Conception d'un Système Numérique

Modélisation SystemVerilog de l'algorithme de chiffrement Ascon-AEAD128

Rédigé :

Imane MOUMOUN

Sous la direction de :

Jean-Max DUTERTRE

Sommaire

Introduction

ASCON-AEAD128 est un algorithme de chiffrement authentifié avec données associées, conçu pour être à la fois léger, rapide et efficace en termes de ressources. Sélectionné comme standard du NIST pour la cryptographie légère, il garantit non seulement la confidentialité des messages, mais aussi leur authenticité grâce à un tag qui est une sorte de signature permettant de vérifier si le message reçu est bien celui envoyé.

Ce projet consiste en la modélisation en SystemVerilog de l'algorithme de chiffrement Ascon-AEAD128. Il est particulièrement pertinent d'utiliser un langage de description matérielle comme le SystemVerilog pour concevoir un circuit dédié, car cela permet non seulement d'améliorer les performances de l'algorithme, mais aussi de libérer des ressources processeur pour d'autres tâches. En réalisant cette implémentation, nous cherchons également à approfondir notre compréhension du SystemVerilog et de son fonctionnement, notamment la différence avec un langage de programmation.

Dans le cadre de notre projet, nous utiliserons le logiciel ModelSim de Mentor Graphics pour simuler nos différents composants et notre circuit final.

Le projet sera composé du présent rapport, mais également d'une archive au format .zip contenant les différents codes .sv et le script de compilation de ces codes.

Bibliothèques utilisées

Pour ce projet, plusieurs fichiers nous ont été fournis et d'autres ont été créer pour faciliter la mise en œuvre du projet. Ces fichiers sont les suivants :

- ***ascon_pack.sv :*** Ce fichier contient la déclaration d'un tableau de 5 lignes de 64 bits ainsi que d'autres tableaux qui nous seront utiles dans les différents modules de ce projet.
- ***mux_state.sv :*** Ce fichier modélise le comportement d'un multiplexeur à 2 entrées.
- ***register_w_en.sv :*** c'est un registre qui va nous permettre de stocker temporairement des valeurs, notamment le cipher et le tag.
- ***reg_state.sv :*** Ce module représente un registre comme *register_w_en.sv* à l'exception que les entrées et sorties ont un type différent.
- ***permutation_simple.sv* :** C'est le module qui représente la permutation simple sans XOR.
- ***xor_begin.sv et xor_end.sv :*** Ces fichiers permettent d'effectuer une opération de type XOR sur différents registres de l'état courant et ont des conditions d'activation différentes.
- ***permutation_xor.sv* :** Il s'agit de la permutation simple à laquelle sont ajoutés les deux opérations *xor_end* et *xor_begin.*
- ***permutation_xor_reg.sv* :** Ce module représente la permutation finale, à laquelle sont ajoutés deux registres (*register_w_en.sv*) permettant de mémoriser le texte chiffré (cipher) et le tag.
- ***compteur_simple_init.sv et compteur_double_init.sv :*** Ces modules permettent d'implémenter des compteurs pour compter le nombre de rondes.
- ***fsm_moore.sv :*** Ce module représente la machine d'états finis qui pilotent les signaux de contrôle de l'algorithme, ainsi que le compteur de rondes.
- **ascon_top.sv :** Il s'agit du module final qui représente l'implémentation complète de l'algorithme de chiffrement ASCON-128

Algorithme Ascon-AEAD128

1. Fonctionnement de l'algorithme Ascon-AEAD128

La version proposée pour ce projet est une version simplifiée de l'algorithme original, afin d'être réalisable dans la durée du projet

Dans cette version, l'algorithme opère sur un état courant (ou state S) de 320 bits, qui est mis à jour avec une opération appelée permutation comprenant 8 ou 12 itérations, respectivement notée p^8 et p^{12}. Les itérations sont aussi appelées rondes.

L'état S de 320 bits est divisé en 5 registres Si de 64 bits chacun

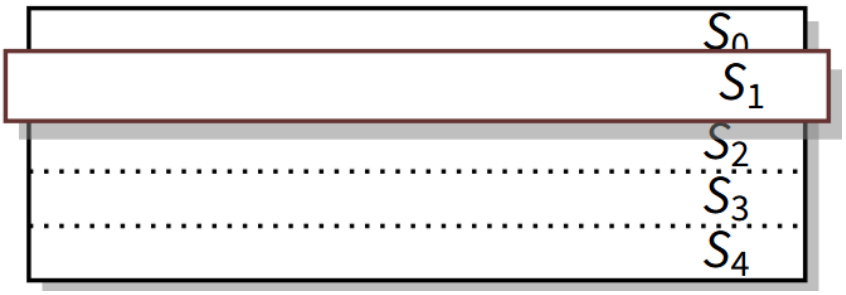


Figure 1 : Représentation de l'état S en 5 registres de 64 bits

L'état peut aussi être interprété comme 64 colonnes de 5 bits chacune

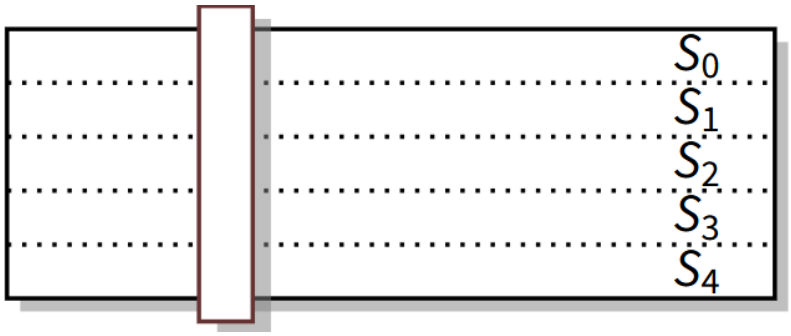


Figure 2 : Représentation de l'état S en 64 colonnes de 5 bits

Dans ce projet, les notations utilisées sont les suivantes :

K : Clé secrète de 16 octets soit 128 bits

N : Nombre arbitraire de 16 octets

T : Tag de 16 octets

P : Texte clair de 47 octets

C : Texte chiffré de 47 octets

A : Données associées traitées par bloc de 16 octets (12 octets + 4 octets de padding)

IV :Vecteur d'initialisation de 8 octets

Le processus de chiffrement se décompose en quatre étapes :

1. La phase d'initialisation de l'état S en fonction d'un vecteur d'initialisation IV donné, de la clé secrète K et du nombre arbitraire N.

2. Le traitement des données associées A, consistant à additionner un bloc de données de 128 bits.

3. Le traitement du texte en clair P, par blocs de données de 128 bits, avec l'addition des blocs de texte en clair dans l'état courant, ainsi que la récupération des blocs de texte chiffré.

4. La finalisation pour terminer le calcul du tag T utilisé pour l'authentification.

Après chaque bloc ajouté à l'état S (à l'exception du dernier bloc de texte clair), la permutation p^8 est appliqué à l'état S. Pendant les phases d'initialisation et de finalisation, la permutation p^{12} est utilisée, contenant 12 rondes au lieu de 8.

