Modélisation SystemVerilog de l'algorithme de chiffrement

Ascon-AEAD128

Nina Perret

Avril-Mai 2025

Sommaire

1 Introduction

1.1 Contextualisation de l'algorithme ASCON

L'algorithme ASCON est fondé sur un système de chiffrement authentifié avec données associées.

Il permet à deux personnes de communiquer par l'intermédiaire de messages qu'elles seules peuvent comprendre.

En premier lieu, les données à échanger sont chiffrées du côté de l'expéditeur. Puis elles sont déchiffrées du côté du destinataire. Ces processus de chiffrement/déchiffrement sont possibles grâce à une clé secrète partagée entre l'expéditeur et le destinataire.

Cependant, la confidentialité du contenu n'est pas le seul objectif atteignable avec l'algorithme ASCON.

En effet, l'authenticité du contenu est également garantie, grâce à l'utilisation d'un tag. Ce dernier est recalculé par le destinataire, à partir du message chiffré reçu. S'il ne correspond pas au tag attendu, le destinataire saura que le contenu du message est corrompu.

Structure du message reçu

$$Message\ recu = Cipher1 \,||\, Cipher2 \,||\, Cipher3 \,||\, Tag$$

Il y a trois textes encryptés (ciphers) car le texte clair (plaintext) est décomposé en trois blocs.

1.2 Objectifs du projet et architecture générale

L'objectif de ce projet est d'implémenter l'algorithme ASCON. Pour ce faire, nous utiliserons SystemVerilog. Il s'agit d'un langage de description et de modélisation matérielle.

L'espace de travail a été organisé en deux bibiothèques. D'une part, la bibliothèque RTL regroupe les modules composant ASCON. D'autre part, la bibiothèque BENCH contient les testbenches. Ces derniers sont des scripts permettant de tester les modules précédemment implémentés.

L'ensemble de ces fichiers permet d'effectuer des simulations Modelsim. L'analyse des chronogrammes générés permet de vérifier le bon fonctionnement des modules.

Liste des modules :

- ascon_pack - fourni avec l'énoncé
- Pc : addition d'une constante de round
- Ps : substitution
- Pl : diffusion linéaire
- xor_begin
- xor_end
- registre_cipher

- registre_tag
- state_registre
- L'ensemble des modules précédents permet d'implémenter le module **Permutation**.
- compteur_double_init (abrégé par "round") - fourni avec l'énoncé
- **FSM_Moore** : machine d'états
- **ascon_top** : assemblage de toute l'architecture

Les modules Pc, Ps, Pl, Permutation, FSM_Moore et ascon_top sont accompagnés d'un testbench.

État courant :

L'algorithme ASCON opère sur un état courant de 320 bits. Ce dernier est divisé en cinq registres de 64 bits : S_0, S_1, S_2, S_3 et S_4. Un tel état sera nommé type_state dans les scripts.

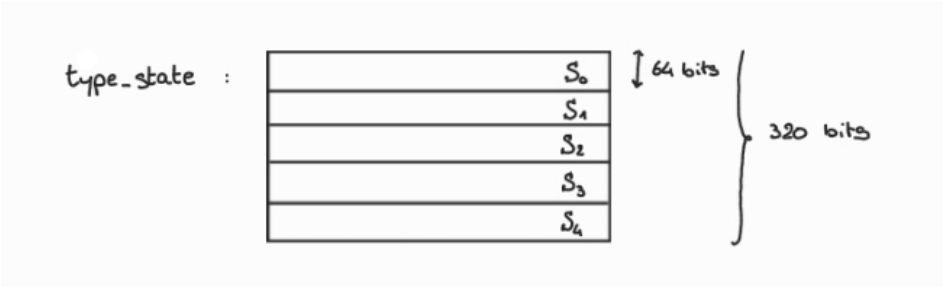

Figure 1: Schéma de la structure type_state

Il est initialisé ainsi :

$$Ligne\ 0 = Initialisation\ Vector = Ox00001000808c0001$$

$$Ligne\ 1 = key\_i[63:0]$$

$$Ligne\ 2 = key\_i[127:64]$$

$$Ligne\ 3 = nonce\_i[63:0]$$

$$Ligne\ 4 = nonce\_i[127:64]$$

IV est une constante définie dans l'algorithme. *Key* correspond à la clé secrète de 128 bits partagée entre l'expéditeur et le destinataire. *Nonce* est un nombre arbitraire de 128 bits.

L'état courant est ensuite transformé progressivement au fil des étapes effectuées.

On débutera chaque module par l'importation du fichier ascon_pack.sv, qui définit ce type_state.

2 Module Pc : addition de la constante

2.1 Théoriquement

Ce module concerne le registre S_2 de l'état courant. Pour la ronde concernée, il s'agit d'ajouter une constante c_r au registre S_2, selon l'opération suivante :

$$S_2 \leftarrow S_2 \oplus c_r$$

Les valeurs affectées à la constante de ronde sont définies dans le module ascon_pack.

Ronde r de p^{12}	Ronde r de p^8	Constante c_r
0		00000000000000000f0
1		00000000000000000e1
2		00000000000000000d2
3		00000000000000000c3
4	4	00000000000000000b4
5	5	00000000000000000a5
6	6	0000000000000000096
7	7	0000000000000000087
8	8	0000000000000000078
9	9	0000000000000000069
10	10	000000000000000005a
11	11	000000000000000004b

Figure 2: Tableau récapitulatif des valeurs affectées à la constante de ronde

2.2 Chronogrammes générés et interprétations

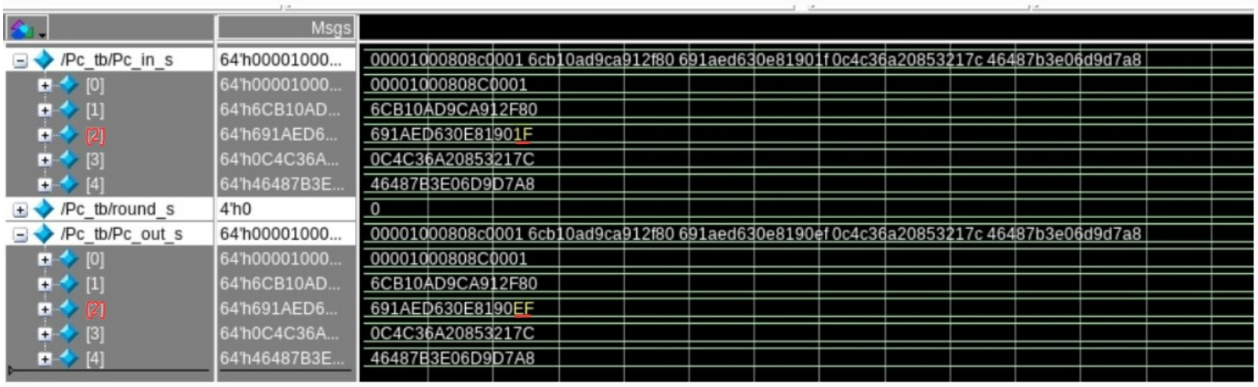

Figure 3: Chronogrammes résultant de la simulation du module Pc

On entre dans la deuxième ligne de l'état courant la valeur suivante : $Ox691aed630e81901f$.

