

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ
Τμήμα Πληροφορικής



ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Ανάλυση αλγορίθμου AES με την χρήση του εργαλείου HLS

Λιαούτσης Αβραάμ-Αναστάσιος, Π14097

Επιβλέπων: Επίκουρος Καθηγητής Μ. Ψαράκης

Πειραιάς, Ιούνιος 2018



Περίληψη

Βασικό αντικείμενο, αυτής της εργασίας, όπως μαρτυρά και ο τίτλος της είναι η ανάλυση διαφόρων υλοποιήσεων του αλγορίθμου κρυπτογράφησης AES , που προκύπτουν από την χρήση του εργαλείου HLS. Πιο συγκεκριμένα, γίνεται ανάλυση της απόδοσης ενός αλγορίθμου κρυπτογράφησης, καθώς χρησιμοποιούμε τρεις διαφορετικές υλοποιήσεις. Εφόσον, φτιάξαμε τις υλοποιήσεις ώστε να είναι συνθέσιμες για το εργαλείο που χρησιμοποιούμε, τις κάνουμε σύνθεση. Μετά την σύνθεση, μπορέσαμε να βρούμε τις αποδόσεις των τριών διαφορετικών υλοποιήσεων. Καθώς, λάβαμε γνώση τις αποδόσεις, εντοπίσαμε τις διαφορές των υλοποιήσεων αυτών. Στην συνέχεια, εφόσον έχουμε εντοπίσει τις διαφορές, μπορέσαμε πιο εύκολα να δούμε τον λόγο, για την διαφορά των αποδόσεων που υπάρχουν. Έπειτα, χάρις κάποιες παραμετροποιήσεις που κάναμε στην πιο αργή υλοποίηση και μέσω κάποιων εργαλείων που μας παρέχει το εργαλείο HLS, μπορέσαμε να μεγαλώσουμε την απόδοση αυτήνης της υλοποίησης, σε ένα πολύ ικανοποιητικό επίπεδο. Στο τέλος, συμπεραίνουμε ότι με την σωστή γραφή του κώδικα καθώς με την χρήση κάποιων εργαλείων που μας προσφέρει το λογισμικό μπορούμε να φτιάξουμε κυκλώματα ενός πολύπλοκου αλγορίθμου, με υψηλή απόδοση.

Απόδοση στα αγγλικά:

The main object of this work, as evidenced by its title, is the analysis of various implementations of the AES encryption algorithm resulting from the use of the HLS tool. More specifically, the performance of an encryption algorithm was analyzed, as we used three different implementations. Since we've made the implementations to be synthesized for the tool we're using, we're making the synthesis for these implementations. After the synthesis, we were able to find the performances of the three different implementations. Since we learned the odds, we identified the differences from the different three designs. Then, once we have identified the differences, we could more easily see the reason for the difference in yields. Then, thanks to some parameterizations we made to the slower implementation and some of the tools provided by the HLS tool, we were able to increase performance of this implementation to a very satisfactory level. Finally, we conclude that with the correct scripting of the code as with the use of some tools offered by the software we can build circuits of a complex algorithm with high efficiency.



Ευχαριστίες

Πρώτα από όλα ευχαριστώ τον Επιβλέπων καθηγητή κ. Μιχάλη Ψαράκη, για τα εποικοδομητικά του σχόλια, τις διευκρινήσεις καθώς και την καθοδήγηση που μου παρείχε. Τον ευχαριστώ θερμά για τις εξαιρετικά ωφέλιμες παρατηρήσεις του, στην επεξεργασία του θέματος αυτού.

Ακόμη ευχαριστώ όλους τους καθηγητές και τις καθηγήτριες μου, που με δίδαξαν τα μαθήματα της σχολής καθώς μου έδωσαν τις απαραίτητες γνώσεις ώστε να εκπονήσω και εγώ με την σειρά μου την πτυχιακή εργασία.

Έπειτα, θα ήθελα να ευχαριστήσω τους συμφοιτητές μου, που μαζί μέσω συνεργασίας και αλληλοϋποστήριξης μπορέσαμε να φθάσουμε στο τελικό στάδιο της σχολής.

Τέλος, θέλω να ευχαριστήσω όλη την οικογένεια και τους φίλους μου, που τόσα χρόνια πίστευαν σε μένα και μου έδιναν κουράγιο και δύναμη να συνεχίσω τα μαθήματα μου καθώς και να ολοκληρώσω αυτήν την εργασία.

Ιούνιος 2018,
Λιαούτσης Αβραάμ



Περιεχόμενα

1.Εισαγωγή.....	5
2.High Level Synthesis	7
2.1 Κατανοώντας το HLS	8
2.2 Test Bench, Υποστήριξη γλωσσών, βιβλιοθήκες της C.	10
2.2.1 Test Bench	10
2.2.2 Υποστήριξη γλωσσών	10
2.2.3 Βιβλιοθήκες της C.....	11
2.4 Σύνθεση	12
2.5 Βελτιστοποίηση.....	12
2.6 Ανάλυση	17
2.7 Επαλήθευση RTL	18
2.8 Προτάσεις.....	18
2.8.1 Κώδικας	18
2.8.2 Γλώσσα Προγραμματισμού	23
2.8.3 Directives	23
3. Μελέτη του αλγορίθμου AES.....	26
3.1 Ανάλυση συναρτήσεων.....	29
4. Ανάλυση υλοποιήσεων	35
4.1 Τροποποιήσεις υλοποιήσεων για σύνθεση	35
4.1.1 Κώδικας Core.....	35
4.1.2 Κώδικας Niyaz.....	37
4.1.3 Κώδικας ChStone.....	39
4.2 Ομογενοποίηση υλοποιήσεων.....	41
4.2.1 Κύκλωμα Niyaz	44
4.2.2 Κύκλωμα ChStone	46
4.3 Σύγκριση των υλοποιήσεων.....	48
5.Βελτιστοποίηση κυκλώματος.....	70
6.Συμπεράσματα.....	79
7.Βιβλιογραφικές Πηγές.....	80



Κεφάλαιο 1^ο

1.Εισαγωγή

Οι βασικές γλώσσες περιγραφής υλικού (Hardware Description Language, HDL) που γράφουν οι προγραμματιστές για να μπορέσουν να προγραμματίσουν συσκευές FPGA, είναι η Verilog και η VHDL. Οι προγραμματιστές για παρά πολλά χρόνια χρησιμοποιούν αυτές τις γλώσσες που καθιστούν πάρα πολύ δύσκολη και χρονοβόρα την διαδικασία να φτιάξουν, καθώς να αποσφαλματώσουν προγράμματα γραμμένα σε γλώσσα περιγραφής υλικού, σε σύγκριση με ένα πρόγραμμα που θα μπορούσαν να είχαν φτιάξει χρησιμοποιώντας γλώσσα υψηλού επιπέδου. Όλο αυτό άλλαξε, καθώς η εταιρεία Xilinx το 2012 έφτιαξε το λογισμικό σχεδίασης Vivado Suite που περιλαμβάνει το εργαλείο HLS (High Level Synthesis ή Σύνθεση υψηλού επιπέδου). Η συγκεκριμένη εταιρεία και στο παρελθόν είχε φτιάξει λογισμικά που χρησιμοποιούνται από πάρα πολλούς προγραμματιστές, ώστε να προγραμματίσουν συσκευές FPGA. Για παράδειγμα είχε φτιάξει το λογισμικό σχεδίασης Xilinx ISE, που χρησιμοποιείτε από πολλούς προγραμματιστές εδώ και παρά πολλά χρόνια. Με την τελευταία προσθήκη του εργαλείου HLS στο λογισμικό σχεδίασης Vivado, η Xilinx έφερε την επανάσταση στο τρόπο προγραμματισμού συσκευών. Πλέον, με αυτό το εργαλείο μπορεί ο κάθε προγραμματιστής να φτιάξει προγράμματα που είναι γραμμένα σε γλώσσα υψηλού επιπέδου για παράδειγμα C, C++ και ακόμη και SystemC, να τα κάνει σύνθεση ώστε να δημιουργήσει μια υλοποίηση που θα τρέχει αυτομάτως σε μια συσκευή FPGA, χωρίς να χρειαστεί να γράψει καθόλου κώδικα περιγραφής υλικού. Με την χρήση αυτού του εργαλείου, μπορεί να σώσει ένα σημαντικό μέρος από το χρόνο του, που θα μπορούσε να δαπανήσει, φτιάχνοντας την υλοποίηση από την αρχή. Προηγούμενες έρευνες έχουν δείξει ότι το εργαλείο Σύνθεσης υψηλού επιπέδου μπορεί αναμφίβολα, να παράξει κυκλώματα για συσκευές που ήταν γραμμένα σε γλώσσα υψηλού επιπέδου και να λειτουργήσουν σωστά. Δυστυχώς όμως, επειδή αποτελεί ένα καινούριο λογισμικό, δεν γνωρίζουμε αν μπορεί να φτιάξει κυκλώματα με υψηλή απόδοση, πράγμα που αποτελεί παρά πολύ κρίσιμος παράγοντας για τους προγραμματιστές. Το ερώτημα αυτό παραμένει ευρέως ανοικτό ειδικά στο τομέα που αφορά την κρυπτογράφηση και ασφάλειας[2]. Ο λόγος που αφορά πιο πολλούς αυτούς τους τομείς, είναι εξαιτίας της πολυπλοκότητας και δυσκολίας που έχουν αυτοί οι αλγόριθμοι, ώστε να υλοποιηθούν. Σκοπός αυτής της εργασίας είναι να αναλύσουμε την απόδοση διαφόρων υλοποιήσεων ενός αλγόριθμου κρυπτογράφησης που έχουν γραφτεί σε γλώσσα υψηλού επιπέδου. Πιο συγκεκριμένα, θα αναλύσουμε την αλγόριθμο AES (Advanced Encryption Standard) που αποτελεί ένας ιδιαίτερα σημαντικός αλγόριθμος που χρησιμοποιείται από πολλούς οργανισμούς καθώς από την υλοποίηση του, καταλαβαίνουμε ότι έχει μεγάλη πολυπλοκότητα. Στην συνέχεια, αναλύουμε τις τρεις διαφορετικές υλοποιήσεις του αλγόριθμου αυτού, καθώς συμπεραίνουμε ότι με την βοήθεια κάποιων εργαλείων που προσφέρει το κύκλωμα καθώς με την κατάλληλη συγγραφή κώδικα μπορούμε να φτιάξουμε κυκλώματα για συσκευές με υψηλή απόδοση.



Η δομή της εργασίας έχει ως εξής:

Στο **Πρώτο Κεφάλαιο** της εργασίας παρατίθενται η εισαγωγή της πτυχιακής εργασίας. Αναφέρουμε συνοπτικά αυτά που κάναμε στην εργασία αυτή.

Στο **Δεύτερο Κεφάλαιο** γίνεται η εισαγωγή του αναγνώστη στο πρόγραμμα που χρησιμοποιούμε, που είναι το εργαλείο HLS. Αναφέρουμε λεπτομερώς την λειτουργία του καθώς τι περιλαμβάνει.

Το **Τρίτο Κεφάλαιο** αφορά τον αλγόριθμο κρυπτογράφησης που υλοποιούμε. Στην αρχή, αναφέρουμε κάποια εισαγωγικά στοιχεία και στην συνέχεια αναλύουμε τα στάδια υλοποίησης του.

Το **Τέταρτο Κεφάλαιο** αφορά την επεξεργασία των διάφορων υλοποιήσεων του αλγόριθμου κρυπτογράφησης ώστε να γίνουν συνθέσιμοι από το εργαλείο HLS. Στην συνέχεια, κάνουμε διάφορες αλλαγές ώστε οι τρεις υλοποιήσεις, να έχουν κοινά ορίσματα για να γίνει σωστή σύγκριση.

Στο **Πέμπτο κεφάλαιο**, κάνουμε τροποποιήσεις, με διάφορους τρόπους ώστε να επιταχύνουμε την απόδοση του πιο αργού κυκλώματος.

Η εργασία ολοκληρώνεται με το **Έκτο Κεφάλαιο** με επιλεκτική σύνοψη και απόδοση των συμπερασμάτων που προκύπτουν.



Κεφάλαιο 2^ο

2.High Level Synthesis

Το εργαλείο high-level synthesis (HLS) ή Σύνθεση Υψηλού Επιπέδου αποτελεί μια επέκταση του λογισμικού σχεδίασης Vivado το οποίο κατασκευάστηκε από την εταιρεία Xilinx τον Απρίλιο του 2012. Το λογισμικό σχεδίασης Vivado αποτελεί ένα πρόγραμμα για τη σχεδίαση και ανάλυση κυκλωμάτων περιγραφής υλικού, αντικαθιστώντας το προηγούμενο λογισμικό που είχε φτιάξει, το οποίο ήταν το Xilinx ISE (Integrated Synthesis Environment). Τα πρόσθετα χαρακτηριστικά του λογισμικού σχεδίασης Vivado είναι η δυνατότητα ανάπτυξης chip καθώς και η προσθήκη της Σύνθεσης Υψηλού Επιπέδου που θα μελετήσουμε περαιτέρω. Το λογισμικό σχεδίασης Vivado αντιπροσωπεύει μια επαναγραφή και επανεξέταση της συνολικής ροής σχεδιασμού (σε σύγκριση με το ISE) και έχει περιγραφεί από πολλούς ως «καλά σχεδιασμένο, γρήγορο και διατηρήσιμο εργαλείο»[6].

Οι επεκτάσεις που έχει το λογισμικό Vivado Design Suite είναι οι παρακάτω:

- Το Vivado TCL Store αποτελεί ένα σύστημα σεναρίων για την ανάπτυξη πρόσθετων επεκτάσεων για το λογισμικό σχεδίασης Vivado. Μπορεί να χρησιμοποιηθεί για να προσθέσει και να τροποποιήσει τις δυνατότητες του λογισμικού σχεδίασης Vivado. Το TCL που προέρχεται από τις λέξεις Tool Command Language, αποτελεί μια γλώσσα σεναρίων στην οποία βασίζεται το ίδιο το λογισμικό σχεδίασης Vivado. Όλες οι λειτουργίες που προσφέρει το εργαλείο μπορούν να χρησιμοποιηθούν και να ελεγχθούν από TCL σενάρια ή αλλιώς scripts.
- Το Vivado Simulator αποτελεί έναν προσομοιωτή το οποίο υποστηρίζει διάφορες γλώσσες προγραμματισμού, TCL scripts, κρυπτογραφημένες IP καθώς και βελτιωμένες επαληθεύσεις για το τελικό αποτέλεσμα μετά την σύνθεση.
- Το Vivado IP Integrator επιτρέπει στους προγραμματιστές να ενσωματώσουν και να ρυθμίσουν γρήγορα τα IP(Intellectual Property) με την βοήθεια της χρήσης της μεγάλης βιβλιοθήκη IP που υποστηρίζει το εργαλείο. Το Integrator είναι επίσης σχεδιασμένο ώστε να δουλεύει κυκλώματα MathWorks Simulink τα οποία όμως έχουν κατασκευαστεί μέσω των εργαλείων Xilinx System Generator και σύνθεσης υψηλού επιπέδου.
- Το εργαλείο σύνθεσης υψηλού επιπέδου αποτελεί έναν μεταγλωττιστή που επιτρέπει στα προγράμματα που έχουν γραφτεί σε γλώσσα C, C++ και SystemC να λειτουργήσουν άμεσα στις συσκευές FPGA (Field-programmable gate array) χωρίς την δημιουργία κυκλώματος RTL(Register Transfer Level). Το εργαλείο σύνθεσης υψηλού επιπέδου αυξάνει την παραγωγικότητα των προγραμματιστών, καθώς έχει επιβεβαιωθεί ότι υποστηρίζει κλάσεις, πρότυπα, συναρτήσεις καθώς και διάφορους τελεστές της γλώσσας C++.



2.1 Κατανοώντας το HLS

Το εργαλείο σύνθεσης υψηλού επιπέδου [3] μπορεί να μετατρέψει συναρτήσεις που είναι γραμμένες σε γλώσσα υψηλού επιπέδου σε IP Block, τα οποία μπορούν να ενσωματωθούν στο υλικό. Επειδή είναι ενσωματωμένο στο λογισμικό σχεδίασης του Vivado, μπορεί να χρησιμοποιηθεί το HLS και από άλλα εργαλεία σχεδιασμού της Xilinx καθώς υποστηρίζει πολλά εργαλεία για την δημιουργία της βέλτιστης υλοποίησης του αλγορίθμου που θα γραφτεί σε γλώσσα υψηλού επιπέδου. Η διαδικασία εκτέλεσης του εργαλείου είναι οι παρακάτω:

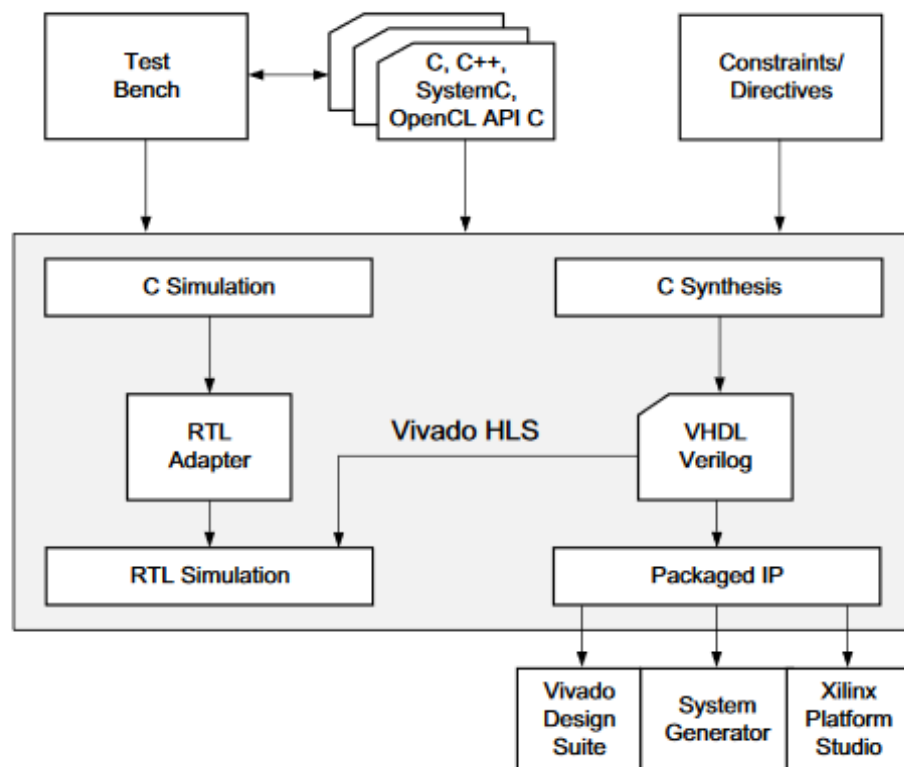
1. Μεταγλώττιση, εκτέλεση (προσομοίωση) και αποσφαλμάτωση του αλγορίθμου C.
2. Σύνθεση του αλγορίθμου C σε υλοποίηση RTL. Μπορούν να χρησιμοποιηθούν διάφορα directives για βελτιστοποίηση.
3. Δημιουργία αναλυτικών αναφορών και ανάλυση της απόδοσης κυκλώματος μετά την σύνθεση.
4. Επαλήθευση της RTL υλοποίησης με την χρήση μιας ροής που θα δημιουργηθεί.
5. Μετατροπή των RTL υλοποιήσεων σε ένα εύρος από IP formats.

Το εργαλείο HLS μπορεί να πάρει ως εισόδους τα παρακάτω:

- Συναρτήσεις γραμμένες σε γλώσσα υψηλού επιπέδου όπως C, C++, SystemC. Αποτελούν την βασική είσοδο για το εργαλείο HLS. Επιπλέον, οι συναρτήσεις αυτές, μπορούν να εμπεριέχουν και άλλες υπό-συναρτήσεις.
- Ορίσματα. Τα ορίσματα είναι υποχρεωτικά και χρειάζονται να τα συμπεριλάβουμε στο κύκλωμα. Τα ορίσματα αυτά είναι η περίοδος του ρολογιού, η απόκλιση που μπορεί να έχει το ρολόι καθώς και η συσκευή FPGA που μπορούμε να χρησιμοποιήσουμε για να τρέξουμε τον αλγόριθμο. Η απόκλιση του ρολογιού είναι περίπου ως προεπιλογή στο 12.5 % της περιόδου του ρολογιού.
- Directives. Τα Directives αποτελούν προαιρετικά εργαλεία που μπορούν να χρησιμοποιηθούν στην σύνθεση του κυκλώματος, ώστε να υλοποιήσουν μια συγκεκριμένη συμπεριφορά ή και ακόμη για να βελτιστοποιήσουν το κύκλωμα.
- Test bench και άλλα αρχεία. Το εργαλείο σύνθεσης υψηλού επιπέδου χρησιμοποιεί C test bench ώστε να προσομοιώσει τον αλγόριθμο πριν την σύνθεση του. Και ακόμη να επαληθεύσει τα αποτελέσματα, που θα βγάλει το κύκλωμα που έχει μετατραπεί σε RTL. Για αυτήν την λειτουργία χρησιμοποιείται η C/RTL συν-προσομοίωση.

Οι εξόδους που παράγει το εργαλείο HLS είναι τα παρακάτω:

- Αρχεία υλοποίησης RTL που είναι γραμμένα σε γλώσσα περιγραφής υλικού. Αποτελεί την βασική έξοδο της Σύνθεσης Υψηλού Επιπέδου. Μπορούμε να μετατρέψουμε το RTL σε υλοποίηση επιπέδου πυλών (gate-level implementation) και να παράξουμε αρχείο FPGA bitstream που θα χρησιμοποιηθεί ώστε να δουλέψει το κύκλωμα στην συσκευή FPGA. Το αρχείο που θα παραχθεί μπορεί να είναι διαθέσιμο είτε σε γλώσσα VHDL είτε σε Verilog.
- Αρχεία ανάλυσης. Αποτελεί την έξοδο που θα βγάλει το κύκλωμα μετά την σύνθεση, C/RTL συν-προσομοίωση και IP Packaging.



