RAPPORT VHDL :

Chiffrement AES

Conception d'un système numérique

Arthur DRIANT

EI-19

Sommaire

Introduction 4

I- L'AES round 5

A- **SubBytes** 5

1- Composant SBox 5

2- Composant SubBytes 6

B- **ShiftRows** 7

1- Modélisation 8

2- Test bench 8

C- **MixColumns** 9

1- Modélisation 9

2- Test bench 11

D- **AddRoundKey** 11

1- Implémentation 12

2- Test bench 13

E- **AESRound** 13

1- Implémentation 14

2- Simulation 14

II – KeyExpansion 16

A- **KeyExpander** 16

1- Implémentation 16

2- Test bench 17

B- **KeyExpander_FSM** 18

1- Modélisation 18

2- Test bench 19

C- **KeyExpander_IO** 20

1- Modélisation 20

2 – Test bench 21

III – Top level, simulation finale 22

A- **FSM_AES** 22

1- Description 22

2- Test bench 24

B- **Toplevel** 25

1- Description 25

2- Simulation 25

Retours sur le projet 27

Table des figures

Figure 1 - Principe du chiffrement AES....4
Figure 2 - Tableau de substitution Sbox....5
Figure 3 - Ports de la SBox....6
Figure 4 - Résultats du test bench de la SBox....6
Figure 5 - Ports de SubBytes....7
Figure 6 - Résultats du test bench de SubBytes....7
Figure 7 - Illustration du ShiftRows....7
Figure 8 - Ports de ShiftRows....8
Figure 9 - Résultats du test bench de ShiftRows....8
Figure 10 – Produit matriciel de MixColumns....9
Figure 11 - Fonction mult2....9
Figure 12 - Fonction mult3....10
Figure 13 - Ports de MixColumns....10
Figure 14 - MUX à l'intérieur du bloc MixColumns....10
Figure 15 - Exemple du sujet pour le round 1....11
Figure 16 - Résultats du test bench de MixColumns....11
Figure 17 - Calcul de la fonction AddRoundKey....11
Figure 18 - Ports de AddRoundKey....12
Figure 19 - Fonction xor_ligne....12
Figure 20 - Exemple tiré du sujet....13
Figure 21 - Résultats du test bench d'AddRoundkey....13
Figure 22 - Schéma de l'AESround....14
Figure 23 - Exemple du sujet....14
Figure 24 - Résultats du test bench pour le round 0....15
Figure 25 - Round 1 tiré du sujet....15
Figure 26 - Résultats du test bench pour le round 1....15
Figure 27 - Ports du bloc keyexpander....16
Figure 28 - Création de la première colonne de la nouvelle clé....17
Figure 29 - Création des 3 autres colonnes de la nouvelle clé....17
Figure 30 - Résultat du test bench de la génération de la clé du round 1....17
Figure 31 - Ports du bloc KeyExpander_FSM....18
Figure 32 - Machine d'état keyexpander_FSM....19
Figure 33 - Résultat du test bench de la FSM du KeyExpander....19
Figure 34 - Ports de Keyexpander_IO....20
Figure 35 - Schéma du Keyexpander_IO....20
Figure 36 - Résultat du test bench de keyexpander_IO....21
Figure 37 - Ports de la FSM_AES....22
Figure 38 - Graphe d'état de la FSM_AES....23
Figure 39 - Sorties de chaque état....23
Figure 40 - Résultats du test bench de la FSM_AES....24
Figure 41 - Schéma du Toplevel de l'AES....25
Figure 42 - Résultats du test bench du toplevel initial....25
Figure 43 - Résultats du test bench du top level modifié....26

Introduction

Le but de ce projet est d'implémenter la méthode de chiffrement AES en VHDL. Ce dernier est structuré en sous blocs pour faciliter sa compréhension.

Le principe du chiffrement AES est le suivant :

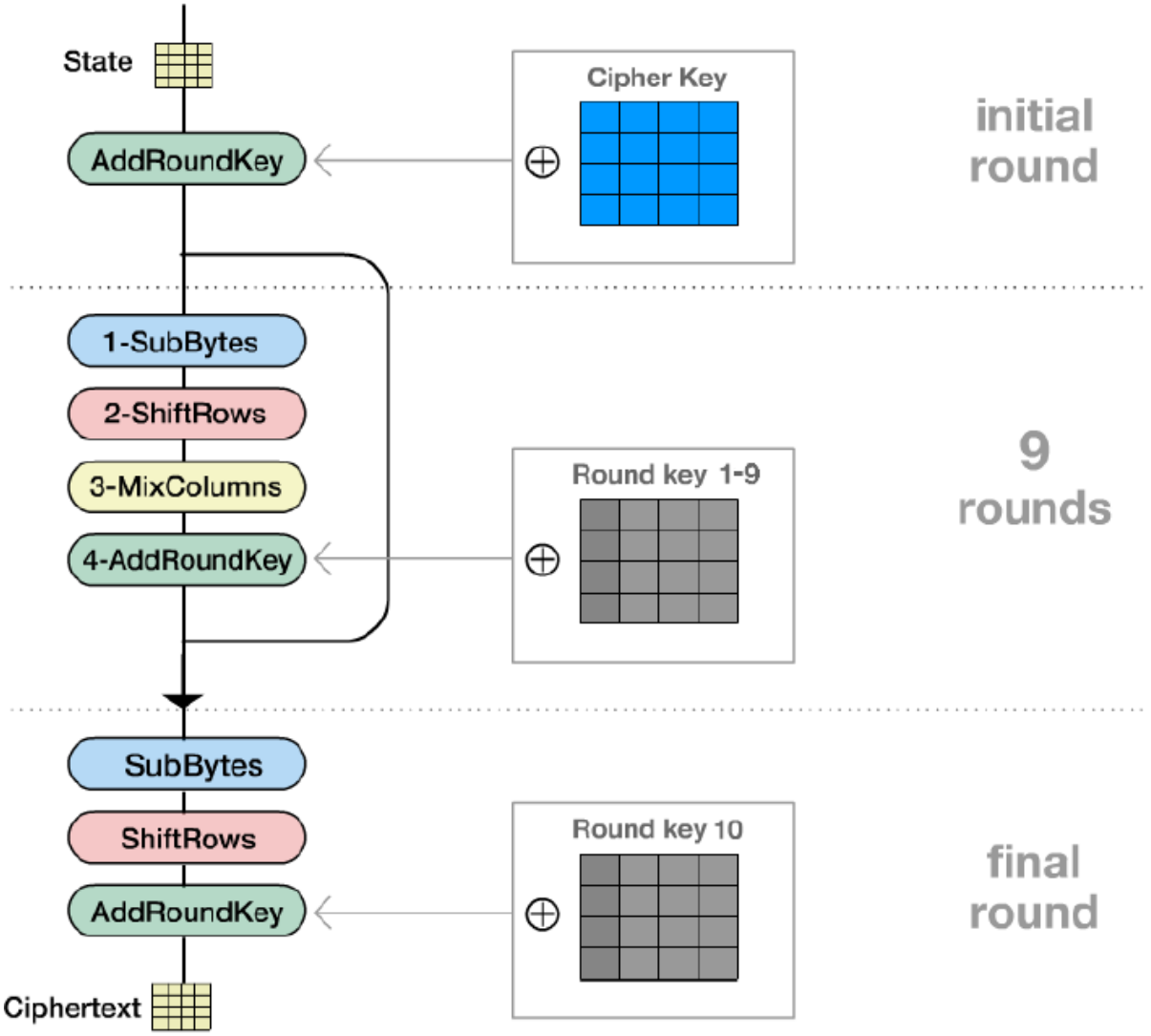

Figure 1 - Principe du chiffrement AES

À travers ce rapport, nous étudierons tous ces différents blocs en expliquant leur fonctionnement puis en les simulant grâce à ModelSim.

I- L'AES round

Le bloc AES round est découpé en 4 étapes : SubBytes, ShiftRows, MixColumns et enfin AddRoundKey. Dans cette partie, nous allons étudier chaque étape en détail.

A- SubBytes

Cette première fonction a pour rôle de substituer chaque valeur en entrée par sa valeur correspondante dans un tableau de substitution appelé SBox. Cette transformation va s'effectuer au sein d'un composant du même nom.

1- Composant SBox

a- Modélisation

SBox possède une entrée de type bit8 et une sortie du même type. La valeur de sortie correspond à la valeur du tableau SBox à l'indice égal à l'entrée du circuit. Ce tableau est standardisé ce qui nous permet de l'écrire en dur dans un fichier.

		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 2 - Tableau de substitution Sbox

Ainsi, si l'entrée est égale à X''1F'', la sortie sera donc égale à SBox(1f), c'est-à-dire X''C0''.

