ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΔΙΑΤΜΗΜΑΤΙΚΟ ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΠΛΗΡΟΦΟΡΙΚΗ ΚΑΙ ΥΠΟΛΟΓΙΣΤΙΚΗ ΒΙΟΙΑΤΡΙΚΗ

Ευριστικές προσεγγίσεις στο πρόβλημα νομιμοποίησης χωροθετημένων ολοκληρωμένων κυκλωμάτων

Γώγου Μαρία

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ
Επιβλέπων
Δαδαλιάρης Αντώνιος

Λαμία, 2018

UNIVERSITY OF THESSALY

SCHOOL OF SCIENCE

INFORMATICS AND COMPUTATIONAL BIOMEDICINE

Heuristic approaches for the legalization problem of integrated circuits

Gogou Maria

Master thesis

Dadaliaris Antonios

Lamia

2018

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΔΙΑΤΜΗΜΑΤΙΚΟ ΜΕΤΑΠΤΥΧΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΠΛΗΡΟΦΟΡΙΚΗ ΚΑΙ ΥΠΟΛΟΓΙΣΤΙΚΗ ΒΙΟΙΑΤΡΙΚΗ

«ΠΛΗΡΟΦΟΡΙΚΗ ΜΕ ΕΦΑΡΜΟΓΕΣ ΣΤΗΝ ΑΣΦΑΛΕΙΑ, ΔΙΑΧΕΙΡΙΣΗ ΜΕΓΑΛΟΥ ΟΓΚΟΥ ΔΕΔΟΜΕΝΩΝ ΚΑΙ ΠΡΟΣΟΜΟΙΩΣΗ»

Ευριστικές προσεγγίσεις στο πρόβλημα νομιμοποίησης χωροθετημένων ολοκληρωμένων κυκλωμάτων

Γώγου Μαρία

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Επιβλέπων
Δαδαλιάρης Αντώνιος

Λαμία, 2018

«Υπεύθυνη Δήλωση μη λογοκλοπής και ανάληψης προσωπικής ευθύνης»

Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, και γνωρίζοντας τις συνέπειες της λογοκλοπής, δηλώνω υπεύθυνα και ενυπογράφως ότι η παρούσα εργασία με τίτλο "Ευριστικές προσεγγίσεις στο πρόβλημα νομιμοποίησης χωροθετημένων ολοκληρωμένων κυκλωμάτων" αποτελεί προϊόν αυστηρά προσωπικής εργασίας και όλες οι πηγές από τις οποίες χρησιμοποίησα δεδομένα, ιδέες, φράσεις, προτάσεις ή λέξεις, είτε επακριβώς (όπως υπάρχουν στο πρωτότυπο ή μεταφρασμένες) είτε με παράφραση, έχουν δηλωθεί κατάλληλα και ευδιάκριτα στο κείμενο με την κατάλληλη παραπομπή και η σχετική αναφορά περιλαμβάνεται στο τμήμα των βιβλιογραφικών αναφορών με πλήρη περιγραφή. Αναλαμβάνω πλήρως, ατομικά και προσωπικά, όλες τις νομικές και διοικητικές συνέπειες που δύναται να προκύψουν στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής.

Ο/Η ΔΗΛΩΝ/-ΟΥΣΑ

Ημερομηνία

Υπογραφή

Ευριστικές προσεγγίσεις στο πρόβλημα νομιμοποίησης γωροθετημένων ολοκληρωμένων κυκλωμάτων

Γώγου Μαρία

Τριμελής Επιτροπή:

ΣΤΑΜΟΥΛΗΣ ΓΕΩΡΓΙΟΣ

ΚΟΖΥΡΗ ΜΑΡΙΑ

ΛΟΥΚΟΠΟΥΛΟΣ ΑΘΑΝΑΣΙΟΣ

Επιστημονικός Σύμβουλος:

ΔΑΔΑΛΙΑΡΗΣ ΑΝΤΩΝΙΟΣ

Ευχαριστίες

Αρχικά, θα ήθελα να ευχαριστήσω τον καθηγητή του τμήματος Πληροφορικής του Πανεπιστημίου Θεσσαλίας κ. Δαδαλιάρη Αντώνιο για την επίβλεψη και την καθοδήγηση που μου παρείχε κατά την εκπόνηση της διπλωματικής μου εργασίας.

Επίσης, θα ήθελα να ευχαριστήσω τον κ. Ιωαννίδη Σταύρο για την παραχώρηση προγενέστερης εργασίας του, μέρος της οποίας χρησιμοποιήθηκε στην παρούσα εργασία.

Τέλος, θα ήθελα να ευχαριστήσω την οικογένειά μου και τους φίλους μου για την συμπαράστασή τους καθ' όλη τη διάρκεια των σπουδών μου.

Περιεχόμενα

Πίνακας εικόνων

Περίληψη

Σκοπός της παρούσας εργασίας είναι η υλοποίηση ευριστικών μεθόδων για το πρόβλημα της νομιμοποίησης των χωροθετημένων κελιών των ολοκληρωμένων κυκλωμάτων κατά το στάδιο της Φυσικής σχεδίασης. Η αρχική χωροθέτηση των κελιών έχει πραγματοποιηθεί με τον αλγόριθμο Gordian, ωστόσο παρουσιάζονται επικαλύψεις μεταξύ των κελιών.

Συνήθως, για την επίλυση των προβλημάτων βελτιστοποίησης χρησιμοποιούνται αλγόριθμοι μαθηματικού προγραμματισμού. Ωστόσο, αρκετά συχνά καταφεύγουμε σε ευριστικές μεθόδους οι οποίες αποσκοπούν στην εύρεση μιας υπο-βέλτιστης λύσης σε σύντομο χρονικό διάστημα. Στην παρούσα εργασία, χρησιμοποιούνται ευριστικές μέθοδοι σε συνδυασμό με τον αλγόριθμο Tetris για τη νομιμοποίηση πρότυπων κελιών (standard cells).

Συνολικά, υλοποιήθηκαν τέσσερις μέθοδοι, εκ των οποίων οι τρεις χρησιμοποιούν τον αλγόριθμο Tetris για την παραγωγή μιας νόμιμης τοποθέτησης των κελιών. Οι τέσσερις μέθοδοι συγκρίνονται μεταξύ τους ως προς το συνολικό μήκος καλωδίου και τον χρόνο εκτέλεσης.

Abstract

This thesis considers the problem of legalizing the cells of integrated circuits using heuristic approaches. The global placement of the cells has been performed with the Gordian algorithm but the solution contains overlaps.

Typically, mathematical programming algorithms are used to solve optimization problems. However, we often resort to heuristic approaches which find a quick, sub-optimal solution. In this thesis, heuristic methods are used in conjunction with the Tetris algorithm for the legalization of standard cells.

In total, four methods were implemented, three of which use the Tetris algorithm to produce a legal placement of the cells. The four methods are compared in terms of total wire length and runtime.

1 ΣΧΕΔΙΑΣΗ ΟΛΟΚΛΗΡΩΜΕΝΩΝ ΚΥΚΛΩΜΑΤΩΝ

1.1 Εισαγωγή

Στο κεφάλαιο αυτό παρουσιάζονται κάποιες βασικές έννοιες και διαδικασίες που αφορούν τη σχεδίαση ολοκληρωμένων κυκλωμάτων. Περιγράφεται η ροή σχεδίασης των ολοκληρωμένων κυκλωμάτων και δίνεται έμφαση στη διαδικασία της φυσικής σχεδίασης, στην οποία εστιάζει η παρούσα εργασία. Επίσης, παρουσιάζονται τα εργαλεία αυτοματοποιημένης σχεδίασης EDA.

1.2 Σχεδίαση ολοκληρωμένων κυκλωμάτων

Η σχεδίαση ολοκληρωμένων κυκλωμάτων αποτελεί αντικείμενο μελέτης των Ηλεκτρολόγων Μηχανικών και χωρίζεται σε δύο κατηγορίες: την αναλογική σχεδίαση (analog IC design) και την ψηφιακή σχεδίαση (digital IC design).

Η ψηφιακή σχεδίαση ολοκληρωμένων κυκλωμάτων, στην οποία εστιάζει η παρούσα εργασία, έχει ως στόχο την παραγωγή κυκλωματικών στοιχείων όπως μνήμες (RAM, ROM, flash memories), μικροεπεξεργαστές (microprocessors), FPGAs (Field Programmable Gate Array) και ASICs (Application Specific Integrated Circuits). Επικεντρώνεται κυρίως:

- στη λογική ορθότητα του κυκλώματος
- στη μεγιστοποίηση της πυκνότητας του κυκλώματος
- στην τοποθέτηση των στοιχείων έτσι ώστε τα σήματα ρολογιού και χρονισμού να δρομολογούνται αποτελεσματικά.

Τα ολοκληρωμένα κυκλώματα αποτελούνται από ηλεκτρικά στοιχεία όπως τρανζίστορ, αντιστάσεις και πυκνωτές, τα οποία είναι συνδεδεμένα έτσι ώστε να σχηματίζουν ένα κύκλωμα. Συμβατικά, ο αριθμός των τρανζίστορ που περιέχει ένα ολοκληρωμένο κύκλωμα (βαθμός ολοκλήρωσης) αποτελεί δείκτη της λειτουργικής πολυπλοκότητάς του. Επίσης, ο αριθμός των τρανζίστορ ανά μονάδα επιφάνειας του ολοκληρωμένου κυκλώματος (πυκνότητα στοιχείων) χαρακτηρίζει τη στάθμη της τεχνολογίας.

Με το πέρασμα των χρόνων, τα ολοκληρωμένα κυκλώματα αποκτούν όλο και μεγαλύτερη πολυπλοκότητα ενώ η επιφάνειά τους συρρικνώνεται. Πλέον, ένα κύκλωμα VLSI έχει τις διαστάσεις που είχε παλαιότερα ένα κύκλωμα LSI. Ομοίως, τη δεκαετία του 1980 ο επεξεργαστής ενός υπολογιστή περιείχε μερικές εκατοντάδες χιλιάδες τρανζίστορ ενώ πλέον, περιέχει δισεκατομμύρια τρανζίστορ.

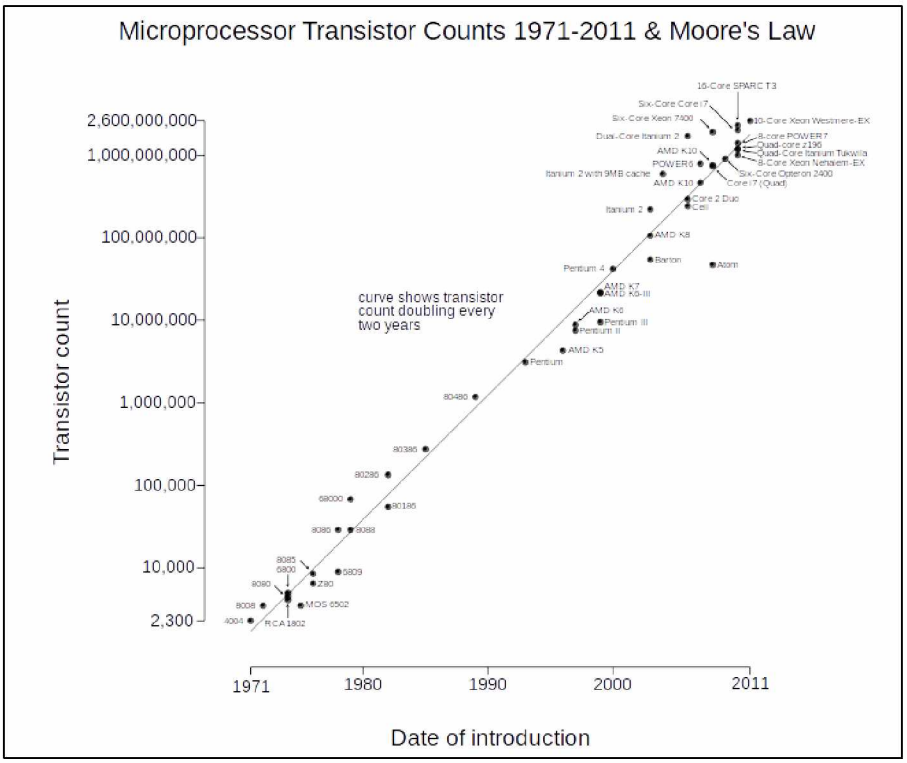
Στην παρακάτω εικόνα απεικονίζεται ο αριθμός των τρανζίστορ που περιέχονται στους μικροεπεξεργαστές ορισμένων γνωστών CPU. Την πρώτη θέση, ανάμεσα στους επεξεργαστές που παράγονται για εμπορική χρήση, καταλαμβάνει ο επεξεργαστής της Intel ο οποίος αποτελείται από 7.2 δισεκατομμύρια τρανζίστορ.

Processor	Transistor count	Date of introduction	Designer	Process	Area
32-core AMD Epyc	19,200,000,000	2017	AMD	14 nm	4× 192 mm^2
Centriq 2400	18,000,000,000[43]	2017	Qualcomm	10 nm	398mm^2
32-core SPARC M7	10,000,000,000[42]	2015	Oracle	20 nm	
IBM z14 Storage Controller	9,700,000,000	2017	IBM	14 nm	696 mm^2
POWER9	8,000,000,000	2017	IBM	14 nm	695 mm^2
22-core Xeon Broadwell-E5	7,200,000,000[41]	2016	Intel	14 nm	456 mm^2
IBM z13 Storage Controller	7,100,000,000	2015	IBM	22 nm	678 mm^2
Xbox One X (Project Scorpio) main SoC	7,000,000,000[40]	2017	Microsoft/AMD	16 nm	360 mm^2[40]
IBM z14	6,100,000,000	2017	IBM	14 nm	696 mm^2
18-core Xeon Haswell-E5	5,560,000,000[39]	2014	Intel	22 nm	661 mm^2
61-core Xeon Phi	5,000,000,000[38]	2012	Intel	22 nm	720 mm^2
Xbox One main SoC	5,000,000,000	2013	Microsoft/AMD	28 nm	363 mm^2
8-core Ryzen	4,800,000,000[37]	2017	AMD	14 nm	192 mm^2
15-core Xeon Ivy Bridge-EX	4,310,000,000[36]	2014	Intel	22 nm	541 mm^2
Apple A11 Bionic (hexa-core ARM64 "mobile SoC")	4,300,000,000	2017	Apple	10 nm	89 mm^2
12-core POWER8	4,200,000,000	2013	IBM	22 nm	650 mm^2
IBM z13	3,990,000,000	2015	IBM	22 nm	678 mm^2
Apple A10 Fusion (quad-core ARM64 "mobile SoC")	3,300,000,000	2016	Apple	16 nm	125 mm^2
10-core Core i7 Broadwell-E	3,200,000,000[34]	2016	Intel	14 nm	246 mm^2[35]
8-core Itanium Poulson	3,100,000,000	2012	Intel	32 nm	544 mm^2

Εικόνα 1: Αριθμός τρανζίστορ σε γνωστές CPU

Η ραγδαία αύξηση της πολυπλοκότητας των ολοκληρωμένων κυκλωμάτων είχε προβλεφθεί το 1975 από τον Gordon Moore, ο οποίος υποστήριζε ότι ο αριθμός των τρανζίστορ σε ένα ολοκληρωμένο κύκλωμα θα διπλασιάζεται κάθε δύο χρόνια. Αυτή η διατύπωση, η οποία είναι γνωστή ως "Νόμος του Moore", έχει επαληθευτεί από τα γεγονότα όπως φαίνεται στην παρακάτω εικόνα. Ωστόσο, αξίζει να σημειωθεί ότι από το 2015 ο ρυθμός της αύξησης του αριθμού των τρανζίστορ σε ένα ολοκληρωμένο κύκλωμα έχει αρχίσει να επιβραδύνεται.

