ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΕΠΕΞΕΡΓΑΣΤΩΝ ΚΑΙ ΥΛΙΚΟΥ

ΕΡΓΑΛΕΙΟ ΑΠΕΙΚΟΝΙΣΗΣ ΑΛΓΟΡΙΘΜΩΝ ΣΕ ΑΝΑΔΙΑΤΑΣΣΟΜΕΝΗ ΛΟΓΙΚΗ

ΗΛΙΑΚΗ ΑΝΑΣΤΑΣΙΑ

Τριμελής Επιτροπή

Καθ. Δόλλας Απόστολος
Αν. Καθ. Σαμολαδάς Βασίλειος
Καθ. Πνευματικάτος Διονύσιος (ΕΜΠ)

ΧΑΝΙΑ, 2020

ΠΕΡΙΛΗΨΗ

Τα τελευταία χρόνια οι εφαρμογές γίνονται όλο και πιο σύνθετες σε πολυπλοκότητα και όγκο δεδομένων με αποτέλεσμα να γίνεται επιτακτική η ανάγκη για ανάπτυξη εργαλείων με σκοπό την αποδοτική σε χώρο και χρόνο εκτέλεσή τους. Ιδιαίτερα αποδοτικές στους τομείς αυτούς έχουν αποδειχθεί μεθοδολογίες που βασίζονται στην Αρχιτεκτονική Διαχωρισμένης Πρόσβασης-Εκτέλεσης. Η αρχιτεκτονική αυτή διαχωρίζει την εκτέλεση των πράξεων από τις προσβάσεις στη μνήμη με σκοπό την παράλληλη εκτέλεσή τους για την ελαχιστοποίηση του χρόνου αναμονής δεδομένων. Στα πλαίσια της παρούσας διπλωματικής αναπτύχθηκε εργαλείο που απεικονίζει εφαρμογές σε υλικό με τη χρήση της αρχιτεκτονικής αυτής. Με τη χρήση τροποποιημένου λεκτικού και συντακτικού αναλυτή για lex και yacc διαβάζεται κώδικας C με οδηγία στο πιο σύνθετο κομμάτι του και παράγονται αυτόματα συναρτήσεις fetch και process, οι οποίες μέχρι πρότινος γράφονταν με το χέρι. Οι συναρτήσεις αυτές σε συνδυασμό με μια μικρή επεξεργασία στον κώδικα εισόδου μέσω σύνθεσης στο Vivado HLS προσομοιώνουν την τελική απεικόνιση σε υλικό κάθε εφαρμογής. Η ορθή λειτουργία του εργαλείου επιβεβαιώθηκε με εφαρμογές πράξεων πινάκων και επεξεργασίας εικόνας που προσομοιώθηκαν μέχρι και το σημείο της απεικόνισης σε FPGA.

Περιεχόμενα

ΛΙΣΤΑ ΕΙΚΟΝΩΝ

ΚΕΦΑΛΑΙΟ 1-ΕΙΣΑΓΩΓΗ

Το πρόβλημα της γεφύρωσης της ταχύτητας της μνήμης με αυτή ενός επεξεργαστή απασχολεί τους αρχιτέκτονες υπολογιστών ήδη από την δεκαετία του 1950. Η έννοια της κρυφής μνήμης (cache) και η ακόμη παλαιότερη (από την δεκαετία του 1950) έννοια της πρόχειρης μνήμης (scratchpad memory) είναι ένας μόνο από τους τρόπους για να αξιοποιηθεί καλύτερα η υπολογιστική ισχύς. Ήδη από τις αρχές της δεκαετίας του 1980, οι σχεδιαστές του Intel 8086/8088 χρησιμοποίησαν μία μέθοδο που ονομάζεται αρχιτεκτονική διαχωρισμένης πρόσβασης - εκτέλεσης (Decoupled Access-Execute - DAE). Η κύρια ιδέα στην μέθοδο DAE σε συμβατικές αρχιτεκτονικές υπολογιστών είναι να υπάρχει ένα υποσύστημα που φέρνει εντολές από την μνήμη, ένα ξεχωριστό υποσύστημα που εκτελεί εντολές, και τα δύο υποσυστήματα να επικοινωνούν μέσα από μία κοινή διεπαφή (συνήθως μία ουρά First In First Out queue – FIFO).

Με την εξέλιξη της τεχνολογίας, υψηλού επιπέδου εργαλεία χρησιμοποιούνται για την απεικόνιση αλγορίθμων σε αναδιατασσόμενη λογική (FPGA - Field Programmable Gate Array). Από την οπτική του σχεδιαστή συστημάτων σε τεχνολογία FPGA η γεφύρωση της ταχύτητας μίας κεντρικής μνήμης με το επεξεργαστικό σύστημα είναι παρόμοιο πρόβλημα με εκείνο που έχουν οι αρχιτέκτονες και σχεδιαστές συμβατικών Κεντρικών Επεξεργαστικών Μονάδων (CPU). Η παρούσα διπλωματική διερευνά την εφαρμογή της αρχιτεκτονικής προσέγγισης DAE σε αναδιατασσόμενη λογική, και πιο συγκεκριμένα στοχεύει στην δημιουργία ενός εργαλείου DAE που να μπορεί να ενσωματωθεί σε μία συνηθισμένη σχεδιαστική ροή για αναδιατασσόμενη λογική.

1.1 Περιγραφή του προβλήματος

Τόσο σε συμβατικές αρχιτεκτονικές όσο και σε αρχιτεκτονικές ειδικού σκοπού απεικονισμένες σε αναδιατασσόμενη λογική, υπάρχει κάποια διεργασία (είτε πρόγραμμα, είτε αλγόριθμος απεικονισμένος σε υλικό) που εκτελείται στην υφιστάμενη τεχνολογία (π.χ. CPU, FPGA, ή άλλη). Η διεργασία ουσιαστικά αποτελεί την εκτέλεση ενός προγράμματος όταν η τεχνολογία-στόχος είναι μία CPU [26]. Η εκτέλεση μιας διεργασίας απαιτεί τη χρήση πόρων του υπολογιστή, όπως χρόνο στον επεξεργαστή, αρχεία μνήμης και συσκευές εισόδου/εξόδου. Οι πόροι αυτοί δύναται να δεσμευθούν τόσο κατά τη δημιουργία, όσο και κατά την εκτέλεση της διεργασίας. Όταν όμως, για παράδειγμα κάποια πρόσβαση στη μνήμη απαιτηθεί κατά την εκτέλεση της διεργασίας, αυτό ενδέχεται να προκαλέσει καθυστέρηση στο χρόνο ολοκλήρωσής της. Η προαναφερθείσα προσέγγιση DAE είναι, όπως αναφέρθηκε, ένας τρόπος γεφύρωσης της ταχύτητας της μνήμης με αυτή της CPU.

Σε τεχνολογία αναδιατασσόμενης λογικής, τα σχεδιαστικά εργαλεία έχουν πλέον εξελιχθεί στον βαθμό που η σχεδίαση υλικού, από την πλευρά του σχεδιαστή, έχει

παρόμοιο τρόπο περιγραφής όπως η σχεδίαση ενός προγράμματος σε γλώσσα προγραμματισμού υψηλού επιπέδου (High Level Language - HLL). Αυτό βέβαια δεν αναιρεί το ότι από άποψη ποιότητας σχεδίασης, μεγέθους αλλά και ταχύτητας συστήματος, ο σχεδιαστής συστημάτων υλικού πρέπει να περιγράψει την σχεδίαση με τέτοιο τρόπο που να μπορεί να αποκαλυφθεί ο παραλληλισμός. Εργαλεία όπως το Vivado HLS (High Level Synthesis) της εταιρίας Xilinx επιτρέπουν στον χρήστη την περιγραφή συστημάτων υλικού σε πολύ υψηλό επίπεδο. Παρόλα ταύτα, η επέκταση των εργαλείων αυτών με δυνατότητες που επιτρέπουν την αποτελεσματική δημιουργία συστημάτων από την περιγραφή υψηλού επιπέδου του χρήστη είναι ένα πεδίο όπου υπάρχει ιδιαίτερα έντονη ερευνητική δραστηριότητα, με πολλά εργαλεία τα οποία είτε αυτοματοποιούν μέρος της διαδικασίας, είτε επιτρέπουν την εύκολη συνένωση συστημάτων με χρήση της γλώσσας Python, είτε αποκαλύπτουν παραλληλισμό σε διεργασίες με σκοπό την δημιουργία ενός γράφου που κατόπιν θα απεικονιστεί στους πόρους της FPGA, κλπ. Στο πλαίσιο αυτό, η παρούσα διπλωματική εργασία διερευνά μία κατηγορία εργαλείων που έως τώρα δεν έχει αξιοποιηθεί από την σχεδιαστική κοινότητα σε FPGA – την δημιουργία ενός εργαλείου για εκτέλεση διεργασιών με την μέθοδο DAE.

1.2 Συνεισφορά της διπλωματικής

Ο σκοπός της δημιουργίας ενός εργαλείου για DAE σε αναδιατασσόμενη λογική θα μπορούσε ενδεχόμενα να επιτευχθεί με διαφορετικούς τρόπους, για παράδειγμα, παραμετροποιημένες βιβλιοθήκες όπου η κάθε μία να έχει πολύ συγκεκριμένη λειτουργικότητα. Με στόχο την δημιουργία ενός εργαλείου ευρείας εφαρμογής της μεθόδου DAE σε αναδιατασσόμενη λογική, η παρούσα διπλωματική προσέγγισε το πρόβλημα από την σκοπιά της τροποποίησης ενός λεκτικού και συντακτικού αναλυτή για την γλώσσα C, η έξοδος του οποίου χρησιμεύει σαν είσοδος στο εργαλείο Vivado HLS με σκοπό την υλοποίηση ενός όσο το δυνατόν γενικότερου σκοπού υποσυστήματος σε DAE. Πιο συγκεκριμένα, η συνεισφορά της διπλωματικής εργασίας έγκειται στα εξής:

- Μελέτη των πρωταρχικών δομών για διατύπωση αρχιτεκτονικών στοιχείων DAE και απεικόνισή τους σε επεκτάσεις του συντακτικού της γλώσσας C, με την δημιουργία πρωταρχικών δομών fetch και process για τα υποσυστήματα της πρόσβασης στην μνήμη και της εκτέλεσης, καθώς και των οδηγιών (directives) για την χρήση τους.
- Χρήση των εργαλείων Lex και Yacc για παραγωγή κώδικα C ο οποίος αντιστοιχεί στο DAE μοντέλο, σύμφωνα με την αποσύνθεση των υποσυστημάτων που έχει ορίσει ο χρήστης.
- Εισαγωγή στο εργαλείο Vivado HLS των υποσυστημάτων που δημιουργεί το νέο εργαλείο που αναπτύχθηκε στα πλαίσια της παρούσας διπλωματικής εργασίας.

- Σύνθεση με τη χρήση του από το εργαλείο Vivado HLS των αντίστοιχων υποσυστημάτων, τα οποία στην παρούσα έκδοση πρέπει να διασυνδεθούν μέσω μίας απλής σχεδιαστικής παρέμβασης του χρήστη.
- Επιβεβαίωση λειτουργίας με πραγματικό κώδικα Vivado HLS που δημιουργήθηκε από τα εργαλεία, πέρασε επιτυχώς από την σχεδιαστική ροή του Vivado HLS, και προσομοιώθηκε σε πλακέτα Virtex UltraScale.

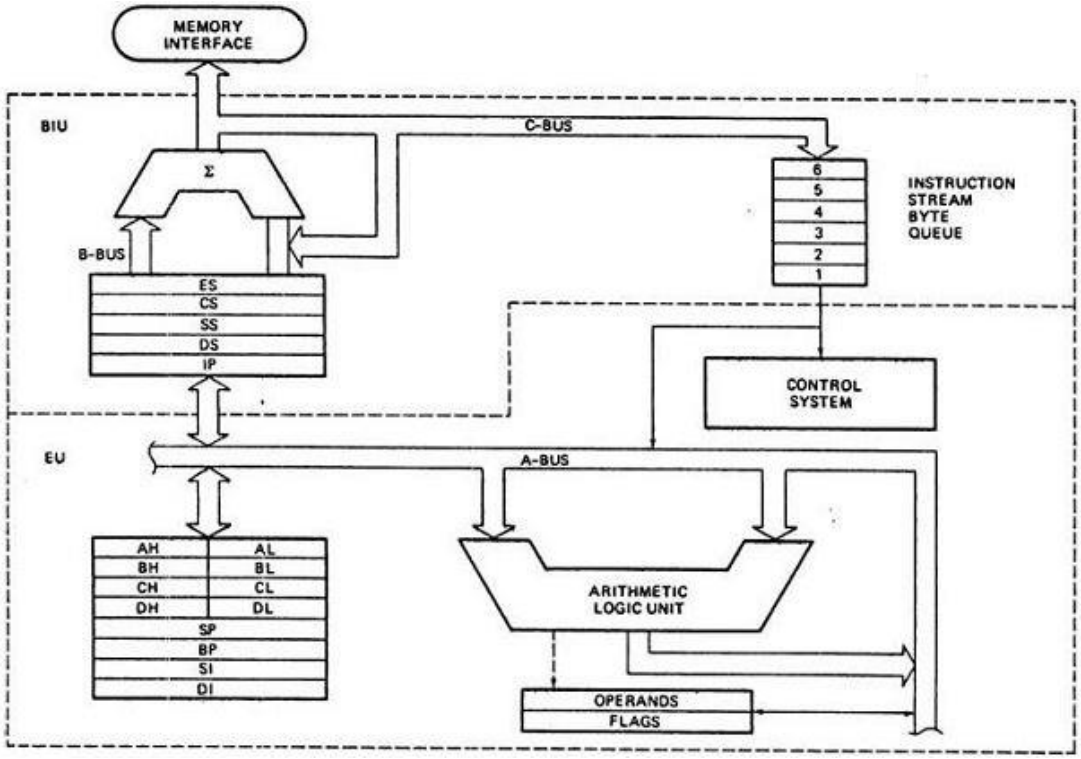
1.3 Οργάνωση της διπλωματικής

Στο κεφάλαιο 2 περιγράφεται σχετική δουλειά άλλων πάνω σε εφαρμογές παράλληλης υλοποίησης υλικού και λογισμικού και αρχιτεκτονικές DAE, καθώς και η έρευνα για την υλοποίηση του εργαλείου που αποτελεί την παρούσα διπλωματική και όλες οι εργασίες που έλαβαν χώρα πριν από την έναρξη της διαδικασίας υλοποίησης της διπλωματικής.
Στο κεφάλαιο 3 περιγράφονται λειτουργικά χαρακτηριστικά των προγραμμάτων που χρησιμοποιήθηκαν, καθώς και τα λειτουργικά χαρακτηριστικά του εργαλείου.
Στο κεφάλαιο 4 αναλύεται η μεθοδολογία η οποία εφαρμόστηκε για την υλοποίηση του εργαλείου.
Στο κεφάλαιο 5 παρουσιάζονται τα πειράματα που πραγματοποιήθηκαν για την επιβεβαίωση της ορθής λειτουργίας του εργαλείου.

ΚΕΦΑΛΑΙΟ 2-ΣΧΕΤΙΚΗ ΕΡΕΥΝΑ

2.1 Ο Μικροεπεξεργαστής Intel 8086

Το 1976 σχεδιάστηκε από την Intel ο Μικροεπεξεργαστής 16-bit 8086, μια ενισχυμένη έκδοση του 8085 [41]. Ήταν ο πρώτος Μικροεπεξεργαστής 16-bit με 16-bit Αριθμητική και Λογική Μονάδα (ALU - Arithmetic Logic Unit), 16-bit καταχωρητές, 16-bit εξωτερικούς δίαυλους δεδομένων, εσωτερικούς δίαυλους δεδομένων και ουρά εντολών με χωρητικότητα 6 bytes εντολών για ταχύτερη επεξεργασία. Επιπλέον, χρησιμοποιεί δύο επίπεδα διοχέτευσης (pipeline), για παράδειγμα για προφόρτωση (fetch) και εκτέλεση (execute) βελτιώνοντας την απόδοση. Η ύπαρξη της ουράς εντολών και της υποστήριξης διοχέτευσης (pipeline), αλλά και η αύξηση του μεγέθους του δίαυλου διεύθυνσης, της μνήμης, του συστήματος εισόδου/εξόδου (I/O), αλλά και του ίδιου του επεξεργαστή, συνιστούν τα ενισχυμένα χαρακτηριστικά του Μικροεπεξεργαστή Intel 8086 σε σχέση με τον 8085.

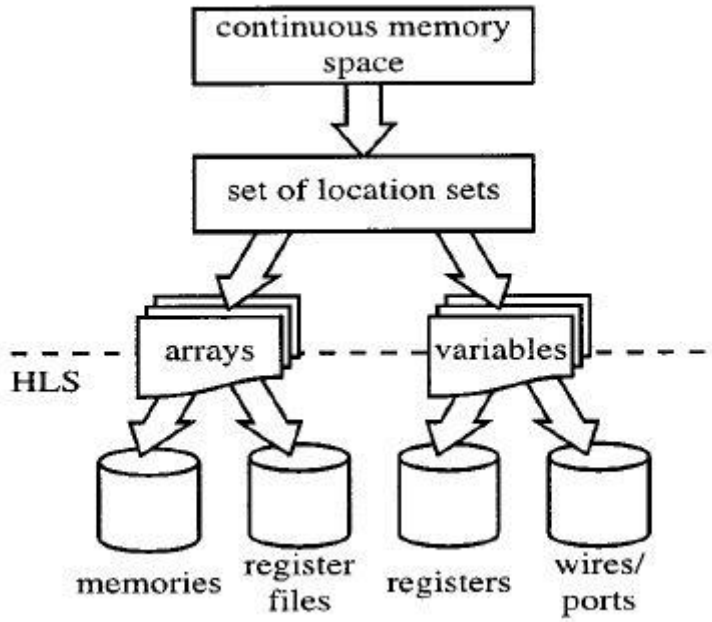


Εικόνα 1: Η αρχιτεκτονική του Μικροεπεξεργαστή Intel 8086 [41]

2.2 Απεικόνιση εφαρμογών λογισμικού σε υλικό

Ενδεχομένως σε κάποιες εφαρμογές να είναι χρήσιμη η απεικόνιση λειτουργιών του λογισμικού σε υλικό. Δημοσιεύσεις παρουσιάζουν την απεικόνιση αυτή με σκοπό διάφορα πλεονεκτήματα.

Στην εργασία τους οι L. Séméria, K. Sato και G. De Micheli (2001) [1] παρουσιάζουν τη διαχείριση μνήμης μέσω υλικού. Υποστηρίζει τις συναρτήσεις malloc και free, καθώς και δείκτες. Για την απεικόνιση μνήμης, χρησιμοποιούνται τα location sets των R. Wilson και M. Lam (1995).



Εικόνα 2: Μετατροπή συνεχόμενου τμήματος μνήμης σε σύνολο μνημών, καταχωρητών και καλωδίων [1]

Η απλούστερη απεικόνιση της μνήμης, με μια διεύθυνση δεν εξυπηρετεί, διότι δεν βελτιστοποιεί την τοπικότητα και τον παραλληλισμό του κώδικα. Για τους δείκτες, επιλέγεται ξανά η υλοποίηση των R. Wilson και M. Lam (1995), διότι παρόλη την πολυπλοκότητά της, παρέχει ακρίβεια. Για την απεικόνιση σε υλικό, κάθε location set αναπαρίσταται με ένα καλώδιο, έναν καταχωρητή, ή ένα κομμάτι μνήμης. Όσον αφορά τους δείκτες, το εργαλείο παράγει το κατάλληλο κύκλωμα για δυναμική πρόσβαση σε καταχωρητές, καλώδια ή πύλες, ανάλογα με την τιμή του δείκτη. Επιπλέον, για τη συνάρτηση δυναμικής δέσμευσης μνήμης (malloc), απαιτείται ο σχεδιαστής να επιλέξει ποιο κομμάτι της μνήμης θα διατεθεί, καθώς και το μέγεθος κάθε κομματιού. Τότε το εργαλείο παράγει για κάθε κομμάτι μνήμης τον αντίστοιχο κατανεμητή και συνθέτει το κατάλληλο κύκλωμα για δέσμευση, αποδέσμευση και πρόσβαση δεδομένων. Τέλος, για να κληθεί η συνάρτηση δυναμικής αποδέσμευσης μνήμης (free), παράγονται εντολές διακλάδωσης, οι οποίες μαζί με την ετικέτα του δείκτη καλούν τους κατάλληλους κατανεμητές. Η τιμή του δείκτη είναι και αυτή που αποστέλλεται τελικά στον κατάλληλο κατανεμητή για να αποδεσμευθεί το κατάλληλο κομμάτι μνήμης.

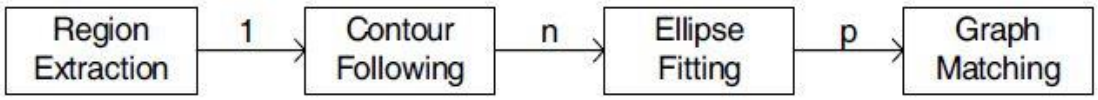
Οι A. Ghosh, J. Kunkel και S. Liao (1999) [2] επιχειρούν να υλοποιήσουν μοντέλα υλικού για εφαρμογές λογισμικού. Υποστηρίζουν ότι ένα πλεονέκτημα του υλικού έναντι του λογισμικού είναι η παραλληλοποίηση. Αυτό διορθώνεται με τη δημιουργία διεργασιών, οι οποίες είναι προγράμματα που εκτελούνται παράλληλα. Τα σήματα μοντελοποιούνται με κλάσεις. Η αντιδραστικότητα με διάφορες

προσεγγίσεις που έχουν να κάνουν με γεγονότα και διεργασίες περιγράφεται πιο αναλυτικά στη δημοσίευση των S. Liao, S. Tjiang and R. Gupta το 1997[15]. Δομές δεδομένων υλικού αναπαριστούνται με κλάσεις. Η σχεδίαση όμως, εξακολουθεί να μην είναι ολοκληρωμένη, διότι απαιτούνται βελτιώσεις για να είναι έτοιμη για σύνθεση η σχεδίαση. Οι βελτιώσεις αυτές αφορούν τα δεδομένα, τον έλεγχο, και την επιβεβαίωση ότι πράγματι τηρήθηκαν οι απαραίτητοι κανόνες. Η διαδικασία αυτή πέτυχε βελτιστοποιήσεις στο χώρο και το χρόνο.

2.3 Απεικόνιση εφαρμογών σε υλικό

Πολλές φορές για να ελαχιστοποιηθεί ο χρόνος εκτέλεσης, ο συνολικός κώδικας απεικονίζεται σε υλικό. Η μέθοδος αυτή μπορεί να χρησιμοποιηθεί σε αρκετές εφαρμογές. Αυτό έχει ως αποτέλεσμα υπάρχουν αρκετές δημοσιεύσεις, στις οποίες περιγράφεται το πώς χρησιμοποιείται η μέθοδος αυτή για να βελτιστοποιήσει διαφορετικά υπολογιστικά προβλήματα.

Οι F. Haim, M. Sen, D.-I. Ko, S. S. Bhattacharyya και W. Wolf (2006) [3] εφαρμόζουν τεχνική μοντελοποίησης ονόματι HPDF (homogeneous parameterized dataflow) που είχαν παρουσιάσει σε προηγούμενη εργασία (M. Sen, S. S. Bhattacharyya, T. Lv, W. Wolf, 2005) σε πλαίσιο CSDF (cyclostatic dataflow) (G. Bilsen, M. Engels, R. Lauwereins, J. A. Peperstraete, 1996). Αυτό το ενοποιημένο μοντέλο χρησιμοποιείται για την απεικόνιση σε υλικό εφαρμογής αναγνώρισης χειραψίας.



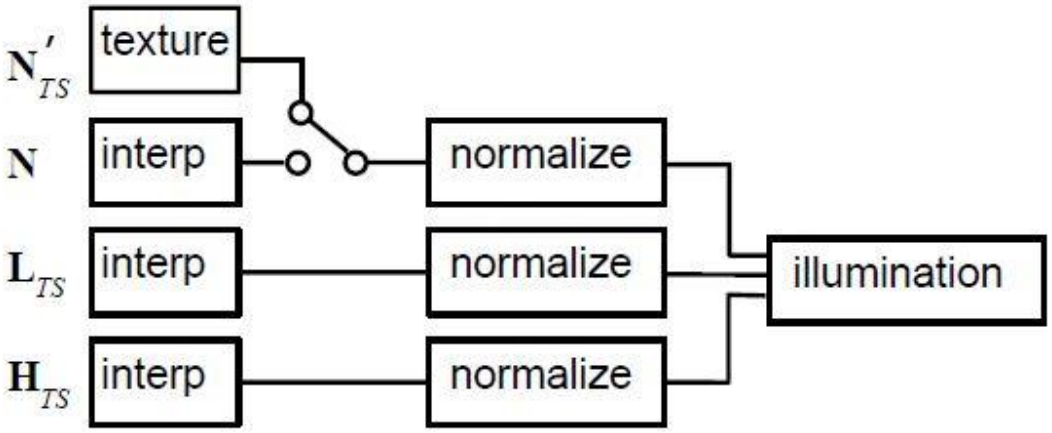
Εικόνα 3: Block διάγραμμα αλγορίθμου αναγνώρισης χειραψίας [3]

Το μοντέλο HPDF μπορεί να εφαρμοστεί σε διάφορα υπολογιστικά μοντέλα ροής δεδομένων με σκοπό να βελτιώσει σημαντικά την εκφραστική τους δύναμη. Η ενσωμάτωση του CSDF στο HPDF παρέχει στο εργαλείο τη δυνατότητα παραλληλισμού των εικονοστοιχείων (pixels), με σκοπό τη βελτιστοποίηση του μεγέθους του προσωρινού πίνακα και της κατανάλωσης ενέργειας. Οι βελτιστοποιήσεις αυτές επιβεβαιώθηκαν πειραματικά. Μένοντας στην επεξεργασία εικόνων, οι Y. Hemaraj, M. Sen, R. Shekhar, S. S. Bhattacharyya (2006)[8] παρουσιάζουν μεθόδους για απεικόνιση εφαρμογών άκαμπτων εγγραφών εικόνας σε υλικό. Το μοντέλο που χρησιμοποιείται είναι το HPDF (M. Sen, S. S. Bhattacharyya, T. Lv, W. Wolf, 2005) σε πλαίσιο CSDF (G. Bilsen, M. Engels, R. Lauwereins, J. A. Peperstraete, 1996) για τη μοντελοποίηση χαμηλού επιπέδου, πολλαπλών φάσεων αλληλεπιδράσεων μεταξύ κορυφών γράφων ροών δεδομένων. Δημιουργούνται πολλαπλά αντίγραφα των κορυφών αυτών με στόχο την παράλληλη υλοποίηση. Όλες αυτές οι κορυφές έχουν τον ίδιο ρυθμό παραγωγής και κατανάλωσης, και έτσι πυροδοτούνται με τον ίδιο ρυθμό. Η αρχιτεκτονική που αναπτύχθηκε βασίστηκε σε αυτόν τον παραλληλισμό και κατά τη διάρκεια της απεικόνισης εκμεταλλεύεται το

χώρο του υλικού. Αυτό έχει ως αποτέλεσμα, ανάλογα με τις ανάγκες της εφαρμογής να ανταλλάσσονται η απόδοση και η χρήση των πόρων του υλικού.

Οι C. Trendall και J. Stewart (2000) [13] εστιάζουν στο υλικό γραφικών, και πώς μπορεί να χρησιμοποιηθεί και σε άλλες εφαρμογές. Η συνέλιξη μπορεί να υπολογιστεί απευθείας. Οι παράγωγοι μπορούν να υπολογισθούν με τη χρήση συνέλιξης. Οι νόρμες για πεδία ύψους μπορούν να υπολογισθούν χρησιμοποιώντας υφές εικονοστοιχείων (pixels) για να κανονικοποιηθούν οι παράγωγοι των πεδίων ύψους. Τα εσωτερικά γινόμενα ενός διανύσματος με έως και τέσσερα άλλα μπορούν να υπολογισθούν με πίνακα χρωμάτων. Άλλες πράξεις που μπορούν να απεικονισθούν είναι η πρόσθεση, η αφαίρεση και ο πολλαπλασιασμός σταθερών ή διανυσμάτων. Πειραματικά αποδεικνύεται ότι παρόλο που το λογισμικό είναι πιο ακριβές από το υλικό σε πράξεις κινητής υποδιαστολής, μπορεί να επιταχύνει αρκετά ένα πιο γενικό σύνολο υπολογισμών που χρησιμοποιείται συχνά στα γραφικά.

Οι M. Peercy, J. Airey και B. Cabral (1997) [9] υλοποίησαν μέθοδο απεικόνισης σε υλικό εξογκωμάτων που χρησιμοποιούν σκίαση Phong.



Εικόνα 4: Μια πιθανή υλοποίηση αλγορίθμου αναγνώρισης εξογκωμάτων [9]

Ο αλγόριθμος, αρχικά προϋπολογίζει μια υφή της διατάραξης του φυσιολογικού σε χώρο εφαπτομένων και μετατρέπει όλα τα διανύσματα σκίασης σε χώρο εφαπτομένων ανά κορυφή. Κατόπιν, υπολογίζει με παρεμβολή και επανακανονικοποιοεί τα διανύσματα σκίασης και κανονικοποεί το διαταραγμένο φυσιολογικό από την υφή. Τέλος, υπολογίζει το μοντέλο φωτισμού με βάση αυτά τα διανύσματα. Η μέθοδος αυτή ελαχιστοποιεί το κόστος ανά εικονοστοιχείο (pixel), με σκοπό την απεικόνιση εξογκωμάτων στο ελάχιστο δυνατό υλικό.

Ανάλογα αποτελέσματα παρατηρούνται και σε άλλου είδους εφαρμογές.

Στην Ελλάδα, οι M. D. Galanis, G. Dimitroulakos και C.E. Goutis (2005) [4] εστιάζουν στην επιτάχυνση της κρίσιμης όδευσης με το να εκτελεστεί σε χονδρόκκοκο υλικό. Διαχωρίζει τον κώδικα σε κομμάτια υλικού διαφορετικού βαθμού ανάλυσης. Λογισμικό που τρέχει σε ενσωματωμένο μικροεπεξεργαστή επιλέγει το κομμάτι του

κώδικα C της εισόδου που παράγει την κρίσιμη όδευση για να απεικονιστεί στο υλικό. Η μέθοδος αυτή πρόσφερε κατά μέσο όρο μια βελτίωση κατά 2.3 σε 5 εφαρμογές πραγματικού χρόνου.

Η δημοσίευση των M. Huang, V. K. Narayana και T. El-Ghazawi (2009) [5] επιχειρεί να βελτιώσει την απόδοση χρησιμοποιώντας μια βιβλιοθήκη υλοποίησης διεργασιών. Η βιβλιοθήκη αυτή περιέχει αρχιτεκτονικές παραλλαγές για κάθε διεργασία υλικού, εκφράζοντας έτσι την ανταλλαγή μεταξύ της καλύτερης αξιοποίησης των πόρων και της διεκπεραιωτικής ικανότητας της εκτέλεσης της διεργασίας.

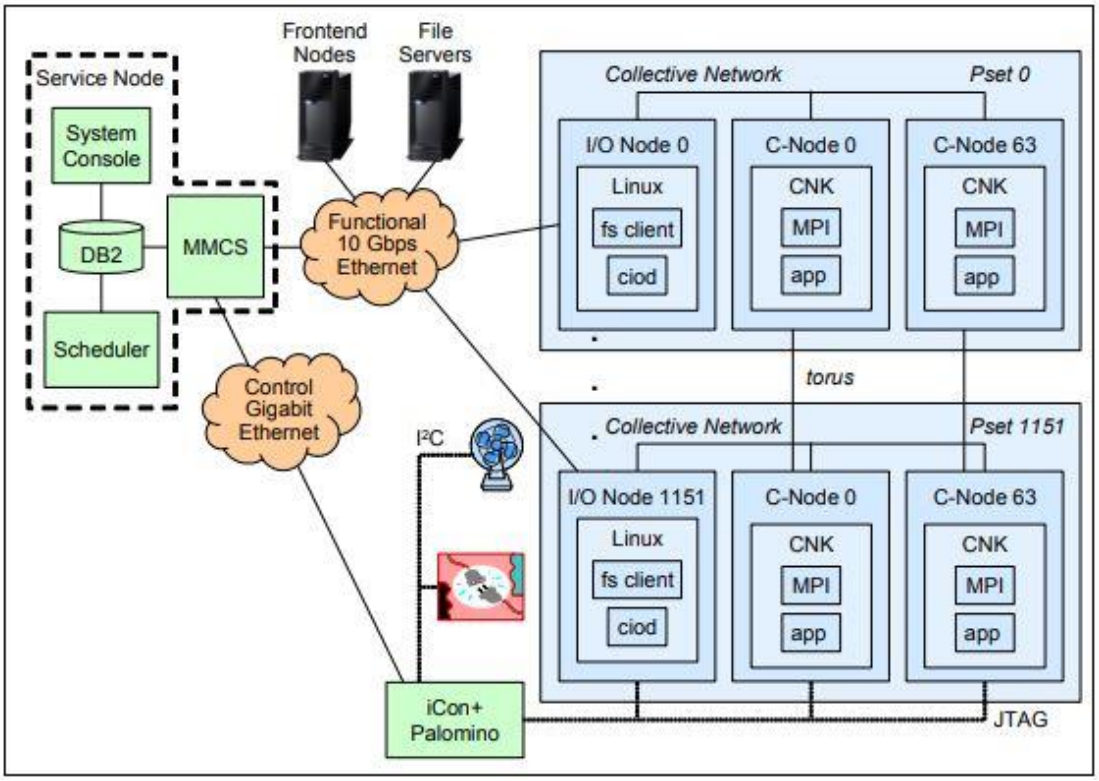
Input: Random initial population of Q chromosomes
Output: New population after genetic evolution
1.1 **repeat**
1.2 Evaluate fitness of each chromosome in population;
1.3 **repeat**
1.4 Select two chromosomes from the current population;
1.5 Crossover based on crossover rate, generate two offsprings;
1.6 Step through all the bits in the offsprings, flip them based on mutation rate;
1.7 **until** *a new generation has been created* ;
1.8 **until** K *generations have been evaluated* ;

Εικόνα 5: Αλγόριθμος γενετικής [5]

Ο αλγόριθμος που υλοποιείται για την επίλυση αυτού του προβλήματος θυμίζει διεργασίες της γενετικής. Κάθε χρωμόσωμα αποτελείται από γονίδια. Τα χρωμοσώματα εκφράζουν τις διεργασίες, και κάθε χρωμόσωμα εκφράζει μια παραλλαγή υλοποίησης της διεργασίας. Ο αλγόριθμος ελέγχει την καλή κατάσταση κάθε χρωμοσώματος, επιλέγει δύο από αυτά, παράγει δύο απογόνους, και τους μετατρέπει σύμφωνα με το δείκτη εξέλιξης, μέχρι να δημιουργηθεί μια νέα γενιά χρωμοσωμάτων. Εκτελείται μέχρι να ελεγχθεί ένας συγκεκριμένος αριθμός γενεών. Πειραματικά ελέγχθηκε η απόδοση γράφου διεργασιών 18 κόμβων και βασίστηκε στους αρχιτεκτονικούς περιορισμούς τεσσάρων αναδιατασσόμενων συστημάτων. Οι βελτιώσεις στο χρόνο εκτέλεσης σε σχέση με τη χρήση μιας συγκεκριμένης παραλλαγής υλοποίησης για κάθε διεργασία έφτασαν στο 85,3%.

Οι D. C. Suresh, W. A. Najjar F. Vahid, J. R. Villarreal και G. Stitt παρατήρησαν και παρουσίασαν το 2003 [11] ότι τα πιο συχνά εκτελεσμένα κομμάτια κώδικα αποτελούνται από βρόχους επανάληψης. Συνεπώς, είναι οι καταλληλότεροι υποψήφιοι για καταμερισμό σε λογισμικό και υλικό. Αρχικά ορίζει ως πυρήνα ένα κομμάτι κώδικα το οποίο αποτελείται από όλους τους βρόχους επανάληψης των οποίων ο χρόνος εκτέλεσης είναι μεγαλύτερος από ένα ορισμένο άνω όριο. Για την βελτιστοποίηση του πυρήνα χρησιμοποιήθηκε ο αναλυτής gcc, ο οποίος μείωσε τον αριθμό δυναμικών εντολών. Για να μειωθεί ο χρόνος εκτέλεσης επετεύχθη η απεικόνιση των βελτιστοποιημένων πυρήνων σε υλικό.

Θετικά αποτελέσματα παρατηρούμε και σε υπερυπολογιστές. Το 2011 [10], οι P. Balaji, R.Gupta, A. Vishnu και P. Beckman πραγματεύτηκαν παράλληλες εφαρμογές που εκτελούνται σε συστήματα υψηλού επιπέδου, όπως ο υπερυπολογιστής Blue Gene της IBM.



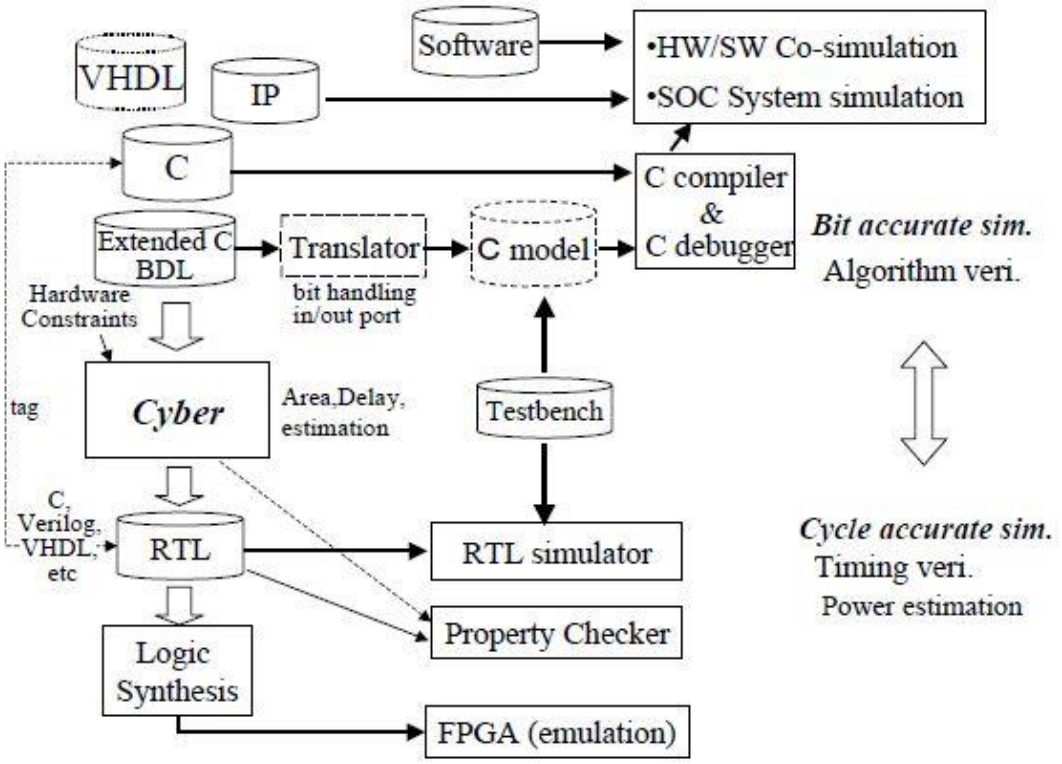
Εικόνα 6: Περιβάλλον Blue Gene/P [42]

Η επικοινωνία μεταξύ διεργασιών συχνά απαιτεί διαμοιρασμό του δικτύου με αποτέλεσμα η απόδοση να εξαρτάται σε μεγάλο βαθμό από το πώς είναι απεικονισμένες στο δίκτυο οι διεργασίες. Όσο αυξάνονται οι διαστάσεις της τοπολογίας, απομακρύνονται οι ομάδες διεργασιών που επικοινωνούν μεταξύ τους με αποτέλεσμα την επικάλυψη επικοινωνίας μεταξύ διαφορετικών ομάδων διεργασιών. Τα σημαντικά για την αποδοτική απεικόνιση των διεργασιών είναι η κατανόηση του μοτίβου επικοινωνίας κάθε εφαρμογής, κατανόηση της τοπολογίας του δικτύου της πλατφόρμας και η απεικόνιση αυτών των πληροφοριών για να υπολογιστούν οι συγκρούσεις στο δίκτυο για συγκεκριμένο μοτίβο εφαρμογών σε συγκεκριμένη πλατφόρμα. Αυτό επιτυγχάνεται με την τοποθέτηση διεργασιών οι επικοινωνούν μεταξύ τους γειτονικά σε καρτεσιανό σύστημα συντεταγμένων. Πειραματικά αποδείχθηκε ότι η διαφορετική απεικόνιση διεργασιών παρουσιάζει σημαντική διαφορά, ειδικά σε μεγάλα συστήματα.

