Rapport de Conception Technique

Conception de Système Numérique

Modélisation SystemVerilog de l'algorithme de chiffrement ASCON

GP - Électronique 1

Réalisé par :
Prusothman VIGNESWARAN
p.vigneswaran@etu.emse.fr

Encadré par :
Olivier POTTIN

8 mai 2025

École Nationale Supérieure des Mines de Saint-Étienne

Table des matières

1 Introduction **2**

2 Principe de fonctionnement de l'algorithme ASCON-AEAD128 **2**
2.1 État S . . . 2
2.2 Notations . . . 3
2.3 Initialisation . . . 4
2.4 Traitement des données associées . . . 4
2.5 Traitement du texte clair . . . 5
2.6 Finalisation et calcul du tag . . . 5

3 Architecture globale de l'algorithme **7**
3.1 Organisation générale . . . 7
3.2 Schéma bloc . . . 7

4 Modélisation et implémentation du bloc `Permutation + XOR` **8**
4.1 Organisation de la permutation . . . 8
4.2 Module P_C . . . 9
4.3 Module P_S . . . 10
4.4 Module P_L . . . 11
4.5 Module `Permutation` . . . 12
4.6 Modules `Xor_Begin` et `Xor_End` . . . 13
4.7 Module `Permutation_XOR` . . . 16
4.8 Registres de sortie : `Cipher` et `Tag` . . . 17
4.9 Module `Permutation_XOR_Final` . . . 17

5 Modélisation de la machine de contrôle (FSM) **18**
5.1 Organisation et justification de la structure de la FSM . . . 18
5.2 Diagramme d'état de la FSM . . . 19
5.3 Table de vérité des signaux de commande . . . 20
5.4 Simulation de la FSM . . . 22

6 Implémentation du système : `ascon_top` **22**

7 Difficultés rencontrées **24**

8 Conclusion **26**

1 Introduction

Dans un contexte où les échanges numériques sont omniprésents, la question de la sécurité des communications devient essentielle. Que ce soit pour protéger la vie privée ou assurer l'intégrité des données, les mécanismes cryptographiques sont au cœur des architectures modernes. Le chiffrement symétrique, et plus spécifiquement le chiffrement authentifié avec données associées (AEAD), permet non seulement de garantir la confidentialité des messages, mais aussi d'assurer leur authenticité.

C'est dans ce cadre qu'intervient l'algorithme ASCON128, conçu pour offrir un bon compromis entre sécurité, efficacité et simplicité matérielle. Cet algorithme est aujourd'hui recommandé dans de nombreux contextes, notamment les communications sécurisées à faible empreinte.

Ce rapport s'inscrit dans un projet de modélisation de l'algorithme ASCON128 à l'aide du langage de description matérielle SystemVerilog. L'objectif est de concevoir une architecture matérielle capable de reproduire les principales fonctionnalités de l'algorithme, en mettant l'accent sur la clarté, la modularité et la vérifiabilité de chaque composant.

Nous détaillerons dans un premier temps les principes du chiffrement ASCON128, avant de présenter l'architecture proposée, les différents modules développés, ainsi que les résultats obtenus via simulation. Enfin, une analyse des difficultés rencontrées et des perspectives d'amélioration conclura ce travail.

2 Principe de fonctionnement de l'algorithme ASCON-AEAD128

L'algorithme ASCON-AEAD128 repose sur le paradigme du chiffrement authentifié avec données associées. Il permet à la fois de garantir la confidentialité d'un message et de vérifier son authenticité à l'aide d'un tag. Sa structure s'inspire de la fonction cryptographique *éponge* et repose sur une permutation de 320 bits, divisée en cinq registres de 64 bits chacun.

L'algorithme se compose de quatre grandes phases : **initialisation**, **traitement des données associées**, **chiffrement du texte clair**, et **finalisation**. Ces étapes sont représentées dans la Figure 3, tandis que l'algorithme 1 en formalise l'enchaînement.

2.1 État S

L'algorithme ASCON repose sur un état interne noté S, de taille fixe 320 bits, qui constitue le cœur de toutes les opérations cryptographiques. Cet état est modifié à chaque itération des phases de permutation.

L'état S est structuré comme une concaténation de **5 registres** notés S_0, S_1, S_2, S_3, S_4, chacun contenant **64 bits**, comme illustré à la Figure 1. Le registre S_0 contient les octets de poids faible, tandis que S_4 contient les octets de poids fort.

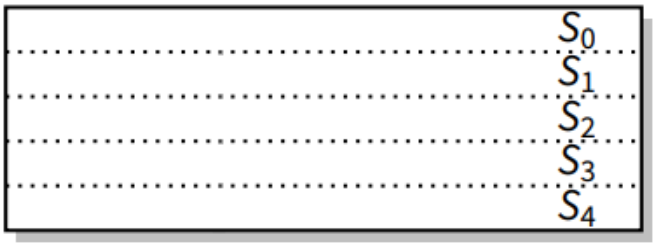

FIGURE 1 – Représentation de l'état S en cinq registres de 64 bits

Pour certaines opérations, S est logiquement divisé en deux sous-parties :
— une **partie externe** notée $S_r = \{S_0, S_1\}$, de 128 bits,
— une **partie interne** notée $S_c = \{S_2, S_3, S_4\}$, de 192 bits.
Cette séparation permet de distinguer les zones modifiées par les données externes (clé, nonce, texte clair) et celles plus internes au fonctionnement de la permutation.

Enfin, une autre représentation utile de S est sous forme de **64 colonnes** de **5 bits** chacune, formées en prenant verticalement un bit dans chaque registre S_i, vue utilisée dans la S-box.

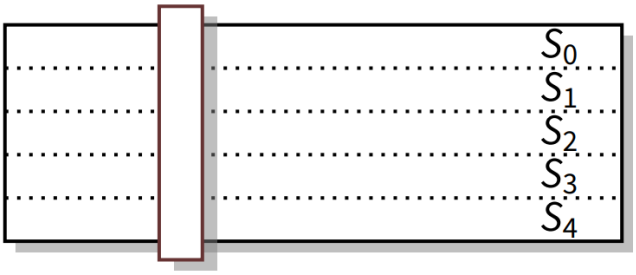

FIGURE 2 – Représentation de l'état S sous forme de colonnes

2.2 Notations

La Table 1 décrit les notations utilisées dans ce document.

K	Clé secrète *key* K de 16 octets
N	Nombre arbitraire *nonce* N de 16 octets
T	*Tag* T de 16 octets
P	Texte clair *plaintext* P de 47 octets
C	Texte chiffré *ciphertext* C de 47 octets
A	Données associées *associated data* A de 12 octets
IV	Vecteur d'initialisation *initialization vector* IV de 8 octets : `0x00001000808c0001`

TABLE 1 – Notations utilisées

Dans l'ensemble de ce document, l'opérateur $\oplus$ est à interpréter au sens d'un OU exclusif (XOR).

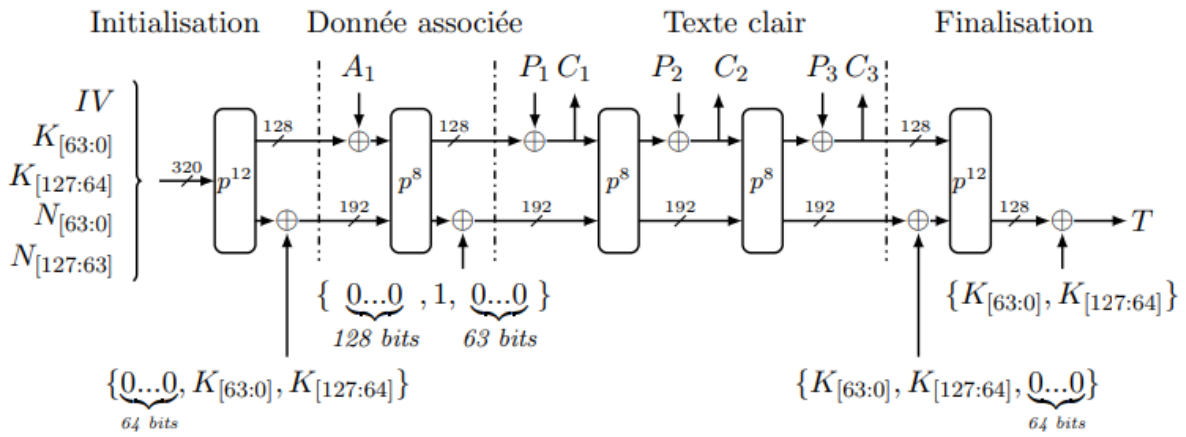

FIGURE 3 – Schéma du chiffrement Ascon-AEAD128

2.3 Initialisation

L'état S est initialisé à partir du vecteur d'initialisation IV, de la clé K (128 bits) et d'un nonce N (128 bits). L'état est ensuite transformé via la permutation p^{12} (12 rondes), puis mélangé à la clé via un opérateur XOR, afin de sécuriser l'état avant les phases de traitement.

Une **ronde** correspond à une transformation complète de l'état S, constituée de trois étapes successives :

— l'ajout d'une constante à S_2 (P_C),
— une substitution non-linéaire sur chaque colonne (P_S),
— une diffusion linéaire par rotation et XOR (P_L).

Le nombre de rondes appliquées dépend de la phase de l'algorithme : p^{12} pour l'initialisation et la finalisation, et p^8 pour les phases intermédiaires.