On peut également représenter la Sbox avec ses ports de la manière suivante :

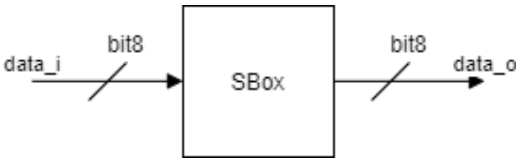

Figure 3 - Ports de la SBox

b- Test bench

On vérifie que notre SBox fonctionne :

Figure 4 - Résultats du test bench de la SBox

Les résultats sont bien ceux attendus, le composant est donc fonctionnel !

2- Composant SubBytes

Le composant SubBytes effectue la substitution SBox sur toute une matrice de taille 4*4 d'éléments de type bit8.

a- Modélisation

Pour réaliser ce composant, il suffit d'initialiser 16 SBox et d'y insérer toutes les valeurs de la matrice. Ce résultat est stocké dans une matrice intermédiaire puis est envoyé sur la sortie. Cette méthode est la plus efficace car l'étape est réalisée en un coup d'horloge puisque toutes les actions se déroulent en parallèle.

Figure 5 - Ports de SubBytes

b- Test bench

Pour effectuer notre simulation, nous allons prendre le message « Es-tu confiné ? » en hexadécimal :

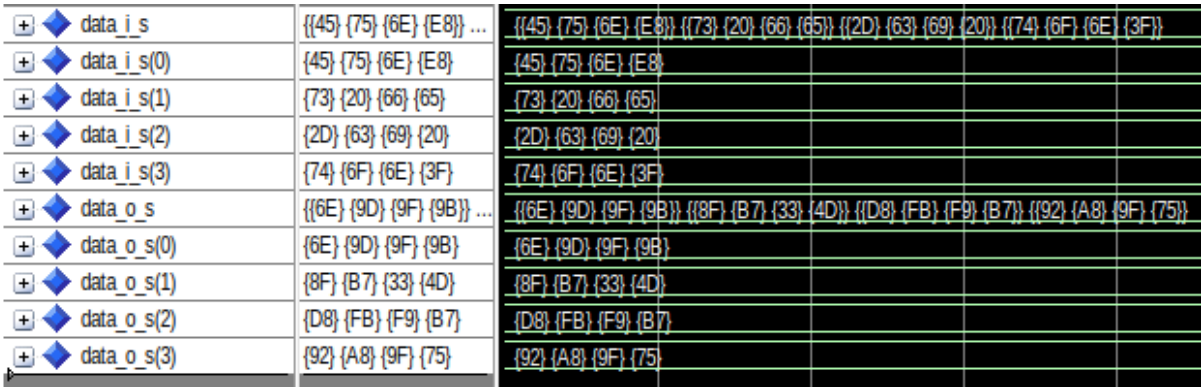

Figure 6 - Résultats du test bench de SubBytes

Les résultats sont bien ceux attendus, le composant est donc fonctionnel !

B- ShiftRows

Cette fonction a pour but d'effectuer un décalage sur chaque ligne de la matrice en entrée. La première ligne n'est pas décalée, la deuxième subit un décalage de 1 vers la gauche, la troisième subit un décalage de 2 vers la gauche et la quatrième ligne subit-elle un décalage de 3 vers la gauche également.

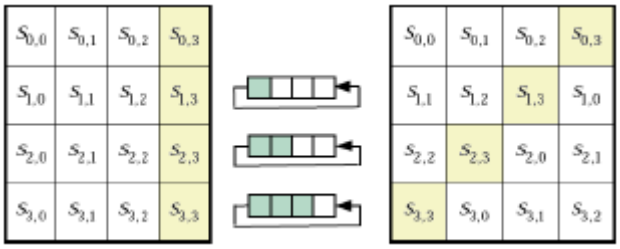

Figure 7 - Illustration du ShiftRows

1- Modélisation

Pour modéliser ce composant, il suffit de réaffecter chaque valeur dans une matrice intermédiaire en respectant les décalages puis d'envoyer cette matrice sur la sortie.

Figure 8 - Ports de ShiftRows

2- Test bench

Pour effectuer notre simulation, nous allons utiliser une matrice avec 4 lignes identiques (1, 2, 3, 4) afin de bien voir le décalage :

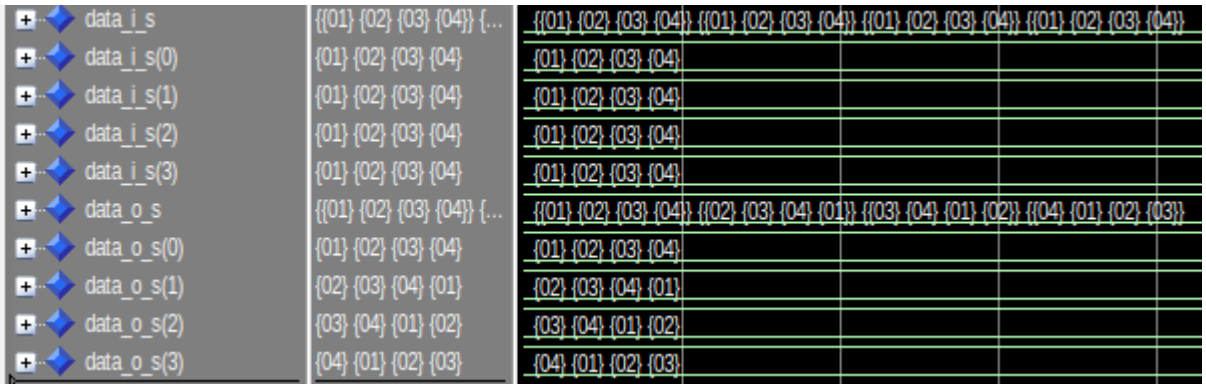

Figure 9 - Résultats du test bench de ShiftRows

Les trois décalages sont bien effectués, le composant est donc fonctionnel !

C- MixColumns

Cette fonction applique une multiplication de deux matrices dans l'espace GF(2^8) en suivant les règles suivantes :

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

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

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

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

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

Figure 10 – Produit matriciel de MixColumns

Le sigle ⊕ signifie le ou exclusif.

1- Modélisation

Pour effectuer la multiplication par 2, notée • {02}, on regarde d'abord si le bit de poids fort de l'octet que l'on traite est à 1, s'il ne l'est pas, dans ce cas la multiplication par 2 consiste en un décalage de l'octet de 1 vers la gauche.
Si le bit de poids fort est à 1, alors on va d'abord effectuer un XOR entre cet octet et un polynôme spécifique : $x^8 + x^4 + x^3 + x + 1$ dont la représentation binaire est b'100011011. Le bit de poids fort sera de ce fait à 0 et l'on pourra effectuer un décalage classique.

```
66    function mult2 (a : bit8) return bit8 is
67    variable pol : bit8;
68    variable result: bit8;
69    begin
70        pol := X"1B" ;
71        IF a(7) = '0' THEN
72            result := a(6 downto 0) & '0';
73        ELSE
74             result := (a(6 downto 0) & '0') XOR pol;
75        END IF;
76        return result;
77    end mult2;
```

Figure 11 - Fonction mult2

Une fois la multiplication par 2 implémentée, la multiplication par 3 est très simple puisqu'il s'agit d'un XOR de l'entrée et du résultat de la multiplication par 2.

```
79      function mult3 (a : bit8) return bit8 is
80      variable result: bit8;
81      begin
82          result := mult2(a) XOR a;
83          return result;
84      end mult3;
```

Figure 12 - Fonction mult3

Ces deux procédés sont codés en tant que fonction dans le fichier « CryptPack.vhd ».

Il a également fallu implémenter un MUX car le MixColumns n'est pas effectué à tous les rounds (cf *figure 1*). Ce multiplexeur aura donc deux entrées : la matrice d'entrée data_i et la matrice à laquelle on a appliqué le MixColumns. Un std_logic appelé « enable_i » nous permettra de choisir quelle matrice envoyer en sortie.

J'ai choisi que si enable_i = '1', dans ce cas c'est la matrice passée dans le MixColumns qui est envoyée en sortie.
A l'inverse, si enable_i = '0', c'est l'entrée qui est directement envoyée en sortie.

