

Rapport de Projet Conception d'un Système Numérique

Implémentation du chiffrement
AES en langage VHDL



Un projet réalisé et ici relaté par Tristan Lorriaux

Groupe IV - Enseignant : M. J-M Dutertre

Le 27/12/2020

ÉCOLE DES MINES DE SAINT-ÉTIENNE
158, cours Fauriel - CS 62362
42023 Saint-Étienne Cedex 2 - France
Tél. +33 (0)4 77 42 01 23
Fax +33 (0)4 77 42 00 00

Campus Georges Charpak - Provence
880, route de Mimet
13541 Gardanne - France
Tél. +33 (0)4 42 61 66 00
Fax +33 (0)4 42 61 66 04
www.mines-stetienne.fr

TABLE DES MATIÈRES

Introduction	4
Origine, fonctionnement & attaques	4
Bref historique	4
Fonctionnement	4
Attaques	4
Description structurelle de l'implémentation	6
Environnement de travail	8
Environnement	8
Compilation	8
Bibliothèques	9
Résultats fonctionnels observés	10
La table de substitution S-Box et la fonction SubBytes	10
La fonction ShiftRows	13
La fonction MixColumns	15
La fonction AddRoundKey	19
Le bloc de génération des clés de ronde	20
Implémentation de l'AESROUND	25
Description	25
Tests	26
Composants annexes	27
Le compteur	27
Implémentation finale de l'AES	28
Machine à états finis de Moore pour la gestion du procédé global	28
Assemblage final des blocs	30
Testbenchs et résultats finaux	31
Problèmes observés et pistes d'amélioration	32
Problèmes rencontrés	32
Voies d'amélioration	32
Conclusion	33
Annexes	34
Annexe 1 :	34
Annexe 2 :	34
Annexe 3 :	34

TABLE DES FIGURES

Figure 1 - Composant AES-128 : Entrées/Sorties	6
Figure 2 - Description graphique des rounds de l'AES	7
Figure 3 - Chronogramme du protocole	7
Figure 4 - Un environnement Xubuntu 20.04 LTS Focal Fossa	8
Figure 5 - En tête type d'un fichier du projet	9
Figure 6 - Principe de la fonction SubBytes	10
Figure 7 - Table de substitution utilisée pour le projet	10
Figure 8 - Conversion du signal pour une sortie de type bit8 de SBox	11
Figure 9 - Résultats du testbench de la SBox	11
Figure 10 - Architecture de SubBytes	11
Figure 11 - Résultats du testbench de SubBytes	12
Figure 12 - Illustration de la fonction ShiftRows	13
Figure 13 - Architecture de ShiftRows	13
Figure 14 - Résultats du testbench de ShiftRows	14
Figure 15 - Produit matriciel de la fonction MixColumns	15
Figure 16 - Application de la fonction MixColumns sur les colonnes de l'état courant	15
Figure 17 - Architecture de MixColumns_Elem	16
Figure 18 - Architecture de MixColumns	17
Figure 19 - Résultat du testbench de MixColumns_Elem	18
Figure 20 - Résultat du testbench de MixColumns	18
Figure 21 - Calcul de la fonction AddRoundKey	19
Figure 22 - Architecture de AddRoundKey	19
Figure 23 - Résultats du testbench de AddRoundKey	19
Figure 24 - Calcul de la première colonne ($i \bmod N_k = 0$) à l'aide de la matrice Rcon	20
Figure 25 - Architecture de KeyExpansion	21
Figure 26 - Fonctionnement détaillé de KeyExpansion	21
Figure 27 - Schéma de principe de la machine à états finis (KeyExpansion_FSM)	22
Figure 28 - Architecture de KeyExpansion_I_O	23
Figure 29 - Fonctionnement et sous-blocs de KeyExpansion_I_O	24
Figure 30 - Résultats du testbench de KeyExpansion_I_O	24
Figure 31 - Schéma des entrées/sorties de l'AESRound	25
Figure 32 - Schéma de principe de l'AESRound	26
Figure 33 - Résultats du testbench de l'AESRound	26
Figure 34 - Schéma des entrées sorties du compteur	27
Figure 35 - Résultats du testbench du compteur	27
Figure 36 - Principe de fonctionnement de la FSM globale	28
Figure 37 - Schéma de principe du protocole AES	30
Figure 38 - Résultats du testbench de l'AES	31

INTRODUCTION

Origine, fonctionnement & attaques

Bref historique

Advanced Encryption Standard ou AES est un algorithme de chiffrement symétrique, produit d'un appel à candidatures international lancé par le National Institute of Standards and Technology en 1997. Successeur du DES, devenu obsolète, il est aujourd'hui beaucoup utilisé dans de nombreux protocoles car simple d'application (id est d'une complexité moindre) et peu gourmand en mémoire. Il est par ailleurs garant d'une forme de sécurité en 2020 car relativement dur à attaquer (approuvé notamment par la NSA).

Fonctionnement

L'algorithme prend basiquement un mot en entrée d'une taille de 128 bits, et une clé de taille 128, 192 ou 256 bits. Il repose sur plusieurs simples protocoles, qui mis bout à bout et bouclés en fond un protocole global de chiffrement sécurisé. Le premier bloc consiste à échanger les 16 octets d'entrée selon une table correspondante (« substitution box »). Puis, formant une matrice carrée de dimension 4 qui se voit « mélangée » selon un pas de décalage relatif à la ligne de la matrice. Puis on applique une transformation linéaire à cette matrice (multiplication de chaque octet par des polynômes d'une matrice annexe). On applique finalement un XOR entre la matrice et une autre matrice (qui hérite de la clé de départ). Ces opérations sont bouclées (dans notre cas, 10 rounds pour une clé de 128 bits).

Le texte chiffré (« cypher ») obtenu l'est à partir de la clef de départ mais aussi de plusieurs sous clés héritant de la clé primaire : peu de corrélation donc entre la donnée de sortie et la clé primaire.

Attaques

L'AES n'est aujourd'hui pas totalement invulnérable même si le protocole n'est pas dans ses fondements remis en question. Les meilleures attaques observées restent aujourd'hui des attaques par force brute. On peut souligner cependant qu'en 2011, les équipes de recherche de Microsoft ont publié un rapport d'attaque sur une version 128 bits 10 rounds de l'AES évaluée à $2^{126,1}$ opérations contre 2^{128} pour une attaque par force brute, aujourd'hui impraticable.

Nous implémenterons donc le protocole AES en VHDL, un langage de description matérielle. L'intérêt est donc de découvrir par exemple comment concevoir un support hardware par exemple au protocole AES, mais aussi de comprendre comment le protocole fonctionne en profondeur (protocole qui nous le rappelons est peu gourmand en puissance donc parfait pour des systèmes embarqués : transmissions sans fil, I2C cryptées etc.). Enfin, l'exercice nous permet évidemment de développer nos compétences de programmation en langage VHDL.

DESCRIPTION STRUCTURELLE DE L'IMPLEMENTATION

L'implémentation concerne le chiffrement des données. On a donc besoin d'une donnée d'entrée, d'un signal de pour lancer le cryptage, d'une clé, d'une horloge et d'un signal de reset du protocole en entrée, et en sortie le « cypher » et un signal indiquant si le protocole est en marche. En résumant, on a donc :

- Une entrée 'data_i' de 128 bits
- Une horloge 'clock_i'
- Un signal d'initialisation 'reset_i' actif à l'état haut
- Un signal 'start_i' indiquant la présence d'un message à chiffrer en entrée
- Une sortie 'data_o' sur 128 bits
- Un signal 'aes_on_o' indiquant un chiffrement en cours du message
- Une clé sur 128 bits 'key_i'

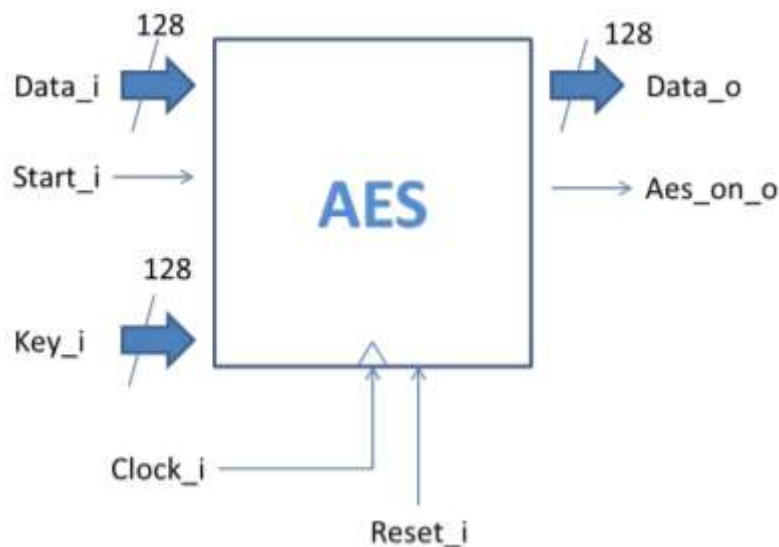


Figure 1 - Composant AES-128 : Entrées/Sorties

Comme expliqué précédemment, le protocole AES comporte 4 sous-fonctions élémentaires : SubBytes, ShiftRows, MixColumns et AddRoundKey. On utilisera une clé primaire de 128 bits, et on fera donc tourner l'algorithme $1+9+1 = 11$ rounds, car on a pour le premier round, un texte en entrée qui subit un AddRoundKey, puis une boucle de 9 rounds, où s'enchaînent SubBytes, un ShiftRows, un MixColumns et un AddRoundKey sur la matrice obtenue. Enfin, un dernier round permet de faire un SubBytes, un ShiftRows et un AddRoundKey, puis en sortie de fournir le « cypher ».

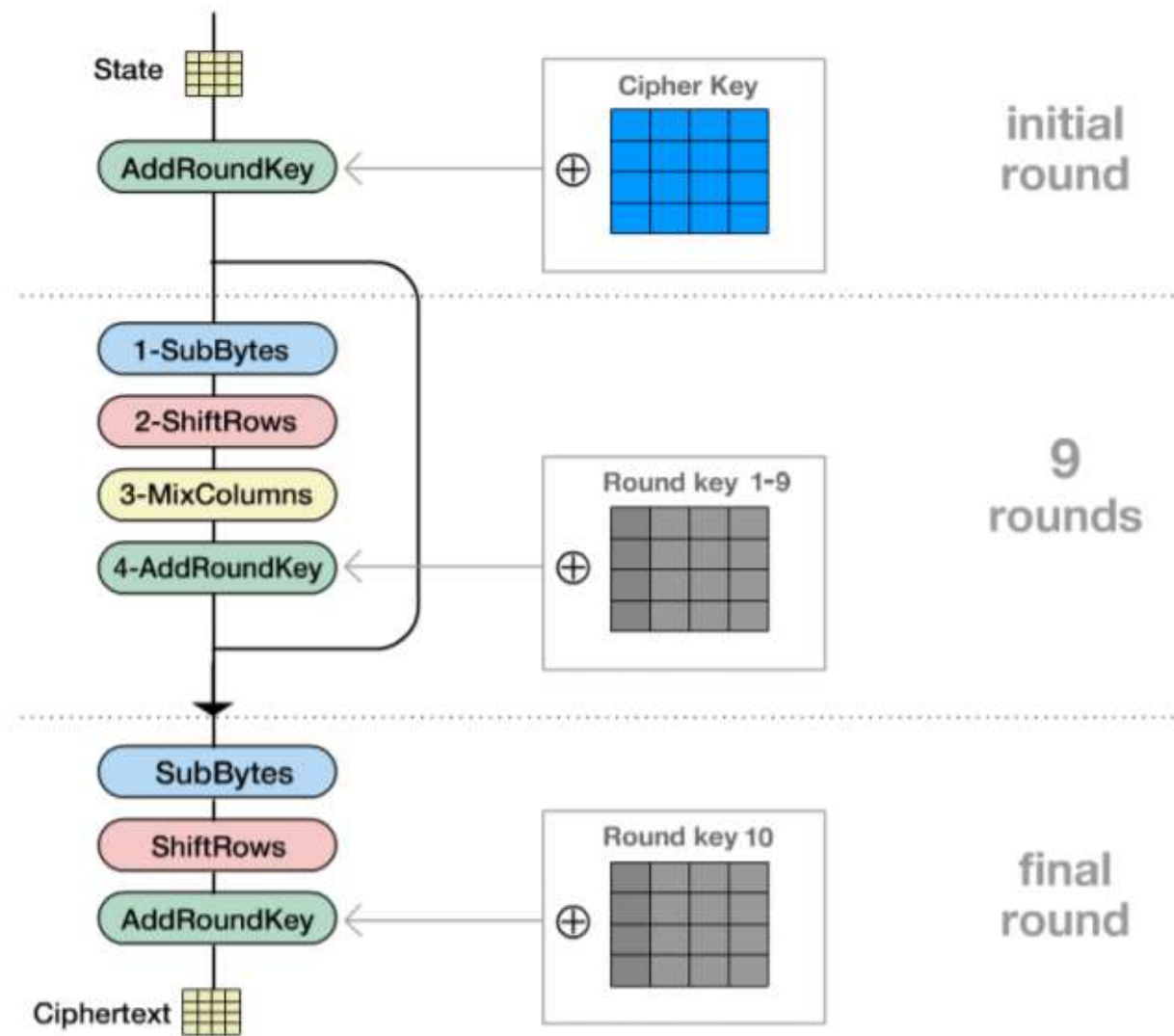


Figure 2 - Description graphique des rounds de l'AES

On va donc utiliser trois modules pour cet AES : AES round, qui est un composant pouvant réaliser les sous fonctions précédemment pour un round, une machine à états finis de Moore pour gérer le protocole, et un générateur de clés de rondes (les sous clés qui héritent de la clé primaire). A ces trois modules, on y ajoute un compteur pour gérer le nombre de rounds

