**a**

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 τρείς 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 αυτηνής 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**

Στην ενότητα αυτή μπορείτε προαιρετικά να προσθέσετε ευχαριστίες σε όποιους επιθυμείτε και θεωρείτε ότι συνέβαλαν με οποιοδήποτε τρόπο στην ολοκλήρωση της εργασίας και των σπουδών σας.

Ιούνιος 2018,

Λιαούτσης Αβραάμ

**Περιεχόμενα**

[1.Εισαγωγή 5](#_Toc516227117)

[2.High Level Synthesis 7](#_Toc516227118)

[2.1 Κατανοώντας το HLS 8](#_Toc516227119)

[2.2 Test Bench, Υποστήριξη γλωσσών, βιβλιοθήκες της C. 10](#_Toc516227120)

[2.2.1 Τest Bench 10](#_Toc516227121)

[2.2.2 Υποστήριξη γλωσσών 10](#_Toc516227122)

[2.2.3 Βιβλιοθήκες της C 11](#_Toc516227123)

[2.4 Σύνθεση 11](#_Toc516227124)

[2.5 Βελτιστοποίηση 12](#_Toc516227125)

[2.6 Ανάλυση 12](#_Toc516227126)

[2.7 Επαλήθευση RTL 13](#_Toc516227127)

[2.8 Προτάσεις 13](#_Toc516227128)

[2.8.1 Κώδικας 13](#_Toc516227129)

[2.8.2 Γλώσσα Προγραμματισμού 14](#_Toc516227130)

[2.8.3 Directives 14](#_Toc516227131)

[3. Μελέτη του αλγορίθμου AES 17](#_Toc516227132)

[3.1 Ανάλυση συναρτήσεων 20](#_Toc516227133)

[4. Ανάλυση υλοποιήσεων 26](#_Toc516227134)

[4.1 Τροποποιήσεις υλοποιήσεων για σύνθεση 26](#_Toc516227135)

[4.1.1 Κώδικας Core 26](#_Toc516227136)

[4.1.2 Κώδικας Niyaz 28](#_Toc516227137)

[4.1.3 Κώδικας ChStone 30](#_Toc516227138)

[4.2 Ομογενοποίηση υλοποιήσεων 32](#_Toc516227139)

[Κύκλωμα Niyaz 34](#_Toc516227140)

[Κύκλωμα ChStone 36](#_Toc516227141)

[4.3 Σύγκριση των υλοποιήσεων 38](#_Toc516227142)

[5.Βελτιστοποίηση κυκλώματος 59](#_Toc516227143)

[6.Συμπεράσματα 67](#_Toc516227144)

[7.Βιβλιογραφικές Πηγές 68](#_Toc516227145)

***Κεφάλαιο 1ο***

# 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 επίπεδου. 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 τρείς 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 τρείς 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 2ο***

# 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 Χilinx 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 ISΕ (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. To 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.
* To 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 κλάσσεις, πρότυπα, συναρτήσεις καθώς και διάφορους τελεστές της γλώσσας C++.

## 2.1 Κατανοώντας το HLS

To 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:

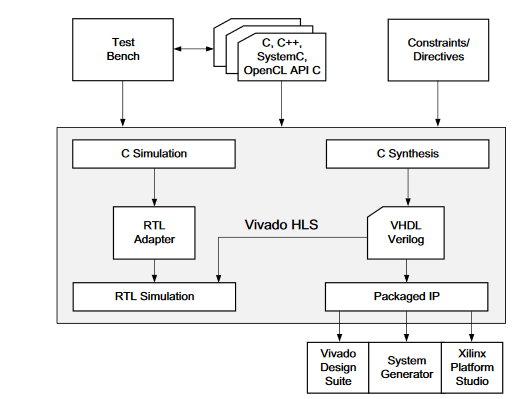
1. Μεταγλώττιση, εκτέλεση (προσομοίωση) και αποσφαλμάτωση του αλγορίθμου C.
2. Σύνθεση του αλγορίθμου C σε υλοποίηση RTL.Μπορούν να χρησιμοποιηθούν διάφορα directives για βελτιστοποίηση.
3. Δημιουργία αναλυτικών αναφορών και ανάλυση της απόδοσης κυκλώματος μετά την σύνθεση.
4. Επαλήθευση της RTL υλοποίησης 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 περίοδος 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 είσοδών 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 συναρτήσεις 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 Τest 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

To 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.

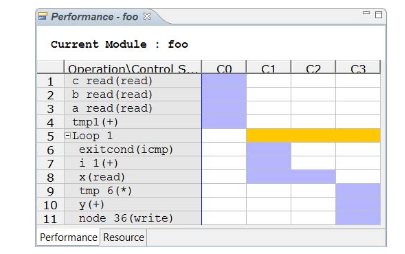
## 2.5 Βελτιστοποίηση

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.
* Με την χρήση του Directive Latency,μπορούμε να καθορίσουμε το χρόνο που χρειάζεται για την ολοκλήρωση των συναρτήσεων και βρόγχων
* Μπορούμε να καθορίσουμε το όριο των πόρων που χρησιμοποιούνται στο κύκλωμα με την χρήση του Directive Allocation.
* Με την χρήση του Directive Inline,μπορούμε να καταργήσουμε τις εξαρτήσεις των εντολών του κώδικα και να εκτελέσουμε κάποιες συγκεκριμένες λειτουργίες.
* Με την χρήση του Directive Protocol,μπορούμε να επιλέξουμε το πρωτόκολλο εισόδου/εξόδου ώστε να σιγουρευτούμε ότι το τελικό σχέδιο μπορεί να συνδεθεί με τα άλλα blocks του υλικού με το ίδιο πρωτόκολλο.

