Projet ASCON128

Rose Cymbler
Mines de Saint-Étienne

12 mai 2024

Table des matières

1 Introduction

Il est aujourd'hui indispensable de pouvoir sécuriser les données de ses conversations privées. Le système de chiffrement ASCON128 intègre des données associées authentifiées. Il est essentiel pour la sécurité des systèmes embarqués. Il est utilisé pour sécuriser les communications dans des dispositifs tels ceux de l'Internet des Objets (IoT).
L'algorithme de chiffrement ASCON128 est reconnu pour sa capacité à assurer la confidentialité et l'authenticité des données.

Dans le cadre de ce projet en Conception de Sytèmes Numériques, nous utilisons le langage *SystemVerilog* pour implémenter une version simplifiée de l'algorithme de chiffrement ASCON128.

L'objectif est de découvrir le langage SystemVerilog, le logiciel ModelSim et surtout comprendre les principes de cryptographie.

2 Description de l'algorithme ASCON128

2.1 Fonctionnement de l'algorithme ASCON128

L'algorithme agit sur un état courant de 320 bits. Celui-ci va être mis à jour grâce à la répétition d'une opération (appelée permutation) utilisée pendant les quatre phases principales du processus de chiffrement :

- **Initialisation de l'état** : Grâce à un vecteur d'initialisation IV (de 64 bits), à la clé secrète K (de 128 bits), et au nombre (*nonce*) N (de 128 bits), l'état courant est initialisé.
- **Traitement des données associées** : Les données associées A sont sur 48 bits. Grâce à un padding, on les étend sur 64 bits. Ainsi, on opère au traitement de ces données.
- **Traitement en texte clair** : On traite le texte clair P en trois blocs de données de 64 bits. Cela nous permet de récupérer les blocs de texte chiffré.
- **Finalisation** : On récupère le tag qui permet l'authentification du message chiffré.

Cet état courant peut être vu comme un tableau de 5 lignes et 64 colonnes.
Afin de réaliser le traitement de l'état, on utilise alors l'opération appelée *permutation*. En réalité, on utilise des permutations successives par 6 ou par 12.
Une permutation est elle-même composée de trois transformations élémentaires :

- L'addition de constante : on ajoute une constante de ronde au registre x_2 de l'état courant
- La couche de substitution : on applique la substitution de 5 bits en colonne en utilisant une table de substitution
- La couche de diffusion : on applique une diffusion à chaque registre x_i.

Enfin, on implémente le "*ascon_top*" qui contrôle l'exécution des permutations selon les phases de l'algorithme de chiffrement. Ces étapes sont gérées par une machine d'états finis et un compteur de rondes.

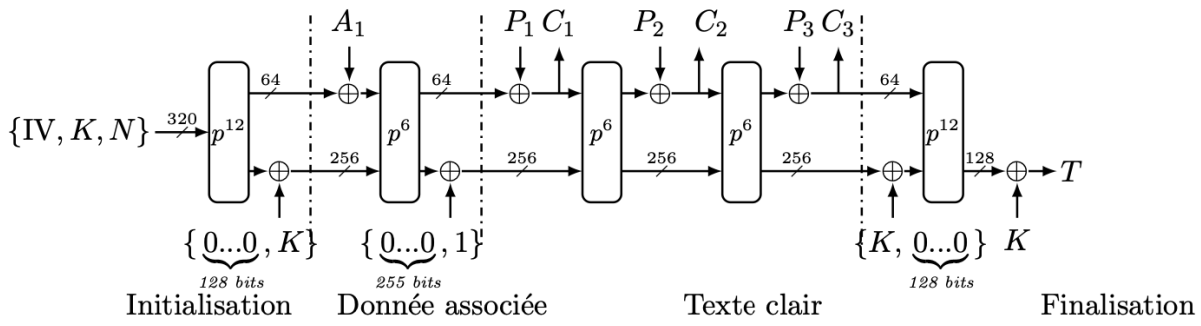

FIGURE 1 – Schéma du chiffrement ASCON128

2.2 Notations utilisées et formalisme

Pour les entrées d'un module, on utilise le suffixe "_i".
Pour les sorties d'un module, on utilise le suffixe "_o".
Pour les signaux, on utilise le suffixe "_s".

2.3 Structure du code, compilation et simulation

On utilise le langage *SystemVerilog*. Pour compiler le projet, voici les commandes à executer :

- bash
- source init_modelsim.txt
- source script.txt

Pour le script de compilation, une seule ligne seulement doit être non commentée pour lancer Modelsim.

3 Modélisation de la permutation

3.1 L'addition de constante p_c

On commence l'algorithme par la première transformation élémentaire de la permutation : l'addition de constante p_c.