2.4 Traitement des données associées

Les données associées A sont tout d'abord soumises à une opération de *padding*, c'est-à-dire un ajout contrôlé de bits pour adapter la donnée à la taille attendue par le système. Dans le cas de ASCON, ce padding consiste à préfixer A par le mot `0x00000001`, ce qui permet de former un bloc A_1 de 128 bits, conforme au format requis.

Ce bloc est ensuite combiné à l'état interne à l'aide d'une opération `Xor_Begin`, qui correspond à une opération de XOR appliquée sur la partie externe de l'état S (les registres S_0 et S_1). Elle permet d'intégrer les données utiles à cette portion visible de l'état. Cette étape est suivie d'une permutation p^8 (8 rondes).

Enfin, une opération `Xor_End` est appliquée, cette fois sur la partie interne de l'état (S_2, S_3, S_4). Les deux opérations `Xor_Begin` et `Xor_End` consistent donc toutes deux en des XORs sélectifs sur des portions différentes de l'état.

2.5 Traitement du texte clair

Le texte clair P est divisé (après padding) en trois blocs P_1, P_2, et P_3, chacun de 128 bits. Pour chaque bloc, une opération XOR est appliquée entre S_r (la partie externe de l'état) et P_i, ce qui donne un bloc chiffré C_i. Une permutation p^8 est ensuite appliquée après chaque bloc sauf le dernier. Le dernier bloc chiffré, C_3, ne contient que les 120 bits les plus significatifs de S_r.

2.6 Finalisation et calcul du tag

La finalisation consiste à appliquer une opération `Xor_End` sur l'état avec un mot construit à partir de la clé et d'un padding, puis à appliquer la permutation p^{12} une dernière fois. Le tag T, servant à authentifier le message, est obtenu en XORant deux registres de l'état final avec la clé K.

Ainsi, l'algorithme retourne le texte chiffré complet $\{C_3, C_2, C_1\}$ et le tag T.

Données associées, texte chiffré et tag : rôle et intérêt
Les **données associées** représentent des informations transmises en clair (non chiffrées), mais dont l'authenticité doit être vérifiée. Il peut s'agir, par exemple, d'en-têtes de protocoles ou de métadonnées critiques. Le **texte chiffré** (`cipher`) est la version confidentielle du message initial, rendue illisible sans la clé. Enfin, le **tag d'authentification** (`T`) permet au destinataire de s'assurer que le message et les données associées n'ont pas été altérés durant la transmission.

Ainsi, `cipher` protège la confidentialité du message, tandis que `tag` garantit son intégrité et son authenticité.

Remarque sur l'ordre des opérations dans l'implémentation

Contrairement au schéma de l'algorithme ASCON-AEAD128 présenté dans la Figure 3, l'implémentation matérielle adoptée dans ce projet inverse volontairement l'ordre des opérations `Xor_Begin` et `Xor_End` dans la phase de finalisation.

En effet, selon le schéma standard, le bloc P_3 est injecté via un `Xor_Begin` dans la pahse Texte clair, tandis que le premier `Xor_End` de Finalisation intervient après la fin de cette phase. Dans notre implémentation, nous avons choisi de déplacer le `Xor_Begin` avec P_3 dans la phase de Finalisation, et de faire précéder la dernière permutation par un `Xor_End`.

Ce changement d'ordre a été motivé par des considérations de simplification de la machine à états finis (FSM). En procédant ainsi, la structure logique des phases de chiffrement du texte clair et de finalisation devient identique à celle du traitement des données associées : un `Xor_Begin`, suivi d'un `Xor_End`, encadrant une permutation p^8.

Cette uniformisation du comportement permet de réutiliser plus efficacement la logique de contrôle et de rendre la FSM plus rapide à concevoir et à valider, tout en conservant la validité fonctionnelle de l'algorithme.

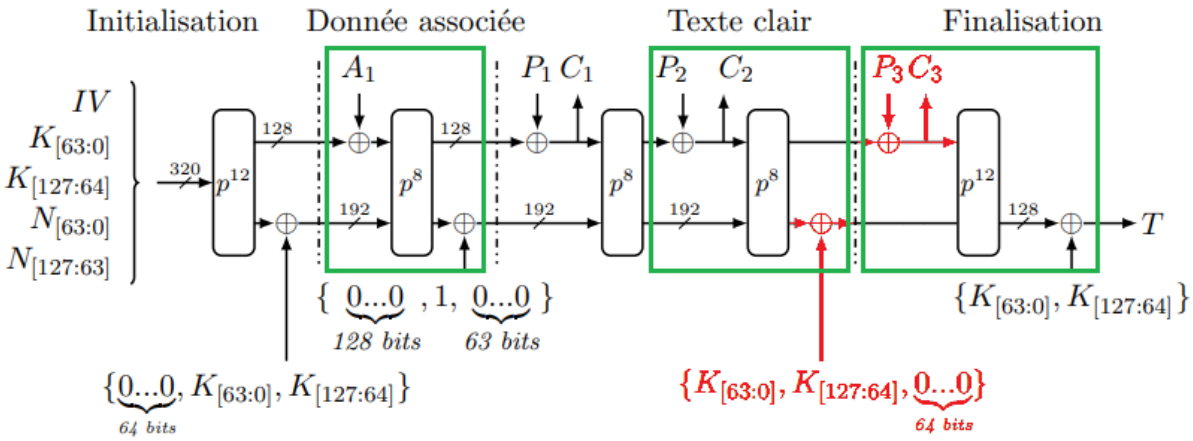

FIGURE 4 – Modification du schéma du chiffrement Ascon-AEAD128

3 Architecture globale de l'algorithme

L'implémentation matérielle de l'algorithme ASCON-AEAD128 repose sur une architecture modulaire, organisée autour de plusieurs blocs fonctionnels principaux. Cette architecture permet de suivre rigoureusement les différentes étapes de l'algorithme de chiffrement, tout en garantissant une exécution séquentielle contrôlée via une machine à états finis (FSM).

3.1 Organisation générale

L'architecture est constituée des éléments suivants :

- **Une machine à états finis (FSM)** qui orchestre les différentes phases de l'algorithme : initialisation, absorption des données associées, chiffrement du texte clair, et finalisation. Elle pilote l'activation des blocs et la synchronisation des signaux.
- **Un module de permutation (`Permutation_Xor`)**, chargé d'appliquer successivement les couches P_C, P_S et P_L à l'état courant S. Ce module inclut également deux opérateurs XOR, un avant la permutation (`Xor_Begin`) et un après (`Xor_End`), permettant d'intégrer les données à l'état ou de produire les sorties.
- **Deux compteurs** :
  - Un compteur de rondes (4 bits) pour suivre le nombre d'itérations dans les permutations p8 ou p12.
  - Un compteur de blocs (2 bits) pour gérer l'avancement dans les blocs de texte clair à chiffrer.
- **Des registres de stockage** pour l'état S (320 bits), les blocs de données d'entrée, les blocs chiffrés C_i, et le tag final T.

3.2 Schéma bloc

La Figure 5 illustre l'organisation de l'architecture numérique telle que décrite ci-dessus. Certains éléments secondaires (signaux de contrôle, registre de sortie) peuvent ne pas apparaître explicitement.

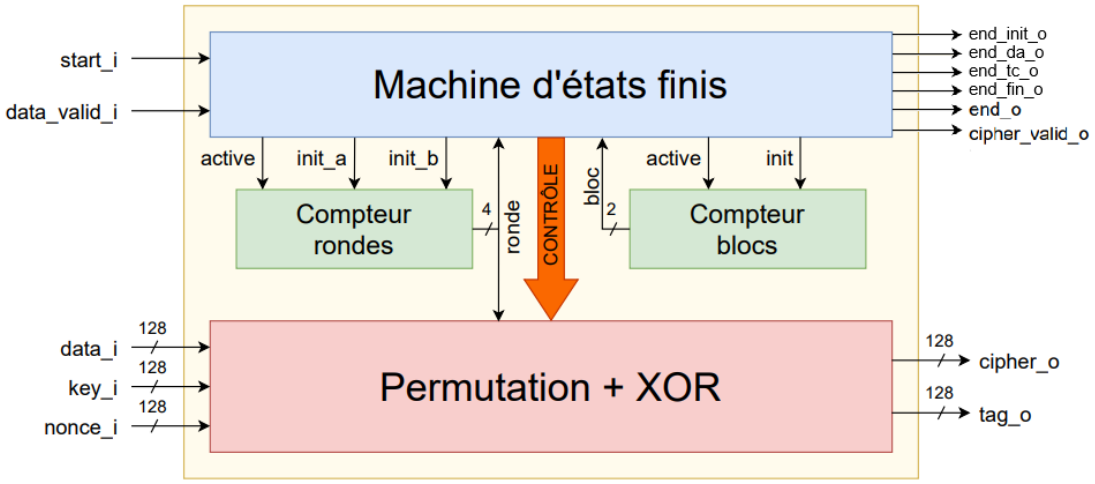

FIGURE 5 – Schéma du chiffrement Ascon-AEAD128

4 Modélisation et implémentation du bloc `Permutation + XOR`

L'implémentation de l'algorithme ASCON repose sur l'utilisation répétée d'une permutation composée de trois couches successives : l'addition de constante (Pc), la substitution non-linéaire (P_S) et la diffusion linéaire (P_L). Ces trois couches sont enchaînées pour former une ronde de permutation, répétée 8 ou 12 fois selon le contexte.

La Figure 6 illustre l'organisation du module de permutation complet, incluant les opérateurs XOR, le multiplexeur d'entrée, et le registre de stockage de l'état.