## 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.



**Εικόνα 2:** Παράδειγμα ανάλυσης απόδοσης

Στην Εικόνα 2 απεικονίζεται ένα παράδειγμα ανάλυσης απόδοσης της συνάρτησης foo.Αρχικά, βλέπουμε στην αρχή στον 1ο κύκλο (C0) να διαβάζει από τις θύρες τις μεταβλητές a,b,c καθώς και την λειτουργία της πρόσθεσης. Δεύτερον ,βλέπουμε ότι το κύκλωμα μπαίνει σε ένα βρόγχο(Loop 1) καθώς παράλληλα ελέγχεται η μεταβλητή που μετράει τον αριθμό των επαναλήψεων του βρόγχου(I 1(+)) και επιπλέον ελέγχεται και η συνθήκη εξόδου για το τερματισμού του βρόγχου(exitcond(icmp)). Τρίτον, βλέπουμε ότι διαβάζονται δεδομένα στην μεταβλητή X(read).Αυτή η ενέργεια απαιτεί 2 κύκλους ρολογιού, επειδή έχει πρόσβαση σε μνήμη RAM(παράγει διεύθυνση σε έναν κύκλο και στην συνέχεια διαβάζει τα δεδομένα στον επόμενο).Τέλος, βλέπουμε στον 4ο κύκλο(C3), γίνονται οι υπολογισμοί(tmp 6(\*), y(+), node 36(write))καθώς η έξοδος γράφεται στην θύρα Υ και ο βρόγχος ξεκινάει από την αρχή.

## 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. To 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 βγάλει 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 τα 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.

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. Οι πίνακες μπορούν να κατανεμηθούν σε μικρότερους πίνακες ή ακόμη και σε καταχωρητές .Ωστόσο, η διαμέριση των μεγάλων πινάκων καθιστά αναγκαία την χρήση ενός πολύ μεγάλου αριθμού καταχωρητών.

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 μας.

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 στα δεδομένα που πρόκειται να διαβάσει. Οι πύλες εισόδου και εξόδου μπορούν να καθυστερήσουν την συνολική απόδοση του κυκλώματος. Η δημιουργία περισσότερων πυλών που μπορούν να χρησιμοποιηθούν είτε για είσοδο είτε για εξόδου, επιβαρύνουν περισσότερο το κύκλωμα[7].

### 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 όποια 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 συγκριμένο 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 η 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 η 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.

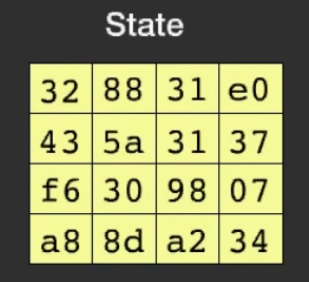
***Κεφάλαιο 3ο***

# 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 αλγόριθμό κρυπτογράφησης αφού είναι ο πρώτος και ο μοναδικός που έχει εγκριθεί από την National Security Agency (NSA) για την ασφάλεια των δεδομένων της[1].

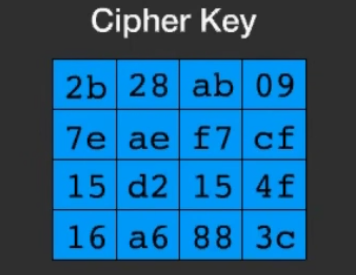
Ο αλγόριθμος αυτός χρειάζεται, όπως όλους τους άλλους αλγορίθμους κρυπτογράφησης ένα κείμενο (Plain text) καθώς και ένα κλειδί(Cypher Key) καθώς με την χρήση του κλειδιού αυτού θα γίνει η κρυπτογράφηση του κειμένου.

Ο αλγόριθμος χρησιμοποιεί κρυπτογράφηση ανά block, καθώς κρυπτογραφεί 16 bits την φορά όπως βλέπουμε στην εικόνα 3.1.

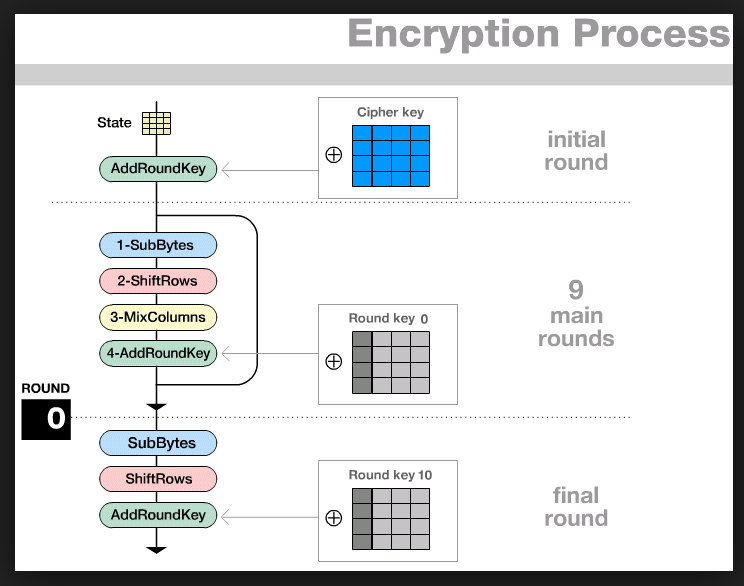


**Εικόνα 3.1:** Αρχικό κείμενο.

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



**Εικόνα 3.2:** Κλειδί κρυπτογράφησης.

:

**Εικόνα 3.3:** Διαδικασία Κρυπτογράφησης.

Όπως βλέπουμε και στην εικόνα 3.3, 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 παραπάνω εικόνα αλλάζει.

