  'ý',  'þ',  'ÿ');
```

Ο ορισμός του προκαθορισμένου τύπου απαρίθμησης character.

Οι πρώτοι 128 χαρακτήρες σε αυτήν την απαρίθμηση είναι οι χαρακτήρες ASCII, οι οποίοι συνιστούν ένα υποσύνολο του συνόλου χαρακτήρων Latin-1. Τα αναγνωριστικά από το nul έως το usp και το del είναι οι μη-εκτυπώσιμοι ASCII χαρακτήρες ελέγχου. Οι χαρακτήρες c128 έως c159 δεν έχουν τυποποιημένα ονόματα, έτσι η VHDL τους δίνει απλώς μη περιγραφικά ονόματα βασισμένα στη θέση τους στο σύνολο χαρακτήρων. Ο χαρακτήρας στη θέση 160 είναι ένας μη-συλλαβιζόμενος χαρακτήρας κενού διαστήματος (non-braking space character), ξεχωριστός από το συνηθισμένο χαρακτήρα κενού διαστήματος (space character), και ο χαρακτήρας στη θέση 173 είναι μια ήπια παύλα (soft hyphen).

Για να εξηγήσουμε τη χρήση του τύπου character, δηλώνουμε μεταβλητές ως εξής:

```
variable cmd_char, terminator : character;
```

και έπειτα κάνουμε τις αναθέσεις

```
cmd_char := 'P';
terminator := cr;
```

VHDL-87

Εφόσον η VHDL-87 χρησιμοποιεί το σύνολων χαρακτήρων ASCII, ο προκαθορισμένος τύπος character περιλαμβάνει μόνο τους πρώτους 128 χαρακτήρες που παρουσιάζονται στην Εικόνα 2-2.

Booleans

Ένας από τους σημαντικότερους προκαθορισμένους τύπους απαρίθμησης στη VHDL είναι ο τύπος Boolean, που ορίζεται ως

```
type boolean is (false, true);
```

Αυτός ο τύπος χρησιμοποιείται για να αναπαραστήσει τιμές συνθήκης, οι οποίες μπορούν να ελέγξουν την εκτέλεση ενός μοντέλου συμπεριφοράς. Υπάρχουν διάφοροι τελεστές τους οποίους μπορούμε να εφαρμόσουμε σε τιμές διαφορετικών τύπων ώστε να παράγουμε τιμές Boolean, και συγκεκριμένα, τους σχεσιακούς και λογικούς τελεστές. Η σχεσιακοί τελεστές ισότητας ("=") και ανισότητας ("/=") μπορούν να εφαρμοστούν στους τελεστέους οποιουδήποτε τύπου (εκτός από τα αρχεία), συμπεριλαμβανομένων των σύνθετων τύπων που θα δούμε αργότερα σε αυτό το κεφάλαιο. Οι τελεστέοι πρέπει και οι δύο να είναι του ίδιου τύπου, και το αποτέλεσμα είναι μια τιμή Boolean. Για παράδειγμα, οι παραστάσεις

```
123 = 123,   'A' = 'A',   7 ns = 7 ns
```

όλες παράγουν την τιμή true (αληθής), και οι παραστάσεις

```
123 = 456,   'A' = 'z',   7 ns = 2 us
```

παράγουν την τιμή false (ψευδής).

Οι σχεσιακοί τελεστές που εξετάζουν τη διάταξη είναι οι μικρότερο-από ("<"), μικρότερο-από-ή-ίσο -("<="), μεγαλύτερο-από (">") και μεγαλύτερο-από-ή-ίσο (">="). Αυτοί μπορούν μόνο να εφαρμοστούν στις τιμές των τύπων που είναι διατεταγμένοι, συμπεριλαμβανομένων όλων των βαθμωτών τύπων που περιγράφτηκαν σε αυτό το κεφάλαιο. Όπως με τους τελεστές ισότητας και ανισότητας, οι τελεστέοι πρέπει να είναι του ίδιου τύπου, και το αποτέλεσμα είναι μια τιμή Boolean. Για παράδειγμα, οι παραστάσεις

```
123 < 456, 789 ps <= 789 ps, '1' > '0'
```

οδηγούν όλες σε true, και οι παραστάσεις

```
96 >= 102, 2 us < 4 ns, 'X' < 'X'
```

οδηγούν όλες σε false.

Οι λογικοί τελεστές **and**, **or**, **nand**, **nor**, **xor**, **xnor** και **not** παίρνουν τελεστέους που πρέπει να είναι τιμές Boolean, και παράγουν αποτελέσματα Boolean. Η Εικόνα 2-3 παρουσιάζει τα αποτελέσματα που παράγονται από τους δυαδικούς λογικούς τελεστές. Το αποτέλεσμα του τελεστή **not** είναι true εάν ο τελεστέος είναι false, και false εάν ο τελεστέος είναι true.

ΕΙΚΟΝΑ 2-3

A	B	A **and** B	A **nand** B	A **or** B	A **nor** B	A **xor** B	A **xnor** B
false	false	false	true	false	true	false	true
false	true	false	true	true	false	true	false
true	false	false	true	true	false	true	false
true	true	true	false	true	false	false	true

Ο πίνακας αληθείας για τους δυαδικούς λογικούς τελεστές.

Οι τελεστές **and**, **or**, **nand** και **nor** καλούνται τελεστές "πρόωρης-αξιολόγησης" (short-circuit), δεδομένου ότι αξιολογούν το δεξιό τελεστέο μόνο εάν ο αριστερός τελεστέος δεν καθορίζει το αποτέλεσμα. Για παράδειγμα, εάν ο αριστερός τελεστέος του τελεστή **and** είναι false, ξέρουμε ότι το αποτέλεσμα είναι false, και έτσι δεν χρειάζεται να εξετάσουμε τον άλλο τελεστέο. Αυτό είναι χρήσιμο στις περιπτώσεις όπου ο αριστερός τελεστέος είναι ένας έλεγχος που μας προστατεύει από το να προκαλέσει ένα σφάλμα ο δεξιός τελεστέος. Εξετάστε την παράσταση

```
(b /= 0) and (a/b > 1)
```

Εάν το b ήταν μηδέν και αξιολογούσαμε το δεξιό τελεστέο, θα προκαλούσαμε ένα σφάλμα λόγω της διαίρεσης με το μηδέν. Εντούτοις, επειδή ο τελεστής **and** είναι τελεστής "πρόωρης-αξιολόγησης", εάν το b ήταν μηδέν, ο αριστερός τελεστέος θα αξιολογούνταν ως false, έτσι ο δεξιός τελεστέος δεν θα αξιολογούνταν καθόλου. Για τον τελεστή **nand**, ο δεξιός τελεστέος ομοίως δεν αξιολογείται εάν ο αριστερός είναι false. Για τους τελεστέους **or** και **nor**, ο δεξιός τελεστέος δεν αξιολογείται εάν ο αριστερός είναι true.

VHDL-87

Ο λογικός τελεστής **xnor** δεν παρέχεται στη VHDL-87.

Bits

Δεδομένου ότι η VHDL χρησιμοποιείται για να μοντελοποιήσει ψηφιακά συστήματα, είναι χρήσιμο να υπάρχει ένας τύπος δεδομένων για να αναπαριστά τις τιμές bit. Ο προκαθορισμένος τύπος απαρίθμησης bit εξυπηρετεί αυτόν τον σκοπό. Ορίζεται ως

```
type bit is ('0', '1');
```

Παρατηρήστε ότι οι χαρακτήρες ‘0’ και ‘1’ είναι υπερφορτωμένοι (overloaded), εφόσον είναι μέλη και των δύο τύπων bit και character. Όπου το ‘0’ ή το ‘1’ εμφανίζεται σε ένα μοντέλο, τα συμφραζόμενα χρησιμοποιούνται για να καθορίσουν ποιος τύπος υπονοείται.

Οι λογικοί τελεστές που αναφέραμε για τις τιμές Boolean μπορούν επίσης να εφαρμοστούν στις τιμές τύπου bit, και παράγουν αποτελέσματα τύπου bit. Η τιμή ‘0’ αντιστοιχεί σε ψευδές (false), και η τιμή ‘1’ αντιστοιχεί σε αληθές (true). Έτσι, για παράδειγμα:

```
'0' and '1' = '0',   '1' xor '1' = '0'
```

Οι τελεστέοι πρέπει ακόμα να είναι του ίδιου τύπου ο ένας με τον άλλο. Κατά συνέπεια δεν είναι νόμιμο να γράψουμε

```
'0' and true
```

Η διαφορά μεταξύ των τύπων boolean και bit είναι ότι οι τιμές Boolean χρησιμοποιούνται για να μοντελοποιήσουν αφηρημένους όρους, ενώ οι τιμές bit χρησιμοποιούνται για να μοντελοποιήσουν επίπεδα λογικής του υλικού. Κατά συνέπεια, το ‘0’ αναπαριστά ένα χαμηλό επίπεδο λογικής και το ‘1’ αναπαριστά ένα υψηλό επίπεδο λογικής. Οι λογικοί τελεστές, όταν εφαρμόζονται σε τιμές bit, καθορίζονται από την άποψη της θετικής λογικής, με το ‘0’ να αναπαριστά την κατάσταση αναίρεσης (negated state) και το ‘1’ να αναπαριστά την κατάσταση ενεργοποίησης (asserted state). Εάν χρειαστεί να χειριστούμε αρνητική λογική, πρέπει να είμαστε προσεκτικοί στο γράψιμο των λογικών παραστάσεων για να πάρουμε το σωστό νόημα της λογικής. Για παράδειγμα, εάν write_enable_n, select_reg_n και write_reg_n είναι μεταβλητές bit αρνητικής λογικής, και εκτελέσουμε την ανάθεση

```
write_reg_n := not ( not write_enable_n and not select_reg_n );
```

Η μεταβλητή write_reg_n ενεργοποιείται (‘0’) μόνο εάν ενεργοποιηθούν και η write_enable_n και η select_reg_n. Διαφορετικά αναιρείται (‘1’).

Πρότυπη Λογική (Standard Logic)

Δεδομένου ότι η VHDL σχεδιάστηκε για τη μοντελοποίηση ψηφιακών συστημάτων, είναι απαραίτητο να περιλαμβάνει τύπους για να αναπαριστά ψηφιακά κωδικοποιημένες τιμές. Ο προκαθορισμένος τύπος bit που παρουσιάστηκε παραπάνω μπορεί να χρησιμοποιηθεί για αυτό σε περισσότερο αφηρημένα μοντέλα, όταν δεν ενδιαφερόμαστε για τις λεπτομέρειες των ηλεκτρικών σημάτων. Εντούτοις, όσο βελτιώνουμε τα μοντέλα μας για να συμπεριλάβουμε περισσότερη λεπτομέρεια, πρέπει να λαμβάνουμε υπόψιν τις ηλεκτρικές ιδιότητες κατά την αναπαράσταση των σημάτων. Υπάρχουν πολλοί τρόποι που μπορούμε να καθορίσουμε τύπους δεδομένων για να επιτύχουμε κάτι τέτοιο, αλλά ο οργανισμός IEEE έχει τυποποιήσει έναν τρόπο σε ένα πακέτο (package) που ονομάζεται std_logic_1164. Ένας από τους τύπους που καθορίζονται σε αυτό το πακέτο είναι ένας τύπος απαρίθμησης που ονομάζεται std_ulogic, και ορίζεται ως

```
type std_ulogic is ('U',    -- Μη αρχικοποιημένο (Uninitialized)
                    'X',    -- Ισχυρό Άγνωστο (Forcing Unknown)
                    '0',    -- Ισχυρό μηδέν (Forcing zero)
                    '1',    -- Ισχυρό ένα (Forcing one)
                    'Z',    -- Υψηλή Εμπέδηση (High Impedance)
                    'W',    -- Ασθενές Άγνωστο (Weak Unknown)
                    'L',    -- Ασθενές μηδέν (Weak zero)
                    'H',    -- Ασθενές ένα (Weak one)
                    '–' );  -- Αδιάφορο (Don't care)
```

Αυτός ο τύπος μπορεί να χρησιμοποιηθεί για να αναπαραστήσει σήματα που οδηγούνται από ενεργούς οδηγούς (ισχυρή δύναμη - forcing strength), ωμικούς οδηγούς (resistive drivers) είτε ελκτικούς προς τα πάνω (pull-ups) είτε ελκτικούς προς τα κάτω (pull-downs) (ασθενές δύναμη - weak strength) ή οδηγούς τριών καταστάσεων (tristate drivers) συμπεριλαμβανομένου μίας κατάστασης υψηλής σύνθετης αντίστασης (high-impedance). Κάθε είδος οδηγού μπορεί να οδηγήσει μία τιμή «μηδέν», «ένα» ή «άγνωστη» (unknown). Μια «άγνωστη» τιμή οδηγείται από ένα μοντέλο όταν δεν είναι δυνατό να καθοριστεί εάν το σήμα πρέπει να είναι «μηδέν» ή «ένα». Για παράδειγμα, η έξοδος μίας πύλης and είναι άγνωστη όταν οι είσοδοί της οδηγούνται από οδηγούς υψηλής σύνθετης αντίστασης. Εκτός από αυτές τις τιμές, η αριστερότερη τιμή του τύπου αναπαριστά μία «μη-αρχικοποιημένη» τιμή (uninitialized). Εάν δηλώσουμε σήματα του τύπου std_ulogic, εξ ορισμού παίρνουν ως αρχική τιμή την τιμή 'U'. Εάν ένα μοντέλο προσπαθήσει να λειτουργήσει σε αυτήν την τιμή αντί μιας πραγματικής λογικής τιμής, έχουμε ανιχνεύσει ένα λάθος σχεδίασης στο οποίο το σύστημα που μοντελοποιείται δεν τίθεται σωστά σε λειτουργία. Η τελευταία τιμή του τύπου std_ulogic είναι μία «αδιάφορη» τιμή (don't care). Αυτό χρησιμοποιείται μερικές φορές από τα εργαλεία λογικής σύνθεσης και μπορεί επίσης να χρησιμοποιηθεί κατά τον καθορισμό των διανυσμάτων δοκιμής (test vectors), για να δείξει ότι η τιμή ενός σήματος που συγκρίνεται με ένα διάνυσμα δοκιμής δεν είναι σημαντική.

Ακόμα κι αν ο τύπος std_ulogic και οι άλλοι τύποι που καθορίζονται στο πακέτο std_logic_1164 δεν είναι πραγματικά ενσωματωμένοι στη γλώσσα VHDL, μπορούμε να γράψουμε μοντέλα σαν να ήταν, με λίγη προετοιμασία. Εάν συμπεριλάβουμε τη γραμμή

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

πριν από κάθε οντότητα ή σώμα αρχιτεκτονικής που χρησιμοποιεί το πακέτο, μπορούμε να γράψουμε μοντέλα σαν να ήταν οι τύποι ενσωματωμένοι στη γλώσσα.

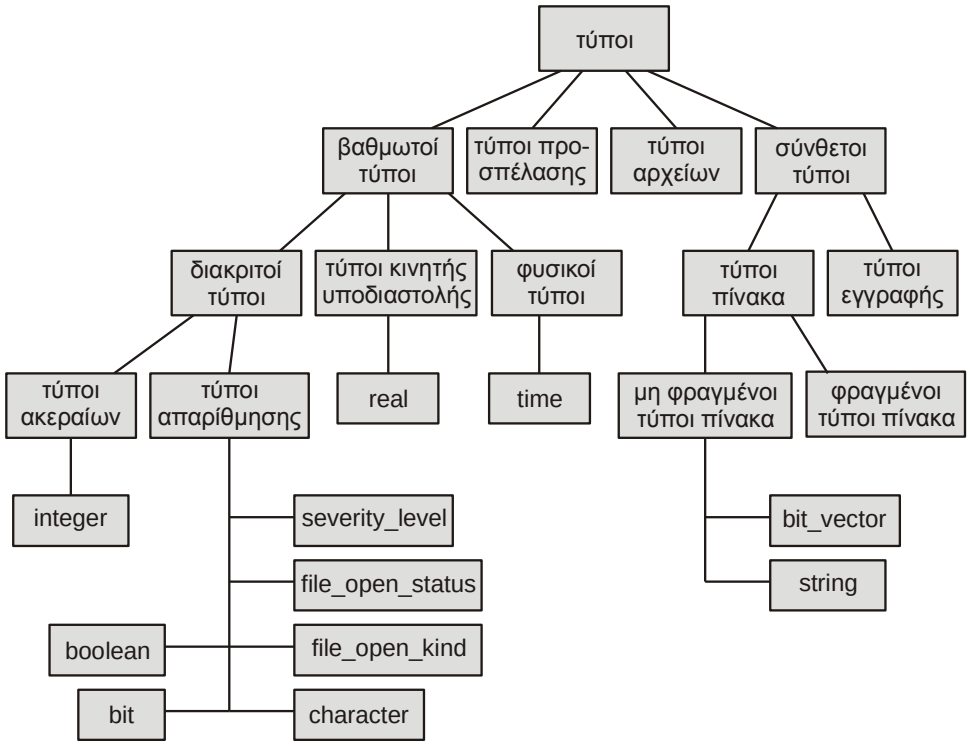
Με αυτήν την προετοιμασία κατά νου, μπορούμε τώρα να δημιουργήσουμε σταθερές, μεταβλητές και σήματα του τύπου std_ulogic. Όπως και με την ανάθεση των τιμών του τύπου, μπορούμε επίσης να χρησιμοποιήσουμε τους λογικούς τελεστές **and**, **or**, **not** και τους λοιπούς. Κάθε ένας από αυτούς λειτουργεί στις τιμές std_ulogic και επιστρέφει ένα αποτέλεσμα std_ulogic με τιμή 'U', 'X', '0' ή '1'. Οι τελεστές είναι «αισιόδοξοι», δεδομένου ότι εάν μπορούν να καθορίσουν ένα αποτέλεσμα '0' ή '1' παρά το ότι υπάρχουν άγνωστες είσοδοι, το κάνουν. Διαφορετικά επιστρέφουν 'X' ή 'U'. Για παράδειγμα '0' **and** 'Z' επιστρέφει '0', επειδή όταν μία είσοδος σε μία πύλη and είναι '0' πάντα προκαλεί την έξοδο να είναι '0', ανεξάρτητα από την άλλη είσοδο.

2.3 Ταξινόμηση Τύπων

Στις προηγούμενες ενότητες εξετάσαμε τους βαθμωτούς τύπους που παρέχονται στη VHDL. Η Εικόνα 2-4 απεικονίζει τις σχέσεις μεταξύ αυτών των τύπων, των προκαθορισμένων βαθμωτών τύπων και των τύπων που θα εξετάσουμε στα επόμενα κεφάλαια.

Βαθμωτοί ονομάζονται όλοι εκείνοι οι τύποι που αποτελούνται από μεμονωμένες τιμές που διατάσσονται με μία σειρά. Οι τύποι ακέραιων και αριθμών κινητής υποδιαστολής διατάσσονται στη γραμμή των αριθμών. Οι φυσικοί τύποι διατάσσονται από τον αριθμό των μονάδων βάσης σε κάθε τιμή. Οι τύποι απαρίθμησης διατάσσονται από τη δήλωσή τους. Διακριτοί τύποι (discrete types) ονομάζονται εκείνοι που αντιπροσωπεύουν διακριτά σύνολα τιμών και περιλαμβάνουν τους τύπους ακέραιων αριθμών και τους τύπους απαρίθμησης. Οι τύποι κινητής υποδιαστολής και οι φυσικοί τύποι δεν είναι διακριτοί, αφού προσεγγίζουν μία αδιάσπαστη αλληλουχία τιμών.

ΕΙΚΟΝΑ 2-4



Μία ταξινόμηση των τύπων της VHDL.

2.3.1 Υποτύποι

Στην Ενότητα 2.2 είδαμε πώς δηλώνουμε έναν τύπο, ο οποίος καθορίζει ένα σύνολο τιμών. Συχνά ένα μοντέλο περιέχει αντικείμενα που πρέπει να πάρουν τιμή μόνο σε ένα περιορισμένο εύρος του πλήρους συνόλου τιμών. Μπορούμε να αναπαραστήσουμε τέτοια αντικείμενα με τη δήλωση ενός *υποτύπου (subtype)*, ο οποίος καθορίζει ένα περιορισμένο σύνολο τιμών από έναν *τύπο βάσης (base type)*. Η συνθήκη που καθορίζει ποιες τιμές ανήκουν στον υποτύπο καλείται *περιορισμός (constraint)*. Η χρησιμοποίηση μιας δήλωσης υποτύπου καθιστά σαφή την πρόθεσή μας περί του ποιες τιμές είναι έγκυρες και έτσι είναι εφικτό να ελεγχθεί ότι δεν χρησιμοποιούνται μη έγκυρες τιμές. Οι απλουστευμένοι συντακτικοί κανόνες για μια δήλωση υποτύπου είναι

subtype_declaration ⇐ **subtype** identifier **is** subtype_indication ;
subtype_indication ⇐
type_mark [**range** simple_expression (**to** I **downto**) simple_expression]

Θα εξετάσουμε πιο προηγμένες μορφές δηλώσεων υποτύπου (subtype indications) στα επόμενα κεφάλαια. Η δήλωση υποτύπου καθορίζει το αναγνωριστικό ως υποτύπο του τύπου βάσης που διευκρινίζεται από την ένδειξη τύπου (type_mark), με τον περιορισμό εύρους να θέτει τα όρια των τιμών του υποτύπου. Ο περιορισμός είναι προαιρετικός, το οποίο σημαίνει ότι είναι δυνατό να υπάρξει ένας υποτύπος που περιλαμβάνει όλες τις τιμές του τύπου βάσης.

ΠΑΡΑΔΕΙΓΜΑ

Εδώ είναι μία δήλωση που ορίζει έναν υποτύπου του τύπου integer:

subtype small_int **is** integer **range** –128 to 127;

Οι τιμές του small_int περιορίζονται ώστε να είναι εντός του εύρους -128 έως 127. Εάν δηλώσουμε κάποιες μεταβλητές:

variable deviation : small_int;
variable adjustment : integer;

μπορούμε να τις χρησιμοποιήσουμε στους υπολογισμούς:

deviation := deviation + adjustment;

Σημειώστε ότι σε αυτήν την περίπτωση, μπορούμε να αναμίξουμε τιμές του υποτύπου και του τύπου βάσης στην πρόσθεση για να παράγουμε μια τιμή τύπου integer, αλλά το αποτέλεσμα πρέπει να είναι εντός του εύρους –128 έως 127 ώστε η ανάθεση να εκτελεστεί επιτυχώς. Εάν δεν είναι, ένα λάθος θα εκδηλωθεί όταν γίνει η ανάθεση στη

μεταβλητή. Όλες οι πράξεις που ισχύουν στον τύπο βάσης μπορούν επίσης να χρησιμοποιηθούν στις τιμές ενός υποτύπου. Οι πράξεις παράγουν τιμές του τύπου βάσης και όχι του υποτύπου. Εντούτοις, η λειτουργία ανάθεσης δεν θα αναθέσει μια τιμή σε μια μεταβλητή του υποτύπου εάν η τιμή δεν ικανοποιεί τον περιορισμό.

Ένα άλλο σημείο που πρέπει να σημειωθεί είναι ότι εάν ένας τύπος βάσης έχει εύρος τιμών μιας κατεύθυνσης (αύξουσας ή φθίνουσας), και ένας υποτύπος καθοριστεί με έναν περιορισμό εύρους της αντίθετης κατεύθυνσης, αυτό που βαρύνει είναι η προδιαγραφή του υποτύπου. Για παράδειγμα, ο προκαθορισμένος τύπος integer είναι μια αύξουσα σειρά. Εάν δηλώσουμε έναν υποτύπο ως

```
subtype bit_index is integer range 31 downto 0;
```

αυτός ο υποτύπος είναι μια φθίνουσα σειρά.

Το πρότυπο της VHDL περιλαμβάνει δύο προκαθορισμένους υποτύπους ακέραιων αριθμών, που ορίζονται ως

```
subtype natural is integer range 0 to highest_integer;
subtype positive is integer range 1 to highest_integer;
```

Όταν η λογική μίας σχεδίασης μαρτυρά ότι ένας αριθμός δεν πρέπει να είναι αρνητικός, είναι καλή πρακτική να χρησιμοποιήσουμε έναν από αυτούς τους υποτύπους παρά τον τύπο βάσης integer. Κατ' αυτόν τον τρόπο, μπορούμε να ανιχνεύσουμε οποιαδήποτε σχεδιαστικά λάθη τα οποία προκαλούν λανθασμένα την παραγωγή αρνητικών αριθμών. Υπάρχει επίσης ένας προκαθορισμένος υποτύπος του φυσικού τύπου time, που ορίζεται ως

```
subtype delay_length is time range 0 fs to highest_time;
```

Αυτός ο υποτύπος πρέπει να χρησιμοποιηθεί οπουδήποτε απαιτείται μια μη αρνητική χρονική καθυστέρηση.

VHDL-87

Ο υποτύπος delay_length δεν είναι προκαθορισμένος στη VHDL-87.

2.3.2 Επεξήγηση τύπου

Μερικές φορές δεν είναι σαφές από τα συμφραζόμενα ποιος είναι ο τύπος μιας συγκεκριμένης τιμής. Στην περίπτωση των υπερφορτωμένων κυριολεκτικών απαρίθμησης, μπορεί να είναι απαραίτητο να διευκρινιστεί ρητά ποιος τύπος εννοείται. Μπορούμε να κάνουμε κάτι τέτοιο χρησιμοποιώντας την *επεξήγηση τύπου (type qualification)*, η οποία αποτελείται από το γράψιμο του ονόματος του τύπου ακολουθούμενο από ένα χαρακτήρα απλού εισαγωγικού (single quote), και κατόπιν μια παράσταση εσωκλειόμενη σε παρενθέσεις. Για παράδειγμα, δοθέντος των τύπων απαρίθμησης

```
type logic_level is (unknown, low, undriven, high);
type system_state is (unknown, ready, busy);
```

μπορούμε να διακρίνουμε μεταξύ των κοινών κυριολεκτικών τιμών γράφοντας

```
logic_level'(unknown),   system_state'(unknown)
```

Η επεξήγηση του τύπου μπορεί επίσης να χρησιμοποιηθεί για να περιορίσει το εύρος μια τιμής σε ένα συγκεκριμένο υποτύπο ενός τύπου βάσης. Για παράδειγμα, εάν ορίσουμε έναν υποτύπο του logic_level

```
subtype valid_level is logic_level range low to high;
```

μπορούμε να διευκρινίσουμε ρητά την τιμή είτε του τύπου είτε του υποτύπου

```
logic_level'(high),   valid_level'(high)
```

Φυσικά, είναι λάθος εάν η παράσταση που επεξηγήθηκε δεν είναι του τύπου ή του υποτύπου που διευκρινίζεται.

2.3.3 Μετατροπή τύπου

Όταν εισαγάγαμε τους αριθμητικούς τελεστές στις προηγούμενες ενότητες, δηλώσαμε ότι οι τελεστέοι πρέπει να είναι του ίδιου τύπου. Αυτό αποκλείει τη μίξη τιμών ακέραιων και αριθμών κινητής υποδιαστολής στις αριθμητικές παραστάσεις. Όπου χρειάζεται να κάνουμε μικτή αριθμητική, μπορούμε να χρησιμοποιήσουμε τις *μετατροπές τύπων (type conversions)* ώστε να μετατρέψουμε τις τιμές μεταξύ ακέραιων αριθμών και αριθμών κινητής υποδιαστολής. Η μορφή μιας μετατροπής τύπου είναι το όνομα του τύπου στον οποίο θέλουμε να μετατρέψουμε, ακολουθούμενο από μια τιμή σε παρενθέσεις. Για παράδειγμα, για να μετατρέψουμε μεταξύ των τύπων integer και real, θα μπορούσαμε να γράψουμε

```
real(123),   integer(3.6)
```

Η μετατροπή ενός ακέραιου αριθμού σε μία τιμή κινητής υποδιαστολής είναι απλά μια αλλαγή στην αναπαράσταση, αν και μπορεί να συμβεί κάποια απώλεια ακρίβειας. Η μετατροπή από μια τιμή κινητής υποδιαστολής σε έναν ακέραιο αριθμό συνεπάγεται τη στρογγυλοποίηση στον κοντινότερο ακέραιο. Οι μετατροπές αριθμητικών τύπων δεν είναι οι μοναδικές μετατροπές που επιτρέπονται. Γενικά, μπορούμε να μετατρέψουμε μεταξύ οποιωνδήποτε

στενά σχετιζόμενων τύπων. Άλλα παραδείγματα στενά σχετιζόμενων τύπων είναι ορισμένοι τύποι πινάκων, που θα δούμε στο Κεφάλαιο 4.

Ένα πράγμα που πρέπει να προσέξουμε είναι η διάκριση μεταξύ της επεξήγησης τύπου και της μετατροπής τύπου. Το πρώτο δηλώνει απλά τον τύπο μιας τιμής, ενώ το τελευταίο αλλάζει την τιμή, ενδεχομένως σε έναν διαφορετικό τύπο. Ένας τρόπος να θυμόμαστε αυτή τη διάκριση είναι να έχουμε κατά νου τη φράση «εισαγωγικά για επεξήγηση» ("*qu*ote for *qu*alification").

2.4 Ιδιότητες των Βαθμωτών Τύπων

Ένας τύπος καθορίζει ένα σύνολο τιμών και ένα σύνολο εφαρμόσιμων πράξεων. Υπάρχει επίσης ένα προκαθορισμένο σύνολο *ιδιοτήτων (attributes)* που χρησιμοποιούνται για να δώσουν πληροφορίες σχετικές με τις τιμές που περιλαμβάνονται στον τύπο. Οι ιδιότητες γράφονται με το όνομα του τύπου ακολουθούμενο από ένα σύμβολο εισαγωγικού (') και το όνομα της ιδιότητας. Η τιμή μιας ιδιότητας μπορεί να χρησιμοποιηθεί σε υπολογισμούς σε ένα μοντέλο. Εξετάζουμε τώρα μερικές από τις ιδιότητες που καθορίζονται για τους τύπους που έχουμε συζητήσει σε αυτό το κεφάλαιο.

Καταρχήν, υπάρχει ένας αριθμός ιδιοτήτων που εφαρμόζονται σε όλους τους βαθμωτούς τύπους και παρέχουν πληροφορίες για το εύρος των τιμών του τύπου. Έστω ότι το T αντιπροσωπεύει οποιοδήποτε βαθμωτό τύπο ή υποτύπο, το x αντιπροσωπεύει μια τιμή αυτού του τύπου και το s αντιπροσωπεύει μια τιμή αλφαριθμητικού (string), οι ιδιότητες είναι

T'left	πρώτη (αριστερότερη) τιμή του T
T'right	τελευταία (δεξιότερη) τιμή του T
T'low	μικρότερη τιμή του T
T'high	μεγαλύτερη τιμή του T
T'ascending	true (αληθές) εάν το T είναι αύξουσα σειρά, διαφορετικά false (ψευδές)
T'image(x)	ένα αλφαριθμητικό που αναπαριστά την τιμή του x
T'value(s)	η τιμή στο T που αναπαριστάται από το s

Το αλφαριθμητικό που παράγεται από την ιδιότητα 'image είναι ένα σωστά διαμορφωμένο κυριολεκτικό σύμφωνα με τους κανόνες που παρουσιάστηκαν στο Κεφάλαιο 1. Τα αλφαριθμητικά που επιτρέπονται στην ιδιότητα 'value πρέπει να ακολουθούν εκείνους τους κανόνες και μπορούν να περιλαμβάνουν αρχικά και τελικά κενά διαστήματα. Αυτές οι δύο ιδιότητες είναι χρήσιμες για την είσοδο και την έξοδο σε ένα μοντέλο, όπως θα δούμε όταν φτάσουμε σε αυτό το θέμα.

ΠΑΡΑΔΕΙΓΜΑ

Για να επεξηγήσουμε τις ιδιότητες που απαριθμούνται παραπάνω, παραθέτουμε μερικές δηλώσεις από προηγούμενα παραδείγματα:

```
type resistance is range 0 to 1E9
    units
        ohm;
        kohm = 1000 ohm;
        Mohm = 1000 kohm;
    end units resistance;
type set_index_range is range 21 downto 11;
type logic_level is (unknown, low, undriven, high);
```

Για αυτούς τους τύπους:

```
resistance'left = 0 ohm
resistance'right = 1E9 ohm
resistance'low = 0 ohm
resistance'high = 1E9 ohm
resistance'ascending = true
resistance'image(2 kohm) = "2000 ohm"
resistance'value("5 Mohm") = 5_000_000 ohm


set_index_range'left = 21
set_index_range'right = 11
set_index_range'low = 11
set_index_range'high = 21
set_index_range'ascending = false
set_index_range'image(14) = "14"
set_index_range'value("20") = 20
```

```
logic_level'left = unknown
logic_level'right = high
logic_level'low = unknown
logic_level'high = high
logic_level'ascending = true
logic_level'image(undriven) = "undriven"
logic_level'value("Low") = low
```

Εν συνεχεία, υπάρχουν ιδιότητες που εφαρμόζονται αποκλειστικά στους διακριτούς και στους φυσικούς τύπους. Για οποιοδήποτε τέτοιο τύπο T, μια τιμή x αυτού του τύπου και έναν ακέραιο n, οι ιδιότητες είναι

T'pos(x)	θέση του αριθμού x στο T
T'val(n)	τιμή του T στη θέση n
T'succ(x)	τιμή του T σε θέση κατά ένα μεγαλύτερη από αυτή του x
T'pred(x)	τιμή του T σε θέση κατά ένα μικρότερη από αυτή του x
T'leftof(x)	τιμή του T σε μια θέση αριστερά από αυτή του x
T'rightof(x)	τιμή του T σε μια θέση δεξιά από αυτή του x

Για τους τύπους απαρίθμησης, οι αριθμοί θέσης αρχίζουν από το μηδέν για το πρώτο στοιχείο που απαριθμείται και αυξάνεται κατά ένα για κάθε στοιχείο στα δεξιά. Έτσι, για τον τύπο logic_level που παρουσιάστηκε παραπάνω, μερικές τιμές ιδιοτήτων είναι

```
logic_level'pos(unknown) = 0
logic_level'val(3) = high
logic_level'succ(unknown) = low
logic_level'pred(undriven) = low
```

Για τους ακέραιους τύπους, ο αριθμός θέσης είναι ο ίδιος με την τιμή του ακέραιου, αλλά ο τύπος του αριθμού θέσης είναι ένας ειδικός ανώνυμος τύπος που καλείται *καθολικός ακέραιος (universal integer)*. Αυτός είναι ο ίδιος τύπος με αυτόν των ακέραιων κυριολεκτικών και, όπου είναι απαραίτητο, μετατρέπεται αυτονόητα σε οποιοδήποτε άλλο δηλωμένο ακέραιο τύπο. Για τους φυσικούς τύπους, ο αριθμός θέσης είναι ο ακέραιος αριθμός των μονάδων βάσης στη φυσική τιμή. Για παράδειγμα:

```
time'pos(4 ns) = 4_000_000
```

αφού η μονάδα βάσης είναι το fs.

ΠΑΡΑΔΕΙΓΜΑ

Μπορούμε να χρησιμοποιήσουμε σε συνδυασμό τις ιδιότητες ‘pos και ‘val για να εκτελέσουμε αριθμητική μικτής-διάστασης με τους φυσικούς τύπους, παράγοντας ένα αποτέλεσμα της σωστής διάστασης. Υποθέστε ότι ορίζουμε φυσικούς τύπους για να αναπαραστήσουμε το μήκος και το εμβαδόν, ως εξής:

```
type length is range integer'low to integer'high
    units
        mm;
    end units length;
type area is range integer'low to integer'high
    units
        square_mm;
end units area;
```

και μεταβλητές αυτού του τύπου:

```
variable L1, L2 : length;
variable A : area;
```

Οι περιορισμοί στον πολλαπλασιασμό των τιμών των φυσικών τύπων μας αποτρέπουν από το να γράψουμε κάτι παρόμοιο με

```
A := L1 * L2;   -- αυτό είναι λανθασμένο
```

Για να επιτύχουμε σωστό αποτέλεσμα, μπορούμε να μετατρέψουμε τις τιμές μήκους σε αφηρημένους ακέραιους αριθμούς χρησιμοποιώντας την ιδιότητα ’pos, και στη συνέχεια να μετατρέψουμε το αποτέλεσμα του πολλαπλασιασμού σε μία τιμή εμβαδού χρησιμοποιώντας την ιδιότητα ‘val, ως εξής:

```
A := area'val( length'pos(L1) * length'pos(L2) );
```

Σημειώστε ότι σε αυτό το παράδειγμα, δεν πρέπει να συμπεριλάβουμε ένα συντελεστή κλίμακας στον πολλαπλασιασμό, αφού η μονάδα βάσης του τύπου area είναι το τετράγωνο της μονάδας βάσης του τύπου length.

Για αύξουσες σειρές, τα T'succ(x) και T'rightof(x) παράγουν την ίδια τιμή, και τα T'pred(x) και T'leftof(x) παράγουν την ίδια τιμή. Για φθίνουσες σειρές, τα T'pred(x) και T'rightof(x) παράγουν την ίδια τιμή, και τα T'succ(x) και T'leftof(x) παράγουν την ίδια τιμή. Για όλες τις σειρές, τα T'succ(T'high), T'pred(T'low), T'rightof(T'right) και T'leftof(T'left) προκαλούν να συμβεί ένα λάθος.

Η τελευταία ιδιότητα που εισάγουμε εδώ είναι η T'base. Για οποιοδήποτε υποτύπο T, αυτή η ιδιότητα παράγει τον τύπο βάσης του T. Το μοναδικό πλαίσιο όπου μπορεί να χρησιμοποιηθεί αυτή η ιδιότητα είναι σαν πρόθεμα μίας άλλης ιδιότητας. Για παράδειγμα, εάν έχουμε τις δηλώσεις

```
type opcode is (nop, load, store, add, subtract, negate, branch, halt);
subtype arith_op is opcode range add to negate;
```

τότε

```
arith_op'base'left = nop
arith_op'base'succ(negate) = branch
```

VHDL-87

Οι ιδιότητες 'ascending, 'image και 'value δεν παρέχονται στη VHDL-87.

2.5 Παραστάσεις και Τελεστές

Στην Ενότητα 2.1 δείξαμε πώς η τιμή που προκύπτει ως αποτέλεσμα της αξιολόγησης μιας παράστασης μπορεί να ανατεθεί σε μια μεταβλητή. Σε αυτήν την ενότητα, συνοψίζουμε τους κανόνες που διέπουν τις παραστάσεις. Μπορούμε να σκεφτούμε μια παράσταση σαν μια φόρμουλα που προσδιορίζει πώς να υπολογίσουμε μια τιμή. Υπό αυτήν τη μορφή, αποτελείται από κύριες τιμές που συνδυάζονται με τελεστές. Οι κύριες τιμές που μπορούν να χρησιμοποιηθούν στις παραστάσεις περιλαμβάνουν

- κυριολεκτικές τιμές,
- αναγνωριστικά που αντιπροσωπεύουν αντικείμενα δεδομένων (σταθερές, μεταβλητές και τα λοιπά),
- ιδιότητες που παράγουν τιμές,
- επεξηγημένες παραστάσεις,
- παραστάσεις με μετατροπή τύπων και
- παραστάσεις σε παρενθέσεις.

Έχουμε δει ανάλογα παραδείγματα σε αυτό το κεφάλαιο και στο Κεφάλαιο 1. Για αναφορά, όλοι οι τελεστές και οι τύποι που μπορούν να εφαρμοστούν συνοψίζονται στην Εικόνα 2-5. Θα συζητήσουμε τους τελεστές πινάκων στο Κεφάλαιο 4.

Οι τελεστές σε αυτόν τον πίνακα ομαδοποιούνται κατά προτεραιότητα, με τους "**", **abs** και **not** να έχουν την υψηλότερη προτεραιότητα και τους λογικούς τελεστές την χαμηλότερη. Αυτό σημαίνει ότι εάν μια παράσταση περιέχει ένα συνδυασμό τελεστών, εκείνοι με την υψηλότερη προτεραιότητα εφαρμόζονται πρώτοι. Οι παρενθέσεις μπορούν να χρησιμοποιηθούν για να αλλάξουν τη σειρά αξιολόγησης, ή για σαφήνεια.

VHDL-87

Οι τελεστές ολίσθησης (**sll**, **srl**, **sla**, **sra**, **rol** και **ror**) και ο τελεστής **xnor** δεν παρέχονται στη VHDL-87.

ΕΙΚΟΝΑ 2-5

Τελεστής	*Πράξη*	*Τύπος αριστερού τελεστέου*	*Τύπος δεξιού τελεστέου*	*Τύπος αποτελέσματος*
**	ύψωση σε δύναμη	ακέραιος ή κινητής υποδιαστολής	ακέραιος	ίδιος με αρ. τελεστέο
abs	απόλυτη τιμή		αριθμητικός	ίδιος με τελεστέο
not	άρνηση		bit, boolean ή μονοδιάστατος πίνακας από bit ή boolean	ίδιος με τελεστέο
*	Πολλαπλασι-ασμός	ακέραιος ή κινητής υποδιαστολής	ίδιος με αρ. τελεστέο	ίδιος με τελεστέους
		φυσικός	integer ή real	ίδιος με αρ. τελεστέο
		integer ή real	φυσικός	ίδιος με δεξί τελεστέο
/	διαίρεση	ακέραιος ή κινητής υποδιαστολής	ίδιος με αρ. τελεστέο	ίδιος με τελεστέους

		φυσικός	integer ή real	ίδιος με αρ. τελεστέο
		φυσικός	ίδιος με αρ. τελεστέο	καθολικός ακέραιος
mod	modulo	ακέραιος	ίδιος με αρ. τελεστέο	ίδιος με τελεστέους
rem	υπόλοιπο διαίρεσης	ακέραιος	ίδιος με αρ. τελεστέο	ίδιος με τελεστέους
+	ταυτότητα		αριθμητικός	ίδιος με τελεστέο
-	αλλαγή προσήμου		αριθμητικός	ίδιος με τελεστέο
+	πρόσθεση	αριθμητικός	ίδιος με αρ. τελεστέο	ίδιος με τελεστέους
-	αφαίρεση	αριθμητικός	ίδιος με αρ. τελεστέο	ίδιος με τελεστέους
&	συνένωση	μονοδιάστατος πίνακας	ίδιος με αρ. τελεστέο	ίδιος με τελεστέους
		μονοδιάστατος πίνακας	τύπος στοιχείων του αρ. τελεστέου	ίδιος με αρ. τελεστέο
		τύπος στοιχείων του αρ. τελεστέου	μονοδιάστατος πίνακας	ίδιος με δεξί τελεστέο
		τύπος στοιχείων του αποτελέσματος	τύπος στοιχείων του αποτελέσματος	μονοδιάστατος πίνακας
sll	λογική αριστερή ολίσθηση	μονοδιάστατος πίνακας από bit ή boolean	ακέραιος	ίδιος με αρ. τελεστέο
srl	λογική δεξιά ολίσθηση			
sla	αριθμητική αριστερή ολίσθηση			
sra	αριθμητική δεξιά ολίσθηση			
rol	αριστερή περιστροφή			
ror	δεξιά περιστροφή			
=	ισότητα	οτιδήποτε εκτός από αρχείο ή προστατευμένο τύπο	ίδιος με αρ. τελεστέο	boolean
/=	ανισότητα			
<	λιγότερο-από	βαθμωτός ή μονοδιάστατος πίνακας οποιουδήποτε διακριτού τύπου	ίδιος με αρ. τελεστέο	boolean
<=	λιγότερο-από-ή-ίσο			
>	μεγαλύτερο-από			
>=	μεγαλύτερο-από-ή-ίσο			
and	λογικό και	bit, boolean ή μονοδιάστατος πίνακας από bit ή boolean	ίδιος με αρ. τελεστέο	ίδιος με τελεστέους
or	λογικό ή			
nand	λογικό όχι-και			
nor	λογικό ούτε			
xor	αποκλειστικό ή			
xnor	αποκλειστικό ούτε			

Οι τελεστές της VHDL με σειρά προτεραιότητας, από τον περισσότερο δεσμευτικό στο λιγότερο δεσμευτικό.

Κεφάλαιο 3: Ακολουθιακές προτάσεις

Στο προηγούμενο κεφάλαιο είδαμε πώς να αναπαριστούμε την εσωτερική κατάσταση των μοντέλων χρησιμοποιώντας τύπους δεδομένων της VHDL. Σε αυτό το κεφάλαιο εξετάζουμε το πώς μπορούμε να χειριστούμε αυτά τα δεδομένα μέσα στις διεργασίες. Αυτό γίνεται χρησιμοποιώντας *ακολουθιακές προτάσεις (sequential statements)*, οι οποίες αποκαλούνται έτσι επειδή εκτελούνται ακολουθιακά. Έχουμε δει ήδη μία από τις βασικές ακολουθιακές προτάσεις, την πρόταση ανάθεσης μεταβλητής, όταν εξετάσαμε τους τύπους δεδομένων και τα αντικείμενα. Οι προτάσεις που εξετάζουμε σε αυτό το κεφάλαιο ασχολούνται με ενέργειες ελέγχου μέσα σε ένα μοντέλο, ως εκ τούτου καλούνται συχνά *δομές ελέγχου (control structures)*. Επιτρέπουν την επιλογή μεταξύ εναλλακτικών ενεργειών, καθώς επίσης και την επανάληψη των ενεργειών.

3.1 Προτάσεις If

Σε πολλά μοντέλα, η συμπεριφορά εξαρτάται από ένα σύνολο συνθηκών που μπορεί να είναι ή να μην είναι αληθείς κατά τη διάρκεια της ροής της προσομοίωσης. Μπορούμε να χρησιμοποιήσουμε μία *πρόταση if (if statement)* για να εκφράσουμε αυτήν τη συμπεριφορά. Ο συντακτικός κανόνας για μία πρόταση if είναι

```
if_statement ⇐
    [ if_label : ]
    if boolean_expression then
        { sequential_statement }
    { elsif boolean_expression then
        { sequential_statement } }
    [ else
        { sequential_statement } ]
    end if [ if_label ] ;
```

Με μια πρώτη ματιά, αυτό μπορεί να φαίνεται κάπως περίπλοκο, έτσι αρχίζουμε με κάποια απλά παραδείγματα και αυξάνουμε τη δυσκολία μέχρι τα παραδείγματα να παρουσιάσουν τη γενική περίπτωση. Η ετικέτα μπορεί να χρησιμοποιηθεί για να προσδιορίσει μια πρόταση if. Ένα απλό παράδειγμα μιας πρότασης if είναι

```
if en = '1' then
    stored_value := data_in;
end if;
```

Η λογική (Boolean) παράσταση μετά από τη λέξη-κλειδί **if** είναι η συνθήκη που χρησιμοποιείται για να ελέγξει εάν η πρόταση μετά από τη λέξη-κλειδί **then** εκτελείται ή όχι. Εάν η συνθήκη αξιολογηθεί ως αληθής, η πρόταση εκτελείται. Σε αυτό το παράδειγμα, εάν η τιμή του αντικειμένου en είναι '1', η ανάθεση γίνεται, διαφορετικά παραλείπεται. Μπορούμε επίσης να καθορίσουμε τις ενέργειες που θα εκτελεστούν εάν η συνθήκη είναι ψευδής. Για παράδειγμα:

```
if sel = 0 then
    result <= input_0; -- εκτελείται εάν sel = 0
else
    result <= input_1; -- εκτελείται εάν sel /= 0
end if;
```

Εδώ, όπως υποδηλώνουν τα σχόλια, η πρώτη πρόταση ανάθεσης σήματος εκτελείται εάν η συνθήκη είναι αληθής, και η δεύτερη πρόταση ανάθεσης σήματος εκτελείται εάν η συνθήκη είναι ψευδής.

Σε πολλά μοντέλα, μπορεί να χρειαστεί να ελέγξουμε έναν αριθμό από διαφορετικές συνθήκες και να εκτελέσουμε μια διαφορετική ακολουθία προτάσεων για κάθε περίπτωση. Μπορούμε να συντάξουμε μια πιο αναπτυγμένη μορφή της πρότασης if ώστε να κάνουμε κάτι τέτοιο, για παράδειγμα:

```
if mode = immediate then
    operand := immed_operand;
elsif opcode = load or opcode = add or opcode = subtract then
    operand := memory_operand;
else
    operand := address_operand;
end if;
```

Σε αυτό το παράδειγμα, αξιολογείται η πρώτη συνθήκη, και εάν είναι αληθής, εκτελείται η πρόταση μετά από την πρώτη λέξη-κλειδί **then**. Εάν η πρώτη συνθήκη είναι ψευδής, αξιολογείται η δεύτερη συνθήκη, και εάν είναι αληθής, εκτελείται η πρόταση μετά από τη δεύτερη λέξη-κλειδί **then**. Εάν η δεύτερη συνθήκη είναι ψευδής, εκτελείται η πρόταση μετά από την λέξη-κλειδί **else.**

Γενικά, μπορούμε να συντάξουμε μια πρόταση if με οποιοδήποτε αριθμό φράσεων **elsif** (ακόμα και καμίας), και μπορούμε να συμπεριλάβουμε ή να παραλείψουμε τη φράση **else**. Η εκτέλεση της πρότασης if αρχίζει με την αξιολόγηση της πρώτης συνθήκης. Εάν είναι ψευδής, οι διαδοχικές συνθήκες αξιολογούνται, με τη σειρά, έως ότου βρεθεί μία που να είναι αληθής, οπότε σ'αυτή την περίπτωση εκτελούνται οι αντίστοιχες προτάσεις. Εάν καμία από τις συνθήκες δεν είναι αληθής, και έχουμε συμπεριλάβει μια φράση **else**, εκτελούνται οι προτάσεις μετά από τη λέξη-κλειδί **else**.

Δεν περιοριζόμαστε σε μόνο μια πρόταση σε κάθε μέρος της πρότασης if. Αυτό διευκρινίζεται από την παρακάτω πρόταση if:

```
if opcode = halt_opcode then
    PC := effective_address;
    executing := false;
    halt_indicator <= true;
end if;
```

Εάν η συνθήκη είναι αληθής, εκτελούνται και οι τρεις προτάσεις, η μια μετά από την άλλη. Από την άλλη μεριά, εάν η συνθήκη είναι ψευδής, δεν εκτελείται καμία από τις προτάσεις. Επιπλέον, κάθε πρόταση που περιέχεται σε μια πρόταση if μπορεί να είναι οποιαδήποτε ακολουθιακή πρόταση. Αυτό σημαίνει ότι μπορούμε να ενθέσουμε - τη μια μέσα στην άλλη – προτάσεις if, για παράδειγμα:

```
if phase = wash then
    if cycle_select = delicate_cycle then
        agitator_speed <= slow;
    else
        agitator_speed <= fast;
    end if;
    agitator_on <= true;
end if;
```

Σε αυτό το παράδειγμα, αξιολογείται αρχικά η συνθήκη phase = wash, και εάν είναι αληθής, εκτελούνται η ένθετη πρόταση if και η ακόλουθη πρόταση ανάθεσης σήματος. Κατά συνέπεια η ανάθεση agitator_speed <= slow εκτελείται μόνο εάν και οι δύο συνθήκες αξιολογηθούν ως αληθής, και η ανάθεση agitator_speed <= fast εκτελείται μόνο εάν η πρώτη συνθήκη είναι αληθής και η δεύτερη συνθήκη είναι ψευδής.

ΠΑΡΑΔΕΙΓΜΑ

Ας αναπτύξουμε ένα μοντέλο συμπεριφοράς για έναν απλό θερμοστάτη μίας θερμάστρας. Η συσκευή μπορεί να μοντελοποιηθεί ως μια οντότητα με δύο εισόδους ακέραιων αριθμών, μια που καθορίζει την επιθυμητή θερμοκρασία και μια άλλη που είναι συνδεδεμένη με ένα θερμόμετρο, και μια έξοδο τύπου Boolean η οποία ανοίγει και κλείνει μια θερμάστρα. Η θερμοστάτης ανοίγει τη θερμάστρα εάν η θερμοκρασία που μετράει το θερμόμετρο πέσει κάτω από δύο βαθμούς λιγότερο από την επιθυμητή θερμοκρασία, και κλείνει τη θερμάστρα εάν η θερμοκρασία ανέβει πάνω από δύο βαθμούς περισσότερο από την επιθυμητή θερμοκρασία. Η Εικόνα 3-1 δείχνει τα σώματα οντότητας και αρχιτεκτονικής για το θερμοστάτη. Η δήλωση της οντότητας καθορίζει τις θύρες εισόδου και εξόδου.

ΕΙΚΟΝΑ 3-1

```
entity thermostat is
    port ( desired_temp, actual_temp : in integer;
           heater_on : out boolean );
end entity thermostat;
```

```
architecture example of thermostat is
begin
    controller : process (desired_temp, actual_temp) is
    begin
        if actual_temp < desired_temp – 2 then
            heater_on <= true;
        elsif actual_temp > desired_temp + 2 then
            heater_on <= false;
        end if;
    end process controller;
end architecture example;
```

Μια οντότητα και ένα σώμα αρχιτεκτονικής για ένα θερμοστάτη θερμάστρας.

Δεδομένου ότι είναι ένα μοντέλο συμπεριφοράς, το σώμα αρχιτεκτονικής περιέχει μόνο μια πρόταση διεργασίας που υλοποιεί την απαιτούμενη συμπεριφορά. Η πρόταση διεργασίας περιλαμβάνει μια *λίστα ευαισθησίας (sensitivity list)* μετά από τη λέξη-κλειδί **process**. Αυτή είναι μια λίστα σημάτων στα οποία η διεργασία είναι ευαίσθητη. Όταν οποιοδήποτε από αυτά τα σήματα αλλάζει τιμή, η διεργασία επαναρχίζει και εκτελεί τις ακολουθιακές προτάσεις. Αφότου έχει εκτελέσει την τελευταία πρόταση, η διεργασία αναστέλλει πάλι την εκτέλεσή της. Σε αυτό το παράδειγμα, η διεργασία είναι ευαίσθητη στις αλλαγές σε οποιαδήποτε από τις θύρες εισόδου. Κατά συνέπεια, εάν ρυθμίσουμε την επιθυμητή θερμοκρασία, ή εάν η θερμοκρασία που μετράει το θερμόμετρο μεταβληθεί, η διεργασία επαναρχίζει. Το σώμα της διεργασίας περιέχει μια πρόταση if που συγκρίνει την πραγματική θερμοκρασία με την επιθυμητή θερμοκρασία. Εάν η πραγματική θερμοκρασία είναι αρκετά χαμηλή, η διεργασία εκτελεί την πρώτη ανάθεση σήματος για να ανοίξει τη θερμάστρα. Εάν η πραγματική θερμοκρασία είναι αρκετά υψηλή, η διεργασία εκτελεί τη δεύτερη ανάθεση σήματος για να κλείσει τη θερμάστρα. Εάν η πραγματική θερμοκρασία είναι μέσα στο επιθυμητό εύρος τιμών, η κατάσταση της θερμάστρας δεν αλλάζει, δεδομένου ότι δεν υπάρχει καμία φράση **else** στην πρόταση if.

VHDL-87

Οι προτάσεις if δεν επιτρέπεται να έχουν ετικέτες στη VHDL-87.

3.2 Προτάσεις Case

Εάν έχουμε ένα μοντέλο του οποίου η συμπεριφορά εξαρτάται από την τιμή μιας απλής παράστασης, μπορούμε να χρησιμοποιήσουμε μια *πρόταση case (case statement)*. Οι συντακτικοί κανόνες είναι οι εξής:

case_statement ⇐

[case_label :]

case expression **is**

(**when** choices => { sequential_statement })

{ … }

end case [case_label] ;

choices _ (simple_expression I discrete_range I **others**) { | … }

Η ετικέτα μπορεί να χρησιμοποιηθεί για να προσδιορίσει την πρόταση case. Αρχίζουμε με μερικά απλά παραδείγματα προτάσεων case και εν συνεχεία αυξάνουμε την πολυπλοκότητά τους. Καταρχήν, υποθέστε ότι μοντελοποιούμε μια αριθμητική/λογική μονάδα, που έχει μια είσοδο ελέγχου, func, η οποία έχει δηλωθεί να είναι τύπου απαρίθμησης:

type alu_func **is** (pass1, pass2, add, subtract);

Θα μπορούσαμε να περιγράψουμε τη συμπεριφορά χρησιμοποιώντας μια πρόταση case:

```
case func is
    when pass1 =>
        result := operand1;
    when pass2 =>
        result := operand2;
    when add =>
        result := operand1 + operand2;
    when subtract =>
        result := operand1 – operand2;
end case;
```

Στην κορυφή αυτής της πρότασης case είναι η *παράσταση επιλογής (selector expression)*, μεταξύ των λέξεων-κλειδιών **case** και **is**. Σε αυτό το παράδειγμα είναι μια απλή παράσταση που αποτελείται απλά από μια κύρια τιμή. Η τιμή αυτής της παράστασης χρησιμοποιείται για να επιλέξει ποιες προτάσεις θα εκτελέσει. Το σώμα της πρότασης case αποτελείται από μια σειρά *εναλλακτικών λύσεων (alternatives)*. Κάθε εναλλακτική λύση αρχίζει με τη λέξη-κλειδί **when** και ακολουθείται από μια ή περισσότερες *επιλογές (choises)* και μια ακολουθία προτάσεων. Οι επιλογές είναι τιμές που συγκρίνονται με την τιμή της παράστασης επιλογής. Πρέπει να υπάρχει ακριβώς μια επιλογή για κάθε πιθανή τιμή. Η πρόταση case βρίσκει την εναλλακτική λύση της οποίας η τιμή επιλογής είναι ίση με την τιμή της παράστασης επιλογής και εκτελεί τις προτάσεις σε εκείνη την εναλλακτική λύση. Σε αυτό το παράδειγμα, οι επιλογές είναι όλες απλές παραστάσεις του τύπου alu_func. Εάν η τιμή του func είναι pass1, εκτελείται η πρόταση result : = operand1; εάν η τιμή είναι pass2, εκτελείται η πρόταση result : = operand2, και τα λοιπά.

Μια πρόταση case εμφανίζει κάποια ομοιότητα με μια πρόταση if δεδομένου ότι και οι δύο επιλέγουν μεταξύ εναλλακτικών ομάδων από ακολουθιακές προτάσεις. Η διαφορά έγκειται στο πώς επιλέγονται οι προτάσεις που

πρόκειται να εκτελεστούν. Είδαμε στην προηγούμενη ενότητα ότι μια πρόταση if αξιολογεί διαδοχικές λογικές (Boolean) παραστάσεις με τη σειρά μέχρι να βρεθεί μία που να είναι αληθής. Έπειτα εκτελείται η ομάδα των προτάσεων που αντιστοιχεί σε εκείνη τη συνθήκη. Από την άλλη μεριά, μια πρόταση case αξιολογεί μια απλή παράσταση επιλογής για να εξάγει μια τιμή επιλογής. Αυτή η τιμή συγκρίνεται έπειτα με τις τιμές επιλογής στις εναλλακτικές λύσεις της πρότασης case για να καθορίσει ποια πρόταση θα εκτελέσει. Μια πρόταση if παρέχει έναν πιο γενικό μηχανισμό για την επιλογή μεταξύ εναλλακτικών λύσεων, δεδομένου ότι οι συνθήκες μπορούν να είναι αυθαίρετες σύνθετες λογικές παραστάσεις. Εντούτοις, οι προτάσεις case είναι ένας σημαντικός και χρήσιμος μηχανισμός μοντελοποίησης, όπως παρουσιάζουν τα παραδείγματα σε αυτήν την ενότητα.

Η παράσταση επιλογής μιας πρότασης case πρέπει να οδηγήσει σε μια τιμή ενός διακριτού τύπου, ή ενός μονοδιάστατου πίνακα με στοιχεία χαρακτήρα, όπως ένα αλφαριθμητικό (character string) ή μια ψηφιοσειρά (bit string) (βλ. Κεφάλαιο 4). Κατά συνέπεια, μπορούμε να έχουμε μια πρόταση case που επιλέγει μια εναλλακτική λύση βασισμένη σε μια ακέραια τιμή. Εάν υποθέσουμε ότι τα index_mode και instruction_register δηλώνονται ως εξής

```
subtype index_mode is integer range 0 to 3;
variable instruction_register : integer range 0 to 2**16 – 1;
```

στη συνέχεια μπορούμε να γράψουμε μια πρόταση case που χρησιμοποιεί μια τιμή αυτού του τύπου:

```
case index_mode'((instruction_register / 2**12) rem 2**2) is
    when 0 =>
        index_value := 0;
    when 1 =>
        index_value := accumulator_A;
    when 2 =>
        index_value := accumulator_B;
    when 3 =>
        index_value := index_register;
end case;
```

Παρατηρήστε ότι σε αυτό το παράδειγμα, χρησιμοποιούμε μια επεξηγημένη παράσταση στην έκφραση επιλογής. Εάν το είχαμε παραλείψει αυτό, το αποτέλεσμα της παράστασης θα ήταν τύπου integer, και θα έπρεπε να συμπεριλάβουμε εναλλακτικές λύσεις ώστε να καλύψουμε όλες τις πιθανές ακέραιες τιμές. Η επεξήγηση τύπου αποφεύγει αυτήν την απαίτηση περιορίζοντας τις πιθανές τιμές της παράστασης.

Ένας άλλος κανόνας που πρέπει να θυμόμαστε είναι ότι ο τύπος κάθε επιλογής πρέπει να είναι ίδιος με τον τύπο που προκύπτει ως αποτέλεσμα της παράστασης επιλογής. Κατά συνέπεια στο παραπάνω παράδειγμα, δεν είναι νόμιμο να συμπεριλάβουμε μια εναλλακτική λύση όπως

```
when 'a' => … -- μη νόμιμο!
```

αφού η επιλογή δεν μπορεί να είναι ένας ακέραιος αριθμός. Μια τέτοια επιλογή δεν έχει νόημα, αφού δεν μπορεί ποτέ να ταιριάξει με μια τιμή του τύπου integer.

Μπορούμε να συμπεριλάβουμε περισσότερες από μια επιλογές σε κάθε εναλλακτική λύση γράφοντας τις επιλογές και χωρίζοντάς τις με το σύμβολο "|". Για παράδειγμα, εάν ο τύπος opcodes δηλώνεται ως εξής

```
type opcodes is
    (nop, add, subtract, load, store, jump, jumpsub, branch, halt);
```

θα μπορούσαμε να γράψουμε μια εναλλακτική λύση που περιλαμβάνει τρεις από αυτές τις τιμές ως επιλογές:

```
when load | add | subtract =>
    operand := memory_operand;
```

Εάν έχουμε έναν αριθμό από εναλλακτικές λύσεις σε μια πρόταση case και θέλουμε να συμπεριλάβουμε μια εναλλακτική λύση που να χειρίζεται όλες τις πιθανές τιμές της παράστασης επιλογής που δεν αναφέρονται στις προηγούμενες εναλλακτικές λύσεις, μπορούμε να χρησιμοποιήσουμε την ειδική επιλογή **others**. Για παράδειγμα, εάν η μεταβλητή opcode είναι μια μεταβλητή του τύπου opcodes, που δηλώθηκε παραπάνω, μπορούμε να γράψουμε

```
case opcode is
    when load | add | subtract =>
        operand := memory_operand;
    when store | jump | jumpsub | branch =>
        operand := address_operand;
    when others =>
        operand := 0;
end case;
```

Σε αυτό το παράδειγμα, εάν η τιμή του opcode δεν είναι καμία από τις επιλογές που απαριθμούνται στην πρώτη και στη δεύτερη εναλλακτική λύση, επιλέγεται η τελευταία εναλλακτική λύση. Μπορεί να υπάρχει μόνο μια εναλλακτική λύση που χρησιμοποιεί την επιλογή **others**, και εάν όντως συμπεριλαμβάνεται, πρέπει να είναι η τελευταία

εναλλακτική λύση στην πρόταση case. Μια εναλλακτική λύση που περιλαμβάνει την επιλογή **others** μπορεί να μην περιλαμβάνει οποιαδήποτε άλλη επιλογή. Σημειώστε ότι, εάν όλες οι πιθανές τιμές της παράστασης επιλογής καλύπτονται από τις προηγούμενες επιλογές, μπορούμε ακόμα και τότε να συμπεριλάβουμε την επιλογή **others**, η οποία όμως δεν πρόκειται ποτέ να επιλεγεί.

Η υπόλοιπη μορφή της επιλογής που δεν έχουμε αναφέρει ακόμα είναι ένα *διακριτό εύρος (discrete range)*, που καθορίζεται από αυτούς τους απλουστευμένους συντακτικούς κανόνες:

discrete_range ⇐

discrete_subtype_indication

I simple_expression (**to** I **downto**) simple_expression

subtype_indication ⇐

type_mark

[**range** simple_expression (**to** I **downto**) simple_expression]

Αυτές οι μορφές μας επιτρέπουν να καθορίσουμε ένα εύρος τιμών σε μια εναλλακτική λύση της πρότασης case. Εάν η τιμή της παράστασης επιλογής ταιριάζει με οποιαδήποτε από τις τιμές του εύρους, εκτελούνται οι προτάσεις στην εναλλακτική λύση. Ο απλούστερος τρόπος να καθορίσουμε ένα διακριτό εύρος είναι απλά να γράψουμε το αριστερό και δεξιό όριο του εύρους, χωρίζοντάς τα με μια λέξη-κλειδί που δηλώνει την κατεύθυνση. Για παράδειγμα, η παραπάνω πρόταση case θα μπορούσε να ξαναγραφεί ως εξής

```
case opcode is
    when add to load =>
        operand := memory_operand;
    when branch downto store =>
        operand := address_operand;
    when others =>
        operand := 0;
end case;
```

Ένας άλλος τρόπος για τον καθορισμό ενός διακριτού εύρους είναι να χρησιμοποιηθεί το όνομα ενός διακριτού τύπου, και ενδεχομένως ένας περιορισμός εύρους που να περιορίζει τις τιμές σε ένα υποσύνολο του τύπου. Για παράδειγμα, εάν δηλώσουμε έναν υποτύπο του τύπου opcodes ως εξής

```
subtype control_transfer_opcodes is opcodes range jump to branch;
```

μπορούμε να ξαναγράψουμε τη δεύτερη εναλλακτική λύση ως εξής

```
when control_transfer_opcodes | store =>
    operand := address_operand;
```

Σημειώστε ότι μπορούμε να χρησιμοποιήσουμε ένα διακριτό εύρος ως επιλογή μόνο εάν η έκφραση επιλογής είναι διακριτού τύπου. Δεν μπορούμε να χρησιμοποιήσουμε ένα διακριτό εύρος εάν η έκφραση επιλογής είναι τύπου πίνακα, όπως ένας τύπος διανύσματος bit (bit vector). Εάν καθορίσουμε ένα εύρος γράφοντας τα όρια και την κατεύθυνση, η κατεύθυνση δεν έχει καμία σημασία εκτός από το να προσδιορίσει τα περιεχόμενα του εύρους.

Ένα σημαντικό σημείο που πρέπει να υπογραμμίσουμε για τις επιλογές σε μια πρόταση case είναι ότι όλες πρέπει να γραφτούν χρησιμοποιώντας *τοπικά στατικές (locally static)* τιμές. Αυτό σημαίνει ότι οι τιμές των επιλογών πρέπει να καθοριστούν κατά τη διάρκεια της φάσης ανάλυσης της σχεδίασης. Όλα τα παραπάνω παραδείγματα ικανοποιούν αυτήν την απαίτηση. Για να δώσουμε ένα παράδειγμα μιας πρότασης case που δεν ικανοποιεί αυτήν την απαίτηση, ας υποθέσουμε ότι έχουμε μια ακέραια μεταβλητή N, που δηλώνεται ως εξής

```
variable N : integer := 1;
```

Εάν γράφαμε την πρόταση case

```
case expression is -- παράδειγμα μιας μη-νόμιμης πρότασης case
    when N | N+1 => …
    when N+2 to N+5 => …
    when others => …
end case;
```

οι τιμές των επιλογών εξαρτώνται από την τιμή της μεταβλητής N. Δεδομένου ότι αυτή θα μπορούσε να αλλάξει κατά τη διάρκεια της εκτέλεσης, αυτές οι επιλογές δεν είναι τοπικά στατικές. Ως εκ τούτου η πρόταση case όπως γράφτηκε δεν είναι νόμιμη. Από την άλλη μεριά, εάν είχαμε δηλώσει το C να είναι ένας σταθερός ακέραιος αριθμός, για παράδειγμα με τη δήλωση

```
constant C : integer := 1;
```

κατόπιν θα μπορούσαμε νόμιμα να γράψουμε την πρόταση case

```
case expression is
    when C | C+1 => …
    when C+2 to C+5 => …
    when others => …
end case;
```

Αυτό είναι νόμιμο, δεδομένου ότι μπορούμε να καθορίσουμε, με την ανάλυση του μοντέλου, ότι η πρώτη εναλλακτική λύση περιλαμβάνει τις επιλογές 1 και 2, η δεύτερη περιλαμβάνει τους αριθμούς μεταξύ 3 και 6 και η τρίτη καλύπτει όλες τις άλλες πιθανές τιμές της παράστασης.

Όλα τα προηγούμενα παραδείγματα παρουσιάζουν μόνο μια πρόταση σε κάθε εναλλακτική λύση. Όπως με την πρόταση if, μπορούμε να γράψουμε έναν αυθαίρετο αριθμό ακολουθιακών προτάσεων οποιουδήποτε είδους σε κάθε εναλλακτική λύση. Αυτό περιλαμβάνει το γράψιμο ένθετων (nested) προτάσεων case, προτάσεων if ή οποιαδήποτε άλλη μορφή ακολουθιακών προτάσεων στις εναλλακτικές λύσεις.

Αν και οι προηγούμενοι κανόνες που διέπουν τις προτάσεις case μπορεί να φαίνονται πολύπλοκοι, στην πράξη υπάρχουν μόνο μερικά πράγματα που πρέπει να θυμόμαστε, και αυτά είναι:

- κάθε μία από τις πιθανές τιμές της παράστασης επιλογής πρέπει να καλύπτεται από μια και μόνο επιλογή,
- οι τιμές στις επιλογές πρέπει να είναι τοπικά στατικές και
- εάν η επιλογή **others** χρησιμοποιείται πρέπει να είναι στην τελευταία εναλλακτική λύση και πρέπει να είναι η μοναδική επιλογή σε αυτήν την εναλλακτική λύση.

ΠΑΡΑΔΕΙΓΜΑ

Μπορούμε να γράψουμε ένα μοντέλο συμπεριφοράς ενός πολυπλέκτη (multiplexer) με μια είσοδο επιλογής sel, τέσσερις εισόδους δεδομένων d0, d1, d2 και d3 και μια έξοδο δεδομένων z. Οι είσοδοι και έξοδοι δεδομένων είναι του τύπου IEEE πρότυπης-λογικής, και η είσοδο επιλογής είναι τύπου sel_range, τον οποίο υποθέτουμε ότι έχουμε κάπου αλλού δηλώσει ως εξής

```
type sel_range is range 0 to 3;
```

Η δήλωση οντότητας που καθορίζει τις θύρες και ένα σώμα αρχιτεκτονικής συμπεριφοράς παρουσιάζονται στην Εικόνα 3-2. Το σώμα αρχιτεκτονικής περιέχει μόνο μια δήλωση διεργασίας. Δεδομένου ότι η έξοδος του πολυπλέκτη πρέπει να αλλάξει εάν αλλάξουν οποιεσδήποτε από τις εισόδους δεδομένων ή επιλογής, η διεργασία πρέπει να είναι ευαίσθητη σε όλες τις εισόδους. Χρησιμοποιεί μια πρόταση case που επιλέγει ποια από τις εισόδους δεδομένων πρόκειται να ανατεθεί στην έξοδο δεδομένων.

ΕΙΚΟΝΑ 3-2

```
library ieee; use ieee.std_logic_1164.all;
entity mux4 is
     port ( sel : in sel_range;
            d0, d1, d2, d3 : in std_ulogic;
            z : out std_ulogic );
end entity mux4;
```

```
architecture demo of mux4 is
begin
    out_select : process (sel, d0, d1, d2, d3) is
    begin
        case sel is
            when 0 =>
                z <= d0;
            when 1 =>
                z <= d1;
            when 2 =>
                z <= d2;
            when 3 =>
                z <= d3;
        end case;
    end process out_select;
end architecture demo;
```

Μια οντότητα και το σώμα αρχιτεκτονικής για ένα πολυπλέκτη 4-εισόδων.

VHDL-87

Οι προτάσεις case δεν επιτρέπεται να έχουν ετικέτες στη VHDL-87.

3.3 Προτάσεις Null

Μερικές φορές όταν γράφουμε μοντέλα χρειάζεται να προδιαγράψουμε ότι όταν προκύψει κάποια συνθήκη, δεν πρόκειται να εκτελεσθεί καμία ενέργεια. Αυτή η ανάγκη προκύπτει συχνά όταν χρησιμοποιούμε προτάσεις case, δεδομένου ότι πρέπει να συμπεριλάβουμε μια εναλλακτική λύση για κάθε πιθανή τιμή της παράστασης επιλογής. Αντί να αφήσουμε κενό το τμήμα των προτάσεων μιας εναλλακτικής λύσης, μπορούμε να χρησιμοποιήσουμε μια *πρόταση null (null statement)* για να δηλώσουμε ρητά ότι δεν πρόκειται να γίνει τίποτα. Ο συντακτικός κανόνας για την πρόταση null είναι απλά

null_statement ⇐ [label :] **null** ;

Η προαιρετική ετικέτα χρησιμεύει στο να προσδιορίσει την πρόταση. Μια απλή πρόταση null χωρίς ετικέτα είναι

```
null;
```

Ένα παράδειγμα της χρήσης της σε μια πρόταση case είναι

```
case opcode is
    when add =>
        Acc := Acc + operand;
    when subtract =>
        Acc := Acc – operand;
    when nop =>
        null;
end case;
```

Μπορούμε να χρησιμοποιήσουμε μια πρόταση null σε οποιοδήποτε μέρος απαιτείται μια ακολουθιακή πρόταση, και όχι μόνο σε μια εναλλακτική λύση μιας πρότασης case. Μια πρόταση null μπορεί να χρησιμοποιηθεί κατά τη διάρκεια της φάσης ανάπτυξης του γραψίματος ενός μοντέλου. Εάν ξέρουμε, για παράδειγμα, ότι θα χρειαστούμε μια οντότητα ως τμήμα ενός συστήματος, αλλά δεν είμαστε ακόμα σε θέση να γράψουμε ένα λεπτομερές μοντέλο για αυτήν, μπορούμε να γράψουμε ένα μοντέλο συμπεριφοράς που δεν κάνει τίποτα. Ένα τέτοιο μοντέλο περιλαμβάνει απλά μια διεργασία με μια πρόταση null στο σώμα της:

```
control_section : process ( sensitivity-list ) is
begin
    null;
end process control_section;
```

Σημειώστε ότι η διεργασία πρέπει να περιλαμβάνει τον κατάλογο ευαισθησίας, για λόγους που θα εξηγηθούν στο Κεφάλαιο 5.

VHDL-87

Οι προτάσεις null δεν επιτρέπεται να έχουν ετικέτες στη VHDL-87.

3.4 Προτάσεις Loop

Συχνά χρειάζεται να γράψουμε μια ακολουθία προτάσεων που πρόκειται να εκτελεσθεί κατ' επανάληψη. Χρησιμοποιούμε μια *πρόταση loop (loop statement)* για να εκφράσουμε αυτήν την συμπεριφορά. Υπάρχουν αρκετές διαφορετικές μορφές προτάσεων loop στη VHDL - η απλούστερη είναι ένας βρόχος που επαναλαμβάνει μια ακολουθία προτάσεων επ' αόριστον, αποκαλούμενη συχνά ως *ατέρμονος βρόχος (infinite loop)*. Ο συντακτικός κανόνας για αυτό το είδος βρόχου είναι

loop_statement ⇐

[loop_label :]

loop
{ sequential_statement }

end loop [loop_label] ;

Στις περισσότερες γλώσσες προγραμματισμού υπολογιστών, ένας ατέρμονος βρόχος δεν είναι επιθυμητός, επειδή αυτό σημαίνει ότι το πρόγραμμα δεν τερματίζει ποτέ. Εντούτοις, όταν μοντελοποιούμε ψηφιακά συστήματα, ένας ατέρμονος βρόχος μπορεί να είναι χρήσιμος, δεδομένου ότι πολλές συσκευές υλικού εκτελούν επανειλημμένα την ίδια λειτουργία έως ότου κλείσουμε την τροφοδοσία τους. Χαρακτηριστικά ένα μοντέλο για ένα τέτοιο σύστημα περιλαμβάνει μια πρόταση loop σε ένα σώμα διεργασίας - ο βρόχος, με τη σειρά του, περιέχει μια πρόταση wait.

ΠΑΡΑΔΕΙΓΜΑ

Η Εικόνα 3-3 είναι ένα μοντέλο για ένα μετρητή (counter) που αρχίζει από το μηδέν και αυξάνει σε κάθε μετάβαση του ρολογιού από '0' σε '1'. Όταν ο μετρητής φθάσει στο 15, γυρνάει πίσω στο μηδέν στην επόμενη μετάβαση του ρολογιού. Το σώμα αρχιτεκτονικής για το μετρητή περιέχει μια διεργασία που πρώτα αρχικοποιεί την έξοδο count στο μηδέν, κατόπιν αναμένει επανειλημμένα μια μετάβαση του ρολογιού προτού αυξήσει την τιμή της μέτρησης. Η πρόταση wait σε αυτό το παράδειγμα προκαλεί τη διεργασία να αναστείλει την εκτέλεσή της στη μέση του βρόχου. Όταν το σήμα clk αλλάξει από '0' σε '1', η διεργασία επαναρχίζει και ενημερώνει την τιμή της μέτρησης και την έξοδο count. Ο βρόχος στη συνέχεια επαναλαμβάνεται αρχίζοντας από την πρόταση wait, έτσι η διεργασία αναστέλλεται πάλι.

ΕΙΚΟΝΑ 3-3

```
entity counter is
    port ( clk : in bit; count : out natural );
end entity counter;
```

```
architecture behavior of counter is
begin
    incrementer : process is
        variable count_value : natural := 0;
    begin
        count <= count_value;
        loop
            wait until clk = '1';
            count_value := (count_value + 1) mod 16;
            count <= count_value;
        end loop;
    end process incrementer;
end architecture behavior;
```

Μια οντότητα και το σώμα αρχιτεκτονικής για ένα μετρητή.

Ένα άλλο σημείο που πρέπει να υπογραμμίσουμε παρεμπιπτόντως είναι ότι η πρόταση διεργασίας δεν περιλαμβάνει λίστα ευαισθησίας. Αυτό είναι επειδή περιλαμβάνει μια πρόταση wait. Μια διεργασία μπορεί να περιέχει είτε μια λίστα ευαισθησίας είτε προτάσεις wait, αλλά όχι και τα δύο. Θα επιστρέψουμε σε αυτό για περισσότερες λεπτομέρειες στο Κεφάλαιο 5.

3.4.1 Προτάσεις Exit

Στο προηγούμενο παράδειγμα, ο βρόχος εκτελεί επανειλημμένα τις εσωκλειόμενες προτάσεις, χωρίς να υπάρχει τρόπος να σταματήσει. Συνήθως χρειάζεται να βγούμε από το βρόχο όταν προκύψει κάποια συνθήκη. Μπορούμε να χρησιμοποιήσουμε μια *πρόταση exit (exit statement)* για να βγούμε από ένα βρόχο. Ο συντακτικός κανόνας είναι

exit_statement ⇐

[label :] **exit** [loop_label] [**when** boolean_expression] ;

Η προαιρετική ετικέτα στην αρχή της πρότασης exit χρησιμεύει ώστε να προσδιορίσει την πρόταση. Η απλούστερη μορφή της πρότασης exit είναι απλά

exit;

Όταν αυτή η πρόταση εκτελεστεί, οποιαδήποτε από τις υπόλοιπες προτάσεις στο βρόχο παραλείπονται, και ο έλεγχος μεταφέρεται στην πρόταση μετά από τις λέξεις-κλειδιά **end loop**. Έτσι σε έναν βρόχο μπορούμε να γράψουμε

```
if condition then
    exit;
end if;
```

όπου *condition* είναι μια λογική (Boolean) παράσταση. Δεδομένου ότι αυτό είναι ίσως η πιο κοινή χρήση της πρότασης wait, η VHDL παρέχει έναν σύντομο τρόπο γραφής για αυτό, με τη χρησιμοποίηση της φράσης **when.** Χρησιμοποιούμε μια πρόταση exit με τη φράση **when** σε ένα βρόχο της μορφής

```
loop
    …
    exit when condition;
    …
end loop;
…       -- ο έλεγχος μεταφέρεται εδώ
        -- όταν το condition γίνει αληθές εντός του βρόχου
```

ΠΑΡΑΔΕΙΓΜΑ

Τώρα αναθεωρούμε το προηγούμενο μοντέλο μετρητή για να εισάγουμε μια είσοδο reset η οποία, όταν είναι '1', προκαλεί το μηδενισμό (αρχικοποίηση στο μηδέν) της εξόδου count. Η έξοδος παραμένει στο μηδέν όσο η είσοδος reset είναι '1' και επαναρχίζει τη μέτρηση στην επόμενη μετάβαση του ρολογιού αφού το reset αλλάξει σε '0'. Η αναθεωρημένη δήλωση της οντότητας, που φαίνεται στην Εικόνα 3-4, περιλαμβάνει τη νέα θύρα εισόδου.

ΕΙΚΟΝΑ 3-4

```
entity counter is
    port ( clk, reset : in bit; count : out natural );
end entity counter;
```

```
architecture behavior of counter is
begin
    incrementer : process is
        variable count_value : natural := 0;
    begin
        count <= count_value;
        loop
                loop
                wait until clk = '1' or reset = '1';
                exit when reset = '1';
                count_value := (count_value + 1) mod 16;
                count <= count_value;
            end loop;
            -- σε αυτό το σημείο, reset = '1'
            count_value := 0;
            count <= count_value;
            wait until reset = '0';
        end loop;
    end process incrementer;
end architecture behavior;
```

Μια οντότητα και το σώμα αρχιτεκτονικής του αναθεωρημένου μετρητή, που περιλαμβάνει μια είσοδο reset..

Το σώμα αρχιτεκτονικής αναθεωρήθηκε με την τοποθέτηση του βρόχου μέσα σε μια άλλη πρόταση loop και την προσθήκη του σήματος reset στην αρχική πρόταση wait. Ο εσωτερικός βρόχος εκτελεί την ίδια λειτουργία με πριν, εκτός από το ότι όταν το reset αλλάξει σε '1', η διεργασία επαναρχίζει, και η πρόταση exit προκαλεί τον εσωτερικό βρόχο να τερματιστεί. Ο έλεγχος μεταφέρεται στην πρόταση αμέσως μετά από το τέλος του εσωτερικού βρόχου. Όπως υποδηλώνει το σχόλιο, γνωρίζουμε ότι αυτό ο έλεγχος μπορεί να φτάσει σε αυτό το σημείο μόνο όταν το reset είναι '1'. Η τιμή μέτρησης και η έξοδος count αρχικοποιούνται, και η διεργασία κατόπιν περιμένει να επιστρέψει το reset στο '0'. Ενώ αναστέλλεται σε αυτό το σημείο, οποιεσδήποτε αλλαγές στην είσοδο του ρολογιού αγνοούνται. Όταν το reset αλλάξει σε '0', η διεργασία επαναρχίζει, και ο εξωτερικός βρόχος επαναλαμβάνεται.

Αυτό το παράδειγμα επεξηγεί επίσης ένα άλλο σημαντικό σημείο. Όταν έχουμε ένθετες (nested) προτάσεις loop, με μια πρόταση exit εντός του εσωτερικού βρόχου, η πρόταση exit προκαλεί τη μεταφορά του ελέγχου μόνο εκτός του εσωτερικού βρόχου, και όχι εκτός του εξωτερικού βρόχου. Εξ ορισμού, μια πρόταση exit μεταφέρει τον έλεγχο εκτός του βρόχου που την εσωκλείει άμεσα.

Σε μερικές περιπτώσεις, μπορεί να επιθυμούμε να μεταφέρουμε τον έλεγχο εκτός ενός εσωτερικού βρόχου καθώς επίσης και εκτός ενός βρόχου που τον εσωκλείει. Μπορούμε να το κάνουμε αυτό δίνοντας ετικέτα στον εξωτερικό βρόχο και χρησιμοποιώντας την ετικέτα στην πρόταση exit. Μπορούμε να γράψουμε

```
loop_name : loop
    …
    exit loop_name;
    …
end loop loop_name ;
```

Αυτό δίνει ετικέτα στο βρόχο με το όνομα loop_name, έτσι ώστε να μπορούμε να προσδιορίσουμε από ποιο βρόχο να εξέλθει η πρόταση exit. Η ετικέτα του βρόχου μπορεί να είναι οποιοδήποτε έγκυρο αναγνωριστικό. Η πρόταση exit που αναφέρεται σε αυτήν την ετικέτα μπορεί να βρεθεί μέσα στις ένθετες προτάσεις loop.

Για να εξηγήσουμε πώς μπορούμε να ενθέσουμε βρόχους, να τους δώσουμε ετικέτες και να εξέλθουμε από αυτούς, ας εξετάσουμε τις ακόλουθες προτάσεις:

```
outer : loop
    …
    inner : loop
        …
        exit outer when condition-1;    -- έξοδος 1
        …
        exit when condition-2;          -- έξοδος 2
        …
    end loop inner;
    … -- στόχος A
    exit outer when condition-3;        -- έξοδος 3
    …
end loop outer;
…                                       -- στόχος B
```

Αυτό το παράδειγμα περιέχει δύο προτάσεις loop, η μία με ετικέτα inner τοποθετημένη μέσα σε μια άλλη με την ετικέτα outer. Η πρώτη πρόταση exit, που φέρει το σχόλιο έξοδος 1, μεταφέρει τον έλεγχο στην πρόταση που φέρει την ένδειξη στόχος B εάν η συνθήκη της είναι αληθής. Η δεύτερη πρόταση exit, που φέρει την ένδειξη έξοδος 2, μεταφέρει τον έλεγχο στην πρόταση στόχος A. Δεδομένου ότι δεν αναφέρεται σε μια ετικέτα, εξέρχεται από την πρόταση loop που την εσωκλείει άμεσα, δηλαδή, το βρόχο inner. Τέλος, η πρόταση exit με την ένδειξη έξοδος 3 μεταφέρει τον έλεγχο στην πρόταση στόχος B.

VHDL-87

Οι προτάσεις exit δεν επιτρέπεται να έχουν ετικέτες στη VHDL-87.

3.4.2 Προτάσεις Next

Ένα άλλο είδος πρότασης που μπορούμε να χρησιμοποιήσουμε για να ελέγξουμε την εκτέλεση των βρόχων είναι η *πρόταση next (next statement)*. Όταν αυτή η πρόταση εκτελείται, η τρέχουσα επανάληψη του βρόχου ολοκληρώνεται χωρίς την εκτέλεση περαιτέρω προτάσεων, και η επόμενη επανάληψη αρχίζει. Ο συντακτικός κανόνας είναι

next_statement ⇐

[label :] **next** [loop_label] [**when** boolean_expression] ;

Η προαιρετική ετικέτα στην αρχή της πρότασης next χρησιμεύει ώστε να προσδιορίσει την πρόταση. Μια πρόταση next μοιάζει πολύ στη μορφή με μια πρόταση exit, με τη διαφορά να είναι η χρήση της λέξης-κλειδί **next** αντί της **exit**. Η απλούστερη μορφή της πρότασης next είναι

next;

η οποία αρχίζει την επόμενη επανάληψη του βρόχου που την εσωκλείει άμεσα. Μπορούμε επίσης να συμπεριλάβουμε μια συνθήκη που πρέπει να εξεταστεί πριν την ολοκλήρωση της επανάληψης:

next when *condition*;

και μπορούμε να συμπεριλάβουμε μια ετικέτα βρόχου που υποδηλώνει για ποιο βρόχο να ολοκληρώσουμε την επανάληψη:

next loop-label; ή

next *loop-label* **when** *condition*;

Μια πρόταση next που εξέρχεται από το βρόχο που την εσωκλείει άμεσα μπορεί να ξαναγραφεί εύκολα ως ένας ισοδύναμος βρόχος με μια πρόταση if να αντικαθιστά την πρόταση next. Για παράδειγμα, οι δύο παρακάτω βρόχοι είναι ισοδύναμοι:

```
loop                                    loop
    statement-1;                            statement-1;
    next when condition;                    if not condition then
    statement-2;                                statement-2;
end loop;                                   end if;
                                        end loop;
```

Εντούτοις, οι ένθετοι βρόχοι με ετικέτες που περιέχουν προτάσεις next οι οποίες αναφέρονται στους εξωτερικούς βρόχους δεν μπορούν να ξαναγραφούν τόσο εύκολα. Γενικώς αυτό που ισχύει είναι ότι εάν συλλάβουμε τους εαυτούς μας να γράφουν μια τέτοια συλλογή βρόχων και προτάσεων next, είναι πιθανώς η στιγμή να σκεφτούμε προσεκτικότερα αυτό που προσπαθούμε να εκφράσουμε. Εάν ελέγξουμε τη λογική του μοντέλου, μπορεί να είμαστε σε θέση να βρούμε μια απλούστερη διατύπωση των προτάσεων loop. Οι περίπλοκες δομές προτάσεων loop/next μπορεί να προκαλούν σύγχυση, καθιστώντας το μοντέλο δύσκολο στην ανάγνωση και την κατανόηση.

VHDL-87

Οι προτάσεις next δεν επιτρέπεται να έχουν ετικέτες στη VHDL-87.

3.4.3 Βρόχοι While

Μπορούμε να συμπληρώσουμε τη βασική πρόταση loop που εισαγάγαμε προηγουμένως για να διαμορφώσουμε ένα *βρόχο while (while loop)*, ο οποίος εξετάζει μια συνθήκη πριν από κάθε επανάληψη. Εάν η συνθήκη είναι αληθής, η επανάληψη προχωρά. Εάν είναι ψευδής, ο βρόχος τερματίζεται. Ο συντακτικός κανόνας για ένα βρόχο while είναι

loop_statement ⇐

[loop_label :]

while boolean_expression **loop**

{ sequential_statement }

end loop [loop_label] ;

Η μόνη διαφορά μεταξύ αυτής της μορφής και της βασικής πρότασης loop είναι ότι έχουμε προσθέσει τη λέξη-κλειδί **while** και τη συνθήκη πριν από τη λέξη κλειδί **loop**. Όλα όσα είπαμε για τη βασική πρόταση loop ισχύουν επίσης για ένα βρόχο while. Μπορούμε να γράψουμε οποιεσδήποτε ακολουθιακές προτάσεις στο σώμα του βρόχου, συμπεριλαμβανομένων των προτάσεων exit και next, και μπορούμε να δώσουμε ετικέτα στο βρόχο γράφοντας την ετικέτα πριν από τη λέξη-κλειδί **while**.

Υπάρχουν τρία σημαντικά σημεία που πρέπει να υπογραμμίσουμε για τους βρόχους while. Το πρώτο σημείο είναι ότι η συνθήκη εξετάζεται πριν από κάθε επανάληψη του βρόχου, συμπεριλαμβανομένης της πρώτης επανάληψης. Αυτό σημαίνει ότι εάν η συνθήκη είναι ψευδής προτού να αρχίσουμε το βρόχο, αυτός τερματίζεται αμέσως, χωρίς να εκτελεστεί καμία επανάληψη. Για παράδειγμα, δοθέντος του βρόχου while

```
while index > 0 loop
    …           -- πρόταση A: κάνει κάτι με το index
end loop;
…               -- πρόταση B
```

εάν μπορούμε να δείξουμε ότι το index δεν είναι μεγαλύτερο από το μηδέν προτού να αρχίσει ο βρόχος, τότε ξέρουμε ότι οι προτάσεις μέσα στο βρόχο δεν θα εκτελεσθούν, και ο έλεγχος θα μεταφερθεί κατ' ευθείαν στην πρόταση B.

Το δεύτερο σημείο είναι ότι ελλείψει των προτάσεων exit μέσα σε ένα βρόχο while, ο βρόχος τερματίζεται μόνο όταν η συνθήκη γίνει ψευδής. Κατά συνέπεια, ξέρουμε ότι η άρνηση της συνθήκης πρέπει να ισχύει όταν ο έλεγχος φθάσει στην πρόταση μετά από το βρόχο. Ομοίως, ελλείψει των προτάσεων next μέσα σε ένα βρόχο while, ο βρόχος εκτελεί μια επανάληψη μόνο όταν η συνθήκη είναι αληθής. Κατά συνέπεια, ξέρουμε ότι η συνθήκη ισχύει όταν αρχίσουμε τις προτάσεις στο σώμα του βρόχου. Στο παραπάνω παράδειγμα, ξέρουμε ότι το index πρέπει να είναι μεγαλύτερο από το μηδέν όταν εκτελέσουμε την πρόταση που φέρει την ένδειξη πρόταση A, και επίσης ότι το index πρέπει να είναι μικρότερο ή ίσο του μηδενός όταν φθάσουμε να εκτελέσουμε την πρόταση B. Αυτή η γνώση μπορεί να μας βοηθήσει να αναλύσουμε λογικά την ορθότητα του μοντέλου που γράφουμε.

Το τρίτο σημείο είναι ότι όταν γράφουμε τις προτάσεις μέσα στο σώμα ενός βρόχου while, πρέπει να σιγουρευτούμε ότι η συνθήκη θα γίνει τελικά ψευδής, ή ότι μια πρόταση exit θα επιφέρει τελικά την έξοδο από το βρόχο. Διαφορετικά ο βρόχος while δεν θα τερματιστεί ποτέ. Προφανώς, εάν σκοπεύαμε να γράψουμε έναν ατέρμονο βρόχο, θα είχαμε χρησιμοποιήσει μια απλή πρόταση loop.

ΠΑΡΑΔΕΙΓΜΑ

Μπορούμε να αναπτύξουμε ένα μοντέλο για μια οντότητα cos (συνημίτονο) η οποία θα μπορούσε να χρησιμοποιηθεί ως τμήμα ενός εξειδικευμένου συστήματος επεξεργασίας σήματος (signal processing system). Η οντότητα έχει μια είσοδο, theta, που είναι ένας πραγματικός αριθμός ο οποίος αναπαριστά μια γωνία σε ακτίνια (radians), και μια έξοδο, result, που αναπαριστά τη συνάρτηση συνημιτόνου (cosine function) της τιμής του theta. Μπορούμε να χρησιμοποιήσουμε τη σχέση

$$\cos\theta = 1 - \frac{\theta^2}{2!} + \frac{\theta^4}{4!} - \frac{\theta^6}{6!} + \ldots$$

προσθέτοντας διαδοχικούς όρους της σειράς έως ότου οι όροι να γίνουν μικρότεροι από το ένα εκατομμυριοστό του αποτελέσματος. Οι δηλώσεις της οντότητας και του σώματος της αρχιτεκτονικής παρουσιάζονται στην Εικόνα 3-5.

ΕΙΚΟΝΑ 3-5

```
entity cos is
    port ( theta : in real; result : out real );
end entity cos;
```

```
architecture series of cos is
begin
    summation : process (theta) is
        variable sum, term : real;
        variable n : natural;
    begin
        sum := 1.0;
        term := 1.0;
        n := 0;
        while abs term > abs (sum / 1.0E6) loop
            n := n + 2;
            term := (–term) * theta**2 / real(((n–1) * n));
            sum := sum+ term;
        end loop;
        result <= sum;
    end process summation;
end architecture series;
```

Μια οντότητα και το σώμα αρχιτεκτονικής για μια λειτουργική μονάδα συνημιτόνου.

Το σώμα αρχιτεκτονικής αποτελείται από μια διεργασία που είναι ευαίσθητη στις αλλαγές στο σήμα εισόδου theta. Αρχικά, οι μεταβλητές sum και term τίθενται στο 1,0, αναπαριστώντας τον πρώτο όρο στη σειρά. Η μεταβλητή n αρχίζει από το 0 για τον πρώτο όρο. Η συνάρτηση συνημιτόνου υπολογίζεται χρησιμοποιώντας ένα βρόχο while που αυξάνει το n κατά δύο και το χρησιμοποιεί για να υπολογίσει τον επόμενο όρο βασιζόμενη στον προηγούμενο όρο. Η επανάληψη προχωρά όσο ο τελευταίος όρος που υπολογίζεται είναι μεγαλύτερος στο μέγεθος από το ένα εκατομμυριοστό του αθροίσματος. Όταν ο τελευταίος όρος πέσει κάτω από αυτό το κατώφλι, ο βρόχος while τερματίζεται. Μπορούμε να καθορίσουμε ότι ο βρόχος θα τερματιστεί, αφού οι τιμές των διαδοχικών όρων της σειράς γίνονται σταδιακά μικρότερες. Αυτό συμβαίνει επειδή η παραγοντική συνάρτηση αυξάνεται με ένα ρυθμό μεγαλύτερο από ότι η εκθετική συνάρτηση.

3.4.4 Βρόχοι For

Ένας άλλος τρόπος με τον οποίο μπορούμε να συμπληρώσουμε τη βασική πρόταση loop είναι ο *βρόχος for (for loop)*. Ένας βρόχος for περιλαμβάνει μια προδιαγραφή του πόσες φορές το σώμα του βρόχου πρόκειται να εκτελεσθεί. Ο συντακτικός κανόνας για το βρόχο for είναι

```
loop_statement ⇐
    [ loop_label : ]
        for identifier in discrete_range loop
            { sequential_statement }
        end loop [ loop_label ] ;
```

Είδαμε ότι ένα διακριτό εύρος μπορεί να είναι της μορφής

simple_expression (**to** I **downto**) simple_expression

αναπαριστώντας όλες τις τιμές μεταξύ του αριστερού και του δεξιού ορίου, συμπεριλαμβανομένων και των ορίων. Το αναγνωριστικό καλείται *παράμετρος βρόχου (loop parameter)*, και για κάθε επανάληψη του βρόχου, παίρνει τις διαδοχικές τιμές του διακριτού εύρους, αρχίζοντας από το αριστερό στοιχείο. Για παράδειγμα, σε αυτό το βρόχο for:

```
for count_value in 0 to 127 loop
    count_out <= count_value;
    wait for 5 ns;
end loop;
```

το αναγνωριστικό count_value παίρνει τις τιμές 0, 1, 2 κτλ., και για κάθε τιμή, προτάσεις ανάθεσης και wait εκτελούνται. Κατά συνέπεια στο σήμα count_out θα ανατεθούν οι τιμές 0, 1, 2 κτλ., μέχρι 127, σε διαστήματα των 5 ns.

Επίσης είδαμε ότι ένα διακριτό εύρος μπορεί να καθοριστεί χρησιμοποιώντας ένα όνομα διακριτού τύπου ή υποτύπου, που ενδεχομένως περιορίζεται περαιτέρω σε ένα υποσύνολο των τιμών από έναν περιορισμό εύρους. Για παράδειγμα, εάν έχουμε τον τύπο απαρίθμησης

```
type controller_state is (initial, idle, active, error);
```

μπορούμε να γράψουμε ένα βρόχο for που επαναλαμβάνεται για κάθε μια από τις τιμές του τύπου:

```
for state in controller_state loop
    …
end loop;
```

Εντός της ακολουθίας προτάσεων στο σώμα του βρόχου for, η παράμετρος βρόχου είναι μια σταθερά ο τύπος της οποίας είναι ο τύπος βάσης του διακριτού εύρους. Αυτό σημαίνει ότι μπορούμε να χρησιμοποιήσουμε την τιμή της συμπεριλαμβάνοντάς την σε μια παράσταση, αλλά δεν μπορούμε να κάνουμε αναθέσεις σε αυτήν. Αντίθετα από άλλες σταθερές, δεν απαιτείται να τη δηλώσουμε. Αντί για αυτό, η παράμετρος βρόχου δηλώνεται αυτονόητα από το βρόχο for. Υφίσταται μόνο όταν εκτελείται ο βρόχος, και όχι πριν ή μετά από αυτόν. Για παράδειγμα, η ακόλουθη πρόταση διεργασίας δείχνει πως να μην χρησιμοποιούμαι την παράμετρο βρόχου:

```
erroneous : process is
    variable i, j : integer;
begin
    i := loop_param;                -- λάθος!
    for loop_param in 1 to 10 loop
        loop_param := 5;            -- λάθος!
    end loop;
    j := loop_param;                -- λάθος!
end process erroneous;
```

Οι αναθέσεις στο i και το j δεν είναι νόμιμες αφού η παράμετρος βρόχου δεν πρέπει να καθορίζεται ούτε πριν ούτε μετά από το βρόχο. Η ανάθεση μέσα στο σώμα του βρόχου δεν είναι νόμιμη επειδή το loop_param είναι μια σταθερά και έτσι δεν μπορεί να τροποποιηθεί.

Ένα επακόλουθο του τρόπου που η παράμετρος βρόχου καθορίζεται είναι ότι κρύβει οποιοδήποτε αντικείμενο έχει καθοριστεί με το ίδιο όνομα έξω από το βρόχο. Για παράδειγμα, σε αυτήν τη διεργασία:

```
hiding_example : process is
    variable a, b : integer;
begin
    a := 10;
    for a in 0 to 7 loop
        b := a;
    end loop;
    -- a = 10, και b = 7
    …
end process hiding_example;
```

η μεταβλητή a τίθεται αρχικά στην τιμή 10, και έπειτα ο βρόχος for εκτελείται, δημιουργώντας μια παραμέτρου βρόχου που επίσης καλείται a. Μέσα στο βρόχο, η ανάθεση στη b χρησιμοποιεί την παράμετρο βρόχου, έτσι ώστε η τελική τιμή της b μετά από την τελευταία επανάληψη είναι 7. Μετά από το βρόχο, η παράμετρος βρόχου δεν υπάρχει πλέον, έτσι εάν χρησιμοποιήσουμε το όνομα a, αναφερόμαστε στο αντικείμενο μεταβλητής, της οποίας η τιμή είναι ακόμα 10.

Όπως αναφέραμε παραπάνω, ο βρόχος for επαναλαμβάνεται με την παράμετρο βρόχου να παίρνει διαδοχικές τιμές από το διακριτό εύρος αρχίζοντας από την αριστερότερη τιμή. Ένα σημαντικό σημείο που πρέπει να υπογραμμίσουμε είναι ότι εάν καθορίσουμε ένα κενό εύρος (null range), το σώμα του βρόχου for δεν εκτελείται καθόλου. Ένα κενό

εύρος μπορεί να προκύψει εάν καθορίσουμε μια αύξουσα σειρά με το αριστερό όριο μεγαλύτερο από το δεξί όριο, ή μια φθίνουσα σειρά με το αριστερό όριο μικρότερο από το δεξί όριο. Για παράδειγμα, ο βρόχος for

```
for i in 10 to 1 loop
    …
end loop;
```

ολοκληρώνεται αμέσως, χωρίς να εκτελέσει τις εσωκλειόμενες προτάσεις. Εάν θέλουμε πραγματικά ο βρόχος να επαναληφθεί με το i να παίρνει τις τιμές 10, 9, 8 κτλ.

