Rapport Projet Ascon

Conception d'un Système Numérique

Élève

Arthur DORADOUX

Professeur

Guillaume REYMOND

15 Mai 2025

Table des matières

Table des figures

Table des tableaux

1 Introduction

1.1 Histoire du chiffrement de données

Avec l'évolution rapide des nouvelles technologies, nos données personnelles sont collectées en permanence. Afin de garantir leur confidentialité et d'assurer des transmissions sécurisées, il est devenu indispensable de les chiffrer. C'est dans ce contexte qu'est née la cryptographie qui est dédiée à la protection des informations par chiffrement contre toute tentative d'interception, d'analyse ou d'altération.

Les premiers modèles de chiffrement ont été créés à l'Antiquité avec notamment le célèbre code de César. Cet algorithme repose sur un principe de décalage : il faut remplacer chaque lettre d'un message par celle située à un certain nombre de positions plus loin dans l'alphabet. Jules César utilisait cette méthode pour sécuriser ses correspondances militaires, rendant ses messages incompréhensibles et indéchiffrables en cas d'interception. Depuis, la cryptographie a évolué vers des systèmes bien plus complexes et résistants face aux tentatives de décryptage.

Parmi les algorithmes les plus utilisés aujourd'hui, on retrouve l'AES (Advanced Encryption Standard), officiellement adopté par le NIST (National Institute of Standards and Technology) en 2001. Il repose sur des opérations de substitution, de permutation et de diffusion des données, rendant les attaques par force brute, consistant à tester tous les cas possibles, inefficaces. L'AES est aujourd'hui omniprésent, notamment pour le chiffrement des données bancaires, des données gouvernementales sensibles et secrètes et des données industrielles.

Dans le secteur bancaire, la cryptographie joue un rôle fondamental. Les transactions financières s'appuient principalement sur des algorithmes comme RSA, qui utilise un système de clés asymétriques (une clé publique et une clé privée) contrairement aux méthodes symétriques comme AES. Cette approche garantit une sécurité élevée même lors de transmissions sur des réseaux non sécurisés. Actuellement, un chiffrement RSA avec une clé de 2048 bits est considéré comme suffisamment sûr, mais il est primordial de voir plus loin et de prévoir de nouveaux algorithmes encore plus sûrs.

En effet, l'informatique quantique constitue une révolution potentielle pour le domaine du chiffrement. Grâce à la capacité de calcul exponentielle des futurs ordinateurs quantiques, la sécurité des algorithmes de chiffrement actuels sera compromise. L'algorithme de Shor, le plus célèbre des algorithmes quantiques, permettrait de factoriser efficacement les grands nombres utilisés par RSA, le rendant ainsi vulnérable. Face à cette menace, les chercheurs développent la cryptographie post-quantique (PQC), conçue pour résister à ces nouvelles technologies. Le NIST coordonne actuellement la standardisation de ces nouveaux algorithmes afin d'assurer notre sécurité numérique dans les années à venir.

L'essor de l'Internet des Objets (IoT) complexifie encore davantage la gestion et la protection de nos données qui sont désormais dans de nombreux produits avec une puissance limitée. Il est donc essentiel de créer des mécanismes de chiffrement légers mais robustes car ces nouveaux objets constitue des nouvelles portes ouvertes d'accès à nos données.

1.2 Historique de l'Ascon-128

Ascon-128 est un algorithme de chiffrement authentifié avec données associées (AEAD). Il garantit à la fois la confidentialité et l'intégrité des messages grâce à la génération d'un tag d'authentification. Il a été développé en 2014 pour répondre aux besoins grandissants en

cryptographie légère et il est particulièrement adapté aux environnements contraints comme les systèmes embarqués et l'IoT.

Il s'est distingué lors de la compétition internationale CAESAR qui est un concours de chiffrement authentifié et a été sélectionné en 2023 par le NIST comme standard officiel pour la cryptographie légère, ce qui en fait un algorithme dans l'air du temps et reconnu. C'est un algorithme avec une très bonne sécurité, des bonnes performances et qui consomme très peu d'énergie.

Le suffixe « 128 » dans le nom Ascon-128 indique que la clé utilisée fait 128 bits, ce qui assure un bon compromis entre sécurité et efficacité. Il existe néanmoins plusieurs variantes d'Ascon qui permettent de répondre à différents besoins en termes de sécurité et de performance.

1.3 Présentation du projet

Le but de ce projet est d'implémenter une version simplifiée de l'algorithme de chiffrement Ascon-AEAD-128. Il permet par exemple de créer des communications confidentielles et sécurisées en ayant une même clé secrète ce qui fait appel à la cryptographie symétrique.

L'en-tête du message n'est pas chiffré mais le corps du message lui l'est. Il y a un tag qui permet d'être certain que l'en-tête est authentique et qui est calculé par le destinataire à partir du message chiffré. Le but ici va donc être d'avoir le bon tag à la fin de la simulation de notre algorithme.

1.3.1 Outils utilisés

Afin de travailler sur ce projet, nous utilisons tallinn, le serveur de l'école ainsi que les outils ModelSim et le langage SystemVerilog.

SystemVerilog est un langage de description matérielle (HDL - Hardware Description Language) qui étend les capacités du langage Verilog, un autre langage de description matérielle. Il permet de décrire la structure et le comportement des circuits numériques.

ModelSim est un environnement de simulation et de vérification pour la conception de circuits numériques, développé par l'entreprise Mentor Graphics, qui nous permet de tester et de valider la description en SystemVerilog de notre chiffrement Ascon.

1.4 Fonctionnement

1.4.1 L'état S

Notre algorithme de chiffrement Ascon opère sur un état de 320 bits noté S. Il est divisé en 5 registres S_i de 64 bits chacun $(S_0, S_1, S_2, S_3, S_4)$.

On peut alors le représenter ainsi :

319	256	S_0
255	192	S_1
191	128	S_2
127	64	S_3
63	0	S_4

FIG. 1 : Représentation de l'état S comme tableaux de registres S_i

On peut aussi diviser l'état en 2 parties :

- Une partie dite externe de r = 128 bits, notée $S_r = \{S_0, S_1\}$
- Une partie dite interne de c = 192 bits, notée $S_c = \{S_2, S_3, S_4\}$

On a donc bien $S = \{S_r, S_c\}$.

On peut également interpréter l'état S comme 64 colonnes de 5 bits chacune :

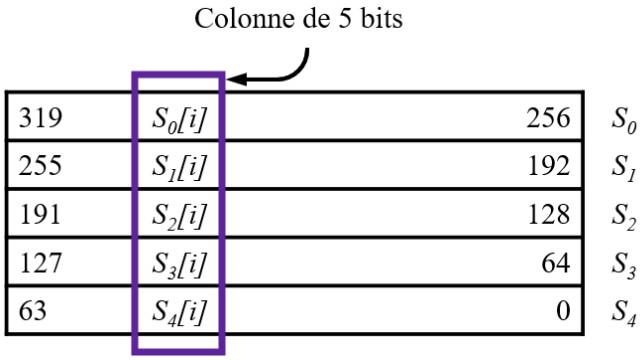

FIG. 2 : Représentation de l'état S sous forme de colonnes de 5 bits

Pour le mettre à jour, on utilise une opération appelée permutation qui est composée de 3 couches :

- la couche d'addition de constante p_c
- la couche de substitution p_s
- la couche de diffusion linéiare p_l

Le fonctionnement de ces 3 couches est détaillé dans chaque partie leur étant dédiée.

On assemble alors ces 3 blocs pour obtenir la permutation $p = p_c \circ p_s \circ p_l$. On peut alors en faire des itérations, appelées rondes, et pour notre algorithme, nous allons en faire 8 ou 12, qu'on notera p^8 ou p^{12}. Associé à plusieurs XORs, cela forme notre projet final qui est découpé en 4 blocs de la manière suivante :

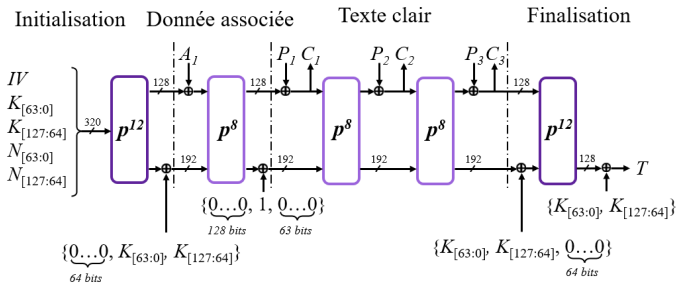

FIG. 3 : Schéma du chiffrement Ascon étudié (AEAD128)

Les différents blocs sont détaillés dans la suite du rapport.

1.4.2 Stratégie d'implémentation en SystemVerilog

Dans un premier temps, j'ai implémenté les 3 couches de la permutation en créant un module par couche avec le testbench associé qui est le module de test. Il est primordial de tester tous les cas possibles grâce au testbench. Par la suite, j'ai implémenté la permutation puis j'ai rajouté les XORs dans une deuxième itération et je l'ai enfin adaptée afin de pouvoir l'utiliser dans ma machine d'état. Pour finir, j'ai codé la machine d'état et enfin le module ascon_top reliant le tout afin de pouvoir tester le chiffrement.

Afin de bien me repérer, le signaux ont des noms choisis pour être adaptés à leur usage et sont suivis des préfixes _i pour les entrées, _s pour les signaux intermédiaires et _o pour les signaux de sortie.

Les différents modules nous ayant été fournis sont détaillés dans la sous-sous-section suivante.

1.4.3 Modules fournis

Pour nous aider à avancer efficacement, plusieurs modules nous ont été fournis :

- ***ascon_pack.sv*** : un package contenant le type type_state permettant de représenter l'état de 320 bits comme un 5 registres de 64 bits.
- ***xor_down.sv*** et ***xor_up.sv*** : ces deux modules représentent les XORs appliqués aux groupements de registres de 128 et 192 bits.
- ***ascon_top_tb.sv*** : un testbench permettant de tester le module *ascon_top* qui relie la machine d'état et tous les autres composants de l'algorithme.
- ***state_register.sv*** que j'ai renommé ***registre.sv*** qui représente un registre.
- ***compteur_double_init.sv*** : module permettant d'initialiser un compteur (ici le compteur de rondes) à 2 valeurs différentes (0 et 4 ici).

