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 Xilinx Vivado HLS. Πιο συγκεκριμένα, γίνεται ανάλυση της επίδοσης 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 επιδόσεων 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, συμπεραίναμε ότι με την σωστή γραφή του κώδικα καθώς με την χρήση των κάποιων εργαλείων που μας προσφέρει το λογισμικό μπορούμε να φτιάξουμε κυκλώματα ενός ακόμη 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 Xilinx Vivado 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 concluded that with the correct writing of the code, as with the use of some software tools, we can build circuits of yet another complex algorithm with high performance.

**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](#_Toc512200228)

[2.High Level Synthesis 7](#_Toc512200229)

[3. Μελέτη του αλγορίθμου AES 18](#_Toc512200230)

[4.1 Ανάλυση υλοποιήσεων 26](#_Toc512200231)

[4.1.1 Τροποποιήσεις υλοποιήσεων ώστε να γίνουν συνθέσιμοι 27](#_Toc512200232)

[4.1.2 Τροποποιήσεις υλοποιήσεων για να γίνουν ομογενείς 34](#_Toc512200233)

[4.2 Σύγκριση των υλοποιήσεων 41](#_Toc512200234)

[5.Βελτιστοποίηση κυκλώματος 62](#_Toc512200235)

[6.Συμπεράσματα 70](#_Toc512200236)

[7.Βιβλιογραφικές Πηγές 71](#_Toc512200237)

***Κεφάλαιο 1ο***

# 1.Εισαγωγή

Οι βασικές γλώσσες περιγραφής υλικού (Hardware Description Language, HDL) που γράφουν οι προγραμματιστές για να μπορέσουν να προγραμματίσουν πλακέτες FPGA, είναι η Verilog και η VHDL. Οι προγραμματιστές για παρά πολλά χρόνια χρησιμοποιούν αυτές τις γλώσσες που καθιστούσαν πάρα πολύ δύσκολη και χρονοβόρα την διαδικασία να φτιάξουν,καθώς να αποσφαλματώσουν προγράμματα γραμμένα σε HDL σε σύγκριση με ένα πρόγραμμα που θα μπορούσαν να φτιάξουν σε μια γλώσσα υψηλού επιπέδου, πιο φιλική δηλαδή για τον χρήστη που θα μπορούσε να είναι η C ή και η C++. Όλο αυτό άλλαξε, καθώς η εταιρεία Xilinx το 2012 έφτιαξε ένα λογισμικό εν ονόματι Vivado Suite που περιλαμβάνει το εργαλείο HLS (High Level Synthesis). Η συγκεκριμένη εταιρεία και στο παρελθόν είχε φτιάξει λογισμικά που χρησιμοποιούνται από πάρα πολλούς προγραμματιστές, ώστε να προγραμματίζουν πλακέτες FPGA. Για παράδειγμα είχε φτιάξει το Xilinx ISE, που χρησιμοποιούνταν από πολλούς προγραμματιστές εδώ και παρά πολλά χρόνια. Με την τελευταία προσθήκη του HLS στο Vivado έφερε την επανάσταση στο τρόπο προγραμματισμού πλακετών. Πλέον, με αυτό το λογισμικό μπορεί ο κάθε προγραμματιστής να μπορέσει να φτιάξει ένα πρόγραμμα που είναι γραμμένο σε γλώσσα υψηλού επιπέδου για παράδειγμα C, C++ και ακόμη και SystemC να το κάνει σύνθεση ώστε να δημιουργήσει μια υλοποίηση που θα τρέχει αυτομάτως σε μια πλακέτα FPGA, χωρίς να χρειαστεί να γράψει καθόλου κώδικα HDL δηλαδή Verilog ή VHDL. Με την χρήση αυτού του εργαλείου θα μπορεί σώσει ένα σημαντικό μέρος από το χρόνο του, που θα μπορούσε να δαπανήσει, φτιάχνοντας την υλοποίηση αμέσως σε γλώσσα περιγραφής υλικού. Προηγούμενες έρευνες έχουν δείξει ότι το εργαλείο HLS μπορεί αναμφίβολα, να παράξει κυκλώματα για πλακέτες που ήταν γραμμένα σε γλώσσα υψηλού επιπέδου και να λειτουργήσουν σωστά. Δυστυχώς όμως, επειδή αποτελεί ένα καινούριο λογισμικό, δεν γνωρίζουμε αν μπορεί να φτιάξει το ίδιο λογισμικό υλοποίησης με υψηλή απόδοση κυκλώματα, 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 που έχουν οι αλγόριθμοι κρυπτογράφησης και ασφάλειας καθώς και την δυσκολία ώστε να υλοποιηθούν. Σκοπός αυτής της εργασίας είναι να αναλύσουμε την απόδοση διαφόρων υλοποιήσεων ενός αλγόριθμου κρυπτογράφησης που έχει γραφτεί σε γλώσσα υψηλού επίπεδου C. 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. Αυτό που κάναμε είναι ότι αναλύουμε τρείς διαφορετικές υλοποιήσεις του αλγόριθμου αυτού ,καθώς στο τέλος συμπεραίνουμε ότι με την βοήθεια κάποιων εργαλείων που προσφέρει το κύκλωμα καθώς με την κατάλληλη συγγραφή του κώδικα που είναι γραμμένη σε γλώσσα υψηλού επιπέδου μπορούμε να φτιάξουμε κυκλώματα RTL με υψηλή απόδοση, απαντώντας στο ερώτημα που ήταν ανοιχτό για τόσα χρόνια.

Η δομή της εργασίας έχει ως εξής:

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

Στο **Δεύτερο Κεφάλαιο** γίνεται η εισαγωγή του αναγνώστη στο εργαλείο που χρησιμοποιούμε στην εργασία που είναι το Xilinx Vivado 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 καθώς οι τρείς υλοποιήσεις να έχουν κοινά ορίσματα για να γίνει σωστή σύγκριση.

Στο **Πέμπτο κεφάλαιο** ,θα κάνουμε τροποποιήσεις, με διάφορους μεθόδους 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 high-level synthesis (HLS) αποτελεί μια επέκταση του λογισμικού Vivado το οποίο κατασκευάστηκε από την εταιρεία Χilinx τον Απρίλιο του 2012. Το Vivado αποτελεί ένα λογισμικό για τη σχεδίαση και ανάλυση HDL κυκλωμάτων, αντικαθιστώντας το προκάτοχο λογισμικό που είχε φτιάξει το οποίο είναι το Xilinx ISΕ (Integrated Synthesis Environment). Τα πρόσθετα χαρακτηριστικά του Vivado είναι η δυνατότητα ανάπτυξης chip καθώς και το εργαλείο high-level synthesis που θα μελετήσουμε περαιτέρω. Το 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 Vivado Design Suite είναι οι παρακάτω:

* Το Vivado TCL Store αποτελεί ένα σύστημα σεναρίων για την ανάπτυξη πρόσθετων επεκτάσεων για το Vivado. Μπορεί να χρησιμοποιηθεί για να προσθέσει και να τροποποιήσει τις δυνατότητες του Vivado. To TCL που προέρχεται από τις λέξεις Tool Command Language, αποτελεί μια γλώσσα σεναρίων στην οποία βασίζεται το ίδιο το 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 Vivado High-Level Synthesis.
* Το εργαλείο Vivado High-Level Synthesis αποτελεί έναν μεταγλωττιστή υψηλού επιπέδου 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 χειροκίνητη δημιουργία RTL(Register Transfer Level). Το vivado HLS αυξάνει την παραγωγικότητα των προγραμματιστών, καθώς έχει επιβεβαιωθεί ότι υποστηρίζει κλάσσεις, πρότυπα, συναρτήσεις καθώς και διάφορους τελεστές της γλώσσας C++.

**Κατανοώντας το HLS**

Αυτό που κάνει το εργαλείο Xilinx Vivado HLS[3] είναι, ότι κάνει σύνθεση συναρτήσεις που είναι γραμμένες σε γλώσσα υψηλού επιπέδου σε IP Block, τα οποία μπορούν να ενσωματωθούν στο υλικό. Επειδή είναι ενσωματωμένο στο λογισμικό του Vivado, μπορεί να χρησιμοποιηθεί το HLS και από άλλα εργαλεία σχεδιασμού της Xilinx καθώς υποστηρίζει πολλά εργαλεία για την δημιουργία του βέλτιστης υλοποίησης του αλγορίθμου που θα γραφτεί σε γλώσσα υψηλού επιπέδου όπως C. 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 του 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 Vivado HLS μπορεί να πάρει ως εισόδους τα παρακάτω:

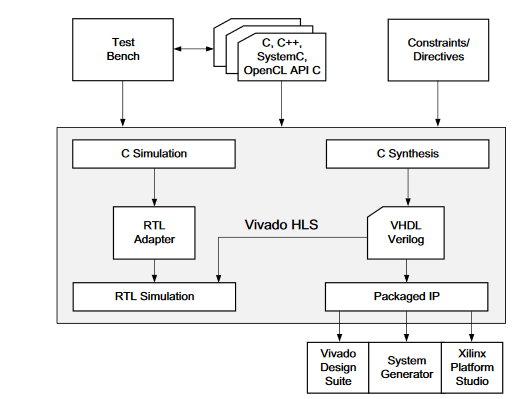
* Συναρτήσεις γραμμένες σε γλώσσα υψηλού επιπέδου όπως C, C++, SystemC. Αποτελούν την βασική είσοδο για το εργαλείο Vivado HLS. Επιπλέον, στις συναρτήσεις μπορούν να εμπεριέχουν και άλλες υπό-συναρτήσεις.
* Ορίσματα. Τα ορίσματα είναι υποχρεωτικά και χρειάζονται για να συμπεριλάβουμε στο κύκλωμα την περίοδο του ρολογιού, την απόκλιση που μπορεί να έχει το ρολόι καθώς και την πλακέτα 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 Vivado HLS 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 το C/RTL συν-προσομοίωση.

Όλα αυτά μπορούν εισαχθούν με την χρήση του γραφικού περιβάλλοντος Vivado (GUI) ή και ακόμη με την χρήση tcl εντολών μέσω της γραμμής εντολών.

Οι εξόδους που παράγει το Vivado HLS είναι τα παρακάτω:

* Αρχεία υλοποίησης RTL που είναι γραμμένα σε γλώσσα περιγραφής υλικού. Αποτελεί την βασική έξοδο του Vivado HLS. Μέσω του Vivado synthesis, μπορούμε να κάνουμε σύνθεση το RTL σε υλοποίηση επιπέδου πυλών (gate - level implementation) και να παράξουμε αρχείο FPGA bitstream που θα χρησιμοποιηθεί ώστε να δουλέψει το παραγόμενο κύκλωμα στην πλακέτα FPGA. Το RTL που θα παραχθεί μπορεί να είναι διαθέσιμο είτε σε μορφή VHDL είτε σε μορφή Verilog.
* Αρχεία ανάλυσης. Αποτελεί την έξοδο που θα βγάλει το κύκλωμα μετά την σύνθεση, C/RTL συν-προσομοίωση και IP packaging.

Στην παρακάτω εικόνα βλέπουμε μια επισκόπηση των εισόδων και εξόδων του εργαλείου Vivado HLS.



**Test Bench, Υποστήριξη γλωσσών, βιβλιοθήκες της C.**

Σε κάθε πρόγραμμα που είναι γραμμένο σε C, η top-level function ονομάζεται main(). Στο Vivado HLS μπορούμε να ορίσουμε οποιαδήποτε υπό-συνάρτηση που είναι κάτω από την main ως top-level function για σύνθεση. Δεν μπορούμε να κάνουμε σύνθεση όμως την top-level function main(). Και επιπλέον πρέπει να ισχύουν όλα τα παρακάτω:

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

**Τest Bench**

Όταν χρησιμοποιούμε το Vivado HLS, είναι πολύ χρονοβόρο να κάνουμε σύνθεση μια συνάρτηση C που δεν λειτουργεί σωστά και μετά να αναλύσουμε τις αναφορές που βγάζει ώστε να ξεκαθαρίσουμε γιατί η συνάρτηση δεν δουλεύει σωστά όπως περιμέναμε. Για να αυξήσουμε την παραγωγικότητα στο συγκεκριμένο χρόνο, μπορούμε να χρησιμοποιήσουμε τα test bench ώστε να εξακριβώσουμε αν δουλεύει σωστά η συνάρτηση πριν την κάνουμε σύνθεση.