La structure de MixColumns est donc la suivante :

Figure 13 - Ports de MixColumns

Voici à quoi ressemble le MUX dans MixColumns :

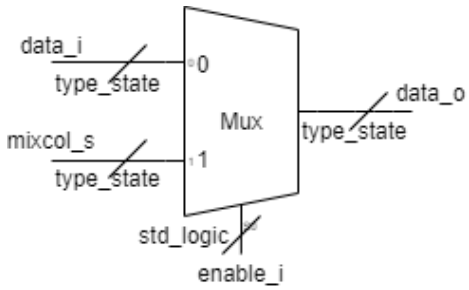

Figure 14 - MUX à l'intérieur du bloc MixColumns

2- Test bench

Pour vérifier si ce composant fonctionne, nous allons utiliser l'exemple du sujet :

```
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
```

Figure 15 - Exemple du sujet pour le round 1

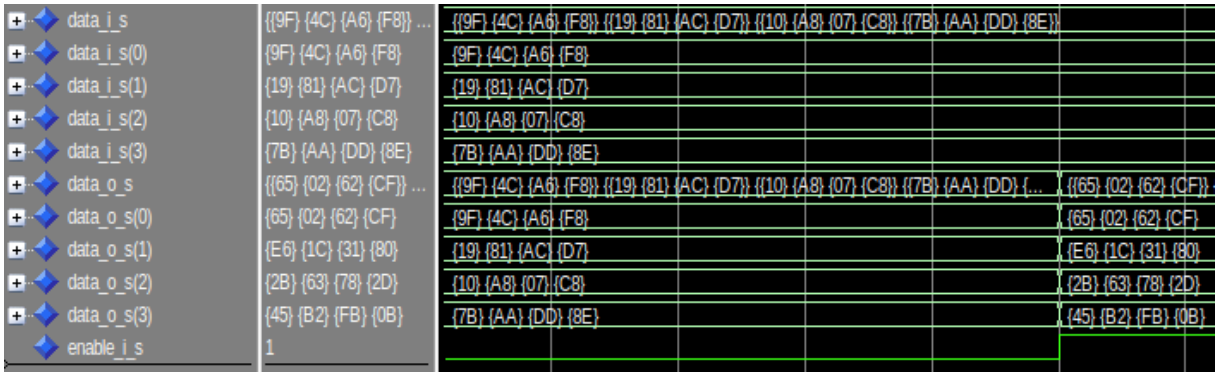

Figure 16 - Résultats du test bench de MixColumns

Comme on peut le voir, la sortie est égale à l'entrée tant que enable_i est égal à 0. A partir du moment où il passe à 1, la sortie devient le résultat de MixColumns, d'ailleurs bien égal à l'exemple. Le composant est fonctionnel !

D- AddRoundKey

Cette fonction ajoute la clé du round courant à l'état. Cette clé est calculée grâce au bloc Keyexpander_IO que l'on détaillera plus tard.

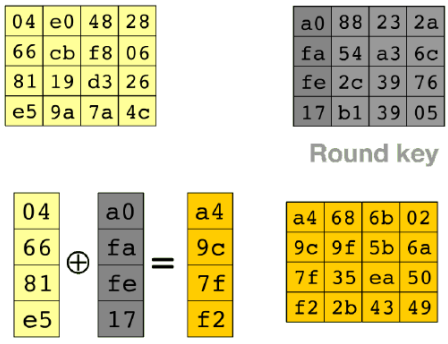

Figure 17 - Calcul de la fonction AddRoundKey

1- Implémentation

Ce composant a deux entrées, la clé du round courant ainsi que l'état, et une sortie, le résultat de la fonction.

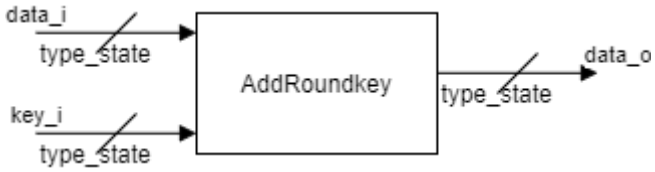

Figure 18 - Ports de AddRoundKey

Le composant AddRoundKey est lui-aussi simple à mettre en place, ce n'est que 4 XOR sur les colonnes des deux entrées.

Pour simplifier le code de ce bloc, j'ai créé une fonction XOR_ligne dans le fichier « CryptPack.vhd » qui prend en paramètres deux lignes de type row_state et retourne le XOR de ces deux lignes. Celui-ci se fait élément par élément des lignes.

Effectuer les XOR ligne par ligne est plus simple que de les faire colonne par colonne dans notre cas parce que nous manipulons des matrices sous formes de tableaux de lignes. De ce fait, nous avons directement accès aux lignes tandis que pour modifier une colonne, nous devons effectuer 4 commandes différentes.

```
57      function xor_ligne ( L,R: row_state ) return row_state is
58      variable RESULT: row_state;
59      begin
60          for i in 0 to 3 loop
61              RESULT(i) := L(i) xor R(i);
62          end loop;
63              return RESULT;
64      end xor_ligne;
```

Figure 19 - Fonction xor_ligne

2- Test bench

Comme pour le test du bloc MixColumns, nous allons utiliser l'exemple décrit sur le sujet :

```
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
```

Figure 20 - Exemple tiré du sujet

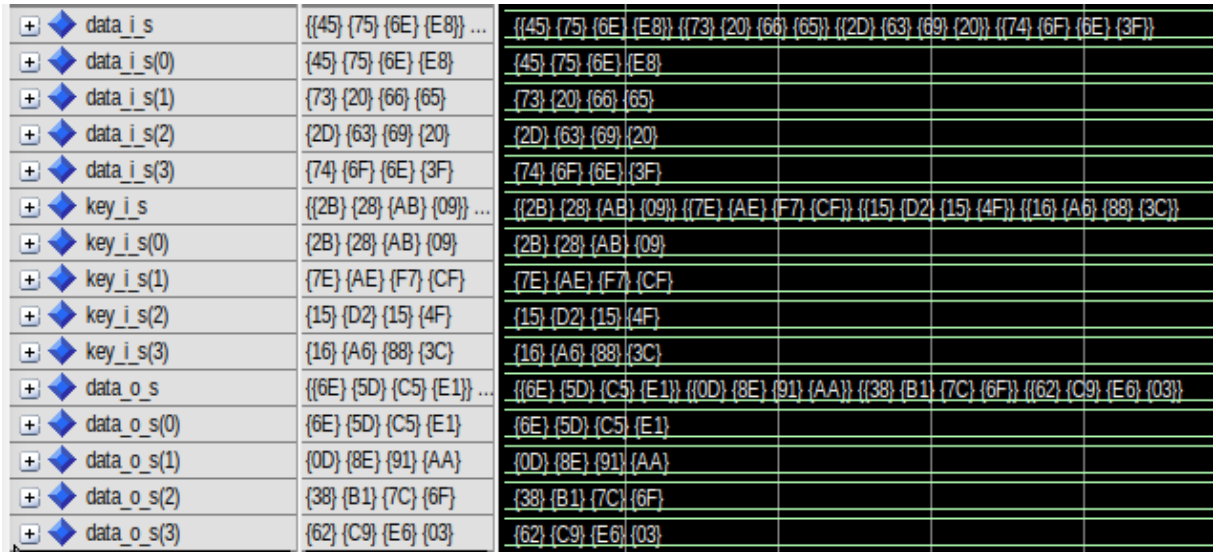

Figure 21 - Résultats du test bench d'AddRoundkey

La sortie est bien égale à ce qui est annoncé dans l'exemple du sujet, ce composant est donc fonctionnel !

E- AESRound

Cette fonction regroupe tous les composants précédemment exposés et permet d'effectuer un round AES normal, mais également le premier (seulement AddRoundKey) et le dernier (tout sauf MixColumns).

1- Implémentation

Comme expliqué précédemment, ce composant va faire appel à tous les autres instanciés jusqu'à présent. AESround est composé dans l'ordre d'un composant SubBytes, puis d'un ShiftRows, d'un Mixcolumns suivi d'un multiplexeur pour traiter le round 0, de l'AddRoundKey et enfin d'un registre pour pouvoir synchroniser la sortie avec l'horloge et ainsi afficher la sortie uniquement lorsqu'un round a été effectué.