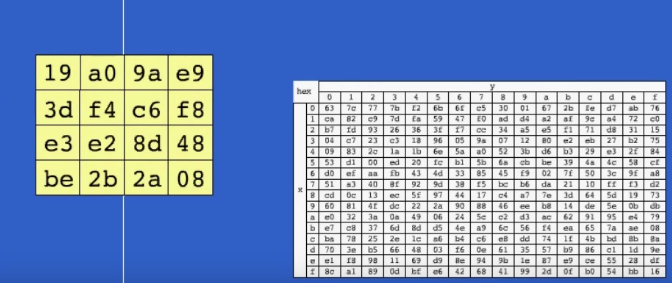
Δηλαδή αν έχουμε 128 bits τότε θα γίνουν 10 επαναλήψεις , αν έχουμε 192 θα γίνουν 12 και αν έχουμε 256 θα γίνουν 14. Δηλαδή ο αριθμός εκτέλεσης των παραπάνω συναρτήσεων φαίνεται στον πίνακα 1:

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

**Πίνακας 1:**Αριθμός Εκτέλεσης των Συναρτήσεων.

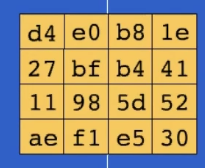
## 3.1 Ανάλυση συναρτήσεων

1. SubBytes:



**Εικόνα 3.4:**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.Στην Εικόνα 3.5 φαίνεται το τροποποιημένο κείμενο:



**Εικόνα 3.5:**Τροποποιημένο κείμενο.

1. ShiftRows:

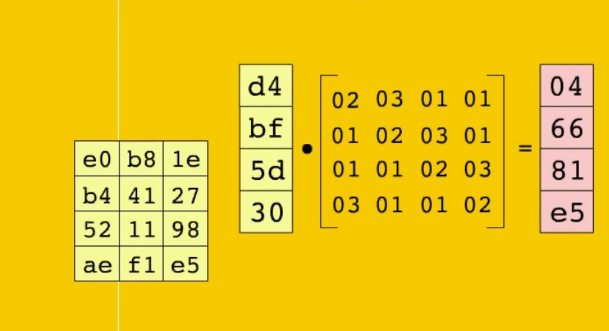
Στην συγκεκριμένη διαδικασία παίρνουμε την δεύτερη σειρά του block και την κάνουμε rotate κατά ένα byte. Στην τρίτη σειρά θα την κάνουμε rotate ανά 2 byte ενώ στο τελευταία σειρά θα την κάνουμε rotate ανά 3 byte όπως φαίνεται στην Εικόνα 3.6:



**Εικόνα 3.6:**Εκτέλεση συνάρτησης ShiftRows.

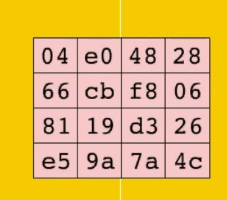
1. MixColumns:

Στην Εικόνα 3.7 απεικονίζεται η διαδικασία που παίρνουμε την πρώτη στήλη του block και την πολλαπλασιάζουμε με έναν συγκεκριμένο πίνακα:



**Εικόνα 3.7:**Εκτέλεση συνάρτησης MixColumns.

Συνεχίζουμε και με τις άλλες στήλες, καθώς βλέπουμε στην Εικόνα 3.8 το τροποποιημένο κείμενο.

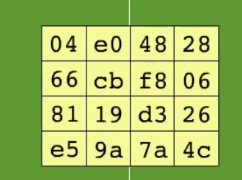


**Εικόνα 3.8:**Τροποποιημένο κείμενο.

1. AddRoundkey:

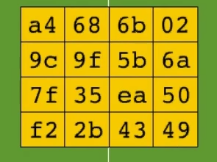
Σε αυτήν την συνάρτηση παίρνουμε το κλειδί (θα δείξουμε μετά πως υπολογίζεται) και το προσθέτουμε στο state που έχουμε.

Πχ . Έστω το state που βλέπουμε στην Εικόνα 3.9:



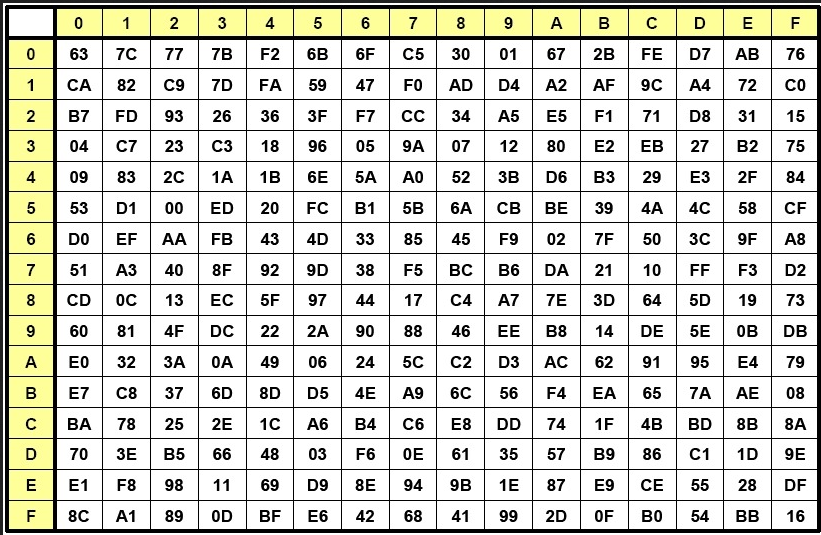
**Εικόνα 3.9:** Τροποποιημένο κείμενο.