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

Το Vivado HLS χρησιμοποιεί τα test benches ώστε να μεταγλωττίσει και να εκτελέσει το C- Simulation. Κατά την διάρκεια της μεταγλώττισης μπορούμε να διαλέξουμε την επιλογή **Launch Debugger** για να κάνουμε αποσφαλμάτωση του προγράμματος σε περίπτωση που χρειαστεί.

Γλώσσες που υποστηρίζει:

* C/C++
* OpenCL
* SystemC

To Vivado HLS υποστηρίζει διάφορα εργαλεία της C, C++, και SystemC καθώς πολλά είδη τύπων για την κάθε γλώσσα όπως διπλής ακρίβειας και πραγματικούς αριθμούς. Ωστόσο, δεν μπορεί να γίνει σύνθεση σε όλα τα εργαλεία που προσφέρουν οι παραπάνω γλώσσες όπως είναι τα παρακάτω:

* Δυναμική κατανομή μνήμης. Μια πλακέτα FPGA έχει ένα σταθερό σύνολο πόρων και η δυναμική δημιουργία και απελευθέρωση πόρων μνήμης δεν υποστηρίζεται.
* Λειτουργίες λειτουργικού συστήματος (OS). Όλα τα δεδομένα από και προς τις πλακέτες 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 test benches.

Εκτός από όλα αυτά, το Vivado HLS υποστηρίζει και παρά πολλές βιβλιοθήκες της γλώσσας C, οι οποίες περιλαμβάνουν συναρτήσεις και άλλα εργαλεία τα οποία είναι χρήσιμα ώστε να γίνει βελτιστοποίηση της υλοποίησης του κυκλώματος σε μια πλακέτα FPGA. Η χρήση αυτών των βιβλιοθηκών συμβάλλει στην εξασφάλιση υψηλής ποιότητας αποτελεσμάτων με αποτέλεσμα να παράξουμε ένα κύκλωμα 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 Vivado HLS 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 πλακέτας ώστε να μεγιστοποιείται η χρησιμότητα των πόρων shift register LUT ( Look Up tables)

**Σύνθεση, βελτιστοποίηση και ανάλυση**

Το Vivado HLS 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. Μπορούμε να αναλύσουμε και να συγκρίνουμε τα αποτελέσματα κάθε υλοποίησης μέσω του γραφικού περιβάλλον του Vivado HLS.

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

1. Δημιουργία project με μια αρχικοποιημένη υλοποίηση δηλαδή γραμμένη σε γλώσσα υψηλού επιπέδου.
2. Έλεγχος της προσομοίωσης σε C ώστε να μην παρουσιάσει κανένα σφάλμα.
3. Σύνθεση της υλοποίησης ώστε να λάβουμε τα αντίστοιχα αποτελέσματα
4. Ανάλυση των αποτελεσμάτων.

Μετά την ανάλυση των αποτελεσμάτων, μπορούμε να δημιουργήσουμε καινούρια λύση- υλοποίηση για το project με διαφορετικά ορίσματα και directives ώστε να κάνουμε σύνθεση την νέα υλοποίηση. Επιπλέον ,μπορούμε να επαναλάβουμε αυτήν την διαδικασία μέχρι το κύκλωμα μας έχει την επιθυμητή απόδοση. Με την χρήση των πολλαπλών λύσεων μπορούμε να εύκολα να αξιολογήσουμε τις διαφορετικές υλοποιήσεις καθώς αναπτύσσουμε την βέλτιστή υλοποίηση για το κύκλωμα μας.

**Βελτιστοποίηση**

Με την χρήση του Vivado HLS, μπορούμε να εφαρμόσουμε στις υλοποιήσεις διάφορα directives ώστε να το βελτιστοποιήσουμε. Κάποια από αυτά είναι:

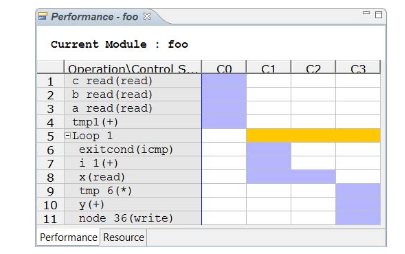
* Μπορούμε να κάνουμε μια εντολή να εκτελεστεί με την χρήση διοχέτευσης (pipeline), επιτρέποντας να εκτελεστεί η επόμενη εντολή πριν ολοκληρωθεί η τρέχουσα εντολή αξιοποιώντας πολύτιμο χρόνο.
* Μπορούμε να καθορίσουμε το χρόνο που χρειάζεται για την ολοκλήρωση των συναρτήσεων και βρόγχων.
* Μπορούμε να καθορίσουμε το όριο των πόρων που χρησιμοποιούνται στο κύκλωμα.
* Μπορούμε να καταργήσουμε τις εξαρτήσεις των εντολών του κώδικα και να εκτελέσουμε κάποιες συγκεκριμένες λειτουργίες. Για παράδειγμα αν είναι εφικτό να αγνοήσουμε αρχικές τιμές που έχουν κάποιες μεταβλητές σε ένα κώδικα όπως είναι μια ροή video, επιτρέποντας να διαβάσει την μνήμη πριν τα γράψει σε αυτήν αν τα αποτελέσματα που βγάλει είναι αποδοτικότερα.
* 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.

Με την χρήση του γραφικού περιβάλλον του Vivado μπορούμε εισάγουμε τα directives που μας ενδιαφέρουν ώστε να βελτιστοποιήσουμε το κύκλωμα.

**Ανάλυση**

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

Όπως φαίνεται στο παρακάτω παράδειγμα:



Αυτή η καρτέλα δείχνει:

* C0: Το πρώτο στάδιο που διαβάζει τις μεταβλητές c, b, a καθώς και την πρόσθεση.
* C1 και C2: Το κύκλωμα μπαίνει σε βρόγχο και ελέγχει τον μετρητή και την συνθήκη για να βγει από την βρόγχο. Το κύκλωμα μετά διαβάζει την μεταβλητή x και βλέπουμε ότι χρειάζεται δύο κύκλους ρολογιού. Αυτό χρειάζεται γιατί το κύκλωμα αποκτάει πρόσβαση στην block Ram καθώς καταναλώνει ένα κύκλο ρολογιού για να βρει την διεύθυνση και έναν ακόμη για να διαβάσει τα δεδομένα.
* C3: Το κύκλωμα εκτελεί τις υπολογισμούς καθώς γράφει την έξοδο στον y. Έπειτα, ο βρόγχος ξεκινάει από την αρχή.

**Επαλήθευση RTL.**

Με την χρήση ενός test bench που είναι γραμμένο σε γλώσσα C , μπορούμε να επαληθεύσουμε εάν λειτουργικά είναι ίδια το παραγόμενο κύκλωμα RTL με το πρωτότυπο που είναι γραμμένο σε C. To test bench αυτό που κάνει, είναι να επαληθεύει την έξοδο από την top-level function που την κάναμε σύνθεση, επιστρέφοντας μηδέν αν RTL είναι λειτουργικά ίδιο με το πρωτότυπο. Αυτό χρησιμοποιείται και στο C simulation και στο C/RTL συν-προσομοίωση. Εάν το αποτέλεσμα που βγάλει είναι διάφορο του μηδέν τότε το Vivado HLS εμφανίζει ότι απέτυχε η προσομοίωση.

**Συμβουλές/προτεινόμενοι μέθοδοι:**

Όσον αφορά τον κώδικα:

Πρώτο μέλημα μας είναι να λιγοστεύσουμε τα δεδομένα που διαβάζει η κύρια συνάρτηση, δηλαδή τα βασικές εισόδους που χρειάζεται η συνάρτηση που θα γίνει η σύνθεση για να λειτουργήσει. Αυτά τα δεδομένα θα μεταφραστούν σε I/O Ports στο κύκλωμα. Όταν τα δεδομένα που έχει ο αλγόριθμος που χρησιμοποιούμε διαβαστούν σε ένα block της πλακέτας, μπορούν εύκολα να εύκολα να τα διαχειριστούμε παράλληλα αλλά τα input ports μπορούν να γίνουν «bottleneck» δηλαδή να καθυστερήσουν το ολικό κύκλωμα με αποτέλεσμα να μειωθεί η απόδοση.

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 τα I/O Ports είναι περιορισμένα σε αριθμό και μπορεί να καθυστερήσουν το ολικό κύκλωμα, καθώς μπορούν να γίνουν και αυτά η καθυστέρηση, «bottleneck» για το ολικό κύκλωμα. Οι πίνακες μπορούν να κατανεμηθούν σε μικρότερους πίνακες και ακόμη και σε ξεχωριστές καταχωρητές, αλλά η διαμέριση των μεγάλων πινάκων θα χρειαστεί 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 , αντί να εκτελούμε άλλες εντολές υπό συνθήκη ακόμη και αν 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 το οποίο είναι υπό συνθήκη θα οδηγήσει στην αύξηση της απόδοσης του κυκλώματος μας.

Εκτός από όλα αυτά πρέπει να λάβουμε υπόψιν και τα outputs. 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. Τα I/O Ports μπορούν να καθυστερήσουν την συνολική απόδοση του κυκλώματος που πρόκειται να δημιουργηθεί μετά την σύνθεση. Η δημιουργία περισσότερων ports τα οποία μπορεί να είναι είτε είσοδο είτε για έξοδο, επιβαρύνει περισσότερο το σύστημα[7].

Όσον αφορά την γλώσσα προγραμματισμού:

Όπως αναφέρουμε πιο πάνω, το εργαλείο Vivado HLS μπορεί να μετατρέψει σε υλοποιήσεις για την πλακέτα, 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 σχεδιασμένη, διαμορφωμένη να σχεδιάζει 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 HDL[8].Όμως, η πολυπλοκότητα της, την κατιστά δύσκολη για παραγωγή κώδικα.

Όσον αφορά την υλοποίηση σε RTL.

Για να αυξήσουμε την απόδοση του κυκλώματος, που παράχθηκε μετά την σύνθεση μπορούμε να χρησιμοποιήσουμε διάφορους οδηγούς που μας παρέχει το εργαλείο Vivado HLS. 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 directive μπορεί οδηγήσει στην χρήση πιο πολλών πόρων του συστήματος με αποτέλεσμα να μειωθεί η απόδοση του συστήματος.

ARRAY\_MAP: Μπορούμε να συνδυάσουμε μικρότερους πίνακες σε ένα μεγαλύτερο ώστε να μειώσουμε την χρήση των Block RAM.

ARRAY\_PARTITION: Μπορούμε να διαμερίσουμε μεγάλους πίνακες σε πολλά μικρότερα και ακόμη σε καταχωρητές, ώστε να αυξήσουμε την προσπέλαση στα δεδομένα καθώς να μειώσουμε τις block RAM καθυστερήσεις.

ARRAY\_RESHAPE: Μπορούμε να επανασχεδιάσουμε έναν πίνακα που μπορεί είτε να έχει ένα στοιχείο είτε πολλά σε ένα πίνακα μεγαλύτερου μεγέθους. Χρήσιμο για αυξήσουμε την προσπέλαση στην Block RAM χωρίς να χρησιμοποιήσουμε και άλλο τέτοιο πόρο.

CLOCK: Για τις υλοποιήσεις που έχουν γραφτεί σε SystemC, μπορούμε να φτιάξουμε νέα ρολόγια πάνω στα SC\_MODULE χρησιμοποιώντας αυτό το Directive.

DATA\_PACK: Ομαδοποιεί τα ορίσματα σε δομή σε ένα scalar χρησιμοποιώντας μεταβλητή μεγαλύτερου μεγέθους.