Le début du script de compilation ***compile_ascon*** nous a également été fourni.

2 Addition de constante

2.1 Explication théorique

La première couche de notre algorithme de chiffrement Ascon est la couche p_c qui correspond à l'addition de constante. Cette opération consiste à ajouter au registre S_2 une constante de ronde notée c_r. Elle dépend de la valeur r de la ronde de p^{12} ou p^8.

On peut retrouver les différentes valeurs de c_r dans le tableau suivant :

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
8	9	0000000000000000069
10	10	000000000000000005a
11	11	000000000000000004b

TABL. 1 : Table des valeurs ce c_r en fonction du numéro r de la ronde

Pour l'ajouter on utilise un XOR sur le dernier octet, ce qui donne de manière schématique :

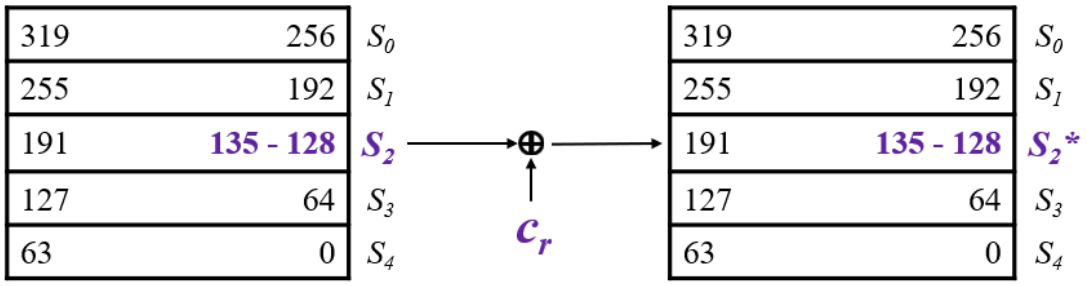

FIG. 4 : Schéma de l'addition de constante

2.2 Implémentation et simulation en SystemVerilog

On va créer un module *addition_constante* qui utilise 3 signaux :

- state_i : comprend l'état S de 320 bits d'entrée
- round_i : compteur de ronde sur 4 bits qui permet de déterminer la valeur à ajouter grâce au XOR
- state_o : comprend l'état S de 320 bits de sortie

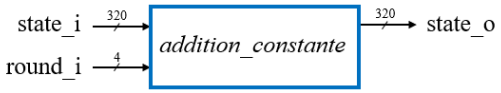

FIG. 5 : Représentation de l'état S comme tableaux de registres S_i

Les valeurs des constantes de rondes se trouvent dans le tableau *round_constant* dans le fichier *ascon_pack.sv* qui était fourni. On peut donc récupérer ainsi rapidement la valeur de la constante de ronde en fonction du numéro de la ronde *round_i*.

```
logic [7:0]  round_constant [0:11] = {8'hF0, 8'hE1, 8'hD2, 8'hC3, 8'hB4, 8'hA5,
8'h96, 8'h87, 8'h78, 8'h69, 8'h5A, 8'h4B};
```

C'est un tableau de 12 valeurs de chacune 8 bits.

Afin de tester ce module, j'ai réalisé un testbench (*addition_constante_tb.sv*) qui simule l'addition des constantes en partant de l'état initial S tel que :

$S_0 = 00001000808C0001$

$S_1 = 6CB10AD9CA912F80$

$S_2 = 691AED630E81901F$

$S_3 = 0C4C36A20853217C$

$S_4 = 46487B3E06D9D7A8$

On peut alors simuler et observer les résultats :

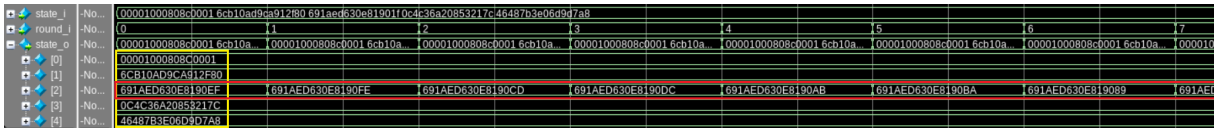

FIG. 6 : Simulation de l'addition de constante pour $c_0 \rightarrow c_6$

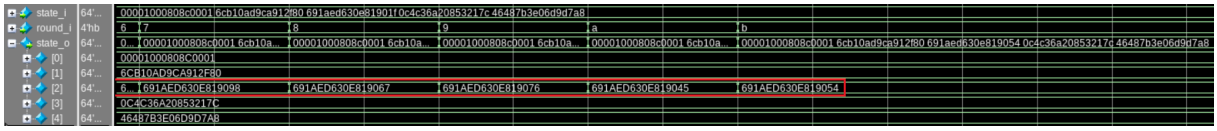

FIG. 7 : Simulation de l'addition de constante pour $c_7 \rightarrow c_{11}$

Après l'addition de la constante c_0, on obtient :

$S_0 = 00001000808C0001$

$S_1 = 6CB10AD9CA912F80$

$S_2 = 691AED630E8190\mathbf{EF}$

$S_3 = 0C4C36A20853217C$

$S_4 = 46487B3E06D9D7A8$

On obtient bien la même valeur que sur le sujet.

On peut vérifier que l'addition des autres constantes fonctionne également. Voici ce qu'on cherche à obtenir :

r	**Constante c_r**	**S_2 après le XOR**
0	00000000000000000f0	691aed630e8190ef
1	00000000000000000e1	691aed630e8190fe
2	00000000000000000d2	691aed630e8190cd
3	00000000000000000c3	691aed630e8190dc
4	00000000000000000b4	691aed630e8190ab
5	00000000000000000a5	691aed630e8190ba
6	00000000000000000096	691aed630e819089
7	00000000000000000087	691aed630e819098
8	00000000000000000078	691aed630e819067
9	00000000000000000069	691aed630e819076
10	0000000000000000005a	691aed630e819045
11	0000000000000000004b	691aed630e819054

TABL. 2 : Table des valeurs de S_2 après chaque addition de constante

A chaque fois, on repart de l'état initial et on effectue l'addition de la constante. Grâce à la simulation, en zoomant pour pouvoir observer tous les bits, on observe bien que toutes les additions sont correctes et que les registres autres que S_2 sont bien inchangés.

Mon module fonctionne donc correctement.

3 Couche de substitution

La deuxième couche dont nous avons besoin est la couche de substitution aussi appelée p_s. Le principe de cette couche est de faire des substitutions sur toutes les colonnes de 5 bits de notre état. On va dans un premier temps créer la S-box qui définit les permutations puis la relier au reste.

3.1 Théorie de la S-box

Dans la S-box, on a une valeur d'entrée x et à cette dernière, on associe une valeur substituée. Toutes les substitutions sont représentées dans les 2 tableaux suivants :

x	**00**	**01**	**02**	**03**	**04**	**05**	**06**	**07**	**08**	**09**	**0A**	**0B**	**0C**	**0D**	**0E**	**0F**
Sbox(x)	04	0B	1F	14	1A	15	09	02	1B	05	08	12	1D	03	06	1C

x	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**1A**	**1B**	**1C**	**1D**	**1E**	**1F**
Sbox(x)	1E	13	07	0E	00	0D	11	18	10	0C	01	19	16	0A	0F	17

TABL. 4 : Tableaux des valeurs substituées par la S-box

Schématiquement, voici comment la S-box fonctionne :

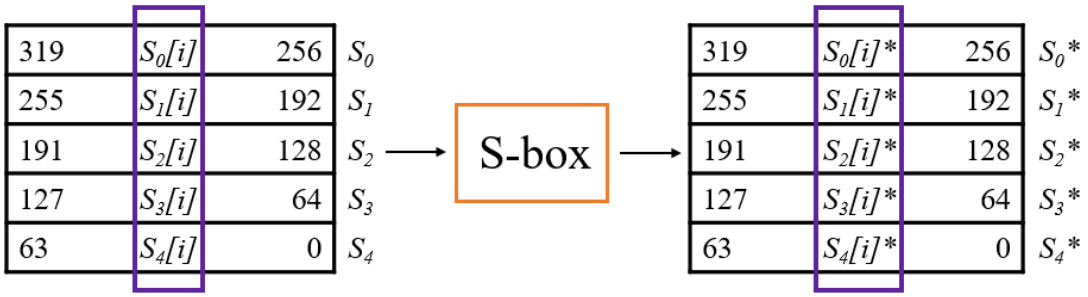

Fig. 8 : Schéma de la S-box

3.2 Implémentation de la S-box en SystemVerilog

Pour implémenter ces valeurs en SystemVerilog dans le module *sbox*, j'ai utilisé un *case* sur l'entrée afin ensuite d'assigner chaque valeur de sortie à chaque valeur d'entrée. J'ai également rajouté un cas par défaut renvoyant 0.

Ce module utilise 2 signaux de 5 bits :

- sbox_i qui représente la colonne de 5 bits à l'entrée de la S-box
- sbox_o qui représente la colonne de 5 bits après la substitution

On ne va pas tester ce module avec un testbench car on va se servir d'un autre module (*couche_substitution*) pour relier la S-box à nos données et c'est ce module que l'on va tester.

