**Σύγκριση κυκλωμάτων**

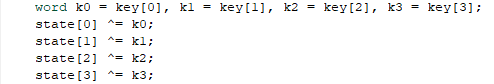
Όπως είδαμε πιο πάνω στο κεφάλαιο που εξηγούμε τον AES, ο αλγόριθμος αυτός ξεκινάει από την συνάρτηση που δημιουργεί τα νέα κλειδιά και τα προσθέτει στο αρχικό κείμενο

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:**

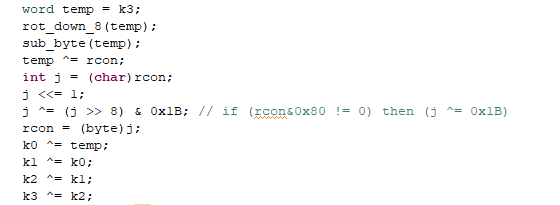


Το κλειδί και το state τα διαβάζει από τον είσοδο, και θα ξεκινήσουν στο 2ο κύκλο ρολογιού και θα κρατήσουν 2 κύκλους για την ολοκλήρωση καθώς διαβάζει και γράφει σε μνήμη και δεν υπάρχουν loops ώστε να περιμένει καθώς γίνεται παράλληλα η εκτέλεση των εντολών.

**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 επαλαναλαμβάνεται μέχρι να φτιάξουμε όλο το νέο κλειδί.

Οι εντολές που γίνεται η δημιουργία κλειδιού είναι οι παρακάτω:

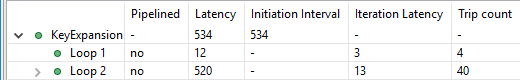


Από το κώδικα βλέπουμε, ότι έχουμε μνήμη rcon που διαβάζεται καθώς επιβαρύνει κατά 2 κύκλους το συνολικό κύκλωμα.

**2ο κύκλωμα:**

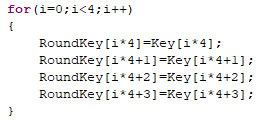
Σε αυτό το κύκλωμα, η συνάρτηση που δημιουργεί τα κλειδιά αποτελεί η KeyExpansion(); και η συνάρτηση που αλλάζει τα κλειδιά, είναι η AddRoundKey();

**KeyExpansion:**



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

Κώδικας:

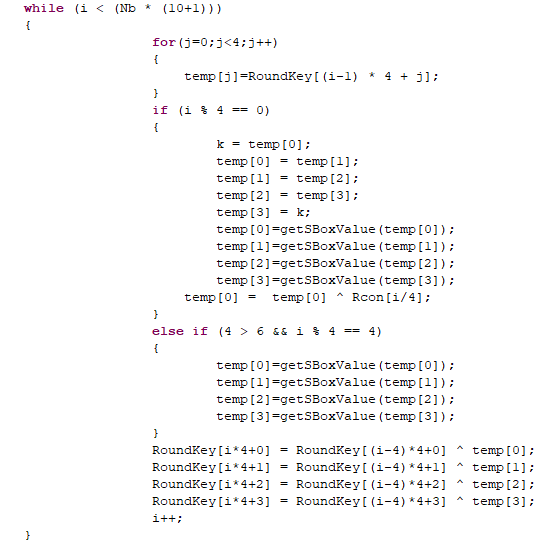


Στο παρόν κώδικα έχουμε 2 μνήμες που είναι το RoundKey που θα κάνει write και το Key που θα κάνει read.

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:**



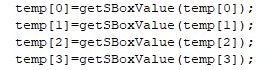
Στο παρόν κώδικα έχουμε 4 μνήμες που είναι το Key που θα κάνει write , το RoundKey που θα κάνει read και write, καθώς και τα sbox και Rcon που θα χρησιμοποιηθούν για read. Από το παραπάνω loop που αναλύσαμε, εύκολα καταλαβαίνουμε ότι θα υπάρχει μεγαλύτερη επιβάρυνση στο latency καθώς έχουμε περισσότερες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

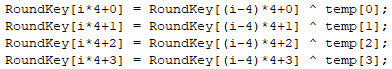
Στις παρακάτω εντολές χρειάζονται 2 κύκλους ρολογιού, έναν για καλέσει την συνάρτηση και έναν ώστε να λάβει την τιμή από αυτή με την χρήση του return.



