Projet Conception d'un Système Numérique

Modélisation VHDL de l'Algorithme de Chiffrement AES

Date : 11 janvier 2021

Auteur : Yavuz AKIN

Groupe : EMSE ISMIN 2A - IV

Cours : Conception d'un Système Numérique

Table des matières

Introduction

L'AES est un algorithme de chiffrement symétrique développé par le NIST en 1997 et qui est encore largement utilisé et fiable aujourd'hui (1).

Dans le cadre du cours Projet Conception d'un Système Numérique, nous devons réaliser la modélisation VHDL de l'algorithme de chiffrement AES 128 bits[1].

Par conséquent, la problématique auquel nous allons répondre est la suivante : Comment modéliser l'architecture d'un AES à l'aide de VHDL ?

Afin de répondre à cette problématique, nous détaillerons premièrement les premiers sous-blocs composants l'AES, puis nous nous intéresserons au bloc de génération de clés qui fonctionne de façon parallèle avec les autres composantes de l'AES et finalement nous verrons comment fonctionne l'ensemble.

I- Composantes de l'AES

Dans cette partie, nous commençons par détailler le fonctionnement de chaque sous entité de l'AES dans le code VHDL de façon indépendante les uns des autres.

On décrira ainsi les fonctionnalités de chacun de ces sous-blocs (entrées, sorties, ...) et leur test bench. Le fonctionnement global de l'AES avec les interactions entre les différents sous-blocs est défini dans la partie III).

L'AES est un algorithme de chiffrement symétrique par bloc de 16 octets. L'algorithme elle-même est divisé en 4 sous-fonctions élémentaires : AddRoundKey, SubBytes, ShiftRows et MixColumns.

a) SubBytes

i) Description

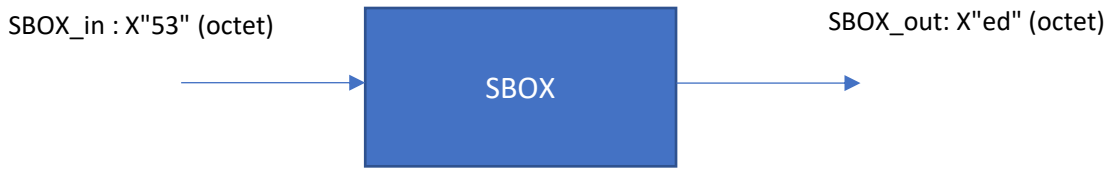
La fonction SubBytes est la première fonction qui s'applique sur les états du message à chiffrer lors des tours 1 à 10. Elle correspond à une transformation non-linéaire de l'état à en substituant chaque élément de l'état par l'élément qui lui correspond dans une table de substitution nommé S-Box.

[1] La clé aura une taille de 128 bits

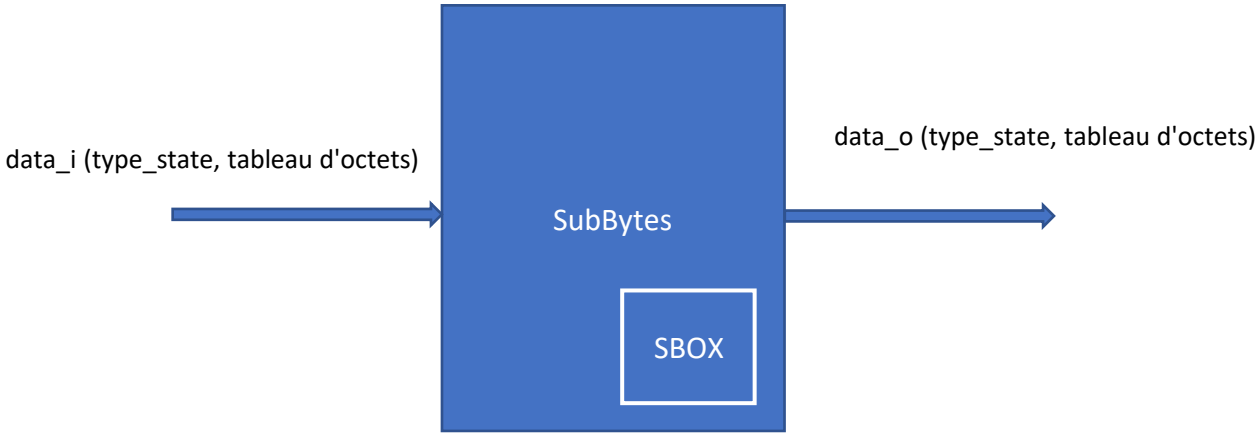
		y														
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	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

Schéma de la SBOX (2)

Dans le code VHDL, l'entité SubBytes contient une autre entité nommée SBOX.
L'entité SBOX applique la substitution pour un octet donné.



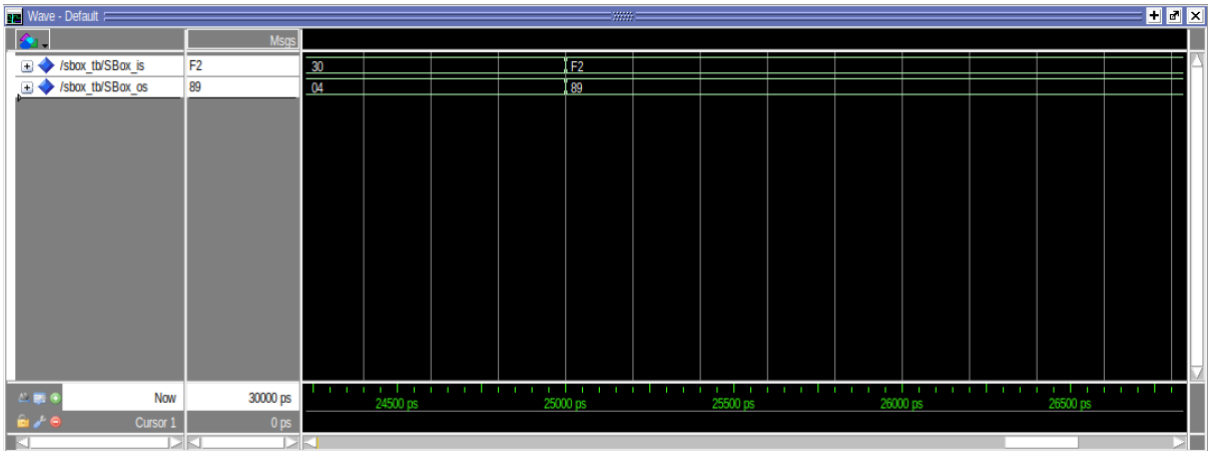
Ainsi SubBytes applique dans une double boucle for de génération l'entité SBOX aux octets de l'état entrant.



ii) Test bench

Pour le Test Bench de SBOX.vhd, on vérifie entre 0 et 25 ns que lorsqu'on a en entrée X"30" on a bien en sortie X"04" et après 25 ns que lorsqu'on a en entrée X"F2 on a en sortie X"89"

Ce qui correspond bien à notre chronogramme dans la modélisation sur ModelSim donc SBOX.vhd semble fonctionner.



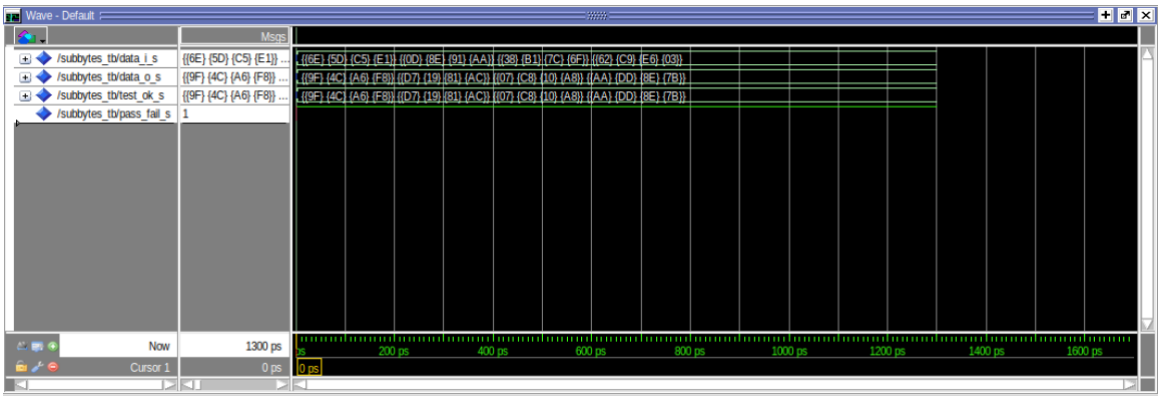
Pour le test bench de SubBytes on prend en entrée l'état :

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

Et on souhaite avoir en sortie :

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

On c'est bien le résultat qu'on a sur ModelSim.



b) ShiftRows

i) Description

La fonction ShiftRows effectue une permutation cyclique de chaque ligne de l'état en faisant un décalage qui correspond pour chaque ligne à l'indice qu'elle possède. Elle est effectuée après SubBytes.

