2. ΒΑΣΙΚΕΣ ΠΥΛΕΣ ΨΗΦΙΑΚΗΣ ΛΟΓΙΚΗΣ

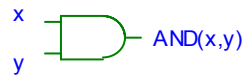
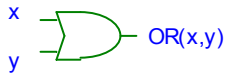
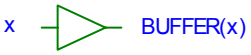
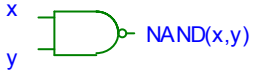
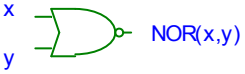
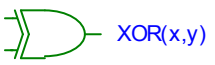
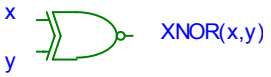
1. Τι σημαίνει «εξέταση της λειτουργίας» μιας ψηφιακής πύλης;

Βασικό στοιχείο στην εισαγωγή στα ψηφιακά ηλεκτρονικά, είναι η κατανόηση της λειτουργίας των λογικών πυλών.
Λέγοντας «εξέταση της λειτουργίας» εννοούμε την έξοδο που δίνουν οι πύλες συναρτήσει του σήματος ή των σημάτων εισόδου.

2. Τι είναι η «ταυτότητα» μιας ψηφιακής πύλης; Παραδείγματα

Η «ταυτότητα» κάθε λογικής πύλης είναι πέρα από το σύμβολό της και την αλγεβρική μορφή της συνάρτησης συμπεριφοράς της, ο πίνακας αληθείας της.
Στην

Εικόνα 3 φαίνονται τα σύμβολα, η συναρτησιακή μορφή και ο πίνακας αληθείας των βασικών ψηφιακών πυλών.

Ονομασία Πύλης	Σύμβολο
AND	
OR	
NOT	
BUFFER	
NAND	
NOR	
EXCLU-SIVE-OR (XOR)	
EQUIVA-LENCE , EXCLU-SIVE OR (XNOR)	

Εικόνα 1a. Οι βασικές πύλες και τα σύμβολά τους.

Ονομασία Πύλης	Αλγεβρική Συνάρτηση
AND	AND(x,y) = x y
OR	OR(x,y) = x + y
NOT	NOT(x) = x’
BUFFER	BUFFER(x) = x
NAND	NAND(x,y) = (x y)’
NOR	NOR(x,y) = (x + y)’
EXCLU-SIVE-OR (XOR)	XOR(x,y) = x’y + xy’ = x⊕y
EQUIVA-LENCE , EXCLU-SIVE OR (XNOR)	XNOR(x,y) = xy + x’y’ = (x⊕y)’

Εικόνα 2b. Οι βασικές πύλες και οι αλγεβρικές τους συναρτήσεις.

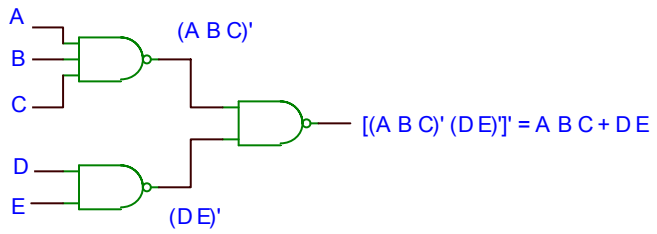
Ονομασία Πύλης	Πίνακας Αληθείας
AND	**AND** **x** **y** **AND(x,y)** 0 0 0 0 1 0 1 0 0 1 1 1
OR	**OR** **x** **y** **OR(x,y)** 0 0 0 0 1 1 1 0 1 1 1 1
NOT	**NOT** **x** **NOT(x)** 0 1 1 0
BUFFER	**BUFFER** **x** **BUFFER(x)** 0 0 1 1

NAND		NAND		
		x	y	NAND(x,y)
		0	0	1
		0	1	1
		1	0	1
		1	1	0
NOR		NOR		
		x	y	NOR(x,y)
		0	0	1
		0	1	0
		1	0	0
		1	1	0
EXCLU-SIVE-OR (XOR)		XOR		
		x	y	XOR(x,y)
		0	0	0
		0	1	1
		1	0	1
		1	1	0
EQUIVA-LENCE , EXCLU-SIVE OR (XNOR)		XNOR		
		x	y	XNOR(x,y)
		0	0	1
		0	1	0
		1	0	0
		1	1	1

Εικόνα 3c. Οι βασικές πύλες και οι πίνακες αληθείας τους.

3. Οι πύλες μπορούν να συνδεθούν μεταξύ τους και να δώσουν πιο σύνθετες πύλες

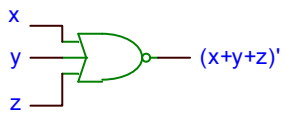
Οι βασικές πύλες μπορούν να συνδεθούν μεταξύ τους και να δώσουν πιο σύνθετες συναρτήσεις (βλέπε **Εικόνα 2**).



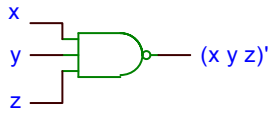
Εικόνα 2. Οι βασικές πύλες μπορούν να συνδεθούν μεταξύ τους και να δώσουν πιο σύνθετες συναρτήσεις.

4. Παραδείγματα ψηφιακών πυλών με τρεις εισόδους

Υπάρχουν περιπτώσεις πυλών με περισσότερες από 1 ή 2 εισόδους, όπως φαίνεται στην **Εικόνα 3(a,b).**



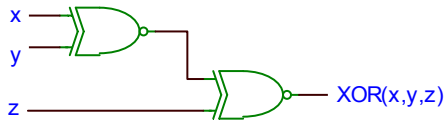
(a)



(b)

Εικόνα 3. Πύλες με 3 εισόδους.

Ο τρόπος δημιουργίας βασικών πυλών με περισσότερες εισόδους φαίνεται στην **Εικόνα 4**.



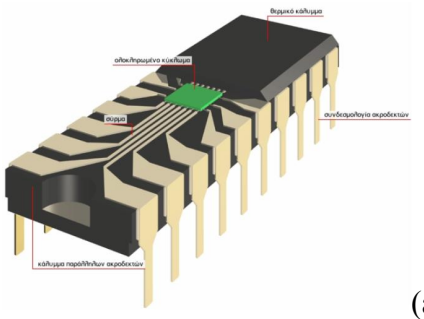
Εικόνα 4. Δημιουργία πύλης τριών εισόδων με πύλες δύο εισόδων.

5. Τι περιέχουν «μέσα τους» οι ψηφιακές πύλες;

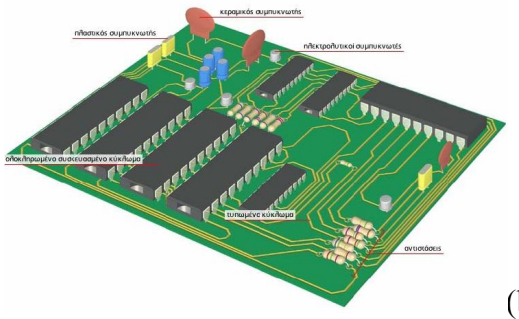
Οι πύλες κατασκευάζονται με τεχνικές ολοκλήρωσης κυκλωμάτων σε ψηφίδες πυριτίου. Στην **Εικόνα 5a** φαίνεται η δομή ενός ολοκληρωμένου κυκλώματος στην τελική μορφή του.

Ένα τυπωμένο κύκλωμα μπορεί να περιέχει πολλά ηλεκτρονικά στοιχεία τόσο διακριτά (π.χ. αντιστάσεις, πυκνωτές) όσο και ολοκληρωμένα κυκλώματα. (**Εικόνα 5b**).

Τα ολοκληρωμένα κυκλώματα (ΟΚ) των βασικών πυλών χωρίζονται σε διάφορες κατηγορίες ή οικογένειες ανάλογα με τον τρόπο κατασκευής τους, τα χαρακτηριστικά τους και τις λειτουργίες που επιτελούν.



(a)



(b)

Εικόνα 5. a) Δομή ενός ολοκληρωμένου κυκλώματος στην τελική μορφή του. b) Το τυπωμένο κύκλωμα μπορεί να περιέχει πολλά ηλεκτρονικά στοιχεία τόσο διακριτά (π.χ. αντιστάσεις, πυκνωτές) όσο και ολοκληρωμένα κυκλώματα.