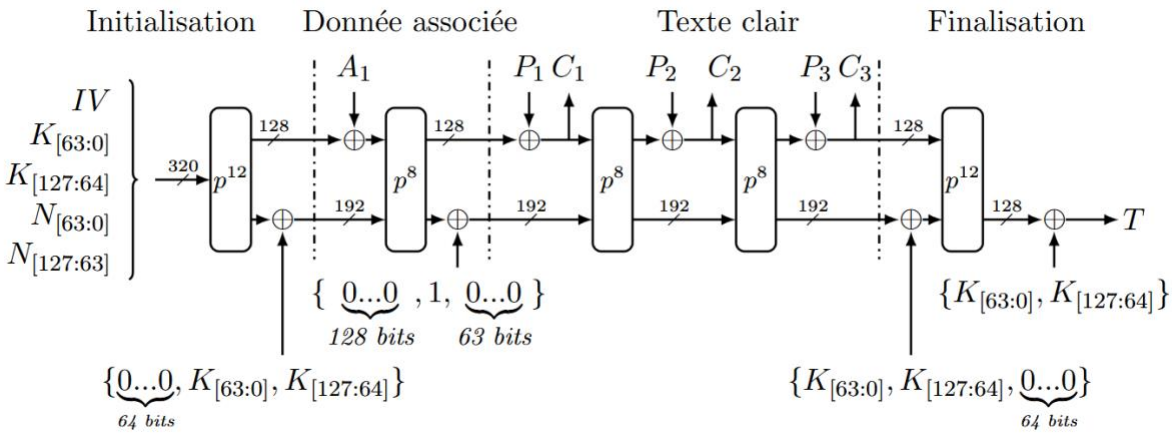


Figure 3 : Schéma du chiffrement Ascon-AEAD128

Ainsi, le module final ascon_top comportera les entrées / sorties suivantes :

- Une horloge clock_i
- Un signal d'initialisation resetb_i, actif à l'état bas
- Une entrée data_i de 128 bits
- Une entrée data_valid_i indiquant la présence d'une donnée valide sur data_i
- Une entrée pour la clé key_i de 128 bits
- Une entrée pour le nombre arbitraire nonce_i de 128 bits
- Une entrée start_i pour commencer le chiffrement
- Une sortie cipher_o sur 128 bits
- Une sortie cipher_valid_o indiquant la validité de la sortie cipher_o
- Une sortie tag_o sur 128 bits
- Une sortie end_o indiquant la fin du chiffrement du message, et la validité de la sortie tag_o

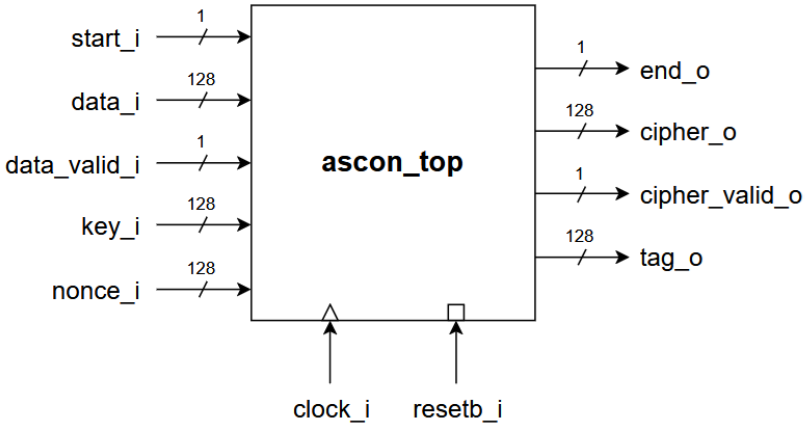


Figure 4 - Entrées/Sorties du module ASCON 128

2. Modélisation des fonctions élémentaires

2.1. L'addition de constante

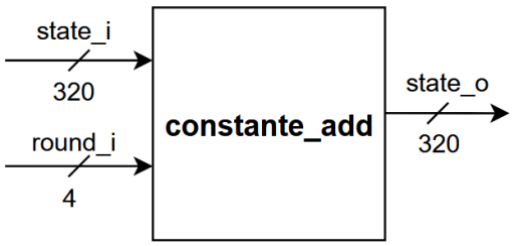


Figure 4 : Module addition de constante

L'addition de constante marque le début de la permutation. Elle consiste à ajouter une constante de ronde c_r au registre S_2 de l'état S au cours de la ronde i, tel que $S_2 \leftarrow S_2 \oplus c_r$.

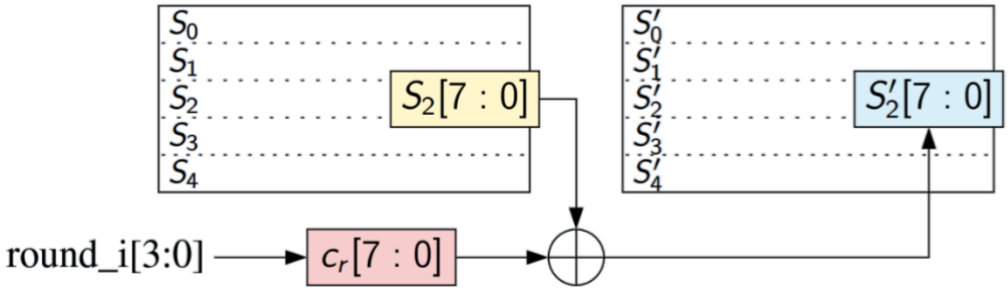


Figure 5 : Principe de l'addition de constante

Les constantes sont définies dans la table suivante :

Ronde r de p^{12}	Ronde r de p^8	Constante c_r
0		000000000000000000f0
1		000000000000000000e1
2		000000000000000000d2
3		000000000000000000c3
4	4	000000000000000000b4
5	5	000000000000000000a5
6	6	0000000000000000096
7	7	0000000000000000087
8	8	0000000000000000078
9	9	0000000000000000069
10	10	000000000000000005a
11	11	000000000000000004b

Table 1 : Liste des constantes pour p^8 et p^{12}

Dans mon implémentation, j'ai utilisé le tableau *round_constant*, défini dans le fichier *ascon_pack*. Ce tableau fournit la valeur appropriée de la constante c_r en fonction du numéro de la ronde.

Dans ce module, les registres 0, 1, 3 et 4 de l'état d'entrée *state_i* ne subissent aucune modification et sont directement recopiés dans l'état de sortie *state_o*. Seul l'octet de poids faible du registre 2 est modifié. Cette modification consiste en un XOR (ou exclusif) entre l'octet de poids faible du registre 2 de l'état d'entrée avec une constante *round_constant*, déterminée en fonction de la valeur de *round_i*.