```
for i in 10 downto 1 loop
    …
end loop;
```

Ένα τελικό πράγμα που πρέπει να υπογραμμίσουμε για τους βρόχους for είναι ότι, όπως και οι βασικές προτάσεις loop, μπορούν να εσωκλείουν αυθαίρετες ακολουθιακές προτάσεις, συμπεριλαμβανομένων των προτάσεων next και exit, και μπορούμε να δώσουμε ετικέτα σε ένα βρόχο for γράφοντας την ετικέτα πριν από τη λέξη-κλειδί **for**.

ΠΑΡΑΔΕΙΓΜΑ

Ξαναγράφουμε τώρα το μοντέλο συνημιτόνου της Εικόνας 3-5 για να υπολογίσουμε το αποτέλεσμα αθροίζοντας μόνο τους 10 πρώτους όρους της σειράς. Η δήλωση της οντότητας παραμένει αμετάβλητη. Το αναθεωρημένο σώμα αρχιτεκτονικής, που παρουσιάζεται στην Εικόνα 3-6, αποτελείται από μια διεργασία που χρησιμοποιεί ένα βρόχο for αντί ενός βρόχου while. Όπως και πριν, οι μεταβλητές sum και term τίθενται στο 1.0, αναπαριστώντας τον πρώτο όρο στη σειρά. Η μεταβλητή n αντικαθίσταται από την παράμετρο βρόχου. Ο βρόχος επαναλαμβάνεται εννέα φορές, υπολογίζοντας τους υπόλοιπους 9 όρους της σειράς.

ΕΙΚΟΝΑ 3-6

```
architecture fixed_length_series of cos is
begin
    summation : process (theta) is
        variable sum, term : real;
    begin
        sum := 1.0; term := 1.0;
        for n in 1 to 9 loop
            term := (–term) * theta**2 / real(((2*n–1) * 2*n));
            sum := sum + term;
        end loop;
        result <= sum;
    end process summation;
end architecture fixed_length_series;
```

Το αναθεωρημένο σώμα αρχιτεκτονικής για τη λειτουργική μονάδα συνημιτόνου.

3.4.5 Περίληψη των Προτάσεων Loop

Οι προηγούμενες ενότητες περιγράφουν λεπτομερώς τις διάφορες μορφές των προτάσεων loop. Αξίζει να συνοψίσουμε αυτές τις πληροφορίες, ώστε να παρουσιάσουμε τα λίγα βασικά σημεία που πρέπει να θυμόμαστε. Πρώτον, ο συντακτικός κανόνας για όλες τις προτάσεις loop είναι

```
loop_statement ⇐
    [ loop_label : ]
        [ while boolean_expression I for identifier in discrete_range ] loop
            { sequential_statement }
    end loop [ loop_label ] ;
```

Δεύτερον, ελλείψει των προτάσεων exit και next, ο βρόχος while επαναλαμβάνεται όσο η συνθήκη παραμένει αληθής, και ο βρόχος for επαναλαμβάνεται με την παράμετρο βρόχου να παίρνει διαδοχικές τιμές από το διακριτό εύρος. Εάν η συνθήκη σε ένα βρόχο while είναι αρχικά ψευδής, ή εάν το διακριτό εύρος σε ένα βρόχο for είναι ένα κενό εύρος, τότε δεν εκτελείται καμία επανάληψη.

Τρίτον, η παράμετρος βρόχου σε ένα βρόχο for δεν μπορεί να δηλωθεί ρητά, και είναι μια σταθερά εντός του σώματος βρόχου. Επίσης σκιάζει οποιοδήποτε άλλο αντικείμενο του ίδιου ονόματος το οποίο δηλώνεται έξω από το βρόχο.

Τέλος, μια πρόταση exit μπορεί να χρησιμοποιηθεί για να τερματίσει οποιοδήποτε βρόχο, και μια πρόταση next μπορεί να χρησιμοποιηθεί για να ολοκληρώσει την τρέχουσα επανάληψη και να αρχίσει την επόμενη επανάληψη. Αυτές οι προτάσεις μπορούν να αναφερθούν στις ετικέτες των βρόχων για να τερματίσουν ή να ολοκληρώσουν την επανάληψη του εξωτερικού επιπέδου ενός συνόλου ένθετων βρόχων.

3.5 Προτάσεις Assertion και Report

Ένας από τους λόγους για το οποίο γράφουμε μοντέλα υπολογιστικών συστημάτων είναι να πιστοποιήσουμε ότι μια σχεδίαση λειτουργεί ορθά. Μπορούμε να δοκιμάσουμε μερικώς ένα μοντέλο εφαρμόζοντας δείγματα εισόδων και ελέγχοντας ότι οι έξοδοι ικανοποιούν τις προσδοκίες μας. Εάν όχι, βρισκόμαστε αντιμέτωποι με το έργο να καθορίσουμε τι πήγε στραβά μέσα στη σχεδίαση. Αυτό το έργο μπορεί να γίνει ευκολότερο χρησιμοποιώντας *προτάσεις ισχυρισμού (assertion statements)* που ελέγχουν ότι οι αναμενόμενες συνθήκες ικανοποιούνται μέσα στο μοντέλο. Μια πρόταση assertion είναι μια ακολουθιακή πρόταση, και έτσι μπορεί να συμπεριληφθεί οπουδήποτε σε ένα σώμα διεργασίας. Ο πλήρης συντακτικός κανόνας για μια πρόταση assertion είναι

```
assertion_statement ⇐
    [ label : ] assert boolean_expression
        [ report expression ] [ severity expression ] ;
```

Η προαιρετική ετικέτα μας επιτρέπει να προσδιορίσουμε την πρόταση assertion. Η απλούστερη μορφή της πρότασης assertion περιλαμβάνει μόνο τη λέξη-κλειδί **assert** ακολουθούμενη από μια λογική (Boolean) παράσταση η οποία αναμένουμε να είναι αληθής όταν εκτελείται η πρόταση assertion. Εάν η συνθήκη δεν ικανοποιείται, λέμε ότι έχει συμβεί μια *παραβίαση ισχυρισμού (assertion violation)*. Εάν μια παραβίαση ισχυρισμού προκύψει κατά τη διάρκεια της προσομοίωσης ενός μοντέλου, ο προσομοιωτής αναφέρει το γεγονός. Κατά τη διάρκεια της σύνθεσης, η συνθήκη σε μια πρόταση assertion μπορεί να ερμηνευθεί ως μια συνθήκη που το εργαλείο σύνθεσης (synthesizer) μπορεί να υποθέσει ότι είναι αληθής. Κατά τη διάρκεια της τυπικής επαλήθευσης, η συνθήκη μπορεί να ερμηνευθεί ως μια συνθήκη που πρέπει να επαληθευτεί από το εργαλείο επαλήθευσης (verifier). Για παράδειγμα, εάν γράψουμε

```
assert initial_value <= max_value;
```

και το initial_value είναι μεγαλύτερο από το max_value όταν εκτελεστεί η πρόταση κατά τη διάρκεια της προσομοίωσης, ο προσομοιωτής θα μας ειδοποιήσει. Κατά τη διάρκεια της σύνθεσης, το εργαλείο σύνθεσης μπορεί να υποθέσει ότι initial_value <= max_value και να βελτιστοποιήσει το κύκλωμα βασισμένο σε αυτήν την πληροφορία. Κατά τη διάρκεια της τυπικής επαλήθευσης, το εργαλείο επαλήθευσης μπορεί να επιχειρήσει να αποδείξει ότι initial_value <= max_value για όλα τα πιθανά ερεθίσματα εισόδου και τα μονοπάτια εκτέλεσης που οδηγούν στην πρόταση assertion.

Εάν έχουμε διάφορες προτάσεις assertion κατά μήκος ενός μοντέλου, είναι χρήσιμο να γνωρίζουμε ποιος ισχυρισμός παραβιάζεται. Μπορούμε να έχουμε τον προσομοιωτή για να παρέχει πρόσθετες πληροφορίες συμπεριλαμβάνοντας μια φράση αναφοράς (**report** clause) σε μια πρόταση assertion, για παράδειγμα:

```
assert initial_value <= max_value
    report "initial value too large";
```

Το αλφαριθμητικό που παρέχουμε χρησιμοποιείται για να αποτελέσει μέρος του μηνύματος παραβίασης του ισχυρισμού. Μπορούμε να γράψουμε οποιαδήποτε παράσταση στη φράση clause υπό τον όρο ότι παράγει μια τιμή αλφαριθμητικού, για παράδειγμα:

```
assert current_character >= '0' and current_character <= '9'
    report "Input number " & input_string & " contains a non–digit";
```

Εδώ το μήνυμα προέρχεται από τη συνένωση τριών τιμών αλφαριθμητικού μαζί.

Στην Ενότητα 2.2, αναφέραμε έναν προκαθορισμένο τύπο απαρίθμησης severity_level, που ορίζεται ως εξής

```
type severity_level is (note, warning, error, failure);
```

Μπορούμε να συμπεριλάβουμε μια τιμή αυτού του τύπου σε μια φράση αυστηρότητας (**severity** clause) μιας πρότασης assertion. Αυτή η τιμή μαρτυρά το βαθμό στον οποίο η παραβίαση του ισχυρισμού επηρεάζει τη λειτουργία του μοντέλου. Η τιμή note μπορεί να χρησιμοποιηθεί για να περάσει ενημερωτικά μηνύματα στο χρήστη κατά την προσομοίωση, για παράδειγμα:

```
assert free_memory >= low_water_limit
    report "low on memory, about to start garbage collect"
    severity note;
```

Το επίπεδο αυστηρότητας (severity level) warning μπορεί να χρησιμοποιηθεί εάν προκύψει μια ασυνήθιστη κατάσταση στην οποία το πρότυπο μπορεί να συνεχίσει να εκτελείται, αλλά μπορεί να παράγει ασυνήθιστα αποτελέσματα, για παράδειγμα:

```
assert packet_length /= 0
    report "empty network packet received"
    severity warning;
```

Μπορούμε να χρησιμοποιήσουμε το επίπεδο αυστηρότητας error για να δείξουμε ότι κάτι έχει πάει σίγουρα στραβά και ότι πρέπει να ληφθούν διορθωτικά μέτρα, για παράδειγμα:

```
assert clock_pulse_width >= min_clock_width
    severity error;
```

Τέλος, η τιμή failure μπορεί να χρησιμοποιηθεί εάν ανιχνεύουμε μια ασυνέπεια που δεν θα έπρεπε να προκύψει ποτέ, για παράδειγμα:

```
assert (last_position – first_position + 1) = number_of_entries
    report "inconsistency in buffer model"
    severity failure;
```

Έχουμε δει ότι μπορούμε να γράψουμε μια πρόταση assertion είτε με μια φράση **report** είτε με μια φράση **severity**, είτε και με τις δύο. Εάν και οι δύο είναι παρούσες, ο συντακτικός κανόνας μας δείχνει ότι η φράση **report** πρέπει να προηγείται. Εάν παραλείψουμε τη φράση **report**, το προκαθορισμένο αλφαριθμητικό στο μήνυμα λάθους είναι "Assertion violation". Εάν παραλείψουμε τη φράση **severity**, η προκαθορισμένη τιμή είναι error. Η τιμή της αυστηρότητας χρησιμοποιείται συνήθως από έναν προσομοιωτή για να αποφασίσει εάν πρέπει ή όχι να συνεχίσει την εκτέλεση μετά από μια παραβίαση ισχυρισμού. Οι περισσότεροι προσομοιωτές επιτρέπουν στο χρήστη να καθορίσει ένα κατώφλι αυστηρότητας (severity threshold), εκτός του οποίου η εκτέλεση σταματά.

Συνήθως, αποτυχία ενός ισχυρισμού σημαίνει είτε ότι η οντότητα χρησιμοποιείται λανθασμένα ως τμήμα μιας μεγαλύτερης σχεδίασης ή ότι το μοντέλο έχει γραφτεί λανθασμένα για την οντότητα. Επεξηγούμε και τις δύο περιπτώσεις.

ΠΑΡΑΔΕΙΓΜΑ

Ένα φλιπ-φλοπ ενεργοποίησης/μηδένισης (set/reset - SR flip-flop) έχει δύο εισόδους, S και R, και μια έξοδο Q. Όταν το S είναι ‘1’, η έξοδος ενεργοποιείται (τίθεται στο ‘1’), και όταν το R είναι ‘1’, η έξοδος μηδενίζεται (τίθεται στο ‘0’). Εντούτοις, τα S και R δεν μπορούν να είναι συγχρόνως ‘1’. Εάν γίνουν, η τιμή της εξόδου είναι απροσδιόριστη. Η Εικόνα 3-7 είναι ένα μοντέλο συμπεριφοράς για ένα φλιπ-φλοπ SR που περιλαμβάνει έναν έλεγχο για αυτήν την παράνομη συνθήκη.

Το σώμα αρχιτεκτονικής περιέχει μια διεργασία ευαίσθητη στις εισόδους S και R. Μέσα στο σώμα της διεργασίας γράφουμε μια πρόταση assertion που απαιτεί ότι τα S και R δεν είναι και τα δύο ‘1’. Εάν και τα δύο είναι ‘1’, ο ισχυρισμός παραβιάζεται, και έτσι ο προσομοιωτής γράφει ένα μήνυμα "Assertion violation" με αυστηρότητα error. Εάν η εκτέλεση συνεχιστεί μετά από τον παραβιασμένο ισχυρισμό, θα ανατεθεί πρώτα η τιμή ‘1’ στο Q, και μετά θα ακολουθήσει η τιμή ‘0’. Η τιμή που θα προκύψει είναι ‘0’. Αυτό επιτρέπεται, δεδομένου ότι η κατάσταση του Q δεν έχει προσδιοριστεί για αυτήν την παράνομη συνθήκη, έτσι έχουμε την ελευθερία να επιλέξουμε οποιαδήποτε τιμή. Εάν ο ισχυρισμός δεν παραβιαστεί, τότε το πολύ μία από τις ακόλουθες προτάσεις εκτελείται, μοντελοποιώντας σωστά τη συμπεριφορά του φλιπ-φλοπ SR.

ΕΙΚΟΝΑ 3-7

```
entity SR_flipflop is
    port ( S, R : in bit; Q : out bit );
end entity SR_flipflop;
```

```
architecture checking of SR_flipflop is
begin
    set_reset : process (S, R) is
    begin
        assert S = '1' nand R = '1';
        if S = '1' then
            Q <= '1';
        end if;
        if R = '1' then
            Q <= '0';
        end if;
    end process set_reset;
end architecture checking;
```

Μια οντότητα και το σώμα αρχιτεκτονικής για ένα φλιπ-φλοπ SR, που περιλαμβάνει έναν έλεγχο για σωστή χρήση.

ΠΑΡΑΔΕΙΓΜΑ

Για να επεξηγήσουμε τη χρήση μιας πρότασης assertion ως "έλεγχος λογικότητας" ας εξετάσουμε ένα μοντέλο, που παρουσιάζεται στην Εικόνα 3-8, για μια οντότητα που έχει τρεις ακέραιες εισόδους, a, b και c, και παράγει μια ακέραια έξοδος z που είναι η μεγαλύτερη από τις εισόδους της.

Το σώμα αρχιτεκτονικής γράφεται χρησιμοποιώντας μια διεργασία που περιέχει ένθετες προτάσεις if. Για αυτό το παράδειγμα έχουμε εισάγει ένα "τυχαίο" λάθος στο μοντέλο. Εάν προσομοιώσουμε αυτό το μοντέλο και βάλουμε τις τιμές a = 7, b = 3 και c = 9 στις θύρες αυτής της οντότητας, αναμένουμε ότι η τιμή του result, και ως εκ τούτου η θύρα εξόδου, θα είναι 9. Ο ισχυρισμός δηλώνει ότι η τιμή του result πρέπει να είναι μεγαλύτερη ή ίση από όλες τις εισόδους. Εντούτοις, το λάθος κωδικοποίησης προκαλεί την ανάθεση της τιμής 7 στο result, και έτσι ο ισχυρισμός παραβιάζεται. Αυτή η παραβίαση μας αναγκάζει να εξετάσουμε το μοντέλο μας πιο προσεκτικά, και να διορθώσουμε το λάθος.

ΕΙΚΟΝΑ 3-8

```
entity max3 is
    port ( a, b, c : in integer; z : out integer );
end entity max3;
```

```
architecture check_error of max3 is
begin
    maximizer : process (a, b, c)
        variable result : integer;
    begin
        if a > b then
            if a > c then
                result := a;
                else
                result := a; — Ουπς! Θα έπρεπε να είναι: result := c;
            end if;
        elsif b > c then
            result := b;
        else
            result := c;
        end if;
        assert result >= a and result >= b and result >= c
            report "inconsistent result for maximum"
            severity failure;
        z <= result;
    end process maximizer;
end architecture check_error;
```

Μια οντότητα και το σώμα αρχιτεκτονικής για μια λειτουργική μονάδα επιλογής της μέγιστης τιμής, που περιλαμβάνει έναν έλεγχο για ένα σωστά παραγόμενο αποτέλεσμα.

Μια άλλη σημαντική χρήση για τις προτάσεις assertion είναι στον έλεγχο των περιορισμών χρονισμού (timing constraints) που ισχύουν σε ένα μοντέλο. Για παράδειγμα, οι περισσότερες συσκευές με ρολόι απαιτούν ότι ο παλμός του ρολογιού (clock pulse) είναι μεγαλύτερος από κάποια ελάχιστη χρονική περίοδο. Μπορούμε να χρησιμοποιήσουμε την προκαθορισμένη συνάρτηση now σε μια παράσταση για να υπολογίσουμε χρονικές περιόδους. Η συνάρτηση now όταν αξιολογείται παράγει τον τρέχοντα χρόνο προσομοίωσης.

ΠΑΡΑΔΕΙΓΜΑ

Ένας καταχωρητής ενεργοποιούμενος σε ακμή (edge-triggered register) έχει μια είσοδο δεδομένων και μια έξοδο δεδομένων τύπου real και μια είσοδο ρολογιού τύπου bit. Όταν το ρολόι αλλάζει από ‘0’ σε ‘1’, δειγματοληπτείται η είσοδος δεδομένων, αποθηκεύεται και μεταδίδεται στην έξοδο. Υποθέστε ότι η είσοδος ρολογιού πρέπει να παραμείνει στο ‘1’ για τουλάχιστον 5 ns. Η Εικόνα 3-9 είναι ένα μοντέλο για αυτόν τον καταχωρητή, που περιλαμβάνει έναν έλεγχο για το νόμιμο πλάτος του παλμού του ρολογιού.

Το σώμα αρχιτεκτονικής περιέχει μια διεργασία που είναι ευαίσθητη στις αλλαγές στην είσοδο του ρολογιού. Όταν το ρολόι αλλάζει από ‘0’ σε ‘1’, η είσοδος αποθηκεύεται, και ο τρέχων χρόνος προσομοίωσης καταγράφεται στη μεταβλητή pulse_start. Όταν το ρολόι αλλάζει από ‘1’ σε ‘0’, η διαφορά μεταξύ pulse_start και του τρέχοντος χρόνου προσομοίωσης ελέγχεται από την πρόταση assertion.

ΕΙΚΟΝΑ 3-6

```
entity edge_triggered_register is
     port ( clock : in bit;
            d_in : in real; d_out : out real );
end entity edge_triggered_register;
```

```
architecture check_timing of edge_triggered_register is
begin
    store_and_check : process (clock) is
        variable stored_value : real;
        variable pulse_start : time;
    begin
        case clock is
            when '1' =>
                pulse_start := now;
                stored_value := d_in;
                d_out <= stored_value;
            when '0' =>
                assert now = 0 ns or (now – pulse_start) >= 5 ns
                      report "clock pulse too short";
        end case;
    end process store_and_check;
end architecture check_timing;
```

Μια οντότητα και το σώμα αρχιτεκτονικής για έναν καταχωρητή ενεργοποιούμενο σε ακμή, που περιλαμβάνει ένα χρονικό έλεγχο για το σωστό πλάτος παλμού στην είσοδο ρολογιού.

VHDL-87

Οι προτάσεις assertion δεν επιτρέπεται να έχουν ετικέτες στη VHDL-87.

Η VHDL μας παρέχει επίσης μια *πρόταση αναφοράς (report statement)*, η οποία είναι παρόμοια με μια πρόταση assertion. Ο συντακτικός κανόνας για την πρόταση report δείχνει αυτήν την ομοιότητα:

report_statement _

[label :] **report** expression [**severity** expression] ;

Οι διαφορές είναι ότι δεν υπάρχει καμία συνθήκη, και εάν το επίπεδο αυστηρότητας δεν καθορίζεται, η προεπιλογή είναι note. Πράγματι, η πρόταση report μπορεί να θεωρηθεί ως πρόταση assertion στην οποία η συνθήκη έχει την τιμή false και η αυστηρότητα είναι note, ως εκ τούτου παράγει πάντα το μήνυμα. Ένας τρόπος με τον οποίο η πρόταση report μπορεί να φανεί χρήσιμη είναι για να συμπεριλάβουμε σε ένα μοντέλο "εγγραφές ίχνους" ώστε να μας βοηθήσουν στην απασφαλμάτωση (debugging).

ΠΑΡΑΔΕΙΓΜΑ

Υποθέστε ότι γράφουμε ένα σύνθετο μοντέλο και δεν είμαστε βέβαιοι ότι έχουμε μεταφέρει τη λογική αρκετά σωστά. Μπορούμε να χρησιμοποιήσουμε προτάσεις report για να κάνουμε τις διεργασίες στο μοντέλο να καταγράφουν μηνύματα, έτσι ώστε να μπορούμε να δούμε πότε αυτές ενεργοποιούνται και τι κάνουν. Ένα παράδειγμα διεργασίας είναι

```
transmit_element : process (transmit_data) is
    … — variable declarations
begin
    report "transmit_element: data = "
           & data_type'image(transmit_data);
    …
end process transmit_element;
```

VHDL-87

Οι προτάσεις report δεν παρέχονται στη VHDL-87. Επιτυγχάνουμε το ίδιο αποτέλεσμα γράφοντας μια πρόταση assertion με την συνθήκη false και ένα επίπεδο αυστηρότητας note. Για παράδειγμα, η πρότασης report της VHDL-93 ή της VHDL-2001

```
report "Initialization complete";
```

μπορεί να γραφτεί στη VHDL-87 ως

```
assert false
    report "Initialization complete" severity note;
```

Κεφάλαιο 4: Σύνθετοι Τύποι Δεδομένων και Πράξεις

Τώρα που έχουμε δει τους βασικούς τύπους δεδομένων και τις ακολουθιακές πράξεις από τις οποίες σχηματίζεται το τμήμα της συμπεριφοράς ενός μοντέλου VHDL, είναι ώρα να δούμε τους σύνθετους τύπους δεδομένων (composite data types). Αυτούς τους αναφέραμε για πρώτη φορά στην κατηγοριοποίηση των τύπων δεδομένων στο Κεφάλαιο 2. Τα αντικείμενα σύνθετων δεδομένων αποτελούνται από σχετιζόμενες συλλογές δεδομένων σε μορφή είτε ενός πίνακα (array) είτε μιας εγγραφής (record). Μπορούμε να χειριζόμαστε ένα αντικείμενο σύνθετου τύπου σαν ένα μοναδικό αντικείμενο ή να χειριζόμαστε τα συστατικά του στοιχεία ξεχωριστά. Στο κεφάλαιο αυτό, βλέπουμε πώς να ορίζουμε σύνθετους τύπους και πώς να τους χειριζόμαστε με τη χρήση τελεστών και ακολουθιακών προτάσεων.

4.1 Πίνακες

Ένας *πίνακας* (array) αποτελείται από μια συλλογή τιμών του ίδιου τύπου. Η θέση κάθε στοιχείου σε έναν πίνακα δίνεται από μια βαθμωτή (scalar) τιμή που ονομάζεται *δείκτης* (index) του στοιχείου. Για να δημιουργήσουμε ένα αντικείμενο πίνακα σε ένα μοντέλο, ορίζουμε πρώτα έναν τύπο πίνακα (array type) σε μια δήλωση τύπου (type declaration). Ο συντακτικός κανόνας για τον ορισμό ενός τύπου πίνακα είναι

array_type_definition ⇐
array (discrete_range **{ , … }**) **of** *element*_subtype_indication

Ο κανόνας αυτός ορίζει έναν τύπο πίνακα προσδιορίζοντας ένα ή περισσότερα εύρη δεικτών (τη λίστα από διακριτά εύρη – discrete ranges) και τον τύπο ή υποτύπο του στοιχείου. Θυμηθείτε από προηγούμενα κεφάλαια ότι ένα διακριτό εύρος είναι ένα υποσύνολο τιμών από ένα διακριτό τύπο (έναν τύπο ακεραίου ή τύπο απαρίθμησης), και μπορεί να προσδιοριστεί όπως φαίνεται στον απλοποιημένο συντακτικό κανόνα

discrete_range ⇐
discrete_subtype_indication
I simple_expression (**to** I **downto**) simple_expression

Θυμηθείτε επίσης ότι μια ένδειξη υποτύπου μπορεί να είναι απλά το όνομα ενός τύπου που έχει δηλωθεί προηγουμένως (μια ένδειξη τύπου) και μπορεί να περιλαμβάνει έναν περιορισμό εύρους για να περιορίσει το σύνολο τιμών από εκείνο τον τύπο, όπως παρουσιάζεται από τον απλουστευμένο κανόνα

subtype_indication ⇐
type_mark [**range** simple_expression (**to** I **downto**) simple_expression]

Επεξηγούμε αυτούς τους κανόνες για τον ορισμό πινάκων με μια σειρά από παραδείγματα. Αρχίζουμε με μονοδιάστατους πίνακες, στους οποίους υπάρχει μόνο ένα εύρος δείκτη. Εδώ είναι ένα απλό παράδειγμα για να ξεκινήσουμε, παρουσιάζοντας τη δήλωση ενός τύπου πίνακα που αναπαριστά λέξεις δεδομένων:

```
type word is array (0 to 31) of bit;
```

Κάθε στοιχείο είναι ένα bit, και τα στοιχεία δεικτοδοτούνται από 0 μέχρι 31. Μια εναλλακτική δήλωση ενός τύπου λέξης, πιο κατάλληλη για συστήματα «μικρού-άκρου» (little-endian), είναι

```
type word is array (31 downto 0) of bit;
```

Η διαφορά εδώ είναι ότι οι τιμές του δείκτη αρχίζουν από 31 για το αριστερότερο στοιχείο στις τιμές αυτού του τύπου και συνεχίζονται προς το 0 για το δεξιότερο στοιχείο. Οι τιμές του δείκτη ενός πίνακα δεν είναι απαραίτητο να είναι αριθμητικές. Για παράδειγμα, δοθέντος της δήλωσης ενός τύπου απαρίθμησης:

```
type controller_state is (initial, idle, active, error);
```

θα μπορούσαμε έπειτα να δηλώσουμε έναν πίνακα ως εξής:

```
type state_counts is array (idle to error) of natural;
```

Αυτό το είδος δήλωσης τύπου πίνακα στηρίζεται στον τύπο του εύρους δείκτη που γίνεται σαφής από τα συμφραζόμενα. Εάν υπήρχαν περισσότεροι τους ενός τύποι απαρίθμησης με τιμές idle και error, δεν θα ήταν σαφές ποιος θα χρησιμοποιηθεί για τον τύπο του δείκτη. Για να το καταστήσουμε σαφές, μπορούμε να χρησιμοποιήσουμε την εναλλακτική μορφή για τον καθορισμό του εύρους του δείκτη, στην οποία ονομάζουμε τον τύπο του δείκτη και συμπεριλαμβάνουμε έναν περιορισμό εύρους. Το προηγούμενο παράδειγμα θα μπορούσε να ξαναγραφεί ως εξής

```
type state_counts is
    array (controller_state range idle to error) of natural;
```

Εάν χρειαζόμαστε ένα στοιχείο πίνακα για κάθε τιμή σε έναν τύπο δείκτη, χρειάζεται μόνο να ονομάσουμε τον τύπο του δείκτη στη δήλωση του πίνακα χωρίς να καθορίσουμε το εύρος. Για παράδειγμα:

```
subtype coeff_ram_address is integer range 0 to 63;
type coeff_array is array (coeff_ram_address) of real;
```

Αφότου έχουμε δηλώσει έναν τύπο πίνακα, μπορούμε να ορίσουμε αντικείμενα αυτού του τύπου, συμπεριλαμβανομένων σταθερών, μεταβλητών και σημάτων. Για παράδειγμα, χρησιμοποιώντας τους τύπους που δηλώθηκαν παραπάνω, μπορούμε να δηλώσουμε μεταβλητές ως εξής:

```
variable buffer_register, data_register : word;
variable counters : state_counts;
variable coeff : coeff_array;
```

Κάθε ένα από αυτά τα αντικείμενα αποτελείται από τη συλλογή των στοιχείων που περιγράφονται από την αντίστοιχη δήλωση τύπου. Ένα μεμονωμένο στοιχείο μπορεί να χρησιμοποιηθεί σε μια παράσταση ή ως στόχος μιας ανάθεσης κάνοντας αναφορά στο αντικείμενο του πίνακα και δίνοντας μια τιμή δείκτη, για παράδειγμα:

```
coeff(0) := 0.0;
```

Εάν active είναι μια μεταβλητή του τύπου controller_state, μπορούμε να γράψουμε

```
counters(active) := counters(active) + 1;
```

Ένα αντικείμενο πίνακα μπορεί επίσης να χρησιμοποιηθεί ως ένα μοναδικό σύνθετο αντικείμενο. Για παράδειγμα, η ανάθεση

```
data_register := buffer_register;
```

αντιγράφει όλα τα στοιχεία του πίνακα buffer_register στα αντίστοιχα στοιχεία του πίνακα data_register.

ΠΑΡΑΔΕΙΓΜΑ

Η Εικόνα 4-1 είναι ένα μοντέλο για μια μνήμη που αποθηκεύει 64 συντελεστές πραγματικών αριθμών, που αρχικοποιούνται στο 0,0. Υποθέτουμε ότι ο τύπος coeff_ram_address έχει δηλωθεί προηγουμένως όπως παραπάνω. Το σώμα αρχιτεκτονικής περιέχει μια διεργασία με μια μεταβλητή πίνακα που αναπαριστά το χώρο αποθήκευσης των συντελεστών. Όταν η διεργασία αρχίζει, αρχικοποιεί τον πίνακα χρησιμοποιώντας ένα βρόχο for. Έπειτα κατά τρόπο επαναλαμβανόμενο περιμένει να αλλάξει οποιαδήποτε από τις θύρες εισόδου. Όταν το rd είναι '1', ο πίνακας δεικτοδοτείται χρησιμοποιώντας την τιμή της διεύθυνσης για να διαβάσει ένα συντελεστή. Όταν το wr είναι '1', η τιμή της διεύθυνσης χρησιμοποιείται για να επιλέξει ποιο συντελεστή να αλλάξει.

ΕΙΚΟΝΑ 4-1

```
entity coeff_ram is
    port ( rd, wr : in bit; addr : in coeff_ram_address;
           d_in : in real; d_out : out real );
end entity coeff_ram;
```

```
architecture abstract of coeff_ram is
begin
    memory : process is
        type coeff_array is array (coeff_ram_address) of real;
        variable coeff : coeff_array;
    begin
        for index in coeff_ram_address loop
            coeff(index) := 0.0;
        end loop;
        loop
            wait on rd, wr, addr, d_in;
            if rd = '1' then
                d_out <= coeff(addr);
            end if;
            if wr = '1' then
                coeff(addr) := d_in;
            end if;
        end loop;
    end process memory;
end architecture abstract;
```

Μια οντότητα και το σώμα αρχιτεκτονικής για μια λειτουργική μονάδα μνήμης που αποθηκεύει συντελεστές πραγματικών αριθμών. Η αποθήκευση μνήμης υλοποιείται με τη χρήση ενός πίνακα.

4.1.1 Πολυδιάστατοι Πίνακες

Η VHDL μας επιτρέπει επίσης να δημιουργήσουμε πολυδιάστατους πίνακες (multidimensional arrays), για παράδειγμα, για να αναπαραστήσουμε μήτρες ή πίνακες που δεικτοδοτούνται από περισσότερες από μια τιμές. Ένας τύπος πολυδιάστατου πίνακα δηλώνεται καθορίζοντας μία λίστα από εύρη δεικτών. Για παράδειγμα, θα μπορούσαμε να συμπεριλάβουμε τις ακόλουθες δηλώσεις τύπων σε ένα μοντέλο για μια μηχανή πεπερασμένων καταστάσεων (finite-state machine):

```
type symbol is ('a', 't', 'd', 'h', digit, cr, error);
type state is range 0 to 6;
type transition_matrix is array (state, symbol) of state;
```

Κάθε εύρος δείκτη μπορεί να καθοριστεί όπως είδαμε προηγουμένως για μονοδιάστατους πίνακες. Τα εύρη των δεικτών για κάθε διάσταση δεν χρειάζεται να είναι όλα από τον ίδιο τύπο, ούτε να έχουν την ίδια κατεύθυνση. Ένα αντικείμενο ενός τύπου πολυδιάστατου πίνακα δεικτοδοτείται γράφοντας μια λίστα από τιμές δεικτών για να επιλέξει ένα στοιχείο. Για παράδειγμα, εάν έχουμε μια μεταβλητή που έχει δηλωθεί ως εξής

```
variable transition_table : transition_matrix;
```

μπορούμε να τη δεικτοδοτήσουμε ως εξής:

```
transition_table(5, 'd');
```

ΠΑΡΑΔΕΙΓΜΑ

Στα τρισδιάστατα γραφικά, ένα σημείο στο χώρο μπορεί να αναπαρασταθεί χρησιμοποιώντας ένα διάνυσμα τριών-στοιχείων [x, y, z] των συντεταγμένων. Μετασχηματισμοί, όπως η αλλαγή κλίμακας (scaling), η περιστροφή (rotation) και η αντανάκλαση (reflection), μπορούν να γίνουν με τον πολλαπλασιασμό ενός διανύσματος με μια μήτρα μετασχηματισμού 3 × 3 από όπου προκύπτει ένα νέο διάνυσμα που αντιπροσωπεύει το μετασχηματισμένο σημείο. Μπορούμε να γράψουμε δηλώσεις τύπων της VHDL για τα σημεία και τις μήτρες μετασχηματισμού:

```
type point is array (1 to 3) of real;
type matrix is array (1 to 3, 1 to 3) of real;
```

Μπορούμε να χρησιμοποιήσουμε αυτούς τους τύπους για να δηλώσουμε τις μεταβλητές σημείου p και q και μια μεταβλητή μήτρας transform:

```
variable p, q : point;
variable transform : matrix;
```

Ο μετασχηματισμός μπορεί να εφαρμοστεί στο σημείο p για να παράγει ένα αποτέλεσμα στο q με τις ακόλουθες προτάσεις:

```
for i in 1 to 3 loop
    q(i) := 0.0;
    for j in 1 to 3 loop
        q(i) := q(i) + transform(i, j) * p(j);
    end loop;
end loop;
```

4.1.2 Συναθροίσεις Πίνακα

Έχουμε δει πώς μπορούμε να γράψουμε κυριολεκτικές τιμές για βαθμωτούς τύπους. Συχνά επίσης χρειάζεται να γράψουμε κυριολεκτικές τιμές για πίνακες, για παράδειγμα, για να αρχικοποιήσουμε μια μεταβλητή ή μια σταθερά τύπου πίνακα. Μπορούμε να κάνουμε κάτι τέτοιο χρησιμοποιώντας μια δομή της VHDL που καλείται *συνάθροιση πίνακα (array aggregate)*, σύμφωνα με το συντακτικό κανόνα

```
aggregate ⇐ ( ( [ choices => ] expression ) { , … } )
```

Ας εξετάσουμε πρώτα τη μορφή της συνάθροισης χωρίς το τμήμα των επιλογών (choices). Αποτελείται απλά από μια λίστα στοιχείων που εσωκλείονται σε παρενθέσεις, για παράδειγμα:

```
type point is array (1 to 3) of real;
constant origin : point := (0.0, 0.0, 0.0);
variable view_point : point := (10.0, 20.0, 0.0);
```

Αυτή η μορφή συνάθροισης πίνακα χρησιμοποιεί *συσχέτιση θέσης (positional association)* για να καθορίσει ποια τιμή στη λίστα αντιστοιχεί σε ποιο στοιχείο του πίνακα. Η πρώτη τιμή είναι το στοιχείο με τον αριστερότερο δείκτη, η δεύτερη είναι ο επόμενος δείκτης στα δεξιά, και τα λοιπά, μέχρι την τελευταία τιμή, η οποία είναι το στοιχείο με το

δεξιότερο δείκτη. Πρέπει να υπάρχει μία-προς-μία αντιστοιχία μεταξύ των τιμών της συνάθροισης και των στοιχείων του πίνακα.

Μια εναλλακτική μορφή της συνάθροισης χρησιμοποιεί *συσχέτιση ονόματος (named association)*, στην οποία η τιμή του δείκτη για κάθε στοιχείο γράφεται ρητά χρησιμοποιώντας το τμήμα των επιλογών που φαίνεται στο συντακτικό κανόνα. Οι επιλογές μπορούν να καθοριστούν με τον ίδιο ακριβώς τρόπο όπως οι επιλογές στις εναλλακτικές λύσεις μιας πρότασης case, που συζητήθηκαν στο Κεφάλαιο 3. Σαν υπενθύμιση, εδώ είναι ο συντακτικός κανόνας για τις επιλογές:

choices ⇐ (simple_expression I discrete_range I **others**) { | … }

Για παράδειγμα, η δήλωση και η αρχικοποίηση της μεταβλητής θα μπορούσε να ξαναγραφτεί ως εξής

variable view_point : point := (1 => 10.0, 2 => 20.0, 3 => 0.0);

Το κύριο πλεονέκτημα της συσχέτισης ονόματος είναι ότι μας δίνει περισσότερη ευελιξία στο γράψιμο των συναθροίσεων για μεγαλύτερους πίνακες. Για να το επεξηγήσουμε, ας επιστρέψουμε στη μνήμη των συντελεστών που περιγράφτηκε παραπάνω. Η δήλωση τύπου ήταν

type coeff_array **is array** (coeff_ram_address) **of** real;

Υποθέστε ότι θέλουμε να δηλώσουμε τη μεταβλητή της μνήμης των συντελεστών, να αρχικοποιήσουμε τις πρώτες θέσεις σε κάποια τιμή διαφορετική από το μηδέν και τις υπόλοιπες στο μηδέν. Ακολουθούν διάφοροι τρόποι γραφής των συναθροίσεων οι οποίοι έχουν όλοι το ίδιο αποτέλεσμα:

variable coeff : coeff_array := (0 => 1.6, 1 => 2.3, 2 => 1.6, 3 **to** 63 => 0.0);

Εδώ χρησιμοποιούμε μια προδιαγραφή εύρους για να αρχικοποιήσουμε το μεγαλύτερο όγκο των τιμών του πίνακα στο μηδέν.

variable coeff : coeff_array := (0 => 1.6, 1 => 2.3, 2 => 1.6, **others** => 0.0);

Η λέξη-κλειδί **others** αντιπροσωπεύει οποιαδήποτε τιμή δείκτη δεν έχει αναφερθεί προηγουμένως στη συνάθροιση. Εάν η λέξη-κλειδί **others** χρησιμοποιείται, πρέπει να είναι η τελευταία επιλογή στη συνάθροιση.

variable coeff : coeff_array := (0 | 2 => 1.6, 1 => 2.3, **others** => 0.0);

Το σύμβολο "|" μπορεί να χρησιμοποιηθεί για να διαχωρίσει μια λίστα από τιμές δεικτών, όπου όλα τα στοιχεία έχουν την ίδια τιμή.

Σημειώστε ότι δεν επιτρέπεται να αναμίξουμε συσχέτιση ονόματος και συσχέτιση θέσης σε μια συνάθροιση πίνακα, εκτός από τη χρήση της επιλογή **others** στην τελική θέση. Κατά συνέπεια, η παρακάτω συνάθροιση δεν είναι νόμιμη:

variable coeff : coeff_array := (1.6, 2.3, 2 => 1.6, **others** => 0.0); -- μη νόμιμη

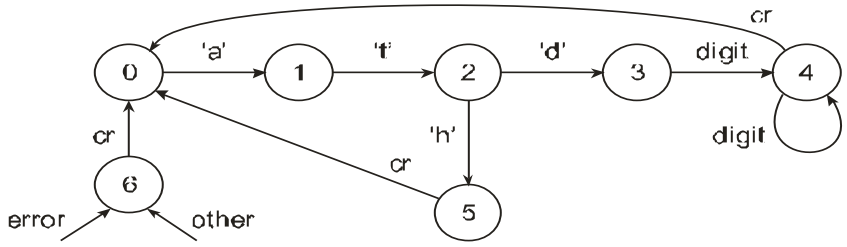
Μπορούμε επίσης να χρησιμοποιήσουμε συναθροίσεις για να γράψουμε τις τιμές πολυδιάστατων πινάκων. Σε αυτήν την περίπτωση, μεταχειριζόμαστε τον πίνακα σαν ήταν ένας πίνακας από πίνακες, γράφοντας πρώτα μια συνάθροιση πίνακα για κάθε μια από τις αριστερότερες τιμές των δεικτών.

ΠΑΡΑΔΕΙΓΜΑ

Μπορούμε να χρησιμοποιήσουμε ένα δισδιάστατο πίνακα για να αναπαραστήσουμε τη μήτρα μετάβασης (transition matrix) μιας μηχανής πεπερασμένων καταστάσεων (finite-state machine, FSM) που ερμηνεύει απλές εντολές ενός μόντεμ (modem). Μια εντολή πρέπει να αποτελείται από την ακολουθία χαρακτήρων "atd" ακολουθούμενη από μια ακολουθία ψηφίων και ένα χαρακτήρα cr, ή από την ακολουθία χαρακτήρων "ath" ακολουθούμενη από cr. Το διάγραμμα μετάβασης καταστάσεων (state transition diagram) παρουσιάζεται στην Εικόνα 4-2. Το σύμβολο "other" αναπαριστά ένα χαρακτήρα διαφορετικό από 'a', 't', 'd', 'h', ένα ψηφίο ή cr. Το περίγραμμα μιας διεργασίας που υλοποιεί τη μηχανή πεπερασμένων καταστάσεων παρουσιάζεται στην Εικόνα 4-3.

Οι δηλώσεις των τύπων symbol και state αναπαριστούν τα σύμβολα εντολής και τις καταστάσεις για τη μηχανή πεπερασμένων καταστάσεων. Η μήτρα μετάβασης, next_state, είναι μια σταθερά δισδιάστατου πίνακα που δεικτοδοτείται από τους τύπους κατάστασης (state) και συμβόλου (symbol). Ένα στοιχείο στη θέση (*i, j*) σε αυτήν τη μήτρα υποδηλώνει την επόμενη κατάσταση της μηχανής όταν αυτή είναι στην κατάσταση *i* και το επόμενο σύμβολο εισόδου είναι *j*. Η μήτρα αρχικοποιείται σύμφωνα με το διάγραμμα μετάβασης. Η διεργασία χρησιμοποιεί τη μεταβλητή current_state και διαδοχικά σύμβολα εισόδου σαν δείκτες στη μήτρα μετάβασης για να καθορίσει την επόμενη κατάσταση. Για κάθε μετάβαση, εκτελεί κάποιες ενέργειες βασισμένη στη νέα κατάσταση. Οι ενέργειες υλοποιούνται μέσα στην πρόταση case.

ΕΙΚΟΝΑ 4-2



Το διάγραμμα μετάβασης καταστάσεων για μια μηχανή πεπερασμένων καταστάσεων των εντολών ενός μόντεμ. Η κατάσταση 0 είναι η αρχική κατάσταση. Η μηχανή επιστρέφει σε αυτήν την κατάσταση αφότου έχει αναγνωρίσει μια σωστή εντολή. Η κατάσταση 6 είναι η κατάσταση λάθους, στην οποία η μηχανή πηγαίνει εάν ανιχνεύσει έναν παράνομο ή μη προβλεπόμενο χαρακτήρα.

ΕΙΚΟΝΑ 4-3