Εικόνα 1:Επισκόπηση εισοδών και εξόδων του λογισμικού σύνθεσης υψηλού επιπέδου



2.2 Test Bench, Υποστήριξη γλωσσών, βιβλιοθήκες της C.

Σε κάθε πρόγραμμα που είναι γραμμένο σε C, η κύρια συνάρτηση ονομάζεται `main()`. Στην σύνθεση υψηλού επιπέδου, μπορούμε να ορίσουμε οποιαδήποτε υπό-συνάρτηση που είναι κάτω από την `main` ως κύρια συνάρτηση για σύνθεση. Ωστόσο, δεν μπορούμε να κάνουμε σύνθεση την συνάρτηση `main()`. Επιπλέον πρέπει να ισχύουν όλα τα παρακάτω:

- Μόνο μια συνάρτηση επιτρέπεται να είναι κύρια συνάρτηση για σύνθεση.
- Όποια συνάρτηση περιλαμβάνεται σε αυτήν και καλείται στην κύρια συνάρτηση τότε και η ίδια θα μετατραπεί.
- Αν θέλουμε να κάνουμε σύνθεση συναρτήσεων τα οποία δεν ανήκουν στην ιεραρχία, δηλαδή δεν είναι κάτω από την κύρια συνάρτηση που θα γίνει σύνθεση, τότε πρέπει να ενώσουμε αυτές τις συναρτήσεις με την κύρια ώστε να μετατραπούν και αυτές.

2.2.1 Test Bench

Όταν χρησιμοποιούμε το εργαλείο HLS, είναι πολύ χρονοβόρο να κάνουμε σύνθεση μια συνάρτηση C, η οποία δεν λειτουργεί σωστά και στην συνέχεια, να αναλύσουμε τα αποτελέσματα που βγάζει, ώστε να μπορέσουμε να εντοπίσουμε τους λόγους που δεν λειτουργεί. Για αυτόν τον λόγο, για να αυξήσουμε την παραγωγικότητα μας, μπορούμε να χρησιμοποιήσουμε τα αρχεία `test benches` ώστε να δούμε αν δουλεύει σωστά η συνάρτηση πριν κάνουμε καν την σύνθεση.

Το αρχείο `test bench` περιέχει την κύρια συνάρτηση `main()` και μπορεί να περιέχει και άλλες συναρτήσεις που μπορεί να μην ανήκουν στην δομή, ιεραρχία της κύρια συνάρτησης που πρόκειται να γίνει σύνθεση. Με αυτές τις συναρτήσεις μπορούμε να ελέγξουμε αν η κύρια συνάρτηση που πρόκειται να γίνει σύνθεση, λειτουργεί σωστά, βγάζοντας την ανάλογη έξοδο στην οθόνη.

Το εργαλείο σύνθεσης υψηλού επιπέδου χρησιμοποιεί τα αρχεία `test benches` για να μεταγλωττίσει και να εκτελέσει την C-Προσομοίωση. Κατά την διάρκεια της μεταγλώττισης μπορούμε να διαλέξουμε την επιλογή **Launch Debugger** για να κάνουμε αποσφαλμάτωση του προγράμματος σε περίπτωση που χρειαστεί.

2.2.2 Υποστήριξη γλωσσών

Γλώσσες που υποστηρίζει:

- C
- C++
- SystemC



Το εργαλείο σύνθεσης υψηλού επιπέδου υποστηρίζει διάφορα εργαλεία της C, C++, και SystemC καθώς πολλά άλλα, όπως πραγματικούς αριθμούς καθώς και αριθμούς διπλής ακρίβειας. Ωστόσο, δεν μπορεί να κάνει σύνθεση σε όλα αυτά που υποστηρίζουν οι παραπάνω γλώσσες, όπως δηλαδή:

- Δυναμική κατανομή μνήμης. Μια συσκευή FPGA έχει ένα σταθερό σύνολο πόρων και η δυναμική δημιουργία καθώς η απελευθέρωση πόρων μνήμης δεν υποστηρίζεται.
- Λειτουργίες λειτουργικού συστήματος (OS). Όλα τα δεδομένα από και προς τις συσκευές FPGA πρέπει να διαβάζονται από τις πύλες εισόδου ή να γράφονται στις πύλες εξόδου. Δεν υποστηρίζονται λειτουργίες του λειτουργικού συστήματος όπως διάβασμα/δημιουργία αρχείου ή συναρτήσεις που χρησιμοποιούν την ώρα και την ημερομηνία. Αντί αυτού μπορούμε να χρησιμοποιήσουμε τις παραπάνω λειτουργίες και να μεταφέρουμε τα δεδομένα στην συνάρτηση που πρόκειται να γίνει σύνθεση μέσω των αρχείων test benches.

2.2.3 Βιβλιοθήκες της C

Εκτός από όλα αυτά, το εργαλείο σύνθεσης υψηλού επιπέδου υποστηρίζει και πολλές βιβλιοθήκες της γλώσσας C, οι οποίες περιλαμβάνουν συναρτήσεις και άλλα εργαλεία τα οποία είναι χρήσιμα για την βελτιστοποίηση του κυκλώματος. Η χρήση αυτών των βιβλιοθηκών, συμβάλλει στην εξασφάλιση υψηλής ποιότητας αποτελεσμάτων ώστε να παράξουμε κυκλώματα υψηλής απόδοσης, εφόσον έχουμε κάνει την βέλτιστη χρήση των πόρων που μας δίνονται. Αυτές οι βιβλιοθήκες μπορούν να χρησιμοποιηθούν για την προσομοίωση του κυκλώματος πριν γίνει η σύνθεση.

Το εργαλείο σύνθεσης υψηλού επιπέδου υποστηρίζει και άλλες βιβλιοθήκες για να επεκτείνει τις δυνατότητες της γλώσσας C και αυτές είναι οι παρακάτω:

- Τύποι δεδομένων αυθαίρετης ακρίβειας
- Τύποι δεδομένων κινητής υποδιαστολής μισής ακρίβειας των 16 bit
- Μαθηματικές συναρτήσεις
- Συναρτήσεις για Video
- Συναρτήσεις Xilinx IP, συμπεριλαμβάνοντας μεθόδους όπως τον Fast Fourier Transform (FFT) και Finite impulse response (FIR)
- Συναρτήσεις πόρων της FPGA συσκευής ώστε να μεγιστοποιείται η χρησιμότητα των πόρων shift register LUT (Look Up tables)



2.4 Σύνθεση

Το εργαλείο σύνθεσης υψηλού επιπέδου αποτελεί πρόγραμμα το οποίο βασίζεται πάνω σε project. Κάθε project εμπεριέχει ένα σύνολο από C κώδικα και μπορεί να εμπεριέχει διάφορες λύσεις-υλοποιήσεις για το project αυτό. Κάθε υλοποίηση μπορεί να εμπεριέχει διάφορα ορίσματα και directives για βελτιστοποίηση. Επιπλέον, μπορούμε να αναλύσουμε και να συγκρίνουμε τα αποτελέσματα κάθε υλοποίησης.

Τα βήματα για την χρήση του λογισμικού αυτού είναι τα παρακάτω:

1. Δημιουργία project με μια αρχικοποιημένη υλοποίηση δηλαδή γραμμένη σε γλώσσα υψηλού επιπέδου.
2. Προσομοίωσης υψηλού επιπέδου ώστε να μην παρουσιαστεί κανένα σφάλμα.
3. Σύνθεση της υλοποίησης.
4. Ανάλυση των αποτελεσμάτων.

Μετά την ανάλυση των αποτελεσμάτων, μπορούμε να δημιουργήσουμε εναλλακτικές λύσεις για το project με διαφορετικά ορίσματα και directives. Επιπλέον, μπορούμε να επαναλάβουμε αυτήν την διαδικασία μέχρι το κύκλωμα μας να έχει την επιθυμητή απόδοση.

2.5 Βελτιστοποίηση

Με την χρήση του εργαλείου σύνθεσης υψηλού επιπέδου, μπορούμε να εφαρμόσουμε στις υλοποιήσεις μας διάφορα directives ώστε να τα βελτιστοποιήσουμε. Κάποια από αυτά είναι:

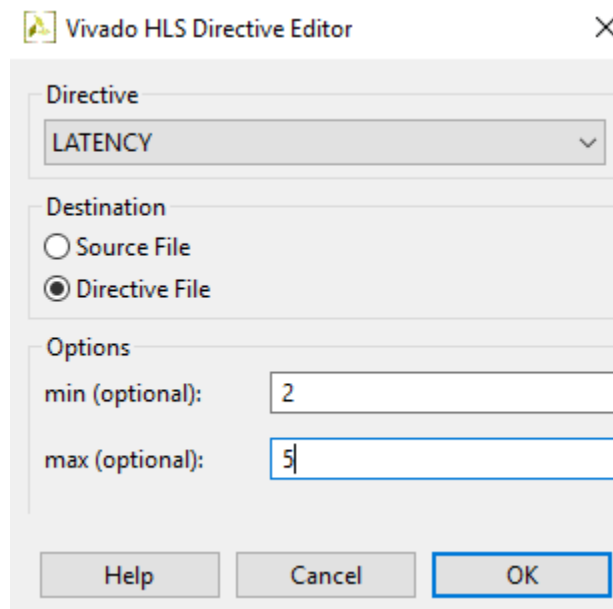
- Με την χρήση του Directive Pipeline, μπορούμε να κάνουμε μια εντολή να εκτελεστεί, πριν ολοκληρωθεί η τρέχουσα εντολή αξιοποιώντας πολύτιμο χρόνο. Για παράδειγμα, αν έχουμε δύο υλοποιήσεις, που στην πρώτη χρησιμοποιούμε το Directive Pipeline μπορούμε να εκτελέσουμε τις εντολές παράλληλα στον ίδιο κύκλο, ενώ στην δεύτερη υλοποίηση που δεν χρησιμοποιείται αυτό το Directive, θα πρέπει η μια εντολή να περιμένει την προηγούμενη εντολή να ολοκληρωθεί, ώστε να εκτελεστεί και αυτή. Στην Εικόνα 2 βλέπουμε έναν βρόγχο στον οποίο εκχωρούμε τον πίνακα temp και temp1 στους πίνακες Key και In.

```
Pipeline:for (i=0;i<16;i++)  
{  
    Key[i]=temp[i];  
    in[i]=temp2[i];  
}
```

Εικόνα 2: Κώδικας

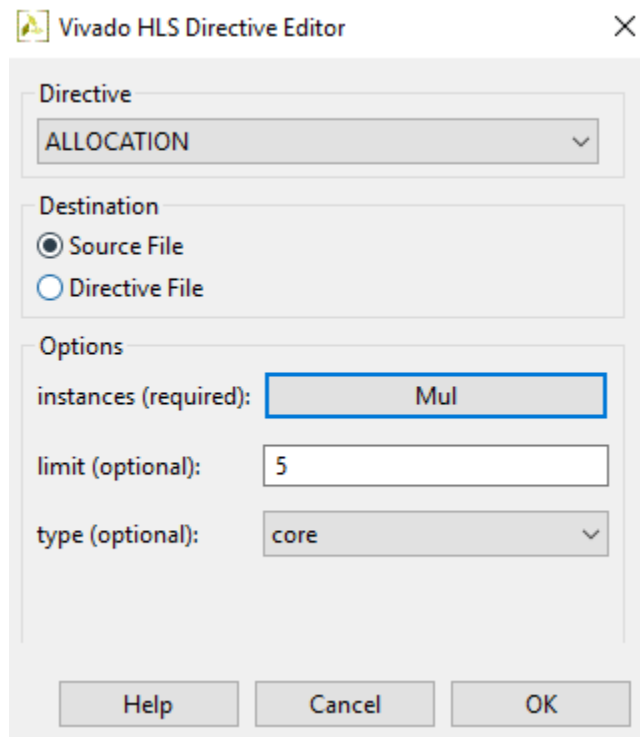
Η αρχική επιβάρυνση στην απόδοση του κυκλώματος ήταν 33 κύκλους ρολογιού, καθώς με την χρήση του Directive Pipeline η καθυστέρηση μειώθηκε στους 18 κύκλους. Από ένα απλό βρόγχο βλέπουμε πόσο καθυστέρηση μπορούμε να γλιτώσουμε με την χρήση του Directive Pipeline.

- Με την χρήση του Directive Latency, μπορούμε να καθορίσουμε το χρόνο που χρειάζεται για την ολοκλήρωση των συναρτήσεων και βρόγχων. Για παράδειγμα, μπορούμε να ορίσουμε τον μέγιστο χρόνο που καταναλώνει μια συνάρτηση ή ένας βρόγχος για να εκτελεστεί ώστε να μην υπάρχει επιπρόσθετη επιβάρυνση στην απόδοση του κυκλώματος. Στην Εικόνα 3, βλέπουμε τα πεδία στα οποία μπορούμε να καθορίσουμε την μέγιστη και την ελάχιστη καθυστέρηση που επιβαρύνει η συνάρτηση ή ο βρόγχος που επιλέξαμε. Στην Συγκεκριμένη περίπτωση, επιλέξαμε ο βρόγχος που υπάρχει στην Εικόνα 2 να έχει ελάχιστη καθυστέρηση 2 κύκλους και μέγιστη 5 κύκλους ρολογιού.



Εικόνα 3: Καθορισμός Χρόνου Επιβάρυνσης Συναρτήσεων

- Μπορούμε να καθορίσουμε το όριο των πόρων που χρησιμοποιούνται στο κύκλωμα με την χρήση του Directive Allocation. Για παράδειγμα, εάν χρησιμοποιούμε μια συσκευή FPGA που δεν υποστηρίζει μεγάλο μέγεθος υλικού, μπορούμε να μειώσουμε τους πόρους που δεσμεύει το κύκλωμα, αυτή όμως η κίνηση θα μειώσει την απόδοση του. Αντιθέτως, εάν έχουμε μια συσκευή FPGA, που υποστηρίζει μεγάλο μέγεθος υλικού μπορούμε να χρησιμοποιήσουμε αυτό το Directive, για να δεσμεύσουμε πιο πολλούς πόρους ώστε να αυξήσουμε την απόδοση του κυκλώματος μας. Στην Εικόνα 4, βλέπουμε τα πεδία που μπορούμε να καθορίσουμε τον τύπο, που μπορεί να είναι είτε υλικό όπως RAM, ROM είτε εντολή όπως πολλαπλασιασμός και διαίρεση, καθώς και το αριθμητικό όριο αυτών στο κύκλωμα μας.



Εικόνα 4: Καθορισμός Πόρων Υλικού



- Με την χρήση του Directive Inline ,μπορούμε να καταργήσουμε τις εξαρτήσεις των εντολών του κώδικα και να εκτελέσουμε κάποιες συγκεκριμένες λειτουργίες. Για παράδειγμα, μπορούμε να κάνουμε τις εντολές να μην έχουν εξαρτήσεις μεταξύ τους, με την δημιουργία νέων καταχωρητών και δέσμευσης υλικού. Σε συνδυασμό με το Directive Pipeline μπορούμε να αυξήσουμε σε ένα πολύ ικανοποιητικό επίπεδο την απόδοση του κυκλώματος. Στην Εικόνα 5 βλέπουμε την συνάρτηση shiftRows που επιβαρύνει την απόδοση του κυκλώματος κατά 117 κύκλους ρολογιού. Με την χρήση του Directive Inline, μπορέσαμε να μειώσουμε την επιβάρυνση κατά 10 κύκλους άρα να τους φθάσουμε στους 107 κύκλους.

```
Inline:void ShiftRows(unsigned char state[4][4])
{
    unsigned char temp;

    // Rotate first row 1 columns to left
    temp=state[1][0];
    state[1][0]=state[1][1];
    state[1][1]=state[1][2];
    state[1][2]=state[1][3];
    state[1][3]=temp;

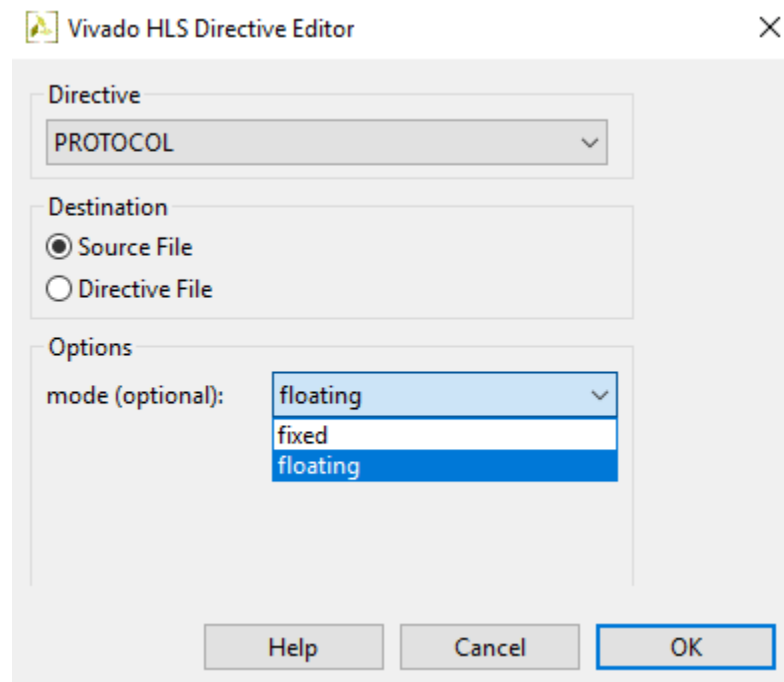
    // Rotate second row 2 columns to left
    temp=state[2][0];
    state[2][0]=state[2][2];
    state[2][2]=temp;

    temp=state[2][1];
    state[2][1]=state[2][3];
    state[2][3]=temp;

    // Rotate third row 3 columns to left
    temp=state[3][0];
    state[3][0]=state[3][3];
    state[3][3]=state[3][2];
    state[3][2]=state[3][1];
    state[3][1]=temp;
}
```

Εικόνα 5: Κώδικας

- Με την χρήση του Directive Protocol, μπορούμε να καθορίσουμε μια περιοχή του κώδικα στην οποία δεν εισάγονται λειτουργίες ρολογιού από το εργαλείο HLS, εκτός αν το ορίζεται ρητά στον κώδικα. Χρησιμοποιείται ώστε να διασφαλίσουμε ότι ο τελικός σχεδιασμός, θα συνδεθεί σωστά με τα άλλα block του υλικού με το ίδιο πρωτόκολλο εισόδου και εξόδου. Στην Εικόνα 6 βλέπουμε τα ορίσματα που μπορεί να λάβει αυτό το Directive που είναι floating και fixed. Η προεπιλεγμένη επιλογή είναι το floating που επιτρέπει τις εντολές που δεν ανήκουν στο περιοχή του πρωτόκολλου, να χρησιμοποιήσουν αυτές που ανήκουν. Σε αντίθεση, το fixed δεν επιτρέπει αυτήν την λειτουργία.



Εικόνα 6: Καθορισμός Πρωτοκόλλου

2.6 Ανάλυση

Όταν ολοκληρωθεί η σύνθεση, το εργαλείο σύνθεσης υψηλού επιπέδου αυτομάτως φτιάχνει μια αναφορά για το παραγόμενο ώστε να μας βοηθήσει να καταλάβουμε την απόδοση της υλοποίησης μας. Στο γραφικό περιβάλλον του εργαλείου HLS το analysis perspective εμπεριέχει το πίνακα απόδοσης, στο οποίο μας επιτρέπει να δούμε και να αναλύσουμε με λεπτομέρεια τα αποτελέσματα.

```
void Add(unsigned char first[4], unsigned char second[4],
         unsigned char out[16])
{
    for(int i=0; i<4; i++){
        out[i] = first[i]+second[i];
    }
}
```

	Operation\Control Step	C0	C1	C2
1	Loop 1			
2	i(phi_mux)			
3	exitcond icmp)			
4	i_1(+)			
5	first_load(read)			
6	second_load(read)			
7	tmp(+)			
8	node_23(write)			

Εικόνα 7: Παράδειγμα ανάλυσης απόδοσης

Στην Εικόνα 7 βλέπουμε ένα παράδειγμα ανάλυσης της απόδοσης της συνάρτησης Add. Πιο συγκεκριμένα, παρατηρούμε ότι η συνάρτηση Add περιλαμβάνει έναν βρόγχο (Loop 1), στον οποίο γίνονται όλες οι πράξεις της συνάρτησης. Οι πράξεις ξεκινούν στο 2^ο κύκλο ρολογιού(C1), εφόσον ο βρόγχος ξεκινάει πάντα στον επόμενο κύκλο. Επίσης, παρατηρούμε ότι γίνονται παράλληλα σε έναν κύκλο(C1) οι εντολές δημιουργίας μεταβλητής i, ο έλεγχος της τιμής της καθώς και αύξηση της κατά μία μονάδα. Στην συνέχεια, βλέπουμε ότι έχουμε δύο πίνακες first και second που διαβάζουν από την μνήμη καθώς αυτή η λειτουργία χρειάζεται συνολικά 2 κύκλους ρολογιού(Από C1 στην C2). Οι υπόλοιπες εντολές που είναι η πρόσθεση και το γράψιμο στον πίνακα βλέπουμε ότι γίνονται παράλληλα στον 3^ο κύκλο (C2).

Για να υπολογίσουμε την συνολική καθυστέρηση που επιβαρύνεται από τις εντολές το κύκλωμα, πρέπει να πολλαπλασιάσουμε τον αριθμό των κύκλων που χρειάζονται ώστε να εκτελεστούν οι πράξεις μέσα στον βρόγχο, με τον αριθμό των επαναλήψεων του βρόγχου καθώς να προσθέσουμε ακόμη έναν κύκλο που οφείλεται στην χρήση του βρόγχου.



2.7 Επαλήθευση RTL

Με την χρήση ενός αρχείου test bench που είναι γραμμένο σε γλώσσα υψηλού επιπέδου, μπορούμε να επαληθεύσουμε εάν λειτουργικά είναι ίδια το παραγόμενο κύκλωμα RTL με το πρωτότυπο. Το αρχείο test bench μπορεί να επαληθεύσει την έξοδο από την κύρια συνάρτηση που κάναμε σύνθεση, επιστρέφοντας μηδέν αν το RTL είναι λειτουργικά ίδιο με το πρωτότυπο. Χρησιμοποιείται στη C-προσομοίωση καθώς και στην C/RTL συν-προσομοίωση. Εάν το αποτέλεσμα που βγάλλει είναι διάφορο του μηδέν τότε το εργαλείο σύνθεσης υψηλού επιπέδου εμφανίζει μήνυμα ότι απέτυχε η προσομοίωση.