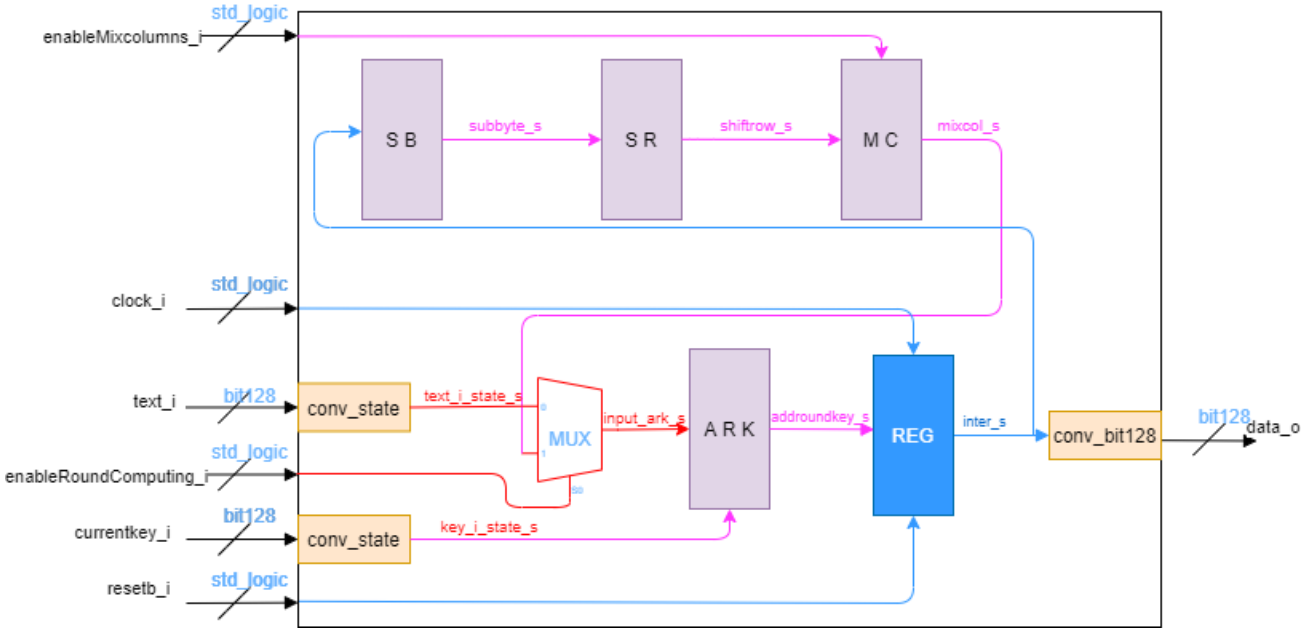

Figure 22 - Schéma de l'AESround

Ainsi, lorsque nous sommes dans le round 0, « enable_Round » est à 0 ce qui nous permet d'uniquement effectuer AddRoundKey sur la donnée texte en entrée. Il sera ensuite à 1 pour tous les autres rounds.

Dans le cas du dernier round, « enable_MC » est à 0 et la sortie de MixColumn reçoit son entrée.

2- Simulation

Nous allons ici simuler le premier round ainsi qu'un round normal en prenant comme référence l'exemple du sujet :

```
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
```

Figure 23 - Exemple du sujet

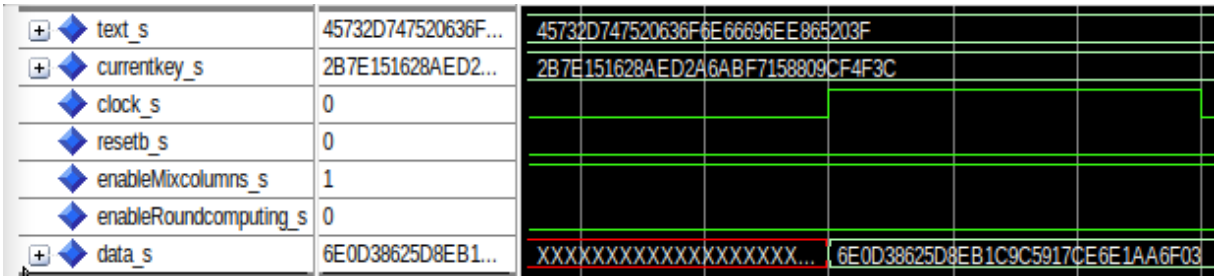

Figure 24 - Résultats du test bench pour le round 0

La sortie est bien celle attendue, l'AESround fonctionne pour le round 0 !

Pour simuler un round normal, il faut injecter le state directement dans le SubBytes. Pour ce faire, j'ai légèrement modifié le code de l'AESround pour lui ajouter une entrée en plus (appelée inter_s_i) directement branchée sur l'entrée de SubBytes.

```
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
```

Figure 25 - Round 1 tiré du sujet

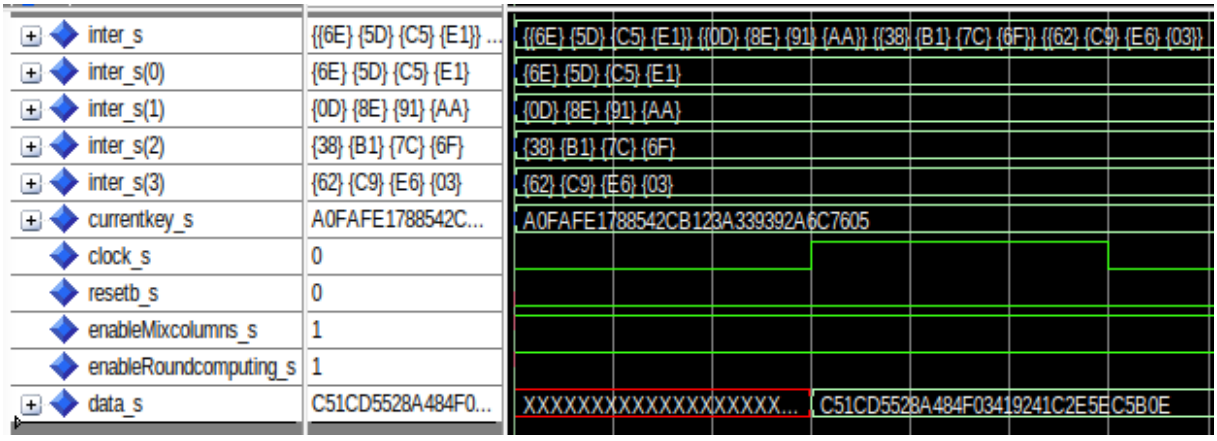

Figure 26 - Résultats du test bench pour le round 1

Ainsi, en utilisant les données du round 1, on retrouve bien data_s égal à l'état après AddRoundKey au début du round 2. L'AESround fonctionne donc pour tous les autres rounds !

II – KeyExpansion

Ce bloc va comporter tous les sous blocs nécessaires à la création des clés de rondes : KeyExpander, KeyExpanderIO et la FSM dirigeant cela.

A- KeyExpander

Cette fonction s'occupe d'effectuer toutes les transformations à la clé en entrée pour générer la clé de ronde.

1- Implémentation

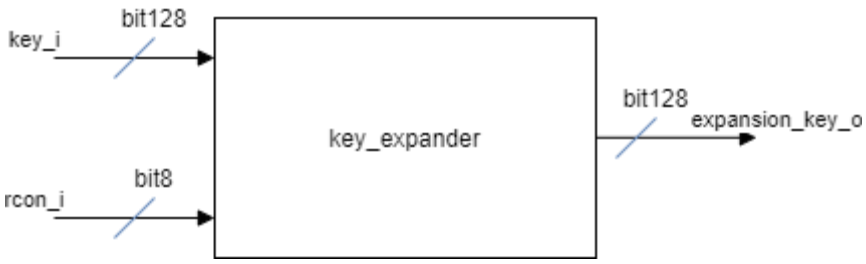

Figure 27 - Ports du bloc keyexpander

Tout d'abord, la clé en entrée de type bit128 est convertie en matrice 4*4 (state_col_t). Nous avons choisi de convertir la clé en un tableau de colonnes (i.e une matrice) parce que dans cette fonction, toutes les opérations sont effectuées sur les colonnes et non les lignes.

Ensuite, on effectue une rotation sur sa dernière colonne, puis on applique le bloc SubBytes à ses 4 éléments. Un XOR est alors effectué entre cette colonne, la première colonne de la matrice et une colonne appelée Rcon. Rcon est une colonne du type [X, 0, 0, 0] où X varie en fonction du numéro du round dans lequel on se trouve.

Une fois ces opérations effectuées, on obtient alors la première colonne de notre nouvelle clé. Obtenir le reste de cette dernière est plus simple, il suffit de faire un XOR entre la colonne 2 de l'ancienne clé et la colonne 1 de la nouvelle pour obtenir la colonne 2 de la nouvelle et ainsi de suite pour les colonnes 3 et 4.