Cette transformation a pour effet de modifier le registre x_2 de l'état S. Elle ajoute une constante de ronde c_r qui correspond à la ronde i. Les constantes sont définies dans le tableau suivant :

Pour développer cet algorithme de l'addition de constante, on utilise le package **ascon_pack** (fichier *ascon_pack.sv*). En effet, il contient le tableau des constantes de rondes. Il s'agit du vecteur *round_constant*.

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

FIGURE 2 – Table des constantes de ronde pour p^6 et p^{12}

On fait un XOR (= OU exclusif) entre la constante de ronde et le dernier octet du registre x_2.

On utilise *round_constant* de *ascon_pack.sv* pour effectuer le XOR entre le registre x_2 de *constant_add_i* et la valeur de constante de ronde.

Nous pouvons modéliser ainsi cette transformation :

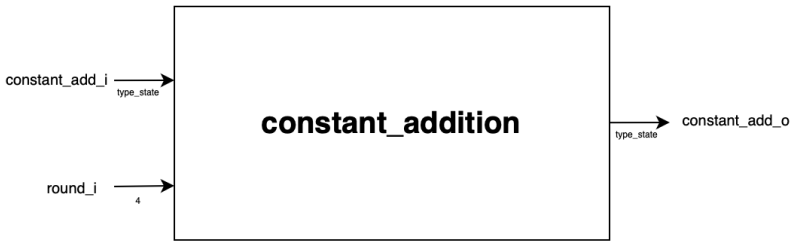

FIGURE 3 – Module *constant_addition*

On réalise un testbench *constant_addition_tb* pour vérifier notre transformation d'addition de constante.

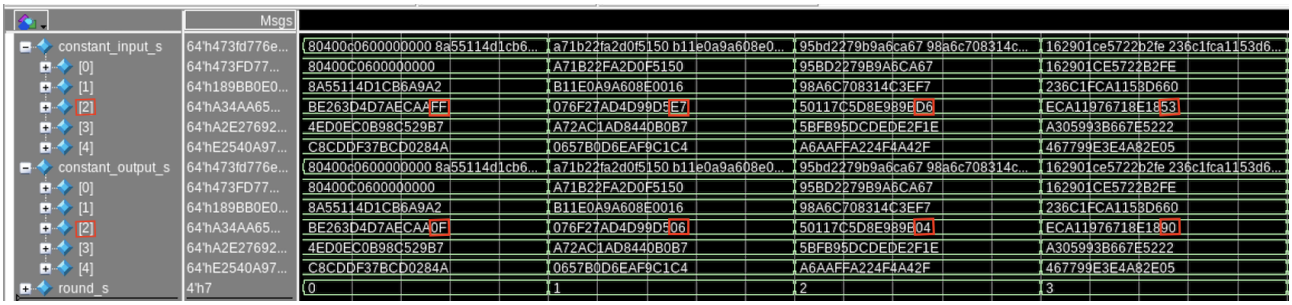

FIGURE 4 – Simulation du testbench du module *constant_addition*

Lorsque l'on s'intéresse aux registres x_2 d'entrée et de sortie, on remarque que seuls les deux derniers octets du registre x_2 ont été modifiés.
L'addition de constante est bien effectuée. On a ici les résultats pour les 4 premières rounds mais la transformation est fonctionnelle pour les rounds de 0 à 11.

3.2 La couche de substitution p_s

Pour réaliser la couche de substitution, on a divisé la substitution p_s en deux modules : le module sbox et le module substitution. La couche de substitution applique la table Sbox aux 5 bits des 64 colonnes de l'état courant.

3.2.1 Le module sbox

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
$S(x)$	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 la SBox

Pour ce module, on a besoin d'utiliser un switch case de 32 cas.
La SBox contient 32 éléments de 5 bits.

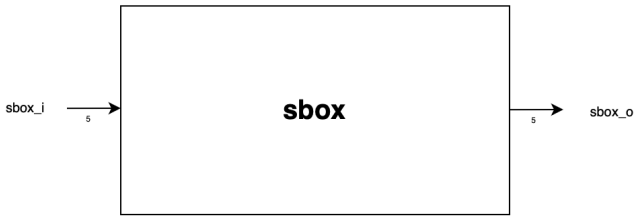

FIGURE 6 – Module SBox

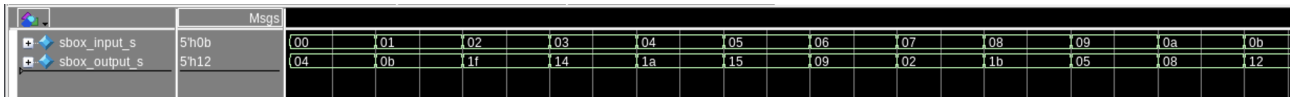

FIGURE 7 – Simulation de la SBox

On remarque bien que les valeurs sont cohérentes avec les résultats de la table de substitution SBox.

3.2.2 Le module de substitution complète

Ce module utilise la sbox afin d'opérer la transformation de substitution.
Pour chacune des 64 colonnes de 5 bits de l'état courant, on applique la SBox.

