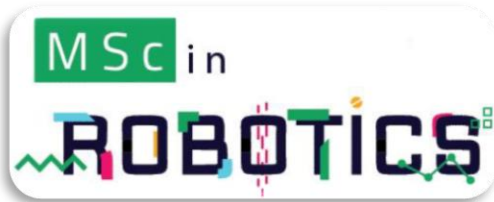
ΔΙΕΘΝΕΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΕΛΛΑΔΟΣ



ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ,
ΥΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Εύρωστη εκτίμηση από μετρήσεις γραμμικών αισθητήρων

Βασίλειος Αλληλόμης

Επιβλέπων:
Ιωάννης Βουρβουλάκης
Επίκουρος Καθηγητής ΔΙ.ΠΑ.Ε.

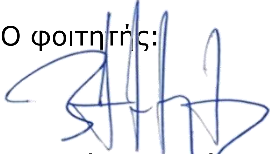
Σέρρες, 2021

ΥΠΕΥΘΥΝΗ ΔΗΛΩΣΗ ΦΟΙΤΗΤΗ:

Ο κάτωθι υπογεγραμμένος φοιτητής, έχοντας επίγνωση των συνεπειών του Νόμου περί λογοκλοπής, δηλώνω υπεύθυνα ότι είμαι συγγραφέας αυτής της Διπλωματικής Εργασίας, αναλαμβάνοντας την ευθύνη επί ολοκλήρου του κειμένου, έχω δε αναφέρει στην Βιβλιογραφία όλες τις πηγές τις οποίες χρησιμοποίησα.

Δηλώνω επίσης ότι, οποιοδήποτε στοιχείο ή κείμενο το οποίο έχω ενσωματώσει στην εργασία μου προερχόμενο από βιβλία, άλλες εργασίες ή το διαδίκτυο, γραμμένο επακριβώς ή παραφρασμένο, το έχω πλήρως αναγνωρίσει ως πνευματικό έργο άλλου συγγραφέα και έχω αναφέρει ανελλιπώς το όνομά του και την πηγή προέλευσης.

Ο φοιτητής:



Αλληλόμης Βασίλειος

Abstract

Robust estimation is the problem of defining a model based on a set of measurements. A common algorithm used for robust estimation is the RANSAC algorithm. It is an iterative algorithm in which all the elements of the set are checked to verify a model that has been identified from a random sample. Using a number of different random samples, different models are identified, of which the one that verifies the largest number of samples in the set is considered the most robust. The most common use of RANSAC is to estimate a line from a set of points. However, the algorithm is met in image processing algorithms such as the removal of erroneous matches of similar images, image alignment, eye tracking, etc. The controller was designed in such a way that it can be used as a component by other robotic vision applications. A testbench was also developed to verify the functional operation of the component.

In the context of this work, the design of a controller in VHDL that implements the RANSAC algorithm for the case of line estimation from a set of points was realized. The design receives as input the coordinates of the points and extracts the a, b and c parameters of the best line fit.

We compare two implementations. In the first one, we include asynchronous calculation of parameters and inliers check in one pulse clock. In the second one, random samples and verification for inliers are performed using sequential structures. The comparison will essentially include the proportion of time required for the reserved resources.

Finally, a test bench was developed to check the proper operation of the component.

Περίληψη

Ως εύρωστη εκτίμηση (robust estimation) χαρακτηρίζεται το πρόβλημα του προσδιορισμού ενός μοντέλου βασιζόμενου σε ένα σύνολο μετρήσεων. Ένας διαδεδομένος αλγόριθμος που χρησιμοποιείται για την εύρωστη εκτίμηση είναι ο αλγόριθμος RANSAC. Αποτελεί έναν επαναληπτικό αλγόριθμο κατά τον οποίο όλα τα στοιχεία του συνόλου ελέγχονται αν επαληθεύουν κάποιο μοντέλο, το οποίο έχει προσδιοριστεί από ένα τυχαίο δείγμα. Χρησιμοποιώντας έναν αριθμό από διαφορετικά τυχαία δείγματα προσδιορίζονται διαφορετικά μοντέλα από τα οποία θεωρείται ως πιο εύρωστο αυτό που επαληθεύει τον μεγαλύτερο αριθμό δειγμάτων του συνόλου. Η κλασικότερη εφαρμογή του RANSAC είναι η εκτίμηση μιας ευθείας από ένα σύνολο σημείων. Ωστόσο, ο αλγόριθμος βρίσκει εφαρμογή σε αλγορίθμους επεξεργασίας εικόνας όπως στην απομάκρυνση εσφαλμένων αντιστοιχιών παρόμοιων εικόνων, στην ευθυγράμμιση εικόνων, στην ιχνηλάτηση οφθαλμού κλπ.

Στα πλαίσια της παρούσας εργασίας πραγματοποιήθηκε ο σχεδιασμός ελεγκτή σε VHDL που υλοποιεί τον αλγόριθμο RANSAC για την περίπτωση εκτίμησης ευθείας από σύνολο σημείων. Ο ελεγκτής σχεδιάστηκε με τέτοιο τρόπο ώστε να μπορεί να χρησιμοποιηθεί ως component από άλλες εφαρμογές ρομποτικής όρασης. Το σχεδιαζόμενο μοντέλο λαμβάνει ως είσοδο τα σημεία από τα οποία ζητείται να υπολογιστεί η ευθεία και παράγει ως έξοδο τις παραμέτρους a, b και c της βέλτιστης ευθείας.

Επίσης, ήταν επιθυμητό να συγκριθούν δύο υλοποιήσεις. Ασύγχρονος υπολογισμός των παραμέτρων σε παλμό 1 ρολογιού, και συγχρονισμένος διαβάζοντας τις μετρήσεις μία προς μία. Η σύγκριση περιλάμβανε το χρόνο και τους δεσμευμένους πόρους που απαιτούνται στις δύο υλοποιήσεις.

Τέλος, αναπτύχθηκε αρχείο δοκιμών (test bench) για τον έλεγχο της σωστής λειτουργίας του component.

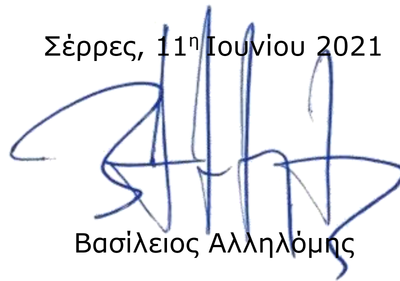
Πρόλογος & Ευχαριστίες

Αυτή η Εργασία είναι μέρος του Μεταπτυχιακού Προγράμματος στη Ρομποτική, στο Διεθνές Πανεπιστήμιο της Ελλάδος (ΔΙ.ΠΑ.Ε.), τμήμα Μηχανικών Πληροφορικής, Υπολογιστών και Τηλεπικοινωνιών (Σέρρες). Η Εργασία ολοκληρώθηκε την άνοιξη του 2021 και ισοδυναμεί με 30 ECTS.

Σκοπός της εργασίας είναι η Σχεδίαση Συστήματος Υψηλών Επιδόσεων για την εφαρμογή του αλγορίθμου RANSAC και την εύρωστη εκτίμηση γραμμικού μοντέλου ευθείας.

Σε αυτό το σημείο θα ήθελα αρχικά να ευχαριστήσω τον επιβλέποντα καθηγητή μου, κ. Ιωάννη Βουρβουλάκη για την εμπιστοσύνη που έδειξε προς το πρόσωπό μου κατά την ανάθεση της παρούσας διπλωματικής καθώς και για την πολύ σημαντική υποστήριξη κατά την ανάπτυξή της. Οι συμβουλές του και οι γνώσεις του πάνω στο θέμα της διπλωματικής αποδείχθηκαν παραπάνω από χρήσιμες και με οδήγησαν αποφασιστικά και αποτελεσματικά στο να την φέρω εις πέρας. Τέλος θα ήθελα να ευχαριστήσω και την οικογένεια μου για την αμέριστη συμπαράστασή της καθ’ όλη τη διάρκεια των σπουδών μου.

Σέρρες, 11η Ιουνίου 2021



Βασίλειος Αλληλόμης

Περιεχόμενα

1 Εισαγωγή – Δομή της Εργασίας 1

1.1 Εισαγωγή 1

1.2 Δομή της Εργασίας 2

2 Εξίσωση Ευθείας - Αλγόριθμος RANSAC 3

2.1 Γενική εξίσωση ευθείας Ax+By+C=0 3

2.2 Αλγόριθμος RANSAC 4

2.2.1 Εισαγωγή στον αλγόριθμο RANSAC 4

2.2.2 Ανάλυση RANSAC 4

2.2.3 Ανάπτυξη και βελτιώσεις 11

2.3 Αλγόριθμος RANSAC & Matlab 12

3 Μεθοδολογία & σχεδιασμός ενσ/νων συσ/των FPGA 14

3.1 Εισαγωγή 14

3.2 Παράλληλα κυκλώματα επεξεργασίας 15

3.2.1 Επεξεργαστής ψηφιακού σήματος (DSP) 15

3.2.2 Μονάδα επεξεργασίας γραφικών (GPU) 15

3.2.3 Ολοκληρωμένο κύκλωμα ειδικής εφαρμογής (ASIC) 16

3.2.4 Field-Programmable Gate Arrays (FPGAs) 16

3.2.4.1 Η Αρχιτεκτονική των FPGAs 21

3.2.4.1.1 LUTs, Flip-Flops, CLBs 22

3.2.4.1.2 Hard Blocks 24

3.2.4.2 Διασυνδέσεις FPGA 24

3.2.5 Σύγκριση κυκλωμάτων παράλληλης επεξεργασίας 25

3.3 Σχεδιασμός με FPGAς 25

3.3.1 Γλώσσα περιγραφής υλικού (HDL) 27

3.3.2 Setup και Hold 27

3.3.3 Pipelining 28

3.3.4 Λειτουργίες Floating Point σε FPGA 29

3.3.5 Διαχείριση μνήμης FPGA 30

3.4 Ενσωματωμένοι επεξεργαστές FPGA 31

3.4.1 NIOS II 32

3.4.2 Λειτουργικό σύστημα πραγματικού χρόνου (RTOS) 33

3.5 Αρχιτεκτονικές Bus FPGA 33

3.6 Ροή ανάπτυξης FPGA 34

3.6.1 Ροή HLS 34

3.6.2 Διαδικασία σχεδίασης κώδικα RTL 35

3.6.3 Προσομοίωση συστήματος ... 38

4 Γλώσσα περιγραφής υλικού VHDL ... 39

4.1 Εισαγωγή ... 39

4.1.1 Ιστορία Γλωσσών Περιγραφής Υλικού ... 40

4.1.2 Επίπεδα Σχεδίασης ... 43

4.1.3 Η ροή σύγχρονου ψηφιακού σχεδιασμού ... 44

4.2 Δομή VHDL ... 46

4.2.1 Τύποι δεδομένων ... 46

4.2.1.1 Αριθμητικοί τύποι ... 46

4.2.1.2 Τύποι εύρους ... 46

4.2.1.3 Φυσικοί τύποι ... 47

4.2.1.4 Τύποι διανυσμάτων ... 47

4.2.1.5 Απαριθμητοί τύποι ... 47

4.2.1.6 Τύπος συστοιχίας ... 48

4.2.1.7 Δευτερεύοντες Τύποι ... 48

4.3 Κατασκευή μοντέλου VHDL ... 48

4.3.1 Βιβλιοθήκες ... 49

4.3.2 Entity ... 49

4.3.3 Architecture ... 50

4.3.3.1 Δηλώσεις σήματος ... 50

4.3.3.2 Δηλώσεις σταθεράς ... 50

4.3.3.3 Δηλώσεις component ... 51

4.4 Μοντελοποίηση ταυτόχρονης λειτουργίας ... 51

4.4.1 Τελεστές (Operators) VHDL ... 51

4.4.1.1 Τελεστής ανάθεσης ... 51

4.4.1.2 Λογικοί τελεστές ... 51

4.4.1.3 Αριθμητικοί τελεστές ... 52

4.4.1.4 Τελεστές σύγκρισης ... 52

4.4.1.5 Τελεστές ολίσθησης ... 52

4.4.1.6 Τελεστές συνένωσης ... 53

4.5 Ταυτόχρονη ανάθεση σήματος με λογικούς τελεστές ... 53

4.6 Εκχωρήσεις υπό όρους ... 53

4.7 Επιλεγμένες αναθέσεις σήματος ... 53

4.8 Δομικός Σχεδιασμός και Ιεραρχία ... 54

4.8.1 Components ... 54

4.8.1.1 Components ... 54

4.8.1.2 Components Instantiation ... 54

4.9 Μοντελοποίηση ακολουθιακής λειτουργίας 55
4.9.1 Η διαδικασία (process) 55
4.9.1.1 Λίστες ευαισθησίας........ 55
4.9.1.2 Δηλώσεις αναμονής 56
4.9.1.3 Μεταβλητές........ 57
4.9.2 Κατασκευές υπό συνθήκη 57
4.9.2.1 Δηλώσεις if/then........ 57
4.9.2.2 Δηλώσεις case........ 57
4.9.2.3 Περιορισμένοι βρόχοι 58
4.9.2.4 While Βρόχοι 58
4.9.2.5 For Βρόχοι 58
4.9.3 Χαρακτηριστικά σήματος........ 59
4.10 Βασικά Πακέτα 60
4.10.1 STD_LOGIC_1164 60
4.10.1.1 STD_LOGIC_1164 Λογικοί τελεστές........ 60
4.10.1.2 STD_LOGIC_1164 Λειτουργίες ανίχνευσης άκρων 60
4.10.1.3 STD_LOGIC_1164 Λειτουργίες μετατροπής τύπων 61
4.10.2 NUMERIC_STD 61
4.10.2.1 NUMERIC_STD Αριθμητικές συναρτήσεις........ 61
4.10.2.2 NUMERIC_STD Λογικές συναρτήσεις 62
4.10.2.3 NUMERIC_STD Συγκριτικές συναρτήσεις........ 62
4.10.2.4 NUMERIC_STD Συναρτήσεις ανίχνευσης άκρων 62
4.10.2.5 NUMERIC_STD Συναρτήσεις μετατροπής........ 62
4.10.2.6 NUMERIC_STD Συναρτήσεις μετάδοσης τύπου 62
4.10.3 TEXTIO και STD_LOGIC_TEXTIO........ 63
4.11 Test-Benches 64
4.11.1 Overview 64
4.12 Μοντελοποίηση FSM (finite state machines) 65
4.12.1 Παράδειγμα διαδικασίας σχεδίασης FSM 65
4.12.1.1 Μοντελοποίηση των finite state machines 65
4.12.1.2 Η Next State λογική διαδικασία 65
4.12.1.3 Η Output λογική διαδικασία........ 66
4.12.2 Παράδειγμα σχεδίασης FSM........ 66
5 Σύνθεση - Προσομοίωση RANSAC........ 68
5.1 Μέθοδος 1 - Pipelined αρχιτεκτονική σχεδίαση (σύγχρονη)........ 68
5.2 Μέθοδος 2 - Παράλληλη αρχιτεκτονική σχεδίαση (ασύγχρονη)........ 77
5.3 Αποτελέσματα και Ανάλυση 79

a. Διαδικασία Προσομοίωσης 79
b. Αποτελέσματα Προσομοίωσης 80
5.4 Συμπεράσματα 98
5.5 Μελλοντική Εργασία 100
Παράρτημα 101
Βιβλιογραφία 155

Κατάλογος σχημάτων

Σχήμα 1 - Γενική αρχιτεκτονική FPGA 1
Σχήμα 2 - Ευθεία από 2 σημεία 3
Σχήμα 3 - Fitting RANSAC 4
Σχήμα 4 - RANSAC 5
Σχήμα 5 - Fitting RANSAC 5
Σχήμα 6 - Βήματα εκτέλεσης αλγορίθμου RANSAC (α) 5
Σχήμα 7 - Βήματα εκτέλεσης αλγορίθμου RANSAC (β), (γ) 6
Σχήμα 8 - Εκτίμηση αποτελέσματος από την εκτέλεση του αλγορίθμου RANSAC 9
Σχήμα 9 - Νόμος Moore 14
Σχήμα 10 - FPGA 16
Σχήμα 11 - SLICE Virtex 6 μπλόκ διάγραμμα 17
Σχήμα 12 - Virtex 6 αρχιτεκτονική στήλης 18
Σχήμα 13 - DSP48 μπλόκ διάγραμμα 18
Σχήμα 14 - GTP μπλόκ διαγράμματα 20
Σχήμα 15 - Σύγκριση μεταξύ hardware και software execution 20
Σχήμα 16 - FPGA XC2064 21
Σχήμα 17 - Διαμορφώσιμο μπλοκ λογικής (CLB) FPGA 22
Σχήμα 18 - 3-LUT 22
Σχήμα 19 - Interconnect Architecture 23
Σχήμα 20 - I/O Blocks 26
Σχήμα 21 - Αρχιτεκτονική τύπου island που εφαρμόζεται στα σημερινά FPGAs 27
Σχήμα 22 - Σχεδιασμός FPGA 25
Σχήμα 23 - Ιεραρχία αρχείων HDL 26
Σχήμα 24 - Setup και Hold 27
Σχήμα 25 - Pipelined vs. not pipelined 28
Σχήμα 26 - Floating point addition latency, συχνότητα λειτουργίας και πόροι 30
Σχήμα 27 - NIOS II μπλόκ διάγραμμα 32
Σχήμα 28 - Interconnection Bus 33
Σχήμα 29 - Διάγραμμα ροής πληροφορίας σε FPGA 39
Σχήμα 30 - Σημαντικά ορόσημα στην προώθηση της ψηφιακής λογικής και των HDLs 42
Σχήμα 31 - Επίπεδα αφαιρετικής σχεδίασης 43
Σχήμα 32 - Ροή ψηφιακής σχεδίασης 44
Σχήμα 33 - Κλασική ροή ψηφιακής σχεδίασης 45
Σχήμα 34 - Ανατομία ενός αρχείου VHDL 48
Σχήμα 35 - Περιγραφή μοντέλου και ορισμός οντότητας ενός δυαδικού μετρητή 66
Σχήμα 36 - Αρχιτεκτονική δυαδικού μετρητή & προσομοίωση 67
Σχήμα 37 - parameters.vhd 68
Σχήμα 38 - parameters rtl 69
Σχήμα 39 - comparator.vhd 69
Σχήμα 40 - comparator rtl 70
Σχήμα 41 - line_comparator.vhd 70
Σχήμα 42 - line_comparator rtl (a) 71
Σχήμα 43 - line_comparator rtl (b) 71
Σχήμα 44 - points_ram.vhd 71
Σχήμα 45 - FSM rtl 72
Σχήμα 46 - Διαδικασία FSM 73
Σχήμα 47 - FSM output 73
Σχήμα 48 - Έλεγχοι "0001" κατάστασης 74
Σχήμα 49 - Μετάβαση στην επόμενη κατάσταση 75
Σχήμα 50 - Διάγραμμα main multi clock 76

Σχήμα 51 - Διάγραμμα main one clock....78
Σχήμα 52 - Quartus Prime 79

Κατάλογος πινάκων

Πίνακας 1 - Ψευδοκώδικας RANSAC 7
Πίνακας 2 - Αριθμός Κ επαναλήψεων.... 8
Πίνακας 3 - Κώδικας Matlab για υλοποίηση RANSAC....12
Πίνακας 4 - Σύγκριση κυκλωμάτων παράλληλης επεξεργασίας 25
Πίνακας 5 - Σημειώσεις VHDL....46
Πίνακας 6 - Αριθμητικοί Τύποι 46
Πίνακας 7 - Τύποι Εύρους....46
Πίνακας 8 - Φυσικοί Τύποι 47
Πίνακας 9 - Τύποι Διανυσμάτων....47
Πίνακας 10 - Λογικοί τελεστές....51
Πίνακας 11 - Αριθμητικοί τελεστές 52
Πίνακας 12 - Τελεστές σύγκρισης 52
Πίνακας 13 - Τελεστές ολίσθησης 52
Πίνακας 14 - Χαρακτηριστικά σήματος....59
Πίνακας 15 - STD_LOGIC_1164 Λειτουργίες μετατροπής τύπων 61
Πίνακας 16 - NUMERIC_STD Συναρτήσεις μετατροπής....62
Πίνακας 17 - NUMERIC_STD Συναρτήσεις μετάδοσης τύπου 62
Πίνακας 18 - Ψευδοκώδικας FSM 73

Συντομογραφίες

IHU	International Hellenic University
FPGA	Field Programmable Gate Array
VHDL	VHSIC Hardware Description Language
SDD	Segment Display Decoder
RANSAC	Random Sample Consensus
ALU	Arithmetic Logic Units
ALM	Adaptive Logic Module
LE	Logic Element

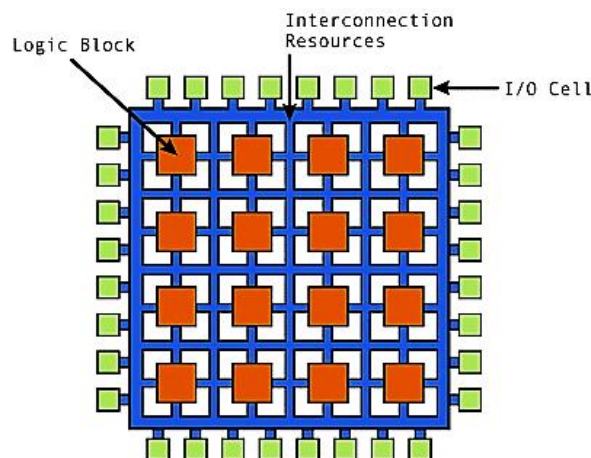
1 Εισαγωγή – Δομή της Εργασίας

1.1 Εισαγωγή

Ο αλγόριθμος Random Sample Consensus (RANSAC) χρησιμοποιείται συνήθως σε πολλές εργασίες εκτίμησης, ειδικά σε εφαρμογές όρασης υπολογιστή λόγω της απλότητάς του, με τον οποίο γίνεται εκτίμηση των παραμέτρων ενός στατιστικού μοντέλου από ένα σύνολο δεδομένων σημείων. Αυτή η εργασία παρουσιάζει μια υλοποίηση σχεδιασμού υλικού / λογισμικού του αλγορίθμου RANSAC για εκτίμηση ευθείας σε ολοκληρωμένα κυκλώματα FPGA.

Η υλοποίηση αλγορίθμων εύρωστης εκτίμησης πάνω σε FPGA από την άλλη, απολαμβάνει τεράστια δημοτικότητα στην επεξεργασία εικόνας, λόγω της υψηλής απόδοσης.

Κάθε κατασκευαστής FPGA έχει τη δική του αρχιτεκτονική FPGA, αλλά σε γενικές γραμμές είναι όλες παραλλαγές της μορφής που απεικονίζεται στο Σχήμα 1. Η αρχιτεκτονική αποτελείται από ρυθμιζόμενα μπλοκ λογικής, διαμορφώσιμα μπλοκ εισόδου / εξόδου και προγραμματιζόμενες διασυνδέσεις. Επίσης, υπάρχει κύκλωμα ρολογιού για την οδήγηση των σημάτων ρολογιού σε κάθε λογικό μπλοκ. Επιπλέον πόροι λογικής όπως LUT, μνήμη και αποκωδικοποιητές, μπορεί επίσης να είναι διαθέσιμοι. Οι τρεις βασικοί τύποι προγραμματιζόμενων στοιχείων για ένα FPGA είναι: static RAM, anti-fuses, και flash EPROM.



Σχήμα 1 – Γενική αρχιτεκτονική FPGA
από http://pldworld.kr/html/technote/pldesignline/allaboutfpgas-bobz.htm

Αναφορικά με την αρθρογραφία επί του αντικειμένου, αυτή εστιάζεται κυρίως σε υλοποιήσεις μηχανικής όρασης και συγκεκριμένα, σε εφαρμογές αντιστοιχιών παρόμοιων εικόνων [1-5], εκτίμηση γεωμετρίας σε πραγματικό χρόνο [6], εύρωστη εκτίμηση έλλειψης [7-9] κ.λ.π.

1.2 Δομή της Εργασίας

Αυτή η διατριβή χωρίζεται σε 5 κεφάλαια, τα οποία οργανώνονται ως εξής:

Το Κεφάλαιο 1 αποτελεί μια εισαγωγή αναφορικά με τον σκοπό αυτής της Εργασίας.

Το Κεφάλαιο 2 περιγράφει την ιδέα πίσω από την υλοποίηση αυτής της Εργασίας καθώς και τον αλγόριθμο RANSAC που θα την εκτελέσει.

Το Κεφάλαιο 3 μας εισάγει στην μεθοδολογία και το σχεδιασμό των ενσωματωμένων συστημάτων που βασίζονται σε FPGA.

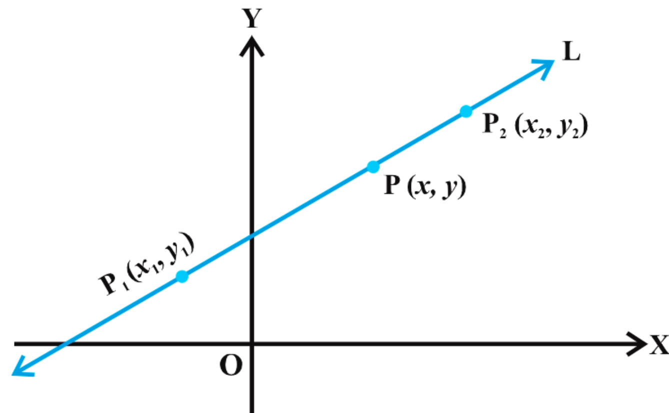
Το Κεφάλαιο 4 εξηγεί την γλώσσα περιγραφής υλικού VHDL με βάση την οποία γίνεται ο σχεδιασμός του μοντέλου.

Το Κεφάλαιο 5 εξηγεί την σύνθεση και παρουσιάζει τα αποτελέσματα της προσομοίωσης. Γίνεται αναφορά σε ορισμένες ιδέες για μελλοντικές βελτιώσεις με βάση τα ευρήματα της παρούσας εργασίας.

2 Εξίσωση Ευθείας - Αλγόριθμος RANSAC

2.1 Γενική εξίσωση ευθείας Ax+By+C=0

Έστω μια ευθεία L που διέρχεται από δύο δεδομένα σημεία P1 (x_1, y_1) και P2 (x_2, y_2). Έστω το σημείο P (x, y) ένα τρίτο σημείο στην L (Σχήμα 2).



Σχήμα 2 – Ευθεία από 2 σημεία

Τα τρία σημεία P_1, P_2 και P είναι συγγραμμικά, επομένως, έχουμε κλίση P_1P = κλίση P_1P_2 Έτσι, η εξίσωση της γραμμής που διέρχεται από τα σημεία (x_1, y_1) και (x_2, y_2) δίνεται από την σχέση (2.1) & (2.2):

$$\frac{y - y1}{x - x1} = \frac{y2 - y1}{x2 - x1} \tag{2.1}$$

ή

$$y - y1 = \frac{y2 - y1}{x2 - x1}(x - x1) \tag{2.2}$$

Στο ίδιο αποτέλεσμα καταλήγουμε εάν θεωρήσουμε οτι ένα σημείο P:(x, y) βρίσκεται στη γραμμή που συνδέει τα P_1 και P_2, εάν και μόνο εάν η περιοχή του παραλληλογράμμου με τις πλευρές P_1P_2 και P_1P είναι μηδέν. Αυτό μπορεί να εκφραστεί ως ορίζουσα 2^{ης} τάξης [10] (2.3):

$$\begin{vmatrix} x2 - x1 & x - x1 \\ y2 - y1 & y - y1 \end{vmatrix} = 0 \Leftrightarrow (y1 - y2)x + (x2 - x1)y + x1y2 - x2y1 = 0 \tag{2.3}$$

από αναγωγή παίρνουμε τις σχέσεις (2.4) έως (2-6):

$$A = y1 - y2 \tag{2.4}$$
$$B = x2 - x1 \tag{2.5}$$
$$C = x1y2 - x2y1 \tag{2.6}$$

Η απόσταση του σημείου M_0 από την ευθεία L [11] δίνεται από τον τύπο (2.7):

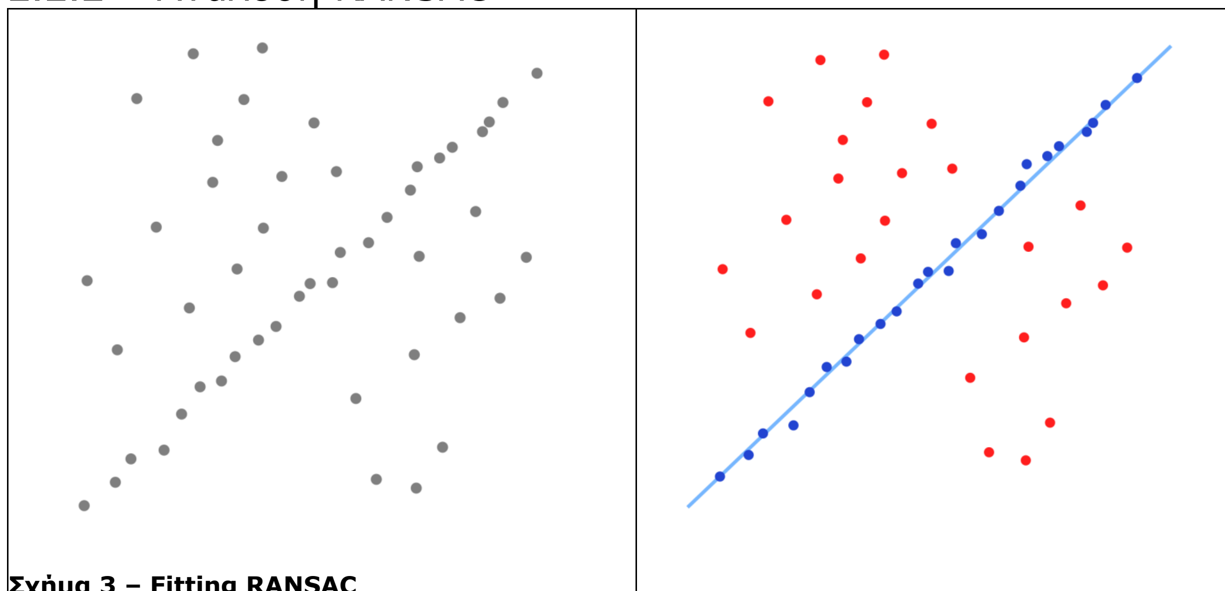
$$d(Mo, L) = \frac{|Axo + Byo + \Gamma|}{\sqrt{A^2 + B^2}} \tag{2.7}$$

2.2 Αλγόριθμος RANSAC

2.2.1 Εισαγωγή στον αλγόριθμο RANSAC

Ο αλγόριθμος RANSAC (RANdom SAmple Consensus) είναι ένας επαναληπτικός αλγόριθμος που εκτιμά τις παραμέτρους του μαθηματικού μοντέλου από ένα σύνολο δεδομένων που περιέχουν "outliers". Τα "Outliers" αναφέρονται γενικά στο θόρυβο στα δεδομένα, όπως αναντιστοιχίες στο ταίριασμα και ακραίες τιμές. Εισήχθη για πρώτη φορά από τους Fischler και Bolles το 1981 [12] στο SRI International για την επίλυση του προβλήματος LDP (Location Determination Problem - Πρόβλημα προσδιορισμού θέσης). Είναι ένας επαναληπτικός, μη ντετερμινιστικός αλγόριθμος, που κατά μία έννοια, θα παράγει ένα λογικό αποτέλεσμα υπό μια ορισμένη πιθανότητα p, η οποία αυξάνεται με την αύξηση του αριθμού των επαναλήψεων. Η βασική προϋπόθεση του RANSAC είναι η παρουσία στο σύνολο δειγμάτων, τόσο εκείνων που ταιριάζουν στο μοντέλο (inliers), όσο και εκείνων που διαφέρουν (outliers). Πραγματοποιείται ο υπολογισμός υποθέσεων μοντέλων από τυχαία δειγματοληπτικά σύνολα δεδομένων και στη συνέχεια γίνεται η επαλήθευση των υποθέσεων στα υπόλοιπα δεδομένα. Η υπόθεση που συγκεντρώνει την υψηλότερη συναίνεση επιλέγεται ως η τελική λύση.

2.2.2 Ανάλυση RANSAC



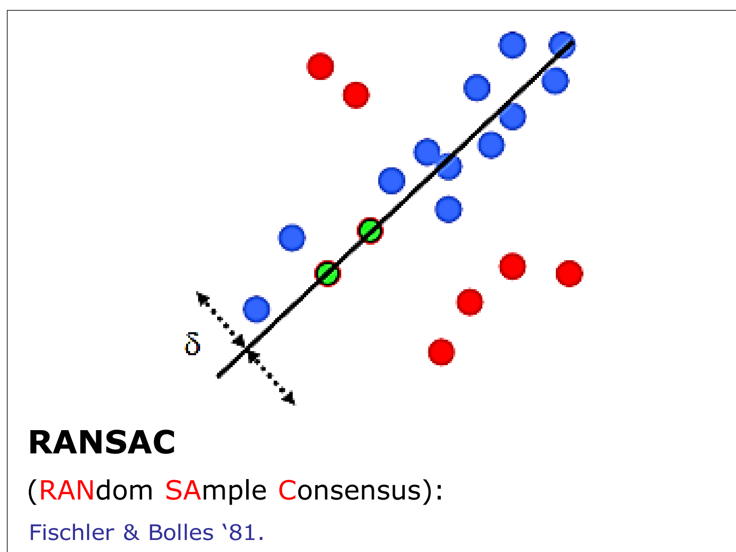
Σχήμα 3 – Fitting RANSAC

Ένα σύνολο δεδομένων με πολλά ακραία σημεία για τα οποία πρέπει να τοποθετηθεί μια γραμμή.

Γραμμή **RANSAC**.
Οι ακραίες τιμές δεν επηρεάζουν το αποτέλεσμα.

Το Fitting είναι η προσπάθεια προσαρμογής των παρατηρούμενων δεδομένων σε ένα παραμετρικό μοντέλο που υποτίθεται ότι ισχύει. Η διαδικασία προσαρμογής ενός τέτοιου μοντέλου στα δεδομένα περιλαμβάνει την εκτίμηση των παραμέτρων που περιγράφουν το μοντέλο, έτσι ώστε να ελαχιστοποιείται το σφάλμα εκτίμησης (Σχήμα 3).

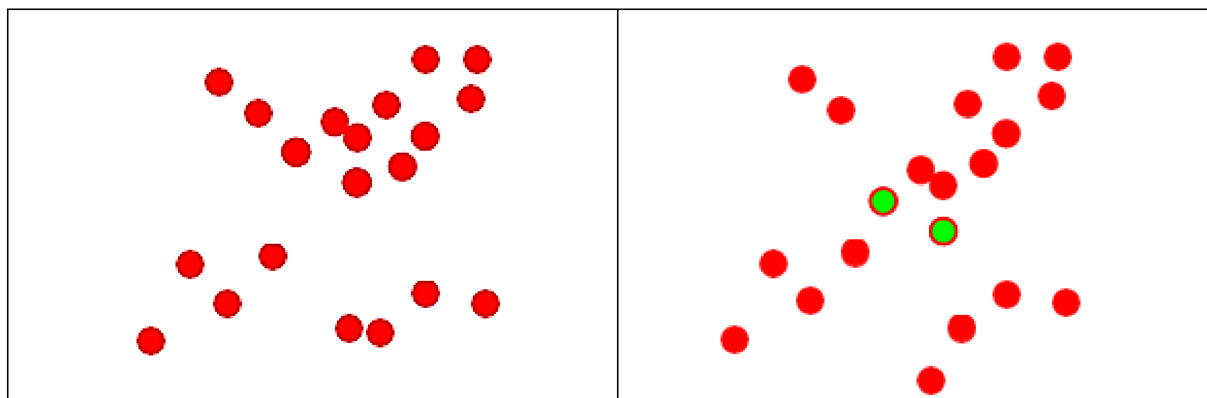
Ένα κλασικό παράδειγμα είναι η προσαρμογή γραμμής: δεδομένου ενός συνόλου σημείων στο 2D χώρο, ο στόχος είναι να βρεθούν οι παράμετροι που περιγράφουν τη γραμμή έτσι ώστε να ταιριάζει καλύτερα στο σύνολο των σημείων 2D. Παρόμοια προβλήματα μπορούν να προσδιοριστούν για άλλες γεωμετρικές ποσότητες όπως καμπύλες, ομογραφικοί μετασχηματισμοί, θεμελιώδεις πίνακες ή ακόμη και σχήματα αντικειμένων.



Σχήμα 4 – RANSAC

Ας υποθέσουμε ότι έχουμε το τυπικό πρόβλημα προσαρμογής γραμμής παρουσία ακραίων τιμών. Μπορούμε να διατυπώσουμε αυτό το πρόβλημα ως εξής:

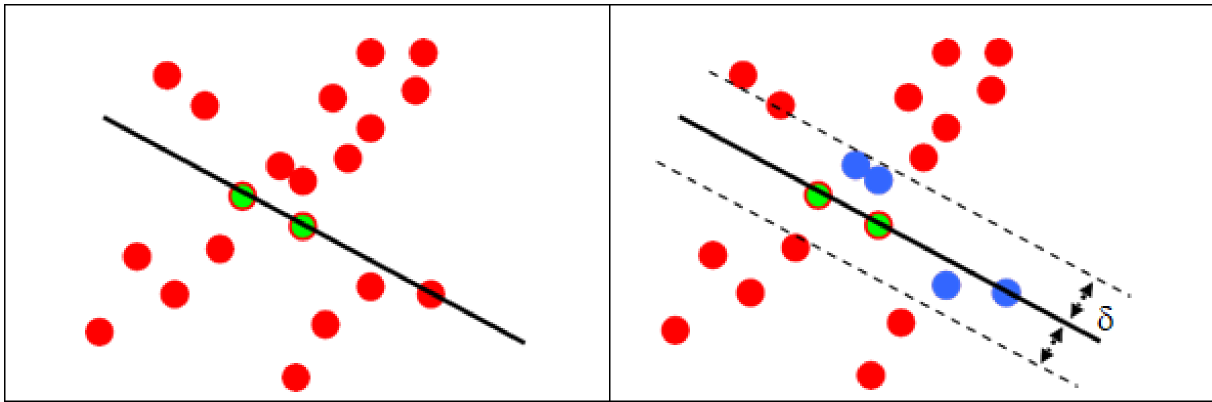
θέλουμε να βρούμε το καλύτερο υποσύνολο των σημείων στο inlier & outlier set, έτσι ώστε να διατηρούμε όσα περισσότερα inliers σημεία εντός μιας προκαθορισμένης τιμής κατωφλίου δ (Σχήμα 4).



Σχήμα 5 – Βήματα εκτέλεσης αλγορίθμου RANSAC (α)

Στην πράξη, το πρόβλημα μπορεί να διατυπωθεί ως εξής: Όπως και πριν, έχουμε ένα σύνολο σημείων 2D στα οποία θέλουμε να χωρέσουμε μια γραμμή. Υπάρχουν τρία σημαντικά βήματα για την υλοποίηση του RANSAC.

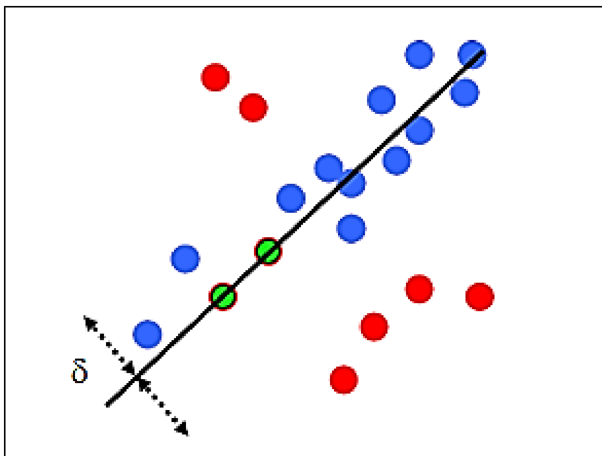
1. Αρχικά, επιλέγουμε ένα τυχαίο δείγμα του ελάχιστου απαιτούμενου αριθμού δειγμάτων για να μπορεί να προσδιοριστεί το μαθηματικό μοντέλο. Σε αυτήν την περίπτωση, μια γραμμή καθορίζεται από τουλάχιστον δύο σημεία. Σε αυτό το παράδειγμα, το τυχαίο δείγμα αποτελείται από τα δύο πράσινα σημεία (Σχήμα 5).



Σχήμα 6 – Βήματα εκτέλεσης αλγορίθμου RANSAC (β), (γ)

2. Στη συνέχεια, υπολογίζουμε τις παραμέτρους του μοντέλου από αυτό το σύνολο δειγμάτων. Χρησιμοποιώντας τα δύο πράσινα σημεία που έχουμε ήδη δειγματοληψία, υπολογίζουμε τη γραμμή που σχηματίζεται απο αυτά τα δύο.

3. Τέλος, καταγράφουμε τον αριθμό των σημείων 2D, που μπορούν να συμπεριληφθούν σε αυτό το μοντέλο, βάσει της προκαθορισμένης τιμής (threshold) - ανοχή δ. Σε αυτήν την περίπτωση, βλέπουμε ότι όλα τα σημεία που είναι τώρα μπλε συμφωνούν με το μοντέλο που υποτίθεται από τα δύο πράσινα σημεία. Έτσι, έχουμε ένα αρχικό σύνολο inliers 6 (2 πράσινα + 4 μπλε) και ένα εξωτερικό σύνολο outliers 14 (Σχήμα 6).



Σχήμα 7 – Μοντέλο με περισσότερους inliers

Επαναλαμβάνουμε τα βήματα 1-3 έως ότου δοκιμάσουμε όλους τους δυνατούς συνδυασμούς δειγμάτων στο σύνολο των σημείων. Το μοντέλο που συμπεριλαμβάνει στη λύση του τα περισσότερα δεδομένα (inliers) είναι το επικρατέστερο (Σχήμα 7).

Γενικά, ο αλγόριθμος RANSAC παρουσιάζεται στον Πίνακα 1:

Πίνακας 1 – Ψευδοκώδικας RANSAC

```
Given:
    data – A set of observations.
    model – A model to explain observed data points.
    n – Minimum number of data points required to estimate model parameters.
    k – Maximum number of iterations allowed in the algorithm.
    t – Threshold value to determine data points that are fit well by model.
    d – Number of close data points required to assert that a model fits
well to data.

Return:
    bestFit – model parameters which best fit the data (or null if no good
model is found)

iterations = 0
bestFit = null
bestErr = something really large

while iterations < k do
    maybeInliers := n randomly selected values from data
    maybeModel := model parameters fitted to maybeInliers
    alsoInliers := empty set
    for every point in data not in maybeInliers do
        if point fits maybeModel with an error smaller than t
             add point to alsoInliers
    end for
    if the number of elements in alsoInliers is > d then
        // This implies that we may have found a good model
        // now test how good it is.
        betterModel := model parameters fitted to all points in maybeInliers
and alsoInliers
        thisErr := a measure of how well betterModel fits these points
        if thisErr < bestErr then
            bestFit := betterModel
            bestErr := thisErr
        end if
    end if
    increment iterations
end while

return bestFit
```

Παράμετροι

Η τιμή κατωφλίου – threshold για να προσδιοριστεί πότε ένα σημείο δειγμάτων ταιριάζει σε ένα μοντέλο t και ο αριθμός των κοντινών σημείων που απαιτούνται για το ταίριασμα ενός μοντέλου με τα δεδομένα d, καθορίζονται με βάση συγκεκριμένες απαιτήσεις της εφαρμογής και πιθανώς βασίζονται σε πειραματική εκτίμηση. Ο αριθμός των επαναλήψεων K, ωστόσο, μπορεί να προσδιοριστεί ως συνάρτηση της επιθυμητής πιθανότητας επιτυχίας P. Έστω P είναι η επιθυμητή πιθανότητα που ο αλγόριθμος RANSAC παρέχει ένα επιτυχές αποτέλεσμα. Ο RANSAC επιστρέφει ένα επιτυχημένο αποτέλεσμα εάν σε κάποια επανάληψη επιλέγει μόνο inliers από το σύνολο δειγμάτων.

Έστω w είναι η πιθανότητα επιλογής ενός inlier, δηλαδή,

w = αριθμός inliers στα δεδομένα / αριθμός σημείων στα δεδομένα

Μια συνηθισμένη περίπτωση είναι ότι το w δεν είναι γνωστό εκ των προτέρων, αλλά μπορεί να δοθεί κάποια τυχαία τιμή. Υποθέτοντας ότι έχουμε n σημεία που απαιτούνται για την εκτίμηση ενός μοντέλου, w^n είναι η πιθανότητα επιλογής μόνο inliers και $1-w^n$ είναι η πιθανότητα ότι τουλάχιστον ένα από τα σημεία είναι ακραία, μια περίπτωση που υπονοεί ότι ένα κακό μοντέλο θα εκτιμηθεί από αυτό το σύνολο σημείων. Αυτή η πιθανότητα ότι ο αλγόριθμος δεν επιλέγει ποτέ ένα σύνολο σημείων τα οποία είναι όλα inliers είναι ίση με το 1-P. Συνεπώς,

$$1 - P = (1 - w\hat{\ }n)\hat{\ }K \qquad (2.8)$$

στο οποίο, μετά τη λήψη του λογάριθμου και των δύο πλευρών, οδηγεί:

$$K = \frac{\log(1-P)}{\log(1-w\hat{\ }n)} \qquad (2.9)$$

Με βάση αυτή τη σχέση, γνωρίζοντας το κλάσμα των inliers w, μετά από k επαναλήψεις του αλγορίθμου RANSAC, θα έχουμε μια πιθανότητα P να βρούμε ένα σύνολο σημείων χωρίς ακραίες τιμές. Για παράδειγμα, εάν θέλουμε πιθανότητα επιτυχίας ίση με P = 99% και γνωρίζουμε ότι το ποσοστό των inliers στο σύνολο δεδομένων είναι w = 50%, τότε λαμβάνουμε k = 17 επαναλήψεις (Πίνακας 2), οι οποίες είναι πολύ λιγότερες από τον αριθμό των πιθανών συνδυασμών, που έπρεπε να ελεγχθούν προηγουμένως.

Πίνακας 2 – Αριθμός K επαναλήψεων

	*Ποσοστό **outliers** 1-w*						
n	5%	10%	20%	25%	30%	40%	50%
2	2	3	5	6	7	11	17
3	3	4	7	9	11	19	35
4	3	5	9	13	17	34	72
5	4	6	12	17	26	57	146
6	4	7	16	24	37	97	293
7	4	8	20	33	54	163	588
8	5	9	26	44	78	272	1177

που απαιτείται για να διασφαλιστεί ότι, με πιθανότητα 99%, τουλάχιστον ένα δείγμα δεν εμπεριέχει outliers.
(David Lowe)

Πλεονεκτήματα

Ένα πλεονέκτημα του RANSAC είναι η ικανότητά του να κάνει ισχυρή εκτίμηση των παραμέτρων του μοντέλου, δηλαδή, μπορεί να εκτιμήσει τις παραμέτρους με υψηλό βαθμό ακρίβειας, ακόμη και όταν υπάρχει σημαντικός αριθμός ακραίων τιμών στο σύνολο δεδομένων.

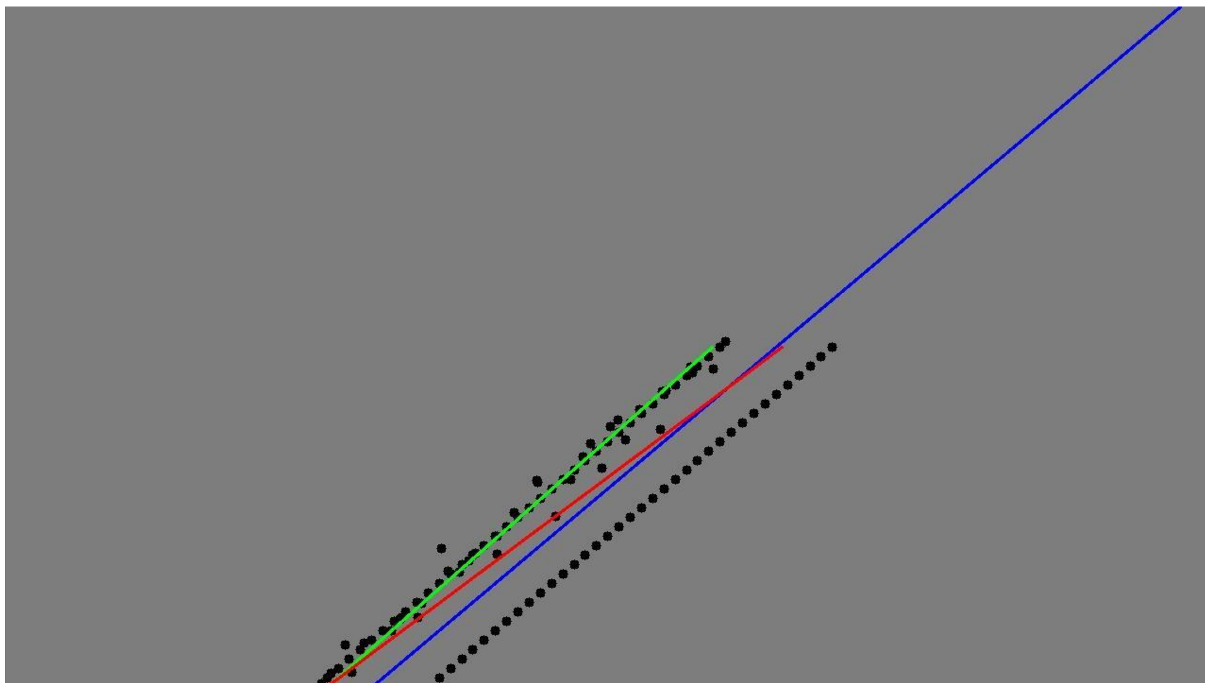
Μειονεκτήματα

Δεν υπάρχει περιορισμός στο χρόνο που απαιτείται για τον υπολογισμό αυτών των παραμέτρων. Όταν ο αριθμός των επαναλήψεων που υπολογίζεται είναι περιορισμένος, η ληφθείσα λύση μπορεί να μην είναι η βέλτιστη, και μπορεί ακόμη και να μην είναι κατάλληλη για τα δεδομένα. Με τον υπολογισμό ενός μεγαλύτερου αριθμού επαναλήψεων αυξάνεται η πιθανότητα να δημιουργηθεί ένα λογικό μοντέλο.

Δεν είναι πάντοτε ικανός να εντοπίσει το βέλτιστο μοντέλο και συνήθως αποτυγχάνει όταν ο αριθμός των inliers είναι μικρότερος από 50%.

Ένα άλλο μειονέκτημα του RANSAC είναι ότι απαιτεί τον καθορισμό συγκεκριμένων thresholds και την εύρεση των βέλτιστων παραμέτρων. Επίσης απαιτεί μεγάλο αριθμό επαναλήψεων με αποτέλεσμα να καθίσταται χρονοβόρος.

Όταν υπάρχουν περισσότερες από μία λύσεις (Σχήμα 8), η εκτέλεση του RANSAC μπορεί να μην βρεί καμία. Μπορεί να εξαλείψει τα outliers, κάτι που είναι λογικό, αλλά δεν είναι και τέλειο. Όταν ταιριάζει σε δύο σημεία που είναι περίπου παράλληλα με μια ευθεία γραμμή, τα αποτελέσματα τοποθέτησης δεν είναι τα καλύτερα, έως και λάθος, όπως φαίνεται στο παρακάτω σχήμα:



Σχήμα 8 – Εκτίμηση αποτελέσματος από την εκτέλεση του αλγορίθμου RANSAC

Πράσινη γραμμή: αποτέλεσμα τοποθέτησης RANSAC.

Κόκκινη γραμμή: αποτέλεσμα ελάχιστων τετραγώνων.

Μπλε γραμμή: επιθυμητό ιδανικό αποτέλεσμα.

Όταν θέλουμε να χωρέσουμε μια ευθεία γραμμή από τα σημεία δεδομένων όπως φαίνεται στο σχήμα, η ιδανική κατάσταση είναι η μπλε γραμμή. Ωστόσο, η πραγματική κατάσταση θα είναι όπως φαίνεται από την πράσινη γραμμή, επειδή το σημείο που δείχνει η μπλε γραμμή δεν διέρχεται από δύο από τα σημεία δεδομένων.

Ο μετασχηματισμός Hough [13] είναι μια εναλλακτική ισχυρή τεχνική εκτίμησης που μπορεί να είναι χρήσιμη όταν υπάρχουν περισσότερα από ένα μοντέλα. Μια άλλη προσέγγιση για την τοποθέτηση πολλαπλών μοντέλων είναι γνωστή ως PEARL [14], που συνδυάζει τη δειγματοληψία μοντέλου από σημεία δεδομένων όπως στον RANSAC με επαναληπτική επανεκτίμηση των inliers.

2.2.3 Ανάπτυξη και βελτιώσεις

Από το 1981, ο αλγόριθμος RANSAC έχει βρει μεγάλη εφαρμογή σε εφαρμογές της μηχανικής όρασης και της επεξεργασίας εικόνας.

Ο RANSAC μπορεί να είναι ευαίσθητος στην επιλογή του σωστού κατωφλίου θορύβου που καθορίζει ποια σημεία δεδομένων ταιριάζουν σε ένα μοντέλο με ένα συγκεκριμένο σύνολο παραμέτρων. Εάν ένα τέτοιο όριο είναι πολύ μεγάλο, τότε όλες οι υποθέσεις τείνουν να κατατάσσονται εξίσου (καλές). Από την άλλη πλευρά, όταν το όριο θορύβου είναι πολύ μικρό, οι εκτιμώμενες παράμετροι τείνουν να είναι ασταθείς (δηλ. απλώς προσθέτοντας ή αφαιρώντας ένα δείγμα στο σύνολο των inliers, η εκτίμηση των παραμέτρων μπορεί παρουσιάζει σημαντικές διαφοροποιήσεις). Για να αντισταθμιστεί εν μέρει αυτό το ανεπιθύμητο αποτέλεσμα, ο Torr et al. πρότεινε δύο τροποποιήσεις του RANSAC που ονομάζονται MSAC (M-εκτιμητής SAmple και Consensus) και MLESAC [15] (Μέγιστη πιθανότητα εκτιμήσεων SAmple και συναίνεση). Η κύρια ιδέα είναι να αξιολογηθεί η ποιότητα του συνόλου συναίνεσης (consensus) (δηλαδή τα δεδομένα που ταιριάζουν σε ένα μοντέλο υπολογίζοντας την πιθανότητά του). Μια επέκταση στο MLESAC που λαμβάνει υπόψη τις προηγούμενες πιθανότητες που σχετίζονται με το σύνολο δεδομένων εισόδου προτείνεται από τον Tordoff [16]. Ο αλγόριθμος που προκύπτει ονομάζεται Guided-MLESAC. Ο Chum [17] πρότεινε να καθοδηγήσει τη διαδικασία δειγματοληψίας εάν είναι γνωστές κάποιες εκ των προτέρων πληροφορίες σχετικά με τα δεδομένα εισόδου, δηλαδή εάν ένα δείγμα είναι πιθανό να είναι inlier ή outlier. Η προτεινόμενη μέθοδος ονομάζεται PROSAC, PROgressive SAmple Consensus.

Επίσης πρότεινε μια παραλλαγή του RANSAC που ονομάζεται R-RANSAC [18] για τη μείωση του υπολογιστικού φόρτου για τον προσδιορισμό ενός καλού συνόλου συναίνεσης. Η βασική ιδέα είναι να αξιολογηθεί αρχικά το ταίριασμα του μοντέλου χρησιμοποιώντας μόνο ένα υποσύνολο σημείων αντί για ολόκληρο το σύνολο δεδομένων. Μια καλή στρατηγική θα εκτιμήσει πότε είναι απαραίτητο να αξιολογηθεί η εφαρμογή ολόκληρου του συνόλου δεδομένων ή όταν το μοντέλο μπορεί να απορριφθεί εύκολα. Ο τύπος στρατηγικής που προτείνεται από τον Chum et al. ονομάζεται πρόγραμμα προτίμησης. Ο Nistér πρότεινε ένα παράδειγμα που ονομάζεται Preemptive RANSAC [19] που επιτρέπει σε πραγματικό χρόνο ισχυρή εκτίμηση της δομής μιας σκηνής και της κίνησης της κάμερας. Η βασική ιδέα της προσέγγισης συνίσταται στη δημιουργία ενός καθορισμένου αριθμού υποθέσεων, έτσι ώστε η σύγκριση να γίνεται σε σχέση με την ποιότητα της παραγόμενης υπόθεσης.

Άλλοι ερευνητές προσπάθησαν να αντιμετωπίσουν δύσκολες καταστάσεις όπου η κλίμακα θορύβου δεν είναι γνωστή ή / και υπάρχουν πολλές παρουσίες μοντέλων. Το πρώτο πρόβλημα αντιμετωπίστηκε στο έργο των Wang και Suter [20]. Ο Toldo et al. αντιπροσωπεύει κάθε δεδομένο με τη χαρακτηριστική συνάρτηση του συνόλου τυχαίων μοντέλων που ταιριάζουν στο σημείο. Στη συνέχεια, πολλά μοντέλα αποκαλύπτονται ως συστάδες που ομαδοποιούν τα σημεία που υποστηρίζουν το ίδιο μοντέλο. Ο αλγόριθμος ομαδοποίησης, που ονομάζεται J-linkage, δεν απαιτεί προηγούμενη προδιαγραφή του αριθμού των μοντέλων, ούτε απαιτεί χειροκίνητο συντονισμό παραμέτρων [21].

Ο RANSAC έχει επίσης προσαρμοστεί για αναδρομικές εφαρμογές εκτίμησης κατάστασης, όπου οι μετρήσεις εισόδου είναι εκφυλισμένες από ακραίες τιμές (KALMANSAC) [22].

2.3 Αλγόριθμος RANSAC & Matlab

Πίνακας 3 – Κώδικας Matlab για υλοποίηση RANSAC

```
close all;
thr_dis = ;

x = [];
y = [];

num_p = length(x);
s = num_p*num_p;
a = zeros(s,1);
b = zeros(s,1);
c = zeros(s,1);
d = zeros(s,1);
m = 1;
num_lines = 0;

% παράμετροι a, b, c κάθε γραμμής
for i = 1:num_p
    for j = (i+1):num_p
            a(m,1) = y(i)-y(j);
            b(m,1) = x(j)-x(i);
            c(m,1) = y(i)*(x(i)-x(j))+x(i)*(y(j)-y(i));
       m = m + 1;
       num_lines = num_lines + 1;
    end
end

% υπολογισμός αποστάσεων
for i = 1:(num_lines)
    for j = 1:(num_p)
            d(i,j) = abs((a(i,1)*x(j)+b(i,1)*y(j)+c(i,1))
                     /(sqrt((power(a(i,1),2))+(power(b(i,1),2)))));
    end
end

% αριθμός inliers and άθροισμα αποστάσεων
inliers_num = zeros(num_lines,1);
sum_dis = zeros(num_lines,1);
for i = 1:(num_lines)
    for j = 1 : num_p

% προσαύξηση inlier_num και sum dis
            If (d(i,j) <= thr_dis)
                inliers_num(i,1) = inliers_num(i,1) + 1;
                sum_dis(i,1) = sum_dis(i,1) + d(i,j);
            end
    end
end
```

```
% εύρεση καλύτερης γραμμής
max = 0;
max_i = 0;
max_sum_dis = 0;
for i = 1:(num_lines)
    if (inliers_num(i,1) > max)
        max = inliers_num(i,1);
        max_i = i;
        max_sum_dis = sum_dis(i,1);
    elseif ((inliers_num(i,1) == max) && (sum_dis(i,1) < max_sum_dis))
        max = inliers_num(i,1);
        max_i = i;
        max_sum_dis = sum_dis(i,1);
    end
end

% max line info ---> terminal
max
max_i
max_sum_dis
a_max = a(max_i,1)
b_max = b(max_i,1)
c_max = c(max_i,1)

% ploting
q = 0:1:100;
w = ((-1*a(max_i,1))/(b(max_i,1)))*q
-(c(max_i,1)/(b(max_i,1)));

% legend
plot (q,w)
hold on
plot (x,y)
legend ('w','y')
```

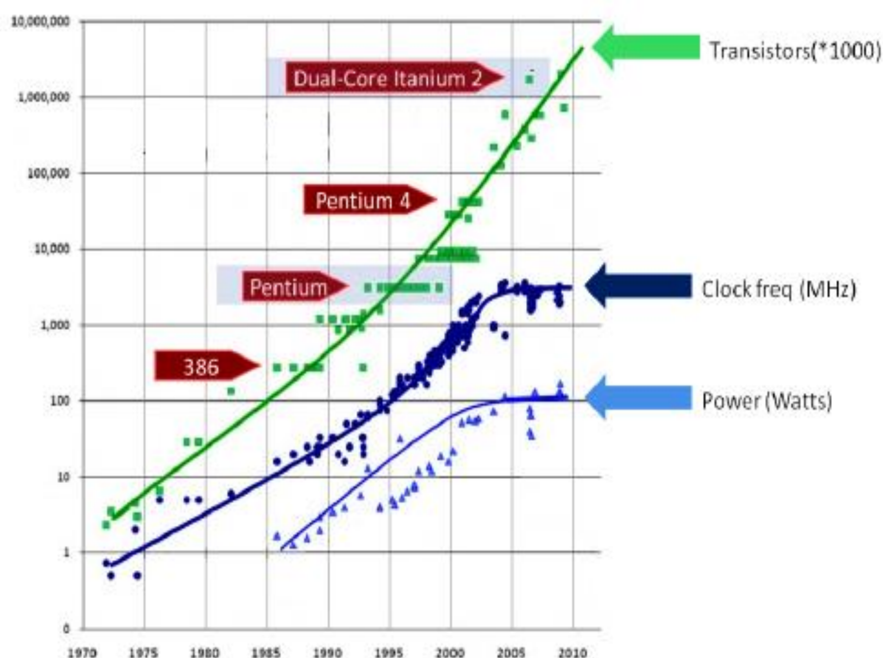
3 Μεθοδολογία και σχεδιασμός ενσωματωμένων συστημάτων που βασίζονται σε FPGA

3.1 Εισαγωγή

Τις τελευταίες δεκαετίες, η εξέλιξη στους ενσωματωμένους υπολογιστές έχει αυξηθεί χάρη στις πολλαπλές προόδους στις τεχνολογίες VLSI. Πριν από το 1970, οι επεξεργαστές αποτελούσαν πολλά ολοκληρωμένα κυκλώματα. Το 1971 ο πρώτος επεξεργαστής σε ένα ενιαίο ολοκληρωμένο κύκλωμα αναπτύχθηκε από την Intel. Αυτός ο επεξεργαστής που ονομάζεται Intel 4004 ήταν επεξεργαστής 4-bit με συχνότητα λειτουργίας περίπου 100 KHz και αποτελούνταν από 2300 τρανζίστορ χρησιμοποιώντας τεχνολογία κατασκευής 10 Micron. Σήμερα έχουμε επεξεργαστές 64-bit που λειτουργούν σε συχνότητες 30000 φορές μεγαλύτερες και αποτελούνται από περισσότερα από 1 δισεκατομμύριο τρανζίστορ με τεχνολογία κατασκευής 14nm (Core i9).

