



ÉCOLE NATIONALE SUPÉRIEURE DES MINES DE SAINT ÉTIENNE

Modélisation SystemVerilog de l'algorithme de chiffrement ASCON

GP - ÉLECTRONIQUE 1

CONCEPTION DE SYSTÈME NUMÉRIQUE

Elève :

Ibrahim HADJ-ARAB

Enseignant :

Jean-Max DUTERTRE



Table des matières

I	Introduction	1
II	Architecture et Implémentation SystemVerilog des différents modules d'ASCON128	1
1	Architecture Globale d'ASCON128	2
2	Détails des Modules SystemVerilog	3
2.1	Le module de permutation	3
3	Le module <code>mux_state</code>	3
4	Une ronde :	4
4.1	L'additionneur de constante :	4
4.2	La couche de substitution :	5
4.3	La couche de diffusion :	6
4.4	Le module <code>state_register_w_en</code>	7
4.5	La permutation sans XOR	8
5	Implémentation des XOR	9
5.1	Le XOR Begin	9
5.2	Le XOR End	10
5.3	Les Registres pour les cipher et le tag	10
III	Implémentation de la FSM et difficultés rencontrées	12
1	Présentation de la FSM	12
1.1	Description des différentes phases	12
1.2	Optimisation	16
2	Quelques modules importants	17
3	Le module <code>ascon_top.sv</code>	18
4	Difficultés rencontrées	20
IV	Conclusion	21

Table des figures

II.1	Schéma du chiffrement ASCON 128 simplifié	1
II.2	Architecture globale d'ASCON 128 simplifié	2
II.3	Schéma du module de permutation avec XOR	3
II.4	Schéma du multiplexeur	3
II.5	Schéma d'une ronde p	4
II.6	Résultat du TestBench du module <code>constant_addition.sv</code>	5
II.7	Résultat du TestBench du module <code>ascon_sbox.sv</code>	6
II.8	Résultat du TestBench du module <code>substitution_layer.sv</code>	6
II.9	Différence entre (x_0 et $(x_0 > > 28)$)	7
II.10	Résultat du TestBench du module <code>diffusion_layer.sv</code>	7
II.11	Schéma du registre d'états <code>state_register_w_en</code>	7
II.12	Schéma du module <code>permutation_V1.sv</code>	8
II.13	Résultat du TestBench du module <code>permutation_V1.sv</code>	9
II.14	Schéma du <code>XOR_begin</code>	9
II.15	Schéma du <code>XOR_end</code>	10

II.16 Schéma des registres pour le cipher et le tag	10
III.1 Schéma du Phase d'Initialisation de la FSM	12
III.2 Schéma du Phase de Donnée Associées de la FSM	13
III.3 Schéma du Phase de Texte Clair de la FSM	14
III.4 Schéma du Phase de Finalisation de la FSM	15
III.5 Schéma du Schéma de la FSM totale	15
III.6 Schéma du Schéma de la FSM optimisée	16
III.7 Schéma du Schéma du compteur de ronde	17
III.8 Schéma du Schéma du compteur de ronde	17
III.9 Schéma du ascon_top	18
III.10 Résultats du ascon_top_tb.sv	19
III.11 Résultats du permutation_xor_tb.sv	20

Liste des tableaux

1 Constante à XOR selon la ronde en cours	4
2 S-box utilisée dans le projet	5

I Introduction

Ce rapport présente l'implémentation d'une version simplifiée de l'algorithme ASCON128 en SystemVerilog. Ce dernier (HDL) est un langage de description matérielle utilisé dans la conception et la vérification de circuits numériques. L'objectif ici est de fournir une description détaillée de l'architecture et du fonctionnement de l'ASCON128 ainsi que des résultats de simulation et de validation. Ce rapport abordera également les défis rencontrés lors du développement et les solutions apportées pour les surmonter.

Dans la section suivante, nous décrirons l'architecture globale de l'ASCON128 et son implémentation en SystemVerilog à travers différents modules. Nous présenterons ensuite la méthodologie de test et les résultats de simulation obtenus, ainsi que les problèmes rencontrés et les solutions apportées. Enfin, nous conclurons en discutant des performances et des problèmes rencontrés lors de l'implémentation de cet algorithme de chiffrement.

II Architecture et Implémentation SystemVerilog des différents modules d'ASCON128

ASCON 128 est un algorithme de chiffrement authentifié avec données associées (AEAD). En effet, il est basé sur une fonction éponge qui est une méthode de construction cryptographique permettant de créer notamment des algorithmes de chiffrement. Ici, l'architecture globale d'ASCON 128 se compose de plusieurs modules interconnectés qui réalisent ensemble les opérations de chiffrement et d'authentification. Dans le cas de notre projet, l'algorithme est simplifié afin qu'il soit réalisable dans le temps qui nous est accordé pour le modéliser.

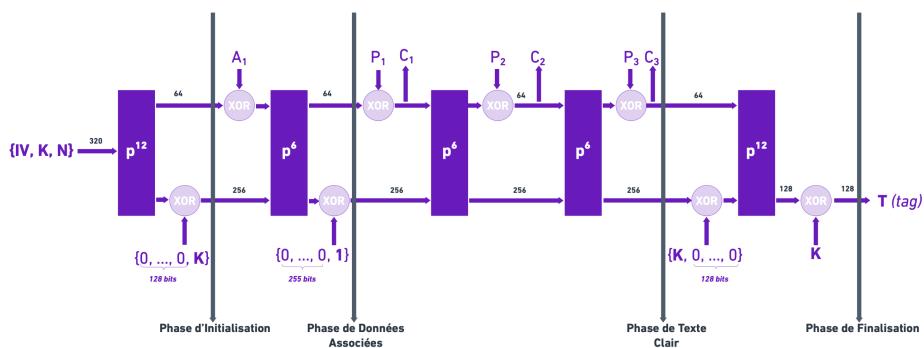


FIGURE II.1 – Schéma du chiffrement ASCON 128 simplifié

Ainsi, ASCON 128 fonctionne comme suit : Après avoir reçu en entrée un vecteur d'initialisation (IV) de 128 bits, la clé de chiffrement (K) de 128 bits, ainsi qu'un nombre aléatoire choisi arbitrairement dans notre projet (également de 128 bits appelé N ou "nonce"), plusieurs étapes de chiffrement se font. En effet, les 320 bits sont décomposés en deux blocs de 256 et 64 bits et parcourent un "chemin" différent.

Enfin, l'algorithme retourne à la fin le texte chiffré (C ou "cipher"), décomposé en deux blocs de 64 et un bloc de 56 bits, ainsi qu'un tag (T) de 128 bits. Le tag permet lui d'assurer l'authentification lors de la réception du message chiffré.

De plus, ASCON 128 récupère également lors des phases de **Données Associées** et de **Texte Clair** les données associées (A) de 48 bits et le message à crypté (P) décomposés tout comme C en deux blocs de 64 et un bloc de 56 bits. Plusieurs opérations de type XOR ainsi que des permutations expliquées plus tard permettent de chiffrer le message.

1 Architecture Globale d'ASCON128

L'algorithme récupère donc en entrée IV, K et N en cinq registres de 64 bits distincts respectivement appelés x_0 , x_1 , x_2 , x_3 et x_4 . Ainsi, ASCON 128 est composé de :

- d'un module de permutation décrit ci-après
- d'un **compteur double** permettant de compter le nombre de rondes faites (la ronde sera également définie plus tard).
- d'un **compteur simple** permettant de compter cette fois-ci le nombre de blocs.
- d'une machine d'états finis (FSM) qui s'assure que toutes les opérations sont faites au bon moment lors de l'exécution de l'algorithme.
- un module appelé **ascon_top** qui joue le rôle d'interface principale pour le chiffrement de notre message.