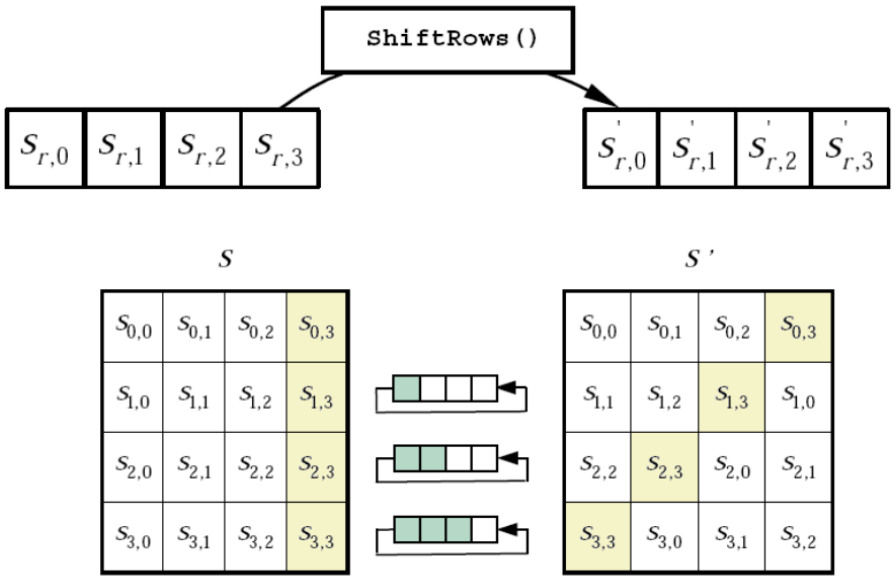
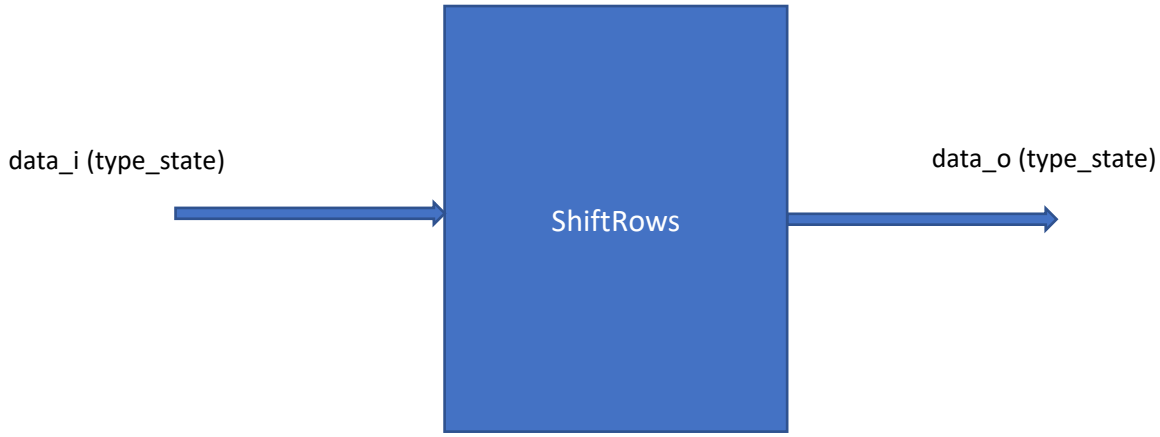


Schéma de ShiftRows (2)

Dans ShiftRows.vhd, ShiftRows est une entité qui dans une double boucle for de generate effectue la permutation des valeurs des lignes de l'état en fonction de l'indice de la ligne.



ii) Test bench

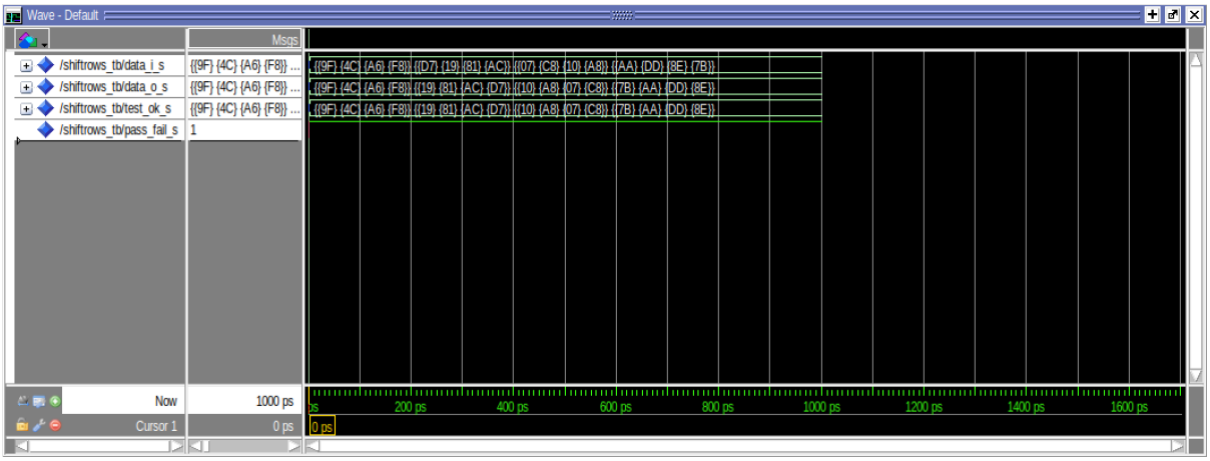
Pour tester l'entité ShiftRows on prend en entrée :

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

Et on attend en sortie :

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

On retrouve bien ce résultat dans la modélisation ModelSim :



c) MixColumns

i) Description

L'entité MixColumns applique une transformation à chacune des colonnes de l'état en entrée. Elle s'applique après ShiftRows durant les tours 1 à 9 mais ne s'applique pas au dernier tour.

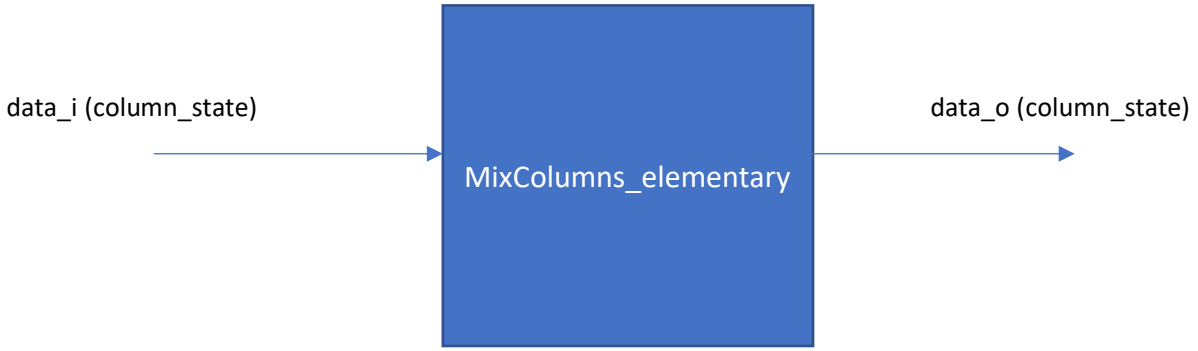
Dans son code VHDL, l'entité MixColumns utilise une autre entité nommée MixColumns_elementary qui applique une transformation à une colonne donnée en entrée.
La transformation en question est un produit matriciel entre la colonne et la matrice suivante:

02	03	01	01
01	02	03	01
01	01	02	03
03	01	01	02

En réalité, pour effectuer le produit avec cette matrice, on multiplie tous les éléments de la colonne en entrée avec 01,02, et 03 et on remplit la colonne de sortie avec les éléments qui correspondent (il n'y a pas d'opérations inutiles car chaque élément calculé est utilisé).

```
data_o(0)<=data_o_2(0) xor data_o_3(1) xor data_i(2) xor data_i(3);
data_o(1)<=data_i(0) xor data_o_2(1) xor data_o_3(2) xor data_i(3);
data_o(2)<=data_i(0) xor data_i(1) xor data_o_2(2) xor data_o_3(3);
data_o(3)<=data_o_3(0) xor data_i(1) xor data_i(2) xor data_o_2(3);
```

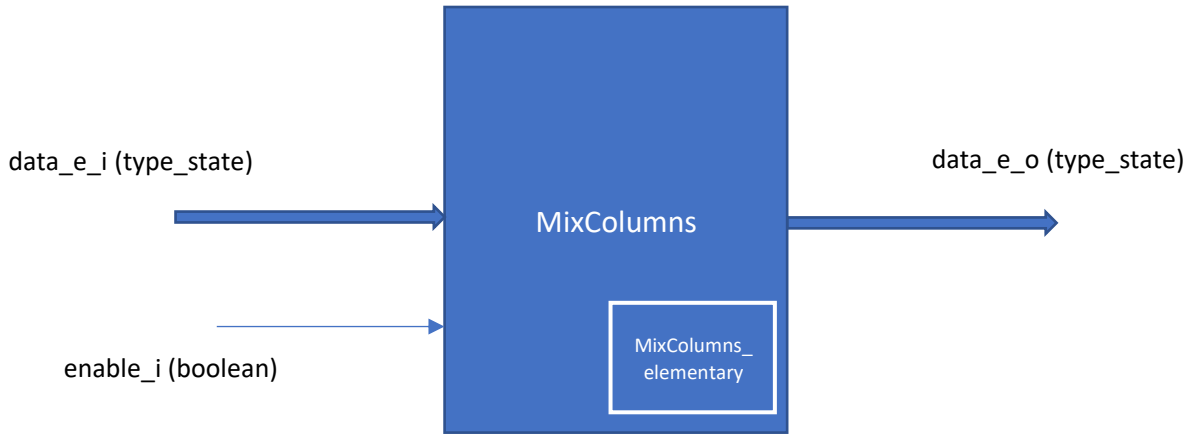
Le schéma de l'entité est le suivant :



L'entité MixColumns applique dans une boucle génération la fonction MixColumns_elementary à chacune des colonnes de l'état d'entrée. Dans la boucle, pour chaque colonne on effectuera une conversion de l'entrée en colonne_state pour pouvoir utiliser MixColumns_elementary puis on reconvertira en type_state une fois que les calculs sont terminés.