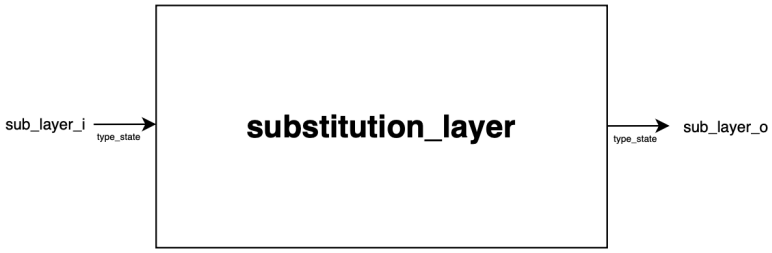

FIGURE 8 – Module *substitution_layer*

On obtient le résultat suivant après avoir réalisé notre testbench.

FIGURE 9 – Testbench de *substitution*

En valeur d'entrée, *sub_layer_input_s*, on a bien utilisé le résultat de l'addition de constante de la ronde 0.
En faisant de même pour les autres rondes, on retrouve bien les résultats attendus pour le module de substitution.

3.3 La couche de diffusion linéaire p_L

La dernière étape de notre permutation est la couche de diffusion linéaire. On applique cette diffusion aux 5 registres qui constituent notre état courant. Les opérations effectuées sont les suivantes.

Par exemple, lorsque l'on a $x_0 >>> 19$, cela signifie que x_0 subit une rotation cyclique de 19 bits vers la droite. Cela s'écrit alors : {diffusion_i[0][18 :0], diffusion_i[0][63 :19]}

On obtient le résultat suivant après avoir réalisé notre testbench.

$$
\begin{aligned}
x_0 &\leftarrow \Sigma_0(x_0) = x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28) \\
x_1 &\leftarrow \Sigma_1(x_1) = x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39) \\
x_2 &\leftarrow \Sigma_2(x_2) = x_2 \oplus (x_2 \ggg 1) \oplus (x_2 \ggg 6) \\
x_3 &\leftarrow \Sigma_3(x_3) = x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17) \\
x_4 &\leftarrow \Sigma_4(x_4) = x_4 \oplus (x_4 \ggg 7) \oplus (x_4 \ggg 41)
\end{aligned}
$$

FIGURE 10 – Opérations pour la couche de diffusion linéaire

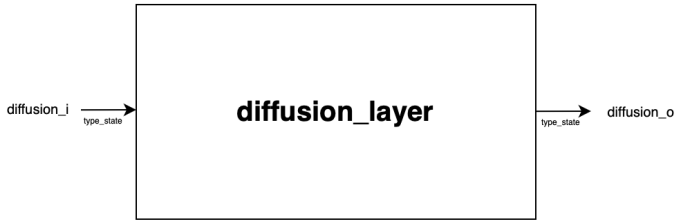

FIGURE 11 – Module de diffusion linéaire

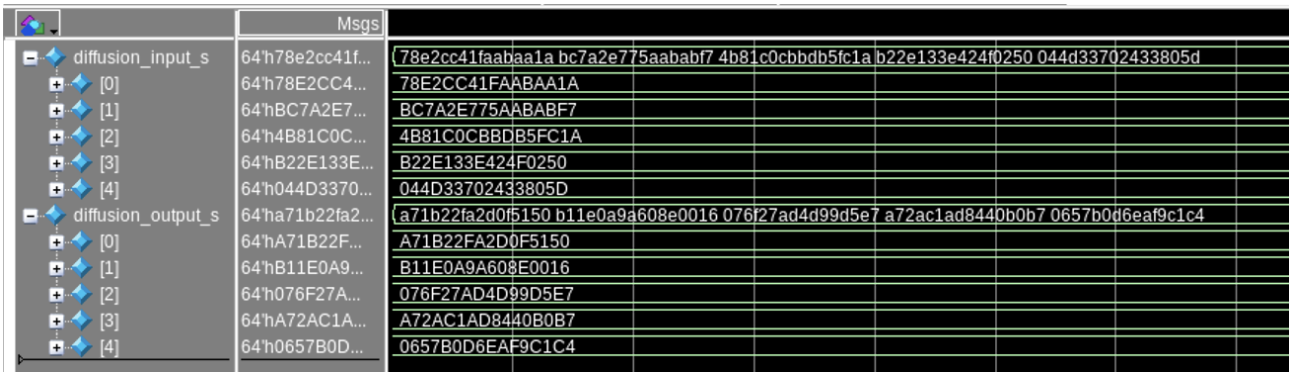

FIGURE 12 – Testbench de la diffusion linéaire

En valeur d'entrée, $diffusion\_input\_s$, on a bien utilisé le résultat de la substitution de la ronde 0.
En faisant de même pour les autres rondes, on retrouve bien les résultats attendus pour le module de diffusion linéaire.