Στην Εικόνα 3.2 έχουμε το κλειδί καθώς με πρόσθεση του στο κείμενο θα παραχθεί αυτό που φαίνεται στην Εικόνα 3.10.

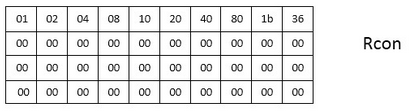


**Εικόνα 3.10:**Τροποποιημένο κείμενο.

1. 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 3.2, 3.11 και 3.12.

**Εικόνα 3.11:** Πίνακας S-Box.



**Εικόνα 3.12:** Πίνακας Rcon.

Αρχικά, παίρνουμε την τελευταία στήλη του κλειδιού κρυπτογράφησης, κάνουμε ένα rotate προς τα πάνω, όπως φαίνεται στην Εικόνα 3.13:



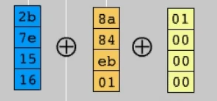
**Εικόνα 3.13:** Τελευταία στήλη του Πίνακα.

Δεύτερον, αντικαθιστούμε τα bits που έχουμε , με τα bits του πίνακα S-Box ώστε να γίνει η στήλη, όπως φαίνεται στην Εικόνα 3.14:



**Εικόνα 3.14:** Τροποποιημένη στήλη.

Τρίτον, προσθέτουμε την 1η στήλη του κλειδιού κρυπτογράφησης με την στήλη που παράξαμε στο 2ο βήμα ,καθώς και με την 1η στήλη του πίνακα Rcon, όπως φαίνεται στην Εικόνα 3.15:



**Εικόνα 3.15:** Πράξη στηλών.

Στην Εικόνα 3.16 φαίνεται το αποτέλεσμα της πρόσθεσης που είναι η πρώτη στήλη του νέου κλειδιού.



**Εικόνα 3.16:** Πρώτη στήλη του νέου κλειδιού.

Για να υπολογίσουμε τις άλλες στήλες θα πρέπει να προσθέσουμε την παραπάνω στήλη που παράξαμε με τις αντίστοιχη 2η, 3η και 4η στήλη του αρχικού κλειδιού κρυπτογράφησης καθώς το αποτέλεσμα που θα μας βγάλει φαίνεται στην εικόνα 3.17:



**Εικόνα 3.17:** Νέο κλειδί.

Για να φτιάξουμε το 2ο 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 1η στήλη του Rcon αλλά την 2η .

***Κεφάλαιο 4ο***

# 4. Ανάλυση υλοποιήσεων

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

Η γλώσσα προγραμματισμού που χρησιμοποιούμε για τις τρείς υλοποιήσεις είναι η γλώσσα C. Η γλώσσα C είναι πιο φιλική προς στον 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 4.1.

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 4.1: Σφάλματα κατά την σύνθεση.

Εικόνα 4.2: Αρχικός Κώδικας

void encrypt\_128\_key\_expand\_inline(word state[], word key[])

Για να το λύσουμε αυτά, κάναμε τις αλλαγές που φαίνονται στην Εικόνα 4.3.

void encrypt\_128\_key\_expand\_inline(word state[4], word key[4])

Εικόνα 4.3: Τροποποιημένος Κώδικας

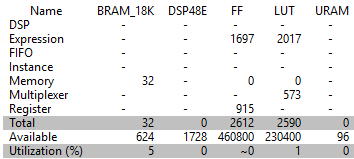
Δηλαδή καθορίσαμε το μέγεθος της μνήμης των μεταβλητών state και key, για το μέγεθος που ορίζεται παρακάτω ,δηλαδή 4.Μετά αυτές τις αλλαγές μπορέσαμε να κάνουμε σύνθεση το κύκλωμα σε RTL μορφή.

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 απόδοση του κυκλώματος Core είναι 51 καθώς και πόροι που δεσμεύει φαίνονται στον πίνακα 2.1.

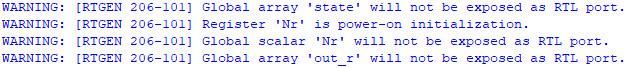


Πίνακας 2.1: Πίνακας Δέσμευσης Πόρων.

### 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 μπουν 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 4.4



Εικόνα 4.4: 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 4.5.

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

void MixColumns(unsigned char state[4][4])

void ShiftRows(unsigned char state[4][4])

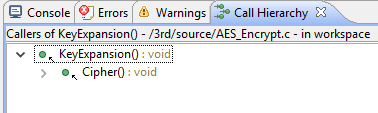
void AddRoundKey(int round,unsigned char state[4][4],unsigned char RoundKey[240])

void SubBytes(unsigned char[4][4])

void KeyExpansion(unsigned char RoundKey[240],unsigned char Key[32],int Rcon[255],int Nk,int Nr)

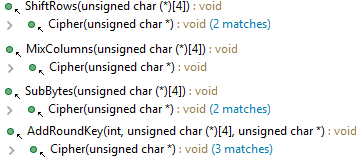
Εικόνα 4.5:Ορίσματα Συναρτήσεων.

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 συνάρτηση **Cipher()** όπως φαίνεται στην Εικόνα 4.7.



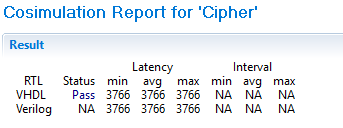
Εικόνα 4.7: Ιεραρχία Συναρτήσεων.

Δηλαδή βλέπουμε στην Εικόνα 4.7 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 4.7.Μετά από αυτές τις αλλαγές μπορέσαμε να κάνουμε επιτυχή σύνθεση.



