5η Εργαστηριακή ΄Ασκηση

Κοτσιµπού Φωτεινή ΑΜ 1059567 Σιγούρου ΄Αλκηστις-Αικατερίνη ΑΜ 1059661

7 Ιουνίου 2021



Εργασία στα πλαίσια του µαθήµατος Σχεδιασµός Συστηµάτων VLSI

Τµήµα Μηχ. Ηλεκτρονικών Υπολογιστών και Πληροφορικής

Περιεχόμενα

1. [Ενότητα Α](#_bookmark0) 1
   1. [Element-wise Multiplexer - VHDL](#_bookmark1) 1
   2. [Element-wise Multiplexer - C++Synthesis](#_bookmark7) 3
      1. [VHDL Comparison](#_bookmark11) 5
   3. [Pragmas Synthesis](#_bookmark12) 5
      1. [VHDL Comparison](#_bookmark13) 5
2. [Ενότητα Β](#_bookmark15) 7
   1. [Matrix Multiplexer - C++Synthesis](#_bookmark16) 7
   2. [Matrix Multiplexer - C++ Pragma Synthesis](#_bookmark16) 8
3. [Υπόμνημα](#_bookmark22) **10**

# Ενότητα Α

## Element-wise Multiplexer - VHDL

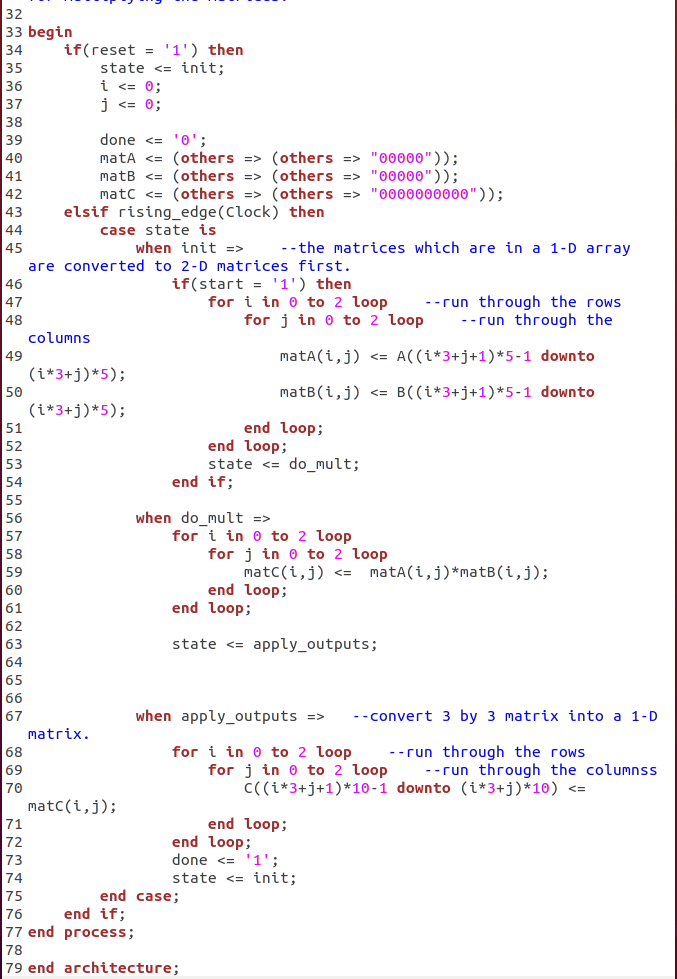
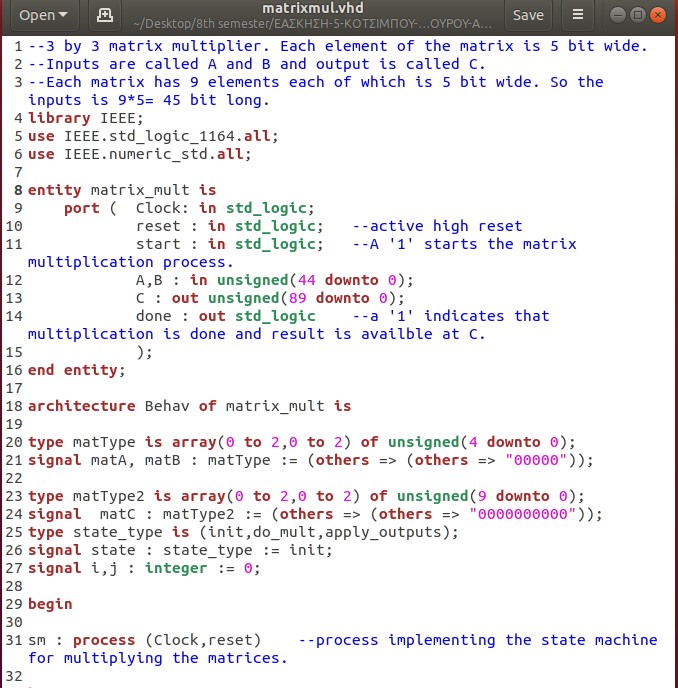
Στο πρώτο ερώτημα κληθήκαµε να δηµιουργήσουµε έναν κώδικα σε VHDL στον οποίο να εκετελέιται ο element-wise πολλαπλασιασµός 2 µητρώων 3x3, 5 bit κάθε στοιχείο.