Tout cela peut se résumer de la sorte :

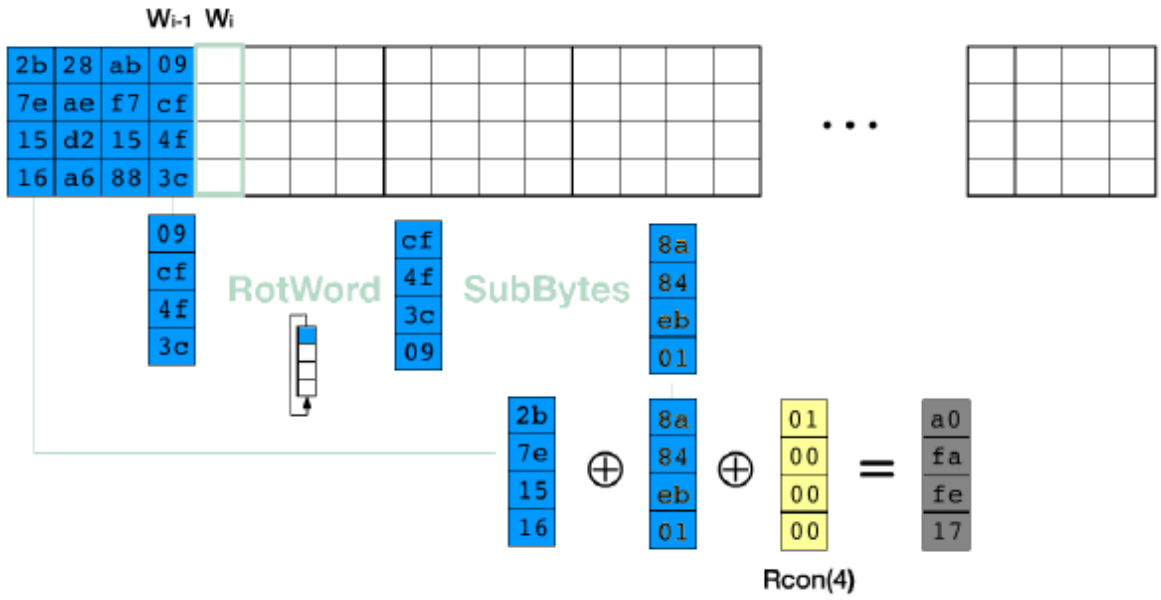

Figure 28 - Création de la première colonne de la nouvelle clé

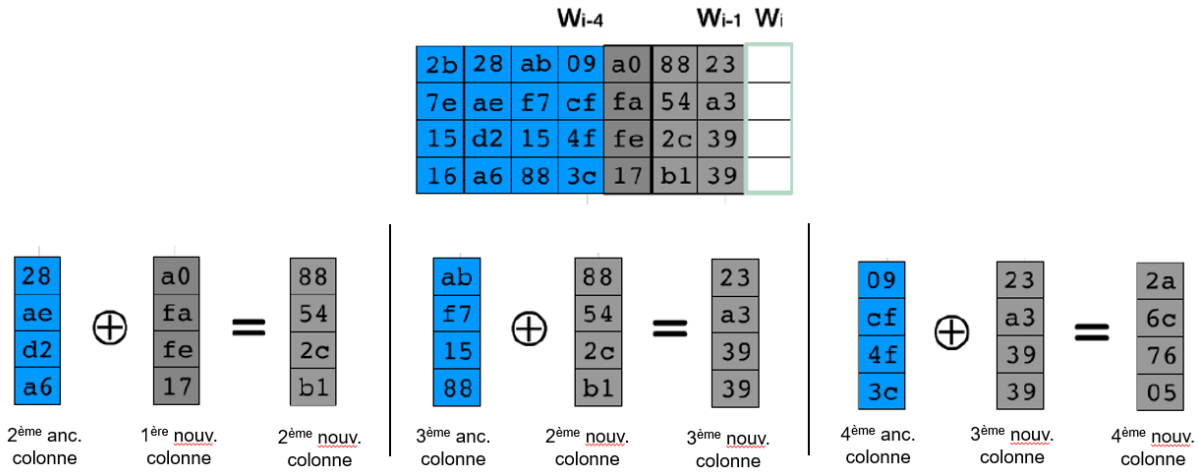

Figure 29 - Création des 3 autres colonnes de la nouvelle clé

2- Test bench

Nous allons ici créer la clé du round 1 à partir de la clé de base. Pour cela, la colonne Rcon à utiliser est [01, 00, 00, 00] (nous verrons plus en détail comment cela fonctionne dans la FSM du KeyExpander)

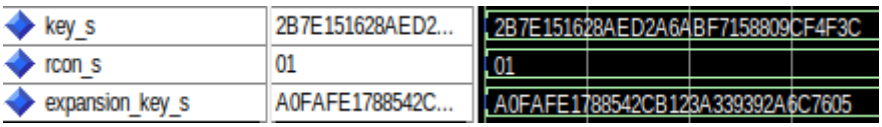

Figure 30 - Résultat du test bench de la génération de la clé du round 1

La clé obtenue est bien celle attendue, le bloc est donc fonctionnel !

B-KeyExpander_FSM_Moore

Cette fonction est le bloc qui va commander la création de chacune des nouvelles clés.

1- Modélisation

La machine d'état est composée de 3 états : init, count et done et comporte 4 entrées et deux sorties :

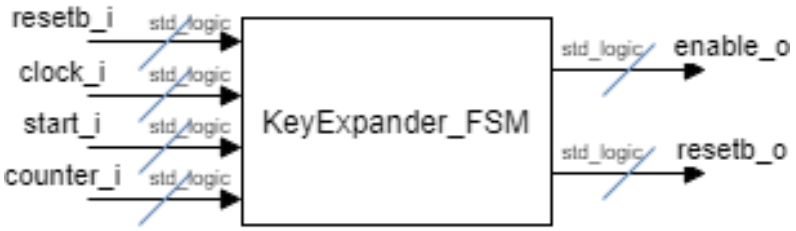

Figure 31 - Ports du bloc KeyExpander_FSM

Pour lancer la machine, il faut que le bit de reset « resetb_i » passe à 1 pendant un front montant de l'horloge, dès lors on basculera sur l'état « init ».
Ensuite, tant que l'entrée « start_i » est égale à 0, on reste sur l'état init. Si elle passe à 1, on rentre alors dans l'état count. Dès lors, le compteur va se mettre en route et va s'incrémenter de 1 à chaque front d'horloge. Une fois que le compteur passe à 9, on rentre dans l'état done et on y reste tant que « start_i » est égale à 1.
Les sorties en fonction des états sont les suivantes :

Init : Enable_o = 0
Resetb_o = 1

Count : Enable_o = 1
Resetb_o = 0

Done : Enable_o = 0
Resetb_o = 0

Cela donne le graphe d'état suivant :

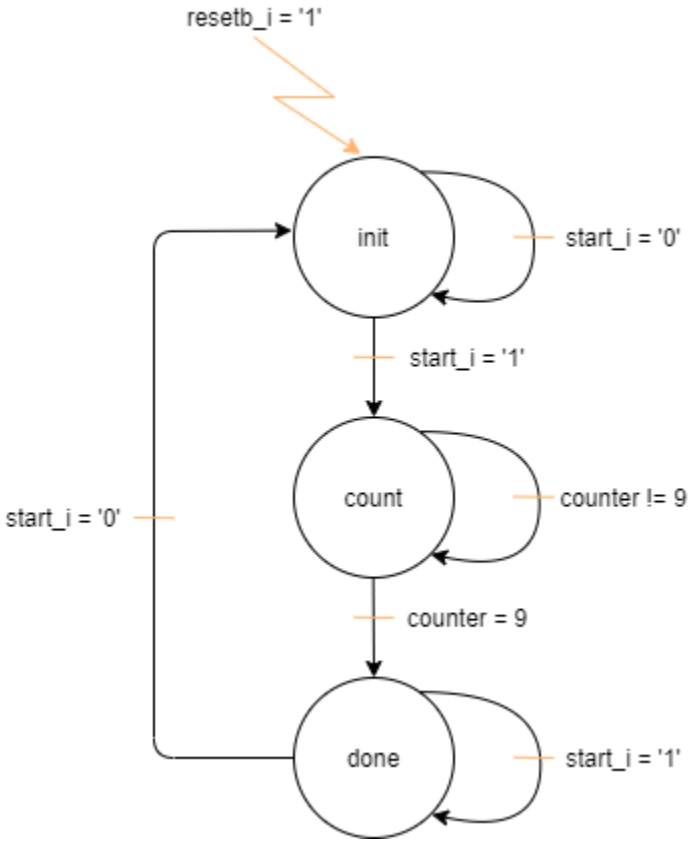

Figure 32 - Machine d'état keyexpander_FSM

2- Test bench

Pour réaliser ce test bench, on observe les sorties en fonction de l'état présent :

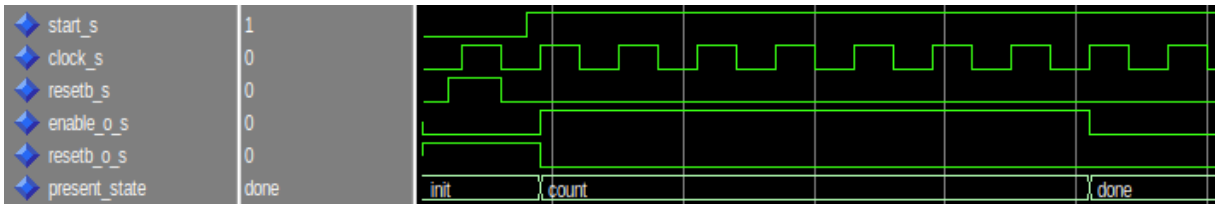

Figure 33 - Résultat du test bench de la FSM du KeyExpander

Les sorties sont cohérentes avec notre graphe d'état, ce bloc est donc fonctionnel !