6. Βασικές οικογένειες ψηφιακής λογικής σε μορφή Ολοκληρωμένου Κυκλώματος (ΟΚ) – Βασικές ονομασίες

Οι βασικές οικογένειες ψηφιακής λογικής ΟΚ φαίνονται στη συνέχεια**.**

Αρχικά Οικογένειας	**Εξήγηση Αρχικών**
RTL	**Resistor – Transistor Logic** (λογική αντίστασης – τρανζίστορ)
DTL	**Diode – Transistor Logic** (λογική διόδου – τρανζίστορ)
TTL	**Transistor – Transistor Logic** (λογική τρανζίστορ – τρανζίστορ)
ECL	**Emitter – Coupled Logic** (λογική σύζευξης εκπομπού)
MOS	**Metal-Oxide-Semiconductor** (λογική μετάλλου – οδειδίου – ημιαγωγού)
CMOS	**Complementary Metal-Oxide-Semiconductor** (συμπληρωματικής λογικής τρανζίστορ μετάλλου – οξειδίου – ημιαγωγού)

Πίνακας 1.Βασικές οικογένειες ψηφιακής λογικής.

Στη συνέχεια φαίνονται διάφορα παραδείγματα κωδικών ονομασιών βασικών ολοκληρωμένων κυκλωμάτων.

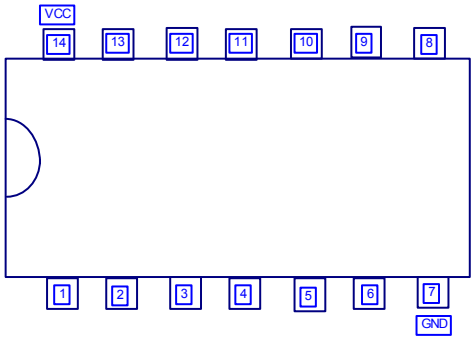
Ονομασία	**Οικογένεια Λογικής**
74LS00	Low-power Schottky TTL
74AL00	Advanced low-power Schottky TTL
74F00	Fast TTL
74HC00	High-Speed CMOS
74LVX00	Low – Voltage CMOS
74ABT00	Advanced BICMOS (TTL/CMOS hybrid)

Πίνακας 2. Παραδείγματα κωδικών ονομασιών βασικών ολοκληρωμένων κυκλωμάτων.

7. Ψηφιακές Πύλες - Τσίπς

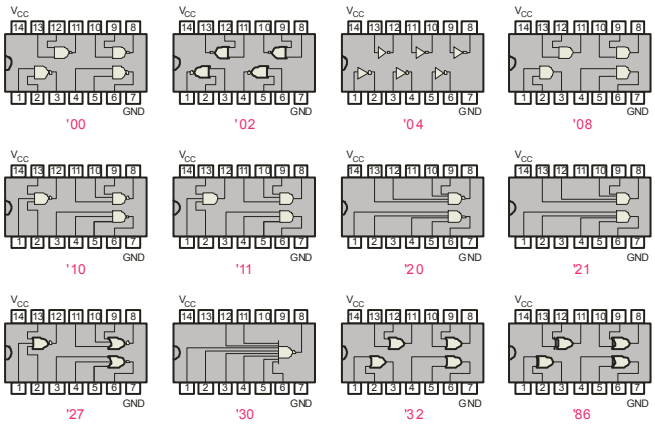
Κατά την εκτέλεση των εργαστηριακών ασκήσεων είναι χρήσιμο να έχετε δίπλα σας τη δομή των πυλών όπως συνδέονται εντός του **ολοκληρωμένου κυκλώματος** (**ΟΚ**) για την ταυτότητα των ακροδεκτών.

Τα ΟΚ ή αλλιώς «τσίπ» (chip), που χρησιμοποιούμε στο εργαστήριο Ψηφιακών Ηλεκτρονικών του Τμήματος Ηλεκτρονικών Μηχανικών του ΤΕΙ Αθήνας, είναι της σειράς **TTL** (**Transistor Transfer Logic**) και έχουν τη γενική τοπολογία που φαίνεται στην **Εικόνα 6**. Οι αριθμοί αντιστοιχούν στους ακροδέκτες (ποδαράκια) του τσίπ. Για να καταλάβουμε ποιος ακροδέκτης είναι ο [1], πρέπει να έχουμε την «εγκοπή» (το ημικύκλιο στην **Εικόνα 6**) στα αριστερά μας, όπως το κοιτάμε από πάνω. Ο ακροδέκτης [1], είναι τότε ο πρώτος κάτω αριστερά. Οι υπόλοιποι ακροδέκτες αριθμούνται στη σειρά μέχρι τον [7] στην κάτω σειρά και συνεχίζουν στην πάνω σειρά από τα δεξιά προς τα αριστερά, από τον [8] μέχρι τον [14]. Τα περισσότερα ΟΚ που θα χρησιμοποιήσουμε στην αρχή στο εργαστήριο θα έχουν 14 ακροδέκτες και πάντα ο [7] θα είναι η γείωση και ο [14] η τροφοδοσία. Στην **Εικόνα 6** έχουν ήδη σημειώθεί με GND και VCC αντίστοιχα. Είναι σημαντικό να έχουμε συνδέσει σωστά τη γείωση και την τροφοδοσία για την ορθή λειτουργία και την αποφυγή καταστροφής («κάψιμο») του ΟΚ.



Εικόνα 6. Η γενική τοπολογία των ΟΚ της σειράς TTL που θα χρησιμοποιήσουμε στο εργαστήριο.

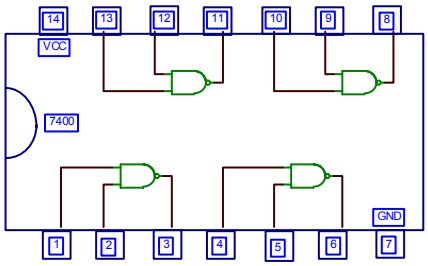
Γενικά, υπάρχουν ΟΚ με περισσότερους ακροδέκτες και διαφορετική τοπολογία, οπότε πρέπει πάντα να έχουμε κατά νου τη σωστή αρίθμηση και την κάτοψη του ΟΚ που δίνει ο κατασκευαστής, και ιδιαίτερα τους ακροδέκτες που αντιστοιχούν στη γείωση και στην τροφοδοσία του ΟΚ. Στην **Εικόνα 7** φαίνονται αναλυτικά ,παραδείγματα τοπολογίας ΟΚ με 14 ακροδέκτες, της σειράς TTL.



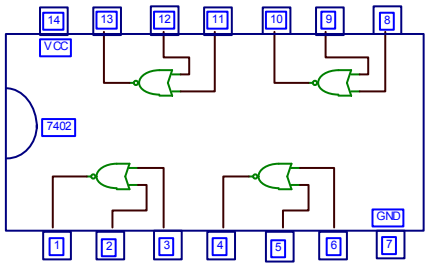
Εικόνα 7. Μερικά βασικά ΟΚ της σειράς TTL που θα χρησιμοποιήσουμε στο εργαστήριο.

Στην **Εικόνα 8** φαίνεται η τοπολογία των ακροδεκτών μερικών από τα βασικά ολοκληρωμένα λογικών πυλών της σειράς 7400, μαζί με τη λογική συνάρτηση που υλοποιούν.

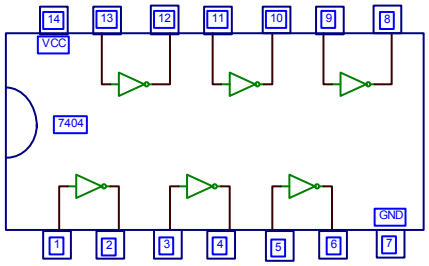
Κάθε ολοκληρωμένο κύκλωμα TTL, για να λειτουργεί χρειάζεται τη σωστή τάση (συνήθως σύνδεση του ακροδέκτη 14 με το λογικό 1 ή τα 5V) και τη γείωση (συνήθως σύνδεση του ακροδέκτη 7 με το λογικό 0).