3.4 Le registre *register_state*

On introduit le module $register\_state$. Il a pour but de mémoriser l'état de fin d'une permutation. En effet, l'état subit un ensemble de transformations lors d'une permutation. Cependant, il n'y jamais qu'une permutation, mais toujours 6 ou 12 qui se succèdent.

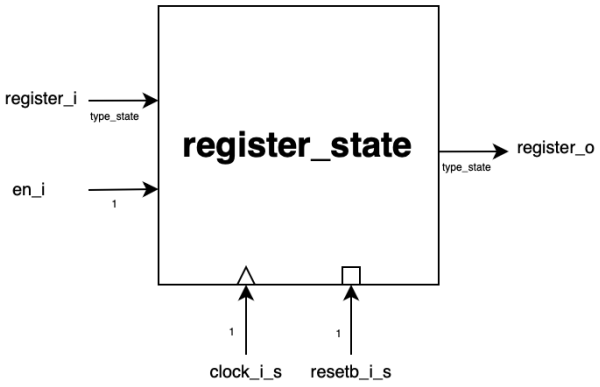

FIGURE 13 – Module *register_state*

3.5 La permutation

L'ensemble des transformations et le registre permettent de créer une permutation. Il ne s'agit pas encore de la permutation complète, que l'on verra dans la partie suivante.

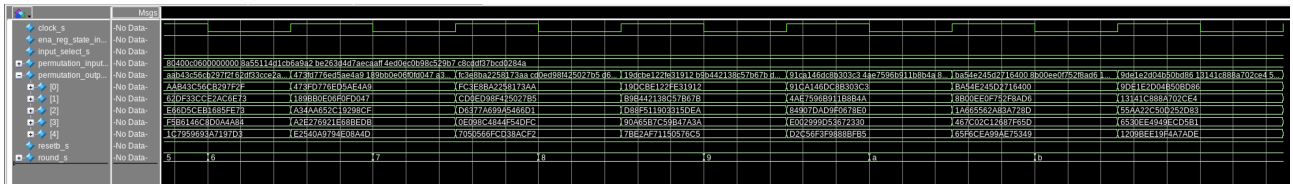

FIGURE 14 – Testbench de la permutation

On voit ici les dernières rondes de la permutation. Nos différentes transformations fonctionnent. Les états à la fin de chaque ronde sont bien mémorisés pour les rondes suivantes.

4 Modélisation de la permutation complète

4.1 Ajout des XOR

On modélise maintenant la permutation complète. Pour cela, on ajoute des XOR. Il en existe de 2 types : les XOR UP et les XOR DOWN.

En bleu, ce sont les XOR UP et en violet, les XOR DOWN.

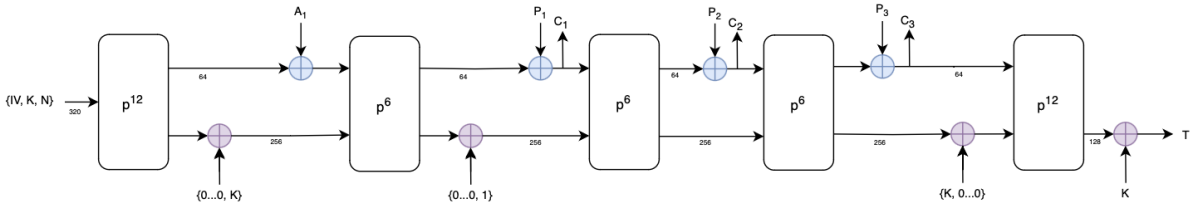

FIGURE 15 – Visualisation des XOR UP et XOR DOWN

4.2 La permutation complète

4.2.1 *permutation_xor*

Dans un premier temps, nous avons implémenté *permutationxor*.
Comme dans le module permutation vu précedemment, on a "instancié" les trois transformations, ainsi que le registre. Cependant, nous avons ajouté les modules XOR UP et DOWN.

