15.10 Εξηγήστε τις αρχές στις οποίες βασίζεται η Ενσωματωμένη Αυτο-Δοκιμή (BIST). Ποια είναι τα πλεονεκτήματα και τα μειονεκτήματά της;

15.11 Σας ζητείται να σχεδιάσετε έναν εξαιρετικά γρήγορο διαιρέτη συχνότητας διά οκτώ, ο οποίος φτάνει στα όρια των δυνατοτήτων της τεχνολογίας κατασκευής που χρησιμοποιείτε. Ποια στρατηγική δοκιμής θα χρησιμοποιήσετε για να ελέγξετε το διαιρέτη; Εξηγήστε τους λόγους της επιλογής σας.

15.12 Σχεδιάστε έναν καταχωρητή, ο οποίος θα ελαχιστοποιεί τον αριθμό των τρανζίστορ, αλλά θα επιτρέπει την υλοποίηση της παράλληλης ανίχνευσης όπως περιγράφεται στο Σχήμα 15.17.

15.13 Εξηγήστε πώς μπορεί να χρησιμοποιηθεί μια γεννήτρια ψευδοτυχαίων ακολουθιών (PRSG) για τη δοκιμή ενός χειριστή δεδομένων των 16 bit. Πώς θα μπορούσαν να συλλεχθούν και να ελεγχθούν οι έξοδοι;

15.14 Σχεδιάστε το διάγραμμα βαθμίδων για μια γεννήτρια δοκιμής η οποία πρόκειται να χρησιμοποιηθεί για τη δοκιμή μιας στατικής RAM 4Kx32.

15.15 Αναζητήστε την προέλευση του όρου «shmoo».

ΠΑΡΑΡΤΗΜΑ

Α

Γλώσσες Περιγραφής Hardware

Α.1 Εισαγωγή

Αυτό το παράρτημα αποτελεί μια σύντομη εισαγωγή στις SystemVerilog και VHDL, δύο Γλώσσες Περιγραφής Υλικού (Hardware Description Languages, HDL). Σε πολλά βιβλία, οι HDL αντιμετωπίζονται ως γλώσσες προγραμματισμού, αλλά στην πραγματικότητα μπορούν να γίνουν καλύτερα κατανοητές ως μια μορφή περιγραφής του hardware ψηφιακών συστημάτων. Ο καλύτερος τρόπος για να ξεκινήσετε ένα σχεδιαστικό έργο είναι καταστρώνοντας, στο μυαλό σας ή σε χαρτί, το hardware του ζητούμενου συστήματος. (Για παράδειγμα, ο επεξεργαστής MIPS περιλαμβάνει έναν ελεγκτή FSM κι ένα μονοπάτι δεδομένων αποτελούμενο από καταχωρητές, αθροιστές, πολυπλέκτες, κ.λπ.). Στη συνέχεια γράφετε τον HDL κώδικα που «περιγράφει» το hardware σ' ένα εργαλείο σύνθεσης. Ένα σύνηθες σφάλμα που κάνουν οι αρχάριοι είναι το ότι ξεκινούν να γράψουν ένα πρόγραμμα χωρίς να μελετήσουν το hardware στο οποίο αναφέρεται. Εάν δεν γνωρίζετε σε τι είδους hardware αναφέρεστε, είναι σχεδόν σίγουρο ότι θα οδηγηθείτε σε εσφαλμένο αποτέλεσμα. Σε ορισμένες περιπτώσεις αυτό μπορεί να εκδηλωθεί με την ύπαρξη επιπλέον μανδαλωτών σε μη αναμενόμενες θέσεις μέσα στο κύκλωμά σας. Σε άλλες περιπτώσεις μπορεί να οδηγήσει σ' ένα κύκλωμα πολύ αργότερο απ' ό,τι απαιτείται, ή ένα κύκλωμα το οποίο χρησιμοποιεί πολύ περισσότερες πύλες απ' όσες θα χρειαζόταν εάν το είχατε εξαρχής περιγράψει πιο προσεκτικά.

Σ' αυτό το παράρτημα ακολουθούμε μια μάλλον ασυνήθιστη προσέγγιση, καλύπτοντας παράλληλα τις SystemVerilog και VHDL. Αφιερώνουμε μια στήλη κειμένου για κάθε γλώσσα, πράγμα το οποίο διευκολύνει τη συγκριτική μελέτη τους. Την πρώτη φορά που θα διαβάσετε το παράρτημα, επικεντρωθείτε μόνο στη μία ή την άλλη γλώσσα. Αφού μάθετε τη μία, θα είστε σε θέση να μάθετε γρήγορα και την άλλη, εάν χρειαστεί. Όσον αφορά το ποια από τις δύο γλώσσες HDL είναι ανώτερη, οι διαμάχες μαίνονται ακόμα με θρησκευτικό ζήλο. Σύμφωνα με μια μεγάλη έρευνα που διεξήχθη το 2007 [Cooley07], το 73% των ερωτηθέντων χρησιμοποιούσαν πρωτίστως τις Verilog/SystemVerilog το 20% την VHDL, αλλά το 41% δήλωσε ότι ήταν υποχρεωμένο να χρησιμοποιεί και τις δύο λόγω παλαιότερου κώδικα που πέρασε στα χέρια τους, λόγω έτοιμων εξαρτημάτων που χρησιμοποιούσαν (intellectual property [IP] blocks), ή απλώς επειδή η Verilog υποστηρίζει καλύτερα τις netlist. Για το λόγο αυτό, πολλοί σχεδιαστές χρειάζεται να γνωρίζουν και τις δύο γλώσσες, ενώ τα περισσότερα εργαλεία CAD τις υποστηρίζουν επίσης.

Από την εμπειρία μας, ο καλύτερος τρόπος για να μάθει κανείς μια γλώσσα HDL είναι διά παραδείγματος. Οι γλώσσες HDL έχουν συγκεκριμένους τρόπους για την περιγραφή των διάφορων κατηγοριών λογικής· αυτοί αποκαλούνται *ιδιώματα* (idioms). Ο στόχος μας σ' αυτό το παράρτημα είναι να σας δείξουμε πώς να γράφετε το σωστό ιδίωμα HDL για κάθε τύπο δομικής μονάδας, καθώς και πώς να συνενώνετε όλες τις δομικές μονάδες μαζί για να παράγετε ένα λειτουργικό σύστημα. Θα επικεντρωθούμε σ' ένα *συνθέσιμο* (synthesizable) υποσύνολο της HDL, επαρκές για την περιγραφή οποιασδήποτε λειτουργίας σε επίπεδο hardware. Σε περιπτώσεις όπου απαιτείται η περιγραφή συγκεκριμένου hardware, αναζητήστε ένα παρόμοιο παράδειγμα και προσαρμόστε το κατάλληλα για τους σκοπούς σας. Οι δύο γλώσσες περιλαμβάνουν πολλές επιπλέον δυνατότητες, οι οποίες είναι κυρίως χρήσιμες για την ανάπτυξη περιβαλλόντων δοκιμής και, ως εκ τούτου, υπερβαίνουν τα όρια του παρόντος βιβλίου. Δεν σκοπεύουμε να περιγράψουμε ενδελεχώς το συντακτικό των γλωσσών, αφενός επειδή είναι κουραστικό και αφετέρου επειδή θεωρούμε ότι ενθαρρύνει την αντιμετώπισή τους ως γλώσσες προγραμματισμού και όχι ως εργαλεία για την περιγραφή του hardware. Θα πρέπει να είστε ιδιαίτερα προσεκτικοί όταν πειραματίζετε με άλλες δυνατότητες σε κώδικα ο οποίος πρόκειται να μεταφερθεί σε κάποιο εργαλείο σύνθεσης. Υπάρχουν πολλοί τρόποι για να γράψει κανείς HDL κώδικα του οποίου η συμπεριφορά στην προσομοίωση και στη σύνθεση θα διαφέρει, γεγονός το οποίο οδηγεί σε εσφαλμένη λειτουργία του ολοκληρωμένου ή στην ανάγκη διόρθωση σφαλμάτων αφού ολοκληρωθεί η σύνθεση. Το υποσύνολο της κάθε γλώσσας που θα καλύψουμε εδώ είναι ειδικά επιλεγμένο ώστε να ελαχιστοποιεί τέτοιου είδους προβλήματα.

Verilog και SystemVerilog

Η Verilog αναπτύχθηκε από την εταιρεία Gateway Design Automation σαν μια «κλειστή» γλώσσα προσομοίωσης λογικής, το 1984. Η Gateway εξαγοράστηκε από την Cadence το 1989 και η Verilog μετατράπηκε σε ανοιχτό πρότυπο το 1990, υπό τον έλεγχο της Open Verilog International. Η γλώσσα έγινε πρότυπο του IEEE το 1995 και ανανεώθηκε το 2001 [IEEE1364-01]. Το 2005 ανανεώθηκε ξανά, με μικρές αλλαγές· το σημαντικότερο ήταν η παρουσίαση της SystemVerilog [IEEE 1800-2009], η οποία απλοποιεί πολλές από τις δυσκολίες της Verilog και προσθέτει δυνατότητες παρόμοιες με αυτές των γλωσσών προγραμματισμού υψηλού επιπέδου, οι οποίες αποδείχτηκαν χρήσιμες για την επαλήθευση κυκλωμάτων.

Υπάρχουν πολλά βιβλία με θέμα την Verilog, αλλά το υλικό τεκμηρίωσης που παρέχει ο IEEE είναι και εκτενές και ευκολονόητο.

VHDL

Το όνομα VHDL είναι ακρωνύμιο του VHSIC Hardware Description Language. Με τη σειρά του, το VHSIC είναι το ακρωνύμιο του έργου Very High Speed Integrated Circuits (ολοκληρωμένα κυκλώματα υπερυψηλής ταχύτητας). Η VHDL πρωτοπαρουσιάστηκε το 1981 από το Υπουργείο Άμυνας, ως μια γλώσσα για την περιγραφή της δομής και της λειτουργίας ενός συστήματος σε επίπεδο hardware. Αντλεί την καταγωγή της από τη γλώσσα προγραμματισμού Ada. Ο οργανισμός IEEE προτυποποίησε την VHDL το 1987 και έκτοτε ενημερώνει το πρότυπο αρκετά συχνά [IEEE1076-08]. Αρχικά η γλώσσα προσανατολίζονταν στην τεκμηρίωση, αλλά σύντομα υιοθετήθηκε επίσης για την προσομοίωση και σύνθεση.

Η VHDL χρησιμοποιείται ευρύτατα από τις εταιρείες που συνεργάζονται με τις Ένοπλες Δυνάμεις των ΗΠΑ και με Ευρωπαϊκές εταιρείες. Από κάποια παραξενιά της τύχης, έχει επίσης την πλειοψηφία των χρηστών σε πανεπιστημιακά περιβάλλοντα.

Ο [Pedroni10] παρέχει εκτενή κάλυψη της γλώσσας.

A.1.1 Λειτουργικές Μονάδες (Modules)

Ως module χαρακτηρίζεται ένα συγκροτημένο σύνολο στοιχείων hardware, το οποίο διαθέτει εισόδους και εξόδους και αποτελεί μια αυτοτελή *λειτουργική μονάδα*. Μια πύλη AND, ένας πολυπλέκτης, ή ένα κύκλωμα προτεραιότητας είναι παραδείγματα λειτουργικών μονάδων hardware. Τα δύο γενικά στιλ περιγραφής της λειτουργικότητας μιας τέτοιας μονάδας είναι το *συμπεριφορικό* και το *δομικό*. Τα συμπεριφορικά μοντέλα περιγράφουν μια λειτουργική μονάδα σε επίπεδο συμπεριφοράς – το τι κάνει. Τα δομικά μοντέλα περιγράφουν πώς δομείται μια μονάδα από απλούστερα εξαρτήματα· είναι μια εφαρμογή ιεραρχίας. Ο κώδικας SystemVerilog και VHDL του Παραδείγματος Α.1 παρουσιάζει περιγραφές επιπέδου συμπεριφοράς μιας λειτουργικής μονάδας hardware που υπολογίζει μια τυχαία Boolean συνάρτηση, $Y = \overline{A}\overline{B}\overline{C} + A\overline{B}\overline{C}$. Η μονάδα έχει τρεις εισόδους, A, B και C και μία έξοδο, Y.

Παράδειγμα Α.1 Συνδυαστική Λογική

SystemVerilog

```
module sillyfunction(input  logic a, b, c,
                     output logic y);

  assign y = ~a & ~b & ~c |
             a & ~b & ~c |
             a & ~b &  c;
endmodule
```

Μια λειτουργική μονάδα (module) ξεκινά με μια λίστα που παραθέτει τις εισόδους και τις εξόδους. Η εντολή `assign` περιγράφει συνδυαστική λογική. Το σύμβολο ~ υποδεικνύει τη λογική πράξη NOT, το & την AND και το | την OR.

Τα λογικά (τύπου `logic`) σήματα, όπως οι είσοδοι και οι έξοδοι, είναι Boolean μεταβλητές (0 ή 1). Μπορούν επίσης να έχουν «αιωρούμενες» και απροσδιόριστες τιμές, όπως θα δούμε στην Ενότητα Α.2.8.

Ο τύπος `logic` περιλαμβάνεται μόνο στη SystemVerilog. Υπερτερεί του τύπου `reg`, ο οποίος υπήρξε πολυετής πηγή σύγχυσης στην Verilog. Ο τύπος `logic` θα πρέπει να χρησιμοποιείται οπουδήποτε, εκτός από περιπτώσεις δικτύων με πολλαπλούς οδηγούς, όπως θα εξηγήσουμε στην Ενότητα Α.7

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sillyfunction is
  port(a, b, c: in  STD_LOGIC;
       y:       out STD_LOGIC);
end;

architecture synth of sillyfunction is
begin
  y <= ((not a) and (not b) and (not c)) or
       (a and (not b) and (not c)) or
       (a and (not b) and c);
end;
```

Ο VHDL κώδικας χωρίζεται σε τρία μέρη: τη δήλωση βιβλιοθήκης (`library`), τη δήλωση οντότητας (`entity`) και τον κορμό αρχιτεκτονικής (`architecture`). Η `library` είναι υποχρεωτική και θα εξεταστεί στην Ενότητα Α.7. Η `entity` παραθέτει τις εισόδους και εξόδους της μονάδας. Ο κορμός αρχιτεκτονικής ορίζει το τι κάνει η λειτουργική μονάδα.

Στην VHDL, τα σήματα όπως οι είσοδοι και έξοδοι πρέπει να έχουν μια *δήλωση τύπου*. Τα ψηφιακά σήματα θα πρέπει να δηλώνονται ως τύπου STD_LOGIC. Τα σήματα STD_LOGIC μπορούν να έχουν τιμή '0' ή '1', καθώς και «αιωρούμενες»/απροσδιόριστες τιμές, όπως θα δούμε στην Ενότητα Α.2.8. Ο τύπος STD_LOGIC ορίζεται στη βιβλιοθήκη IEEE.STD_LOGIC_1164 και αυτός είναι ο λόγος για τον οποίο πρέπει να χρησιμοποιηθεί η δήλωση `library`.

Η VHDL δεν διαθέτει ένα καλά ορισμένο μηχανισμό για την προτεραιότητα εκτέλεσης των πράξεων, οπότε θα πρέπει να χρησιμοποιούνται παρενθέσεις στις Boolean εκφράσεις, για την επιβολή της επιθυμητής σειράς εκτέλεσης.

Η πραγματική ισχύς των γλωσσών HDL απορρέει από το υψηλότερο επίπεδο αφαίρεσης που παρέχουν, σε σύγκριση με τα σχηματικά. Για παράδειγμα, το σχηματικό ενός 32-bit αθροιστή είναι μια πολύπλοκη δομή. Ο σχεδιαστής πρέπει να επιλέξει ποιον τύπο αρχιτεκτονικής αθροιστή θα χρησιμοποιήσει. Ένας αθροιστής κύματος κρατουμένου (carry ripple) έχει 32 κύτταρα πλήρους αθροιστή, καθένα εκ των οποίων, με τη σειρά του, περιέχει μισή ντουζίνα πύλες ή αλλιώς μια πλειάδα τρανζίστορ. Ωστόσο, σε επίπεδο συμπεριφοράς, αυτός ο ίδιος αθροιστής μπορεί να καθοριστεί με μία μόνο γραμμή HDL κώδικα, όπως υποδεικνύει το Παράδειγμα Α.2.

Παράδειγμα Α.2 32-Bit Αθροιστής

SystemVerilog

```
module adder(input  logic [31:0] a,
             input  logic [31:0] b,
             output logic [31:0] y);

  assign y = a + b;
endmodule
```

Σημειώστε ότι οι είσοδοι και έξοδοι είναι 32-bit δίαυλοι.

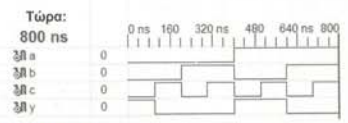
VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity adder is
  port(a, b: in  STD_LOGIC_VECTOR(31 downto 0);
       y:    out STD_LOGIC_VECTOR(31 downto 0));
end;

architecture synth of adder is
begin
  y <= a + b;
end;
```

Σημειώστε ότι οι είσοδοι και οι έξοδοι είναι 32-bit διανύσματα. Αυτά θα πρέπει να δηλώνονται ως STD_LOGIC_VECTOR.

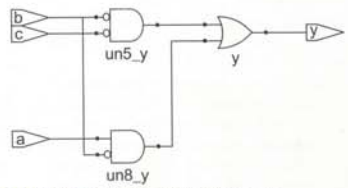
A.1.2 Προσομοίωση και Σύνθεση

Οι δύο κύριοι στόχοι που υπηρετούν οι γλώσσες HDL είναι η *προσομοίωση λογικής* και η *σύνθεση*. Κατά τη διάρκεια της προσομοίωσης, εφαρμόζονται είσοδοι σε μια μονάδα και ελέγχονται οι έξοδοί της για να επαληθευτεί ότι η μονάδα λειτουργεί σωστά. Κατά τη διάρκεια της σύνθεσης, η περιγραφή κειμένου μιας λειτουργικής μονάδας μετασχηματίζεται σε λογικές πύλες.



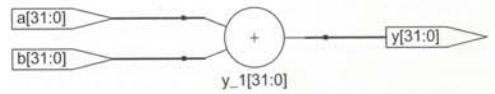
ΣΧΗΜΑ Α.1 Κυματομορφές που παρήγαγε η προσομοίωση.

A.1.2.1 Προσομοίωση. Το Σχήμα Α.1 παρουσιάζει τις κυματομορφές που έδωσε η προσομοίωση της παραπάνω λειτουργικής μονάδας `sillyfunction` με το Modelsim, οι οποίες αποδεικνύουν ότι η μονάδα δουλεύει σωστά. Η έξοδος Y είναι true όταν οι είσοδοι A, B και C είναι 000, 100, ή 101, όπως καθορίζει η Boolean έκφραση.



ΣΧΗΜΑ Α.2 Σύνθεση του κυκλώματος `silly_function`.

A.1.2.2 Σύνθεση. Η λογική σύνθεση μετασχηματίζει τον HDL κώδικα σε μια περιγραφή *netlist*, η οποία περιγράφει το hardware στο επίπεδο των λογικών πυλών και των αγωγών που τις συνδέουν. Το εργαλείο λογικής σύνθεσης μπορεί επίσης να εκτελεί βελτιστοποιήσεις με στόχο τη μείωση του απαιτούμενου ποσού hardware. Η netlist μπορεί να είναι ένα αρχείο κειμένου, ή μπορεί να εμφανίζεται σε μορφή σχηματικού, ώστε να βοηθά το χρήστη να κατανοήσει καλύτερα το κύκλωμα. Το Σχήμα Α.2 παρουσιάζει το αποτέλεσμα που έδωσε η σύνθεση της λειτουργικής μονάδας `sillyfunction` με το Synplify Pro. Παρατηρήστε ότι με τη βελτιστοποίηση που εφαρμόστηκε, οι τρεις πύλες AND 3 εισόδων έχουν μειωθεί σ' ένα ζεύγος πυλών AND 2 εισόδων. Παρόμοια, το Σχήμα Α.3 απεικονίζει ένα σχηματικό για τη λειτουργική μονάδα του αθροιστή. Από εδώ και στο εξής, κάθε παράδειγμα κώδικα θα ακολουθείται από το αντίστοιχο σχηματικό του.



ΣΧΗΜΑ Α.3 Σύνθεση του αθροιστή.

A.2 Συνδυαστική Λογική

Οι έξοδοι ενός κυκλώματος συνδυαστικής λογικής εξαρτώνται μόνο από τις τρέχουσες εισόδους· η συνδυαστική λογική δεν έχει μνήμη. Σ' αυτή την ενότητα θα μάθετε πώς να γράφετε μοντέλα συμπεριφοράς για κυκλώματα συνδυαστικής λογικής χρησιμοποιώντας γλώσσες HDL.

A.2.1 Bitwise Τελεστές

Οι *bitwise* (επιπέδου bit) τελεστές επενεργούν σε σήματα του ενός bit, ή σε διαύλους πολλαπλών bit. Η λειτουργική μονάδα inv στο Παράδειγμα A.3 περιγράφει τέσσερις αντιστροφείς συνδεδεμένους σε διαύλους των 4 bit.

Παράδειγμα A.3 Αντιστροφείς

SystemVerilog

```
module inv(input  logic [3:0] a,
           output logic [3:0] y);

  assign y = ~a;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity inv is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of inv is
begin
  y <= not a;
end;
```

ΣΧΗΜΑ A.4 Η μονάδα inv.

Η λειτουργική μονάδα gates στο Παράδειγμα A.4 επιδεικνύει τη χρήση bitwise τελεστών σε 4-bit διαύλους για διάφορες άλλες λογικές συναρτήσεις.

Παράδειγμα A.4 Λογικές Πύλες

SystemVerilog

```
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2,
                                y3, y4, y5);

  /* 5 διαφορετικές λογικές πύλες 2 εισόδων
     που επενεργούν σε διαύλους 4 bit */
  assign y1 = a & b;     // AND
  assign y2 = a | b;     // OR
  assign y3 = a ^ b;     // XOR
  assign y4 = ~(a & b); // NAND
  assign y5 = ~(a | b); // NOR
endmodule
```

Τα σύμβολα ~, ^ και | είναι παραδείγματα *τελεστών* της SystemVerilog, ενώ τα a, b και y1 είναι *τελεστέοι*. Οι συνδυασμοί τελεστών και τελεστέων, όπως π.χ. a & b, ή ~(a | b), σχηματίζουν *εκφράσεις*. Μια πλήρης «οδηγία», όπως η `assign y4 = ~(a & b);` αποκαλείται *εντολή* (statement).

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity gates is
  port(a, b:  in  STD_LOGIC_VECTOR(3 downto 0);
       y1, y2, y3, y4,
       y5: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of gates is
begin
  -- 5 διαφορετικές λογικές πύλες 2 εισόδων
  -- που επενεργούν σε διαύλους 4 bit
  y1 <= a and b;
  y2 <= a or b;
  y3 <= a xor b;
  y4 <= a nand b;
  y5 <= a nor b;
end;
```

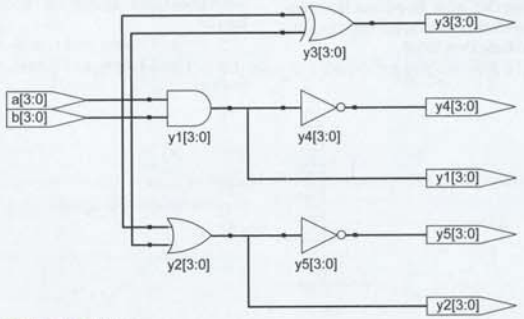

SystemVerilog (συνέχεια)

Η `assign out = in1 op in2;` αποκαλείται *εντολή συνεχούς ανάθεσης* (continuous assignment statement). Οι εντολές συνεχούς ανάθεσης τερματίζονται μ' ένα χαρακτήρα ελληνικού ερωτηματικού. Οποτεδήποτε αλλάζουν οι είσοδοι στη δεξιά πλευρά του = σε μια εντολή συνεχούς ανάθεσης, η έξοδος στην αριστερή πλευρά υπολογίζεται εκ νέου. Δηλαδή, οι εντολές συνεχούς ανάθεσης περιγράφουν συνδυαστική λογική.

VHDL (συνέχεια)

Οι not, xor και or είναι *τελεστές* της VHDL, ενώ τα a, b και y1 είναι *τελεστέοι*. Οι συνδυασμοί τελεστών με τελεστέους, όπως π.χ. a and b, ή a nor b, σχηματίζουν *εκφράσεις*. Μια πλήρης «οδηγία» –όπως η y4 <= a nand b;–, αποκαλείται *εντολή* (statement).

Η `out <= in1 op in2` είναι μια *εντολή ταυτόχρονης ανάθεσης σήματος* (concurrent signal assignment statement). Στην VHDL, οι εντολές ανάθεσης τερματίζονται μ' ένα χαρακτήρα ελληνικού ερωτηματικού. Οποτεδήποτε αλλάζουν οι είσοδοι στη δεξιά πλευρά του <= σε μια εντολή ταυτόχρονης ανάθεσης σήματος, η έξοδος στην αριστερή πλευρά υπολογίζεται εκ νέου. Δηλαδή, οι εντολές ταυτόχρονης ανάθεσης σήματος περιγράφουν συνδυαστική λογική.



ΣΧΗΜΑ A.5 Πύλες

A.2.2 Σχόλια και κενός χώρος

Το Παράδειγμα A.4 παρουσιάζει τη μορφοποίηση των σχολίων. Οι SystemVerilog και VHDL δεν είναι εκλεκτικές όσον αφορά στη χρήση κενού χώρου – δηλαδή, κενών διαστημάτων, στηλοθετών (tabs) και αλλαγών γραμμών. Σε κάθε περίπτωση όμως, η χρήση κατάλληλων επιπέδων εσοχών και κενών γραμμών είναι σημαντική για την αναγνωσιμότητα πολύπλοκων σχεδιάσεων. Θα πρέπει να υιοθετήσετε ένα συνεπές και ομοιόμορφο σχήμα όσον αφορά τη χρήση κεφαλαίων/πεζών και χαρακτήρων κάτω παύλας στα ονόματα των σημάτων και των λειτουργικών μονάδων.

SystemVerilog

Στην SystemVerilog, τα σχόλια είναι όμοια με αυτά των γλωσσών C και Java. Τα σχόλια που ξεκινούν με τους χαρακτήρες /* συνεχίζουν πιθανώς καταλαμβάνοντας πολλαπλές γραμμές, έως το επόμενο ζεύγος χαρακτήρων */. Τα σχόλια που ξεκινούν με το // συνεχίζουν έως το τέλος της ίδιας γραμμής. Η SystemVerilog κάνει διάκριση μεταξύ κεφαλαίων και πεζών χαρακτήρων. Τα y1 και Y1 θεωρούνται διαφορετικά σήματα στην SystemVerilog. Ωστόσο, η χρήση σημάτων που διαφέρουν μόνο ως προς την αναγραφή κεφαλαίων/πεζών προκαλεί σύγχυση και είναι μια επικίνδυνη πρακτική.

VHDL

Στην VHDL, τα σχόλια ξεκινούν με τους χαρακτήρες – και συνεχίζουν έως το τέλος της γραμμής. Τα σχόλια που εκτείνονται σε πολλαπλές γραμμές πρέπει να περιλαμβάνουν τους χαρακτήρες -- στην αρχή κάθε γραμμής. Η VHDL δεν κάνει διάκριση μεταξύ κεφαλαίων-πεζών. Τα y1 και Y1 είναι το ίδιο σήμα για την VHDL. Ωστόσο, είναι πιθανό να κάνουν διάκριση μεταξύ κεφαλαίων-πεζών άλλα εργαλεία στα οποία μπορεί να μεταφέρετε τα αρχεία σας, πράγμα το οποίο οδηγεί σε δυσεπίλυτα σφάλματα εάν αναμειγνύετε αδιακρίτως κεφαλαίους και πεζούς χαρακτήρες.

A.2.3 Τελεστές Μείωσης

Οι τελεστές μείωσης υποδηλώνουν την ύπαρξη μιας πύλης πολλαπλών εισόδων που επενεργεί σ' ένα μεμονωμένο δίαυλο. Το Παράδειγμα A.5 περιγράφει μια 8-εισόδων πύλη AND, με εισόδους a_0, a_1, ..., a_7.

Παράδειγμα Α.5 AND 8 εισόδων

SystemVerilog

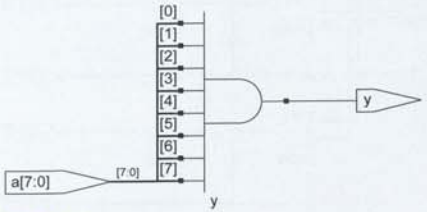
```
module and8(input  logic [7:0] a,
            output logic       y);

   assign y = &a;

   // το &a είναι πολύ ευκολότερο από το
   // assign y = a[7] & a[6] & a[5] & a[4] &
   //            a[3] & a[2] & a[1] & a[0];
endmodule
```

Όπως θα περίμενε κανείς, οι τελεστές μείωσης `|`, `^`, `~&` και `~|` είναι διαθέσιμοι και για τις πύλες OR, XOR, NAND και NOR. Θυμηθείτε ότι μια XOR πολλαπλών εισόδων εκτελεί έλεγχο ισοτιμίας, επιστρέφοντας TRUE εάν περιττός αριθμός είναι TRUE.

VHDL

Η VHDL δεν διαθέτει τελεστές μείωσης. Αντ' αυτών, παρέχει την εντολή `generate` (βλ. Ενότητα Α.8). Εναλλακτικά, η ζητούμενη λειτουργία μπορεί να διατυπώνεται ρητά:

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity and8 is
  port(a: in  STD_LOGIC_VECTOR(7 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of and8 is
begin
  y <= a(7) and a(6) and a(5) and a(4) and
       a(3) and a(2) and a(1) and a(0);
end;
```



ΣΧΗΜΑ Α.6 AND 8 εισόδων.

Α.2.4 Ανάθεση Υπό Συνθήκη

Οι εντολές *ανάθεσης υπό συνθήκη* επιλέγουν την έξοδο από ένα σύνολο εναλλακτικών επιλογών βασιζόμενες σε μια είσοδο που αποκαλείται *συνθήκη*. Το Παράδειγμα Α.6 παρουσιάζει έναν πολυπλέκτη 2:1 που χρησιμοποιεί ανάθεση υπό συνθήκη.

Παράδειγμα Α.6 Πολυπλέκτης 2:1

SystemVerilog

Ο *υπό συνθήκη τελεστής* ?: επιλέγει μεταξύ δύο εκφράσεων, βασιζόμενος σε μία άλλη έκφραση. Η πρώτη έκφραση είναι η *συνθήκη*. Εάν η συνθήκη είναι 1, ο τελεστής επιλέγει τη δεύτερη έκφραση. Εάν η συνθήκη είναι 0, ο τελεστής επιλέγει την τρίτη έκφραση.

Ο τελεστής ?: είναι ιδιαίτερα χρήσιμος για την περιγραφή ενός πολυπλέκτη επειδή, ανάλογα με την πρώτη είσοδο που λαμβάνει κάνει μία επιλογή μεταξύ των δύο άλλων. Ο ακόλουθος κώδικας επιδεικνύει τη σύνταξη (ιδίωμα) για έναν πολυπλέκτη 2:1 με εισόδους και εξόδους των 4 bit, χρησιμοποιώντας τον υπό συνθήκη τελεστή.

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output logic [3:0] y);

   assign y = s ? d1 : d0;
endmodule
```

Εάν `s = 1`, τότε `y = d1`. Εάν `s = 0`, τότε `y = d0`.

VHDL

Οι εντολές *υπό συνθήκη ανάθεσης σήματος* εκτελούν διαφορετικές λειτουργίες, με βάση κάποια συνθήκη. Είναι ιδιαίτερα χρήσιμες για την περιγραφή ενός πολυπλέκτη. Για παράδειγμα, ένας πολυπλέκτης 2:1 μπορεί να χρησιμοποιεί υπό συνθήκη ανάθεση σήματος για να επιλέγει μία από δύο εισόδους των 4 bit.

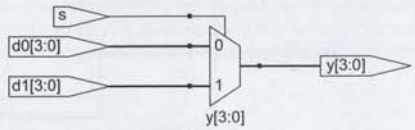
```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  port(d0, d1:in  STD_LOGIC_VECTOR(3 downto 0);
       s:     in  STD_LOGIC;
       y:     out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of mux2 is
begin
  y <= d0 when s = '0' else d1;
end;
```


SystemVerilog (συνέχεια)

Ο τελεστής ?: αποκαλείται επίσης *τριαδικός* (ternary) επειδή δέχεται τρεις εισόδους. Χρησιμοποιείται για τον ίδιο σκοπό, όπως και στις γλώσσες προγραμματισμού C & Java.

VHDL (συνέχεια)

Η υπό συνθήκη ανάθεση σήματος θέτει την έξοδο `y` σε `d0` εάν το `s` είναι 0. Διαφορετικά θέτει την `y` σε `d1`.



ΣΧΗΜΑ Α.7 Η μονάδα mux.

Το Παράδειγμα Α.7 παρουσιάζει έναν πολυπλέκτη 4:1, ο οποίος βασίζεται στην ίδια αρχή.

Παράδειγμα Α.7 Πολυπλέκτης 4:1

SystemVerilog

Ένας πολυπλέκτης 4:1 μπορεί να επιλέγει μία από τέσσερις εισόδους, χρησιμοποιώντας περισσότερους του ενός υπό συνθήκη τελεστές, ένθετους τον ένα μέσα στον άλλο.