On obtient en sortie la valeur : $Ox691aed630e8190ef$.

Ce résultat correspond bien à l'ajout de la première constante de ronde : $c_0 = Ox00000000000000f0$.

3 Module Ps : substitution

3.1 Théoriquement

Le but de ce module est de modifier l'état courant en effectuant des substitutions sur les colonnes.

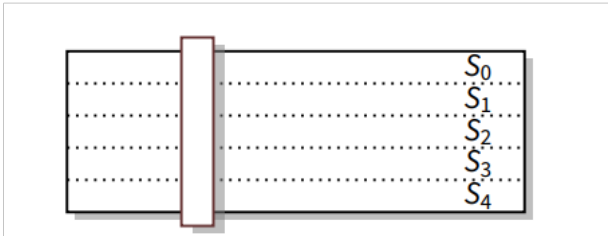

Figure 4: Schéma représentant une colonne de l'état courant

Dans le fichier Ps.sv, on définit une fonction Sbox basée sur la table 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

Figure 5: Table de substitution utilisée

Les substitutions sont effectuées suivant cette table. Chaque colonne fait 5 bits de largeur. Ainsi, une colonne vaut entre 00000 (=0 en décimal) et 11111 (= 31 en décimal). C'est pourquoi la fonction Sbox prend des valeurs entre 0 et 31.

$$\{S_0[i], S_1[i], S_2[i], S_3[i], S_4[i]\} \leftarrow \text{Sbox}(\{S_0[i], S_1[i], S_2[i], S_3[i], S_4[i]\})$$

Figure 6: Fonction Sbox

3.2 Chronogrammes générés et interprétations

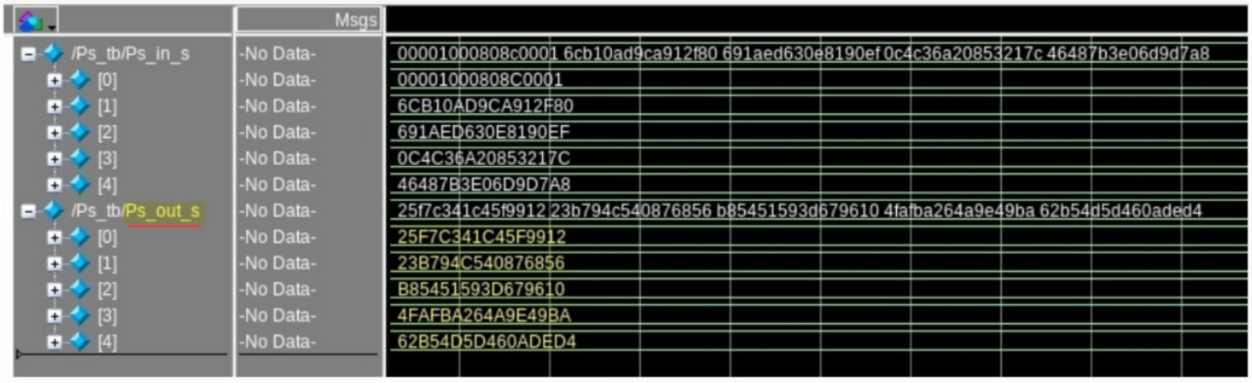

Figure 7: Chronogrammes résultant de la simulation du module Ps

```
-- Permutation (r=00)
Addition constante : 00001000808c0001 6cb10ad9ca912f80 691aed630e8190ef 0c4c36a20853217c 46487b3e06d9d7a8
Substitution S-box : 25f7c341c45f9912 23b794c540876856 b85451593d679610 4fafba264a9e49ba 62b54d5d460aded4
Diffusion linéaire : 932c16dd634b9585 b48a3c3fe8fb45ce a69f28b0c721c340 05e1761f1e1fcb67 64d322a896b791cf
```

Figure 8: Valeurs d'entrée et de sortie utilisées pour le test du module Ps

Les valeurs surlignées en jaune correspondent aux valeurs en entrée. Celles surlignées en vert correspondent aux valeurs en sortie. On obtient bien les résultats attendus.

4 Module Pl : diffusion

4.1 Théoriquement

Le module Ps effectuait des substitutions sur les colonnes de l'état courant. A l'inverse, le module Pl effectue des opérations sur les lignes de l'état courant.

Il s'agit d'effectuer les opérations suivantes :

$$
\begin{aligned}
S_0 &\leftarrow \Sigma_0(S_0) = S_0 \oplus (S_0 \text{>>>} 19) \oplus (S_0 \text{>>>} 28) \\
S_1 &\leftarrow \Sigma_1(S_1) = S_1 \oplus (S_1 \text{>>>} 61) \oplus (S_1 \text{>>>} 39) \\
S_2 &\leftarrow \Sigma_2(S_2) = S_2 \oplus (S_2 \text{>>>} 1) \oplus (S_2 \text{>>>} 6) \\
S_3 &\leftarrow \Sigma_3(S_3) = S_3 \oplus (S_3 \text{>>>} 10) \oplus (S_3 \text{>>>} 17) \\
S_4 &\leftarrow \Sigma_4(S_4) = S_4 \oplus (S_4 \text{>>>} 7) \oplus (S_4 \text{>>>} 41)
\end{aligned}
$$

>>> correspond à une rotation cyclique vers la droite.

4.2 Chronogrammes générés et interprétations

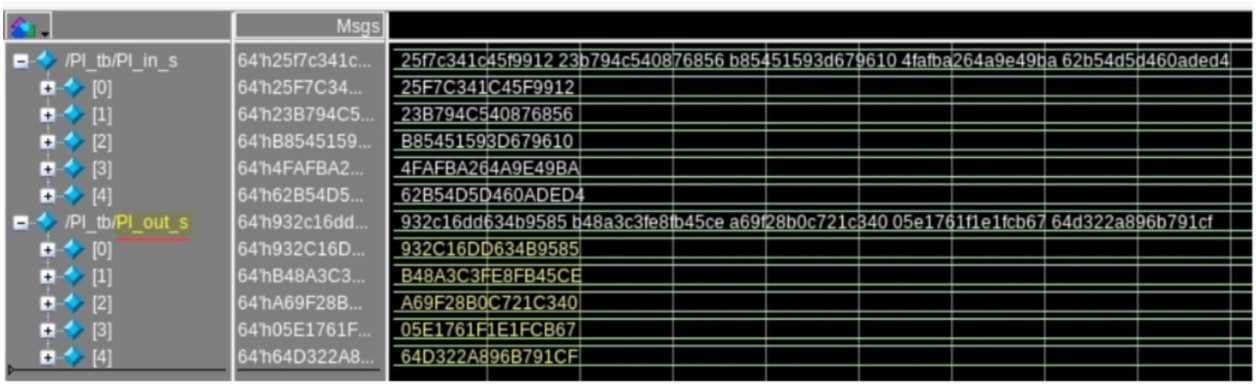

Figure 9: Chronogrammes résultant de la simulation du module Pl

Les valeurs surlignées en jaune correspondent aux valeurs en entrée. Celles surlignées en vert correspondent aux valeurs en sortie. On obtient bien les résultats attendus.

```
-- Permutation (r=00)
Addition constante : 00001000808c0001 6cb10ad9ca912f80 691aed630e8190ef 0c4c36a20853217c 46487b3e06d9d7a8
Substitution S-box : 25f7c341c45f9912 23b794c540876856 b85451593d679610 4fafba264a9e49ba 62b54d5d460aded4
Diffusion linéaire : 932c16dd634b9585 b48a3c3fe8fb45ce a69f28b0c721c340 05e1761f1e1fcb67 64d322a896b791cf
```