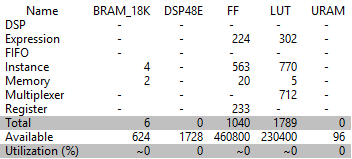
Εικόνα 4.8: Πίνακας Ιεραρχίας Συναρτήσεων.

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



Πίνακας 2.2: Πίνακας απόδοσης της συνάρτηση Cipher.

Ο λόγος που χρησιμοποιούμε την συν-προσομοίωση είναι, για να βρούμε την τελική απόδοση. 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 συνολική απόδοση. Αντιθέτως, επειδή ο αριθμός των επαναλήψεων, πιο συγκεκριμένα το Nr και το Nk τα λαμβάνει από το αρχείο test bench, στην σύνθεση δεν μπορεί να υπολογιστεί η απόδοση. Επειδή, όμως εμείς θα ασχοληθούμε με τον αλγόριθμο AES για 128 bit, ο αριθμός Nk, Nr είναι στατικός. Αντικαθιστώντας τα Nk, Nr με τις τιμές που λαμβάνουν δηλαδή 4 και 10 αντιστοίχως, η συνολική απόδοση έφθασε στα 1976 δηλαδή κατέβηκε κατά 47.53%.



Πίνακας 2.3: Πίνακας Δέσμευσης Πόρων.

### 4.1.3 Κώδικας ChStone

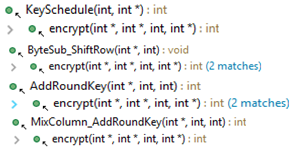
Όσον αφορά τον κώδικα ChStone:

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 υπόλοιπά 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 ,θα πρέπει να ανήκουν στην ιεραρχία της συνάρτησης που πρόκειται να γίνει σύνθεση, όπως φαίνεται στην Εικόνα Χ.



Εικόνα 4.9.4: Πίνακας Ιεραρχίας Συναρτήσεων.

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

int type

int nb;

int round\_val;

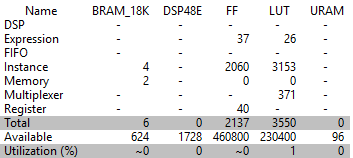
int key[32];

int statemt[32];

int word[4][120];

Εικόνα 4.9: Global Μεταβλητές

Για αυτόν τον λόγο, τις εισάγουμε 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. Το κύκλωμα που δημιουργήθηκε έχει απόδοση 1092 καθώς βλέπουμε τους πόρους που δεσμεύει στον Πίνακα 2.4.



Πίνακας 2.4: Πίνακας Δέσμευσης Πόρων

Στην συνέχεια, για να μπορέσουμε να κάνουμε την σύγκριση τον 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()**.

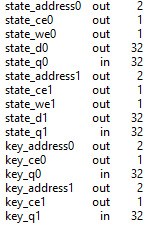
Η συνάρτηση αυτή βλέπουμε ότι έχει ως ορίσματα τις μεταβλητές state και key, οι οποίες είναι τύπου word.

word state[4];

word key[8];

encrypt\_128\_key\_expand\_inline\_no\_branch(state, key);

Καθώς οι θύρες που χρησιμοποιεί το κύκλωμα.



**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];

Καθώς οι θύρες που χρησιμοποιεί το κύκλωμα.



**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 stamt,key,out τα οποία είναι όλα τύπου int.

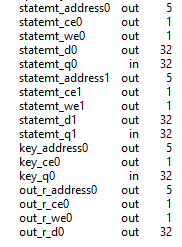
int out[32];

int key[32];

int statemt[32];

aes\_main(statemt,key,out);

Καθώς οι θύρες που χρησιμοποιεί το κύκλωμα.



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 αυτα κυκλώματα στην κύρια συνάρτηση τους, ορίσματα τύπου word οι οποίες θα είναι οι μεταβλητές state και key.

### 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 ,0x08 ,0x09 ,0x0a ,0x0b ,0x0c ,0x0d ,0x0e ,0x0f};

unsigned char temp2[32]= {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];

}

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

word out[32];

word Key[32];

word in[32];

unsigned char in[16] = {0x00 ,0x01 ,0x02 ,0x03 ,0x04 ,0x05 ,0x06 ,0x07 ,0x08 ,0x09 ,0x0a ,0x0b ,0x0c ,0x0d ,0x0e ,0x0f};

unsigned char key[16]= {0x00 ,0x11 ,0x22 ,0x33 ,0x44 ,0x55 ,0x66 ,0x77 ,0x88 ,0x99 ,0xaa ,0xbb ,0xcc ,0xdd ,0xee ,0xff};

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 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];

}

}

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

}

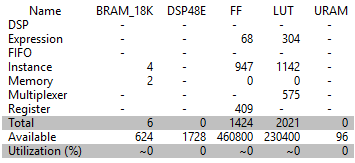
}

Printf(“\n\n”);

}

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

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



Πίνακας 2.5: Πίνακας Δέσμευσης Πόρων

### 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];

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];

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");

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])

{

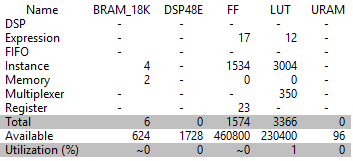
encrypt (statemt, key, 128128);

}

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.

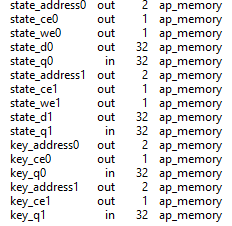
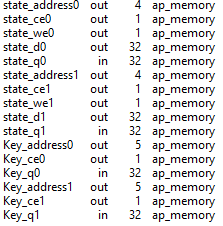
Μετά τις αλλαγές αυτές, η απόδοση του κυκλώματος παρέμεινε ίδια αλλά έχουν αλλάξει οι πόροι που χρησιμοποιεί, καθώς πλέον χρησιμοποιεί λιγότερους. Πιο συγκριμένα, από 2317 Flip-Flop που είχαμε, μειώθηκαν στα 1575 καθώς μειώθηκαν τα Look Up Tables από 3550 στα 3366.