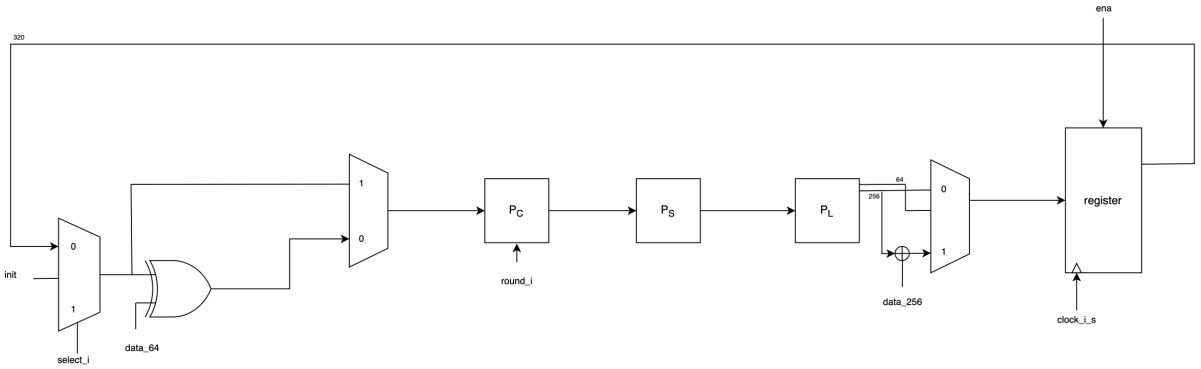

FIGURE 16 – Permutation complète

On réalise notre testbench pour la permutation et on obtient le résultat suivant

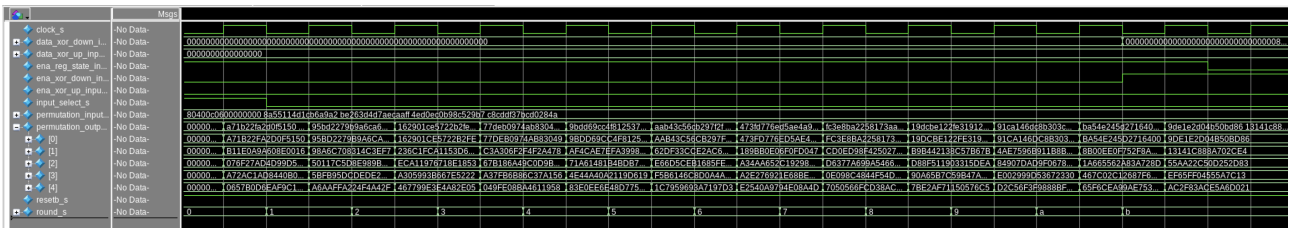

FIGURE 17 – Testbench de la permutation complète

Pour ce testbench, on a pris les valeurs de la phase d'initialisation. On retrouve bien à la fin, les valeurs attendues pour le début de la phase "donnée associée".

4.2.2 *permutation_xor_pour_fsm*

On va devoir utiliser notre permutation_xor dans notre machine d'état. J'ai recréé un module permutation_xor_pour_fsm. Il s'agit du même que permutation_xor mais j'ai ajouté un output "cipher_state_o". C'est un état intermédiaire qui capture le résultat de l'opération XOR.

5 Modélisation du top level (*ascon_top*)

5.1 Le compteur de permutations (*compteur_double_init*)

On implémente un compteur de permutations afin de pouvoir réaliser les blocs de 6 ou 12 permutations.
À chaque front montant d'horloge, ce compteur est actualisé.
Le compteur est initialisé à 0 ou à 6 ce qui permet bien de gérer les blocs de 6 ou 12 permutations.
On incrémente alors dans tous les cas, le compteur jusqu'à 11.

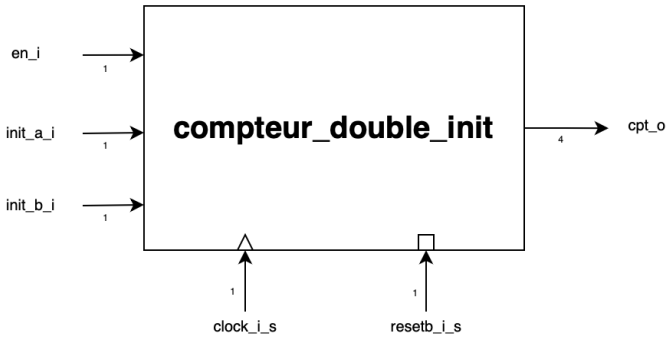

FIGURE 18 – Module compteur_double_init

5.2 La machine d'états finis

On utilise une machine d'états finis de Moore (fsm). Elle contrôle le compteur et les permutations. Elle gère les entrées, les initialise, les active et désactive.

Les entrées sont :

- clock_i,
- resetb_i,
- round_i,
- start_i,
- data_valid_i,

et les sorties

- cipher_valid_o,
- end_o,
- input_select_o,
- ena_xor_up_o,
- ena_xor_down_o,
- ena_reg_state_o,
- conf_xor_down_o,