Αρχικά ορίζουµε την οντότητα matrix mult , η οποία αποτελείται απο τα σήµατα clock, reset, start και done.Επίσης ορίζουµε 3 στοιχεία (A,B,C) τα οποία θα χρησιµοποιήσουµε ως εισόδους για τους πίνακες µας. Για να το πετύχουµε αυτό υπολογίζουµε το µέγεθος του πολλαπλασιάζοντας το πλήθος των στοιχείων των πινάκων µας (3\*3=9) µε το µήκος της κάθε λέξης που είναι 5 bit, έτσι καταλήγουµε οτι θέλουµε εισόδους µήκους 45 bit και έξοδο 90 bit (µετα τον πολλαπλασιασµό).

Στην συνέχεια ορίζουµε την αρχιτεκτονική της συναρτησης µας. Ορίζουµε τους τύπους των

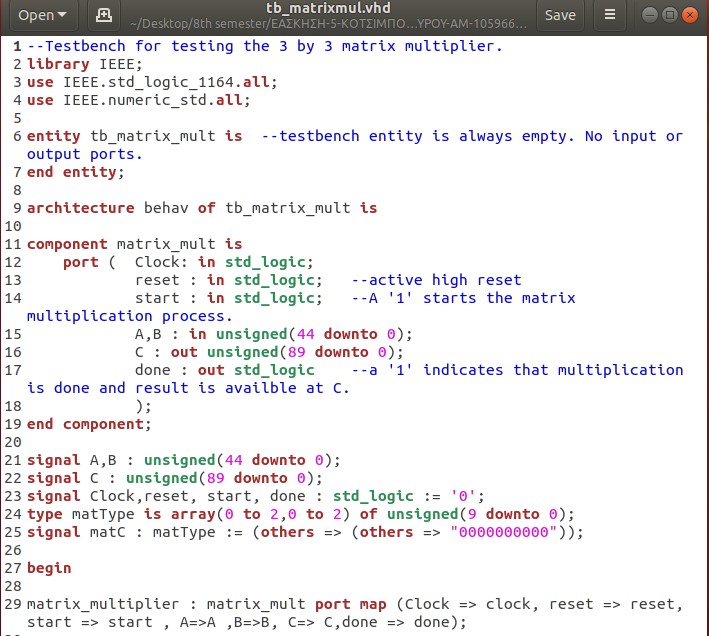
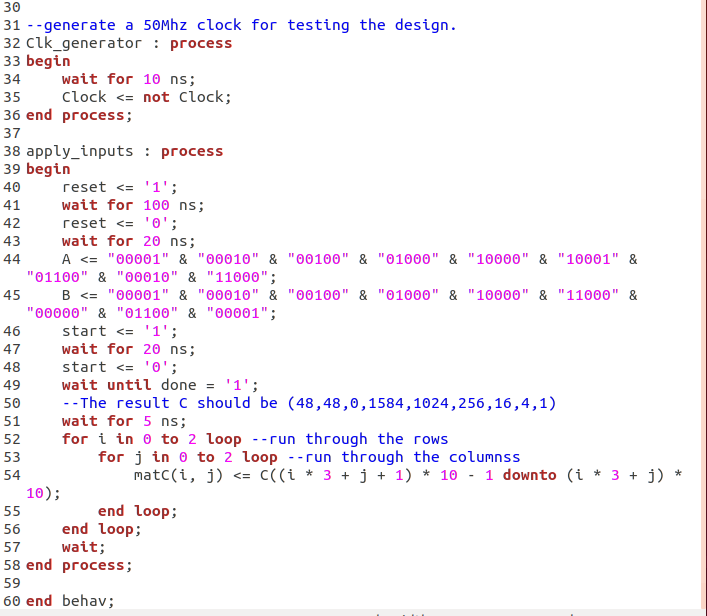
πινάκων που χρειαζόµαστε και το µεγεθός τους, ενώ παράλληλα τους αρχικοποιούµε µε µηδενικά. Ακόµα ορίζουµε τις 3 φάσεις της συνάρτησης µας (init: αρχικοποίηση πινάκων, do mult: η πράξη του πολλαπλασιασµού, apply outputs: αποθήκευεση των στοιχείων στην έξοδο C ).

Τέλος, µέσω τις εντολής case ορίζουµε τι θα εκτελεστεί σε κάθε φάση απο τις προαναφερθέντες. Ο κώδικας παρατίθεται στο Σχήμα [1.](#_bookmark2)



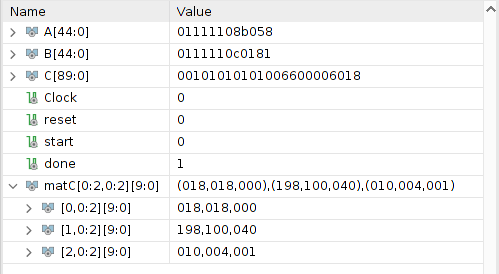
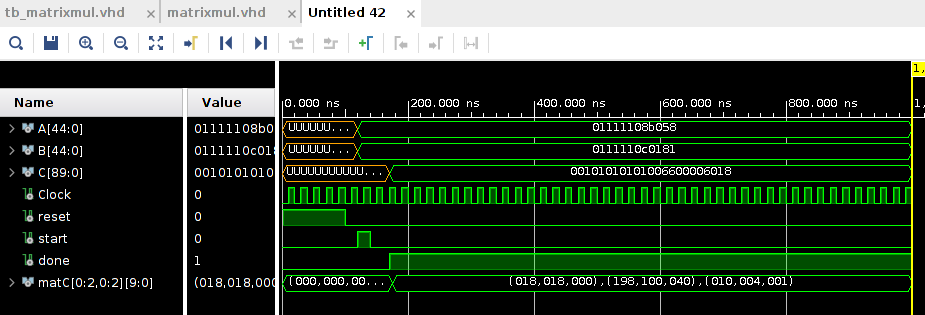
Σχήµα 1: Element Wise VHDL code