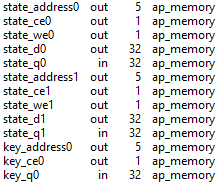
Πίνακας 2.6: Πίνακας Δέσμευσης Πόρων

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

Kύκλωμα Core: Kύκλωμα Niyaz:

Κύκλωμα ChStone:



Η βασική διαφορά του 3ου 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.

**Kύκλωμα Core:**

Σε αυτό το κύκλωμα δεν έχει υλοποιηθεί καμία συνάρτηση που φτιάχνει και προσθέτει κλειδιά. Η Πρόσθεση καθώς η δημιουργία κλειδιών γίνεται στις παρακάτω εντολές.

**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;

Οι μεταβλητές κλειδί και state, διαβάζονται από την είσοδο, καθώς θα ξεκινήσουν στο 2ο κύκλο ρολογιού .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-Βox και Rcon .Η διαδικασία είναι, ότι παίρνουμε την τελευταία στήλη του κλειδιού κρυπτογράφησης, την κάνουμε rotate, αντικαταθιστούμε 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 είναι οι παρακάτω:

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;

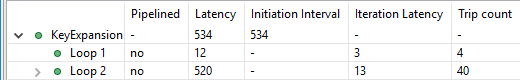
k3 ^= k2;

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:**



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

**Κώδικας**:

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];

}

Στο παρόν κώδικα έχουμε 2 μνήμες που είναι το RoundKey που κάνει write και το Key που κάνει read.

Η καθυστέρηση, δημιουργείται επειδή διαβάζει από την μνήμη το Key. To read χρειάζεται δύο κύκλους ρολογιού για να γίνει, ενώ οι άλλες πράξεις όπως πολλαπλασιασμό, πρόσθεση, εκχώρηση καθώς και το write γίνονται σε ένα κύκλο.

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

**Κώδικας:**

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++;}

Στο παρόν κώδικα έχουμε 4 μνήμες που είναι το Key που θα κάνει write, το RoundKey που θα κάνει read και write, καθώς και τα S-Box και Rcon που θα κάνουν και αυτές read. Από το παραπάνω βρόγχο που αναλύσαμε, εύκολα καταλαβαίνουμε, ότι θα υπάρχει μεγαλύτερη επιβάρυνση στην απόδοση του κυκλώματος καθώς έχουμε περισσοτέρους πίνακες που διαβάζονται από την μνήμη.

Στο 1ο 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 RoundKey και συνεπώς ο αριθμός κύκλων που θα χρειαστεί όλο ο βρόγχος είναι δύο.

Στο επόμενο κύκλο ρολογιού ξεκινάει ο 2ος βρόγχος οι οποίος εμπεριέχει έναν μεγάλο αριθμό από μνήμες, οι οποίες θα 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]);

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. Συνεπώς, θα ξεκινήσει μετά από δύο κύκλους.

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

Εκτός από αυτά, έχουμε και πίνακες που διαβάζουν την μνήμη όπως φαίνεται παρακάτω.

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];

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

Η συνάρτηση αυτή ξεκινάει στον 2ο κύκλο ρολογιού, εφόσον περιμένει τα ορίσματα από την κύρια συνάρτηση.

Συνολικά, επιβαρύνει την απόδοση του κυκλώματος κατά 534 κύκλους.

**AddRoundKey:**



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

**Κώδικας**:

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];

}

}

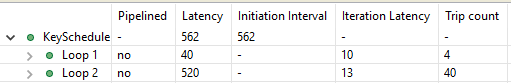
Στο παρόν κώδικα έχουμε 2 μνήμες που είναι το RoundKey που θα κάνει read και το state που θα κάνει write.

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 1ο βρόγχος, καθώς, στον επόμενο δηλαδή στον 3ο 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 παραπάνω iteration Latency.

**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:**



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

**Κώδικας:**

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];

}

Όπως και κάθε άλλη συνάρτηση, θα ξεκινήσει και αυτή στο 2ο 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 2ο 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 διαβάζεται από την μνήμη με αποτέλεσμα και αυτό να καταναλώνει 2 κύκλους.

**AddRoundKey:**



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

**Κώδικας:**

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];

}

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 παρακάτω πίνακα φαίνεται η απόδοση των παρακάτω συναρτήσεων

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

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-Βox αποτελεί η table\_lookup όπως φαίνεται παρακάτω:

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

Η δομή της είναι η εξής:

#define table\_lookup { \

p0 = t0[b[0]]; \

p1 = t0[b[1]]; \

p2 = t0[b[2]]; \

p3 = t0[b[3]]; \

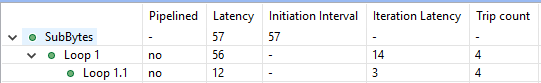
}

Αυτή η συνάρτηση από τι βλέπουμε, διαβάζει από την μνήμη τον πίνακα b[] καθώς περιμένει η t0 , με αποτέλεσμα, όπως και στα παραπάνω κυκλώματα να υπάρχει μια καθυστέρηση 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:**



Ο 1ος βρόγχος επαναλαμβάνεται 4 φορές και 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.