— init_a_o,
— init_b_o,
— ena_cpt_o,

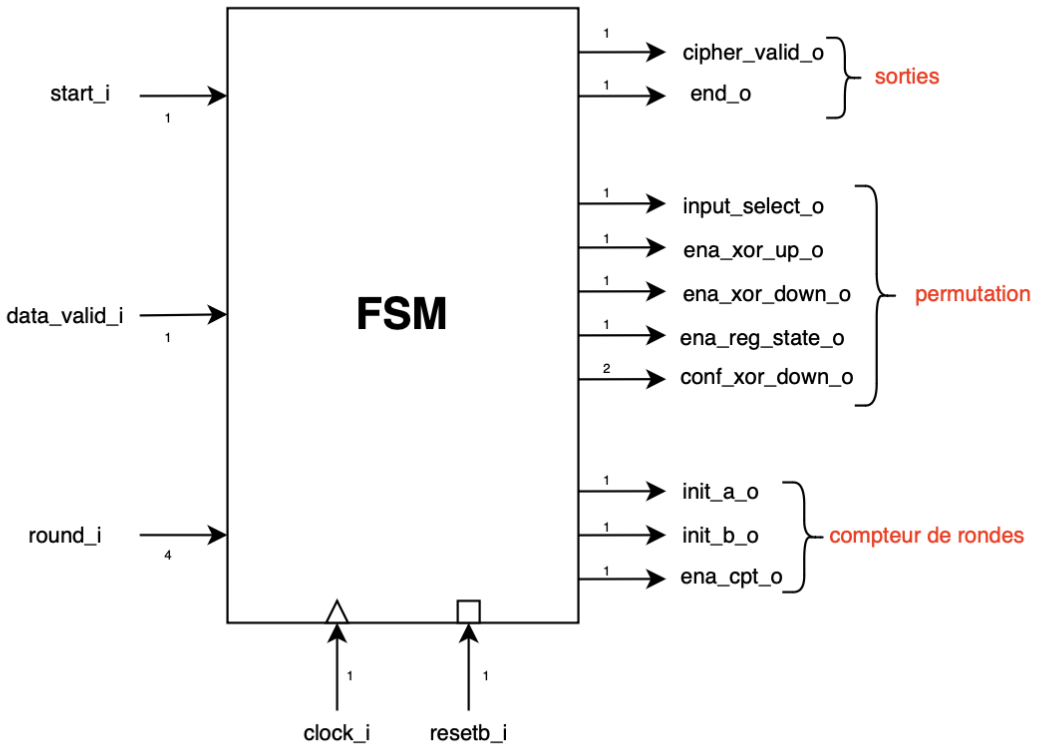

FIGURE 19 – Module FSM

On définit plusieurs états pour reproduire les quatre étapes de l'algorithme.