DEPENDENCE: Χρησιμοποιείται ώστε να παρέχει επιπλέον πληροφορίες με σκοπό να ξεπεράσει εξαρτήσεις μεταξύ των βρόγχων 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 τα RTL ports έχουν δημιουργηθεί από την περιγραφή της συνάρτησης.

LATENCY: Μπορούμε να ορίσουμε το ελάχιστο και μέγιστο latency που χρησιμοποιείται στο συγκεκριμένο πόρο.

LOOP\_FLATEN: Επιτρέπει τους εμφωλευμένους βρόγχους να εκτελεστούν σε ένα βρόγχο αλλά με αυξημένο καθυστέρηση.

LOOP\_MERGE: Ενώνει διαδοχικούς βρόγχους για μειώσει το latency, αυξάνοντας το μοιρασμό των πόρων.

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 ότι 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 το directive, μπορούμε να προσθέσουμε ή να αφαιρέσουμε σε μια μεταβλητή το state δηλαδή αν είναι global ή static.

RESOURCE: Μπορούμε να ορίσουμε μια συγκεκριμένη βιβλιοθήκη να χρησιμοποιηθεί ώστε να υλοποιήσουμε μια μεταβλητή που μπορεί να είναι πίνακας, αριθμητικός τελεστής και 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 top-level συνάρτηση για σύνθεση ορίζεται στα project settings. Αυτός ο οδηγός μπορεί να χρησιμοποιηθεί ώστε να ορίσουμε οποιαδήποτε ως top-level. Αυτό μας διευκολύνει ώστε να βρούμε την καλύτερη λύση για το κύκλωμα χωρίς να φτιάξουμε νέο project.

UNROLL: Κάνει Unroll βρόγχους τύπου for ώστε να εκτελέσει τις εντολές που μπορεί να είναι ανεξάρτητες η μία από την άλλη σειριακά αλλά και μπορεί και παράλληλα να χρησιμοποιηθεί μαζί με άλλο directive.

Όλα αυτά που αναλύσαμε πιο πάνω θα τα χρησιμοποιήσουμε στο δικό μας αλγόριθμο Αυτό που κάνουμε είναι να αναλύσουμε τις διαφορετικές υλοποιήσεις που έχουμε φτιάξει για τον αλγόριθμο αυτόν, καθώς να σχολιάσουμε τους λόγους για τους οποίους είναι αποδοτικότερη η πρώτη υλοποίηση από τις άλλους δύο. Στην συνέχεια, θα χρησιμοποιήσουμε διάφορες τεχνικές ώστε να βελτιστοποιήσουμε την απόδοση της πιο αργής υλοποίησης. Στο τέλος θα μελετήσουμε διάφορες τεχνικές ώστε να διευκολύνουμε αυτούς που πρόκειται να χρησιμοποιήσουν το εργαλείο αυτό , να γράψουν αποδοτικότερες υλοποιήσεις για το κύκλωμα που σκοπεύουν να φτιάξουν.

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

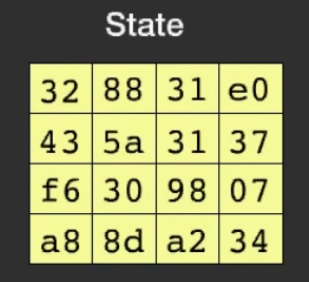
***Κεφάλαιο 3ο***

# 3. Μελέτη του αλγορίθμου AES

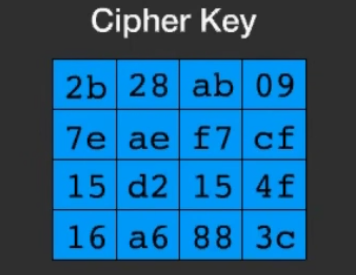
Το όνομα αυτού νου του αλγορίθμου βγαίνει από τις λέξεις Advanced Encryption Standard. Το εναλλακτικό του όνομα είναι Rijndael και δημιουργήθηκε από δύο Βέλγους κρυπτογράφους το 2001. Χρησιμοποιήθηκε στην αρχή από την αμερικάνικη κυβέρνηση και τώρα χρησιμοποιείται διεθνώς από πολλούς οργανισμούς. Δημιουργήθηκε για να αντικαταστήσει έναν άλλο αλγόριθμο κρυπτογράφησης εν ονόματι 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 ασφαλές 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 τους άλλους συνηθισμένους 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.

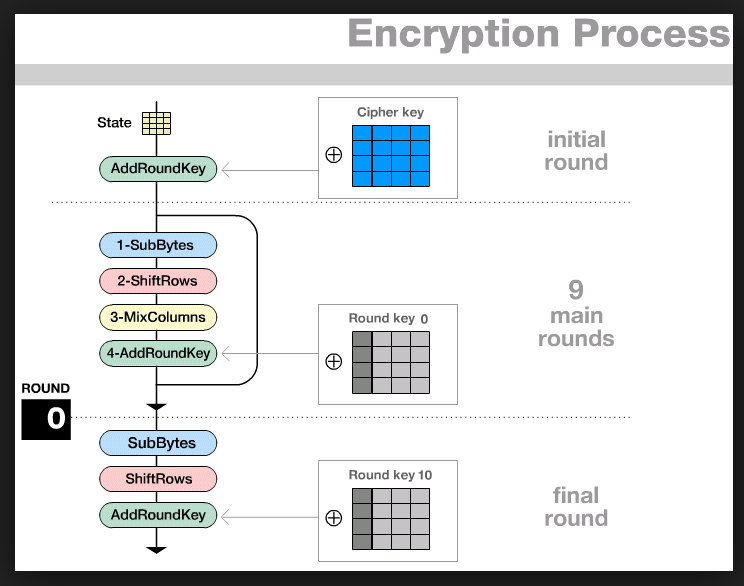
Ο παρακάτω αλγόριθμος χρησιμοποιεί κρυπτογράφηση ανά block, καθώς κρυπτογραφεί 16 bits την φορά όπως βλέπουμε παρακάτω:



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



Διαδικασία:



Όπως βλέπουμε και στην εικόνα, ο αλγόριθμος AES για την κρυπτογράφηση του μηνύματος εκτελεί τις ακόλουθες συναρτήσεις:

* AddRoundKey
* SubBytes
* MixColumns
* AddRoundkey

Υπάρχουν 3 είδη κατηγοριών που μπορεί να έχουμε σε αυτόν τον αλγόριθμο.

Να είναι το κείμενο (Plain Text):

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

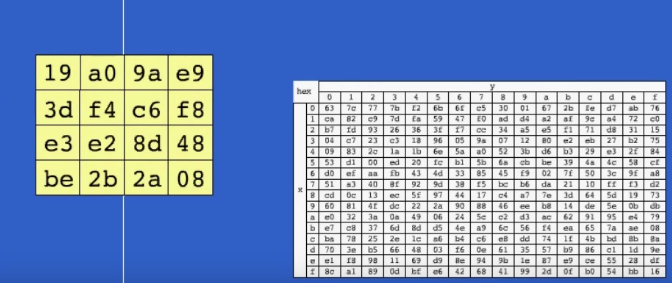
Για τις παραπάνω κατηγορίες οι αριθμοί των γύρων που βλέπουμε στην παραπάνω εικόνα αλλάζει.

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:

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

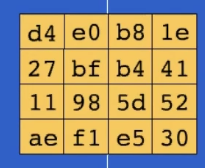
**Ανάλυση συναρτήσεων:**

1. 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.

Στο τέλος θα γίνει σύμφωνα με τον πίνακα έτσι:



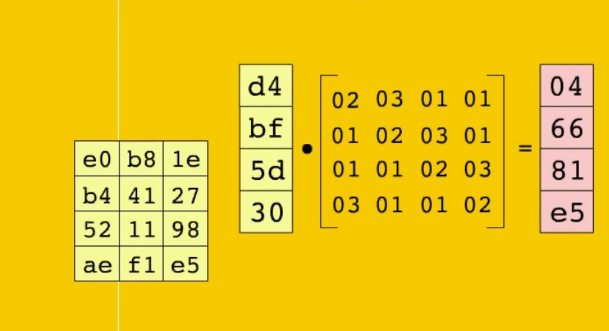
1. ShiftRows:

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 και θα γίνει κάπως έτσι:

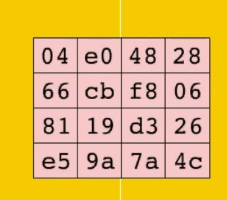


1. MixColumns:

Σε αυτήν την διαδικασία παίρνουμε την πρώτη στήλη του block και την πολλαπλασιάζουμε με έναν συγκεκριμένο πίνακα όπως φαίνεται και στην εικόνα:



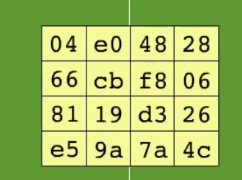
Συνεχίζουμε και με τις άλλες στήλες, ώστε το αποτέλεσμα να είναι κάπως έτσι:



1. AddRoundkey:

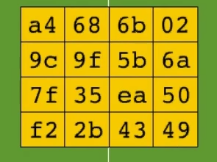
Σε αυτήν την συνάρτηση παίρνουμε το παραγόμενο κλειδί (θα δείξουμε μετά πως υπολογίζεται) και το προσθέτουμε στο state που έχουμε.

Πχ . Έστω το state που είχαμε υπολογίσαμε πιο πριν:



Και το παραγόμενο κλειδί (Round Key):

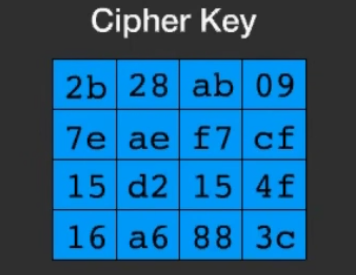
Μετά την πρόσθεση θα γίνει έτσι το state μας:



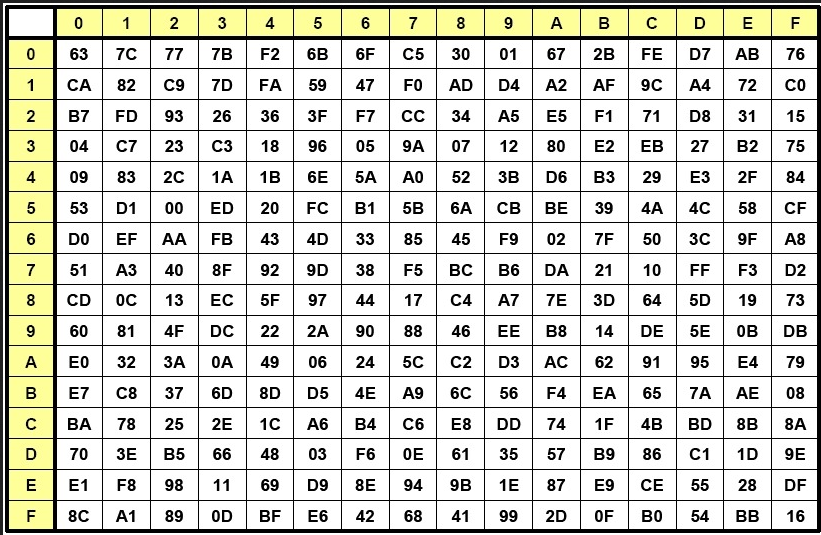
1. Υπολογισμός παραγόμενου κλειδιού

Για την παραγωγή κλειδιών θα χρησιμοποιήσουμε το αρχικό κλειδί που δίνει ο χρήστης καθώς τους πίνακες S-BOX και Rcon που είναι τα παρακάτω:

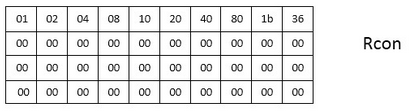
Αρχικό κλειδί που εισάγαμε:



S-BOX:



Rcon:



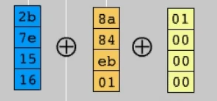
Πρώτον, παίρνουμε την τελευταία στήλη του cypher key μας, κάνουμε ένα rotate προς τα πάνω ώστε να γίνει έτσι:



Δεύτερον, αντικαθιστούμε τα bits που έχουμε , με τα bits του πίνακα S-BOX ώστε να γίνει έτσι:



Τρίτον, προσθέτουμε την πρώτη στήλη του cypher key με την στήλη που παράξαμε στο 2ο βήμα καθώς με την 1 στήλη του πίνακα Rcon, δηλαδή όπως φαίνεται παρακάτω:



Η πρόσθεση αυτή θα μας βγάλει την πρώτη στήλη του νέου round key.