Η ολοένα αυξανόμενη πολυπλοκότητα των ολοκληρωμένων κυκλωμάτων αλλά και οι ανάγκες της αγοράς, η οποία πάντα απαιτούσε τα ολοκληρωμένα κυκλώματα να παράγονται μαζικά και γρήγορα, οδήγησαν στην ανάπτυξη εργαλείων αυτοματοποιημένης σχεδίασης (automated design tools).



Εικόνα 2:Απεικόνιση του Νόμου του Moore

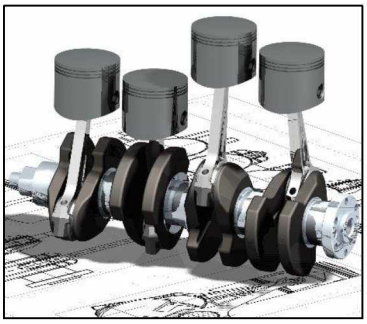
1.3 Εργαλεία CAD

Ο όρος Computer-aided design (CAD) αναφέρεται στην χρήση ηλεκτρονικών υπολογιστών κατά τη διαδικασία σχεδιασμού ενός αντικειμένου. Τα εργαλεία CAD έκαναν την εμφάνισή τους τη δεκαετία του 1980 και χρησιμοποιούνται ευρέως γιατί προσφέρουν:

- αύξηση της παραγωγικότητας των σχεδιαστών διότι καθιστούν τη διαδικασία της σχεδίασης πιο εύκολη και λιγότερο χρονοβόρα
- βελτίωση της ποιότητας της σχεδίασης, καθώς επιτρέπουν μεγαλύτερη λεπτομέρεια στα σχέδια
- εύκολη ανταλλαγή πληροφοριών που αφορούν το αντικείμενο ανάμεσα σε συνεργαζόμενες ομάδες επιστημόνων
- δημιουργία βάσης δεδομένων με πληροφορίες που αφορούν το αντικείμενο και βοηθούν στον προγραμματισμό των παραγωγικών διαδικασιών

Τα εργαλεία CAD που χρησιμοποιούνται για τον σχεδιασμό ηλεκτρονικών κυκλωμάτων, είναι γνωστά ως EDA (Electronic Design Automation). Τα εργαλεία EDA είναι πλέον απαραίτητα για την υλοποίηση ενός ολοκληρωμένου κυκλώματος και χρησιμοποιούνται για τη σχεδίαση, τον έλεγχο και την επαλήθευση της λειτουργικότητάς

του. Στην ενότητα που ακολουθεί γίνεται μια ιστορική αναδρομή στην εξέλιξη των εργαλείων αυτών.



Εικόνα 3: 3D CAD model

1.4 Εργαλεία EDA

Αρχικά, η σχεδίαση των ολοκληρωμένων κυκλωμάτων γινόταν δια χειρός από τους σχεδιαστές. Ουσιαστικά, γινόταν η γραφική αναπαράσταση του κυκλώματος σύμφωνα με την ηλεκτρονική περιγραφή του. Η γνωστότερη εταιρεία εκείνης της εποχής ήταν η Calma, η οποία δημιούργησε το πρότυπο GDSII το οποίο χρησιμοποιείται ακόμη και σήμερα.

Στα μέσα της δεκαετίας του 1970 άρχισαν να αναπτύσσονται κάποια πρώιμα εργαλεία για την διευκόλυνση της διαδικασίας της τοποθέτησης (placement) και της δρομολόγησης (routing) ενός κυκλώματος, δηλαδή την επιλογή των κατάλληλων σημείων για την τοποθέτηση των κυκλωματικών στοιχείων εντός της περιοχής που καταλαμβάνει το ολοκληρωμένο κύκλωμα.

Το 1980 η πρωτοποριακή δημοσίευση "Introduction to VLSI Systems" των Mead και Conway έθεσε ως ιδέα την χρήση λογισμικού για την κατασκευή ολοκληρωμένων κυκλωμάτων. Έτσι ξεκίνησε η ανάπτυξη λογισμικού προσομοίωσης, γεγονός που είχε ως αποτέλεσμα την αύξηση της πολυπλοκότητας των κυκλωμάτων που μπορούσαν να σχεδιαστούν αλλά και την παραγωγή πιο αξιόπιστων κυκλωμάτων, καθώς μπορούσε να γίνει επαλήθευση της σωστής λειτουργίας τους πριν την έναρξη της παραγωγικής διαδικασίας.

Την ίδια περίοδο, άρχισαν να παράγονται τα πρώτα εργαλεία αυτοματοποιημένης σχεδίασης για ακαδημαϊκούς σκοπούς. Το δημοφιλέστερο από αυτά ήταν το "VLSI Tools Tarball" που αναπτύχθηκε από το πανεπιστήμιο του Berkeley και περιελάβανε εφαρμογές σε περιβάλλον UNIX για τον σχεδιασμό των VLSI συστημάτων εκείνης της εποχής.

Ωστόσο, η ανάπτυξη των εργαλείων EDA δεν είχε εδραιωθεί ακόμη ως βιομηχανία. Οι εταιρείες του κλάδου ανέπτυσσαν μόνες τους τα εργαλεία EDA που χρησιμοποιούσαν για τον σχεδιασμό των ολοκληρωμένων κυκλωμάτων που κατασκεύαζαν. Σταδιακά, όμως, οι σχεδιαστές που εργάζονταν σε αυτές τις εταιρείες άρχισαν να ιδρύουν τις δικές

τους επιχειρήσεις που ειδικεύονταν σε αυτό τον τομέα. Μεταξύ άλλων, ιδρύθηκαν και οι εταιρείες Daisy Systems, Mentor Graphics και Valid Logic Systems, οι οποίες αποτέλεσαν τις σημαντικότερες εταιρείες της εποχής που δραστηριοποιούνταν σε αυτόν τον κλάδο.

Το 1984 άρχισε να αναπτύσσεται η Verilog ως υψηλού επιπέδου γλώσσα περιγραφής υλικού και το 1987 έκανε την εμφάνισή της η VHDL. Οι γλώσσες αυτές άρχισαν να χρησιμοποιούνται από τους προσομοιωτές, οι οποίοι διάβαζαν τις εντολές της εκάστοτε σχεδίασης και προσομοίωναν τη λειτουργία της προκειμένου να γίνει η επαλήθευσή της. Σύντομα, άρχισε και η ανάπτυξη των πρώτων εργαλείων λογικής σύνθεσης (logic synthesis).

Στις μέρες μας, υπάρχουν διάφορα εργαλεία EDA που καλύπτουν όλα τα στάδια σχεδίασης ενός ολοκληρωμένου κυκλώματος. Αρχικά, πραγματοποιείται η περιγραφή του κυκλώματος με χρήση κάποιας γλώσσας περιγραφής υλικού σε επίπεδο κελιών, τα οποία είναι τεχνολογικά ανεξάρτητα. Στη συνέχεια, με τη χρήση των κατάλληλων βιβλιοθηκών, γίνεται η προσομοίωση της λειτουργίας του κυκλώματος και, τέλος, παρέχονται οι τελικές προδιαγραφές για τις συνθήκες λειτουργίας του.

1.5 Τομείς εφαρμογής των εργαλείων EDA

Τα εργαλεία EDA που έχουν αναπτυχθεί για να καλύψουν τα στάδια της σχεδίασης ενός ολοκληρωμένου κυκλώματος ειδικεύονται στους ακόλουθους τομείς:

- Σχεδιασμός (Design)
    - High Level Synthesis
    - Logic Synthesis
    - Schematic Capture
    - Layout
- Προσομοίωση (Simulation)
    - Logic Simulation
    - Behavioral Simulation
    - Hardware Emulation
- Ανάλυση και Επαλήθευση (Analysis & Verification)
    - Functional Verification
    - Formal Verification

- Equivalence Checking
- Static Timing Analysis
- Physical Verification

- Κατασκευή (Manufacturing)
  - Mask Data Preparation

1.6 Ροή σχεδίασης ολοκληρωμένων κυκλωμάτων

Η διαδικασία που ακολουθείται για την σχεδίαση ενός ολοκληρωμένου κυκλώματος ονομάζεται ροή σχεδίασης (design flow) και αποτελείται από πολλά βήματα. Τα σημαντικότερα από αυτά είναι τα παρακάτω:

1. Feasibility study & die size estimate: προσδιορισμός του στόχου και των λειτουργικών προδιαγραφών, εκτίμηση του κόστους παραγωγής και των διαθέσιμων πόρων, εκτίμηση των φυσικών διαστάσεων του ολοκληρωμένου κυκλώματος

2. Functional verification: επαλήθευση της λογικής της σχεδίασης

3. RTL design & simulation: περιγραφή του κυκλώματος σε επίπεδο καταχωρητών και παρομοίωσή του

4. Logic simulation: προσομοίωση της λειτουργίας της σχεδίασης χρησιμοποιώντας κατάλληλο λογισμικό

5. Floor planning: σχηματική αναπαράσταση της πρώιμης τοπολογίας των τμημάτων της σχεδίασης

6. Layout: αναπαράσταση του ολοκληρωμένου κυκλώματος με γεωμετρικά σχήματα

7. Static timing analysis: ανάλυση του χρονισμού του ολοκληρωμένου κυκλώματος χωρίς να διεξαχθεί ξανά προσομοίωση της λειτουργίας του

8. Layout review: επανεξέταση της διάταξης, και αν κριθεί απαραίτητο, γίνεται ανατροφοδότηση προηγούμενων βημάτων

9. Design for testability: χρήση τεχνικών σχεδίασης που διευκολύνουν τη διαδικασία ελέγχου της ορθής λειτουργίας του ολοκληρωμένου κυκλώματος

10. Automatic test pattern generation: εύρεση της κατάλληλης αλληλουχίας εισόδου η οποία, ανάλογα με την ακολουθία που παράγεται στην έξοδο, βοηθά στον εντοπισμό ελλαττωματικών κυκλωμάτων

11. Design for manufacturability: τροποποίηση του κυκλώματος προς διευκόλυνση της παραγωγικής διαδικασίας, συνήθως για μείωση του κόστους παραγωγής

12. Mask data preparation: μετατροπή των γεωμετρικών σχημάτων που αναπαριστούν τα στοιχεία του κυκλώματος σε εντολές που μπορούν να γίνουν κατανοητές από έναν photomask writer

13. Wafer fabrication: η διαδικασία κατασκευής του δισκίου του ολοκληρωμένου κυκλώματος, η οποία περιλαμβάνει τεχνικές χημικής χάραξης και φωτολιθογραφίας

14. Packaging: το δισκίο τοποθετείται σε μια ανθεκτική συσκευασία τόσο για την προστασία του από φθορές όσο και για την εύκολη τοποθέτησή του σε κάποιο σύστημα

15. Device characterization: πραγματοποίηση μετρήσεων στο κύκλωμα με κατάλληλα όργανα για τον καθορισμό των χαρακτηριστικών του

16. Yield analysis: συλλογή στοιχείων που βοηθούν στον εντοπισμό αστοχιών του κυκλώματος με σκοπό τη διόρθωσή τους

Ωστόσο, θα μπορούσε να γίνει σύνοψη της διαδικασίας σε τρία στάδια:

1. **Σχεδίαση σε επίπεδο συστήματος (System level design):** σε αυτό το στάδιο γίνεται η σύλληψη της ιδέας του κυκλώματος από τους σχεδιαστές, πραγματοποιείται ο προσδιορισμός του στόχου και των λειτουργικών προδιαγραφών του, γίνεται εκτίμηση των προβλημάτων που ενδέχεται να παρουσιαστούν και αξιολογούνται εναλλακτικές ιδέες για την επίλυσή τους, αποφασίζεται ποιες μεθόδους επαλήθευσης θα ακολουθηθούν και εκτιμάται η φυσική διάσταση του ολοκληρωμένου κυκλώματος. Το αποτέλεσμα αυτής της διαδικασίας είναι ένα πρώιμο μοντέλο το οποίο μπορεί να περιγραφεί με τη χρήση γλωσσών υψηλού επιπέδου, όπως C, C++ και MATLAB, και να προσομοιωθεί.

2. **Λογική σχεδίαση (RTL design):** σε αυτό το στάδιο χρησιμοποιείται κάποια γλώσσα περιγραφής υλικού, όπως η Verilog και η VHDL, για την περιγραφή του κυκλώματος σε επίπεδο καταχωρητών. Στη συνέχεια, γίνεται προσομοίωση της λειτουργίας του κυκλώματος ώστε να επαληθευτεί η αναμενόμενη λειτουργία του (π.χ. χρονισμός).

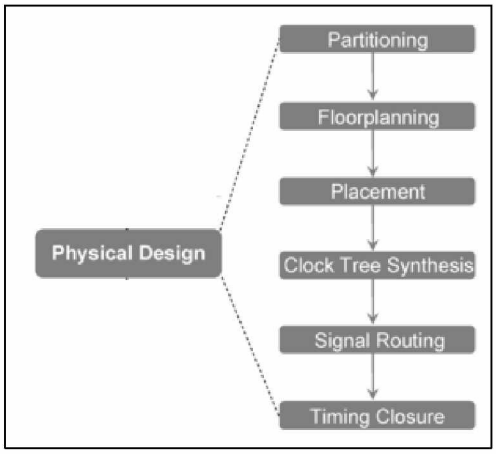
3. **Φυσική σχεδίαση (Physical design):** σε αυτό το στάδιο χρησιμοποιείται η λογική σχεδίαση σε συνδυασμό με μια βιβλιοθήκη που περιλαμβάνει τις διαθέσιμες λογικές πύλες για την υλοποίηση του ολοκληρωμένου κυκλώματος. Καθορίζονται οι πύλες που θα χρησιμοποιηθούν, η απαιτούμενη επιφάνεια, τα σημεία όπου θα τοποθετηθούν τα στοιχεία και η μεταξύ τους καλωδίωση.

1.7 Φυσική σχεδίαση

Κατά το στάδιο της φυσικής σχεδίασης, το οποίο ανήκει στο back-end της διαδικασίας σχεδίασης, πραγματοποιείται η γεωμετρική αναπαράσταση των στοιχείων του κυκλώματος.

