École des Mines de Saint-Étienne

# Rapport projet PCSN

Lucas MAISONNAVE 2A

Lucas Maisonnave
25/12/2020

# I– Introduction

Le projet PCSN a pour objectif de réaliser un modèle VHDL du chiffrement AES. Le bloc AES doit être capable de chiffrer une donnée sur 128 bits et nécessite une clé de même taille pour encrypter la donnée. De plus, le modèle doit fonctionner sur front d'horloge et doit avoir un pin de start et de restart. Finalement, le bloc AES doit suivre le schéma structurel suivant :

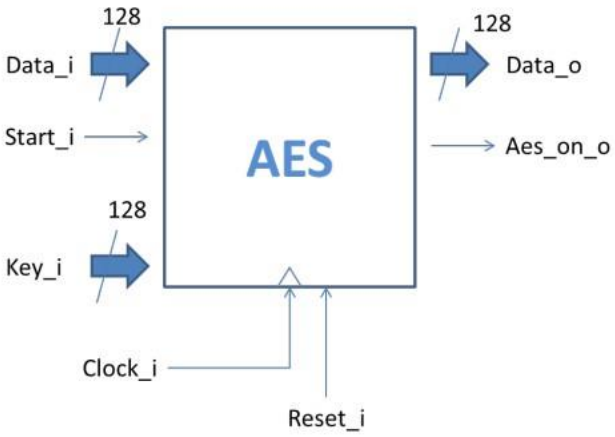


*Figure 1 - Schéma structurel de l'AES*

Pour réaliser ce projet, il est nécessaire de décomposer ce bloc en plusieurs sous-blocs répondant à des objectifs divers qui seront développés par la suite. De plus, le bloc AES suivra le protocole de chiffrement standard, 1 ronde avec AddRoundKey seulement, 9 rondes de SubByte, ShiftRow, MixColumns et AddRoundKey et 1 ronde de SubByte, ShiftRow et AddRoundKey. Pour cela, il sera nécessaire d'implémenter deux machines à états finis : la première pour compter le nombre de rondes effectuées et la deuxième pour générer des clés à chaque ronde. Après avoir implémenté chaque bloc, on obtient le résultat suivant sur la simulation du test bench du bloc AES :

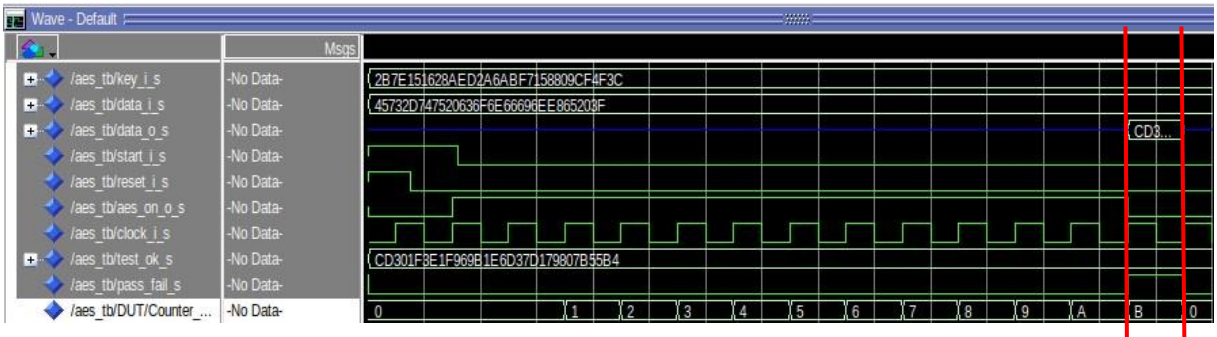


*Figure 2 - Simulation de l'AES*

Sur cette simulation on observe le bon fonctionnement de l'AES qui renvoie la donnée correctement encryptée comme on peut le constater avec la variable pass_fail à 1 à l'issue des 10 rondes. On peut aussi observer les différentes étapes d'encryptage de la donnée et des générations des clés :