Για να βγάλουμε τις άλλες στήλες θα πρέπει να προσθέσουμε την παραπάνω στήλη που παράξαμε με τις αντίστοιχες 2η, 3η και 4η στήλη του αρχικού cypher key και το αποτέλεσμα που θα μας βγάλει θα είναι το παρακάτω:

Νέο round Key:



Για να φτιάξουμε το 2ο round key ακολουθεί η ίδια περίπου διαδικασία , καθώς οι διαφορές που συναντάμε είναι ότι αντί να πάρουμε το block του cypher key που έβγαλε ο χρήστης, θα πάρουμε το νέο round key που παράξαμε καθώς και θα προσθέσουμε σε αυτό όχι την 1η στήλη του Rcon αλλά την 2η .

***Κεφάλαιο 4ο***

# Ανάλυση υλοποιήσεων

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

Η γλώσσα προγραμματισμού που χρησιμοποιούμε στις τρείς υλοποιήσεις είναι η γλώσσα C. Ο λόγος που χρησιμοποιούμε αυτήν την γλώσσα, είναι επειδή είναι πιο φιλική προς στον χρήστη από την SystemC καθώς έχει λιγότερη πολυπλοκότητα όσον αφορά την υλοποίηση του αλγορίθμου. Για την πρώτη υλοποίηση χρησιμοποιήσαμε τον κώδικα tiny\_aes[10], στην δεύτερη τον encrypt[12] και στην τρίτη τον aes[11]. Φυσικά, έπρεπε να κάνουμε κάποιες τροποποιήσεις στους παραπάνω κώδικες ώστε να μπορέσουμε να κάνουμε σύνθεση σε RTL με σωστό λειτουργικά τρόπο.

Αυτές οι τροποποιήσεις γίνονται σύμφωνα με το εγχειρίδιο του Vivado HLS[3]. Με βάση αυτό μπορέσαμε να καταλάβουμε ποιες εντολές μπορεί να κάνει σύνθεση το εργαλείο που χρησιμοποιούμε καθώς με ποιον τρόπο θα μπορέσουμε να γράψουμε τον κώδικα με τέτοιον τρόπο ώστε ο κώδικας που είναι γραμμένος σε γλώσσα υψηλού επιπέδου C να μετατραπεί εξ ολοκλήρου με σωστό τρόπο σε RTL.

## Τροποποιήσεις υλοποιήσεων ώστε να γίνουν συνθέσιμοι

**1ος κώδικας:**

Όσον αφορά τον 1ο κώδικα:

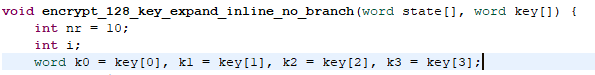
Δεν κάνουμε σημαντικές αλλαγές στον κώδικα καθώς είναι σχετικά έτοιμος για σύνθεση. Κάποιες χρήσιμες αλλαγές που πρέπει να κάνουμε αφορά την μνήμη.

Πιο συγκεκριμένα τα σφάλματα που συναντήσαμε κατά την σύνθεση του 1ου κώδικα είναι τα παρακάτω:

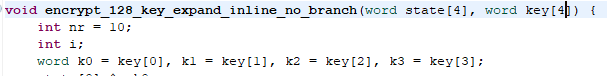
ERROR: [SYNCHK 200-61] ../NotEdited/1st/aes.c:119: unsupported memory access on variable 'key' which is (or contains) an array with unknown size at compile time.

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.

Αυτό το σφάλμα το έβγαλε στην παρακάτω γραμμή:



Για να το λύσουμε απλώς κάναμε τις παρακάτω αλλαγές:



Δηλαδή καθορίσαμε το μέγεθος της μνήμης του state και του key για το size που ορίζεται παρακάτω δηλαδή 4.

Μετά αυτές τις αλλαγές μπορέσαμε να κάνουμε σύνθεση το κύκλωμα σε RTL.

Κάποιες αλλαγές που κάνουμε επίσης είναι ότι αφαιρούμε κάποιες συναρτήσεις τις οποίες δεν τις χρειαζόμαστε για την σύνθεση. Πιο συγκεκριμένα ο συγκεκριμένος κώδικας περιλαμβάνει 4 συναρτήσεις και είναι οι παρακάτω:

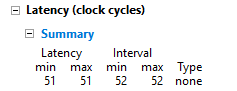
1. encrypt\_128\_key\_expand\_inline
2. encrypt\_128\_key\_expand\_inline\_no\_branch
3. encrypt\_192\_key\_expand\_inline\_no\_branch
4. encrypt\_256\_key\_expand\_inline\_no\_branch

Επειδή και στις άλλες υλοποιήσεις, θα εξετάσουμε όπως είπαμε μόνο την περίπτωση με την χρήση 128 bit θα χρειαστούμε από τις τέσσερις συναρτήσεις, τις δύο πρώτες.

Εν συνέχεια, επειδή εκ ζητούμε την υλοποίηση με το καλύτερη απόδοση θα διαλέξουμε να έχουμε μόνο την δεύτερη συνάρτηση καθώς δεν χρησιμοποιεί διακλαδώσεις.

Άρα για τον 1ο κώδικα θα χρησιμοποιήσουμε μόνο την συνάρτηση **encrypt\_128\_key\_expand\_inline\_no\_branch()** και συνεπώς βγάζουμε τις υπόλοιπες.

To latency του 1ης υλοποίησης είναι 51.



**2ος κώδικας:**

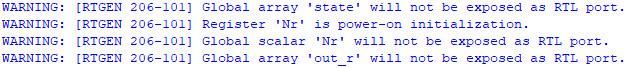
Όσον αφορά τον 2ο κώδικα:

Απ’ ότι βλέπουμε ο κώδικας αυτός εμπεριέχει και την συνάρτηση main. Η main συνάρτηση πρέπει να είναι μόνο στο testbench. Για αυτόν τον λόγο, χωρίζουμε το κώδικα σε δύο αρχεία που είναι θα είναι testbench και το AES\_Encrypt.c. Στο testbench θα μπει η συνάρτηση main, καθώς στο AES\_Encrypt.c θα μπουν οι υπόλοιπες συναρτήσεις. Επιπλέον, πρέπει να ορίσουμε την συνάρτηση για την 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 γίνουν τα παραπάνω, και για αυτόν τον λόγο μπαίνουν στο testbench.

Επιπλέον, ο κώδικας περιλαμβάνει πολλές global μεταβλητές όπως βλέπουμε παρακάτω



Για να διορθώσουμε τα παραπάνω warnings, αυτό που κάναμε είναι εκχωρήσουμε στις διάφορες συναρτήσεις τα ορίσματα που αποτελούν global όπως φαίνεται παρακάτω:

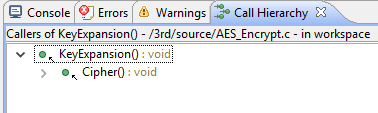


Κάνοντας αυτό σε όλες τις συναρτήσεις, δεν χρειάζεται να ορίσουμε στην αρχή τις global μεταβλητές και έτσι αποφεύγουμε αυτό το πρόβλημα.

Λύνοντας αυτό το πρόβλημα, θα μπορέσουμε να σιγουρέψουμε ότι οι μεταβλητές αυτές που χρειάζονται για το αλγόριθμο θα προσπελαθούν σωστά ώστε να βγάλουμε στο τέλος τα σωστά συμπεράσματα καθώς σιγουρευόμαστε ότι και αυτά θα γίνουν σύνθεση.

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 παρακάτω εικόνα:

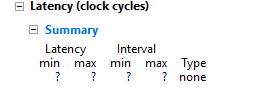


Δηλαδή βλέπουμε ότι εδώ ότι η Cipher καλεί την **KeyExpansion()**.

Άρα μαζί με την σύνθεση της συνάρτησης **Cipher()**,θα γίνει σύνθεση και η **KeyExpansion()**.

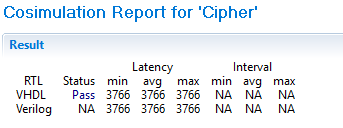
Επιπλέον, βλέπουμε ότι και οι άλλες συναρτήσεις καλούνται από την **Cipher()**

Μετά από αυτές τις αλλαγές μπορέσαμε να κάνουμε επιτυχή σύνθεση με αποτέλεσμα το παρακάτω:



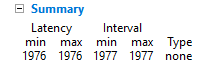
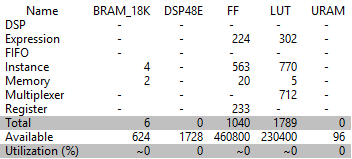
Δυστυχώς όμως, επειδή είναι μεταβλητός ο αριθμός των επαναλήψεων που κάνει η κύρια συνάρτηση **Cipher()** δεν μπορεί να υπολογίσει το latency.

Για αυτόν τον λόγο θα χρησιμοποιήσουμε την συν-προσομοίωση ώστε να υπολογίσουμε το latency του παραπάνω κυκλώματος το οποίο είναι σύμφωνα με την εικόνα 3766.



Ο λόγος που χρησιμοποιούμε την συν-προσομοίωση για να βρούμε το τελικό latency είναι επειδή κατά την διάρκεια της προσομοίωσης διαβάζει την μεταβλητή το πρόγραμμα που καθορίζει τον αριθμό των επαναλήψεων με αποτέλεσμα να μπορέσει να υπολογίσει το συνολικό latency. Σε αντίθεση επειδή ο αριθμός των επαναλήψεων, πιο συγκεκριμένα το Nr και το Nk τα λαμβάνει από το testbench, στην σύνθεση δεν μπορεί να υπολογίσει την συνολική απόδοση.

Επειδή, εμείς θα ασχοληθούμε με τον αλγόριθμο AES για 128 bit, ο αριθμός Nk, Nr είναι στατικός και δεν χρειάζεται να καλούμαι από την κύρια συνάρτηση. Βγάζοντας τις μεταβλητές αυτές, το latency πήγε 1976 δηλαδή κατέβηκε κατά 47.53%.Στην συνέχεια, βλέπουμε τι υλικό δεσμεύει.



**3ος κώδικας:**

Όσον αφορά τον 3ο κώδικα:

Από τι βλέπουμε ο 3ος κώδικας περιλαμβάνει πολλά αρχεία τα οποία είναι πέντε.

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 AES όσον αφορά την κρυπτογράφηση.

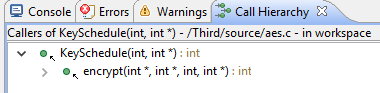
Όλες τις υπόλοιπες συναρτήσεις θα τις χρειαστούμε γιατί μέσω αυτές μπορεί δουλέψει σωστά ο παραπάνω αλγόριθμος.

Αρχικά πρέπει να διαλέξουμε την συνάρτηση την οποία θα επιλέξουμε για να γίνει η σύνθεση η οποία αποτελεί η συνάρτηση **encrypt().**

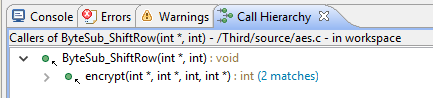
Εν συνέχεια, πρέπει να σιγουρέψουμε ότι όλες οι συναρτήσεις που θα χρησιμοποιήσουμε για την υλοποίηση του αλγορίθμου θα πρέπει να είναι στην ιεραρχία της συνάρτησης που πρόκειται να γίνει σύνθεση.