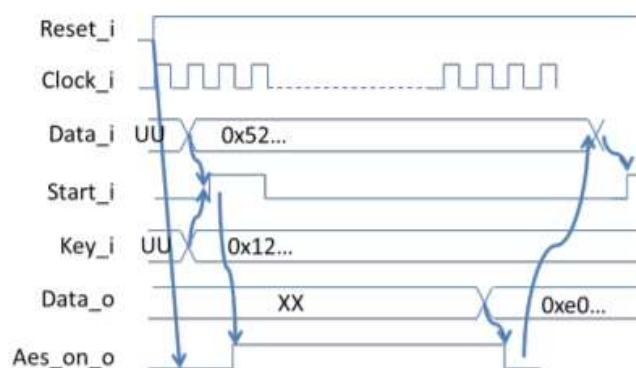


Figure 3 - Chronogramme du protocole

ENVIRONNEMENT DE TRAVAIL

Environnement

Nous travaillerons sous environnement Linux (distribution Xubuntu fournie par l'EMSE), sous Visual Studio.

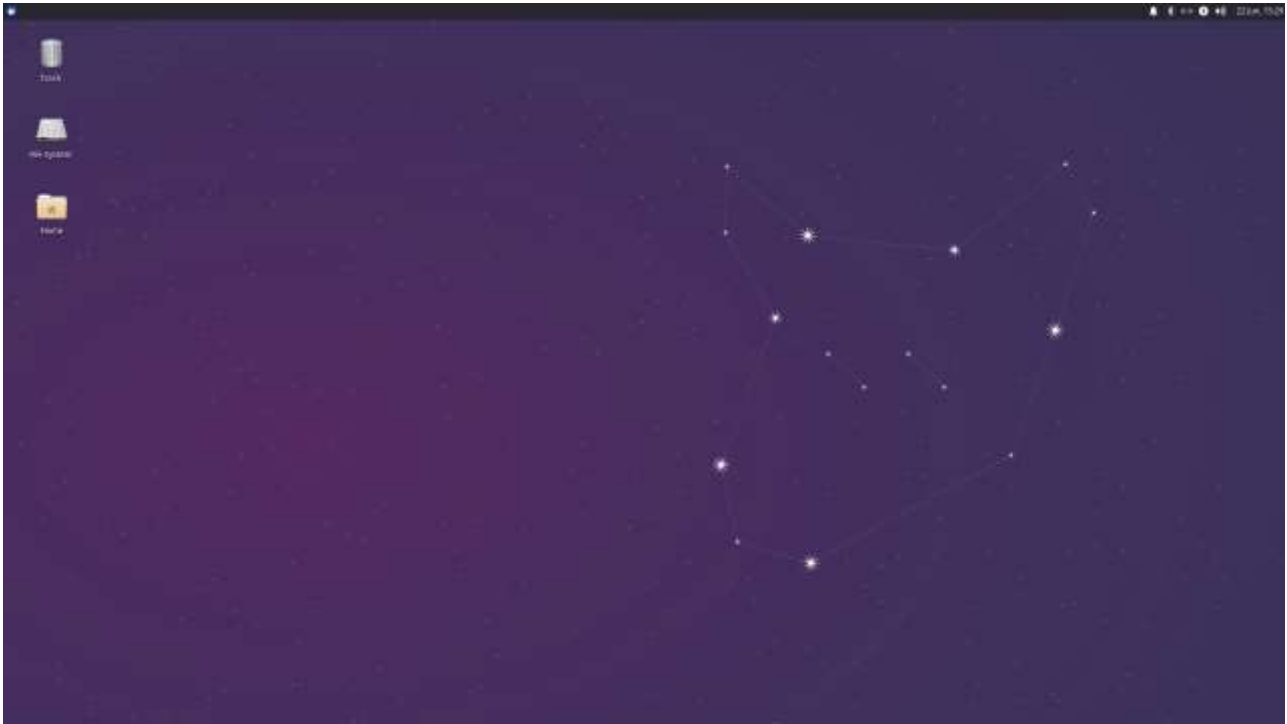


Figure 4 - Un environnement Xubuntu 20.04 LTS Focal Fossa

Compilation

Pour compiler, nous avons réuni l'ensemble des commandes du projet (commandes pour le shell) dans un fichier texte. La commande de compilation est donc « source compile_src.txt ». Le fichier compile_src.txt permet par ailleurs compiler tous les fichiers dans le bon ordre : suppression et création de nouvelles librairies, puis THIRDPARTY (le crypt_pack, voir ci-dessous) puis RTL puis les BENCHS, puis éventuellement un lancement de la simulation.

Bibliothèques

Nous utiliserons tout au long de ce projet la bibliothèque IEEE, courante en VHDL (notamment pratique pour les types qu'elle contient !). De cette bibliothèque, nous utiliserons les notamment le pack de fonctions de `std_logic_1164` qui permettent de modéliser les signaux logiques, mais aussi les fonctions de `numeric_std` pour les éventuelles conversions de bits en entiers et inversement.

En plus de IEEE, nous recourrons à une bibliothèque propre au projet nommée `crypt_pack`, où sera défini des types utiles tout au long de nos travaux comme les `bit4`, `bit8`, `bit16` etc. mais aussi `type_state` (une matrice 4x4), `row_state` (une ligne de ladite matrice), `colum_state` (une colonne de ladite matrice). Le `crypt_pack` se situe dans le dossier `THIRDPARTY`, dans le fichier `CryptPack.vhd`.

Cette bibliothèque contient aussi des fonctions utilitaires comme une fonction « ou exclusif » XOR, utile à notre projet.

Ces bibliothèque seront importées dans la majorité des en-têtes de nos `.vhd` de tous les fichiers du projet.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

library LIB_AES;
use LIB_AES.crypt_pack.all;

library LIB_RTL;
use LIB_RTL.all;
```

Figure 5 - En tête type d'un fichier du projet

Les fichiers VHDL seront localisés dans `SRC`, les fichiers de l'AES dans `RTL` et seront compilés dans `LIB_RTL`. Leurs testbenchs seront localisés dans `BENCH` et compilés dans `LIB_BENCH`. Nous choisirons d'inclure les configurations en suffixe des fichiers VHDL.

RESULTATS FONCTIONNELS OBSERVES

Nous effectuerons l'analyse de manière analogue avec laquelle nous avons codé ce projet : par sous-blocs.

La table de substitution S-Box et la fonction SubBytes

La fonction SubBytes est un sous-bloc qui consiste à remplacer chaque octet du tableau en entrée par son octet correspondant dans la table de substitution. On modélisera la table de correspondance par une S-Box en VHDL.

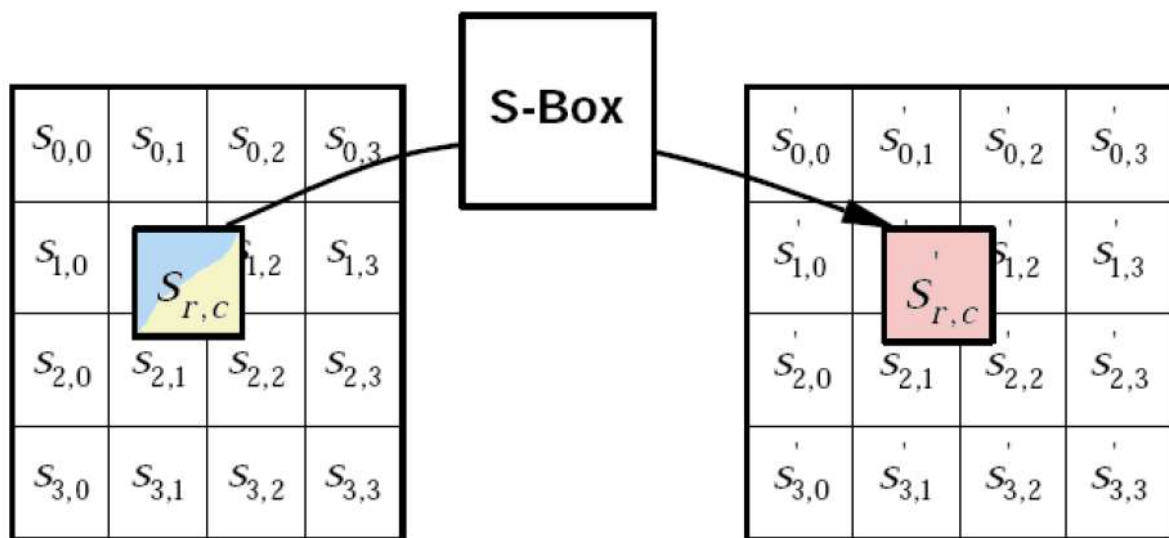


Figure 6 - Principe de la fonction SubBytes

On utilise pour notre S-BOX la table de correspondance suivante :

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

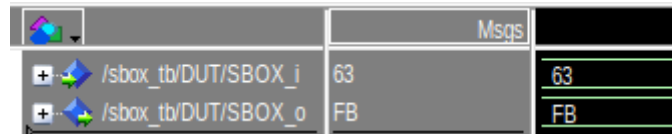
Figure 7 - Table de substitution utilisée pour le projet

La fonction S-Box est basique : une entrée et une sortie de type bit8. La table sera modélisée par une constante. On convertit la sortie en bit8 par la le procédé suivant :

```
begin
  SBOX_o <= std_logic_vector(sboxtab(to_integer(unsigned(SBOX_i))));
```

Figure 8 - Conversion du signal pour une sortie de type bit8 de SBox

On teste la S-Box sur un octets puis on passe au SubBytes.



Signal	Value	Unit
/sbox_tb/DUT/SBOX_i	63	63
/sbox_tb/DUT/SBOX_o	FB	FB

Figure 9 - Résultats du testbench de la SBox

Pour implémenter la fonction SubBytes, nous aurons besoin de 16 s-Box pour substituer les 16 octets de la matrice d'entrée. Le SubBytes dispose d'une entrée et d'une sortie, de type type_state. Pour générer les 16 S-Box, on utilise le mot-clé generate (boucle de port map). Pour SubBytes, on utilise un composant S-Box, on écrit donc une configuration pour l'utiliser.