```
module mux4(input  logic [3:0] d0, d1, d2, d3,
            input  logic [1:0] s,
            output logic [3:0] y);

   assign y = s[1] ? (s[0] ? d3 : d2)
                   : (s[0] ? d1 : d0);
endmodule
```

Εάν `s[1] = 1`, τότε ο πολυπλέκτης επιλέγει την πρώτη έκφραση, `(s[0] ? d3 : d2)`. Με τη σειρά της, αυτή η έκφραση επιλέγει είτε το `d3` είτε το `d2`, ανάλογα με το `s[0]` (`y = d3` εάν `s[0] = 1` και `d2` εάν `s[0] = 0`). Εάν `s[1] = 0`, τότε ο πολυπλέκτης επιλέγει, κατά παρόμοιο τρόπο, τη δεύτερη έκφραση, η οποία δίνει είτε `d1` είτε `d0` αναλόγως του `s[0]`.

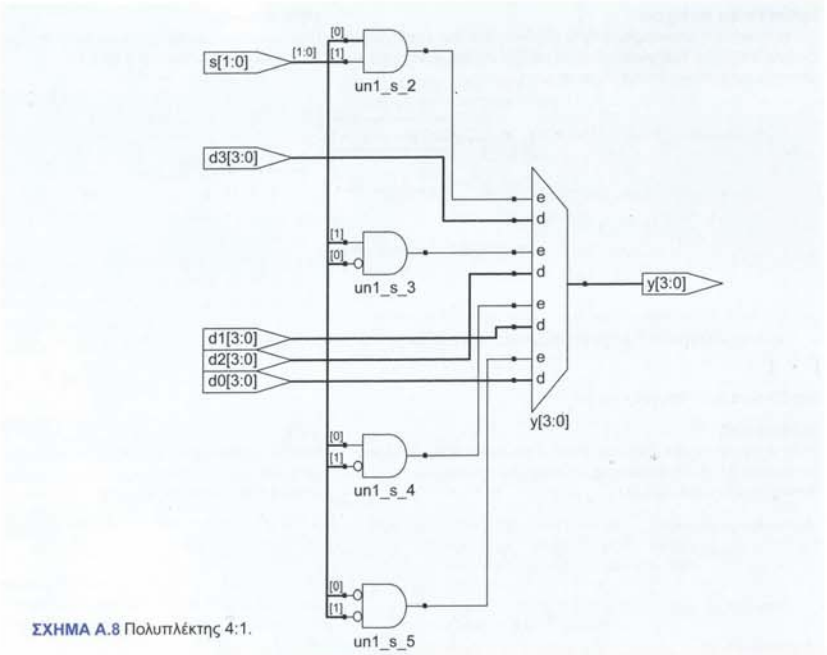
VHDL

Ένας πολυπλέκτης 4:1 μπορεί να επιλέγει μία από τέσσερις εισόδους, χρησιμοποιώντας πολλαπλούς όρους (clauses) `else` στην εντολή υπό συνθήκη ανάθεσης σήματος.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux4 is
  port(d0, d1,
    d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
    s:      in  STD_LOGIC_VECTOR(1 downto 0);
    y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth1 of mux4 is
begin
  y <= d0 when s = "00" else
       d1 when s = "01" else
       d2 when s = "10" else
       d3;
end;
```

Η VHDL υποστηρίζει επίσης εντολές *ανάθεσης επιλεγμένου σήματος* ως ένα μηχανισμό συντόμευσης για τις περιπτώσεις όπου απαιτείται η επιλογή μιας από πολλαπλές πιθανές επιλογές. Είναι ανάλογες με τη χρήση μιας εντολής `case` αντί πολλαπλών εντολών `if/else` στις περισσότερες γλώσσες προγραμματισμού. Χρησιμοποιώντας ανάθεση επιλεγμένου σήματος, ο πολυπλέκτης 4:1 μπορεί να αναδιατυπωθεί ως εξής:

```
architecture synth2 of mux4 is
begin
  with s select y <=
    d0 when "00",
    d1 when "01",
    d2 when "10",
    d3 when others;
end;
```



ΣΧΗΜΑ Α.8 Πολυπλέκτης 4:1.

Το Σχήμα Α.8 παρουσιάζει το σχηματικό για τον πολυπλέκτη 4:1 που παρήγαγε το Synplify Pro. Η συγκεκριμένη εφαρμογή χρησιμοποιεί διαφορετικό σύμβολο για τον πολυπλέκτη από αυτό που χρησιμοποιήσαμε έως τώρα στο παρόν βιβλίο. Ο πολυπλέκτης έχει πολλαπλές εισόδους δεδομένων (d) και μία είσοδο enable (e), τύπου "one-hot" (επισημαίνεται η θέση του πιο σημαντικού 1, ενώ τα υπόλοιπα ψηφία μηδενίζονται). Όταν δίνεται ένα από τα σήματα enable, τα σχετικά μ' αυτό δεδομένα περνιούνται στην έξοδο. Για παράδειγμα, όταν `s[1] = s[0] = 0`, η κάτω πύλη AND, `un1_s_5`, παράγει ένα 1, ενεργοποιώντας την κάτω είσοδο του πολυπλέκτη και αναγκάζοντάς την να επιλέξει την `d0[3:0]`.

A.2.5 Εσωτερικές Μεταβλητές

Συχνά, είναι βολικό οι πολύπλοκες συναρτήσεις να διαχωρίζονται σε μικρότερα, διαδοχικά βήματα. Για παράδειγμα, ένας πλήρη αθροιστή όπως αυτός που περιγράφεται στην Ενότητα 11.2.1, είναι ένα κύκλωμα με τρεις εισόδους και δύο εξόδους, οι οποίες ορίζονται από τις σχέσεις

$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in} \quad \textbf{(A.1)}$$

Εάν ορίσουμε τα ενδιάμεσα σήματα P και G

$$P = A \oplus B$$
$$G = AB \quad \textbf{(A.2)}$$

μπορούμε να αναδιατυπώσουμε τον πλήρη αθροιστή ως εξής:

$$S = P \oplus C_{in}$$
$$C_{out} = G + PC_{in} \quad \textbf{(A.3)}$$

Τα P και G αποκαλούνται *εσωτερικές μεταβλητές* (internal variables) επειδή δεν είναι ούτε είσοδοι ούτε έξοδοι – χρησιμοποιούνται μόνο εσωτερικά στη λειτουργική μονάδα. Είναι παρόμοιες με τις τοπικές μεταβλητές στις γλώσσες προγραμματισμού. Το Παράδειγμα Α.8 δείχνει πώς χρησιμοποιούνται σε γλώσσες HDL.

Παράδειγμα Α.8 Πλήρης αθροιστής

SystemVerilog

Στην SystemVerilog, τα «εσωτερικής χρήσης» σήματα δηλώνονται συνήθως ως `logic`.

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);

  logic p, g;

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```

VHDL

Στην VHDL, χρησιμοποιούνται *σήματα* για την αναπαράσταση εσωτερικών μεταβλητών, των οποίων οι τιμές ορίζονται από *εντολές ταυτόχρονης ανάθεσης σήματος*, όπως η `p <= a xor b`.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

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

architecture synth of fulladder is
  signal p, g: STD_LOGIC;
begin
  p <= a xor b;
  g <= a and b;

  s <= p xor cin;
  cout <= g or (p and cin);
end;
```

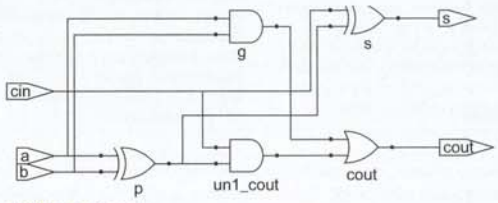


FIGURE A.9 fulladder

ΣΧΗΜΑ Α.9 Πλήρης αθροιστής.

Στις γλώσσες HDL, οι εντολές ανάθεσης (`assign` στην SystemVerilog και `<=` στην VHDL) λαμβάνουν χώρα ταυτόχρονα. Αυτό διαφέρει από τις συμβατικές γλώσσες προγραμματισμού όπως οι C και Java, στις οποίες οι εντολές αποτιμώνται με τη σειρά με την οποία γράφονται. Σε μια συμβατική γλώσσα, είναι σημαντικό το $S = P \oplus C_{in}$ να έπεται του $P = A \oplus B$, επειδή οι εντολές εκτελούνται ακολουθιακά. Σε μια γλώσσα HDL, η σειρά αποτίμησης είναι άνευ σημασίας· οι εντολές ανάθεσης αποτιμώνται οποτεδήποτε τα σήματα στη δεξιά πλευρά της εντολής αλλάζουν την τιμή τους, ανεξάρτητα από τη σειρά με την οποία εμφανίζονται σε μια λειτουργική μονάδα.

A.2.6 Προτεραιότητα και Άλλοι Τελεστές

Παρατηρήστε ότι περικλείουμε σε παρενθέσεις τον υπολογισμό cout, για να καθορίσουμε την σειρά εκτέλεσης των πράξεων ως $C_{out} = G + (P \cdot C_{in})$, και όχι ως $C_{out} = (G + P) \cdot C_{in}$. Εάν δεν είχαμε χρησιμοποιήσει παρενθέσεις, θα ίσχυε η προκαθορισμένη από τη γλώσσα σειρά εκτέλεσης των πράξεων. Το Παράδειγμα A.9 επιδεικνύει την προτεραιότητα των τελεστών, από το υψηλότερο έως το χαμηλότερο επίπεδο, για κάθε γλώσσα.

Παράδειγμα A.9 Προτεραιότητα Τελεστών

SystemVerilog

ΠΙΝΑΚΑΣ A.1 Προτεραιότητα τελεστών στην SystemVerilog

	Op	Meaning
Υψηλότερη	~	NOT
	*, /, %	MUL, DIV, MOD
	+, -	PLUS, MINUS
	<<, >>	Λογική ολίσθηση αριστερά/δεξιά
	<<<, >>>	Αριθμητική ολίσθηση αριστερά/δεξιά
	<, <=, >, >=	Σχετική σύγκριση
	==, !=	Σύγκριση ισότητας
	&, ~&	AND, NAND
	^, ~^	XOR, XNOR
	|, ~|	OR, NOR
Χαμηλότερη	?:	Εκτέλεση υπό συνθήκη

Στην SystemVerilog, η προτεραιότητα τελεστών είναι παρόμοια με αυτή που χρησιμοποιείται σε άλλες γλώσσες προγραμματισμού. Συγκεκριμένα, όπως υποδεικνύει ο Πίνακας A.1, η AND έχει προτεραιότητα έναντι της OR. Εκμεταλλευόμενοι αυτή την σειρά προτεραιότητας, μπορούμε να εξαλείψουμε τις παρενθέσεις στο ακόλουθο παράδειγμα.

```
assign cout = g | p & cin;
```

VHDL

ΠΙΝΑΚΑΣ A.2 Προτεραιότητα τελεστών στην VHDL

	Op	Meaning
Υψηλότερη	not	NOT
	*, /, mod, rem	MUL, DIV, MOD, REM
	+, -, &	PLUS, MINUS, CONCATENATE
	rol, ror, srl, sll, sra, sla	Περιστροφή Λογική ολίσθηση Αριθμητική ολίσθηση
	=, /=, <, <=, >, >=	Σύγκριση
Χαμηλότερη	and, or, nand, nor, xor	Λογικές πράξεις

Όπως υποδεικνύει ο Πίνακας A.2, στην VHDL ο πολλαπλασιασμός έχει προτεραιότητα έναντι της πρόσθεσης, όπως είναι αναμενόμενο. Ωστόσο, όλες οι λογικές λειτουργίες (and, or, κ.α.) έχουν ίδια προτεραιότητα, αντίθετα με ό,τι θα περίμενε κανείς από την άλγεβρα Boole. Συνεπώς, στο παράδειγμά μας οι παρενθέσεις είναι αναγκαίες· διαφορετικά, η έκφραση `cout <= g or p and cin` θα διερμηνεύονταν με σειρά από τα αριστερά προς τα δεξιά, ως `cout <= (g or p) and cin`.

Οι παραπάνω πίνακες προτεραιότητας περιλαμβάνουν και άλλους τελεστές – αριθμητικούς, ολίσθησης (μετατόπισης) και σύγκρισης. Για υλοποιήσεις αυτών των λειτουργιών σε επίπεδο hardware, ανατρέξτε στο Κεφάλαιο 11. Η αφαίρεση απαιτεί υπολογισμό συμπληρώματος ως προς δύο και πρόσθεση. Οι πολλαπλασιαστές και οι ολισθητές καταναλώνουν σημαντικά περισσότερη επιφάνεια (εκτός κι αν στις πράξεις εμπλέκονται εύκολες σταθερές). Η υλοποίηση των πράξεων διαίρεσης και υπολοίπου διαίρεσης (modulus) σε hardware έχει τόσο μεγάλο κόστος, που μπορεί να μην είναι εφικτή με τη χρήση εργαλείων σύνθεσης. Οι συγκρίσεις ισότητας προϋποθέτουν N πύλες XOR 2 εισόδων για τον καθορισμό της ισότητας κάθε bit και μια πύλη AND N εισόδων για το συνδυασμό όλων των bit. Η σχετική σύγκριση απαιτεί μια πράξη αφαίρεσης.

A.2.7 Αριθμοί

Οι αριθμοί μπορούν να καθορίζονται σε διάφορα αριθμητικά συστήματα. Οι χαρακτήρες κάτω παύλας στους αριθμούς αγνοούνται και μπορούν να χρησιμοποιούνται βοηθητικά, για το διαχωρισμό μεγάλου μήκους αριθμών σε μικρότερα, πιο ευανάγνωστα τμήματα. Το Παράδειγμα A.10 εξηγεί πώς γράφονται οι αριθμοί σε κάθε γλώσσα.

Παράδειγμα A.10 Αριθμοί

SystemVerilog

Όπως υποδεικνύει ο Πίνακας A.3, στην SystemVerilog οι αριθμοί μπορούν να καθορίζονται με τη βάση και το μέγεθός τους (το πλήθος των bit που χρησιμοποιείται για την αναπαράστασή τους). Η μορφή για τη δήλωση σταθερών είναι `N'Bvalue`, όπου το `N` είναι το μέγεθος σε bit, το `B` είναι η βάση και το `value` δίνει την τιμή. Για παράδειγμα, το `9'h25` υποδεικνύει έναν αριθμό των 9 bit με τιμή 2516 = 3710 = 0001001012. Η SystemVerilog υποστηρίζει τους συμβολισμούς 'b για το δυαδικό (βάση 2), 'o για το οκταδικό (βάση 8), 'd για το δεκαδικό (βάση 10) και 'h για το δεκαεξαδικό (βάση 16). Εάν παραληφθεί η βάση, χρησιμοποιείται η προεπιλεγμένη ρύθμιση βάσης που αντιστοιχεί στο δεκαδικό.

Εάν δεν δίνεται το μέγεθος, ο αριθμός υποτίθεται ότι έχει όσα bit υπάρχουν στην έκφραση στην οποία χρησιμοποιείται. Προστίθενται αυτόματα μηδενικά στην αρχή του αριθμού για να διαμορφωθεί στο πλήρες μέγεθός του. Για παράδειγμα, εάν το w αντιστοιχεί σ' ένα δίαυλο των 6 bit, η εντολή `assign w = 'b11` δίνει στο w την τιμή 000011. Η καλύτερη πρακτική είναι να δίνεται ρητά το μέγεθος των αριθμών. Μια εξαίρεση σ' αυτό τον κανόνα είναι οι συντομεύσεις '0 και '1 που παρέχει η SystemVerilog για το γέμισμα ενός διαύλου με όλα τα ψηφία 0 ή 1, αντίστοιχα.

ΠΙΝΑΚΑΣ A.3 Χρήση αριθμών στην SystemVerilog

Αριθμοί	Bits	Βάση	Τιμή που αποθηκεύεται	
3'b101	3	2	5	101
'b11	?	2	3	000...0011
8'b11	8	2	3	00000011
8'b1010_1011	8	2	171	10101011
3'd6	3	10	6	110
6'o42	6	8	34	100010
8'hAB	8	16	171	10101011
42	?	10	42	00...0101010
'1	?	n/a		11...111

VHDL

Στην VHDL, οι αριθμοί τύπου STD_LOGIC γράφονται στο δυαδικό και περικλείονται σε αποστρόφους. Τα '0' και '1' υποδεικνύουν το λογικό 0 και το λογικό 1, αντίστοιχα.

Οι αριθμοί τύπου STD_LOGIC_VECTOR γράφονται στο δυαδικό ή στο δεκαεξαδικό και περικλείονται σε εισαγωγικά. Η βάση είναι εξ ορισμού στο δυαδικό και μπορεί να ορίζεται ρητά με τη χρήση του προθέματος X για το δεκαεξαδικό ή B για το δυαδικό, όπως υποδεικνύει ο Πίνακας A.4.

ΠΙΝΑΚΑΣ A.4 Χρήση αριθμών στην VHDL

Αριθμοί	Bits	Βάση	Τιμή που αποθηκεύεται	
"101"	3	2	5	101
B"101"	3	2	5	101
X"AB"	8	16	161	10101011

A.2.8 Τα ειδικά σύμβολα Z και X

Οι γλώσσες HDL χρησιμοποιούν το σύμβολο z για να υποδείξουν μια «αιωρούμενη» (floating) τιμή. Το z είναι ιδιαίτερα χρήσιμο για την περιγραφή ενός τρισταθούς απομονωτή, του οποίου η έξοδος «αιωρείται» όταν το σήμα enable (ενεργοποίησης/επίτρεψης) είναι 0. Ένας δίαυλος μπορεί να οδηγείται από περισσότερους του ενός τρισταθείς απομονωτές, εκ των οποίων μόνο ένας θα πρέπει να είναι ενεργοποιημένος. Το Παράδειγμα A.11 παρουσιάζει τη σύνταξη για έναν τρισταθή απομονωτή. Εάν ο απομονωτής είναι ενεργοποιημένος, η έξοδος είναι ίδια με την είσοδο. Εάν είναι απενεργοποιημένος, η έξοδος λαμβάνει μια «αιωρούμενη» τιμή (z).

Παράδειγμα A.11 Τρισταθής Απομονωτής

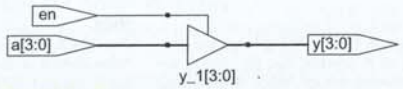
SystemVerilog

```
module tristate(input  logic [3:0] a,
                input  logic       en,
                output tri   [3:0] y);

  assign y = en ? a : 4'bz;
endmodule
```

Παρατηρήστε ότι το `y` δηλώνεται ως `tri` και όχι ως `logic`. Τα σήματα `logic` μπορούν να έχουν μόνο έναν οδηγό (driver). Οι τρισταθείς δίαυλοι μπορούν να έχουν πολλαπλούς οδηγούς και γι' αυτό θα πρέπει να δηλώνονται ως *net* (δίκτυο). Οι δύο τύποι δικτύων στην SystemVerilog αποκαλούνται `tri` και `trireg`. Τυπικά, μόνο ένας οδηγός σ' ένα δίκτυο είναι ενεργός οποτεδήποτε και το δίκτυο λαμβάνει αυτή την τιμή. Εάν δεν είναι ενεργός κανένας οδηγός, ένας κόμβος `tri` «αιωρείται» (`z`), ενώ ένας κόμβος `trireg` διατηρεί την προηγούμενη τιμή του. Εάν δεν καθορίζεται τύπος για μια είσοδο ή έξοδο, υποτίθεται ότι είναι τύπου `tri`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity tristate is
  port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
       en: in  STD_LOGIC;
       y:  out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of tristate is
begin
  y <= "ZZZZ" when en = '0' else a;
end;
```



ΣΧΗΜΑ Α.10 Τρισταθές στοιχείο.

Παρόμοια, οι γλώσσες HDL χρησιμοποιούν το σύμβολο x για να υποδείξουν μια άκυρη λογική στάθμη. Εάν ένας δίαυλος οδηγείται ταυτόχρονα σε 0 και 1 από δύο ενεργοποιημένους τρισταθείς απομονωτές (ή άλλες πύλες), το αποτέλεσμα είναι x και υποδεικνύει μια κατάσταση διαμάχης. Εάν όλοι οι τρισταθείς απομονωτές που οδηγούν ένα δίαυλο είναι ταυτόχρονα OFF, ο δίαυλος θα «αιωρείται» και αυτό υποδεικνύεται με το σύμβολο z.

Κατά την έναρξη της προσομοίωσης, οι κόμβοι κατάστασης, όπως π.χ. οι έξοδοι των flip-flop, αρχικοποιούνται σε μια άγνωστη κατάσταση (x στην SystemVerilog και u στην VHDL). Αυτό είναι ένα χρήσιμο βοήθημα για τον εντοπισμό των σφαλμάτων που οφείλονται στο ότι δεν έχει προβλεφθεί επαναφορά (reset) ενός flip-flop πριν χρησιμοποιηθεί η έξοδός του.

Εάν μια πύλη λάβει μια αιωρούμενη είσοδο, μπορεί παράγει μια έξοδο x στις περιπτώσεις που δεν μπορεί να καθορίσει τη σωστή τιμή για την έξοδο. Παρόμοια, εάν λάβει μια μη-έγκυρη ή μη-αρχικοποιημένη είσοδο, μπορεί να παράγει μια έξοδο x. Το Παράδειγμα Α.12 δείχνει πώς οι SystemVerilog και VHDL συνδυάζουν αυτές τις διαφορετικές τιμές σημάτων σε λογικές πύλες.

Παράδειγμα Α.12 Πίνακες Αληθείας με Απροσδιόριστες και Αιωρούμενες Εισόδους

SystemVerilog

Στην SystemVerilog, οι τιμές σήματος είναι 0, 1, z και x. Οι σταθερές που ξεκινούν με το z ή το x γεμίζονται με σύμβολα z ή x στην αρχή τους (αντί για 0) ώστε να φτάσουν στο πλήρες μήκος τους, όταν αυτό είναι αναγκαίο. Ο Πίνακας Α.5 παρουσιάζει έναν πίνακα αληθείας για μια πύλη AND που χρησιμοποιεί και τις τέσσερις πιθανές τιμές σήματος. Σημειώστε ότι, σε ορισμένες περιπτώσεις η πύλη μπορεί να καθορίσει την έξοδο, παρά το γεγονός ότι κάποιες είσοδοι είναι άγνωστες. Για παράδειγμα το 0 & z επιστρέφει 0, επειδή η έξοδος μιας πύλης AND είναι πάντα 0 εάν οποιαδήποτε από τις δύο εισόδους είναι 0. Διαφορετικά, οι αιωρούμενες ή άκυρες είσοδοι προκαλούν άκυρες εξόδους, οι οποίες υποδεικνύονται ως x.

ΠΙΝΑΚΑΣ Α.5 Πίνακας αληθείας για την πύλη AND με εισόδους z και x

&		A			
		0	1	z	x
B	0	0	0	0	0
	1	0	1	x	x
	z	0	x	x	x
	x	0	x	x	x

VHDL

Στην VHDL, τα σήματα τύπου STD_LOGIC μπορούν να είναι '0', '1', 'z', 'x' και 'u'.

Ο Πίνακας Α.6 παρουσιάζει τον πίνακα αληθείας για μια πύλη AND που χρησιμοποιεί και τις πέντε πιθανές τιμές σήματος. Σημειώστε ότι η πύλη μπορεί, σε ορισμένες περιπτώσεις, να καθορίσει την έξοδο παρά το γεγονός ότι οι είσοδοι είναι άγνωστες. Για παράδειγμα, τα '0' και 'z' επιστρέφουν '0' επειδή η έξοδος μιας πύλης AND είναι πάντα '0' εάν οποιαδήποτε από τις εισόδους είναι '0'. Διαφορετικά, οι αιωρούμενες ή άκυρες είσοδοι προκαλούν άκυρες εξόδους, οι οποίες υποδεικνύονται ως 'x' στην VHDL. Οι μη-αρχικοποιημένες είσοδοι παράγουν μη-αρχικοποιημένες εξόδους, οι οποίες υποδεικνύονται ως 'u' στην VHDL.

ΠΙΝΑΚΑΣ Α.6 Πίνακας αληθείας για την πύλη AND, με εισόδους z, x και u

AND		A				
		0	1	z	x	u
B	0	0	0	0	0	0
	1	0	1	x	x	u
	z	0	x	x	x	u
	x	0	x	x	x	u
	u	0	u	u	u	u

Η εμφάνιση τιμών x ή u σε μια προσομοίωση αποτελεί σχεδόν πάντα ένδειξη κάποιου σφάλματος ή εσφαλμένης συγγραφής του κώδικα. Στο συντιθέμενο κύκλωμα, αυτό αντιστοιχεί σε μια αιωρούμενη είσοδο πύλης ή σε μη-αρχικοποιημένη κατάσταση. Οι είσοδοι x ή u μπορεί να διερμηνευτούν αυθαίρετα από το κύκλωμα ως 0 ή 1, πράγμα το οποίο οδηγεί σε απρόβλεπτη συμπεριφορά.

Α.2.9 Ανάδευση Bit

Συχνά, είναι αναγκαίο να εκτελεστεί μια λειτουργία σ' ένα υποσύνολο ενός διαύλου, ή να συνενωθούν σήματα για το σχηματισμό διαύλων. Αυτές οι λειτουργίες αποκαλούνται «ανάδευση bit» (bit swizzling). Στο Παράδειγμα Α.13, το y λαμβάνει την 9-bit τιμή $c_2c_1d_0d_0d_0c_0101$ χρησιμοποιώντας λειτουργίες bit swizzling.

Παράδειγμα Α.13 Ανάδευση Bit

SystemVerilog

```
assign y = {c[2:1], {3{d[0]}}, c[0], 3'b101};
```

Ο τελεστής { } χρησιμοποιείται για τη συνένωση διαύλων. Το {3{d[0]}} υποδεικνύει τρία αντίγραφα του d[0].

Δεν θα πρέπει να συγχέετε την 3-bit δυαδική σταθερά 3'b101 με το δίαυλο b. Σημειώστε ότι ήταν σημαντικό να καθοριστεί το μήκος των 3 bit στη σταθερά· διαφορετικά, θα είχε άγνωστο αριθμό αρχικών μηδενικών, τα οποία ενδεχομένως να εμφανίζονταν στο μέσον της y.

Εάν το y είχε εύρος μεγαλύτερο από 9 bit, θα τοποθετούνταν μηδενικά στη θέση των περισσότερο σημαντικών bit.

VHDL

```
y <= c(2 downto 1) & d(0) & d(0) & d(0) &
     c(0) & "101";
```

Ο τελεστής & χρησιμοποιείται για τη *συνένωση* (concatenation) διαύλων. Το y πρέπει να είναι τύπου STD_LOGIC_VECTOR και μεγέθους 9 bit. Δεν θα πρέπει να συγχέετε τον τελεστή & με τον τελεστή and της VHDL.

Το Παράδειγμα Α.14 δείχνει πώς μπορεί μία έξοδος να διαχωριστεί σε δύο μέρη χρησιμοποιώντας λειτουργίες bit swizzling. Το Παράδειγμα Α.15 δείχνει πώς εκτελείται η επέκταση πρόσημου ενός 16-bit αριθμού στα 32 bits, αντιγράφοντας το περισσότερο σημαντικό στις 16 ανώτερες θέσεις.

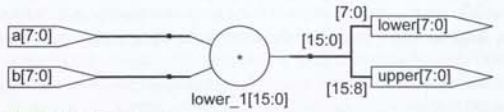
Παράδειγμα Α.14 Διαχωρισμός της Εξόδου

SystemVerilog

```
module mul(input  logic [7:0] a, b,
           output logic [7:0] upper, lower);

  assign {upper, lower} = a*b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity mul is
  port(a, b: in  STD_LOGIC_VECTOR(7 downto 0);
       upper, lower:
            out STD_LOGIC_VECTOR(7 downto 0));
end;
architecture behave of mul is
  signal prod: STD_LOGIC_VECTOR(15 downto 0);
begin
  prod <= a * b;
  upper <= prod(15 downto 8);
  lower <= prod(7 downto 0);
end;
```



ΣΧΗΜΑ Α.11 Πολλαπλασιαστές

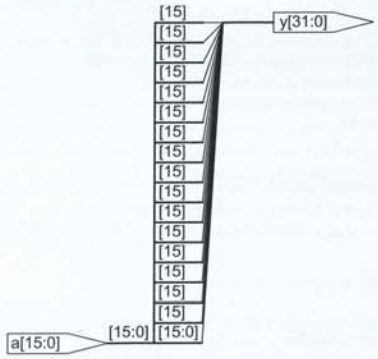
Παράδειγμα Α.15 Επέκταση πρόσημου

SystemVerilog

```
module signextend(input  logic [15:0] a,
                  output logic [31:0] y);

   assign y = {{16{a[15]}}, a[15:0]};
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity signext is -- sign extender
  port(a: in  STD_LOGIC_VECTOR (15 downto 0);
       y: out STD_LOGIC_VECTOR (31 downto 0));
end;
architecture behave of signext is
begin
  y <= X"0000" & a when a (15) = '0' else X"ffff" & a;
end;
```



ΣΧΗΜΑ Α.12 Επέκταση πρόσημου

Α.2.10 Καθυστερήσεις

Στις γλώσσες HDL, οι εντολές μπορούν να συσχετίζονται με καθυστερήσεις οι οποίες καθορίζονται σε αυθαίρετες μονάδες. Αυτό είναι χρήσιμο κατά την προσομοίωση, επιτρέποντάς σας να προβλέψετε πόσο γρήγορα θα λειτουργεί ένα κύκλωμα (εάν, βέβαια, καθορίσετε λογικές καθυστερήσεις), καθώς επίσης και για σκοπούς αποσφαλμάτωσης, επειδή σας επιτρέπει να κατανοήσετε το αίτιο και το αποτέλεσμα (ο εντοπισμός του αιτίου μιας προβληματικής εξόδου είναι δύσκολη υπόθεση εάν όλα τα σήματα μεταβάλλονται ταυτόχρονα στα αποτελέσματα της προσομοίωσης). Αυτές οι καθυστερήσεις αγνοούνται κατά τη σύνθεση· η καθυστέρηση μιας πύλης που παράγεται από το εργαλείο σύνθεσης εξαρτάται από τις προδιαγραφές της για τους χρόνους t_{pd} και t_{cd} – όχι από τιμές στον HDL κώδικα.

Το Παράδειγμα Α.16 προσθέτει καθυστερήσεις στην αρχική συνάρτηση του Παραδείγματος Α.1: $Y = \overline{A}\overline{B}\overline{C} + A\overline{B}\overline{C} + A\overline{B}C$. Το παράδειγμα υποθέτει ότι οι αντιστροφείς έχουν καθυστέρηση 1 ns, οι πύλες AND 3 εισόδων έχουν καθυστέρηση 2 ns και οι πύλες OR 3 εισόδων έχουν καθυστέρηση 4 ns. Το Σχήμα Α.13 παρουσιάζει τις κυματομορφές που δίνει η προσομοίωση, με το y να καθυστερεί κατά 7 ns μετά από τις εισόδους. Σημειώστε ότι το y είναι άγνωστο κατά την έναρξη της προσομοίωσης.

Παράδειγμα Α.16 Λογικές Πύλες με Καθυστερήσεις

SystemVerilog

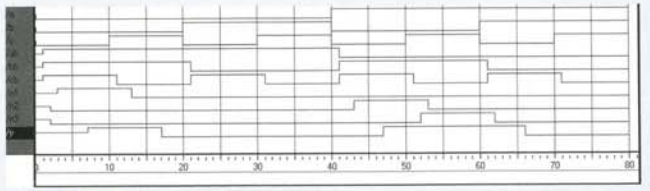
```
 timescale 1ns/1ps

module example(input  logic a, b, c,
               output logic y);

  logic ab, bb, cb, n1, n2, n3;

  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```

Τα αρχεία της SystemVerilog μπορούν να περιλαμβάνουν μια ντιρεκτίβα κλίμακας χρόνου (`timescale`), η οποία υποδεικνύει την τιμή κάθε μονάδας χρόνου. Η δήλωση γι' αυτό έχει τη μορφή `timescale unit/step` (μονάδα/βήμα). Σ' αυτό το αρχείο, κάθε μονάδα χρόνου είναι 1ns και η προσομοίωση έχει ανάλυση (resolution) 1 ps. Εάν δεν δίνεται ντιρεκτίβα timescale στο αρχείο, χρησιμοποιούνται οι προεπιλεγμένες ρυθμίσεις για τη μονάδα και το βήμα (συνήθως 1 ns και για τα δύο). Στην SystemVerilog, χρησιμοποιείται ένα σύμβολο # για να υποδείξει τον αριθμό μονάδων καθυστέρησης. Μπορεί να τοποθετείται σε εντολές ανάθεσης (`assign`), καθώς και σε πράξεις μη-κλειδωμένης (nonblocking, <=) και κλειδωμένης (blocking, =) ανάθεσης, τις οποίες θα εξετάσουμε στην Ενότητα Α.5.4.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity example is
  port(a, b, c: in  STD_LOGIC;
       y:       out STD_LOGIC);
end;


architecture synth of example is
  signal ab, bb, cb, n1, n2, n3: STD_LOGIC;
begin
  ab <= not a after 1 ns;
  bb <= not b after 1 ns;
  cb <= not c after 1 ns;
  n1 <= ab and bb and cb after 2 ns;
  n2 <= a and bb and cb after 2 ns;
  n3 <= a and bb and c after 2 ns;
  y  <= n1 or n2 or n3 after 4 ns;
end;
```

Στην VHDL, ο όρος `after` χρησιμοποιείται για να υποδείξει καθυστέρηση. Οι μονάδες, σ' αυτή την περίπτωση, καθορίζονται σε νανοδευτερόλεπτα.



ΣΧΗΜΑ Α.13 Παράδειγμα κυματομορφών προσομοίωσης με καθυστερήσεις

Α.3 Μοντελοποίηση σε Επίπεδο Δομής

Στην προηγούμενη ενότητα εξετάσαμε τη *μοντελοποίηση σε επίπεδο συμπεριφοράς*, η οποία περιγράφει μια λειτουργική μονάδα βάσει των σχέσεων μεταξύ των εισόδων και των εξόδων. Σ' αυτή την ενότητα θα εξετάσουμε τη *μοντελοποίηση σε επίπεδο δομής*, η οποία περιγράφει μια λειτουργική μονάδα βάσει του πώς «συναρμολογείται» από απλούστερες μονάδες.

Το Παράδειγμα Α.17 δείχνει πώς «συναρμολογείται» ένας πολυπλέκτης 4:1 από τρεις πολυπλέκτες 2:1. Κάθε αντίγραφο του πολυπλέκτη 2:1 αποκαλείται *εμφάνιση* ή *υπόδειγμα* (υπόδειγμα/υποδείγματος). Τα πολλαπλά υποδείγματα της ίδιας μονάδας διαφοροποιούνται με τη χρήση διαφορετικών ονομάτων. Αυτό είναι ένα παράδειγμα της αρχής της κανονικότητας, βάσει της οποίας ο πολυπλέκτης 2:1 επαναχρησιμοποιείται συνολικά τρεις φορές.

