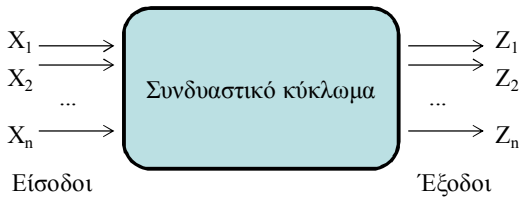
# 8. ΠΡΑΚΤΙΚΑ ΣΥΝΔΥΑΣΤΙΚΑ ΚΥΚΛΩΜΑΤΑ

## 1. Εισαγωγή

Ένα συνδυαστικό κύκλωμα αποτελείται γενικά από διάφορα σήματα εισόδου, εξόδου, και πύλες διασύνδεσης (**Εικόνα 1**). Κάθε έξοδος είναι συνάρτηση των εισόδων. Τα σήματα εισόδου και εξόδου αντιστοιχούν σε δυαδικές μεταβλητές.

Τα συνδυαστικά κυκλώματα είναι τα βασικά στοιχεία κάθε ψηφιακού συστήματος. Ένα ακολουθιακό κύκλωμα είναι ένα συνδυασμός ενός κυκλώματος λογικής και στοιχείων μνήμης, τα οποία στις περισσότερες περιπτώσεις είναι μια διασύνδεση πυλών.




**Εικόνα 1.** Γενική δομή συνδυαστικού κυκλώματος.

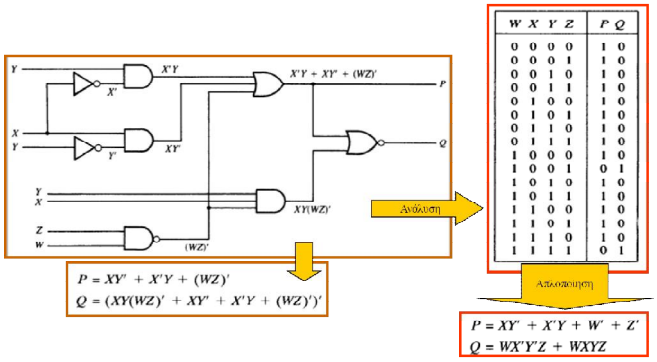
## 2. Συναρτησιακή ανάλυση

Η **συναρτησιακή ανάλυση** ασχολείται με τη συναρτησιακή λειτουργία ή αλλιώς με τη σχέση των εξόδων με τις εισόδους. Το αποτέλεσμα της συναρτησιακής ανάλυσης είναι είτε ένας πίνακας αληθείας ή η συναρτησιακή αναπαράσταση κάθε εξόδου σε σχέση με τις εισόδους.

Κάθε πύλη στο κύκλωμα παράγει μια καθυστέρηση μετάδοσης στη ροή του σήματος μέσα από το κύκλωμα. Η **απόκριση χρόνου** ενός κυκλώματος είναι μια συνάρτηση των χαρακτηριστικών καθυστέρησης των πυλών του κυκλώματος. Η **χρονική ανάλυση** προσδιορίζει το χρόνο απόκρισης (άρα και την ταχύτητα) και άλλα χρονικά χαρακτηριστικά που μπορεί να οδηγήσουν σε εσφαλμένη συμπεριφορά του κυκλώματος.

Η **ανάλυση φόρτου** προσδιορίζει αν υπάρχουν τα υπερβολικά φορτία στο κύκλωμα. Κάθε πύλη που συνδέεται με άλλες λέμε ότι φορτώνει την πύλη που ακολουθεί. Υπάρχουν περιορισμοί που περιορίζουν το φόρτο σε κάθε πύλη και άρα τον αριθμό των πυλών που μπορεί να συνδέονται στην έξοδό της.

Οι συναρτήσεις Boole πρώτα ελαχιστοποιούνται και μετά φτιάχνουμε το κύκλωμα. Αλλά αν δίνεται ένα κύκλωμα, πρώτα αναλύουμε την έξοδό του και στη συνέχεια εξετάζουμε αν μπορεί να απλοποιηθεί.




**Εικόνα 2.** Παράδειγμα συναρτησιακής ανάλυσης. Εξαγωγή κανονικής και απλοποιημένης μορφής λογικής συνάρτησης ψηφιακού κυκλώματος.

## 3. Σχεδίαση συνδυαστικού κυκλώματος

Η σχεδίαση είναι η διαδικασία μετατροπή μιας συναρτησιακής διατύπωσης της εξόδου ενός κυκλώματος (όπως π.χ. προκύπτει από τον πίνακα αληθείας) σε ένα λογικό κύκλωμα πυλών.

Η λογική σχεδίαση συνδυαστικών κυκλωμάτων αποτελείται από τα επόμενα βήματα:

1.Από τη φραστική διατύπωση της λειτουργία του κυκλώματος λογικής, πρέπει να εξάγουμε τα σήματα εισόδου και εξόδου.

2.Αναλύουμε τη φραστική περιγραφή και παράγουμε τις σχέσεις κάθε εξόδου συναρτήσει των εισόδων, χρησιμοποιώντας τον πίνακα αληθείας.

3.Παράγουμε μια «ελάχιστη» συνάρτηση λογικής για κάθε έξοδο.

4.Παράγουμε το κύκλωμα των ελάχιστων συναρτήσεων εξόδου.

Κάθε κύκλωμα λογικής μπορεί να παραχθεί σε επίπεδο πυλών με διάφορους τρόπους συνδυασμών πυλών. Οι δημοφιλέστεροι είναι οι επόμενοι:

1.AND-OR

2.NAND-NAND

3.OR-AND

4.NOR-NOR

## 4. Υλοποίηση NAND-NAND και NOR – NOR 2 επιπέδων

Η υλοποίηση NAND-NAND μπορεί να παραχθεί από την AND-OR αντικαθιστώντας κάθε πύλη στο κύκλωμα AND-OR με μια πύλη NAND με τον ίδιο αριθμό εισόδων όπως η πύλη που αντικαθιστά.

Η υλοποίηση NOR-NOR μπορεί να παραχθεί από την OR-AND αντικαθιστώντας κάθε πύλη στο κύκλωμα OR-AND με μια πύλη NOR με τον ίδιο αριθμό εισόδων όπως η πύλη που αντικαθιστά.

Η διαδικασία μετατροπής των υλοποιήσεων AND-OR και OR-AND σε NAND – NAND και NOR – NOR αντίστοιχα, μπορεί να εφαρμοστεί σε όλες τις 2-επιπεδες υλοποιήσεις εφόσον δεν υπάρχει απευθείας είσοδος στο 2ο επίπεδο πυλών !!

Διαφορετικά, η είσοδος που εισάγεται απευθείας στο 2ο επίπεδο πυλών, θα πρέπει να αντιστραφεί πριν εισαχθεί στο 2ο επίπεδο της NAND – NAND ή NOR – NOR υλοποίησης.

Αν το κύκλωμα που θέλουμε να μετασχηματίσουμε αποτελείται από περισσότερα από 2 επίπεδα πυλών, η προηγούμενη διαδικασία δεν ισχύει !!!! Κάθε πύλη στο κύκλωμα πρέπει να αντικατασταθεί από το αντίστοιχο ισοδύναμο NAND ή NOR κύκλωμα. Στη συνέχεια το κύκλωμα θα πρέπει να αναλυθεί παραπέρα για να απομακρύνουμε τις πλεονάζουσες πύλες.

Επειδή κάθε κουτί IC περιέχει αρκετές πύλες του ίδιου τύπου, οι υλοποιήσεις με NAND ή NOR είναι προτιμότερες έναντι των υλοποιήσεων AND-OR, ή OR-AND, στην περίπτωση χρήσης SSI (small scale Ics) και MSI (medium scale Ics). Η χρήση ενός είδους πύλης δεν απαιτεί την αλλαγή του IC και οδηγεί σε καλύτερη αξιοποίηση των πυλών που περιέχει κάθε IC.

Η καθολικότητα (οικουμενικότητα) των πυλών NAND, NOR δίνει τη δυνατότητα να παίρνουμε συμπεριφορά AND και OR, αντίστοιχα, συμπληρώνοντας τις εξόδους των πρώτων. Η κατασκευή NAND και NOR ICs είναι λοιπόν χρησιμότερη, ωστόσο σε επίπεδο LSI (large scale integration) και πάνω, τα ολοκληρωμένα αυτά κυκλώματα είναι πολύπλοκα οπότε μπορούμε να καταφύγουμε στην άμεση υλοποίηση πιο πολύπλοκων πυλών κατευθείαν στο πυρίτιο.

Οι ελαχιστοποιημένες μορφές της λογικής συνάρτησης (άθροισμα γινομένων ή γινόμενο αθροισμάτων) γενικά περιέχουν διαφορετικό αριθμό εισόδων, οπότε η υλοποίηση της κάθε μορφής θα είναι γενικά διαφορετική.

## 5. Συγκριτής 1 bit

| A | B | F1: A=B | F2:A>B | F3:A<B |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 |

**Πίνακας 1.** Πίνακας αληθείας συγκριτή 1bit.

Ο πίνακας αληθείας για έναν συγκριτή 1bit φαίνεται στον **Πίνακα 1**. Το κύκλωμα έχει δύο εισόδους τις A και B και τρεις εξόδους τις F1, F2, F3. Η F1 είναι 1 μόνο όταν A = B. Η F2 = 1 όταν A>B και η F3 = 1 όταν A<B.

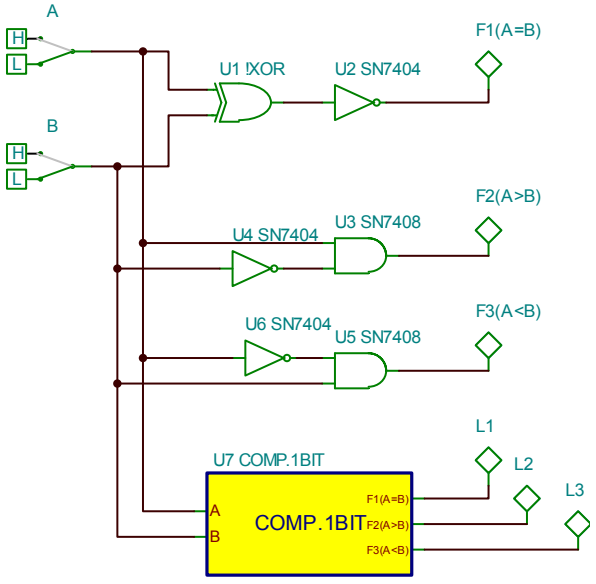
Η λογική συνάρτηση για καθεμία από τις τρεις εξόδους είναι αντίστοιχα:

F1 = A'B' + AB = XNOR(A,B)

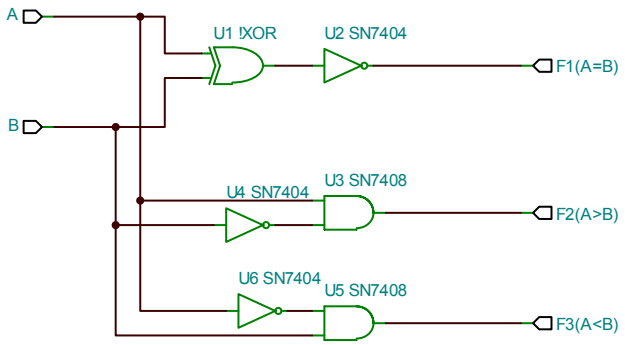
F2 = AB'

F3 = A'B

Στην **Εικόνα 3** φαίνεται η AND-OR υλοποίηση του κυκλώματος του συγκριτή 1bit στο TINA καθώς και η block υλοποίηση του ίδιο κυκλώματος. Στην **Εικόνα 4** φαίνεται το περιεχόμενο του block με τους ακροδέκτες σύνδεσης Pin1 για τη σύνδεση του εσωτερικού του block με τα εξωτερικά κυκλωματικά στοιχεία.




**Εικόνα 3.** AND-OR υλοποίηση του κυκλώματος του συγκριτή 1bit στο TINA καθώς και η block υλοποίηση του ίδιο κυκλώματος.




**Εικόνα 4.** Το περιεχόμενο του block με τους ακροδέκτες σύνδεσης.

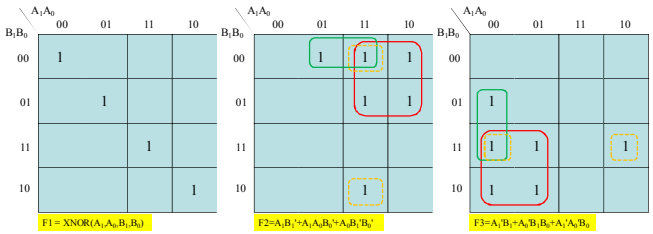
## 6. Συγκριτής 2 bits

| $A_1$ | $A_0$ | $B_1$ | $B_0$ | F1 (A=B) | F2 (A>B) | F3 (A<B) |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 |

**Πίνακας 2.** Πίνακας αληθείας συγκριτή δύο ακεραίων 2bit.

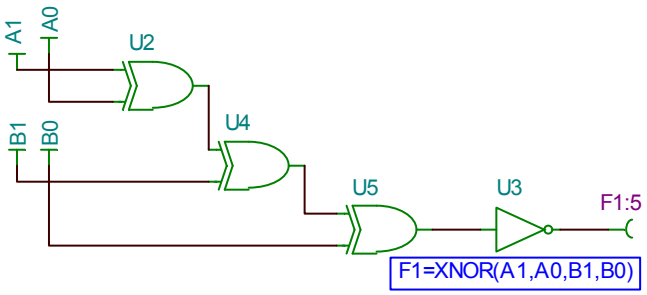
Ο συγκριτής 2bits συγκρίνει δύο ακεραίους 2bit. Οπότε θα έχει 4 εισόδους (από 2bit για καθένα ακέραιο) και τρεις εξόδους. Ο πίνακας αληθείας του είναι ο **Πίνακας 2.**

Οι λογικές συναρτήσεις εξόδου για την ισότητα και τις ανισότητες, του προηγούμενου συγκριτή φαίνονται στην **Εικόνα 5**. Υλοποιούμε τις συναρτήσεις αυτές στο TINA και χρησιμοποιούμε τον jumper (**Εικόνα 6**) για να δώσουμε κοινή ονομασία όπου χρειάζεται στις εισόδους των πυλών όπως φαίνεται στις **Εικόνες 7-9.**

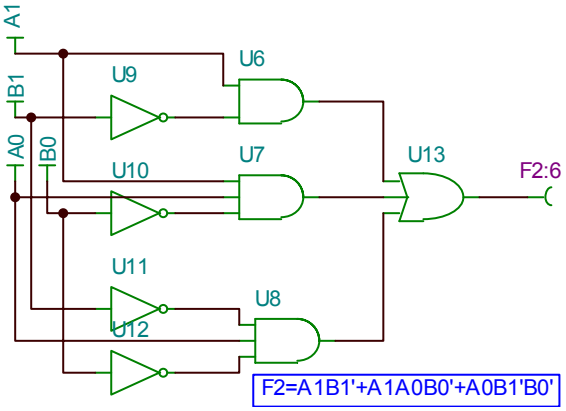



**Εικόνα 5.** ΧΚ για την απλοποίηση της λογικής συνάρτησης των εξόδων του συγκριτή.

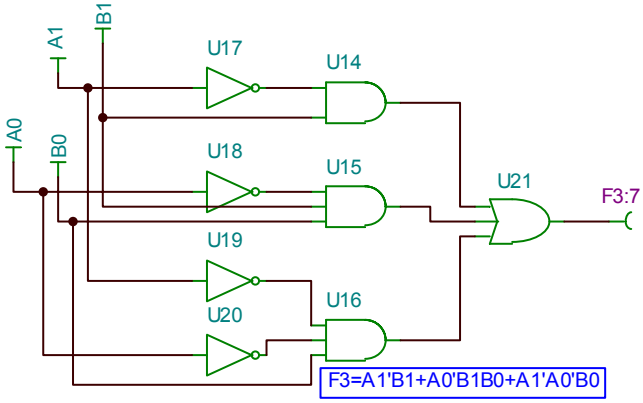

**Εικόνα 6.** Η θέση του Jumperστην παλέτα Special του TINA.




**Εικόνα 7.** Υλοποίηση της συνάρτησης F1.

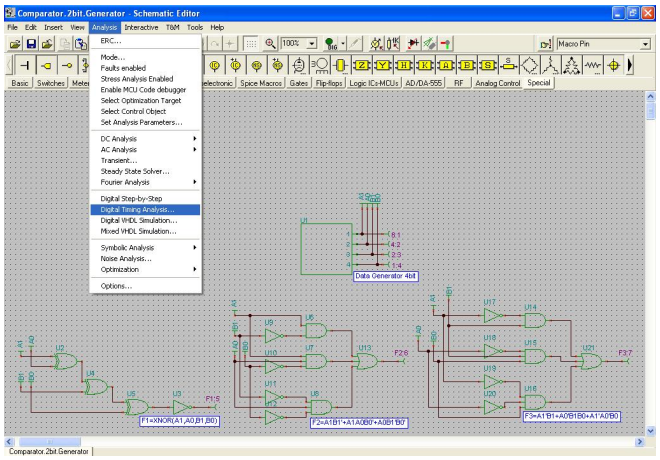



**Εικόνα 8.** Υλοποίηση της συνάρτησης F2.

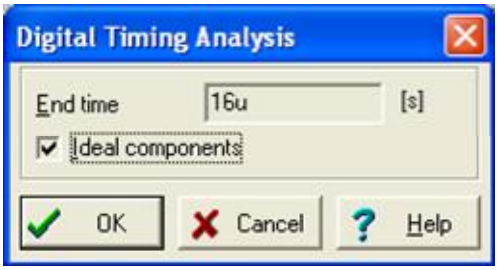



**Εικόνα 9.** Υλοποίηση της συνάρτησης F3.

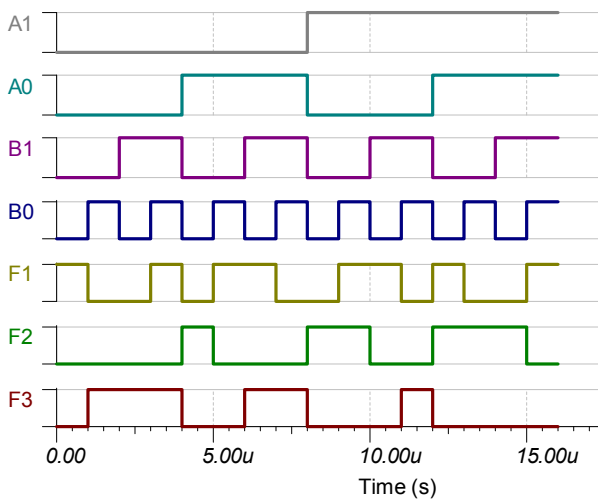
Στην **Εικόνα 10** φαίνεται η επιφάνεια εργασίας στο TINA με όλες τις συναρτήσεις F1, F2, F3 και τη γεννήτρια παραγωγής 4bit αριθμών για την αυτόματη απόδοση των τιμών A1A0 και B1B0 όπου χρειάζεται μέσω των jumpers.



**Εικόνα 10.** Εκκίνηση της ψηφιακής χρονικής ανάλυσης.



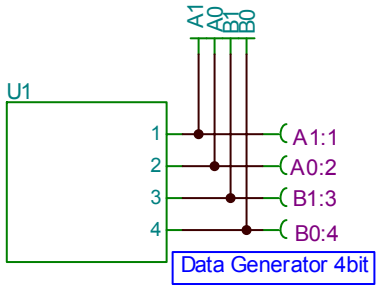

**Εικόνα 11.** Πλαίσιο διαλόγου ψηφιακής χρονικής ανάλυσης.




**Εικόνα 12.** Αποτελέσματα ψηφιακής χρονικής ανάλυσης για την περίπτωση του συγκριτή δύο 2bit ακεραίων.

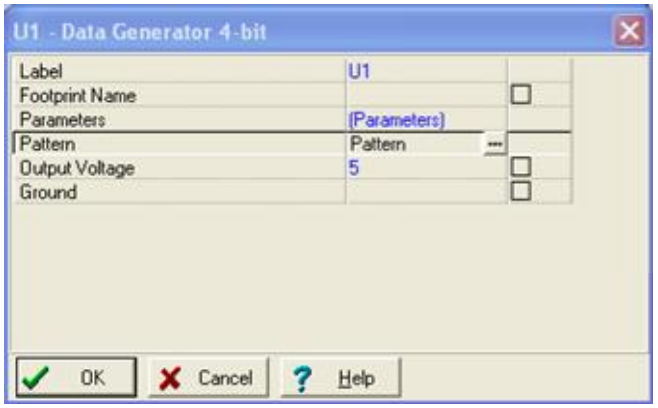
Με **Analysis ➔ Digital Timing Analysis** ξεκινά η χρονική ανάλυση και στο πλαίσιο διαλόγου που εμφανίζεται (**Εικόνα 11**) πρέπει να εισάγουμε το χρόνο που θα τελειώσει (Εφόσον κάθε bit θα διατηρείται για 1μs, συνολικά οι 16 συνδυασμοί του πίνακα αληθείας θα χρειαστούν 16μs ή 16u στο TINA) (**Εικόνα 12**).

Στην **Εικόνα 13** παρουσιάζεται η γεννήτρια παραγωγής 4bit αριθμών από την παλέτα εργαλείων του TINA (**Εικόνα 14**). Για να αρχικοποιήσουμε τη γεννήτρια παραγωγής 4bit αριθμών, πρέπει αρχικά να ανοίξουμε – με διπλό κλικ πάνω της – το πλαίσιο διαλόγου – παραμέτρων (**Εικόνα 15**) και στη συνέχεια να πιέσουμε πάνω στο Pattern (**Εικόνα 16**) και να εισάγουμε τον τρόπο λειτουργίας της (Affected address (low)=00, Affected address (high)=0F, Step time = 1u, Start address = 00, Stop address = 0F, Fill).

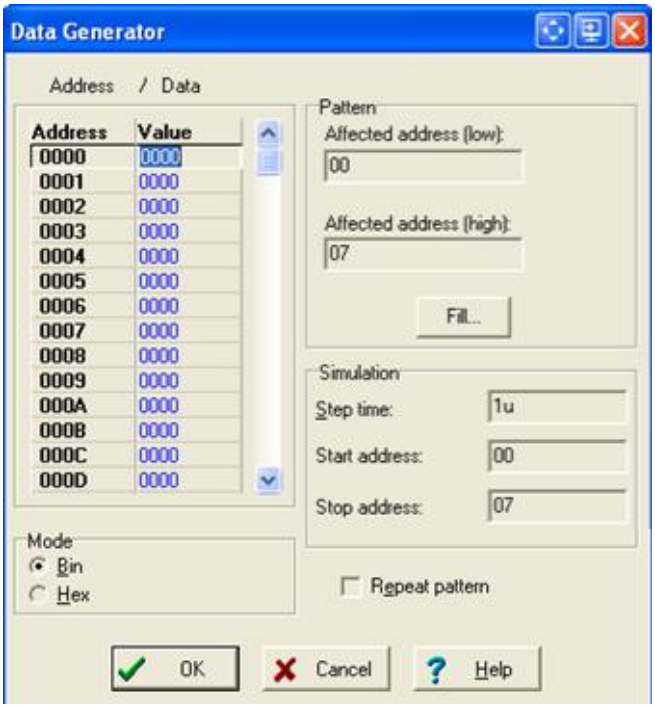



**Εικόνα 13.** Σύμβολο στο TINA για τη γεννήτρια 4bit ακεραίων.


**Εικόνα 14.** Η θέση της γεννήτριας 4bit στην παλέτα Sources του TINA.




**Εικόνα 15.** Πλαίσιο διαλόγου για την γεννήτρια 4bit.




**Εικόνα 16.** Πλαίσιο εισαγωγής δεδομένων για τη γεννήτρια 4bit.

## 7. Κωδικοποιητής (Coder)

Ένας κωδικοποιητής, είναι ένα ψηφιακό κύκλωμα που εκτελεί την αντίστροφη λειτουργία απ' ότι ο αποκωδικοποιητής. Ένας κωδικοποιητής έχει $2^n$ (ή λιγότερες) γραμμές εισόδου και n γραμμές εξόδου. Οι γραμμές εξόδου παράγουν το δυαδικό κώδικα που αντιστοιχεί στις μεταβλητές εισόδου. Υποτίθεται ότι μόνο μια είσοδος έχει την τιμή 1 σε μια δοσμένη χρονική στιγμή, αλλιώς το κύκλωμα δεν έχει νόημα.

Στον **Πίνακα 3** φαίνεται ο πινακας αληθείας ενός κωδικοποιητή από 8 σε 3 γραμμές.

Ο κωδικοποιητής μπορεί να υλοποιηθεί με πύλες OR, των οποίων οι είσοδοι καθορίζονται κατευθείαν από τον πίνακα αληθείας ως εξής για τον προηγούμενο κωδικοποιητή:

$x = D_4 + D_5 + D_6 + D_7$

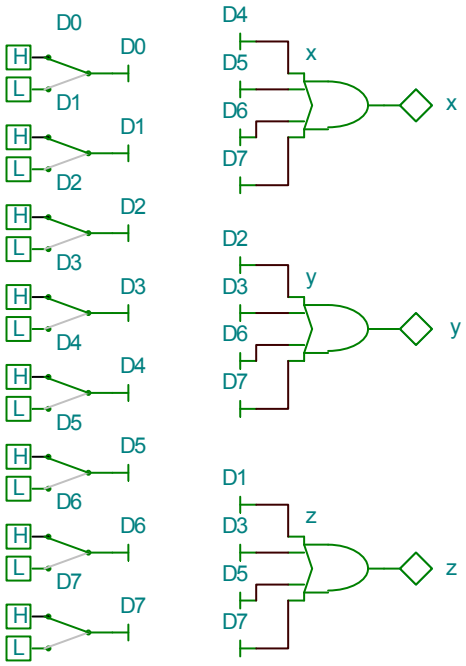
$y = D_2 + D_3 + D_6 + D_7$

$z = D_1 + D_3 + D_5 + D_7$

Η υλοποίησή του με πύλες φαίνεται στην **Εικόνα 17.**

| $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ | $D_7$ | x | y | z |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

**Πίνακας 3.** Κωδικοποιητής από 8 σε 3 γραμμές. Πίνακας αληθείας.




**Εικόνα 17.** Υλοποίηση με πύλες κωδικοποιητή 8 σε 3.

Ο προηγούμενος κωδικοποιητής, έχει τον περιορισμό ότι μόνο μια είσοδος μπορεί να είναι ενεργός σε κάθε χρονική στιγμή. Αν δύο είσοδοι είναι ενεργές ταυτόχρονα, η έξοδος παράγει έναν απροσδιόριστο συνδυασμό. Για την άρση αυτής της αβεβαιότητας, τα κυκλώματα του κωδικοποιητή θα πρέπει να καθορίζουν μια προτεραιότητα που να εξασφαλίζει, ότι μόνο μια είσοδος κωδικοποιείται.

Μια άλλη αβεβαιότητα στον κωδικοποιητή από 8δικό-σε-2δικό είναι, ότι μια έξοδος όλο μηδενικά δημιουργείται, όταν όλες οι είσοδοι είναι 0. Το πρόβλημα είναι ότι και όταν η $D_0$=1, τότε και πάλι η έξοδος έχει όλο μηδενικά. Για να

άρουμε και αυτή την αβεβαιότητα, θα πρέπει να δημιουργήσουμε μια επιπλέον έξοδο, η οποία καθορίζει την κατάσταση κατά την οποία καμιά από τις εισόδους δεν είναι έγκυρη.

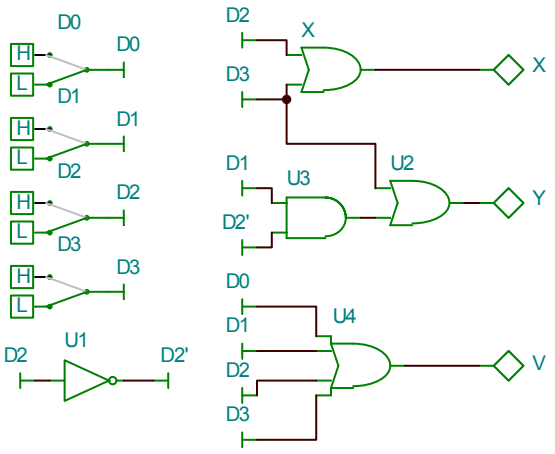
## 8. Κωδικοποιητής προτεραιότητας (Priority Coder)

Ένα κωδικοποιητής προτεραιότητας είναι ένα κύκλωμα κωδικοποιητή που περιλαμβάνει τη συνάρτηση προτεραιτότητας. Η λειτουργία του είναι τέτοια που, αν δύο ή περισσότερες εισόδοι είναι ίσες με 1 ταυτόχρονα, η έξοδος με τη μεγαλύτερη προτεραιότητα καθορίζει την έξοδο. Ο **Πίνακας 4** παρουσιάζει τον πίνακα αληθείας κωδικοποιητή προτεραιότητας 4 εισόδων.

| $D_0$ | $D_1$ | $D_2$ | $D_3$ | X | Y | V |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | X | X | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| X | 1 | 0 | 0 | 0 | 1 | 1 |
| X | X | 1 | 0 | 1 | 0 | 1 |
| X | X | X | 1 | 1 | 1 | 1 |

**Πίνακας 4.** Κωδικοποιητής προτεραιότητας. Πίνακας αληθείας.

Η είσοδος $D_3$ έχει τη μεγαλύτερη προτεραιότητα. Έτσι άσχεται από τις τιμές των άλλων εισόδων, όταν αυτή η είσοδος είναι 1, η έξοδος για τα xy είναι 11 (δυαδικό 3). Η $D_2$ είναι η αμέσως επόμενη σε προτεραιότητα. Η έξοδος είναι 10 αν $D_2$=1, και με την προϋπόθεση ότι $D_3$=0, άσχεται από τις τιμές των άλλων δύο μικρότερης προτεραιότητας εισόδων. Η έξοδος για τη $D_1$ παράγεται μόνο αν οι μεγαλύτερες προτεραιότητας είσοδοι είναι 0 κτλ. Ο δείκτης έγκυρης εξόδου V, παίρνει την τιμή 1, μόνο όταν μία ή περισσότερες από τις εισόδους είναι ίσες με 1. Αν όλες οι είσοδοι είναι 0, ο V είναι 0 και οι άλλες δύο έξοδοι του κυκλώματος δε χρησιμοποιούνται. Στην **Εικόνα 17** φαίνονται οι ΧΚ και οι απλοποιήσεις για τις X και Y.