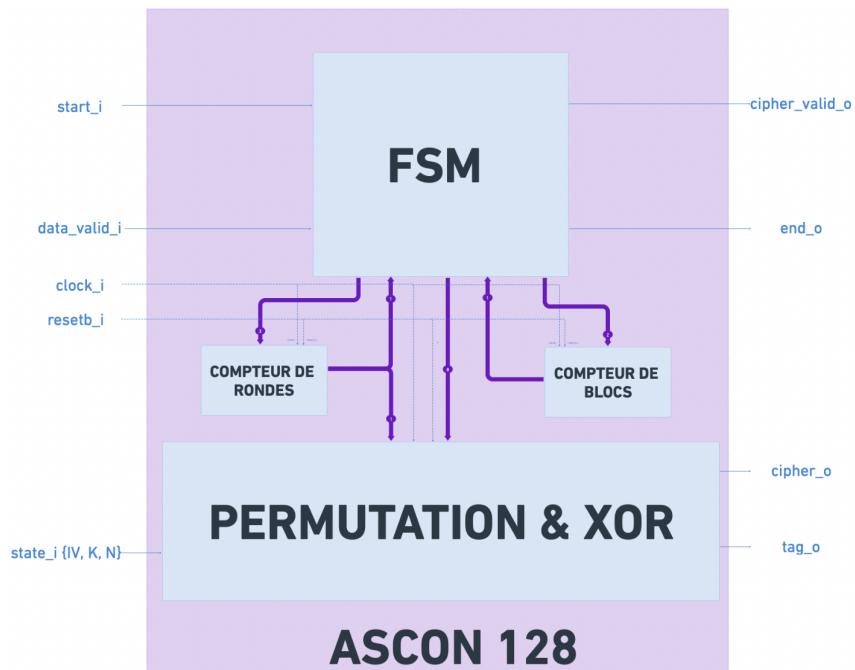


FIGURE II.2 – Architecture globale d'ASCON 128 simplifié

2 Détails des Modules SystemVerilog

2.1 Le module de permutation

Une permutation peut être définie comme une transformation réversible appliquée aux x_i pour i allant de 0 à 4. En effet, cette permutation consiste en un réarrangement des éléments selon un schéma prédéfini, en utilisant les opérations suivantes :

- l'addition de constante
- la substitution
- la diffusion
- ainsi que le XOR des données associées, du texte clair ainsi que de la clef avec le signal

Son rôle est donc de renforcer la résistance de l'algorithme en cas d'attaques cryptanalytiques.

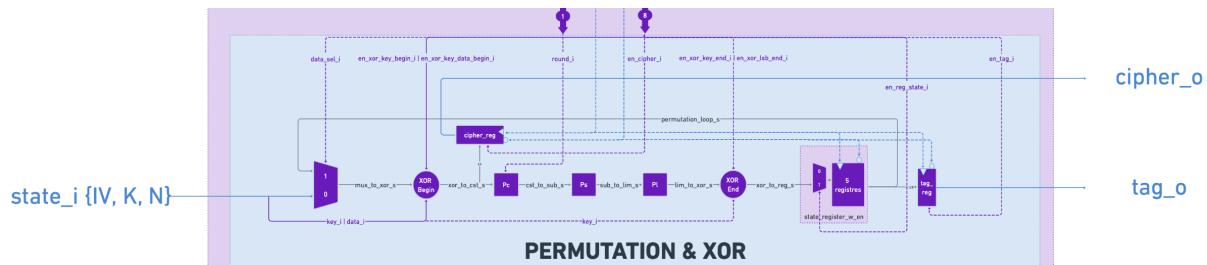


FIGURE II.3 – Schéma du module de permutation avec XOR

3 Le module mux_state

Le module `mux_state` est tout simplement un multiplexeur, c'est-à-dire qu'il permet de sélectionner une entrée parmi plusieurs et la transmet vers une sortie unique. Ici, `mux_state` choisit soit l'entrée composée de IV, K et N soit le signal en sortie de la permutation appelé `state_s` ou `permutation_loop_s` selon la valeur de `data_sel_i`.

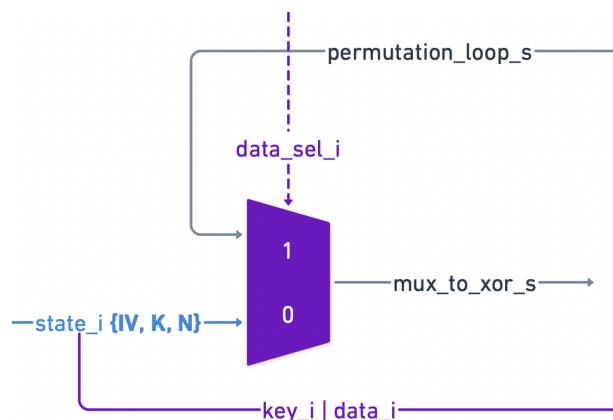


FIGURE II.4 – Schéma du muxtiplexeur

4 Une ronde :

Une ronde p est une étape de transformation répétitive appliquée à l'état interne. Chaque ronde consiste en une série d'opérations élémentaires qui sont la substitution, l'addition de constante et la diffusion. En effet, elles sont appliquées de manière itérative pour assurer la confusion et la diffusion des données au sein de l'état interne. Le nombre de rondes est ici soit de 6 soit de 12 selon à quel état nous sommes dans l'algorithme :

$$p = p_L \circ p_S \circ p_C \quad (1)$$

avec p_L la couche de diffusion, p_S celle de substitution et p_C l'additionneur de constante.

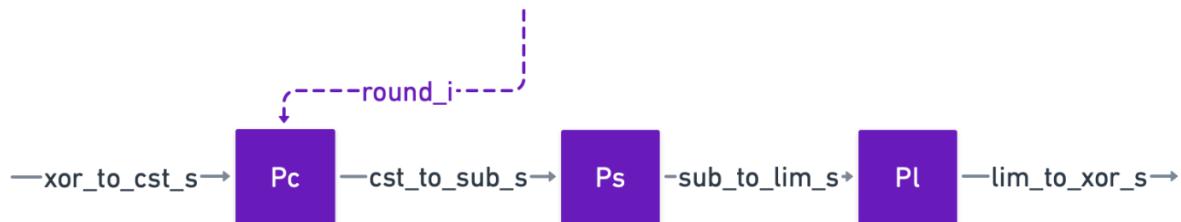


FIGURE II.5 – Schéma d'une ronde p

4.1 L'additionneur de constante :

L'additionneur de constante p_C est un module qui reçoit en entrée le signal `state_s` XORé ou non et renvoie en sortie le même signal mis à part les deux derniers bits du registre x_2 modifiée selon l'équation :

$$x_2 = x_2 \oplus c_r \quad (2)$$

où c_r dépend de la ronde r où l'algorithme se situe :

Ronde r de p ¹²	Ronde r de p ⁶	Constante c _r
0		00000000000000000000f0
1		00000000000000000000e1
2		00000000000000000000d2
3		00000000000000000000c3
4		00000000000000000000b4
5		00000000000000000000a5
6	0	0000000000000000000096
7	1	0000000000000000000087
8	2	0000000000000000000078
9	3	0000000000000000000069
10	4	000000000000000000005a
11	5	000000000000000000004b

TABLE 1 – Constante à XOR selon la ronde en cours

On définit donc c_r dans le module `ascon_pack.sv` :

```

1 // Round constant for constant addition
2 localparam logic [7:0]      round_constant [0:11] = {8'hF0, 8'hE1, 8'
3   hD2, 8'hC3, 8'hB4, 8'hA5, 8'h96, 8'h87, 8'h78, 8'h69, 8'h5A, 8'h4B};

```

Listing 1 – définition de la constante de ronde dans `ascon_pack.sv`

On obtient donc pour la première ronde r_0 :

```

1 pc(80400c0600000000 0001020304050607 08090a0b0c0d0eff 0011223344556677
2   8899aabbcdddeeff) =
3   80400c0600000000 0001020304050607 08090a0b0c0d0eff 0011223344556677
   8899aabbcdddeeff