```
entity SubBytes is
port(
  data_i: in type_state;
  data_o: out type_state);
end entity SubBytes;

architecture SubBytes_arch of SubBytes is
  component SBOX
  port(
    SBOX_i : in bit8;
    SBOX_o : out bit8);
  end component;
begin
  BOX : SBOX port map(
    data_i(0)(0),data_o(0)(0));
  G1 : for h in 0 to 3 generate -- parcours de chaque colonne
    G2 : for i in 0 to 3 generate -- parcours de chaque octet/"mot"
      BOX2 : SBOX port map(data_i(h) (i),data_o(h) (i));
    end generate G2;
  end generate G1;
end architecture SubBytes_arch;

configuration SubBytes_conf of SubBytes is
  for SubBytes_arch
    for G1
      for G2
        for all: SBOX
          use entity LIB_RTL.SBOX(SBOX_arch);
        end for;
      end for;
    end for;
  end for;
end configuration SubBytes_conf;
```

Figure 10 - Architecture de SubBytes

Puis c'est au tour du SubBytes de passer au testbench : on applique le sous bloc sur un tableau d'octets, et on obtient bien toutes les valeurs de sortie attendues, modifiées par correspondance dans la table.

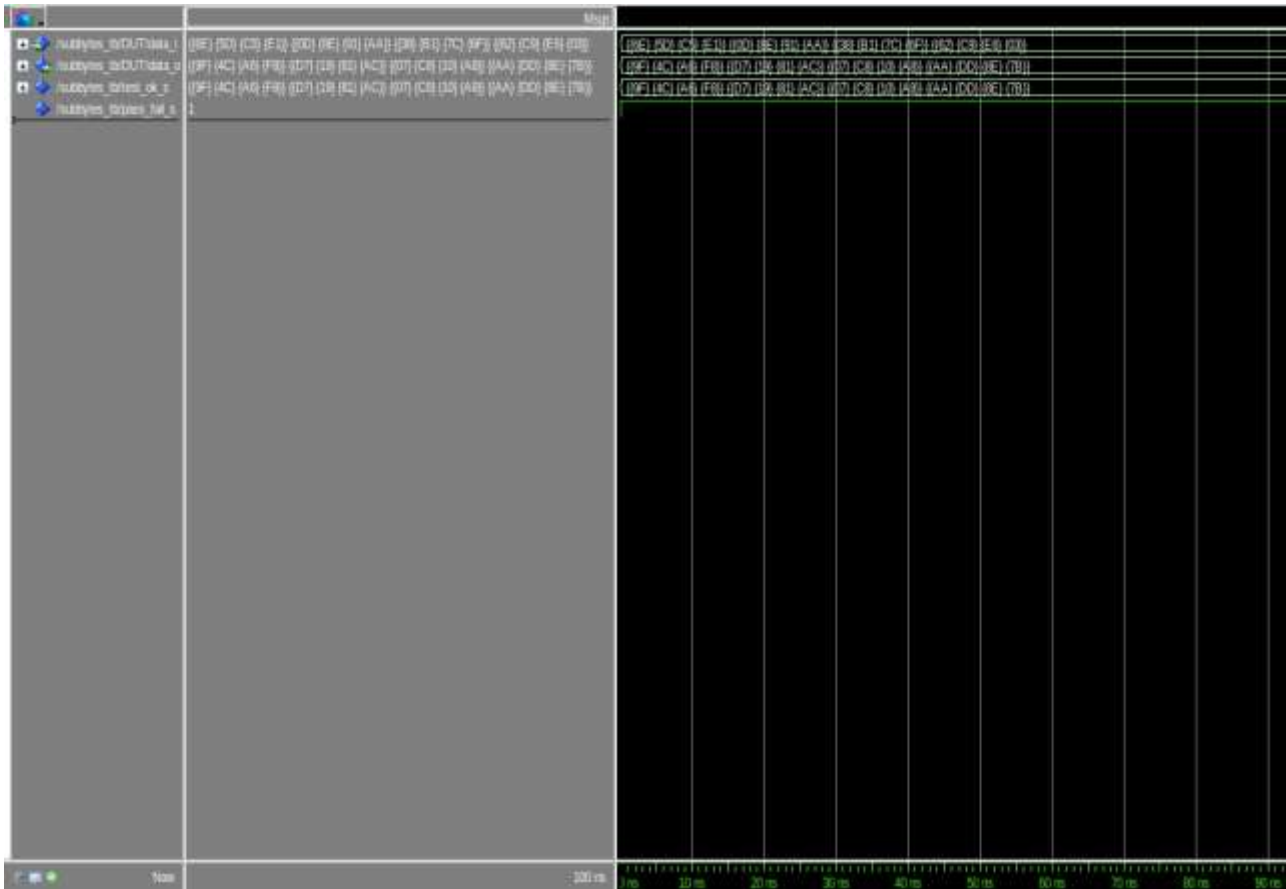


Figure 11 - Résultats du testbench de SubBytes

La fonction ShiftRows

La fonction ShiftRows a pour but d'opérer une permutation cyclique (ie.) une rotation sur les octets des lignes en fonction du numéro de la ligne. Concrètement, la ligne 0 reste telle quelle, la ligne 1 subit un décalage d'un octet vers la gauche, la ligne 2 un décalage de 2 octets vers la gauche et la ligne 3, un décalage de 3 octets vers la gauche.

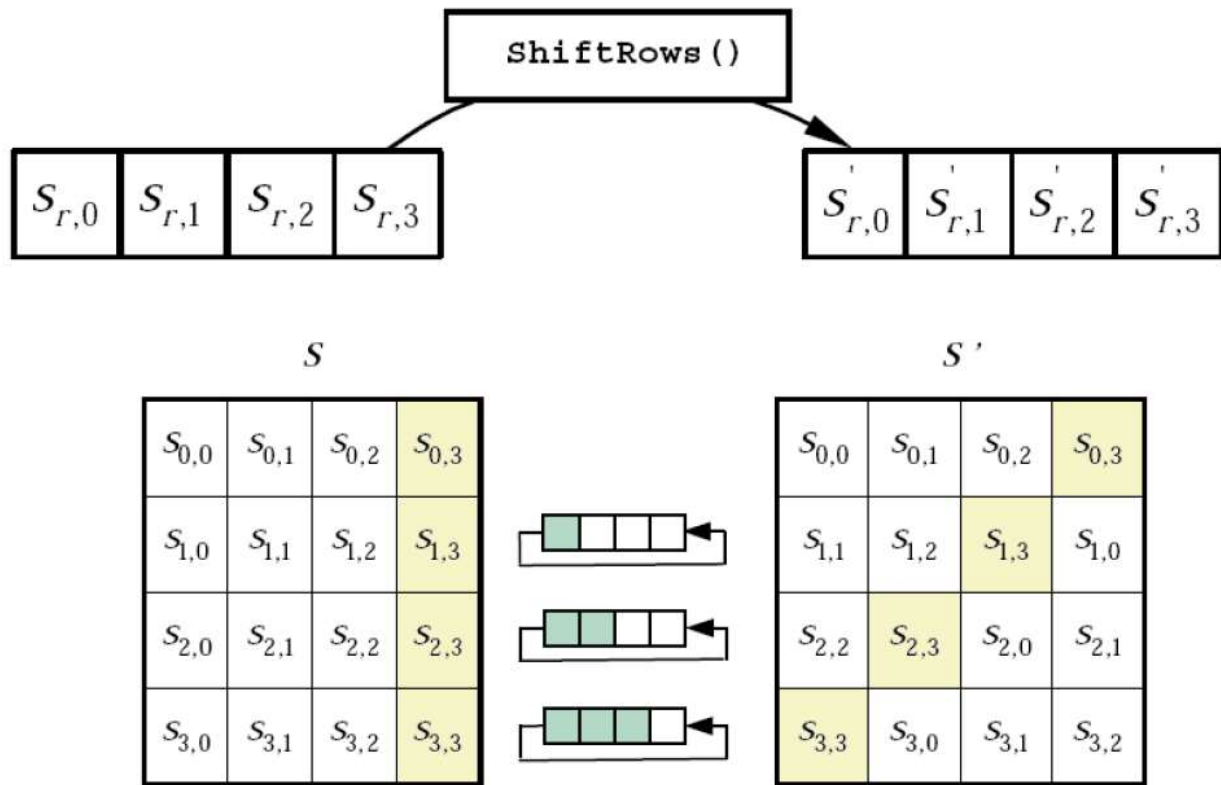


Figure 12 - Illustration de la fonction ShiftRows

Cela se traduit par une entité ayant une entrée et une sortie de type `type_state` (une matrice 4x4 d'octets), et une architecture employant 2 generate (bouclage d'un port map on le rappelle) pour faire ce décalage.

```
entity shiftrows is
    port (
        data_i : in type_state;
        data_o : out type_state
    );
end shiftrows;

architecture shiftrows_arch of shiftrows is
begin
    Row: for i in 0 to 3 generate
        Col: for j in 0 to 3 generate
            data_o(i)(j) <= data_i(i)((i+j) mod 4);
        end generate Col;
    end generate Row;
end shiftrows_arch;
```

Figure 13 - Architecture de ShiftRows

Nous pouvons dès lors tester ce sous bloc sur une matrice 4x4 d'octets.

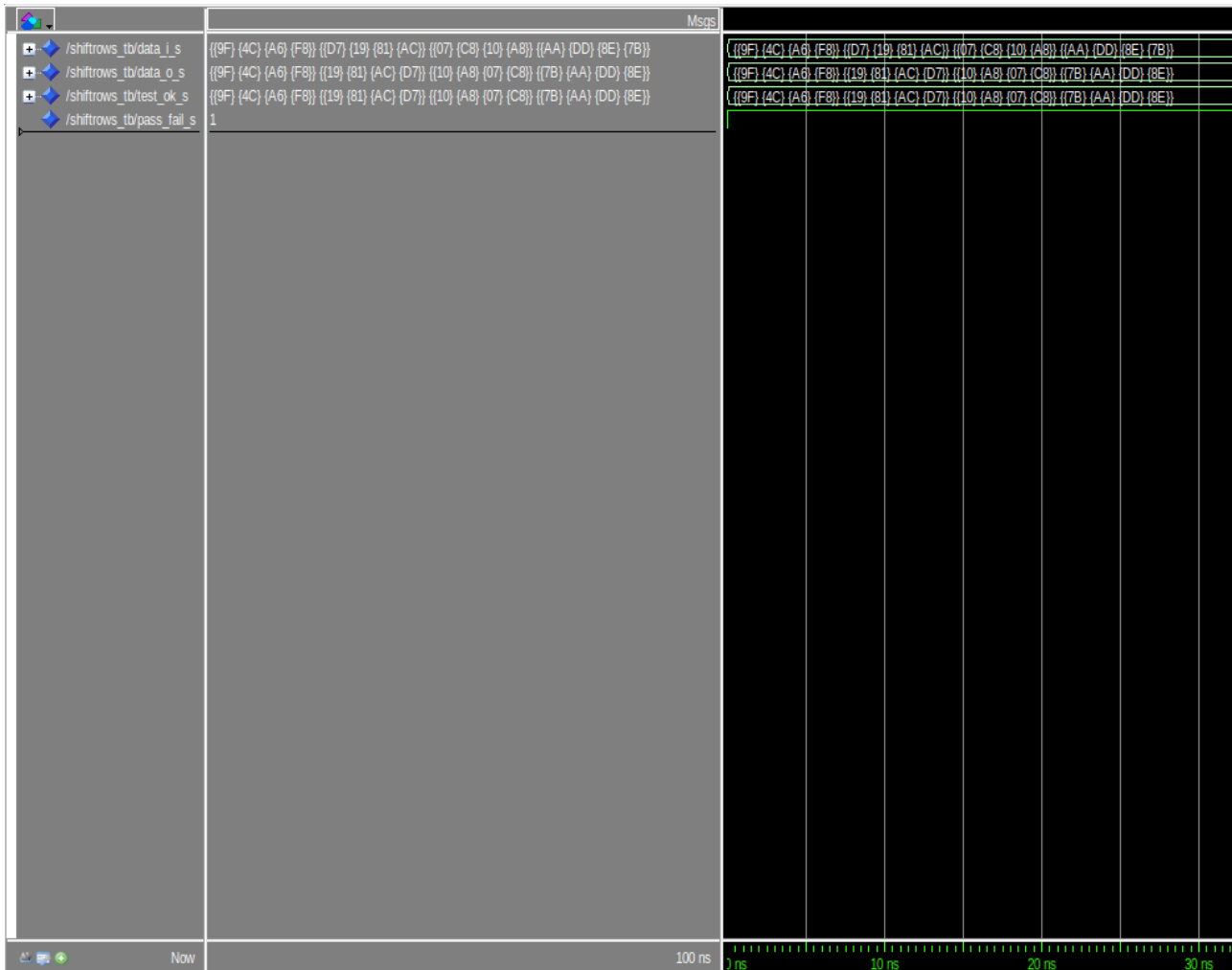


Figure 14 - Résultats du testbench de ShiftRows

La fonction MixColumns

La fonction MixColumns a à la fois pour but de faire une transformation linéaire de la matrice d'entrée mais aussi de la xorer avec une autre matrice.

La fonction MixColumns est le sous-bloc le plus complexe, mathématiquement parlant. Il consiste en une transformation linéaire de la matrice d'entrée, à laquelle on applique un opération booléenne (on applique un « ou exclusif » entre cette matrice d'entrée et une autre matrice).

$$\begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

$$\begin{aligned} s'_{0,c} &= (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\ s'_{1,c} &= s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c} \\ s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c}) \\ s'_{3,c} &= (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}) \end{aligned}$$

Figure 15 - Produit matriciel de la fonction MixColumns

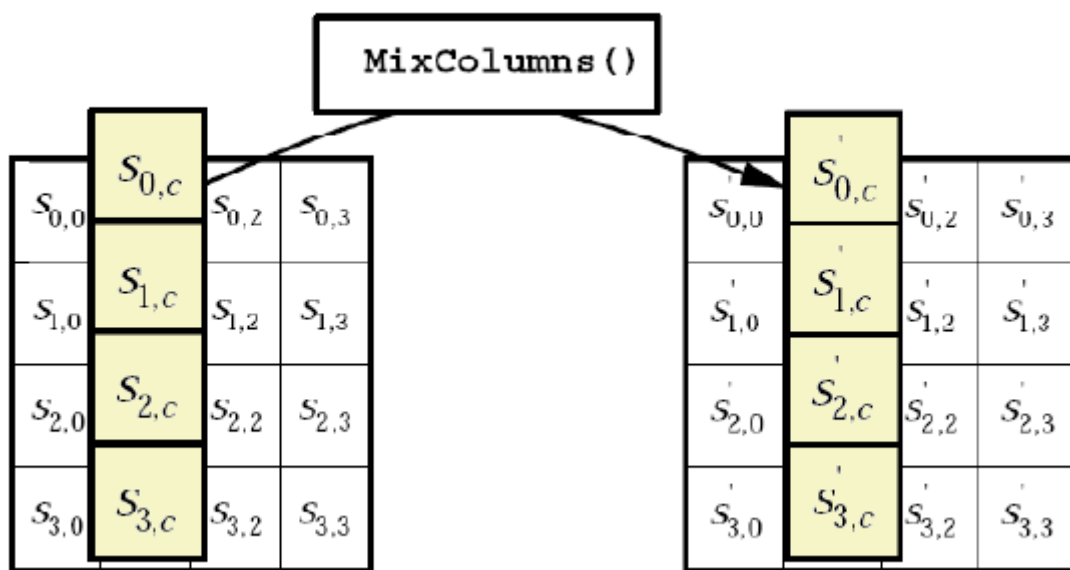


Figure 16 - Application de la fonction MixColumns sur les colonnes de l'état courant

Nous nous permettons de construire cette fonction MixColumns de manière analogue à SubBytes : on construit un bloc élémentaire opérationnel, qu'on appelle MixColumns_Elem, qui prend en entrée un type_column et qui lui applique l'opération MixColumn. Puis nous bouclerons cette opération élémentaire 4 fois dans un second procédé, plus haut niveau (MixColumns), un par colonne pour qu'en entrée et en sortie du sous-bloc on puisse appliquer un type_state.

Le procédé de calcul pour une colonne (mixcolumns_elem) est le suivant.

En entrée datae_i, puis on construit deux signaux : data2_s qui consiste en une multiplication polynomiale (cf. image suivante) par x02 ainsi que data3_s qui consiste en une multiplication polynomiale par x03 (un XOR entre data2_s et datae_i).

Une fois tous les produits stockés dans les signaux, on peut xorer pour avoir la colonne résultante, et on obtient ainsi datae_o.