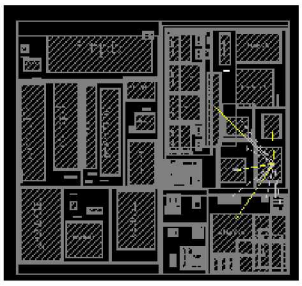
Η φυσική σχεδίαση βασίζεται στο netlist που προκύπτει από τη διαδικασία της λογικής σύνθεσης (logic synthesis), δηλαδή τη διαδικασία κατά την οποία η σχεδίαση μεταφράζεται από HDL σε ένα netlist σε επίπεδο πυλών το οποίο προσδιορίζεται από μια τεχνολογική βιβλιοθήκη. Το netlist περιέχει πληροφορίες σχετικά με τα κελιά, τις συνδέσεις τους, την επιφάνεια που καλύπτουν κ.ά. Ορισμένα γνωστά εργαλεία που χρησιμοποιούνται για τη διαδικασία της σύνθεσης είναι το Cadence RTL Compiler και το Synopsys Design Compiler. Μόνο όταν πραγματοποιηθεί η επαλήθευση του netlist ως προς τη λειτουργικότητα και τον χρονισμό του, μπορεί να συνεχιστεί το στάδιο της φυσικής σχεδίασης.

Στην παρακάτω εικόνα παρουσιάζονται τα σημαντικότερα επιμέρους βήματα που συνθέτουν το στάδιο αυτό:



Εικόνα 4:Βήματα κατά το στάδιο της φυσικής σχεδίασης

1. **Floorplanning**: Σε αυτό το βήμα εκτιμάται ο συνολικός χώρος που απαιτείται για την τοποθέτηση των βασικών λειτουργικών μονάδων του κυκλώματος και η θέση τους εντός της διαθέσιμης επιφάνειας. Η επιλογή των θέσεων είναι ζωτικής σημασίας καθώς επηρεάζει το κόστος και την απόδοση του ολοκληρωμένου κυκλώματος. Μια κακή επιλογή των θέσεων μπορεί να οδηγήσει σε σπατάλη χώρου ή συμφόρηση της δρομολόγησης (routing congestion).



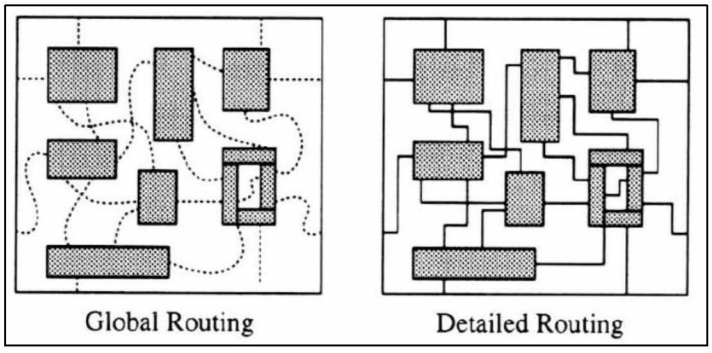
Εικόνα 5:Floorplan ενός ολοκληρωμένου κυκλώματος

2. **Partitioning**: Σε αυτό το βήμα η συνολική επιφάνεια του ολοκληρωμένου κυκλώματος διαιρείται σε υποπεριοχές. Με τον τρόπο αυτό διευκολύνεται η διαχείρισή τους και μπορούν να εκτελεστούν ευκολότερα οι διαδικασίες του placement και του routing.

3. **Placement**: Σε αυτό το βήμα επιλέγεται η βέλτιστη θέση για την τοποθέτηση των κελιών εντός της προκαθορισμένης επιφάνειας. Η επιλογή των θέσεων των κελιών παίζει σπουδαίο ρόλο καθώς επηρεάζει την κατανομή της θερμότητας, το μήκος των καλωδίων, την κατανάλωση ισχύος κ.ά. Στην χείριστη περίπτωση, όπου το μήκος των καλωδίων είναι άνω του επιτρεπτού ορίου, η υλοποίηση του ολοκληρωμένου κυκλώματος καθίσταται αδύνατη.

4. **Clock Tree Synthesis**: Ενώ στα προηγούμενα βήματα το σήμα του ρολογιού θεωρείται ιδανικό και δεν λαμβάνονται υπόψη τα delays, σε αυτό το βήμα πραγματοποιείται η ελαχιστοποίηση του skew (παραμόρφωση του ρολογιού) και του insertion delay.

5. **Routing**: Σε αυτό το βήμα προστίθενται οι καλωδιώσεις που συνδέουν τα στοιχεία του κυκλώματος, όπως ορίζονται από το netlist. Το routing γίνεται σε δύο στάδια. Πρώτα, εφαρμόζεται το global routing το οποίο υπολογίζει μια προσεγγιστική δρομολόγηση για κάθε net και το detailed routing το οποίο προσδιορίζει επακριβώς τα μονοπάτια (routes) σε συγκεκριμένα μεταλλικά στρώματα.



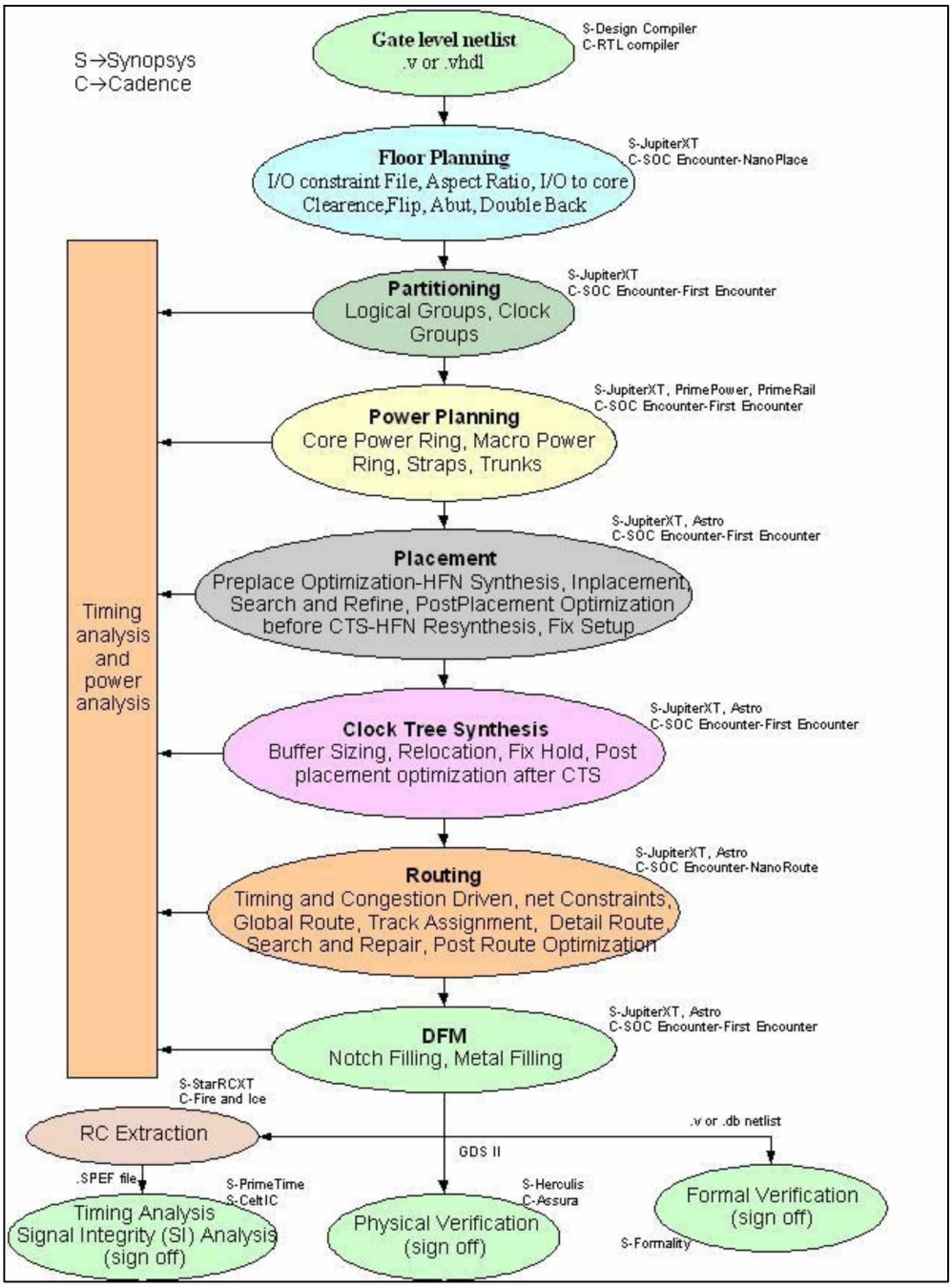
Εικόνα 6:Global routing και detailed routing

6. **Physical Verification**: Το βήμα αυτό αποτελεί το τελευταίο στάδιο της φυσικής σχεδίασης, όπου πραγματοποιείται ο έλεγχος της ορθότητας της σχεδίασης που έχει παραχθεί.

Ορισμένα από τα εργαλεία EDA που χρησιμοποιούνται για την υλοποίηση της διαδικασίας της φυσικής σχεδίασης είναι:

- Synopsys (Design Compiler, IC Compiler)
- Mentor Graphics (Olympus SoC, Calibre)
- Magma (BlastFusion)
- Cadence (Cadence Encounter RTL Compiler, Encounter Digital Implementation, Cadence Voltus IC Power Integrity Solution, Cadence Tempus Timing Signoff Solution)

Στην παρακάτω εικόνα παρουσιάζονται αναλυτικότερα τα βήματα της φυσικής σχεδίασης, καθώς και τα εργαλεία EDA που συνήθως χρησιμοποιούνται για την υλοποίησή τους.



Εικόνα 7:Ροή της φυσικής σχεδίασης

2 ΧΩΡΟΘΕΤΗΣΗ ΟΛΟΚΛΗΡΩΜΕΝΩΝ ΚΥΚΛΩΜΑΤΩΝ

2.1 Εισαγωγή

Το κεφάλαιο αυτό αναφέρεται στη χωροθέτηση (placement) των κελιών, η οποία αποτελεί ένα από τα σημαντικότερα βήματα της φυσικής σχεδίασης. Γίνεται αναφορά στα επιμέρους στάδια στα οποία υποδιαιρείται το πρόβλημα του placement και παρουσιάζονται δύο από τις δημοφιλέστερες μεθόδους που χρησιμοποιούνται κατά το στάδιο του global placement και του legalization των κελιών.

2.2 Χωροθέτηση (placement)

Κατά το στάδιο της χωροθέτησης, επιλέγεται η βέλτιστη θέση για την τοποθέτηση των κελιών εντός μιας καθορισμένης επιφάνειας (core area) προκειμένου να ελαχιστοποιηθεί η τιμή μιας αντικειμενικής συνάρτησης. Δηλαδή, το εργαλείο που θα χρησιμοποιηθεί για την χωροθέτηση (placer) καλείται, αφενός να τοποθετήσει τα στοιχεία του κυκλώματος έτσι ώστε να μην υπάρχουν επικαλύψεις (overlaps) μεταξύ αυτών και, αφετέρου, να βελτιώσει ορισμένες παραμέτρους που χρήζουν βελτίωσης ώστε το κύκλωμα να πληροί τους στόχους που έχουν τεθεί. Συνήθως, τα χαρακτηριστικά που επιδέχονται βελτίωσης είναι τα παρακάτω:

- **Total wirelength**: Η ελαχιστοποίηση του συνολικού μήκους των καλωδίων αποτελεί τον σημαντικότερο στόχο των εργαλείων placement. Αυτό συμβαίνει γιατί το μήκος των καλωδίων επηρεάζει το μέγεθος του ολοκληρωμένου κυκλώματος, και επομένως, το κόστος παραγωγής του καθώς και την κατανάλωση ισχύος και το delay.

- **Timing**: Ο κύκλος του ρολογιού προσδιορίζεται από την καθυστέρηση του μεγαλύτερου μονοπατιού (critical path). Ένα εργαλείο placement που αποσκοπεί στην βελτίωση του χρονισμού του κυκλώματος, πρέπει να εξασφαλίσει την μη ύπαρξη μονοπατιού με καθυστέρηση μεγαλύτερη από αυτή του critical path.

- **Congestion**: Η ελαχιστοποίηση του συνολικού μήκους των καλωδίων πρέπει να πραγματοποιηθεί χωρίς να προκληθεί "συνωστισμός" καλωδίων στις υποπεριοχές του ολοκληρωμένου κυκλώματος. Η συμφόρηση των καλωδίων μπορεί να οδηγήσει σε υπερβολικές παρακάμψεις στη δρομολόγηση ή να καταστήσει αδύνατη τη σύνδεση κάποιων στοιχείων.

- **Power**: Η ελαχιστοποίηση της κατανάλωσης ισχύος προϋποθέτει την τοποθέτηση των στοιχείων έτσι ώστε να καταναλώνεται όσο το δυνατόν λιγότερη ισχύς.

Ο placer, χρησιμοποιεί το netlist που προέκυψε από το στάδιο της λογικής σύνθεσης και μια τεχνολογική βιβλιοθήκη προκειμένου να υπολογίσει μια ορθή διάταξη (layout). Στη συνέχεια, πραγματοποιεί τις βελτιστοποιήσεις και, αν χρειαστεί, ανατροφοδοτούνται προηγούμενα βήματα της σχεδίασης του ολοκληρωμένου κυκλώματος.

Το layout που προκύπτει ως αποτέλεσμα της διαδικασίας του placement ανήκει σε κάποια από τις κατηγορίες που περιγράφονται παρακάτω. Καθένα από τα layouts έχει διαφορετικούς περιορισμούς και πλεονεκτήματα.

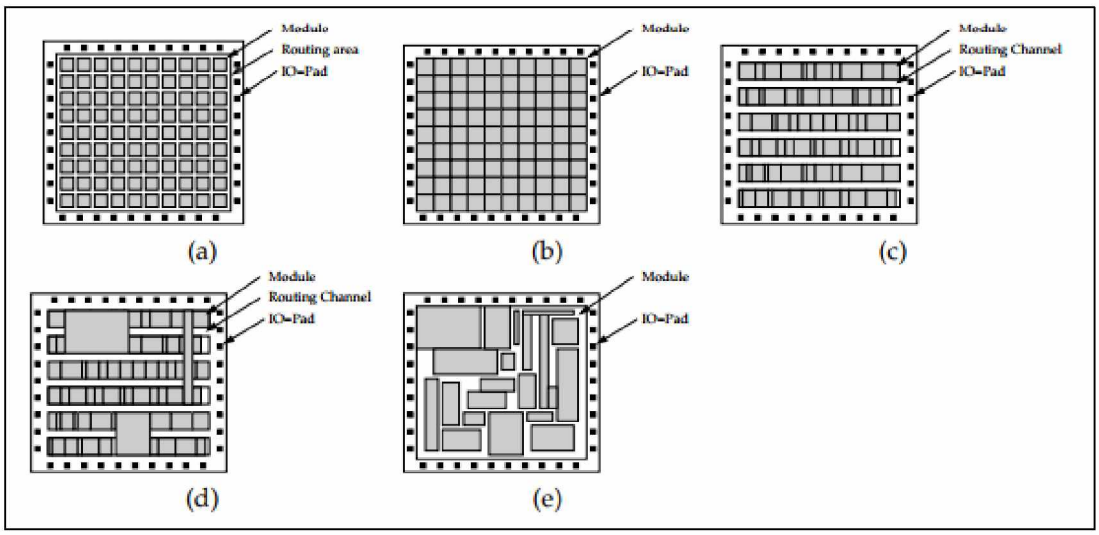
1. **Gate array**: Υπάρχουν πανομοιότυπες, προκαθορισμένες θέσεις στις οποίες μπορούν να τοποθετηθούν τα κελιά, δηλαδή όλα τα κελιά έχουν το ίδιο ύψος και πλάτος. Επίσης, υπάρχουν κανάλια για τα καλώδια που ενώνουν τα κελιά.