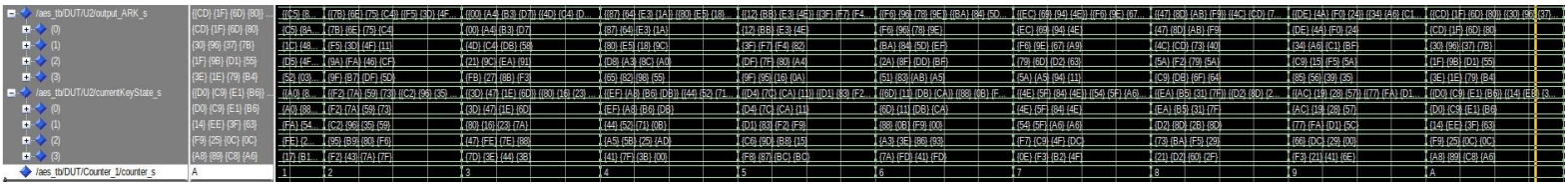


*Figure 3 - Donnée et clé à chaque ronde*

On voit bien que les clés sont générées correctement à chaque étape et que la donnée correspond bien à ce qui est spécifié dans le sujet.

Maintenant, nous pouvons nous intéresser à chaque bloc et vérifier leur bon fonctionnement.

# II– Sous blocs et fonctionnement

Nous nous intéresserons tout d'abord aux blocs dont l'implémentation et la compréhension sont les plus simples, à savoir SubBytes, ShiftRows, MixColumns et Counter.

Pour **SubBytes,** il s'agit simplement de l'implémentation d'un tableau qui va échanger les octets de la donnée d'entrée :

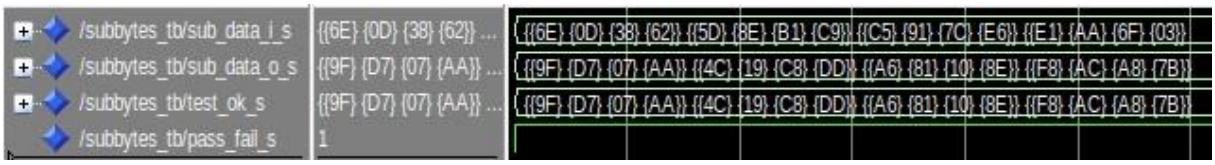


*Figure 4 - Test bench SubBytes*

Pour **ShiftRows,** on décale chaque ligne un nombre de fois égale au numéro de la ligne :

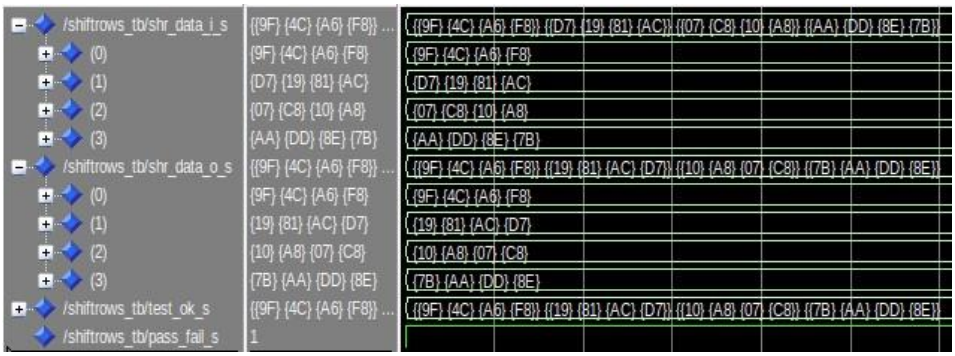


*Figure 5 - Test bench de ShiftRows*

Ici on observe bien le décalage de chaque ligne ce qui correspond au résultat obtenu sur la variable pass_fail.