```



FIGURE II.6 – Résultat du TestBench du module `constant_addition.sv`

4.2 La couche de substitution :

La couche de substitution p_s permet quant à elle de transformer de manière non-linéaire des données au sein de l'état interne de l'algorithme. En effet, p_s fonctionne par le biais d'une boîte de substitution ou **S-box** qui est une fonction de substitution non-linéaire utilisée pour transformer des blocs de données d'une taille fixe. Ainsi, la **S-box** est définie comme une table de recherche qui associe chaque entrée possible à une sortie unique :

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

TABLE 2 – S-box utilisée dans le projet

Ici, elle est implémentée directement dans le module `ascon_pack.sv` :

```

1 // ASCON_Sbox (array)
2 localparam logic [4:0] sbox_t [0:31] = {5'h04, 5'h0B, 5'h1F, 5'h14,
3   5'h1A, 5'h15, 5'h09, 5'h02, 5'h1B, 5'h05, 5'h08, 5'h12, 5'h1D, 5'h03,
4   5'h06, 5'h1C, 5'h1E, 5'h13, 5'h07, 5'h0E, 5'h00, 5'h0D, 5'h11, 5'h18
5   , 5'h10, 5'h0C, 5'h01, 5'h19, 5'h16, 5'h0A, 5'h0F, 5'h17};

```

Listing 2 – définition de la S-box dans `ascon_pack.sv`

On obtient donc depuis ModelSim :

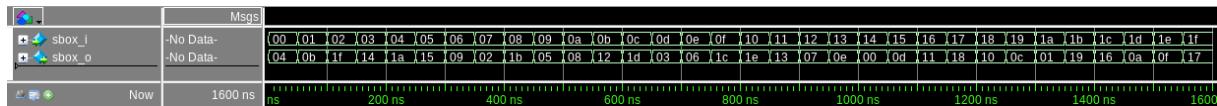


FIGURE II.7 – Résultat du TestBench du module `ascon_sbox.sv`

Ainsi, le rôle de la couche de substitution est de briser la linéarité des opérations de transformation et d'introduire une non-linéarité dans l'algorithme pour protéger l'algorithme. P_S est donc appliquée à chaque élément de l'état interne de l'algorithme, en utilisant la S-box pour transformer chaque bloc x_i où i va de 0 à 4.

Ce bloc a est également simulé depuis ModelSim en utilisant la sortie de `texttpc` entrée :



FIGURE II.8 – Résultat du TestBench du module `substitution_layer.sv`

Ce qui donne donc :

```

1 ps(80400c0600000000 8a55114d1cb6a9a2 be263d4d7aecaa0f
2   4ed0ec0b98c529b7 c8cdff37bcd0284a) =
3   78e2cc41faabaa1a bc7a2e775aababf7 4b81c0cbbdb5fc1a b22e133e424f0250
4   044d33702433805d

```

4.3 La couche de diffusion :

Enfin, la couche de diffusion p_L assure tout comme p_S la propagation des effets des opérations de transformation non-linéaires à travers l'ensemble de l'état interne de l'algorithme. p_L est donc une transformation réalisée à l'aide d'une fonction de diffusion linéaire. Cette dernière est une transformation linéaire définie comme suit :

$$\begin{aligned}
 p_L(x_0) &= x_0 \oplus (x_0 >>> 19) \oplus (x_0 >>> 28) \\
 p_L(x_1) &= x_1 \oplus (x_1 >>> 61) \oplus (x_1 >>> 39) \\
 p_L(x_2) &= x_2 \oplus (x_2 >>> 1) \oplus (x_2 >>> 6) \\
 p_L(x_3) &= x_3 \oplus (x_3 >>> 10) \oplus (x_3 >>> 17) \\
 p_L(x_4) &= x_4 \oplus (x_4 >>> 7) \oplus (x_4 >>> 41)
 \end{aligned} \tag{3}$$

Avec $(x_0 >>> 28)$ définit comme l'opération de rotation suivante :

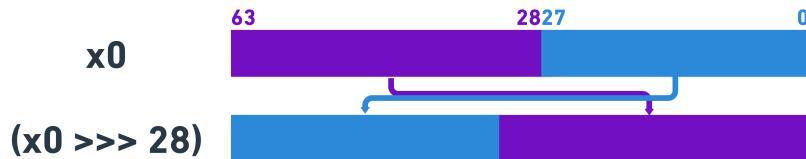


FIGURE II.9 – Différence entre $(x_0$ et $(x_0 >>> 28))$

On obtient donc en sortie de $(p_L$ depuis ModelSim :

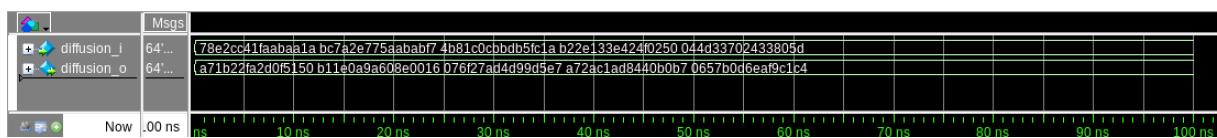


FIGURE II.10 – Résultat du TestBench du module diffusion_layer.sv

```

1 p_L(78e2cc41faabaa1a bc7a2e775aababf7 4b81c0cbbdb5fc1a b22e133e424f0250
     044d33702433805d) =
2      a71b22fa2d0f5150 b11e0a9a608e0016 076f27ad4d99d5e7
3      a72ac1ad8440b0b7
4      0657b0d6eaf9c1c4

```

4.4 Le module state_register_w_en

Après avoir effectué une ronde, le signal est écrit depuis le module `state_register_w_en`. Ce dernier est un registre d'état simple dont les entrées sont :

- un signal de réinitialisation `en_i` (dans notre cas c'est `en_reg_state_i`)
- un signal d'horloge `clock_i`
- un signal de réinitialisation `resetb_i`
- et le signal d'entrée `data_i` (dans le module permutation c'est `xor_to_reg_s`)

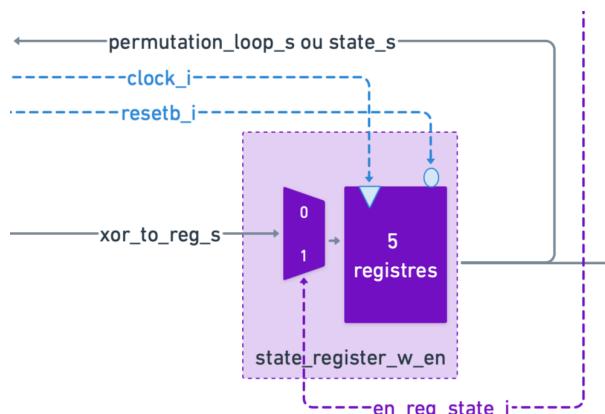


FIGURE II.11 – Schéma du registre d'états `state_register_w_en`

Ainsi, le registre d'état dont le signal interne `data_s` est mis à jour à chaque front montant de l'horloge `clock_i` ou à chaque front descendant de `resetb_i`. Si `resetb_i` est inactif (à l'état bas), le registre d'état est alors réinitialisé à une valeur nulle. Sinon, si le signal `en_i` est actif (à l'état haut), le registre d'état est mis à jour avec les données d'entrée `data_i`. Enfin, si `en_i` est inactif, le registre d'état conserve sa valeur précédente.

Pour finir, le registre retourne en sortie `data_o` dont la valeur est assignée à la valeur de `data_s`.

4.5 La permutation sans XOR

Ainsi, une permutation simple dont le module est `permutation_V1.sv` a été faite sans les blocs `XOR_begin` et `XOR_end`.