Δηλαδή όπως βλέπουμε παρακάτω:

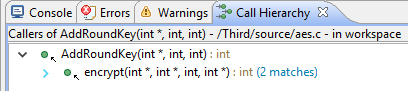
Όσον αφορά την 1η συνάρτηση **KeySchedule()**:



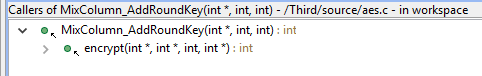
Όσον αφορά την 2η συνάρτηση **ByteSub\_ShiftRow()**:



Όσον αφορά την 3η συνάρτηση **AddRoundKey()**:



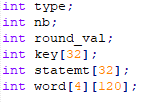
Όσον αφορά την 4η συνάρτηση **MixColumn\_AddRoundKey()**:



Από τις παραπάνω εικόνες βλέπουμε ότι όλες οι συναρτήσεις ανήκουν στην ιεραρχία την συνάρτησης **encrypt()**.

Στην συνέχεια παρατηρήσαμε, ότι στα παραπάνω αρχεία υπάρχουν εντολές του συστήματος όπως printf, τα οποία δεν μπορούν αν μπουν στην σύνθεση και για αυτόν τον λόγο τα βγάλαμε.

Εκτός από όλα αυτά, είδαμε ότι υπάρχουν διάφορες μεταβλητές global, οι οποίες και αυτές να πρέπει μπουν στην σύνθεση για να προσπελαθούν σωστά και είναι τα παρακάτω:

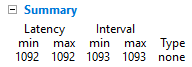
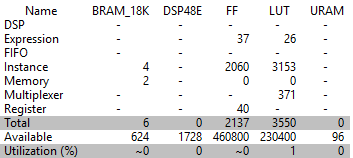


Για αυτόν τον λόγο, τις εισαγάγαμε στις συναρτήσεις που είναι κάτω από την κυριαρχία της κύριας που θα γίνει η σύνθεση, ώστε να γίνουν σύνθεση RTL και αυτές.

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 αλγόριθμος είναι έτοιμος για σύνθεση.

Το Αποτέλεσμα της σύνθεσης είναι το παρακάτω και παρατηρούμε ότι έχει 1092 latency καθώς βλέπουμε τους πόρους που δεσμεύει το παρακάτω κύκλωμα.



Στην συνέχεια, για να μπορέσουμε να κάνουμε την σύγκριση τον κυκλωμάτων ορθά, θα πρέπει να δέχονται ίδια, όσον αφορά τα δεδομένα, από τα testbenches δηλαδή να έχουν κοινά ορίσματα καθώς να βγάζουν μετά την σύνθεση τα παρόμοια inputs και outputs ports.

Στην αρχή θα δούμε, τι δεδομένα δέχονται οι συναρτήσεις που θα γίνουν σύνθεση από τα testbenches.

## Τροποποιήσεις υλοποιήσεων για να γίνουν ομογενείς

Δεδομένα που στέλνει το testbench:

**Κύκλωμα 1:**

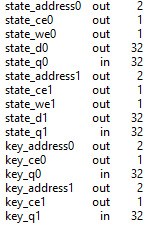
Όσον αφορά το κύκλωμα 1, η συνάρτηση που γίνεται σύνθεση είναι η **encrypt\_128\_key\_expand\_inline\_no\_branch()**.



Άρα θα δούμε τι ορίσματα παίρνει αυτή η συνάρτηση και σύμφωνα από παρακάτω, βλέπουμε ότι παίρνει μεταβλητή state και key οι οποίες είναι τύπου word.



Και παρακάτω βλέπουμε τα ports που παράγει το κύκλωμα.



**Κύκλωμα 2:**

Όσον αφορά το κύκλωμα 2, η συνάρτηση που γίνεται σύνθεση είναι η cipher().



Άρα θα δούμε τι ορίσματα παίρνει αυτή η συνάρτηση και σύμφωνα από παρακάτω, βλέπουμε ότι παίρνει μεταβλητή out η οποία είναι τύπου unsigned char.



.



Και παρακάτω βλέπουμε τα ports που παράγει το κύκλωμα.



**Κύκλωμα 3:**

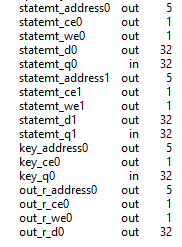
Όσον αφορά το κύκλωμα 3, η συνάρτηση που γίνεται σύνθεση είναι η aes\_main().



Άρα θα δούμε τι ορίσματα παίρνει αυτή η συνάρτηση και σύμφωνα από παρακάτω, βλέπουμε ότι παίρνει μεταβλητές stamt,key,out τα οποία είναι όλα τύπου int.



Και παρακάτω βλέπουμε τα ports που παράγει το κύκλωμα.



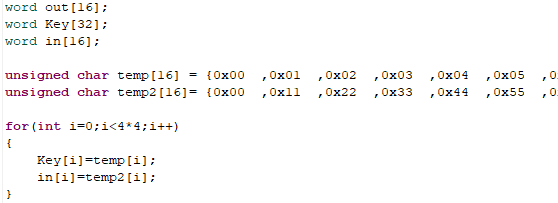
Εφόσον είδαμε τα είδαμε, θα προσπαθήσουμε να όλες οι συναρτήσεις να δέχονται, τις ίδιες εισόδους και εξόδους.

Για να γίνει αυτό, θα προσπαθήσουμε να κάνουμε τα κυκλώματα 2 και 3 , να μοιάσουν με το πρώτο, αφού το πρώτο έχει την καλύτερη απόδοση. Άρα θα κάνουμε το κύκλωμα 2 και 3 να έχουν ως ορίσματα στην κύρια συνάρτηση μεταβλητές τύπου word οι οποίες θα είναι state και key.

**Κύκλωμα 2:**

Για να γίνει αυτό στο κύκλωμα 2 θα πρέπει να γίνουν οι εξής αλλαγές:

θα πρέπει να βάλουμε την είσοδο καθώς και το κλειδί μέσα στο testbench, στην αρχή ήταν μέσα στην συνάρτηση που θα γίνει η σύνθεση. Έπειτα να κάνουμε τύπου word τις μεταβλητές in, key καθώς και το out.



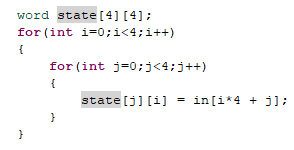
Έπειτα θα βάλουμε τους πίνακες, στα in και key χωρίς την χρήση loop:



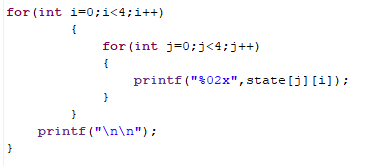
Στην συνέχεια, θα πρέπει να αντικαταστήσουμε το out και το key με μία μεταβλητή που θα είναι η state και θα είναι τύπου word. Αυτή η αλλαγή πρέπει να γίνει και στα δύο αρχεία testbench και AES\_Encrypt.c.



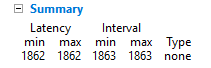
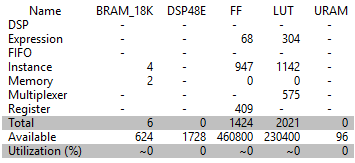
Στην αρχή θα πάρουμε να πάρουμε τον κώδικα που μεταφέρει το in στο state από τον κώδικα από τον AES\_Encrypt.c και θα το βάλουμε στο testbench.



Τώρα, θα αλλάξουμε τα input της συνάρτησης και θα βάλουμε αντί για out, state και key.Καθώς θα μεταφέρουμε τον κώδικα που υπάρχει στον AES\_Encrypt.c, στο testbench και με χρήση printf θα μπορέσουμε να δούμε το state που αποτελεί το κρυπτογραφημένο μήνυμα.



Τελευταία αλλαγή που κάναμε, είναι μετατρέψουμε όλα τα key, state σε τύπο word.Μετά από αυτές τις αλλαγές το latency αυτού νου του κυκλώματος πήγε 1862 καθώς και τους πόρους που χρησιμοποιεί. Διαπιστώνουμε εκτός από την αύξηση της απόδοσης, τα Flip Flop αυξήθηκαν κατά 1424 καθώς και τα Look Up Tables αυξήθηκαν από 1789 στα 2021.



**Κύκλωμα 3:**

Οι αλλαγές που πρέπει να γίνουν ώστε για να έχουμε τις ίδιες εισόδους και εξόδους με το κύκλωμα 1 είναι οι εξής:

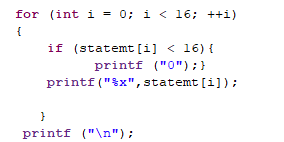
Καταρχάς θα κάνουμε όλες τις μεταβλητές εισόδου και εξόδου τύπου word.



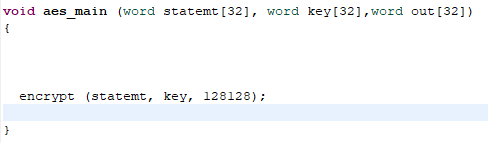
Όμως, επειδή έχουμε και μια άλλη μεταβλητή που λέγεται word , θα την ονομάσουμε σε worda και θα την κάνουμε και αυτή τύπο word.



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



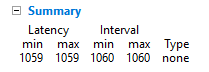
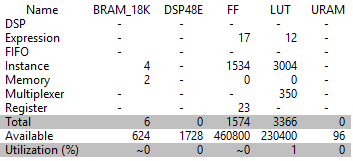
Επειδή βλέπουμε στον κώδικα ότι η aes\_main το μόνο που κάνει είναι να καλεί την encrypt.



Δεν χρειάζεται να υπάρχει η aes\_main καθώς θα βάλουμε το testbench να καλεί αμέσως την μέθοδο encrypt(), καθώς πλέον αυτήν, θα κάνουμε σύνθεση.

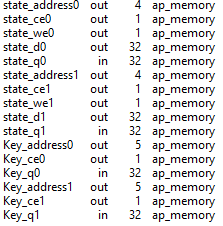
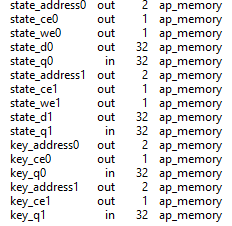
Για λόγους ομοιομορφίας, θα μετονομάσουμε το statemt σε state.

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

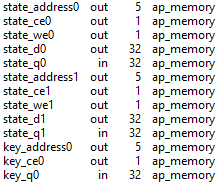


Μετά τις αλλαγές αυτές βλέπουμε τα ports που έχει φτιάξει για το κάθε κύκλωμα:

1ο κύκλωμα: 2ο κύκλωμα:



3ο κύκλωμα:



Η βασική διαφορά του 3ου με τα υπόλοιπα είναι ότι αποθηκεύει το key σε ένα δυσδιάστατο πίνακα και όχι σε ένα μονοδιάστατο όπως το κύκλωμα 1 και 2 και για αυτόν τον λόγο δεν φτιάχνει και άλλο port στο 3ο για να χωρέσει το κλειδί.

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

## 4.2 Σύγκριση των υλοποιήσεων

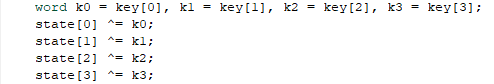
Όπως είδαμε πιο πάνω στο κεφάλαιο που εξηγούμε τον AES, ο αλγόριθμος αυτός ξεκινάει από την συνάρτηση που δημιουργεί τα νέα κλειδιά και τα προσθέτει στο αρχικό κείμενο

Θα αναλύσουμε πως οι συναρτήσεις αυτές έχουν υλοποιηθεί στα τρία κυκλώματα καθώς και την καθυστέρηση που επιβαρύνουν.

