Rapport PCSN

Introduction

Le but de ce projet fut de développer l'algorithme de chiffrement symétrique, dit inviolable, nommé AES. L'entièreté de l'algorithme de chiffrement fut développée en VHDL, puis il fut testé sur ModelSim. Cet outil m'a permis de tester les différents blocs fonctionnels de la machine de chiffrement, en étudiant leurs signaux entrants, sortants et internes.

Dans ce rapport, j'ai commenté le développement de chaque bloc et sous-bloc que je devais monter. Chaque commentaire se divise en 2 ou 3 parties :

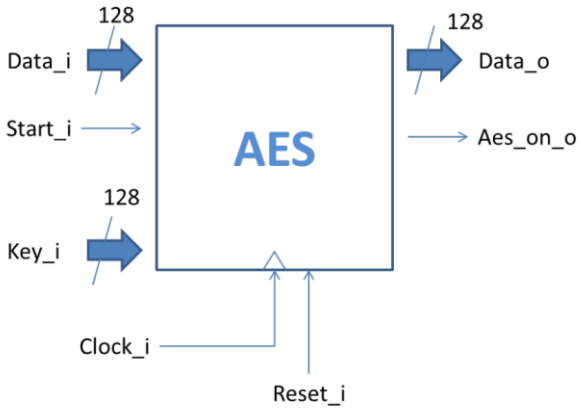
- La description fonctionnelle du bloc (avec comme support, des schémas tirés du sujet ou réalisés sur draw_io).
- La description du code qui concrétise, en langage VHDL, les idées présentées au premier point.
- Enfin, si la simulation du bloc est judicieuse et / ou demandée dans le sujet, la description de la simulation et de ses résultats à l'aide de chronogramme obtenu sur ModelSim.
- J'ai en plus ajouté des commentaires dans les parties qui m'ont posé des problèmes et y ai développé ce que j'en retenais, pour de futurs projets.

Les types de signaux utilisés dans le code, appartiennent soit à la librairie fournie CryptPack (qui permet une structure uniforme entre les projets des élèves), soit à IEEE.

Contents

I. Description de l'AES

1. Architecture globale



Les dimensions des signaux entrant Data_i et sortant Data_o sont fixées à une taille de 128 bits. La taille de la clef Key_i peut varier (128|192|256 bits). Dans ce projet nous avons utilisé une clef de 128 bits.

Fig1 : Entrées/Sorties de l'entité AES

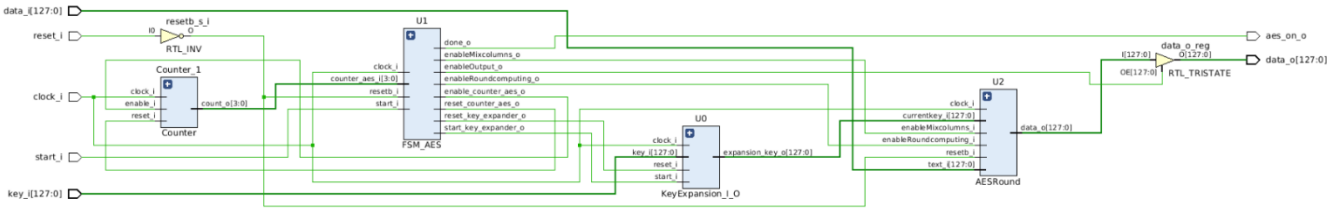


Fig2 : Architecture globale de l'AES

L'algorithme AES se décompose en 4 fonctions élémentaires : AddRoundKey, SubBytes, ShiftRows et MixColumns. L'exécution de celles-ci est coordonnée dans le bloc structurel AESRound. Les blocs Counter et FSM_AES assure le déroulement de l'algorithme en 11 rounds. Au cours de ces rounds, la clef que l'on met en entrée est utilisée par AddRoundKey.

Le bloc structurel KeyExpansion_I_O nous permet de changer cette clef de Round en Round. Pour faire ceci, on passe la clef courante à la fin d'un Round, dans la fonction KeyExpander, avec en entrée la valeur Rcon_i, qui grâce à un Compteur, indique à quel round on en est. Cela influe la façon dont la clef est transformée comme on le verra par la suite. De plus, ce bloc fonctionnel emploie une FSM (KeyExpander_FSM) qui permet au bloc d'agir différemment selon où on se trouve dans l'encryptions (round initial | intermédiaire |final).

Seront expliqués par la suite, le développement des blocs KeyExpansion_I_O et AESRound, la description de leurs composants et le commentaire de leurs tests, en plus du commentaire du test de l'AES.

2. Les Blocs Fonctionnels principaux

a. AESRound

La fonction AESRound est le bloc représentant un Round de chiffrement de l'AES.

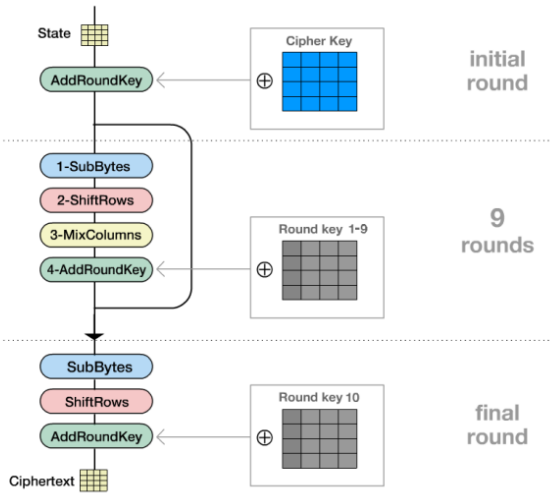


Fig3 : Schéma de déroulé du chiffrement AES 128bit

Ainsi, l'AESRound est appelé 11 fois au cours de l'exécution de l'AES, et à chaque appel, il exécute différentes fonctions selon le Round. Pour être plus précis, au round initial, le message est directement passé dans AddRoundKey, durant les Rounds intermédiaires, le message passe au sein des 4 fonctions, et au dernier, il passe au sein de toutes les fonctions excepté la MixColumns.

i. SBox

Avant de pouvoir parler de la fonction SubBytes, nous devons parler de son composant SBox.

A. *Description de la fonction*

La fonction SBox prend en entrée un octet et renvoie en sortie un octet. Avec l'octet d'entrée (sous forme hexadécimal) on détermine l'octet de substitution grâce à la table de substitution ci-dessous (qui reste inchangée au cours de l'AES) avec les 4 bits de poids fort comme indice X et les 4 bits de poids faible comme indice Y.

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

Fig4 : Table de substitution

B. *Description du code*

Le code de SBox est très direct. Il consiste à, dans un premier temps, déclarer le type de la table de substitution et la table de substitution (sous le nom de sbox_c). Une fois les 256 valeurs entrées (à la main !) dans la table, la seule ligne figurant dans l'architecture est l'assimilation de l'octet de sortie à la valeur indiquée par l'octet d'entrée dans la matrice.