Το 1996 [12] παρουσιάστηκε από τους X. Fang, P. Thole, J. Göppert και W. Rosenstiel ένα σύστημα υλικού το οποίο επιτρέπει τον παράλληλο υπολογισμό της Ευκλείδειας απόστασης χρησιμοποιώντας αυτο-οργανώμενους χάρτες του Kohonen για μια ειδική διαδικτυακή εφαρμογή. Χρησιμοποιούνται δύο επίπεδα παραλληλισμού, μεταξύ των νευρώνων, και των συνάψεων. Επιλέγεται ο παραλληλισμός συνάψεων, διότι ο παραλληλισμός νευρώνων είναι πολυέξοδος όσον αφορά την επικοινωνία για

να βρει την ελάχιστη απόσταση σε όλο το χάρτη. Για να επιτευχθεί ο παραλληλισμός μεταξύ των συνάψεων, χρησιμοποιείται ένα στοιχείο επεξεργαστή για κάθε τμήμα του νευρώνα. Αυτό έχει ως αποτέλεσμα να εκτελείται ο υπολογισμός της απόστασης παράλληλα ανά κομμάτι του χάρτη, και έτσι η μικρότερη απόσταση ανανεώνεται τακτικά. Ο υπολογισμός της απόστασης επιτυγχάνεται με τέσσερα επίπεδα διασωλήνωσης, τα οποία είναι τα παρακάτω: φόρτωση από τη μνήμη της τρέχουσας απόστασης, τετραγωνισμός της απόστασης, συσσώρευση των αποτελεσμάτων σε ένα δέντρο αθροιστών και σύγκριση του αθροίσματος με την τρέχουσα μικρότερη τιμή. Ο αλγόριθμος είναι γραμμένος σε VHDL και η απεικόνιση γίνεται αυτόματα από εργαλεία της XILINX. Οι τεχνικές αυτές αποσκοπούν στην ραγδαία εύρεση της ελάχιστης απόστασης και του νευρώνα στον οποίο αντιστοιχεί.

Σε άλλες εφαρμογές, το 1999 [7] ο K. Wakabayashi εισάγει την έννοια του «Cyber», ενός περιβάλλοντος σύνθεσης.



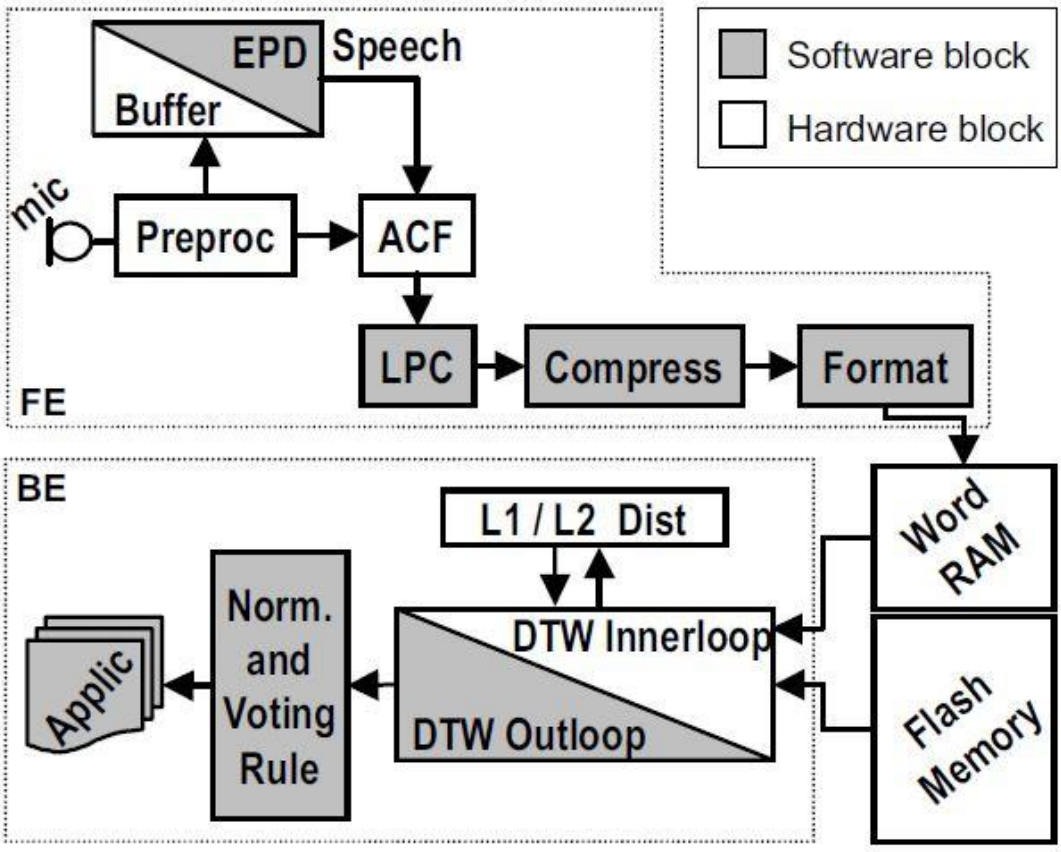
Εικόνα 7: Ενσωμάτωση του Cyber σε περιβάλλον σχεδίασης υλικού βασισμένο σε C [7]

Το «Cyber» δέχεται C (BDL - Behavior Description Language) και VHDL (VHSIC-HDL, Very High Speed Integrated Circuit Hardware Description Language) και παράγει RT (register-transfer) επιπέδου περιγραφές σε C (BDL), VDHL, Verilog, κλπ. Ο κώδικας C (BDL) που παράγει είναι συμβατός με συνηθισμένο αναλυτή, και είναι αρκετά γρηγορότερος από RT-VHDL και RT_Verilog.

2.4 Παράλληλη χρήση υλικού και λογισμικού

Υπάρχουν όμως και περιπτώσεις όπου για να ελαχιστοποιηθεί ο χρόνος εκτέλεσης, ο συνολικός κώδικας διαχωρίζεται σε υλικό και λογισμικό.

Στην Ιταλία, οι M. Besana και M. Borgatti (2003) [6] υποστήριξαν ότι τα λειτουργικά συστήματα πραγματικού χρόνου υψηλού επιπέδου παρέχουν συνέπεια, ακρίβεια και αποδοτικότητα σε προσεγγίσεις από πάνω προς τα κάτω. Αυτά τα συστήματα είναι αρκετά αποδοτικά σε συστήματα που συνδυάζουν λογισμικό και υλικό.



Εικόνα 8: Σχεδιαστική ροή δεδομένων [6]

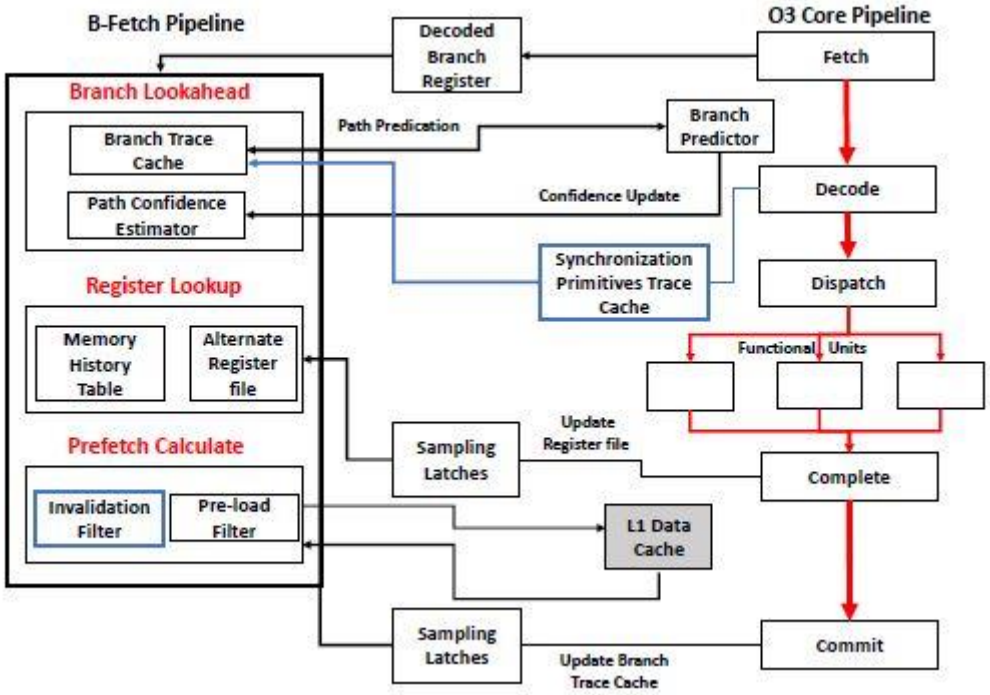
Η εργασία αυτή εφαρμόζει το μοντέλο αυτό σε ένα σύστημα αναγνώρισης ομιλίας. Αρχικά, ο όγκος του συστήματος διαχωρίζεται σε 21 διεργασίες. Κατόπιν, ο επεξεργαστής ARM7TDMI που χρησιμοποιείται καθορίζει το πρόγραμμα εκτέλεσης των διεργασιών και τις καθυστερήσεις που σχετίζονται με την εναλλαγή των διεργασιών. Το λειτουργικό σύστημα μετατρέπει μια ακολουθία διεργασιών σε μία μοναδική διεργασία, μειώνοντας τον χρόνο εκτέλεσης του λειτουργικού. Κατόπιν, οι διεργασίες απεικονίζονται στην αντίστοιχη πλατφόρμα. Πειραματικά αποδείχθηκε ότι η υλοποίηση αυτή προσφέρει συνέπεια στη σχεδιαστική ροή, 0,66% ακρίβεια και βελτίωση 11,8% στην ταχύτητα.

Στα πλαίσια της δημοσίευσης των M. B. Srivastava, J. S. Sun και R. W. Brodersen το 1992 [14] αναπτύχθηκε το σύστημα SIERA, το οποίο παράγει ραγδαία δομικά

στοιχεία υλικού και λογισμικού συστημάτων πραγματικού χρόνου. Το πρόβλημα αυτό είναι γενικά πολύ δύσκολο διότι δεν υπάρχει συγκεκριμένη μεθοδολογία που να είναι χρήσιμη σε όλες τις εφαρμογές και καμία σύνθεση αρχιτεκτονικής ή απεικόνιση δεν δουλεύει παντού. Σε επίπεδο τσιπ όπως, βοήθησε πολύ ένα σετ συνθέσεων ειδικής συμπεριφοράς ή εργαλείων απεικόνισης για τη δημιουργία μονάδων γενικής χρήσης. Παράγονται δύο τύποι μονάδων υλικού. Αρχικά, η Μονάδα Επεξεργασίας Δεδομένων, η οποία υλοποιεί διάφορους αλγόριθμους λαμβάνοντας υπόψιν τους περιορισμούς υπολογιστικότητας και Εισόδου/Εξόδου με στόχο τη λειτουργία σε πραγματικό χρόνο. Η Μονάδα Εσωτερικής Σύνδεσης από την άλλη, συνδέει Μονάδες Επεξεργασίας Δεδομένων, δεδομένα, Εισόδους/Εξόδους, και συγχρονίζει τη μεταφορά δεδομένων σύμφωνα με τους χρονικούς περιορισμούς που καθορίζονται από τις Εισόδους/Εξόδους και τις Μονάδες Επεξεργασίας Δεδομένων. Για πιο σύνθετες εφαρμογές δημιουργούνται και ορισμένα μοντέλα λογισμικού για να αλληλεπιδράσουν με αυτά του υλικού. Το γεγονός αυτό όμως, δημιουργεί προκλήσεις στην επικοινωνία μεταξύ υλικού και λογισμικού. Για να αντιμετωπιστεί αυτό το ζήτημα, δημιουργήθηκαν ξεχωριστές βιβλιοθήκες μοντέλων λογισμικού για επικοινωνία με τα μοντέλα του επεξεργαστή. Πειραματικά αποδείχθηκε ότι συστήματα υλικού παρέχουν τεράστια πλεονεκτήματα απόδοσης.

2.5 Εφαρμογές παράλληλης υλοποίησης

Το 2018 [16], οι L. M. AlBarakat, P. V. Gratz και D. A. Jimenez, ισχυρίστηκαν ότι για να εκμεταλλευτούμε πλήρως την κλιμακωμένη απόδοση σε τσιπ πολυεπεξεργαστών, είναι απαραίτητο να διαιρεθούν οι εφαρμογές σε ημιαυτόνομες διεργασίες. Οι διεργασίες αυτές πρέπει να μπορούν να τρέχουν ταυτόχρονα σε διαφορετικούς πυρήνες ενός συστήματος. Για να επιτευχθεί αυτό, θα πρέπει ο προγραμματιστής να εισάγει αρχές συγχρονισμού των νημάτων, με σκοπό τον συγχρονισμό προσπέλασης δεδομένων μεταξύ των διεργασιών. Οι αρχές αυτές όμως, μπορεί να καθυστερούν στην αναμονή για κλειδώματα ή φράγματα, με αποτέλεσμα να υπάρχει αστάθεια στα νήματα και χαμηλή κλιμακωμένη απόδοση. Για να ξεπεραστεί το πρόβλημα αυτό, υλοποιήθηκε σε υλικό τεχνική για ασφαλή προφόρτωση (fetch) δεδομένων σε τσιπ πολυεπεξεργαστών. Η προφόρτωση (fetch) δεδομένων είναι μια αρκετά δημοφιλής τεχνική που βασίζεται στο γέμισμα της κρυφής μνήμης με χρήσιμα δεδομένα πριν έρθει από τον επεξεργαστή το αίτημα φόρτωσής τους. Συνήθως οι μέθοδοι προφόρτωσης (fetch) περιμένουν να προκύψει κάποια αποτυχία στην κρυφή μνήμη και τότε διαβάζουν μια ή περισσότερες ομάδες γραμμών κώδικα σχετικές με τη συγκεκριμένη αποτυχία. Άλλες μέθοδοι, όπως η B-Fetch ενεργοποιούνται με τη φόρτωση εντολών από τον επεξεργαστή. Το σύστημα που περιγράφει η δημοσίευση αυτή, το MTB-Fetch (Multi-Thread B-Fetch) εμπεριέχει την B-Fetch και μερικά ακόμα μέρη.

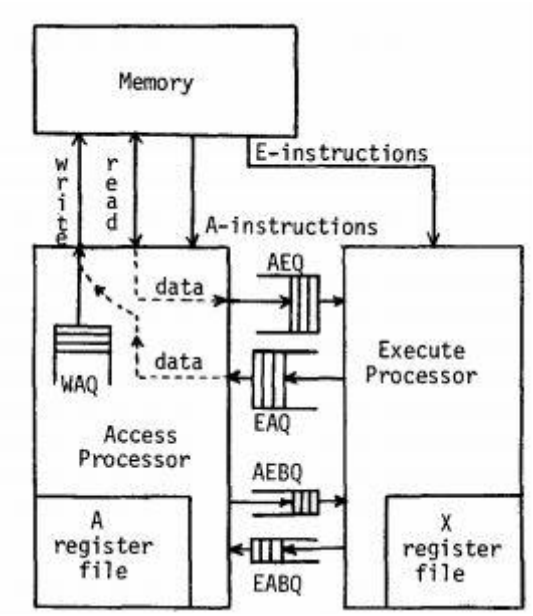


Εικόνα 9: Μικροαρχιτεκτονική MTB-Fetch/B-Fetch [16]

Η B-Fetch αποτελείται από τρία βασικά μέρη, την Πρόβλεψη Βρόχου, που παράγει τη διαδρομή εκτέλεσης, την Αναζήτηση Καταχωρητή που παρέχει πληροφορίες για την παραγωγή διευθύνσεων και τον Υπολογισμό Προφόρτωσης (fetch), που παράγει τη διεύθυνση προφόρτωσης (fetch). Η σχεδίαση αυτή όμως, έχει προβλήματα, τα οποία επιλύονται με δύο μέρη που προσθέτονται στην αρχική σχεδίαση, υλοποιώντας το MTB-Fetch. Η Κρυφή Μνήμη Εύρεσης Πρωτόγονων Συγχρονισμών, η οποία προφορτώνει δεδομένα από το επόμενο κομμάτι, και το Φίλτρο Ακύρωσης, που εντοπίζει τα δεδομένα που προφορτώθηκαν από κάποιο πυρήνα για να μην προφορτωθούν από άλλους πυρήνες πρόωρα. Η υλοποίηση αυτή, πετυχαίνει βελτίωση της τάξης του 9,3% η οποία είναι διπλάσια του ανταγωνιστή ελαφρύ προφορτωτή για τον δεδομένο όγκο δουλειάς.

2.6 Αρχιτεκτονικές Διαχωρισμένης Πρόσβασης-Εκτέλεσης (DAE)

Η Αρχιτεκτονική Διαχωρισμένης Πρόσβασης-Εκτέλεσης (DAE) παρουσιάστηκε για πρώτη φορά το 1985 από τον J. E. Smith [40]. Σκοπός της ήταν να ελαφρύνει τον φόρτο του προγραμματιστή στο κομμάτι του συντονισμού και συγχρονισμού ροών εντολών. Αυτό επιτυγχάνεται με το διαχωρισμό της πρόσβασης από την εκτέλεση και την επικοινωνία τους με ουρές.

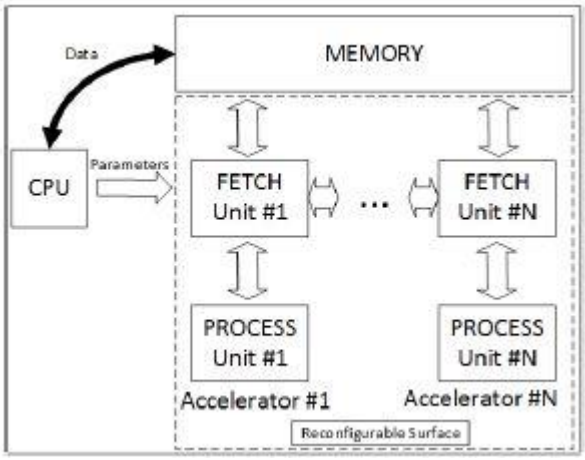


Εικόνα 10: Η αρχιτεκτονική DAE [40]

Πιο συγκεκριμένα, η αρχιτεκτονική διαχωρίζεται σε δύο λειτουργικές μονάδες. Κάθε μονάδα έχει δική της ροή εντολών, (τους επεξεργαστές Α και Ε) με τη δική της σειρά καταχωρητών η κάθε μια. Οι επεξεργαστές έχουν παρόμοια δομή, αλλά εκτελούν διαφορετικές πράξεις. Ο επεξεργαστής Α (Access - Πρόσβαση) εκτελεί όλες τις απαραίτητες διεργασίες για την μεταφορά δεδομένων από και προς την κύρια μνήμη, δηλαδή τον υπολογισμό των διευθύνσεων και την υλοποίηση των αιτήσεων από και προς τη μνήμη. Τα δεδομένα που παράγονται είτε επιστρέφονται εντός της μονάδας, είτε αποθηκεύονται στην ουρά AEQ (Access to Execute Queue) για αποστολή στη δεύτερη μονάδα, τον επεξεργαστή Ε (Execute - Εκτέλεση). Η μονάδα αυτή λαμβάνει δεδομένα από την ουρά AEQ, εκτελεί τις απαραίτητες πράξεις και επιστρέφει τα αποτελέσματα στην ουρά EAQ (Execute to Access Queue). Η μονάδα πρόσβασης περιέχει και την ουρά WAQ, στην οποία γράφονται διευθύνσεις για την αποθήκευση των αποτελεσμάτων αμέσως μόλις υπολογιστούν, χωρίς να χρειάζεται να ληφθούν τα δεδομένα προς αποθήκευση από την ουρά EAQ. Οι διεργασίες αυτές εκτελούνται παράλληλα, με αποτέλεσμα μόλις ληφθούν τα δεδομένα από την ουρά EAQ να συνδυάζονται η πρώτη διεύθυνση με την αντίστοιχη στην ουρά WAQ (Write Address Queue) και να αποστέλλονται απευθείας στη μνήμη. Από την εισαγωγή της αρχιτεκτονικής αυτής μέχρι και σήμερα μεγάλος αριθμός ερευνητών χρησιμοποίησε παραλλαγές της με σκοπό την εκμετάλλευση των πλεονεκτημάτων της στις εφαρμογές τους.

Σύμφωνα με τους G. Charitopoulos, C. Vatsolakis, G. Chrysos, D. N. Pnevmatikatos (2018) [17], η επιτάχυνση εφαρμογών χρησιμοποιώντας αναδιατασσόμενη λογική και επιταχυντές, είναι μια ανερχόμενη τεχνική που υπόσχεται να παρέχει υπολογιστικές επιταχύνσεις με βελτίωση στην ενεργειακή αποδοτικότητα. Στη δημοσίευση αυτή περιγράφεται Αρχιτεκτονική Διαχωρισμένης Πρόσβασης-Εκτέλεσης και ένα πλαίσιο για Αναδιατασσόμενους επιταχυντές (DAER - Decoupled Access-Execute architecture and framework for Reconfigurable accelerators). Η

εργασία της εφαρμογής χωρίζεται σε δύο μέρη, την επεξεργασία δεδομένων που εκτελεί όλους τους υπολογισμούς και την φόρτωση δεδομένων, η οποία χρησιμοποιείται για τις συναλλαγές δεδομένων εισόδου και εξόδου της μνήμης.

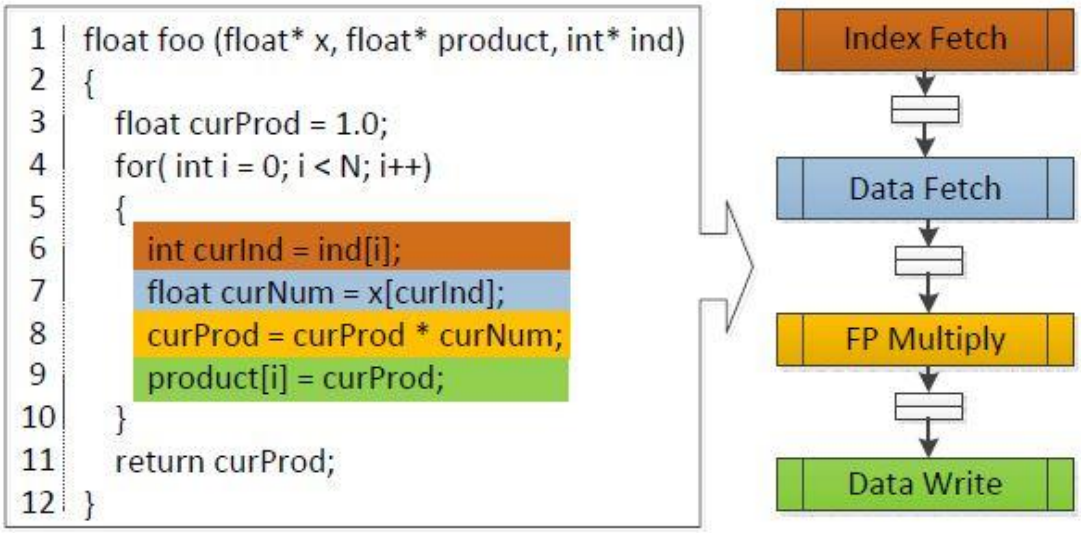


Εικόνα 11: Η αρχιτεκτονική DAER [17]

Το συγκεκριμένο πλαίσιο προσφέρει πλεονεκτήματα σε σχέση με άλλα όσον αφορά τον τύπο των αλγορίθμων που υποστηρίζει και της απόδοσης που πετυχαίνει. Το πρώτο κομμάτι, αυτό της φόρτωσης δεδομένων συνδέεται με τη μνήμη για να φέρνει δεδομένα εισόδου και να στέλνει πίσω στη μνήμη αποτελέσματα. Επίσης, συνδέεται και με τον επεξεργαστή για να μεταφέρει παραμέτρους σχετικές με τη μνήμη, όπως τις αρχικές θέσεις μνήμης, ή το μέγεθος πινάκων. Το δεύτερο κομμάτι, αυτό της επεξεργασίας, περιέχει αριθμητικές και λογικές πράξεις. Συνδέεται με το κομμάτι φόρτωσης δεδομένων με δομές FIFO. Επιτάχυνση των δύο αυτών κομματιών επιτυγχάνεται με τη χρήση οδηγιών Σύνθεσης Υψηλού Επιπέδου. Στοκ κώδικα τύπου Διαχωρισμένης Πρόσβασης-Εκτέλεσης προστέθηκαν οδηγίες προκειμένου να δημιουργηθούν γρήγορα κομμάτια κώδικα διοχέτευσης (pipeline) για τις μονάδες φόρτωσης και επεξεργασίας, καθώς και για τη δημιουργία διεπαφών τύπου FIFO χαμηλής καθυστέρησης μεταφοράς δεδομένων μεταξύ των μονάδων φόρτωσης και επεξεργασίας. Οι εφαρμογές που αξιολόγησαν την απόδοση του πλαισίου ήταν εφαρμογές-σημεία αναφοράς που χρησιμοποιούνται ευρέως και έχουν διαφορετικά χαρακτηριστικά. Χρησιμοποιήθηκαν γενικός πολλαπλασιασμός πινάκων, ο αλγόριθμος Needleman-Wunsch, πολλαπλασιασμός πινάκων αραιών διανυσμάτων, διδιάστατος μετασχηματισμός Laplace, και αριθμητικός μέσος για διανύσματα γράφων. Για να μελετηθεί η απόδοση των παραπάνω αλγορίθμων, αρχικά απεικονίσθηκαν σε αναδιατασσόμενα συστήματα με τη χρήση αρχιτεκτονικής Διαχωρισμένης Πρόσβασης-Εκτέλεσης και ένα πλαίσιο για Αναδιατασσόμενους επιταχυντές. Έγινε σύγκριση των αποτελεσμάτων τους με αυτά βελτιστοποιημένων και μη υλοποιήσεων υλικού. Η αρχιτεκτονική αυτή παρουσίασε καλύτερα αποτελέσματα σε σχέση με παλαιότερες σχετικές υλοποιήσεις τόσο σε υλικό, όσο και σε λογισμικό.

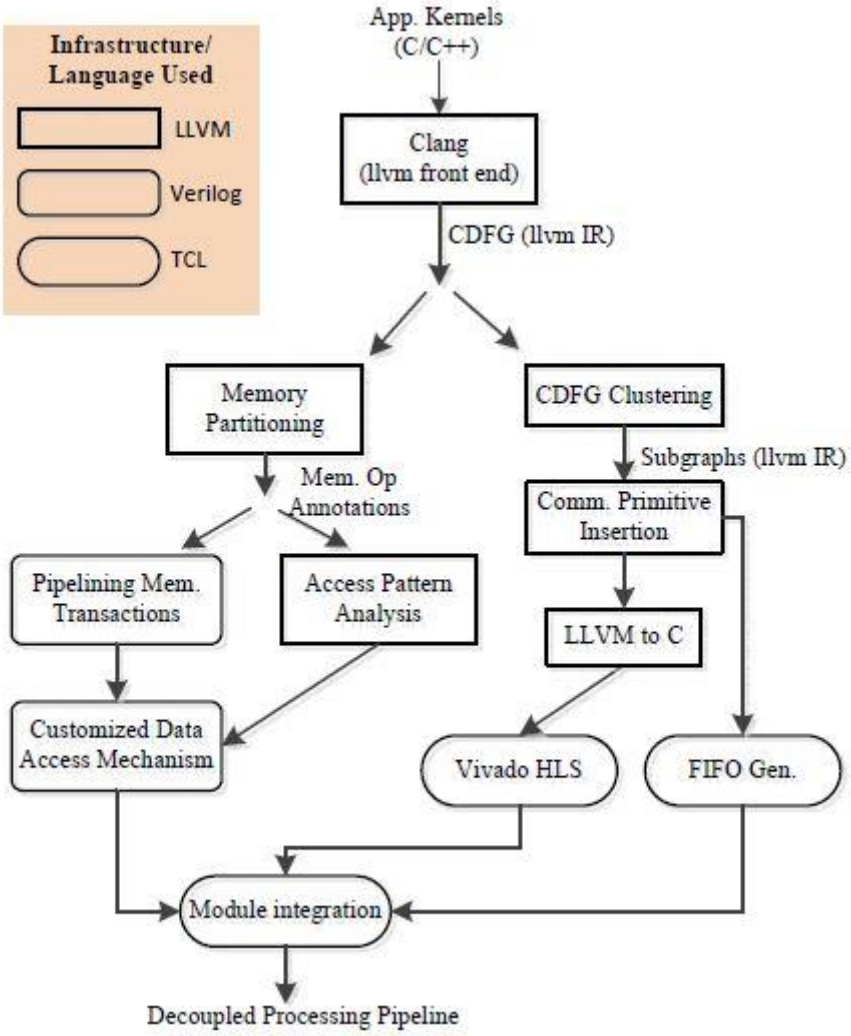
Οι S. Cheng and J. Wawrzynek το (2014) [18] παρουσίασαν μια αυτόματη ροή που επαναδομεί υλοποιήσεις υλικού με επίκεντρο τον επεξεργαστή με σκοπό να είναι πιο

αποδοτικές σε πλατφόρμες με FPGA. Η μεθοδολογία αυτή παράγει διοχετεύσεις (pipeline) που διαχωρίζουν τις διεργασίες της μνήμης και την πρόσβαση στα δεδομένα από τους υπολογισμούς.



Εικόνα 12: Μετατροπή σε διαχωρισμένα επίπεδα [18]

Τα αποτελέσματα έχουν πολύ καλύτερη διεκπεραιωτικότητα, διότι χρησιμοποιούν πιο αποδοτικά το εύρος της μνήμης και έχουν βελτιωμένη ανοχή στις καθυστερήσεις μεταφοράς στην πρόσβαση δεδομένων. Η εργασία αυτή στοχεύει στο να μικρύνει το κενό ανάμεσα στους μηχανισμούς υλικού και λογισμικού με το να μετατρέπει αυτόματα πυρήνες εφαρμογών, σε επίπεδα διοχέτευσης (pipeline). Αρχικά, ο γράφος της απόδοσης του κρίσιμου βρόχου, χωρίζεται σε κομμάτια που τοποθετώνται σε υπογράφους, οι οποίοι συνδέονται με άκυκλη επικοινωνία. Για να μεγιστοποιηθεί όμως η απόδοση, απαιτείται να ληφθούν υπ'όψιν ορισμένες προϋποθέσεις. Αρχικά, είναι σημαντικό να ελαχιστοποιηθούν οι μεταβάσεις από υπογράφο σε υπογράφο, καθώς η επικοινωνία μεταξύ των δομών προσθέτει καθυστέρηση. Επιπλέον, είναι ωφέλιμο να αποκοπούν οι διεργασίες της μνήμης από κύκλους απαιτήσεων με μεγάλη υπολογιστική καθυστέρηση, έτσι ώστε τυχόν αποτυχίες στην κρυφή μνήμη να επισκιάζονται από τον αργό ρυθμό χρήσης δεδομένων. Τέλος, ο αριθμός των διεργασιών της μνήμης ελαχιστοποιείται, με σκοπό την τοπικοποίηση καθυστερήσεων που προκαλούνται από αποτυχίες στην κρυφή μνήμη.

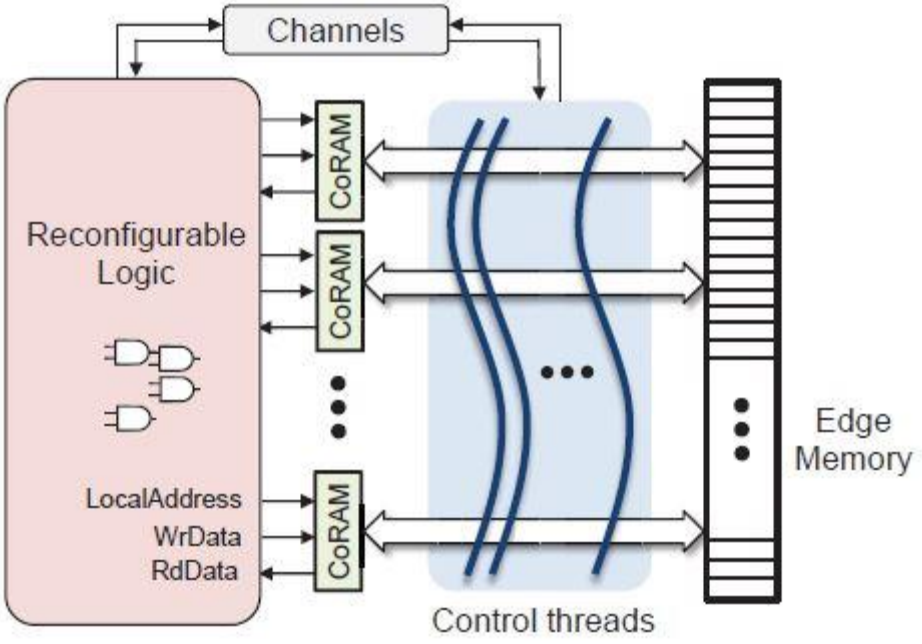


Εικόνα 13: Σχεδιαστική ροή διοχέτευσης (pipeline) [18]

Για τη δημιουργία των γράφων, υλοποιήθηκε αλγόριθμος, ο οποίος βρίσκει μέρη με δυνατές συνδέσεις στον αρχικό γράφο, τους συμπτύσσει σε κόμβους και ευρηματικά χωρίζει τους νέους γράφους σε νήματα με ισορροπημένο φορτίο. Το επόμενο βήμα, είναι η εκτέλεση ορισμένων μετασχηματισμών στις διεργασίες της μνήμης για να επιτραπεί η διοχέτευση (pipeline) αιτήσεων σε εκκρεμότητα στο υποσύστημα της μνήμης. Σε περίπτωση αποτυχίας φόρτωσης, όλη η δομή θα καθυστερήσει, με αποτέλεσμα να παύσει η εκκίνηση νέων συναλλαγών στη μνήμη. Αυτό είναι πρόβλημα, διότι είναι πιθανό η διεύθυνση για την επόμενη φόρτωση να μπορεί να υπολογισθεί. Για να επιλυθεί το ζήτημα αυτό, διασπάται η διεργασία πρόσβασης μνήμης σε δύο κομμάτια, την εντολή φόρτωσης στον αρχικό υπογράφο, και τον έλεγχο δεδομένων της FIFO με τα δεδομένα απόκρισης από τερματισμό ομάδας μετά από νέα πρόσβαση στη μνήμη. Μόλις υπάρχει αρκετός χώρος στη FIFO, εκκινείται μια νέα συναλλαγή στη μνήμη. Αυτό έχει ως αποτέλεσμα, κάθε κόμβος πρόσβασης μνήμης να μπορεί να διοχετεύσει (pipeline) πολλαπλές αιτήσεις όσο είναι έτοιμη η διεπαφή της μνήμης. Οι δομές υλικού που συνδέουν τον επιταχυντή και τη μνήμη, παράγονται με βάση ορισμένα μοτίβα στην πρόσβαση δεδομένων. Κάθε ανεξάρτητη διεπαφή πρόσβασης δεδομένων που ανταποκρίνεται σε έναν διαμελισμό μνήμης

υποστηρίζεται διαφορετικά, ανάλογα με τη φύση της ροής διευθύνσεων που παράγει. Για παράδειγμα, οι προσβάσεις τύπου streaming, όπου δεν επαναχρησιμοποιούνται δεδομένα, δεν κατανέμονται στον φορτωτή της FPGA. Στην περίπτωση αυτή αποστέλλονται αιτήσεις φόρτωσης και αποθήκευσης στην αρχική εκτέλεση του προγράμματος. Αντιθέτως, ο φορτωτής της FPGA ωφελεί στις περιπτώσεις που υπάρχει κύκλος απαιτήσεων από τη μνήμη. Για την περίπτωση αυτή, αλλά και για την περίπτωση τυχαίων προσβάσεων, έχει προστεθεί μια κρυφή μνήμη γενικής χρήσης, της οποίας το μέγεθος και η προσεταιριστικότητα μπορούν να προσαρμοστούν ανάλογα με τις απαιτήσεις της εφαρμογής. Η εφαρμογή αυτή παρουσίασε απόδοση βελτιωμένη κατά μέσο όρο 5.6 φορές σε σχέση με αυτή άλλων εργαλείων σύνθεσης υψηλού επιπέδου.

Οι E. S. Chung, J. C. Hoe, και K. Mai (2011) [19] ισχυρίζονται ότι οι FPGA προσφέρουν βελτιώσεις σε τάξεις μεγέθους σε σχέση με συμβατικούς μικροεπεξεργαστές τόσο σε απόδοση, όσο και σε ενεργειακή αποδοτικότητα. Παρόλα αυτά όμως, η χρήση τους δεν είναι ευρέως διαδεδομένη λόγω της έλλειψης επεκτάσιμης αρχιτεκτονικής της μνήμης. Η δημοσίευσή τους, εισάγει μια νέα αρχιτεκτονική μνήμης, την CoRAM (Connected RAM), η οποία λειτουργεί ως γέφυρα μεταξύ των κατανεμημένων υπολογιστικών πυρήνων και των εξωτερικών διεπαφών της μνήμης. Η αρχιτεκτονική αυτή καθορίζει ένα φορητό περιβάλλον εφαρμογής το οποίο διαχωρίζει τους υπολογισμός, από τη διαχείριση μνήμης πάνω στο τσιπ.