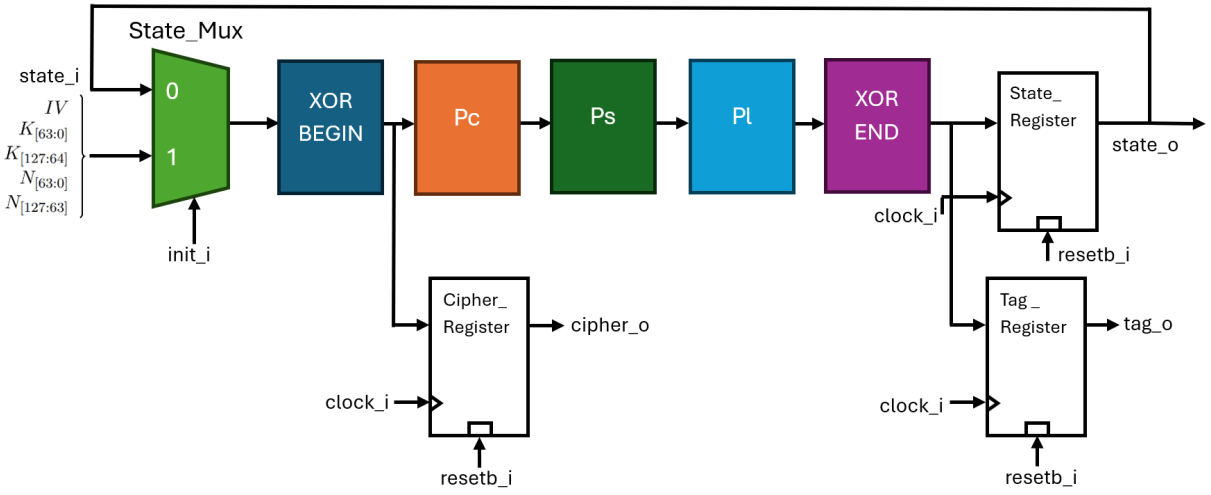

FIGURE 6 – Architecture du module de permutation ($Xor\_Begin \rightarrow P_C \rightarrow P_S \rightarrow P_L \rightarrow Xor\_End$)

4.1 Organisation de la permutation

Avant d'appliquer la permutation, l'état S est combiné avec une entrée externe (clé, donnée associée, ou bloc de texte clair) via un opérateur XOR (`xor_begin`). Cette opération permet d'injecter dynamiquement les données dans le processus de chiffrement.

La permutation proprement dite applique ensuite les trois modules en série :

1. P_C : ajout d'une constante de ronde à $S2$
2. P_S : application de la S-Box sur les colonnes de l'état
3. P_L : diffusion linéaire des bits dans chaque registre S_i

Une fois la permutation effectuée, un second XOR est appliqué via le module `Xor_End`, notamment pour générer le tag de vérification ou extraire le bloc chiffré.

Multiplexeur d'entrée (`State_Mux`)

Le multiplexeur `State_Mux` joue un rôle essentiel dans le contrôle du flux de données à l'entrée du bloc de permutation. Il permet de choisir dynamiquement, à chaque itération, quelle donnée doit être injectée dans le registre d'état S :

— Lors de l'initialisation, il sélectionne un bloc formé à partir de plusieurs sources : la clé K (128 bits), le nonce N (128 bits) et le vecteur d'initialisation IV (64 bits).

— Dans les itérations suivantes (pour les différentes phases de permutation), il sélectionne la sortie précédente du bloc de permutation.

Remarque : Le signal de sélection de ce multiplexeur est positionné à 1 uniquement au tout début du lancement de l'algorithme ASCON-AEAD128, afin de charger l'état initial avec les données d'entrée. Dès que cette phase d'initialisation est terminée, le signal de sélection bascule à 0 et le reste jusqu'à la fin du chiffrement : la permutation prend alors en entrée la sortie précédente du bloc, assurant ainsi la continuité des transformations successives.

Registre d'état (`State_Register`)

Afin de maintenir la cohérence du traitement entre les cycles, l'état cryptographique S est stocké dans un registre mémoire contrôlé par un signal d'activation (`enable`). Ce registre joue un rôle central :

— Il conserve les modifications successives de S à travers les rondes.

— Il permet de geler l'état pendant les phases de chargement ou de synchronisation.

Son contrôle est assuré par la FSM, qui déclenche l'écriture dans le registre uniquement lorsque la permutation est prête.

L'ensemble de cette architecture modulaire permet une exécution claire, séquencée et facilement vérifiable via des testbenchs. La suite de cette section détaille la conception et la validation de chaque module.

4.2 Module P_C

But du module : Ce module implémente l'opération d'ajout de constante p_C définie dans la spécification ASCON. À chaque ronde r, une constante c_r est appliquée au registre S_2 de l'état S via une opération XOR :

$$S_2 \leftarrow S_2 \oplus c_r$$

L'ensemble des constantes utilisées dépend du nombre de rondes de permutation appliquées. En effet, ASCON utilise deux types de permutations :

— p^{12} : utilisée lors de l'initialisation et de la finalisation (12 rondes)

— p^8 : utilisée dans les phases intermédiaires, comme le traitement des données associées ou du texte clair (8 rondes)

Pour chaque type de permutation, les constantes c_r associées à chaque ronde r sont différentes. La Table 2 montre l'ensemble des constantes utilisées pour les deux permutations.

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

TABLE 2 – Constantes d'ajout utilisées pour les permutations p^8 et p^{12}

Simulation : Un testbench a été mis en place afin d'observer l'évolution de l'état S après l'application de la première permutation, en utilisant la valeur initiale donnée dans le sujet du projet ASCON. L'objectif était de vérifier que l'addition de constante était correctement réalisée sur le registre S_2 au cours de cette première étape.

D'après la spécification, seule la portion basse (8 bits de poids faible) du registre S_2 est affectée par l'opération d'ajout de constante lors de la première ronde. Plus précisément, la constante $c_0 = 0xF0$ appliquée sur la valeur initiale de S_2 modifie son octet de fin de $0x1F$ à $0xEF$. Ce résultat est cohérent avec l'opération de XOR : $0xF0 \oplus 0x1F = 0xEF$, et plus précisément $0xF \oplus 0x1 = 0xE$ sur le dernier demi-octet.

L'observation de l'évolution de l'état dans la Figure 5 du sujet confirme ce comportement, validant ainsi que le module applique correctement la constante à la bonne position et sur la bonne portion de l'état.

```
Valeur initiale    : 00001000808C0001 6CB10AD9CA912F80 691AED630E81901F 0C4C36A20853217C 46487B3E06D9D7A8
********************************************************************************************************
-- Permutation (r=00)
Addition constante : 00001000808c0001 6cb10ad9ca912f80 691aed630e8190ef 0c4c36a20853217c 46487b3e06d9d7a8
```

FIGURE 7 – Extrait du sujet ASCON

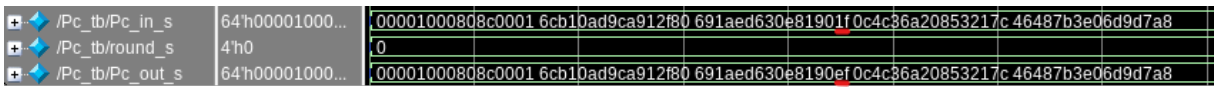

FIGURE 8 – Testbench du module P_c

4.3 Module P_S

But du module : Le module P_S réalise la couche P_S en appliquant une substitution de 5 bits par colonne à l'état S. Celui-ci est interprété comme un tableau de 64 colonnes de 5 bits, formées à partir des bits alignés des registres S_0 à S_4.

Pour chaque colonne $i \in [0, 63]$, le quintuplet $\{S_0[i], S_1[i], S_2[i], S_3[i], S_4[i]\}$ est remplacé par la valeur correspondante dans la table S-box du sujet. $S_0[i]$ représente le bit de poids fort, $S_4[i]$ le bit de poids faible.

Cette opération est appliquée colonne par colonne et constitue la transformation non-linéaire de l'algorithme.

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
Sbox(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 9 – Table de substitution utilisée pour le projet

Simulation : La simulation du module P_S repose sur une étape préalable indispensable : l'implémentation correcte du module `Sbox`, qui constitue le cœur de la substitution appliquée colonne par colonne. En effet, le module P_S effectue des appels répétés à la `Sbox` sur chaque colonne de l'état S, et il est donc essentiel de vérifier le bon fonctionnement de la `Sbox` avant de pouvoir valider le module global.

Un testbench a donc d'abord été élaboré pour `Sbox`. Plusieurs entrées ont été choisies aléatoirement afin de parcourir différentes combinaisons. À chaque fois, la sortie a été comparée à la valeur attendue selon la table de substitution donnée dans l'énoncé (**Figure 6**). Les résultats obtenus sont strictement conformes à la spécification, ce qui confirme que la S-box est fonctionnelle.