3.3 Implémentation et tests du module de la couche de substitution

Comme expliqué précédemment, j'ai créé le module *couche_substitution* pour relier la S-box au reste. Voici comment cela fonctionne schématiquement :

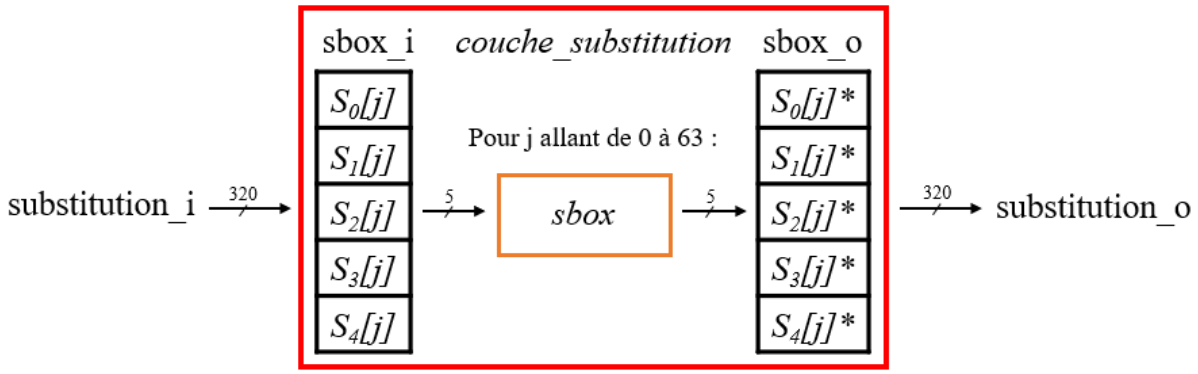

Fig. 9 : Schéma de la couche de substitution

Afin de tester ce module, j'ai réalisé un testbench (*couche_substitution_tb*). Je lui donne l'état S après la première addition de constante :

S_0 = 00001000808c0001

S_1 = 6cb10ad9ca912f80

S_2 = 691aed630e8190ef

S_3 = 0c4c36a20853217c

S_4 = 46487b3e06d9d7a8

Il va donc récupérer chaque colonne de 5 bits et les substituer grâce à la S-box.

Voici le résultat de la simulation :

FIG. 10 : Simulation de la couche de substitution

On obtient bien le même résultat que sur le sujet :

S_0 = 25f7c341c45f9912

S_1 = 23b794c540876856

S_2 = b85451593d679610

S_3 = 4fafba264a9e49ba

S_4 = 62b54d5d460aded4

Le module fonctionne donc correctement.

4 Diffusion linéaire

4.1 Explication théorique

La couche de diffusion linéaire, aussi notée p_l est la troisième et dernière couche de notre permutation. Elle consiste en le fait d'appliquer une diffusion à l'état S en utilisant 2 XORs sur chaque ligne S_i de 64 bits afin de les modifier. Voici les opérations utilisées :

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

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

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

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

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

L'opérateur $\ggg$ représente un décalage arithmétique à droite qui est ici cyclique, c'est à dire qu'il y aura toujours les mêmes valeurs dans la lignes mais qu'elles sont décalées cycliquement du nombre de bits suivant l'opérateur.

4.2 Implémentation en SystemVerilog et simulation

Pour implémenter la couche de diffusion linéaire, j'ai créé le module *diffusion_lineaire* qui est composé de diffusion_i et diffusion_o qui contiennent l'état avant la diffusion et l'état après la diffusion. Ce qui donne schématiquement :

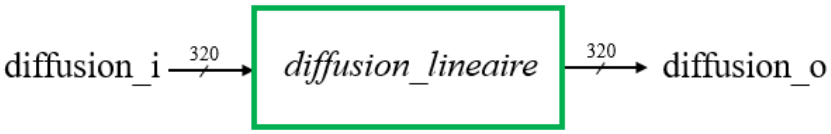

Fig. 11 : Schéma de la couche de diffusion linéaire

Pour implémenter ces décalages, je découpe mon registre S_i appelé ici diffusion_i en fonction des bits dont j'ai besoin pour les XORs afin de bien rendre le décalage cyclique. Je concatène ensuite les différentes parties découpées avant de faire le XOR.

J'ai réalisé un testbench pour ce module qui va initialiser S à sa valeur après la première addition de constante et la substitution, ce qui donne :

S_0 = 00001000808c0001

S_1 = 6cb10ad9ca912f80

S_2 = 691aed630e8190ef

S_3 = 0c4c36a20853217c

S_4 = 46487b3e06d9d7a8

Voici le résultat de la simulation :

Fig. 12 : Simulation de la couche de diffusion

On obtient bien le même résultat que sur le sujet :

S_0 = 932c16dd634b9585

S_1 = b48a3c3fe8fb45ce

S_2 = a69f28b0c721c340

S_3 = 05e1761f1e1fcb67

S_4 = 64d322a896b791cf

Le module fonctionne donc correctement.

Nos 3 couches fonctionnent donc toutes correctement, il faut désormais les assembler afin de réaliser le module de permutation.

5 Permutation sans les XORs

5.1 Principe général de la permutation

Les 3 couches rassemblées forment la permutation qui est un des éléments clés de notre algorithme de chiffrement Ascon. Ici, on utilise 2 permutations, l'une fait 8 rondes (p^8) et l'autre 12 (p^{12}), tout le reste est identique. Voici comment sont assemblées nos blocs :

$$p = p_c \circ p_s \circ p_l$$

Il faut également un multiplexeur pour déterminer si on reprend la sortie de la permutation pour la nouvelle ronde ou si on prend l'entrée. On a aussi besoin d'un registre pour stocker les valeurs de sortie avec la possibilité d'activer l'écriture dans ce dernier ou non.

Afin de segmenter le travail et les tests, j'ai réalisé plusieurs modules pour cette permutation. Dans un premier temps une permutation simple puis une permutation avec les XORs et enfin j'ai adapté cette dernière afin de pouvoir l'utiliser dans mon top level avec ma machine d'état. La section 5. traite de la première permutation implémentée, la permutation simple.

On peut représenter cette permutation simple de manière schématique ainsi :

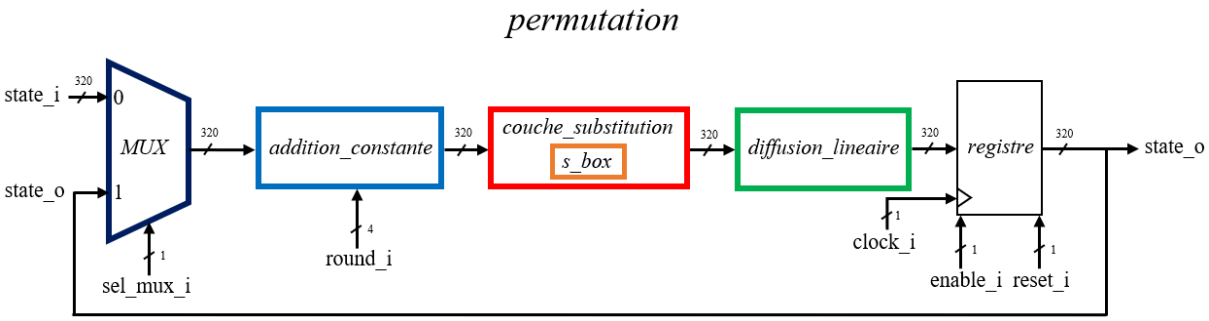

FIG. 13 : Schéma du module *permutation*

Je me sert donc du module ***registre.sv*** pour implémenter le registre et je représente le multiplexeur directement dans le code :

```
assign state_mux_s = (sel_mux_i) ? state_o : state_i;
```

Ainsi, dans `state_mux_s` je récupère l'état d'entrée ou de sortie de ma permutation en fonction du sélecteur `sel_mux_i`. Si `sel_mux_i` vaut 1 je récupère l'état de sortie `state_o`, sinon, l'état d'entrée `state_i`.

5.2 Implementation en SystemVerilog et simulation

Afin de représenter cette permutation, on se sert de tous les modules précédemment codés que l'on combine afin que la sortie de l'un soit l'entrée du suivant. Ainsi, on peut réaliser les rondes et tester les 3 couches de la permutation ensembles.

Pour vérifier le bon fonctionnement de ce module, j'ai implémenté un testbench qui teste la permutation simple sur 12 rondes et qui prend pour entrée la valeur initiale de S.

Voici le résultat de la simulation :

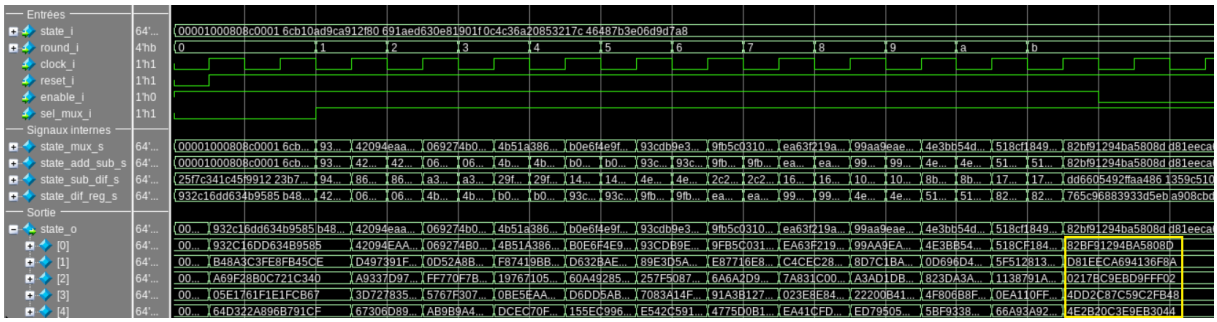

FIG. 14 : Simulation de la permutation simple

On remarque que l'on obtient bien le même état de sortie qu'indiqué sur le sujet :

S_0 = 82bf91294ba5808d

S_1 = d81eeca694136f8a

S_2 = 0217bc9ebd9fff02

S_3 = 4dd2c87c59c2fb48

S_4 = 4e2b20c3e9eb3044

Le module représentant la permutation simple fonctionne donc correctement.

6 Permutation avec les XORs

6.1 Première version avec seulement les XORs d'ajoutés

6.1.1 Explication des modifications

Dans un premier temps j'ajoute uniquement les XORs à la permutation afin de poursuivre mes tests et mes simulations et ainsi valider le bon fonctionnement de mon algorithme de manière progressive.