Προκειµένου να είναι εκτελέσιµη η συνάρτηση µας, συνοδέυουµε το αρχικό αρχείο κώδικα µε ένα testbench . Στον κώδικα, εδώ αρχικοποιούµε τα vectors Α, Β µε κάποιες τιµές µέσα στο process , ορίζουµε ρολόι και συχνότητα εκτέλεσης κάθε εντολής. Ο κώδικας για το testbench παρατίθεται στο Σχήμα [2.](#_bookmark3)

Σχήμα 2: Element Wise VHDL Testbench code

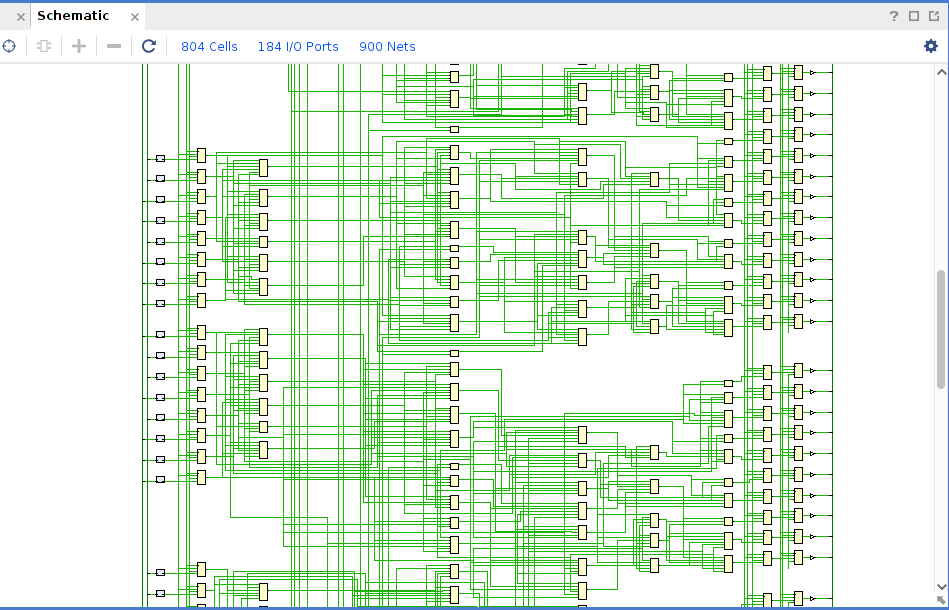
΄Υστερα εκτελούµε simulation , τα αποτελέσµατα φαίνονται στο Σχήμα [3,](#_bookmark4) Στο Σχήμα [4](#_bookmark6) βλέπουµε το σχήµατικό. Ενώ κάποιες υποδειγµατικές µετρήσεις παρουσιάζονται στον πίνακα [1](#_bookmark5)

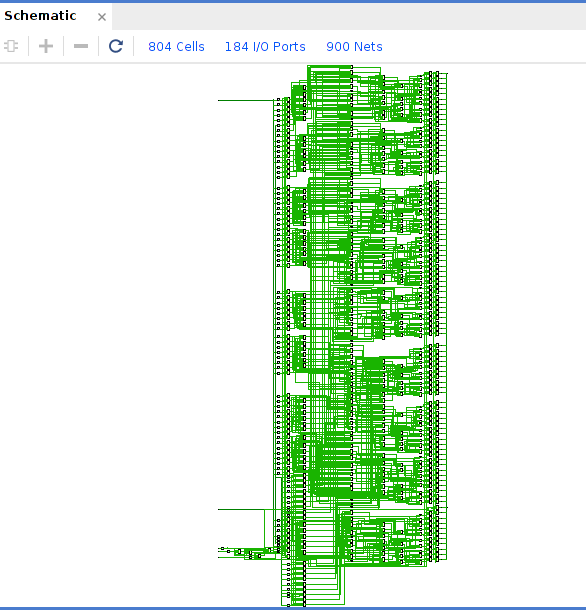


Σχήµα 3: Element Wise VHDL Simulation Results

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| File Code | Part | % Used IOB | % Used LUT | Total nets | Total Power |
| matrix*mult.vhd* | xczu3eg-sfvc784-2LV-e | 73,02% | 0.37% | 900 | 64.922W |

Πίνακας 1: Ενδεικτικά αποτελέσµατα Σύνθεσης





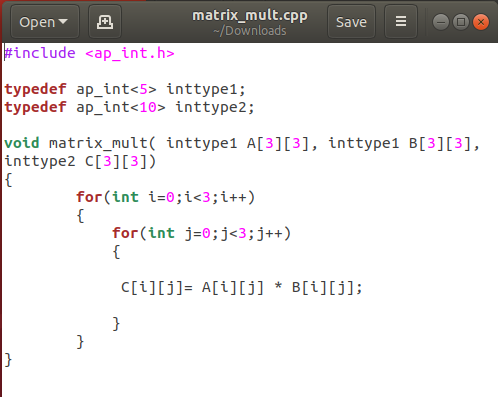
Σχήµα 4: Element Wise VHDL Schematic

Για λεπτοµερή αναφορά των αποτελεσµάτων της σύνθεσης, µπορείτε να µεταβείται στο spreadsheet µε όνοµα SyntesisReport.xlsx, που θα βρείτε στο συµπιεσµένο αρχείο που παραδώσαµε.