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.

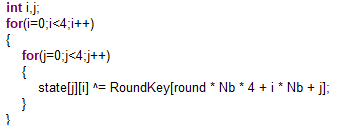
Και βλέπουμε από τον πίνακα ότι επιβαρύνει το κύκλωμα κατά 534 Latency

**AddRoundKey:**



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

**Κώδικας**:



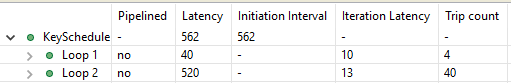
Στο παρόν κώδικα έχουμε 2 μνήμες που είναι το RoundKey που θα κάνει read και το state που θα κάνει write.

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:**

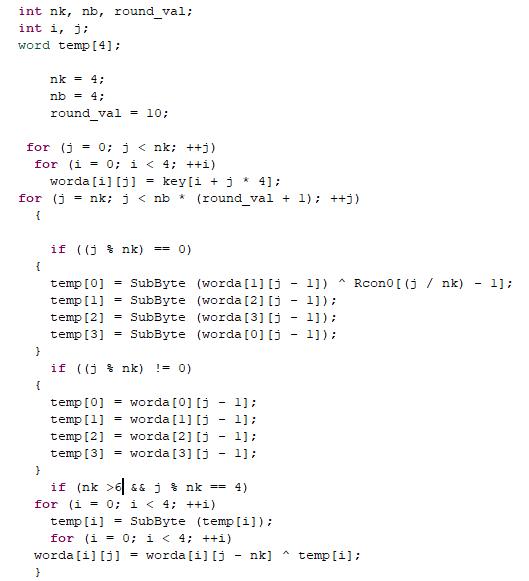
Σε αυτό το κύκλωμα, η συνάρτηση που δημιουργεί τα κλειδιά αποτελεί η KeySchedule(); και η συνάρτηση που αλλάζει τα κλειδιά, είναι η AddRoundKey();

**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.

**Κώδικας:**



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.

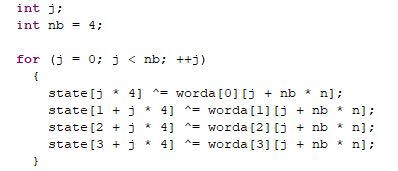
Εκτός από όλα αυτά βλέπουμε ότι στο εμφωλευμένο loop, έχουμε read στην μνήμη key ,με αποτέλεσμα να επιβαρύνει το κύκλωμα κατά δύο κύκλους. Στην συνέχεια στο 2ο εξωτερικό loop βλέπουμε ότι καλεί και την συνάρτηση SubByte που και αυτή καταναλώνει 2 κύκλους, έναν για καλέσει την συνάρτηση και έναν για επιστρέψει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

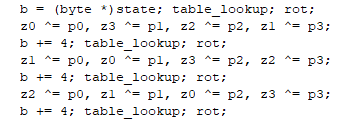
|  |  |  |  |
| --- | --- | --- | --- |
|  | 1ο Κύκλωμα | 2ο κύκλωμα | 3ο κύκλωμα |
| Πρόσθεση Κλειδιού | ? | 41 | 17 |
| Δημιουργία Κλειδιών | ? | 534 | 562 |
| Συνολικό | ? | 575 | 589 |

Το «?» έχει μπει, γιατί το εργαλείο δεν μπόρεσε να υπολογίσει τους κύκλους, καθώς οι συναρτήσεις δεν καλούνται.

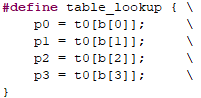
Η επόμενη συνάρτηση που θα μελετήσουμε είναι η SubBytes που αυτό που κάνει είναι να αντικαθιστά τα στοιχεία του block με τα στοιχεία του S-box.