Παράδειγμα A.17 Δομικό Μοντέλο Ενός Πολυπλέκτη 4:1

SystemVerilog

```
module mux4(input  logic [3:0] d0, d1, d2, d3,
            input  logic [1:0] s,
            output logic [3:0] y);

  logic [3:0] low, high;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 highmux(d2, d3, s[0], high);
  mux2 finalmux(low, high, s[1], y);
endmodule
```

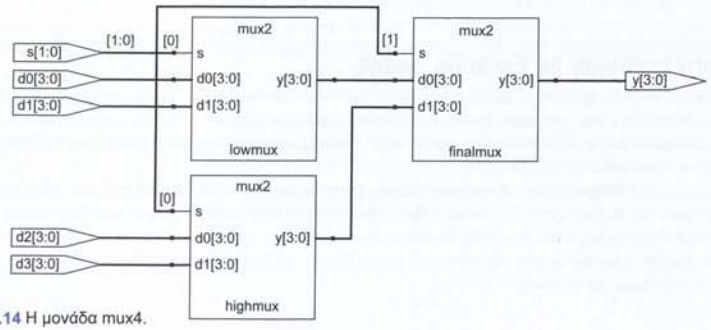
Τα τρία υποδείγματα της μονάδας `mux2` ονομάζονται `lowmux`, `highmux` και `finalmux`. Η μονάδα `mux2` πρέπει να έχει οριστεί σ' ένα άλλο σημείο του κώδικα.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4 is
  port(d0, d1,
       d2, d3: in  STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC_VECTOR(1 downto 0);
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture struct of mux4 is
  component mux2
    port(d0,
      d1: in  STD_LOGIC_VECTOR(3 downto 0);
      s:  in  STD_LOGIC;
      y:  out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal low, high: STD_LOGIC_VECTOR(3 downto 0);
begin
  lowmux:    mux2 port map(d0, d1, s(0), low);
  highmux:   mux2 port map(d2, d3, s(0), high);
  finalmux:  mux2 port map(low, high, s(1), y);
end;
```

Η εντολή `architecture` πρέπει κατ' αρχήν να δηλώσει τις θύρες της μονάδας `mux2` χρησιμοποιώντας τη δήλωση `component` (συστατικό). Αυτό επιτρέπει στα εργαλεία της VHDL να ελέγξουν εάν το συστατικό που θέλετε να χρησιμοποιήσετε έχει τις ίδιες θύρες με το συστατικό που δηλώθηκε κάπου αλλού, στη δήλωση μιας άλλης οντότητας (`entity`), αποτρέποντας έτσι τα σφάλματα που προκαλούνται εάν αλλάξει αυτή η οντότητα αλλά όχι το χρησιμοποιούμενο υπόδειγμά της. Ωστόσο, η ανάγκη για δήλωση των συστατικών στον VHDL κώδικα είναι κοπιαστική υπόθεση.

Σημειώστε ότι η αρχιτεκτονική για τη συγκεκριμένη μονάδα mux4 ονομάστηκε *struct*, ενώ οι αρχιτεκτονικές των λειτουργικών μονάδων με περιγραφές συμπεριφοράς (από την Ενότητα A.2) λάμβαναν το όνομα *synth*. Η VHDL επιτρέπει πολλαπλές αρχιτεκτονικές (υλοποιήσεις) για την ίδια οντότητα· οι αρχιτεκτονικές διαφοροποιούνται βάσει ονόματος. Τα ίδια τα ονόματα δεν έχουν σημασία για τα εργαλεία CAD, αλλά τα struct και synth έχουν καθιερωθεί και χρησιμοποιούνται ευρέως. Ωστόσο, επειδή γενικά ο συνθέσιμος VHDL κώδικας περιέχει μόνο μία αρχιτεκτονική για κάθε οντότητα, δεν θα περιγράψουμε τη σύνταξη που χρησιμοποιείται στην VHDL για να καθοριστεί ποια αρχιτεκτονική χρησιμοποιείται στις περιπτώσεις όπου ορίζονται πολλαπλές αρχιτεκτονικές.



ΣΧΗΜΑ A.14 Η μονάδα mux4.

Κατά παρόμοιο τρόπο, το Παράδειγμα A.18 κατασκευάζει έναν πολυπλέκτη 2:1 από ένα ζεύγος τριτσταθών απομονωτών.

Παράδειγμα A.18 Δομικό Μοντέλο Πολυπλέκτη 2:1

SystemVerilog

```
module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output tri   [3:0] y);

  tristate t0(d0, ~s, y);
  tristate t1(d1, s, y);
endmodule
```

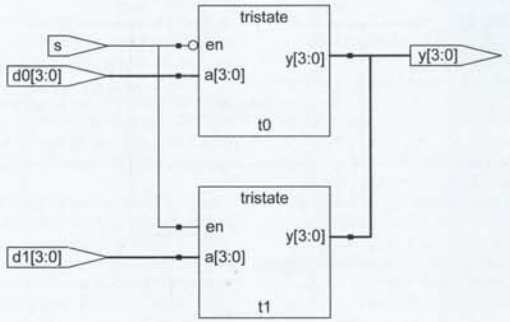
Στην SystemVerilog, επιτρέπεται η χρήση εκφράσεων όπως η `~s` στη λίστα θυρών για ένα υπόδειγμα οντότητας. Επιπλέον, εκφράσεις οποιουδήποτε βαθμού πολυπλοκότητας είναι έγκυρες, αλλά η πρακτική αυτή δεν συνιστάται επειδή δυσχεραίνει την κατανόηση του κώδικα.

Σημειώστε ότι το `y` δηλώνεται ως τύπου `tri` και όχι ως `logic` επειδή έχει δύο οδηγούς.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  port(d0, d1: in STD_LOGIC_VECTOR(3 downto 0);
       s:      in  STD_LOGIC;
       y:      out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture struct of mux2 is
  component tristate
    port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
         en: in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal sbar: STD_LOGIC;
begin
  sbar <= not s;
  t0: tristate port map(d0, sbar, y);
  t1: tristate port map(d1, s, y);
end;
```

Στην VHDL, εκφράσεις όπως η `not s` δεν επιτρέπονται στο χάρτη θυρών (port map) για ένα υπόδειγμα οντότητας. Συνεπώς, το `sbar` πρέπει να ορίζεται ως ξεχωριστό σήμα.



ΣΧΗΜΑ A.15 Η μονάδα mux2.

Το Παράδειγμα A.19 δείχνει πώς μία λειτουργική μονάδα μπορεί να προσπελάζει μέρος ενός διαύλου. Ένας πολυπλέκτης 2:1 με εύρος 8 bit κατασκευάζεται χρησιμοποιώντας δύο υποδείγματα του 4-bit πολυπλέκτη 2:1 που έχουμε ήδη ορίσει, τα οποία επενεργούν στο κατώτερο και ανώτερο τμήμα του byte.

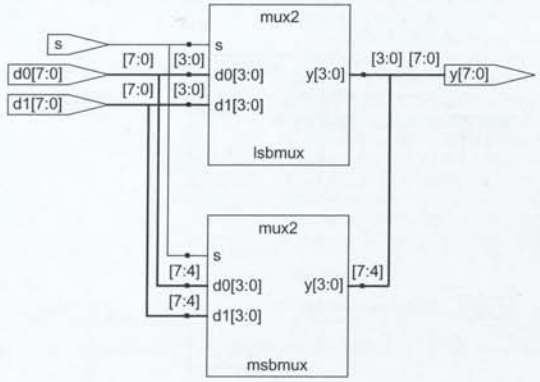
Παράδειγμα A.19 Προσπέλαση Μέρους Ενός Διαύλου

SystemVerilog

```
module mux2_8(input  logic [7:0] d0, d1,
              input  logic       s,
              output logic [7:0] y);

  mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2_8 is
  port(d0, d1:in  STD_LOGIC_VECTOR(7 downto 0);
       s:     in  STD_LOGIC;
       y:     out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture struct of mux2_8 is
  component mux2
    port(d0, d1: in  STD_LOGIC_VECTOR(3
                                      downto 0);
         s:  in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(3 downto 0));
  end component;
begin

  lsbmux:  mux2
    port map(d0(3 downto 0), d1(3 downto 0),
             s, y(3 downto 0));
  msbhmux: mux2
    port map(d0(7 downto 4), d1(7 downto 4),
             s, y(7 downto 4));
end;
```



ΣΧΗΜΑ A.16 Η μονάδα mux2_8.

Γενικά, τα πολύπλοκα συστήματα σχεδιάζονται *ιεραρχικά*. Το συνολικό σύστημα περιγράφεται, σε επίπεδο δομής, δημιουργώντας υποδείγματα των σημαντικών συστατικών του. Κάθε ένα από αυτά τα συστατικά περιγράφεται, σε επίπεδο δομής, μέσω των δικών του δομικών συστατικών και αυτή η διαδικασία επαναλαμβάνεται με αναδρομικό τρόπο έως ότου να καταλήξουμε σε μονάδες επαρκώς απλές για να περιγράψουμε το σύστημα σε επίπεδο συμπεριφοράς. Μια καλή πρακτική είναι να αποφεύγετε (ή τουλάχιστον να ελαχιστοποιείτε) την ανάμειξη περιγραφών δομής και συμπεριφοράς μέσα στην ίδια λειτουργική μονάδα.

A.4 Ακολουθιακή Λογική

Τα HDL εργαλεία σύνθεσης αναγνωρίζουν συγκεκριμένα ιδιώματα (στιλ γραφής κώδικα) και τα μετατρέπουν σε συγκεκριμένα ακολουθιακά κυκλώματα. Άλλα στιλ κώδικα μπορεί να περνούν σωστά από την προσομοίωση, αλλά να συντίθενται σε κυκλώματα με κραυγαλέα ή αδιόρατα σφάλματα. Σ' αυτή την ενότητα θα παρουσιάσουμε τα σωστά στιλ γραφής κώδικα για την περιγραφή καταχωρητών και μανδαλωτών.

A.4.1 Καταχωρητές

Η συντριπτική πλειονότητα των σύγχρονων εμπορικών συστημάτων κατασκευάζονται με καταχωρητές, χρησιμοποιώντας θετικά ακμοπυροδότητα D flip-flop. Το Παράδειγμα A.20 παρουσιάζει τη σύνταξη για ένα τέτοιο flip-flop.

Παράδειγμα A.20 Καταχωρητές

SystemVerilog

```
module flop(input  logic       clk,
            input  logic [3:0] d,
            output logic [3:0] q);

  always_ff @(posedge clk)
    q <= d;
endmodule
```

Στην Verilog, μια εντολή `always` γράφεται με τη μορφή

```
always @(λίστα ευαισθησίας)
  εντολή;
```

Η αναφερόμενη εντολή εκτελείται μόνο όταν λαμβάνει χώρα το συμβάν που καθορίζεται στη λίστα ευαισθησίας (sensitivity list). Σ' αυτό το παράδειγμα, η εντολή είναι `q <= d` (διατυπώνεται ως «η έξοδος q λαμβάνει την είσοδο d»). Δηλαδή, το flip-flop αντιγράφει την είσοδο d στην έξοδο q κατά τη θετική ακμή του ρολογιού· διαφορετικά, θυμάται την προηγούμενη κατάσταση της q.

Ο τελεστής <= εκτελεί *μη-κλειδωμένη ανάθεση* (nonblocking assignment). Για την ώρα, μπορείτε να τον αντιμετωπίσετε ως συμβατικό τελεστή =· θα ασχοληθούμε περισσότερο μαζί του στην Ενότητα A.5.4. Σημειώστε ότι το <= χρησιμοποιείται αντί της `assign` μέσα σε μια εντολή `always`.

Όπως θα δούμε σε επόμενες ενότητες, οι εντολές always μπορούν να χρησιμοποιούνται για να υποδηλώσουν flip-flop, μανδαλωτές, ή συνδυαστική λογική, ανάλογα με τη «λίστα ευαισθησίας» και την «εντολή». Λόγω αυτής της ευελιξίας, είναι πολύ εύκολο να καταλήξει κανείς μ' ένα λανθασμένο σύστημα. Για να μειώσει τον κίνδυνο τέτοιων κοινών σφαλμάτων, η SystemVerilog εισάγει τις εντολές `always_ff`, `always_latch` και `always_comb`. Η `always_ff` συμπεριφέρεται όμοια με την `always`, αλλά χρησιμοποιείται για να υποδηλώσει αποκλειστικά flip-flop και επιτρέπει στα εργαλεία να παράγουν ένα προειδοποιητικό μήνυμα εάν υποδηλώνεται οτιδήποτε διαφορετικό.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flop is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC_VECTOR(3 downto 0);
       q:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of flop is
begin
  process(clk) begin
    if clk'event and clk = '1' then
      q <= d;
    end if;
  end process;
end;
```

Στην VHDL, μια εντολή "διεργασίας", `process`, έχε τη μορφή

```
process(λίστα ευαισθησίας) begin
  εντολή;
end process;
```

Η αναφερόμενη εντολή εκτελείται όταν μεταβάλλεται οποιαδήποτε από τις μεταβλητές της λίστας ευαισθησίας. Σ' αυτό το παράδειγμα, η εντολή `if` εκτελείται όταν μεταβάλλεται το `clk`, πράγμα το οποίο υποδεικνύεται από ένα συμβάν `clk'event`. Εάν η μεταβολή είναι μια ανοδική ακμή ρολογιού (`clk` = '1' μετά από το συμβάν), τότε `q <= d`. Δηλαδή, το flip-flop αντιγράφει την είσοδο d στην έξοδο q κατά τη θετική ακμή του ρολογιού· διαφορετικά, θυμάται την προηγούμενη κατάσταση της q. Μια εναλλακτική σύνταξη της VHDL για ένα flip-flop είναι η ακόλουθη

```
process(clk) begin
  if RISING_EDGE(clk) then
    q <= d;
  end if;
end process;
```

Το `RISING_EDGE(clk)` είναι ταυτόσημο με τα `clk'event` και `clk = '1'`.

ΣΧΗΜΑ A.17 Flip-flop.

Στις εντολές `always` (SystemVerilog) και `process` (VHDL), τα σήματα διατηρούν την παλαιά τιμή τους έως ότου λάβει χώρα ένα συμβάν, το οποίο προκαλεί ρητά την αλλαγή τους. Αυτό σημαίνει ότι τέτοιος κώδικας, με κατάλληλες λίστες ευαισθησίας, μπορεί να χρησιμοποιηθεί για να περιγράψει ακολουθιακά κυκλώματα με μνήμη. Για παράδειγμα, το flip-flop περιλαμβάνει μόνο το σήμα `clk` στη λίστα ευαισθησίας. Θυμάται την παλαιά τιμή του q έως την επόμενη ανοδική ακμή του `clk`, ακόμα κι αν το d αλλάξει στο διάστημα που μεσολαβεί.

Εν αντιθέσει, οι εντολές συνεχούς ανάθεσης της SystemVerilog και οι εντολές ταυτόχρονης ανάθεσης της VHDL αποτιμώνται εκ νέου οποτεδήποτε αλλάζει οποιαδήποτε από τις εισόδους που αναφέρονται στη δεξιά πλευρά της εντολής. Συνεπώς, αυτό το είδος κώδικα περιγράφει κατ' ανάγκην συνδυαστική λογική.

A.4.2 Καταχωρητές με Δυνατότητα Επαναφοράς (Resettable)

Κατά την έναρξη της προσομοίωσης ή της τροφοδοσίας ενός κυκλώματος, η έξοδος του flip-flop είναι άγνωστη. Αυτό υποδεικνύεται με το x στην SystemVerilog και με το 'u' στην VHDL. Γενικά, είναι καλή πρακτική να χρησιμοποιείτε καταχωρητές με δυνατότητα επαναφοράς (resettable), έτσι ώστε κατά την έναρξη λειτουργίας να μπορείτε να θέτετε το σύστημά σας σε μια γνωστή κατάσταση. Το σήμα reset μπορεί να είναι είτε σύγχρονο είτε ασύγχρονο. Όπως γνωρίζετε, το σύγχρονο reset λαμβάνει χώρα κατά την ανοδική ακμή του ρολογιού, ενώ το ασύγχρονο reset λαμβάνει χώρα αμέσως. Το Παράδειγμα A.21 επιδεικνύει τη σύνταξη για flip-flop με σύγχρονο και ασύγχρονο reset. Σημειώστε ότι η διάκριση μεταξύ των σύγχρονων από και ασύγχρονων reset σ' ένα σχηματικό μπορεί να είναι δύσκολη υπόθεση. Το σχηματικό που παράγει το Synplify Pro τοποθετεί το σύγχρονο reset στην αριστερή πλευρά ενός flip-flop και το σύγχρονο reset στην κάτω.

Το σύγχρονο reset απαιτεί λιγότερα τρανζίστορ και μειώνει τον κίνδυνο προβλημάτων χρονισμού στην καθοδική ακμή του σήματος reset. Ωστόσο, εάν χρησιμοποιείται σύνδεση με ρολόι μέσω πύλης (clock gating), θα πρέπει να δίνετε ιδιαίτερη προσοχή ώστε όλα τα flip-flop να τίθενται σωστά κατά την εκκίνηση.

Παράδειγμα A.21 Καταχωρητές με Δυνατότητα Επαναφοράς

SystemVerilog

```
module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // synchronous reset
  always_ff @(posedge clk)
    if (reset) q <= 4'b0;
    else       q <= d;
endmodule

module flopr(input  logic       clk,
             input  logic       reset,
             input  logic [3:0] d,
             output logic [3:0] q);

  // asynchronous reset
  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 4'b0;
    else       q <= d;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flopr is
  port(clk,
       reset: in  STD_LOGIC;
       d:     in  STD_LOGIC_VECTOR(3 downto 0);
       q:     out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synchronous of flopr is
begin
  process(clk) begin
    if clk'event and clk = '1' then
      if reset = '1' then
        q <= "0000";
      else q <= d;
      end if;
    end if;
  end process;
end;
```


SystemVerilog (συνέχεια)

Τα σήματα στη λίστα ευαισθησίας μιας `always` διαχωρίζονται με κόμματα, ή με τη λέξη or. Παρατηρήστε ότι το σήμα `posedge reset` περιλαμβάνεται στη λίστα ευαισθησίας του flip-flop με ασύγχρονο reset, αλλά όχι σ' αυτήν του flip-flop με σύγχρονο reset. Αυτό σημαίνει ότι το flip-flop με ασύγχρονο reset ανταποκρίνεται άμεσα σε μια ανοδική ακμή του σήματος reset, ενώ το flip-flop με σύγχρονο `reset` ανταποκρίνεται στο σήμα `reset` μόνο κατά την ανοδική ακμή του ρολογιού.

Επειδή οι παραπάνω λειτουργικές μονάδες έχουν το ίδιο όνομα(`flopr`), θα πρέπει να συμπεριλάβετε μόνο τη μία ή την άλλη στη σχεδίασή σας.

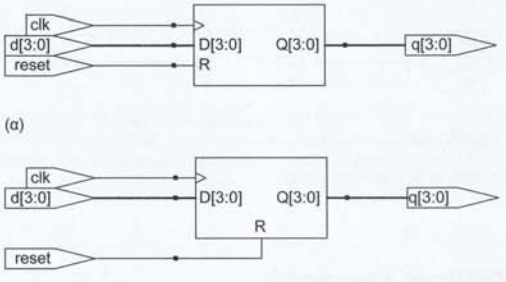
VHDL (συνέχεια)

```
architecture asynchronous of flopr is
begin
  process(clk, reset) begin
    if reset = '1' then
      q <= "0000";
    elsif clk'event and clk = '1' then
      q <= d;
    end if;
  end process;
end;
```

Τα σήματα στη λίστα ευαισθησίας μιας `process` διαχωρίζονται με κόμματα. Παρατηρήστε ότι το σήμα reset περιλαμβάνεται στη λίστα ευαισθησίας για το flip-flop με ασύγχρονο reset, αλλά όχι σ' αυτήν για το flip-flop με σύγχρονο `reset`. Συνεπώς, το flip-flop με ασύγχρονο `reset` ανταποκρίνεται άμεσα σε μια ανοδική ακμή του σήματος reset, ενώ το flip-flop με σύγχρονο reset ανταποκρίνεται στο σήμα reset μόνο κατά την ανοδική ακμή του ρολογιού.

Θυμηθείτε ότι η κατάσταση ενός flip-flop αρχικοποιείται σε 'u' κατά την έναρξη της προσομοίωσης στην VHDL.

Όπως προαναφέραμε, το όνομα της αρχιτεκτονικής (asynchronous ή synchronous στο παράδειγμά μας) αγνοείται από τα εργαλεία VHDL, αλλά μπορεί να είναι χρήσιμο βοήθημα για κάποιον που διαβάζει τον κώδικα. Επειδή αμφότερες οι αρχιτεκτονικές περιγράφουν την οντότητα flopr, θα πρέπει να συμπεριλάβετε μόνο τη μία ή την άλλη στη σχεδίασή σας.



ΣΧΗΜΑ A.18 Η μονάδα flopr (α) με σύγχρονο reset, (β) με ασύγχρονο reset.

A.4.3 Καταχωρητές με Enable

Οι καταχωρητές με σήμα enable ανταποκρίνονται στο ρολόι μόνο όταν δίνεται το σήμα enable (en). Το Παράδειγμα A.22 παρουσιάζει έναν καταχωρητή με σύγχρονο reset και enable, ο οποίος διατηρεί την παλαιά του τιμή εάν αμφότερα τα σήματα `reset` και en είναι FALSE.

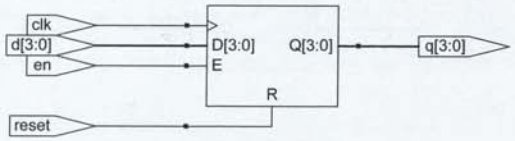
Παράδειγμα A.22 Καταχωρητής με reset και enable

SystemVerilog

```
module flopenr(input  logic       clk,
               input  logic       reset,
               input  logic       en,
               input  logic [3:0] d,
               output logic [3:0] q);

  // synchronous reset
  always_ff @(posedge clk)
    if      (reset) q <= 4'b0;
    else if (en)    q <= d;
endmodule
```

VHDL

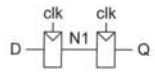
```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity flopenr is
  port(clk,
       reset,
       en: in  STD_LOGIC;
       d:  in  STD_LOGIC_VECTOR(3 downto 0);
       q:  out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synchronous of flopenr is
-- synchronous reset
begin
  process(clk) begin
    if clk'event and clk = '1' then
      if reset = '1' then
        q <= "0000";
      elsif en = '1' then
        q <= d;
      end if;
    end if;
  end process;
end;
```



ΣΧΗΜΑ A.19 Η μονάδα flopenr.

A.4.4 Πολλαπλοί Καταχωρητές

Μία και μόνο εντολή `always/process` μπορεί να χρησιμοποιηθεί για την περιγραφή πολλαπλών μονάδων hardware. Για παράδειγμα, εξετάστε την περιγραφή ενός συγχρονιστή αποτελούμενου από δύο flip-flop συνδεδεμένα «πλάτη με πλάτη», όπως παρουσιάζεται στο Σχήμα A.20. Το Παράδειγμα A.23 περιγράφει το συγχρονιστή. Στην ανοδική ακμή του `clk`, το `d` αντιγράφεται στο `n1`. Την ίδια στιγμή, το `n1` αντιγράφεται στην έξοδο `q`.



ΣΧΗΜΑ A.20 Το κύκλωμα του συγχρονιστή.

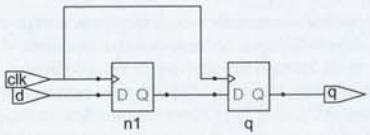
Παράδειγμα A.23 Συγχρονιστής

SystemVerilog

```
module sync(input  logic clk,
            input  logic d,
            output logic q);

  logic n1;

  always_ff @(posedge clk)
    begin
      n1 <= d;
      q <= n1;
    end
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity sync is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC;
       q:   out STD_LOGIC);
end;

architecture synth of sync is
  signal n1: STD_LOGIC;
begin
  process(clk) begin
    if clk'event and clk = '1' then
      n1 <= d;
      q <= n1;
    end if;
  end process;
end;
```



ΣΧΗΜΑ A.21 Η μονάδα sync.

A.4.5 Μανδαλωτές

Όπως γνωρίζετε, ένας *D* μανδαλωτής είναι διαφανής όταν το ρολόι είναι HIGH, επιτρέποντας ροή δεδομένων από την είσοδο στην έξοδο. Ο μανδαλωτής γίνεται αδιαφανής όταν το ρολόι είναι LOW, οπότε και διατηρεί την παλαιά του κατάσταση. Το Παράδειγμα A.24 παρουσιάζει τη σύνταξη για έναν *D* μανδαλωτή.

Παράδειγμα A.24 D Μανδαλωτής

SystemVerilog

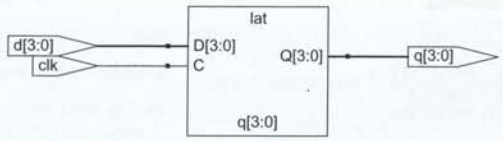
```
module latch(input  logic       clk,
             input  logic [3:0] d,
             output logic [3:0] q);

  always_latch
    if (clk) q <= d;
endmodule
```

Η `always_latch` είναι ισοδύναμη με την `always @(clk, d)` και αποτελεί τον προτιμώμενο τρόπο περιγραφής ενός μανδαλωτή στην SystemVerilog. Αποτιμάται οποτεδήποτε μεταβάλλεται το `clk` ή το `d`. Εάν το `clk` είναι HIGH, υπάρχει ροή δεδομένων από την είσοδο `d` έως την έξοδο `q`, πράγμα το οποίο σημαίνει ότι αυτός ο κώδικας περιγράφει ένα μανδαλωτή ευαίσθητο σε θετική στάθμη. Διαφορετικά, η έξοδος `q` διατηρεί την παλαιά τιμή της. Η SystemVerilog μπορεί να παράγει ένα προειδοποιητικό μήνυμα εάν το μπλοκ κώδικα της `always_latch` δεν υποδηλώνει ένα μανδαλωτή.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity latch is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC_VECTOR(3 downto 0);
       q:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of latch is
begin
  process(clk, d) begin
    if clk = '1' then q <= d;
    end if;
  end process;
end;
```

Η λίστα ευαισθησίας περιλαμβάνει αμφότερα τα `clk` και `d`, οπότε η `process` αποτιμάται οποτεδήποτε μεταβάλλεται το `clk` ή το `d`. Εάν το `clk` είναι HIGH, υπάρχει ροή δεδομένων από την είσοδο `d` στην έξοδο `q`.



ΣΧΗΜΑ A.22 Μανδαλωτής.

Δεν υποστηρίζουν καλά τους μανδαλωτές όλα τα εργαλεία σύνθεσης. Καλό θα είναι να τους αποφεύγετε, και να χρησιμοποιείτε ακμοπυροδότητα flip-flop στη θέση τους, εκτός κι αν γνωρίζετε ότι το εργαλείο σας υποστηρίζει μανδαλωτές κι έχετε καλό λόγο για να τους χρησιμοποιήσετε. Επιπλέον, θα πρέπει να φροντίζετε ώστε η HDL περιγραφή σας να μην υποδηλώνει την ύπαρξη μανδαλωτών που δεν έχετε καθορίσει εσείς ρητά, κάτι το οποίο είναι εύκολο να συμβεί εάν δεν είστε προσεκτικοί. Πολλά εργαλεία σύνθεσης εμφανίζουν προειδοποιητικά μηνύματα όταν δημιουργείται ένας μανδαλωτής· εάν δεν τον ορίσατε εσείς, αναζητήστε το αίτιο του σφάλματος στον HDL κώδικά σας. Και, εάν δεν είστε σε θέση να εξακριβώσετε εάν ο μανδαλωτής τοποθετήθηκε σκόπιμα από εσάς ή όχι, αυτό υποδηλώνει ότι αντιμετωπίζετε τις γλώσσες HDL ως γλώσσες προγραμματισμού και κατά πάσα πιθανότητα ελλοχεύουν μεγαλύτερα προβλήματα στον κώδικά σας.

A.4.6 Μετρητές

Θα εξετάσουμε δύο τρόπους περιγραφής ενός μετρητή των 4 bit με σύγχρονο reset. Η πρώτη προσέγγιση (σε επίπεδο συμπεριφοράς) υποδηλώνει ένα ακολουθιακό κύκλωμα, το οποίο περιλαμβάνει τον 4-bit καταχωρητή κι έναν αθροιστή. Η δεύτερη προσέγγιση (σε επίπεδο δομής) δηλώνει ρητά τις λειτουργικές μονάδες για τον καταχωρητή και τον αθροιστή. Αμφότερες οι προσεγγίσεις είναι καλές για ένα απλό κύκλωμα, όπως ο μετρητής. Ωστόσο, καθώς αναπτύσσετε πολυπλοκότερες μηχανές πεπερασμένων καταστάσεων, καλή ιδέα είναι να διαχωρίζετε τη λογική για την επόμενη κατάσταση από τους καταχωρητές στον HDL κώδικά σας. Τα Παραδείγματα A.25 και A.26 επιδεικνύουν αυτές τις δύο προσεγγίσεις.

Παράδειγμα A.25 Μετρητής (προσέγγιση σε επίπεδο συμπεριφοράς)

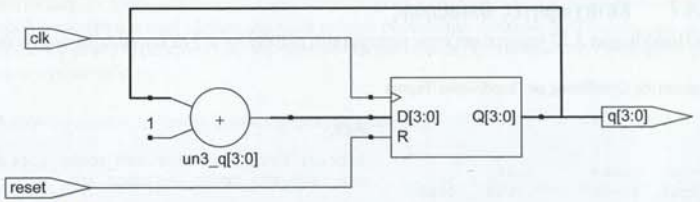
SystemVerilog

```
module counter(input  logic       clk,
               input  logic       reset,
               output logic [3:0] q);

  always_ff @(posedge clk)
    if (reset) q <= 4'b0;
    else       q <= q+1;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity counter is
  port(clk:   in  STD_LOGIC;
       reset: in  STD_LOGIC;
       q:     out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of counter is
  signal q_int: STD_LOGIC_VECTOR(3 downto 0);
begin
  process(clk) begin
    if clk'event and clk = '1' then
      if reset = '1' then q_int <= "0000";
      else q_int <= q_int + "0001";
      end if;
    end if;
  end process;
  q <= q_int;
end;
```

Στην VHDL, μια έξοδος δεν μπορεί να χρησιμοποιείται και στην αριστερή και στη δεξιά πλευρά μιας έκφρασης· η έκφραση `q <= q + 1` δεν είναι έγκυρη. Για το λόγο αυτό, ορίζεται ένα εσωτερικό σήμα κατάστασης `q_int` και η έξοδος `q` είναι ένα αντίγραφο του `q_int`. Θα μιλήσουμε περισσότερο γι' αυτό στην Ενότητα A.7.



ΣΧΗΜΑ A.23 Μετρητής (προσέγγιση σε επίπεδο συμπεριφοράς)

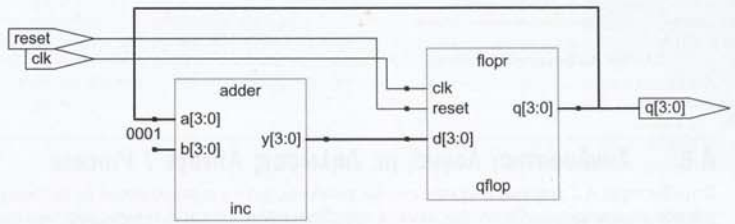
Παράδειγμα A.26 Μετρητής (προσέγγιση σε επίπεδο δομής)

SystemVerilog

```
module counter(input  logic       clk,
               input  logic       reset,
               output logic [3:0] q);

   logic [3:0] nextq;

   flopr qflop(clk, reset, nextq, q);
   adder inc(q, 4'b0001, nextq);
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity counter is
  port(clk:   in  STD_LOGIC;
       reset: in  STD_LOGIC;
       q:     out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture struct of counter is
  component flopr
    port(clk:   in  STD_LOGIC;
         reset: in  STD_LOGIC;
         d:     in  STD_LOGIC_VECTOR(3 downto 0);
         q:     out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  component adder
    port(a, b: in  STD_LOGIC_VECTOR(3 downto 0);
         y:    out STD_LOGIC_VECTOR(3 downto 0));
  end component;
  signal nextq, q_int: STD_LOGIC_VECTOR(3 downto 0);
begin
  qflop: flopr port map(clk, reset, nextq, q_int);
  inc:   adder port map(q_int, "0001", nextq);
  q <= q_int;
end;
```



ΣΧΗΜΑ A.24 Μετρητής (προσέγγιση σε επίπεδο δομής)

A.4.7 Καταχωρητές Ολίσθησης

Το Παράδειγμα A.27 περιγράφει έναν καταχωρητή ολίσθησης μ' ένα παράλληλο φορτίο στην είσοδο.

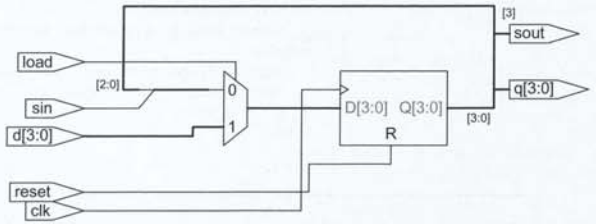
Παράδειγμα A.27 Καταχωρητής Ολίσθησης με Παράλληλο Φορτίο

SystemVerilog

```
module shiftreg(input  logic       clk,
                input  logic       reset, load,
                input  logic       sin,
                input  logic [3:0] d,
                output logic [3:0] q,
                output logic       sout);

  always_ff @(posedge clk)
    if (reset)     q <= 0;
    else if (load) q <= d;
    else           q <= {q[2:0], sin};

  assign sout = q[3];
endmodule
```

VHDL

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

entity shiftreg is
  port(clk, reset,
       load: in  STD_LOGIC;
       sin:  in  STD_LOGIC;
       d:    in  STD_LOGIC_VECTOR(3 downto 0);
       q:    out STD_LOGIC_VECTOR(3 downto 0);
       sout: out STD_LOGIC);
end;

architecture synth of shiftreg is
  signal q_int: STD_LOGIC_VECTOR(3 downto 0);
begin
  process(clk) begin
    if clk'event and clk = '1' then
      if reset = '1' then q_int <= "0000";
      elsif load = '1' then q_int <= d;
      else q_int <= q_int(2 downto 0) & sin;
      end if;
    end if;
  end process;

  q    <= q_int;
  sout <= q_int(3);
end;
```



ΣΧΗΜΑ A.25 Η μονάδα shiftreg.

A.5 Συνδυαστική Λογική με Δηλώσεις Always / Process

Στην Ενότητα A.2 χρησιμοποιήσαμε εντολές ανάθεσης για να περιγράψουμε τη συνδυαστική λογική σε επίπεδο συμπεριφοράς. Οι εντολές `always` στην SystemVerilog και οι εντολές `process` στην VHDL χρησιμοποιούνται για την περιγραφή ακολουθιακών κυκλωμάτων, επειδή «θυμούνται» την παλαιά κατάσταση όταν δεν προδιαγράφεται νέα κατάσταση. Ωστόσο, οι εντολές `always/process` μπορούν επίσης να χρησιμοποιούνται για την περιγραφή συνδυαστικής λογικής σε επίπεδο συμπεριφοράς, εάν η λίστα

ευαισθησίας διατυπωθεί με τρόπο ώστε να ανταποκρίνεται σε μεταβολές όλων των εισόδων και στον κορμό της εντολής προδιαγράφεται η τιμή εξόδου για κάθε πιθανό συνδυασμό εισόδων. Το Παράδειγμα A.28 χρησιμοποιεί εντολές `always/process` για να περιγράψει μια συστοιχία τεσσάρων αντιστροφέων (βλ. Σχήμα A.4 για το σχηματικό).

Παράδειγμα A.28 Αντιστροφέας (με χρήση εντολών `always/process`)

SystemVerilog

```
module inv(input  logic [3:0] a,
           output logic [3:0] y);

  always_comb
    y = ~a;
endmodule
```

Η `always_comb` είναι ισοδύναμη με την `always @(*)` και αποτελεί τον προτιμώμενο τρόπο περιγραφής της συνδυαστικής λογικής στην SystemVerilog. Η `always_comb` αποτιμά εκ νέου τις εντολές που περιλαμβάνονται μέσα στην `always` οποτεδήποτε μεταβάλλεται κάποιο από τα σήματα που αναφέρονται στη δεξιά πλευρά του <= ή = μέσα στην εντολή `always`. Αυτό σημαίνει ότι η `always_comb` είναι ένας ασφαλής τρόπος για τη μοντελοποίηση συνδυαστικής λογικής. Σ' αυτό το συγκεκριμένο παράδειγμα, θα αρκούσε επίσης η `always @(a)`.

Η λειτουργία του τελεστή = στην εντολή `always` αποκαλείται *κλειδωμένη ανάθεση* (blocking assignment), σε αντίθεση με τον τελεστή <= που εκτελεί μη-κλειδωμένη ανάθεση. Στην SystemVerilog, μια καλή πρακτική είναι να χρησιμοποιείτε τελεστές κλειδωμένης ανάθεσης για τα κυκλώματα συνδυαστικής λογικής και μη-κλειδωμένης ανάθεσης για τα κυκλώματα ακολουθιακής λογικής. Θα εξετάσουμε αναλυτικά αυτό το θέμα στην Ενότητα A.5.4.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity inv is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture proc of inv is
begin
  process(a) begin
    y <= not a;
  end process;
end;
```

Οι εντολές `begin` και `end process` είναι απαιτούμενες στην VHDL, παρόλο που η `process` περιέχει μόνο μία ανάθεση.