C-KeyExpander_IO

Cette fonction vient regrouper KeyExpander et KeyExpander_FSM. Elle englobe également un registre en sortie de KeyExpander, un MUX en sortie du registre et un compteur.

1- Modélisation

Dans ce bloc, nous allons commander tous les autres blocs permettant la génération des clés de rondes.

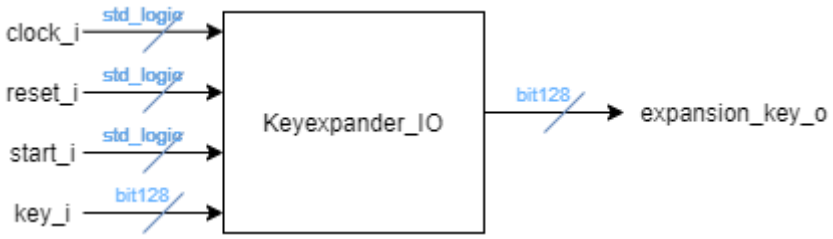

Figure 34 - Ports de Keyexpander_IO

Tout d'abord, la FSM contrôler un compteur qui permet d'envoyer en entrée du keyexpander la bonne valeur de Rcon via l'entrée « Rcon_i ». Le tableau étant stocké dans « Cryptpack.vhd », il suffit juste de le parcourir grâce aux indices.

Ensuite, en sortie du keyexpander se trouve un registre de type bit128. Il a pour rôle de synchroniser la sortie avec l'horloge. Un multiplexeur est placé par la suite pour pouvoir gérer le round 0 où la clé de ronde est simplement la clé de base. Lui aussi est piloté par le compteur.

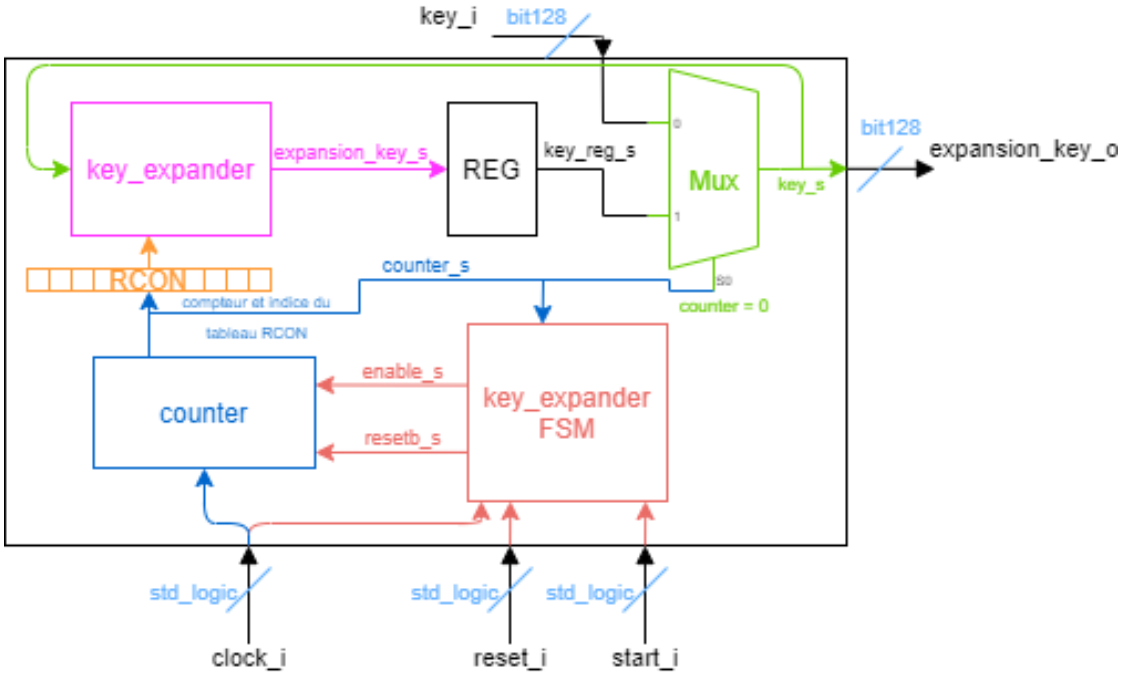

Figure 35 - Schéma du Keyexpander_IO

2 – Test bench

Pour vérifier que les clés générées sont les bonnes, on effectue une simulation avec la clé de l'énoncé en entrée :

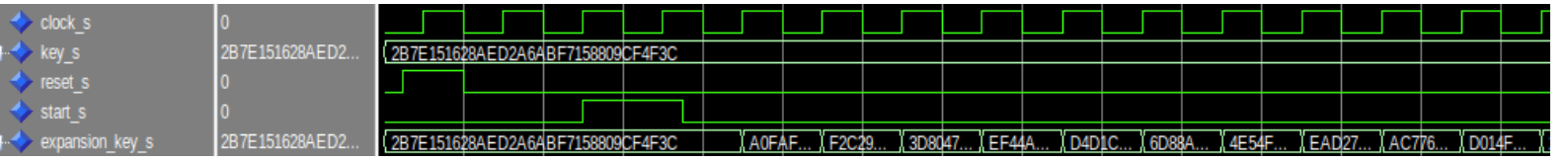

Figure 36 - Résultat du test bench de keyexpander_IO

Les 9 clés de rondes sont bien les bonnes, cependant je n'ai pas réussi à faire en sorte que la dernière clé de ronde reste en sortie du bloc puisque lorsque l'on reste dans l'état « done », keyexpander continue de fonctionner. En revanche, si l'on repasse à l'état « init » en mettant start_i à 0, c'est la clé initiale qui est envoyée en sortie.

Pour autant, les clés générées sont bien les bonnes, le bloc est donc fonctionnel.

III – Top level, simulation finale

Dans cette partie, nous allons traiter la machine d'état contrôlant l'AES ainsi que le toplevel. Comme ces deux fichiers étaient fournis, nous ne nous pencherons pas sur leur modélisation mais uniquement sur leur fonctionnement et leur simulation.

A- FSM_AES

1- Description

Le rôle de cette fonction est d'orchestrer tous les autres blocs de l'AES : le compteur, l'AESround et le Keyexpander_IO comme on peut le voir avec les couleurs des différentes sorties :

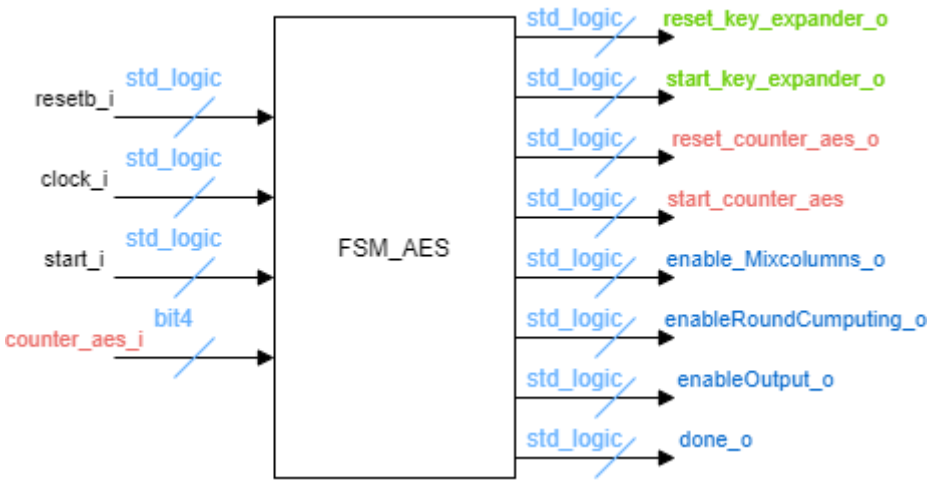

Figure 37 - Ports de la FSM_AES

Représentons à présent les différents états :

