

Conception de système numérique
Projet ASCON128

Noé Backert

Novembre 2023



INSPIRING
INNOVATION
SINCE 1816

Table des matières

1	Introduction	3
2	Description générale du projet	3
2.1	Fonctionnement de ASCON128	3
2.2	Notations	4
2.3	Structure du code	4
2.4	Comment compiler ?	5
3	Résultat de la simulation	5
4	Tests de chaque éléments	6
4.1	Permutation	6
4.1.1	L'addition de constante p_C	6
4.1.2	La couche de substitution p_S	6
4.1.3	La couche de diffusion linéaire p_L	7
4.1.4	La permutation complète	8
4.2	Développement des permutations p^6 et p^{12}	8
4.2.1	Fonctionnement général	8
4.2.2	Développement du multiplexeur	9
4.2.3	Développement de la DFF	9
4.2.4	Test de l'initialisation	9
4.3	Ajout des XOR	10
4.3.1	Fonctionnement des XOR	10
4.3.2	Implémentation des XOR pour la partie Data Associée	10
4.3.3	Test des text clairs	11
5	Machine d'état	12
5.1	Schéma de fonctionnement	12
5.2	Description des états	13
5.3	Chronogramme de la machine d'état (FSM)	14
6	Difficultés rencontrées	14
6.1	Problème XOR	14
6.2	Génération de multiples modules rotation	14
6.3	Autres erreurs	14
7	Conclusion	15

1 Introduction

La sécurité des données dans les systèmes de communications numériques est une priorité pour rendre des conversations privées. Le projet vise à concevoir, simuler et analyser le système de cryptage ASCON128, une solution légère et efficace, utilisée surtout dans le domaine de l'IoT, où les ressources disponibles sont faibles.[1]

Objectif du Projet

L'objectif principal de ce projet est de développer une compréhension approfondie du fonctionnement du système de cryptage ASCON128 et de découvrir le langage de description SystemVerilog. Nous allons aborder la conception de sa machine d'états, simuler chaque composant et analyser les résultats pour garantir le fonctionnement du système.

Ressources

Pour aborder et tester cette conception de système électronique, nous utiliserons ModelSim sur le serveur "Tallinn" de l'école.

2 Description générale du projet

2.1 Fonctionnement de ASCON128

Le module ASCON128 est divisé en plusieurs blocs :

1. La machine d'états
2. Le bloc permutation
3. Les blocs XOR
4. Les compteurs de permutation et de blocs (cipher)

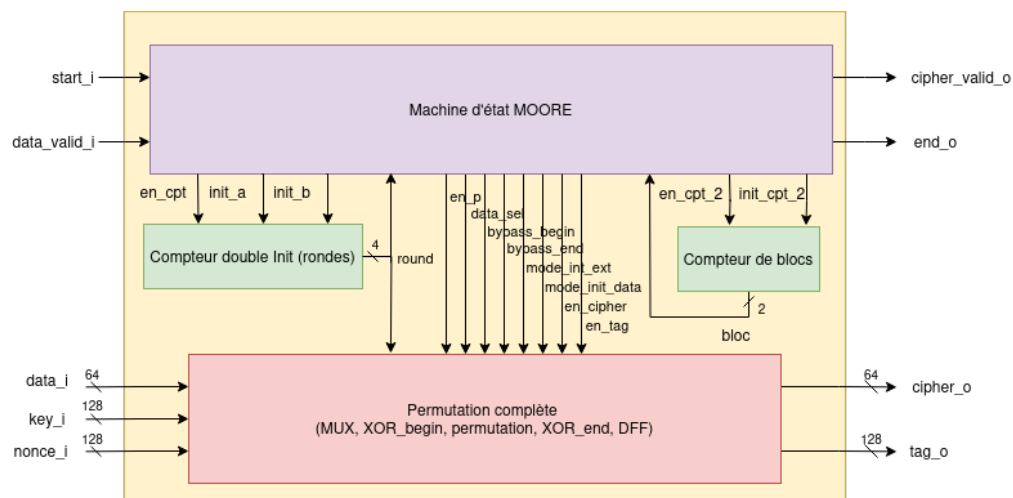


Figure 1: Schéma de fonctionnement général de la méthode de chiffrement ASCON128

Cette procédure de chiffrement se déroule en plusieurs étapes selon le schéma suivant :