Afin de vérifier le bon fonctionnement de ce module, j'ai conçu un testbench dans lequel j'ai spécifié les différentes valeurs d'entrée du système, à savoir state_i et round_i, et qui me permet d'observer la sortie state_o.

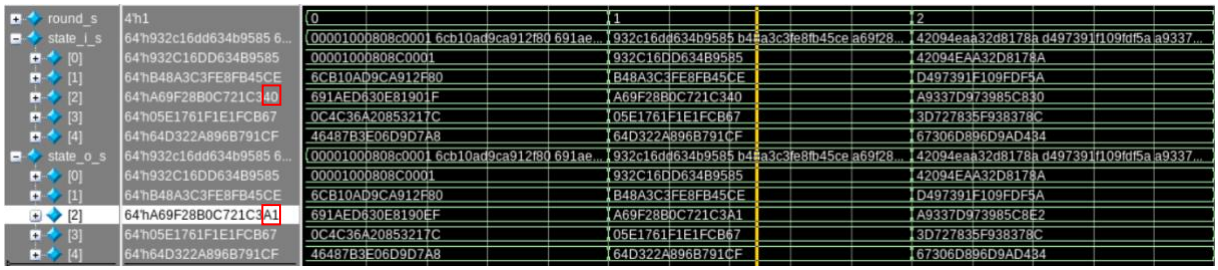


Figure 6 : Simulation du module constante_add

Comme illustré dans la figure 7, on observe bien une modification de l'octet de poids faible du registre state_o_s[2]. Par exemple, lors de la première ronde, pour une entrée state_i_s[2]=64'ha69f28b0c721c340, la sortie obtenue est state_o_s[2]=64'ha69f28b0c721c3a1, ce qui correspond à la valeur attendue. Le bon fonctionnement du module est ainsi confirmé.

2.2. La couche de substitution

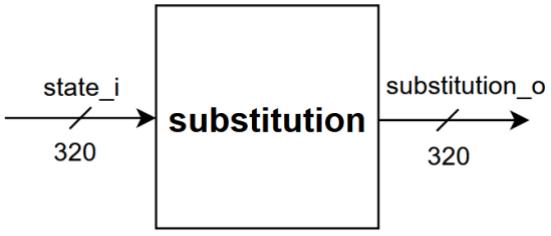


Figure 7 : Module de couche de substitution

La couche de substitution est la 2ème étape de la permutation. Elle modifie l'état S en effectuant une substitution de 5 bits par colonne via la table de substitution suivante :

x	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
Sbox(x)	04	0B	1F	14	1A	15	09	02	1B	05	08	12	1D	03	06	1C	1E	13	07	0E	00	0D	11	18	10	0C	01	19	16	0A	0F	17

Table 2 : Table de substitution utilisée pour le projet

Pour effectuer la substitution sur chaque colonne, j'ai implémenté un module intermédiaire sbox, qui prend une entrée de 5 bits et retourne une valeur substituée.

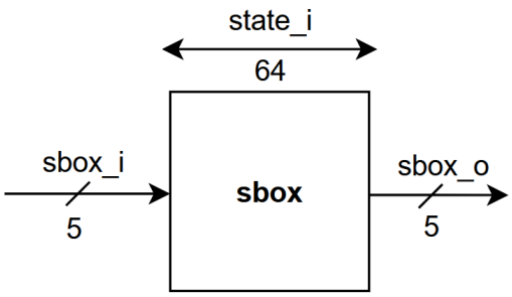


Figure 8 : Module intermédiaire sbox

Tout comme pour l'addition de constante, j'ai utilisé un tableau présent dans ascon_pack (*sbox_t*) qui contient les valeurs de substitution.

Ainsi, pour implémenter cette couche de substitution, il suffit de parcourir l'état d'entrée state_i à l'aide d'une boucle. À chaque itération, la sbox substitue les bits de la ligne correspondante par les nouvelles valeurs selon la transformation suivante : $\{S_0[i], S_1[i], S_2[i], S_3[i], S_4[i]\} \leftarrow \text{Sbox}(\{S_0[i], S_1[i], S_2[i], S_3[i], S_4[i]\})$, avec i∈[1; 64], S0[i] représente le bit de poids fort et S4[i] le bit de poids faible.

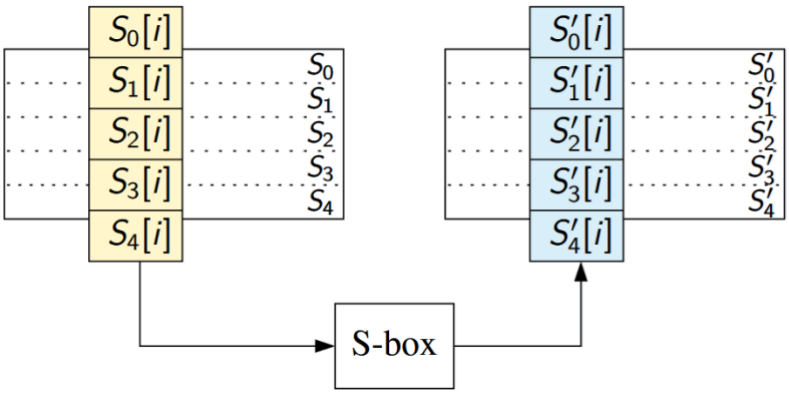


Figure 9 : Principe de la couche de substitution

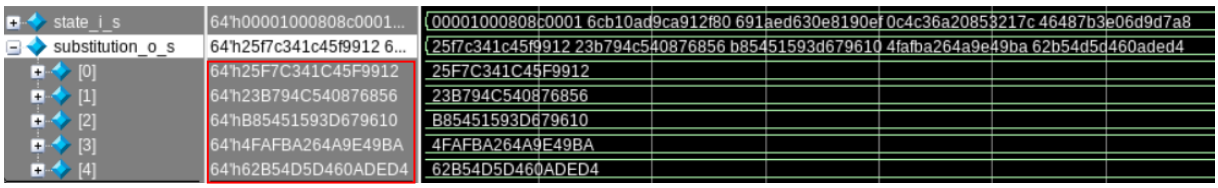


Figure 10 : Simulation du module substitution

Pour vérifier la validité de ce module, on injecte en entrée le résultat de l'addition de constante, et l'on s'attend à retrouver en sortie la valeur fournie dans la couche de substitution. Effectivement, comme le montre la figure 10, les résultats obtenus sont conformes aux valeurs attendues, ce qui confirme le bon fonctionnement et la validité du module implémenté.

2.3. La couche de diffusion

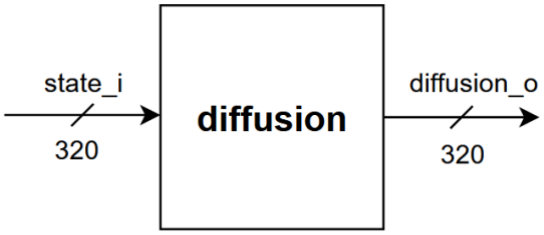


Figure 11 : Module de couche de diffusion linéaire

La couche de diffusion linéaire constitue la troisième et dernière étape du processus de permutation. Elle transforme l'état courant S en appliquant une opération de diffusion à chacune de ses lignes de 64 bits ou registres S_i. Cette transformation consiste à effectuer une rotation cyclique sur chaque ligne suivie d'une opération de type XOR, modifiant ainsi leur contenu.