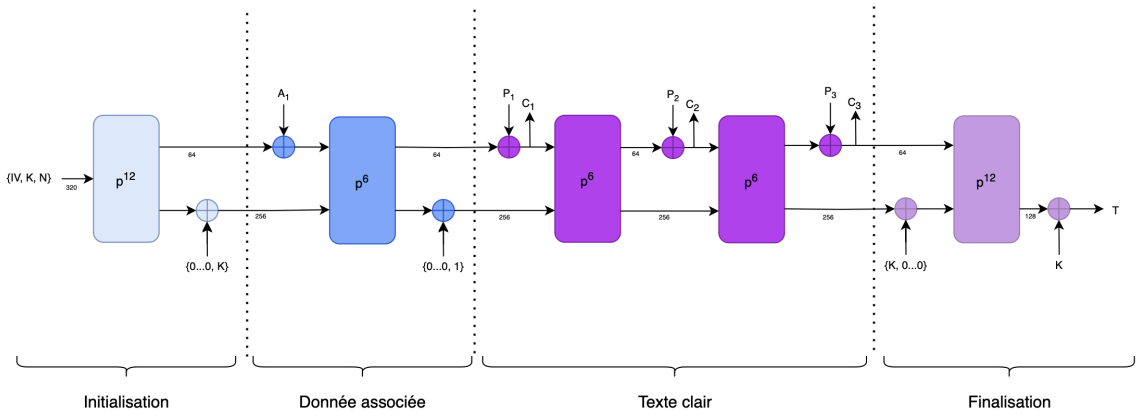

FIGURE 20 – Etapes de l'algorithme ASCON128

On a alors les états suivants

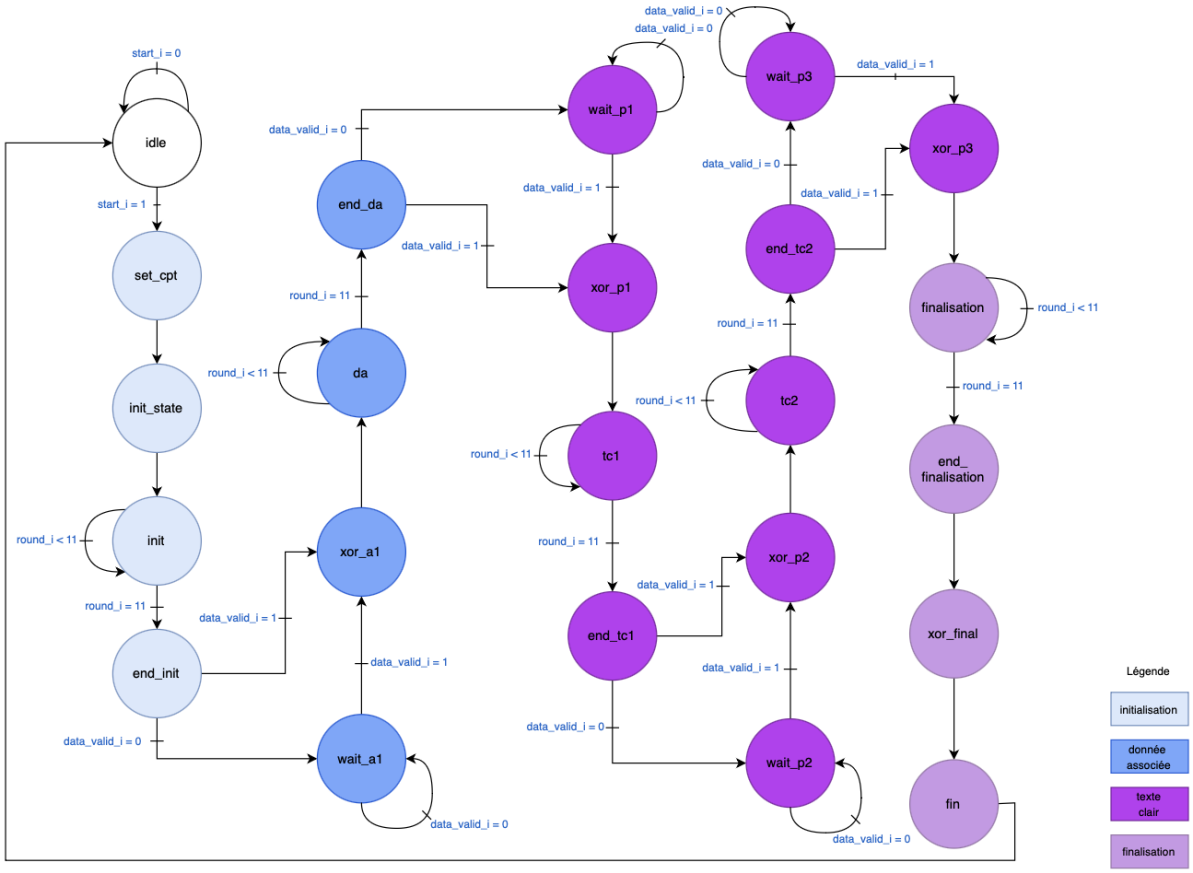

FIGURE 21 – FSM pour ASCON128

5.3 Ascon_top

On implémente l'architecture globale, *ascon_top*. *ascon_top* lie tous les modules : permutation_xor_pour_fsm, compteur_double_init, FSM, register_tag et register_cipher.

Register_tag et register_cipher sont eux modules réalisés sur le modèle de register_state. Ils ont pour but de mémoriser les valeurs de cipher et de tag.

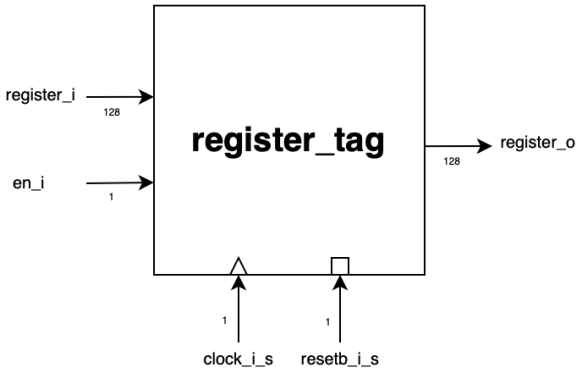

FIGURE 22 – register_tag

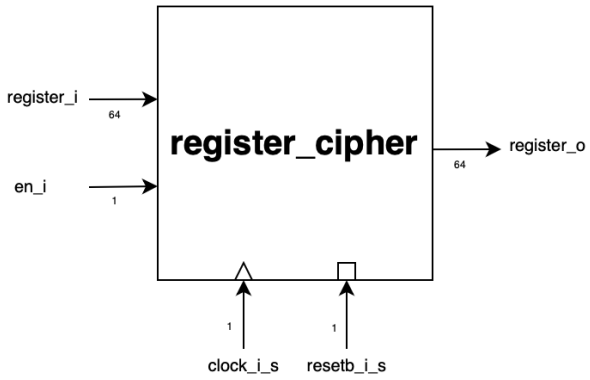

FIGURE 23 – register_cipher

ascon_top est modélisé comme suit :