2. **Sea-of-gates**: Παρόμοια περίπτωση με το gate array, με τη διαφορά ότι δεν υπάρχουν κανάλια για την καλωδίωση.

3. **Standard-cell** (πρότυπα κελιά): Αποτελεί τη δημοφιλέστερη κατηγορία. Όλα τα κελιά έχουν το ίδιο ύψος αλλά μπορεί να διαφέρουν σε πλάτος. Τα κελιά τοποθετούνται σε προκαθορισμένες γραμμές οι οποίες απέχουν συγκεκριμένη απόσταση μεταξύ τους. Στην παρούσα εργασία χρησιμοποιείται αυτό το layout.

4. **Mixed-cell**: Παρόμοια περίπτωση με το standard-cell, με τη διαφορά ότι τα κελιά ποικίλουν τόσο ως προς το πλάτος όσο και ως προς το ύψος. Ένα κελί μπορεί να επεκτείνεται σε παραπάνω από μία γραμμή (macro block).

5. **General-cell (Macros)**: Δίνει στον σχεδιαστή πλήρη ελευθερία για την τοποθέτηση των κελιών. Τα κελιά μπορούν να έχουν οποιοδήποτε μέγεθος και μπορούν να τοποθετηθούν οπουδήποτε εντός της διαθέσιμης περιοχής.



Εικόνα 8:Layout Models

(a) Gate array. (b) Sea-of-gates. (c) Standard-cells. (d) Mixed cell. (e) General-cell layout.

Το πρόβλημα της χωροθέτησης υποδιαιρείται σε πέντε κατηγορίες:

1. **Global placement**: Τα κελιά τοποθετούνται όσο το δυνατόν πιο κοντά στις "ιδανικές" τους θέσεις, ωστόσο, υπάρχουν επικαλύψεις μεταξύ των κελιών. Η διαδικασία μπορεί να επαναληφθεί μέχρι να παραχθεί ένα σχετικά καλό αποτέλεσμα.

2. **Final placement**: Γίνεται βελτιστοποίηση των θέσεων των κελιών που έχουν προκύψει στο παραπάνω βήμα. Το αποτέλεσμα αυτής της διαδικασίας είναι, συνήθως, η τελική τοποθέτηση των κελιών, χωρίς επικαλύψεις.

3. **Area minimization**: Επιχειρείται η ελαχιστοποίηση της περιοχής που καταλαμβάνουν τα κελιά. Το πρόβλημα αυτό ανήκει στην κατηγορία των NP-hard προβλημάτων.

4. **Legalization**: Εάν εξακολουθούν να υπάρχουν επικαλύψεις μεταξύ των κελιών, εφαρμόζονται τεχνικές νομιμοποίησης για την εξάλειψή τους. Η παρούσα εργασία μελετά τέτοιου είδους τεχνικές.

5. **Post optimization**: Εφαρμογή τεχνικών για την βελτιστοποίηση της τελικής λύσης, αλλάζοντας τη θέση ελάχιστων κελιών.

Η παρούσα εργασία επικεντρώνεται στη διαδικασία του legalization πρότυπων κελιών (standard cells). Η αρχική τοποθέτηση των κελιών εντός της διαθέσιμης περιοχής, έχει πραγματοποιηθεί με τον αλγόριθμο Gordian ο οποίος περιγράφεται στην επόμενη ενότητα.

2.3 Global Placement με τη μέθοδο Gordian

Ο Gordian[1] αποτελεί έναν από τους δημοφιλέστερους αλγόριθμους που χρησιμοποιούνται για το global placement και είναι αυτός ο οποίος χρησιμοποιήθηκε στην παρούσα περίπτωση.

Τα δύο βασικά χαρακτηριστικά του είναι ότι πραγματοποιεί global optimization σε κάθε βήμα και ότι δημιουργεί αποκλειστικά ορθογώνιες περιοχές κατά τον διαμερισμό της περιοχής.

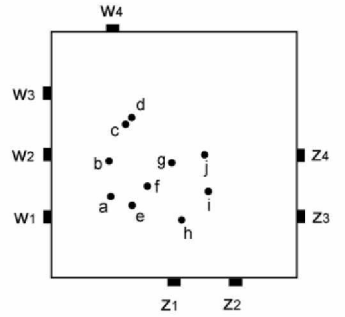
Τα βήματά του περιγράφονται ως εξής:

- Τοποθέτηση όλων των κελιών εντός της διαθέσιμης περιοχής.

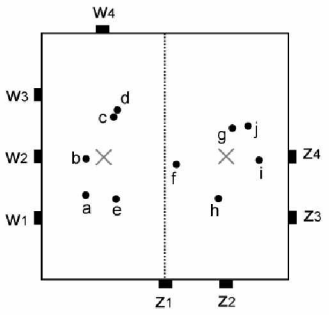
- Δημιουργία ομάδων κελιών χρησιμοποιώντας τεχνικές τμηματοποίησης τηρώντας τους κανόνες καθολικής βελτιστοποίησης που έχουν τεθεί.

- Αν οι ομάδες κελιών που έχουν προκύψει αποτελούνται από αριθμό κελιών μικρότερο από αυτόν που έχει προκαθοριστεί, τότε γίνεται τοποθέτηση των κελιών στον διαθέσιμο χώρο.

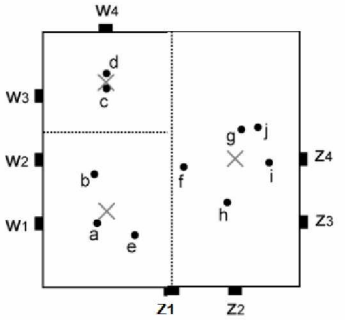
Στο παράδειγμα που ακολουθεί, ο Gordian εφαρμόζεται σε ένα κύκλωμα αποτελούμενο από δέκα κελιά με ελάχιστο αριθμό κελιών ανά περιοχή ίσο με τρία.



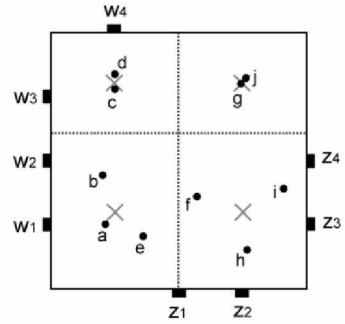
Τυχαία τοποθέτηση των κελιών εντός της διαθέσιμης περιοχής. Περιμετρικά βρίσκονται τα I/O pins.



1ο partitioning



2ο partitioning



3ο partitioning

Εικόνα 9:Παράδειγμα εφαρμογής του Gordian

Τα κελιά αναπαρίστανται ως σημεία, αλλά στις πραγματικές διαστάσεις ενδεχομένως να υπάρχουν επικαλύψεις. Αν μετά το 3ο partitioning υπάρχουν επικαλύψεις, τα κελιά μετατοπίζονται εντός της περιοχής τους έτσι ώστε αυτές να εξαλειφθούν.

2.4 Νομιμοποίηση (legalization)

Η χωροθέτηση που προκύπτει από την διαδικασία του global placement, είναι μια αρχική τοποθέτηση των κελιών και είναι πολύ πιθανό να υπάρχουν επικαλύψεις μεταξύ αυτών ή να υπάρχουν κελιά που βρίσκονται εκτός των ορίων της προκαθορισμένης περιοχής. Η διαδικασία του global placement μπορεί να επαναληφθεί ορισμένες φορές προκειμένου να παραχθεί ένα καλύτερο αποτέλεσμα, αλλά αρκετά συχνά απαιτείται η χρήση εργαλείων νομιμοποίησης (legalization) για την πλήρη εξάλειψη των επικαλύψεων.

Έχουν αναπτυχθεί διάφορες μέθοδοι για τη διαδικασία του legalization. Ορισμένοι από τους πιο διαδεδομένους αλγόριθμους είναι ο Tetris, ο Abacus, ο Jezz, ο HiBin και ο BonnPlace.

Συνήθως, για την επίλυση των προβλημάτων βελτιστοποίησης χρησιμοποιούνται αλγόριθμοι μαθηματικού προγραμματισμού. Ωστόσο, αρκετά συχνά καταφεύγουμε σε ευριστικές μεθόδους οι οποίες βασίζονται στη δημιουργική σκέψη του ανθρώπου και στην κοινή λογική. Οι μέθοδοι αυτές αποσκοπούν στην εύρεση μια υπο-βέλτιστης λύσης σε σύντομο χρονικό διάστημα. Στην παρούσα εργασία, χρησιμοποιούνται ευριστικές μέθοδοι σε συνδυασμό με τον αλγόριθμο Tetris για τη νομιμοποίηση πρότυπων κελιών (standard cells).

2.5 Legalization με τη μέθοδο Tetris

Μια από τις δημοφιλέστερες προσεγγίσεις για την διαδικασία της νομιμοποίησης των κελιών, είναι ο αλγόριθμος Tetris[2]. Τα βήματά του συνοψίζονται ως εξής:

- Ταξινόμηση των κελιών σε αύξουσα διάταξη ως προς τη συντεταγμένη x.
- Υπολογισμός του κόστους μετακίνησης του κελιού σε καθεμία από τις γραμμές στις οποίες υπάρχει διαθέσιμη θέση. Το κόστος υπολογίζεται ως η ευκλείδεια απόσταση μεταξύ της αρχικής και της υποψήφιας τελικής θέσης.
- Τοποθέτηση του κελιού στην πρώτη διαθέσιμη θέση της γραμμής με το μικρότερο κόστος.
- Ενημέρωση της τιμής της πρώτης διαθέσιμης θέσης της επιλεγμένης γραμμής.

Ο αλγόριθμος αυτός είναι απλός, αποτελεσματικός, εύκολος στην υλοποίησή του και ενδείκνυται για απλές σχεδιάσεις. Χρησιμοποιείται ως μέτρο σύγκρισης για άλλους αλγόριθμους και έχουν παρουσιαστεί διάφορες παραλλαγές του.

Παρακάτω, παρατίθεται ο ψευδοκώδικας του αλγόριθμου:

```
{C} = cells to be legalized
Sort cells in C by their coordinate X
for i=1 to the number of cells in C do
best = max_cost;
for j=1 to the number of rows do
LM = leftmost of row j
if LM + cell_sizeX < core_area_width
CP = current position of the cell
cost = displacement of moving cell i to LM from CP
if cost <= best
best = cost;
best_row = j;
 end if;
   end if;
       end for;
Move cell i to best_row;
Update leftmost LM of best_row;
end for;
```

3 ΜΕΘΟΔΟΙ ΠΟΥ ΥΛΟΠΟΙΗΘΗΚΑΝ

3.1 Εισαγωγή

Οι μέθοδοι υλοποιήθηκαν χρησιμοποιώντας την γλώσσα προγραμματισμού C. Οι πληροφορίες που περιγράφουν λεπτομερώς ένα κύκλωμα αντλούνται από αρχεία τα οποία περιέχουν τα ονόματα των κελιών, τις διαστάσεις τους, τις συνδέσεις μεταξύ τους (nets) και τις αρχικές τους θέσεις εντός της διαθέσιμης περιοχής. Τα pins βρίσκονται περιμετρικά της core area και έχουν προκαθορισμένες θέσεις οι οποίες δεν αλλάζουν κατά τη διαδικασία του legalization.

3.2 Πρώτος αλγόριθμος

Στον αλγόριθμο αυτό, ξεκινάμε από το net με τα περισσότερα κελιά, τοποθετούμε τα κελιά που ανήκουν στο net χρησιμοποιώντας τον αλγόριθμο Tetris και συνεχίζουμε με το επόμενο σε μέγεθος net.

Αρχικά, γίνεται ταξινόμηση των net ως προς την τιμή connectivity, η οποία δηλώνει το πλήθος των κελιών που ανήκουν στο net. Στη συνέχεια, σε κάθε net εντοπίζουμε τα κελιά τα οποία δεν είναι terminal και τα τοποθετούμε στην προκαθορισμένη περιοχή χρησιμοποιώντας τον αλγόριθμο Tetris. Τα κελιά που ανήκουν σε παραπάνω από ένα net, αρκεί να τοποθετηθούν την πρώτη φορά που θα εξετασθούν.

Στον παρακάτω ψευδοκώδικα περιγράφονται τα βασικά βήματα του αλγόριθμου:

```
{N} = All nets
Calculate connectivity of nets in {N}
Sort nets in {N} by their connectivity value
for i=1 to the number of nets in N do
{c} = cells in each net that have not been already placed
if |c| > 0
Apply Tetris for cells in {c}
      end if;
end for;
```

3.3 Δεύτερος αλγόριθμος

Σε αυτόν τον αλγόριθμο δημιουργούμε κατηγορίες από nets ανάλογα με το πόσα κοινά κελιά έχουν μεταξύ τους. Τοποθετούμε τα κελιά του κάθε net ξεκινώντας από το net με τους περισσότερους συσχετισμούς και συνεχίζουμε με τα υπόλοιπα. Για το legalization των nets χρησιμοποιείται ο αλγόριθμος Tetris.

Για να εντοπίσουμε τα κοινά κελιά μεταξύ των net, πρέπει για κάθε κελί του net να ελέγξουμε αν ανήκει και σε κάποιο άλλο net. Κάθε φορά που εντοπίζουμε ένα κοινό κελί μεταξύ δύο net, ενημερώνουμε την τιμή του πεδίου connectivity και στα δύο nets. Στη συνέχεια, αφού γίνει ταξινόμηση των nets βάσει του connectivity, αρχίζουμε να τοποθετούμε τα κελιά του κάθε net χρησιμοποιώντας τον αλγόριθμο Tetris. Τα κελιά που ανήκουν σε παραπάνω από ένα net, αρκεί να τοποθετηθούν την πρώτη φορά που θα εξετασθούν.

Ο ψευδοκώδικας του αλγόριθμου παρουσιάζεται παρακάτω:

```
{N} = All nets
Create net categories
Sort nets in {N} by their connectivity
for i=1 t to the number of nets in {N} do
{c} = cells in each net that have not been already placed
if |c| > 0
Apply Tetris for cells in {c}
      end if;
end for;
```

3.4 Τρίτος αλγόριθμος

Ο αλγόριθμος αυτός εξετάζεται σε κυκλώματα που περιέχουν κελιά που συνδέονται το πολύ με τρία pins. Βρίσκουμε ποια κελιά έχουν pins στα nets που ανήκουν και τα τοποθετούμε στην καλύτερη δυνατή θέση. Η τοποθέτηση των υπόλοιπων κελιών πραγματοποιείται με τον αλγόριθμο Tetris.