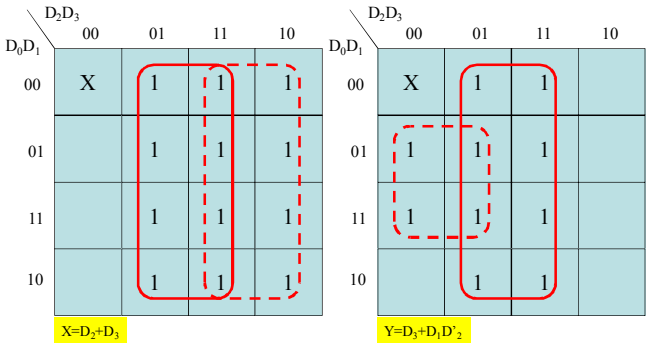



**Εικόνα 18.** Υλοποίηση κωδικοποιητή προτεραιότητας.

Συγκεντρωτικά έχουμε τις επόμενες σχέσεις:

$X = D_2 + D_3$
$Y = D_3 + D_1 D_2'$
$V = D_0 + D_1 + D_2 + D_3$

Οι υλοποίηση με πύλες του κωδικοποιητή προτεραιότητας φαίνεται στην **Εικόνα 18**.




**Εικόνα 17.** Απλοποιήσεις ΧΚ για τον αποκωδικοποιητή προτεραιότητας.

## 9. Αποκωδικοποιητής (Decoder)

Οι διακριτές πληροφορίες στα ψηφιακά συστήματα συμβολίζονται με δυαδικούς κώδικες. Ένας δυαδικός κώδικας των n-bits είναι ικανός να παραστήσει ως $2^n$ διαφορετικά στοιχεία της κωδικοποιημένης πληροφορίας. Ένας αποκωδικοποιητής (decoder) είναι ένα συνδυαστικό κύκλωμα που μετατρέπει τη δυαδική πληροφορία n γραμμών εισόδου σε έως $2^n$ μοναδικές γραμμές εξόδου. Αν τα n-bits πληροφορίας έχουν αχρησιμοποίητους ή αδιάφορους όρους, ο αποκωδικοποιητής θα έχει λιγότερες από $2^n$ εξόδους. Οι αποκωδικοποιητές που παρουσιάζονται εδώ ονομάζονται

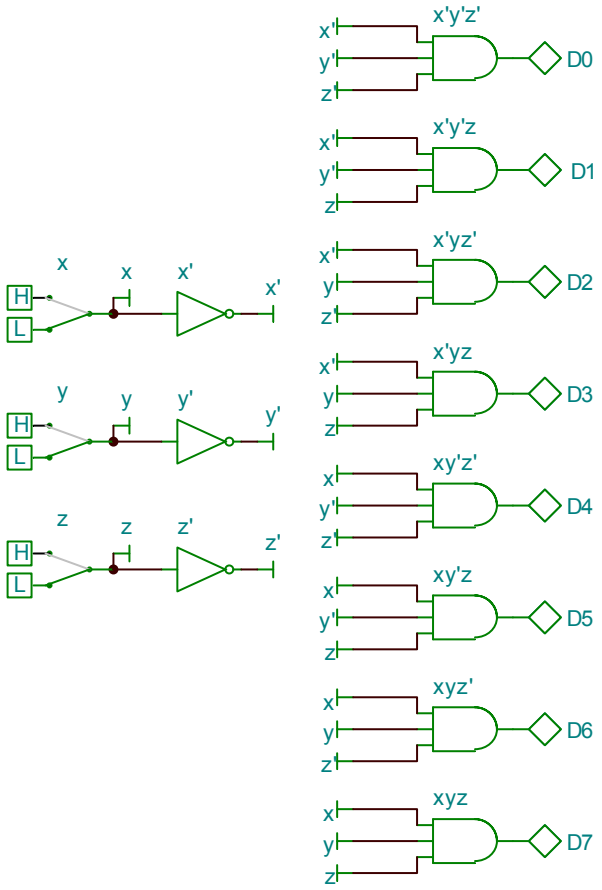
αποκωδικοποιητές από n-σε-m γραμμές όπου $m \leq 2^n$. Σκοπός τους είναι να παράγουν τους $2^n$ ελαχιστόρους των n μεταβλητών εισόδου.

Στον **Πίνακα 5** φαίνεται ο πίνακας αληθείας ενός αποκωδικοποιητή από 3 σε 8 γραμμές και στην **Εικόνα 19** η υλοποίησή του με πύλες στο TINA.

| x | y | z | $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ | $D_7$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

**Πίνακας 5.** Πίνακας αληθείας ενός αποκωδικοποιητή από 3 σε 8 γραμμές.

Ένας αποκωδικοποιητής μας δίνει τους $2^n$ ελαχιστόρους των n μεταβλητών εισόδου. Αφού κάθε συνάρτηση Boole μπορεί να εφραστεί σε κανονική μορφή αθροίσματος ελαχιστόρων, μπορεί κανείς να χρησιμοποιήσει έναν αποκωδικοποιητή για να παράγει τους ελαχιστόρους και μια εξωτερική πύλη OR για να σχηματίσει το άθροισμα. Με τον τρόπο αυτό κάθε συνδυαστικό κύκλωμα με n εισόδους και m εξόδους μπορεί να υλοποιηθεί με έναν αποκωδικοποιητή n-σε-$2^n$ γραμμών και με m πύλες OR.




**Εικόνα 19.** Υλοποίηση του αποκωδικοποιητή από 3 σε 8 γραμμές με πύλες.

Για την υλοποίηση ενός συνδυαστικού κυκλώματος με αποκωδικοποιητή και πύλες OR, χρειαζόμαστε τις συναρτήσεις Boole του κυκλώματος σε μορφή αθροίσματος ελαχιστόρων. Αυτή η μορφή μπορεί να προκύψει εύκολα από τον πίνακα αληθείας ή αναπτύσσοντας τις συναρτήσεις στο άθροισμα των ελαχιστόρων τους. Μετά διαλέγουμε έναν αποκωδικοποιητή που να παράγει όλους τους ελαχιστόρους των n μεταβλητών εισόδου. Σαν εισόδους κάθε πύλης OR, διαλέγουμε μερικές εξόδους του αποκωδικοποιητή, σύμφωνα με το ποιους ελαχιστόρους έχει η κάθε συνάρτηση.

## 10. Αποκωδικοποιητής με είσοδο επίτρεψης (Enable)

Με μία επιπλέον είσοδο επίτρεψης (enable) (την E στα επόμενα) μπορούμε να ελέγχουμε τη λειτουργία του αποκωδικοποιητή ως εξής π.χ. για αποκωδικοποιητή από 2 σε 4 γραμμές:

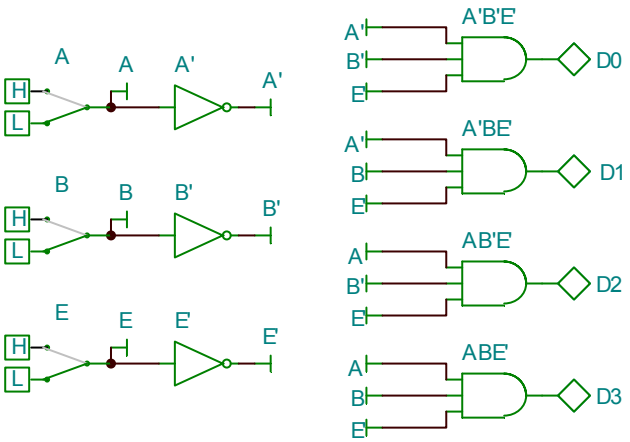
- Με E=1, ανεξάρτητα από τις τιμές των εισόδων A,B, όλες οι έξοδοι είναι 1.

- Με E=0, ανάλογα με τις τιμές των εισόδων A,B κάθε φορά μόνο μια έξοδος είναι 0, αυτή που αντιστοιχεί στο δυαδικό συνδυασμό που υποδεικνύουν οι A, B.

Ο πίνακας αληθείας αυτού του αποκωδικοποιητή φαίνεται στον **Πίνακα 6.** Η υλοποίηση του κυκλώματος αυτού με πύλες φαίνεται στην **Εικόνα 20.**

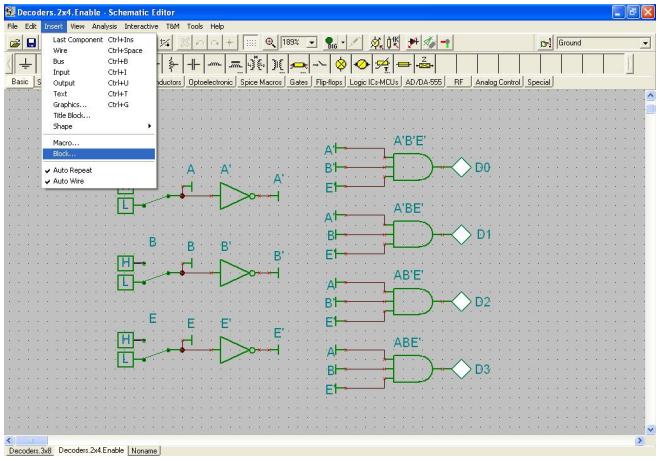
| E | A | B | $D_0$ | $D_1$ | $D_2$ | $D_3$ |
|---|---|---|---|---|---|---|
| 1 | X | X | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |

**Πίνακας 6.** Πίνακας αληθείας του αποκωδικοποιητή 2 σε 4 με επίτρεψη (enable).

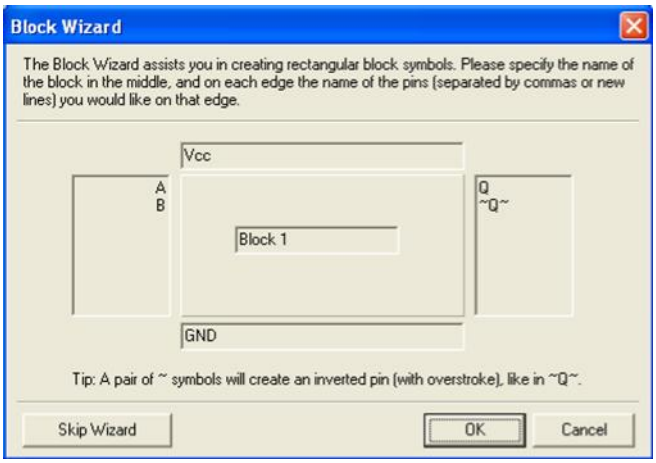



**Εικόνα 20.** Υλοποίηση του αποκωδικοποιητή 2 σε 4 με επίτρεψη (enable) με πύλες.

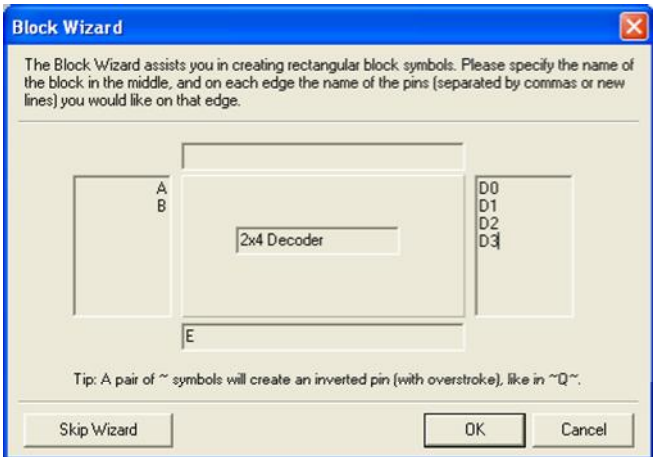
Στο σημείο αυτό θα δείξουμε αναλυτικά τον τρόπο που μπορούμε να δημιουργήσουμε ιεραρχική σχεδίαση στο TINA ώστε να μην περιπλέκεται το σχέδιο από τις πολλές πύλες.
Από το μενού εργαλείων του TINA επιλέγουμε **Insert ➔ Block** όπως φαίνεται στην **Εικόνα 21**. Εκκινείται ο **Block Wizard** όπως φαίνεται στην **Εικόνα 22**.




**Εικόνα 21.** Για να ξεκινήσουμε ιεραρχική σχεδίαση ξεκινάμε με την εισαγωγή block.




**Εικόνα 22.** Το αρχικό πλαίσιο διαλόγου για τη δημιουργία block.



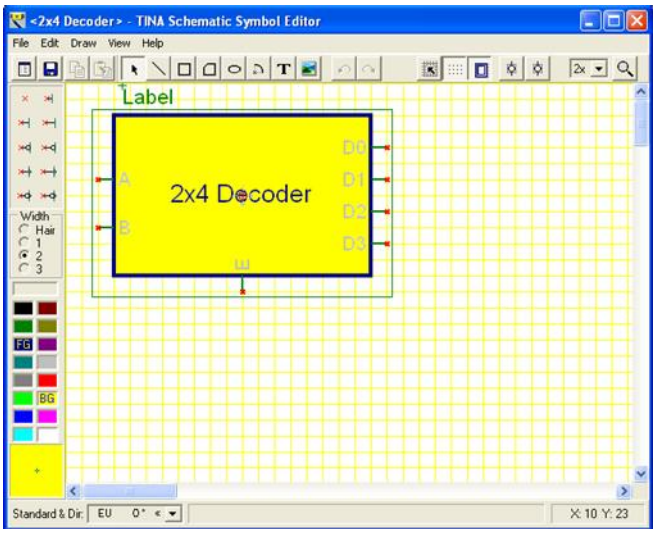

**Εικόνα 23.** Αλλαγές στο πλαίσιο διαλόγου για την εισαγωγή εισόδων A, B, E και εξόδων D0, D1, D2, D3.

Θεωρούμε ότι στην αριστερή πλευρά του block θα έχουμε τις εισόδου A, B, στην κάτω πλευρά, την είσοδο επίπτρεψης E και στη δεξιά πλευρά, τις εξόδου D0 ως D3. Η μορφή του Block

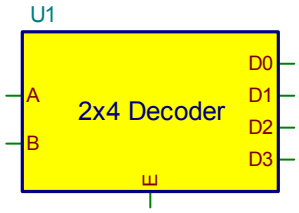
Wizard μετά τις αλλαγές μας γίνεται όπως στην **Εικόνα 23**Εικόνα . Μετά πατάμε το πλήκτρο **OK**.

Αμέσως ενεργοποιείται ο **Schematic Symbol Editor** που δείχνει τη μορφή του block του κυκλώματος που θέλουμε να δημιουργήσουμε (**Εικόνα 24**). Από μόνος του ο **Schematic Symbol Editor** είναι ένα λογισμικό σχεδίασης και επεξεργασίας σχημάτων με πολλές δυνατότητες τις οποίες δε θα αναλύσουμε στο σημείο αυτό, αλλά μπορούμε να τις διαβάσουμε από το αρχείο βοηθείας του.

Εφόσον δε θέλουμε να κάνουμε άλλες αλλαγές στη μορφή του block, επιλέγουμε **File ➔ OK** από το μενού του **Schematic Symbol Editor**. Αμέσως μπορούμε να τοποθετήσουμε το block που φαίνεται στην **Εικόνα** στην επιφάνεια σχεδίασης (**Εικόνα 25**).

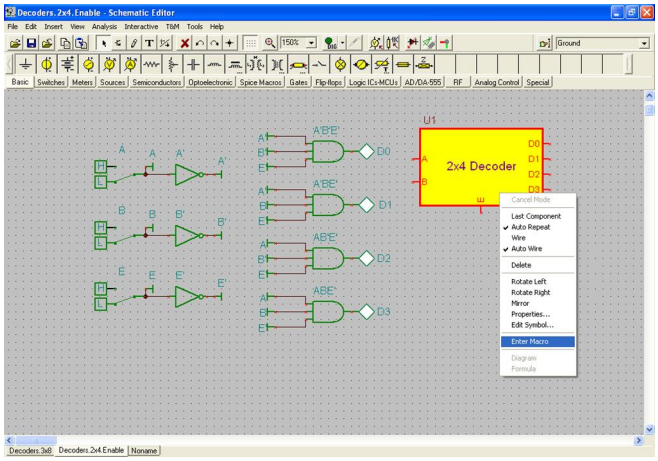



**Εικόνα 24.** Το περιβάλλον του schematic symbol editor.




**Εικόνα 25.** Σχηματικό σύμβολο του 2x4 αποκωδικοποιητή.

Τώρα, πρέπει να δημιουργήσουμε τα περιεχόμενα του block. Δηλαδή μέσα στο block θα βρίσκεται «κρυμμένο» το κύκλωμά μας. Για το σκοπό αυτό με το ποντίκι πάνω στο block κάνουμε δεξί κλικ και από το μενού που προκύπτει επιλέγουμε **Enter Macro**, όπως φαίνεται στην **Εικόνα 26**.

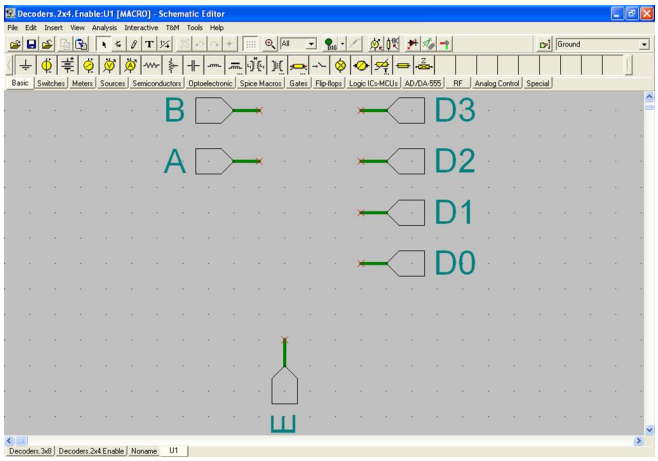



**Εικόνα 26.** Τοποθέτηση του συμβόλου στο χώρο σχεδίου. Με δεξί κλικ στο πλαίσιο διαλόγου που προκύπτει, επιλέγουμε Enter macro για να εισέλθουμε στο «εσωτερικό» του block και να σχεδιάσουμε τα περιεχόμενά του.

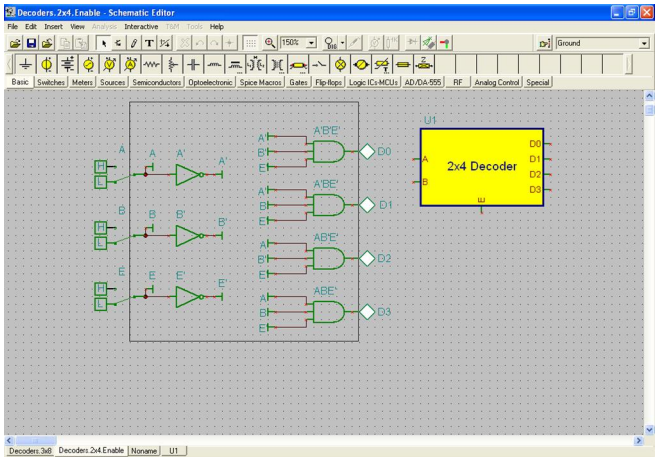
Αμέσως ανοίγει ένα νέο παράθυρο που μας δείχνει τα περιεχόμενα του block ως έχουν μέχρι εκείνη τη στιγμή. Αυτό φαίνεται στην **Εικόνα 27**. Το παράθυρο αυτό έχει όνομα U1 όπως βλέπουμε κάτω δεξιά στις ονομασίες των σχεδίων. Παρατηρούμε ότι η σύνδεση του εσωτερικού του block με τον έξω κόσμο γίνεται μέσω των ακροδεκτών  . (**Εικόνα 28**). Επιστρέφουμε στην καρτέλα σχεδίου του αποκωδικοποιητή με τις πύλες, και επιλέγουμε το κομμάτι που πρέπει να τοποθετήσουμε μέσα στο block. Αυτό φαίνεται στην **Εικόνα 29**.

Το τμήμα που επιλέξαμε γίνεται κόκκινο. Επιλέγουμε **Edit ➔ Copy** όπως στην **Εικόνα 30**.

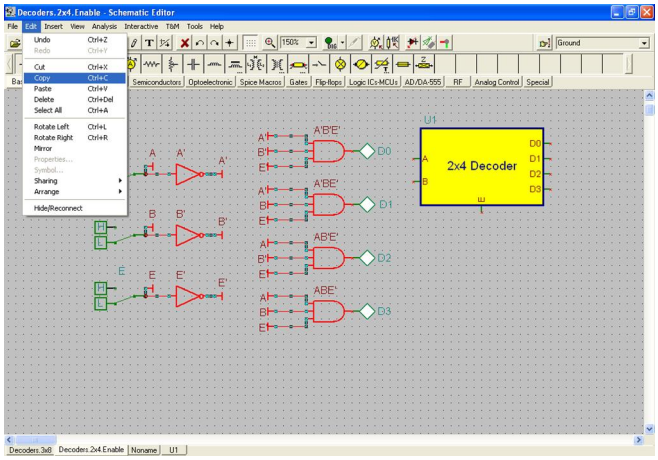
Μεταφερόμαστε στο εσωτερικό του Block U1 και επιλέγουμε **Edit ➔ Paste** όπως φαίνεται στην **Εικόνα 31**.



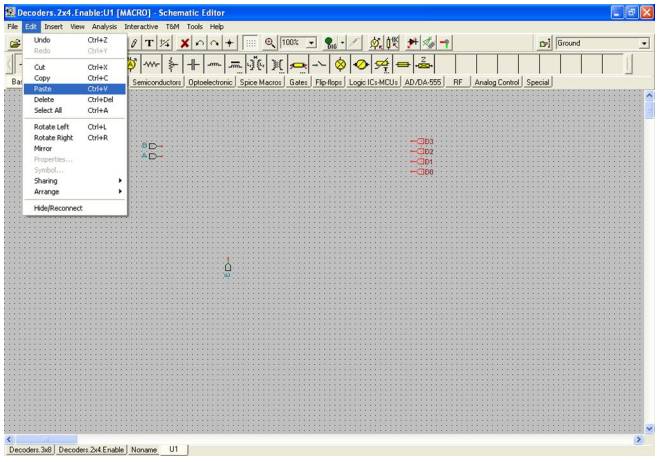

**Εικόνα 27.** Στο εσωτερικό του block, αρχικά υπάρχουν μόνον οι ακροδέκτες σύνδεσης με το εξωτερικό περιβάλλον.




**Εικόνα 28.** Επιστρέφουμε στην καρτέλα του σχηματικού και επιλέγουμε το κομμάτι του κυκλώματος που θέλουμε να αντιγράψουμε και στο εσωτερικό του block.




**Εικόνα 29.** Επιλογή και αντιγραφή του κυκλώματος.




**Εικόνα 30.** Επιστροφή στην καρτέλα του block και επικόλληση του αντιγραμμένου κυκλώματος.

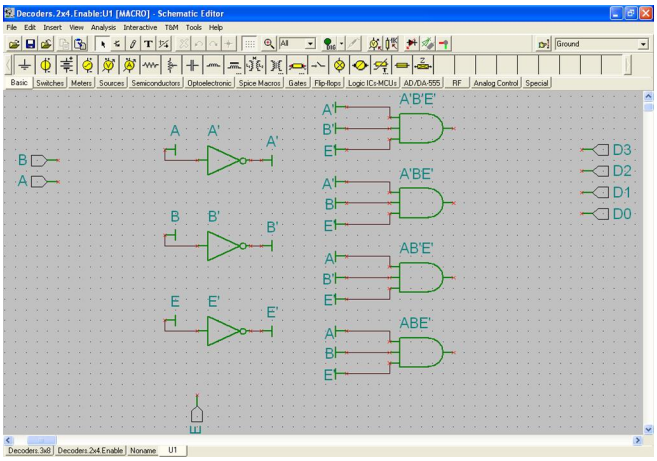
Το σχηματικό μέσα στο U1 έχει τη μορφή που φαίνεται στην **Εικόνα 30**. Αφού κάνουμε τις διασυνδέσεις όπως φαίνεται στην **Εικόνα 31**, είμαστε έτοιμοι και κλείνουμε το U1 πατώντας το

.

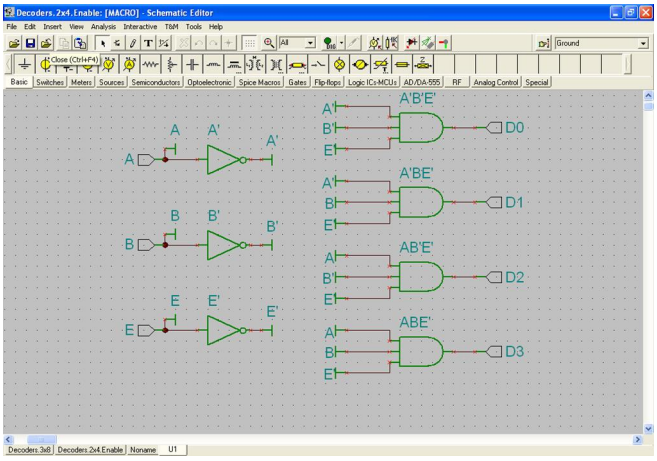
Για να ελέγξουμε ότι τόσο το κύκλωμα με τις πύλες όσο και αυτό που φτιάξαμε σε μορφή block, θα λειτουργούν με τον ίδιο τρόπο, κάνουμε τις αλλαγές και τις προσθήκες που φαίνονται στην **Εικόνα 32** για να ετοιμάσουμε το σχέδιο για διαδραστική προσομοίωση.

Επιλέγουμε την ψηφιακή προσομοίωση (**Digital**) όπως φαίνεται στην **Εικόνα 33** και πατάμε το κουμπί ενεργοποίησης ώστε να γίνει ανοικτό πράσινο. Η προσομοίωση ξεκινά όπως φαίνεται στην **Εικόνα 34**.

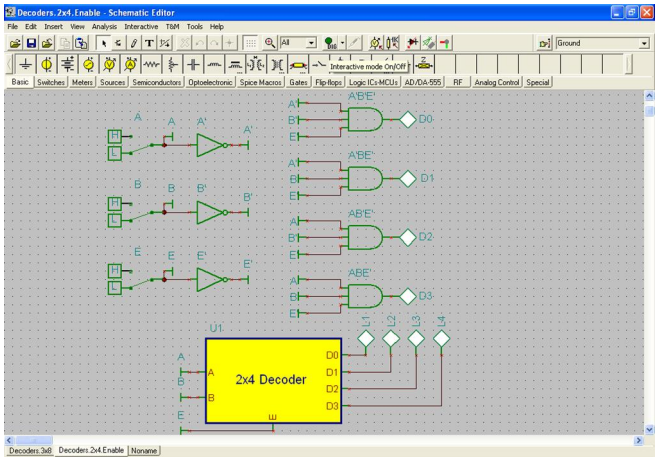
Κάτω δεξιά στο παράθυρο βλέπουμε και το χρόνο προσομοίωσης (**Εικόνα 35**). Επειδή A=0, B=0, E=0, πρέπει να είναι ενεργός ο ενδείκτης D0 πράγμα και το οποίο συμβαίνει όπως βλέπουμε από το κόκκινο χρώμα που έχουν στο σχηματικό ο D0 και ο L1 που αντιστοιχεί στο D0 του block.



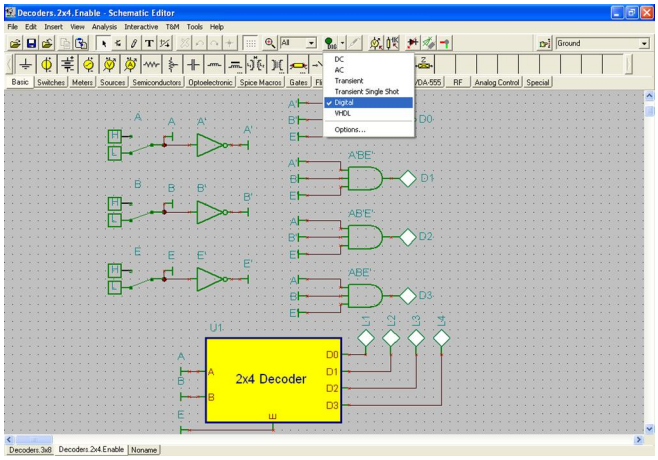

**Εικόνα 31.** Το επικολλημένο κύκλωμα είναι έτοιμο για να συνδεθεί με τους ακροδέκτες.



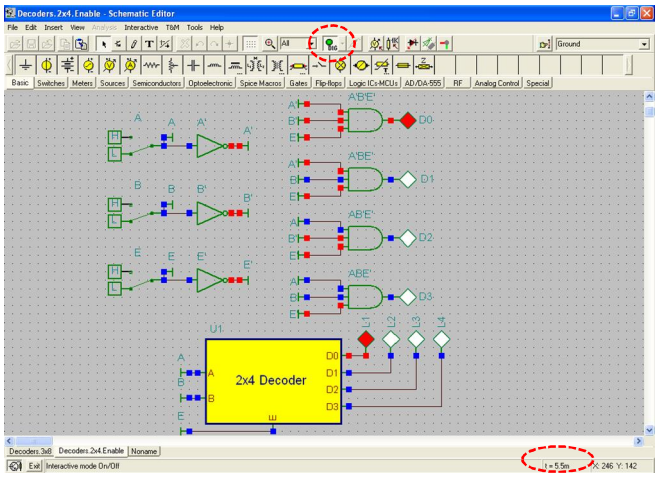

**Εικόνα 32.** Η τελική σύνδεση με τους ακροδέκτες του block.




**Εικόνα 33.** Προετοιμασία για έλεγχο της ορθής λειτουργίας του block.




**Εικόνα 34.** Επιλέγουμε ψηφιακή διαδραστική ανάλυση.




**Εικόνα 35.** Διαδραστική χρονική ανάλυση για ABE = 000.

Επιλέγουμε από τους διακόπες το συνδυασμό 01 για ενεργοποίηση της εξόδου D1 (**Εικόνα 36**).