Figure 10: Valeurs d'entrée et de sortie utilisées pour le test du module Pl

5 Permutation sans XORs

Nous avons construit les modules Pc, Ps et Pl. Dans le module Permutation, ces derniers s'articulent de la manière suivante :

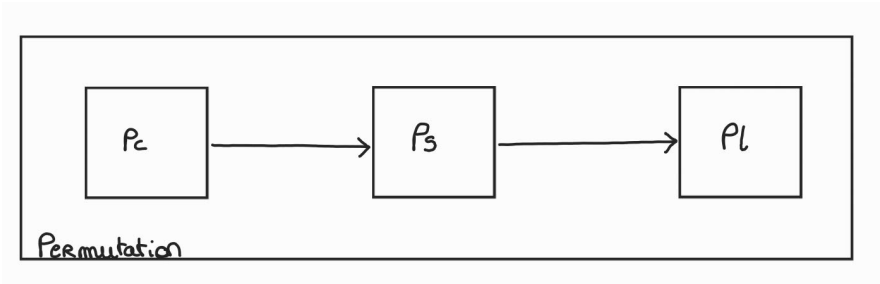

Figure 11: Utilisation des modules Pc, Ps et Pl dans le module Permutation

Ainsi, pour chaque ronde, on effectue successivement :

- L'addition d'une constante de ronde, sur S_2 (Pc).
- Des substitutions selon la fonction Sbox, sur chaque colonne de l'état courant (Ps).
- Une diffusion linéaire sur chacune des lignes de l'état courant (Pl). Les opérations réalisées sont décrites dans le 4.1.

6 Module Permutation, avec XORs

6.1 Théorie module Permutation

En plus des modules Pc, Ps et Pl, le module Permutation fait également intervenir cinq autres modules : xor_begin, xor_end, registre_cipher, registre_tag, et state_registre.

Créer ces modules annexes est un choix personnel permettant d'identifier les erreurs de manière plus aisée lors de la compilation et de la simulation.

Entrées :

- *clock_i*
- *resetb_i*
- *init_i* : signal de sélection du premier multiplexeur (voir schéma figure 12, ci-dessous)
- *round_i* : compteur de rondes
- *enable_xe_i* : signal de sélection du multiplexeur du module xor_end
- *enable_xb_i* : signal de sélection du multiplexeur du module xor_begin
- *data_i* : A_1, P_1, P_2 *et* P_3 (donnée associée et les 3 textes clairs). Il s'agit de la donnée XORée avec le module xor_begin.
- *key_i*: clé secrète partagée entre l'expéditeur et le destinataire
- *nonce_i* : valeur arbitraire
- *enable_p_i* : write enable du state register (voir schéma figure 19)
- *enable_cipher_i* : write enable du cipher register (voir schéma figure 17)
- *enable_tag_i* : write enable du tag register (voir schéma figure 18)

Sorties :

- *tag_o* : Le tag fait partie du message reçu par le destinataire. Ce dernier doit le recalculer lorsqu'il reçoit le message. Le tag permet au destinataire de s'assurer de l'authenticité du message reçu. En effet, s'il ne correspond pas à la valeur attendue, cela signifie que le message a été corrompu. Le processus de calcul du tag ne sera pas détaillé ici.

- *cipher_o* : textes encryptés, il y en aura trois, C_1, C_2 *et* C_3.

Signaux internes :

Logique : x_a_y_s correspond à un signal interne allant de x à y. Par exemple, mux_a_xb_s correspond au signal interne allant du multiplexeur à XOR begin.

- *state_in_s* : type_state, utile à l'initialisation de l'état courant
- *mux_a_xb_s* : type_state, état courant entre le multiplexeur et xor_begin
- *xb_a_pc_s* : type_state, état courant entre xor_begin et Pc
- *pc_a_ps_s* : type_state, état courant entre Pc et Ps
- *ps_a_pl_s* : type_state, état courant entre Ps et Pl
- *pl_a_xe_s* : type_state, état courant entre Pl et xor_end
- *xe_a_reg_s* : type_state, état courant entre xor_end et state register
- *memoire_s* : type_state, état courant mémorisé dans state register
- *memoire_cipher_s* : type_state, état courant mémorisé dans cipher register
- *memoire_tag_s* : type_state, état courant mémorisé dans tag register
- *reg_a_mux_s* : type_state, signal qui retourne au multiplexeur

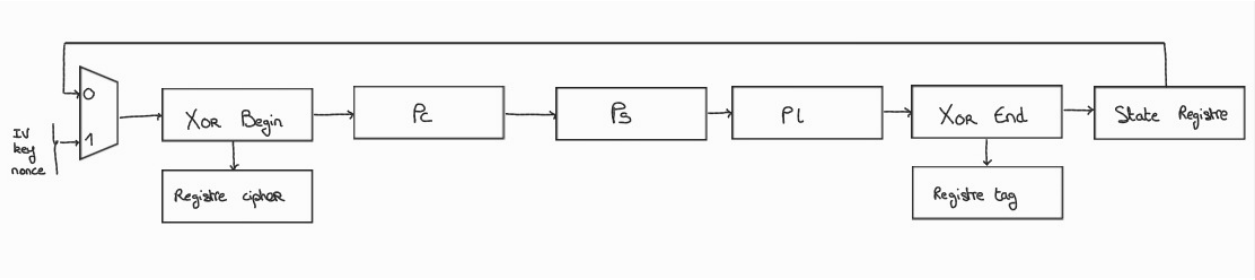

Figure 12: Schéma décrivant l'organisation du module Permutation

6.2 Théorie XOR begin

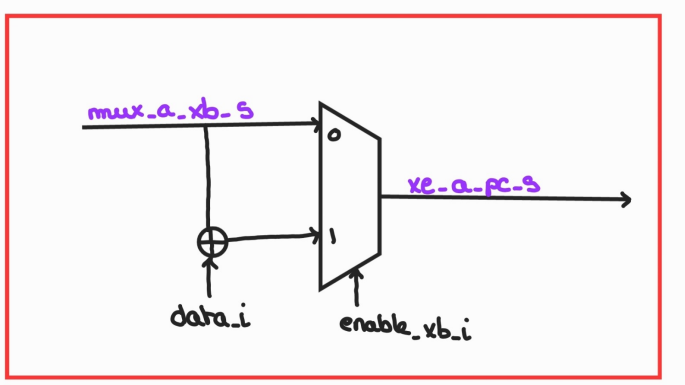

Figure 13: XOR begin

Le module est composé d'un multiplexeur à 2 entrées. Ainsi, le signal de sélection (*enable_xb_i*) peut prendre les valeurs 0 ou 1.