La fonction MixColumns peut être activé ou désactivé avec la commande enable_i.

Voici le schéma de l'entité :

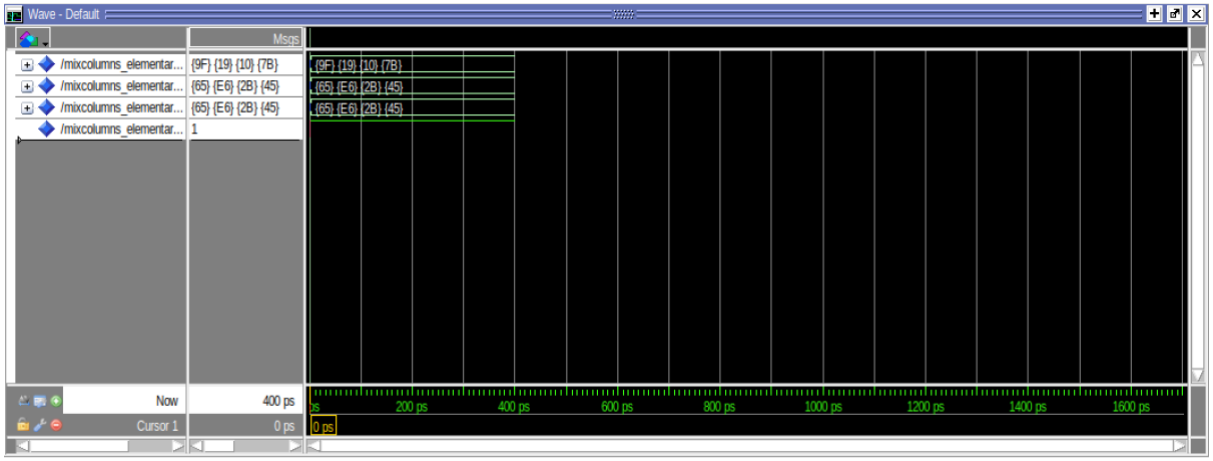


ii) Test bench

Pour tester MixColumns_elementary, on prend en entrée la première colonne et on attend en sortie la deuxième colonne :

i	o
9f	65
19	e6
10	2b
7b	45

Sur ModelSim on obtient :



Et donc MixColumns_elementary passe bien le test bench.

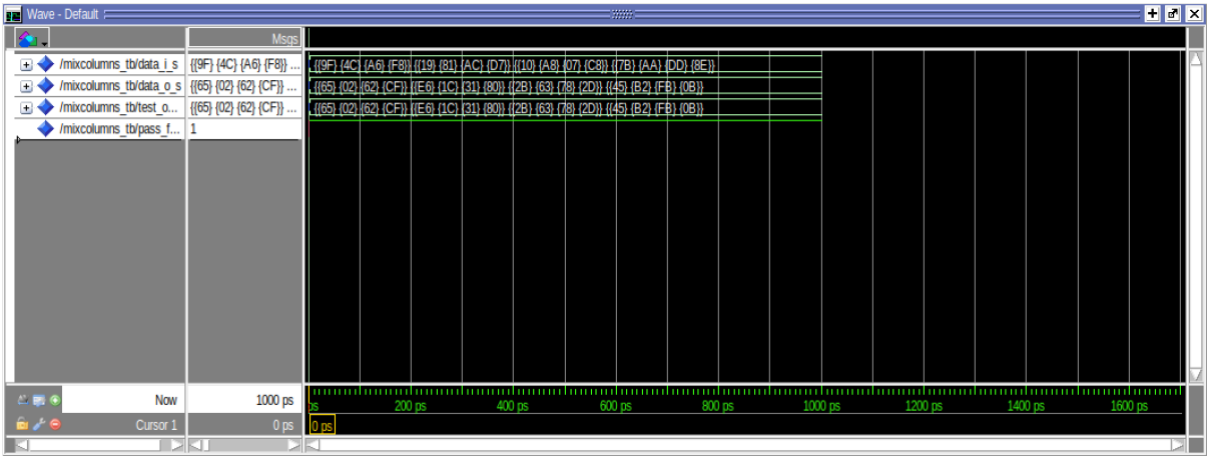
Pour le test bench de MixColumns, on prend en entrée le tableau suivant :

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

On attend en sortie :

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

D'après ModelSim, l'entité passe bien le test bench (data_o_s et test_o_s sont égales) :



d) AddRoundKey

i) Description

L'entité AddRoundKey effectue l'opération XOR (bit à bit) entre l'état et la clé de ronde courante. Elle s'applique directement sur le message en ronde 0 ou après la fonction MixColumns ou ShiftRows (ronde 10).

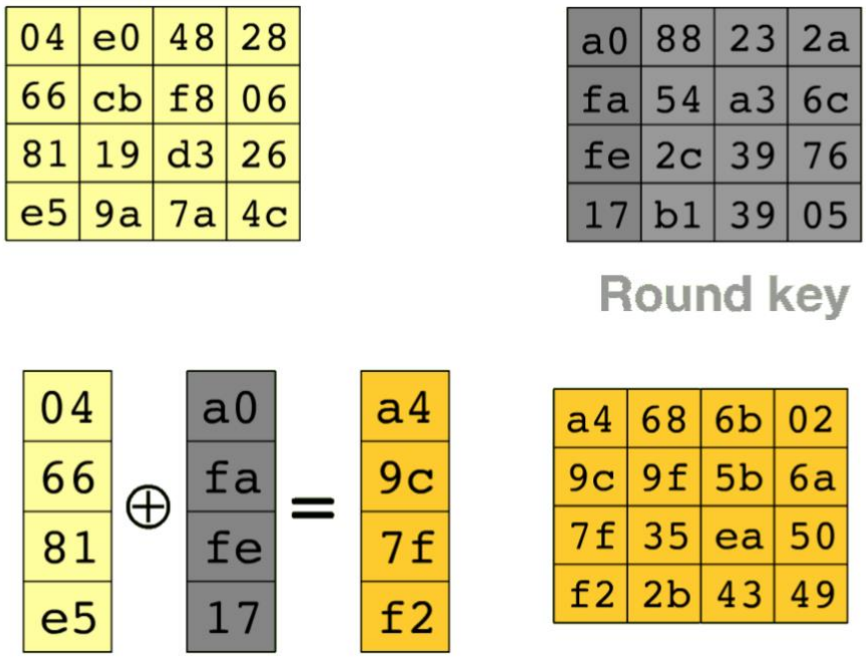
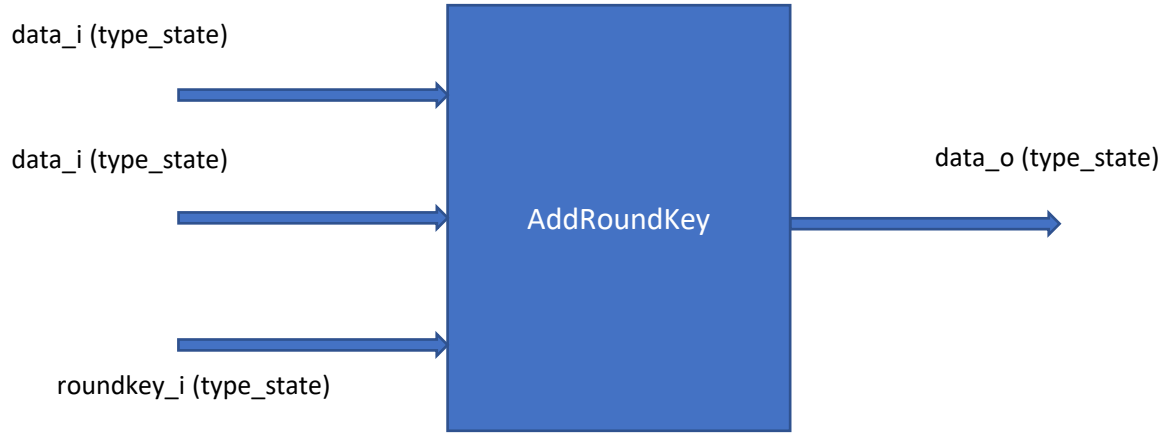


Schéma de fonctionnement de AddRoundKey (2)

Pour appliquer l'opération XOR bit à bit entre l'état et la clé, l'entité se sert d'un double boucle génération.