```
modem_controller : process is
    type symbol is ('a', 't', 'd', 'h', digit, cr, other);
    type symbol_string is array (1 to 20) of symbol;
    type state is range 0 to 6;
    type transition_matrix is array (state, symbol) of state;
    constant next_state : transition_matrix :=
       ( 0 => ('a' => 1, others => 6),
         1 => ('t' => 2, others => 6),
         2 => ('d' => 3, 'h' => 5, others => 6),
         3 => (digit => 4, others => 6),
         4 => (digit => 4, cr => 0, others => 6),
         5 => (cr => 0, others => 6),
         6 => (cr => 0, others => 6) );
    variable command : symbol_string;
    variable current_state : state := 0;
begin
    …
    for index in 1 to 20 loop
        current_state := next_state( current_state, command(index) );
        case current_state is
            …
        end case;
    end loop;
    …
end process modem_controller;
```

Το περίγραμμα μιας διεργασίας που υλοποιεί τη μηχανή πεπερασμένων καταστάσεων ώστε να δέχεται τις εντολές ενός μόντεμ.

Ένα άλλο μέρος στο οποίο μπορούμε να χρησιμοποιήσουμε μια συνάθροιση είναι ο στόχος μιας ανάθεσης μεταβλητής ή μιας ανάθεσης σήματος. Ο πλήρης συντακτικός κανόνας για μια πρόταση ανάθεσης μεταβλητής είναι

variable_assignment_statement ⇐

[label :] (name I aggregate) := expression ;

Εάν ο στόχος είναι μια συνάθροιση, πρέπει να περιέχει ένα όνομα μεταβλητής σε κάθε θέση στοιχείου. Επιπλέον, η παράσταση στη δεξιά πλευρά της ανάθεσης πρέπει να παράγει μια σύνθετη τιμή του ίδιου τύπου με τη συνάθροιση του στόχου. Κάθε στοιχείο της δεξιάς πλευράς ανατίθεται στην αντίστοιχη μεταβλητή στη συνάθροιση του στόχου. Ο πλήρης συντακτικός κανόνας για μια ανάθεση σήματος επιτρέπει επίσης στο στόχο να είναι υπό μορφή συνάθροισης, με κάθε στοιχείο να είναι ένα όνομα σήματος. Μπορούμε να χρησιμοποιήσουμε τις αναθέσεις αυτής της μορφής για να διασπάσουμε μια σύνθετη τιμή σε έναν αριθμό βαθμωτών σημάτων. Για παράδειγμα, εάν έχουμε μια μεταβλητή flag_reg, η οποία είναι ένα διάνυσμα bit (bit vector) τεσσάρων-στοιχείων, μπορούμε να εκτελέσουμε την ακόλουθη ανάθεση σήματος σε τέσσερα σήματα τύπου bit:

(z_flag, n_flag, v_flag, c_flag) <= flag_reg;

Δεδομένου ότι η δεξιά πλευρά είναι ένα διάνυσμα bit, ο στόχος λαμβάνεται ως μία συνάθροιση διανύσματος bit. Το αριστερότερο στοιχείο του flag_reg ανατίθεται στο z_flag, το δεύτερο στοιχείο του flag_reg ανατίθεται στο n_flag, και τα λοιπά. Αυτή η μορφή πολλαπλής ανάθεσης είναι πιο εύχρηστη από το να γράψουμε τέσσερις χωριστές προτάσεις ανάθεσης.

4.1.3 Ιδιότητες Πινάκων

Στο Κεφάλαιο 2 είδαμε ότι μπορούμε να χρησιμοποιήσουμε ιδιότητες για να αναφερθούμε σε πληροφορίες σχετικές με βαθμωτούς τύπους. Υπάρχουν επίσης ιδιότητες που εφαρμόζονται σε τύπους πινάκων και αναφέρονται σε πληροφορίες σχετικές με το εύρος των δεικτών. Οι ιδιότητες πινάκων μπορούν επίσης να εφαρμοστούν σε αντικείμενα πινάκων, όπως σταθερές, μεταβλητές και σήματα, για να αναφερθούν σε πληροφορίες σχετικές με τους τύπους των αντικειμένων. Λαμβάνοντας υπόψιν κάποιον τύπο ή αντικείμενο πίνακα A, και έναν ακέραιο αριθμό N μεταξύ του 1 και του αριθμού των διαστάσεων του A, η VHDL ορίζει τις ακόλουθες ιδιότητες:

A'left(N)	Αριστερό όριο του εύρους του δείκτη της διάστασης N του A
A'right(N)	Δεξί όριο του εύρους του δείκτη της διάστασης N του A
A'low(N)	Κάτω όριο του εύρους του δείκτη της διάστασης N του A
A'high(N)	Άνω όριο του εύρους του δείκτη της διάστασης N του A
A'range(N)	Εύρος του δείκτη της διάστασης N του A
A'reverse_range(N)	Αντίστροφο εύρος του δείκτη της διάστασης N του A
A'length(N)	Μήκος του εύρους του δείκτη της διάστασης N του A
A'ascending(N)	true (αληθές) εάν το εύρος του δείκτη της διάστασης N του A είναι μια αύξουσα σειρά, διαφορετικά false (ψευδές)

Για παράδειγμα, δοθέντος της δήλωσης πίνακα

```
type A is array (1 to 4, 31 downto 0) of boolean;
```

μερικές τιμές ιδιοτήτων είναι

A'left(1) = 1	A'low(1) = 1
A'right(2) = 0	A'high(2) = 31
A'range(1) = 1 **to** 4	A'reverse_range(2) = 0 **to** 31
A'length(1) = 4	A'length(2) = 32
A'ascending(1) = true	A'ascending(2) = false

Για όλες αυτές τις ιδιότητες, για να αναφερθούμε στην πρώτη διάσταση (ή εάν υπάρχει μόνο μια διάσταση), μπορούμε να παραλείψουμε τον αριθμό διάστασης στις παρενθέσεις, για παράδειγμα:

A'low = 1	A'length = 4

Στην επόμενη ενότητα, βλέπουμε πώς αυτές οι ιδιότητες πινάκων μπορούν να χρησιμοποιηθούν για να μεταχειριστούν θύρες πίνακα. Μια άλλη σημαντική χρήση είναι στην εγγραφή βρόχων for που επαναλαμβάνουν την εκτέλεσή τους στα στοιχεία ενός πίνακα. Για παράδειγμα, δοθέντος μιας μεταβλητής πίνακα free_map που είναι ένα πίνακας από bit, μπορούμε να γράψουμε ένα βρόχο for για να μετρήσουμε τον αριθμό των bit ‘1’ χωρίς να γνωρίζουμε το πραγματικό μέγεθος του πίνακα:

```
count := 0;
for index in free_map'range loop
    if free_map(index) = '1' then
        count := count + 1;
    end if;
end loop;
```

Οι ιδιότητες 'range και 'reverse_range μπορούν να χρησιμοποιηθούν σε οποιοδήποτε σημείο ενός VHDL μοντέλου όπου απαιτείται η προδιαγραφή ενός εύρους, ως εναλλακτική λύση για να καθορίσουμε το αριστερό και το δεξί όριο και την κατεύθυνση του εύρους. Κατά συνέπεια, μπορούμε να χρησιμοποιήσουμε τις ιδιότητες στους ορισμούς τύπων και υποτύπων, στους περιορισμούς υποτύπων, στις προδιαγραφές παραμέτρου βρόχων for, στις προτάσεις case και τα λοιπά. Το πλεονέκτημα της χρήσης αυτής της μεθόδου είναι ότι μπορούμε να καθορίσουμε το μέγεθος του πίνακα σε ένα σημείο του μοντέλου και σε όλα τα άλλα σημεία να χρησιμοποιήσουμε ιδιότητες πινάκων. Εάν πρέπει αργότερα να αλλάξουμε το μέγεθος του πίνακα για κάποιους λόγους, χρειάζεται μόνο να αλλάξουμε το μοντέλο σε ένα σημείο.

VHDL-87

Η ιδιότητα πινάκων ‘ascending δεν παρέχεται στη VHDL-87.

4.2 Τύποι Πινάκων χωρίς Περιορισμούς

Οι τύποι πινάκων που έχουμε δει μέχρι τώρα σε αυτό το κεφάλαιο καλούνται πίνακες *με περιορισμούς* (*constrained* arrays), δεδομένου ότι ο ορισμός του τύπου περιορίζει τις τιμές των δεικτών να είναι εντός ενός καθορισμένου εύρους. Η VHDL μας επιτρέπει επίσης να ορίσουμε τύπους πινάκων *χωρίς περιορισμούς* (*unconstrained* arrays), στους οποίους απλά υποδηλώνουμε τον τύπο των τιμών των δεικτών, χωρίς να διευκρινίζουμε τα όρια τους. Ένας ορισμός τύπου πίνακα χωρίς περιορισμούς περιγράφεται από τον εναλλακτικό συντακτικό κανόνα

array_type_definition ⇐

array ((type_mark **range** <>) { , … })

of *element*_subtype_indication

Το σύμβολο "<>", αποκαλούμενο συχνά ως «κουτί», μπορεί να θεωρηθεί ως μέρος τοποθέτησης του εύρους του δείκτη, το οποίο συμπληρώνεται αργότερα όταν χρησιμοποιείται ο τύπος. Ένα παράδειγμα μιας δήλωσης τύπου πίνακα χωρίς περιορισμούς είναι

```
type sample is array (natural range <>) of integer;
```

Ένα σημαντικό σημείο που πρέπει να κατανοήσουμε για τους τύπους πινάκων χωρίς περιορισμούς είναι ότι όταν δηλώνουμε ένα αντικείμενο ενός τέτοιου τύπου, πρέπει να παρέχουμε έναν περιορισμό ο οποίος να διευκρινίζει τα όρια του δείκτη. Μπορούμε να το κάνουμε αυτό με διάφορους τρόπους. Ένας τρόπος είναι να παρέχουμε τον περιορισμό όταν δημιουργείται ένα αντικείμενο, για παράδειγμα:

```
variable short_sample_buf : sample(0 to 63);
```

Αυτό υποδηλώνει ότι οι τιμές του δείκτη για τη μεταβλητή short_sample_buf είναι φυσικοί αριθμοί εντός της αύξουσας σειράς 0 έως 63. Ένας άλλος τρόπος να διευκρινίσουμε τον περιορισμό του εύρους είναι να δηλωθεί ένας υποτύπος του τύπου πίνακα χωρίς περιορισμούς. Τα αντικείμενα μπορούν έπειτα να δημιουργηθούν χρησιμοποιώντας αυτόν τον υποτύπο, για παράδειγμα:

```
subtype long_sample is sample(0 to 255);
variable new_sample_buf, old_sample_buf : long_sample;
```

Αυτά είναι και τα δύο παραδείγματα μιας νέας μορφής ένδειξης υποτύπου (subtype_indication) που δεν έχουμε δει ακόμα. Ο συντακτικός κανόνας είναι

subtype_indication ⇐ type_mark [(discrete_range { , … })]

Το σημάδι τύπου (type_mark) είναι το όνομα του τύπου πίνακα χωρίς περιορισμούς, και οι προδιαγραφές βαθμωτού εύρους περιορίζουν τον τύπο του δείκτη σε ένα υποσύνολο των τιμών που χρησιμοποιούνται για την δεικτοδότηση των στοιχείων του πίνακα. Κάθε βαθμωτό εύρος πρέπει να είναι του ίδιου τύπου με τον αντίστοιχο τύπο δείκτη.

Όταν δηλώνουμε μια σταθερά ενός τύπου πίνακα χωρίς περιορισμούς, υπάρχει ένας τρίτος τρόπος με τον οποίο μπορούμε να παρέχουμε έναν περιορισμό εύρους. Μπορούμε να τον συμπεράνουμε από την παράσταση που χρησιμοποιείται για να αρχικοποιήσει τη σταθερά. Εάν η παράσταση αρχικοποίησης είναι μια συνάθροιση πίνακα γραμμένη με συσχέτιση ονόματος, οι τιμές του δείκτη στη συνάθροιση υπονοούν το εύρος του δείκτη της σταθεράς. Για παράδειγμα, στη δήλωση σταθεράς

```
constant lookup_table : sample := ( 1 => 23, 3 => –16, 2 => 100, 4 => 11);
```

το εύρος του δείκτη είναι 1 έως 4.

Εάν η παράσταση είναι μια συνάθροιση που χρησιμοποιεί συσχέτιση θέσης, η τιμή του δείκτη του πρώτου στοιχείου υποτίθεται ότι είναι η αριστερότερη τιμή στον υποτύπο πίνακα. Για παράδειγμα, στη δήλωση σταθεράς

```
constant beep_sample : sample := ( 127, 63, 0, –63, –127, –63, 0, 63 );
```

το εύρος του δείκτη είναι 0 έως 7, δεδομένου ότι ο υποτύπος του δείκτη είναι natural. Η κατεύθυνση του δείκτη είναι αύξουσα, δεδομένου ότι ο τύπος natural έχει οριστεί ως αύξουσα σειρά.

4.2.1 Αλφαριθμητικά

Η VHDL παρέχει έναν προκαθορισμένο τύπο πίνακα χωρίς περιορισμούς που καλείται string, και δηλώνεται ως εξής

```
type string is array (positive range <>) of character;
```

Σε γενικές γραμμές το εύρος του δείκτη για ένα αλφαριθμητικό με περιορισμούς (constrained string) μπορεί να είναι είτε αύξουσα είτε φθίνουσα σειρά, με οποιουσδήποτε θετικούς ακέραιους αριθμούς για τα όρια του δείκτη. Εντούτοις, οι περισσότερες εφαρμογές χρησιμοποιούν απλά μια αύξουσα σειρά που αρχίζει από το 1. Για παράδειγμα:

```
constant LCD_display_len : positive := 20;
subtype LCD_display_string is string(1 to LCD_display_len);
variable LCD_display : LCD_display_string := (others => ' ');
```

4.2.2 Διανύσματα Bit

Η VHDL παρέχει επίσης έναν προκαθορισμένο τύπο πίνακα χωρίς περιορισμούς που καλείται bit_vector, και δηλώνεται ως εξής

```
type bit_vector is array (natural range <>) of bit;
```

Αυτός ο τύπος μπορεί να χρησιμοποιηθεί για να αναπαραστήσει λέξεις δεδομένων στο επίπεδο αρχιτεκτονικής της μοντελοποίησης. Για παράδειγμα, υποτύποι για την αναπαράσταση ψηφιολέξεων σε έναν επεξεργαστή «μικρού-άκρου» (little-endian processor) θα μπορούσαν να δηλωθούν ως εξής

```
subtype byte is bit_vector(7 downto 0);
```

Εναλλακτικά, μπορούμε να παρέχουμε τον περιορισμό εύρους όταν δηλώνουμε ένα αντικείμενο, για παράδειγμα:

```
variable channel_busy_register : bit_vector(1 to 4);
```

4.2.3 Πίνακες Πρότυπης Λογικής

Το πακέτο πρότυπης λογικής std_logic_1164 παρέχει έναν τύπο πίνακα χωρίς περιορισμούς για διανύσματα με τιμές της πρότυπης λογικής. Δηλώνεται ως εξής

```
type std_ulogic_vector is array ( natural range <> ) of std_ulogic;
```

Αυτός ο τύπος μπορεί να χρησιμοποιηθεί με έναν τρόπο παρόμοιο με τα διανύσματα bit, αλλά παρέχει περισσότερη λεπτομέρεια στην αναπαράσταση των ηλεκτρικών επιπέδων που χρησιμοποιούνται σε μια σχεδίαση. Μπορούμε να ορίσουμε υποτύπους του τύπου διανύσματος πρότυπης λογικής, για παράδειγμα:

```
subtype std_ulogic_word is std_ulogic_vector(0 to 31);
```

Ή μπορούμε άμεσα να δημιουργήσουμε ένα αντικείμενο του τύπου διανύσματος πρότυπης λογικής:

```
signal csr_offset : std_ulogic_vector(2 downto 1);
```

4.2.4 Κυριολεκτικά Αλφαριθμητικών και Ψηφιοσειρών

Στο Κεφάλαιο 1, είδαμε ότι ένα αλφαριθμητικό κυριολεκτικό μπορεί να χρησιμοποιηθεί για να γράψουμε μια τιμή που αναπαριστά μια ακολουθία χαρακτήρων. Μπορούμε να χρησιμοποιήσουμε ένα αλφαριθμητικό κυριολεκτικό αντί μιας συνάθροισης πίνακα για μια τιμή τύπου string. Για παράδειγμα, μπορούμε να αρχικοποιήσουμε μια σταθερά αλφαριθμητικού ως εξής:

```
constant ready_message : string := "Ready ";
```

Μπορούμε επίσης να χρησιμοποιήσουμε αλφαριθμητικά κυριολεκτικά για οποιοδήποτε άλλο τύπο μονοδιάστατου πίνακα του οποίου τα στοιχεία είναι ενός τύπου απαρίθμησης που περιλαμβάνει χαρακτήρες. Ο τύπος πίνακα της IEEE πρότυπης λογικής std_ulogic_vector είναι ένα παράδειγμα. Κατά συνέπεια θα μπορούσαμε να δηλώσουμε και να αρχικοποιήσουμε μια μεταβλητή ως εξής:

```
variable current_test : std_ulogic_vector(0 to 13) := "ZZZZZZZZZZ----";
```

Στο Κεφάλαιο 1 είδαμε επίσης τα κυριολεκτικά ψηφιοσειρών ως έναν τρόπο για να γράψουμε μια ακολουθία τιμών bit. Οι ψηφιοσειρές μπορούν να χρησιμοποιηθούν αντί των συναθροίσεων πίνακα για να γράψουμε τιμές τύπου διανύσματος bit. Για παράδειγμα, η μεταβλητή channel_busy_register που ορίστηκε παραπάνω μπορεί να αρχικοποιηθεί με μια ανάθεση:

```
channel_busy_register := b"0000";
```

Μπορούμε επίσης να χρησιμοποιήσουμε κυριολεκτικά ψηφιοσειρών για άλλους τύπους μονοδιάστατων πινάκων των οποίων τα στοιχεία είναι ενός τύπου απαρίθμησης που περιλαμβάνει τους χαρακτήρες ‘0’ και ‘1’. Κάθε χαρακτήρας στο κυριολεκτικό ψηφιοσειράς αναπαριστά ένα, τρία ή τέσσερα διαδοχικά στοιχεία της τιμής του πίνακα, ανάλογα με το εάν η βάση που καθορίζεται στο κυριολεκτικό είναι δυαδική, οκταδική ή δεκαεξαδική. Πάλι, χρησιμοποιώντας ως παράδειγμα τον τύπο std_ulogic_vector, μπορούμε να γράψουμε μια δήλωση σταθεράς χρησιμοποιώντας ένα κυριολεκτικό ψηφιοσειράς:

```
constant all_ones : std_ulogic_vector(15 downto 0) := X"FFFF";
```

VHDL-87

Στη VHDL-87, τα κυριολεκτικά ψηφιοσειρών μπορούν να χρησιμοποιηθούν ως κυριολεκτικά μόνο για τους τύπους πινάκων στους οποίους τα στοιχεία είναι τύπου bit. Ο προκαθορισμένος τύπος bit_vector είναι ένας τέτοιος τύπος. Αντίθετα, ο τύπος πρότυπης λογικής std_ulogic_vector δεν είναι. Για τύπους πινάκων, όπως το std_ulogic_vector, μπορούμε να χρησιμοποιήσουμε αλφαριθμητικά κυριολεκτικά.

4.2.5 Θύρες Πίνακα χωρίς Περιορισμούς

Μια σημαντική χρήση ενός τύπου πίνακα χωρίς περιορισμούς είναι να καθορίσει τον τύπο μιας θύρας πίνακα. Αυτή η χρήση μας επιτρέπει να γράψουμε τη διασύνδεση μιας οντότητας με ένα γενικό τρόπο, έτσι ώστε να μπορεί να συνδεθεί με σήματα πίνακα οποιουδήποτε μεγέθους ή με οποιοδήποτε εύρος τιμών των δεικτών. Όταν δημιουργήσουμε ένα στιγμιότυπο της οντότητας, τα όρια των δεικτών του σήματος πίνακα που συνδέεται με τη θύρα χρησιμοποιούνται ως όρια της θύρας.

ΠΑΡΑΔΕΙΓΜΑ

Υποθέστε ότι επιθυμούμε να μοντελοποιήσουμε μια οικογένεια από πύλες and (λογικό ΚΑΙ), κάθε μια με ένα διαφορετικό αριθμό εισόδων. Δηλώνουμε τη διασύνδεση της οντότητας όπως φαίνεται στην Εικόνα 4-4. Η θύρα εισόδου είναι του τύπου bit_vector χωρίς περιορισμούς. Το σώμα αρχιτεκτονικής περιλαμβάνει μια διεργασία που είναι ευαίσθητη στις αλλαγές στη θύρα εισόδου. Όταν οποιοδήποτε στοιχείο αλλάζει, η διεργασία εκτελεί μια λογική λειτουργία and στα στοιχεία του πίνακα εισόδου. Χρησιμοποιεί την ιδιότητα ‘range για να καθορίσει το εύρος του δείκτη του πίνακα, εφόσον το εύρος του δείκτη δεν είναι γνωστό έως ότου δημιουργηθεί το στιγμιότυπο της οντότητας.

ΕΙΚΟΝΑ 4-4

```
entity and_multiple is
    port ( i : in bit_vector; y : out bit );
end entity and_multiple;
```

```
architecture behavioral of and_multiple is
begin
    and_reducer : process ( i ) is
        variable result : bit;
    begin
        result := '1';
        for index in i'range loop
            result := result and i(index);
        end loop;
        y <= result;
    end process and_reducer;
end architecture behavioral;
```

Μια οντότητα και το σώμα αρχιτεκτονικής για μια πύλη and με μια θύρα εισόδου πίνακα χωρίς περιορισμούς.

Για να εξηγήσουμε τη χρήση της οντότητας της πύλης πολλαπλών-εισόδων, ας υποθέσουμε ότι έχουμε τα ακόλουθα σήματα:

```
signal count_value : bit_vector(7 downto 0);
signal terminal_count : bit;
```

Δημιουργούμε ένα στιγμιότυπο της οντότητας, συνδέοντας τη θύρα εισόδου της με το σήμα διανύσματος bit:

```
tc_gate : entity work.and_multiple(behavioral)
    port map ( i => count_value, y => terminal_count);
```

Για αυτό το στιγμιότυπο, η θύρα εισόδου περιορίζεται από το εύρος του δείκτη του σήματος. Το στιγμιότυπο συμπεριφέρεται ως πύλη and οκτώ-εισόδων.

4.3 Λειτουργίες και Αναφορές Πινάκων

Αν και ένας πίνακας είναι μια συλλογή τιμών, ξοδεύουμε μεγάλο μέρος του χρόνου ενεργώντας σε ένα στοιχείο του πίνακα κάθε φορά, χρησιμοποιώντας τους τελεστές που περιγράφονται στο Κεφάλαιο 2. Εντούτοις, εάν χειριζόμαστε μονοδιάστατους πίνακες βαθμωτών τιμών, μπορούμε να χρησιμοποιήσουμε μερικούς από τους τελεστές για να λειτουργήσουμε σε ολόκληρους τους πίνακες, συνδυάζοντας τα στοιχεία ανά ζευγάρια.

Πρώτον, οι λογικοί τελεστές (**and**, **or**, **nand**, **nor**, **xor** και **xnor**) μπορούν να εφαρμοστούν σε δύο μονοδιάστατους πίνακες με στοιχεία bit ή Boolean. Οι τελεστέοι πρέπει να είναι του ίδιου μήκους και τύπου, και το αποτέλεσμα υπολογίζεται εφαρμόζοντας τον τελεστή στα στοιχεία που αντιστοιχούν από κάθε πίνακα για να παράγει έναν πίνακα του ίδιου μήκους. Τα στοιχεία αντιστοιχίζονται αρχίζοντας από την αριστερότερη θέση σε κάθε πίνακα. Ένα στοιχείο σε μια δεδομένη θέση από τα αριστερά σε έναν πίνακα αντιστοιχίζεται με το στοιχείο στην ίδια θέση από τα αριστερά στον άλλο πίνακα. Ο τελεστής **not** μπορεί επίσης να εφαρμοστεί σε ένα μονό πίνακα με στοιχεία τύπου bit ή Boolean, με το αποτέλεσμα να είναι ένας πίνακας του ίδιου μήκους και τύπου με τον τελεστέο. Οι ακόλουθες δηλώσεις και προτάσεις επεξηγούν αυτήν τη χρήση των λογικών τελεστών όταν εφαρμόζονται σε διανύσματα bit:

```
subtype pixel_row is bit_vector (0 to 15);
variable current_row, mask : pixel_row;
current_row := current_row and not mask;
current_row := current_row xor X"FFFF";
```

Δεύτερον, οι τελεστές ολίσθησης (shift operators) που εισάγονται στο Κεφάλαιο 2 (**sll**, **srl**, **sla**, **sra**, **rol** και **ror**) μπορούν να χρησιμοποιηθούν με ένα μονοδιάστατο πίνακα με τιμές bit ή Boolean σαν αριστερό τελεστέο και έναν ακέραιο αριθμό σαν δεξί τελεστέο. Μια λειτουργία λογικής αριστερής ολίσθησης ολισθαίνει τα στοιχεία του πίνακα *n* θέσεις στα αριστερά (με το *n* να είναι ο δεξιός τελεστέος), συμπληρώνοντας τις θέσεις που άδειασαν με '0' ή false και «πετώντας» τα *n* αριστερότερα στοιχεία. Εάν ο ακέραιος *n* είναι αρνητικός, τα στοιχεία μετατοπίζονται αντ' αυτού προς τα δεξιά. Μερικά παραδείγματα είναι

```
B"10001010" sll 3 = B"01010000"    B"10001010" sll –2 = B"00100010"
```

Η λειτουργία λογικής δεξιάς ολίσθησης ολισθαίνει ομοίως τα στοιχεία *n* θέσεις στα δεξιά για θετικό *n*, ή στα αριστερά για αρνητικό *n*, για παράδειγμα:

```
B"10010111" srl 2 = B"00100101"    B"10010111" srl –6 = B"11000000"
```

Οι επόμενες δύο πράξεις ολίσθησης, αριθμητική αριστερή ολίσθηση και αριθμητική δεξιά ολίσθηση, λειτουργούν όμοια, αλλά αντί να συμπληρώνουν τις θέσεις που αδειάζουν με '0' ή false, τις γεμίζουν με ένα αντίγραφο του στοιχείου που βρίσκεται στην άκρη που αδειάζει, για παράδειγμα:

```
B"01001011" sra 3 = B"00001001"    B"10010111" sra 3 = B"11110010"
B"00001100" sla 2 = B"00110000"    B"00010001" sla 2 = B"01000111"
```

Όπως με τις λογικές ολισθήσεις, εάν ο ακέραιος *n* είναι αρνητικός, η ολίσθηση πραγματοποιείται προς την αντίθετη κατεύθυνση, για παράδειγμα:

```
B"00010001" sra –2 = B"01000111"    B"00110000" sla –2 = B"00001100"
```

Μια πράξη αριστερής περιστροφής μετακινεί τα στοιχεία του πίνακα *n* θέσεις προς τα αριστερά, μεταφέροντας τα *n* στοιχεία από το αριστερό άκρο του πίνακα πίσω στις θέσεις που άδειασαν στο δεξιό άκρο. Μια πράξη δεξιάς περιστροφής κάνει το ίδιο πράγμα, αλλά στην αντίθετη κατεύθυνση. Όπως με τις πράξεις ολίσθησης, ένα αρνητικό δεξί όρισμα αντιστρέφει την κατεύθυνση της περιστροφής. Μερικά παραδείγματα είναι

```
B"10010011" rol 1 = B"00100111"    B"10010011" ror 1 = B"11001001"
```

Οι σχεσιακοί τελεστές συγκροτούν την τρίτη ομάδα πράξεων που μπορούν να εφαρμοστούν στους μονοδιάστατους πίνακες. Τα στοιχεία των πινάκων μπορούν να είναι οποιουδήποτε διακριτού τύπου. Οι δύο τελεστέοι δεν χρειάζονται να είναι του ίδιου μήκους, εφ' όσον έχουν τον ίδιο τύπο στοιχείων. Ο τρόπος που αυτοί οι τελεστές λειτουργούν μπορεί να φανεί ευκολότερα όταν εφαρμόζονται σε ακολουθίες χαρακτήρων, οπότε σ'αυτήν την περίπτωση συγκρίνονται σύμφωνα με τη διάταξη λεξικών με διάκριση πεζών/κεφαλαίων.

Για να δούμε πώς η σύγκριση λεξικού μπορεί να γενικευτεί σε μονοδιάστατους πίνακες με άλλους τύπους στοιχείων, ας θεωρήσουμε ότι ο τελεστής "<" εφαρμόζεται σε δύο πίνακες a και b. Εάν και οι δύο πίνακες a και b έχουν μήκος 0, τότε a < b είναι ψευδές. Εάν ο a έχει μήκος 0, και ο b έχει μη-μηδενικό μήκος, τότε a < b. Εναλλακτικά, εάν και οι δύο πίνακες a και b έχουν μη-μηδενικό μήκος, τότε a < b εάν a(1) < b(1), ή εάν a(1) = b(1) και το υπόλοιπο του a < από το υπόλοιπο του b. Στην εναπομείνασα περίπτωση, όπου ο a έχει μη-μηδενικό μήκος και ο b έχει μήκος 0, τότε a < b είναι ψευδές. Η σύγκριση που χρησιμοποιεί τους άλλους σχεσιακούς τελεστές εκτελείται ανάλογα.

Ένας ακόμα τελεστής που μπορεί να εφαρμοστεί στους μονοδιάστατους πίνακες είναι ο τελεστής συνένωσης ("&"), ο οποίος ενώνει δύο τιμές πινάκων άκρο προς άκρο. Για παράδειγμα, όταν εφαρμόζεται σε διανύσματα bit, παράγει ένα νέο διάνυσμα bit με μήκος ίσο με το άθροισμα των μηκών των δύο τελεστέων. Κατά συνέπεια, b"0000" & b"1111" παράγει b"0000_1111".

Ο τελεστής συνένωσης μπορεί να εφαρμοστεί σε δύο τελεστέους, ο ένας από τους οποίους είναι ένας πίνακας και άλλος είναι ένα απλό βαθμωτό στοιχείο. Μπορεί επίσης να εφαρμοστεί σε δύο βαθμωτές τιμές για να παράγει έναν πίνακα μήκους 2. Μερικά παραδείγματα είναι

```
"abc" & 'd' = "abcd"
'w' & "xyz" = "wxyz"
'a' & 'b' = "ab"
```

VHDL-87

Ο λογικός τελεστής **xnor** και οι τελεστές ολίσθησης **sll**, **srl**, **sla**, **sra**, **rol** και **ror** δεν παρέχονται στη VHDL-87.

4.3.1 Τμήματα Πινάκων

Συχνά θέλουμε να αναφερθούμε σε ένα συνεχόμενο υποσύνολο στοιχείων ενός πίνακα, αλλά όχι σε ολόκληρο τον πίνακα. Μπορούμε να το κάνουμε αυτό χρησιμοποιώντας τη σημειογραφία *τμήματος (slice)*, στην οποία καθορίζουμε την αριστερή και δεξιά τιμή του δείκτη ενός τμήματος ενός πίνακα. Για παράδειγμα, δοθέντος των πινάκων a1 και a2 που έχουν δηλωθεί ως εξής:

```
type array1 is array (1 to 100) of integer;
type array2 is array (100 downto 1) of integer;
variable a1 : array1;
variable a2 : array2;
```

μπορούμε να αναφερθούμε στο τμήμα του πίνακα a1(11 **to** 20), το οποίο είναι ένας πίνακας 10 στοιχείων που έχουν τους δείκτες 11 έως 20. Ομοίως, το τμήμα a2(50 **downto** 41) είναι ένας πίνακας 10 στοιχείων αλλά με μια φθίνουσα σειρά δείκτη. Σημειώστε ότι τα τμήματα a1(10 **to** 1) και a2(1 **downto** 10) είναι *κενά* τμήματα (*null* slices), αφού το εύρος του δείκτη καθορίζεται να είναι κενό. Επιπλέον, το εύρος που καθορίζεται στο τμήμα πρέπει να έχει την ίδια κατεύθυνση με τον αρχικό πίνακα. Κατά συνέπεια δεν είναι νόμιμο να γράψουμε a1(10 **downto** 1) ή a2(1 **to** 10).

ΠΑΡΑΔΕΙΓΜΑ

Η Εικόνα 4-5 είναι ένα μοντέλο συμπεριφοράς για ένα κύκλωμα εναλλαγής ψηφιολέξεων (byte-swapper) που έχει μια θύρα εισόδου και μια θύρα εξόδου, κάθε μία από τις οποίες είναι διάνυσμα bit του υποτύπου halfword, που έχει δηλωθεί ως εξής:

```
subtype halfword is bit_vector(0 to 15);
```

Η διεργασία στο σώμα αρχιτεκτονικής εναλλάσσει τη μία ψηφιολέξη εισόδου με την άλλη. Δείχνει πώς μπορεί να χρησιμοποιηθεί η σημειογραφία τμήματος για σήματα τύπου πίνακα στις προτάσεις ανάθεσης σημάτων.

ΕΙΚΟΝΑ 4-5

```
entity byte_swap is
    port (input : in halfword; output : out halfword);
end entity byte_swap;
```

```
architecture behavior of byte_swap is
begin
    swap : process (input)
    begin
        output(8 to 15) <= input(0 to 7);
        output(0 to 7) <= input(8 to 15);
    end process swap;
end architecture behavior;
```

Μια οντότητα και το σώμα αρχιτεκτονικής για μια λειτουργική μονάδα εναλλαγής ψηφιολέξεων.

VHDL-87

Στη VHDL-87, το εύρος που καθορίζεται σε ένα τμήμα μπορεί να έχει την αντίθετη κατεύθυνση από αυτή του εύρους του δείκτη του πίνακα. Σε αυτήν την περίπτωση, το τμήμα είναι ένα μηδενικό τμήμα.

4.3.2 Μετατροπές Τύπων

Στο Κεφάλαιο 2 εισάγαμε την ιδέα της μετατροπής τύπου μιας αριθμητικής τιμής σε μια άλλη τιμή ενός στενά σχετιζόμενου τύπου. Μια τιμή ενός τύπου πίνακα μπορεί επίσης να μετατραπεί σε μια τιμή ενός άλλου τύπου πίνακα, υπό τον όρο ότι και οι δύο τύποι πινάκων έχουν τον ίδιο τύπο στοιχείων, τον ίδιο αριθμό διαστάσεων και τύπους δεικτών που να μπορούν να μετατραπούν. Η μετατροπή τύπων παράγει απλά μια νέα τιμή πίνακα του καθορισμένου τύπου, με κάθε δείκτη να μετατρέπεται στην τιμή της αντίστοιχης θέσης του εύρους του δείκτη του νέου τύπου.

Για να επεξηγήσουμε την ιδέα της μετατροπής τύπου των τιμών του πίνακα, υποθέστε ότι έχουμε τις ακόλουθες δηλώσεις σε ένα μοντέλο:

```
subtype name is string(1 to 20);
type display_string is array (integer range 0 to 19) of character;
variable item_name : name;
```

```
variable display : display_string;
```

Δεν μπορούμε να αναθέσουμε απευθείας την τιμή του item_name στο display, αφού οι τύποι είναι διαφορετικοί. Εντούτοις, μπορούμε να το κάνουμε χρησιμοποιώντας μια μετατροπή τύπου:

```
display := display_string(item_name);
```

Αυτό παράγει ένα νέο πίνακα, με το αριστερό στοιχείο να έχει το δείκτη 0 και το δεξιό στοιχείο να έχει το δείκτη 19, ο οποίος είναι συμβατός με το στόχο της ανάθεσης.

Μια συνηθισμένη περίπτωση στην οποία δεν χρειαζόμαστε μετατροπή τύπου είναι η ανάθεση μιας τιμής πίνακα ενός υποτύπου σε ένα αντικείμενο πίνακα ενός διαφορετικού υποτύπου του ίδιου τύπου βάσης. Αυτό συμβαίνει στις περιπτώσεις όπου τα εύρη των δεικτών του στόχου και του τελεστέου έχουν διαφορετικά όρια ή κατευθύνσεις. Η VHDL περιλαμβάνει αυτόματα μια αυτονόητη μετατροπή υποτύπου στην ανάθεση. Για παράδειγμα, δοθέντος των υποτύπων και των μεταβλητών που δηλώνονται έτσι:

```
subtype big_endian_upper_halfword is bit_vector(0 to 15);
subtype little_endian_upper_halfword is bit_vector(31 downto 16);
variable big : big_endian_upper_halfword;
variable little : little_endian_upper_halfword;
```

θα μπορούσαμε να κάνουμε τις ακόλουθες αναθέσεις χωρίς να συμπεριλάβουμε ρητά μετατροπές τύπων:

```
big := little;
little := big;
```

4.4. Εγγραφές

Σε αυτήν την ενότητα, συζητάμε τη δεύτερη κατηγορία σύνθετων τύπων, τις *εγγραφές (records)*. Μια εγγραφή είναι μια σύνθετη τιμή που αποτελείται από στοιχεία των οποίων οι τύποι μπορούν να είναι διαφορετικοί. Κάθε στοιχείο προσδιορίζεται από ένα όνομα, το οποίο είναι μοναδικό εντός της εγγραφής. Αυτό το όνομα χρησιμοποιείται για να επιλέξει το στοιχείο από την τιμή της εγγραφής. Ο συντακτικός κανόνας για έναν ορισμό τύπου εγγραφής είναι

```
record_type_definition ⇐
    record
        ( identifier { , … } : subtype_indication ; )
        { … }
    end record [ identifier ]
```

Κάθε ένα από τα ονόματα στις λίστες των αναγνωριστικών δηλώνει ένα στοιχείο του δηλωμένου τύπου ή υποτύπου. Θυμηθείτε ότι τα άγκιστρα στους κανόνες σύνταξης υποδηλώνουν ότι το τμήμα που εσωκλείουν μπορεί να επαναληφθεί επ' αόριστον. Κατά συνέπεια, μπορούμε να συμπεριλάβουμε διάφορα στοιχεία διαφορετικών τύπων μέσα στην εγγραφή. Το προσδιοριστικό στο τέλος του ορισμού του τύπου εγγραφής, εάν περιλαμβάνεται, πρέπει να επαναλάβει το όνομα του τύπου εγγραφής.

VHDL-87

Το όνομα του τύπου εγγραφής δεν επιτρέπεται να συμπεριληφθεί στο τέλος ενός ορισμού τύπου εγγραφής στη VHDL-87.

Τα παρακάτω είναι ένα παράδειγμα μιας δήλωσης τύπου εγγραφής και δηλώσεων μεταβλητών που χρησιμοποιούν τον τύπο εγγραφής:

```
type time_stamp is record
        seconds : integer range 0 to 59;
        minutes : integer range 0 to 59;
        hours : integer range 0 to 23;
    end record time_stamp;
variable sample_time, current_time : time_stamp;
```

Ολόκληρες τιμές εγγραφής μπορούν να ανατεθούν στις μεταβλητές χρησιμοποιώντας προτάσεις ανάθεσης, για παράδειγμα:

```
sample_time := current_time;
```

Μπορούμε επίσης να αναφερθούμε σε ένα στοιχείο σε μια εγγραφή χρησιμοποιώντας ένα *επιλεγμένο όνομα (selected name)*, για παράδειγμα:

```
sample_hour := sample_time.hours;
```

Στην παράσταση στα δεξιά του συμβόλου ανάθεσης, το πρόθεμα πριν από την τελεία ονομάζει την τιμή της εγγραφής, και το επίθεμα μετά από την τελεία επιλέγει το στοιχείο από την εγγραφή. Ένα επιλεγμένο όνομα μπορεί επίσης να χρησιμοποιηθεί στην αριστερή πλευρά μιας ανάθεσης για να προσδιορίσει ένα στοιχείο της εγγραφής που πρόκειται να τροποποιηθεί, για παράδειγμα:

```
current_time.seconds := clock mod 60;
```

ΠΑΡΑΔΕΙΓΜΑ

Στα αρχικά στάδια της σχεδίασης του συνόλου εντολών για μια ΚΜΕ (CPU), δεν θέλουμε να δεσμεύσουμε μια κωδικοποίηση για τους τελεστές και τους τελεστέους μέσα σε μια λέξη εντολής. Αντί για αυτό χρησιμοποιούμε έναν τύπο εγγραφής για να αναπαραστήσουμε τα συστατικά μιας εντολής. Αυτό το επεξηγούμε στην Εικόνα 4-6, που είναι το περίγραμμα ενός μοντέλου συμπεριφοράς στο επιπέδου συστήματος μιας ΚΜΕ και της μνήμης που χρησιμοποιεί τύπους εγγραφής για να αναπαραστήσει εντολές και δεδομένα.

ΕΙΚΟΝΑ 4-6

```
architecture system_level of computer is
    type opcodes is (add, sub, addu, subu, jmp, breq, brne, ld, st, …);
    type reg_number is range 0 to 31;
    constant r0 : reg_number := 0; constant r1 : reg_number := 1; …
    type instruction is record
            opcode : opcodes;
            source_reg1, source_reg2, dest_reg : reg_number;
            displacement : integer;
        end record instruction;
    type word is record
            instr : instruction;
            data : bit_vector(31 downto 0);
        end record word;
    signal address : natural;
    signal read_word, write_word : word;
    signal mem_read, mem_write : bit := '0';
    signal mem_ready : bit := '0';
begin
    cpu : process is
        variable instr_reg : instruction;
        variable PC : natural;
        …       -- other declarations for register file, etc.
    begin
        address <= PC;
        mem_read <= '1';
        wait until mem_ready = '1';
        instr_reg := read_word.instr;
        mem_read <= '0';
        PC := PC + 4;
        case instr_reg.opcode is        -- execute the instruction
            …
        end case;
    end process cpu;
    memory : process is
        subtype address_range is natural range 0 to 2**14 – 1;
        type memory_array is array (address_range) of word;
        variable store : memory_array :=
           ( 0 => ( ( ld, r0, r0, r2, 40 ), X"00000000" ),
             1 => ( ( breq, r2, r0, r0, 5 ), X"00000000" ),
             …
             40 => ( ( nop, r0, r0, r0, 0 ), X"FFFFFFFE"),
             others => ( ( nop, r0, r0, r0, 0 ), X"00000000") );
    begin
        …
    end process memory;
end architecture system_level;
```

Το περίγραμμα του σώματος αρχιτεκτονικής συμπεριφοράς ενός υπολογιστικού συστήματος που αποτελείται από μια ΚΜΕ και μια μνήμη και χρησιμοποιεί τιμές εγγραφής για να αναπαραστήσει τιμές εντολών και δεδομένων.

Ο τύπος εγγραφής instruction αναπαριστά τις πληροφορίες που περιλαμβάνονται σε κάθε εντολή ενός προγράμματος και περιλαμβάνει τον κωδικό της πράξης (opcode), τους αριθμούς των καταχωρητών πηγής (source) και προορισμού (destination) και μια μετατόπιση (displacement). Ο τύπος εγγραφής word αναπαριστά μια λέξη που αποθηκεύεται στη μνήμη. Δεδομένου ότι μια λέξη ενδεχομένως να αναπαριστά μια εντολή ή δεδομένα, συμπεριλαμβάνονται στην εγγραφή στοιχεία και για τις δύο εκδοχές. Αντίθετα από πολλές συμβατικές γλώσσες προγραμματισμού, η VHDL δεν παρέχει εναλλακτικά μέρη στις τιμές των εγγραφών. Ο τύπος εγγραφής word επεξηγεί πώς οι σύνθετες τιμές δεδομένων μπορούν να περιλαμβάνουν στοιχεία που είναι τα ίδια σύνθετες τιμές, υπό τον όρο ότι τα συμπεριλαμβανόμενα στοιχεία είναι ενός υποτύπου με περιορισμούς. Τα σήματα στο μοντέλο χρησιμοποιούνται για τις συνδέσεις διευθύνσεων, δεδομένων και ελέγχου μεταξύ της ΚΜΕ και της μνήμης.

Μέσα στη διεργασία της ΚΜΕ η μεταβλητή instr_reg αναπαριστά τον καταχωρητή εντολής που περιέχει την τρέχουσα οδηγία που εκτελείται. Η διεργασία προσκομίζει μια λέξη από τη μνήμη και αντιγράφει το στοιχείο της εντολής από την εγγραφή στον καταχωρητή εντολής. Χρησιμοποιεί έπειτα το πεδίο opcode της τιμής για να καθορίσει πώς θα εκτελέσει την εντολή.

Η διεργασία της μνήμης περιέχει μια μεταβλητή που είναι ένας πίνακας από εγγραφές λέξεων που αναπαριστά την αποθήκευση της μνήμης. Ο πίνακας αρχικοποιείται με ένα πρόγραμμα και δεδομένα. Οι λέξεις που αναπαριστούν εντολές αρχικοποιούνται με μια συνάθροιση εγγραφής που περιέχει μια συνάθροιση της εγγραφής εντολής και ένα διάνυσμα bit, το οποίο αγνοείται. Ομοίως, οι λέξεις που αναπαριστούν δεδομένα αρχικοποιούνται με μια συνάθροιση που περιέχει μια συνάθροιση εντολής, η οποία αγνοείται, και το διάνυσμα bit των δεδομένων.

4.4.1 Συναθροίσεις Εγγραφών

Μπορούμε να χρησιμοποιήσουμε μια συνάθροιση εγγραφής για να γράψουμε μια κυριολεκτική τιμή ενός τύπου εγγραφής – για παράδειγμα, για να αρχικοποιήσουμε μια μεταβλητή ή μια σταθερά εγγραφής. Η χρησιμοποίηση μιας συνάθροισης εγγραφής είναι ανάλογη με τη χρησιμοποίηση μιας συνάθροισης πίνακα για το γράψιμο μιας κυριολεκτικής τιμής ενός τύπου πίνακα. Μια συνάθροιση εγγραφής συντάσσεται με το γράψιμο μιας λίστας στοιχείων που εσωκλείονται σε παρενθέσεις. Μια συνάθροιση που χρησιμοποιεί συσχέτιση θέσης απαριθμεί τα στοιχεία με την ίδια σειρά όπως αυτά εμφανίζονται στη δήλωση του τύπου εγγραφής. Για παράδειγμα, δοθέντος του τύπου εγγραφής time_stamp που παρουσιάστηκε παραπάνω, μπορούμε να αρχικοποιήσουμε μια σταθερά ως εξής:

```
constant midday : time_stamp := (0, 0, 12);
```

Μπορούμε επίσης να χρησιμοποιήσουμε συσχέτιση ονόματος, στην οποία προσδιορίζουμε κάθε στοιχείο στη συνάθροιση από το όνομά του. Η σειρά των στοιχείων που προσδιορίζονται χρησιμοποιώντας συσχέτιση ονόματος δεν έχει επιπτώσεις στην τιμή της συνάθροισης. Το παραπάνω παράδειγμα θα μπορούσε να ξαναγραφεί ως εξής