Plus précisément, cette couche applique une fonction linéaire notée Σ_i à chaque registre d'entrée, selon les formules suivantes :

$$S_0 \leftarrow \Sigma_0(S_0) = S_0 \oplus (S_0 \gg 19) \oplus (S_0 \gg 28)$$

$$S_1 \leftarrow \Sigma_1(S_1) = S_1 \oplus (S_1 \gg 61) \oplus (S_1 \gg 39)$$

$$S_2 \leftarrow \Sigma_2(S_2) = S_2 \oplus (S_2 \gg 1) \oplus (S_2 \gg 6)$$

$$S_3 \leftarrow \Sigma_3(S_3) = S_3 \oplus (S_3 \gg 10) \oplus (S_3 \gg 17)$$

$$S_4 \leftarrow \Sigma_4(S_4) = S_4 \oplus (S_4 \gg 7) \oplus (S_4 \gg 41)$$

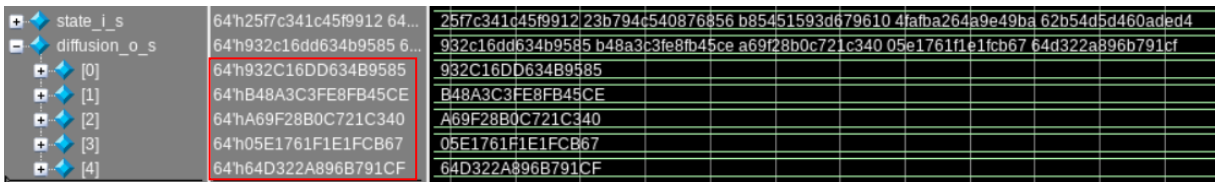


Figure 12 : Simulation du module diffusion

Pour vérifier la validité de ce module, on injecte en entrée le résultat de la couche de substitution, et l'on s'attend à retrouver en sortie la valeur attendue. Effectivement, comme le montre la figure 12, les résultats obtenus sont conformes aux valeurs attendues, ce qui confirme le bon fonctionnement et la validité du module implémenté.

3. Modélisation des permutations p8 et p12

Les composantes principales de l'algorithme Ascon-AEAD128 sont les deux permutations p^8 et p^{12} de l'état S sur 320 bits. En effet, la permutation repose sur l'application successive des différentes couches implémentées précédemment, selon la composition suivante :

$$p = p_C \circ p_S \circ p_L$$

où :

- p_C : correspond à l'addition de constante
- p_S : désigne la couche de substitution
- p_L : représente la couche de diffusion linéaire.

Il convient de noter que les permutations p^8 et p^{12} se distinguent uniquement par le nombre de rondes appliquées : 8 pour p^8 et 12 pour p^{12}.

Par souci de clarté et de modularité, j'ai structuré l'implémentation de la permutation en trois étapes. Tout d'abord, j'ai créé le module *permutation_simple* intégrant les transformations p_C, p_S, p_L pour le calcul des permutations, la mémorisation de l'état S par un registre de 320 bits et un multiplexeur pour la sélection de l'état à choisir. Ensuite, j'ai implémenté le module *permutation_xor_end* en y ajoutant l'opération XOR en aval de la permutation (*xor_end)*. Puis, j'ai implémenté le module *permutation_xor* en ajoutant l'opération XOR en amont de la permutation (*xor_begin).* Enfin, la dernière étape consiste à

implémenter la permutation finale, via l'ajout des deux registres qui mémorisent le texte chiffré *cipher_o* et le tag *tag_o*.

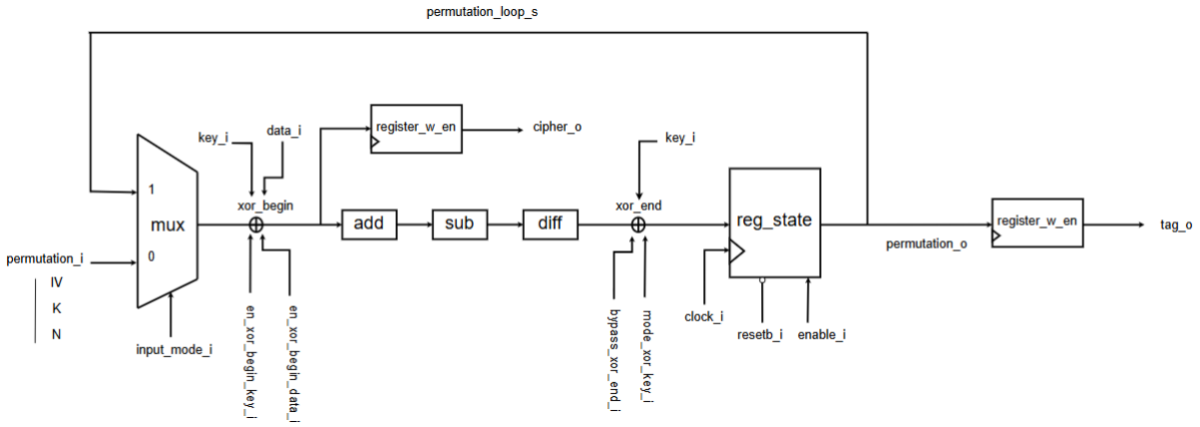


Figure 13 : Modélisation de la permutation finale avec les 2 xor

Le fonctionnement de la permutation est assez simple, lors de la 1ère ronde on sélectionne la valeur en entrée du multiplexeur et lui applique successivement différentes opérations comme 2 opérations de type XOR mais également les opérations développées plus tôt c'est-à-dire l'addition de constante, la substitution et la diffusion. La valeur finale est ensuite réinjectée dans le même circuit jusqu'à atteindre 8 ou 12 rondes. Les blocs *xor_begin* et *xor_end* sont des éléments essentiels de cette architecture. Placé juste après le multiplexeur, *xor_begin* permet d'introduire, selon le cas, soit la clé (*key_i*) soit les données d'entrée (*data_i*) dans le calcul. En phase de donnés associés et chiffrement, il applique un XOR avec les données, tandis qu'en phase de finalisation, il effectue un XOR avec la clé. Ce comportement est contrôlé via les signaux *en_xor_begin_data_i* et *en_xor_begin_key_s,* ce qui permet de désactiver le XOR si nécessaire. De la même manière*, xor_end* intervient à la fin du traitement. Il peut effectuer un XOR avec la clé ou simplement inverser un seul bit de l'état, selon le mode choisi. Le tout est piloté par les signaux *bypass_xor_end_i* et *mode_xor_key_i*.