Οι γλώσσες HDL υποστηρίζουν πράξεις *κλειδωμένης (blocking)* και *μη-κλειδωμένης (nonblocking) ανάθεσης* σε μια εντολή always/process. Μια ομάδα πράξεων κλειδωμένης ανάθεσης αποτιμώνται με τη σειρά με την οποία εμφανίζονται στον κώδικα, ακριβώς όπως θα περίμενε κανείς από μια τυπική γλώσσα προγραμματισμού. Μια ομάδα πράξεων μη-κλειδωμένης ανάθεσης αποτιμώνται ταυτόχρονα· όλες οι εκφράσεις της δεξιάς πλευράς αποτιμώνται πριν ενημερωθούν οι εκφράσεις της αριστερής πλευράς. Για λόγους τους οποίους θα δούμε στην Ενότητα A.5.4, είναι αποτελεσματικότερο να χρησιμοποιείτε κλειδωμένη ανάθεση για τη συνδυαστική λογική και ασφαλέστερο να χρησιμοποιείτε μη-κλειδωμένη ανάθεση για την ακολουθιακή λογική.

SystemVerilog

Σε μια εντολή `always`, ο τελεστής = υποδεικνύει κλειδωμένη ανάθεση, ενώ ο τελεστής <= υποδεικνύει μη-κλειδωμένη ανάθεση.

Δεν θα πρέπει να συγχέετε αυτούς τους δύο τύπους ανάθεσης με τη συνεχή ανάθεση που εκτελεί η `assign`. Κανονικά, οι εντολές `assign` χρησιμοποιούνται έξω από τις εντολές `always` και αποτιμώνται επίσης ταυτόχρονα.

VHDL

Σε μια εντολή `process`, ο τελεστής := υποδεικνύει κλειδωμένη ανάθεση, ενώ ο τελεστής <= υποδεικνύει μη-κλειδωμένη ανάθεση (αποκαλείται επίσης ταυτόχρονη ανάθεση). Αυτή είναι η πρώτη ενότητα όπου χρησιμοποιείται ο τελεστής :=.

Οι πράξεις μη-κλειδωμένης ανάθεσης εκτελούνται για εξόδους και σήματα. Οι πράξεις κλειδωμένης ανάθεση εκτελούνται για μεταβλητές, οι οποίες δηλώνονται σε εντολές `process` (βλ. το επόμενο παράδειγμα). Ο τελεστής <= μπορεί επίσης να εμφανίζεται έξω από μία εντολή `process`, οπότε και αποτιμάται ταυτόχρονα.

Το Παράδειγμα A.29 ορίζει έναν πλήρη αθροιστή, χρησιμοποιώντας τα ενδιάμεσα σήματα `p` και `g` για τον υπολογισμό των `s` και `cout`. Παράγει το ίδιο κύκλωμα με το Σχήμα A.9, αλλά χρησιμοποιεί εντολές `always/process` στη θέση των εντολών ανάθεσης.

Παράδειγμα A.29 Πλήρης αθροιστής (με χρήση εντολών `always/process`)

SystemVerilog

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);

  logic p, g;

  always_comb
    begin
      p = a ^ b;  // blocking
      g = a & b;  // blocking

      s = p ^ cin;
      cout = g | (p & cin);
    end
endmodule
```

Σ' αυτή την περίπτωση, η `always @(a, b, cin)` ή η `always @(*)` θα ήταν ισοδύναμες με την `always_comb`. Και οι τρεις αποτιμούν εκ νέου ρα περιεχόμενα του μπλοκ always οποτεδήποτε αλλάζουν τα a, b, ή `cin`. Ωστόσο, η `always_comb` προτιμάται επειδή είναι πιο λακωνική και επιτρέπει στα εργαλεία της SystemVerilog να παράγουν ένα προειδοποιητικό μήνυμα εάν το μπλοκ περιγράφει κατά λάθος ακολουθιακή λογική.

Σημειώστε ότι η δομή `begin / end` είναι αναγκαία, επειδή περιλαμβάνονται πολλαπλές εντολές μέσα στο μπλοκ της `always`. Αυτή η πρακτική είναι ανάλογη με τη χρήση των άγκιστρων ({ }) στην C ή την Java. Η δομή `begin / end` δεν ήταν αναγκαία στο παράδειγμα της μονάδας flopr, επειδή η `if / else` θεωρείται μία εντολή.

Το παράδειγμα αυτό χρησιμοποιεί κλειδωμένη ανάθεση, υπολογίζοντας πρώτα το p, κατόπιν το g, στη συνέχεια το s και, τέλος, το `cout`.

VHDL

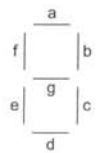
```
library IEEE; use IEEE.STD_LOGIC_1164.all;

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

architecture synth of fulladder is
begin
  process (a, b, cin)
    variable p, g: STD_LOGIC;
  begin
    p := a xor b; -- blocking
    g := a and b; -- blocking

    s <= p xor cin;
    cout <= g or (p and cin);
  end process;
end;
```

Η λίστα ευαισθησίας της `process` πρέπει να περιλαμβάνει τα a, b και `cin`, επειδή τα κυκλώματα συνδυαστικής λογικής θα πρέπει να ανταποκρίνονται στις αλλαγές οποιασδήποτε εισόδου. Εάν παραληφθεί οποιαδήποτε από αυτές τις εισόδους, ο κώδικας θα μπορούσε να καταλήξει σε ακολουθιακή λογική ή να συμπεριφέρεται διαφορετικά στην προσομοίωση και τη σύνθεση.

Το παράδειγμα αυτό χρησιμοποιεί κλειδωμένη ανάθεση για τα p και g, έτσι ώστε να λαμβάνουν τις νέες τιμές τους πριν χρησιμοποιηθούν για τον υπολογισμό των s και `cout`, τα οποία εξαρτώνται από τα p και g.

Επειδή τα p και g εμφανίζονται στην αριστερή πλευρά μιας έκφρασης κλειδωμένης ανάθεσης (:=) σε μία εντολή `process`, πρέπει να δηλωθούν ως μεταβλητές (`variable`) και όχι ως σήματα (`signal`). Η δήλωση μιας μεταβλητής εμφανίζεται πριν από την begin στην εντολή `process` όπου χρησιμοποιείται η μεταβλητή.

Στην πραγματικότητα, τα δύο παραπάνω παραδείγματα αποτελούν αντιπαραδείγματα εφαρμογής των εντολών `always/process` για τη μοντελοποίηση συνδυαστικής λογικής, επειδή απαιτούν περισσότερες γραμμές απ' ό,τι η ισοδύναμη προσέγγιση με εντολές `assign` (βλ. Ενότητα A.2.1). Επιπλέον, ενέχουν τον κίνδυνο να οδηγήσουν κατά λάθος σε ακολουθιακή λογική, εάν παραληφθούν κάποιες είσοδοι από την λίστα ευαισθησίας. Ωστόσο, οι εντολές `case` και `if` είναι βολικές για την μοντελοποίηση πολυπλοκότερης συνδυαστικής λογικής. Οι εντολές `case` και `if` μπορούν να εμφανίζονται μόνο μέσα σε εντολές `always/process`.



ΣΧΗΜΑ A.26 Οθόνη 7 ενδείξεων.

A.5.1 Εντολές Case

Μια καλύτερη εφαρμογή των εντολών `always/process` για κυκλώματα συνδυαστικής λογικής είναι το ακόλουθο παράδειγμα ενός αποκωδικοποιητή με οθόνη 7 ενδείξεων. Το παράδειγμα αυτό αξιοποιεί τις δυνατότητες της εντολής `case`, η οποία πρέπει να εμφανίζεται μέσα σε μια εντολή `always/process`.

Η διαδικασία σχεδιασμού για την περιγραφή μεγάλων μπλοκ συνδυαστικής λογικής με Boolean εκφράσεις είναι κοπιαστική και επιρρεπής σε σφάλματα. Οι γλώσσες HDL παρέχουν σημαντική ευκολία σ' αυτό τον τομέα, επιτρέποντάς σας να καθορίσετε τη συνάρτηση που υλοποιεί το μπλοκ σ' ένα υψηλότερο επίπεδο αφαίρεσης και κατόπιν συνθέτουν αυτόματα τη συνάρτηση σε επίπεδο πυλών. Το Παράδειγμα A.30 χρησιμοποιεί εντολές `case` για να περιγράψει μια οθόνη 7 ενδείξεων για έναν αποκωδικοποιητή, βάσει του πίνακας αληθείας του. Μια σχηματική αναπαράσταση της οθόνης 7 ενδείξεων απεικονίζεται στο Σχήμα A.26. Ο αποκωδικοποιητής δέχεται έναν 4-bit αριθμό και εμφανίζει τη δεκαδική τιμή του αναβοσβήνοντας τις κατάλληλες ενδείξεις στην οθόνη. Για παράδειγμα, ο αριθμός 0111 = 7 θα πρέπει να ανάψει τις ενδείξεις *a*, *b* και *c*.

Η εντολή `case` εκτελεί διαφορετικές ενέργειες, ανάλογα με την τιμή της εισόδου της. Μια εντολή `case` υποδηλώνει συνδυαστική λογική εάν συνυπολογίζονται όλοι οι πιθανοί συνδυασμοί εισόδων· διαφορετικά, υποδηλώνει ακολουθιακή λογική, επειδή η έξοδος θα διατηρήσει την παλαιά τιμή της στις «απροσδιόριστες» περιπτώσεις (`case`).

Παράδειγμα A.30 Οθόνη 7 Ενδείξεων για Έναν Αποκωδικοποιητή

SystemVerilog

```
module sevenseg(input  logic [3:0] data,
                output logic [6:0] segments);

  always_comb
    case (data)
      //               abc_defg
      0: segments = 7'b111_1110;
      1: segments = 7'b011_0000;
      2: segments = 7'b110_1101;
      3: segments = 7'b111_1001;
      4: segments = 7'b011_0011;
      5: segments = 7'b101_1011;
      6: segments = 7'b101_1111;
      7: segments = 7'b111_0000;
      8: segments = 7'b111_1111;
      9: segments = 7'b111_1011;
      default: segments = 7'b000_0000;
    endcase
endmodule
```

Ο όρος `default` αποτελεί ένα βολικό τρόπο για τον ορισμό της εξόδου σε όλες εκείνες τις περιπτώσεις που δεν κατονομάζονται ρητά στην εντολή `case`, διασφαλίζοντας έτσι ότι το αποτέλεσμα θα είναι πάντα συνδυαστική λογική.

Στην SystemVerilog, οι εντολές `case` πρέπει να εμφανίζονται μέσα σε εντολές `always`.

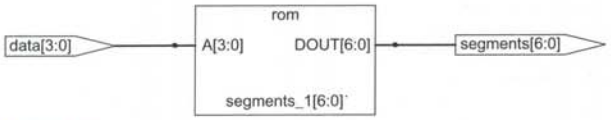
VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity seven_seg_decoder is
  port(data:  in  STD_LOGIC_VECTOR(3 downto 0);
       segments: out STD_LOGIC_VECTOR(6 downto 0));
end;

architecture synth of seven_seg_decoder is
begin
  process(data) begin
    case data is
--                              abcdefg
      when X"0" => segments <= "1111110";
      when X"1" => segments <= "0110000";
      when X"2" => segments <= "1101101";
      when X"3" => segments <= "1111001";
      when X"4" => segments <= "0110011";
      when X"5" => segments <= "1011011";
      when X"6" => segments <= "1011111";
      when X"7" => segments <= "1110000";
      when X"8" => segments <= "1111111";
      when X"9" => segments <= "1111011";
      when others => segments <= "0000000";
    end case;
  end process;
end;
```

Η εντολή `case` ελέγχει την τιμή του ορίσματος `data`. Όταν το `data` είναι 0, η εντολή εκτελεί την ενέργεια που κατονομάζεται μετά το =>, θέτοντας την `segments` σε 1111110. Κατά παρόμοιο τρόπο, η `case` ελέγχει τις άλλες τιμές `data` έως την 9 (παρατηρήστε τη χρήση του X για τους δεκαεξαδικούς αριθμούς). Ο όρος `others` αποτελεί ένα βολικό τρόπο ορισμού της εξόδου για όλες εκείνες τις περιπτώσεις που δεν κατονομάζονται ρητά στην `case`, διασφαλίζοντας έτσι ότι το αποτέλεσμα θα είναι πάντα συνδυαστική λογική.

Ανόμοια με την Verilog, η VHDL υποστηρίζει εντολές ανάθεσης επιλεγμένου σήματος (βλ. Ενότητα A.2.4), οι οποίες μοιάζουν αρκετά με τις εντολές `case`, αλλά μπορούν να εμφανίζονται έξω από εντολές `process`. Συνεπώς, έχετε έναν λιγότερο λόγο να χρησιμοποιείτε εντολές `process` για να περιγράφετε συνδυαστική λογική.

Το Synplify Pro συνθέτει την οθόνη 7 ενδείξεων του αποκωδικοποιητή σε μια μνήμη ROM, η οποία περιέχει τις επτά εξόδους για κάθε μία από τις 16 πιθανές εισόδους. Άλλα εργαλεία ενδεχομένως να παρήγαγαν μεγάλο αριθμό πυλών.



ΣΧΗΜΑ Α.27 Η μονάδα sevenseg.

Εάν παραλείπονταν ο όρος `default` ή others από την εντολή `case`, ο αποκωδικοποιητής θα θυμούνταν την προηγούμενη έξοδό του οποτεδήποτε η `data` θα ήταν στο εύρος τιμών 10–15. Η συμπεριφορά αυτή είναι ιδιόμορφη και δεν συνάδει με τη συνδυαστική λογική.

Η εντολή `case` χρησιμοποιείται ευρέως για την περιγραφή συμβατικών αποκωδικοποιητών. Το Παράδειγμα Α.31 περιγράφει έναν αποκωδικοποιητή 3:8.

Παράδειγμα Α.31 Αποκωδικοποιητής 3:8

SystemVerilog

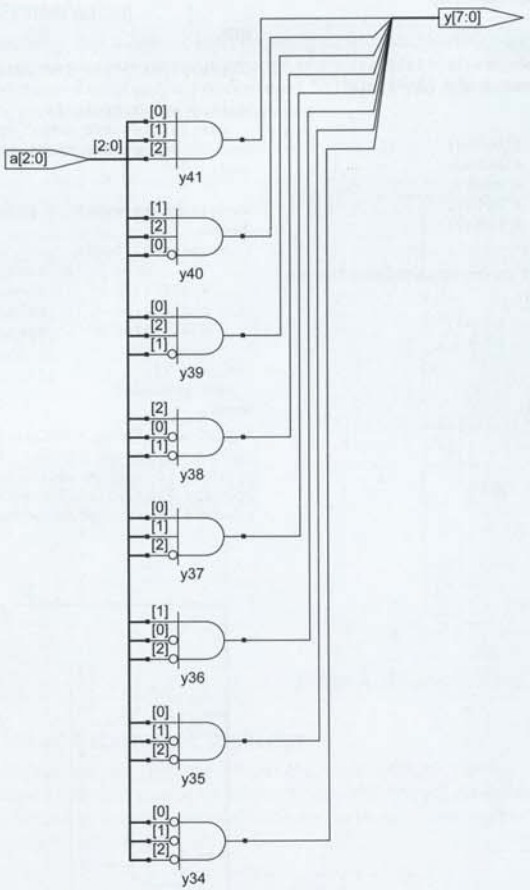
```
module decoder3_8(input  logic [2:0] a,
                  output logic [7:0] y);

  always_comb
    case (a)
      3'b000: y = 8'b00000001;
      3'b001: y = 8'b00000010;
      3'b010: y = 8'b00000100;
      3'b011: y = 8'b00001000;
      3'b100: y = 8'b00010000;
      3'b101: y = 8'b00100000;
      3'b110: y = 8'b01000000;
      3'b111: y = 8'b10000000;
    endcase
endmodule
```

Σ' αυτή την περίπτωση δεν απαιτείται όρος `default`, επειδή καλύπτονται ρητά όλες οι περιπτώσεις.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity decoder3_8 is
  port(a: in  STD_LOGIC_VECTOR(2 downto 0);
       y: out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture synth of decoder3_8 is
begin
  process(a) begin
    case a is
      when "000"  => y <= "00000001";
      when "001"  => y <= "00000010";
      when "010"  => y <= "00000100";
      when "011"  => y <= "00001000";
      when "100"  => y <= "00010000";
      when "101"  => y <= "00100000";
      when "110"  => y <= "01000000";
      when "111"  => y <= "10000000";
      when others => y <= (OTHERS => 'X');
    end case;
  end process;
end;
```

Ορισμένα εργαλεία VHDL απαιτούν έναν όρο `others` επειδή δεν καλύπτονται συνδυασμοί εισόδων όπως π.χ. ο «`1zx`». Η έκφραση `y <= (OTHERS => 'X')` θέτει όλα τα bit της εξόδου `y` σε X· αυτή είναι μια εντελώς διαφορετική χρήση της δεσμευμένης λέξης OTHERS.



ΣΧΗΜΑ Α.28 Αποκωδικοποιητής 3:8.

Α.5.2 Εντολές If

Οι εντολές `always/process` μπορούν επίσης να περιέχουν εντολές `if` στον κορμό τους. Μια εντολή `if` μπορεί να ακολουθείται από μια εντολή `else`. Στις περιπτώσεις όπου υπάρχει μέριμνα για όλους τους πιθανούς συνδυασμούς εισόδων, η εντολή `if` υποδηλώνει συνδυαστική λογική· διαφορετικά, παράγει ακολουθιακή λογική (όπως ο μανδαλωτής της Ενότητας Α.4.5).

Το Παράδειγμα Α.32 χρησιμοποιεί εντολές `if` για την περιγραφή ενός κυκλώματος προτεραιότητας των 4 bit, το οποίο θέτει μία έξοδο σε τιμή TRUE ανάλογα με την περισσότερο σημαντική είσοδο που είναι TRUE.

Παράδειγμα A.32 Κύκλωμα προτεραιότητας

SystemVerilog

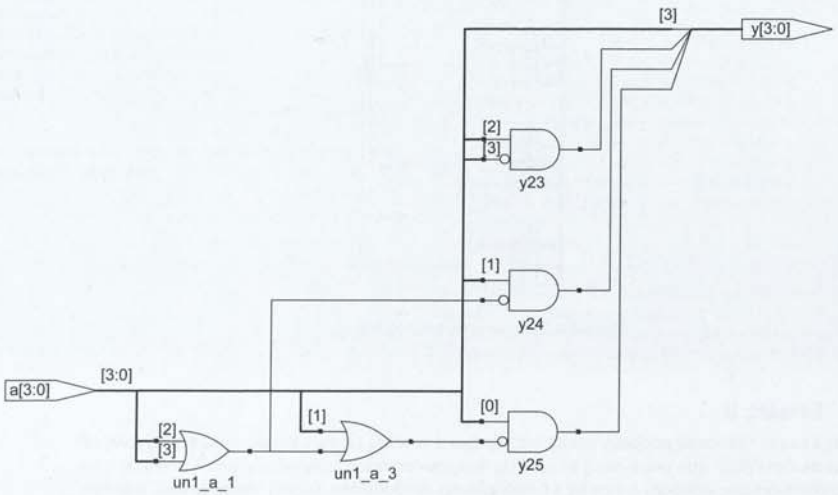
```
module priorityckt(input  logic [3:0] a,
                   output logic [3:0] y);

  always_comb
    if      (a[3]) y = 4'b1000;
    else if (a[2]) y = 4'b0100;
    else if (a[1]) y = 4'b0010;
    else if (a[0]) y = 4'b0001;
    else           y = 4'b0000;
endmodule
```

Στην SystemVerilog, οι εντολές `if` πρέπει να εμφανίζονται στον κορμό μιας εντολής `always`.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity priorityckt is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of priorityckt is
begin
  process(a) begin
    if    a(3) = '1' then y <= "1000";
    elsif a(2) = '1' then y <= "0100";
    elsif a(1) = '1' then y <= "0010";
    elsif a(0) = '1' then y <= "0001";
    else                  y <= "0000";
    end if;
  end process;
end;
```

Ανόμοια με την Verilog, η VHDL υποστηρίζει εντολές υπό συνθήκη ανάθεσης σήματος (βλ. Ενότητα A.2.4), οι οποίες είναι παρόμοιες με τις εντολές `if` αλλά μπορούν να εμφανίζονται έξω από μία εντολή `process`. Συνεπώς, έχετε έναν λιγότερο λόγο να χρησιμοποιείτε την `process` για να περιγράψετε συνδυαστική λογική.



ΣΧΗΜΑ A.29 Κύκλωμα προτεραιότητας.

A.5.3 Η Casez της SystemVerilog

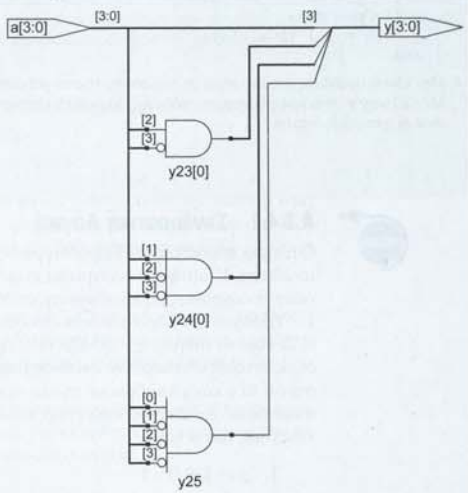
(Οι χρήστες της VHDL μπορούν να παρακάμψουν αυτή την ενότητα). Η SystemVerilog παρέχει επίσης την εντολή `casez`, η οποία χρησιμοποιείται για την περιγραφή πινάκων αληθείας που περιλαμβάνουν αδιάφορες τιμές (υποδεικνύονται με το χαρακτήρα `?` στην `casez`). Το Παράδειγμα A.33 παρουσιάζει την περιγραφή ενός κυκλώματος προτεραιότητας με την εντολή `casez`.

Παράδειγμα A.33 Κύκλωμα προτεραιότητας με χρήση της `casez`

SystemVerilog

```
module priority_casez(input  logic [3:0] a,
                      output logic [3:0] y);

  always_comb
    casez(a)
      4'b1???: y = 4'b1000;
      4'b01??: y = 4'b0100;
      4'b001?: y = 4'b0010;
      4'b0001: y = 4'b0001;
      default: y = 4'b0000;
    endcase
endmodule
```

Το Synplify Pro συνθέτει ένα ελαφρώς διαφορετικό κύκλωμα γι' αυτή τη λειτουργική μονάδα (βλ. Σχήμα A.30), σε σύγκριση με το κύκλωμα προτεραιότητας του Σχήματος A.29. Ωστόσο, τα δύο κυκλώματα είναι λογικά ισοδύναμα.



ΣΧΗΜΑ A.30 Η μονάδα priority_casez.

A.5.4 Κλειδωμένες και μη-Κλειδωμένες Αναθέσεις

Οι ακόλουθες οδηγίες εξηγούν πότε και πώς χρησιμοποιείται κάθε τύπος ανάθεσης. Εάν δεν τις ακολουθήσετε, είναι πιθανό να γράψετε κώδικα ο οποίος θα δείχνει να δουλεύει στην προσομοίωση, αλλά η σύνθεσή του θα δίνει λάθος hardware. Στο (προαιρετικό) υπόλοιπο αυτής της ενότητας θα εξηγήσουμε τις αρχές στις οποίες βασίζονται αυτές οι οδηγίες.

SystemVerilog

1. Να χρησιμοποιείτε την `always_ff @(posedge clk)` και μη-κλειδωμένες (nonblocking) αναθέσεις για τη μοντελοποίηση σύγχρονης ακολουθιακής λογικής.

```
always_ff @(posedge clk)
  begin
    n1 <= d; // nonblocking
    q <= n1; // nonblocking
  end
```

2. Να χρησιμοποιείτε εντολές συνεχούς ανάθεσης για τη μοντελοποίηση απλής συνδυαστικής λογικής.

```
assign y = s ? d1 : d0;
```

VHDL

1. Να χρησιμοποιείτε την `process(clk)` και μη-κλειδωμένες (nonblocking) αναθέσεις για τη μοντελοποίηση σύγχρονης ακολουθιακής λογικής.

```
process(clk) begin
  if clk'event and clk = '1' then
    n1 <= d; -- nonblocking
    q <= n1; -- nonblocking
  end if;
end process;
```

2. Να χρησιμοποιείτε εντολές ταυτόχρονης ανάθεσης έξω από τον κορμό των εντολών `process` για τη μοντελοποίηση απλής συνδυαστικής λογικής.

```
y <= d0 when s = '0' else d1;
```

SystemVerilog (συνέχεια)

3. Να χρησιμοποιείτε την `always_comb` και κλειδωμένες (blocking) αναθέσεις για τη μοντελοποίηση πολυπλοκότερης συνδυαστικής λογικής, όπου η `always` μπορεί να φανεί χρήσιμη.

```
always_comb
  begin
    p = a ^ b;  // blocking
    g = a & b;  // blocking
    s = p ^ cin;
    cout = g | (p & cin);
  end
```

4. Μην κάνετε αναθέσεις στο ίδιο σήμα σε περισσότερες από μία εντολές `always` ή εντολές συνεχούς ανάθεσης. Μοναδική εξαίρεση είναι οι τρισταθείς δίαυλοι.

VHDL (συνέχεια)

3. Να χρησιμοποιείτε την `process(in1, in2, ...)` για τη μοντελοποίηση πολυπλοκότερης συνδυαστικής λογικής, όπου η `process` μπορεί να φανεί χρήσιμη.
   Να χρησιμοποιείτε κλειδωμένες (blocking) αναθέσεις για τις εσωτερικές μεταβλητές.

```
process(a, b, cin)
   variable p, g: STD_LOGIC;
 begin
   p := a xor b; -- blocking
   g := a and b; -- blocking
   s <= p xor cin;
   cout <= g or (p and cin);
 end process;
```

4. Μην κάνετε αναθέσεις στην ίδια μεταβλητή σε περισσότερες από μία εντολές `process` ή εντολές ταυτόχρονης ανάθεσης. Μοναδική εξαίρεση είναι οι τρισταθείς δίαυλοι.

A.5.4.1 Συνδυαστική Λογική

Ο πλήρης αθροιστής του Παραδείγματος Α.29 μοντελοποιείται σωστά με τη χρήση εκφράσεων κλειδωμένης ανάθεσης. Σ' αυτή την ενότητα θα διερευνήσουμε πώς λειτουργεί και πώς θα διέφερε εάν είχαν χρησιμοποιηθεί εκφράσεις μη-κλειδωμένης ανάθεσης.

Υποθέστε ότι αρχικά όλες οι είσοδοι, `a`, `b` και `cin` είναι 0. Συνεπώς, τα p, g, s και cout είναι επίσης 0. Σε κάποια στιγμή, η a αλλάζει σε 1, προκαλώντας την εκτέλεση της εντολής `always/process`. Οι τέσσερις εντολές κλειδωμένης ανάθεσης αποτιμώνται με τη σειρά που παρουσιάζεται στη συνέχεια. Σημειώστε ότι τα p και g λαμβάνουν τη νέα τιμή τους πριν υπολογιστούν τα s και cout, λόγω των κλειδωμένων αναθέσεων. Αυτό είναι σημαντικό, επειδή θέλουμε τα s και cout να υπολογιστούν χρησιμοποιώντας τις νέες τιμές των p και g.

1. $p \leftarrow 1 \oplus 0 = 1$
2. $g \leftarrow 1 \cdot 0 = 0$
3. $s \leftarrow 1 \oplus 0 = 1$
4. $cout \leftarrow 0 + 1 \cdot 0 = 0$

Το Παράδειγμα Α.34 παρουσιάζει τη χρήση εντολών μη-κλειδωμένης ανάθεσης, μια πρακτική η οποία δεν συνιστάται.

Παράδειγμα Α.34 Πλήρης Αθροιστής, με Χρήση Εντολών Μη-Κλειδωμένης Ανάθεσης

SystemVerilog

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);

  logic p, g;

  always_comb
    begin
      p <= a ^ b; // nonblocking
      g <= a & b; // nonblocking

      s <= p ^ cin;
      cout <= g | (p & cin);
    end
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

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

architecture nonblocking of fulladder is
  signal p, g: STD_LOGIC;
begin
  process (a, b, cin, p, g) begin
    p <= a xor b; -- nonblocking
    g <= a and b; -- nonblocking
```


VHDL (συνέχεια)

```
    s <= p xor cin;
    cout <= g or (p and cin);
  end process;
end;
```

Επειδή τα p και g εμφανίζονται στην αριστερή πλευρά μιας μη-κλειδωμένης ανάθεσης σε μια εντολή `process`, θα πρέπει να είναι δηλωμένα ως `signal` και όχι ως `variable`. Η δήλωσή τους ως `signal` εμφανίζεται πριν από το μπλοκ `begin` στην ενότητα `architecture` και όχι στην `process`.

Ας δούμε την ίδια περίπτωση ενός ανοδικού σήματος από 0 σε 1 ενόσω τα b και cin είναι 0. Οι τέσσερις μη-κλειδωμένες αναθέσεις αποτιμώνται ταυτόχρονα, ως εξής:

$p \leftarrow 1 \oplus 0 = 1 \quad g \leftarrow 1 \cdot 0 = 0 \quad s \leftarrow 0 \oplus 0 = 0 \quad cout \leftarrow 0 + 0 \cdot 0 = 0$

Παρατηρήστε ότι, επειδή το s υπολογίζεται ταυτόχρονα με το pand, χρησιμοποιεί την παλαιά τιμή του p και όχι τη νέα τιμή του. Συνεπώς, το s παραμένει 0 αντί να γίνει 1. Ωστόσο, το p αλλάζει από 0 σε 1. Αυτή η αλλαγή προκαλεί τον υπολογισμό της `always/process` για δεύτερη φορά, ως εξής:

$p \leftarrow 1 \oplus 0 = 1 \quad g \leftarrow 1 \cdot 0 = 0 \quad s \leftarrow 1 \oplus 0 = 1 \quad cout \leftarrow 0 + 1 \cdot 0 = 0$

Αυτή τη φορά, το p ήταν ήδη 1, οπότε το s σωστά μεταβάλλεται σε 1. Οι μη-κλειδωμένες αναθέσεις έφτασαν τελικά στην απάντηση της δεξιάς πλευράς, αλλά η εντολή `always/process` έπρεπε να αποτιμηθεί δύο φορές. Αυτό καθιστά την προσομοίωση πιο χρονοβόρα, παρότι η σύνθεση καταλήγει στο ίδιο hardware.

Ένα άλλο μειονέκτημα των μη-κλειδωμένων αναθέσεων κατά τη μοντελοποίηση συνδυαστικής λογικής είναι ότι η περιγραφή HDL θα παράγει λανθασμένο αποτέλεσμα εάν ξεχάσετε να συμπεριλάβετε τις ενδιάμεσες μεταβλητές στη λίστα ευαισθησίας, όπως παρουσιάζεται στη συνέχεια.

SystemVerilog

Εάν η λίστα ευαισθησίας της εντολής `always` είχε διατυπωθεί ως `always @(a, b, cin)` και όχι ως `always_comb` ή `always @(*)`, τότε η εντολή δεν θα αποτιμούνταν για δεύτερη φορά όταν θα άλλαξε το p ή το g. Στο προηγούμενο παράδειγμα, το s θα παρέμενε σε 0 και δεν θα λάμβανε την τιμή 1, κάτι το οποίο είναι λάθος.

VHDL

Εάν η λίστα ευαισθησίας της εντολής `process` είχε διατυπωθεί ως `process (a, b, cin)` και όχι ως `process (a, b, cin, p, g)`, τότε η εντολή δεν θα αποτιμούνταν για δεύτερη φορά όταν θα άλλαζε το p ή το g. Στο προηγούμενο παράδειγμα, το s θα παρέμενε σε 0 και δεν θα λάμβανε την τιμή 1, κάτι το οποίο είναι λάθος.

Ακόμα χειρότερα, σε μια τέτοια περίπτωση ορισμένα εργαλεία σύνθεσης θα συνθέσουν το σωστό hardware ακόμα κι όταν μια εσφαλμένη λίστα ευαισθησίας προκαλεί λανθασμένη προσομοίωση. Αυτό οδηγεί σε αναντιστοιχία μεταξύ των αποτελεσμάτων της προσομοίωσης και της λειτουργίας που επιτελεί πραγματικά το hardware.

A.5.4.2 Ακολουθιακή Λογική

Ο συγχρονιστής από το Παράδειγμα Α.23 μοντελοποιείται σωστά χρησιμοποιώντας μη-κλειδωμένες αναθέσεις. Στην ανοδική ακμή του ρολογιού, το d αντιγράφεται στο n1 την ίδια στιγμή που το n1 αντιγράφεται στο q, οπότε ο κώδικας περιγράφει σωστά δύο καταχωρητές. Για παράδειγμα, υποθέστε ότι αρχικά d = 0, n1 = 1 και q = 0. Στην ανοδική ακμή του ρολογιού, οι δύο ακόλουθες αναθέσεις συμβαίνουν ταυτόχρονα, οπότε μετά την ακμή του ρολογιού έχουμε n1 = 0 και q = 1.

$$n1 \leftarrow d = 0 \quad q \leftarrow n1 = 1$$

Το Παράδειγμα Α.35 εσφαλμένα επιχειρεί να περιγράψει την ίδια λειτουργική μονάδα χρησιμοποιώντας κλειδωμένες αναθέσεις. Στην ανοδική ακμή του `clk`, το `d` αντιγράφεται στο `n1`. Αυτή η νέα τιμή του `n1` αντιγράφεται κατόπιν στο `q`, με αποτέλεσμα η τιμή του `d` να εμφανίζεται εσφαλμένα τόσο στο `n1` όσο και στο `q`. Εάν `d`= 0 και `n1`= 1, τότε μετά από την ακμή του ρολογιού θα έχουμε `n1` = `q` = 0.

1. `n1` ← `d` = 0
2. `q` ← `n1` = 0

Επειδή το `n1` είναι αόρατο για τον έξω κόσμο και δεν επηρεάζει τη συμπεριφορά του `q`, το εργαλείο σύνθεσης το βελτιστοποιεί ολοκληρωτικά, όπως υποδεικνύει το Σχήμα Α.31.

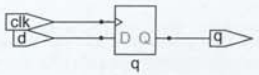
Παράδειγμα Α.35 Προβληματική Υλοποίηση Συγχρονιστή με Εντολές Κλειδωμένης (Blocking) Ανάθεσης

SystemVerilog