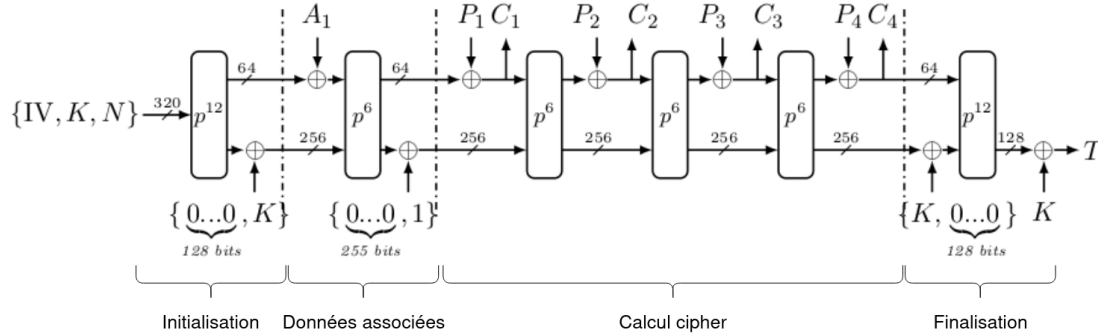


Figure 2: Schéma de fonctionnement de l'ASCON 128

Il est important de mémoriser les différentes étapes du processus pour le reste du rapport.

2.2 Notations

On rappelle les notations du sujet :

- L'algorithme opère sur un état courant (ou state S) de 320 bits (on verra qu'en pratique, on utilisera un type personnalisé nommé *type_state* défini dans le module **ascon_pack**) qui sera composé d'un vecteur de 5×64 bits que l'on nommera également $(x_0, x_1, x_2, x_3, x_4)$.
- Cet état est mis à jour avec une opération appelée permutation. La permutation comprend soit 6 itérations, soit 12 itérations, et sera noté p^6 (respectivement p^{12}).

L'état se subdivise en deux parties :

- Une partie interne de 256 bits, notée $Sc(= x_1, x_2, x_3, x_4)$
- Une partie externe de 64 bits, notée $Sr(= x_0)$

2.3 Structure du code

Lors de ce projet, et en règle générale, pour ne pas se perdre lors de la conception de systèmes numériques, il est nécessaire de séparer les codes sources des testbench et des fichiers compilés. Pour cela, nous allons utiliser l'architecture suivante :

```

ascon_project
├── compil_ascon.txt
├── DO
├── init_modelsim.txt
├── LIB
│   ├── LIB_BENCH
│   └── LIB_RTL
├── modelsim.ini
├── SRC
│   ├── BENCH
│   └── RTL
├── transcript
└── vsim.wlf

```

- Les fichiers sources sont dans le dossier ./SRC/RTL.
- Les fichiers de test sont dans le dossier ./SRC/BENCH.
- Les bibliothèques compilées et les fichiers exécutables sont dans les dossiers ./LIB/LIB_BENCH et ./LIB/LIB_RTL selon leur nature.
- Nous avons ajouté un dossier ./DO nous permettant de ranger tous nos scripts d'exécution de simulation.
- Le fichier **init_modelsim.txt** permet d'exécuter les commandes afin d'initialiser correctement le logiciel ModelSim.
- Le fichier **compil_ascon.txt** permet d'exécuter toutes les commandes de compilation en une seule commande.

2.4 Comment compiler ?

La compilation se fait par intermédiaire du script de compilation présent dans **compil_ascon.txt**. Il faut donc décommenter les testbench que l'on souhaite exécuter dans **compil_ascon.txt** puis simplement entrer les commandes suivantes dans le terminal SSH de tallinn :

```
1  bash
2  source init_modelsim.txt
3  source compil_ascon.txt
```