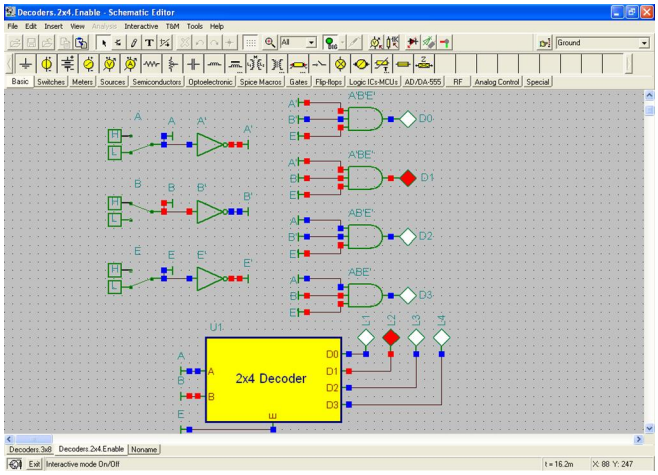
Επιλέγουμε από τους διακόπες το συνδυασμό 10 για ενεργοποίηση της εξόδου D2 (**Εικόνα 37**).

Επιλέγουμε από τους διακόπες το συνδυασμό 11 για ενεργοποίηση της εξόδου D3 (**Εικόνα 38**).

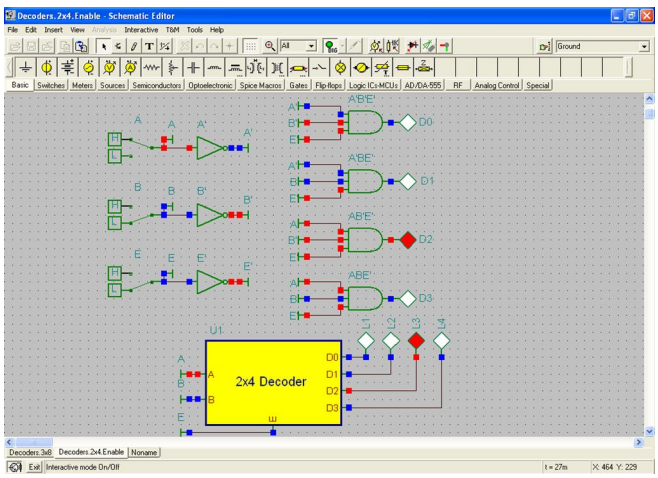
Με το συνδυασμό 11, κάνουμε την E=1 και βλέπουμε ότι οι έξοδοι είναι όλες 0 (**Εικόνα 39**), πράγμα που επαληθεύεται και από τον πίνακα αληθείας του κυκλώματος του αποκωδικοποιητή με επίτρεψη.

**ΣΗΜΑΝΤΙΚΟ**: Μην ξεχνάμε να απενεργοποιούμε το διαδραστικό τρόπο προσομοίωσης ώστε το κουμπί  μετά την προσομοίωση να είναι σκούρο πράσινο που σημαίνει OFF γιατί διαφορετικά η προσομοίωση θα συνεχίζεται στο υπόβαθρο και θα δυσχεραίνεται η λειτουργία του προγράμματος και του υπολογιστή.

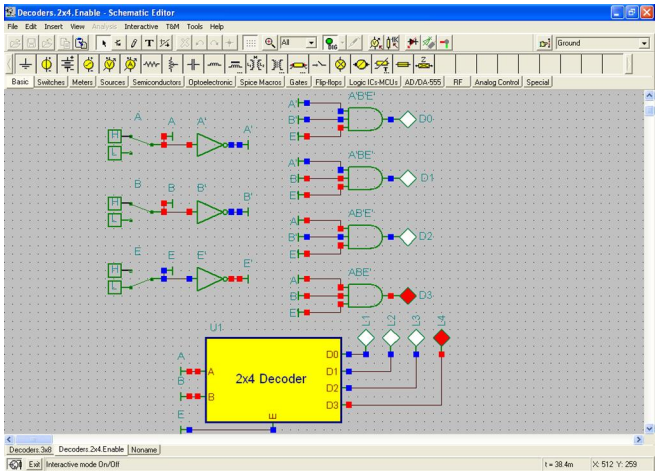




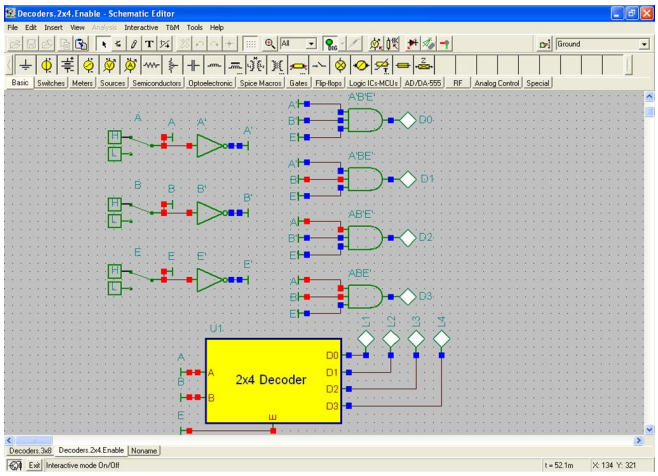

**Εικόνα 36.** Διαδραστική χρονική ανάλυση για ABE = 010.




**Εικόνα 37.** Διαδραστική χρονική ανάλυση για ABE = 100.



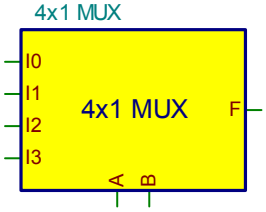

**Εικόνα 38.** Διαδραστική χρονική ανάλυση για ABE = 110.




**Εικόνα 39.** Διαδραστική χρονική ανάλυση για ABE = 111.

## 11. Πολυπλέκτης (Multiplexer)

Ο πολυπλέκτης (multiplexer) είναι ένα ψηφιακό κύκλωμα το οποίο αποτελείται από n γραμμές επιλογής (**select**), $2^n$ γραμμές εισόδου, και μία έξοδο. Συνήθως προσδιορίζουμε τους πολυπλέκτες με την κωδική λέξη MUX και ακολουθεί το πλήθος των εισόδων επί το 1 (δηλαδή το πλήθος των εξόδων). Π.χ. στην **Εικόνα 40** φαίνεται ένας MUX 4x1.




**Εικόνα 40.** Το μπλόκ-διάγραμμα ενός πολυπλέκτη 4x1.

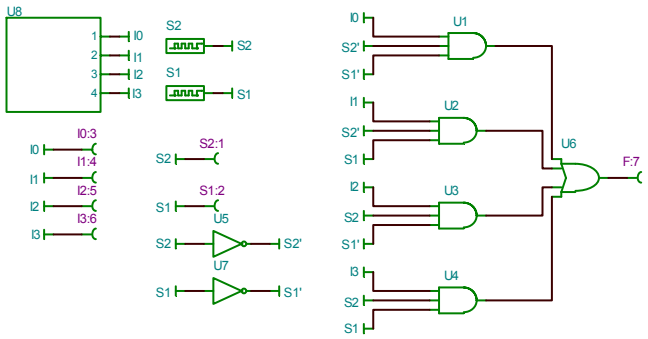
Η λειτουργία των γραμμών επιλογής A και B είναι να προσδιορίζουν ποια από τις γραμμές εισόδου, θα στείλει το σήμα της στην έξοδο F. Ο πίνακας αληθείας αυτού του MUX φαίνεται στον **Πίνακα 7**.

| A | B | F |
|---|---|---|
| 0 | 0 | I0 |
| 0 | 1 | I1 |
| 1 | 0 | I2 |
| 1 | 1 | I3 |

**Πίνακας 7.** Πίνακας αληθείας πολυπλέκτη 4x1.

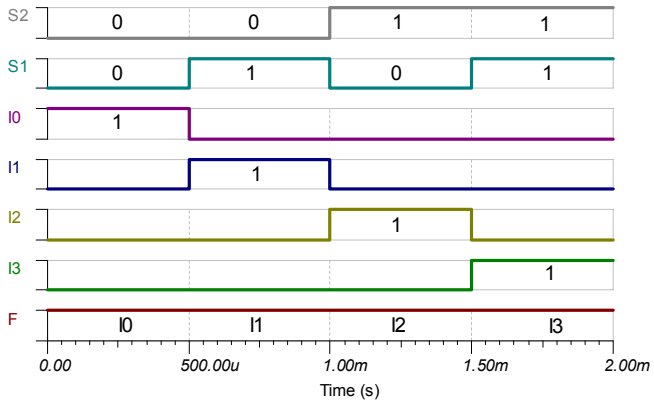
**Άσκηση 1. Να σχεδιάσετε με πύλες το λογικό κύκλωμα ενός MUX 4x1.**

Στην **Εικόνα 41** φαίνεται το κύκλωμα ενός MUX 4x1 με πύλες, υλοποιημένο στο TINA. Στην **Εικόνα 42** φαίνονται τα αποτελέσματα της χρονικής προσομοίωσης που επαληθεύουν τον πίνακα αληθείας του (**Πίνακας 7**).




**Εικόνα 41.** Λογικό κύκλωμα ενός MUX 4x1.

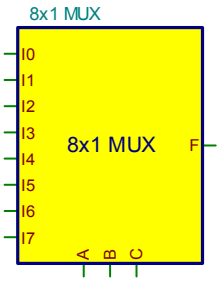
Η λογική συνάρτηση που υλοποιεί ο MUX 4x1 φαίνεται από τη δομή του ότι είναι η:
$F = I0\ 0_2 + I1\ 1_2 + I2\ 2_2 + I3\ 3_2$
F = I0 S2' S1' + I1 S2' S1 + I2 S2 S1' + I3 S2 S1.




**Εικόνα 42.** Προσομοίωση λειτουργίας λογικού κυκλώματος MUX 4x1.

#

Αντίστοιχα, ένας MUX 8x1, θα χρειάζεται 3 γραμμές επιλογής και θα έχει τη δομή μπλοκ που φαίνεται στην **Εικόνα 43** και τον πίνακα αληθείας τού **Πίνακα 8**.

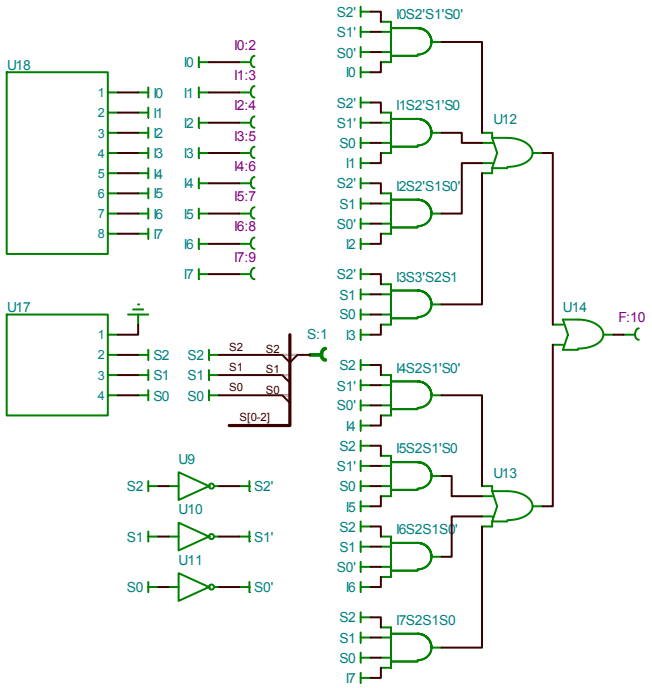



**Εικόνα 43.** Το μπλόκ-διάγραμμα ενός πολυπλέκτη 8x1.

| A | B | C | F |
|---|---|---|---|
| 0 | 0 | 0 | I0 |
| 0 | 0 | 1 | I1 |
| 0 | 1 | 0 | I2 |
| 0 | 1 | 1 | I3 |
| 1 | 0 | 0 | I4 |
| 1 | 0 | 1 | I5 |
| 1 | 1 | 0 | I6 |
| 1 | 1 | 1 | I7 |

**Πίνακας 8.** Πίνακας αληθείας πολυπλέκτη 8x1.

**Άσκηση 2. Να σχεδιάσετε με πύλες το λογικό κύκλωμα ενός MUX 8x1.**



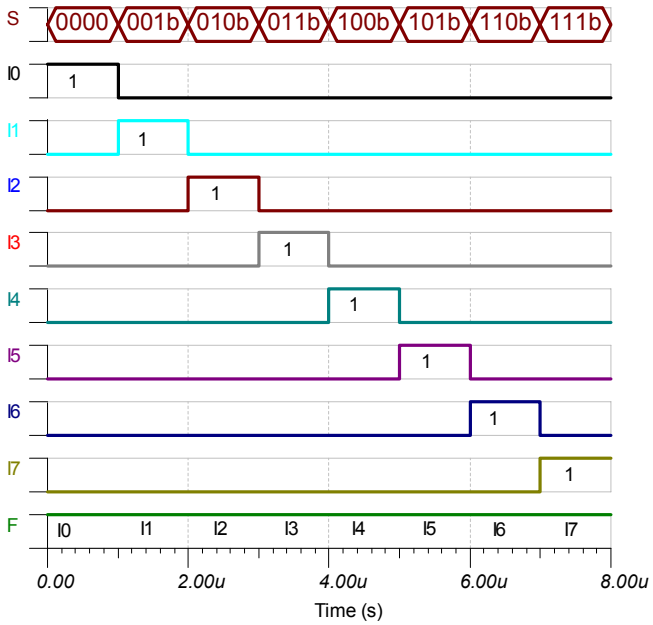

**Εικόνα 44.** Λογικό κύκλωμα ενός MUX 8x1.

Στην **Εικόνα 44** φαίνεται η δομή με πύλες υλοποιημένη στο TINA για τον MUX 8x1. Στην **Εικόνα 45** φαίνονται τα αποτελέσματα της χρονικής προσομοίωσης που επαληθεύουν τον πίνακα αληθείας του (**Πίνακας 8**).

Η λογική συνάρτηση που υλοποιεί ο MUX 4x1 φαίνεται από τη δομή του ότι είναι η:

$F = I0\ 0_2 + I1\ 1_2 + I2\ 2_2 + I3\ 3_2 + I4\ 4_2 + I5\ 5_2 + I6\ 6_2 + I7\ 7_2$

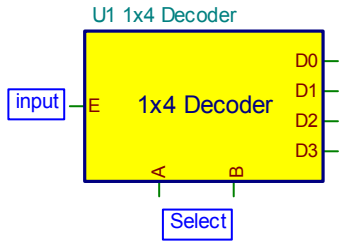
F = I0 S2’ S1’ S0’ + I1 S2’ S1’ S0 + I2 S2’ S1 S0’ +
I3 S2’ S1 S0 + I4 S2 S1’ S0’ + I5 S2 S1’ S0 +
I6 S2 S1 S0’ + I7 S2 S1 S0.




**Εικόνα 45.** Προσομοίωση λειτουργίας λογικού κυκλώματος MUX 4x1.

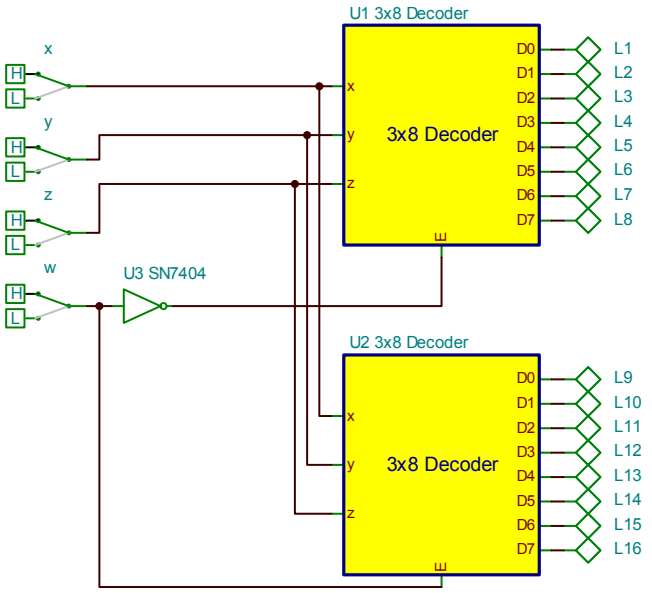
**Άσκηση 3. Ο Αποκωδικοποιητής ως αποπλέκτης.**

Ένας αποκωδικοποιητής με είσοδο επίτρεψης, μπορεί να χρησιμοποιηθεί και ως αποπλέκτης (demultiplexer) (**Εικόνα 46**). Ο αποπλέκτης είναι ένα κύκλωμα που δέχεται πληροφορίες από μια απλή γραμμή και τις μεταβιβάζει σε μια από τις $2^n$ δυνατές γραμμές εξόδου. Η επιλογή μιας συγκεκριμένες γραμμής εξόδου γίνεται ανάλογα με τις τιμές των n γραμμών επιλογής.




**Εικόνα 46.** 1x4 αποκωδικοποιητής ως πολυπλέκτης.

Π.χ. αν οι γραμμές επιλογής είναι AB=10, η έξοδος $D_2$ είναι ίδια με το E, ενώ οι άλλες έξοδοι διατηρούνται στο 1.




**Εικόνα 47.** Σύνδεση δύο αποκωδικοποιητών 3-σε-8, με εισόδους επίτρεψης που συνδέονται για να σχηματίσουν έναν αποκωδικοποιητή σε 4-σε-16.

**Επειδή οι λειτουργίες του αποκωδικοποιητή και του αποπλέκτη λαμβάνονται από το ίδιο κύκλωμα, ένας αποκωδικοποιητής με είσοδο επίτρεψης ονομάζεται συνήθως αποκωδικοποιητής / αποπλέκτης. Το κύκλωμα αυτό γίνεται αποπλέκτης χάρη στην είσοδο επίτρεψης.**

Ο αποκωδικοποιητής δομικά, μπορεί να χρησιμοποιεί πύλες AND, NAND ή NOR.

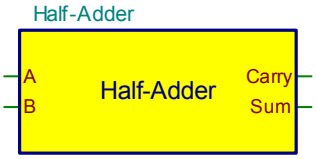
Τα τσιπς αποκωδικοποιητή/αποπλέκτη μπορούν να συνδεθούν μεταξύ τους και να σχηματίσουν ένα μεγαλύτερο κύκλωμα αποκωδικοποιητή. Το κύκλωμα στην **Εικόνα 47** δείχνει τη σύνδεση δύο αποκωδικοποιητών 3-σε-8, με εισόδους επίτρεψης που συνδέονται για να σχηματίσουν έναν αποκωδικοποιητή σε 4-σε-16. Όταν το w=0, «επιτρέπεται» η λειτουργία του πάνω αποκωδικοποιητή και «αποτρέπεται» η λειτουργία του κάτω. Το αντίθετο συμβαίνει όταν w=1.

## 12. Ημιαθροιστής (Half-Adder)

Ο ημιαθροιστής 2-bits είναι το απλούστερο κύκλωμα εκτέλεσης πρόσθεσης δύο bits. Ο πίνακας αληθείας της λειτουργίας του ημιαθροιστή φαίνεται στον **Πίνακα 9**. Το μπλοκ – διάγραμμα ενός ημιαθροιστή 2-bits φαίνεται στην **Εικόνα 48.** Έχει δύο εισόδους τις A και B και δύο εξόδους τις Sum (το άθροισμα A+B) και Carry (για την περίπτωση κρατουμένου).

| A | B | Carry | Sum |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 |

**Πίνακας 9.**Πίνακας αληθείας



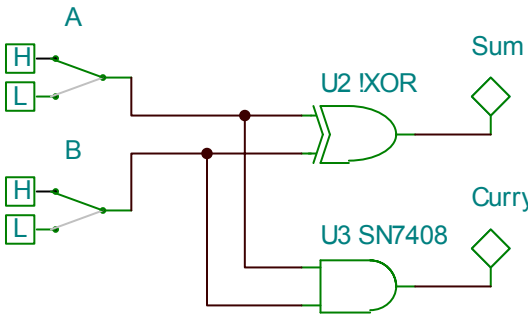

**Εικόνα 48.** Block σύμβολο ημιαθροιστή.

Η λογική συνάρτηση της πράξης που εκτελεί ο ημιαθροιστής είναι:

Sum = A' B + A B' = XOR(A,B) = A ⊕B

Carry = A B

Το αντίστοιχο κύκλωμα του ημιαθροιστή με πύλες AND – OR φαίνεται στην **Εικόνα 49**.



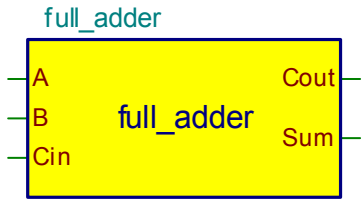

**Εικόνα 49.** Το κύκλωμα με πύλες του ημιαθροιστή δύο bits.

## 13. Πλήρης αθροιστής (Full Adder)

Η διαφορά του πλήρους αθροιστή δύο bits από τον αντίστοιχο ημιαθροιστή, είναι ότι έχει μια επιπλέον είσοδο (τη Cin) σε ρόλο κρατουμένου εισόδου. Ο πίνακας αληθείας της λειτουργίας του πλήρους αθροιστή φαίνεται στον **Πίνακα 10.** Το μπλοκ – διάγραμμα του πλήρους αθροιστή δύο bits φαίνεται στην **Εικόνα 50**.

| A | B | Cin | Cout | Sum |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

**Πίνακας 10.** Πίνακας αληθείας πλήρους αθροιστή.

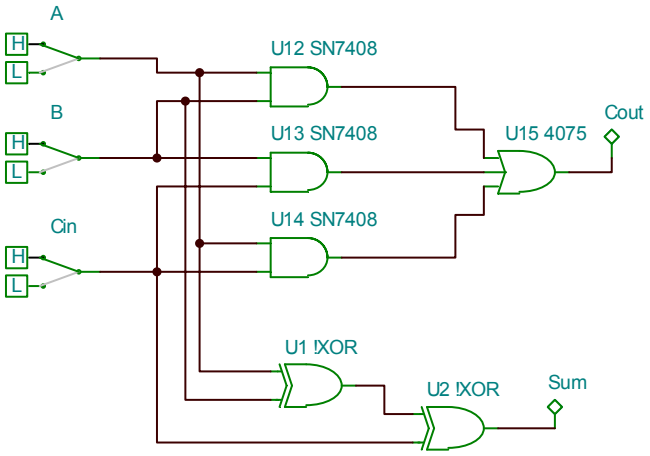



**Εικόνα 50.** Μπλοκ – διάγραμμα πλήρους αθροιστή 2 bits.

Έχει τρεις εισόδους τις A και B και Cin και δύο εξόδους τις Sum (το άθροισμα A+B) και Cout (για την περίπτωση κρατουμένου). Η λογική συνάρτηση της πράξης που εκτελεί ο πλήρης αθροιστής είναι:

Sum = Cin' A' B + Cin' A B' + Cin A B + Cin A' B' = Cin ⊕ A ⊕ B
Cout = Cin' A B + Cin A' B + Cin A B' + Cin A B = A B + Cin B + Cin A

Το αντίστοιχο κύκλωμα με πύλες φαίνεται στην **Εικόνα 51**.




**Εικόνα 51.** Δομή κυκλώματος με πύλες για πλήρη αθροιστή 2bits.

## 14. Ημιαφαιρέτης

Θα σχεδιάσουμε έναν ημιαφαιρέτη με εισόδους x και y και εξόδους D και B. Το κύκλωμα θα αφαιρεί τα bits x – y και θα τοποθετεί τη διαφορά στο D και το δανεικό bit στο B.

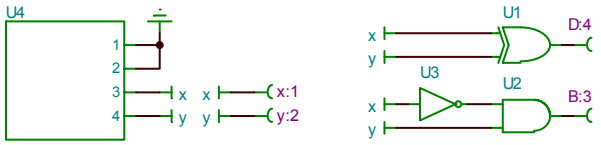
Υλοποιούμε τον πίνακα αληθείας του ημιαφαιρέτη, όπως φαίνεται στον **Πίνακα 11**.

Οι εκφράσεις Boole για τις εξόδους όπως προκύπτουν από τον πίνακα αληθείας είναι:
D=x'y+xy' = xor(x,y)
B=x'y

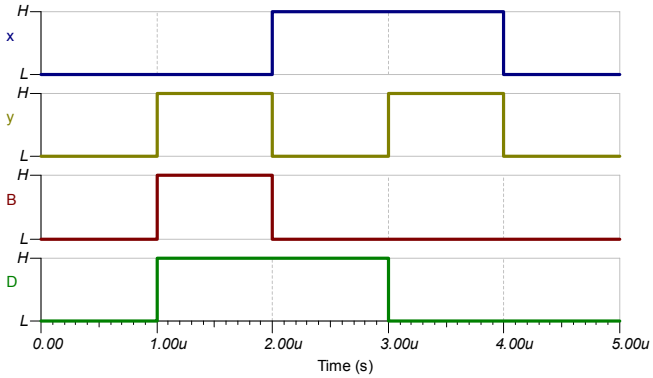
| x | y | D | B |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |

**Πίνακας 11.** Πίνακας αληθείας ημιαφαιρέτη.

Το λογικό διάγραμμα του κυκλώματος φαίνεται στην **Εικόνα 52**. Η χρονική προσομοίωση για την επαλήθευση του πίνακα αληθείας φαίνεται στην **Εικόνα 53**.




**Εικόνα 52.** Κύκλωμα ημιαφαιρέτη.




**Εικόνα 53.** Προσομοίωση λειτουργίας ημιαφαιρέτη.

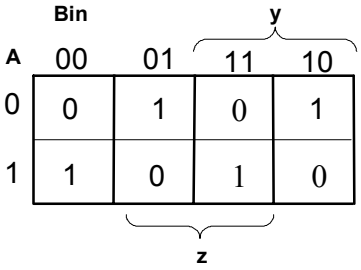
## 15. Πλήρης αφαιρέτης

Θα σχεδιάσουμε έναν πλήρη αφαιρέτη με τρεις εισόδους A, B, και Bin όπου Cin είναι σε ρόλο δανεικού εισόδου και δύο εξόδους, το δανεικό εξόδου Bout και τη διαφορά Diff.
Ο πίνακας αληθείας του κυκλώματος φαίνεται στη συνέχεια:

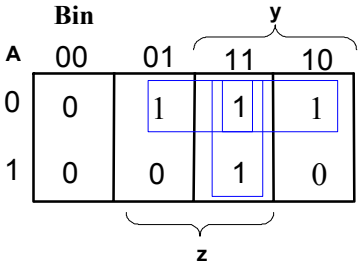
| A | B | Bin | Diff | Bout |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

**Πίνακας 12.** Πίνακας αληθείας πλήρους αφαιρέτη.

Οι απλοποιημένες εκφράσεις Boole για καθεμία από τις εξόδους προκύπτουν μέσω των ΧΚ που φαίνονται στην **Εικόνα 54**.



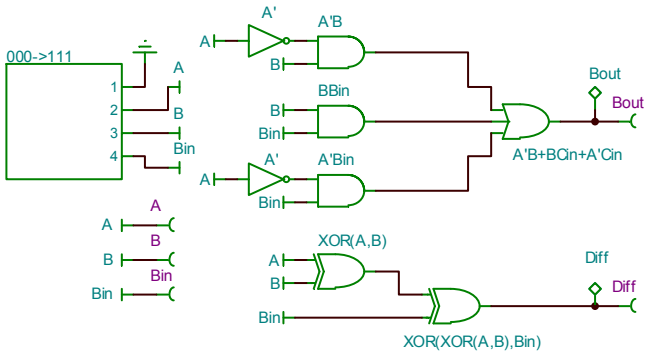

Diff= xor(A,B,Bin)



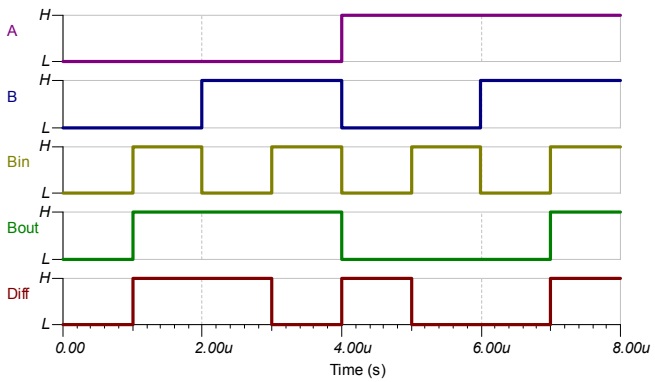

Bout=BCin+A'B+A'Bin

**Εικόνα 54.** ΧΚ για την απλοποίηση της διαφοράς Diff και του δανεικού Bout.

Το υλοποιημένο κύκλωμα με πύλες φαίνεται στην **Εικόνα 55** ενώ στην **Εικόνα 56** φαίνεται η χρονική προσομοίωση του, που επαληθεύει τον πίνακα αληθείας του.

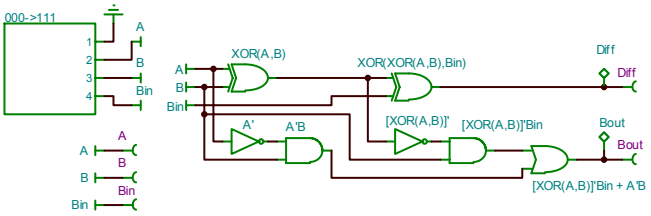



**Εικόνα 55.** Υλοποίηση πλήρους αφαιρέτη.




**Εικόνα 56.** Προσομοίωση λειτουργίας πλήρους αφαιρέτη.

Εναλλακτικά μπορούμε να υλοποιήσουμε τον πλήρη αφαιρέτη με τη χρήση δύο ημιαφαιρετών φαίνεται στην **Εικόνα 57**.




**Εικόνα 57.** Υλοποίηση πλήρους αφαιρέτη με χρήση δύο ημιαφαιρετών.

## 16. Παράλληλος δυαδικός αθροιστής

Ο **παράλληλος δυαδικός αθροιστής** (**PBA**) πραγματοποιεί την πρόσθεση χρησιμοποιώντας δύο είδη αθροιστών: τον ημιαθροιστή (HA) και τον πλήρη αθροιστή (FA).

Η πρόσθεση δύο n-bit αριθμών χρησιμοποιεί έναν ημιαθροιστή στο LSB και n-1 πλήρεις αθροστές για τα υπόλοιπα bits.