Pour **MixColumns** on doit appliquer le produit matriciel de chaque colonne avec une matrice prédéfinie : $\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$

Pour cela, on doit calculer le produit par 2 et par 3 pour obtenir la donnée modifiée en sortie. Il est aussi important d'implémenter l'entrée *enable_MixColumns* pour autoriser ou non l'appel de la fonction :


*Figure 6 - Test bench MixColumns, enable = 1*

La fonction MixColumns fonctionne donc bien dans ces conditions. On vérifie à présent que le bloc renvoie la donnée en entrée quand *enable_MixColumns* est à l'état bas :

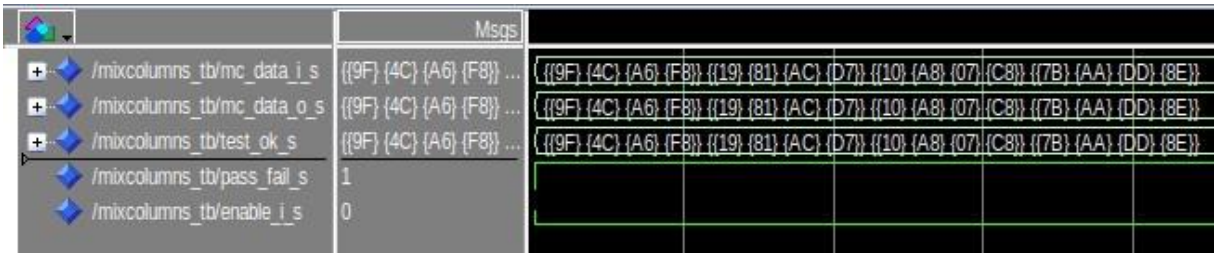


*Figure 7 - Test bench MixColumns, enable = 0*

On s'intéresse maintenant au bloc **Counter** qui doit compter le nombre de fronts montant de la *clock*, ce bloc doit aussi posséder un pin *enable* qui doit pouvoir stopper le décompte :

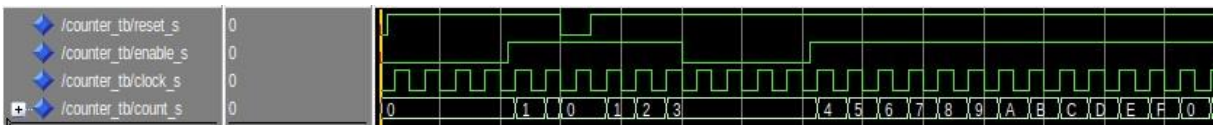


*Figure 8 - Test bench Counter*

Sur ce chronogramme, on observe bien toutes les fonctionnalités du compteur, il compte jusqu'à 15 (X ''F'') et se remet à 0, il s'arrête quand *enable* est à 0, et il se remet à 0 quand *reset* est à 0.

Maintenant que nous avons vu ces 4 blocs, nous pouvons les utiliser pour créer de nouveaux blocs ayant des fonctions plus complexes. Nous verrons dans un premier temps AddRoundKey et AESRound.

Dans le protocole AES, la fonction **AddRoundKey** intervient à la sortie de la fonction MixColumns et a pour objectif d'encrypter cette donnée avec une clé envoyée par le bloc KeyExpansion_I_O qui sera détaillé plus tard. Pour encrypter la donnée, la fonction AddRoundKey applique simplement un XOR entre chacun des octets qui la composent.

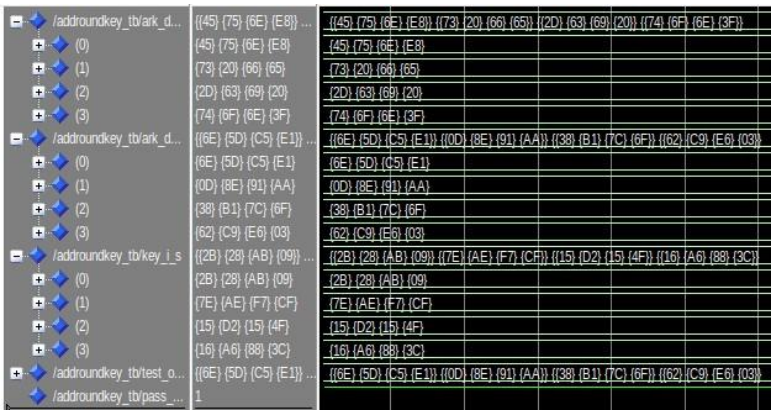


*Figure 9 - Test bench AddRoundKey*

Ici, on utilise les données de référence du sujet pour la ronde 1 et on obtient une donnée correctement encryptée en sortie. Ensuite, le bloc **AESRound** est composé des blocs SubBytes, ShiftRows, MixColumns, AddRoundKey ainsi que d'un MUX et d'un Registre, connectés comme suit :