FIGURE 10 – Testbench du module *Sbox*

Une fois cette vérification effectuée, le testbench du module P_S a été mis en place. Il consiste à appliquer une entrée complète représentant les 5 registres S_0 à S_4, puis à observer leur transformation colonne par colonne. L'entrée utilisée pour cette simulation correspond à la sortie du module P_C, évaluée précédemment lors de l'étape d'initialisation de l'algorithme de chiffrement. Les sorties obtenues sont cohérentes avec celles attendues selon la logique de la substitution, ce qui valide le fonctionnement du module P_S.

```
Addition constante : 00001000808c0001 6cb10ad9ca912f80 691aed630e8190ef 0c4c36a20853217c 46487b3e06d9d7a8
Substitution S-box : 25f7c341c45f9912 23b794c540876856 b85451593d679610 4fafba264a9e49ba 62b54d5d460aded4
```

FIGURE 11 – Extrait du sujet ASCON

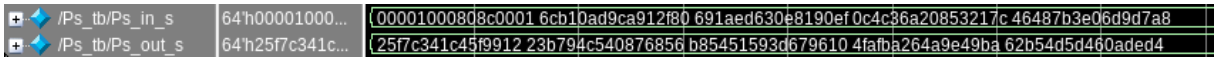

FIGURE 12 – Testbench du module P_s

4.4 Module P_L

But du module : Le module P_L implémente la couche de diffusion linéaire P_L de l'algorithme ASCON. Il applique une transformation à chacun des registres S_i de l'état S selon une fonction Σ_i spécifique, visant à assurer la diffusion des bits.

Pour chaque registre S_i, l'opération consiste à faire un XOR entre le registre initial et deux de ses versions cycliquement décalées (rotations à droite) selon des valeurs définies dans la spécification.

$$S_i \leftarrow S_i \oplus (S_i \gg r_1) \oplus (S_i \gg r_2)$$

Les décalages (r_1, r_2) varient selon le registre, comme défini dans le sujet. Cette opération garantit une meilleure propagation des dépendances entre bits dans tout l'état.

Simulation : Pour simuler le module P_L, la sortie du module P_S, validée précédemment, a été utilisée comme entrée. Cette approche permet de s'inscrire dans la continuité logique du déroulement de l'algorithme ASCON, notamment au sein de l'étape d'initialisation.

Le testbench applique l'opération de diffusion à chacun des registres S_i en vérifiant que les bits de sortie sont cohérents avec l'opération définie dans la spécification. Les résultats observés sont conformes aux attentes, confirmant que chaque Σ_i applique correctement les deux rotations spécifiques, puis la combinaison via XOR.

```
Substitution S-box : 25f7c341c45f9912 23b794c540876856 b85451593d679610 4fafba264a9e49ba 62b54d5d460aded4
Diffusion linéaire : 932c16dd634b9585 b48a3c3fe8fb45ce a69f28b0c721c340 05e1761f1e1fcb67 64d322a896b791cf
```

FIGURE 13 – Extrait du sujet ASCON

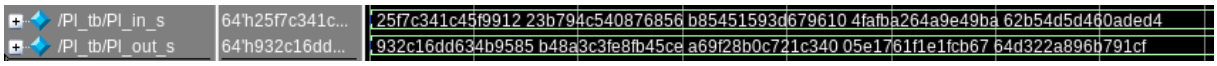

FIGURE 14 – Testbench du module P_L

La validité du module est ainsi assurée, et l'état transformé est prêt à être réinjecté dans le processus de permutation globale.

4.5 Module `Permutation`

But du module : Le module `Permutation` a pour objectif de regrouper les trois sous-modules fondamentaux de la permutation ASCON : P_C, P_S et P_L, correspondant respectivement à l'addition de constante, à la substitution et à la diffusion. Ce module sert d'étape intermédiaire pour tester le bon enchaînement des trois transformations sur l'état S, en reproduisant fidèlement la logique d'une ronde complète.

Il ne s'agit pas ici d'un composant final de l'architecture, mais d'un outil de validation permettant de vérifier que la permutation combinée fonctionne comme attendu lorsqu'elle est appliquée en séquence sur l'état.

Simulation : Pour tester le module `Permutation`, l'approche a été d'observer non plus une seule transformation isolée, mais l'enchaînement complet des 12 rondes de la phase d'initialisation de l'algorithme ASCON.

Un testbench a donc été mis en place pour appliquer ces 12 rondes successives sur un état initial défini, en s'assurant que chaque appel à P_C, P_S et P_L s'effectue dans le bon

ordre et avec les bons paramètres (en particulier la constante de ronde pour `pC`). À chaque étape, l'état résultant a été comparé à celui attendu selon la spécification.

```
-- Permutation (r=11)
Addition constante : 518cf1849f50d5a4 5f512813fd2164fd 1138791a37c91dec 0ea110ff054a5ef7 66a93a92765f6c33
Substitution S-box : 175590728cf2d63b 68dcb2fb13e4c28c d19e84f64702a6ee 37e5aae435bd8ef5 20580a6c673576ac
Diffusion linéaire : 82bf91294ba5808d d81eeca694136f8a 0217bc9ebd9fff02 4dd2c87c59c2fb48 4e2b20c3e9eb3044
```

FIGURE 15 – Extrait du sujet ASCON

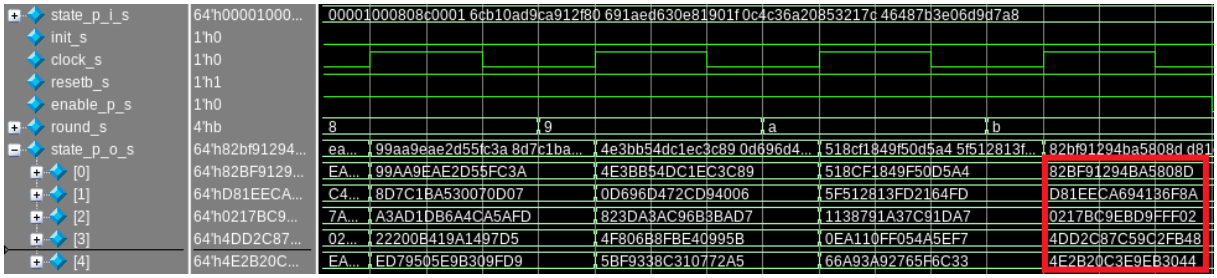

FIGURE 16 – Testbench du module `Permutation`

Les résultats de simulation montrent que le module enchaîne correctement les transformations, et que l'état évolue de manière cohérente à travers les 12 rondes. Cela confirme le bon fonctionnement de l'assemblage des trois sous-modules dans la logique de permutation.

4.6 Modules `Xor_Begin` et `Xor_End`

But des modules : Les modules `Xor_Begin` et `Xor_End` sont essentiels à l'implémentation de l'algorithme ASCON-AEAD128. Leur rôle est de permettre, à des étapes spécifiques du chiffrement, l'injection conditionnelle de données dans l'état S via une opération XOR. Ces opérations varient en fonction de la phase de l'algorithme (initialisation, donnée associée, texte clair, ou finalisation) et des portions de l'état ciblées (début ou fin).

Le `Xor_Begin` est utilisé principalement lors du traitement des données associées et du texte clair. Il effectue un XOR sur les 128 bits de poids fort de l'état, c'est-à-dire les registres S_0 et S_1.

Le `Xor_End`, quant à lui, intervient plutôt en fin de phase (par exemple, lors du calcul du tag). Il applique un XOR sur les 192 bits de poids faible (les registres S_2, S_3, S_4), avec des valeurs différentes selon la situation (clé, ...).

Structure des modules : Le module `Xor_Begin` est construit autour d'un multiplexeur à 2 entrées. L'une des entrées correspond au passage direct de l'état sans modification, tandis que l'autre applique un XOR entre l'état et une donnée externe (par exemple, les données associées ou le texte clair). Le signal de sélection (1 bit) détermine si l'on effectue ou non le XOR.

Le module `Xor_End` est conçu pour gérer plusieurs situations prévues par l'algorithme. Il comporte un multiplexeur à 4 entrées. La première permet, comme dans `Xor_Begin`, de

laisser l'état inchangé. Les trois autres permettent d'appliquer un XOR avec différentes valeurs, selon le contexte : la clé secrète, ou une constante spécifique. Le signal de sélection est codé sur 2 bits, ce qui permet d'activer précisément le traitement adapté à chaque étape du protocole.

Simulation : Pour tester le `Xor_Begin`, un testbench a été conçu autour de la donnée associée A_1 définie dans le sujet. Le but était de vérifier que le module applique correctement le XOR sur les 128 bits de poids fort (MSB) lorsque le signal de sélection est activé. Les résultats de simulation montrent que la donnée est correctement injectée dans l'état selon l'attendu, validant le comportement du module.

```
Initialisation      : 82BF91294BA5808D D81EECA694136F8A 0217BC9EBD9FFF02 2163C2A59353D4C8 2731CDA0E76AA05B
**********************************************************************************************************
État ^ donnée A1    : edcbb14c28ceccc d81eeca7f67c2daa 0217bc9ebd9fff02 2163c2a59353d4c8 2731cda0e76aa05b
```

FIGURE 17 – Extrait du sujet ASCON

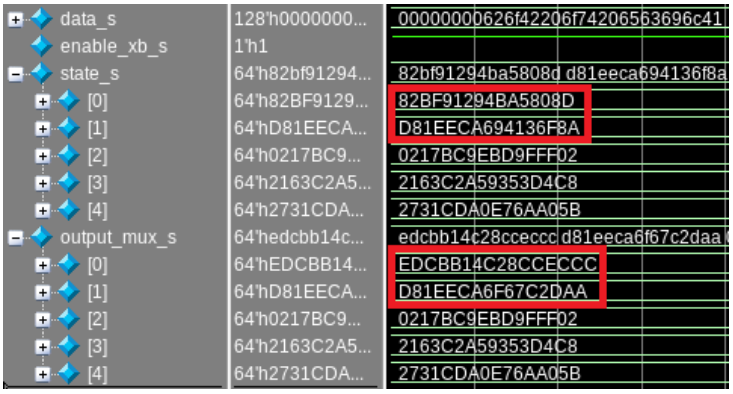

FIGURE 18 – Testbench du module *Xor_Begin*

En ce qui concerne le `Xor_End`, la simulation a été menée en prenant pour entrée la sortie de la phase d'initialisation. Un XOR a été appliqué avec la donnée spécifiée dans le schéma de l'algorithme, sur les 192 bits de poids faible. Le testbench a permis d'observer une sortie conforme aux spécifications, prouvant que le module applique correctement la bonne donnée en fonction du signal de sélection.

```
Diffusion linéaire : 82bf91294ba5808d d81eeca694136f8a 0217bc9ebd9fff02 4dd2c87c59c2fb48 4e2b20c3e9eb3044
État ^ (0...0 & K) : 82bf91294ba5808d d81eeca694136f8a 0217bc9ebd9fff02 2163c2a59353d4c8 2731cda0e76aa05b
```