Με την τρέχουσα τεχνολογία φτάνουμε στο όριο λειτουργίας συχνότητας λόγω περιορισμών στη διαδικασία κατασκευής τρανζίστορ. Στη νανομετρική κλίμακα που χρησιμοποιείται σήμερα, η πύλη του τρανζίστορ είναι πολύ λεπτή, γεγονός που δημιουργεί αύξηση στο ρεύμα διαρροής, αυξάνοντας δραματικά τη στατική ισχύ. Στην περίπτωση της κατασκευής FPGA, χρησιμοποιώντας εξειδικευμένες διαδικασίες κατασκευής [23] [24], οι μηχανικοί μπόρεσαν να δημιουργήσουν υψηλότερη απόδοση και τρανζίστορ χαμηλότερης ισχύος.

Σύμφωνα με το νόμο του Moore, ο αριθμός των τρανζίστορ αυξάνεται, αλλά οι ταχύτητες του ρολογιού επιβαρύνονται. Απεικονίζει την απόδοση ανά watt και ανά περιοχή τσιπ σε συνάρτηση με παράλληλες τεχνολογίες επεξεργασίας (βλέπε Σχήμα 9).



Σχήμα 9 – Νόμος Moore
από https://xlabs.ai/moores-law-resurrected-by-marketing/

Η υπολογιστική πληροφορική έχει εξελιχθεί από την αποκλειστική επεξεργασία που πραγματοποιείται από έναν επεξεργαστή, σε μια πραγματική παράλληλη επεξεργασία. Όταν γίνεται χρήση ενός μόνο επεξεργαστή, κάποια έννοια της παράλληλης επεξεργασίας μπορεί να εξομοιωθεί με ένα Λειτουργικό Σύστημα (OS), το οποίο χρησιμοποιεί χρονική πολυπλεξία πόρων μεταξύ των διαφορετικών νημάτων (threads). Χρησιμοποιώντας DSP, ASIC, FPGA, GPU ή σύστημα πολλαπλών πυρήνων, οι πραγματικές παράλληλες λειτουργίες μπορούν να εκτελεστούν ταυτόχρονα χάρη στις ιδιαίτερες αρχιτεκτονικές τους.

3.2 Παράλληλες πλατφόρμες επεξεργασίας

Για περιπτώσεις παράλληλης επεξεργασίας σήματος με μεγάλο όγκο δεδομένων, υπάρχουν διαφορετικές διαθέσιμες αρχιτεκτονικές. Οι πιο συχνά χρησιμοποιούμενα σήμερα είναι αυτές που βασίζονται σε: DSP, GPU, ASIC και FPGA. Κάθε μία από τις προαναφερθείσες αρχιτεκτονικές παρουσιάζει πλεονεκτήματα και μειονεκτήματα και χρησιμοποιείται για διαφορετικούς σκοπούς.

3.2.1 Επεξεργαστής ψηφιακού σήματος (DSP)

Το DSP είναι ένα σύστημα επεξεργαστή βελτιστοποιημένο για την εφαρμογή επεξεργασίας σήματος σε πολύ υψηλή ταχύτητα. Τα DSP περιλαμβάνουν μια εξειδικευμένη αρχιτεκτονική που επιτρέπει παράλληλη επεξεργασία στο επίπεδο εντολών, SIMD (Single Intruction Multiple Data). Στην αγορά υπάρχουν floating και fixed point DSP˙s.

3.2.2 Μονάδα επεξεργασίας γραφικών (GPU)

Αρχικά οι GPU σχεδιάστηκαν ως επιταχυντές γραφικών, έως ότου οι μηχανικοί άρχισαν να τις χρησιμοποιούν για παράλληλη επεξεργασία. Η GPU αποτελείται από εκατοντάδες μικρούς πυρήνες που μπορούν να χρησιμοποιηθούν για εφαρμογές γραφικών ή υπολογιστές υψηλής απόδοσης. Οι GPU χρησιμοποιούνται από πλατφόρμες προγραμματισμού όπως το CUDA και το OpenCL, βασισμένες σε γλώσσες υψηλού επιπέδου όπως C, C++ ή Fortran.

Κατά την εργασία με GPU, το threading αντιμετωπίζεται αυτόματα από τον διαχειριστή νήματος υλικού (thread manager). Ο προγραμματιστής δεν έχει άμεσο έλεγχο των επεξεργαστών της GPU. Όλα γίνονται μέσω διεπαφών προγραμματισμού εφαρμογών (API).

Στην τρέχουσα αγορά βρίσκουμε τρεις κύριους κατασκευαστές GPU: NVidia, Intel και AMD. Μερικές από τις δημοφιλείς GPU από τη Nvidia είναι οι Geforce, ION, Quadro και Tesla. Η αρχιτεκτονική Tesla είναι μια πολύ ισχυρή GPU με δυνατότητα εκτέλεσης έως 4 teraflops σε απλή ακρίβεια και 80 gigaflops σε διπλή ακρίβεια. Οι GPU Quadro προορίζονται για επαγγελματική χρήση. Η GeForce προορίζεται για την παραδοσιακή αγορά· η αρχιτεκτονική Nvidia GeForce8 διαθέτει 128 thread επεξεργαστές (processors), καθένας ικανός να διαχειρίζεται έως και 96 ταυτόχρονα νήματα, για μέγιστο 12.288 νήματα.

Ένα από τα κύρια μειονεκτήματα των GPU είναι ο όγκος τους, με αποτέλεσμα να είναι περίπλοκη η εγκατάστασή τους σε μικρούς χώρους. Επίσης ειδική μνεία πρέπει να ληφθεί σε ότι αφορά την ψύξη τους και κατανάλωση ενέργειας. Οι GPU προορίζονται για χρήση ως συν-επεξεργαστές, δλδ. λειτουργούν πάντα παράλληλα με μια CPU, πράγμα που περιορίζει τη χρήση τους σε ενσωματωμένα συστήματα.

Στην [25] παρουσιάζεται μια σύγκριση χρόνου μεταξύ πολλαπλών αλγορίθμων επεξεργασίας εικόνας, που εφαρμόζονται σε CPU / CPU με GPU, που στη 2η περίπτωση (GPU ως συν-επεξεργαστής) επιταχύνθηκαν σχεδόν 100 φορές. Στην [26], παρουσιάζεται μια σύγκριση μεταξύ των επιδόσεων FPGA, GPU και CPU για διαφορετικούς αλγόριθμους επεξεργασίας εικόνας (stereo vision και δισδιάστατα φίλτρα). Στην [27], η τομογραφία επιταχύνεται 80 φορές σε σύγκριση της απλής υλοποίησης CPU χρησιμοποιώντας CUDA.

3.2.3 Ολοκληρωμένο κύκλωμα ειδικής εφαρμογής (ASIC)

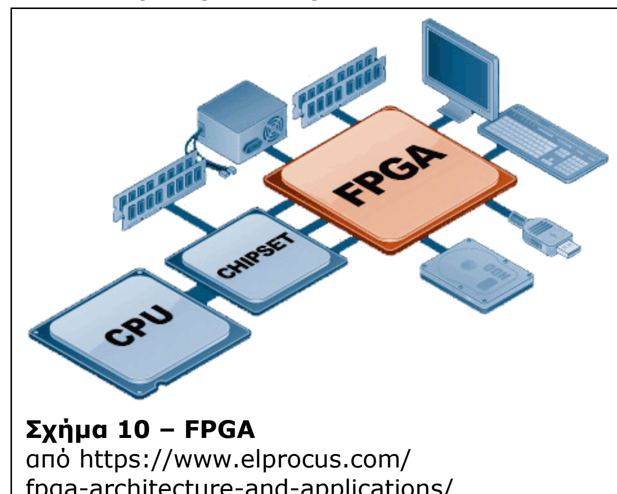
Το ASIC είναι ένα ολοκληρωμένο κύκλωμα που προορίζεται για την εκτέλεση συγκεκριμένων εργασιών και στο οποίο η αρχιτεκτονική έχει σχεδιαστεί χρησιμοποιώντας μια γλώσσα περιγραφής υλικού (HDL). Το σχέδιο είναι χαραγμένο στο πυρίτιο. Ο κύριος περιορισμός του ASIC είναι η τιμή και ο χρόνος διάθεσης στην αγορά, ο οποίος περιορίζει τη χρήση σε εφαρμογές που απαιτούν πολύ υψηλή απόδοση ή παραγωγή μεγάλου όγκου. Χρησιμοποιείται ευρέως σε εφαρμογές που χρησιμοποιούν σήματα RF όπως ραντάρ, τηλεόραση, κ.λπ.

Ο σχεδιασμός που χρησιμοποιεί ASIC προσφέρει καλύτερη απόδοση, πυκνότητα και κατανάλωση ισχύος σε σύγκριση με ένα FPGA. Το πρωτότυπο ASIC μπορεί να γίνει χρησιμοποιώντας FPGAs, κάτι που επιτρέπει την επανάληψη του προγραμματισμού FPGAs. Στην [28] περιγράφεται μια μεθοδολογία σχεδιασμού για τη διευκόλυνση του σχεδιασμού ASIC με βάση FPGAs.

3.2.4 Field-Programmable Gate Arrays (FPGA)

Τα FPGAs (Field-Programmable Gate Arrays), αποτελούν ολοκληρωμένα κυκλώματα που μπορούν να εκτελούν πάσης φύσεως λογικές συναρτήσεις. (Σχήμα 10).

Χάρη στη δυνατότητά τους να επαναπρογραμματίζονται όσες φορές χρειάζεται, προσφέρουν απαράμιλλη ευελιξία και υψηλή αποδοτικότητα.



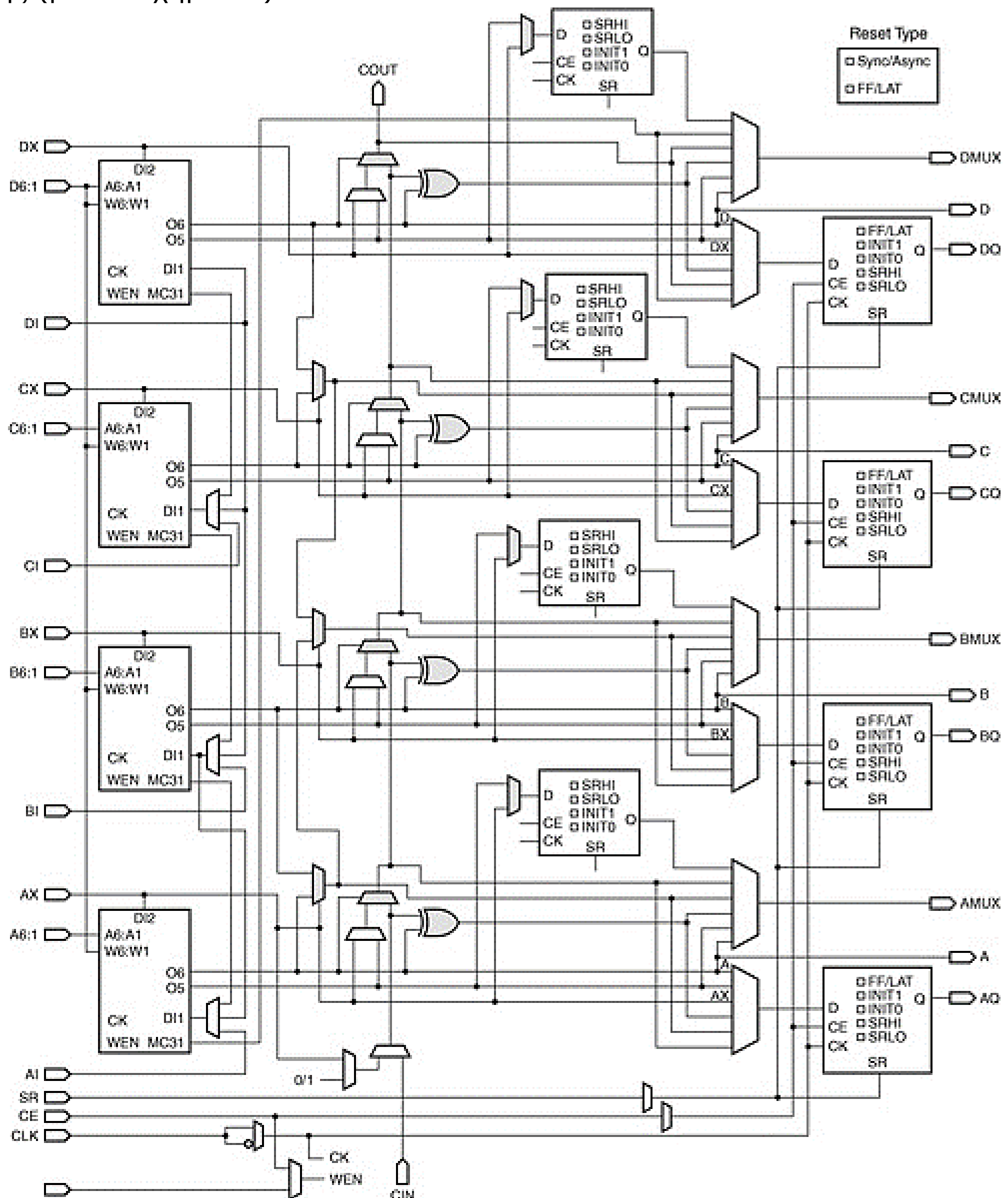
Σχήμα 10 – FPGA
από https://www.elprocus.com/fpga-architecture-and-applications/

Η ικανότητα παραλληλισμού των διεργασιών, τα καθιστούν ιδανικά για επεξεργασία δεδομένων, καθώς ένα FPGA μπορεί να είναι τάξεις μεγέθους γρηγορότερο από μία CPU σε συγκεκριμένες εφαρμογές (Hardware Acceleration).

Αναφορικά με την χρήση των FPGAs, η ικανότητα των διαχείρισης απλών ηλεκτρικών σημάτων, τα καθιστούν ιδανικά για πολλές εφαρμογές, όπως ηλεκτρονικοί εγκέφαλοι δορυφόρων, αεροπλάνων, παλμογράφων, διακομιστών (servers), μαγνητικών και αξονικών τομογράφων, συστημάτων επεξεργασίας εικόνας, οθονών, και άλλα πολλά.

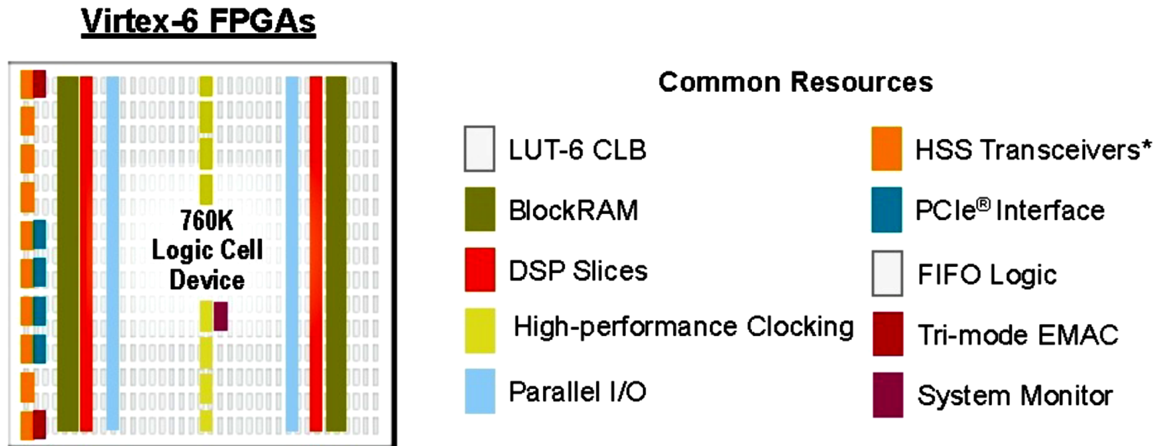
Ένα FPGA είναι μια συσκευή πυριτίου γενικής χρήσης που αποτελείται από προγραμματιζόμενες πύλες, πόρους διασύνδεσης και άλλες διαμορφώσιμες μονάδες που μπορούν να χρησιμοποιηθούν για να ταιριάζουν στις ανάγκες μιας συγκεκριμένης εφαρμογής.

Το κύριο στοιχείο του FPGA είναι το Logic Element (LE). Πολλά LE δρομολογούνται μαζί με προκατασκευασμένους πόρους διασύνδεσης και προγραμματιζόμενους διακόπτες. Το LE των Altera και Xilinx ονομάζεται Adaptive Logic Module (ALM) και Slice αντίστοιχα. Το LE περιέχει πίνακες LookUp, πολυπλέκτες, καταχωρητές και άλλους πόρους λογικής και σύνδεσης (βλέπε Σχήμα 11).



Σχήμα 11 – SLICE Virtex 6 μπλόκ διάγραμμα
από https://mithro-vtr.readthedocs.io/en/latest/tutorials/arch/xilinx_virtex_6_like.html

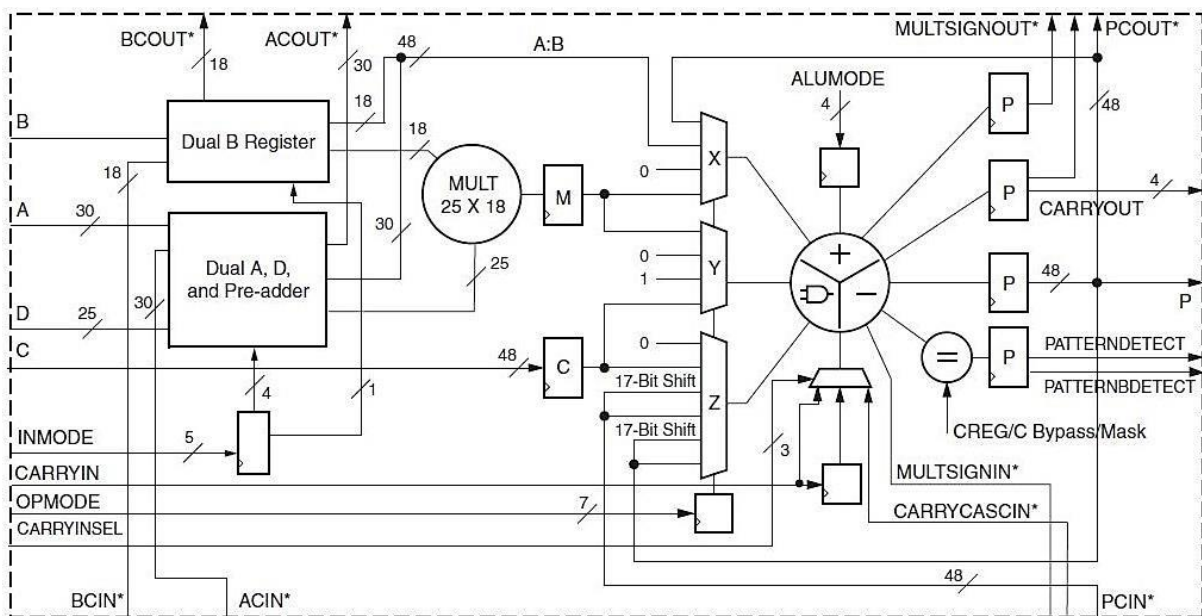
Τα Virtex 6 FPGA έχουν αρχιτεκτονική με βάση τη στήλη, με όλους τους διαφορετικούς πόρους σε στήλες (βλέπε Σχήμα 12).



Σχήμα 12 – Virtex 6 αρχιτεκτονική στήλης
από Xilinx documentation

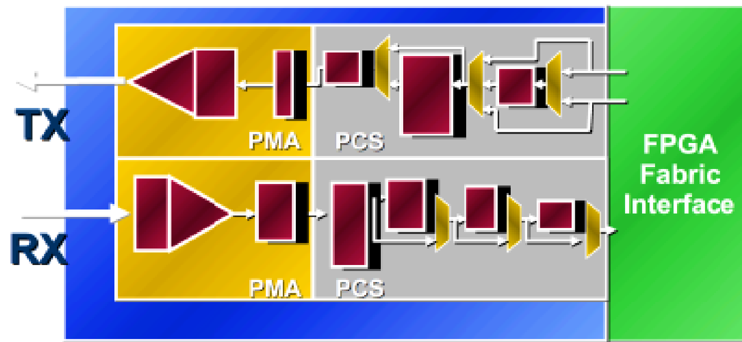
Στα FPGAs διατίθενται πόροι πυριτίου για χρήση από τον σχεδιαστή. Οι κύριοι πόροι που διατίθενται στα τρέχοντα FPGAs είναι hard Processors, μνήμη RAM, Slices, Slices DSP, Πολυπλέκτες, πομποδέκτες Gigabit, Triple-Speed Ethernet MAC, PCIexpress, Phase Locked Loop (PLL) κ.λ.π.

Στα FPGA υπάρχουν πολλαπλασιαστές πυριτίου, πράγμα το οποίο είναι χρήσιμο στην επεξεργασία σήματος. Στην περίπτωση της Xilinx, σε ορισμένα FPGA υπάρχουν εκατοντάδες μικρά μπλοκ DSP που ονομάζονται DSP48. Αυτή η μονάδα μπορεί να χρησιμοποιηθεί για την εφαρμογή πολλαπλασιασμού σταθερών σημείων, προσθέσεων, πολλαπλής συσσώρευσης (accumulation) (MACC), ανίχνευσης προτύπων κ.λ.π. Η μέγιστη απόδοση ενός DSP48 που υπάρχει στο Virtex 6 FPGA είναι 600 MHz, που λαμβάνεται όταν χρησιμοποιούνται όλα τα στάδια αγωγών (pipelines) εντός της μονάδας (βλέπε Σχήμα 13).



Σχήμα 13 – DSP48 μπλόκ διάγραμμα
από https://www.programmersought.com/article/5944117976/

Τα πρωτόκολλα υψηλής ταχύτητας επικοινωνίας (PCIe, Aurora, Ethernet κ.λ.π.) είναι απαραίτητα σε πολλές εφαρμογές. Για το λόγο αυτό, οι κατασκευαστές FPGA ενσωματώνουν gigabit transceivers FPGA (GTP) για μετάδοση και λήψη. (βλέπε Σχήμα 14).



Σχήμα 14 – GTP μπλόκ διαγράμματα (α)
από https://www.so-logic.net/documents/trainings/02_so_hssio_xilinx_mgt.pdf

Χάρη στην εξέλιξη του VLSI και τη μικροδομή των τρανζίστορ, η χωρητικότητα των FPGAs έχει αυξηθεί πάνω από έξι φορές. Σήμερα, βρίσκονται FPGA με πάνω από 1 εκατομμύριο LUTS, πάνω από 2 εκατομμύρια καταχωρητές, χιλιάδες εσωτερικά μπλοκ RAM και DSP, πολλούς πομποδέκτες πολλαπλών Gigabit και πολλούς άλλους πόρους. Για το λόγο αυτό, προκειμένου να επιτευχθούν οι σχεδιαστικοί στόχοι, οι μεθοδολογίες που χρησιμοποιούνται για την ανάπτυξη και τη χρήση FPGAs είναι σε συνεχή εξέλιξη.

Σύμφωνα με την [29], τα FPGAs έχουν απόδοση 100x υψηλότερη από DSP σε κάποιες εφαρμογές επεξεργασίας σήματος που είναι κατά υψηλό βαθμό παραλληλίσιμες.

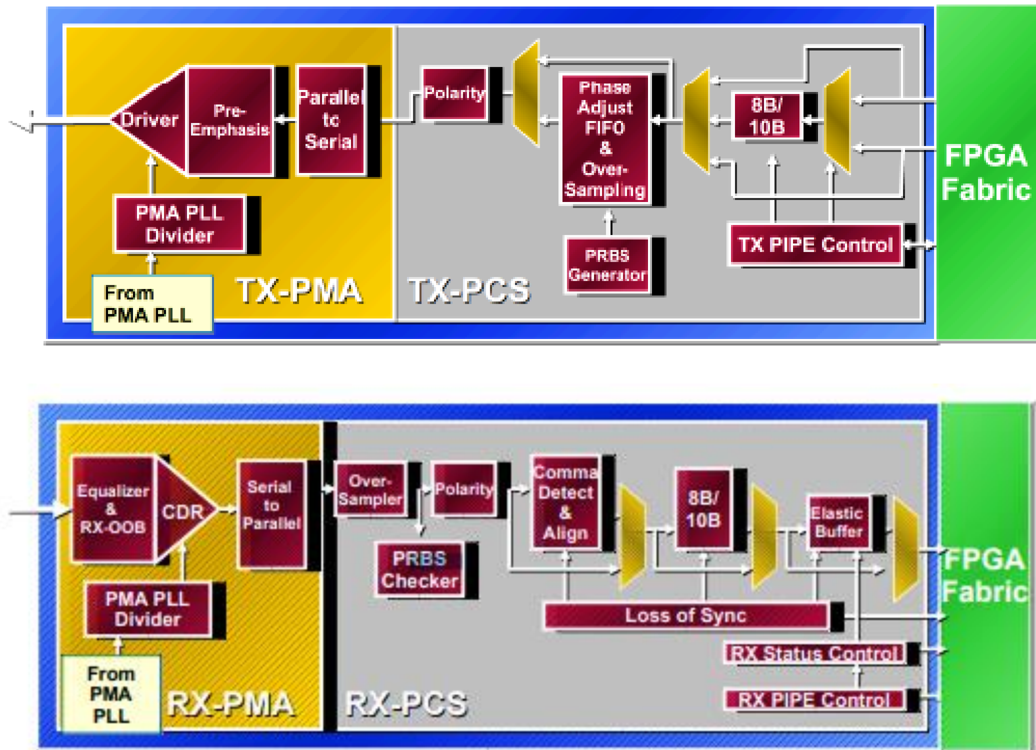
Ένα από τα κύρια πλεονεκτήματα ενός FPGA είναι η ευελιξία του να αναπτύσσει πολλαπλές εργασίες παράλληλα και σε πραγματικό χρόνο. Με έναν επεξεργαστή γενικού σκοπού (GPP), ένα πρόγραμμα εκτελείται με διαδοχικό τρόπο, ενώ σε ένα FPGA, πολλαπλές διεργασίες μπορούν να εκτελεστούν παράλληλα.

Συνήθως, ένα GPP έχει μια μοναδική αριθμητική λογική μονάδα (ALU), η οποία περιορίζει τον επεξεργαστή να εκτελεί μόνο μία αριθμητική λειτουργία κάθε φορά. Σε ένα GPP, η εκτέλεση εντολών βελτιστοποιείται για να τροφοδοτεί συνεχώς την ALU με τελεστές, αλλά γενικά μόνο ένα αποτέλεσμα μπορεί να ληφθεί σε κάθε παλμό ρολογιού.

Ένας GPP εξομοιώνει το multi-threading χρησιμοποιώντας πόρους επεξεργαστή με αποτελεσματικό τρόπο. Για αυτό, ενας scheduller ελέγχει την πολυπλεξία πόρων μεταξύ των διαφορετικών νημάτων, αλλά μόνο ένα νήμα εκτελείται κάθε φορά.

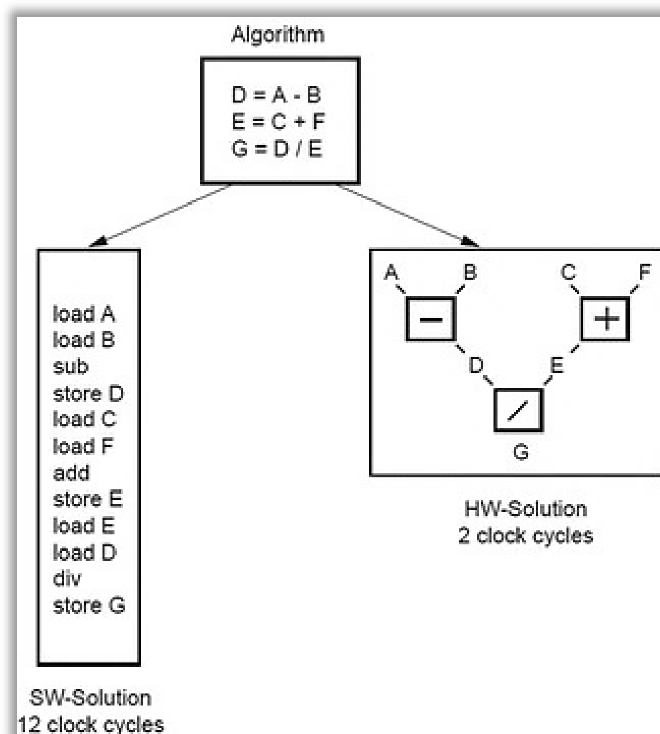
Ένα FPGA συνήθως λειτουργεί με χαμηλότερες ταχύτητες από τον GPP. Σήμερα, οι επεξεργαστές μπορούν να εκτελέσουν έως και 3 GHz, ενώ το πιο ισχυρό FPGA λειτουργεί σε λιγότερο από 1 GHz. Αυτό που κάνει τη διαφορά κατά την εργασία με FPGAs είναι ότι πολλές διαδικασίες μπορούν να εκτελεστούν σε κάθε παλμό ρολογιού. Ο αριθμός των διαδικασιών που μπορούν να υλοποιηθούν εξαρτάται από τον αλγόριθμο και τους διαθέσιμους πόρους.

Η μέγιστη ταχύτητα που επιτυγχάνεται για την εκτέλεση μιας λειτουργίας σε ένα FPGA εξαρτάται από τη φύση του και τον τρόπο με τον οποίο κωδικοποιήθηκε. Για παράδειγμα, σε ένα Virtex 6 FPGA πολλοί πολλαπλασιασμοί fixed point μπορούν να εκτελεστούν στα 400 MHz χρησιμοποιώντας τους ενσωματωμένους πολλαπλασιαστές που είναι διαθέσιμοι στα FPGA. Στην περίπτωση του V6lx240t υπάρχουν 768 DSP48 [30]. Αυτό σημαίνει, θεωρητικά, ότι θα μπορούσε να είναι ισοδύναμο με τον επεξεργαστή 460 GHz (768x600 MHz).



Σχήμα 14 – GTP μπλόκ διαγράμματα (β)
από https://www.so-logic.net/documents/trainings/02_so_hssio_xilinx_mgt.pdf

Χάρη στην παράλληλη αρχιτεκτονική που υπάρχει σε ένα FPGA, η επεξεργασία δεδομένων μπορεί να κλιμακωθεί δραστικά. Η εκτέλεση του ίδιου αλγορίθμου απλώς σε λογισμικό σε σύγκριση με την hardware υλοποίηση απαιτεί περισσότερους κύκλους ρολογιού, λόγω της διαδοχικής φύσης της προσέγγισης λογισμικού (βλέπε Σχήμα 15).



Σχήμα 15 – Σύγκριση μεταξύ hardware και software execution
από Hans-Peter Rosinger, Connecting Customized IP to the MicroBlaze Soft Processor Using the Fast Simplex Link (FSL) Channel.

Υπάρχουν 3 βασικές τεχνολογίες FPGA: Static RAM (SRAM) Flash και Anti-fuse. Σύμφωνα με την εκάστοτε εφαρμογή, είναι σημαντική η επιλογή της κατάλληλης τεχνολογίας FPGA.

Στην τεχνολογία SRAM, τα τρανζίστορ επιτρέπουν συνδέσεις σύμφωνα με τις πληροφορίες που είναι αποθηκευμένες σε μια εσωτερική μνήμη SRAM. Το κύριο πλεονέκτημα αυτής της τεχνολογίας είναι ότι τα εξαρτήματα μπορούν να αναδιαμορφωθούν, ενώ το μειονέκτημα είναι η επιφάνεια που απαιτείται.

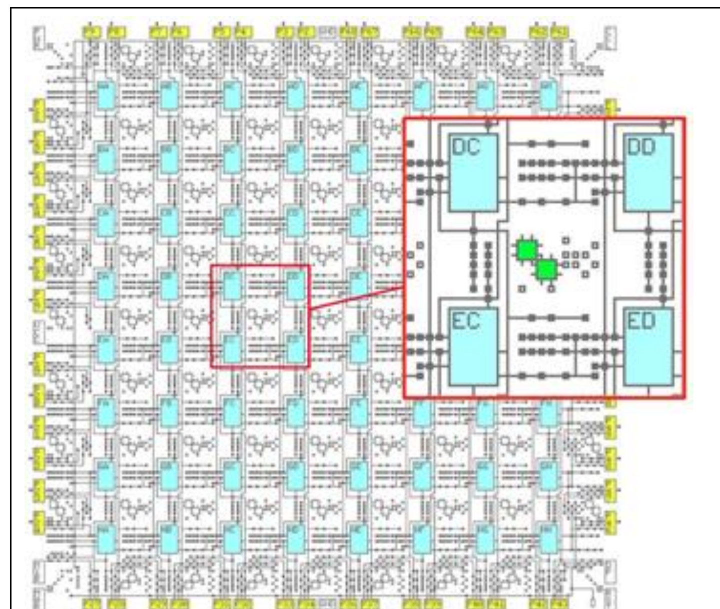
Τα FPGA της τεχνολογίας Flash είναι μη πτητικά, έχουν μικρότερη κατανάλωση ενέργειας κατά την εκκίνηση και προσφέρουν περισσότερη αντοχή έναντι της ακτινοβολίας σε σχέση με την τεχνολογία SRAM. Ωστόσο, δεν έχουν το ίδιο επίπεδο συρρίκνωσης με την τεχνολογία SRAM, η οποία μειώνει την πυκνότητα και αυξάνει το κόστος.

Στην τεχνολογία Anti-fuse, χρησιμοποιώντας ένα ρεύμα προγραμματισμού, πραγματοποιείται μόνιμα μια φυσική σύνδεση μεταξύ διαφορετικών σημείων. Αυτή η τεχνολογία είναι λιγότερο ακριβή και παρέχει καλύτερη απόδοση χρονισμού, ωστόσο, τα στοιχεία δεν μπορούν να επαναπρογραμματιστούν.

3.2.4.1 Η Αρχιτεκτονική των FPGAs

Η ιστορία του FPGA ξεκινάει το 1980 όταν, ο Ross Freeman, δημιούργησε το πρώτο FPGA XC2064, όπως φαίνεται στο Σχήμα 16, αποτελούμενο από ένα πλέγμα 8 * 8 διαμορφώσιμα μπλόκ λογικής (Configurable logic blocks (CLBs)).

Από αυτή την πρωτότυπη έκδοση, μπορεί κανείς να δει τη βασική δομή ενός FPGA. Σε αυτό το απλοποιημένο FPGA, υπάρχουν 64 CLBs. Κάθε CLB έχει τέσσερις εισόδους (A, B, C, D) και δύο εξόδους (X και Y). Ενδιάμεσα συνδυαστική λογική, η οποία μπορεί να προγραμματιστεί για την εφαρμογή οποιασδήποτε επιθυμητής λογικής λειτουργίας. Το CLB περιέχει επίσης ένα flip flop, επιτρέποντας στο FPGA να εφαρμόζει ακολουθιακή λογική, δηλαδή μετρητές, καταχωρητές αλλαγής, μηχανές κατάστασης και άλλα κυκλώματα κατάστασης.



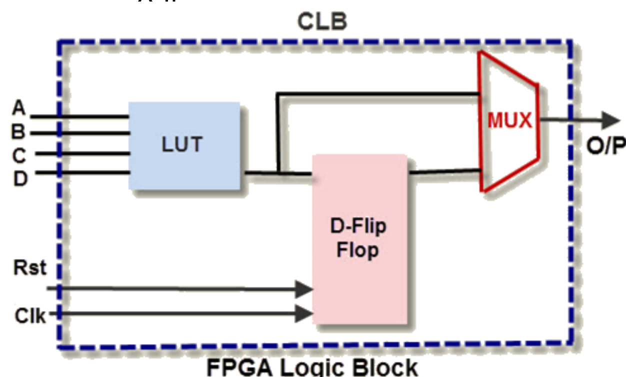
Σχήμα 16 – FPGA XC2064
από https://tech-en.netlify.app/articles/en520420/index.html

Υπάρχουν πολυπλέκτες, οι οποίοι μπορούν να προγραμματιστούν για να περάσουν από οποιαδήποτε από τις εισόδους τους για την εφαρμογή της συνδυαστικής λογικής. Οι πολυπλέκτες επιτρέπουν τη διαμόρφωση του CLB για μια συγκεκριμένη εργασία. Πολλαπλά CLBs μπορούν επίσης να προγραμματιστούν για την εφαρμογή μίας μόνο λειτουργίας. Η επικοινωνία μεταξύ CLBs μπορεί να γίνει μέσω αρχιτεκτονικής διασύνδεσης. Έτσι, εν συντομία, κάθε FPGA έχει τρία βασικά στοιχεία που μπορούν να αποτελέσουν τον πυρήνα της σύγχρονης αρχιτεκτονικής FPGA:

1. Διαμορφώσιμο λογικό μπλοκ CLB.
2. Αρχιτεκτονική διασύνδεσης.
3. Μπλοκ εισόδου / εξόδου.

3.2.4.1.1 LUTs, Flip-Flops, CLBs

Τα Διαμορφώσιμα μπλοκ λογικής (CLBs) περιέχουν τη λογική για το FPGA, καθώς και λοιπά κυκλώματα για τη δημιουργία μηχανών πεπερασμένων καταστάσεων (Finite State Machines), όπως φαίνεται στο Σχήμα 17.

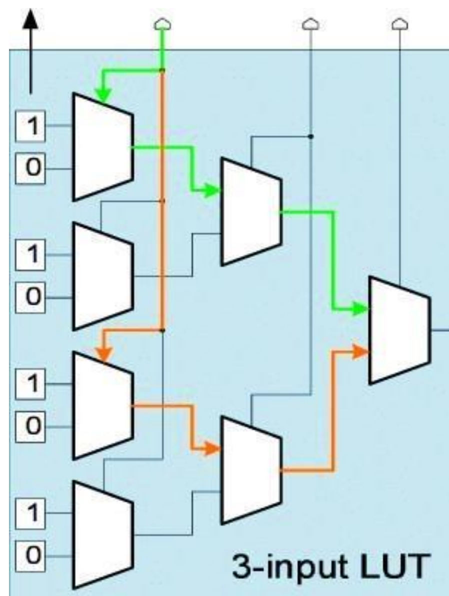


Σχήμα 17 – Διαμορφώσιμο μπλοκ λογικής (CLB) FPGA
από https://www.elprocus.com/fpga-architecture-and-applications/

Το μπλοκ περιέχει τρία βασικά στοιχεία: LUTs, Πολυπλέκτες (multiplexer) και Flip-flop. Το LUT είναι το κύριο στοιχείο για την εφαρμογή της λογικής λειτουργίας. Ο Πολυπλέκτης χρησιμοποιείται για την συλλογή των δεδομένων μεταξύ συνδυαστικής και ακολουθιακής λογικής ενώ τα flip-flops για της εφαρμογή της ακολουθιακής λογικής.

LUT (Look-up-Table)
Ένα από τα πιο σημαντικά στοιχεία της αρχιτεκτονικής FPGA είναι το LUT - είναι ο πυρήνας της αρχιτεκτονικής FPGA. Το LUT έχει σχεδιαστεί για να εφαρμόζει οποιαδήποτε εξίσωση Boolean. Εντός του LUT, υπάρχουν πολυπλέκτες και τα κελιά SRAM που περιέχουν τις εξόδους με βάση τις επιλεγμένες γραμμές. Για τη χρήση k-εισόδων LUT (k-LUT) - ένα LUT που μπορεί να εφαρμόσει οποιαδήποτε λειτουργία k εισόδων - απαιτούνται 2^k bits SRAM και 2^k:1 πολυπλέκτης.

Το Σχήμα 18 δείχνει ένα 3-LUT, το οποίο αποτελείται από 8 bit SRAM και έναν πολυπλέκτη 8:1 που υλοποιείται ως ένα δέντρο πολλυπλεκτών 2:1. Το 3-LUT μπορεί να εφαρμόσει οποιαδή-ποτε λειτουργία 3 εισόδων (A, B, C) ορίζοντας την κατάλληλη τιμή στη μάσκα κελιών LUT SRAM.

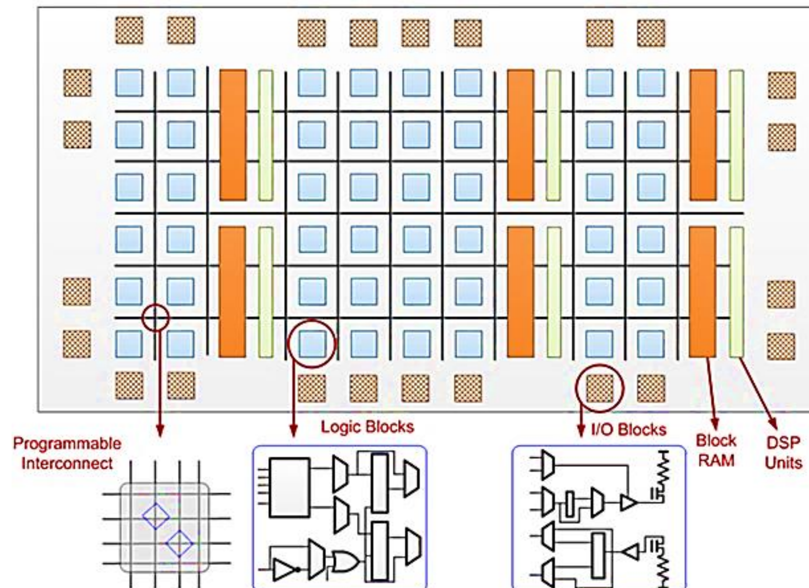


Σχήμα 18 – 3-LUT
απο https://www.researchgate.net

Πολλά LUT συνδεδεμένα μεταξύ τους, υλοποιούν οποιαδήποτε λογική συνάρτηση. Ο Synthesizer (ο αντίστοιχος compiler για τις HDL), ορίζει το κάθε LUT του FPGA να υλοποιήσει ένα μικρό μέρος της συστοιχίας λογικών πυλών, ώστε συνδυαστικά να παραχθεί το τελικό σήμα στην έξοδο.

Interconnect Architecture

Όλα τα λογικά στοιχεία μέσα σε ένα FPGA συνδέονται σε μια διασύνδεση. Η διασύνδεση είναι μια μήτρα δρομολόγησης που αποτελείται από προγραμματιζόμενους διακόπτες και καλώδια. Τα στοιχεία δρομολόγησης παρέχουν μια σύνδεση μεταξύ μπλοκ εισόδου / εξόδου, μπλοκ λογικής και μεταξύ ενός CLB με άλλο CLB. Η προγραμματιζόμενη διασύνδεση, όπως φαίνεται στο Σχήμα 19, αποτελείται από διακόπτες και καλώδια.



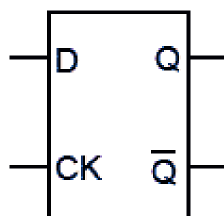
Σχήμα 19 – Interconnect Architecture
από https://towardsdatascience.com/introduction-to-fpga-and-its-architecture-20a62c14421c

Input / Output Blocks

Ένα μπλοκ I/O είναι το μπλοκ εισόδου / εξόδου που μπορεί να χρησιμοποιηθεί τόσο για την είσοδο όσο και για την έξοδο. Οι διαδρομές εισόδου και εξόδου περιέχουν D flip-flops (βλέπε Σχήμα 20) που διεγείρονται από παλμούς ρολογιού (rising-edges). Ο σκοπός των μπλοκ I/O είναι να παρέχουν τη διεπαφή χρήστη από τον εξωτερικό κόσμο στην εσωτερική αρχιτεκτονική του FPGA.

Στα κυκλώματα ακολουθιακής λογικής, το αποτέλεσμα στην έξοδο εξαρτάται και από προηγούμενες καταστάσεις, που έλαβαν χώρα στο κύκλωμα, και όχι μόνον από την κατάσταση των σημάτων στην είσοδο. Για το λόγο αυτό, είναι απαραίτητη η αποθήκευση της πληροφορίας σε κάποιον Καταχωρητή (register), όπως είναι ένα D flip-flop για 1 bit πληροφορίας, που σε συνδυασμό με την ύπαρξη ενός ρολογιού, προωθεί την τιμή στην έξοδο του flip-flop.

Inputs		Outputs	
CK	D	Q	$\overline{Q}$
0	X	No change	
1	0	0	1
1	1	1	0



Σχήμα 20 – D flip-flops
από https://www.electronics-tutorial.net/sequential-logic-circuits/d-flip-flop/

3.2.4.1.2 Hard Blocks

Τα Hard Blocks (εξειδικευμένα μπλοκ) είναι προκατασκευασμένα υποσυστήματα, με σκοπό την εκτέλεση συγκεκριμένων λειτουργιών και κατ᾽ επέκταση την αποδέσμευση πόρων.

Ετσι υπάρχουν μπλοκ για προσθαφαιρέσεις, μπλοκ πολλαπλασιαστών (Multipliers) [31, 32], κλπ., τα οποία ενεργοποιούνται όταν ο Synthesizer αντιληφθεί ότι μέσα στον κώδικα HDL περιλαμβάνονται αριθμητικές πράξεις.

Επίσης υπάρχουν ενσωματωμένες μνήμες RAM (Block RAM) που χρησιμοποιούνται σε περίπλοκες εφαρμογές όπου απαιτείται προσωρινή αποθήκευση πληροφορίας. Η πιο συνηθισμένη εφαρμογή των RAM είναι οι μνήμες FIFO.

Transceivers

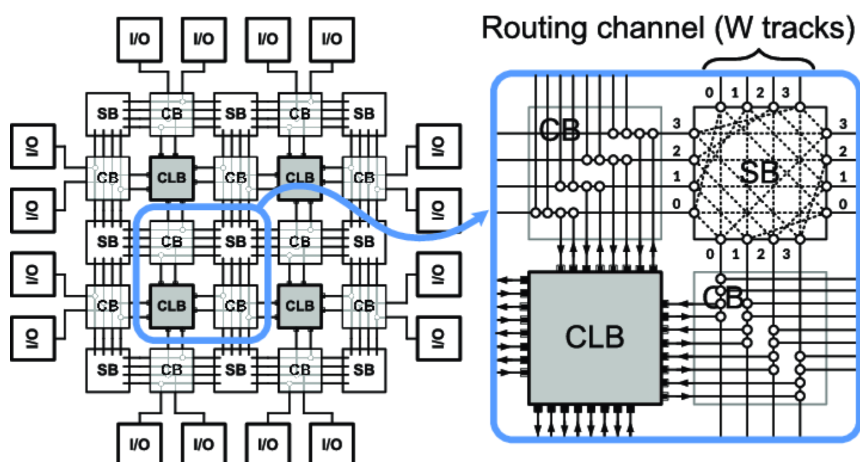
Τα κυκλώματα αυτά είναι υποσυστήματα που υλοποιούν σειριακές διασυνδέσεις υψηλής ταχύτητας για την επικοινωνία του FPGA με τον έξω κόσμο. Μπορούν να υλοποιήσουν μία πληθώρα πρωτοκόλλων επικοινωνίας, όπως PCI Express, SATA, SAS, και Ethernet.

Το FPGA μπορεί να μεταδώσει και να δεχθεί σήματα από πολλά διαφορετικά σημεία.
Τόσο στις εξόδους όσο και στις εισόδους των I/O Blocks, υπάρχουν flip-flops, ώστε τα χρονισμένα σήματα να μπορούν να εξάγονται απευθείας στις ακίδες και να μειώνεται η απαίτηση χρόνου αναμονής του FPGA αντίστοιχα.

Ο Synthesizer σε τελική φάση θα ενεργοποιήσει τα αντίστοιχα μπλοκ και τις διασυνδέσεις που θα μεταφέρουν τα σήματα από και προς τα διαφορετικά μπλοκ.

3.2.4.2 Διασυνδέσεις FPGA

Η μεγαλύτερη επιφάνεια των FPGAs, που φτάνει έως και 90% [33], δεσμεύεται από το εσωτερικό δίκτυο διασυνδέσεων. Η γενική μορφή των FPGAs, όπως συναντάται σήμερα, είναι η Island-Style Architecture [31]. Κάθε λογικό μπλοκ είναι ένα σύμπλεγμα από CLBs, Connection Blocks (CB) και Switch Boxes (SB). Αυτά τα λογικά μπλοκ συνδέονται με άλλα λογικά μπλοκ μέσω σταθερών οριζόντιων και κατακόρυφων καλωδίων και μπορούν να προγραμματιστούν. Οι διασταυρώσεις αυτών των οριζόντιων και κατακόρυφων καλωδίων / κομματιών διαχειρίζονται από κουτιά διακόπτη (Switch Boxes). Προκειμένου να αποφεύγονται φαινόμενα καθυστέρησης, τα σήματα μπορούν να διαδοθούν και με μακρύτερα καλώδια (Long Lines) [31]. Η όλη αρχιτεκτονική, απεικονίζεται στο Σχήμα 21.



Σχήμα 21 – Αρχιτεκτονική τύπου island που εφαρμόζεται στα σημερινά FPGA
από https://www.researchgate.net/figure/Island-style-global-FPGA-architecture
-A-unit-tile-consists-of-Configurable-Logic-Block_fig6_323820898

3.2.5 Σύγκριση κυκλωμάτων παράλληλης επεξεργασίας

Τα ολοκληρωμένα κυκλώματα χρησιμοποιούνται ευρέως σήμερα παγκοσμίως για διαφορετικές εφαρμογές. Τα FPGA παρουσιάζουν πλεονεκτήματα & μειονεκτήματα σε σχέση με τα υπόλοιπα (βλέπε Πίνακα 4).

Το κύριο μειονέκτημα των FPGAs είναι η πολυπλοκότητα σχεδιασμού που απαιτεί μεγάλη τεχνική έρευνα, η οποία συνεπάγεται μεγαλύτερο χρόνο στην αγορά. Σε πολύπλοκα FPGA, μόνο το compilation μπορεί να διαρκέσει ώρες. Η επαλήθευση (validation) του σχεδιασμού μπορεί να διαρκέσει σχεδόν το 70% του χρόνου ανάπτυξης.

Πίνακας 4 – Σύγκριση κυκλωμάτων παράλληλης επεξεργασίας

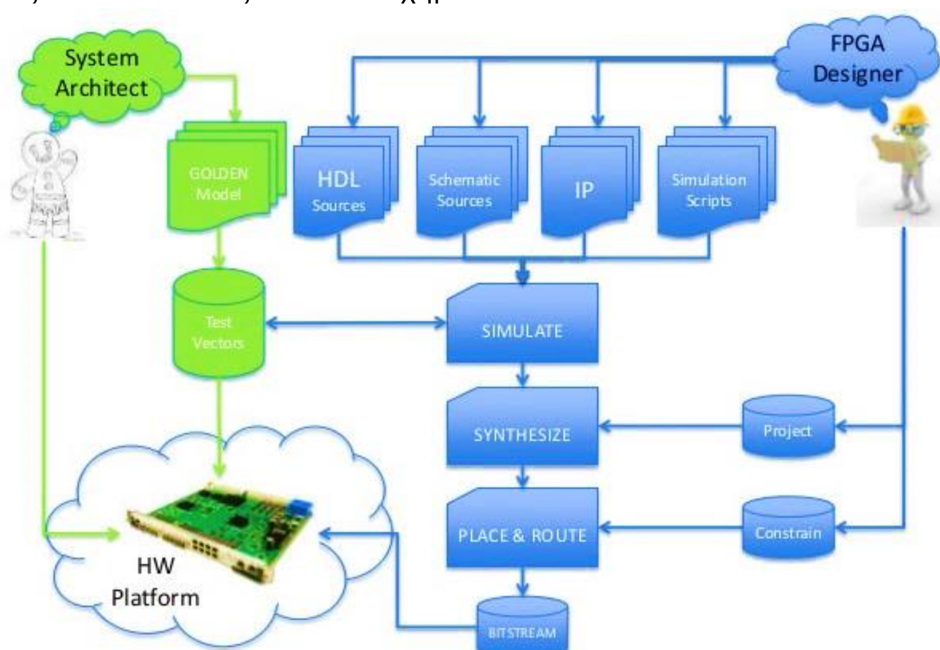
Κύκλωμα	Μέγεθος	Ισχύς	Flexibility	Reliability	Parallelism	Συχνότητα λειτουργίας	Πολυπλ/τα σχεδιασμού	Κόστος
FPGA	+	+	++	++	+++	+	-	-
GPU	-	-	+++	+	++	+++	+	+
ASIC	+++	+++	-	+++	+++	+++	-	(*)
DSP	++	++	+++	+	+	++	++	++
GPP	++	++	+++	+	+	+++	+++	+++

* Στην περίπτωση του ASIC, το κόστος κατασκευής εξαρτάται από το είδος της παραγωγής. Για μαζική παραγωγή, το κόστος αποσβένεται, ωστόσο, για τιμή/μονάδα παραγωγής, ο σχεδιασμός ASIC είναι πολύ ακριβός.

3.3 Σχεδιασμός με FPGAς

Κατά τη χρήση ενός FPGA, ο σχεδιαστής πρέπει να λαμβάνει υπόψη την αρχιτεκτονική της τελικής εφαρμογής, για να λύσει αποτελεσματικά ένα πρόβλημα χρησιμοποιώντας με τον καλύτερο τρόπο τους διαθέσιμους πόρους υλικού. Ο κώδικας εφαρμόζει φυσικές συνδέσεις μεταξύ ηλεκτρονικών κυκλωμάτων. Αυτά τα ηλεκτρονικά κυκλώματα ακολουθούν φυσικούς νόμους που περιορίζουν τη χρήση του (καθυστερήσεις διάδοσης, fan out, κατανάλωση ενέργειας, ...). Ο σχεδιαστής πρέπει να λάβει υπόψη όλες τις παραμέτρους για να επωφεληθεί από την αρχιτεκτονική FPGA.

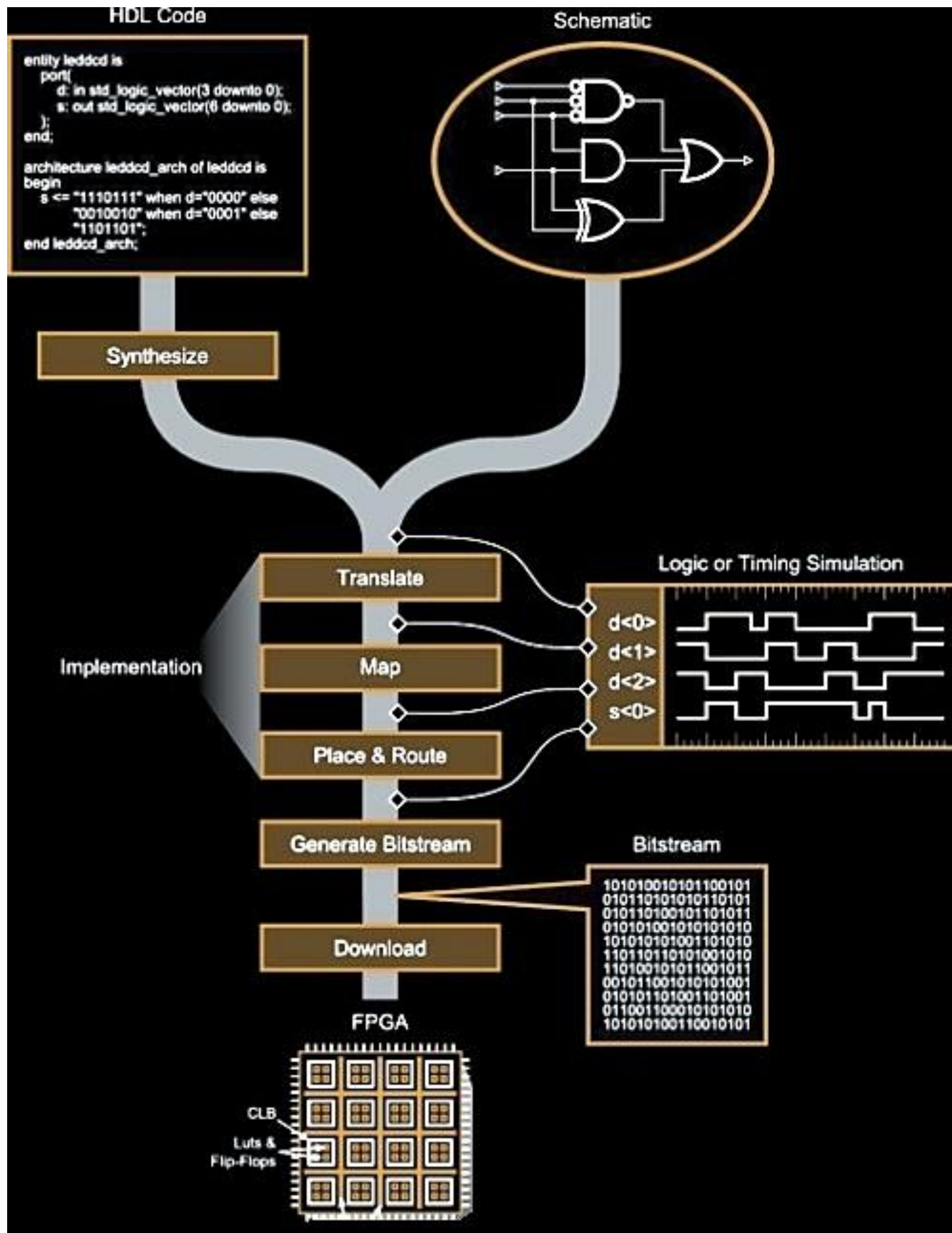
Τα πιο σημαντικά βήματα και εργαλεία της ροής σχεδιασμού για την παραγωγή κυκλώματος FPGA απεικονίζονται στο Σχήμα 22.



Σχήμα 22 – Σχεδιασμός FPGA
από https://slidetodoc.com/fpga-for-dummies-design-flow-ess-fpga-for/

Οι κύριες σχεδιαστικές καταχωρήσεις για την ανάπτυξη εφαρμογών βασισμένες σε FPGA είναι τα αρχεία περιγραφής υλικού και το αρχείο περιορισμών Constraint.

Τα αρχεία περιγραφής υλικού δημιουργούνται από τον χρήστη χρησιμοποιώντας μια γλώσσα περιγραφής υλικού (HDL) ή μπορούν να παρασχεθούν από τρίτο ή τον κατασκευαστή FPGA. Συνήθως, ένας σχεδιασμός περιέχει πολλά αρχεία HDL οργανωμένα με ιεραρχικό τρόπο (βλέπε Σχήμα 23).



Σχήμα 23 – Ιεραρχία αρχείων HDL
από https://allaboutfpga.com/fpga-design-flow/

Οι προδιαγραφές περιορισμών συνήθως πραγματοποιούνται χρησιμοποιώντας αρχεία κειμένου. Το αρχείο περιορισμών (SDC για Altera και UCF για Xilinx) καθορίζει σημαντικά χαρακτηριστικά, όπως λειτουργίες συχνότητας, καθυστερήσεις, αντισταθμίσεις χρονισμού, διαδρομές πολλαπλών κύκλων, πληροφορίες τοποθέτησης και δρομολόγησης κ.λ.π.

3.3.1 Γλώσσα περιγραφής υλικού (HDL)

Οι γλώσσες περιγραφής υλικού (HDL) μπορούν να χρησιμοποιηθούν για να περιγράψουν τη λειτουργία ψηφιακού ή αναλογικού ηλεκτρονικού κυκλώματος. Με την HDL, η λειτουργική περιγραφή ενός ψηφιακού συστήματος γίνεται χρησιμοποιώντας τυποποιημένες εκφράσεις που βασίζονται σε κείμενο. Οι γλώσσες HDL που χρησιμοποιούνται περισσότερο είναι οι VHDL, Verilog και System Verilog.

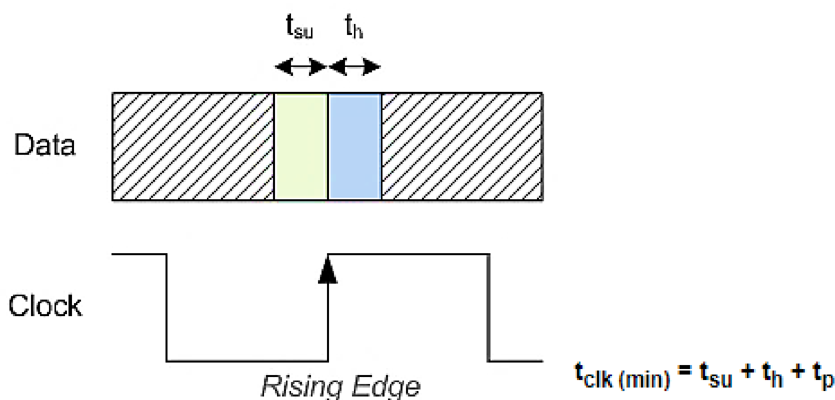
Το κύριο χαρακτηριστικό μιάς HDL είναι ότι επιτρέπει τον ορισμό των ταυτόχρονων δηλώσεων, επιτρέποντας την περιγραφή πολλών διαδικασιών που εκτελούνται παράλληλα.

Για την ανάπτυξη αυτής της εργασίας χρησιμοποιήσαμε την VHDL, η οποία είναι μια πολύ πλούσια γλώσσα που χρησιμοποιείται ευρέως σε όλο τον κόσμο. Η ανάπτυξη της VHDL ξεκίνησε το 1981 από το Υπουργείο Άμυνας των Ηνωμένων Πολιτειών. Σήμερα, υπάρχουν τρεις κύριες αναθεωρήσεις του VHDL (1993, 2000 και 2002). Η VHDL-1993 είναι η πιο κοινή χρησιμοποιούμενη αναθεώρηση.

3.3.2 Setup και Hold

Λόγω της υψηλής πυκνότητας της τελευταίας γενεάς FPGA’s και της υψηλής συχνότητας που απαιτείται από ορισμένους αλγόριθμους, πρέπει να ληφθούν υπόψη οι καθυστερήσεις διάδοσης που προκαλούνται από τις λογικές διαδρομές.