```
entity mixcolumns_elem is
  port (
    datae_i : in column_state;
    datae_o : out column_state
  );
end mixcolumns_elem;

architecture mixcolumns_elem_arch of mixcolumns_elem is

  signal decal_octet_s : column_state ;
  signal data2_s : column_state ; --datae_i x 0x02
  signal data3_s : column_state ; --datae_i x 0x03

begin
  G1 : for i in 0 to 3 generate
    --On multiplie par 2
    decal_octet_s(i) <= datae_i(i)(6 downto 0)&'0';
    data2_s(i) <= decal_octet_s(i) xor ("000" & datae_i(i)(7) & datae_i(i)(7) & '0' & datae_i(i)(7) & datae_i(i)(7));
    --On multiplie par 3
    data3_s(i) <= data2_s(i) xor datae_i(i);
  end generate G1;

  datae_o(0) <= data2_s(0) xor data3_s(1) xor datae_i(2) xor datae_i(3);
  datae_o(1) <= datae_i(0) xor data2_s(1) xor data3_s(2) xor datae_i(3);
  datae_o(2) <= datae_i(0) xor datae_i(1) xor data2_s(2) xor data3_s(3);
  datae_o(3) <= data3_s(0) xor datae_i(1) xor datae_i(2) xor data2_s(3);

end architecture mixcolumns_elem_arch;
```

Figure 17 - Architecture de MixColumns_Elem

On boucle ce procédé comme expliqué précédemment dans MixColumns.


```

entity mixcolumns is
port(
    data_i: in type_state;
    enable_i: in std_logic;
    data_o: out type_state);
end entity mixcolumns;

architecture mixcolumns_arch of mixcolumns is

    component mixcolumns_elem is
    port (
        datae_i : in  column_state;
        datae_o : out column_state
    );
    end component mixcolumns_elem;

    signal data_o_s : type_state;

begin
    G1 : for i in 0 to 3 generate
    inter : mixcolumns_elem port map(
        datae_i(0) => data_i(0)(i),
        datae_i(1) => data_i(1)(i),
        datae_i(2) => data_i(2)(i),
        datae_i(3) => data_i(3)(i),

        datae_o(0) => data_o_s(0)(i),
        datae_o(1) => data_o_s(1)(i),
        datae_o(2) => data_o_s(2)(i),
        datae_o(3) => data_o_s(3)(i)
    );
    data_o <= data_o_s when enable_i = '1' else data_i;
    end generate G1;
end architecture mixcolumns_arch;

```

Figure 18 - Architecture de MixColumns

Nous affichons ici le testbench directement du processus MixColumns_Elem, puis celui de MixColumns réalisé dans la foulée (avec une entrée bit128 de l'annexe et une conversion sur le volet).

	Msgs	
+ /mix_columns_elem_tb/datae_i_s	{9F} {19} {10} {7B}	{9F} {19} {10} {7B}
+ /mix_columns_elem_tb/datae_o_s	{65} {E6} {2B} {45}	{65} {E6} {2B} {45}
+ /mix_columns_elem_tb/test_ok_s	{65} {E6} {2B} {45}	{65} {E6} {2B} {45}
+ /mix_columns_elem_tb/pass_fail_s	1	

Figure 19 - Résultat du testbench de MixColumns_Elem

	Msgs	
+ /mixcolumns_tb/data_i_s	{{AF} {16} {CE} {BC}} {{E6} {91} {62} {44}} {{01} {06} {D3} {20}} {{D5} {AB} {B1} {AE}}	{{AF} {16} {CE} {BC}} {{E6} {91} {62} {44}} {{01} {06} {D3} {20}} {{D5} {AB} {B1} {AE}}
+ /mixcolumns_tb/data_o_s	{{A0} {29} {43} {21}} {{AE} {8E} {D5} {FA}} {{2F} {6D} {D9} {51}} {{BC} {E0} {81} {FC}}	{{A0} {29} {43} {21}} {{AE} {8E} {D5} {FA}} {{2F} {6D} {D9} {51}} {{BC} {E0} {81} {FC}}
+ /mixcolumns_tb/init_dat...	AFE601D5169106ABCE62D3B1BC4420AE	AFE601D5169106ABCE62D3B1BC4420AE
+ /mixcolumns_tb/enable_s	1	

Figure 20 - Résultat du testbench de MixColumns

La fonction AddRoundKey

Le sous-bloc AddRoundKey consiste simplement à crypter une entrée par une clé de ronde, les deux étant fournies en entrée (de type type_state). Il en résulte une sortie, chiffrée (de type type_state elle aussi). L'architecture du sous-bloc est basique : il consiste juste en un « ou exclusif » appliqué à chaque colonne de la matrice d'entrée par celle de la clé.

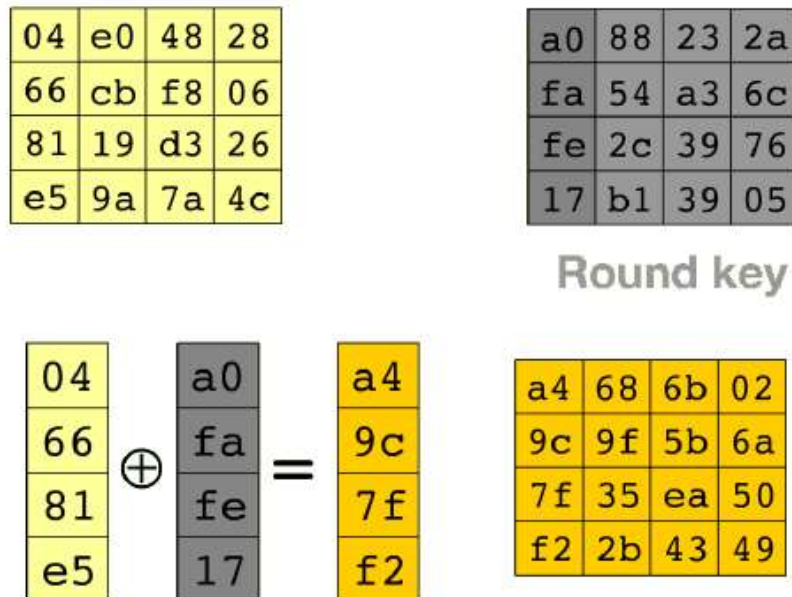


Figure 21 - Calcul de la fonction AddRoundKey

On utilise donc un double generate pour appliquer ces XOR, octet par octet. Nous testons ensuite et pouvons vérifier à l'aide de l'annexe la bonne exécution des fonctions.

```
entity addroundkey is
port(
    data_i1: in type_state;
    data_i2: in type_state;
    data_o: out type_state);
end entity addroundkey;

architecture addroundkey_arch of addroundkey is
begin
    Rang: for i in 0 to 3 generate
        Col: for j in 0 to 3 generate
            data_o(i)(j) <= data_i1(i)(j) xor data_i2(i)(j);
        end generate Col;
    end generate Rang;
end architecture addroundkey_arch;
```

Figure 22 - Architecture de AddRoundKey

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

Figure 23 - Résultats du testbench de AddRoundKey

Le bloc de génération des clés de ronde

Pour des raisons de simplicité de lecture, nous ne mentionnerons ici que les résultats obtenus dans la fonction `KeyExpansion_I_O`. Les testbenchs de `KeyExpansion` et de `KeyExpansion_FSM` sont fournis sur le git en Annexe 1.

La fonction élémentaire `KeyExpansion`

La génération des clefs de ronde s'applique à partir de la ronde 1. Les clefs de rondes sont issues d'un processus faisant appel à une fonction de rotation appliquée sur une colonne (`RotWord`), de l'appel à la table de substitution (`SubBytes`) et la fonction booléenne XOR. En entrée du procédé : la « vieille » clé de ronde ainsi que `Rcon`, une matrice constante définie dans `crypt_pack`. En sortie la nouvelle clé de ronde.

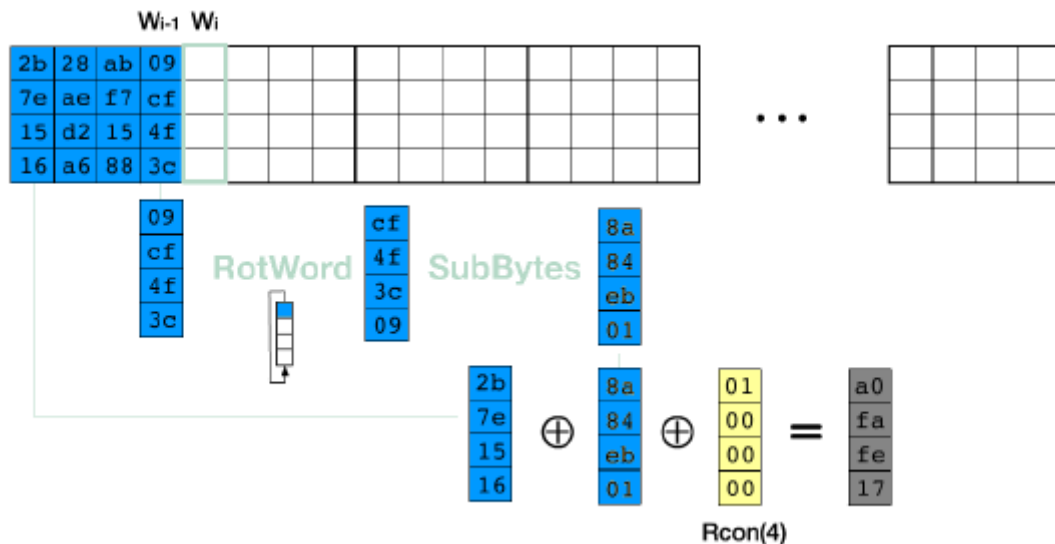


Figure 24 - Calcul de la première colonne ($i \bmod N_k = 0$) à l'aide de la matrice `Rcon`

Le procédé est le suivant :

On prend en entrée `key_i`, de type bit 128. On la convertit en type `key_state` : la clé devient un `word_i`. La dernière colonne subit une rotation selon `RotWord`, puis la colonne subit une substitution selon `SBox` : le résultat est stocké dans un signal : `word_rotSBOX_s`. La sortie résultante des deux procédés est « xorée » avec la colonne de `Rcon`, et avec la « vieille » colonne 0 de la clé entrante. On obtient ainsi la nouvelle colonne 0 de la future clé de ronde. Puis pour la colonne 1, on « xor » la vieille colonne 1 par la nouvelle colonne 0, et ainsi de suite selon le graphe suivant. Enfin on reconvertit le tout en sortie en bit128. Concrètement, en VHDL, cela se traduit ainsi :