```
module syncbad(input  logic clk,
               input  logic d,
               output logic q);

  logic n1;

  always_ff @(posedge clk)
    begin
      n1 = d; // blocking
      q = n1; // blocking
    end
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity syncbad is
  port(clk: in  STD_LOGIC;
       d:   in  STD_LOGIC;
       q:   out STD_LOGIC);
end;

architecture bad of syncbad is
begin
  process(clk)
    variable n1: STD_LOGIC;
  begin
    if clk'event and clk = '1' then
      n1 := d;  -- blocking
      q <= n1;
    end if;
  end process;
end;
```

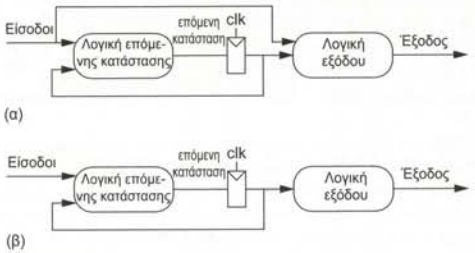


ΣΧΗΜΑ Α.31 Η μονάδα syncbad.

Το ηθικό δίδαγμα αυτού του παραδείγματος είναι απλό: θα πρέπει να χρησιμοποιείτε αποκλειστικά μη-κλειδωμένη ανάθεση σε εντολές `always` όταν μοντελοποιείτε ακολουθιακή λογική. Με έξυπνες τεχνικές, όπως η αντιστροφή της σειράς των αναθέσεων, θα μπορούσατε να κάνετε τις κλειδωμένες αναθέσεις να δουλεύουν σωστά· ωστόσο, δεν σας παρέχουν κανένα πλεονέκτημα και απλώς εισάγουν τον κίνδυνο απρόβλεπτης συμπεριφοράς. Συγκεκριμένα ακολουθιακά κυκλώματα δεν πρόκειται να δουλέψουν καθόλου όταν χρησιμοποιούνται κλειδωμένες αναθέσεις, ανεξαρτήτως της σειράς τους.

Α.6 Μηχανές Πεπερασμένων Καταστάσεων

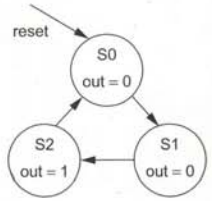
Υπάρχουν δύο στιλ μηχανών πεπερασμένων καταστάσεων (FSM). Στις *μηχανές Mealy* (Σχήμα Α.32(α)), η έξοδος είναι συνάρτηση της τρέχουσας κατάστασης και των εισόδων. Στις *μηχανές Moore* (Σχήμα Α.32(β)), η έξοδος είναι συνάρτηση μόνο της τρέχουσας κατάστασης. Για αμφότερους τους τύπους, η FSM μπορεί να διαμεριστεί σ' έναν καταχωρητή κατάστασης, τη λογική για την επόμενη κατάσταση και τη λογική εξόδου. Οι HDL περιγραφές των μηχανών καταστάσεων διαχωρίζονται, αντίστοιχα, στα ίδια τρία/τρεις μέρη.



ΣΧΗΜΑ Α.32 Μηχανές πεπερασμένων καταστάσεων Mealy και Moore.

Α.6.1 Παράδειγμα FSM

Το Παράδειγμα Α.36 περιγράφει τη μηχανή πεπερασμένων καταστάσεων «διαίρεσης διά 3» από το Σχήμα Α.33. Παρέχει ένα σύγχρονο σήμα reset για την αρχικοποίηση της FSM. Ο καταχωρητής κατάστασης χρησιμοποιεί τη συνηθισμένη σύνταξη για τα flip-flop. Τα μπλοκ για την επόμενη κατάσταση και τη λογική εξόδου είναι συνδυαστικής λογικής. Αυτή η FSM είναι ένα παράδειγμα μηχανής Moore· στην πραγματικότητα, δεν έχει εισόδους παρά μόνο ένα σήμα ρολογιού κι ένα σήμα reset.



ΣΧΗΜΑ Α.33 Διάγραμμα μετάβασης καταστάσεων για το μετρητή της «διαίρεσης διά 3».

Παράδειγμα Α.36 Μηχανή πεπερασμένων καταστάσεων για τη «διαίρεση διά 3»

SystemVerilog

```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic y);

  logic [1:0] state, nextstate;

  // Καταχωρητής κατάστασης
  always_ff @(posedge clk)
    if (reset) state <= 2'b00;
    else       state <= nextstate;

  // Λογική επόμενης κατάστασης
  always_comb
    case (state)
      2'b00: nextstate = 2'b01;
      2'b01: nextstate = 2'b10;
      2'b10: nextstate = 2'b00;
      default: nextstate = 2'b00;
    endcase
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity divideby3FSM is
  port(clk, reset: in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of divideby3FSM is
  signal state, nextstate:
    STD_LOGIC_VECTOR(1 downto 0);
begin
  -- καταχωρητής κατάστασης
  process(clk) begin
    if clk'event and clk = '1' then
      if reset = '1' then state <= "00";
      else state <= nextstate;
      end if;
    end if;
  end process;
```

(συνεχίζεται)

SystemVerilog (συνέχεια)

```
// λογική εξόδου
  assign y = (state == 2'b00);
endmodule
```

Παρατηρήστε ότι χρησιμοποιείται μια εντολή `case` για να ορίσει τον πίνακα μετάβασης καταστάσεων. Επειδή η λογική για την επόμενη κατάσταση θα πρέπει να είναι συνδυαστική, η `default` είναι αναγκαία ακόμα κι αν η κατάσταση 11 δεν πρόκειται να προκύψει ποτέ.

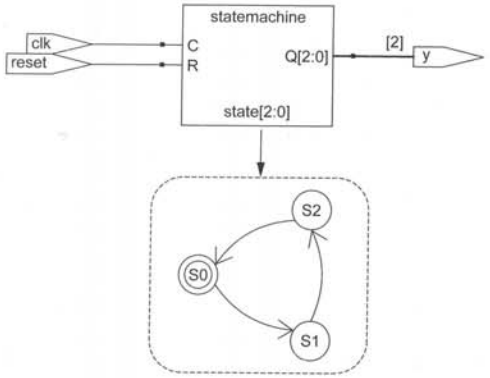
Η έξοδος `y` είναι 1 όταν η κατάσταση είναι 00. Η σύγκριση ισότητας `a == b` αποτιμάται σε 1 εάν το `a` ισούται με το `b` και διαφορετικά σε 0. Η σύγκριση ανισότητας `a != b` κάνει το αντίθετο, δίνοντας 1 εάν το `a` δεν ισούται με το `b`.

VHDL (συνέχεια)

```
-- λογική επόμενης κατάστασης
 nextstate <= "01" when state = "00" else
              "10" when state = "01" else
              "00";

 -- λογική εξόδου
 y <= '1' when state = "00" else '0';
end;
```

Η έξοδος `y` είναι 1 όταν η κατάσταση είναι 00. Η σύγκριση ισότητας `a = b` αποτιμάται σε `true` εάν το `a` ισούται με το `b` και διαφορετικά σε `false`. Η σύγκριση ανισότητας `a /= b` κάνει το αντίθετο, δίνοντας `true` εάν το `a` δεν ισούται με το `b`.

Το Synplify Pro παράγει απλώς ένα διάγραμμα βαθμίδων κι ένα διάγραμμα μετάβασης καταστάσεων για τις μηχανές κατάστασης· δεν υποδεικνύει τις λογικές πύλες ή τις εισόδους & εξόδους στα τόξα μετάβασης και στις καταστάσεις. Συνεπώς, θα πρέπει να προσέχετε ιδιαίτερα κατά τον καθορισμό της FSM στον HDL κώδικά σας. Το Design Compiler και άλλα εργαλεία σύνθεσης υποδεικνύουν την υλοποίηση σε επίπεδο πυλών. Το Σχήμα A.34 παρουσιάζει ένα διάγραμμα μετάβασης καταστάσεων· ο διπλός κύκλος υποδεικνύει ότι το `S0` είναι η κατάσταση `reset`.



ΣΧΗΜΑ A.34 Η μονάδα divideby3fsm.

Σημειώστε ότι κάθε εντολή `always/process` υποδηλώνει ένα ξεχωριστό μπλοκ λογικής. Συνεπώς, ένα δεδομένο σήμα μπορεί να ανατίθεται μόνο σε μία `always/process`. Διαφορετικά, υπονοούνται δύο μονάδες hardware με βραχυκυκλωμένες εξόδους.

A.6.2 Απαρίθμηση Καταστάσεων

Οι SystemVerilog και VHDL υποστηρίζουν *τύπους απαρίθμησης* (απαρίθμηση) ως έναν αφηρημένο τρόπο αναπαράστασης της πληροφορίας, χωρίς αντιστοίχιση σε συγκεκριμένες δυαδικές κωδικοποιήσεις. Για παράδειγμα, η μηχανή πεπερασμένων καταστάσεων «διαίρεσης διά 3» που περιγράφεται στο Παράδειγμα A.36 χρησιμοποιεί τρεις καταστάσεις. Μπορούμε να δώσουμε τις καταστάσεις ονόματα, χρησιμοποιώντας έναν τύπο απαρίθμησης, αντί να αναφερόμαστε σ' αυτές με τις δυαδικές τιμές τους. Αυτό καθιστά τον κώδικα περισσότερο ευανάγνωστο και ευκολότερο σε αλλαγές. Το Παράδειγμα A.37 αναδιατυπώνει τον

κώδικα της FSM «διαίρεσης διά 3» χρησιμοποιώντας απαριθμητές καταστάσεις· το hardware δεν μεταβάλλεται κατά κανένα τρόπο.

Παράδειγμα A.37 Απαρίθμηση Καταστάσεων

SystemVerilog

```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic y);

  typedef enum logic [1:0] {S0, S1, S2} statetype;
  statetype state, nextstate;

  // Καταχωρητής κατάστασης
  always_ff @(posedge clk)
    if (reset) state <= S0;
    else       state <= nextstate;

  // Λογική επόμενης κατάστασης
  always_comb
    case (state)
      S0: nextstate = S1;
      S1: nextstate = S2;
      S2: nextstate = S0;
      default: nextstate = S0;
    endcase

  // Λογική εξόδου
  assign y = (state == S0);
endmodule
```

Η εντολή `typedef` ορίζει την `statetype` ως μία `logic` τιμή των δύο bit, με μία από τις εξής τρεις πιθανότητες: `S0`, `S1`, ή `S2`. Τα `state` και `nextstate` είναι σήματα τύπου `statetype`.

Οι απαριθμητές κωδικοποιήσεις χρησιμοποιούν εξ ορισμού αριθμητική σειρά: `S0` = 00, `S1` = 01 και `S2` = 10. Οι κωδικοποιήσεις μπορούν να ορίζονται ρητά από το χρήστη. Το ακόλουθο απόσπασμα κωδικοποιεί τις καταστάσεις ως one-hot τιμές των 3 bit:

```
typedef enum logic [2:0] {S0 = 3'b001,
                          S1 = 3'b010,
                          S2 = 3'b100} statetype;
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity divideby3FSM is
  port(clk, reset: in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of divideby3FSM is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
  -- καταχωρητής κατάστασης
  process(clk) begin
    if clk'event and clk = '1' then
      if reset = '1' then state <= S0;
      else state <= nextstate;
      end if;
    end if;
  end process;

  -- λογική επόμενης κατάστασης
  nextstate <= S1 when state = S0 else
               S2 when state = S1 else
               S0;

  -- λογική εξόδου
  y <= '1' when state = S0 else '0';
end;
```

Το Παράδειγμα αυτό ορίζει ένα νέο τύπο απαρίθμησης, τον `statetype`, με τρεις πιθανότητες: `S0`, `S1` και `S2`. Τα `state` και `nextstate` είναι σήματα τύπου `statetype`.

Το εργαλείο σύνθεσης μπορεί να επιλέξει την κωδικοποίηση των τύπων απαρίθμησης. Ένα καλό εργαλείο μπορεί να επιλέξει μια κωδικοποίηση η οποία θα απλοποιεί την υλοποίηση του hardware.

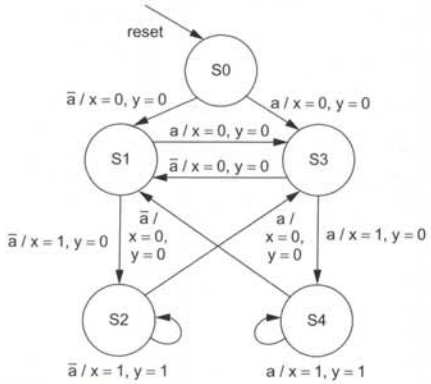
Εάν, για κάποιο λόγο, θέλαμε η έξοδος να είναι HIGH στις καταστάσεις S0 και S1, η λογική της εξόδου θα έπρεπε να τροποποιηθεί ως εξής:

SystemVerilog

```
// λογική εξόδου
assign y = (state == S0 | state == S1);
```

VHDL

```
-- λογική εξόδου
y <= '1' when (state = S0 or state = S1) else '0';
```



ΣΧΗΜΑ A.35 Το διάγραμμα μετάβασης καταστάσεων για την FSM.

A.6.3 FSM με Εισόδους

Η FSM «διαίρεσης διά 3» είχε μία έξοδο και καθόλου εισόδους. Το Παράδειγμα A.38 περιγράφει μια μηχανή πεπερασμένων καταστάσεων με μία είσοδο a και δύο εξόδους, η οποία παρουσιάζεται στο Σχήμα A.35. Η έξοδος x είναι true όταν η είσοδος την παρούσα στιγμή είναι ίδια όπως στον αμέσως προηγούμενο κύκλο. Η έξοδος y είναι true όταν η είσοδος την παρούσα στιγμή είναι ίδια όπως στους δύο προηγούμενους κύκλους. Το διάγραμμα μετάβασης καταστάσεων υποδεικνύει ότι πρόκειται για μια μηχανή Mealy, επειδή η έξοδος εξαρτάται από τις τρέχουσες εισόδους, καθώς και από την κατάσταση. Οι έξοδοι τιτλοφορούνται σε κάθε μετάβαση με βάση την είσοδο.

Παράδειγμα A.38

SystemVerilog

```
module historyFSM(input  logic clk,
                  input  logic reset,
                  input  logic a,
                  output logic x, y);

  typedef enum logic [2:0]
    {S0, S1, S2, S3, S4} statetype;
  statetype state, nextstate;

  // Καταχωρητής κατάστασης
  always_ff @(posedge clk)
    if (reset) state <= S0;
    else       state <= nextstate;

  // Λογική επόμενης κατάστασης
  always_comb
    case (state)
      S0: if (a) nextstate = S3;
          else   nextstate = S1;
      S1: if (a) nextstate = S3;
          else   nextstate = S2;
      S2: if (a) nextstate = S3;
          else   nextstate = S2;
      S3: if (a) nextstate = S4;
          else   nextstate = S1;
      S4: if (a) nextstate = S4;
          else   nextstate = S1;
      default:   nextstate = S0;
    endcase
```

VHDL

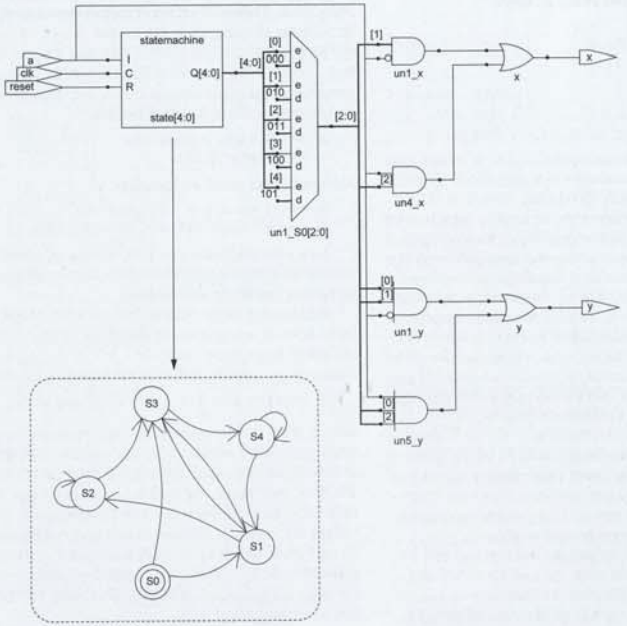
```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity historyFSM is
  port(clk, reset: in  STD_LOGIC;
       a:          in  STD_LOGIC;
       x, y:       out STD_LOGIC);
end;

architecture synth of historyFSM is
  type statetype is (S0, S1, S2, S3, S4);
  signal state, nextstate: statetype;
begin
  -- Καταχωρητής κατάστασης
  process(clk) begin
    if clk'event and clk = '1' then
      if reset = '1' then state <= S0;
      else state <= nextstate;
      end if;
    end if;
  end process;

  -- Λογική επόμενης κατάστασης
  process(state, a) begin
    case state is
      when S0 =>if a = '1' then nextstate <= S3;
                else            nextstate <= S1;
                end if;
      when S1 => if a = '1' then nextstate <= S3;
                 else            nextstate <= S2;
                 end if;
```


SystemVerilog (συνέχεια)

```
// Λογική εξόδου
assign x = ((state == S1 | state == S2) & ~a) |
           ((state == S3 | state == S4) & a);
assign y = (state == S2 & ~a) | (state == S4 & a);
endmodule
```

VHDL (συνέχεια)

```
      when S2 => if a = '1' then nextstate <= S3;
                 else            nextstate <= S2;
                 end if;
      when S3 => if a = '1' then nextstate <= S4;
                 else            nextstate <= S1;
                 end if;
      when S4 => if a = '1' then nextstate <= S4;
                 else            nextstate <= S1;
                 end if;
      when others =>             nextstate <= S0;
    end case;
  end process;

  -- Λογική εξόδου
  x <= '1' when
       ((state = S1 or state = S2) and a = '0') or
       ((state = S3 or state = S4) and a = '1')
    else '0';
    y <= '1' when
      (state = S2 and a = '0') or
      (state = S4 and a = '1')
    else '0';
end;
```



ΣΧΗΜΑ A.36 Η μονάδα historyFSM.

A.7 Ιδιώματα των Τύπων

Σ' αυτή την ενότητα θα εξηγήσουμε πιο αναλυτικά ορισμένα λεπτά σημεία αναφορικά με τους τύπους (δεδομένων) των SystemVerilog και VHDL.

SystemVerilog

Η στάνταρ `Verilog` χρησιμοποιεί κυρίως δύο τύπους: τους `reg` και `wire`. Παρά την ονομασία του τύπου του, ένα σήμα `reg` μπορεί να σχετίζεται μ' έναν καταχωρητή, αλλά μπορεί και όχι. Αυτό αποτέλεσε πηγή μεγάλης σύγχυσης για όσους προσπάθησαν να μάθουν τη γλώσσα. Η `SystemVerilog` εισήγαγε τον τύπο `logic` και χαλάρωσε κάποιες απαιτήσεις, σε μια προσπάθεια να εξαλείψει τη σύγχυση· γι' αυτό και τα παραδείγματα στο παρόν παράρτημα χρησιμοποιούν τον τύπο `logic`. Σ' αυτή την ενότητα θα περιγράψουμε πιο αναλυτικά τους τύπους `reg` και `wire`, για εκείνους τους χρήστες που καλούνται να χειριστούν παλαιότερο κώδικα `Verilog`.

Στην Verilog, εάν ένα σήμα εμφανίζεται στην αριστερή πλευρά του τελεστή <= ή = σ' ένα μπλοκ `always`, θα πρέπει να δηλώνεται ως `reg`. Διαφορετικά, θα πρέπει να δηλώνεται ως `wire`. Συνεπώς, ένα σήμα `reg` θα μπορούσε να είναι η έξοδος ενός flip-flop, ενός μανδαλωτή, ή ενός μπλοκ συνδυαστικής λογικής, ανάλογα με τη λίστα ευαισθησίας και τη διατύπωση του μπλοκ `always`.

Οι θύρες εισόδου και εξόδου χρησιμοποιούν εξ ορισμού τον τύπο `wire`, εκτός κι αν ο τύπος τους καθορίζεται ρητά ως `reg`. Το ακόλουθο παράδειγμα δείχνει πώς περιγράφεται ένα flip-flop σε συμβατική `Verilog`. Παρατηρήστε ότι τα `clk` και `d` έχουν εξ ορισμού τύπο wire, ενώ το `q` ορίζεται ρητά ως τύπου `reg` επειδή εμφανίζεται στην αριστερή πλευρά του τελεστή <= στο μπλοκ `always`.

```
module flop(input            clk,
            input      [3:0] d,
            output reg [3:0] q);

  always @(posedge clk)
    q <= d;
endmodule
```

Η `SystemVerilog` εισάγει τον τύπο `logic`, ο οποίος είναι ένα συνώνυμο για τον `reg` και αποτρέπει τη σύγχυση των χρηστών αναφορικά με το εάν αφορά πράγματι ένα flip-flop. Επιπλέον, η `SystemVerilog` χαλαρώνει τους κανόνες για τις εντολές `assign` και την ιεραρχική δημιουργία υποδειγμάτων (instances) θυρών, πράγμα το οποίο σημαίνει ότι ο τύπος `logic` μπορεί να χρησιμοποιείται έξω από μπλοκ `always`, σε σημεία όπου κατά παράδοση θα απαιτούνταν ένα σήμα `wire`. Συνεπώς, σχεδόν όλα τα σήματα στην `SystemVerilog` μπορούν να είναι τύπου `logic`. Μοναδική εξαίρεση είναι ότι τα σήματα με πολλαπλούς οδηγούς (π.χ., ένας τρισταθής δίαυλος) πρέπει να δηλώνονται ως ένα δίκτυο, όπως περιγράφεται στο Παράδειγμα A.11. Αυτός ο κανόνας επιτρέπει στην `SystemVerilog` να παράγει ένα μήνυμα σφάλματος αντί για μια τιμή `x` όταν ένα σήμα `logic` συνδέεται κατά λάθος σε πολλαπλούς οδηγούς.

Ο πιο κοινός τύπος δικτύου αποκαλείται `wire` ή `tri`. Αυτά τα δύο προσδιοριστικά τύπου είναι συνώνυμα, αλλά το `wire` χρησιμοποιείται κατά σύμβαση όταν υπάρχει μόνο ένας οδηγός, ενώ το `tri` χρησιμοποιείται όταν υπάρχουν πολλαπλοί οδηγοί. Συνεπώς, ο τύπος `wire` έχει παροπλιστεί στην `SystemVerilog`, επειδή προτιμάται ο τύπος `logic` για τα σήματα μ' έναν μεμονωμένο οδηγό.

Όταν ένα δίκτυο `tri` οδηγείται σε μια μεμονωμένη τιμή από έναν ή περισσότερους οδηγούς, λαμβάνει αυτή την τιμή. Όταν δεν οδηγείται, μένει αιωρούμενο (`z`). Όταν οδηγείται σε διαφορετικές τιμές (0, 1, ή `x`) από πολλαπλούς οδηγούς, είναι σε κατάσταση διαμάχης (`x`).

Υπάρχουν και άλλοι τύποι δικτύων, οι οποίοι συμπεριφέρονται διαφορετικά όταν οδηγούνται ή δεν οδηγούνται από πολλαπλές πηγές.

VHDL

Ανόμοια με την SystemVerilog, η VHDL επιβάλλει ένα αυστηρό σύστημα καθορισμού τύπων δεδομένων, το οποίο μπορεί να προστατέψει το χρήστη από κάποια σφάλματα, αλλά περιστασιακά αποδεικνύεται άβολο.

Παρά τη θεμελιακή σημασία του, ο τύπος `STD_LOGIC` δεν είναι εγγενής στην VHDL. Αντιθέτως, αποτελεί μέρος της βιβλιοθήκης `IEEE.STD_LOGIC_1164`. Συνεπώς, κάθε αρχείο θα πρέπει να περιέχει τις κατάλληλες εντολές για την πρόσβαση στη βιβλιοθήκη, όπως είδαμε σε προηγούμενα παραδείγματα.

Επιπλέον, ο τύπος `IEEE.STD_LOGIC_1164` δεν διαθέτει δυνατότητες για βασικές λειτουργίες, όπως η πρόσθεση, η σύγκριση, οι ολισθήσεις και η μετατροπή σε ακεραίους για δεδομένα τύπου `STD_LOGIC_VECTOR`. Οι περισσότεροι κατασκευαστές εφαρμογών CAD έχουν υιοθετήσει βιβλιοθήκες που παρέχουν αυτές τις λειτουργίες:

```
IEEE.STD_LOGIC_UNSIGNED και
IEEE.STD_LOGIC_SIGNED.
```

Η VHDL διαθέτει επίσης έναν τύπο `BOOLEAN` με δύο τιμές, τις `true` και `false`. Τιμές τύπου BOOLEAN επιστρέφονται από συγκρίσεις (π.χ. `s = '0'`) και χρησιμοποιούνται σε εντολές εκτέλεσης υπό συνθήκη, όπως η when. Παρά την προφανή αντιστοίχιση που θα μπορούσε να κάνει κάποιος, ότι η BOOLEAN τιμή `true` ισοδυναμεί με την `STD_LOGIC '1'` και η BOOLEAN τιμή `false` ισοδυναμεί με την `STD_LOGIC '0'`, αυτοί οι δύο τύποι δεν είναι ταυτόσημοι και δεν μπορούν να χρησιμοποιούνται ο ένας στη θέση του άλλου. Συνεπώς, ο ακόλουθος κώδικας δεν είναι έγκυρος:

```
y <= d1 when s else d0;
q <= (state = S2);
```

Αντ' αυτού, θα έπρεπε να γράψουμε

```
y <= d1 when s = '1' else d0;
q <= '1' when state = S2 else '0';
```

Αν και δεν θα δηλώσουμε κανένα σήμα ως τύπου BOOLEAN, αυτό υπονοείται αυτόματα όταν γίνονται συγκρίσεις και χρησιμοποιείται από τις εντολές εκτέλεσης υπό συνθήκη.

Παρόμοια, η VHDL διαθέτει έναν τύπο INTEGER ο οποίος αναπαριστά θετικούς και αρνητικούς ακέραιους αριθμούς. Τα σήματα τύπου INTEGER έχουν πεδίο τιμών $-2^{31} \ldots 2^{31}-1$. Οι ακέραιες τιμές χρησιμοποιούνται ως ενδείκτες διαύλων. Για παράδειγμα, στην εντολή

```
y <= a(3) and a(2) and a(1) and a(0);
```

τα 0, 1, 2 και 3 είναι ακέραιοι που εξυπηρετούν ως ενδείκτες για την επιλογή των bits του σήματος a. Δεν μπορούμε να «δεικτοδοτήσουμε» απευθείας ένα δίαυλο μ' ένα σήμα `STD_LOGIC` ή `STD_LOGIC_VECTOR`. Αντ' αυτού, θα πρέπει να μετατρέψουμε το σήμα σε τύπο INTEGER. Αυτό επιδεικνύεται στο Παράδειγμα A.39 για έναν πολυπλέκτη 8:1, ο οποίος επιλέγει ένα bit από ένα διάνυσμα χρησιμοποιώντας έναν 3-bit δείκτη. Η συνάρτηση `CONV_INTEGER` ορίζεται στη βιβλιοθήκη `STD_LOGIC_UNSIGNED` και εκτελεί τη μετατροπή από τον τύπο `STD_LOGIC_VECTOR` στον τύπο `INTEGER` για θετικές (μη-προσημασμένες) τιμές.

SystemVerilog (συνέχεια)

Οι τύποι αυτοί χρησιμοποιούνται σπανίως, αλλά μπορούν να εμφανίζονται σε οποιαδήποτε θέση θα έπρεπε κανονικά να εμφανίζεται ένα δίκτυο `tri` (π.χ., για σήματα με πολλαπλούς οδηγούς). Οι τύποι περιγράφονται στον Πίνακα A.7:

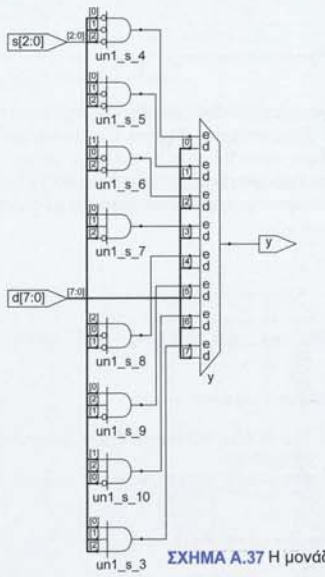
ΠΙΝΑΚΑΣ A.7 Χειρισμός δικτύων

Τύπος δικτύου	Χωρίς οδηγό	Συγκρουόμενοι οδηγοί
`tri`	`z`	`x`
`triand`	`z`	`0` εάν οποιοιδήποτε είναι `0`
`trior`	`z`	`1` εάν οποιοιδήποτε είναι `1`
`trireg`	προηγούμενη τιμή	`x`
`tri0`	`0`	`x`
`tri1`	`1`	`x`

Οι περισσότερες λειτουργίες, όπως οι πράξεις πρόσθεσης/αφαίρεσης και η Boolean λογική είναι πανομοιότυπες ανεξάρτητα από το εάν ένας αριθμός είναι προσημασμένος ή όχι. Ωστόσο, η σύγκριση μεγεθών (απόλυτων τιμών), ο πολλαπλασιασμός και η αριθμητική ολίσθηση δεξιά εκτελούνται διαφορετικά για τους προσημασμένους αριθμούς.

Στην Verilog, τα δίκτυα θεωρούνται εξ ορισμού μη-προσημασμένα μεγέθη. Η προσθήκη του τροποποιητή *signed* (π.χ., `logic signed a [31:0]`) επιβάλλει την αντιμετώπιση του δικτύου ως προσημασμένο μέγεθος.

Παράδειγμα A.39 Πολυπλέκτης 8:1 με Μετατροπή Τύπου



ΣΧΗΜΑ A.37 Η μονάδα mux8.

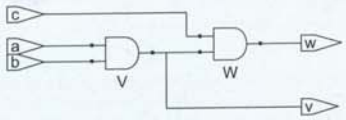
VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity mux8 is
  port(d: in  STD_LOGIC_VECTOR(7 downto 0);
       s: in  STD_LOGIC_VECTOR(2 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of mux8 is
begin
  y <= d(CONV_INTEGER(s));
end;
```

Η VHDL έχει επίσης αυστηρούς κανόνες όσον αφορά την χρήση θυρών out αποκλειστικά για έξοδο. Για παράδειγμα, ο ακόλουθος κώδικας για τις 2-/3-εισόδων πύλες AND δεν είναι έγκυρος, επειδή το v χρησιμοποιείται για τον υπολογισμό του w και επίσης ως έξοδος

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity and23 is
  port(a, b, c: in  STD_LOGIC;
       v, w:    out STD_LOGIC);
end;

architecture synth of and23 is
```

Παράδειγμα A.39 Πολυπλέκτης 8:1 με Μετατροπή Τύπου (συνέχεια)



ΣΧΗΜΑ A.38 Η μονάδα and23.

```
begin
  v <= a and b;
  w <= v and c;
end;
```

Η VHDL ορίζει έναν ειδικό τύπο θύρας με όνομα `buffer` για την επίλυση αυτού του προβλήματος. Ένα σήμα συνδεδεμένο σε μια θύρα τύπου `buffer` συμπεριφέρεται ως έξοδος, αλλά θα μπορούσε επίσης να χρησιμοποιείται εντός της λειτουργικής μονάδας. Δυστυχώς, οι θύρες τύπου `buffer` προκαλούν προβλήματα στην ιεραρχική σχεδίαση, επειδή κάποιες έξοδοι σε υψηλότερα επίπεδα της ιεραρχίας μπορεί επίσης να χρειάζεται να μετατραπούν σε `buffer`. Μια καλύτερη εναλλακτική είναι να δηλώνετε ένα εσωτερικό σήμα και κατόπιν να οδηγείτε την έξοδο βασιζόμενοι σ' αυτό το σήμα, ως εξής:

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity and23 is
  port(a, b, c: in  STD_LOGIC;
       v, w:    out STD_LOGIC);
end;

architecture synth of and23 is
  signal v_int: STD_LOGIC;
begin
  v_int <= a and b;
  v <= v_int;
  w <= v_int and c;
end;
```

OPTIONAL

A.8 Παραμετροποιημένες Μονάδες

Μέχρι τώρα, όλες οι λειτουργικές μονάδες που καθορίσαμε είχαν εισόδους και εξόδους σταθερού εύρους (πλάτους). Για παράδειγμα, χρειάστηκε να καθορίσουμε ξεχωριστές μονάδες για τους πολυπλέκτες 2:1 με εύρος 4 και 8 bit. Οι γλώσσες HDL επιτρέπουν μεταβλητό εύρος bit μέσω των παραμετροποιημένων λειτουργικών μονάδων. Το Παράδειγμα A.40 δηλώνει έναν παραμετροποιημένο πολυπλέκτη 2:1 με προκαθορισμένο εύρος 8 και κατόπιν τον χρησιμοποιεί για να δημιουργήσει πολυπλέκτες 4:1 με εύρος 8 και 12 bit.

Παράδειγμα A.40 Παραμετροποιημένοι Πολυπλέκτες των *N* bit

SystemVerilog

```
module mux2
  #(parameter width = 8)
   (input  logic [width-1:0] d0, d1,
    input  logic             s,
    output logic [width-1:0] y);

   assign y = s ? d1 : d0;
endmodule
```

Η SystemVerilog επιτρέπει τη χρήση μιας εντολής `#(parameter ...)` πριν από τις εισόδους και εξόδους για τον ορισμό παραμέτρων. Η εντολή `parameter` περιλαμβάνει μια προκαθορισμένη τιμή (8) της παραμέτρου, `width`. Ο αριθμός των bit στις εισόδους και τις εξόδους μπορεί να εξαρτάται από αυτή την παράμετρο.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux2 is
  generic(width: integer := 8);
  port(d0,
    d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
    s:  in  STD_LOGIC;
    y:  out STD_LOGIC_VECTOR(width-1 downto 0));
end;

architecture synth of mux2 is
begin
  y <= d0 when s = '0' else d1;
end;
```


SystemVerilog (συνέχεια)

```
module mux4_8(input  logic [7:0] d0, d1, d2, d3,
              input  logic [1:0] s,
              output logic [7:0] y);

  logic [7:0] low, hi;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 himux(d2, d3, s[0], hi);
  mux2 outmux(low, hi, s[1], y);
endmodule
```

Η μονάδα του 8-bit πολυπλέκτη 4:1 δημιουργεί τρία υποδείγματα πολυπλεκτών 2:1 χρησιμοποιώντας το προκαθορισμένο εύρος για το καθένα.

Εν αντιθέσει, μια μονάδα 12-bit πολυπλέκτη 4:1 (mux4_12) θα έπρεπε να παρακάμψει (override) το προκαθορισμένο εύρος χρησιμοποιώντας τη σύνταξη #() πριν από το όνομα υποδείγματος, όπως παρουσιάζεται στη συνέχεια.