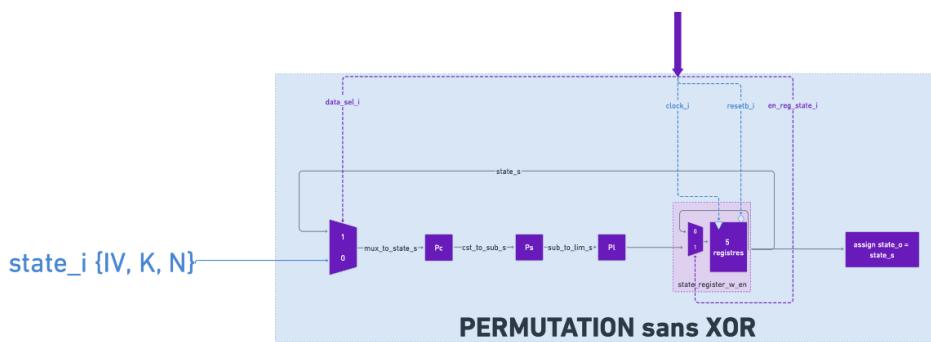


FIGURE II.12 – Schéma du module `permutation_V1.sv`

En plus des trois blocs `pC`, `ps` et `pL`, un multiplexeur a été ajouté avant l'additionneur de constante pour déterminer la nature de l'entrée. En effet, selon `data_sel_i`, l'entrée est soit `state_i`, soit la sortie du bloc rebouclée `state_s`. Cela permet donc de faire 6 ou douze permutations selon l'endroit où l'on se situe dans l'algorithme.

Par la suite, les permutations avec les blocs `XOR_begin` et `XOR_end` ont été introduites. Pour prendre en compte les opérations précédent et suivant les permutations, le module `XOR_begin` a été placé entre le multiplexeur et `pC`. De la même manière, le module `XOR_end` a été placé entre `pL` et `state_register_w_en`.

Ainsi, les signaux `en_xor_data_begin_i`, `en_xor_key_begin_i`, `en_xor_lsb_begin_i` et `en_xor_key_end_i` permettent d'activer ou désactiver ces modules si besoin.

On obtient donc :

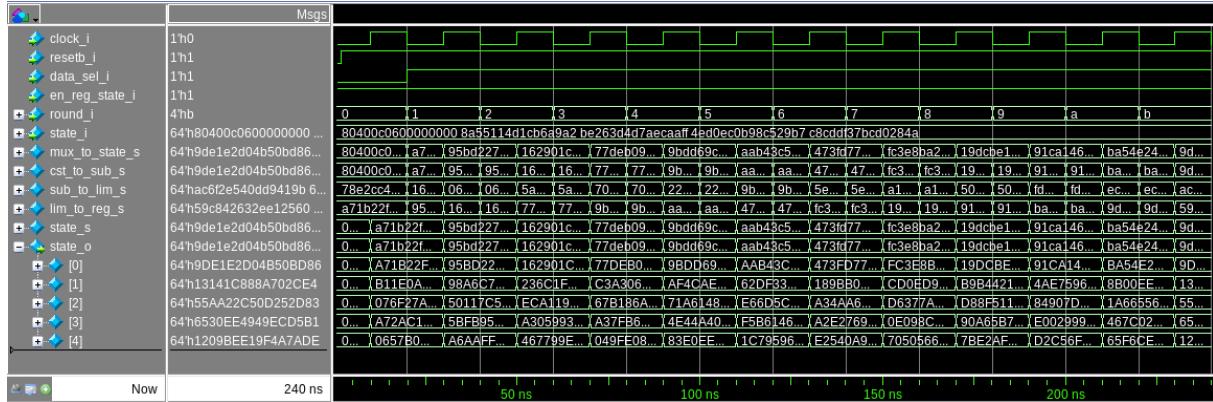


FIGURE II.13 – Résultat du TestBench du module permutation_V1.sv

```

1 p12(80400c0600000000 8a55114d1cb6a9a2 be263d4d7aecaff 4ed0ec0b98c529b7
      c8cddf37bcd0284a) =
2 9de1e2d04b50bd86 13141c888a702ce4 55aa22c50d252d83 6530ee4949ecd5b1
3 1209bee19f4a7ade

```

Ainsi tous les modules fonctionnent car les résultats sont cohérents avec ceux données par le sujet.

5 Implémentation des XOR

5.1 Le XOR Begin

Lors du début de la phase de Données Associées, de la phase de Texte Clair et du début de la phase de Finalisation, le signal est XOR avec soit une donnée soit la clef K.

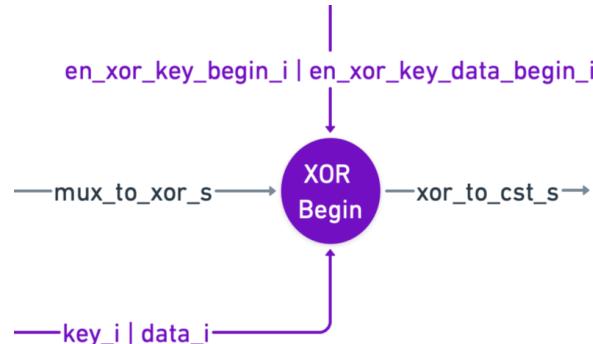


FIGURE II.14 – Schéma du XOR_begin

Ainsi, ce module est en fait la composition de deux **XORs** ainsi que deux **multiplexeurs** qui effectuent un XOR sur différents registres. En effet, si **en_xor_data_begin_i** est activé, les 64 premiers bits (x_0) sont XOR avec les données associées ou le texte clair (i.e. le message envoyé). De la même manière, si **en_xor_key_begin_i** est activé, les 256 derniers bits (x_1, x_2, x_3 et x_4) sont XOR avec K. On obtient donc en sortie l'état interne mis à jour **state_o** (ici c'est **xor_to_cst_s**).

5.2 Le XOR End

Tout comme le module `XOR_begin`, le `XOR_end` permet au signal d'entrée d'être XOR en fin de permutation. En effet, lors de la fin de la phase d'Initialisation, de Données Associées et de Finalisation le signal est XOR avec soit la clef K (ce sont donc x_3 et x_4 qui sont XOR) soit avec 255 bits nul et un bit égal à 1 (LSB ou Less Significant Bit).

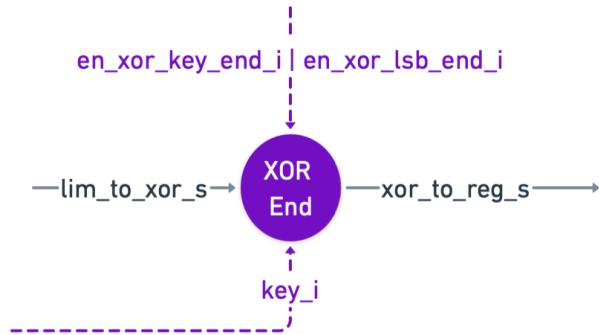


FIGURE II.15 – Schéma du `XOR_end`

Ainsi il faut donc aussi à ce XOR deux signaux d'activation qui sont respectivement `en_xor_key_end_i` et `en_xor_lsb_end_i`.