Ένα Flip-Flop, που υπάρχει στο LE, καθορίζει ένα χρόνο Setup και Hold για να εγγυηθεί μια προβλέψιμη συμπεριφορά. Τα δεδομένα που φθάνουν στο Flip-Flop πρέπει να ισχύουν τουλάχιστον έναν καθορισμένο χρόνο (setup) πριν από τον ανερχόμενο παλμό ρολογιού και πρέπει να παραμείνουν έγκυρα τουλάχιστον έναν άλλο καθορισμένο χρόνο (hold) μετά από αυτό (βλέπε Σχήμα 24).



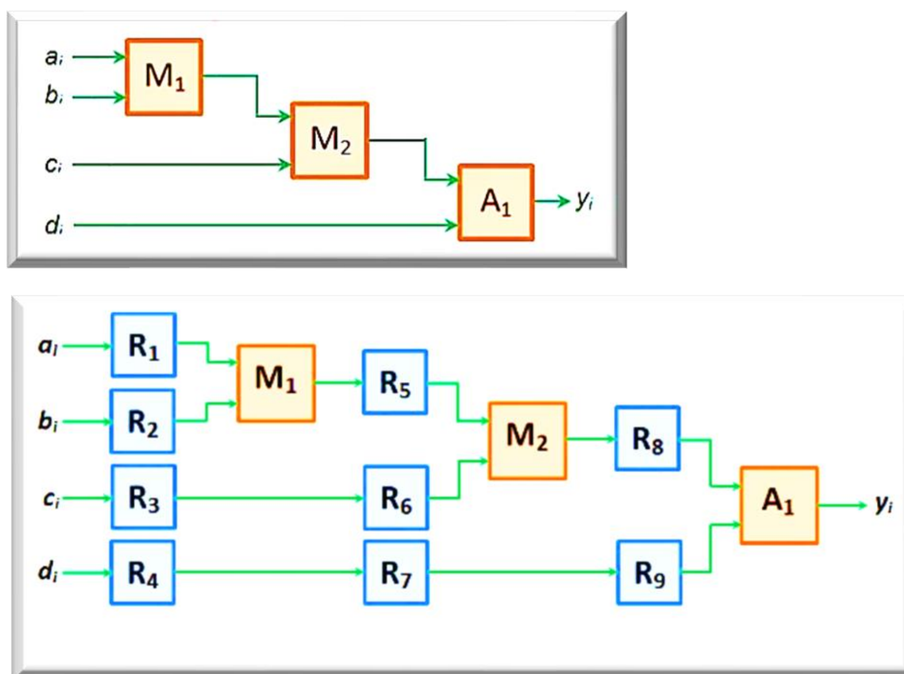
Σχήμα 24 – Setup και Hold
από https://www.nandland.com/articles/setup-and-hold-time-in-an-fpga.html

Όταν υπάρχουν παραβιάσεις των χρονισμών αυτών, μπορεί να συμβεί μια κατάσταση μεταστατικότητας, με αποτέλεσμα τη μη σωστή λειτουργία του συστήματος. Στη σχεδίαση FPGA, οι απαιτήσεις χρονισμού είναι ένα από τα πιο περίπλοκα τμήματα της διαδικασίας σχεδίασης. Για την ικανοποίηση των απαιτήσεων χρονισμού και την αποφυγή προβλημάτων μεταστατικότητας χρησιμοποιούνται τεχνικές pipeline.

3.3.3 Pipelining

Ο σκοπός του pipelining είναι να αυξήσει τη μέγιστη συχνότητα λειτουργίας ενός συστήματος. Έτσι, μια συνδυαστική λειτουργία υποδιαιρείται σε βήματα και υπολογίζεται σε πολλαπλούς κύκλους ρολογιού. Η συχνότητα λειτουργίας ενός συστήματος FPGA περιορίζεται από τον αριθμό των επιπέδων λογικής που πρέπει να διασχίσει το σήμα. Τα λογικά στοιχεία που υπάρχουν σε ένα FPGA δεν είναι τέλεια και τα κατασκευαστικά στοιχεία συνεπάγονται καθυστερήσεις διάδοσης. Εάν ένα σήμα πρέπει να μεταδοθεί μέσω περισσότερων λογικών στοιχείων, οι καθυστερήσεις θα είναι πιο σημαντικές και οι χρόνοι Setup και Hold ενδέχεται να μην ικανοποιούνται.

Το Σχήμα 25 δείχνει δύο διαφορετικούς τρόπους υπολογισμού της ίδιας αριθμητικής λειτουργίας. Στην υλοποίηση του κάτω μέρους, προστέθηκαν καταχωρητές μεταξύ των πολλαπλασιαστών & προσθετέων. Θα περίμενε κανείς ότι το κύκλωμα στο επάνω μέρος λειτουργεί πιο γρήγορα από το κύκλωμα στο κάτω μέρος, ωστόσο αυτό δεν είναι προφανές, όταν οι εργασίες πρέπει να γίνουν με υψηλή συχνότητα και οι καθυστερήσεις διάδοσης είναι σημαντικές, κάτι που συμβαίνει στο μεγαλύτερο ποσοστό των περιπτώσεων σχεδιασμού FPGAs.



Σχήμα 25 – Pipelined vs. not pipelined
από https://www.allaboutcircuits.com/technical-articles/why-how-pipelining-in-fpga/

Κατά την προσθήκη αγωγού - pipeline, απαιτείται ο συγχρονισμός δεδομένων. Συνήθως, ένα σύστημα με στάδια pipelines έχει περισσότερη καθυστέρηση, λόγω των καταχωρητών που προστίθενται, ωστόσο, μετά από έναν αρχικό λανθάνοντα χρόνο σε κάθε κύκλο ρολογιού, ένα νέο αποτέλεσμα λαμβάνεται από το σύστημα. Αυτή η ευελιξία αντισταθμίζεται με την υψηλότερη συχνότητα λειτουργίας, επιτρέποντας καλύτερη απόδοση του συστήματος.

3.3.4 Λειτουργίες Floating Point σε FPGA

Οι λειτουργίες κινητής υποδιαστολής (floating point) είναι δαπανηρές σε πόρους όταν εκτελούνται σε FPGA, ωστόσο, σε ορισμένους αλγόριθμους είναι υποχρεωτικές. Σε ένα FPGA, μια λειτουργία fixed point μπορεί να εκτελεστεί σε ένα μόνο κύκλο ρολογιού σε υψηλή συχνότητα και χρησιμοποιώντας μικρή ποσότητα πόρων. Στην περίπτωση πολλαπλασιασμού κινητής υποδιαστολής απαιτούνται περισσότεροι πόροι, διότι δεν υπάρχουν διαθέσιμοι πολλαπλασιαστές κινητής υποδιαστολής πυριτίου στα τρέχοντα FPGAs και έτσι αυτό το είδος των λειτουργιών υλοποιείται χρησιμοποιώντας LE.

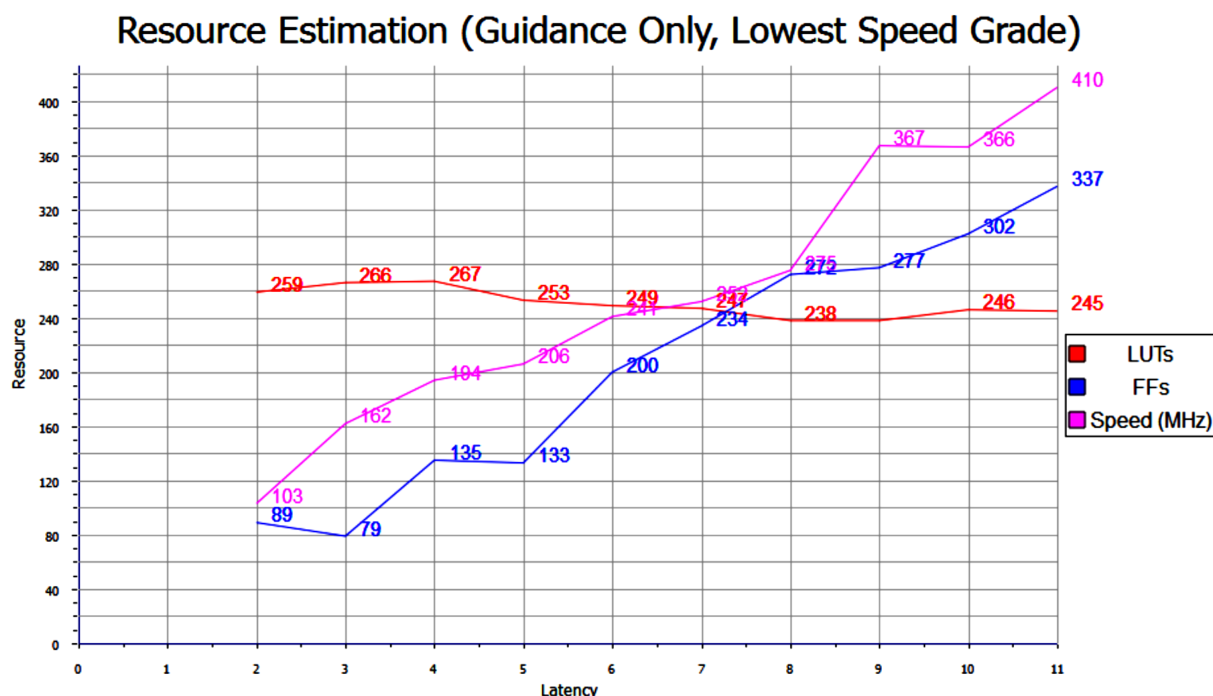
Κατά την χρήση ενός ενσωματωμένου επεξεργαστή, μια μονάδα κινητής υποδιαστολής (FPU - Floating Point Unit) μπορεί να συνδεθεί με το σύστημα για την εφαρμογή λειτουργιών κινητής υποδιαστολής.

Στην περίπτωση μόνο λογισμικού, μπορεί να εκτελεστεί μόνο μία λειτουργία κινητής υποδιαστολής τη φορά. Η εκτέλεση αυτής της λειτουργίας απαιτεί πολλούς κύκλους ρολογιού για να ολοκληρωθεί εάν δεν χρησιμοποιείται pipelining.

Αντίθετα σε μια FPU, οι λειτουργίες είναι σωληνωτές και μια νέα λειτουργία αρχικοποιείται σε κάθε κύκλο ρολογιού FPU. Συνολικά, ο PPC (PowerPC) μπορεί να συνεχίσει να εκτελεί οδηγίες ενώ η FPU υπολογίζει άλλες λειτουργίες. Έτσι, εκτελούνται περισσότερες από μία λειτουργίες την ίδια στιγμή.

Οι λειτουργίες κινητής υποδιαστολής μπορούν επίσης να εφαρμοστούν χωρίς ενσωματωμένο επεξεργαστή. Διαφορετικές υλοποιήσεις FPU διανέμονται ως κωδικοί ανοιχτού κώδικα. Οι κατασκευαστές FPGA παρέχουν επίσης IP cores για τη σύνθεση λειτουργιών κινητής υποδιαστολής.

Το Σχήμα 26 δείχνει ένα γράφημα που συγκρίνει διαφορετικές υλοποιήσεις προσθέσεων single precision κινητής υποδιαστολής χρησιμοποιώντας Virtex 5 FPGA. Χρησιμοποιώντας αυτό το γράφημα μπορεί να επιλεγεί η καθυστέρηση των λειτουργιών ανάλογα με τη συχνότητα λειτουργίας και τους πόρους που απαιτούνται. Η υλοποίηση με το μικρότερο λανθάνοντα χρόνο έχει τη χειρότερη απόδοση χρονισμού. Όταν ο σχεδιασμός επιτρέπει περισσότερη καθυστέρηση, χρησιμοποιούνται τεχνικές αγωγών για τη βελτίωση της απόδοσης χρονισμού. Στην καλύτερη περίπτωση, μπορεί να επιτευχθεί μέγιστη απόδοση 410 MHz.



Σχήμα 26 – Floating point addition latency, συχνότητα λειτουργίας και πόροι
από Xilinx Virtex 5 documentation

3.3.5 Διαχείριση μνήμης FPGA

Κατά το σχεδιασμό με ενσωματωμένα συστήματα, η διαχείριση της μνήμης μπορεί να καταστεί εμπόδιο εάν δεν ληφθούν οι σωστές αποφάσεις.

Το κύριο πλεονέκτημα της εσωτερικής μνήμης του FPGA είναι ότι πολλές μνήμες είναι προσβάσιμες παράλληλα, με πολύ υψηλή ταχύτητα και σχετικά εύκολα. Χρησιμοποιώντας την εσωτερική RAM, ο χρήστης μπορεί να δημιουργήσει διαφορετικές αρχιτεκτονικές αποθήκευσης, όπως FIFOs, πραγματική μνήμη διπλής θύρας, απλή μνήμη διπλής θύρας κ.λ.π.

Ο κύριος περιορισμός των εσωτερικών μνημών είναι το μέγεθος. Για αλγόριθμους επεξεργασίας εικόνας, η αποθήκευση ολόκληρης της εικόνας στην εσωτερική μνήμη RAM απαιτεί αρκετούς πόρους.

Όταν ένας σχεδιασμός χρειάζεται επιπλέον μνήμη, μπορεί να χρησιμοποιηθεί εξωτερική μνήμη. Η επιλογή της μνήμης εξαρτάται από το απαιτούμενο μέγεθος, τη συχνότητα λειτουργίας και τον τύπο πρόσβασης.
Η SDRAM είναι μια μνήμη υψηλής πυκνότητας και πολύ υψηλής ταχύτητας. Το κύριο μειονέκτημα της μνήμης SDRAM είναι ότι εάν οι προσβάσεις δεν γίνουν διαδοχικά (sequentially), η απόδοση επηρεάζεται δραστικά. Στις υπάρχουσες πλακέτες FPGA μπορούμε να βρούμε μνήμες DDR2 - DDR4 SDRAM.

Η SRAM είναι κατασκευασμένη με τεχνολογία ακριβότερη από την SDRAM. Το μέγεθος της SRAM που διατίθεται στην αγορά είναι μικρότερο από αυτό της SDRAM. Το κύριο πλεονέκτημα της μνήμης SRAM είναι η δυνατότητα πρόσβασης με μη διαδοχικούς τρόπους χωρίς χρονική καθυστέρηση.

Η μνήμη Flash είναι μια μη πτητική μνήμη που χρησιμοποιείται συνήθως για την αποθήκευση δεδομένων διαμόρφωσης. Έχει το μειονέκτημα ότι είναι σχετικά αργή, οπότε για τη λειτουργία σε πραγματικό χρόνο, δεν είναι επιθυμητή η εργασία απευθείας από αυτή· εναλλακτικά ο χρήστης πρέπει να αντιγράψει τα απαραίτητα δεδομένα στη μνήμη RAM, επιτρέποντας έτσι την πρόσβαση σε αυτές τις πληροφορίες σε πραγματικό χρόνο. Οι κατασκευαστές FPGA διαθέτουν διαφορετικό ελεγκτή μνήμης για τη διαχείριση εξωτερικών μνημών.

3.4 Ενσωματωμένοι επεξεργαστές FPGA

Οι ενσωματωμένοι επεξεργαστές υπάρχουν σχεδόν στα 2/3 των νέων ενσωματωμένων συστημάτων. Τα τελευταία χρόνια η τάση να χρησιμοποιούνται επεξεργαστές ταυτόχρονα με τα FPGAs έχει αυξηθεί. Αυτός ο τομέας της έρευνας ονομάζεται Co-Design. Κατά το σχεδιασμό με FPGA, η χρήση ενός επεξεργαστή μπορεί να απλοποιήσει πολύ τη δουλειά του σχεδιαστή.

Ορισμένες μη κρίσιμες εργασίες, όπως η διαχείριση της αργής ταχύτητας περιφερειακών, μπορούν να σχεδιαστούν ευκολότερα χρησιμοποιώντας έναν επεξεργαστή. Με αυτόν τον τρόπο οι σχεδιαστές FPGA μπορούν να επικεντρώσουν τις προσπάθειές τους στην επιτάχυνση των εργασιών που απαιτούν κρίσιμους χρονισμούς. Ο σχεδιασμός με έναν επεξεργαστή είναι ευκολότερος από το σχεδιασμό απλώς χρησιμοποιώντας τα LE του FPGA, λόγω των εκτιμήσεων που σχετίζονται με το χρονισμό και τους πόρους κατά το σχεδιασμό με υλικό.

Το 2002, η Xilinx ενσωμάτωσε έναν επεξεργαστή PowerPC και σήμερα οι δύο ηγέτες FPGA (Xilinx και Altera) προσφέρουν διαφορετικές αρχιτεκτονικές χρησιμοποιώντας ενσωματωμένους επεξεργαστές. Το 2012, οι προμηθευτές FPGA άρχισαν να συγκλίνουν σε μια παρόμοια αρχιτεκτονική που βασίζεται σε επεξεργαστές ARM.

Σήμερα, μπορούμε να βρούμε δύο τύπους επεξεργαστών: soft επεξεργαστές και hard επεξεργαστές. Οι soft επεξεργαστές υλοποιούνται εξ ολοκλήρου χρησιμοποιώντας τους πόρους του FPGA (Slices, BRAM, κ.λπ.), και συνήθως περιγράφονται χρησιμοποιώντας μια γλώσσα περιγραφής υλικού. Οι hard επεξεργαστές είναι επεξεργαστές κατασκευασμένοι από πυρίτιο και συνδέονται με τους υπόλοιπους λογικούς πόρους του FPGA.

Η Altera προσφέρει τον soft επεξεργαστή Nios II. Η Xilinx προσφέρει soft επεξεργαστές PicoBlaze και MicroBlaze. Η Xilinx παρέχει επίσης hard-core επεξεργαστές, συμπεριλαμβανομένων των PowerPC440 και PowerPC405.

Ορισμένα Xilinx FPGA προσφέρουν διπύρηνους PowerPC ή ARM επεξεργαστές και εάν ο σχεδιαστής χρειάζεται επιπλέον, έχει την επιλογή να συνδέσει έναν ή περισσότερους επιπλέον soft επεξεργαστές στο σύστημα. Θεωρητικά, το όριο του αριθμού των soft επεξεργαστών που πρέπει να ενσωματωθούν σε ένα FPGA εξαρτάται από το μέγεθος του FPGA.

3.4.1 NIOS II

Ο Nios II είναι ένας soft επεξεργαστής 32 bit από την Altera με αρχιτεκτονική RISC (βλέπε Σχήμα 27). Σε αυτόν τον επεξεργαστή, τα instruction και data buses διαχωρίζονται (Harvard Architecture). Ο επεξεργαστής συνδέεται χρησιμοποιώντας το Avalon Switch Fabric Bus.

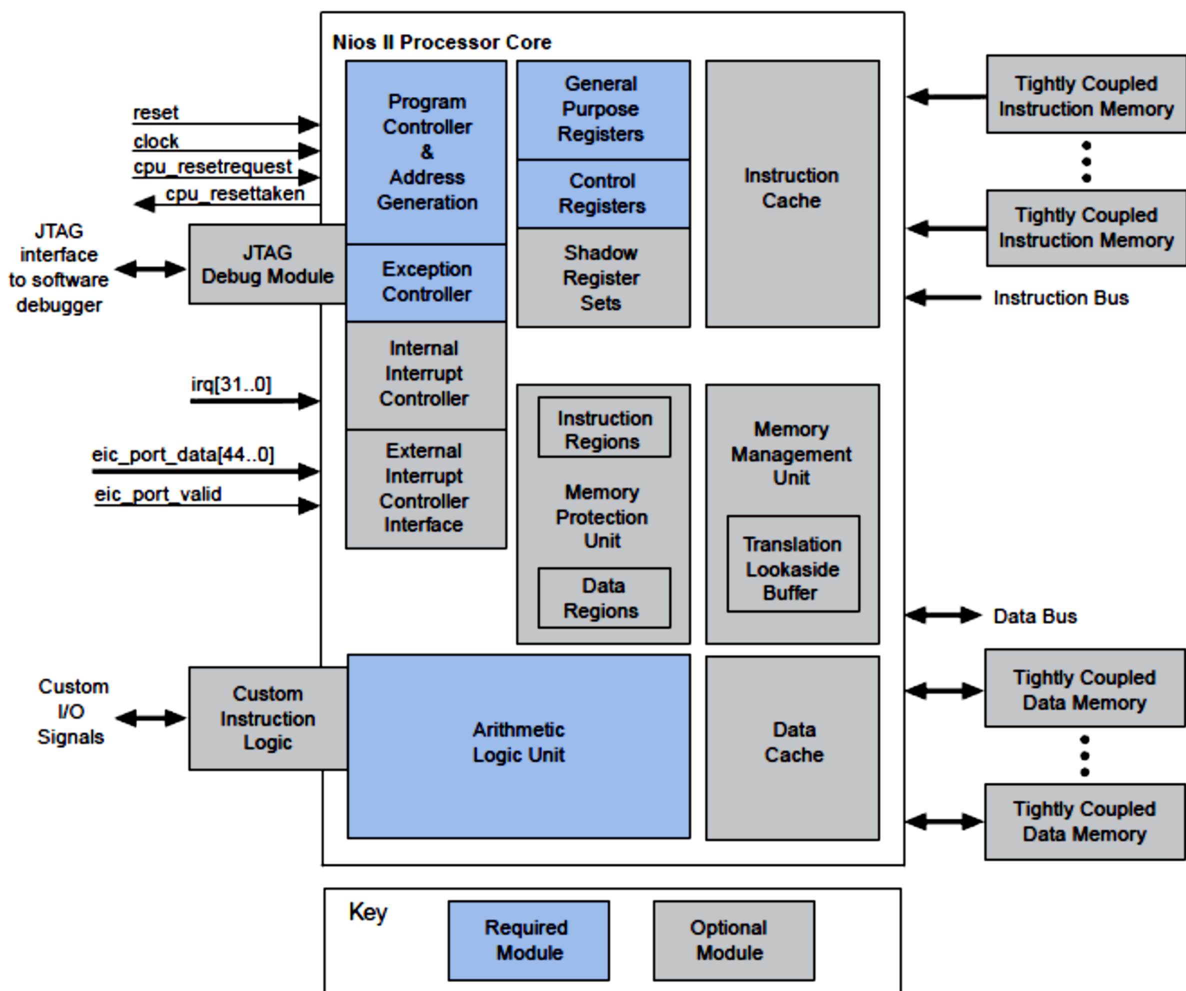
Nios® II

Σύμφωνα με τις ανάγκες της εφαρμογής, το NIOS II διατίθεται σε τρεις διαφορετικές διαμορφώσεις:

(Γρήγορη): μέγιστη απόδοση

(Ισορροπία): μεταξύ απόδοσης και κόστους.

(Μικρότερο): προορίζεται για χρήση σε FPGA χαμηλού κόστους.



Σχήμα 27 – NIOS II μπλόκ διάγραμμα

από https://hackaday.com/2018/10/05/easy-fpga-cpu-with-max1000/

3.4.2 Λειτουργικό σύστημα πραγματικού χρόνου (RTOS)

Σε ορισμένες εφαρμογές που χρησιμοποιούν επεξεργαστές, πρέπει να εκτελούνται περισσότερα από ένα threads. Ένα σύστημα λειτουργίας επιτρέπει την εκτέλεση πολλαπλών threads χρησιμοποιώντας μια πολυπλεξία χρόνου, επιτρέποντας την κοινή χρήση πόρων μεταξύ των νημάτων.

Ένα λειτουργικό σύστημα διαχειρίζεται τους πόρους ενός επεξεργαστή, αποτελώντας τη διεπαφή μεταξύ των εκτελεσθέντων threads και των πόρων υλικού. Υπάρχουν δύο τύποι λειτουργικών συστημάτων: General Purpose Operating Systems (GPOS) και Real Time Operating Systems (RTOS). Η κύρια διαφορά μεταξύ των RTOS και GPOS είναι ο τρόπος με τον οποίο το λειτουργικό πρόγραμμα προγραμματίζει όλα τα συμβάντα.

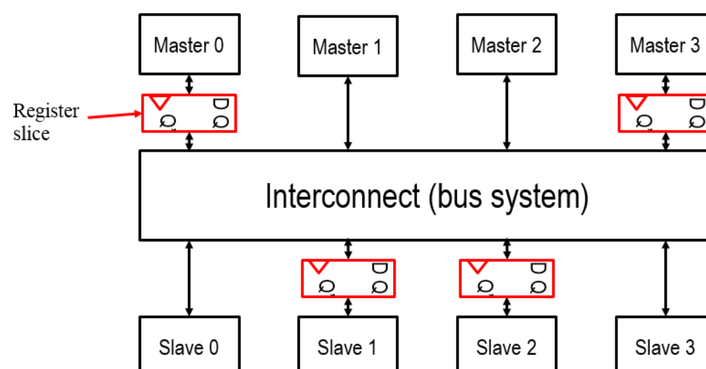
Ένα RTOS εγγυάται ντετερμινιστική και σύντομη απόκριση για συμβάντα. Στο RTOS, τα κρίσιμα threads εκτελούνται κατά προτεραιότητα και μόνο ένα thread υψηλότερης προτεραιότητας μπορεί να διακόψει την εκτέλεση άλλων threads. Αυτό το σχήμα προτεραιότητας των εκτελέσεων thread επιτρέπει την ικανοποίηση των απαιτήσεων χρονισμού των κρίσιμων εργασιών.

Ένα GPOS δεν εγγυάται μέγιστους χρόνους απόκρισης, αποτυγχάνοντας να ολοκληρώσει εγκαίρως την εκτέλεση κάποιων κρίσιμων εργασιών. Κατά τη χρήση ενός GPOS, ο χρήστης δεν έχει τον ακριβή έλεγχο της σειράς εκτέλεσης του λογισμικού. Στην έκδοση 2.6 του Linux έχει προστεθεί ένα είδος προτίμησης, επιτρέποντας εργασίες να διακοπούν ανάλογα με την προτεραιότητά τους.

3.5 Αρχιτεκτονικές Bus FPGA

Το Bus είναι μια υποδομή που υποστηρίζει τη μεταφορά δεδομένων μεταξύ συσκευών masters και slaves. Σε μια αρχιτεκτονική διαύλου, τα στοιχεία που συνδέονται με το bus αντιστοιχίζονται στη μνήμη. Για την σύνδεση του επεξεργαστή με άλλους πόρους υλικού, που είναι διαθέσιμοι στο σύστημα, υπάρχουν ορισμένα τυποποιημένα buses και διασυνδέσεις. Αυτές οι διασυνδέσεις καθορίζουν το πρωτόκολλο επικοινωνίας μεταξύ του επεξεργαστή και της λογικής.

Το Σχήμα 28 δείχνει ένα τυπικό σύστημα, συμπεριλαμβανομένου ενός διαύλου διασύνδεσης. Μπορούμε να δούμε πολλούς masters και slaves· για παράδειγμα, ο Master 0 μπορεί να είναι επεξεργαστής και ο Master 3 ενα DMA controller, και οι δύο έχουν πρόσβαση στον Slave 1, που μπορεί να είναι μνήμη. Σε αυτό το σενάριο, κάποιος πρέπει να ρυθμίζει το δίαυλο και να δίνει στο DMA πρόσβαση στο δίαυλο όταν ο επεξεργαστής δεν τον χρησιμοποιεί. Αυτό σημαίνει ότι πρέπει να οριστεί κάποιο είδος σηματοδότησης για να δημιουργηθεί η επικοινωνία χωρίς συγκρούσεις.



Σχήμα 28 – Interconnection Bus

Η Altera χρησιμοποιεί ένα κεντρικό Bus που ονομάζεται Avalon Switch Fabric για τη σύνδεση του επεξεργαστή με το υπόλοιπο λογικό κύκλωμα. Η Xilinx χρησιμοποιεί ένα παρόμοιο Bus που ονομάζεται Processor Local Bus (PLB).

Στα τελευταία FPGA οι κύριοι κατασκευαστές συγκλίνουν να χρησιμοποιούν επεξεργαστές ARM με την ίδια αρχιτεκτονική διαύλου AXI4. Χάρη σε αυτό, η ευκολία του σχεδιασμού μεταξύ ολοκληρωμένων κυκλωμάτων διαφορετικών κατασκευαστών γίνεται ευκολότερη.

3.6 Ροή ανάπτυξης FPGA

Υπάρχουν δύο βασικές ροές ανάπτυξης:

- Η περιγραφή βασίζεται σε εργαλεία High Level Synthesis (HLS).
- Περιγραφή του συστήματος που χρησιμοποιεί γλώσσες HDL

3.6.1 Ροή HLS

Στην [34] παρουσιάζεται μια πλήρης ανασκόπηση διαφορετικών εργαλείων HLS. Σύμφωνα με αυτή, βρισκόμαστε σε ένα μεταβατικό στάδιο χάρη στην πρόοδο των εργαλείων τελευταίας γενιάς HLS, σε συνδυασμό με την υψηλή πυκνότητα των τελευταίων FPGA και την πολυπλοκότητα των νέων εφαρμογών.

Σύμφωνα με την [36], ένας σχεδιασμός 1-M gate elements απαιτεί 300k γραμμές κώδικα RTL, με αποτέλεσμα την αύξηση 7x έως 10x της πολυπλοκότητας του σχεδιασμού σε σύγκριση με την υλοποίηση HLS.

Ένα μεγάλο πλεονέκτημα των εργαλείων HLS είναι ότι αυτή η ροή διευκολύνει τη συν-σχεδίαση λογισμικού / hardware, λόγω της ίδιας γλώσσας προγραμματισμού που χρησιμοποιείται για την περιγραφή του υλικού και τον προγραμματισμό του ενσωματωμένου επεξεργαστή. Οι γλώσσες υψηλού επιπέδου είναι πιο συμπαγείς, καθιστούν γρηγορότερη την ανάπτυξη προγραμμάτων και προσφέρουν καλύτερη αναγνωσιμότητα από την HDL.

3.6.2 Διαδικασία σχεδίασης κώδικα RTL

Το πρώτο βήμα για τη ροή ανάπτυξης, είναι ο προσδιορισμός και ο ορισμός των απαιτήσεων του συστήματος. Από τις απαιτήσεις του συστήματος, μια υλοποίηση λογισμικού δημιουργείται για την επαλήθευση των αλγορίθμων και της υπόθεσης που προτείνεται. Η υλοποίηση του λογισμικού γίνεται με τη χρήση ενός γενικού σκοπού επεξεργαστή και τη χρήση διαθέσιμων βιβλιοθηκών που θα μπορούσαν να διευκολύνουν την ανάπτυξη λογισμικού. Η υλοποίηση του λογισμικού γίνεται χρησιμοποιώντας γλώσσες υψηλού επιπέδου όπως SystemC, Matlab, C ή C++.

Μετά την επαλήθευση των αλγορίθμων, πρέπει να ληφθεί η απόφαση για τη χρήση μόνο software, software ή hardware ή συν-σχεδιασμού. Πρέπει να γίνει μια αρχική προσέγγιση σχετικά με την ανάγκη χρήσης ενός επεξεργαστή, καθώς και το είδος του επεξεργαστή. Ο software σχεδιασμός είναι ευκολότερος από τον σχεδιασμό hardware, πράγμα που σημαίνει ότι εάν ένας επεξεργαστής είναι διαθέσιμος, ο σχεδιασμός θα πρέπει να υλοποιηθεί ως επί το πλείστον στο software. Η χρήση ενός επεξεργαστή επιτρέπει τη μείωση του χρόνου ανάπτυξης.

Σε αυτό το σημείο σχεδιάζεται η αρχιτεκτονική, αναγνωρίζονται οι διάφορες εργασίες υλικού και λογισμικού. Δημιουργούνται διαγράμματα μπλοκ για τον προσδιορισμό των διαφορετικών λειτουργικών μονάδων του συστήματος. Μια επίπεδη αρχιτεκτονική μπορεί να χρησιμοποιηθεί για την υλοποίηση του σχεδιασμού. Ωστόσο, μια ιεραρχική προσέγγιση μπορεί να διευκολύνει την συνολική εργασία.

Σε έναν ιεραρχικό σχεδιασμό, μια σύνθετη ενότητα (module) υποδιαιρείται σε μικρές ενότητες, όπου η κάθε μια είναι εξειδικευμένη για την υλοποίηση ενός μέρους του σχεδιασμού. Το κύριο πλεονέκτημα του ιεραρχικού σχεδιασμού είναι ότι διευκολύνει την αναγνωσιμότητα και τον εντοπισμό σφαλμάτων.

Είναι βολικό να σχεδιάζονται προσωρινά διαγράμματα των κύριων σημάτων, τα οποία θα είναι χρήσιμα για τη διαδικασία επαλήθευσης. Σε αυτό το βήμα του σχεδιασμού, το target FPGA πρέπει να επιλεγεί ώστε να εκμεταλευτούμε τους διαθέσιμους πόρους και να επιλεγεί το μέγεθος του FPGA που απαιτείται.

Είναι σύνηθες να επιλέγεται μέγεθος FPGA μεγαλύτερο από το απολύτως απαραίτητο για την πρόβλεψη μελλοντικής αναβάθμισης του σχεδιασμού και διευκόλυνση των εργασιών εντοπισμού σφαλμάτων. Ποτέ δεν είναι δυνατόν να χρησιμοποιηθεί το 100% των διαθέσιμων πόρων του FPGA.

Μόλις οριστεί η αρχιτεκτονική, τα διάφορα modules πρέπει να δημιουργηθούν. Εάν χρησιμοποιούνται ειδικοί πόροι υλικού (FIFO, buffer, Gigabit πομποδέκτες, ...), μπορούν να δημιουργηθούν χρησιμοποιώντας ένα IPCore generator που παρέχεται από τον κατασκευαστή FPGA.

Μετά την κωδικοποίηση και τη δημιουργία της αρχιτεκτονικής, το πρόγραμμα προβολής RTL μπορεί να χρησιμοποιηθεί για τη δημιουργία του σχηματικού διαγράμματος. Αυτό το διάγραμμα μπορεί να συγκριθεί με τα διαγράμματα μπλοκ που υλοποιήθηκαν κατά την ανάπτυξη των προδιαγραφών του συστήματος. Μπορεί να επαληθευτεί εάν ο synthesizer σύνθεσε το σωστό λογικό κύκλωμα σύμφωνα με τις απαιτήσεις του σχεδιασμού. Επίσης, μπορούν να εντοπιστούν λανθασμένες βελτιστοποιήσεις, ανεπιθύμητα latches που συνάγονται από ελλιπείς δηλώσεις και άλλα σφάλματα.

Στη συνέχεια, όλες οι λειτουργικές μονάδες πρέπει να ελεγχθούν χρησιμοποιώντας προσομοίωση χαμηλού επιπέδου. Αυτή η πρώτη προσομοίωση δεν λαμβάνει υπόψη τις καθυστερήσεις που υπάρχουν στους διαφορετικούς λογικούς πόρους. Τα test-benches δημιουργούνται χρησιμοποιώντας κώδικα HDL που δημιουργεί διαφορετικό σήμα διέγερσης (stimulus) στα μπλοκ.

Εάν η συμπεριφορά προσομοίωσης δεν συμφωνεί με τη σχεδιαζόμενες προδιαγραφές, το σφάλμα πρέπει να εντοπιστεί. Πρέπει να προσδιοριστεί εάν το σφάλμα προκύπτει από τη συσκευή υπό δοκιμή (DUT) ή από το test-bench. Εάν το σφάλμα προέρχεται από την DUT, πρέπει να επαληθευτεί εάν πρόκειται για σφάλμα γραφής HDL ή εάν προέρχεται από τη σχεδιασμένη αρχιτεκτονική. Εάν το σφάλμα προέρχεται από την αρχιτεκτονική, θα πρέπει να ελεγχθούν οι προδιαγραφές αρχιτεκτονικής.

Μόλις όλες οι λειτουργικές μονάδες δοκιμαστούν ανεξάρτητα, ο σχεδιαστής προχωρά στη σύνδεση των διαφορετικών μονάδων μεταξύ τους. Είναι βολικό να εφαρμοστεί μια πλήρης προσομοίωση του σχεδιασμού και να διορθωθούν πιθανά σφάλματα ή να τροποποιηθούν τμήματα της αρχιτεκτονικής.

Στη συνέχεια, οι λειτουργικές μονάδες μπορούν να εισαχθούν στο System Generator για την υλοποίηση προσομοίωσης υψηλότερου επιπέδου. Χάρη στις υψηλού επιπέδου λειτουργίες του Matlab (όπως εικόνα ανάγνωση / γραφή, γραφική παράσταση, δημιουργία τυχαίων αριθμών, κ.λπ.) η διαδικασία επαλήθευσης είναι ευκολότερη. Επιπλέον, το System Generator επιτρέπει την εγκατάσταση επιπλέον Xilinx IPCores. Στο System Generator, οι λειτουργικές μονάδες HDL εισάγονται ως Black Boxes και μπορούν να συγκριθούν με ένα μοντέλο Simulink, το οποίο επιτρέπει την εκτίμηση της απώλειας ακριβείας του χειριστή, που είναι εγγενής στην πληκτρολόγηση στοιχείων και στη διαστασιολόγηση. Για άλλη μια φορά, εάν το σύστημα έχει σφάλματα, πρέπει να προσδιοριστούν εάν τα σφάλματα προέρχονται από τη σχεδιασμένη αρχιτεκτονική, πράγμα που θα σήμαινε επαναπροσδιορισμό ορισμένων λειτουργικών μονάδων.

Μια πολύ σημαντική παράμετρος σχεδιασμού στο FPGA είναι η δημιουργία περιορισμών του συστήματος. Σε αυτό το βήμα, πρέπει να προστεθούν οι απαραίτητοι περιορισμοί (χρονικές περίοδοι, pin out, pin standards, offsets, multi-cycle paths ...), δίχως ομως να περιπλακεί η εργασία υλοποίησης.

Η εργασία υλοποίησης, αποτελείται από τρία βήματα: Μετάφραση, Χάρτης και "Place Y Route". Η κατανόηση αυτών των φάσεων επιτρέπει την αποτελεσματική βελτιστοποίηση του σχεδιασμού. Κάθε μία από τις φάσεις παρέχει σημαντικές πληροφορίες που μπορούν να χρησιμοποιηθούν για το κλείσιμο των απαιτήσεων χρονισμού.

Η πρώτη φάση της διαδικασίας εφαρμογής είναι το βήμα Translate. Εδώ, οι δημιουργημένες Netlists από τη σύνθεση, συγχωνεύονται σε μία μόνο Netlist. Η δεύτερη φάση αντιστοιχεί στον χάρτη του σχεδιασμού. Σε αυτήν τη φάση, τα λογικά σύμβολα από την Netlist χαρτογραφούνται σε φυσικά στοιχεία (slices, DSP48, κ.λπ.). Στην τελευταία φάση, Place και Route, τα components τοποθετούνται στο τσιπ.

Μετά την υλοποίηση του σχεδιασμού, οι αναφορές χρονισμού πρέπει να αναλυθούν για να επαληθευτεί εάν η συγκεκριμένη αρχιτεκτονική ικανοποιεί τους περιορισμούς χρονισμού του συστήματος. Εάν δεν ικανοποιηθούν οι περιορισμοί, πρέπει να ακολουθηθεί μια στρατηγική timing closure.

Χρησιμοποιώντας ένα λογισμικό που ονομάζεται PlanAhead της Xilinx, μπορούν να αναγνωριστούν τα κρίσιμα nets που δεν ικανοποιούν τους χρονικούς περιορισμούς στο FPGA. Με αυτόν τον τρόπο, μπορούν να συμπεριληφθούν περιορισμοί περιοχής ή χρονικοί περιορισμοί ή μπορεί να γίνει ειδική βελτιστοποίηση σε ένα συγκεκριμένο net.

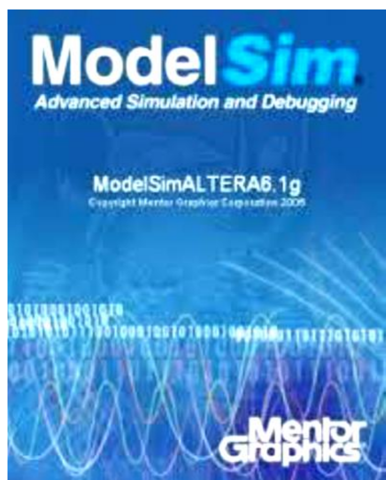
Εάν οι προδιαγραφές χρονισμών πληρούνται, ο σχεδιασμός μπορεί να χρησιμοποιηθεί είτε για τον προγραμματισμό του FPGA είτε για την ανάπτυξη του λογισμικού για τον ενσωματωμένο επεξεργαστή, εάν χρησιμοποιείται. Για τον προγραμματισμό FPGA, δημιουργείται ένα Bitstream αρχείο που περιέχει τις πληροφορίες προγραμματισμού. Όταν χρησιμοποιείται ενσωματωμένος επεξεργαστής, ο σχεδιασμός εισάγεται στο SDK. Στο SDK, δημιουργείται ένα αρχείο που περιέχει το χάρτη μνήμης όλων των περιφερειακών που υπάρχουν στο δίαυλο.

Οι πληροφορίες προγραμματισμού υλικού FPGA (Bitstream) και το λογισμικό επεξεργαστή (ELF) μπορούν να συγχωνευτούν σε ένα μόνο αρχείο που μπορεί να χρησιμοποιηθεί για τον προγραμματισμό του FPGA από ένα compact flash για αυτόνομη λειτουργία. Αυτό το αρχείο περιέχει όλες τις απαραίτητες πληροφορίες προγραμματισμού για τη διαμόρφωση του FPGA.

Ένα πολύ ισχυρό εργαλείο που χρησιμοποιείται κατά την επαλήθευση του σχεδιασμού FPGA είναι το scope viewer, το οποίο επιτρέπει την θέαση των εσωτερικών σημάτων του συστήματος. Το κύριο πλεονέκτημα αυτού του εργαλείου είναι ότι το πραγματικό σύστημα δοκιμάζεται όπως ένας φυσικός ψηφιοποιητής. Μπορούν να χρησιμοποιηθούν πολλές επιλογές για την ενεργοποίηση συμβάντων για τη διευκόλυνση του εντοπισμού σφαλμάτων. Μόλις επικυρωθεί ο σχεδιασμός, η τελική εφαρμογή μπορεί να μεταφερθεί σε μικρότερη συσκευή, επειδή οι πόροι που απαιτούνται για το το scope viewer δεν θα ήταν πλέον απαραίτητοι.

3.6.3 Προσομοίωση συστήματος

Η προσομοίωση αποτελεί ένα πολύ σημαντικό στάδιο στο σχεδιασμό του συστήματος. Για FPGAs, η προσομοίωση γίνεται συνήθως με λογισμικό τρίτου μέρους, όπως π.χ. το ModelSim από τη Menthor Graphics.



Οι Xilinx και Altera ενσωματώνουν έναν προσομοιωτή στο περιβάλλον ανάπτυξής τους. Η χρήση του ενσωματωμένου προσομοιωτή αντί του ModelSim διευκολύνει τις εργασίες, επειδή όλες οι βιβλιοθήκες έχουν ήδη συνδεθεί με τα στοιχεία του FPGA.
Για προσομοίωση, ένα test-bench αρχείο αναπτύσσεται σε HDL για τη δημιουργία του σήματος διέγερσης της Device Under Test (DUT).

Κατά το σχεδιασμό ενός προσαρμοσμένου περιφερειακού PLB, μια προσομοίωση Bus Functional Model (BFM) μπορεί να είναι χρήσιμη. Αυτό το μοντέλο επιτρέπει την προσομοίωση του διαύλου PLB, το οποίο είναι πολύ χρήσιμο για τον πρώιμο εντοπισμό σφαλμάτων. Το κύριο πλεονέκτημα αυτής της προσομοίωσης είναι ότι είναι ταχύτερη από την προσομοίωση μετά τη σύνθεση. Το BFM είναι ένα εργαλείο που δημιουργήθηκε από την Xilinx.

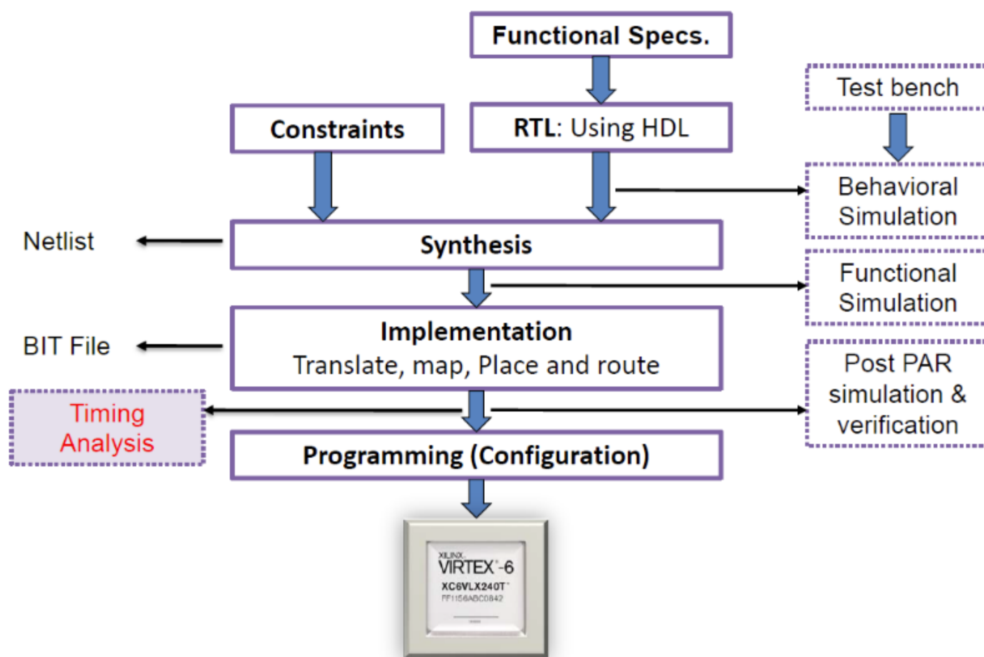
Διατίθεται επίσης ένα άλλο είδος προσομοίωσης που περιλαμβάνει το Hardware in the Loop. Η Co-προσομοίωση γίνεται με το πραγματικό υλικό για την προσομοίωση των αλγορίθμων. Κατά την συν-προσομοίωση, ο χρόνος εκτέλεσης μειώνεται δραστικά επιτρέποντας την εφαρμογή περισσότερων εξαντλητικών test-benches.

Το πακέτο VHDL STD LOGIC TEXTIO επιτρέπει την ανάγνωση και τη σύνταξη αρχείων. Με αυτόν τον τρόπο ο χρήστης μπορεί να διαβάσει ένα αρχείο εισόδου για να τροφοδοτήσει το σχέδιο με σήματα διέγερσης που περιέχονται σε ένα αρχείο. Για την επαλήθευση της σωστής λειτουργίας του σχεδιασμού, οι έξοδοι μπορούν να γραφτούν σε άλλο αρχείο, το οποίο μπορεί να επαληθευτεί χρησιμοποιώντας ένα script.

4 Γλώσσα περιγραφής υλικού VHDL

4.1 Εισαγωγή

Ο σκοπός των γλωσσών περιγραφής υλικού είναι να περιγράψουν τα ψηφιακά κυκλώματα χρησιμοποιώντας μια γλώσσα που βασίζεται σε κείμενο. Οι HDL αποτελούν ένα μέσο για την περιγραφή μεγάλων ψηφιακών συστημάτων χωρίς την ανάγκη σχημάτων, τα οποία μπορούν να γίνουν ανέφικτα σε πολύ μεγάλα σχέδια. Οι HDL έχουν εξελιχθεί για να υποστηρίξουν την προσομοίωση λογικής σε διαφορετικά επίπεδα αφαίρεσης. Αυτό παρέχει τη δυνατότητα του σχεδιασμού και επαλήθευσης της λειτουργικότητας μεγάλων συστημάτων σε υψηλό επίπεδο αφαίρεσης και την αναβολή των λεπτομερειών της εφαρμογής του κυκλώματος αργότερα στον κύκλο σχεδιασμού. Αυτό επιτρέπει μια προσέγγιση σχεδιασμού από πάνω προς τα κάτω που είναι επεκτάσιμη σε διαφορετικές οικογένειες λογικής. Οι HDL έχουν επίσης εξελιχθεί για να υποστηρίζουν την αυτοματοποιημένη σύνθεση, η οποία επιτρέπει στα εργαλεία CAD να λαμβάνουν μια λειτουργική περιγραφή ενός συστήματος (π.χ. έναν πίνακα αλήθειας) και να δημιουργούν αυτόματα το κύκλωμα επιπέδου πύλης που θα εφαρμοστεί σε πραγματικό υλικό (βλ. Σχήμα 29).



Σχήμα 29 – Διάγραμμα ροής πληροφορίας σε FPGA
από https://www.chegg.com/homework-help/questions-and-answers/3-explain-fpga-design-flow-drawn-2-marks-functional-specs-test-bench-constraints-rtl-using-q41430977

Υπάρχουν δύο κυρίαρχες γλώσσες περιγραφής υλικού, οι VHDL και Verilog. Η VHDL σημαίνει γλώσσα περιγραφής υλικού υψηλής ταχύτητας ολοκληρωμένου κυκλώματος. Η Verilog είναι εμπορικό όνομα. Η χρήση αυτών των δύο HDL συναντάται σχεδόν εξίσου στον κλάδο της ψηφιακής σχεδίασης.

Η VHDL είναι πιο αυστηρή στην σύνταξη από την Verilog, και παρέχει καλύτερη περιγραφή των κυκλωμάτων.

Η διαδικασία μετάφρασης ενός κώδικα περιγραφής hardware (HDL language), σε κυκλώματα λογικής και διασυνδέσεις στο FPGA, είναι αντικείμενο εκτενούς μελέτης [31, 32, 33, 35]. Τα βήματα που ακολουθούνται είναι τα εξής [31, 35]:

- Logic Synthesis: Η διαδικασία μετατροπής της HDL, μέσω του Synthesizer, σε πύλες OR, XOR, NAND κ.λ.π. και flip-flop, καθώς και τις αντίστοιχες διασυνδέσεις μεταξύ των πυλών.

- Technology Mapping: Η διαδικασία μετατροπής των συμβατικών πυλών στη φυσική μορφή των πυλών, όπως συναντώνται στο FPGA.

- Packing: Η διαδικασία ομαδοποίησης των LUT και flip-flops, που συνδέονται άμεσα μεταξύ τους, στα CLB. Η τελική μορφή είναι CLBs και I/O blocks.

- Placement: Η τοποθέτηση των CLBs και I/O blocks στο FPGA.

- Routing: Ο προγραμματισμός των SB/CB (Switch Boxes/Connection Blocks) για την ορθή διασύνδεση των CLB, ώστε να εκτελούν συλλογικούς υπολογισμούς λογικών συναρτήσεων, μέσω του Router.

- Bitstream Generation: Η ροή σειριακών ψηφιακών σημάτων (bitstream) για τον προγραμματισμό των μνημών SRAM, ώστε να ενεργοποιηθούν τα αντίστοιχα LUT, hard blocks, και οι διασυνδέσεις μεταξύ τους.

Τα FPGA στην σημερινή εποχή είναι κυρίως SRAM-Based [31, 33, 35]. Κάθε LUT προγραμματίζεται από αντίστοιχες μνήμες SRAM. Τοιουτοτρόπως, η ενεργοποίηση - μικροαλλαγές στις λειτουργίες των hard blocks, όπως και οι διασυνδέσεις με τη σειρά τους, ελέγχονται από μνήμες SRAM. Όταν η διαδικασία του Routing ενεργοποιήσει κάποια διασύνδεση, τότε η αντίστοιχη SRAM στο αντίστοιχο SB, θα πάρει την τιμή 1 για να κάνει τη σύνδεση.

4.1.1 Ιστορία Γλωσσών Περιγραφής Υλικού

Η εφεύρεση του ολοκληρωμένου κυκλώματος πιστώνεται συνήθως σε δύο άτομα, που οδήγησαν σε διπλώματα ευρεσιτεχνίας σε διαφορετικές παραλλαγές της ίδιας βασικής έννοιας το 1959. Ο Jack Kilby ηγήθηκε του πρώτου διπλώματος ευρεσιτεχνίας στο ολοκληρωμένο κύκλωμα τον Φεβρουάριο του 1959 με τίτλο «Miniaturized Electronic Circuits». Ο Robert Noyce ήταν ο δεύτερος που κατέληξε σε δίπλωμα ευρεσιτεχνίας στο ολοκληρωμένο κύκλωμα τον Ιούλιο του 1959 με τίτλο «Semiconductor Device and Lead Structure». Ο Kilby κέρδισε το βραβείο Νόμπελ Φυσικής το 2000 για την εφεύρεσή του, ενώ ο Noyce συνίδρυσε την Intel Corporation το 1968 με τον Gordon Moore. Το 1971, η Intel παρουσίασε τον πρώτο μικροεπεξεργαστή ενός τσιπ χρησιμοποιώντας τεχνολογία ολοκληρωμένου κυκλώματος, το Intel 4004. Αυτό το IC μικροεπεξεργαστή περιείχε 2300 τρανζίστορ. Αυτή η σειρά εφευρέσεων ξεκίνησε τη βιομηχανία ημιαγωγών, η οποία ήταν η κινητήρια δύναμη πίσω από την ανάπτυξη της Silicon Valley, και οδήγησε σε 40 χρόνια πρωτοφανούς προόδου στην τεχνολογία, που επηρέασε κάθε πτυχή του σύγχρονου κόσμου.

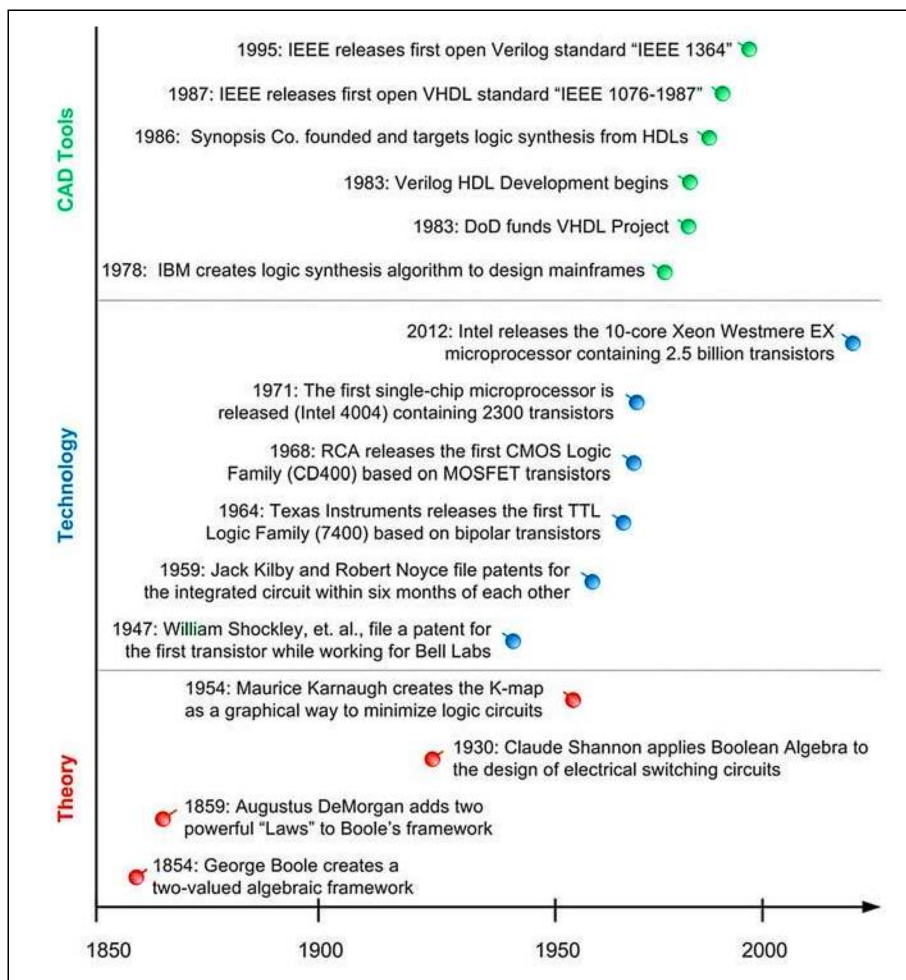
Ο Gordon Moore, συνιδρυτής της Intel, προέβλεψε το 1965 ότι ο αριθμός των τρανζίστορ σε ένα ολοκληρωμένο κύκλωμα θα διπλασιαζόταν κάθε 2 χρόνια. Αυτή η πρόβλεψη, γνωστή τώρα ως Νόμος του Moore, ισχύει από τότε έως την εφεύρεση του ολοκληρωμένου κυκλώματος. Όταν εφευρέθηκε ο πρώτος μικροεπεξεργαστής το 1971, η ικανότητα των εργαλείων CAD αυξήθηκε γρήγορα επιτρέποντας την επίτευξη μεγαλύτερων σχεδίων. Αυτά τα μεγαλύτερα σχέδια, συμπεριλαμβανομένων νεότερων μικροεπεξεργαστών, επέτρεψαν στα εργαλεία CAD να γίνουν ακόμη πιο εξελιγμένα και, με τη σειρά τους, να αποδώσουν ακόμη μεγαλύτερα σχέδια. Η ταχεία επέκταση των ηλεκτρονικών συστημάτων που βασίζονται σε ψηφιακά ολοκληρωμένα κυκλώματα απαιτούσε από διαφορετικούς κατασκευαστές να παράγουν σχέδια που ήταν συμβατά μεταξύ τους. Η υιοθέτηση λογικών οικογενειακών προτύπων βοήθησε τους κατασκευαστές να διασφαλίσουν ότι τα ανταλλακτικά τους θα ήταν συμβατά με άλλους κατασκευαστές στο φυσικό στρώμα (π.χ. τάση και ρεύμα). Ωστόσο, μια πρόκληση που αντιμετώπισε ο κλάδος ήταν ο τρόπος τεκμηρίωσης της περίπλοκης συμπεριφοράς των μεγαλύτερων συστημάτων. Η χρήση σχημάτων για την τεκμηρίωση μεγάλων ψηφιακών σχεδίων έγινε πολύ δυσκίνητη και δύσκολη για να κατανοήσει κανείς. Οι περιγραφές λέξεων για τη συμπεριφορά ήταν πιο κατανοητές, αλλά ακόμη και αυτή η μορφή τεκμηρίωσης έγινε πολύ ογκώδης για να είναι αποτελεσματική για το μέγεθος των σχεδίων που αναδύονταν.

Το 1983, το Υπουργείο Άμυνας (Department of Defence) των ΗΠΑ (DoD) χρηματοδότησε ένα πρόγραμμα για τη δημιουργία ενός μέσου για την καταγραφή - documentation της συμπεριφοράς των ψηφιακών συστημάτων που θα μπορούσαν να χρησιμοποιηθούν σε όλους τους προμηθευτές του. Το DoD σύναψε συμβάσεις με τρεις εταιρείες (Texas Instruments, IBM και Intermetrics), ώστε να αναπτύξουν ένα τυποποιημένο εργαλείο για documentation, που παρείχε λεπτομερείς πληροφορίες τόσο για τη διεπαφή (δηλαδή, εισόδους και εξόδους) όσο και για τη συμπεριφορά των ψηφιακών συστημάτων. Το νέο εργαλείο επρόκειτο να εφαρμοστεί σε μορφή παρόμοια με μια γλώσσα προγραμματισμού. Η ικανότητα προσομοίωσης ήταν επιθυμητή να καλύπτει πολλαπλά επίπεδα αφαίρεσης για να παρέχει τη μέγιστη ευελιξία. Το 1985 κυκλοφόρησε η πρώτη έκδοση αυτού του εργαλείου, που ονομάζεται VHDL. Προκειμένου να επιτευχθεί ευρεία υιοθέτηση και να διασφαλιστεί η συνοχή της χρήσης σε ολόκληρο τον κλάδο, η VHDL παραδόθηκε στο Ινστιτούτο Ηλεκτρολόγων και Ηλεκτρονικών Μηχανικών (IEEE) για προτυποποίηση. Το IEEE είναι μια επαγγελματική ένωση που καθορίζει ένα ευρύ φάσμα προτύπων ανοικτής τεχνολογίας. Το 1987, η IEEE κυκλοφόρησε την πρώτη πρότυπη έκδοση της VHDL. Η κυκλοφορία είχε τον τίτλο IEEE 1076-1987. Το feedback από την αρχική έκδοση οδήγησε σε μια σημαντική αναθεώρηση του προτύπου το 1993 με τίτλο IEEE 1076-1993. Ενώ έχουν γίνει πολλές μικρές αναθεωρήσεις στην κυκλοφορία του 1993, το πρότυπο 1076-1993 περιέχει τη συντριπτική πλειονότητα των λειτουργιών VHDL που χρησιμοποιούνται σήμερα. Το πιο πρόσφατο πρότυπο VHDL είναι το IEEE 1076-2008.

Επίσης το 1983, η Verilog HDL αναπτύχθηκε από την Automated Integrated Design Systems ως γλώσσα λογικής προσομοίωσης. Τα αυτοματοποιημένα ολοκληρωμένα συστήματα σχεδίασης (μετονομάστηκαν σε Gateway Design Automation το 1985) αποκτήθηκαν από τον προμηθευτή εργαλείων CAD Cadence Design Systems το 1990. Σε απάντηση στην ταχεία υιοθέτηση του ανοιχτού προτύπου VHDL, η Cadence έκανε την Verilog HDL ανοιχτή στο κοινό για να παραμείνει ανταγωνιστική. Το IEEE ανέπτυξε για άλλη μια φορά το ανοιχτό πρότυπο για αυτή την HDL και το 1995 κυκλοφόρησε το πρότυπο Verilog με τίτλο IEEE 1364.

Η ανάπτυξη εργαλείων CAD για την επίτευξη αυτοματοποιημένης λογικής σύνθεσης χρονολογείται από τη δεκαετία του 1970, όταν η IBM άρχισε να αναπτύσσει μια σειρά πρακτικών μηχανών σύνθεσης που χρησιμοποιήθηκε στο σχεδιασμό των υπολογιστών τους. Ωστόσο, η κύρια πρόοδος στη λογική σύνθεση ήρθε με την ίδρυση μιας εταιρείας που ονομάζεται Synopsis το 1986. Η Synopsis ήταν η πρώτη εταιρεία που εστίαζε στη λογική σύνθεση απευθείας από HDLs. Λόγω της πολυπλοκότητας της σύνθεσης που παρουσίαζαν οι λειτουργικές μονάδες, μόνο χαμηλότερα επίπεδα σχεδίασης που επεξεργάστηκαν διεξοδικά μπορούσαν αρχικά να συντεθούν. Με το πέρασμα των χρόνων, κατέστη δυνατή η σύνθεση υψηλότερων επιπέδων, αλλά ακόμη και σήμερα δεν μπορούν να συντεθούν όλες οι λειτουργίες που μπορούν να περιγραφούν σε HDL.