Voici le schéma de l'entité :



ii) Test bench

Pour le test bench du composant on met en entrée :

L'état :

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

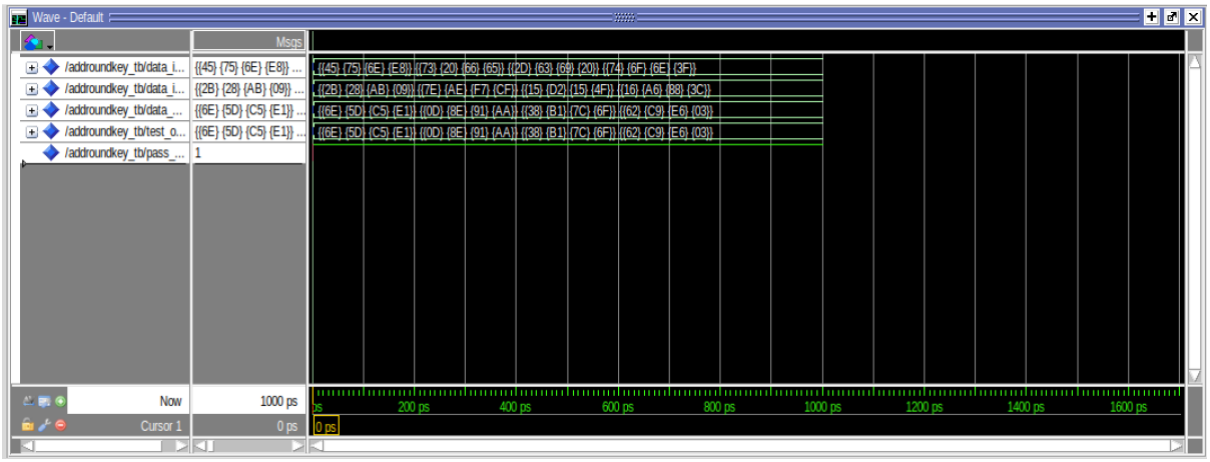
Key State :

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

On attend en sortie :

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

D'après ModelSim, l'entité passe bien le test bench (test_o_s=data_o_s) :



e) AESRound

i) Description

L'AESRound est une entité qui utilise les sous-entités précédemment décrites afin de d'exécuter un Round de l'AES.

Ci-dessous se trouve le schéma de celui-ci :

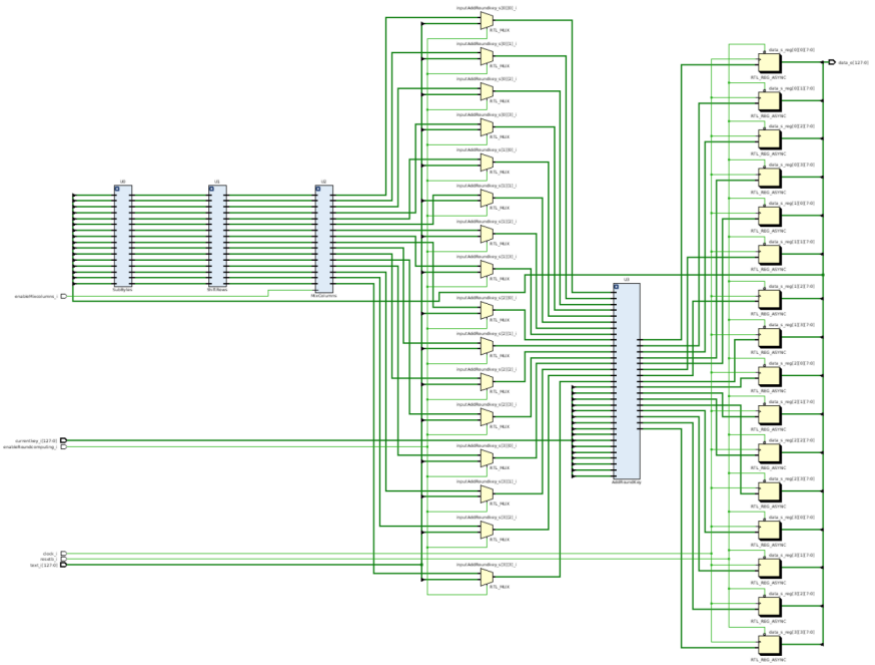


Schéma de AESRound (2)

Dans le code VHDL de cette entité, on convertit le texte en entrée et la clé à un format convenable. Puis à l'aide d'un process, on applique les sous-fonctions SubBytes, ShiftRows, MixColumns (si elle est activée) et AddRoundKey en prenant en compte les RoundKeys à

chaque front montant d'horloge Puis on reconvertit le texte de sortie au format initiale. Cela forme 1 Round. Au total on y a recours 10 fois.

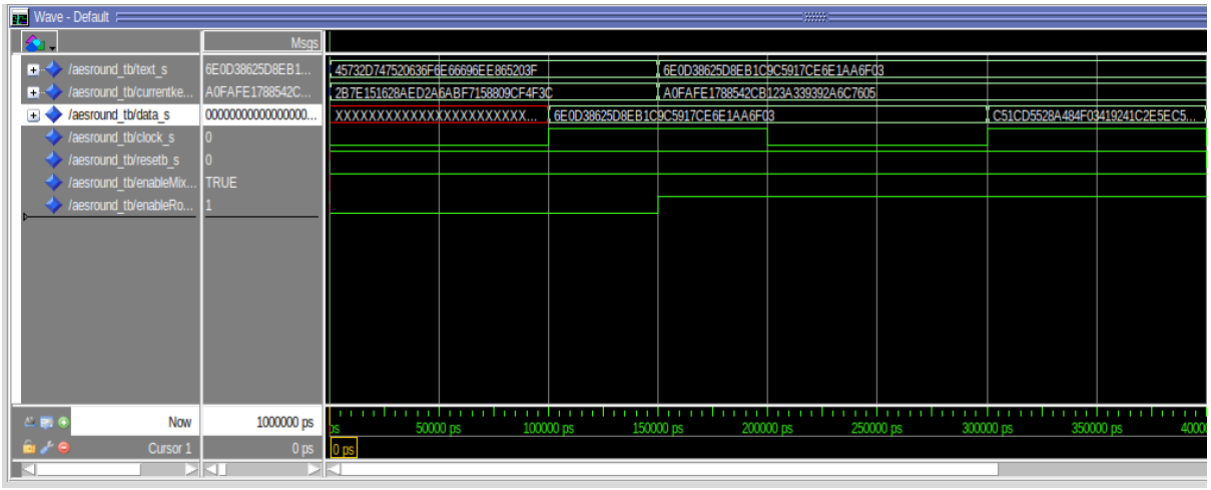
ii) Test bench

Pour le test bench de cette entité nous allons effectuer un Round de l'AES :

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



En regardant data_s, on remarque que celui-ci est bien modifié lors d'un front montant d'horloge et la donnée de sortie correspond bien aux données indiquées plus haut. Donc AESRound fonctionne bien.

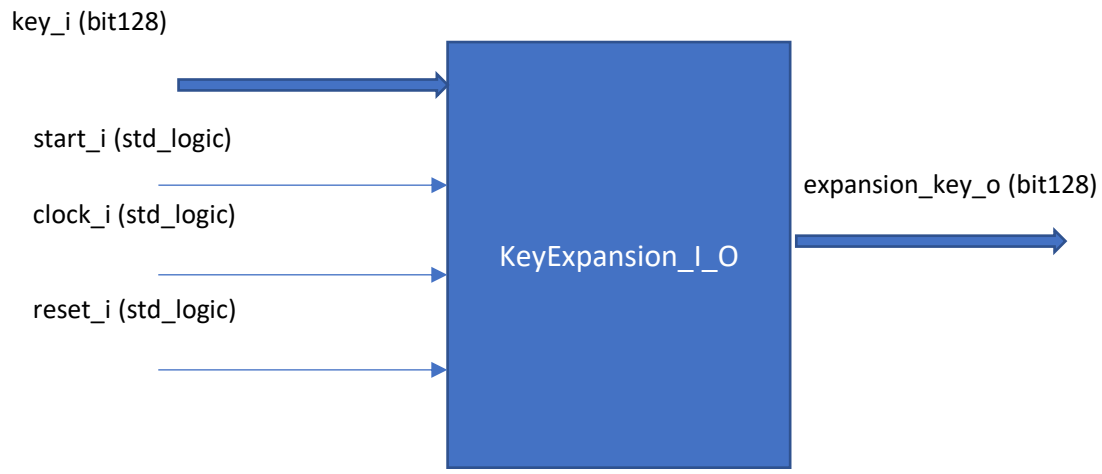
II - Diversification des Clés

Lors du processus d'obtention du message chiffré grâce à l'AES, la clé employée à chaque tour est différente de la précédente. Il existe un bloc de fonctions au sein de l'AES qui sont chargés de calculer la succession de clés.