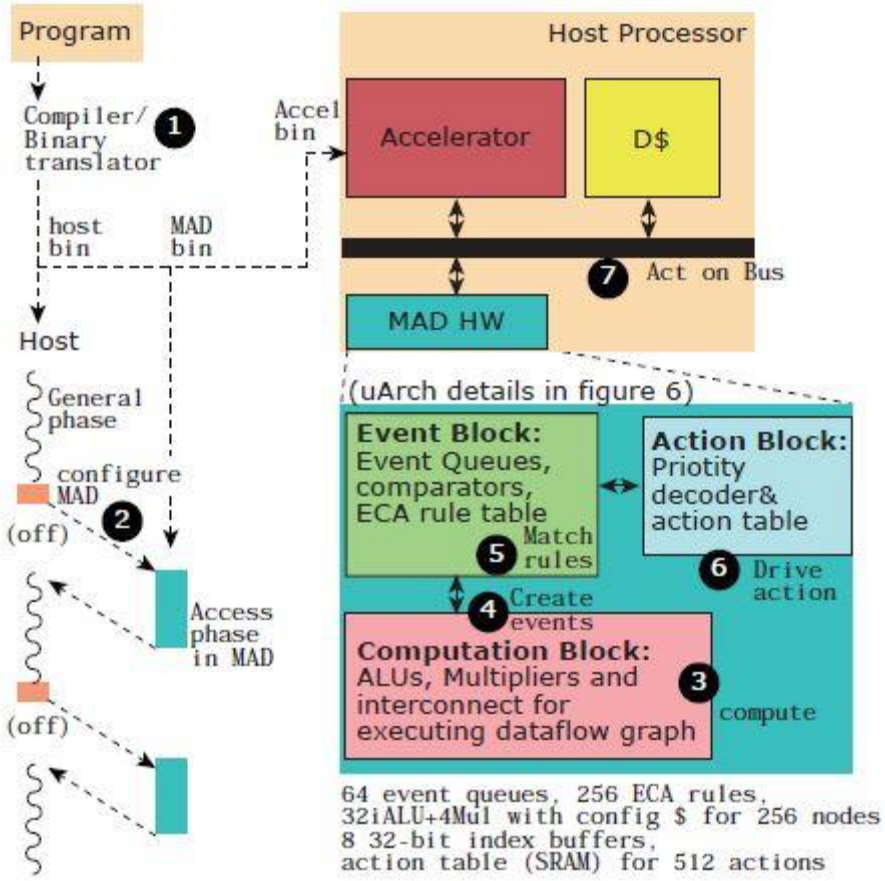


Εικόνα 14: Αρχιτεκτονική Μνήμης με υποστήριξη CoRAM [19]

Η αρχιτεκτονική αυτή αποτελείται από πέντε μέρη. Τις μνήμες CoRAM, οι οποίες διατηρούν τα χαρακτηριστικά μιας απλής SRAM, περίπου όπως στις τωρινές αρχιτεκτονικές μνήμης των σημερινών FPGA. Η διαφορά τους είναι ότι σε αντίθεση με τις συμβατικές SRAM, τα περιεχόμενα δεδομένων κάθε μιας CoRAM, διαχειρίζονται από μηχανές πεπερασμένων καταστάσεων που ονομάζονται νήματα ελέγχου. Τα νήματα αυτά, σχηματίζουν μια κατανεμημένη συλλογή από λογικές και

ασύγχρονες μηχανές πεπερασμένων καταστάσεων, οι οποίες μεσολαβούν στη μεταφορά δεδομένων από τις CoRAM, στη διεπαφή της μνήμης. Κάθε CoRAM διαχειρίζεται από έως ένα νήμα ελέγχου, ενώ ένα νήμα ελέγχου μπορεί να διαχειρίζει πολλαπλές CoRAM, με τα νήματα αυτά να είναι ο μόνος τρόπος πρόσβασης στην κύρια μνήμη. Επικοινωνούν με τη λογική του χρήστη με αμφίδρομο κανάλι, και ενημερώνουν τη λογική για το πότε μια CoRAM είναι έτοιμη για προσπέλαση μέσω των τοπικών διεπαφών SRAM. Οι προσβάσεις στη μνήμη εκφράζονται μέσω ενός συνόλου αρχήν μνήμης και αρχών επικοινωνίας, που ονομάζονται ενέργειες ελέγχου, και περιγράφουν λογικές εντολές μεταφοράς μνήμης μέσω συγκεκριμένων ενσωματωμένων CoRAM και τη διεπαφή της μνήμης. Αναλυτικότερα, τα νήματα ελέγχου μπορούν να χρησιμοποιήσουν εξειδικευμένες ενέργειες ελέγχου, με σκοπό την αύξηση του παραλληλισμού της μνήμης, ή να προσαρμόσουν τον διαμελισμό της μνήμης. Κάθε πυρήνας αυτής της αρχιτεκτονικής έχει μοναδικές προϋποθέσεις μοτίβου πρόσβασης μνήμης. Αυτό έχει ως αποτέλεσμα η ανάπτυξη εφαρμογών με τη χρήση αυτής τη εφαρμογής να βοηθάει στη συνολική προγραμματισιμότητα των βασισμένων σε FPGA εφαρμογών. Απαιτείται η ανάπτυξη βελτιστοποιημένων πυρήνων επεξεργασίας, όμως δεν απαιτείται οι σχεδιαστές να επέμβουν σε χαμηλότερα επίπεδα διαχείρισης μνήμης και κατανομής δεδομένων. Ουσιαστικά, η αρχιτεκτονική αυτή ενεργοποιεί την αλληλεπίδραση πολλαπλών υλοποιήσεων υλικού. Για την αξιολόγηση της εφαρμογής επιλέχθηκε προσομοιωτής της Virtex-6 LX760, της πιο δυνατής FPGA της Xilinx την προκειμένη στιγμή, και ως εφαρμογή, ο Πολλαπλασιασμός Αραιών πινάκων, η εφαρμογή με τις πιο εις βάθος απαιτήσεις μνήμης. Παρατηρήθηκε ότι για όλες τις συχνότητες, η μεγαλύτερη απόδοση επιτυγχάνεται σε σχεδίαση σε μικρό χώρο, καθώς και σε σχεδιάσεις που απαιτούν μεγαλύτερη ισχύ. Από την εργασία αυτή, οι συγγραφείς συμπέραναν ότι για την επιτυχία μιας σχεδίασης σε FPGA είναι απαραίτητη η κατάλληλη πρόσβαση στη μνήμη, για αυτό και η αρχιτεκτονική που προτείνεται υποστηρίζει προσβάσεις στη μνήμη μέσα από την ίδια την FPGA.

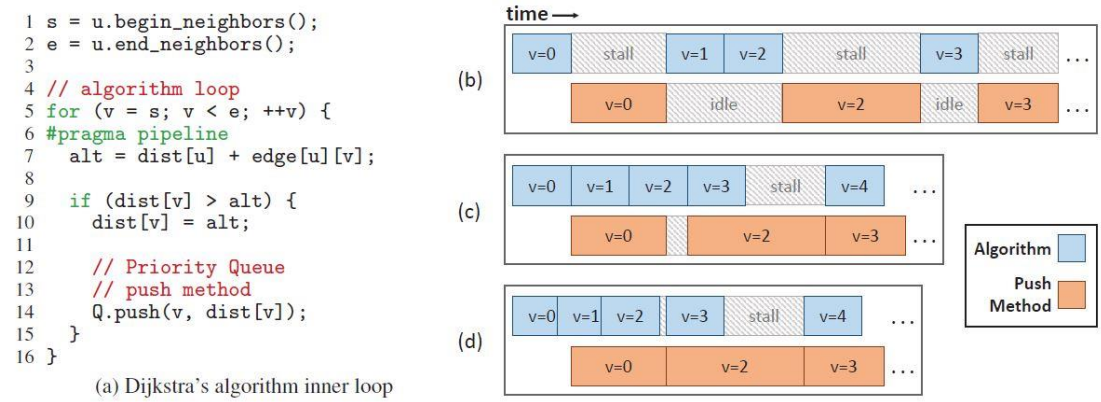
Οι C.-H. Ho, S. J. Kim και K. Sankaralingam το 2015 [20] ασχολήθηκαν επίσης με τις προσβάσεις στη μνήμη. Το μοντέλο εκτέλεσης που υλοποίησαν, ονομάζεται Ροή δεδομένων (dataflow) Πρόσβασης Μνήμης (MAD - Memory Access Dataflow). Το μοντέλο αυτό κωδικοποιεί τους υπολογισμούς ροής δεδομένων, τους κανόνες συνθηκών-ενεργειών και ξεκάθαρων ενεργειών. Η χρήση του έχει ως αποτέλεσμα να παρέχεται απόδοση δυναμικού πυρήνα με ένα πολύ μικρό μέρος κόστους ισχύος.



Εικόνα 15: Σύνοψη μοντέλου MAD [20]

Η αρχιτεκτονική αυτή αποτελείται από δύο τμήματα. Το πρώτο τμήμα είναι ψευδοκώδικας, ο οποίος αναλαμβάνει την αποστολή δεδομένων στην αναδιατασσόμενη αρχιτεκτονική μέσω ενός βρόχου επανάληψης. Το δεύτερο τμήμα, αυτό του υλικού, είναι υπεύθυνο για την εκτέλεση των ενεργειών και αποτελείται από τρία επιμέρους τμήματα. Το υπολογιστικό κομμάτι, το οποίο υπολογίζει τους ρυθμισμένους από πριν γράφους ροής δεδομένων με τέσσερις συναρτήσεις και τέσσερις διακόπτες, οι οποίοι σχηματίζουν ένα σύμπλεγμα. Κάθε σύμπλεγμα έρχεται σε επαφή με μια ουρά από συμβάντα και έχει υλοποιηθεί με διοχέτευση (pipeline) και έναν μηχανισμό ελέγχου ροής δεδομένων (dataflow). Η δίοδος των δεδομένων, περιλαμβάνει έτοιμα σήματα, με αποτέλεσμα να μην απαιτείται δυναμικός σχεδιασμός τελεστών ή συναρτησιακών μονάδων. Έτσι, η ρύθμιση της πληροφορίας περνά μέσα από στατικά μονοπάτια. Το κομμάτι συμβάντων ελέγχει τη δίοδο των δεδομένων με τη μονάδα φόρτωσης-αποθήκευσης, το κομμάτι υπολογισμού και τον επιταχυντή εκτέλεσης. Τρίτο και τελευταίο, είναι το κομμάτι ενεργειών, το οποίο ευθύνεται για τον έλεγχο της μετακίνησης δεδομένων μεταξύ της διόδου δεδομένων και της μονάδας φόρτωσης-αποθήκευσης. Ένα πλεονέκτημα της αρχιτεκτονικής αυτής, είναι ότι μπορεί με μικρές τροποποιήσεις να ενσωματωθεί σε ποικίλους επιταχυντές για την υλοποίηση της διεπαφής του πυρήνα τους. Η αποδοτικότητα της εφαρμογής, καθώς και ενσωματώσεών της, δοκιμάστηκε σε διάφορους επιταχυντές, με τη χρήση προσαρμοσμένων μοντέλων. Σε όλους παρουσιάστηκε επιτάχυνση στην εκτέλεση, καθώς και μείωση στην κατανάλωση της ενέργειας.

Οι υπαρκτές αρχιτεκτονικές υψηλού επιπέδου είναι ιδιαιτέρως αποδοτικές σε προγράμματα που χρησιμοποιούν στατικές δομές δεδομένων [21]. Για αυτό και οι R. Zhao, G. Liu, S. Srinath, C. Batten, Z. Zhang ασχολήθηκαν το 2016 στην εργασία τους με δυναμικές δομές δεδομένων, όπως ουρές, στοίβες και δέντρα. Η υλοποίηση των δομών αυτών περιλαμβάνει τη χρήση μεθόδων με εξαρτήσεις δεδομένων και ακανόνιστα μοτίβα πρόσβασης στη μνήμη. Για να ξεπεράσουν τα προβλήματα αυτά, διαχωρίζουν τις σύνθετες αυτές δομές από τον αλγόριθμο με τη χρήση μιας διεπαφής. Η σχεδίαση της διεπαφής αυτής βασίζεται σε δύο ιδέες. Τον διαχωρισμό προσβάσεων και υπολογισμών, και την αξιοποίηση του χονδρόκοκκου παραλληλισμού του υλικού. Αρχικά, ο αποστολέας λαμβάνει τις κλήσεις μεθόδων από τον αλγόριθμο στη μορφή μηνύματος το οποίο περιέχει πληροφορία για τις παραμέτρους και τον τύπο της μεθόδου. Κατόπιν, η πληροφορία αποκωδικοποιείται και η διεργασία προς υλοποίηση αποστέλλεται στη μονάδα εξιδεικευμένων μεθόδων. Η μονάδα αυτή εκτελεί τις μεθόδους πρόσβασης και μετάλλαξης με τη χρήση διαφορετικών καναλιών με σκοπό την υποστήριξη παράλληλων εκτελέσεων και διαφορετικών μηνυμάτων για κάθε μέθοδο. Κάθε μέθοδος πρόσβασης, δημιουργεί μια μονάδα τροποποιημένης μεθόδου πρόσβασης, ενώ κάθε μέθοδος μετάλλαξης, δημιουργεί μια μονάδα τροποποιημένης μεθόδου μετάλλαξης. Σκοπός των μονάδων αυτών είναι η παραγωγή αποτελεσμάτων για τον συλλέκτη, ο οποίος κατόπιν τα επιστρέφει στον αρχικό αλγόριθμο. Με τη χρήση της αρχιτεκτονικής αυτής υλοποιήθηκαν ποικίλες εφαρμογές με διαφορετικές απαιτήσεις. Για παράδειγμα, ο αλγόριθμος του Dijkstra, ο οποίος καλεί τη μέθοδο push όταν βρεθεί βέλτιστη διαδρομή.



Εικόνα 16: Εκτέλεση προγράμματος που περιέχει σύνθετη μέθοδο μετάλλαξης (a) Κώδικας εσωτερικού βρόχου αλγορίθμου Dijkstra (b) Βασική εκτέλεση χωρίς βελτιστοποίηση (c) Διαχωρισμένη εκτέλεση με επικαλύψεις που ελαχιστοποιεί περιττές καθυστερήσεις (d) Διοχέτευση του αλγορίθμου πετυχαίνοντας επιπλέον βελτιστοποίηση [21]

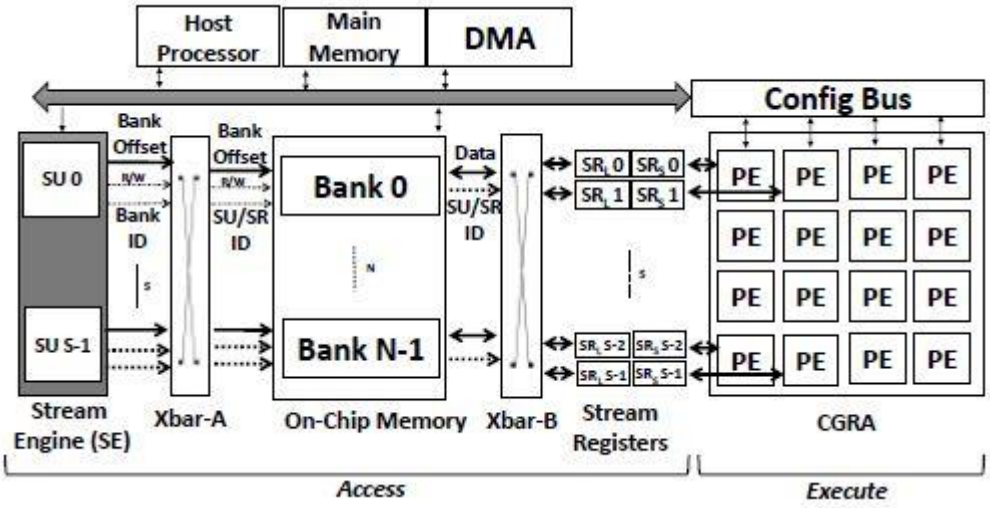
Με τη χρήση διαχωρισμού, είναι δυνατό να εκτελούνται πολλαπλές επαναλήψεις το αλγορίθμου μέχρι να έρθει η στιγμή να κληθεί εκ νέου η push. Επίσης, στην περίπτωση αναζήτησης κλειδιού σε τρεις λωρίδες, μπορεί να υπάρξει καθυστέρηση σε μια μέσω παράλληλης εκτέλεσης κρύβει τυχόν καθυστερήσεις και επιτρέπει σε δομές δεδομένων να καλούν συναρτήσεις malloc και free με αποτέλεσμα τη δυνατότητα μεταβολής του μεγέθους τους την ώρα της εκτέλεσης. Το πρώτο βήμα για την σύνθεση του αλγορίθμου είναι να αποφασιστεί ποιες μέθοδοι θα

διαχωριστούν, καθώς και αν είναι πρόσβασης ή μετάλλαξης. Από τον τύπο της μεθόδου καθορίζονται και τα μηνύματα αιτήσεων και απαντήσεων. Κατόπιν, οι μέθοδοι χρησιμοποιούνται για να δημιουργήσουν αυτόνομες συναρτήσεις οι οποίες παράγουν και τις τελικές μονάδες υλικού, με αποτέλεσμα οι δομές δεδομένων να μετατρέπονται σε μονάδες εξιδεικευμένων μεθόδων χωρίς να τροποποιηθεί ο αρχικός αλγόριθμος. Επιπλέον, ανάλογα με την διεκπεραιωτικότητα του αλγορίθμου καθορίζεται ο αριθμός των λωρίδων για παράλληλη εκτέλεση. Για να ελεγχθεί πειραματικά η απόδοση της αρχιτεκτονικής υλοποιήθηκε ο αλγόριθμος του Dijkstra, ένας αλγόριθμος ταξινόμησης στοιχείων σε ουρά προτεραιότητας και ο αλγόριθμος αναζήτησης σε γράφο DFS (Depth-first search). Όλοι τους χρησιμοποιούν αρκετά τις συναρτήσεις push και pop, οι οποίες διαχωρίζονται με σκοπό την αλληλοεπικαλυπτόμενη εκτέλεση, και παρουσίασαν επιτάχυνση κοντά στο 1,5. Επιπλέον υλοποιήθηκαν δύο πυρήνες οι οποίοι καλούν σύνθετες μεθόδους πρόσβασης. Ένας που ομαδοποιεί εικονοστοιχεία, και ένας πίνακας κατακερματισμού 2000 θέσεων. Επιτεύχθηκε επιτάχυνση της τάξης του 2 και 1,57 αντίστοιχα.

Η προσέγγιση των K. Koukos, P. Ekemark, G. Zacharopoulos, V. Spiliopoulos, S. Kaxiras και A. Jimborean το 2016 [22] στοχεύει στη βελτίωση της απόδοσης και την ενεργειακή αποδοτικότητα μέσω μιας μεθόδου την ώρα της εκτέλεσης η οποία διαχωρίζει την εφαρμογή χρησιμοποιώντας απλές και αποδοτικές ευρετικές μεθόδους. Το σύστημα που υλοποίησαν αποτελείται από έναν αναλυτή, ο οποίος παράγει πολλαπλές εκδόσεις πρόσβασης και δύο βιβλιοθήκες, οι οποίες υπολογίζουν το αντίκτυπο κάθε έκδοσης στην απόδοση και την ενέργεια αντίστοιχα. Βρόχοι επανάληψης μέσα στις συναρτήσεις, οι οποίοι περιέχουν σύνθετη ροή ελέγχου και προσβάσεις στη μνήμη οι οποίες εκτελούνται πλαγίως, ή με τη χρήση δεικτών, χωρίζονται σε κομμάτια από τον αναλυτή για να χωράνε στην κρυφή μνήμη του πυρήνα. Κατόπιν, για κάθε κομμάτι παράγονται φάσεις πρόσβασης και εκτέλεσης, οι οποίες εκτελούνται διαδοχικά. Μετά το πέρας της εκτέλεσης, επιλέγεται η έκδοση πρόσβασης με καλύτερη απόδοση. Η έκδοση πρόσβασης είναι ένας απλοποιημένος κλώνος του αρχικού κώδικα, ο οποίος περιέχει τον υπολογισμό διεύθυνσης και τις εντολές που συντηρούν τη ροή ελέγχου, ενώ κατά τη φάση εκτέλεσης προφορτώνονται τα δεδομένα που απαιτούνται από τη φάση εκτέλεσης. Κατόπιν αφαιρείται ο περιττός κώδικας. Οι διάφορες εκδόσεις πολλαπλών προσβάσεων έχουν ως σκοπό την εξισορρόπηση του κόστους και της αποδοτικότητας. Κάθε έκδοση εφαρμόζει διαφορετικό τρόπο επιλογής των δεδομένων για προφόρτωση (fetch). Πειραματικά αποδείχθηκε ότι η τεχνική αυτή, ακόμα και σε σύνθετο κώδικα πετυχαίνει σημαντική μείωση της κατανάλωσης ενέργειας και μικρή αύξηση της απόδοσης. Η συνολική αποδοτικότητα μετριέται με τη χρήση του γινομένου χρόνου και ενέργειας, μέγεθος το οποίο παρουσίασε βελτίωση από 20% έως 70%.

Σε πιο πρόσφατη εργασία, οι D. Wijerathne, Z. Li, M. Karunaratne, A. Pathania και T. Mitra [37] υλοποίησαν το CASCADE, μια σχεδίαση χονδρόκοκκης διαχωρισμένης πρόσβασης-εκτέλεσης. Η χρήση της αρχιτεκτονικής διαχωρισμένης πρόσβασης-

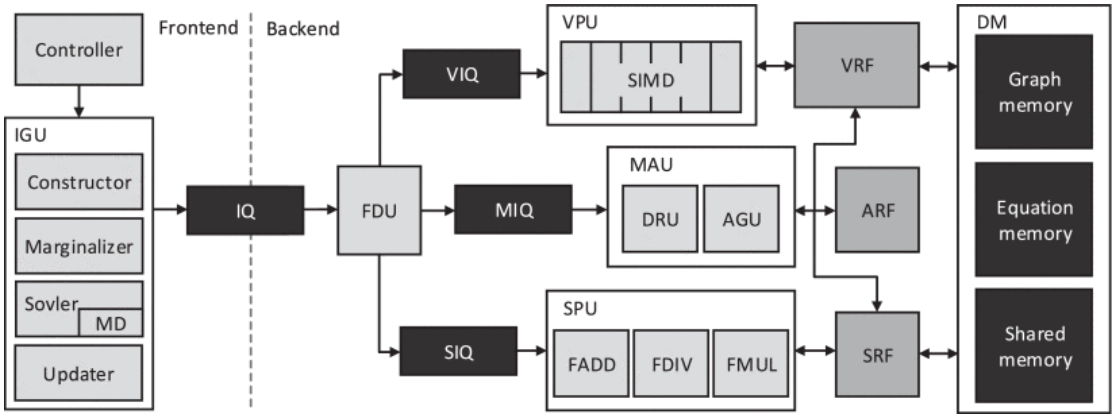
εκτέλεσης, στην περίπτωση αυτή εξυπηρετεί στην απομόνωση των προσβάσεων στη μνήμη από τους υπολογισμούς, αυξάνοντας έτσι την διεκπεραιωτικότητα, σε αντίθεση με μια συμβατική χονδρόκοκκη σχεδίαση.



Εικόνα 17: Η αρχιτεκτονική CASCADE

Την σχεδίαση του χονδρόκκοκου πίνακα αναδιατασσόμενης λογικής απαρτίζουν πολλαπλά Επεξεργαστικά Στοιχεία, τα οποία αποτελούνται από μια Αριθμητική και Λογική Μονάδα (ALU), ένα αρχείο καταχωρητών και μια μνήμη. Η αρχιτεκτονική αυτή στοχεύει στην επιτάχυνση των συχνών σύνθετων βρόχων επανάληψης. Η ιδέα του διαχωρισμού προήλθε από την κατανάλωση πολύτιμων πόρων εντός των Επεξεργαστικών Στοιχείων. Όμως, οι υφιστάμενοι μέθοδοι διαχωρισμού δεν υποστηρίζουν τις κατάλληλες μνήμες για υψηλό εύρος, ούτε δεν μπορούν να υποστηρίξουν προσβάσεις στη μνήμη δίχως συγκρούσεις, με αποτέλεσμα να μην επιτυγχάνεται η μέγιστη δυνατή απόδοση. Το CASCADE με τη χρήση τροποποιημένου υλικού υλοποιεί μνήμες με υψηλό εύρος και με διευρυμένο αλγόριθμο ανιχνεύει συγκρούσεις στις προσβάσεις μνήμης. Πειραματικά παρατηρήθηκε πως η σχεδίαση αυτή παρουσίασε κατά μέσο όρο βελτίωση 3x στην απόδοση και μείωση τρεις τάξεις μεγέθους στον χρόνο μεταγλώττισης σε σχέση με άλλες συμβατικές χονδρόκοκκες σχεδιάσεις.

Φέτος, οι R. Sun, P. Liu, J. Xue, S. Yang, J. Qian και R. Ying [38] κατάφεραν με τη συνεισφορά μιας Αρχιτεκτονικής Διαχωρισμένης Πρόσβασης-Εκτέλεσης να υλοποιήσουν το σύστημα BAX, έναν επιταχυντή υλικού.



Εικόνα 18: Η αρχιτεκτονική BAX [38]

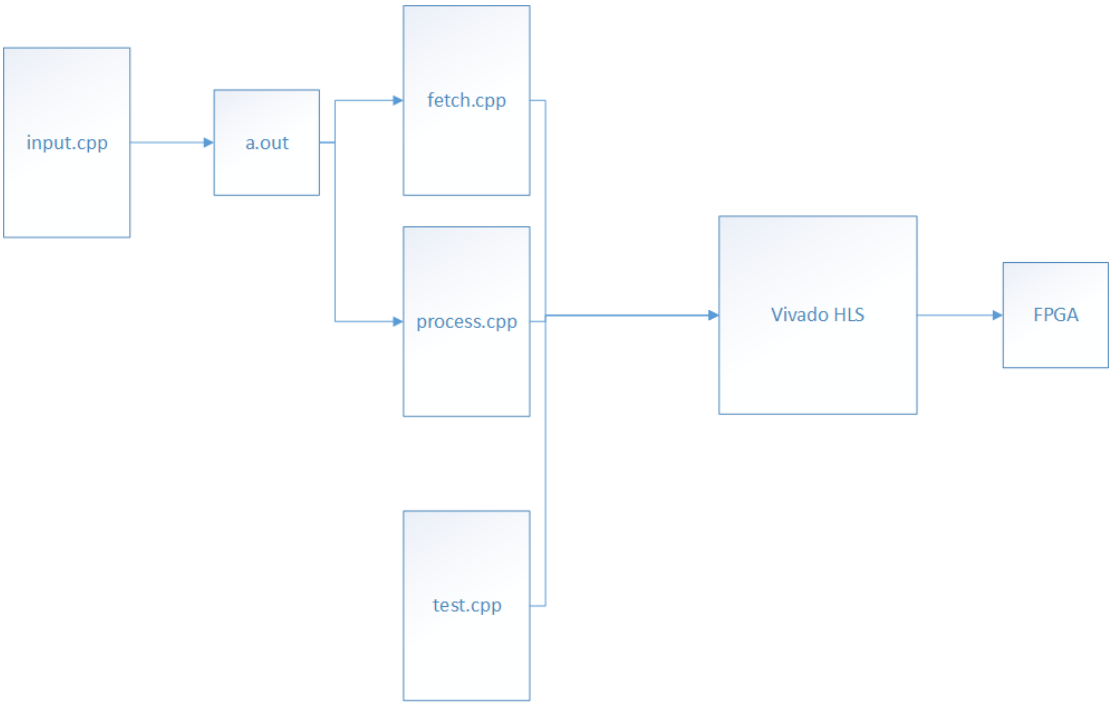
Το πρόβλημα που στοχεύουν να λύσουν είναι αυτό της Ρύθμισης Δεσμίδων, το οποίο αφορά στην ελαχιστοποίηση των σφαλμάτων όλων των ακμών μιας 3D κατασκευής μέσω του τυλίγματος από φως κάθε σημείου. Τέτοιες εφαρμογές περιλαμβάνουν αρκετές αντιμεταθέσεις και πολλαπλασιασμούς πινάκων. Η συνεισφορά της Αρχιτεκτονικής Διαχωρισμένης Πρόσβασης-Εκτέλεσης στο σύστημα BAX αποσκοπεί στην ελαχιστοποίηση της καθυστέρησης εσωτερικά στη μνήμη με την προφόρτωση πινάκων και διανυσμάτων. Ο επιταχυντής πέτυχε σημαντικές βελτιώσεις σε σχέση με την Κεντρική Μονάδα Επεξεργασίας (CPU) και τη Μονάδα Επεξεργασίας Γραφικών (GPU) του υπολογιστή.

Μόλις πριν από λίγους μήνες στο ίδρυμα, παρουσιάστηκε σχετική διπλωματική εργασία του ίδιου εργαστηρίου, του Εργαστηρίου Μικροεπεξεργαστών και Υλικού από τον φοιτητή Ι. Μοριανό με θέμα «Απεικόνιση επιταχυντών συστημάτων υψηλής απόδοσης στην πλατφόρμα HARP2 (Intel’s Hardware Accelerator Research Program) χρησιμοποιώντας την αρχιτεκτονική αποζευγμένης επεξεργασίας και πρόσβασης δεδομένων DAER» [39]. Χρησιμοποιεί Αρχιτεκτονική Διαχωρισμένης Πρόσβασης-Εκτέλεσης για την απεικόνιση στην πλατφόρμα HARP του αλγόριθμου Jacobi, ο οποίος ανήκει στους αλγόριθμους δομημένων πλεγμάτων που περιλαμβάνονται στη λίστα των 13 νάνων, οι οποίοι αντιπροσωπεύουν ενεργές περιοχές στον παράλληλο προγραμματισμό. Υλοποιούνται δύο εκδοχές της Αρχιτεκτονικής DAE. Η πρώτη χρησιμοποιεί δύο παράλληλες Μηχανές Πεπερασμένων Καταστάσεων (FSM – Finite State Machine), μια για την ανάγνωση δεδομένων από τη μνήμη και μια για τον υπολογισμό των λύσεων των εξισώσεων Laplace και την εγγραφή τους στη μνήμη. Η δεύτερη χρησιμοποιεί μια τρίτη Μηχανή Πεπερασμένων Καταστάσεων (FSM) για την εγγραφή των δεδομένων στη μνήμη για να μην περιμένει η Μηχανή Πεπερασμένων Καταστάσεων (FSM) της Επεξεργασίας απόκριση από την αίτηση εγγραφής και προχωράει στον επόμενο υπολογισμό. Τα πειράματα που πραγματοποιήθηκαν έδειξαν πως αυτές οι παράλληλες αρχιτεκτονικές πετυχαίνουν μέχρι και 2x επιτάχυνση της απόδοσης σε σύγκριση με την αντίστοιχη σειριακή υλοποίηση.

ΚΕΦΑΛΑΙΟ 3-ΔΟΜΗ ΚΑΙ ΛΕΙΤΟΥΡΓΙΚΑ ΧΑΡΑΚΤΗΡΙΣΤΙΚΑ

3.1 Λειτουργικότητα εργαλείου

Σκοπός της παρούσας διπλωματικής είναι η απεικόνιση εφαρμογών σε υλικό με τη χρήση αρχιτεκτονικής DAE. Η απεικόνιση αυτή επιτυγχάνεται μέσω του διαχωρισμού των πιο σύνθετων πράξεων του κώδικα εισόδου της εφαρμογής λογισμικού σε συναρτήσεις fetch και process. Στα πλαίσια των εφαρμογών που εκτελούνται στο Εργαστήριο Μικροεπεξεργαστών και Υλικού της σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Πολυτεχνείου Κρήτης οι συναρτήσεις αυτές γράφονται με το χέρι. Προέκυψε έτσι η ιδέα με κάποιο τρόπο οι συναρτήσεις αυτές να παράγονται αυτόματα. Ο πιο αποδοτικός τρόπος για να γίνει αυτό κρίθηκε πως ήταν με τη χρήση ενός λεκτικού και συντακτικού αναλυτή. Επειδή οι περιγραφές τους καθορίζουν τι θεωρείται λεκτικά και συντακτικά ορθό, υπάρχει η δυνατότητα τοποθέτησης επιπλέον λειτουργικότητας αν υπάρχουν στον κώδικα εισόδου συγκεκριμένα στοιχεία. Οπότε, θεωρήθηκε πως θα εξυπηρετούσε τις ανάγκες της ιδέας αυτής η χρήση ενός τροποποιημένου λεκτικού και συντακτικού αναλυτή που θα διαβάζει κώδικα εισόδου και μόλις εντοπιστεί το σύνθετο κομμάτι κώδικα για απεικόνιση υλικού, με την ανάγνωση των εντολών θα παράγονται αυτόματα τα αρχεία των δύο συναρτήσεων fetch και process. Κατόπιν οι δύο αυτές συναρτήσεις ενώνονται μέσω κώδικα υψηλού επιπέδου για υλικό με σκοπό την σύνθεσή τους από το Vivado HLS και την τελική απεικόνιση της εφαρμογής στην επιθυμητή FPGA.



Εικόνα 19: Λειτουργικότητα εργαλείου

Πρώτο βήμα ήταν, όπως προαναφέρθηκε, η τροποποίηση της περιγραφής λεκτικού και συντακτικού αναλυτή της γλώσσας C με σκοπό την παραγωγή νέου ο οποίος παρέχει επιπλέον λειτουργικότητα και δημιουργεί τα απαραίτητα αρχεία (a.out). Ο νέος λεκτικός και συντακτικός αναλυτής διαβάζει αρχείο .cpp γραμμένο σε C (input.cpp) με οδηγία στις πράξεις που θα εκτελεστούν σε υλικό και παράγει τα αρχεία των συναρτήσεων fetch και process, οι οποίες υλοποιούν αυτήν την εκτέλεση. Στη συνέχεια, το αρχείο αυτό τροποποιείται με μέθοδο που θα περιγραφεί σε παρακάτω κεφάλαιο (test.cpp) ώστε να είναι σύμφωνο με τις προδιαγραφές του Vivado HLS. Το αρχείο αυτό, καθώς και οι έτοιμες από το εργαλείο συναρτήσεις απαρτίζουν ένα Vivado HLS project μέσω του οποίου υλοποιείται η σύνθεση για εκτέλεση της εφαρμογής σε επιλεγμένη FPGA. Με τον τρόπο αυτό επιτυγχάνεται η απεικόνιση εφαρμογών σε υλικό με τη χρήση της αρχιτεκτονικής DAE με απλούστερο τρόπο τόσο σε σχεδιαστές λογισμικού, καθώς τους επιτρέπει να εκμεταλλευτούν τα πλεονεκτήματα του υλικού στις σχεδιάσεις τους χωρίς να τους είναι απαραίτητη η γνώση κάποιας γλώσσας υλικού, αλλά και στους σχεδιαστές του υλικού, καθώς τους δίνει τη δυνατότητα να αυτοματοποιήσουν τις σχεδιάσεις τους με τη λήψη μέσω του εργαλείου έτοιμου κώδικα ο οποίος μέχρι πρότινος θα έπρεπε να γραφεί με το χέρι.

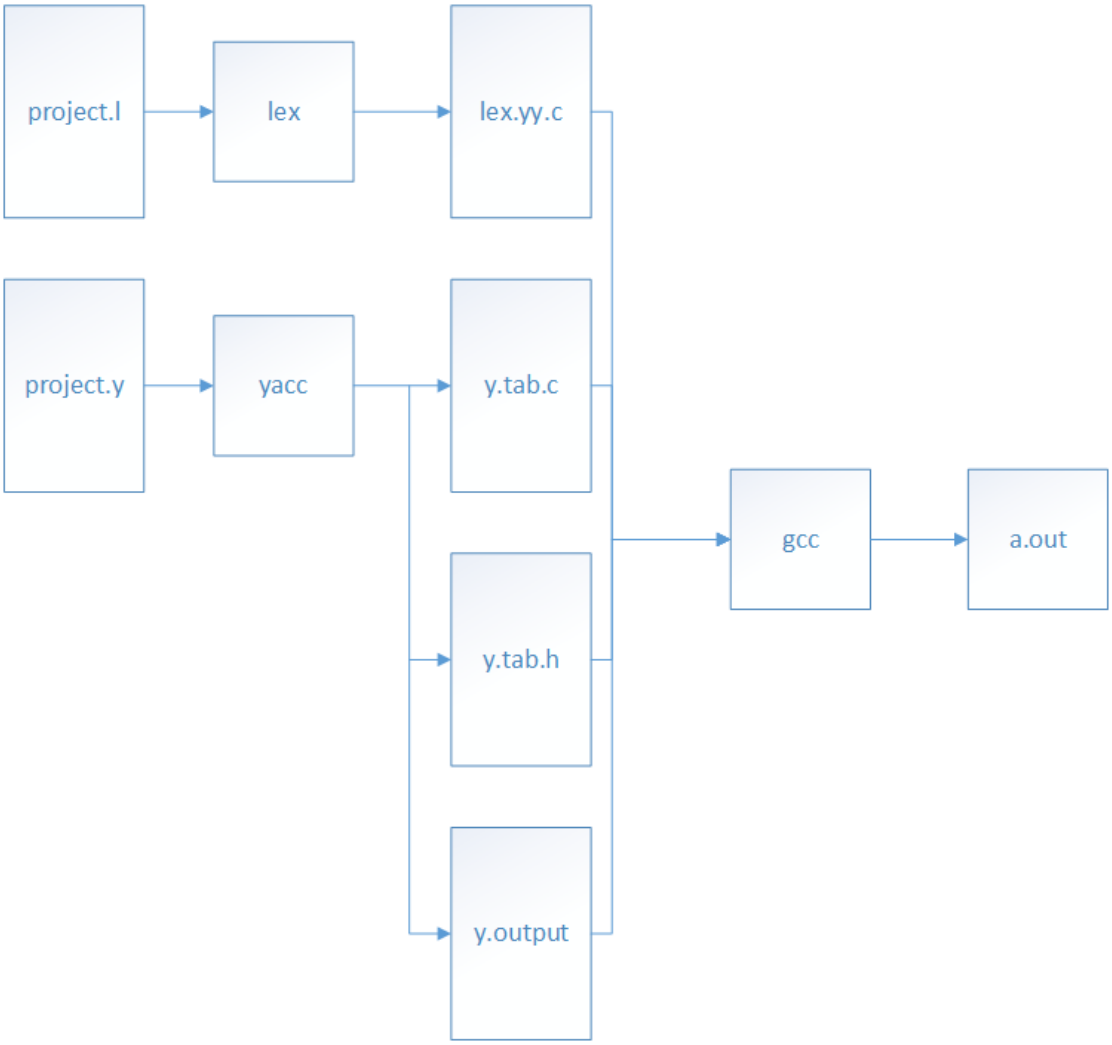
3.2 Λειτουργικά χαρακτηριστικά εργαλείου

3.2.1 Λειτουργικά χαρακτηριστικά έτοιμου λεκτικού και συντακτικού αναλυτή

Πριν από το 1975 η συγγραφή ενός λεκτικού και συντακτικού αναλυτή (parser) ήταν μια αρκετά χρονοβόρα διαδικασία [26]. Αυτή ήταν και η χρονιά που κυκλοφόρησαν οι δημοσιεύσεις των Lesk και Johnson για τα εργαλεία lex και yacc, τα οποία απλοποίησαν αρκετά τη συγγραφή συντακτικών αναλυτών. Συντακτικοί αναλυτές ονομάζονται οι μηχανές οι οποίες αποφασίζουν σωστά εάν μια συμβολοσειρά (string) ανήκει σε μία γλώσσα χωρίς συμφραζόμενα και στην περίπτωση θετικής απάντησης παράγουν το αντίστοιχο συντακτικό δέντρο [25]. Οι αναλυτές διαβάζουν τον κώδικα εισόδου, τον χωρίζουν σε σύμβολα (tokens) και κατασκευάζουν το δέντρο σύνταξης του κώδικα εισόδου. Ήταν απαραίτητο να βρεθεί ένας έτοιμος λεκτικός και συντακτικός αναλυτής της γλώσσας C ο οποίος να κάνει ήδη αυτή τη δουλειά προκειμένου να τροποποιηθεί και να επεξεργάζεται όποια σύμβολα (tokens) είναι απαραίτητο για να εξυπηρετηθούν οι ανάγκες της διπλωματικής. Μετά από αναζήτηση στο διαδίκτυο βρέθηκε ένας συντακτικός και λεκτικός αναλυτής της γλώσσας C. Ο αναλυτής αυτός επιλέχθηκε διότι κρίθηκε απλός και μετά από δοκιμές αποτελεσματικός.