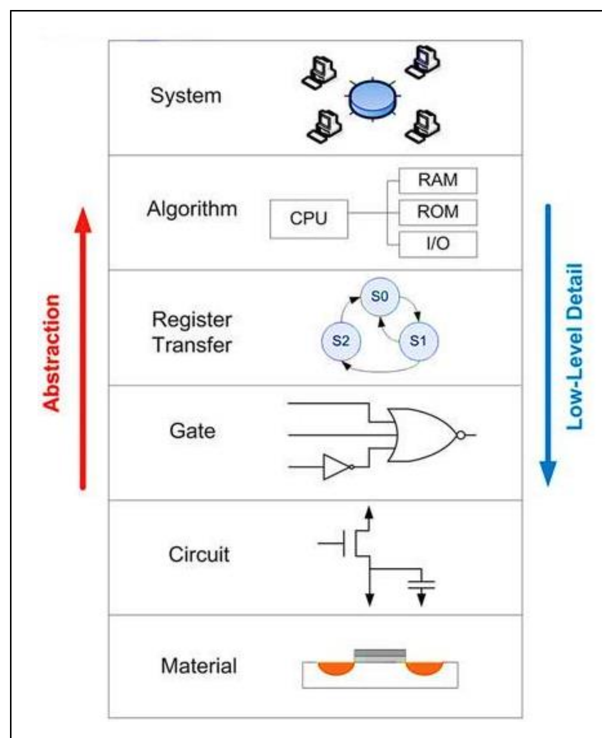
Το Σχήμα 30 δείχνει ένα χρονοδιάγραμμα ορισμένων από τα σημαντικότερα ορόσημα τεχνολογίας στον τομέα της ψηφιακής λογικής και των HDL.



Σχήμα 30 – Σημαντικά ορόσημα στην προώθηση της ψηφιακής λογικής και των HDLs

4.1.2 Επίπεδα Σχεδίασης

Αρχικά ορίστηκαν οι HDL ώστε να μπορούν να μοντελοποιούν τη συμπεριφορά σε πολλαπλά επίπεδα σχεδίασης. Ο αφαιρετικός τρόπος σχεδίασης είναι μια σημαντική ιδέα στον σχεδιασμό, επειδή μας επιτρέπει να καθορίσουμε τον τρόπο λειτουργίας των συστημάτων χωρίς να αναλωνόμαστε στις λεπτομέρειες εφαρμογής. Επίσης, αφαιρώντας τις λεπτομέρειες της εφαρμογής χαμηλότερου επιπέδου, οι προσομοιώσεις μπορούν να διεξαχθούν σε εύλογα χρονικά διαστήματα για τη μοντελοποίηση της λειτουργικότητας υψηλότερου επιπέδου. Το Σχήμα 31 δείχνει μια γραφική απεικόνιση των διαφόρων επιπέδων αφαιρετικής σχεδίασης ενός ψηφιακού συστήματος.

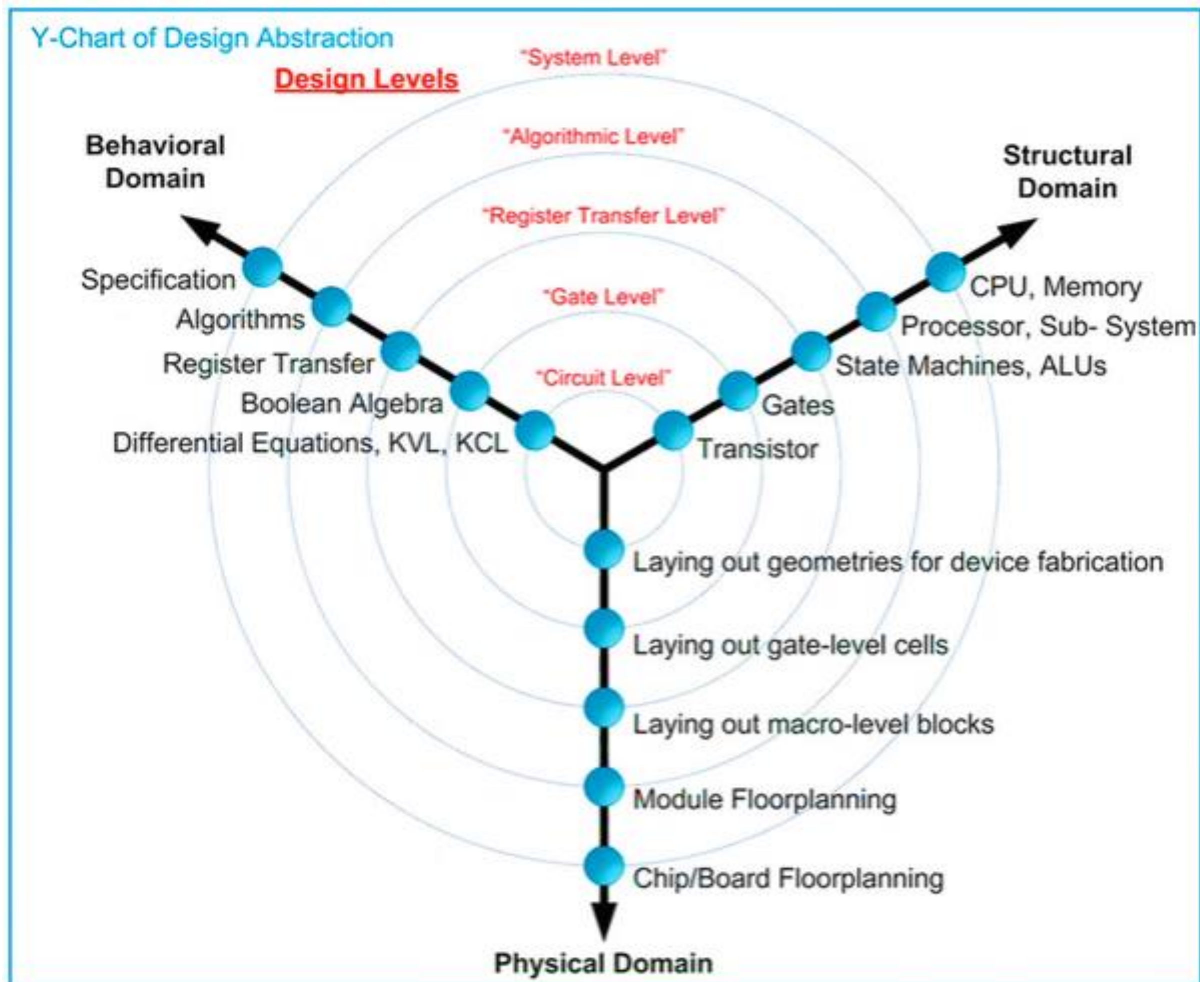


Σχήμα 31 – Επίπεδα αφαιρετικής σχεδίασης
από https://www.quora.com/
What-are-the-various-stages-of-a-modern-digital-design-flow-using-Gajski-and-Kuhn-s-Y-chart

Το υψηλότερο επίπεδο σχεδίασης είναι το επίπεδο συστήματος, στο οποίο η συμπεριφορά ενός συστήματος περιγράφεται δηλώνοντας ένα σύνολο προδιαγραφών. Αυτές οι προδιαγραφές δεν υπαγορεύουν τις λεπτομέρειες χαμηλότερου επιπέδου, όπως τον τύπο της λογικής οικογένειας ή της αρχιτεκτονικής του υπολογιστή. Ένα επίπεδο κάτω από το επίπεδο του συστήματος είναι το αλγοριθμικό επίπεδο. Σε αυτό το επίπεδο, οι προδιαγραφές αρχίζουν να κατανέμονται σε υποσυστήματα, το καθένα με μια σχετική συμπεριφορά που θα ολοκληρώσει ένα μέρος της κύριας εργασίας. Ένα επίπεδο κάτω από το αλγοριθμικό επίπεδο είναι το επίπεδο register-transfer level (RTL), όπου περιγράφονται οι λεπτομέρειες σχετικά με τον τρόπο μετακίνησης των δεδομένων μεταξύ και εντός υποσυστημάτων, καθώς και τον τρόπο χειρισμού των δεδομένων με βάση τις εισόδους του συστήματος. Ένα επίπεδο κάτω είναι το επίπεδο πύλης, όπου ο σχεδιασμός περιγράφεται χρησιμοποιώντας βασικές πύλες και καταχωρητές (ή στοιχεία αποθήκευσης). Πιο κάτω είναι το επίπεδο κυκλώματος που περιγράφει τη λειτουργία των βασικών πυλών και των καταχωρητών χρησιμοποιώντας τρανζίστορ, αγωγοί και άλλα ηλεκτρικά εξαρτήματα όπως αντιστάσεις και πυκνωτές. Τέλος, το χαμηλότερο επίπεδο σχεδίασης είναι το επίπεδο υλικού. Αυτό το επίπεδο περιγράφει τον τρόπο συνδυασμού και διαμόρφωσης διαφορετικών υλικών προκειμένου να υλοποιηθούν τα τρανζίστορ, οι συσκευές και τα καλώδια από το επίπεδο του κυκλώματος.

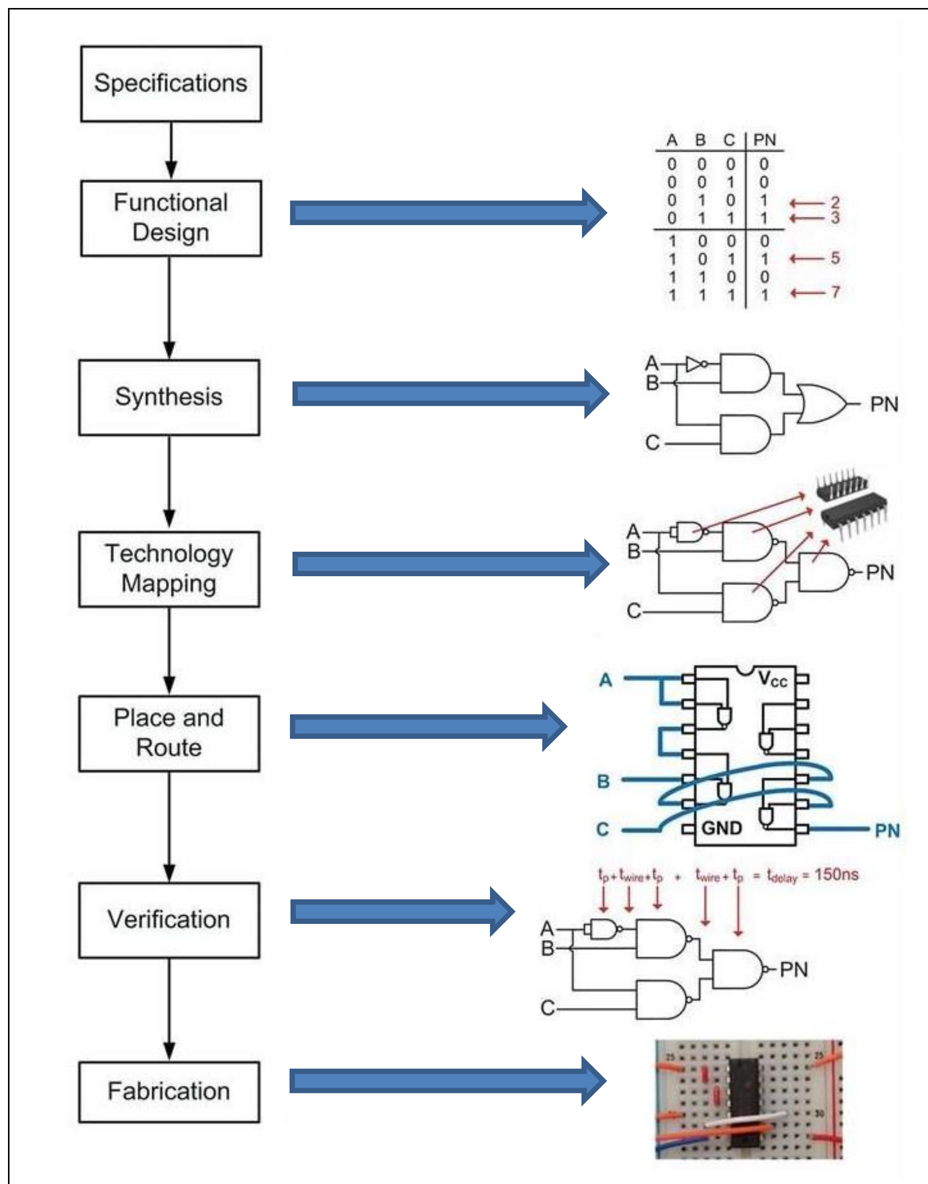
4.1.3 Η ροή σύγχρονου ψηφιακού σχεδιασμού

Κατά την εκτέλεση μικρότερου σχεδιασμού ή του σχεδιασμού υποσυστημάτων, η διαδικασία μπορεί να χωριστεί σε μεμονωμένα βήματα. Αυτά τα βήματα φαίνονται στο Σχήμα 32 και ισχύουν τόσο για τον κλασικό όσο και για τον σύγχρονο ψηφιακό σχεδιασμό. Η διάκριση μεταξύ κλασικού και σύγχρονου είναι ότι ο σύγχρονος ψηφιακός σχεδιασμός χρησιμοποιεί HDLs και αυτοματοποιημένα εργαλεία CAD για προσομοίωση, σύνθεση, τοποθέτηση & δρομολόγηση συνδέσεων και επαλήθευση.



Σχήμα 32 – Ροή ψηφιακής σχεδίασης
από https://www.quora.com/
What-are-the-various-stages-of-a-modern-digital-design-flow-using-Gajski-and-Kuhn-s-Y-chart

Ο σύγχρονος ψηφιακός σχεδιασμός επιτρέπει επιπρόσθετη επαλήθευση σε κάθε βήμα χρησιμοποιώντας αυτοματοποιημένα εργαλεία CAD. Το Σχήμα 33 δείχνει πώς χρησιμοποιείται αυτή η ροή στην προσέγγιση κλασικού σχεδιασμού ενός συνδυαστικού λογικού κυκλώματος.



Σχήμα 33 – Κλασική ροή ψηφιακής σχεδίασης
από https://www.quora.com/
What-are-the-various-stages-of-a-modern-digital-design-flow-using-Gajski-and-Kuhn-s-Y-chart

Ο σύγχρονος σχεδιασμός που βασίζεται σε HDL περιλαμβάνει τη δυνατότητα προσομοίωσης της λειτουργικότητας σε κάθε βήμα της διαδικασίας. Λειτουργικές προσομοιώσεις μπορούν να πραγματοποιηθούν στην αρχική behavioral περιγραφή του συστήματος. Σε κάθε βήμα της διαδικασίας σχεδιασμού, η λειτουργικότητα περιγράφεται με περισσότερες λεπτομέρειες, και τελικά κινείται προς το στάδιο κατασκευής. Σε κάθε επίπεδο, οι λεπτομερείς πληροφορίες μπορούν να συμπεριληφθούν στην προσομοίωση για να επιβεβαιωθεί ότι η λειτουργικότητα εξακολουθεί να είναι σωστή και ότι ο σχεδιασμός εξακολουθεί να πληροί τις αρχικές προδιαγραφές.

4.2 Δομή VHDL

Στην παράγραφο αυτή εξετάζεται η βασική δομή ενός μοντέλου VHDL, καλύπτοντας τα ενσωματωμένα χαρακτηριστικά του, συμπεριλαμβανομένης της δομής του αρχείου, των τύπων δεδομένων, των τελεστών και των δηλώσεων. Κάθε εκχώρηση, ορισμός ή δήλωση VHDL τερματίζεται με ερωτηματικό (;). Επιτρέπεται η αναδίπλωση γραμμών χωρίς να σημαίνει το τέλος μιας ανάθεσης, ορισμού ή δήλωσης. Τα σχόλια στην VHDL προηγούνται με δύο παύλες (δηλαδή, --) και συνεχίζονται μέχρι το τέλος της γραμμής. Όλα τα ονόματα που ορίζονται από το χρήστη στην VHDL πρέπει να ξεκινούν με ένα αλφαβητικό γράμμα και όχι έναν αριθμό. Τα ονόματα που καθορίζονται από τον χρήστη δεν επιτρέπεται να είναι ίδια με οποιαδήποτε "λέξη-κλειδί" VHDL (βλέπε Πίνακα 5).

Πίνακας 5 – Σημειώσεις VHDL

bold	= VHDL λέξη-κλειδί
italics	= Ονόματα που δίδονται από τον χρήστη
<>	= Απαιτούμενο χαρακτηριστικό όπως data type, input/output, κ.λ.π.

4.2.1 Τύποι δεδομένων

Στην VHDL, σε κάθε σήμα, σταθερά, μεταβλητή και συνάρτηση πρέπει να εκχωρηθεί ένας τύπος δεδομένων. Το τυποποιημένο πακέτο IEEE παρέχει μια ποικιλία προκαθορισμένων τύπων δεδομένων. Ορισμένοι τύποι δεδομένων είναι συνθέσιμοι, ενώ άλλοι είναι μόνο για μοντελοποίηση της αφαιρετικής συμπεριφοράς. Οι ακόλουθοι είναι οι πιο συχνά χρησιμοποιούμενοι τύποι δεδομένων στο τυπικό πακέτο VHDL.

4.2.1.1 Αριθμητικοί τύποι

Ένας αριθμητικός τύπος είναι αυτός στον οποίο καθορίζονται οι ακριβείς τιμές που μπορεί να πάρει ο τύπος (Πίνακας 6).

Πίνακας 6 – Αριθμητικοί Τύποι

Τύπος	Δυνατές τιμές
bit	{0, 1}
boolean	{false, true}
character	{"256 ASCII χαρακτήρες ISO 8859-1"}

Οι τύπου bit είναι συνθέσιμοι, ενώ οι Boolean και χαρακτήρα δεν είναι.

4.2.1.2 Τύποι εύρους

Ένας τύπος εύρους είναι αυτός που μπορεί να λάβει οποιαδήποτε τιμή εντός ενός εύρους (βλέπε Πίνακα 7).

Πίνακας 7 – Τύποι Εύρους

Τύπος	Δυνατές τιμές
integer	Αριθμοί μεταξύ -2,147,483,648 έως +2,147,483,647
real	Κλασματικοί αριθμοί μεταξύ $-1.7e^{38}$ έως $+1.7e^{38}$

Ο τύπος integer είναι ακέραιος εύρους 32-bit, προσημασμένος, με τις αρνητικές τιμές να παριστάνονται με τη μορφή συμπληρώματος ως προς δύο. Εάν το πλήρες εύρος των ακέραιων τιμών δεν είναι επιθυμητό, αυτός ο τύπος μπορεί να περιοριστεί συμπεριλαμβάνοντας το εύρος <min> έως <max>. Ο τύπος real είναι 32-bit, floating point και δεν μπορεί να γίνει σύνθεση άμεσα, εκτός εάν περιλαμβάνεται ένα πρόσθετο πακέτο που καθορίζει τη μορφή floating point. Οι τιμές αυτών των τύπων υποδεικνύονται απλώς χρησιμοποιώντας τον αριθμό χωρίς εισαγωγικά (π.χ. 33, 3.14).

4.2.1.3 Φυσικοί τύποι

Ένας φυσικός τύπος είναι αυτός που περιέχει μια τιμή και μονάδες.
Στην VHDL, ο χρόνος είναι ο κύριος φυσικός τύπος που υποστηρίζεται (Πίνακας 8).

Πίνακας 8 – Φυσικοί Τύποι

Τύπος	Δυνατές τιμές	
time	Αριθμοί μεταξύ -2,147,483,648 έως 2,147,483,647	
(μονάδες)	fs (βασική μονάδα)	(femtosecond, 10^{-15})
	ps = 10^3 fs	(picosecond, 10^{-12})
	ns = 10^3 ps	(nanosecond, 10^{-9})
	ns = 10^3 ps	(microsecond, 10^{-6})
	ms = 10^3 μs	(millisecond, 10^{-3})
	s = 1000 ms	(second)
	min = 60s	(minute)
	h = 60min	(hour)

Η βασική μονάδα για το χρόνο είναι fs, που σημαίνει ότι, εάν δεν παρέχονται μονάδες, η τιμή θεωρείται ότι είναι σε femtoseconds. Η τιμή του χρόνου διατηρείται ως 32-bit, signed αριθμός και δεν μπορεί να γίνει σύνθεση.

4.2.1.4 Τύποι διανυσμάτων

Ένας τύπος διανύσματος αποτελείται από μια γραμμική συστοιχία βαθμωτών τύπων (βλέπε Πίνακα 9).

Πίνακας 9 – Τύποι Διανυσμάτων

Τύπος	Δομή
bit_vector	Μια γραμμική συστοιχία τύπου bit
string	Μια γραμμική συστοιχία τύπου χαρακτήρα

Το μέγεθος ενός διανύσματος καθορίζεται με τη συμπερίληψη του μέγιστου αριθμού ευρετηρίου, της λέξης-κλειδιού *downto* και του ελάχιστου αριθμού ευρετηρίου. Για παράδειγμα, εάν δόθηκε ο τύπος bit_vector (7 downto 0), θα δημιουργούσε ένα διάνυσμα με 8 στοιχεία, καθένα από αυτά να είναι τύπου bit. Η αριστερότερη βαθμίδα θα έχει δείκτη 7 και η δεξιότερη βαθμίδα θα έχει δείκτη 0. Κάθε μία από τις μεμονωμένες βαθμίδες εντός του διανύσματος μπορεί να προσεγγιστεί παρέχοντας τον αριθμό του στοιχείου σε παρενθέσεις. Οι δείκτες δεν χρειάζεται πάντα να έχουν ελάχιστη τιμή 0, αλλά αυτή είναι η πιο κοινή προσέγγιση ευρετηρίου στο λογικό σχεδιασμό. Ο τύπος bit_vector είναι συνθέσιμος, ενώ η συμβολοσειρά δεν είναι. Οι τιμές αυτών των τύπων υποδεικνύονται περικλείοντάς τις εντός διπλών εισαγωγικών (π.χ. "0011", "abcd").

4.2.1.5 Απαριθμητοί τύποι

```
type name is (value1, value2, ...);
```

4.2.1.6 Τύπος συστοιχίας

Ένας πίνακας - συστοιχία περιέχει πολλά στοιχεία του ίδιου τύπου. Τα στοιχεία σε έναν πίνακα μπορεί να είναι βαθμωτά ή διανύσματα. Το εύρος του πίνακα πρέπει να καθοριστεί στη δήλωση τύπου πίνακα. Το εύρος καθορίζεται με ακέραιους αριθμούς (ελάχιστο & μέγιστο) και είτε με τις λέξεις-κλειδιά *down to* είτε *to*. Η δημιουργία ενός τύπου πίνακα φαίνεται παρακάτω:

```
type name is array (<range>) of <element_type>;
type ihu_serres is array (0 to 7) bit_vector(15 downto 0);
signal my_array : ihu_serres;
```

Σε αυτό το παράδειγμα, ο νέος τύπος πίνακα δηλώνεται με οκτώ στοιχεία. Ο αρχικός δείκτης του πίνακα είναι 0 και ο τελικός δείκτης είναι 7. Κάθε στοιχείο στον πίνακα είναι ένα διάνυσμα 16-bit τύπου bit_vector.

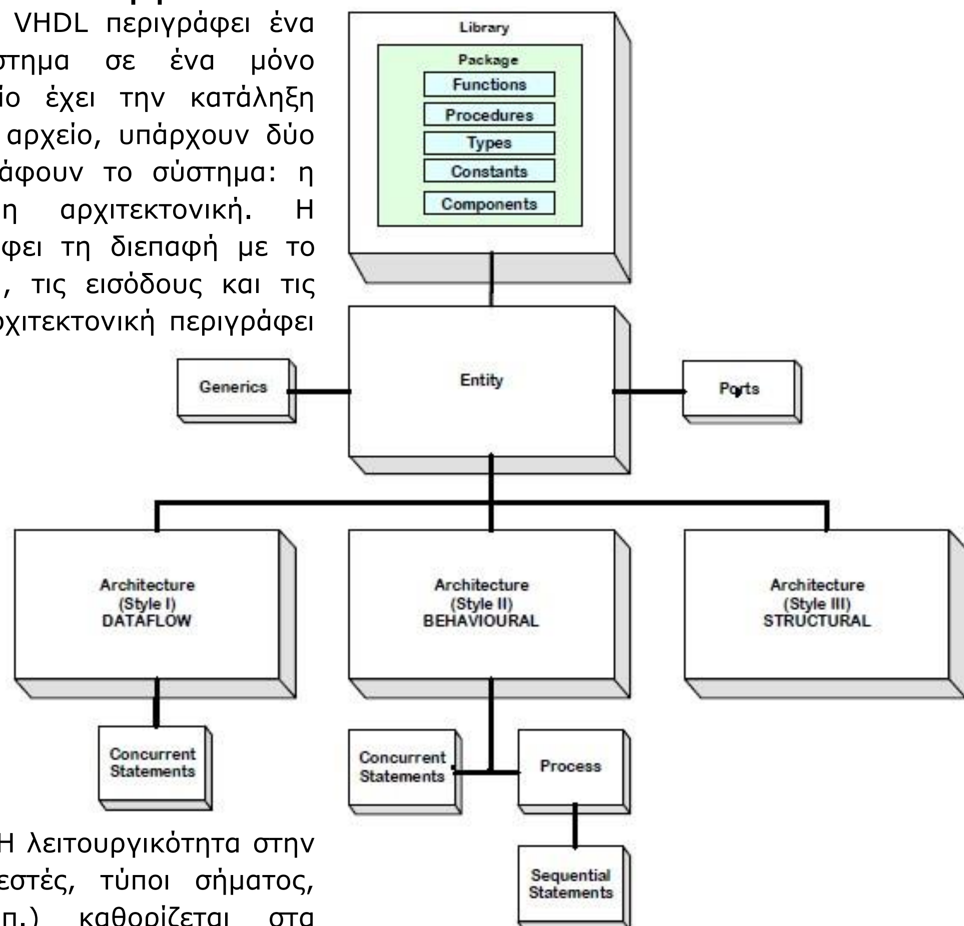
4.2.1.7 Δευτερεύοντες Τύποι

Ένας δευτερεύων τύπος είναι μια περιορισμένη έκδοση ή υποσύνολο άλλου τύπου. Οι δευτερεύοντες τύποι καθορίζονται από τον χρήστη, αν και μερικοί τύποι που χρησιμοποιούνται συνήθως είναι προκαθορισμένοι στο τυπικό πακέτο. Το παρακάτω είναι η σύνταξη για τη δήλωση ενός υποτύπου.

```
subtype name is <type> range <min> to <max>;
```

4.3 Κατασκευή μοντέλου VHDL

Ένα κύκλωμα σε VHDL περιγράφει ένα μεμονωμένο σύστημα σε ένα μόνο αρχείο. Το αρχείο έχει την κατάληξη *.vhd. Μέσα στο αρχείο, υπάρχουν δύο μέρη που περιγράφουν το σύστημα: η οντότητα και η αρχιτεκτονική. Η οντότητα περιγράφει τη διεπαφή με το σύστημα (δηλαδή, τις εισόδους και τις εξόδους) και η αρχιτεκτονική περιγράφει τη συμπεριφορά. Η λειτουργικότητα στην VHDL (π.χ., τελεστές, τύποι σήματος, συναρτήσεις κ.λπ.) καθορίζεται στα πακέτα. Τα πακέτα ομαδοποιούνται σε μια βιβλιοθήκη.



Σχήμα 34 – Ανατομία ενός αρχείου VHDL από http://vlsi-project.blogspot.com/2010/09/vhdl-design-is-composed-of-following.html

Η βιβλιοθήκη IEEE καθορίζει το βασικό σύνολο λειτουργιών για την VHDL. Η συμπερίληψη της βιβλιοθήκης και των πακέτων αναφέρεται στην αρχή ενός αρχείου VHDL πριν από την οντότητα και την αρχιτεκτονική.

Πρόσθετη λειτουργικότητα μπορεί να προστεθεί στην VHDL συμπεριλαμβάνοντας άλλα πακέτα, αλλά όλα τα πακέτα βασίζονται στη βασική λειτουργικότητα που καθορίζεται στα στάνταρ πακέτα. Το Σχήμα 34 δείχνει μια γραφική απεικόνιση ενός αρχείου VHDL.

4.3.1 Βιβλιοθήκες

Όπως αναφέρθηκε προηγουμένως, η βιβλιοθήκη IEEE υπονοείται κατά την χρήση της VHDL. Ωστόσο, μπορούμε να τη χρησιμοποιήσουμε ως παράδειγμα του τρόπου συμπερίληψης πακέτων στην VHDL. Η λέξη-κλειδί *library* χρησιμοποιείται για να δηλώσει ότι τα πακέτα πρόκειται να προστεθούν στο σχέδιο VHDL από την καθορισμένη βιβλιοθήκη. Το όνομα της βιβλιοθήκης ακολουθεί αυτήν τη λέξη-κλειδί. Για την συμπερίληψη ενός συγκεκριμένου πακέτου από τη βιβλιοθήκη, χρησιμοποιείται μια νέα γραμμή με τη χρήση της λέξης-κλειδιού ακολουθούμενη από τις λεπτομέρειες του πακέτου. Η σύνταξη του πακέτου έχει τρία πεδία διαχωρισμένα με τελεία. Ο πρώτος τομέας είναι το όνομα της βιβλιοθήκης. Ο δεύτερος τομέας είναι το όνομα του πακέτου. Ο τρίτος τομέας είναι η συγκεκριμένη λειτουργικότητα του πακέτου που θα συμπεριληφθεί. Αν πρόκειται να χρησιμοποιηθεί όλη η λειτουργικότητα ενός πακέτου, τότε η λέξη-κλειδί *all* χρησιμοποιείται στο τρίτο μέρος:

```
library IEEE;
use IEEE.std_logic_1164.all;
```

4.3.2 Entity

Η οντότητα στην VHDL περιγράφει τις εισόδους και τις εξόδους του συστήματος. Κάθε σήμα πρέπει να έχει όνομα, λειτουργία και τύπο δεδομένων που παριστάνει. Το όνομα καθορίζεται από το χρήστη. Η λειτουργία περιγράφει τα δεδομένα κατεύθυνσης που μεταφέρονται μέσω της θύρας και μπορούν να λάβουν τιμές εισόδου, εξόδου, εισόδου-εξόδου και εξόδου με δυνατότητα ανάγνωσης της τιμής εξόδου (buffer). Τα ονόματα των θυρών με την ίδια λειτουργία και τον ίδιο τύπο μπορούν να αναγράφονται στην ίδια γραμμή διαχωρισμένα με κόμματα. Ο ορισμός μιας οντότητας δίνεται παρακάτω.

```
entity_name is
port
    (port_name : <mode> <type>;
end entity;
```

4.3.3 Architecture

Η αρχιτεκτονική στην VHDL περιγράφει τη συμπεριφορά ενός συστήματος. Η αρχιτεκτονική είναι εκεί όπου πραγματοποιείται το μεγαλύτερο μέρος του σχεδιασμού. Η μορφή μιας γενικής αρχιτεκτονικής δίνεται παρακάτω.

```
architecture architecture_name of <entity> is

-- δηλώσεις αριθμημένων τύπων
-- δηλώσεις σημάτων
-- δηλώσεις σταθεράς
-- δηλώσεις component

begin

-- behavioral περιγραφή συστήματος

end architecture;
```

4.3.3.1 Δηλώσεις σήματος

Ένα σήμα που χρησιμοποιείται για εσωτερικές συνδέσεις μέσα σε ένα σύστημα δηλώνεται στην αρχιτεκτονική. Κάθε σήμα πρέπει να δηλώνεται με έναν τύπο. Το σήμα μπορεί να χρησιμοποιηθεί μόνο για την πραγματοποίηση συνδέσεων παρόμοιων τύπων. Ένα σήμα δηλώνεται με τη λέξη-κλειδί *signal* ακολουθούμενο από ένα καθορισμένο από το χρήστη όνομα, άνω και κάτω τελεία και τον τύπο. Σήματα παρόμοιου τύπου μπορούν να δηλωθούν στην ίδια γραμμή διαχωρισμένα με κόμμα. Τα σήματα δεν μπορούν να έχουν το ίδιο όνομα με μια θύρα στο σύστημα στο οποίο βρίσκονται. Η σύνταξη για μια δήλωση σήματος έχει ως εξής:

```
signal ihu_serres : bit_vector (15 downto 0);
```

Η VHDL υποστηρίζει μια ιεραρχική προσέγγιση σχεδιασμού. Τα ονόματα σημάτων μπορούν να είναι τα ίδια σε ένα υποσύστημα με αυτά σε υψηλότερο επίπεδο χωρίς σύγκρουση.

4.3.3.2 Δηλώσεις σταθεράς

Μια σταθερά είναι χρήσιμη για την αναπαράσταση μιας ποσότητας που θα χρησιμοποιηθεί πολλές φορές στην αρχιτεκτονική. Η σύνταξη για τη δήλωση σταθεράς έχει ως εξής:

```
constant ihu_serres : integer := 32;
```

Μόλις δηλωθεί, η σταθερά μπορεί να χρησιμοποιηθεί σε όλη την αρχιτεκτονική. Το παρακάτω παράδειγμα δείχνει πώς μπορούμε να χρησιμοποιήσουμε μια σταθερά για να προσδιορίσουμε το μέγεθος ενός διανύσματος. Επειδή ορίσαμε τη σταθερά να είναι το πραγματικό πλάτος του διανύσματος (δηλαδή, 32-bits), πρέπει να αφαιρέσουμε ένα από την τιμή του κατά τον καθορισμό των δεικτών (δηλαδή, 31 έως 0).

```
signal IHU : bit_vector (ihu_serres-1 downto 0);
```

4.3.3.3 Δηλώσεις component

Ένα component είναι ο όρος που χρησιμοποιείται για ένα υποσύστημα VHDL, το οποίο δημιουργείται σε ένα σύστημα υψηλότερου επιπέδου. Εάν ένα στοιχείο πρόκειται να χρησιμοποιηθεί σε ένα σύστημα, πρέπει να δηλωθεί στην αρχιτεκτονική πριν από τη δήλωση έναρξης. Η σύνταξη για μια δήλωση component έχει ως εξής:

```
component_name
port
   (port_name : <mode> <type>;
end component;
```

4.4 Μοντελοποίηση ταυτόχρονης λειτουργίας

4.4.1 Τελεστές (Operators) VHDL

Υπάρχει μια ποικιλία προκαθορισμένων τελεστών στα τυπικά πακέτα της IEEE. Οι τελεστές εφαρμόζονται σε συγκεκριμένους τύπους δεδομένων και δεν είναι όλοι συνθέσιμοι. Στην VHDL, οι γραμμές κώδικα αντιπροσωπεύουν τη συμπεριφορά του πραγματικού υλικού. Ως αποτέλεσμα, όλες οι αναθέσεις - εκχωρήσεις σήματος εκτελούνται ταυτόχρονα, εκτός αν αναφέρεται διαφορετικά.

4.4.1.1 Τελεστής ανάθεσης

Η VHDL χρησιμοποιεί τον τελεστή <= για όλες τις αναθέσεις σήματος και := για όλες τις αναθέσεις μεταβλητών και αρχικοποίησης. Αυτοί οι τελεστές ανάθεσης χρησιμοποιούνται σε όλους τους τύπους δεδομένων. Ο στόχος της ανάθεσης πηγαίνει στα αριστερά αυτών των τελεστών και η τιμή ανάθεσης πηγαίνει δεξιά.

```
D1 <= A;        -- D1 και A είναι του ίδιου μεγέθους-τύπου
D2 <= '0';      -- D2 είναι τύπου bit
D3 <= "0000";   -- D3 είναι τύπου bit_vector(3 downto 0)
D4 <= x"1A";    -- D4 είναι τύπου bit_vector(7 downto 0) (x"1A" HEX)_
```

4.4.1.2 Λογικοί τελεστές

Η VHDL περιέχει τους ακόλουθους λογικούς τελεστές (Πίνακας 10):

Πίνακας 10 – Λογικοί τελεστές

Τελεστές
NOT
AND
NAND
OR
NOR
Exclusive-OR
Exclusive-NOR

Αυτοί οι τελεστές λειτουργούν σε τύπους bit, bit_vector και boolean. Για λειτουργίες στον τύπο bit_vector, τα διανύσματα εισόδου πρέπει να έχουν το ίδιο μέγεθος.

Η σειρά προτεραιότητας στην VHDL είναι διαφορετική από την άλγεβρα Boole. Ο τελεστής NOT έχει υψηλότερη προτεραιότητα από όλους τους άλλους τελεστές. Όλοι οι άλλοι λογικοί τελεστές έχουν την ίδια προτεραιότητα.

Αυτό σημαίνει ότι στην VHDL, ο τελεστής AND δεν θα προηγηθεί της λειτουργίας OR όπως και στην άλγεβρα Boolean. Οι παρενθέσεις χρησιμοποιούνται για να περιγράψουν ρητά την προτεραιότητα. Εάν χρησιμοποιούνται τελεστές που έχουν την ίδια προτεραιότητα και δεν παρέχονται παρενθέσεις, τότε οι αναθέσεις θα πραγματοποιηθούν στα σήματα από τα αριστερά προς τα δεξιά.

4.4.1.3 Αριθμητικοί τελεστές

Η VHDL περιέχει τους ακόλουθους αριθμητικούς τελεστές (Πίνακας 11):

Πίνακας 11 – Αριθμητικοί τελεστές

Τελεστές	Λειτουργία
+	Πρόσθεση
-	Αφαίρεση
*	Πολλαπλασιασμός
/	Διαίρεση
mod	Modulus
rem	Υπόλοιπο
abs	Απόλυτη τιμή
**	Εκθετικός

Αυτοί οι τελεστές χρησιμοποιούνται σε τύπους ακέραιου και real. Το προεπιλεγμένο πρότυπο VHDL δεν υποστηρίζει αριθμητικούς τελεστές σε τύπους bit και bit_vector.

4.4.1.4 Τελεστές σύγκρισης

Η VHDL περιέχει τους ακόλουθους συγκριτικούς τελεστές (Πίνακας 12). Αυτοί οι τελεστές συγκρίνουν δύο εισόδους του ίδιου τύπου και επιστρέφουν τον τύπο Boolean (δηλαδή, true ή false).

Πίνακας 12 – Τελεστές σύγκρισης

Τελεστές	Λειτουργία
=	Ίσο
/=	Διάφορο
<	Λιγότερο απο
<=	Λιγότερο από ή ίσο
>	Μεγαλύτερο απο
>=	Μεγαλύτερο απο ή ίσο

4.4.1.5 Τελεστές ολίσθησης

Η VHDL περιέχει τους ακόλουθους τελεστές ολίσθησης. Αυτοί οι τελεστές εφαρμόζονται σε τύπους διανυσμάτων bit_vector και string (βλέπε Πίνακα 13).

Πίνακας 13 – Τελεστές Ολίσθησης

Τελεστές	Λειτουργία
sll	Ολίσθηση αριστερά (λογική)
srl	Ολίσθηση δεξιά (λογική)
sla	Ολίσθηση αριστερά (αριθμητική)
sra	Ολίσθηση δεξιά (αριθμητική)
rol	Ολίσθηση αριστερά
ror	Ολίσθηση δεξιά

Η σύνταξη παρέχει το όνομα του διανύσματος ακολουθούμενο από τον επιθυμητό τελεστή ολίσθησης, ακολουθούμενο από έναν ακέραιο που δείχνει πόσες λειτουργίες ολίσθησης θα εκτελεστούν. Ο στόχος της ανάθεσης πρέπει να είναι του ίδιου τύπου και μεγέθους με την είσοδο.

4.4.1.6 Τελεστές συνένωσης

Στην VHDL το «&» χρησιμοποιείται για τη συνένωση πολλαπλών σημάτων. Το σήμα προορισμός αυτής της λειτουργίας πρέπει να έχει το ίδιο μέγεθος με το σήμα που προκύπτει μετά την ένωση των σημάτων δεξιά του τελεστή.

4.5 Ταυτόχρονη ανάθεση σήματος με λογικούς τελεστές

Οι ταυτόχρονες εκχωρήσεις σήματος επιτυγχάνονται χρησιμοποιώντας απλώς το τελεστή <= μετά τη δήλωση έναρξης στην αρχιτεκτονική. Κάθε μεμονωμένη εκχώρηση θα εκτελείται ταυτόχρονα και θα γίνεται σύνθεση ως ξεχωριστό λογικό κύκλωμα.

```
X <= A;
Y <= B;
Z <= C;
```

Όταν προσομοιωθούν, αυτές οι τρεις γραμμές VHDL θα κάνουν τρεις ξεχωριστές αναθέσεις σήματος την ίδια στιγμή.

4.6 Εκχωρήσεις υπό όρους

Σε μια εκχώρηση σήματος υπό όρους, η λέξη-κλειδί *when* χρησιμοποιείται για να περιγράψει την εκχώρηση σήματος για μια συγκεκριμένη κατάσταση Boolean. Η λέξη-κλειδί *else* χρησιμοποιείται για να περιγράψει τις εκχωρήσεις σήματος για οποιεσδήποτε άλλες συνθήκες. Πολλαπλές Boolean συνθήκες μπορούν να χρησιμοποιηθούν για να περιγράψουν πλήρως την έξοδο του κυκλώματος υπό όλες τις συνθήκες εισόδου. Λογικοί τελεστές μπορούν επίσης να χρησιμοποιηθούν στις συνθήκες Boolean για τη δημιουργία πιο εξελιγμένων συνθηκών. Οι συνθήκες Boolean μπορούν να περιληφθούν σε παρενθέσεις για ευκολία.

Μια σημαντική θεώρηση των εκχωρήσεων υπό όρους σήματος είναι ότι εξακολουθούν να εκτελούνται ταυτόχρονα. Κάθε εκχώρηση αντιπροσωπεύει ένα ξεχωριστό, συνδυαστικό λογικό κύκλωμα.

4.7 Επιλεγμένες αναθέσεις σήματος

Η επιλεγμένη εκχώρηση σήματος αποτελεί μια άλλη τεχνική για την εφαρμογή ταυτόχρονων εκχωρήσεων σήματος. Σε αυτήν την προσέγγιση, η εκχώρηση σήματος βασίζεται σε μια συγκεκριμένη τιμή στο σήμα εισόδου. Η λέξη-κλειδί *with* χρησιμοποιείται για την έναρξη της επιλεγμένης ανάθεσης σήματος. Στη συνέχεια ακολουθεί το όνομα της εισόδου που θα χρησιμοποιηθεί για να υπαγορεύσει την τιμή της εξόδου. Μόνο ένα όνομα μεταβλητής μπορεί να καταχωρηθεί ως είσοδος. Αυτό σημαίνει ότι εάν η εκχώρηση θα βασίζεται σε πολλές μεταβλητές, πρέπει πρώτα να συνενωθούν σε ένα μόνο όνομα φορέα πριν ξεκινήσει η ανάθεση σήματος. Μετά την καταχώριση της εισαγωγής, η λέξη-κλειδί *select* σηματοδοτεί την αρχή των εκχωρήσεων σήματος. Μια ανάθεση γίνεται σε ένα σήμα βασισμένη σε μια λίστα πιθανών τιμών εισαγωγής που ακολουθούν τη λέξη-κλειδί *when*. Μπορούν να χρησιμοποιηθούν πολλαπλές τιμές των κωδικών εισαγωγής και διαχωρίζονται με κόμματα. Η λέξη-κλειδί *others* χρησιμοποιείται για την κάλυψη τιμών εισαγωγής που δεν αναφέρονται ρητά.

4.8 Δομικός Σχεδιασμός και Ιεραρχία

Ο δομικός σχεδιασμός στην VHDL αναφέρεται στην συμπερίληψη υποσυστημάτων χαμηλότερου επιπέδου σε ένα σύστημα υψηλότερου επιπέδου προκειμένου να παραχθεί η επιθυμητή λειτουργικότητα. Ένας καθαρά δομικός σχεδιασμός VHDL δεν θα περιείχε behavioral μοντελοποίηση στην αρχιτεκτονική, όπως εκχώρηση σήματος, αλλά αντίθετα θα περιλάμβανε μόνο την εγκατάσταση και διασυνδέσεις άλλων υποσυστημάτων.

4.8.1 Components

4.8.1.1 Components

Ένα υποσύστημα ονομάζεται component στην VHDL. Για οποιοδήποτε component πρόκειται να χρησιμοποιηθεί σε μια αρχιτεκτονική, πρέπει να δηλωθεί πριν από τη δήλωση begin. Ένα συγκεκριμένο component πρέπει να δηλωθεί μόνο μία φορά. Μετά τη δήλωση begin, μπορεί να χρησιμοποιηθεί όσες φορές χρειάζεται. Κάθε component εκτελείται ταυτόχρονα.

Ο όρος instantiation αναφέρεται στη χρήση ή συμπερίληψη του component στη σχεδίαση σε VHDL. Όταν ένα component αρχικοποιείται, πρέπει να του δοθεί ένα μοναδικό όνομα αναγνώρισης. Το όνομα της ετικέτας δίνεται πρώτο, ακολουθούμενο από άνω και κάτω τελεία και στη συνέχεια το όνομα του component. Το τελευταίο μέρος της δημιουργίας είναι η σύνδεση σημάτων στις θύρες του. Το τμήμα της σύνταξης στο οποίο συνδέονται τα σήματα στις θύρες του component δηλώνεται με τις δεσμευμένες λέξεις port map.

4.8.1.2 Components Instantiation

Υπάρχουν δύο τεχνικές για τη σύνδεση σημάτων στις θύρες του component, explicit port mapping και positional port mapping.

Σε ρητή αντιστοίχιση σημάτων, δίνεται το όνομα κάθε σήματος του component, ακολουθούμενο από την ένδειξη σύνδεσης =>, ακολουθούμενο από το σήμα στο οποίο συνδέεται. Οι αντιστοιχίσεις σημάτων μπορούν να παρατίθενται με οποιαδήποτε σειρά, καθώς οι λεπτομέρειες της σύνδεσης (δηλ. όνομα θύρας έως όνομα σήματος) είναι σαφείς. Κάθε όνομα σύνδεσης χωρίζεται με κόμμα.

Στην αντιστοίχιση σημάτων ανάλογα με τη θέση, τα ονόματα των σημάτων του component δεν αναφέρονται ρητά. Αντίθετα, τα σήματα παρατίθενται με την ίδια σειρά που καθορίστηκαν τα σήματα στο component. Κάθε όνομα σήματος διαχωρίζεται με κόμμα. Αυτή η προσέγγιση απαιτεί λιγότερο κείμενο για να περιγραφεί, αλλά μπορεί επίσης να οδηγήσει σε εσφαλμένες συνδέσεις λόγω αναντιστοιχιών με τη σειρά των σημάτων που συνδέονται.

4.9 Μοντελοποίηση ακολουθιακής λειτουργίας

Παρακάτω περιγράφονται δομές VHDL σε μοντέλα εκχωρήσεων σήματος που ενεργοποιούνται από ένα παλμό ρολογιού για να διαμορφώσουν με ακρίβεια τη διαδοχική - ακολουθιακή λογική.

4.9.1 Η διαδικασία (process)

Η VHDL χρησιμοποιεί μια process για τη μοντελοποίηση αναθέσεων σήματος που γίνονται σε μέτωπα παλμών ρολογιού. Μια διαδικασία είναι μια τεχνική για τη μοντελοποίηση της συμπεριφοράς ενός συστήματος. Έτσι, μια διαδικασία τοποθετείται στην αρχιτεκτονική VHDL μετά τη δήλωση begin. Οι εκχωρήσεις σήματος σε μια διαδικασία έχουν μοναδικά χαρακτηριστικά που τους επιτρέπουν να μοντελοποιήσουν με ακρίβεια την ακολουθιακή λογική. Πρώτον, οι εκχωρήσεις σήματος δεν πραγματοποιούνται έως ότου η διαδικασία τελειώσει ή διακοπεί. Δεύτερον, οι εκχωρήσεις σήματος θα γίνονται μόνο μία φορά, κάθε φορά που ενεργοποιείται η διαδικασία. Τέλος, οι εκχωρήσεις σήματος θα εκτελεστούν με τη σειρά που εμφανίζονται κατά τη διαδικασία. Αυτή η συμπεριφορά ανάθεσης ονομάζεται διαδοχική εκχώρηση σήματος. Οι διαδοχικές εκχωρήσεις σήματος επιτρέπουν σε μια διαδικασία να μοντελοποιήσει τη συμπεριφορά επιπέδου μεταφοράς καταχωρητή όπου ένα σήμα μπορεί να χρησιμοποιηθεί τόσο ως ο τελεστής μιας εκχώρησης όσο και ως προορισμός μιας διαφορετικής εκχώρησης εντός της ίδιας διαδικασίας. Η VHDL παρέχει δύο τεχνικές για την ενεργοποίηση μιας διαδικασίας, τη λίστα ευαισθησίας και τη δήλωση αναμονής.

4.9.1.1 Λίστες ευαισθησίας

Μια λίστα ευαισθησίας είναι ένας μηχανισμός για τον έλεγχο της ενεργοποίησης μιας διαδικασίας (ή της έναρξης). Μια λίστα ευαισθησίας περιέχει μια λίστα σημάτων στα οποία η διαδικασία είναι ευαίσθητη. Εάν υπάρχει μετάβαση σε οποιοδήποτε από τα σήματα της λίστας, η διαδικασία θα ενεργοποιηθεί και θα πραγματοποιηθούν οι εκχωρήσεις σήματος στη διαδικασία.

4.9.1.2 Δηλώσεις αναμονής

Η δήλωση αναμονής είναι ένας μηχανισμός αναστολής (ή διακοπής) μιας διαδικασίας και επιτρέπει την εκτέλεση αναθέσεων σήματος χωρίς την ανάγκη λήξης της διαδικασίας. Όταν χρησιμοποιείτε μια δήλωση αναμονής, δεν χρησιμοποιείται μια λίστα ευαισθησίας και η διαδικασία θα ενεργοποιηθεί αμέσως. Μέσα στη διαδικασία, η δήλωση αναμονής χρησιμοποιείται για να σταματήσει και να ξεκινήσει η διαδικασία. Υπάρχουν τρεις τρόποι με τους οποίους μπορούν να χρησιμοποιηθούν οι δηλώσεις αναμονής. Η πρώτη είναι μια ατέρμων αναμονή. Στο ακόλουθο παράδειγμα, η λέξη-κλειδί *wait* χρησιμοποιείται για την αναστολή της διαδικασίας. Μόλις επιτευχθεί αυτή η δήλωση, οι εκχωρήσεις σήματος στα D1 και D2 θα εκτελεστούν και η διαδικασία θα ανασταλεί απεριόριστα.

```
process1
begin
  D1 <= ΄0΄;
  D2 <= ΄1΄;
   wait;
end process;
```

Η δεύτερη τεχνική για την χρήση μιας δήλωσης αναμονής για την αναστολή μιας διαδικασίας είναι σε συνδυασμό με τη λέξη-κλειδί *for* και μια έκφραση χρόνου. Στο ακόλουθο παράδειγμα, η διαδικασία θα ενεργοποιηθεί αμέσως, μόλις η διαδικασία φτάσει στη δήλωση αναμονής, θα ανασταλεί και μετά θα εκτελέσει την πρώτη ανάθεση σήματος στο CLK (CLK <= ‘0’). Μετά από 10ns, η διαδικασία θα ξεκινήσει ξανά. Μόλις φτάσει στη δεύτερη δήλωση αναμονής, θα ανασταλεί και μετά θα εκτελέσει τη δεύτερη ανάθεση σήματος στο CLK (CLK <= ‘1’). Μετά από άλλα 10ns, η διαδικασία θα ξεκινήσει ξανά και θα τερματιστεί αμέσως λόγω της δήλωσης τερματισμού της διαδικασίας. Αφού ολοκληρωθεί η διαδικασία, θα ενεργοποιηθεί πάλι λόγω της έλλειψης λίστας ευαισθησίας και θα επαναλάβει τη συμπεριφορά που μόλις περιγράφηκε. Αυτή η συμπεριφορά θα συνεχιστεί απεριόριστα. Αυτό το παράδειγμα δημιουργεί έναν τετραγωνικό παλμό που ονομάζεται CLK με περίοδο 20ns.

```
process 2
begin
   CLK <= ’0’; wait for 10ns;
   CLK <= ’1’;  wait for 10ns;
end process;
```

Η τρίτη τεχνική είναι σε συνδυασμό με τη λέξη-κλειδί *until* και μια συνθήκη Boolean. Στο ακόλουθο παράδειγμα, η διαδικασία θα ενεργοποιηθεί και θα ανασταλεί αμέσως. Θα συνεχιστεί μόνο όταν μια κατάσταση Boolean γίνει πραγματικότητα (Μετρητής > 12). Μόλις ισχύσει αυτή η συνθήκη, η διαδικασία θα ξεκινήσει ξανά. Μόλις φτάσει στη δεύτερη δήλωση αναμονής, θα εκτελέσει την πρώτη ανάθεση σήματος στο RollOver (RollOver <= ‘1’). Μετά από 5ns, η διαδικασία θα συνεχιστεί. Μόλις ολοκληρωθεί η διαδικασία, θα εκτελέσει τη δεύτερη εκχώρηση σήματος στο RollOver (RollOver <= ‘0’).

```
process 3
begin
wait until (Counter > 12);              -- 1η δήλωση αναμονής
   RollOver <= ’1’; wait for 5ns;      -- 2η δήλωση αναμονής
   RollOver <= ’0’;
end process;
```

Οι δηλώσεις αναμονής συνήθως δεν μπορούν να συντεθούν και χρησιμοποιούνται συχνότερα για τη δημιουργία ακολουθιών διέγερσης σε αρχεία δοκιμών (testbenches).

4.9.1.3 Μεταβλητές

Υπάρχουν καταστάσεις εντός των διαδικασιών στις οποίες είναι επιθυμητό οι αναθέσεις τιμών να γίνονται άμεσα και όχι όταν η διαδικασία αναστέλλεται. Για αυτές τις καταστάσεις, η VHDL παρέχει την έννοια μιας μεταβλητής. Μια μεταβλητή έχει τα ακόλουθα χαρακτηριστικά:

• Οι μεταβλητές υπάρχουν μόνο σε μια διαδικασία.
• Οι μεταβλητές καθορίζονται σε μια διαδικασία πριν από τη δήλωση έναρξης.
• Μόλις ολοκληρωθεί η διαδικασία, οι μεταβλητές αφαιρούνται από το σύστημα. Αυτό σημαίνει ότι οι εκχωρήσεις σε μεταβλητές δεν μπορούν να γίνουν από σημεία εκτός της διαδικασίας.
• Οι αναθέσεις σε μεταβλητές γίνονται χρησιμοποιώντας τον τελεστή ":=".
• Οι αναθέσεις σε μεταβλητές γίνονται στιγμιαία.

Μια μεταβλητή δηλώνεται πριν από την αρχική δήλωση σε μια διαδικασία. Η σύνταξη για τη δήλωση μιας μεταβλητής έχει ως εξής:

```
variable variable_name : <type> :=<init_value>;
```

4.9.2 Δομές υπό συνθήκη

Ένα από τα πιο ισχυρά χαρακτηριστικά που παρέχουν οι διεργασίες στην VHDL είναι η δυνατότητα χρήσης προγραμματιστικών εντολών όπως δηλώσεις if/then, case και loops.

4.9.2.1 Δηλώσεις if/then

Μια δήλωση if/then παρέχει έναν τρόπο για αναθέσεις σήματος υπό όρους με βάση τις συνθήκες Boolean. Το τμήμα if της δήλωσης ακολουθείται από μια συνθήκη Boolean που αν αξιολογηθεί TRUE θα προκαλέσει την εκχώρηση σήματος μετά την *then* δήλωση. Εάν η συνθήκη Boolean αξιολογηθεί FALSE, δεν γίνεται εκχώρηση. Η VHDL παρέχει πολλές παραλλαγές της δήλωσης if/then. Μια δήλωση if/then/else παρέχει μια οριστική εκχώρηση σήματος που θα γίνει εάν η συνθήκη Boolean αξιολογηθεί λανθασμένη. Μια δήλωση if/then/elsif επιτρέπει τη χρήση πολλών Boolean συνθηκών.

4.9.2.2 Δηλώσεις case

Όπως με τη δήλωση if/then, μια δήλωση case μπορεί να χρησιμοποιηθεί μόνο μέσα σε μια διαδικασία. Η δήλωση ξεκινά με την λέξη-κλειδί *case* ακολουθούμενη από το όνομα σήματος εισόδου από την τιμή του οποίου θα εξαρτώνται οι αναθέσεις τιμών. Το όνομα σήματος εισόδου μπορεί προαιρετικά να περικλείεται σε παρενθέσεις για αναγνωσιμότητα. Η λέξη-κλειδί *when* χρησιμοποιείται για τον καθορισμό μιας συγκεκριμένης τιμής (ή επιλογής) του σήματος εισόδου που θα οδηγήσει σε σχετικές εκχωρήσεις διαδοχικών σημάτων. Οι εργασίες αναγράφονται μετά το σύμβολο =>. Όταν δεν έχουν καθοριστεί όλες οι πιθανές συνθήκες εισόδου, η *when others* χρησιμοποιείται για την παροχή εκχωρήσεων σήματος για όλες τις άλλες συνθήκες εισόδου. Πολλαπλές επιλογές που αντιστοιχούν στις ίδιες εκχωρήσεις σήματος μπορούν να οριοθετηθούν στη δήλωση case.

Το σήμα εισόδου για μια δήλωση case πρέπει να είναι μοναδικό όνομα σήματος. Εάν πρόκειται να χρησιμοποιηθούν πολλαπλές κλίμακες ως είσοδος για μια δήλωση case, θα πρέπει να συνενωθούν είτε έξω από τη διαδικασία με αποτέλεσμα ένα νέο σήμα είτε εντός της διαδικασίας με αποτέλεσμα μια νέα μεταβλητή.

Οι if/then δηλώσεις μπορούν να ενσωματωθούν σε μια δήλωση case και αντιστρόφως.

4.9.2.3 Περιορισμένοι βρόχοι

Ένας βρόχος στη VHDL παρέχει έναν μηχανισμό για την εκτέλεση επαναλαμβανόμενων εργασιών σε αόριστο χρόνο. Αυτό είναι χρήσιμο σε test-benches για τη δημιουργία σημάτων διέγερσης (stimulus) όπως ρολόγια ή άλλες περιοδικές κυματομορφές. Ένας βρόχος μπορεί να χρησιμοποιηθεί μόνο σε μια διαδικασία. Η λέξη-κλειδί *loop* χρησιμοποιείται για να δηλώσει την αρχή του βρόχου. Στη συνέχεια εισάγονται διαδοχικές εκχωρήσεις σήματος. Το τέλος του βρόχου επισημαίνεται με τις λέξεις-κλειδιά *end loop*. Μέσα στο βρόχο, η *wait for*, η *wait until* και *after* δηλώσεις είναι όλες νόμιμες. Οι εκχωρήσεις σημάτων εντός ενός βρόχου θα εκτελούνται επανειλημμένα για πάντα εκτός αν συναντηθεί έξοδος ή επόμενη δήλωση. Η έκφραση *exit* παρέχει μια συνθήκη Boolean που θα αναγκάσει το βρόχο να τερματιστεί εάν η συνθήκη αξιολογηθεί αληθής. Όταν γίνεται χρήση της δήλωσης *exit*, μια πρόσθετη εκχώρηση σήματος τοποθετείται συνήθως μετά το βρόχο για να παρέχει την επιθυμητή συμπεριφορά όταν ο βρόχος δεν είναι ενεργός. Η χρήση δηλώσεων ελέγχου ροής όπως η *wait for* και η *wait after* παρέχουν ένα μέσο για να αποφευχθεί η άμεση εκτέλεση του βρόχου μετά την έξοδο. Η έκφραση *next* παρέχει έναν τρόπο για την παράλειψη των υπόλοιπων εκχωρήσεων σήματος και να ξεκινήσει η επόμενη επανάληψη του βρόχου.

4.9.2.4 While Βρόχοι

Ο βρόχος *while* αποτελεί ένα βρόχο με μια συνθήκη Boolean που ελέγχει την εκτέλεση του. Ο βρόχος θα εκτελεστεί μόνο εφόσον η κατάστασή του αξιολογηθεί αληθής.

4.9.2.5 For Βρόχοι

Η *for loop* παρέχει τη δυνατότητα δημιουργίας ενός βρόχου που θα εκτελείται για έναν προκαθορισμένο αριθμό επαναλήψεων. Το εύρος του βρόχου καθορίζεται με ακέραιους αριθμούς (min, max) στην αρχή του *for loop*. Μια μεταβλητή βρόχου δηλώνεται στον βρόχο που θα αυξάνει (ή θα μειώνεται) από το ελάχιστο έως το μέγιστο του εύρους. Η μεταβλητή βρόχου είναι ακέραιου τύπου. Εάν είναι επιθυμητό να αυξηθεί η μεταβλητή βρόχου από min έως max, η λέξη-κλειδί *for* χρησιμοποιείται κατά τον καθορισμό του εύρους του βρόχου. Για μείωση έως το ελάχιστο, η λέξη-κλειδί *downto* χρησιμοποιείται κατά τον καθορισμό του εύρους του βρόχου. Η μεταβλητή βρόχου μπορεί να χρησιμοποιηθεί εντός του βρόχου ως ευρετήριο διανυσμάτων· έτσι η *for for loop* είναι χρήσιμη για αυτόματη πρόσβαση και εκχώρηση πολλαπλών σημάτων μέσα σε μια δομή ενός βρόχου.

Οι *for loops* είναι χρήσιμες για test benches στα οποία πρόκειται να δημιουργηθεί μια σειρά προτύπων. Οι *for loops* μπορούν επίσης να συντεθούν αρκεί η πλήρης συμπεριφορά του επιθυμητού συστήματος να περιγράφεται από τον βρόχο.