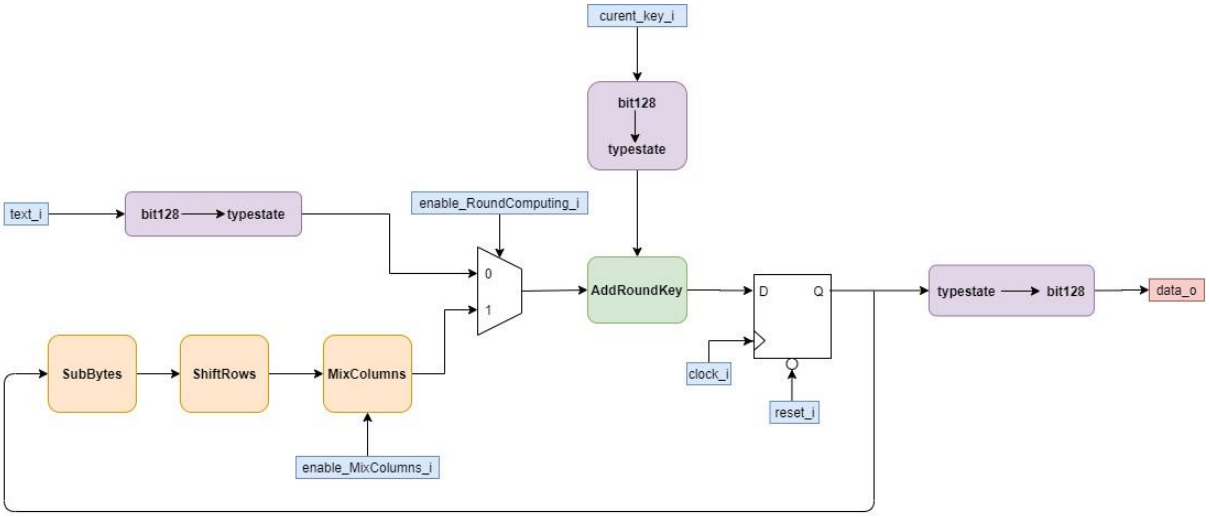


*Figure 10 - Schéma AESRound*

On relie tous les blocs de cette façon et on met en place un test bench pour vérifier que la donnée est bien encryptée à chaque ronde :

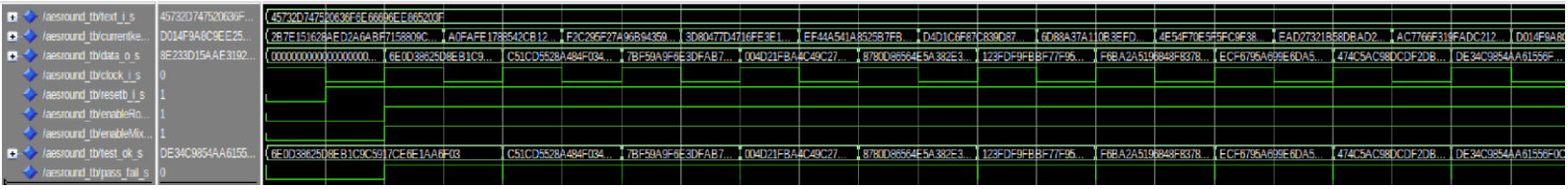


*Figure 11 - Test bench AESRound*

Sur ce chronogramme, on observe bien qu'à chaque ronde le bloc renvoie la donnée correctement encryptée et que les variables *enable* permettent d'encrypter *text_i* à la première ronde.