2.8 Προτάσεις

2.8.1 Κώδικας

Πρώτο μέλημα μας είναι να λιγοστεύσουμε τα δεδομένα που διαβάζει η κύρια συνάρτηση, δηλαδή τις βασικές εισόδους που χρειάζεται η συνάρτηση για να λειτουργήσει. Αυτά τα δεδομένα θα μεταφραστούν σε πύλες εισόδου και εξόδου στο κύκλωμα. Τα δεδομένα που έχει ο αλγόριθμος, τα οποία θα διαβαστούν σε ένα block της συσκευής, μπορούν εύκολα να τα διαχειριστούμε και να εκτελεστούν παράλληλα, αλλά οι πύλες εισόδου και εξόδου μπορούν να γίνουν ο λόγος καθυστέρησης του κυκλώματος. Για παράδειγμα, αν έχουμε δύο υλοποιήσεις όπου η πρώτη, διαβάζει την είσοδο από μια θύρα, ενώ η δεύτερη έχει αποθηκευμένο το περιεχόμενο της εισόδου μέσα στον κώδικα της, τότε η πρώτη υλοποίηση θα αναγκαστεί να περιμένει μέχρι να διαβαστεί η είσοδος, για να εκτελέσει τις επόμενες εντολές, δημιουργώντας καθυστέρηση στο κύκλωμα. Αντιθέτως, στην δεύτερη υλοποίηση δεν θα δημιουργηθεί αυτή η καθυστέρηση αφού όλα όσα χρειάζεται, περιλαμβάνονται μέσα στον κώδικα της καθώς οι εντολές της μπορεί να εκτελεστούν παράλληλα. Στην Εικόνα 8 βλέπουμε ένα κώδικα που λαμβάνει ως εισόδους δύο μεταβλητές Nr και Nk . Μεταφέροντας τις τιμές αυτών των μεταβλητών, στο κύριο κώδικα χωρίς να τις λαμβάνει ως είσοδο, η συνολική καθυστέρηση του κυκλώματος, από 3766 μειώθηκε στους 1976 κύκλους ρολογιού.



```
void Cipher(int Nr, int Nk, unsigned char out[16])
{
    int Rcon[255] = {0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20,
    0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f,
    0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d,
    0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61,
    0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d,
    0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10,
    0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a,
    0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3,
    0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd,
    0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83,
    0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08,
    0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d,
    0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4,
    0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3,
    0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc,
    0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04,
    0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab,
    0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a,
    0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4,
    0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66,
    0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02,
    0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8,
    0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35,
    0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72,
    0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33,
    0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb };
    int i,j,round=0; unsigned char state[4][4]; unsigned char
    RoundKey[240];
    unsigned char Key[32]; unsigned char in[16];
    unsigned char temp[16] = {0x00 ,0x01 ,0x02 ,0x03 ,0x04
    ,0x05 ,0x06 ,0x07 ,0x08 ,0x09 ,0x0a ,0x0b ,0x0c ,0x0d
    ,0x0e ,0x0f};
    unsigned char temp2[16] = {0x00 ,0x11 ,0x22 ,0x33 ,0x44
    ,0x55 ,0x66 ,0x77 ,0x88 ,0x99 ,0xaa ,0xbb ,0xcc ,0xdd
    ,0xee ,0xff};
    for(i=0;i<4*4;i++){Key[i]=temp[i];in[i]=temp2[i];}
    KeyExpansion(RoundKey,Key,Rcon,Nk,Nr);
    for(i=0;i<4;i++){for(j=0;j<4;j++){state[j][i] = in[i*4 + j];}}
    AddRoundKey(0,state,RoundKey);
    for(round=1;round<Nr;round++){SubBytes(state);ShiftRows(state);
    MixColumns(state);AddRoundKey(round,state,RoundKey);}
    SubBytes(state);ShiftRows(state);AddRoundKey(Nr,state,RoundKey)
    ;for(i=0;i<4;i++){for(j=0;j<4;j++){out[i*4+j]=state[j][i];}}}
```

Εικόνα 8: Αρχικός Κώδικας



```
void Cipher(unsigned char out[16])
{
    Int Nk=4;Int Nr=10;
    int Rcon[255] = {0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20,
    0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a,0x2f,
    0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d,
    0xfa, 0xef, 0xc5, 0x91, 0x39,0x72, 0xe4, 0xd3, 0xbd, 0x61,
    0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d,
    0x3a,0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10,
    0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8,0xab, 0x4d, 0x9a,
    0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3,
    0x7d, 0xfa, 0xef,0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd,
    0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc,0x83,
    0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08,
    0x10, 0x20, 0x40, 0x80, 0x1b,0x36, 0x6c, 0xd8, 0xab, 0x4d,
    0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4,
    0xb3,0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3,
    0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94,0x33, 0x66, 0xcc,
    0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04,
    0x08, 0x10, 0x20,0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab,
    0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35,0x6a,
    0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4,
    0xd3, 0xbd, 0x61, 0xc2, 0x9f,0x25, 0x4a, 0x94, 0x33, 0x66,
    0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02,
    0x04,0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8,
    0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63,0xc6, 0x97, 0x35,
    0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72,
    0xe4, 0xd3, 0xbd,0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33,
    0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb };
    int i,j,round=0;unsigned char state[4][4];unsigned char
    RoundKey[240];
    unsigned char Key[32];unsigned char in[16];
    unsigned char temp[16] = {0x00 ,0x01 ,0x02 ,0x03 ,0x04
    ,0x05 ,0x06 ,0x07 ,0x08 ,0x09 ,0x0a ,0x0b ,0x0c ,0x0d
    ,0x0e ,0x0f};
    unsigned char temp2[16]= {0x00 ,0x11 ,0x22 ,0x33 ,0x44
    ,0x55 ,0x66 ,0x77 ,0x88 ,0x99 ,0xaa ,0xbb ,0xcc ,0xdd
    ,0xee ,0xff};
    for(i=0;i<4*4;i++){Key[i]=temp[i];in[i]=temp2[i];}
    KeyExpansion(RoundKey,Key,Rcon,Nk,Nr);
    for(i=0;i<4;i++){for(j=0;j<4;j++){state[j][i] = in[i*4 + j];}}
    AddRoundKey(0,state,RoundKey);
    for(round=1;round<Nr;round++){SubBytes(state);ShiftRows(state);
    MixColumns(state);AddRoundKey(round,state,RoundKey);}
    SubBytes(state);ShiftRows(state);AddRoundKey(Nr,state,RoundKey)
    ;for(i=0;i<4;i++){for(j=0;j<4;j++){out[i*4+j]=state[j][i];}}}
```

Εικόνα 9: Τροποποιημένος Κώδικας



Εκτός από την μείωση των δεδομένων που χρησιμοποιούνται σαν είσοδο στην κύρια συνάρτηση, θα πρέπει να λιγοστεύσουμε τις προσπελάσεις που γίνονται σε πίνακες, ειδικά στα μεγάλους πίνακες. Οι πίνακες μετά την σύνθεση υλοποιούνται σε Block RAM οι οποίες όπως και οι πύλες εισόδου, εξόδου είναι περιορισμένα σε αριθμό και μπορεί να καθυστερήσουν το κύκλωμα. Ωστόσο, οι πίνακες μπορούν να κατανεμηθούν σε μικρότερους πίνακες ή ακόμη και σε καταχωρητές. Στην Εικόνα 10, βλέπουμε έναν πίνακα που χρησιμοποιείται στον κώδικα της Εικόνας 9. Κατανέμοντας αυτόν τον πίνακα σε μικρότερους με την χρήση Directive ARRAY PARTITION, μπορέσαμε να μειώσουμε την καθυστέρηση που επιβαρύνεται το κύκλωμα από 1976 στους 1816 κύκλους ρολογιού.

```
int sbox[256] = {
//0      1      2      3      4      5      6      7      8      9      A
B      C      D      E      F
0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67,
0x2b, 0xfe, 0xd7, 0xab, 0x76, //00xca, 0x82, 0xc9, 0x7d, 0xfa,
0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
//10xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5,
0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, //20x04, 0xc7, 0x23, 0xc3,
0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2,
0x75, //30x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52,
0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, //40x53, 0xd1, 0x00,
0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c,
0x58, 0xcf, //50xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85,
0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, //60x51, 0xa3,
0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10,
0xff, 0xf3, 0xd2, //70xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44,
0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73, //80x60,
0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14,
0xde, 0x5e, 0x0b, 0xdb, //90xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06,
0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
//A0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56,
0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, //B0xba, 0x78, 0x25, 0x2e,
0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b,
0x8a, //C0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61,
0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e, //D0xe1, 0xf8, 0x98,
0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55,
0x28, 0xdf, //E0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68,
0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16 };
#pragma HLS ARRAY_PARTITION variable=sbox complete factor=2
dim=1
```

Εικόνα 10: Κώδικας



Επιπλέον, πρέπει να επιδιώξουμε να εκτελούμε διακλαδώσεις υπό συνθήκη εντός σε εντολές που εκτελούνται με την χρήση διοχέτευσης. Οι συνθήκες θα υλοποιηθούν σε διαφορετικά μονοπάτια με την χρήση διοχέτευσης. Με το να επιτρέπεται στα δεδομένα από μια εντολή να χρησιμοποιούνται παράλληλα, καθώς εκτελείται η επόμενη εντολή η οποία είναι υπό συνθήκη, θα οδηγήσει στην αύξηση της απόδοσης του κυκλώματος. Για παράδειγμα, αν έχουμε δύο υλοποιήσεις όπου και οι δύο έχουν διακλαδώσεις υπό συνθήκη αλλά η πρώτη χρησιμοποιεί την μέθοδο της διοχέτευσης ενώ η δεύτερη όχι, τότε η πρώτη θα μπορέσει να χρησιμοποιήσει τις άλλες εντολές που είναι υπό συνθήκη, ενώ η δεύτερη θα αναγκαστεί να περιμένει μέχρι να ελεγχθεί η συνθήκη, ώστε να συνεχίσει στις επόμενες εντολές, επιβαρύνοντας την απόδοση του κυκλώματος. Στην Εικόνα 11, βλέπουμε έναν μέρος του κώδικα της Εικόνας 9, με την χρήση της διοχέτευσης στο συγκεκριμένο βρόγχο, μπορέσαμε να μειώσουμε την συνολική καθυστέρηση που επιβαρύνεται το κύκλωμα από 1816 σε 787 κύκλους ρολογιού

```
for (round=1; round<Nr; round++)  
#pragma HLS PIPELINE  
{  
SubBytes (state) ; ShiftRows (state) ;  
MixColumns (state) ;  
AddRoundKey (round, state, RoundKey) ;  
}
```

Εικόνα 11: Κώδικας

Εκτός από όλα αυτά πρέπει να λάβουμε υπόψιν και τις εξόδους. Τα δεδομένα που θα γράψει η κύρια συνάρτηση πρέπει να περιοριστούν για τον ίδιο λόγο που θα πρέπει και για τα δεδομένα που πρόκειται να διαβάσει[7]. Συνεπώς, αποτελεί παρόμοιο με το πρώτο παράδειγμα που σχετίζεται με τον κώδικα 9.



2.8.2 Γλώσσα Προγραμματισμού

Όπως αναφέρουμε πιο πάνω, το εργαλείο σύνθεσης υψηλού επιπέδου μπορεί να μετατρέψει σε γλώσσα περιγραφής υλικού τις γλώσσες C,C++ και SystemC.

Η Προτεινομένη γλώσσα αποτελεί η SystemC.

Οι βασικές διαφορές που έχει η SystemC με τις άλλες γλώσσες είναι ότι η SystemC είναι ειδικά διαμορφωμένη να σχεδιάζει προγράμματα σε επίπεδο συστήματος. Εκτός από τις βασικές κλάσεις που έχει η C++, παρέχει εργαλεία ώστε να προσομοιώνει διαδικασίες που εκτελούνται παράλληλα με την χρήση ρολογιών τα οποία μπορεί να εισάγει ο προγραμματιστής, στην υλοποίηση που πρόκειται να σχεδιάσει. Έκτος από όλα αυτά, παρέχει πύλες, σήματα, κανάλια και γεγονότα τα οποία υπάρχουν και σε μια γλώσσα περιγραφής υλικού[8].Ωστόσο, η πολυπλοκότητα της, την καθιστά δύσκολη για συγγραφή κώδικα.

2.8.3 Directives

Για να αυξήσουμε την απόδοση του κυκλώματος, που παράχθηκε μετά την σύνθεση μπορούμε να χρησιμοποιήσουμε διάφορους οδηγούς που μας παρέχει το εργαλείο σύνθεσης υψηλού επιπέδου. Αυτοί οι οδηγοί λέγονται directives και μπορούν να χρησιμοποιηθούν για την αύξηση της απόδοσης του συστήματος καθώς και για την μείωση των πόρων που χρησιμοποιούνται στο κύκλωμα.

Τα Directives που μπορούν να χρησιμοποιηθούν είναι τα παρακάτω:

ALLOCATION: Μπορούμε να ορίσουμε ένα αριθμητικό όριο για εκτέλεση πράξεων, συναρτήσεων καθώς και πυρήνων που χρησιμοποιούνται. Η χρήση αυτού του οδηγού μπορεί να οδηγήσει στην χρήση πιο πολλών πόρων του συστήματος με αποτέλεσμα να αυξηθεί η απόδοση του συστήματος.

ARRAY_MAP: Μπορούμε να συμπτύξουμε μικρότερους πίνακες σε ένα μεγαλύτερο ώστε να μειώσουμε την χρήση των Block RAM.

ARRAY_PARTITION: Μπορούμε να διαμερίσουμε μεγάλους πίνακες σε πολλούς μικρότερους και ακόμη σε καταχωρητές, ώστε να αυξήσουμε την προσπέλαση στα δεδομένα καθώς να μειώσουμε τις καθυστερήσεις που γίνονται από τις προσπελάσεις σε block RAM.

ARRAY_RESHAPE: Μπορούμε να μετατρέψουμε έναν πίνακα που μπορεί είτε να έχει ένα στοιχείο είτε πολλά, σε ένα πίνακα μεγαλύτερου μεγέθους. Χρήσιμο για να αυξήσουμε την προσπέλαση στην Block RAM χωρίς να χρησιμοποιήσουμε και άλλους πόρους.

CLOCK: Για τις υλοποιήσεις που έχουν γραφτεί σε SystemC, μπορούμε να φτιάξουμε νέα ρολόγια πάνω στα SC_MODULE χρησιμοποιώντας αυτό τον οδηγό.

DATA_PACK: Ομαδοποιεί τα ορίσματα από δομή σε scalar χρησιμοποιώντας μεταβλητή μεγαλύτερου μεγέθους.



DEPENDENCE: Χρησιμοποιείται ώστε να παρέχει επιπλέον πληροφορίες για τις εξαρτήσεις μεταξύ των βρόγχων, με σκοπό να τις ξεπεράσει, ώστε να επιτευχθεί η χρήση της μεθόδου διοχέτευσης σε αυτά.

EXPRESSION_BALANCE: Απενεργοποιεί την αυτόματη εξισορρόπηση των expression.

FUNCTION_INSTANTIATE: Επιτρέπει διαφορετικά instances της ίδιας συνάρτησης να βελτιστοποιηθούν.

INLINE: Κάνει inline σε μια συνάρτηση, ανεξαρτήτως την ιεραρχία που έχουν μεταξύ τους, οι συναρτήσεις. Χρησιμοποιείται ώστε να αυξήσει την απόδοση του συστήματος μειώνοντας τα function call overhead.

INTERFACE: Ορίζει πώς οι πύλες RTL έχουν δημιουργηθεί από την περιγραφή της συνάρτησης.

LATENCY: Μπορούμε να ορίσουμε την ελάχιστη και μέγιστη καθυστέρηση που δημιουργείται από τον συγκεκριμένο πόρο.

LOOP_FLATEN: Επιτρέπει τους εμφωλευμένους βρόγχους να εκτελεστούν σε έναν αλλά με μεγαλύτερη καθυστέρηση.

LOOP_MERGE: Ενώνει διαδοχικούς βρόγχους για μειώσει την καθυστέρηση, αυξάνοντας όμως το αριθμό των πόρων που χρησιμοποιούνται στο κύκλωμα.

LOOP_TRIPCOUNT: Χρησιμοποιείται στους βρόγχους, στους οποίους είναι μεταβλητό ο αριθμός των επαναλήψεων. Παρέχει μια εκτίμηση για το αριθμό των επαναλήψεων. Δεν έχει επίδραση στην σύνθεση, αλλά μόνο στην τελική αναφορά.

OCCURRENCE: Χρησιμοποιείται όταν χρησιμοποιούμε την μέθοδο διοχέτευσης σε συναρτήσεις, βρόγχους, για να προσδιορίσουμε, αν ο κώδικας στο συγκεκριμένο πεδίο χρησιμοποιείται σε χαμηλότερο ρυθμό σε σύγκριση με το υπόλοιπο κώδικα.

PIPELINE: Μειώνει το διάστημα έναρξης, επιτρέποντας την παράλληλη εκτέλεση εντολών που ανήκουν σε ένα βρόγχο ή σε μια συνάρτηση.

PROTOCOL: Με αυτήν την εντολή, μπορούμε να ορίσουμε ένα μέρος του κώδικα να είναι πρωτόκολλο region. Ένα protocol region μπορεί να χρησιμοποιηθεί ώστε να ορίσουμε ένα interface πρωτόκολλο.

RESET: Με αυτό τον οδηγό, μπορούμε να προσθέσουμε ή να αφαιρέσουμε σε μια μεταβλητή το state δηλαδή αν είναι καθολική ή στατική.

RESOURCE: Μπορούμε να ορίσουμε μια συγκεκριμένη βιβλιοθήκη να χρησιμοποιηθεί ώστε να υλοποιήσουμε μια μεταβλητή που μπορεί να είναι, είτε πίνακας είτε αριθμητικός τελεστής είτε όρισμα σε μια συνάρτηση, στο RTL κύκλωμα.

STREAM: Μπορούμε να ορίσουμε έναν συγκεκριμένο πίνακα να υλοποιηθεί με την χρήση FIFO ή RAM για βελτιστοποίηση του κυκλώματος.



TOP: Η κύρια συνάρτηση για σύνθεση ορίζεται στα project settings. Αυτός ο οδηγός μπορεί να χρησιμοποιηθεί ώστε να ορίσουμε οποιαδήποτε ως συνάρτηση για σύνθεση. Αυτό μας διευκολύνει ώστε να βρούμε την καλύτερη λύση για το κύκλωμα χωρίς να φτιάξουμε νέο project.

UNROLL: Ξετυλίγει τους βρόγχους τύπου for, ώστε να εκτελέσει τις εντολές που μπορεί να είναι ανεξάρτητες ή μία από την άλλη, σειριακά αλλά και επιπλέον μπορεί παράλληλα να χρησιμοποιηθεί μαζί με άλλο οδηγό.

Όλα αυτά που αναλύσαμε, πιο πάνω θα τα χρησιμοποιήσουμε στο δικό μας αλγόριθμο. Στην αρχή, αναλύουμε τις υλοποιήσεις που έχουμε φτιάξει για τον αλγόριθμο, καθώς σχολιάζουμε τους λόγους για τους οποίους είναι αποδοτικότερη ή μια από την άλλη. Στην συνέχεια, χρησιμοποιούμε διάφορες τεχνικές μεθόδους ώστε να βελτιστοποιήσουμε την απόδοση της πιο αργής υλοποίησης. Στο τέλος, αναλύουμε διάφορες τεχνικές - μεθόδους ώστε να διευκολύνουμε αυτούς που πρόκειται να χρησιμοποιήσουν το εργαλείο αυτό, να γράψουν αποδοτικότερες υλοποιήσεις.

Ο αλγόριθμός που έχουμε αποφασίσει να εξετάσουμε είναι αλγόριθμός κρυπτογράφησης AES.

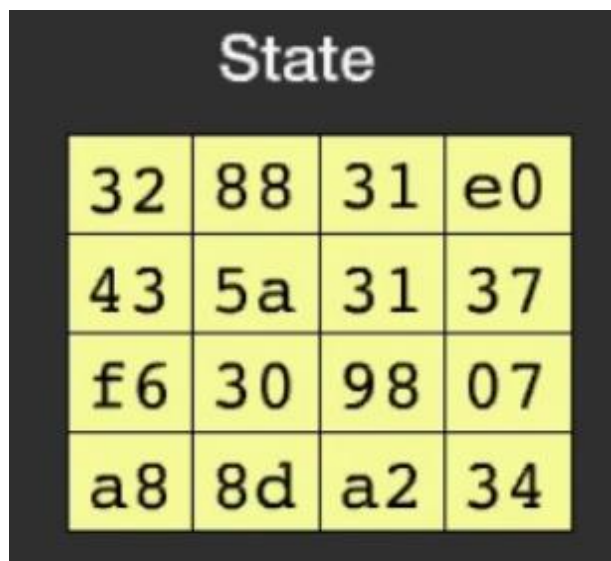
Κεφάλαιο 3^ο

3. Μελέτη του αλγορίθμου AES

Το όνομα του αλγορίθμου προέρχεται από τις λέξεις Advanced Encryption Standard. Το εναλλακτικό του όνομα είναι Rijndael και δημιουργήθηκε από δύο Βέλγους κρυπτογράφους το 2001. Χρησιμοποιήθηκε στην αρχή από την αμερικάνικη κυβέρνηση και τώρα χρησιμοποιείται διεθνώς από πολλούς οργανισμούς. Δημιουργήθηκε για να αντικαταστήσει έναν άλλο αλγόριθμο κρυπτογράφησης που είναι ο Data Encryption Standard (DES), ο οποίος δημοσιοποιήθηκε το 1977. Ο AES αποτελεί έναν συμμετρικό αλγόριθμο κρυπτογράφησης, δηλαδή χρησιμοποιεί το ίδιο κλειδί για να κρυπτογραφήσει και να αποκρυπτογραφήσει δεδομένα. Αποτελεί έναν πολύ ασφαλής αλγόριθμο κρυπτογράφησης αφού είναι ο πρώτος και ο μοναδικός που έχει εγκριθεί από την National Security Agency (NSA) για την ασφάλεια των δεδομένων της[1].