J'aurais eu tendance à dire que le code suivait une modélisation comportementale étant donné qu'aucun composant n'était câblé dans son architecture. Mais comme je n'expliquais pas entièrement le comportement du bloc, cette modélisation ne tenait pas non plus. J'ai alors compris que j'avais à faire ici à une modélisation de type dataflow.

C. ***Test***

Fig5 : Test SBox

Une fois le test-Bench écrit, en injectant la valeur X:53 comme dans la figure 4 nous obtenons bien X :ed en sortie. Le test fut réalisé avec plusieurs entrées, et chaque fois en consultant la table à la main, les sorties étaient bonnes.

ii. SubBytes

A. ***Description de la fonction***

La fonction SubBytes prend en entrée un type_state (matrice 4x4 d'octet) et renvoie un type_state en sortie. Tout ce qu'elle fait est d'appliquer la fonction SBox aux 16 octets de la matrice comme on le voit ci-dessous.

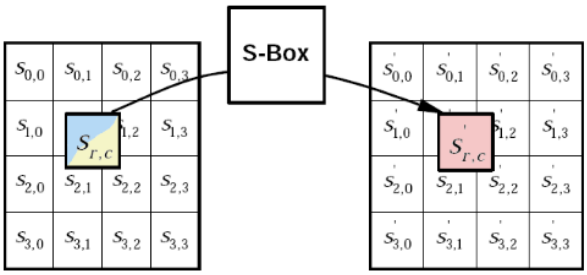


Fig6 : Principe de fonctionnement de SubBytes

B. ***Description du code***

On utilise dans l'architecture de SubBytes 16 composants SBox ce qui en fait une modélisation structurelle. On mappe chaque élément de la matrice d'entrée à un octet d'entrée de SBox et l'octet de sortie est mappé à l'élément (de même indice que celui d'entrée) de la matrice de sortie.

```
22  begin
23      SBox_00 : SBox
24          port map(
25              Sbox_i => Mat_i(0)(0),
26              Sbox_o => Mat_o(0)(0));
27      SBox_01 : SBox
28          port map(
29              Sbox_i => Mat_i(0)(1),
30              Sbox_o => Mat_o(0)(1));
31      SBox_02 : SBox
32          port map(
33              Sbox_i => Mat_i(0)(2),
34              Sbox_o => Mat_o(0)(2));
```

Fig7 : Mapping des composants SBox dans l'architecture de SubBytes

C. ***Test***

Pour le test de SubBytes nous avons pris le message en début de round 1 :

```
Round 1
State after AddRoudkey: 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
```

Tout en faisant attention à entrer les valeurs dans le bon ordre, car en effet pour nos type_state, on entre en ligne alors que les lignes de 256 bits de la partie validation du sujet sont écrites dans l'ordre des colonnes (les 4 premières valeurs représentent donc la première colonne et non pas la première ligne). On aura donc, dans plusieurs tests, besoin de garder cela en tête, pour confirmer le fonctionnement des blocs.

Ainsi on écrit dans le test-bench :

```
Mat_s_in <=   ((X"6e",X"5d",X"c5",X"e1"),(X"0d",X"8e",X"91",X"aa"),(X"38",X"b1",X"7c",X"6f"),(X"6f",X"c9",X"e6",X"03"));
```

On obtient ainsi dans ModelSim :

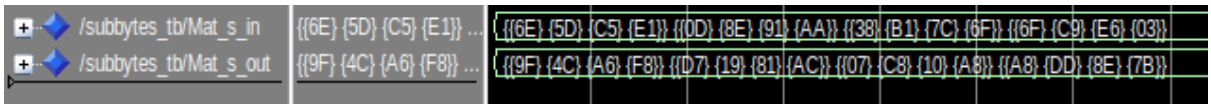


Fig8 : Test SubBytes

Ainsi en comparant les colonnes de Mat_s_out à celles de « After SBox » ci-dessus, nous pouvons valider le fonctionnement de SubBytes.

iii. ShiftRows

A. ***Description de la fonction***

La fonction ShiftRows nous permet d'effectuer une permutation cyclique des octets des lignes d'une matrice 4x4. Le décalage est déterminé par le numéro de la ligne. Donc, le décalage de la ligne 0 sera de 0, celui de la ligne 3 sera de 3, comme montré ci-dessous.

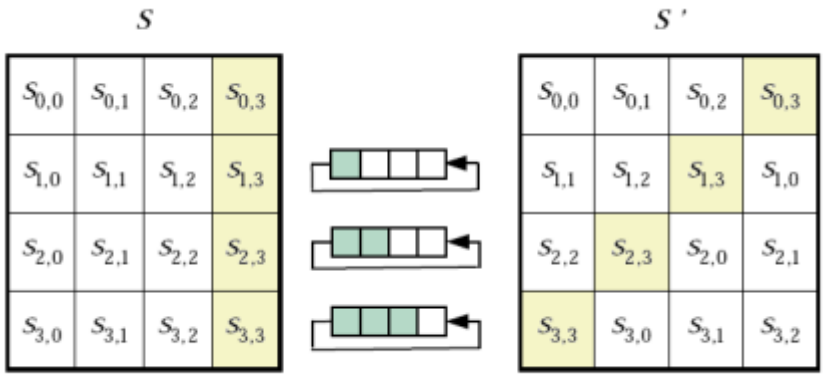


Fig9 : illustration de la fonction ShiftRows

B. ***Description du code***

La fonction ShiftRows prend en entrée une matrice type_state et renvoie en sortie une matrice type_state.

Le code de cette fonction suit une modélisation de type dataflow. Pour chaque octet de la matrice de sortie nous avons spécifié quel élément de la matrice d'entrée y serait injecté. Nous avons généralisé le code en adressant les éléments de la matrice par leurs indices.

C. ***Test***

On reste sur le même round que précédemment :

```
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
```

On garde en tête ce que l'on a dit précédemment sur les lignes et les colonnes, puis on entre dans le test-bench :

```
Mat_s_in <= ((X"9f",X"4c",X"a6",X"f8"),(X"d7",X"19",X"81",X"ac"),(X"07",X"c8",X"10",X"a8"),(X"aa",X"dd",X"8e",X"7b"));
```

Enfin on lance la simulation et on compare à After ShiftRows :

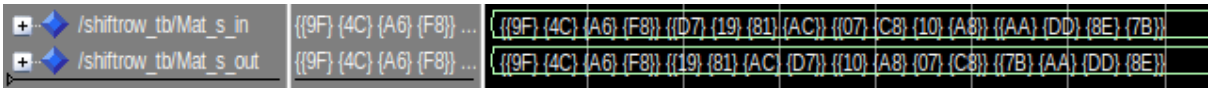


Fig10 : Test ShiftRows

On remarquera qu'il n'est même pas nécessaire de regarder le sujet pour voir que la fonction marche. En effet on peut directement remarquer pour chaque ligne, le décalage entre Mat_s_in et Mat_s_out dans la figure 10.

iv. AddRoundKey

A. ***Description de la fonction***

La fonction AddRoundKey consiste à ajouter la clef du round courant à la matrice de l'état actuel. L'addition se traduit par l'exécution de l'opération ou-exclusif entre chaque élément de même indice des deux matrices.