**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:

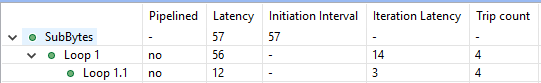


Αυτή η συνάρτηση από τι βλέπουμε διαβάζει από την μνήμη b[] καθώς την περιμένει η t0 , με αποτέλεσμα, όπως και στα παραπάνω κυκλώματα να υπάρχει μια καθυστέρηση κατά 2 κύκλους. Η διαφορά με εδώ είναι ότι δεν υπάρχει εξάρτηση, καθώς εκτελούνται όλα παράλληλα.

**2ο κύκλωμα:**

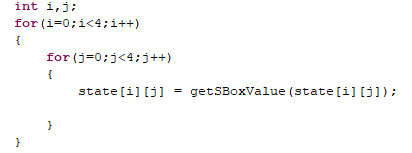
Σε αυτό το κύκλωμα, η συνάρτηση που δημιουργεί τα κλειδιά αποτελεί η SubBytes();.

**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:**



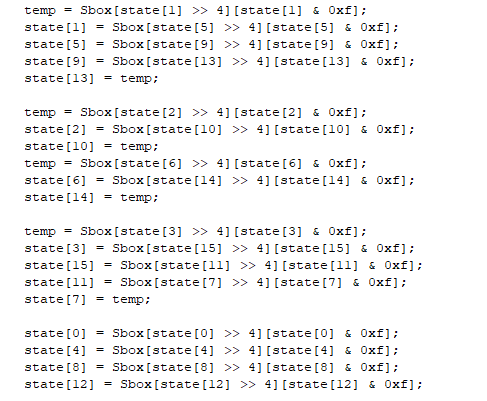
Βλέπουμε το εμφωλεύμενο loop θα αρχίσει ένα κύκλο μετά σε σύγκριση με το εξωτερικό. Στην συνέχεια, διαβάζεται από την μνήμη η state και να χρησιμοποιεί δύο κύκλους ρολογιού και επιπλέον, βλέπουμε ότι την χρησιμοποιεί σαν όρισμα στην συνάρτηση getSBoxValue με αποτέλεσμα να χρειάζεται άλλους δύο κύκλους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

**3ο κύκλωμα:**

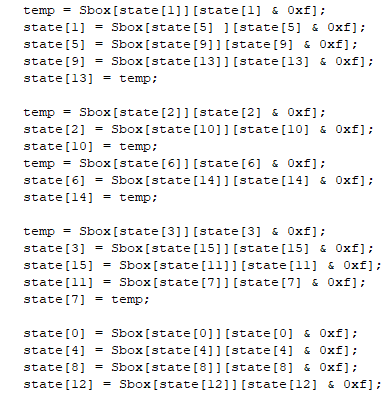
Σε αυτό το κύκλωμα, η συνάρτηση που δημιουργεί τα κλειδιά αποτελεί η SubBytes\_ShiftRow();.Επιπλέον, αυτή η συνάρτηση περιλαμβάνει και μια άλλη διαδικασία που κάνει ο αλγόριθμος που είναι το shiftRow αλλά εμείς θα προσπαθήσουμε να βρούμε τους χρόνους εκτέλεσης μόνο της SubBytes.

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:



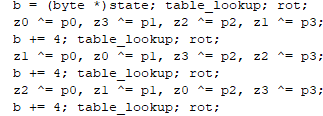
To latency που επιβαρύνουν οι παραπάνω συναρτήσεις που είδαμε φαίνονται στο παρακάτω πίνακα.

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

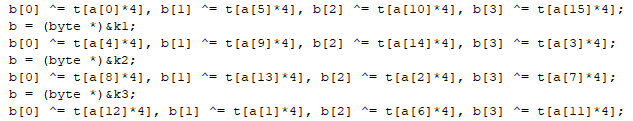
Στην συνέχεια, θα αναλύσουμε τις επόμενες δύο μεθόδους του αλγορίθμου που έμειναν, που είναι η ολίσθηση και η ανάμιξη πινάκων.

**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:



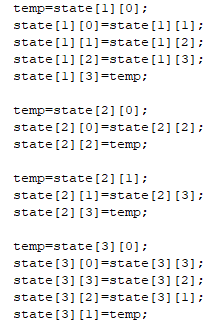
Με αποτέλεσμα να γλιτώνουμε καθυστέρηση η οποία θα είχε προκύψει.

