Rapport Projet : Modélisation SystemVerilog de l'algorithme de chiffrement ASCON

ATIF Mohammed

26 November 2023

Professeur: Jean-Baptiste Rigaud
ISMIN EI22

Contents

1 Introduction

Dans ce projet, nous allons étudier l'algorithme ASCON128. Cet algorithme garantit la confidentialité des messages en chiffrant leur contenu. De plus, il assure l'authenticité de l'en-tête grâce à l'ajout d'un tag. Le destinataire peut ainsi vérifier ce tag pour s'assurer que le message provient bien de l'expéditeur légitime. Ce mécanisme renforce la confiance dans l'origine et l'intégrité des messages échangés entre deux utilisateurs.

Dans la suite, nous allons découvrir les différentes étapes du chiffrement en détaillant la machine d'état, les deux permutations $p6$ et $p12$, ainsi que l'implémentation de l'ensemble en SystemVerilog.

Formalisme :
Dans toute la suite du projet, nous adopterons les règles suivantes :

1. Le nom du fichier, ainsi que le nom du module correspond toujours au nom de l'instance associée.
2. Les signaux entrants sont complétés par le suffixe "_i".
3. Les signaux sortants sont complétés par le suffixe "_o".
4. Les signaux intermédiaires sont complétés par le suffixe "_s".

Le fichier de compilation permet de lancer directement la simulation du système complet **(ASCON_TOP)** après sa compilation. Les autres scripts pour lancer les autres simulations sont commentés.

2 Vue globale du système

2.1 Entrées/Sorties du système

1. **Les signaux de contrôle** :

   - Une horloge **clock_i**
   - Un signal d'initialisation **resetb_i**

2. **Les entrées :**

   - **data_i** de 64 bits
   - **data_valid_i** indiquant la présence d'une donnée valide sur **data_i**
   - La clé **key_i** de 128 bits
   - Le nombre arbitraire **nonce_i** de 128 bits
   - **start_i** pour commencer le chiffrement

3. **Les sorties** :

   - **cipher_o** sur 64 bits
   - **cipher_valid_o** indiquant la validité de la sortie **cipher_o**
   - **tag_o** sur 128 bits
   - **end_o** indiquant la fin du chiffrement du message

2.2 Blocs élémentaires du système

La Figure 1 présente les différents blocs élémentaires du système ASCON. Chaque bloc a un rôle spécifique dans le fonctionnement global de l'algorithme de chiffrement. Les blocs incluent :

- FSM (Machine à États Finis) : Il s'agit de la machine d'état qui contrôle le système, coordonnant les différentes étapes du processus de chiffrement.
- Permutation : Ce bloc applique des transformations sur la trame de données de 320 bits. Chaque permutation peut effectuer 6 ou 12 rotations en fonction du Compteur Double Init.
- Compteur de Blocs : Ce bloc gère les données chiffrées, comptant le nombre de blocs traités dans le processus global de chiffrement.

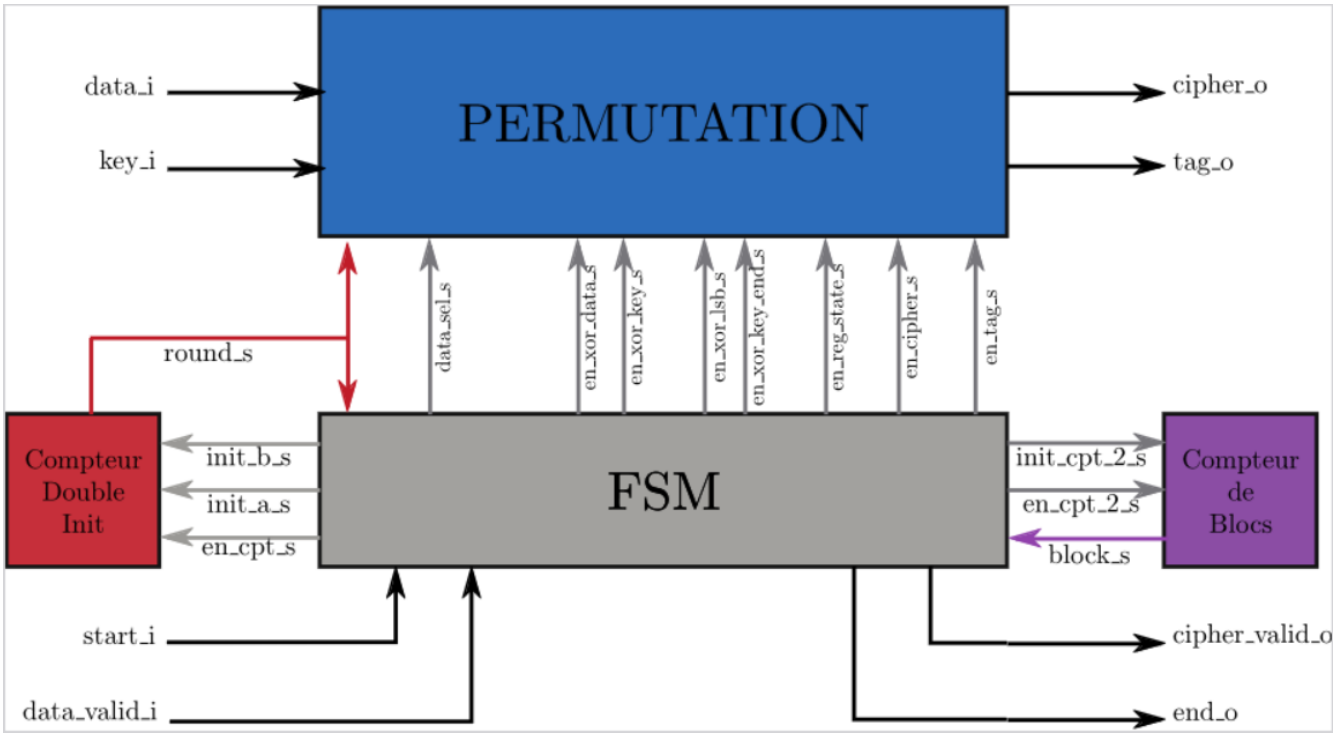

Figure 1: Divers blocs du système ASCON

3 Modélisation des blocs élémentaires (implémentation en SystemVerilog)

Les données d'entrée, qui sont de 320 bits, sont divisées en 5 registres de 64 bits que nous appellerons x_0, x_1, x_2, x_3, x_4, tels que x_0 représente les 64 bits de poids fort et x_4 les 64 bits de poids faible.

3.1 Permutation globale:

La permutation de base établit une connexion entre cinq instances, comme illustré dans la figure ci-dessous (Figure 2), formant la base du système ASCON128. Ces instances sont reliées par des signaux intermédiaires de 320 bits chacun. Lorsqu'une donnée est sélectionnée par le multiplexeur à l'entrée(state_i ou reg_to_mux_s), elle subit trois transformations avant d'atteindre le registre.

À la sortie de la permutation unitaire, un registre est mis en place pour stocker la donnée. Lors du prochain front montant de l'horloge, la sortie du registre est renvoyée au multiplexeur, qui bascule vers l'autre entrée jusqu'à ce que le nombre de rounds souhaité soit atteint.

Ainsi, la permutation unitaire sélectionne entre deux entrées de données de type_state, puis stocke la donnée en sortie dans un registre après avoir effectué toutes les opérations.

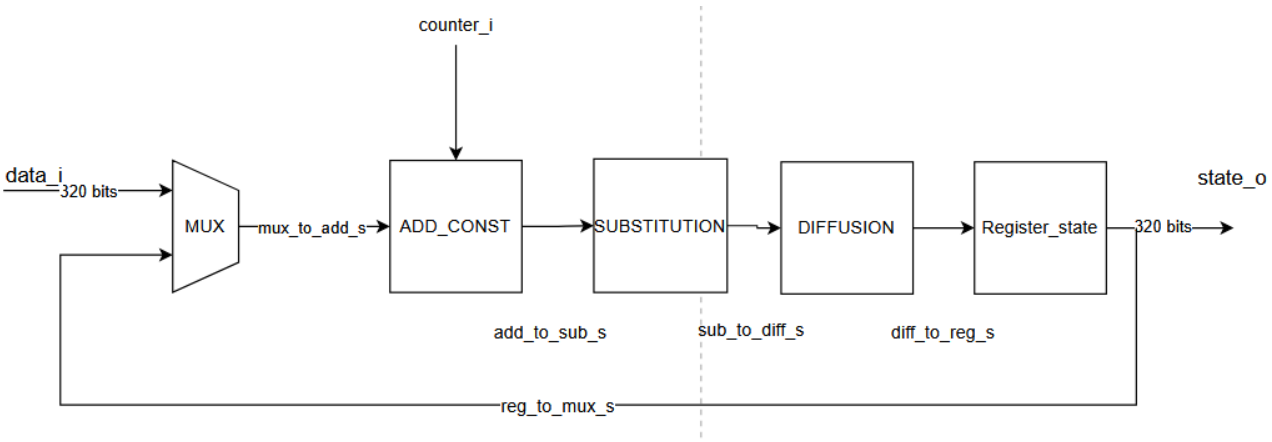

Figure 2: Permutation de base dans le système ASCON128

Pour simplifier la construction du système ASCON128, des opérations ou exclusif (`XOR begin et XOR end`) ont été ajoutées, placées juste avant les couches add constante et après la diffusion (Figure 3).