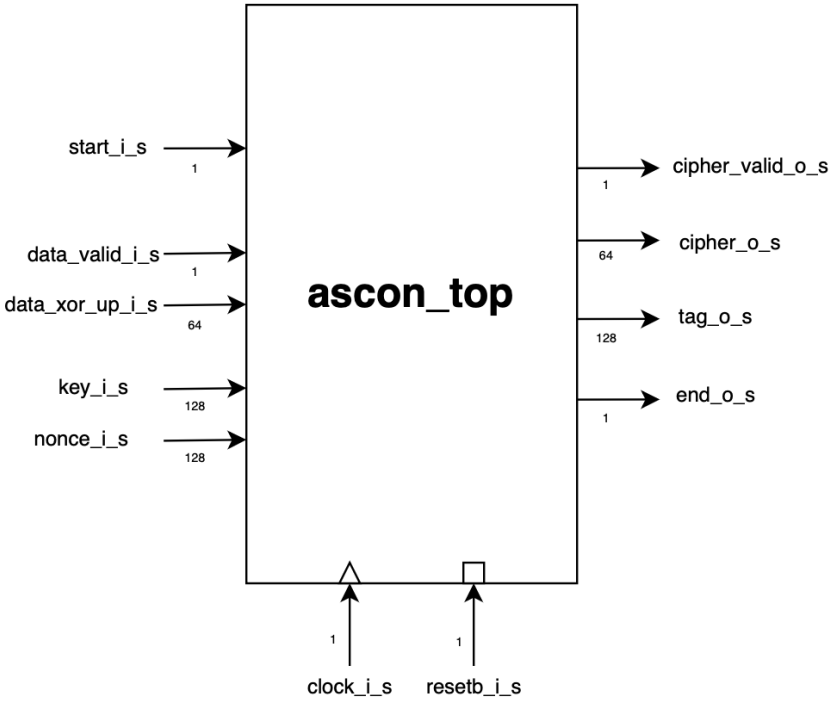

FIGURE 24 – Module ascon_top

6 Simulation et résultats de l'algorithme ASCON128

On réalise le testbench pour tester notre module *ascon_top*.

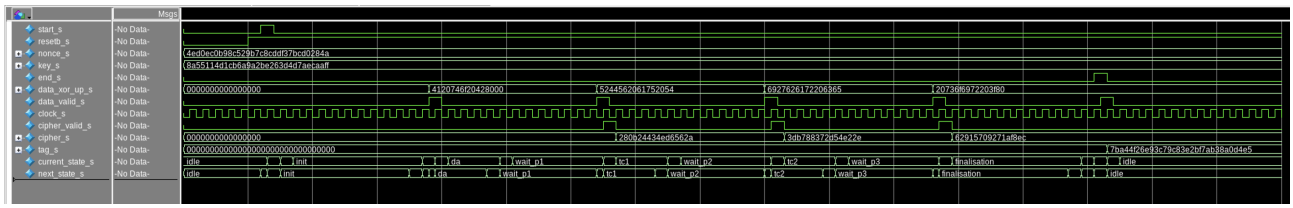

FIGURE 25 – Testbench du module ascon_top

On retrouve le bon tag et le bon cipher. Il y a juste 2 bits en trop pour le cipher. Cela est du au padding que l'on a réalisé précédemment.

7 Difficultés rencontrées et stratégies de résolution

7.1 Défis majeurs lors de la conception et de l'implémentation

Le langage SystemVerilog était nouveau pour moi. J'ai du m'approprier les formalismes et la façon de coder dans ce langage.
Coder dans ce langage demande aussi une maitrise du cours d'électronique numérique du premier semestre et il fallait alors pouvoir dresser les parallèles entre ces deux cours.

J'ai rencontré quelques difficultés pour la machine d'états.
Dans un premier temps, tracer le diagramme et trouver les bons états a été une première épreuve.
Ensuite, trouver les bonnes valeurs à mettre dans le OUTPUT_LOGIC m'a demandé de la réflexion et du temps.

7.2 Pistes non explorées et raisonnement

Je n'ai pas fait apparaitre le compteur de blocs dans mon algorithme par manque de temps.
Pour impélmenter le compteur de blocs, j'aurais du opérer à quelques modifications.

Premièrement, j'aurais du récupérer le fichier *compteur_simple_init*.

Ensuite,il aurait été nécessaire de modifier ma fsm (fichier $fsm.sv$). En effet, le compteur de blocs permet d'alléger le nombre d'états de la fsm. La partie *texte clair* aurait été bien plus courte. Au lieu de faire plusieurs états pour les différentes étapes de de la partie *texte clair*, j'aurais pu faire autant d'états que pour les autres parties (texteclair, end_texteclair, wait_texteclair et xor_texteclair).

Puis, j'aurais pu instancier le compteur de blocs dans mon fichier *ascon_top*.

Il aurait été aussi nécessaire de bien faire attention à ne pas oublier de définir les variables liées au compteur de blocs.

8 Conclusion

Grâce à ce projet, j'ai appris un nouveau langage de programmation $SystemVerilog$. Ce langage m'a lui fait découvrir les algorithmes de chiffrement et de cryptographie.
Je trouve que le fait de pouvoir réaliser un projet entier avec un langage que l'on vient de découvrir est très valorisant et formateur.

Je tiens à remercier M. Reymond pour ses explications claires et son aide précieuse.