On ajoute alors 2 modules :

- ***xor_up*** : module qui permet de réaliser les XORs sur les 2 premiers registres S_0 et S_1. Il réalise le XOR entre ces registres et une donnée qui est transmise par le signal *data_i*.
- ***mux_xor_down*** : module qui permet de réaliser les XORs sur les 3 derniers registres S_2, S_3 et S_4 en utilisant le module ***xor_down*** et en choisissant les données à XOR grâce à un multiplexeur.

Cela donne donc de manière schématique :

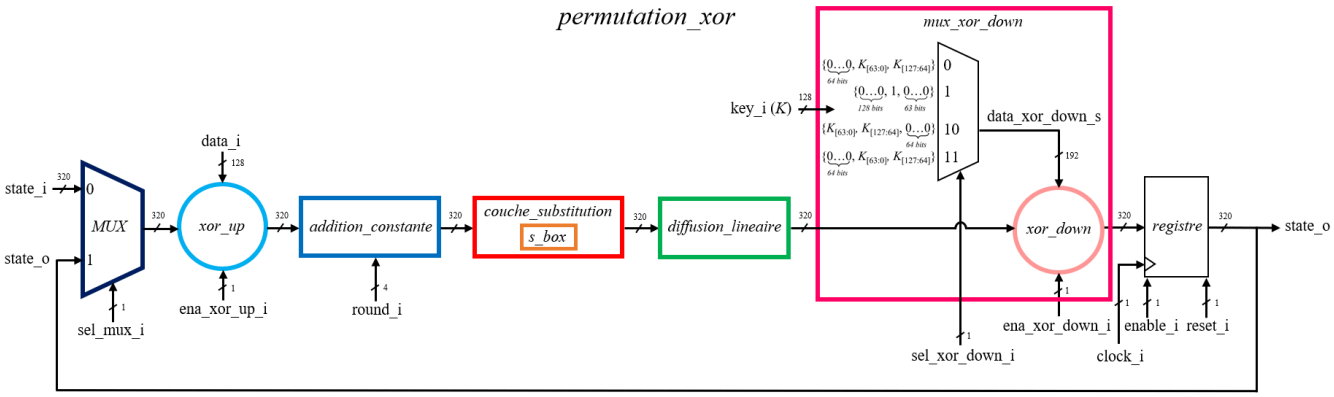

FIG. 15 : Schéma du module *permutation_xor*

6.1.2 Simulation du module

Le testbench va tester la permutation avec le xor_down utilisant la première valeur du multiplexeur. On part donc ici aussi de l'état à sa valeur initiale.

Voici le résultat de la simulation :

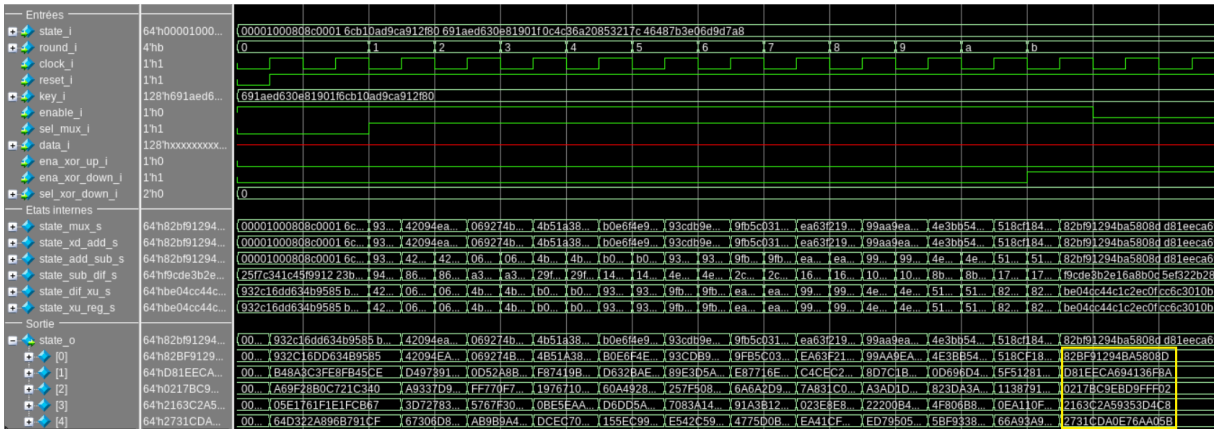

FIG. 16 : Simulation de la permutation simple

On obtient bien le même état de sortie qu'indiqué sur le sujet :

S_0 = 82bf91294ba5808d

S_1 = d81eeca694136f8a

S_2 = 0217bc9ebd9fff02

S_3 = 2163c2a59353d4c8

S_4 = 2731cda0e76aa05b

Le signal *data_i* apparaît en rouge car il n'est jamais initialisé dans le testbench mais cela ne pose pas de problème car on n'utilise pas le XOR up dans notre simulation.

La permutation avec les XORs fonctionne donc correctement.

6.2 Version finale de la permutation

6.2.1 Explication des modifications

J'ajoute maintenant les registres permettant de mémoriser le cipher et le tag. Il faut donc rajouter le calcul de ces 2 signaux ainsi que les signaux permettant d'utiliser les registres.

Pour implémenter les registres, je crée une copie de mon module ***registre*** que je nomme ***registre_cipher_tag*** et je remplace l'entrée et la sortie de type `type_state` par un type `logic [127:0]` car le cipher et le tag sont tous deux sur 128 bits. Il suffit donc de les instancier avec un nom différent dans le nouveau module de permutation que je nomme ***permutation_xor_top***.

Ce module n'étant que peu différent du précédent et n'utilisant pas ce qui concerne le cipher et le tag pendant l'initialisation, je n'ai pas réalisé de testbench pour ce module mais je vais pouvoir vérifier son bon fonctionnement grâce au testbench du module ***ascon_top***.

On obtient alors le schéma suivant :

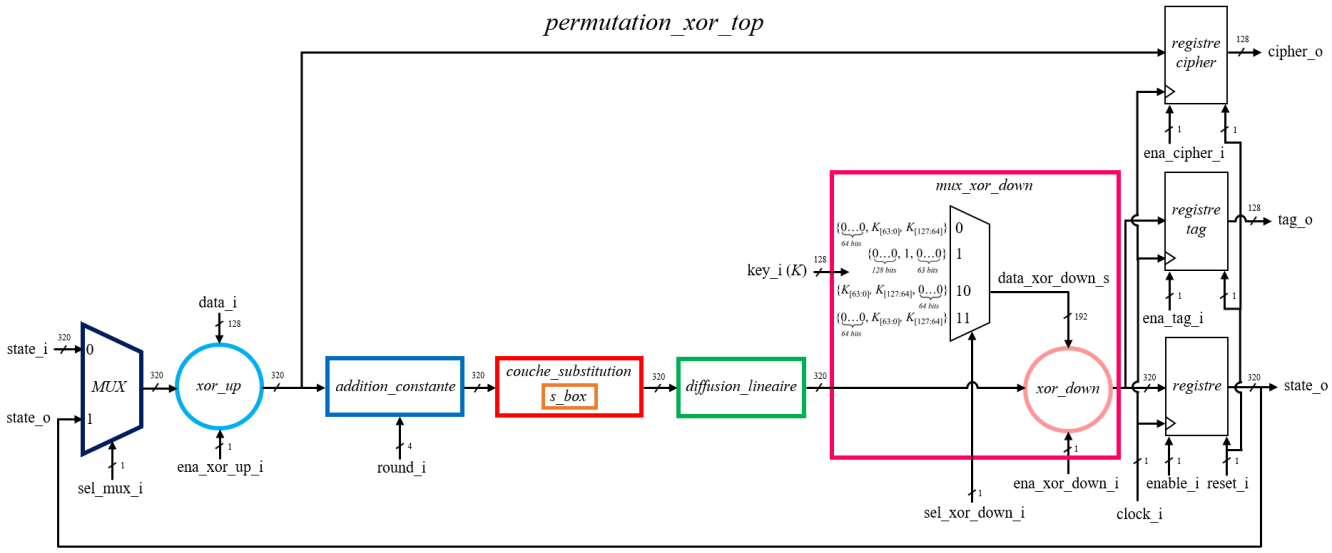

FIG. 17 : Schéma du module *permutation_xor_top*

7 Machine d'état de Moore

7.1 Explications de la machine d'état

Afin de pouvoir contrôler efficacement tous les signaux de notre algorithme nous utilisons une machine d'état et plus précisément une machine de Moore dans mon implémentation. On l'appelle également FSM pour Finite State Machine.

Elle est composée de deux processus combinatoires différents. Le premier est celui qui représente les changements d'état tandis que le deuxième décrit les changements des valeurs des sorties à chaque état.