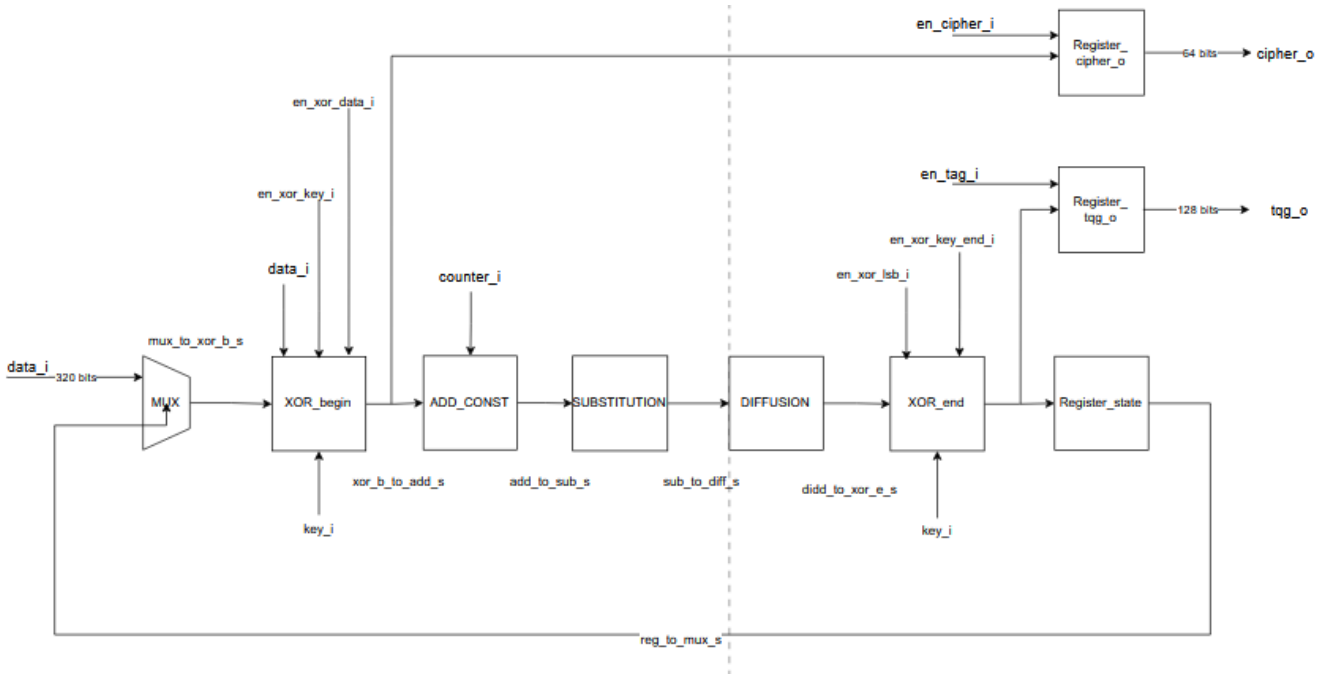

Figure 3: permutation complète

De cette manière, il est possible de construire un système ASCON128 performant en créant des connexions entre plusieurs permutations.

3.1.1 Couche d'addition de constante :

Cette couche permet d'ajouter une constante au registre x2. Cela revient à appliquer une opération de XOR (ou exclusif) entre le registre x2 et une constante prédéfinie indixée par la valeur de round_i. Cette valeur est obtenue à l'aide du compteur double init et représente le nombre de fois que la permutation a tourné.

```
assign state_o[2][7:0] = state_i[2][7:0] ^ round_constant[round_i];
```

Figure 4: section du code add_const

Simulation

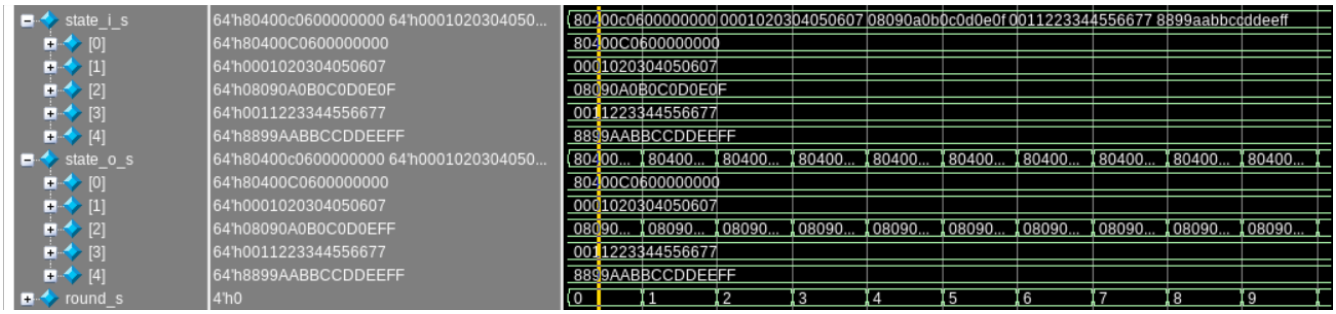

Figure 5: Chronogramme du signal à la sortie de la couche

Dans la simulation du test bench, on a pris comme valeur de `state_i` le signal `state_i` du début, et à chaque round, on remarque qu'on obtient bien les valeurs voulues. Sur la simulation (voir Figure 5), nous avons placé le curseur sur la valeur de sortie à la ronde 0, et à la lecture de la valeur de x2 sur la fenêtre de gauche, nous pouvons affirmer que notre couche fonctionne correctement.

3.1.2 Couche de substitution

Cette couche sert à substituer chaque bloc de cinq bits, qui représente les bits des mêmes colonnes dans les cinq registres (voir Figure 6), par leurs équivalents stockés dans la table S-box. Cela est réalisé pour tous les 64 bits présents dans un registre.

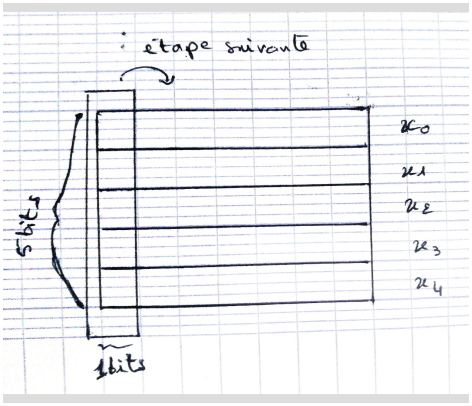

Figure 6: Représentation de la sélection des cinq bits par colonnes.

```
for ( i =0; i <64; i ++)
begin : label1

s_box s_box_inst (
. substitute_i ({ state_i[0][i] , state_i[1][i] , state_i[2][i] , state_i[3][i] , state_i[4][i] } ) ,
. substitute_o ({ data_inter[0][i] , data_inter[1][i] , data_inter[2][i] , data_inter[3][i] , data_inter[4][i] })

);
```

Figure 7: Section du code traduisant l'algorithme de la couche substitution

Simulation:

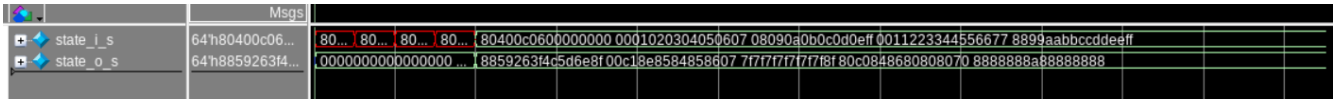

Figure 8: Chronogramme du signal de sortie

Les résultats sont vérifiés pour `round_i` = 0, en prenant comme signal d'entrée le signal de sortie de la couche addition constante pour le même `round_i`.

3.1.3 Couche de diffusion linéaire

Cette transformation prend en entrée les cinq registres de 64 bits concaténés dans state_i. Ensuite, la transformation effectue des rotations cycliques de différentes valeurs, puis elle applique des opérations (OU exclusif) sur les différents registres intermédiaires, pour finalement les affecter aux sorties state_o.

```
assign state_o[0] = state_i[0] ^ {state_i[0][18:0],state_i[0][63:19]} ^ {state_i[0][27:0],state_i[0][63:28]};
assign state_o[1] = state_i[1] ^ {state_i[1][60:0],state_i[1][63:61]} ^ {state_i[1][38:0],state_i[1][63:39]};
assign state_o[2] = state_i[2] ^ {state_i[2][0],state_i[2][63:1]} ^ {state_i[2][5:0],state_i[2][63:6]};
assign state_o[3] = state_i[3] ^ {state_i[3][9:0],state_i[3][63:10]} ^ {state_i[3][16:0],state_i[3][63:17]};
assign state_o[4] = state_i[4] ^ {state_i[4][6:0],state_i[4][63:7]} ^ {state_i[4][40:0],state_i[4][63:41]};
```

Figure 9: Les opérations appliquées aux cinq registres

Simulation:

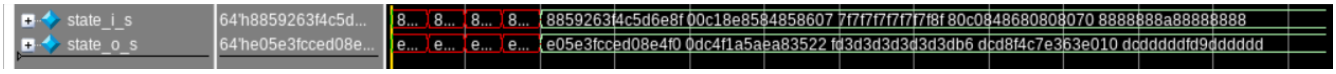

Figure 10: Chronogramme du signal de sortie

Cette couche a été testée pour la ronde 0. En lisant les valeurs obtenues à la sortie de l'unité (voir Figure 10), on peut affirmer que cette couche fonctionne correctement.