## Element-wise Multiplexer - C++Synthesis

Σε αυτό το ερώτηµα, υλοποιούµε πάλι Element-wise Multiplexer σε C++ . Οι κώδικες παρατίθενται στα Σχήματα [5](#_bookmark8) και [6.](#_bookmark9)

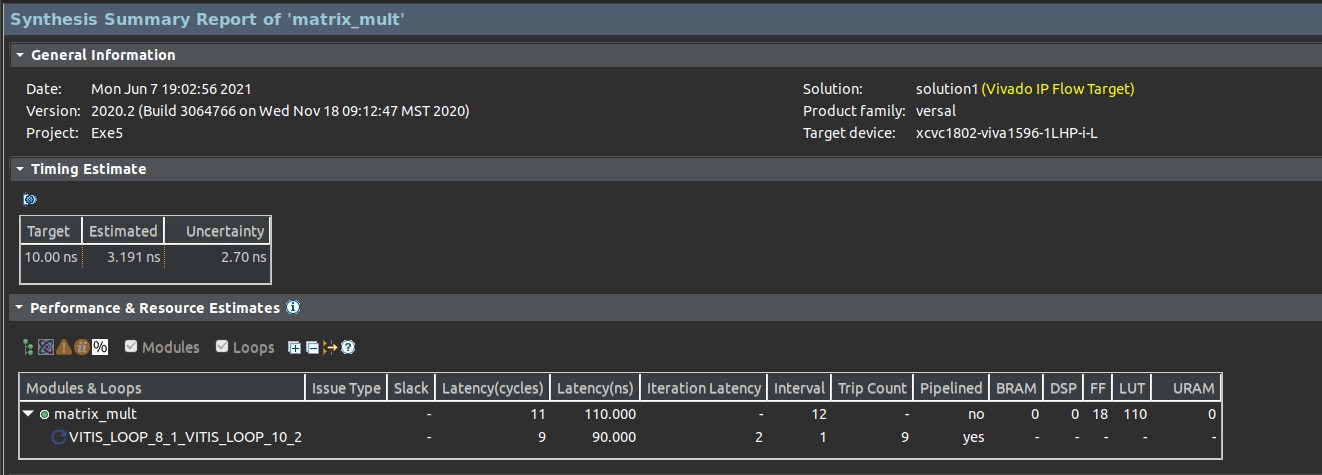
΄Υστερα εκτελούµε synthesis , τα αποτελέσµατα φαίνονται στο Σχήμα [7.](#_bookmark10)



Σχήµα 5: Element Wise C++ code



Σχήµα 6: Element Wise C++ Testbench code



Σχήµα 7: Synthesis Results

### VHDL Comparison

Από την C++ Synthesis προκύπτει ενα αρχείο vhdl (vitis matrix mult.vhd). Συγκρίνοντας τον κώδικα αυτόν µε τον δικό µας, παρατηρούµε ότι ο παραγόµενος κώδικας είναι πολύ πιο σύνθετος. Συγκεκριµένα, παρατηρούµε πως ο παραγόµενος κώδικας δηµιουργεί ένα πλήθος από διαφορετικά port, signals και constants για τη κατασκευή ενός κώδικα που όµως έχει την ίδια λειτουργικότητα µε τον δικό µας. Η δήλωση αυτών των µεταβλητων αναµένεται να δεσµεύσει ένα µεγάλο µέρος πόρων που όµως δεν είναι απαραίτητοι. Εκτός από όλες τις επιπλέον δηλώσεις το σύστηµα φυσικά χρησιµοποιεί όλες τις µεταβλητές που δηλώσαµε στον δικό µας VHDL και C κώδικα όπως clock, reset, start και done .

## Pragmas Synthesis

Στην συνέχεια προσθέτουµε στον αρχικό µας κώδικα pragmas µε την προοπτική να µειώσουµε το

latency της εκτέλεσης του. Συγκεκριµένα χρησιµοποιήσαµε τις εντολές:

”#pragma HLS array partition variable=A complete” η οποία χρησιµοποιήθηκε 3 φορές, µία για κάθε πίνακα που δηµιουργήσαµε στον κυρίως µας κώδικα και

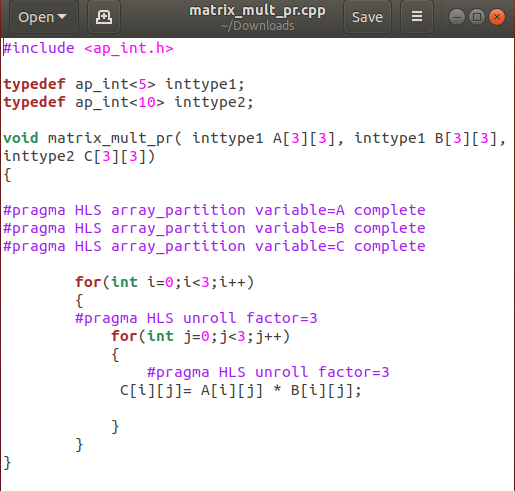
”#pragma HLS unroll factor=4” η οποία χρησιµοποιήθηκε 2 φόρες, µία για κάθε βρόγχο for που είχαµε στον κυρίως κώδικά µας.