```

begin
  -- conversion en key state (4x4 bits array)
  col : for j in 0 to 3 generate
    row : for i in 0 to 3 generate
      word_i_s(j)(i) <= key_i(127-32*j-8*i downto 120-32*j-8*i);
    end generate;
  end generate;

  -- on calcule WB
  -- transfo SBOX
  SB : for i in 0 to 3 generate
    cell : SBOX port map i;
    SBOX_i => word_i_s(3)((i+1) mod 4);
    SBOX_o => word_rotSBOX_s(i);
  end generate SB;

  word_o_s(0) <= word_rotSBOX_s xor (rcon_i, X'00", X'00", X'00") xor word_i_s(0);

  -- on calcule le reste
  WDCol : for j in 1 to 3 generate
    word_o_s(j) <= word_o_s(j-1) xor word_i_s(j);
  end generate;

  -- reconversion en bit 128
  KDRow : for j in 0 to 3 generate
    KRow : for i in 0 to 3 generate
      expansion_key_o(127-32*j-8*i downto 120-32*j-8*i) <= word_o_s(j)(i);
    end generate;
  end generate;

end architecture;

```

Figure 25 - Architecture de KeyExpansion

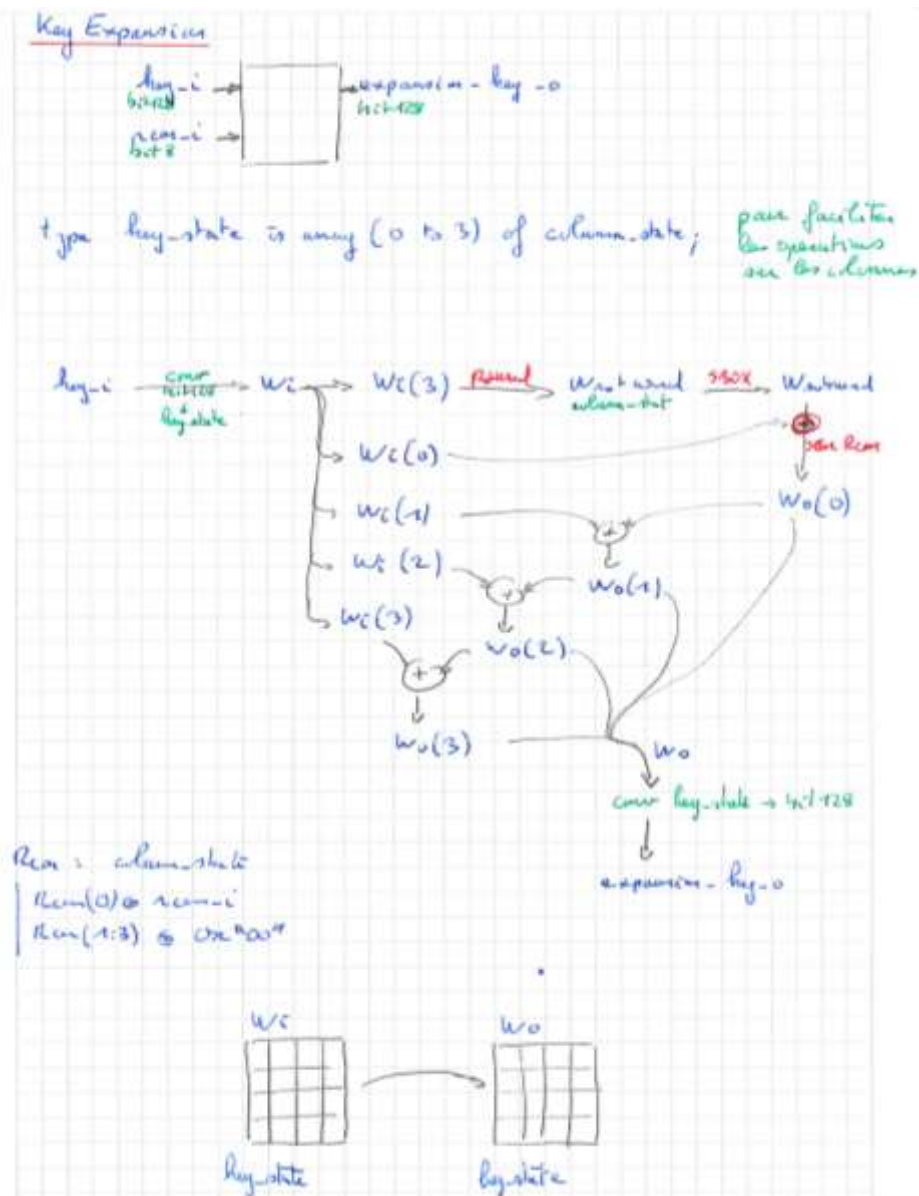


Figure 26 - Fonctionnement détaillé de KeyExpansion

La machine à états finis de Moore pour la génération de clés de ronde

On se servira d'une machine à états finis de Moore qui fonctionne selon le graphe suivant pour la gestion de la génération des clés de ronde (pour une simplification de KeyExpansion_I_O).

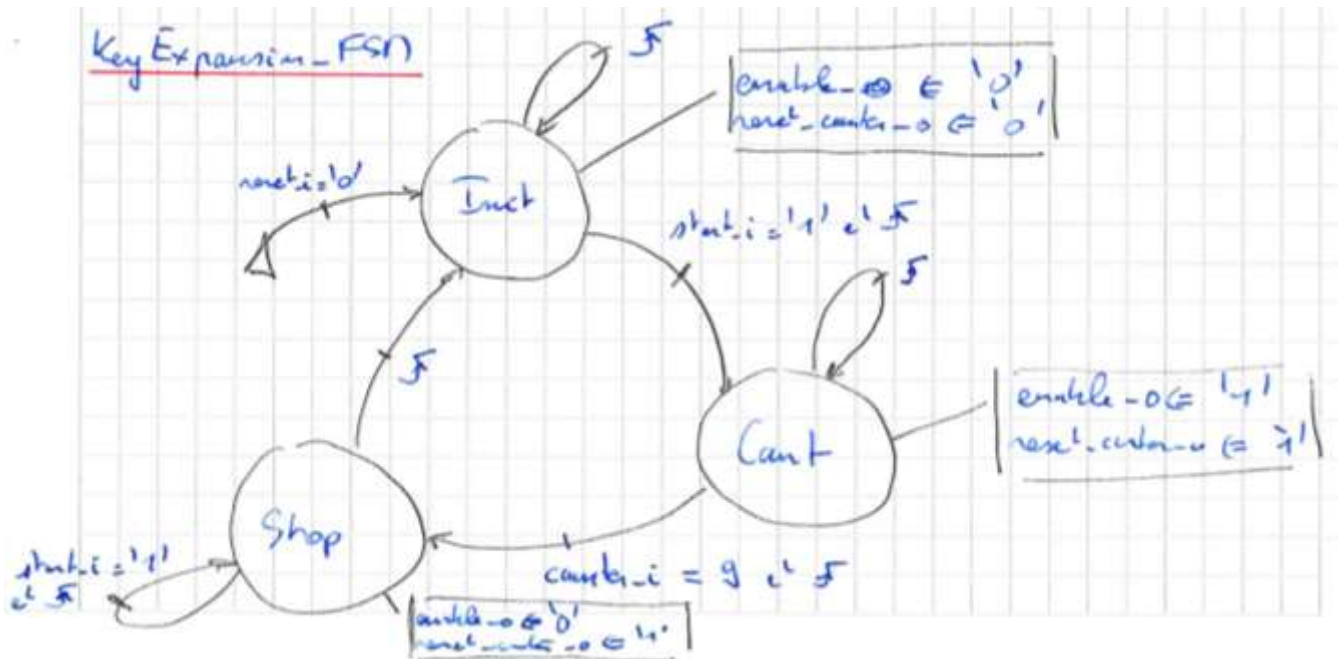


Figure 27 - Schéma de principe de la machine à états finis (KeyExpansion_FSM)

La fonction haut niveau KeyExpansion_I_O

La fonction haut niveau KeyExpansion_I_O permet la gestion de la génération des clés de ronde. Elle est un composant principal du procédé global qu'est l'AES. Cette fonction KeyExpansion_I_O comprend :

- le composant primaire KeyExpansion
- un compteur
- la Machine de Moore bâtie pour sa gestion.
- un registre placé avant la sortie et un démultiplexeur

Le bloc fonctionne selon le schéma suivant :

Détaillons un peu ledit procédé. En entrée :

- start_i qui permet de démarrer le procédé
- clock_i, l'horloge qui régit le procédé
- reset_i, un reset asynchrone
- key_i, pour introduire la clé primaire (de type bit128)

En sortie, les clés de rondes successives : expansion_key_o (de type bit128).

Ces composants sont câblés selon le graphe suivant :

Cela donne en VHDL :

```
architecture keyexpansion_I_O_arch of keyexpansion_I_O is
    component keyexpansion is
        port ( key_i      : in  bit128;
              rcon_i     : in  bit8;
              expansion_key_o : out bit128;
        end component keyexpansion;

    component keyexpansionFSM is
        port ( start_i : in std_logic;
              clock_i  : in std_logic;
              counter_i : in bit4;
              reset_i   : in std_logic;
              enable_o  : out std_logic;
              reset_counter_o : out std_logic;
        end component keyexpansionFSM;

    component Counter is
        port(reset_i : in std_logic;
              enable_i : in std_logic;
              clock_i : in std_logic;
              count_o : out bit4);
    end component Counter;

    signal counter_s : bit4;
    signal rcon_s : bit8;
    signal enable_s : std_logic;
    signal reset_counter_o_s : std_logic;
    signal key_reg_s : bit128;
    signal keystate_s : bit128;
    signal expansion_key_o_s : bit128;

begin
    rcon_s <= Rcon(to_integer(unsigned(counter_s))and 10);
    --keyexpansion component
    U0 : keyexpansion
    port map(
        key_i => keystate_s,
        rcon_i => rcon_s,
        expansion_key_o => expansion_key_o_s);
    --keyexpansionFSM component
    U1 : keyexpansionFSM
    port map(
        start_i => start_i_IO,
        clock_i => clock_i_IO,
        reset_i => reset_i_IO,
        counter_i => counter_s,
        enable_o => enable_s,
        reset_counter_o => reset_counter_o_s
    );

    --counter component
    U2 : Counter
    port map(
        reset_i => reset_counter_o_s,
        enable_i => enable_s,
        clock_i => clock_i_IO,
        count_o => counter_s
    );