La présentation de ce bloc se fera de telle sorte : d'abord nous décrirons le fonctionnement du système simplement et globalement. Puis nous étudierons en détails chaque composante de ce bloc. Au final, nous assemblerons ces sous-blocs pour donner la description détaillée du bloc entier de diversification de clés.

a) Fonctionnement général : KeyExpansion_I_O

Le calcul et le choix des clés se fait à l'intérieur de la fonction KeyExpansion_I_O.
Le schéma de cette fonction est la suivante :



Cette fonction reçoit en argument :

- key_i, la première clé symétrique key_i,
- start_i, le signal pour indiquer le début du processus de génération de clés
- clock_i, l'horloge du système
- reset_i, pour annuler le processus de génération et revenir à l'état initiale

Il renvoie :

- expansion_key_o, la clé qui correspond au round dans lequel on se trouve

L'entité KeyExpansion_I_O est composé de :

-3 sous entités qui assurent le fonctionnement de la génération de clés : KeyExpansion, FSM_KeyExpansion et Counter

- Un registre et un multixplexeur

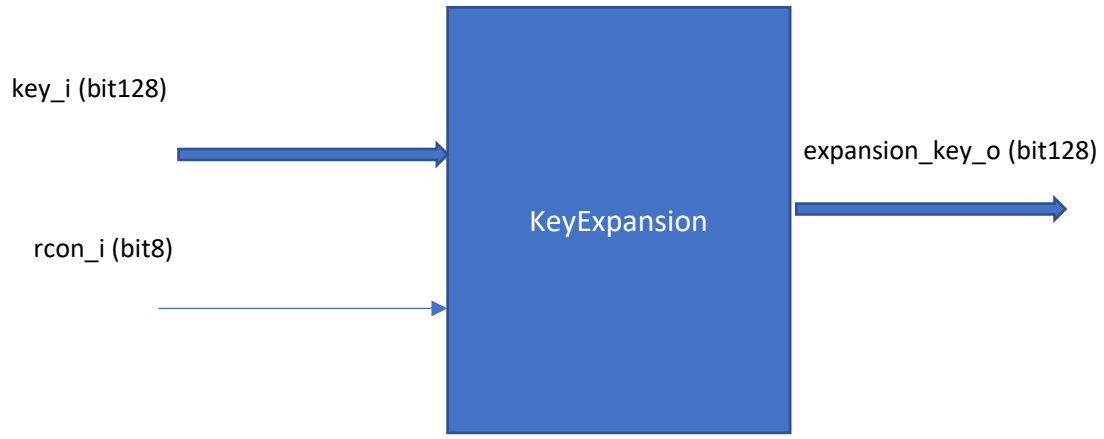
Ci-après se trouvent d'abord la description des entités composant KeyExpansion_I_O, et ensuite l'architecture de KeyExpansion_I_O formé par ces entités.

b) Construction de clé : KeyExpansion

i) Description

KeyExpansion est l'entité qui calcule la prochaine clé de ronde à être utilisée.

Son schéma est le suivant :



key_i est la clé d'entrée, rcon_i est la valeur du tableau Rcon qui est utilisé pour le calcul de la nouvelle clé. En sortie, expansion_key_o est la clé suivante à utiliser.

Elle est constituée de plusieurs étapes :

1- On effectue une rotation de la première colonne.
2- On applique SubBytes à la colonne obtenue (plus précisément S-BOX à chacun des éléments de la colonne).
3- On effectue un xor entre la première colonne de la clé d'entrée, la colonne rotation, la colonne Rcon correspond au tour dans laquelle on se trouve, le résultat est stocké dans la première colonne de la nouvelle clé.
5- Les autres colonnes de la clé sont calculés en faisant un XOR entre une colonne de la première clé et une colonne de la deuxième clé (voir le code pour plus de détails).

ii) Test bench

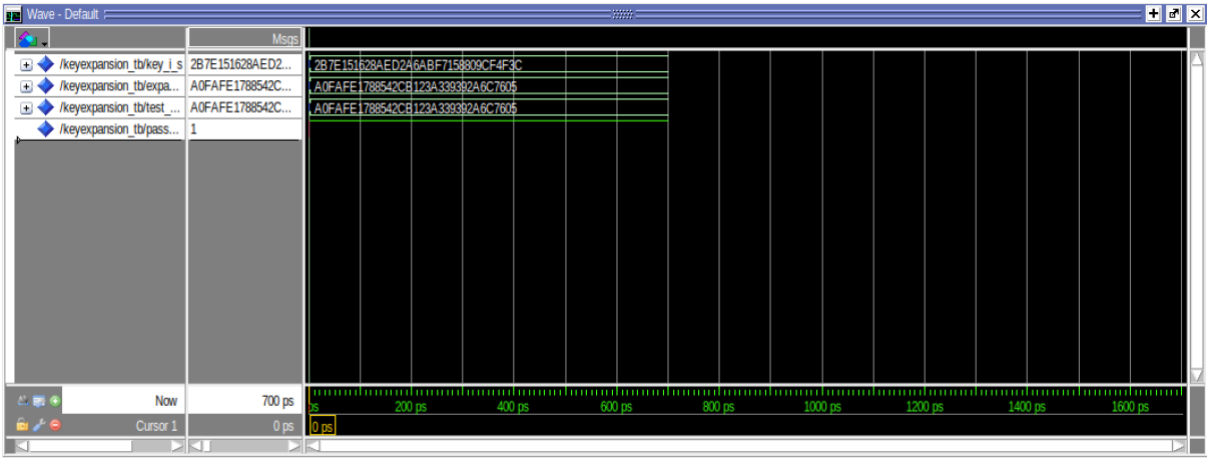
Pour tester le code on met en entrée de KeyExpansion la clé :

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

Et la valeur rcon_i : **X"01"**

En sortie on devrait avoir : **a0 fa fe 17 88 54 2c b1 23 a3 39 39 2a 6c 76 05**

D'après ModelSim, l'entité passe bien le test bench :

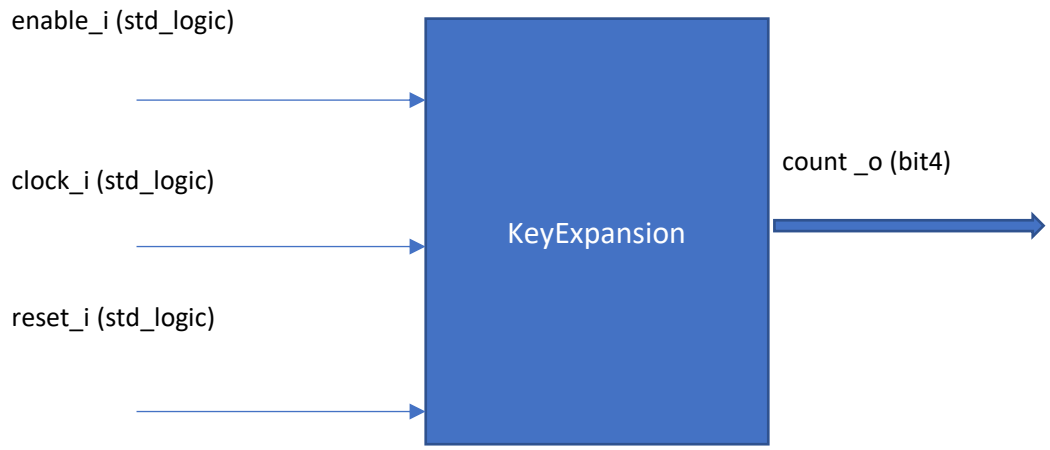


c) Gérer la synchronisation : Counter

i) Description

Le Counter permet de compter le nombre de fronts montants de l'horloge en entrée. Elle permet d'assurer la propriété de synchronisation du système. Ainsi on est capable de donner en sortie de KeyExpansion_I_O la bonne clé de ronde qui correspond au tour présent.