**1ο κύκλωμα:**

Σε αυτό το κύκλωμα δεν υλοποιηθεί καμία συνάρτηση η οποία φτιάχνει κλειδιά καθώς μετά τα προσθέτει στο plaintext. Όλα αυτά έχουν γίνει με εντολές και είναι οι παρακάτω:

**Πρόσθεση κλειδιού:**

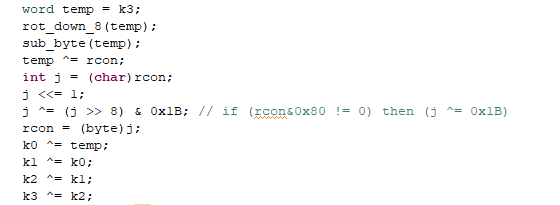


Το κλειδί και το state τα διαβάζει από τον είσοδο, και θα ξεκινήσουν στο 2ο κύκλο ρολογιού και θα κρατήσουν 2 κύκλους για την ολοκλήρωση καθώς διαβάζει και γράφει σε μνήμη και δεν υπάρχουν loops ώστε να περιμένει καθώς γίνεται παράλληλα η εκτέλεση των εντολών.

**Δημιουργία Κλειδιού:**

Για την δημιουργία κλειδιού χρησιμοποιείται κανονικά σύμφωνα με τον αλγόριθμο το Sbox και το Rcon.Ο τρόπος που γίνεται, είναι, να πάρουμε την τελευταία στήλη του cypherkey, να την κάνουμε rotate, να αντικαταστήσουμε τα bits, με τα bits του πίνακα S-Box καθώς μετά να την προσθέσουμε με την πρώτη στήλη του πίνακα Rcon και αυτό επαναλαμβάνεται μέχρι να φτιάξουμε όλο το νέο κλειδί.

Οι εντολές που γίνεται η δημιουργία κλειδιού είναι οι παρακάτω:

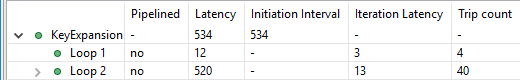


Από το κώδικα βλέπουμε, ότι έχουμε μνήμη rcon που διαβάζεται καθώς επιβαρύνει κατά 2 κύκλους το συνολικό κύκλωμα.

**2ο κύκλωμα:**

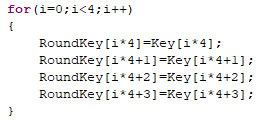
Σε αυτό το κύκλωμα, η συνάρτηση που δημιουργεί τα κλειδιά αποτελεί η **KeyExpansion()** και η συνάρτηση που αλλάζει τα κλειδιά, είναι η **AddRoundKey()**.

**KeyExpansion:**



To 1o loop επαναλαμβάνεται 4 φορές και η κάθε επανάληψη επιβαρύνει το κύκλωμα κατά 3 Latency.

Κώδικας:

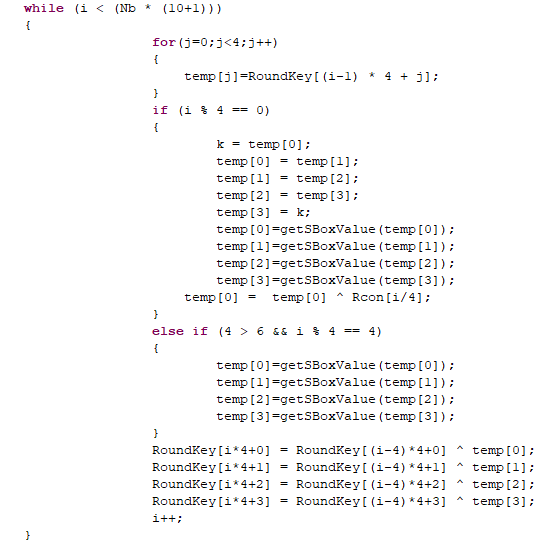


Στο παρόν κώδικα έχουμε 2 μνήμες που είναι το RoundKey που θα κάνει write και το Key που θα κάνει read.

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 ρολογιού.

To 2o loop επαναλαμβάνεται 40 φορές και η κάθε επανάληψη επιβαρύνει το κύκλωμα κατά 13 Latency.

**Κώδικας:**



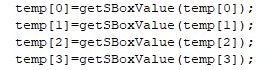
Στο παρόν κώδικα έχουμε 4 μνήμες που είναι το Key που θα κάνει write, το RoundKey που θα κάνει read και write, καθώς και τα sbox και Rcon που θα χρησιμοποιηθούν για read. Από το παραπάνω loop που αναλύσαμε, εύκολα καταλαβαίνουμε ότι θα υπάρχει μεγαλύτερη επιβάρυνση στο latency καθώς έχουμε περισσότερες μνήμες που διαβάζονται.

Στο 1ο loop βλέπουμε ότι κάνει μια πρόσθεση, ένα γινόμενο και μια σύγκριση, και αυτό το εργαλείο να έκανε σύνθεση ώστε να γίνουν όλα αυτά 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 το loop είναι δύο.

Στο επόμενο κύκλο ρολογιού ξεκινάει το 2ο loop το οποίο από τις πολλές μνήμες που εμπεριέχει που διαβάζονται θα επιβαρύνει σε σημαντικό βαθμό το κύκλωμα.

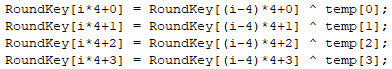
Στις παρακάτω εντολές χρειάζονται 2 κύκλους ρολογιού, έναν για καλέσει την συνάρτηση και έναν ώστε να λάβει την τιμή από αυτή με την χρήση του return.



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 και άρα θα ξεκινήσει μετά από δύο κύκλους.



Εκτός από αυτά, έχουμε και μνήμες που διαβάζονται όπως φαίνεται παρακάτω και οι οποίες καταναλώνουν δύο κύκλους.



Και ανήκουν σε loop 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.

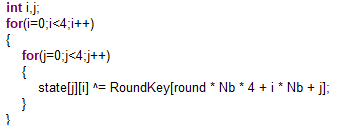
Και βλέπουμε από τον πίνακα ότι επιβαρύνει το κύκλωμα κατά 534 Latency.

**AddRoundKey:**



To 1o loop επαναλαμβάνεται 4 φορές και η κάθε επανάληψη επιβαρύνει το κύκλωμα κατά 10 Latency.

**Κώδικας**:



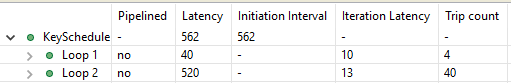
Στο παρόν κώδικα έχουμε 2 μνήμες που είναι το RoundKey που θα κάνει read και το state που θα κάνει write.

Από τον κώδικα καταλαβαίνουμε ότι, θα πρέπει να περιμένει ένα κύκλο ώστε να πάρει τα ορίσματα η συνάρτηση. Στην συνέχεια, δηλαδή στον επόμενο κύκλο θα ξεκινήσει το 1ο loop, καθώς και στον επόμενο δηλαδή στον 3ο κύκλο θα ξεκινήσει το εμφωλευμένο loop. 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.

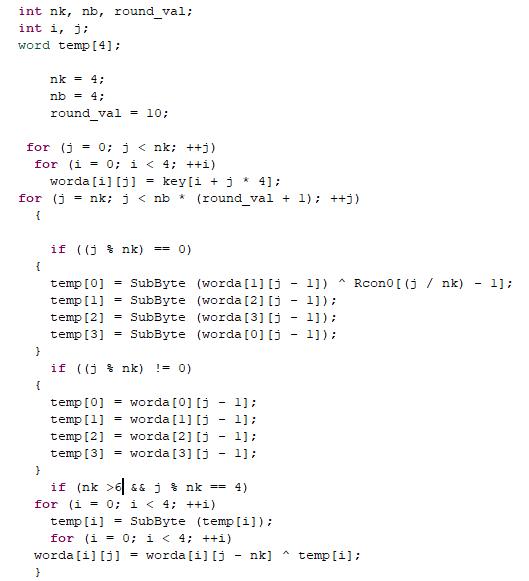
**3ο κύκλωμα:**

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

**KeySchedule:**



Το 1ο loop επαναλαμβάνεται 4 φορές και η κάθε επανάληψη επιβαρύνει το κύκλωμα κατά 10 Latency.Ενώ το δεύτερο, επαναλαμβάνεται 40 φορές ενώ επιβαρύνει το κύκλωμα κατά 13.



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

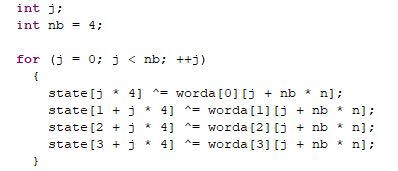
Εκτός από όλα αυτά βλέπουμε ότι στο εμφωλευμένο loop, έχουμε read στην μνήμη key ,με αποτέλεσμα να επιβαρύνει το κύκλωμα κατά δύο κύκλους. Στην συνέχεια στο 2ο εξωτερικό loop βλέπουμε ότι καλεί και την συνάρτηση **SubByte()** που και αυτή καταναλώνει 2 κύκλους, έναν για καλέσει την συνάρτηση και έναν 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:**



To 1o loop επαναλαμβάνεται 4 φορές και η κάθε επανάληψη επιβαρύνει το κύκλωμα κατά 4 Latency.

**Κώδικας:**



Ξεκινάει στο επόμενο κύκλο που καλείται, καθώς βλέπουμε ότι χρησιμοποιεί δύο μνήμες οι οποίες είναι η worda και η state, που την πρώτα την διαβάζει μόνο ενώ την δεύτερη την διαβάζει και γράφει σε αυτήν. Επειδή υπάρχει αλληλεξάρτηση μεταξύ των εντολών αυτών είναι φυσιολογικό να μην μπορούν να εκτελεσθούν οι εντολές παράλληλα πράγμα που δικαιολογεί το iteration latency που βγάζει.

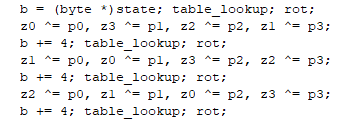
To latency που επιβαρύνουν οι παραπάνω συναρτήσεις που είδαμε φαίνονται στο παρακάτω πίνακα.

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

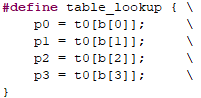
Η επόμενη συνάρτηση που θα μελετήσουμε είναι η **SubBytes()** που αυτό που κάνει είναι να αντικαθιστά τα στοιχεία του block με τα στοιχεία του S-box.

**1ο κύκλωμα:**

Σε αυτό το κύκλωμα η συνάρτηση η οποία αντικαθιστά το block με τα στοιχεία του πίνακα Sbox αποτελεί η table\_lookup όπως φαίνεται παρακάτω:



Η δομή της είναι η εξής:

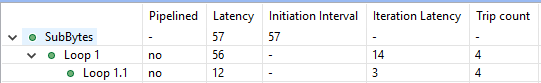


Αυτή η συνάρτηση από τι βλέπουμε διαβάζει από την μνήμη b[] καθώς την περιμένει η t0 , με αποτέλεσμα, όπως και στα παραπάνω κυκλώματα να υπάρχει μια καθυστέρηση κατά 2 κύκλους. Η διαφορά με εδώ είναι ότι δεν υπάρχει εξάρτηση, καθώς εκτελούνται όλα παράλληλα.

**2ο κύκλωμα:**

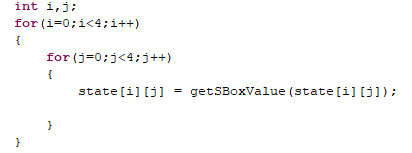
Σε αυτό το κύκλωμα, η συνάρτηση που δημιουργεί τα κλειδιά αποτελεί η **SubBytes().**

**SubBytes:**



To 1o loop επαναλαμβάνεται 4 φορές και η κάθε επανάληψη επιβαρύνει το κύκλωμα κατά 14 Latency.Επιπλέον, βλέπουμε ότι περιλαμβάνει και ένα εμφωλευμένο loop το οποίο εκτελείται 4 φορές καθώς επιβαρύνει το κύκλωμα κατά 3 latency. Άρα είναι φυσιολογικό να έχει τόσο latency όλο το loop.