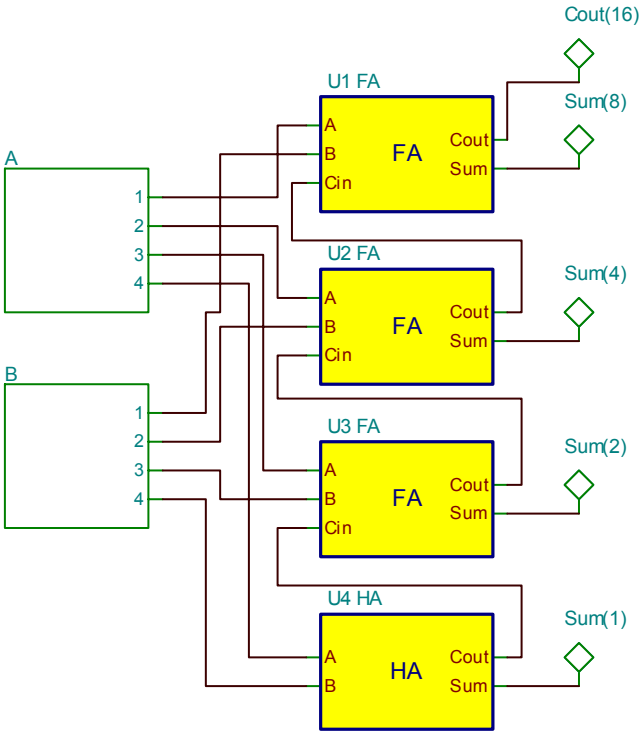
Η πρόσθεση των bits γίνεται παράλληλα επειδή διαθέτουμε n κυκλώματα αθροιστών, ένα για κάθε bit.

Το κρατούμενο μεταφέρεται μέσω των διαδοχικών σταδίων άθροισης από το LSB στο MSB. Το αποτέλεσμα της πράξης θα είναι σωστό μόνον αφότου το κρατούμενο φτάσει στο MSB. Λόγω της διαδρομής που ακολουθεί το κρατούμενο λέμε ότι «κυματίζει» (ripples).

Κάθε στάδιο στη διαδικασία εισάγει καθυστέρηση. Αν ο αριθμός των bits είναι μεγάλος, το κρατούμενο θα χρειαστεί περισσότερο χρόνο να κυματίσει μέχρι το MSB οπότε το κύκλωμα του PBA θα είναι αργό. Υπάρχουν διάφορες εναλλακτικές σχεδιάσεις που αντιμετωπίζουν αυτή την περίπτωση.

Λόγω της αναμονής μέχρι να φτάσει το κρατούμενο από το προηγούμενο στάδιο στη σωστή θέση, η διαδικασία της πρόσθεσης δεν είναι αυστηρά παράλληλη και ο αθροιστής αυτός λέγεται και ψευδοπαράλληλος αθροιστής. Λέγεται επίσης και αθροιστής κυμάτωσης λόγω της διαδρομής που ακολουθεί το κρατούμενο.

Το μπλόκ-διάγραμμα ενός παράλληλου δυαδικού αθροιστή 4 bits φαίνεται στην **Εικόνα 58.**




**Εικόνα 58.** Το μπλόκ-διάγραμμα ενός παράλληλου δυαδικού αθροιστή 4 bits.

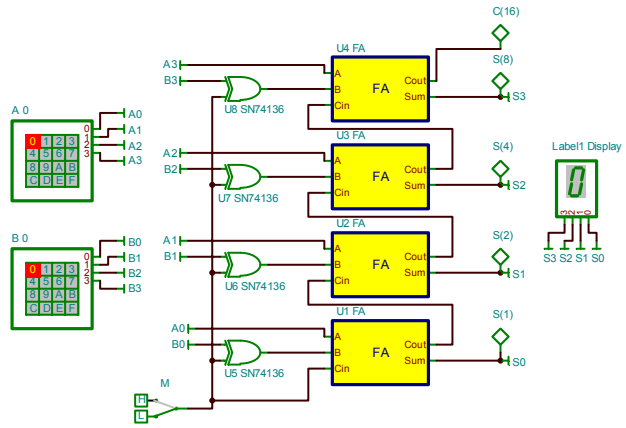
## 17. Παράλληλος δυαδικός αθροιστής-αφαιρέτης

Χρησιμοποιώντας την έννοια του συμπληρώματος ως προς 2, μπορούμε να αναπαραστήσουμε αρνητικούς αριθμούς και χρησιμοποιώντας αθροιστή να κάνουμε την πράξη της αφαίρεσης.

Στην **Εικόνα 59** παρουσιάζεται το κύκλωμα του παράλληλου αθροιστή – αφαιρέτη 4bits.

Οι πύλες XOR χρησιμοποιούνται για να πάρουμε το συμπλήρωμα ως προς 1 ενός αριθμού (του Β στα παραδείγματα).

Η XOR βγάζει 1 όταν τα bits εισόδου είναι διαφορετικά. Αλλιώς βγάζει 0.




**Εικόνα 59.** Παράλληλος αθροιστής – αφαιρέτης 4bits.

Όταν η είσοδος **M είναι 0** το κύκλωμα λειτουργεί ως αθροιστής διότι οι πύλες XOR απλά μεταφέρουν τον Β όπως είναι στους πλήρεις αθροιστές.

A: 0110
B: 1101
1101 (Μετά την XOR, με M=0, ο Β δεν αλλάζει)
A: 0110 ($=6_{10}$
B: 1101 ($=13_{10}$
----------(+)
10011 ($=19_{10}$

Όταν η είσοδος **M είναι 1** το κύκλωμα λειτουργεί ως αφαιρέτης διότι οι πύλες XOR παράγουν το συμπλήρωμα ως προς 1 του αριθμού Β.

A: 0110
B: 1101
0010 (Μετά την XOR, με M=1, ο Β αλλάζει στο Σ1 (B'))
A: 0110 ($=6_{10}$
B: 1101 ($=13_{10}$
B': 0010 ($=-13_{10}$
----------(+ A με το B')
1000 (=Σ1
0111 ($=7_{10}$

Στην περίπτωση αφαίρεσης, όταν συμβεί υπερχείλιση το bit υπερχείλισης πρέπει να προστεθεί πίσω στο LSB.

A: 0110
B: 0100
1011 (Μετά την XOR, με M=1, ο Β αλλάζει στο Σ-1 (B'))
A: 0110 ($=6_{10}$
B: 0100 ($=4_{10}$
B': 1011 ($=-4_{10}$
----------(+ A με το B')
10001 (=Σ1
\+ **1** Η υπερχείλιση προστίθεται πίσω στο LSB (ισοδυναμεί με συμπλήρωμα)
----------(+)
0010 ($=2_{10}$

## 18. Ασκήσεις

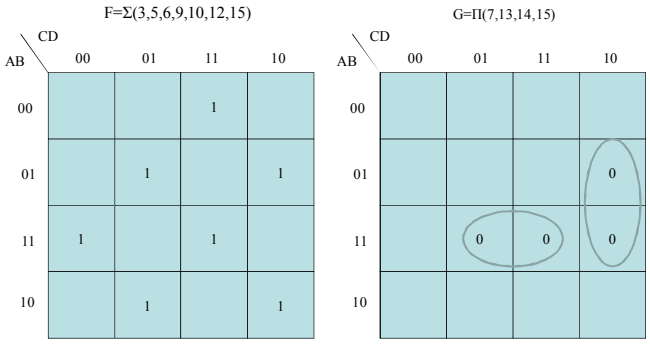
1. **Θα σχεδιάσουμε με πύλες AND-OR κύκλωμα 4 εισόδων και 2 εξόδων ως εξής: Η έξοδος F είναι 1 όταν το πλήθος των 1 στις γραμμές εισόδου είναι άρτιο και 0 όταν είναι περιττό. Η έξοδος G είναι 0 όταν το**

**πλήθος των 1 στις γραμμές εισόδου είναι πλειοψηφία έναντι των 0 και 1 όταν είναι μειοψηφία ή ίσο με το πλήθος των 0.**

Λύση.


| | 8 | 4 | 2 | 1 | | |
|---|---|---|---|---|---|---|
| 10δικός | A | B | C | D | F | G |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 2 | 0 | 0 | 1 | 0 | 0 | 1 |
| 3 | 0 | 0 | 1 | 1 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 | 1 |
| 5 | 0 | 1 | 0 | 1 | 1 | 1 |
| 6 | 0 | 1 | 1 | 0 | 1 | 1 |
| 7 | 0 | 1 | 1 | 1 | 0 | 0 |
| 8 | 1 | 0 | 0 | 0 | 0 | 1 |
| 9 | 1 | 0 | 0 | 1 | 1 | 1 |
| 10 | 1 | 0 | 1 | 0 | 1 | 1 |
| 11 | 1 | 0 | 1 | 1 | 0 | 1 |
| 12 | 1 | 1 | 0 | 0 | 1 | 1 |
| 13 | 1 | 1 | 0 | 1 | 0 | 0 |
| 14 | 1 | 1 | 1 | 0 | 0 | 0 |
| 15 | 1 | 1 | 1 | 1 | 1 | 0 |

**Εικόνα 60.** Παράδειγμα μπλοκ διαγράμματος συνδυαστικού κυκλώματος με 4 εισόδους και 2 εξόδους και ο πίνακας αληθείας του.




**Εικόνα 61.** ΧΚ για τις 2 εξόδους του συνδυαστικού κυκλώματος. Μετά τις δυνατές απλοποιήσεις μπορούμε να προχωρήσουμε στην υλοποίηση του κυκλώματος με πύλες.

Το μπλοκ-διάγραμμα και ο πίνακας αληθείας του κυκλώματος φαίνονται στην **Εικόνα 60** και στην **Εικόνα 61** οι ΧΚ για τις 2 εξόδους του συνδυαστικού κυκλώματος. Μετά τις δυνατές απλοποιήσεις μπορούμε να προχωρήσουμε στην υλοποίηση του κυκλώματος με πύλες.

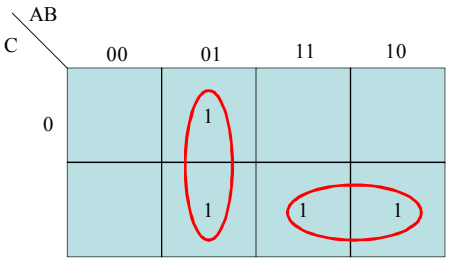
## 2. Κυκλωματική υλοποίηση από πίνακα αληθείας

Στην **Εικόνα 62** παρουσιάζεται ο πίνακας αληθείας και ο ΧΚ για την απλοποίηση του λογικού κυκλώματος, καθώς και η τελική απλοποιημένη μορφή της λογικής συνάρτησης. Το κύκλωμα αυτό είναι τριών-επιπέδων, επειδή το μέγιστο πλήθος πυλών από τις οποίες περνάει το σήμα είναι 3 (από την είσοδο στην έξοδο): 1ο επίπεδο= NOT, 2ο επίπεδο= AND, 3ο επίπεδο=OR.

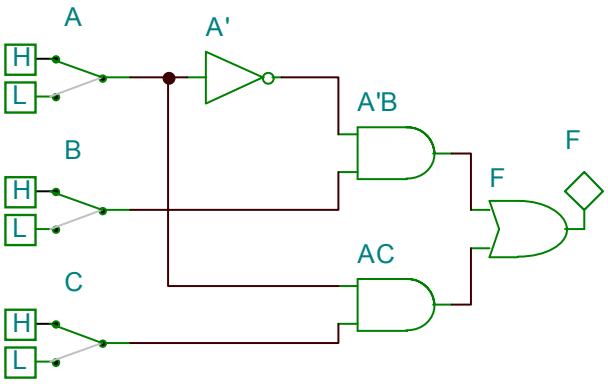
Υποθέσαμε ότι τα σήματα είναι διαθέσιμα μόνο στην κανονική μορφή και όχι στη συμπληρωματική και για το λόγο αυτό χρησιμοποιούμε την πύλη NOT όταν χρειαζόμαστε το συμπλήρωμα. Πρακτικά τόσο η κανονική όσο και η συμπληρωματική μορφή του σήματος είναι διαθέσιμη, οπότε η πρώτη πύλη NOT δεν είναι απαραίτητη.

Άρα το κύκλωμα αυτό μπορεί να θεωρηθεί υλοποίηση AND-OR δύο επιπέδων. Η υλοποίησή του με πύλες AND – OR φαίνεται στην **Εικόνα 63**.

| A | B | C | F |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |

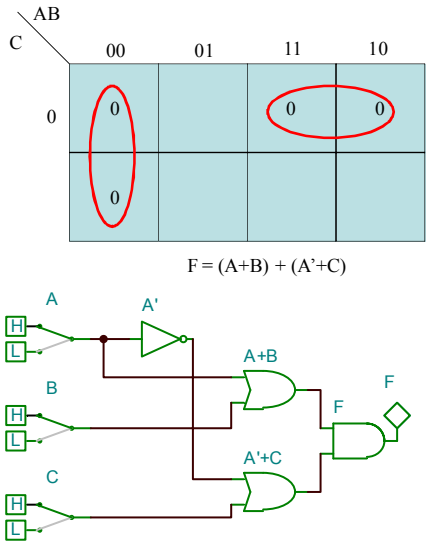



**Εικόνα 62.** Ο πίνακας αληθείας και ο ΧΚ για την απλοποίηση του λογικού κυκλώματος, καθώς και η τελική απλοποιημένη μορφή της λογικής συνάρτησης.




**Εικόνα 63.** Υλοποίηση AND- OR της F = A'B+AC.

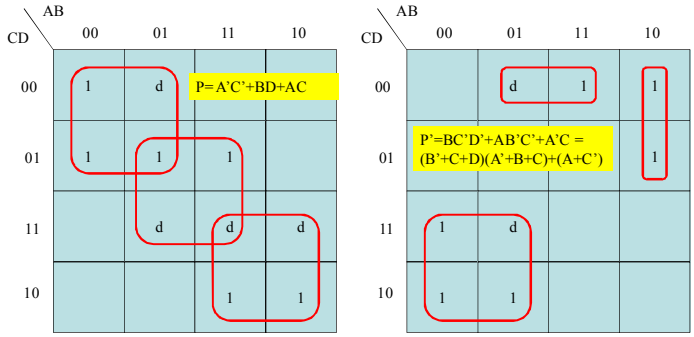
Στην **Εικόνα 64** φαίνεται η OR – AND υλοποίηση της ίδιας λογικής συνάρτησης, όπου χρησιμοποιήσαμε τα 0 του ΧΚ για την απλοποίηση.




**Εικόνα 64.** OR – AND υλοποίηση.

3. **Θα υλοποίησουμε τις P και P', όπου: P(A,B,C,D) = Σ(0,1,5,10,13,14) + d(4,7,11,15).**

Λύση.




**Εικόνα 65.** ΧΚ για την απλοποίηση των λογικών συναρτήσεων.

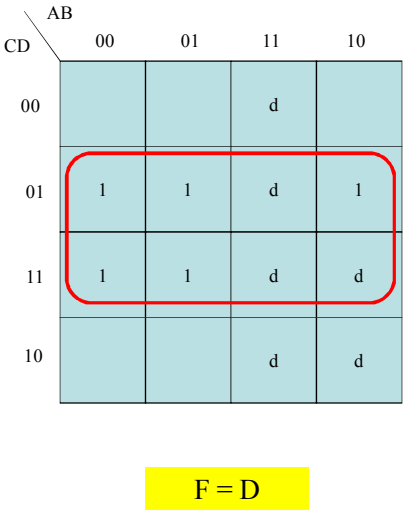
Στην **Εικόνα 65** φαίνεται ο ΧΚ για την P και P'. Παρατηρούμε ότι η υλοποίηση για την P χρειάζεται τα σήματα A,A',B,C,C',D δηλαδή 6 εισόδους, ενώ για την P' χρειάζεται τα σήματα A,A',B,B',C,C',D,D' ή 8 εισόδους.

4. **BCD μορφή αριθμών. Να σχεδιάσετε ένα συνδυαστικό κύκλωμα που να δέχεται δεκαδικά ψηφία σε BCD μορφή και να παράγει 1 αν η είσοδός του αντιστοιχεί σε περιττό αριθμό.**

Λύση.

Στην **Εικόνα 66** φαίνεται ο πίνακας αληθείας και ο ΧΚ για την απλοποίηση της λογικής συνάρτησης. Επειδή εκμεταλλευόμαστε τους αδιάφορους όρους, πετυχαίνουμε σημαντική απλοποίηση.

| A | B | C | D | F |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | **d** |
| 1 | 0 | 1 | 1 | **d** |
| 1 | 1 | 0 | 0 | **d** |
| 1 | 1 | 0 | 1 | **d** |
| 1 | 1 | 1 | 0 | **d** |
| 1 | 1 | 1 | 1 | **d** |

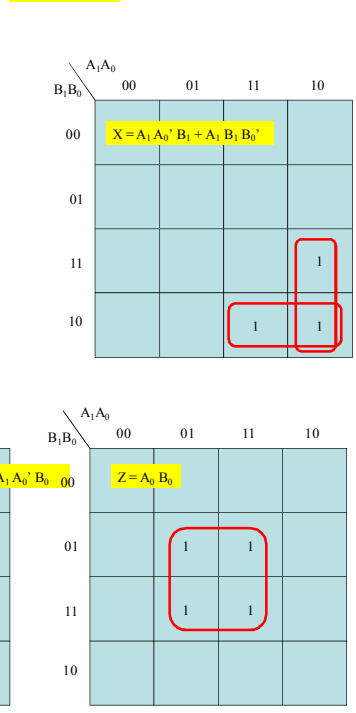



**Εικόνα 66.** Ο πίνακας αληθείας και ο ΧΚ για την απλοποίηση της λογικής συνάρτησης.

5. **Κύκλωμα παραγωγής γινομένου ακεραίων. Να σχεδιάσετε ένα συνδυαστικό κύκλωμα που να παράγει το γινόμενο δύο 2bit αριθμών.**

Λύση.

| $A_1$ | $A_0$ | $B_1$ | $B_0$ | W | X | Y | Z |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | **0** | **0** | **0** | **0** |
| 0 | 0 | 0 | 1 | **0** | **0** | **0** | **0** |
| 0 | 0 | 1 | 0 | **0** | **0** | **0** | **0** |
| 0 | 0 | 1 | 1 | **0** | **0** | **0** | **0** |
| 0 | 1 | 0 | 0 | **0** | **0** | **0** | **0** |
| 0 | 1 | 0 | 1 | **0** | **0** | **0** | **1** |
| 0 | 1 | 1 | 0 | **0** | **0** | **1** | **0** |
| 0 | 1 | 1 | 1 | **0** | **0** | **1** | **1** |
| 1 | 0 | 0 | 0 | **0** | **0** | **0** | **0** |
| 1 | 0 | 0 | 1 | **0** | **0** | **1** | **0** |
| 1 | 0 | 1 | 0 | **0** | **1** | **0** | **0** |
| 1 | 0 | 1 | 1 | **0** | **1** | **1** | **0** |
| 1 | 1 | 0 | 0 | **0** | **0** | **0** | **0** |
| 1 | 1 | 0 | 1 | **0** | **0** | **1** | **1** |
| 1 | 1 | 1 | 0 | **0** | **1** | **1** | **0** |
| 1 | 1 | 1 | 1 | **1** | **0** | **0** | **1** |

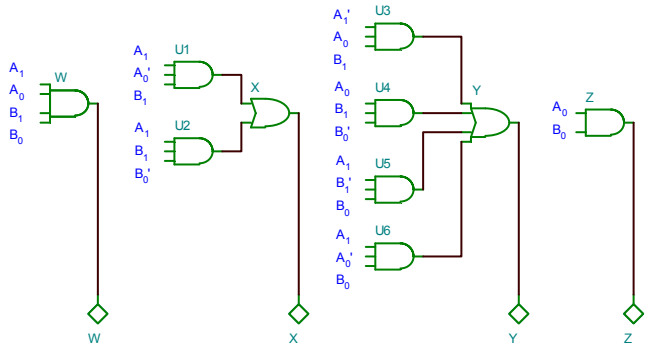



**Εικόνα 67.** Ο πίνακας αληθείας που περιγράφει τη λειτουργία του κυκλώματος

πολλαπλασιασμού καθώς και οι ΧΚ για την απλοποίηση των εξόδων του κυκλώματος.

Επειδή το γινόμενο δύο 2bit αριθμών μπορεί να είναι το πολύ 4bit αριθμός, το κύκλωμα θα έχει 4 εξόδους. Θα έχει και 4 εισόδους. Οι είσοδοι $A_1A_0$ παριστάνουν τον πρώτο ακέραιο (0-3) και οι $B_1B_0$ το δεύτερο ακέραιο (0-3). Το γινόμενο αυτών των δύο θα είναι το πολύ 9 (δηλαδή χρειαζόμαστε 4bit για την αναπαράστασή του).

Στην **Εικόνα 68** φαίνεται ο πίνακας αληθείας που περιγράφει τη λειτουργία του κυκλώματος πολλαπλασιασμού καθώς και τους ΧΚ για την απλοποίηση των εξόδων X, Y, Z. Για την έξοδο W η λογική συνάρτηση προκύπτει κατευθείαν από τον πίνακα αληθείας. Στην **Εικόνα 69** φαίνεται η υλοποίηση του κυκλώματος με πύλες AND-OR.

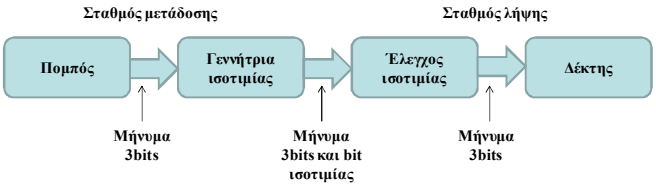



**Εικόνα 68** Υλοποίηση του κυκλώματος πολλαπλασιασμού με πύλες AND-OR.

## 6. Κύκλωμα ψηφιακού πομπού-δέκτη. Έλεγχος ισοτιμίας.

Θεωρούμε το σύστημα στην **Εικόνα 69.** Ο σταθμός μετάδοσης στέλνει μηνύματα των 3bit στο σταθμό λήψης. Ο πομπός παράγει ένα μήνυμα των 3bit. Η γεννήτρια ισοτιμίας (parity) εισάγει στα 3bit του μηνύματος ένα ακόμα bit, το parity bit (μπιτ ισοτιμίας). Το parity bit = 1 αν το πλήθος των 1 στο μήνυμα είναι περιττό, διαφορετικά είναι 0. Άρα ένα 4bit μήνυμα θα φύγει από τον πομπό και θα έχει πάντα άρτιο πλήθος 1. Αυτό είναι ένα σχήμα άρτιου parity. Αν το parity bit είναι τέτοιο ώστε το συνολικό πλήθος των 1 στο μήνυμα να είναι άρτιο, τότε μιλάμε για σχήμα περιττής parity.

Ο ελεγκτής ισοτιμίας (parity checker) στο δέκτη, ελέγχει το εισερχόμενο 4bit μήνυμα για την ισοτιμία του. Αν το πλήθος των 1 είναι άρτιο, τότε το μήνυμα που λαμβάνουμε δεν έχεις σφάλμα. Αν το πλήθος των 1 είναι περιττό, έχει διαγνώσει σφάλμα. Μόλις βρεθεί το μήνυμα με το σφάλμα, πρέπει να ζητηθεί από τον πομπό να ξαναστείλει το μήνυμα. Το προηγούμενο είναι ένα απλό παράδειγμα σχήματα ελέγχου μοναδικού – λάθους σε μήνυμα. Αν θέσουμε το P=0 στο κύκλωμα του parity – checker μπορούμε να πάρουμε από αυτό τη λειτουργία του parity generator! Δηλαδή να χρησιμοποιήσουμε το ίδιο κύκλωμα και για παραγωγή και για έλεγχο της ισοτιμίας.

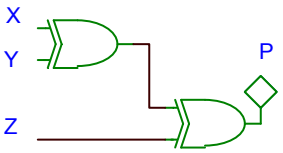



**Εικόνα 69.** Απλό ψηφιακό σύστημα μετάδοσης μυνημάτων.

Γεννήτρια ισοτιμίας

| X | Y | Z | P |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

$P = X \oplus Y \oplus Z$




**Εικόνα 70.** Πίνακας αληθείας και το κύκλωμα της γεννήτριας ισοτιμίας.

Στην **Εικόνα 70** φαίνεται ο πίνακας αληθείας και το κύκλωμα της γεννήτριας ισοτιμίας και στην **Εικόνα 71** ο πίνακας αληθείας και το κύκλωμα ελέγχου ισοτιμίας.

## 7. Κωδικοποιητής προτεραιότητας από οκταδικό σε δυαδικό σύστημα. Προσδιορίστε τον πίνακα αληθείας ενός κωδικοποιητή προτεραιότητας για μετατροπή από οκταδικό (8 είσοδοι) σε δυαδικό (3 έξοδοι). Να υπάρχει και μια έξοδος V η οποία θα υποδεικνύει ότι τουλάχιστον μία από τις εισόδους είναι ενεργή. Η είσοδος με το μεγαλύτερο δείκτη θα έχει τη μεγαλύτερη προτεραιότητα. Ποια θα είναι η τιμή των 4 εξόδων αν οι είσοδοι $D_5$ και $D_3$ είναι 1 ταυτόχρονα;

Λύση.

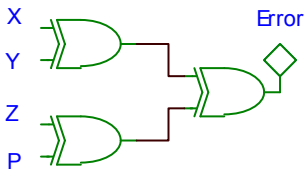
| D 0 | D 1 | D 2 | D 3 | D 4 | D 5 | D 6 | D 7 | X | Y | Z | V |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | X | X | X | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| X | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| X | X | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| X | X | X | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| X | X | X | X | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| X | X | X | X | X | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| X | X | X | X | X | X | 1 | 0 | 1 | 1 | 0 | 1 |
| X | X | X | X | X | X | X | 1 | 1 | 1 | 1 | 1 |

**Πίνακας 13.** Πίνακας αληθείας για τον κωδικοποιητή προτεραιότητας οκταδικού σε δυαδικό.

Έλεγχος ισοτιμίας

| X | Y | Z | P | Error |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 |

$Error = X \oplus Y \oplus Z \oplus P$



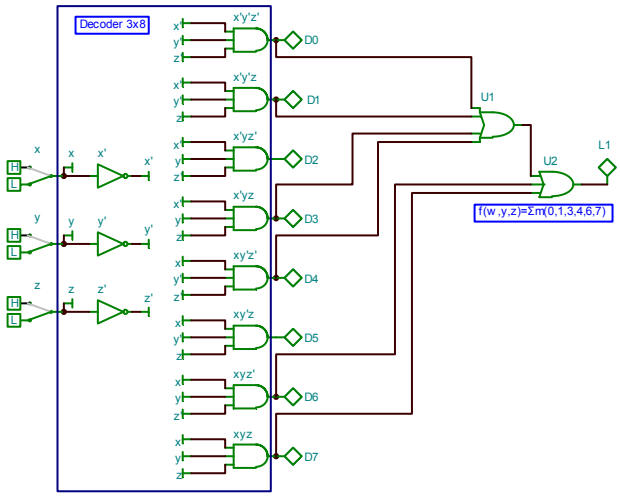

**Εικόνα 71.** Πίνακας αληθείας και το κύκλωμα ελέγχου ισοτιμίας.

Στον **Πίνακα 13** παρουσιάζουμε τον πίνακα αληθείας για τον κωδικοποιητή προτεραιότητας οκταδικού σε δυαδικό. Η έξοδος V υποδεικνύει ότι τουλάχιστον μια από τις εισόδους είναι ενεργή. Η είσοδος με το μεγαλύτερο δείκτη έχει τη μεγαλύτερη προτεραιότητα. Η τιμή των 4 εξόδων οι είσοδοι $D_5$ και $D_3$ είναι 1 ταυτόχρονα, είναι X=1, Y=0, Z=1, V=1.

**8. Να υλοποιηθεί η συνάρτηση f(w,y,z)=Σm(0,1,3,4,6,7) με 3-σε-8 αποκωδικοποιητή και μια πύλη OR.**

Λύση.

Όπως είναι γνωστό, ένας αποκωδικοποιητή παράγει το σύνολο των ελαχιστόρων μιας λογικής συνάρτησης. Με μια πύλη OR, μπορούμε να συλλέξουμε εκείνους τους ελαχιστόρους, στους οποίους η λογική συνάρτηση είναι 1. Το κύκλωμα της δεδομένης λογικής συνάρτησης είναι συνεπώς όπως αυτό που φαίνεται στην **Εικόνα 72**.

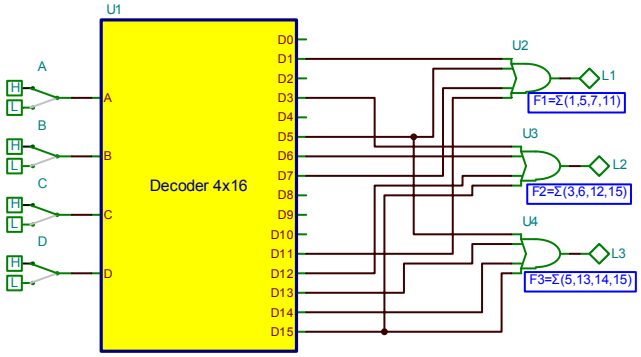



**Εικόνα 72.** Υλοποίηση της συνάρτηση f(w,y,z)=Σm(0,1,3,4,6,7) με 3-σε-8 αποκωδικοποιητή και μια πύλη OR.

**9. Να υλοποιήσετε με decoder (αποκωδικοποιητή) ένα κύκλωμα 4 εισόδων και τριών εξόδων της μορφής F1=Σ(1,5,7,11), F2=Σ(3,6,12,15), F3=Σ(5,13,14,15).**

Λύση.

Στην **Εικόνα 73** φαίνεται η υλοποίηση του κυκλώματος.