FIGURE 19 – Extrait du sujet ASCON

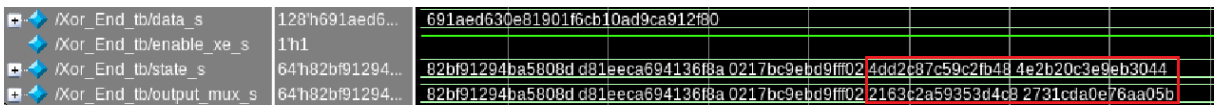

FIGURE 20 – Testbench du module *Xor_End*

Ces deux modules permettent ainsi d'ajouter ou non certaines données à l'état S à des moments précis de l'algorithme, en fonction de la phase dans laquelle on se trouve. Cette flexibilité est essentielle pour respecter la logique du chiffrement ASCON.

Remarque – Unification potentielle des modules `Xor` (`Begin` / `End`)

Dans la structure actuelle de l'implémentation, on distingue deux types principaux de `XOR` utilisés durant le déroulement de l'algorithme : `Xor_Begin` et `Xor_End`. Ces deux opérations interviennent à différentes étapes du chiffrement, avec des fonctions bien précises. Par exemple, dans la phase d'initialisation, on applique un `Xor_End` sur une disposition particulière de la clé, ciblant spécifiquement les 192 derniers bits de l'état S (`S2`, `S3` et `S4`). Ensuite, lors de la phase de données associées, on applique un `Xor_Begin` sur les 128 premiers bits (`S0` et `S1`) avec la donnée associée A_1.

On pourrait envisager une réorganisation complète de ces opérations pour n'en faire qu'une seule unifiée, via un module *XOR universel*, qui viendrait remplacer les `Xor_Begin` et `Xor_End` actuels. Ce module serait plus générique, capable d'appliquer un `XOR` sur tout ou partie de l'état S, selon les besoins.

Concrètement, cette approche reviendrait à fusionner les opérations de `XOR` en un seul état combiné. Par exemple, l'opération `Xor_End` sur les 192 bits de clé (`S2` à `S4`), suivie d'un `Xor_Begin` sur les 128 bits de données associées (`S0` et `S1`), pourraient être exécutées simultanément dans un même cycle, avec un seul `XOR` universel appliqué sur les 320 bits complets de l'état S.

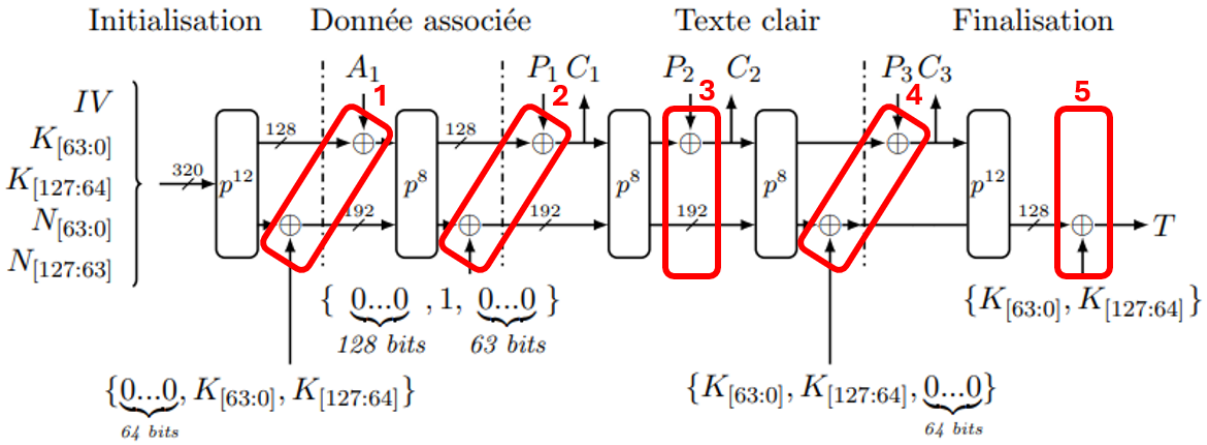

FIGURE 21 – Unification potentielle des modules `XOR`

Le fonctionnement de ce module universel repose sur un **multiplexeur à 6 entrées**, chacune correspondant à un mode spécifique d'application du XOR. Ces entrées permettent de couvrir l'ensemble des cas d'usage identifiés dans l'algorithme :

- **Entrée 0** : aucune opération. L'état entrant S est renvoyé tel quel, sans modification. Cette option est utilisée dans les phases où aucun XOR n'est requis.
- **Entrées 1 à 5** : elles correspondent aux différents schémas de combinaison XOR définis dans l'algorithme.

Ces six modes d'opération sont résumés visuellement dans la **figure 21**, qui précise les plages de bits concernées par chaque entrée du multiplexeur.

Cette approche présente une alternative structurée et modulaire qui permettrait, au prix d'une complexité logique légèrement accrue, de réduire le nombre de blocs `XOR` distincts et de simplifier potentiellement les transitions d'état.

Cependant, bien que séduisante sur le plan de la factorisation, cette solution ne sera **pas retenue** dans la version actuelle du projet, principalement pour des raisons de clarté, de lisibilité et de pédagogie. En effet, garder une distinction nette entre `XOR_BEGIN` et `XOR_END` permet une meilleure compréhension du fonctionnement de l'algorithme dans un contexte d'apprentissage, notamment en lien avec les différentes étapes du chiffrement ASCON.

4.7 Module `Permutation_XOR`

But du module : Le module `Permutation_XOR` constitue l'un des éléments centraux du projet. Il combine la logique de permutation (P_C, P_S, P_L) avec les opérations de XOR au début et à la fin de chaque ronde. C'est cette combinaison qui permet à l'algorithme ASCON de chiffrer les données en fonction des différentes étapes (initialisation, donnée associée, texte clair...).

Contrairement au module `Permutation` précédent qui servait à valider l'enchaînement $P_C \rightarrow P_S \rightarrow P_L$, le module `Permutation_XOR` permet de vérifier que l'ensemble de la permutation complète, avec les XOR associés, fonctionne correctement. C'est donc un point pivot de l'architecture, sa réussite conditionne la validité de toute la suite du projet.

Simulation : Un testbench a été mis en place pour appliquer les 12 rondes de la phase d'initialisation, cette fois-ci en incluant également les modules `Xor_Begin` et `Xor_End`. L'objectif était de vérifier que les opérations de permutation et de XOR s'enchaînaient correctement sur l'état S.

Les résultats observés montrent que l'état évolue à chaque étape de manière conforme à la spécification ASCON. L'intégration des modules précédemment validés dans un seul bloc fonctionne comme prévu, ce qui valide l'ensemble du module `Permutation_XOR`.

```
-- Permutation (r=11)
Addition constante : 518cf1849f50d5a4 5f512813fd2164fd 1138791a37c91dec 0ea110ff054a5ef7 66a93a92765f6c33
Substitution S-box : 175590728cf2d63b 68dcb2fb13e4c28c d19e84f64702a6ee 37e5aae435bd8ef5 20580a6c673576ac
Diffusion linéaire : 82bf91294ba5808d d81eeca694136f8a 0217bc9ebd9fff02 4dd2c87c59c2fb48 4e2b20c3e9eb3044
État ^ (0...0 & K) : 82bf91294ba5808d d81eeca694136f8a 0217bc9ebd9fff02 2163c2a59353d4c8 2731cda0e76aa05b
```

FIGURE 22 – Extrait du sujet ASCON

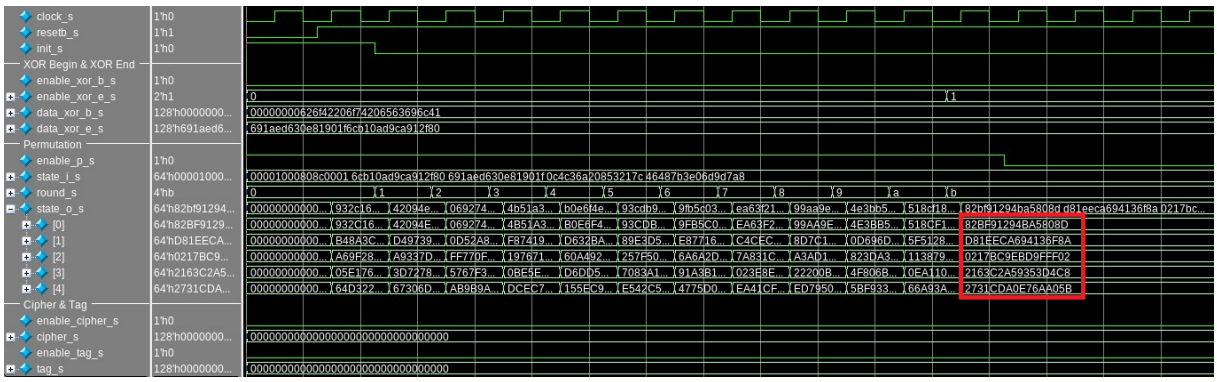

FIGURE 23 – Testbench du module *Permutation_XOR*

4.8 Registres de sortie : `Cipher` et `Tag`

But du module : Les registres `Cipher` et `Tag` servent à stocker respectivement les blocs chiffrés C_i et le tag final T généré par l'algorithme ASCON-AEAD128. Ces éléments constituent la sortie principale du chiffrement.