3.1.4 Preuve du bon fonctionnement des trois transformations

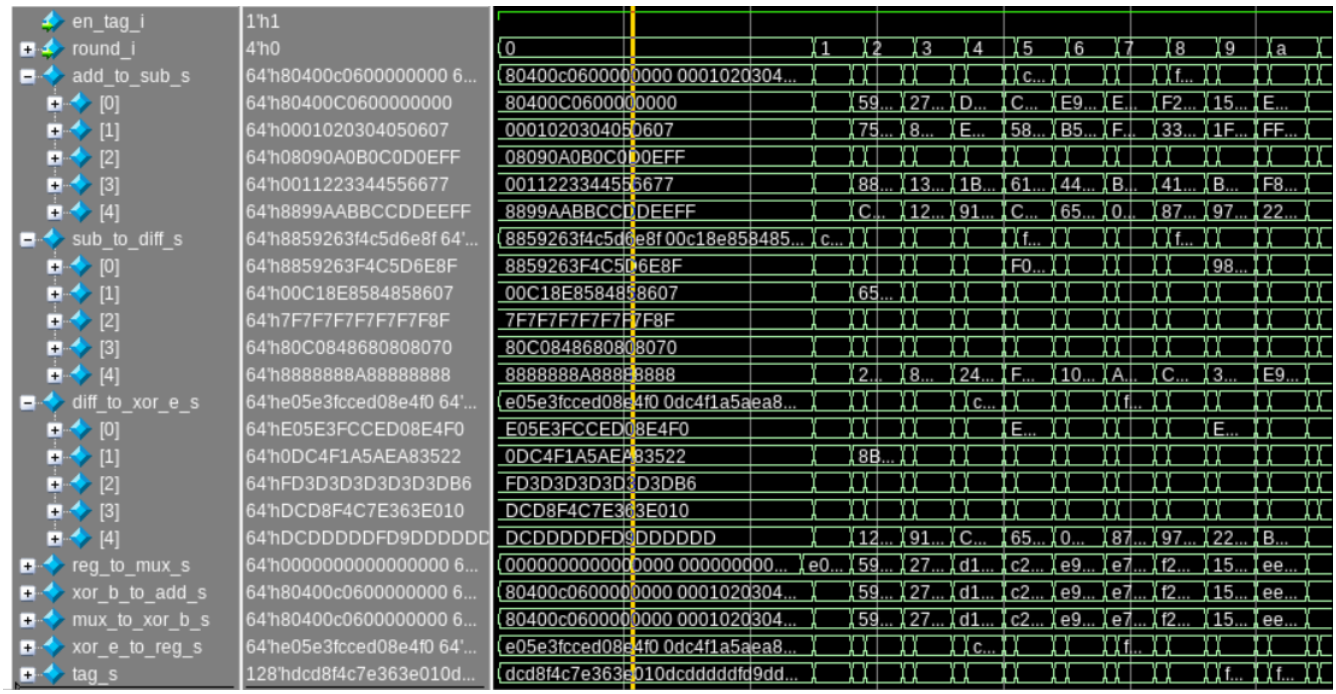

Figure 11: Chronogramme des signaux obtenus à la sortie de chaque couche pour la *round_i = 0*.

Commentaire : On remarque bien que les signaux dans la simulation, récupérés à la sortie de chacune des trois couches, correspondent bien aux résultats attendus.

La sortie de la couche **add_constante** est le signal intermédiaire **add_to_sub_s**.

La sortie de la couche **substitution** est le signal intermédiaire **sub_to_diff_s**.

La sortie de la couche **diffusion** est le signal intermédiaire **diff_to_xor_e_s.**

Au cours du fonctionnement de la permutation, les données d'entrée passent par neuf opérations XOR, tant à l'entrée qu'à la sortie de la permutation. Ces opérations se déclinent en quatre types distincts :

- Un XOR avec une donnée data_i de 64 bits à l'entrée de la permutation.
- Un XOR avec la clé Key_i de 128 bits, l'entrée ou à la sortie de la permutation.
- Un XOR avec une donnée de 256 bits formée par la concaténation de 255 bits nuls et d'un bit égal à 1 en LSB à la sortie d'une permutation.

Ainsi, on peut regrouper les opérations XOR en deux ensembles distincts, à savoir (XOR begin) et (XOR end)

3.1.5 XOR_begin

Cette instance sert à effectuer l'opération XOR. Elle a deux signaux de contrôle(Figure 12). Le premier est `en_xor_data_i` pour les blocs de données. Il effectue l'opération XOR entre `state_i[0]` et `data_i` si

`en_xor_data_i = 1'b1`. Le deuxième est `en_xor_key_i` pour la clé key_i. Il effectue la même opération entre la clé et {`state_i[1]`, `state_i[2]`} si `en_xor_key_i = 1'b1`.

Et à la sortie, on récupère les 5 registres concaténés sous forme d'une donnée de 320 bits `state_o`.

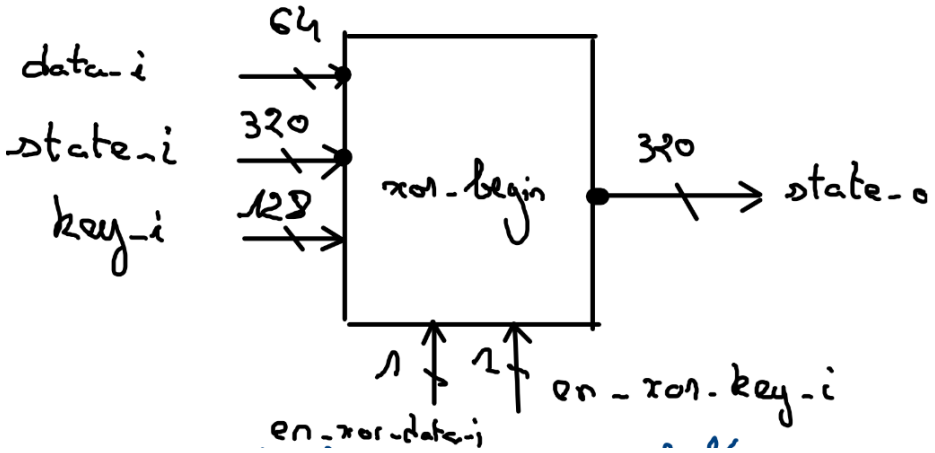

Figure 12: XOR begin

```
assign state_o[0] = en_xor_data_i == 1'b1 ? state_i[0]^data_i : state_i[0];

assign data_inter = en_xor_key_i == 1'b1 ? key_i ^ {state_i[1], state_i[2]} : {state_i[1], state_i[2]};
```

Figure 13: code XOR begin

3.1.6 XOR_end

De la même manière que l'instance précédente, `XOR end` récupère la donnée à son entrée et effectue deux types d'opération XOR selon les deux signaux de contrôle. Si `en_xor_key_end_i = 1'b1`, alors {`state_i[3]`, `state_i[4]`} entre en XOR avec la clé key_i. Si `en_xor_lsb_i = 1'b1`, seule $state\_i[4][0] \oplus 1$ est effectuée.

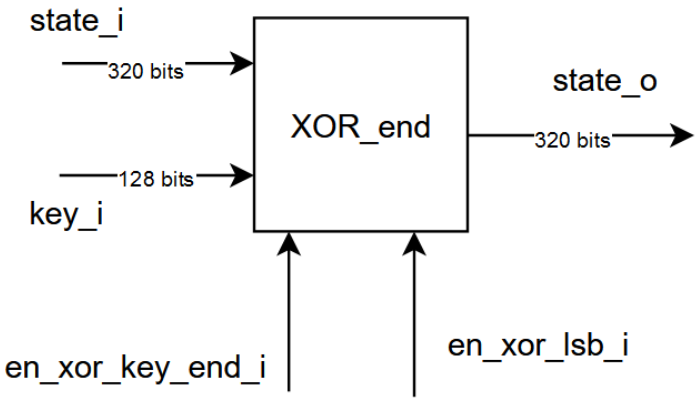

Figure 14: XOR_end

Sur mon code verilog, J'ai utilisé l'opérateur ternaire (Figure 15)qui traduit l'algorithme suivant.

- Si en_xor_lsb_i est égal à 1'b1, alors data_inter prend la valeur **state_i[3], state_i[4]** xor en_xor_lsb_i.
- Sinon, si en_xor_key_end_i est égal à 1'b1, alors data_inter prend la valeur key_i xor **state_i[3], state_i[4]**.
- Sinon, data_inter prend la valeur **state_i[3], state_i[4]**.