**Κώδικας:**

int i,j;

for(i=0;i<4;i++)

{

for(j=0;j<4;j++)

{

state[i][j] = getSBoxValue(state[i][j]);

}

}

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

**Κύκλωμα ChStone:**

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

Για να γίνει όμως αυτό θα πρέπει να κάνουμε κάποιες τροποποιήσεις του κώδικα καθώς θα αφαιρέσουμε την **ShiftRows()** δηλαδή την ολίσθηση.

**Αρχικός κώδικας:**

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];

**Τροποποιημένος 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];

Παρατηρούμε ότι ο παραπάνω κώδικας περιλαμβάνει το πίνακα state,ο οποίος διαβάζει από την μνήμη και γράφει σε αυτήν, σε σύγκριση με τον πίνακα S-Βox 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.

Απόδοση τροποποιημένης συνάρτησης:



Στον παρακάτω πίνακα φαίνεται η απόδοση των παρακάτω συναρτήσεων

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

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 παρακάτω

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;

Ενώ η ανάμιξη πινάκων γίνεται μέσω της πράξης xor, που εκτελείται παράλληλα και αυτή μετ τις άλλες εντολές όπως φαίνεται παρακάτω στην Εικόνα Χ.

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];

**Κύκλωμα Niyaz:**

Σε αυτό το κύκλωμα, η ολίσθηση γίνεται μέσω της συνάρτηση **ShiftRows()** καθώς η ανάμιξη πινάκων γίνεται με της συνάρτησης **MixColumns()** όπως φαίνεται παρακάτω:

**ShiftRows**:

**Κώδικας**:

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;

Βλέπουμε από τον κώδικα, ότι διαβάζεται ο πίνακας state από τον μνήμη, με αποτέλεσμα να δημιουργεί καθυστέρηση αφού χρειάζονται δύο κύκλοι για να ολοκληρωθεί το read καθώς να εκτελεστεί η επόμενη εντολή. Επιπλέον, παρατηρούμε ότι υπάρχει εξάρτηση μεταξύ των εντολών με αποτέλεσμα να επιβαρύνει και άλλο το κύκλωμα.

Η απόδοση της συνάρτησης φαίνεται παρακάτω:



**MixColumns**:

**Κώδικας**:

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 ;

}

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

Latency:



**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 Χ.

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];

H ανάμιξη των πινάκων γίνεται μέσω της συνάρτησης **Mix\_Column\_AddRoundKey()**.

Θα χρειαστεί να κάνουμε κάποιες αλλαγές στον κώδικα για να βρούμε το latency που επιβαρύνει μόνο 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.

**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];}

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];}

**Τροποποιημένος 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];

}

Στην Εικόνα Χ, βλέπουμε ότι έχουμε έναν βρόγχο, στον οποίο ο πίνακας state διαβάζει 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 απόδοση των παραπάνω συναρτήσεις που είδαμε φαίνονται στο πίνακα Χ.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 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 |

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:**

**Κώδικας:**

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;

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:**

**Κώδικας:**

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.

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);

**Κύκλωμα ChStone:**

**Κώδικας:**

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);

Σύμφωνα με τον παραπάνω κώδικα, 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.

**Συνολικός Πίνακας Latency των παραπάνω κυκλωμάτων**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Κύκλωμα Core | Κύκλωμα Niyaz | Κύκλωμα ChStone |
| Latency | 51 | 1862 | 1059 |

Από τους παραπάνω κώδικες, βλέπουμε ότι αυτό που επιδρά περισσότερο στην απόδοση των παρακάτω 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 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 2ο κύκλωμα που είναι η συνάρτηση **getSboxValue()** και στο 3ο που είναι η συνάρτηση **subByte()**.

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

***Κεφάλαιο 5ο***

# 5.Βελτιστοποίηση κυκλώματος

Σε αυτό το κεφάλαιο, θα ασχοληθούμε με την βελτιστοποίηση του πιο αργού κυκλώματος που είναι το 2ο. 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 1η έκδοση δηλαδή την έκδοση που ήταν συνθέσιμη από την αρχή. Η αρχική απόδοση του κυκλώματος είναι 3766 κύκλους ρολογιού.

**Μεθοδολογία**:

Στον παρακάτω πίνακα παρουσιάζουμε τις μεθόδους που θα χρησιμοποιήσουμε για να βελτιστοποιήσουμε το κύκλωμα.

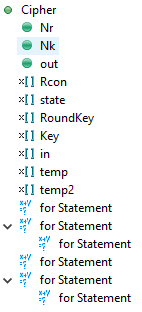
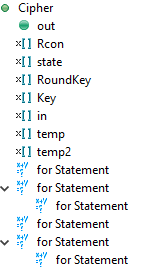
|  |  |  |
| --- | --- | --- |
| **α/α** | **Μέθοδος** | **Όρισμα** |
| **1** | Απαλοιφή μεταβλητών με σταθερές | Nr,Nk |
| **2** | Μεταφορά δεδομένων | Key,In,state |
| **3** | Ενσωμάτωση συναρτήσεων | SubBytes,AddRoundKey,MixColumns |
| **4** | Unroll | KeyExpansion(For),AddRoundKey(For) |
| **5**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

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])

Άρα περιμένει να λάβει αυτά τα ορίσματα, το οποίο δημιουργεί καθυστέρηση στο κύκλωμα.Επιπλεόν, επειδή εμείς σε αυτην την εργασία ασχολούμαστε μόνο με την κρυπτογράφιση για 128 bits. Οι μεταβλητές Nr, Νk 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:

Μετά την αλλαγή αυτή, η απόδοση του κυκλώματος έφθασε στα 1976 κύκλους.