Si $enable\_xb\_i = 0$, le signal d'entrée (provenant de state_registre) est inchangé.

Si $enable\_xb\_i = 1$, alors un XOR est effectué avec data_i.

Remarque : *data_i* correspond à $A1$, $P1$, $P2$ ou $P3$ en fonction de la phase concernée. Ces informations sont définies sur le schéma intitulé "Algorithme de chiffrement" (figure 15).

6.3 Zoom sur le XOR end

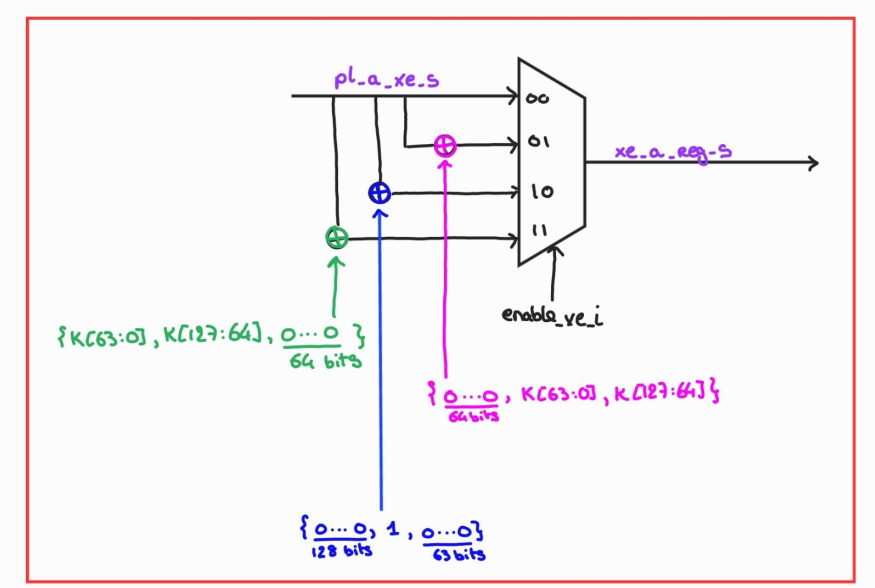

Figure 14: XOR end

Le module est composé d'un multiplexeur à 4 entrées. Ainsi, le signal de sélection (*enable_xe_i*) peut prendre les valeurs 00, 01, 10 ou 11.

Si $enable\_xe\_i = 00$, le signal d'entrée (provenant de la sortie de Pl) est inchangé.

Si $enable\_xe\_i = 01$, le signal d'entrée est xoré avec la clé "paddée" rose (selon le code couleur des figures 14 et 15).

Si $enable\_xe\_i = 10$, le signal d'entrée est xoré avec le signal bleu.

Enfin, si $enable\_xe\_i = 11$, le signal d'entrée est xoré avec la clé "paddée" verte.

6.4 Schéma récapitulatif XORs

Sur le schéma ci-contre, les xor_begin sont ceux de la ligne du haut tandis que les xor_end sont ceux de la ligne du bas.

Les différentes combinaisons possibles pour xor_end sont représentées par des couleurs différentes.

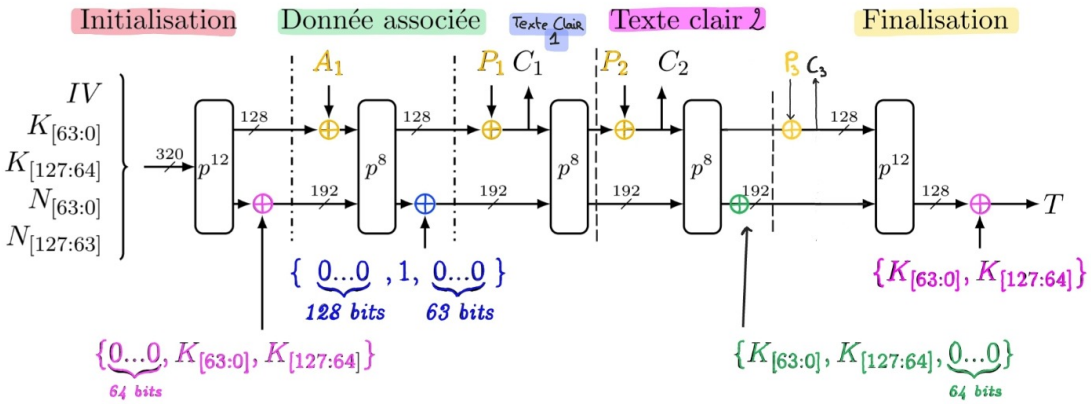

Figure 15: Algorithme de chiffrement

Les XORs sont codés selon la figure 16 : xor_begin utilise les lignes 0 et 1 de l'état courant tandis que xor_end utilise les lignes 2, 3 et 4 de l'état courant.

Remarque : Les entrées et les sorties des modules xor sont des type_state.