Αρχικά, εντοπίζουμε τα κελιά που ανήκουν σε nets που περιέχουν pins. Τα κελιά αυτά τοποθετούνται κοντά στην περίμετρο της προκαθορισμένης περιοχής, έτσι ώστε να είναι όσο το δυνατόν πιο κοντά στα pins.

Συγκεκριμένα, τα κελιά που συνδέονται με 1 pin το οποίο βρίσκεται στην κάτω ή επάνω πλευρά της περιοχής, τοποθετούνται στην πρώτη ή τελευταία γραμμή αντίστοιχα. Τα κελιά που συνδέονται με 1 pin που βρίσκεται στην αριστερή ή δεξιά πλευρά της περιοχής, τοποθετούνται στην πρώτη διαθέσιμη θέση στο αριστερό (leftmost) ή δεξί (rightmost) άκρο της κατάλληλης γραμμής. Αν η γραμμή δεν έχει χώρο, επιλέγεται κάποια γειτονική.

Για τα κελιά που συνδέονται με 2 pins βρίσκουμε το μέσο της ευθείας που συνδέει τα 2 pins. Οι συντεταγμένες του μέσου είναι $x = (x1+x2) / 2$ και $y = (y1+y2) / 2$

Για τα κελιά που συνδέονται με 3 pins, βρίσκουμε τις συντεταγμένες του βαρύκεντρου του τριγώνου που σχηματίζεται αν θεωρήσουμε ότι τα pins βρίσκονται στις κορυφές του τριγώνου. Οι συντεταγμένες του βαρύκεντρου υπολογίζονται συναρτήσει των συντεταγμένων των κορυφών του τριγώνου: x= (x1+x2+x3) / 3 και y = (y1+y2+y3) / 3

Και στις δύο περιπτώσεις, η συντεταγμένη y βοηθά στον υπολογισμό της κατάλληλης γραμμής ενώ από την τιμή της συντεταγμένης x αποφασίζουμε αν το κελί θα τοποθετηθεί στην πρώτη διαθέσιμη θέση στο αριστερό ή στο δεξί άκρο της γραμμής. Αν η γραμμή δεν έχει χώρο επιλέγεται κάποια γειτονική.

Τέλος, εφαρμόζεται ο αλγόριθμος Tetris για τα εναπομείναντα κελιά. Ωστόσο, επειδή οι γραμμές ενδέχεται να περιέχουν κελιά τόσο στο αριστερό όσο και στο δεξί άκρο, τροποποιούμε τον Teris εφαρμόζοντας τον έλεγχο που φαίνεται στον παρακάτω ψευδοκώδικα για να ελέγξουμε αν η γραμμή έχει χώρο για το κελί που εξετάζουμε:

```
LM = leftmost of row j
RM = rightmost of row j
if (RM – LM) >= cell_i_sizeX
      CP = current position of cell i
cost = displacement of moving cell i to LM from CP
if cost <= best
best = cost;
best_row = j;
end if;
end if;
```

Ο ψευδοκώδικας του αλγόριθμου παρουσιάζεται παρακάτω:

```
{C1} = The cells that belong to nets with pins
{C2} = Remaining cells
Sort cells in C1 by coordinate X
for i=1 to the number of cells in C1 do
      Find the pin(s) that are connected to cell i
      Find the preferred row for the cell (calculate coordinate Y)
      Find if the cell should be placed to the leftmost or rightmost of
      the row (calculate coordinate X)
      Place the cell
      Update leftmost or rightmost of the row
end for;
Apply Tetris for cells in C2
```

3.5 Τέταρτος αλγόριθμος

Στον συγκεκριμένο αλγόριθμο, ξεκινάμε βάζοντας τα κελιά στην κατάλληλη γραμμή, η οποία έχει προκύψει από την εφαρμογή του Gordian κατά τη διαδικασία του global placemnt των κελιών. Σε κάθε γραμμή, εντοπίζουμε το κεντρικό κελί και το τοποθετούμε στο μέσο της γραμμής. Στη συνέχεια, αρχίζουμε να τοποθετούμε αριστερά (leftmost) και δεξιά (rightmost) τα υπόλοιπα κελιά. Σε περίπτωση που γεμίσει η γραμμή, ελέγχουμε εάν μπορούμε να τα μετακινήσουμε σε κάποια γειτονική. Αν το κελί προς μετακίνηση βρίσκεται αριστερά του κεντρικού κελιού, το τοποθετούμε σε κάποια γραμμή που έχει διαθέσιμη θέση στα αριστερά. Ομοίως, αν το κελί βρίσκεται δεξιά του κεντρικού κελιού, το τοποθετούμε σε κάποια γραμμή που έχει διαθέσιμη θέση στα δεξιά.

Για κάθε γραμμή, πρέπει να εντοπίσουμε τα κελιά που της ανήκουν και να υπολογίσουμε το συνολικό μήκος που καταλαμβάνουν τα κελιά όταν τοποθετηθούν στη σειρά. Έτσι μπορούμε να χωρίσουμε τις γραμμές σε δύο κατηγορίες:

1. σε εκείνες όπου όλα τα κελιά τους χωρούν στη γραμμή, επομένως αρκεί να μετακινήσουμε τα κελιά εντός της γραμμής για να αποφύγουμε τις επικαλύψεις

2. και σε εκείνες όπου παρουσιάζεται "υπερχείλιση" κελιών, επομένως πρέπει να μετακινήσουμε τα κελιά που περισσεύουν σε γειτονικές γραμμές

Πρώτα, ο αλγόριθμος εξαλείφει τις επικαλύψεις των κελιών στην πρώτη κατηγορία γραμμών και συνεχίζει με τη δεύτερη κατηγορία.

Τα βήματα του αλγόριθμου περιγράφονται παρακάτω.

```
{C} = all cells
for i=1 to the number of cells in {C} do
      place cell i to the row defined by Gordian
end for;

//step 1:legalize the rows where the length of the row's cells is <=
coreAreaWidth
for i=1 to the number of rows do
      Calculate total length of cells in row i
      if length_i <= coreAreaWidth
            {c1} = the cells that belong to row i
            sort cells in {c1} by coordinate X
            find middle cell and place it in the middle of the row i
            for j=1 to the number of cells in {c1}

                  if cell_j_centerX < coreAreaWidth/2
                        place cell j to the leftmost of row i
                  else
                        place cell j to the rightmost of row i
                  end if;

            end for;
            mark row i as legalized
      end if;
end for;
```

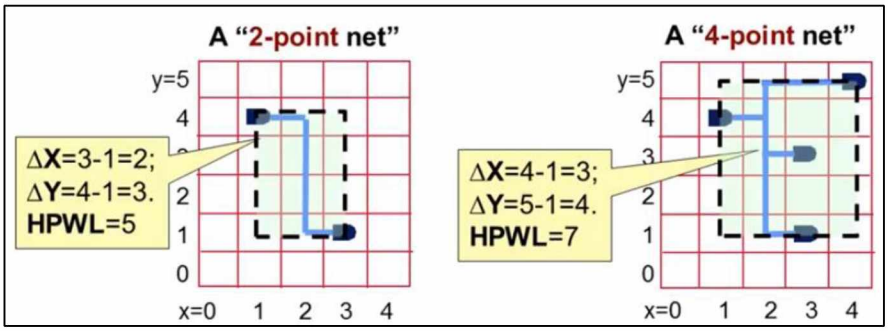
```
//step 2:legalize remaining rows, by moving cells that don't fit in to
neighboring //legalized rows
for i=1 to the number of rows do
	Calculate total length of cells in row i
	if length_i > coreAreaWidth
		{c2} = the cells that belong to row i
		sort cells in {c2} by coordinate X
		find middle cell and place it in the middle of the row
		for j=1 to the number of cells in {c2}
			if cell_j_centerX < coreAreaWidth/2
				if leftmost - cell_j_sizeX > 0
					place cell j to the leftmost of row i
				else
					find a neighboring legalized row
				end if;
			else
				if rightmost + cell_j_sizeX <= coreAreaWidth
					place cell j to the rightmost of row i
				else
					find a neighboring legalized row
				end if;
			end if;
		end for;
		mark row i as legalized
	end if;
end for;
```

4 ΣΥΜΠΕΡΑΣΜΑΤΑ

4.1 Εισαγωγή

Σε αυτό το κεφάλαιο παρουσιάζονται τα αποτελέσματα από την εκτέλεση των αλγορίθμων που υλοποιήθηκαν. Τα μεγέθη που συγκρίνονται είναι το half-perimeter wire length (hpwl) και ο χρόνος εκτέλεσης του αλγόριθμου σε ms. Το half-perimeter wire length είναι η ημι-περίμετρος του ορθογωνίου που περικλείει όλα τα κελιά που ανήκουν σε ένα net και υπολογίζεται με τον εξής τρόπο:

- Βρίσκουμε το μικρότερο ορθογώνιο που περικλείει όλα τα κελιά που ανήκουν στο net
- Προσθέτουμε το πλάτος (ΔX) και ύψος (ΔY) του ορθογωνίου



Εικόνα 9:Υπολογισμός hpwl

Για τον έλεγχο των μεθόδων που υλοποιηθήκαν χρησιμοποιήθηκαν τα ISPD04 IBM Standard Cell Benchmarks[3]. Τα χαρακτηριστικά των κυκλωμάτων φαίνονται στον παρακάτω πίνακα:

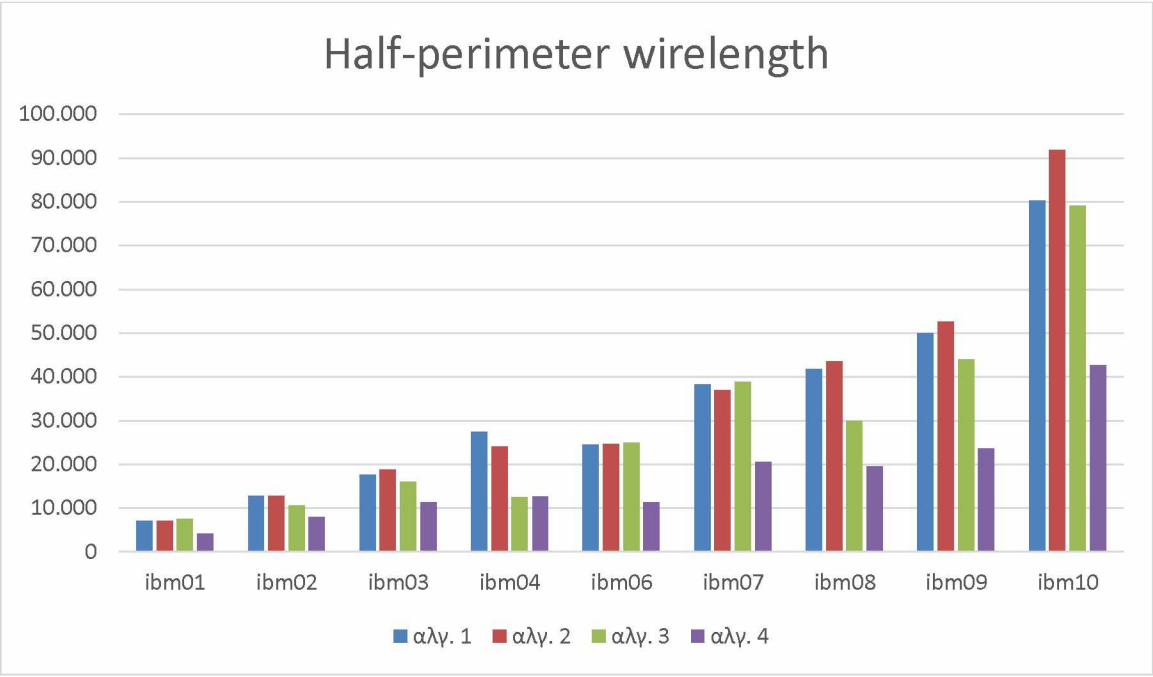
Κύκλωμα	Initial HPWL (e6)	Αρ. κελιών	Αρ. nets	Αρ. γραμμών
ibm01	2.679	12506	14111	96
ibm02	6.693	19342	19584	109
ibm03	8.742	22853	27401	121
ibm04	10.156	27220	31970	136
ibm06	7.722	32332	34826	126
ibm07	15.965	45639	48117	166
ibm08	14.616	51023	50513	170
ibm09	15.759	53110	60902	183
ibm10	32.677	68685	75196	234

4.2 Συγκεντρωτικά αποτελέσματα

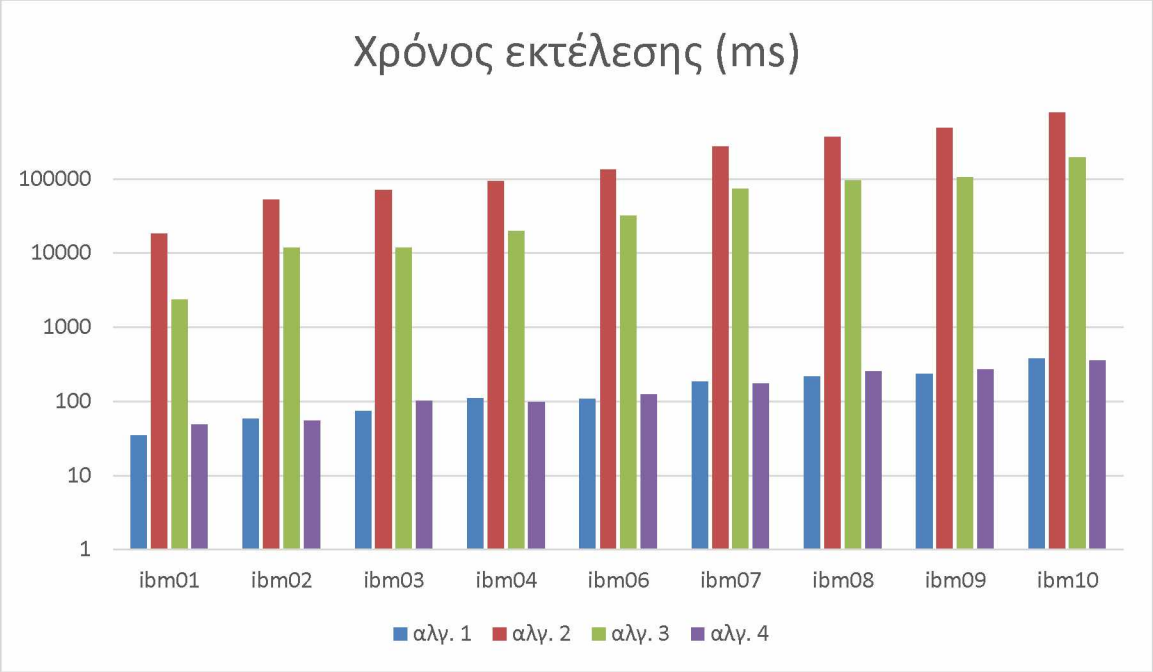
Στον παρακάτω πίνακα παρουσιάζονται τα συγκεντρωτικά αποτελέσματα της εκτέλεσης των αλγορίθμων.

