Modélisation SystemVerilog de l'algorithme de chiffrement ASCON

1A - Conception d'un Système Numérique

AHNANI Ali

AVRIL 2025

Contents

List of Figures

1

Introduction

La sécurisation des échanges numériques est aujourd'hui une composante essentielle de toute architecture de communication. L'algorithme **ASCON128**, dédié au chiffrement authentifié avec données associées (AEAD), répond à cette exigence en combinant efficacité, légèreté et robustesse.

Il permet non seulement de garantir la confidentialité d'un message en s'assurant que seul le destinataire légitime puisse le déchiffrer mais aussi d'en vérifier l'intégrité grâce à un *tag* d'authenticité. Cette double fonctionnalité protège les données sensibles tout en validant l'identité des communicateurs.

Reconnue pour ses performances et sa fiabilité, l'architecture ASCON128 a été sélectionnée comme l'un des gagnants du concours **CAESAR** (*Competition for Authenticated Encryption: Security, Applicability, and Robustness*).[2]

Ce rapport présente la modélisation matérielle de l'algorithme ASCON128 à l'aide du langage *SystemVerilog*. L'objectif est d'implémenter une version simplifiée mais fonctionnelle de cet algorithme sous forme de modules numériques hiérarchisés.

La conception inclut notamment les transformations élémentaires (`PC`, `PS`, `PL`), les blocs de `XOR`, ainsi qu'une machine à états finis (`FSM`) pour orchestrer les différentes phases du chiffrement.

Les différentes simulations sont réalisées avec le logiciel `ModelSim`, en vue de valider le comportement fonctionnel de l'ensemble du système.

2

Description de l'algorithme ASCON-AEAD128

2.1 Formalisme et notations

2.1.1 L'état courant S

L'algorithme ASCON manipule un état courant S de 320 bits, divisé en cinq registres S_i de 64 bits chacun $S = \{S_0, S_1, S_2, S_3, S_4\}$. Cette structure est représentée dans la Figure 2.1. [4]

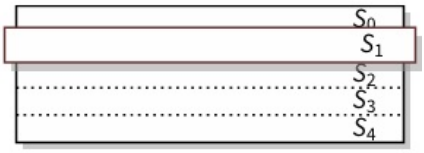

Figure 2.1: État S vu comme un tableau de registres

L'état se subdivise également en :

- $S_r = \{S_0, S_1\}$: partie externe (128 bits)
- $S_c = \{S_2, S_3, S_4\}$: partie interne (192 bits)

Il peut enfin être vu comme 64 colonnes de 5 bits, utile dans certaines opérations de permutation [4](Figure 2.2).

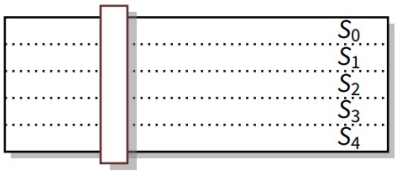

Figure 2.2: État S sous forme de colonnes

2.1.2 Notations utilisées dans la version simplifiée de l'algorithme ASCON128

La Table 2.1 décrit les notations utilisées dans ce document pour représenter les différentes données manipulées par l'algorithme. [4]

Symbole	Description
K	Clé secrète *key* K de 16 octets (128 bits)
N	Nombre arbitraire *nonce* N de 16 octets (128 bits)
T	Tag T de 16 octets (128 bits)
P	Texte clair *plaintext* P de 47 octets (376 bits)
C	Texte chiffré *ciphertext* C de 47 octets (376 bits)
A	Données associées *associated data* A de 12 octets (96 bits)
IV	Vecteur d'initialisation *initialisation vector* IV de 8 octets (64 bits) : `0x00001000808c0001`

Table 2.1: Notations

2.2 Architecture globale de l'algorithme ASCON-AEAD128

2.2.1 Structure fonctionnelle de l'algorithme

L'algorithme ASCON-AEAD128 repose sur un chiffrement authentifié organisé en quatre grandes étapes : l'initialisation de l'état interne S, le traitement des données associées A, le chiffrement du texte clair P, puis la finalisation avec génération du tag T. [4]

L'état S est initialisé à partir du vecteur d'initialisation IV, de la clé secrète K et du nonce N. À chaque étape, des blocs sont injectés dans l'état et modifiés par une permutation. Le traitement du message s'effectue bloc par bloc, et les transformations successives permettent d'assurer à la fois la confidentialité et l'intégrité des données.

Ces étapes s'articulent autour de blocs de permutation p^a et p^b appliqués à l'état interne, comme illustré dans la Figure 2.3, représentant l'architecture interne complète du chiffrement.