4.9.3 Χαρακτηριστικά σήματος

Υπάρχουν καταστάσεις όπου θέλουμε να περιγράψουμε συμπεριφορά που βασίζεται σε κάτι περισσότερο από την τρέχουσα τιμή ενός σήματος. Για να μοντελοποιήσουμε αυτήν τη συμπεριφορά, πρέπει να καθορίσουμε περισσότερες πληροφορίες σχετικά με το σήμα. Αυτό επιτυγχάνεται χρησιμοποιώντας attributes. Τα χαρακτηριστικά παρέχουν πρόσθετες πληροφορίες για ένα σήμα διαφορετικό από την τρέχουσα τιμή του. Ένα χαρακτηριστικό μπορεί να παρέχει πληροφορίες όπως προηγούμενες τιμές, εάν μια ανάθεση έγινε σε ένα σήμα ή πότε την τελευταία φορά μια ανάθεση είχε ως αποτέλεσμα μια αλλαγή τιμής. Ένα χαρακτηριστικό σήματος υλοποιείται τοποθετώντας ένα απόστροφο (‘) μετά το όνομα του σήματος και στη συνέχεια παραθέτοντας τη λέξη-κλειδί του χαρακτηριστικού VHDL. Άλλα χαρακτηριστικά μπορούν να χρησιμοποιηθούν για τον καθορισμό του εύρους των νέων διανυσμάτων με αναφορά του μεγέθους των υπαρχόντων διανυσμάτων ή προσδιορίζοντας αυτόματα τον αριθμό των επαναλήψεων σε ένα βρόχο. Τέλος, ορισμένα χαρακτηριστικά μπορούν να χρησιμοποιηθούν για τη δημιουργία self-checking test-benches που παρακολουθούν την επίδραση των καθυστερήσεων κυκλωμάτων στη λειτουργικότητα ενός συστήματος. Ακολουθεί μια λίστα με τα κοινά χρησιμοποιούμενα, προκαθορισμένα χαρακτηριστικά σήματος VHDL (βλέπε Πίνακα 14). Το παράδειγμα ονόματος σήματος F χρησιμοποιείται για να δείξει πώς λειτουργούν τα χαρακτηριστικά βαθμωτών σημάτων. Το παράδειγμα σήματος G χρησιμοποιείται για να δείξει πώς λειτουργούν τα διανυσματικά χαρακτηριστικά με τον τύπο bit_vector (7 έως 0).

Πίνακας 14 – Χαρακτηριστικά σήματος

Χαρακτηριστικά	Πληροφορία	Τύπος
F‘event	Αληθής όταν το σήμα F αλλάζει	boolean
F‘active	Αληθής όταν γίνεται ανάθεση στο F	boolean
F‘last_event	Χρόνος τελευταίας αλλαγής του F	time
F‘last_active	Χρόνος τελευταίας ανάθεσης του F	time
F‘last_value	Προηγούμενη τιμή του F	ίδιου τύπου με F
G‘length	Μέγεθος διανύσματος (π.χ., 8)	integer
G‘left	Αριστερό άκρο διανύσματος (π.χ., 7)	integer
G‘right	Δεξί άκρο διανύσματος (π.χ., 0)	integer
G‘range	Εύρος διανύσματος “(7 downto 0)”	string

4.10 Βασικά Πακέτα

Ένα από τα μειονεκτήματα του τυπικού πακέτου VHDL είναι ότι παρέχει περιορισμένη λειτουργικότητα στους συνθέσιμους τύπους δεδομένων του. Το bit και bit_vector, ενώ μπορούν να γίνουν σύνθεση σε κύκλωμα, δεν έχουν την ικανότητα να μοντελοποιήσουν με ακρίβεια πολλές από τις τοπολογίες που εφαρμόζονται σε σύγχρονα ψηφιακά συστήματα. Πρωταρχικού ενδιαφέροντος είναι τοπολογίες που περιλαμβάνουν πολλαπλούς οδηγούς συνδεδεμένους σε ένα καλώδιο. Το τυπικό πακέτο δεν επιτρέπει αυτόν τον τύπο σύνδεσης. Επιπλέον, το τυπικό πακέτο δεν παρέχει πολλές χρήσιμες λειτουργίες για αυτούς τους τύπους, όπως αριθμητική χρήση των τελεστών + και -, λειτουργίες μετατροπής τύπου ή η ικανότητα ανάγνωσης / εγγραφής εξωτερικών αρχείων. Για να την αύξηση της λειτουργικότητας της VHDL, τα παρακάτω πακέτα περιλαμβάνονται στο σχεδιασμό, που είναι τα πιο συνηθισμένα στα σύγχρονα μοντέλα VHDL:

4.10.1 STD_LOGIC_1164

Στα τέλη της δεκαετίας του 1980, κυκλοφόρησε το πρότυπο IEEE 1164 που πρόσθεσε λειτουργικότητα στη VHDL για να επιτρέψει ένα λογικό σύστημα πολλαπλών τιμών (δηλαδή, ένα σήμα μπορεί να έχει περισσότερες τιμές από το 0 και το 1). Αυτό το πρότυπο παρείχε επίσης έναν μηχανισμό για τη σύνδεση πολλών οδηγών στο ίδιο σήμα. Μια ενημερωμένη έκδοση το 1993 με την ονομασία IEEE 1164-1993 ήταν η πιο σημαντική ενημέρωση σε αυτό το πρότυπο και περιέχει την πλειονότητα των λειτουργιών που χρησιμοποιούνται στην VHDL σήμερα. Σχεδόν όλα τα συστήματα που περιγράφονται στην VHDL περιλαμβάνουν το πρότυπο 1164 ως πακέτο. Αυτό το πακέτο περιλαμβάνεται με την προσθήκη της ακόλουθης σύνταξης στην αρχή του αρχείου VHDL.

```
library IEEE;
use IEEE.std_logic_1164.all;
```

Αυτό το πακέτο ορίζει τέσσερις νέους τύπους δεδομένων: std_ulogic, std_ulogic_vector, std_logic και std_logic_vector. Οι std_ulogic και std_logic είναι απαριθμητοί τύποι, βαθμωτοί, που μπορούν να παρέχουν ένα λογικό σύστημα πολλαπλών τιμών. Οι τύποι std_ulogic_vector και std_logic_vector είναι διανυσματικοί τύποι που περιέχουν μια γραμμική συστοιχία σημάτων τύπου std_ulogic και std_logic, αντίστοιχα.

4.10.1.1 STD_LOGIC_1164 Λογικοί τελεστές

Το std_logic_1164 περιέχει επίσης νέους ορισμούς για όλους τους λογικούς τελεστές (and, nand, or, nor, xor, xnor, not) για τους τύπους std_ulogic και std_logic. Απαιτούνται διότι αυτοί οι τύποι δεδομένων μπορούν να έχουν περισσότερες λογικές τιμές από 0 και 1.

4.10.1.2 STD_LOGIC_1164 Λειτουργίες ανίχνευσης άκρων

Το std_logic_1164 παρέχει επίσης λειτουργίες για την ανίχνευση των μεταβάσεων ενός σήματος. Οι συναρτήσεις rising edge() και falling edge() παρέχουν μια πιο ευανάγνωστη μορφή αυτής της λειτουργικότητας σε σύγκριση με την προσέγγιση (Clock�() event and Clock = '1').

4.10.1.3 STD_LOGIC_1164 Λειτουργίες μετατροπής τύπων

Πίνακας 15 – STD_LOGIC_1164 Λειτουργίες μετατροπής τύπων

Όνομα μετατροπής	Τύπος Εισόδου	Επιστρεφόμενος Τύπος
σε bit()	std_ulogic	bit
σε bitvector()	std_ulogic_vector	bit_vector
σε bitvector()	std_logic_vector	bit_vector
σε StdULogic()	bit	std_ulogic
σε StdULogicVector()	bit_vector	std_ulogic_vector
σε StdULogicVector()	std_logic_vector	std_ulogic_vector
σε StdLogicVector()	bit_vector	std_logic_vector
σε StdLogicVector()	std_ulogic_vector	std_logic_vector

Κατά τη χρήση αυτών των λειτουργιών (βλέπε Πίνακα 15), το όνομα λειτουργίας και το σήμα εισόδου τοποθετούνται στα δεξιά του τελεστή ανάθεσης και το σήμα προορισμού τοποθετείται στα αριστερά.

4.10.2 NUMERIC_STD

Το πακέτο numeric_std παρέχει αριθμητικό υπολογισμό για τους τύπους std_logic και std_logic_vector. Κατά την εκτέλεση δυαδικής αριθμητικής, τα αποτελέσματα των αριθμητικών πράξεων και συγκρίσεων ποικίλλουν σε μεγάλο βαθμό ανάλογα με το αν ο δυαδικός αριθμός είναι unsigned ή signed. Ως αποτέλεσμα, το πακέτο numeric_std καθορίζει δύο νέους τύπους δεδομένων, unsigned και signed. Ένας unsigned τύπος ορίζεται για να έχει το MSB του στην αριστερή θέση του διανύσματος και το LSB στην πιο δεξιά θέση του διανύσματος. Ένας signed αριθμός χρησιμοποιεί την παράσταση συμπληρώματος ως προς δύο για τους αρνητικούς αριθμούς με το αριστερότερο bit του διανύσματος να είναι το bit προσήμου. Η χρήση unsigned/signed τύπων παρέχει την ερμηνεία του τρόπου με τον οποίο θα λειτουργούν οι τελεστές αριθμητικής, λογικής και σύγκρισης. Αυτό σημαίνει επίσης ότι το πακέτο numeric_std απαιτεί το std_logic_1164 να περιλαμβάνεται πάντα. Ενώ το πακέτο numeric_std περιλαμβάνει μια κλήση συμπερίληψης του πακέτου std_logic_1164, είναι συνηθισμένο να συμπεριλαμβάνονται ρητά τόσο τα πακέτα std_logic_1164 όσο και τα πακέτα numeric_std στο κύριο αρχείο VHDL. Ο μεταγλωττιστής VHDL θα αγνοήσει τις περιττές δηλώσεις πακέτων. Η σύνταξη για τη συμπερίληψη αυτών των πακέτων έχει ως εξής:

```
library IEEE;
use IEEE.std_logic_1164.all; -- καθορίζει τύπους std_ulogic & std_logic
use IEEE.numeric_std.all;    -- καθορίζει τύπους unsigned & signed
```

4.10.2.1 NUMERIC_STD Αριθμητικές συναρτήσεις

Το πακέτο numeric_std παρέχει υποστήριξη για μια ποικιλία αριθμητικών συναρτήσεων για τους τύπους unsigned ή signed. Αυτές περιλαμβάνουν τις λειτουργίες +, -, *, /, mod, rem, και abs. Αυτές οι αριθμητικές λειτουργίες συμπεριφέρονται διαφορετικά για τους τύπους unsigned έναντι signed, αλλά ο μεταγλωτιστής VHDL θα χρησιμοποιήσει αυτόματα τη σωστή λειτουργία βάσει των τύπων των ορισμάτων εισαγωγής.

Τα περισσότερα εργαλεία σύνθεσης υποστηρίζουν τους τελεστές πρόσθεσης, αφαίρεσης και πολλαπλασιασμού σε αυτό το πακέτο. Η χρήση του πακέτου numeric_std δίνει τη δυνατότητα μοντελοποίησης αυτών των αριθμητικών λειτουργιών με σύνθετο τύπο δεδομένων χρησιμοποιώντας τους πιο οικείους μαθηματικούς τελεστές. Οι λειτουργίες διαίρεσης, modulo, υπολοίπου και απόλυτης τιμής δεν γίνονται σύνθεση άμεσα από αυτό το πακέτο.

4.10.2.2 NUMERIC_STD Λογικές συναρτήσεις
Το πακέτο numeric_std παρέχει υποστήριξη για όλους τους λογικούς τελεστές (and, nand, or, nor, xor, xnor, not) για τύπους unsigned ή signed. Παρέχει επίσης δύο νέες λειτουργίες shift_left () και shift_right (). Αυτές οι λειτουργίες ολίσθησης θα συμπληρώσουν την κενή θέση στο διάνυσμα μετά την ολίσθηση με 0 (λογικές ολισθήσεις). Αυτό το πακέτο παρέχει επίσης δύο νέες λειτουργίες περιστροφής rotate_left () και rotate_right ().

4.10.2.3 NUMERIC_STD Συγκριτικές συναρτήσεις
Το πακέτο numeric_std παρέχει υποστήριξη για όλες τις λειτουργίες σύγκρισης για τύπους χωρίς unsigned ή signed. Αυτές περιλαμβάνουν >, <, <=, >=, = και /=. Αυτές οι συγκρίσεις επιστρέφουν τύπο Boolean.

4.10.2.4 NUMERIC_STD Συναρτήσεις ανίχνευσης άκρων
Το numeric_std παρέχει επίσης τις συναρτήσεις rising edge () και falling edge () για την ανίχνευση μετάβασης rising ή falling edge για τύπους μή ή και προσημασμένους.

4.10.2.5 NUMERIC_STD Συναρτήσεις μετατροπής
Το πακέτο numeric_std περιέχει μια ποικιλία χρήσιμων λειτουργιών μετατροπής. Ιδιαίτερης χρησιμότητας είναι οι λειτουργίες μεταξύ του ακέραιου τύπου και προς/από τον unsigned/signed. Αυτό επιτρέπει την εφαρμογή μοντέλων συμπεριφοράς για μετρητές, προσθέτες και αφαιρέτες χρησιμοποιώντας τον πιο ευανάγνωστο ακέραιο τύπο. Αφού περιγραφεί η λειτουργικότητα, μπορεί να χρησιμοποιηθεί μια μετατροπή για να μετατρέψει το αποτέλεσμα σε τύπους unsigned ή signed για να παρέχει μια συνθέσιμη έξοδο. Κατά τη μετατροπή ενός ακέραιου αριθμού σε ένα διάνυσμα, περιλαμβάνεται ένα όρισμα μεγέθους. Το όρισμα μεγέθους είναι ακέραιου τύπου και παρέχει τον αριθμό των bit στο διάνυσμα στο οποίο θα μετατραπεί ο ακέραιος (βλέπε Πίνακα 17).

Πίνακας 16 – NUMERIC_STD Συναρτήσεις μετατροπής

Όνομα συνάρτησης μετατροπής	Τύπος Εισόδου	Επιστρεφόμενος Τύπος
to_integer()	unsigned	integer
to_integer()	signed	integer
to_unsigned()	integer, <size>	unsigned (size-1 downto 0)
to_signed()	integer, <size>	signed (size-1 downto 0)

4.10.2.6 NUMERIC_STD Συναρτήσεις μετάδοσης τύπου
Η VHDL περιέχει ένα σύνολο ενσωματωμένων λειτουργιών που χρησιμοποιούνται συνήθως με το πακέτο numeric_std για τη μετατροπή μεταξύ std_logic_vector και unsigned/signed. Δεδομένου ότι οι τύποι unsigned & signed βασίζονται στον υποκείμενο τύπο std_logic_vector, η μετατροπή είναι απλώς γνωστή ως casting. Τα παρακάτω είναι οι ενσωματωμένες δυνατότητες τύπου casting σε VHDL (Πίνακας 18).

Πίνακας 17 – NUMERIC_STD Συναρτήσεις μετάδοσης τύπου

Όνομα συνάρτησης μετατροπής	Τύπος Εισόδου	Επιστρεφόμενος Τύπος
std_logic_vector()	unsigned	std_logic_vector
std_logic_vector()	signed	std_logic_vector
unsigned()	std_logic_vector	unsigned
signed()	std_logic_vector	signed

Κατά τη χρήση αυτών των τύπων, τοποθετούνται στη δεξιά πλευρά της ανάθεσης ακριβώς ως λειτουργία μετατροπής.

Οι τύποι μετάδοσης και οι συναρτήσεις μετατροπής μπορούν να συνδυαστούν για την εκτέλεση πολλαπλών μετατροπών σε μία ανάθεση. Αυτό είναι χρήσιμο κατά τη μετατροπή μεταξύ τύπων που δεν έχουν άμεση λειτουργία μετάδοσης ή μετατροπής.

4.10.3 TEXTIO και STD_LOGIC_TEXTIO

Το πακέτο textio παρέχει τη δυνατότητα ανάγνωσης και εγγραφής προς/από εξωτερική είσοδο/έξοδο (I/O). Το εξωτερικό I/O αναφέρεται σε στοιχεία όπως αρχεία ή την τυπική είσοδο/έξοδο ενός υπολογιστή. Αυτό το πακέτο περιέχει συναρτήσεις που επιτρέπουν την ανάγνωση και εγγραφή των τιμών των σημάτων και των μεταβλητών εκτός από τις συμβολοσειρές. Αυτό επιτρέπει τη δημιουργία πιο εξελιγμένων μηνυμάτων εξόδου σε σύγκριση με τη δήλωση αναφοράς μόνο, η οποία μπορεί να παράγει μόνο συμβολοσειρές εξόδου. Η ικανότητα ανάγνωσης σε τιμές από ένα αρχείο επιτρέπει τη δημιουργία εξελιγμένων προτύπων δοκιμών εκτός της VHDL και στη συνέχεια την ανάγνωση κατά τη διάρκεια της προσομοίωσης για τη δοκιμή ενός συστήματος.
Το πακέτο textio δεν μπορεί να γίνει σύνθεση και χρησιμοποιείται μόνο σε test-benches. Το πακέτο textio βρίσκεται στη βιβλιοθήκη STD και περιλαμβάνεται σε σχεδιασμό VHDL χρησιμοποιώντας την ακόλουθη σύνταξη:

```
library STD;
use STD.textio.all;
```

Αυτό το πακέτο από μόνο του υποστηρίζει τύπους ανάγνωσης και γραφής bit, bit_vector, integer, χαρακτήρα και string. Δεδομένου ότι η πλειονότητα των συνθέσιμων σχεδίων χρησιμοποιούν τύπους std_logic και std_logic_vector, δημιουργήθηκε ένα πρόσθετο πακέτο που πρόσθεσε υποστήριξη για αυτούς τους τύπους. Το πακέτο ονομάζεται std_logic_textio και βρίσκεται στη βιβλιοθήκη IEEE. Η σύνταξη για τη συμπερίληψη αυτού του πακέτου είναι παρακάτω:

```
library IEEE;
use IEEE.std_logic_textio.all;
```

Το πακέτο textio καθορίζει δύο νέους τύπους για διασύνδεση με εξωτερικά στοιχεία I/O. Αυτοί οι τύποι είναι το file και line. Ο τύπος file χρησιμοποιείται για την ανάγνωση/εγγραφή εντός του σχεδιασμού σε VHDL.

Οι πληροφορίες μέσα σε ένα αρχείο είναι προσβάσιμες (για ανάγνωση ή για έγγραφή) χρησιμοποιώντας την έννοια της γραμμής. Στο πακέτο textio, ένα αρχείο ερμηνεύεται ως ακολουθία γραμμών, η καθεμία περιέχει είτε μια σειρά χαρακτήρων είτε μια ακέραια τιμή. Ο τύπος γραμμής χρησιμοποιείται ως προσωρινή μνήμη (buffer) κατά την πρόσβαση σε μια γραμμή εντός του αρχείου. Κατά την πρόσβαση σε ένα αρχείο, δημιουργείται μια μεταβλητή τύπου γραμμής. Αυτή η μεταβλητή χρησιμοποιείται έπειτα είτε για να κρατήσει πληροφορίες που διαβάζονται από μια γραμμή στο αρχείο είται για να κρατήσει τις πληροφορίες που πρόκειται να γραφτούν σε μια γραμμή στο αρχείο. Μια μεταβλητή είναι απαραίτητη για αυτήν τη συμπεριφορά, καθώς οι αναθέσεις προς/από το αρχείο πρέπει να γίνουν αμέσως. Ως εκ τούτου, μια μεταβλητή γραμμής δηλώνεται πάντα μέσα σε μια διαδικασία πριν από τη δήλωση begin.

Υπάρχουν δύο συναρτήσεις που επιτρέπουν τη μεταφορά πληροφοριών μεταξύ μιας μεταβλητής γραμμής στην VHDL και μιας γραμμής σε ένα αρχείο. Αυτές είναι οι readline () και writeline ().

Η μεταφορά πληροφοριών μεταξύ μιας μεταβλητής γραμμής και μιας γραμμής σε ένα αρχείο χρησιμοποιώντας αυτές τις συναρτήσεις πραγματοποιείται σε ολόκληρη τη γραμμή.

Δεν υπάρχει μηχανισμός ανάγνωσης ή εγγραφής μόνο ενός μέρους της γραμμής σε ένα αρχείο. Μόλις ανοίξει/δημιουργηθεί ένα αρχείο χρησιμοποιώντας μια δήλωση αρχείου, αποκτούμε πρόσβαση στις γραμμές με τη σειρά που εμφανίζονται στο αρχείο. Η πρώτη συνάρτηση που ονομάζεται (είτε readline () είτε writeline ()) θα έχει πρόσβαση στην πρώτη γραμμή του αρχείου. Την επόμενη φορά που θα κληθεί η συνάρτηση, θα έχει πρόσβαση στη δεύτερη γραμμή του αρχείου.

Αυτό θα συνεχιστεί έως ότου έχουν προσπελαστεί όλες οι γραμμές. Το πακέτο textio παρέχει μια λειτουργία που υποδεικνύει πότε έχει φτάσει το τέλος του αρχείου κατά την εκτέλεση μιας γραμμής ανάγνωσης. Αυτή η συνάρτηση ονομάζεται endfile () και επιστρέφει μια τιμή τύπου Boolean. Αυτή η συνάρτηση θα επιστρέψει true μόλις φτάσουμε το τέλος του αρχείου. Παρέχονται δύο επιπλέον συναρτήσεις για την προσθήκη ή ανάκτηση πληροφοριών από/προς τη μεταβλητή γραμμής εντός του VHDL test bench, οι read () και write ().

Όταν χρησιμοποιείται η συνάρτηση read (), οι πληροφορίες στη μεταβλητή γραμμής αντιμετωπίζονται ως οριοθετημένες μέσα στη γραμμή. Αυτό σημαίνει ότι κάθε συνάρτηση read () θα ανακτήσει τις πληροφορίες από τη μεταβλητή γραμμής έως ότου φτάσει σε ένα κενό διάστημα. Αυτό επιτρέπει τη χρήση πολλαπλών read () για την ανάλυση των πληροφοριών στην ίδια γραμμή. Η μεταβλητή προορισμού πρέπει να είναι του κατάλληλου τύπου και μεγέθους των πληροφοριών που διαβάζονται από το αρχείο.

Κατά τη χρήση της συνάρτησης write (), το source_destination θεωρείται ότι είναι τύπου bit, bit_vector, integer, std_logic ή std_logic_vector. Πολλαπλές συναρτήσεις εγγραφής μπορούν να χρησιμοποιηθούν για την εισαγωγή πληροφοριών στη μεταβλητή γραμμής. Κάθε επόμενη εγγραφή προσαρτά τις πληροφορίες στο τέλος της συμβολοσειράς. Αυτό επιτρέπει την παρεμβολή διαφορετικών τύπων πληροφοριών (π.χ. κείμενο, τιμή σήματος, κείμενο κ.λπ.).

4.11 Test-Benches

Η λειτουργική επαλήθευση ενός κυκλώματος σε VHDL επιτυγχάνεται μέσω προσομοίωσης χρησιμοποιώντας ένα test-bench. Ένα test-bench είναι ένα σύστημα VHDL που δημιουργεί το σύστημα που πρόκειται να δοκιμαστεί ως component και στη συνέχεια δημιουργεί τα μοτίβα εισόδου και παρατηρεί τις εξόδους. Η VHDL παρέχει μια ποικιλία δυνατοτήτων σχεδιασμού test-benches που μπορούν να αυτοματοποιήσουν την παραγωγή σημάτων διέγερσης και να παρέχουν αυτοματοποιημένο έλεγχο εξόδου. Αυτές οι δυνατότητες μπορούν να επεκταθούν συμπεριλαμβάνοντας πακέτα που εκμεταλλεύονται την ανάγνωση/εγγραφή σε αρχεία για είσοδο ή έξοδο αντίστοιχα.

4.11.1 Overview

Σε δοκιμές βασισμένες σε HDL, το υπό δοκιμή σύστημα ονομάζεται συχνά συσκευή υπό δοκιμή (DUT) ή μονάδα υπό δοκιμή (UUT). Τα test-benches χρησιμοποιούνται μόνο για προσομοίωση, ώστε να μπορούμε να εναλλάσσουμε το διάνυσμα των σημάτων στην είσοδο και να παρατηρούμε το διάνυσμα των σημάτων στην έξοδο. Η VHDL περιέχει επίσης συγκεκριμένη λειτουργικότητα για να αναφέρει την κατάσταση μιας δοκιμής και επίσης να ελέγχει αυτόματα ότι οι έξοδοι είναι σωστές.

4.12 Μοντελοποίηση FSM (finite state machines)

Σε αυτή την ενότητα, θα εξετάσουμε τη μοντελοποίηση μηχανών πεπερασμένων καταστάσεων (FSM). Οι FSM αποτελούν από τα πιο ισχυρά κυκλώματα σε ένα ψηφιακό σύστημα επειδή μπορεί να λάβουν αποφάσεις για την επόμενη έξοδο με βάση, τόσο τις εισόδους αλλά και με βάση την παρούσα κατάσταση.

4.12.1 Παράδειγμα διαδικασίας σχεδίασης FSM

Η πιο συνηθισμένη πρακτική μοντελοποίησης για FSM είναι η δημιουργία ενός νέου απαριθμητού τύπου δεδομένων καθορισμένου από τον χρήστη που μπορεί να πάρει ονόματα από το διάγραμμα καταστάσεων. Στη συνέχεια δημιουργούνται δύο σήματα αυτού του τύπου, το current_state και το next_state. Μόλις δημιουργηθούν αυτά τα σήματα, όλα τα λειτουργικά μπλοκ στις μηχανές καταστάσεων μπορούν να χρησιμοποιήσουν αυτά τα ονόματα κατάστασης στην εκχώρηση σήματος υπό όρους. Ο synthesizer θα εκχωρήσει αυτόματα τους κωδικούς κατάστασης με βάση την πιο αποτελεσματική χρήση της τεχνολογίας στόχου (π.χ. δυαδικός, κ.λ.π.). Μέσα στο state machine μοντέλο της VHDL, χρησιμοποιούνται τρεις διαδικασίες για την περιγραφή κάθε λειτουργικού μπλοκ, μνήμης κατάστασης, λογικής επόμενης κατάστασης και λογικής εξόδου.

4.12.1.1 Μοντελοποίηση των finite state machines

Το πρώτο βήμα είναι η δημιουργία ενός νέου, προσδιοριζόμενου από τον χρήστη τύπου δεδομένων που μπορεί να λάβει τιμές που ταιριάζουν με τα περιγραφικά ονόματα κατάστασης που έχουμε επιλέξει στο διάγραμμα κατάστασης (δηλαδή, ihu_closed και ihu_open). Αυτό επιτυγχάνεται δηλώνοντας έναν νέο τύπο πριν από την αρχική δήλωση στην αρχιτεκτονική με τον τύπο λέξης-κλειδιού. Π.χ., θα δημιουργήσουμε έναν νέο τύπο που ονομάζεται State_Type και θα απαριθμήσουμε ρητά τις τιμές που μπορεί να λάβει. Αυτός ο τύπος μπορεί τώρα να χρησιμοποιηθεί σε μελλοντικές δηλώσεις σημάτων/σήματος. Στη συνέχεια, δημιουργούμε δύο νέα σήματα που ονομάζονται current_state και next_state του τύπου State_Type. Αυτά τα δύο σήματα θα χρησιμοποιηθούν σε όλο το VHDL μοντέλο για να παρέχεται μια υψηλού επιπέδου, ευανάγνωστη περιγραφή της συμπεριφοράς των μηχανών πεπερασμένων καταστάσεων. Η ακόλουθη σύνταξη περιγράφει τα προλεγόμενα:

```
type  State_Type is (ihu_closed, ihu_open);
signal current_state, next_state : State_Type;
```

4.12.1.2 Η Next State λογική διαδικασία

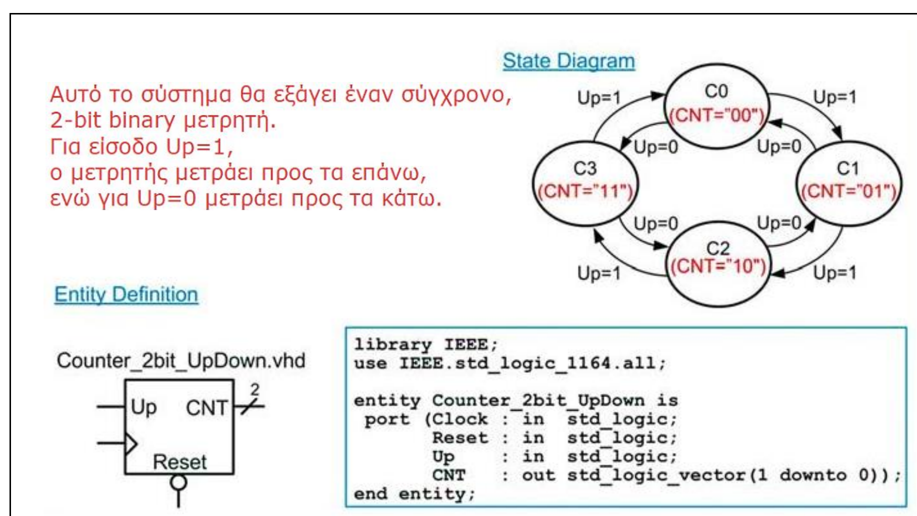
Η next state logic είναι συνδυαστική λογική, επομένως πρέπει να συμπεριλάβουμε όλα τα σήματα εισόδου που το κύκλωμα λαμβάνει υπόψη στον επόμενο υπολογισμό κατάστασης στη λίστα ευαισθησίας. Το σήμα current_state θα συμπεριλαμβάνεται πάντα στη λίστα ευαισθησίας της επόμενης διαδικασίας λογικής κατάστασης επιπλέον τυχόν εισόδων στο σύστημα.

4.12.1.3 Η Output λογική διαδικασία

Η λογική εξόδου είναι συνδυαστική λογική και πρέπει να συμπεριλάβουμε όλα τα σήματα εισόδου που αυτό το κύκλωμα λαμβάνει υπόψη στις εκχωρήσεις εξόδου. Το current_state θα συμπεριλαμβάνεται πάντα στη λίστα ευαισθησίας. Εάν η FSM είναι μηχανή Mealy, τότε οι είσοδοι του συστήματος θα συμπεριληφθούν επίσης στη λίστα ευαισθησίας. Εάν είναι μηχανή Moore, τότε μόνο το current_state θα υπάρχει στη λίστα ευαισθησίας. Πάντα συμπεριλαμβάνουμε μια ρήτρα when others για να διασφαλίσουμε ότι η state machine έχει σαφή συμπεριφορά εξόδου σε περίπτωση σφάλματος. Συνδυάζοντας όλα αυτά στην αρχιτεκτονική VHDL αποδίδει ένα λειτουργικό μοντέλο για την FSM που μπορεί να προσομοιωθεί και να γίνει σύνθεση.

4.12.2 Παράδειγμα σχεδίασης FSM

Το Σχήμα 35 δείχνει την περιγραφή του σχεδιασμού και τον ορισμό οντότητας για έναν δυαδικό μετρητή FSM.



Σχήμα 35 – Περιγραφή μοντέλου και ορισμός οντότητας ενός δυαδικού μετρητή

Το Σχήμα 36 δείχνει την αρχιτεκτονική για τον προηγούμενο μετρητή.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Counter_2bit_UpDown is
 port (Clock, Reset : in  std_logic;
       Up           : in  std_logic;
       CNT          : out std_logic_vector(1 downto 0));
end entity;

architecture Counter_2bit_UpDown_arch of Counter_2bit_UpDown is

 type   State_Type is (C0, C1, C2, C3);
 signal current_state, next_state : State_Type;

 begin
---------------------------------------------------------
  STATE_MEMORY : process (Clock, Reset)
    begin
      if (Reset = '0') then
        current_state <= C0;
      elsif (Clock'event and Clock='1') then
        current_state <= next_state;
      end if;
  end process;
---------------------------------------------------------
  NEXT_STATE_LOGIC : process (current_state, Up)
    begin
      case (current_state) is
        when C0     => if (Up = '1') then
                         next_state <= C1;
                       else
                         next_state <= C3;
                       end if;
        when C1     => if (Up = '1') then
                         next_state <= C2;
                       else
                         next_state <= C0;
                       end if;
        when C2     => if (Up = '1') then
                         next_state <= C3;
                       else
                         next_state <= C1;
                       end if;
        when C3     => if (Up = '1') then
                         next_state <= C0;
                       else
                         next_state <= C2;
                       end if;
        when others => next_state <= C0;
      end case;
  end process;
---------------------------------------------------------
  OUTPUT_LOGIC : process (current_state)
    begin
      case (current_state) is
        when C0     => CNT <= "00";
        when C1     => CNT <= "01";
        when C2     => CNT <= "10";
        when C3     => CNT <= "11";
        when others => CNT <= "00";
      end case;
  end process;

end architecture;
```

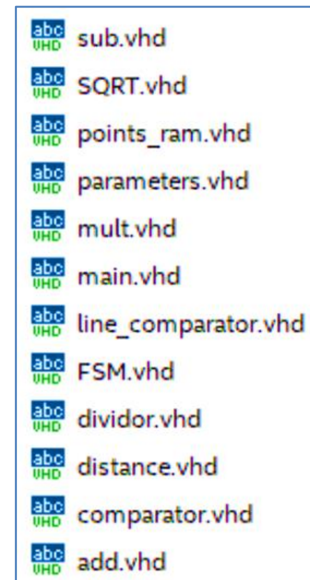
Ο μετρητής είναι μια μηχανή Moore, οπότε η έξοδος εξαρτάται μόνο από την current state.

Σχήμα 36 – Αρχιτεκτονική δυαδικού μητρητή & προσομοίωση

5 Σύνθεση - Προσομοίωση RANSAC

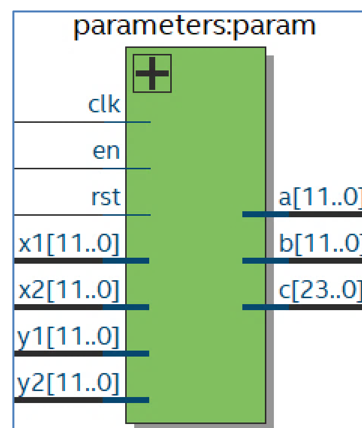
5.1 Μέθοδος 1 - Pipelined αρχιτεκτονική σχεδίαση (σύγχρονη)

Η φιλοσοφία σχεδίασης για την αρχιτεκτονική αυτή είναι η εξής: Μέσω ενός αρχείου ανάγνωσης (points_ram), εισάγουμε στις παραμέτρους (parameters), τις συντεταγμένες δύο σημείων και υπολογίζουμε τα a, b, c της ευθείας που συνθέτουν. Στη συνέχεια μέσω του distance module υπολογίζουμε τις αποστάσεις δύο σημείων κάθε φορά από την ευθεία. Εισάγουμε τις δύο αποστάσεις στο comparator module και σε περίπτωση που κάποια εξ᾽αυτών είναι μικρότερη από την τιμή κατωφλίου θεωρούμε το σημείο ως inlier. Τελικώς λαμβάνουμε τον αριθμό των inliers για κάθε ευθεία καθώς και το άθροισμα των αποστάσεών τους. Ένας τελικός έλεγχος γίνεται για την περίπτωση που προκύψουν ευθείες με τον ίδιο αριθμό inliers, οπότε ως best line fit επιλέγεται μέσω του line_comparator module, αυτή με το μικρότερο άθροισμα των αποστάσεων.
Αναλυτικά τα διάφορα modules περιγράφονται παρακάτω:



Add, Sub, Mult [Παράρτημα 132]
Λαμβάνονται δύο σήματα εισόδου a, b, με πλάτος που έχει αρχικοποιηθεί από την παράμετρο WIDTH στη λίστα generic, κι όποτε υπάρχει αλλαγή σε μία από τις δύο εισόδους εκτελείται αντίστοιχα πρόσθεση, αφαίρεση ή πολλαπλασιασμός μεταξύ τους.

Parameters [Παράρτημα 133]
Υπολογίζονται οι παράμετροι (βλ. σχ. 37) μιας εξίσωσης γραμμής μεταξύ δύο σημείων (x_1, y_1) & (x_2, y_2), δηλ. τα a, b, c της γενικής εξίσωσης $ax + by + c = 0$. Η αναπαράσταση των αριθμών είναι fixed point 12 bit αριθμοί (1_7.4) για a, b και 24 bit (1_15.8) για το c, ως εξής:
ένα bit για το σύμβολο,
7 ή 15 bit για τον ακέραιο αριθμό
και 4 ή 8 bit για το κλάσμα.

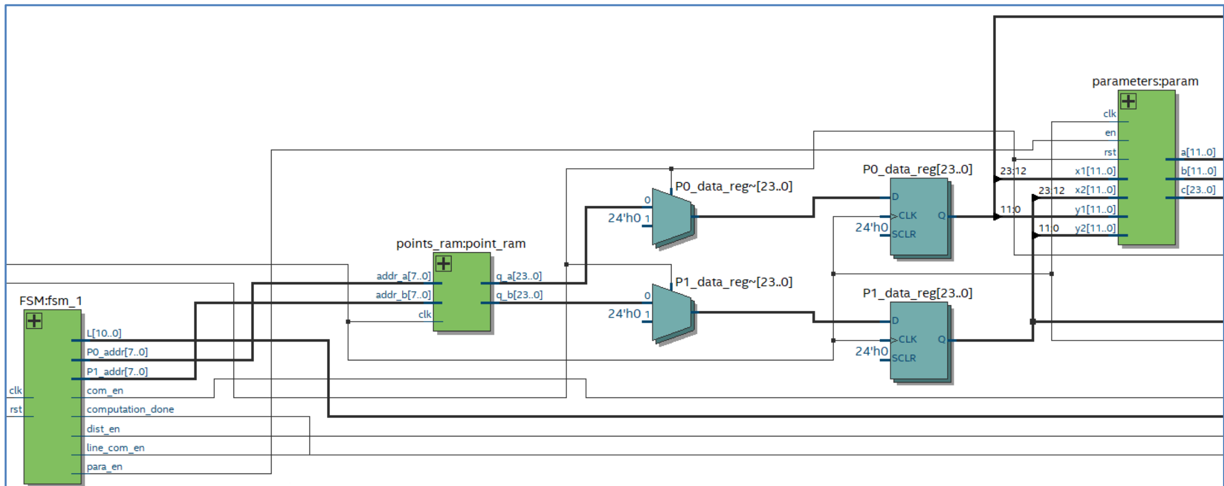


Σχήμα 37 – parameters.vhd

Οι εξισώσεις που εφαρμόζονται είναι:

$a = y_1 - y_2;$
$b = x_2 - x_1;$
$c = y_1*(x_1 - x_2) + x_1*(y_2 - y_1);$

Στο τέλος γίνεται χρήση μιας process για να λάβουμε τις τιμές των a, b, c στην έξοδο (βλ. σχ. 38).



Σχήμα 38 – paramers rtl

Distance [Παράρτημα 135]
Υπολογίζεται η απόσταση μεταξύ της γραμμής με παραμέτρους a, b, c και του σημείου (x, y). Εκτελούνται όλοι οι υπολογισμοί ως συνδυαστικά κυκλώματα. Χρησιμοποιούμε τις ενότητες add, sub, mult, dividor και sqrt.

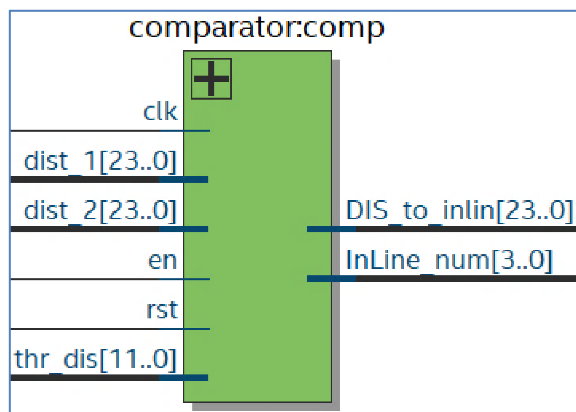
Οι εξισώσεις που εφαρμόζονται είναι

dist = abs((ax+by+c) / (sqrt(a^2 + b^2)))

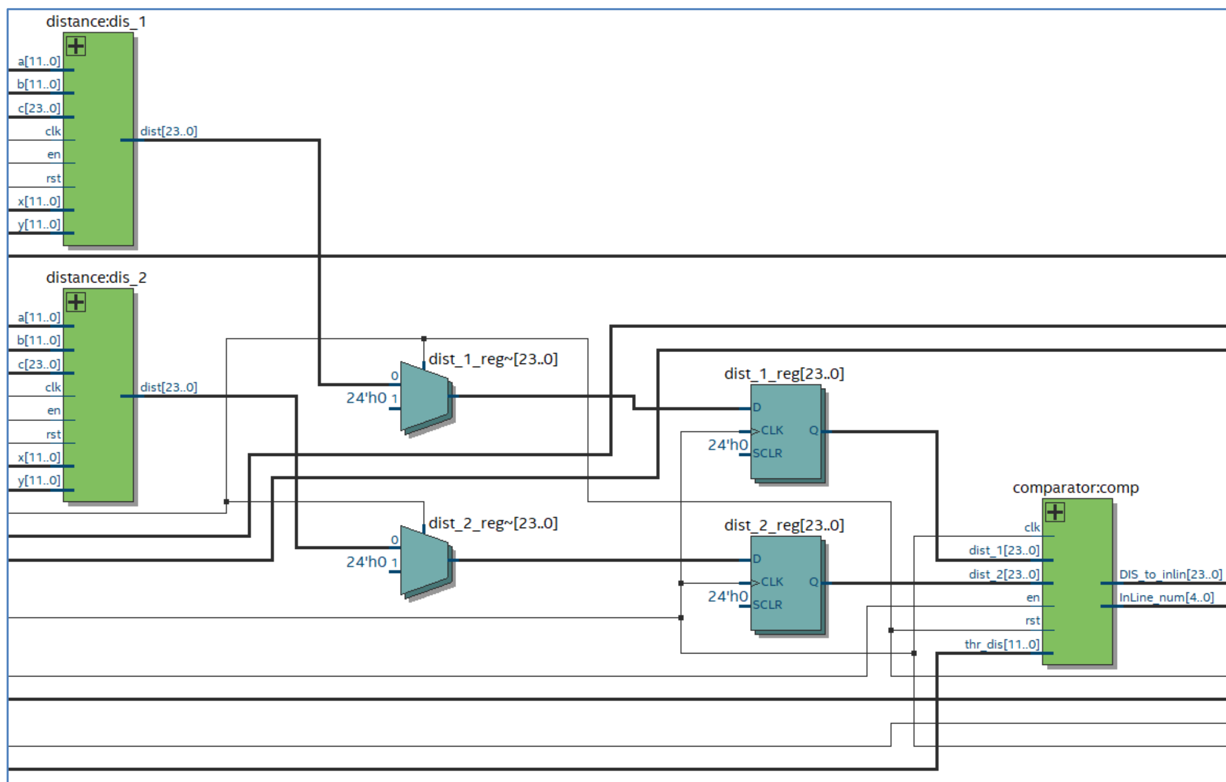
SQRT [Παράρτημα 136]
Χρησομοποιήθηκε κώδικας από το παρακάτω blog:
https://vhdlguru.blogspot.com/2010/03/vhdl-function-for-finding-square-root.html

Dividor [Παράρτημα 136]
Η είσοδος (a) αναπαρίσταται ως σταθερό σημείο (24.8) και πρέπει να μετατοπιστεί αριστερά κατα 8 που σημαίνει πολλαπλασιασμός επί 2^8.
Η είσοδος b είναι σταθερό σημείο (24.8) χωρίς μετατόπιση.
Η έξοδος c δεν είναι σταθερή, αλλά τα τελευταία 8 σημεία αντιπροσωπεύουν τον δεκαδικό αριθμό οπότε απαιτείται πολλαπλασιασμός επί 2^8 πρίν τον dividor και διαίρεση μετά, ώστε να ληφθούν τα 8 ψηφία που αντιπροσωπεύουν τον δεκαδικό αριθμό.

Comparator [Παράρτημα 137]
Αυτή η λειτουργική μονάδα λαμβάνει τις αποστάσεις έως δύο κάθε φορά, ελέγχει εάν οποιαδήποτε από τις δύο αποστάσεις είναι μικρότερη από την απόσταση κατωφλίου (thr_dis), που σημαίνει ότι αυτές οι αποστάσεις είναι σημείων inliers και αυξάνει μια μεταβλητή μετρητή κατά ένα (βλ. σχ. 39).
Στη συνέχεια προσθέτει την απόσταση των inliers σε μια μεταβλητή που ονομάζεται (DIS_to_inlin).
Τελικά τόσο ο αριθμός των inliers όσο και το άθροισμα των αποστάσεών τους λαμβάνονται ως έξοδοι (βλ. σχ. 40).



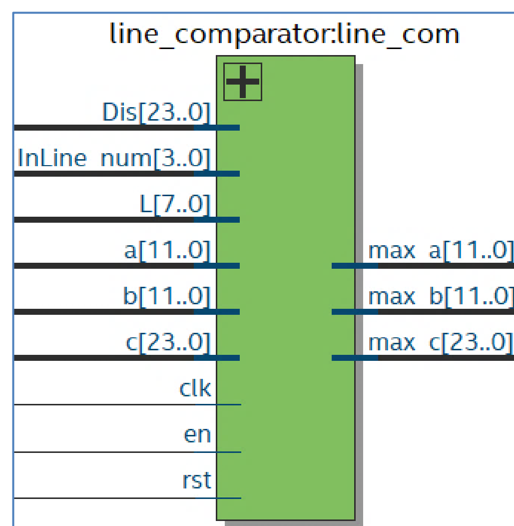
Σχήμα 39 – comparator.vhd



Σχήμα 40 – comparator rtl

Line Comparator [Παράρτημα 138]
Μετά από κάθε στάδιο υπολογισμού γραμμής, καταλήγουμε με παραμέτρους γραμμής a, b, c, τον αριθμό των inliers καθώς και την απόσταση μεταξύ της γραμμής και των inliers (βλ. σχ. 41).

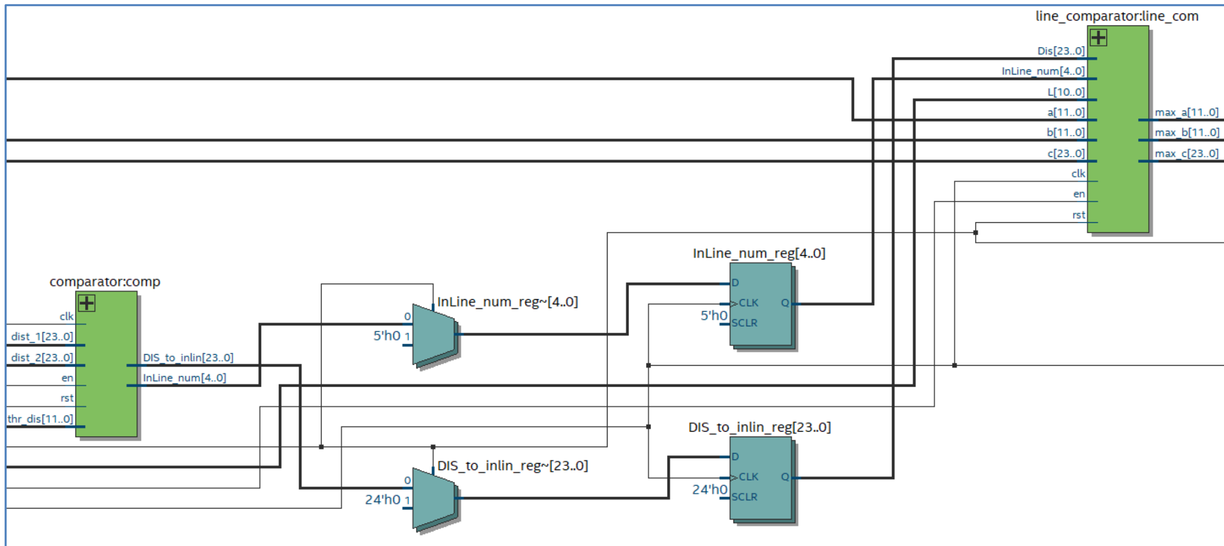
Σε αυτήν την οντότητα αναζητάμε τη γραμμή με τον μέγιστο αριθμό inliers. Στην περίπτωση που εντοπίζονται γραμμές με τον ίδιο αριθμό inliers, εξετάζουμε την απόσταση μεταξύ γραμμής - inliers και επιλέγουμε αυτήν με την ελάχιστη απόσταση (βλ. σχ. 42).



Σχήμα 41 – line_comparator.vhd

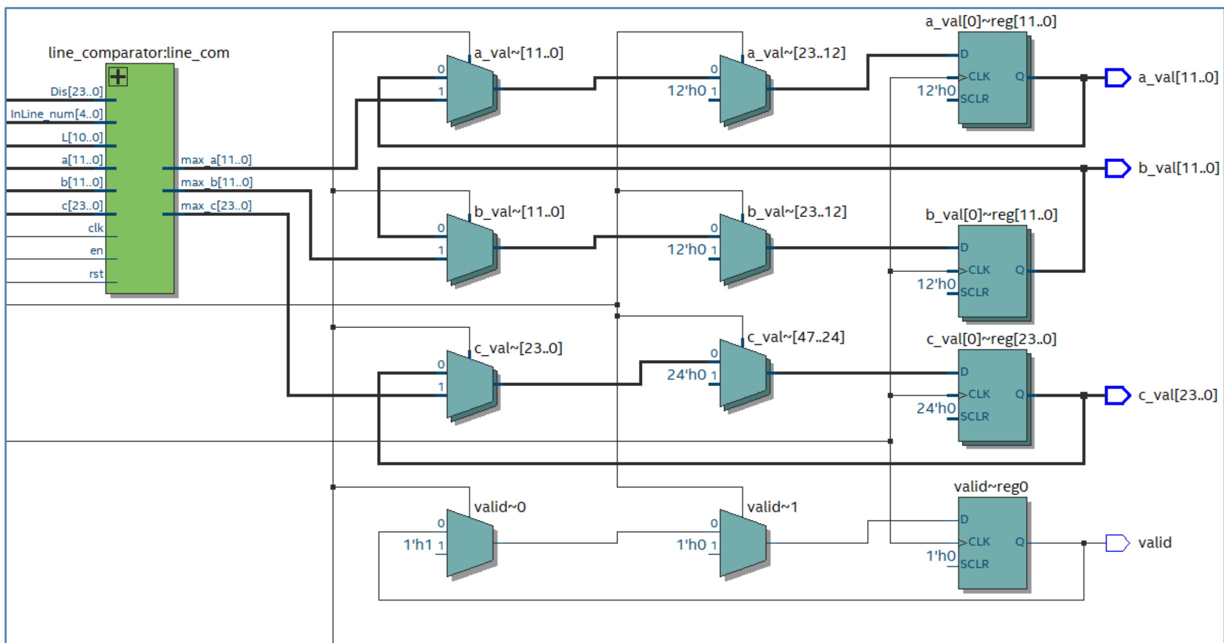
Θα μπορούσε να εκφραστεί ως:

```
max = 0;
Arrays [5];
for (int I = 0, I <5, I++)
{
	if (s[i]> max)
	{
	max = s[i]
	}
}
```



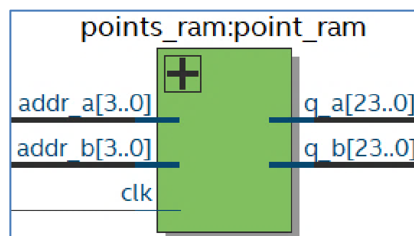
Σχήμα 42 – line_comparator rtl (a)

Κάθε στοιχείο συγκρίνεται με ενδιάμεσες τιμές που ονομάζονται inter_InLine_num αρχικά = μηδέν (βλ. σχ. 43) και αν βρεθεί μια γραμμή με μεγαλύτερο αριθμό inliers_num από την τρέχουσα, αποθηκεύουμε όλες τις νέες τιμές. Τέλος, αφού δώσουμε στην οντότητα όλες τις γραμμές του προβλήματός μας, θα περιέχει τις πληροφορίες της καλύτερης γραμμής.



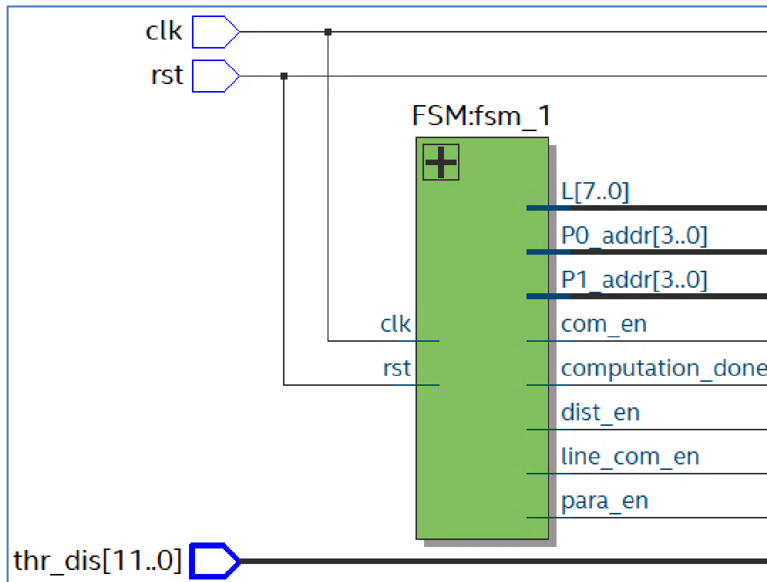
Σχήμα 43 – line_comparator rtl (b)

Points_ram [Παράρτημα 139]
Είναι μια μνήμη ανάγνωσης τιμών από αρχείο και τις παρέχει ανα δύο στο σύστημα από τις παρεχόμενες διευθύνσεις στις θύρες εισόδου (βλ. σχ. 44).

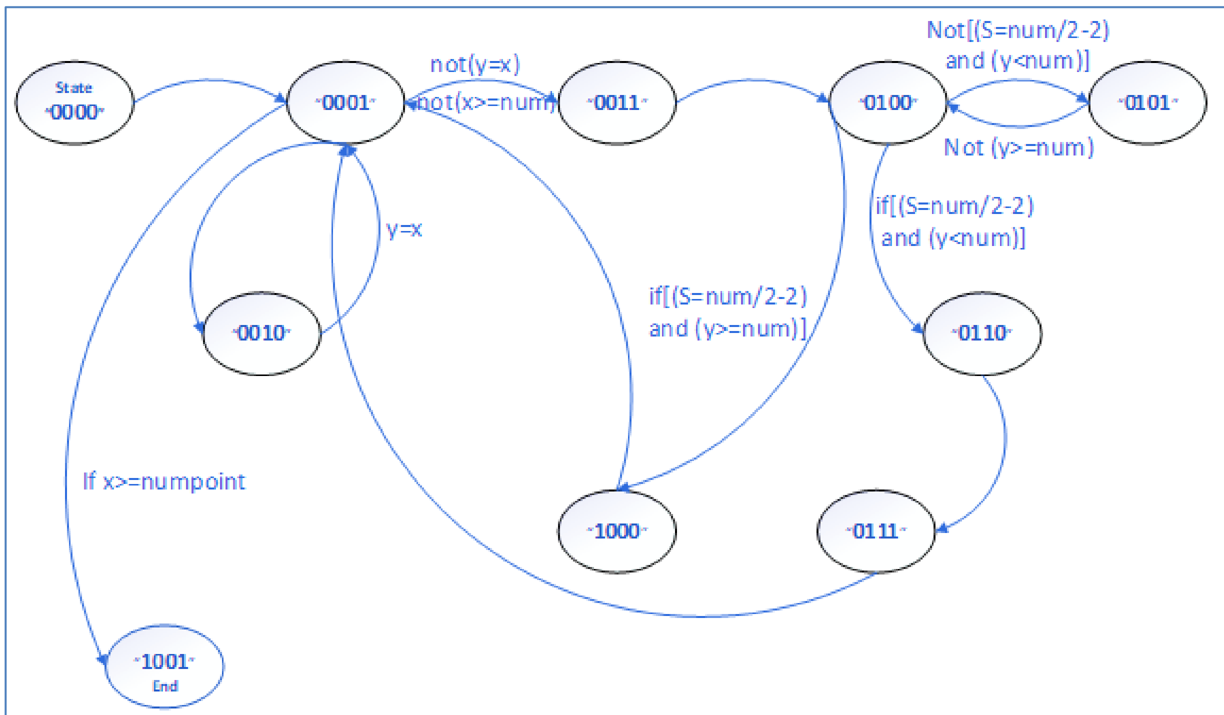


Σχήμα 44 – points_ram.vhd

FSM



Σχήμα 45 – FSM rtl



Σχήμα 46 – Διαδικασία FSM

FSM Αλγόριθμος

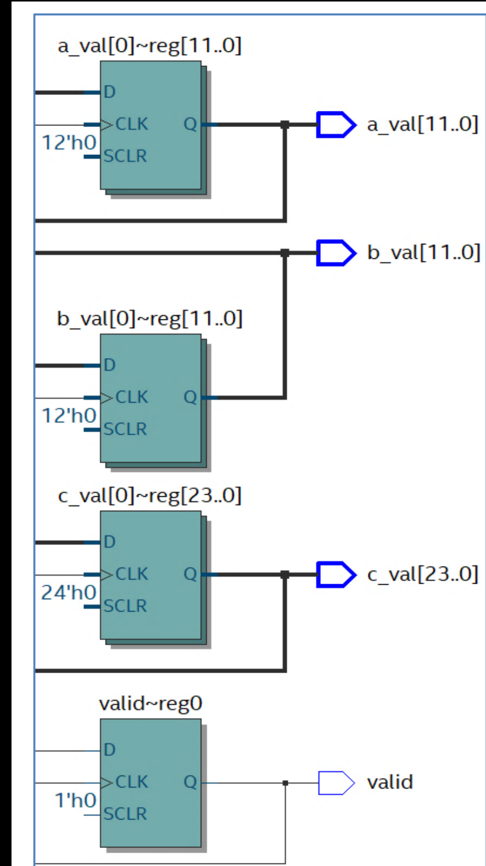
Πίνακας 18 – Ψεοδοκώδικας FSM

```
for [x = 0 to (points_num-1)]
{

for [y = x + 1 to (points_num-1)]
{
    if (x >= points_num)
    {
        finish;
    }

    else

    {   Param (x, y), m=0, n=1
    for [s = 0 to points_num/2]
    {
        dist (m, n)
        comp (m, n)
        m = m + 1, n = n + 1;
    }
        Line_comp (x, y)
    end if;
}
}
```



Σχήμα 47 – FSM output

FSM Περιγραφή [Παράρτημα 146]

- Κατάσταση "0000"

Επαναφορά όλων των σημάτων enable στο 0.

- Κατάσταση "0001"

Λαμβάνουμε δεδομένα δύο σημείων από την μνήμη και τα διαθέτουμε στην λειτουργική μονάδα parameters για εξαγωγή των a, b, c. Οι δύο αριθμοί έχουν διευθύνσεις αποθηκευμένες σε δύο μεταβλητές x και y, αρχικά μηδενικές και αυξανόμενες κατά την διαδικασία της FSM.
Επανερχόμαστε σε αυτήν την κατάσταση πολλές φορές κατά τον υπολογισμό των παραμέτρων κάθε νέας γραμμής, οπότε θα πρέπει να ελέγξουμε ότι το x εισόδου δεν είναι ίσο ή μεγαλύτερο από τον αριθμό των σημείων (num-1), οπότε και θα πρέπει να τερματίσουμε το σύστημα. Αυτός ο έλεγχος γίνεται με την ακόλουθη if συνθήκη (Σχήμα 48). Εάν είναι ψευδής, μπορούμε να περάσουμε τα x και y στην μνήμη και να συνεχίσουμε.