Με τη χρήση της πρώτης εντολής καταφέρνει το σύστηµα να χωρίσει τη µνήµη και τους πίνακες σε µικρότερα τµήµατα (ή και µεµονωµένα στοιχεία όπως στη δική µας περίπτωση εξαιτίας της λέξης complete ), κάνοντας την επεξεργασία τους δυνατή από πολλά διαφορετικά τµήµατα της µνήµης που προκύπτει. Με τη δεύτερη εντολή επιτρέπεται στο σύστηµα να πραγµατοποιήσει µερικές (ή και όλες στη δική µας περίπτωση καθώς το factor έχει τιµή όση και οι επαναλήψεις του βρόγχου) από τις επαναλήψεις του κάθε βρόγχου παράλληλα. Αυτό πραγµατοποιείται µέσα από τα πολλαπλά αντίγραφα του κάθε βρόγχου που δηµιουργεί η καθε συγκεκριµένη εντολή pragmas που χρησιµοποιούµε.

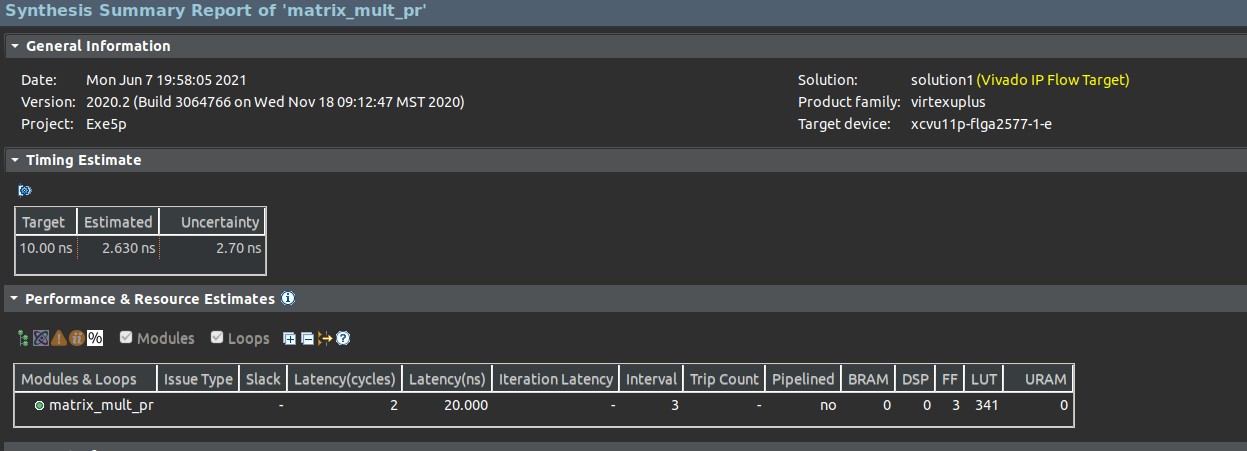
Τα αποτελέσµατα του synthesis , φαίνονται στο Σχήμα [9.](#_bookmark14)

### VHDL Comparison

Από την C++ Synthesis προκύπτει ενα αρχείο vhdl (vitis matrix mult pr.vhd). Συγκρίνοντας τον κώδικα αυτόν µε τον δικό µας, παρατηρούµε ότι ο παραγόµενος κώδικας είναι πολύ πιο σύνθετος τόσο απο τον δικό µας όσο και απο τον παραγόµενο κωδικα απο το vitis χωρις την προσθήκη των pragmas. Αυτό προκύπτει από τις εντολές pragmas που χρησιµοποιήσαµε. Με τη χρήση των συγκεκριµένων εντολών εκτός από την βελτιστοποίηση στην απόδοση, προκαλείται και επιπλέον δέσµευση πόρων.



Σχήµα 8: Element Wise C++ pragmas code



Σχήµα 9: Synthesis Results

Αυτό γίνεται καθώς µε την µία εντολή χωρίζεται η µνήµη και οι πίνακες σε µικρότερα τµήµατα, κάνοντας την επεξεργασία τους δυνατή από πολλά διαφορετικά τµήµατα της µνήµης που προκύπτει.

Ωστόσο, όσο µεγαλύτερη τµηµατοποίηση γίνεται εξαιτίας της παραπάνω εντολής, τόσο αυξάνεται και η ανάγκη για πόρους του συστήµατος κάτι που είναι φανερό και στα αποτελέσµατα της σύνθεσης αλλά και µε µία πιο προσεκτική µατιά στο κώδικα, εντοπίζοντας όλες τις νέες µεταβλητές, ports και signals που δεσµέυει. Με µία άλλη εντολή παραλληλοποίησης που χρησιµοποιεί το σύστηµα γίνεται ταυτόχρονη επεξεργασία διαφόρων επαναλήψεων ενός βρόγχου επανάληψης.Αυτό πραγµατοποιείται µέσα από τα πολλαπλά αντίγραφα του κάθε βρόγχου που δηµιουργεί η καθε συγκεκριµένη εντολή pragmas που χρησιµοποιείται. Πάραυτα µε τη δηµιουργία του κάθε αντιγράφου, επεκτείνεται και το σχέδιο RTL µε άµεσο αποτέλεσµα τη χρήση επιπλέον πόρων αυξάνοντας τη περιπλοκότητα του κώδικα ραγδαία.