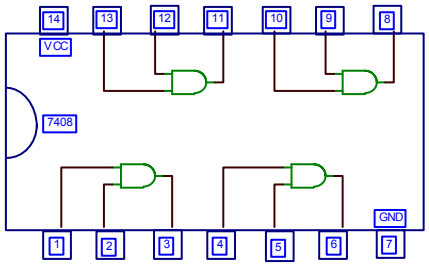
NAND(A,B) = (A*B)' = A' + B'



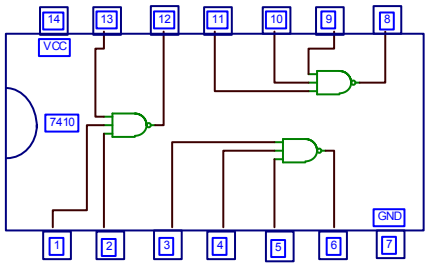
NOR(A,B) = (A+B)' = A' * B'



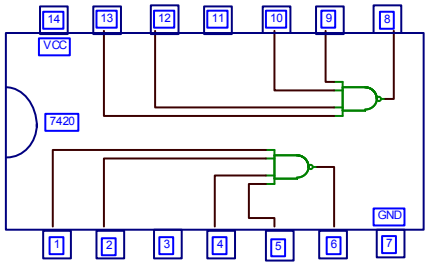
NOT(A) = A'



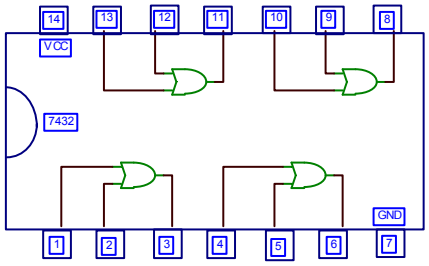
AND(A,B) = A*B



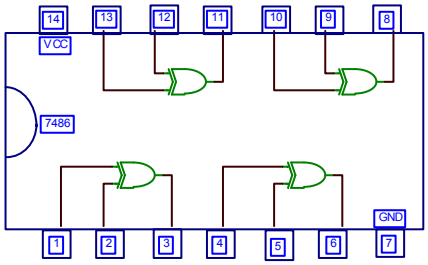
NAND(A,B,C) = (A*B*C)' = A'+B'+C'



NAND(A,B,C,D) = (A*B*C*D)' = A'+B'+C'+D'



OR(A,B) = A + B

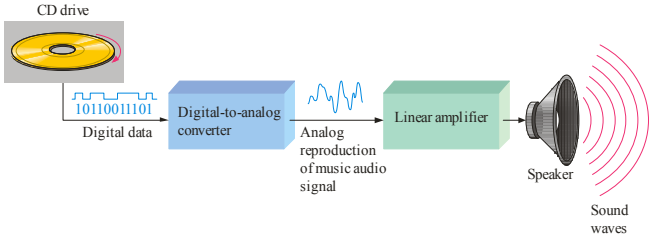


XOR(A,B) = A'B + AB'

Εικόνα 8. Τοπολογία των ακροδεκτών μερικών από τα βασικά ολοκληρωμένα λογικών πυλών της σειράς 7400 μαζί με τη λογική συνάρτηση που υλοποιούν.

8. Αρχή λειτουργίας ανάγνωσης από δίσκο ψηφιακών δεδομένων (CD)

Σε έναν ψηφιακό δίσκο τα δεδομένα είναι αποθηκευμένα σε ψηφιακή μορφή (**Εικόνα 9**). Πρακτικά πάνω στο πολυμερές (πλαστικό) υλικό του δίσκου είναι αποθηκευμένα τα δεδομένα με τη μορφή βαθουλωμάτων (εγκοπών). Ένα κύκλωμα μετατροπέα-ψηφιακού-σε- αναλογικό-σήμα, μετατρέπει την ψηφιακή αναπαράσταση των δεδομένων σε συνεχή αναλογική τιμή τάσης. Στην περίπτωση π.χ. που τα δεδομένα ήταν ένα αρχείο ήχου, μετά τη μετατροπή σε αναλογικό σήμα, μπορούμε να το ενισχύσουμε και να το στείλουμε σε ένα μεγάφωνο για αναπαραγωγή του ήχου.

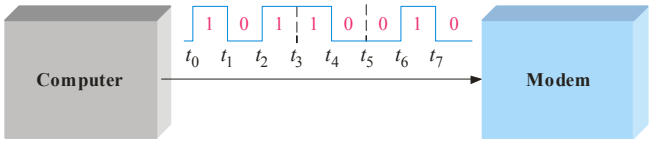


Εικόνα 9. Ποιοτική περιγραφή αναπαραγωγή ψηφιακού δίσκου.

Άσκηση. Περιγράψτε πως γίνεται η εγγραφή των δεδομένων σε έναν ψηφιακό δίσκο.

9. Σειριακή και παράλληλη ψηφιακή επικοινωνία

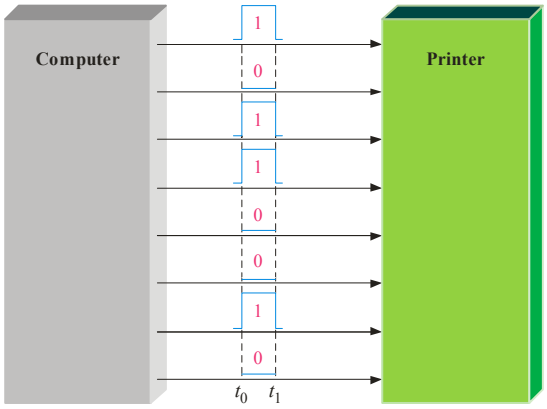
Στη σειριακή ψηφιακή επικοινωνία, τα δεδομένα μεταφέρονται το ένα μετά το άλλο (**Εικόνα 10**).



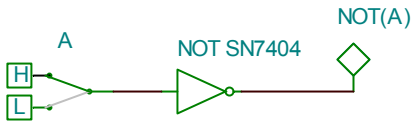
Εικόνα 10. Ποιοτική περιγραφή σειριακής ψηφιακής επικοινωνίας.

Στην παράλληλη ψηφιακή επικοινωνία η αντίστοιχη μεταφορά δεδομένων γίνεται παράλληλα (**Εικόνα 11**).

Μπορείτε να σκεφτείτε πλεονεκτήματα / μειονεκτήματα για τα δύο παραπάνω είδη επικοινωνίας;



Εικόνα 11. Ποιοτική περιγραφή παράλληλης ψηφιακής επικοινωνίας.

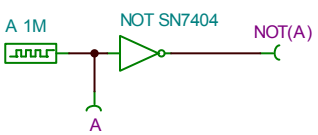


Εικόνα 12. Παράδειγμα προσομοίωσης της λειτουργίας μιας πύλης αντιστροφέα.

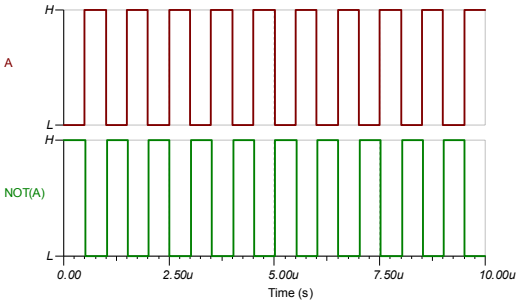
10. Παράδειγμα προσομοίωσης της λειτουργίας μιας πύλης αντιστροφέα

Στην **Εικόνα 12** παρουσιάζουμε ένα παράδειγμα προσομοίωσης της λειτουργίας μιας πύλης αντιστροφέα.. Χρησιμοποιούμε έναν διακόπτη (A), μια πύλη NOT SN7404 και έναν δείκτη λογικής NOT(A).

Στην **Εικόνα 13** φαίνεται η χρήση ρολογιού (A) συχνότητας 1MHz και ακροδεκτών ελέγχου (A, NOT(A)).



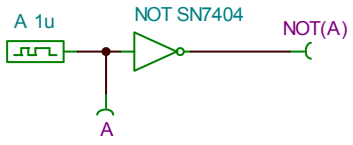
a)



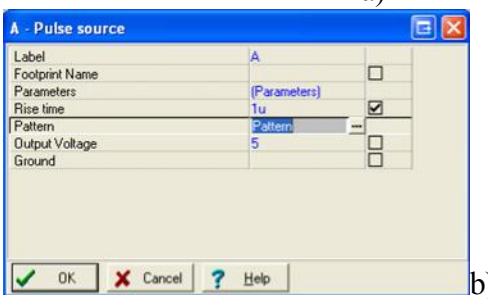
b)

Εικόνα 13. a) Η χρήση ρολογιού (A) συχνότητας 1MHz και ακροδεκτών ελέγχου (A, NOT(A)). b) Διάγραμμα χρονισμού.

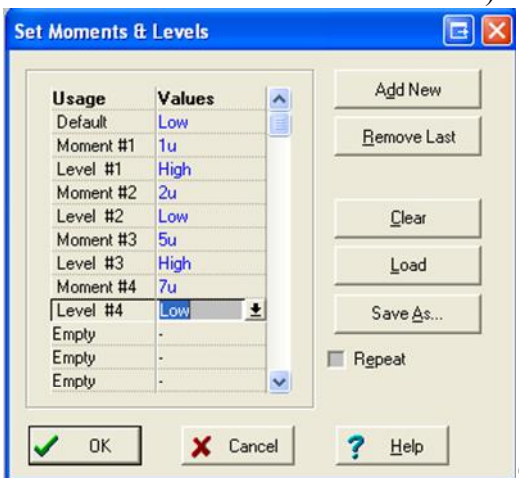
Στην **Εικόνα 14** φαίνεται η χρήση παλμογεννήτριας (A). Πρέπει να κάνουμε διπλό κλίκ πάνω στη γεννήτρια παλμών και να εισάγουμε το σχέδιο παλμών που επιθυμούμε. Επιλέγουμε Pattern με κλικ πάνω στις τρεις τελείες ... Εισάγουμε την ακολουθία παλμών πατώντας κάθε φορά Add New για να προσθέσουμε αλλαγή και γράφοντας την κατάλληλη χρονική στιγμή της αλλαγής (Το u αντιστοιχεί στο ελληνικό μ και σημαίνει micro-sec).