Βάση του λεκτικού και συντακτικού αναλυτή είναι δύο αρχεία, ένα αρχείο l. που περιέχει την περιγραφή του λεκτικού αναλυτή και ένα αρχείο .y που περιέχει την περιγραφή του συντακτικού αναλυτή. Για την παραγωγή των αναλυτών απαιτείται η εκτέλεση των παραπάνω αρχείων με lex και yacc αντίστοιχα. Η εκτέλεση του αρχείου .y με yacc, παράγει τον συντακτικό αναλυτή σε C (y.tab.c) και δύο άλλα βοηθητικά

αρχεία, τα y.tab.h και y.output. Η εκτέλεση του αρχείου .l με lex παράγει τον λεκτικό αναλυτή σε C, το αρχείο lex.yy.c. Κατόπιν, με την εκτέλεση του αναλυτή της C, του gcc παράγεται ο λεκτικός και συντακτικός αναλυτής της γλώσσας, ο οποίος ονομάστηκε a.out. Η λειτουργία αυτή απεικονίζεται στο παρακάτω διάγραμμα:



Εικόνα 20: Παραγωγή του τροποποιημένου λεκτικού και συντακτικού αναλυτή

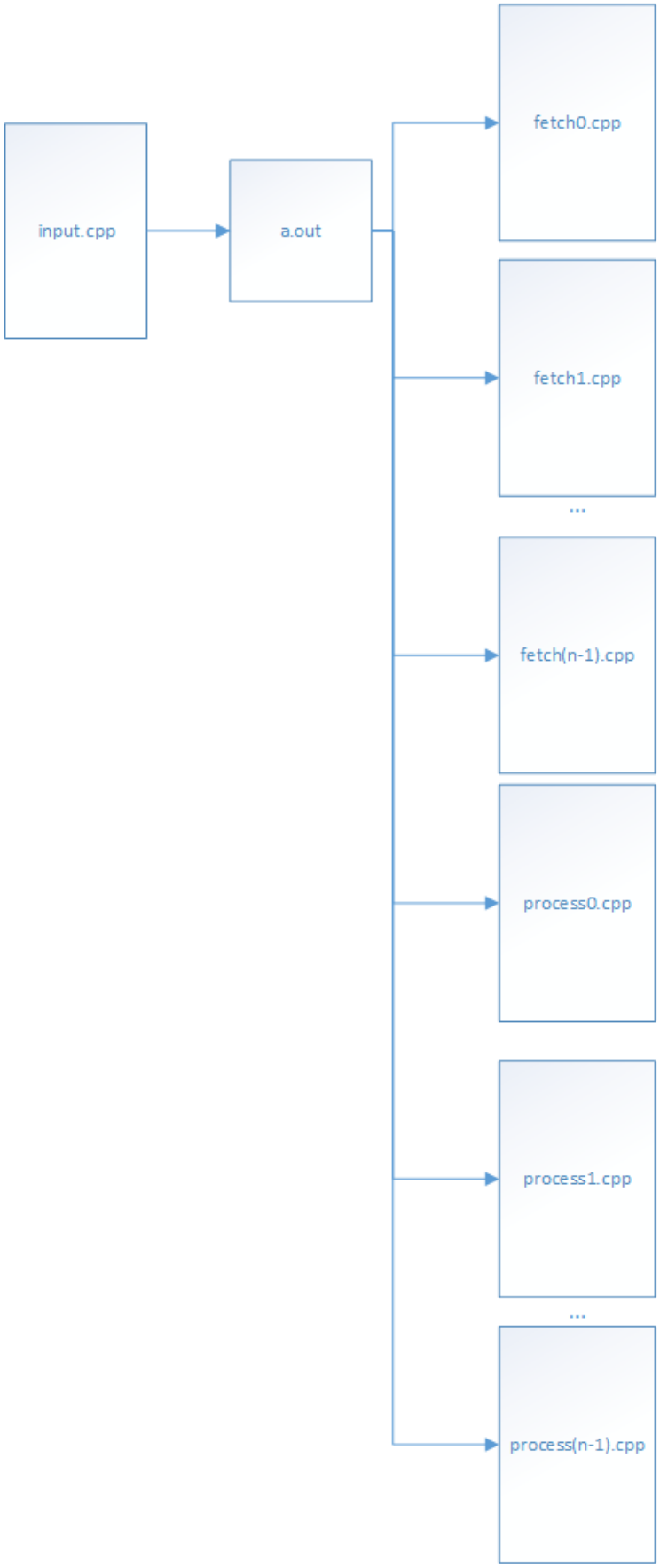
Η περιγραφή του λεκτικού αναλυτή ουσιαστικά χωρίζει τον κώδικα εισόδου σε σύμβολα (tokens). Αρχικά εμπεριέχει τη δήλωση απαραίτητων βιβλιοθηκών της C. Ακολουθεί το κομμάτι των ορισμών. Αρχικά, περιλαμβάνει την περιγραφή ορισμών των βιβλιοθηκών, καθώς και τυχών σταθερών, διότι αυτά τα δύο περιλαμβάνονται στην αρχή ενός προγράμματος C. Ακολουθεί λίστα των τύπων μεταβλητών, καθώς και άλλων λέξεων-κλειδιά που χρησιμοποιούνται από τη γλώσσα C και δεν μπορούν να αποτελέσουν το όνομα μιας μεταβλητής. Το μέρος των ορισμών ολοκληρώνεται με τη δήλωση των υπόλοιπων στοιχείων ενός προγράμματος C, τα οποία είναι τελεστές, σημεία στίξης, μεταβλητές και σχόλια. Για κάθε ένα από αυτά τα στοιχεία που προαναφέρθηκαν, επιστρέφεται το σύμβολο (token) που περιγράφει τον τύπο του. Τρίτο και τελευταίο κομμάτι της περιγραφής του λεκτικού αναλυτή είναι λοιπός κώδικας σε C. Το συγκεκριμένο αρχείο περιείχε την εκτέλεση της συνάρτησης

yywrap(). Η συνάρτηση αυτή επιστρέφει τον ακέραιο 1, πράγμα που σημαίνει τον τερματισμό της λεκτικής ανάλυσης.

Η περιγραφή του συντακτικού αναλυτή καθορίζει την ιεραρχική δομή του προγράμματος και αποτελείται από τέσσερα μέρη. Αρχίζει και αυτός με τη δήλωση απαραίτητων βιβλιοθηκών της C, μαζί με τη δήλωση μεταβλητών και συναρτήσεων που θα χρησιμοποιηθούν αργότερα. Ακολουθεί η δήλωση ως σύμβολο (token) όλων των στοιχείων που δηλώθηκαν στην περιγραφή του λεκτικού αναλυτή, δηλαδή τα τερματικά σύμβολα (tokens), καθώς και δηλώσεις προτεραιότητας και προσεταιριστικότητας, αλλά και το αρχικό σύμβολο (token) program_unit. Στη συνέχεια, περιγράφονται οι κανόνες σύνταξης. Η δομή των κανόνων είναι δενδρική. Στη ρίζα του δέντρου βρίσκεται το αρχικό σύμβολο (token) program_unit, και ακολουθούν διάφορα πιθανά στοιχεία, δημιουργώντας παρακλάδια στο δέντρο. Κάθε κανόνας εμπεριέχει έναν ή περισσότερους κανόνες-παρακλάδια, μπορεί και τον εαυτό του, δημιουργώντας έτσι μονοπάτια στο δέντρο, τα οποία αποτελούν εκδοχές αποδεκτών προγραμμάτων σε C. Τελευταίο κομμάτι της περιγραφής του λεκτικού και συντακτικού αναλυτή είναι λοιπός κώδικας σε C. Αρχικά, καλείται η yyparse(), η οποία επιστρέφει 0 αν η συντακτική ανάλυση ολοκληρωθεί χωρίς σφάλματα, διαφορετικά επιστρέφει 1. Στην περίπτωση της ολοκλήρωσης της συντακτικής ανάλυσης χωρίς σφάλματα, εμφανίζεται ανάλογο μήνυμα. Αν όμως υπάρχει κάποιο σφάλμα, εκτελείται η yyerror(), η οποία με τη χρήση της ακέραιας μεταβλητής yylineno, αλλά και της συμβολοσειράς (string) msg την οποία παίρνει ως είσοδο, εμφανίζει τη γραμμή στην οποία εντοπίστηκε το σφάλμα, καθώς και σχετικό μήνυμα.

3.2.2 Λειτουργικά χαρακτηριστικά τροποποιημένου λεκτικού και συντακτικού αναλυτή

Από αυτά τα δύο αρχεία, τις περιγραφές του λεκτικού και συντακτικού αναλυτή, τα οποία τροποποιήθηκαν όπως περιγράφεται σε παρακάτω κεφάλαιο για να παράγουν τα επιθυμητά αποτελέσματα, παράγονται οι δύο αναλυτές, καθώς και ο τελικός αναλυτής. Κατόπιν εκτελείται ο λεκτικός και συντακτικός αναλυτής με αρχείο εισόδου τον κώδικα C που περιέχει την οδηγία. Η διαδικασία αυτή παράγει δύο αρχεία κώδικα C για τις συναρτήσεις fetch και process. Τα αρχεία αυτά περιέχουν τις αντίστοιχες συναρτήσεις. Η οδηγία ενδέχεται να βρίσκεται παραπάνω από μια φορές στον κώδικα εισόδου με σκοπό τον διαχωρισμό περισσοτέρων από μιας βρόχων επανάληψης με πράξεις. Στην περίπτωση αυτή για κάθε οδηγία παράγονται ξεχωριστά αρχεία fetch και process. Την πρώτη φορά που θα εντοπιστεί η οδηγία θα παραχθούν τα αρχεία fetch0.cpp και process0.cpp, τη δεύτερη τα αρχεία fetch1.cpp και process1.cpp, την τρίτη τα fetch2.cpp και process2.cpp κ.ο.κ. Οπότε ο συνολικός αριθμός αρχείων κώδικα C που παράγονται είναι διπλάσιος του αριθμού των οδηγιών που εμπεριέχονται στον κώδικα εισόδου. Για n αριθμό οδηγιών DIRECTIVES το εργαλείο λειτουργεί σύμφωνα με το παρακάτω διάγραμμα:



Εικόνα 21: Παραγωγή αρχείων των συναρτήσεων με τη χρήση του τροποποιημένου λεκτικού κα ισυντακτικού αναλυτή

Τα αρχεία εξόδου περιλαμβάνουν και τις οδηγίες (pragmas) που απαιτούνται για να τρέξει σε εργαλείο σύνθεσης υλικού υψηλού επιπέδου (Vivado HLS).

3.2.3 Τύποι και προδιαγραφές αρχείων εισόδου

Το εργαλείο δέχεται οποιοδήποτε αρχείο .c ή .cpp, ένα κάθε φορά, το οποίο περιέχει κώδικα C/C++ και τουλάχιστον μια φορά τη λέξη-κλειδί DIRECTIVES με τον αντίστοιχο κώδικα σε αγκύλες προκειμένου να παραχθούν τα αρχεία των συναρτήσεων fetch και process. Το εργαλείο όμως είναι υλοποιημένο με τέτοιο τρόπο που υπάρχουν ορισμένοι περιορισμοί για να λειτουργήσει σωστά και επεξεργάζεται συγκεκριμένες πράξεις στα πλαίσια αυτής της διπλωματικής.

Πρώτα από όλα, οι μεταβλητές οι οποίες θα χρησιμοποιηθούν σε κάποια πράξη εντός της οδηγίας DIRECTIVES πρέπει να δηλωθούν σε ξεχωριστή γραμμή η κάθε μια και χωρίς αρχικοποίηση. Αυτό έγινε διότι ο κώδικας που αποθηκεύει τις μεταβλητές και τους τύπος τους σε πίνακες τοποθετήθηκε στον κανόνα που αναγνωρίζει πρώτα τον τύπο μιας μεταβλητής, μετά το όνομά της, και τέλος το ερωτηματικό.

Επίσης, οι βρόχοι επανάληψης που θα συμπεριλαμβάνονται μέσα στην οδηγία DIRECTIVES είναι απαραίτητο να είναι της μορφής FOR ‘(‘ exp ‘;’ exp ‘;’ exp ‘)’ stat, όπου exp οι εκφράσεις του βρόχου, πχ i=0, i<10, i++ και stat το σώμα του βρόχου με τις πράξεις που απαιτείται να εκτελεστούν σε κάθε επανάληψη. Όπως περιγράφεται και στον κανόνα του stat, μπορεί να εμπεριέχει βρόχους επανάληψης, δίνοντας έτσι τη δυνατότητα ύπαρξης εμφωλευμένων βρόχων. Ο λόγος είναι ίδιος με τις μεταβλητές, ο κώδικας που επεξεργάζεται τους βρόχους, λαμβάνει τα απαραίτητα στοιχεία και τους αποθηκεύει για εκτύπωση στα τελικά αρχεία των συναρτήσεων περιλαμβάνεται στον κανόνα που αναγνωρίζει βρόχο επανάληψης της μορφής που αναφέρθηκε νωρίτερα στην παράγραφο.

Οι πράξεις που αναγνωρίζει και επεξεργάζεται ο λεκτικός και συντακτικός αναλυτής περιλαμβάνουν απαραιτήτως τον τελεστή ισότητας. Επιπλέον, κάθε γραμμή μπορεί να περιέχει έναν ή περισσότερους τελεστές βασικών πράξεων (+, -, *, /). Επίσης, μπορεί να περιλαμβάνει σύνθετους, τους +=, -=, *=, /=. Οι πράξεις αυτές μπορεί να εκτελούνται μεταξύ σταθερών μεταβλητών, στοιχείων πινάκων, ή και αριθμών. Οι αριθμοί μπορεί να είναι ακέραιοι ή δεκαδικοί. Περιορισμοί στα ονόματα πέραν από τους περιορισμούς της ίδιας της γλώσσας C δεν υπάρχουν. Για ευκολία όμως στον διαχωρισμό μεταξύ πράξεων και μεταβλητών είναι απαραίτητο οι τελεστές, οι αριθμοί και οι μεταβλητές να βρίσκονται ανάμεσα σε κενούς χαρακτήρες (space).

Στην παρούσα φάση όμως ο λεκτικός και συντακτικός αναλυτής δεν επεξεργάζεται άλλες εντολές, για παράδειγμα συνθήκης ή εκτύπωσης, όπως και πράξεις ή αναθέσεις εκτός έστω και ενός βρόχου, παρά μόνο όσες περιλαμβάνονται εντός όλων των βρόχων.

3.3 Περιγραφή και λειτουργικά χαρακτηριστικά εργαλείων

Για την επεξεργασία του λεκτικού και συντακτικού αναλυτή χρησιμοποιήθηκαν τα lex και yacc. Όχι απαραίτητα γιατί έχουν κάτι καλύτερο από τις νεότερες εκδόσεις τους, flex και bison, αλλά γιατί ο συγγραφέας του έτοιμου λεκτικού και συντακτικού αναλυτή παρείχε εντολές για εκτέλεση της περιγραφής συντακτικού και λεκτικού αναλυτή με yacc και lex αντίστοιχα. Δεν υπήρξε κάποιο πρόβλημα με την εκτέλεση των εντολών αυτών, οπότε κρίθηκε πως δεν υπάρχει λόγος να μην χρησιμοποιηθούν τα lex κα yacc.

3.3.1 Το εργαλείο lex

Το lex, είναι ένα μεταεργαλείο [24], δηλαδή είναι ένα εργαλείο-παραγωγός άλλων εργαλείων. Είναι σχεδιασμένο να επεξεργάζεται λεκτικά ροές χαρακτήρων εισόδου [27] και παράγει λεκτικούς αναλυτές. Παίρνει ως είσοδο ένα αρχείο τύπου .l το οποίο περιέχει ένα σύνολο από κανονικές εκφράσεις και ενέργειες και γράφεται από το χρήστη. Το αρχείο C που παράγει, το lex.yy.c, υλοποιεί τον τελικό λεκτικό αναλυτή όπως αυτός περιεγράφηκε στο αρχείο .l. Πιο συγκεκριμένα, ο λεκτικός και συντακτικός αναλυτής υλοποιείται μέσω της συνάρτησης yylex(). Η συνάρτηση αυτή, καλείται από τον προγραμματιστή μέσα στη συνάρτηση main() στο αρχείο .l. Η λειτουργία του λεκτικού και συντακτικού αναλυτή είναι η αναγνώριση εκφράσεων μέσα στις ροές χαρακτήρων εισόδου, ο διαχωρισμός τους σε συμβολοσειρές (strings) και η αντιστοίχισή τους με τις εκφράσεις που βρίσκονται στην περιγραφή του λεκτικού αναλυτή. Υπάρχει η δυνατότητα ανάμεσα στις εκφράσεις μέσα σε αγκύλες να γράφεται κώδικας από τον χρήστη, ο οποίος εκτελείται μόλις εμφανιστεί η αντίστοιχη έκφραση. Ο αναλυτής που παράγει το lex είναι αρκετά γρήγορος, επειδή το ντετερμινιστικό πεπερασμένο αυτόματο που παράγεται μεταφράζεται αντί να μεταγλωττίζεται. Η ταχύτητα της αναγνώρισης και του διαχωρισμού της ροής χαρακτήρων εισόδου εξαρτάται από το μέγεθος της εισόδου και από μέγεθος του αρχείου που παράγει το lex και όχι από τους κανόνες που γράφτηκαν στο lex ή την πολυπλοκότητά τους.

3.3.2 Το εργαλείο yacc

Το εργαλείο yacc είναι όπως και το εργαλείο lex ένα μεταεργαλείο [24], το οποίο παράγει γλωσσικούς επεξεργαστές και συντακτικούς αναλυτές. Είναι ένα γενικότερο εργαλείο περιγραφής εισόδου ενός άλλου προγράμματος [28]. Η είσοδός του είναι ένα αρχείο .y, το οποίο περιέχει δηλωτική περιγραφή της γραμματικής που θα αναγνωρίσει ο συντακτικός αναλυτής καθώς και σημασιολογικούς κανόνες. Μέσω των κανόνων αυτών διαμορφώνεται η δομή του αρχείου εισόδου. Όπως και στο lex, δίνεται η δυνατότητα σε κάθε κανόνα εντός αγκυλών να συμπεριληφθεί κώδικας ο οποίος εκτελείται μόλις εντοπιστεί ο αντίστοιχος κανόνας. Ως έξοδο, δημιουργεί τα παρακάτω τρία αρχεία: ένα αρχείο σε C με όνομα y.tab.c που υλοποιεί το συντακτικό

αναλυτή που περιεγράφηκε με το αρχείο .y, ένα αρχείο επικεφαλίδας σε C με όνομα y.tab.h που ορίζει σταθερές για το λεκτικά (τερματικά) σύμβολα (tokens) και ένα αρχείο κειμένου με όνομα y.output που επεξηγεί τις καταστάσεις και τις μεταπτώσεις του αυτομάτου. Πιο συγκεκριμένα, ο αναλυτής υλοποιείται μέσω της συνάρτησης yyparse(). Η συνάρτηση αυτή, καλείται από τον προγραμματιστή μέσα στη συνάρτηση main() στο αρχείο .y.

3.3.3 Το Vivado HLS

Το Vivado HLS (High Level Synthesis) είναι ένα εργαλείο το οποίο μετατρέπει κώδικα C και παραλλαγών της σε υλοποίηση RTL (register-transfer level) με σκοπό τη σύνθεση σε FPGA της Xilinx [35]. Η χρήση του ωφελεί προγραμματιστές τόσο υλικού όσο και λογισμικού. Οι προγραμματιστές υλικού πετυχαίνουν αύξηση της παραγωγικότητας με τη δυνατότητα εργασίας σε υψηλότερο επίπεδο αφαίρεσης δημιουργώντας υλικό υψηλών επιδόσεων, ενώ οι προγραμματιστές λογισμικού έχουν τη δυνατότητα να επιταχύνουν την εκτέλεση υπολογιστικά πολύπλοκων τμημάτων αλγορίθμων με τη μεταγλώττισή τους σε FPGA.

Το Vivado HLS λειτουργεί σε τρία στάδια. Αρχικά, καθορίζει ποιες λειτουργίες θα εκτελεστούν σε ποιο κύκλο ρολογιού με βάση το μέγεθος του κύκλου ή τη συχνότητα του ρολογιού, το χρόνο που απαιτείται για την περάτωση της λειτουργίας και τυχόν οδηγίες βελτιστοποίησης από το χρήστη. Στη συνέχεια αντιστοιχίζει τις λειτουργίες στις δομές του υλικού με στόχο την επίτευξη της καλύτερης δυνατής λύσης. Τέλος, δημιουργεί την μηχανή πεπερασμένων καταστάσεων που υλοποιεί τις διαδικασίες με σχεδίαση RTL.

Η σύνθεση της C επιτυγχάνεται με την αντιστοίχιση των παραμέτρων της κορυφαίας συνάρτησης (top function) σε RTL πύλες εισόδου-εξόδου, τη μετατροπή συναρτήσεων σε blocks, οι πίνακες αποθηκεύονται σε μνήμες RAM της FPGA. Όσον αφορά τους βρόχους επανάληψης, το εργαλείο παράγει τη λογική για τον έναν βρόχο και η RTL σχεδίαση εκτελεί τη λογική αυτή για όλες τις επαναλήψεις. Παρέχεται η δυνατότητα, όπως θα περιγραφεί παρακάτω, με τη χρήση του unroll οι επαναλήψεις να εκτελεστούν παράλληλα.

Το Vivado HLS δέχεται ως είσοδο αρχείο γραμμένο σε C ή κάποια παραλλαγή της, της το οποίο είναι η βασική είσοδος του Vivado HLS και περιέχει μια συνάρτηση, η οποία μπορεί να περιλαμβάνει και μια ιεραρχία υπο-ρουτίνων, υποχρεωτικά περιορισμούς όσον αφορά την περίοδο του ρολογιού, την αβεβαιότητα του ρολογιού και την FPGA για την οποία προορίζεται ο κώδικας εισόδου, τυχόν οδηγίες για να κατευθύνουν τη σύνθεση και να υλοποιήσουν συγκεκριμένη συμπεριφορά και βελτιστοποίηση και ένα δεύτερο αρχείο C το οποίο προορίζεται για τον έλεγχο (test bench) και την προσομοίωση της συνάρτησης C για να επιβεβαιωθεί η έξοδος RTL. Στον κώδικα εισόδου, όπως και σε κάθε πρόγραμμα C, η κορυφαία συνάρτηση (top function) είναι

η main(). Η main() δεν είναι δυνατόν να οριστεί ως κορυφαία συνάρτηση (top function) της σύνθεσης, δύναται όμως αντί για αυτήν να οριστεί οποιαδήποτε άλλη συνάρτηση του προγράμματος. Μια μόνο ορίζεται ως κορυφαία συνάρτηση (top function) της σύνθεσης, οι υπόλοιπες απλώς περνάνε από τη διαδικασία της σύνθεσης.

Με το πέρας της σύνθεσης το Vivado HLS παράγει αναφορά των αποτελεσμάτων. Η αναφορά αυτή περιλαμβάνει τα παρακάτω:

- πόροι υλικού με βάση τους διαθέσιμους της FPGA, πιο συγκεκριμένα lookup tables (LUTs), καταχωρητές, block RAMs και DSP48s
- αριθμός κύκλων ρολογιού που απαιτείται για να υπολογιστούν όλες οι τιμές εξόδου
- αριθμός κύκλων ρολογιού που απαιτούνται μέχρι να μπορέσει η συνάρτηση να δεχτεί νέες τιμές εισόδου
- αριθμός κύκλων ρολογιού που απαιτούνται για να ολοκληρωθεί μια επανάληψη του βρόχου
- αριθμός κύκλων ρολογιού που απαιτούνται για να ξεκινήσει η νέα επανάληψη να επεξεργάζεται δεδομένα
- αριθμός κύκλων ρολογιού που απαιτούνται για να εκτελεστούν όλες οι επαναλήψεις

Οι τιμές αυτές έχουν σκοπό την αξιολόγηση της απόδοσης από τον χρήστη για να εφαρμοστούν αργότερα οι κατάλληλες οδηγίες βελτιστοποίησης ως προς την τροποποίηση και έλεγχο της συμπεριφοράς της εσωτερικής λογικής και των θυρών εισόδου/εξόδου.

Ένα ακόμα πλεονέκτημα του Vivado HLS είναι η δυνατότητα ελέγχου της C σύνθεσης μέσω οδηγιών (directives) βελτιστοποίησης, δημιουργώντας έτσι συγκεκριμένες υλοποιήσεις υλικού υψηλού επιπέδου. Μέσω αυτού του πλεονεκτήματος στοχεύει η παρούσα διπλωματική να βελτιστοποιήσει την απόδοση εφαρμογών.

Όλα τα προαναφερθέντα πλεονεκτήματα του Vivado HLS συντέλεσαν στο να επιλεγεί έναντι άλλων προγραμμάτων για τη σύνθεση των συναρτήσεων και την εκτέλεσή τους σε υλικό.

3.4 Μοντελοποίηση

Βασικό στοιχείο της διπλωματικής είναι το εργαλείο να αναγνωρίζει ποιο κομμάτι κώδικα θα εκτελεστεί σε υλικό και στη συνέχεια με την ανάγνωσή του να παράγει τις δύο συναρτήσεις που είναι απαραίτητες για αυτή την εκτέλεση.

Ουσιαστικά πρόκειται για επεξεργασία, ανάγνωση και παραγωγή κώδικα C, άρα ήταν ξεκάθαρο πως αυτές οι λειτουργίες θα πραγματοποιούνταν μέσω ενός λεκτικού και συντακτικού αναλυτή της γλώσσας C υλοποιημένο σε προγράμματα που παράγουν

τέτοιου είδους αναλυτές. Η υλοποίηση ενός λεκτικού και συντακτικού αναλυτή της γλώσσας C είναι ένα τετριμμένο πρόβλημα και όχι το βασικό της διπλωματικής, άρα αναζητήθηκε έτοιμος λεκτικός και συντακτικός αναλυτής στο διαδίκτυο για να αποτελέσει τη βάση του εργαλείου. Μετά από αναζήτηση και δοκιμές βρέθηκε ένας λειτουργικός και εύκολος στη χρήση και επεξεργασία λεκτικός και συντακτικός αναλυτής.

Το πρώτο πρόβλημα που παρουσιάστηκε ήταν το πώς θα αναγνωρίζει το εργαλείο ποιο κομμάτι κώδικα έχει επιλεγεί από το χρήστη για το σκοπό αυτό. Επειδή πρόκειται για οδηγία, ο κώδικας προς επεξεργασία αναγνωρίζεται με τη λέξη-κλειδί DIRECTIVES. Καθώς στη γλώσσα C κώδικες εντός βρόχων επανάληψης, συνθηκών και άλλων τοποθετούνται μέσα σε αγκύλες για να αναγνωρίζεται εύκολα το πού ξεκινάνε και πού τελειώνουν θεωρήθηκε πως αυτή είναι και μια εύκολη και ταιριαστή στη γλώσσα C λύση στο πρόβλημα. Συνεπώς, ο χρήστης εντοπίζει στον κώδικα εισόδου ποιο κομμάτι θέλει να εκτελέσει σε υλικό, το περικλείει μέσα σε αγκύλες και πάνω από το άνοιγμα αγκύλης, γράφει τη λέξη-κλειδί DIRECTIVES.

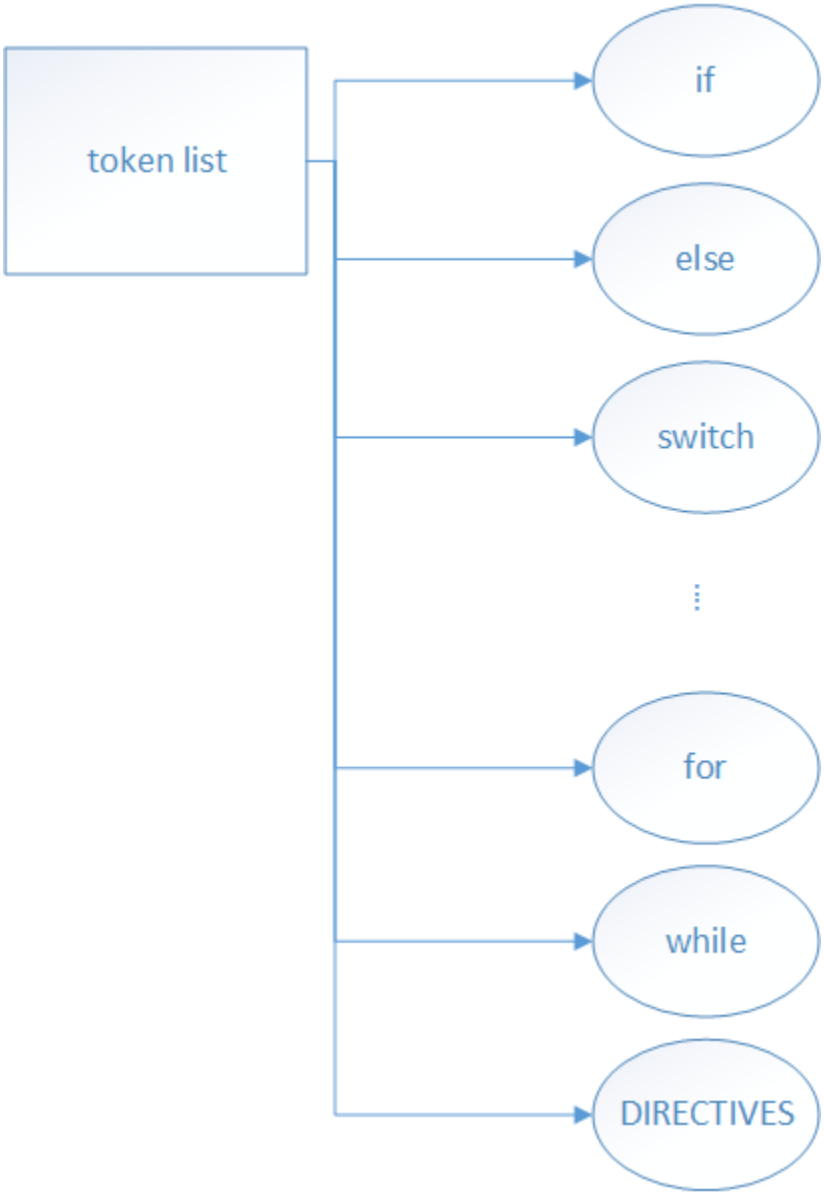
Στη συνέχεια, έπρεπε να αποφασιστεί με ποιο τρόπο θα γίνεται η ανάγνωση και λήψη των απαραίτητων πληροφοριών από τον πηγαίο κώδικα για την παραγωγή των δύο συναρτήσεων. Η πρώτη σκέψη ήταν στον κανόνα κάθε παράστασης που εμφανίζεται εντός της οδηγίας, για παράδειγμα κάποια πράξη να υπάρχει κώδικας που να επεξεργάζεται και να παίρνει τα απαραίτητα δεδομένα. Όμως, κατά τη διαδικασία υλοποίησης αυτής της σκέψης, παρατηρήθηκε πως υπάρχουν προβλήματα. Ο κανόνας της πρόσθεσης είναι της μορφής «exp ‘+’ exp», οπότε διαβάζει πράξεις με ένα τελεστή. Για παράδειγμα σε μια πρόσθεση τριών μεταβλητών, της μορφής x + y + z θα υπήρχε πρόβλημα, καθώς θα διάβαζε δύο πράξεις. Πρώτα την ‘x’ + ‘y’ και μετά την ‘x + y’ + ‘z’. Έτσι, θα ήταν δύσκολο να ξεχωριστεί ποιες εκφράσεις πράξεων βρίσκονται εντός της οδηγίας για να παραχθούν σωστά οι συναρτήσεις. Επίσης, αν ένας βρόχος επανάληψης εντός της οδηγίας είχε ως τερματικό στοιχείο μια έκφραση της μορφής length - 1, το εργαλείο δεν μπορεί να αναγνωρίσει ότι η αφαίρεση αυτή βρίσκεται μέσα σε βρόχο και θα συμπεριλαμβανόταν στις πράξεις που παράγει το εργαλείο. Οπότε η σκέψη αυτή εγκαταλείφθηκε και προτιμήθηκε οι πράξεις εντός της οδηγίας να διαβάζονται χαρακτήρα-χαρακτήρα σε άλλο σημείο του τροποποιημένου λεκτικού και συντακτικού αναλυτή.

Ένα άλλο πρόβλημα που παρουσιάστηκε, το οποίο ήταν άμεσα συσχετιζόμενο με τα προβλήματα που προαναφέρθηκαν, ήταν το πώς θα ξεχωρίζει το εργαλείο ποιες ακριβώς πράξεις βρίσκονται εντός της οδηγίας. Ο λεκτικός και συντακτικός αναλυτής δεν έχει τρόπο κατά την ανάγνωση ενός στοιχείου να γνωρίζει τα παραπάνω. Οι εφαρμογές με την οδηγία ενδέχεται να εμπεριέχουν και άλλα στοιχεία, όπως πράξεις και πριν και μετά την οδηγία. Με την υλοποίηση που αναφέρθηκε στην παραπάνω παράγραφο, αν ο λεκτικός και συντακτικός αναλυτής διαβάσει σε ένα σημείο μια πράξη ή ένα βρόχο επανάληψης, δεν υπάρχει τρόπος στο σημείο αυτό να ξέρει αν βρίσκονται εντός ή όχι της οδηγίας. Οπότε, μετά από πολύ σκέψη και αποτυχημένες

δοκιμές, όπως το να τοποθετηθεί η ανάγνωση και επεξεργασία την περιγραφή του λεκτικού αναλυτή μετά την εμφάνιση του συμβόλου (token) DIRECTIVES και την εγγραφή της οδηγίας σε νέο αρχείο και εκ νέου ανάγνωση του αρχείου αυτού από τον συντακτικό αναλυτή, αποφασίστηκε να κρατώνται διάφορα στοιχεία σε κανόνες και όπως προαναφέρθηκε οι πράξεις να διαβάζονται και να επεξεργάζονται χαρακτήρα-χαρακτήρα. Η καλύτερη λύση εκτιμήθηκε πως ήταν αυτή η ανάγνωση να υλοποιείται εντός νέου κανόνα στην περιγραφή του συντακτικού αναλυτή που θα δημιουργηθεί για να δέχεται ο λεκτικός και συντακτικός αναλυτής τη σύνταξη της οδηγίας, που σημαίνει ότι αν δεν εμφανιστεί ο κανόνας αυτός δεν γίνεται καμία ανάλυση πράξεων. Με τον τρόπο αυτό, επεξεργάζονται από το εργαλείο μόνο πράξεις που βρίσκονται εντός της οδηγίας. Περισσότερες λεπτομέρειες όσον αφορά την υλοποίηση βρίσκονται σε παρακάτω κεφάλαιο.

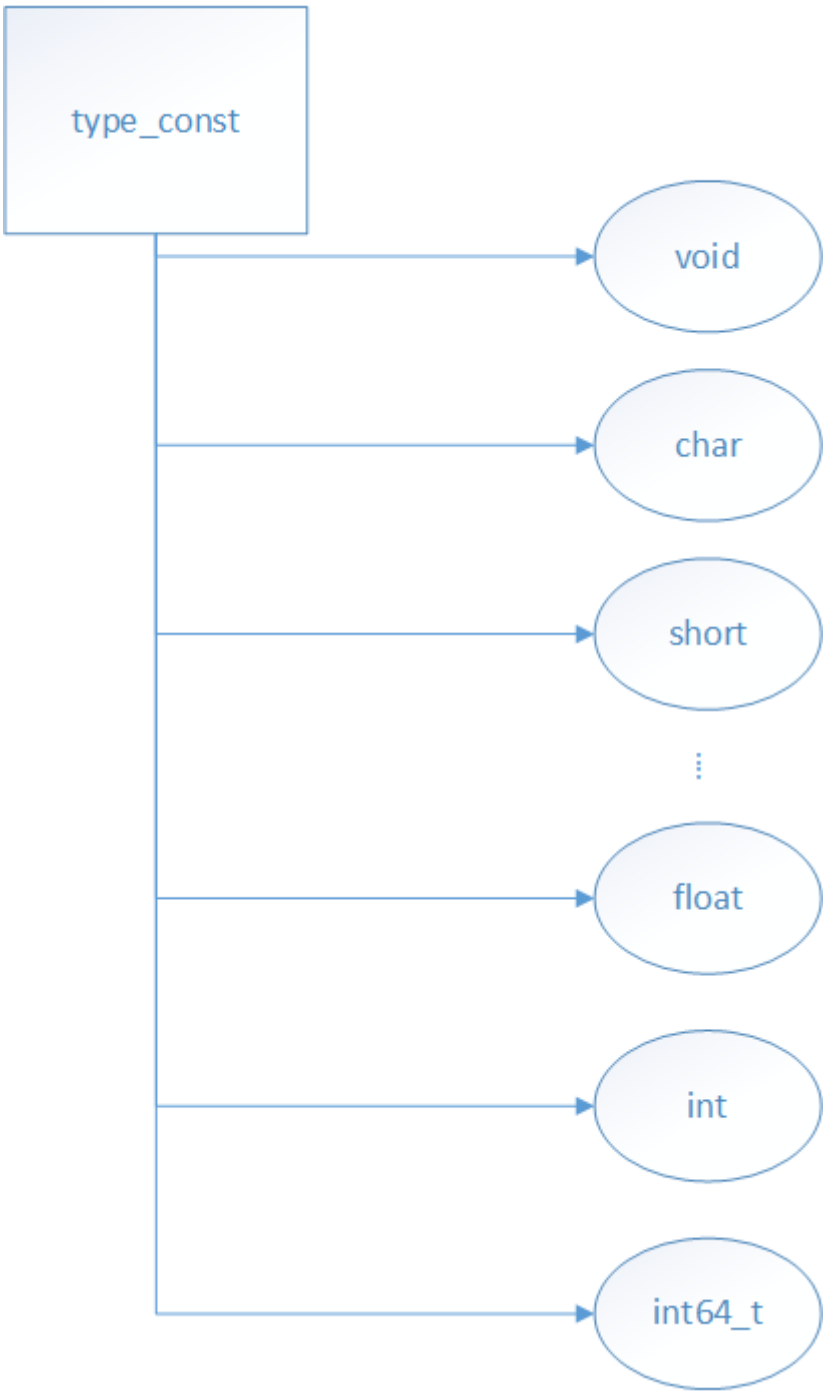
Αυτά ήταν τα βασικότερα προβλήματα που προέκυψαν κατά τη διαδικασία παραγωγής του εργαλείου, όποιο ζήτημα προέκυψε από εκεί και μετά ήταν μικρότερης σημασίας και αφορούσε τον τρόπο με τον οποίο θα γίνεται η ανάγνωση και επεξεργασία των δεδομένων εντός των κανόνων του τροποποιημένου λεκτικού και συντακτικού αναλυτή.