```
            if ((x = points_num - 1) then

                P0_addr     <= 0;
                P1_addr     <= 0;
                nextstate   <= "1001";

            else

                P0_addr     <= x;
                P1_addr     <= y;
                nextstate   <= "0011";
                y           <= y;
                line_num    <= line_num;
            end if;
```

Σχήμα 48 – Έλεγχοι "0001" κατάστασης

Αν το x είναι μεγαλύτερο από ή ίσο με τον αριθμό των σημείων, τότε θα πρέπει να φτιάξουμε μηδενικά P0_add και P1_addr και να πάμε στην κατάσταση "1001" που είναι κατάσταση τερματισμού, δλδ. ο αλγόριθμος έχει φτάσει στο τέλος του.
Διαφορετικά, εάν τα x και y είναι ίσα, θα πρέπει απλώς να αυξήσουμε το y κατά ένα για να έχουμε το επόμενο σημείο και να παρέχουμε μηδέν στο P0_add και P1_addr, καθώς δεν χρειάζεται να έχουμε τιμές από την μνήμη μέχρι να έχουμε έγκυρες διευθύνσεις x και y. Συνεχίζουμε στην κατάσταση "0010" που με τη σειρά της μας επιστρέφει ξανά στην "0001" για να ορίσουμε τη νέα τιμή του y.
Τέλος, εάν δεν υφίστανται και οι δύο προηγούμενες περιπτώσεις, περνάμε το x στην P0_addr και το y στην P1_addr και πηγαίνουμε στην επόμενη κατάσταση "0011", που είναι η κατάσταση υπολογισμού των αποστάσεων.

- Κατάσταση "0010"

Αυτή η κατάσταση χρησιμοποιείται ως κατάσταση βρόχου που μας στέλνει στην προηγούμενη κατάσταση που είναι η "0001".

➢ Κατάσταση "0011"

Σε αυτή την κατάσταση χρειαζόμαστε τις παραμέτρους a, b, c, οπότε θέτουμε το σήμα param_en σε 1 και ζητάμε από την μνήμη να μας παρέχει τις τιμές στις διευθύνσεις m και n, ώστε να είμαστε έτοιμοι για τον υπολογισμό απόστασης στην επόμενη κατάσταση. Αυτά (m και n) ξεκινούν από 0 και 1 αντίστοιχα και αυξάνονται κατά δύο κάθε φορά, οπότε χρησιμοποιώντας τα παίρνουμε δύο σημεία από την points_ram κάθε φορά. Στο τέλος της κατάστασης αυξάνουμε κατά δύο τις τιμές των m και n και πηγαίνουμε στην "0100".

➢ Κατάσταση "0100"

Σε αυτήν την κατάσταση κάνουμε υπολογισμούς απόστασης και συγκρίσεις, και έτσι τα dist_en και comp_en είναι στη θέση high.
Αυξάνουμε τα m και n κατά δύο.
s είναι ο αριθμός των υπολογισμένων σημείων και αυξάνεται κατά ένα κάθε φορά.
Τέλος, η επόμενη κατάσταση (βλ. σχήμα 49) αποφασίζεται από την ακόλουθη συνθήκη if:

```
if ((s = (points_num/2)- 2) and (y < points_num - 1)) then
    nextstate <= "0110";
elsif ((s = (points_num/2)- 2) and (y >= points_num - 1)) then
    nextstate <= "1000";
else
    nextstate <= "0101";
end if;
```

Σχήμα 49 – Μετάβαση στην επόμενη κατάσταση

Εάν φτάσαμε στο s = (αριθμός σημείων / 2) - 2 που συμβαίνει όταν υπολογίζουμε τις δύο τελευταίες αποστάσεις των δύο τελευταίων σημείων με την πρόσφατη γραμμή. Εξετάζουμε την τιμή του y, εάν είναι μικρότερη από τον αριθμό των σημείων, μπορούμε να πάμε στην κατάσταση "0110", να αυξήσουμε το y κατά ένα και στη συνέχεια να πάμε στο πρώτο μέρος του βρόχου.
Εάν είναι μεγαλύτερη ή ίση με τον αριθμό των σημείων, τότε πηγαίνουμε στην κατάσταση "1000", αυξάνουμε το x κατά ένα, μηδενίζουμε το y και αρχίζουμε ξανά τον βρόχο.
Διαφορετικά, απλώς πηγαίνουμε στην κατάσταση "0101".

➢ Κατάσταση "0101"

Αυτή είναι μια κατάσταση βρόχου για την κατάσταση "0100", όπου υπολογίζονται δύο νέες αποστάσεις μεταξύ της γραμμής μας και δύο νέων σημείων.

➢ Κατάσταση "0110" & "0111"

Αρχικά πηγαίνουμε στην κατάσταση "0110" όταν το πρώτο μέρος της if είναι true και συνεχίζουμε απλώς τη σύγκριση των τελευταίων υπολογισμένων αποστάσεων και στη συνέχεια στην "0111", όπου αυξάνουμε το y κατά ένα και επαναφέρουμε το s => 0, m => 0, n => 1, και τελικώς στην κατάσταση "0001" για να ξεκινήσει ξανά ο βρόχος με μια νέα γραμμή.

➢ Κατάσταση "0100"

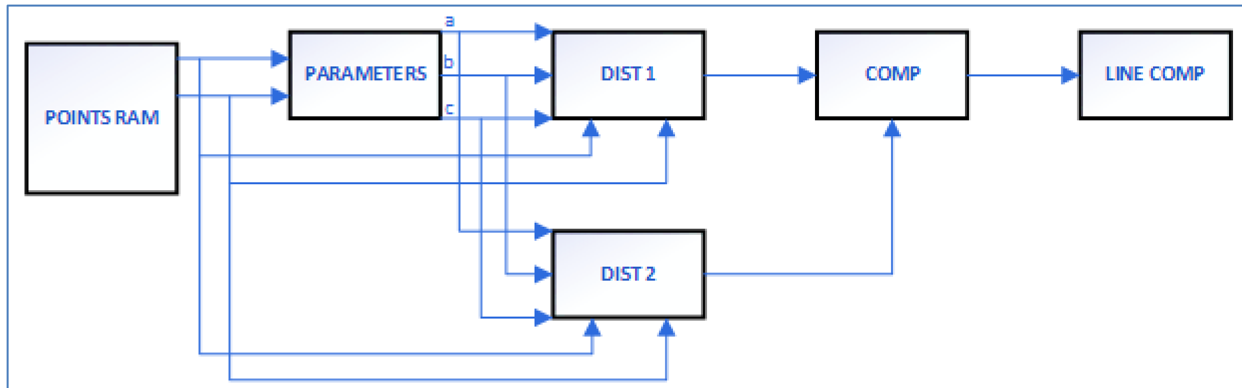
Αρχικά πηγαίνουμε στην κατάσταση "1000" όταν το δεύτερο μέρος του προηγούμενου if είναι true. Και εδώ κάνουμε το ίδιο πράγμα με την "0111", μόνο που επαναφέρουμε το y στο μηδέν και αυξάνουμε το x κατά ένα και στη συνέχεια πηγαίνουμε στην κατάσταση "0001" για να ξεκινήσει ξανά ο βρόχος με μια νέα γραμμή.

➢ Κατάσταση "1001"

Αυτή είναι η κατάσταση τερματισμού όπου απλώς κάνουμε το σήμα computation_done high.

Main [Παράρτημα 143]

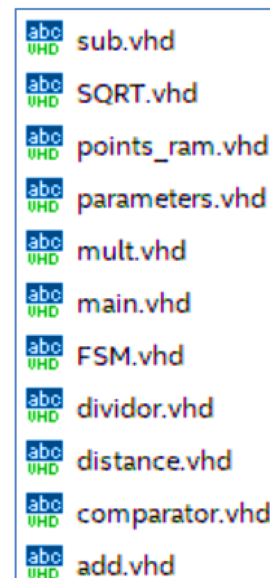
Αυτή είναι η κύρια λειτουργική μονάδα (βλ. σχήμα 50) του συστήματος με όλα τα components που έχουν αρχικοποιηθεί και συνδεθεί μεταξύ τους.



Σχήμα 50 – Διάγραμμα main multi clock

5.2 Μέθοδος 2 - Παράλληλη αρχιτεκτονική σχεδίαση (ασύγχρονη)

Η φιλοσοφία σχεδίασης για την αρχιτεκτονική αυτή είναι η εξής: Μέσω ενός αρχείου – πακέτου ανάγνωσης (ram_pkg), όπου αποθηκεύονται όλες οι συντεταγμένες των σημείων σε μορφή array, εισάγουμε στις παραμέτρους (parameters), τις συντεταγμένες δύο σημείων και υπολογίζουμε τα a, b, c της ευθείας που συνθέτουν. Στη συνέχεια μέσω αριθμού distance modules ίσου με τον αριθμό των σημείων, υπολογίζουμε τις αποστάσεις όλων των σημείων από την ευθεία. Εισάγουμε τις αποστάσεις ανα δύο στο comparator module και σε περίπτωση που κάποια εξ΄αυτών είναι μικρότερη από την τιμή κατωφλίου θεωρούμε το σημείο ως inlier. Τελικώς μέσω ενός μετρητή εξάγουμε τον αριθμό των inliers για κάθε ευθεία.
Αναλυτικά τα διάφορα modules, που εμφανίζουν διαφορές ως προς την προηγούμενη μέθοδο περιγράφονται παρακάτω:



Comparator [Παράρτημα 148]
Λαμβάνονται οι αποστάσεις εως δύο κάθε φορά, ελέγχεται εάν οποιαδήποτε από τις δύο αποστάσεις είναι μικρότερη από την απόσταση κατωφλίου (thr_dis), που σημαίνει ότι πρόκειται για inlier, και όταν ανιχνευθεί η απόσταση 1 ότι είναι από inlier σημείο, παίρνουμε σήμα εξόδου IN_1_val υψηλό και IN_2_val για την απόσταση 2. Καθώς χρησιμοποιούμε παράλληλο υπολογισμό, αυτός ο συγκριτής θα έχει μόνο ένα ζεύγος αποστάσεων κάθε φορά.

Points_ram [Παράρτημα 149]
Αυτή η ενότητα έχει σχεδιαστεί για να διαβάζει δύο αρχεία, καθένα από αυτά περιέχει τις τιμές των x και y για τα σημεία μας και μας παρέχει δύο εξόδους τύπου RamType που είναι ένας τύπος που ορίζεται σε ένα πακέτο που ονομάζεται ram_pkg στο κύριο αρχείο.
Τελικά θα πάρουμε x_array και y_array για τις τιμές των x και y.

FSM [Παράρτημα 150]

- Κατάσταση "0000"

Επαναφορά όλων των enables στο 0.

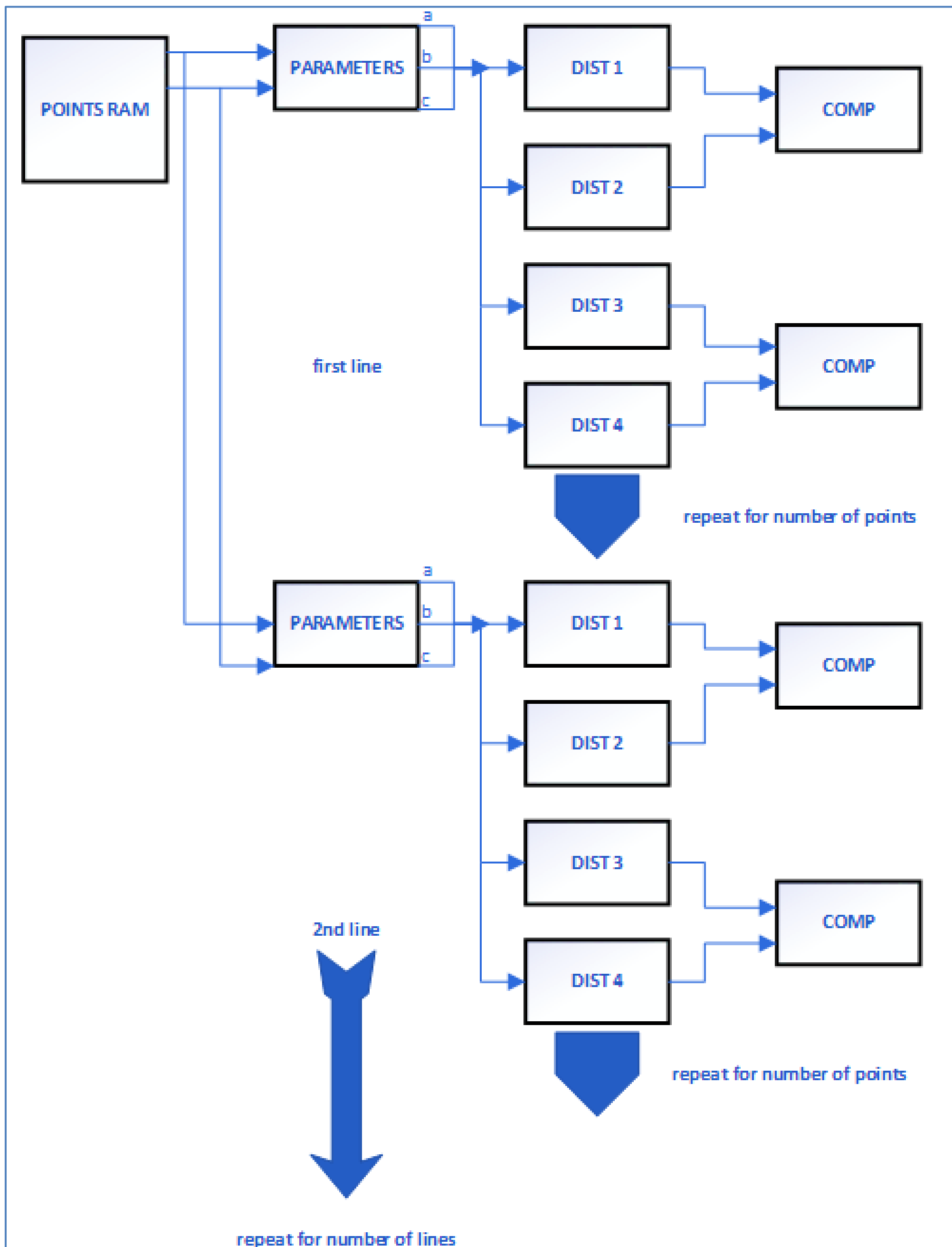
- Κατάσταση "0001" και "0010"

Χρησιμοποιούνται για την ενεργοποίηση των parameter, distance και comparing modules.

- Κατάσταση "0011"

Σε αυτή την κατάσταση γίνεται η καταμέτρηση των inliers για κάθε γραμμή, και τελικά επιλέγετα η best line fit.

Main [Παράρτημα 153]
Αυτή είναι η κύρια λειτουργική μονάδα (βλ. σχήμα 51) του συστήματος με όλα τα components που έχουν αρχικοποιηθεί και συνδεθεί μεταξύ τους.



Σχήμα 51 – Διάγραμμα main one clock

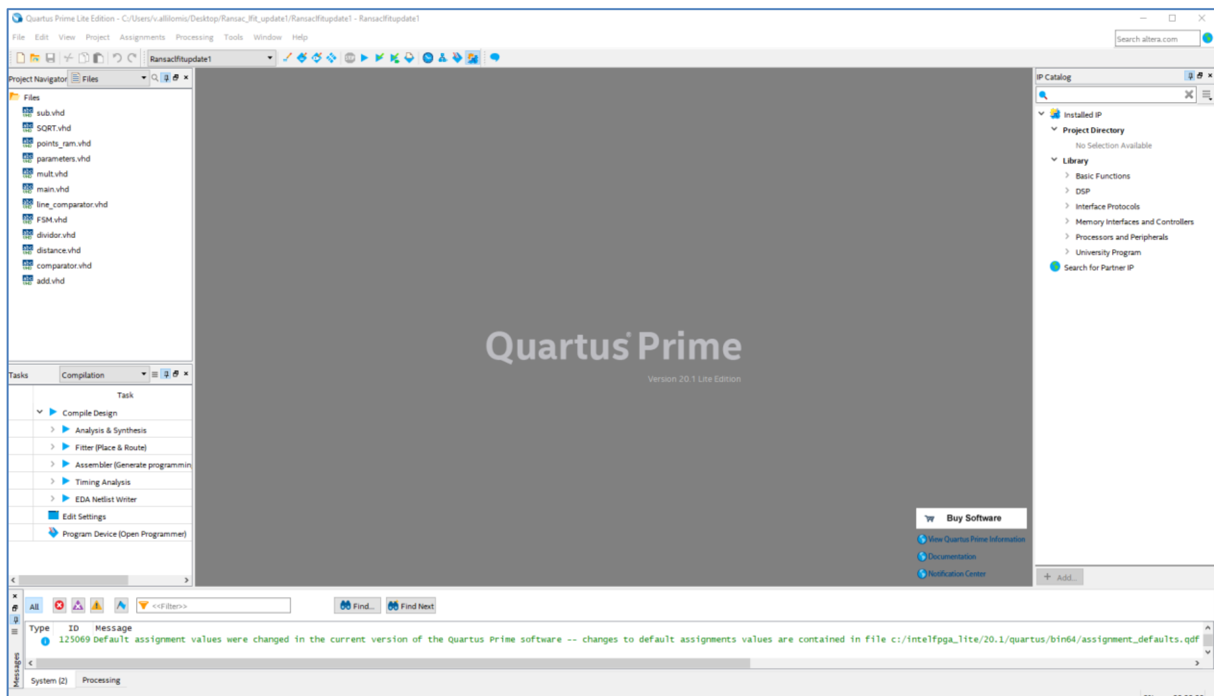
5.3 Αποτελέσματα και Ανάλυση

α. Διαδικασία Προσομοίωσης

Και για τις δύο μεθόδους επιλέγονται τα ίδια 16, 32, 64, 128 σημεία στο χώρο. Τα βήματα που ακολουθήθηκαν κατά την προσομοίωση είναι τα εξής:

Μέθοδος 1 & 2

- Compilation Quartus (σχήμα 52) για Μέθοδο 1 & 2.
- Test-bench Modelsim για τέσσερις (4) τιμές κατωφλίου - threshold (thr_dis) με εξαγωγή των παραμέτρων a, b, c της εξίσωσης ευθείας, αριθμού inliers και άθροισμα απόστασης αυτών από την ευθεία.
- Γραφική παράσταση των ευθειών για βέλτιστη τιμή κατωφλίου.
- Επαλήθευση μέσω Matlab (ενδεικτικά) και σύγκριση των αποτελεσμάτων.



Σχήμα 52 – Quartus Prime

b. Αποτελέσματα Προσομοίωσης

Για την προσομοίωση χρησιμοποιήθηκαν τα προγράμματα Quartus και Modelsim, σε επεξεργαστή Intel quad-core i7-7700 CPU 3,60 GHz.

Τα αποτελέσματα σχετικά με τους δεσμευμένους πόρους (Quartus) και την αναλογία χρόνου (Moselsim) που απαιτήθηκε, παρατίθενται παρακάτω:

Πίνακας - Δεσμευμένοι πόροι & αναλογία χρόνου για multi / one clock υπολογισμούς

	QUARTUS COMPILATION					
ΜΕΘΟΔΟΣ / Α/Α	Option	ALMs	Total registers	Total pins	Total DSP Blocks	Total time (min)
1	multi clock	1.575	253	63	5	21:06
2	one clock	161.137	6.179	99	476	01:09:29

	ΧΡΟΝΟΣ ΠΡΟΣΟΜΟΙΩΣΗΣ ΣΤΟ MODELSIM (ps)	
ΣΗΜΕΙΑ / Α/Α	ΜΕΘΟΔΟΣ 1	ΜΕΘΟΔΟΣ 2
16	20.000.000	85.000
32	163.000.000	85.000
64	1.310.000.000	85.000
128	10.483.000.000	Ø

Fitter Summary

<<Filter>>

Fitter Status	Successful - Sat May 22 07:34:01 2021
Quartus Prime Version	18.1.0 Build 222 09/21/2018 SJ Pro Edition
Revision Name	Ransaclfitupdate1
Top-level Entity Name	main
Family	Stratix 10
Device	1SG280LN2F43E1VG
Timing Models	Final
Logic utilization (in ALMs)	1,575 / 933,120 (< 1 %)
Total dedicated logic registers	253
Total pins	63 / 912 (7 %)
Total block memory bits	0 / 240,046,080 (0 %)
Total RAM Blocks	0 / 11,721 (0 %)
Total DSP Blocks	5 / 5,760 (< 1 %)
Total HSSI RX channels	0 / 48 (0 %)
Total HSSI TX channels	0 / 48 (0 %)
Total PLLs	0 / 88 (0 %)

QUARTUS COMPILATION

ΜΕΘΟΔΟΣ 1

Fitter Summary

<<Filter>>

Fitter Status	Successful - Sat May 22 10:29:14 2021
Quartus Prime Version	18.1.0 Build 222 09/21/2018 SJ Pro Edition
Revision Name	Ransaclfitupdate2
Top-level Entity Name	main
Family	Stratix 10
Device	1SG280LN2F43E1VG
Timing Models	Final
Logic utilization (in ALMs)	161,137 / 933,120 (17 %)
Total dedicated logic registers	6179
Total pins	99 / 912 (11 %)
Total block memory bits	0 / 240,046,080 (0 %)
Total RAM Blocks	0 / 11,721 (0 %)
Total DSP Blocks	476 / 5,760 (8 %)
Total HSSI RX channels	0 / 48 (0 %)
Total HSSI TX channels	0 / 48 (0 %)
Total PLLs	0 / 88 (0 %)

QUARTUS COMPILATION

ΜΕΘΟΔΟΣ 2

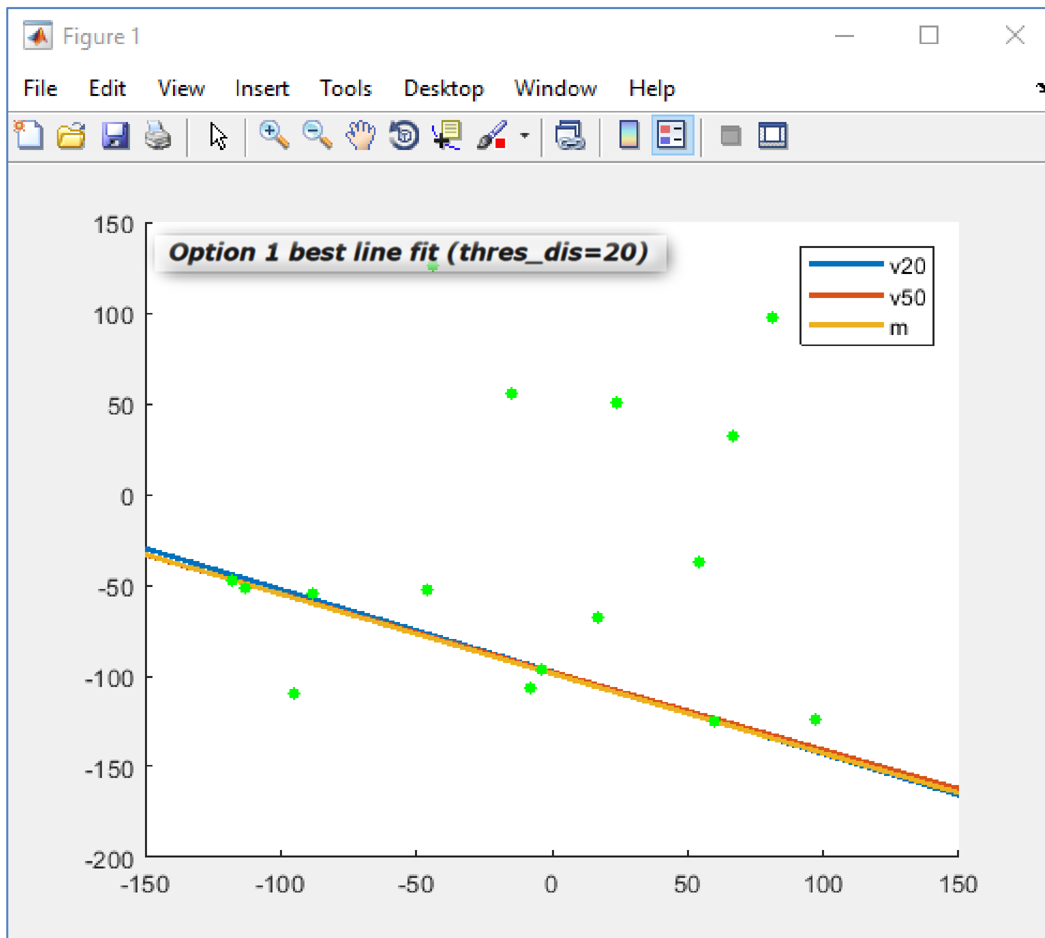
A/A	x	y	xdec2bin	ydec2bin	x12bit	y12bit
1	67	32	01000011	00100000	010000110000	001000000000
2	81	97	01010001	01100001	010100010000	011000010000
3	-4	-96	11111100	10100000	111111000000	101000000000
4	60	-125	00111100	10000011	001111000000	100000110000
5	97	-124	01100001	10000100	011000010000	100001000000
6	-15	56	11110001	00111000	111100010000	001110000000
7	-8	-107	11111000	10010101	111110000000	100101010000
8	54	-37	00110110	11011011	001101100000	110110110000
9	-88	-55	10101000	11001001	101010000000	110010010000
10	17	-68	00010001	10111100	000100010000	101111000000
11	-118	-47	10001010	11010001	100010100000	110100010000
12	-95	-110	10100001	10010010	101000010000	100100100000
13	-46	-53	11010010	11001011	110100100000	110010110000
14	-44	126	11010100	01111110	110101000000	011111100000
15	-113	-52	10001111	11001100	100011110000	110011000000
16	24	51	00011000	00110011	000110000000	001100110000

	Option 1 / 2 VHDL							
max_a	-70	-10	100	16	29	44	-49	-15
max_b	-29	-84	-50	-126	64	109	-114	-89
max_c	574	-5128	-5100	-4744	6248	10640	-11140	-6323
max_Inlier_num	4	5	5	5	7	7	11	10
max_Dis	3	Ø	12	Ø	42	Ø	205	Ø
thres_dis	5		10		20		50	

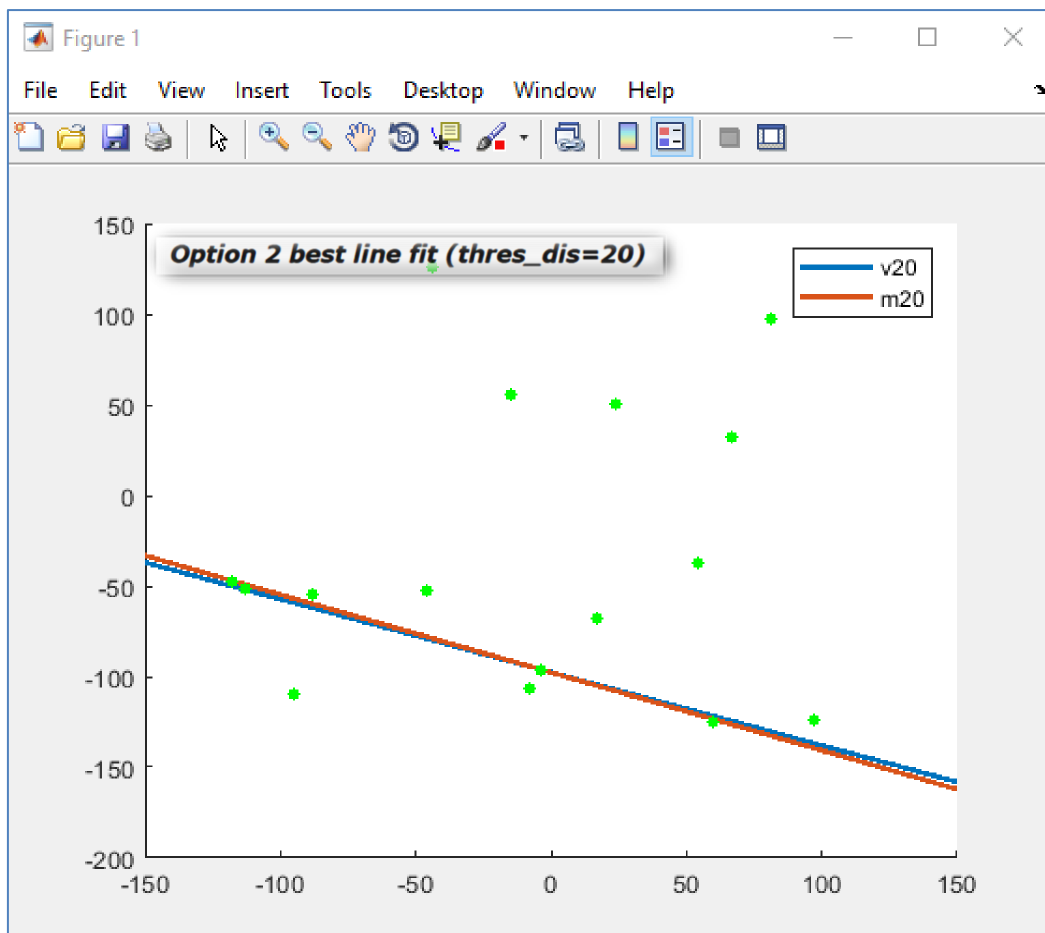
	MATLAB			
max_a	-78	-73	-70	-49
max_b	-178	-173	-148	-114
max_c	-17570	-17425	-14300	-11140
max_Inlier_num	5	6	8	10
max_Dis	8	21	64	140
thres_dis	5	10	20	50

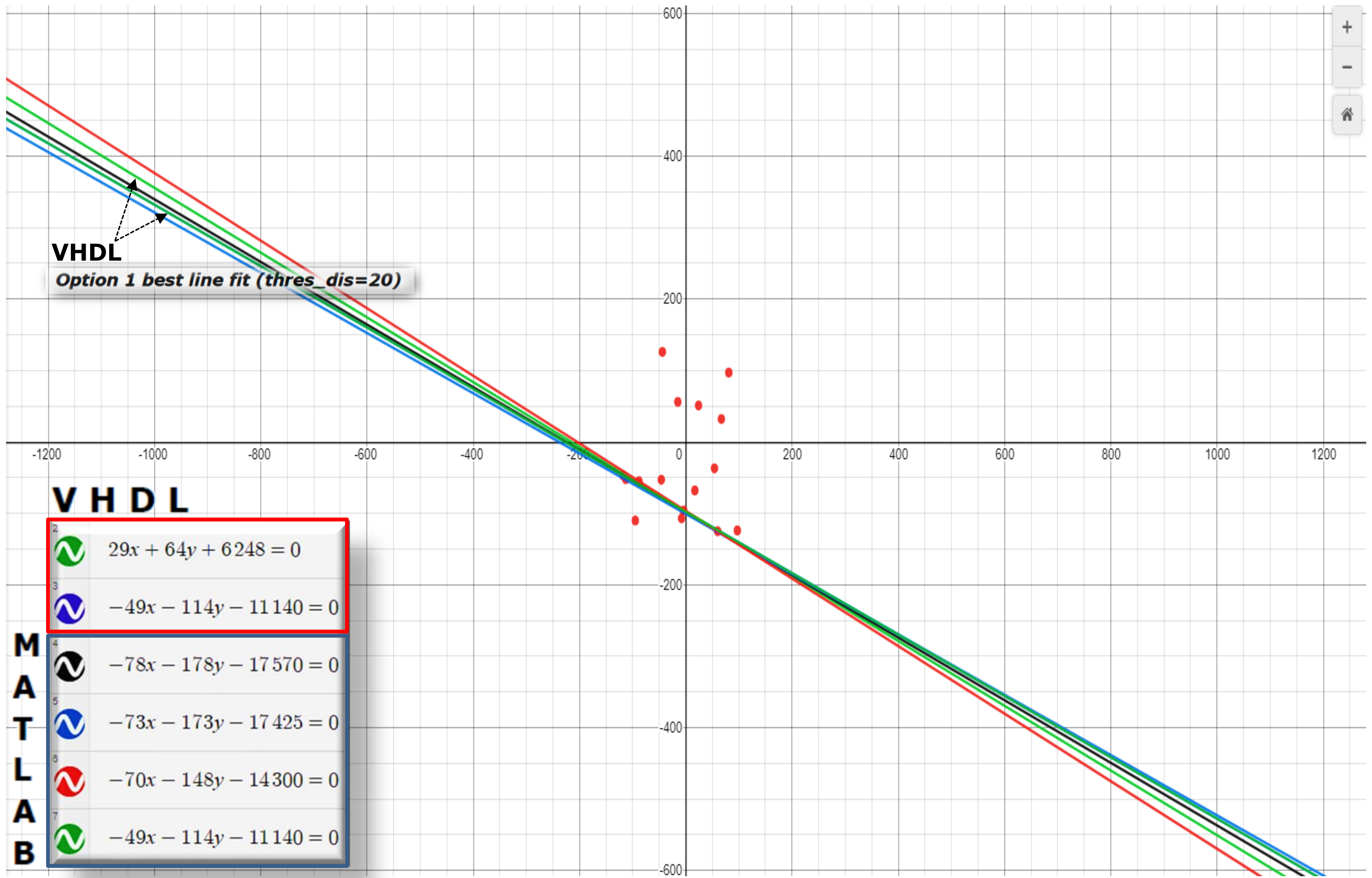
a/a	max_i	max_a	max_b	max_c	sum_dis
1	7	49	114	11140	34.33
2	7	78	178	17570	34.73
3	7	44	109	10640	35.82
4	7	73	173	17245	36.61
5	7	29	64	6260	37.86
6	7	72	210	19056	45.57
7	7	41	84	8228	47.03
8	7	69	185	16247	48.39
9	8	70	148	14300	64.01
10	8	77	215	19191	64.74
11	7	8	30	2354	67.77
12	7	28	101	9808	69.63

Matlab sort thres_dis 20

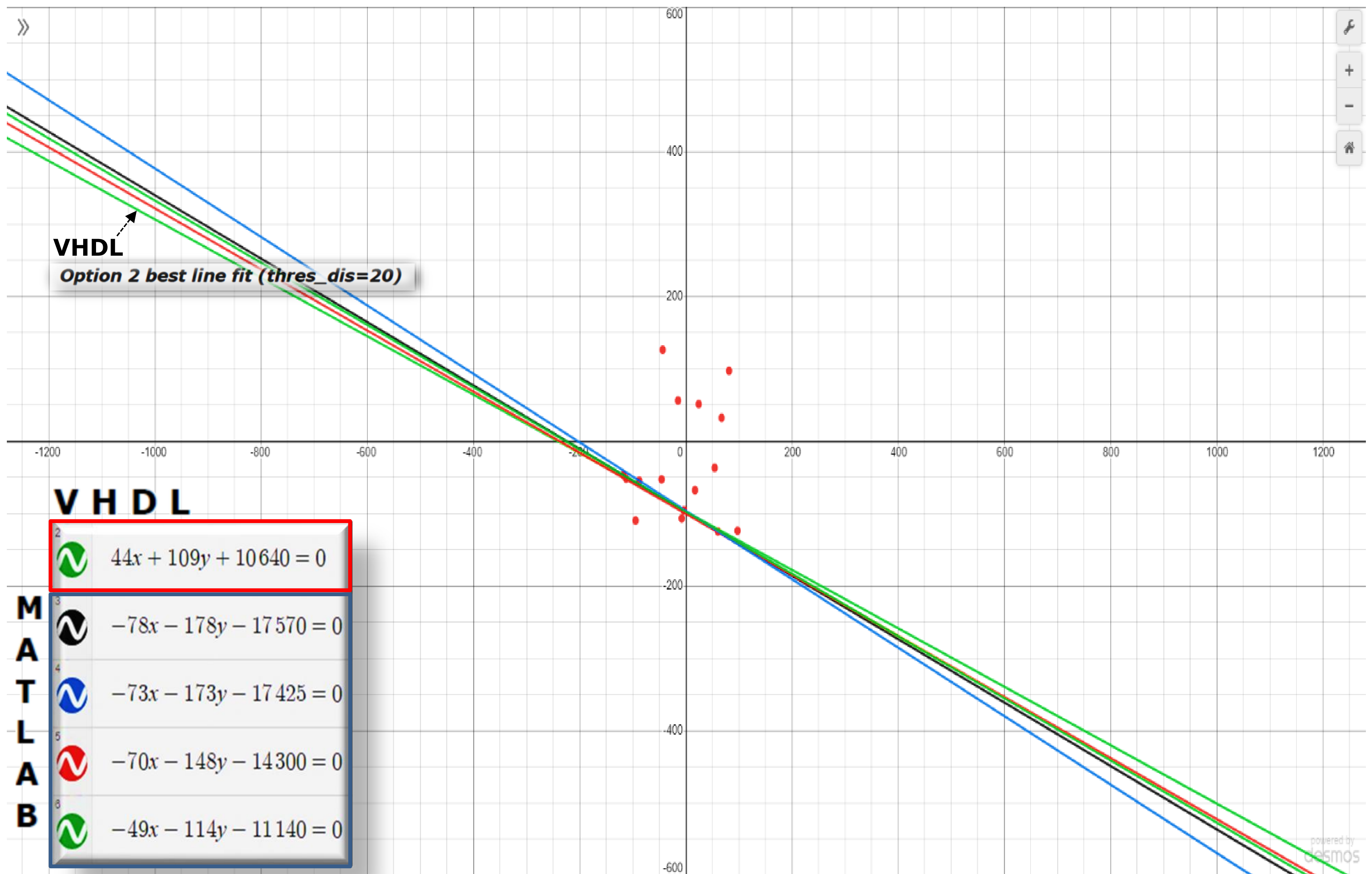


best line fit





Εικόνα - Γραφική παράσταση Option 1 best line fit (thr_dis 20,50) + Matlab (thr_dis 5-50)



Εικόνα - Γραφική παράσταση Option 2 best line fit (thr_dis 20) + Matlab (thr_dis 5-50)

A/A	x	y		xdec2bin	ydec2bin		x12bit	y12bit
1	16	33		00010000	00100001		000100000000	001000010000
2	18	122		00010010	01111010		000100100000	011110100000
3	49	117		00110001	01110101		001100010000	011101010000
4	-74	106		10110110	01101010		101101100000	011010100000
5	3	-75		00000011	10110101		000000110000	101101010000
6	-92	56		10100100	00111000		101001000000	001110000000
7	-119	-119		10001001	10001001		100010010000	100010010000
8	63	36		00111111	00100100		001111110000	001001000000
9	66	-58		01000010	11000110		010000100000	110001100000
10	-84	33		10101100	00100001		101011000000	001000010000
11	60	-104		00111100	10011000		001111000000	100110000000
12	-66	118		10111110	01110110		101111100000	011101100000
13	104	33		01101000	00100001		011010000000	001000010000
14	44	110		00101100	01101110		001011000000	011011100000
15	-89	27		10100111	00011011		101001110000	000110110000
16	30	-80		00011110	10110000		000111100000	101100000000
17	23	62		00010111	00111110		000101110000	001111100000
18	-29	51		11100011	00110011		111000110000	001100110000
19	-109	55		10010011	00110111		100100110000	001101110000
20	34	52		00100010	00110100		001000100000	001101000000
21	-55	-85		11001001	10101011		110010010000	101010110000
22	47	-124		00101111	10000100		001011110000	100001000000
23	43	-22		00101011	11101010		001010110000	111010100000
24	-10	-40		11110110	11011000		111101100000	110110000000
25	-18	-113		11101110	10001111		111011100000	100011110000
26	-20	64		11101100	01000000		111011000000	010000000000
27	45	6		00101101	00000110		001011010000	000001100000
28	-114	-65		10001110	10111111		100011100000	101111110000
29	33	29		00100001	00011101		001000010000	000111010000
30	73	-95		01001001	10100001		010010010000	101000010000
31	125	-64		01111101	11000000		011111010000	110000000000
32	106	-93		01101010	10100011		011010100000	101000110000

Option 1 / 2 VHDL								
max_a	-64	64	124	-70	37	-37	112	112
max_b	-70	21	40	-16	7	-7	-25	121
max_c	-3440	-3006	-5252	3212	-2036	2036	-5366	-619
max_Inlier_num	8	6	9	9	15	12	21	19
max_Dis	18	Ø	31	Ø	166	Ø	505	Ø
thres_dis	5		10		20		50	

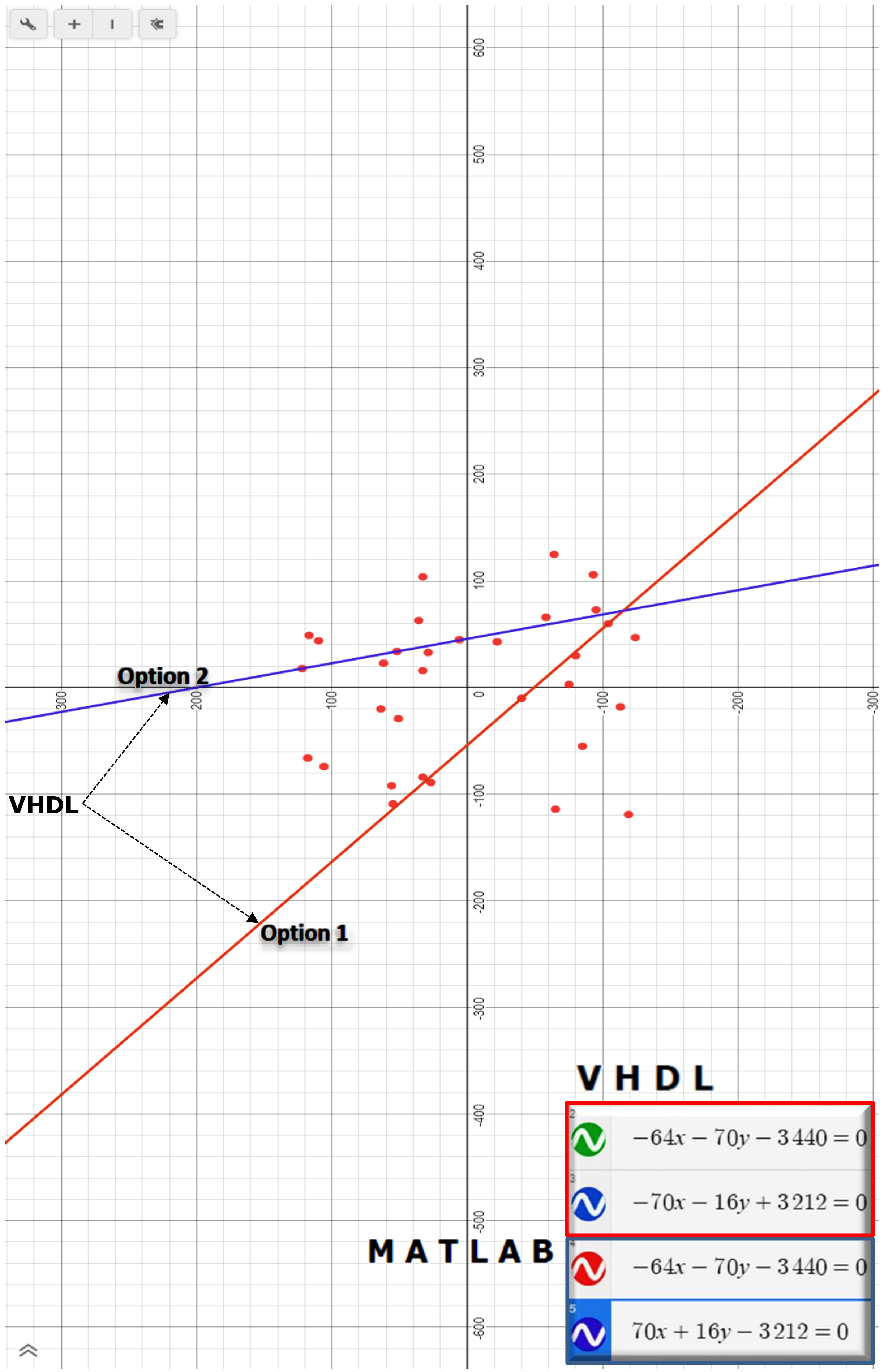
Εικόνα - Παράμετροι a, b, c ---> VHDL

MATLAB				
max_a	-64	70	176	-142
max_b	-70	16	13	-7
max_c	-3440	-3212	-6660	3700
max_Inlier_num	6	9	12	19
max_Dis	14	44	116	435
thres_dis	5	10	20	50

Εικόνα - Παράμετροι a, b, c ---> Matlab

a/a	max_i	max_a	max_b	max_c	sum_dis
1	6	64	70	3440	13.64
2	6	156	26	-6656	15.62
3	6	64	21	-3006	16.45

Matlab sort thres_dis 5



Εικόνα - Γραφική παράσταση Option 1 & 2 best line fit (thr_dis 5,10) + Matlab (thr_dis 5,10)

A/A	x	y		xdec2bin	ydec2bin		x12bit	y12bit
1	16	33		00010000	00100001		000100000000	001000010000
2	18	122		00010010	01111010		000100100000	011110100000
3	49	117		00110001	01110101		001100010000	011101010000
4	-74	106		10110110	01101010		101101100000	011010100000
5	3	-75		00000011	10110101		000000110000	101101010000
6	-92	56		10100100	00111000		101001000000	001110000000
7	-119	-119		10001001	10001001		100010010000	100010010000
8	63	36		00111111	00100100		001111110000	001001000000
9	66	-58		01000010	11000110		010000100000	110001100000
10	-84	33		10101100	00100001		101011000000	001000010000
11	60	-104		00111100	10011000		001111000000	100110000000
12	-66	118		10111110	01110110		101111100000	011101100000
13	104	33		01101000	00100001		011010000000	001000010000
14	44	110		00101100	01101110		001011000000	011011100000
15	-89	27		10100111	00011011		101001110000	000110110000
16	30	-80		00011110	10110000		000111100000	101100000000
17	23	62		00010111	00111110		000101110000	001111100000
18	-29	51		11100011	00110011		111000110000	001100110000
19	-109	55		10010011	00110111		100100110000	001101110000
20	34	52		00100010	00110100		001000100000	001101000000
21	-55	-85		11001001	10101011		110010010000	101010110000
22	47	-124		00101111	10000100		001011110000	100001000000
23	43	-22		00101011	11101010		001010110000	111010100000
24	-10	-40		11110110	11011000		111101100000	110110000000
25	-18	-113		11101110	10001111		111011100000	100011110000
26	-20	64		11101100	01000000		111011000000	010000000000
27	45	6		00101101	00000110		001011010000	000001100000
28	-114	-65		10001110	10111111		100011100000	101111110000
29	33	29		00100001	00011101		001000010000	000111010000
30	73	-95		01001001	10100001		010010010000	101000010000
31	125	-64		01111101	11000000		011111010000	110000000000
32	106	-93		01101010	10100011		011010100000	101000110000
33	3	79		00000011	01001111		000000110000	010011110000
34	25	-117		00011001	10001011		000110010000	100010110000
35	-40	123		11011000	01111011		110110000000	011110110000
36	92	-26		01011100	11100110		010111000000	111001100000
37	-56	23		11001000	00010111		110010000000	000101110000
38	91	19		01011011	00010011		010110110000	000100110000
39	-15	-126		11110001	10000010		111100010000	100000100000
40	-83	111		10101101	01101111		101011010000	011011110000
41	20	-50		00010100	11001110		000101000000	110011100000
42	100	-47		01100100	11010001		011001000000	110100010000
43	68	35		01000100	00100011		010001000000	001000110000
44	36	-108		00100100	10010100		001001000000	100101000000
45	122	107		01111010	01101011		011110100000	011010110000
46	97	-71		01100001	10111001		011000010000	101110010000
47	-115	-47		10001101	11010001		100011010000	110100010000

48	37	-56		00100101	11001000		001001010000	110010000000
49	-54	-7		11001010	11111001		110010100000	111110010000
50	-102	27		10011010	00011011		100110100000	000110110000
51	107	2		01101011	00000010		011010110000	000000100000
52	-25	75		11100111	01001011		111001110000	010010110000
53	-60	101		11000100	01100101		110001000000	011001010000
54	-20	-115		11101100	10001101		111011000000	100011010000
55	-48	35		11010000	00100011		110100000000	001000110000
56	64	15		01000000	00001111		010000000000	000011110000
57	19	-66		00010011	10111110		000100110000	101111100000
58	66	2		01000010	00000010		010000100000	000000100000
59	-4	-14		11111100	11110010		111111000000	111100100000
60	-73	-12		10110111	11110100		101101110000	111101000000
61	16	-59		00010000	11000101		000100000000	110001010000
62	-2	46		11111110	00101110		111111100000	001011100000
63	-12	84		11110100	01010100		111101000000	010101000000
64	96	27		01100000	00011011		011000000000	000110110000

	Option 1 / 2 VHDL							
max_a	40	45	95	33	83	86	-109	-60
max_b	47	20	105	57	-60	124	-95	-54
max_c	-2082	460	-4905	-4086	-4660	-7364	1684	1014
max_Inlier_num	10	9	14	14	21	21	39	37
max_Dis	20	Ø	52	Ø	185	Ø	1052	Ø
thres_dis	5		10		20		50	

Εικόνα - Παράμετροι a, b, c ---> VHDL

	MATLAB			
max_a	-48	-165	60	-181
max_b	-75	-185	89	-106
max_c	5724	8785	-5175	5798
max_Inlier_num	9	14	21	37
max_Dis	14	59	231	982
thres_dis	5	10	20	50

Εικόνα - Παράμετροι a, b, c ---> Matlab

a/a	max_i	max_a	max_b	max_c	sum_dis
1	9	48	75	-5724	14.39
2	9	26	40	-3078	15.34
3	9	110	51	1154	15.94
4	9	58	65	-3000	16.71
5	9	128	145	-6720	16.89
6	9	110	127	-5622	17.12
7	9	40	47	-2082	18.24

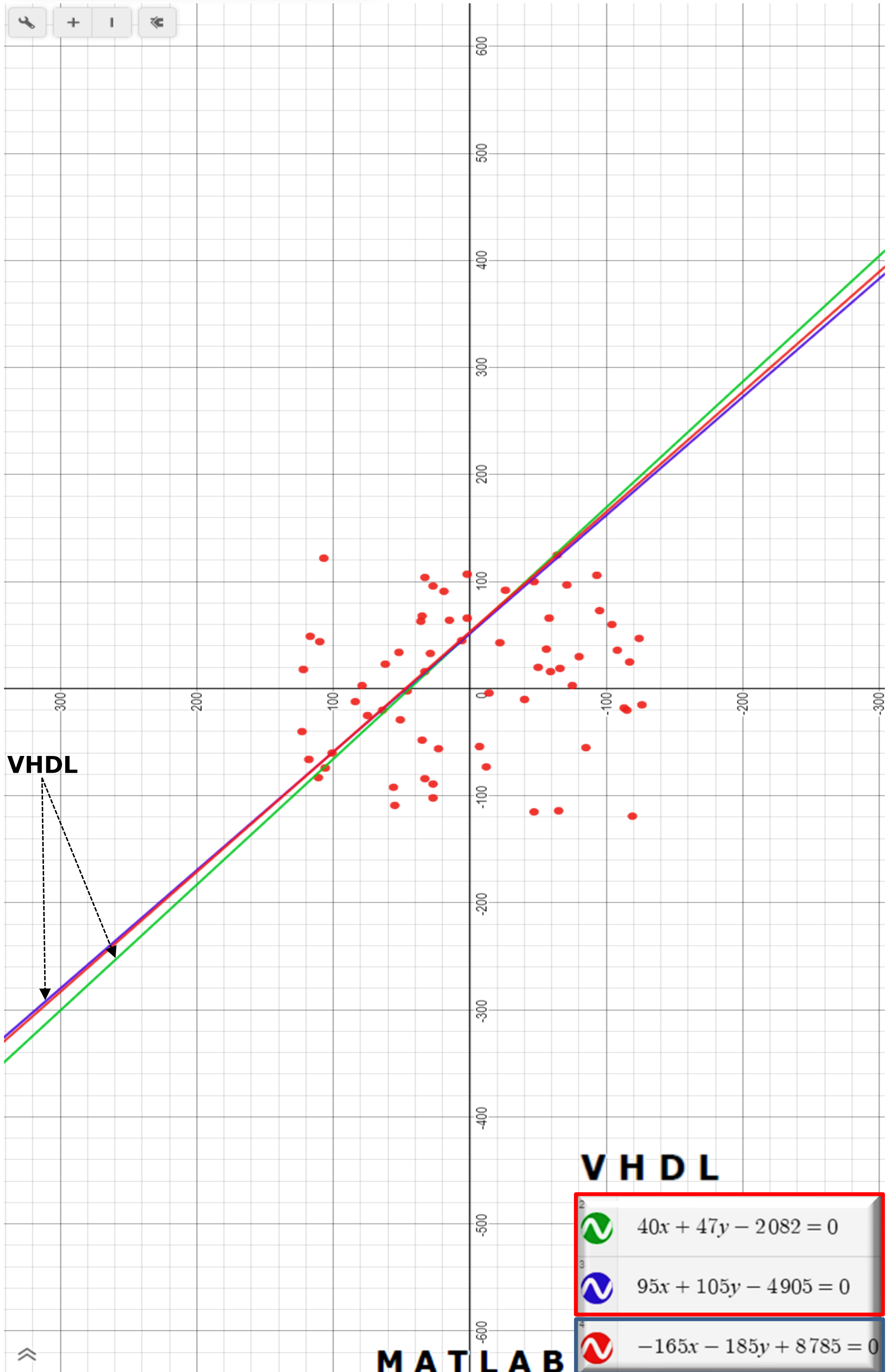
Matlab sort thres_dis 5

a/a	max_i	max_a	max_b	max_c	sum_dis
1	13	97	109	-5149	48.82
2	13	68	76	-3596	49.19
3	13	31	36	-1684	49.61
4	13	58	65	-3000	49.84
5	13	95	105	-4905	50.50
6	13	73	90	-4138	51.82
7	13	60	72	-3192	52.37
8	13	65	123	-8523	53.30
9	13	63	-38	-3160	53.74
10	13	84	179	-12897	55.76
11	13	74	156	-11316	56.46
12	13	79	170	-12174	56.61
13	13	65	81	-3596	56.80
14	13	39	83	-6043	57.74
15	13	127	152	-7732	57.89
16	13	35	73	-5331	58.59
17	14	165	185	-8785	59.12

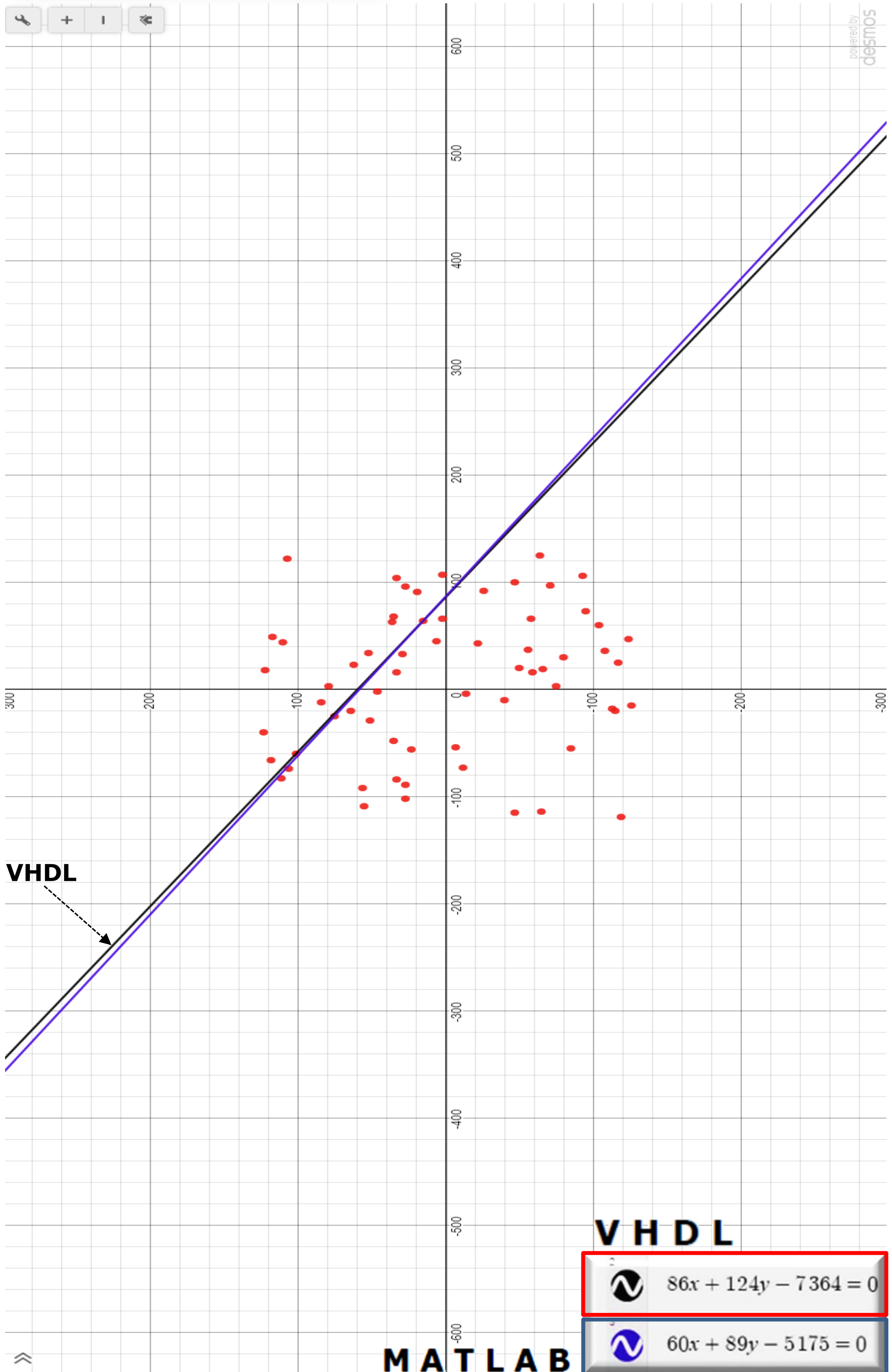
Matlab sort thres_dis 10

a/a	max_i	max_a	max_b	max_c	sum_dis
1	21	60	89	-5175	231.22
2	21	86	124	-7364	232.37
3	21	91	138	-7894	232.49
4	21	103	130	-8542	256.56

Matlab sort thres_dis 20



Εικόνα - Γραφική παράσταση Option 1 best line fit (thr_dis 5, 10) + Matlab (thr_dis 10)



Εικόνα - Γραφική παράσταση Option 2 best line fit (thr_dis 20) + Matlab (thr_dis 20)

A/A	x	y	xdec2bin	ydec2bin	x12bit	y12bit
1	16	33	00010000	00100001	000100000000	001000010000
2	18	122	00010010	01111010	000100100000	011110100000
3	49	117	00110001	01110101	001100010000	011101010000
4	-74	106	10110110	01101010	101101100000	011010100000
5	3	-75	00000011	10110101	000000110000	101101010000
6	-92	56	10100100	00111000	101001000000	001110000000
7	-119	-119	10001001	10001001	100010010000	100010010000
8	63	36	00111111	00100100	001111110000	001001000000
9	66	-58	01000010	11000110	010000100000	110001100000
10	-84	33	10101100	00100001	101011000000	001000010000
11	60	-104	00111100	10011000	001111000000	100110000000
12	-66	118	10111110	01110110	101111100000	011101100000
13	104	33	01101000	00100001	011010000000	001000010000
14	44	110	00101100	01101110	001011000000	011011100000
15	-89	27	10100111	00011011	101001110000	000110110000
16	30	-80	00011110	10110000	000111100000	101100000000
17	23	62	00010111	00111110	000101110000	001111100000
18	-29	51	11100011	00110011	111000110000	001100110000
19	-109	55	10010011	00110111	100100110000	001101110000
20	34	52	00100010	00110100	001000100000	001101000000
21	-55	-85	11001001	10101011	110010010000	101010110000
22	47	-124	00101111	10000100	001011110000	100001000000
23	43	-22	00101011	11101010	001010110000	111010100000
24	-10	-40	11110110	11011000	111101100000	110110000000
25	-18	-113	11101110	10001111	111011100000	100011110000
26	-20	64	11101100	01000000	111011000000	010000000000
27	45	6	00101101	00000110	001011010000	000001100000
28	-114	-65	10001110	10111111	100011100000	101111110000
29	33	29	00100001	00011101	001000010000	000111010000
30	73	-95	01001001	10100001	010010010000	101000010000
31	125	-64	01111101	11000000	011111010000	110000000000
32	106	-93	01101010	10100011	011010100000	101000110000
33	3	79	00000011	01001111	000000110000	010011110000
34	25	-117	00011001	10001011	000110010000	100010110000
35	-40	123	11011000	01111011	110110000000	011110110000
36	92	-26	01011100	11100110	010111000000	111001100000
37	-56	23	11001000	00010111	110010000000	000101110000
38	91	19	01011011	00010011	010110110000	000100110000
39	-15	-126	11110001	10000010	111100010000	100000100000
40	-83	111	10101101	01101111	101011010000	011011110000
41	20	-50	00010100	11001110	000101000000	110011100000
42	100	-47	01100100	11010001	011001000000	110100010000
43	68	35	01000100	00100011	010001000000	001000110000
44	36	-108	00100100	10010100	001001000000	100101000000
45	122	107	01111010	01101011	011110100000	011010110000
46	97	-71	01100001	10111001	011000010000	101110010000
47	-115	-47	10001101	11010001	100011010000	110100010000

48	37	-56		00100101	11001000		001001010000	110010000000
49	-54	-7		11001010	11111001		110010100000	111110010000
50	-102	27		10011010	00011011		100110100000	000110110000
51	107	2		01101011	00000010		011010110000	000000100000
52	-25	75		11100111	01001011		111001110000	010010110000
53	-60	101		11000100	01100101		110001000000	011001010000
54	-20	-115		11101100	10001101		111011000000	100011010000
55	-48	35		11010000	00100011		110100000000	001000110000
56	64	15		01000000	00001111		010000000000	000011110000
57	19	-66		00010011	10111110		000100110000	101111100000
58	66	2		01000010	00000010		010000100000	000000100000
59	-4	-14		11111100	11110010		111111000000	111100100000
60	-73	-12		10110111	11110100		101101110000	111101000000
61	16	-59		00010000	11000101		000100000000	110001010000
62	-2	46		11111110	00101110		111111100000	001011100000
63	-12	84		11110100	01010100		111101000000	010101000000
64	96	27		01100000	00011011		011000000000	000110110000
65	104	56		01101000	00111000		011010000000	001110000000
66	107	98		01101011	01100010		011010110000	011000100000
67	110	118		01101110	01110110		011011100000	011101100000
68	-126	-100		10000010	10011100		100000100000	100111000000
69	67	-28		01000011	11100100		010000110000	111001000000
70	-52	14		11001100	00001110		110011000000	000011100000
71	53	-45		00110101	11010011		001101010000	110100110000
72	62	38		00111110	00100110		001111100000	001001100000
73	-24	-49		11101000	11001111		111010000000	110011110000
74	-79	101		10110001	01100101		101100010000	011001010000
75	-119	59		10001001	00111011		100010010000	001110110000
76	1	-13		00000001	11110011		000000010000	111100110000
77	-112	-79		10010000	10110001		100100000000	101100010000
78	-125	19		10000011	00010011		100000110000	000100110000
79	111	-120		01101111	10001000		011011110000	100010000000
80	125	70		01111101	01000110		011111010000	010001100000
81	43	-76		00101011	10110100		001010110000	101101000000
82	74	-109		01001010	10010011		010010100000	100100110000
83	117	28		01110101	00011100		011101010000	000111000000
84	31	-100		00011111	10011100		000111110000	100111000000
85	-105	-4		10010111	11111100		100101110000	111111000000
86	75	-66		01001011	10111110		010010110000	101111100000
87	120	-68		01111000	10111100		011110000000	101111000000
88	58	78		00111010	01001110		001110100000	010011100000
89	56	-122		00111000	10000110		001110000000	100001100000
90	-126	-6		10000010	11111010		100000100000	111110100000
91	-77	-122		10110011	10000110		101100110000	100001100000
92	50	39		00110010	00100111		001100100000	001001110000
93	84	-56		01010100	11001000		010101000000	110010000000
94	-24	-10		11101000	11110110		111010000000	111101100000
95	-68	107		10111100	01101011		101111000000	011010110000

96	-78	107		10110010	01101011		101100100000	011010110000
97	5	114		00000101	01110010		000001010000	011100100000
98	-13	24		11110011	00011000		111100110000	000110000000
99	100	-86		01100100	10101010		011001000000	101010100000
100	-117	91		10001011	01011011		100010110000	010110110000
101	90	78		01011010	01001110		010110100000	010011100000
102	113	94		01110001	01011110		011100010000	010111100000
103	-62	-24		11000010	11101000		110000100000	111010000000
104	53	-33		00110101	11011111		001101010000	110111110000
105	0	-73		00000000	10110111		000000000000	101101110000
106	-13	41		11110011	00101001		111100110000	001010010000
107	68	87		01000100	01010111		010001000000	010101110000
108	-90	60		10100110	00111100		101001100000	001111000000
109	-75	-122		10110101	10000110		101101010000	100001100000
110	80	-74		01010000	10110110		010100000000	101101100000
111	19	-37		00010011	11011011		000100110000	110110110000
112	-97	-48		10011111	11010000		100111110000	110100000000
113	-95	-119		10100001	10001001		101000010000	100010010000
114	104	-73		01101000	10110111		011010000000	101101110000
115	92	108		01011100	01101100		010111000000	011011000000
116	-18	-114		11101110	10001110		111011100000	100011100000
117	13	46		00001101	00101110		000011010000	001011100000
118	-111	2		10010001	00000010		100100010000	000000100000
119	-79	36		10110001	00100100		101100010000	001001000000
120	78	-16		01001110	11110000		010011100000	111100000000
121	106	106		01101010	01101010		011010100000	011010100000
122	104	37		01101000	00100101		011010000000	001001010000
123	67	8		01000011	00001000		010000110000	000010000000
124	-25	-36		11100111	11011100		111001110000	110111000000
125	-101	94		10011011	01011110		100110110000	010111100000
126	84	124		01010100	01111100		010101000000	011111000000
127	84	-13		01010100	11110011		010101000000	111100110000
128	-21	-28		11101011	11100100		111010110000	111001000000

	Option 1 / 2 VHDL							
max_a	-28	Ø	108	Ø	-43	Ø	77	Ø
max_b	16	Ø	-64	Ø	24	Ø	66	Ø
max_c	1272	Ø	-5200	Ø	2603	Ø	-583	Ø
max_Inlier_num	16	Ø	22	Ø	34	Ø	68	Ø
max_Dis	37	Ø	99	Ø	377	Ø	1671	Ø
thres_dis	5		10		20		50	

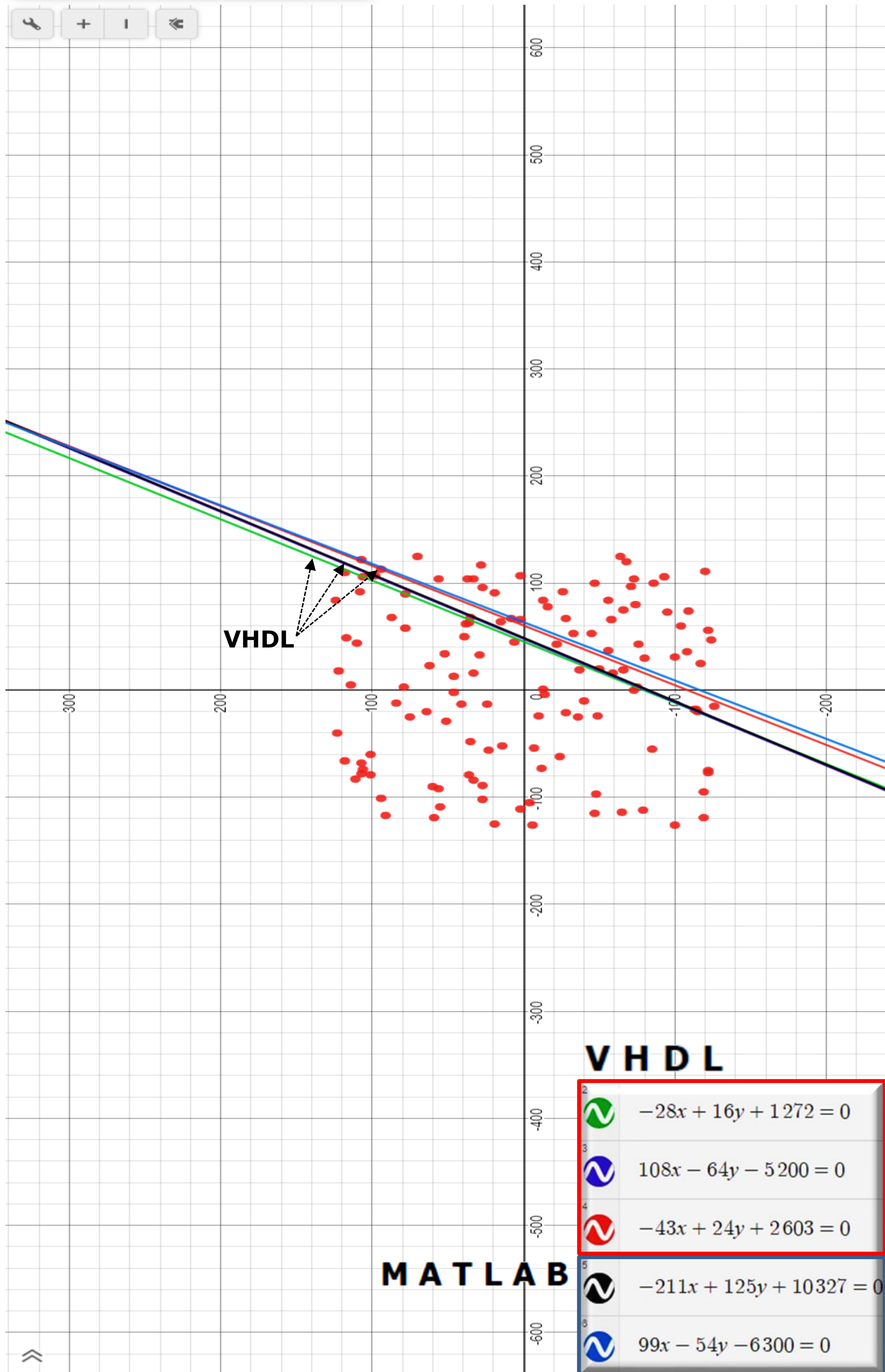
Εικόνα - Παράμετροι a, b, c ---> VHDL

	MATLAB			
max_a	-23	-211	99	-127
max_b	-182	125	-54	-129
max_c	8001	10327	-6300	-2143
max_Inlier_num	15	22	37	70
max_Dis	28	104	423	1853
thres_dis	5	10	20	50

Εικόνα - Παράμετροι a, b, c ---> Matlab

a/a	max_i	max_a	max_b	max_c	sum_dis
1	20	28	205	-8223	101.85
2	20	108	-68	-4964	102.55
3	20	152	-43	-8831	103.21
4	22	211	-125	-10327	104.19

Matlab sort thres_dis 10



Εικόνα - Γραφική παράσταση Option 1 best line fit (thr_dis 5-20) + Matlab (thr_dis 10, 20)

5.4 Συμπεράσματα

Απο τα ευρήματα αυτής της εργασίας διαπιστώνεται ότι είναι πιθανή η εύρωστη εκτίμηση μοντέλου ευθείας από σύνολο δειγμάτων υλοποιώντας τον αλγόριθμο RANSAC σε FPGA με χρήση γλώσσας περιγραφής υλικού VHDL.

Από πειραματική εκτίμηση, σε συνδυασμό με κάποιο ενδεικτικό μέτρο σύγκρισης, ώστε να βρεθεί η βέλτιστη από περισσότερες από μία λύσεις, διαπιστώνεται ότι κρίνεται απαραίτητη η ύπαρξη ικανού αριθμού μετρήσεων - δειγμάτων.

Εμφανίζονται μικρές αποκλίσεις στις τιμές που λαμβάνονται από το Matlab λόγω μεγαλύτερης ακρίβειας υπολογισμών.

Ο καθορισμός των ορίων της τιμής κατωφλίου των inliers είναι ένας σημαντικός παράγοντας. Στην συγκεκριμένη περίπτωση διαπιστώθηκε ότι οι βέλτιστες τιμές κυμαίνονται μεταξύ 4-8% του εύρους κλίμακας των μετρήσεων.

Η μέθοδος των παράλληλων υπολογισμών (ασύγχρονη) είναι από 235 φορές (16 σημεία) έως και 123.000 φορές (128 σημεία) ταχύτερη από την συγχρονισμένη.

Η μέθοδος των παράλληλων υπολογισμών είναι σταθερή στον χρόνο εκτέλεσης, ανεξαρτήτως του αριθμού των σημείων, σε αντίθεση με την σύγχρονη μέθοδο, όπου ο χρόνος αυξάνεται εκθετικά.

Η όλη διαδικασία εκτελέστηκε σε εικονικό περιβάλλον, ωστόσο θα ήταν χρήσιμη και η δοκιμή σε πραγματική συσκευή με τον απαραίτητο εξοπλισμό.

5.5 Μελλοντική Εργασία

Όπως προαναφέρθηκε με την θέση εξαρχής μιας συσκευής ως target technology, θα μπορούσαν να βελτιστοποιηθούν συγκεκριμένες παράμετροι όπως ταχύτητα, κατανάλωση κ.ο.κ.

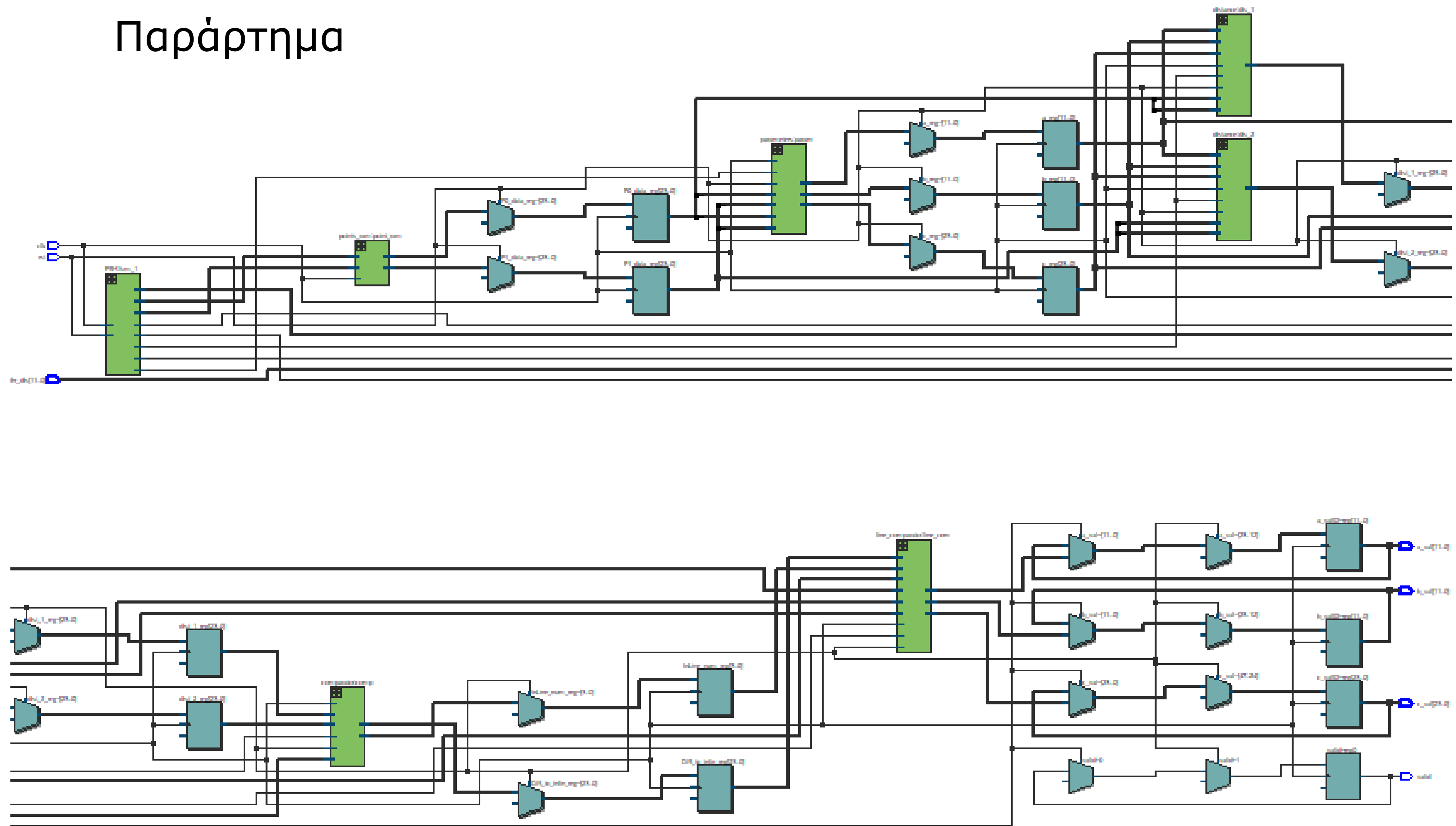
Προκύπτει ότι ο αριθμός των inliers κυμαίνεται (για το βέλτιστο threshold), από 50% (για 16, 32 σημεία) έως 40% (64 σημεία) & 30% (128 σημεία), με αποτέλεσμα μια εναλλακτική πρόταση να είναι η αναζήτηση αυτού του ποσοστού, αντί του ελέγχου όλων των συνδυασμών.

Σε ότι αφορά την μέθοδο παράλληλων υπολογισμών, θα ήταν χρήσιμη ή εύρεση τρόπων βελτιστοποίησης, όπως π.χ. με την χρήση των built-in εργαλείων για εξοικονόμηση πόρων και χρόνου.

Μια εναλλακτική θα ήταν η υλοποίση του R-RANSAC για τον προσδιορισμό ενός υποσυνόλου δεδομένων και την μείωση του υπολογιστικού φόρτου ή ο προέλεγχος της ποιότητας του συνόλου των μετρήσεων, με βάση τα αποτελέσματα της ασύγχρονης μεθόδου.

Τέλος, στην parallel αρχιτεκτονική σχεδίαση, κρίσιμη θεωρείται και η εξαγωγή του αθροίσματος των αποστάσεων των inliers μέσω της προσομοίωσης στο Modelsim.

Παράρτημα



Εικόνα - RTL Option 1

Εικόνα - Quartus Compilation Report Option 1

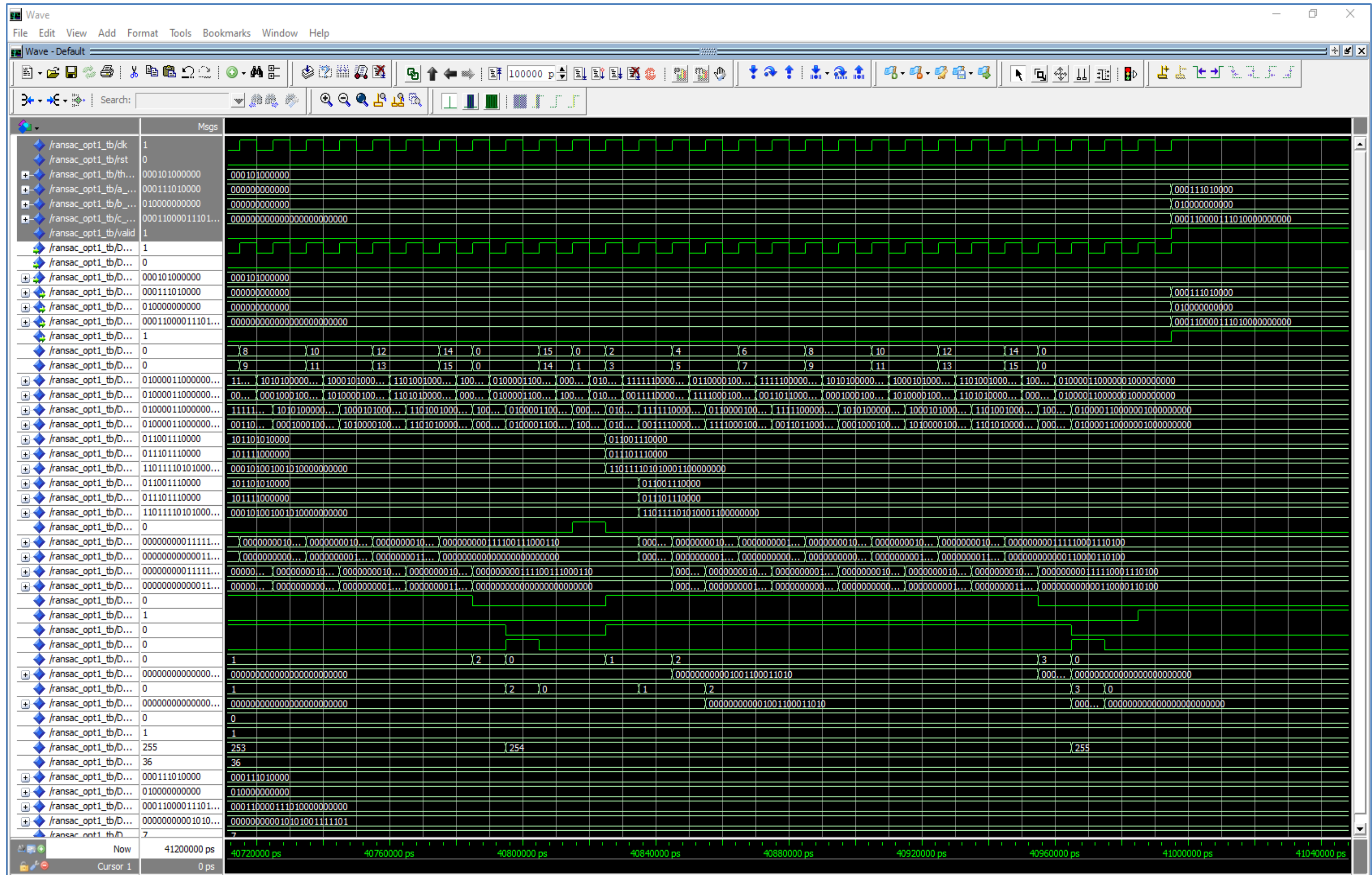
Compilation Flow	Time
Compile Design	02:10:17
IP Generation	00:00:04
Analysis & Synthesis	00:07:30
Fitter	02:01:03
Fitter (Implement)	
Plan	00:04:56
Early Place	
Place	00:54:09
Route	00:30:30
Retime	00:05:01
Fast Forward Timing Closure Recommendations	
Fitter (Finalize)	00:26:22
Timing Analysis (Signoff)	00:01:06
Power Analysis	
Assembler (Generate programming files)	
EDA Netlist Writer	