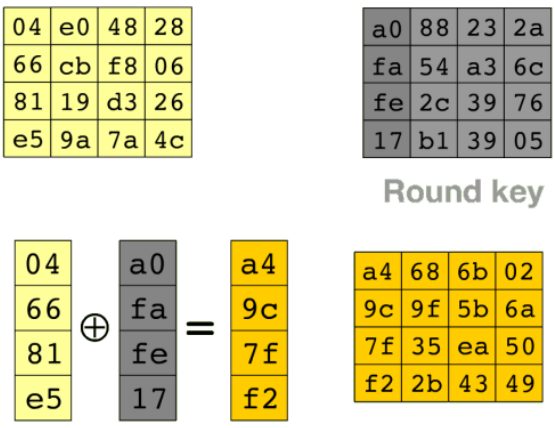


Fig11 : illustration de la fonction AddRoundKey

B. ***Description du code***

Le code de cette fonction prend en entrée la clef du round Key_i et une matrice Mat_i et il renvoie une matrice Mat_o. Toutes sont des type_state

Il suit une modélisation de type dataflow. A l'aide de deux generates imbriqués (deux boucles allant de 0 à 3), on parcourt tous les indices de la matrice de sortie et on y injecte le résultat de l'addition.

C. ***Test***

La fonction AddRoundKey lie les Rounds entre eux ; ainsi en restant sur le Round 1, la fonction nous fait basculer dans le Round 2.

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

Mat_i = After MixColumn
Key_i = Key state
Mat_o = After AddRoundKey

On entre les valeurs dans le test-bench en faisant attention aux colonnes et aux lignes.

```
Key_s_in <= ((X"A0",X"88",X"23",X"2A"),(X"FA",X"54",X"A3",X"6C"),(X"FE",X"2C",X"39",X"76"),(X"17",X"B1",X"39",X"05"));
Mat_s_in <= ((X"65",X"02",X"62",X"cf"),(X"e6",X"1c",X"31",X"80"),(X"2b",X"63",X"78",X"2d"),(X"45",X"b2",X"fb",X"0b"));
```

On lance la simulation et on compare la sortie à After AddRoundKey.

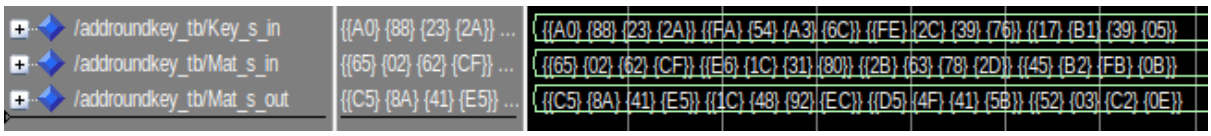


Fig12 : Test AddRoundKey

v. *MixColumns*

A. ***Description de la fonction***

La fonction MixColumns consiste à effectuer un produit matriciel de chacune de ces colonnes avec une matrice. La matrice en question est donnée ci-contre : $\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$

Le produit matriciel se défini comme suit :

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

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

Fig13 : Produit matriciel de la fonction MixColumns

Quand on multiplie un octet par {03}, ça revient à le multiplier par {01} + {02}. Quand on le multiplie par {02}, si son bit de poids fort est nul, cela revient à décaler chaque bit à gauche. Par contre si le bit de poids fort vaut 1, alors on effectue un décalage à gauche des bits et on prend le résultat d'un ou-exclusif entre l'octet décalé et la valeur X:1b.

B. Description du code

Le code de MixColumns prend en entrée et renvoie en sortie une matrice type_state. On a en plus de cela un signal interne sx_2_s type_state.

La structure est de type dataflow étant donné que nous nous servons de boucles et d'adressage indiciel des éléments des matrices (il serait inconcevable de créer une structure comportementale vu le nombre immense d'entrées possibles).

L'architecture se sépare en deux parties. Dans un premier temps, on injecte dans sx_2_s le résultat du produit de chaque élément de la matrice d'entrée par {02}. Ceci est fait grâce à deux generates imbriqués pour parcourir la matrice d'entrée, et une condition when permettant de distinguer les cas d'octet à bit de poids fort nul, des autres.

Une fois la matrice sx_2_s remplie, on parcourt les 4 colonnes de la matrice d'entrée grâce à un generate et on injecte dans la matrice de sortie les opérations vues dans la figure 13.

Remarque/ Problèmes rencontrés :

J'ai eu du mal à comprendre la façon dont il fallait lire les lignes et les colonnes sur le sujet et ai longtemps eu du mal à obtenir un bon résultat. N'ayant pas compris que les états sur le sujet se lisaient en colonnes, mon MixColumns renvoyait pendant longtemps la transposée des matrices que je voulais, sans que je m'en aperçoive. Je voyais bien que quelque chose ne collait pas au niveau des indices, mais ce n'est que lorsque j'ai programmé le convertisseur bit128 -> type_state utilisé dans AESRound, que je compris mon erreur.

Si je devais refaire un projet de ce type, je prendrais plus de temps pour m'assurer que je comprenne toutes les conventions et notations du sujet, avant de me lancer dans la programmation.

C. Test

On test la fonction en se plaçant dans le Round 1 :

```
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
```

On écrit dans le test-bench :

```
Mat_s_in <= ((X"9f",X"4c",X"a6",X"f8"),(X"19",X"81",X"ac",X"d7"),(X"10",X"a8",X"07",X"c8"),(X"7b",X"aa",X"dd",X"8e"));
```

On lance la simulation et on compare la sortie à After MixColumn :

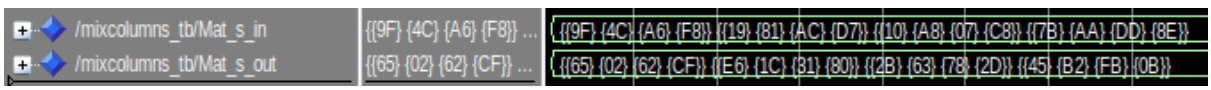


Fig14 : Test Mixcolumn

vi. AESRound

A. ***Description de la fonction***

La fonction AESRound a pour but de réaliser un Round de l'AES. Ainsi elle prend toutes les fonctions décrites précédemment comme composant et un signal d'entrée qu'elle fait passer dans les différents composants. Il est important de garder en tête que selon le round, le signal ne subit pas le même enchainement de fonctions.

B. ***Description du code***

AESRound prend en entrée : les bits clock_i, resetb _i, enableMixColumns_i, et enableRoundComputing_i et les bit128 currentkey_i et Message_i. Elle renvoie le bit128 Message_o.