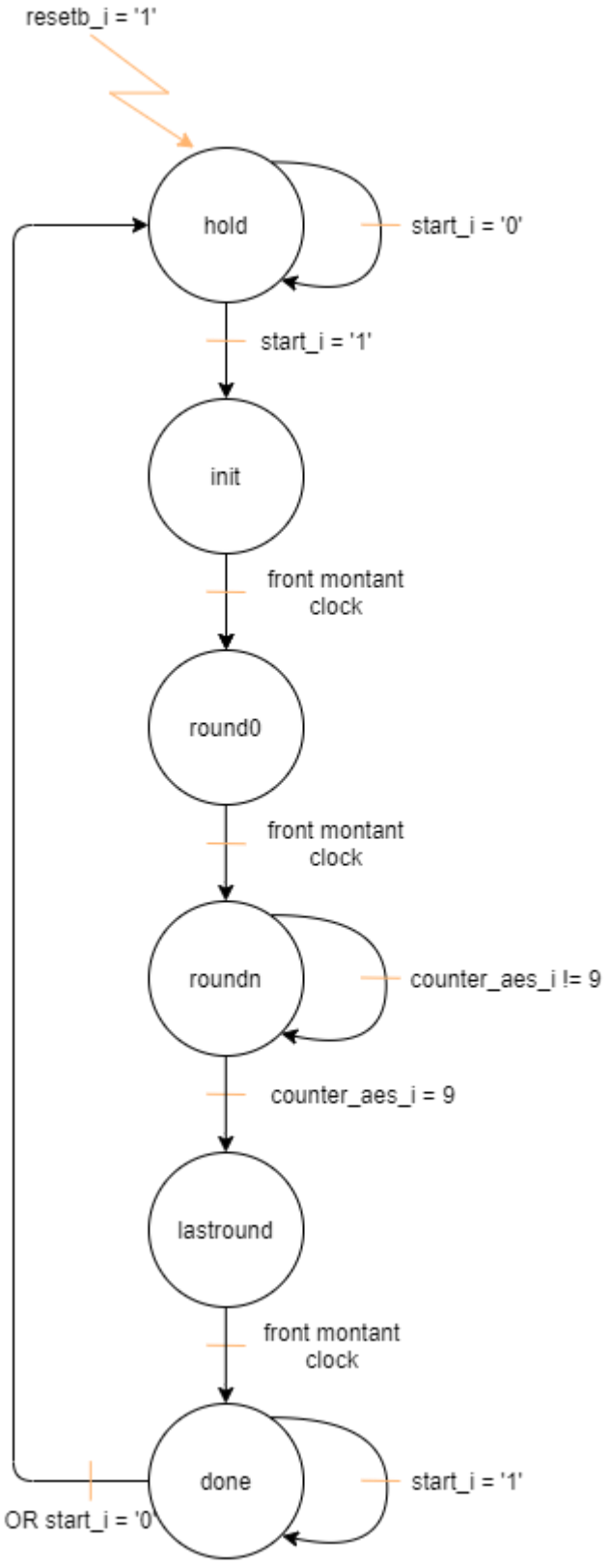

Figure 38 - Graphe d'état de la FSM_AES

```
when hold=>
  reset_key_expander_o   <= '1';
  start_key_expander_o   <= '0';
  reset_counter_aes_o    <= '1';
  enable_counter_aes_o   <= '0';
  enableMixColumns_o     <= '1';
  enableRoundcomputing_o <= '0';
  enableOutput_o         <= '0';
  done_o                 <= '0';
when init =>
  reset_key_expander_o   <= '0';
  start_key_expander_o   <= '1';
  reset_counter_aes_o    <= '0';
  enable_counter_aes_o   <= '0';
  enableMixColumns_o     <= '1';
  enableRoundcomputing_o <= '0';
  enableOutput_o         <= '0';
  done_o                 <= '1';
when round0 =>
  reset_key_expander_o   <= '0';
  start_key_expander_o   <= '0';
  reset_counter_aes_o    <= '0';
  enable_counter_aes_o   <= '1';
  enableMixColumns_o     <= '1';
  enableRoundcomputing_o <= '0';
  enableOutput_o         <= '0';
  done_o                 <= '1';
when roundn =>
  reset_key_expander_o   <= '0';
  start_key_expander_o   <= '0';
  reset_counter_aes_o    <= '0';
  enable_counter_aes_o   <= '1';
  enableMixColumns_o     <= '1';
  enableRoundcomputing_o <= '1';
  enableOutput_o         <= '0';
  done_o                 <= '1';
when lastround =>
  reset_key_expander_o   <= '0';
  start_key_expander_o   <= '0';
  reset_counter_aes_o    <= '0';
  enable_counter_aes_o   <= '1';
  enableMixColumns_o     <= '0';
  enableRoundcomputing_o <= '1';
  enableOutput_o         <= '0';
  done_o                 <= '1';
when done =>
  reset_key_expander_o   <= '0';
  start_key_expander_o   <= '0';
  reset_counter_aes_o    <= '0';
  enable_counter_aes_o   <= '0';
  enableMixColumns_o     <= '0';
  enableRoundcomputing_o <= '0';
  enableOutput_o         <= '1';
  done_o                 <= '0';
```

Figure 39 - Sorties de chaque état

Comme pour la FSM du keyexpander, j'ai considéré que reset avait lieu si le bit « resetb_i » est à '1'.

Pour résumer les sorties de chaque état :

L'état *« hold »* est l'état de standby. De ce fait, on maintient les reset à '1'.
Ensuite vient l'état *« init »*, état dans lequel on ordonne à keyexpander de se mettre en marche et ainsi de générer les clés.
L'état *« round0 »* correspondant au round 0 de l'AESround, on n'autorise pas le RoundComputing car il n'y a que AddRoundKey lors de ce round (cf I-E-1-Implémentation pour plus de détails).
L'état *« roundn »* correspondant aux rounds normaux, le RoundComputing est bien entendu autorisé tout comme le mixcolumns.
L'état *« lastround »* correspond au dernier round, round au cours duquel on n'effectue pas le mixcolumns.
Enfin, l'état *« done »* permet simplement d'afficher en sortie le message chiffré.

2- Test bench

En simulant un compteur pour pouvoir passer d'un état à un autre, on effectue la simulation :

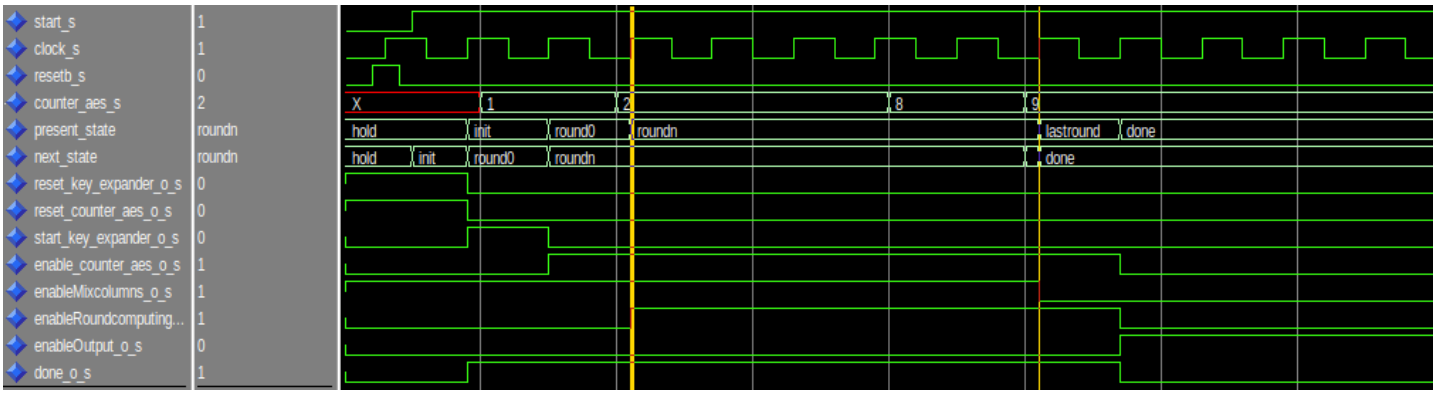

Figure 40 - Résultats du test bench de la FSM_AES

On observe que enableRoundComputing = '1' quand on sort du round0 et que enableMixColumns = '0' lors du lastround. Les autres sorties sont également cohérentes avec la machine d'état précédemment expliquée, le bloc est donc fonctionnel !

B-Toplevel

1- Description

Le toplevel rassemble l'ensemble des blocs programmés jusqu'à présent.

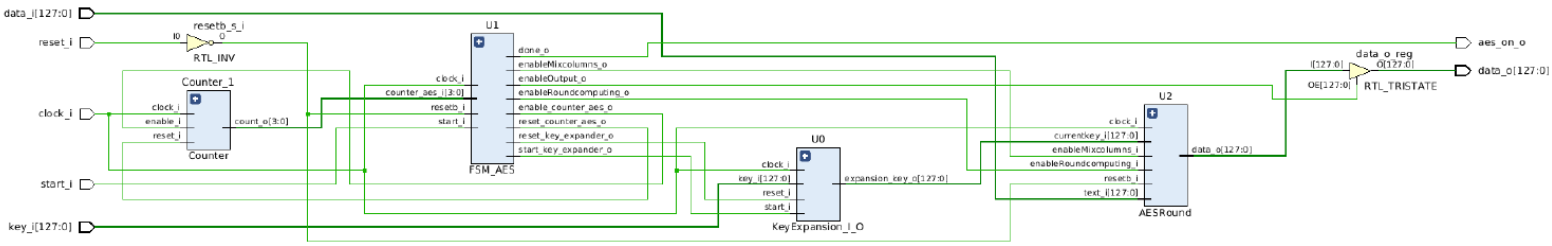

Figure 41 - Schéma du Toplevel de l'AES

On y retrouve donc le compteur de rounds, l'AESround, le keyexpander_IO ainsi que la FSM_AES.