L'implémentation en SystemVerilog n'est pas compliquée également. En effet, le module permutation repose sur l'ensemble des modules précédemment crées et ceux fournis. Il suffit alors d'instancier les différents modules et de les faire communiquer à l'aide de variables internes dans lesquels s'échangeront les données entre chaque bloc.

Comme pour les modules précédents, il est important de s'assurer de la validité du module de permutation afin de garantir sa fiabilité. Il est donc nécessaire de concevoir également un testbench pour pouvoir le tester.

Pour ce faire, j'ai commencé par tester le module *permutation_simple* (sans xor), en injectant en entrée les valeurs initiales et on s'attend à trouver en sortie les valeurs dans la couche de diffusion.

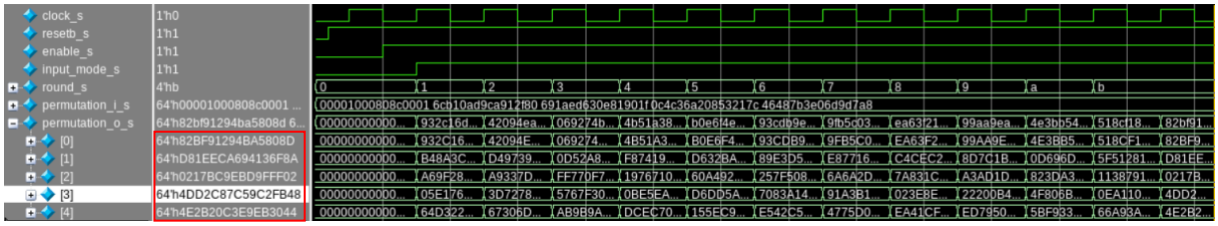


Figure 14 : Simulation du module permutation_simple

Effectivement, comme le montre la figure 14, les résultats obtenus sont conformes aux valeurs attendues, ce qui confirme le bon fonctionnement et la validité de la permutation simple.

Ensuite, j'ai testé la *permutation_xor_end* (sans *xor_begin*), en injectant en entrée le résultat de la couche de diffusion de la 11ième ronde, et l'on s'attend à retrouver en sortie le résultat suivant :

```
État ^ (0...0 & K) : 82bf91294ba5808d d81eeca694136f8a 0217bc9ebd9fff02 2163c2a59353d4c8 2731cda0e76aa05b
```

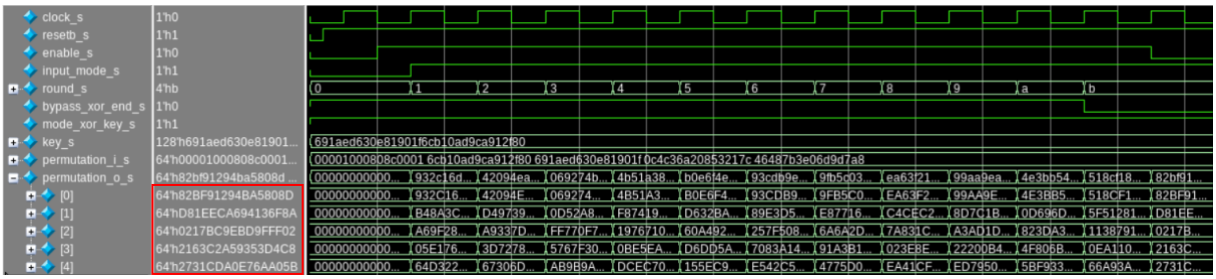


Figure 15 : Simulation du module permutation_xor_end

Effectivement, comme le montre la figure 15, les résultats obtenus sont conformes aux valeurs attendues, ce qui confirme le bon fonctionnement et la validité du module implémenté.

Pour tester la *permutation_xor* j'ai choisi de travailler dans la phase des données associées (voir figure 16), où figurent à la fois le *xor_begin* en haut et *xor_end* en bas.

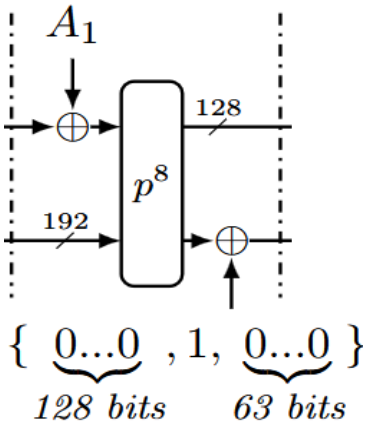


Figure 16 : Phase données associées

On s'attend à obtenir le résultat suivant à la 11ième ronde :

```
État ^ (0...0 & 1) : b49b67cedab66470 b98e693352383f33 67ab18622472e85f 14a6b7765af4dadc bd181140fcca9854
```

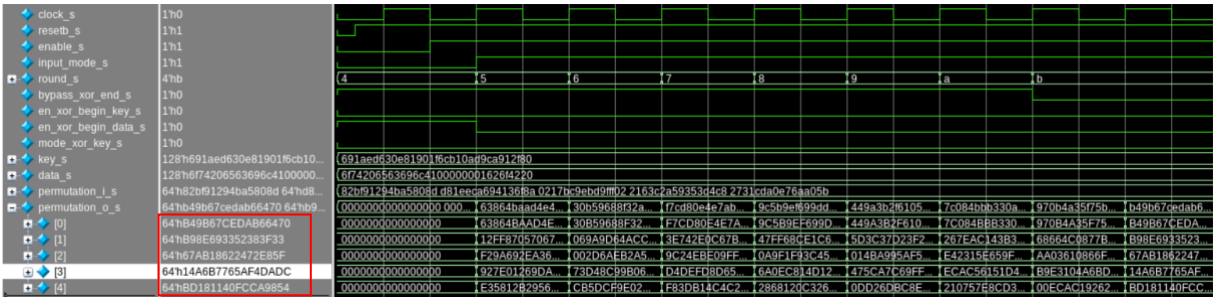


Figure 17 : Simulation du module permutation_xor

Effectivement, comme le montre la figure 17, les résultats obtenus sont conformes aux valeurs attendues, ce qui confirme le bon fonctionnement et la validité de la permutation avec les 2 XOR.

Il ne reste maintenant qu'à tester le module final. Pour ce faire, j'ai choisi de travailler dans la phase de finalisation (voir figure 18), où la sortie est le tag recherché.