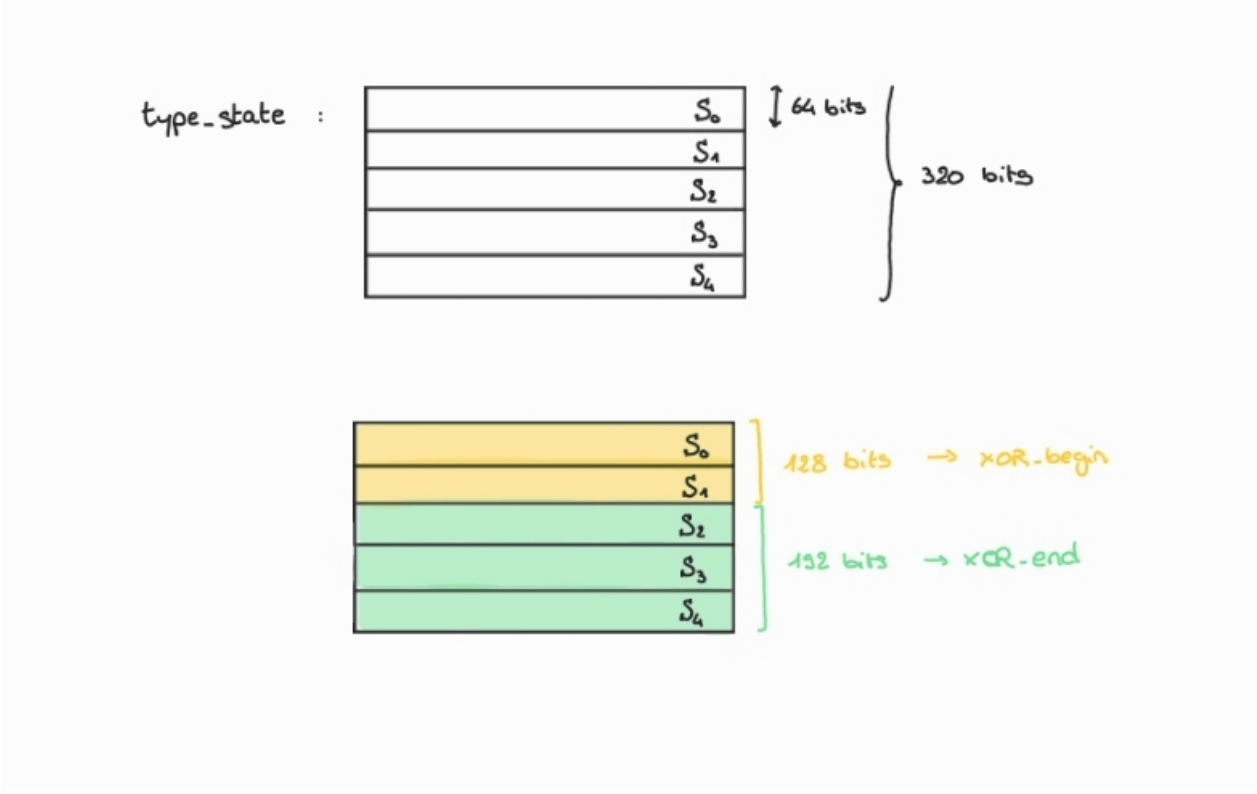

Figure 16: XORs et type_state

6.5 Zoom sur le registre cipher

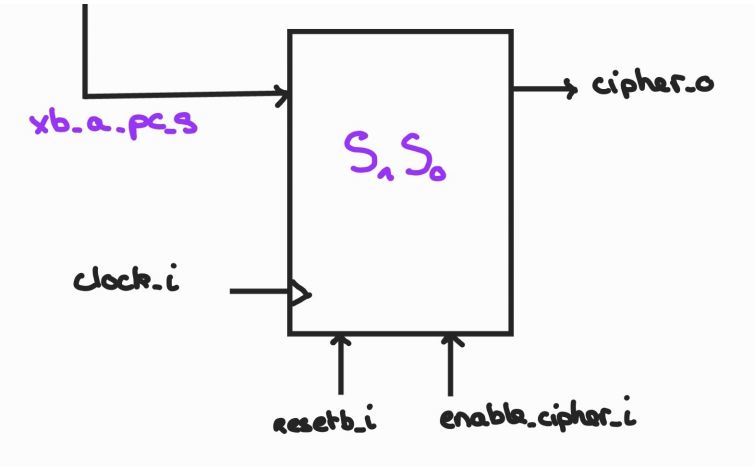

Figure 17: Schéma cipher register

6.6 Zoom sur le registre tag

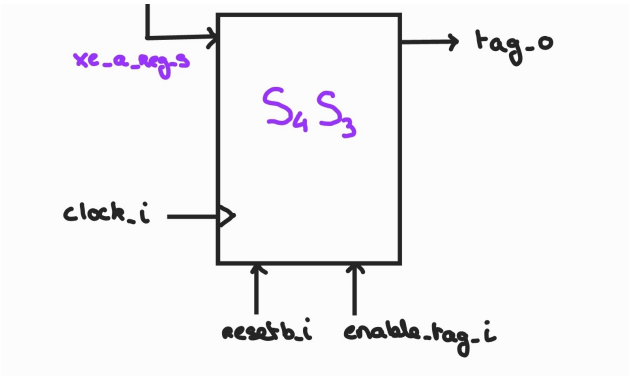

Figure 18: Schéma tag register

6.7 Zoom sur le state registre

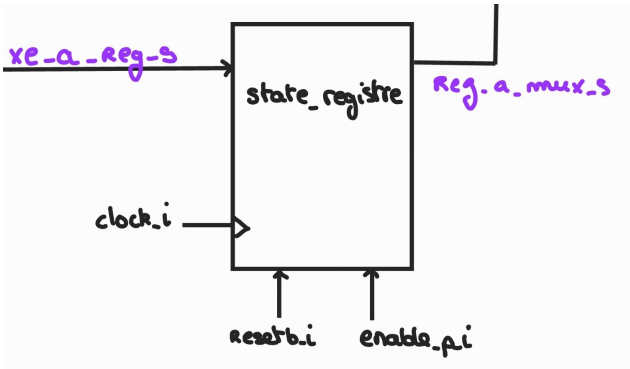

Figure 19: Schéma state register

Le signal interne *memoire_s* permet de mémoriser une valeur dans le registre.

Le signal enable_p_i permet d'autoriser l'écriture dans le registre.

Si resetb_i = 1 (désactivé - reset bas), que l'horloge est en front montant et que enable_p_i = 1, alors on écrit dans le registre. Cela signifie qu'on actualise la donnée stockée en mémoire.

En revanche, si enable_p_i = 0, la valeur stockée en mémoire est conservée, et ce même si l'horloge est en front montant.

6.8 Chronogrammes générés et interprétations

Vue d'ensemble

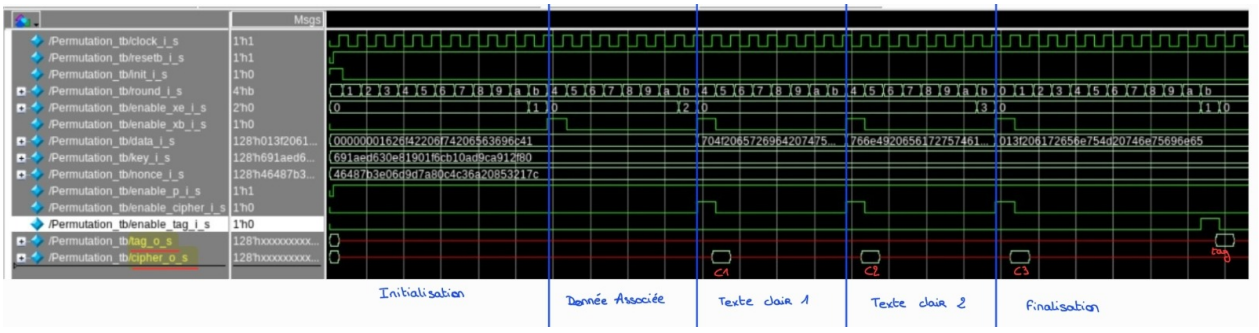

Figure 20: Chronogrammes résultant de la simulation du module Permutation

Le testbench associé au module Permutation a permis de simuler les cinq phases décrites sur le schéma de

la figure 15. L'automatisation du contrôle de ces phases sera détaillée dans la partie portant sur la machine d'états.

On a bien deux types de sorties :

- Trois ciphers
- Le tag

Mise en évidence des XORs

Figure 21: Mise en évidence des XORs

Le code couleur est le même que sur la figure 15.