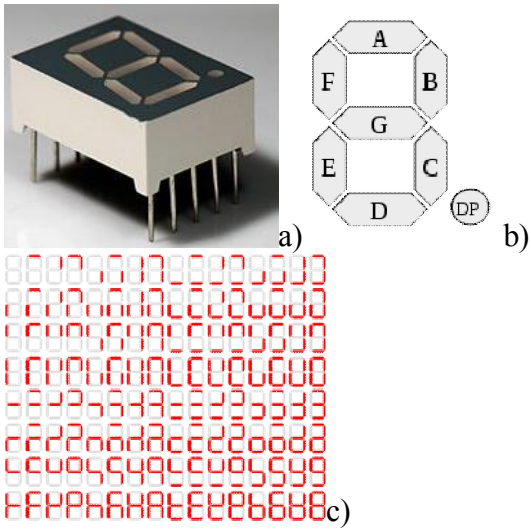
**Εικόνα 73.** Υλοποίηση με αποκωδικοποιητή κυκλώματος 4 εισόδων και τριών εξόδων της μορφής F1=Σ(1,5,7,11), F2=Σ(3,6,12,15), F3=Σ(5,13,14,15).

Η μέθοδος του αποκωδικοποιητή μπορεί να χρησιμοποιηθεί για την υλοποίηση οποιουδήποτε συνδυαστικού κυκλώματος. Ωστόσο, η υλοποίηση του πρέπει να συγκριθεί με όλες τις άλλες δυνατές υλοποιήσεις για να προσδιορίσουμε την καλύτερη λύση. Σε μερικές περιπτώσεις, αυτή η μέθοδος μπορεί να δώσει την καλύτερη υλοποίηση, ειδικά αν το συνδυαστικό κύκλωμα έχει πολλές εξόδους και αν κάθε συνάρτηση εξόδου (ή το συμπλήρωμά της) εκφράζεται με μικρό αριθμό ελαχιστόρων.

Μια συνάρτηση με πολλούς ελαχιστόρους απαιτεί μια πύλη OR με πολλές εισόδους. Μια συνάρτηση F που έχει k ελαχιστόρους μπορεί να εκφραστεί στη συμπληρωματική της μορφή F' με $2^n$-k ελαχιστόρους. Αν ο αριθμός των ελαχιστόρων μιας συνάρτησης είναι μεγαλύτερος από $2^n/2$, τότε η F' μπορεί να εκφραστεί με λιγότερους ελαχιστόρους απ' όσους η F. Σε αυτή την περίπτωση είναι προτιμότερο να χρησιμοποιούμε μια πύλη NOR για να αθροίσουμε τους ελαχιστόρους της F'. Η έξοδος της πύλης NOR θα παράγει την κανονική έξοδο F.

**10. BCD-to-7-segment decoder. Ένας BCD-to-seven-segment decoder είναι ένα συνδυαστικό κύκλωμα που μετατρέπει ένα ψηφίο του δεκαδικού συστήματος στη (BCD) στον κατάλληλο κώδικα για την επιλογή των τμημάτων στην οθόνη 7-τμημάτων (7-segment display) και άρα στην απεικόνιση του αριθμού στην οθόνη. Ο αποκωδικοποιητής θα πρέπει λοιπόν να έχει 7 εξόδους. Η αντιστοιχία των τμημάτων (a, b, c, d, e, f, g) στα «μέρη» του αριθμού φαίνεται στην Εικόνα 74 Στην άσκηση αυτή θέλουμε να σχεδιάσουμε το κύκλωμα που αποκωδικοποιητή BCD-to-seven-segment κάνοντας χρήση ελάχιστου πλήθους πυλών Οι 6 αδιάφοροι όροι θα πρέπει να δίνουν κενή οθόνη (δηλαδή όλα τα τμήματα σε κατάσταση off.**

Λύση.




**Εικόνα 74.** a) Τυπική οθόνη LED 7τμημάτων και με δεκαδικό σημείο (τελεία). b) Αντιστοιχία τμημάτων 7-segment-display για την αναπαράσταση BCD μορφή ψηφίου. DP decimal point (an "eighth segment") is used for the display of non-integer numbers. c) 16x8-πλέγμα που δείχνει τις 128 καταστάσεις της οθόνης 7 τμημάτων [http://en.wikipedia.org/wiki/Seven-segment_display].

Υλοποιούμε τον πίνακα αληθείας του κυκλώματος, όπως φαίνεται στον **Πίνακα 14**. Εκφράζουμε τις εξόδους σε άθροισμα ελαχιστόρων:

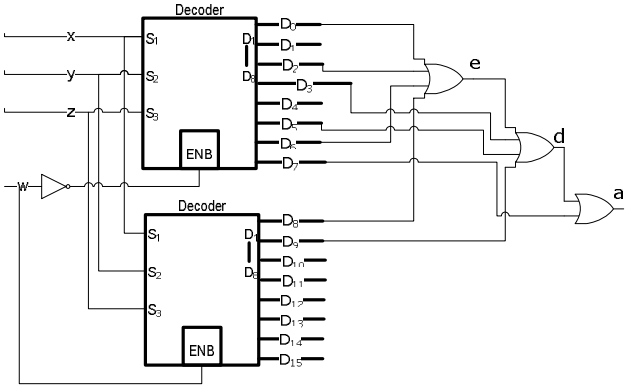
a(w,x,y,z)=Σ(0,2,3,5,6,7,8,9)
b(w,x,y,z)=Σ(0,1,2,3,4,7,8,9)
c(w,x,y,z)=Σ (0,1,3,4,5,6,7,8,9)
d(w,x,y,z)=Σ(0,2,3,5,6,8,9)
e(w,x,y,z)=Σ(0,2,6,8)
f(w,x,y,z)=Σ(0,4,5,6,8,9)
g(w,x,y,z)=Σ(2,3,4,5,6,8,9)

| w | x | y | z | a | b | c | d | e | f | g |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Πίνακας 14.** Πίνακας αληθείας οθόνης 7 τμημάτων.

Είμαστε έτοιμοι για την υλοποίηση του κυκλώματος. Χρησιμοποιούμε δύο 3-to-8-line decoders με εισόδου enable (ENB) συνδεδεμένους κατάλληλα ώστε να σχηματιστεί ένας 4-to-16-line decoder. Με τον τρόπο αυτό παράγονται όλοι οι ελαχιστόροι, οπότε με χρήση πυλών OR συνδυάζουμε τους όρους που χρειάζονται για κάθε έξοδο.

Στην **Εικόνα 75** φαίνεται το κύκλωμα για τις εξόδους a, d, e. Η ίδια διαδικασία ακολουθείται για τις υπόλοιπες συναρτήσεις εξόδου.




**Εικόνα 75.** Το κύκλωμα για τις εξόδους a, d, e. Η ίδια διαδικασία ακολουθείται για τις υπόλοιπες συναρτήσεις εξόδου.

**11. 2-to-4 αποκωδικοποιητής με πύλες NOR. Σχεδιάστε το κύκλωμα ενός 2-to-4 line decoder με χρήση μόνο NOR Να συμπεριλάβετε και ένα enable στην είσοδο του αποκωδικοποιητή.**

Λύση.

Ο πίνακας αληθείας για το κύκλωμα φαίνεται στον **Πίνακα 15**.

| E | A | B | $D_0$ | $D_1$ | $D_2$ | $D_3$ |
|---|---|---|---|---|---|---|
| 0 | X | X | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 |

**Πίνακας 15.** Πίνακας αληθείας αποκωδικοποιητή 2-σε-4.

Οι συναρτήσεις των εξόδων προκύπτουν ως εξής:

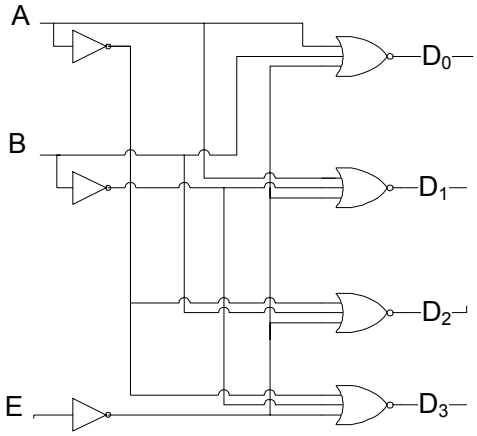
$D_0$= EA'B'=(E'+A+B)'

$D_1$= EA'B =(E'+A+B')'

$D_2$= EAB' =(E'+A'+B)'

$D_3$= EAB =(E'+A'+B')'

Το λογικό διάγραμμα του κυκλώματος φαίνεται στην **Εικόνα 76.**




**Εικόνα 76.** Αποκωδικοποιητής 2-to-8 με enable (E). Υλοποίηση με NOR.

**12. Υλοποίηση ημιαθροιστή σε VHDL.**

Μια υλοποίηση με VHDL οντότητας και αρχιτεκτονικής ημιαθροιστή καθώς και κώδικας ελέγχου παρουσιάζεται στη συνέχεια (**Κώδικας 1**).

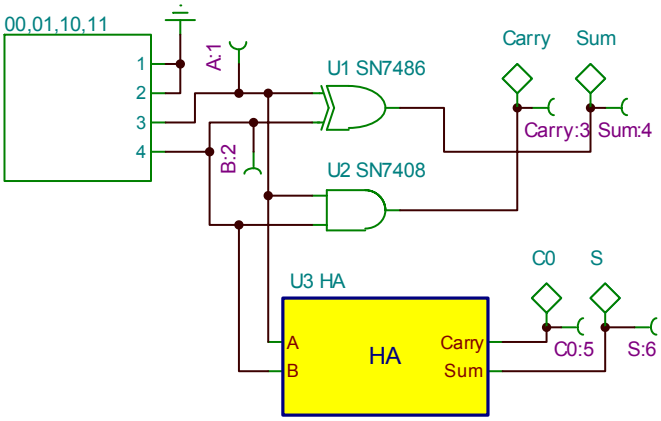
```
library ieee;
use ieee.std_logic_1164.all;

entity half_adder is
port (A ,B:in std_logic;
      S, C:out std_logic);
end half_adder;

architecture behavioral of half_adder is
begin
       S <= A xor B after 1 ns;
       C <= A and B after 1 ns;
end behavioral;
-------------------------------------------------
--TEST BENCH
library ieee;
use ieee.std_logic_1164.all;
entity half_adder_tb is
end half_adder_tb;

architecture behavior of half_adder_tb is
component half_adder is
port (A ,B : in std_logic;
S, C : out std_logic);
end component;
signal A, B, sum, Cout : std_logic;
begin
       mapping : half_adder port map (A, B, sum,
Cout);
       A <= '0';
       B <= '0';
       process
       begin
              A <= '0';
              B <= '0';
              wait for 5 ns;
              A <= '0';
              B <= '1';
              wait for 5 ns;
              A <= '1';
              B <= '0';
              wait for 5 ns;
              A <= '1';
              B <= '1';
              wait for 5 ns;
       end process;
end behavior;
```

**Κώδικας 1.** Υλοποίηση ημιαθροιστή σε VHDL.

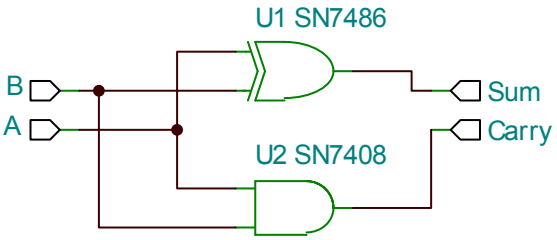
**13. a) Σχεδιάστε και προσομοιώστε στο TINA κύκλωμα ημιαθροιστή. b) Υλοποιήστε τον ημιαθροιστή σε μπλόκ διάγραμμα.**

Λύση.

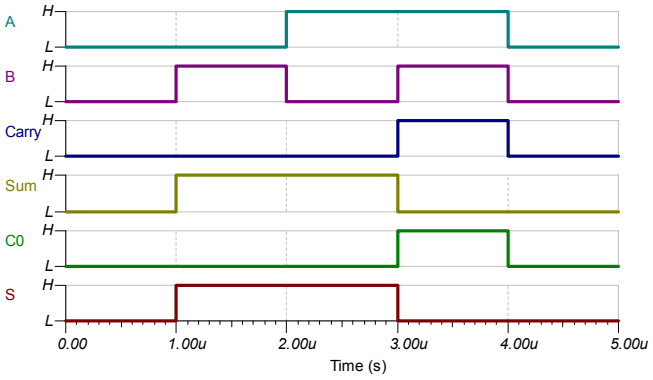
Τα βήματα της υλοποίησης φαίνονται επόμενες τρεις **Εικόνες 77-79**.




**Εικόνα 77.** Κύκλωμα ελέγχου λειτουργίας ημιαθροιστή και του μπλόκ διαγράμματός του.




**Εικόνα 78.** Περιεχόμενα του μπλόκ διαγράμματος ημιαθροιστή.

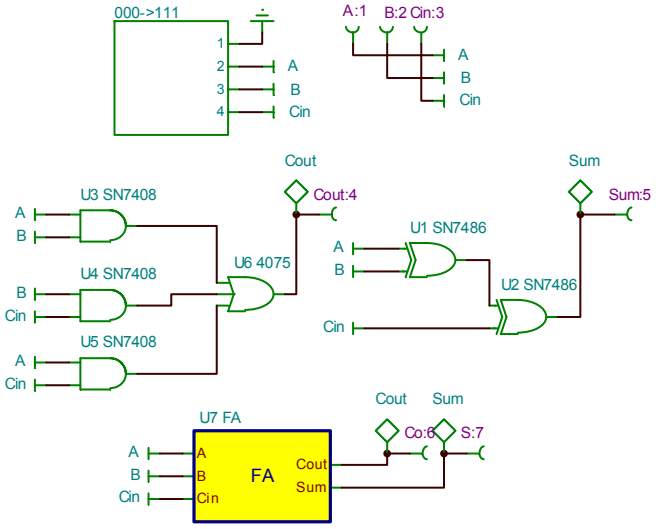



**Εικόνα 79.** Αποτελέσματα χρονικής ημιαθροιστή ανάλυσης για AB = 00, 01, 10, 11.

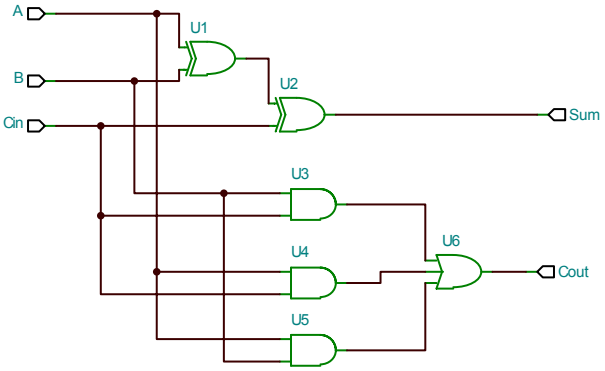
**14. a)Υλοποιήστε στο TINA το μπλοκ του πλήρη αθροιστή. b) Κάντε χρονική ανάλυση για επαλήθευση της λειτουργίας του.**

Λύση.

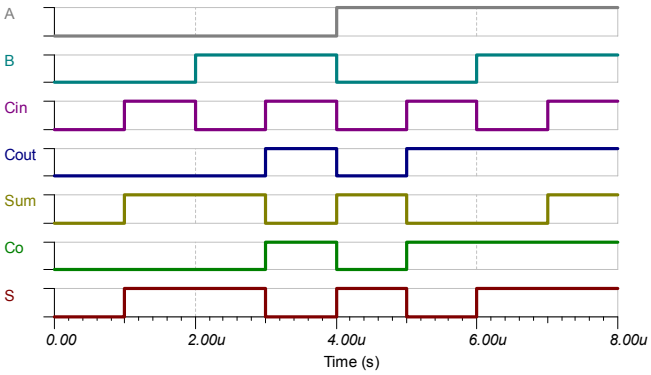
Στις επόμενες τρεις **Εικόνες 80-82** φαίνεται η διαδικασία της υλοποίησης και τα αποτελέσματα της προσομοίωσης.




**Εικόνα 80.** Κύκλωμα ελέγχου του μπλοκ του πλήρους αθροιστή.




**Εικόνα 81.** Περιεχόμενα του μπλοκ του πλήρους αθροιστή.




**Εικόνα 82.** Αποτελέσματα χρονικής ανάλυσης πλήρους αθροιστή.

**15. Υλοποιήστε πλήρη αθροιστή με VHDL.**

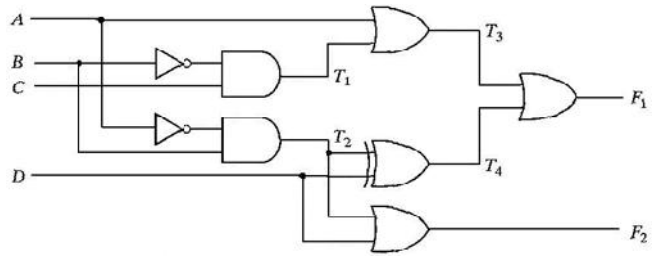
Υλοποίηση πλήρη αθροιστή με VHDL (**Κώδικας 2**).

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

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

architecture behavior of full_adder is
begin
cout <= ((a and b) or (cin and b) or (cin and a))
after 1 ns;
sum  <= ((cin xor a) xor b) after 1 ns;
end behavior;

-- TEST BENCH of full adder
library ieee;
use ieee.std_logic_1164.all;

entity full_adder_tb is
end full_adder_tb;

architecture behavior of full_adder_tb is
        component full_adder is
port(a, b, cin : in std_logic;
     cout, sum : out std_logic);
end component;

signal A, B, Cin, sum, Cout : std_logic;
begin
mapping : full_adder port map (A, B, Cin, Cout,
sum);
        A <= '0';
        B <= '0';
        Cin <= '0';
        Cout <= '0';
        sum <= '0';
        process
        begin
                wait for 5 ns;
                A <= '0';
                B <= '0';
                Cin <= '0';
                wait for 5 ns;
                A <= '0';
                B <= '0';
                Cin <= '1';
                wait for 5 ns;
                A <= '0';
                B <= '1';
                Cin <= '0';
                wait for 5 ns;
                A <= '0';
                B <= '1';
                Cin <= '1';
                wait for 5 ns;
                A <= '1';
                B <= '0';
                Cin <= '0';
                wait for 5 ns;
                A <= '1';
                B <= '0';
                Cin <= '1';
                wait for 5 ns;
                A <= '1';
                B <= '1';
                Cin <= '0';
                wait for 5 ns;
                A <= '1';
                B <= '1';
                Cin <= '1';
                wait for 5 ns;
        end process;
end behavior;
```

**Κώδικας 2.** Υλοποίηση πλήρη αθροιστή με VHDL.

**16. Θεωρήστε το κύκλωμα της εικόνας Εικόνα . a) Να εξάγετε την έκφραση Boole για τις εξόδους $T_1$ ως $T_4$. b) Προσδιορίστε τις εξόδους $F_1$ και $F_2$ συναρτήσει των τεσσάρων εισόδων. c) Διατυπώστε τον πίνακα αληθείας με τους 16 συνδυασμούς των μεταβλητών εισόδου, τις τιμές των $T_1$ ως $T_4$ και τις εξόδους $F_1$ και $F_2$. d) Σχεδιάστε τους ΧΚ των $F_1$ και $F_2$ και προδιορίστε τις απλοποιημένες μορφές τους.**

Λύση.




**Εικόνα 83.**

a) Οι εκφράσεις Boole για τις $T_1$ ως $T_4$ και $F_1$ και $F_2$ συναρτήσει των 4 εισόδων φαίνονται στη συνέχεια.

| | |
|---|---|
| $T_1 = B'C$ | $T_4 = T_2 \oplus D$ |
| $T_2 = A'B$ | $= A'BD' + (A'B)'D$ |
| $T_3 = A + T_1 = A + B'C$ | $= A'BD' + (A + B')D$ |
| | $= A'BD' + AD + B'D$ |

b) Οι εκφράσεις Boole για τις $F_1$ και $F_2$ συναρτήσει των 4 εισόδων φαίνονται στη δίπλα.

$$F_1 = T_3 + T_4$$
$$= A + AD + A'BD' + B'C + B'D$$
$$= A(1 + D) + A'BD' + B'C + B'D$$
$$= (A + A')(A + BD') + B'C + B'D$$
$$= A + BD' + B'C + B'D$$

$$F_2 = D + T_2 = D + A'B$$

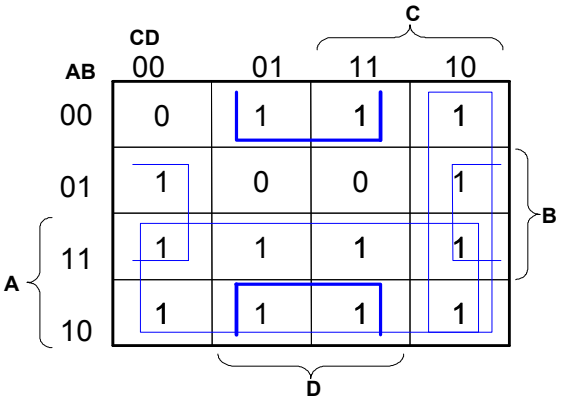
c) Ο πίνακας αληθείας με τους 16 συνδυασμούς των 4 εισόδων για τις $T_1$ ως $T_4$ και τις εξόδους $F_1$ και $F_2$:

| A | B | C | D | $T_1$ | $T_2$ | $T_3$ | $T_4$ | $F_1$ | $F_2$ |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |

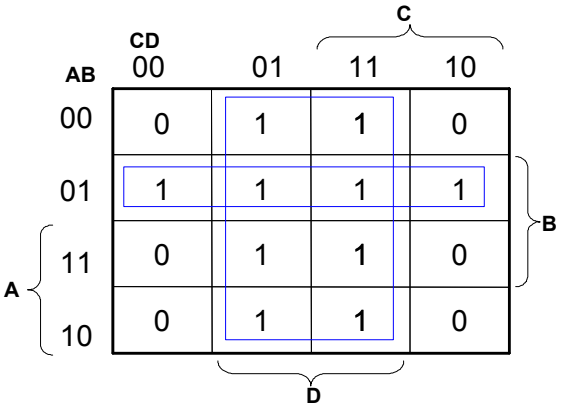
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |

**Πίνακας 16.**

d) Οι ΧΚ για τις εξόδους και οι απλοποιήσεις φαίνονται στη συνέχεια:




$$F_1 = A + BC + BD' + B'D$$




$$F_2 = D + A'B$$

**Εικόνα 84.**

**17. Σχεδιάστε ένα συνδυαστικό κύκλωμα με τρεις εισόδους x, y and z, και τρεις εξόδους A, B, and C. Όπου η δυαδική είσοδος είναι 0, 1, 2, ή 3, η δυαδική έξοδος να είναι μεγαλύτερη κατά 1 από την είσοδο. Όπου η δυαδική είσοδος είναι 4, 5, 6, ή 7, η δυαδική έξοδος είναι κατά ένα λιγότερη από την είσοδο.**

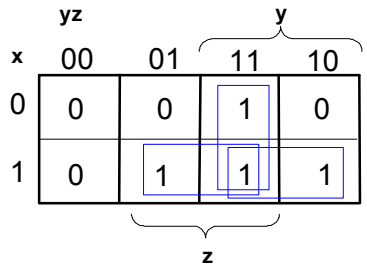
Λύση.

Για τη σχεδίαση του κυκλώματος, πρέπει πρώτα να ξεκινήσουμε από τον πίνακα αληθείας σύμφωνα με τα δεδομένα της εκφώνησης:

| x | y | z | A | B | C |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 |

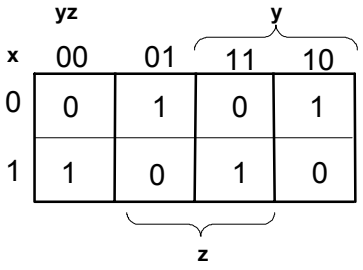
**Πίνακας 17.**

Για καθεμία από τις εξόδους σχεδιάζουμε το ΧΚ και κάνουμε τις απλοποιήσεις, όπως φαίνεται στη συνέχεια:



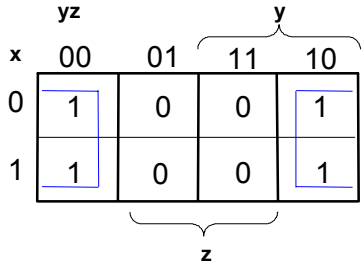

A➔

$$A = xz + xy + yz$$




B➔

$$B = x'y'z + x'yz' + xy'z' + xyz$$

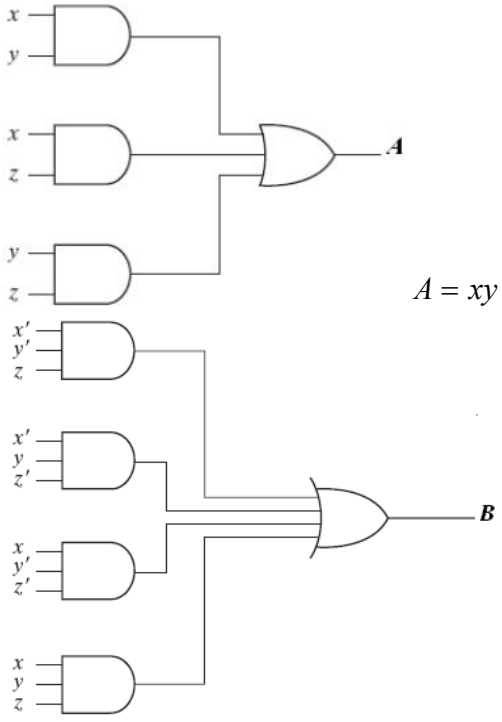



C➔

$$C = z'$$

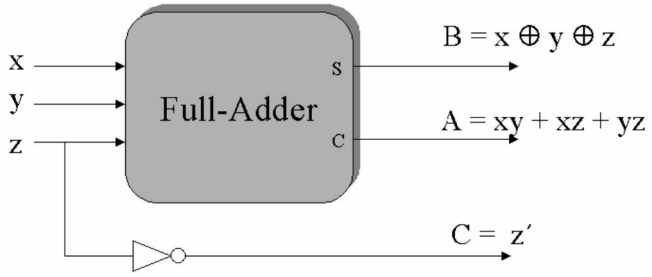
**Εικόνα 85.**

Στη συνέχεια σχεδιάζουμε το λογικό διάγραμμα με πύλες των εξόδων:




**Εικόνα 86.**

Ένας έμπειρος σχεδιαστής μπορεί να «δει» ότι το κύκλωμα υλοποιείται και ως εξής (**Εικόνα 87**):




**Εικόνα 87.**

## 18. Υλοποίηση Πλήρη Αθροιστή με Αποκωδικοποιητή και Δύο Πύλες OR.

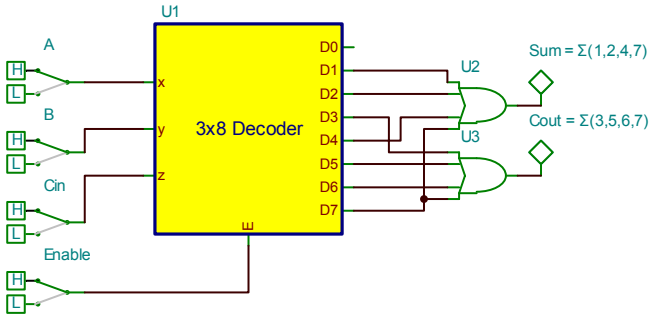
Λύση.

| i | A | B | Cin | Cout | Sum |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 |
| 2 | 0 | 1 | 0 | 0 | 1 |
| 3 | 0 | 1 | 1 | 1 | 0 |
| 4 | 1 | 0 | 0 | 0 | 1 |
| 5 | 1 | 0 | 1 | 1 | 0 |
| 6 | 1 | 1 | 0 | 1 | 0 |
| 7 | 1 | 1 | 1 | 1 | 1 |

**Πίνακας 18.** Από τον πίνακα αληθείας του πλήρη αθροιστή προκύπτει Sum(A,B,Cin) = Σ(1,2,4,7) και Cout(A,B,Cin) = Σ(3,5,6,7).

Από τον πίνακα αληθείας (**Πίνακας 18**) του πλήρη αθροιστή προκύπτει Sum(A,B,Cin) = Σ(1,2,4,7) και Cout(A,B,Cin) = Σ(3,5,6,7).

Η υλοποίηση πλήρους αθροιστή με χρήση αποκωδικοποιητή 3x8 φαίνεται στην **Εικόνα 88**.




**Εικόνα 88.** Υλοποίηση πλήρους αθροιστή με χρήση αποκωδικοποιητή 3x8.

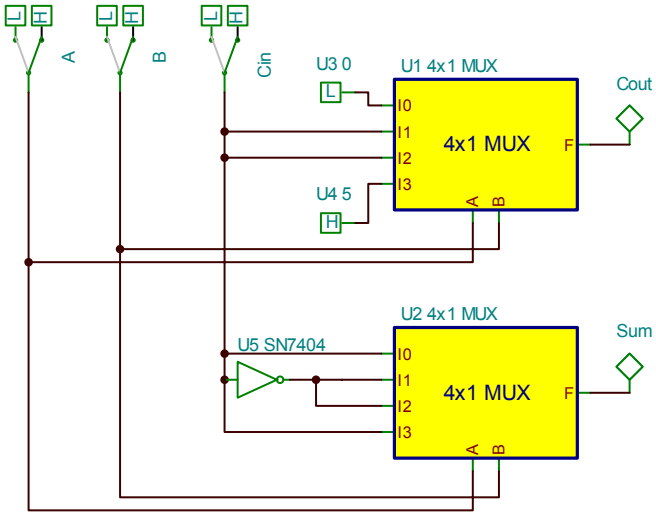
## 19. Υλοποίηση πλήρους αθροιστή με χρήση πολυπλεκτών

Ο πίνακας αληθείας πλήρους αθροιστή και προσδιορισμός εισόδων πολυπλέκτές για την παραγωγή του bit άθροισης και του bit κρατουμένου φαίνεται στον **Πίνακα 19**.

| | | | | | 1ος MUX.4x1 | | 2ος MUX.4x1 |
|---|---|---|---|---|---|---|---|
| i | A | B | Cin | Cout | F = Cout | Sum | F=Sum |
| 0 | 0 | 0 | 0 | 0 | | 0 | |
| 1 | 0 | 0 | 1 | 0 | I0=0 | 1 | I0=Cin |
| 2 | 0 | 1 | 0 | 0 | | 1 | |
| 3 | 0 | 1 | 1 | 1 | I1=Cin | 0 | I1=Cin' |
| 4 | 1 | 0 | 0 | 0 | | 1 | |
| 5 | 1 | 0 | 1 | 1 | I2=Cin | 0 | I2=Cin' |
| 6 | 1 | 1 | 0 | 1 | | 0 | |
| 7 | 1 | 1 | 1 | 1 | I3=1 | 1 | I3=Cin |

**Πίνακας 19.** Πίνακας αληθείας πλήρους αθροιστή και προσδιορισμός εισόδων πολυπλέκτες για την παραγωγή του bit άθροισης και του bit κρατουμένου.

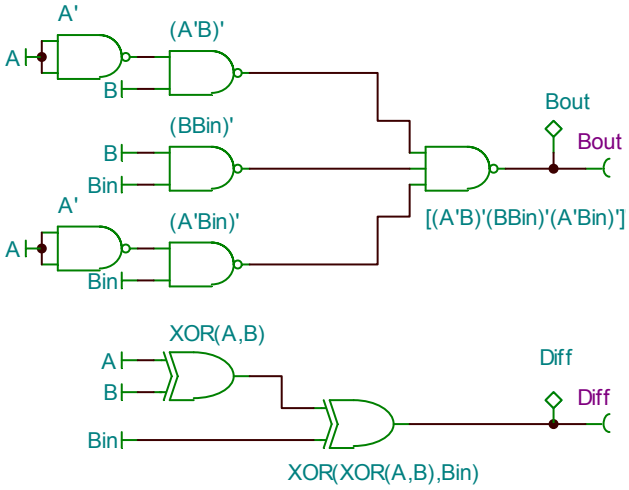
Η υλοποίηση πλήρους αθροιστή με χρήση δύο πολυπλεκτών 4x1 φαίνεται στην **Εικόνα 89**.



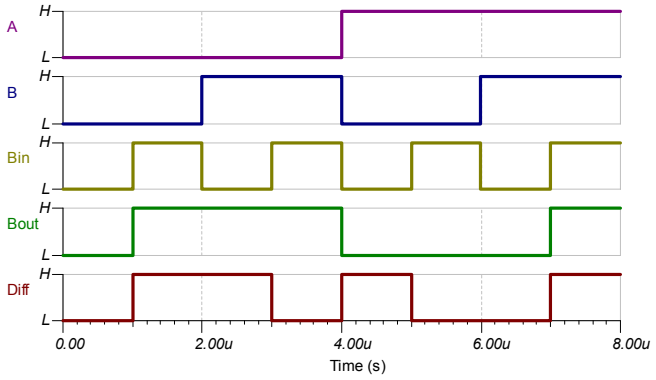

**Εικόνα 89.** Υλοποίηση πλήρους αθροιστή με χρήση δύο πολυπλεκτών 4x1.

## 20. Υλοποίηση πλήρους αφαιρέτη με πύλες XOR και NAND.

Λύση.
Βλέπε **Εικόνες 90 και 91**.




**Εικόνα 90.** Πλήρης αφαιρέτης με πύλες NAND και XOR




**Εικόνα 91.** Προσομοίωση πλήρους αφαιρέτη κατασκευασμένου με πύλες NAND και XOR.

## 21. (VHDL) Υλοποίηση συγκριτή 2bit με χρήση std_logic_vector (προσομοίωση στο sonata).

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

entity comparator_2bit is
port(
a : in STD_LOGIC_VECTOR(1 downto 0);
b : in STD_LOGIC_VECTOR(1 downto 0);
equal : out STD_LOGIC;
greater : out STD_LOGIC;
lower : out STD_LOGIC
);
end comparator_2bit;

architecture comparator_2bit_arc
of comparator_2bit is
begin

    equal <= '1' when (a=b) else
'0';

    greater <= '1' when (a<b) else
'0';

    lower <= '1' when (a>b) else
'0';
end comparator_2bit_arc;
```

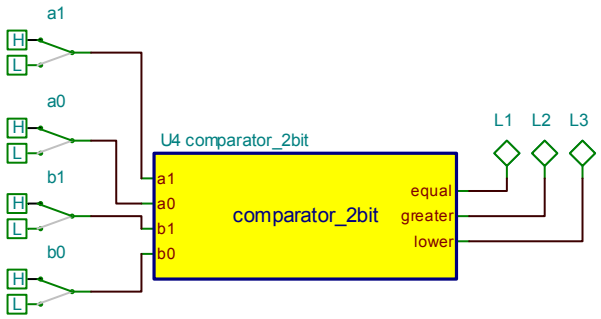
**Κώδικας.** Οντότητα συγκριτή 2bit.