Puis la commande dans ModelSim :

```
1  do D0/[module.do]
```

3 Résultat de la simulation

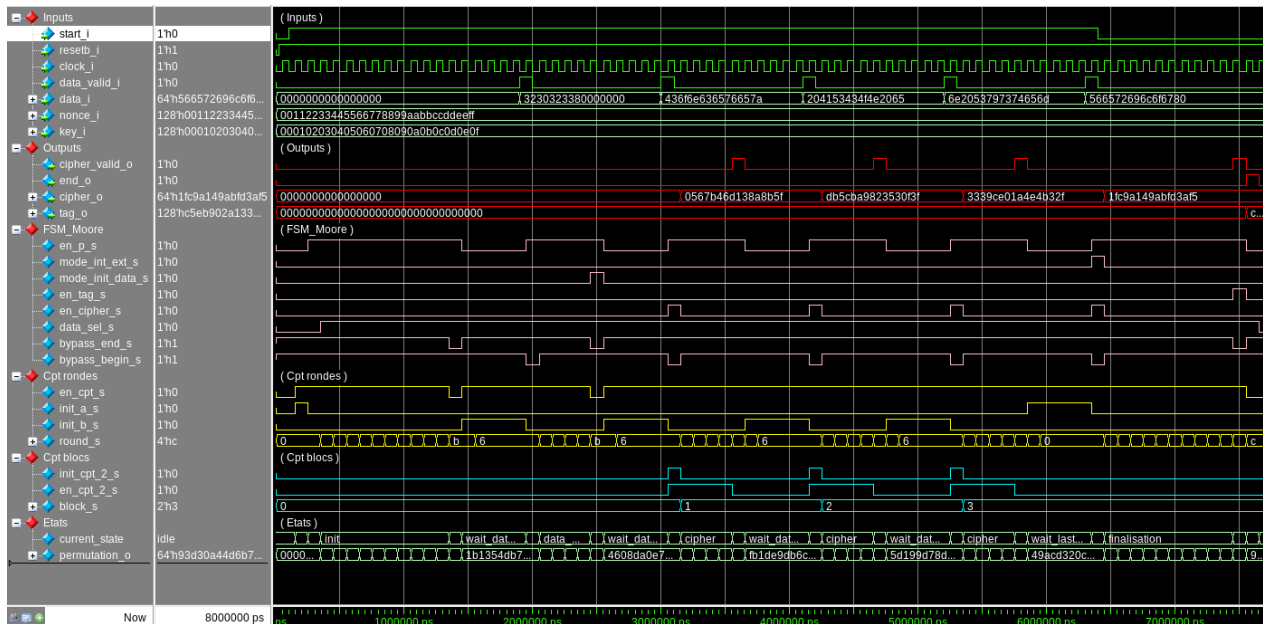


Figure 3: Résultat de la simulation complète
(NB : La qualité permet de zoomer sur le document)

On remarque à première vue que les résultats sont bons. En effet, on retrouve bien la valeur de l'exemple de validation du fonctionnement avec pour cypher :

C1 = 05 67 b4 6d 13 8a 8b 5f

C2 = db 5c ba 98 23 53 0f 3f

C3 = 33 39 ce 01 a4 e4 b3 2f

C4 = 1f c9 a1 49 ab fd 3a (f5)

Le dernier plain text n'étant que sur 7 octets au lieu de 8, on ne compte pas le dernier octet (f5) de C4.

De plus, on peut également voir que le tag final correspond bien à celui de validation :

T = C5 EB 90 2A 13 30 70 77 47 00 39 BC 3A 69 28 EC

4 Tests de chaque éléments

4.1 Permutation

4.1.1 L'addition de constante p_C

L'addition de constante se fera à l'aide d'un tableau de constante défini dans le package **ascon_pack**:

```
1 localparam 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};
```

/pc_tb/round_s	4'h0
/pc_tb/pcin_s	64'h80400c0600000000 64'h0001020304050607 64'h080...
[0]	64'h80400c0600000000
[1]	64'h0001020304050607
[2]	64'h08090A0B0C0D0E0F
[3]	64'h0011223344556677
[4]	64'h8899AABBCCDDEEFF
/pc_tb/pcout_s	64'h80400c0600000000 64'h0001020304050607 64'h080...
[0]	64'h80400c0600000000
[1]	64'h0001020304050607
[2]	64'h08090A0B0C0D0EFF
[3]	64'h0011223344556677
[4]	64'h8899AABBCCDDEEFF

Figure 4: Résultat de la couche d'addition de constante de la permutation

On remarque sur la figure 4 que cette couche fonctionne correctement, puisque l'addition a bien été effectuée. Il faudra vérifier par la suite que celle-ci fonctionne également pour les autres rondes, pour ne pas avoir de soucis plus tard.

4.1.2 La couche de substitution p_S

Pour une question de simplicité, j'ai décidé de diviser la de substitution p_S en deux modules :

- *sbox* qui permettra de retourner la valeur des 5 bits correspondants au résultat du tableau, suite à la concaténation des vecteurs $x_0(i), x_1(i), x_2(i), x_3(i), x_4(i)$ selon le tableau 3 du sujet. Ce module devra donc être construit à partir d'un switch-case de 2^5 valeurs = 32 cas.

/sbox_tb/sbox_in	5'h14
/sbox_tb/sbox_out	5'h00

Figure 5: Résultat de *sbox* avec une entrée de 0x14

- *ps* permet de concaténer les différentes valeurs trouvées par la table de couche de substitution pour tous les bits de l'état (de 0 à 63).

/ps_tb/psin_s	64'h80400c0600000000 64'h00...
[0]	64'h80400c0600000000
[1]	64'h0001020304050607
[2]	64'h08090A0B0C0D0EFF
[3]	64'h0011223344556677
[4]	64'h8899AABBCCDDEEFF
/ps_tb/psout_s	64'h8859263f4c5d6e8f 64'h00c...
[0]	64'h8859263f4c5d6e8f
[1]	64'h00c18e8584858607
[2]	64'h7f7f7f7f7f7f7f8f
[3]	64'h80c0848680808070
[4]	64'h08888888A8888888

Figure 6: Résultat de *ps* avec l'entrée issue des données de validations

Ainsi, on aura l'état de sortie de p_s : $S_{out}(i) = \{x_0(i), x_1(i), x_2(i), x_3(i), x_4(i)\}$

4.1.3 La couche de diffusion linéaire p_L

Pour la couche de diffusion linéaire, j'ai décidé à nouveau de séparer l'opération en deux modules :

- Un module **rotation** permettant d'effectuer les rotations cycliques nécessaires au fonctionnement de la couche. Ce module prendra alors un argument N , qui représentera le nombre de bits à décaler à droite.