7.2 Graphe des états

Afin de mieux comprendre comment elle fonctionne, les états qui la compose et les changements de sorties, on peut se référer au graphe des états suivant :

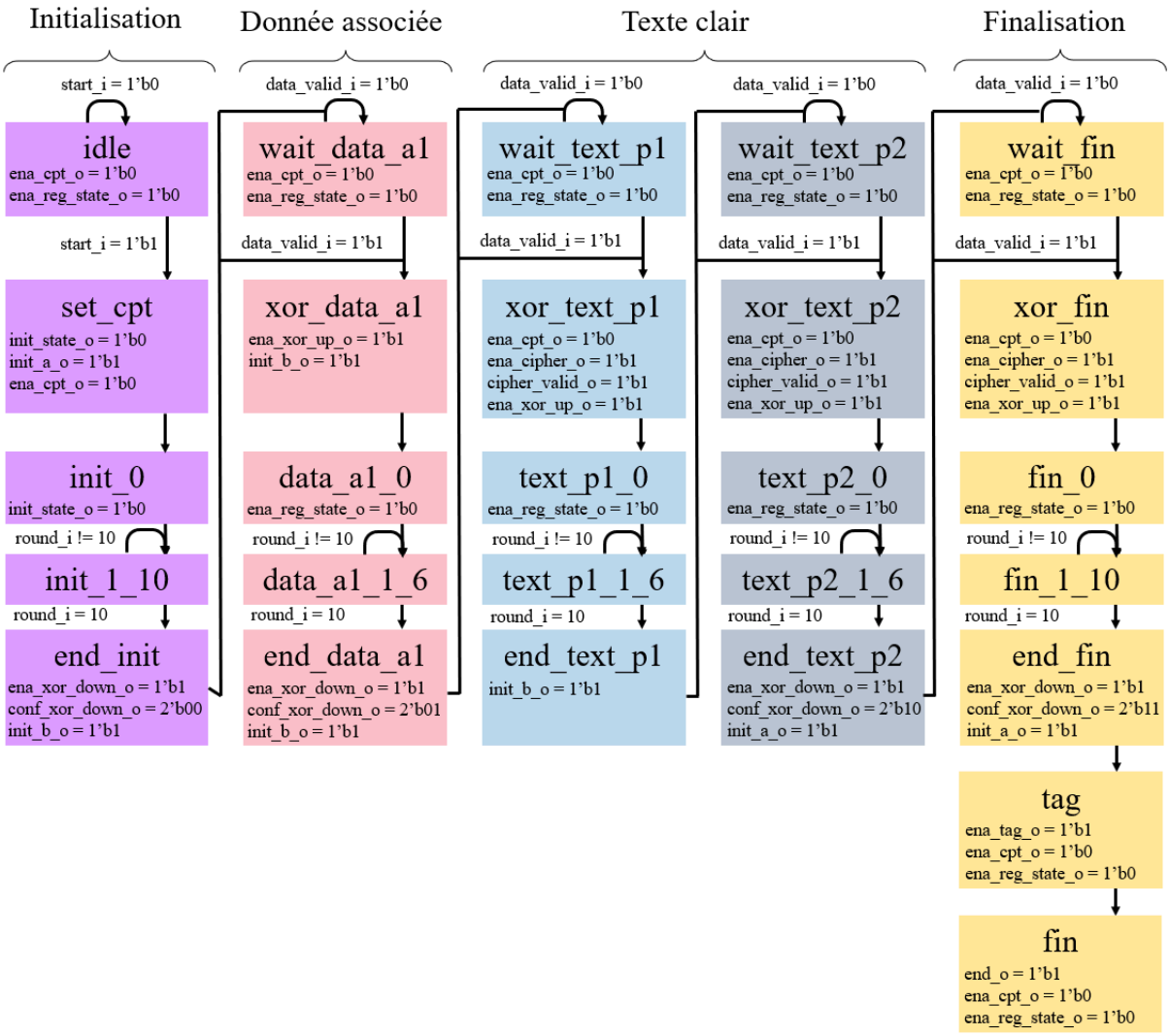

FIG. 18 : Graphe des états de la fsm

On retrouve plusieurs fois la même structure découpée ainsi :

- Un état d'attente qui boucle sur lui même : **idle**, **wait_data_a1**, **wait_text_p1**, **wait_text_p2** et **wait_fin**.
- Un état d'initialisation du compteur de rondes pour l'initialisation **set_cpt** et un état utilisé pour le xor up dans les blocs suivants : **xor_data_a1**, **xor_text_p1**, **xor_text_p2** et **xor_fin**.
- Un état qui réalise la première ronde de la permutation : **init_0**, **data_a1_0**, **text_p1_0**, **text_p2_0**, **fin_0**.
- Un état où sont réalisées toutes les rondes suivantes sauf la dernière : **init_1_10**, **data_a1_1_6**, **text_p1_1_6**, **text_p2_1_6**, **fin_1_10**.

– Un état de fin de permutation qui réalise la dernière ronde avec le xor down avec la bonne valeur lorsqu'il est nécessaire.

Il y a également 2 états supplémentaires :

– **tag** qui permet de récupérer le tag et de l'écrire dans le registre dédié.

– **fin** qui permet de terminer correctement notre algorithme de chiffrement Ascon.

7.3 Tables de vérité

Pour plus de lisibilité, j'ai rassemblé les signaux de sortie et leur valeur à chaque état dans des tables de vérité par bloc.

7.3.1 Initialisation

	idle	**set_cpt**	**init_0**	**init_1_10**	**end_init**
cipher_valid_o	0	0	0	0	0
end_o	0	0	0	0	0
init_state_o	1	0	0	1	1
ena_cpt_o	0	0	1	1	1
init_a_o	0	1	0	0	0
init_b_o	0	0	0	0	1
ena_xor_up_o	0	0	0	0	0
ena_xor_down_o	0	0	0	0	1
conf_xor_down_o	00	00	00	00	00
ena_reg_state_o	0	1	1	1	1
ena_cipher_o	0	0	0	0	0
ena_tag_o	0	0	0	0	0

TABL. 5 : Table des valeurs des sorties dans chaque état de l'initialisation

7.3.2 Donnée associée

Afin que tous les états puissent rentrer dans le tableau, je raccourcis leur nom en enlevant « data_ ».

	wait_a1	**xor_a1**	**a1_0**	**a1_1_6**	**end_a1**
cipher_valid_o	0	0	0	0	0
end_o	0	0	0	0	0
init_state_o	1	1	1	1	1
ena_cpt_o	0	1	1	1	1
init_a_o	0	0	0	0	0
init_b_o	0	1	0	0	1
ena_xor_up_o	0	1	0	0	0
ena_xor_down_o	0	0	0	0	1
conf_xor_down_o	00	00	00	00	01
ena_reg_state_o	0	1	0	1	1
ena_cipher_o	0	0	0	0	0
ena_tag_o	0	0	0	0	0

TABL. 6 : Table des valeurs des sorties dans chaque état du bloc donnée associée

7.3.3 Texte clair

Je fais de même pour p1 et p2 en enlevant « text_ ».

	wait_p1	**xor_p1**	**p1_0**	**p1_1_6**	**end_p1**
cipher_valid_o	0	1	0	0	0
end_o	0	0	0	0	0
init_state_o	1	1	1	1	1
ena_cpt_o	0	0	1	1	1
init_a_o	0	0	0	0	0
init_b_o	0	0	0	0	1
ena_xor_up_o	0	1	0	0	0
ena_xor_down_o	0	0	0	0	0
conf_xor_down_o	00	00	00	00	00
ena_reg_state_o	0	1	0	1	1
ena_cipher_o	0	1	0	0	0
ena_tag_o	0	0	0	0	0

TABL. 7 : Table des valeurs des sorties dans chaque état de p1 du texte clair

	wait_p2	xor_p2	p2_0	p2_1_6	end_p2
cipher_valid_o	0	1	0	0	0
end_o	0	0	0	0	0
init_state_o	1	1	1	1	1
ena_cpt_o	0	0	1	1	1
init_a_o	0	0	0	0	1
init_b_o	0	0	0	0	0
ena_xor_up_o	0	1	0	0	0
ena_xor_down_o	0	0	0	0	1
conf_xor_down_o	00	00	00	00	10
ena_reg_state_o	0	1	0	1	1
ena_cipher_o	0	1	0	0	0
ena_tag_o	0	0	0	0	0

TABL. 8 : Table des valeurs des sorties dans chaque état de p2 du texte clair

7.3.4 Finalisation

	wait_fin	xor_fin	fin_0	fin_1_10	end_fin	tag	fin
cipher_valid_o	0	1	0	0	0	0	0
end_o	0	0	0	0	0	0	1
init_state_o	1	1	1	1	1	1	1
ena_cpt_o	0	0	1	1	1	0	0
init_a_o	0	0	0	0	1	0	0
init_b_o	0	0	0	0	0	0	0
ena_xor_up_o	0	1	0	0	0	0	0
ena_xor_down_o	0	0	0	0	1	0	0
conf_xor_down_o	00	00	00	00	11	00	00
ena_reg_state_o	0	1	0	1	1	0	0
ena_cipher_o	0	1	0	0	0	0	0
ena_tag_o	0	0	0	0	0	1	0

TABL. 9 : Table des valeurs des sorties dans chaque état de la finalisation

8 Top level

8.1 Théorie du top level

Dans le but d'automatiser notre algorithme à travers sa machine d'état, nous utilisons un module ***ascon_top***, représentant le top level de notre projet, qui permet d'instancier les modules nécessaires au chiffrement ensemble. La machine d'état y est reliée à la permutation xor finale ainsi qu'au compteur de ronde double qui permet d'initialiser le compteur de rondes à 0 ou à 4 en fonction de si l'on souhaite faire p^{12} ou p^8.

J'ai choisi de ne pas utiliser le compteur de blocs pour simplifier dans un premier temps l'implémentation de ma machine d'état et la résolution des problèmes en résultant.