Ο αλγόριθμος αυτός χρειάζεται, όπως όλους τους άλλους αλγορίθμους κρυπτογράφησης ένα κείμενο (Plain text) καθώς και ένα κλειδί (Cipher Key) καθώς με την χρήση του κλειδιού αυτού θα γίνει η κρυπτογράφηση του κειμένου.

Ο αλγόριθμος χρησιμοποιεί κρυπτογράφηση ανά block, καθώς κρυπτογραφεί 16 bits την φορά όπως βλέπουμε στην Εικόνα 12.



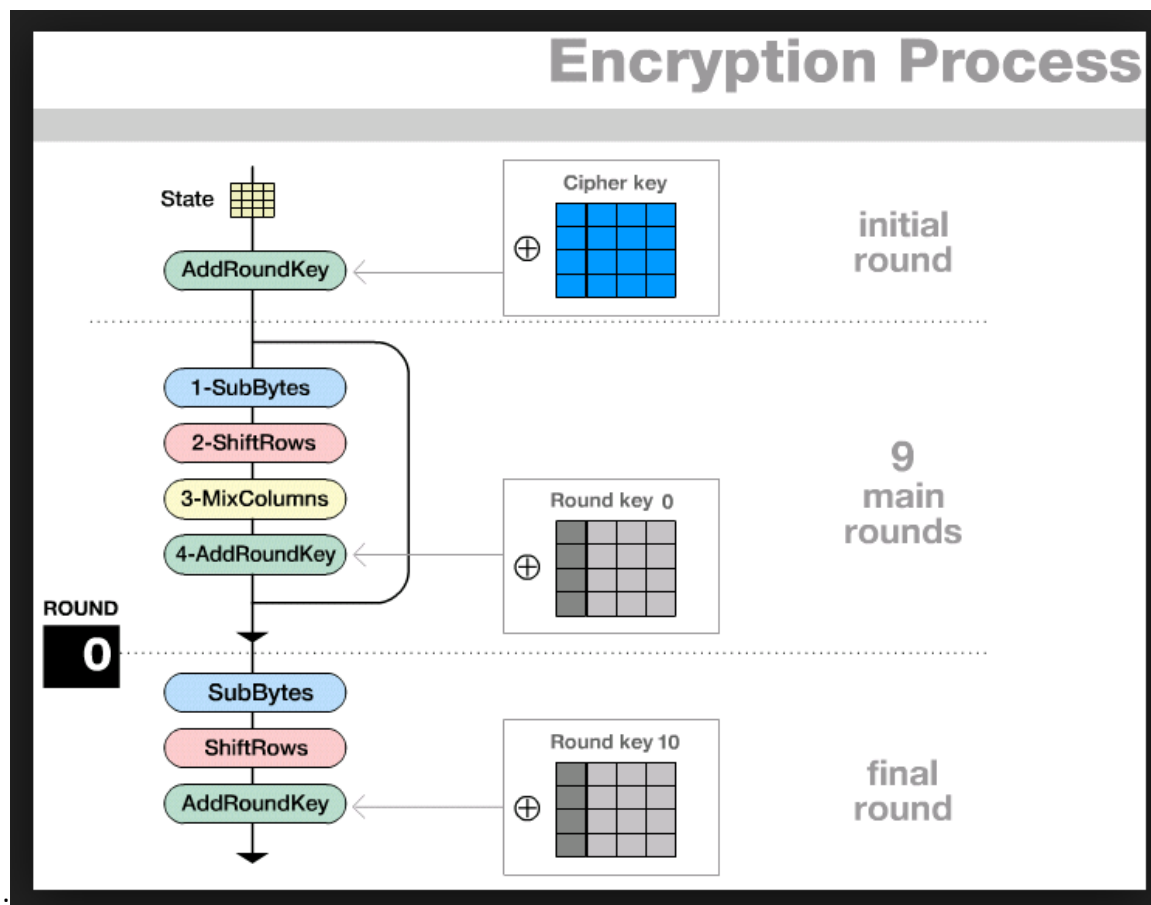
32	88	31	e0
43	5a	31	37
f6	30	98	07
a8	8d	a2	34

Εικόνα 12: Αρχικό κείμενο

Και το κλειδί θα είναι διαιρεμένο σε 16 bits την φορά:

Cipher Key			
2b	28	ab	09
7e	ae	f7	cf
15	d2	15	4f
16	a6	88	3c

Εικόνα 13:Κλειδί κρυπτογράφησης



Εικόνα 14:Διαδικασία Κρυπτογράφησης



Όπως βλέπουμε και στην Εικόνα 14, ο αλγόριθμος AES για την κρυπτογράφηση του κειμένου εκτελεί τις ακόλουθες συναρτήσεις:

- AddRoundKey
- SubBytes
- MixColumns
- AddRoundkey

Υπάρχουν 3 είδη κατηγοριών που έχουμε σε αυτόν τον αλγόριθμο.

Να είναι το κείμενο (Plain Text):

1. 128 bits
2. 192 bits
3. 256 bits

Για τις παραπάνω κατηγορίες, οι αριθμοί των επαναλήψεων που βλέπουμε στην Εικόνα 14 αλλάζει.

Δηλαδή αν έχουμε 128 bits τότε θα γίνουν 10 επαναλήψεις, αν έχουμε 192 θα γίνουν 12 και αν έχουμε 256 θα γίνουν 14. Δηλαδή ο αριθμός εκτέλεσης των παραπάνω συναρτήσεων φαίνεται στον Πίνακα 1.

Αριθμός Bits:	128	192	256
AddRoundKey	12	14	16
SubBytes	11	13	15
MixColumns	10	12	14
ShiftRows	11	13	15

Πίνακας 1:Αριθμός Επαναλήψεων των Συναρτήσεων

3.1 Ανάλυση συναρτήσεων

1. SubBytes:

19	a0	9a	e9
3d	f4	c6	f8
e3	e2	8d	48
be	2b	2a	08

		y															
hex		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	16	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	d1
	f	8c	a1	89	0d	b7	ef	e6	42	68	41	99	2d	0f	b0	54	bb

Εικόνα 15:Εκτέλεση συνάρτησης SubBytes

Καθώς έχουμε το παραπάνω block ή αλλιώς state, παίρνουμε και αντικαθιστούμε το κάθε bit με την τιμή που έχει στον άσπρο πίνακα που λέγεται S-Box. Αυτό το κάνουμε σε όλα τα bits του block.

d4	e0	b8	1e
27	bf	b4	41
11	98	5d	52
ae	f1	e5	30

Εικόνα 16:Τροποποιημένο κείμενο

2. ShiftRows:

Στην συγκεκριμένη διαδικασία παίρνουμε την δεύτερη σειρά του block και την κάνουμε rotate κατά ένα byte. Στην τρίτη σειρά θα την κάνουμε rotate ανά 2 byte ενώ στο τελευταία σειρά θα την κάνουμε rotate ανά 3 byte όπως φαίνεται στην Εικόνα 17.

d4	e0	b8	1e
bf	b4	41	27
5d	52	11	98
30	ae	f1	e5

Εικόνα 17:Εκτέλεση συνάρτησης ShiftRows

3. MixColumns:

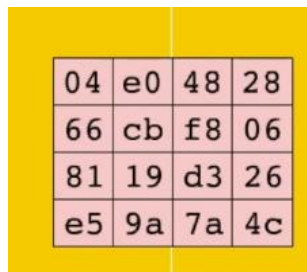
Στην Εικόνα 18 απεικονίζεται η διαδικασία που παίρνουμε την πρώτη στήλη του block και την πολλαπλασιάζουμε με έναν συγκεκριμένο πίνακα.

$$\begin{bmatrix} e0 & b8 & 1e \\ b4 & 41 & 27 \\ 52 & 11 & 98 \\ ae & f1 & e5 \end{bmatrix} \cdot \begin{bmatrix} d4 \\ bf \\ 5d \\ 30 \end{bmatrix} = \begin{bmatrix} 04 \\ 66 \\ 81 \\ e5 \end{bmatrix}$$

The diagram illustrates the MixColumns operation. On the left, a 4x3 matrix of bytes is shown. This matrix is multiplied (indicated by a dot) by a 4x1 column vector of bytes. The result is a 4x1 column vector of bytes shown on the right. The multiplication is performed using a specific 4x4 matrix of constants, which is shown in the middle of the equation.

Εικόνα 18:Εκτέλεση συνάρτησης MixColumns

Συνεχίζουμε και με τις άλλες στήλες, καθώς βλέπουμε στην Εικόνα 19 το τροποποιημένο κείμενο.




04	e0	48	28
66	cb	f8	06
81	19	d3	26
e5	9a	7a	4c

Εικόνα 19:Τροποποιημένο κείμενο

4. AddRoundkey:

Σε αυτήν την συνάρτηση παίρνουμε το κλειδί (θα δείξουμε μετά πως υπολογίζεται) και το προσθέτουμε στο state που έχουμε.

Πχ . Έστω το state που βλέπουμε στην Εικόνα 20.



04	e0	48	28
66	cb	f8	06
81	19	d3	26
e5	9a	7a	4c

Εικόνα 20:Τροποποιημένο κείμενο

Στην Εικόνα 13 έχουμε το κλειδί καθώς με πρόσθεση του στο κείμενο θα παραχθεί αυτό που φαίνεται στην Εικόνα 21.

a4	68	6b	02
9c	9f	5b	6a
7f	35	ea	50
f2	2b	43	49

Εικόνα 21:Τροποποιημένο κείμενο

5. Υπολογισμός παραγόμενου κλειδιού

Για την παραγωγή κλειδιών θα χρησιμοποιήσουμε το αρχικό κλειδί που δίνει ο χρήστης καθώς και τους πίνακες S-Box και Rcon όπως φαίνονται αντίστοιχα στις Εικόνες 13, 22 και 23.

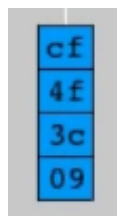
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Εικόνα 22:Πίνακας S-Box

01	02	04	08	10	20	40	80	1b	36
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00

Εικόνα 23:Πίνακας Rcon

Αρχικά, παίρνουμε την τελευταία στήλη του κλειδιού κρυπτογράφησης, κάνουμε ένα rotate προς τα πάνω, όπως φαίνεται στην Εικόνα 24:



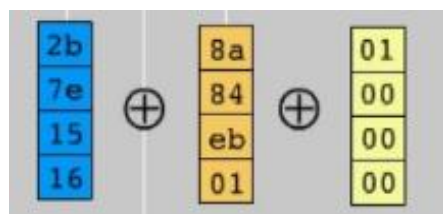
Εικόνα 24:Τελευταία στήλη του Πίνακα

Δεύτερον, αντικαθιστούμε τα bits που έχουμε, με τα bits του πίνακα S-Box ώστε να γίνει η στήλη, όπως φαίνεται στην Εικόνα 25:



Εικόνα 25:Τροποποιημένη στήλη

Τρίτον, προσθέτουμε την 1^η στήλη του κλειδιού κρυπτογράφησης με την στήλη που παράξαμε στο 2^ο βήμα, καθώς και με την 1^η στήλη του πίνακα Rcon, όπως φαίνεται στην Εικόνα 26:



Εικόνα 26:Πράξη στηλών

Στην Εικόνα 27 φαίνεται το αποτέλεσμα της πρόσθεσης που είναι η πρώτη στήλη του νέου κλειδιού.

a0
fa
fe
17

Εικόνα 27:Πρώτη στήλη του νέου κλειδιού

Για να υπολογίσουμε τις άλλες στήλες θα πρέπει να προσθέσουμε την παραπάνω στήλη που παράξαμε με τις αντίστοιχη 2^η, 3^η και 4^η στήλη του αρχικού κλειδιού κρυπτογράφησης καθώς το αποτέλεσμα που θα μας βγάλει φαίνεται στην Εικόνα 28:

a0	88	23	2a
fa	54	a3	6c
fe	2c	39	76
17	b1	39	05

Εικόνα 28:Νέο κλειδί

Για να φτιάξουμε το 2^ο νέο κλειδί ακολουθεί η ίδια περίπου διαδικασία , καθώς οι διαφορές που συναντάμε είναι ότι αντί να πάρουμε το block του κλειδιού κρυπτογράφησης που έβαλε ο χρήστης, θα πάρουμε το block του νέου κλειδιού, καθώς θα προσθέσουμε σε αυτό όχι την 1^η στήλη του Rcon αλλά την 2^η .



Κεφάλαιο 4^ο

4. Ανάλυση υλοποιήσεων

Σε αυτό το κεφάλαιο, θα αναλύσουμε τις τρεις υλοποιήσεις που έχουμε βρει για τον αλγόριθμο AES.

Η γλώσσα προγραμματισμού που χρησιμοποιούμε για τις τρεις υλοποιήσεις είναι η γλώσσα C. Η γλώσσα C είναι πιο φιλική προς τον χρήστη σε σύγκριση με την SystemC καθώς έχει λιγότερη πολυπλοκότητα, όσον αφορά την συγγραφή κώδικα. Για την πρώτη υλοποίηση χρησιμοποιήσαμε τον κώδικα Core[10], για την δεύτερη τον κώδικα Niyaz[12] και για την τρίτη τον κώδικα ChStone[11]. Φυσικά, έπρεπε να κάνουμε κάποιες τροποποιήσεις στους παραπάνω κώδικες για γίνουν συνθέσιμοι.

Αυτές οι τροποποιήσεις γίνονται σύμφωνα με το εγχειρίδιο του εργαλείου HLS[3]. Με βάση αυτό μπορέσαμε να καταλάβουμε ποιες εντολές είναι συνθέσιμες καθώς με ποιον τρόπο θα μπορούμε να γράψουμε κώδικα ώστε να μετατραπεί εξ ολοκλήρου σε RTL μορφή.

4.1 Τροποποιήσεις υλοποιήσεων για σύνθεση

4.1.1 Κώδικας Core

Όσον αφορά τον κώδικα Core:

Δεν κάνουμε σημαντικές αλλαγές στον κώδικα καθώς είναι σχετικά έτοιμος για σύνθεση. Κάποιες χρήσιμες αλλαγές που πρέπει να κάνουμε, αφορούν την μνήμη. Τα σφάλματα που συναντήσαμε κατά την σύνθεση του κώδικα Core φαίνονται στην Εικόνα 29.

```
ERROR: [SYNCHK 200-61] ../NotEdited/1st/aes.c:119:  
unsupported memory access on variable 'key' which is (or  
contains) an array with unknown size at compile time.  
ERROR: [SYNCHK 200-61] ../NotEdited/1st/aes.c:121:  
unsupported memory access on variable 'state' which is (or  
contains) an array with unknown size at compile time.
```

Εικόνα 29: Σφάλματα κατά την σύνθεση



```
void encrypt_128_key_expand_inline(word state[], word key[])
```

Εικόνα 30: Αρχικός Κώδικας

Για να το λύσουμε αυτά, κάναμε τις αλλαγές που φαίνονται στην Εικόνα 31.

```
void encrypt_128_key_expand_inline(word state[4], word  
key[4])
```

Εικόνα 31: Τροποποιημένος Κώδικας

Δηλαδή καθορίσαμε το μέγεθος της μνήμης των μεταβλητών state και key, για το μέγεθος που ορίζεται παρακάτω, δηλαδή 4. Μετά αυτές τις αλλαγές μπορέσαμε να κάνουμε σύνθεση το κύκλωμα σε RTL μορφή.

Στην συνέχεια, αφαιρούμε κάποιες συναρτήσεις τις οποίες δεν χρειαζόμαστε για την σύνθεση. Πιο συγκεκριμένα ο συγκεκριμένος κώδικας περιλαμβάνει 4 συναρτήσεις που είναι οι παρακάτω:

1. encrypt_128_key_expand_inline
2. encrypt_128_key_expand_inline_no_branch
3. encrypt_192_key_expand_inline_no_branch
4. encrypt_256_key_expand_inline_no_branch

Επειδή και στις άλλες υλοποιήσεις, θα εξετάσουμε μόνο την περίπτωση κρυπτογράφησης 128 bit, από τις τέσσερις θα χρειαστούμε τις δύο πρώτες. Στην συνέχεια, επειδή ζητούμε την υλοποίηση με το καλύτερη απόδοση, θα διαλέξουμε την δεύτερη συνάρτηση καθώς δεν χρησιμοποιεί διακλαδώσεις.

Άρα για τον κώδικα Core θα χρησιμοποιήσουμε μόνο την συνάρτηση **encrypt_128_key_expand_inline_no_branch()** και συνεπώς βγάζουμε τις υπόλοιπες. Η καθυστέρηση που προκύπτει από την σύνθεση της παραπάνω συνάρτησης είναι 51 κύκλοι, άρα μετά από 51 κύκλους ρολογιού θα γίνει κρυπτογράφηση του κειμένου. Όταν λέμε κύκλο ρολογιού, εννοούμε το χρονικό περιθώριο που χρειάζεται να αλλάξει μια κατάσταση σε μια συσκευή από 0 σε 1 ή μπορούμε να το ορίσουμε, ως την χρονική περίοδο μεταξύ δυο παλμών του ρολογιού που έχει η συσκευή που έχουμε, στην περίπτωση μας, μια συσκευή FPGA. Στο 2^ο πίνακα βλέπουμε τους πόρους που δεσμεύει το παραπάνω κύκλωμα.



Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	1697	2017	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	32	-	0	0	-
Multiplexer	-	-	-	573	-
Register	-	-	915	-	-
Total	32	0	2612	2590	0
Available	624	1728	460800	230400	96
Utilization (%)	5	0	~0	1	0

Πίνακας 2: Πίνακας Δέσμευσης Πόρων

4.1.2 Κώδικας Niyaz

Όσον αφορά τον κώδικα Niyaz:

Απ' ότι βλέπουμε, ο κώδικας αυτός εμπεριέχει την συνάρτηση main. Η συνάρτηση main πρέπει να βρίσκεται μόνο στο αρχείο test bench. Για αυτόν τον λόγο, χωρίζουμε το κώδικα σε δύο αρχεία που θα είναι το αρχείο test bench και το αρχείο AES_Encrypt.c. Στο αρχείο test bench θα μπει η συνάρτηση main, καθώς στο AES_Encrypt.c θα μπου οι υπόλοιπες συναρτήσεις. Επιπλέον, πρέπει να ορίσουμε την συνάρτηση για σύνθεση, η οποία θα είναι η **Cipher()**. Στην συνέχεια, βλέπουμε ότι αυτή η υλοποίηση περιλαμβάνει διάφορες εντολές του συστήματος όπως είναι η printf καθώς και scanf. Όμως, από τα παραπάνω κεφάλαια βλέπουμε ότι δεν μπορούν να μετατραπούν αυτά σε RTL μορφή, και για αυτόν τον λόγο μπαίνουν στο αρχείο test bench. Επιπλέον, ο κώδικας περιλαμβάνει πολλές global μεταβλητές όπως φαίνεται στην Εικόνα 32.

```
WARNING: [RTGEN 206-101] Global array 'state' will not be exposed as RTL port.
WARNING: [RTGEN 206-101] Register 'Nr' is power-on initialization.
WARNING: [RTGEN 206-101] Global scalar 'Nr' will not be exposed as RTL port.
WARNING: [RTGEN 206-101] Global array 'out_r' will not be exposed as RTL port.
```

Εικόνα 30: Προειδοποιήσεις κατά την σύνθεση

Για να διορθώσουμε τα παραπάνω προειδοποιήσεις, πρέπει να εκχωρήσουμε στις διάφορες συναρτήσεις ως ορίσματα, τις μεταβλητές που είναι global όπως φαίνεται στην Εικόνα 33.

```
void Cipher(int Nr,int Nk,unsigned char out[16])
void MixColumns(unsigned char state[4][4])
void ShiftRows(unsigned char state[4][4])
void AddRoundKey(int round,unsigned char
state[4][4],unsigned char RoundKey[240])
void SubBytes(unsigned char[4][4])
void KeyExpansion(unsigned char RoundKey[240],unsigned char
Key[32],int Rcon[255],int Nk,int Nr)
```

Εικόνα 31: Ορίσματα Συναρτήσεων



Κάνοντας αυτό σε όλες τις συναρτήσεις, δεν χρειάζεται να ορίσουμε στην αρχή τις global μεταβλητές και έτσι αποφεύγουμε το πρόβλημα που παρουσιάστηκε. Λύνοντας αυτό το πρόβλημα, σιγουρέψαμε ότι οι μεταβλητές αυτές, που χρειάζονται για το αλγόριθμο θα προσπελαθούν σωστά, ώστε να βγάλουμε στο τέλος, τα σωστά συμπεράσματα. Εκτός από όλα αυτά, πρέπει να σιγουρέψουμε ότι η συνάρτηση που πρόκειται να κάνουμε σύνθεση, περιλαμβάνει όλες τις υπό-συναρτήσεις που χρειαζόμαστε. Για αυτόν τον λόγο προσθέσαμε την συνάρτηση **KeyExpansion()** κάτω από την ιεραρχία της συνάρτησης **Cipher()** όπως φαίνεται στην Εικόνα 34.

```
● KeyExpansion() : void  
> ● Cipher() : void
```

Εικόνα 32:Ιεραρχία Συναρτήσεων

Δηλαδή βλέπουμε στην Εικόνα 34 ότι η **Cipher()** καλεί την **KeyExpansion()**.Άρα μαζί με την σύνθεση της συνάρτησης **Cipher()**,θα γίνει σύνθεση και η **KeyExpansion()**.Επιπλέον, βλέπουμε ότι και οι άλλες συναρτήσεις καλούνται από την **Cipher()** όπως φαίνεται στην Εικόνα 35.Μετά από αυτές τις αλλαγές μπορέσαμε να κάνουμε επιτυχή σύνθεση.

```
● ShiftRows(unsigned char (*)[4]) : void  
> ● Cipher(unsigned char *) : void (2 matches)  
  
● MixColumns(unsigned char (*)[4]) : void  
> ● Cipher(unsigned char *) : void  
  
● SubBytes(unsigned char (*)[4]) : void  
> ● Cipher(unsigned char *) : void (2 matches)  
  
● AddRoundKey(int, unsigned char (*)[4], unsigned char *) : void  
> ● Cipher(unsigned char *) : void (3 matches)
```

Εικόνα 33:Ιεραρχία Συναρτήσεων

Δυστυχώς όμως, επειδή είναι μεταβλητός ο αριθμός των επαναλήψεων που εκτελείται ο βρόγχος της κύριας συνάρτησής **Cipher()** δεν μπορεί να υπολογιστεί η καθυστέρηση. Για αυτόν τον λόγο θα χρησιμοποιήσουμε την συν-προσομοίωση ώστε να υπολογίσουμε την καθυστέρηση που επιβαρύνεται η απόδοση του κυκλώματος από την σύνθεση της παραπάνω συνάρτησης, η οποία είναι 3766 κύκλους ρολογιού.