**2ο κύκλωμα:**

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

**ShiftRows**:

**Κώδικας**:



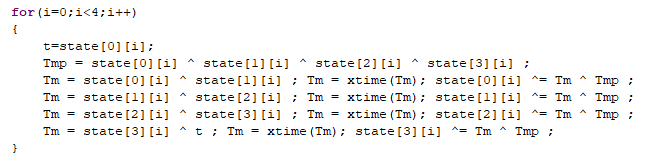
Βλέπουμε από τον κώδικα, διαβάζει το state που είναι αποθηκευμένο σε μνήμη, με αποτέλεσμα να δημιουργεί καθυστέρηση αφού χρειάζονται δύο κύκλοι για να ολοκληρωθεί το read καθώς να εκτελεστεί η επόμενη εντολή. Επιπλέον, παρατηρούμε ότι υπάρχει εξάρτηση μεταξύ των εντολών με αποτέλεσμα να επιβαρύνει και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

To Latency φαίνεται παρακάτω:



**MixColumns**:

**Κώδικας**:



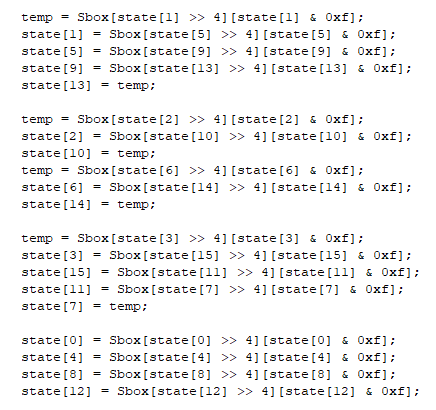
Ομοίως, με την προηγούμενη συνάρτηση, διαβάζει από την μνήμη τον πίνακα state και επειδή υπάρχει μεταξύ των εντολών εξάρτηση, επιβαρύνεται και άλλο το κύκλωμα.

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 | ? | 41 | 17 |
| Δημιουργία Κλειδιών | ? | 534 | 562 |
| Αντικατάσταση με Πίνακα | ? | 57 | 15 |
| Ολίσθηση | ? | 11 | 0 |
| Ανάμιξη Πινάκων | ? | 17 | 1 |
| Συνολικό | ? | 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;

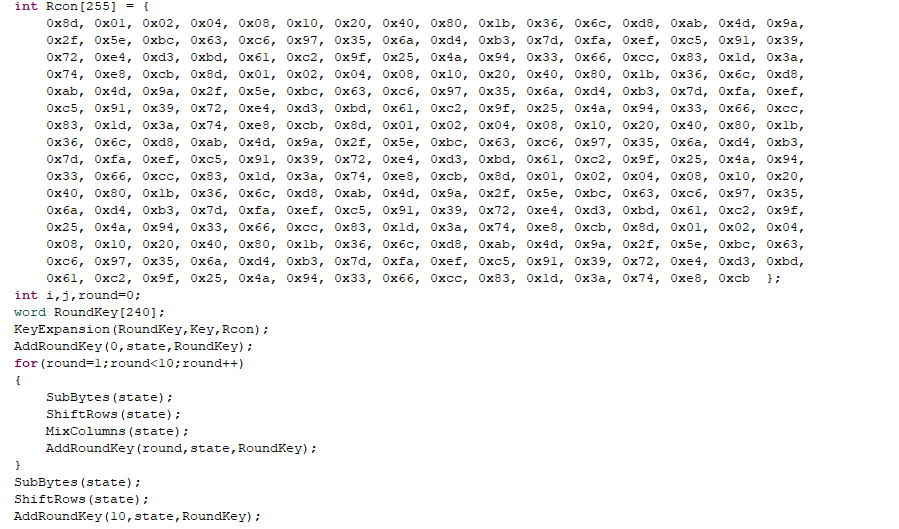
state[2] = k2;

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:**

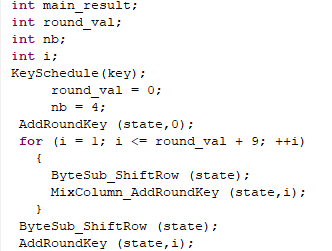
**Κώδικας:**



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, 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.