  Avec data_inter une donnee intermediaire recuperee a la sortie

```
assign data_inter = en_xor_lsb_i == 1'b1 ? {state_i[3],state_i[4]^en_xor_lsb_i} : en_xor_key_end_i == 1'b1 ? key_i ^ {state_i[3], state_i[4]} : {state_i[3],
state_i[4]};
```

Figure 15: code-xor-end

3.1.7 Registre state

Pour chaque ronde de la permutation, on a besoin d'enregistrer la donnée à la sortie. Pour cela, on a placé un registre à la sortie de celle-ci. Le registre a un signal de contrôle, qui est `en_reg_state_i`, et il est connecté aux signaux `clock_i` et `resetb_i` du système. À sa sortie, à chaque coup d'horloge, on récupère la donnée stockée de `type_state` sur 320 bits pour la réutiliser à l'itération suivante, après avoir changé l'entrée du multiplexeur.

Ce composant synchrone se remet à zéro lorsque le signal de reset est à l'état haut.

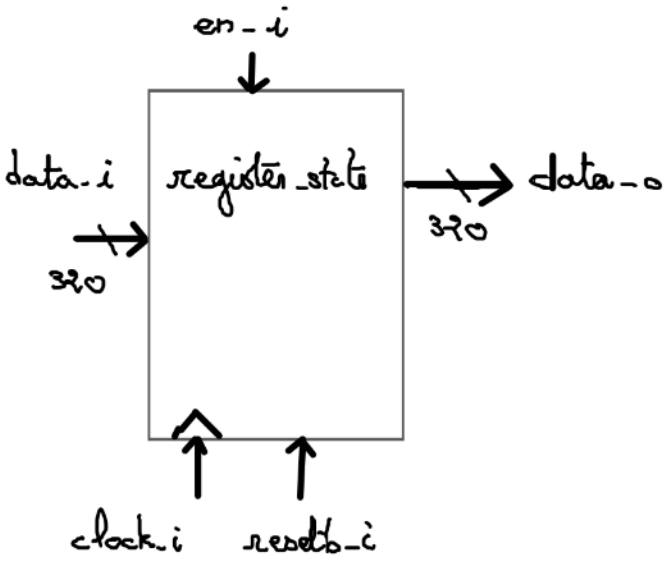

Figure 16: regiter state

3.1.8 Registre cipher_o

Ce registre permet de stocker le texte chiffré C représenté par `cipher_o`. Il est décomposé en quatre blocs, C_1, C_2, C_3, et C_4, qui sont des blocs de 64 bits (Figure 17).

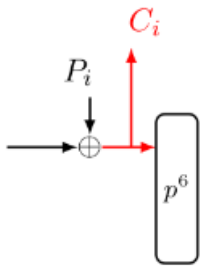

Figure 17: Cipher en sortie après traitement du texte clair

On commence à récupérer ces blocs à la phase de **Texte_clair**, et le nombre de ces blocs récupérés est compté par un compteur de blocs auquel on a connecté le système. Ce compteur assure la récupération exacte de quatre blocs.

La mise en œuvre matérielle d'un registre de 320 bits pourrait nécessiter des circuits plus complexes par rapport à un registre de 64 bits, alors nous avons choisit de connecter directement notre signal state_o[0] de 64 bits à l'entrée de notre registre(Figure 18), Afin de réduire la consommation de l'énergie

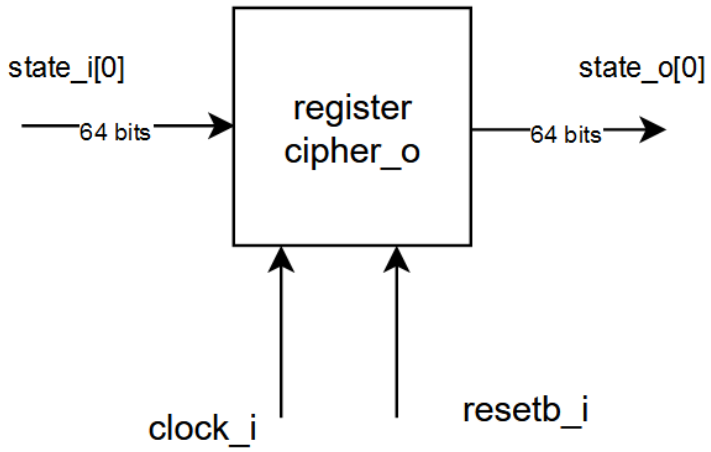

Figure 18: registre cipher_o

3.1.9 Registre tag_o

Ce troisième registre permet de stocker le tag T. Ce tag est formé de la concaténation de **state_o[3] et state_o[4]** à la sortie du système ASCON, à la fin de la phase de finalisation.

Pour les meme raison que le registre d'avant, nous avons choisit un registre de 128 bits pour stocker notre signal tag_o.

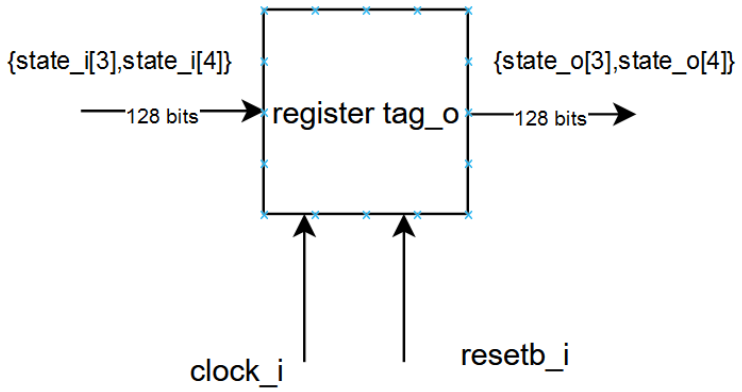

Figure 19: registre tag_o

3.1.10 Simulation et Validation de la permutation complète

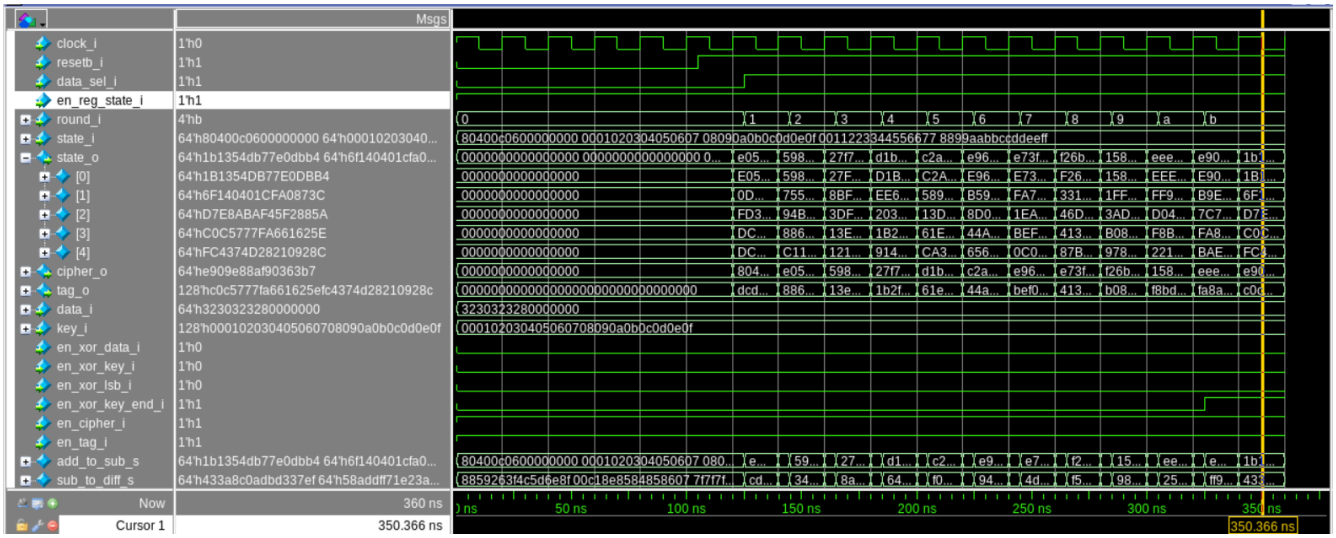

Figure 20: Chronogramme de la permutation complète

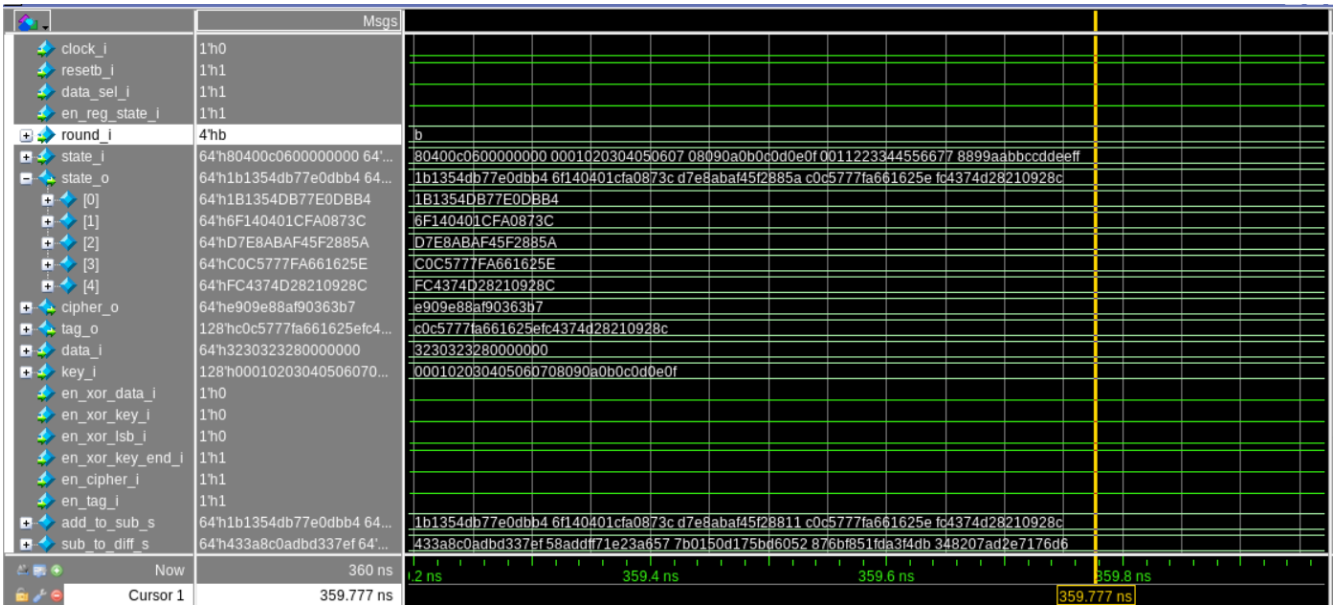

Figure 21: Resultat pour round_i = 4'hb apres le XOR end

Commentaire :