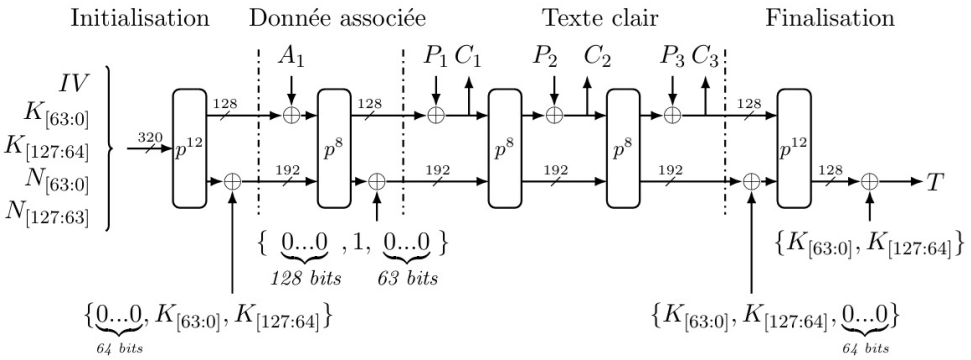

Figure 2.3: Structure fonctionnelle du chiffrement ASCON-AEAD128

Le processus complet est illustré par la Figure 2.4, qui détaille les quatre phases de traitement, de l'entrée jusqu'à la sortie du tag.

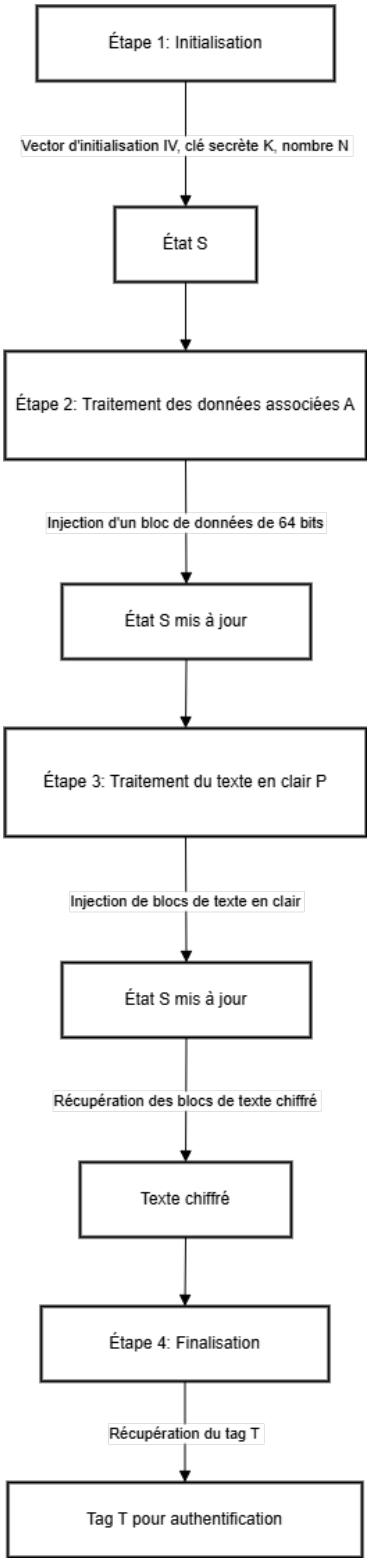

Figure 2.4: Étapes de traitement de l'algorithme ASCON-AEAD128

2.2.2 Architecture matérielle du circuit `ascon_top`

La figure 2.5 illustre l'architecture matérielle du module `ascon_top` implémenté en SystemVerilog. Ce circuit numérique a pour rôle de chiffrer un message clair tout en générant un tag d'authentification à l'aide de l'algorithme ASCON-AEAD128.

Le module prend en charge les signaux suivants :

- `clock_i` : horloge principale pilotant les éléments séquentiels du circuit.
- `resetb_i` : signal de réinitialisation asynchrone actif à l'état bas.
- `start_i` : signal de déclenchement du chiffrement.
- `data_i` : entrée de données de 128 bits à chiffrer.
- `data_valid_i` : indique la validité de la donnée présente sur `data_i`.
- `key_i` : clé de chiffrement de 128 bits.
- `nonce_i` : nonce arbitraire de 128 bits utilisé pour l'initialisation.

En sortie, le circuit fournit :

- `cipher_o` : bloc chiffré de 128 bits.
- `cipher_valid_o` : signale la validité des données présentes sur `cipher_o`.
- `tag_o` : tag d'authentification de 128 bits.
- `end_o` : signal indiquant la fin du processus de chiffrement, et que la sortie `tag_o` est disponible.

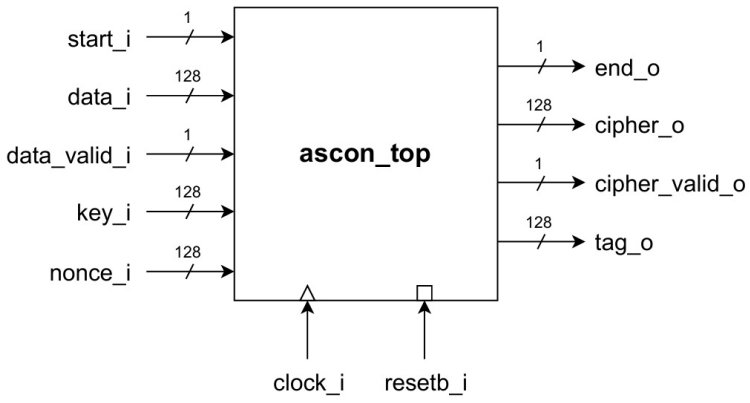

Figure 2.5: Schéma d'interface du module `ascon_top`

2.2.3 Description formelle de l'algorithme ASCON-AEAD128

Comme vu précédemment, l'algorithme ASCON-AEAD128 suit une structure composée de quatre grandes étapes : l'initialisation, le traitement des données associées, le traitement du texte en clair, et la finalisation. Ces étapes s'effectuent sur un état interne S composé de cinq mots de 64 bits.