5.3 Les Registres pour les cipher et le tag

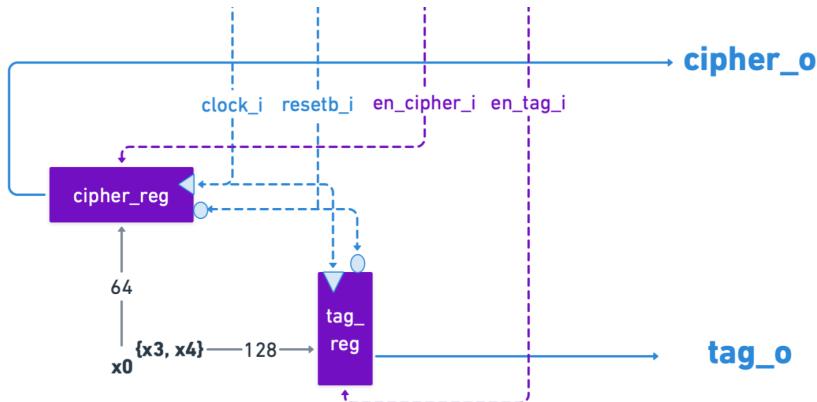


FIGURE II.16 – Schéma des registres pour le cipher et le tag

Afin de pouvoir enregistrer le message chiffré ainsi que le tag, deux registres sont nécessaires. Ils fonctionnent de la même manière que le registre `state_register_w_en` à l'exception que l'on peut choisir le nombre de bit que l'on enregistre.

```

1  `timescale 1 ns/ 1 ps
2
3 module register_w_en import ascon_pack::*;
4   #(
5     parameter nb_bits_g=32
6   )
7   (
8     input logic clock_i,
9     input logic resetb_i,
10    input logic en_i,
11    input logic [nb_bits_g-1 : 0] data_i,
12    output logic [nb_bits_g-1 : 0] data_o
13  ) ;

```

Listing 3 – Définition des Entrées/Sorties dans le module `register_w_en.sv`

Ainsi on peut constater dans le module `register_w_en.sv` que le nombre de bits enregistrés est mis en paramètre et initialisé à 0. Dans notre cas nous utiliserons ce module pour modéliser les registres pour le `cipher` et le `tag` en mettant en paramètre 64 et 128 bits.

III Implémentation de la FSM et difficultés rencontrées

Afin de pouvoir modéliser l'algorithme, il faut modéliser une FSM (Machine d'États Finis). En effet, elle permet de gérer les différentes étapes de l'algorithme qui sont (dans l'ordre) la phase d'Initialisation, la phase de Données Associées, la phase de Texte Clair et la phase de Finalisation.

1 Présentation de la FSM

1.1 Description des différentes phases

La phase d'Initialisation : La phase d'Initialisation consiste à comme son nom l'indique à initialiser l'état interne de l'algorithme en intruisant en entrée IV, K et N. On applique donc ensuite 12 permutations et se termine par un XOR_end avec la clef sur les 128 derniers bits.

C'est pourquoi on peut la décomposer en 5 états :

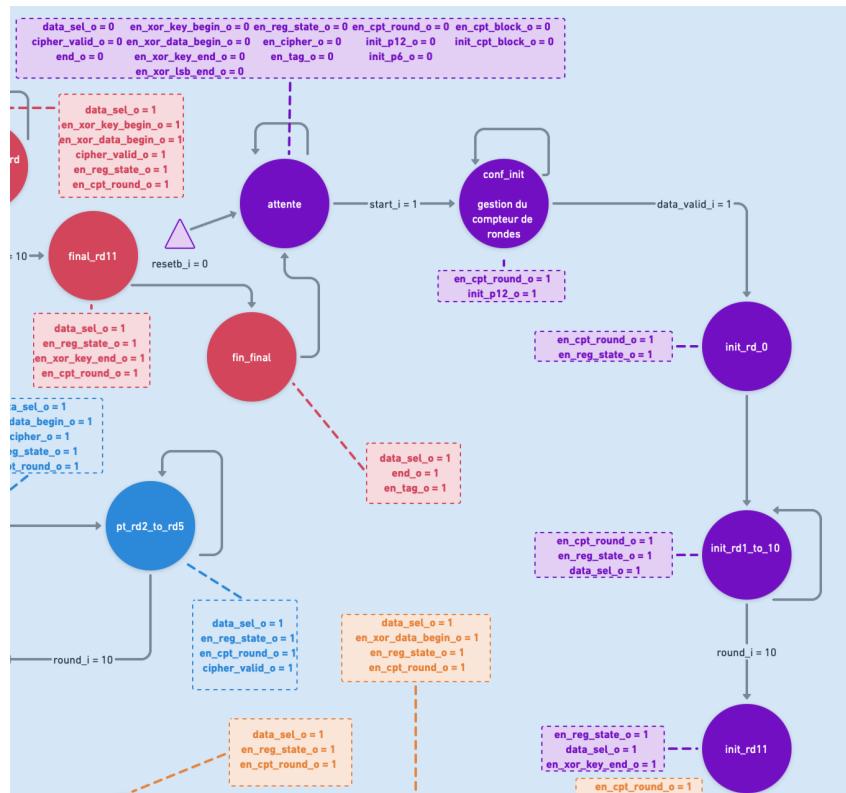


FIGURE III.1 – Schéma du Phase d'Initialisation de la FSM

- l'état `attente` remet toutes les sorties à zéro.
- l'état `conf_init` permet de réinitialiser le compteur de ronde pour 12 permutations.
- `init_rd0` est dissocié des états suivant car il faut faire passer le selectionneur du multiplexeur à 1 afin de faire reboucler le signal.
- `init_rd1_to_rd10` fait 10 rondes.

- `init_rd11` est aussi séparé des autres rondes car il doit faire un `XOR_end` comme dit précédemment.

La phase de Données Associées : Cette phase permet de traiter les données associées qui ne sont pas chiffrées mais authentifiées. Les données associées sont absorbées dans l'état interne en appliquant un `XOR_begin` aux 64 premiers bits et en faisant 6 permutations successives. Enfin, la phase se conclut par un `XOR_end` pour le dernier bit avec un 1 (Less Significant Bit).

Cette phase est donc composée de 4 états :

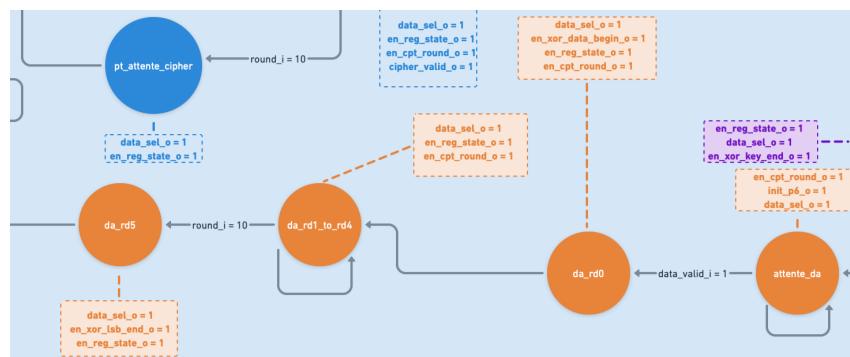


FIGURE III.2 – Schéma du Phase de Donnée Associées de la FSM

- `attente_da` comme `conf_init` remet à zéro le compteur de rondes pour 6 permutations cette fois.
- l'état `conf_init` permet de réinitialiser le compteur de ronde pour 12 permutations.
- `da_rd0`, lui, fait un `XOR_begin` avec la donnée associée A.
- `da_rd1_to_rd4` fait quant à lui 4 rondes.
- `da_rd5` est aussi un état séparé de la boucle `da_rd1_to_rd4` car il doit faire un `XOR_end` comme dit précédemment.

La phase de Texte Clair : Elle permet à chiffrer le texte clair en texte chiffré. Le texte clair est donc divisé deux blocs de 128 bits (P_1 , P_2) et un bloc de j bits (P_3). Ces derniers sont traités en appliquant un XOR_begin aux 64 premiers bits. Entre les deux premiers blocs de texte chiffrés retournés (C_1 et C_2), 6 permutations sont faites.