```
Worst-case setup slack is -50.994                              332146
Worst-case hold slack is 0.197                                 332146
Worst-case recovery slack is -6.405                            332146
Worst-case removal slack is 0.191                              332146
Worst-case minimum pulse width slack is -0.118                 332146
Design is not fully constrained for setup requirements         332102
Design is not fully constrained for hold requirements          332102
Quartus Prime Timing Analyzer was successful. 0 errors, 3 warnings
  Peak virtual memory: 12752 megabytes
  Processing ended: Sat May 22 10:30:42 2021
```

Εικόνα - Quartus Compilation Report Option 2

Εικόνα - Modelsim Option 1 (16 points (thr_dis 5))

Εικόνα - Modelsim Option 1 (16 points (thr_dis 10))



Εικόνα - Modelsim Option 1 (16 points (thr_dis 20))

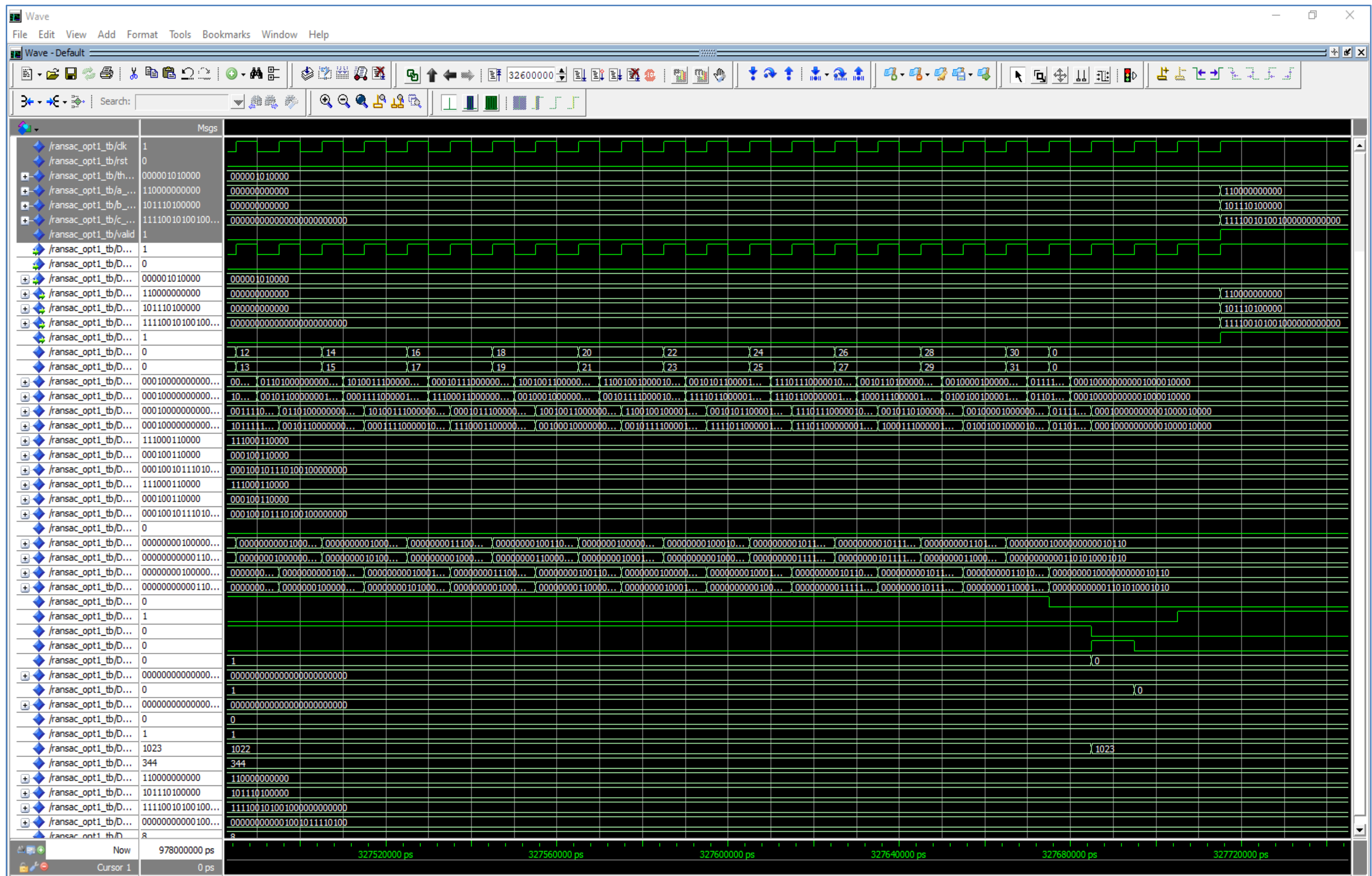
Εικόνα - Modelsim Option 1 (16 points (thr_dis 50))

Εικόνα - Modelsim Option 2 (16 points (thr_dis 5))

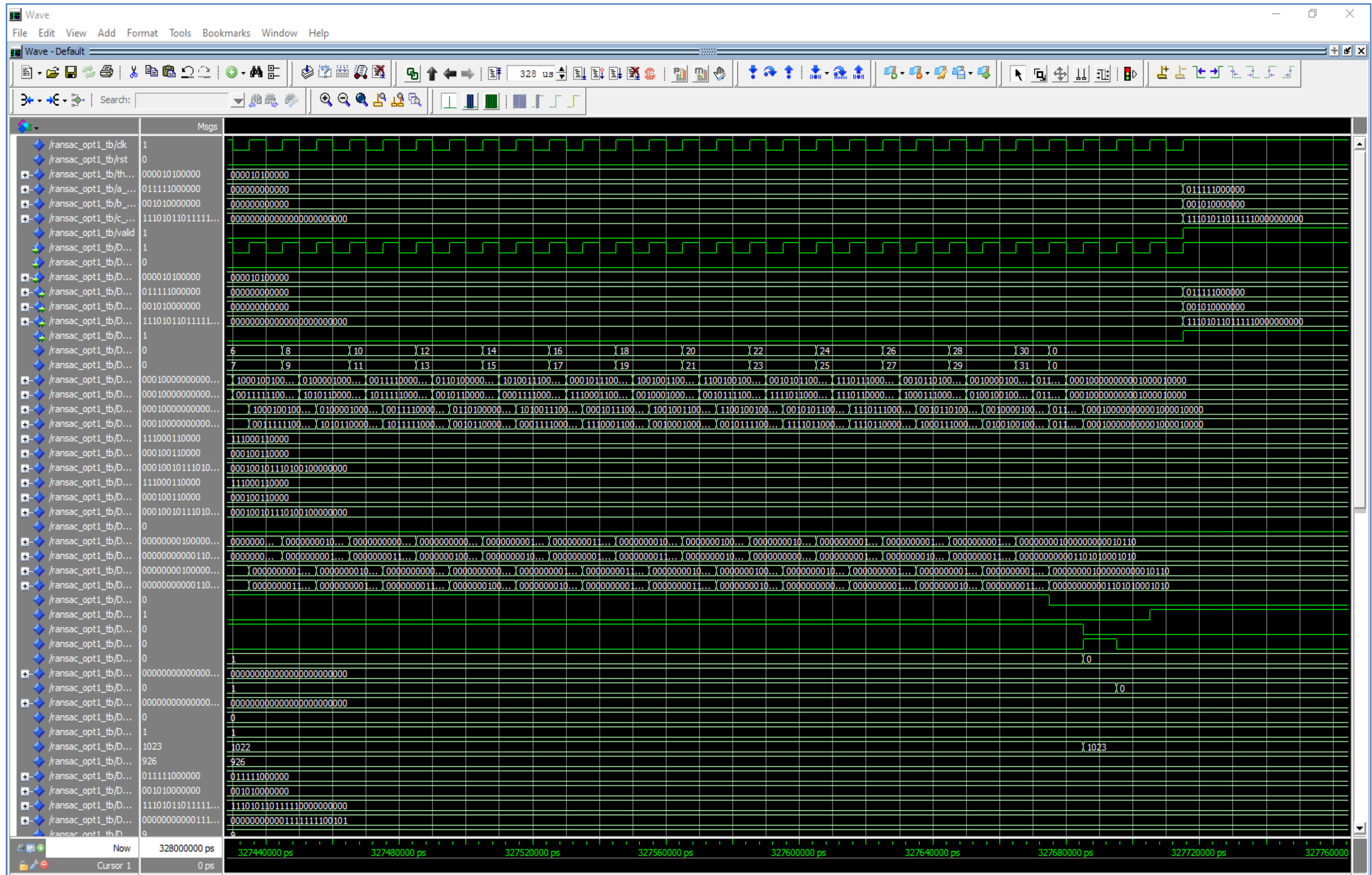
Εικόνα - Modelsim Option 2 (16 points (thr_dis 10))

Εικόνα - Modelsim Option 2 (16 points (thr_dis 20))

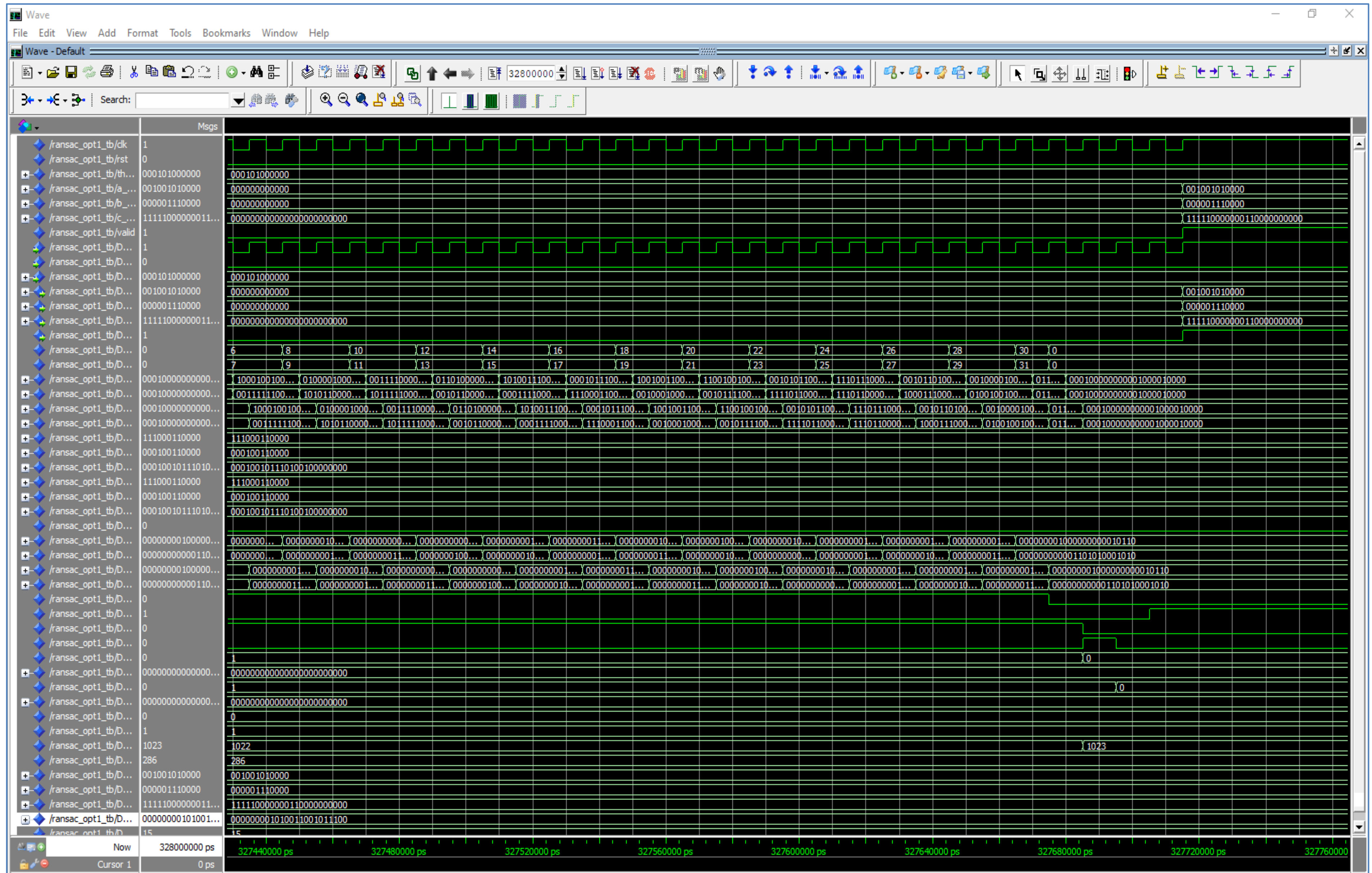
Εικόνα - Modelsim Option 2 (16 points (thr_dis 50))



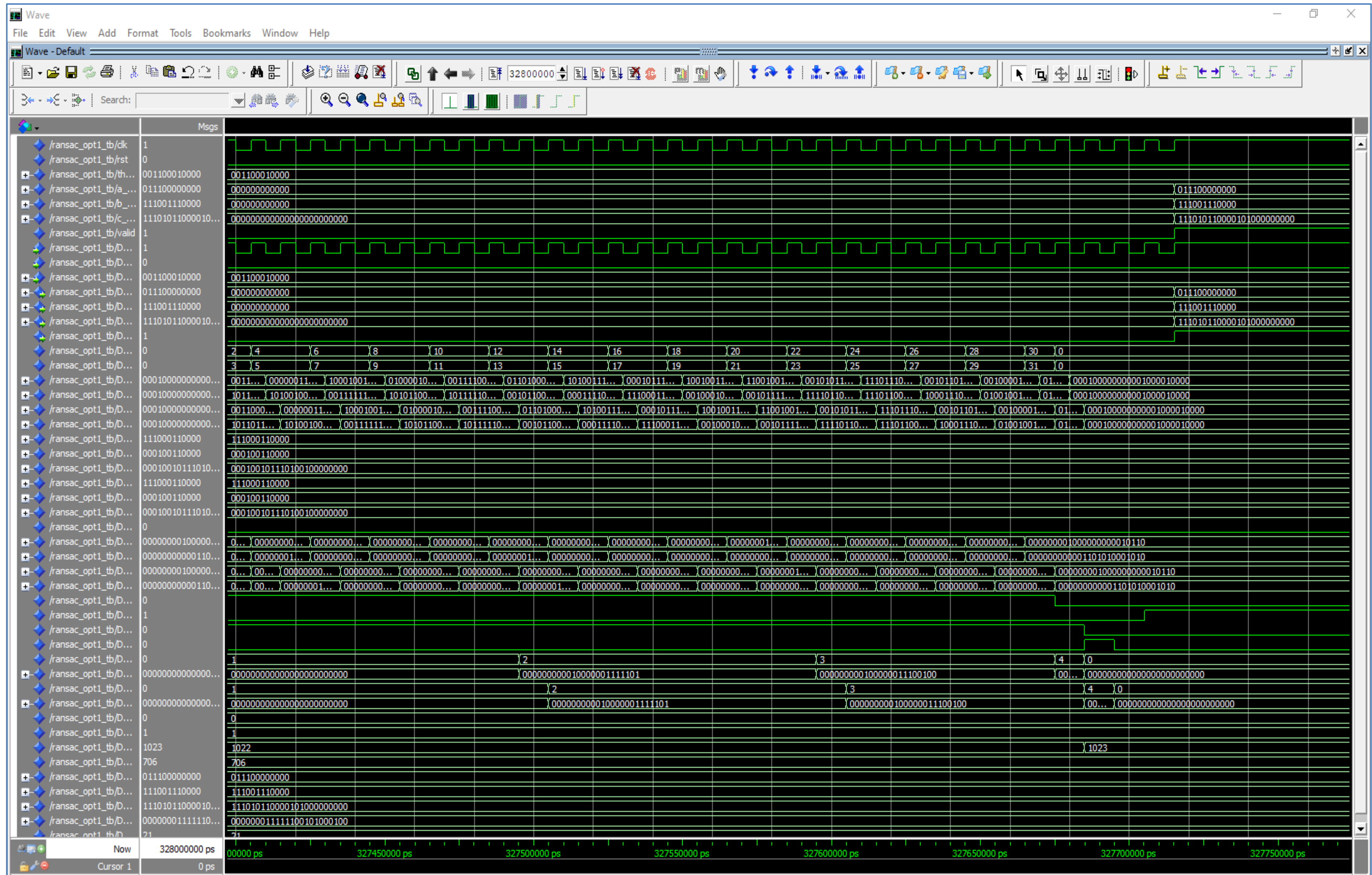
Εικόνα - Modelsim Option 1 (32 points (thr_dis 5))



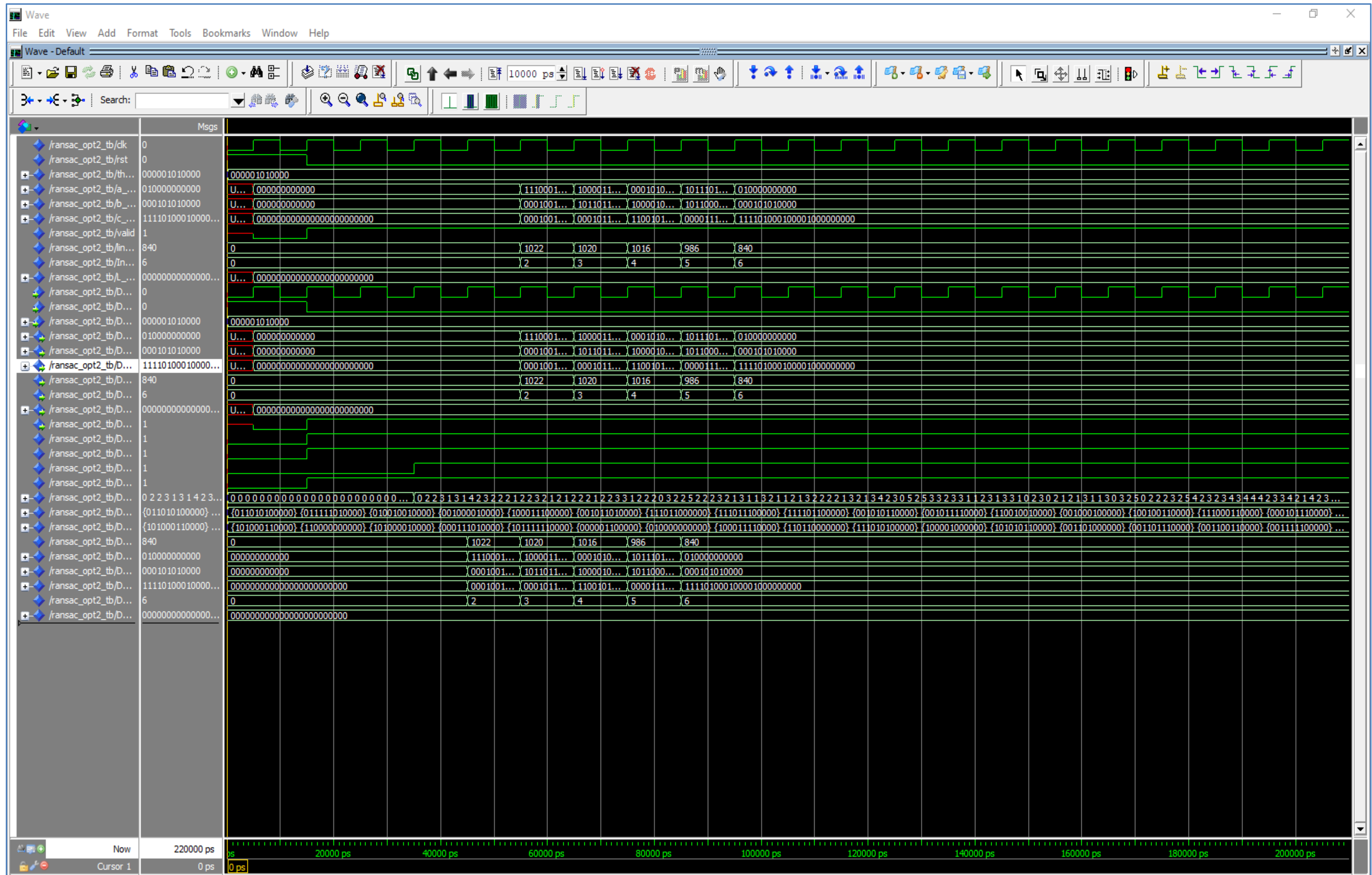
Εικόνα - Modelsim Option 1 (32 points (thr_dis 10))



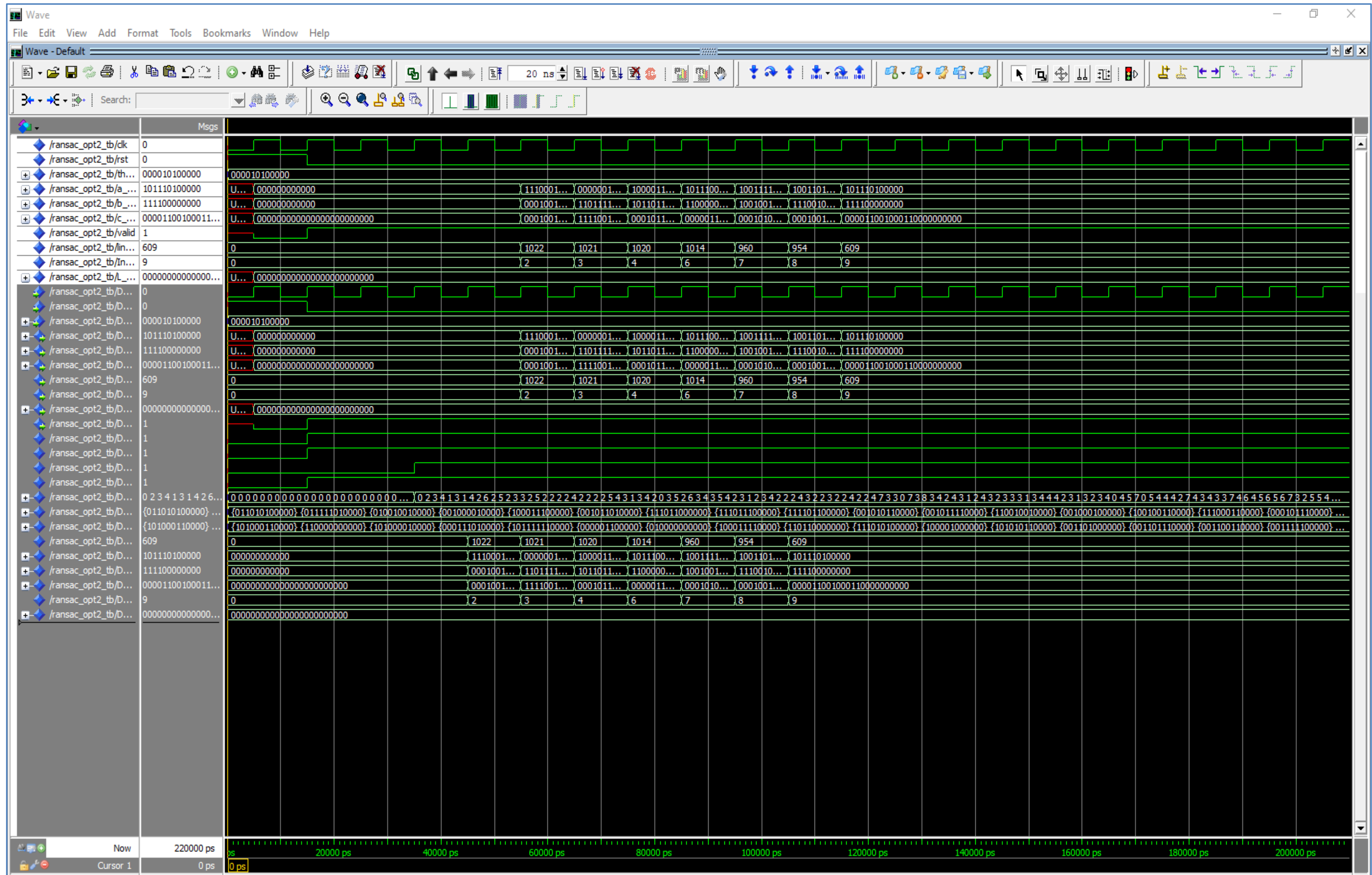
Εικόνα - Modelsim Option 1 (32 points (thr_dis 20))



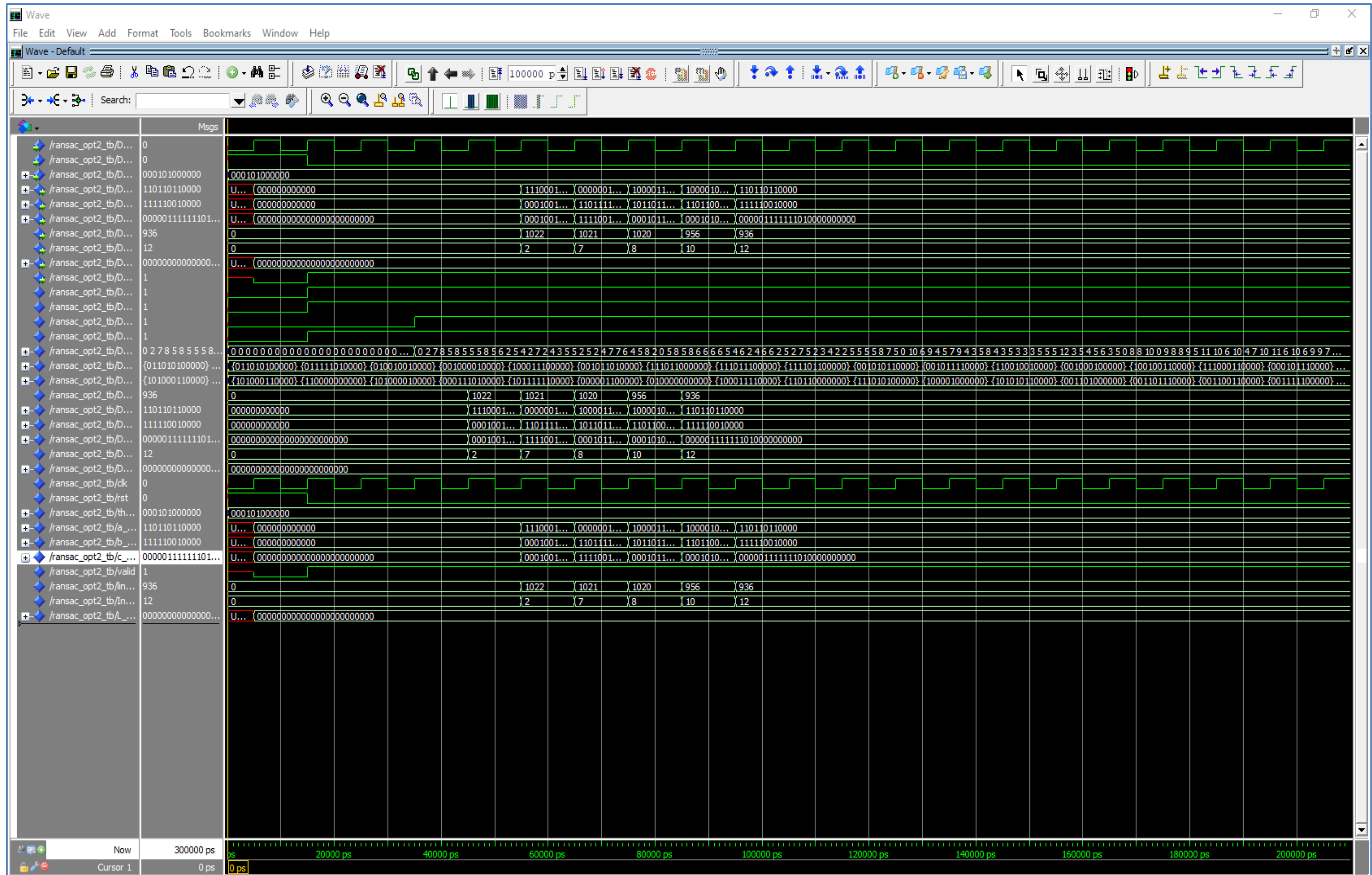
Εικόνα - Modelsim Option 1 (32 points (thr_dis 50))



Εικόνα - Modelsim Option 2 (32 points (thr_dis 5))



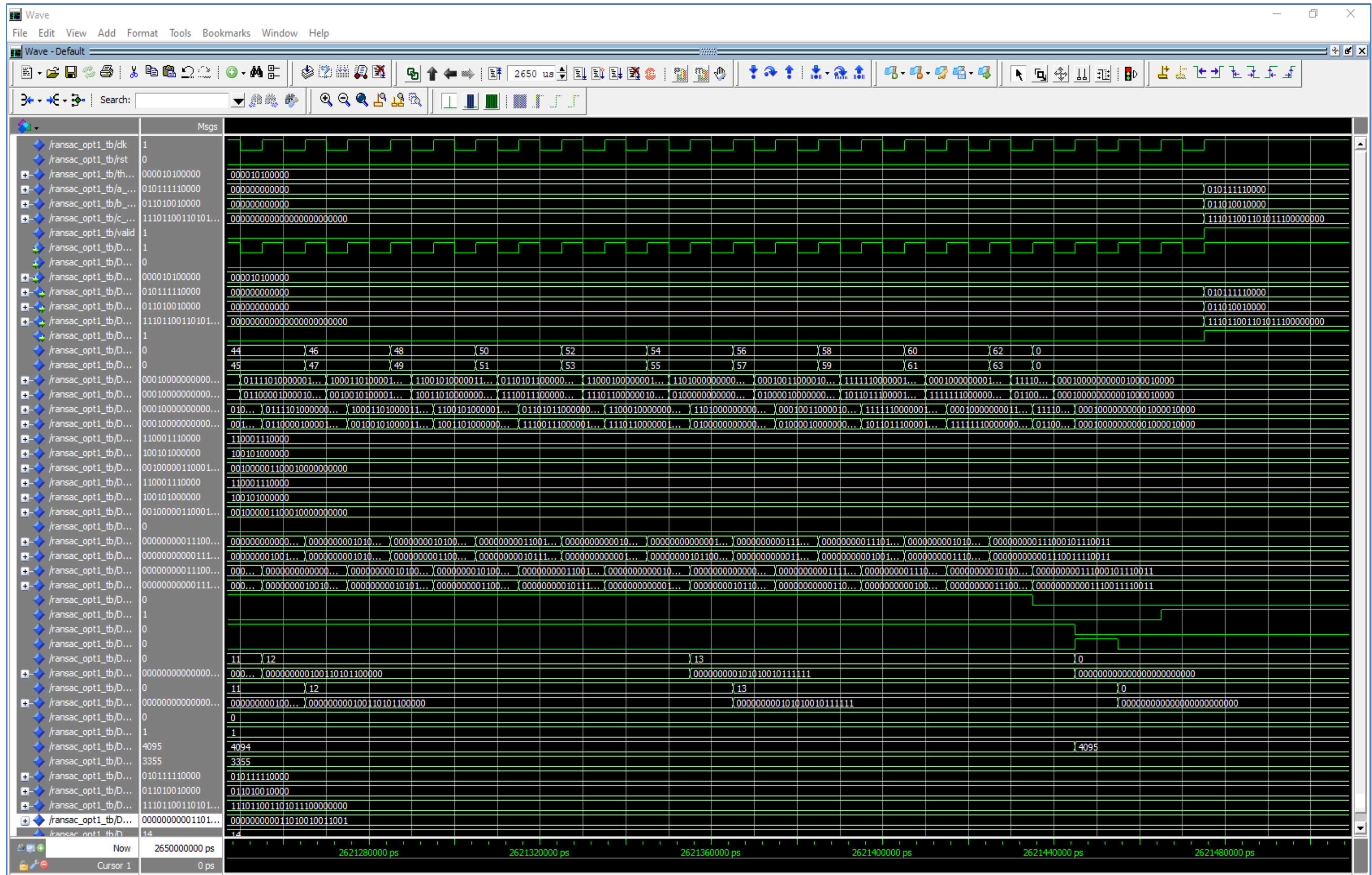
Εικόνα - Modelsim Option 2 (32 points (thr_dis 10))



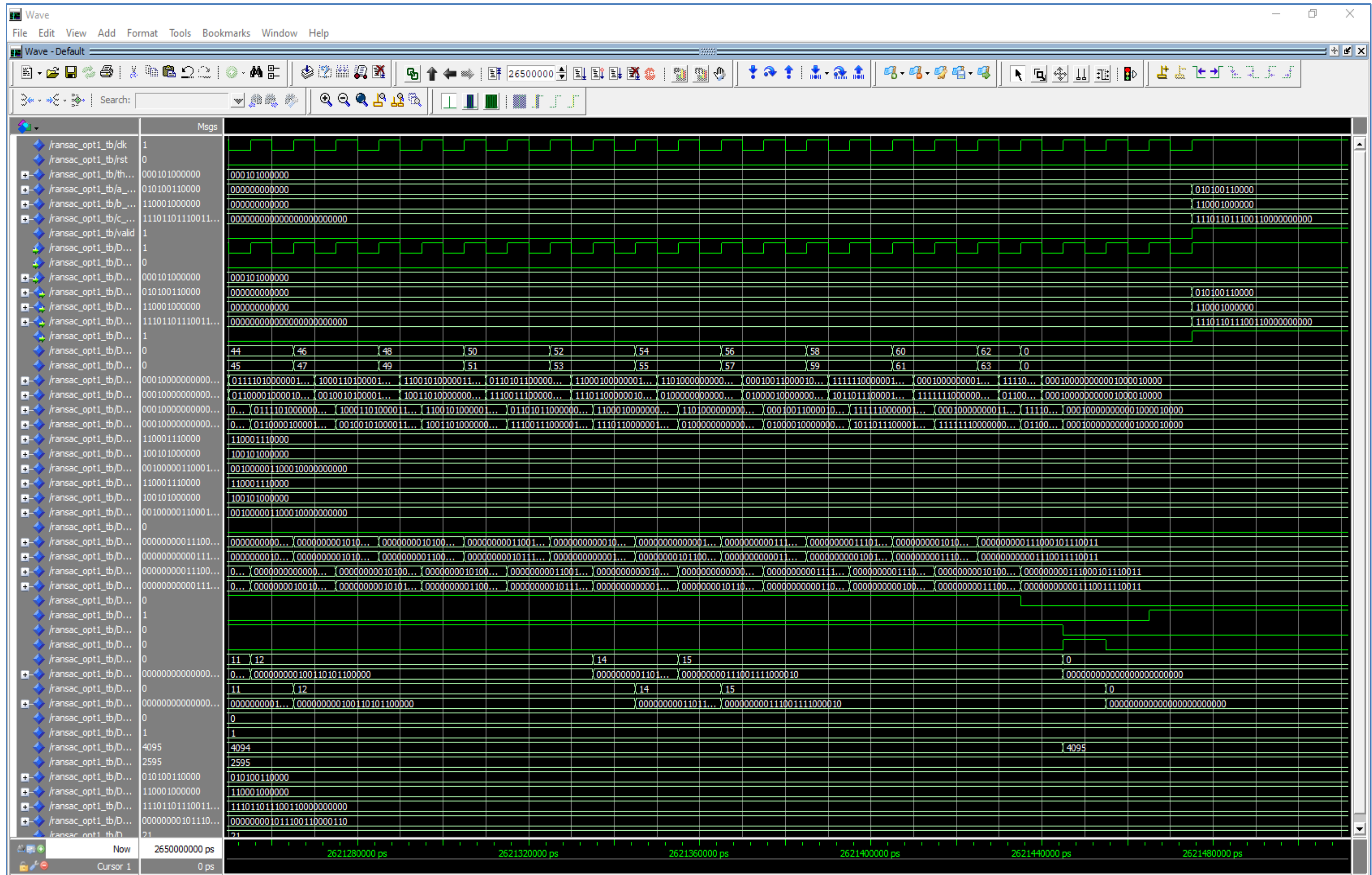
Εικόνα - Modelsim Option 2 (32 points (thr_dis 20))

Εικόνα - Modelsim Option 2 (32 points (thr_dis 50))

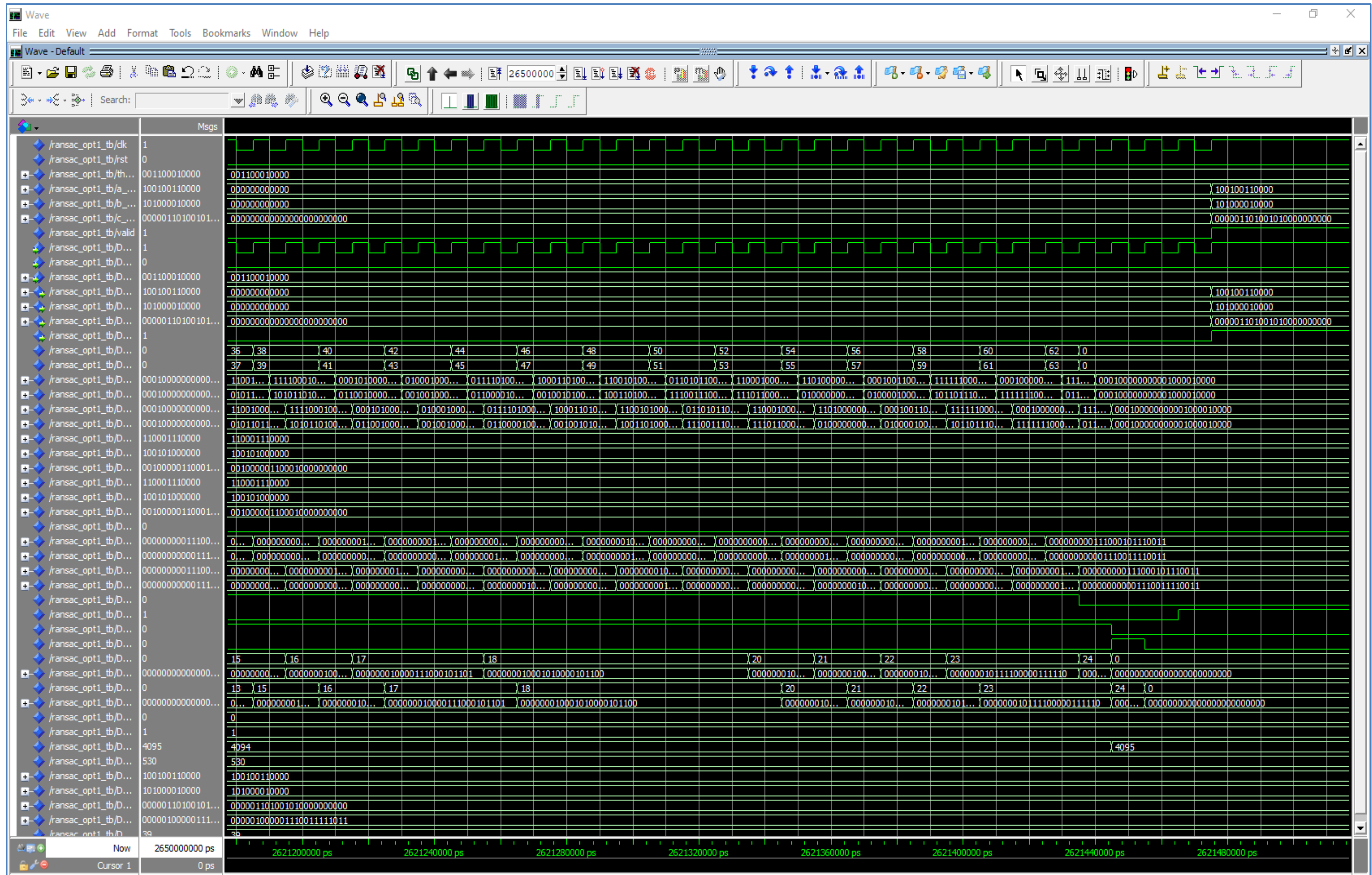
Εικόνα - Modelsim Option 1 (64 points (thr_dis 5))



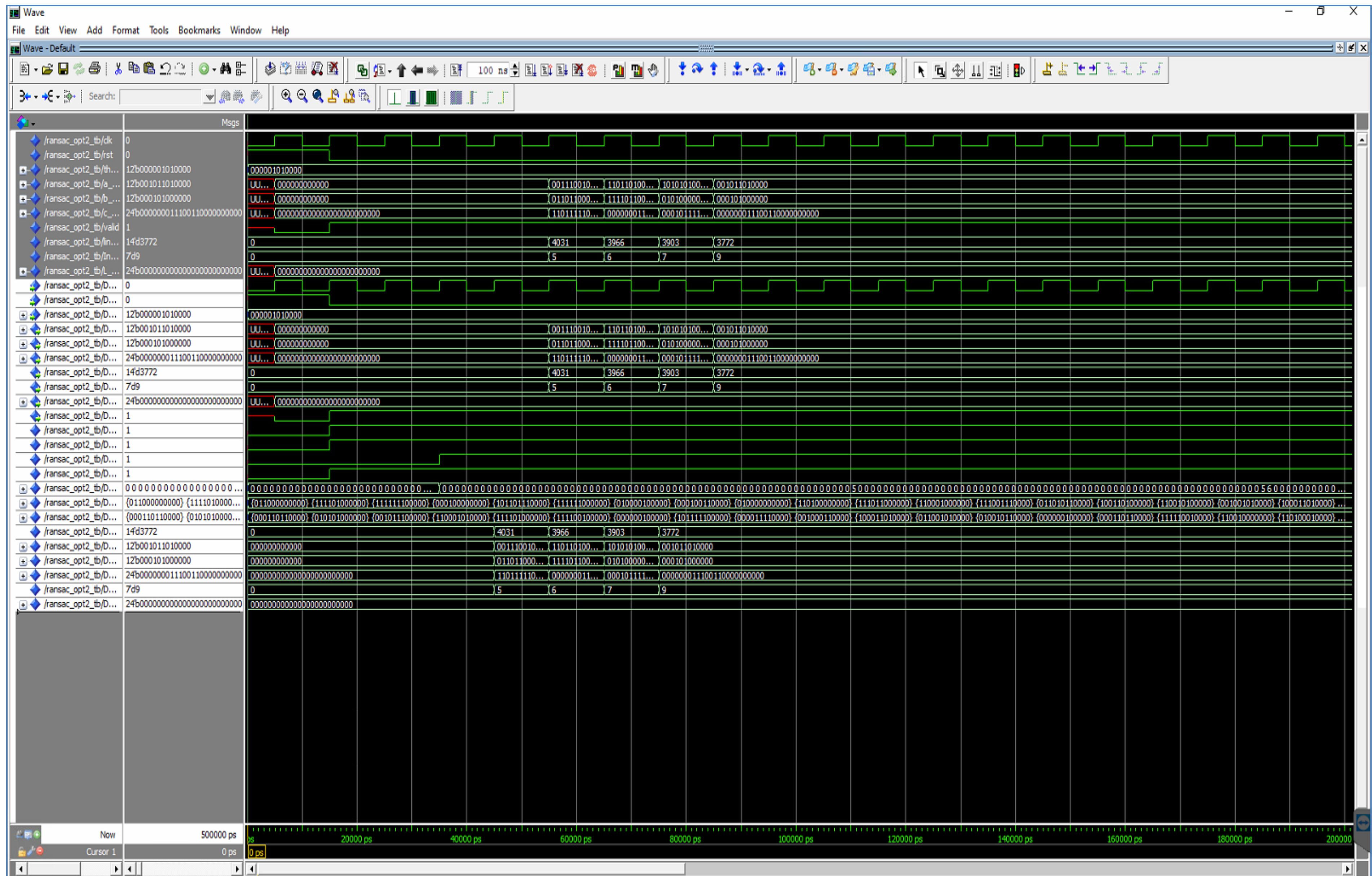
Εικόνα - Modelsim Option 1 (64 points (thr_dis 10))



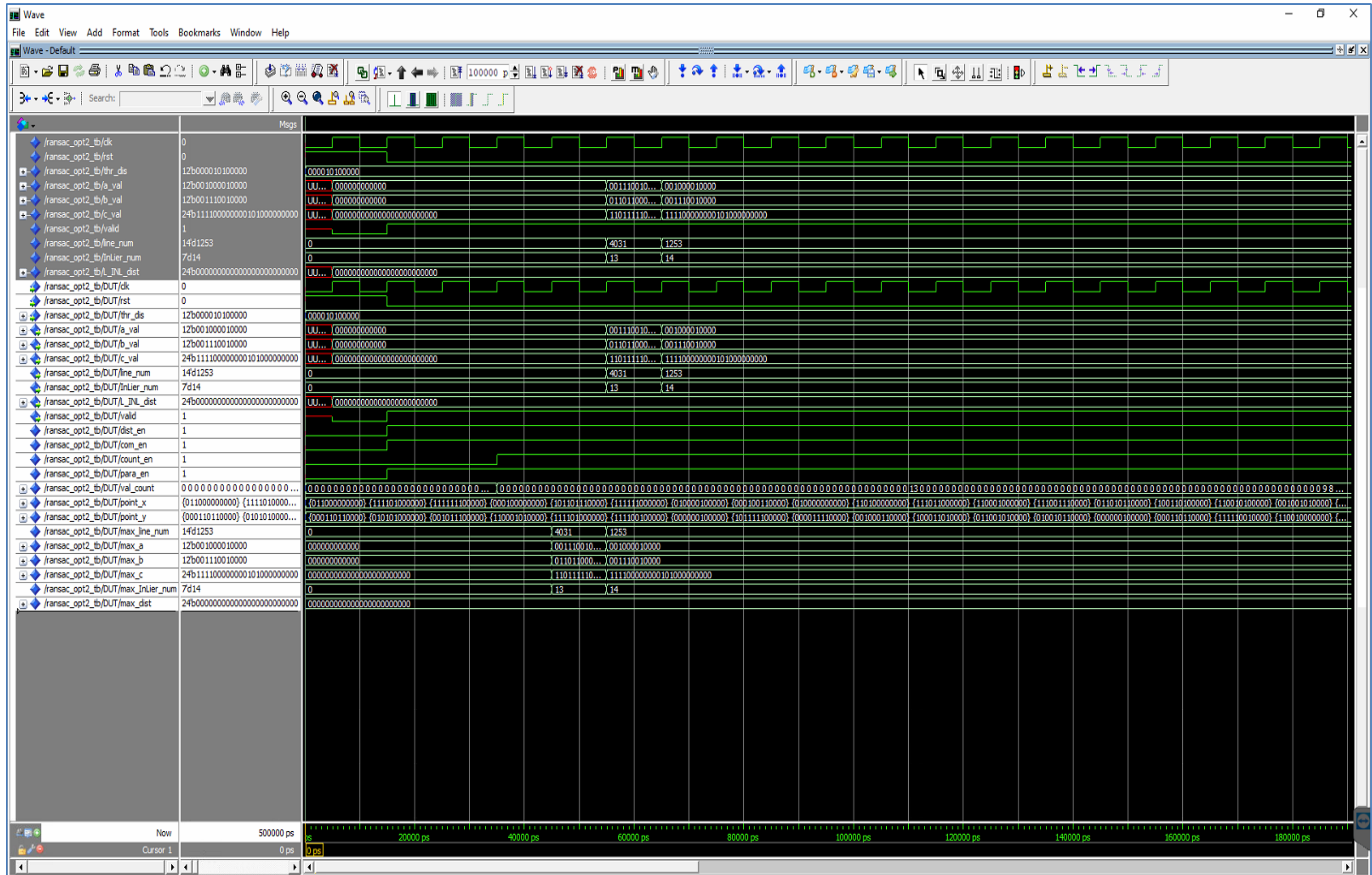
Εικόνα - Modelsim Option 1 (64 points (thr_dis 20))



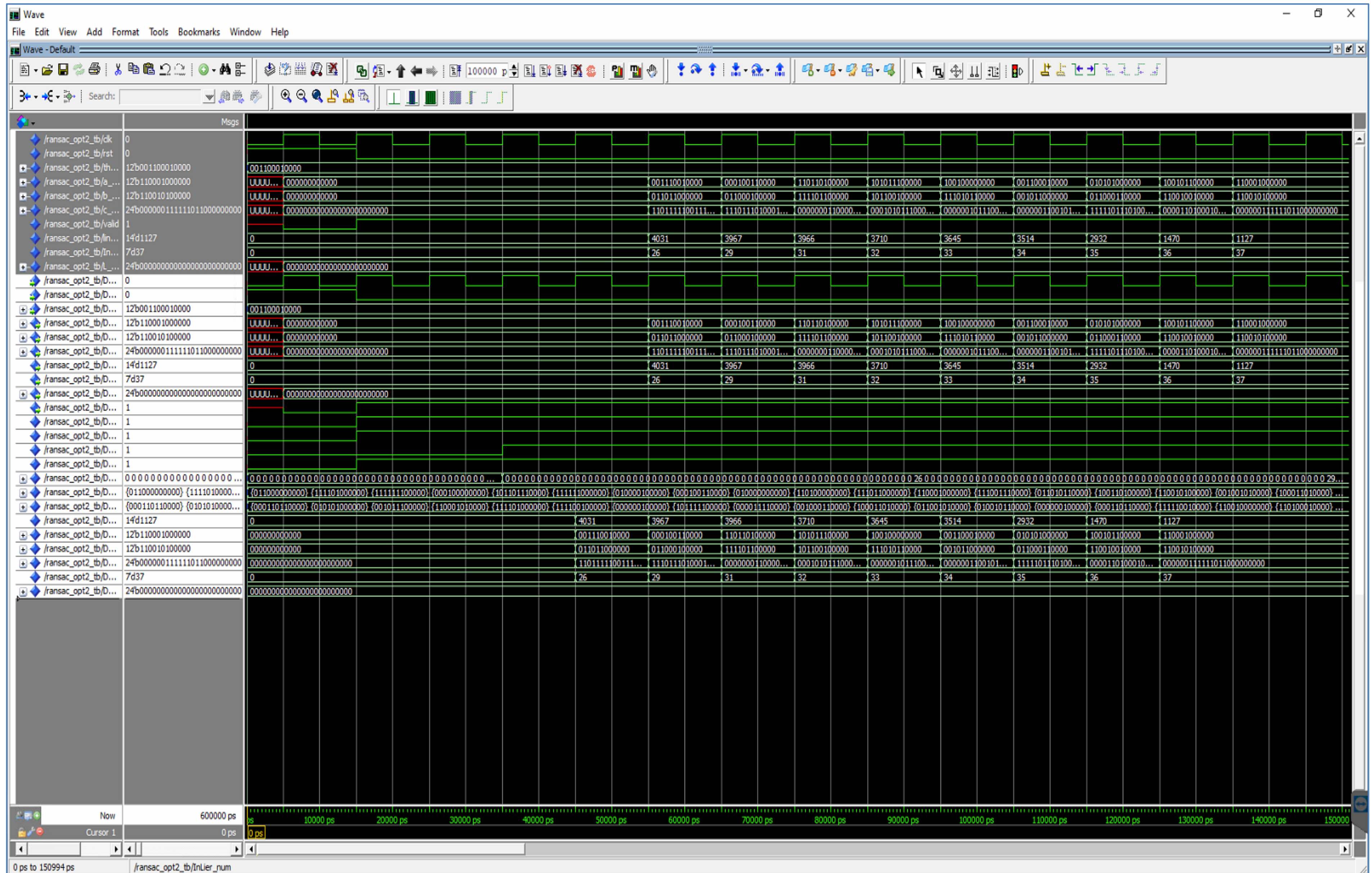
Εικόνα - Modelsim Option 1 (64 points (thr_dis 50))



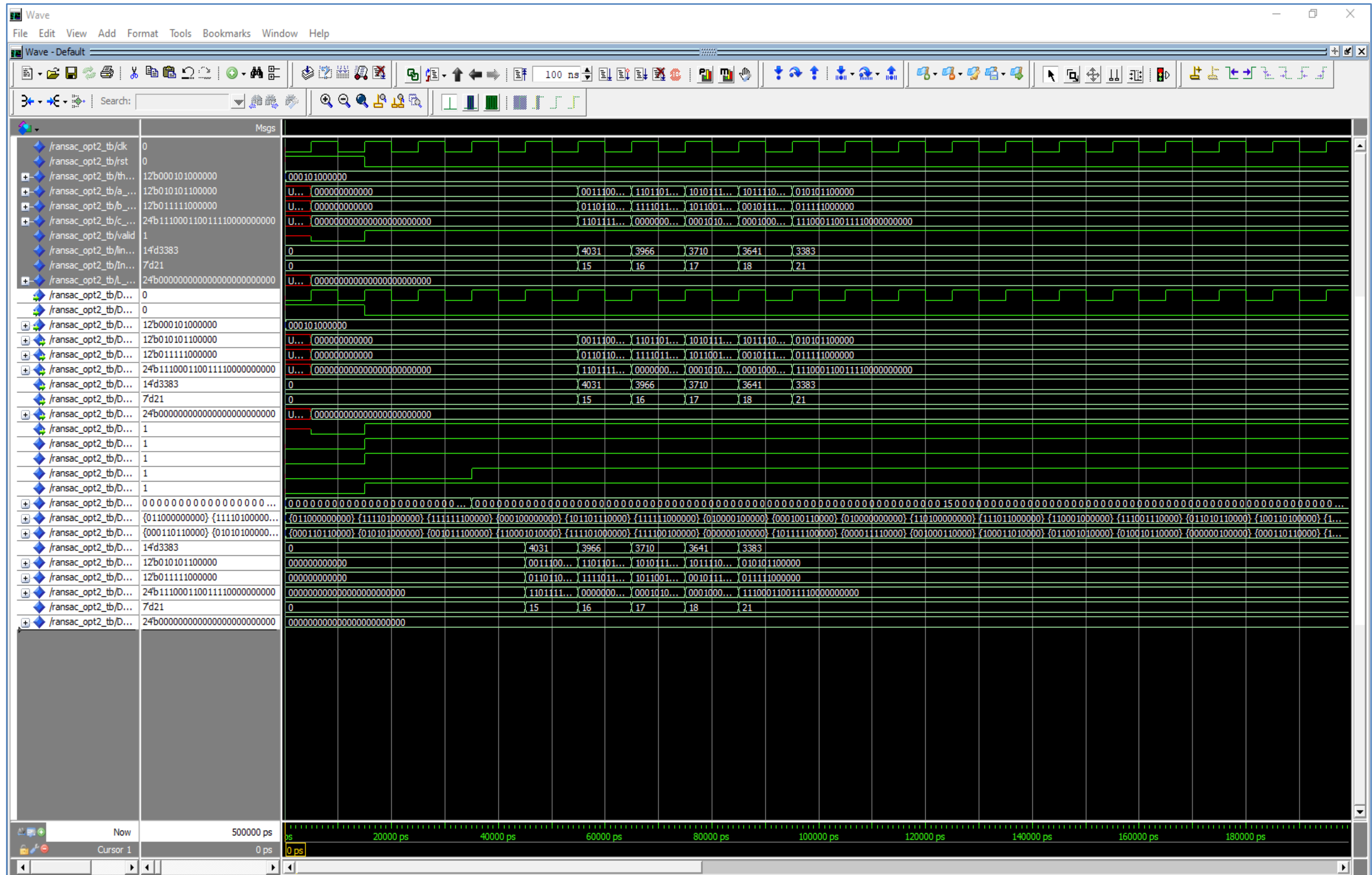
Εικόνα - Modelsim Option 2 (64 points (thr_dis 5))



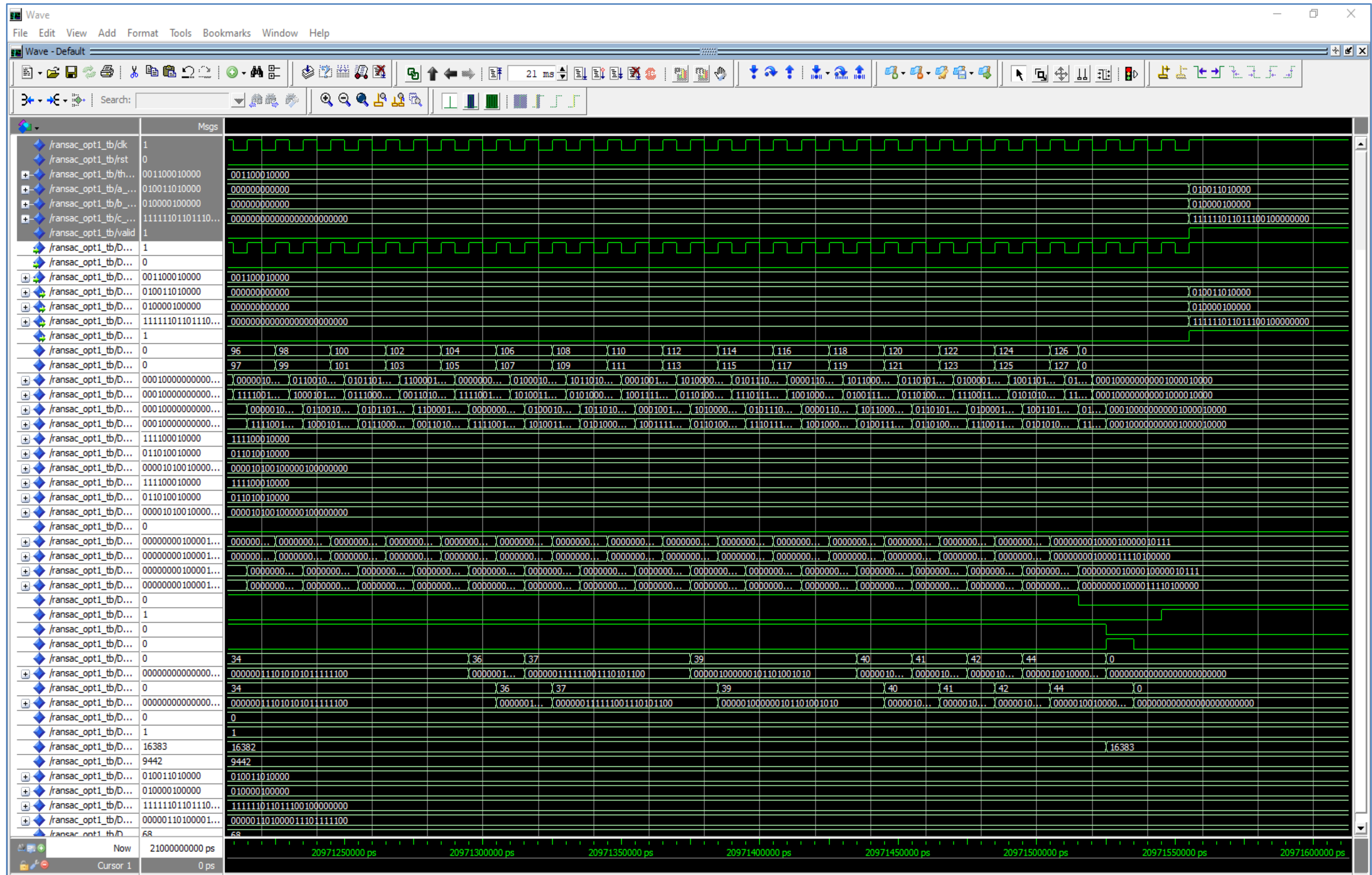
Εικόνα - Modelsim Option 2 (64 points (thr_dis 10))



Εικόνα - Modelsim Option 2 (64 points (thr_dis 20))

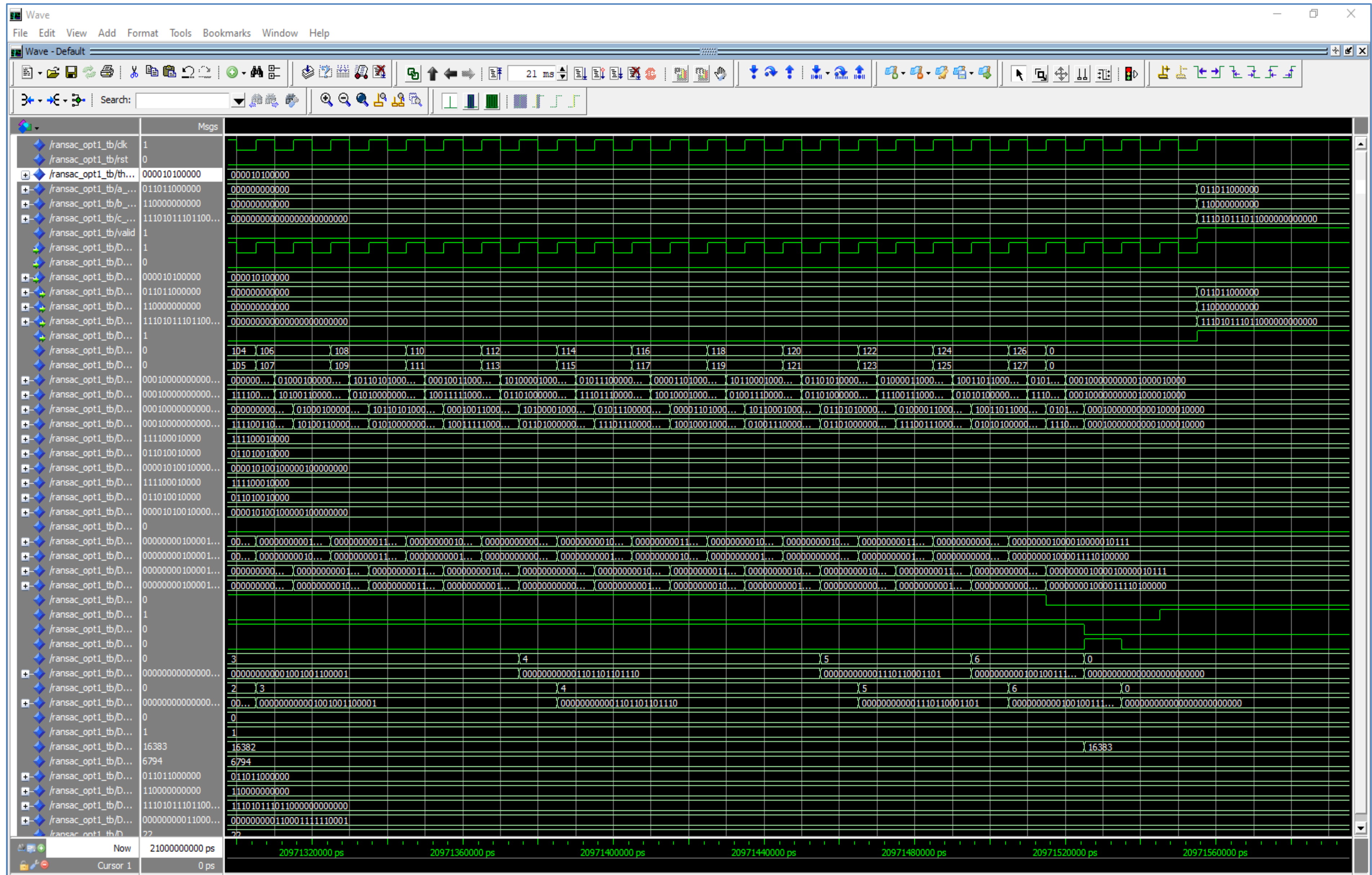


Εικόνα - Modelsim Option 2 (64 points (thr_dis 50))



Εικόνα - Modelsim Option 1 (128 points (thr_dis 5))

Εικόνα - Modelsim Option 1 (128 points (thr_dis 10))



Εικόνα - Modelsim Option 1 (128 points (thr_dis 20))

Εικόνα - Modelsim Option 1 (128 points (thr_dis 50))

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
entity add is generic (width : integer := UD);
port(
    a, b   : in  signed (width-1 downto 0);
    result : out signed (width-1 downto 0));
end add;
architecture rtl_add of add is
begin
    process (a, b)
    begin
        result <= a + b;
    end process;
end rtl_add;
```