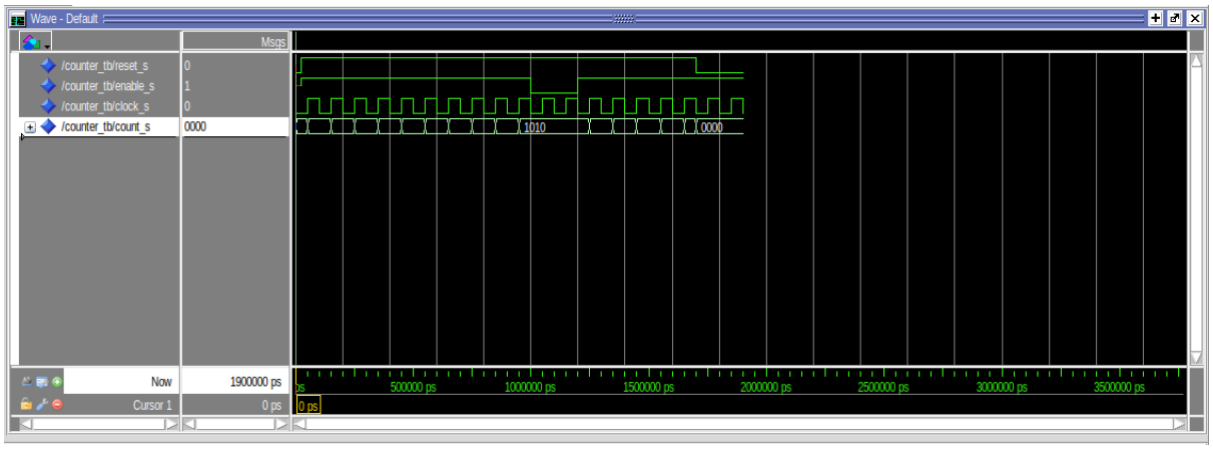
Voici le schéma de l'entité :



Le signal de sortie du Counter est un nombre 4bits qui indique le nombre de fronts montants de clock_i. Elle compte jusqu'à 15 puis redémarre, mais dans tous les cas on comptera que jusqu'à 10 car ce qui est compté est le nombre de rondes.

ii) Test bench

Pour le test bench, on met en entrée une clock et on vérifie que le compteur compte bien les fronts montants. On peut aussi tester les boutons reset et enable.



Ici, on remarque que le counter compte bien jusqu'à 15. Lorsque enable est désactive (à 1010), il arrête de compter et lorsque reset est appuyé le compteur redémarre.

d) Machine d'état gérant la diversification des clés : FSM_KeyExpansion

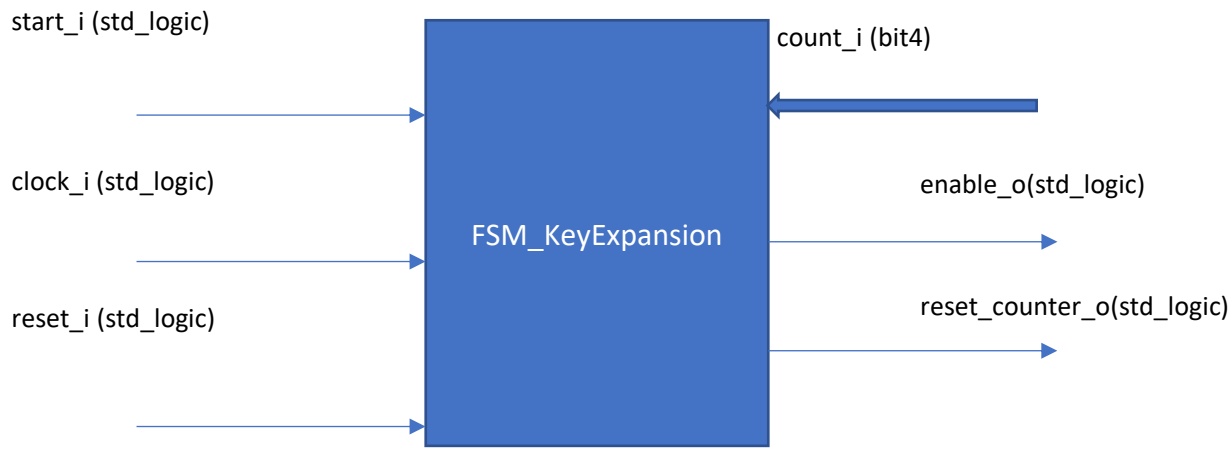
i) Description

FSM_KeyExpansion est la machine d'état qui commande les autres sous-entités de KeyExpansion_I_O. **Son architecture est celui d'une machine de Moore** (3).

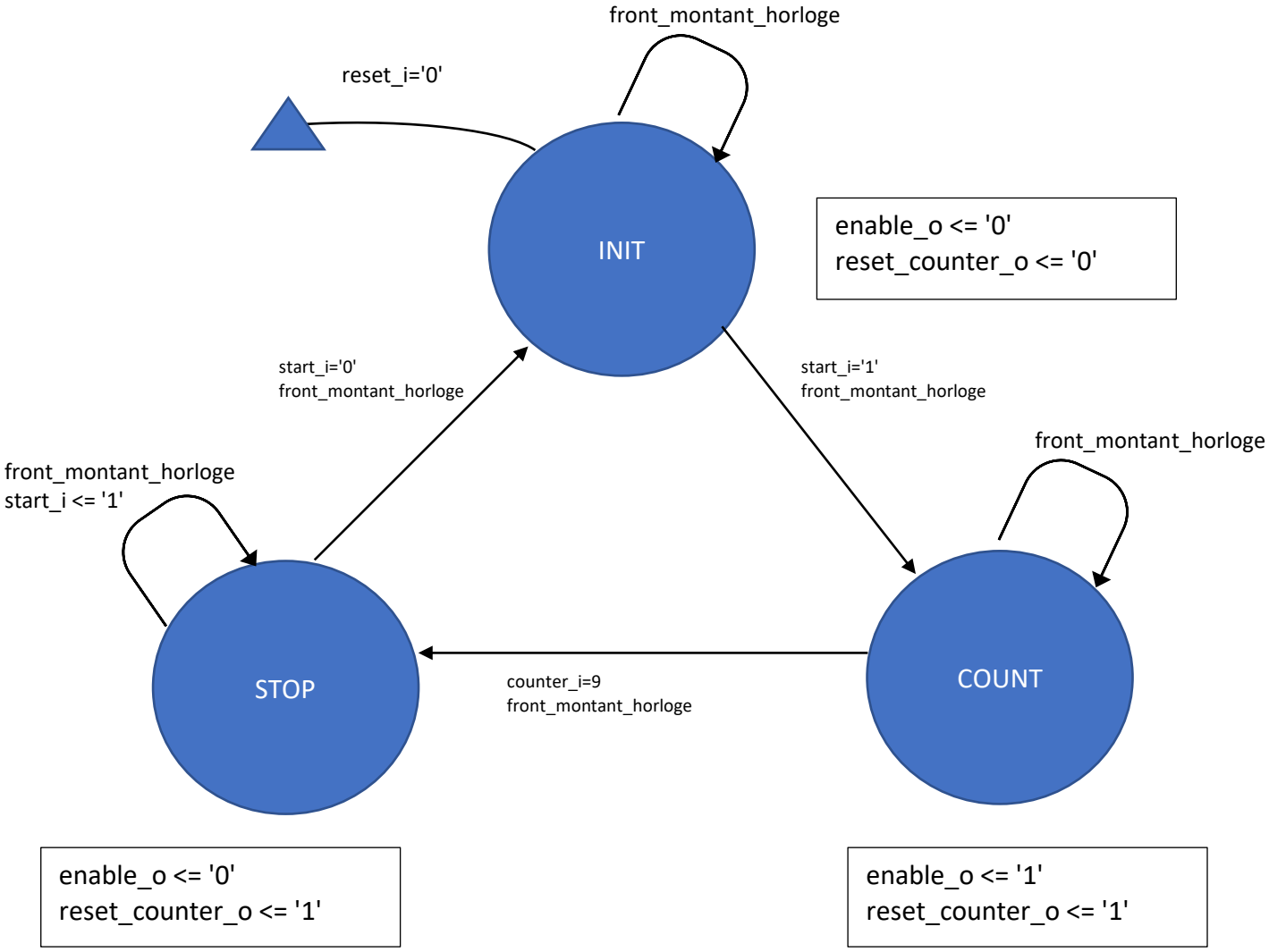
Les sorties de cette machine ne dépendent que de l'état présent (de façon synchrone), et l'état futur est calculé à partir de l'état présent et des entrées.

Voici le schéma de cette entité :

(voir ci-après)



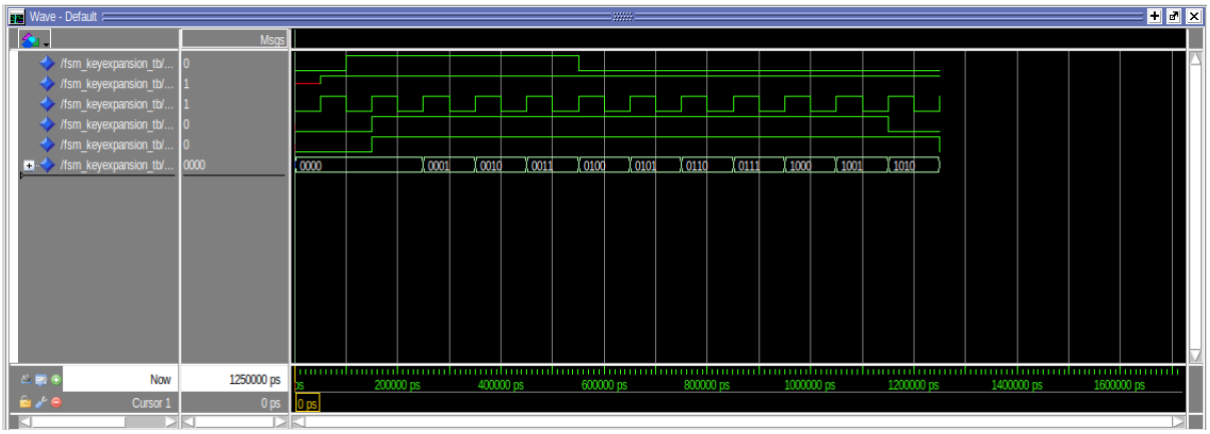
Voici le diagramme d'état de cette machine :



Elle dispose de 3 états : INIT, COUNT, et STOP. INIT correspond à l'état d'initialisation de KeyExpander_I_O. L'état count correspond à l'état de calcul des clés. L'état STOP correspond à celle ou les clés ont été calculés.

ii) Test Bench

Pour le test bench de cet élément, on fait passer la machine par l'état INIT puis COUNT puis STOP et on retourne à INIT. Durant tout le long, on vérifie que les valeurs indiquées sur le schéma plus haut sont bien respectés.