Voici une représentation schématique de ce module :

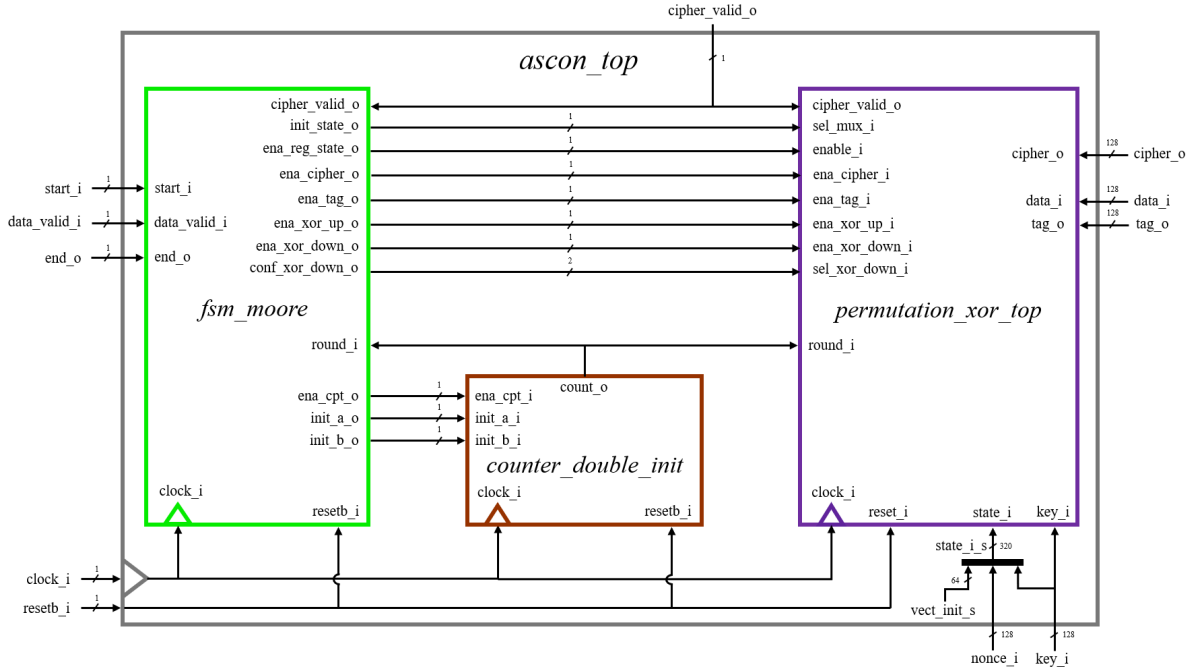

FIG. 19 : Schéma du module ***ascon_top***

Pour former le signal state_i_s, on utilise les 3 signaux vect_init_s, key_i et nonce_i qu'on utilise ainsi :

$S_0 = vect\_init\_s = 64'h00001000808c0001$

$S_1 = key\_i[63:0]$

$S_2 = key\_i[127:64]$

$S_3 = nonce\_i[63:0]$

$S_4 = nonce\_i[127:64]$

On n'a désormais plus qu'à simuler et tester nos modules afin de vérifier le bon fonctionnement de notre algorithme de chiffrement Ascon.

8.2 Testbench et simulation

8.2.1 Explications

Afin de vérifier le bon fonctionnement de notre module ***ascon_top*** et en même temps de la machine d'état et de notre algorithme complet, j'ai utilisé le testbench ***ascon_top_tb*** que j'ai du légèrement modifier à 2 endroits. En effet, dû à mon implémentation de la machine d'état, j'avais une erreur à cause de l'instant d'activation de data_valid_i qui était décalé d'un

demi puis d'un cycle d'horloge. J'ai donc augmenté le temps en ns d'attente avant de mettre data_valid_i à 1.

8.2.2 Simulation de la FSM grâce au testbench

Grâce au testbench du top level on peut également simuler la machine d'état.

On obtient alros :

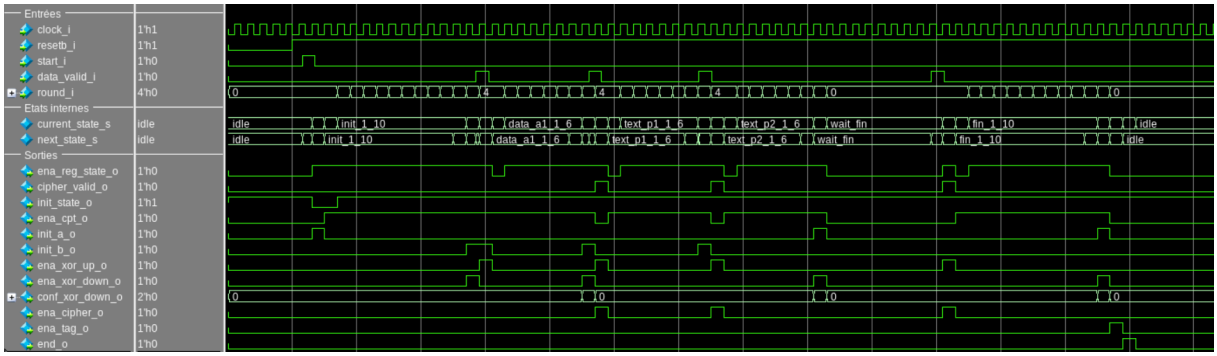

Fig. 20 : Simulation de la FSM

On remarque que l'on passe par l'état d'attente pour le cipher 3 et c'est bien ce qui est attendu comme indiqué dans en commentaire dans le fichier du testbench.

J'ai choisi volontairement d'initialiser à nouveau le compteur à 0 à la fin afin d'arriver dans l'état idle avec un compteur nul mais ce n'est vraisemblablement pas nécessaire.

8.2.3 Simulation du top level

Je me concentre désormais sur les signaux du module ***ascon_top***.

8.2.3.1 Simulation générale

Voici ce que donne la simulation du module :

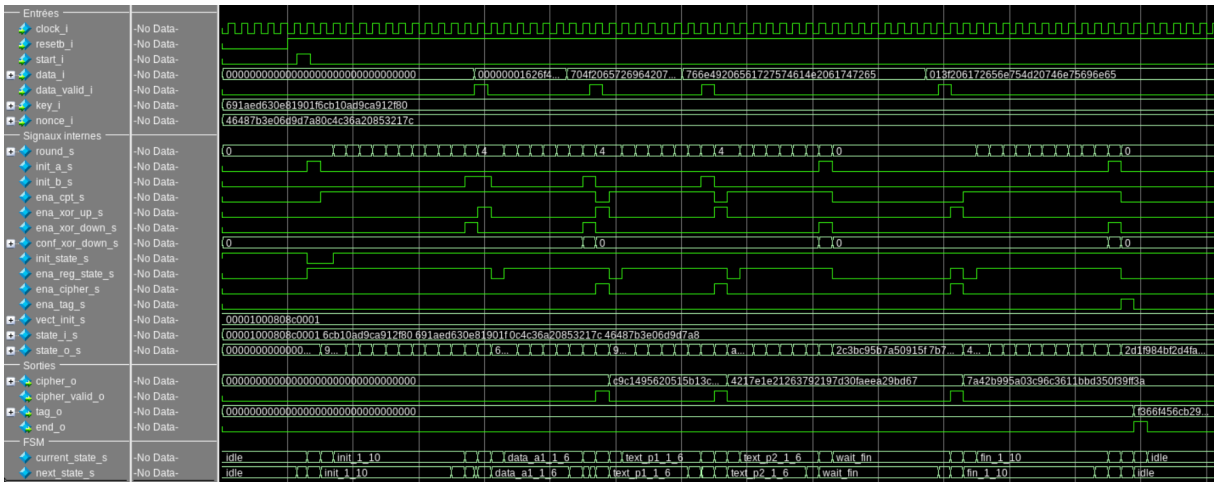

Fig. 21 : Simulation du top level

J'ai également rajouté dans la fenêtre Wave de ModelSim les 2 signaux internes de la FSM indiquant l'état actuel et l'étant suivant afin de pouvoir mieux me repérer et vérifier que tout fonctionne correctement, ce qui est le cas.

Au vu du temps de simulation, on ne peut pas relever les valeurs sur cette image de la Wave. Je vais donc détailler le bon fonctionnement de mon algorithme dans les prochaines sections.

8.2.3.2 Initialisation

Pour la phase d'initialisation, on cherche à vérifier que l'on obtient bien la même chose que dans le testbench de la permutation xor, c'est à dire le résultat de 12 rondes avec un XOR. On obtient alors :

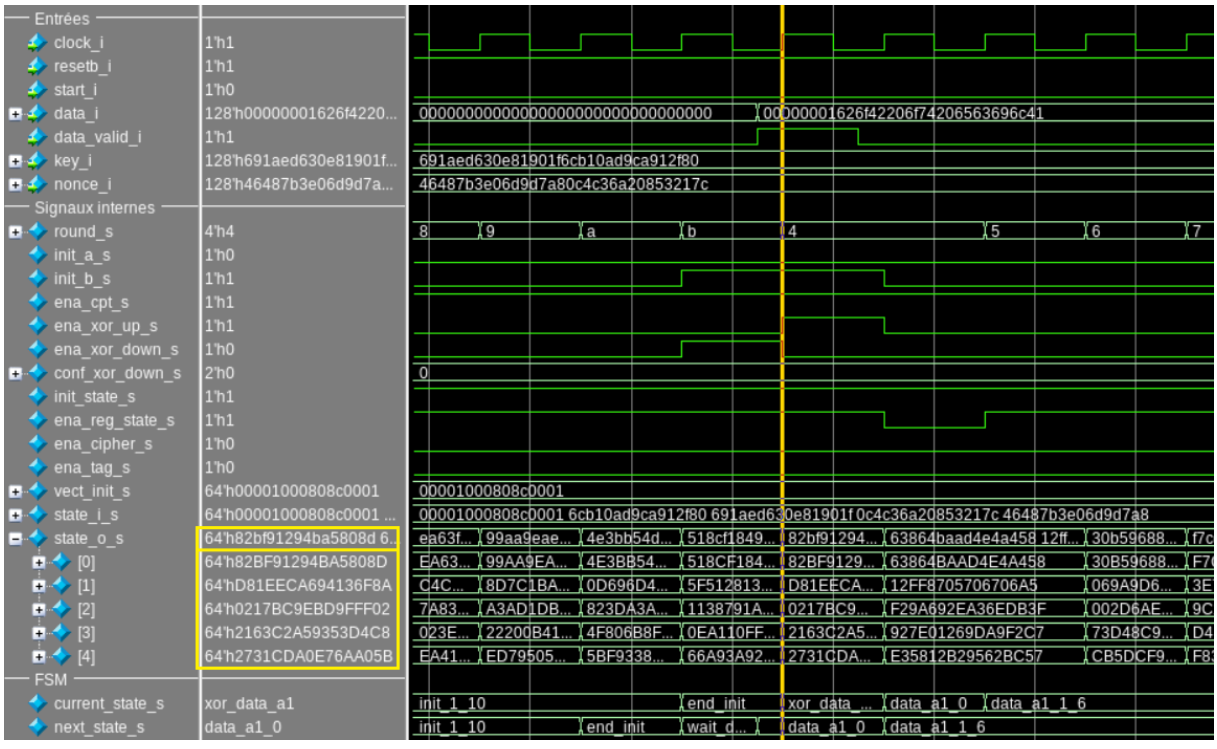

Fig. 22 : Simulation de l'initialisation

J'obtiens bien la valeur attendue à la fin de l'initialisation.