2.Έπειτα, θα 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 ,0x08 ,0x09 ,0x0a ,0x0b ,0x0c ,0x0d ,0x0e ,0x0f};

unsigned char Key[32]= {0x00 ,0x11 ,0x22 ,0x33 ,0x44 ,0x55 ,0x66 ,0x77 ,0x88 ,0x99 ,0xaa ,0xbb ,0xcc ,0xdd ,0xee ,0xff};

Cipher(Key,in,out);

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, βγάζουμε 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];

}

}

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

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

Μετά από αυτές τις αλλαγές, το απόδοση του κυκλώματος έφθασε 1862 κύκλους.

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

**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]);

}

}

}

**Kώδικας SubBytes() μαζι με shiftRows():**

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;

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 Χ.

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 ;}

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

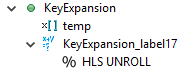
Η συνάρτηση **SubBytes().**

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

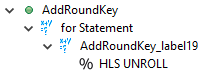
Μετά τις αλλαγές αυτές η απόδοση του κυκλώματος έχει διαμορφωθεί στους 1696 κύκλους.

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 1ο βρόγχο for της συνάρτησης **KeyExpansion()**

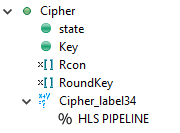


Στην συνέχεια, θα χρησιμοποιήσουμε το Directive unroll στο εμφωλευμένο βρόγχο for της συνάρτησης **AddRoundKey()**.

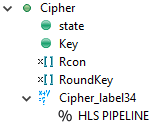


Μετά τις αλλαγές αυτές η απόδοση του κυκλώματος έχει διαμορφωθεί στους 1667 κύκλους.

5.θα χρησιμοποιήσουμε το Directive pipeline στο βρόγχο while της συνάρτησης **KeyExpansion()** .



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



Καθώς η απόδοση του κύκλώματος έχει φθάσει στους 509 κύκλους.

6..Eπιπλέον, με την χρήση του Directive ARRAY\_RESHAPE, φτιάχνουμε έναν νέο πίνακα με λιγότερα στοιχεία αλλά μεγαλύτερα σε μέγεθος.

Αυτό κάναμε στο state:

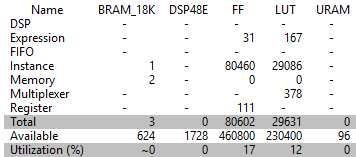


Και η απόδοση του κύκλώματος έφθασε στα 365 κύκλους.

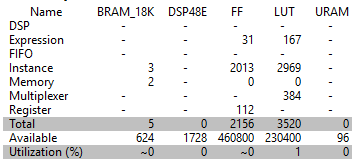
8.Χρησιμοποιώντας το Directive, ARRAY\_PARTITION που χωρίζει τον πίνακα σε μικρότερους καθώς δεσμεύει καταχωρητές, στον πίνακα S-Βox, δουλεύει αποτελεσματικότερα η μέθοδος της διοχέτευσης[16]. Τον χωρίζουμε σε δύο πίνακες καθώς με την χρήση αυτου του Directive διαμορφώσαμε την απόδοση στα 349 κύκλους.



Καθώς οι πόροι που δεσμεύονται είναι οι παρακάτω φαίνονται στην Πίνακα Χ.

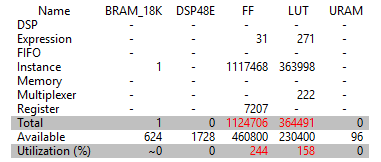


Ενώ χωρίς την χρήση αυτού του Directive, οι πόροι που δεσμεύαμε φαίνονται στην Πίνακα Χ.



Βλέπουμε, ότι μειώθηκαν τα ΒRAM 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, το η απόδοση του κυκλώματος έφθασε στα 121 αλλά επειδή όμως αυτό το Directive δεσμεύει παραπολύ χώρο, δεν μπορεί πλέον το υλικό να το υποστηρίξει (βλ. κόκκινα γράμματα).



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

Στον Πίνακα Χ βλέπουμε τους πόρους που δεσμεύει το κύκλωμα από όλες τις τροποποιήσεις που έχουμε κάνει .Επιπλεόν, βλέπουμε την τεράστια αλλαγή στην δέσμευση πόρων που κάνει το Directive ΑRRAY\_PARTITION σε σύγκριση με τις άλλες μεθόδους που είναι οι αλλαγές κώδικα, καθώς και η χρήση Directive Pipeline, Unroll και ARRAY\_RESHAPE επηρρεάζουν κατά ελάχιστα τους πόρους που χρησιμοποιεί.

**Πίνακας πόρων:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Μέθοδος** | **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 αποδόσεων:**

|  |  |
| --- | --- |
| **Μέθοδος** | **Latency** |
| *0* | 3766 |
| *1* | 1976 |
| *2* | 1862 |
| *3* | 1696 |
| *4* | 1667 |
| *5* | 509 |
| *6* | 365 |
| *7* | 349 |

Αρχική απόδοση: 3766 Τελική Απόδοση:349

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 6η 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 6ο***

# 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 τρείς 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 εξόδου 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 τέτοιον 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. Ένα ερώτημα που παραμένει ανοιχτό, είναι αν υπάρχει τρόπος σύνθεσης ενός κώδικα γραμμένο σε γλώσσα υψηλού επιπέδου, σε γλώσσα περιγραφής υλικού, ο οποίος να είναι εύκολα αναγνώσιμος και κατανοητός από τους προγραμματιστές και επιπλέον πώς θα μπορούμε, να έχουμε κυκλώματα με υψηλή απόδοση με την χρήση όσο πιο δυνατόν λιγότερων πόρων.

***Κεφάλαιο 7ο***

# 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\_Toturial.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