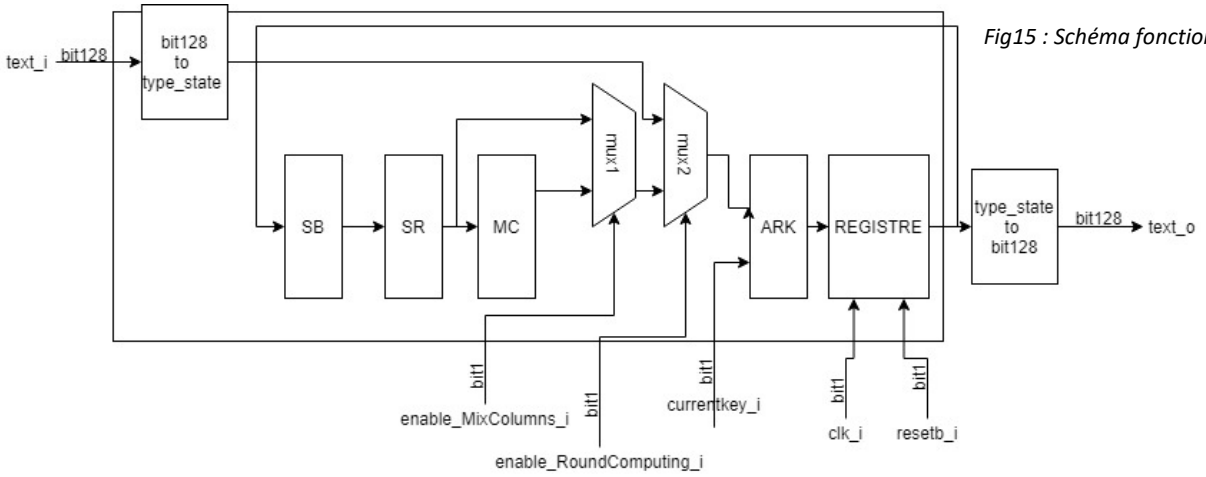


Fig15 : Schéma fonctionnel de AESRound

AESRound est codé avec les composants des 4 fonctions vues jusqu'ici (SBox n'est pas compté). Ainsi, nous l'avons développé avec une modélisation structurelle.

Tout d'abord, sachant que les textes/clef d'entrée et de sortie sont en bit128, et que toutes nos fonctions prennent des type_state, nous avons conçu des convertisseurs pouvant faire les conversions dans les deux sens.

Nous voyons ci-dessus, l'enchainement des trois fonctions SubBytes, ShiftRows et Mixcolumns que l'on voit dans la figure 3, pour les 9 Rounds intermédiaires. Nous introduisont ainsi des signaux internes permettant de passer le texte After SubBytes dans le ShiftRows puis le After ShiftRows dans le MixColumns.

Puis nous avons 2 multiplexeurs. Leur but est de prendre en compte les 2 bits enableMixColumns_i, et enableRoundComputing_i. Le but de enableMixColumns_i est d'indiquer si on prend en compte ou non le bloc MC (pour 1 on le prend en compte, 0 non) et enableRoundComputing_i sert à indiquer si on se trouve au premier Round ou non. Si oui (en_RC = 0) alors on passe directement text_i dans ARK en ignorant les 3 autres blocs. Si les 2 bits valent 1 on est dans les Rounds intermédiaires et on exécute toutes les fonctions dans l'ordre vu figure 3. Ces deux multiplexeurs ont été réalisés avec de simples conditions when.

	Round initial	Round intermédiaire	Round final
enableRoundComputing_i	0	1	1
enableMixColumns_i	1	1	0

Fig16 : évolution des bits enable au cours de Rounds

Le registre après le ARK sert d'abord, à initialiser un signal intermédiaire type_state interne lorsqu'on se trouve au Round 0. Par la suite, à chaque front montant de clock_i, le registre injecte dans ce dernier la sortie de la fonction AddRoundKey. La fonction SubBytes prend en entrée la sortie de ce registre. C'est cette rétroaction qui permettra par la suite d'enchainer les Rounds.

Enfin la sortie du registre passe dans un convertisseur et nous obtenons notre message de sortie en bit128.

C. ***Test***

Pour le round 0 on a un test bench avec les valeurs suivantes :

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

```
--Test round 0
mess_s_in <= X"45732d747520636f6e66696ee865203f";
currk_i <= X"2b7e151628aed2a6abf7158809cf4f3c";
ARK_en <= '0';
MC_en <= '1';
```

```
rstb_i <= '1';
clk_i <= '0', '1' after 10 ns;
```

La simulation nous donne :

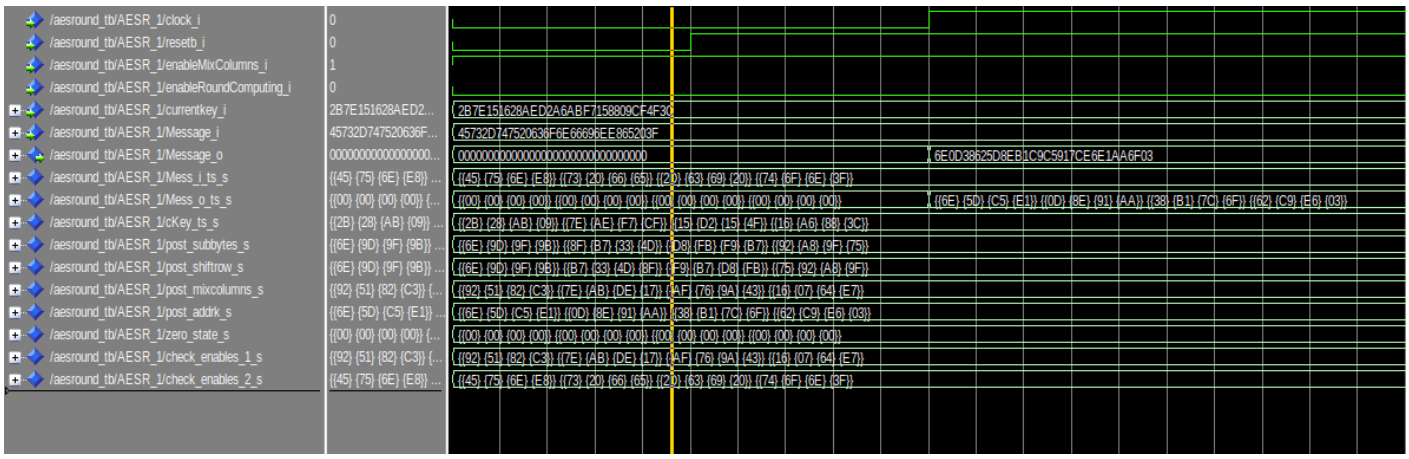


Fig17 : Test AESRound Round 0

On constate qu'en sortie de check_enables_2_s (le deuxième multiplexeur) on retrouve le signal d'entrée Message_i. Cela fait preuve du bon fonctionnement des multiplexeurs pour le Round 0.