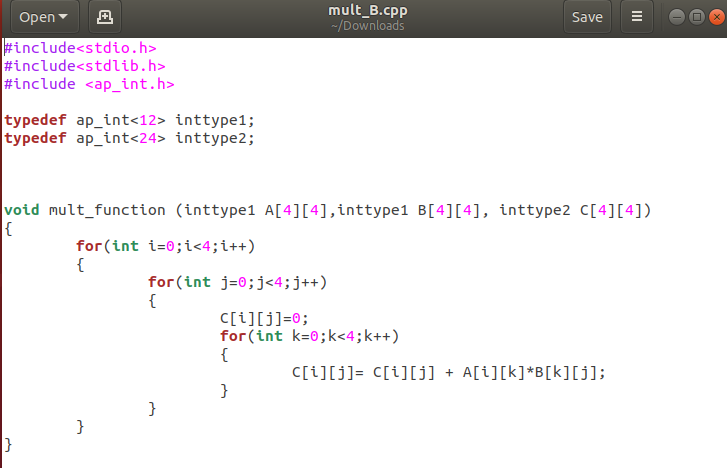
# Ενότητα Β

## Matrix Multiplexer - C++Synthesis

Στον συγκεκριµένο κώδικα έχουµε αναλάβει τη δηµιουργία ένος πολλαπλασιαστή δισδιάστατων πινάκων. Σκοπός του συγκεκριµένου υποερωτήµατος αποτελεί η σύνταξη ενός κώδικα σε γλώσσα προγραµµατισµού C που θα πραγµατοποιεί πολλαπλασιασµό πινάκων ανάµεσα σε 2 δισδιάστατους τετραγωνικούς πίνακες των διαστάσεων 4x4 µε στοιχεία µεγέθους 12-bit . Για τη επιτυχή ολοκλήρωση του ερωτήµατος κατασκευάσαµε έναν κώδικα σε γλώσσα C και ένα test bench το οποίο χρησιµοποιήθηκε για τον έλεγχο της ορθής λειτουργίας του κώδικα. Στο κώδικα C ξεκινάµε µε την εισαγωγή όλων των απαραίτητων βιβλιοθηκών και τη δήλωση του µεγέθους των στοιχείων των πινάκων. Για την έξοδο χρειαζόµαστε τα διπλάσια bit για να είµαστε σίγουροι πως θα µπορέσει να τυπωθεί ολόκληρο το αποτέλεσµα κάθε φορά εποµένως τη δηλώνουµε κατά αντίστοιχο τρόπο. Στη συνέχεια συντάσουµε µία συνάρτηση void η οποία θα κληθεί αργότερα απο το test bench . Στη συνάρτηση αυτή, αφού ορίσουµε το µέγεθος και των τριών πινάκων (2 πίνακες εισόδου και ένας εξόδου), µε τη χρήση 3 εµφωλευµένων for loop πραγµατοποιείται η πράξη του πολλαπλασιασµού και γεµίζει ο πίνακας C , δηλαδή ο πίνακας εξόδου µε τα αποτελέσµατα.

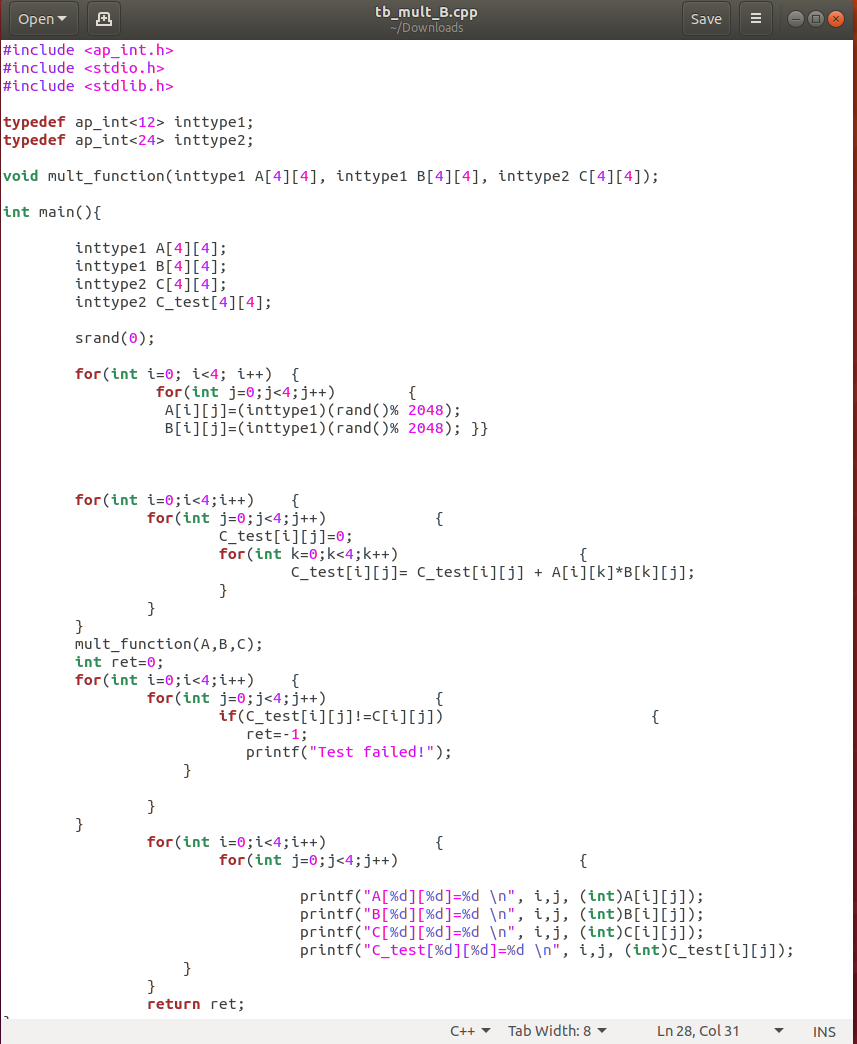
Στον κώδικα για το test bench ξεκινάµε µε την εισαγωγή των βιβλιοθηκών κατα αναλογία και µε τον κώδικα test bench της ενότητας Α και συνεχίζουµε µε τη δήλωση της συνάρτησης mult function που θα φτιάξουµε παρακάτω. Μετά τη δήλωση της συνάρτησης ξεκινάµε τη main . Στην αρχή της main δηλώνουµε πάλι τους πινακές µας καθώς και έναν επιπλέον πίνακα που θα λειτουργήσει σαν έλεγχος των αποτελεσµάτων µας. Για την αρχικοποίηση των πινάκων εισόδου µε τιµές γίνεται χρήση µίας συνάρτησης rand η οποία παράγει τυχαίες τιµές από το 0 εώς και το 2048, δηλαδή τιµές µέχρι 12 bit . Στο βρόγχο for που ακολουθεί υπολογίζουµε τα αποτελέσµατα του πολλαπλασιασµού ανάµεσα στους 2 πίνακες εισόδου και τα αποθηκεύουµε στον πίνακα ελέγχου που έχουµε φτιάξει. Με τον επόµενο βρόγχο πραγµατοποιείται ο έλεγχος ανάµεσα στους δύο πίνακες και σε περίπτωση ασυµφωνίας θα γίνεται εκτύπωση αντίστοιχου µηνύµατος στην κονσόλα. Τέλος, µε µία ακόµη σειρά από εµφωλευµένους βρόχγους γίνεται η τύπωση των αποτελεσµάτων στη κονσόλα.