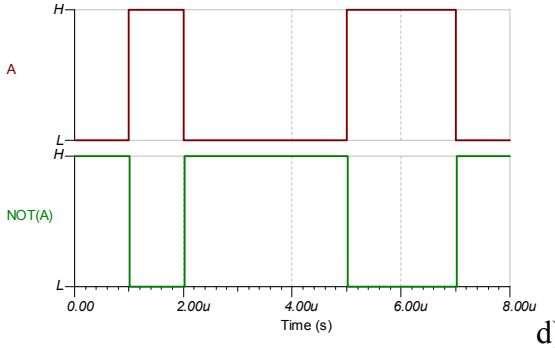
a)



b)



c)

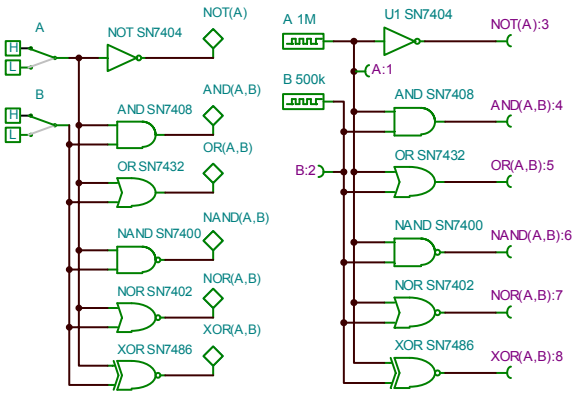


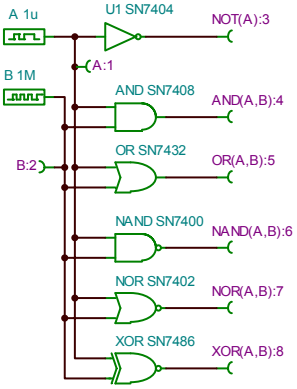
d)

Εικόνα 14. a) Χρήση γεννήτριας παλμών. b) Κάνουμε διπλό κλίκ πάνω στη γεννήτρια παλμών και να εισάγουμε το σχέδιο παλμών που επιθυμούμε. c) Εισάγουμε την ακολουθία παλμών πατώντας κάθε φορά Add New. d) Παράδειγμα διαγράμματος χρονισμού.

11. Ασκήσεις

1. **Εκτελέστε τις προσομοιώσεις των επόμενων κυκλωμάτων και επαληθεύστε την ορθότητα των αποτελεσμάτων σας με βάση τις τιμές του πίνακα αληθείας κάθε πύλης.**





Εικόνα 15.

2. **Σχεδιάστε τα περιεχόμενα των ολοκληρωμένων κυκλωμάτων και αριθμήστε τους ακροδέκτες τους.**
3.

Ονομασία Πύλης	**Κωδικός Ολοκληρωμένου TTL**	**Διάγραμμα σύνδεσης ολοκληρωμένου κυκλώματος**

AND		
OR		
NOT		
NAND		
NOR		
NOT		

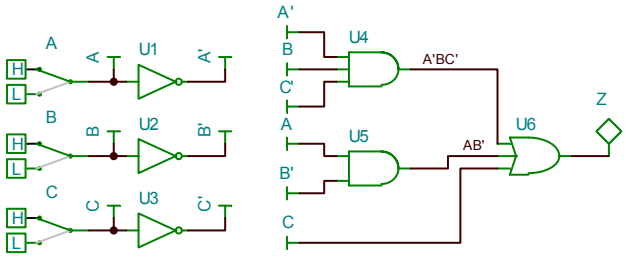
4. Εξακριβώστε την ορθή λειτουργία όλων των πυλών που περιέχονται στα ολοκληρωμένα 74LS08, 74LS32 και 74LS04 πάνω στο PENCIL BOX logic designer (PBLD) (Βλέπε Παράρτημα – 2).

Ονομασία Πύλης	**Κωδικός Ολοκληρωμένου TTL**	**Διάγραμμα σύνδεσης ολοκληρωμένου κυκλώματος**
AND		
OR		
NOT		
NAND		
NOR		
NOT		

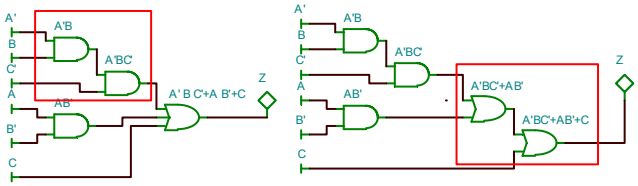
5. (a) Να σχεδιαστεί και να προσομοιωθεί το κύκλωμα της λογικής συνάρτησης Z = A' B C' + A B' + C με λογική σχεδίασης AND-OR. (b) Να σημειωθούν όλα τα άκρα των πυλών που θα χρησιμοποιηθούν. (c) Πόσα και ποια IC απαιτούνται για την πραγματοποίηση της σχεδίασης; (d) Υλοποιήστε το στο PENCIL BOX.

Λύση.

Η σχεδίαση και η προσομοίωση φαίνεται στη συνέχεια.

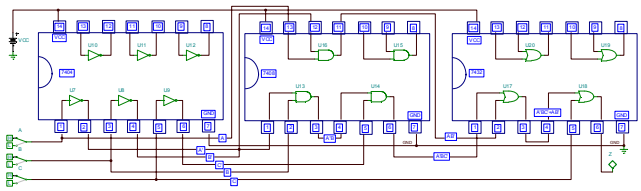


Εικόνα 16. Z = A' B C' + A B' + C με λογική σχεδίασης AND-OR.



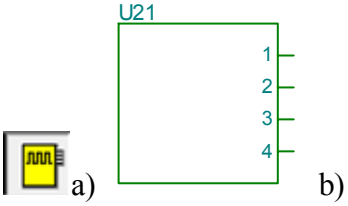
Εικόνα 17. Μπορούμε να υλοποιήσουμε την AND τριών εισόδων με δύο AND δύο εισόδων και αντίστοιχα την OR τριών εισόδων με δύο OR δύο εισόδων.

Μπορούμε να υλοποιήσουμε την AND τριών εισόδων με δύο AND δύο εισόδων και αντίστοιχα την OR τριών εισόδων με δύο OR δύο εισόδων, όπως φαίνεται στην **Εικόνα 17**. Άρα χρειαζόμαστε τρία OK με τη συνδεσμολογία που φαίνεται στην **Εικόνα 18**.



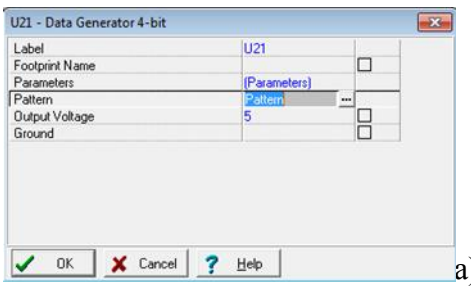
Εικόνα 18. Συνδεσμολογία με OK.

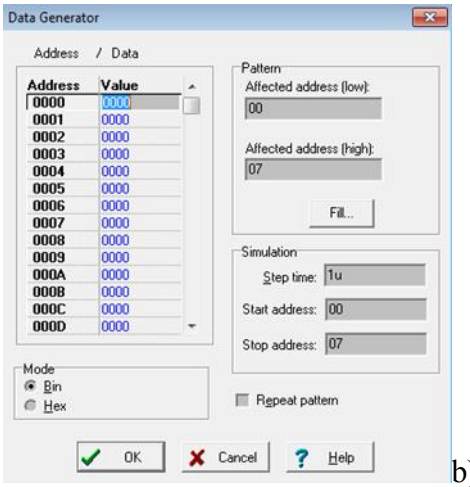
Για μια προσομοίωση, χωρίς να βάλουμε με το χέρι όλους τους συνδυασμούς μέσω των διακοπτών A, B, C, μπορούμε να χρησιμοποιήσουμε τη **γεννήτρια 4bit** (**Εικόνα 19**) που βρίσκεται στην παλέτα **Sources**.