```
constant midday : time_stamp := (hours => 12, minutes => 0, seconds => 0);
```

Αντίθετα από τις συναθροίσεις πινάκων, μπορούμε να αναμίξουμε τη συσχέτιση θέση και τη συσχέτιση ονόματος στις συναθροίσεις εγγραφών, υπό τον όρο ότι όλα τα στοιχεία που προσδιορίζονται με συσχέτιση ονόματος ακολουθούν οποιαδήποτε στοιχεία που προσδιορίζονται με συσχέτιση θέσης. Μπορούμε επίσης να χρησιμοποιήσουμε τα σύμβολα "|" και **others** κατά το γράψιμο των επιλογών. Εδώ είναι μερικά ακόμη παραδείγματα, που χρησιμοποιούν τους τύπους instruction και time_stamp που δηλώθηκαν παραπάνω:

```
constant nop_instr : instruction :=
        ( opcode => addu,
          source_reg1 | source_reg2 | dest_reg => 0,
          displacement => 0 );
variable latest_event : time_stamp := (others => 0); — initially midnight
```

Σημειώστε ότι αντίθετα από τις συναθροίσεις πινάκων, δεν μπορούμε να χρησιμοποιήσουμε ένα εύρος τιμών για να προσδιορίσουμε τα στοιχεία σε μια συνάθροιση εγγραφής, αφού τα στοιχεία προσδιορίζονται από ονόματα, και δεν δεικτοδοτούνται από ένα διακριτό εύρος.

Κεφάλαιο 5: Βασικές Δομές Μοντελοποίησης

Η περιγραφή μιας λειτουργικής μονάδας σε ένα ψηφιακό σύστημα μπορεί να διαιρεθεί σε δύο πλευρές: την εξωτερική άποψη και την εσωτερική άποψη. Η εξωτερική άποψη περιγράφει τη διασύνδεση στη λειτουργική μονάδα, συμπεριλαμβανομένου του αριθμού και των τύπων των εισόδων και των εξόδων. Η εσωτερική άποψη περιγράφει πως η λειτουργική μονάδα υλοποιεί τη συνάρτησή της. Στη VHDL, μπορούμε να διαχωρίσουμε την περιγραφή μιας λειτουργικής μονάδας σε μια *δήλωση οντότητας (entity declaration)*, που περιγράφει την εξωτερική διασύνδεση, και ένα ή περισσότερα *σώματα αρχιτεκτονικής (architecture bodies)*, τα οποία περιγράφουν τις εναλλακτικές εσωτερικές υλοποιήσεις. Αυτές οι έννοιες εισήχθησαν στο Κεφάλαιο 1 και συζητούνται λεπτομερώς σε αυτό το κεφάλαιο. Εξετάζουμε επίσης το πως μία σχεδίαση επεξεργάζεται ώστε να προετοιμαστεί για προσομοίωση ή σύνθεση.

5.1 Δηλώσεις Οντότητας

Αρχικά θα εξετάσουμε τους συντακτικούς κανόνες για μια δήλωση οντότητας και έπειτα θα παρουσιάσουμε μερικά παραδείγματα. Ξεκινάμε με μια απλουστευμένη περιγραφή των δηλώσεων οντότητας και προχωρούμε σε μια πλήρη περιγραφή αργότερα σε αυτό το κεφάλαιο. Οι συντακτικοί κανόνες για αυτήν την απλουστευμένη μορφή δήλωσης οντότητας είναι

entity_declaration ⇐

entity identifier **is**

[**port** (port_interface_list) ;]

{ entity_declarative_item }

end [**entity**] [identifier] ;

interface_list ⇐

(identifier { , … } : [mode] subtype_indication [:= expression]) { ; … }

mode ⇐ **in** I **out** I **inout**

Το αναγνωριστικό σε μια δήλωση οντότητας ονομάζει τη λειτουργική μονάδα έτσι ώστε να μπορεί να γίνει αναφορά σε αυτήν αργότερα. Εάν το αναγνωριστικό συμπεριλαμβάνεται στο τέλος της δήλωσης, πρέπει να επαναληφθεί το όνομα της οντότητας. Η φράση θύρας (port clause) ονομάζει κάθε μία από τις *θύρες (ports)*, οι οποίες όλες μαζί συγκροτούν τη διασύνδεση στην οντότητα. Μπορούμε να αναλογιστούμε τις θύρες ως παρεμφερή των ακροδεκτών ενός κυκλώματος - δηλαδή είναι τα μέσα από τα οποία οι πληροφορίες εισάγονται και εξάγονται από το κύκλωμα. Στη VHDL, κάθε θύρα μιας οντότητας έχει έναν *τύπο (type)*, που προσδιορίζει το είδος της πληροφορίας που μπορεί να μεταβιβαστεί, και μία *κατάσταση (mode)*, η οποία προσδιορίζει εάν η ροή της πληροφορίας μέσω της θύρας είναι προς το εσωτερικό ή προς το εξωτερικό της οντότητας. Αυτές οι πτυχές του τύπου και της κατεύθυνσης είναι σε συμφωνία με τη φιλοσοφία αυστηρής τυποποίησης της VHDL, η οποία μας βοηθά να αποφεύγουμε τις λανθασμένες περιγραφές κυκλωμάτων. Ένα απλό παράδειγμα μιας δήλωσης οντότητας είναι

```
entity adder is
    port (a : in word;
          b : in word;
          sum : out word );
end entity adder;
```

Αυτό το παράδειγμα περιγράφει μια οντότητα που ονομάζεται adder, με δύο θύρες εισόδου και μία θύρα εξόδου, όλες του τύπου word, που υποθέτουμε ότι καθορίζεται αλλού. Μπορούμε να απαριθμήσουμε τις θύρες με οποιαδήποτε σειρά - δεν είναι απαραίτητο να βάλουμε τις εισόδους πριν από τις εξόδους. Επίσης, μπορούμε να συμπεριλάβουμε μία λίστα θυρών της ίδιας κατάστασης και του ίδιου τύπου αντί να γράψουμε την καθεμία χωριστά. Κατά συνέπεια η παραπάνω δήλωση θα μπορούσε να γραφτεί εξίσου σωστά ως εξής:

```
entity adder is
    port (a, b : in word;
          sum : out word );
end entity adder;
```

Σε αυτό το παράδειγμα έχουμε δει θύρες εισόδου και εξόδου. Μπορούμε επίσης να έχουμε θύρες διπλής κατεύθυνσης, με την κατάσταση **inout**. Αυτές οι θύρες μπορούν να χρησιμοποιηθούν για να μοντελοποιήσουν συσκευές που εναλλακτικά διαβάζουν ή οδηγούν δεδομένα μέσω ενός ακροδέκτη. Τέτοια μοντέλα πρέπει να χειριστούν την πιθανότητα περισσότερες από μία συνδεδεμένες συσκευές να οδηγούν ένα δεδομένο σήμα την ίδια χρονική στιγμή. Η VHDL παρέχει ένα μηχανισμό για αυτόν το σκοπό, την *ανάλυση σήματος (signal resolution)*.

Η ομοιότητα μεταξύ της περιγραφής μίας θύρας σε μια δήλωση οντότητας και της δήλωσης μιας μεταβλητής μπορεί να είναι προφανής. Αυτή η ομοιότητα δεν είναι συμπτωματική, και μπορούμε να επεκτείνουμε την αναλογία προσδιορίζοντας μια προκαθορισμένη τιμή στην περιγραφή μίας θύρας, για παράδειγμα:

```
entity and_or_inv is
    port ( a1, a2, b1, b2 : in bit := '1';
           y : out bit );
end entity and_or_inv;
```

Η προκαθορισμένη τιμή, σε αυτήν την περίπτωση το '1' στις θύρες εισόδου, υποδηλώνει την τιμή που κάθε θύρα πρέπει να υποθέσει εάν ένα μοντέλο που την εσωκλείει την αφήσει ασύνδετη. Μπορούμε να το αναλογιστούμε σαν την περιγραφή της τιμής την οποία η θύρα θα πάρει εάν δεν χρησιμοποιηθεί. Αφετέρου, εάν η θύρα χρησιμοποιηθεί, η προκαθορισμένη τιμή αγνοείται. Θα αναφέρουμε περισσότερα για τη χρήση των προκαθορισμένων τιμών όταν εξετάσουμε την εκτέλεση ενός μοντέλου.

Ένα άλλο σημείο που πρέπει να επισημάνουμε για τις δηλώσεις οντοτήτων είναι ότι η πρόταση θύρας είναι προαιρετική. Έτσι, μπορούμε να γράψουμε μια δήλωση οντότητας ως εξής

```
entity top_level is
end entity top_level;
```

η οποία περιγράφει μια απολύτως αυτόνομη λειτουργική μονάδα. Όπως υπονοεί το όνομα σε αυτό το παράδειγμα, αυτό το είδος λειτουργικής μονάδας αναπαριστά συνήθως το υψηλότερο επίπεδο μιας σχεδιαστικής ιεραρχίας.

Επίσης, μπορούμε να συμπεριλάβουμε δηλώσεις στοιχείων μέσα σε μια δήλωση οντότητας. Αυτές περιλαμβάνουν δηλώσεις σταθερών, τύπων, σημάτων και άλλων ειδών στοιχείων που θα δούμε αργότερα σε αυτό το κεφάλαιο. Τα στοιχεία μπορούν να χρησιμοποιηθούν σε όλα τα σώματα αρχιτεκτονικής που αντιστοιχούν στην οντότητα. Κατά συνέπεια, έχει νόημα να συμπεριλάβουμε δηλώσεις που είναι σχετικές με την οντότητα και όλες τις πιθανές υλοποιήσεις της. Αντίθετα οτιδήποτε είναι μέρος μόνο μιας συγκεκριμένης υλοποίησης πρέπει να δηλωθεί μέσα στο αντίστοιχο σώμα αρχιτεκτονικής.

ΠΑΡΑΔΕΙΓΜΑ

Υποθέστε ότι σχεδιάζουμε έναν ενσωματωμένο ελεγκτή (embedded controller) χρησιμοποιώντας έναν μικροεπεξεργαστή με ένα πρόγραμμα που αποθηκεύεται σε μια μνήμη μόνο για ανάγνωση (read-only memory, ROM). Το πρόγραμμα που αποθηκεύεται στη ROM είναι αμετάβλητο, αλλά χρειάζεται να μοντελοποιήσουμε τη ROM σε διαφορετικά επίπεδα λεπτομέρειας. Μπορούμε να συμπεριλάβουμε δηλώσεις που περιγράφουν το πρόγραμμα στη δήλωση οντότητας της ROM, όπως φαίνεται στην Εικόνα 5-1. Αυτές οι δηλώσεις δεν είναι άμεσα προσβάσιμες σε ένα χρήστη της οντότητας ROM, αλλά χρησιμεύουν στην τεκμηρίωση των περιεχομένων της ROM. Κάθε σώμα αρχιτεκτονικής που αντιστοιχεί στην οντότητα μπορεί να χρησιμοποιήσει τη σταθερά program για να αρχικοποιήσει οποιαδήποτε δομή χρησιμοποιεί εσωτερικά για να υλοποιήσει τη ROM.

ΕΙΚΟΝΑ 5-1

```
entity program_ROM is
    port ( address : in std_ulogic_vector(14 downto 0);
           data : out std_ulogic_vector(7 downto 0);
           enable : in std_ulogic );
    subtype instruction_byte is bit_vector(7 downto 0);
    type program_array is array (0 to 2**14 – 1) of instruction_byte;
    constant program : program_array
        := (  X"32", X"3F", X"03",        — LDA $3F03
              X"71", X"23",               — BLT $23
              …
           );
end entity program_ROM;
```

Μία δήλωση οντότητας για μία ROM, συμπεριλαμβανομένων των δηλώσεων που περιγράφουν το πρόγραμμα που περιέχεται σε αυτήν.

VHDL-87

Η δεσμευμένη λέξη **entity** δεν επιτρέπεται να συμπεριληφθεί στο τέλος μίας δήλωσης οντότητας στη VHDL-87.

5.2 Σώματα Αρχιτεκτονικής

Η εσωτερική λειτουργία μιας λειτουργικής μονάδας περιγράφεται από ένα σώμα αρχιτεκτονικής. Ένα σώμα αρχιτεκτονικής γενικά εφαρμόζει κάποιες λειτουργίες στις τιμές των θυρών εισόδου, παράγοντας τιμές που ανατίθενται στις θύρες εξόδου. Οι λειτουργίες μπορούν να περιγραφούν είτε από διεργασίες (processes), οι οποίες περιέχουν ακολουθιακές προτάσεις (sequential statements) που ενεργούν στις τιμές, ή από μια συλλογή συστατικών στοιχείων (components) που αναπαριστούν υποκυκλώματα. Όπου η λειτουργία απαιτεί την παραγωγή ενδιάμεσων τιμών, αυτές μπορούν να περιγραφούν με τη χρήση *σημάτων (signals)*, σε αντιστοιχία με τα εσωτερικά καλώδια μιας λειτουργικής μονάδας. Ο συντακτικός κανόνας για τα σώματα αρχιτεκτονικής παρουσιάζει ένα γενικό περίγραμμα:

architecture_body ⇐

architecture identifier **of** entity_name **is**

{ block_declarative_item }

begin
{ concurrent_statement }

end [**architecture**] [identifier] ;

Το αναγνωριστικό κατονομάζει αυτό το συγκεκριμένο σώμα αρχιτεκτονικής, και το όνομα της οντότητας καθορίζει ποιανής μονάδας η λειτουργία περιγράφεται από αυτό το σώμα αρχιτεκτονικής. Εάν το αναγνωριστικό συμπεριλαμβάνεται στο τέλος του σώματος αρχιτεκτονικής, πρέπει να επαναληφθεί το όνομα του σώματος αρχιτεκτονικής. Μπορούν να υπάρχουν πολλά διαφορετικά σώματα αρχιτεκτονικής που αντιστοιχούν σε μια απλή οντότητα, κάθε ένα από τα οποία περιγράφει έναν εναλλακτικό τρόπο υλοποίησης της λειτουργίας της μονάδας. Τα δηλωτικά στοιχεία του μπλοκ (block_declarative_item) σε ένα σώμα αρχιτεκτονικής είναι δηλώσεις απαραίτητες για την υλοποίηση των λειτουργιών. Τα στοιχεία μπορεί να περιλαμβάνουν δηλώσεις τύπων και σταθερών, δηλώσεις σημάτων και άλλα είδη δηλώσεων.

VHDL-87

Η δεσμευμένη λέξη **architecture** δεν επιτρέπεται να συμπεριληφθεί στο τέλος ενός σώματος αρχιτεκτονικής στη VHDL-87.

5.2.1 Ταυτόχρονες Προτάσεις

Οι *ταυτόχρονες προτάσεις (concurrent statements)* σε ένα σώμα αρχιτεκτονικής περιγράφουν τη λειτουργία της μονάδας. Μια μορφή ταυτόχρονης πρότασης, που έχουμε ήδη δει, είναι η πρόταση διεργασίας (process statement). Βάζοντας αυτό μαζί με τον κανόνα για τη γραφή σωμάτων αρχιτεκτονικής, μπορούμε να εξετάσουμε ένα απλό παράδειγμα ενός σώματος αρχιτεκτονικής που αντιστοιχεί στην οντότητα adder που είδαμε προηγουμένως:

```
architecture abstract of adder is
begin
    add_a_b : process (a, b) is
    begin
        sum <= a + b;
    end process add_a_b;
end architecture abstract;
```

Το σώμα αρχιτεκτονικής ονομάζεται abstract, και περιέχει μια διεργασία add_a_b, η οποία περιγράφει τη λειτουργία της οντότητας. Η διεργασία υποθέτει ότι ο τελεστής + ορίζεται για τον τύπο word, τον τύπο των a και b. Θα μπορούσαμε επίσης να εξετάσουμε επιπλέον σώματα αρχιτεκτονικής που περιγράφουν τον αθροιστή με διαφορετικούς τρόπους, υπό τον όρο ότι όλα συμβαδίζουν με την εξωτερική διασύνδεση που διατυπώνεται από τη δήλωση της οντότητας.

Έχουμε εξετάσει πρώτα τις διεργασίες επειδή είναι η πιο θεμελιώδης μορφή ταυτόχρονης πρότασης. Όλες οι άλλες μορφές μπορούν εν τέλει να μειωθούν σε μια ή περισσότερες διεργασίες. Οι ταυτόχρονες προτάσεις αποκαλούνται έτσι επειδή εννοιολογικά μπορούν να ενεργοποιηθούν και να εκτελέσουν τις ενέργειές τους μαζί, δηλαδή ταυτόχρονα. Συγκρίνετε αυτό με τις ακολουθιακές προτάσεις εντός μιας διεργασίας, οι οποίες εκτελούνται η μία μετά από την άλλη. Ο παραλληλισμός είναι χρήσιμος για να μοντελοποιήσουμε τον τρόπο που συμπεριφέρονται τα πραγματικά κυκλώματα. Εάν έχουμε δύο πύλες των οποίων οι είσοδοι αλλάζουν, κάθε μια υπολογίζει τη νέα της έξοδο ανεξάρτητα από την άλλη. Δεν υπάρχει καμία έμφυτη ακολουθία που να διέπει την σειρά με την οποία υπολογίζονται. Εξετάζουμε τις προτάσεις διεργασίας λεπτομερέστερα στην Ενότητα 5.3. Στη συνέχεια, στην Ενότητα 5.4, εξετάζουμε μια άλλη μορφή ταυτόχρονης πρότασης, την πρόταση εμφάνισης στιγμιότυπου συστατικού στοιχείου (component instantiation statement), που χρησιμοποιείται για να περιγράψει πως μια λειτουργική μονάδα απαρτίζεται από διασυνδεδεμένες λειτουργικές υπομονάδες.

5.2.2 Δηλώσεις Σημάτων

Όταν απαιτείται να χρησιμοποιήσουμε εσωτερικά σήματα σε ένα σώμα αρχιτεκτονικής, πρέπει να τα ορίσουμε χρησιμοποιώντας *δηλώσεις σημάτων (signal declarations)*. Η σύνταξη για μια δήλωση σήματος ταιριάζει πολύ με τη σύνταξη για μια δήλωση μεταβλητής:

signal_declaration ⇐

signal identifier { , … } : subtype_indication [:= expression] ;

Αυτή η δήλωση απλά ονομάζει κάθε σήμα, καθορίζει τον τύπο του και προαιρετικά περιλαμβάνει μια αρχική τιμή για όλα τα σήματα που δηλώνονται στη δήλωση.

ΠΑΡΑΔΕΙΓΜΑ

Η Εικόνα 5-2 είναι ένα παράδειγμα ενός σώματος αρχιτεκτονικής για την οντότητα and_or_inv, που ορίστηκε προηγουμένως. Το σώμα αρχιτεκτονικής περιλαμβάνει τις δηλώσεις μερικών σημάτων που είναι εσωτερικά στο σώμα της αρχιτεκτονικής. Μπορούν να χρησιμοποιηθούν από τις διεργασίες εντός του σώματος της αρχιτεκτονικής αλλά δεν είναι προσβάσιμα έξω από αυτό, δεδομένου ότι ένας χρήστης της λειτουργικής μονάδας δεν χρειάζεται να ενδιαφερθεί για τις εσωτερικές λεπτομέρειες της υλοποίησής της. Στα σήματα ανατίθενται τιμές με τη χρήση προτάσεων ανάθεσης σήματος (signal assignment statements) εντός των διεργασιών. Τα σήματα μπορούν να διαβαστούν από τις διεργασίες για να αναγνώσουν τις τιμές τους.

ΕΙΚΟΝΑ 5-2

```
architecture primitive of and_or_inv is
    signal and_a, and_b : bit;
    signal or_a_b : bit;
begin
    and_gate_a : process (a1, a2) is
    begin
        and_a <= a1 and a2;
    end process and_gate_a;
    and_gate_b : process (b1, b2) is
    begin
        and_b <= b1 and b2;
    end process and_gate_b;
    or_gate : process (and_a, and_b) is
    begin
        or_a_b <= and_a or and_b;
    end process or_gate;
    inv : process (or_a_b) is
    begin
        y <= not or_a_b;
    end process inv;
end architecture primitive;
```

Εικόνα 5-2. *Ένα σώμα αρχιτεκτονικής που αντιστοιχεί στην οντότητα* and_or_inv.

Ένα σημαντικό σημείο που εμφανίζεται σε αυτό το παράδειγμα είναι ότι οι θύρες της οντότητας είναι επίσης ορατές στις διεργασίες μέσα στο σώμα αρχιτεκτονικής και χρησιμοποιούνται με τον ίδιο τρόπο όπως τα σήματα. Αυτό αντιστοιχεί στην άποψη μας για τις θύρες ότι είναι οι εξωτερικοί ακροδέκτες του κυκλώματος: από την εσωτερική άποψη, ένας ακροδέκτης είναι απλώς ένα καλώδιο με μια εξωτερική σύνδεση. Έτσι έχει νόημα για τη VHDL να μεταχειρίζεται τις θύρες όπως ακριβώς τα σήματα μέσα σε μια αρχιτεκτονική της οντότητας.

5.3 Περιγραφές Συμπεριφοράς

Στο πιο θεμελιώδες επίπεδο, η συμπεριφορά μιας λειτουργικής μονάδας περιγράφεται από τις προτάσεις ανάθεσης σημάτων μέσα στις διεργασίες. Μπορούμε να σκεφτούμε μια διεργασία ως τη βασική μονάδα της περιγραφής συμπεριφοράς (behavioral description). Μια διεργασία εκτελείται σαν απόκριση στις αλλαγές των τιμών των σημάτων και χρησιμοποιεί τις παρούσες τιμές των σημάτων που διαβάζει για να καθορίσει τις νέες τιμές για άλλα σήματα. Μια ανάθεση σήματος είναι μια ακολουθιακή πρόταση και επομένως μπορεί να εμφανιστεί μόνο μέσα σε μια διεργασία. Σε αυτήν την ενότητα, εξετάζουμε λεπτομερώς την αλληλεπίδραση μεταξύ σημάτων και διεργασιών.

5.3.1 Ανάθεση Σήματος

Σε όλα τα παραδείγματα που έχουμε εξετάσει μέχρι τώρα, έχουμε χρησιμοποιήσει μια απλή μορφή πρότασης ανάθεσης σήματος. Κάθε ανάθεση απλά παρέχει μια νέα τιμή για ένα σήμα. Η τιμή καθορίζεται με τον υπολογισμό μιας παράστασης, το αποτέλεσμα της οποίας πρέπει να ταιριάζει με τον τύπο του σήματος. Αυτό που δεν έχουμε εξετάσει ακόμα είναι το ζήτημα του χρονισμού: πότε το σήμα παίρνει τη νέα τιμή; Αυτό είναι θεμελιώδες στη μοντελοποίηση υλικού, όπου τα γεγονότα συμβαίνουν κατά τη διάρκεια του χρόνου. Αρχικά, ας εξετάσουμε τη σύνταξη για μια βασική πρόταση ανάθεσης σήματος σε μια διεργασία:

signal_assignment_statement ⇐

[label :] name <= [delay_mechanism] waveform ;

waveform ⇐ (value_expression [**after** time_expression]) { , … }

Η προαιρετική ετικέτα μας επιτρέπει να προσδιορίσουμε την πρόταση. Οι συντακτικοί κανόνες μας λένε ότι μπορούμε να καθορίσουμε ένα μηχανισμό καθυστέρησης, στον οποίο θα επανέλθουμε σύντομα, και ένα ή περισσότερα στοιχεία κυματομορφής, κάθε ένα από τα οποία αποτελείται από μια νέα τιμή και έναν προαιρετικό χρόνο καθυστέρησης. Αυτοί οι χρόνοι καθυστέρησης σε μια ανάθεση σήματος είναι που μας επιτρέπουν να καθορίσουμε πότε θα πρέπει να εφαρμοστεί η νέα τιμή. Για παράδειγμα, θεωρήστε την παρακάτω ανάθεση:

```
y <= not or_a_b after 5 ns;
```

Αυτό καθορίζει ότι η χρονική στιγμή που το σήμα y πρόκειται να πάρει τη νέα τιμή είναι 5 ns αργότερα από τη χρονική στιγμή κατά την οποία εκτελείται η πρόταση. Η καθυστέρηση μπορεί να διαβαστεί με δύο τρόπους, ανάλογα με το εάν το μοντέλο χρησιμοποιείται καθαρά για την περιγραφική του αξία ή για προσομοίωση. Στην πρώτη περίπτωση, η καθυστέρηση μπορεί να θεωρηθεί υπό μια αφηρημένη έννοια ως προδιαγραφή της καθυστέρησης διάδοσης της λειτουργικής μονάδας: όποτε η είσοδος αλλάζει, η έξοδος ενημερώνεται 5 ns αργότερα. Στη δεύτερη περίπτωση, μπορεί να θεωρηθεί υπό μια λειτουργική έννοια, με αναφορά σε μια μηχανή υπηρεσίας (host machine) που προσομοιώνει τη λειτουργία της μονάδας εκτελώντας το μοντέλο. Κατά συνέπεια εάν η παραπάνω πρόταση εκτελεστεί τη χρονική στιγμή 250 ns, και το or_a_b έχει την τιμή '1' εκείνη τη στιγμή, τότε το σήμα y θα πάρει την τιμή '0' τη χρονική στιγμή 255 ns. Σημειώστε ότι η ίδια η πρόταση εκτελείται σε μηδενικό χρόνο μοντελοποίησης.

Η χρονική διάσταση στην οποία αναφερόμαστε όταν το μοντέλο εκτελείται είναι ο *χρόνος προσομοίωσης (simulation time)*, δηλαδή ο χρόνος στον οποίο το κύκλωμα που μοντελοποιήθηκε εκτιμάται ότι θα λειτουργήσει. Αυτό είναι διαφορετικό από τον πραγματικό χρόνο εκτέλεσης σε μία μηχανή υπηρεσίας που «τρέχει» μια προσομοίωση. Μετράμε το χρόνο προσομοίωσης αρχίζοντας από τη χρονική στιγμή μηδέν στην έναρξη της εκτέλεσης και αυξάνοντάς τον με διακριτά βήματα όσο συμβαίνουν γεγονότα στο μοντέλο. Όπως ήταν αναμενόμενο, αυτή η τεχνική καλείται *προσομοίωση διακριτών γεγονότων (discrete event simulation)*. Ένας προσομοιωτής διακριτών γεγονότων πρέπει να διατηρεί ένα ρολόι χρόνου προσομοίωσης, και όταν εκτελείται μια πρόταση ανάθεσης σήματος, η καθορισμένη καθυστέρηση προστίθεται στον τρέχοντα χρόνο προσομοίωσης για να καθοριστεί πότε πρόκειται να εφαρμοστεί η νέα τιμή στο σήμα. Λέμε ότι η ανάθεση σήματος χρονοπρογραμματίζει (schedule) μια *συναλλαγή (transaction)* για το σήμα, όπου η συναλλαγή αποτελείται από τη νέα τιμή και το χρόνο προσομοίωσης κατά τον οποίο πρόκειται να εφαρμοστεί. Όταν ο χρόνος προσομοίωσης φτάσει στο χρόνο κατά τον οποίο μια συναλλαγή είναι χρονοπρογραμματισμένη, το σήμα ενημερώνεται με τη νέα τιμή. Λέμε ότι το σήμα είναι *ενεργό (active)* κατά τη διάρκεια εκείνου του κύκλου προσομοίωσης. Εάν η νέα τιμή ενός σήματος δεν είναι ίση με την παλιά τιμή που αντικαθιστά, λέμε ότι ένα *γεγονός (event)* συμβαίνει στο σήμα. Η σημασία αυτής της διάκρισης είναι ότι οι διεργασίες αποκρίνονται στα γεγονότα των σημάτων, και όχι στις συναλλαγές.

Οι συντακτικοί κανόνες για τις αναθέσεις σημάτων δείχνουν ότι μπορούμε να χρονοπρογραμματίσουμε έναν αριθμό συναλλαγών για ένα σήμα, οι οποίες πρόκειται να εφαρμοστούν μετά από διαφορετικές καθυστερήσεις. Για παράδειγμα, μια διεργασία που οδηγεί ένα ρολόι θα μπορούσε να εκτελέσει την ακόλουθη ανάθεση για να παράγει τις επόμενες δύο ακμές ενός σήματος ρολογιού (υποθέτοντας ότι T_pw είναι μια σταθερά που αντιπροσωπεύει το πλάτος παλμού του ρολογιού):

```
clk <= '1' after T_pw, '0' after 2*T_pw;
```

Εάν αυτή η δήλωση εκτελεστεί σε χρόνο προσομοίωσης 50 ns και το T_pw έχει τιμή 10 ns, μια συναλλαγή χρονοπρογραμματίζεται για τη χρονική στιγμή 60 ns που θέτει το clk στο '1', και μια δεύτερη συναλλαγή χρονοπρογραμματίζεται για τη χρονική στιγμή 70 ns που θέτει το clk στο '0'. Εάν υποθέσουμε ότι το clk έχει την τιμή '0' όταν εκτελείται η ανάθεση, και οι δύο συναλλαγές παράγουν γεγονότα στο clk.

Αυτή η δήλωση ανάθεσης σήματος δείχνει ότι όταν συμπεριλαμβάνονται περισσότερες από μια συναλλαγές σε μια ανάθεση, όλες οι καθυστερήσεις μετρώνται με βάση τον τρέχοντα χρόνο προσομοίωσης, και όχι με βάση το χρόνο από την προηγούμενη συναλλαγή. Επιπλέον, οι συναλλαγές στη λίστα πρέπει να έχουν αυστηρά αυξανόμενες καθυστερήσεις, έτσι ώστε η λίστα να διαβάζεται με τη σειρά με την οποία οι τιμές θα εφαρμοστούν στο σήμα.

ΠΑΡΑΔΕΙΓΜΑ

Μπορούμε να γράψουμε μια δήλωση διεργασίας για μια γεννήτρια ρολογιού χρησιμοποιώντας την παραπάνω πρόταση ανάθεσης σήματος για να παράγουμε ένα συμμετρικό σήμα ρολογιού με πλάτος παλμού T_pw. Η δυσκολία έγκειται στο να κάνουμε τη διεργασία να εκτελείται τακτικά σε κάθε κύκλο ρολογιού. Ένας τρόπος να γίνει αυτό είναι να κάνουμε τη διεργασία να αρχίζει εκ νέου όποτε το ρολόι αλλάζει και να χρονοπρογραμματίσουμε τις επόμενες δύο μεταβάσεις όταν αλλάζει σε '0'. Αυτή η προσέγγιση παρουσιάζεται στην Εικόνα 5-3.

ΕΙΚΟΝΑ 5-3

```
clock_gen : process (clk) is
begin
    if clk = '0' then
        clk <= '1' after T_pw, '0' after 2*T_pw;
    end if;
end process clock_gen;
```

Μία διεργασία που παράγει την κυματομορφή ενός συμμετρικού ρολογιού.

Δεδομένου ότι μια διεργασία είναι η βασική μονάδα μιας περιγραφής συμπεριφοράς, έχει διαισθητικό νόημα να επιτρέπεται μια απλή διεργασία να περιλαμβάνει περισσότερες από μια δηλώσεις ανάθεσης σήματος για ένα δεδομένο σήμα. Μπορούμε να το σκεφτούμε ως περιγραφή των διαφορετικών τρόπων με τους οποίους η τιμή ενός σήματος μπορεί να παραχθεί από τη διεργασία σε διαφορετικές χρονικές στιγμές.

ΠΑΡΑΔΕΙΓΜΑ

Μπορούμε να γράψουμε μια διεργασία που μοντελοποιεί έναν πολυπλέκτη δύο-εισόδων (two-input multiplexer) όπως φαίνεται στην Εικόνα 5-4. Η τιμή της θύρας sel χρησιμοποιείται για να επιλέξει ποια ανάθεση σήματος θα εκτελεστεί για να καθορίσει την τιμή της εξόδου.

ΕΙΚΟΝΑ 5-4

```
mux : process (a, b, sel) is
begin
    case sel is
        when '0' =>
            z <= a after prop_delay;
        when '1' =>
            z <= b after prop_delay;
    end case;
end process mux;
```

Μία διεργασία που μοντελοποιεί έναν πολυπλέκτη δύο-εισόδων

Λέμε ότι μια διεργασία καθορίζει έναν *οδηγό (driver)* για ένα σήμα εάν και μόνο εάν περιέχει τουλάχιστον μια δήλωση ανάθεσης σήματος για αυτό το σήμα. Έτσι αυτό το παράδειγμα προσδιορίζει έναν οδηγό για το σήμα z. Εάν μια διεργασία περιέχει τις δηλώσεις ανάθεσης σήματος για διάφορα σήματα, καθορίζει οδηγούς για κάθε ένα από εκείνα τα σήματα. Ένας οδηγός είναι μια *πηγή (source)* για ένα σήμα δεδομένου ότι παρέχει τις τιμές που εφαρμόζονται στο σήμα. Ένας σημαντικός κανόνας που πρέπει να θυμόμαστε είναι ότι για τα κανονικά σήματα, μπορεί να υπάρχει μόνο μια πηγή. Αυτό σημαίνει ότι δεν μπορούμε να γράψουμε δύο διαφορετικές διεργασίες και η καθεμία να περιέχει προτάσεις ανάθεσης σήματος για το ίδιο σήμα. Εάν θέλουμε να μοντελοποιήσουμε τέτοια πράγματα όπως διαύλους (buses) ή σήματα καλωδιωμένου-Ή (wired-or signals), πρέπει να χρησιμοποιήσουμε ένα ειδικό είδος σήματος που αποκαλείται *επιλυμένο σήμα (resolved signal)*.

VHDL-87

Οι προτάσεις ανάθεσης σήματος δεν επιτρέπεται να έχουν ετικέτες στη VHDL-87.

5.3.2 Ιδιότητες Σήματος

Στο Κεφάλαιο 2 εισάγαμε την ιδέα των ιδιοτήτων (attributes) των τύπων, οι οποίες παρέχουν πληροφορίες σχετικά με τις επιτρεπόμενες τιμές των τύπων. Κατόπιν, στο Κεφάλαιο 4, είδαμε πώς θα μπορούσαμε να χρησιμοποιήσουμε τις ιδιότητες των αντικειμένων τύπου πίνακα για να εξάγουμε πληροφορίες σχετικά με το εύρος των δεικτών τους. Μπορούμε επίσης να αναφερθούμε σε ιδιότητες σημάτων για να βρούμε πληροφορίες για το ιστορικό των συναλλαγών και των γεγονότων τους. Δοθέντος ενός σήματος S, και μιας τιμής T του τύπου time, η VHDL ορίζει τις ακόλουθες ιδιότητες:

S'delayed(T)	Ένα σήμα που λαμβάνει την ίδια τιμή με το S αλλά είναι καθυστερημένο κατά χρόνο T.
S'stable(T)	Ένα σήμα Boolean που είναι αληθές εάν δεν έχει συμβεί γεγονός στο S κατά το χρονικό διάστημα από την χρονική στιγμή T έως την τρέχουσα χρονική στιγμή, διαφορετικά είναι ψευδές.
S'quiet(T)	Ένα σήμα Boolean που είναι αληθές εάν δεν έχει συμβεί συναλλαγή στο S κατά το χρονικό διάστημα από την χρονική στιγμή T έως την τρέχουσα χρονική στιγμή, διαφορετικά είναι ψευδές.
S'transaction	Ένα σήμα τύπου bit που αλλάζει τιμή από '0' σε '1' ή αντίστροφα κάθε φορά που συμβαίνει μία συναλλαγή στο S.
S'event	Αληθές εάν υπάρχει ένα γεγονός στο S στον τρέχοντα κύκλο προσομοίωσης, διαφορετικά ψευδές.
S'active	Αληθές εάν υπάρχει μία συναλλαγή στο S στον τρέχοντα κύκλο προσομοίωσης, διαφορετικά ψευδές.
S'last_event	Το χρονικό διάστημα από το τελευταίο γεγονός στο S.
S'last_active	Το χρονικό διάστημα από την τελευταία συναλλαγή στο S.
S'last_value	Η τιμή του S ακριβώς πριν το τελευταίο γεγονός στο S.

Οι πρώτες τρεις ιδιότητες παίρνουν μια προαιρετική χρονική παράμετρο. Εάν παραλείψουμε την παράμετρο, η τιμή 0 fs συμπεραίνεται εκ προοιμίου. Αυτές οι ιδιότητες χρησιμοποιούνται συχνά στον έλεγχο της συμπεριφοράς χρονισμού μέσα σε ένα μοντέλο. Για παράδειγμα, μπορούμε να επαληθεύσουμε ότι ένα σήμα d καλύπτει την απαίτηση για ελάχιστο χρόνο προπαρασκευής (setup time) Tsu πριν από μια ανοδική άκρη ενός ρολογιού clk τύπου std_ulogic ως εξής:

```
if clk'event and (clk = '1' or clk = 'H')
        and (clk'last_value = '0' or clk'last_value = 'L') then
    assert d'last_event >= Tsu
        report "Timing error: d changed within setup time of clk";
end if;
```

Ομοίως, θα μπορούσαμε να ελέγξουμε ότι το πλάτος του παλμού μιας εισόδου σήματος ρολογιού σε μια λειτουργική μονάδα δεν υπερβαίνει μια μέγιστη συχνότητα δοκιμάζοντας το πλάτος του παλμού:

```
assert (not clk'event) or clk'delayed'last_event >= Tpw_clk
    report "Clock frequency too high";
```

Σημειώστε ότι εξετάζουμε το χρόνο από το τελευταίο γεγονός σε μια καθυστερημένη έκδοση του σήματος ρολογιού. Όταν υπάρχει σ' αυτήν την περίοδο ένα γεγονός σε ένα σήμα, η ιδιότητα 'last_event επιστρέφει την τιμή 0 fs. Σε αυτήν την περίπτωση, προσδιορίζουμε το χρόνο από το προηγούμενο γεγονός εφαρμόζοντας την ιδιότητα 'last_event στο σήμα καθυστερημένο κατά 0 fs. Μπορούμε να το σκεφτούμε ως μια απειροελάχιστη καθυστέρηση. Θα επιστρέψουμε σε αυτήν την ιδέα αργότερα σε αυτό το κεφάλαιο, στη συζήτησή μας για τις καθυστερήσεις δέλτα (delta delays).

ΠΑΡΑΔΕΙΓΜΑ

Μπορούμε να χρησιμοποιήσουμε μια παρόμοια δοκιμή για την ανοδική ακμή ενός σήματος ρολογιού για να μοντελοποιήσουμε μια λειτουργική μονάδα ενεργοποιούμενη σε ακμή, όπως ένα φλιπ-φλοπ (flip-flop). Το φλιπ-φλοπ πρέπει να φορτώνει την τιμή της εισόδου του D σε μια ανοδική ακμή του clk, αλλά να καθαρίζει ασύγχρονα τις εξόδους όποτε το clr γίνει '1'. Η δήλωση της οντότητας και το σώμα μίας αρχιτεκτονικής συμπεριφοράς παρουσιάζονται στην Εικόνα 5-5.

ΕΙΚΟΝΑ 5-5

```
entity edge_triggered_Dff is
     port ( D : in bit; clk : in bit; clr : in bit;
            Q : out bit );
end entity edge_triggered_Dff;
```

```
architecture behavioral of edge_triggered_Dff is
begin
    state_change : process (clk, clr) is
    begin
        if clr = '1' then
            Q <= '0' after 2 ns;
        elsif clk'event and clk = '1' then
            Q <= D after 2 ns;
        end if;
    end process state_change;
end architecture behavioral;
```

Η οντότητα και το σώμα αρχιτεκτονικής για ένα φλιπ-φλοπ ενεργοποιούμενο σε ακμή, που χρησιμοποιεί την ιδιότητα 'event *για να ελέγξει τις αλλαγές στο σήμα* clk.

Εάν το φλιπ-φλοπ δεν είχε την ασύγχρονη είσοδο καθαρισμού (asynchronous clear input), το μοντέλο θα μπορούσε να χρησιμοποιεί μια απλή πρόταση wait όπως

```
wait until clk = '1';
```

για να ενεργοποιείται σε μια ανοδική ακμή. Εντούτοις, με την παρουσία της εισόδου καθαρισμού, η διεργασία πρέπει να είναι ευαίσθητη στις αλλαγές και στο clk και στο clr οποιαδήποτε στιγμή. Ως εκ τούτου χρησιμοποιεί την ιδιότητα 'event για να διακρίνει μεταξύ της αλλαγής του clk σε '1' και της επιστροφής του clr σε '0' ενώ το clk είναι σταθερό στο '1'.

VHDL-87

Στη VHDL-87, η ιδιότητα 'last_value για ένα σύνθετο σήμα επιστρέφει τη συνάθροιση των προηγουμένων τιμών για κάθε ένα από τα βαθμωτά στοιχεία του σήματος. Για παράδειγμα, υποθέστε ότι ένα σήμα s διανύσματος bit έχει αρχικά την τιμή B"00" και αλλάζει σε B"01" και έπειτα σε B"11" σε διαδοχικά γεγονότα. Μετά από το τελευταίο γεγονός, το αποτέλεσμα του s'last_value είναι B"00" στη VHDL-87. Στη VHDL-93 και στη VHDL-2001 είναι B"01", δεδομένου ότι αυτή είναι η τελευταία τιμή του συνολικού σύνθετου σήματος.

5.3.3 Προτάσεις Wait

Τώρα που έχουμε δει πώς να αλλάζουμε τις τιμές των σημάτων κατά τη διάρκεια του χρόνου, το επόμενο βήμα στη μοντελοποίηση συμπεριφοράς είναι να διευκρινίσουμε πότε οι διεργασίες ανταποκρίνονται στις αλλαγές στις τιμές των σημάτων. Αυτό γίνεται χρησιμοποιώντας *προτάσεις wait (wait statements)*. Μια πρόταση wait είναι μια ακολουθιακή πρόταση με τον ακόλουθο συντακτικό κανόνα:

wait_statement ⇐

[label :] **wait** [**on** signal_name { , … }]

[**until** boolean_expression]

[**for** time_expression] ;

Η προαιρετική ετικέτα μας επιτρέπει να προσδιορίσουμε την πρόταση. Ο σκοπός της πρότασης wait είναι να προκαλέσουμε τη διεργασία που εκτελεί την πρόταση να αναστείλει την εκτέλεση. Η *φράση ευαισθησίας (sensitivity clause)*, η *φράση συνθήκης (condition clause)* και η *φράση χρονικής υπέρβασης (timeout clause)* προσδιορίζουν πότε η διεργασία θα συνεχίσει την εκτέλεση. Μπορούμε να συμπεριλάβουμε οποιοδήποτε συνδυασμό αυτών των φράσεων, ή να παραλείψουμε και τις τρεις. Ας δούμε κάθε φράση ξεχωριστά και να περιγράψουμε τι προσδιορίζει.

Η φράση ευαισθησίας, αρχίζοντας από τη λέξη **on**, μας επιτρέπει να προσδιορίσουμε έναν κατάλογο σημάτων στα οποία η διεργασία αποκρίνεται. Εάν απλά συμπεριλάβουμε μια φράση ευαισθησίας σε μια πρόταση wait, η διεργασία θα συνεχίσει την εκτέλεση όποτε οποιοδήποτε από τα σήματα της λίστας αλλάξει τιμή, δηλαδή όποτε ένα γεγονός συμβεί σε οποιαδήποτε από τα σήματα. Αυτό το στυλ της πρότασης wait είναι χρήσιμο σε μια διεργασία που μοντελοποιεί έναν μπλοκ συνδυαστικής λογικής, δεδομένου ότι οποιαδήποτε αλλαγή στις εισόδους μπορεί να οδηγήσει σε νέες τιμές εξόδου. Για παράδειγμα:

```
half_add : process is
begin
    sum <= a xor b after T_pd;
    carry <= a and b after T_pd;
    wait on a, b;
end process half_add;
```

Η διεργασία αρχίζει την εκτέλεση με την παραγωγή τιμών για τα sum και carry βασιζόμενη στις αρχικές τιμές των a και b, κατόπιν αναστέλλει την εκτέλεση στην πρόταση wait έως ότου αλλάξει τιμή είτε το a είτε το b (είτε και τα δύο). Όταν συμβεί αυτό, η διεργασία συνεχίζει και αρχίζει την εκτέλεση από την κορυφή.

Αυτή η μορφή διεργασίας είναι τόσο κοινή στη μοντελοποίηση ψηφιακών συστημάτων ώστε η VHDL παρέχει τη σημειογραφία συντόμευσης που έχουμε δει σε πολλά παραδείγματα στα προηγούμενα κεφάλαια. Μια διεργασία με μία λίστα ευαισθησίας στην επικεφαλίδα της είναι ακριβώς ισοδύναμη με μια διεργασία με μια πρόταση wait στο τέλος, που περιέχει μια φράση ευαισθησίας που ονομάζει τα σήματα στη λίστα ευαισθησίας. Έτσι η παραπάνω διεργασία half_add θα μπορούσε να ξαναγραφεί ως εξής

```
half_add : process (a, b) is
begin
    sum <= a xor b after T_pd;
    carry <= a and b after T_pd;
end process half_add;
```

ΠΑΡΑΔΕΙΓΜΑ

Ας επιστρέψουμε στο μοντέλο ενός πολυπλέκτη δύο-εισόδων που παρουσιάστηκε στην Εικόνα 5-4. Η διεργασία σε εκείνο το μοντέλο είναι ευαίσθητη σε όλα τα τρία σήματα εισόδου. Αυτό σημαίνει ότι θα συνεχίσει την εκτέλεση στις αλλαγές σε οποιαδήποτε είσοδο δεδομένων, ακόμα και αν μόνο μια από αυτές επιλέγεται οποιαδήποτε στιγμή. Εάν ενδιαφερόμαστε για αυτήν τη μικρή έλλειψη αποδοτικότητας στην προσομοίωση, μπορούμε να γράψουμε τη διεργασία διαφορετικά, χρησιμοποιώντας προτάσεις wait για να είναι πιο επιλεκτική στα σήματα στα οποία η διεργασία είναι ευαίσθητη κάθε φορά που αναστέλλει την εκτέλεσή της. Το αναθεωρημένο μοντέλο παρουσιάζεται στην Εικόνα 5-6. Σε αυτό το μοντέλο, όταν έχει επιλεγεί η είσοδος a, η διεργασία αναμένει μόνο τις αλλαγές στην είσοδο επιλογής και στην είσοδο a. Οποιεσδήποτε αλλαγές στην είσοδο b αγνοούνται. Ομοίως, εάν έχει επιλεγεί η είσοδος b, η διεργασία αναμένει τις αλλαγές στις εισόδους sel και b, αγνοώντας τις αλλαγές στην είσοδο a.

ΕΙΚΟΝΑ 5-6

```
entity mux2 is
     port ( a, b, sel : in bit;
            z : out bit );
end entity mux2;
```

```
architecture behavioral of mux2 is
    constant prop_delay : time := 2 ns;
begin
    slick_mux : process is
    begin
        case sel is
            when '0' =>
                z <= a after prop_delay;
                wait on sel, a;
            when '1' =>
                z <= b after prop_delay;
                wait on sel, b;
        end case;
    end process slick_mux;
end architecture behavioral;
```

Η οντότητα και το σώμα αρχιτεκτονικής για έναν πολυπλέκτη που αποφεύγει τη συνέχιση της εκτέλεσής του σε απόκριση στις αλλαγές στο σήμα εισόδου που δεν έχει επιλεγεί την τρέχουσα χρονική στιγμή.

Η φράση συνθήκης σε μια δήλωση wait, αρχίζοντας από τη λέξη **until**, μας επιτρέπει να καθορίσουμε μία συνθήκη που πρέπει να είναι αληθής ώστε η διεργασία να ξαναρχίσει την εκτέλεσή της. Για παράδειγμα, η δήλωση wait