Il faut maintenant gérer la génération des clés à chaque ronde, pour cela on met en place le bloc **KeyExpansion_I_O** qui fait appel aux blocs KeyExpansion, KeyExpansionFSM et Counter, d'un registre et d'un MUX, connectés comme ceci :

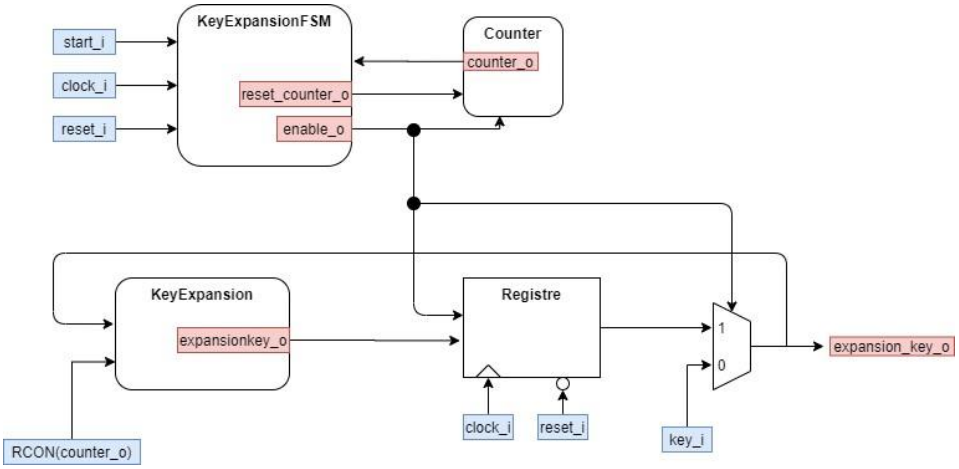


*Figure 12 - Schéma KeyExpansion_I_O*

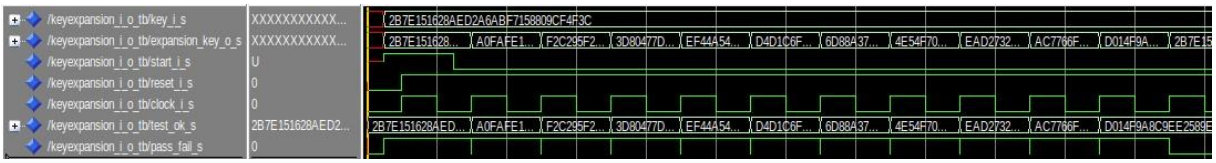

*Figure 13 - Test bench KeyExpansion_I_O*

On observe bien que la sortie du bloc renvoie sur front montant les différentes clés pour chaque ronde. On remarque bien que la première clé renvoyée est bien la clé mise en entrée et qu'au bout de 10 rondes la clé de sortie est de nouveau égale à la clé d'entrée donc la FSM fonctionne correctement.

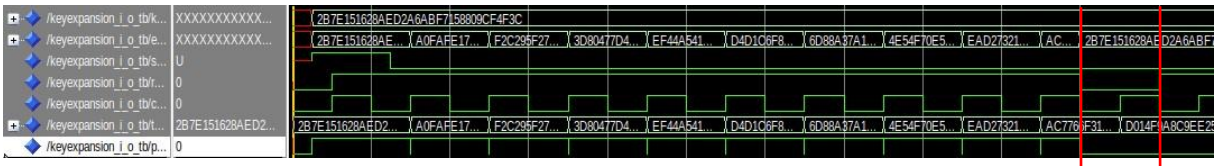


*Figure 14 - Test bench KeyExpansion_I_O reset*

On vérifie ici que le reset fonctionne correctement. On voit bien que la clé de sortie est à nouveau égale à l'entrée.

On va maintenant s'intéresser aux sous-blocs de KeyExpansion_I_O et vérifier leur bon fonctionnements. La machine à états **KeyExpansionFSM** est une machine de Moore qui contrôle la génération des clés et doit donc permettre de renvoyer *key_i* à la première ronde et se réinitialiser au bout de 10 rondes. Cette FSM suit le schéma d'état suivant :