On remarque ainsi dans le chronogramme que, au départ nous sommes dans la situation INIT avec enable_o et reset_counter_o qui sont à 0, lorsqu'on a un front montant et start_i=1. La machine passe à l'état COUNT (reset_counter_o=1) et compte les fronts montants. Au 9ème décompte, elle passe à l'état STOP (reset_counter_o=1, enable_o=0). Finalement, en passant le start_i à 0, elle passe de nouveau à l'état INIT.

e) Architecture détaillé de KeyExpansion_I_O

i) Etude de l'architecture

Dans cette partie est décrite l'architecture détaillé de l'entité KeyExpansion_I_O formé par les entités précédemment décrites.

Ci-dessous se trouve le schéma qui décrit cette architecture.
On remarque que le contrôle de cette entité se fait grâce au sous-bloc FSM_KeyExpansion qui envoie un signal d'enable au Counter et au Registre pour qu'ils fonctionnent.

Le Counter permet de se situer au niveau des rounds dans lesquels on se trouve. Elle permet au FSM_KeyExpansion de terminer le processus à la dernière ronde.

KeyExpansion fait simplement le calcul de la prochaine clé de ronde et le registre stocke celui-ci lors d'un front montant de l'horloge.

Quant au MUX, celui-ci est présente afin de prendre en compte la situation initiale ou la clé est key_i.

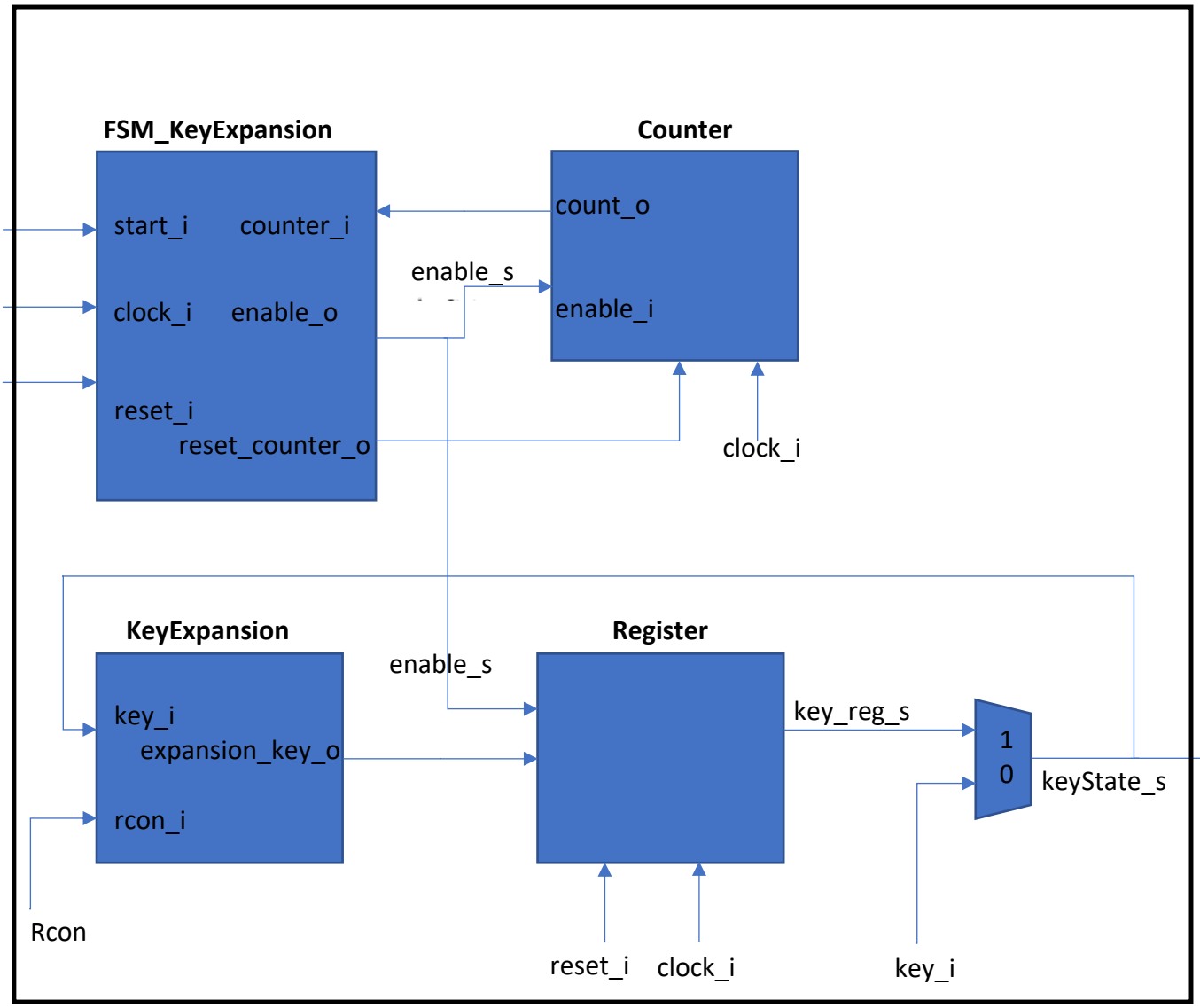


Schéma détaillé de l'architecture de KeyExpansion_I_O

ii) Test du bloc de génération de clé : KeyExpansion_I_O

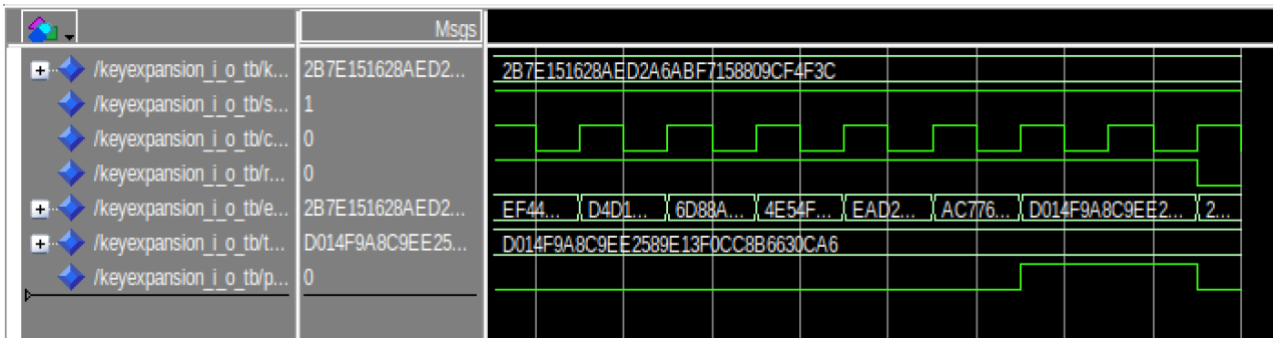
Pour le test bench de KeyExpansion_I_O, on met en entrée la clé primaire :

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

Pour vérifier que les clés en sorties sont correctes il suffit de vérifier que la dernière clé en sortie est correcte :

d0 14 f9 a8 c9 ee 25 89 e1 3f 0c c8 b6 63 0c a6

Voici le chronogramme du test bench :



On remarque que le bloc se met bien en fonctionnement lorsque start_s passe à un 1 (reset_s est à 1 aussi). A chaque front montant de l'horloge correspond une clé. Comme à la fin pass_fail_s passe à 1 la dernière clé est bien correcte et le bloc passe bien à l'état stop à partir de là jusqu'à l'appui du signal reset_s qui fait redémarrer le fonctionnement.

On en déduit donc que KeyExpansion_I_O, le bloc de génération de clés fonctionne correctement.