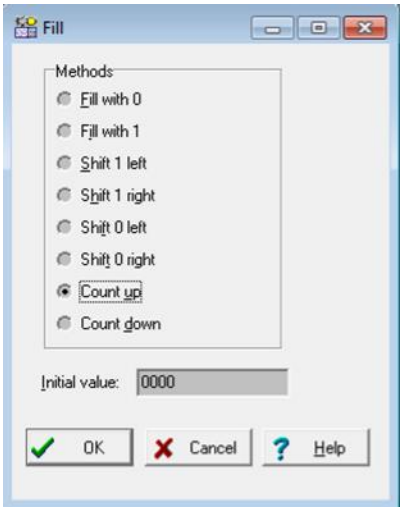
Εικόνα 19. a) Γεννήτρια 4bits από την παλέτα Sources. b) Γεννήτρια 4bit (σχηματικό).

Τοποθετούμε ένα αντίγραφό της στην επιφάνεια σχεδίασης του TINA και κάνουμε διπλό κλικ πάνω της. Στη συνέχεια επιλέγουμε **Pattern** πατώντας πάνω στις τελίτσες (**...**). Στο νέο πλαίσιο διαλόγου που προκύπτει σος **Affected address (low)** βάζουμε 00 και στο **Affected address (high)** βάζουμε 07 (γιατί με τρεις μεταβλητές έχουμε 8 περιπτώσεις από το 000 μέχρι το 111). Παρατηρούμε ότι στο **Simulation** το **Step time** είναι 1u δηλαδή 1μs ανά βήμα. Πατάμε το κουμπί **Fill** και στη συνέχεια τσεκάρουμε το Count Up και πατάμε **OK**. Επιστρέφοντας στο προηγούμενο πλαίσιο διαλόγου βλέπουμε ότι οι διευθύνεις έχουν γεμίσει με τους συνδυασμούς 000 μέχρι 111.

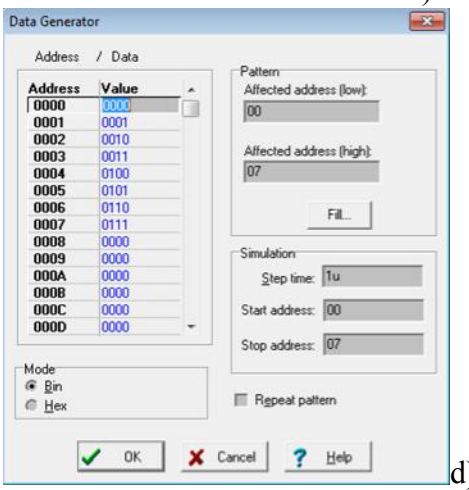




b)



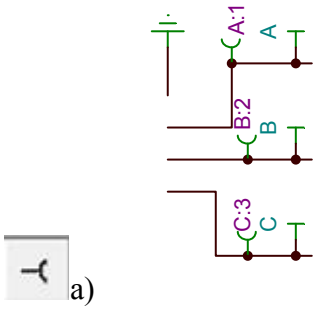
c)



d)

Εικόνα 20. a) Γεννήτρια 4bit. b) Εισαγωγή δεδομένων. c) Επιλογές για αυτόματο γέμισμα διευθύνσεων. d) Μετά το αυτόματο γέμισμα.

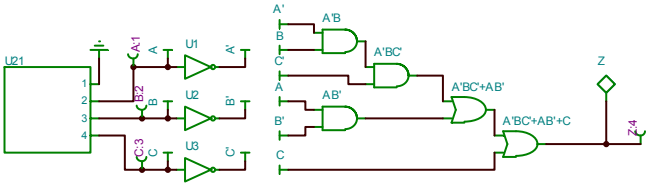
Για να δουμε τα αποτελέσματα της προσομοίωσης σε διάγραμμα χρονισμού, πρέπει στο κύκλωμα μας να βάλουμε σημεία-εξόδου (**voltage pins**). Πρόκειται για σημεία μέτρησης της τάσης. Το TINA έχει ειδικό εργαλείο για αυτή τη δουλειά στην παλέτα **Meters**. Τοποθετούμε τα voltage pins όπου θέλουμε να καταγράψουμε τιμές τάσης. Με αρίθμηση, με άνω-κάτω τελεία και αριθμό, μετά το όνομα του voltage pin επιλέγουμε τη σειρά με την οποία θα εμφανίζονται τα αποτελέσματα στο διάγραμμα χρονισμού (**Εικόνα 21**).



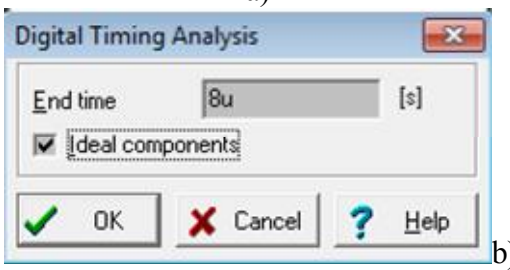
a) b)

Εικόνα 21. a) Voltage pin απο την παλέτα Meters. b) Σημεία στο κύκλωμα που βάλμα το Voltage pin. Με τις αριθμήσεις A:1, B:2, C:3, τα αποτελέσματα της χρονικής ανάλυσης θα προκύψουν με τη σειρά που θέλουμε.

Το τελικό κύκλωμά μας με τις προσθήκες και τις αλλαγές είναι έτοιμο. Για να πάρουμε το διάγραμμα χρονισμού επιλέγουμε **Analysis ➔ Digital Timing Analysis** και εισάγουμε 8u για χρόνο προσομοίωσης (**Εικόνα 22**).



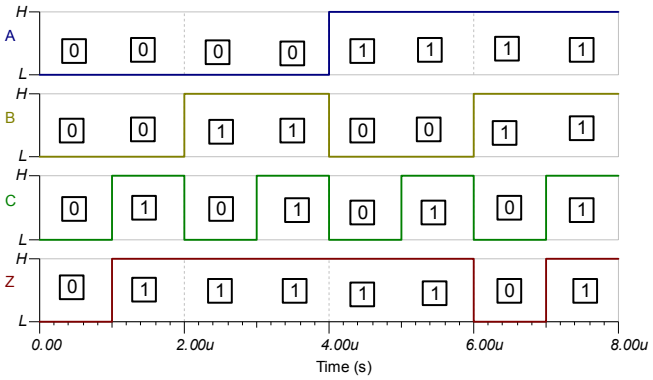
a)



b)

Εικόνα 22. a) Το τελικό κύκλωμα έτοιμο για προσομοίωση. b) Με Analysis ➔ Digital Timing Analysis επιλέγουμε τον τελικό χρόνο προσομοίωσης και τσεκάρουμε Ideal components για να μη ληφθούν υπόψη καθυστερήσεις στη λειτουργία των πυλών.

Το διάγραμμα χρονισμού που προκύπτει φαίνεται στη συνέχεια (**Εικόνα 23**).

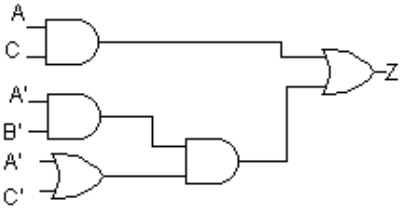


Εικόνα 23. Το αποτέλεσμα της χρονικής ανάλυσης.

Από το διάγραμμα χρονισμού προκύπτει πολύ εύκολα ο πίνακας αληθείας του κυκλώματος που φαίνεται στη συνέχεια.

A	B	C	Z
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

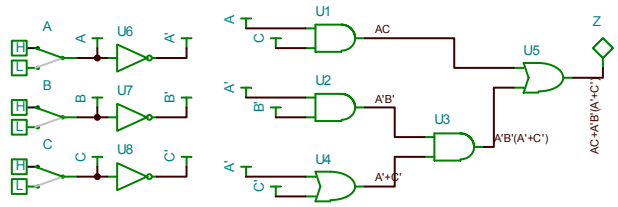
6. (a) Να βρεθεί η έξοδος του λογικού κυκλώματος. (b) Να σημειωθούν οι έξοδοι των πυλών και η τελική έκφραση της Z. (c) Να γίνει προσομοίωση της λειτουργίας του. (d) Πόσα και ποια IC απαιτούνται για την πραγματοποίηση της σχεδίασης;



Εικόνα 24. Κύκλωμα εργασίας 5.

Λύση.

Στην επόμενη εικόνα φαίνεται η υλοποίηση του κυκλώματος στο TINA και οι έξοδοι των πυλών.



Εικόνα 25. Η υλοποίηση του κυκλώματος στο TINA.