- Orange : xor_begin
- Rose : 1er xor_end
- Bleu : 2d xor_end
- Vert : dernier xor_end

Zoom sur cipher 1

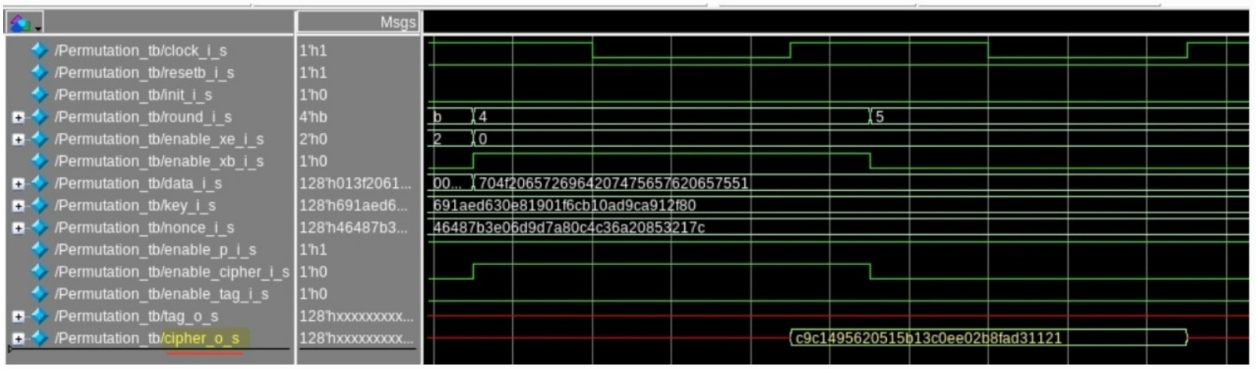

Figure 22: Zoom sur cipher 1

On obtient bien la valeur attendue pour le cipher 1.

De même, on vérifie que les valeurs de cipher 2, cipher 3 et du tag sont cohérentes : c'est bien le cas.

7 Machine d'états et module correspondant

7.1 Graphe d'états

D'après la figure 15, l'algorithme de chiffrement est constitué de cinq phases :

- Initialisation (Init)
- Donnée associée (Da)
- Texte clair 1 (Tc1)
- Texte clair 2 (Tc2)
- Finalisation (Fin)

Le graphe d'états (figure 23) reprend le même code couleur que la figure 15. Ceci permet d'identifier les cinq phases citées précédemment.

Il s'agit d'une machine de Moore car les sorties dépendent uniquement de l'état présent, et non des entrées.

Chacune des cinq phases reprend une structure similaire.

- Idle : état d'attente
- Conf : configuration des signaux, notamment du compteur
- End_conf : xor_begin + première permutation
- Init/Da/Tc/Fin : boucle. Réalisation des permutations 2 à 11 (ou 2 à 7), respectivement pour p_{12} et p_8
- End : xor_end + dernière permutation (sauf pour la phase Tc1, où cet état est absent car on n'effectue pas de xor_end).

Le dernier état ("stop") permet de terminer l'algorithme en évitant de boucler sur le dernier xor_end.

Le graphe d'état ci-contre a été établi en s'appuyant sur le schéma de la figure 15.

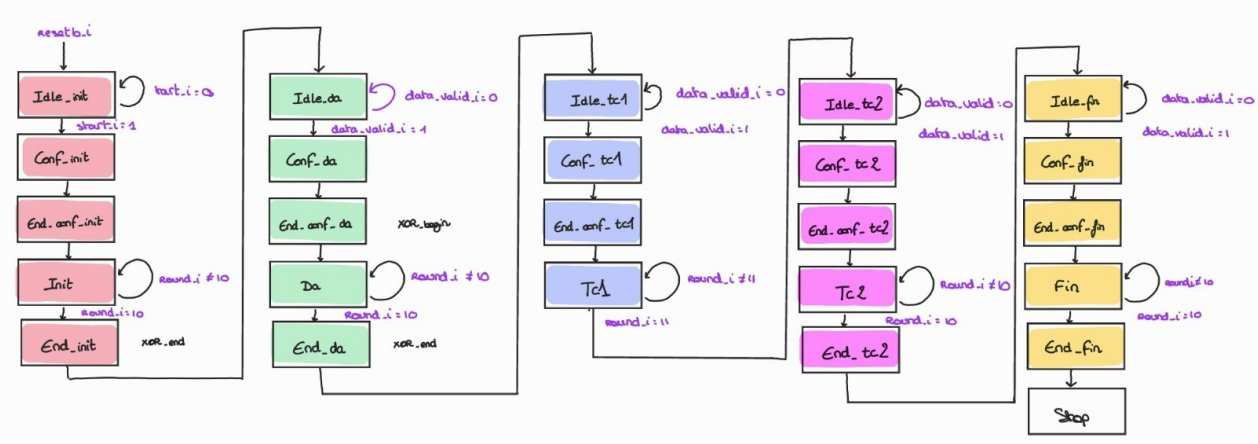

Figure 23: Graphe d'états - Machine de Moore

7.2 Entrées et sorties de la machine d'état

Entrées de la machine d'états :

- *clock_i*
- *resetb_i*
- *start_i* : passage de idle_init à conf_init - autrement dit, démarrage du processus
- *data_valid_i* : lorsque data_valid_i=1, cela signifie qu'une nouvelle donnée est disponible. Cela se produit par exemple lorsque A_1 est disponible, ou encore lorsque P_1 est disponible.
- Entrée associée au module round : *round_i* : compteur de rondes

Sorties de la machine d'états :

Sorties vers le module round (i.e. : ce sont des entrées dans le module round):

- *active_round_o* : permet d'incrémenter le compteur de rondes
- *init_round_p*8_*o* : initialisation du compteur de rondes à 4, car on va effectuer 8 permutations.
- *init_round_p*12_*o* : initialisation du compteur de rondes à 0, car on va effectuer 12 permutations.

Sorties vers le module Permutation (i.e. : ce sont des entrées dans le module Permutation):

- *init_o* : signal de sélection du multiplexeur
- *enable_cipher_o* : write enable du registre cipher
- *enable_tag_o* : write enable du registre tag
- *enable_p_o* : write enable du state registre
- *enable_xe_o* : signal de sélection du multiplexeur de xor_end
- *enable_xb_o* : signal de sélection du multiplexeur de xor_begin

Sorties de la machine d'états et d'ascon_top :

- *end_o* : Ce signal a un rôle double. D'une part, il sert à indiquer la sortie du tag, au même titre que *cipher_valid_o* indique la sortie d'un cipher. D'autre part, il indique la fin de l'algorithme.
- *cipher_valid_o* : lorsque $cipher\_valid\_o = 1$, cela signifie qu'un nouveau cipher est disponible. Par exemple, $cipher\_valid\_o = 1$ quand C_1 est disponible.
- *end_init_o* : fin de la phase initialisation
- *end_da_o* : fin de la phase donnée associée
- *end_cipher_o* : lorsqu'on a obtenu C_1, C_2 et C_3, $end\_cipher\_o = 1$. Autrement dit, lorsqu'on obtient C_3, $end\_cipher\_o = 1$.

La figure 25 présente dans la partie détaillant la structure d'ascon_top permet d'avoir une visualisation de ces signaux.