Jusqu'ici ont été présentés le fonctionnement des différents composants de l'AES et de son bloc de génération de clés. Nous pouvons maintenant nous intéresser au fonctionnement global de l'AES et aux interactions entre les divers sous-blocs de celui-ci.

iii) Difficultés rencontrées et solutions

```
begin

    --We initialize the RCON first row ( the last isn't used but it's there isn't an error during a compilation)
    rcon_array<=(X"01",X"02",X"04",X"08",X"10",X"20",X"40",X"80",X"1b",X"36",X"00");
```

J'ai rencontré une difficulté lors de l'emploi de la matrice Rcon. Dans le code VHDL employé le choix est faite de représenter la matrice Rcon à travers un vecteur contenant uniquement la première ligne. Des erreurs de longueur de vecteurs apparaissaient dans le code bien que le vecteur Rcon était bien dimensionné. Ceci était due au fait que le curseur donné dans le code avançait encore d'un cran à la fin du programme et dépassait donc la taille du vecteur mais faisait quand même appelle à ce vecteur.

Pour résoudre ce problème, j'ai artificiellement ajouté un dernier élément au vecteur Rcon : X"00", qui n'a aucun impact sur le fonctionnement du code mais qui enlève l'erreur en question.

III- Fonctionnement de l'ensemble

a) Description

Le fonctionnement global de l'AES prend tous les sous-blocs que nous avons précédemment développé :

- KeyExpansion_I_O
- FSM_AES (déjà développé dans l'énoncé)
- AESRound
- Counter

Le Schéma de la composition de l'entité AES est donné ci-après :

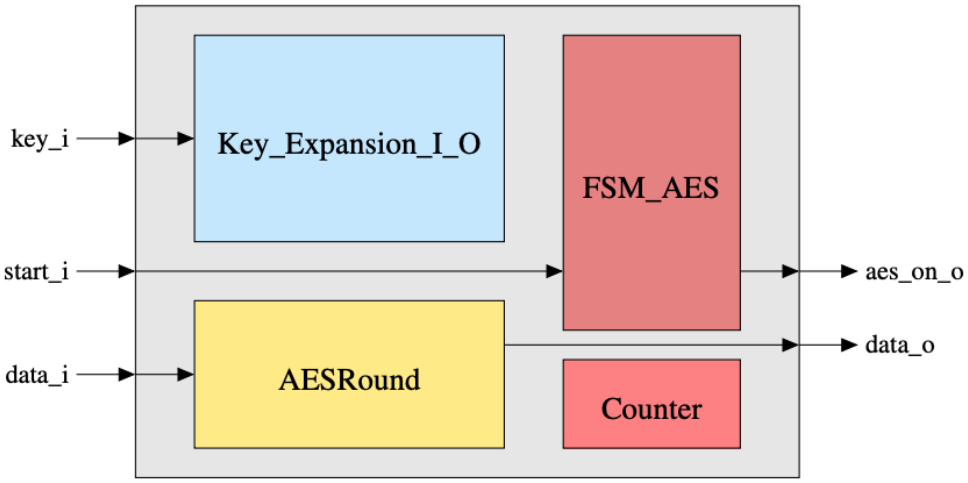


Schéma du modèle de l'AES (2)

Le fonctionnement du code VHDL est le suivant :

Comme indiqué dans le I-e), en partant de l'état initial, on exécute l'AESRound du round 0 au round 10. L'AESRound donne en sortie data_o qui correspond au message chiffré par l'entité.

À chaque tour, l'AESRound requiert une clé de ronde, celui-ci est fournie par l'entité Key_Expansion_I_O à l'AESRound.

Le fonctionnement et la temporalité de l'AES est géré grâce à la machine à états finis FSM_AES. Celui-ci se sert du Counter pour repérer les fronts montants de l'horloge et détermine dans quel état se trouve l'AES (hold, init, round0, roundn, lastround, done).
En fonction de cela il fournit des commandes au KeyExpansion_I_O (start_key_expander_o, ...) ou à l'AESRound (enableMixColumns_o, ...).

Voici un schéma du cours qui résume ce fonctionnement :

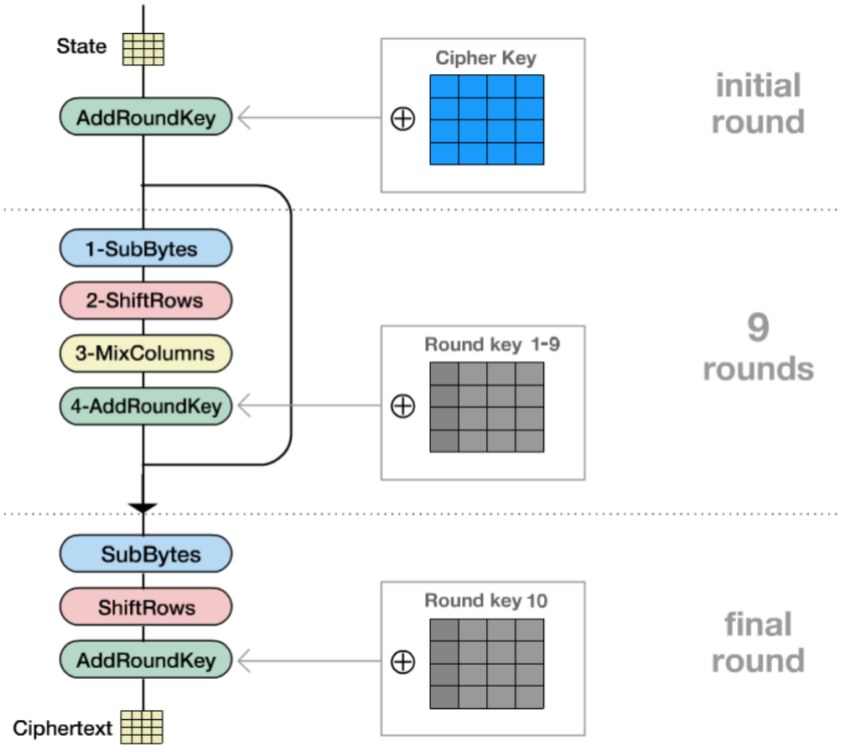


Schéma de l'AES (2)

b) Test Bench

Pour le test bench final, nous allons voir si à l'aide d'une clé et d'un message donnée, l'entité AES sort bien le bon message chiffré.

Voici le message à chiffrer :

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

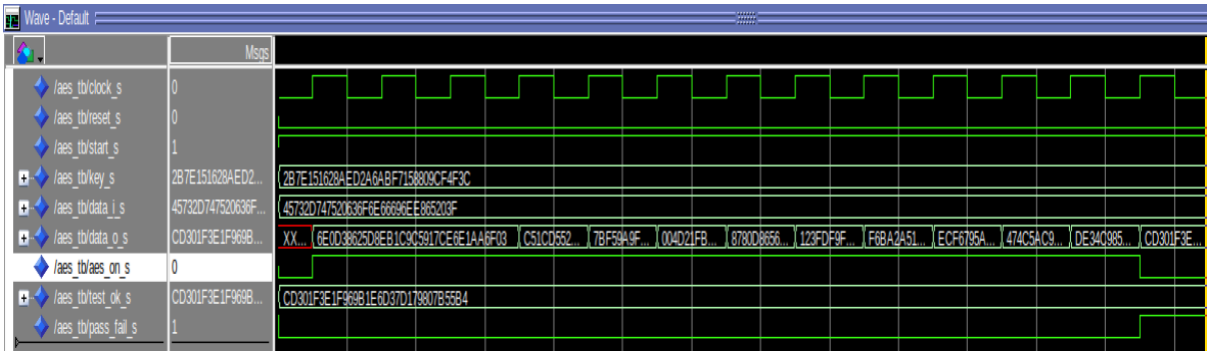
La clé employée est la suivante :

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

Le message chiffré attendu est :

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´

Voici le résultat du test bench sur ModelSim :



On remarque ainsi que data_o_s évolue à chaque front montant et qu'à la fin de la ronde 10, la variable aes_on_s passe à 0, ce qui veut dire l'opération de génération du message chiffré est terminé et comme pass_fail_s passe à 1, on en déduit qu'on a bien le résultat que l'on attendait.

On peut donc conclure que le modèle de l'AES fonctionne.

Conclusion

En conclusion, nous avons bien réalisé une modélisation de l'AES en employant le langage VHDL. Nous avons analysé et testé chacun des sous-entités de l'AES et avons étudié les interactions qui existent entre ces entités pour finalement étudier le fonctionnement détaillé du système dans sa globalité. Bien que nous ayons rencontré certains problèmes nous avons été capables d'interpréter ces erreurs et de les résoudre.

Il serait intéressant de pouvoir appliquer à ce système les mécanismes de défense que nous avons vu en cours de Sécurité des Systèmes Embarqués afin de pouvoir notamment se protéger contre les attaques par canaux cachés.

Bibliographie

1. **Wikimédia.** Advanced Encryption Standard. *Wikipédia.* [En ligne] https://fr.wikipedia.org/wiki/Advanced_Encryption_Standard.
2. **Jean-Max Dutertre, Olivier Potin, Guillaume Reymond, Jean-Baptiste Rigaud.** *An Introduction to VHDL.* Campus Georges Charpak : EMSE - ISMIN, 2020.
3. **Dutertre, Jean-Max.** *Projet Conception d'un système Numérique.* Campus Georges Charpak : EMSE - ISMIN, 2020.