Afin de tester le bon fonctionnement de tous les blocs de notre permutation, nous avons créé un test bench dans lequel nous incrémentons, à chaque coup d'horloge, nous-même la valeur du compteur round qui va de 0 à 11. Ce test bench permet de comparer les résultats obtenus par la simulation avec ceux disponibles pour la phase d'initialisation. , nous avons également activé les registres du cipher pour voir si nous récupérons toujours `state_o[0]`.

Résultats :

Tous les résultats de sorties pour chaque ronde sont bien exacts, sur la Fenêtre gauche de la simulation (Figure 20) on peut lire les valeur de sortie ,même pour la dernière boucle effectuée par la permutation (`round_i = 11`). Nous avons activé le XOR final avec la clé, et le résultat est valide (Figure 21), ce qui assure le bon fonctionnement de notre permutation.

3.2 Les deux Compteurs

1. **Compteur_double_init:**

   Pour contrôler le nombre de fois que la permutation a tourné de manière automatisée, on met en place un compteur `ronde`, qui permet à la permutation de fonctionner 6 ou 12 fois. Ce compteur a trois signaux de contrôle : `init_a_i`, `init_b_i` et `en_cpt_i`. `en_cpt_i` active le compteur.

   Si on veut que le compteur compte 6 fois, on passe `init_b_i` à 1, et le compteur est initialisé à 6 et finit par 11. Sinon, on passe `init_a_i` à 1, et notre compteur est initialisé à 0 et finit par 11 pour avoir 12 valeurs.

   Si les deux signaux `init_a_i` et `init_b_i` sont à l'état bas, avec `en_cpt_i` à l'état haut, le compteur s'incrémente. La valeur de sortie `round_o` est connectée à la fois à la machine d'état et à la permutation.

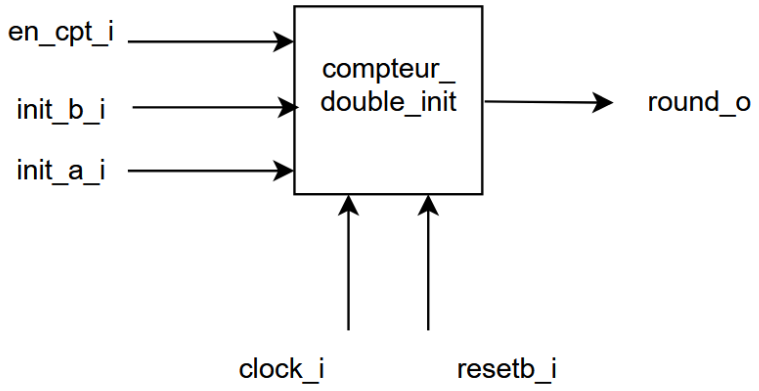

Figure 22: Compteur_double_init

Simulation et validation :

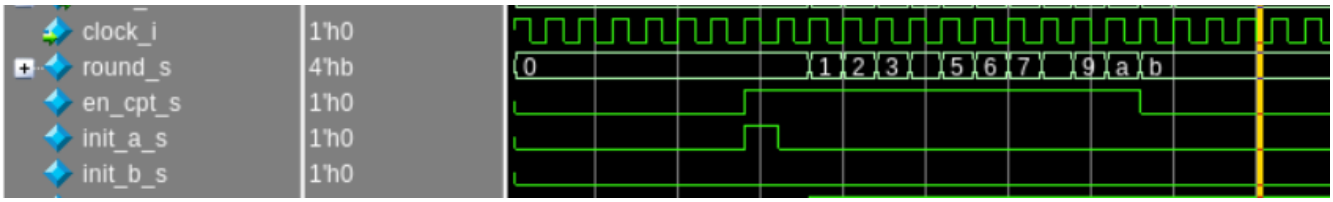

Figure 23: Chronogramme du Compteur_double_init

2. **Compteur_blocs :**

   Ce compteur a deux signaux d'entrées, qui sont *en_cpt_2_i* et *init_cpt_2_i*. Il est initialisé à zéro au début du calcul et permet de savoir le nombre de cipher récupérés.

   Chaque fois que le signail *en_cpt_2_i* passe à l'état haut, la valeur à la sortie du compteur s'incrémente. Cela est illustré dans la (Figure 25).

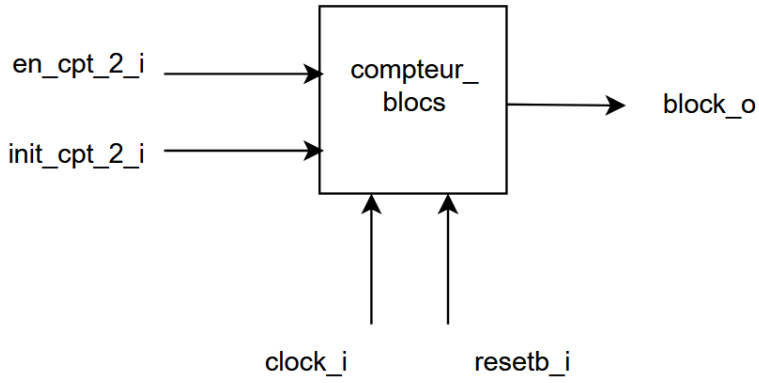

Figure 24: Compteur_blocs

Simulation et validation :

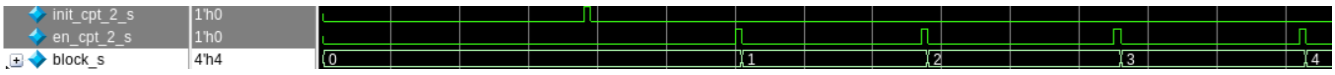

Figure 25: Chronogramme du Compteur_blocs

3.3 Machine d'état de Moore

La machine d'état est l'unité centrale du système. Elle gère son fonctionnement en contrôlant les signaux d'entrée des trois autres blocs. En tout, elle pilote 13 signaux de contrôle qui sont répartis de la manière suivante :

8 signaux qui pilotent la permutation :

- `data_sel_i` pour le multiplexeur.
- `en_xor_key_i` et `en_xor_data_i` pour le XOR begin.
- `en_xor_key_end_i` et `en_xor_lsb_i` pour le XOR end.
- `en_reg_state_i` pour le `reg_state`.
- `en_cipher_i` pour la donnée chiffrée.
- `en_tag_i` pour le tag.

3 signaux pour piloter le compteur double init.
2 signaux pour piloter le compteur de blocs.

La machine d'état dispose de 13 sorties comme signaux de contrôle et 2 autres signaux qui sont `end_o` et `data_valid_o`.

Et ces entrées sont `start_i` pour lancer le calcul et `data_valid_i` qui vaut 1 lorsque les données `data_i` sont valides.

Nous avons effectué un bilan des différents signaux connectés à la machine d'état, on en arrive au choix de la machine d'état adéquate.

Le choix d'utiliser une machine d'état de type **Moore** pour contrôler le flux de signaux dans le chiffrement ASCON128 a été motivé par la facilité de synthèse, notamment parce que les mises à jour d'état ne se produisent qu'au rythme des impulsions d'horloge, à l'exception du reset qui est asynchrone. Cette machine d'état englobe les quatre phases du chiffrement : l'initialisation, le traitement des données associées, le chiffrement du texte clair et la finalisation.