Παρατηρούμε ότι:

$Z = AC + A'B'(A'+C') = AC + A'B'A' + A'B'C'$

Επειδή $xy = yx$ και $xx=x$ θα είναι $A'B'A' = A'A'B' = A'B'$, οπότε:

$Z = AC + A'B' + A'B'C' = AC + A'B'(1+C')$

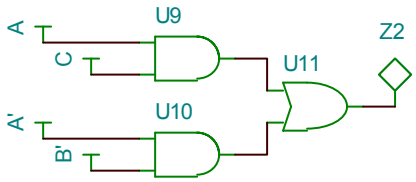
Επειδή $1+x=1$ και $1x=x$ προκύπτει:

$Z = AC + A'B'(1+C') = AC + A'B'1 = AC + A'B'$

Δηλαδή:

$Z = AC + A'B'(A'+C') = AC + A'B'$

Αυτό σημαίνει ότι το αρχικό κύκλωμα, είναι ισοδύναμο με το επόμενο (**Εικόνα 26**).



Εικόνα 26. Τελικό απλοποιημένο κύκλωμα.

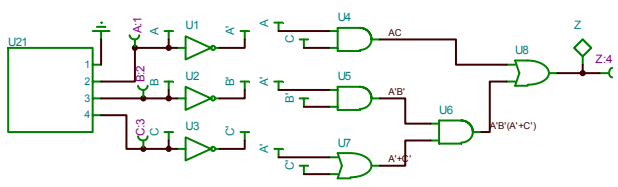
Ισοδύναμο, σημαίνει ότι έχουν ακριβώς τον ίδιο πίνακα αληθείας, δηλαδή εκτελούν ακριβώς τις ίδιες λειτουργίες. Μπορούμε να προσομοιώσουμε τη λειτουργία τους στο TINA για όλες τις τιμές και συνδυασμούς τιμών εισόδων A, B, C και να επιβεβαιώσουμε την ισοδυναμία τους.

Εναλλακτικά μπορούμε να επιβεβαιώσουμε την ισοδυναμία των δύο κυκλωμάτων και μέσω του πίνακα αληθείας. Στη συνέχεια φαίνεται ένας αναλυτικός πίνακας αληθείας που περιέχει τις μεταβλητές εισόδου στα αριστερά και τις εξόδους Z και Z2 στα δεξιά. Η Z = Z2 αν τα αντιστοιχα στοιχεία τους είναι ίσα σε κάθε γραμμή. Προσθέτουμε και τις βοηθητικές τιμές A', B', C', AC, A'B', A'+C', A'B'(A'+C') μεταξύ των μεταβλητών εισόδου και των εξόδων. Φροντίζουμε κάθε βοηθητική στήλη να έχει στα αριστερά τις τις μεταβλητές που χρειάζεται. Π.χ. η AC έχει στα αριστερά της την A και τη C και έχει 1 εκεί που και η A και η C είναι 1.

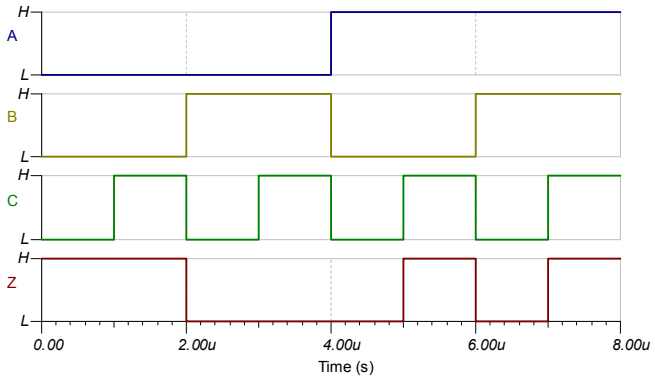
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
A	B	C	A'	B'	C'	AC	A'B'	A'+C'
0	0	0	1	1	1	0	1	1
0	0	1	1	1	0	0	1	1
0	1	0	1	0	1	0	0	1
0	1	1	1	0	0	0	0	1
1	0	0	0	1	1	0	0	1
1	0	1	0	1	0	1	0	0
1	1	0	0	0	1	0	0	1
1	1	1	0	0	0	1	0	0

(10)=(8)(9)	(11)=(7)+(10)	(12)=(7)+(8)
A'B'(A'+C')	Z	Z2
1	1	1
1	1	1

0	0	0
0	0	0
0	0	0
0	1	1
0	0	0
0	1	1

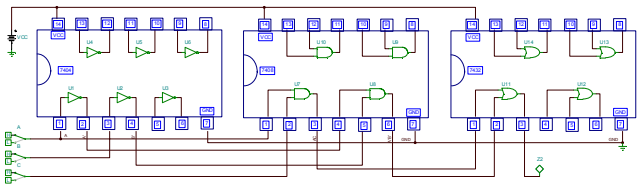


Εικόνα 27. Το κύκλωμα έτοιμο για προσομοίωση χρονισμού.



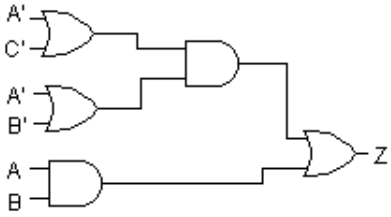
Εικόνα 28. Το διάγραμμα χρονισμού.

Με την απλοποίηση παρατηρούμε ότι χρειαζόμαστε τα τρία ολοκληρωμένα που φαίνονται στην **Εικόνα 29**. Χρειαστήκαμε δύο NOT, δύο πύλες AND και μία πύλη OR.



Εικόνα 29. Υλοποίηση με OK.

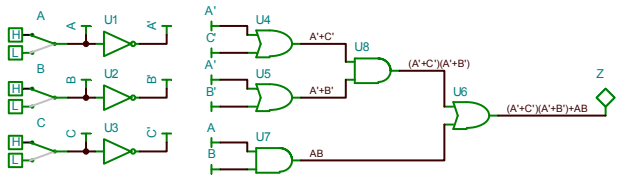
7. (a) Να βρεθεί η έξοδος του λογικού κυκλώματος. (b) Να σημειωθούν οι έξοδοι των πυλών και η τελική έκφραση της Z. (c) Να γίνει προσομοίωση της λειτουργίας του. (d) Πόσα και ποια IC απαιτούνται για την πραγματοποίηση της σχεδίασης; (ε) Υλοποιήστε το στο PENCIL BOX.



Εικόνα 30. Κύκλωμα εργασίας 6.

Λύση.

Υλοποιήσαμε το κύκλωμα στο TINA (**Εικόνα 31**).

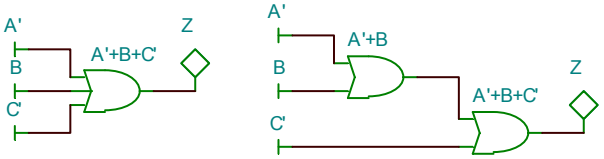


Εικόνα 31. Σχεδιασμός του κυκλώματος στο TINA.

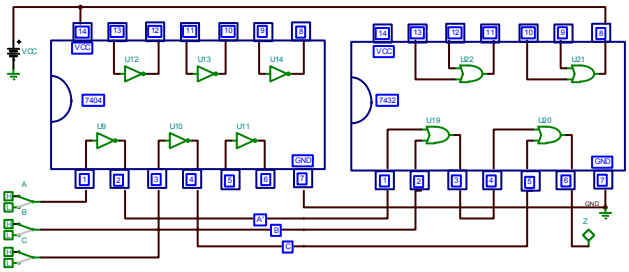
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
A	**B**	**C**	**A'**	**B'**	**C'**	**A'+C'**	**A'+B'**
0	0	0	1	1	1	1	1
0	0	1	1	1	0	1	1
0	1	0	1	0	1	1	1
0	1	1	1	0	0	1	1
1	0	0	0	1	1	1	1
1	0	1	0	1	0	0	1
1	1	0	0	0	1	1	0
1	1	1	0	0	0	0	0

(9)	(10)=(7)(8)	(11)=(9)+(10)	(12)=(4)+(2)+(6)
AB	**(A'+C')(A'+B')**	**(A'+C') (A'+B')+ AB**	**(A'+B+C')**
0	1	1	1
0	1	1	1
0	1	1	1
0	1	1	1
0	1	1	1
0	0	0	0
1	0	1	1
1	0	1	1

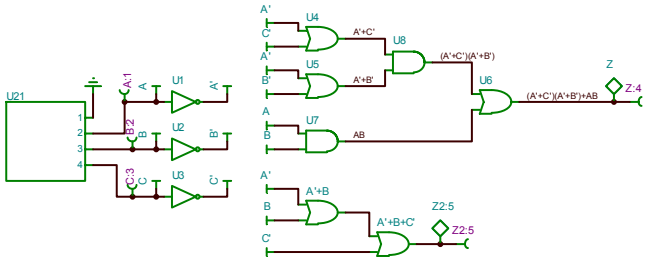
Παρατηρώ ότι Z = A'+B+C'. Δηλαδή η υλοποίηση μπορεί να γίνει δύο NOT και με μια πύλη OR τριών εισόδων ή δύο πύλες OR δύο εισόδων, όπως φαίνεται στην **Εικόνα 32**.