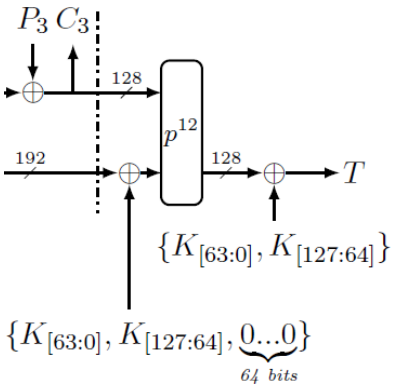


Figure 18 : Phase de finalisation

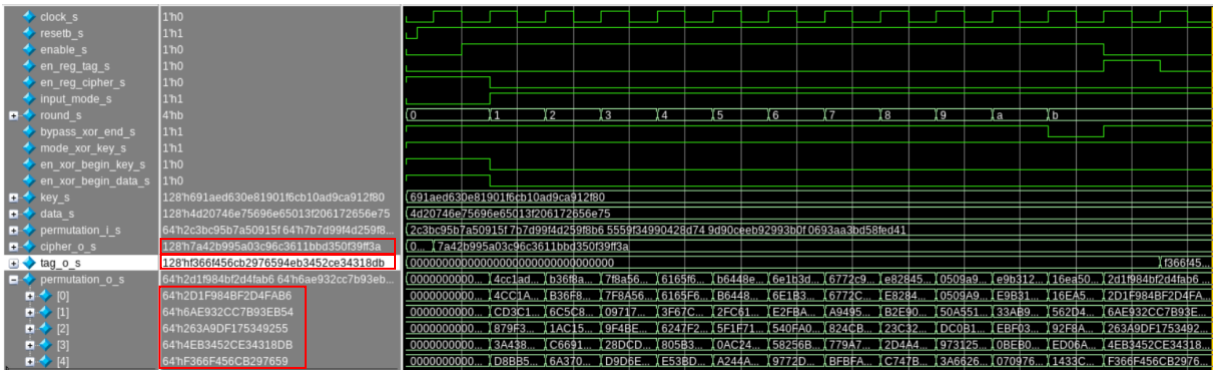


Figure 19 : Simulation du module permutation_xor_reg

Effectivement, comme le montre la figure 19, j'obtiens les bonnes valeurs du cipher C_3 et du tag, ce qui confirme la validité de ma permutation finale.

4. Modélisation de la machine d'état

La machine de Moore constitue le cœur de l'algorithme Ascon-AEAD128. Elle joue un rôle essentiel en orchestrant l'ensemble des signaux nécessaires au bon déroulement de l'algorithme.

La machine d'état doit pouvoir contrôler l'ensemble des signaux d'entrée de la permutation ainsi que le compteur de rondes, chargé de décompter les itérations des permutations et de fournir la valeur de round_i, afin d'assurer le bon déroulement des quatre phases de l'algorithme (initialisation, donnée associée, texte clair, finalisation).

J'ai fait le choix d'utiliser seulement le compteur de rondes fourni et de ne pas utiliser le compteur de bloc.

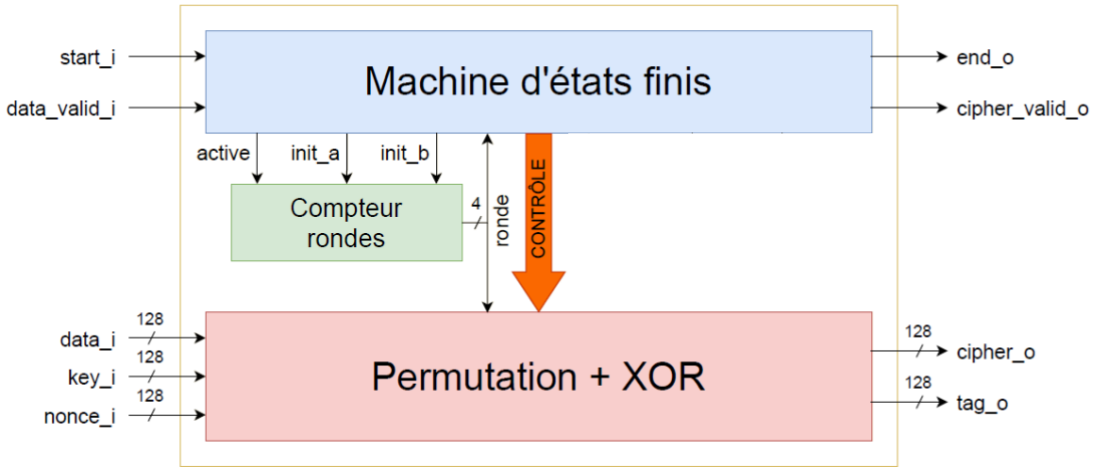


Figure 20 : Architecture globale de Ascon-AEAD128

La machine d'état se compose des entrées/sorties suivantes :

Entrées :

- start_i
- data_valid_i
- round_i
- clock_i
- resetb_i

Sorties :

- input_mode_o
- en_reg_state_o
- en_xor_begin_data_o
- en_xor_begin_key_o
- bypass_xor_end_o
- mode_xor_key_o
- en_reg_cipher_o
- en_reg_tag_o
- en_cpt_double_o
- init_p12_o
- init_p8_o
- cipher_valid_o
- end_initialisation_o
- end_associate_o
- end_cipher1_o
- end_cipher2_o
- end_o

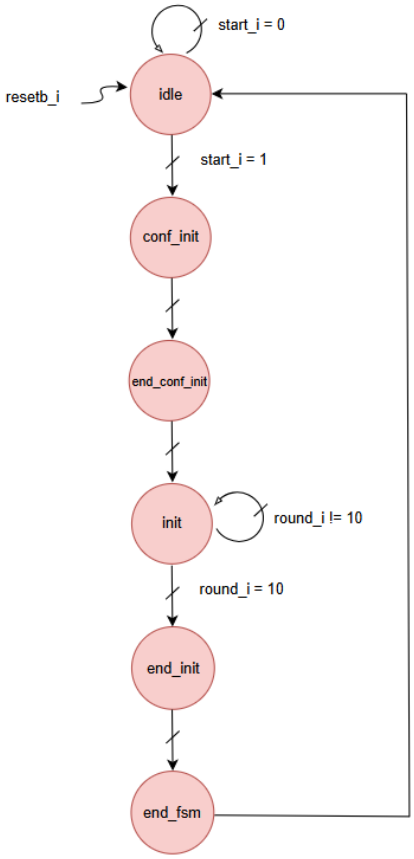


Figure 21 : Diagramme d'état de la phase d'initialisation

Le diagramme d'états ci-dessus peut être généralisé à l'ensemble des phases de l'algorithme Ascon-AEAD128 (voir figure 22). Afin d'assurer une transition correcte entre les différentes phases, il est nécessaire d'introduire des états d'attente permettant la validation des données en sortie et la préparation de la phase suivante. Par ailleurs, une attention particulière doit être portée à l'initialisation du compteur, qui doit être réglé à 8 ou 12 rondes selon les exigences de la phase en cours.