```
wait until clk = '1';
```

αναγκάζει τη διεργασία να αναστείλει την εκτέλεσή της έως ότου η τιμή του σήματος clk αλλάξει σε '1'. Η παράσταση της συνθήκης εξετάζεται ενώ η διεργασία έχει ανασταλεί για να αποφασίσει εάν θα ξαναρχίσει την

εκτέλεση της διεργασίας. Επακόλουθο αυτού είναι ότι ακόμα κι αν η συνθήκη είναι αληθής όταν η πρόταση wait εκτελείται, η διεργασία θα αναστείλει την εκτέλεση της μέχρι τα κατάλληλα σήματα αλλάξουν και προκαλέσουν την συνθήκη να είναι πάλι αληθής. Εάν η πρόταση wait δεν περιλαμβάνει μια φράση ευαισθησίας, η συνθήκη εξετάζεται όποτε ένα γεγονός συμβεί σε οποιοδήποτε από τα σήματα που αναφέρονται στη συνθήκη.

ΠΑΡΑΔΕΙΓΜΑ

Η διεργασία γεννήτριας ρολογιού μπορεί να ξαναγραφεί χρησιμοποιώντας μια πρόταση wait με μια φράση συνθήκης, όπως φαίνεται στην Εικόνα 5-7. Κάθε φορά που η διεργασία εκτελεί την πρόταση wait, το clk έχει την τιμή ‘0’. Εντούτοις, η διεργασία είναι ακόμα σε αναστολή, και η συνθήκη εξετάζεται κάθε φορά που υπάρχει ένα γεγονός στο clk. Όταν το clk αλλάξει σε ‘1’, τίποτα δεν συμβαίνει, αλλά όταν αλλάξει σε ‘0’ πάλι, η διεργασία συνεχίζει την εκτέλεσή της και χρονοπρογραμματίζει τις συναλλαγές για τον επόμενο κύκλο.

ΕΙΚΟΝΑ 5-7

```
clock_gen : process is
begin
    clk <= '1' after T_pw, '0' after 2*T_pw;
    wait until clk = '0';
end process clock_gen;
```

Η αναθεωρημένη διεργασία γεννήτριας ρολογιού.

Εάν μια πρόταση wait συμπεριλαμβάνει μια φράση ευαισθησίας καθώς επίσης και μια φράση συνθήκης, η συνθήκη εξετάζεται μόνο όταν εμφανίζεται ένα γεγονός σε οποιοδήποτε από τα σήματα στη φράση ευαισθησίας. Για παράδειγμα, εάν μια διεργασία αναστείλει την εκτέλεσή της στην ακόλουθη πρόταση wait:

```
wait on clk until reset = '0';
```

η συνθήκη εξετάζεται σε κάθε αλλαγή στην τιμή του clk, ανεξάρτητα από οποιεσδήποτε αλλαγές στο reset.

Η φράση χρονικής υπέρβασης σε μια πρόταση wait, αρχίζοντας από τη λέξη **for**, μας επιτρέπει να προσδιορίσουμε ένα μέγιστο διάστημα του χρόνου προσομοίωσης για το οποίο η διεργασία πρέπει να ανασταλεί. Εάν επίσης συμπεριλάβουμε μια φράση ευαισθησίας ή συνθήκης, αυτές μπορεί να αναγκάσουν τη διεργασία να συνεχίσει την εκτέλεσή της νωρίτερα. Για παράδειγμα, η πρόταση wait

```
wait until trigger = '1' for 1 ms;
```

αναγκάζει την εκτελούσα διεργασία να αναστείλει την εκτέλεσή της μέχρι το trigger να αλλάξει σε ‘1’, ή μέχρι να παρέλθει 1 ms χρόνος προσομοίωσης, οποιοδήποτε από τα δύο συμβεί πρώτο. Εάν απλά συμπεριλάβουμε μια φράση χρονικής υπέρβασης μόνη της σε μια πρόταση wait, η διεργασία θα αναστείλει την εκτέλεσή της για το δοθέντα χρόνο.

ΠΑΡΑΔΕΙΓΜΑ

Μπορούμε να ξαναγράψουμε τη διεργασία γεννήτριας ρολογιού ακόμη μία φορά, χρησιμοποιώντας αυτή τη φορά μια πρόταση wait με μια φράση χρονικής υπέρβασης, όπως φαίνεται στην Εικόνα 5-8. Σε αυτήν την περίπτωση καθορίζουμε την περίοδο του ρολογιού ως τη χρονική υπέρβαση, μετά από την οποία η διεργασία πρόκειται να συνεχίσει την εκτέλεσή της.

ΕΙΚΟΝΑ 5-8

```
clock_gen : process is
begin
    clk <= '1' after T_pw, '0' after 2*T_pw;
    wait for 2*T_pw;
end process clock_gen;
```

Μία τρίτη έκδοση της διεργασίας γεννήτριας ρολογιού.

Εάν επιστρέψουμε στο συντακτικό κανόνα για μια πρόταση wait που παρουσιάστηκε προηγουμένως, διαπιστώνουμε ότι είναι νόμιμο να γράψουμε

```
wait;
```

Αυτή η μορφή προκαλεί τη διεργασία να αναστείλει την εκτέλεσή της για το υπόλοιπο της προσομοίωσης. Αν και αυτό μπορεί αρχικά να φανεί παράξενο να θελήσει κάποιος να το κάνει, στην πράξη είναι αρκετά χρήσιμο. Μια περίπτωση όπου χρησιμοποιείται είναι σε μία διεργασία της οποίας σκοπός είναι να παράγει ερεθίσματα (stimuli) για μια προσομοίωση. Μια τέτοια διεργασία πρέπει να παράγει μια ακολουθία συναλλαγών σε σήματα που συνδέονται με άλλα μέρη ενός μοντέλου και μετά να σταματήσει. Για παράδειγμα, η διεργασία

```
test_gen : process is
begin
    test0 <= '0' after 10 ns, '1' after 20 ns, '0' after 30 ns, '1' after 40 ns;
    test1 <= '0' after 10 ns, '1' after 30 ns;
    wait;
end process test_gen;
```

παράγει και τους τέσσερις πιθανούς συνδυασμούς τιμών στα σήματα test0 και test1. Εάν η τελική πρόταση wait είχε παραληφθεί, η διεργασία θα επαναλάμβανε την εκτέλεσή της για πάντα, επαναλαμβάνοντας τις προτάσεις ανάθεσης σημάτων χωρίς να διακόπτεται, και η προσομοίωση δεν θα σημείωνε καμία πρόοδο.

VHDL-87

Οι προτάσεις wait δεν επιτρέπεται να έχουν ετικέτες στη VHDL-87.

5.3.4 Καθυστερήσεις Δέλτα

Ας επιστρέψουμε τώρα στο θέμα των καθυστερήσεων στις αναθέσεις σημάτων. Σε πολλά από τα παραδείγματα ανάθεσης σήματος στα προηγούμενα κεφάλαια, παραλείψαμε το μέρος της καθυστέρησης των στοιχείων κυματομορφής. Αυτό είναι ισοδύναμο με τον προσδιορισμό μιας καθυστέρησης των 0 fs. Η τιμή πρόκειται να εφαρμοστεί στο σήμα στον τρέχοντα χρόνο προσομοίωσης. Εντούτοις, είναι σημαντικό να σημειωθεί ότι η τιμή του σήματος δεν αλλάζει μόλις εκτελείται η πρόταση ανάθεσης σήματος. Αντίθετα, η ανάθεση χρονοπρογραμματίζει μια συναλλαγή για το σήμα, η οποία εφαρμόζεται αφότου ανασταλεί η διεργασία. Κατά συνέπεια η διεργασία δεν «βλέπει» την επίδραση της ανάθεσης μέχρι την επόμενη φορά που θα επαναλάβει την εκτέλεσή της, ακόμα κι αν αυτό είναι στον ίδιο χρόνο προσομοίωσης. Για αυτόν το λόγο, μια καθυστέρηση 0 fs σε μια ανάθεση σήματος καλείται *καθυστέρηση δέλτα (delta delay)*.

Για να κατανοήσει κανείς γιατί οι καθυστερήσεις δέλτα δουλεύουν κατ' αυτόν τον τρόπο, είναι απαραίτητο να επανεξετάσει τον κύκλο προσομοίωσης, που εισάγαμε στο Κεφάλαιο 1. Θυμηθείτε ότι ο κύκλος προσομοίωσης αποτελείται από δύο φάσεις: μια φάση ενημέρωσης του σήματος που ακολουθείται από μια φάση εκτέλεσης της διεργασίας. Στη φάση ενημέρωσης του σήματος, ο χρόνος προσομοίωσης προχωράει έως τη χρονική στιγμή της νωρίτερα χρονοπρογραμματισμένης συναλλαγής, και για όλες τις συναλλαγές που είναι χρονοπρογραμματισμένες για αυτήν τη χρονική στιγμή οι τιμές εφαρμόζονται στα αντίστοιχα σήματα Αυτό μπορεί να προκαλέσει να συμβούν γεγονότα σε μερικά σήματα. Στη φάση εκτέλεσης της διεργασίας, όλες οι διεργασίες που αποκρίνονται σε αυτά τα γεγονότα ξαναρχίζουν την εκτέλεσή τους έως ότου ανασταλούν πάλι στις προτάσεις wait. Ο προσομοιωτής έπειτα επαναλαμβάνει τον κύκλο προσομοίωσης.

Ας εξετάσουμε τώρα τι συμβαίνει όταν μια διεργασία εκτελέσει μια πρόταση ανάθεσης σήματος με καθυστέρηση δέλτα, για παράδειγμα:

```
data <= X"00";
```

Υποθέστε ότι αυτή εκτελείται στο χρόνο προσομοίωσης *t* κατά τη διάρκεια της φάσης εκτέλεσης της διεργασίας του τρέχοντος κύκλου προσομοίωσης. Η επίδραση της ανάθεσης είναι να χρονοπρογραμματιστεί μια συναλλαγή που θέτει τη τιμή X"00" στο σήμα data στο χρόνο *t*. Η συναλλαγή δεν εφαρμόζεται αμέσως, αφού ο προσομοιωτής είναι στη φάση εκτέλεσης της διεργασίας. Ως εκ τούτου η διεργασία συνεχίζει, με το σήμα data αμετάβλητο. Όταν όλες οι διεργασίες έχουν ανασταλεί, ο προσομοιωτής αρχίζει τον επόμενο κύκλο προσομοίωσης και ενημερώνει το χρόνο προσομοίωσης. Δεδομένου ότι η νωρίτερη συναλλαγή είναι τώρα στο χρόνο *t*, ο χρόνος προσομοίωσης παραμένει αμετάβλητος. Ο προσομοιωτής εφαρμόζει τώρα την τιμή X"00" στη χρονοπρογραμματισμένη συναλλαγή στο σήμα data, και κατόπιν ξαναρχίζει την εκτέλεση όλων των διεργασιών που αποκρίνονται στη νέα τιμή.

Το γράψιμο ενός μοντέλου με καθυστερήσεις δέλτα είναι χρήσιμο όταν εργαζόμαστε σε υψηλό επίπεδο αφαίρεσης και δεν ενδιαφερόμαστε ακόμα για το λεπτομερή χρονισμό. Εάν το μόνο που ενδιαφερόμαστε είναι να περιγράψουμε τη σειρά με την οποία εκτελούνται οι λειτουργίες, οι καθυστερήσεις δέλτα παρέχουν έναν τρόπο για να αγνοήσουμε τα «μπλεξίματα» με το χρονισμό. Αυτό το έχουμε δει σε πολλά από τα παραδείγματα των προηγούμενων κεφαλαίων. Εντούτοις, πρέπει να σημειώσουμε μια κοινή παγίδα που πέφτουν οι περισσότεροι αρχάριοι σχεδιαστές στη VHDL κατά τη χρησιμοποίηση των καθυστερήσεων δέλτα: ξεχνούν ότι η διεργασία δεν βλέπει την επίδραση της ανάθεσης αμέσως. Για παράδειγμα, μπορεί να γράψουμε μια διεργασία που περιλαμβάνει τις ακόλουθες προτάσεις:

```
s <= '1';
…
if s = '1' then …
```

και να αναμένουμε η διεργασία να εκτελέσει την πρόταση if υποθέτοντας ότι το σήμα s έχει την τιμή '1'. Τότε θα περνούσαμε άκαρπες ώρες διορθώνοντας το μοντέλο μας έως ότου θυμόμασταν ότι το s έχει ακόμα την παλιά τιμή του μέχρι τον επόμενο κύκλο προσομοίωσης, μετά από την αναστολή της διεργασίας.

ΠΑΡΑΔΕΙΓΜΑ

Η Εικόνα 5-9 είναι το περίγραμμα ενός αφηρημένου μοντέλου ενός υπολογιστικού συστήματος. Η Κεντρική Μονάδα Επεξεργασίας (ΚΜΕ) και η μνήμη συνδέονται με τα σήματα διευθύνσεων και δεδομένων. Συγχρονίζουν τη λειτουργία τους με τα σήματα ελέγχου mem_read και mem_write και το σήμα κατάστασης mem_ready. Καμία καθυστέρηση δεν προσδιορίζεται στις προτάσεις ανάθεσης σημάτων, έτσι ο συγχρονισμός επιτυγχάνεται μετά από έναν αριθμό κύκλων καθυστέρησης δέλτα, όπως φαίνεται στην Εικόνα 5-10.

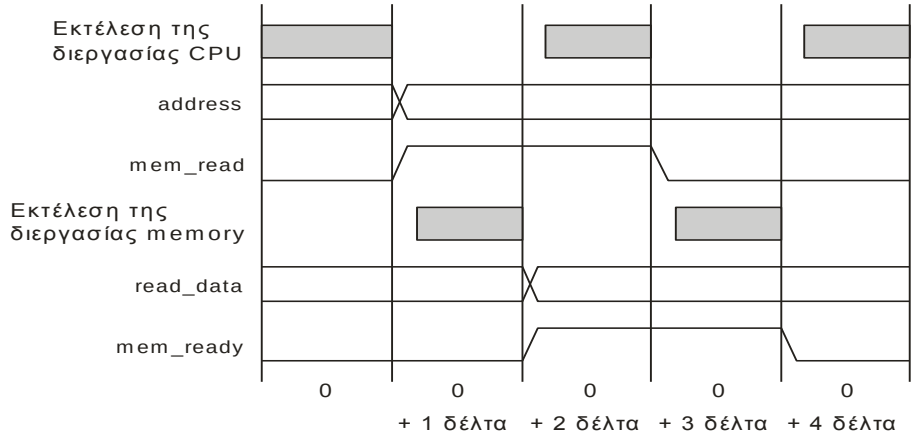
ΕΙΚΟΝΑ 5-9

```
architecture abstract of computer_system is
    subtype word is bit_vector(31 downto 0);
    signal address : natural;
    signal read_data, write_data : word;
    signal mem_read, mem_write : bit := '0';
    signal mem_ready : bit := '0';
begin
    cpu : process is
        variable instr_reg : word;
        variable PC : natural;
        …       -- άλλες δηλώσεις
    begin
        loop
            address <= PC;
            mem_read <= '1';
            wait until mem_ready = '1';
            instr_reg := read_data;
            mem_read <= '0';
            wait until mem_ready = '0';
            PC := PC + 4;
            …          -- εκτελεί την εντολή
        end loop;
end process cpu;
    memory : process is
        type memory_array is array (0 to 2**14 – 1) of word;
        variable store : memory_array := (
            …
            );
    begin
        wait until mem_read = '1' or mem_write = '1';
        if mem_read = '1' then
            read_data <= store( address / 4 );
            mem_ready <= '1';
            wait until mem_read = '0';
            mem_ready <= '0';
        else
            …          -- διεξάγει προσπέλαση εγγραφής
        end if;
    end process memory;
end architecture abstract;
```

Το περίγραμμα ενός αφηρημένου μοντέλου ενός υπολογιστικού συστήματος, που αποτελείται από μια ΚΜΕ και μια μνήμη. Οι διεργασίες χρησιμοποιούν καθυστερήσεις δέλτα για να συγχρονίσουν την επικοινωνία, και όχι λεπτομερή χρονική μοντελοποίηση των συναλλαγών του διαύλου.

Όταν αρχίζει η προσομοίωση, η διεργασία CPU αρχίζει την εκτέλεση των προτάσεων και η διεργασία memory αναστέλλεται. Η διεργασία CPU χρονοπρογραμματίζει συναλλαγές για να εκχωρήσει τη διεύθυνση της επόμενης εντολής στο σήμα address και την τιμή ‘1’ στο σήμα mem_read, και έπειτα αναστέλλεται. Στον επόμενο κύκλο προσομοίωσης, αυτά τα σήματα ενημερώνονται και η διεργασία memory ξαναρχίζει την εκτέλεσή της, αφού περιμένει ένα γεγονός στο σήμα mem_read. Η διεργασία memory χρονοπρογραμματίζει τα δεδομένα στο σήμα read_data και την τιμή ‘1’ στο σήμα mem_ready, και έπειτα αναστέλλεται. Στον τρίτο κύκλο, αυτά τα σήματα ενημερώνονται και η διεργασία CPU ξαναρχίζει την εκτέλεσή της. Χρονοπρογραμματίζει την τιμή ‘0’ στο σήμα mem_read και αναστέλλεται. Κατόπιν, στον τέταρτο κύκλο, το σήμα mem_read ενημερώνεται και η διεργασία memory ξαναρχίζει την εκτέλεσή της, χρονοπρογραμματίζοντας την τιμή ‘0’ στο σήμα mem_ready για να ολοκληρώσει τη χειραψία (handshake). Τέλος, στον πέμπτο κύκλο, το σήμα mem_ready ενημερώνεται και η διεργασία CPU ξαναρχίζει την εκτέλεσή της και εκτελεί την εντολή που έχει προσκομίσει.

ΕΙΚΟΝΑ 5-10



Συγχρονισμός σε διαδοχικούς κύκλους δέλτα στην προσομοίωση μίας λειτουργίας ανάγνωσης μεταξύ της ΚΜΕ και της μνήμης που φαίνεται στην Εικόνα 5-9.

5.3.5 Μηχανισμοί Καθυστέρησης Μεταφοράς και Αδράνειας

Μέχρι τώρα στη συζήτησή μας για τις αναθέσεις σημάτων, έχουμε υποθέσει σιωπηρά ότι δεν υπήρχε καμία εκκρεμής συναλλαγή που να ήταν ήδη χρονοπρογραμματισμένη για ένα σήμα όταν εκτελούνταν μια πρόταση ανάθεσης σήματος. Σε πολλά μοντέλα, ιδιαίτερα σε πιο υψηλά επίπεδα αφαίρεσης, αυτό είναι συνηθισμένο. Εάν, από την άλλη μεριά, υπάρχουν εκκρεμείς συναλλαγές, οι νέες συναλλαγές συγχωνεύονται με αυτές με έναν τρόπο που εξαρτάται από το *μηχανισμό καθυστέρησης (delay mechanism)* που χρησιμοποιείται στην πρόταση ανάθεσης σήματος. Αυτό είναι ένα προαιρετικό μέρος της σύνταξης ανάθεσης σήματος. Ο συντακτικός κανόνας για το μηχανισμό καθυστέρησης είναι

delay_mechanism ⇐ **transport** I [**reject** time_expression] **inertial**

Σε μια ανάθεση σήματος το να παραλείπεται ο μηχανισμός καθυστέρησης είναι ισοδύναμο με το να καθορίζεται ο μηχανισμός **inertial**. Εξετάζουμε πρώτα το μηχανισμό καθυστέρησης *μεταφοράς (transport)*, δεδομένου ότι είναι απλούστερος, και στη συνέχεια επιστρέφουμε στο μηχανισμό καθυστέρησης *αδράνειας (inertial)*.

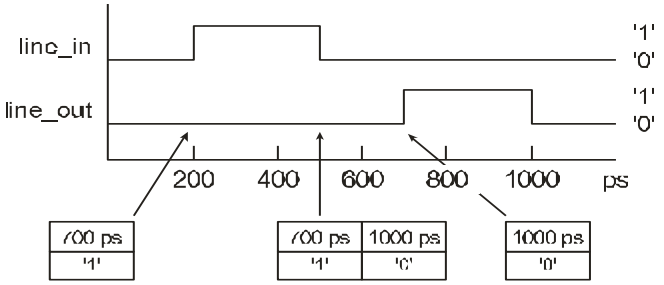
Χρησιμοποιούμε το μηχανισμό καθυστέρησης μεταφοράς όταν μοντελοποιούμε μια ιδανική συσκευή με άπειρη απόκριση συχνότητας, στην οποία οποιοσδήποτε παλμός εισόδου, ανεξάρτητα από το πόσο περιορισμένη είναι η διάρκειά του, παράγει έναν παλμό εξόδου. Ένα παράδειγμα μιας τέτοιας συσκευής είναι μια ιδανική γραμμή μετάδοσης, η οποία μεταδίδει όλες τις αλλαγές στην είσοδο καθυστερημένες κατά μία ποσότητα χρόνου. Μια διεργασία που μοντελοποιεί μια γραμμή μετάδοσης με καθυστέρηση 500 ps είναι

```
transmission_line : process (line_in) is
begin
    line_out <= transport line_in after 500 ps;
end process transmission_line;
```

Σε αυτό το μοντέλο η έξοδος ακολουθεί οποιεσδήποτε αλλαγές στην είσοδο, αλλά καθυστερημένες κατά 500 ps. Εάν η είσοδος αλλάξει δύο φορές ή περισσότερες εντός μιας χρονικής περιόδου μικρότερης από 500 ps, οι χρονοπρογραμματισμένες συναλλαγές απλά τοποθετούνται από τον οδηγό στην ουρά και παραμένουν σε αυτήν μέχρι τη χρονική στιγμή της προσομοίωσης στην οποία πρόκειται να εφαρμοστούν, όπως φαίνεται στην Εικόνα 5-11.

Σε αυτό το παράδειγμα, κάθε νέα συναλλαγή που παράγεται από μια πρόταση ανάθεσης σήματος χρονοπρογραμματίζεται για ένα χρόνο προσομοίωσης που είναι μεταγενέστερος από τις εκκρεμείς συναλλαγές που τοποθετήθηκαν από τον οδηγό στην ουρά. Η κατάσταση γίνεται λίγο πιο σύνθετη όταν χρησιμοποιούνται μεταβλητές καθυστερήσεις, δεδομένου ότι μπορούμε να χρονοπρογραμματίσουμε μια συναλλαγή για μια χρονική στιγμή προγενέστερη από μια εκκρεμή συναλλαγή. Η σημασιολογία του μηχανισμού καθυστέρησης μεταφοράς καθορίζει ότι εάν υπάρχουν εκκρεμείς συναλλαγές σε έναν οδηγό που είναι χρονοπρογραμματισμένες για μια χρονική στιγμή μεταγενέστερη ή ίση με μια νέα συναλλαγή, εκείνες οι πιο μεταγενέστερες συναλλαγές διαγράφονται.

ΕΙΚΟΝΑ 5-11



Συναλλαγές που τοποθετούνται από έναν οδηγό στην ουρά και χρησιμοποιούν καθυστέρηση μεταφοράς. Στη χρονική στιγμή 200 ps η είσοδος αλλάζει και μια συναλλαγή χρονοπρογραμματίζεται για τα 700 ps. Στη χρονική στιγμή 500 ps, η είσοδος αλλάζει πάλι, και μια άλλη συναλλαγή χρονοπρογραμματίζεται για τα 1000 ps. Αυτή τοποθετείται από τον οδηγό στην ουρά πίσω από την προηγούμενη συναλλαγή. Όταν ο χρόνος προσομοίωσης φθάσει στα 700 ps, η πρώτη συναλλαγή εφαρμόζεται, και η δεύτερη συναλλαγή παραμένει στην ουρά. Τέλος, ο χρόνος προσομοίωσης φθάνει στα 1000 ps, και η τελευταία συναλλαγή εφαρμόζεται, αφήνοντας την ουρά αναμονής του οδηγού κενή.

ΠΑΡΑΔΕΙΓΜΑ

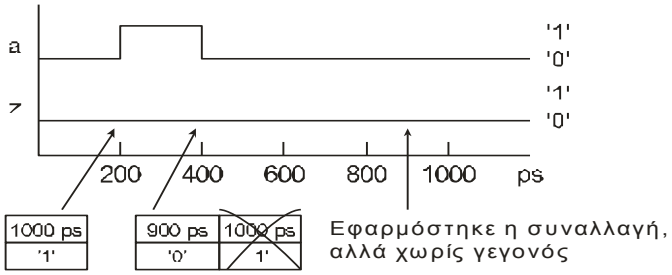
Η Εικόνα 5-12 είναι μια διεργασία που περιγράφει τη συμπεριφορά ενός ασύμμετρου στοιχείου καθυστέρησης, με διαφορετικούς χρόνους καθυστέρησης για τις ανοδικές και καθοδικές μεταβάσεις. Η καθυστέρηση για τις ανοδικές μεταβάσεις είναι 800 ps και για τις καθοδικές μεταβάσεις 500 ps. Εάν εφαρμόζαμε έναν παλμό εισόδου διάρκειας μόνο 200 ps, θα αναμέναμε να μην αλλάξει η έξοδος, αφού η καθυστερημένη καθοδική μετάβαση θα «προσπερνούσε» την καθυστερημένη ανοδική μετάβαση. Εάν απλά προσθέταμε κάθε μετάβαση στην ουρά αναμονής του οδηγού όταν εκτελούνταν μια πρόταση ανάθεσης σήματος, δεν θα παίρναμε αυτήν τη συμπεριφορά. Εντούτοις, η σημασιολογία των μηχανισμού καθυστέρησης μεταφοράς παράγει την επιθυμητή συμπεριφορά, όπως παρουσιάζει η Εικόνα 5-13.

ΕΙΚΟΝΑ 5-12

```
asym_delay : process (a) is
    constant Tpd_01 : time := 800 ps;
    constant Tpd_10 : time := 500 ps;
begin
    if a = '1' then
        z <= transport a after Tpd_01;
    else    -- a = '0'
        z <= transport a after Tpd_10;
    end if;
end process asym_delay;
```

Μία διεργασία που περιγράφει ένα στοιχείο καθυστέρησης με ασύμμετρες καθυστερήσεις για τις ανοδικές και καθοδικές μεταβάσεις.

ΕΙΚΟΝΑ 5-13



Συναλλαγές σε έναν οδηγό που χρησιμοποιεί ασύμμετρη καθυστέρηση μεταφοράς. Στη χρονική στιγμή 200 ps η είσοδος αλλάζει, και μια συναλλαγή χρονοπρογραμματίζεται για τα 1000 ps. Στη χρονική στιγμή 400 ps, η είσοδος αλλάζει πάλι, και μια άλλη συναλλαγή χρονοπρογραμματίζεται για τα 900 ps. Δεδομένου ότι αυτή είναι προγενέστερη από την εκκρεμή συναλλαγή στα 1000 ps, η εκκρεμής συναλλαγή διαγράφεται. Όταν ο χρόνος προσομοίωσης φθάσει στα 900 ps, η εναπομείνασα συναλλαγή εφαρμόζεται, αλλά δεδομένου ότι η τιμή είναι '0', κανένα γεγονός δεν συμβαίνει στο σήμα.

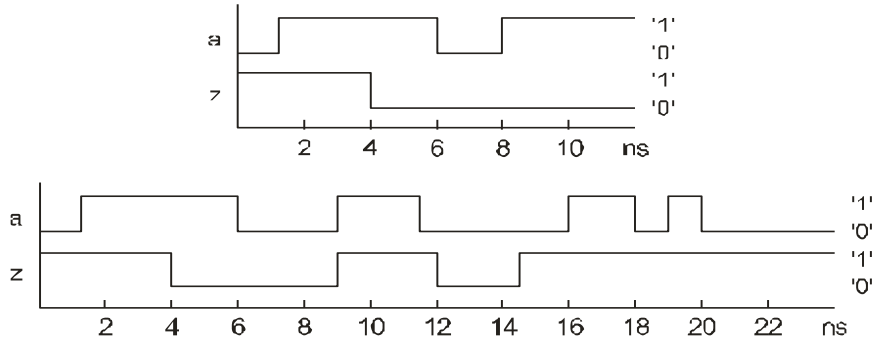
Τα περισσότερα πραγματικά ηλεκτρονικά κυκλώματα δεν έχουν άπειρη απόκριση συχνότητας, έτσι δεν είναι σωστό να τα μοντελοποιούμε χρησιμοποιώντας την καθυστέρηση μεταφοράς. Στις πραγματικές συσκευές, η αλλαγή τιμών στους εσωτερικούς κόμβους και στις εξόδους συνεπάγεται τη μεταφορά ηλεκτρονικού φορτίου με την παρουσία χωρητικότητας (capacitance), αυτεπαγωγής (inductance) και αντίστασης (resistance). Αυτό δίνει στη συσκευή κάποια αδράνεια; τείνει να παραμείνει στην ίδια κατάσταση εκτός αν την αναγκάσουμε εφαρμόζοντας εισόδους για επαρκώς μακρά χρονική διάρκεια. Αυτός είναι ο λόγος που η VHDL περιλαμβάνει το μηχανισμό καθυστέρησης αδράνειας, για να μας επιτρέψει να μοντελοποιήσουμε συσκευές που απορρίπτουν παλμούς εισόδου τόσο σύντομους ώστε να υπερκαλύψουν την αδράνειά τους. Η καθυστέρηση αδράνειας είναι ο μηχανισμός που χρησιμοποιείται εξ' ορισμού σε μια ανάθεση σήματος, ή μπορούμε να την προσδιορίσουμε ρητά γράφοντας τη λέξη **inertial**.

Για να εξηγήσουμε πώς λειτουργεί η καθυστέρηση αδράνειας, ας εξετάσουμε ένα μοντέλο στο οποίο όλες οι αναθέσεις σήματος για ένα δεδομένο σήμα χρησιμοποιούν την ίδια τιμή καθυστέρησης, ας πούμε, 3 ns, όπως στο μοντέλο αντιστροφέα (inverter):

```
inv : process (a) is
begin
    z <= inertial not a after 3 ns;
end process inv;
```

Όσο τα γεγονότα εισόδου συμβαίνουν σε απόσταση μεγαλύτερη των 3 ns, αυτό το μοντέλο δεν παρουσιάζει κανένα πρόβλημα. Κάθε φορά που εκτελείται μια ανάθεση σήματος, δεν υπάρχει καμία εκκρεμής συναλλαγή, έτσι μια νέα συναλλαγή χρονοπρογραμματίζεται, και η έξοδος αλλάζει τιμή μετά από 3 ns. Εντούτοις, εάν μια είσοδος αλλάξει σε λιγότερο από 3 ns μετά από την προηγούμενη αλλαγή, αυτό αντιπροσωπεύει έναν παλμό μικρότερης διάρκειας από την καθυστέρηση διάδοσης της συσκευής, έτσι πρέπει να απορριφθεί. Αυτή η συμπεριφορά εμφανίζεται στην κορυφή της Εικόνας 5-14. Σε ένα απλό μοντέλο όπως αυτό, μπορούμε να ερμηνεύσουμε την καθυστέρηση αδράνειας λέγοντας ότι εάν μια ανάθεση σήματος επρόκειτο να παράγει έναν παλμό εξόδου μικρότερης διάρκειας από την καθυστέρηση διάδοσης, τότε ο παλμός εξόδου δεν συμβαίνει.

ΕΙΚΟΝΑ 5-14



Αποτελέσματα αναθέσεων σημάτων που χρησιμοποιούν το μηχανισμό καθυστέρησης αδράνειας. Στην πάνω κυματομορφή, καθορίζεται μια καθυστέρηση αδράνειας 3 ns. Η αλλαγή της εισόδου τη χρονική στιγμή 1 ns αντανακλάται στην έξοδο στη χρονική στιγμή 4 ns. Ο παλμός με διάρκεια από τα 6 έως τα 8 ns είναι μικρότερος από την καθυστέρηση διάδοσης, έτσι δεν έχει επιπτώσεις στην έξοδο. Στην κάτω κυματομορφή, καθορίζονται μια καθυστέρηση αδράνειας 3 ns και ένα όριο απόρριψης παλμού 2 ns. Οι αλλαγές της εισόδου στα 1, 6, 9 και 11,5 ns όλες αντανακλώνται στην έξοδο, δεδομένου ότι συμβαίνουν σε απόσταση μεγαλύτερη από 2 ns. Εντούτοις, το μήκος των επόμενων παλμών εισόδου είναι μικρότερο ή ίσο με το όριο απόρριψης παλμού, και έτσι δεν έχουν επιπτώσεις στην έξοδο.

Στη συνέχεια, ας επεκτείνουμε αυτό το μοντέλο καθορίζοντας ένα όριο απόρριψης παλμού, μετά από τη λέξη **reject** στην ανάθεση σήματος:

```
inv : process (a) is
begin
    z <= reject 2 ns inertial not a after 3 ns;
end process inv;
```

Μπορούμε να το ερμηνεύσουμε αυτό λέγοντας ότι εάν μια ανάθεση σήματος επρόκειτο να παράγει έναν παλμό εξόδου διάρκειας μικρότερης από (ή ίσης με) το όριο απόρριψης παλμού, τότε ο παλμός εξόδου δεν συμβαίνει. Σε αυτό το απλό μοντέλο, όσο οι αλλαγές στην είσοδο συμβαίνουν σε απόσταση μεγαλύτερη από 2 ns μεταξύ τους, τότε παράγουν αλλαγές στην έξοδο 3 ns αργότερα, όπως παρουσιάζεται στο κάτω μέρος της Εικόνας 5-14. Σημειώστε ότι το όριο απόρριψης παλμού που καθορίζεται πρέπει να είναι μεταξύ 0 fs και της καθυστέρησης που καθορίζεται στην ανάθεση σήματος. Το να παραλείψουμε το όριο απόρριψης παλμού είναι το ίδιο με το να καθορίσουμε ένα όριο ίσο με την καθυστέρηση, και το να καθορίσουμε ένα όριο ίσο με 0 fs είναι το ίδιο με το να προσδιορίσουμε καθυστέρηση μεταφοράς.

Τώρα ας εξετάσουμε την πλήρη ιστορία της καθυστέρησης αδράνειας, που μας επιτρέπει να μεταβάλλουμε το χρόνο καθυστέρησης και το όριο απόρριψης παλμού στις διαφορετικές αναθέσεις σήματος που εφαρμόζονται στο ίδιο σήμα. Όπως με την καθυστέρηση μεταφοράς, η κατάσταση γίνεται πιο πολύπλοκη, και είναι καλύτερο να την περιγράψουμε από την άποψη της διαγραφής συναλλαγών από τον οδηγό. Εκείνοι που δεν πρόκειται να γράψουν μοντέλα που εξετάζουν το χρονισμό σε αυτό το επίπεδο λεπτομέρειας μπορούν αν επιθυμούν να προχωρήσουν στην επόμενη ενότητα.

Μια ανάθεση σήματος με καθυστέρηση αδράνειας εμπεριέχει την εξέταση των εκκρεμών συναλλαγών σε έναν οδηγό κατά τον προσθήκη μιας νέας συναλλαγής. Υποθέστε ότι μια ανάθεση σήματος χρονοπρογραμματίζει μια νέα συναλλαγή για τη χρονική στιγμή t_{new}, με ένα όριο απόρριψης παλμού t_r. Πρώτον, οποιεσδήποτε εκκρεμείς συναλλαγές που είναι χρονοπρογραμματισμένες για χρονική στιγμή μεταγενέστερη ή ίση με t_{new} διαγράφονται, όπως ακριβώς θα συνέβαινε εάν χρησιμοποιούνταν η καθυστέρηση μεταφοράς. Κατόπιν η νέα συναλλαγή προστίθεται στον οδηγό. Δεύτερον, οποιεσδήποτε εκκρεμείς συναλλαγές που είναι χρονοπρογραμματισμένες για το χρονικό διάστημα t_{new} - t_r έως t_{new} εξετάζονται. Εάν υπάρχει μια σειρά διαδοχικών συναλλαγών που προηγούνται άμεσα της νέας συναλλαγής και έχουν την ίδια τιμή με τη νέα συναλλαγή, τότε παραμένουν στον οδηγό. Όλες οι άλλες συναλλαγές στο διάστημα διαγράφονται.

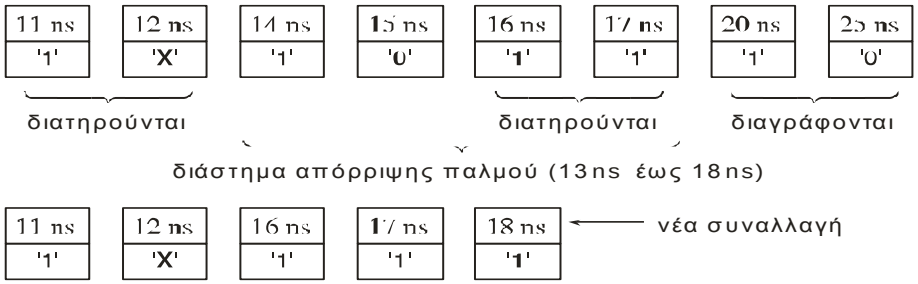
Ένα παράδειγμα θα το καταστήσει πιο σαφές. Υποθέστε ότι ένας οδηγός για το σήμα s περιέχει εκκρεμείς συναλλαγές όπως φαίνεται στο πάνω μέρος της Εικόνας 5-15, και η διεργασία που περιέχει τον οδηγό εκτελεί την παρακάτω πρόταση ανάθεσης σήματος τη χρονική στιγμή 10 ns:

```
s <= reject 5 ns inertial '1' after 8 ns;
```

Οι εκκρεμείς συναλλαγές μετά από αυτήν την ανάθεση φαίνονται στο κάτω μέρος της Εικόνας 5-15.

Ένα τελικό σημείο που πρέπει να σημειώσουμε για τον καθορισμό του μηχανισμού καθυστέρησης σε μια πρόταση ανάθεσης σήματος είναι ότι εάν συμπεριλαμβάνεται ένας αριθμός στοιχείων κυματομορφής, ο καθορισμένος μηχανισμός εφαρμόζεται μόνο στο πρώτο στοιχείο. Όλα τα επόμενα στοιχεία χρονοπρογραμματίζουν τις συναλλαγές χρησιμοποιώντας καθυστέρηση μεταφοράς. Δεδομένου ότι οι καθυστερήσεις για τα πολλαπλά στοιχεία κυματομορφών πρέπει να είναι σε αύξουσα σειρά, αυτό σημαίνει ότι όλες οι συναλλαγές μετά από την πρώτη απλά προστίθενται στην ουρά αναμονής των συναλλαγών του οδηγού με τη σειρά που είναι γραμμένες.

ΕΙΚΟΝΑ 5-15



Συναλλαγές πριν (πάνω μέρος) και μετά (κάτω μέρος) από μια ανάθεση σήματος με καθυστέρηση αδράνειας. Οι συναλλαγές στα 20 και 25 ns διαγράφονται επειδή είναι χρονοπρογραμματισμένες για χρονική στιγμή μεταγενέστερη της νέας συναλλαγής. Εκείνες στα 11 και 12 ns διατηρούνται επειδή βρίσκονται πριν από το διάστημα απόρριψης παλμού. Οι συναλλαγές στα 16 και 17 ns βρίσκονται εντός του διαστήματος απόρριψης, αλλά σχηματίζουν μια σειρά που καταλήγει στη νέα συναλλαγή, με την ίδια τιμή με τη νέα συναλλαγή - ως εκ τούτου διατηρούνται επίσης. Οι άλλες συναλλαγές στο διάστημα απόρριψης διαγράφονται.

ΠΑΡΑΔΕΙΓΜΑ

Ένα λεπτομερές μοντέλο μιας πύλης AND 2-εισόδων παρουσιάζεται στην Εικόνα 5-16. Όταν μια αλλαγή σε κάποιο από τα σήματα εισόδου οδηγεί στο χρονοπρογραμματισμό μιας αλλαγής για την έξοδο, η διεργασία delay προσδιορίζει την καθυστέρηση διάδοσης που θα χρησιμοποιηθεί. Σε μια ανοδική μετάβαση της εξόδου, αιχμές (spikes) μικρότερες από 400 ps απορρίπτονται, ενώ σε μια καθοδική ή άγνωστη μετάβαση, απορρίπτονται αιχμές μικρότερες από 300 ps. Σημειώστε ότι το αποτέλεσμα του τελεστή **and**, όταν εφαρμόζεται σε τιμές πρότυπης λογικής, είναι πάντα 'U', 'X', '0' ή '1'. Ως εκ τούτου η διεργασία delay δεν χρειάζεται να συγκρίνει το αποτέλεσμα με 'X' ή 'L' όταν ελέγχει για ανοδικές ή καθοδικές μεταβάσεις.

ΕΙΚΟΝΑ 5-16

```
library ieee; use ieee.std_logic_1164.all;
entity and2 is
    port ( a, b : in std_ulogic; y : out std_ulogic );
end entity and2;
```

```
architecture detailed_delay of and2 is
    signal result : std_ulogic;
begin
    gate : process (a, b) is
    begin
        result <= a and b;
    end process gate;
    delay : process (result) is
    begin
        if result = '1' then
            y <= reject 400 ps inertial '1' after 1.5 ns;
        elsif result = '0' then
            y <= reject 300 ps inertial '0' after 1.2 ns;
        else
            y <= reject 300 ps inertial 'X' after 500 ps;
        end if;
    end process delay;
end architecture detailed_delay;
```

Μια οντότητα και το σώμα αρχιτεκτονικής για μια πύλη AND 2-εισόδων. Η διεργασία gate *υλοποιεί τη λογική συνάρτηση της οντότητας, και η διεργασία* delay *υλοποιεί τα λεπτομερή χαρακτηριστικά χρονισμού της οντότητας χρησιμοποιώντας αναθέσεις σημάτων με καθυστέρηση αδράνειας. Μια καθυστέρηση 1,5 ns χρησιμοποιείται για τις ανοδικές μεταβάσεις, και 1,2 ns για τις καθοδικές μεταβάσεις.*

VHDL-87

Η VHDL-87 δεν επιτρέπει τον καθορισμό ορίου απόρριψης παλμού σε ένα μηχανισμό καθυστέρησης. Ο συντακτικός κανόνας στη VHDL-87 είναι

delay_mechanism ⇐ **transport**

Εάν ο μηχανισμός καθυστέρησης παραλείπεται, χρησιμοποιείται η καθυστέρηση αδράνειας, με ένα όριο απόρριψης παλμού ίσο με την καθυστέρηση που καθορίζεται στο στοιχείο κυματομορφής.

5.3.6 Προτάσεις Διεργασίας

Έχουμε χρησιμοποιήσει τις διεργασίες εκτενώς στα παραδείγματα και σε αυτό και στα προηγούμενα κεφάλαια, έτσι έχουμε δει τις περισσότερες από τις λεπτομέρειες για το πώς γράφονται και χρησιμοποιούνται. Για να συνοψίσουμε, ας δούμε την τυπική σύνταξη για μια πρόταση διεργασίας και ας επανεξετάσουμε τη λειτουργία της διεργασίας. Ο συντακτικός κανόνας είναι

```
process_statement ⇐
    [ process_label : ]
    process [ ( signal_name { , … } ) ] [ is ]
        { process_declarative_item }
    begin
        { sequential_statement }
    end process [ process_label ] ;
```

Να υπενθυμίσουμε ότι μια πρόταση διεργασίας είναι μια ταυτόχρονη πρόταση που μπορεί να συμπεριληφθεί σε ένα σώμα αρχιτεκτονικής για να εφαρμόσει το σύνολο ή μέρος της συμπεριφοράς μιας λειτουργικής μονάδας. Η ετικέτα της διεργασίας προσδιορίζει τη διεργασία. Ενώ είναι προαιρετικό, είναι καλή ιδέα να επισυνάπτουμε μια ετικέτα σε κάθε διεργασία. Μια ετικέτα καθιστά ευκολότερη την αποσφαλμάτωση (debugging) κατά την προσομοίωση ενός συστήματος, δεδομένου ότι οι περισσότεροι προσομοιωτές παρέχουν έναν τρόπο προσδιορισμού μιας διεργασίας από την ετικέτα της. Οι περισσότεροι προσομοιωτές επίσης παράγουν ένα προκαθορισμένο όνομα για μια διεργασία εάν παραλείψουμε την ετικέτα στην πρόταση διεργασίας. Προσδιορίζοντας μια διεργασία, μπορούμε να εξετάσουμε το περιεχόμενο των μεταβλητών της ή να θέσουμε σημεία διακοπής (breakpoints) στις προτάσεις μέσα στη διεργασία.

Τα δηλωτικά στοιχεία σε μια πρόταση διεργασίας μπορούν να περιλαμβάνουν δηλώσεις σταθερών, τύπων και μεταβλητών. Σημειώστε ότι οι συνηθισμένες μεταβλητές μπορούν να δηλωθούν μόνο μέσα στις προτάσεις διεργασίας, και όχι έξω από αυτές. Οι μεταβλητές χρησιμοποιούνται για να αναπαραστήσουν την κατάσταση της διεργασίας, όπως έχουμε δει στα παραδείγματα. Οι ακολουθιακές προτάσεις που συγκροτούν το σώμα της διεργασίας μπορούν να

περιλαμβάνουν οποιεσδήποτε από εκείνες τις προτάσεις που εισάγαμε στο Κεφάλαιο 3, και επιπλέον τις προτάσεις ανάθεσης σήματος και αναμονής. Όταν μια διεργασία ενεργοποιείται κατά τη διάρκεια της προσομοίωσης, αρχίζει την εκτέλεση από την πρώτη ακολουθιακή πρόταση και συνεχίζει έως ότου φθάσει στην τελευταία. Έπειτα αρχίζει πάλι από την πρώτη. Αυτό θα ήταν ένας ατέρμονος βρόχος, χωρίς να σημειώνεται πρόοδο στην προσομοίωση, εάν δεν συνυπολογίζαμε τις προτάσεις wait, οι οποίες αναστέλλουν την εκτέλεση της διεργασίας έως ότου συμβεί κάποιο σχετικό γεγονός. Οι προτάσεις wait είναι οι μόνες προτάσεις που παίρνουν περισσότερο από μηδενικό χρόνο προσομοίωσης για να εκτελεστούν. Μόνο μέσω της εκτέλεσης των προτάσεων wait είναι δυνατόν να προχωρήσει ο χρόνος προσομοίωσης.

Μια διεργασία μπορεί να περιλαμβάνει μια λίστα ευαισθησίας σε παρενθέσεις μετά από τη λέξη-κλειδί **process**. Η λίστα ευαισθησίας προσδιορίζει ένα σύνολο σημάτων τα οποία η διεργασία παρακολουθεί για γεγονότα. Εάν η λίστα ευαισθησίας παραλειφθεί, η διεργασία πρέπει να περιλαμβάνει μία ή περισσότερες προτάσεις wait. Από την άλλη μεριά, εάν περιλαμβάνεται λίστα ευαισθησίας, τότε το σώμα της διεργασίας δεν μπορεί να περιλαμβάνει οποιαδήποτε πρόταση wait. Αντί γι' αυτό, υπονοείται μια πρόταση wait, αμέσως πριν από τις λέξεις-κλειδιά **end process**, η οποία περιλαμβάνει τα σήματα που απαριθμούνται στην λίστα ευαισθησίας ως σήματα σε μια φράση **on**.

VHDL-87

Η λέξη-κλειδί **is** δεν επιτρέπεται να συμπεριληφθεί στην επικεφαλίδα μιας πρότασης διεργασίας στη VHDL-87.

5.3.7 Ταυτόχρονες Προτάσεις Ανάθεσης Σήματος

Η μορφή της πρότασης διεργασίας που έχουμε χρησιμοποιήσει ως τώρα είναι η βάση για τη μοντελοποίηση συμπεριφοράς (behavioral modeling) στη VHDL, αλλά για απλές περιπτώσεις, μπορεί να θεωρηθεί κάπως άβολη και φλύαρη. Για αυτόν το λόγο, η VHDL μας παρέχει μερικές χρήσιμες σημειογραφίες συντόμευσης για *μοντελοποίηση συνάρτησης (functional modeling)*, που είναι μοντελοποίηση συμπεριφοράς στην οποία η λειτουργία που περιγράφεται είναι ένας απλός συνδυαστικός μετασχηματισμός των εισόδων σε μια έξοδο. Εξετάζουμε τη βασική μορφή δύο από αυτές τις προτάσεις, τις *ταυτόχρονες προτάσεις ανάθεσης σήματος (concurrent signal assignment statements)*, οι οποίες είναι ταυτόχρονες προτάσεις που είναι ουσιαστικά αναθέσεις σήματος. Αντίθετα από τις συνηθισμένες αναθέσεις σήματος, οι ταυτόχρονες προτάσεις ανάθεσης σήματος μπορούν να συμπεριληφθούν στο μέρος των προτάσεων ενός σώματος αρχιτεκτονικής. Ο συντακτικός κανόνας είναι

concurrent_signal_assignment_statement ⇐

[label :] conditional_signal_assignment

I [label :] selected_signal_assignment

ο οποίος μας λέει ότι οι δύο μορφές καλούνται *ανάθεση σήματος υπό συνθήκη (conditional signal assignment)* και *ανάθεση σήματος με επιλογή (selected signal assignment)*. Κάθε μία από αυτές μπορεί να περιλαμβάνει μια ετικέτα, η οποία εξυπηρετεί ακριβώς τον ίδιο σκοπό με μια ετικέτα σε μια πρόταση διεργασίας: επιτρέπει στην πρόταση να προσδιοριστεί από το όνομά της κατά τη διάρκεια της προσομοίωσης ή της σύνθεσης.

Προτάσεις Ανάθεσης Σήματος υπό Συνθήκη

Η πρόταση ανάθεσης σήματος υπό συνθήκη είναι μια συντόμευση για μια συλλογή συνηθισμένων αναθέσεων σήματος που εμπεριέχονται σε μια πρόταση if, η οποία με τη σειρά της εμπεριέχεται σε μια πρόταση διεργασίας. Ο απλουστευμένος συντακτικός κανόνας για μια ανάθεση σήματος υπό συνθήκη είναι

conditional_signal_assignment ⇐

name <= [delay_mechanism]

{ waveform **when** boolean_expression **else** }

waveform [**when** boolean_expression] ;

Η ανάθεση σήματος υπό συνθήκη μας επιτρέπει να προσδιορίσουμε μια κυματομορφή ανάμεσα σε έναν αριθμό κυματομορφών ανάλογα με τις τιμές μερικών συνθηκών η οποία πρέπει να ανατεθεί σε ένα σήμα. Ας εξετάσουμε μερικά παραδείγματα για να δείξουμε πώς κάθε ανάθεση σήματος υπό συνθήκη μπορεί να μετασχηματιστεί σε μια ισοδύναμη πρόταση διεργασίας. Καταρχήν, η πρόταση στην κορυφή της Εικόνας 5-17 είναι μια περιγραφή της συνάρτησης ενός πολυπλέκτη, με τέσσερις εισόδους δεδομένων (d0, d1, d2 και d3), δύο εισόδους επιλογής (sel0 και sel1) και μια έξοδο δεδομένων (z). Όλα αυτά τα σήματα είναι τύπου bit. Αυτή η πρόταση έχει ακριβώς την ίδια έννοια με την πρόταση διεργασίας που φαίνεται στο κάτω μέρος της Εικόνας 5-17.

ΕΙΚΟΝΑ 5-17