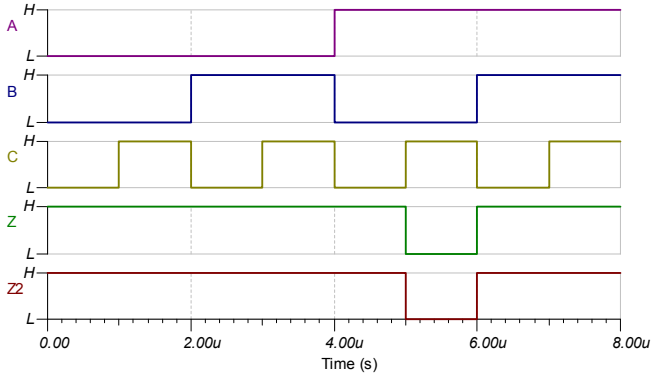
Εικόνα 32. Απλοποιήσεις.



Εικόνα 33. Υλοποίηση του κυκλώματος με OK.



Εικόνα 34. Κύκλωμα έτοιμο για διάγραμμα χρονισμού. Θα γίνει σύγκριση μεταξύ του αρχικού και του απλοποιημένου κυκλώματος.



Εικόνα 35. Διάγραμμα χρονισμού.

Παρατηρούμε την ταύτιση μεταξύ Z και Z2.

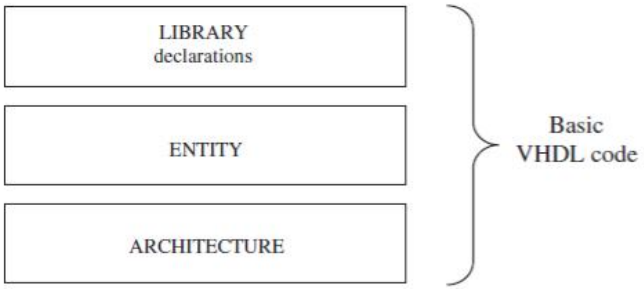
VHDL περιγραφή βασικών πυλών ψηφιακής λογικής

Η VHDL γλώσσα περιγραφής υλικού. Ο κώδικας περιγράφει τη συμπεριφορά ή τη δομή ενός ηλεκτρονικού κυκλώματος από το οποίο ένα συμβατό φυσικό κύκλωμα μπορεί να συμπεριληφθεί σε ένα μεταγλωττιστή. Η VHDL επιτρέπει τη σύνθεση κυκλωμάτων καθώς επίσης και τη προσομοίωση τους

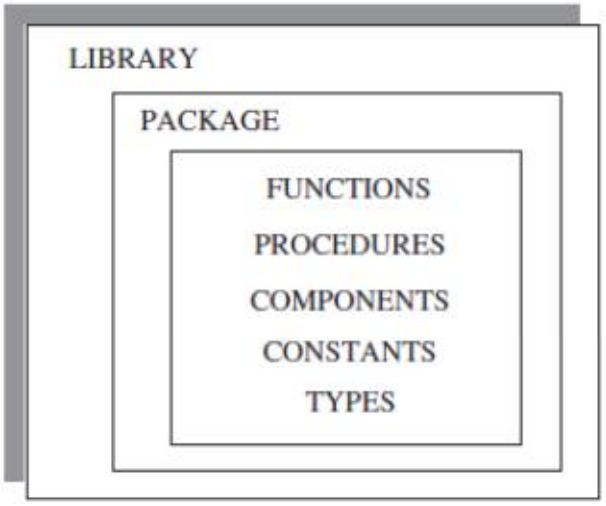
Οι κύριες εφαρμογές της περιλαμβάνουν τη σύνθεση ψηφιακών κυκλωμάτων πάνω σε CPLD/FPGA ολοκληρωμένα κυκλώματα και τον σχεδιασμό ή τη παραγωγή μάσκας για κατασκευή ASIC (Application Specific Integrated Circuit).

Η βασικότερη δομή ενός κώδικα VHDL περιλαμβάνει τις δηλώσεις βιβλιοθήκης, τη δήλωση της οντότητας και τη δήλωση της αρχιτεκτονικής, όπως φαίνεται στην **Εικόνα 36**.

Οι δηλώσεις βιβλιοθήκης είναι μία λίστα από όλες τις βιβλιοθήκες των αντίστοιχων πακέτων που χρειάζεται ο μεταγλωττιστής για να προχωρήσει στη διαδικασία σχεδιασμού. Η βασική δομή μιας βιβλιοθήκης φαίνεται στην **Εικόνα 37**.



Εικόνα 36. Βασική δομή κώδικα VHDL.



Εικόνα 37. Βασική δομή βιβλιοθήκης στη VHDL.

Η βιβλιοθήκη **std** περιέχει ορισμούς για τους βασικούς τύπους δεδομένων (**BIT**, **BOOLEAN**, **INTEGER**, **BIT_VECTOR** κ.τ.λ.) καθώς και πληροφορίες για το κείμενο και το χειρισμό αρχείων.

Η βιβλιοθήκη **work** είναι ο προορισμός όπου αποθηκεύονται τα αρχεία σχεδιασμού.

Ένα πακέτο που συχνά χρειάζεται να συμπεριληφθεί σε αυτή τη λίστα είναι το **std_logic_1164** από την βιβλιοθήκη **ieee**, το οποίο ορίζει τον τύπο **STD_LOGIC**. Το κύριο πλεονέκτημα του **STD_LOGIC** έναντι του **BIT** είναι ότι επιτρέπει την υψηλή εμπέδηση (‘Z’) και τις προδιαγραφές για τις αδιάφορες καταστάσεις (‘-’).

Για να δηλωθεί το παραπάνω πακέτο (ή οτιδήποτε άλλο) χρειάζονται δύο γραμμές κώδικα, μία για να δηλώσουμε τη βιβλιοθήκη και μία που να χρησιμοποιεί την εντολή **use**, που δείχνει προς το συγκεκριμένο πακέτο μέσα στη βιβλιοθήκη, όπως απεικονίζεται παρακάτω.

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

Η οντότητα είναι μία λίστα με τις προδιαγραφές όλων των θυρών εισόδου/εξόδου του κυκλώματος. Περιέχει επίσης τις ενδογενείς (**Generic**) παραμέτρους, καθώς και τις δηλώσεις για υποπρογράμματα και ταυτόχρονες δηλώσεις. Η σύνταξη της φαίνεται στη συνέχεια.

```
ENTITY entity_name IS
GENERIC
(constant_name: constant_type := constant_value;
...);
PORT
(port_name: signal_mode signal_type;
...);
[declarative part]
[BEGIN]
statement part]
END entity_name;
```

Η λέξη **GENERIC** επιτρέπει τη δήλωση των γενικών σταθερών, οι οποίες μπορούν να χρησιμοποιηθούν οπουδήποτε μέσα στο κώδικα. Παράδειγμα:

```
GENERIC (number_of_bits : INTEGER : = 16);
```

PORT, signal mode: IN, OUT, INOUT, BUFFER. Τα δύο πρώτα σήματα είναι μίας κατεύθυνσης, το τρίτο είναι αμφίδρομο και το τέταρτο χρειάζεται όταν ένα σήμα εξόδου πρέπει να διαβαστεί εσωτερικά.

PORT, signal type: BIT, BIT_VECTOR, INTEGER, STD_LOGIC, STD_LOGIC_ VECTOR, BOOLEAN, κ.τ.λ.

Declarative part: Μπορεί να περιέχει δηλώσεις των **TYPE, SUBTYPE, CONSTANT, SIGNAL, FILE, ALIAS, USE** και **ATTRIBUTE**, συν τα σώματα των **FUNCTION** και **PROCEDURE**. Σπάνια χρησιμοποιούνται με αυτόν τον τρόπο.

Statement part: Μπορεί να περιέχει ταυτόχρονες δηλώσεις (σπάνια χρησιμοποιείται).

Η αρχιτεκτονική περιέχει τον κατάλληλο κώδικα (συμπεριφοράς ή δομικού χαρακτήρα) που προορίζεται για τη περιγραφή του κυκλώματος. Μπορεί να είναι είτε **ταυτόχρονος** είτε **διαδοχικός** κώδικας. Η πρώτη περίπτωση είναι κατάλληλη για το σχεδιασμό των **συνδυαστικών κυκλωμάτων** ενώ η δεύτερη περίπτωση μπορεί να χρησιμοποιηθεί για το σχεδιασμό και των **ακολουθιακών κυκλωμάτων**. Η σύνταξή της φαίνεται παρακάτω.

```
ARCHITECTURE architecture_name OF entity_name IS
   [declarative part]
BEGIN
    (code)
END architecture_name;
```

Declarative part: Μπορεί να περιέχει τα ίδια στοιχεία με το τμήμα δήλωσης της οντότητας, συν τις δηλώσεις **COMPONENT** και **CONFIGURATION**.