Ο λόγος που χρησιμοποιούμε την συν-προσομοίωση, είναι για να βρούμε την τελική καθυστέρηση που επιβαρύνεται το κύκλωμα. Πιο συγκεκριμένα, κατά την διάρκεια της προσομοίωσης, το πρόγραμμα διαβάζει την μεταβλητή που καθορίζει τον αριθμό των επαναλήψεων του κύριου βρόγχου, με αποτέλεσμα να μπορέσει να υπολογίσει την καθυστέρηση. Αντιθέτως, επειδή ο αριθμός των επαναλήψεων, πιο συγκεκριμένα το N_r και το N_k , τα λαμβάνει από το αρχείο test bench, στην σύνθεση δεν μπορεί να υπολογιστεί η καθυστέρηση. Επειδή, όμως εμείς θα ασχοληθούμε με τον αλγόριθμο AES για 128 bit, ο αριθμός N_k , N_r είναι στατικός.



Αντικαθιστώντας τα N_k , N_r με τις τιμές που λαμβάνουν δηλαδή 4 και 10 αντιστοίχως, η συνολική καθυστέρηση που επιβαρύνεται η απόδοση του κυκλώματος έφθασε στους 1976 κύκλους ρολογιού δηλαδή μειώθηκε κατά 47.53%.

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	224	302	-
FIFO	-	-	-	-	-
Instance	4	-	563	770	-
Memory	2	-	20	5	-
Multiplexer	-	-	-	712	-
Register	-	-	233	-	-
Total	6	0	1040	1789	0
Available	624	1728	460800	230400	96
Utilization (%)	~0	0	~0	~0	0

Πίνακας 3: Πίνακας Δέσμευσης Πόρων

4.1.3 Κώδικας ChStone

Όσον αφορά τον κώδικα ChStone:

Από τι βλέπουμε, περιλαμβάνει πέντε αρχεία.

Τα αρχεία είναι τα παρακάτω:

1. aes.h
2. aes_dec.c
3. aes_enc.c
4. aes_func.c
5. aes_key.c

Από όλα αυτά, δεν θα χρειαστούμε το αρχείο aes_dec.c καθώς εμείς μελετάμε την αλγόριθμο όσον αφορά την κρυπτογράφηση. Όλα τα υπόλοιπα αρχεία περιλαμβάνουν συναρτήσεις που θα τις χρειαστούμε, γιατί μέσω αυτών θα δουλέψει σωστά ο παραπάνω αλγόριθμος. Αρχικά πρέπει να διαλέξουμε την συνάρτηση που θα γίνει σύνθεση και αυτή θα είναι η **encrypt()**. Στην συνέχεια, πρέπει να σιγουρέψουμε ότι όλες οι συναρτήσεις που θα χρησιμοποιήσουμε για την υλοποίηση του αλγορίθμου, ανήκουν στην ιεραρχία της συνάρτησης που πρόκειται να γίνει σύνθεση, όπως φαίνεται στην Εικόνα 36.



```

KeySchedule(int, int *) : int
> encrypt(int *, int *, int, int *) : int
ByteSub_ShiftRow(int *, int) : void
> encrypt(int *, int *, int, int *) : int (2 matches)
AddRoundKey(int *, int, int) : int
> encrypt(int *, int *, int, int *) : int (2 matches)
MixColumn_AddRoundKey(int *, int, int) : int
> encrypt(int *, int *, int, int *) : int

```

Εικόνα 34:Ιεραρχία Συναρτήσεων

Στην συνέχεια παρατηρούμε, ότι στα παραπάνω αρχεία υπάρχουν εντολές του συστήματος όπως η printf, τα οποία δεν μπορούν να μουν στην σύνθεση και για αυτόν τον λόγο τα βγάλαμε. Εκτός από όλα αυτά, είδαμε ότι υπάρχουν διάφορες μεταβλητές global, οι οποίες και αυτές πρέπει να μουν στην σύνθεση για να προσπελαθούν σωστά.

```

int type
int nb;
int round_val;
int key[32];
int statemt[32];
int word[4][120];

```

Εικόνα 35:Global Μεταβλητές

Για αυτόν τον λόγο, εισάγουμε τις global μεταβλητές στις συναρτήσεις που είναι κάτω από την ιεραρχία της κύριας που θα γίνει η σύνθεση, ώστε να γίνουν σύνθεση και αυτές. Τελευταία τροποποίηση που κάναμε, είναι να σιγουρέψουμε ότι έχουμε σβήσει τον κώδικα όσον αφορά την κρυπτογράφηση παραπάνω από 128 bit καθώς και την αποκρυπτογράφηση. Πέρα από όλα αυτά, δεν χρειάστηκε να κάνουμε καμία άλλη αλλαγή καθώς ο κώδικας είναι έτοιμος για σύνθεση. Η συνολική καθυστέρηση που επιβαρύνεται η απόδοση του κυκλώματος από την σύνθεση της παραπάνω συνάρτησης, είναι 1092 κύκλους ρολογιού καθώς οι πόροι που δεσμεύει φαίνονται στον Πίνακα 4.

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	37	26	-
FIFO	-	-	-	-	-
Instance	4	-	2060	3153	-
Memory	2	-	0	0	-
Multiplexer	-	-	-	371	-
Register	-	-	40	-	-
Total	6	0	2137	3550	0
Available	624	1728	460800	230400	96
Utilization (%)	~0	0	~0	1	0

Πίνακας 4:Πίνακας Δέσμευσης Πόρων



Στην συνέχεια, για να μπορέσουμε να κάνουμε την σύγκριση των κυκλωμάτων σωστά, θα πρέπει να δέχονται ίδια δεδομένα, από τα αρχεία test bench, δηλαδή να έχουν κοινά ορίσματα καθώς να βγάζουν μετά την σύνθεση παρόμοιες εισόδους και εξόδους. Στην αρχή θα δούμε, τι δεδομένα δέχονται οι συναρτήσεις που θα γίνουν σύνθεση από τα αρχεία test bench.

4.2 Ομογενοποίηση υλοποιήσεων

Δεδομένα που στέλνει το αρχείο test bench:

Κύκλωμα Core:

Όσον αφορά αυτό το κύκλωμα, η συνάρτηση που γίνεται σύνθεση είναι η **encrypt_128_key_expand_inline_no_branch()**.

Η συνάρτηση αυτή βλέπουμε ότι έχει ως ορίσματα τις μεταβλητές state και key, οι οποίες είναι τύπου word.

```
word state[4];  
word key[8];  
  
encrypt_128_key_expand_inline_no_branch(state, key);
```

Εικόνα 36:Κώδικας Core



Καθώς οι θύρες που παράγει το κύκλωμα φαίνονται στην Εικόνα 39.

state_address0	out	2
state_ce0	out	1
state_we0	out	1
state_d0	out	32
state_q0	in	32
state_address1	out	2
state_ce1	out	1
state_we1	out	1
state_d1	out	32
state_q1	in	32
key_address0	out	2
key_ce0	out	1
key_q0	in	32
key_address1	out	2
key_ce1	out	1
key_q1	in	32

Εικόνα 37:Θύρες

Κύκλωμα Niyaz:

Όσον αφορά αυτό το κύκλωμα , η συνάρτηση που γίνεται σύνθεση είναι η **cipher()**.Η συνάρτηση αυτή βλέπουμε ότι έχει ως ορίσματα την μεταβλητή out η οποία είναι τύπου unsigned char.

```
unsigned char out[16];  
void Cipher(unsigned char out[16];
```

Εικόνα 38:Κώδικας Niyaz

Καθώς οι θύρες που παράγει το κύκλωμα φαίνονται στην Εικόνα 41.

out_r_address0	out	4
out_r_ce0	out	1
out_r_we0	out	1
out_r_d0	out	8

Εικόνα 39:Θύρες



Κύκλωμα ChStone:

Όσον αφορά αυτό το κύκλωμα , η συνάρτηση που γίνεται σύνθεση είναι η **aes_main()**.

Η συνάρτηση αυτή βλέπουμε ότι έχει ως ορίσματα τις μεταβλητές statemt, key, out τα οποία είναι όλα τύπου int.

```
int out[32];  
int key[32];  
int statemt[32];  
  
aes_main(statemt, key, out);
```

Εικόνα 40:Κώδικας ChStone

Καθώς οι θύρες που παράγει το κύκλωμα φαίνονται στην Εικόνα 43.

statemt_address0	out	5
statemt_ce0	out	1
statemt_we0	out	1
statemt_d0	out	32
statemt_q0	in	32
statemt_address1	out	5
statemt_ce1	out	1
statemt_we1	out	1
statemt_d1	out	32
statemt_q1	in	32
key_address0	out	5
key_ce0	out	1
key_q0	in	32
out_r_address0	out	5
out_r_ce0	out	1
out_r_we0	out	1
out_r_d0	out	32

Εικόνα 41:Θύρες

Τώρα θα προσπαθήσουμε να δέχονται όλες οι συναρτήσεις, τις ίδιες εισόδους και εξόδους.

Για να γίνει αυτό, θα τροποποιήσουμε τα κυκλώματα Niyaz και ChStone ,ώστε να μοιάσουν με το πρώτο, αφού το πρώτο έχει την καλύτερη απόδοση. Άρα θα πρέπει να έχουν αυτά τα κυκλώματα στην κύρια συνάρτηση τους, ορίσματα τύπου word οι οποίες θα είναι οι μεταβλητές state και key.

4.2.1 Κύκλωμα Niyaz

Για να γίνει αυτό στο κύκλωμα Niyaz θα πρέπει να γίνουν οι εξής αλλαγές:

θα πρέπει να βάλουμε το κείμενο που πρόκειται να κρυπτογραφηθεί, καθώς το κλειδί μέσα στο αρχείο test bench. Στην αρχή, το κείμενο ήταν μέσα στην συνάρτηση που θα γίνει η σύνθεση. Έπειτα πρέπει να γίνουν τύπου word οι μεταβλητές in, key καθώς και το out.

```
word out[16];
word Key[32];
word in[16];

unsigned char temp[32] =
{0x00 ,0x01 ,0x02 ,0x03 ,0x04 ,0x05 ,0x06 ,0x07 ,0x
08 ,0x09 ,0x0a ,0x0b ,0x0c ,0x0d ,0x0e ,0x0f};
unsigned char temp2[32]=
{0x00 ,0x11 ,0x22 ,0x33 ,0x44 ,0x55 ,0x66 ,0x77 ,0x
88 ,0x99 ,0xaa ,0xbb ,0xcc ,0xdd ,0xee ,0xff};

for(i=0;i<4*4;i++)
{
    Key[i]=temp[i];
    in[i]=temp2[i];
}
```

Εικόνα 42:Κώδικας

```
word out[32];
word Key[32];
word in[32];

unsigned char in[16] =
{0x00 ,0x01 ,0x02 ,0x03 ,0x04 ,0x05 ,0x06 ,0x07 ,0x
08 ,0x09 ,0x0a ,0x0b ,0x0c ,0x0d ,0x0e ,0x0f};
unsigned char key[16]=
{0x00 ,0x11 ,0x22 ,0x33 ,0x44 ,0x55 ,0x66 ,0x77 ,0x
88 ,0x99 ,0xaa ,0xbb ,0xcc ,0xdd ,0xee ,0xff};
```

Εικόνα 43:Κώδικας

Έπειτα θα βάλουμε το περιεχόμενο των πινάκων temp και temp2, στα in και key χωρίς την χρήση βρόγχου όπως φαίνεται στην Εικόνα 45.

Στην συνέχεια, θα πρέπει να αντικαταστήσουμε την μεταβλητή out με μια μεταβλητή που θα την ονομάσουμε state και θα είναι τύπου word. Αυτή η αλλαγή πρέπει να γίνει στο αρχείο test bench αλλά και στο αρχείο AES_Encrypt.c.



Έπειτα, θα πάρουμε να πάρουμε τον κώδικα που μεταφέρει τον πίνακα in στον πίνακα state από το αρχείο AES_Encrypt.c και θα το βάλουμε στο αρχείο test bench.

```
word state[4][4];
for(int i=0;i<4;i++)
{
    For(int j=0;j<4;j++)
    {
        State[j][i]=in[i*4 + j];
    }
}
```

Εικόνα 44:Κώδικας

Τώρα, θα αλλάξουμε τα ορίσματα της συνάρτησης και θα βάλουμε αντί την μεταβλητή out, τις μεταβλητές state και key.

```
void Cipher(word Key[32],word state[4][4])
```

Εικόνα 45:Συνάρτηση Cipher

Και στην συνέχεια, θα μεταφέρουμε τον κώδικα που υπάρχει στο αρχείο AES_Encrypt.c, στο αρχείο test bench καθώς με την χρήση printf θα διαβάσουμε την μεταβλητή state που είναι το κρυπτογραφημένο μήνυμα.

```
for(int i=0;i<4;i++)
{
    for(int j=0;j<4;j++)
    {
        Printf("%02x",state[j][i]);
    }
    Printf("\n\n");
}
```

Εικόνα 46:Κώδικας

Στο τέλος, μετατρέπουμε τις μεταβλητές key και state σε τύπο word. Μετά από αυτές τις αλλαγές η καθυστέρηση που επιβαρύνεται η απόδοση του κυκλώματος έφθασε στους 1862 κύκλους ρολογιού καθώς βλέπουμε ότι τα Flip Flop αυξήθηκαν κατά 1424 καθώς και τα Look Up Tables αυξήθηκαν κατά 232.



Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	68	304	-
FIFO	-	-	-	-	-
Instance	4	-	947	1142	-
Memory	2	-	0	0	-
Multiplexer	-	-	-	575	-
Register	-	-	409	-	-
Total	6	0	1424	2021	0
Available	624	1728	460800	230400	96
Utilization (%)	~0	0	~0	~0	0

Πίνακας 5:Πίνακας Δέσμευσης Πόρων

4.2.2 Κύκλωμα ChStone

Οι αλλαγές που πρέπει να γίνουν ώστε να έχουμε τις ίδιες εισόδους και εξόδους με το κύκλωμα Core είναι οι εξής:

Καταρχάς θα κάνουμε όλες τις μεταβλητές εισόδου και εξόδου τύπου word.

```
word out[32];
word key[32];
word statemt[32];
```

Εικόνα 47:Μεταβλητές Εισόδου και Εξόδου

Όμως, επειδή έχουμε και μια άλλη μεταβλητή που λέγεται word , θα μετονομάσουμε αυτήν σε worda και θα την κάνουμε και αυτή τύπο word.

```
word worda[4][120];
```

Εικόνα 48:Μεταβλητή Word

Στην συνέχεια, θα βγάλουμε την μεταβλητή out και από τους δύο κώδικες και θα αφήσουμε μόνο την μεταβλητή statemt, και με βάση αυτό θα πάρουμε τον κώδικα που καταχωρεί το τελευταίο statemt στο out, καθώς αυτό θα εμφανίσουμε στην οθόνη.

```
for (int i = 0; i < 16; ++i)
{
    if (statemt[i] < 16)
        printf ("0");
    printf ("%x", statemt [i]);
}
printf ("\n");
```

Εικόνα 49:Κώδικας



Επειδή βλέπουμε στον κώδικα ότι η συνάρτηση `aes_main` ,καλεί μόνο την συνάρτηση `encrypt`.

```
void aes_main (word statemt[32], word key[32], word  
out[32])  
{  
    encrypt (statemt, key, 128128);  
}
```

Εικόνα 50:Συνάρτηση `aes_main`

Δεν χρειάζεται να υπάρχει ,καθώς θα βάλουμε το αρχείο `test bench` να καλεί αμέσως την συνάρτηση `encrypt()`.

Για λόγους ομοιομορφίας, θα μετονομάσουμε την μεταβλητή `statemt` σε `state`.

Μετά τις αλλαγές αυτές, η απόδοση του κυκλώματος παρέμεινε ίδια αλλά έχουν αλλάξει οι πόροι που χρησιμοποιεί, καθώς πλέον χρησιμοποιεί λιγότερους. Πιο συγκριμένα, από 2317 Flip-Flop που είχαμε, μειώθηκαν στα 1575 καθώς μειώθηκαν τα Look Up Tables από 3550 στα 3366.

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	17	12	-
FIFO	-	-	-	-	-
Instance	4	-	1534	3004	-
Memory	2	-	0	0	-
Multiplexer	-	-	-	350	-
Register	-	-	23	-	-
Total	6	0	1574	3366	0
Available	624	1728	460800	230400	96
Utilization (%)	~0	0	~0	1	0

Πίνακας 6:Πίνακας Δέσμευσης Πόρων



Μετά τις αλλαγές αυτές βλέπουμε στην Εικόνα 53 τις θύρες που έχει φτιάξει το πρόγραμμα για το κάθε κύκλωμα:

Κύκλωμα Core:

state_address0	out	2	ap_memory
state_ce0	out	1	ap_memory
state_we0	out	1	ap_memory
state_d0	out	32	ap_memory
state_q0	in	32	ap_memory
state_address1	out	2	ap_memory
state_ce1	out	1	ap_memory
state_we1	out	1	ap_memory
state_d1	out	32	ap_memory
state_q1	in	32	ap_memory
key_address0	out	2	ap_memory
key_ce0	out	1	ap_memory
key_q0	in	32	ap_memory
key_address1	out	2	ap_memory
key_ce1	out	1	ap_memory
key_q1	in	32	ap_memory

Κύκλωμα Niyaz:

state_address0	out	4	ap_memory
state_ce0	out	1	ap_memory
state_we0	out	1	ap_memory
state_d0	out	32	ap_memory
state_q0	in	32	ap_memory
state_address1	out	4	ap_memory
state_ce1	out	1	ap_memory
state_we1	out	1	ap_memory
state_d1	out	32	ap_memory
state_q1	in	32	ap_memory
Key_address0	out	5	ap_memory
Key_ce0	out	1	ap_memory
Key_q0	in	32	ap_memory
Key_address1	out	5	ap_memory
Key_ce1	out	1	ap_memory
Key_q1	in	32	ap_memory

Κύκλωμα ChStone:

state_address0	out	5	ap_memory
state_ce0	out	1	ap_memory
state_we0	out	1	ap_memory
state_d0	out	32	ap_memory
state_q0	in	32	ap_memory
state_address1	out	5	ap_memory
state_ce1	out	1	ap_memory
state_we1	out	1	ap_memory
state_d1	out	32	ap_memory
state_q1	in	32	ap_memory
key_address0	out	5	ap_memory
key_ce0	out	1	ap_memory
key_q0	in	32	ap_memory

Εικόνα 51:Θύρες

Η βασική διαφορά του 3^{ου} με τα υπόλοιπα είναι ότι αποθηκεύει το key σε ένα δυσδιάστατο πίνακα και όχι σε ένα μονοδιάστατο και για αυτόν τον λόγο δεν φτιάχνει και άλλη θύρα.

Πλέον οι τρείς υλοποιήσεις, εφόσον έχουν σχεδόν τις ίδιες εισόδους και εξόδους, μπορούμε να κάνουμε σωστή σύγκριση.

4.3 Σύγκριση των υλοποιήσεων

Όπως είδαμε στα προηγούμενα κεφάλαια, ο αλγόριθμος AES ξεκινάει από την συνάρτηση που δημιουργεί νέα κλειδιά και τα χρησιμοποιεί στο αρχικό κείμενο.

Εδώ Θα αναλύσουμε πως οι συναρτήσεις αυτές έχουν υλοποιηθεί στα τρία κυκλώματα που έχουμε.



Κύκλωμα Core:

Σε αυτό το κύκλωμα δεν έχει υλοποιηθεί καμία συνάρτηση που φτιάχνει και προσθέτει κλειδιά. Η πρόσθεση καθώς η δημιουργία κλειδιών γίνεται στις εντολές που φαίνονται στην Εικόνα 54.

Πρόσθεση κλειδιού:

```
word k0 = key[0], k1 = key[1], k2 = key[2], k3 = key[3];
state[0] ^= k0;
state[1] ^= k1;
state[2] ^= k2;
state[3] ^= k3;
```

Εικόνα 52:Κώδικας

Οι μεταβλητές κλειδί και state, διαβάζονται από την είσοδο, καθώς θα ξεκινήσουν στο 2^ο κύκλο. Επιπλέον, ο πίνακας Key θα χρειαστεί 2 κύκλους επειδή διαβάζει από την μνήμη.

Δημιουργία Κλειδιού:

Για την δημιουργία κλειδιού χρησιμοποιείται, κανονικά σύμφωνα με τον αλγόριθμο οι πίνακες S-Box και Rcon .Η διαδικασία είναι, ότι παίρνουμε την τελευταία στήλη του κλειδιού κρυπτογράφησης, την κάνουμε rotate, αντικαθιστούμε τα bits του μηνύματος, με τα bits του πίνακα S-Box.Στην συνέχεια, την προσθέτουμε, με την πρώτη στήλη του πίνακα Rcon και αυτό επαναλαμβάνεται μέχρι να φτιάξουμε όλο το νέο κλειδί.