Οι κώδικες παρατίθενται στα Σχήματα [10,11](#_bookmark18)

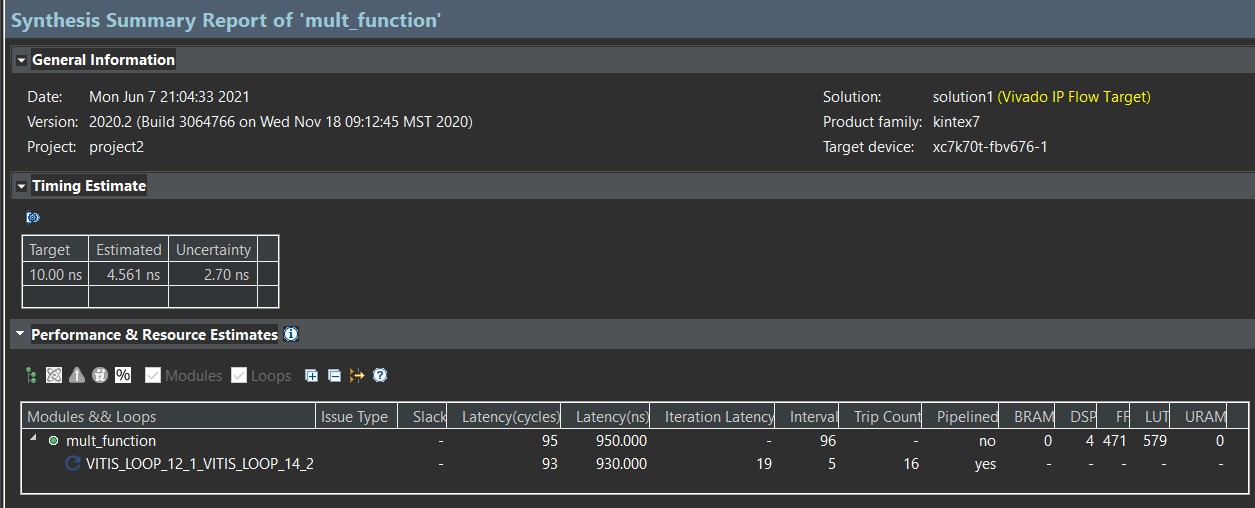


Σχήµα 10: Matrix Multiplication C++ code

Στο Σχήμα [12](#_bookmark19) παρατίθενται τα αποτελέσματα του Synthesis .



Σχήµα 11: Matrix Multiplication C++ Testbench code



Σχήµα 12: Synthesis Results

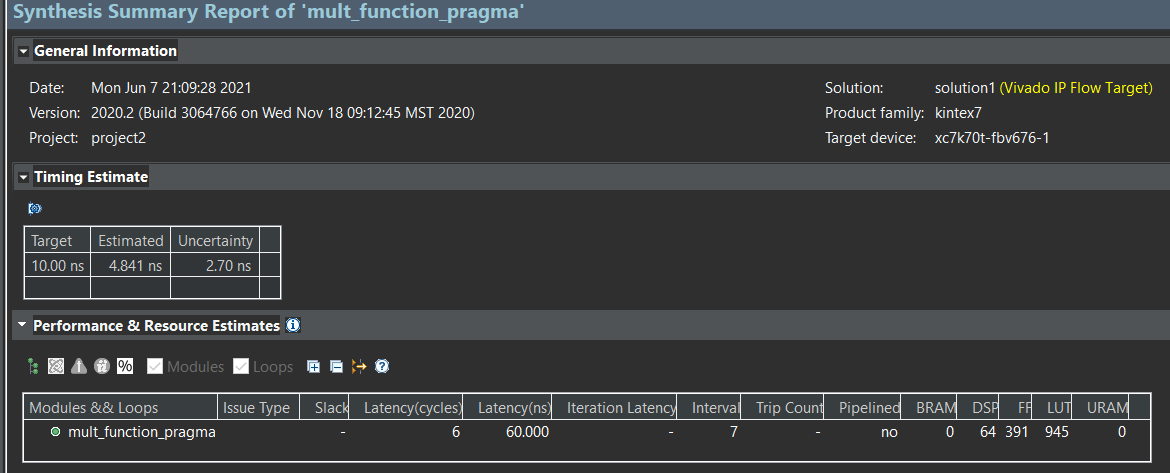
* 1. **Matrix Multiplexer - Pragmas Synthesis**