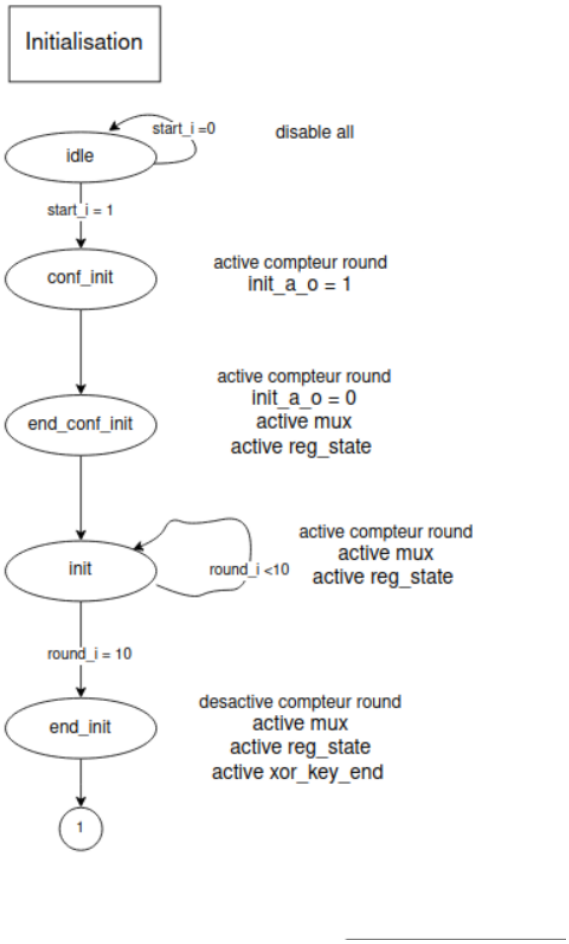

Figure 26: Diagramme de la machine d'état

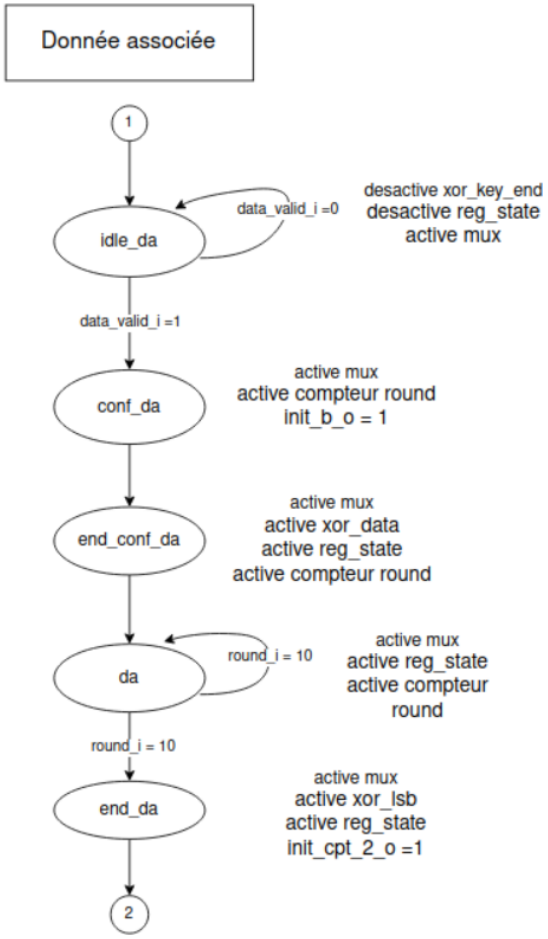

Figure 27: Diagramme de la machine d'état

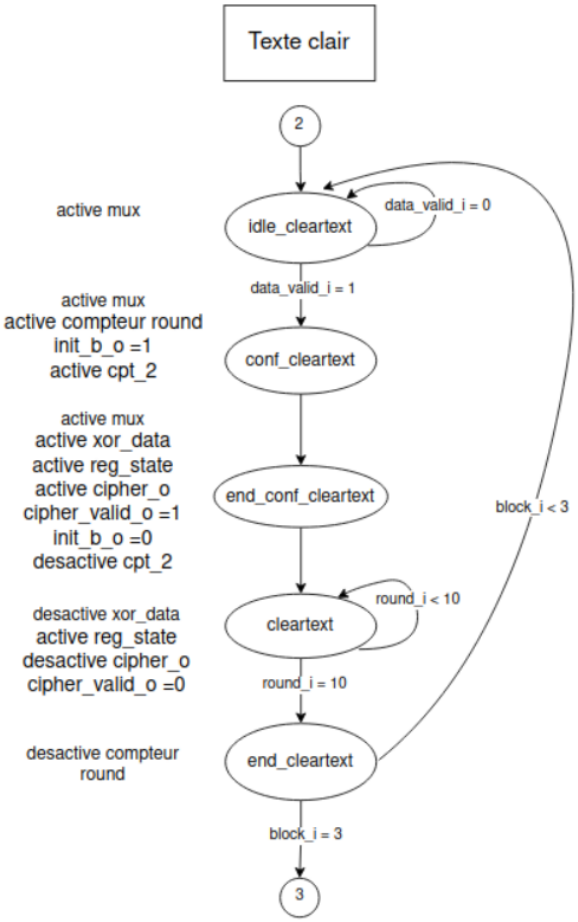

Figure 28: Diagramme de la machine d'état

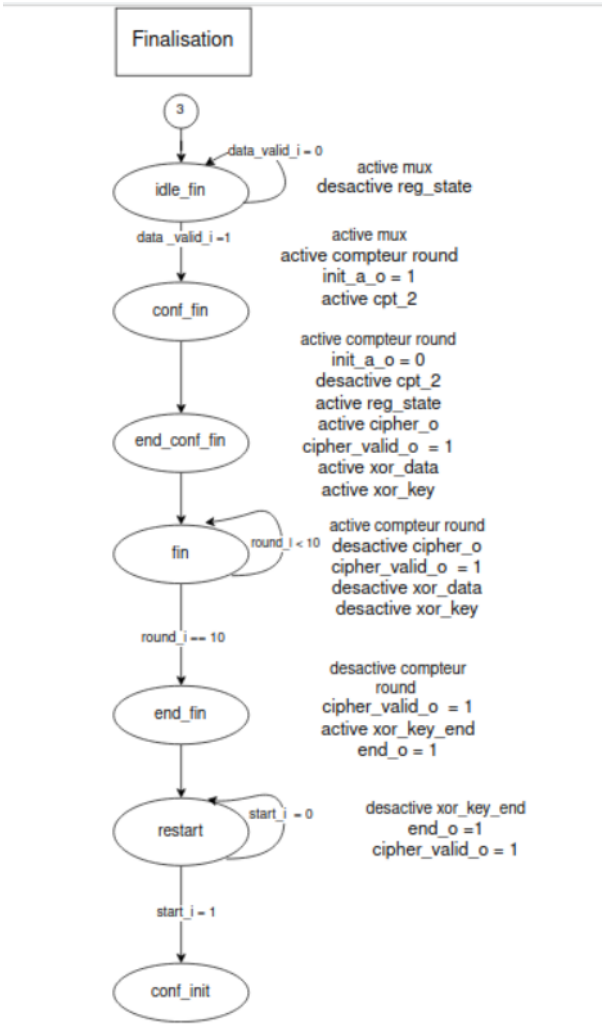

Figure 29: Diagramme de la machine d'état

Analyse des états:

Au total, nous avons utilisé 21 états tout au long du fonctionnement de la machine d'état.On démare lorsque le signal start_i est à l'état haut. À chaque phase, les permutations sont effectuées 6 ou 12 fois. À chaque itération, nous testons si le nombre de rounds_i nécessaire a été atteint. Si c'est le cas, on test le signal data_valid_i, qui permet la transition d'une phase à une autre, s'il est à l'état bas on reste bloquer sur l'état au début de la phase actuelle repérée par idle.

Dans chaque phase on teste si on est arrivé au nombre de round voulu. Par exemple dans la partie initialisation (Figure 26), dans l'état init on test si round = 10, mais on a besoin que la permutation tourne 12 fois. En effet, avant l'état init on a déjà récuppéré la première donnée de la permutation, alors la pemutaion a tourné une fois et round = 1 ,après à l'état init elle tourne 10 fois, alors on a à cette étape une permutation qui a tourné 11 fois, la dernière round de la permutation est éffectuée à l'état end_init, ce qui fait 12 au total, c'est le même principe.

Chaque fois que le signal data_valid_i est à l'état haut, nous indiquons la présence d'une donnée associée, et nous avons correctement récupéré les bonnes valeurs des quatre blocs de cipher_o.

Pour la phase du texte clair(Figure 28), nous sommes censés récupérer 4 textes clair, qui sont les données chiffrées. Cependant, la condition que nous avons ajoutée sur le compteur de blocs indique que

nous n'avons récupéré que 3 chiffres. Le quatrième sera récupéré au début de la phase de finalisation, car il est obtenu après une opération XOR avec P4 à la sortie de la troisième permutation P6. La conception de notre module XOR_end ne permet pas une telle opération. Cependant, cela peut être permis en utilisant le module XOR_begin à l'entrée de la dernière permutation P12.

La partie texte clair est une association en série de trois permutation P6 identiques, alors on met un test qui permet de revenir à l'état idle_cleartext tant que block_i < 3.

Le tag_o est obtenu à l'état "end_fin"(Figure 29), et la valeur obtenue lors de la simulation correspond bien au résultat souhaité, ce qui valide le bon fonctionnement de notre système ASCON128.

Après l'étape de finalisation(Figure 29), le système passe à l'état restart, où il attend de nouvelles données et le prochain signal `start_i` à l'état haut.

Pour la conception de la machine d'état, nous avons procédé étape par étape. Au début, nous avons simulé uniquement la partie d'initialisation. Après nous être assurés des résultats en sortie, nous avons ajouté les états associés à la donnée, puis nous les avons intégrés dans notre machine à états finis (FSM). Ensuite, nous avons lancé la simulation, et après vérification, nous sommes passés à l'étape suivante. Nous avons répété ce processus pour chaque étape successivement.