```
module mux4_12(input logic [11:0] d0, d1, d2, d3,
               input logic [1:0]  s,
               output logic [11:0] y);

  logic [11:0] low, hi;

  mux2 #(12) lowmux(d0, d1, s[0], low);
  mux2 #(12) himux(d2, d3, s[0], hi);
  mux2 #(12) outmux(low, hi, s[1], y);
endmodule
```

Μην συγχέετε τη χρήση του συμβόλου # που υποδεικνύει καθυστερήσεις με τη χρήση του #(...) που χρησιμοποιείται για τον ορισμό και την παράκαμψη παραμέτρων.

VHDL (συνέχεια)

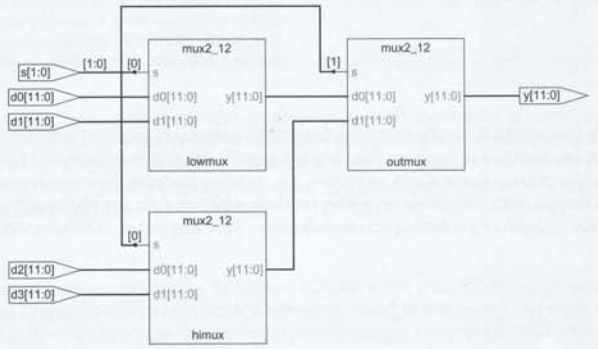
Η εντολή generic περιλαμβάνει μια προεπιλεγμένη τιμή (8) για την παράμετρο `width`. Η τιμή είναι ένας ακέραιος.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity mux4_8 is
  port(d0, d1, d2,
       d3: in  STD_LOGIC_VECTOR(7 downto 0);
       s:  in  STD_LOGIC_VECTOR(1 downto 0);
       y:  out STD_LOGIC_VECTOR(7 downto 0));
end;

architecture struct of mux4_8 is
  component mux2
    generic(width: integer := 8);
  port(d0,
    d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
    s:  in  STD_LOGIC;
    y:  out STD_LOGIC_VECTOR(width-1 downto 0));
  end component;
  signal low, hi: STD_LOGIC_VECTOR(7 downto 0);
begin
  lowmux: mux2 port map(d0, d1, s(0), low);
  himux:  mux2 port map(d2, d3, s(0), hi);
  outmux: mux2 port map(low, hi, s(1), y);
end;
```

Ο 8-bit πολυπλέκτης 4:1 δημιουργεί τρία υποδείγματα πολυπλεκτών 2:1 χρησιμοποιώντας το προκαθορισμένο εύρος για το καθένα.

Εν αντιθέσει, μια μονάδα 12-bit πολυπλέκτη 4:1 (mux4_12) θα έπρεπε να παρακάμψει το προκαθορισμένο εύρος χρησιμοποιώντας τη σύνταξη generic map όπως παρουσιάζεται στη συνέχεια.

```
lowmux: mux2 generic map(12)
             port map(d0, d1, s(0), low);
himux:  mux2 generic map(12)
             port map(d2, d3, s(0), hi);
outmux: mux2 generic map(12)
             port map(low, hi, s(1), y);
```



ΣΧΗΜΑ A.39 Η μονάδα mux4_12.

Το Παράδειγμα Α.41 παρουσιάζει έναν αποκωδικοποιητή, ο οποίος είναι μια ακόμη καλύτερη περίπτωση εφαρμογής των παραμετροποιημένων λειτουργικών μονάδων. Ο καθορισμός ενός μεγάλου N:2^N αποκωδικοποιητή είναι κουραστικός όταν γίνεται με εντολές case, αλλά εύκολος όταν χρησιμοποιείται παραμετροποιημένος κώδικας, ο οποίος θέτει απλώς το κατάλληλο bit εξόδου σε 1. Συγκεκριμένα, ο αποκωδικοποιητής χρησιμοποιεί κλειδωμένες αναθέσεις για τον ορισμό όλων των bit σε 0 και κατόπιν αλλάζει το κατάλληλο bit σε 1. Το Σχήμα Α.28 παρουσιάζει το σχηματικό για έναν αποκωδικοποιητή 3:8.

Παράδειγμα Α.41 Παραμετροποιημένος Αποκωδικοποιητής N:2^N

SystemVerilog

```
module decoder #(parameter N = 3)
               (input  logic [N-1:0]
                output logic [2**N-1:0]

  always_comb
    begin
      y = 0;
      y[a] = 1;
    end
endmodule
```

Το 2**N υποδεικνύει 2^N.

VHDL

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

entity decoder is
  generic(N: integer := 3);
  port(a: in  STD_LOGIC_VECTOR(N-1 downto 0);
       y: out STD_LOGIC_VECTOR(2**N-1 downto 0))
end;

architecture synth of decoder is
begin
  process (a)
    variable tmp: STD_LOGIC_VECTOR(2**N-1 downto 0)
  begin
    tmp := CONV_STD_LOGIC_VECTOR(0, 2**N);
    tmp(CONV_INTEGER(a)) := '1';
    y <= tmp;
  end process;
end;
```

Το 2**N υποδεικνύει 2^N.

Η συνάρτηση `CONV_STD_LOGIC_VECTOR(0, 2**N)` παράγει ένα διάνυσμα `STD_LOGIC_VECTOR` μήκους 2^N το οποίο περιέχει 0 σε όλες τις θέσεις. Για τη χρήση της απαιτείται η βιβλιοθήκη `STD_LOGIC_ARITH`. Η συγκεκριμένη συνάρτηση είναι χρήσιμη και σε άλλες παραμετροποιημένες λειτουργίες, όπως flip-flop με δυνατότητα reset, τα οποία πρέπει να παράγουν σταθερές με παραμετροποιημένο το πλήθος των bit. Επειδή στην VHDL ο δείκτης που χρησιμοποιείται για τη δεικτοδότηση των bit πρέπει να είναι ακέραιος, η συνάρτηση `CONV_INTEGER` χρησιμοποιείται για τη μετατροπή του a από διάνυσμα `STD_LOGIC_VECTOR` σε ακέραιο.

Οι γλώσσες HDL παρέχουν επίσης μια εντολή `generate`, οι οποίες παράγουν μεταβλητή «ποσότητα» hardware ανάλογα με την τιμή μιας παραμέτρου. Η `generate` υποστηρίζει βρόχους for και εντολές `if`, μέσω των οποίων καθορίζεται το πλήθος και το είδος των μονάδων hardware που θα παραχθούν. Το Παράδειγμα Α.42 επιδεικνύει τη χρήση εντολών `generate` για την παραγωγή μιας συνάρτησης AND N εισόδων από μια σειρά διαδοχικά συνδεδεμένων (cascade) πυλών AND 2 εισόδων.

Παράδειγμα Α.42 Παραμετροποιημένη Πύλη AND N Εισόδων

SystemVerilog

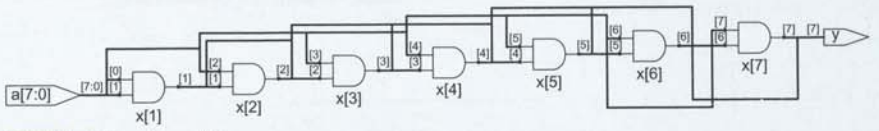
```
module andN
  #(parameter width = 8)
   (input  logic [width-1:0] a,
    output logic             y);

  genvar i;
  logic [width-1:1] x;

  generate
    for (i=1; i<width; i=i+1) begin:forloop
      if (i == 1)
        assign x[1] = a[0] & a[1];
      else
        assign x[i] = a[i] & x[i-1];
    end
  endgenerate
  assign y = x[width-1];
endmodule
```

Ο βρόχος της for επαναλαμβάνεται για i = 1, 2, ..., width–1 φορές, παράγοντας πολλές διαδοχικές πύλες AND. Η δήλωση begin σ' ένα βρόχο `generate` for πρέπει να ακολουθείται από ένα χαρακτήρα `:` και μια αυθαίρετα επιλεγμένη ετικέτα (`forloop` στο παράδειγμα). Φυσικά, το να γράψουμε `assign y = &a` θα ήταν πολύ πιο εύκολο!

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity andN is
  generic(width: integer := 8);
  port(a: in  STD_LOGIC_VECTOR(width-1 downto 0);
       y: out STD_LOGIC);
end;

architecture synth of andN is
  signal x: STD_LOGIC_VECTOR(width-1 downto 1);
begin
  AllBits: for i in 1 to width-1 generate
    LowBit: if i = 1 generate
      A1: x(1) <= a(0) and a(1);
    end generate;
    OtherBits: if i /= 1 generate
      Ai: x(i) <= a(i) and x(i-1);
    end generate;
  end generate;
  y <= x(width-1);
end;
```

Η μεταβλητή `i` του βρόχου δεν χρειάζεται να δηλωθεί.



ΣΧΗΜΑ Α.40 Η μονάδα andN.

Να χρησιμοποιείτε την εντολή `generate` με προσοχή· είναι πολύ εύκολο να παράγει κατά λάθος πολύ μεγάλες ποσότητες hardware!

Α.9 Μνήμη

OPTIONAL

Μνήμες όπως οι RAM και ROM είναι απλό να μοντελοποιηθούν σε μια HDL. Δυστυχώς, οι αποτελεσματικές κυκλωματικές υλοποιήσεις είναι τόσο εξειδικευμένες και εξαρτώμενες από την τεχνολογία κατασκευής, που τα περισσότερα εργαλεία δεν έχουν δυνατότητα άμεσης σύνθεσης των μνημών. Αντ' αυτού, μπορεί να χρησιμοποιείται κάποια ειδική εφαρμογή-γεννήτρια μνημών, ή βιβλιοθήκες μνημών· εναλλακτικά, η μνήμη μπορεί να σχεδιάζεται επί τούτου.

Α.9.1 RAM

Το Παράδειγμα Α.43 περιγράφει μια σύγχρονη, 64-λέξεων × 32-bit RAM μιας θύρας, με ξεχωριστούς διαύλους ανάγνωσης και εγγραφής δεδομένων. Όταν λαμβάνεται το σήμα επίτρεψης εγγραφής, we, η επιλεγμένη διεύθυνση της RAM γράφεται με την `din` στην ανοδική ακμή του ρολογιού. Σε κάθε περίπτωση, η RAM διαβάζεται στην `dout`.

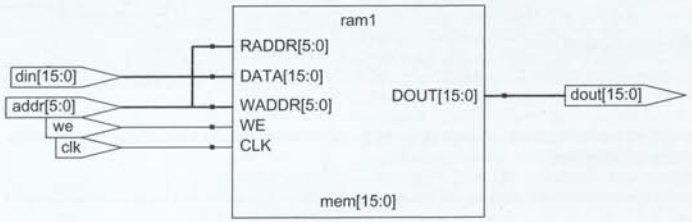
Παράδειγμα A.43 RAM

SystemVerilog

```
module ram #(parameter N = 6, M = 32)
            (input  logic         clk,
             input  logic         we,
             input  logic [N-1:0] adr,
             input  logic [M-1:0] din,
             output logic [M-1:0] dout);

  logic [M-1:0] mem[2**N-1:0];

  always @(posedge clk)
    if (we) mem[adr] <= din;

  assign dout = mem[adr];
endmodule
```

VHDL

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

entity ram_array is
  generic(N: integer := 6; M: integer := 32);
  port(clk,
    we:   in  STD_LOGIC;
    adr:  in  STD_LOGIC_VECTOR(N-1 downto 0);
    din:  in  STD_LOGIC_VECTOR(M-1 downto 0);
    dout: out STD_LOGIC_VECTOR(M-1 downto 0));
end;

architecture synth of ram_array is
  type mem_array is array((2**N-1) downto 0)
      of STD_LOGIC_VECTOR(M-1 downto 0);
  signal mem: mem_array;
begin
  process(clk) begin
    if clk'event and clk = '1' then
      if we = '1' then
        mem(CONV_INTEGER(adr)) <= din;
      end if;
    end if;
  end process;

  dout <= mem(CONV_INTEGER(adr));
end;
```



ΣΧΗΜΑ A.41 Συνθετικά παραγόμενη μονάδα ram

Το Παράδειγμα A.44 δείχνει πώς μπορεί να τροποποιηθεί η RAM ώστε να έχει μόνο έναν, αμφίδρομο δίαυλο δεδομένων. Αυτό μειώνει τον αριθμό των απαιτούμενων αγωγών, αλλά απαιτεί την προσθήκη τρισταθών οδηγών και στα δύο άκρα του διαύλου. Συνήθως η διασύνδεση «από σημείο σε σημείο» προτιμάται έναντι των τρισταθών διαύλων σε υλοποιήσεις VLSI.

Παράδειγμα A.44 RAM με Αμφίδρομο Δίαυλο Δεδομένων

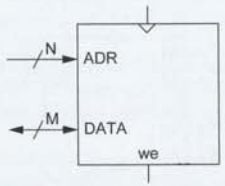
SystemVerilog

```
module ram #(parameter N = 6, M = 32)
            (input  logic         clk,
             input  logic         we,
             input  logic [N-1:0] adr,
             inout  tri   [M-1:0] data);

  logic [M-1:0] mem[2**N-1:0];

  always @(posedge clk)
    if (we) mem[adr] <= data;

  assign data = we ? 'z : mem[adr];
endmodule
```

Παρατηρήστε ότι η `data` δηλώνεται ως μια θύρα inout επειδή μπορεί να χρησιμοποιείται τόσο ως είσοδος όσο και ως έξοδος. Επίσης, το 'z είναι μια συντομογραφία που αναπαριστά το γέμισμα ενός διαύλου τυχαίου μήκους με τιμές z.

VHDL

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

entity ram_array is
  generic(N: integer := 6; M: integer := 32);
  port(clk,
    we:   in    STD_LOGIC;
    adr:  in    STD_LOGIC_VECTOR(N-1 downto 0);
    data: inout STD_LOGIC_VECTOR(M-1 downto 0));
end;

architecture synth of ram_array is
  type mem_array is array((2**N-1) downto 0)
      of STD_LOGIC_VECTOR(M-1 downto 0);
  signal mem: mem_array;
begin
  process(clk) begin
    if clk'event and clk = '1' then
      if we = '1' then
        mem(CONV_INTEGER(adr)) <= data;
      end if;
    end if;
  end process;

  data <= (OTHERS => 'Z') when we = '1'
          else mem(CONV_INTEGER(adr));
end;
```



ΣΧΗΜΑ A.42 Συνθετικά παραγόμενη μονάδα ram με αμφίδρομο δίαυλο δεδομένων.

A.9.2 Πολύθυρα Αρχεία Καταχωρητών

Ένα πολύθυρο αρχείο καταχωρητών (multiported register file) έχει πολλαπλές θύρες ανάγνωσης και/ή εγγραφής. Το Παράδειγμα A.45 περιγράφει ένα σύγχρονο αρχείο καταχωρητών με τρεις θύρες. Οι θύρες 1 και 2 είναι ανάγνωσης, ενώ η θύρα 3 είναι εγγραφής.

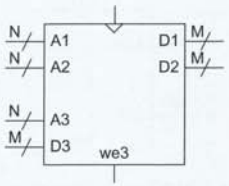
Παράδειγμα A.45 Αρχείο Καταχωρητών με Τρεις Θύρες

SystemVerilog

```
module ram3port #(parameter N = 6, M = 32)
                 (input  logic         clk,
                  input  logic         we3,
                  input  logic [N-1:0] a1, a2, a3,
                  output logic [M-1:0] d1, d2,
                  input  logic [M-1:0] d3);

  logic [M-1:0] mem[2**N-1:0];

  always @(posedge clk)
    if (we3) mem[a3] <= d3;

  assign d1 = mem[a1];
  assign d2 = mem[a2];
endmodule
```

VHDL

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

entity ram3port is
  generic(N: integer := 6; M: integer := 32);
  port(clk,
    we3:      in  STD_LOGIC;
    a1,a2,a3: in  STD_LOGIC_VECTOR(N-1 downto 0);
    d1, d2:   out STD_LOGIC_VECTOR(M-1 downto 0);
    d3:       in  STD_LOGIC_VECTOR(M-1 downto 0));
end;

architecture synth of ram3port is
  type mem_array is array((2**N-1) downto 0)
       of STD_LOGIC_VECTOR(M-1 downto 0);
  signal mem: mem_array;
begin
  process(clk) begin
    if clk'event and clk = '1' then
      if we3 = '1' then
        mem(CONV_INTEGER(a3)) <= d3;
      end if;
    end if;
  end process;

  d1 <= mem(CONV_INTEGER(a1));
  d2 <= mem(CONV_INTEGER(a2));
end;
```



ΣΧΗΜΑ A.43 Αρχείο καταχωρητών με τρεις θύρες.

A.9.3 ROM

Μια μνήμη μόνο-ανάγνωσης (ROM) μοντελοποιείται συνήθως με τη χρήση μιας εντολής `case` που περιλαμβάνει μία καταχώριση για κάθε λέξη. Το Παράδειγμα A.46 περιγράφει μια 4-λέξεων X 3-bit ROM. Συχνά, οι μνήμες ROM παράγονται συνθετικά, σε μπλοκ τυχαίας λογικής τα οποία εκτελούν την ισοδύναμη λειτουργία. Για μικρές ROM, αυτό μπορεί να είναι πιο αποτελεσματικό. Για μεγαλύτερες ROM, είναι συνήθως καλύτερο να χρησιμοποιείται μια γεννήτρια ROM ή κάποια βιβλιοθήκη. Στο Σχήμα A.27 απεικονίζεται το σχηματικό ενός αποκωδικοποιητή 7 ενδείξεων υλοποιημένου με μια ROM.

Παράδειγμα A.46 ROM

SystemVerilog

```
module rom(input  logic [1:0] adr,
           output logic [2:0] dout);

  always_comb
    case(adr)
      2'b00: dout = 3'b011;
      2'b01: dout = 3'b110;
      2'b10: dout = 3'b100;
      2'b11: dout = 3'b010;
    endcase
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity rom is
  port(adr: in  STD_LOGIC_VECTOR(1 downto 0);
       dout: out STD_LOGIC_VECTOR(2 downto 0));
end;

architecture synth of rom is
begin
  process(adr) begin
    case adr is
      when "00" => dout <= "011";
      when "01" => dout <= "110";
      when "10" => dout <= "100";
      when "11" => dout <= "010";
      when others => dout <= (OTHERS => 'X');
    end case;
  end process;
end;
```

A.10 Προγράμματα Δοκιμής

Ένα testbench (πρόγραμμα ελέγχου/δοκιμής) είναι μια λειτουργική μονάδα HDL, η οποία χρησιμοποιείται για τη δοκιμή μιας άλλης λειτουργικής μονάδας, του αποκαλούμενου *στοιχείου υπό έλεγχο/δοκιμή* (device under test, DUT). Το testbench περιέχει εντολές οι οποίες εφαρμόζουν εισόδους στο DUT και, στην ιδανική περίπτωση, ελέγχουν ότι παράγονται οι σωστές έξοδοι. Τα μοτίβα εισόδων και επιθυμητών εξόδων αποκαλούνται *διανύσματα ελέγχου* (test vectors).

Ας εξετάσουμε τη δοκιμή της λειτουργικής μονάδας `sillyfunction` από την Ενότητα A.1.1, η οποία υπολογίζει τη συνάρτηση $Y = \overline{A}\overline{B}\overline{C} + A\overline{B}\overline{C} + A\overline{B}C$. Δεδομένου ότι πρόκειται για μια απλή λειτουργική μονάδα, μπορούμε να εκτελέσουμε εξαντλητικούς ελέγχους εφαρμόζοντας και τα οκτώ πιθανά διανύσματα ελέγχου.

Το Παράδειγμα A.47 επιδεικνύει ένα απλό testbench: δημιουργεί τρία υποδείγματα (instances) του DUT και κατόπιν εφαρμόζει τις εισόδους. Κλειδωμένες αναθέσεις και καθυστερήσεις χρησιμοποιούνται για την εφαρμογή των εισόδων με την κατάλληλη σειρά. Ο χρήστης πρέπει να εμφανίσει τα αποτελέσματα της προσομοίωσης και να επαληθεύσει, με οπτική εξέταση, ότι έχουν παραχθεί οι σωστές έξοδοι. Τα testbench προσομοιώνονται όπως και οποιαδήποτε άλλη λειτουργική μονάδα. Ωστόσο, δεν είναι συνθέσιμα.

Παράδειγμα Α.47 Testbench

SystemVerilog

```
module testbench1();
  logic a, b, c;
  logic y;

  // δημιουργία υποδείγματος για το DUT
  sillyfunction dut(a, b, c, y);

  // εφαρμογή εισόδων μία προς μία

  initial begin
    a = 0; b = 0; c = 0; #10;
    c = 1;               #10;
    b = 1; c = 0;        #10;
    c = 1;               #10;
    a = 1; b = 0; c = 0; #10;
    c = 1;               #10;
    b = 1; c = 0;        #10;
    c = 1;               #10;
  end
endmodule
```

Η εντολή `initial` εκτελεί τις εντολές που περιλαμβάνονται στον κορμό της, στην αρχή της προσομοίωσης. Σ' αυτή την περίπτωση, αρχικά εφαρμόζει το μοτίβο εισόδου 000 και αναμένει για 10 μονάδες χρόνου. Στη συνέχεια εφαρμόζει το 001 και αναμένει για 10 επιπλέον μονάδες χρόνου, κ.ο.κ., έως ότου εφαρμοστούν όλες οι δυνατές είσοδοι. Εντολές `initial` θα πρέπει να χρησιμοποιούνται μόνο σε testbenches για σκοπούς προσομοίωσης – όχι σε λειτουργικές μονάδες που προορίζονται για σύνθεση σε πραγματικό hardware. Το hardware δεν διαθέτει κάποια μαγική ικανότητα να εκτελεί μια ακολουθία ειδικών βημάτων κατά την εκκίνηση της λειτουργίας του.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity testbench1 is -- no inputs or outputs
end;

architecture sim of testbench1 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
         y:       out STD_LOGIC);
  end component;
  signal a, b, c, y: STD_LOGIC;
begin
  -- δημιουργία υποδείγματος για το DUT
  dut: sillyfunction port map(a, b, c, y);

  -- εφαρμογή εισόδων μία προς μία
  process begin
    a <= '0'; b <= '0'; c <= '0'; wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    b <= '1'; c <= '0';           wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    a <= '1'; b <= '0'; c <= '0'; wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    b <= '1'; c <= '0';           wait for 10 ns;
    c <= '1';                     wait for 10 ns;
    wait; -- αναμονή επ' άπειρον
  end process;
end;
```

Η εντολή `process` εφαρμόζει κατ' αρχήν το μοτίβο εισόδου 000 και αναμένει για 10 ns. Στη συνέχεια εφαρμόζει το μοτίβο 001 και αναμένει για 10 επιπλέον ns, κ.ο.κ., έως ότου εφαρμοστούν όλες οι (οκτώ) πιθανές είσοδοι.

Στο τέλος, η διαδικασία αναμένει επ' αόριστον· διαφορετικά, θα ξεκινούσε και πάλι από την αρχή, εφαρμόζοντας κατ' επανάληψη τα διανύσματα ελέγχου.

Ο χειροκίνητος έλεγχος της ορθότητας των εξόδων είναι κουραστικός και επιρρεπής σε λάθη. Επιπλέον, το να εξακριβώσετε ποιες είναι οι σωστές έξοδοι είναι πολύ ευκολότερο όταν το σχέδιο είναι ακόμα φρέσκο στο μυαλό σας· εάν κάνετε δευτερεύουσας σημασίας αλλαγές και χρειαστεί να επαναλάβετε τον έλεγχο μερικές εβδομάδες αργότερα, το να εξακριβώσετε ποιες είναι οι σωστές έξοδοι γίνεται κουραστικό. Μια πολύ καλύτερη προσέγγιση είναι να γράψετε ένα πρόγραμμα testbench με δυνατότητα αυτο-ελέγχου, όπως δείχνει το Παράδειγμα Α.48.

Παράδειγμα Α.48 Πρόγραμμα Testbench με Δυνατότητα Αυτο-Ελέγχου

SystemVerilog

```
module testbench2();
  logic a, b, c;
  logic y;

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // apply inputs one at a time
  // checking results

  initial begin
    a = 0; b = 0; c = 0; #10;
    assert (y === 1) else $error("000 failed.");
    c = 1;               #10;
    assert (y === 0) else $error("001 failed.");
    b = 1; c = 0;        #10;
    assert (y === 0) else $error("010 failed.");
    c = 1;               #10;
    assert (y === 0) else $error("011 failed.");
    a = 1; b = 0; c = 0; #10;
    assert (y === 1) else $error("100 failed.");
    c = 1;               #10;
    assert (y === 1) else $error("101 failed.");
    b = 1; c = 0;        #10;
    assert (y === 0) else $error("110 failed.");
    c = 1;               #10;
    assert (y === 0) else $error("111 failed.");
  end
endmodule
```

Η εντολή `assert` της SystemVerilog ελέγχει εάν μια καθορισθείσα συνθήκη είναι αληθής (true). Εάν όχι, εκτελεί την `else`. Ο όρος `$error` της `else` εκτυπώνει ένα μήνυμα σφάλματος, το οποίο περιγράφει το πρόβλημα. Η `assert` αγνοείται κατά τη σύνθεση.

Στην SystemVerilog, η σύγκριση με χρήση των τελεστών == ή != υποδεικνύει εσφαλμένα ισότητα εάν ένας από τους τελεστέους είναι x ή z. Αντ' αυτών θα πρέπει να χρησιμοποιούνται οι τελεστές === και !== σε testbenches, επειδή δουλεύουν σωστά με τα σήματα x και z.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity testbench2 is -- no inputs or outputs
end;

architecture sim of testbench2 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
         y:       out STD_LOGIC);
  end component;
  signal a, b, c, y: STD_LOGIC;
begin
  -- instantiate device under test
  dut: sillyfunction port map(a, b, c, y);

  -- apply inputs one at a time
  -- checking results
  process begin
    a <= '0'; b <= '0'; c <= '0'; wait for 10 ns;
      assert y = '1' report "000 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "001 failed.";
    b <= '1'; c <= '0';           wait for 10 ns;
      assert y = '0' report "010 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "011 failed.";
    a <= '1'; b <= '0'; c <= '0'; wait for 10 ns;
      assert y = '1' report "100 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '1' report "101 failed.";
    b <= '1'; c <= '0';           wait for 10 ns;
      assert y = '0' report "110 failed.";
    c <= '1';                     wait for 10 ns;
      assert y = '0' report "111 failed.";
    wait; -- wait forever
  end process;
end;
```

Η εντολή `assert` ελέγχει μια συνθήκη και εκτυπώνει το μήνυμα που δίνεται στον όρο report εάν η συνθήκη δεν ικανοποιείται. Η `assert` αγνοείται κατά τη σύνθεση.

Η συγγραφή κώδικα για κάθε διάνυσμα ελέγχου είναι επίσης κουραστική υπόθεση, ιδιαίτερα για λειτουργικές μονάδες που απαιτούν μεγάλο αριθμό διανυσμάτων. Μια καλύτερη προσέγγιση είναι να τοποθετείτε τα διανύσματα ελέγχου σ' ένα ξεχωριστό αρχείο. Το testbench διαβάζει απλώς τα διανύσματα ελέγχου, εφαρμόζει το διάνυσμα εισόδου, περιμένει, ελέγχει ότι οι τιμές εξόδου ταιριάζουν με το διάνυσμα εξόδου και επαναλαμβάνει τη διαδικασία έως ότου να φτάσει στο τέλος του αρχείου.

Το Παράδειγμα Α.49 επιδεικνύει ένα τέτοιο πρόγραμμα testbench. Το πρόγραμμα παράγει ένα σήμα ρολογιού χρησιμοποιώντας μια εντολή `always/process` χωρίς λίστα διεγέρσεων, έτσι ώστε να αποτιμάται σε διαρκή επανάληψη. Στην αρχή της προσομοίωσης, το πρόγραμμα διαβάζει τα διανύσματα ελέγχου

από ένα αρχείο και δίνει σήμα reset για δύο κύκλους. Το example.tv είναι ένα αρχείο κειμένου, το οποίο περιέχει τις εισόδους και τις αναμενόμενες εξόδους σε δυαδική μορφή:

```
000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
```

Νέες είσοδοι εφαρμόζονται στην ανοδική ακμή του ρολογιού και η έξοδος ελέγχεται στην καθοδική ακμή του ρολογιού. Αυτό το σήμα ρολογιού (και το reset) θα έπρεπε επίσης να παρέχονται στη μονάδα DUT, εάν η υπό έλεγχο μονάδα χρησιμοποιεί ακολουθιακή λογική. Τα σφάλματα αναφέρονται τη στιγμή που συμβαίνουν. Στο τέλος της προσομοίωσης, το πρόγραμμα testbench εκτυπώνει το συνολικό αριθμό διανυσμάτων ελέγχου που εφαρμόστηκαν και το πλήθος των σφαλμάτων που ανιχνεύτηκαν.

Το συγκεκριμένο testbench είναι υπερβολικό για ένα τόσο απλό κύκλωμα. Ωστόσο, μπορεί εύκολα να τροποποιηθεί ώστε να ελέγχει πολυπλοκότερα κυκλώματα, αλλάζοντας κατάλληλα το αρχείο example.tv, δημιουργώντας το υπόδειγμα για τη νέα μονάδα DUT και προσαρμόζοντας μερικές γραμμές κώδικα για τον ορισμό των εισόδων και τον έλεγχο των εξόδων.

Παράδειγμα Α.49 Πρόγραμμα Testbench με Δυνατότητα Αυτο-Ελέγχου

SystemVerilog

```
module testbench3();
  logic        clk, reset;
  logic        a, b, c, yexpected;
  logic        y;
  logic [31:0] vectornum, errors;
  logic [3:0]  testvectors[10000:0];

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // generate clock
  always
    begin
      clk = 1; #5; clk = 0; #5;
    end

  // at start of test, load vectors
  // and pulse reset
  initial
    begin
      $readmemb("example.tv", testvectors);
      vectornum = 0; errors = 0;
      reset = 1; #27; reset = 0;
    end

  // apply test vectors on rising edge of clk
  always @(posedge clk)
    begin
      #1; {a, b, c, yexpected} =
            testvectors[vectornum];
    end
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use STD.TEXTIO.all;

entity testbench3 is -- no inputs or outputs
end;

architecture sim of testbench3 is
  component sillyfunction
    port(a, b, c: in  STD_LOGIC;
         y:       out STD_LOGIC);
  end component;
  signal a, b, c, y: STD_LOGIC;
  signal clk, reset: STD_LOGIC;
  signal yexpected: STD_LOGIC;
  constant MEMSIZE: integer := 10000;
  type tvarray is array(MEMSIZE downto 0) of
    STD_LOGIC_VECTOR(3 downto 0);
  signal testvectors: tvarray;
  shared variable vectornum, errors: integer;
begin
  -- instantiate device under test
  dut: sillyfunction port map(a, b, c, y);

  -- generate clock
  process begin
    clk <= '1'; wait for 5 ns;
    clk <= '0'; wait for 5 ns;
  end process;

  -- at start of test, load vectors
  -- and pulse reset
```


SystemVerilog (συνέχεια)

```
// check results on falling edge of clk
  always @(negedge clk)
   if (~reset) begin // skip during reset
     if (y !== yexpected) begin
       $display("Error: inputs = %b", {a, b, c});
       $display("  outputs = %b (%b expected)",
                y, yexpected);
       errors = errors + 1;
     end
     vectornum = vectornum + 1;
     if (testvectors[vectornum] === 'bx) begin
       $display("%d tests completed with %d
                 errors", vectornum, errors);
       $finish;
     end
   end
endmodule
```

Η `$readmemb` διαβάζει ένα αρχείο δυαδικών αριθμών σε μια διάταξη (array). Η `$readmemh` είναι παρόμοια, αλλά διαβάζει ένα αρχείο δεκαεξαδικών αριθμών.

Το επόμενο μπλοκ κώδικα αναμένει για μία μονάδα χρόνου μετά από την ανοδική ακμή του ρολογιού (για την αποφυγή ενδεχόμενων μπερδεμάτων σε περίπτωση που αλλάζουν ταυτόχρονα το ρολόι και τα δεδομένα), και κατόπιν ορίζει τις τρεις εισόδους και την αναμενόμενη έξοδο βάσει των 4 bits του τρέχοντος διανύσματος ελέγχου.

Η `$display` εκτελεί μια εργασία σε επίπεδο συστήματος, για την εκτύπωση του παραθύρου του προσομοιωτή. Η `$finish` τερματίζει την προσομοίωση.

Σημειώστε ότι αν και η λειτουργική μονάδα της SystemVerilog υποστηρίζει έως 10001 διανύσματα ελέγχου, θα τερματίσει την προσομοίωση αφού εκτελεστούν τα 8 διανύσματα που περιέχει το αρχείο.

Για περισσότερες πληροφορίες σχετικά με τα προγράμματα testbench και τις διαδικασίες επαλήθευσης με την SystemVerilog, συμβουλευτείτε το [Bergeron05].

VHDL (συνέχεια)

```
process is
    file tv: TEXT;
    variable i, j: integer;
    variable L: line;
    variable ch: character;
  begin
    -- read file of test vectors
    i := 0;
    FILE_OPEN(tv, "example.tv", READ_MODE);
    while not endfile(tv) loop
      readline(tv, L);
      for j in 0 to 3 loop
        read(L, ch);
        if (ch = '_') then read(L, ch);
        end if;
        if (ch = '0') then
          testvectors(i)(j) <= '0';
        else testvectors(i)(j) <= '1';
        end if;
      end loop;
      i := i + 1;
    end loop;

    vectornum := 0; errors := 0;
    reset <= '1'; wait for 27 ns; reset <= '0';
    wait;
  end process;
  -- apply test vectors on rising edge of clk
  process (clk) begin
    if (clk'event and clk = '1') then

      a <= testvectors(vectornum)(0) after 1 ns;
      b <= testvectors(vectornum)(1) after 1 ns;
      c <= testvectors(vectornum)(2) after 1 ns;
      yexpected <= testvectors(vectornum)(3)
        after 1 ns;
    end if;
  end process;

  -- check results on falling edge of clk
  process (clk) begin
    if (clk'event and clk = '0' and reset = '0') then
      assert y = yexpected
        report "Error: y = " & STD_LOGIC'image(y);
      if (y /= yexpected) then
        errors := errors + 1;
      end if;
      vectornum := vectornum + 1;
      if (is_x(testvectors(vectornum))) then
        if (errors = 0) then
          report "Just kidding -- " &
                 integer'image(vectornum) &
                 " tests completed successfully."
                 severity failure;
```

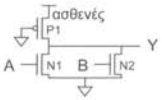
(συνεχίζεται)

Παράδειγμα Α.49 Πρόγραμμα Testbench με Αρχείο Διανυσμάτων Ελέγχου (συνέχεια)

VHDL (συνέχεια)

```
        else
          report integer'image(vectornum) &
                 " tests completed, errors = " &
                 integer'image(errors)
                 severity failure;
        end if;
      end if;
    end if;
  end process;
end;
```

Ο VHDL κώδικας είναι μάλλον άκομψος και χρησιμοποιεί εντολές ανάγνωσης αρχείων των οποίων η παρουσίαση υπερβαίνει τα όρια του παρόντος παραρτήματος. Παρά ταύτα, δίνει μια καλή αίσθηση του πώς δείχνει ένα πρόγραμμα testbench με δυνατότητα αυτο-ελέγχου.



ΣΧΗΜΑ Α.44 Ψευδο-nMOS πύλη NOR.

Α.11 Περιγραφές Netlist της SystemVerilog

Όπως αναφέραμε στην Ενότητα 1.8.4, η Verilog παρέχει πρωταρχικές οντότητες (primitives) σε επίπεδο τρανζίστορ και σε επίπεδο πυλών, οι οποίες είναι χρήσιμες για την περιγραφή των netlist. Η VHDL δεν διαθέτει ανάλογες δυνατότητες.

Στις πρωταρχικές πύλες περιλαμβάνονται οι not, and, or, xor, nand, nor και xnor. Η έξοδος δηλώνεται πρώτη και μπορούν να ακολουθούν πολλαπλές είσοδοι. Για παράδειγμα, μια πύλη AND 4 εισόδων μπορεί να καθορίζεται ως

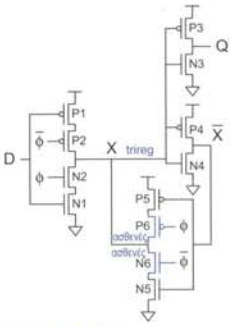
```
and g1(y, a, b, c, d);
```

Στα πρωταρχικά τρανζίστορ περιλαμβάνονται τα `tranif1`, `tranif0`, `rtranif1` και `rtranif0`. Το `tranif1` είναι nMOS (δηλαδή, ένα τρανζίστορ που άγει όταν η πύλη είναι '1'), ενώ το `tranif0` είναι pMOS. Τα rtranif είναι ωμικά τρανζίστορ· δηλαδή, ασθενή τρανζίστορ στα οποία μπορεί να υπερισχύσει ένας ισχυρότερος οδηγός. Οι λογικές τιμές 0 και 1 (GND και V_{DD}) καθορίζονται με τους τύπους `supply0` και `supply1`. Για παράδειγμα, μια ψευδο-nMOS πύλη NOR του Σχήματος Α.44 με ασθενές δίκτυο οδήγησης πάνω μοντελοποιείται με τρία τρανζίστορ. Σημειώστε ότι το y πρέπει να δηλωθεί ως δίκτυο tri, επειδή θα μπορούσε να οδηγείται από πολλαπλά τρανζίστορ.