**Κώδικας:**



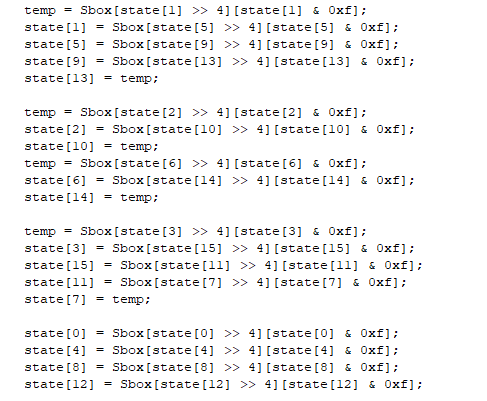
Βλέπουμε το εμφωλεύμενο loop θα αρχίσει ένα κύκλο μετά σε σύγκριση με το εξωτερικό. Στην συνέχεια, διαβάζεται από την μνήμη η state και να χρησιμοποιεί δύο κύκλους ρολογιού και επιπλέον, βλέπουμε ότι την χρησιμοποιεί σαν όρισμα στην συνάρτηση **getSBoxValue()** με αποτέλεσμα να χρειάζεται άλλους δύο κύκλους ρολογιού με αποτέλεσμα να καθυστερήσει και άλλο το κύκλωμα.

**3ο κύκλωμα:**

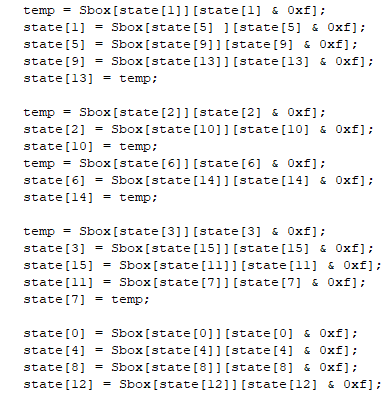
Σε αυτό το κύκλωμα, η συνάρτηση που δημιουργεί τα κλειδιά αποτελεί η **SubBytes\_ShiftRow().**Επιπλέον, αυτή η συνάρτηση περιλαμβάνει και μια άλλη διαδικασία που κάνει ο αλγόριθμος που είναι το **shiftRow()** αλλά εμείς θα προσπαθήσουμε να βρούμε τους χρόνους εκτέλεσης μόνο της **SubBytes()**.

Για να γίνει όμως αυτό θα πρέπει να κάνουμε κάποιες τροποποιήσεις του κώδικα καθώς θα βγάλουμε την ολίσθηση.

**Αρχικός κώδικας:**



**Τροποποιημένος κώδικας:**



Παρατηρούμε ότι ο παραπάνω κώδικας περιλαμβάνει την μνήμη state την οποία, την διαβάζει και γράφει σε αυτήν, καθώς περιέχει και τον πίνακα Sbox που τον διαβάζει από την μνήμη. Άρα, διαβάζει την state η οποία διαρκεί δύο κύκλους ρολογιού και στον επόμενο, διαβάζει την Sbox δηλαδή η Sbox περιμένει να διαβαστεί η state για να μπορέσει να διαβαστεί και αυτή. Όλη αυτή η εξάρτηση επιβαρύνει πολύ το κύκλωμα.

Latency τροποποιημένης συνάρτησης:



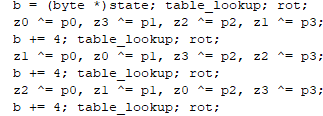
To latency που επιβαρύνουν οι παραπάνω συναρτήσεις που είδαμε φαίνονται στο παρακάτω πίνακα.

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

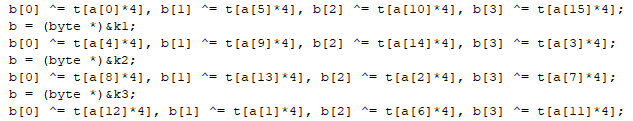
Στην συνέχεια, θα αναλύσουμε τις επόμενες δύο μεθόδους του αλγορίθμου που έμειναν, που είναι η ολίσθηση και η ανάμιξη πινάκων.

**1ο κύκλωμα:**

Η **ShiftRows()** εκτελείται από την συνάρτηση rot καθώς εκτελείται παράλληλα με τις άλλες συναρτήσεις ,όπως φαίνεται παρακάτω



Ενώ η **MixColumns()** γίνεται με μια πράξη xor παράλληλα και αυτή όπως φαίνεται παρακάτω:



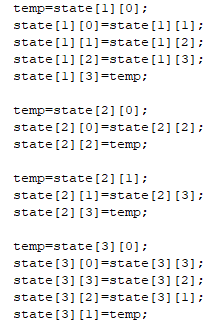
Με αποτέλεσμα να γλιτώνουμε καθυστέρηση η οποία θα είχε προκύψει.

**2ο κύκλωμα:**

Σε αυτό το κύκλωμα, υλοποιούνται και οι δύο μέθοδοι και είναι οι παρακάτω:

**ShiftRows**:

**Κώδικας**:



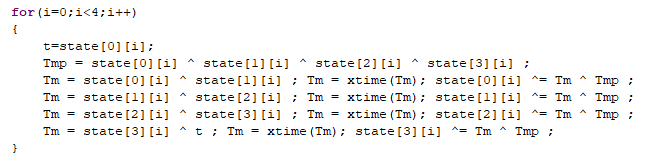
Βλέπουμε από τον κώδικα, διαβάζει το state που είναι αποθηκευμένο σε μνήμη, με αποτέλεσμα να δημιουργεί καθυστέρηση αφού χρειάζονται δύο κύκλοι για να ολοκληρωθεί το read καθώς να εκτελεστεί η επόμενη εντολή. Επιπλέον, παρατηρούμε ότι υπάρχει εξάρτηση μεταξύ των εντολών με αποτέλεσμα να επιβαρύνει και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

To Latency φαίνεται παρακάτω:



**MixColumns**:

**Κώδικας**:



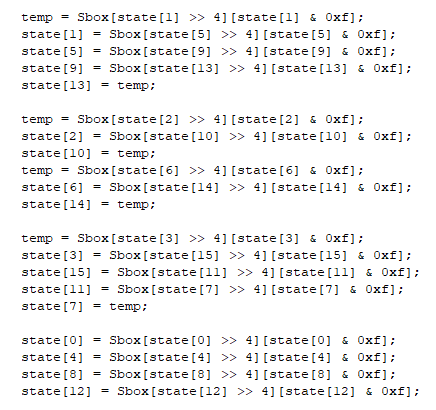
Ομοίως, με την προηγούμενη συνάρτηση, διαβάζει από την μνήμη τον πίνακα state και επειδή υπάρχει μεταξύ των εντολών εξάρτηση, επιβαρύνεται και άλλο το κύκλωμα.

Latency:



**3ο κύκλωμα:**

Η shift\_Rows σε αυτόν το κύκλωμα εκτελείται παράλληλα με **ByteSub()**, εφόσον ανήκουν στην ίδια συνάρτηση **ByteSub\_ShiftRow()**. Και από τον προηγούμενο κώδικα βλέπουμε ότι δεν επηρεάζει την απόδοση του κυκλώματος, εφόσον το μόνο που κάνει είναι μια ολίσθηση όπως φαίνεται στο παρακάτω κώδικα:



Σε σύγκριση η Mix\_Column υλοποιείται στην συνάρτηση **Mix\_Column\_AddRoundKey()**.

Θα χρειαστεί να κάνουμε κάποιες τροποποιήσεις για να βρούμε το latency που επιβαρύνει μόνο το Mix\_Column.

**Αρχικός κώδικας:**

**int** ret[8 \* 4], j;

**register** **int** x;

**int** nb=4;

**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];

}

Από τι βλέπουμε από τον κώδικα, έχουμε έναν βρόγχο, στον οποίο διαβάζεται ο πίνακας state από την μνήμη. Επιπλέον, βλέπουμε ότι το ίδιο γίνεται με τον πίνακα worda. Μια τεράστια διαφορά με το κύκλωμα 2 είναι ότι δεν υπάρχει εξάρτηση μεταξύ των εντολών και με αυτόν τον τρόπο μπορούν να εκτελεστούν στο ίδιο κύκλο ρολογιού.

To latency που επιβαρύνουν οι παραπάνω συναρτήσεις που είδαμε φαίνονται στο παρακάτω πίνακα.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1ο Κύκλωμα | 2ο κύκλωμα | 3ο κύκλωμα |
| 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 είχε σημαντικό ρόλο στην απόδοση των κυκλωμάτων, τα memory read, η ύπαρξη βρόγχων καθώς και οι εξαρτήσεις που υπάρχουν εφόσον, ανάμεσα στις εντολές.

Μέχρι στιγμής είδαμε τις συναρτήσεις που καλούνται. Στην συνέχεια, θα δούμε και τις κύριες συναρτήσεις που επιλέξαμε για σύνθεση.

**1ο κύκλωμα:**

**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;

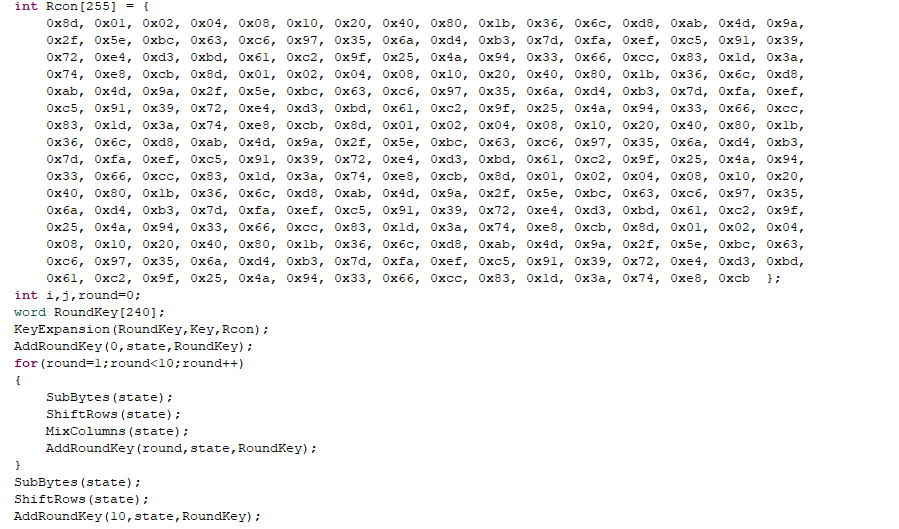
state[2] = k2;

state[3] = k3;

Αυτό που αξίζει να σημειώσουμε για αυτόν τον κώδικα είναι ο αριθμός των επαναλήψεων που κάνει ο βρόγχος, καθώς όλα τα υπόλοιπα είναι γράφτηκαν για υλοποιήσουν τον αλγόριθμο AES.

**2ο κύκλωμα:**

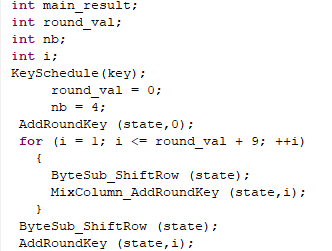
**Κώδικας:**



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.

**3ο κύκλωμα:**

**Κώδικας:**



Σύμφωνα με τον παραπάνω κώδικα, βλέπουμε κάποιες μεταβλητές που στην συνέχεια, θα αποθηκευτούν 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:**

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1ο κύκλωμα | 2ο κύκλωμα | 3ο κύκλωμα |
| Latency | 51 | 1862 | 1059 |

Από τους παραπάνω κώδικες, βλέπουμε ότι ο βασικότερος λόγος που επιδρά στην απόδοση των παρακάτω κυκλωμάτων είναι, η αλληλεξάρτηση που έχουν μεταξύ τους οι εντολές, δηλαδή για το πώς μπορούν να εκτελεσθούν παράλληλα, όπως φαίνεται στο κύκλωμα 1. Στα άλλα δύο κυκλώματα υπάρχουν εξαρτήσεις μεταξύ των εντολών, με αποτέλεσμα να μην εντολή χρειάζεται δύο κύκλους ρολογιού, θα πρέπει να περιμένει η επόμενη εντολή μέχρι να ολοκληρωθεί για να μπορέσει να λάβει την μεταβλητή που χρειάζεται.