    PB : process(expansion_key_o_s, clock_i_IO, reset_i_IO, enable_s) --registre
    begin
        if reset_i_IO = '0' then -- reset asynchrone
            for row in 0 to 3 loop
                for col in 0 to 3 loop
                    key_reg_s(127 - 32*col - 8*row downto 120 - 32*col - 8*row) <= (others => '0');
                end loop;
            end loop;
        elsif (clock_i_IO'event and clock_i_IO = '1') then
            if (enable_s = '1') then
                key_reg_s <= expansion_key_o_s;
            else
                key_reg_s <= key_i_IO; --128
            end if;
        end if;
    end process PB;

    keystate_s <= key_reg_s when enable_s = '1' else key_i_IO; --mux
    expansion_key_o_IO <= keystate_s;
end keyexpansion_I_O_arch;
```

Figure 28 - Architecture de KeyExpansion_I_O

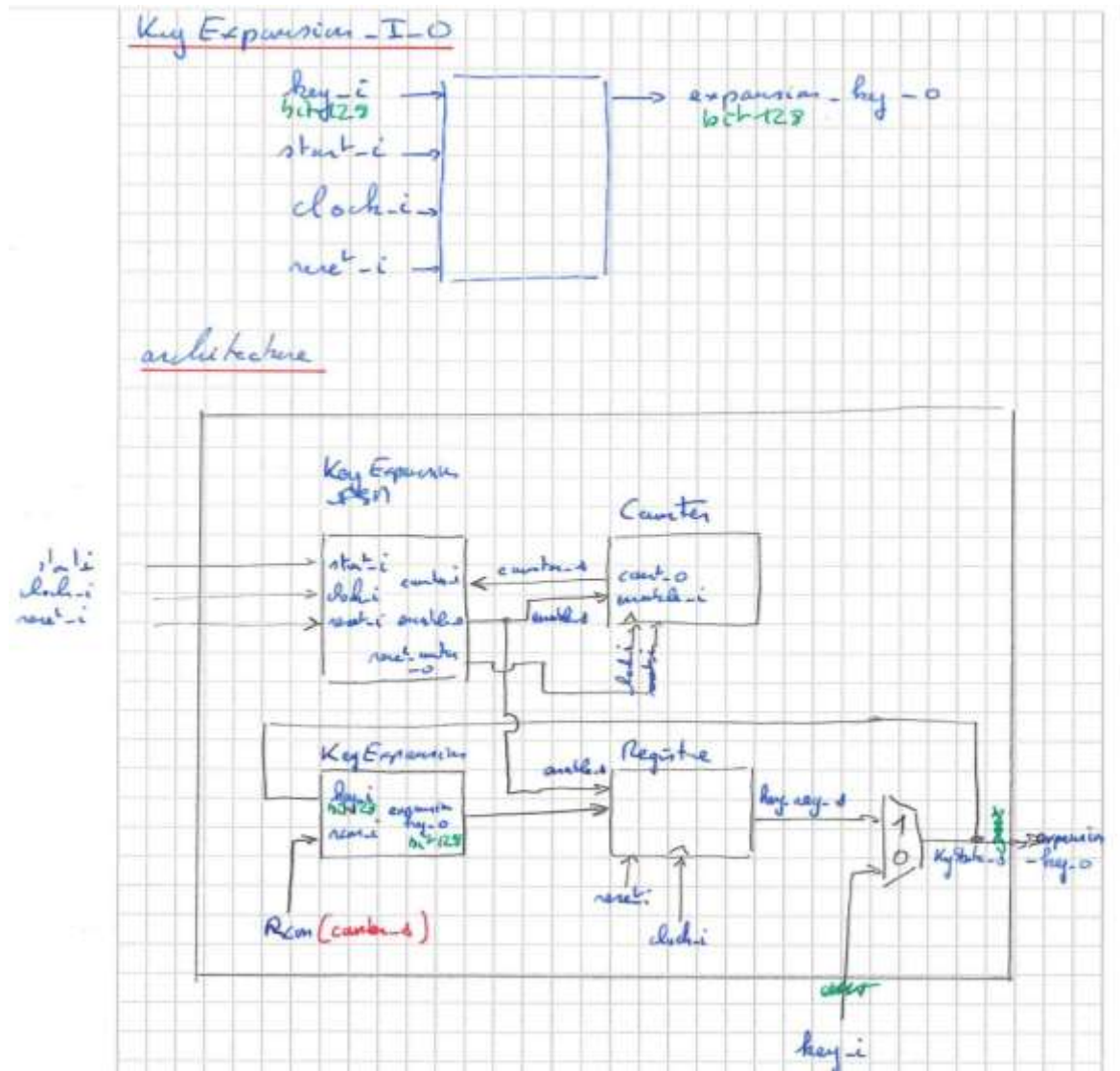


Figure 29 - Fonctionnement et sous-blocs de KeyExpansion_I_O

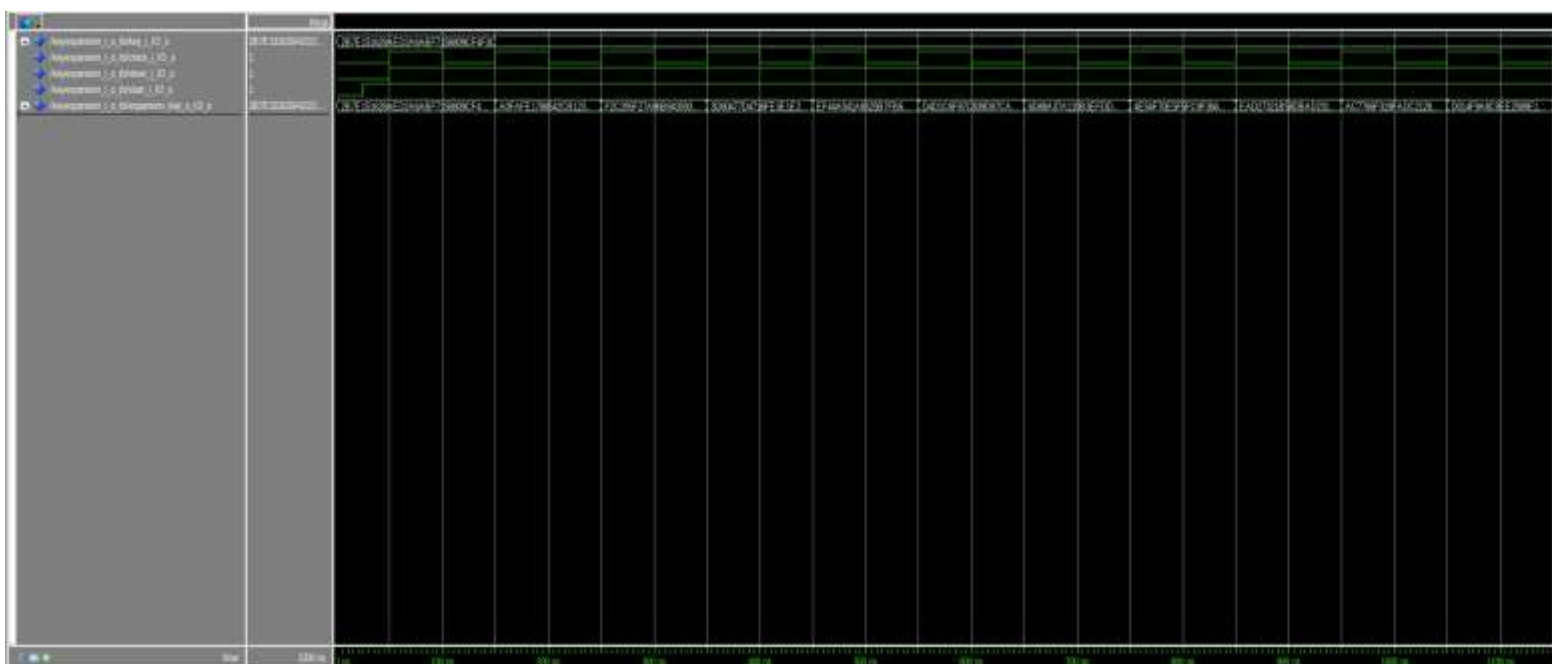


Figure 30 - Résultats du testbench de KeyExpansion_I_O

IMPLEMENTATION DE L'AESROUND

Description

L'AESRound consiste, basiquement, en une ronde élémentaire de l'AES, qu'on bouclera par la suite, qu'elle soit la première, la dernière ou une ronde intermédiaire. Le but est donc de minimiser le nombre de blocs au sein de l'AES, en réunissant tous les sous-blocs dans un AES global.

Ici un graphe des entrées/sorties du composant AES :

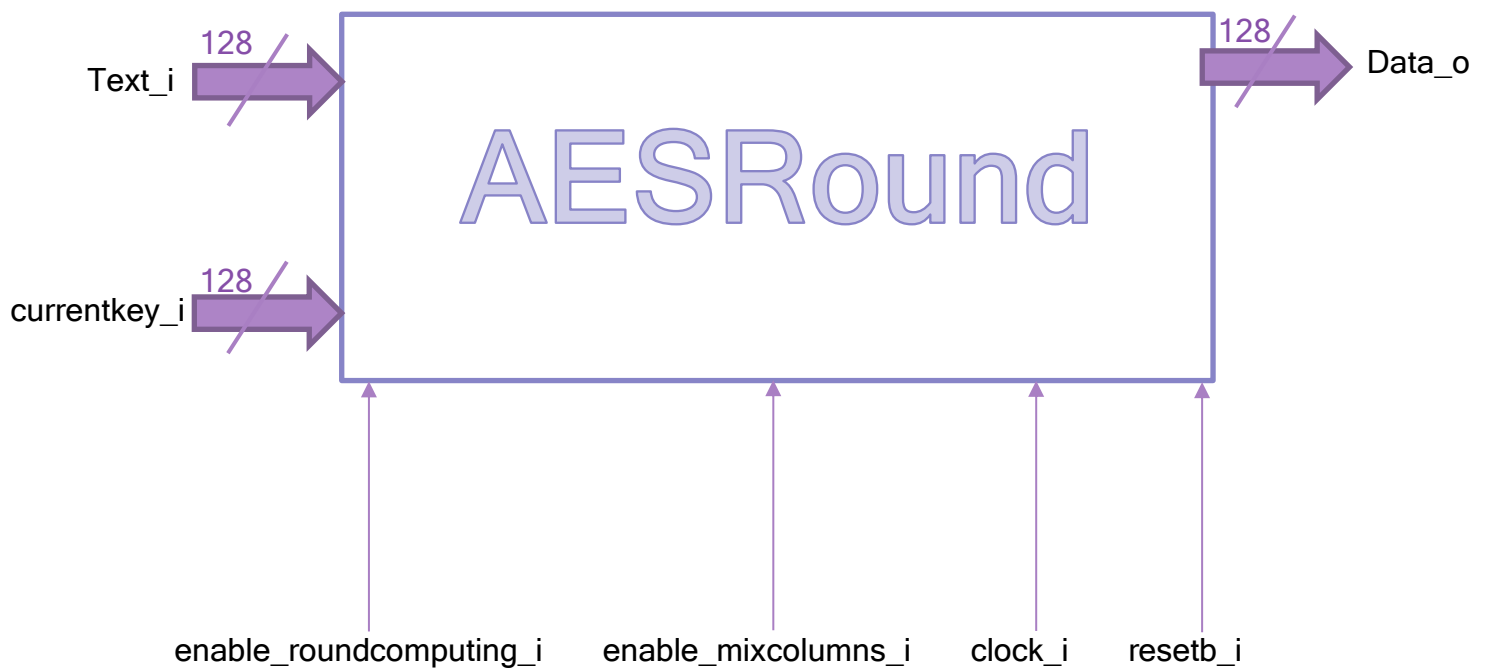


Figure 31 - Schéma des entrées/sorties de l'AESRound

On utilise `enable_roundcomputing_i` et `enable_mixcolumns` pour s'occuper des rondes 0 et 10, particulières. On a dès lors intégré dans le procédé un mux pour récupérer l'entrée au round 1, ainsi qu'un registre (basculé D). Le procédé prend en entrée et en sortie des bits 128, mais les sous-blocs nécessitent des `type_state` donc une conversion des différentes entrées/sorties s'impose. On obtient donc un AESRound comme ceci :

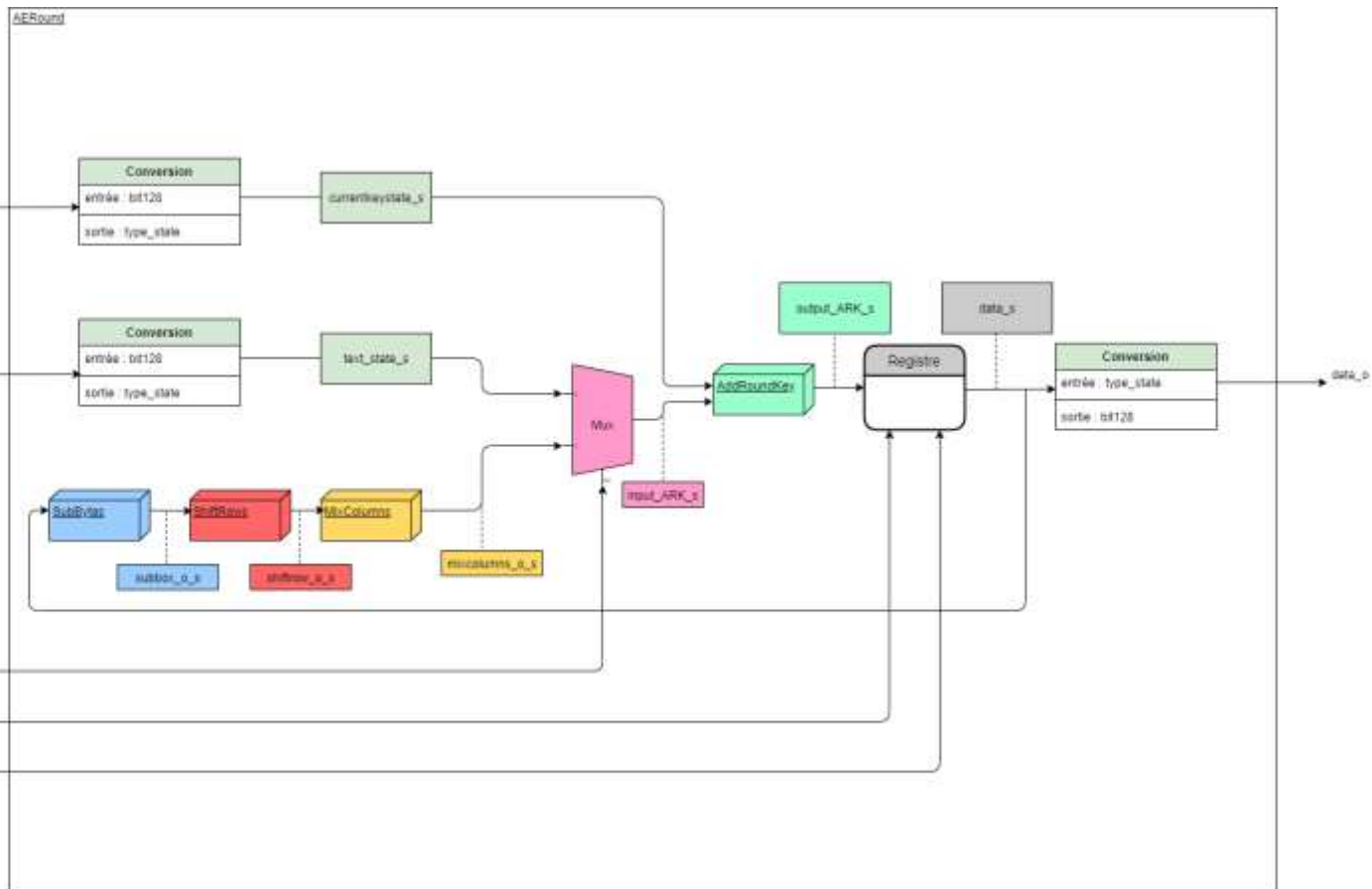


Figure 32 - Schéma de principe de l'AESRound

Tests

En prenant les signaux de l'annexe pour un test :



Figure 33 - Résultats du testbench de l'AESRound

On observe que le procédé s'est effectué sans pépins, et que nous pouvons donc passer à son bouclage, en changeant les clés en entrée cette fois ci !

COMPOSANTS ANNEXES

Le compteur

Le compteur permet de donner le numéro du round à la FSM de l'AES ainsi à la FSM du bloc générateur des clés de ronde. Concrètement, le compteur réside en un procédé comptant le nombre de front montants de l'horloge, qui est une entrée. On a aussi en entrée une réinitialisation et un enable_i qui permet de l'activer ou non, c'est un process qui compte le nombre de front montant de l'horloge, on peut le remettre à 0 via son entrée init_i, et il n'est actif que si enable_i est à 1. Il prend aussi en compte le reset, et renvoie en sortie un bit4 qui sera le numéro du round.



Figure 34 - Schéma des entrées sorties du compteur

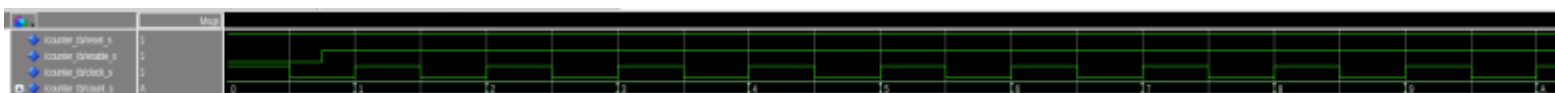


Figure 35 - Résultats du testbench du compteur

IMPLEMENTATION FINALE DE L'AES

Machine à états finis de Moore pour la gestion du procédé global

On utilise une machine à états finis de Moore pour la gestion globale de tout le procédé.

Cette machine a été construite en amont du projet. Elle comporte 6 étapes: hold, init, round0, roundn, lastround, done.

Elle fonctionne selon ce diagramme :

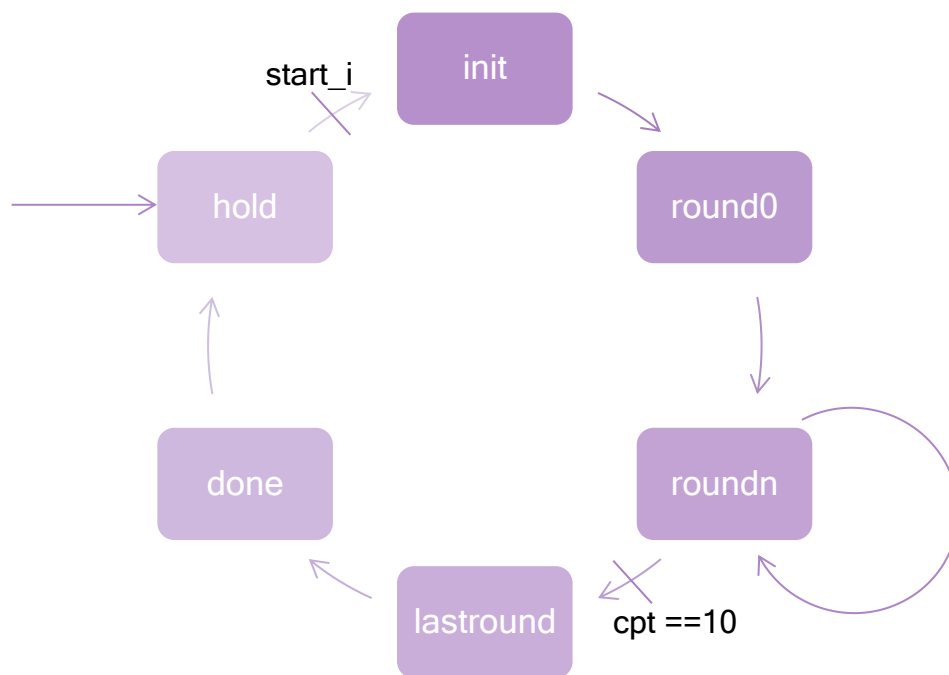


Figure 36 - Principe de fonctionnement de la FSM globale

0 – Etat initial

L'état initial est l'état de hold. Dans cet état, l'AES ne renvoie que des zéros et le procédé de chiffage est inactif. Toutes les sorties sont à 0, sauf `enable_mixcolumns_o` qui est à 1. Passe à l'état 1 si `start_i` vaut 1 sur un front montant d'horloge.

1 – init

On passe à un `start_keyexpander`, `reset_keyexpander` (qui restera haut tout le long du procédé). On indique `aes_on`.

2 – Round 0

Il s'agit du round 0, on initialise le compteur.
Passe à l'état 3 au prochain coup d'horloge.

3 – Round 1-9 (roundn)

On passe `enable_round_computing` à 1.
Passe à l'état 4 si `compteur_i` vaut 10.

4 – lastround

Passe `enable_MC_o` à 0, `init_cpt` à 1.
Passe à l'état 5 au prochain coup d'horloge.

5 – done

L'état 5 est uniquement un état d'output, il permet au compteur de se réinitialiser, et à l'AES d'afficher le résultat sur un coup d'horloge, avant de revenir en position reset à l'état initial. Par choix, on laissera l'output `aes_on` à 1 dans cet état, car l'aes n'est pas encore prêt à recevoir de nouvelles instructions.

Assemblage final des blocs

On a donc bâti et testé tous les composants nécessaires à la réalisation de l'AES.

On passe à l'assemblage final de l'AES. Le composant prend en entrée la clock, le reset, le start, et le texte clair, et en sortie, le texte chiffré et un signal aes_on. L'architecture de l'AES comportera les composants AESRound, Compteur, FSM AES, Key_Expansion_I_O, mais aussi un multiplexeur qui permettra de décider s'il faut sortir un vecteur nul ou le texte chiffré. Ci-dessous un diagramme résumant l'entité :

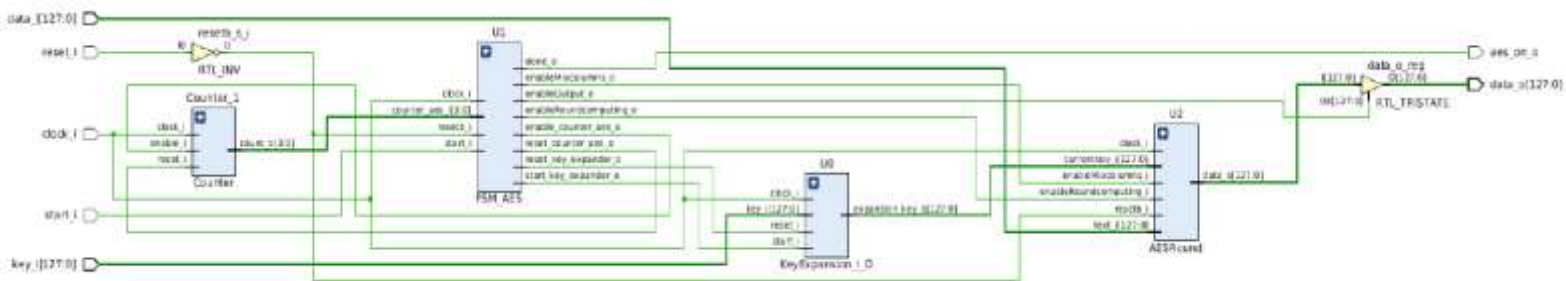


Figure 37 - Schéma de principe du protocole AES

Testbenchs et résultats finaux

Le code de test du processus final est basique : il s'agit simplement de définir une clock, une clé, le texte à crypter. Nous pouvons donc observer les résultats de notre implémentation. Durant tout le long du protocole, la sortie est nul : seulement à la fin, nous pouvons observer la sortie, qui s'avère être correcte.