  /rotation_tb/roti_s	64'b1000000001000000000011000000001100000000000000000000000000000000
  /rotation_tb/roto_s	64'b000000000000000000001000000000100000000001100000000110000000000000

Figure 7: Exemple de rotation avec $N = 19$ et entrée à 0x80400c0600000000

- Un module **pl** qui effectuera alors toutes les opérations pour chaque état x_i selon les consignes suivantes issues de la figure 8.

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

Figure 8: Opérations de diffusion linéaire






















  /pl_tb/plin_s	64'h8859263f4c5d6e8f 64'h00c18e8584858607 64'h7f7f...
  [0]	64'h8859263f4c5d6e8f
  [1]	64'h00c18e8584858607
  [2]	64'h7f7f7f7f7f7f7f8f
  [3]	64'h80c0848680808070
  [4]	64'h8888888a88888888
  /pl_tb/plout_s	64'he05e3fced08e4f0 64'h0dc4f1a5aea83522 64'hfd3d...
  [0]	64'he05e3fced08e4f0
  [1]	64'h0dc4f1a5aea83522
  [2]	64'hfd3d3d3d3d3d3db6
  [3]	64'hdc8f4c7e363e010
  [4]	64'hdcddddd9d9d9d9d

Figure 9: Test de la diffusion linéaire avec l'entrée de validation

4.1.4 La permutation complète

Une fois les 3 modules effectués, la permutation complète, n'est que la concaténation des 3 modules précédents.














 /p_tb/pcin_s	64'h80400c0600000000 64'h00010...
 [0]	64'h80400C0600000000
 [1]	64'h0001020304050607
 [2]	64'h08090A0B0C0D0E0F
 [3]	64'h0011223344556677
 [4]	64'h8899AABBCCDDEEFF
 /p_tb/pcout_s	64'he05e3fced08e4f0 64'h0dc4f1a...
 [0]	64'hE05E3FCCED08E4F0
 [1]	64'h0DC4F1A5AEA83522
 [2]	64'hFD3D3D3D3D3D3DB6
 [3]	64'hDCD8F4C7E363E010
 [4]	64'hDCDDDDDFD9DDDDDD
 /p_tb/round_s	4'h0

Figure 10: Test de la permutation complète avec l'entrée de validation

4.2 Développement des permutations p^6 et p^{12}

4.2.1 Fonctionnement général

Pour pouvoir effectuer un certain nombre de fois la permutation, nous aurons besoin d'un compteur, qui gèrera la variable *round*, et pour la suite, d'un compteur de blocs (pour les calculs des textes clairs). Voici le schéma de fonctionnement général, avec les entrées contrôlées par la machine d'état décrite auparavant.

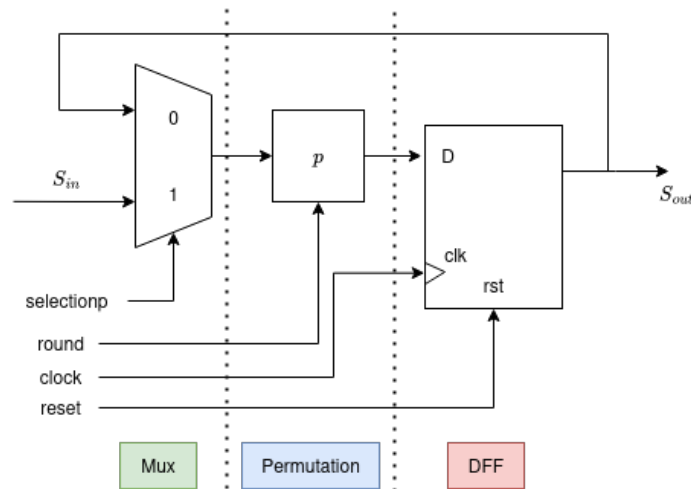


Figure 11: Schéma de fonctionnement général de la permutation

Cette architecture est importante pour le développement d'une permutation car tous les composants sont nécessaires. En effet, multiplexeur (mux) permet de laisser passer la première entrée lors du commencement de l'algorithme, puis de choisir durant tous le processus restant la sortie de la D flip-flop, qui permet de garder en mémoire l'état et de le retourner au prochain coup d'horloge pour progresser dans l'algorithme de traitement des cipher. Par la suite, nous intégrerons les XOR nécessaires au bon déroulement du reste du programme.