Πρώτη από τις τροποποιήσεις που πραγματοποιήθηκαν στην περιγραφή του λεκτικού αναλυτή ήταν η τοποθέτηση της εντολής yylval σε κάθε σύμβολο (token) με σκοπό τη δυνατότητα ανάγνωσής τους. Κατόπιν, στη λίστα των συμβόλων (tokens) τοποθετήθηκε η λέξη «DIRECTIVES» έτσι ώστε να αναγνωρίζεται ως λέξη-κλειδί από τον αναλυτή, όπως οι if, while, and κ.ο.κ.



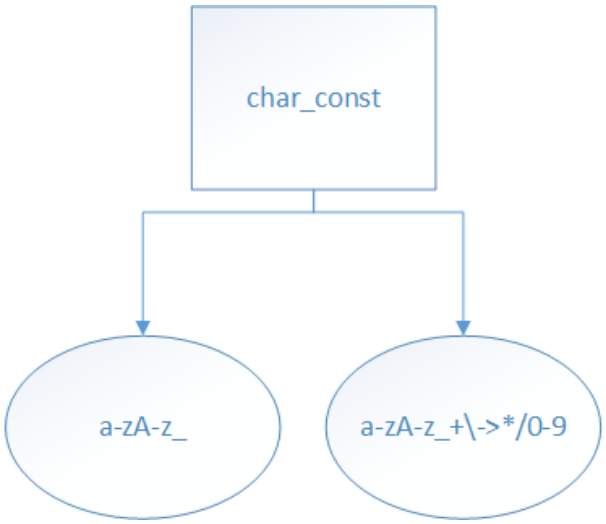
Εικόνα 22: Λίστα συμβόλων (tokens) του τροποποιημένου λεκτικού αναλυτή

Οι συναρτήσεις που παράγει το εργαλείο προορίζονται για εκτέλεση σε λογισμικό σχεδίασης υλικού υψηλού επιπέδου. Για το λόγο αυτό είναι απαραίτητο οι ακέραιες μεταβλητές να έχουν συγκεκριμένο μήκος. Για τον λόγο αυτό, για να καλυφθούν οι ανάγκες των εφαρμογών, προστέθηκε στη λίστα με τους τύπους μεταβλητών, δηλαδή στο σύμβολο (token) type_const ένας επιπλέον τύπος μεταβλητών υλικού, ο int64_t που προορίζεται για ακεραίους μεγέθους 64 bits.



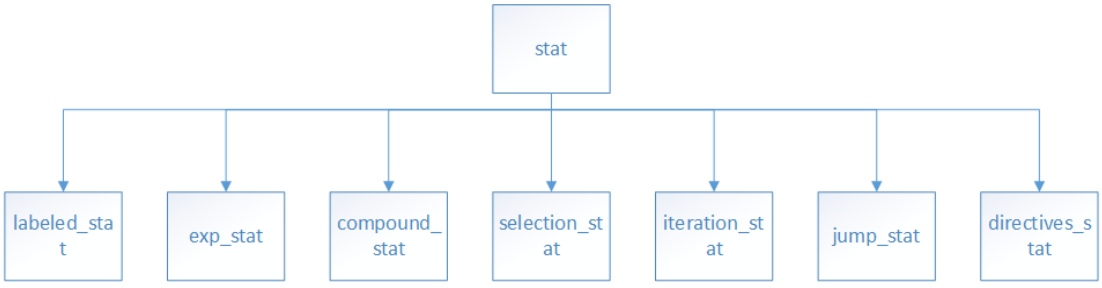
Εικόνα 23: Απεικόνιση τύπων μεταβλητών που αναγνωρίζονται ως σύμβολο (token) type_const

Επίσης, είναι απαραίτητο να υπάρχουν εντός των μεταβλητών τελεστές πράξης για να αναγνωρίζονται από τον αναλυτή στοιχεία πινάκων όπως το a[i+1]. Για τον λόγο αυτό, προστέθηκαν στη δήλωση μεταβλητών, στο σύμβολο (token) char_const, και οι τελεστές πράξεων, όπως και ο χαρακτήρας «>» που χρησιμοποιείται σε λίστες. Με τον τρόπο αυτό, αναγνωρίζονται ως μεταβλητή κάποια από τις εκφράσεις χαρακτήρων, αριθμών ή των τελεστών που προστέθηκαν.



Εικόνα 24: Απεικόνιση εκφράσεων του συμβόλου (token) char_const

Όπως στην περιγραφή του λεκτικού αναλυτή, έτσι και στην περιγραφή του συντακτικού έλειπε η δυνατότητα ανάγνωσης των κανόνων, για αυτό προστέθηκε σε κάθε κανόνα η εντολή strdup, η οποία επιστρέφει δείκτη συμβολοσειράς (string). Στη συνέχεια, με τη χρήση της εντολής union και τη δήλωση όλων των κανόνων και συμβόλων (tokens) ως stringValue, επιτεύχθηκε η σύνδεση των δύο αναλυτών και η μεταφορά των συμβόλων (tokens) από τον λεκτικό στον συντακτικό αναλυτή. Κατόπιν, δημιουργήθηκε και εντάχθηκε στο συντακτικό της γλώσσας ένας νέος κανόνας με σκοπό την ανάγνωση και επεξεργασία του κατάλληλου κώδικα παραγωγή των αρχείων των συναρτήσεων fetch και process μόλις εμφανιστεί στον κώδικα εισόδου η οδηγία DIRECTIVES, αλλά και κώδικας με επιπλέον λειτουργικότητα στους κανόνες αρχικοποίησης και βρόχων επανάληψη για τη λήψη απαραίτητων για την παραγωγή των συναρτήσεων πληροφοριών. Η οδηγία directives μπορεί να τοποθετηθεί σε οποιοδήποτε σημείο εντός της main και για το λόγο αυτό τοποθετήθηκε εντός του κατάλληλου κανόνα. Ο κανόνας αυτός υποδηλώνει ότι η οδηγία αυτή μπορεί να εμφανιστεί στο ίδιο σημείο με μια εκτέλεση πράξης ή έναν βρόχο επανάληψης.



Εικόνα 25: Ένταξη του νέου κανόνα directives στο συντακτικό της γλώσσας

ΚΕΦΑΛΑΙΟ 4-ΠΕΡΙΓΡΑΦΗ ΥΛΟΠΟΙΗΣΗΣ

4.1 Λεκτικός και συντακτικός αναλυτής

4.1.1 Τροποποιήσεις στον έτοιμο λεκτικό και συντακτικό αναλυτή

Το πρώτο κομμάτι του εργαλείου είναι ένας λεκτικός και συντακτικός αναλυτής. Ο αναλυτής αυτός βασίστηκε σε έναν έτοιμο της γλώσσας C και τροποποιήθηκε για να εξυπηρετήσει τις ανάγκες της διπλωματικής. Η είσοδος του λεκτικού και συντακτικού αναλυτή είναι ένα αρχείο κώδικα C. Ο κώδικας όμως αυτός περιέχει οδηγία η οποία δεν εμπεριέχεται στους κανόνες της γλώσσας C. Για τον λόγο αυτό ήταν απαραίτητη η τροποποίηση του λεκτικού και συντακτικού αναλυτή.

Ο παραπάνω αναλυτής απλώς επιβεβαίωνε αν το πρόγραμμα εισόδου είναι λεκτικά και συντακτικά ορθό ή όχι. Μια από τις ανάγκες της διπλωματικής είναι η ανάγνωση και επεξεργασία μεταβλητών, πράγμα αδύνατο στην αρχική μορφή του αναλυτή. Αρχικά τροποποιήθηκε ο λεκτικός αναλυτής. Σε κάθε σύμβολο (token) προστέθηκε η εντολή `yylval.stringValue = strdup(yytext);` Η οποία επιστρέφει την τιμή του ως συμβολοσειρά (string). Έπειτα, στον συντακτικό αναλυτή, όλα τα σύμβολα (tokens) δηλώθηκαν ως `%token <stringValue>`, όπως και στον λεκτικό αναλυτή. Η καθολική μεταβλητή yylval δηλώνεται στο αρχείο y.tab.h και λαμβάνει την τιμή του αντίστοιχου συμβόλου από τη συνάρτηση yytext. Η yytext καλείται από τον λεκτικό αναλυτή για να λάβει το επόμενο σύμβολο (token) [24]. Επίσης, προστέθηκε η δομή union, η οποία συνδέει τους δύο αναλυτές, με σκοπό τη μεταφορά των σύμβολα (tokens) από τον λεκτικό στον συντακτικό. Ακόμη, δηλώθηκαν ως %type <stringValue> όλοι οι κανόνες της γλώσσας, δηλαδή ως συμβολοσειρές (strings), όπως και τα σύμβολα (tokens).

Επιπλέον, πειραματικά παρατηρήθηκε ότι οι κανόνες του αρχικού λεκτικού αναλυτή ήταν ελλιπείς. Ενώ αναγνώριζε σαν μεταβλητή στοιχείο ενός πίνακα στη θέση μιας μεταβλητής, για παράδειγμα a[i], δεν αναγνώριζε σαν μεταβλητή στοιχείο ενός πίνακα στη θέση μιας μεταβλητής αλλαγμένης με κάποια πράξη, για παράδειγμα a[i+1]. Για να αλλάξει αυτό, τροποποιήθηκε η περιγραφή του λεκτικού αναλυτή, συγκεκριμένα στον κανόνα που αφορά τα ονόματα των μεταβλητών. Ο αρχικός κώδικας υποστήριζε την ύπαρξη χαρακτήρων και αριθμών μέσα στις αγκύλες και μετά τα ονόματα. Οπότε, για το τι μπορεί να ακολουθεί το όνομα του πίνακα, οι τέσσερις τελεστές βασικών πράξεων, οι +, -, * και /. Ο τελεστής της αφαίρεσης όμως, έχει και άλλη χρήση στον lex, συμβολίζει όλους τους χαρακτήρες μέσα σε ένα διάστημα. Οπότε, για να αναγνωριστεί ως χαρακτήρας από τον λεκτικό και συντακτικό αναλυτή, έπρεπε στον κανόνα να απεικονισθεί ως “/-“. Στον λεκτικό αναλυτή, πίνακες και μεταβλητές δηλώνονται με τον ίδιο κανόνα. Η προσθήκη των τελεστών στον κανόνα αυτό, έκανε απαραίτητη την προσθήκη και άλλων χαρακτήρων

στον κανόνα αυτό. Μετά από αυτήν την προσθήκη όμως, ο συντακτικός αναλυτής δεν αναγνώριζε πλέον τον τελεστή βέλος (->). Το πρόβλημα αυτό διορθώθηκε με την προσθήκη του χαρακτήρα > στον κανόνα.

Στη συνέχεια, στον κανόνα που περιγράφονται οι τύποι των μεταβλητών προστέθηκε ο τύπος int64_t για τη δήλωση ακεραίων μεγέθους 64 bits. Με τον τρόπο αυτόν υπάρχει η δυνατότητα προσθήκης τύπων μεταβλητών διαφορετικών μεγεθών σε περίπτωση που το απαιτούν οι ανάγκες άλλων εφαρμογών.

Προκειμένου να δουλέψει ο λεκτικός και συντακτικός αναλυτής που βρέθηκε, έπρεπε να γίνει άλλη μια τροποποίηση, αυτή τη φορά στο αρχείο .y του yacc. Σε κάθε κανόνα της γραμματικής, προστέθηκε ένα συγκεκριμένο κομμάτι κώδικα. Δουλειά του κομματιού αυτού, είναι να επιστρέφει τη γραμμή που διαβάζει κάθε φορά ο λεκτικός και συντακτικός αναλυτής. Για την επίτευξη αυτού, χρησιμοποιείται αρχικά η μεταβλητή str τύπου συμβολοσειράς (string) με μέγεθος 10.000. Το μέγεθος αυτό θεωρήθηκε υπέρ αρκετό για να χωρέσει τους χαρακτήρες μιας γραμμής κώδικα. Πρώτη γραμμή του κώδικα κάθε κανόνα είναι η «`char str[10000] = "";`». Δηλώνεται σε κάθε κανόνα έτσι, διότι αυτός είναι ένας απλός τρόπος να αδειάζει η συμβολοσειρά (string) στην αρχή κάθε κανόνα. Κατόπιν, με την χρήση της εντολής 46 print, γεμίζει η συμβολοσειρά (string) με τους χαρακτήρες της γραμμής που διαβάστηκε. Τέλος, με τη χρήση της εντολής strdup αντιγράφεται στη μεταβλητή $$ η συμβολοσειρά (string) str. Με τον τρόπο αυτό, επιστρέφεται η γραμμή που διαβάστηκε στα πιο πάνω μέρη του κανόνα ως συμβολοσειρά (string), διαδικασία απαραίτητη για να επιτευχθεί η εκτύπωση και επεξεργασία κομματιών του κώδικα εισόδου σε C.

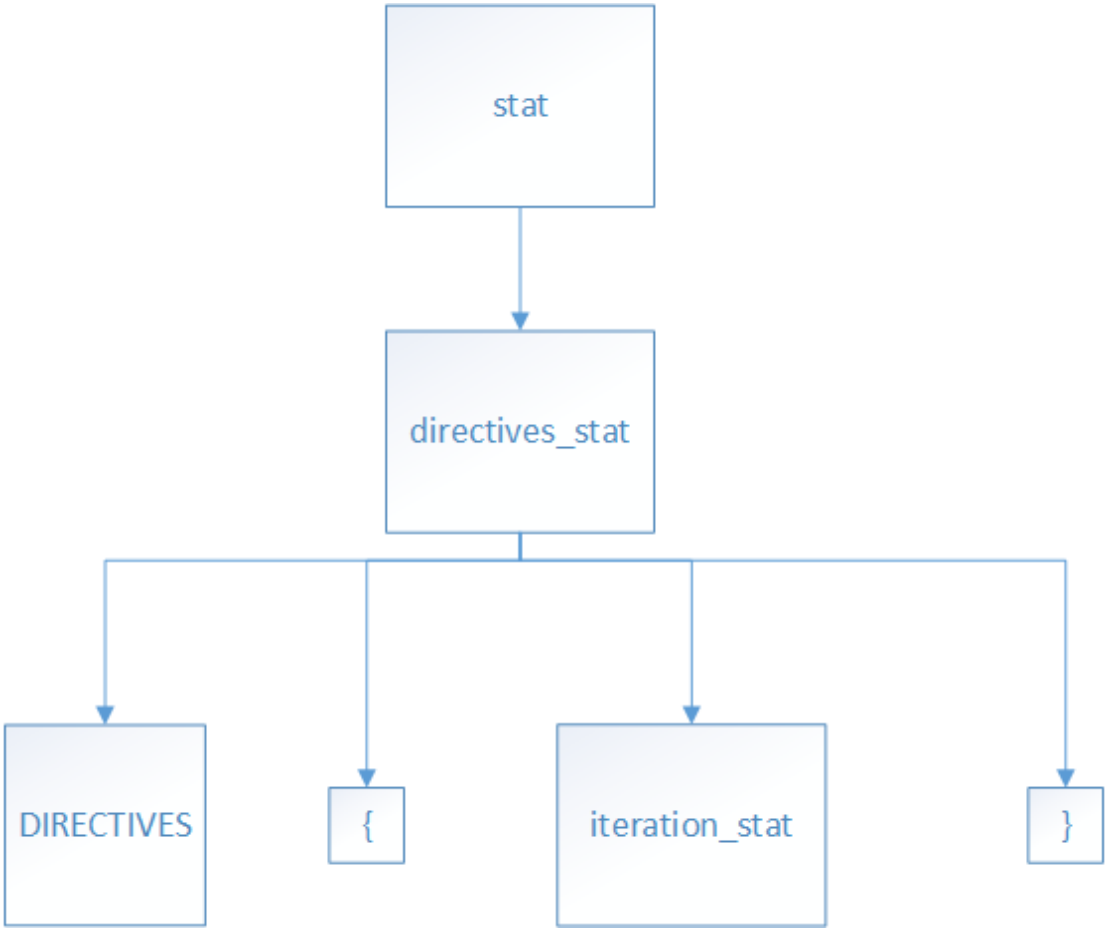
Η επεξεργασία του λεκτικού και συντακτικού αναλυτή έχει ως στόχο τη αναγνώριση της γλώσσας C και ενός επιπλέον κανόνα. Ο κανόνας αυτός έχει τη μορφή:

```
DIRECTIVES
{
      ……
}
```

Εικόνα 26: Μορφή κανόνα

Μέσα στις αγκύλες εμπεριέχεται ένας βρόχος επανάληψης for, απλός ή με εμφωλευμένους. Για να αναγνωριστεί από τον λεκτικό και συντακτικό αναλυτή η οδηγία αυτή ήταν απαραίτητο να προστεθεί στον λεκτικό αναλυτή ως σύμβολο (token) η λέξη κλειδί DIRECTIVES και η προσθήκη ενός κανόνα στον συντακτικό αναλυτή. Ο κανόνας αυτός αναγνωρίζει τη λέξη-κλειδί DIRECTIVES, τις αγκύλες και έναν βρόχο επανάληψης απλό ή με εμφωλευμένους. Μόλις εντοπιστεί η οδηγία αυτή, εκτελείται συγκεκριμένος κώδικας, ο οποίος σε συνδυασμό με κώδικα που εκτελείται στις οδηγίες αρχικοποίησης και των βρόχων επανάληψης, παράγει τις επιθυμητές συναρτήσεις.

Η σύνταξη αυτή δηλώνεται με τον κανόνα DIRECTIVES_TK '{' iteration_stat '}'. Το περιεχόμενο του iteration_stat, δηλαδή ο βρόχος επανάληψης, ο οποίος είναι απλός ή με εμφωλευμένους, βρίσκεται στην Τρίτη θέση του κανόνα, και κατά συνέπεια στο σύμβολο (token) $3. Το περιεχόμενο του συμβόλου αποθηκεύεται στη συμβολοσειρά (string) stat για περαιτέρω επεξεργασία. Η δήλωση του κανόνα αυτού, όπως ορίζουν οι κανόνες του yacc, πρέπει να τοποθετηθεί σε έναν ήδη υπάρχοντα κανόνα της γραμματικής της γλώσσας και δεν μπορεί να στηριχθεί μόνος του. Για να μην εξαρτάται η ύπαρξη του κανόνα directives από άλλα στοιχεία, όπως βρόχοι επανάληψης και συνθήκες, τα οποία δεν είναι απαραίτητα για να λειτουργήσει ένα πρόγραμμα C, και να μπορεί να βρίσκεται σε οποιοδήποτε σημείο του κώδικα εισόδου που να έχει νόημα, ο κανόνας αυτός τοποθετήθηκε σχετικά υψηλά στο συντακτικό δέντρο, και συγκεκριμένα στον κανόνα stat. Ο κανόνας stat εμπεριέχει μεταξύ άλλων εντολές επαναλήψεων, επιλογών και πράξεων, πράγμα που σημαίνει πως η λέξη-κλειδί directives μπορεί να τοποθετηθεί σε οποιοδήποτε σημείο που θα τοποθετούνταν και ένα από τα προαναφερθέντα στοιχεία. Στο παρακάτω σχεδιάγραμμα απεικονίζεται τμήμα του συντακτικού δέντρου για τον κανόνα που προστέθηκε. Με ορθογώνια απεικονίζονται άλλοι κανόνες της γλώσσας (stat, directives_stat, iteration_stat) και με τετράγωνα τα σύμβολα (tokens), δηλαδή η λέξη-κλειδί DIRECTIVES και οι χαρακτήρες « { » και « } ».



Εικόνα 27: Συντακτικό δέντρο του νέου κανόνα

Στον κανόνα αρχικοποίησης αποθηκεύονται στους πίνακες totalType και totalVars οι τύποι των μεταβλητών του προγράμματος και τα ονόματα των μεταβλητών. Ο μόνος περιορισμός για να δουλέψει σωστά είναι κάθε μεταβλητή να δηλώνεται στην αρχή του προγράμματος ξεχωριστά χωρίς κάποια αρχικοποίηση. Οι μεταβλητές δηλώνονται εκεί μαζί με τους τύπους τους, συνεπώς αυτός ήταν ο μοναδικός τρόπος να ληφθεί η πληροφορία για τον τύπο κάθε μεταβλητής.

Στον κανόνα των βρόχων επανάληψης της μορφής «FOR '(' exp ';' exp ';' exp ')' stat» κρατάμε στον πίνακα από συμβολοσειρές (strings) fin τους βρόχους επανάληψης για εκτυπωθούν αργότερα. Για την παραγωγή των συναρτήσεων είναι απαραίτητο να γνωρίζουμε τη μεταβλητή της επανάληψης, καθώς και την μεταβλητή τερματισμού. Η μεταβλητή αυτή λαμβάνεται από την πρώτη έκφραση, που είναι της μορφής `i = x`. Η έκφραση αυτή βρίσκεται στο σύμβολο (token) $3, οπότε το περιεχόμενο του $3 μεταφέρεται σε προσωρινή μεταβλητή, της οποίας το πρώτο στοιχείο είναι η μεταβλητή της επανάληψης και αποθηκεύεται στον πίνακα χαρακτήρων vari. Η μεταβλητή τερματισμού λαμβάνεται από τη συνθήκη τερματισμού του βρόχου, που είναι για παράδειγμα της μορφής `i <= xmax`, `j > cols - 1` ή `k <= 2`. Η έκφραση αυτή βρίσκεται στο σύμβολο (token) $5, οπότε το περιεχόμενο του $5 μεταφέρεται σε προσωρινή μεταβλητή. Για ευκολία στην ανάγνωσή του, τοποθετείται στο τέλος της προσωρινής μεταβλητής ο χαρακτήρας ; και τότε ξεκινάει η ανάγνωση της προσωρινής αυτής μεταβλητής χαρακτήρα-χαρακτήρα. Η ανάγνωση αυτή έχει δύο χρήσεις. Αρχικά, όπως προαναφέρθηκε κρατάει τη μεταβλητή τερματισμού. Οι μεταβλητές αυτές αποθηκεύονται στον πίνακα συμβολοσειρών (strings) fin. Μόλις λοιπόν συναντήσει κάποιον τελεστή ανισότητας ακολουθούμενο από τον κενό χαρακτήρα, ξεκινάει την εγγραφή, από τον χαρακτήρα που ακολουθεί τον κενό. Αποθηκεύει στον πίνακα συμβολοσειρών (strings) fin τους χαρακτήρες που διαβάζει μέχρι να συναντήσει κάποιο τερματικό στοιχείο της συνθήκης. Τα τερματικά αυτά στοιχεία είναι κάποιος τελεστής πράξης και το ερωτηματικό. Η δεύτερη χρήση της ανάγνωσης αυτής, είναι η εξέταση της πιθανότητας το τερματικό στοιχείο του βρόχου επανάληψης να είναι κάποιος πίνακας. Στην περίπτωση αυτή, απαιτείται επιπλέον αρχείο fetch, για να φέρνει τα στοιχεία του πίνακα αυτού. Επιπλέον ενέργειες για το γεγονός αυτό εκτελούνται αργότερα. Στο σημείο αυτό, στον πίνακα fin αποθηκεύεται για τερματικό στοιχείο η λέξη «temp». Τα υπόλοιπα στοιχεία του πίνακα παραμένουν ανεπηρέαστα. Τέλος, ο συνολικός αριθμός επαναλήψεων κρατείται στην ακέραια μεταβλητή numOfIters και ο βρόχος επανάληψης κρατείται στον πίνακα συμβολοσειρών (strings) iter. Η διαδικασία αυτή υλοποιείται από τον παρακάτω κώδικα:

```
sprintf(iter[numOfIters], "for(%s; %s; %s)\n", $3, $5,
$7); //copying each for loop in a string table
strcpy(temp, $3); //token of the loop initialization
int tempCount=0;
```

```
vari[numOfIters]=temp[0]; //first element of the token is
//the loop variable
strcpy(temp, "");
strcpy(temp, $5); //token of the condition that ends the
//loop
strncat(temp, ";", 1); //copy a colon to the end of the
//string as a termination point
//the following loop searches the token of the loop
//termination to save the length of the table; number or
//variable
for (i=0;i<=20;i++)
{
     if(((temp[i]=='<')||(temp[i]=='>')||(temp[i]=='='))
     &&(temp[i+1]==' ')) //after the comparison operators
     {
          for(j=i+2;j<=20;j++)
          {
               if((temp[j]=='+')||(temp[j]=='-
')||(temp[j]==';')) //terminate writing on an operator or
//the colon
               {
                    i=20;
                    j=20;
               }
               else
               {
                    fin[numOfIters][tempCount]=temp[j];
                    tempCount++;
               }
          }
     }
}
strcpy(temp, "");
numOfIters++;
```

Εικόνα 28: Υπολογισμός και αποθήκευση αριθμού βρόχων επανάληψης, μεταβλητών επανάληψης και τερματικών στοιχείων:

Υπάρχει τρόπος αργότερα, εντός της οδηγίας directives και του αντίστοιχου κανόνα να εντοπιστεί το ποιοι ακριβώς βρόχοι επανάληψης βρίσκονται εντός της οδηγίας στον πηγαίο κώδικα. Οπότε κρίθηκε σωστό οι απαραίτητες πληροφορίες για τους βρόχους επανάληψης να λαμβάνονται από τον κανόνα των βρόχων, διότι οι βρόχοι στο σημείο αυτό είναι σπασμένοι σε σύμβολα (tokens) με αποτέλεσμα να είναι απλούστερη η ανάγνωση και επεξεργασία τους.

4.1.2 Συνάρτηση process

Ο νέος κανόνας που υλοποιήθηκε για την οδηγία, με την ανάγνωση τον κώδικα που περιέχεται μέσα σε αυτήν και τη χρήση των πληροφοριών που κρατούνται στους προηγούμενους κανόνες παράγει τις επιθυμητές συναρτήσεις. Αρχικά κρίθηκε

απαραίτητο να γνωρίζεται πόσοι εμφωλευμένοι βρόχοι επανάληψης βρίσκονται στην οδηγία. Για να επιτευχθεί αυτό, αρχικά καταμετράται πόσες φορές βρίσκονται στη σειρά τα γράμματα f, o και r, σχηματίζοντας τη λέξη-κλειδί for. Οι φορές που θα εντοπισθεί αυτή η αλληλουχία γραμμάτων αποθηκεύεται στη μεταβλητή forCount. Κατόπιν, ξεκινάει εκ νέου να διαβάζει χαρακτήρα-χαρακτήρα τη συνάρτηση και να αποθηκεύει το πόσους χαρακτήρες) βρίσκει στη μεταβλητή parenthesisCount. Κάθε βρόχος επανάληψης τελειώνει με χαρακτήρα) και δεν περιέχει άλλον όμοιο. Μόλις λοιπόν διαβαστούν όλοι οι χαρακτήρες), και ο αριθμός parenthesisCount γίνει ίσος με τον αριθμό forCount σημαίνει ότι έχει ολοκληρωθεί η ανάγνωση των βρόχων επανάληψης και οτιδήποτε ακολουθεί είναι οι πράξεις προς επεξεργασία. Αυτή η διαδικασία ήταν απαραίτητη για να γνωρίζει ο λεκτικός και συντακτικός αναλυτής πού ακριβώς ξεκινούν οι απαραίτητες πράξεις και να αγνοεί όσα στοιχεία βρίσκονται εντός των εντολών των βρόχων. Τα στοιχεία αυτά στο σημείο αυτό δεν είναι πλέον χρήσιμα, καθώς όλες οι απαραίτητες πληροφορίες έχουν ήδη αποθηκευτεί στον κανόνα των βρόχων επανάληψης.

Μόλις διαβαστεί ο απαραίτητος αριθμός παρενθέσεων ξεκινάει η παραγωγή των πράξεων για τη συνάρτηση process με την ανάγνωση κάθε χαρακτήρα από εκείνο το σημείο και μετά. Οι πράξεις αποθηκεύονται σε ένα πίνακα από συμβολοσειρές (strings) με όνομα operation. Για την αποθήκευση των συμβολοσειρών (strings) χρησιμοποιείται η εντολή `strncat(char * destination, const char * source, size_t num);` η οποία προσθέτει στο τέλος της συμβολοσειράς (string) destination, τους χαρακτήρες της συμβολοσειράς (string) source και αυξάνει το μήκος της κατά τον αριθμό num. Αρχικά, αν διαβαστεί κάποιος από τους τέσσερις τελεστές πράξεων (+, -, *, /) και ακολουθείται από χαρακτήρα διάφορο του =, τότε ο τελεστής αυτός ανάμεσα σε κενούς χαρακτήρες περνάει τον πίνακα operation. Αν όμως ο τελεστής ακολουθείται από τον χαρακτήρα = σημαίνει ότι έχουμε κάποια πράξη προσαύξησης. Οπότε, αντί να αποθηκευθούν στον πίνακα operation οι τελεστές, ανάλογα με τον τελεστή, αποθηκεύεται μια από τις παρακάτω συμβολοσειρές (strings):

- sum = sum +
- dif = dif -
- prod = prod *
- quot = quot /

Στην περίπτωση αυτή, ενεργοποιείται και η αντίστοιχη flag μεταβλητή (flag_sum, flag_diff, flag_prod, flag_quot), η οποία θα χρειαστεί αργότερα. Επιπλέον, αποθηκεύεται στον πίνακα συμβολοσειρών (strings) variableAfter που περιέχει τα νέα ονόματα των μεταβλητών η αντίστοιχη λέξη. Αν από την άλλη ο χαρακτήρας = βρίσκεται ανάμεσα σε κενά, τότε σημαίνει ότι ακολουθεί κάποια άλλη πράξη, οπότε ο χαρακτήρας = και τα κενά που τον περιβάλλουν αποθηκεύονται στον πίνακα operation. Χαρακτήρες που περνάνε αυτούσιοι είναι η τελεία, οι παρενθέσεις και οι αριθμοί. Αυτό όμως δεν συμβαίνει σε περίπτωση που βρεθεί αριθμός αμέσως μετά από χαρακτήρα, γιατί αυτό σημαίνει ότι ο αριθμός αποτελεί μέρος το ονόματος της μεταβλητής και δεν πρέπει να περάσει αυτούσιος στον πίνακα operation.

Οι μεταβλητές αναγνωρίζονται αν μετά από κενό ακολουθεί κάποιος χαρακτήρας, και όχι η λέξη-κλειδί if. Ως τερματικοί για μεταβλητή θεωρούνται οι χαρακτήρες κενό και ;. Μόλις βρεθεί κάποιος από αυτούς ο αριθμός των μεταβλητών αυξάνεται κατά 1, αλλιώς οι χαρακτήρες που αποτελούν το όνομα της μεταβλητής αποθηκεύεται στον πίνακα από συμβολοσειρές (strings) με όνομα variableBefore όπου κρατούνται τα ονόματα των μεταβλητών. Μόλις βρεθούν ένας από τους χαρακτήρες κενό ή ; η εγγραφή τερματίζεται. Κατόπιν, διαβάζονται όλα τα ονόματα μεταβλητών που υπάρχουν στον πίνακα variableBefore. Αν το όνομα της μεταβλητής που μόλις διαβάστηκε βρίσκεται ήδη στον πίνακα, τότε σημαίνει ότι η μεταβλητή αυτή υπήρχε και πριν σε κάποια άλλη πράξη, οπότε τοποθετείται στον πίνακα variableAfter το όνομα που της είχε ανατεθεί τότε, και το όνομα αυτό περνάει και στον πίνακα operation.

```
if((strcmp(variableBefore[variableId-1],
variableBefore[w])==0) && flag_op==0) //if variable name
//is found in a previous element, give the new name
{
     strcpy(var, variableAfter[w]);
     duplicate=0;
     strncat(operation[number], var, 7);
     w=variableId;
}
```

Εικόνα 29: Ανίχνευση και ονοματοδοσία διπλότυπων μεταβλητών

Το όνομα που παίρνουν όσες μεταβλητές δεν είναι κάποιο άθροισμα, διαφορά, γινόμενο ή πηλίκο και περνάει τους πίνακες operation και variableAfter είναι varX, όπου X ο αριθμός που ξεκινάει από το 1 και αυξάνεται κάθε φορά που θα βρεθεί μια διαφορετική των παραπάνω μεταβλητή.

```
if(duplicate==1) //if the variable is new
{
     sprintf(var, "var%d", variables);
     strncat(operation[number], var, 5);
     variables++;
     strcpy(variableAfter[variableId-1], var);
}
```

Εικόνα 30: Ονοματοδοσία νέας μεταβλητής

Η συνάρτηση process επιστρέφει το τελικό αποτέλεσμα της πράξης. Για να εντοπιστεί αυτό, στα σημεία όπου διαβάζεται ο χαρακτήρας «=» και παράγεται μια μεταβλητή variableAfter περιλαμβάνεται η ανάθεση στην μεταβλητή resultVar το όνομα της μεταβλητής που αποθηκεύεται και στον πίνακα variableAfter. Αυτό έχει ως αποτέλεσμα, το αριστερό σκέλος της τελευταίας πράξης, το οποίο είναι και το τελικό αποτέλεσμα που επιστρέφει η συνάρτηση process να είναι αποθηκευμένο και στην μεταβλητή resultVar. Για να γίνει αυτή η αντιστοίχιση όμως, πρέπει η πράξη υλοποιείται με κάποιο απλό τελεστή και όχι με κάποιον από τους σύνθετους +=, -= ,

*= ή /=, διότι για αυτούς η μεταβλητή του πίνακα variableAfter έχει ήδη παραχθεί. Οπότε, με νέο βρόχο επανάληψης, αν ανιχθευθεί κάποιος τελεστής πράξης που δεν ακολουθείται από τον χαρακτήρα «=», ενεργοποιείται η μεταβλητή flag_op. Ο βρόχος τερματίζει μόλις διαβαστεί το ερωτηματικό. Η συνθήκη των αναθέσεων που περιεγράφηκε στην παραπάνω παράγραφο περιλαμβάνει και τη συνθήκη η flag_op να είναι 0, δηλαδή η πράξη δεν υλοποιείται με σύνθετο τελεστή. Η μεταβλητή flag_op μηδενίζεται όταν ολοκληρωθεί ο έλεγχος για τη συμπλήρωση του συγκεκριμένου πεδίου του πίνακα variableAfter. Για παράδειγμα, το κομμάτι αυτό αποφεύγει σε περίπτωση που εντοπιστεί η πράξη, για παράδειγμα, x += 1 να αντιστοιχηθεί στο x στον πίνακα variableAfter κάποιο όνομα που ξεκινάει με var, διότι νωρίτερα στο x θα έχει ήδη ανατεθεί το όνομα sum στον πίνακα variableAfter.

Αν βρεθεί αρχή αγκύλης, δηλαδή ο χαρακτήρας «[», ότι υπάρχει μετά από αυτόν, μέχρι να βρεθεί κλείσιμο αγκύλης, δηλαδή ο χαρακτήρας «]» αγνοείται, καθώς οι αγκύλες και τα περιεχόμενά τους δεν έχουν κάποια χρήση στη συνάρτηση process. Αν βρεθεί ο χαρακτήρας ; σημαίνει ότι τελείωσε κάποια πράξη, άρα περνάει στον πίνακα operation ο χαρακτήρας ; και μετά από αυτόν ο χαρακτήρας αλλαγής γραμμής, και η μεταβλητή number που κρατάει τον αριθμό των πράξεων αυξάνεται κατά 1.

Ο κώδικας που ακολουθεί ανιχνεύει τους τύπους των μεταβλητών. Αρχικά, όλες οι μεταβλητές που βρίσκονται στον πίνακα variableBefore περνάνε σε νέο προσωρινό, τον newBef, αλλά χωρίς τις αγκύλες και τα περιεχόμενά τους για να μην υπάρχουν διαφορές ανάμεσα στα διαφορετικά στοιχεία του ίδιου πίνακα. Οι μεταβλητές του πίνακα newBef αναζητούνται στον πίνακα που δημιουργήθηκε στον κανόνα αρχικοποίησης, τον totalVars, και από εκεί αποθηκεύονται στον πίνακα varType οι τύποι των μεταβλητών. Για τις μεταβλητές όμως κάποιου αθροίσματος, διαφοράς, γινόμενου ή πηλίκου, ο τύπος αποθηκεύεται για ευκολία στις μεταβλητές sumType, difType, prodType και quotType αντίστοιχα.

Με τη διαδικασία αυτή έχουν ληφθεί οι απαραίτητες πληροφορίες και μπορεί να δημιουργηθεί η συνάρτηση process. Όπως έχει προαναφερθεί, ανάλογα με το πόσες φορές εμφανίζονται τα directives στον κώδικα εισόδου, δημιουργείται και ο αντίστοιχος αριθμός αρχείων process. Κατά συνέπεια ήταν απαραίτητο να υλοποιηθεί η δυνατότητα δημιουργίας δυναμικού αριθμού αρχείων. Για να επιτευχθεί αυτό, κρατείται σε μεταβλητή ο αριθμός των αρχείων που είναι απαραίτητο να δημιουργηθούν. Τα αρχεία αυτά έχουν ονόματα processX, όπου X οι αριθμοί 0 έως τον αριθμό των αρχείων που είναι απαραίτητο να δημιουργηθούν μείον ένα. Με βρόχο επανάληψης for, στον κανόνα X παράγονται τα αρχεία. Θεωρήθηκε πως για τις ανάγκες της διπλωματικής δεν θα είναι απαραίτητο να παραχθούν για μια εφαρμογή πάνω από 11 αρχεία process, οπότε δημιουργήθηκε ο πίνακας processName με χωρητικότητα 11 συμβολοσειρές (strings). Οι συμβολοσειρές (strings) αυτές αποτελούν τα ονόματα των αρχείων, process0.cpp, process1.cpp, process2.cpp κ.ο.κ. Κατόπιν, δημιουργείται σε κάθε επανάληψη και

από ένα αρχείο με όνομα αυτό που βρίσκεται στην ανάλογη θέση του πίνακα processName. Ο αριθμός του τρέχοντος αρχείου process βρίσκεται στη μεταβλητή numberOfProcess, η οποία αυξάνεται κατά 1 στην παρούσα φάση, και έτσι το όνομα του αρχείου και στοιχείο του πίνακα processName είναι το process(numberOfProcess-1).cpp. Η διαδικασία αυτή ήταν απαραίτητη, καθώς δεν ήταν δυνατό να δημιουργηθούν αρχεία με μη σταθερό όνομα με μια εντολή.