```
module nor_pseudonmos(input  logic a, b,
                      output tri   y);

  supply0 gnd;
  supply1 vdd;

  tranif1  n1(y, gnd, a);
  tranif1  n2(y, gnd, b);
  rtranif0 p1(y, vdd, gnd);
endmodule
```

Η μοντελοποίηση ενός μανδαλωτή στην Verilog απαιτεί ιδιαίτερη προσοχή, επειδή το μονοπάτι ανάδρασης είναι ON την ίδια στιγμή που το μονοπάτι πρόσθιας τροφοδότησης είναι OFF, καθώς ο μανδαλωτής γίνεται αδιαφανής. Ανάλογα με τις συνθήκες ανταγωνισμού, υπάρχει κίνδυνος ο κόμβος κατάστασης να μείνει αιωρούμενος ή να αντιμετωπίσει συνθήκες διαμάχης. Για την λύση αυτού του προβλήματος, ο κόμβος κατάστασης μοντελοποιείται ως `trireg` (έτσι ώστε να μην μείνει αιωρούμενος) και τα τρανζίστορ ανάδρασης μοντελοποιούνται ως ασθενή (έτσι ώστε να μην προκαλούν συνθήκες διαμάχης). Οι άλλοι κόμβοι είναι δίκτυα tri επειδή μπορούν να οδηγούνται από πολλαπλά τρανζίστορ. Το Σχήμα Α.45 επανασχεδιάζει το μανδαλωτή του Σχήματος 10.17(στ) σε επίπεδο τρανζίστορ και επισημαίνει τα ασθενή τρανζίστορ και τον κόμβο κατάστασης.


```
module latch(input  logic ph, phb, d,
             output tri   q);

  trireg  x;
  tri     xb, nn12, nn56, pp12, pp56;
  supply0 gnd;
  supply1 vdd;

  // input stage
  tranif1 n1(nn12, gnd, d);
  tranif1 n2(x, nn12, ph);
  tranif0 p1(pp12, vdd, d);
  tranif0 p2(x, pp12, phb);

  // output inverter
  tranif1 n3(q, gnd, x);
  tranif0 p3(q, vdd, x);

  // xb inverter
  tranif1 n4(xb, gnd, x);
  tranif0 p4(xb, vdd, x);

  // feedback tristate
  tranif1  n5(nn56, gnd, xb);
  rtranif1 n6(x, nn56, phb);
  tranif0  p5(pp56, vdd, xb);
  rtranif0 p6(x, pp56, ph);
endmodule
```



ΣΧΗΜΑ Α.45 Ο μανδαλωτής.

Τα περισσότερα εργαλεία σύνθεσης εκτελούν αντιστοίχιση μόνο σε πύλες, όχι σε τρανζίστορ, οπότε τα πρωταρχικά τρανζίστορ είναι μόνο για σκοπούς προσομοίωσης.

Τα tranif είναι στοιχεία δύο κατευθύνσεων· δηλαδή, η πηγή και η υποδοχή είναι συμμετρικές. Η Verilog υποστηρίζει επίσης μονοκατευθυντικά πρωταρχικά τρανζίστορ nmos και pmos, τα οποία επιτρέπουν σ' ένα σήμα να ρέει μόνο από τον ακροδέκτη εισόδου στον ακροδέκτη εξόδου. Επειδή τα πραγματικά τρανζίστορ είναι εκ φύσεως δύο κατευθύνσεων, τα μονοκατευθυντικά μοντέλα μπορεί να οδηγήσουν σε περιπτώσεις όπου η προσομοίωση δεν καταφέρνει να εντοπίσει σφάλματα τα οποία θα υπήρχαν σε πραγματικό hardware. Λόγω αυτού, τα πρωταρχικά στοιχεία tranif προτιμώνται για την προσομοίωση.

Α.12 Παράδειγμα: Ο Επεξεργαστής MIPS

Σαν ένα πιο ρεαλιστικό παράδειγμα σχεδίασης με χρήση HDL, σ' αυτή την ενότητα θα παρουσιάσουμε τον κώδικα και το πρόγραμμα testbench για τον επεξεργαστή MIPS που περιγράψαμε στο Κεφάλαιο 1. Το παράδειγμα χειρίζεται μόνο τις εντολές LB, SB, ADD, SUB, AND, OR, SLT, BEQ και J. Χρησιμοποιεί μια 8-bit μονάδα επεξεργασίας δεδομένων και μόνο οκτώ καταχωρητές. Επειδή οι εντολές έχουν εύρος 32 bits, φορτώνονται σε τέσσερις διαδοχικούς κύκλους προσκόμισης, μέσω ενός 8-bit μονοπατιού, σε εξωτερική μνήμη.

A.12.1 Το Πρόγραμμα Testbench

Το πρόγραμμα testbench αρχικοποιεί μια μνήμη των 256 bytes με εντολές και δεδομένα από ένα αρχείο κειμένου. Ο κώδικας ελέγχει κάθε μία από τις εντολές. Στη συνέχεια παρουσιάζονται το αρχείο mipstest.asm (σε assembly) και το αρχείο κειμένου memfile.dat. Το πρόγραμμα testbench τρέχει έως ότου παρατηρήσει μια εγγραφή στη μνήμη. Εάν γραφεί η τιμή 7 στη διεύθυνση 76, ο κώδικας πιθανώς έτρεξε σωστά. Εάν όλα πάνε καλά, το testbench θα πρέπει να χρειαστεί 100 κύκλους (1000 ns) για να τρέξει.

```
# mipstest.asm
# 9/16/03 David Harris David_Harris@hmc.edu
#
# Test MIPS instructions.  Assumes little-endian memory was
# initialized as:
# word 16: 3
# word 17: 5
# word 18: 12

main:     #Assembly Code       effect                  Machine Code
          lb $2, 68($0)        # initialize $2 = 5      80020044
          lb $7, 64($0)        # initialize $7 = 3      80070040
          lb $3, 69($7)        # initialize $3 = 12     80e30045
          or $4, $7, $2        # $4 <= 3 or 5 = 7       00e22025
          and $5, $3, $4       # $5 <= 12 and 7 = 4     00642824
          add $5, $5, $4       # $5 <= 4 + 7 = 11       00a42820
          beq $5, $7, end      # shouldn't be taken     10a70008
          slt $6, $3, $4       # $6 <= 12 < 7 = 0       0064302a
          beq $6, $0, around   # should be taken        10c00001
          lb $5, 0($0)         # shouldn't happen       80050000
around:   slt $6, $7, $2       # $6 <= 3 < 5 = 1        00e2302a
          add $7, $6, $5       # $7 <= 1 + 11 = 12      00c53820
          sub $7, $7, $2       # $7 <= 12 - 5 = 7       00e23822
          j end                # should be taken        0800000f
          lb $7, 0($0)         # shouldn't happen       80070000
end:      sb $7, 71($2)        # write adr 76 <= 7      a0470047
          .dw 3                                       00000003
          .dw 5                                       00000005
          .dw 12                                      0000000c
```

```
memfile.dat
80020044
80070040
80e30045
00e22025
00642824
00a42820
10a70008
0064302a
10c00001
80050000
00e2302a
00c53820
00e23822
0800000f
80070000
a0470047
00000003
00000005
0000000c
```


A.12.2 SystemVerilog

```
//------------------------------------------------
// mips.sv
// Max Yi (byyi@hmc.edu) and
//   David_Harris@hmc.edu 12/9/03
// Changes 7/3/07 DMH
//   Updated to SystemVerilog
//   fixed memory endian bug
//
// Model of subset of MIPS processor from Ch 1
//  note that no sign extension is done because
//  width is only 8 bits
//---------------------

// states and instructions

  typedef enum logic [3:0]
    {FETCH1 = 4'b0000, FETCH2, FETCH3, FETCH4,
     DECODE, MEMADR, LBRD, LBWR, SBWR,
     RTYPEEX, RTYPEWR, BEQEX, JEX} statetype;
  typedef enum logic [5:0] {LB    = 6'b100000,
                            SB    = 6'b101000,
                            RTYPE = 6'b000000,
                            BEQ   = 6'b000100,
                            J     = 6'b000010} opcode;
  typedef enum logic [5:0] {ADD = 6'b100000,
                            SUB = 6'b100010,
                            AND = 6'b100100,
                            OR  = 6'b100101,
                            SLT = 6'b101010} functcode;

// testbench
module testbench #(parameter WIDTH = 8, REGBITS = 3)();

  logic             clk;
  logic             reset;
  logic             memread, memwrite;
  logic [WIDTH-1:0] adr, writedata;
  logic [WIDTH-1:0] memdata;

  // instantiate devices to be tested
  mips #(WIDTH,REGBITS) dut(clk, reset, memdata, memread,
                            memwrite, adr, writedata);

  // external memory for code and data
  exmemory #(WIDTH) exmem(clk, memwrite, adr, writedata, memdata);

  // initialize test
  initial
    begin
      reset <= 1; # 22; reset <= 0;
    end

  // generate clock to sequence tests
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5;
    end
```

```
    always @(negedge clk)
      begin
        if(memwrite)
          assert(adr == 76 & writedata == 7)
            $display("Simulation completely successful");
          else $error("Simulation failed");
      end
  endmodule

  // external memory accessed by MIPS
  module exmemory #(parameter WIDTH = 8)
                   (input  logic             clk,
                    input  logic             memwrite,
                    input  logic [WIDTH-1:0] adr, writedata,
                    output logic [WIDTH-1:0] memdata);

    logic [31:0]       mem [2**(WIDTH-2)-1:0];
    logic [31:0]       word;
    logic [1:0]        bytesel;
    logic [WIDTH-2:0] wordadr;

    initial
      $readmemh("memfile.dat", mem);

    assign bytesel = adr[1:0];
    assign wordadr = adr[WIDTH-1:2];

    // read and write bytes from 32-bit word
    always @(posedge clk)
      if(memwrite)
        case (bytesel)
          2'b00: mem[wordadr][7:0]   <= writedata;
          2'b01: mem[wordadr][15:8]  <= writedata;
          2'b10: mem[wordadr][23:16] <= writedata;
          2'b11: mem[wordadr][31:24] <= writedata;
        endcase

    assign word = mem[wordadr];
    always_comb
      case (bytesel)
        2'b00: memdata = word[7:0];
        2'b01: memdata = word[15:8];
        2'b10: memdata = word[23:16];
        2'b11: memdata = word[31:24];
      endcase
  endmodule

  // simplified MIPS processor
  module mips #(parameter WIDTH = 8, REGBITS = 3)
              (input  logic             clk, reset,
               input  logic [WIDTH-1:0] memdata,
               output logic             memread, memwrite,
               output logic [WIDTH-1:0] adr, writedata);

    logic [31:0] instr;
    logic        zero, alusrca, memtoreg, iord, pcen,
                 regwrite, regdst;
    logic [1:0]  pcsrc, alusrcb;
    logic [3:0]  irwrite;
```



```
    logic [2:0]  alucontrol;
    logic [5:0]  op, funct;

    assign op = instr[31:26];
    assign funct = instr[5:0];

    controller cont(clk, reset, op, funct, zero, memread, memwrite,
                    alusrca, memtoreg, iord, pcen, regwrite, regdst,
                    pcsrc, alusrcb, alucontrol, irwrite);
    datapath    #(WIDTH, REGBITS)
      dp(clk, reset, memdata, alusrca, memtoreg, iord, pcen,
         regwrite, regdst, pcsrc, alusrcb, irwrite, alucontrol,
         zero, instr, adr, writedata);
  endmodule

  module controller(input logic clk, reset,
                    input  logic [5:0] op, funct,
                    input  logic       zero,
                    output logic       memread, memwrite, alusrca,
                    output logic       memtoreg, iord, pcen,
                    output logic       regwrite, regdst,
                    output logic [1:0] pcsrc, alusrcb,
                    output logic [2:0] alucontrol,
                    output logic [3:0] irwrite);

    statetype       state;
    logic           pcwrite, branch;
    logic     [1:0] aluop;

    // control FSM
    statelogic statelog(clk, reset, op, state);
    outputlogic outputlog(state, memread, memwrite, alusrca,
                          memtoreg, iord,
                          regwrite, regdst, pcsrc, alusrcb, irwrite,
                          pcwrite, branch, aluop);

    // other control decoding
    aludec  ac(aluop, funct, alucontrol

    // program counter enable
    assign pcen = pcwrite | (branch & zero);
  endmodule

  module statelogic(input  logic       clk, reset,
                    input  logic [5:0] op,
                    output statetype   state);

    statetype nextstate;

    always_ff @(posedge clk)
      if (reset) state <= FETCH1;
      else       state <= nextstate;

    always_comb
      begin
        case (state)
          FETCH1:  nextstate = FETCH2;
          FETCH2:  nextstate = FETCH3;
          FETCH3:  nextstate = FETCH4;
```

```
          FETCH4:  nextstate = DECODE;
          DECODE:  case(op)
                     LB:      nextstate = MEMADR;
                     SB:      nextstate = MEMADR;
                     RTYPE:   nextstate = RTYPEEX;
                     BEQ:     nextstate = BEQEX;
                     J:       nextstate = JEX;
                     default: nextstate = FETCH1;
                              // should never happen
                   endcase
          MEMADR:  case(op)
                     LB:      nextstate = LBRD;
                     SB:      nextstate = SBWR;
                     default: nextstate = FETCH1;
                              // should never happen
                   endcase
          LBRD:    nextstate = LBWR;
          LBWR:    nextstate = FETCH1;
          SBWR:    nextstate = FETCH1;
          RTYPEEX: nextstate = RTYPEWR;
          RTYPEWR: nextstate = FETCH1;
          BEQEX:   nextstate = FETCH1;
          JEX:     nextstate = FETCH1;
          default: nextstate = FETCH1;
                   // should never happen
        endcase
      end
endmodule

module outputlogic(input statetype state,
                   output logic       memread, memwrite, alusrca,
                   output logic       memtoreg, iord,
                   output logic       regwrite, regdst,
                   output logic [1:0] pcsrc, alusrcb,
                   output logic [3:0] irwrite,
                   output logic       pcwrite, branch,
                   output logic [1:0] aluop);

  always_comb
    begin
      // set all outputs to zero, then
      // conditionally assert just the appropriate ones
      irwrite = 4'b0000;
      pcwrite = 0; branch = 0;
      regwrite = 0; regdst = 0;
      memread = 0; memwrite = 0;
      alusrca = 0; alusrcb = 2'b00; aluop = 2'b00;
      pcsrc = 2'b00;
      iord = 0; memtoreg = 0;

      case (state)
        FETCH1:
          begin
            memread = 1;
            irwrite = 4'b0001;
            alusrcb = 2'b01;
            pcwrite = 1;
          end
```



```
        FETCH2:
          begin
            memread = 1;
            irwrite = 4'b0010;
            alusrcb = 2'b01;
            pcwrite = 1;
          end
        FETCH3:
          begin
            memread = 1;
            irwrite = 4'b0100;
            alusrcb = 2'b01;
            pcwrite = 1;
          end
        FETCH4:
          begin
            memread = 1;
            irwrite = 4'b1000;
            alusrcb = 2'b01;
            pcwrite = 1;
          end
        DECODE: alusrcb = 2'b11;
        MEMADR:
          begin
            alusrca = 1;
            alusrcb = 2'b10;
          end
        LBRD:
          begin
            memread = 1;
            iord    = 1;
          end
        LBWR:
          begin
            regwrite = 1;
            memtoreg = 1;
          end
        SBWR:
          begin
            memwrite = 1;
            iord     = 1;
          end
        RTYPEEX:
          begin
            alusrca = 1;
            aluop   = 2'b10;
          end
        RTYPEWR:
          begin
            regdst   = 1;
            regwrite = 1;
          end
        BEQEX:
          begin
            alusrca = 1;
            aluop   = 2'b01;
            branch  = 1;
            pcsrc   = 2'b01;
          end
```

```
        JEX:
          begin
            pcwrite  = 1;
            pcsrc    = 2'b10;
          end
      endcase
    end
endmodule

module aludec(input  logic [1:0] aluop,
              input  logic [5:0] funct,
              output logic [2:0] alucontrol);

  always_comb
    case (aluop)
      2'b00: alucontrol = 3'b010;  // add for lb/sb/addi
      2'b01: alucontrol = 3'b110;  // subtract (for beq)
      default: case(funct)       // R-Type instructions
                 ADD: alucontrol = 3'b010;
                 SUB: alucontrol = 3'b110;
                 AND: alucontrol = 3'b000;
                 OR:  alucontrol = 3'b001;
                 SLT: alucontrol = 3'b111;
                 default:   alucontrol = 3'b101;
                            // should never happen
               endcase
    endcase
endmodule

module datapath #(parameter WIDTH = 8, REGBITS = 3)
                 (input  logic             clk, reset,
                  input  logic [WIDTH-1:0] memdata,
                  input  logic             alusrca, memtoreg, iord,
                  input  logic             pcen, regwrite, regdst,
                  input  logic [1:0]       pcsrc, alusrcb,
                  input  logic [3:0]       irwrite,
                  input  logic [2:0]       alucontrol,
                  output logic             zero,
                  output logic [31:0]      instr,
                  output logic [WIDTH-1:0] adr, writedata);

  logic [REGBITS-1:0] ra1, ra2, wa;
  logic [WIDTH-1:0]   pc, nextpc, data, rd1, rd2, wd, a, srca,
                      srcb, aluresult, aluout, immx4;

  logic [WIDTH-1:0] CONST_ZERO = 0;
  logic [WIDTH-1:0] CONST_ONE =  1;

  // shift left immediate field by 2
  assign immx4 = {instr[WIDTH-3:0],2'b00};

  // register file address fields
  assign ra1 = instr[REGBITS+20:21];
  assign ra2 = instr[REGBITS+15:16];
  mux2       #(REGBITS) regmux(instr[REGBITS+15:16],
                               instr[REGBITS+10:11], regdst, wa);

  // independent of bit width,
  // load instruction into four 8-bit registers over four cycles
```



```
  flopen     #(8)      ir0(clk, irwrite[0], memdata[7:0], instr[7:0]);
  flopen     #(8)      ir1(clk, irwrite[1], memdata[7:0], instr[15:8]);
  flopen     #(8)      ir2(clk, irwrite[2], memdata[7:0], instr[23:16]);
  flopen     #(8)      ir3(clk, irwrite[3], memdata[7:0], instr[31:24]);

  // datapath
  flopenr    #(WIDTH)  pcreg(clk, reset, pcen, nextpc, pc);
  flop       #(WIDTH)  datareg(clk, memdata, data);
  flop       #(WIDTH)  areg(clk, rd1, a);
  flop       #(WIDTH)  wrdreg(clk, rd2, writedata);
  flop       #(WIDTH)  resreg(clk, aluresult, aluout);
  mux2       #(WIDTH)  adrmux(pc, aluout, iord, adr);
  mux2       #(WIDTH)  src1mux(pc, a, alusrca, srca);
  mux4       #(WIDTH)  src2mux(writedata, CONST_ONE, instr[WIDTH-1:0],
                               immx4, alusrcb, srcb);
  mux3       #(WIDTH)  pcmux(aluresult, aluout, immx4,
                             pcsrc, nextpc);
  mux2       #(WIDTH)  wdmux(aluout, data, memtoreg, wd);
  regfile    #(WIDTH,REGBITS) rf(clk, regwrite, ra1, ra2,
                                 wa, wd, rd1, rd2);
  alu        #(WIDTH)  alunit(srca, srcb, alucontrol, aluresult, zero);
endmodule

module alu #(parameter WIDTH = 8)
            (input  logic [WIDTH-1:0] a, b,
             input  logic [2:0]       alucontrol,
             output logic [WIDTH-1:0] result,
             output logic             zero);

  logic [WIDTH-1:0] b2, andresult, orresult,
                    sumresult, sltresult;

  andN    andblock(a, b, andresult);
  orN     orblock(a, b, orresult);
  condinv binv(b, alucontrol[2], b2);
  adder   addblock(a, b2, alucontrol[2], sumresult);
  // slt should be 1 if most significant bit of sum is 1
  assign sltresult = sumresult[WIDTH-1];

  mux4 resultmux(andresult, orresult, sumresult,
                 sltresult, alucontrol[1:0], result);
  zerodetect #(WIDTH) zd(result, zero);
endmodule

module regfile #(parameter WIDTH = 8, REGBITS = 3)
                (input  logic             clk,
                 input  logic             regwrite,
                 input  logic [REGBITS-1:0] ra1, ra2, wa,
                 input  logic [WIDTH-1:0]   wd,
                 output logic [WIDTH-1:0]   rd1, rd2);

   logic [WIDTH-1:0] RAM [2**REGBITS-1:0];

  // three ported register file
  // read two ports combinationally
  // write third port on rising edge of clock
  // register 0 hardwired to 0
  always @(posedge clk)
    if (regwrite) RAM[wa] <= wd;
```

```
  assign rd1 = ra1 ? RAM[ra1] : 0;
  assign rd2 = ra2 ? RAM[ra2] : 0;
endmodule

module zerodetect #(parameter WIDTH = 8)
                   (input  logic [WIDTH-1:0] a,
                    output logic             y);

  assign y = (a==0);
endmodule

module flop #(parameter WIDTH = 8)
             (input  logic             clk,
              input  logic [WIDTH-1:0] d,
              output logic [WIDTH-1:0] q);

  always_ff @(posedge clk)
    q <= d;
endmodule

module flopen #(parameter WIDTH = 8)
               (input  logic             clk, en,
                input  logic [WIDTH-1:0] d,
                output logic [WIDTH-1:0] q);

  always_ff @(posedge clk)
    if (en) q <= d;
endmodule

module flopenr #(parameter WIDTH = 8)
                (input  logic             clk, reset, en,
                 input  logic [WIDTH-1:0] d,
                 output logic [WIDTH-1:0] q);

  always_ff @(posedge clk)
    if      (reset) q <= 0;
    else if (en)    q <= d;
endmodule

module mux2 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1,
              input  logic             s,
              output logic [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule

module mux3 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1, d2,
              input  logic [1:0]       s,
              output logic [WIDTH-1:0] y);

  always_comb
    casez (s)
      2'b00: y = d0;
      2'b01: y = d1;
      2'b1?: y = d2;
    endcase
endmodule
```



```
module mux4 #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] d0, d1, d2, d3,
              input  logic [1:0]       s,
              output logic [WIDTH-1:0] y);

  always_comb
    case (s)
      2'b00: y = d0;
      2'b01: y = d1;
      2'b10: y = d2;
      2'b11: y = d3;
    endcase
endmodule

module andN #(parameter WIDTH = 8)
             (input  logic [WIDTH-1:0] a, b,
              output logic [WIDTH-1:0] y);

  assign y = a & b;
endmodule

module orN #(parameter WIDTH = 8)
            (input  logic [WIDTH-1:0] a, b,
             output logic [WIDTH-1:0] y);

  assign y = a | b;
endmodule

module inv #(parameter WIDTH = 8)
            (input  logic [WIDTH-1:0] a,
             output logic [WIDTH-1:0] y);

  assign y = ~a;
endmodule

module condinv #(parameter WIDTH = 8)
                (input  logic [WIDTH-1:0] a,
                 input  logic             invert,
                 output logic [WIDTH-1:0] y);

  logic [WIDTH-1:0] ab;

  inv  inverter(a, ab);
  mux2 invmux(a, ab, invert, y);
endmodule

module adder #(parameter WIDTH = 8)
              (input  logic [WIDTH-1:0] a, b,
               input  logic             cin,
               output logic [WIDTH-1:0] y);

  assign y = a + b + cin;
endmodule
```

A.12.3 VHDL

```
------------------------------------------------------------
-- mips.vhd
-- David_Harris@hmc.edu 9/9/03
-- Model of subset of MIPS processor described in Ch 1
------------------------------------------------------------

------------------------------------------------------------
-- Entity Declarations
------------------------------------------------------------

library IEEE; use IEEE.STD_LOGIC_1164.all; use IEEE.STD_LOGIC_UNSIGNED.all;
entity top is -- top-level design for testing
    generic(width:   integer := 8;      -- default 8-bit datapath
            regbits: integer := 3);     -- and 3 bit register addresses (8 regs)
end;

library IEEE; use IEEE.STD_LOGIC_1164.all; use STD.TEXTIO.all;
use IEEE.STD_LOGIC_UNSIGNED.all;  use IEEE.STD_LOGIC_ARITH.all;
entity memory is -- external memory accessed by MIPS
    generic(width: integer);
    port(clk, memwrite:  in STD_LOGIC;
         adr, writedata: in STD_LOGIC_VECTOR(width-1 downto 0);
         memdata:        out STD_LOGIC_VECTOR(width-1 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mips is -- simplified MIPS processor
    generic(width:   integer := 8;      -- default 8-bit datapath
            regbits: integer := 3);     -- and 3 bit register addresses (8 regs)
    port(clk, reset:        in  STD_LOGIC;
         memdata:           in  STD_LOGIC_VECTOR(width-1 downto 0);
         memread, memwrite: out STD_LOGIC;
         adr, writedata:    out STD_LOGIC_VECTOR(width-1 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity controller is -- control FSM
    port(clk, reset:                    in  STD_LOGIC;
         op:                            in  STD_LOGIC_VECTOR(5 downto 0);
         zero:                          in  STD_LOGIC;
         memread, memwrite, alusrca, memtoreg,
         iord, pcen, regwrite, regdst: out STD_LOGIC;
         pcsrc, alusrcb, aluop:     out STD_LOGIC_VECTOR(1 downto 0);
         irwrite:                       out STD_LOGIC_VECTOR(3 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity alucontrol is -- ALU control decoder
    port(aluop:   in  STD_LOGIC_VECTOR(1 downto 0);
         funct:   in  STD_LOGIC_VECTOR(5 downto 0);
         alucont: out STD_LOGIC_VECTOR(2 downto 0));
end;
```



```
library IEEE; use IEEE.STD_LOGIC_1164.all; use IEEE.STD_LOGIC_ARITH.all;
entity datapath is  -- MIPS datapath
    generic(width, regbits: integer);
    port(clk, reset:        in  STD_LOGIC;
         memdata:           in  STD_LOGIC_VECTOR(width-1 downto 0);
         alusrca, memtoreg, iord, pcen,
         regwrite, regdst:  in  STD_LOGIC;
         pcsrc, alusrcb:    in  STD_LOGIC_VECTOR(1 downto 0);
         irwrite:           in  STD_LOGIC_VECTOR(3 downto 0);
         alucont:           in  STD_LOGIC_VECTOR(2 downto 0);
         zero:              out STD_LOGIC;
         instr:             out STD_LOGIC_VECTOR(31 downto 0);
         adr, writedata:    out STD_LOGIC_VECTOR(width-1 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all; use IEEE.STD_LOGIC_UNSIGNED.all;
entity alu is -- Arithmetic/Logic unit with add/sub, AND, OR, set less than
    generic(width: integer);
    port(a, b:    in  STD_LOGIC_VECTOR(width-1 downto 0);
         alucont: in  STD_LOGIC_VECTOR(2 downto 0);
         result:  out STD_LOGIC_VECTOR(width-1 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all; use IEEE.STD_LOGIC_ARITH.all;
entity regfile is -- three-port register file of 2**regbits words x width bits
    generic(width, regbits: integer);
    port(clk:          in  STD_LOGIC;
         write:        in  STD_LOGIC;
         ra1, ra2, wa: in  STD_LOGIC_VECTOR(regbits-1 downto 0);
         wd:           in  STD_LOGIC_VECTOR(width-1 downto 0);
         rd1, rd2:     out STD_LOGIC_VECTOR(width-1 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity zerodetect is -- true if all input bits are zero
    generic(width: integer);
    port(a: in  STD_LOGIC_VECTOR(width-1 downto 0);
         y: out STD_LOGIC);
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity flop is -- flip-flop
    generic(width: integer);
    port(clk: in  STD_LOGIC;
         d:   in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:   out STD_LOGIC_VECTOR(width-1 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity flopen is -- flip-flop with enable
    generic(width: integer);
    port(clk, en: in  STD_LOGIC;
         d:       in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:       out STD_LOGIC_VECTOR(width-1 downto 0));
end;
```

```
library IEEE; use IEEE.STD_LOGIC_1164.all; use IEEE.STD_LOGIC_ARITH.all;
entity flopenr is -- flip-flop with enable and synchronous reset
    generic(width: integer);
    port(clk, reset, en: in  STD_LOGIC;
         d:              in  STD_LOGIC_VECTOR(width-1 downto 0);
         q:              out STD_LOGIC_VECTOR(width-1 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux2 is -- two-input multiplexer
    generic(width: integer);
    port(d0, d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
         s:      in  STD_LOGIC;
         y:      out STD_LOGIC_VECTOR(width-1 downto 0));
end;

library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux4 is  -- four-input multiplexer
    generic(width: integer);
    port(d0, d1, d2, d3: in  STD_LOGIC_VECTOR(width-1 downto 0);
         s:              in  STD_LOGIC_VECTOR(1 downto 0);
         y:              out STD_LOGIC_VECTOR(width-1 downto 0));
end;

------------------------------------------------------
-- Architecture Definitions
------------------------------------------------------
architecture test of top is
    component mips generic(width:   integer := 8;   -- default 8-bit datapath
                           regbits: integer := 3);  -- and 3 bit register addresses (8 regs)
        port(clk, reset:        in  STD_LOGIC;
             memdata:           in  STD_LOGIC_VECTOR(width-1 downto 0);
             memread, memwrite: out STD_LOGIC;
             adr, writedata:    out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component memory generic(width: integer);
        port(clk, memwrite:  in  STD_LOGIC;
             adr, writedata: in  STD_LOGIC_VECTOR(width-1 downto 0);
             memdata:        out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    signal clk, reset, memread, memwrite: STD_LOGIC;
    signal memdata, adr, writedata: STD_LOGIC_VECTOR(width-1 downto 0);
begin
    -- mips being tested
    dut: mips generic map(width, regbits)
              port map(clk, reset, memdata, memread, memwrite, adr, writedata);
    -- external memory for code and data
    exmem: memory generic map(width)
                  port map(clk, memwrite, adr, writedata, memdata);

    -- Generate clock with 10 ns period
    process begin
        clk <= '1';
        wait for 5 ns;
        clk <= '0';
        wait for 5 ns;
    end process;
```



```
    -- Generate reset for first two clock cycles
    process begin
        reset <= '1';
        wait for 22 ns;
        reset <= '0';
        wait;
    end process;

    -- check that 7 gets written to address 76 at end of program
    process (clk) begin
        if (clk'event and clk = '0' and memwrite = '1') then
            if (conv_integer(adr) = 76 and conv_integer(writedata) = 7) then
                report "Simulation completed successfully";
            else report "Simulation failed.";
            end if;
        end if;
    end process;
end;

architecture synth of memory is
begin
    process is
        file mem_file: text open read_mode is "memfile.dat";
        variable L: line;
        variable ch: character;
        variable index, result: integer;
        type ramtype is array (255 downto 0) of STD_LOGIC_VECTOR(7 downto 0);
        variable mem: ramtype;
    begin
        -- initialize memory from file
        -- memory in little-endian format
        --  80020044 means mem[3] = 80 and mem[0] = 44
        for i in 0 to 255 loop -- set all contents low
            mem(conv_integer(i)) := "00000000";
        end loop;
        index := 0;
        while not endfile(mem_file) loop
            readline(mem_file, L);
            for j in 0 to 3 loop
                result := 0;
                    for i in 1 to 2 loop
                        read(L, ch);
                        if '0' <= ch and ch <= '9' then
                            result := result*16 + character'pos(ch)-character'pos('0');
                        elsif 'a' <= ch and ch <= 'f' then
                            result := result*16 + character'pos(ch)-character'pos('a')+10;
                        else report "Format error on line " & integer'image(index)
                             severity error;
                        end if;
                    end loop;
                mem(index*4+3-j) := conv_std_logic_vector(result, width);
            end loop;
            index := index + 1;
        end loop;
        -- read or write memory
        loop
            if clk'event and clk = '1' then
                if (memwrite = '1') then mem(conv_integer(adr)) := writedata;
                end if;
```

```
            end if;
            memdata <= mem(conv_integer(adr));
            wait on clk, adr;
        end loop;
    end process;
end;

architecture struct of mips is
    component controller
        port(clk, reset:        in  STD_LOGIC;
             op:                in  STD_LOGIC_VECTOR(5 downto 0);
             zero:              in  STD_LOGIC;
             memread, memwrite, alusrca, memtoreg,
             iord, pcen, regwrite, regdst: out STD_LOGIC;
             pcsrc, alusrcb, aluop:  out STD_LOGIC_VECTOR(1 downto 0);
             irwrite:           out STD_LOGIC_VECTOR(3 downto 0));
    end component;
    component alucontrol
        port(aluop:             in  STD_LOGIC_VECTOR(1 downto 0);
             funct:             in  STD_LOGIC_VECTOR(5 downto 0);
             alucont:           out STD_LOGIC_VECTOR(2 downto 0));
    end component;
    component datapath generic(width, regbits: integer);
        port(clk, reset:        in  STD_LOGIC;
             memdata:           in  STD_LOGIC_VECTOR(width-1 downto 0);
             alusrca, memtoreg, iord, pcen,
             regwrite, regdst:  in  STD_LOGIC;
             pcsrc, alusrcb:    in  STD_LOGIC_VECTOR(1 downto 0);
             irwrite:           in  STD_LOGIC_VECTOR(3 downto 0);
             alucont:           in  STD_LOGIC_VECTOR(2 downto 0);
             zero:              out STD_LOGIC;
             instr:             out STD_LOGIC_VECTOR(31 downto 0);
             adr, writedata:    out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    signal instr: STD_LOGIC_VECTOR(31 downto 0);
    signal zero, alusrca, memtoreg, iord, pcen, regwrite, regdst: STD_LOGIC;
    signal aluop, pcsrc, alusrcb: STD_LOGIC_VECTOR(1 downto 0);
    signal irwrite: STD_LOGIC_VECTOR(3 downto 0);
    signal alucont: STD_LOGIC_VECTOR(2 downto 0);
begin
    cont: controller port map(clk, reset, instr(31 downto 26), zero,
                              memread, memwrite, alusrca, memtoreg,
                              iord, pcen, regwrite, regdst,
                              pcsrc, alusrcb, aluop, irwrite);
    ac: alucontrol port map(aluop, instr(5 downto 0), alucont);
    dp: datapath generic map(width, regbits)
                 port map(clk, reset, memdata, alusrca, memtoreg,
                          iord, pcen, regwrite, regdst,
                          pcsrc, alusrcb, irwrite,
                          alucont, zero, instr, adr, writedata);
end;

architecture synth of controller is
    type statetype is (FETCH1, FETCH2, FETCH3, FETCH4, DECODE, MEMADR,
                       LBRD, LBWR, SBWR, RTYPEEX, RTYPEWR, BEQEX, JEX);
    constant LB:    STD_LOGIC_VECTOR(5 downto 0) := "100000";
    constant SB:    STD_LOGIC_VECTOR(5 downto 0) := "101000";
    constant RTYPE: STD_LOGIC_VECTOR(5 downto 0) := "000000";
```