4.2.2 Développement du multiplexeur

Le développement du mux est facile, il suffit de faire correspondre la sortie en fonction de la valeur de *selection_s*. dans notre cas, on retourne l'entrée si *selection_s* = 0 et la sortie de la permutation si *selection_s* = 1

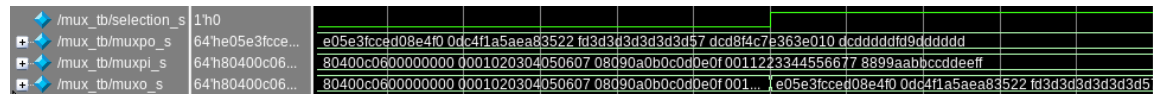


Figure 12: Test Bench du multiplexeur en entrée de la permutation

4.2.3 Développement de la DFF

Le rôle de la DFF est de maintenir l'état, jusqu'à la permutation suivante. On peut vérifier le fonctionnement avec le test suivant :

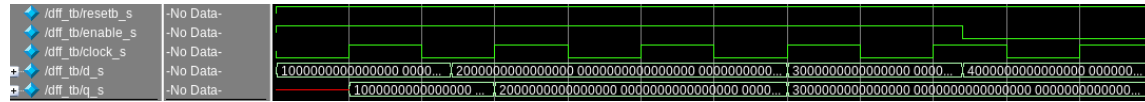


Figure 13: Test de la DFF

On observe bien le fonctionnement de la DFF, puisque lorsque *enable* = 1, on observe le maintiens de $q_s[i] = d_s[i]$ en suivant les ticks d'horloge. Et lorsque *enable* = 0, $q_s[i] = q_s[i - 1]$

4.2.4 Test de l'initialisation

En cumulant l'ensemble de ces blocs, selon le schéma de fonctionnement Figure 11, on peut dès à présent tester le bloc complet de l'initialisation décrit dans le schéma Figure 2.

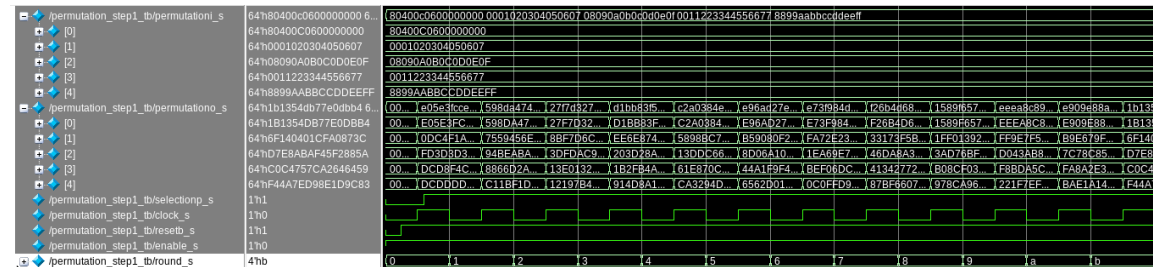


Figure 14: Test de l'initialisation

On peut en conclure le fonctionnement de ce bloc test avec la valeur de la dernière permutation 1b1354db77e0dbb4 6f140401cfa0873c d7e8abaf45f2885a c0c4757ca2646459 f44a7ed98e1d9c83

4.3 Ajout des XOR

4.3.1 Fonctionnement des XOR

Il y a 4 types de XOR : (on rappelle que $S = \{S_r, S_c\} = \{x_0, x_1, x_2, x_3, x_4\}$)

- $S_r \oplus A_i$
- $S_c \oplus \{0_{64}, 0_{64}, K[127 : 64], K[63 : 0]\}$
- $S_c \oplus \{0_{64}, 0_{64}, 0_{64}, 0_{63}, 1\}$
- $S_c \oplus \{K[127 : 64], K[63 : 0], 0_{64}, 0_{64}\}$

De plus, ils sont appliqués à différents endroits de l'algorithme. J'ai donc choisi de créer deux modules XOR distincts, chacun avec son propre mode de fonctionnement.

- XOR_begin : Correspondent aux XOR présents avant des permutations (première itération)
 - $\text{mode_int_ext} \in \{0, 1\}$: Mode pour sélectionner si le XOR correspond à celui à l'intérieur des texts clairs ou à l'extérieur (voir Figure 2).
 - * 0 : $S_r \oplus A_i$
 - * 1 : $S_c \oplus \{K[127 : 64], K[63 : 0], 0_{64}, 0_{64}\}$
- XOR_end : Correspondent aux XOR présents après des permutations (dernière itération)
 - $\text{mode_init_data} \in \{0, 1\}$: Mode pour sélectionner si le XOR correspond à celui à l'initialisation ou à celui de la phase de data associée
 - * 0 : $S_c \oplus \{0_{64}, 0_{64}, K[127 : 64], K[63 : 0]\}$
 - * 1 : $S_c \oplus \{0_{64}, 0_{64}, 0_{64}, 0_{63}, 1\}$

En réalité, pour simplifier l'utilisation unique du XOR_begin avec $\text{mode_int_ext} = 0$, j'ai utilisé un XOR de type : $S \oplus \{A_i, K[127 : 64], K[63 : 0], 0_{64}, 0_{64}\}$ (voir Problème dans la section 6.1).