Πρώτο στοιχείο ενός προγράμματος C είναι η δήλωση βιβλιοθηκών που είναι απαραίτητες για τη λειτουργία των εφαρμογών. Για τον λόγο αυτό, πρώτα εκτυπώνονται στο αρχείο οι δηλώσεις των παρακάτω βιβλιοθηκών:

- <stdio.h>, η οποία περιλαμβάνει τις βασικές συναρτήσεις εισόδου/εξόδου
- <stdlib.h> η οποία περιλαμβάνει βασικές συναρτήσεις χρηστικότητας, όπως τη malloc και τη rand
- <stdint.h>, η οποία περιλαμβάνει διάφορους τύπους ακεραίων
- <time.h>, η οποία περιλαμβάνει συναρτήσεις ημερομηνίας και ώρας
- <string.h>, η οποία περιλαμβάνει συναρτήσεις διαχείρισης συμβολοσειρών (strings)
- <math.h> , η οποία περιλαμβάνει βασικούς μαθηματικούς υπολογισμούς, για παράδειγμα δυνάμεις, λογαρίθμους και ρίζες
- <inttypes.h>, η οποία περιλαμβάνει διάφορους τύπους ακεραίων με συγκεκριμένο μέγεθος

Κατόπιν, εκτυπώνεται η δήλωση της συνάρτησης, με τα ορίσματά της. Για το πρώτο αρχείο, η δήλωση γίνεται με την εκτύπωση της ακολουθίας «`void processUnit0(`». Όπως και στο όνομα του αρχείου, ο αριθμός προκύπτει από την πράξη numberOfProcess-1. Πρώτα ορίσματα είναι οι μεταβλητές των τελεστών των πράξεων οι οποίες λαμβάνονται από τον πηγαίο κώδικα. Κάθε μια από αυτές τις μεταβλητές είναι απαραίτητο να έχει τον ίδιο τύπο με την αντίστοιχή της στον κώδικα εισόδου. Για την πρώτη μεταβλητή, η προσωρινή μεταβλητή tmpVar λαμβάνει την τιμή var1. Ένας βρόχος επανάληψης αντιστοιχίζει την προσωρινή αυτή μεταβλητή, με την αντίστοιχή της στον πίνακα variableAfter με σκοπό να λάβει τον τύπο της μέσω του πίνακα varType. Μόλις συμβεί αυτό εκτυπώνεται στο τελικό αρχείο η ακολουθία «`X *val1`». Κατόπιν, με βρόχο επανάληψης, για i από 2 έως variables-1, που είναι ο συνολικός αριθμός των μεταβλητών, πραγματοποιείται η ίδια αναζήτηση. Μόλις γίνει η ταυτοποίηση εκτυπώνεται στο τελικό αρχείο για κάθε μεταβλητή «`,Y *valX`», όπου Y ο τύπος της μεταβλητής από τον πίνακα varType και X η τιμή i. Η διαδικασία αυτή εκτελείται με τον παρακάτω κώδικα:

```
sprintf(tmpVar, "var1"); //print the first variable
for(j=1; j<=variableId-1; j++)
{
     if(strcmp(variableAfter[j], tmpVar)==0)
     {
```

```
            fprintf(f, "%s *val1", varType[j]);
      }
}

for (i=2;i<=variables-1;i++) //print the rest of the
//variables
{
      sprintf(tmpVar, "var%d", i);
      for(j=1; j<=variableId-1; j++)
      {
            if(strcmp(variableAfter[j], tmpVar)==0)
            {
                  fprintf(f, ", %s *val%d", varType[j], i);
            }
      }
}
```

Εικόνα 31: Μέρος παραγωγής ορισμάτων της συνάρτησης process

Μόλις ολοκληρωθεί η διαδικασία αυτή, η μεταβλητή i έχει πάρει την τιμή variables, και σε περίπτωση που υπάρχει κάποια μεταβλητή από πράξη με σύνθετο τελεστή, ελέγχεται με το αντίστοιχο flag (flag_sum, flag_dif, flag_prod, flag_quot) και για παράδειγμα για το sum εκτυπώνεται στο τελικό αρχείο η ακολουθία «`, Y *valX`», όπου Y το sumType και X η μεταβλητή i. Σε περίπτωση που το τελικό αποτέλεσμα είναι το στοιχείο ενός πίνακα, είναι απαραίτητο να επιστρέφεται από τη συνάρτηση process εκτός από την τιμή, και η διεύθυνσή του. Με βρόχο επανάληψης, αναζητείται το resultVar στον πίνακα variableAfter, με σκοπό να ληφθεί το αντίστοιχο variableBefore. Η τιμή του variableBefore αντιγράφεται στην προσωρινή μεταβλητή tmpVar για ευκολότερη επεξεργασία και διαβάζεται χαρακτήρα-χαρακτήρα. Αν εντοπιστεί κάποια αγκύλη, σημαίνει ότι το αποτέλεσμα είναι στοιχείο πίνακα, οπότε πρέπει να επιστρέφεται και η διεύθυνσή του. Η διεύθυνση είναι πάντα unsigned int, οπότε στα ορίσματα προστίθεται η ακολουθία «`, uint64_t *addr, uint64_t init_address`» και ενεργοποιείται η προσωρινή μεταβλητή addrFlag για να ανιχνευθεί σε παρακάτω σημεία εύκολα το ότι το τελικό αποτέλεσμα είναι στοιχείο πίνακα και χρειάζεται να επιστρέφει η συνάρτηση process και τη διεύθυνσή του.

Τελευταία ορίσματα της συνάρτησης process είναι τα τερματικά στοιχεία των βρόχων επανάληψης. Σε περίπτωση που οι βρόχοι τερματίζουν όχι σε κάποιον συγκεκριμένο αριθμό, αλλά σε μια μεταβλητή, είναι απαραίτητο η μεταβλητή αυτή να περάσει και στη συνάρτηση fetch για να γνωρίζει πόσες φορές θα εκτελεστεί ο συγκεκριμένος βρόχος. Ο τρόπος για να γίνει αυτό είναι να περάσει στη συνάρτηση ως όρισμα. Επίσης, σε περίπτωση που περισσότεροι από ένας βρόχοι έχουν το ίδιο τερματικό στοιχείο, είναι απαραίτητο το εργαλείο να το αναγνωρίζει για να μην υπάρχουν διπλότυπα στα ορίσματα. Όπως έχει περιγραφεί σε παραπάνω ενότητα, τα τερματικά στοιχεία των βρόχων επανάληψης έχουν αποθηκευθεί από τον κανόνα των βρόχων επανάληψης στον πίνακα συμβολοσειρών (strings) fin και οι βρόχοι που βρίσκονται μέσα στην οδηγία είναι αυτοί στις θέσεις numofIters-i, όπου i τιμές από το 1 έως και

το forCount, όπου forCount ο συνολικός αριθμός βρόχων επανάληψης εντός της οδηγίας. Για τους βρόχους αυτούς λοιπόν, με νέο βρόχο επανάληψης διαβάζεται ο πρώτος χαρακτήρας κάθε τερματικού στοιχείου. Αν ο χαρακτήρας είναι κάποιο γράμμα, τότε σημαίνει ότι το τερματικό στοιχείο του βρόχου αυτού είναι μεταβλητή και όχι αριθμός και έτσι είναι απαραίτητη η τοποθέτησή του στα ορίσματα. Στο ενδεχόμενο αυτό, με εμφωλευμένο βρόχο ελέγχεται αν το στοιχείο αυτό υπάρχει προηγουμένως στον πίνακα fin. Αν ναι, τότε σημαίνει ότι η μεταβλητή αυτή έχει εντοπιστεί από πριν και εκτυπώθηκε στα ορίσματα, οπότε ενεργοποιείται η προσωρινή μεταβλητή finFlag. Αν η μεταβλητή finFlag δεν έχει ενεργοποιηθεί, και η τιμή της βρίσκεται στην αρχική 0, τότε σημαίνει ότι η μεταβλητή αυτή δεν έχει εκτυπωθεί στα ορίσματα, οπότε εκτυπώνεται στο αρχείο εξόδου «`, int64_t X`», όπου X η μεταβλητή που εντοπίστηκε στον πίνακα fin. Για τις ανάγκες της διπλωματικής αυτής, θεωρείται πως όλα τα τερματικά στοιχεία ενός βρόχου επανάληψης που δεν είναι αριθμοί θα είναι ακέραιοι, και συγκεκριμένα 64 bits, για αυτό εκτυπώνεται απευθείας στο αρχείο εξόδου ως τύπος της μεταβλητής ο int64_t και δεν πραγματοποιείται σάρωση στις μεταβλητές του πίνακα newBef με σκοπό να εντοπιστεί ο τύπος.

Αφού έχουν εκτυπωθεί όλα τα ορίσματα της συνάρτησης, εκτυπώνεται η ακολουθία «`)\n{`». Κλείνει η παρένθεση της συνάρτησης, κατόπιν ανοίγει σε νέα γραμμή η αγκύλη που θα περιλαμβάνει τον πηγαίο κώδικα της συνάρτησης. Τέλος, εκτυπώνεται μια γραμμή για να ξεκινήσει από εκεί η εκτύπωση του πηγαίου κώδικα για λόγους ομοιομορφίας. Μόλις εκτελεστεί ο παραπάνω κώδικας, για παράδειγμα, για αρχείο εισόδου με μια πράξη ανάμεσα σε 3 στοιχεία πινάκων τύπου double με τελικό αποτέλεσμα στοιχείο πίνακα, και διπλό βρόχο επανάληψης με τερματικά στοιχεία xmax και ymax θα παραχθεί από το εργαλείο ο παρακάτω ορισμός συνάρτησης process:

```
void processUnit0(double *val1, double *val2, double
*val3, int64_t *addr, int64_t init_address, int64_t xmax,
int64_t ymax)
```

Εικόνα 32: Παράδειγμα ορισμού συνάρτησης process

Στο σημείο αυτό, ξεκινάνε και παράγονται τα πρώτα HLS Pragmas, απαραίτητα για την εκτέλεση της συνάρτησης στο Vivado. Πρώτα τοποθετούνται με μια εντολή εκτύπωσης της ακολουθίας «`#pragma HLS CLOCK domain=default\n#pragma HLS DATAFLOW\n`» τα pragmas για το ρολόι και τη ροή δεδομένων (dataflow), καθώς και μια ακόμα νέα γραμμή για να ξεχωρίσουν τα pragmas αυτά από τα επόμενα που αφορούν τα ορίσματα της συνάρτησης. Στη συνέχεια, όπως παράγονται τα pragmas για τα τερματικά στοιχεία των βρόχων επανάληψης. Όπως αναφέρθηκε και παραπάνω τα τερματικά στοιχεία που περνάνε ως ορίσματα στη συνάρτηση είναι μόνο οι μεταβλητές και δεν επιτρέπονται διπλότυπα. Για το λόγο αυτό η παραγωγή των pragmas τους πραγματοποιείται με ακριβώς τον ίδιο κώδικα για την εκτύπωσή τους στα ορίσματα της συνάρτησης με τη διαφορά ότι στο τελικό αρχείο εκτυπώνεται η ακολουθία «`#pragma HLS`

INTERFACE ap_stable port=X\n», όπου X το κατάλληλο στοιχείο του πίνακα fin, καθώς και μια ακόμα γραμμή, για να ξεχωρίζουν οι μεταβλητές, που ακολουθούν αμέσως μετά. Η ιδιαιτερότητα των pragmas που αφορούν τις μεταβλητές είναι ότι υπάρχουν διαφορές στον τύπο της δήλωσης. Λόγω περιορισμών του Vivado HLS, όλες οι μεταβλητές που θα αποθηκευτούν σε FIFO πρέπει να δηλωθούν στα pragmas ως τέτοιες. Οπότε, με απλό βρόχο επανάληψης για i από 1 έως variables-1 εκτυπώνεται το pragma «#pragma HLS INTERFACE ap_fifo depth=4096 port=vali» και μια νέα γραμμή. Σε περίπτωση που υπάρχει μεταβλητή από σύνθετο τελεστή, κάποιο από τα αντίστοιχα flags (flag_sum, flag_dif, flag_prod, flag_quot) θα είναι ενεργοποιημένο και θα εκτυπωθεί ένα ακόμα pragma με την ίδια ακριβώς με παραπάνω εντολή. Το i μετά το πέρας του προηγούμενου βρόχου επανάληψης θα έχει πάρει την τιμή variables, έτσι ώστε με το vali να δημιουργηθεί η μια επιπλέον μεταβλητή που χρειάζεται για το sum, dif, prod ή quot. Κατόπιν, από το addrFlag ελέγχεται αν το τελικό αποτέλεσμα της συνάρτησης είναι στοιχείο πίνακα, στην οποία περίπτωση πρέπει να παραχθούν και τα pragmas για τη διεύθυνση του αποτελέσματος, τα οποία είναι τα παρακάτω:

«#pragma HLS INTERFACE ap_stable port=init_address\n»
«#pragma HLS INTERFACE ap_fifo depth=4096 port=addr\n»

Μόλις εκτελεστεί ο κώδικας αυτός, για το παράδειγμα που περιεγράφηκε παραπάνω, θα παραχθούν τα παρακάτω pragmas:

```
#pragma HLS INTERFACE ap_stable port=xmax
#pragma HLS INTERFACE ap_stable port=ymax
#pragma HLS INTERFACE ap_fifo depth=4096 port=val1
#pragma HLS INTERFACE ap_fifo depth=4096 port=val2
#pragma HLS INTERFACE ap_fifo depth=4096 port=val3
#pragma HLS INTERFACE ap_stable port=init_address
#pragma HLS INTERFACE ap_fifo depth=4096 port=addr
```

Εικόνα 33:Παράδειγμα δήλωσης των process pragmas

Μετά την εκτύπωση των pragmas, η παραγωγή στοιχείων της συνάρτησης συνεχίζεται. Επειδή έχει εκτυπωθεί η αγκύλη που ανοίγει τη συνάρτηση, στο σημείο αυτό ξεκινάνε να παράγονται για με σκοπό την ομοιόμορφη στοίχιση του τελικού αποτελέσματος και τα πρώτα tabs, με την αύξηση της συμβολοσειράς (string) tabs κατά ένα χαρακτήρα «\t» (tab). Κατόπιν, ξεκινάει η παραγωγή των εσωτερικών μεταβλητών της συνάρτησης, με τις μεταβλητές προσαύξησης των βρόχων. Όπως αναφέρθηκε σε παραπάνω κεφάλαιο, οι μεταβλητές αυτές βρίσκονται αποθηκευμένες στον πίνακα vari. Για τις ανάγκες τις διπλωματικής, θεωρείται πως όπως και τα τερματικά στοιχεία των βρόχων επανάληψης, έτσι και οι μεταβλητές προσαύξησης θα είναι τύπου int64_t. Για να εκτυπωθούν όλες οι απαραίτητες μεταβλητές σε μια γραμμή αρχικά εκτυπώνεται η πρώτη, που βρίσκεται στη θέση numOfIters-1 του πίνακα vari με την εκτύπωση στο αρχείο της ακολουθίας «Xint64_t Y», όπου X η συμβολοσειρά (string) tabs και Y η πρώτη μεταβλητή προσαύξησης. Διπλότυπα στις μεταβλητές αυτές δεν είναι δυνατό να υπάρχουν, οπότε με έναν απλό βρόχο επανάληψης για i από 2 έως forCount εκτυπώνονται στο αρχείο οι υπόλοιπες

μεταβλητές με την ακολουθία «, X», όπου X το στοιχείο vari[numOfIters-i]. Η γραμμή αυτή ολοκληρώνεται με την εκτύπωση του απαραίτητου ; καθώς και δύο κενών γραμμών για να ξεχωρίσει από τα επόμενα στοιχεία.

Ακολουθεί η αρχικοποίηση των προσωρινών μεταβλητών. Οι μεταβλητές αυτές παράγονται με τον ίδιο ακριβώς τρόπο με τις μεταβλητές val στα ορίσματα της συνάρτησης. Με τον ίδιο διπλό βρόχο επανάληψης, για όλες τις μεταβλητές εκτυπώνεται η ακολουθία «`XY varZ = 0;\n`», όπου X η συμβολοσειρά (string) tabs, Y ο τύπος της μεταβλητής όπως έχει ληφθεί μέσω του βρόχου από τον πίνακα varType και Z ο αύξων αριθμός της μεταβλητής. Ελέγχεται εκ νέου αν υπάρχει κάποια μεταβλητή από σύνθετο τελεστή. Αν κάποιο από αυτά τα 4 flags είναι ενεργοποιημένο τότε εκτυπώνεται η αντίστοιχη από τις παρακάτω ακολουθίες:
«`XY sum = 0;\n`», όπου X η συμβολοσειρά (string) tabs και Y το sumType
«`XY dif = 0;\n`», όπου X η συμβολοσειρά (string) tabs και Y το difType
«`XY prod = 1;\n`», όπου X η συμβολοσειρά (string) tabs και Y το prodType
«`XY quot = 1;\n`», όπου X η συμβολοσειρά (string) tabs και Y το quotType

Στη συνέχεια εκτυπώνονται οι βρόχοι επανάληψης που βρίσκονται μέσα στην οδηγία μαζί με τα pragma τους. Κάθε βρόχος, εκτός από τον τελευταίο εκτυπώνεται με έναν απλό βρόχο επανάληψης για j από 1 μέχρι forCount-1 με δύο εντολές δύο ακολουθίες. Πρώτα ο βρόχος επανάληψης με την ακολουθία «`XprocessUnitYlabelZ:UT{\n`», όπου X η συμβολοσειρά (string) tabsSize, Y η τιμή numberOfProcess-1, δηλαδή ο αριθμός του αρχείου process το οποίο παράγεται σε αυτή τη φάση, Z η τιμή j-1, U ο βρόχος επανάληψης (που βρίσκεται στη θέση iter[numOfIters-j]) μαζί με το χαρακτήρα του «\n» και T η συμβολοσειρά (string) tabs, και ύστερα το ακόλουθο pragma «#pragma HLS LOOP_MERGE». Στο τέλος κάθε επανάληψης προστίθεται και από ένα tab στη συμβολοσειρά (string) tabs. Με το πέρας του βρόχου επανάληψης, η μεταβλητή j έχει πάρει την τιμή forCount, οπότε στη θέση numOfIters-j βρίσκεται ο τελευταίος βρόχος που πρέπει να εκτυπωθεί. Κατόπιν, εκτυπώνεται και ο τελευταίος βρόχος με την εντολή της εκτύπωσής του είναι ακριβώς ίδια με τους προηγούμενους. Επειδή είναι ο τελευταίος, έπρεπε από κάτω του να παραχθεί το pragma «#pragma HLS PIPELINE\n"#pragma HLS UNROLL factor=1\n». Το pragma αυτό είναι και ο λόγος που εκτυπώνεται μόνος του στο τέλος, εξωτερικά από τον βρόχο επανάληψης που εκτυπώνει τους υπόλοιπους.

Κατόπιν αντιστοιχίζονται οι μεταβλητές val που περνάνε ως όρισμα στη συνάρτηση από τον κώδικα εισόδου με τις προσωρινές μεταβλητές var που παράγονται εντός της συνάρτησης. Η αντιστοίχιση αυτή γίνεται για όλες τις μεταβλητές var, εκτός από το τελικό αποτέλεσμα με την εκτύπωση της εντολής ανάθεσης: «`X*vari = vali;\n`», όπου X η συμβολοσειρά (string) tabs. Σε περίπτωση που υπάρχει μεταβλητή από σύνθετο τελεστή, κάποιο από τα αντίστοιχα flags (flag_sum, flag_dif, flag_prod, flag_quot) θα είναι ενεργοποιημένο και θα εκτυπωθεί μια ακόμα ανάθεση με την ίδια ακριβώς με παραπάνω συμβολοσειρά. Για παράδειγμα, αν υπάρχει κάποιο sum η παραπάνω εντολή μετατρέπεται στην: «`X*sum = vali;\n`», Το i

μετά το πέρας του προηγούμενου βρόχου επανάληψης θα έχει πάρει την τιμή variables, έτσι ώστε με το vali να αντιστοιχίζεται στην επιπλέον μεταβλητή που χρειάζεται για το sum, dif, prod ή quot. Ακολουθεί η εκτύπωση των πράξεων από τον πίνακα operation και η παραγωγή των απαραίτητων διευθύνσεων. Με τον ίδιο βρόχο επανάληψης που αντιστοιχίζει τις μεταβλητές του κώδικα εισόδου με αυτές εντός της συνάρτησης, γίνεται έλεγχος για να αντιστοιχηθεί με την εντολή ανάθεσης: «`X*vali = vari;\n`», όπου X η συμβολοσειρά (string) tabs μόνο για το τελικό αποτέλεσμα. Αν δεν εντοπιστεί το τελικό αποτέλεσμα σε κάποια μεταβλητή var, αυτό σημαίνει πως το τελικό αποτέλεσμα είναι κάποιο από τα sum, diff, prod, quot, οπότε αντιστοιχίζεται στο val το περιεχόμενο της μεταβλητής resultVar, που θα είναι μια από αυτές τις τέσσερις λέξεις και εκτυπώνεται και η εντολή «`Xvali++;\n`», όπου X η συμβολοσειρά (string) tabs και i η τιμή της μεταβλητής variables, η οποία επιτρέπει την αποθήκευση της νέας τιμής της μεταβλητής val στην επόμενη θέση της FIFO. Ομοίως με προηγουμένως, αν υπάρχει μεταβλητή από σύνθετο τελεστή, εκτυπώνεται ένα ακόμα «`Xvali++;\n`». Τέλος, εκτυπώνεται μια νέα γραμμή για να διαχωρίσει το τμήμα αυτό από το επόμενο.

Σε περίπτωση που το addrFlag είναι ενεργοποιημένο, δηλαδή η ένδειξη ότι το τελικό αποτέλεσμα είναι κάποιο στοιχείο ενός πίνακα, στο σημείο αυτό παράγεται η διεύθυνσή του. Η διεύθυνση υπολογίζεται από το από το άθροισμα της αρχικής διεύθυνσης του πίνακα και της θέσης του στοιχείου επί το μέγεθος μιας uint64_t μεταβλητής, δηλαδή 64 bits. Σε περίπτωση που ο πίνακας του αποτελέσματος έχει παραπάνω από μια διάσταση, είναι απαραίτητο η θέση κάθε στοιχείου να υπολογιστεί με τη γνώση των διαστάσεων του πίνακα. Για παράδειγμα για διδιάστατο πίνακα 10x10, δηλαδή 100 στοιχείων, που διασχίζεται από βρόχους επανάληψης για i από 0 έως 9 και j επίσης από 0 έως 9, η θέση κάθε στοιχείου είναι η i*10+j. Αρχικά, για i=0 παράγονται οι διευθύνσεις των στοιχείων στις θέσεις 0-9, για i=1 παράγονται οι διευθύνσεις των στοιχείων 1*10+(0-9), 10-19, κ.ο.κ. Για να γίνει αυτός ο υπολογισμός, απαιτείται ακόμα βοηθητική μεταβλητή που χρησιμοποιείται για τον υπολογισμό της διεύθυνσης είναι η numAd η οποία αρχικοποιείται στην τιμή 1 και κρατάει των αριθμό των διαστάσεων του αποτελέσματος που έχουν προσπελαστεί. Αρχικά παράγεται το πρώτο κομμάτι υπολογισμού της διεύθυνσης, με τις έως τώρα πληροφορίες μέσω της ακολουθίας «`Xaddr = init_address + (`», όπου X η συμβολοσειρά (string) tabs.

Κατόπιν, αναζητείται το resultVar στον πίνακα variableAfter για να ληφθεί το αντίστοιχο variableBefore, το οποίο αποθηκεύεται στην προσωρινή μεταβλητή tmpVar. Η μεταβλητή αυτή διαβάζεται χαρακτήρα-χαρακτήρα με σκοπό να μετρηθούν οι χαρακτήρες «[», με σκοπό να κρατηθεί στη μεταβλητή bracketCount ο αριθμός των διαστάσεων του πίνακα από τον οποίο προέρχεται το τελικό αποτέλεσμα. Στη συνέχεια διαβάζεται εκ νέου η προσωρινή μεταβλητή μέχρι πάλι να εντοπιστεί η πρώτη αγκύλη «[». Οι χαρακτήρες που ακολουθούν γράφονται σε μια νέα προσωρινή μεταβλητή, τη varAd, μέχρι να εντοπιστεί κάποια αγκύλη «]». Εδώ συγκρίνονται οι μεταβλητές bracketCount+1 και η numAd. Αν είναι ίσες, σημαίνει ότι

έχουν διαβαστεί όλοι οι απαραίτητοι δείκτες, οπότε αδειάζει η μεταβλητή varAd, μηδενίζεται το bracketCount, και το varAd γίνεται 1. Αλλιώς, το περιεχόμενο του varAd εκτυπώνεται στο τελικό αρχείο. Αν το varAd είναι πάνω από 1, σημαίνει ότι πρόκειται για τουλάχιστον τον δεύτερο δείκτη, οπότε το περιεχόμενο του varAd αντιστοιχίζεται με αυτά του πίνακα vari, που περιέχει τις μεταβλητές προσαύξησης των βρόχων. Όταν εντοπιστεί, εκτυπώνεται στο τελικό αρχείο η ακολουθία «*X+», όπου X το αντίστοιχο περιεχόμενο του πίνακα fin, που περιέχει τα τερματικά στοιχεία των βρόχων επανάληψης. Η διαδικασία αυτή εκτελείται με τον παρακάτω κώδικα:

```
for(k=0;k<30;k++)
{
     if(tmpVar[k]=='[')
     {
          for(l=k+1;l<20;l++)
          {
               if(tmpVar[l]==']')
               {
                    l++;
                    if(numAd==bracketCount+1)       //if
//characters in all brackets have been read end loop and
//reset variables
                    {
                         k=30;
                         l=20;
                         numAd=1;
                         bracketCount=0;
                         strcpy(varAd, "");
                    }
                    else
                    {
                         if(numAd>1)
                         {
                              for(n=numOfIters-forCount;
n<=numOfIters-1;  n++)  //for  all  for  loops  inside
//directives
                              {
                                   if(vari[n]==varAd[0])
                                   {
               //print increment variable
               fprintf(f,"*%s+", fin[n]);
                                   }
                              }
                         }
                         //print array length
                         fprintf(f,"(%s)",varAd);
                         numAd++;
                    }
                    strcpy(varAd, "");
               }
```

```
                else
                {
                        strcpy(charac, &tmpVar[l]);
                        strncat(varAd, charac, 1);
                }
            }
        }
}
//print variable type
fprintf(f, ")*sizeof(%s);\n", varType[j]);
```

Εικόνα 34: Κώδικας παραγωγής υπολογισμού διεύθυνσης

Μόλις τελειώσει η διαδικασία αυτή, εκτυπώνεται στο τελικό αρχείο και η ακολουθία «`)*sizeof(X);\n`», όπου X ο τύπος της μεταβλητής κλείνοντας έτσι τον υπολογισμό της διεύθυνσης. Στη συνέχεια εκτυπώνεται και η εντολή «`Xaddr++;\n`», όπου X η συμβολοσειρά (string) tabs, η οποία επιτρέπει την αποθήκευση της νέας τιμής της μεταβλητής addr στην επόμενη θέση της FIFO.

Κατόπιν, με βρόχο επανάληψης, όπως και προηγουμένως, εκτυπώνεται για κάθε μεταβλητή η εντολή «`Xvali++;\n`», όπου X η συμβολοσειρά (string) tabs και i η τιμή της μεταβλητής variables, η οποία επιτρέπει την αποθήκευση της νέας τιμής της μεταβλητής vali στην επόμενη θέση της FIFO.

Στη συνέχεια, εκτυπώνονται οι υπόλοιπες αγκύλες, ένα tab πιο μέσα τη φορά για ομοιόμορφη στοίχιση του αποτελέσματος, για να κλείσουν οι βρόχοι επανάληψης και η συνάρτηση process. Τελευταίο βήμα είναι το άδειασμα με βρόχο επανάληψης του πίνακα operation και ο μηδενισμός της μεταβλητής number, για να είναι έτοιμες για να παράξουν το επόμενο αρχείο process.

4.1.3 Συνάρτηση fetch

Σε αντίθεση με τη συνάρτηση process, η συνάρτηση fetch δεν χρειάζεται καμία άλλη γνώση, παρά μόνο τις μεταβλητές για τις οποίες θα φέρει τη διεύθυνση. Για το λόγο αυτό, η παραγωγή της ξεκινάει αμέσως, πανομοιότυπα με την συνάρτηση process, με τη δημιουργία και εγγραφή του αρχείου fetchnumOfFetch.cpp, όπου numOfFetch ο αριθμός του αρχείου fetch που παράγεται εκείνη τι στιγμή. Για το πρώτο αρχείο, η μεταβλητή numOfFetch έχει την τιμή 0, καθώς έτσι έχει αρχικοποιηθεί στην αρχή του προγράμματος. Έτσι, αποθηκεύεται στην προσωρινή μεταβλητή tmpFetchName η συμβολοσειρά (string) fetch0.cpp και δημιουργείται το αρχείο με αυτό το όνομα. Ύστερα εκτυπώνονται στην αρχή του προγράμματος οι ίδιες βιβλιοθήκες με τη συνάρτηση process, και μετά από μια ακόμα κενή γραμμή η αρχή του ορισμού της συνάρτησης «`void fetchUnitnumOfFetch(`», δηλαδή «void fetchUnit0(» για το πρώτο αρχείο. Κατόπιν η μεταβλητή numOfFetch αυξάνεται κατά 1 με αποτέλεσμα

το επόμενο αρχείο να ονομαστεί μέσω της μεταβλητής tmpFetchName σε fetch1.cpp, και αντίστοιχα η συνάρτησή του «`void fetchUnit1(`», κ.ο.κ.

Αιτήσεις προσβάσεων στη μνήμη ορίζονται μόνο για στοιχεία που βρίσκονται μέσα σε πίνακα, καθώς και στο δεξιό μέλος μιας ισότητας. Αυτό ισχύει διότι πριν την εκτέλεση της πράξης της συνάρτησης process είναι απαραίτητο να γνωρίζονται οι διευθύνσεις στη μνήμη των τελεστών της πράξης. Οπότε επόμενη δουλειά του κώδικα που παράγει τα αρχεία fetch είναι να ξεχωρίσει τους πίνακες.

Όπως αναφέρθηκε σε προηγούμενη ενότητα, όλες οι μεταβλητές που βρίσκονται ενός των βρόχων επανάληψης κάτω από τη λέξη-κλειδί DIRECTIVES έχουν αποθηκευτεί στον πίνακα συμβολοσειρών (strings) variableBefore. Μέσα σε βρόχο επανάληψης, το ένα μετά το άλλο, τα στοιχεία του πίνακα variableBefore αποθηκεύεται στην προσωρινή μεταβλητή tmpFetch. Όλες οι μεταβλητές, εκτός από το αποτέλεσμα που επιστρέφεται, μέσω της προσωρινής μεταβλητής tmpFetch διαβάζονται χαρακτήρα-χαρακτήρα. Μόλις εντοπιστεί σε κάποια μεταβλητή ο χαρακτήρας “[”, σημαίνει ότι η μεταβλητή αυτή είναι κάποια θέση πίνακα, οπότε αποθηκεύεται στον πίνακα fetchVars και τερματίζεται η ανάγνωση. Η δεύτερη περίπτωση που τερματίζεται η ανάγνωση είναι μόλις διαβαστεί ο χαρακτήρας “\0”, γεγονός που δηλώνει ότι διαβάστηκε ολόκληρη η μεταβλητή και δεν εντοπίστηκε κάποια αγκύλη, άρα πρόκειται για απλή μεταβλητή.

Στη συνέχεια, οι μεταβλητές του πίνακα ξαναδιαβάζονται με νέο βρόχο επανάληψης. Κάθε χαρακτήρας της μεταβλητής αποθηκεύεται σε νέο πίνακα συμβολοσειρών (strings) με όνομα newFetch. Μόλις εντοπιστεί ο χαρακτήρας «» τερματίζεται η ανάγνωση και ο βρόχος επανάληψης πάει στην επόμενη μεταβλητή. Με τη διαδικασία αυτή διαχωρίζεται το όνομα ενός πίνακα από τα στοιχεία του. Για παράδειγμα, για το στοιχείο πίνακα a[i], ο πίνακας fetchVars αποθηκεύει a[i], ενώ ο πίνακας newFetch αποθηκεύει a. Σκοπός του διαχωρισμού αυτού είναι να αναγωριστεί από το εργαλείο αν υπάρχουν παραπάνω από ένα στοιχεία του ίδιου πίνακα για να αντιστοιχηθεί στη διεύθυνση του στοιχείου η κατάλληλη αρχική διεύθυνση (init_address) του πίνακα.

Κατόπιν, ακολουθεί η παραγωγή των αρχικών διευθύνσεων για να τοποθετηθούν στα ορίσματα της συνάρτησης. Όλες οι διευθύνσεις πρέπει να είναι τύπου unsigned int, δηλαδή uint64_t στο Vivado. Είναι δεδομένο πως στον πηγαίο κώδικα, στις πράξεις που θα διαβαστούν μέσα στην οδηγία θα υπάρχει τουλάχιστον ένα στοιχείο ενός πίνακα. Ξεκινάει λοιπόν η διαδικασία εύρεσης του πίνακα αυτού στον πίνακα με τους τύπους μεταβλητών. Το πρώτο στοιχείο του πίνακα newFetch θεωρείται πως θα είναι ο πίνακας στον οποίο θα αντιστοιχίζεται η αρχική διεύθυνση init_address1. Οπότε εκτυπώνεται στο αρχείο η ακολουθία «`uint64_t init_address1`».

Κατόπιν, ξεκινάει βρόχος επανάληψης για να παραχθούν οι υπόλοιπες αρχικές διευθύνσεις. Αρχικά είναι απαραίτητο να εντοπιστούν διπλότυπα μέσα στον πίνακα

newFetch. Είναι αποδεκτό να συμμετέχουν σε πράξεις μέσα στην οδηγία παραπάνω από ένα στοιχεία του ίδιου πίνακα, είναι όμως λάθος να βρίσκεται τόσες φορές η αρχική διεύθυνση όσες και τα στοιχεία στα ορίσματα της συνάρτησης. Στον πρώτο πίνακα, στον οποίο έχει αντιστοιχθεί η αρχική διεύθυνση init_address1, αναθέτεται και ένας αριθμός, ο tableCount, αρχικά στην τιμή 1. Ο αριθμός αυτός κρατείται και στον πίνακα tableId για μελλοντική χρήση. Για κάθε στοιχείο μέσα στον πίνακα newFetch εκτελείται βρόχος επανάληψης που το συγκρίνει με όλα τα προηγούμενα στοιχεία του πίνακα. Αν βρεθεί ίδιο, σημαίνει ότι υπάρχει διπλότυπο οπότε αντιστοιχίζεται στο στοιχείο αυτό το tableId του προηγούμενου του ίδιου πίνακα ενεργοποιείται η προσωρινή μεταβλητή flagDup και τερματίζεται ο βρόχος επανάληψης χωρίς να εκτυπωθεί κάποιο init_address για το στοιχείο αυτό. Σε περίπτωση που δεν βρεθεί διπλότυπο, σημαίνει ότι η μεταβλητή αυτή είναι ενός διαφορετικού πίνακα, οπότε η προσωρινή μεταβλητή flagDup θα βρίσκεται ακόμη στην αρχική τιμή της 0, οπότε θα αυξηθεί η μεταβλητή tableCount κατά 1 και θα αντιστοιχηθεί στο στοιχείο αυτό το νέο tableId. Στο τελικό αρχείο εξόδου εκτυπώνεται για κάθε νέο πίνακα «, `uint64_t init_addressX`», όπου X η τιμή της μεταβλητής tableCount.

Στη συνέχεια, παράγεται η ομάδα των επόμενων ορισμάτων της συνάρτησης, οι μεταβλητές στις οποίες επιστρέφονται οι διευθύνσεις. Για i από 1 έως fetchCount, που είναι ο συνολικός αριθμός των μεταβλητών της fetch, εκτυπώνεται στο τελικό αρχείο για κάθε μεταβλητή «, `uint64_t *vali`».

Τελευταία ομάδα ορισμάτων της συνάρτησης fetch είναι όπως και στη συνάρτηση process τυχόν τερματικά στοιχεία βρόχων επανάληψης. Επειδή οι βρόχοι επανάληψης των δύο συναρτήσεων είναι ολόιδιοι, η παραγωγή των τερματικών τους στοιχείων υλοποιείται με τον ίδιο ακριβώς τρόπο που περιεγράφηκε και παραπάνω για τη συνάρτηση process.

Αφού έχουν εκτυπωθεί όλα τα ορίσματα της συνάρτησης, όπως ακριβώς και στη συνάρτηση process, εκτυπώνεται η ακολουθία «`)\n{`». Κλείνει η παρένθεση της συνάρτησης, κατόπιν ανοίγει σε νέα γραμμή η αγκύλη που θα περιλαμβάνει τον πηγαίο κώδικα της συνάρτησης. Τέλος, εκτυπώνεται μια γραμμή για να ξεκινήσει από εκεί η εκτύπωση του πηγαίου κώδικα για λόγους ομοιομορφίας. Με τη διαδικασία αυτή, για παράδειγμα για αρχείο εισόδου με τρεις μεταβλητές-στοιχεία δύο πινάκων ακεραίων, μέσα σε τριπλό βρόχο επανάληψης που τερματίζει στους ακεραίους xmax και ymax, και τον αριθμό 10, ο ορισμός της συνάρτησης του πρώτου αρχείου fetch (fetch0.cpp) που θα παραχθεί από το εργαλείο είναι ο παρακάτω:

```
void fetchUnit0(uint64_t init_address1, uint64_t
init_address2, uint64_t *val1, uint64_t *val2, uint64_t
*val3, int64_t xmax, int64_t ymax)
```

Εικόνα 35: Παράδειγμα ορισμού συνάρτησης fetch

Όπως και στη συνάρτηση process, παράλληλα με τα υπόλοιπα στοιχεία της συνάρτησης παράγονται και τα απαραίτητα για την ορθή λειτουργία στο Vivado, HLS Pragmas. Στο σημείο αυτό τοποθετούνται, όπως ακριβώς και στη συνάρτηση process, τα pragmas για το ρολόι, τη ροή δεδομένων (dataflow) και τα τερματικά στοιχεία των βρόχων επανάληψης. Το πόσα είναι τα υπόλοιπα ορίσματα είναι γνωστό στο εργαλείο, καθώς ο αριθμός των μεταβλητών βρίσκεται αποθηκευμένος στην μεταβλητή fetchCount, ενώ των πινάκων στην μεταβλητή tableCount. Είναι επίσης γνωστό ότι οι αρχικές διευθύνσεις είναι σταθερές μεταβλητές, ενώ οι διευθύνσεις αποθηκεύονται σε FIFO. Δεν χρειάζονται στην παρούσα φάση περαιτέρω πληροφορίες, για το λόγο αυτό τα pragmas για τις μεταβλητές αυτές παράγονται με έναν απλό βρόχο επανάληψης. Για τις μεταβλητές από 1 έως fetchCount εκτυπώνεται η συμβολοσειρά «`#pragma HLS INTERFACE ap_fifo depth=4096 port=valX\n`», όπου X ο αύξων αριθμός της μεταβλητής, ενώ για τις αρχικές διευθύνσεις, από 1 έως tableCount εκτυπώνεται η ακολουθία «`#pragma HLS INTERFACE ap_stable port=init_addressX\n`», όπου X ο αύξων αριθμός της αρχικής διεύθυνσης. Μετά από κάθε pragma εκτυπώνεται νέα γραμμή για λόγους ομοιομορφίας. Με τη διαδικασία αυτή, για το παραπάνω παράδειγμα των τριών μεταβλητών, των δύο πινάκων και των δύο τερματικών στοιχείων-μεταβλητές θα παραχθούν τα παρακάτω pragmas:

```
#pragma HLS INTERFACE ap_stable port=xmax
#pragma HLS INTERFACE ap_stable port=ymax
#pragma HLS INTERFACE ap_fifo depth=4096 port=val1
#pragma HLS INTERFACE ap_fifo depth=4096 port=val2
#pragma HLS INTERFACE ap_fifo depth=4096 port=val3
#pragma HLS INTERFACE ap_stable port=init_address1
#pragma HLS INTERFACE ap_stable port=init_address2
```

Εικόνα 36: Παράδειγμα δήλωσης των fetch pragmas

Ακολουθεί η παραγωγή σε μια γραμμή των μεταβλητών προσαύξησης των βρόχων επανάληψης, η οποία παράγεται με τον ίδιο ακριβώς τρόπο με την αντίστοιχη στη συνάρτηση process και έχει περιεγραφεί παραπάνω. Η αρχικοποίηση των μεταβλητών ολοκληρώνεται με τις προσωρινές που χρησιμοποιούνται στον υπολογισμό των διευθύνσεων. Η παραγωγή τους γίνεται με τον ίδιο τρόπο με αυτή των ορισμάτων val. Για i από 1 έως fetchCount, που είναι ο συνολικός αριθμός των μεταβλητών της fetch, εκτυπώνεται στο τελικό αρχείο για κάθε μεταβλητή η ακολουθία «`Xuint64_t addri = 0;\n`», όπου X η συμβολοσειρά (string) tabs. Κάθε μια από αυτές δηλώνεται σε δική της γραμμή όπως φαίνεται και από την εκτύπωση του χαρακτήρα «\n» στο τέλος και αρχικοποιείται στην τιμή 0. Για το παράδειγμα που αναφέρθηκε παραπάνω για μεταβλητές προσαύξησης i, j και k θα παραχθεί η παρακάτω αρχικοποίηση:

```
int64_t c, d, k;
```

```
uint64_t addr1 = 0;
uint64_t addr2 = 0;
uint64_t addr3 = 0;
```

Εικόνα 37: Παράδειγμα αρχικοποίησης μεταβλητών συνάρτησης fetch

Στη συνέχεια εκτυπώνονται οι βρόχοι επανάληψης που βρίσκονται μέσα στην οδηγία μαζί με τα pragma τους. Εφόσον είναι ακριβώς οι ίδιοι με αυτούς της συνάρτησης process εκτυπώνονται με την ίδια ακριβώς διαδικασία, απλά χωρίς την αρχικοποίηση των προσωρινών μεταβλητών, γιατί εδώ έχει ήδη προηγηθεί, και με διαφορετικά labels στους βρόχους. Αντί για το label «`processUnitYlabelZ:`», όπου Y η τιμή numberOfProcess-1, δηλαδή ο αριθμός του αρχείου process το οποίο παράγεται σε αυτή τη φάση, τα labels του fetch είναι «`fetchUnitYlabelZ:`», όπου Y η τιμή numberOfFetch-1, δηλαδή ο αριθμός του αρχείου fetch το οποίο παράγεται σε αυτή τη φάση. Στην περίπτωση αυτή, η μόνη του διαφορά του τελευταίου βρόχου από τους προηγούμενους, και ο λόγος που δεν εκτυπώθηκε εντός του βρόχου μαζί με αυτούς είναι το pragma του, το οποίο όπως και στην process, είναι pipeline με unroll factor αντί για loop_merge.