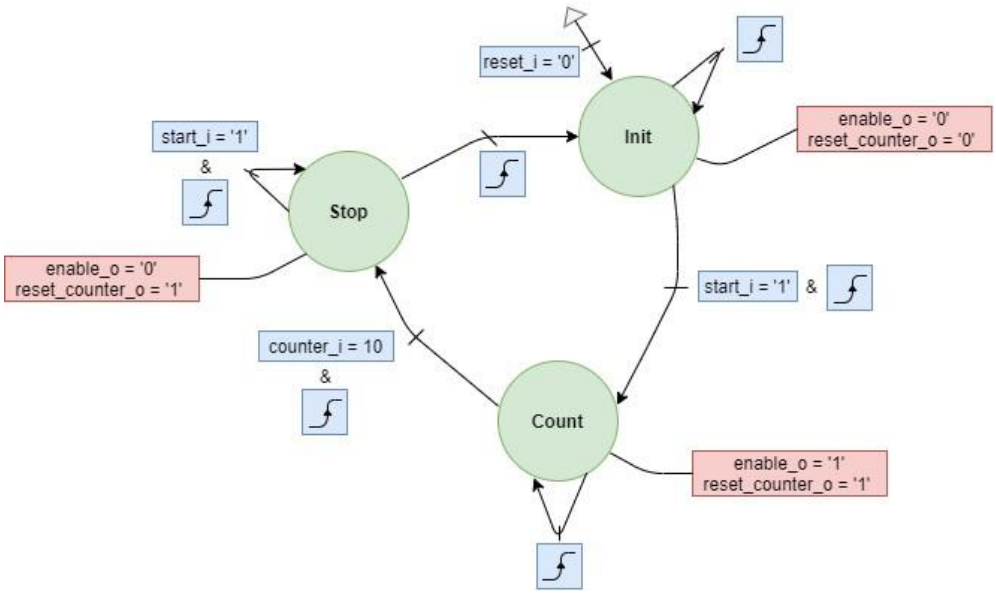


*Figure 15 - Schéma KeyExpansionFSM*

Cette FSM contient donc 3 états dont les transitions se font sur front montant de l'horloge. L'état Init est un état d'attente de start où l'on n'autorise pas la génération d'une nouvelle clé et où le reset est à l'état dominant. Puis le décompte est lancé, on passe dans l'état Count : le processus de génération de clé démarre, on autorise les nouvelles clés en sortie (*enable_o* = '1') et on ne reset plus le compteur. Lorsque la variable counter_i est à 10 (X « A »), la FSM détecte la fin de la génération des 10 rondes. Elle passe alors à l'état Stop, un état d'attente du passage à l'état bas de la variable start_i. Quand start_i passe à l'état bas, la FSM est réinitialisée, évitant ainsi une boucle infinie.

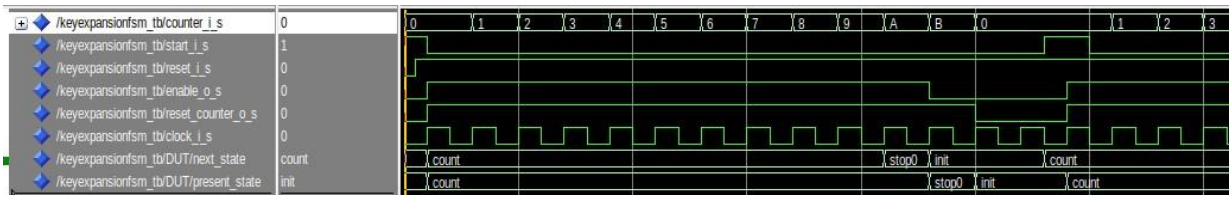


*Figure 16 - Test bench KeyExpansionFSM*

Pour cette simulation, en plus du bloc KeyExpansionFSM, on ajoute un compteur connecté à la machine à état. On observe donc le fonctionnement de la FSM avec les transitions d'états tout au long du processus. On peut noter que le compteur s'arrête à 11 mais que la variable *enable_o* est bien à 0. Ainsi, l'excès du compteur n'a pas de répercussions sur le processus. On peut constater qu'une nouvelle impulsion de *start_i* redémarre le processus et le compteur repart comme voulu. Le bloc KeyExpansionFSM fonctionne donc correctement.