Ainsi la phase de **Texte Clair** est scindée elle en 6 états muni d'une boucle :

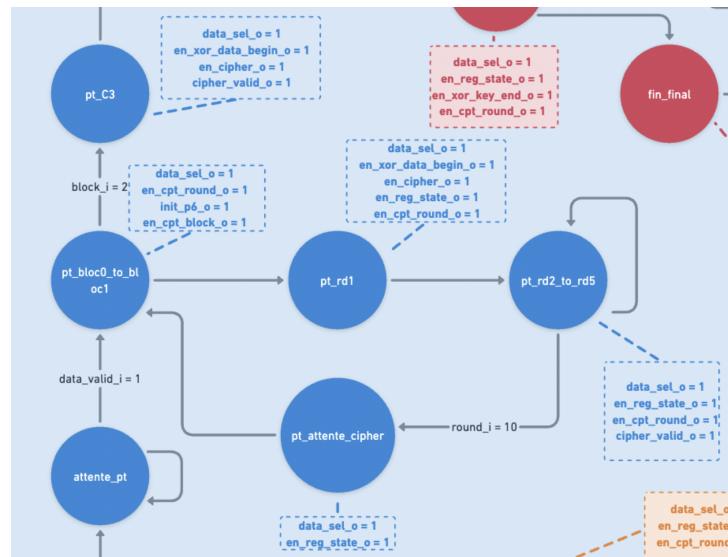


FIGURE III.3 – Schéma du Phase de Texte Clair de la FSM

- **attente_pt** permet d'attendre que la valeur du premier texte clair P_1 est bien reçue
- **pt_bloc0_to_bloc1** est l'état le plus important de cette phase. En effet, il est le lien entre le début de la "boucle" de texte clair puisque chaque bloc P_i (où i varie entre 1 et 3) subit 6 permutations. Ainsi l'algorithme sort de cette boucle après l'avoir parcourue deux fois.
- **pt_rd1** est aussi séparée de l'état suivant car il doit y avoir un XOR_begin à la première ronde.
- **pt_rd2_to_rd4** fait quant à lui 4 rondes.
- **pt_attente_cipher** est lui un état permettant d'attendre un coup d'horloge et donc de mettre à jour la nouvelle valeur du signal.
- **pt_C3** est pour finir un état qui écrit la valeur C_3 dans le registre de texte chiffré.

La phase de Finalisation : Enfin, cette phase a pour but de retourner le tag T. Elle commence donc par faire un XOR_begin entre la clef et x_1 et x_2 puis 12 permutations. Enfin l'algorithme retourne uniquement x_3 et x_4 ayant d'avoir fait un XOR_end avec K.

On peut donc la scinder tout comme la phase d'Initialisation en 5 états différents :

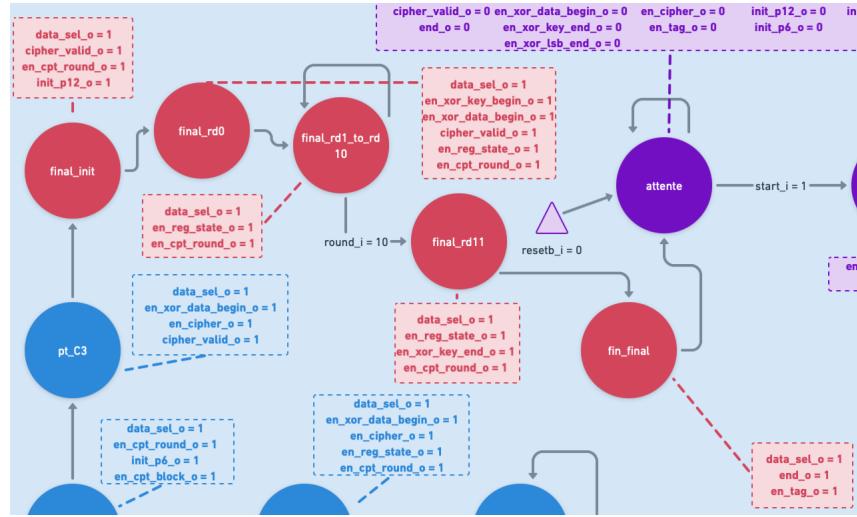


FIGURE III.4 – Schéma du Phase de Finalisation de la FSM

- **final_init** permet de remettre à zéro le compteur de ronde pour 12 permutations et renvoyer le signal de l'envoi de C3 (**cipher_valid_o**).
- l'état **final_rd0** permet d'appliquer le **XOR_begin** de P3 durant la première ronde qui n'a pas été enregistré précédemment.
- **final_rd1_to_rd10** applique au signal 10 rondes
- **final_rd11** fait une ronde tout en appliquant un **XOR_end** entre K et x_3 et x_4 .
- **fin_final** est le dernier état de la FSM : il enregistre le tag qui sera effectivement affiché sur le chronogramme dans l'état d'attente.

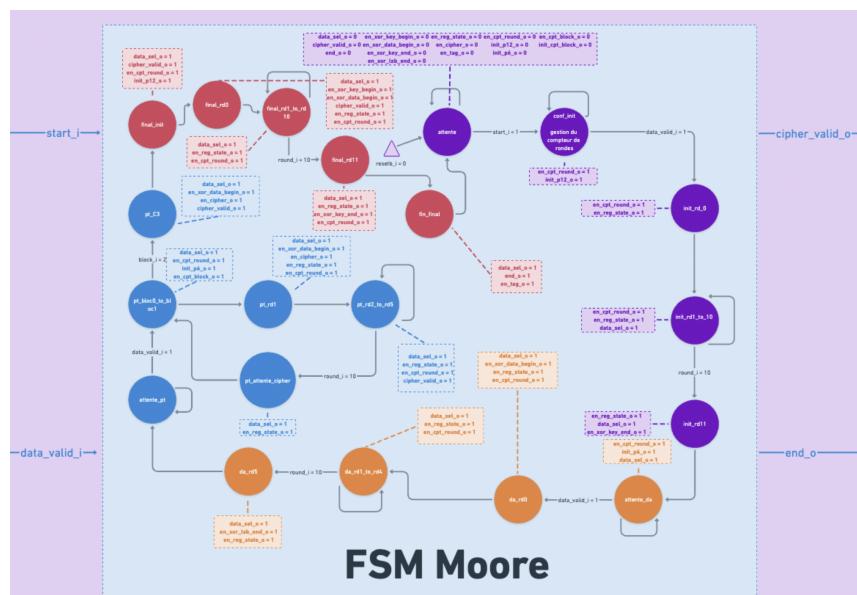


FIGURE III.5 – Schéma du Schéma de la FSM totale

1.2 Optimisation

Après avoir modéliser la FSM je me suis rendu compte que cette dernière possèdait des états qui pouvaient se regrouper en un seul et donc la transformer en machine de Mealy

En effet, voici la liste des nouveaux états (dits **super-états** ou **états composites**) :