4.3.2 Implémentation des XOR pour la partie Data Associée

Pour avancer dans l'algorithme, il faut alors implémenter les deux types de XOR qui se trouvent dans l'algorithme.

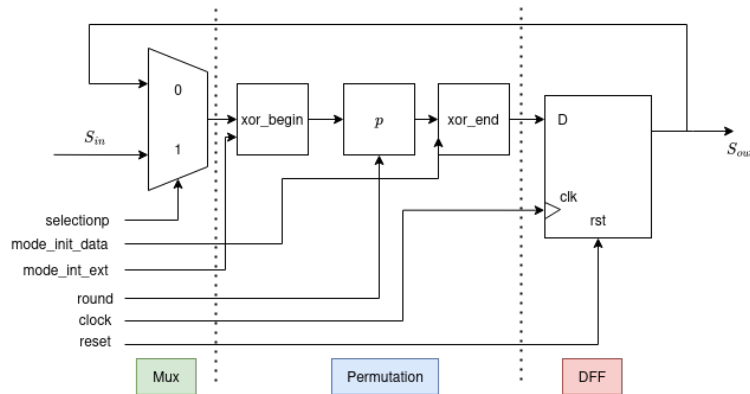


Figure 15: Schéma de fonctionnement des permutations avec XOR

On les ajoute donc dans le module **permutation_step2bis.sv** qui nous permet de tester leurs fonctionnements dans la partie Data Associée de la figure 2. **permutation_step2bis.sv** est une

version plus avancée de **permutation_step2b**.sv qui effectue la même chose en prenant cette fois-ci en compte le XOR_end de la fin de Data Associée.

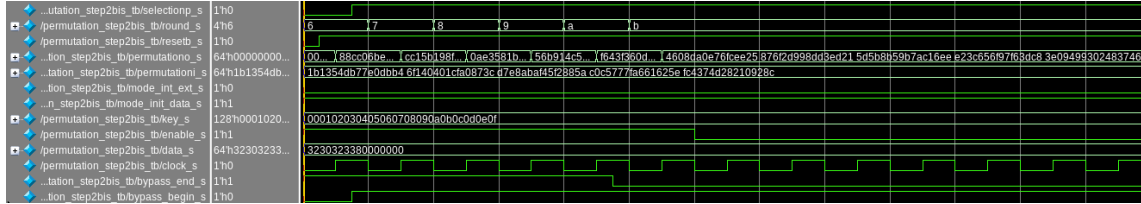


Figure 16: Résultat de simulation de Data Associée en ayant ajouté les XOR

Ainsi, on remarque qu'on peut aller plus loin, mais que pour cela il faudra bientôt une machine d'état afin de dire à quel endroit du système on se trouve.

4.3.3 Test des text clairs

On peut maintenant tester l'étape suivante avec le test de génération des ciphers

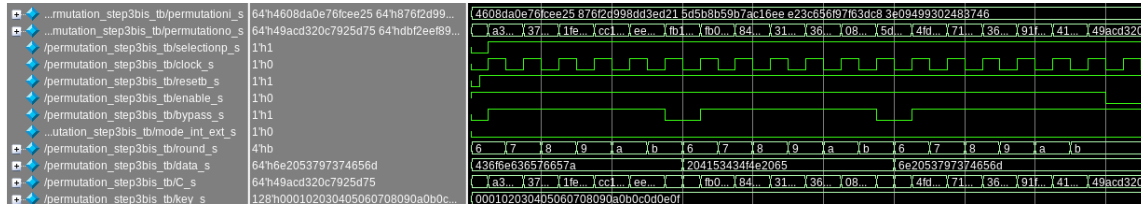


Figure 17: Test bench de permutation_step3bis

permutation_step3bis_tb.sv est simplement l'amélioration (calcul des 3 textes clairs) de **permutation_step3**.tb.sv, passant de 1 calcul de cipher aux 3 premiers, dans un but de test.

On peut remarquer dans la simulation **permutation_step3bis** que les cipher (valeurs de C_s) obtenus après la dernière itération de permutation est bien égale au cipher des données de validations, nous confortant pour la suite.

Nous pouvons donc passer à l'ajout des compteurs et à l'intégration de la machine d'état.