2- Simulation

Pour réaliser la simulation finale, on saisit le message « Es-tu confinèe ? » et la clé « 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c » :

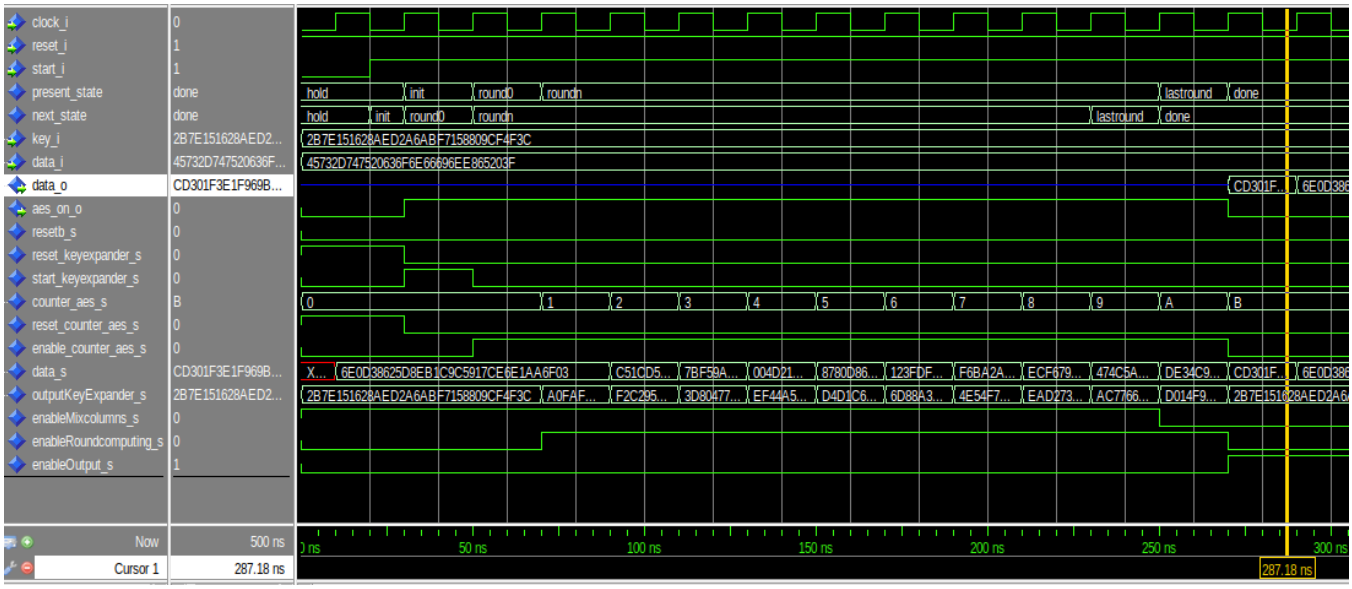

Figure 42 - Résultats du test bench du toplevel initial

Sur ce chronogramme on voit bien toutes les clés générées au fil des rounds. On voit également l'état dans lequel on se trouve l'AES à chaque tour.

On remarque cependant qu'une fois que le message a été chiffré et que les rounds sont finis, la sortie reçoit le message initial en sortie du round 0. C'est dû au fait que lorsque « start_i » est à '1', on retourne à l'état « init » et le round0 s'effectue.

Pour résoudre ce problème, on pourrait ajouter un état avant « done » que l'on appellerait « afficher » où l'on autoriserait la sortie pour un seul coup d'horloge, puis on basculerait dans l'était « done » où cette fois-ci la sortie ne serait pas autorisée.

Une fois la modification faite, voici le test bench que l'on obtient :

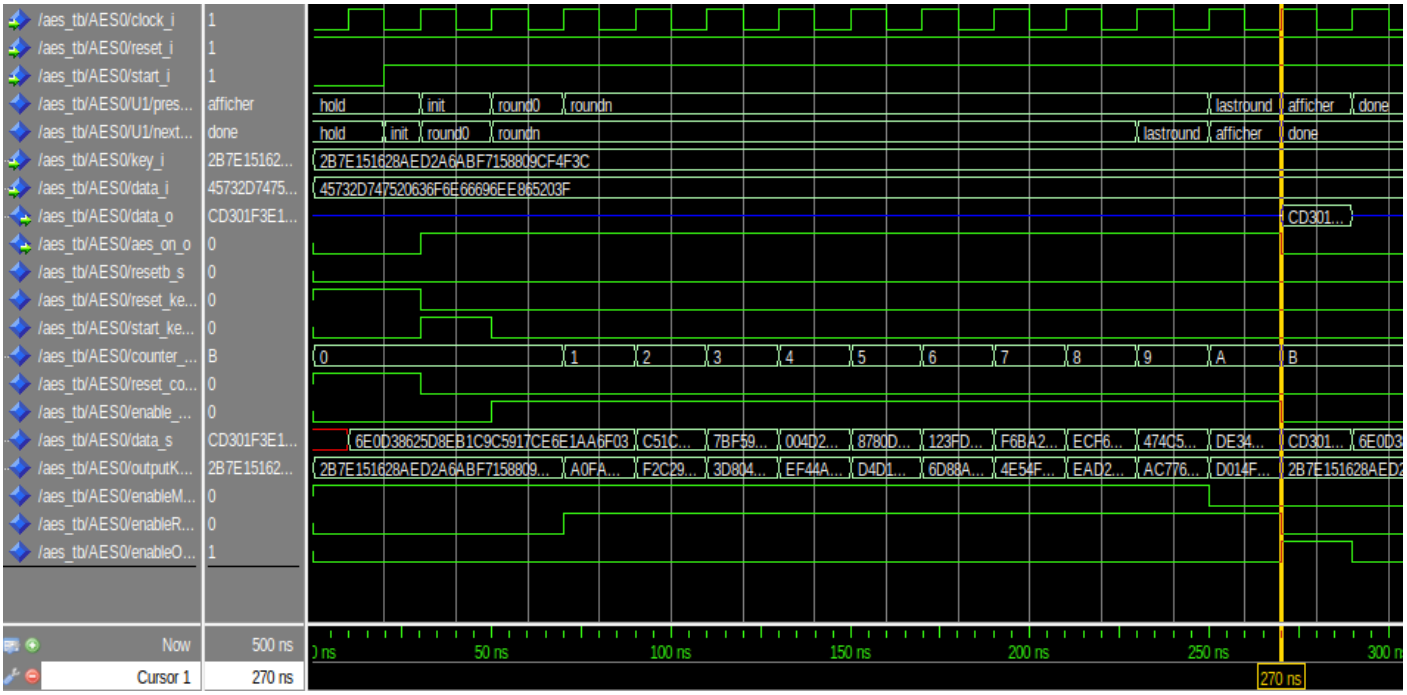

Figure 43 - Résultats du test bench du top level modifié

La sortie de l'AES est bien le message attendu, l'AES est donc fonctionnel !

Retours sur le projet

En somme, mon projet est fonctionnel. Les clés calculées sont les bonnes et le message est chiffré correctement.

Je n'ai cependant pas réussi à résoudre mon problème au niveau de la création des clés de ronde qui ne s'arrête pas. Pour autant ce n'est pas un point bloquant puisque la machine d'état de l'AES interrompt bien au bon moment l'utilisation de ces clés, et si l'on venait à coder un autre message à la suite, il y aurait un reset du keyexpander_IO et de ce fait des clés de ronde.

La deuxième difficulté que j'ai rencontrée vient du toplevel. En effet, la sortie changeait une fois le bon message chiffré affiché. J'ai pallié ce problème en ajoutant un état qui affiche seulement au bon moment le message chiffré, mais je pense que l'on aurait pu mettre un registre où la sortie est actualisée seulement si on a un front descendant du signal « aes_on », ce qui correspond à la fin du chiffrement.

J'ai trouvé ce projet particulièrement intéressant, non seulement parce qu'il m'a fait apprendre un nouveau langage, mais également parce que ce langage n'a rien en commun avec d'autres langages informatiques que je maîtrise déjà. Même si les cours en distanciel ont rendu, je trouve, l'apprentissage de ce langage plus compliqué, les séances de suivi du projet et les schémas explicatifs m'ont été bénéfiques et m'ont permis de ne pas décrocher.