## 22. (VHDL) Υλοποίηση συγκριτή 2bit κατάλληλη για χρήση στο TINA.

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

entity comparator_2bit is
     port(
          a1 : in STD_LOGIC;
          a0 : in STD_LOGIC;
          b1 : in STD_LOGIC;
          b0 : in STD_LOGIC;
          equal : out STD_LOGIC;
          greater : out STD_LOGIC;
          lower : out STD_LOGIC
          );
end comparator_2bit;

architecture    comparator_2bit_arc    of
comparator_2bit is
begin

    equal   <=   '1'   when   ((a1=b1)   and
(a0=b0)) else '0';

    greater   <=   '1'   when   ((a1<b1)   or
((a1=b1) and (a0<b0)) ) else  '0';

    lower   <=   '1'   when   ((a1>b1)   or
((a1=b1) and (a0>b0)) ) else  '0';
```

```
end comparator_2bit_arc;
```

**Κώδικας.** Οντότητα συγκριτή 2bit.




**Εικόνα 92.** Μοντέλο ελέγχου στο TINA για τον συγκριτή 2bit.

### Τύποι δεδομένων στη VHDL

(Τα επόμενα είναι από το βιβλίο του Pedroni, Circuit Design with VHDL του 2005).

### Προδιαγεγραμμένο σύνολο από τύπους δεδομένων

Η VHDL περιέχει ένα προδιαγεγραμμένο σύνολο από τύπους δεδομένων, οι οποίοι ορίζονται στα επόμενα πακέτα/βιβλιοθήκες της γλώσσας:

- Package standard ή library std
- Package std_logic_1164 ή library ieee
- Package std_logic_arith ή library ieee
- Packages std_logic_signed και std_logic_unsigned ή library ieee.

Στο πακέτο standard ορίζονται οι τύποι BIT, BOOLEAN, INTEGER και REAL.

Στο πακέτο std_logic_1164 ορίζονται οι τύποι STD_LOGIC και STD_ULOGIC.

Στο πακέτο std_logic_arith ορίζονται οι τύποι SIGNED και UNSIGNED καθώς και διάφορες συναρτήσεις μετατροπής όπως οι:

- con_integer(p),
- conv_unsigned(p,b),
- conv_signed(p, b),
- conv_std_logic_vector(p,b)

Στα πακέτα std_logic_signed και std_logic_unsigned περιέχονται συναρτήσεις που επιτρέπουν λειτουργίες σε δεδομένα τύπου STD_LOGIC_VECTOR θεωρώντας αντίστοιχα τα δεδομένα τύπου SIGNED ή UNSINGED αντίστοιχα.

### Ο τύπος BIT

Ο τύπος BIT (και αντίστοιχα ο BIT_VECTOR) ορίζει δεδομένα δύο τιμών λογικής (0 και 1). Σε μια μεταβλητή αυτού του τύπου μπορούμε να της δώσουμε δεδομένα με χρήση του τελεστή ”<=”.

### Άσκηση.

Εξηγείστε το είδος των δεδομένων που ορίζουν οι επόμενες δηλώσεις:

```
Signal x : bit;
Signal y : bit_vector(3 downto 0);
Signal w : bit_vector(0 to 7);
x<='1';
y<="0111";
w<="01110001";
```

### Ο τύπος STD_LOGIC

Ο τύπος STD_LOGIC (αντίστοιχα ο STD_LOGIC_VECTOR) καταλαβαίνει 8 τιμές λογικής οι οποίες καθορίζονται από το πρότυπο IEEE 1164. Συγκεκριμένα τις:

- ‘X’ : επιβαλλόμενος άγνωστος (συνθέσιμος άγνωστος)
- ‘0’ : επιβαλλόμενο χαμηλό (συνθέσιμη λογική ‘1’)
- ‘1’: επιβαλλόμενο υψηλό (συνθέσιμη λογική ‘0’)
- ‘Z’ : υψηλή εμπέδηση
- ‘W’ : ασθενικός άγνωστος
- ‘L‘ : ασθενές χαμηλό
- ‘H’ : ασθενές υψηλό
- ‘-‘ : αδιάφορος όρος

Ο τελεστές “:=” χρησιμοποιείται σε αυτή την περίπτωση για την απόδοση αρχικής τιμής.

Τα περισσότερα από τα επίπεδα λογικής του τύπου std_logic απευθύνονται στην προσομοίωση. Ωστόσο, τα ‘0’, ‘1’, και ‘Z’ είναι συνθέσιμα χωρίς περιορισμούς. Σε σχέση με τις ‘ασθενείς’ τιμές, σε κόμβους που έχουν πολλαπλή οδήγηση από σήματα, υπερισχύου οι ‘επιβαλλόμενες’ τιμές. Πράγματι, αν οποιαδήποτε δύο σήματα τύπου std_logic συνδέονται στον ίδιο κόμβο, τότε συγκρουόμενα επίπεδα λογικής θα διακρίνονται σύμφωνα με τον επόμενο πίνακα:

| | X | 0 | 1 | Z | W | L | H | - |
|---|---|---|---|---|---|---|---|---|
| X | X | X | X | X | X | X | X | X |
| 0 | X | 0 | X | 0 | 0 | 0 | 0 | X |
| 1 | X | X | 1 | 1 | 1 | 1 | 1 | X |
| Z | X | 0 | Z | Z | W | L | H | X |
| W | X | 0 | W | W | W | W | W | X |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| L | X | 0 | L | L | W | L | W | X |
| H | X | 0 | H | H | W | W | H | X |
| - | X | X | X | X | X | X | X | X |

## Άσκηση.

Εξηγείστε το είδος των δεδομένων που ορίζουν οι επόμενες δηλώσεις:

```
SIGNAL x: STD_LOGIC;
SIGNAL y: STD_LOGIC_VECTOR (3 DOWNTO 0) := "0001";
```

Το σύστημα STD_LOGIC είναι υποσύνολο του STD_ULOGIC. Το επιπλέον U στο όνομα του τύπου σημαίνει "unresolved". Δηλαδή, σε αντίθεση με το λογική του STD_LOGIC, συγκρουόμενα επίπεδα λογικής δεν επιλύονται αυτόματα, οπότε τα καλώδια εξόδου δεν πρέπει να συνδέονται μαζί. Αν δύο καλώδια εξόδου δεν συνδέονται μεταξύ τους, τότε αυτό το σύστημα λογικής μπορεί να χρησιμοποιηθεί για την ανίχνευση σφαλμάτων σχεδιαστικής λογικής.

## Ο τύπος BOOLEAN

Παίρνει τις τιμές True ή False.

## Ο τύπος INTEGER

Αντιστοιχεί στους 32-bit ακεραίους μεταξύ των τιμών – 2,147,483,647 μέχρι + 2,147,483,647.

## Ο τύπος NATURAL

Αντιστοιχεί στους μη αρνητικούς ακεραίους μεταξύ των τιμών 0 μέχρι + 2,147,483,647.

## Ο τύπος REAL

Αντιστοιχεί στους πραγματικούς αριθμούς μεταξύ -1.0E38 μέχρι + 1.0E38. Δεν είναι συνθέσιμος τύπος.

## Φυσικές σταθερές

Χρησιμοποιούνται για την περιγραφή φυσικών ποσοτήτων όπως ο χρόνος, η τάση κ.α. Είναι χρήσιμες στις προσομοιώσεις. Δεν είναι συνθέσιμος τύπος.

## Σταθερές χαρακτήρων

Πρόκειται για μεμονωμένους χαρακτήρες ASCII ή σειρές (strings) τέτοιων χαρακτήρων. Δεν είναι συνθέσιμος τύπος.

## Ο τύπος SIGNED και UNSIGNED

Οι τύπος αυτοί ορίζονται στο πακέτο std_logic_arith της βιβλιοθήκης ieee. Εμφανίζονται ως STD_LOGIC_VECTOR αλλά δέχονται αριθμητικές πράξεις, τυπικές για τύπους δεδομένων INTEGER.

## Παραδείγματα τύπων δεδομένων

```
x0 <= '0';
-- bit, std_logic, or
-- std_ulogic value '0'

x1 <= "00011111";
-- bit_vector, std_logic_vector,
-- std_ulogic_vector, signed,
-- or unsigned

x2 <= "0001_1111";
-- underscore allowed to
-- ease visualization

x3 <= "101111"
-- binary representation
-- of decimal 47

x4 <= B"101111"
-- binary representation
-- of decimal 47

x5 <= O"57"
-- octal representation
-- of decimal 47

x6 <= X"2F"
-- hexadecimal representation
-- of decimal 47

n <= 1200;
-- integer

m <= 1_200;
-- integer, underscore allowed

IF ready THEN...
-- Boolean, executed if ready=TRUE

y <= 1.2E-5;
-- real, not synthesizable

q <= d after 10 ns;
```

```
-- physical, not synthesizable
```

## Παραδείγματα επιτρεπόμενων και μη πράξεων μεταξύ διαφορετικών τύπων δεδομένων

```
SIGNAL a: BIT;

SIGNAL b: BIT_VECTOR(7 DOWNTO 0);

SIGNAL c: STD_LOGIC;

SIGNAL d: STD_LOGIC_VECTOR(7 DOWNTO
0);
SIGNAL e: INTEGER RANGE 0 TO 255;

...

a <= b(5);
-- legal (same scalar type: BIT)

b(0) <= a;
-- legal (same scalar type: BIT)

c <= d(5);
-- legal (same scalar type:
-- STD_LOGIC)

d(0) <= c;
-- legal (same scalar type:
-- STD_LOGIC)

a <= c;
-- illegal (type mismatch:
-- BIT x STD_LOGIC)

b <= d;
-- illegal (type mismatch:
-- BIT_VECTOR x STD_LOGIC_VECTOR)

e <= b;
-- illegal (type mismatch:
-- INTEGER x BIT_VECTOR)

e <= d;
-- illegal (type mismatch: INTEGER x
-- STD_LOGIC_VECTOR)
```

## Τύποι δεδομένων ορισμένοι από το χρήστη

Η VHDL μας επιτρέπει να ορίσουμε τους δικούς μας τύπους δεδομένων. Στη συνέχεια παρουσιάζουμε δύο τύπους δεδομένων οριζόμενους από το χρήστης: τους integer και τους enumerated.

## Οριζόμενοι από το χρήστη τύποι integer

```
TYPE integer IS RANGE -2147483647 TO
+2147483647;
-- This is indeed the
-- pre-defined type INTEGER.

TYPE natural IS RANGE 0 TO
+2147483647;
-- This is indeed the pre-defined
-- type NATURAL.

TYPE my_integer IS RANGE -32 TO 32;
-- A user-defined subset
-- of integers.

TYPE student_grade IS RANGE 0 TO
100;
-- A user-defined subset of
-- integers or naturals.
```

## Οριζόμενοι από το χρήστη τύποι enumerated

```
TYPE bit IS ('0', '1');
-- This is indeed the
-- pre-defined type BIT

TYPE my_logic IS ('0', '1', 'Z');
-- A user-defined subset
-- of std_logic.

TYPE bit_vector IS ARRAY (NATURAL
RANGE <>) OF BIT;
-- This is indeed the
-- pre-defined type BIT_VECTOR.
-- RANGE <> is used to indicate that
-- the range is unconstrained.
-- NATURAL RANGE <>, on the other
-- hand, indicates that the only
-- restriction is that the range
-- must fall within the NATURAL
-- range.

TYPE state IS (idle, forward,
backward, stop);
-- An enumerated data type, typical
-- of finite state machines.

TYPE color IS (red, green, blue,
white);
-- Another enumerated data type.
```

Η κωδικοποίηση των τύπων enumerated γίνεται ακολουθιακά και αυτόματα (εκτός και οριστεί διαφορετικά από το χρήστη). Π.χ. για τον τύπο color στο προηγούμενο παράδειγμα, χρειαζόμαστε δύο bits για την κωδικοποίηση των καταστάσεων (επειδή είναι 4), τα 00 για την

κατάσταση red, 01 για την green, 10 για την blue και 11 για την white.

## Υποτύποι (subtypes)

Ένας υποτύπος είναι ένας τύπος δεδομένων με περιορισμού. Ο κύριος λόγος χρήσης των υποτύπων από το να ορίσουμε έναν νέο τύπο, είναι διότι παρόλο που οι πράξεις μεταξύ δεδομένων διαφορετικού τύπου, δεν επιτρέπονται, επιτρέπονται μεταξύ του υποτύπου και του κυρίως τύπου.

Παραδείγματα:

```
SUBTYPE natural IS INTEGER RANGE 0
TO INTEGER'HIGH;
-- As expected, NATURAL is a subtype
--(subset) of INTEGER.

SUBTYPE my_logic IS STD_LOGIC RANGE
'0' TO 'Z';
-- Recall that STD_LOGIC=
--('X','0','1','Z','W','L','H','-').
-- Therefore, my_logic=
--('0','1','Z').

SUBTYPE my_color IS color RANGE red
TO blue;
-- Since color=(red, green,
-- blue, white), then
-- my_color=(red, green, blue).

SUBTYPE small_integer IS INTEGER
RANGE -32 TO 32;
-- A subtype of INTEGER.
```

## Παραδείγματα επιτρεπόμενων και μη πράξεων μεταξύ τύπων και υποτύπων

```
SUBTYPE my_logic IS STD_LOGIC RANGE
'0' TO '1';

SIGNAL a: BIT;

SIGNAL b: STD_LOGIC;

SIGNAL c: my_logic;

...

b <= a;
-- illegal (type mismatch:
-- BIT versus STD_LOGIC)

b <= c;
-- legal (same "base" type:
-- STD_LOGIC)
```

## Πίνακες (arrays)

Οι πίνακες είναι συλλογές αντικειμένων του ίδιου τύπου. Μπορεί να είναι μονοδιάστατοι, δισδιάστατοι, τρισδιάστατοι ή μεγαλύτερης διάστασης αλλά τότε δεν είναι συνθέσιμοι. Οι εξορισμού συνθέσιμοι πίνακες στη VHDL είναι ο βαθμωτός (single bit) και ο μονοδιάστατος (vector):

```
Scalars:
BIT,
STD_LOGIC,
STD_ULOGIC,
BOOLEAN.

Vectors:
BIT_VECTOR,
STD_LOGIC_VECTOR,
STD_ULOGIC_VECTOR,
INTEGER,
SIGNED,
UNSIGNED.
```

Δεν υπάρχουν εξορισμού δισδιάστατοι πίνακες. Αν είναι απαραίτητοι θα πρέπει να τους ορίσει ο χρήστης. Αυτό μπορεί να γίνει με τον επόμενο τρόπο:

```
TYPE type_name IS
ARRAY (specification) OF data_type;

SIGNAL signal_name:
type_name [:= initial_value];
```

Στην παραπάνω σύνταξη χρησιμοποιήσαμε το SIGNAL. Ωστόσο μπορούμε να χρησιμοποιήσουμε αντίστοιχα CONSTANT ή VARIABLE. Η δήλωση της αρχικής τιμής είναι κατ' επιλογή (χρησιμοποιείται μόνο σε περίπτωση προσομοίωσης).

## Παραδείγματα δισδιάστατων πινάκων

```
TYPE row IS ARRAY (7 DOWNTO 0) OF
STD_LOGIC;
-- 1D array

TYPE matrix IS ARRAY (0 TO 3) OF
row;
-- 1Dx1D array

SIGNAL x: matrix;
-- 1Dx1D signal
```

Ένας άλλος τρόπος για τη δημιουργία του παραπάνω πίνακα:

```
TYPE matrix IS ARRAY (0 TO 3)
OF STD_LOGIC_VECTOR(7 DOWNTO 0);
```

Υλοποίηση πραγματικά δισδιάστατου πίνακα (χωρίς τη χρήση vectors):

```
TYPE matrix2D IS ARRAY (0 TO 3, 7
DOWNTO 0) OF STD_LOGIC;
-- 2D array
```

## Παραδείγματα αρχικοποίησης πινάκων

```
... :="0001";
-- for 1D array

... :=('0','0','0','1')
-- for 1D array

... :=(
('0','1','1','1'),
('1','1','1','0')
);
-- for 1Dx1D or
-- 2D array
```

## Παραδείγματα επιτρεπόμενων και μη αναθέσεων σε πίνακες

```
TYPE row IS ARRAY (7 DOWNTO 0) OF
STD_LOGIC;
-- 1D array

TYPE array1 IS ARRAY (0 TO 3) OF
row;
-- 1Dx1D array

TYPE array2 IS ARRAY (0 TO 3) OF
STD_LOGIC_VECTOR(7 DOWNTO 0);
-- 1Dx1D

TYPE array3 IS ARRAY (0 TO 3, 7
DOWNTO 0) OF STD_LOGIC;
-- 2D array

SIGNAL x: row;
SIGNAL y: array1;
SIGNAL v: array2;
SIGNAL w: array3;

-- Legal scalar assignments:
-- The scalar (single bit)
-- assignments below are all legal,
-- because the "base" (scalar) type
-- is STD_LOGIC for all signals
-- (x,y,v,w).

x(0) <= y(1)(2);
-- notice two pairs of parenthesis
-- (y is 1Dx1D)

x(1) <= v(2)(3);
-- two pairs of parenthesis
-- (v is 1Dx1D)

x(2) <= w(2,1);
-- a single pair of parenthesis
--(w is 2D)

y(1)(1) <= x(6);

y(2)(0) <= v(0)(0);

y(0)(0) <= w(3,3);

w(1,1) <= x(7);

w(3,0) <= v(0)(3);

-- Vector assignments:

x <= y(0);
-- legal (same data types: ROW)

x <= v(1);
-- illegal (type mismatch: ROW x
-- STD_LOGIC_VECTOR)

x <= w(2);
-- illegal (w must have 2D index)

x <= w(2, 2 DOWNTO 0);
-- illegal (type mismatch: ROW x
-- STD_LOGIC)

v(0) <= w(2, 2 DOWNTO 0);
-- illegal (mismatch:
-- STD_LOGIC_VECTOR
-- x STD_LOGIC)

v(0) <= w(2);
-- illegal (w must have 2D index)

y(1) <= v(3);
-- illegal (type mismatch: ROW x
-- STD_LOGIC_VECTOR)

y(1)(7 DOWNTO 3) <= x(4 DOWNTO 0);
-- legal (same type,
-- same size)

v(1)(7 DOWNTO 3) <= v(2)(4 DOWNTO
0);
-- legal (same type,
-- same size)

w(1, 5 DOWNTO 1) <= v(2)(4 DOWNTO
0);
-- illegal (type mismatch)
```

## Πίνακας θυρών (port array)

Στη VHDL δεν υπάρχουν προκαθορισμένοι τύπου δεδομένων με περισσότερες από μία διαστάσεις. Ωστόσο, στον προσδιορισμό της εισόδου ή των εισόδων (PORTS) ενός κυκλώματος (που γίνεται στην

ENTITY), πιθανώς θα χρειάζεται να προσδιορίσουμε τις θύρες ως πίνακες διανυσμάτων. Επειδή οι δηλώσεις TYPE δεν επιτρέπονται σε μια ENTITY, η λύση είναι να δηλώσουμε τύπους δεδομένων προσδιορισμένων από το χρήστη σε ένα PACKAGE, το οποίο θα είναι στη συνέχεια φανερό σε ολόκληρο το σχέδιο (άρα και στην ENTITY). Στη συνέχεια παρουσιάζουμε ένα παράδειγμα:

```
------- Package:
LIBRARY ieee;
USE ieee.std_logic_1164.all;
----------------------------
PACKAGE my_data_types IS
TYPE vector_array IS ARRAY (NATURAL
RANGE <>) OF
STD_LOGIC_VECTOR(7 DOWNTO 0);
END my_data_types;
-----------------------------------
------- Main code: -----------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.my_data_types.all;
-- user-defined package
----------------------------
ENTITY mux IS
PORT (inp: IN VECTOR_ARRAY (0 TO 3);
... );
END mux;
... ;
-----------------------------------
```

Όπως φαίνεται στον κώδικα, ορίσαμε έναν τύπο δεδομένων με το όνομα vector_array που περιέχει απροσδιόριστο πλήθος διανυσμάτων μεγέθους οκτώ bits το καθένα. (Η δήλωση NATURAL RANGE <> σημαίνει ότι η εμβέλεια δεν είναι προσδιορισμένη, αλλά με την προϋπόθεση ότι πρέπει να είναι εντός της περιοχής των NATURAL). Ο τύπος δεδομένων σώζεται ως PACKAGE και ονομάζεται my_data_types. Στη συνέχεια χρησιμοποιείται στην ENTITY για να προσδιορίσει μια PORT που λέγεται inp. Σημειώνεται η χρήση μια επιπλέον κλήσης USE προς το πακέτο my_data_types που το κάνει ορατό σε όλο το σχέδιο.

Μια άλλη δυνατότητα για το PACKAGE θα φανεί παρακάτω, στην οποία γίνεται χρήση της CONSTANT:

```
------- Package: -------------------
------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
----------------------------
PACKAGE my_data_types IS
CONSTANT b: INTEGER := 7;
TYPE vector_array IS ARRAY (NATURAL
RANGE <>) OF
STD_LOGIC_VECTOR(b DOWNTO 0);
END my_data_types;
-----------------------------------
```

## Καταγραφές (records)

Οι καταγραφές είναι αντίστοιχες με τους πίνακες με τη διαφορά ότι μπορούν να περιέχουν αντικείμενα διαφορετικών τύπων. Παράδειγμα:

```
TYPE birthday IS RECORD
day: INTEGER RANGE 1 TO 31;
month: month_name;
END RECORD;
```

## Προσημασμένοι και μη τύποι δεδομένων

Οι τύποι SIGNED και UNSIGNED ορίζονται στο πακέτο std_logic_arith της βιβλιοθήκης ieee. Η σύνταξή τους φαίνεται στη συνέχεια:

```
SIGNAL x: SIGNED (7 DOWNTO 0);
SIGNAL y: UNSIGNED (0 TO 3);
```

Παρατηρούμε ότι η σύνταξή τους είναι παρόμοια με αυτή του STD_LOGIC_VECTOR και όχι σαν ενός INTEGER όπως θα περιμέναμε.

Μια UNSINGED τιμή είναι ένας αριθμός ποτέ αρνητικός. Για παράδειγμα ο '0101' παριστάνει τον ακέραιο 5 ενώ ο '1101' τον 13. Αν χρησιμοποιήσουμε τον τύπο SIGNED τότε η τιμή μπορεί να είναι θετική ή αρνητική (στο συμπλήρωμα ως προς δύο). Άρα το '0101' θα παριστάνει τον ακέραιο 5, αλλά το '1101' θα παριστάνει το -3.

Για να χρησιμοποιήσουμε τύπους SIGNED ή UNSIGNED θα πρέπει να κάνουμε χρήση του πακέτου std_logic_arith της βιβλιοθήκης ieee.

Παρόλη τη σύνταξή τους, οι τύποι SIGNED και UNSIGNED προορίζονται κυρίως για αριθμητικές πράξεις, δηλαδή σε αντίθεση με τον STD_LOGIC_VECTOR, επιδέχονται αριθμητικές πράξεις. Αντίθετα, δεν επιδέχονται πράξεις λογικής. Ως προς τις σχεσιακές (συγκριτικές) πράξεις, δεν υπάρχουν περιορισμοί για τους τύπους αυτούς.

## Παράδειγμα σε νόμιμες και μη πράξεις σε signed/unsigned τύπου δεδομένων

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
-- extra package necessary
...
SIGNAL a: IN SIGNED (7 DOWNTO 0);
SIGNAL b: IN SIGNED (7 DOWNTO 0);
```

```
SIGNAL x: OUT SIGNED (7 DOWNTO 0);
...
v <= a + b;
-- legal (arithmetic operation OK)

w <= a AND b;
-- illegal (logical operation
-- not OK)
```

**Παράδειγμα νόμιμων και μη πράξεων σε τύπους std-logic_vector**

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-- no extra package required
...
SIGNAL a: IN
STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL b: IN
STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL x: OUT
STD_LOGIC_VECTOR (7 DOWNTO 0);
...
v <= a + b;
-- illegal (arithmetic operation
-- not OK)

w <= a AND b;
-- legal (logical operation OK)
```

**Τρόπος χρήσης std_logic_vector σε αριθμητικές πράξεις**

Η βιβλιοθήκη ieee παρέχει τα πακέτα std_logic_signed και std_logic_unsigned που επιτρέπουν πράξεις με δεδομένα τύπου std_logic_vector, σα να ήταν τύπου SIGNED ή UNSIGNED. Π.χ.:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
-- extra package included
...
SIGNAL a:
IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL b:
IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL x:
OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
...
v <= a + b;
-- legal (arithmetic operation OK),
-- unsigned

w <= a AND b;
-- legal (logical operation OK)
```

**Μετατροπή δεδομένων**

Η VHDL δεν επιτρέπει άμεσες πράξεις (αριθμητικής, λογικής κτλ.) μεταξύ δεδομένων διαφορετικών τύπων. Συνεπώς είναι συχνά απαραίτητο να μετατρέπουμε τα δεδομένα από έναν τύπο σε άλλο. Αυτό μπορεί να γίνει βασικά με δύο τρόπους: ο ένας είναι να γράψουμε ειδικό VHDL κώδικα για τη μετατροπή και ο άλλος να καλέσουμε μια FUNCTION από ένα προ-ορισμένο PACKAGE κατάλληλη για τη μετατροπή αυτή.