Επιπλέον, οι προσπελάσεις στην μνήμη επιδρούν αρνητικά στην απόδοση του κυκλώματος όπως φαίνεται πιο πολύ στο κύκλωμα 2 και 3. Η προσπέλαση μνήμης, δηλαδή να διαβάζει το κύκλωμα από την μνήμη , χρειάζεται δύο κύκλους με αποτέλεσμα να καθυστέρει 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. Η καθυστέρηση αυτή, γίνεται διπλή όταν υπάρχουν και εμφωλευμένοι βρόγχοι όπως στο κύκλωμα 2 και 3.

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

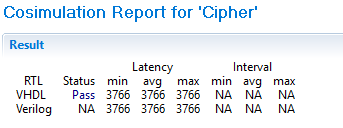
Συνεπώς, κλειδί για υψηλή απόδοση όπως φαίνεται από τα παραπάνω είναι οι λιγότερες προσπελάσεις μνήμη(read) , η ελαχιστοποίηση των βρόγχων καθώς η μη-εξάρτηση των εντολών αν έχουμε εντολές που χρειάζονται παραπάνω από έναν κύκλο.

***Κεφάλαιο 5ο***

# 5.Βελτιστοποίηση κυκλώματος

Σε αυτό το κεφάλαιο, θα ασχοληθούμε με την βελτιστοποίηση του πιο αργού κυκλώματος που είναι το 2ο κύκλωμα. Σε αυτό το κύκλωμα έχουμε κάνει διάφορες αλλαγές πρώτον για γίνει συνθέσιμο και δεύτερον να μοιάζει πιο πολύ με το πρώτο κύκλωμα που είναι σε απόδοση το καλύτερο. Όμως, θα βελτιστοποιήσουμε την 1η έκδοση δηλαδή την έκδοση που ήταν συνθέσιμη από την αρχή.

Αρχικό Latency:



**Μεθοδολογία**:

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

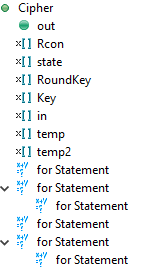
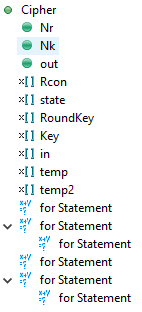
|  |  |  |
| --- | --- | --- |
| **α/α** | **Μέθοδος** | **Όρισμα** |
| **1** | Απαλοιφή μεταβλητών με σταθερές | Nr,Nk |
| **2** | Μεταφορά δεδομένων | Key,In,state |
| **3** | Ενσωμάτωση συναρτήσεων | SubBytes,AddRoundKey,MixColumns |
| **4** | Unroll | KeyExpansion(For),AddRoundKey(For) |
| **5**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

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



Άρα περιμένει να λάβει αυτά τα ορίσματα, το οποίο δημιουργεί καθυστέρηση στο ολικό 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 και οι οποίες θα είναι Nr=10 και Nk=4.Στην συνέχεια, θα τις αντικαταστήσουμε στο κώδικα τις παραπάνω μεταβλητές, με τις τιμές αυτές.

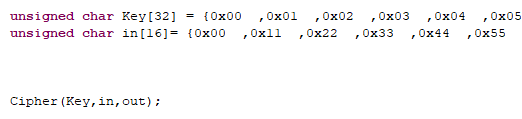
Πριν τις αλλαγές: Μετά τις αλλαγές:



Μετά την αλλαγή αυτή, το latency έφθασε να είναι 1976.

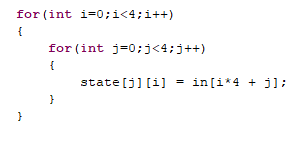
2.Έπειτα, θα μεταφέρουμε όσο πιο πολύ κώδικα από την συνάρτηση που θα γίνει σύνθεση, στο testbench.

Θα μεταφέρουμε το κείμενο, και το κλειδί στο testbench, καθώς θα τα βάλουμε ως όρισμα στην κύρια συνάρτηση ώστε να μπορέσει να τα χρησιμοποιήσει.



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

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



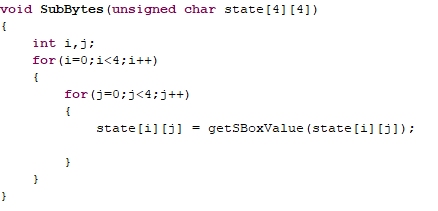
Καθώς, πρέπει να μπει ως όρισμα μέσα στην κύρια συνάρτηση:



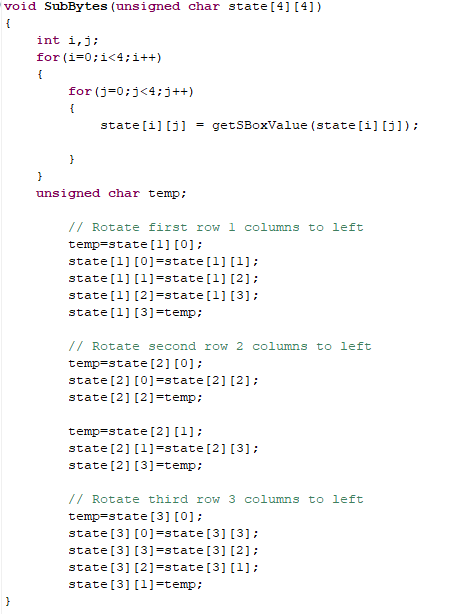
Και τέλος πρέπει να κάνουμε printf το state για να βγάλει τα αποτελέσματα. Μετά από αυτές τις αλλαγές, το latency έφθασε 1862.

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

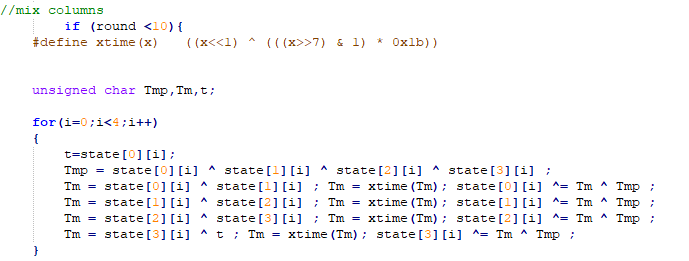
**Αρχικός κώδικας subBytes():**



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



Το ίδιο πράγμα θα κάνουμε 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 θα γίνει ο κώδικας:



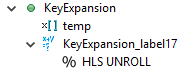
Ενώ για την AddRoundKey το μόνο πράγμα που χρείαζεται να βάλουμε να παίρνει για ορίσματα το RoundKey και το Round η συνάρτηση **SubBytes().**



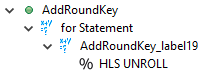
Το Latency διαμορφώθηκε στα 1696.

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

Αρχικά,θα βάλουμε unroll 1ο βρόγχο for της συνάρτησης **KeyExpansion()**

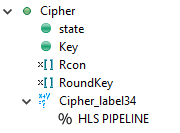


Στην συνέχεια, θα βάλουμε unroll στο εμφωλευμένο βρόγχο for της συνάρτησης **AddRoundKey()**.

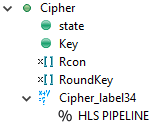


To Latency διαμορφώθηκε στα 1667.

5.θα βάλουμε pipeline στο βρόγχο while της συνάρτησης **KeyExpansion()** .



θα βάλουμε pipeline στο βρόγχο for της συνάρτησης **Cipher()** .



Και προκύπτει latency 509.

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

Αυτό κάναμε στο state:

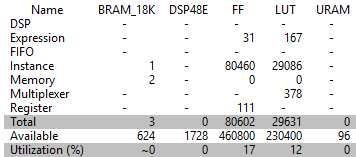


Και το latency έφθασε στα 365

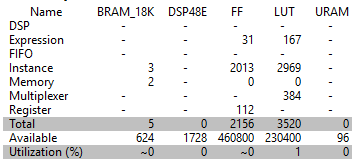
6.Χρησιμοποιώντας το Directive, ARRAY\_PARTITION που χωρίζει τον πίνακα σε μικρότερους καθώς δεσμεύει κιάλλους καταχωρητές, στον πίνακα Sbox και με αυτόν τον τρόπο δουλεύει αποτελεσματικότερα η μέθοδος της διοχέτευσης[16]. Τον χωρίζουμε σε δύο πίνακες καθώς με την χρήση αυτου του Directive μειώσαμε το Latency στα 349.



Καθώς οι πόροι που δεσμεύονται είναι οι παρακάτω:

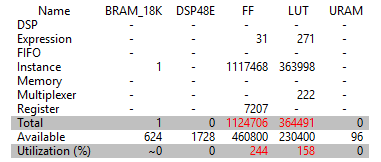


Ενώ χωρίς την χρήση αυτού του Directive, οι πόροι που δεσμεύαμε ήταν:



Βλέπουμε, ότι μειώθηκαν τα ΒRAM κατά 2 ενώ τα FF αυξήθκαν κατά 3.638% καθώς και τα LUT κατά 741%.Καταλαβαίνουμε από τα ποσοστά, πόσο σημαντική είναι δέσμευση των πόρων με την χρήση αυτού του Directive.

Χρησιμοποιώντας άλλη μια φορά το Directive ARRAY\_PARTITION στον πίνακα RoundKey, μείωσαμε κίαλλο το latency 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 πέσει κιάλλο το latency αλλά θα συνεχίζει να ζητάει περισσότερους πόρους.

Στον παρακάτω πίνακα βλέπουμε τις πόρους που δεσμεύει από όλες τις τροποποιήσεις που κάναμε στο κύκλωμα καθώς συμπεραίνουμε ότι τεράστια αλλαγή στην δέσμευση πόρων κάνει το Directive ΑRRAY\_PARTITION.Ενώ οι άλλοι μέθοδοι που είναι οι αλλαγές του κώδικα καθώς και η χρήση Directive Pipeline, Unroll καθώς ARRAY\_RESHAPE επηρρεάζουν ελάχιστα του 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 |

Αρχικό Latency: 3766 Τελικό Latency:349

Με βάση τα παραπάνω μπορέσαμε να μειώσουμε το latency του κυκλώματος κατά 90.73% αλλα επιπλέον, μείωσαμε τα BRAM 50% καθώς αυξήσαμε κατά 2854% και τα LUT κατά 869%.Αν μέναμε όμως στην 6η μέθοδο τότε θα είχαμε μείωση κατά 26% τα FF επιπλέον, 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 ανάλυση του εργαλείου HLS, με την χρήση τριών διαφορετικών υλοποιήσεων του αλγορίθμου κρυπτογράφησης AES.Στην αρχή, εφόσον πήραμε τρείς διαφορετικές υλοποιήσεις που είναι γραμμένοι σε γλώσσα υψηλού επιπέδου C, κάναμε διάφορες τροποποιήσεις ώστε να γίνουν συνθέσιμοι από το εργαλείο HLS. Εφόσον, κάναμε διάφορες αλλαγές ώστε να έχουν ίδιες πύλες εισόδους και εξόδου και καθώς αναλύσαμε την σύνθεση τους, μπορέσαμε να καταλήξουμε στο συμπέρασμα ότι κλειδί για την σύνθεση για 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 διαφόρων 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 επιταχύναμε συνολικά την απόδοση του κατά 90.73%.Συνεπώς, ακόμη σε έναν πολύπλοκο αλγόριθμο κρυπτογράφησης, μπορέσαμε να έχουμε κυκλώματα με μια ικανοποιητική απόδοση, πράγμα που είναι απάντηση σε ένα ερώτημα που ήταν ανοιχτό εδώ και χρόνια. Ένα ερώτημα που παραμένει ανοιχτό, είναι όμως, αν υπάρχει τρόπος σύνθεσης ενός κώδικα γραμμένο σε γλώσσα υψηλού επιπέδου, σε γλώσσα HDL, ο οποίος να είναι εύκολα αναγνώσιμος και κατανοητός από τους προγραμματιστές και επιπλέον πώς θα μπορούσαμε, να έχουμε κυκλώματα με υψηλή απόδοση με την χρήση όσων το πιο δυνατών 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