	1ος αλγόριθμος		2ος αλγόριθμος		3ος αλγόριθμος		4ος αλγόριθμος	
Κύκλωμα	**Final HPWL**	**Time (ms)**	**Final HPWL**	**Time (ms)**	**Final HPWL**	**Time (ms)**	**Final HPWL**	**Time (ms)**
ibm01	7.133	35	7.118	18191	7.536	2351	4.205	49
ibm02	12.804	58	12.757	53001	10.601	11948	7.971	55
ibm03	17.524	74	18.835	70601	16.090	11844	11.342	102
ibm04	27.507	110	24.027	93840	12.473	19881	12.648	97
ibm06	24.446	108	24.728	133353	24.862	32327	11.331	125
ibm07	38.288	185	36.960	273499	38.828	74548	20.502	173
ibm08	41.762	216	43.504	367432	29.985	95661	19.550	256
ibm09	49.943	235	52.683	489388	44.012	105224	23.702	268
ibm10	80.300	377	91.781	795229	79.056	195967	42.668	355

Στο γράφημα που ακολουθεί, απεικονίζεται το hpwl που παράγει για κάθε κύκλωμα ο κάθε αλγόριθμος. Συνολικά, ο τέταρτος αλγόριθμος είναι αυτός που παράγει το μικρότερο hpwl, καθώς προσπαθεί να διατηρήσει τα κελιά όσο το δυνατόν πλησιέστερα στις αρχικές θέσεις που προέκυψαν από τον αλγόριθμο Gordian.



Στο γράφημα που ακολουθεί, απεικονίζεται ο χρόνος εκτέλεσης του κάθε αλγορίθμου για καθένα από τα κυκλώματα. Ο δεύτερος αλγόριθμος είναι αυτός που απαιτεί τον περισσότερο χρόνο, καθώς για κάθε κελί που ανήκει σε ένα net πρέπει να ελεγχθεί αν ανήκει και σε άλλα nets. Επίσης, αρκετά μεγάλο χρόνο εκτέλεσης έχει και ο τρίτος αλγόριθμος, ο οποίος ελέγχει αν ένα κελί συνδέεται με κάποιο pin.



5 ΜΕΛΛΟΝΤΙΚΕΣ ΕΠΕΚΤΑΣΕΙΣ

Οι αλγόριθμοι που παρουσιάστηκαν χρησιμοποίησαν την αρχική διάταξη που προέκυψε από την εφαρμογή του αλγόριθμου Gordian κατά τη διαδικασία του global placement. Ωστόσο, θα μπορούσαμε να χρησιμοποιήσουμε κάποιον άλλο αλγόριθμο και να μελετήσουμε τη συμπεριφορά των αλγορίθμων σε αυτές τις περιπτώσεις.

Επιπλέον, οι αλγόριθμοι χρησιμοποιήθηκαν σε συνδυασμό με τον αλγόριθμο Tetris για το legalization των κελιών. Αντ' αυτού θα μπορούσε να χρησιμοποιηθεί κάποιος άλλος αλγόριθμος.

Επίσης, ο τρίτος αλγόριθμος θα μπορούσε να επεκταθεί έτσι ώστε να μπορεί να εφαρμοστεί και σε κυκλώματα τα οποία περιέχουν κελιά που συνδέονται με παραπάνω από τρία pins.

Τέλος, οι αλγόριθμοι θα μπορούσαν να τροποποιηθούν έτσι ώστε να μπορούν να κάνουν legalization σε σχεδιάσεις που περιέχουν και άλλα layouts, εκτός από standard cells.

6 Αναφορές

[1] J. Kleinhans, G. Sigl, F. Johannes and K. Antreich, "GORDIAN: VLSI placement by quadratic programming and slicing optimization,"Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on , vol.10, no.3, pp.356-365, Mar 1991

[2] D. Hill, "Method and system for high speed detailed placement of cells within an integrated circuit design, April 2002

[3] Natarajan Viswanathan and Chris Chu. FastPlace: Efficient Analytical Placement using Cell Shifting, Iterative Local Refinement and a Hybrid Net Model. In Proc. International Symposium on Physical Design, pages 26-33, 2004. Last modified: Dec 07, 2004 http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.public.iastate.edu/~nataraj/ISPD04_Bench.html

7 Βιβλιογραφία

[1] Place and route https://en.wikipedia.org/wiki/Place_and_route

[2] M.M.M. Sarcar ,K. Mallikarjuna Rao, K. Lalit Narayan (2008), "Computer Aided Design and Manufacturing", PHI Learning Pvt. Ltd

[3] Συστήματα Σχεδιομελέτης και Παραγωγής με Χρήση Η/Υ http://www.cadlab.tuc.gr/courses/cad/chap1.pdf

[4] Static timing analysis https://en.wikipedia.org/wiki/Static_timing_analysis

[5] Design for testing https://en.wikipedia.org/wiki/Design_for_testing

[6] Physical design (electronics) https://en.wikipedia.org/wiki/Physical_design_(electronics)

[7] (Blog) All About VLSI Engineering (BackEnd-Physical Design-Place and Route) http://neeasic.blogspot.gr/

[8] http://vlsi.pro/

[9] Naveen Kumar (2012), "Introduction to routing in VLSI design" (notes) https://www.slideshare.net/NaveenKumar11/vlsi-routing

[10] Charles J. Alpert, Dinesh P. Mehta, Sachin S. Sapatnekar (2008), "Handbook of Algorithms for Physical Design Automation", CRC Press

[11] Sherief Reda, "Physical Design of Digital Integrated Circuits" (notes) http://scale.engin.brown.edu/classes/EN0291S40F06/lecture07.pdf

[12] Rob A. Rutenbar, "VLSI CAD" https://www.coursera.org/learn/vlsi-cad-layout/home/welcome

[13] Dadaliaris Antonios, Reliability-driven placement of integrated circuits, PhD thesis, Computer Science Dept., University of Thessaly, June 2012

[14] Jens Egeblad, Placement Techniques for VLSI Layout Using Sequence-Pair Legalization, Master Thesis, Computer Science Dept., University of Copenhagen, July 2003

[15] Alberto Sangiovanni-Vincentelli, "The Tides of EDA", University of California at Berkeley, 2003

[16] Kenneth Yun, "Cell Design and Layout", University of San Diego (notes)

[17] Stavros K. Ioannidis, Implementation and optimization of an integrated circuit placement algorithm in parallel environment, Master Thesis, Department Of Electrical And Computer Engineering, March 2014

8 Υπόμνημα

Ακολουθεί ο κώδικας των αλγορίθμων που υλοποιήθηκαν σε γλώσσα προγραμματισμού C.

Για την αναπαράσταση των κελιών χρησιμοποιείται το struct:

```
struct cell_t {
    char name[MAX_STRING_LENGTH];
    double centerX;     //position of the center
    double centerY;     //position of the center
    double sizeX;
    double sizeY;
    double coordinates_x;
    double coordinates_y;
    double area;
    short terminal;
    short fixed;
    bool placed;
    char orientation[3];
    int level;
    int index;
    double clique_weight;
    bool visited;
    bool already_checked;
    double distance;
    double displacement;
    struct distance_info_t distance_info[9];
    struct cell_t *next;
    int square_id;
    int degree;
    int net_cnt; //the number of nets that the cell belongs to
    int nets_index[3]; //indices of the nets array
};
```

Για την αναπαράσταση των nets χρησιμοποιείται το struct:

```
struct net_t {
    int connectionsCtr;
    struct cell_t **connections;
    int connectivity;
} *nets;
```

Για την αναπαράσταση των γραμμών χρησιμοποιείται το struct:

```
struct row_t{
    double x1;
    double x2;
    double y1;
    double y2;
    double area;
    double density;
    double left_most;
```

```
    double right_most;
    bool legalized;
};
```

Συναρτήσεις που χρησιμοποιούνται για αρχικοποίηση / ενημέρωση πινάκων και μεταβλητών:

```
/* 1 */
/* Marks all movable cells as "unplaced" and "unchecked". */
void init_cells_info() {
  int i;
  for (i = 0; i < movablesCtr; i++) {
    movable_cells_list[i]->placed = false;
    movable_cells_list[i]->already_checked = false;
    movable_cells_list[i]->net_cnt = 0;
  }
}

/* 2 */
/* Initializes the net ralated info. */
void init_nets_info() {
  int i = 0;

  for (i = 0; i < netsCtr; i++) {
      nets[i].connectivity = 0;
      nets[i].hasPin = false;
  }
}

/* 3 */
/* Initializes the row related info. */
void init_rowslist() {
  int number_of_rows = (int)(coreAreaHeight / stdCellHeight);

  rows_list = (struct row_t **)malloc(number_of_rows * sizeof(struct
row_t *));
  int i;
  for (i = 0; i < number_of_rows; i++) {
    rows_list[i] = (struct row_t *)malloc(sizeof(struct row_t));
    rows_list[i]->x1 = 0.0;
    rows_list[i]->x2 = coreAreaWidth;
    rows_list[i]->y1 = i * stdCellHeight;
    rows_list[i]->y2 = (i * stdCellHeight) + stdCellHeight;
    rows_list[i]->density = 0.0;
    rows_list[i]->area = stdCellHeight * coreAreaWidth;
    rows_list[i]->left_most = 0.0;
    rows_list[i]->right_most = coreAreaWidth;
  }
}

/* 4 */
/* Initializes the arrays used in 4th algorith which contain info about
the rows. */
void init_arrays(int *rowcount, double *rowlength, struct cell_t
**rowcells, int ROWSIZE) {
  int i = 0;
  int number_of_rows = (int)(coreAreaHeight / stdCellHeight);

  for (i = 0; i < number_of_rows; i++) {
    rowcount[i] = 0;
```

```
    rowlength[i] = 0.0;
    rows_list[i]->legalized = false;
  }
  for (i = 0; i < ROWSIZE; i++) {
    rowcells[i] = NULL;
  }
}

/* 5 */
void refresh_cell_coordinates(struct cell_t **cells_array, int count) {
  int j;
  for (j = 0; j < count; j++) {
    cells_array[j]->coordinates_x =
        cells_array[j]->centerX - (cells_array[j]->sizeX / 2.0);
    cells_array[j]->coordinates_y =
        cells_array[j]->centerY - (cells_array[j]->sizeY / 2.0);
  }
}
```

Αλγόριθμος Tetris

```
void basicTetris(double coreHeight, double coreWidth, double cellHeight,
struct cell_t ** movable_cells, int movable_cells_cnt) {
    int i = 0;
    int j = 0;

    double best = 0.0;
    double cost = 0.0;

    int best_row = 0;

    int leftCtr = 0;
    int rightCtr = 0;

    int number_of_rows = (int)(coreHeight / cellHeight);

    quicksort (movable_cells, 0, movable_cells_cnt - 1, 'X');

    for(i=0; i < movable_cells_cnt; i++){
        best = DBL_MAX;

        for(j = 0; j < number_of_rows; j++){
            if(rows_list[j]->left_most + movable_cells[i]->sizeX >
coreWidth){
                continue;
            }

            cost = sqrt(pow((rows_list[j]->left_most +
(movable_cells[i]->sizeX / 2.0) - movable_cells[i]->centerX), 2.0) +
pow((rows_list[j]->y1 + ((rows_list[j]->y2 - rows_list[j]->y1) / 2.0)) -
movable_cells[i]->centerY, 2.0));
            if(cost <= best){
                best = cost;
                best_row = j;
            }
        }

        movable_cells[i]->displacement = abs(movable_cells[i]->centerX -
rows_list[best_row]->left_most - (movable_cells[i]->sizeX / 2.0)) +
abs(movable_cells[i]->centerY - rows_list[best_row]->y1 -
((rows_list[best_row]->y2 - rows_list[best_row]->y1) / 2.0));
```

```
        movable_cells[i]->centerX = rows_list[best_row]->left_most +
(movable_cells[i]->sizeX / 2.0);
        movable_cells[i]->centerY = rows_list[best_row]->y1 +
((rows_list[best_row]->y2 - rows_list[best_row]->y1) / 2.0);
        refresh_cell_coordinates(movable_cells, movable_cells_cnt);
        rows_list[best_row]->left_most = rows_list[best_row]->left_most
+ movable_cells[i]->sizeX;
    }
}
```

Πρώτος αλγόριθμος

```
void legalizer_1 () {
    //init info
    init_cells_info();
    init_nets_info();
    init_rowslist();
    //calculate the number of movable cells in each net
    calculate_net_connectivity();
    //sort nets in descending order by their connectivity number
    netMergeSort();
    //legalization
    legalize_cells();
}

/*  Calculates the number of movable cells for each net
    in the nets array and stores it at connectivity member variable.
*/
void calculate_net_connectivity() {
    int i, j;
    for(i = 0; i < netsCtr; i++) {
        for(j = 0; j < nets[i].connectionsCtr; j++) {
            if(!nets[i].connections[j]->terminal) {
                nets[i].connectivity++;
            }
        }
    }
}

void legalize_cells() {
    int i, j, movables_count;
    struct cell_t ** movables;
    for(i = 0; i < netsCtr; i++) {
        movables_count = 0;
        movables = (struct cell_t **) malloc(nets[i].connectionsCtr *
sizeof(struct cell_t *));
        for(j = 0; j < nets[i].connectionsCtr; j++) {
            if((!nets[i].connections[j]->terminal) &&
(!nets[i].connections[j]->placed)) {
                movables[movables_count] = nets[i].connections[j];
                movables_count++;
                //mark this cell as placed
                nets[i].connections[j]->placed = 1;
            }
        }
        // do not run tetris for trivial nets (with zero-unplaced cells)
        if(movables_count > 0) {
            basicTetris(coreAreaHeight, coreAreaWidth, stdCellHeight,
movables, movables_count);
```

```
            }
        }
        free(movables);
}
```

Δεύτερος αλγόριθμος

```
void legalizer_2() {
    //init info
    init_cells_info();
    init_nets_info();
    init_rowslist();
    //calculate connectivity
    calculate_connectivity_between_nets();
    //sort nets in descending order by their connectivity number
    netMergeSort();
    //legalization
    legalize_cells();
}


void calculate_connectivity_between_nets() {
    //find how many cells each net shares with other nets
    for(int i = 0; i < netsCtr - 1; i++) {
        for(int j = i + 1; j < netsCtr; j++) {
            for(int k = 0; k < nets[i].connectionsCtr; k++) {
                for(int l = 0; l < nets[j].connectionsCtr; l++) {
                    if(strcmp(nets[i].connections[k]->name,
nets[j].connections[l]->name) == 0) {
                        nets[i].connectivity++;
                        nets[j].connectivity++;
                        break;
                    }
                }
            }
        }
    }
}
```