Les blocs chiffrés sont produits lors de la phase de traitement du texte clair, tandis que le tag est généré lors de la finalisation. Ce tag joue un rôle crucial dans l'authentification du message : il permet au destinataire de vérifier que le message n'a pas été modifié et qu'il provient bien de l'expéditeur.

Ce module ne nécessite pas de testbench spécifique, puisqu'il s'agit simplement de mémoriser les données à la sortie des modules précédents.

4.9 Module `Permutation_XOR_Final`

But du module : Le module `Permutation_XOR_Final` regroupe l'ensemble du processus de permutation complète, incluant les opérations de XOR au début et à la fin, ainsi que les registres de sortie `Cipher` et `Tag`. Il représente une version intégrée et complète du chiffrement ASCON dans sa phase de permutation.

Aucune simulation dédiée n'a été effectuée pour ce module. En effet, tous les sous-modules qu'il contient (`Permutation_XOR`) ayant été préalablement testés et validés, il n'était pas nécessaire de reproduire un test supplémentaire. Le comportement attendu étant la combinaison directe de modules fonctionnels, on en déduit que ce module est lui aussi valide.

La construction de ce bloc clôt ainsi la première étape du projet, qui consistait à modéliser et vérifier l'ensemble du processus de permutation et de sortie dans ASCON-AEAD128.

5 Modélisation de la machine de contrôle (FSM)

La machine à états finis (FSM) constitue le module de contrôle central de l'architecture ASCON-AEAD128. Elle pilote le déroulement complet de l'algorithme en orchestrant les différentes étapes dans l'ordre défini : initialisation, traitement des données associées, chiffrement du texte clair, puis finalisation.

À chaque cycle d'horloge, la FSM gère la transition d'un état à un autre et produit les signaux nécessaires pour activer ou désactiver les différents blocs fonctionnels : permutation, opérations de XOR, écriture dans les registres de sortie, etc. Elle coordonne également l'activation du compteur de rondes et la synchronisation globale du système.

Dans cette implémentation, une machine de Moore a été choisie. Cela permet de définir les signaux de contrôle uniquement à partir de l'état courant, ce qui rend le code plus structuré et plus lisible. Ainsi, pour chaque état actif, les signaux de contrôle correspondants sont déterminés de manière explicite, sans dépendre directement des entrées extérieures.

Par souci de simplification, le design repose uniquement sur le compteur double fourni, sans ajouter de compteur de blocs séparé. Ce choix entraîne alors un nombre plus important d'états dans la FSM, mais permet de centraliser toute la logique de séquencement dans un seul bloc de compteur. Cette FSM joue ainsi un rôle clé dans la fiabilité et la synchronisation de l'ensemble du chiffrement ASCON.

5.1 Organisation et justification de la structure de la FSM

La structure de la FSM suit une organisation régulière et lisible qui pilote le chiffrement ASCON à travers ses différentes phases : initialisation, données associées, chiffrement et finalisation. Chaque phase est construite autour d'un motif de cinq états, sauf la phase de chiffrement qui est dupliquée pour gérer deux blocs P_1 et P_2. Le motif général est le suivant :

- `idle_xxx` : état d'attente, conditionné par un signal externe.
- `conf_xxx` : état de configuration, initialise le compteur.
- `end_conf_xxx` : fin de la configuration, active la permutation et le `xor_begin` si nécessaire.
- `xxx` : état principal, exécution des rondes.
- `end_xxx` : état final, application du `xor_end`, passage à la suite.

Cette organisation n'est pas arbitraire. Elle reflète un besoin clair de séquencement sécurisé et de synchronisation. Voici une explication plus détaillée du rôle de chaque état :

- **États `idle_xxx`** : ils servent à s'assurer que toutes les conditions d'entrée sont réunies avant d'entrer dans une phase. Par exemple, `idle` attend l'activation de `start_i`, tandis que `idle_da`, `idle_tc`, etc., attendent un `data_valid_i = 1` pour garantir que les données nécessaires (clé, données associées, texte clair, etc.) sont disponibles.
- **États `conf_xxx`** : ils sont utilisés pour initialiser les compteurs de permutation (8 ou 12 rondes) à travers les signaux `init_round_p12_o` ou `init_round_p8_o`. Cet

état permet une configuration isolée, sans activer d'autres blocs trop tôt, ce qui est crucial pour éviter les conflits entres les données.

— **États `end_conf_xxx`** : une fois les compteurs prêts, cet état enclenche la première ronde réelle en activant `enable_p_o` et `active_round_o`. Si la phase l'exige, le `xor_begin` est aussi déclenché ici via `enable_xor_b_o = 1`. Ce choix permet d'effectuer le premier XOR et la première permutation dans un même cycle, assurant un démarrage cohérent.

— **États `xxx`** : c'est ici que la FSM reste pendant plusieurs cycles pour exécuter les rondes successives de permutation (jusqu'à ce que `round_i = 4'ha`). Ce bloc ne fait que cela, sans déclencher d'autres signaux perturbateurs.

— **États `end_xxx`** : ce dernier état termine la phase proprement. Il applique un `xor_end` (si besoin) via `enable_xor_e_o`, déclenche éventuellement l'écriture dans un registre de sortie (tag ou ciphertext), puis désactive le compteur. Il sert aussi à générer les signaux de fin comme `end_init_o`, `end_tc_o`, etc., pour indiquer que la phase est terminée.

L'intérêt de cette organisation par motifs est double : d'un côté, elle assure la clarté et la sécurité du déroulement des opérations ; de l'autre, elle rend le design extensible et adaptable à des variantes de l'algorithme (nombre de blocs, types de données, changements de permutation). C'est aussi un atout pédagogique important pour comprendre et simuler chaque étape sans ambiguïté.

5.2 Diagramme d'état de la FSM

Le diagramme d'état ci-dessous illustre la transition entre les différents états décrits précédemment. Il offre une vue globale du déroulement de l'algorithme à travers la FSM.

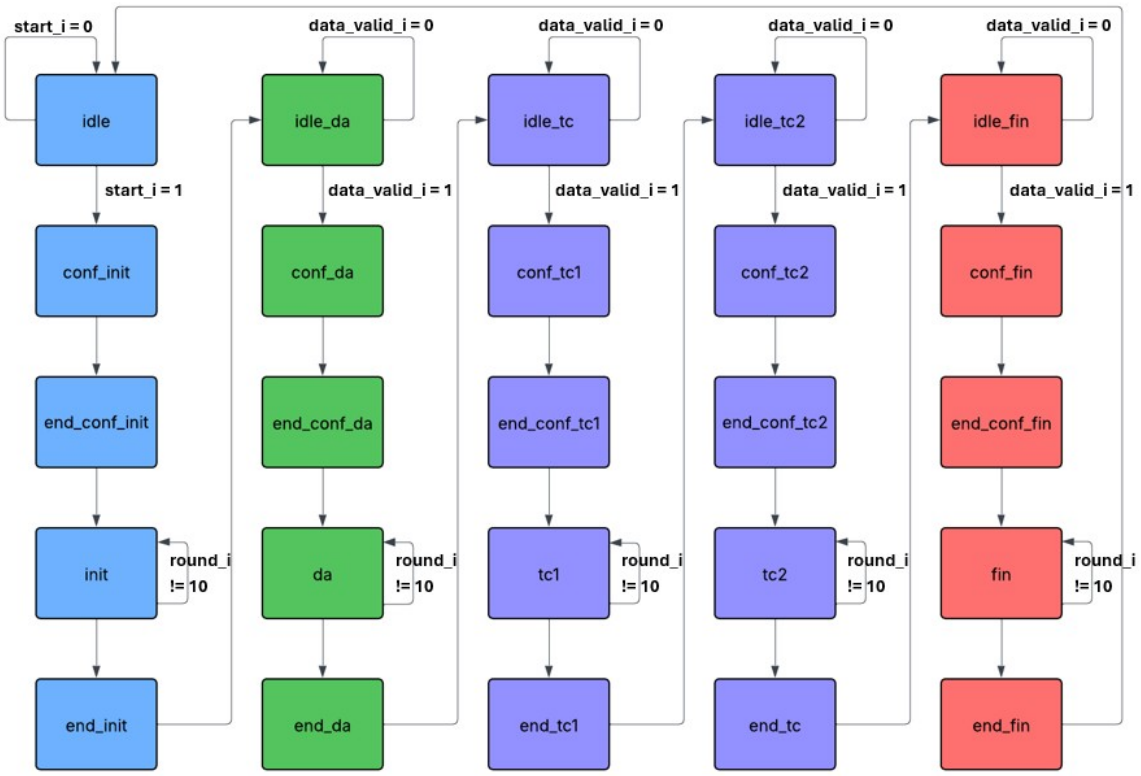

FIGURE 24 – Diagramme d'état de la Machine d'état)

Remarque : Le dernier état de la machine de contrôle, `end_fin`, est connecté directement au tout premier état `idle`. Cela permet de refermer la boucle de la FSM et de revenir à un état d'attente, marquant ainsi la fin de l'algorithme. L'état `idle` n'autorise une transition vers l'étape suivante que si le signal `start_i` est activé. Or, dans notre architecture, `start_i` n'est activé qu'une seule fois au tout début de l'exécution, puis repasse à 0. Ce fonctionnement garantit qu'une fois le cycle complet du chiffrement terminé, le système reste figé en `idle`, en attendant un nouveau déclenchement manuel, assurant ainsi une exécution unique et contrôlée du chiffrement.