5 Machine d'état

5.1 Schéma de fonctionnement

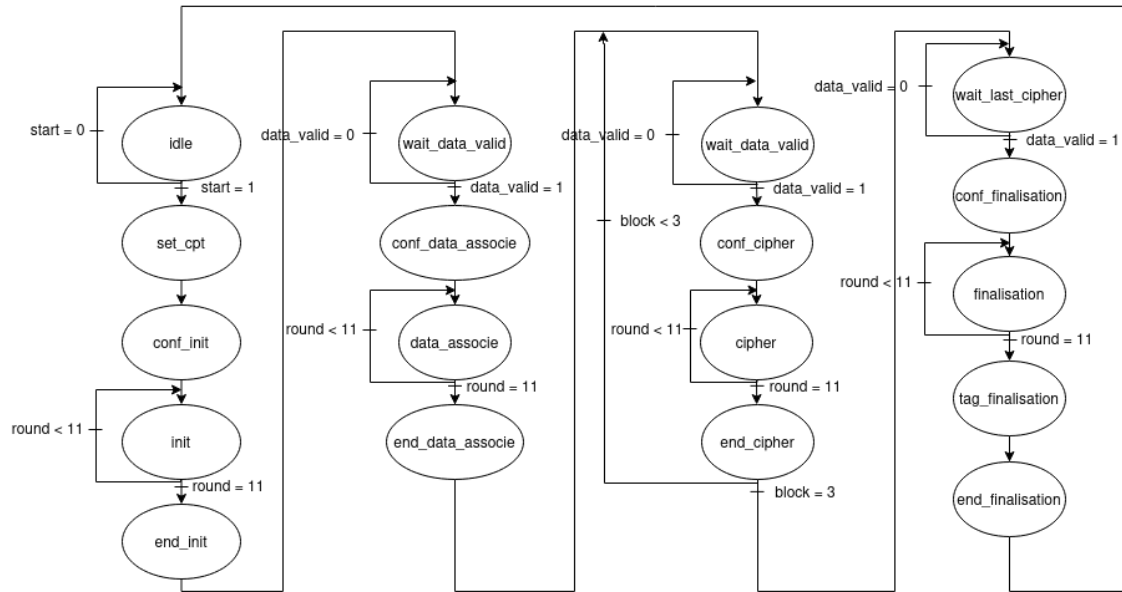


Figure 18: Diagramme d'état de la méthode de chiffrement ASCON128

Ainsi, les états doivent contrôler les différentes variables de contrôles utilisées lors de la permutation. (cf Figure 1)

- en_p
- data_sel
- bypass_begin
- bypass_end
- mode_int_ext
- mode_init_data
- en_cipher
- en_tag

De plus, la machine d'état contrôle également les compteurs, donc les variables suivantes :

- en_cpt
- init_a
- init_b
- en_cpt_2
- init_cpt_2

Puis les deux dernières variables contrôlées par la machine d'état sont ses sorties :

- cipher_valid
- end

5.2 Description des états

- **idle** : maintient toutes les variables à 0, sauf les bypass des fonctions XOR et attend le start pour continuer.
- **set_cpt** : Initialise le compteur de rondes à 0 (pour compter jusqu'à 11, donc 12 permutations)
- **conf_init** : Début des permutations lors du prochain coup d'horloge.
- **init** : Permutations tant qu'on a pas $round_s = 11$.
- **end_init** : Dernière permutation avec $bypass_end = 0$ pour activer le XOR de sortie de la partie initialisation.
- **wait_datavalid** : Attend que le signal *data_valid* en entrée passe à 1. Passe à l'état suivant qui a été configuré (ici *conf_data_associe*) lors du prochain coup d'horloge si la condition est vérifiée. Initialise également le compteur de rondes à 6 (pour n'avoir que $12-6 = 6$ permutations lorsqu'on le démarrera).
- **conf_data_associe** : Démarre les permutations avec le bypass du XOR_begin à 0 pour effectuer le XOR en mode init (donc $mode_init_data = 0$).
- **data_associe** : Effectue les permutations tant que $round \neq 11$.
- **end_data_associe** : Configure la dernière permutation avec le XOR_end. Donc mets le $bypass_end$ à 0 et $mode_init_data$ à 1. (Puisqu'on est encore dans l'initialisation)

A ce moment là, on entre dans une boucle gérée par le compteur de bloc. On a ainsi, tant que $bloc \neq 3$ faire:

- **wait_data_valid** : Même état que décrit précédemment avec pour retour d'état cette fois-ci l'état suivant : *conf_cipher*.
- **conf_cipher** : Première permutation, donc XOR_begin à 1, début du compteur de ronde.
- **cipher** : Comptage des rondes, jusqu'à ce que $round = 11$.
- **end_cipher** : Validation du cipher.

Ensuite, une fois que $bloc = 3$:

- **wait_last_cipher** : J'ai choisi de ne pas utiliser *wait_data_valid*, pour faire l'initialisation des rondes directement lors de l'attente du dernier cipher. En effet, lors des prochaines permutations, il ne faudra plus effectuer p^6 mais bien p^{12} . Donc setup de $round = 0$ pour bien compter jusqu'à 12.
- **conf_finalisation** : Première permutation avec XOR_begin et le *mode_int_end* d'activé (1) pour effectuer le XOR avec la clé.
- **finalisation** : Permutation jusqu'à 12.
- **tag_finalisation** : Dernière permutation avec calcul du tag durant le XOR_end avec la clé.
- **end** : Etat de fin, retour de la variable *end*, en sortie, pour indiquer que l'algo a fini de calculer les cyphertext et retour à l'état d'attente **idle**