Nous avons par ailleurs affiché les états de la FSM de Moore, pour bien comprendre l'état du protocole à tout instant

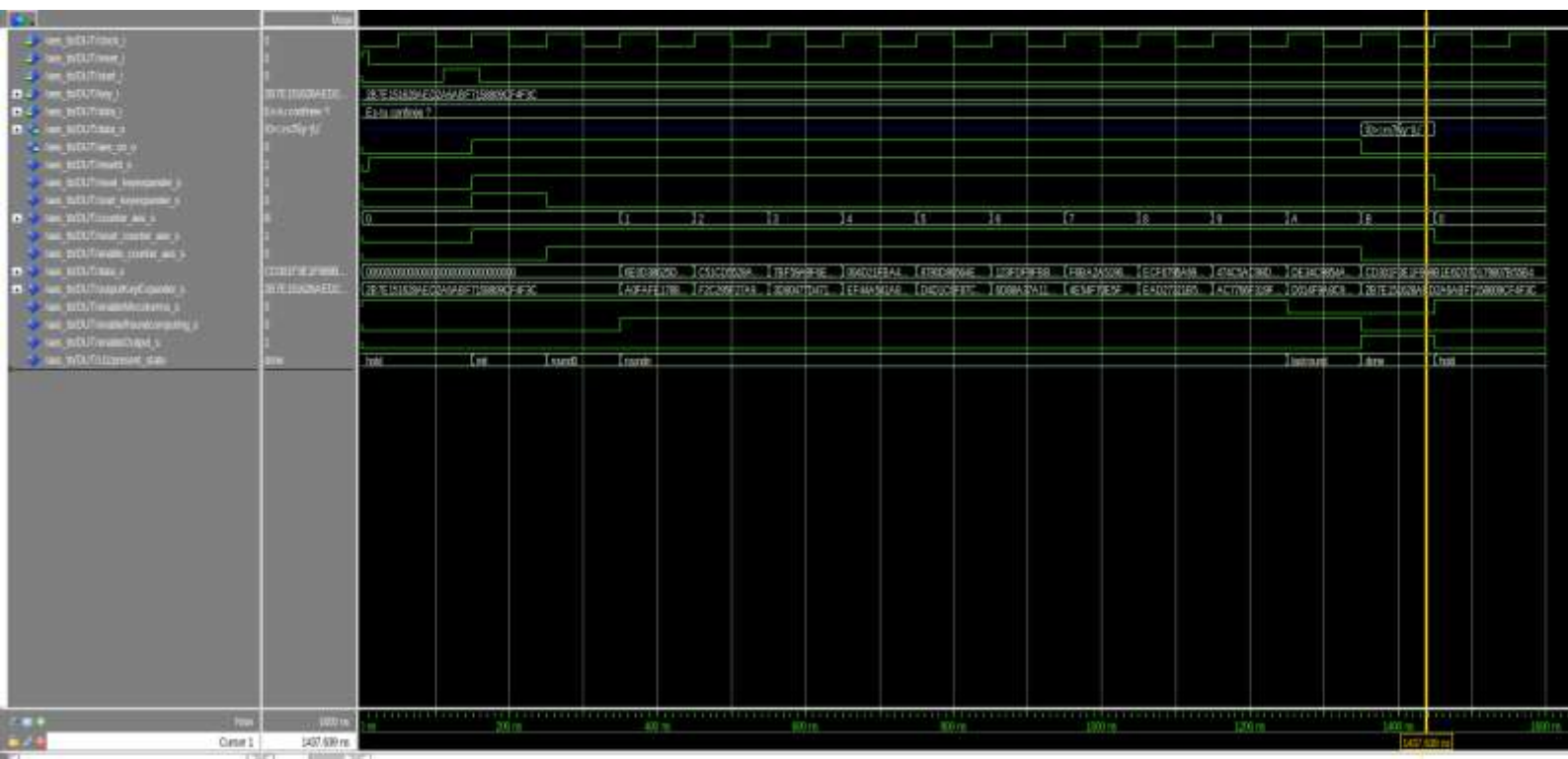


Figure 38 - Résultats du testbench de l'AES

PROBLEMES OBSERVES ET PISTES D'AMELIORATION

Problèmes rencontrés

J'ai, durant le développement de ce projet, rencontré peu de grosses embûches.

Le premier problème de taille que j'ai rencontré est survenu lors du test de l'AES final : un SubBytes défectueux était passé entre les mailles du filet, et j'avais donc un AESRound défectueux. Il a donc fallu que je diagnostique le problème (la sortie était affichée comme indéterminée, « XXXXXXXXXXXX »), puis que je reprenne mon SubBytes (après une relecture de ma SBox). Par la suite, ma sortie n'était plus undefined.

Mais un second problème est donc survenu par la suite : la sortie ne donnait jamais le résultat. J'ai d'abord pensé à un problème relatif au compteur, mais j'avais corrigé tous les compteurs récemment. En vérité, le problème venait de keyexpansion_I_O : un simple modulo sur l'attribution du Rcon pour le sous bloc suffisait à régler le problème.

Enfin pour terminer, je suis un peu frustré (comme tous j'imagine) de m'être rendu compte des possibilités qu'offraient Modelsim (raccourcis, sous signaux etc.). En ayant conscience de ces différentes possibilités, j'aurais plus bien évidemment déboguer mon code beaucoup plus rapidement.

Voies d'amélioration

On peut imaginer plusieurs voies d'améliorations actuellement sur notre code.

La première serait d'implémenter le décryptage en plus de l'encryptage, en VHDL, comme on peut le trouver sur certains Git.

Puis, on pourrait imaginer un AES qui pourrait encrypter avec les trois tailles de clés explicitées au début de ce document, avec un système de détection de taille de clé par exemple !

CONCLUSION

J'étais dans l'expectative d'apprendre un langage de description matériel comme VHDL, et je dois dire que je n'ai pas été déçu. Le projet semblait de prime abord ambitieux, mais le découpage des séances et la manière dont nous l'avons réalisé (de manière modulaire) a permis la bonne compréhension et la bonne exécution de chaque sous-bloc.

Je termine donc ce projet, satisfait du travail accompli et de mes résultats, qui, je dois le dire, a pu mobiliser beaucoup de ressources dans une période où nous avons eu 5 rapports et 3 livrables de Projet Industriel à rendre ; pas toujours facile donc de consacrer un temps plein à ce projet, mais il était au demeurant plaisant à concevoir. Je suis heureux d'avoir pu enfin comprendre en détail comment le chiffrement symétrique AES fonctionnait : nous ne l'avions que survolé en cours de Sécurité des Systèmes Embarqués, et cette implémentation matérielle m'a permis d'en comprendre toutes les nuances.

Je tiens à remercier M. Dutertre, enseignant du groupe IV, pour sa pédagogie et sa bienveillance, et pour son temps accordé tout au long de nos travaux (notamment ces petites minutes supplémentaires en fin de cours ou encore la séance supplémentaire), du temps primordial pour des élèves lancés dans un projet tel que celui-ci.

Je me félicite des progrès réalisés en VHDL et de la manière dont on nous l'a enseigné, et aborde sereinement un avenir rempli de conception matérielle !

ANNEXES

Annexe 1 :

Code complet disponible [ici](#).

Annexe 2 :

Sources web de l'intro :

<https://www.securiteinfo.com/cryptographie/aes.shtml>

<https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf>

https://en.wikipedia.org/wiki/Advanced_Encryption_Standard

Annexe 3 :

Etapes normales d'un AES vérifié.

A titre d'évaluation du bon fonctionnement de votre développement, les états intermédiaires après chaque étape de calcul de l'AES pour le chiffrement du message

"Es-tu confinée ?" avec la clé " 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c " sont listés ci-dessous.

Plain text : (Hex) 45 73 2d 74 75 20 63 6f 6e 66 69 6e e8 65 20 3f

(ASCII) Es-tu confinée ?

Cipher text at the end: (Hex) cd 30 1f 3e 1f 96 9b 1e 6d 37 d1 79 80 7b 55 b4

(ASCII) í0>m7Ñy{U´

Round 0

State : 45 73 2d 74 75 20 63 6f 6e 66 69 6e e8 65 20 3f

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

Round 1

State after AddRoundKey: 6e 0d 38 62 5d 8e b1 c9 c5 91 7c e6 e1 aa 6f 03

After SBox : 9f d7 07 aa 4c 19 c8 dd a6 81 10 8e f8 ac a8 7b

After ShiftRow : 9f 19 10 7b 4c 81 a8 aa a6 ac 07 dd f8 d7 c8 8e

After MixColumn : 65 e6 2b 45 02 1c 63 b2 62 31 78 fb cf 80 2d 0b

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

Round 2

State after AddRoundKey: c5 1c d5 52 8a 48 4f 03 41 92 41 c2 e5 ec 5b 0e

After SBox : a6 9c 03 00 7e 52 84 7b 83 4f 83 25 d9 ce 39 ab

After ShiftRow : a6 52 83 ab 7e 4f 39 00 83 ce 03 7b d9 9c 84 25

After MixColumn : 89 37 0f 6d 14 ab 43 f4 2c 7a c6 a5 b7 48 39 22

Key state: f2 c2 95 f2 7a 96 b9 43 59 35 80 7a 73 59 f6 7f

Round 3

State after AddRoundKey: 7b f5 9a 9f 6e 3d fa b7 75 4f 46 df c4 11 cf 5d

After SBox : 21 e6 b8 db 9f 27 2d a9 9d 84 5a 9e 1c 82 8a 4c

After ShiftRow : 21 27 5a 4c 9f 84 8a db 9d 82 b8 a9 1c e6 2d 9e

After MixColumn : 3d cd 66 86 e3 d2 62 19 ad f8 94 cf ba 22 19 c8

Key state: 3d 80 47 7d 47 16 fe 3e 1e 23 7e 44 6d 7a 88 3b

Round 4

State after AddRoundKey: 00 4d 21 fb a4 c4 9c 27 b3 db ea 8b d7 58 91 f3

After SBox : 63 e3 fd 0f 49 1c de cc 6d b9 87 3d 0e 6a 81 0d

After ShiftRow : 63 1c 87 0d 49 b9 81 0f 6d 6a fd cc 0e e3 de 3d

After MixColumn : 68 c4 7d 24 cc b7 f8 fd 55 69 a9 a3 c1 97 0d 55