Αν τα δεδομένα είναι στενά συνδεδεμένα, (δηλαδή και οι δύο τελεστέοι έχουν τον ίδιο τύπο βάσης, παρότι έχουν δηλωθεί ως διαφορετικών τύπων), τότε το std_logic_1164 της ieee παρέχει τις απαραίτητες συναρτήσεις μετατροπής. Π.χ.:

```
TYPE long IS
INTEGER RANGE -100 TO 100;
TYPE short IS INTEGER RANGE -10 TO
10;
SIGNAL x : short;
SIGNAL y : long;
...
y <= 2*x + 5;
-- error, type mismatch
y <= long(2*x + 5);
-- OK, result converted
-- into type long
```

Οι συναρτήσεις μετατροπής δεδομένων του πακέτου std_logic_arith της ieee είναι οι εξής:

```
conv_integer(p) :
Converts a parameter p of type
INTEGER, UNSIGNED, SIGNED, or
STD_ULOGIC to an INTEGER value.
Notice that STD_LOGIC_VECTOR is not
included.

conv_unsigned(p, b):
Converts a parameter p of type
INTEGER, UNSIGNED, SIGNED, or
STD_ULOGIC to an UNSIGNED value with
size b bits.

conv_signed(p, b):
Converts a parameter p of type
INTEGER, UNSIGNED, SIGNED, or
STD_ULOGIC to a SIGNED value with
size b bits.

conv_std_logic_vector(p, b):
Converts a parameter p of type
INTEGER, UNSIGNED, SIGNED, or
STD_LOGIC to a STD_LOGIC_VECTOR
value with size b bits.
```

Παράδειγμα:

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

```
USE ieee.std_logic_arith.all;
...
SIGNAL a: IN
UNSIGNED (7 DOWNTO 0);
SIGNAL b: IN
UNSIGNED (7 DOWNTO 0);
SIGNAL y: OUT
STD_LOGIC_VECTOR (7 DOWNTO 0);
...

y <= CONV_STD_LOGIC_VECTOR ((a+b),
8);
-- Legal operation: a+b is
-- converted from UNSIGNED to an
-- 8-bit STD_LOGIC_VECTOR value,
-- then assigned to y.
```

Η άλλη εναλλακτική για την οποία έγινε ήδη αναφορά είναι μέσω της χρήσης των std_logic_signed και std_logic_unsigned πακέτων της ieee. Τέτοια πακέτα επιτρέπουν πράξεις με STD_LOGIC_VECTOR δεδομένα σα να ήταν τύπου SIGNED ή UNSIGNED αντίστοιχα.

**Συνοπτικός πίνακας συνθέσιμων τύπων δεδομένων**

| Τύπος δεδομένων | Συνθέσιμες τιμές |
|---|---|
| BIT, BIT_VECTOR | 0', '1' |
| STD_LOGIC, STD_LOGIC_VECTOR | 'X', '0', '1', 'Z' (resolved) |
| STD_ULOGIC, STD_ULOGIC_VECTOR | 'X', '0', '1', 'Z' (unresolved) |
| BOOLEAN | True, False |
| NATURAL | From 0 to +2, 147, 483, 647 |
| INTEGER | From -2,147,483,647 to +2,147,483,647 |
| SIGNED | From -2,147,483,647 to +2,147,483,647 |
| UNSIGNED | From 0 to +2,147,483,647 |
| User-defined integer type | Subset of INTEGER |
| User-defined enumerated type | Collection enumerated by user |
| SUBTYPE | Subset of any type (pre- or user-defined) |
| ARRAY | Single-type collection of any type above |
| RECORD | Multiple-type collection of any types above |

**Παράδειγμα νόμιμων και μη αναθέσεων και πράξεων σε συνθέσιμου τύπου δεδομένων**

```
TYPE byte IS ARRAY (7 DOWNTO 0)
OF STD_LOGIC;
-- 1D array

TYPE mem1 IS ARRAY
(0 TO 3, 7 DOWNTO 0) OF STD_LOGIC;
-- 2D array

TYPE mem2 IS ARRAY (0 TO 3) OF byte;
-- 1Dx1D array

TYPE mem3 IS ARRAY (0 TO 3)
OF STD_LOGIC_VECTOR(0 TO 7);
-- 1Dx1D array

SIGNAL a: STD_LOGIC;
-- scalar signal

SIGNAL b: BIT;
-- scalar signal

SIGNAL x: byte;
-- 1D signal

SIGNAL y: STD_LOGIC_VECTOR
(7 DOWNTO 0);
-- 1D signal

SIGNAL v: BIT_VECTOR (3 DOWNTO 0);
-- 1D signal

SIGNAL z: STD_LOGIC_VECTOR
(x'HIGH DOWNTO 0);
-- 1D signal

SIGNAL w1: mem1;
-- 2D signal

SIGNAL w2: mem2;
-- 1Dx1D signal

SIGNAL w3: mem3;
-- 1Dx1D signal

-------- Legal scalar assignments:

x(2) <= a;
-- same types (STD_LOGIC),
-- correct indexing

y(0) <= x(0);
-- same types (STD_LOGIC),
-- correct indexing

z(7) <= x(5);
-- same types (STD_LOGIC),
-- correct indexing

b <= v(3);
-- same types (BIT),
-- correct indexing

w1(0,0) <= x(3);
-- same types (STD_LOGIC),
-- correct indexing
```

```
w1(2,5) <= y(7);
-- same types (STD_LOGIC),
-- correct indexing

w2(0)(0) <= x(2);
-- same types (STD_LOGIC),
-- correct indexing

w2(2)(5) <= y(7);
-- same types (STD_LOGIC),
-- correct indexing

w1(2,5) <= w2(3)(7);
-- same types (STD_LOGIC),
-- correct indexing

------- Illegal scalar assignments:

b <= a;
-- type mismatch (BIT x STD_LOGIC)

w1(0)(2) <= x(2);
-- index of w1 must be 2D

w2(2,0) <= a;
-- index of w2 must be 1Dx1D

------- Legal vector assignments:

x <= "11111110";
y <=
('1','1','1','1','1','1','0','Z');

z <= "11111" & "000";

x <= (OTHERS => '1');

y <= (7 =>'0', 1 =>'0',
OTHERS => '1');

z <= y;

y(2 DOWNTO 0) <= z(6 DOWNTO 4);

w2(0)(7 DOWNTO 0) <= "11110000";

w3(2) <= y;

z <= w3(1);

z(5 DOWNTO 0) <= w3(1)(2 TO 7);

w3(1) <= "00000000";

w3(1) <= (OTHERS => '0');

w2 <= ((OTHERS=>'0'),(OTHERS=>'0'),
(OTHERS=>'0'),(OTHERS=>'0'));

w3 <= ("11111100",
 ('0','0','0','0','Z','Z','Z','Z',),
(OTHERS=>'0'), (OTHERS=>'0'));

w1   <=   ((OTHERS=>'Z'),   "11110000"
,"11110000", (OTHERS=>'0'));

------ Illegal array assignments:

x <= y;
-- type mismatch

y(5 TO 7) <= z(6 DOWNTO 0);
-- wrong direction of y

w1 <= (OTHERS => '1');
-- w1 is a 2D array

w1(0, 7 DOWNTO 0) <="11111111";
-- w1 is a 2D array

w2 <= (OTHERS => 'Z');
-- w2 is a 1Dx1D array

w2(0, 7 DOWNTO 0) <= "11110000";
-- index should be 1Dx1D

-- Example of data type independent
-- array initialization:

FOR i IN 0 TO 3 LOOP
FOR j IN 7 DOWNTO 0 LOOP
x(j) <= '0';
y(j) <= '0'
z(j) <= '0';
w1(i,j) <= '0';
w2(i)(j) <= '0';
w3(i)(j) <= '0';
END LOOP;
END LOOP;
```

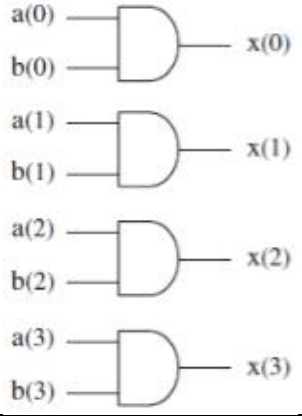
## Παράδειγμα διαφορών μεταξύ τύπου BIT και BIT_VECTOR

Στο παράδειγμα αυτό θα δούμε τις διαφορές μεταξύ τύπου μοναδικού bit και bit vector. Δηλαδή μεταξύ BIT και BIT_VECTOR, STD_LOGIC και STD_LOGIC_VECTOR ή STD_ULOGIC και STD_ULOGIC_VECTOR.

Στη συνέχεια θα παρουσιάσουμε δύο VHDL κώδικες. Και οι δύο υλοποιούν την πράξη AND μεταξύ σημάτων εισόδου και αποδίδουν το αποτέλεσμα σε ένα σήμα εξόδου. Η διαφορά μεταξύ τους είναι στο πλήθος των bits των σημάτων εισόδου και εξόδου (ένα bit στον πρώτο κώδικα και τέσσερα bits στον δεύτερο).

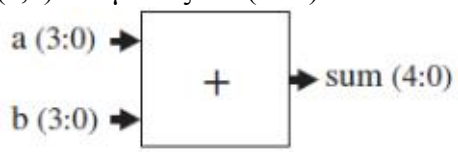
```
ENTITY and2
ISPORT
(a, b: IN BIT;
x: OUT BIT);
END and2;
```

```
ENTITY and2
IS PORT
(a, b: IN
BIT_VECTOR
0 TO 3);
```

| | |
|---|---|
| | x: OUT BIT_VECTOR 0 TO 3));<br>END and2; |
| ARCHITECTURE and2 OF and2 IS<br>BEGIN<br>x <= a AND b;<br>END and2; | ARCHITECTURE and2 OF and2 IS<br>BEGIN<br>x <= a AND b;<br>END and2; |
|  |  |

## Παράδειγμα δύο υλοποιήσεων Adder

Θα υλοποιήσουμε με VHDL έναν αθροιστή 4 bit. Το κύκλωμα έχει δύο εισόδους (a,b) και μια έξοδο (sum):




Στην πρώτη υλοποίηση όλα τα σήματα θα είναι SIGNED:

```
1 ----- Solution 1: in/out=SIGNED --
--------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 USE ieee.std_logic_arith.all;
5  ---------------------------------
--------
6 ENTITY adder1 IS
7 PORT ( a, b : IN SIGNED (3 DOWNTO
0);
8 sum : OUT SIGNED (4 DOWNTO 0));
9 END adder1;
10  ---------------------------------
---------
11 ARCHITECTURE adder1 OF adder1 IS
12 BEGIN
13 sum <= a + b;
14 END adder1;
```

Παρατηρούμε στη γραμμή 4, τη αναγκαία χρήση του πακέτου std_logic_arith, στο οποίο ορίζονται οι τύπου SIGNED. Επίσης υπενθυμίζουμε ότι, όπως φαίνεται στις γραμμές 7 και 8, οι SIGNED τιμές παριστάνονται ως διανύσματα, δηλαδή αντίστοιχα με τις STD_LOGIC_VECTOR και όχι ως INTEGER.

Στη δεύτερη υλοποίηση, η έξοδος θα είναι INTEGER:

```
1 ------ Solution 2: out=INTEGER ---
--------
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 USE ieee.std_logic_arith.all;
5  ---------------------------------
--------
6 ENTITY adder2 IS
7 PORT ( a, b : IN
SIGNED (3 DOWNTO 0);
8 sum : OUT
INTEGER RANGE -16 TO 15);
9 END adder2;
10 ---------------------------------
11 ARCHITECTURE adder2 OF adder2 IS
12 BEGIN
13 sum <= CONV_INTEGER(a + b);
14 END adder2;
```

Παρατηρούμε στη γραμμή 13 τη χρήση της συνάρτησης μετατροπής, διότι το a+b δεν είναι ίδιου τύπου με το sum.

Στον επόμενο πίνακα παραθέτουμε αποτελέσματα προσομοίωσης (και για τις δύο υλοποιήσεις). Οι αριθμοί αναπαρίστανται σε δεκαεξαδική μορφή συμπληρώματος ως προς 2. Υπενθυμίζουμε το συμπλήρωμα ως προς 2 για έναν 4bit ακέραιο:

| Two's complement | Decimal |
|---|---|
| 0111 | 7 |
| 0110 | 6 |
| 0101 | 5 |
| 0100 | 4 |
| 0011 | 3 |
| 0010 | 2 |
| 0001 | 1 |
| 0000 | 0 |
| 1111 | −1 |
| 1110 | −2 |
| 1101 | −3 |
| 1100 | −4 |
| 1011 | −5 |
| 1010 | −6 |
| 1001 | −7 |
| 1000 | −8 |

| a | | b | | sum | |
|---|---|---|---|---|---|
| H00 | =0 | H00 | =0 | H0000 | =0 |

| H02 | =2 | H04 | =4 | H0006 | =6 |
|---|---|---|---|---|---|
| H04 | =4 | H08 | =-8 | H001C | =-4 |
| H06 | =6 | H0C | =-4 | H0002 | =2 |
| H08 | =-8 | H00 | =0 | H0018 | =-8 |
| H0A | =-6 | H04 | =4 | H001E | = |
| H0C | =-4 | H08 | =-8 | H0014 | = |
| H0E | =-1 | H0C | =-4 | H001A | = |

## Υλοποίηση πλήρη αθροιστή 4 bit σε VHDL

Λύση.

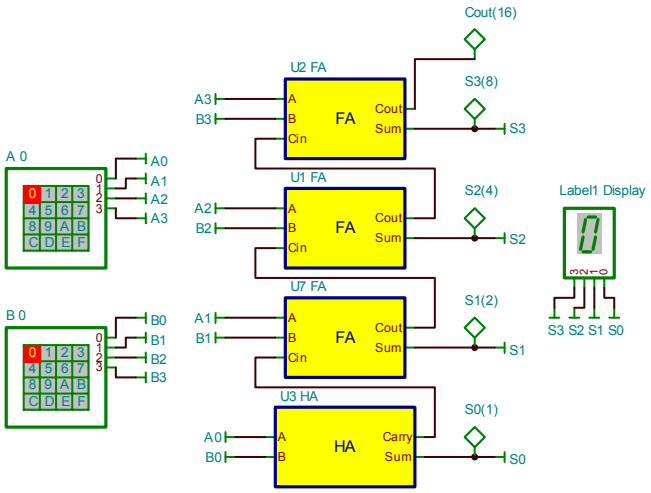
```
-- 1-bit Adder

LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

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

architecture BHV of BIT_ADDER is
begin
        -- Calculate the sum of the 1-BIT adder.
        sum <=  (not a and not b and cin) or
                (not a and b and not cin) or
                (a and not b and not cin) or
                (a and b and cin);

        -- Calculates the carry out of the 1-BIT
adder.
        cout <= (not a and b and cin) or
                (a and not b and cin) or
                (a and b and not cin) or
                (a and b and cin);
end BHV;

--4 bit Adder
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity add4 is
port( a, b              : in    STD_LOGIC_VECTOR(3
downto 0);
ans             : out   STD_LOGIC_VECTOR(3     downto
0);
cout            : out   STD_LOGIC               );
end add4;

architecture STRUCTURE of add4 is

component BIT_ADDER
port( a, b, cin                 : in  STD_LOGIC;
sum, cout               : out STD_LOGIC );
end component;

signal c0, c1, c2, c3 : STD_LOGIC;

begin
c0 <= '0';
b_adder0:
BIT_ADDER port map (a(0),b(0),c0, ans(0), c1);
b_adder1:
BIT_ADDER port map (a(1),b(1),c1, ans(1), c2);
b_adder2:
BIT_ADDER port map (a(2),b(2),c2, ans(2), c3);
b_adder3:
BIT_ADDER port map (a(3),b(3),c3,ans(3), cout);
END STRUCTURE;

-- 4-bit Adder Testbench
LIBRARY IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity TEST_ADD4 is
end TEST_ADD4;
```

```
architecture TEST of TEST_ADD4 is
        component add4
        port( a, b      :   in      STD_LOGIC_VECTOR(3
downto 0);
                ans     : out     STD_LOGIC_VECTOR(3
downto 0);
                cout    : out STD_LOGIC);
        end component;

        for       U1:       add4       use       entity
WORK.ADD4(STRUCTURE);
        signal a, b     :  STD_LOGIC_VECTOR(3  downto
0);
        signal ans      :  STD_LOGIC_VECTOR(3  downto
0);
        signal cout     : STD_LOGIC;

        begin
        U1: add4 port map (a,b,ans,cout);

        process
        begin
        -- Case 1 that we are testing.
        a <= "0000";
        b <= "0000";
        wait for 10 ns;
        assert ( ANS = "0000" )
report "Failed Case 1 - ANS" severity error;
        assert ( Cout = '0' )
report "Failed Case 1 - Cout" severity error;
        wait for 40 ns;
        -- Case 2 that we are testing.
        a <= "1111";
        b <= "1111";
        wait for 10 ns;
        assert ( ANS = "1110" )
report "Failed Case 2
- ANS" severity error;
        assert ( Cout = '1' )
report "Failed Case 2
-- Cout" severity error;
        wait for 40 ns;
        end process;
END TEST;
```

**Κώδικας.** Υλοποίηση πλήρη αθροιστή 4 bit με VHDL.

## Πλήρης αθροιστής στο TINA με χρήση των πληκτρολογίων για την εισαγωγή των αριθμών Α και Β

Λύση.




**Εικόνα.** Υλοποίηση – προσομοίωση λειτουργίας πλήρη αθροιστή στο TINA.

## Πλήρης αθροιστής-αφαιρέτης 4bit

```
This is the XOR gate
library ieee;
use ieee.std_logic_1164.all;
--
entity xorGate is
   port( A, B : in std_logic;
         F : out std_logic);
end xorGate;
--
architecture func of xorGate is
begin
   F <= A xor B;
end func;

-- This is the FULL ADDER
library ieee;
use ieee.std_logic_1164.all;
--
entity Full_Adder is
   port( X, Y, Cin : in std_logic;
         sum, Cout : out std_logic);
end Full_Adder;
--Dataflow architecture.
--See Full Adder on Teahlab.com for structural
version
architecture func of Full_Adder is
begin
   sum <= (X xor Y) xor Cin;
   Cout <= (X and (Y or Cin)) or (Cin and Y);
end func;

--Now we build the four bit Adder Subtractor
library ieee;
use ieee.std_logic_1164.all;
entity adderSubtractor is
   port( mode            : in std_logic;
         A3, A2, A1, A0  : in std_logic;
         B3, B2, B1, B0  : in std_logic;
         S3, S2, S1, S0  : out std_logic;
                Cout, V : out std_logic);
end adderSubtractor;
--Structural architecture
architecture struct of adderSubtractor is

   component xorGate is
--XOR component
       port( A, B : in std_logic;
             F : out std_logic);
   end component;

   component Full_Adder is
--FULL ADDER component
      port( X, Y, Cin : in std_logic;
            sum, Cout : out std_logic);
   end component;

   --interconnecting wires
   signal C1, C2, C3, C4: std_logic;
--intermediate carries
   signal xor0, xor1, xor2, xor3 : std_logic;
--xor outputs

begin
   GX0: xorGate port map(mode, B0, xor0);
   GX1: xorGate port map(mode, B1, xor1);
   GX2: xorGate port map(mode, B2, xor2);
   GX3: xorGate port map(mode, B3, xor3);

   FA0:
Full_Adder port map(A0, xor0, mode,  S0, C1);-- S0
   FA1:
Full_Adder port map(A1, xor1, C1,  S1, C2);  -- S1
   FA2:
Full_Adder port map(A2, xor2, C2,  S2, C3);  -- S2
   FA3:
Full_Adder port map(A3, xor3, C3,  S3, C4);  -- S3

   Vout:   xorGate   port   map(C3,   C4,   V);
-- V
   Cout                  <=                  C4;
-- Cout
end struct;

--  Program:  Four  Bit  Adder-Substractor  VHDL
Testbench

library ieee;
use ieee.std_logic_1164.all;
entity adderSubtractor_tb is -- void input; void
output
end adderSubtractor_tb;

architecture testbench of adderSubtractor_tb is
   component adderSubtractor is
      port( mode : in std_logic;
         A3, A2, A1, A0 : in std_logic;
         B3, B2, B1, B0  : in std_logic;
         S3, S2, S1, S0  : out std_logic;
                Cout, V  : out std_logic);
   end component;

   signal mode, A3, A2, A1, A0 : std_logic;
   signal B3, B2, B1, B0 : std_logic;
   signal S3, S2, S1, S0, Cout, V : std_logic;

begin
   mapping: adderSubtractor port map(
            mode,      A3, A2, A1, A0,
                       B3, B2, B1, B0,
                       S3, S2, S1, S0,
                             Cout, V);
   --concurrent processes. just to save space
   process
   begin
      mode <= '1'; -- do subtraction
      wait for 10 ns;
      mode <= '0'; -- do addition
      wait for 10 ns;
   end process;

   process
      variable errCnt : integer :=0;
   begin
      --TEST 1
      A3 <= '1';
      A2 <= '0';
      A1 <= '1';
      A0 <= '1';
      --
      B3 <= '0';
      B2 <= '1';
      B1 <= '1';
      B0 <= '0';
      --
      wait for 20 ns;
      assert (Cout = '1')
      report "Error" severity error;
      assert  (S3  =  '0')  report  "Error"  severity
error;
      assert  (S2  =  '0')  report  "Error"  severity
error;
      assert  (S1  =  '0')  report  "Error"  severity
error;
      assert  (S0  =  '1')  report  "Error"  severity
error;
      assert  (V  =  '0')  report  "Error"  severity
error;
      if(Cout /= '1' or V /= '0') then
         errCnt:= errCnt + 1;
      end if;

      --TEST 2
      A3 <= '0';
      A2 <= '1';
      A1 <= '1';
      A0 <= '1';
      --
      B3 <= '0';
      B2 <= '1';
      B1 <= '0';
      B0 <= '1';
      --
      wait for 20 ns;
      assert (Cout = '0')
      report "Error" severity error;
```

```
      assert (S3 = '1') report "Error" severity
error;
      assert (S2 = '1') report "Error" severity
error;
      assert (S1 = '0') report "Error" severity
error;
      assert (S0 = '0') report "Error" severity
error;
      assert (V = '1') report "Error" severity
error;
      if(Cout /= '0' or V /= '1') then
         errCnt:= errCnt + 1;
      end if;

      ------------ SUMMARY ----------------
      if(errCnt = 0) then
         assert false report "Good!"    severity
note;
      else
         assert false report "Error!"    severity
error;
      end if;

   end process;
end testbench;
-----------------------------------------
configuration cfg_tb of adderSubtractor_tb is
   for testbench
   end for;
end cfg_tb;
```

**Κώδικας.** Υλοποίηση πλήρη αθροιστή-αφαιρέτη 4 bit με VHDL.

## Ασκήσεις

Χρησιμοποιείστε τις επόμενες δηλώσεις τύπων και σημάτων για να απαντήσετε τις ασκήσεις που ακολουθούν:

```
TYPE array1 IS ARRAY (7 DOWNTO 0)
OF STD_LOGIC;

TYPE array2 IS
ARRAY (3 DOWNTO 0, 7 DOWNTO 0)
OF STD_LOGIC;

TYPE array3 IS ARRAY (3 DOWNTO 0)
OF array1;

SIGNAL a : BIT;

SIGNAL b : STD_LOGIC;

SIGNAL x : array1;

SIGNAL y : array2;

SIGNAL w : array3;

SIGNAL z :
STD_LOGIC_VECTOR (7 DOWNTO 0);
```

1) Προσδιορίστε την διάσταση (βαθμωτός, 1D, 2D, 1Dx1D) των παραπάνω σημάτων. Γράψτε ένα παράδειγμα σε κάθε περίπτωση.

2) Συμπληρώστε ποιες από τις παρακάτω δηλώσεις είναι νόμιμες ή όχι και γιατί και τη διάσταση των σημάτων σε κάθε μέλος τους.

| **Δήλωση** | **Διάσταση κάθε μέλους** | **Νόμιμη ή όχι και γιατί** |
|---|---|---|
| `a <= x(2);` | | |
| `b <= x(2);` | | |
| `b <= y(3,5);` | | |
| `b <= w(5)(3);` | | |
| `y(1)(0) <= z(7);` | | |
| `x(0) <= y(0,0);` | | |
| `x <= "1110000";` | | |
| `a <= "0000000";` | | |
| `y(1) <= x;` | | |
| `w(0) <= y;` | | |
| `w(1) <= (7=>'1', OTHERS=>'0');` | | |
| `y(1) <= (0=>'0', OTHERS=>'1');` | | |
| `w(2)(7 DOWNTO 0) <= x;` | | |
| `w(0)(7 DOWNTO 6) <= z(5 DOWNTO 4);` | | |
| `x(3) <= x(5 DOWNTO 5);` | | |
| `b <= x(5 DOWNTO 5);` | | |
| `y <= ((OTHERS=>'0'), (OTHERS=>'0'), (OTHERS=>'0'), "10000001");` | | |
| `z(6) <= x(5);` | | |
| `z(6 DOWNTO 4) <= x(5 DOWNTO 3);` | | |
| `z(6 DOWNTO 4) <= y(5 DOWNTO 3);` | | |
| `y(6 DOWNTO 4) <= z(3 TO 5);` | | |
| `y(0, 7 DOWNTO 0) <= z;` | | |
| `w(2,2) <= '1';` | | |

3) Θεωρήστε του βασικούς τύπους INTEGER και STD_LOGIC_VECTOR. Θεωρήστε επίσης τους τύπου ARRAY1 και ARRAY2 που ορίσαμε παραπάνω. Για κάθε έναν γράψτε ένα παράδειγμα ενός SUBTYPE.

4) Γράψτε έναν κώδικα περιγραφής απλού αθροιστή, όπως είδαμε παραπάνω,

χρησιμοποιώντας τον STD_LOGC_VECTOR για όλα τα σήματα εισόδου και εξόδου.

## Τελεστές και ιδιότητες (Operators and Attributes)

### Τελεστές (Operators)

Η VHDL παρέχει διάφορα είδη τελεστών ορισμένων στις βασικές της βιβλιοθήκες:

- Τελεστές απόδοσης τιμής (assignment)
- Τελεστές λογικών πράξεων (logical)
- Τελεστές αριθμητικών πράξεων (arithmetic)
- Τελεστές σχεσιακών πράξεων (relational)
- Τελεστές ολίσθησης (shift)
- Τελεστές συρραφής (concatenation)

### Τελεστές απόδοσης τιμής

Χρησιμοποιούνται για την απόδοση τιμών σε σήματα, μεταβλητές και σταθερές. Πρόκειται για τους:

| | |
|---|---|
| <= | Χρησιμοποιείται για την απόδοση τιμή σε σήμα. |
| := | Χρησιμοποιείται για την απόδοση τιμής σε VARIABLE, CONSTANT, ή GENERIC. Χρησιμοποιείται επίσης για την απόδοση αρχικών τιμών. |
| => | Χρησιμοποιείται για την απόδοση τιμών σε επιμέρους στοιχεία διανυσμάτων ή με την OTHERS. |

Παραδείγματα:

```
SIGNAL x : STD_LOGIC;

VARIABLE y :
STD_LOGIC_VECTOR(3 DOWNTO 0);
-- Leftmost bit is MSB

SIGNAL w:
STD_LOGIC_VECTOR(0 TO 7);
-- Rightmost bit is MSB

x <= '1';
-- '1' is assigned to SIGNAL x
-- using "<="

y := "0000";
-- "0000" is assigned to VARIABLE y
-- using ":="

w <= "10000000";
-- LSB is '1', the others are '0'

w <= (0 =>'1', OTHERS =>'0');
-- LSB is '1', the others are '0'
```

### Τελεστές πράξεων λογικής

Χρησιμοποιούνται για την εκτέλεση πράξεων λογικής. Τα δεδομένα πρέπει να είναι τύπου BIT, STD_LOGIC, ή STD_ULOGIC (ή αντίστοιχα οι επεκτάσεις τους: BIT_VECTOR, STD_LOGIC_VECTOR ή STD_ULOGIC_VECTOR). Πρόκειται για τους:

- NOT
- AND
- OR
- NAND
- NOR
- XOR
- XNOR

Ο τελεστής NOT έχει προτεραιότητα έναντι των άλλων.

Παραδείγματα:

```
y <= NOT a AND b; -- (a'.b)
y <= NOT (a AND b); -- (a.b)'
y <= a NAND b; -- (a.b)'
```

### Τελεστές αριθμητικών πράξεων

Χρησιμοποιούνται για την εκτέλεση αριθμητικών πράξεων. Τα δεδομένα πρέπει να είναι τύπου INTEGER, SIGNED, UNSIGED ή REAL (ο τελευταίος τύπος δε είναι άμεσα συνθέσιμος). Αν κάνουμε χρήση του πακέτου std_logic_signed ή του std_logic_unsigned της ieee, τότε ο τύπος STD_LOGIC_VECTOR μπορεί να χρησιμοποιηθεί άμεσα σε πράξεις πρόσθεσης και αφαίρεσης. Οι τελεστές αυτοί είναι:

| | |
|---|---|
| + | Πρόσθεση |
| - | Αφαίρεση |
| * | Πολλαπλασιασμός |
| / | Διαίρεση |
| ** | Εκθέτης |
| MOD | Modulus |
| REM | Υπόλοιπο |
| ABS | Απόλυτη τιμή |

Για την πρόσθεση και την αφαίρεση δεν υπάρχουν περιορισμοί ως προς τη συνθεσιμότητα. Το ίδιο ισχύει γενικά και για τον πολλαπλασιασμό. Για τη διαίρεση ωστόσο, μόνον διαιρέτες που είναι δυνάμεις του 2, επιτρέπονται (δηλαδή ολίσθηση). Για δυνάμεις, μόνον στατικές τιμές βάσης και εκθέτη επιτρέπονται. Όσον αφορά το mod και το rem, το y mod x επιστρέφει το υπόλοιπο της y/x με το σήμα x, ενώ το y rem x επιστρέφει το υπόλοιπο της y/x με το σήμα y. Για τους mod, rem, abs, γενικά υπάρχει περιορισμένη ή καθόλου υποστήριξη σύνθεσης.