Pour gérer la génération de nouvelles clés, il faut s'intéresser au bloc **KeyExpansion** qui prend en entrée une clé *key_i* et une colonne *RCON_i* de la matrice RCON : et renvoie la nouvelle clé *w_o*, en suivant le procédé ci-dessous (le « + » étant la fonction XOR) :

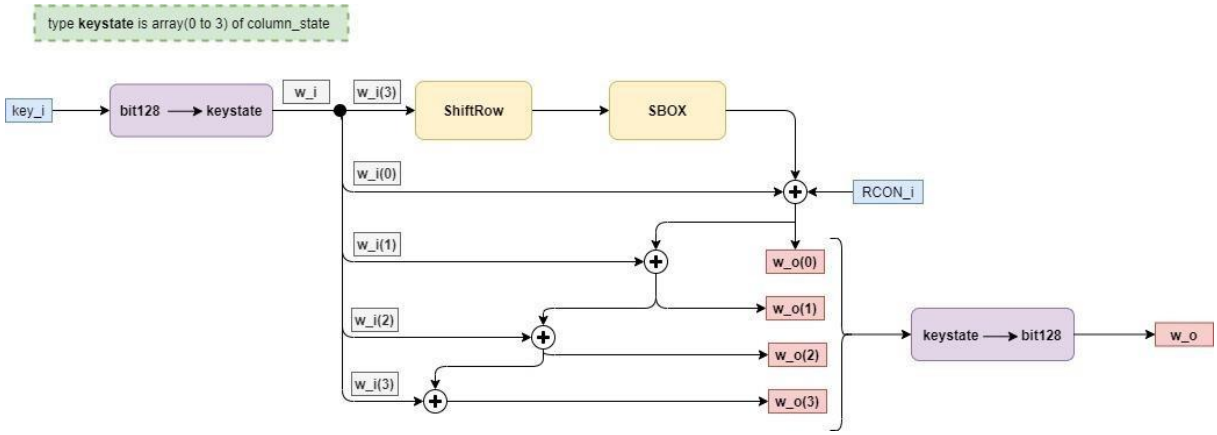


*Figure 17 - Schéma KeyExpansion*

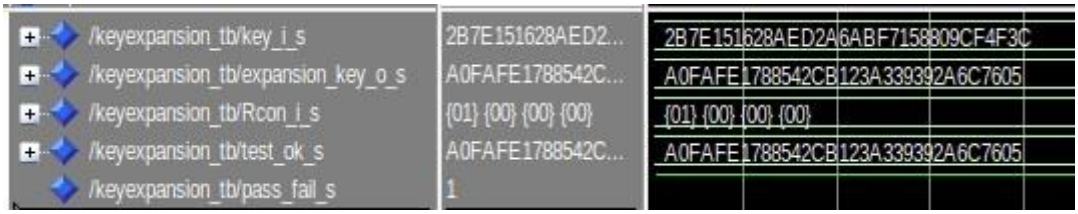


*Figure 18 - Test bench KeyExpansion*

Avec cette simulation on vérifie que la clé X''2b7e151628aed2a6abf7158809cf4f3c'' de la ronde 0 devient bien la clé X''a0fafe1788542cb123a339392a6c7605'' de la ronde 1 en ayant inséré la première colonne de la matrice RCON.

Ainsi nous avons vérifié le fonctionnement de tous les sous-blocs de l'AES.

# III – Conclusion

Pour conclure, le modèle implémenter du chiffrement AES fonctionne correctement et nécessite 12 coups d'horloge pour être achevé, ce qui en fait un processus très rapide malgré la complexité et le nombre de sous-blocs qui le composent.

On pourrait cependant imaginer des pistes d'améliorations notamment sur les machines à états finis qui pourraient ne former qu'une. En effet on pourrait imaginer une seule et unique FSM pour tout le système plutôt que de diviser des tâches qui sont presque similaires, ce qui permettrait de simplifier le modèle et d'avoir moins de sous-blocs. On pourrait aussi se poser la question de la pertinence d'utiliser un langage de représentation comme VHDL pour un tel projet. En effet nous avons

vu en cours de cryptographie qu'il était possible de cracker des systèmes sécurisés en prélevant directement la consommation énergétique du système ET en ayant connaissance du fonctionnement du circuit électronique. Or utiliser du VHDL a pour objectif de créer un système embarqué composé de blocs répondant à des fonctions bien défini. N'est-ce pas dangereux de créer un système dont le support même est une potentielle faille à son fonctionnement ? Ne serait-il pas plus pertinent d'utiliser un langage de programmation standard comme le Python ou le C qui utilisent les processeurs et la RAM d'une carte électronique moins sensibles à cette attaque ?