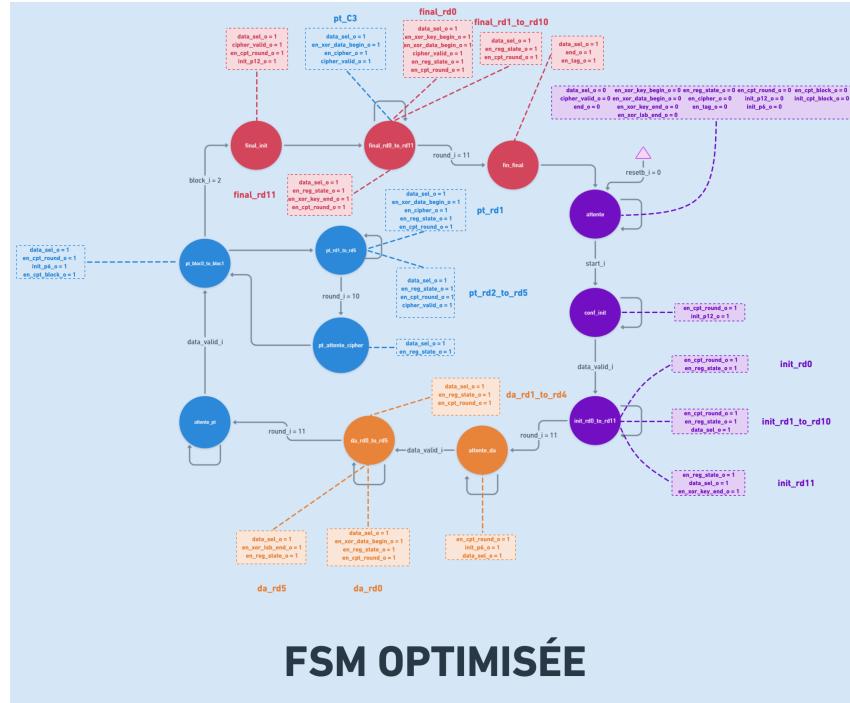


FIGURE III.6 – Schéma du Schéma de la FSM optimisée

- **init_rd0_to_rd11** est la fusion de **init_rd0**, **init_rd1_to_rd10** et **init_rd11**. En effet, une condition dans l'état permet d'indiquer quel signal on doit appliquer à chaque sous-état. Ainsi, la ronde est non nulle, alors on change l'état du multiplexeur. Et plus particulièrement si la ronde est la onzième, alors on active le **XOR_end**. On change également la condition de transition à 11 également car la dernière ronde se fait aussi dans cet état.
- l'état **da_rd0_to_rd4** permet de faire lui toutes les rondes de la phase de **Données Associées** avec trois conditions **if** selon la ronde dans laquelle l'algorithme se situe. La transition passe également à 11.
- **pt_rd1_to_rd5** combine lui les états **pt_rd1** et **pt_rd2_to_rd4**. Il fonctionne de la même manière que les autres avec une condition **if / else**.
- **final_rd0_to_rd11** permet de supprimer l'état **pt_C3** car le **XOR_begin** se fait déjà à l'état **final_rd0**. Il fusionne donc toutes les rondes de la phase de **Finalisation** en un seul état.

2 Quelques modules importants

Avant de pouvoir présenter le module `ascon_top.sv` qui est le cœur de l'algorithme, il faut présenter le compteur de bloc et de ronde qui sont nécessaires :

Compteur de ronde : Le compteur de ronde permet de faire savoir à l'algorithme à quel ronde il se situe. En effet, il possède 5 entrées qui sont respectivement `init_a_i`, `init_b_i`, `en_i`, `clock_i`, et `resetb_i`. Ainsi, le compteur est **double** puisque selon la valeur de `init_a_i` ou `init_b_i` activée, celui-ci va mettre la valeur du compteur à 0 ou 6 (dans le cas où l'on veut faire 6 ou 12 permutations). Il faut également que `en_i` soit à l'état haut. Dans le cas où seul `en_i` est activé, le compteur s'incrémentera à chaque coup d'horloge. Enfin le signal `resetb_i` permet de réinitialiser le compteur si il est à zéro.



FIGURE III.7 – Schéma du Schéma du compteur de ronde

Compteur de bloc : Le compteur de bloc fonctionne de la même manière que le compteur de ronde à la différence qu'il est **simple**. En effet, ce dernier n'a pas de signal `init_b_i`, ce qui rend le compteur beaucoup plus facile à réaliser. Ainsi à chaque coup d'horloge si `init_a_i` et `en_i` sont activés le compteur se réinitialise à zéro.



FIGURE III.8 – Schéma du Schéma du compteur de ronde

3 Le module ascon_top.sv

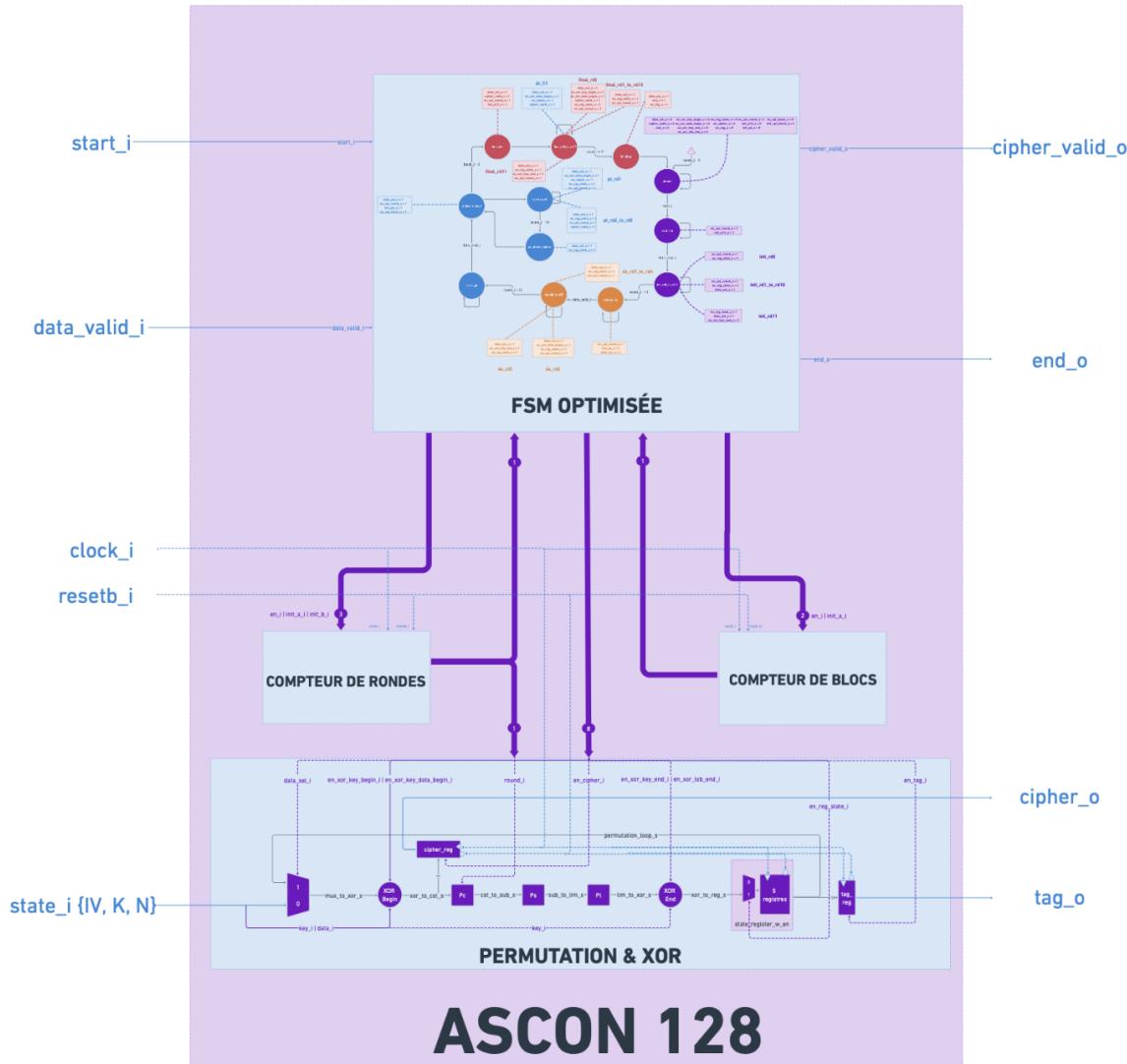


FIGURE III.9 – Schéma du ascon_top

Le module `ascon_top` permet de relier tous les modules ensemble. En effet, il sert à relier la FSM avec le module `permutation`, le compteur de bloc et de ronde ainsi que le compteur de ronde au module `permutation`.