La totalité des résultats de validation sera développée dans la partie suivante.

4 Modélisation du fonctionnement global (test bench et résultats)

Pour simplifier la vérification des données obtenues lors de la simulation, nous avons ajouté une sortie `state_o` qui représente la sortie du registre `reg_state` dans la permutation globale. Sur les différents chronogrammes, nous affichons la sortie `state_o` d'une manière détaillée, avec `state_o[0]`, `state_o[1]`, `state_o[2]`, `state_o[3]` et `state_o[4]`. Nous avons choisi de visualiser aussi les signaux internes de notre système pour une étude plus détaillée. Les chronogrammes suivants sont des simulations du DUT de notre test bench.

4.1 Sortie initialisation

Après le passage de la donnée {64'h80400C0600000000, key_i, nonce_i} par la permutation P12 et l'opération XOR à la sortie avec key_i, on récupère les bonnes données lors de la simulation.

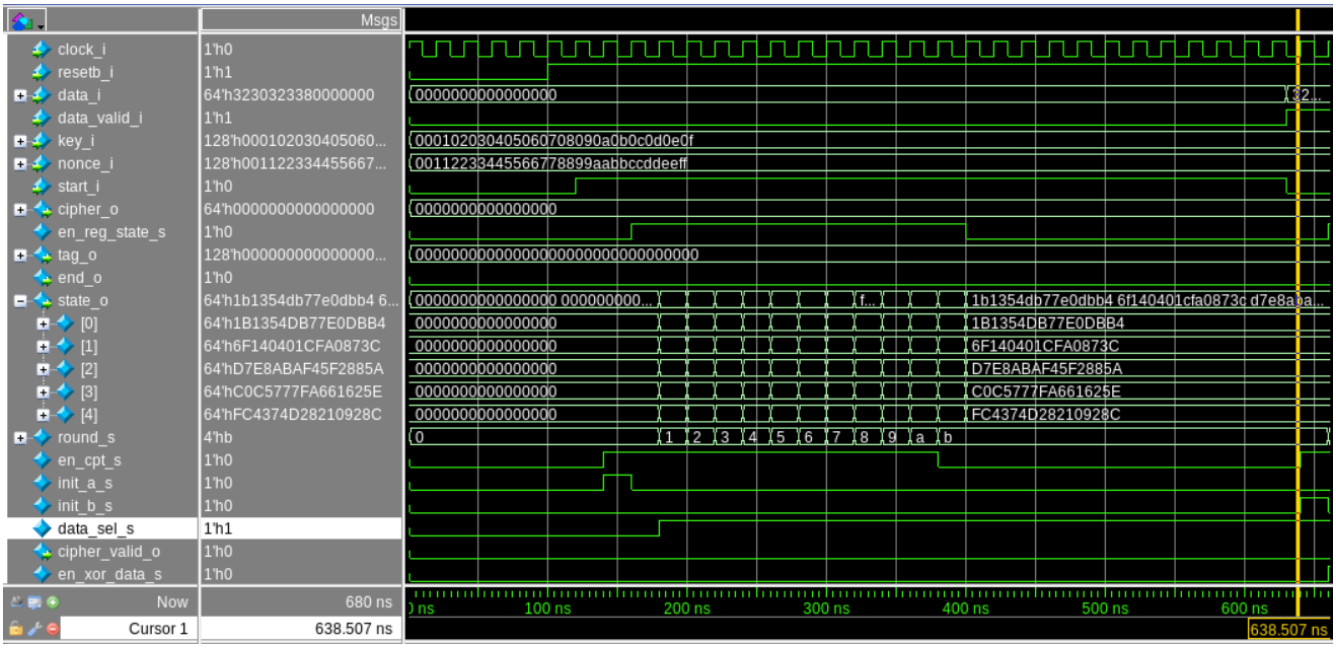

Figure 30: Chronogramme de la phase initialisation

4.2 Sortie donnée associée

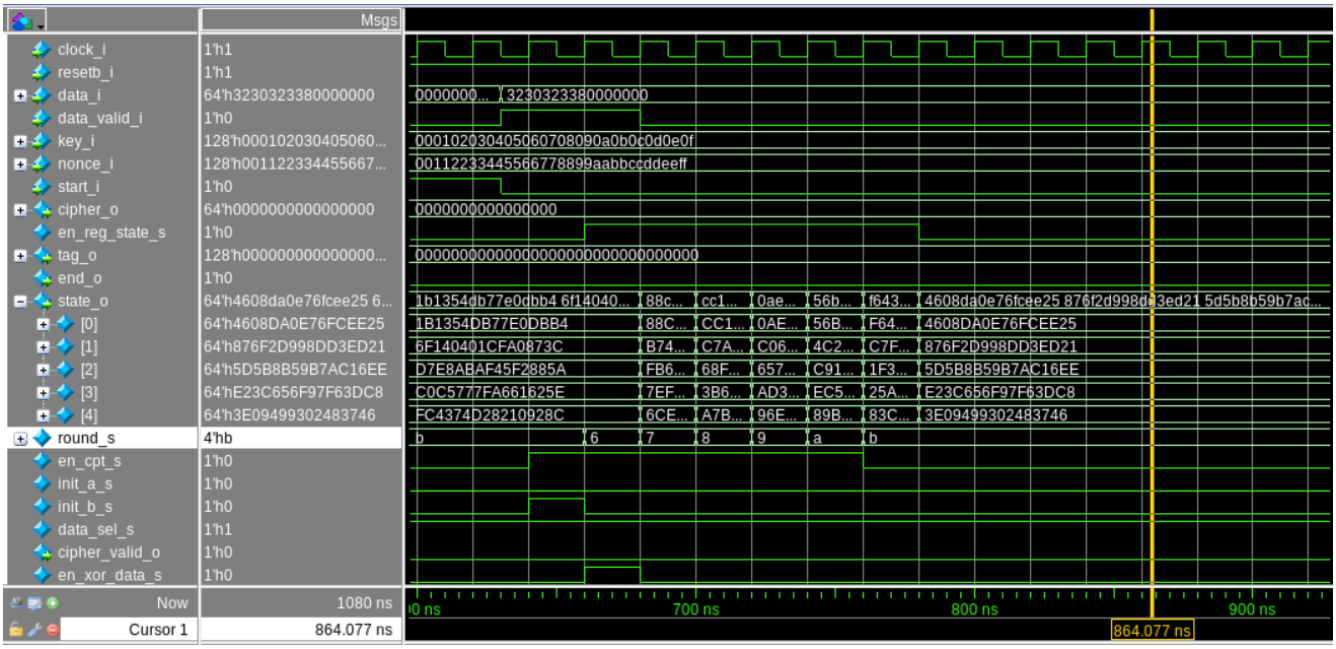

Figure 31: Chronogramme du bloc donnée associée

Sur l'emplacement du curseur, les cinq signaux de sortie de cette phase sont clairement visibles : state_o[0], state_o[1], state_o[2], state_o[3] et state_o[4]. Ces signaux sont détaillés sur mon chronogramme, comme illustré dans la Figure 31. À la fin de cette phase, le signal register_end est désactivé.

4.3 Sorties texte clair

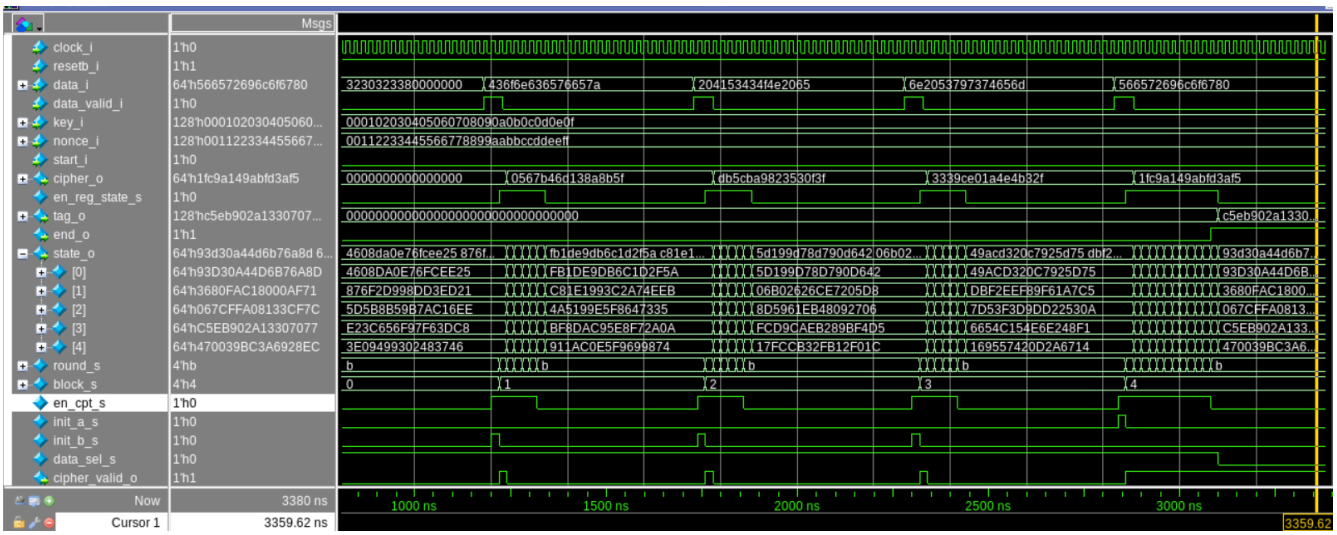

Figure 32: Chronogramme du bloc le texte clair obtenu

Pour chaque incrémentation de `bloc_i` ,representé sur la simulation par bloc_s, on récupère une donnée C_i de 64 bits. Au `bloc_i` = 4, on récupère le dernier *cipher*, et en concaténant les quatre valeurs obtenues, on remarque que cela correspond à la valeur voulue de `cipher_o`. Sur le chronogramme on peut lire les valeurs de cipher_o (Figure 32).

```
C 05 67 B4 6D 13 8A 8B 5F DB 5C BA 98 23 53 0F 3F 33 39 CE 01 A4 E4 B3 2F 1F C9 A1
  49 AB FD 3A (31 octets)
```