7.3 Table de vérité

La table de vérité suivante recense l'ensemble des signaux contrôlés par la machine de Moore. Elle décrit leur évolution en fonction des différents états traversés.

	active_round_o	init_round_p8_o	init_round_p12_o	init_o	enable_cipher_o	enable_tag_o	enable_p_o	enable_xe_o	enable_xb_o	end_o	cipher_valid_o	end_init_o	end_da_o	end_cipher_o
Idle_init	0	0	0	0	0	0	0	00	0	0	0	0	0	0
Conf_init	1	0	1	0	0	0	0	00	0	0	0	0	0	0
End_conf_init	1	0	0	1	0	0	1	00	0	0	0	0	0	0
Init	1	0	0	0	0	0	1	00	0	0	0	0	0	0
End_init	0	0	0	0	0	0	1	01	0	0	0	1	0	0
Idle_da	0	0	0	0	0	0	0	00	0	0	0	0	0	0
Conf_da	1	1	0	0	0	0	0	00	0	0	0	0	0	0
End_conf_da	1	0	0	0	0	0	1	00	1	0	0	0	0	0
Da	1	0	0	0	0	0	1	00	0	0	0	0	0	0
End_da	0	0	0	0	0	0	1	10	0	0	0	0	1	0
Idle_tc1	0	0	0	0	0	0	0	00	0	0	0	0	0	0
Conf_tc1	1	1	0	0	0	0	0	00	0	0	0	0	0	0
End_conf_tc1	1	0	0	0	1	0	1	00	1	0	1	0	0	0
Tc1	1	0	0	0	0	0	1	00	0	0	0	0	0	0
Idle_tc2	0	0	0	0	0	0	0	00	0	0	0	0	0	0
Conf_tc2	1	1	0	0	0	0	0	00	0	0	0	0	0	0
End_conf_tc2	1	0	0	0	1	0	1	00	1	0	1	0	0	0
Tc2	1	0	0	0	0	0	1	00	0	0	0	0	0	0
End_tc2	0	0	0	0	0	0	1	11	0	0	0	0	0	0
Idle_fin	0	0	0	0	0	0	0	00	0	0	0	0	0	0
Conf_fin	1	0	1	0	0	0	0	00	0	0	0	0	0	0
End_conf_fin	1	0	0	0	1	0	1	00	1	0	1	0	0	1
Fin	1	0	0	0	0	0	1	00	0	0	0	0	0	0
End_fin	0	0	0	0	0	1	1	01	0	1	0	0	0	0
Stop	0	0	0	0	0	0	0	00	0	0	0	0	0	0

Remarque : le signal *active_round_o* est désactivé dans les états ”end” pour éviter une incrémentation inutile jusqu'à $Ox\ c$.

7.4 Chronogrammes générés

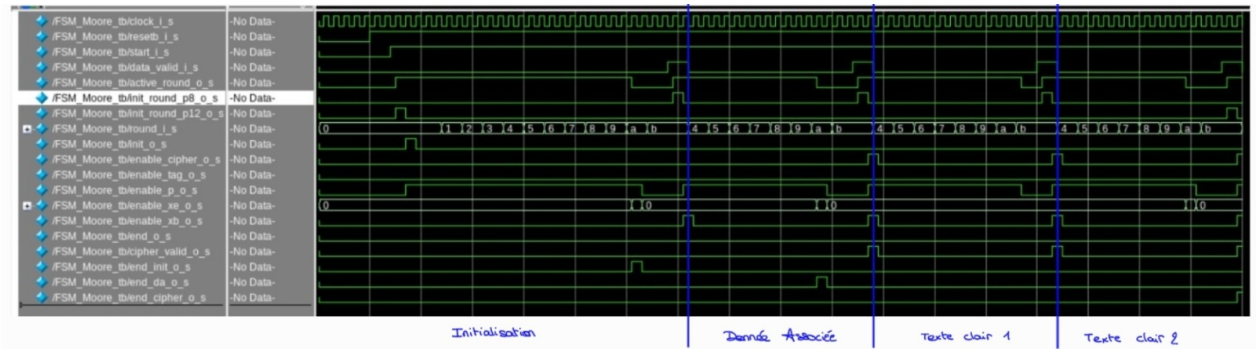

Figure 24: Chronogrammes résultant de la simulation du module FSM

La phase finalisation apparait également dans le chronogramme. Elle n'est pas représentée ci-contre par soucis de lisibilité.

8 Module ASCON top

8.1 Structure

ASCON top fait intervenir trois modules : la machine d'état, le module permutation et le compteur de rondes.

L'énoncé suggèrait d'utiliser également un module bloc. J'ai fait le choix de fonctionner sans ce dernier. Au lieu d'utiliser le module bloc dans la phase "texte clair", j'ai ajouté des états supplémentaires dans la FSM (phase texte clair 2).

La structure du module ASCON top peut être résumée par le schéma suivant.

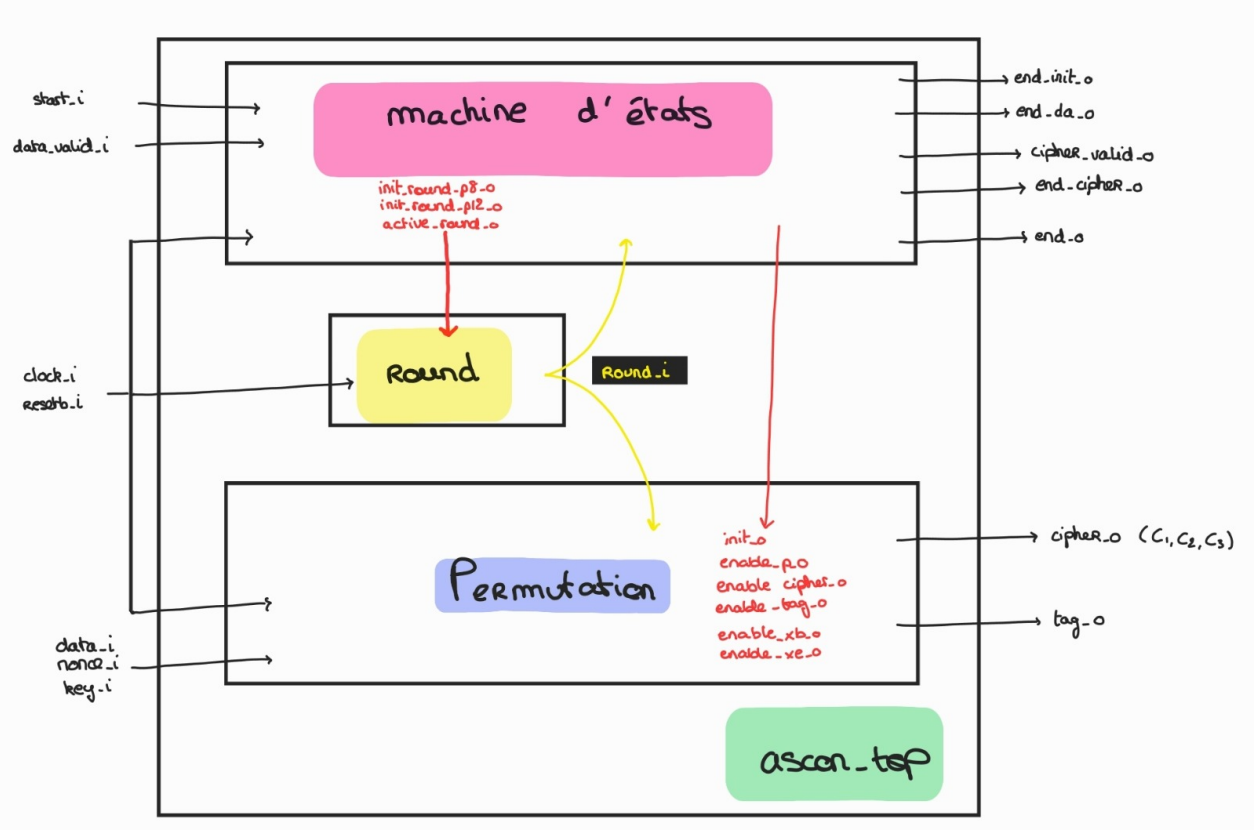

Figure 25: Schéma d'ASCON top

8.1.1 FSM

Comme évoqué plus haut, il y a plusieurs types de sorties dans le module FSM_Moore.