5.3 Table de vérité des signaux de commande

Pour chaque état de la FSM, les signaux de contrôle générés sont listés dans le tableau ci-dessous. Celui-ci permet d'analyser le rôle de chaque signal dans le déroulement des étapes, ainsi que leur déclenchement à des instants précis de l'algorithme.

	Initialisation					Traitement des données associées				
	idle	conf_init	nd_conf_ir	init	end_init	idle_da	conf_da	nd_conf_d	da	end_da
init_o	0	1	1	0	0	0	0	0	0	0
enable_p_o	0	0	1	1	1	0	0	1	1	1
enable_xor_b_o	0	0	0	0	0	0	0	1	0	0
enable_xor_e_o	00	00	00	00	01	00	00	00	00	10
active_round_o	0	0	1	1	0	0	1	1	1	0
init_round_p12_o	0	1	0	0	0	0	0	0	0	0
init_round_p8_o	0	0	0	0	0	0	1	0	0	0
enable_tag_register	0	0	0	0	0	0	0	0	0	0
enable_cipher_register	0	0	0	0	0	0	0	0	0	0
cipher_valid_o	0	0	0	0	0	0	0	0	0	0
end_o	0	0	0	0	0	0	0	0	0	0
end_init_o	0	0	0	0	0	1	0	0	0	0
end_da_o	0	0	0	0	0	0	0	0	0	0
end_tc_o	0	0	0	0	0	0	0	0	0	0
end_final_o	0	0	0	0	0	0	0	0	0	0

FIGURE 25 – Table de vérite de la phase Initialisation & Donnée associée

	Traitement du texte en clair									
	idle_tc	conf_tc1	nd_conf_t	tc1	end_tc1	idle_tc2	conf_tc2	nd_conf_t	tc2	end_tc
init_o	0	0	0	0	0	0	0	0	0	0
enable_p_o	0	0	1	1	1	0	0	1	1	1
enable_xor_b_o	0	0	1	0	0	0	0	1	0	0
enable_xor_e_o	00	00	00	00	00	00	00	00	00	11
active_round_o	0	1	1	1	0	0	1	1	1	0
init_round_p12_o	0	0	0	0	0	0	0	0	0	0
init_round_p8_o	0	1	0	0	0	0	1	0	0	0
enable_tag_register	0	0	0	0	0	0	0	0	0	0
enable_cipher_register	0	0	1	0	0	0	0	1	0	0
cipher_valid_o	0	0	1	0	0	0	0	1	0	0
end_o	0	0	0	0	0	0	0	0	0	0
end_init_o	0	0	0	0	0	0	0	0	0	0
end_da_o	1	0	0	0	0	0	0	0	0	0
end_tc_o	0	0	0	0	0	1	0	0	0	0
end_final_o	0	0	0	0	0	0	0	0	0	0

FIGURE 26 – Table de vérite de la phase Texte clair

	Finalisation				
	idle_fin	conf_fin	nd_conf_fi	fin	end_fin
init_o	0	0	0	0	0
enable_p_o	0	0	1	1	1
enable_xor_b_o	0	0	1	0	0
enable_xor_e_o	00	00	00	00	01
active_round_o	0	1	1	1	0
init_round_p12_o	0	1	0	0	0
init_round_p8_o	0	0	0	0	0
enable_tag_register	0	0	0	0	1
enable_cipher_register	0	0	1	0	0
cipher_valid_o	0	0	1	0	0
end_o	0	0	0	0	0
end_init_o	0	0	0	0	0
end_da_o	0	0	0	0	0
end_tc_o	0	0	0	0	0
end_final_o	0	0	0	0	1

FIGURE 27 – Table de vérite de la phase Finalisation

5.4 Simulation de la FSM

Afin de valider le bon fonctionnement de la machine de contrôle, une simulation complète a été réalisée à l'aide d'un testbench. Celle-ci permet d'observer l'évolution des signaux de commande et des transitions d'états en fonction des entrées (`start_i`, `data_valid_i`, etc.).

— Elle montre que la FSM reste en `idle` tant que `start_i` n'est pas activé.
— Chaque phase est correctement déclenchée, et les signaux de contrôle s'activent aux bons cycles.
— Les transitions suivent exactement le modèle défini, sans erreur de séquencement.

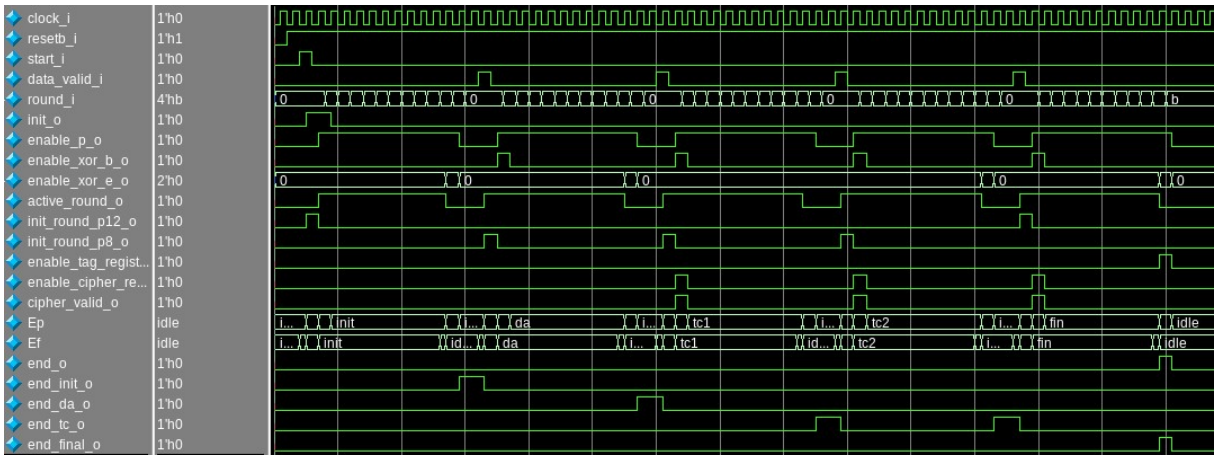

FIGURE 28 – Extrait de la simulation : transitions d'états et signaux de contrôle

Ce type de simulation permet également d'ajuster le timing des signaux si nécessaire, et de confirmer que l'implémentation est exempte de conflits entre données.

6 Implémentation du système : `ascon_top`

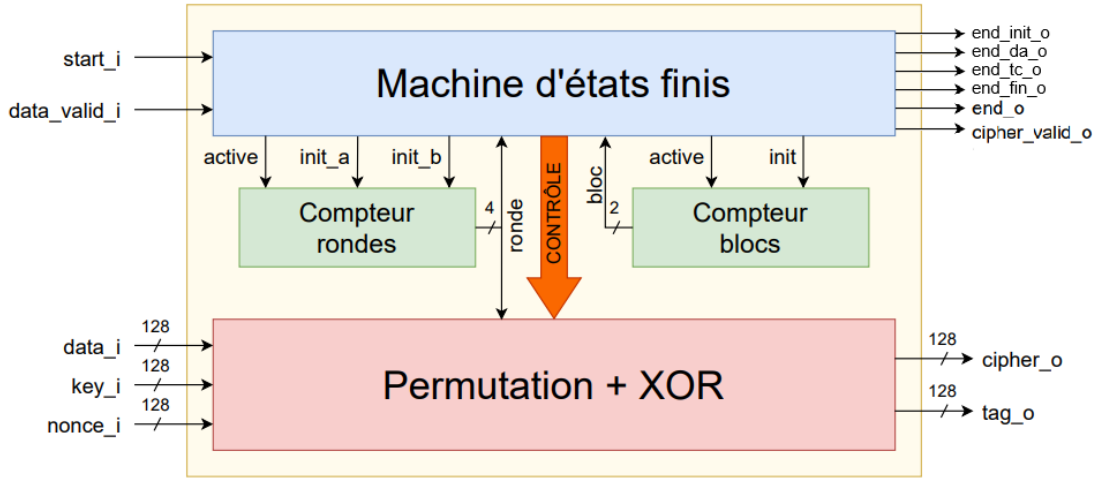

FIGURE 29 – Schéma du chiffrement Ascon-AEAD128 (module `ascon_top`)

But du module : Le module `ascon_top` représente l'assemblage final de l'ensemble des composants de l'architecture ASCON-AEAD128. Il connecte les différents blocs fonctionnels précédemment décrits : modules de permutation, XOR, registres, FSM, compteurs, pour former une solution matérielle complète, autonome et séquencée de chiffrement authentifié.