```
zmux : z <= d0 when sel1 = '0' and sel0 = '0' else
           d1 when sel1 = '0' and sel0 = '1' else
           d2 when sel1 = '1' and sel0 = '0' else
           d3 when sel1 = '1' and sel0 = '1';
```

```
zmux : process is
begin
    if sel1 = '0' and sel0 = '0' then
        z <= d0;
    elsif sel1 = '0' and sel0 = '1' then
        z <= d1;
    elsif sel1 = '1' and sel0 = '0' then
        z <= d2;
    elsif sel1 = '1' and sel0 = '1' then
        z <= d3;
    end if;
    wait on d0, d1, d2, d3, sel0, sel1;
end process zmux;
```

Πάνω: ένα μοντέλο συνάρτησης ενός πολυπλέκτη, που χρησιμοποιεί μια πρόταση ανάθεσης σήματος υπό συνθήκη. Κάτω: η ισοδύναμη πρόταση διεργασίας.

Το πλεονέκτημα της μορφής ανάθεσης σήματος υπό συνθήκη έναντι της ισοδύναμης διεργασίας είναι σαφώς εμφανές από αυτό το παράδειγμα. Ο απλός συνδυαστικός μετασχηματισμός είναι προφανής στον αναγνώστη, απαλλαγμένος από τις περιττές λεπτομέρειες του μηχανισμού της διεργασίας. Αυτό δεν πάει να πει ότι οι διεργασίες είναι ένα κακό πράγμα, αλλά ότι σε απλές περιπτώσεις, θα προτιμούσαμε μάλλον να κρύψουμε αυτές τις λεπτομέρειες για να καταστήσουμε το πρότυπο πιο σαφές. Η εξέταση της ισοδύναμης διεργασίας μας παρουσιάζει κάτι σημαντικό για την πρόταση ανάθεσης σήματος υπό συνθήκη, ότι δηλαδή είναι ευαίσθητη σε όλα τα σήματα που αναφέρονται στις κυματομορφές και τις συνθήκες. Έτσι όταν οποιοδήποτε από αυτά αλλάξει τιμή, η ανάθεση υπό συνθήκη επαναξιολογείται και μια νέα συναλλαγή χρονοπρογραμματίζεται για τον οδηγό του σήματος προορισμού.

Εάν εξετάσουμε πιο προσεκτικά το μοντέλο του πολυπλέκτη, επισημαίνουμε ότι η τελευταία συνθήκη είναι περιττή, δεδομένου ότι τα σήματα sel0 και sel1 είναι τύπου bit. Εάν καμία από τις προηγούμενες συνθήκες δεν είναι αληθής, στο σήμα πρέπει να ανατεθεί η τελευταία κυματομορφή. Έτσι μπορούμε να ξαναγράψουμε το παράδειγμα όπως φαίνεται στην Εικόνα 5-18.

Μια πολύ κοινή περίπτωση στη μοντελοποίηση συνάρτησης είναι να γράψουμε μια ανάθεση σήματος υπό συνθήκη χωρίς συνθήκες, όπως στο ακόλουθο παράδειγμα:

```
PC_incr : next_PC <= PC + 4 after 5 ns;
```

Με μια πρώτη ματιά αυτό εμφανίζεται να είναι μια συνηθισμένη ακολουθιακή πρόταση ανάθεσης σήματος, η οποία σύμφωνα με τους κανόνες οφείλει να είναι μέσα σε ένα σώμα διεργασίας. Εντούτοις, εάν εξετάσουμε τον συντακτικό κανόνα για μια ταυτόχρονη ανάθεση σήματος, παρατηρούμε ότι μια τέτοια ανάθεση σήματος μπορεί στην πραγματικότητα να αναγνωριστεί υπό αυτήν τη μορφή εάν όλα τα προαιρετικά μέρη εκτός από την ετικέτα παραληφθούν. Σε αυτήν την περίπτωση, η ισοδύναμη πρόταση διεργασίας είναι

```
PC_incr : process is
begin
    next_PC <= PC + 4 after 5 ns;
    wait on PC;
end process PC_incr;
```

ΕΙΚΟΝΑ 5-18

```
zmux : z <= d0 when sel1 = '0' and sel0 = '0' else
           d1 when sel1 = '0' and sel0 = '1' else
           d2 when sel1 = '1' and sel0 = '0' else
           d3;
```

```
zmux : process is
begin
    if sel1 = '0' and sel0 = '0' then
        z <= d0;
    elsif sel1 = '0' and sel0 = '1' then
        z <= d1;
    elsif sel1 = '1' and sel0 = '0' then
        z <= d2;
    else
        z <= d3;
    end if;
    wait on d0, d1, d2, d3, sel0, sel1;
end process zmux;
```

Ένα αναθεωρημένο μοντέλο συνάρτησης για τον πολυπλέκτη, με την ισοδύναμή του πρόταση διεργασίας.

Μια άλλη περίπτωση που προκύπτει μερικές φορές όταν γράφουμε μοντέλα συνάρτησης είναι η ανάγκη για μια διεργασία που χρονοπρογραμματίζει ένα αρχικό σύνολο συναλλαγών και στη συνέχεια δεν κάνει τίποτα άλλο για το υπόλοιπο της προσομοίωσης. Ένα παράδειγμα είναι η παραγωγή ενός σήματος μηδένισης (reset). Ένας τρόπος για να γίνει αυτό είναι ο ακόλουθος:

```
reset_gen : reset <= '1', '0' after 200 ns when extended_reset else
                     '1', '0' after 50 ns;
```

Αυτό που πρέπει να σημειώσουμε εδώ είναι ότι δεν υπάρχει κανένα σήμα σε κάποια από τις κυματομορφές ή τις συνθήκες (υποθέτοντας ότι το extended_reset είναι σταθερά). Αυτό σημαίνει ότι η πρόταση εκτελείται μια φορά όταν αρχίζει η προσομοίωση, χρονοπρογραμματίζει δύο συναλλαγές στο reset και στη συνέχεια παραμένει αδρανής. Η ισοδύναμη διεργασία είναι

```
reset_gen : process is
begin
    if extended_reset then
        reset <= '1', '0' after 200 ns;
    else
        reset <= '1', '0' after 50 ns;
    end if;
    wait;
end process reset_gen;
```

Δεδομένου ότι δεν υπάρχει κανένα σήμα που να εμπλέκεται, η πρόταση wait δεν έχει καμία λίστα ευαισθησίας. Κατά συνέπεια μετά από την εκτέλεση της πρότασης if, η διεργασία αναστέλλεται για πάντα.

Εάν συμπεριλάβουμε ένα μηχανισμό καθυστέρησης σε μια πρόταση ανάθεσης σήματος υπό συνθήκη, χρησιμοποιείται όποια από τις κυματομορφές κι αν επιλεγεί. Έτσι θα μπορούσαμε να ξαναγράψουμε το μοντέλο για το ασύμμετρο στοιχείο καθυστέρησης που παρουσιάστηκε στο σχήμα 5-12 ως εξής

```
asym_delay : z <= transport a after Tpd_01 when a = '1' else
                              a after Tpd_10;
```

Ένα πρόβλημα με τις αναθέσεις σήματος υπό συνθήκη, όπως τις έχουμε περιγράψει έως τώρα, είναι ότι θέτουν πάντα μια νέα τιμή σε ένα σήμα. Μερικές φορές μπορεί να μην θέλουμε να αλλάξουμε την τιμή ενός σήματος, ή πιο συγκεκριμένα, μπορεί να μην θέλουμε να χρονοπρογραμματίσουμε νέες συναλλαγές στο σήμα. Μπορούμε να χρησιμοποιήσουμε τη λέξη-κλειδί **unaffected** αντί μίας κανονικής κυματομορφής για αυτές τις περιπτώσεις, όπως φαίνεται στο πάνω μέρος της Εικόνας 5-19.

ΕΙΚΟΝΑ 5-19

```
scheduler :
    request <= first_priority_request after scheduling_delay
                    when priority_waiting and server_status = ready else
               first_normal_request after scheduling_delay
                    when not priority_waiting and server_status = ready else
               unaffected
                    when server_status = busy else
               reset_request after scheduling_delay;
```

```
scheduler : process is
begin
    if priority_waiting and server_status = ready then
        request <= first_priority_request after scheduling_delay;
    elsif not priority_waiting and server_status = ready then
        request <= first_normal_request after scheduling_delay;
    elsif server_status = busy then
        null;
    else
        request <= reset_request after scheduling_delay;
    end if;
    wait on first_priority_request, priority_waiting, server_status,
            first_normal_request, reset_request;
end process scheduler;
```

Πάνω: μια πρόταση ανάθεση σήματος υπό συνθήκη που δείχνει τη χρήση της κυματομορφής unaffected. *Κάτω: η ισοδύναμη πρόταση διεργασίας.*

Η επίδραση της κυματομορφής **unaffected** είναι να συμπεριληφθεί μια πρόταση null στην ισοδύναμη διεργασία, που την αναγκάζει να παρακάμψει το χρονοπρογραμματισμό μιας συναλλαγής όταν η αντίστοιχη συνθήκη είναι αληθής. (Θυμηθείτε ότι το αποτέλεσμα της ακολουθιακής πρότασης null είναι να μην γίνει τίποτα.) Έτσι το παράδειγμα στο πάνω μέρος της Εικόνας 5-19 είναι ισοδύναμο με τη διεργασία που φαίνεται στο κάτω μέρος. Σημειώστε ότι μπορούμε να χρησιμοποιήσουμε τη λέξη-κλειδί **unaffected** μόνο σε μια ταυτόχρονη ανάθεση σήματος, και όχι σε μια ακολουθιακή ανάθεση σήματος.

VHDL-87

Στη VHDL-87 ο συντακτικός κανόνας για μια πρόταση ανάθεσης σήματος υπό συνθήκη είναι

conditional_signal_assignment ⇐

name <= [**transport**]

{ waveform **when** boolean_expression **else** }

waveform ;

Ο μηχανισμός καθυστέρησης περιορίζεται στη λέξη-κλειδί **transport**. Η τελική κυματομορφή δεν επιτρέπεται να είναι υπό συνθήκη. Επιπλέον, δεν επιτρέπεται να χρησιμοποιήσουμε τη λέξη-κλειδί **unaffected**. Εάν η απαιτούμενη συμπεριφορά δεν μπορεί να εκφραστεί με αυτούς τους περιορισμούς, πρέπει να γράψουμε μια πλήρη πρόταση διεργασίας αντί για μια πρόταση ανάθεσης σήματος υπό συνθήκη.

Προτάσεις Ανάθεσης Σήματος με Επιλογή

Η πρόταση ανάθεσης σήματος με επιλογή είναι παρόμοια από πολλές απόψεις με την πρόταση ανάθεσης σήματος υπό συνθήκη. Είναι και αυτή μια συντόμευση για έναν αριθμό συνηθισμένων αναθέσεων σήματος ενσωματωμένων σε μια διεργασία. Αλλά για μια ανάθεση σήματος με επιλογή, η ισοδύναμη διεργασία περιέχει μια πρόταση case αντί για μια πρόταση if. Ο απλουστευμένος συντακτικός κανόνας είναι

```
selected_signal_assignment ⇐
    with expression select
        name <= [ delay_mechanism ]
            { waveform when choices , }
            waveform when choices ;
```

Αυτή η πρόταση μας επιτρέπει να επιλέξουμε μεταξύ διαφορετικών κυματομορφών που ανατίθενται σε ένα σήμα ανάλογα με την τιμή μιας παράστασης. Για παράδειγμα, ας εξετάσουμε την ανάθεση σήματος με επιλογή που φαίνεται στο πάνω μέρος της Εικόνας 5-20. Αυτό έχει την ίδια έννοια με την πρόταση διεργασίας που περιέχει μια πρόταση case και φαίνεται στο κάτω μέρος της Εικόνας 5-20.

Μια πρόταση ανάθεσης σήματος με επιλογή είναι ευαίσθητη σε όλα τα σήματα στην παράσταση επιλογής και στις κυματομορφές. Αυτό σημαίνει ότι η ανάθεση σήματος με επιλογή στην Εικόνα 5-20 είναι ευαίσθητη στο σήμα b και θα ξαναρχίσει την εκτέλεσή της εάν το b αλλάξει τιμή, ακόμα κι αν η τιμή του σήματος alu_function είναι alu_pass_a.

Ένα σημαντικό σημείο που πρέπει να επισημάνουμε για μια πρόταση ανάθεσης σήματος με επιλογή είναι ότι η πρόταση case στην ισοδύναμη διεργασία πρέπει να ακολουθεί όλους τους κανόνες που περιγράψαμε στο Κεφάλαιο 3. Αυτό σημαίνει ότι κάθε πιθανή τιμή για την παράσταση επιλογής πρέπει να συνυπολογισθεί σε κάθε μια από τις επιλογές, ότι καμία τιμή δεν εμπεριέχεται σε περισσότερες από μια επιλογές, και τα λοιπά.

Εκτός από τη διαφορά στην ισοδύναμη διεργασία, η ανάθεση σήματος με επιλογή είναι παρόμοια με την ανάθεση υπό συνθήκη. Κατά συνέπεια η ειδική κυματομορφή **unaffected** μπορεί να χρησιμοποιηθεί για να καθορίσει ότι καμία

ανάθεση δεν πραγματοποιείται για μερικές τιμές της παράστασης επιλογής. Επίσης, εάν ένας μηχανισμός καθυστέρησης καθορίζεται στην πρόταση, αυτός ο μηχανισμός χρησιμοποιείται σε κάθε ακολουθιακή ανάθεση σήματος μέσα στην ισοδύναμη διεργασία.

ΕΙΚΟΝΑ 5-20

```
alu : with alu_function select
        result <= a + b after Tpd    when alu_add | alu_add_unsigned,
                  a – b after Tpd    when alu_sub | alu_sub_unsigned,
                  a and b after Tpd  when alu_and,
                  a or b after Tpd   when alu_or,
                  a after Tpd        when alu_pass_a;
```

```
alu : process is
begin
    case alu_function is
        when alu_add | alu_add_unsigned      => result <= a + b after Tpd;
        when alu_sub | alu_sub_unsigned      => result <= a – b after Tpd;
        when alu_and                         => result <= a and b after Tpd;
        when alu_or                          => result <= a or b after Tpd;
        when alu_pass_a                      => result <= a after Tpd;
    end case;
    wait on alu_function, a, b;
end process alu;
```

Πάνω: μια πρόταση ανάθεσης σήματος με επιλογή. Κάτω: η ισοδύναμή της διεργασία.

ΠΑΡΑΔΕΙΓΜΑ

Μπορούμε να χρησιμοποιήσουμε μια ανάθεση σήματος με επιλογή για να εκφράσουμε μια συνδυαστική λογική συνάρτηση σε μορφή πίνακα αληθείας (truth table). Η Εικόνα 5-21 παρουσιάζει μια δήλωση οντότητας και ένα σώμα αρχιτεκτονικής για έναν πλήρη αθροιστή. Η πρόταση ανάθεσης σήματος με επιλογή έχει, ως παράσταση επιλογής, ένα διάνυσμα bit που σχηματίζεται με τη συνάθροιση των σημάτων εισόδου. Οι επιλογές απαριθμούν όλες τις πιθανές τιμές των εισόδων, και για κάθε μια, δίνονται οι τιμές για τις εξόδους c_out και s.

ΕΙΚΟΝΑ 5-21

```
entity full_adder is
    port ( a, b, c_in : in bit; s, c_out : out bit );
end entity full_adder;
```

```
architecture truth_table of full_adder is
begin
    with bit_vector'(a, b, c_in) select
        (c_out, s) <= bit_vector'("00") when "000",
                      bit_vector'("01") when "001",
                      bit_vector'("01") when "010",
                      bit_vector'("10") when "011",
                      bit_vector'("01") when "100",
                      bit_vector'("10") when "101",
                      bit_vector'("10") when "110",
                      bit_vector'("11") when "111";
end architecture truth_table;
```

Μια δήλωση οντότητας και το σώμα αρχιτεκτονικής συνάρτησης για έναν πλήρη αθροιστή.

Αυτό το παράδειγμα επεξηγεί την πιο κοινή χρήση των συναθροίσεων στους στόχους των αναθέσεων σήματος. Να σημειώσουμε ότι η επεξήγηση τύπου απαιτείται στην παράσταση επιλογής για να διευκρινίσει τον τύπο της συνάθροισης. Η επεξήγηση τύπου είναι αναγκαία στις τιμές εξόδου για να διακρίνει τα αλφαριθμητικά κυριολεκτικά τύπου bit vector από τα αλφαριθμητικά κυριολεκτικά τύπου character.

VHDL-87

Στη VHDL-87, ο μηχανισμός καθυστέρησης περιορίζεται στη λέξη-κλειδί **transport**. Επιπλέον, η λέξη-κλειδί **unaffected** δεν επιτρέπεται να χρησιμοποιηθεί. Εάν η απαιτούμενη συμπεριφορά δεν μπορεί να εκφραστεί χωρίς τη χρησιμοποίηση της λέξης-κλειδί **unaffected**, πρέπει να γράψουμε μια πλήρη πρόταση διεργασίας αντί για μια πρόταση ανάθεσης σήματος με επιλογή.

5.3.8 Ταυτόχρονες Προτάσεις Ισχυρισμού

Η VHDL παρέχει άλλη μία σημειογραφία συντόμευσης της διεργασίας, την *ταυτόχρονη πρόταση ισχυρισμού (concurrent assertion statement)*, η οποία μπορεί να χρησιμοποιηθεί στη μοντελοποίηση συμπεριφοράς. Όπως υπονοεί το όνομά της, μια ταυτόχρονη πρόταση ισχυρισμού αναπαριστά μια διεργασία της οποίας το σώμα περιέχει μια συνηθισμένη ακολουθιακή πρόταση ισχυρισμού. Ο συντακτικός κανόνας είναι

concurrent_assertion_statement ⇐

[label :]

assert boolean_expression

[**report** expression] [**severity** expression] ;

Αυτή η σύνταξη εμφανίζεται να είναι ακριβώς ίδια με αυτήν της ακολουθιακής πρότασης ισχυρισμού, αλλά η διαφορά είναι ότι μπορεί να εμφανιστεί ως ταυτόχρονη πρόταση. Η προαιρετική ετικέτα στην πρόταση εξυπηρετεί τον ίδιο σκοπό με αυτόν σε μια πρόταση διεργασίας: να μας παρέχει έναν τρόπο για να αναφερόμαστε στην πρόταση κατά τη διάρκεια της προσομοίωσης ή της σύνθεσης. Η διεργασία που είναι ισοδύναμη με έναν ταυτόχρονο ισχυρισμό περιέχει έναν ακολουθιακό ισχυρισμό με την ίδια συνθήκη (condition), φράση αναφοράς (report clause) και φράση αυστηρότητας (severity clause). Ο ακολουθιακός ισχυρισμός ακολουθείται από μια πρόταση wait της οποίας η λίστα ευαισθησίας περιλαμβάνει τα σήματα που αναφέρονται στην παράσταση συνθήκης. Κατά συνέπεια η επίδραση της ταυτόχρονης πρότασης ισχυρισμού είναι να ελεγχθεί ότι η συνθήκη είναι αληθής κάθε φορά που οποιαδήποτε από τα σήματα που αναφέρονται στη συνθήκη αλλάξουν τιμή. Οι ταυτόχρονοι ισχυρισμοί παρέχουν έναν πολύ συμπαγή και χρήσιμο τρόπο για να εισάγουμε ελέγχους χρονισμού και ορθότητας σε ένα μοντέλο.

ΠΑΡΑΔΕΙΓΜΑ

Μπορούμε να χρησιμοποιήσουμε ταυτόχρονες προτάσεις ισχυρισμού για να ελέγξουμε τη σωστή χρήση ενός φλιπ-φλοπ ενεργοποίησης/μηδένισης (set/reset flip-flop), με δύο εισόδους s και r και δύο εξόδους q και q_n, όλες τύπου bit. Η απαίτηση για τη χρήση του φλιπ-φλοπ είναι ότι το s και το r δεν μπορεί να είναι συγχρόνως και τα δύο '1'. Η οντότητα και το σώμα αρχιτεκτονικής παρουσιάζονται στην Εικόνα 5-22.

Η πρώτη και η δεύτερη ταυτόχρονη πρόταση υλοποιούν τη συνάρτηση του μοντέλου. Η τρίτη ελέγχει τη σωστή χρήση και ξαναρχίζει την εκτέλεσή της όταν είτε το s είτε το r αλλάξει τιμή, αφού αυτά είναι τα σήματα που αναφέρονται στη λογική συνθήκη (Boolean condition) . Εάν και τα δύο σήματα είναι '1', τότε αναφέρεται παραβίαση του ισχυρισμού. Η ισοδύναμη διεργασία για τον ταυτόχρονο ισχυρισμό είναι

```
check : process is
begin
    assert not (s = '1' and r = '1')
        report "Incorrect use of S_R_flip_flop: s and r both '1'";
    wait on s, r;
end process check;
```

ΕΙΚΟΝΑ 5-22

```
entity S_R_flipflop is
    port ( s, r : in bit; q, q_n : out bit );
end entity S_R_flipflop;
architecture functional of S_R_flipflop is
begin
    q <=    '1' when s = '1' else
            '0' when r = '1';
    q_n <= '0' when s = '1' else
            '1' when r = '1';
    check : assert not (s = '1' and r = '1')
        report "Incorrect use of S_R_flip_flop: s and r both '1'";
end architecture functional;
```

Μια οντότητα και το σώμα αρχιτεκτονικής για ένα φλιπ-φλοπ ενεργοποίησης/μηδένισης, που περιλαμβάνει μια ταυτόχρονη πρόταση ισχυρισμού για να ελέγξει τη σωστή χρήση.

5.3.9 Οντότητες και Παθητικές Διεργασίες

Ολοκληρώνουμε αυτήν την ενότητα για τη μοντελοποίηση συμπεριφοράς επιστρέφοντας στις δηλώσεις οντοτήτων. Μπορούμε να συμπεριλάβουμε ορισμένα είδη ταυτόχρονων προτάσεων σε μια δήλωση οντότητας, για να παρακολουθούμε τη χρήση και τη λειτουργία της οντότητας. Ο εκτεταμένος συντακτικός κανόνας για μια δήλωση οντότητας που δείχνει αυτό είναι

entity_declaration ⇐

entity identifier **is**

[**port** (port_interface_list) ;]

{ entity_declarative_item }

[**begin**

{ concurrent_assertion_statement

I passive_concurrent_procedure_call_statement

I passive_process_statement }]

end [**entity**] [identifier] ;

Οι ταυτόχρονες προτάσεις που περιλαμβάνονται σε μια δήλωση οντότητας πρέπει να είναι *παθητικές (passive)*, που σημαίνει ότι δεν μπορούν να επηρεάσουν τη λειτουργία της οντότητας με κανένα τρόπο. Μια ταυτόχρονη πρόταση ισχυρισμού καλύπτει αυτήν την απαίτηση, δεδομένου ότι εξετάζει απλά μια συνθήκη όποτε συμβαίνουν γεγονότα στα σήματα στα οποία είναι ευαίσθητη. Μια πρόταση διεργασίας είναι παθητική εάν δεν περιέχει καθόλου προτάσεις ανάθεσης σήματος ή κλήσεις σε διαδικασίες που περιέχουν προτάσεις ανάθεσης σήματος. Μια τέτοια διεργασία μπορεί να χρησιμοποιηθεί για να παρακολουθεί γεγονότα που συμβαίνουν στις εισόδους της οντότητας.

ΠΑΡΑΔΕΙΓΜΑ

Μπορούμε να ξαναγράψουμε τη δήλωση οντότητας για το φλιπ-φλοπ ενεργοποίησης/μηδένισης της Εικόνας 5-22 όπως φαίνεται στην Εικόνα 5-23. Εάν κάνουμε αυτό, ο έλεγχος συμπεριλαμβάνεται σε κάθε πιθανή υλοποίηση του φλιπ-φλοπ και δεν είναι ανάγκη να συμπεριληφθεί στα αντίστοιχα σώματα αρχιτεκτονικής.

ΕΙΚΟΝΑ 5-23

```
entity S_R_flipflop is
    port ( s, r : in bit; q, q_n : out bit );
begin
    check : assert not (s = '1' and r = '1')
                report "Incorrect use of S_R_flip_flop: s and r both '1'";
end entity S_R_flipflop;
```

Η αναθεωρημένη δήλωση οντότητας για το φλιπ-φλοπ ενεργοποίησης/μηδένισης, που περιλαμβάνει την ταυτόχρονη πρόταση ισχυρισμού για να ελέγχει τη σωστή χρήση.

ΠΑΡΑΔΕΙΓΜΑ

Η Εικόνα 5-24 παρουσιάζει μια δήλωση οντότητας για μια μνήμη μόνο για ανάγνωση (Read-Only Memory, ROM). Περιλαμβάνει μια παθητική διεργασία, trace_reads, η οποία είναι ευαίσθητη στις αλλαγές στη θύρα enable. Όταν η τιμή της θύρας αλλάζει σε '1', η διεργασία αναφέρει ένα μήνυμα που επισημαίνει το χρόνο και τη διεύθυνση της λειτουργίας ανάγνωσης. Η διεργασία δεν επηρεάζει την πορεία της προσομοίωσης με κανένα τρόπο, αφού δεν περιλαμβάνει καμία ανάθεση σήματος.

ΕΙΚΟΝΑ 5-24

```
entity ROM is
    port (  address : in natural;
            data : out bit_vector(0 to 7);
            enable : in bit );
begin
    trace_reads : process (enable) is
    begin
        if enable = '1' then
            report "ROM read at time " & time'image(now)
                        & " from address " & natural'image(address);
        end if;
    end process trace_reads;
end entity ROM;
```

Μια δήλωση οντότητας για μια ROM, που περιλαμβάνει μια παθητική διεργασία για να παρακολουθεί τις λειτουργίες ανάγνωσης.

5.4 Περιγραφές Δομής

Μια περιγραφή δομής ενός συστήματος εκφράζεται υπό την μορφή υποσυστημάτων που διασυνδέονται με σήματα. Κάθε υποσύστημα μπορεί με τη σειρά του να αποτελείται από μια διασύνδεση των υπο-υποσυστημάτων, και ούτω καθεξής, έως ότου φθάσουμε τελικά σε ένα επίπεδο που αποτελείται από στοιχειώδη συστατικά (primitive components), που περιγράφονται αποκλειστικά από τη σκοπιά της συμπεριφοράς τους. Κατά συνέπεια το σύστημα στην κορυφή μπορεί να θεωρηθεί σαν να έχει μια ιεραρχική δομή. Σε αυτήν την ενότητα, εξετάζουμε το πώς γράφουμε σώματα αρχιτεκτονικής δομής για να εκφράσουμε αυτήν την ιεραρχική οργάνωση.

5.4.1 Εμφάνιση Στιγμιότυπου Συστατικού και Αντιστοιχίσεις Θυρών

Έχουμε δει νωρίτερα σε αυτό το κεφάλαιο ότι οι ταυτόχρονες προτάσεις σε ένα σώμα αρχιτεκτονικής περιγράφουν την υλοποίηση μιας διασύνδεσης της οντότητας. Προκειμένου να γράψουμε μια υλοποίηση δομής, πρέπει να χρησιμοποιήσουμε μια ταυτόχρονη πρόταση που ονομάζεται πρόταση *εμφάνισης στιγμιότυπου συστατικού (component instantiation)*, η απλούστερη μορφή της οποίας διέπεται από το συντακτικό κανόνα

component_instantiation_statement ⇐
    instantiation_label :
        entity entity_name [(architecture_identifier)]
            [**port map** (port_association_list)] ;

Αυτή η μορφή πρότασης εμφάνισης στιγμιότυπου συστατικού εκτελεί *άμεση εμφάνιση στιγμιότυπου (direct instantiation)* μιας οντότητας. Μπορούμε να σκεφτούμε την εμφάνιση στιγμιότυπου συστατικού ως δημιουργία ενός αντιγράφου της ονομαζόμενης οντότητας, με το αντίστοιχο σώμα αρχιτεκτονικής να αντικαθιστά το στιγμιότυπο του συστατικού. Η αντιστοίχιση θυρών καθορίζει ποιες θύρες της οντότητας συνδέονται σε ποια σήματα του σώματος αρχιτεκτονικής που την εσωκλείει. Ο απλουστευμένος συντακτικός κανόνας για μια λίστα συσχέτισης θυρών είναι

port_association_list ⇐

    ([port_name =>] (signal_name I expression I **open**)) { , … }

Κάθε στοιχείο στη λίστα συσχέτισης συσχετίζει μία θύρα της οντότητας είτε με ένα σήμα του σώματος αρχιτεκτονικής που την εσωκλείει είτε με την τιμή μιας παράστασης, είτε αφήνει τη θύρα ασυσχέτιστη, όπως υποδεικνύεται από τη λέξη-κλειδί **open**.

Ας εξετάσουμε μερικά παραδείγματα για να επεξηγήσουμε τις προτάσεις εμφάνισης στιγμιότυπου συστατικού και την συσχέτιση θυρών με σήματα. Υποθέστε ότι έχουμε μια οντότητα που έχει δηλωθεί ως εξής

```
entity DRAM_controller is
    port ( rd, wr, mem: in bit;
           ras, cas, we, ready : out bit );
end entity DRAM_controller;
```

και μια αντίστοιχη αρχιτεκτονική που ονομάζεται fpld. Θα μπορούσαμε να δημιουργήσουμε ένα στιγμιότυπο αυτής της οντότητας ως εξής:

```
main_mem_controller : entity work.DRAM_controller(fpld)
    port map ( cpu_rd, cpu_wr, cpu_mem,
               mem_ras, mem_cas, mem_we, cpu_rdy );
```

Σε αυτό το παράδειγμα, το όνομα work αναφέρεται στην τρέχουσα βιβλιοθήκη εργασίας στην οποία αποθηκεύονται οι οντότητες και τα σώματα αρχιτεκτονικής. Θα επιστρέψουμε στο θέμα των βιβλιοθηκών στην επόμενη ενότητα. Η αντιστοίχιση θυρών αυτού του παραδείγματος απαριθμεί τα σήματα του σώματος αρχιτεκτονικής που εσωκλείει την οντότητα στα οποία συνδέονται οι θύρες του αντιγράφου της οντότητας. Χρησιμοποιείται *συσχέτιση θέσης (positional association)*: κάθε σήμα που απαριθμείται στην αντιστοίχιση θυρών συνδέεται με τη θύρα που βρίσκεται στην ίδια θέση στη δήλωση της οντότητας. Έτσι το σήμα cpu_rd συνδέεται με τη θύρα rd, το σήμα cpu_wr συνδέεται με τη θύρα wr και ούτω καθεξής.

Ένα από τα προβλήματα με τη συσχέτιση θέσης είναι ότι δεν είναι αμέσως σαφές ποια σήματα συνδέονται σε ποιες θύρες. Κάποιος που διαβάζει την περιγραφή πρέπει να ανατρέξει στη δήλωση της οντότητας για να ελέγξει τη σειρά των θυρών στη διασύνδεση της οντότητας. Ένας καλύτερος τρόπος για να γράψουμε μια πρόταση εμφάνισης στιγμιότυπου συστατικού είναι να χρησιμοποιήσουμε *συσχέτιση ονόματος (named association)*, όπως φαίνεται στο ακόλουθο παράδειγμα:

```
main_mem_controller : entity work.DRAM_controller(fpld)
    port map ( rd => cpu_rd, wr => cpu_wr,
               mem => cpu_mem, ready => cpu_rdy,
               ras => mem_ras, cas => mem_cas, we => mem_we );
```

Εδώ, κάθε θύρα ονομάζεται ρητά μαζί με το σήμα στο οποίο συνδέεται. Η σειρά με την οποία οι συνδέσεις παρατίθενται δεν έχει σημασία. Το πλεονέκτημα αυτής της προσέγγισης είναι ότι είναι αμέσως προφανές στον αναγνώστη πως συνδέεται η οντότητα με τη δομή του σώματος αρχιτεκτονικής που την εσωκλείει.

Στο προηγούμενο παράδειγμα έχουμε ονομάσει ρητά το σώμα αρχιτεκτονικής που χρησιμοποιείται και αντιστοιχεί στην οντότητα της οποίας έχουμε το στιγμιότυπο. Εντούτοις, ο συντακτικός κανόνας για τις προτάσεις εμφάνισης στιγμιότυπου συστατικού το παρουσιάζει αυτό ως προαιρετικό. Εάν επιθυμούμε, μπορούμε να παραλείψουμε τον καθορισμό του σώματος αρχιτεκτονικής, οπότε σ' αυτήν την περίπτωση αυτό που χρησιμοποιείται μπορεί να επιλεγεί όταν το συνολικό μοντέλο επεξεργάζεται για προσομοίωση, σύνθεση ή κάποιο άλλο σκοπό. Εκείνη τη χρονική στιγμή, εάν καμία άλλη επιλογή δεν διευκρινίζεται, επιλέγεται το σώμα αρχιτεκτονικής που έχει αναλυθεί πιο πρόσφατα. Θα επιστρέψουμε στο θέμα της ανάλυσης των μοντέλων στην επόμενη ενότητα.

ΠΑΡΑΔΕΙΓΜΑ

Στην Εικόνα 5-5 εξετάσαμε ένα μοντέλο συμπεριφοράς ενός φλιπ-φλοπ ενεργοποιούμενο σε ακμή (edge-triggered flip-flop). Μπορούμε να χρησιμοποιήσουμε το φλιπ-φλοπ ως βάση ενός τετράμπιτου καταχωρητή ενεργοποιούμενο σε ακμή (four-bit edge-triggered register). Η Εικόνα 5-25 παρουσιάζει τη δήλωση οντότητας και ένα σώμα αρχιτεκτονικής δομής.

ΕΙΚΟΝΑ 5-25

```
entity reg4 is
    port ( clk, clr, d0, d1, d2, d3 : in bit; q0, q1, q2, q3 : out bit );
end entity reg4;
```

```
architecture struct of reg4 is
begin
    bit0 : entity work.edge_triggered_Dff(behavioral)
        port map (d0, clk, clr, q0);
    bit1 : entity work.edge_triggered_Dff(behavioral)
        port map (d1, clk, clr, q1);
    bit2 : entity work.edge_triggered_Dff(behavioral)
        port map (d2, clk, clr, q2);
    bit3 : entity work.edge_triggered_Dff(behavioral)
        port map (d3, clk, clr, q3);
end architecture struct;
```

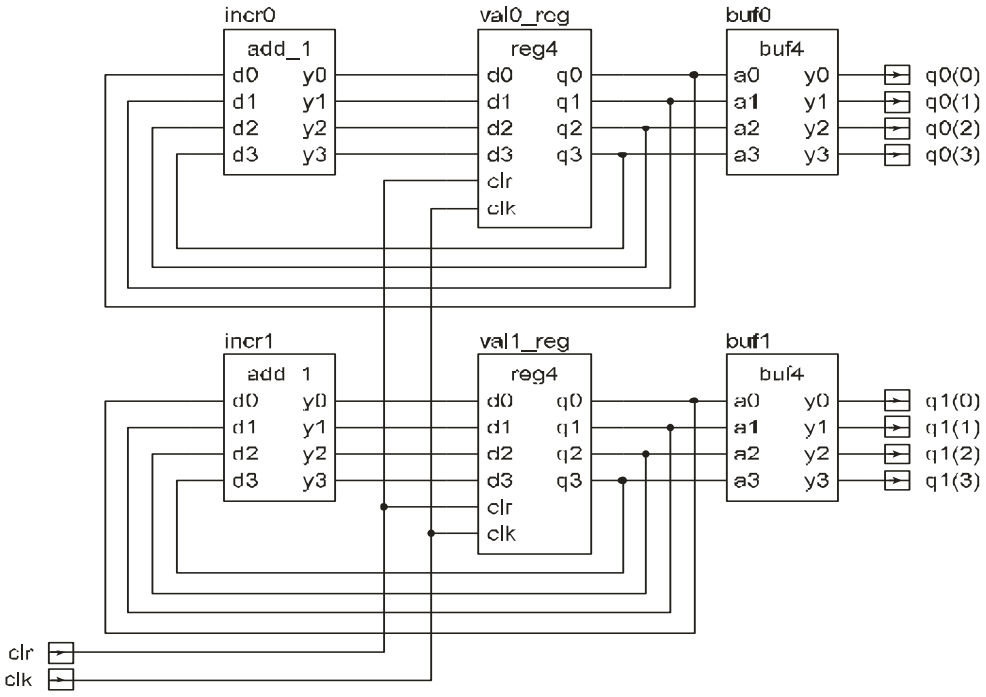
Μια οντότητα και το σώμα αρχιτεκτονικής δομής για έναν τετράμπιτο καταχωρητή ενεργοποιούμενο σε ακμή (four-bit edge-triggered register), με ασύγχρονη είσοδο καθαρισμού (asynchronous clear input).

Μπορούμε να χρησιμοποιήσουμε την οντότητα καταχωρητή, μαζί με άλλες οντότητες, ως τμήμα μιας αρχιτεκτονικής δομής για το διψήφιο δεκαδικό μετρητή (two-digit decimal counter) που αναπαριστάται από το σχηματικό διάγραμμα της Εικόνας 5-26. Υποθέστε ότι ένα ψηφίο αναπαριστάται ως ένα διάνυσμα μήκους τεσσάρων bit, που περιγράφεται από τη δήλωση υποτύπου

```
subtype digit is bit_vector(3 downto 0);
```

Η Εικόνα 5-27 παρουσιάζει τη δήλωση οντότητας για το μετρητή, μαζί με ένα περίγραμμα του σώματος αρχιτεκτονικής δομής. Αυτό το παράδειγμα επεξηγεί διάφορα σημαντικά σημεία για τα στιγμιότυπα συστατικών και τις αντιστοιχίσεις θυρών. Πρώτον, τα δύο στιγμιότυπα συστατικών val0_reg και val1_reg είναι και τα δύο στιγμιότυπα του ίδιου ζευγαριού οντότητας/αρχιτεκτονικής. Αυτό σημαίνει ότι δημιουργούνται δύο ξεχωριστά αντίγραφα της αρχιτεκτονικής struct του reg4, μια για κάθε στιγμιότυπο συστατικού. Θα επιστρέψουμε σε αυτό το σημείο όταν συζητήσουμε το θέμα της ανάπτυξης (elaboration) στην επόμενη ενότητα. Δεύτερον, σε καθεμία από τις αντιστοιχίσεις θυρών, οι θύρες της οντότητας της οποίας έχει εμφανιστεί το στιγμιότυπο συσχετίζονται με χωριστά στοιχεία του πίνακα σημάτων. Αυτό επιτρέπεται, αφού ένα σήμα που είναι σύνθετου τύπου, όπως ένας πίνακας, μπορεί να αντιμετωπιστεί ως συλλογή σημάτων, ένα ανά στοιχείο. Τρίτον, μερικά από τα σήματα που συνδέονται με τα στιγμιότυπα συστατικών είναι σήματα που δηλώνονται μέσα στο σώμα αρχιτεκτονικής που εσωκλείει τα στιγμιότυπα, της αρχιτεκτονικής registered, ενώ το σήμα clk είναι μια θύρα της οντότητας counter. Αυτό επεξηγεί πάλι το σημείο ότι μέσα σε ένα σώμα αρχιτεκτονικής, οι θύρες της αντίστοιχης οντότητας αντιμετωπίζονται ως σήματα.

ΕΙΚΟΝΑ 5-26



Ένα σχηματικό διάγραμμα για ένα διψήφιο μετρητή (two-digit counter) που χρησιμοποιεί την οντότητα reg4.

ΕΙΚΟΝΑ 5-27

```
entity counter is
     port (  clk, clr : in bit;
               q0, q1 : out digit );
end entity counter;
_____________________________________________________

architecture registered of counter is
    signal current_val0, current_val1, next_val0, next_val1 : digit;
begin
    val0_reg : entity work.reg4(struct)
         port map (  d0 => next_val0(0), d1 => next_val0(1),
                     d2 => next_val0(2), d3 => next_val0(3),
                     q0 => current_val0(0), q1 => current_val0(1),
                     q2 => current_val0(2), q3 => current_val0(3),
                     clk => clk, clr => clr );
    val1_reg : entity work.reg4(struct)
          port map ( d0 => next_val1(0), d1 => next_val1(1),
                     d2 => next_val1(2), d3 => next_val1(3),
                     q0 => current_val1(0), q1 => current_val1(1),
                     q2 => current_val1(2), q3 => current_val1(3),
                     clk => clk, clr => clr );
    incr0 : entity work.add_1(boolean_eqn) …;
    incr1 : entity work.add_1(boolean_eqn) …;
    buf0 : entity work.buf4(basic) …;
    buf1 : entity work.buf4(basic) …;
end architecture registered;
```

Μια δήλωση οντότητας για ένα διψήφιο δεκαδικό μετρητή (two-digit decimal counter), με ένα περίγραμμα ενός σώματος αρχιτεκτονικής που χρησιμοποιεί την οντότητα reg4.

Είδαμε στο παραπάνω παράδειγμα ότι μπορούμε να συσχετίσουμε χωριστές θύρες ενός στιγμιότυπου με μεμονωμένα στοιχεία ενός πραγματικού σήματος ενός σύνθετου τύπου, όπως ένας τύπος πίνακα ή εγγραφής. Εάν ένα στιγμιότυπο έχει μία σύνθετη θύρα, μπορούμε να γράψουμε συσχετίσεις με τον ανάποδο τρόπο - δηλαδή μπορούμε να συσχετίσουμε χωριστά πραγματικά σήματα με μεμονωμένα στοιχεία της θύρας. Αυτό καλείται μερικές φορές *συσχέτιση υποστοιχείου (subelement association)*. Για παράδειγμα, εάν το στιγμιότυπο DMA_buffer έχει μια θύρα status του τύπου FIFO_status, που έχει δηλωθεί ως

```
type FIFO_status is record
        nearly_full, nearly_empty, full, empty : bit;
    end record FIFO_status;
```

θα μπορούσαμε να συσχετίσουμε ένα σήμα με κάθε στοιχείο της θύρας ως εξής:

```
DMA_buffer : entity work.FIFO
    port map ( …, status.nearly_full => start_flush,
                  status.nearly_empty => end_flush,
                  status.full => DMA_buffer_full,
                  status.empty => DMA_buffer_empty, … );
```

Αυτό επεξηγεί δύο σημαντικά σημεία για τη συσχέτιση υποστοιχείου. Πρώτον, όλα τα στοιχεία της σύνθετης θύρας πρέπει να συσχετιστούν με ένα πραγματικό σήμα. Δεν μπορούμε να συσχετίσουμε μερικά στοιχεία και να αφήσουμε τα υπόλοιπα ασυσχέτιστα. Δεύτερον, όλες οι συσχετίσεις για μια συγκεκριμένη θύρα πρέπει να ομαδοποιηθούν σε μια λίστα συσχέτισης, χωρίς οποιεσδήποτε συσχετίσεις για άλλες θύρες μεταξύ αυτών.

Μπορούμε να χρησιμοποιήσουμε τη συσχέτιση υποστοιχείου για θύρες τύπου πίνακα γράφοντας ένα όνομα στοιχείου με δείκτη στην αριστερή πλευρά της συσχέτισης. Επιπλέον, μπορούμε να συσχετίσουμε ένα τμήμα της θύρας με ένα πραγματικό σήμα που είναι ένας μονοδιάστατος πίνακας, όπως παρουσιάζει το παρακάτω παράδειγμα.

ΠΑΡΑΔΕΙΓΜΑ

Υποθέστε ότι έχουμε μια οντότητα καταχωρητή, που έχει δηλωθεί όπως φαίνεται στο πάνω μέρος της Εικόνας 5-28. Οι θύρες d και q είναι πίνακες δυαδικών ψηφίων. Το σώμα αρχιτεκτονικής για ένα μικροεπεξεργαστή, που περιγράφεται στο κάτω μέρος της Εικόνας 5-28, εμφανίζει ένα στιγμιότυπο αυτής της οντότητα ως τον καταχωρητή κατάστασης προγράμματος (program status register, PSR). Τα μεμονωμένα δυαδικά ψηφία μέσα στον καταχωρητή αναπαριστούν τις σημαίες συνθήκης (condition flag) και διακοπής (interrupt flag), και το πεδίο από το ψηφίο 6 έως το ψηφίο 4 αναπαριστά το τρέχον επίπεδο προτεραιότητας διακοπής (interrupt priority level).

ΕΙΚΟΝΑ 5-28

```
entity reg is
     port ( d : in bit_vector(7 downto 0);
            q : out bit_vector(7 downto 0);
            clk : in bit );
end entity reg;
```

```
architecture RTL of microprocessor is
    signal interrupt_req : bit;
    signal interrupt_level : bit_vector(2 downto 0);
    signal carry_flag, negative_flag, overflow_flag, zero_flag : bit;
    signal program_status : bit_vector(7 downto 0);
    signal clk_PSR : bit;
    …
begin
    PSR : entity work.reg
        port map (  d(7) => interrupt_req,
                    d(6 downto 4) => interrupt_level,
                    d(3) => carry_flag, d(2) => negative_flag,
                    d(1) => overflow_flag, d(0) => zero_flag,
                    q => program_status,
                    clk => clk_PSR );
    …
end architecture RTL;
```

Μια δήλωση οντότητας για έναν καταχωρητή με θύρες τύπου πίνακα, και ένα περίγραμμα ενός σώματος αρχιτεκτονικής που εμφανίζει στιγμιότυπο της οντότητας. Η αντιστοίχιση θυρών περιλαμβάνει συσχετίσεις υποστοιχείων με μεμονωμένα στοιχεία και ένα τμήμα της θύρας d.

Στην αντιστοίχιση θυρών του στιγμιότυπου, η συσχέτιση υποστοιχείου χρησιμοποιείται για τη θύρα εισόδου d για να συνδέσει μεμονωμένα στοιχεία της θύρας με χωριστά πραγματικά σήματα της αρχιτεκτονικής. Ένα τμήμα της θύρας συνδέεται με το σήμα interrupt_level. Η θύρα εξόδου q, από την άλλη, συνδέεται ολόκληρη με το σήμα διανύσματος bit program_status.

Μπορούμε επίσης να χρησιμοποιήσουμε συσχέτιση υποστοιχείου για μια θύρα που είναι ενός τύπου πίνακα χωρίς περιορισμούς. Τα όρια του δείκτη της θύρας καθορίζονται από την ελάχιστη και τη μέγιστη τιμή του δείκτη που

χρησιμοποιείται στη λίστα συσχέτισης, και η κατεύθυνση του εύρους του δείκτη καθορίζεται από τον τύπο της θύρας. Για παράδειγμα, υποθέστε ότι δηλώνουμε μια οντότητα της πύλης and:

```
entity and_gate is
    port ( i : in bit_vector; y : out bit );
end entity and_gate;
```

και διάφορα σήματα:

```
signal serial_select, write_en, bus_clk, serial_wr : bit;
```

Μπορούμε να εμφανίσουμε στιγμιότυπο της οντότητας ως μία πύλη and 3-εισόδων:

```
serial_write_gate : entity work.and_gate
    port map ( i(1) => serial_select,
               i(2) => write_en,
               i(3) => bus_clk,
               y => serial_wr );
```

Δεδομένου ότι η θύρα i δεν έχει περιορισμούς, οι τιμές του δείκτη στις συσχετίσεις υποστοιχείου καθορίζουν τα όρια του δείκτη για αυτό το στιγμιότυπο. Η ελάχιστη τιμή είναι ένα και η μέγιστη τιμή είναι τρία. Ο τύπος της θύρας είναι bit_vector, το οποίο έχει μια αύξουσα σειρά δείκτη. Κατά συνέπεια, το εύρος του δείκτη για τη θύρα στο στιγμιότυπο είναι μια αύξουσα σειρά από το ένα έως το τρία.

Ο συντακτικός κανόνας για μια λίστα συσχέτισης θυρών δείχνει ότι μια θύρα ενός στιγμιότυπου συστατικού μπορεί να συσχετιστεί με μια παράσταση αντί για ένα σήμα. Σε αυτήν την περίπτωση, η τιμή της παράστασης χρησιμοποιείται ως σταθερή τιμή για τη θύρα καθ' όλη τη διάρκεια της προσομοίωσης. Εάν το πραγματικό υλικό είχε συντεθεί από το μοντέλο, η θύρα του στιγμιότυπου του συστατικού θα «δενόταν» σε μια σταθερή τιμή που καθορίζεται από την παράσταση. Η συσχέτιση με μια παράσταση είναι χρήσιμη όταν έχουμε μια οντότητα που παρέχεται ως τμήμα μιας βιβλιοθήκης, αλλά δεν χρειάζεται να χρησιμοποιήσουμε όλες τις λειτουργίες που παρέχονται από την οντότητα. Όταν συσχετίζουμε μια θύρα με μια παράσταση, η τιμή της παράστασης πρέπει να είναι *συνολικά στατική (globally static)*; που σημαίνει ότι πρέπει να είμαστε σε θέση να καθορίσουμε την τιμή από τις σταθερές που ορίζονται όταν αναπτύσσεται το μοντέλο. Έτσι, για παράδειγμα, η παράσταση δεν πρέπει να περιλαμβάνει καθόλου αναφορές σε σήματα.

ΠΑΡΑΔΕΙΓΜΑ

Δοθέντος ενός πολυπλέκτη 4-εισόδων που περιγράφεται από τη δήλωση οντότητας

```
entity mux4 is
    port ( i0, i1, i2, i3, sel0, sel1 : in bit;
           z : out bit );
end entity mux4;
```

μπορούμε να τον χρησιμοποιήσουμε ως πολυπλέκτη δύο-εισόδων εμφανίζοντας ένα στιγμιότυπό του ως εξής:

```
a_mux : entity work.mux4
    port map ( sel0 => select_line, i0 => line0, i1 => line1,
               z => result_line,
               sel1 => '0', i2 => '1', i3 => '1' );
```

Για αυτό το στιγμιότυπο συστατικού, το υψηλότερο ψηφίο επιλογής «καρφώνεται» στο '0', εξασφαλίζοντας ότι μόνο ένα από τα line0 ή line1 περνούν στην έξοδο. Έχουμε επίσης ακολουθήσει την πρακτική, που συστήνεται για πολλές οικογένειες λογικής, δένοντας τις αχρησιμοποίητες εισόδους σε μια σταθερή τιμή, σε αυτήν την περίπτωση στο '1'.

Μερικές οντότητες μπορούν να σχεδιαστούν ώστε να επιτρέπουν στις εισόδους τους να αφεθούν ανοικτές διευκρινίζοντας μιας προκαθορισμένη τιμή για μια θύρα. Όταν εμφανιστεί στιγμιότυπο της οντότητας, μπορούμε να διευκρινίσουμε ότι μια θύρα πρόκειται να αφεθεί ανοικτή με τη χρησιμοποίηση της λέξης-κλειδί **open** στη λίστα συσχέτισης θυρών.

ΠΑΡΑΔΕΙΓΜΑ

Η δήλωση της οντότητας and_or_inv περιλαμβάνει μια προκαθορισμένη τιμή '1' για καθεμία από τις θύρες εισόδους της, όπως πάλι παρουσιάζεται εδώ:

```
entity and_or_inv is
    port ( a1, a2, b1, b2 : in bit := '1';
           y : out bit );
end entity and_or_inv;
```

Μπορούμε να γράψουμε μια εμφάνιση συστατικού για να εκτελέσουμε τη συνάρτηση **not** ((A **and** B) **or** C) χρησιμοποιώντας αυτήν την οντότητα ως εξής:

```
f_cell : entity work.and_or_inv
    port map ( a1 => A, a2 => B, b1 => C, b2 => open, y => F );
```

Η θύρα b2 αφήνεται ανοικτή, έτσι υποθέτει την προκαθορισμένη τιμή '1' που διευκρινίζεται στη δήλωση της οντότητας.

Υπάρχει κάποια ομοιότητα μεταξύ της διευκρίνισης μιας προκαθορισμένης τιμής για μια θύρα εισόδου και της συσχέτισης μιας θύρας εισόδου με μια παράσταση. Και στις δύο περιπτώσεις η παράσταση πρέπει να είναι συνολικά στατική (δηλαδή πρέπει να είμαστε σε θέση να καθορίσουμε την τιμή της όταν αναπτύσσεται το μοντέλο). Η διαφορά είναι ότι μια προκαθορισμένη τιμή χρησιμοποιείται μόνο εάν η θύρα αφήνεται ανοικτή όταν εμφανίζεται ένα στιγμιότυπο της οντότητας, ενώ η συσχέτιση με μια παράσταση διευκρινίζει ότι η τιμή της παράστασης πρόκειται να χρησιμοποιηθεί για να οδηγήσει τη θύρα για ολόκληρη την προσομοίωση ή τη ζωή του στιγμιότυπου του συστατικού. Εάν μια θύρα δηλωθεί με μια προκαθορισμένη τιμή και έπειτα συσχετιστεί με μια παράσταση, χρησιμοποιείται η τιμή της παράστασης, αγνοώντας την προκαθορισμένη τιμή.

Θύρες εξόδου και αμφίδρομες θύρες μπορούν επίσης να αφεθούν ασυσχέτιστες χρησιμοποιώντας τη λέξη-κλειδί **open**, υπό τον όρο ότι δεν είναι τύπου πίνακα χωρίς περιορισμούς. Εάν μια θύρα τύπου **out** αφεθεί ανοικτή, οποιαδήποτε τιμή οδηγείται από την οντότητα αγνοείται. Εάν μια θύρα τύπου **inout** αφεθεί ανοικτή, η τιμή που χρησιμοποιείται εσωτερικά από την οντότητα (*πραγματική τιμή – effective value*) είναι η τιμή που οδηγεί η οντότητα προς τη θύρα.

Ένα τελευταίο σημείο που πρέπει να θίξουμε για τις ασυσχέτιστες θύρες είναι ότι μπορούμε απλά να παραλείψουμε μια θύρα από μια λίστα συσχέτισης θυρών για να διευκρινίσουμε ότι παραμένει ανοικτή. Έτσι, δοθέντος μιας οντότητας που δηλώνεται ως εξής:

```
entity and3 is
    port ( a, b, c : in bit := '1';
           z, not_z : out bit);
end entity and3;
```

η εμφάνιση ενός στιγμιότυπου συστατικού

```
g1 : entity work.and3 port map ( a => s1, b => s2, not_z => ctrl1 );
```

έχει την ίδια έννοια με

```
g1 : entity work.and3 port map ( a => s1, b => s2, not_z => ctrl1,
c => open, z => open );
```

Η διαφορά είναι ότι η δεύτερη έκδοση καθιστά σαφές ότι οι αχρησιμοποίητες θύρες αφήνονται σκόπιμα ανοικτές, και δεν αγνοήθηκαν τυχαία κατά τη διαδικασία σχεδίασης. Αυτά είναι χρήσιμες πληροφορίες για κάποιον που διαβάζει το μοντέλο.

VHDL-87

Η VHDL-87 δεν επιτρέπει άμεση εμφάνιση στιγμιότυπου. Αντ' αυτού, πρέπει να δηλώσουμε ένα *συστατικό (component)* με μια παρόμοια διασύνδεση με την οντότητα, να εμφανίσουμε ένα στιγμιότυπο του συστατικού και να *δεσμεύσουμε (bind)* κάθε στιγμιότυπο του συστατικού με την οντότητα και ένα σχετικό σώμα αρχιτεκτονικής.

Η VHDL-87 δεν επιτρέπει τη συσχέτιση μιας παράστασης με μια θύρα σε μια απεικόνιση θυρών. Εντούτοις, μπορούμε να επιτύχουμε ένα παρόμοιο αποτέλεσμα με τη δήλωση ενός σήματος, την αρχικοποίησή του στην τιμή της παράστασης και τη συσχέτιση του σήματος με τη θύρα. Για παράδειγμα, εάν δηλώσουμε δύο σήματα

```
signal tied_0 : bit := '0';
signal tied_1 : bit := '1';
```

μπορούμε να ξαναγράψουμε την απεικόνιση θυρών ως εξής

```
port map ( sel0 => select_line, i0 => line0, i1 => line1,
           z => result_line,
           sel1 => tied_0, i2 => tied_1, i3 => tied_1 );
```

5.5 Επεξεργασία Σχεδίασης

Τώρα που έχουμε δει πώς μια σχεδίαση μπορεί να περιγραφεί υπό τη μορφή οντοτήτων, αρχιτεκτονικών, στιγμιότυπων συστατικών, σημάτων και διεργασιών, είναι ώρα να δούμε μια εφαρμογή στην πράξη. Μια VHDL περιγραφή μίας σχεδίασης συνήθως χρησιμοποιείται για να προσομοιώσει τη σχεδίαση και ίσως για να συνθέσει το υλικό. Αυτό εμπεριέχει την επεξεργασία της περιγραφής χρησιμοποιώντας εργαλεία βασισμένα σε υπολογιστή για να δημιουργήσει είτε ένα πρόγραμμα προσομοίωσης (simulation program) που πρόκειται να τρέξει ή μια λίστα συνδέσεων υλικού (hardware netlist) που πρόκειται να κατασκευάσει. Και η προσομοίωση και η σύνθεση απαιτούν δύο προπαρασκευαστικά βήματα: ανάλυση (analysis) και ανάπτυξη (elaboration). Η προσομοίωση έπειτα εμπεριέχει την εκτέλεση του αναπτυγμένου μοντέλου, ενώ η σύνθεση εμπεριέχει τη δημιουργία μιας λίστας συνδέσεων των θεμελιωδών στοιχείων του κυκλώματος που εκτελεί την ίδια λειτουργία με το αναπτυγμένο μοντέλο. Σε αυτήν την ενότητα, εξετάζουμε τις λειτουργίες ανάλυσης, ανάπτυξης και εκτέλεσης που εισάγονται στο Κεφάλαιο 1.

5.5.1 Ανάλυση

Το πρώτο βήμα στην επεξεργασία μίας σχεδίασης είναι να αναλυθούν οι VHDL περιγραφές. Μια σωστή περιγραφή πρέπει να τηρεί τους κανόνες σύνταξης και σημασιολογίας που έχουμε ήδη συζητήσει επί μακρόν. Ένας *αναλυτής (analyzer)* είναι ένα εργαλείο που ελέγχει την τήρηση των κανόνων. Εάν μια περιγραφή αποτύχει να ανταποκριθεί σε έναν κανόνα, ο αναλυτής παράγει ένα μήνυμα που υποδεικνύει τη θέση του προβλήματος και ποιος κανόνας παραβιάστηκε. Μπορούμε έπειτα να διορθώσουμε το λάθος και να ξαναδοκιμάσουμε την ανάλυση. Ένα άλλο έργο που εκτελείται από τον αναλυτή στα περισσότερα συστήματα VHDL είναι να μεταφράσει την περιγραφή σε μια εσωτερική μορφή της οποίας η επεξεργασία να μπορεί να γίνει ευκολότερα από τα υπόλοιπα εργαλεία. Ανεξαρτήτως εάν μια τέτοια μετάφραση γίνει ή όχι, ο αναλυτής τοποθετεί κάθε αναλυθείσα περιγραφή σε μια *βιβλιοθήκη σχεδίασης (design library)*.

Μια πλήρης VHDL περιγραφή συνήθως αποτελείται από διάφορες δηλώσεις οντοτήτων και τα αντίστοιχα σώματα αρχιτεκτονικής τους. Κάθε ένα από αυτά καλείται μία *μονάδα σχεδίασης (design unit)*. Η οργάνωση μίας σχεδίασης ως ιεραρχία λειτουργικών μονάδων, παρά ως μία μεγάλη επίπεδη σχεδίαση, είναι ορθή πρακτική σχεδίασης. Καθιστά πολύ ευκολότερη την κατανόηση και τη διαχείριση της περιγραφής.

Ο αναλυτής αναλύει κάθε μονάδα σχεδίασης χωριστά και τοποθετεί την εσωτερική μορφή στη βιβλιοθήκη ως *μονάδα βιβλιοθήκης (library unit)*. Εάν μια μονάδα που αναλύεται χρησιμοποιεί μια άλλη μονάδα, ο αναλυτής εξάγει πληροφορίες για την άλλη μονάδα από τη βιβλιοθήκη, για να ελέγξει ότι η μονάδα χρησιμοποιείται σωστά. Για παράδειγμα, εάν ένα σώμα αρχιτεκτονικής εμφανίζει το στιγμιότυπο μιας οντότητας, ο αναλυτής πρέπει να ελέγξει τον αριθμό, τον τύπο και την κατάσταση των θυρών της οντότητας για να σιγουρευτεί ότι το στιγμιότυπο είναι σωστό. Για να το κάνει αυτό, απαιτεί να έχει αναλυθεί προηγουμένως η οντότητα και να έχει αποθηκευτεί στη βιβλιοθήκη. Κατά συνέπεια, βλέπουμε ότι υπάρχουν σχέσεις εξάρτησης μεταξύ των μονάδων της βιβλιοθήκης σε μια πλήρη περιγραφή που επιβάλλουν μια σειρά στην ανάλυση των αρχικών μονάδων σχεδίασης.

Για να διευκρινίσουμε αυτό το σημείο, διαιρούμε τις μονάδες σχεδίασης σε *πρωτεύουσες μονάδες (primary units)*, που περιλαμβάνουν τις δηλώσεις οντοτήτων, και δευτερεύουσες μονάδες *(secondary units)*, που περιλαμβάνουν τα σώματα αρχιτεκτονικής. Μια πρωτεύουσα μονάδα καθορίζει την εξωτερική άποψη ή τη διασύνδεση μιας λειτουργικής μονάδας, ενώ μια δευτερεύουσα μονάδα περιγράφει μια υλοποίηση της λειτουργικής μονάδας. Κατά συνέπεια η δευτερεύουσα μονάδα εξαρτάται από την αντίστοιχη πρωτεύουσα μονάδα και πρέπει να αναλυθεί αφότου έχει αναλυθεί η πρωτεύουσα μονάδα. Επιπλέον, μια μονάδα βιβλιοθήκης μπορεί να αντλήσει πληροφορίες που έχουν καθοριστεί για κάποια άλλη πρωτεύουσα μονάδα, όπως στην περίπτωση ενός σώματος αρχιτεκτονικής που εμφανίζει στιγμιότυπο κάποιας άλλης οντότητας. Σε αυτήν την περίπτωση, υπάρχει μια περαιτέρω εξάρτηση μεταξύ της δευτερεύουσας μονάδας και της πρωτεύουσας μονάδας στην οποία αναφέρεται. Κατά συνέπεια μπορούμε να κατασκευάσουμε ένα δίκτυο εξαρτήσεων των μονάδων επάνω στις πρωτεύουσες μονάδες. Η ανάλυση πρέπει να γίνει με μια σειρά ώστε μια μονάδα να αναλύεται πριν από οποιαδήποτε μονάδα που εξαρτάται από αυτήν. Επιπλέον, όποτε αλλάζουμε και αναλύουμε ξανά μια πρωτεύουσα μονάδα, όλες οι εξαρτώμενες μονάδες πρέπει επίσης να αναλυθούν ξανά. Σημειώστε, εντούτοις, ότι δεν υπάρχει κανένας τρόπος με τον οποίο οποιαδήποτε μονάδα μπορεί να εξαρτάται από μια δευτερεύουσα μονάδα - αυτός είναι ο λόγος που καθιστά μια μονάδα δευτερεύουσα. Αυτό μπορεί να φανεί μάλλον περίπλοκο, και πράγματι, σε μια μεγάλη σχεδίαση, οι σχέσεις εξάρτησης μπορεί να διαμορφώνουν ένα σύνθετο δίκτυο. Για αυτόν το λόγο, τα περισσότερα συστήματα VHDL περιλαμβάνουν εργαλεία για να διαχειριστούν τις εξαρτήσεις, αναλύοντας ξανά αυτόματα τις μονάδες όπου είναι απαραίτητο για να εξασφαλίσουν ότι μια μη-ενημερωμένη μονάδα δεν χρησιμοποιείται ποτέ.

ΠΑΡΑΔΕΙΓΜΑ

Η αρχιτεκτονική δομής της λειτουργικής μονάδας counter, που περιγράφεται στην Εικόνα 5-27, οδηγεί στο δίκτυο εξαρτήσεων που παρουσιάζεται στην Εικόνα 5-29. Μια πιθανή σειρά μεταγλώττισης για αυτό το σύνολο μονάδων σχεδίασης είναι

entity edge_triggered_Dff

architecture behav of edge_triggered_Dff

entity reg4

architecture struct of reg4

entity add_1

architecture boolean_eqn of add_1

entity buf4

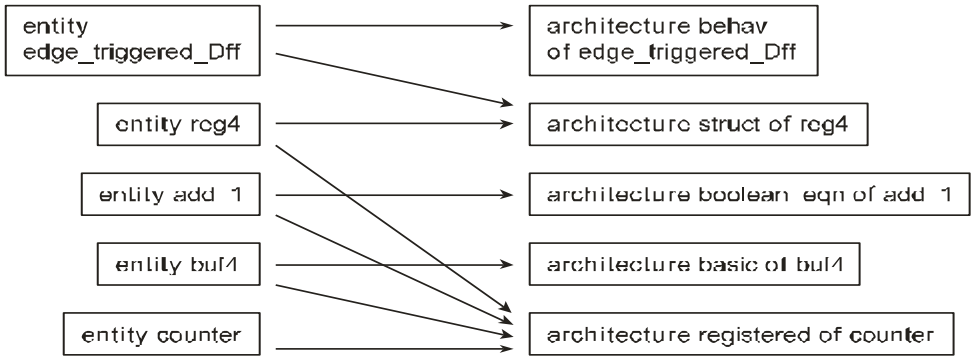
architecture basic of buf

entity counter

architecture registered of counter

Σε αυτήν τη σειρά, κάθε πρωτεύουσα μονάδα αναλύεται αμέσως πριν από την αντίστοιχή της δευτερεύουσα μονάδα, και κάθε πρωτεύουσα μονάδα αναλύεται πριν από οποιαδήποτε δευτερεύουσα μονάδα που εμφανίζει στιγμιότυπο αυτής. Αυτή δεν είναι η μόνη πιθανή σειρά μεταγλώττισης. Μια άλλη εναλλακτική λύση είναι να αναλυθούν όλες οι δηλώσεις οντοτήτων πρώτα, και κατόπιν να αναλυθούν τα σώματα αρχιτεκτονικής με αυθαίρετη σειρά.

ΕΙΚΟΝΑ 5-29



Το δίκτυο εξαρτήσεων για τη λειτουργική μονάδα counter. *Τα βέλη δείχνουν από μια πρωτεύουσα μονάδα σε μια εξαρτώμενη δευτερεύουσα μονάδα.*

5.5.2 Βιβλιοθήκες Σχεδίασης, Φράσεις Βιβλιοθήκης και Φράσεις Χρήσης

Μέχρι τώρα, δεν έχουμε πει πραγματικά τι είναι μια βιβλιοθήκη σχεδίασης, εκτός από το ότι είναι το μέρος όπου αποθηκεύονται οι μονάδες βιβλιοθήκης. Πράγματι, αυτό είναι που καθορίζεται από τη προδιαγραφή της γλώσσας VHDL, αφού για να πάει κάποιος πιο πέρα πρέπει να εισχωρήσει στην περιοχή του λειτουργικού συστήματος του υπολογιστή υπηρεσίας στον οποίο τρέχουν τα εργαλεία της VHDL. Μερικά συστήματα μπορεί να χρησιμοποιούν μια βάση δεδομένων για να αποθηκεύουν τις μονάδες που αναλύονται, ενώ άλλα μπορεί απλά να χρησιμοποιούν έναν κατάλογο στο σύστημα αρχείων του υπολογιστή υπηρεσίας ως βιβλιοθήκη σχεδίασης. Η τεκμηρίωση για κάθε σουίτα εργαλείων VHDL προσδιορίζει τι πρέπει να γνωρίζουμε γύρω από το πώς η σουίτα μεταχειρίζεται τις βιβλιοθήκες σχεδίασης.

Μια σουίτα εργαλείων VHDL πρέπει επίσης να παρέχει μερικά μέσα για τη χρησιμοποίηση ενός αριθμού ξεχωριστών βιβλιοθηκών σχεδίασης. Όταν μια σχεδίαση αναλύεται, διορίζουμε μια από τις βιβλιοθήκες ως *βιβλιοθήκη εργασίας (working library)*, και η αναλυθείσα σχεδίαση αποθηκεύεται σε αυτήν την βιβλιοθήκη. Χρησιμοποιούμε το ειδικό όνομα βιβλιοθήκης work στα VHDL μοντέλα μας για να αναφερθούμε στην τρέχουσα βιβλιοθήκη εργασίας. Έχουμε δει παραδείγματα αυτού στις προτάσεις εμφάνισης στιγμιότυπου συστατικού σε αυτό το κεφάλαιο, στις οποίες το στιγμιότυπο μιας οντότητας που έχει προηγουμένως αναλυθεί εμφανίζεται σε ένα σώμα αρχιτεκτονικής.

Εάν χρειάζεται να έχουμε πρόσβαση στις μονάδες βιβλιοθήκης που είναι αποθηκευμένες σε άλλες βιβλιοθήκες, αναφερόμαστε στις βιβλιοθήκες ως *βιβλιοθήκες πόρων (resource libraries)*. Κάνουμε κάτι τέτοιο εισάγοντας μια *φράση βιβλιοθήκης (library clause)* αμέσως πριν από μια μονάδα σχεδίασης που προσπελάζει τις βιβλιοθήκες πόρων. Ο συντακτικός κανόνας για μια φράση βιβλιοθήκης είναι

library_clause ⇐ **library** identifier { , … } ;

Τα αναγνωριστικά χρησιμοποιούνται από τον αναλυτή και το λειτουργικό σύστημα του υπολογιστή υπηρεσίας για να εντοπίσουν τις βιβλιοθήκες σχεδίασης, έτσι ώστε οι μονάδες που περιλαμβάνονται σε αυτές να μπορούν να χρησιμοποιηθούν στην περιγραφή που αναλύεται. Ο ακριβής τρόπος που χρησιμοποιούνται τα αναγνωριστικά ποικίλλει ανάμεσα στις διαφορετικές σουίτες εργαλείων και δεν καθορίζεται από την προδιαγραφή της γλώσσας VHDL. Σημειώστε ότι δεν χρειάζεται να εισάγουμε το όνομα της βιβλιοθήκης work σε μια πρόταση βιβλιοθήκης - η τρέχουσα βιβλιοθήκη εργασίας είναι αυτόματα διαθέσιμη.

ΠΑΡΑΔΕΙΓΜΑ

Υποθέστε ότι εργαζόμαστε σε ένα κομμάτι ενός μεγάλου σχεδιαστικού έργου με το κωδικό όνομα Wasp, και χρησιμοποιούμε τμήματα πρότυπων κυψελών (standard cells) που παρέχονται από την εταιρεία Widget Designs, Α.Ε. Ο διαχειριστής του συστήματός μας έχει φορτώσει τη βιβλιοθήκη σχεδίασης για τις κυψέλες της Widget σε έναν κατάλογο στο σύστημα αρχείων του σταθμού εργασίας μας με το όνομα /local/widget/cells, και ο υπεύθυνος του έργου έχει εγκαταστήσει μια άλλη βιβλιοθήκη σχεδίασης στον κατάλογο /projects/wasp/lib για μερικές εσωτερικές κυψέλες που πρέπει να χρησιμοποιήσουμε. Συμβουλευόμαστε το εγχειρίδιο χρήσης του VHDL αναλυτή μας και χρησιμοποιούμε τις εντολές του λειτουργικού συστήματος για να οργανώσουμε την κατάλληλη αντιστοίχιση από τα αναγνωριστικά widget_cells και wasp_lib σε αυτούς τους καταλόγους βιβλιοθηκών. Μπορούμε έπειτα να εμφανίσουμε στιγμιότυπα οντοτήτων από αυτές τις βιβλιοθήκες, μαζί με τις οντότητες που έχουμε προηγουμένως αναλύσει, στη δική μας βιβλιοθήκη εργασίας, όπως φαίνεται στην Εικόνα 5-30.

ΕΙΚΟΝΑ 5-30

```
library widget_cells, wasp_lib;
architecture cell_based of filter is
    -- δήλωση σημάτων, κτλ.
    …
begin
    clk_pad : entity wasp_lib.in_pad
        port map ( i => clk, z => filter_clk );
    accum : entity widget_cells.reg32
        port map (   en => accum_en, clk => filter_clk, d => sum,
                     q => result );
    alu : entity work.adder
        port map ( a => alu_op1, b => alu_op2, y => sum, c => carry );
    -- άλλες εμφανίσεις στιγμιότυπων συστατικών
…
end architecture cell_based;
```

Ένα περίγραμμα μιας μονάδας βιβλιοθήκης που αναφέρεται σε οντότητες από τις βιβλιοθήκες πόρων widget_cells *και* wasp_lib.

Εάν πρέπει να κάνουμε συχνή αναφορά στις μονάδες βιβλιοθήκης από μια βιβλιοθήκη σχεδίασης, μπορούμε να εισάγουμε μια *φράση χρήσης (use clause)* στο μοντέλο μας ώστε να αποφύγουμε να γράφουμε το όνομα της βιβλιοθήκης κάθε φορά. Οι απλουστευμένοι συντακτικοί κανόνες είναι

use_clause ⇐ **use** selected_name { , … } ;

selected_name ⇐ name . (identifier I **all**)

Εάν εισάγουμε μια πρόταση χρήσης με ένα όνομα βιβλιοθήκης ως πρόθεμα του επιλεγμένου ονόματος (που προηγείται της τελείας), και ένα όνομα μονάδας από τη βιβλιοθήκη ως κατάληξη (μετά από την τελεία), η μονάδα βιβλιοθήκης γίνεται *άμεσα ορατή (directly visible)*. Αυτό σημαίνει ότι οι επόμενες αναφορές του μοντέλου στη μονάδα βιβλιοθήκης δεν χρειάζεται να συνοδεύουν το όνομα της μονάδας βιβλιοθήκης με το πρόθεμα του ονόματος της βιβλιοθήκης. Για παράδειγμα, θα μπορούσαμε να προτάσσουμε του σώματος αρχιτεκτονικής στο προηγούμενο παράδειγμα τις παρακάτω φράσεις βιβλιοθήκης και χρήσης:

```
library widget_cells, wasp_lib;
use widget_cells.reg32;
```

Αυτό καθιστά το reg32 άμεσα ορατό μέσα στο σώμα αρχιτεκτονικής, έτσι μπορούμε να παραλείψουμε το όνομα βιβλιοθήκης όταν αναφερόμαστε σε αυτό στα στιγμιότυπα του συστατικού. Για παράδειγμα:

```
accum : entity reg32
    port map ( en => accum_en, clk => filter_clk, d => sum,
               q => result );
```

Εάν εισάγουμε τη λέξη-κλειδί **all** σε μια φράση χρήσης, όλες οι μονάδες βιβλιοθήκης μέσα στη βιβλιοθήκη που ονομάζουμε γίνονται άμεσα ορατές. Για παράδειγμα, εάν θέλαμε να καταστήσουμε άμεσα ορατές όλες τις μονάδες της βιβλιοθήκης του έργου Wasp, θα μπορούσαμε να εισάγουμε μια μονάδα βιβλιοθήκης με τη φράση χρήσης:

```
use wasp_lib.all;
```

Προσοχή πρέπει να δοθεί όταν χρησιμοποιούμε αυτή τη μορφή φράσης χρήσης με διάφορες βιβλιοθήκες ταυτόχρονα. Εάν δύο βιβλιοθήκες περιέχουν μονάδες βιβλιοθήκης με το ίδιο όνομα, η VHDL αποφεύγει την ασάφεια με το να μην καταστήσει καμία από αυτές άμεσα ορατή. Η λύση είναι είτε να χρησιμοποιήσουμε το πλήρες επιλεγμένο όνομα για να αναφερθούμε στη συγκεκριμένη μονάδα βιβλιοθήκης που επιθυμούμε, είτε να εισάγουμε στις φράσεις χρήσης μόνο εκείνες τις μονάδες βιβλιοθήκης που χρειαζόμαστε πραγματικά σε ένα μοντέλο.

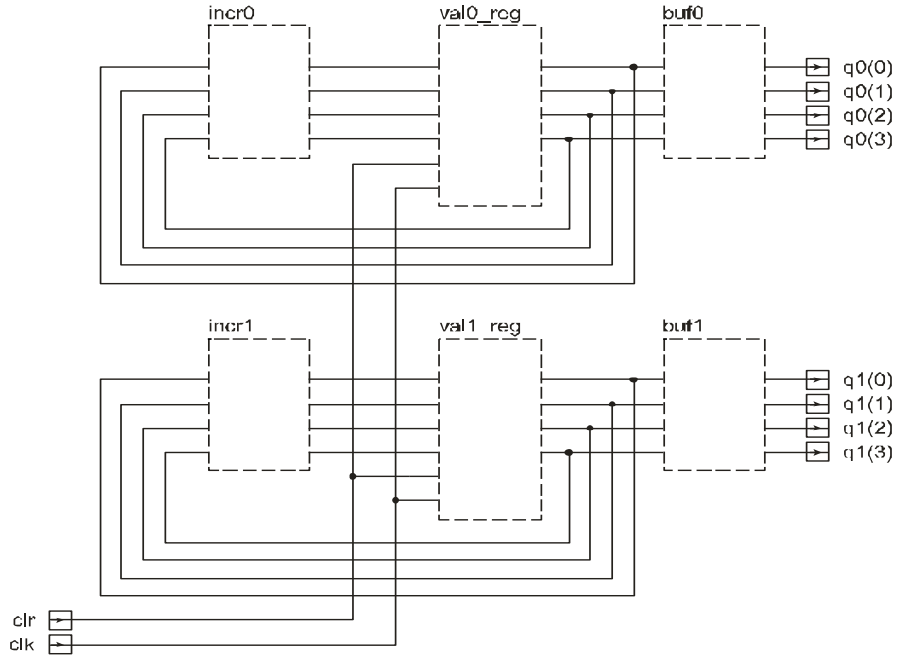
Οι φράσεις χρήσης μπορούν επίσης να χρησιμοποιηθούν για να καταστήσουν τα ονόματα από τα πακέτα άμεσα ορατά.

5.5.3 Ανάπτυξη

Μόλις αναλυθούν όλες οι μονάδες σε μια ιεραρχία σχεδίασης, η ιεραρχία σχεδίασης μπορεί να *αναπτυχθεί (elaborated)*. Η επίδραση της ανάπτυξης είναι να «συμπληρώσει τα κενά» της ιεραρχίας, παράγοντας μια συλλογή διεργασιών που διασυνδέονται με *καλωδιώσεις (nets)*. Αυτό γίνεται αντικαθιστώντας με τα περιεχόμενα ενός σώματος αρχιτεκτονικής κάθε εμφάνιση στιγμιότυπου της αντίστοιχης οντότητάς του. Κάθε καλωδίωση στην αναπτυγμένη σχεδίαση αποτελείται από ένα σήμα και τις θύρες των σωμάτων αρχιτεκτονικής στις οποίες συνδέεται το σήμα. (Θυμηθείτε ότι η θύρα μιας οντότητας αντιμετωπίζεται ως ένα σήμα μέσα σε ένα αντίστοιχο σώμα αρχιτεκτονικής.) Ας περιγράψουμε πώς προχωράει η ανάπτυξη, επεξηγώντας τη βήμα προς βήμα με ένα παράδειγμα.

Η ανάπτυξη είναι μια αναδρομική λειτουργία, που αρχίζει με την οντότητα που βρίσκεται στην κορυφή μιας ιεραρχίας σχεδίασης. Χρησιμοποιούμε το παράδειγμα του counter από την Εικόνα 5-27 ως την οντότητα στο υψηλότερο επίπεδο. Το πρώτο βήμα είναι να δημιουργηθούν οι θύρες της οντότητας. Έπειτα, επιλέγεται ένα σώμα αρχιτεκτονικής που αντιστοιχεί στην οντότητα. Εάν δεν διευκρινίζουμε ρητά ποιο σώμα αρχιτεκτονικής θα επιλεγεί, χρησιμοποιείται το σώμα αρχιτεκτονικής που έχει αναλυθεί πιο πρόσφατα. Για αυτήν την απεικόνιση, χρησιμοποιούμε την αρχιτεκτονική registered. Αυτό το σώμα αρχιτεκτονικής αναπτύσσεται έπειτα, πρώτα με τη δημιουργία όλων των σημάτων που δηλώνει, και στη συνέχεια με την ανάπτυξη κάθε μιας από τις ταυτόχρονες προτάσεις στο σώμα του. Η Εικόνα 5-31 παρουσιάζει τη σχεδίαση counter με τα σήματα που δημιουργούνται.

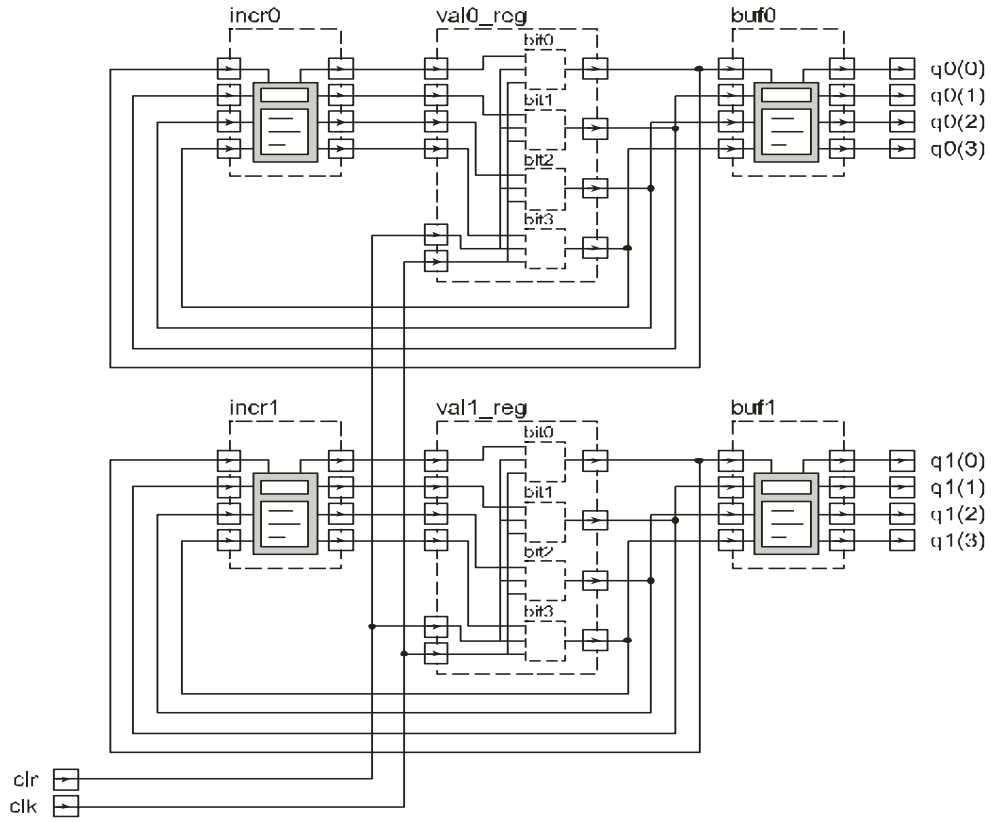
ΕΙΚΟΝΑ 5-31



Το πρώτο στάδιο της ανάπτυξης της οντότητας counter. *Έχουν δημιουργηθεί οι θύρες, έχει επιλεγεί η αρχιτεκτονική* registered*, και έχουν δημιουργηθεί τα σήματα της αρχιτεκτονικής.*

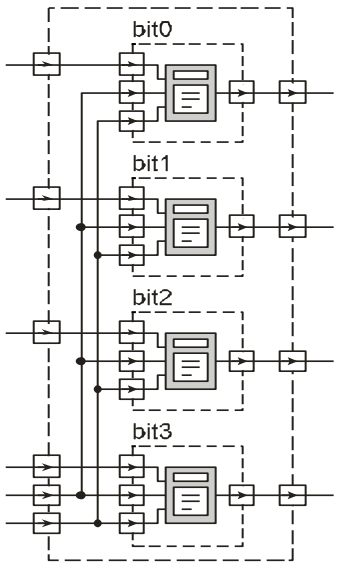
Οι ταυτόχρονες προτάσεις σε αυτήν την αρχιτεκτονική είναι όλες προτάσεις εμφάνισης στιγμιότυπου συστατικού. Καθεμία από αυτές αναπτύσσεται δημιουργώντας νέα στιγμιότυπα των θυρών που καθορίζονται από την οντότητα που έχει εμφανιστεί το στιγμιότυπό της και συνδέοντάς τες με τις καλωδιώσεις που αναπαριστώνται από τα σήματα με τα οποία συσχετίζονται. Κατόπιν η εσωτερική δομή του σώματος αρχιτεκτονικής που έχει καθοριστεί για το στιγμιότυπο της οντότητας αντιγράφεται στη θέση του στιγμιότυπου του συστατικού, όπως φαίνεται στην Εικόνα 5-32. Οι αρχιτεκτονικές που αντικαθιστούν τα στιγμιότυπα των οντοτήτων add_1 και buf4 είναι και οι δύο αρχιτεκτονικές συμπεριφοράς, αποτελούμενες από διεργασίες που διαβάζουν τις θύρες εισόδου και κάνουν αναθέσεις στις θύρες εξόδου. Ως εκ τούτου η ανάπτυξη είναι πλήρης για αυτές τις αρχιτεκτονικές. Εντούτοις, η αρχιτεκτονική struct, που αντικαθιστά κάθε μια από τα στιγμιότυπα του reg4, περιέχει περαιτέρω σήματα και στιγμιότυπα συστατικών. Ως εκ τούτου αναπτύσσονται με τη σειρά τους, παράγοντας τη δομή που φαίνεται στην Εικόνα 5-33 για κάθε στιγμιότυπο. Έχουμε φθάσει τώρα σε ένα στάδιο όπου έχουμε μια συλλογή καλωδιώσεων που περιλαμβάνουν σήματα και θύρες, και τις διεργασίες που διαβάζουν και οδηγούν τις καλωδιώσεις.

ΕΙΚΟΝΑ 5-32



Η σχεδίαση του μετρητή αναπτύχθηκε περαιτέρω. Αρχιτεκτονικές συμπεριφοράς αποτελούμενες απλά από διεργασίες, έχουν αντικαταστήσει τα στιγμιότυπα των οντοτήτων add_1 *και* buf4*. Μια αρχιτεκτονική δομή έχει αντικαταστήσει κάθε στιγμιότυπο της οντότητας* reg4.

ΕΙΚΟΝΑ 5-33



Ένας καταχωρητής μέσα στη δομή του μετρητή που αναπτύχθηκε προς τα κάτω μέχρι τις αρχιτεκτονικές που αποτελούνται μόνο από διεργασίες και σήματα.

Κάθε πρόταση διεργασίας στη σχεδίαση αναπτύσσεται δημιουργώντας νέα στιγμιότυπα των μεταβλητών που δηλώνει και δημιουργώντας έναν οδηγό για κάθε ένα από τα σήματα για τα οποία έχει προτάσεις ανάθεσης σήματος. Οι οδηγοί ενώνονται στις καλωδιώσεις που περιέχουν τα σήματα που οδηγούν. Για παράδειγμα, η διεργασία store μέσα στο bit0 του val0_reg έχει έναν οδηγό για τη θύρα q, η οποία είναι μέρος της καλωδίωσης που βασίζεται στο σήμα current_val0(0).

Μόλις αναπτυχθούν όλες τα στιγμιότυπα συστατικών και όλες οι διεργασίες που θα προκύψουν, η ανάπτυξη της ιεραρχίας σχεδίασης έχει ολοκληρωθεί. Έχουμε τώρα μια πλήρως συμπληρωμένη έκδοση της σχεδίασης, που αποτελείται από διάφορα στιγμιότυπα διεργασιών και έναν αριθμό καλωδιώσεων που τα συνδέουν. Σημειώστε ότι υπάρχουν πολλά ξεχωριστά στιγμιότυπα για μερικές από τις διεργασίες, ένα για κάθε χρήση μιας οντότητας που περιέχει τη διεργασία, και κάθε στιγμιότυπο διεργασίας έχει την δική του ξεχωριστή έκδοση των μεταβλητών της διεργασίας. Κάθε καλωδίωση στη σχεδίαση που έχει αναπτυχθεί αποτελείται από ένα σήμα, μια συλλογή θυρών που συσχετίζονται με αυτό και έναν οδηγό μέσα σε ένα στιγμιότυπο διεργασίας.

5.5.4 Εκτέλεση

Τώρα που έχουμε μια ιεραρχία σχεδίασης που έχει αναπτυχθεί, μπορούμε να την εκτελέσουμε για να προσομοιώσουμε τη λειτουργία του συστήματος που περιγράφει. Ένα μεγάλο μέρος της προηγούμενής μας συζήτησης για τις προτάσεις της VHDL αναλώθηκε στο τι συμβαίνει όταν αυτές εκτελούνται, έτσι εδώ δεν εξετάζουμε πάλι την εκτέλεση των προτάσεων. Αντ' αυτού, επικεντρωνόμαστε στον αλγόριθμο προσομοίωσης που εισάγεται στο Κεφάλαιο 1.

Θυμηθείτε ότι ο αλγόριθμος προσομοίωσης αποτελείται μια φάση αρχικοποίησης (initialization phase) που ακολουθείται από έναν επαναλαμβανόμενο κύκλο προσομοίωσης (repeated simulation cycle). Ο προσομοιωτής διατηρεί ένα ρολόι για να μετρήσει τη διέλευση του χρόνου προσομοίωσης. Στη φάση αρχικοποίησης, ο χρόνος προσομοίωσης τίθεται στο μηδέν. Κάθε οδηγός αρχικοποιείται για να οδηγήσει το σήμα του με την αρχική τιμή που δηλώνεται για το σήμα ή την προκαθορισμένη τιμή για το σήμα εάν καμία αρχική τιμή δεν έχει δηλωθεί. Έπειτα, κάθε ένα από τα στιγμιότυπα διεργασίας στη σχεδίαση αρχίζει και εκτελεί τις ακολουθιακές προτάσεις στο σώμα του. Γράφουμε συνήθως ένα μοντέλο έτσι ώστε τουλάχιστον μερικές από αυτές τις αρχικές προτάσεις να χρονοπρογραμματίσουν μερικές συναλλαγές που θα θέσουν την προσομοίωση σε εξέλιξη, και στη συνέχεια να αναστείλουν την εκτέλεση με μια πρόταση wait. Όταν όλα τα στιγμιότυπα διεργασίας αναστείλουν την εκτέλεσή τους, η αρχικοποίηση έχει ολοκληρωθεί και ο προσομοιωτής μπορεί να αρχίσει τον πρώτο κύκλο προσομοίωσης.

Στην αρχή ενός κύκλου προσομοίωσης, μπορεί να υπάρξουν διάφοροι οδηγοί με συναλλαγές που έχουν χρονοπρογραμματιστεί για αυτούς και διάφορα στιγμιότυπα διεργασίας που έχουν χρονοπρογραμματίσει χρονικές υπερβάσεις (timeouts). Το πρώτο βήμα στον κύκλο προσομοίωσης είναι να προχωρήσει το ρολόι του χρόνου προσομοίωσης στη νωρίτερη χρονική στιγμή κατά την οποία έχει χρονοπρογραμματιστεί μια συναλλαγή ή η χρονική υπέρβαση μιας διεργασίας. Δεύτερον, όλες οι συναλλαγές που είναι χρονοπρογραμματισμένες για αυτήν τη χρονική στιγμή εκτελούνται, ενημερώνοντας τα αντίστοιχα σήματα και προκαλώντας ενδεχομένως γεγονότα σε αυτά τα σήματα. Τρίτον, όλα τα στιγμιότυπα διεργασίας που είναι ευαίσθητα σε οποιαδήποτε από αυτά τα γεγονότα ξαναρχίζουν την εκτέλεσή τους. Επιπλέον, τα στιγμιότυπα διεργασίας των οποίων η χρονική υπέρβαση λήγει στον τρέχοντα χρόνο προσομοίωσης ξαναρχίζουν την εκτέλεσή τους κατά τη διάρκεια αυτού του βήματος. Όλες αυτές οι

διεργασίες εκτελούν τις ακολουθιακές προτάσεις τους, χρονοπρογραμματίζοντας ενδεχομένως περισσότερες συναλλαγές ή χρονικές υπερβάσεις, και τελικά αναστέλλονται πάλι με την εκτέλεση προτάσεων wait. Όταν όλες αναστείλουν την εκτέλεσή τους, ο κύκλος προσομοίωσης ολοκληρώνεται και ο επόμενος κύκλος μπορεί να αρχίσει. Εάν δεν υπάρχουν άλλες χρονοπρογραμματισμένες συναλλαγές ή χρονικές υπερβάσεις, ή εάν ο χρόνος προσομοίωσης φθάσει στο time'high (τη μεγαλύτερη χρονική στιγμή που μπορεί να αναπαρασταθεί), η προσομοίωση έχει ολοκληρωθεί.

Η περιγραφή της λειτουργίας ενός προσομοιωτή κατ' αυτό τον τρόπο μοιάζει λίγο με το στήσιμο μιας παράστασης σε ένα θέατρο χωρίς καθόλου καθίσματα — κανένας δεν βρίσκεται εκεί για να την παρακολουθήσει, επομένως δεν υπάρχει κανένα νόημα! Στην πραγματικότητα, ένας προσομοιωτής είναι μέρος μιας σουίτας εργαλείων VHDL και μας παρέχει διάφορα μέσα για να ελέγξουμε και να παρακολουθήσουμε την πρόοδο της προσομοίωσης. Οι τυπικοί προσομοιωτές μας επιτρέπουν να εκτελέσουμε βήμα-βήμα το μοντέλο μια γραμμή κάθε φορά ή να θέσουμε σημεία διακοπής (breakpoints), αναγκάζοντας την προσομοίωση να σταματήσει όταν εκτελεστεί μια γραμμή του μοντέλου ή όταν μια συγκεκριμένη τιμή ανατεθεί σε ένα σήμα. Παρέχουν συνήθως εντολές για να απεικονίσουν την τιμή των σημάτων ή των μεταβλητών. Πολλοί προσομοιωτές παρέχουν επίσης μια γραφική παράσταση κυματομορφής που απεικονίζει τις τιμές των σημάτων στο χρόνο παρόμοια με την απεικόνιση ενός λογικού αναλυτή (logic analyzer), και επιτρέπουν την αποθήκευση και στη συνέχεια την επανάκτηση του ιστορικού των τιμών για μεταγενέστερη ανάλυση. Αυτές οι ευκολίες καθιστούν χρήσιμη την προσομοίωση. Δυστυχώς, δεδομένου ότι υπάρχουν μεγάλες διαφορές μεταξύ των λειτουργιών που παρέχονται από τους διαφορετικούς προσομοιωτές, δεν είναι πρακτικό να μπούμε σε λεπτομέρειες. Οι κατασκευαστές προσομοιωτών παρέχουν συνήθως τεκμηρίωση εκμάθησης και σειρές εργαστηριακών σεμιναρίων που εξηγούν πώς να χρησιμοποιούμε τις λειτουργίες που παρέχονται από τα προϊόντα τους.