Ακολουθεί το πιο ουσιώδες κομμάτι της συνάρτησης fetch, η παραγωγή των διευθύνσεων. Για τον υπολογισμό της διεύθυνσης κάθε μεταβλητής χρειάζονται ακόμα και τα περιεχόμενα των αγκυλών της, οπότε στο σημείο αυτό, οι μεταβλητές διαβάζονται από τον πίνακα fetchVars και όχι από τον newFetch. Επίσης, όλες οι μεταβλητές των πράξεων εντός της οδηγίας έχουν λάβει από τη διαδικασία παραγωγής της συνάρτησης process ένα όνομα της μορφής varX, όπου X ο αύξων αριθμός. Τα ονόματα αυτά βρίσκονται στον πίνακα variableAfter. Ξεκινάει βρόχος επανάληψης για κάθε μεταβλητή που χρειάζεται διεύθυνση, δηλαδή για i από 1 έως forCount. Αρχικά, η μεταβλητή όπως βρίσκεται στον πίνακα fetchVars και το αντίστοιχο όνομα από τον πίνακα variableAfter αποθηκεύονται στις προσωρινές μεταβλητές tmpVar και tmpVar2 για ευκολότερη επεξεργασία, οι οποίες έχουν προηγουμένως αδειάσει. Αρχικά παράγεται το πρώτο κομμάτι υπολογισμού της διεύθυνσης, με τις έως τώρα πληροφορίες μέσω της ακολουθίας «`XaddrY = init_addressZ + (`», όπου X η συμβολοσειρά (string) tabs, Y η τιμή του i και Z το id του πίνακα που βρίσκεται η μεταβλητή από τον πίνακα tableId. Το υπόλοιπο κομμάτι υπολογισμού χρειάζεται τα δεδομένα εντός των αγκυλών της μεταβλητής και εκτελείται με τον ίδιο ακριβώς τρόπο με τη διεύθυνση της μεταβλητής αποτελέσματος της συνάρτησης process.

Στη συνέχεια, όπως και στη συνάρτηση process εκτυπώνεται και η εντολή «`X*valY = addrY;\n`», όπου X η συμβολοσειρά (string) tabs, και Y η τιμή του i, η οποία αναθέτει τη διεύθυνση που μόλις υπολογίστηκε στην αντίστοιχη μεταβλητή val, για να επιστραφεί στον πηγαίο κώδικα ως όρισμα της συνάρτησης και η εντολή «`Xvali++;\n`», όπου X η συμβολοσειρά (string) tabs και i η τιμή της μεταβλητής variables, η οποία επιτρέπει την αποθήκευση της νέας τιμής της μεταβλητής val στην επόμενη θέση της FIFO.

Τελευταία βήματα όπως και στη συνάρτηση process, είναι η εκτύπωση των υπόλοιπων αγκυλών, ένα tab πιο μέσα τη φορά για ομοιόμορφη στοίχιση του αποτελέσματος, για να κλείσουν οι βρόχοι επανάληψης και η συνάρτηση fetch. Τέλος, με βρόχο επανάληψης αδειάζουν οι πίνακες fetchVars, newFetch, newVars, variableBefore και variableAfter για να είναι έτοιμες για να παράξουν το επόμενο αρχεία συναρτήσεων fetch και process.

4.2 HLS Pragmas

Προκειμένου τα αρχεία .cpp να είναι έτοιμα για να εκτελεστούν στην FPGA, ήταν απαραίτητο να προστεθούν λίγες γραμμές κώδικα της μορφής #pragma HLS, τα λεγόμενα HLS pragma. Η παραγωγή των γραμμών αυτών επιτεύχθηκε με τη χρήση των εντολών printf(); σε διάφορα σημεία στον κανόνα της πρόσθετης οδηγίας. Τα pragmas είναι κοινά για τις δύο συναρτήσεις.

Τα πρώτα pragmas τα οποία αφορούν το ρολόι, τη ροή δεδομένων (dataflow), τις μεταβλητές και τις διευθύνσεις και τοποθετούνται ανάμεσα στη δήλωση της συνάρτησης και τις μεταβλητές της. Αρχικά παράγεται φυσικά το ρολόι, απαραίτητο στοιχείο για κάθε ψηφιακό κύκλωμα. Δηλώνεται με την εντολή `#pragma HLS CLOCK domain=default` με όνομα default. Κατόπιν, δηλώνεται το pragma που ενεργοποιεί τη διοχέτευση (pipeline) σε επίπεδο διεργασιών, το `#pragma HLS dataflow` [30]. Το Vivado HLS, μέσω του pragma αυτού επιτρέπει σε μεταγενέστερες συναρτήσεις ή βρόχους επανάληψης να εκτελούν πράξεις πριν ολοκληρωθεί η προσπέλαση όλων των δεδομένων προηγούμενων. Με τον τρόπο αυτό ελαχιστοποιούνται καθυστερήσεις και επιτυγχάνεται ταυτοχρονισμός διεργασιών.

Ακολουθεί η δήλωση εισόδου/εξόδου. Τα ορίσματα της συνάρτησης ορίζονται ως #pragma HLS interface. Το pragma αυτό ορίζει τον τρόπο με τον οποίο δημιουργούνται θύρες RTL από τα ορίσματα της συνάρτησης κατά τη σύνθεση της διεπαφής [31]. Οι μεταβλητές των οποίων οι τιμές δεν αλλάζουν ορίζονται ως stable, δηλαδή με το pragma `#pragma HLS INTERFACE ap_stable port=var`. Το Vivado θεωρεί πως η τιμή τους δεν μεταβάλλεται μετά το reset, οπότε εσωτερικές βελτιστοποιήσεις αφαιρούν περιττούς καταχωρητές. Οι υπόλοιπες δηλώνονται ως μια απλή FIFO με πύλες εισόδου και εξόδου με το pragma `#pragma HLS INTERFACE ap_fifo depth=4096 port=var`. Το μέγεθός της ορίστηκε αυθαίρετα ως 4096 θέσεις.

Μετά τις δηλώσεις, ξεκινάει η εκτέλεση των πράξεων. Οι πράξεις βρίσκονται εντός βρόχων επανάληψης. Κάτω από κάθε βρόχο, εκτός από τον τελευταίο ο οποίος θα αναλυθεί παρακάτω, τοποθετείται το pragma `#pragma HLS LOOP_MERGE`. Το pragma αυτό μειώνει τη συνολική καθυστέρηση με δύο τρόπους. Μειώνει τον αριθμό

κύκλων ρολογιού που απαιτούνται για τις μεταβάσεις στις υλοποιήσεις εντός του βρόχου, και επιτρέπει όπου είναι δυνατό να υλοποιηθούν βρόχοι παράλληλα [32].

Στον τελευταίο βρόχο προσθέτονται δύο μοναδικά pragmas, τα pipeline και unroll. Το pipeline δηλώνεται ως `#pragma HLS PIPELINE` και επιτρέπει την παράλληλη εκτέλεση διεργασιών [33]. Προεπιλέγει το διάστημα ενσωμάτωσης στο 1, γεγονός το οποίο σημαίνει ότι ανά ένα, δηλαδή σε κάθε κύκλο επεξεργάζεται νέες εισόδους. Το pragma unroll δημιουργεί πολλαπλά αντίγραφα των πράξεων εντός του βρόχου, επιτρέποντας έτσι σε μερικές ή όλες τις επαναλήψεις του βρόχου να εκτελεστούν παράλληλα [34]. Ορίζεται ως `#pragma HLS UNROLL factor=1`. Ο συντελεστής 1 επιτρέπει τη δημιουργία ενός αντιγράφου του σώματος του βρόχου σε κάθε επανάληψη, παράγοντας έτσι τη μικρότερη δυνατή παραλληλία, με αποτέλεσμα τη μείωση του αριθμού των επαναλήψεων.

ΚΕΦΑΛΑΙΟ 5-ΕΠΙΒΕΒΑΙΩΣΗ ΛΕΙΤΟΥΡΓΙΑΣ ΚΑΙ ΑΝΑΛΥΣΗ ΑΠΟΔΟΣΗΣ ΣΥΣΤΗΜΑΤΟΣ

5.1 Επιβεβαίωση λειτουργίας έτοιμου λεκτικού και συντακτικού αναλυτή

Πρώτο βήμα ήταν η επιβεβαίωση της ορθής λειτουργίας του έτοιμου λεκτικού και συντακτικού αναλυτή. Για να παραχθούν ο λεκτικός και ο συντακτικός αναλυτής από τις περιγραφές που δόθηκαν, ήταν απαραίτητο να εγκατασταθούν τα εργαλεία lex και yacc σε λειτουργικό σύστημα τύπου Ubuntu [23], όπως και έγινε. Αρχικά, με την εκτέλεση της εντολής μεταγλώττισης με τη χρήση του lex, `lex project.l`, παράχθηκε ο λεκτικός αναλυτής. Έπειτα παράχθηκε ο συντακτικός αναλυτής, μέσω της εντολής μεταγλώττισης με τη χρήση του yacc, `yacc -v project.y`, όπως πρότεινε ο συγγραφέας της περιγραφής. Διαπιστώθηκε όμως ότι αυτό ήταν λάθος, διότι η εντολή αυτή δεν δημιούργησε το αρχείο y.tab.h, το οποίο περιέχει τους ορισμούς των συμβόλων. Από εμπειρία πάνω στη συγγραφή μεταγλωττιστών γρήγορα επιβεβαιώθηκε ότι για να παραχθεί το αρχείο αυτό, έπρεπε το -v της εντολής να αντικατασταθεί με -d. Ο τελικός λεκτικός και συντακτικός αναλυτής a.out, παράχθηκε μέσω της εντολής `gcc -o a.out y.tab.c lex.yy.c -lfl -lm`.

Με δύο παραδείγματα επιβεβαιώθηκε η λειτουργία του αρχικού αναλυτή, ο οποίος απλώς επιβεβαίωνε αν το πρόγραμμα εισόδου είναι λεκτικά και συντακτικά ορθό ή όχι. Αρχικά, εκτελέστηκε με την εντολή `./a.out < inp` το πρόγραμμα που παρείχε ο προγραμματιστής που έγραψε τις περιγραφές των δύο αναλυτών. Το πρόγραμμα αυτό υλοποιεί ένα δένδρο και περιείχε δομές δεδομένων και ελέγχου, δείκτες και συνδεδεμένες λίστες. Το δεύτερο δείγμα κώδικα συγγράφηκε για να επιβεβαιωθεί ότι περνάνε από λεκτική και συντακτική ανάλυση άλλα στοιχεία της γλώσσας C. Το δείγμα αυτό περιείχε δομές επανάληψης, πράξεις, πίνακες, αναγνώσεις από το πληκτρολόγιο, εκτυπώσεις στην οθόνη και συναρτήσεις. Και τα δύο προγράμματα πέρασαν επιτυχώς τη συντακτική και λεκτική ανάλυση, με αποτέλεσμα ο λεκτικός και συντακτικός αναλυτής να κριθεί έγκυρος και κατάλληλος για τροποποίηση για να εξυπηρετηθούν οι ανάγκες της διπλωματικής.

5.2 Επιβεβαίωση λειτουργίας εργαλείου

Προκειμένου να επιβεβαιωθεί η ορθή λειτουργία και σωστή συνεργασία των συναρτήσεων fetch και process, και του εργαλείου γενικότερα, είναι απαραίτητο να γίνει σύνθεση των δύο συναρτήσεων με το Vivado HLS. Για το σκοπό αυτό, επιλέχθηκαν τρεις κώδικες σε C με διαφορετικά χαρακτηριστικά ο καθένας με σκοπό

να αναδειχθεί η καθολικότητα των εφαρμογών που δέχεται και επεξεργάζεται το εργαλείο.

Για κάθε εφαρμογή, ο κώδικας εισόδου στο εργαλείο με την οδηγία DIRECTIVES μετατρέπεται σε test bench. Αρχικά, αφαιρείται η οδηγία DIRECTIVES και οι αγκύλες της, έτσι ώστε κατά την εκτέλεση του προγράμματος να υπολογιστούν τα αποτελέσματα των πράξεων χωρίς τη χρήση των συναρτήσεων fetch και process. Τα αποτελέσματα αυτά θα συγκριθούν αργότερα με τις αντίστοιχες εξόδους της συνάρτησης process προκειμένου να επιβεβαιωθεί η ορθή λειτουργία των συναρτήσεων fetch και process. Εντός του test bench καλούνται οι συναρτήσεις των αρχείων fetch.cpp και process.cpp. Στη συνέχεια, ακολουθεί η δήλωση των FIFO συγκεκριμένων θέσεων στις οποίες αποθηκεύονται τα δεδομένα που επιστρέφουν οι συναρτήσεις. Κάθε μια από τις εφαρμογές έχει ανάγκη διαφορετικούς πόρους της FPGA, οπότε το μέγεθος των FIFO είναι διαφορετικός για κάθε μια. Οι διευθύνσεις που παράγει και επιστρέφει η συνάρτηση fetch αποθηκεύονται στις αντίστοιχες FIFO. Στη συνέχεια, τα δεδομένα που επιστρέφονται από την ανάγνωση των θέσεων μνήμης που έχουν προκύψει από τη συνάρτηση fetch δίνονται ως ορίσματα στη συνάρτηση process για να εκτελεστούν οι πράξεις. Τα αποτελέσματα ελέγχονται σε όλες τις περιπτώσεις και παρατηρείται πως ταυτίζονται με αυτά του λογισμικού επιβεβαιώνοντας έτσι την ορθή εκτέλεση των αποτελεσμάτων που υπολογίζει η συνάρτηση process και κατ’ επέκτασιν την ορθότητα των συναρτήσεων που παράγει το εργαλείο.

Η απεικόνιση σε υλικό υλοποιείται πολλαπλές φορές για κάθε εφαρμογή, κάθε φορά με μεγαλύτερο μέγεθος πινάκων που συμμετέχουν στις πράξεις για να συγκριθεί η απόδοση σε συνάρτηση με το μέγεθος. Για απεικόνιση σε υλικό, το Vivado HLS δέχεται μια από της συναρτήσεις fetch και process ως top. Οπότε πραγματοποιήθηκαν δοκιμές για όλα τα μεγέθη και όλες τις συναρτήσεις ως top. Σύγκριση και σχολιασμός των αποτελεσμάτων αυτών, καθώς και λεπτομέρειες σχετικά με την υλοποίηση κάθε εφαρμογής στο Vivado HLS παρατίθεται στις επόμενες ενότητες.

Οι δοκιμές πραγματοποιήθηκαν σε λειτουργικό σύστημα Ubuntu έκδοσης 16.04. Το Vivado HLS έκδοσης 2017.2 από τον server zeus του Εργαστηρίου Μικροεπεξεργαστών και Υλικού της σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Πολυτεχνείου Κρήτης και η FPGA για την οποία προσομοιώθηκε η απεικόνιση σε υλικό είναι η XCVU190-FLGB2104-1-I της οικογένειας Virtex UltraScale, η FPGA με τη μεγαλύτερη χωρητικότητα από τις διαθέσιμες του server.

5.2.1 Εφαρμογή 1 - Πράξεις στοιχείων διδιάστατων πινάκων

Πρώτη εφαρμογή είναι μια πράξη δύο διδιάστατων πινάκων που αποθηκεύει το αποτέλεσμα της πράξης σε τρίτο πίνακα. Η πράξη είναι η παρακάτω:

$$C_{ij} = (A_{ij} \times B_{ij}) + (A_{ji} \times B_{ji})$$

Εικόνα 38: Βασική πράξη Εφαρμογής 1

Η εφαρμογή αυτή γράφτηκε με σκοπό να αποδείξει ότι το εργαλείο μπορεί να εκτελέσει σωστά πράξεις πινάκων και να φέρει από τη μνήμη και να επεξεργαστεί ταυτόχρονα διαφορετικά στοιχεία του ίδιου πίνακα. Γεμίζει αυτόματα τους δύο διδιάστατους πίνακες με τυχαίους ακεραίους και εντός της οδηγίας DIRECTIVES περιέχει την παρακάτω πράξη μεταξύ των πινάκων:

```
for(c=0;c<m;c++)
{
      for(d=0;d<q;d++)
      {
            multiply[c][d] = first[c][d] * second[c][d] -
(first[d][c] * second[d][c]);
      }
}
```

Εικόνα 39: Κώδικας Εφαρμογής 1 για απεικόνιση σε υλικό

Η πράξη αυτή θα μετασχηματιστεί από το εργαλείο στην παρακάτω:

```
for(c=0;c<m;c++)
{
      for(d=0;d<q;d++)
      {
            var1 = var2 * var3 - (var4 * var5);
      }
}
```

Εικόνα 40: Πράξη Εφαρμογής 1 (process)

Χρειάζονται συνολικά 9 FIFO. 4 τύπου uint64_t για τις διευθύνσεις και 5 για τα δεδομένα, ίδιου τύπου με τις αντίστοιχες μεταβλητές, int64_t στο συγκεκριμένο παράδειγμα. Οι FIFO των διευθύνσεων είναι οι address0, address1, address2 και address3. Οι μεταβλητές var2, var3, var4 και var5 αντιστοιχίζονται στις FIFO data0, data1, data2 και data3. Η μεταβλητή var1, εφόσον είναι το τελικό αποτέλεσμα αντιστοιχίζεται στη FIFO result. Λόγω περιορισμών του Vivado HLS, δεν είναι δυνατόν να περάσουν σε συνάρτηση ως ορίσματα στοιχεία διδιάστατων πινάκων, οπότε τα δεδομένα των πινάκων first, second και multiply περνάνε με μετασχηματισμό σε μονοδιάστατους. Στο συγκεκριμένο, που οι πίνακες είναι στην πρώτη δοκιμή 10 θέσεων, δημιουργήθηκαν οι αντίστοιχοι πίνακες για το υλικό παραπάνω θέσεων, πάλι τύπου int64_t, με ονόματα multiply_hw, first_hw, second_hw. Μετά την αντιγραφή των πινάκων του λογισμικού στους αντίστοιχους του υλικού λαμβάνονται οι διευθύνσεις των πινάκων του υλικού μέσω της συνάρτησης fetch και αποθηκεύονται στις FIFO των διευθύνσεων. Στη συνέχεια, με βρόχο επανάληψης ίδιο με αυτόν εντός της οδηγίας λαμβάνονται τα δεδομένα μέσω των διευθύνσεων εντός των FIFO των διευθύνσεων και αποθηκεύονται στις FIFO των δεδομένων.

```
for (c = 0; c < m; c++)
{
     for (d = 0; d < q; d++)
     {
          data0[c*q+d] = first_hw[(address0[c*q+d]-
(int64_t)&first_hw[0])/sizeof(int64_t)];
          data1[c*q+d] = second_hw[(address1[c*q+d]-
(int64_t)&second_hw[0])/sizeof(int64_t)];
          data2[d*m+c] = first_hw[(address2[d*m+c]-
(int64_t)&first_hw[0])/sizeof(int64_t)];
          data3[d*m+c] = second_hw[(address3[d*m+c]-
(int64_t)&second_hw[0])/sizeof(int64_t)];
     }
}
```

Εικόνα 41: Προφόρτωση διευθύνσεων Εφαρμογής 1 (fetch)

Κατόπιν, οι FIFO των δεδομένων δίδονται ως ορίσματα στη συνάρτηση process για να εκτελέσει τις πράξεις και τα αποτελέσματα αποθηκεύονται στη FIFO result. Για την εξακρίβωση των αποτελεσμάτων ελέγχεται αν όλα τα περιεχόμενα του πίνακα multiply είναι ίδια με αυτά της FIFO result. Εντοπίστηκαν όλα ίδια, οπότε επιβεβαιώνεται ότι η χρήση των συναρτήσεων fetch και process του υλικού παράγει ακριβώς τα ίδια αποτελέσματα με αυτά του κώδικα λογισμικού. Τελικό βήμα πριν την σταδιακή αύξηση του μεγέθους των πινάκων για να ληφθούν οι απαραίτητες μετρήσεις είναι ο καθορισμός του μεγέθους των FIFO, το οποίο ήταν 4096 όπως ορίστηκε αυθαίρετα από το εργαλείο. Μετά από δοκιμές καθορίστηκε το μεγαλύτερο δυνατό μέγεθος FIFO ως 65536. Οπότε, στο test bench και στις συναρτήσεις fetch και process το μέγεθος των FIFO άλλαξε από 4096 σε 65536.

5.2.2 Εφαρμογή 2 - Άθροισμα ανά γραμμή και ανά στήλη

Η δεύτερη εφαρμογή βρέθηκε μετά από αναζήτηση στο διαδίκτυο και υλοποιεί το άθροισμα πρώτα των γραμμών και μετά των στηλών ενός διδιάσταστου πίνακα.[36] Τα αποτελέσματα αποθηκεύονται σε δύο μονοδιάστατους και υπολογίζονται σε διαφορετικούς εμφωλευμένους βρόχους επανάληψης.

$$S_i = S_i + X_{ij}$$

$$S = [\Sigma_{i1}\Sigma_{i2}\Sigma_{ij}]$$

Εικόνα 42: Άθροισμα ανά γραμμή

$$S_j = S_j + X_{ji}$$

$$S = \begin{bmatrix} \Sigma_{1j} \\ \Sigma_{2j} \\ \Sigma_{ij} \end{bmatrix}$$

Εικόνα 43: Άθροισμα ανά στήλη

Η οδηγία DIRECTIVES τοποθετήθηκε και στους δύο βρόχους με σκοπό να αποδείξει ότι για παραπάνω από μια οδηγίες το εργαλείο παράγει τον παραπάνω απαιτούμενο αριθμό αρχείων. Μια τροποποίηση έγινε στον αρχικό κώδικα έχει να κάνει με τον υπολογισμό της ίδιας της πράξης. Στον πρωτότυπο κώδικα, για τον υπολογισμό των αθροισμάτων ανά γραμμή υπήρχε ο παρακάτω κώδικας:

```
for(i=0;i<n;i++)
{
      rsum[i] = 0;
      for(j=0;j<n;j++)
            rsum[i] = rsum[i] + arr1[i][j];
}
```

Εικόνα 44: Κώδικας Εφαρμογής 2 για το άθροισμα ανά γραμμή

Επειδή όμως στο υλικό δεν είναι δυνατόν να γίνεται ανάγνωση και εγγραφή ταυτόχρονα στην ίδια διεύθυνση χρησιμοποιήθηκε η προσωρινή μεταβλητή sum και ο κώδικας που πέρασε από το εργαλείο υπολογίζει την ίδια πράξη με τον παρακάτω τρόπο:

```
for(i=0;i<n;i++)
{
      for(j=0;j<n;j++)
      {
            sum +=  arr1[i][j];
            rsum[i] = sum;
      }
      sum = 0;
}
```

Εικόνα 45: Κώδικας Εφαρμογής 2 για το άθροισμα ανά στήλη

Με τον τρόπο αυτό αποδεικνύεται επίσης ότι λειτουργεί σωστά η πράξη με σύνθετο τελεστή, μετατρέποντας στην συνάρτηση process την έκφραση «sum +=» σε «sum = sum + ». Επιπλέον παρατηρείται ότι αναγνωρίζεται και δουλεύει σωστά και η πράξη ανάθεσης, αλλά και το γεγονός ότι το εργαλείο αναγνωρίζει πως η μεταβλητή με το όνομα sum βρίσκεται μέσα στους βρόχους πάνω από μια φορά και της αναθέτει τη δεύτερη φορά το ίδιο όνομα με αυτό της πρώτης φοράς και δεν της δίνει κάποιο νέο.

Στην πρωτότυπη εφαρμογή οι πίνακες είχαν μέγεθος 10 θέσεων και γέμιζαν από τον χρήστη με το πληκτρολόγιο, για λόγους όμως ευκολίας στις δοκιμές αυτό άλλαξε. Το γέμισμά τους γίνεται πλέον με τυχαίους ακεραίους, παρόμοια με την πρώτη εφαρμογή. Αρχικά υλοποιείται σε λογισμικό και σε υλικό το άθροισμα ανά γραμμή και μετά σε λογισμικό και σε υλικό το άθροισμα ανά στήλη με την ίδια ακριβώς διαδικασία, που περιγράφεται παρακάτω. Καθώς η πράξη είναι ίδια και για τα δύο αθροίσματα και αλλάζουν μόνο οι διευθύνσεις, παράχθηκαν από το εργαλείο δύο αρχεία process με την ίδια ακριβώς πράξη, την παρακάτω:

```
for(i = 0; i < n; i++)
```

```
{
      for(j = 0; j < n; j++)
      {
            var1 = *val1;

            sum = sum + var1;
            var2 = sum;

            *val2 = var2;

            *addr = init_address + ((i))*sizeof(int64_t);
            addr++;

            val1++;
            val2++;
      }
}
```

Εικόνα 46: Πράξεις Εφαρμογής 2 (process)

Η πράξη του αθροίσματος ανά στήλη δεν υλοποιήθηκε στο λογισμικό με τον σύνθετο τελεστή +=, αλλά με τον απλό τελεστή πρόσθεσης με τον παρακάτω κώδικα:

```
for(i=0;i<n;i++)
{
      for(j=0;j<n;j++)
      {
            sum = sum + arr1[j][i];
            csum[i] = sum;
      }
      sum = 0;
}
```

Εικόνα 47: Παραλλαγή αθροίσματος ανά στήλη

Και στις δύο περιπτώσεις, η πράξη είναι ίδια, γράφεται όμως με διαφορετικό τρόπο στην κάθε μια. Σκοπός αυτού είναι να δείξει πως το εργαλείο αναγνωρίζει τις πράξεις ως ίδιες και παράγει πανομοιότυπα αρχεία process, το οποίο για την δεύτερη περίπτωση είναι το παρακάτω:

```
for(i = 0; i < n; i++)
{
      for(j = 0; j < n; j++)
      {
            var1 = *val1;
            var2 = *val2;

            var1 = var1 + var2;

            var3 = var1;

            *val3 = var3;
```

```
            *addr = init_address + ((i))*sizeof(uint64_t);
            addr++;

            val1++;
            val2++;
            val3++;
      }
}
```

Εικόνα 48: Πράξεις παραλλαγής Εφαρμογής 2 (process)

Οι διαφορές ανάμεσα στις δύο μεθόδους, είναι πως η πρώτη αντιμετωπίζει το sum ως εσωτερική μεταβλητή και απαιτείται να τοποθετηθεί από τον χρήστη μηδενισμός της στο κατάλληλο σημείο που ορίζει ο κώδικας λογισμικού. Στην δεύτερη περίπτωση δεν απαιτείται ο μηδενισμός, μιας και το εργαλείο παράγει μια FIFO για τη μεταβλητή var3, ίδιου τύπου με την αντίστοιχη του κώδικα λογισμικού, int64_t στο συγκεκριμένο παράδειγμα, στην οποία αποθηκεύονται σε κάθε επανάληψη οι νέες τιμές που αποκτά η μεταβλητή.

Χρειάζονται συνολικά, εκτός από την προαναφερθείσα, 3 FIFO για κάθε περίπτωση. Μια τύπου uint64_t για τις διευθύνσεις και 2 για τα δεδομένα, ίδιου τύπου με τις αντίστοιχες μεταβλητές, int64_t στο συγκεκριμένο παράδειγμα. Οι FIFO των διευθύνσεων είναι οι address0 και address1. Η μεταβλητή var2 κάθε εφαρμογής αντιστοιχίζονται στις FIFO data0 και data1. Η μεταβλητή var1, εφόσον είναι το τελικό αποτέλεσμα αντιστοιχίζεται στις FIFO result0 και result1, ανάλογα με την περίπτωση. Επειδή όμως η FIFO που αποθηκεύονται τα αποτελέσματα είναι μονοδιάστατη με θέσεις i και το κάθε άθροισμα των στοιχείων j αποθηκεύεται σε νέα θέση είναι απαραίτητο το sum να μηδενίζεται μετά το πέρας του βρόχου επανάληψης με βήμα j. Για το λόγο αυτό, τοποθετήθηκε εξωτερικά του, αλλά εσωτερικά του βρόχου επανάληψης με βήμα i η εντολή sum = 0 για να αποθηκεύονται τα σωστά αποτελέσματα στη FIFO. Με πανομοιότυπο τρόπο με την εφαρμογή 1 γίνεται η αντιστοίχιση των πινάκων στις FIFO του υλικού, η παραγωγή των διευθύνσεων μέσω της fetch και αποθήκευσή τους στις FIFO των διευθύνσεων, η αντιστοίχιση των στοιχείων των διευθύνσεων στις FIFO των δεδομένων, και η χρήση αυτών για την εκτέλεση των πράξεων με τη συνάρτηση fetch. Η διαδικασία για την εξακρίβωση των αποτελεσμάτων είναι ίδια με την εφαρμογή 1, με μια μικρή διαφορά. Το αποτέλεσμα παράγεται με τη μεταβλητή var2 και μέσω της εντολής val2++ αποθηκεύεται στη FIFO. Η εντολή αυτή βρίσκεται εντός και των δύο βρόχων επανάληψης, γεγονός που σημαίνει ότι αποθηκεύονται σε αυτήν όλα τα επιμέρους αθροίσματα. Οπότε το τελικό άθροισμα της πρώτης γραμμής του αρχικού πίνακα θα βρίσκεται στη θέση n-1 της FIFO, το τελικό άθροισμα της δεύτερης στη θέση 2*(n-1) κ.ο.κ. Για το λόγο αυτό, στον τελικό έλεγχο των αποτελεσμάτων, με βρόχο επανάληψης για i από 0 έως n, με n για παράδειγμα ίσο με 10, ελέγχονται τα αποτελέσματα του πίνακα λογισμικού στις θέσεις 0, 1, 2 κ.ο.κ και δείκτη i με τις αντίστοιχες θέσεις 9, 19, 29 κ.ο.κ της FIFO και δείκτη (i+1)*n-1. Όπως και στην πρώτη εφαρμογή, εντοπίστηκαν όλα τα ανωτέρω στοιχεία ίδια, οπότε επιβεβαιώνεται ότι η χρήση των συναρτήσεων fetch και process

του υλικού παράγει ακριβώς τα ίδια αποτελέσματα με αυτά του κώδικα λογισμικού. Τελικό βήμα πριν την σταδιακή αύξηση του μεγέθους των πινάκων για να ληφθούν οι απαραίτητες μετρήσεις είναι ο καθορισμός του μεγέθους των FIFO, το οποίο ήταν 4096 όπως ορίστηκε αυθαίρετα από το εργαλείο. Μετά από δοκιμές καθορίστηκε το μεγαλύτερο δυνατό μέγεθος FIFO ως 131072. Οπότε, στο test bench και στις συναρτήσεις fetch και process το μέγεθος των FIFO άλλαξε από 4096 σε 131072.

5.2.3 Εφαρμογή 3 - Ανίχνευση ακμών

Η τρίτη εφαρμογή είναι αλγόριθμος επεξεργασίας εικόνας, και συγκεκριμένα ανίχνευσης ακμών. Οι λόγοι επιλογής του συγκεκριμένου αλγορίθμου είναι ο μεγάλος αριθμός βρόχων επανάληψης και κατά συνέπεια η σωστή επιλογή τερματικών στοιχείων, καθώς και οι εξαρτήσεις μεταξύ διάφορων μεταβλητών. Ο αλγόριθμος περιλαμβάνεται στο 5ο κεφάλαιο του βιβλίου Image processing in C [28].

$$C_{ij} = \sum_{1}^{rows-1} \sum_{1}^{cols-1} A_{(i+a)(j+b)} \times B_{(a+1)(b+1)}$$

Εικόνα 49: Ανίχνευση ακμών εικόνας A με τη χρήση μάσκας B

Λόγω περιορισμών στην υλοποίηση του εργαλείου και παλαιότητας του κώδικα, ο κώδικας υπέστη μερική επεξεργασία με σκοπό την εκτέλεσή τους η οποία όμως δεν αλλοιώνει τη λειτουργικότητά τους. Η εφαρμογή αυτή έχει πράξη απλούστερη των ανωτέρω εφαρμογών, την παρακάτω:

```
for(i=1; i<rows-1; i++)
{
      for(j=1; j<cols-1; j++)
      {
            sum = 0;
            for(a=-1; a<2; a++)
            {
                  for(b=-1; b<2; b++)
                  {
                        sum = sum + image[i+a][j+b] *
mask_0[a+1][b+1];
                  }
            }
            out_image[i][j] = sum;
      }
}
```

Εικόνα 50: Κώδικας Εφαρμογής 3 για απεικόνιση σε υλικό

Η ιδιαιτερότητα όμως αυτής της εφαρμογής ήταν πως περιέχει αυτόν τον κώδικα συνολικά 8 φορές, με άλλη μάσκα κάθε φορά. Οπότε προέκυψε η ευκαιρία για δοκιμή του εργαλείου με αυτόν τον κώδικα με δύο τρόπους. Επειδή το μόνο που άλλαζε ήταν η μάσκα, ο ένας τρόπος ήταν να τοποθετηθεί η οδηγία DIRECTIVES και

στους 8 βρόχους με αποτέλεσμα την παραγωγή 8 ζευγαριών fetch και process για την απεικόνιση της εφαρμογής σε υλικό, και ο άλλος ήταν η συγχώνευση των 8 βρόχων σε έναν με την παραγωγή διαφορετικών αποτελεσμάτων για κάθε μάσκα με αποτέλεσμα την παραγωγή ενός μόνο ζευγαριού fetch και process για την απεικόνιση της εφαρμογής σε υλικό. Ο συγχωνευμένος βρόχος είναι ο παρακάτω:

```
for(i=1; i<rows-1; i++)
{
     for(j=1; j<cols-1; j++)
     {
          sum0 = 0;
          sum1 = 0;
          sum2 = 0;
          sum3 = 0;
          sum4 = 0;
          sum5 = 0;
          sum6 = 0;
          sum7 = 0;

          for(a=-1; a<t-1; a++)
          {
               for(b=-1; b<t-1; b++)
               {
                    im = image[i+a][j+b];
                    sum0     =     sum0     +     im     *
     mask_0[a+1][b+1];
                    sum1     =     sum1     +     im     *
     mask_1[a+1][b+1];
                    sum2     =     sum2     +     im     *
     mask_2[a+1][b+1];
                    sum3     =     sum3     +     im     *
     mask_3[a+1][b+1];
                    sum4     =     sum4     +     im     *
     mask_4[a+1][b+1];
                    sum5     =     sum5     +     im     *
     mask_5[a+1][b+1];
                    sum6     =     sum6     +     im     *
     mask_6[a+1][b+1];
                    sum7     =     sum7     +     im     *
     mask_7[a+1][b+1];
               }
          }
          out_image0[i][j] = sum0;
          out_image1[i][j] = sum1;
          out_image2[i][j] = sum2;
          out_image3[i][j] = sum3;
          out_image4[i][j] = sum4;
          out_image5[i][j] = sum5;
          out_image6[i][j] = sum6;
          out_image7[i][j] = sum7;
```

```
	}
}
```

Εικόνα 51: Κώδικας Εφαρμογής 4 για απεικόνιση σε υλικό

Οι πράξεις της συνάρτησης process που παράχθηκαν για την πρώτη περίπτωση για κάθε βρόχο ήταν οι παρακάτω:

```
for(i=1; i<rows-1; i++)
{
	for(j=1; j<cols-1; j++)
	{
		var1 = 0;

		for(a=-1; a<t-1; a++)
		{
			for(b=-1; b<t-1; b++)
			{
				var2 = *val2;
				var3 = *val3;

				var1 = var1 + var2 * var3;

				val2++;
				val3++;
			}
		}

		*val1 = var1;
		val1++;
	}
}
```

Εικόνα 52: Πράξεις Εφαρμογής 3 (process)

Χρειάζονται συνολικά 5 FIFO για κάθε συνάρτηση. 2 τύπου uint64_t για τις διευθύνσεις και 3 για τα δεδομένα, ίδιου τύπου με τις αντίστοιχες μεταβλητές, int64_t στο συγκεκριμένο παράδειγμα. Οι FIFO των διευθύνσεων για την πρώτη συνάρτηση είναι οι address00 και address10. Οι μεταβλητές var2 και var3 αντιστοιχίζονται στις FIFO data0 και, data10. Η μεταβλητή var1, εφόσον είναι το τελικό αποτέλεσμα αντιστοιχίζεται στη FIFO result0. Στις υπόλοιπες συναρτήσεις ακολουθείται η ίδια ονοματολογία, το μόνο που αλλάζει είναι το τελευταίο ψηφίο, το οποίο ανάλογα με την συνάρτηση παίρνει τιμή από 0 έως 7. Για να μηδενίζεται όποτε πρέπει το αποτέλεσμα και για να αποθηκεύονται τα σωστά δεδομένα στη FIFO, η αρχικοποίηση και η αποθήκευση στη FIFO τοποθετήθηκαν χειροκίνητα εκτός των βρόχων επανάληψης που διαβάζουν τα στοιχεία της μάσκας για την εκτέλεση τη πράξης.