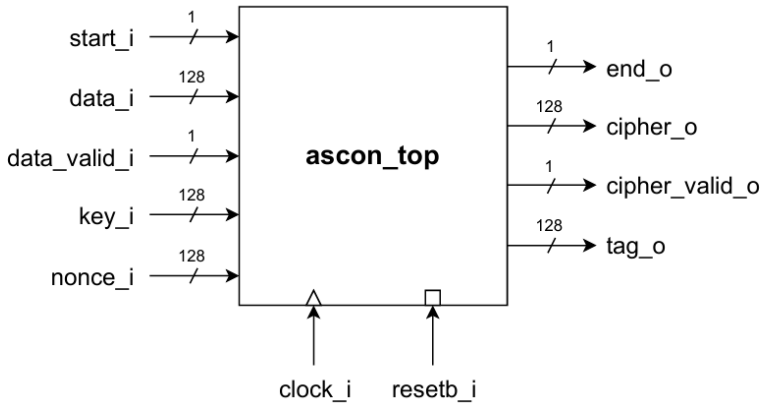

FIGURE 30 – Entrées / Sorties du module Ascon-AEAD128

Ce module centralise :
— les entrées utilisateurs (clé, nonce, données associées, texte clair),
— l'état interne S de 320 bits,
— les blocs de contrôle (FSM, compteur de rondes),
— les opérations de permutation via le bloc `permutation_xor_final`,
— la sortie chiffrée (`cipher`) et le tag d'authentification,
— les signaux de contrôle de fin de phase : `end_o`, `end_init_o`, `end_da_o`, `end_tc_o`, `end_final_o`.

À noter que les signaux de contrôle ***`end_init_o`****,* ***`end_da_o`****,* ***`end_tc_o`****,* ***`end_final_o`*** *et* ***`end_o`****, bien qu'ils ne soient pas explicitement représentés sur le schéma, sont bel et bien générés par le module et jouent un rôle clé dans la synchronisation globale du système.*

Il joue un rôle fondamental, car c'est lui qui orchestre le bon enchaînement des opérations en regroupant les modules fonctionnels sous le pilotage d'une logique de contrôle intégrée. C'est aussi le seul module qui sera instancié au plus haut niveau dans le testbench global.

Simulation : La simulation du module `ascon_top` a été menée pour valider le bon fonctionnement global de l'architecture. Elle s'appuie sur un testbench contenant une séquence complète de chiffrement, simulant l'appel à l'algorithme avec un ensemble d'entrées bien définies (clé, nonce, données associées, texte clair), comme spécifié dans le sujet.

Les différents signaux ont été observés tout au long de la simulation pour vérifier :
— la bonne progression de la FSM à travers les quatre phases de l'algorithme,
— le déclenchement correct des modules de permutation et de XOR,
— la cohérence des valeurs intermédiaires de l'état S,

— l'extraction correcte des blocs C_1, C_2, C_3 et du tag final T.

C 0x42B995A03C96C3611BBD350F39FF3A
0x4217E1E21263792197D30FAEEA29BD67
0xC9C1495620515B13C0EE02B8FAD31121 (47 octets)

T 0xF366F456CB2976594EB3452CE34318DB (16 octets)

FIGURE 31 – Extrait du sujet ASCON

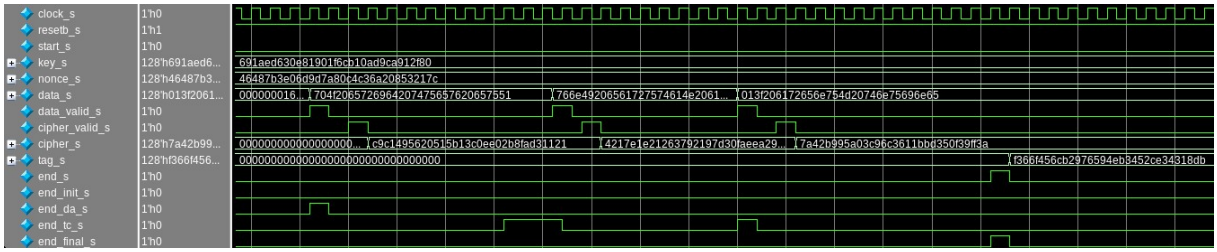

FIGURE 32 – Extrait de la simulation : transitions d'états et signaux de contrôle

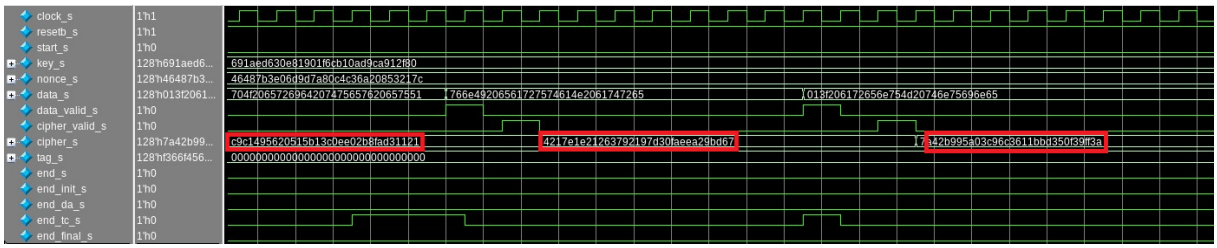

FIGURE 33 – Extrait de la simulation : transitions d'états et signaux de contrôle

FIGURE 34 – Extrait de la simulation : transitions d'états et signaux de contrôle

En conclusion, les résultats obtenus confirment la conformité de l'implémentation avec la spécification officielle d'ASCON-AEAD128. L'implémentation finale permet donc de valider l'ensemble du processus de chiffrement authentifié de manière fiable et séquencée. La progression des états, le déclenchement précis des signaux de contrôle, ainsi que les valeurs de sortie (`cipher`, tag) sont en parfaite adéquation avec les résultats attendus.

7 Difficultés rencontrées

Le projet a présenté plusieurs difficultés, liées autant à la prise en main de l'environnement qu'à la compréhension fine du fonctionnement de l'algorithme et des outils de description matérielle.

1. Prise en main de SystemVerilog et changement de paradigme
La première difficulté a résidé dans la compréhension de ce que signifie "décrire un circuit" plutôt que de "coder" un programme. En tant qu'étudiants habitués à des langages comme Python ou C, il a fallu changer totalement de logique pour comprendre que SystemVerilog est un langage de description matérielle. Cela a demandé un temps d'adaptation important au début du projet.

2. Débogage de testbenchs et erreurs de synchronisation
Les simulations ont parfois présenté des résultats incohérents dus à de simples oublis de constantes de temps ou de décalages entre signaux.

3. Compréhension de l'ordre des opérations
Bien que l'enchaînement général des opérations soit cohérent lorsqu'on analyse la logique de l'algorithme, il serait trompeur de considérer chaque module (XOR, permutation, etc.) comme totalement indépendant dans le temps. Par exemple, il ne s'agit pas simplement d'appliquer un `Xor_Begin`, d'enchaîner ensuite 12 rondes de permutation, puis de conclure avec un `Xor_End`. En réalité, certaines opérations de XOR doivent intervenir pendant une ronde spécifique de permutation, généralement la première ou la dernière. Ainsi, pour qu'un `Xor_Begin` ait bien lieu au début d'une phase, il faut activer son signal de contrôle précisément lors du premier round de permutation.

4. Représentation binaire particulière pour les données associées
Dans la phase de traitement des données associées, il est demandé d'appliquer une opération de XOR l'état avec une structure très spécifique : 128 bits de zéros, suivis d'un bit à 1, puis 63 bits de zéros. Bien que cette représentation soit triviale à exprimer en hexadécimal, une simple erreur d'attention, comme confondre un `0x1` avec un `0x8`, peut avoir des conséquences importantes. Un détail aussi minime, s'il est mal géré, compromet directement la validité du XOR appliqué à cette étape critique du chiffrement.

5. FSM : une implémentation exigeante
La modélisation de la machine de contrôle a été un véritable défi. La bonne synchronisation des signaux, le suivi de l'état S, et la gestion des blocs d'enchaînement (texte clair, données associées, finalisation) ont demandé une vision globale du comportement du chiffrement. Une fois la logique de données associées comprise, les autres parties ont pu être déduites plus facilement.

Malgré ces difficultés, chaque erreur a constitué une source d'apprentissage précieuse. Le débogage répété, la lecture de simulations et la recherche d'équilibre entre modules ont permis de mieux comprendre la logique matérielle et le comportement global d'un algorithme cryptographique comme ASCON.
Une piste d'amélioration notable serait d'intégrer un compteur de blocs, ce qui permettrait de simplifier la FSM en réduisant le nombre d'états nécessaires pour gérer la progression des données associées et des blocs de texte clair.

8 Conclusion

Ce projet a permis de concevoir et de simuler une architecture matérielle complète de l'algorithme de chiffrement authentifié ASCON-AEAD128. À travers une approche modulaire et progressive, chaque bloc fonctionnel a été conçu, testé et validé, de la permutation élémentaire à l'intégration finale via la machine à états finis et le module `ascon_top`.

Les principales difficultés rencontrées ont concerné la prise en main du langage de description matériel (SystemVerilog), l'ordonnancement précis des opérations, et la synchronisation des signaux au sein de la FSM. Néanmoins, l'expérience acquise a permis de surmonter ces obstacles et de mieux appréhender les contraintes spécifiques à la conception de circuits numériques sécurisés.

Le fonctionnement global de l'implémentation a été validé à travers des simulations représentatives, confirmant la cohérence des résultats obtenus avec les spécifications du chiffrement ASCON. Ce travail offre ainsi une base solide pour toute amélioration ou extension future.

Ce projet constitue une expérience formatrice, mêlant algorithmique cryptographique, rigueur de la modélisation matérielle et mise en œuvre concrète d'un protocole de sécurité moderne.