Τρίτος αλγόριθμος

```
void legalizer_3() {
    int number_of_rows = (int)(coreAreaHeight / stdCellHeight);
    int cells_with_pins_cnt = 0; //the number of cells that belong to
nets with pins
    int i = 0, j = 0, k = 0;
    struct cell_t **cells_with_pins;
    struct cell_t **cells_without_pins;

    //initialize info
    init_cells_info();
    init_nets_info();
    init_rowslist();

    //find how many cells belong to nets that have pins
    cells_with_pins_cnt = cells_in_nets_with_pins();
    cells_with_pins = (struct cell_t
**)malloc(cells_with_pins_cnt*sizeof(struct cell_t *));
    //the remaining cells are (movablesCtr - cells_with_pins_cnt)
```

```
    cells_without_pins = (struct cell_t **)malloc((movablesCtr -
cells_with_pins_cnt)*sizeof(struct cell_t *));
    //store cells to arrays
    for(i = 0; i < movablesCtr; i++) {
        if(movable_cells_list[i]->net_cnt > 0) {
            cells_with_pins[j] = movable_cells_list[i];
            j++;
        } else {
            cells_without_pins[k] =  movable_cells_list[i];
            k++;
        }
    }

    //sort cells in cells_with_pins array in ascending order by X
    mergeSort(cells_with_pins, 0, cells_with_pins_cnt);
    //find best position for each cell that belongs to nets with pins
    best_position_for_cells(cells_with_pins, cells_with_pins_cnt);
    //apply tetris for remaining cells
    newTetris(coreAreaHeight, coreAreaWidth, stdCellHeight,
cells_without_pins, movablesCtr - cells_with_pins_cnt);

    free(cells_with_pins);
    free(cells_without_pins);
}

/*  Marks the cells that belong to nets with 1 or 2 pins.
    @return the total number of cells that belong to nets with 1 or 2
pins. Cells that belong to more
    than one net with pin, are only counted once.
*/
int cells_in_nets_with_pins() {
    int i, j, k;
    int pin_cnt = 0; //total number of cells that belong to a net with
pins

    for(i = 0; i < netsCtr; i++) {
        for(j = 0; j < nets[i].connectionsCtr; j++) {
            //if the net has a pin
            if(nets[i].connections[j]->terminal) {
                //go through the list of cells
                for(k = 0; k < nets[i].connectionsCtr; k++) {
                    if(!nets[i].connections[k]->terminal) {
                        //store the index of the net
                        nets[i].connections[k]-
>nets_index[nets[i].connections[k]->net_cnt] = i;
                        //update net counter for the cell
                        nets[i].connections[k]->net_cnt++;
                        //update count that stores total number of cells
that belong to a net with pin
                        if(!nets[i].connections[k]->already_checked) {
                            pin_cnt++;
                            nets[i].connections[k]->already_checked =
true;
                        }
                    }
                }
                break; //because the net has at least one pin
            }
        }
    }
    return(pin_cnt);
```

```
}

/*  Finds the best position for each cell that belongs to nets with 1 or
2 pins.
    @param cells_with_pins an array of cells that belong to nets with
pins
    @param cells_with_pins_cnt the length of the array  */
void best_position_for_cells(struct cell_t **cells_with_pins, int
cells_with_pins_cnt) {
    int i = 0;
    struct cell_t **pins;

    for(i = 0; i < cells_with_pins_cnt; i++) {
        //create array for storing the pins
        pins = (struct cell_t **)malloc(cells_with_pins[i]-
>net_cnt*sizeof(struct cell_t *));
        //find the pin(s) that belongs to the same net with the current
cell
        find_pins(pins, cells_with_pins[i]);
        switch (cells_with_pins[i]->net_cnt) {
            case 1 :
                //if the net has one pin place the cell closer to that
pin
                //place the cell as close as possible to the pin
                place_cell_one_pin(pins, cells_with_pins[i]);
            break;
            case 2:
                //place the cell in (y1 + y2) / 2
                place_cell_two_pins(pins, cells_with_pins[i]);
                break;
            case 3:
                place_cell_three_pins(pins, cells_with_pins[i]);
                break;
            default:
                printf("Pin count %d \n", cells_with_pins[i]->net_cnt);
                break;
        }
        free(pins);
    }
}

void find_pins(struct cell_t **pins, struct cell_t *cur_cell) {
    int j = 0, k = 0;
    struct net_t *cur_net;

    for(j = 0; j < cur_cell->net_cnt; j++) {
        //find the net we are looking for by looking up its index in the
cell info
        cur_net = &nets[cur_cell->nets_index[j]];
        //in that net, find the pins
        for(k = 0; k < cur_net->connectionsCtr; k++) {
            //by checking if the cell is terminal
            if(cur_net->connections[k]->terminal) {
                pins[j] = cur_net->connections[k];
            }
        }
    }
}

void place_cell_one_pin(struct cell_t **pins, struct cell_t *cur_cell) {
    int number_of_rows = (int)(coreAreaHeight / stdCellHeight);
```

```
    //find the cells in that row
    int ROWSIZE = movablesCtr / 30;
    // the number of cells in the row
    int *rowcount = (int *) malloc(number_of_rows*sizeof(int));
    // the length of the cells that belong to the row
    double *rowlength = (double *)
malloc(number_of_rows*sizeof(double));
    // the cells that belong to the row
    struct cell_t **rowcells = (struct cell_t **)
malloc(ROWSIZE*sizeof(struct cell_t *));
    int preferred_row;
    double X_coordinate;

    //find the preferred row for the cell depending on the pin's
orientation
    if(strcmp(pins[0]->orientation, "FS") == 0) {
        //if the pin is at the top of the area
        preferred_row = (int) (coreAreaHeight / stdCellHeight) - 1;
//the last row
        while((rows_list[preferred_row]->right_most -
rows_list[preferred_row]->left_most) < cur_cell->sizeX) {
            preferred_row--;
        }
    }
    else if(strcmp(pins[0]->orientation, "N") == 0) {
        //if the pin is at the bottom of the area
        preferred_row = 0; //the first row
        while((rows_list[preferred_row]->right_most -
rows_list[preferred_row]->left_most) < cur_cell->sizeX) {
            preferred_row++;
        }
    }
    else if(strcmp(pins[0]->orientation, "E") == 0) {
        //if the pin's orientation is E
        preferred_row = pins[0]->centerY / stdCellHeight; //the same row
with the pin
        if( (rows_list[preferred_row]->right_most -
rows_list[preferred_row]->left_most) < cur_cell->sizeX) {
            preferred_row = find_new_row(preferred_row, cur_cell);
        }
        rows_list[preferred_row]->left_most = rows_list[preferred_row]-
>left_most + cur_cell->sizeX;
    }
    else if(strcmp(pins[0]->orientation, "FW") == 0) {
        //if the pin's orientation is FW
        preferred_row = pins[0]->centerY / stdCellHeight; //the same row
with the pin
        //printf("\npreferred row : %d\n%d",
preferred_row,number_of_rows);
        if( (rows_list[preferred_row]->right_most -
rows_list[preferred_row]->left_most) < cur_cell->sizeX) {
            preferred_row = find_new_row(preferred_row, cur_cell);
            //printf("\nnew row : %d\n", preferred_row);
        }
        rows_list[preferred_row]->right_most = rows_list[preferred_row]-
>right_most - cur_cell->sizeX;
    }

    //place the cell in the preferred row
    if(X_coordinate <= coreAreaWidth / 2.0) {
        cur_cell->coordinates_x = rows_list[preferred_row]->left_most;
```

```
        rows_list[preferred_row]->left_most += cur_cell->sizeX;
    }
    else {
        cur_cell->coordinates_x = rows_list[preferred_row]->right_most -
cur_cell->sizeX;
        rows_list[preferred_row]->right_most -= cur_cell->sizeX;
    }
    //cur_cell->coordinates_x = X_coordinate;
    cur_cell->centerX = cur_cell->coordinates_x + (cur_cell->sizeX /
2.0);
    cur_cell->coordinates_y = rows_list[preferred_row]->y1;
    cur_cell->centerY = cur_cell->coordinates_y + (stdCellHeight / 2.0);
    cur_cell->placed = true;
}

void place_cell_two_pins(struct cell_t **pins, struct cell_t *cur_cell)
{
    int number_of_rows = (int)(coreAreaHeight / stdCellHeight);
    int i = 0, preferred_row, neighboring_row;
    double preferred_row_Y;
    double X_coordinate;

    preferred_row_Y = (pins[0]->centerY + pins[1]->centerY) / 2.0;
    preferred_row = (int) preferred_row_Y / stdCellHeight;
    //because the pins are placed outside the area
    if(preferred_row >= number_of_rows - 1) {
        preferred_row = number_of_rows - 1;
        while((rows_list[preferred_row]->right_most -
rows_list[preferred_row]->left_most) < cur_cell->sizeX) {
            preferred_row--;
        }
    }
    if(preferred_row <= 0) {
        preferred_row = 0;
        while((rows_list[preferred_row]->right_most -
rows_list[preferred_row]->left_most) < cur_cell->sizeX) {
            preferred_row++;
        }
    }

    X_coordinate = (pins[0]->centerX + pins[1]->centerX) / 2.0;

    //place the cell in the preferred row
    if(X_coordinate <= coreAreaWidth / 2.0) {
        cur_cell->coordinates_x = rows_list[preferred_row]->left_most;
        rows_list[preferred_row]->left_most += cur_cell->sizeX;
    }
    else {
        cur_cell->coordinates_x = rows_list[preferred_row]->right_most -
cur_cell->sizeX;
        rows_list[preferred_row]->right_most -= cur_cell->sizeX;
    }
    //cur_cell->coordinates_x = X_coordinate;
    cur_cell->centerX = cur_cell->coordinates_x + (cur_cell->sizeX /
2.0);
    cur_cell->coordinates_y = rows_list[preferred_row]->y1;
    cur_cell->centerY = cur_cell->coordinates_y + (stdCellHeight / 2.0);
    cur_cell->placed = true;
}
```

```
void place_cell_three_pins(struct cell_t **pins, struct cell_t
*cur_cell) {
    int number_of_rows = (int)(coreAreaHeight / stdCellHeight);
    int i = 0, preferred_row, neighboring_row;
    double preferred_row_Y;
    double X_coordinate;


    //centroid of the triangle
    preferred_row_Y = (1/3) * (pins[0]->centerY + pins[1]->centerY +
pins[2]->centerY);
    preferred_row = (int) preferred_row_Y / stdCellHeight;

    //because the pins are placed outside the area
   if(preferred_row >= number_of_rows - 1) {
       preferred_row = number_of_rows - 1;
       while((rows_list[preferred_row]->right_most -
rows_list[preferred_row]->left_most) < cur_cell->sizeX) {
           preferred_row--;
       }
   }
   if(preferred_row <= 0) {
       preferred_row = 0;
       while((rows_list[preferred_row]->right_most -
rows_list[preferred_row]->left_most) < cur_cell->sizeX) {
           preferred_row++;
       }
   }

   X_coordinate = (1/3) * (pins[0]->centerX + pins[1]->centerX +
pins[2]->centerX);
   //place the cell in the preferred row
    if(X_coordinate <= coreAreaWidth / 2.0) {
        cur_cell->coordinates_x = rows_list[preferred_row]->left_most;
        rows_list[preferred_row]->left_most += cur_cell->sizeX;
    }
    else {
        cur_cell->coordinates_x = rows_list[preferred_row]->right_most -
cur_cell->sizeX;
        rows_list[preferred_row]->right_most -= cur_cell->sizeX;
    }
    cur_cell->centerX = cur_cell->coordinates_x + (cur_cell->sizeX /
2.0);
    cur_cell->coordinates_y = rows_list[preferred_row]->y1;
    cur_cell->centerY = cur_cell->coordinates_y + (stdCellHeight / 2.0);
    cur_cell->placed = true;

}

int find_new_row(int cur_row, struct cell_t *cur_cell) {
    int number_of_rows = (int)(coreAreaHeight / stdCellHeight);
    int i = 1;
    bool found = false, found1 = false, found2 = false;;
    int best1, best2, best = -1;

    while(!found) {
        if(cur_row - i >= 0) {
            if((rows_list[cur_row-i]->right_most - rows_list[cur_row-i]-
>left_most) > cur_cell->sizeX) {
                best1 = cur_row - i;
                found1 = true;
```

```
            }
        }
        if(cur_row + i < number_of_rows) {
            if((rows_list[cur_row+i]->right_most - rows_list[cur_row+i]-
>left_most) > cur_cell->sizeX) {
                best2 = cur_row + i;
                found2 = true;
            }
        }
        if(found1 && found2) {
            if( (rows_list[best1]->right_most - rows_list[best1]-
>left_most) > (rows_list[best2]->right_most - rows_list[best2]-
>left_most)) {
                best = best1;
            } else {
                best = best2;
            }
            found = true;
        }
        else if(found1) {
            best = best1;
            found = true;
        }
        else if(found2) {
            best = best2;
            found = true;
        }
        else {
            i++;
        }
    }
    return best;
}

void newTetris(double coreHeight, double coreWidth, double cellHeight,
struct cell_t ** movable_cells, int movable_cells_cnt) {
    int i = 0;
    int j = 0;

    double best = 0.0;
    double cost = 0.0;

    int best_row = 0;

    int leftCtr = 0;
    int rightCtr = 0;

    int number_of_rows = (int)(coreHeight / cellHeight);

    quicksort (movable_cells, 0, movable_cells_cnt - 1, 'X');

    for(i=0; i < movable_cells_cnt; i++){
        best = DBL_MAX;

        for(j = 0; j < number_of_rows; j++){
            if( (rows_list[j]->right_most - rows_list[j]->left_most) <
movable_cells[i]->sizeX) {
                continue;
            }
```

```
            cost = sqrt(pow((rows_list[j]->left_most +
(movable_cells[i]->sizeX / 2.0) - movable_cells[i]->centerX), 2.0) +
pow((rows_list[j]->y1 + ((rows_list[j]->y2 - rows_list[j]->y1) / 2.0)) -
movable_cells[i]->centerY, 2.0));
            if(cost <= best){
                best = cost;
                best_row = j;
            }
        }

        movable_cells[i]->displacement = abs(movable_cells[i]->centerX -
rows_list[best_row]->left_most - (movable_cells[i]->sizeX / 2.0)) +
abs(movable_cells[i]->centerY - rows_list[best_row]->y1 -
((rows_list[best_row]->y2 - rows_list[best_row]->y1) / 2.0));
        movable_cells[i]->centerX = rows_list[best_row]->left_most +
(movable_cells[i]->sizeX / 2.0);
        movable_cells[i]->centerY = rows_list[best_row]->y1 +
((rows_list[best_row]->y2 - rows_list[best_row]->y1) / 2.0);
        refresh_cell_coordinates(movable_cells, movable_cells_cnt);
        rows_list[best_row]->left_most = rows_list[best_row]->left_most
+ movable_cells[i]->sizeX;
    }
}
```