## Τελεστές σύγκρισης

Χρησιμοποιούνται για να κάνουμε συγκρίσεις. Τα δεδομένα μπορεί να είναι οποιουδήποτε από τους προαναφερθέντες τύπους. Οι τελεστές αυτοί είναι οι:

- =
- /=
- <
- >
- <=
- >=

## Τελεστές ολίσθησης

Χρησιμοποιούνται για την ολίσθηση δεδομένων. Η σύνταξή τους έχει τη δομή: <left operand> <shift operation> <right operand>. Ο αριστερός τελεστέος πρέπει είναι τύπου BIT_VECTOR, ενώ ο δεξιός τελεστέος πρέπει να είναι INTEGER (+ ή – μπροστά του, είναι δεκτό). Οι πράξεις ολίσθησης είναι οι επόμενες:

| | |
|---|---|
| SLL | Shift Left Logic: οι θέσεις στα δεξιά γεμίζουμε με 0. |
| SRL | Shift Right Logic: οι θέσεις στα αριστερά γεμίζουν με 0. |

## Ιδιότητες δεδομένων

Οι ορισμένες στη VHDL συνθέσιμες ιδιότητες δεδομένων είναι οι:

| | |
|---|---|
| D'LOW | Επιστρέφει το μικρότερο δείκτη πίνακα |
| D'HIGH | Επιστρέφει τον πάνω δείκτη πίνακα |
| D'LEFT | Επιστρέφει τον πιο αριστερό δείκτη πίνακα |
| D'RIGHT | Επιστρέφει τον πιο δεξιό δείκτη πίνακα |
| D'LENGTH | Επιστρέφει το μήκος τους διανύσματος |
| D'RANGE | Επιστρέφει την εμβέλεια του διανύσματος |
| D'REVERSE_RANGE | Επιστρέφει την εμβέλεια του διανύσματος με αντίστροφη σειρά |

Παραδείγματα:

```
SIGNAL d :
STD_LOGIC_VECTOR (7 DOWNTO 0);

Τότε:
d'LOW=0,
d'HIGH=7,
d'LEFT=7,
d'RIGHT=0,
d'LENGTH=8,
d'RANGE=(7 downto 0),
d'REVERSE_RANGE=(0 to 7)
```

```
SIGNAL x: STD_LOGIC_VECTOR (0 TO 7);

-- all four LOOP statements below
are
-- synthesizable and equivalent.

FOR i IN RANGE (0 TO 7) LOOP ...
FOR i IN x'RANGE LOOP ...
FOR i IN RANGE (x'LOW TO x'HIGH)
LOOP ...
FOR i IN RANGE (0 TO x'LENGTH-1)
LOOP ...
```

Αν το σήμα είναι απαριθμήσιμου τύπου, τότε:

| | |
|---|---|
| d'VAL(pos) | Επιστρέφει τιμή στη θέση που προσδιορίζεται |
| d'POS(value) | Επιστρέφει θέση στην τιμή που προσδιορίζεται |
| d'LEFTOF(value) | Επιστρέφει τιμή στη θέση στα αριστερά της τιμής που προσδιορίζεται |
| d'VAL(row, column) | Επιστρέφει τιμή στη θέση που προσδιορίζεται |

Για τους απαριθμήσιμους τύπους υπάρχει περιορισμένη ή καθόλου συνθεσιμότητα.

## Ιδιότητες σημάτων

Έστω ένα σήμα s. Τότε:

| | |
|---|---|
| s'EVENT | Επιστρέφει true όταν συμβεί γεγονός στο s |
| s'STABLE | Επιστρέφει true αν δεν συμβεί γεγονός στο s |
| s'ACTIVE | Επιστρέφει true αν s='1' |
| s'QUIET <time> | Επιστρέφει true αν δε συμβεί γεγονός στο προσδιορισμένο χρονικό διάστημα |
| s'LAST_EVENT | Επιστρέφει το χρόνο που πέρασε από το τελευταίο συμβάν |
| s'LAST_ACTIVE | Επιστρέφει το χρόνο |

| | |
|---|---|
| | που πέρασε από την τελευταία που φορά που s='1' |
| s'LAST_VALUE | Επιστρέφει την τιμή του s πριν από το τελευταίο συμβάν |

Οι περισσότερες ιδιότητες σημάτων χρησιμοποιούνται μόνο για προσομοίωση. Ωστόσο οι πρώτες δύο της παραπάνω λίστας είναι συνθέσιμες. Η s'EVENT είναι μάλιστα η ιδιότητα που χρησιμοποιείται συχνότερα.

Παράδειγμα. Οι τέσσερεις δηλώσεις που ακολουθούν είναι συνθέσιμες και ισοδύναμες. Επιστρέφουν true όταν συμβεί ένα γεγονός (μια αλλαγή) στο clk και εφόσον αυτή η αλλαγή είναι προς τα πάνω (δηλαδή στην ακμή ανύψωσης του clk):

```
IF (clk'EVENT AND clk='1')...
-- EVENT attribute used with IF
```

```
IF (NOT clk'STABLE AND
clk='1')...
-- STABLE attribute used with IF
```

```
WAIT UNTIL (clk'EVENT AND
clk='1');
-- EVENT attribute used with WAIT
```

```
IF RISING_EDGE(clk)...
-- call to a function
```

## Ιδιότητες ορισμένες από το χρήστη

Η VHDL επιτρέπει στο χρήση να ορίσει δικές του ιδιότητες στα δεδομένα. Μια τέτοια ιδιότητα πρέπει να δηλωθεί και να οριστεί. Η σύνταξης δήλωσης της ιδιότητας έχει ως εξής:

```
ATTRIBUTE attribute_name:
attribute_type;
```

Η σύνταξης ορισμού της ιδιότητας ως εξής:

```
ATTRIBUTE attribute_name
OF target_name: class IS value;
```

Όπου:

| | |
|---|---|
| attribute_type | Οποιοσδήποτε τύπος δεδομένων (BIT, INTEGER, STD_LOGIC_VECTOR, etc.) |
| class | TYPE, SIGNAL, FUNCTION, etc. |
| value | '0', 27, ''00 11 10 01'', etc. |

Παραδείγματα:

```
ATTRIBUTE number_of_inputs: INTEGER;
-- declaration

ATTRIBUTE number_of_inputs OF nand3:
SIGNAL IS 3;
-- specification
...
inputs <= nand3'number_of_pins;
-- attribute call, returns 3
```

## Παράδειγμα κωδικοποίησης απαριθμητού τύπου

Μια δημοφιλής ιδιότητα ορισμένη από χρήστη, που παρέχεται και από τους κατασκευαστές εργαλείων σύνθεσης, είναι η enum_encoding. Εξ' ορισμού, οι απαριθμητοί τύποι δεδομένων, κωδικοποιούνται σειριακά Π.χ. έστω ο απαριθμητός τύπος δε δομένων color:

```
TYPE color IS
(red, green, blue, white);
```

Οι καταστάσεις τους θα κωδικοποιηθούν ως

```
red = ''00'', green = ''01'',
 blue = ''10'', white =''11''
```

Η enum_encoding μας επιτρέπει να αλλάξουμε την εξορισμού ακολουθιακή αντιστοίχιση. Π.χ.:

```
ATTRIBUTE enum_encoding OF color:
TYPE IS "11 00 10 01";
```

Μια ιδιότητα ορισμένη από χρήστη, μπορεί να δηλωθεί σε οποιοδήποτε σημείο του κώδικα, εκτός από το σώμα του PACKAGE. Όταν το εργαλείο σύνθεσης, δεν την αναγνωρίζει, την αγνοεί ή παράγει ένα σήμα ενημέρωσης.

## Υπερφόρτωση τελεστών (Operator overloading)

Όπως είδαμε, ιδιότητες δεδομένων, μπορούν να οριστούν από το χρήστη. Το ίδιο μπορεί να γίνει και για τους τελεστές. Π.χ. ας θεωρήσουμε τους ορισμένους στη VHDL αριθμητικούς τελεστές (+, -, *, / , κτλ). Αυτοί προσδιορίζουν το είδος των αριθμητικών πράξεων μεταξύ δεδομένων συγκεκριμένου τύπου (π.χ. INTEGER). Όμως, για παράδειγμα, ο «+» δεν επιτρέπει πρόσθεση μεταξύ δεδομένων τύπου BIT. Μπορούμε να ορίσουμε τους δικούς μας τελεστές, χρησιμοποιώντας ακόμα και το ίδιο όνομα με αυτούς που ήδη έχει η VHDL. Για παράδειγμα, μπορούμε να χρησιμοποιήσουμε τον «+» για να υποδεικνύει ένα νέο είδος πρόσθεση π.χ. μεταξύ δεδομένα τύπου BIT_VECTOR. Αυτή η τεχνική ονομάζεται υπερφόρτωση τελεστών.

Παράδειγμα: Έστω ότι θέλουμε να προσθέσουμε ένα ακέραιο σε έναν δυαδικό αριθμό 1-bit. Η επόμενη FUNCTION μπορεί να χρησιμοποιηθεί για το σκοπό αυτό:

```
FUNCTION "+"
(a: INTEGER, b: BIT)
RETURN INTEGER IS
BEGIN
```

```
IF (b='1') THEN RETURN a+1;
ELSE RETURN a;
END IF;
END "+";
```

Η κλήση της συνάρτησης αυτή μπορεί να γίνει όπως φαίνεται στο ακόλουθο παράδειγμα:

```
SIGNAL inp1, outp: INTEGER RANGE 0
TO 15;
SIGNAL inp2: BIT;
(...)
outp <= 3 + inp1 + inp2;
(...)
```

Στην outp <= 3 + inp1 + inp2; το πρώτο + είναι ο εξορισμού τελεστής πρόσθεσης (που προσθέσει δύο ακεραίους), ενώ το δεύτερο + είναι το υπερφορτωμένο σύμβολο της πρόσθεσης ακεραίου με bit.

## Ενδογενείς παράμετροι (GENERIC)

Οι παράμετροι αυτοί είναι στατικές. Σκοπός τους είναι να κάνουν τον κώδικα πιο ευέλικτο και εύχρηστο.

Μια δήλωση παραμέτρου GENERIC, πρέπει να γίνει μέσα στη ENTITY. Η παράμετρος αυτή στη συνέχεια θα είναι πραγματικά καθολική (δηλαδή φανερή σε όλο το σχέδιο συμπεριλαμβανομένης και την ίδιας της ENTITY). Η σύνταξη τους φαίνεται στη συνέχεια:

```
GENERIC (parameter name : parameter
type := parameter value);
```

Παράδειγμα: Η παράμετρος GENERIC παρακάτω ορίζει μια παράμετρο που λέγεται n, τύπου INTEGER, της οποίας η εξορισμού τιμή είναι 8. Άρα όπου στην ENTITY ή στην ARCHITECTURE εμφανίζεται το n, η τιμή τους θα είναι 8:

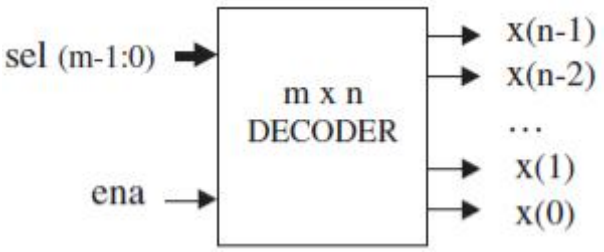
```
ENTITY my_entity IS
GENERIC (n : INTEGER := 8);
PORT (...);
END my_entity;
ARCHITECTURE my_architecture OF
my_entity IS
...
END my_architecture:
```

Στην ENTITY μπορούμε να ορίσουμε περισσότερες από μια GENERIC παραμέτρους. Π.χ.:

```
GENERIC (n: INTEGER := 8;
vector: BIT_VECTOR := "00001111");
```

## Παράδειγμα. M x N ενδογενής αποκωδικοποιητής

Θα υλοποιήσουμε το κύκλωμα της επόμενης εικόνας:




| ena | sel | x |
|---|---|---|
| 0 | 00 | 1111 |
| 1 | 00 | 1110 |
| | 01 | 1101 |
| | 10 | 1011 |
| | 11 | 0111 |

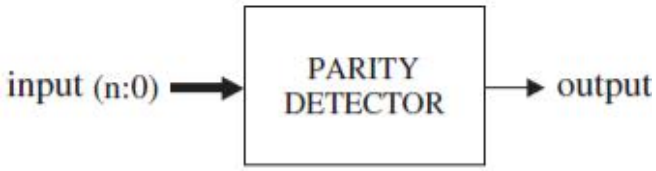
Υποθέτουμε ότι το n είναι δύναμη του 2, οπότε $m=\log_2 n$.

Στη συνέχεια φαίνεται η υλοποίηση:

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

ENTITY decoder IS
PORT ( ena : IN STD_LOGIC;
sel : IN
STD_LOGIC_VECTOR (2 DOWNTO 0);
 x : OUT
STD_LOGIC_VECTOR (7 DOWNTO 0));
END decoder;

ARCHITECTURE generic_decoder OF
decoder IS
BEGIN

PROCESS (ena, sel)
VARIABLE temp1 :
STD_LOGIC_VECTOR (x'HIGH DOWNTO 0);
VARIABLE temp2 :
INTEGER RANGE 0 TO x'HIGH;
BEGIN
temp1 := (OTHERS => '1');
temp2 := 0;
IF (ena='1') THEN
FOR i IN sel'RANGE LOOP
-- sel range is 2 downto 0
IF (sel(i)='1') THEN
-- Bin-to-Integer conversion
temp2:=2*temp2+1;
ELSE
temp2 := 2*temp2;
END IF;
END LOOP;
temp1(temp2):='0';
END IF;
x <= temp1;
END PROCESS;
END generic_decoder;
```

## Παράδειγμα. Ενδογενής ελεγκτής ισοτιμίας

Θα υλοποιήσουμε έναν ελεγκτή ισοτιμίας του οποίου το μπλοκ διάγραμμα φαίνεται στη συνέχεια:

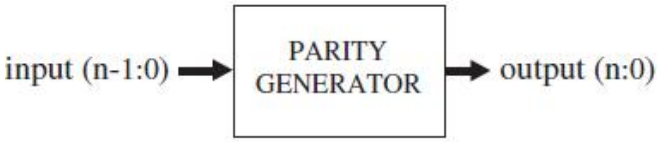



Το κύκλωμα πρέπει να παράγει 0 όταν το πλήθος των 1 στο διάνυσμα εισόδου είναι άρτιος και 1 αν είναι περιττός.

```
ENTITY parity_det IS
GENERIC (n : INTEGER := 7);
PORT ( input: IN
BIT_VECTOR (n DOWNTO 0);
output: OUT BIT);
END parity_det;

ARCHITECTURE parity OF parity_det IS
BEGIN
PROCESS (input)
VARIABLE temp: BIT;
BEGIN
temp := '0';
FOR i IN input'RANGE LOOP
temp := temp XOR input(i);
END LOOP;
output <= temp;
END PROCESS;

END parity;
```

## Παράδειγμα. Γεννήτρια ισοτιμίας

Θα υλοποιήσουμε μια γεννήτρια ισοτιμίας, της οποίας το μπλοκ διάγραμμα φαίνεται στη συνέχεια:




Πρόκειται για ένα κύκλωμα που θα προσθέτει ένα επιπλέον bit στα αριστερά του διανύσματα εισόδου. Αυτό το προστιθέμενο bit θα πρέπει να είναι 0 αν το πλήθος των 1 στο διάνυσμα εισόδου είναι άρτιο, ή 1 αν είναι περιττό. Έτσι, ο τελικό διάνυσμα εξόδου θα πρέπει να περιέχει πάντα άρτιο πλήθος από 1 (άρτια ισοτιμία).

```
ENTITY parity_gen IS
GENERIC (n : INTEGER := 7);
PORT ( input: IN
BIT_VECTOR (n-1 DOWNTO 0);
output: OUT
BIT_VECTOR (n DOWNTO 0));
END parity_gen;

ARCHITECTURE parity OF parity_gen IS
BEGIN

PROCESS (input)
VARIABLE temp1: BIT;
VARIABLE temp2:
BIT_VECTOR (output'RANGE);
BEGIN
temp1 := '0';
FOR i IN input'RANGE LOOP
temp1 := temp1 XOR input(i);
temp2(i) := input(i);
END LOOP;
temp2(output'HIGH) := temp1;
output <= temp2;
END PROCESS

END parity;
```

## Σχεδίαση συνδυαστικών κυκλωμάτων με ακολουθιακό κώδικα

Ο ακολουθιακός κώδικας είναι κατάλληλος για την υλοποίηση τόσο των ακολουθιακών, όσο και των συνδυαστικών κυκλωμάτων. Στην πρώτη περίπτωση, ο μεταγλωττιστής θα πρέπει να υποθέσει την παρουσία καταχωρητών. Όχι όμως στη δεύτερη. Επιπλέον, αν ο κώδικας προορίζεται για συνδυαστικά κυκλώματα, τότε θα πρέπει να προσδιορίζει τον πλήρη πίνακα αληθείας του κυκλώματος.

Για να ικανοποιηθούν τα παραπάνω κριτήρια, θα πρέπει να ακολουθήσουμε τους επόμενους κανόνες:

Κανόνας 1: Όλα τα σήματα εισόδου σε μια PROCESS θα πρέπει να εμφανίζονται στη λίστα ορισμάτων της.

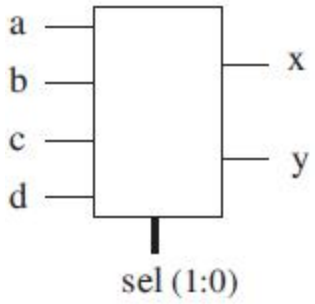
Κανόνας 2: Όλοι οι συνδυασμοί σημάτων εισόδου / εξόδου θα πρέπει να αναφέρονται μέσα στον κώδικα. Δηλαδή κοιτάζοντας τον κώδικα, θα πρέπει να μπορούμε να δούμε την αναπαραγωγή όλου του πίνακα αληθείας τους κυκλώματος (αυτό πρέπει βέβαια να ισχύει τόσο για τον ακολουθιακό όσο και για την παράλληλο κώδικα).

Αδυναμία υπακοής στον κανόνα 1 θα οδηγήσει το μεταγλωττιστή στην εκτύπωση μηνύματος που να λέει ότι ένα σήμα δεν υπάρχει στη λίστα παραμέτρων της διαδικασία και στη συνέχεια να συνεχίσει θεωρώντας ότι το σήμα βρίσκεται στη λίστα παραμέτρων. Αν και τελικά δεν υφίσταται λάθος στο σχέδιο, καλό είναι να ακολουθούμε τον κανόνα 1 και να μην αφήνουμε το μεταγλωττιστή να υποθέτει.

Όσον αφορά στον κανόνα 2, οι συνέπειες μπορεί να είναι σημαντικές, γιατί ελλιπής

προσδιορισμός των σημάτων εξόδου, μπορεί να κάνει το συνθέτη να υποθέσει την ύπαρξη μανδάλων (latches) για να διατηρεί προηγούμενες τιμές.

Στη συνέχεια θα δείξουμε ένα παράδειγμα τέτοιου κακού κώδικα. Ξεκινάμε θεωρώντας το επόμενο κύκλωμα:




Το x θέλουμε να συμπεριφέρεται ως έξοδος πολυπλέκτη, δηλαδή να είναι ίσο με την αντίστοιχη είσοδο που επιλέγεται μέσω του sel. Για το y θέλουμε να είναι ίσο με 0 όταν sel=00, ή 1 αν sel=01.

Οι προδιαγραφές αυτές είναι ελλιπείς όσον αφορά την πλήρωση πίνακα αληθείας. Η σωστές προδιαγραφές θα πρέπει να δώσουν το επόμενο πίνακα αληθείας:

| sel | x | y |
|---|---|---|
| 00 | a | 0 |
| 01 | b | 1 |
| 10 | c | X |
| 11 | d | X |

Συνεπώς, μια υλοποίηση του συνδυαστικού αυτού κυκλώματος είναι η επόμενη:

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

ENTITY example IS
PORT (a, b, c, d: IN STD_LOGIC;
sel: IN INTEGER RANGE 0 TO 3;
x, y: OUT STD_LOGIC);
END example;

ARCHITECTURE example OF example IS
BEGIN
PROCESS (a, b, c, d, sel)
BEGIN
IF (sel=0) THEN
x<=a;
y<='0';
ELSIF (sel=1) THEN
x<=b;
y<='1';
ELSIF (sel=2) THEN
x<=c;
y<='X';
ELSE
x<=d;
y<='X';
END IF;
END PROCESS;
END example;
```

Μετά την μεταγλώττιση αυτού του κώδικα, ο έλεγχος των αρχείων αναφοράς θα δείξει ότι δεν έχουν θεωρηθεί flip-flops (όπως ήταν αναμενόμενο).

## Ασκήσεις

Οι επόμενες ασκήσεις βασίζονται στους ακόλουθους ορισμούς σημάτων:

```
SIGNAL a : BIT := '1';

SIGNAL b :
BIT_VECTOR (3 DOWNTO 0) := "1100";

SIGNAL c :
BIT_VECTOR (3 DOWNTO 0) := "0010";

SIGNAL d :
BIT_VECTOR (7 DOWNTO 0);

SIGNAL e :
INTEGER RANGE 0 TO 255;

SIGNAL f :
INTEGER RANGE -128 TO 127;
```

1) Συμπληρώστε τα κενά:

```
x1 <= a & c; ->
x1 <= _________

x2 <= c & b; ->
x2 <= _________

x3 <= b XOR c; ->
x3 <= _________

x4 <= a NOR b(3); ->
x4 <= _________

x5 <= b sll 2; ->
x5 <= _________

x6 <= b sla 2; ->
x6 <= _________

x7 <= b rol 2; ->
x7 <= _________

x8 <= a AND NOT b(0) AND NOT c(1);
->
x8 <= _________

d <= (5=>'0', OTHERS=>'1'); ->
d<= _________
```

2) Συμπληρώστε τα κενά:

```
c'LOW -> _______
d'HIGH -> _______
c'LEFT -> _______
d'RIGHT -> _______
c'RANGE -> _______
d'LENGTH -> _______
c'REVERSE_RANGE -> _______
```

3) Να διαπιστώσετε ποιες από τις επόμενες πράξεις είναι νόμιμες και ποιες όχι.

| Πράξη | Νόμιμη | Όχι |
|---|---|---|
| b(0) AND a | | |
| a + d(7) | | |
| NOT b XNOR c | | |
| c + d | | |
| e - f | | |
| IF (b<c) ... | | |
| IF (b>=a) ... | | |
| IF (f/=e) ... | | |
| IF (e>d) ... | | |
| b sra 1 | | |
| c srl -2 | | |
| f ror 3 | | |
| e*3 | | |
| 5**5 | | |
| f/4 | | |
| e/3 | | |
| d <= c | | |
| d(6 DOWNTO 3) := b | | |
| e <= d | | |
| f := 100 | | |

**Διατάξεις Προγραμματιζόμενης Λογικής (Programmable Logic Devices) (PLDs)**

## Εισαγωγή

Οι διατάξεις προγραμματιζόμενης λογικής (PLDs) έκαναν την εμφάνιση τους στα μέσα της δεκαετίας του 70. Η ιδέα ήταν να κατασκευαστεί συνδυαστική λογική που να μπορεί να προγραμματιστεί. Ωστόσο, σε αντίθεση με τους μικροεπεξεργαστές που εκτελούν έναν πρόγραμμα αλλά βασίζονται σε δεδομένο υλικό, ο προγραμματισμός των PLDs στοχεύει σε επίπεδο υλικού. Δηλαδή ένα PLD είναι ένα τσιπ γενικού σκοπού, με το υλικό του να διαμορφώνεται κάθε φορά έτσι ώστε να καλύπτει τις ανάγκες και τις απαιτήσεις μας.

Τα πρώτα PLDs ονομάζονταν PAL (Programmable Array Logic) ή PLA (Programmable Logic Array), ανάλογα με το στυλ του προγραμματισμού που επιδέχονταν. Χρησιμοποιούσαν μόνο πύλες λογικής (όχι flip-flops) οπότε επέτρεπαν μόνο την υλοποίηση συνδυαστικών κυκλωμάτων. Στη συνέχεια κατασκευάστηκαν και PLDs με καταχωρητές. Αυτά περιείχαν και ένα flip-flop σε κάθε έξοδο του κυκλώματος. Οπότε ήταν δυνατό να υλοποιήσουν και απλά ακολουθιακά κυκλώματα.

Στις αρχές της δεκαετίας του 80 προστέθηκε επιπλέον κυκλωματική λογική σε κάθε έξοδο ενός PLD. Η νέα κυψελίδα εξόδου, που ονομάστηκε μακροκυψέλη (macrocell), περιείχε, εκτός από το flip-flop, πύλες λογικής και πολυπλέκτες. Επιπλέον, η ίδια η κυψελίδα ήταν προγραμματιζόμενη, δίνοντας τη δυνατότητα διαφορετικών τρόπων λειτουργίας. Επίσης, υπήρχε η δυνατότητα ανάδρασης από την έξοδο του κυκλώματος πίσω στον προγραμματιζόμενο πίνακα, προσδίδοντας μεγαλύτερη ευελιξία στον προγραμματισμό. Αυτή η νέα δομή του PLD ονομάστηκε generic PAL (GAL). Μια παρόμοια αρχιτεκτονική ήταν η PALCE (PAL CMOS erasable/programmable device).

Όλα αυτά τα τσιπς (PAL, PLA, registered PLD, GAL/PALCE) αναφέρονται με ην ονομασία SPLDs (Simple PLDs). Η GAL/PALCE τεχνική είναι η μόνη που εξακολουθεί να κατασκευάζεται ως αυτόνομο τσιπ.

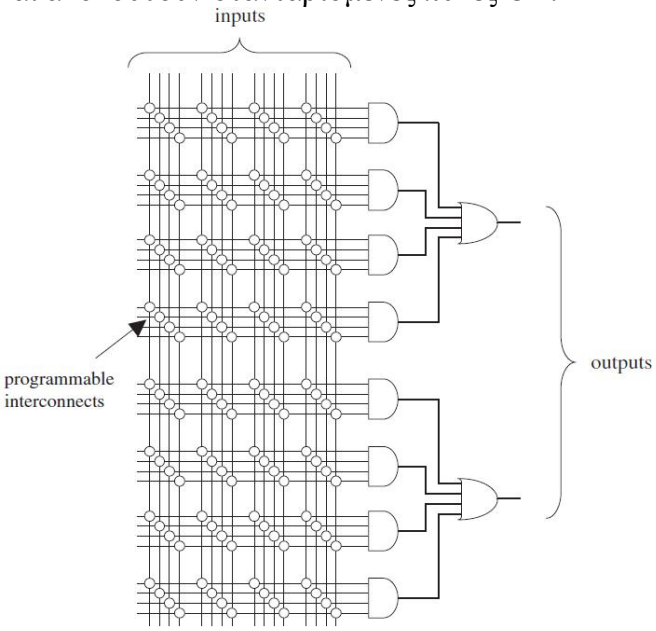
Στη συνέχεια, αρκετές GAL διατάξεις κατασκευάστηκαν στον ίδιο τσιπ, με περίπλοκα σχήματα καλωδίωσης-διασυνδέσεων, πιο εξελιγμένη τεχνολογία πυριτίου και διάφορα άλλα χαρακτηριστικά (όπως υποστήριξη JTAG για σύνδεση με άλλα στάνταρ λογικής). Η προσέγγιση αυτή έγινε γνωστή με την ονομασία CPLD (Complex PLD). Τα CPLDs είναι και σήμερα αρκετά δημοφιλή γιατί έχουν υψηλή πυκνότητα διατάξεων, υψηλή απόδοση και χαμηλό κόστος.

Στα μέσα της δεκαετίας του 80, έγινε η εισαγωγή των FPGAs (Field Programmable Gate Arrays). Τα FPGAs διαφέρουν από τα CPLDs στην αρχιτεκτονική, στην τεχνολογία, στις ενδογενείς λειτουργίες και στο κόστος. Στοχεύουν κυρίως στην υλοποίηση κυκλωμάτων μεγάλου μεγέθους και υψηλής απόδοσης.

Όλα τα PLDs (απλά ή περίπλοκα) είναι μόνιμου προγραμματισμού (non-volatile).Μπορούν να προγραμματιστούν μια φορά (OTP = one time programming), δηλαδή χρησιμοποιούν fuses ή antifuses ή μπορούν να προγραμματιστούν πολλές φορές με χρήση EEPROM ή μνήμης Flash. Τα FPGAs είναι μη-μόνιμου προγραμματισμού (volatile), γιατί χρησιμοποιούν SRAM για να αποθηκεύσουν τις συνδεσμολογία, οπότε χρειάζονται μια ROM διαμόρφωσης για να φορτώσουν τη συνδεσμολογία με το που τροφοδοτούνται με ρεύμα. Η ROM βέβαια χρησιμοποιεί fuse ή antifuse τεχνολογία.

## Διατάξεις PAL

Οι διατάξεις PAL (Progammable Array Logic) εισήχθησαν στην αγορά από την Monolithic Memories στα μέσα της δεκαετίας του 70. Η βασική τους αρχιτεκτονική φαίνεται στην επόμενη εικόνα. Οι μικροί κύκλοι παριστάνουν προγραμματιζόμενες συνδέσεις. Όπως βλέπουμε, το κύκλωμα αποτελείται από προγραμματιζόμενους πίνακες από πύλες AND και ακολουθούν στανταρισμένες πύλες OR.




Η υλοποίηση αυτής της αρχιτεκτονικής βασίζεται στο ότι κάθε συνδυαστική συνάρτηση μπορεί να αναπαρασταθεί ως άθροισμα – γινομένων (SOP = Sum Of Products),δηλαδή ως άθροισμα κατάλληλων ελαχιστόρων της συνάρτησης. Οπότε, οι πύλες AND χρησιμοποιούνται για τη δημιουργία των ελαχιστόρων και οι πύλες OR για τη συλλογή τους σε άθροισμα.

Ο κύριος περιορισμός αυτής της μεθοδολογίας, είναι ότι επιτρέπει την υλοποίηση μόνο συνδυαστικών κυκλωμάτων. Ωστόσο, προς το τέλος της δεκαετίας του 70, κατασκευάστηκαν οι registered PALs. Αυτές περιελάμβαναν ένα flip-flop σε κάθε έξοδο (δηλαδή μετά την κάθε πύλη OR), οπότε επέτρεπαν την υλοποίηση και ακολουθιακών συναρτήσεων (βέβαια των πολύ απλών).

Ένα παράδειγμα των τότε δημοφιλών PAL, ήταν το τσιπ PAL16L8. Αυτό περιείχε 16 εισόδου και 8 εξόδους και βέβαια είσοδο για VCC και GND. Το αντίστοιχο με καταχωρητές είχε την ονομασία 16R8.

Η τεχνολογία που υλοποιούσε αυτές τις διατάξεις ήταν η διπολική με τάση παροχής τα 5 και κατανάλωση ρεύματος (με ανοιχτές εξόδους) γύρω στα 200mA. Η μέγιστη συχνότητα λειτουργίας ήταν γύρω στα 100MHz και οι προγραμματιζόμενες κυψέλες όταν τύπου PROM (δίκτυα ασφαλειών (fuses)) ή EPROM.