On a bien Mess_o_ts_s = zero_state_s quand resteb_i = 0.

On voit également que post_addrk_s contient dès la première nano seconde, la valeur de After AddRoundKey. Cette valeur n'est injectée dans Mess_o_ts_s qu'après le front montant de clock_i ce qui confirme le bon fonctionnement du registre.

Pour les Rounds 1 à 10 :

La façon dont AESRound est écrite au moment présent, fait qu'il n'est pas possible d'effectuer un test pour les Rounds succédant le Round 0. Ce qui pose un problème est la rétroaction allant de la sortie du registre à l'entrée de SubBytes. Elle fonctionne quand on lance l'AES (et est indispensable) car le signal interne Mess_o_ts_s conserve la valeur du Round précédant comme expliqué plus tôt. Mais quand on veut effectuer le test pour un Round intermédiaire (ou le dernier Round) le signal Mess_o_ts_s ne contient pas cette information cruciale pour le chiffrement.

Afin de quand même pouvoir effectuer les tests, on a modifié temporairement le code. On envoie directement dans SubBytes Mess_i_ts_s (comme si Mess_o_ts_s contenait bien la valeur After AddRoundKey du Round précédent et l'avait injecté dans SubBytes).

Cela étant fait, nous avons effectué deux tests :

- Round 1 :

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

```
--Test round 1
mess_s_in <= X"6e0d38625d8eb1c9c5917ce6e1aa6f03";
currk_i <=   X"a0fafe1788542cb123a339392a6c7605";
ARK_en <= '1';
MC_en <= '1';
```

```
rstb_i <= '1';
clk_i <= '0', '1' after 10 ns;
```

La simulation nous donne :

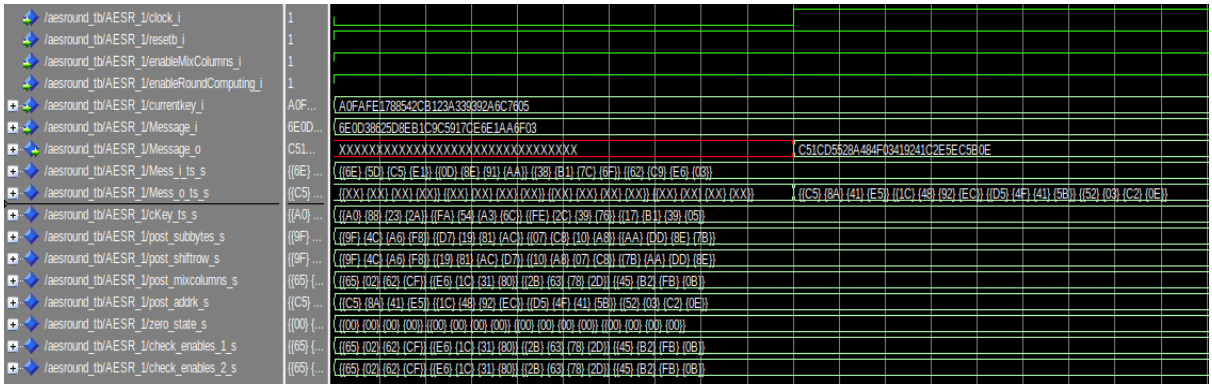


Fig18 : Test AESRound Round 1

On constate qu'en sortie de check_enables_2_s on retrouve le signal de sortie de MixColumns. Cela est la preuve du bon fonctionnement des multiplexeurs pour les Round 1 à 9.

On constate que la sortie correspond bien à l'entrée du Round 2. De plus on peut préciser que si ces signaux étaient tirés d'une simulation de l'AES, avant le front montant, le signal Mess_o_ts_s contiendrait la valeur After AddRoundKey du Round 0.

- Dernier Round :

```
Round 10
State after AddRoudkey: 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
```

```
--Test last round
mess_s_in <= X"de34c9854aa61556f0c1f53924bf5a35";
currk_i <=   X"d014f9a8c9ee2589e13f0cc8b6630ca6";
ARK_en <= '1';
MC_en <= '0';
```

```
rstb_i <= '1';
clk_i <= '0', '1' after 10 ns;
```

La simulation nous donne :

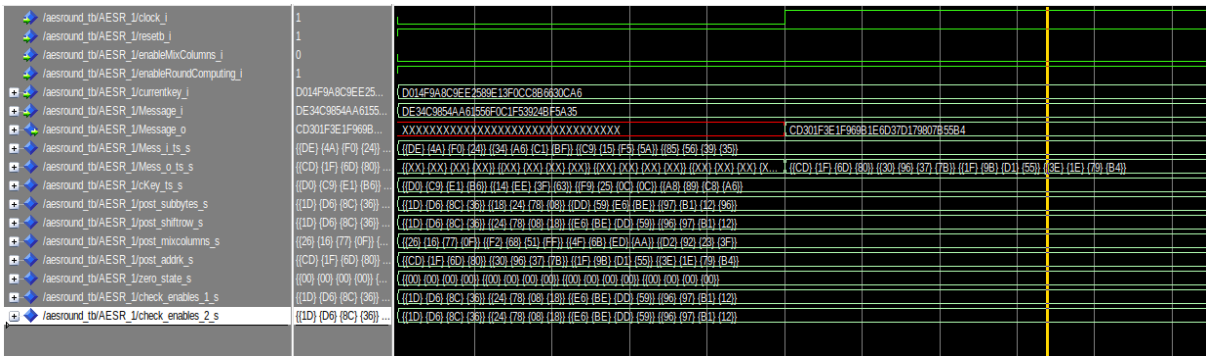
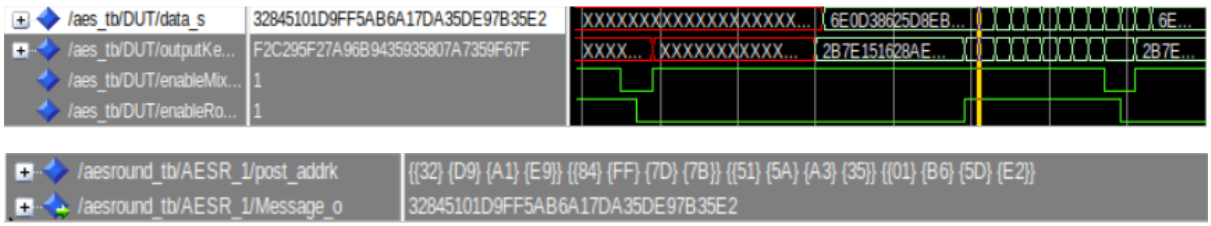


Fig19 : Test AESRound Round 10

On constate qu'en sortie de check_enables_2_s on retrouve le signal de sortie de ShiftRow impliquant que MixColumns a bien été sauté pour ce dernier Round. Cela fait preuve du bon fonctionnement des multiplexeurs pour le Round 10.