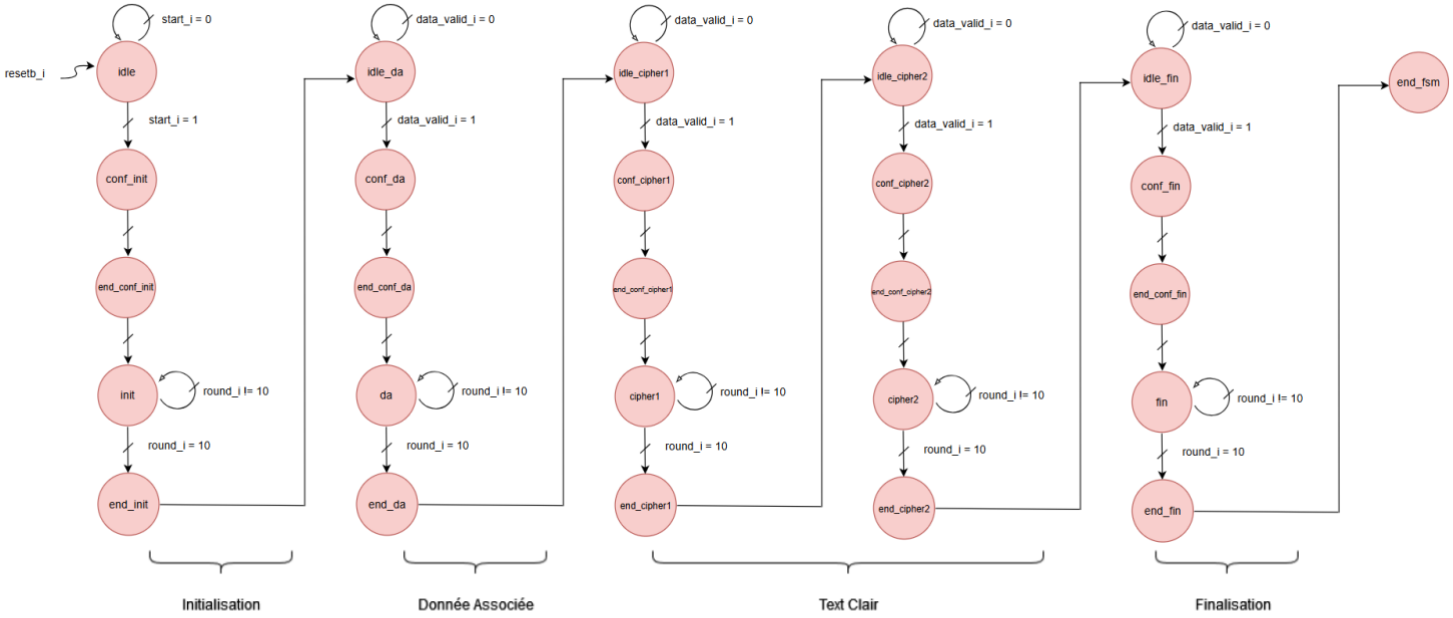


Figure 22 : Diagramme d'état complet

Le code de la machine à états finis (FSM) est structuré en deux parties principales. La première gère la transition entre les états, tandis que la seconde définit les valeurs de sortie.

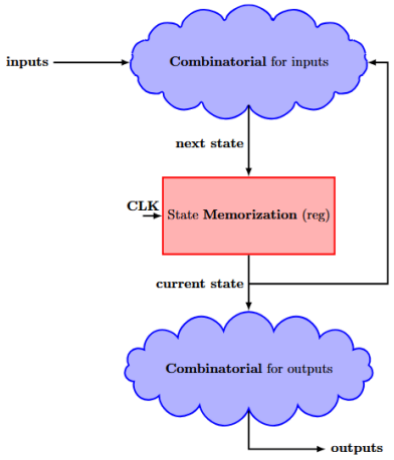


Figure 23 : Architecture générale d'une FSM

Ces deux blocs ont été développés à l'aide de structures procédurales de type always_comb, ce qui facilite la gestion des transitions entre les états et les différentes phases de l'algorithme.

Pour représenter les différents états internes, j'ai utilisé un type *enum*, permettant de définir de manière claire et lisible l'ensemble des états possibles de la FSM.

Dans la première partie, chaque état est associé à son état suivant selon des conditions bien définies. Ainsi, dans les états d'attente, une vérification est systématiquement effectuée pour s'assurer de la validité des données avant de permettre la transition vers l'état suivant. Par exemple, la validité d'une donnée en entrée *(data_i*), c'est-à-dire que la donnée peut être utilisé dans le calcul, est indiquée par *data_valid_i*. De façon analogue, la disponibilité d'un bloc de texte chiffré (*cipher_o*) sera indiquée par le signal de validité *cipher_valid_o*.

La seconde partie permet d'assigner les valeurs de sortie en fonction de l'état actuel. Cela permet aux autres modules du système de savoir, par exemple, si un signal doit être activé à un moment donné, ou non.

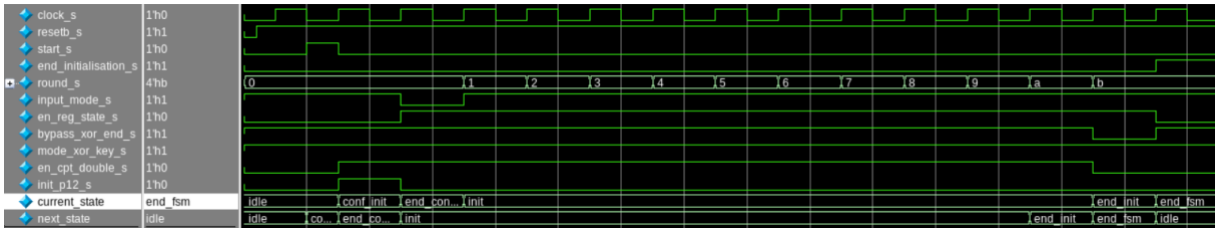


Figure 24 : Simulation de la FSM dans la phase d'initialisation

J'ai commencé par tester la machine d'états durant la phase d'initialisation. Comme le montre la figure 24, les signaux s'activent au moment attendu et les états courants et suivants observés sont cohérents, ce qui confirme la validité de la FSM en phase d'initialisation.

J'ai choisi d'activer les signaux *xor_begin* à l'entrée du module de permutation, c'est-à-dire dès la première ronde, et non à sa sortie. Par exemple, lors de la ronde 0 de la permutation p^{12} dans la phase de finalisation, j'ai activé à la fois le *xor_begin* avec le bloc de texte clair P_3 (*en_xor_begin_data_o*) et celui avec la clé (*en_xor_begin_key_o*).