Οι εντολές που γίνεται αυτή η διαδικασία φαίνονται στην Εικόνα 55.

```
word temp = k3;
rot_down_8(temp);
sub_byte(temp);
temp ^= rcon;
int j = (char)rcon;
j <= 1;
j ^= (j >> 8) & 0x1B; // if (rcon&0x80 != 0) then (j ^= 0x1B)
rcon = (byte)j;
k0 ^= temp;
k1 ^= k0;
k2 ^= k1;
k3 ^= k2;
```

Εικόνα 53:Κώδικας

Από το κώδικα βλέπουμε, ότι έχουμε τον πίνακα που Rcon που διαβάζει από την μνήμη ,καθώς αυτό επιβαρύνει κατά 2 κύκλους το συνολικό κύκλωμα.



Κύκλωμα Niyaz:

Σε αυτό το κύκλωμα, η συνάρτηση που δημιουργεί τα κλειδιά είναι η **KeyExpansion()** και η συνάρτηση τα προσθέτει στο κείμενο, είναι η **AddRoundKey()**.

KeyExpansion:

	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
● KeyExpansion	-	534	534	-	-
● Loop 1	no	12	-	3	4
> ● Loop 2	no	520	-	13	40

Πίνακας 7:Επιβάρυνση Συνάρτησης KeyExpansion

Ο 1ος βρόγχος επαναλαμβάνεται 4 φορές καθώς η κάθε επανάληψη επιβαρύνει το κύκλωμα κατά 3 κύκλους ρολογιού.

```
for (i=0; i<4; i++)  
{  
    RoundKey[i*4]=Key[i*4];  
    RoundKey[i*4+1]=Key[i*4+1];  
    RoundKey[i*4+2]=Key[i*4+2];  
    RoundKey[i*4+3]=Key[i*4+3];  
}
```

Εικόνα 54:Κώδικας

Στο παρόν κώδικα έχουμε 2 πίνακες που είναι το RoundKey που κάνει γράφει και το Key που κάνει διαβάζει την μνήμη.

Η καθυστέρηση, δημιουργείται επειδή διαβάζει από την μνήμη ο πίνακας Key. Το διάβασμα μνήμης χρειάζεται δύο κύκλους ρολογιού για να γίνει, ενώ οι άλλες πράξεις όπως πολλαπλασιασμό, πρόσθεση, εκχώρηση καθώς και το γράψιμο στην μνήμη γίνονται σε ένα κύκλο.

Ο 2ος βρόγχος επαναλαμβάνεται 40 φορές καθώς η κάθε επανάληψη επιβαρύνει το κύκλωμα κατά 13 κύκλους ρολογιού.



```
while (i < (4 * (10+1)))
{for(j=0;j<4;j++)
{temp[j]=RoundKey[(i-1) * 4 + j];}
if (i % Nk == 0){
// This function rotates the 4 bytes in a word to the
left once.// [a0,a1,a2,a3] becomes [a1,a2,a3,a0]//
Function RotWord(){k = temp[0];
temp[0] = temp[1];temp[1] = temp[2];temp[2] =
temp[3];temp[3] = k;
}// SubWord() is a function that takes a four-byte
input word and // applies the S-box to each of the four
bytes to produce an output word.// Function Subword(){
temp[0]=getSBoxValue(temp[0]);
temp[1]=getSBoxValue(temp[1])
temp[2]=getSBoxValue(temp[2]);
temp[3]=getSBoxValue(temp[3]);}
temp[0] = temp[0] ^ Rcon[i/4];}
else if (4 > 6 && i % 4 == 4)
{// Function Subword(){
temp[0]=getSBoxValue(temp[0]);
temp[1]=getSBoxValue(temp[1]);
temp[2]=getSBoxValue(temp[2]);
temp[3]=getSBoxValue(temp[3]);}}
RoundKey[i*4+0] = RoundKey[(i-4)*4+0] ^
temp[0];RoundKey[i*4+1] = RoundKey[(i-4)*4+1] ^
temp[1];
RoundKey[i*4+2] = RoundKey[(i-4)*4+2] ^ temp[2];
RoundKey[i*4+3] = RoundKey[(i-4)*4+3] ^ temp[3];
i++;}
```

Εικόνα 55: Κώδικας



Στο κώδικα της Εικόνας 57 έχουμε τέσσερις πίνακες που είναι το Key που θα γράψει, το RoundKey που θα διαβάσει και θα γράψει, καθώς και τα S-Box και Rcon που θα μόνο διαβάσουν την μνήμη. Από το παραπάνω βρόγχο που αναλύσαμε, εύκολα καταλαβαίνουμε, ότι θα υπάρχει μεγαλύτερη επιβάρυνση στην απόδοση του κυκλώματος καθώς έχουμε περισσότερους πίνακες που διαβάζουν την μνήμη.

Στο 1^ο βρόγχο βλέπουμε ότι κάνει μια πρόσθεση, ένα γινόμενο και μια σύγκριση καθώς όλα αυτά γίνονται σε 1 κύκλο ρολογιού.

Εκτός από αυτά, βλέπουμε από τις παρακάτω εντολές ότι διαβάζει μια φορά την μνήμη ο πίνακας RoundKey και συνεπώς ο αριθμός κύκλων που θα χρειαστεί όλο ο βρόγχος είναι δύο.

Στο επόμενο κύκλο ξεκινάει ο 2^{ος} βρόγχος οι οποίος εμπεριέχει έναν μεγάλο αριθμό από πίνακες, οι οποίοι επιβαρύνουν την απόδοση του κυκλώματος.

Στις παρακάτω εντολές χρειάζονται 2 κύκλους ρολογιού, έναν για καλέσει την συνάρτηση και έναν για να λάβει την τιμή από αυτή, με την χρήση του return.

```
temp[0]=getSBoxValue(temp[0]);  
temp[1]=getSBoxValue(temp[1]);  
temp[2]=getSBoxValue(temp[2]);  
temp[3]=getSBoxValue(temp[3]);
```

Εικόνα 56:Κώδικας

Στην συνέχεια του κώδικα, βλέπουμε την παρακάτω εντολή που θα ξεκινήσει όταν λάβει την τιμή από το getSBoxValue που όπως αναφέραμε πιο πριν χρειάζεται δύο κύκλους ρολογιού.

```
temp[0] = temp[0] ^ Rcon[i/4];
```

Εικόνα 57:Κώδικας

Συνεπώς, θα ξεκινήσει μετά από δύο κύκλους.

Εκτός από αυτά, έχουμε και πίνακες που διαβάζουν την μνήμη όπως φαίνεται στην Εικόνα 60.

```
RoundKey[i*4+0] = RoundKey[(i-Nk)*4+0] ^ temp[0];  
RoundKey[i*4+1] = RoundKey[(i-Nk)*4+1] ^ temp[1];  
RoundKey[i*4+2] = RoundKey[(i-Nk)*4+2] ^ temp[2];  
RoundKey[i*4+3] = RoundKey[(i-Nk)*4+3] ^ temp[3];
```

Εικόνα 58:Κώδικας

Επιπλέον, αυτά ανήκουν σε βρόγχο και είναι φυσιολογικό να επιβαρύνουν πιο πολύ το κύκλωμα.

Η συνάρτηση αυτή ξεκινάει στον 2^ο κύκλο, εφόσον περιμένει τα ορίσματα από την κύρια συνάρτηση. Συνολικά, επιβαρύνει την απόδοση του κυκλώματος κατά 534 κύκλους ρολογιού.

AddRoundKey:

	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
● AddRoundKey	-	41	41	-	-
> ● Loop 1	no	40	-	10	4

Εικόνα 59: Επιβάρυνση Συνάρτησης AddRoundKey

Ο 1ος βρόγχος επαναλαμβάνεται 4 φορές καθώς η κάθε επανάληψη επιβαρύνει την απόδοση του κυκλώματος κατά 10 κύκλους ρολογιού.

```
int i, j;
for (i=0; i<4; i++)
{
    for (j=0; j<4; j++)
    {
        state[j][i] ^= RoundKey[round * Nb * 4 + i * Nb + j];
    }
}
```

Εικόνα 60: Κώδικας

Στο παρόν κώδικα έχουμε 2 πίνακες που είναι το RoundKey που θα διαβάσει ενώ το state που θα κάνει γράψει στην μνήμη.

Από τον κώδικα καταλαβαίνουμε ότι, θα πρέπει να περιμένει ένα κύκλο ώστε να πάρει τα ορίσματα η συνάρτηση. Στην συνέχεια, στον επόμενο κύκλο θα ξεκινήσει ο 1^ο βρόγχος, καθώς, στον επόμενο δηλαδή στον 3^ο κύκλο θα ξεκινήσει ο εμφωλευμένος βρόγχος. Επειδή έχουμε διάβασμα μνήμη, οι εντολές του εμφωλευμένου βρόγχου θα τελειώσουν μετά από δύο κύκλους ρολογιού. Λόγο των επαναλήψεων των βρόγχων προκύπτει το iteration Latency που φαίνεται στην Εικόνα 61.

Κύκλωμα ChStone:

Σε αυτό το κύκλωμα, η συνάρτηση που δημιουργεί τα κλειδιά αποτελεί η **KeySchedule()** και η συνάρτηση που αλλάζει τα κλειδιά, είναι η **AddRoundKey()**.

KeySchedule:

	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
● KeySchedule	-	562	562	-	-
> ● Loop 1	no	40	-	10	4
> ● Loop 2	no	520	-	13	40

Εικόνα 61: Επιβάρυνση Συνάρτησης KeySchedule



Ο 1^ο βρόγχος επαναλαμβάνεται 4 φορές καθώς η κάθε επανάληψη του επιβαρύνει την απόδοση του κυκλώματος κατά 10 κύκλους. Ενώ ο δεύτερος, επαναλαμβάνεται 40 φορές καθώς η κάθε επανάληψη επιβαρύνει την απόδοση του κυκλώματος κατά 13 κύκλους ρολογιού.

```
int nk, nb, round_val;
int i, j, temp[4];
nk = 4;
nb = 4;
round_val = 10;
break;
for (j = 0; j < nk; ++j)
    for (i = 0; i < 4; ++i)
/* 0 word */
        word[i][j] = key[i + j * 4];
/* expanded key is generated */
for (j = nk; j < nb * (round_val + 1); ++j)
{
/* RotByte */
    if ((j % nk) == 0)
    {
temp[0] = SubByte (word[1][j - 1]) ^ Rcon0[(j / nk) - 1];
temp[1] = SubByte (word[2][j - 1]);
temp[2] = SubByte (word[3][j - 1]);
temp[3] = SubByte (word[0][j - 1]);
    }
    if ((j % nk) != 0)
    {
temp[0] = word[0][j - 1];
temp[1] = word[1][j - 1];
temp[2] = word[2][j - 1];
temp[3] = word[3][j - 1];
    }
    if (nk > 6 && j % nk == 4)
        for (i = 0; i < 4; ++i)
            temp[i] = SubByte (temp[i]);
    for (i = 0; i < 4; ++i)
        word[i][j] = word[i][j - nk] ^ temp[i];
}
```

Εικόνα 62:Κώδικας

Όπως και κάθε άλλη συνάρτηση, θα ξεκινήσει και αυτή στο 2^ο κύκλο εφόσον περιμένει ορίσματα. Το κακό με αυτήν την συνάρτηση είναι ότι περιλαμβάνει βρόγχους, οι οποίοι περιλαμβάνουν και άλλους εμφωλευμένους βρόγχους οι οποίοι επιβαρύνουν πιο πολύ το κύκλωμα.



Εκτός από όλα αυτά βλέπουμε ότι στο εμφωλευμένο βρόγχο, το key διαβάζει από την μνήμη, με αποτέλεσμα να επιβαρύνει το κύκλωμα κατά δύο κύκλους. Στην συνέχεια στο 2^ο εξωτερικό βρόγχο, βλέπουμε ότι καλείται η συνάρτηση **SubByte()** που και αυτή καταναλώνει 2 κύκλους, έναν για να καλέσει την συνάρτηση και έναν για επιστρέψει την τιμή της. Και τέλος βλέπουμε ότι και το worda διαβάζει από την μνήμη με αποτέλεσμα και αυτό να καταναλώνει 2 κύκλους ρολογιού.

AddRoundKey:

	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
● AddRoundKey	-	17	17	-	-
● Loop 1	no	16	-	4	4

Εικόνα 63: Επιβάρυνση Συνάρτησης AddRoundKey

Ο 1^{ος} βρόγχος επαναλαμβάνεται 4 φορές καθώς η κάθε επανάληψη επιβαρύνει την απόδοση του κυκλώματος κατά 4 κύκλους ρολογιού.

```
int j, nb;
nb=4;
for (j = 0; j < nb; ++j)
{
    statemt[j * 4] ^= word[0][j + nb * n];
    statemt[1 + j * 4] ^= word[1][j + nb * n];
    statemt[2 + j * 4] ^= word[2][j + nb * n];
    statemt[3 + j * 4] ^= word[3][j + nb * n];
}
```

Εικόνα 64:Κώδικας

Ο Κώδικας της Εικόνας 66 ξεκινάει στον επόμενο κύκλο επειδή καλείται από άλλη συνάρτηση, καθώς βλέπουμε ότι χρησιμοποιεί δύο μνήμες οι οποίες είναι η worda και η state, που την πρώτη την διαβάζει ,ενώ την δεύτερη την διαβάζει και γράφει σε αυτήν. Επειδή υπάρχει εξάρτηση μεταξύ των εντολών αυτών, είναι φυσιολογικό να μην μπορούν να εκτελεσθούν οι εντολές παράλληλα πράγμα που δικαιολογεί το iteration latency που βγάζει.

Στον Πίνακα 8 φαίνεται η επιβάρυνση που προκύπτει από τις συναρτήσεις.

	1 ^ο Κύκλωμα	2 ^ο κύκλωμα	3 ^ο κύκλωμα
Πρόσθεση Κλειδιού	4	41	17
Δημιουργία Κλειδιών	2	534	562
Συνολικό	6	575	589

Πίνακας 8: Επιβάρυνση Συναρτήσεων



Η επόμενη συνάρτηση που θα μελετήσουμε είναι η `SubBytes`, που αντικαθιστά τα στοιχεία του block με τα στοιχεία του S-box.

Κύκλωμα Core:

Σε αυτό το κύκλωμα η συνάρτηση η οποία αντικαθιστά το block με τα στοιχεία του πίνακα S-Box αποτελεί η `table_lookup` όπως φαίνεται στην Εικόνα 67.

```
b = (byte *)state; table_lookup; rot;
z0 ^= p0, z3 ^= p1, z2 ^= p2, z1 ^= p3;
b += 4; table_lookup; rot;
z1 ^= p0, z0 ^= p1, z3 ^= p2, z2 ^= p3;
b += 4; table_lookup; rot;
z2 ^= p0, z1 ^= p1, z0 ^= p2, z3 ^= p3;
b += 4; table_lookup; rot;
```

Εικόνα 65:Κώδικας

Η δομή της φαίνεται στην Εικόνα 68.

```
#define table_lookup { \
    p0 = t0[b[0]];      \
    p1 = t0[b[1]];      \
    p2 = t0[b[2]];      \
    p3 = t0[b[3]];      \
}
```

Εικόνα 66:Υλοποίηση Table_lookup

Αυτή η συνάρτηση από τι βλέπουμε, ο πίνακας `b[]` διαβάζει την μνήμη καθώς τον περιμένει να τελειώσει η μεταβλητή `t0`, με αποτέλεσμα, όπως και στα παραπάνω κυκλώματα να υπάρχει μια επιβάρυνση της απόδοσης κατά 2 κύκλους. Η διαφορά εδώ, είναι ότι δεν υπάρχει εξάρτηση, καθώς εκτελούνται όλα παράλληλα.

Κύκλωμα Niyaz:

Σε αυτό το κύκλωμα, η συνάρτηση που αντικαθιστά τα στοιχεία του κειμένου με τον πίνακα S-Box είναι η `SubBytes()`.

SubBytes:

	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
● SubBytes	-	57	57	-	-
▼ ● Loop 1	no	56	-	14	4
● Loop 1.1	no	12	-	3	4

Εικόνα 67: Επιβάρυνση Συνάρτησης SubBytes



Ο 1^{ος} βρόγχος επαναλαμβάνεται 4 φορές καθώς η κάθε επανάληψη επιβαρύνει το κύκλωμα κατά 14 κύκλους. Επιπλέον, βλέπουμε ότι περιλαμβάνει και ένα εμφωλευμένο βρόγχο ο οποίος εκτελείται 4 φορές καθώς επιβαρύνει το κύκλωμα κατά 3 κύκλους ανά επανάληψη. Άρα είναι φυσιολογικό να έχει τόση απόδοση όλο ο βρόγχος.

```
int i, j;  
for(i=0; i<4; i++)  
{  
    for(j=0; j<4; j++)  
    {  
        state[i][j] = getSBoxValue(state[i][j]);  
    }  
}
```

Εικόνα 68:Κώδικας

Βλέπουμε ότι ο εσωτερικός βρόγχος θα αρχίσει ένα κύκλο μετά, σε σύγκριση με το εξωτερικό. Έπειτα βλέπουμε ότι, ο πίνακας state διαβάζει την μνήμη καθώς για αυτήν την διαδικασία, χρησιμοποιεί δύο κύκλους και επιπλέον, βλέπουμε ότι το χρησιμοποιεί σαν όρισμα στην συνάρτηση **getSBoxValue()** με αποτέλεσμα να χρειάζεται άλλους δύο κύκλους.

Κύκλωμα ChStone:

Σε αυτό το κύκλωμα, η συνάρτηση η οποία αντικαθιστά το block με τα στοιχεία του πίνακα S-Box αποτελεί η συνάρτηση **SubBytes_ShiftRow()**. Επιπλέον, αυτή η συνάρτηση περιλαμβάνει και μια άλλη διαδικασία που κάνει ο αλγόριθμος που είναι το **shiftRow()** δηλαδή η ολίσθηση. Αλλά εμείς θα προσπαθήσουμε να βρούμε τους χρόνους εκτέλεσης μόνο της **SubBytes()**.

Για να γίνει όμως αυτό θα πρέπει να κάνουμε κάποιες αλλαγές στον κώδικα καθώς θα αφαιρέσουμε την **ShiftRows()** δηλαδή την συνάρτηση για την ολίσθηση. Η αλλαγή που χρειάστηκε να κάνουμε μόνο, ήταν να αφαιρέσουμε την ολίσθηση που γίνεται με την χρήση της πράξης shift right logical (>>).



```
temp = Sbox[statemt[1] >> 4][statemt[1] & 0xf];
statemt[1] = Sbox[statemt[5] >> 4][statemt[5] & 0xf];
statemt[5] = Sbox[statemt[9] >> 4][statemt[9] & 0xf];
statemt[9] = Sbox[statemt[13] >> 4][statemt[13] & 0xf];
statemt[13] = temp;
temp = Sbox[statemt[2] >> 4][statemt[2] & 0xf];
statemt[2] = Sbox[statemt[10] >> 4][statemt[10] & 0xf];
statemt[10] = temp;
temp = Sbox[statemt[6] >> 4][statemt[6] & 0xf];
statemt[6] = Sbox[statemt[14] >> 4][statemt[14] & 0xf];
statemt[14] = temp;
temp = Sbox[statemt[3] >> 4][statemt[3] & 0xf];
statemt[3] = Sbox[statemt[15] >> 4][statemt[15] & 0xf];
statemt[15] = Sbox[statemt[11] >> 4][statemt[11] & 0xf];
statemt[11] = Sbox[statemt[7] >> 4][statemt[7] & 0xf];
statemt[7] = temp;
statemt[0] = Sbox[statemt[0] >> 4][statemt[0] & 0xf];
statemt[4] = Sbox[statemt[4] >> 4][statemt[4] & 0xf];
statemt[8] = Sbox[statemt[8] >> 4][statemt[8] & 0xf];
statemt[12] = Sbox[statemt[12] >> 4][statemt[12] & 0xf];
```

Εικόνα 69: Αρχικός Κώδικας

```
temp = Sbox[statemt[1]][statemt[1] & 0xf];
statemt[1] = Sbox[statemt[5]][statemt[5] & 0xf];
statemt[5] = Sbox[statemt[9]][statemt[9] & 0xf];
statemt[9] = Sbox[statemt[13]][statemt[13] & 0xf];
statemt[13] = temp;
temp = Sbox[statemt[2]][statemt[2] & 0xf];
statemt[2] = Sbox[statemt[10]][statemt[10] & 0xf];
statemt[10] = temp;
temp = Sbox[statemt[6]][statemt[6] & 0xf];
statemt[6] = Sbox[statemt[14]][statemt[14] & 0xf];
statemt[14] = temp;
temp = Sbox[statemt[3]][statemt[3] & 0xf];
statemt[3] = Sbox[statemt[15]][statemt[15] & 0xf];
statemt[15] = Sbox[statemt[11]][statemt[11] & 0xf];
statemt[11] = Sbox[statemt[7]][statemt[7] & 0xf];
statemt[7] = temp;
statemt[0] = Sbox[statemt[0]][statemt[0] & 0xf];
statemt[4] = Sbox[statemt[4]][statemt[4] & 0xf];
statemt[8] = Sbox[statemt[8]][statemt[8] & 0xf];
statemt[12] = Sbox[statemt[12]][statemt[12] & 0xf];
```

Εικόνα 70: Τροποποιημένος Κώδικας



Παρατηρούμε ότι ο κώδικας που φαίνεται στην Εικόνα 72 περιλαμβάνει το πίνακα state, ο οποίος διαβάζει από την μνήμη και γράφει σε αυτήν, σε σύγκριση με τον πίνακα S-Box που διαβάζεται μόνο. Επίσης βλέπουμε ότι, για να διαβαστεί ο πίνακας S-Box, πρέπει να περιμένει να διαβαστεί ο πίνακας state. Όλη αυτή η εξάρτηση επιβαρύνει πολύ την απόδοση του κυκλώματος.

	Pipelined	Latency	Initiation Interval
● ByteSub_ShiftRow	-	15	15

Εικόνα 71: Επιβάρυνση Συνάρτησης ByteSub

Στον Πίνακα 9 φαίνεται η επιβάρυνση που προκύπτει από τις συναρτήσεις.