<u>Remarque/ Problèmes rencontrés :</u>

Initialement, je pensais avoir un AESRound fonctionnel car il renvoyait les bons messages pour tous les Rounds. C'est quand je lançai la simulation de l'AES que j'ai constaté qu'à chaque Round, au lieu de récupérer le message du Round précédent et de le transformer correctement, je reprenais le message d'entrée. J'ai confirmé ce doute en effectuant un test avec AESRound :



Au-dessus on a la simulation du AES défectueux, et en dessous la simulation de AESRound avec la clef du Round 1, le message d'entrée du Round 0 et les deux bits enable à 1. On voit donc bien que chaque Round récupère le message d'entrée.

Ça n'est qu'après avoir compris ce problème, qu'on m'expliqua que je n'avais pas fait la rétroaction du signal de sortie à l'entrée de SubBytes. J'y mettais à la place, le signal contenant l'information du text_i. L'erreur était évidente mais je ne l'avais pourtant pas vue.

En dessinant le schéma fonctionnel de l'AESRound, je compris l'importance de ce genre de travail précédant la programmation, car avec ce dernier (figure 15) sous les yeux, je n'aurais pas pu me tromper au niveau des entrées des blocs et aurais économisé plus de temps que je pensais gagner à ne pas le faire.

b. KeyExpander_I_O

i. KeyExpander

A. Description de la fonction

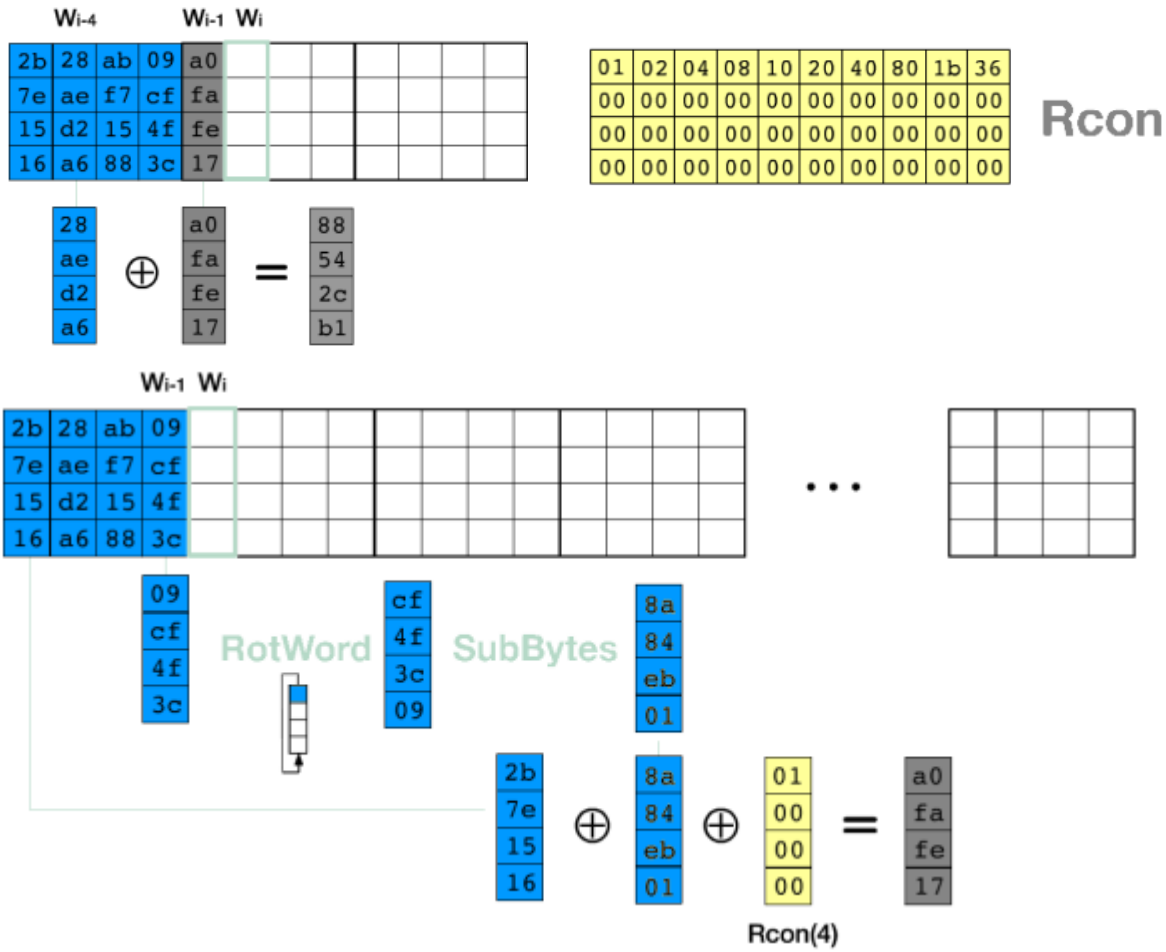


Fig20 : Fonctionnement de KeyExpander

La fonction KeyExpander nous permet entre chaque Round de l'AES de transformer la clef courante. Pour se faire, on s'y prend en deux temps.

Pour la première colonne de la nouvelle matrice clef, on exécute une rotation vers le haut de la dernière colonne de la clef actuelle. Puis on injecte ses 4 éléments dans 4 instances SBox. Enfin on

récupère le résultat de la somme (xor) entre cette colonne, la première colonne de la clef courante et une colonne tirée de la matrice Rcon que l'on voit ci-dessus.

Pour les trois colonnes restantes, on récupère le résultat du xor entre la colonne i de l'ancienne clef et la colonne i-1 de la nouvelle clef (cela nous donne la colonne i de la nouvelle clef).

B. ***Description du code***

Utilisant des composants, nous avons à faire à une modélisation structurelle. La fonction prend en entrée la clef actuelle et renvoie la nouvelle clef en bit128. De plus elle prend en entrée Rcon_i en bit8. Ce-dernier nous permet d'utiliser la colonne de Rcon qu'il nous faut. Nous verrons par la suite comment on fait en sorte d'avoir la bonne valeur de Rcon en entrée.

Pour faciliter les additions entre les colonnes, sachant qu'il nous est fournie une fonction xor pour colonnes dans le cryptpack, nous introduirons un type qui est une liste de 4 column_state.

Nous avons utilisé un convertisseur bit128 à type_key_coulunm_ss (le type introduit ci-dessus). Une fois la clef courante convertie dans un signal intermédiaire, nous avons exécuté les opérations décrites précédemment, pour former la première colonne de la nouvelle clef. La rotation et le SBox sont effectués en même temps grâce à un décalage d'indice pour chaque élément de la colonne.

Puis, grâce à une boucle, on a récupéré les trois autres colonnes.

Enfin, nous effectuons la conversion inverse à celle du début.