- Les signaux de sortie d'ascon_top : end_o, end_init_o, end_da_o, end_cipher_o, cipher_valid_o
- Les signaux de contrôle du module Permutation : init_o, enable_p_o, enable_cipher_o, enable_tag_o, enable_xb_o, enable_xe_o
- Les signaux de contrôle du module round (compteur_double_init dans le code, abrégé en "round") : init_round_p8_o, init_round_p12_o et active_round_o

A propos des entrées :

- *start_i* permet de débuter l'algorithme.
- *data_valid_i* informe de la disponibilité d'une nouvelle donnée, plaintext ou donnée associée.

8.1.2 Round

Round_i contrôle la machine d'états et le module Permutation. Il s'agit d'un signal d'entrée du point de vue de ces modules.

En fonction du nombre de permutations à effecter, $init\_round\_p8\_o$ ou $init\_round\_p12\_o$ est activé.

Pour effectuer 12 permutations, le compteur est initialisé à 0 ($init\_round\_p12\_o = 1$).

Pour effecteur 8 permutations, le compteur est initialisé à 4 ($init\_round\_p8\_o = 1$).

8.1.3 Permutation

En entrée, on retrouve *data_i*, *key_i* et *nonce_i*.

En sortie, *tag_o* et *cipher_o*. *cipher_o* prendra succesivement les valeurs de C_1, C_2 et C_3.

Remarque : Pour les trois sous-parties précédentes, les signaux *clock_i* et *resetb_i* font également partie des signaux d'entrée.

8.2 Entrées et sorties d'ascon_top

Entrées :

- *clock_i*

- *resetb_i*
- *start_i*
- *data_i*
- *key_i*
- *nonce_i*

Sorties :

- *end_o*
- *end_init_o*
- *end_da_o*
- *end_cipher_o*
- *cipher_valid_o*
- *cipher_o*
- *tag_o*

Tous ces signaux ont les mêmes rôles que cités précédemment.

8.3 Chronogrammes générés et interprétations

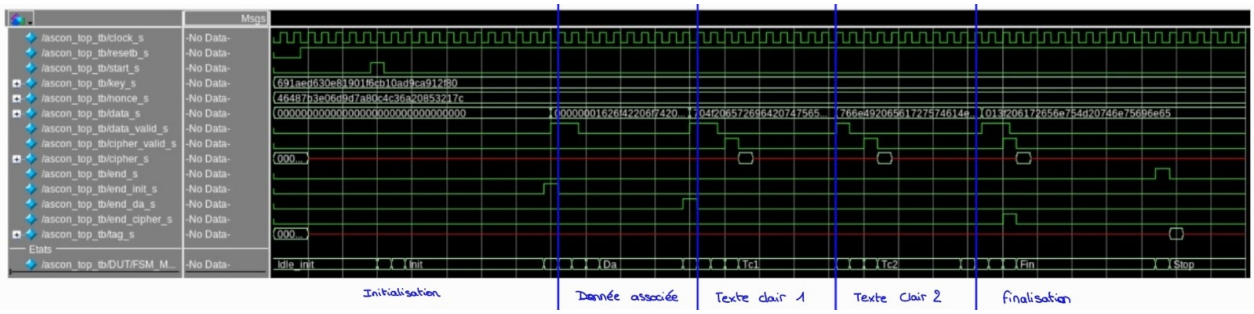

Figure 26: Chronogrammes résultant de la simulation du module ASCON top - Vue d'ensemble

On observe bien :

- Les cinq phases attendues
- Les trois ciphers

- Le tag à la fin

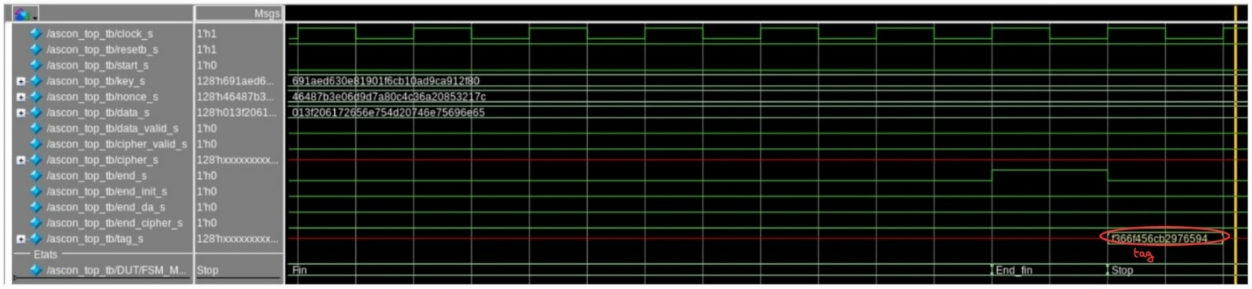

Figure 27: Zoom sur le tag

On a bien la valeur attendue pour le tag :$0xF366F456CB2976594EB3452CE34318DB$.

De même, on vérifie que les trois ciphers sont cohérents : c'est le cas.

9 Conclusion

9.1 Difficultés rencontrées

Les difficultés que j'ai rencontrées concernent principalement le module Permutation.

Initialement, j'avais tout écrit dans le même fichier, mais j'avais du mal à gérer les erreurs. J'ai donc fait le choix de diviser le module Permutation en plusieurs modules intermédiaires : xor_begin, xor_end, state_registre, registre_cipher et registre_tag. Par la suite, cette organisation plus méthodique a permis d'aboutir à un code fonctionnel.

Par ailleurs, j'ai eu du mal à comprendre la gestion du module Bloc suggéré par l'énoncé. C'est pourquoi j'ai préféré ajouter une seconde phase "texte clair", quitte à ajouter des états supplémentaires dans la machine d'états.

9.2 Ressenti

Ce projet m'a permis de comprendre le cours Conception d'un Système Numérique en appliquant concrètement les concepts étudiés.

Il m'a permis d'apprendre à maitriser System Verilog, mais aussi de retravailler des notions abordées plus tôt dans le semestre, comme par exemple les graphes d'états.

Il s'agissait d'un projet ambitieux, surtout dans le cadre d'un semestre avec une charge de travail particulièrement importante. Malgré tout, je suis parvenue à obtenir les résultats attendus. Je suis donc satisfaite du travail que j'ai réalisé sur ce projet.