	1 ^ο Κύκλωμα	2 ^ο κύκλωμα	3 ^ο κύκλωμα
Πρόσθεση Κλειδιού	4	41	17
Δημιουργία Κλειδιών	2	534	562
Αντικατάσταση με Πίνακα	2	57	15
Συνολικό	8	632	604

Πίνακας 9: Επιβάρυνση Συναρτήσεων

Στην συνέχεια, θα αναλύσουμε τις επόμενες δύο μεθόδους του αλγορίθμου που έμειναν, που είναι η ολίσθηση και η ανάμιξη πινάκων.

Κύκλωμα Core:

Η ολίσθηση γίνεται εδώ από την συνάρτηση **rot** καθώς εκτελείται παράλληλα με τις άλλες συναρτήσεις, όπως φαίνεται στην Εικόνα 74.

```
b = (byte *)state; table_lookup; rot;
z0 ^= p0, z3 ^= p1, z2 ^= p2, z1 ^= p3;
b += 4; table_lookup; rot;
z1 ^= p0, z0 ^= p1, z3 ^= p2, z2 ^= p3;
b += 4; table_lookup; rot;
z2 ^= p0, z1 ^= p1, z0 ^= p2, z3 ^= p3;
b += 4; table_lookup; rot;
```

Εικόνα 72: Κώδικας



Ενώ η ανάμιξη πινάκων γίνεται μέσω της πράξης **xor**, που εκτελείται παράλληλα και αυτή με τις άλλες εντολές όπως φαίνεται στην Εικόνα 75.

Με αποτέλεσμα να γλιτώνουμε την καθυστέρηση η οποία θα είχε προκύψει.

```
b = (byte *)&k0;
b[0] ^= t[a[0]*4], b[1] ^= t[a[5]*4], b[2] ^= t[a[10]*4],
b[3] ^= t[a[15]*4];
b = (byte *)&k1;
b[0] ^= t[a[4]*4], b[1] ^= t[a[9]*4], b[2] ^= t[a[14]*4],
b[3] ^= t[a[3]*4];
b = (byte *)&k2;
b[0] ^= t[a[8]*4], b[1] ^= t[a[13]*4], b[2] ^= t[a[2]*4],
b[3] ^= t[a[7]*4];
b = (byte *)&k3;
b[0] ^= t[a[12]*4], b[1] ^= t[a[1]*4], b[2] ^= t[a[6]*4],
b[3] ^= t[a[11]*4];
```

Εικόνα 73:Κώδικας

Κύκλωμα Niyaz:

Σε αυτό το κύκλωμα, η ολίσθηση γίνεται μέσω της συνάρτησης **ShiftRows()** καθώς η ανάμιξη πινάκων γίνεται με της συνάρτησης **MixColumns()** όπως φαίνεται στην Εικόνα 76.

ShiftRows:

```
temp=state[1][0];
state[1][0]=state[1][1];
state[1][1]=state[1][2];
state[1][2]=state[1][3];
state[1][3]=temp;

temp=state[2][0];
state[2][0]=state[2][2];
state[2][2]=temp;

temp=state[2][1];
state[2][1]=state[2][3];
state[2][3]=temp;

temp=state[3][0];
state[3][0]=state[3][3];
state[3][3]=state[3][2];
state[3][2]=state[3][1];
state[3][1]=temp;
```

Εικόνα 74:Κώδικας



Βλέπουμε από τον κώδικα, ότι διαβάζει ο πίνακας state την μνήμη, με αποτέλεσμα να δημιουργεί καθυστέρηση αφού χρειάζονται δύο κύκλοι για να ολοκληρωθεί το διάβασμα και να εκτελεστεί η επόμενη εντολή. Επιπλέον, παρατηρούμε ότι υπάρχει εξάρτηση μεταξύ των εντολών με αποτέλεσμα να δημιουργεί κι άλλη καθυστέρηση.

	Pipelined	Latency	Initiation Interval
● ShiftRows	-	11	12

Εικόνα 75: Επιβάρυνση Συνάρτησης ShiftRows

MixColumns:

```
for(i=0;i<4;i++){
t=state[0][i];
Tmp = state[0][i] ^ state[1][i] ^ state[2][i] ^ state[3][i] ;
Tm = state[0][i] ^ state[1][i] ; Tm = xtime(Tm); state[0][i]
^= Tm ^ Tmp ;
Tm = state[1][i] ^ state[2][i] ; Tm = xtime(Tm); state[1][i]
^= Tm ^ Tmp ;
Tm = state[2][i] ^ state[3][i] ; Tm = xtime(Tm); state[2][i]
^= Tm ^ Tmp ;
Tm = state[3][i] ^ t ; Tm = xtime(Tm); state[3][i] ^= Tm ^
Tmp ;
}
```

Εικόνα 76:Κώδικας

Ομοίως, με την προηγούμενη συνάρτηση, ο πίνακας state διαβάζει την μνήμη και επειδή υπάρχει μεταξύ των εντολών εξάρτηση, επιβαρύνεται και άλλο το κύκλωμα.

	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
● MixColumns	-	17	18	-	-
● Loop 1	no	16	-	4	4

Εικόνα 77: Επιβάρυνση Συνάρτησης MixColumns



Κύκλωμα ChStone:

Η ολίσθηση γίνεται μέσω της συνάρτησης **ShiftRows()** που σε αυτό το κύκλωμα εκτελείται παράλληλα με την συνάρτηση **ByteSub**. Επιπλέον, από πριν, βλέπουμε ότι δεν επηρεάζεται η απόδοση του κυκλώματος, εφόσον το μόνο που κάνει είναι μια ολίσθηση όπως φαίνεται στην Εικόνα 80.

```
temp = Sbox[statemt[1] >> 4][statemt[1] & 0xf];
statemt[1] = Sbox[statemt[5] >> 4][statemt[5] & 0xf];
statemt[5] = Sbox[statemt[9] >> 4][statemt[9] & 0xf];
statemt[9] = Sbox[statemt[13] >> 4][statemt[13] & 0xf];
statemt[13] = temp;
temp = Sbox[statemt[2] >> 4][statemt[2] & 0xf];
statemt[2] = Sbox[statemt[10] >> 4][statemt[10] & 0xf];
statemt[10] = temp;
temp = Sbox[statemt[6] >> 4][statemt[6] & 0xf];
statemt[6] = Sbox[statemt[14] >> 4][statemt[14] & 0xf];
statemt[14] = temp;
temp = Sbox[statemt[3] >> 4][statemt[3] & 0xf];
statemt[3] = Sbox[statemt[15] >> 4][statemt[15] & 0xf];
statemt[15] = Sbox[statemt[11] >> 4][statemt[11] & 0xf];
statemt[11] = Sbox[statemt[7] >> 4][statemt[7] & 0xf];
statemt[7] = temp;
statemt[0] = Sbox[statemt[0] >> 4][statemt[0] & 0xf];
statemt[4] = Sbox[statemt[4] >> 4][statemt[4] & 0xf];
statemt[8] = Sbox[statemt[8] >> 4][statemt[8] & 0xf];
statemt[12] = Sbox[statemt[12] >> 4][statemt[12] & 0xf];
```

Εικόνα 78:Κώδικας

Η ανάμιξη των πινάκων γίνεται μέσω της συνάρτησης **Mix_Column_AddRoundKey()**.

Θα χρειαστεί να κάνουμε κάποιες αλλαγές στον κώδικα για να βρούμε την καθυστέρηση που επιβαρύνει το κύκλωμα, η ανάμιξη των πινάκων. Η αλλαγή που κάναμε, είναι ότι αφαιρέσαμε τον τελευταίο βρόγχο του κυκλώματος καθώς σε αυτόν υλοποιείται η μέθοδος AddRoundKey που προσθέτει τα κλειδιά στο κείμενο.



```
for (j = 0; j < nb; ++j) {ret[j * 4] = (state[j * 4] << 1);
if ((ret[j * 4] >> 8) == 1) ret[j * 4] ^= 283;x = state[1 +
j * 4];
x ^= (x << 1);
if ((x >> 8) == 1) ret[j * 4] ^= (x ^ 283);
else
ret[j * 4] ^= x;
ret[j * 4] ^= state[2 + j * 4] ^ state[3 + j * 4] ^
worda[0][j + nb * n];
ret[1 + j * 4] = (state[1 + j * 4] << 1);
if ((ret[1 + j * 4] >> 8) == 1)
ret[1 + j * 4] ^= 283;
x = state[2 + j * 4];
x ^= (x << 1);
if ((x >> 8) == 1)
ret[1 + j * 4] ^= (x ^ 283);
else
ret[1 + j * 4] ^= x;
ret[1 + j * 4] ^= state[3 + j * 4] ^ state[j * 4] ^
worda[1][j + nb * n];
ret[2 + j * 4] = (state[2 + j * 4] << 1);
if ((ret[2 + j * 4] >> 8) == 1)
ret[2 + j * 4] ^= 283;
x = state[3 + j * 4];
x ^= (x << 1);
if ((x >> 8) == 1)
ret[2 + j * 4] ^= (x ^ 283);
else
ret[2 + j * 4] ^= x;
ret[2 + j * 4] ^=state[j * 4] ^ state[1 + j * 4] ^
worda[2][j + nb * n];
ret[3 + j * 4] = (state[3 + j * 4] << 1);
if ((ret[3 + j * 4] >> 8) == 1)
ret[3 + j * 4] ^= 283;
x = state[j * 4];
x ^= (x << 1);
if ((x >> 8) == 1)
ret[3 + j * 4] ^= (x ^ 283);
else
ret[3 + j * 4] ^= x;
ret[3 + j * 4] ^=state[1 + j * 4] ^ state[2 + j * 4] ^
worda[3][j + nb * n];}
for (j = 0; j < nb; ++j) {
state[j * 4] = ret[j * 4]; state[1 + j * 4] = ret[1 + j *
4];state[2 + j * 4] = ret[2 + j * 4];state[3 + j * 4] =
ret[3 + j * 4];}
```

Εικόνα 79:Αρχικός Κώδικας



```
for (j = 0; j < nb; ++j)
{ret[j * 4] = (state[j * 4] << 1);
  if ((ret[j * 4] >> 8) == 1)ret[j * 4] ^= 283;
  x = state[1 + j * 4];
  x ^= (x << 1);
  if ((x >> 8) == 1) ret[j * 4] ^= (x ^ 283);
  else ret[j * 4] ^= x;
  ret[j * 4] ^=state[2 + j * 4] ^ state[3 + j * 4] ^
worda[0][j + nb * n];
  ret[1 + j * 4] = (state[1 + j * 4] << 1);
  if ((ret[1 + j * 4] >> 8) == 1) ret[1 + j * 4] ^=
283;
  x = state[2 + j * 4]; x ^= (x << 1);
  if ((x >> 8) == 1)
ret[1 + j * 4] ^= (x ^ 283);
  else
ret[1 + j * 4] ^= x;
  ret[1 + j * 4] ^= state[3 + j * 4] ^ state[j * 4] ^
worda[1][j + nb * n];
  ret[2 + j * 4] = (state[2 + j * 4] << 1);
  if ((ret[2 + j * 4] >> 8) == 1) ret[2 + j * 4] ^=
283;
  x = state[3 + j * 4];
  x ^= (x << 1);
  if ((x >> 8) == 1)
ret[2 + j * 4] ^= (x ^ 283);
  else
ret[2 + j * 4] ^= x;
  ret[2 + j * 4] ^=state[j * 4] ^ state[1 + j * 4] ^
worda[2][j + nb * n];
  ret[3 + j * 4] = (state[3 + j * 4] << 1);
  if ((ret[3 + j * 4] >> 8) == 1)
ret[3 + j * 4] ^= 283;
  x = state[j * 4];
  x ^= (x << 1);
  if ((x >> 8) == 1)
ret[3 + j * 4] ^= (x ^ 283);
  else
ret[3 + j * 4] ^= x;
  ret[3 + j * 4] ^=state[1 + j * 4] ^ state[2 + j * 4]
^ worda[3][j + nb * n];
}
```

Εικόνα 80:Τροποποιημένος Κώδικας



Στην Εικόνα 82, βλέπουμε ότι έχουμε έναν βρόγχο, στον οποίο υπάρχει ο πίνακας state που διαβάσει από την μνήμη. Επιπλέον, βλέπουμε ότι το ίδιο γίνεται με τον πίνακα worda. Μια τεράστια διαφορά με το κύκλωμα 2 είναι ότι ,εδώ δεν υπάρχει εξάρτηση μεταξύ των εντολών και με αυτόν τον τρόπο οι εντολές μπορούν να εκτελεστούν στο ίδιο κύκλο ρολογιού.

Η επιβάρυνση που προκύπτει από τις συναρτήσεις φαίνεται στον Πίνακα 10.

	Κύκλωμα Core	Κύκλωμα Niyaz	Κύκλωμα ChStone
Πρόσθεση Κλειδιού	4	41	17
Δημιουργία Κλειδιών	2	534	562
Αντικατάσταση με Πίνακα	2	57	15
Ολίσθηση	2	11	0
Ανάμιξη Πινάκων	2	17	1
Συνολικό	16	660	605

Πίνακας 10: Επιβάρυνση Συναρτήσεων

Σύμφωνα με τα παραπάνω, συμπεραίνουμε ότι σημαντικό ρόλο στην απόδοση των κυκλωμάτων, έχουν οι προσπελάσεις στην μνήμη, η ύπαρξη βρόγχων καθώς και οι εξαρτήσεις που υπάρχουν, ανάμεσα στις εντολές.

Μέχρι στιγμής είδαμε τις συναρτήσεις που καλούνται. Στην συνέχεια, θα αναλύσουμε τον κώδικα των κύριων συναρτήσεων που επιλέξαμε για σύνθεση.

Κύκλωμα Core:

```
int nr = 10;int i;
word k0 = key[0], k1 = key[1], k2 = key[2], k3 = key[3];
state[0] ^= k0;state[1] ^= k1;state[2] ^= k2;state[3] ^= k3;
word *t0 = (word *)table_0;word p0, p1, p2, p3;
byte *b;byte rcon = 1;
for(i=1; i<nr; i++) {
    word temp = k3;
    rot_down_8(temp);
    sub_byte(temp);
    temp ^= rcon;
    int j = (char)rcon;
    j <= 1;
    j ^= (j >> 8) & 0x1B; // if (rcon&0x80 != 0) then (j ^=
0x1B)
    rcon = (byte)j;
    k0 ^= temp;k1 ^= k0; k2 ^= k1;k3 ^= k2;
    word z0 = k0, z1 = k1, z2 = k2, z3 = k3;
    b = (byte *)state; table_lookup; rot;
    z0 ^= p0, z3 ^= p1, z2 ^= p2, z1 ^= p3;
    b += 4; table_lookup; rot;
    z1 ^= p0, z0 ^= p1, z3 ^= p2, z2 ^= p3;
    b += 4; table_lookup; rot;
    z2 ^= p0, z1 ^= p1, z0 ^= p2, z3 ^= p3;
    b += 4; table_lookup; rot;
    state[0] = z0 ^ p3;state[1] = z1 ^ p2;state[2] = z2 ^
p1;state[3] = z3 ^ p0; }
    word temp = k3;
    rot_down_8(temp);
    sub_byte(temp);
    temp ^= rcon;
    k0 ^= temp;k1 ^= k0;k2 ^= k1;k3 ^= k2;
    byte *a = (byte *)state, *t = table_0;
    b = (byte *)&k0;
    b[0] ^= t[a[0]*4], b[1] ^= t[a[5]*4], b[2] ^= t[a[10]*4],
b[3] ^= t[a[15]*4];
    b = (byte *)&k1;
    b[0] ^= t[a[4]*4], b[1] ^= t[a[9]*4], b[2] ^= t[a[14]*4],
b[3] ^= t[a[3]*4];
    b = (byte *)&k2;
    b[0] ^= t[a[8]*4], b[1] ^= t[a[13]*4], b[2] ^= t[a[2]*4],
b[3] ^= t[a[7]*4];
    b = (byte *)&k3;
    b[0] ^= t[a[12]*4], b[1] ^= t[a[1]*4], b[2] ^= t[a[6]*4],
b[3] ^= t[a[11]*4];
    state[0] = k0;state[1] = k1;state[2] = k2;state[3] = k3;
```

Εικόνα 81:Κώδικας Core



Αυτό που αξίζει να σημειώσουμε για αυτόν τον κώδικα είναι ο αριθμός των επαναλήψεων που κάνει ο βρόγχος που είναι σύμφωνα με τον αλγόριθμο 10. Όλα τα άλλα γράφτηκαν για να υλοποιήσουν τον αλγόριθμο AES.

Κύκλωμα Niyaz:

```
int Rcon[255] = {
0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b,
0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63,
0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5,
0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25,
0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8,
0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80,
0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc,
0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa,
0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2,
0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a,
0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20,
0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f,
0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d,
0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61,
0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d,
0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10,
0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a,
0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35,
0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72,
0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33,
0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01,
0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c,
0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97,
0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39,
0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94,
0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb };
int i, j, round=0;
word RoundKey[240];
KeyExpansion(RoundKey, Key, Rcon);
AddRoundKey(0, state, RoundKey);
for(round=1; round<10; round++)
{
    SubBytes(state);
    ShiftRows(state);
    MixColumns(state);
    AddRoundKey(round, state, RoundKey);
}
SubBytes(state);
ShiftRows(state);
AddRoundKey(10, state, RoundKey);
```

Εικόνα 82:Κώδικας Niyaz



Βλέπουμε ότι έχουμε δύο πίνακες `Rcon` και `RoundKey`, οι οποίοι αποθηκεύονται σε μνήμες. Στην συνέχεια, βλέπουμε ότι καλούνται οι συναρτήσεις που είδαμε παραπάνω. Πιο συγκεκριμένα, η **keyExpansion()** εκτελείται μια φορά, η **addRoundKey()** δύο φορές, καθώς **SubBytes()**, **ShiftRows()**, **AddRoundKey()** δέκα φορές και η **MixColumn()** εννιά φορές.

Κύκλωμα ChStone:

```
int main_result;
int round_val;
int nb;
int i;
KeySchedule(key);
    round_val = 0;
    nb = 4;
AddRoundKey (state, 0);
for (i = 1; i <= round_val + 9; ++i)
{
    ByteSub_ShiftRow (state);
    MixColumn_AddRoundKey (state,i);
}
ByteSub_ShiftRow (state);
AddRoundKey (state,i);
```

Εικόνα 83:Κώδικας ChStone

Σύμφωνα με τον κώδικα της Εικόνας 85, βλέπουμε ότι κάποιες μεταβλητές αποθηκεύονται σε καταχωρητές, και στην συνέχεια βλέπουμε να καλούνται οι συναρτήσεις που είδαμε παραπάνω. Πιο συγκεκριμένα, η **keySchedule()** εκτελείται μια φορά, η **addRoundKey()** δύο φορές, καθώς **byteSub_ShiftRow()** δέκα φορές και η **MixColumn_AddRoundKey()** εννιά φορές.



	Κύκλωμα Core	Κύκλωμα Niyaz	Κύκλωμα ChStone
Latency	51	1862	1059

Πίνακας 11:Συνολική Απόδοση Κυκλωμάτων

Από τους παραπάνω κώδικες, βλέπουμε ότι αυτό που επιδρά περισσότερο στην απόδοση των κυκλωμάτων είναι η αλληλεξάρτηση που έχουν μεταξύ τους οι εντολές, δηλαδή αν υπάρχει η δυνατότητα να εκτελεσθούν παράλληλα οι εντολές, όπως γίνεται στο κύκλωμα Core. Στα άλλα δύο κυκλώματα υπάρχουν εξαρτήσεις μεταξύ των εντολών, με αποτέλεσμα να περιμένει η επόμενη εντολή μέχρι να ολοκληρωθεί η προηγούμενη.

Επιπλέον, οι προσπελάσεις στην μνήμη επιδρούν αρνητικά στην απόδοση του κυκλώματος όπως φαίνεται πιο πολύ στο κύκλωμα Niyaz και ChStone. Η προσπέλαση μνήμης, χρειάζεται δύο κύκλους ρολογιού με αποτέλεσμα να καθυστερεί όλο το κύκλωμα.

Εκτός από τις μνήμες, και οι βρόγχοι επιδρούν αρνητικά στην απόδοση του κυκλώματος. Όπως βλέπουμε παραπάνω στα κυκλώματα, η πρώτη εντολή που είναι μέσα στο βρόγχο εκτελείται στο επόμενο κύκλο του ρολογιού. Η καθυστέρηση αυτή, γίνεται διπλή όταν υπάρχουν και εμφωλευμένοι βρόγχοι όπως γίνεται στο κύκλωμα Niyaz και ChStone.

Το ίδιο γίνεται και με τις συναρτήσεις που διαβάζουν θέσεις από την μνήμη όπως στο 2^ο κύκλωμα που είναι η συνάρτηση **getSboxValue()** και στο 3^ο που είναι η συνάρτηση **subByte()**.

Συνεπώς, κλειδί για υψηλή απόδοση όπως φαίνεται από τα παραπάνω είναι οι λιγότερες προσπελάσεις μνήμη, η ελαχιστοποίηση των βρόγχων καθώς η μη-εξάρτηση των εντολών αν έχουμε εντολές που χρειάζονται παραπάνω από έναν κύκλο.



Κεφάλαιο 5^ο

5.Βελτιστοποίηση κυκλώματος

Σε αυτό το κεφάλαιο, θα ασχοληθούμε με την βελτιστοποίηση του πιο αργού κυκλώματος που είναι το 2^ο. Σε αυτό το κύκλωμα έχουμε κάνει διάφορες αλλαγές, για γίνει συνθέσιμο και για να μοιάζει πιο πολύ με το πρώτο κύκλωμα που είναι σε απόδοση το καλύτερο. Ωστόσο, θα βελτιστοποιήσουμε την 1^η έκδοση δηλαδή την έκδοση που ήταν συνθέσιμη από την αρχή. Η αρχική καθυστέρηση που προκύπτει από την σύνθεση του 2^{ου} κυκλώματος είναι 3766 κύκλους ρολογιού.

Μεθοδολογία:

Στον Πίνακα 12 παρουσιάζουμε τις μεθόδους που θα χρησιμοποιήσουμε για να βελτιστοποιήσουμε το κύκλωμα.