C. ***Test***

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

On se place au Round 0. On prend l'élément de Rcon de même indice que le Round, on lit dans cryptpack la valeur : X :01. On met la clef courante en entrée dans le test-bench.

```
ck_s_i <= X"2b7e151628aed2a6abf7158809cf4f3c";
rcon_s_i <= X"01";
```

La simulation nous donne :

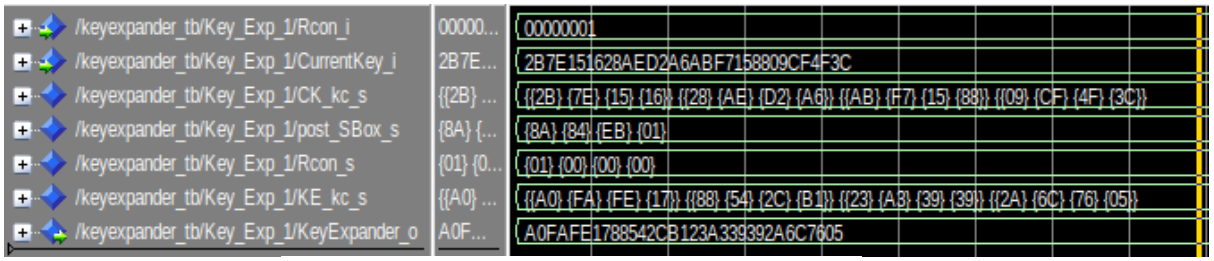


Fig21 : Test KeyExpander

On retrouve bien la clef du Round 1. Cette fonction n'est pas sensée agir différemment selon les Rounds, donc si elle marche pour le 0, elle marche pour les autres.

ii. KeyExpander_FSM

*A. **Description de la fonction***

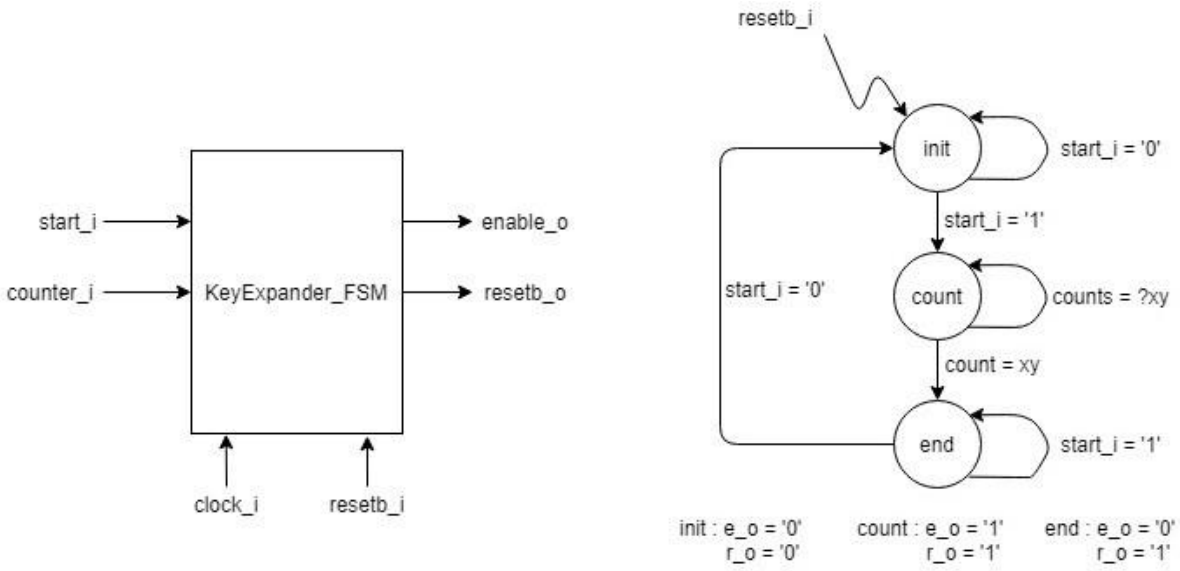


Fig22 : Schéma bloc/fonctionnel de la machine d'état KeyExpander _FSM

Cette fonction est une simple machine de Moore. Elle a pour but de renvoyer des sorties correspondant à son état présent. On voit sous le diagramme les sorties renvoyées selon l'état courant. Cette machine nous sera indispensable pour la conception de KeyExpander_I_O.

*B. **Description du code***

On a défini l'état présent et futur de la machine comme des signaux state.

Un premier process P0 nous permet (si resetb_i vaut 0) d'initialiser la machine et (sur les fronts montant de la clock_i) de faire passer l'état présent à l'état suivant.

Un deuxième process C0 nous permet selon l'état présent, le compteur et le bit start_i de déterminer l'état futur. L'écriture des différents cas est rendue très facile grâce au case.

Enfin un troisième process C1 nous sert à déduire de l'état présent, les sorties que la machine doit nous renvoyer.

Nous avons donc à faire à une structure de type dataflow.

iii. KeyExpander_I_O

A. ***Description de la fonction***

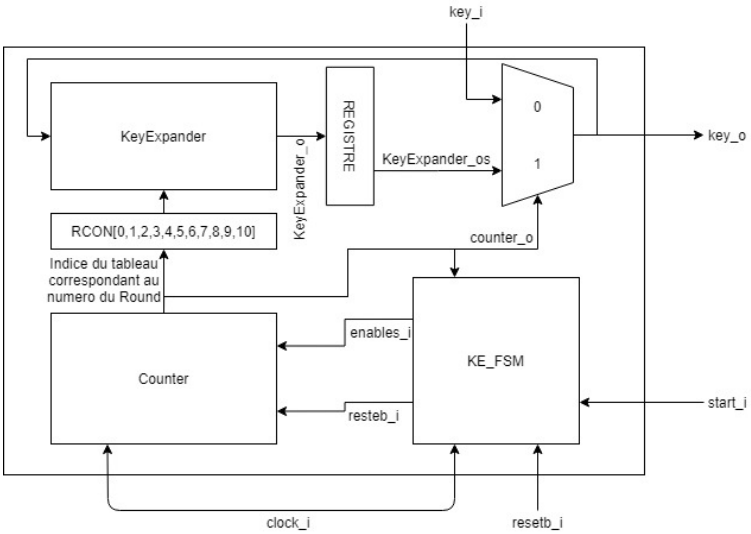


Fig23 : Schéma fonctionnel de KeyExpander_I _O

La fonction KeyExpander_I_O a pour but de permettre la transformation de la clef (grâce à KeyExpander) entre chaque round afin que le chiffrement soit mené à bien.

Pour savoir à quel Round on se trouve et donc quel élément de Rcon on utilise dans KeyExpander, on aura recours à une instance de compteur.

L'instance de la FSM décrite précédemment sera utilisée pour fournir les entrées du compteur, qui varient selon où on se trouve dans le processus de l'AES.