8.2.3.3 Donnée associée A

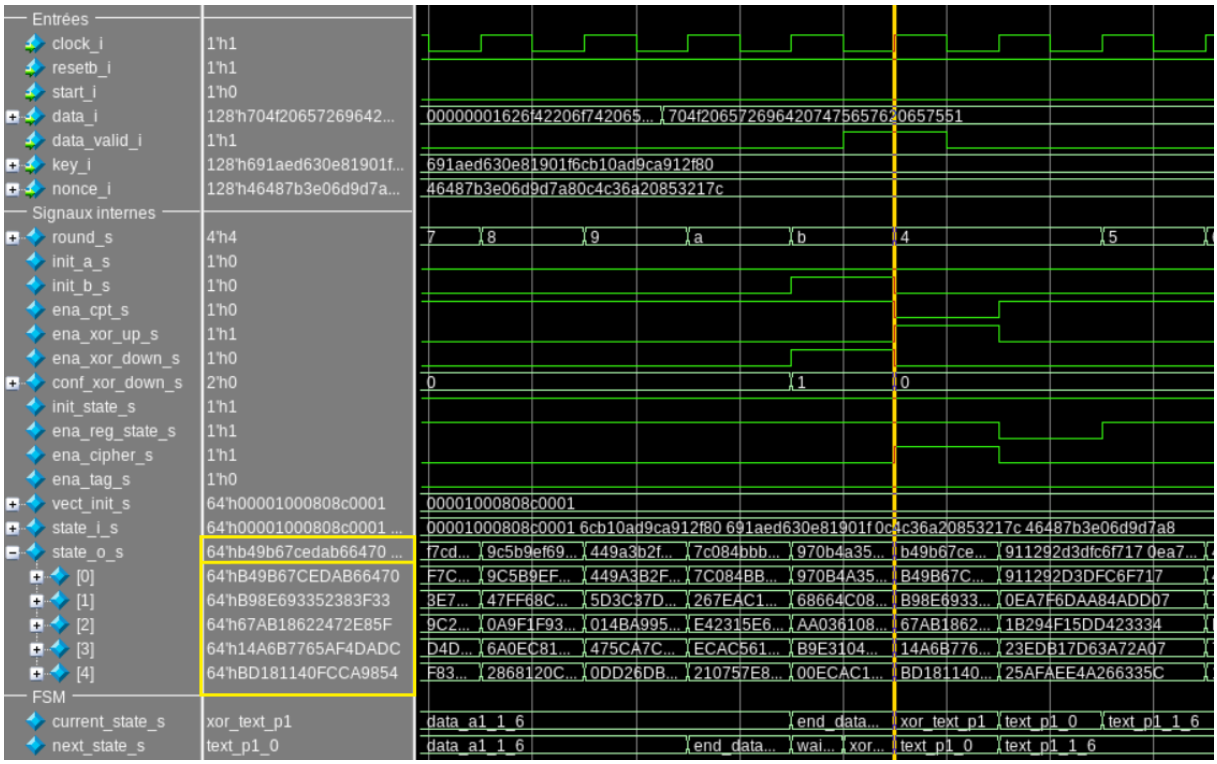

Fig. 23 : Simulation jusqu'à la donnée associée

Ici aussi, j'obtiens le bon résultat.

8.2.3.4 Texte chiffré C1

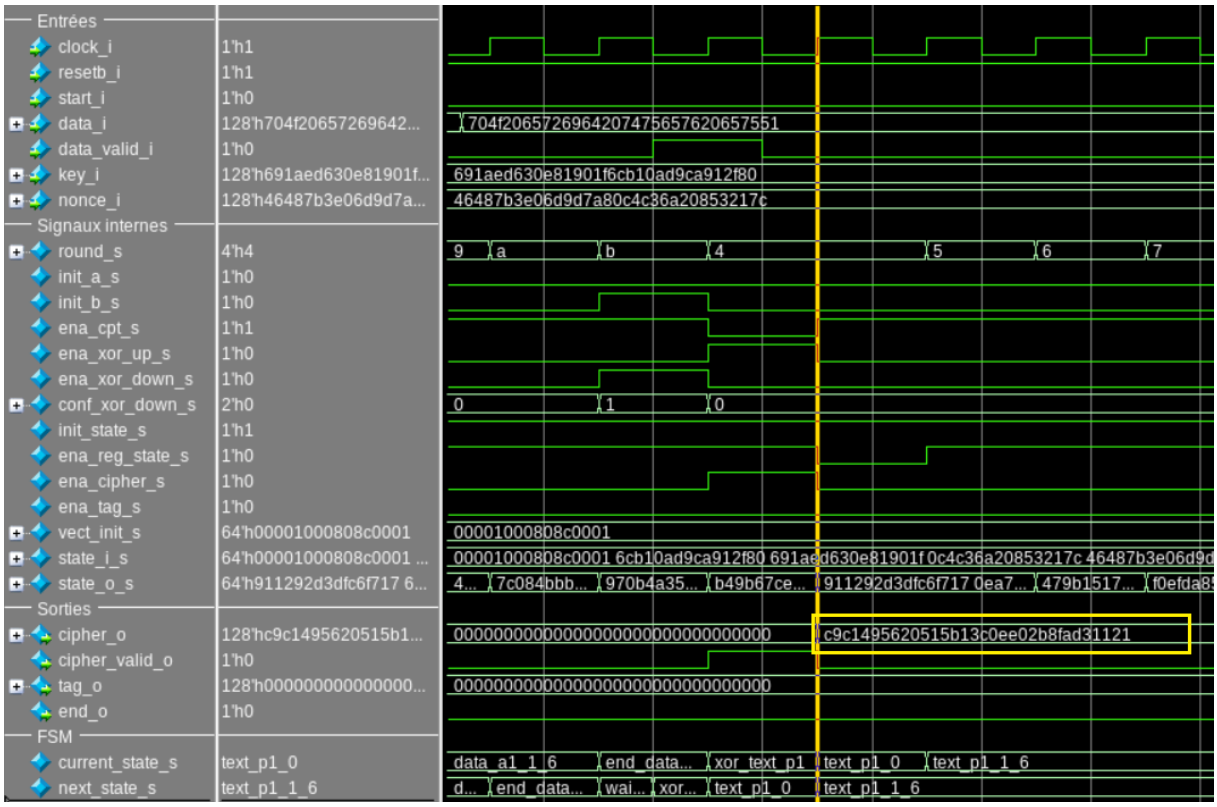

Fig. 24 : Simulation du cipher 1

Le *cipher_valid_o* s'active lors du coup d'horloge précédent l'obtention de la valeur dans *cipher_o*, ce qui s'explique par le besoin d'avoir un front montant d'horloge pour enregistrer la valeur du cipher dans le registre dédié.

La valeur de C1 est correcte.

8.2.3.5 Texte chiffré C2

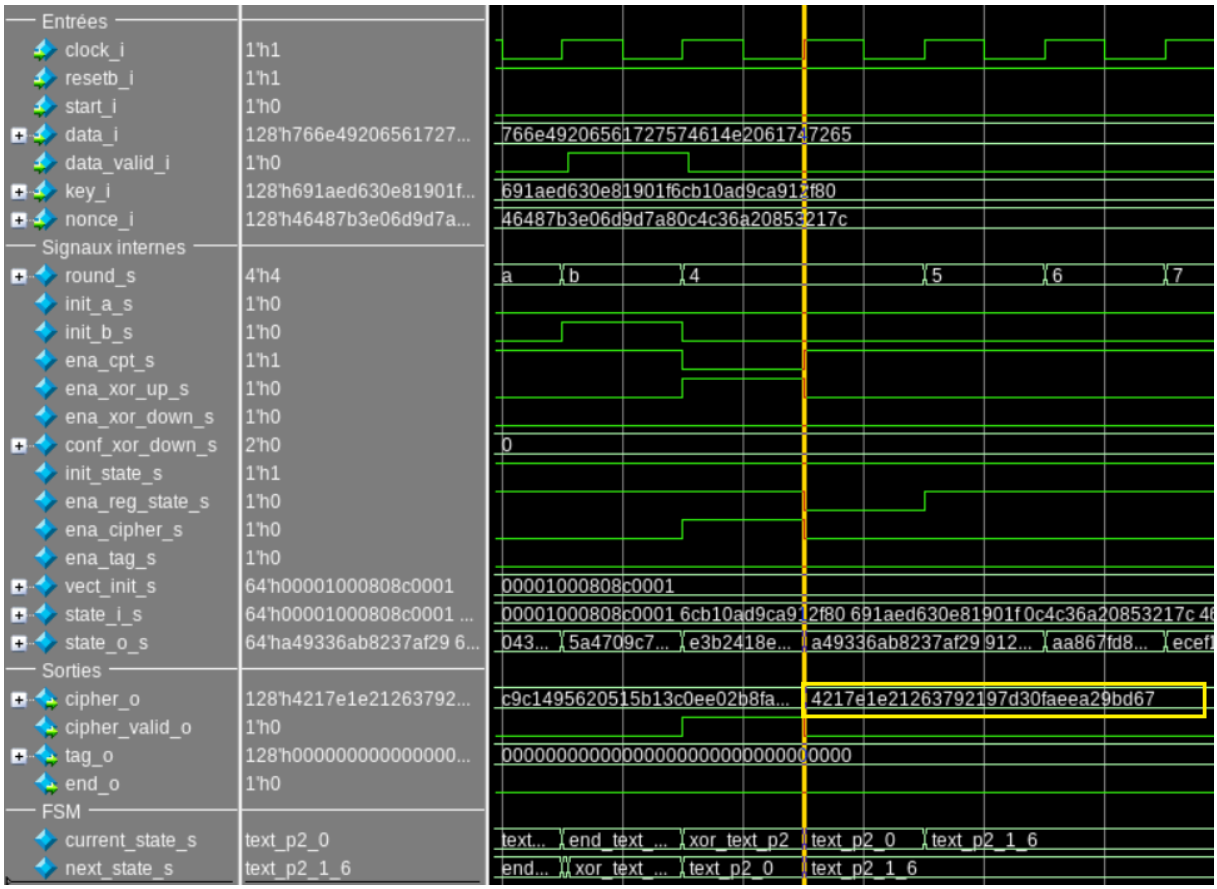

Fig. 25 : Simulation du cipher 2

La valeur de C2 est, elle aussi, identique au sujet.