Figure 33: Le texte chiffré C de 248 bits

4.4 Sortie finalisation

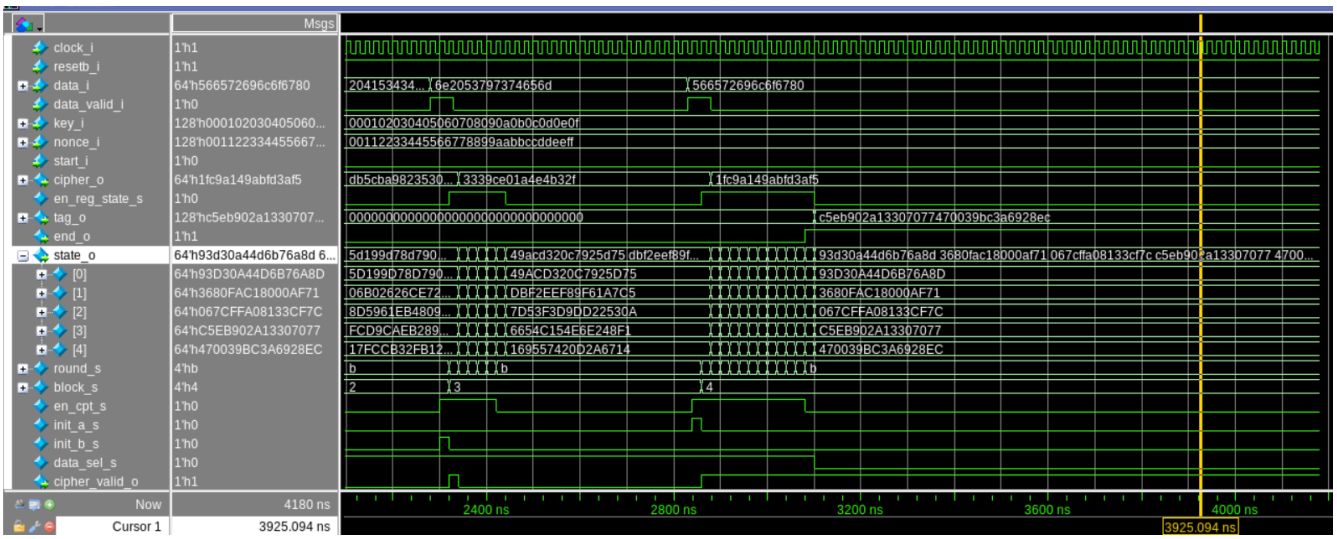

Figure 34: Chronogramme affichant les signaux à la sortie du système

À l'emplacement du curseur, on peut lire la valeur de `tag_o`, elle correspond bien à la valeur voulue (Figure 34).

T C5 EB 90 2A 13 30 70 77 47 00 39 BC 3A 69 28 EC (16 octets)

Figure 35: valeur demandée du tag_o

4.5 Chronogramme des entrées-sorties pendant tout le fonctionnement du système. :

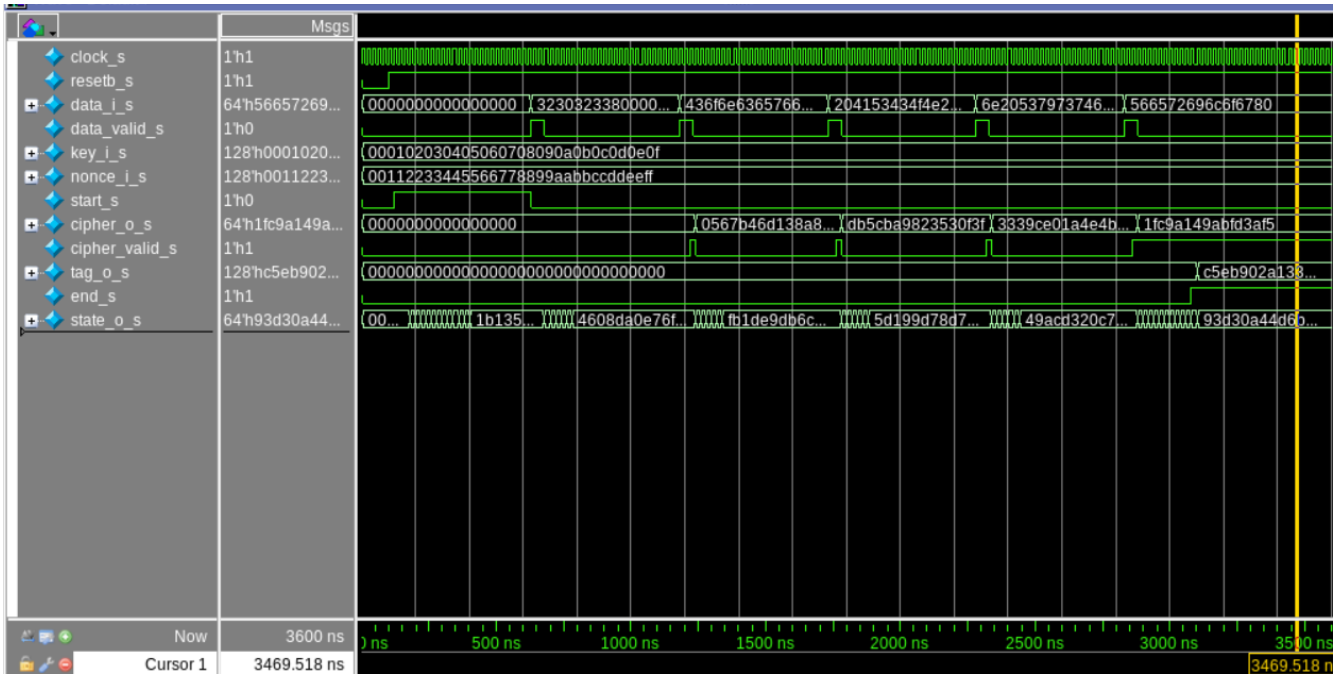

Figure 36: Vue globale du Chronogramme durant tout le fonctionnement

5 Problèmes rencontrés

Pour la mise en marche de notre permutation, nous n'avons pas rencontré beaucoup de problèmes avec sa conception, puisque nous avons vérifié dès le début le fonctionnement de ces unités à chaque étape. Cependant, concevoir un test bench valide était la partie la plus difficile.

Dans la conception de la machine d'état pour la phase d'initialisation, nous n'avons pas trouvé de grands problèmes, car nous avons bien compris le principe de fonctionnement de la permutation à cette phase. Mais au moment où nous avons voulu l'implémenter en code SystemVerilog, nous avons rencontré une erreur bizarre qui nécessite, à chaque état, que nous mettions le bloc contenant nos signaux dans un bloc qui commence par `begin` et finit par `end`. Cette erreur était pour nous invisible puisqu'il n'y a aucune partie dans le cours qui traite ce cas spécial.

En passant à la deuxième phase de donnée associée, nous avons rencontré un problème en essayant de garantir une continuité de données avec l'initialisation. Nous avons maintenu le multiplexeur sur la deuxième entrée liée au registre pour récupérer les données stockées à la sortie de l'initialisation.

Ensuite, nous initialisons notre compteur juste à l'état suivant avant de faire le XOR des données. Au début, nous inversions les deux en supposant que notre donnée d'entrée devait subir l'opération XOR avant que notre compteur ne démarre, ce qui a créé un décalage entre les données et le round de la permutation. Cette phase nous a pris beaucoup de temps, mais elle nous a beaucoup aidé à compléter notre machine d'état pour le reste des phases.

6 Conclusion

En résumé, ce projet de développement en SystemVerilog a été une opportunité concrète d'appliquer les connaissances acquises en langage de description matérielle. Nous avons réussi à mettre en œuvre le chiffrement ASCON128 en respectant scrupuleusement les spécifications du standard, en adoptant une structure en couches pour séparer les différentes fonctionnalités de l'algorithme. De plus, la mise en place d'un processus de test rigoureux a été cruciale pour garantir la qualité de notre implémentation.

Par ailleurs, nous avons exploité des outils de synthèse et de simulation afin de confirmer le bon fonctionnement de notre code sur une cible matérielle. En somme, ce projet a constitué une occasion exceptionnelle pour mettre en pratique diverses compétences techniques et plonger profondément dans le développement de circuits numériques complexes.