Pour retenir la sortie de KeyExpander d'un Round à l'autre, une instance de registre est utilisée.

Pour pouvoir initialiser le processus avec la clef d'entrée key_i, un multiplexeur nous permettra avec la sortie du compteur de déterminer s'il faut prendre cette clef (compteur = '0000') ou celle stockée dans le registre de sortie (compteur ! = '0000').

Enfin pour pouvoir transformer la clef du Round i-1 pour le Round i, un boucle de rétroaction est faite de la sortie de ce multiplexeur à l'entrée de l'instance de KeyExpander.

B. ***Description du code***

Toutes les fonctions utilisées dans KeyExpander_I_O sont appelées sous forme de composants ce qui en fait une modélisation structurelle.

Pour déterminer quel indice de Rcon doit être utilisé dans KeyExp (KeyExpander) on a écrit un processus P1 qui détermine en fonction de counter_s (un signale interne dans lequel on stock la sortie du compteur) et de resetb_i la valeur de l'indice.

Le registre en sortie de KeyExp est un processus nommer P0, qui en fonction de resetb_i, enable_s (sortie de la FSM), clock_i et key_expander_s (sortie de KeyExp) s'initialise (resetb_i = 1), prend la valeur de sortie de KeyExp sur un front montant (si enable_s = 1) ou ne change pas de valeur.

Le multiplexeur de sortie est simplement écrit avec une condition when counter_s = '0000'.

Enfin la boucle de rétroaction est réalisée en mettant en entrée de KeyExp la valeur en sortie du multiplexeur.

II. Test de l'AES

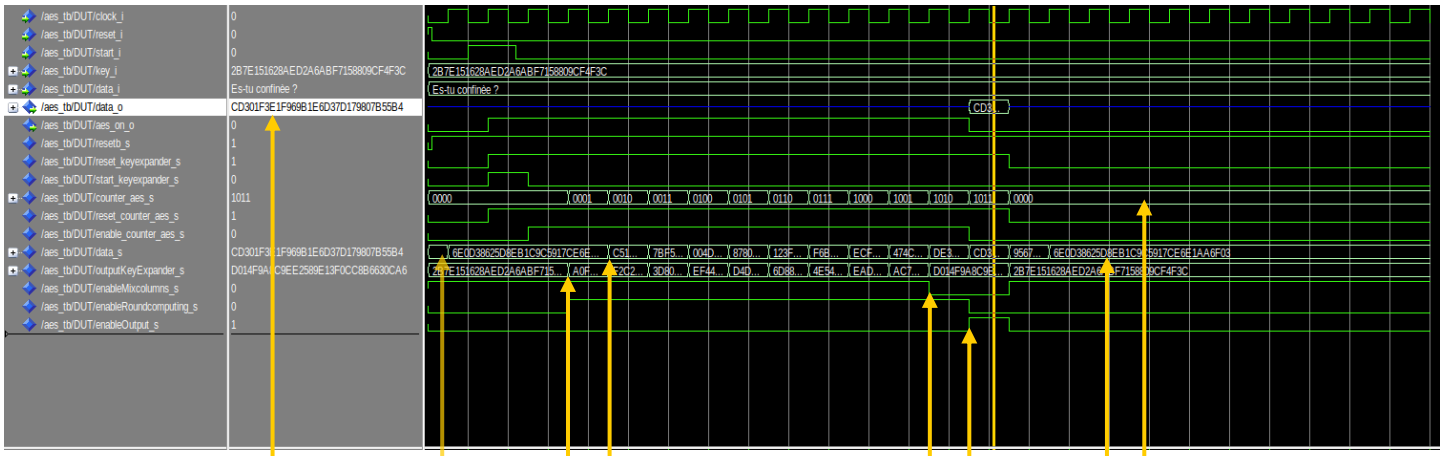
```
Plain text : (Hex) 45 73 2d 74 75 20 63 6f 6e 66 69 6e e8 65 20 3f
 (ASCII) Es-tu confinèe ?

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

On rentre dans le test-bench le texte fourni dans le sujet

```
data_is <= X"45732d747520636f6e66696ee865203f";
key_is  <= X"2b7e151628aed2a6abf7158809cf4f3c";
```

```
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´
```



On peut commencer par constater que l'on obtient le bon message chiffré.

On voit que la sortie de l'AESRound est bien à 0 avant le premier front montant due au reset_i nul durant les 10 premières ns.

On constate que le multiplexeur en sortie du KE_I_O marche bien car dès que l'on passe de '0000' à '0001', la clef change.

On constate que les multiplexeurs d'AESRound fonctionnent bien, et que dès le premier front montant de l'horloge après qu'enableRoundcomputing passe à 1, la sortie d'AESRound est égale à After AddRoundKey du Round 1.

En arrivant au dernier Round, enableMixColumns passe à 0, comme voulu.

Le multiplexeur de sortie de l'AES fonctionne bien car data_o ne s'affiche que lorsqu' enableOutputs_s passe à 1.

Le compteur repasse a '0000'

On revient à l'état initial avec les deux enables, ainsi on retrouve la sortie du AESRound pour le Round 0.

Conclusion

Le développement d'une machine AES et la rédaction de ce rapport m'ont apporté beaucoup en termes de connaissances et surtout de savoir-faire. Que je continue dans la voie des systèmes embarqués et de la microélectronique ou non, ne dévalue pas l'expérience apportée par ce projet.

Ce fut très intéressant de travailler sur un sujet avec autant de facettes. Je me suis surpris à très souvent sortir du code d'un bloc pour regarder l'ensemble de la machine, afin de comprendre exactement ce que je faisais. Ce recul à prendre, me semble essentiel à tout projet. S'il est difficile de synthétiser le projet dans une vue d'ensemble, il faut cesser le micro-développement et repasser à l'échelle macro pour clarifier le tout.

Les difficultés auxquelles j'ai fait face ont mis en évidence des défauts dans mes méthodes de travail (notamment par rapport au point adressé ci-dessus). Le blocage qu'a causé mon AESRound défectueux, par exemple, aurait pu être évité grâce à un simple schéma fonctionnel du bloc. Mais le temps que j'ai pris à débugger n'était pas perdu ; il m'a poussé à réaliser un schéma pour le Key_Expansion_I_O avant de passer à l'implémentation en VHDL, ce qui a grandement facilité et accéléré celle-ci.

Ce que je retiens de ce projet est l'importance de la préparation et de l'anticipation. Le langage qu'on utilise et les simulateurs restent tous de simples outils, nous permettant d'atteindre un but précis. Mais la répartition du temps de travail doit être équilibrée entre la définition précise du projet, le travail d'anticipation (comme les schémas) et la concrétisation du tout. Sans cette répartition, il y a un risque à se retrouver bloqué sans même savoir d'où le problème peut venir. Ainsi, dans le futur je ferai en sorte de prendre plus de temps pour analyser et comprendre les sujets avant de foncer tête baissé dans la pratique.