8.2.3.6 Texte chiffré C3

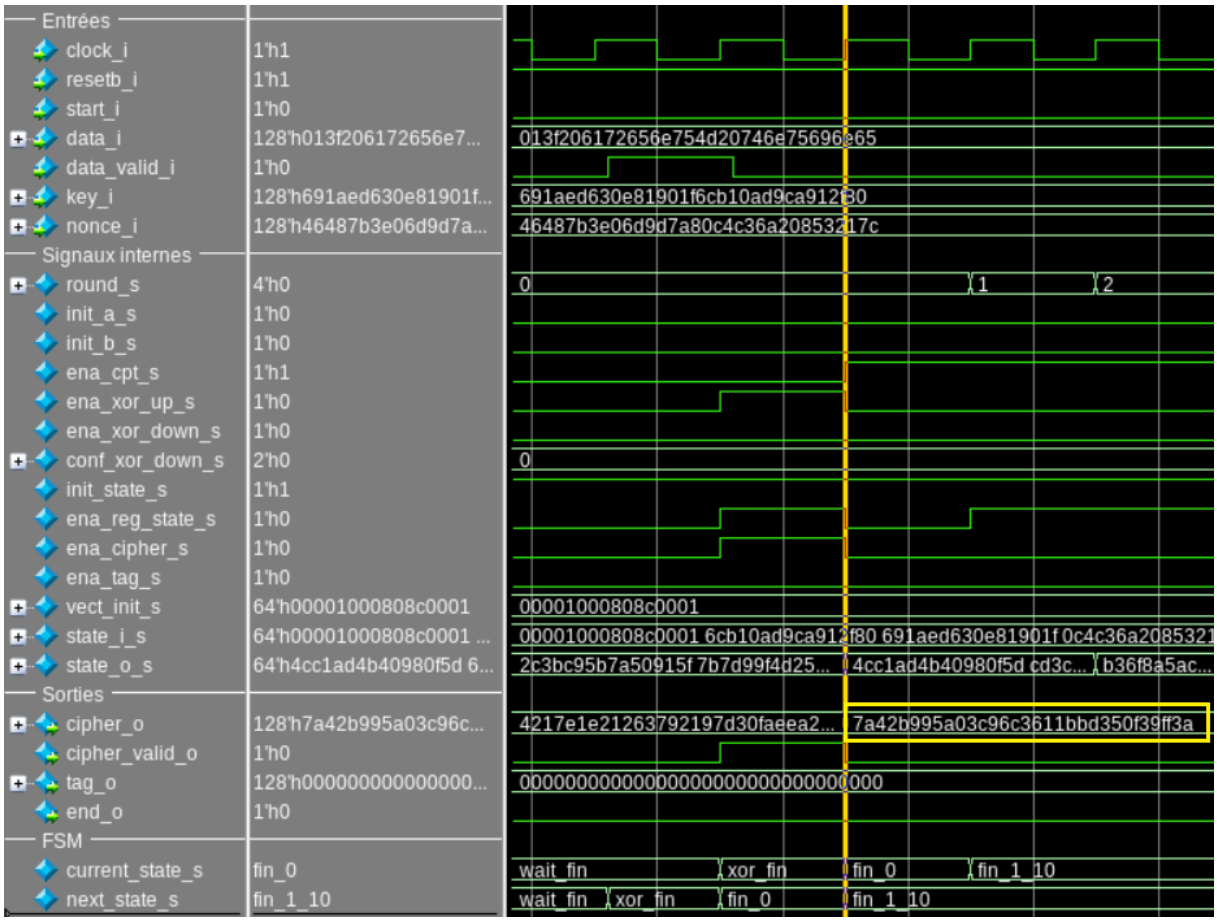

Fig. 26 : Simulation du cipher 3

Ici, on récupère bien la bonne valeur si on enlève le premier octet, c'est à dire le « **ca** ». Cela est dû au fait que le C3 n'est pas sur 16 octets, c'est à dire 128 bits, comme C1 et C2 mais sur 15 octets soit 120 bits. Cependant, ici je garde le cipher dans un registre de 128 bits.

La valeur est donc bien correcte.

8.2.3.7 Tag

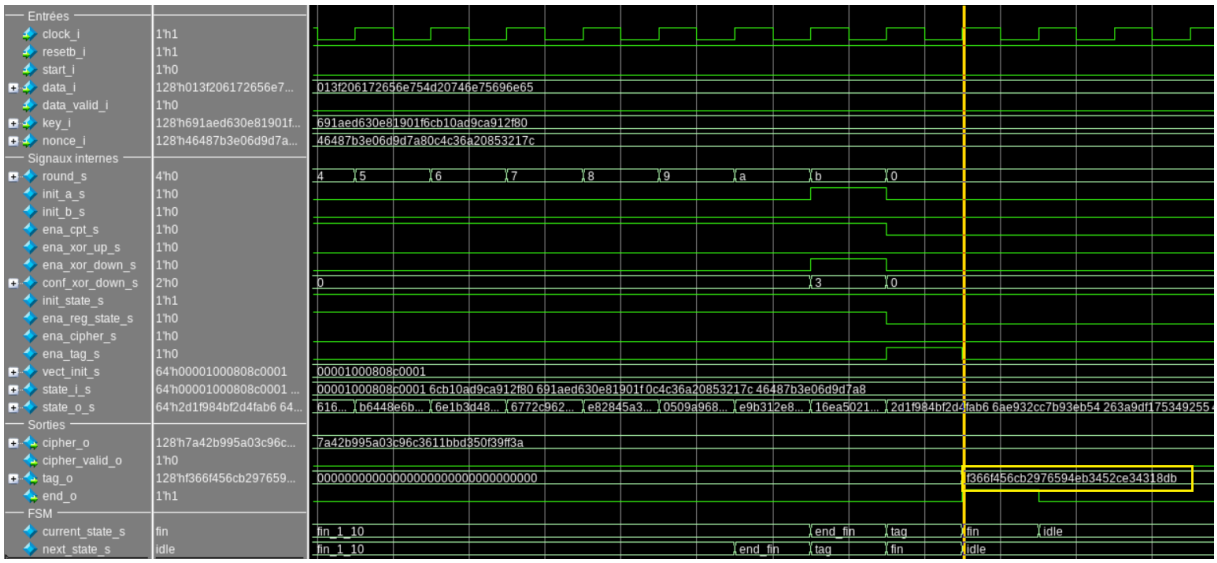

Fig. 27 : Simulation du tag

Finalement, j'obtiens le bon tag et mon algorithme fonctionne correctement jusqu'à la fin.

De plus, cela se vérifie avec le message dans l'onglet « Transcript » de ModelSim qui confirme que mon cipher et mon tag sont bien identiques à ceux attendus et donc que mon algorithme est fonctionnel.

```
VSIM 5> run
# End of computation at 1411.00000 ns
# Computed ciphertext is 42b995a03c96c3611bbd350f39ff3a4217e1e21263792197d30faeea29bd67c9c1495620515b13c0ee02b8fad31121
# Expected ciphertext is 42b995a03c96c3611bbd350f39ff3a4217e1e21263792197d30faeea29bd67c9c1495620515b13c0ee02b8fad31121
# Computed tag is f366f456cb2976594eb3452ce34318db
# Expected tag is f366f456cb2976594eb3452ce34318db
```

Fig. 28 : Affichage des résultats obtenus dans Transcript dans ModelSim

9 Conclusion personnelle et difficultés rencontrées

Ce projet était une première expérience pour moi dans le domaine de la cryptographie et de la description matérielle et j'ai trouvé très enrichissant de découvrir cela à travers un projet complet, concret et actuel.

Les principales difficultés que j'ai rencontré proviennent de lacunes dans les fondamentaux nécessaires pour ce projet qui font que j'ai pris beaucoup de temps à comprendre réellement le fonctionnement de l'algorithme et en particulier de la machine d'état. Ce projet a été l'occasion pour moi de combler ces lacunes et bien qu'il m'ait demandé un investissement important en temps, il m'a permis de consolider mes connaissances et de découvrir un nouveau domaine passionnant qui, auparavant, ne m'intéressait pas particulièrement car je n'en comprenais pas du tout le fonctionnement.

Je suis satisfait de ce que j'ai réussi à faire au vu des nombreuses erreurs, principalement de logique mais aussi de programmation, qui m'ont parfois fait douter de ma capacité à obtenir le tag si bien que j'ai abondamment commenté mon code pour garder toutes les informations essentielles à portée de main et être certain de ne pas générer d'erreurs supplémentaires.

Le plus grand enseignement que j'ai tiré de ce projet réside en le fait de faire des schémas et de bien comprendre le fonctionnement de l'algorithme avec de l'implémenter. Malgré ma réticence au départ, les schémas m'ont été extrêmement utiles et même nécessaire pour pouvoir corriger rapidement les erreurs et problèmes de mes modules et de leurs simulations.