5.3 Chronogramme de la machine d'état (FSM)

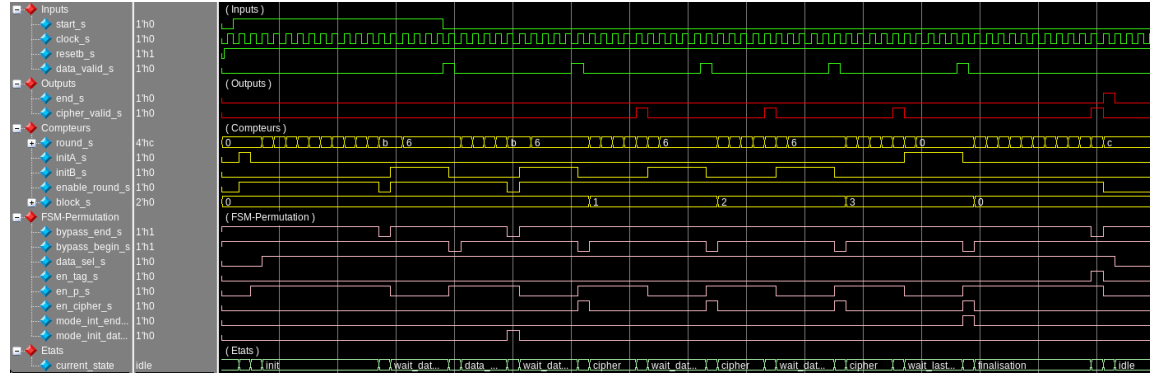


Figure 19: Chronogramme de la machine d'état

6 Difficultés rencontrées

6.1 Problème XOR

Lors de la finalisation du programme, j'ai eu du mal à trouver comment, sans ajouter de permutation supplémentaire, effectuer à la fois le dernier XOR_begin avec les le plain text 4 : P_4 et à la fois le XOR_begin avec la clé avant d'entrer dans la finalisation.

Pour remédier à ce problème, j'ai ajouté au XOR_begin ($\text{mode_int_ext} = 1$) qui n'est utilisé qu'une fois le XOR de la donnée.

Le XOR résultant est donc : $S \oplus \{A_i, K[127 : 64], K[63 : 0], 0_{64}, 0_{64}\}$. Cela résout le problème et permet de ne pas introduire un état supplémentaire afin de choisir le nombre de permutations avant d'effectuer p^{12} .

6.2 Génération de multiples modules rotation

Un autre problème où je me suis heurté était dans le début, dans la réalisation de p_l . En effet, l'idée de construire un module intermédiaire **rotation** était dans l'intention de faire une boucle pour déclarer les modules.

Il s'avère qu'il est impossible de créer des modules avec un paramètre différent dans un **generate**. Cela s'est donc avéré contre productif de créer $5 * 2 = 10$ modules (1 pour chaque type de rotation cyclique).

Une simple concaténation dans p_L aurait pu faire l'affaire et aurait même été plus efficace.

6.3 Autres erreurs

Les autres erreurs qui m'ont coûté du temps, ont surtout été les erreurs initiales d'attention, comme dans le tableau de **switch-case** de la sbx, où des valeurs erronées étaient présentes. Je n'avais pas décelé l'erreur immédiatement puisque l'erreur ne se propageait pas beaucoup, et je ne faisais le test en général que sur les premières itérations de permutation, en ne regardant que les premiers octets.

Cette erreur, me donnait donc plus loin des valeurs d'états totalement erronées dû à l'effet avalanche recherché des fonctions cryptographiques.

J'ai donc du retester les blocs du début 1 par 1 en regardant tous les résultats pour trouver qu'au final, c'était une faute de frappe dans la sbbox.

7 Conclusion

En résumé, ce projet sur l'ASCON128 a été une expérience intéressante. Nous avons réussi à mettre en place tous les éléments nécessaires, de la machine d'états aux permutations, et les résultats de la simulation (voir figure 3) montrent que le chiffrement fonctionne correctement.

Bien que le projet ait abouti à des résultats positifs, il reste des aspects à améliorer. Des pistes d'optimisation des performances, des explorations de variantes de sécurité renforcée et des adaptations pour d'autres environnements spécifiques pourraient constituer des axes de développement futurs.

Par exemple, l'implémentation d'une machine d'état de Mealy à la place d'une machine d'état de Moore, pourrait améliorer les performances du système électronique en temps. Cependant, cette implémentation peut également introduire une complexité accrue inutile dans la gestion des transitions d'état et des sorties.

En conclusion, ce projet m'a permis d'acquérir des connaissances approfondies dans le domaine de la conception de systèmes numériques et ouvre la voix à notre cours de cryptographie.

References

- [1] NIST (National Institute of Standards and Technology). Nist selects 'lightweight cryptography' algorithms to protect small devices, Feb 2023.