add.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
entity sub is generic (width : integer := UD);
port(
    a, b   : in  signed (width-1 downto 0);
    result : out signed (width-1 downto 0));
end add;
architecture rtl_sub of sub is
begin
    process (a, b)
    begin
        result <= a - b;
    end process;
end rtl_sub;
```

sub.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
entity mult is generic (width : integer := UD);
port(
    a, b   : in  signed (width-1 downto 0);
    result : out signed ((width*2)-1 downto 0));
end add;
architecture rtl_mult of mult is
begin
    process (a, b)
    begin
        result <= a * b;
    end process;
end rtl_mult;
```

mult.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
entity parameters is generic (width : integer := UD);
port(
    clk, rst, en: in std_logic;
    x1, y1, x2, y2 : in signed (width-1 downto 0);
    a,b : out signed (width-1 downto 0);
    c : out signed (2*width-1 downto 0));
end parameters;
architecture behavioral of parameters is
component add generic (width : integer);
port(
    a, b   : in  signed (width-1 downto 0);
    result: out signed (width-1 downto 0));
end component;
component sub generic (width : integer);
port(
    a, b   : in  signed (width-1 downto 0);
    result: out signed (width-1 downto 0));
end component;
component mult generic (width : integer := UD);
port(
    a, b   : in  signed (width-1 downto 0);
    result: out signed (2*width-1 downto 0));
end component;
signal m, m_comp, n, n_comp : signed (width-1 downto 0);
signal k, h, d : signed (2*width-1 downto 0);
begin
subtr_1 : sub  generic map (width => width) port map (a=>y1, b=>y2, result=>m);
subtr_2 : sub  generic map (width => width) port map (a=>y2, b=>y1, result=>m_comp);
subtr_3 : sub  generic map (width => width) port map (a=>x2, b=>x1, result=>n);
subtr_4 : sub  generic map (width => width) port map (a=>x1, b=>x2, result=>n_comp);
multr_1 : mult generic map (width => width) port map (a=>y1, b=>n_comp, result=>k);
multr_2 : mult generic map (width => width) port map (a=>x1, b=>m_comp, result=>h);
adder_1 : add  generic map (width => 2*width) port map (a=>k, b=>h, result=>d);
proc: process (clk, rst, en, m, n, d)
begin
    if (rising_edge (clk)) then
        if (en = '1') then
            a <= m;
            b <= n;
            c <= d;
        elsif (rst = '1') then
            a <= (others => '0');
            b <= (others => '0');
            c <= (others => '0');
        end if;
    end if;
end process;
end behavioral;
```
parameters.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
entity distance is generic (width : integer := UD);
port(
    clk, rst, en: in std_logic;
    x, y, a, b : in signed (width-1 downto 0);
    c : in signed (2*width-1 downto 0);
    dist : out unsigned (2*width-1 downto 0) := (others =>'0'));
end distance;
architecture behavioral of distance is
component add generic (width : integer := UD);
port(
    a, b   : in  signed (width-1 downto 0);
    result : out signed (width-1 downto 0));
end component;
component sub generic (width : integer := UD);
port(
    a, b   : in  signed (width-1 downto 0);
    result : out signed (width-1 downto 0));
end component;
component mult generic (width : integer := UD);
port(
    a, b   : in  signed (width-1 downto 0);
    result : out signed (2*width-1 downto 0));
end component;
component SQRT is generic (N : integer := UD);
port(
    input   : in unsigned (N-1 downto 0);
    sq_root : out unsigned (N/2-1 downto 0));
end component;
component dividor is generic (N : integer := UD);
port(
    a,b : signed (N-1 downto 0);
    c   : out signed (N-1 downto 0);
end component;
signal k, h, l, v, V_c, aa, bb, w, S_24 : signed (2*width - 1 downto 0);
signal s : signed (width - 1 downto 0);
signal d : unsigned (2*width-1 downto 0);
begin
multr_1 : mult generic map (width => width) port map (a=>a, b=>x, result=>k);
multr_2 : mult generic map (width => width) port map (a=>b, b=>y, result=>h);
adder_1 : add  generic map (width => 2*width) port map (a=>k, b=>h, result=>l);
adder_2 : add  generic map (width => 2*width) port map (a=>l, b=>c, result=>v);
multr_3 : mult generic map (width => width) port map (a=>a, b=>a, result=>aa);
multr_4 : mult generic map (width => width) port map (a=>b, b=>b, result=>bb);
adder_3 : add  generic map (width => 2*width) port map (a=>aa, b=>bb, result=>w);
SQRT_1  : SQRT generic map (N => 2*width)
```

```
port map (input => unsigned (w), signed (sq_root) => s);
process (v)
begin
    if (v (2*width-1) = '1') then
        V_c <= not(v) + "000000000000000000000001";
    else
        v_c <= v;
    end if;
end process;
s_24 <= "00000000" & s & "0000";
div: dividor generic map (N => (2*width))
port map (a => v_c, b => s_24, unsigned (c) => d);
proc_1: process (clk)
begin
    if (rising_edge (clk)) then
        if (rst = '1') then
            dist <= (others => '0');
        elsif (en = '1') then
            dist <= d ;
        end if;
    end if;
end process;
end behavioral;
```
distance.vhd

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity SQRT is generic (N : integer := UD);
port (input : in  unsigned (N-1 downto 0);
    sq_root : out unsigned (N/2-1 downto 0));
end SQRT;
architecture behav of SQRT is
        function sqrt_1 (d : unsigned) return unsigned is
        variable a : unsigned (N-1 downto 0) := d;
        variable q : unsigned ((N/2)-1 downto 0) := (others => '0');
        variable left, right, r : unsigned ((N/2) + 1 downto 0) := (others => '0');
        variable i : integer:= 0;
        begin
        for i in 0 to (N/2) - 1 loop
        right (0) := '1';
        right (1) := r ((N/2) + 1);
        right ((N/2)+1 downto 2) := q;
        left (1 downto 0) := a (N-1 downto N-2);
        left ((N/2)+1 downto 2) := r ((N/2) - 1 downto 0);
        a (N-1 downto 2) := a (N-3 downto 0);
        if (r ((N/2)+1) = '1') then
        r := left + right;
        else
        r := left - right;
        end if;
        q ((N/2)-1 downto 1) := q (10 downto 0);
        q (0) := not r ((N/2) + 1);
        end loop;
        return q;
        end sqrt_1;
begin
    sq_root <= sqrt_1 (d => input);
end behav;
```

SQRT.vhd

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity dividor is generic (N : integer := UD);
port(
    a, b : in  signed (N-1 downto 0);
    c    : out signed (N-1 downto 0));
end dividor;
architecture behavioral of dividor is
signal a_shifted : signed (N+8-1 downto 0);
signal b_shifted : signed (N+8-1 downto 0);
signal c_shifted : signed (N+8-1 downto 0);
begin
    a_shifted <= a & "00000000";
    b_shifted <= "00000000" & b;
    c_shifted <= a_shifted / b_shifted ;
    c <= c_shifted (N-1 downto 0);
end behavioral;
```

dividor.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
entity comparator is generic (width : integer := UD; points_num : integer := UD);
port(
    clk, rst, en : in std_logic;
    thr_dis      : in unsigned (width-1 downto 0);
    dist_1       : in unsigned (2*width-1 downto 0);
    dist_2       : in unsigned (2*width-1 downto 0);
    InLier_num   : out integer range 0 to points_num;
    DIS_to_inl   : out unsigned (2*width-1 downto 0));
end comparator;
architecture behavioral of comparator is
signal DIS_1, DIS_2 : unsigned (2*width-1 downto 0);
signal counter: integer range 0 to points_num := 0;
signal count_1: integer range 0 to points_num := 0;
signal count_2: integer range 0 to points_num := 0;
signal thr: unsigned (2*width-1 downto 0);
begin
thr <= "00000000" & thr_dis & "0000";
counter <= count_1 + count_2;
InLier_num <= counter;
DIS_to_inl <= DIS_1 + DIS_2;
distance_1: process (rst, en, dist_1)
begin
    if ((rst = '1') or (en = '0')) then
        count_1 <= 0;
        DIS_1 <= (others => '0');
    elsif (en = '1') then
        if (counter < points_num) then
            if (dist_1 <= thr) then
                count_1 <= count_1 + 1;
                DIS_1 <= DIS_1 + dist_1;
            end if;
        end if;
    end if;
end process;
distance_2: process (rst, en, dist_2)
begin
    if ((rst = '1') or (en = '0')) then
        count_2 <= 0;
        DIS_2 <= (others => '0');
    elsif (en = '1') then
        if (counter < points_num) then
            if (dist_2 <= thr) then
                count_2 <= count_2 + 1;
                DIS_2 <= DIS_2 + dist_2;
            end if;
        end if;
    end if;
end process;
end behavioral;
```
comparator.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
entity line_comparator is generic (width : integer := UD; points_num : integer := UD);
port(
    clk, rst, en    : in std_logic;
    L               : in integer range 0 to ((points_num**2)*2) := 0;
    a,b             : in signed (width-1 downto 0);
    c               : in signed (2*width-1 downto 0);
    Dis             : in unsigned (2*width-1 downto 0);
    InLier_num      : in integer range 0 to points_num := 0;
    max_L           : out integer range ((points_num**2)*2) downto 0;
    max_a,max_b     : out signed (width-1 downto 0);
    max_c           : out signed (2*width-1 downto 0);
    max_Dis         : out unsigned (2*width-1 downto 0);
    max_InLier_num  : out integer range 0 to (points_num));
end line_comparator;
architecture behavioral of line_comparator is
signal inter_L          : integer range ((points_num**2)*2) downto 0:= 0;
signal inter_a,inter_b  : signed (width-1 downto 0) := (others => '0');
signal inter_c          : signed (2*width-1 downto 0) := (others => '0');
signal inter_Dis        : unsigned (2*width-1 downto 0) := (others => '0');
signal inter_InLier_num : integer range 0 to points_num := 0;
begin
    max_L <= inter_L;
    max_a <= inter_a;
    max_b <= inter_b;
    max_c <= inter_c;
    max_Dis <= inter_Dis;
    max_InLier_num <= inter_InLier_num;
process (clk)
begin
    if (rising_edge (clk)) then
        if (rst = '1') then
            inter_L <= 0;
            inter_a <= (others => '0');
            inter_b <= (others => '0');
            inter_c <= (others => '0');
            inter_Dis <= (others => '0');
            inter_InLier_num <= 0;
        elsif (en = '1') then
            if ((InLier_num > inter_InLier_num) or
((InLier_num = inter_InLier_num) and (Dis < inter_Dis))) then
            inter_L <= L;
            inter_a <= a;
            inter_b <= b;
            inter_c <= c;
            inter_Dis <= Dis;
            inter_InLier_num <= InLier_num;
            end if;
        end if;
    end if;
end process;
end Behavioral;
```
line_comparator.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
use std.textio.all;
use ieee.std_logic_textio.all;
entity points_ram is generic (data_w : integer := UD; addr_w : integer := UD);
port(
    addr_a, b : in integer range 0 to (2**addr_w) := 0;
    clk        : in std_logic;
    q_a, q_b   : out std_logic_vector (data_w-1 downto 0) := (others => '0');
end points_ram;
architecture rtl of points_ram is
    subtype word_t is std_logic_vector (data_w-1 downto 0);
    type RamType is array (0 to 2**addr_w-1) of word_t;
    impure function InitRamFromFile (RamFileName : in string) return RamType is
        file RamFile : text is in RamFileName;
        variable RamFileLine : line;
        variable RAM : RamType;
        begin
        for I in RamType'range loop
        readline (RamFile, RamFileLine);
        read (RamFileLine, RAM(I));
        end loop;
        return RAM;
    end function;
    signal RAM : RamType := InitRamFromFile ("C:\");
begin
    process(clk)
    begin
        if (falling_edge (clk)) then
            q_a <= RAM(addr_a);
        end if;
    end process;
    process (clk)
    begin
        if (falling_edge (clk)) then
            q_b <= RAM(addr_b);
        end if;
    end process;
end rtl;
```
points_ram.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity main is
generic (points_num: integer := UD; P_addr_w: integer := UD;
P_data_w : integer := UD; width : integer := UD);
port(
    clk, rst    : in std_logic;
    thr_dis     : in unsigned (width-1 downto 0);
    a_val,b_val : out signed (width-1 downto 0);
    c_val       : out signed (2*width-1 downto 0);
    valid       : out std_logic);
end main;
architecture behav of main is
component distance is generic (width : integer := width);
port(
    clk  : in std_logic;
    rst  : in std_logic;
    en   : in std_logic;
    x,y  : in signed (width-1 downto 0);
    a,b  : in signed (width-1 downto 0);
    c    : in signed (2*width-1 downto 0);
    dist : out unsigned (2*width-1 downto 0) := (others => '0'));
end component;
component parameters is generic (width : integer := width);
port(
    clk          : in std_logic;
    rst          : in std_logic;
    en           : in std_logic;
    x1,y1,x2,y2 : in signed (width-1 downto 0);
    a,b          : out signed (width-1 downto 0);
    c            : out signed (2*width-1 downto 0));
end component;
component comparator is
generic (width : integer := width; points_num : integer := UD);
port(
    clk         : in std_logic;
    rst         : in std_logic;
    en          : in std_logic;
    thr_dis     : in unsigned (width-1 downto 0);
    dist_1      : in unsigned (2*width-1 downto 0);
    dist_2      : in unsigned (2*width-1 downto 0);
    InLier_num : out integer range 0 to points_num ;
    DIS_to_inl : out unsigned (2*width-1 downto 0));
end component;
component line_comparator is generic (width : integer := width;
points_num : integer := UD);
port(
    clk, rst, en   : in std_logic;
    L              : in integer range ((points_num**2)*2) downto 0 := 0;
    a,b            : in signed (width-1 downto 0);
    c              : in signed (2*width-1 downto 0);
    Dis            : in unsigned (2*width-1 downto 0);
    InLier_num     : in integer range points_num downto 0 := 0;
    max_L          : out integer range ((points_num**2)*2) downto 0;
    max_a,max_b    : out signed (width-1 downto 0);
    max_c          : out signed (2*width-1 downto 0);
    max_Dis        : out unsigned (2*width-1 downto 0);
    max_InLier_num : out integer range 0 to points_num);
end component;
```

```
component points_ram is generic (data_w : integer := UD; addr_w : integer := UD);
port(
    addr_a, addr_b : in integer range 0 to (2**addr_w) := 0;
    clk       : in std_logic;
    q_a, q_b : out std_logic_vector (data_w-1 downto 0) := (others => '0');
end component;
component FSM is generic (data_w : integer := UD; P_addr_w : integer := UD;
points_num : integer := UD);
port(
    clk, rst : in std_logic;
    para_en, dist_en, com_en, line_com_en : out std_logic;
    P0_addr, P1_addr : out integer range 0 to (2**P_addr_w) := 0;
    computation_done : out std_logic;
    L                : out integer range ((points_num**2)*2) downto 0 := 0);
end component;
signal P0_addr, P1_addr : integer range 0 to (2**P_addr_w) := 0;
signal P0_data, P1_data : std_logic_vector (P_data_w-1 downto 0);
signal P0_data_reg, P1_data_reg : std_logic_vector (P_data_w-1 downto 0);
signal c, c_reg     : signed (2*width-1 downto 0);
signal a_reg,b_reg : signed (width-1 downto 0);
signal para_en, dist_en : std_logic;
signal dist_1, dist_2 : unsigned (2*width-1 downto 0);
signal dist_1_reg, dist_2_reg : unsigned (2*width-1 downto 0);
signal computation_done, com_en, line_com_en : std_logic;
signal InLier_num, InLier_num_reg : integer range 0 to points_num := 0;
signal DIS_to_inl, DIS_to_inl_reg : unsigned (2*width-1 downto 0);
signal x : integer range points_num downto 0 := 0;
signal y : integer range points_num downto 0 := 1;
signal line_num : integer range ((points_num**2)*2) downto 0 := 0;
signal max_L          : integer range ((points_num**2)*2) downto 0;
signal max_a,max_b    : signed (width-1 downto 0) := (others => '0');
signal max_c          : signed (2*width-1 downto 0) := (others => '0');
signal max_Dis        : unsigned (2*width-1 downto 0);
signal max_InLier_num : integer range 0 to points_num;
begin
point_ram: points_ram generic map (data_w => P_data_w, addr_w => P_addr_w)
port map
(clk => clk, addr_a => P0_addr, addr_b => P1_addr, q_a => P0_data, q_b => P1_data);
param: parameters generic map (width => width)
port map
    (
    clk => clk, rst => rst, en => para_en,
    x1 => signed (P0_data_reg (P_data_w-1 downto P_data_w/2)),
    y1 => signed (P0_data_reg (P_data_w/2-1 downto 0)),
    x2 => signed (P1_data_reg (P_data_w-1 downto P_data_w/2)),
    y2 => signed (P1_data_reg (P_data_w/2-1 downto 0)),
    a => a, b => b, c => c);
dis_1: distance generic map (width => width)
port map
    (
    clk => clk, rst => rst, en => dist_en,
    x => signed (P0_data_reg(P_data_w-1 downto P_data_w/2)),
    y => signed (P0_data_reg(P_data_w/2-1 downto 0)),
    a => a_reg, b => b_reg, c => c_reg,
    dist => dist_1 (2*width-1 downto 0));
```

```
dis_2: distance generic map (width => width)
port map
    (
    clk => clk, rst => rst, en => dist_en ,
    x => signed (P1_data_reg (P_data_w-1 downto P_data_w/2)),
    y => signed (P1_data_reg (P_data_w/2-1 downto 0)),
    a => a_reg, b => b_reg, c => c_reg,
    dist => dist_2 (2*width - 1 downto 0));

comp: comparator generic map (width => width, points_num => points_num)
port map
    (
    clk => clk, rst => rst,
    thr_dis => thr_dis,
    en => com_en,
    dist_1 => dist_1_reg,
    dist_2 => dist_2_reg,
    InLier_num => InLier_num,
    DIS_to_inl => DIS_to_inl);
fsm_1: FSM generic map (P_addr_w => P_addr_w, points_num => points_num)
port map
    (
    clk => clk, rst => rst,
    para_en => para_en,
    dist_en => dist_en,
    com_en => com_en,
    line_com_en => line_com_en,
    P0_addr => P0_addr,
    P1_addr => P1_addr,
    computation_done => computation_done,
    L => line_num);
line_com: line_comparator generic map (width => width, points_num => points_num)
port map
    (
    clk => clk, rst => rst,
    en  => line_com_en,
    L => line_num,
    a => a_reg, b => b_reg, c => c_reg,
    Dis => DIS_to_inl_reg,
    InLier_num => InLier_num_reg,
    max_L => max_L,
    max_a => max_a,
    max_b => max_b,
    max_c => max_c,
    max_Dis => max_Dis,
    max_InLier_num => max_InLier_num);
ram_reg: process (clk)
```

```
begin
    if (rising_edge (clk)) then
        if (rst = '1') then
            P0_data_reg     <= (others => '0');
            P1_data_reg     <= (others => '0');
            a_reg           <= (others => '0');
            b_reg           <= (others => '0');
            c_reg           <= (others => '0');
            dist_1_reg      <= (others => '0');
            dist_2_reg      <= (others => '0');
            InLier_num_reg  <= 0;
            DIS_to_inl_reg  <= (others => '0');
       else
            P0_data_reg     <= P0_data;
            P1_data_reg     <= P1_data;
            a_reg           <= a;
            b_reg           <= b;
            c_reg           <= c;
            dist_1_reg      <= dist_1;
            dist_2_reg      <= dist_2;
            InLier_num_reg  <= InLier_num;
            DIS_to_inl_reg  <= DIS_to_inl;
        end if;
    end if;
end process;
proc: process (clk)
    begin
        if (rising_edge (clk)) then
            if (rst = '1') then
                a_val <= (others => '0');
                b_val <= (others => '0');
                c_val <= (others => '0');
                valid <= '0';
            elsif (computation_done = '1') then
                a_val <= max_a;
                b_val <= max_b;
                c_val <= max_c;
                valid <= '1';
            end if;
        end if;
    end process;
end architecture;
```

main.vhd

```
library ieee;
use ieee.STD_LOGIC_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
use std.textio.all;
use ieee.std_logic_textio.all;
entity FSM is
generic(data_w : integer := UD; P_addr_w : integer := UD; points_num : integer := UD);
port(
    clk, rst         : in std_logic;
    para_en          : out std_logic;
    dist_en          : out std_logic;
    com_en           : out std_logic;
    line_com_en      : out std_logic;
    P0_addr          : out integer range 0 to (2**P_addr_w) := 0;
    P1_addr          : out integer range 0 to (2**P_addr_w) := 0;
    computation_done : out std_logic;
    L                : out integer range ((points_num**2)*2) downto 0);
end FSM;
architecture rtl of FSM is
signal line_num : integer range ((points_num**2)*2) downto 0 := 0;
signal x : integer range (points_num + 2) downto 0 := 0;
signal y : integer range (points_num + 2) downto 0 := 1;
signal s : integer range (points_num/2 + 2) downto 0 := 0;
signal m : integer range (points_num + 10) downto 0 := 0;
signal n : integer range (points_num + 10) downto 0 := 1;
signal state, nextstate : std_logic_vector (3 downto 0);
begin
L <= line_num;
FSM: process (state)
begin
    case (state) is
        when "0000" =>
            para_en             <= '0';
            dist_en             <= '0';
            com_en              <= '0';
            line_com_en         <= '0';
            computation_done    <= '0';
            P0_addr             <=  0;
            P1_addr             <=  0;
            nextstate <= "0010";
        when "0001" =>
            para_en             <= '0';
            dist_en             <= '0';
            com_en              <= '0';
            line_com_en         <= '0';
            computation_done    <= '0';
            if ((x = points_num-1)) then
                P0_addr         <=  0;
                P1_addr         <=  0;
                nextstate       <= "1001";
              else
                P0_addr         <= x;
                P1_addr         <= y;
                nextstate       <= "0011";
                y               <= y;
                line_num        <= line_num;
            end if;
        when "0010" =>
            nextstate <= "0001";
```

```
        when "0011" =>
            para_en             <= '1';
            dist_en             <= '0';
            com_en              <= '0';
            line_com_en         <= '0';
            computation_done    <= '0';
            P0_addr             <=  m;
            P1_addr             <=  n;
            m <= m + 2;
            n <= n + 2;
            nextstate <= "0100";
        when "0100" =>
            para_en             <= '0';
            dist_en             <= '1';
            com_en              <= '1';
            line_com_en         <= '0';
            computation_done    <= '0';
            P0_addr             <=  m;
            P1_addr             <=  n;
            m <= m + 2;
            n <= n + 2;
            s <= s + 1;
            if ((s = (points_num/2)-2) and
            (y < points_num-1)) then
            nextstate <= "0110";
              elsif ((s = (points_num/2)-2) and
            (y >= points_num-1)) then
            nextstate <= "1000";
              else nextstate <= "0101";
            end if;
        when "0101" =>
            nextstate <= "0100";
        when "0110" =>
            para_en             <= '0';
            dist_en             <= '0';
            com_en              <= '1';
            line_com_en         <= '0';
            computation_done    <= '0';
            P0_addr             <=  0;
            P1_addr             <=  0;
            nextstate <= "0111";
        when "0111" =>
            para_en             <= '0';
            dist_en             <= '0';
            com_en              <= '0';
            line_com_en         <= '1';
            computation_done    <= '0';
            P0_addr             <=  0;
            P1_addr             <=  0;
            m <= 0;
            n <= 1;
            s <= 0;
            y <= y + 1;
            line_num <= line_num + 1;
            nextstate <= "0001";
```

```
        when "1000" =>
            para_en             <= '0';
            dist_en             <= '0';
            com_en              <= '0';
            line_com_en         <= '1';
            computation_done    <= '0';
            P0_addr             <=  0;
            P1_addr             <=  0;
            m <= 0;
            n <= 1;
            s <= 0;
            x <= x + 1;
            y <= x + 2;
            line_num <= line_num + 1;
            nextstate <= "0001";
        when "1001" =>
            para_en             <= '0';
            dist_en             <= '0';
            com_en              <= '0';
            line_com_en         <= '0';
            computation_done    <= '1';
            P0_addr             <=  0;
            P1_addr             <=  0;
            nextstate <= "1001";
        when others =>
            para_en             <= '0';
            dist_en             <= '0';
            com_en              <= '0';
            line_com_en         <= '0';
            computation_done    <= '0';
            P0_addr             <=  0;
            P1_addr             <=  0;
            nextState <= "0000";
    end case;
end process fsm;
states: process (clk, rst)
begin
    if (rising_edge (clk)) then
        if (rst = '1') then
            state <= "0000";
        else
            state <= nextstate;
        end if;
    end if ;
end process states;
end architecture;
```
FSM.vhd

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity RANSAC_opt1_tb is
end RANSAC_opt1_tb;
architecture simulate of RANSAC_opt1_tb is
constant width : integer := UD;
constant clk_period: time := 10 ns;
component main
generic (points_num: integer := UD; P_addr_w: integer := UD;
P_data_w : integer := 24; width : integer := UD);
port
    (
     clk, rst        : in  std_logic;
     thr_dis         : in  unsigned (width-1 downto 0);
     a_val,b_val     : out signed (width-1 downto 0);
     c_val           : out signed (2*width-1 downto 0);
     valid           : out std_logic);
end component;
signal clk, rst     : std_logic;
signal thr_dis      : unsigned (width-1 downto 0);
signal a_val,b_val : signed (width-1 downto 0);
signal c_val        : signed (2*width-1 downto 0);
signal valid        : std_logic;
begin
dut: main generic map (points_num => UD, P_addr_w => UD, P_data_w => UD, width => UD)
port map
    (
     clk => clk,
     rst => rst,
     thr_dis => thr_dis,
     a_val => a_val,
     b_val => b_val,
     c_val => c_val,
     valid => valid);
clock: process
begin
      clk <= '0';
      wait for clk_period/2;
      clk <= '1' ;
      wait for clk_period/2;
        if (valid = '1') then wait;
      end if;
end process clock;
stimulus: process
begin
      thr_dis <= "UD";
      rst <= '1';
      wait for 1.5*clk_period;
      rst <= '0';
      wait until (valid = '1');
      wait;
end process;
END simulate;
```
RANSAC_opt1_tb.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
entity comparator is generic (width : integer := UD);
port
   (
    clk       : in std_logic;
    rst       : in std_logic;
    en        : in std_logic;
    thr_dis   : in unsigned (width-1 downto 0);
    dist_1    : in unsigned (2*width-1 downto 0);
    dist_2    : in unsigned (2*width-1 downto 0);
    IN_1_val : out std_logic := '0';
    IN_2_val : out std_logic := '0');
end comparator;
architecture Behavioral of comparator is
signal thr: unsigned (2*width-1 downto 0);
begin
thr <= "00000000" & thr_dis & "0000";
distance_1: process (rst, en, dist_1)
begin
    if ((rst = '1') or (en = '0')) then
            IN_1_val <= '0';
    elsif (en = '1') then
        if (dist_1 <= thr) then
            IN_1_val <= '1';
          else
            IN_1_val <= '0';
          end if;
      end if;
end process;
distance_2: process (rst, en, dist_2)
begin
    if ((rst = '1') or (en = '0')) then
            IN_2_val <= '0';
    elsif (en = '1') then
        if (dist_2 <= thr) then
            IN_2_val <= '1';
        else
            IN_2_val <= '0';
        end if;
    end if;
end process;
end behavioral;
```
comparator.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
use work.ram_pkg.all;
use std.textio.all;
use ieee.std_logic_textio.all;
entity points_ram is generic (data_num : integer := UD; data_w : integer := UD);
port(
    x_array: out RamType (data_num-1 downto 0);
    y_array: out RamType (data_num-1 downto 0));
end points_ram;
architecture rtl of points_ram is
    impure function InitRamFromFile (RamFileName : in string) return RamType is
        file RamFile : text is in RamFileName;
        variable RamFileLine : line;
        variable RAM : RamType (data_num-1 downto 0);
begin
        for I in 0 to (data_num-1) loop
            readline (RamFile, RamFileLine);
            read (RamFileLine, RAM(I));
        end loop;
    return RAM;
    end function;
signal RAM_x : RamType (data_num-1 downto 0) := InitRamFromFile ("C:\");
signal RAM_y : RamType (data_num-1 downto 0) := InitRamFromFile ("C:\");
begin
             x_array <= RAM_x;
             y_array <= RAM_y;
end rtl;
```

points_ram.vhd

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.NUMERIC_STD.all;
use std.textio.all;
use ieee.std_logic_textio.all;
entity FSM is
port(
	clk, rst : in std_logic;
	para_en, dist_en, com_en, count_en : out std_logic;
	);
end FSM;
architecture rtl of FSM is
signal state, nextstate : std_logic_vector (3 downto 0);
begin
FSM: process (state)
begin
	case(state) is
		when "0000" =>
			dist_en     <= '0';
			com_en      <= '0';
			count_en    <= '0';
			para_en     <= '0';
			nextstate   <= "0001";
		when "0001" =>
			dist_en     <= '1';
			com_en      <= '1';
			count_en    <= '0';
			para_en     <= '1';
			nextstate   <= "0010";
		when "0010" =>
			dist_en     <= '1';
			com_en      <= '1';
			count_en    <= '0';
			para_en     <= '1';
			nextstate   <= "0011";
		when "0011" =>
			dist_en     <= '1';
			com_en      <= '1';
			count_en    <= '1';
			para_en     <= '1';
			nextstate   <= "0011";
		when others =>
			dist_en     <= '0';
			com_en      <= '0';
			count_en    <= '0';
			para_en     <= '0';
			NextState   <= "0000";
	end case;
end process FSM;
states: process (clk, rst)
begin
	if (rising_edge (clk)) then
		if (rst = '1') then
			state <= "0000";
		else
			state <= nextstate;
		end if;
	end if ;
end process states;
end architecture;
```

FSM.vhd

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
package ram_pkg is
constant width : integer := UD;
type RamType is array (integer range <>) of std_logic_vector (width-1 downto 0);
type signed_array_1 is array (integer range <>) of signed (width-1 downto 0);
type signed_array_2 is array (integer range <>) of signed (2*width-1 downto 0);
type unsigned_2D_2 is array (integer range <>, integer range <>) of
unsigned (2*width-1 downto 0);
type unsigned_array_2 is array (integer range <>) of unsigned (2*width-1 downto 0);
type std_logic_2D is array (integer range <>, integer range <>) of std_logic;
type integer_array is array (integer range <>) of integer;
end package;
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.ram_pkg.all;
entity main is generic (points_num: integer := UD; width : integer := UD);
port(
    clk, rst     : in  std_logic;
    thr_dis      : in  unsigned (width-1 downto 0);
    a_val,b_val : out signed (width-1 downto 0);
    c_val        : out signed (2*width-1 downto 0);
    line_num     : out integer range ((points_num**2)*2) downto 0 := 0;
    InLier_num   : out integer range 0 to points_num := 0;
    L_INL_dist   : out unsigned (2*width-1 downto 0);
    Valid        : out std_logic);
end main;
architecture behav of main is
component distance is generic (width : integer := width);
port(
    clk, rst, en: in std_logic;
    x,y          : in signed (width-1 downto 0);
    a,b          : in signed (width-1 downto 0);
    c            : in signed (2* width-1 downto 0);
    dist         : out unsigned (2*width-1 downto 0) := (others =>'0'));
end component;
component parameters is generic (width : integer := width);
port(
    clk, rst, en: in std_logic;
    x1,y1,x2,y2 : in signed (width-1 downto 0);
    a,b          : out signed (width-1 downto 0);
    c            : out signed (2*width-1 downto 0));
end component;
component comparator is generic (width : integer := UD);
port(
    clk, rst, en: in std_logic;
    thr_dis      : in unsigned (width-1 downto 0);
    dist_1       : in unsigned (2*width-1 downto 0);
    dist_2       : in unsigned (2*width-1 downto 0);
    IN_1_val     : out std_logic;
    IN_2_val     : out std_logic);
end component;
component points_ram is generic (data_num : integer := UD; data_w : integer := UD);
port(
    x_array: out RamType (data_num-1 downto 0);
    y_array: out RamType (data_num-1 downto 0));
end component;
```

```
component FSM is
port(
    clk, rst : in std_logic;
    para_en, dist_en, com_en, count_en : out std_logic);
end component;
function count (num : std_logic_2D; en : std_logic) return integer_array is
variable counter : integer_array (points_num*points_num-1 downto 0) := (others => 0);
begin
    if (en = '1') then
        for i in 0 to (points_num*points_num-1) loop
        for j in 0 to (points_num-1) loop
            if num(i,j) = '1' then
                counter(i) := counter(i) + 1;
            end if;
        end loop;
        end loop;
    end if;
return counter;
end function count;
signal a, b : signed_array_1 (points_num*points_num-1 downto 0)
:= (others => "000000000000");
signal c : signed_array_2 (points_num*points_num-1 downto 0)
:= (others => "000000000000000000000000");
signal dist_to_points : unsigned_2D_2 (points_num*points_num-1 downto 0,
points_num-1 downto 0);
signal dist_en, com_en : std_logic := '0';
signal count_en,para_en : std_logic := '0';
signal IN_val : std_logic_2D (points_num*points_num-1 downto 0,
points_num-1 downto 0);
signal val_count : integer_array (points_num*points_num-1 downto 0);
signal point_x, point_y : RamType (points_num-1 downto 0);
signal dist_to_inliers : unsigned_array_2 (points_num*points_num-1 downto 0);
signal max_line_num : integer range ((points_num**2)*2) downto 0 := 0;
signal max_a, max_b : signed (width-1 downto 0) := (others => '0');
signal max_c : signed (2*width-1 downto 0) := (others => '0');
signal max_InLier_num : integer range 0 to points_num := 0;
signal max_dist : unsigned (2*width-1 downto 0) := (others => '0');
begin
ram: points_ram generic map (data_num => points_num, data_w => width)
port map (x_array => point_x, y_array => point_y);
fsm_1: FSM port map (
    clk => clk,
    rst => rst,
    para_en => para_en,
    dist_en => dist_en,
    com_en => com_en,
    count_en => count_en);
GEN_x: for i in 0 to (points_num-1) generate
GEN_y: for j in (i + 1) to (points_num-1) generate
    param: parameters generic map (width => width)
    port map (clk => clk, rst => rst, en => para_en,
    x1 => signed (point_x(i)), y1 => signed (point_y(i)),
    x2 => signed (point_x(j)), y2 => signed (point_y(j)),
    a => a (points_num*i + j), b => b(points_num*i + j), c => c(points_num*i + j));
    dist => dist_to_points (points_num*i + j, 2*m));
GEN_m: for m in 0 to (points_num/2-1) generate
```

```
dis_1: distance generic map (width => width)
    port map (clk => clk, rst => rst, en => dist_en,
    x => signed (point_x (2*m)),
    y => signed (point_y (2*m)),
    a => a (points_num*i + j), b => b(points_num*i + j), c => c(points_num*i + j),
    dist => dist_to_points(points_num*i + j, 2*m));
    dis_2: distance generic map (width => width)
    port map (clk => clk, rst => rst, en => dist_en,
    x => signed (point_x (2*m + 1)),
    y => signed (point_y (2*m + 1)),
    a => a (points_num*i + j), b => b(points_num*i + j), c => c(points_num*i + j),
    dist => dist_to_points(points_num*i + j, 2*m + 1));
    comp: comparator generic map (width => width)
    port MAP (clk => clk, rst => rst,
    thr_dis => thr_dis,
    en => com_en,
    dist_1 => dist_to_points (points_num*i + j, 2*m),
    dist_2 => dist_to_points (points_num*i + j, 2*m + 1),
    N_1_val => IN_val (points_num*i + j, 2*m),
    IN_2_val => IN_val (points_num*i + j, 2*m + 1));
    end generate GEN_m;
end generate GEN_y;
end generate GEN_x;
val_count <= count (IN_val, count_en);
search: process (clk)
begin
    if (rising_edge (clk)) then
    searching: for s in 0 to (points_num*points_num-1) loop
cond:   if (val_count(s) > max_InLier_num) then
            max_line_num <= s;
            max_a <= a(s);
            max_b <= b(s);
            max_c <= c(s);
            max_InLier_num <= val_count(s);
        end if;
        end loop;
    end if;
end process;
proc: process (clk)
begin
    if (rising_edge (clk)) then
        if (rst = '1') then
            a_val <= (others => '0');
            b_val <= (others => '0');
            c_val <= (others => '0');
            line_num <= 0;
            InLier_num <= 0;
            L_INL_dist <= (others => '0');
            valid <= '0';
        else
            a_val <= max_a;
            b_val <= max_b;
            c_val <= max_c;
            line_num <= max_line_num;
            InLier_num <= max_InLier_num;
            L_INL_dist <= max_dist;
            valid <= '1';
        end if;
    end if;
end process;
end architecture;
```
main.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity RANSAC_opt2_tb is
end RANSAC_opt2_tb;
architecture simulate of RANSAC_opt2_tb is
constant width : integer := UD;
constant points_num: integer := UD;
constant clk_period: time := 10 ns;
component main generic (points_num: integer := UD; width : integer := UD);
port(
    clk, rst          : in std_logic;
    thr_dis           : in unsigned (width-1 downto 0);
    a_val, b_val      : out signed (width-1 downto 0);
    c_val             : out signed (2*width-1 downto 0);
    line_num          : out integer range ((points_num**2)*2) downto 0 := 0;
    InLier_num        : out integer range 0 to points_num := 0;
    L_INL_dist        : out unsigned (2*width-1 downto 0);
    valid             : out std_logic);
end component;
signal clk, rst       : std_logic;
signal thr_dis        : unsigned (width-1 downto 0);
signal a_val, b_val   : signed (width-1 downto 0);
signal c_val          : signed (2*width-1 downto 0);
signal valid          : std_logic;
signal line_num       : integer range ((points_num**2)*2) downto 0 := 0;
signal InLier_num     : integer range 0 to points_num := 0;
signal L_INL_dist     : unsigned (2*width-1 downto 0);
begin
dut: main generic map (points_num => points_num, width => width)
port map(
        clk => clk,
        rst => rst,
        thr_dis => thr_dis,
        a_val => a_val,
        b_val => b_val,
        c_val => c_val,
        line_num => line_num,
        InLier_num => InLier_num,
        L_INL_dist => L_INL_dist,
        valid => valid);
    clock: process
        begin
            clk <= '0';
            wait for clk_period/2;
            clk <= '1' ;
            wait for clk_period/2;
    end process clock;
    stimulus: process
        begin
            thr_dis <= "UD";
            rst <= '1';
            wait for 1.5*clk_period;
            rst <= '0';
            wait until (valid = '1');
            wait;
    end process;
end simulate;
```
RANSAC_opt2_tb.vhd

Βιβλιογραφία

1. An FPGA-based hardware accelerator of ransac algorithm for matching of images feature points, Z. Zhao, F. Wang, Q. Ni, 2019. 1

2. FPGA accelerator for real-time SIFT matching with RANSAC support, J. Vourvoulakis, J. Kalomiros, J. Lygouras, 2017. 1

3. Acceleration of RANSAC algorithm for images with affine transformation, J. Vourvoulakis, J. Kalomiros, J. Lygouras, 2016. 1

4. FPGA- FPGA-based architecture of a real-time SIFT matcher and RANSAC algorithm for robotic vision applications, J. Vourvoulakis, J. Kalomiros, J. Lygouras, 2018. 1

5. FPGA implementation of a feature detection and tracking algorithm for real-time applications, B. Tippetts, S. Fowers, K. Lillywhite, D. Lee, J. Archibald, 2007. 1

6. FPGA implementation of RANSAC algorithm for real-time image geometry estimation, J. Tang, N. Shaikh-Husin, U. Sheikh, 2013. 1

7. Comparative evaluation of FPGA implementation alternatives for real-time robust ellipse estimation based on RANSAC algorithm, T. Thu, J. Hamamura, R. Soejima, Y. Shibata, K. Oguri, 2017. 1

8. FPGA-based robust ellipse estimation for circular road sign detection, S. Martelli, R. Marzotto, A. Colombari, V. Murino, 2010. 1

9. Deep-pipelined FPGA implementation of ellipse estimation for eye tracking, K. Dohi, Y. Hatanaka, K. Negi, Y. Shibata, K. Oguri, 2012. 1

10. https://esm.uoi.gr/wp-content/uploads/2019/06/

11. http://papdes.webpages.auth.gr/portal/images/Pdf/Eutheia2.2.pdf. 3

12. Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography, M. A. Fischler & R. C. Bolles, 1981. 4

13. https://en.wikipedia.org/wiki/Hough_transform. 10

14. Energy-based Geometric Multi-Model Fitting, Hossam Isack, Yuri Boykov, 2012. 10

15. MLESAC: A new robust estimator with application to estimating image geometry, P.H.S. Torr, A. Zisserman, 2000. 11

16. Guided-MLESAC: Faster image transform estimation by using matching priors, B. J. Tordoff, D. W. Murray, 2005. 11

17. Matching with PROSAC – progressive sample consensus, Proceedings of Conference on Computer Vision and Pattern Recognition (San Diego), vol. 1, June 2005, pp. 220–226. 11

18. Randomized RANSAC with Td,d test, O. Chum, J. Matas, 13th British Machine Vision Conference, 2002. 11

19. Preemptive RANSAC for live structure and motion estimation, D. Nistér, 2003. 11

20. Robust adaptive-scale parametric model estimation for computer vision, H. Wang, D. Suter, 2004. 11

21. Robust multiple structures estimation with jlinkage, R. Toldo, A. Fusiello, 2008. 11

22. KALMANSAC: Robust filtering by consensus, A. Vedaldi, H. Jin, P. Favaro, S. Soatto, 2005. 11

23. Prabhuram Gopalan Xin Wu and Greg Lara. Xilinx Next Generation 28 nmFPGA Technology Overview.WP312 (v1.1), March 26, 2011. 14

24. Altera Corporation. Introducing Innovations at 28 nm to Move Beyond Moores Law. White paper, WP-01125-1.2, June, 2012. 14

25. Zhiyi Yang, Yating Zhu, and Yong Pu. Parallel Image Processing Based on CUDA. In Computer Science and Software Engineering, 2008 International Conference on,3, pages 198–201, December 2008. 16

26. S. Asano, T. Maruyama, and Y. Yamaguchi. Performance comparison of FPGA, GPU and CPU in image processing. InInternational Conference on Field Programmable Logic and Applications, 2009. FPL 2009, pages 126–131, September 2009. 16

27. Benyi Shi, Sihai Chen, Feifei Huang, Cheng Wang, and Kun Bi. The Parallel Processing Based on CUDA for Convolution Filter FDK Reconstruction of CT. In Parallel Architectures, Algorithms and Programming (PAAP), 2010 Third International Symposium on, pages 149–153, December 2010. 16

28. Hee Kong Phoon, M. Yap, and Chuan Khye Chai. A Highly Compatible Architecture Design for Optimum FPGA to Structured-ASIC Migration. In IEEE International Conference on Semiconductor Electronics, 2006. ICSE ’06, pages 506–510,December 2006. 16

29. Inc. Berkeley Design Technology. High-Level Synthesis Tools for Xilinx FP-GAs. Berkeley Design Technology, Inc., 20xx. 19

30. Xilinx. Virtex-6 FPGA DSP48E1 Slice. User Guide, UG369 (v1.3), February 14,2011. 19

31. S. Hauck, A. DeHon - Reconfigurable Computing. The Theory and Practice of FPGA-Based Computation. Elsevier, 2008. 24

32. I. Kuon, R. Tessier, J. Rose - FPGA Architecture: Survey and Challenges. Foundations and Trends in Electronics Design Automation, vol. 2, no 2, pp. 135-253, 2007. 24

33. H. Parvez, H. Mehrez - Application-Specific, Mesh-Based, Heterogenous FPGA Architectures. Springer, 2011. 24

34. J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Zhiru Zhang.High-Level Synthesis for FPGAs: From Prototyping to Deployment.IEEETransactions on Computer-Aided Design of Integrated Circuits and Systems,30(4):473–491, April 2011. 34.

35. V. Betz, J. Rose, A. Marquardt - Architecture and CAD for Deep-Submicron FPGAS. Springer, 1999. 40

36. Kazutoshi Wakabayashi.C-based behavioral synthesis and verification analysison industrial design examples. InProceedings of the 2004 Asia and South PacificDesign Automation Conference, ASP-DAC ’04, page 344–348, Piscataway, NJ, USA, 2004. IEEE Press. 34.

37. M. Zuliani, RANSAC for Dummies, 2008-2012.

38. Ι. Καλόμοιρος, Εισαγωγή στη γλώσσα VHDL, 2015.

39. Ι. Βουρβουλάκης, Προηγμένα Ψηφιακά Συστήματα, Εργαστηριακές Ασκήσεις, 2020.