Για τη βελτιστοποίηση και τη καλύτερη απόδοση του κώδικά µας, επιλέξαµε να προσθέσουµε 2 από τα 3 pragmas που διδαχθήκαµε. Το πρώτο pragmas που προσθέσαµε στον κώδικά µας ήταν η εντολή ”#pragma HLS array partition variable=A complete” η οποία χρησιµοποιήθηκε 3 φορές, µία για

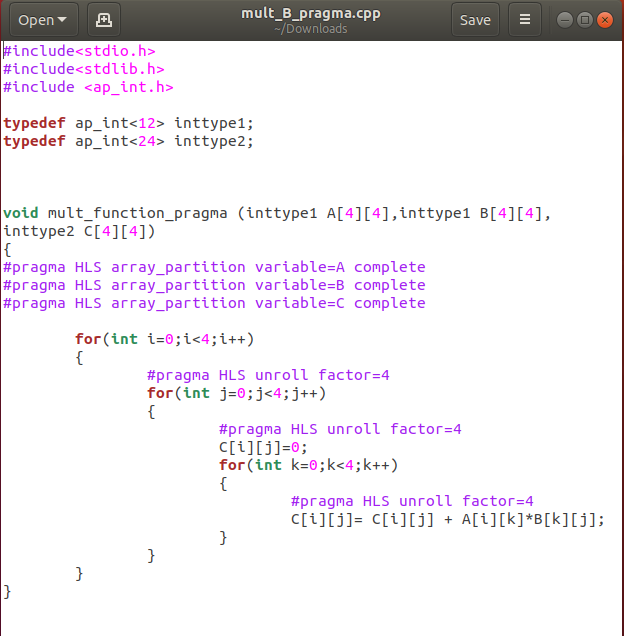
κάθε πίνακα που δηµιουργήσαµε στον κυρίως µας κώδικα. Η εντολή αυτή καταφέρνει να χωρίσει τη µνήµη και τους πίνακες σε µικρότερα τµήµατα (µεµονωµένα στοιχεία εξαιτίας της λέξης complete ), κάνοντας την επεξεργασία τους δυνατή από πολλά διαφορετικά τµήµατα της µνήµης που προκύπτει. Παρατηρούµε ωστόσο κάτι που περιµέναµε, καθώς όσο µεγαλύτερη τµηµατοποίηση γίνεται εξαιτίας της παραπάνω εντολής, τόσο αυξάνεται και η ανάγκη για πόρους του συστήµατος. Αυτό είναι εύκολα αντιληπτό από τα αποτελέσµατα της σύνθεσης των 2 διαφορετικών προγραµµάτων καθώς µετά τη χρήση των pragmas εντοπίζουµε µεγάλη µείωση στο latency και ταυτόχρονη άνοδο στα LUT και στα DSP . Κατασκευάζοντας και επεξεργάζοντας 3 διαφορετικούς πίνακες στον κωδικά µας θεωρήσαµε πως το συγκεκριµένο pragmas θα έκανε µεγάλη διαφορά στους χρόνους και στην απόδοση. Εκτός των πολλών µητρώων, στον κώδικά µας γίνεται και µεγάλη χρήση των βρόγχων for . Συγκεκριµένα, στον κυρίως κώδικά µας γίνεται η χρήση 3 βρόγχων, εµφωλευµένα το ένα στο άλλο. Για να µπορέσουµε λοιπόν να βελτιώσουµε την απόδοση του συγκεκριµένου τµήµατος κώδικα, χρησιµοποιούµε ένα ακόµη pragmas , το ”#pragma HLS unroll factor=4” . Με τη συγκεκριµένη εντολή επιτρέπεται στο σύστηµα να πραγµατοποιήσει µερικές (ή και όλες στη δική µας περίπτωση καθώς το factor έχει τιµή όση και οι επαναλήψεις του βρόγχου) από τις επαναλήψεις του κάθε βρόγχου παράλληλα. Αυτό πραγµατοποιείται µέσα από τα πολλαπλά αντίγραφα του κάθε βρόγχου που δηµιουργεί η καθε συγκεκριµένη εντολή pragmas που χρησιµοποιούµε. Εµείς χρησιµοποιήσαµε τη συγκεκριµένη εντολή 3 φορές, όσα και τα εµφωλευµένα for . Με τη δηµιουργία του κάθε αντιγράφου παρόλάυτά, επεκτείνεται και το σχέδιο RTL µε άµεσο αποτέλεσµα τη χρήση επιπλέον πόρων.

Στο Σχήμα [13](#_bookmark21) παρατίθενται τα αποτελέσματα του Synthesis . Ενώ στο Σχήμα [14](#_bookmark23) βλέπουμε τον

κώδικα µε τα PRAGMAS.



Σχήµα 13: Synthesis Results



# Υπόμνημα

Σχήµα 14: Matrix Multiplexer C++ pragmas code

Για λεπτοµερή αναφορά των αποτελεσµάτων της σύνθεσης, µπορείτε να µεταβείται στο spreadsheet

µε όνοµα SyntesisReport.xlsx, που θα βρείτε στο συµπιεσµένο αρχείο που παραδώσαµε.