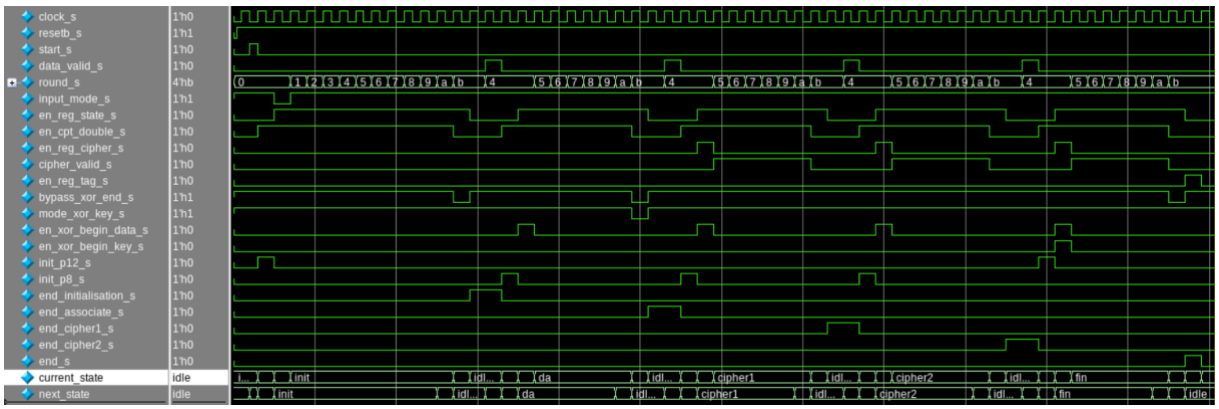


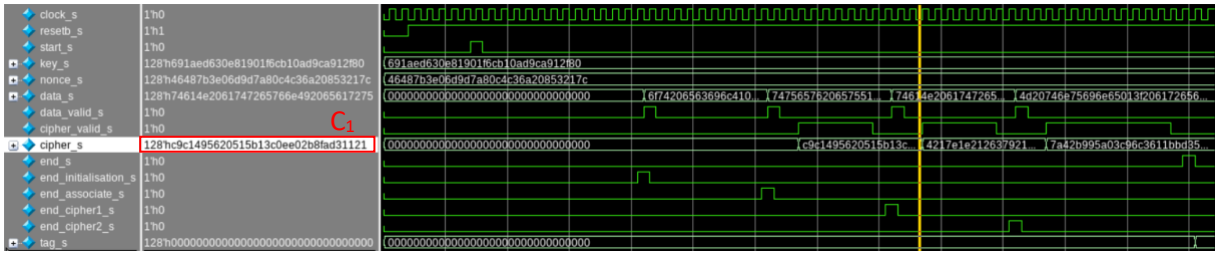
Figure 25 : Simulation de la FSM finale

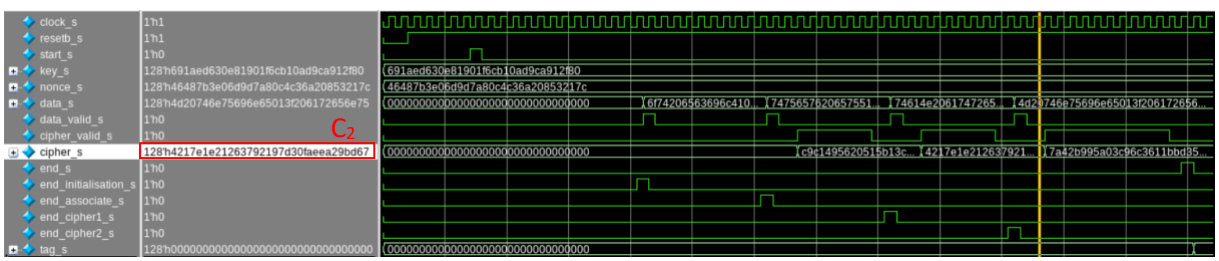
Comme le montre la figure 25, on obtient bien des résultats cohérents, ce qui confirme le bon fonctionnement de la machine d'état.

5. Modélisation final d' Ascon-AEAD128

Le module final, nommé ascon_top, représente l'implémentation complète de l'algorithme de chiffrement ASCON-128. Il permet de simuler l'ensemble de l'algorithme en intégrant les différents modules développés au cours de ce projet. Plus précisément, ascon_top assure la coordination entre les modules de permutation, la machine à états finis (FSM) et le compteur de rondes. Il joue ainsi un rôle central en orchestrant l'interaction entre ces blocs fonctionnels pour garantir le bon déroulement de l'algorithme.

Il est important de bien inverser les valeurs fournies pour les données associées et les textes clairs, tout en veillant à appliquer le padding au bon endroit afin d'éviter toute erreur dans les valeurs obtenues.





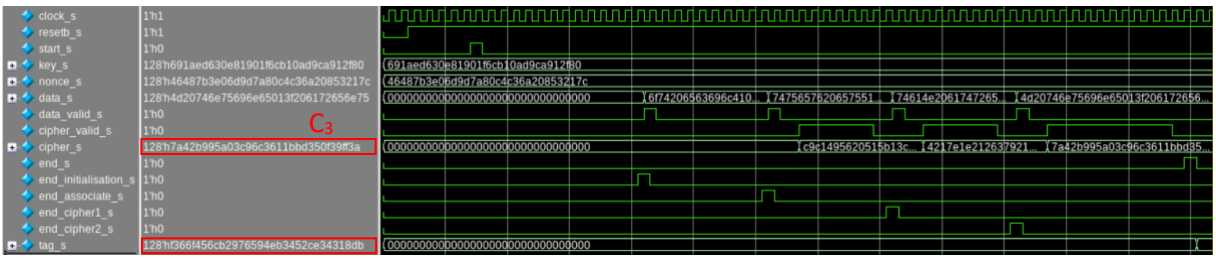


Figure 26 : Simulations d'ascon_top

Comme le montrent les simulations de la figure 26, les valeurs des textes chiffrés C_1, C_2, C_3 ainsi que le tag sont correctes, et les signaux s'activent au moment attendu. Nous pouvons donc valider le résultat final ainsi que l'ensemble du projet.

Difficultés rencontrées

J'ai rencontré plusieurs difficultés au cours de ce projet. La première a été liée au niveau d'abstraction et de recul nécessaire pour comprendre l'algorithme dans sa globalité. Au départ, il m'a semblé complexe d'identifier clairement les différentes étapes de l'algorithme et de saisir la manière dont elles s'articulent entre elles. Ensuite, le développement des différents modules ainsi que de leurs testbenchs a également posé plusieurs problèmes. J'ai été confrontée à de nombreuses erreurs, dont les causes n'étaient pas toujours évidentes à identifier ni à corriger. La prise en main d'un nouveau langage de description matériel a constitué une difficulté supplémentaire, d'autant plus qu'il fallait l'utiliser pour mener à bien l'ensemble du projet.

Conclusion

Ce projet m'a permis d'explorer un domaine totalement nouveau et particulièrement stimulant. J'ai été étonnée de constater qu'un algorithme de chiffrement, souvent associé à une implémentation logicielle en C ou Python, pouvait être entièrement réalisé au niveau matériel.
Cette expérience a été enrichissante et s'est globalement bien déroulée, bien qu'il subsiste certains aspects perfectibles dans l'implémentation. Des optimisations pourraient notamment être envisagées au niveau de la machine à états, en réduisant le nombre d'états grâce à l'intégration d'un compteur de blocs, ou encore en remplaçant la machine de Moore par une machine de Mealy, ce qui améliorerait les performances du système.
En somme, ce projet m'a permis de consolider mes connaissances en SystemVerilog, tout en me familiarisant avec les concepts fondamentaux de la cryptographie matérielle à travers l'algorithme ASCON.