Code: Μπορεί να είναι **ταυτόχρονος**, **διαδοχικός** ή και **μικτός**. Για να είναι διαδοχικός οι δηλώσεις πρέπει να τοποθετούνται μέσα σε μία **PROCESS**. Ωστόσο σαν σύνολο ο VHDL κώδικας είναι πάντα **ταυτόχρονος** που σημαίνει ότι όλα τα μέρη του αντιμετωπίζονται ‘’ταυτόχρονα’’, δηλαδή χωρίς προτεραιότητα. Συνεπώς, οποιαδήποτε διαδικασία (**PROCESS**) συντάσσεται ταυτόχρονα με οποιεσδήποτε άλλες δηλώσεις που βρίσκονται έξω από αυτήν. Η μόνη άλλη επιλογή για την κατασκευή διαδοχικού κώδικα VHDL είναι μέσω υποπρογραμμάτων (**FUNCTION** και **PROCEDURE**).

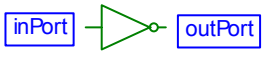
Για να γράψετε απλώς ταυτόχρονο κώδικα, μπορούμε να χρησιμοποιήσουμε **τελεστές** καθώς και δηλώσεις **WHEN** και **GENERATE**. Για να γράψουμε διαδοχικό κώδικα (μέσα σε μία **PROCESS**, **FUNCTION** ή **PROCEDURE**) οι επιτρεπόμενες δηλώσεις είναι οι **IF**, **CASE**, **LOOP** και **WAIT** (συν τους τελεστές).

Τα πιο πολλά από τα παραπάνω θα αναλυθούν στα επόμενα κεφάλαια. Προς το παρόν, στη συνέχεια θα δούμε εφαρμογή στην περιγραφή των βασικών ψηφιακών πυλών. Η προσομοίωση έγινε στο Sonata του SimiliEDA. Μαζί με κάθε περιγραφή πύλης στη VHDL παραθέτουμε και τον κώδικα ελέγχου της (testbench). Πρόκειται επίσης για VHDL κώδικα ο οποίος έχει μοναδικό σκοπό να ελέγξει την ορθή λειτουργία της κάθε πύλης.

Για προσομοίωση στο TINA, αρκεί μόνον ο κώδικας της οντότητας και της αρχιτεκτονικής. Ουσιαστικά, σε αυτή την περίπτωση, το testbench κομμάτι, είναι το να συνδέσουμε πηγές σημάτων ή διακόπτες και να εκτελέσουμε την προσομοίωση.

Πύλη NOT

```
--import std_logic from the IEEE library
library ieee;
use ieee.std_logic_1164.all;

--ENTITY DECLARATION:
--name, inputs, outputs
entity notGate is
   port( inPort  : in std_logic;
         outPort : out std_logic);
end notGate;
```



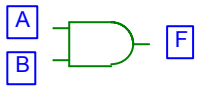
```
--FUNCTIONAL DESCRIPTION:
--how the Inverter works
architecture func of notGate is
begin
   outPort <= not inPort;
end func;
```

Test Bench:

```
--import std_logic from the IEEE library
library ieee;
use ieee.std_logic_1164.all;

--ENTITY DECLARATION:
--no inputs, no outputs
entity notGate_tb is
end notGate_tb;

-- Describe how to test the inverter
architecture tb of notGate_tb is
   --pass notGate entity to the
   --testbench as component
   component notGate is
      port( inPort  : in std_logic;
            outPort  : out std_logic);
   end component;

   signal  goIn, goOut : std_logic;
begin
   --map the testbench signals
   --to the ports of the notGate
   mapping: notGate port map( goIn,
goOut);

   process
      --variable to track errors
      variable errCnt : integer := 0;
   begin
      --TEST 1
      goIn <= '0';
      wait for 15 ns;
      assert(goOut = '1') report
"Error 1" severity error;
      if(goOut /= '1') then
         errCnt := errCnt + 1;
      end if;

      --TEST 2
      goIn <= '1';
      wait for 15 ns;
      assert(goOut = '0') report
"Error 2" severity error;
      if(goOut /= '0') then
         errCnt := errCnt + 1;
      end if;

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

   end process;
end tb;
---------------------------------------
configuration cfg_tb of notGate_tb is
   for tb
   end for;
end cfg_tb;
```

Πύλη AND

```
--import std_logic from the IEEE library
library ieee;
use ieee.std_logic_1164.all;

--ENTITY DECLARATION:
--name, inputs, outputs
entity andGate is
   port( A, B : in std_logic;
         F : out std_logic);
end andGate;
```




```
--FUNCTIONAL DESCRIPTION:
--how the AND Gate works
architecture func of andGate is
begin
  F <= A and B;
end func;
```

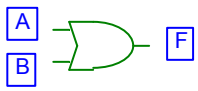
Test Bench:

```
--import std_logic from the IEEE library
library ieee;
use ieee.std_logic_1164.all;

--ENTITY DECLARATION:
--no inputs, no outputs
entity andGate_tb is
end andGate_tb;

-- Describe how to test the AND Gate
architecture tb of andGate_tb is
   --pass andGate entity to the
testbench as component
   component andGate is
   port( A, B : in std_logic;
         F : out std_logic);
   end component;

   signal  inA, inB, outF : std_logic;
begin
   --map the testbench signals to the
ports of the andGate
   mapping: andGate port map(inA, inB,
outF);

   process
      --variable to track errors
      variable errCnt : integer := 0;
   begin
      --TEST 1
```

```
      inA <= '0';
      inB <= '0';
      wait for 15 ns;
      assert(outF = '0')  report "Error
1"  severity error;
      if(outF /= '0') then
         errCnt := errCnt + 1;
      end if;

      --TEST 2
      inA <= '0';
      inB <= '1';
      wait for 15 ns;
      assert(outF = '0')  report "Error
2"  severity error;
      if(outF /= '0') then
         errCnt := errCnt + 1;
      end if;

      --TEST 3
      inA <= '1';
      inB <= '1';
      wait for 15 ns;
      assert(outF = '1')  report "Error
3"  severity error;
      if(outF /= '1') then
         errCnt := errCnt + 1;
      end if;

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

   end process;
end tb;
-----------------------------------------
----
configuration cfg_tb of andGate_tb is
   for tb
   end for;
end cfg_tb;
```

Πύλη OR

```
--import std_logic from the IEEE library
library ieee;
use ieee.std_logic_1164.all;

--ENTITY DECLARATION:
--name, inputs, outputs
entity orGate is
   port( A, B : in std_logic;
            F : out std_logic);
end orGate;
```




```
--FUNCTIONAL DESCRIPTION:
--how the OR Gate works
architecture func of orGate is
begin
   F <= A or B;
end func;
```

Test Bench:

```
--import std_logic from the IEEE library
library ieee;
use ieee.std_logic_1164.all;

--ENTITY DECLARATION:
--no inputs, no outputs
entity orGate_tb is
end orGate_tb;

-- Describe how to test the OR Gate
architecture tb of orGate_tb is
   --pass orGate entity
   --to the testbench as component
   component orGate is
   port( A, B : in std_logic;
            F : out std_logic);
   end component;

   signal  inA, inB, outF : std_logic;
begin
   --map the testbench signals
   --to the ports of the orGate
   mapping: orGate port map(inA, inB,
outF);

   process
      --variable to track errors
      variable errCnt : integer := 0;
   begin
      --TEST 1
      inA <= '0';
      inB <= '0';
      wait for 15 ns;
      assert(outF = '0')  report "Error
1"  severity error;
      if(outF /= '0') then
         errCnt := errCnt + 1;
      end if;

      --TEST 2
      inA <= '0';
      inB <= '1';
      wait for 15 ns;
      assert(outF = '1')  report "Error
2"  severity error;
      if(outF /= '1') then
         errCnt := errCnt + 1;
      end if;

      --TEST 3
      inA <= '1';
      inB <= '1';
```

```
      wait for 15 ns;
      assert(outF = '1')  report "Error
3"  severity error;
      if(outF /= '1') then
         errCnt := errCnt + 1;
      end if;

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

   end process;
end tb;
----------------------------------------
----
configuration cfg_tb of orGate_tb is
   for tb
   end for;
end cfg_tb;
```