Τέταρτος αλγόριθμος

```
void legalizer_4() {
    int i = 0, mid_pos = 0;
    //number of cells of a row that moved to neighboring rows
    int moved_cells = 0;
    int number_of_rows = (int)(coreAreaHeight / stdCellHeight);
    /*  ROWSIZE: The estimated number of cells that could fit in a row.
It should
        be a large enough number but smaller than movablesCtr. */
    int ROWSIZE = movablesCtr / 30;
    // the number of cells in the row
    int *rowcount = (int *) malloc(number_of_rows*sizeof(int));
    // the length of the cells that belong to the row
    double *rowlength = (double *)
malloc(number_of_rows*sizeof(double));
    // the cells that belong to the row
    struct cell_t **rowcells = (struct cell_t **)
malloc(ROWSIZE*sizeof(struct cell_t *));
    double offset = 0.0;

    //init info
    init_cells_info();
    init_rowslist();
    init_arrays(rowcount, rowlength, rowcells, ROWSIZE);

    //place cells to their default row and calculate the length and
number of cells of each row
    calculate_rows(rowlength, rowcount);

    //first, legalize rows where rowlength <= coreAreaWidth (all cells
fit)
    for(i = 0; i < number_of_rows; i++) {
        if(rowlength[i] <= coreAreaWidth) {
            //find the cells that belong to the current row
```

```
            find_cells_in_a_row(rowcells, rowlength, rowcount, i,
ROWSIZE);
            //sort cells according to X-coordinate
            mergeSort(rowcells, 0, rowcount[i]);
            //find the index of the middle cell
            find_middle_cell(rowcells, rowcount[i], rowlength[i],
&offset, &mid_pos);
            //place the middle cell in the middle of the row
            rowcells[mid_pos]->centerX = (coreAreaWidth -
rowlength[i])/2.0 + offset + (rowcells[mid_pos]->sizeX / 2.0);
            rowcells[mid_pos]->coordinates_x = rowcells[mid_pos]-
>centerX - (rowcells[mid_pos]->sizeX / 2.0);
            //legalize the row
            legalize_single_row(rowcells, rowlength, rowcount, i,
mid_pos);
            //mark this row as legalized
            rows_list[i]->legalized = true;
        }
    }
    //then, legalize the remaining rows, by moving cells that don't fit
to neighboring legalized rows
     for(i = 0; i < number_of_rows; i++) {
        if(rowlength[i] > coreAreaWidth) {
            //find the cells that belong to the current row
            find_cells_in_a_row(rowcells, rowlength, rowcount, i,
ROWSIZE);
            //sort cells according to X-coordinate
            mergeSort(rowcells, 0, rowcount[i]);
            //find the index of the middle cell
            find_middle_cell(rowcells, rowcount[i], rowlength[i],
&offset, &mid_pos);
            //place the middle cell in the middle of the row
            rowcells[mid_pos]->centerX = (coreAreaWidth -
rowlength[i])/2.0 + offset + (rowcells[mid_pos]->sizeX / 2.0);
            rowcells[mid_pos]->coordinates_x = rowcells[mid_pos]-
>centerX - (rowcells[mid_pos]->sizeX / 2.0);
            //legalize cells to the left side of the area
            legalize_cells_to_the_left(rowcells, rowlength, rowcount, i,
mid_pos, &moved_cells);
            //legalize cells to the right side of the area
            legalize_cells_to_the_right(rowcells, rowlength, rowcount,
i, mid_pos, &moved_cells);
            rowcount[i] -= moved_cells;
            rows_list[i]->legalized = true;
        }
    }
}

/*  Calculates length and number of cells for all rows.
    Places cells to their "default" row depending on the centerY
coordinate.
    @param rowlength the length of the row
    @param roucount number of cells in the row
*/
void calculate_rows(double *rowlength, int *rowcount) {
    int i;
    int number_of_rows = (int) (coreAreaHeight / stdCellHeight);
    int cur_row = 0, j = 0;
    bool foundRow = false;

    for(i = 0; i < movablesCtr; i++) {
```

```
        cur_row = 0;
        foundRow = false;
        while(cur_row <= number_of_rows && !foundRow) {
            if(movable_cells_list[i]->centerY <= rows_list[cur_row]->y2
&& movable_cells_list[i]->centerY >= rows_list[cur_row]->y1) {
                // if cell's centerY is within the row, place it in this
row
                movable_cells_list[i]->centerY = rows_list[cur_row]->y1
+ (stdCellHeight / 2.0);
                movable_cells_list[i]->coordinates_y =
rows_list[cur_row]->y1;
                // update info related to current row
                rowcount[cur_row]++;
                rowlength[cur_row] += movable_cells_list[j]->sizeX;
                foundRow = true; // stop searching row
            } else {
                // else check next row
                cur_row++;
            }
        }
    }
}

/*  Finds the cells that belong to a row and stores them in rowcells
array.
    @param rowcells an array where we store the lists of cells
    @param rowlength an array where we store the length of each row
    @param rowcount an array where we store the number of cells of each
row
    @param cur_row the row we are examining
    @param ROWSIZE the maximum number of cells that could fit in a row
*/
void find_cells_in_a_row(struct cell_t **rowcells, double *rowlength,
int *rowcount, int cur_row, int ROWSIZE) {
    int i = 0;
    // initialize info for current row
    rowlength[cur_row] = 0.0;
    rowcount[cur_row] = 0;

    for(i = 0; i < movablesCtr; i++) {
        if(movable_cells_list[i]->centerY < 0.0 ||
movable_cells_list[i]->centerY > coreAreaHeight) {
            printf("cell center out of core area\n");
        }
       if(movable_cells_list[i]->centerY == rows_list[cur_row]->y1 +
stdCellHeight/2.0) {
            rowcells[rowcount[cur_row]] = movable_cells_list[i];
            rowcount[cur_row]++;
            rowlength[cur_row] += movable_cells_list[i]->sizeX;
        }
    }
}

/*  Finds the index of middle cell in the given row.
    Places the cells in line until we reach length = coreAreaWidth / 2,
    that is the middle of the line.
    @param rowcells an array where we store the lists of cells
    @param rowcount the number of cells of the row
    @param rowlength the length of the row
    @param offset the length of the cells when placed in a line
    @return the index of the middle cell in the rowcells array
```

```
*/
void find_middle_cell(struct cell_t **rowcells, int rowcount, int
rowlength, double *offset, int *mid_pos) {
    int i = 0;

   (*offset) = 0.0;
   (*mid_pos) = 0;
   for(i = 0; i < rowcount; i++) {
       (*offset) += rowcells[i]->sizeX;
       if((*offset) >= rowlength/2.0) {
           (*offset) = (*offset) - rowcells[i]->sizeX;
           (*mid_pos) = i;
           break;
       }
   }
}

/* Legalizes a single row where rowlength <= coreAreaWidth. */
void legalize_single_row(struct cell_t **rowcells, double *rowlength,
int *rowcount, int cur_row, int mid_pos) {
    int i = 0;

    //update row info
    rows_list[cur_row]->left_most = rowcells[mid_pos]->centerX -
(rowcells[mid_pos]->sizeX / 2.0);
    rows_list[cur_row]->right_most = rowcells[mid_pos]->centerX +
(rowcells[mid_pos]->sizeX / 2.0);
    //first, place remaining cells to the left of the middle cell
    for(i = mid_pos - 1; i >= 0; i--) {
        rowcells[i]->centerX = rows_list[cur_row]->left_most -
(rowcells[i]->sizeX/2.0);
        rowcells[i]->coordinates_x = rowcells[i]->centerX -
(rowcells[i]->sizeX / 2.0);
        rows_list[cur_row]->left_most = rowcells[i]->coordinates_x;
    }
    //then, place remaining cells to the right of the middle cell
    for(i = mid_pos + 1; i < rowcount[cur_row]; i++) {
        rowcells[i]->centerX = rows_list[cur_row]->right_most +
(rowcells[i]->sizeX / 2.0);
        rowcells[i]->coordinates_x = rowcells[i]->centerX -
(rowcells[i]->sizeX / 2.0);
        rows_list[cur_row]->right_most = rowcells[i]->coordinates_x +
rowcells[i]->sizeX;
    }
}

/*  Tries to add a cell to the left_most of the row. If the cell does
not fit, it is placed in a neighboring
    legalized row. */
void legalize_cells_to_the_left(struct cell_t **rowcells, double
*rowlength, int *rowcount, int cur_row, int mid_pos, int *moved_cells) {
    int i = 0, neighboring_row = 0;

    //update row info
    rows_list[cur_row]->left_most = rowcells[mid_pos]->centerX -
(rowcells[mid_pos]->sizeX / 2.0);
    for(i = mid_pos - 1; i >= 0; i--) {
        //if the cell does not fit in the row, find a neighboring row
        if((rows_list[cur_row]->left_most - rowcells[i]->sizeX) < 0.0) {
            find_neighboring_row(cur_row, rowlength, rowcells[i]->sizeX,
&neighboring_row, LEFT);
```

```
            //place cell to the found row
            rowcells[i]->centerY = rows_list[neighboring_row]->y1 +
stdCellHeight/2.0;
            rowcells[i]->coordinates_y = rows_list[neighboring_row]->y1;
            rowcells[i]->coordinates_x = rows_list[neighboring_row]-
>left_most - rowcells[i]->sizeX;
            rowcells[i]->centerX = rows_list[neighboring_row]->left_most
- (rowcells[i]->sizeX/2.0);
            //update row info
            rows_list[neighboring_row]->left_most = rowcells[i]-
>coordinates_x;
            rowcount[neighboring_row]++;
            rowlength[neighboring_row] += rowcells[i]->sizeX;
            rowlength[cur_row] -= rowcells[i]->sizeX;
            moved_cells++;
        }
        else {
            //else place the cell to the current row
            rowcells[i]->centerX = rows_list[cur_row]->left_most -
(rowcells[i]->sizeX/2.0);
            rowcells[i]->coordinates_x = rowcells[i]->centerX -
(rowcells[i]->sizeX / 2.0);
            rows_list[cur_row]->left_most = rowcells[i]->coordinates_x;
        }
    }
}

/*  Tries to add a cell to the right_most of the row. If the cell does
not fit, it is placed in a neighboring
    legalized row. */
void legalize_cells_to_the_right(struct cell_t **rowcells, double
*rowlength, int *rowcount, int cur_row, int mid_pos, int *moved_cells) {
    int i = 0, neighboring_row = 0;

    //update row info
    rows_list[cur_row]->right_most = rowcells[mid_pos]->centerX +
(rowcells[mid_pos]->sizeX / 2.0);
    for(i = mid_pos + 1; i < rowcount[cur_row]; i++) {
        if((rows_list[cur_row]->right_most + rowcells[i]->sizeX) >
coreAreaWidth) {
            find_neighboring_row(cur_row, rowlength, rowcells[i]->sizeX,
&neighboring_row, RIGHT);
            //place cell to the found row
            rowcells[i]->centerY = rows_list[neighboring_row]->y1 +
stdCellHeight/2.0;
            rowcells[i]->coordinates_y = rows_list[neighboring_row]->y1;
            rowcells[i]->centerX = rows_list[neighboring_row]-
>right_most + (rowcells[i]->sizeX/2.0);
            rowcells[i]->coordinates_x = rows_list[neighboring_row]-
>right_most;
            //update row info
            rows_list[neighboring_row]->right_most = rowcells[i]-
>coordinates_x + rowcells[i]->sizeX;
            rowcount[neighboring_row]++;
            moved_cells++;
            rowlength[neighboring_row] += rowcells[i]->sizeX;
            rowlength[cur_row] -= rowcells[i]->sizeX;
        }
        else {
            rowcells[i]->centerX = rows_list[cur_row]->right_most +
(rowcells[i]->sizeX/2.0);
```

```
            rowcells[i]->coordinates_x = rowcells[i]->centerX -
(rowcells[i]->sizeX / 2.0);
            rows_list[cur_row]->right_most = rowcells[i]->coordinates_x
+ rowcells[i]->sizeX;
        }
    }
}

/*  Finds the first available legalized row above or below the current
row that has room for a cell of sizeX.
    (Legalized rows are rows that have been legalized in the previous
step of the algorithm and have rowlength <= coreAreaWidth,
    so they may have room for more cells.)
    @param cur_row the current row
    @param rowlength an array where we store the length of each row
    @param sizeX the size of the cell to be placed
    @param neighboring_row for storing the index of the row where the
cell should be placed
    @param side an integer indicating that we are placing a cell to the
left/right of the middle cell in order to check if
*/
void find_neighboring_row(int cur_row, double *rowlength, int sizeX, int
*neighboring_row, int side) {
    int number_of_rows = (int)(coreAreaHeight / stdCellHeight);
    double min, min1, min2;
    int i, best_row, best1, best2;
    int limit, offset, row_up, row_down;
    bool foundRow, foundUp, foundDown;

    //init values
    min1 = min2 = rowlength[cur_row];
    best1 = best2 = cur_row;
    foundUp = foundDown = false;
    offset = 1;
    limit = 4;
    foundRow = false;

    while(!foundRow) {
        //check for room in cur_row ± i where i = [offset, limit]
        for(i = offset; i < limit; i++) {
            //first check cur_row + i
            if(cur_row + i < number_of_rows && !foundUp) {
                //if we are searching the left side of the area
                if(side == LEFT) {
                    if(rows_list[cur_row + i]->legalized &&
rows_list[cur_row + i]->left_most - sizeX >= 0) {
                        min1 = rowlength[cur_row + i];
                        best1 = cur_row + i;
                        foundUp = true;
                    }
                }
                //if we are searching the right side of the area
                if(side == RIGHT) {
                    if(rows_list[cur_row + i]->legalized &&
rows_list[cur_row + i]->right_most + sizeX <= coreAreaWidth) {
                        min1 = rowlength[cur_row + i];
                        best1 = cur_row + i;
                        foundUp = true;
                    }
                }
            }
```

```
            //then check cur row - i
            if(cur_row - i >= 0 && !foundDown) {
                if(side == LEFT) {
                    if(rows_list[cur_row - i]->legalized &&
rows_list[cur_row - i]->left_most - sizeX >= 0) { //? <= ?
                        min2 = rowlength[cur_row - i];
                        best2 = cur_row - i;
                        foundDown = true;
                    }
                }
                if(side == RIGHT) {
                    if(rows_list[cur_row - i]->legalized &&
rows_list[cur_row - i]->right_most + sizeX <= coreAreaWidth) {
                        min2 = rowlength[cur_row - i];
                        best2 = cur_row - i;
                        foundDown = true;
                    }
                }
            }
            if(foundUp || foundDown) {
                break;
            }
        }
        //if no row was found, increase offset and limit values and
search again
        if(!foundUp && !foundDown) {
            offset = limit;
            limit += 2;
            foundRow = false;
        }
        else {
            foundRow = true;
            //if 2 rows where found, choose the one with the smallest
rowlength
            if(foundUp && foundDown) {
                if(min1 < min2) {
                    best_row = best1;
                }
                else {
                    best_row = best2;
                }
            }
            else if(foundUp) {
                best_row = best1;
            }
            else if(foundDown) {
                best_row = best2;
            }
            else {
                printf("best row not found\n");
            }
        }
    }
    if(best_row < 0 || best_row > number_of_rows) {
        printf("best row out of area %d\n", best_row);
    }

    (*neighboring_row) = best_row;
}
```