```
    constant BEQ:   STD_LOGIC_VECTOR(5 downto 0) := "000100";
    constant J:     STD_LOGIC_VECTOR(5 downto 0) := "000010";
    signal state, nextstate:    statetype;
    signal pcwrite, pcwritecond: STD_LOGIC;
begin
    process (clk) begin -- state register
        if clk'event and clk = '1' then
            if reset = '1' then state <= FETCH1;
            else state <= nextstate;
            end if;
        end if;
    end process;

    process (state, op) begin -- next state logic
        case state is
            when FETCH1 => nextstate <= FETCH2;
            when FETCH2 => nextstate <= FETCH3;
            when FETCH3 => nextstate <= FETCH4;
            when FETCH4 => nextstate <= DECODE;
            when DECODE => case op is
                               when LB | SB => nextstate <= MEMADR;
                               when RTYPE => nextstate <= RTYPEEX;
                               when BEQ => nextstate <= BEQEX;
                               when J => nextstate <= JEX;
                               when others => nextstate <= FETCH1; -- should never happen
                           end case;
            when MEMADR => case op is
                               when LB => nextstate <= LBRD;
                               when SB => nextstate <= SBWR;
                               when others => nextstate <= FETCH1; -- should never happen
                           end case;
            when LBRD => nextstate <= LBWR;
            when LBWR => nextstate <= FETCH1;
            when SBWR => nextstate <= FETCH1;
            when RTYPEEX => nextstate <= RTYPEWR;
            when RTYPEWR => nextstate <= FETCH1;
            when BEQEX => nextstate <= FETCH1;
            when JEX => nextstate <= FETCH1;
            when others => nextstate <= FETCH1; -- should never happen
        end case;
    end process;

    process (state) begin
        -- set all outputs to zero, then conditionally assert just the appropriate ones
        irwrite <= "0000";
        pcwrite <= '0'; pcwritecond <= '0';
        regwrite <= '0';  regdst <= '0';
        memread <= '0'; memwrite <= '0';
        alusrca <= '0'; alusrcb <= "00"; aluop <= "00";
        pcsrc <= "00";
        iord <= '0'; memtoreg <= '0';

        case state is
            when FETCH1 => memread <= '1';
                           irwrite <= "0001";
                           alusrcb <= "01";
                           pcwrite <= '1';
            when FETCH2 => memread <= '1';
                           irwrite <= "0010";
```

```
                        alusrcb <= "01";
                        pcwrite <= '1';
            when FETCH3 => memread <= '1';
                        irwrite <= "0100";
                        alusrcb <= "01";
                        pcwrite <= '1';
            when FETCH4 => memread <= '1';
                        irwrite <= "1000";
                        alusrcb <= "01";
                        pcwrite <= '1';
            when DECODE => alusrcb <= "11";
            when MEMADR => alusrca <= '1';
                        alusrcb <= "10";
            when LBRD =>   memread <= '1';
                        iord <= '1';
            when LBWR =>   regwrite <= '1';
                        memtoreg <= '1';
            when SBWR =>   memwrite <= '1';
                        iord <= '1';
            when RTYPEEX => alusrca <= '1';
                         aluop <= "10";
            when RTYPEWR => regdst <= '1';
                         regwrite <= '1';
            when BEQEX =>  alusrca <= '1';
                        aluop <= "01";
                        pcwritecond <= '1';
                        pcsrc <= "01";
            when JEX =>    pcwrite <= '1';
                        pcsrc <= "10";
        end case;
    end process;

    pcen <= pcwrite or (pcwritecond and zero); -- program counter enable
end;

architecture synth of alucontrol is
begin
    process(aluop, funct) begin
        case aluop is
            when "00" => alucont <= "010"; -- add (for lb/sb/addi)
            when "01" => alucont <= "110"; -- sub (for beq)
            when others => case funct is         -- R-type instructions
                               when "100000" => alucont <= "010"; -- add (for add)
                               when "100010" => alucont <= "110"; -- subtract (for sub)
                               when "100100" => alucont <= "000"; -- logical and (for and)
                               when "100101" => alucont <= "001"; -- logical or (for or)
                               when "101010" => alucont <= "111"; -- set on less (for slt)
                               when others   => alucont <= "---"; -- should never happen
                           end case;
        end case;
    end process;
end;

architecture struct of datapath is
    component alu generic(width: integer);
        port(a, b:    in  STD_LOGIC_VECTOR(width-1 downto 0);
             alucont: in  STD_LOGIC_VECTOR(2 downto 0);
             result:  out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
```



```
    component regfile generic(width, regbits: integer);
        port(clk:          in  STD_LOGIC;
             write:        in  STD_LOGIC;
             ra1, ra2, wa: in  STD_LOGIC_VECTOR(regbits-1 downto 0);
             wd:           in  STD_LOGIC_VECTOR(width-1 downto 0);
             rd1, rd2:     out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component zerodetect generic(width: integer);
        port(a: in  STD_LOGIC_VECTOR(width-1 downto 0);
             y: out STD_LOGIC);
    end component;
    component flop generic(width: integer);
        port(clk: in  STD_LOGIC;
             d:   in  STD_LOGIC_VECTOR(width-1 downto 0);
             q:   out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component flopen generic(width: integer);
        port(clk, en: in  STD_LOGIC;
             d:       in  STD_LOGIC_VECTOR(width-1 downto 0);
             q:       out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component flopenr generic(width: integer);
        port(clk, reset, en: in  STD_LOGIC;
             d:              in  STD_LOGIC_VECTOR(width-1 downto 0);
             q:              out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component mux2 generic(width: integer);
        port(d0, d1: in  STD_LOGIC_VECTOR(width-1 downto 0);
             s:      in  STD_LOGIC;
             y:      out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    component mux4 generic(width: integer);
        port(d0, d1, d2, d3: in  STD_LOGIC_VECTOR(width-1 downto 0);
             s:              in  STD_LOGIC_VECTOR(1 downto 0);
             y:              out STD_LOGIC_VECTOR(width-1 downto 0));
    end component;
    constant CONST_ONE:  STD_LOGIC_VECTOR(width-1 downto 0) := conv_std_logic_vector(1, width);
    constant CONST_ZERO: STD_LOGIC_VECTOR(width-1 downto 0) := conv_std_logic_vector(0, width);
    signal ra1, ra2, wa: STD_LOGIC_VECTOR(regbits-1 downto 0);
    signal pc, nextpc, md, rd1, rd2, wd, a,
           src1, src2, aluresult, aluout, dp_writedata, constx4: STD_LOGIC_VECTOR(width-1 downto 0);
    signal dp_instr: STD_LOGIC_VECTOR(31 downto 0);

begin
    -- shift left constant field by 2
    constx4 <= dp_instr(width-3 downto 0) & "00";

    -- register file address fields
    ra1 <= dp_instr(regbits+20 downto 21);
    ra2 <= dp_instr(regbits+15 downto 16);
    regmux:  mux2 generic map(regbits) port map(dp_instr(regbits+15 downto 16),
                                                dp_instr(regbits+10 downto 11), regdst, wa);

    -- independent of bit width, load dp_instruction into four 8-bit registers over four cycles
    ir0: flopen generic map(8) port map(clk, irwrite(0), memdata(7 downto 0), dp_instr(7 downto 0));
    ir1: flopen generic map(8) port map(clk, irwrite(1), memdata(7 downto 0), dp_instr(15 downto 8));
    ir2: flopen generic map(8) port map(clk, irwrite(2), memdata(7 downto 0), dp_instr(23 downto 16));
    ir3: flopen generic map(8) port map(clk, irwrite(3), memdata(7 downto 0), dp_instr(31 downto 24));
```

```
    -- datapath
    pcreg:    flopenr generic map(width) port map(clk, reset, pcen, nextpc, pc);
    mdr:      flop generic map(width) port map(clk, memdata, md);
    areg:     flop generic map(width) port map(clk, rd1, a);
    wrd:      flop generic map(width) port map(clk, rd2, dp_writedata);
    res:      flop generic map(width) port map(clk, aluresult, aluout);
    adrmux:   mux2 generic map(width) port map(pc, aluout, iord, adr);
    src1mux:  mux2 generic map(width) port map(pc, a, alusrca, src1);
    src2mux:  mux4 generic map(width) port map(dp_writedata, CONST_ONE,
                                        dp_instr(width-1 downto 0), constx4, alusrcb, src2);
    pcmux:    mux4 generic map(width) port map(aluresult, aluout, constx4, CONST_ZERO, pcsrc, nextpc);
    wdmux:    mux2 generic map(width) port map(aluout, md, memtoreg, wd);
    rf:       regfile generic map(width, regbits) port map(clk, regwrite, ra1, ra2, wa, wd, rd1, rd2);
    alunit:   alu generic map(width) port map(src1, src2, alucont, aluresult);
    zd:       zerodetect generic map(width) port map(aluresult, zero);

    -- drive outputs
    instr <= dp_instr; writedata <= dp_writedata;
end;

architecture synth of alu is
    signal b2, sum, slt: STD_LOGIC_VECTOR(width-1 downto 0);
begin
    b2 <= not b when alucont(2) = '1' else b;
    sum <= a + b2 + alucont(2);
    -- slt should be 1 if most significant bit of sum is 1
    slt <= conv_std_logic_vector(1, width) when sum(width-1) = '1'
           else conv_std_logic_vector(0, width);
    with alucont(1 downto 0) select result <=
        a and b when "00",
        a or b  when "01",
        sum     when "10",
        slt     when others;
end;

architecture synth of regfile is
    type ramtype is array (2**regbits-1 downto 0) of STD_LOGIC_VECTOR(width-1 downto 0);
    signal mem: ramtype;
begin
    -- three-ported register file
    -- read two ports combinationally
    -- write third port on rising edge of clock
    process(clk) begin
        if clk'event and clk = '1' then
            if write = '1' then mem(conv_integer(wa)) <= wd;
            end if;
        end if;
    end process;
    process(ra1, ra2) begin
        if (conv_integer(ra1) = 0) then rd1 <= conv_std_logic_vector(0, width); -- register 0 holds 0
        else rd1 <= mem(conv_integer(ra1));
        end if;
        if (conv_integer(ra2) = 0) then rd2 <= conv_std_logic_vector(0, width);
        else rd2 <= mem(conv_integer(ra2));
        end if;
    end process;
end;
```



```
architecture synth of zerodetect is
    signal i: integer;
    signal x: STD_LOGIC_VECTOR(width-1 downto 1);
begin -- N-bit AND of inverted inputs
    AllBits: for i in width-1 downto 1 generate
        LowBit: if i = 1 generate
                    A1: x(1) <= not a(0) and not a(1);
                end generate;
                OtherBits: if i /= 1 generate
                    Ai: x(i) <= not a(i) and x(i-1);
                end generate;
        end generate;
    y <= x(width-1);
end;

architecture synth of flop is
begin
    process(clk) begin
        if clk'event and clk = '1' then -- or use "if RISING_EDGE(clk) then"
            q <= d;
        end if;
    end process;
end;

architecture synth of flopen is
begin
    process(clk) begin
        if clk'event and clk = '1' then
            if en = '1' then q <= d;
            end if;
        end if;
    end process;
end;

architecture synchronous of flopenr is
begin
    process(clk) begin
        if clk'event and clk = '1' then
            if reset = '1' then
                q <= CONV_STD_LOGIC_VECTOR(0, width); -- produce a vector of all zeros
            elsif en = '1' then q <= d;
            end if;
        end if;
    end process;
end;

architecture synth of mux2 is
begin
    y <= d0 when s = '0' else d1;
end;

architecture synth of mux4 is
begin
    y <= d0 when s = "00" else
         d1 when s = "01" else
         d2 when s = "10" else
         d3;
end;
```

Ασκήσεις

Μπορείτε να εκτελέσετε τις ακόλουθες ασκήσεις χρησιμοποιώντας την HDL που προτιμάτε. Εάν έχετε διαθέσιμο έναν προσομοιωτή, ελέγξτε τη σχεδίασή σας. Εκτυπώστε τις κυματομορφές και εξηγήστε πώς αποδεικνύουν ότι ο κώδικας δουλεύει σωστά. Εάν έχετε διαθέσιμο ένα εργαλείο σύνθεσης, συνθέστε τον κώδικά σας. Εκτυπώστε το παραγόμενο κυκλωματικό διάγραμμα και εξηγήστε γιατί ταιριάζει με τις προσδοκίες σας.

A.1 Σχεδιάστε ένα σχηματικό του κυκλώματος που περιγράφεται από τον ακόλουθο HDL κώδικα. Απλοποιήστε το στον ελάχιστο αριθμό πυλών.

SystemVerilog

```
module exercise1(input  logic a, b, c,
                 output logic y, z);

  assign y = a & b & c | a & b & ~c | a & ~b & c;
  assign z = a & b | ~a & ~b;
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity exercise1 is
  port(a, b, c: in  STD_LOGIC;
       y, z:    out STD_LOGIC);
end;

architecture synth of exercise1 is
begin
  y <= (a and b and c) or (a and b and (not c)) or
       (a and (not b) and c);
  z <= (a and b) or ((not a) and (not b));
end;
```

A.2 Σχεδιάστε ένα σχηματικό του κυκλώματος που περιγράφεται από τον ακόλουθο HDL κώδικα. Απλοποιήστε το στον ελάχιστο αριθμό πυλών.

SystemVerilog

```
module exercise2(input  logic [3:0] a,
                 output logic [1:0] y);

  always_comb
    if (a[0])      y = 2'b11;
    else if (a[1]) y = 2'b10;
    else if (a[2]) y = 2'b01;
    else if (a[3]) y = 2'b00;
    else           y = a[1:0];
endmodule
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity exercise2 is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(1 downto 0));
end;

architecture synth of exercise2 is
begin
  process(a) begin
    if    a(0) = '1' then y <= "11";
    elsif a(1) = '1' then y <= "10";
    elsif a(2) = '1' then y <= "01";
    elsif a(3) = '1' then y <= "00";
    else                  y <= a(1 downto 0);
    end if;
  end process;
end;
```

A.3 Γράψτε μια λειτουργική μονάδα σε HDL, η οποία θα υπολογίζει μια συνάρτηση XOR 4 εισόδων. Η είσοδος είναι $A_{3:0}$ και η έξοδος είναι Y.

A.4 Γράψτε ένα πρόγραμμα testbench με δυνατότητα αυτο-ελέγχου για την Άσκηση A.3. Δημιουργήστε ένα αρχείο διανυσμάτων ελέγχου, το οποίο θα περιλαμβάνει και τις 16 περιπτώσεις ελέγχου. Προσομοιώστε το κύκλωμα και αποδείξτε ότι δουλεύει. Εισάγετε ένα σφάλμα στο αρχείο διανυσμάτων ελέγχου και δείξτε ότι αναφέρει μια αναντιστοιχία.

A.5 Γράψτε μια λειτουργική μονάδα HDL με όνομα minority. Η μονάδα δέχεται τρεις εισόδους, A, B και C και παράγει μία έξοδο Y η οποία είναι TRUE εάν τουλάχιστον δύο από τις εισόδους είναι FALSE.

A.6 Γράψτε μία λειτουργική μονάδα HDL για την οθόνη ενός αποκωδικοποιητή 7 (σε δεκαεξαδικό). Ο αποκωδικοποιητής θα πρέπει να χειρίζεται τα ψηφία A, B, C, D, E και F, καθώς και τα 0-9.

A.7 Γράψτε ένα πρόγραμμα testbench με δυνατότητα αυτο-ελέγχου για την Άσκηση A.6. Δημιουργήστε ένα αρχείο διανυσμάτων ελέγχου, το οποίο θα περιλαμβάνει και τις 16 περιπτώσεις ελέγχου. Προσομοιώστε το κύκλωμα και δείξτε ότι δουλεύει. Εισάγετε ένα σφάλμα στο αρχείο διανυσμάτων ελέγχου και δείξτε ότι αναφέρει μια αναντιστοιχία.

A.8 Γράψτε μια λειτουργική μονάδα με όνομα mux8 για έναν πολυπλέκτη 8:1, με εισόδους $S_{2:0}$, D0, D1, D2, D3, D4, D5, D6, D7 και έξοδο Y.

A.9 Γράψτε μια δομική μονάδα για τον υπολογισμό της συνάρτησης $Y = A\overline{B} + \overline{BC} + \overline{A}BC$ χρησιμοποιώντας λογική πολυπλέκτη. Χρησιμοποιήστε τον πολυπλέκτη 8:1 από την Άσκηση A.8.

A.10 Επαναλάβετε την Άσκηση A.9 χρησιμοποιώντας έναν πολυπλέκτη 4:1 και όσες πύλες NOT χρειάζεστε.

A.11 Στην Ενότητα A.5.4 επισημάναμε ότι ένας συγχρονιστής θα μπορούσε να περιγραφεί σωστά με εντολές κλειδωμένης ανάθεσης, εάν αυτές δίνονταν με τη σωστή σειρά. Σκεφτείτε κάποιο άλλο, απλό ακολουθιακό κύκλωμα το οποίο δεν μπορεί να περιγραφεί σωστά με κλειδωμένες αναθέσεις, ανεξαρτήτως σειράς.

A.12 Γράψτε μια λειτουργική μονάδα HDL για ένα κύκλωμα προτεραιότητας 8 εισόδων.

A.13 Γράψτε μια λειτουργική μονάδα HDL για έναν αποκωδικοποιητή 2:4.

A.14 Γράψτε μια λειτουργική μονάδα HDL για έναν αποκωδικοποιητή 6:64, χρησιμοποιώντας τρεις από τους αποκωδικοποιητές 2:4 της Άσκησης A.13, μαζί με 64 πύλες AND 3 εισόδων.

A.15 Σχεδιάστε το διάγραμμα μετάβασης καταστάσεων για την FSM που περιγράφει ο ακόλουθος HDL κώδικας.

SystemVerilog

```
module fsm2(input  logic clk, reset,
            input  logic a, b,
            output logic y);

  typedef enum logic [1:0]
    {S0, S1, S2, S3} statetype;

  statetype state, nextstate;

  always_ff @(posedge clk)
    if (reset) state <= S0;
    else       state <= nextstate;

  always_comb
    case (state)
      S0: if (a ^ b) nextstate = S1;
          else       nextstate = S0;
      S1: if (a & b) nextstate = S2;
          else       nextstate = S0;
      S2: if (a | b) nextstate = S3;
          else       nextstate = S0;
      S3: if (a | b) nextstate = S3;
          else       nextstate = S0;
    endcase
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fsm2 is
  port(clk, reset: in  STD_LOGIC;
       a, b:       in  STD_LOGIC;
       y:          out STD_LOGIC);
end;

architecture synth of fsm2 is
  type statetype is (S0, S1, S2, S3);
  signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset = '1' then state <= S0;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

process (state, a, b) begin
    case state is
      when S0 => if (a xor b) = '1' then
                   nextstate <= S1;
                 else nextstate <= S0;
                 end if;
```

(συνεχίζεται)

SystemVerilog (συνέχεια)

```
assign y = (state == S1) || (state == S2);
endmodule
```

VHDL (συνέχεια)

```
      when S1 => if (a and b) = '1' then
                   nextstate <= S2;
                 else nextstate <= S0;
                 end if;
      when S2 => if (a or b) = '1' then
                   nextstate <= S3;
                 else nextstate <= S0;
                 end if;
      when S3 => if (a or b) = '1' then
                   nextstate <= S3;
                 else nextstate <= S0;
                 end if;
    end case;
  end process;

  y <= '1' when ((state = S1) or (state = S2))
       else '0';
end;
```

A.16 Σχεδιάστε το διάγραμμα μετάβασης καταστάσεων για την FSM που περιγράφεκ ο ακόλουθος HDL κώδικας. Μια FSM αυτού του είδους χρησιμοποιείται σε μια μονάδα πρόβλεψης διακλάδωσης ορισμένων μικροεπεξεργαστών.

SystemVerilog

```
module fsm1(input  logic clk, reset,
            input  logic taken, back,
            output logic predicttaken);

  typedef enum logic [4:0]
    {S0 = 5'b00001,
     S1 = 5'b00010,
     S2 = 5'b00100,
     S3 = 5'b001000,
     S4 = 5'b10000} statetype;

  statetype state, nextstate;

  always_ff @(posedge clk)
    if (reset) state <= S2;
    else       state <= nextstate;

  always_comb
    case (state)
      S0: if (taken) nextstate = S1;
          else       nextstate = S0;
      S1: if (taken) nextstate = S2;
          else       nextstate = S0;
      S2: if (taken) nextstate = S3;
          else       nextstate = S1;
      S3: if (taken) nextstate = S4;
          else       nextstate = S2;
      S4: if (taken) nextstate = S4;
          else       nextstate = S3;
      default:       nextstate = S2;
    endcase
```

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fsm1 is
  port(clk, reset:   in  STD_LOGIC;
       taken, back:  in  STD_LOGIC;
       predicttaken: out STD_LOGIC);
end;

architecture synth of fsm1 is
  type statetype is (S0, S1, S2, S3, S4);
  signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset = '1' then state <= S2;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

  process (state, taken) begin
    case state is
      when S0 => if taken = '1' then
                   nextstate <= S1;
                 else nextstate <= S0;
                 end if;
      when S1 => if taken = '1' then
                   nextstate <= S2;
                 else nextstate <= S0;
                 end if;
```

(συνεχίζεται)

SystemVerilog (συνέχεια)

```
  assign predicttaken = (state == S4) ||
                        (state == S3) ||
                        (state == S2 && back);
endmodule
```

VHDL (συνέχεια)

```
      when S2 => if taken = '1' then
                   nextstate <= S3;
                 else nextstate <= S1;
                 end if;
      when S3 => if taken = '1' then
                   nextstate <= S4;
                 else nextstate <= S2;
                 end if;
      when S4 => if taken = '1' then
                   nextstate <= S4;
                 else nextstate <= S3;
                 end if;
      when others =>  nextstate <= S2;
    end case;
  end process;

  -- output logic
  predicttaken <= '1' when
                  ((state = S4) or (state = S3) or
                   (state = S2 and back = '1'))
    else '0';
end;
```

A.17 Γράψτε μια λειτουργική μονάδα HDL για έναν SR μανδαλωτή.

A.18 Γράψτε μια λειτουργική μονάδα HDL για ένα *JK flip-flop*. Το flip-flop έχει εισόδους *clk*, *J* και *K* και έξοδο *Q*. Στην ανοδική ακμή του ρολογιού, η έξοδος *Q* διατηρεί την παλαιά τιμή της εάν $J = K = 0$. Θέτει την *Q* σε 1 εάν $J = 1$, επαναφέρει την *Q* σε 0 εάν $K = 1$ και αντιστρέφει την *Q* εάν $J = K = 1$.

A.19 Γράψτε μια γραμμή HDL κώδικα, η οποία θα συνδέει μέσω πύλης έναν 32-bit δίαυλο με όνομα `data` μ' ένα άλλο σήμα `sel`, για την παραγωγή ενός 32-bit αποτελέσματος, `result`. Εάν το `sel` είναι TRUE, `result = data`. Διαφορετικά, το `result` θα πρέπει να περιέχει μόνο 0.

Ασκήσεις πάνω στην SystemVerilog

Οι ακόλουθες ασκήσεις αφορούν ειδικά την SystemVerilog.

A.20 Εξηγήστε τη διαφορά μεταξύ κλειδωμένης καιμη-κλειδωμένης ανάθεσης στην SystemVerilog. Δώστε παραδείγματα.

A.21 Τι κάνει η ακόλουθη εντολή της SystemVerilog;

```
result = |(data[15:0] & 16'hC820);
```

A.22 Ξαναγράψτε τη λειτουργική μονάδα syncbad από την Ενότητα A.5.4. Χρησιμοποιήστε μη-κλειδωμένες αναθέσεις, αλλά τροποποιήστε τον κώδικα ώστε να παράγει ένα σωστό συγχρονιστή με δύο flip-flop.

A.23 Δίνονται τα ακόλουθα δύο αποσπάσματα κώδικα SystemVerilog. Έχουν την ίδια λειτουργία; Σχεδιάστε το hardware που αναπαριστά το καθένα.

```
module code1(input  logic clk, a, b, c,
             output logic y);
  logic x;

  always_ff @(posedge clk) begin
    x <= a & b;
    y <= x | c;
  end
endmodule
```

```
module code2(input  logic a, b, c, clk,
             output logic y);
  logic x;

  always_ff @(posedge clk) begin
    y <= x | c;
    x <= a & b;
  end
endmodule
```

A.24 Επαναλάβετε την Άσκηση A.23 για την περίπτωση όπου ο τελεστής <= αντικαθίσταται από τον τελεστή = παντού στον κώδικα.

A.25 Οι ακόλουθες λειτουργικές μονάδες SystemVerilog περιέχουν σφάλματα, τα οποία οι συγγραφείς έχουν δει να γίνονται από σπουδαστές. Εξηγήστε το σφάλμα σε κάθε λειτουργική μονάδα και τον τρόπο διόρθωσής του.

```
module latch(input  logic       clk,
             input  logic [3:0] d,
             output logic [3:0] q);

   always @(clk)
     if (clk) q <= d;
endmodule

module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5);

   always @(a)
     begin
       y1 = a & b;
       y2 = a | b;
       y3 = a ^ b;
       y4 = ~(a & b);
       y5 = ~(a | b);
     end
endmodule

module mux2(input  logic [3:0] d0, d1,
            input  logic       s,
            output logic [3:0] y);

   always @(posedge s)
     if (s) y <= d1;
     else   y <= d0;

endmodule

module twoflops(input  logic clk,
                input  logic d0, d1,
                output logic q0, q1);

   always @(posedge clk)
     q1 = d1;
     q0 = d0;
endmodule

module FSM(input  logic clk,
           input  logic a,
           output logic out1, out2);

   logic state;
```



```
   // next state logic and register (sequential)
   always_ff @(posedge clk)
     if (state == 0) begin
       if (a) state <= 1;
     end else begin
       if (~a) state <= 0;
     end

   always_comb // output logic (combinational)
     if (state == 0) out1 = 1;
     else            out2 = 1;
endmodule

module priority(input  logic [3:0] a,
                output logic [3:0] y);

   always_comb
     if      (a[3]) y = 4'b1000;
     else if (a[2]) y = 4'b0100;
     else if (a[1]) y = 4'b0010;
     else if (a[0]) y = 4'b0001;
endmodule

module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic out);

   typedef enum logic [1:0] {S0, S1, S2} statetype;

   statetype state, nextstate;

   // State Register
   always_ff @(posedge clk)
     if (reset) state <= S0;
     else       state <= nextstate;

   // Next State Logic
   always_comb
     case (state)
       S0: nextstate = S1;
       S1: nextstate = S2;
       S2: nextstate = S0;
     endcase

   // Output Logic
   assign out = (state == S2);
endmodule

module mux2tri(input  logic [3:0] d0, d1,
               input  logic       s,
               output tri   [3:0] y);

   tristate t0(d0, s, y);
   tristate t1(d1, s, y);
endmodule

module floprsen(input  logic       clk,
                input  logic       reset,
                input  logic       set,
```

```
                input  logic [3:0] d,
                output logic [3:0] q);

   always_ff @(posedge clk)
      if (reset) q <= 0;
      else       q <= d;

   always @(set)
     if (set) q <= 1;
endmodule

module and3(input  logic a, b, c,
            output logic y);

   logic tmp;

   always @(a, b, c)
   begin
      tmp <= a & b;
      y   <= tmp & c;
   end
endmodule
```

Ασκήσεις πάνω στην VHDL

Οι ακόλουθες ασκήσεις αφορούν ειδικά την VHDL.

A.26 Στην VHDL, γιατί είναι αναγκαίο να γράφουμε το

```
q <= '1' when state = S0 else '0';
```

αντί απλώς το

```
q <= (state = S0); ?
```

A.27 Κάθε μία από τις ακόλουθες λειτουργικές μονάδες VHDL περιέχει ένα σφάλμα. Χάριν συντομίας, παρουσιάζεται μόνο η ενότητα architecture· υποθέστε ότι οι δηλώσεις για τη χρήση βιβλιοθηκών και η δήλωση οντότητας (entity) είναι σωστές. Εξηγήστε το σφάλμα και τον τρόπο διόρθωσής του.

```
architecture synth of latch is
begin
  process(clk) begin
    if clk = '1' then q <= d;
    end if;
  end process;
end;

architecture proc of gates is
begin
  process(a) begin
    y1 <= a and b;
    y2 <= a or b;
    y3 <= a xor b;
    y4 <= a nand b;
    y5 <= a nor b;
  end process;
end;

architecture synth of flop is
```



```
begin
  process(clk)
    if clk'event and clk = '1' then
      q <= d;
end;

architecture synth of priority is
begin
  process(a) begin
    if    a(3) = '1' then y <= "1000";
    elsif a(2) = '1' then y <= "0100";
    elsif a(1) = '1' then y <= "0010";
    elsif a(0) = '1' then y <= "0001";
    end if;
  end process;
end;

architecture synth of divideby3FSM is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
  process(clk, reset) begin
    if reset = '1' then state <= S0;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;

   process(state) begin
    case state is
      when S0 => nextstate <= S1;
      when S1 => nextstate <= S2;
      when S2 => nextstate <= S0;
    end case;
  end process;

  q <= '1' when state = S0 else '0';
end;

architecture struct of mux2 is
  component tristate
    port(a:  in  STD_LOGIC_VECTOR(3 downto 0);
         en: in  STD_LOGIC;
         y:  out STD_LOGIC_VECTOR(3 downto 0));
  end component;
begin
  t0: tristate port map(d0, s, y);
  t1: tristate port map(d1, s, y);
end;

architecture asynchronous of flopr is
begin
  process(clk, reset) begin
    if reset = '1' then
      q <= '0';
    elsif clk'event and clk = '1' then
      q <= d;
    end if;
  end process;
```

```
  process(set) begin
    if set = '1' then
      q <= '1';
    end if;
  end process;
end;

architecture synth of mux3 is
begin
  y <= d2 when s(1) else
       d1 when s(0) else d0;
end;
```

Βιβλιογραφικές Παραπομπές

Μεγάλο μέρος των εργασιών που αναφέρονται εδώ προέρχονται από δημοσιεύσεις του IEEE, οι οποίες είναι διαθέσιμες στους ενδιαφερόμενους αναγνώστες από τον ιστότοπο ieeexplore.ieee.org. Επιπλέον, θα βρείτε την παρούσα βιβλιογραφία σε ηλεκτρονική μορφή, εμπλουτισμένη με συνδέσεις προς τις σχετικές ιστοσελίδες, στον ιστότοπο www.cmosvlsi.com. Στο κείμενο υπάρχουν συχνές παραπομπές σε υλικό από την περιοδική έκδοση του IEEE για τα κυκλώματα στερεάς κατάστασης (*IEEE Journal of Solid-State Circuits*), η οποία αναφέρεται ως *JSSC*.

[Abdollahi04] A. Abdollahi, F. Fallah, and M. Pedram, «Leakage current reduction in CMOS VLSI circuits by input vector control», *IEEE Trans. VLSI* , τομ. 12, αρ. 2, Feb. 2004, σελ. 140–154.

[Acken83] J. Acken, «Testing for bridging faults (shorts) in CMOS circuits», *Proc. Design Automation Conf.*, 1983, σελ. 717–718.

[Afghahi90] M. Afghahi and C. Svensson, «A unified single-phase clocking scheme for VLSI systems», *JSSC*, τομ. 25, αρ. 1, Feb. 1990, σελ. 225–233.

[Agans06] D. Agans, *Debugging*, New York: Amacon, 2006, www.debuggingrules.com.

[Agarwal01] V. Agarwal, S. Keckler, and D. Burger, «The effect of technology scaling on microarchitectural structures», *Computer Architecture and Technology Laboratory Technical Report TR2000–02*, University of Texas at Austin, 2001.

[Agarwal04] A. Agarwal, V. Zolotov, and D. Blaauw, «Statistical clock skew analysis considering intra-die process variations», *IEEE Trans. CAD*, τομ. 23, αρ. 8, Aug. 2004, σελ. 1231–1242.

[Agarwal07] A. Agarwal, K. Kang, S. Bhunia, J. Gallagher, and K. Roy, «Device-aware yield-centric dual-V_t design under parameter variations in nanoscale technologies», *IEEE Trans. VLSI*, τομ. 15, αρ. 6, Jun. 2007, σελ. 660–671.

[Agarwal07b] K. Agarwal, R. Rao, D. Sylvester, and R. Brown, «Parametric yield analysis and optimization in leakage dominated technologies», *IEEE Trans. VLSI*, τομ. 15, αρ. 6, Jun. 2007, σελ. 613–623.

[Agrawal08] B. Agrawal and T. Sherwood, «Ternary CAM power and delay model: extensions and uses», *IEEE Trans. VLSI*, τομ. 16, αρ. 5, May 2008, σελ. 554–564.

[Aisaka02] K. Aisaka κ. α., «Design rule for frequency-voltage cooperative power control and its application to an MPEG-4 decoder», *Proc. VLSI Circuits Symp.*, 2002, σελ. 216–217.

[Alexander75] J. Alexander, «Clock recovery from random binary signals», *Electronics Letters*, τομ. 11, αρ. 22, Oct. 30, 1975, σελ. 541–542.

[Allam00] M. Allam, M. Anis, and M. Elmasry, «High-speed dynamic logic styles for scaled-down CMOS and MTCMOS technologies», *Proc. Intl. Symp. Low Power Electronics and Design*, 2000, σελ. 155–160.

[Alon05] E. Alon, V. Stojanovic, and M. Horowitz, «Circuits and techniques for high-resolution measurement of on-chip power supply noise», *JSSC*, τομ. 40, αρ. 4, Apr. 2005, σελ. 820–828.

[Alvandpour02] A. Alvandpour, R. Krishnamurthy, K. Soumyanath, and S. Borkar, «A sub-130-nm conditional keeper technique», *JSSC*, τομ. 37, αρ. 5, May 2002, σελ. 633–638.

[Amrutur98] B. Amrutur and M. Horowitz, «A replica technique for wordline and sense control in low-power SRAM's», *JSSC*, τομ. 33, αρ. 8, Aug. 1998, σελ. 1208–1219.