Οι πράξεις της συνάρτησης process που παράχθηκαν για την δεύτερη περίπτωση για κάθε βρόχο ήταν οι παρακάτω:

```
for(i=1; i<rows-1; i++)
{
     for(j=1; j<cols-1; j++)
     {
          var3 = 0;
          var5 = 0;
          var7 = 0;
          var9 = 0;
          var11 = 0;
          var13 = 0;
          var15 = 0;
          var17 = 0;

          for(a=-1; a<t-1; a++)
          {
               for(b=-1; b<t-1; b++)
               {
                    var1 = *val1;
                    var2 = *val2;
                    var4 = *val4;
                    var6 = *val6;
                    var8 = *val8;
                    var10 = *val10;
                    var12 = *val12;
                    var14 = *val14;
                    var16 = *val16;
                    var18 = *val18;

                    var1 = var2;

                    var3 = var3 + var2 * var4;
                    var5 = var5 + var2 * var6;
                    var7 = var7 + var2 * var8;
                    var9 = var9 + var2 * var10;
                    var11 = var11 + var2 * var12;
                    var13 = var13 + var2 * var14;
                    var15 = var15 + var2 * var16;
                    var17 = var17 + var2 * var18;

                    val1++;
                    val2++;
                    val4++;
                    val6++;
                    val8++;
                    val10++;
                    val12++;
                    val14++;
                    val16++;
                    val18++;
               }
```

```
            }
            *val3 = var3;
            *val5 = var5;
            *val7 = var7;
            *val9 = var9;
            *val11 = var11;
            *val13 = var13;
            *val15 = var15;
            *val17 = var17;

            val3++;
            val5++;
            val7++
            val9++;
            val11++;
            val13++;
            val15++;
            val17++;
      }
}
```

Εικόνα 53: Πράξεις Εφαρμογής 4 (process)

Για τη δεύτερη περίπτωση της εφαρμογής χρειάζονται συνολικά 26 FIFO. 9 τύπου uint64_t για τις διευθύνσεις και 17 για τα δεδομένα, ίδιου τύπου με τις αντίστοιχες μεταβλητές, int64_t στο συγκεκριμένο παράδειγμα. Οι FIFO των διευθύνσεων είναι οι address0 έως address8. Οι μεταβλητές var2, var4, var6, var8, var10, var12, var14, var16, var18 αντιστοιχίζονται στις fifo data0 έως data8. Η μεταβλητές var1, var33, var5, var7, var9, var11, var13, var15, var17 εφόσον είναι τα τελικά αποτέλεσματα αντιστοιχίζονται στις FIFO result0 έως result7. Όπως και στην πρώτη περίπτωση, για να μηδενίζεται όποτε πρέπει το αποτέλεσμα και για να αποθηκεύονται τα σωστά δεδομένα στη FIFO, η αρχικοποίηση και η αποθήκευση στη FIFO τοποθετήθηκαν χειροκίνητα εκτός των βρόχων επανάληψης που διαβάζουν τα στοιχεία της μάσκας για την εκτέλεση τη πράξης. Επίσης, δεν υπάρχει τρόπος να γνωρίζει το εργαλείο ότι επιστρέφονται ως αποτέλεσμα τα στοιχεία παραπάνω από μιας FIFO, οι αρχικοποιήσεις και επιστροφές όλων των αποτελεσμάτων τοποθετήθηκαν χειροκίνητα εκτός των απαιτούμενων βρόχων.

Η διαδικασία για την απεικόνιση σε υλικό της εφαρμογής εντός του test bench είναι και για τις δύο περιπτώσεις ίδια με τις προηγούμενες δύο εφαρμογές. Όπως και στις προηγούμενες εφαρμογές, εντοπίστηκαν όλα τα αποτελέσματα υλικού ίδια με τα αντίστοιχα του λογισμικού, οπότε επιβεβαιώνεται ότι η χρήση των συναρτήσεων fetch και process του υλικού παράγει ακριβώς τα ίδια αποτελέσματα με αυτά του κώδικα λογισμικού. Τελικό βήμα πριν την σταδιακή αύξηση του μεγέθους των πινάκων για να ληφθούν οι απαραίτητες μετρήσεις είναι ο καθορισμός του μεγέθους των FIFO, το οποίο ήταν 4096 όπως ορίστηκε αυθαίρετα από το εργαλείο. Μετά από δοκιμές καθορίστηκε το μεγαλύτερο δυνατό μέγεθος FIFO ως 32768. Οπότε, στο test

bench και στις συναρτήσεις fetch και process το μέγεθος των FIFO άλλαξε από 4096 σε 32768.

5.2.4 Ανάλυση αποτελεσμάτων

Πρώτο βήμα για την απεικόνιση σε υλικό είναι η σύνθεση των εφαρμογών. Πριν ξεκινήσει η διαδικασία αυτή, ήταν απαραίτητο να καθοριστεί η συχνότητα του ρολογιού κάθε εφαρμογής. Το Vivado HLS επιτρέπει στον προγραμματιστή να ρυθμίσει την περίοδο του ρολογιού. Έτσι, μειώνοντας τη συχνότητα πριν από κάθε σύνθεση παρέχεται η δυνατότητα της εύρεσης της χαμηλότερης δυνατής περιόδου. Έτσι, με διαδοχικές μειώσεις της, παρατηρήθηκε η παρακάτω βέλτιστη περίοδος ρολογιού και κατάλληλη συχνότητα για κάθε εφαρμογή.

	Περίοδος ρολογιού (ns)	Συχνότητα ρολογιού (MHz)
Εφαρμογή 1 - Πράξεις στοιχείων διδιάστατων πινάκων	3	330
Εφαρμογή 2 – Άθροισμα ανά γραμμή και ανά στήλη	1,5	660
Εφαρμογή 3 - Ανίχνευση ακμών (πρώτος τρόπος)	1,6	625
Εφαρμογή 4 - Ανίχνευση ακμών (δεύτερος τρόπος)	1,6	625

Εικόνα 54: Περίοδος και συχνότητα ρολογιού για κάθε εφαρμογή

Κατόπιν, όπως προαναφέρθηκε, για κάθε εφαρμογή επιλέχθηκαν ως κορυφαίες όλες οι fetch και process συναρτήσεις της, η μια μετά την άλλη, για να καταμετρηθούν οι πόροι της FPGA που καταλαμβάνονται κάθε φορά.

BRAM_18K	DSP48E	FF (Flip Flops)	LUT (LookUp Tables)
7560	1800	2148480	1074240

Εικόνα 55: Διαθέσιμοι πόροι της FPGA της Virtex UltraScale XCVU190-FLGB2104-1-I

Ακολουθούν 4 πίνακες, ένας για κάθε εφαρμογή, οι οποίοι περιέχουν τους πόρους κάθε κορυφαίας συνάρτησης.

Top Function	BRAM_18K	DSP48E	FF	LUT
fetchUnit0	-	26	3161	1106
processUnit0	-	45	3538	1076

Εικόνα 56: Πόροι Εφαρμογής 1: Πράξεις στοιχείων διδιάστατων πινάκων

Top Function	BRAM_18K	DSP48E	FF	LUT
fetchUnit0	-	-	1993	533
processUnit0	-	-	1698	493

fetchUnit1	-	-	2037	539
processUnit1	-	-	1698	484

Εικόνα 57: Πόροι Εφαρμογής 2: Άθροισμα ανά γραμμή και ανά στήλη

Top Function	BRAM_18K	DSP48E	FF	LUT
fetchUnit0	-	32	4705	2066
processUnit0	-	16	3154	1252
fetchUnit1	-	32	4705	2066
processUnit1	-	16	3154	1252
fetchUnit2	-	32	4705	2066
processUnit2	-	16	3154	1252
fetchUnit3	-	32	4705	2066
processUnit3	-	16	3154	1252
fetchUnit4	-	32	4705	2066
processUnit4	-	16	3154	1252
fetchUnit5	-	32	4705	2066
processUnit5	-	16	3154	1252
fetchUnit6	-	32	4705	2066
processUnit6	-	16	3154	1252
fetchUnit7	-	32	4705	2066
processUnit7	-	16	3154	1252

Εικόνα 58: Πόροι Εφαρμογής 3: Ανίχνευση ακμών (πρώτος τρόπος)

Top Function	BRAM_18K	DSP48E	FF	LUT
fetchUnit0	-	32	6532	2619
processUnit0	-	128	12737	4738

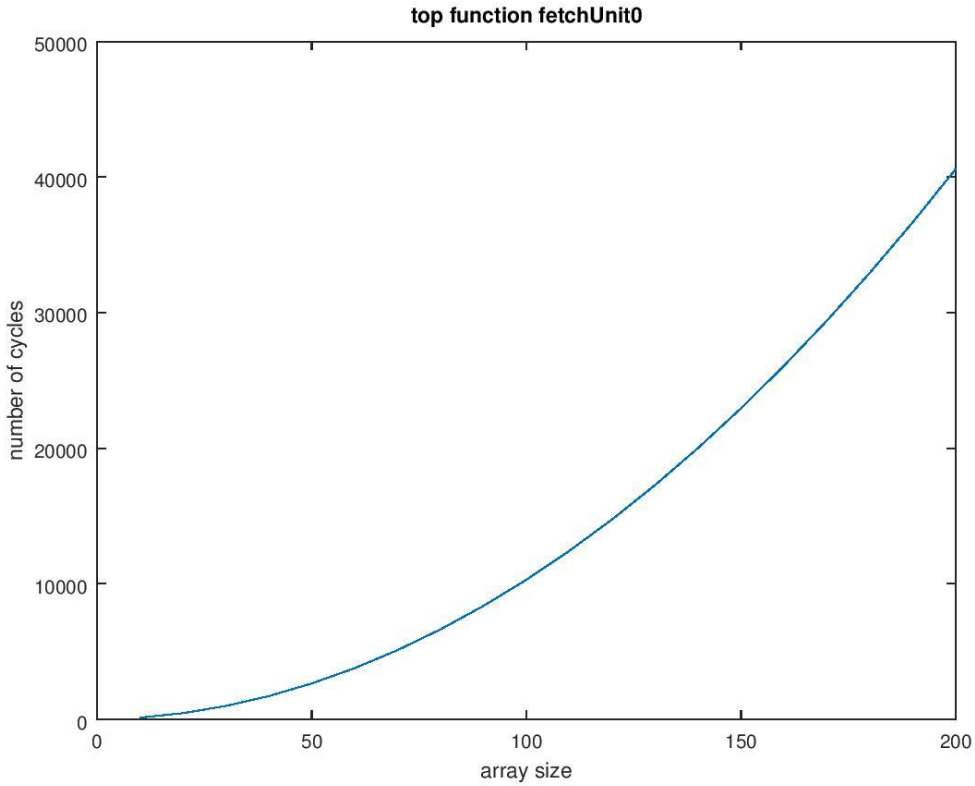
Εικόνα 59: Πόροι Εφαρμογής 4: Ανίχνευση ακμών (δεύτερος τρόπος)

Τελευταία διαδικασία ήταν η κοινή προσομοίωση C/RTL, κατά την οποία για κάθε μια από της συναρτήσεις fetch και process κάθε εφαρμογής μετρήθηκε ο αριθμός των κύκλων που απαιτείται για την εκτέλεση της εφαρμογής. Το Vivado HLS δεν παράγει τα κατάλληλα αρχεία για απεικόνιση εφαρμογής στην FPGA. Όμως, κατά τη διαδικασία αυτή, υλοποιεί πλήρη προσομοίωση της απεικόνισης αυτής και εξάγει εκ νέου τα αποτελέσματα του κώδικα στην κονσόλα, επιβεβαιώνοντας την επιτυχή ολοκλήρωση της διαδικασίας του place & route με τη χρήση άλλου λογισμικού. Για κάθε συνάρτηση ακολουθήθηκε η ίδια διαδικασία. Το μέγεθος των πινάκων κάθε πράξης αυξανόταν κάθε φορά κατά 10 και πραγματοποιούνταν απεικόνιση σε υλικό μέχρι να βρεθεί το μέγιστο δυνατό μέγεθος κάθε εφαρμογής και σημειωνόταν ο αριθμός των κύκλων που απαιτήθηκε για να εκτελεστεί κάθε εφαρμογή. Σε κάθε εφαρμογή, για το ίδιο μέγεθος πινάκων, παρατηρήθηκαν δύο διαφορετικοί αριθμοί κύκλων, ένας για όλες τις συναρτήσεις fetch και ένας για όλες τις συναρτήσεις process. Παρακάτω παρατίθεται οι αντίστοιχες μετρήσεις σε πίνακες και διαγράμματα για κάθε εφαρμογή ξεχωριστά.

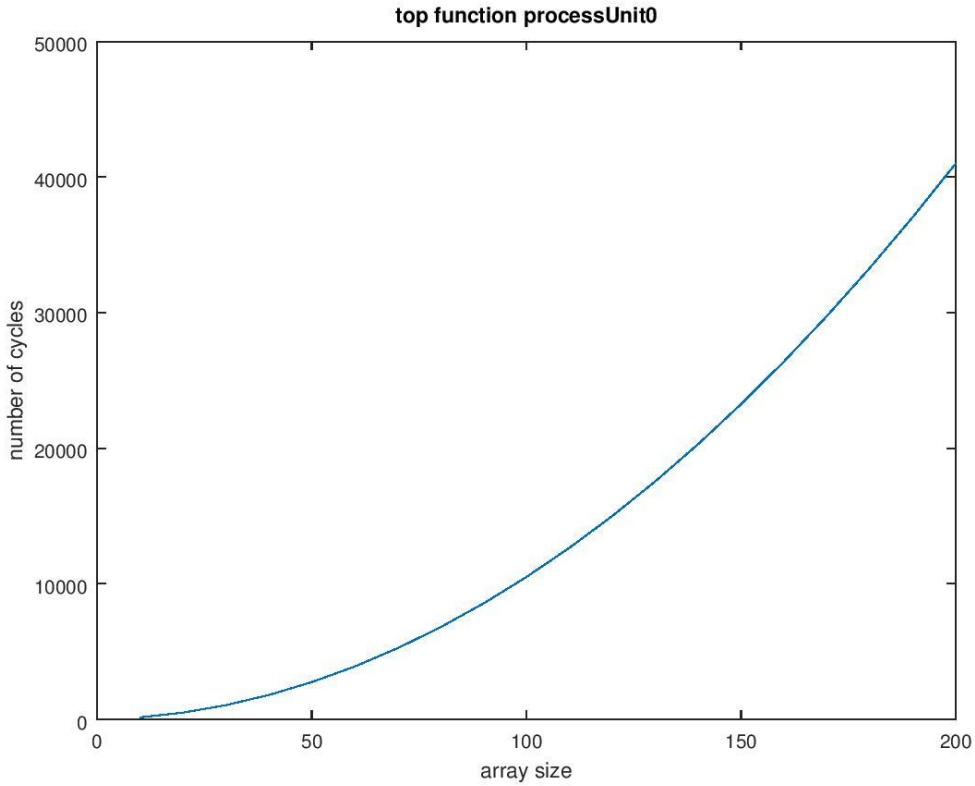
Εφαρμογή 1 - Πράξεις στοιχείων διδιάστατων πινάκων:

Μέγεθος πινάκων (θέσεις)	Κορυφαία συνάρτηση fetch (n κύκλων)	Κορυφαία συνάρτηση process (n κύκλων)
10	131	151
20	461	501
30	991	1051
40	1721	1801
50	2651	2751
60	3781	3901
70	5111	5251
80	6641	6801
90	8371	8551
100	10301	10501
110	12431	12651
120	14761	15001
130	17291	17551
140	20021	20301
150	22951	23251
160	26081	26401
170	29411	29751
180	32941	33301
190	36671	37051
200	40601	41001

Εικόνα 60: Αριθμός κύκλων ανά συνάρτηση Εφαρμογής 1



Εικόνα 61: Συνάρτηση fetch Εφαρμογής 1: αριθμός κύκλων ανά μέγεθος πίνακα

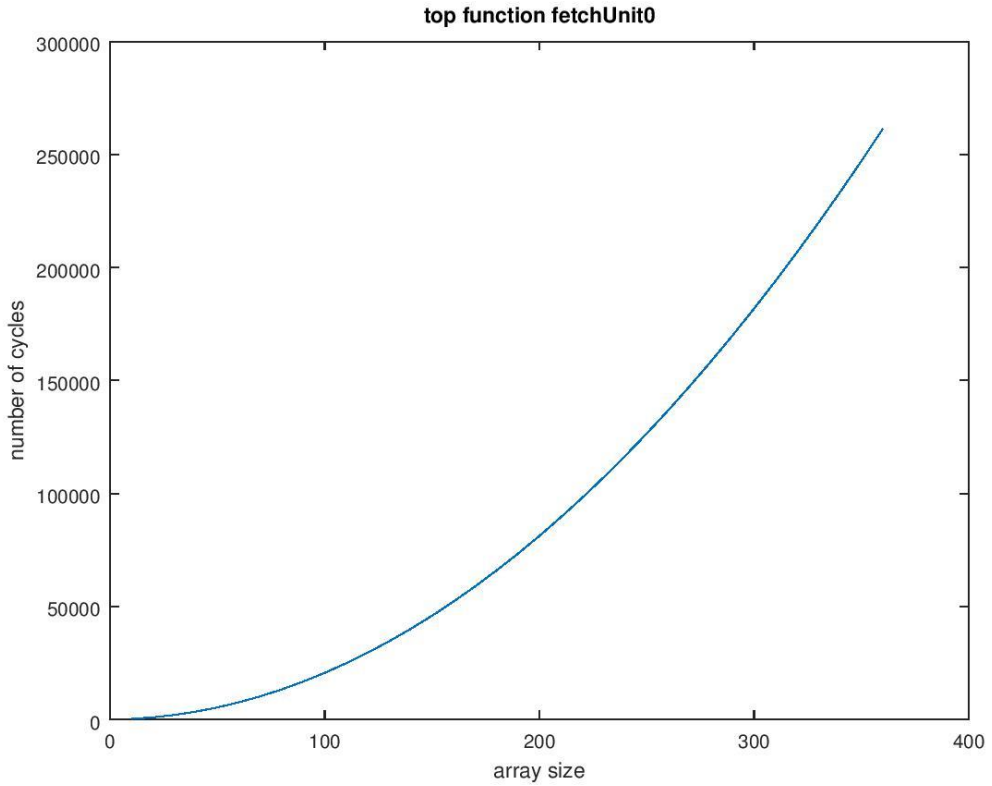


Εικόνα 62: Συνάρτηση process Εφαρμογής 1: αριθμός κύκλων ανά μέγεθος πίνακα

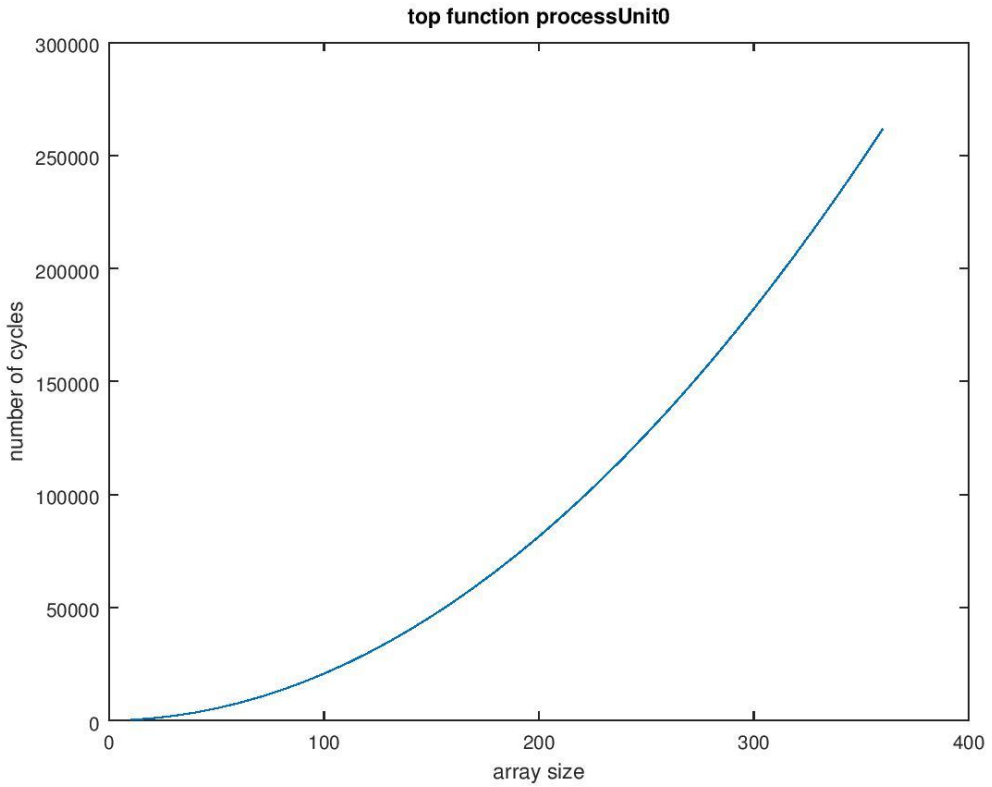
Εφαρμογή 2 - Άθροισμα ανά γραμμή και ανά στήλη:

Μέγεθος πινάκων (θέσεις)	Κορυφαία συνάρτηση fetch (n κύκλων)	Κορυφαία συνάρτηση process (n κύκλων)
10	262	272
20	922	942
30	1982	2012
40	3442	3482
50	5302	5352
60	7562	7622
70	10222	10292
80	13282	13362
90	16742	16832
100	20602	20702
110	24862	24972
120	29522	29642
130	34582	34712
140	40042	40182
150	45902	46052
160	52162	52322
170	58822	58992
180	65882	66062
190	73342	73532
200	81202	81402
210	89462	89672
220	98122	98342
230	107182	107412
240	116642	116882
250	126502	126752
260	136762	137022
270	147422	147692
280	158482	158762
290	169942	170232
300	181802	182102
310	194062	194372
320	206722	207042
330	219782	220112
340	233242	233582
350	247102	247452
360	261362	261722

Εικόνα 63: Αριθμός κύκλων ανά συνάρτηση Εφαρμογής 2



Εικόνα 64: Συνάρτηση fetch Εφαρμογής 2: αριθμός κύκλων ανά μέγεθος πίνακα

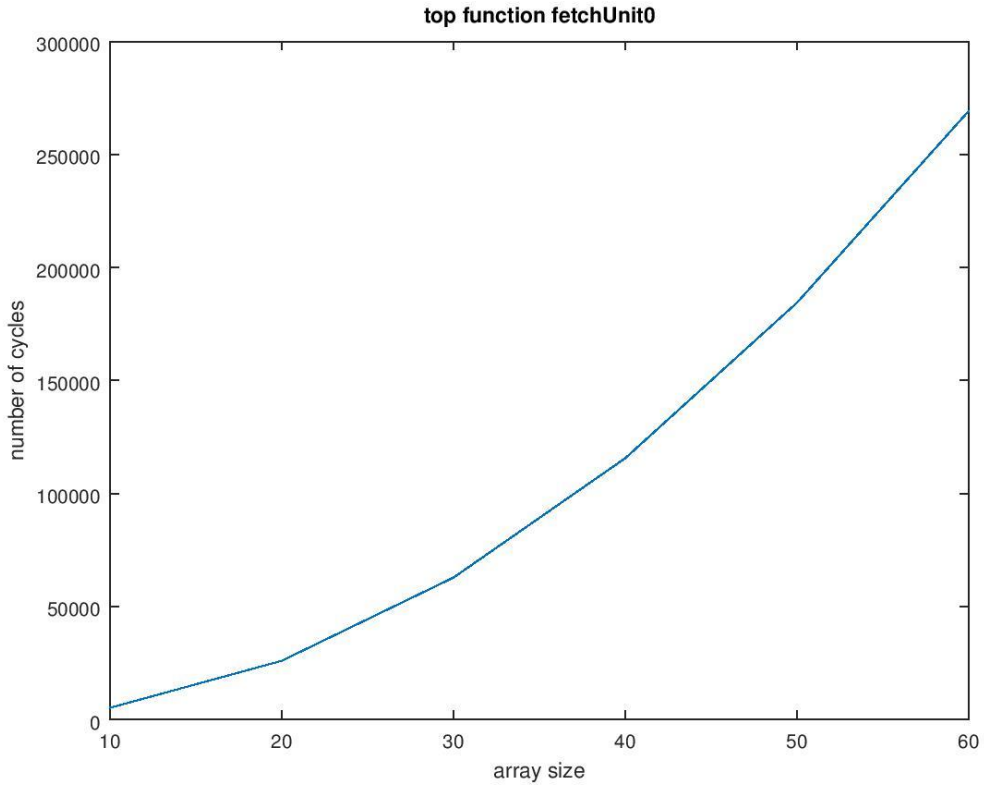


Εικόνα 65: Συνάρτηση process Εφαρμογής 2: αριθμός κύκλων ανά μέγεθος πίνακα

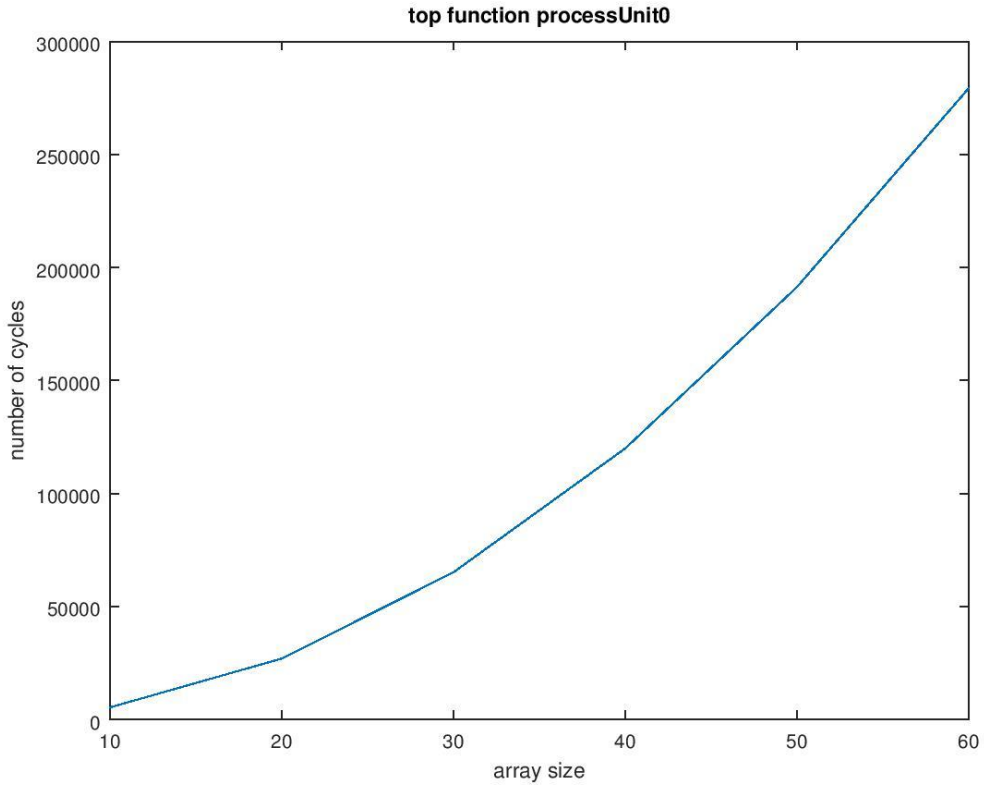
Εφαρμογές 3 και 4 - Ανίχνευση ακμών με δυο τρόπους:

Μέγεθος πινάκων (θέσεις)	Κορυφαία συνάρτηση fetch (n κύκλων)	Κορυφαία συνάρτηση process (n κύκλων)
10	5137	5329
20	25957	26929
30	62777	65129
40	115597	119929
50	184417	191329
60	269237	279329

Εικόνα 66: Αριθμός κύκλων ανά συνάρτηση Εφαρμογών 3 και 4



Εικόνα 67: Συνάρτηση fetch Εφαρμογών 3 και 4: αριθμός κύκλων ανά μέγεθος πίνακα



Εικόνα 68: Συνάρτηση process Εφαρμογών 3 και 4: αριθμός κύκλων ανά μέγεθος πίνακα

ΚΕΦΑΛΑΙΟ 6-ΣΥΜΠΕΡΑΣΜΑΤΑ ΚΑΙ ΜΕΛΛΟΝΤΙΚΗ ΕΡΓΑΣΙΑ

6.1 Συμπεράσματα

Στην παρούσα διπλωματική παρουσιάστηκε εργαλείο για απεικόνιση αλγορίθμων σε αναδιατασσόμενη λογική. Οι εφαρμογές που απεικονίστηκαν περιείχαν πράξεις πινάκων ή αλγόριθμους επεξεργασίας εικόνας. Όλες οι εφαρμογές πέρασαν από το ίδιο DAE framework με πολύ καλά αποτελέσματα. Οι συναρτήσεις fetch και process, οι οποίες τρέχουν παράλληλα χωρίς αναμονή δεδομένων βελτιώνοντας έτσι το συνολικό χρόνο εκτέλεσης. Επιπλέον, με τη χρήση του εργαλείου αυτού ελαχιστοποιείται ο φόρτος του προγραμματιστή, καθώς τα αρχεία των δύο συναρτήσεων παράγονται αυτόματα.

Για την εκτέλεση των εφαρμογών επιλέχθηκε το Vivado HLS, ένα μοντέρνο εργαλείο σύνθεσης υλικού με ποικίλα πλεονεκτήματα. Δέχεται κώδικα εισόδου σε γλώσσα προγραμματισμού υψηλού επιπέδου, με αποτέλεσμα να μπορεί να χρησιμοποιηθεί και από προγραμματιστές λογισμικού χωρίς να είναι απαραίτητη η γνώση κάποιας γλώσσας περιγραφής υλικού, παράγει λεπτομερή αναφορά αποτελεσμάτων και έχει τη δυνατότητα βελτιστοποίησης μέσω οδηγιών.

6.2 Μελλοντική εργασία

Το εργαλείο που υλοποιήθηκε στα πλαίσια αυτής της διπλωματικής δέχεται αρκετές και ποικίλες, όπως περιεγράφηκε σε παραπάνω κεφάλαια. Υπάρχει η δυνατότητα να συνεχισθεί η επεξεργασία του με σκοπό τη διάσπαση και εκτέλεση σε υλικό ενός ευρύτερου φάσματος εφαρμογών. Για να επιτευχθεί αυτό είναι απαραίτητη η αναγνώριση και επεξεργασία επιπλέον πράξεων που δεν αναγνωρίζονται αυτή τη στιγμή από το εργαλείο, για παράδειγμα τελεστές που αλλάζουν την τιμή μεταβλητής κατά 1 (++, --) και λογικούς τελεστές (&&, ||, !). Επιπλέον, δύναται να επεκταθεί έτσι ώστε να αναγνωρίζει εντός της οδηγίας επιπλέον εντολές, όπως επιπλέον βρόχους επανάληψης (while, do) και διαφορετικές μορφές τους, εντολές αποφάσεων (if) ή περιπτώσεων (case).

Μια άλλη λειτουργικότητα που θα μπορούσε μελλοντικά να υποστηρίζει το εργαλείο είναι η δημιουργία ενός επιπλέον αρχείου. Το αρχείο αυτό θα μπορεί να εμπεριέχει τις τροποποιήσεις στον κώδικα εισόδου στο Vivado HLS που απαιτείται να γραφούν στην παρούσα φάση από τον προγραμματιστή. Δηλαδή τον κώδικα που λαμβάνει τις διευθύνσεις από τη συνάρτηση fetch, τις αποθηκεύει σε FIFO, αντιστοιχίζει τις διευθύνσεις στα δεδομένα τους και δίνει τις κατάλληλες FIFO δεδομένων ως είσοδο στη συνάρτηση process.

ΒΙΒΛΙΟΓΡΑΦΙΑ

1) L. Séméria, K. Sato, G. and De Micheli. (2002). Synthesis of hardware models in C with pointers and complex data structures. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on. 9. 743 - 756. 10.1109/92.974889.
2) A. Ghosh, J. Kunkel and S. Liao, "Hardware synthesis from C/C++," Design, Automation and Test in Europe Conference and Exhibition, 1999. Proceedings (Cat. No. PR00078), Munich, Germany, 1999, pp. 387-389. 10.1109/DATE.1999.761152
3) F. Haim, M. Sen, D.-I. Ko, S. S. Bhattacharyya and W. Wolf, "Mapping Multimedia Applications Onto Configurable Hardware With Parameterized Cyclo-Static Dataflow Graphs," 2006 IEEE International Conference on Acoustics Speech and Signal Processing Proceedings, Toulouse, 2006, pp. III-III. 10.1109/ICASSP.2006.1660838
4) M. D. Galanis, G. Dimitroulakos and C.E. Goutis (2005). Accelerating applications by mapping critical kernels on coarse-grain reconfigurable hardware in hybrid systems. 301 - 302. 10.1109/FCCM.2005.15.
5) M. Huang, V. K. Narayana and T. El-Ghazawi. (2009). Efficient Mapping of Hardware Tasks on Reconfigurable Computers Using Libraries of Architecture Variants. Proceedings - IEEE Symposium on Field Programmable Custom Computing Machines, FCCM 2009. 247-250. 10.1109/FCCM.2009.20.
6) M. Besana, M. Borgatti. (2003). Application mapping to a hardware platform through automated code generation targeting a RTOS: a design case study. 41-44 suppl.. 10.1109/DATE.2003.1186669.
7) K. Wakabayashi. (1999). C-based synthesis experiences with a behavior synthesizer, Cyber. 390 - 393. 10.1109/DATE.1999.761153.
8) Y. Hemaraj, M. Sen, R. Shekhar and S. S. Bhattacharyya, "Model-Based Mapping of Image Registration Applications onto Configurable Hardware," 2006 Fortieth Asilomar Conference on Signals, Systems and Computers, Pacific Grove, CA, 2006, pp. 1453-1457. 10.1109/ACSSC.2006.354999
9) M. Peercy, J. Airey and B. Cabral. (1970). Efficient Bump Mapping Hardware. Proceedings of the ACM SIGGRAPH Conference on Computer Graphics. 10.1145/258734.258873.
10) P. Balaji, R. Gupta, A. Vishnu et al. Comput Sci Res Dev. (2011). Mapping communication layouts to network hardware characteristics on massive-scale blue gene systems. Computer Science - Research and Development. Volume 26 Issue 3-4, June 2011 Pages 247-256
11) D. C. Suresh, W. A. Najjar F. Vahid, J. R. Villarreal and G. Stitt. (2003). Profiling tools for hardware/software partitioning of embedded applications. Sigplan Notices - SIGPLAN. 38. 189-198. 10.1145/780731.780759.

12) X. Fang, P. Thole, J. Göppert and W. Rosenstiel. (1996). A hardware supported system for a special online application of self-organizing map. 956 - 961 vol.2. 10.1109/ICNN.1996.549026.
13) C. Trendall, J. Stewart. (2001). General Calculations Using Graphics Hardware, with application to interactive caustics. Eurographics Rendering Workshop. 10.1007/978-3-7091-6303-0_26.
14) M. B. Srivastava, J. S. Sun and R. W. Brodersen, "Hardware and software prototyping for application-specific real-time systems," [1991 Proceedings] The Second International Workshop on Rapid System Prototyping, Research Triangle Park, NC, USA, 1991, pp. 101-102. 10.1109/IWRSP.1991.218620
15) S. Liao, S. Tjiang and R. Gupta, "An Efficient Implementation Of Reactivity For Modeling Hardware In The Scenic Design Environment," *Proceedings of the 34th Design Automation Conference*, Anaheim, CA, USA, 1997, pp. 70-75.
16) L. M. AlBarakat, P. V. Gratz and D. A. Jiménez, "MTB-Fetch: Multithreading Aware Hardware Prefetching for Chip Multiprocessors," in *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 175-178, 1 July-Dec. 2018.
17) G. Charitopoulos, C. Vatsolakis, G. Chrysos, D. Pnevmatikatos (2018). A decoupled access-execute architecture for reconfigurable accelerators. 244-247. 10.1145/3203217.3203267.
18) S. Cheng and J. Wawrzynek, "Architectural synthesis of computational pipelines with decoupled memory access," 2014 International Conference on Field-Programmable Technology (FPT), Shanghai, 2014, pp. 83-90.
19) S. Chung, Eric & C. Hoe, James & Mai, Ken. (2011). CoRAM: An in-fabric memory architecture for FPGA-based computing. 97-106. 10.1145/1950413.1950435.
20) C. Ho, S. J. Kim and K. Sankaralingam, "Efficient execution of memory access phases using dataflow specialization," *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, Portland, OR, 2015, pp. 118-130. doi: 10.1145/2749469.2750390
21) R. Zhao, G. Liu, S. Srinath, C. Batten and Z. Zhang, "Improving high-level synthesis with decoupled data structure optimization," 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, 2016, pp. 1-6.doi: 10.1145/2897937.2898030
22) K. Koukos, P. Ekemark, Per, G. Zacharopoulos, V. Spiliopoulos, Vasileios S. Kaxiras, A. Jimborean. (2016). Multiversioned decoupled access-execute: the key to energy-efficient compilation of general-purpose programs. 10.1145/2892208.2892209.
23) https://github.com/SilverScar/C-Language-Parser
24) Δ. Σπινέλλης, Τμήμα Διοικητικής Επιστήμης και Τεχνολογίας, Οικονομικό Πανεπιστήμιο Αθηνών
25) H. R. Lewis, X. X. Παπαδημητρίου, Στοιχεία Θεωρίας Υπολογισμού, Εκδόσεις Κριτική, ISBN 978-960-218-397-7, Σελίδα 223
26) http://ftp.mozgan.me/Compiler_Manuals/LexAndYaccTutorial.pdf
27) http://dinosaur.compilertools.net/lex/index.html

28) http://dinosaur.compilertools.net/yacc/index.html
29) Image Processing in C by Dwayne Phillips Second Edition Published 1994 ISBN-10: 0-13-104548-2
30) https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/sxx1504034358866.html
31) https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/jit1504034365862.html
32) https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/rzx1504034366923.html
33) https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/fde1504034360078.html
34) https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/uyd1504034366571.html
35) https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_2/ug902-vivado-high-level-synthesis.pdf
36) https://www.w3resource.com/c-programming-exercises/array/c-array-exercise-25.php
37) D. Wijerathne, Z. Li, M. Karunaratne, A. Pathania, T. Mitra (2019). CASCADE: High Throughput Data Streaming via Decoupled Access-Execute CGRA. ACM Transactions on Embedded Computing Systems. 18. 1-26. 10.1145/3358177.
38) R. Sun, P. Liu, J. Xue, S. Yang, J. Qian and R. Ying, "BAX: A Bundle Adjustment Accelerator With Decoupled Access/Execute Architecture for Visual Odometry," in IEEE Access, vol. 8, pp. 75530-75542, 2020, doi: 10.1109/ACCESS.2020.2988527.
39) http://purl.tuc.gr/dl/dias/5E7B75A7-45A5-44B8-A300-DC1BB45AB939
40) J. E. Smith. 1982. Decoupled access/execute computer architectures. In Proceedings of the 9th annual symposium on Computer Architecture (ISCA '82). IEEE Computer Society Press, Washington, DC, USA, 112–119.
41) https://www.tutorialspoint.com/microprocessor/microprocessor_8086_overview.htm
42) IBM System Blue Gene Solution: Blue Gene/P Application Development