```

Key state: ef 44 a5 41 a8 52 5b 7f b6 71 25 3b db 0b ad 00
Round 5
State after AddRoundKey: 87 80 d8 65 64 e5 a3 82 e3 18 8c 98 1a 9c a0 55
After SBox : 17 cd 61 4d 43 d9 0a 13 11 ad 64 46 a2 de e0 fc
After ShiftRow : 17 d9 64 fc 43 ad e0 4d 11 de 61 13 a2 cd 0a 46
After MixColumn : c6 ee 19 67 c7 74 e2 12 29 06 38 aa 5f 7b b1 b6
Key state: d4 d1 c6 f8 7c 83 9d 87 ca f2 b8 bc 11 f9 15 bc
Round 6
State after AddRoundKey: 12 3f df 9f bb f7 7f 95 e3 f4 80 16 4e 82 a4 0a
After SBox : c9 75 9e db ea 68 d2 2a 11 bf cd 47 2f 13 49 67
After ShiftRow : c9 68 cd 67 ea bf 49 db 11 13 9e 2a 2f 75 d2 47
After MixColumn : 9b 32 89 2b 87 8f b1 7e a3 a4 5b ea 54 ef 2c 58
Key state: 6d 88 a3 7a 11 0b 3e fd db f9 86 41 ca 00 93 fd
Round 7
State after AddRoundKey: f6 ba 2a 51 96 84 8f 83 78 5d dd ab 9e ef bf a5
After SBox : 42 f4 e5 d1 90 5f 73 ec bc 4c c1 62 0b df 08 06
After ShiftRow : 42 5f c1 06 90 4c 08 d1 bc df e5 ec 0b f4 73 62
After MixColumn : a2 a2 8e 54 36 c1 a4 56 10 c1 9d 26 00 0f bf 5e
Key state: 4e 54 f7 0e 5f 5f c9 f3 84 a6 4f b2 4e a6 dc 4f
Round 8
State after AddRoundKey: ec f6 79 5a 69 9e 6d a5 94 67 d2 94 4e a9 63 11
After SBox : ce 42 b6 be f9 0b 3c 06 22 85 b5 22 2f d3 fb 82
After ShiftRow : ce 0b b5 82 f9 85 fb be 22 d3 b6 06 2f 42 3c 22
After MixColumn : ad 9e 29 e8 38 40 48 09 9a 58 8c 0f 86 cd 73 4b
Key state: ea d2 73 21 b5 8d ba d2 31 2b f5 60 7f 8d 29 2f
Round 9
State after AddRoundKey: 47 4c 5a c9 8d cd f2 db ab 73 79 6f f9 40 5a 64
After SBox : a0 29 be dd 5d bd 89 b9 62 8f b6 a8 99 09 be 43
After ShiftRow : a0 bd b6 43 5d 8f be dd 62 09 be b9 99 29 89 a8
After MixColumn : 72 43 af 76 53 5c c9 77 d8 10 dc 78 73 e3 5a 5b
Key state: ac 77 66 f3 19 fa dc 21 28 d1 29 41 57 5c 00 6e
Round 10
State after AddRoundKey: de 34 c9 85 4a a6 15 56 f0 c1 f5 39 24 bf 5a 35
After SBox : 1d 18 dd 97 d6 24 59 b1 8c 78 e6 12 36 08 be 96
After ShiftRow : 1d 24 e6 96 d6 78 be 97 8c 08 dd b1 36 18 59 12
Key state: d0 14 f9 a8 c9 ee 25 89 e1 3f 0c c8 b6 63 0c a6
Cipher text after 10 round: cd 30 1f 3e 1f 96 9b 1e 6d 37 d1 79 80 7b 55
b4

```