α/α	Μέθοδος	Όρισμα
1	Απαλοιφή μεταβλητών με σταθερές	Nr,Nk
2	Μεταφορά δεδομένων	Key,In,state
3	Ενσωμάτωση συναρτήσεων	SubBytes,AddRoundKey,MixColumns
4	Unroll	KeyExpansion(For),AddRoundKey(For)
5	Pipeline	KeyExpansion(While), Cipher(For)
6	Array_Reshape	state
7	Array_Partition	sbox

Πίνακας 12:Μέθοδοι Βελτιστοποίησης Κυκλώματος

1.Αρχικά, Βλέπουμε ότι κύρια συνάρτηση που πρόκειται να κάνουμε σύνθεση ζητάει τα Nr, Nk από το testbench.

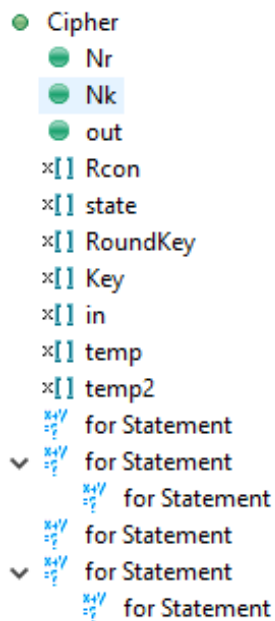
```
void Cipher(int Nr,int Nk, unsigned char out[16])
```

Εικόνα 84:Συνάρτηση Cipher

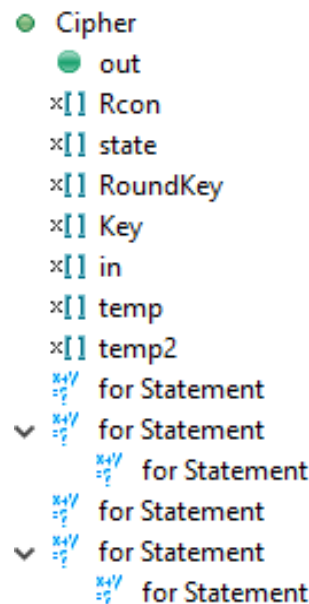
Άρα περιμένει να λάβει αυτά τα ορίσματα, το οποίο δημιουργεί καθυστέρηση στο κύκλωμα.Επιπλέον, επειδή εμείς σε αυτήν την εργασία ασχολούμαστε μόνο με την κρυπτογράφηση για 128 bits. Οι μεταβλητές Nr, Nk μπορούν να πάρουν σταθερές τιμές εφόσον είναι γνωστές, οι οποίες είναι Nr=10 και Nk=4.Στην συνέχεια, θα αντικαταστήσουμε στο κώδικα τις παραπάνω μεταβλητές, με τις τιμές αυτές.



Πριν τις αλλαγές:



Μετά τις αλλαγές:



Εικόνα 85:Στοιχεία Συνάρτησης Cipher

Μετά την αλλαγή αυτή, η καθυστέρηση που επιβαρύνεται το κύκλωμα έφθασε στους 1976 κύκλους ρολογιού.

2.Έπειτα, θα προσπαθήσουμε να μεταφέρουμε όλο και περισσότερο κώδικα από την συνάρτηση που θα γίνει σύνθεση, στο αρχείο test bench .

Θα μεταφέρουμε το κείμενο που θα κρυπτογραφηθεί, και το κλειδί στο αρχείο test bench, καθώς θα τα βάλουμε ως όρισμα στην κύρια συνάρτηση ώστε να μπορέσει να τα χρησιμοποιήσει.

```

unsigned char in[32] =
{0x00 ,0x01 ,0x02 ,0x03 ,0x04 ,0x05 ,0x06 ,0x07 ,0x
08 ,0x09 ,0x0a ,0x0b ,0x0c ,0x0d ,0x0e ,0x0f};
unsigned char Key[32]=
{0x00 ,0x11 ,0x22 ,0x33 ,0x44 ,0x55 ,0x66 ,0x77 ,0x
88 ,0x99 ,0xaa ,0xbb ,0xcc ,0xdd ,0xee ,0xff};
Cipher(Key,in,out);
    
```

Εικόνα 86:Κώδικας

Στην συνέχεια, βλέπουμε ότι το κείμενο αποθηκεύεται σε έναν πίνακα state, και έπειτα το τελειοποιημένο ,κρυπτογραφημένο κείμενο αποθηκεύεται από το state σε έναν πίνακα out. Για αυτόν τον λόγο, αντί για in και για out τα οποία, τα χρησιμοποιούμε για είσοδο και για έξοδο, θα χρησιμοποιήσουμε μόνο το πίνακα state που θα κάνει και τα δύο. Συνεπώς, βγάξουμε το in και out από τους κώδικες.



Για να γίνει αυτό όμως πρώτα, πρέπει να εκχωρήσουμε το πίνακα in στον πίνακα state.

```
word state[4][4];
for(int i=0;i<4;i++)
{
    For(int j=0;j<4;j++)
    {
        State[j][i]=in[i*4 + j];
    }
}
```

Εικόνα 87:Κώδικας

Καθώς, πρέπει να μπει ως όρισμα μέσα στην κύρια συνάρτηση:

```
void Cipher(unsigned char state[4][4],unsigned char
Key[32])
```

Εικόνα 88:Κώδικας

Μετά από αυτές τις αλλαγές, η καθυστέρηση που επιβαρύνεται το κύκλωμα έφθασε στους 1862 κύκλους ρολογιού.

3.Επιπλέον, βλέπουμε ότι η συνάρτηση **shiftRows()** καλείται πάντα μετά την **subBytes()**. Για αυτόν τον λόγο, θα προσπαθήσουμε ενσωματώσουμε την **shiftRows()** στην **SubBytes()**.

```
void SubBytes(unsigned char state[4][4])
{
    int i,j;
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
            state[i][j] = getSboxValue(state[i][j]);
        }
    }
}
```

Εικόνα 89:Αρχικός κώδικας subBytes()



```
void SubBytes(unsigned char state[4][4])
{
    int i,j;
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
            state[i][j] = getSBoxValue(state[i][j]);
        }
    }
}

unsigned char temp;
temp=state[1][0];state[1][0]=state[1][1];
state[1][1]=state[1][2];state[1][2]=state[1][3];
state[1][3]=temp;
temp=state[2][0];state[2][0]=state[2][2];
state[2][2]=temp;
temp=state[2][1];state[2][1]=state[2][3];
state[2][3]=temp;
temp=state[3][0];state[3][0]=state[3][3];
state[3][3]=state[3][2];state[3][2]=state[3][1];
state[3][1]=temp;
```

Εικόνα 92:Κώδικας SubBytes() μαζί με shiftRows()

Το ίδιο πράγμα θα κάνουμε με την συνάρτηση **MixColumns()** και την συνάρτηση **AddRoundKey()**. Η **MixColumns()** καλείται 9 φορές, καθώς η **AddRoundKey()** καλείται 10 φορές. Για αυτό θα βάλουμε συνθήκη για την συνάρτηση **MixColumns()** και ο κώδικας θα γίνει όπως φαίνεται στην Εικόνα 92.

```
if (round <10){
#define xtime(x) ((x<<1) ^ ((x>>7) & 1) * 0x1b))
unsigned char Tmp,Tm,t;
for(i=0;i<4;i++){
    t=state[0][i];
    Tmp = state[0][i] ^ state[1][i] ^ state[2][i] ^
state[3][i] ;
    Tm = state[0][i] ^ state[1][i] ; Tm = xtime(Tm);
state[0][i] ^= Tm ^ Tmp ;
    Tm = state[1][i] ^ state[2][i] ; Tm = xtime(Tm);
state[1][i] ^= Tm ^ Tmp ;
    Tm = state[2][i] ^ state[3][i] ; Tm = xtime(Tm);
state[2][i] ^= Tm ^ Tmp ;
    Tm = state[3][i] ^ t ; Tm = xtime(Tm); state[3][i] ^=
Tm ^ Tmp ;}
```

Εικόνα 90:Κώδικας



Ενώ για την AddRoundKey το μόνο πράγμα που χρειάζεται να αλλάξουμε,είναι να προσθέσουμε τα ορίσματα στην συνάρτηση **SubBytes()** τις μεταβλητές RoundKey και Round.

```
void SubBytes(unsigned char[4][4],int round,unsigned char  
RoundKey[240])
```

Εικόνα 93:Συνάρτηση SubBytes

Μετά τις αλλαγές αυτές η καθυστέρηση που επιβαρύνεται το κύκλωμα έφθασε στους 1696 κύκλους ρολογιού.

4.Εκτός από αλλαγές κώδικα, ένας άλλος τρόπος για να βελτιστοποιήσουμε το κύκλωμα μας είναι να βάλουμε directives.

Αρχικά,θα χρησιμοποιήσουμε το Directive unroll στο 1^ο βρόγχο for της συνάρτησης **KeyExpansion()**.

▼ ● KeyExpansion
 ×[1] temp
 ▼ ×[4] KeyExpansion_label17
 %0 HLS UNROLL

Εικόνα 914:Εισαγωγή Directive

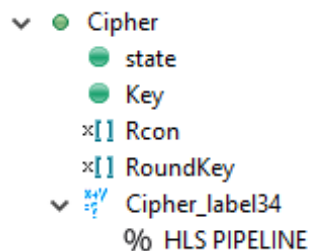
Στην συνέχεια, θα χρησιμοποιήσουμε το Directive unroll στο εμφωλευμένο βρόγχο for της συνάρτησης **AddRoundKey()**.

▼ ● AddRoundKey
 ▼ ×[4] for Statement
 ▼ ×[4] AddRoundKey_label19
 %0 HLS UNROLL

Εικόνα 925:Εισαγωγή Directive

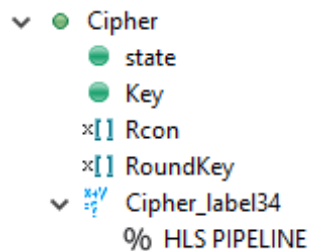
Μετά τις αλλαγές αυτές η καθυστέρηση που επιβαρύνεται το κύκλωμα έφθασε στους 1667 κύκλους ρολογιού.

5.Θα χρησιμοποιήσουμε το Directive Pipeline στο βρόγχο while της συνάρτησης **KeyExpansion()** .



Εικόνα 936:Εισαγωγή Directive

Έπειτα, θα χρησιμοποιήσουμε το Directive Pipeline στο βρόγχο for της συνάρτησης **Cipher()** .

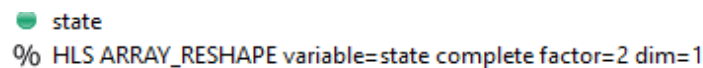


Εικόνα 947:Εισαγωγή Directive

Καθώς η καθυστέρηση που επιβαρύνεται το κύκλωμα έφθασε στους 509 κύκλους ρολογιού.

6..Επιπλέον, με την χρήση του Directive ARRAY_RESHAPE, φτιάχνουμε έναν νέο πίνακα με λιγότερα στοιχεία αλλά μεγαλύτερα σε μέγεθος.

Αυτό κάναμε στο state όπως φαίνεται στην Εικόνα 98.



Εικόνα 958:Εισαγωγή Directive

Και η καθυστέρηση που επιβαρύνεται το κύκλωμα έφθασε στους 365 κύκλους ρολογιού.



8.Χρησιμοποιώντας το Directive, ARRAY_PARTITION που χωρίζει τον πίνακα σε μικρότερους καθώς δεσμεύει καταχωρητές, στον πίνακα S-Box, δουλεύει αποτελεσματικότερα η μέθοδος της διοχέτευσης[16]. Τον χωρίζουμε σε δύο πίνακες καθώς με την χρήση αυτού του Directive διαμορφώσαμε την καθυστέρηση που επιβαρύνει το κύκλωμα στους 349 κύκλους ρολογιού.

```
×[1] sbox
% HLS ARRAY_PARTITION variable=sbox complete factor=2 dim=1
```

Εικόνα 99:Εισαγωγή Directive

Καθώς οι πόροι που δεσμεύονται φαίνονται στην Πίνακα 13.

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	31	167	-
FIFO	-	-	-	-	-
Instance	1	-	80460	29086	-
Memory	2	-	0	0	-
Multiplexer	-	-	-	378	-
Register	-	-	111	-	-
Total	3	0	80602	29631	0
Available	624	1728	460800	230400	96
Utilization (%)	~0	0	17	12	0

Πίνακας 13:Πίνακας Δέσμευσης Πόρων

Ενώ χωρίς την χρήση αυτού του Directive, οι πόροι που δεσμεύαμε φαίνονται στην Πίνακα 14.

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	31	167	-
FIFO	-	-	-	-	-
Instance	3	-	2013	2969	-
Memory	2	-	0	0	-
Multiplexer	-	-	-	384	-
Register	-	-	112	-	-
Total	5	0	2156	3520	0
Available	624	1728	460800	230400	96
Utilization (%)	~0	0	~0	1	0

Πίνακας 14:Πίνακας Δέσμευσης Πόρων

Βλέπουμε, ότι μειώθηκαν τα BRAM κατά 2 ενώ τα FF αυξήθηκαν κατά 3.638% καθώς τα LUT αυξήθηκαν κατά 741%.Καταλαβαίνουμε από τα ποσοστά, πόσο μεγάλη είναι η δέσμευση των πόρων με την χρήση αυτού του Directive.



Χρησιμοποιώντας άλλη μια φορά το Directive ARRAY_PARTITION στον πίνακα RoundKey, η καθύστερηση που επιβαρύνεται το κύκλωμα έφθασε στους 121 κύκλους ρολογιού αλλά επειδή όμως αυτό το Directive δεσμεύει παραπολύ χώρο, δεν μπορεί πλέον το υλικό να το υποστηρίξει (βλ. κόκκινα γράμματα) όπως φαίνεται στον Πίνακα 15.

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	31	271	-
FIFO	-	-	-	-	-
Instance	1	-	1117468	363998	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	222	-
Register	-	-	7207	-	-
Total	1	0	1124706	364491	0
Available	624	1728	460800	230400	96
Utilization (%)	~0	0	244	158	0

Πίνακας 15: Πίνακας Δέσμευσης Πόρων

Το ίδιο μπορεί να γίνει και στα άλλους πίνακες ώστε να αυξηθεί κι άλλο η απόδοση κυκλώματος. Ωστόσο, θα συνεχίζει να ζητάει περισσότερους πόρους.

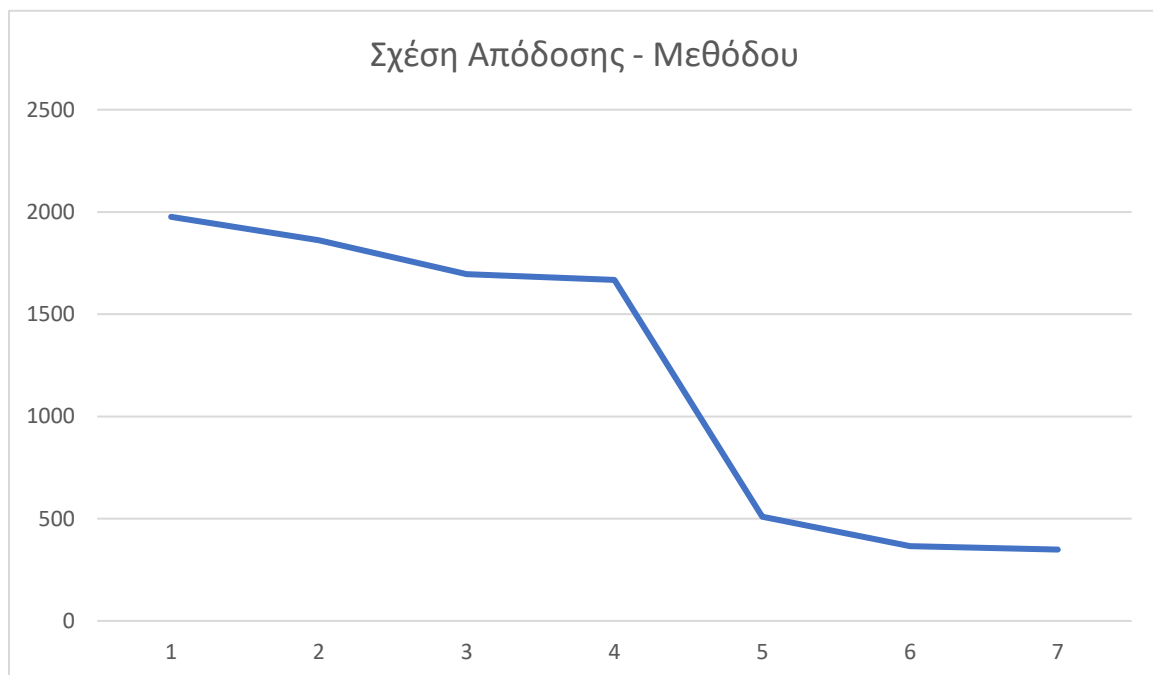
Στον Πίνακα 16 βλέπουμε τους πόρους που δεσμεύει το κύκλωμα από όλες τις τροποποιήσεις που έχουμε κάνει. Επιπλέον, βλέπουμε την τεράστια αλλαγή στην δέσμευση πόρων που κάνει το Directive ARRAY_PARTITION σε σύγκριση με τις άλλες μεθόδους που είναι οι αλλαγές κώδικα, καθώς η χρήση των Directive Pipeline, Unroll και ARRAY_RESHAPE επηρεάζουν κατά ελάχιστα τους πόρους που χρησιμοποιεί.

ΜΕΘΟΔΟΣ	BRAM	FF	LUT
0	6	2728	3055
1	6	1040	1789
2	6	1424	2021
3	6	1281	1281
4	6	1305	2126
5	5	2474	4117
6	5	2156	3520
7	3	80602	29631

Πίνακας 16: Πίνακας Πόρων

ΜΕΘΟΔΟΣ	LATENCY
0	3766
1	1976
2	1862
3	1696
4	1667
5	509
6	365
7	349

Πίνακας 17: Πίνακας Απόδοσης ανά μέθοδο



Εικόνα 100: Διάγραμμα Σχέσης Απόδοσης και Μεθόδου

Με βάση τα παραπάνω μπορέσαμε να αυξήσουμε την απόδοση του κυκλώματος κατά 90.73%. Επιπλέον, μείωσαμε τα BRAM 50% καθώς αυξήσαμε κατά 2854% και τα LUT κατά 869%. Αν μέναμε όμως στην 6^η μέθοδο τότε θα είχαμε αύξηση των FF κατά 26%, αύξηση των LUT κατά 15.2% καθώς θα είχαμε βελτίωση της απόδοσης κατά 90.3%.



Κεφάλαιο 6^ο

6.Συμπεράσματα

Σκοπός της παρούσας πτυχιακής εργασίας είναι η ανάλυση του εργαλείου HLS, με την χρήση τριών διαφορετικών υλοποιήσεων του αλγορίθμου κρυπτογράφησης AES. Στην αρχή, εφόσον πήραμε τρεις διαφορετικές υλοποιήσεις που είναι γραμμένες σε γλώσσα υψηλού επιπέδου C, κάναμε διάφορες τροποποιήσεις ώστε να γίνουν συνθέσιμοι από το εργαλείο HLS. Αφού, κάναμε διάφορες αλλαγές ώστε να έχουν ίδιες εισόδους και εξόδους και αναλύσαμε την σύνθεση τους, μπορέσαμε να καταλήξουμε στο συμπέρασμα ,ότι κλειδί για την σύνθεση με υψηλή απόδοση, ενός κώδικα γραμμένο σε γλώσσα υψηλού επιπέδου ,είναι η ελαχιστοποίηση προσπελάσεων σε μνήμη, η μείωση των βρόγχων και μεταφορά δεδομένων από την είσοδο και στην έξοδο του κυκλώματος καθώς και η συγγραφή κώδικα με τέτοιο τρόπο ώστε να μην υπάρχουν εξαρτήσεις μεταξύ των εντολών. Εφόσον γίνουν, όλα αυτά, με την χρήση των κατάλληλων Directives μπορούμε κι άλλο να επιταχύνουμε την απόδοση του κυκλώματος όπως κάναμε και πιο πάνω στην πιο αργή υλοποίηση ,που το επιταχύναμε συνολικά κατά 90.73%.Συνεπώς, ακόμη σε έναν πολύπλοκο αλγόριθμο κρυπτογράφησης, μπορέσαμε να παράξουμε κυκλώματα με ικανοποιητική απόδοση.



Κεφάλαιο 7^ο

7.Βιβλιογραφικές Πηγές

1. https://en.wikipedia.org/wiki/Advanced_Encryption_Standard
2. <https://www.xilinx.com/video/hardware/vivado-hls-in-depth-technical-overview.html>
3. <https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0012-vivado-high-level-synthesis-hub.html>
4. http://www.eit.lth.se/fileadmin/eit/courses/etin45/Lab_Files/Catapult_Work_Flow_Tutorial.pdf
5. http://users.ece.utexas.edu/~gerstl/ee382v_f14/soc/vivado_hls/VivadoHLS_Overview.pdf
6. https://en.wikipedia.org/wiki/Xilinx_Vivado
7. https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf
8. <https://www.semiwiki.com/forum/content/1865-systemc-vs-c-high-level-synthesis.html>
9. http://nefeli.lib.teicrete.gr/browse/sdo/fi/2012/AndroulakiAthina/attached-document-1343751992-644452-12293/Androulaki_Athina_2012.pdf
10. https://opencores.org/project,tiny_aes
11. <http://www.ertl.jp/chstone/>
12. <http://www.hoozi.com/posts/advanced-encryption-standard-aes-implementation-in-cc-with-comments-part-1-encryption/>
13. "Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis",
Journal of Information Processing, Vol. 17, pp.242-254, (2009)
14. "An Implementation of the AES cipher using HLS", Rodrigo Schmitt Meurer, Tiago Rogério Mück, Antônio Augusto Fröhlich (2013)
15. "Can High-Level Synthesis Compete Against a Hand-Written Code in the Cryptographic Domain? A Case Study*" Ekawat Homsirikamol and Kris Gaj, (2015)
16. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug1270-vivado-hls-opt-methodology-guide.pdf