Un fichier .do a été fait pour faciliter la simulation. Pour l'exécuter il faut faire la commande depuis Modelsim :

```
1 do ./SRC/BENCH/ascon_top_tb.do
```

On obtient donc ces résultats :

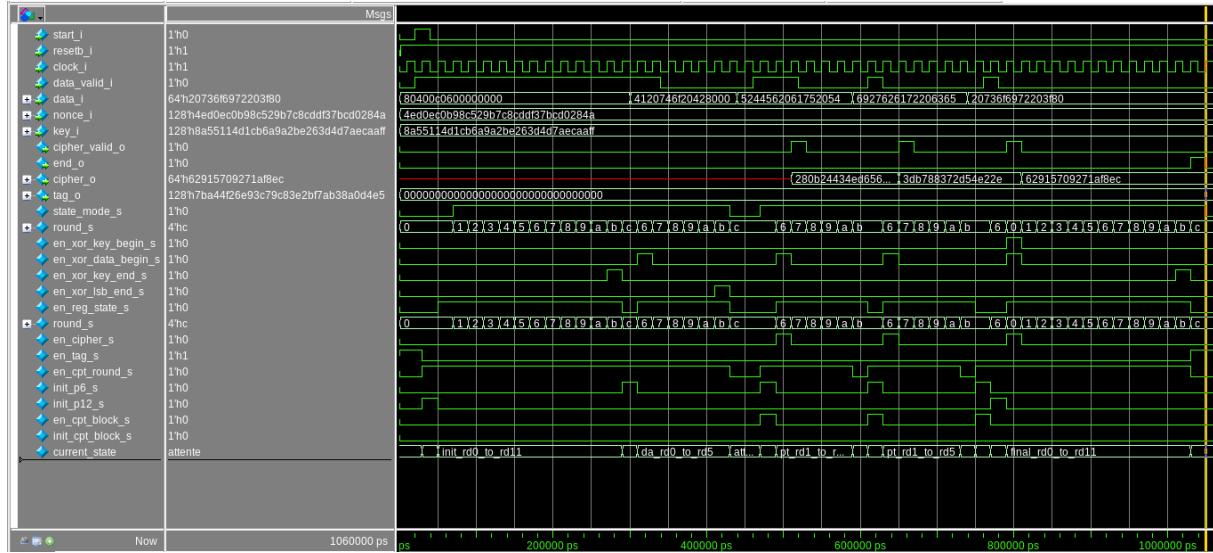


FIGURE III.10 – Résultats du ascon_top_tb.sv

Ce qui donne donc :

```

1 C1 = 28 0B 24 43 4E D6 56 2A
2 C2 = 3D B7 88 37 2D 54 E2 2E
3 C3 = 62 91 57 09 27 1A F8 (EC)
4 T = 7B A4 4F 26 E9 3C 79 C8 3E 2B F7 AB 38 A0 D4 E5

```

Comme on peut le constater comme le registre pour C₁, C₂ et C₃ est le même il écrit un octet en trop pour troisième texte chiffré (l'octet est mis en parenthèse).

4 Difficultés rencontrées

L'algorithme : Je n'avais pas compris au premier abord la structure de l'algorithme. Ainsi, durant le projet j'ai dû tester le module `permutation_V1.sv` (le module de permutation sans l'ajout des XOR), puis le module `permutation_xor.sv`. Cependant, pour tester l'efficacité de ce module j'ai fait un `Test Bench` qui fait toutes les phases de l'algorithme. Cela m'a permis de mieux appréhender la FSM. J'ai donc appris à mieux manipuler les boucles for et les délais d'attentes de coup d'horloge pour rendre le code plus compact.

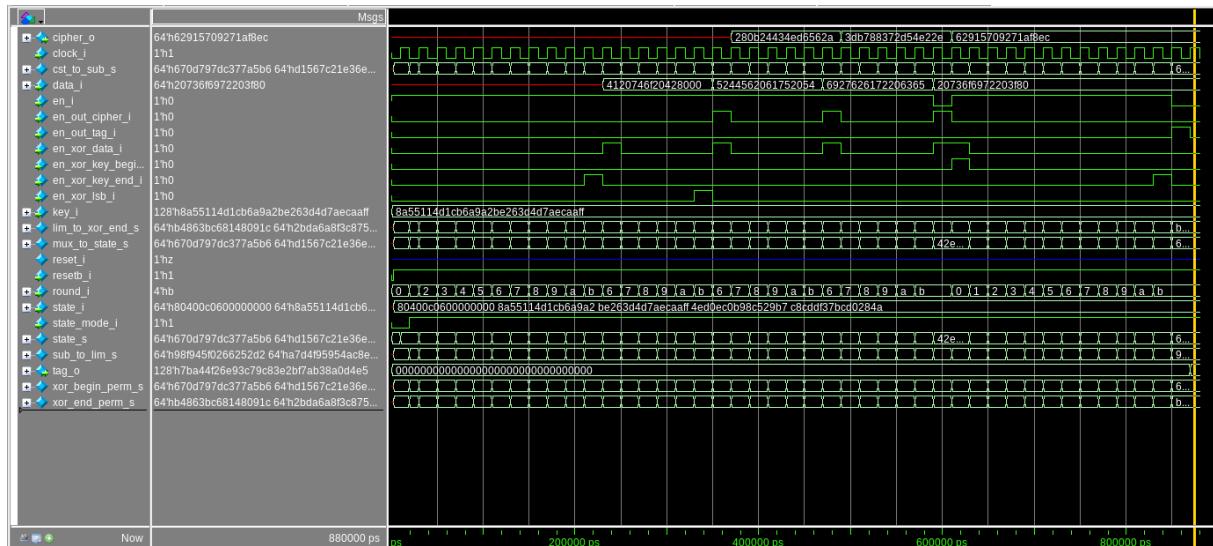


FIGURE III.11 – Résultats du `permutation_xor_tb.sv`

```

1 // Phase d'initialisation
2
3     for (int round = 1; round < 12; round++) begin
4         round_input_s = round; // Incrementation du compteur de
5         tours
6         en_xor_key_end_input_s = (round == 11) ? 1'b1 : 1'b0; // Activation du XOR avec la clef a la fin de l'initialisation
7         @(posedge clock_input_s); // Attendre un cycle d'horloge
8     end
9
10    en_xor_key_end_input_s = 1'b0; // Desactivation du XOR avec la
11    clef
12
13    // Fin de la phase d'initialisation

```

Listing 4 – Exemple d'une boucle for dans le module `permutation_xor_tb.sv` (voir le code pour plus d'informations)

Voici la commande .do a exécuter pour simuler ce `Test Bench` :

```

1 do ./SRC/BENCH/permutation_xor_tb.do

```

Compteur simple et double : Durant la mise en place de ma FSM, j'ai dû comprendre en profondeur les différents modules donnés. Toutefois, ces derniers avaient quelques coquilles dans le code ce qui a favorisé ma compréhension dans le langage Verilog.

```

1 module compteur_simple_init import ascon_pack::*;
2 (
3     input logic    clock_i ,
4     input logic    resetb_i ,
5     input logic    en_i ,
6     input logic    init_a_i ,
7     output logic [3 : 0] cpt_o          //ET pas data_o Ici
8 ) ;

```

Listing 5 – Définition des Entrées/Sorties mal instantiées dans le module
compteur_simple_init.sv

Optimisation de la FSM : L'optimisation de la FSM en machine de Mealy m'a pris plus de temps que je ne le pensait. En effet, le fait de regrouper certains états était contre-intuitif à première vue (mettre l'état pt_C3 dans un super-état de la phase de Finalisation) mais très gratifiant.

IV Conclusion

Pour conclure, le projet d'implémentation de l'algorithme ASCON128 simplifié m'a permis de concevoir et de tester une architecture matérielle pour le chiffrement authentifié avec données associées (AEAD).

Le projet m'a également permis de mettre en pratique les connaissances acquises en matière de conception de circuits numériques. Les différentes phases du projet, allant de la spécification à la vérification, ont été menées avec succès.