Nous résumons ci-dessous la spécification fonctionnelle de l'algorithme :

Étape	**Description**
Entrées	$K \in \mathbb{F}_2^{128}$, $N \in \mathbb{F}_2^{128}$, $A \in \mathbb{F}_2^{96}$, $P \in \mathbb{F}_2^{376}$
Sorties	Texte chiffré C et tag d'authentification $T \in \mathbb{F}_2^{128}$
Initialisation	$S \leftarrow \{IV, K_{63:0}, K_{127:64}, N_{63:0}, N_{127:64}\}$ $S \leftarrow p^{12}(S) \oplus \{0^{192}, K_{63:0}, K_{127:64}\}$
Données associées	Padding $A \rightarrow A_1 = \text{pad}(A)$ $S \leftarrow p^8(\{S_r \oplus A_1, S_c\})$ $S \leftarrow S \oplus \{0^{256}, 1, 0^{63}\}$
Texte en clair	Padding $P \rightarrow \{P_3, P_2, P_1\} = \text{pad}(P)$ Pour $i = 1$ à 2 : $S_r \leftarrow S_r \oplus P_i$ $C_i \leftarrow S_r$ $S \leftarrow p^8(S)$ Fin boucle $S_r \leftarrow S_r \oplus P_3$, $C_3 \leftarrow S_r[119:0]$
Finalisation	$S \leftarrow p^{12}(S \oplus \{0^{64}, K_{63:0}, K_{127:64}, 0^{128}\})$ $T \leftarrow \{S_4, S_3\} \oplus \{K_{127:64}, K_{63:0}\}$
Retour	$C = \{C_3, C_2, C_1\}$, T

Table 2.2: Algorithme ASCON-AEAD128 — résumé formel des étapes de traitement

Ce tableau résume l'enchaînement logique des opérations de permutation, de `XOR`, de padding et d'injection de données. Il constitue une référence essentielle à la modélisation matérielle et à la vérification des modules présentés dans les sections suivantes.

3

Modélisation des transformations élémentaires

Cette section présente les différentes opérations élémentaires utilisées dans l'algorithme ASCON, en détaillant leur modélisation matérielle ainsi que leur interprétation mathématique.

3.1 L'addition de constante pc

Le module `pc.sv` applique une transformation à l'état $S = \{S_0, S_1, S_2, S_3, S_4\}$ en y ajoutant une constante de ronde, selon la règle suivante :

$$S'_i = \begin{cases} S_i & \text{pour } i \in \{0, 1, 3, 4\} \\ S_2[63:8] \parallel (S_2[7:0] \oplus c_r) & \text{pour } i = 2 \end{cases}$$

où :

- S_i : registre d'entrée,
- S'_i : registre de sortie,
- c_r : constante de ronde d'indice $r =$ `round_i`,
- $S_2[63:8]$: bits de poids fort de S_2 (inchangés),
- $S_2[7:0]$: 8 bits de poids faible de S_2 XORés avec c_r,
- $\parallel$: concaténation.

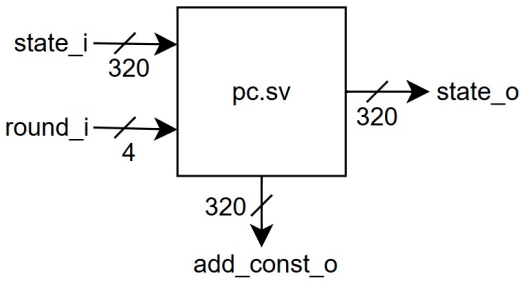

Figure 3.1: Schéma du module `pc.sv` : ajout de constante à l'état

Le module `pc.sv` reçoit en entrée l'état courant `state_i`, ainsi que l'indice de la ronde `round_i`, utilisé pour sélectionner une constante de ronde c_r. Cette constante est injectée uniquement dans le registre S_2, en effectuant un XOR avec ses 8 bits de poids faible. L'état résultant, modifié uniquement sur S_2, est renvoyé en sortie sur le port `state_o`.

En parallèle, la version intermédiaire de l'état après ajout de constante est également transmise sur la sortie `add_const_o`, afin d'être utilisée dans les étapes suivantes de la permutation.

Justification mathématique de l'opération `XOR`

L'opération `XOR` utilisée dans le module `pc.sv` peut être interprétée mathématiquement comme une addition dans le corps fini $\mathbb{F}_2 = \mathbb{Z}/2\mathbb{Z}$. En effet, pour deux bits $a, b \in \{0, 1\}$:

$$a \oplus b = a + b \mod 2$$

Cette propriété s'étend naturellement à des vecteurs de bits. Dans le cadre de l'algorithme ASCON, les états internes sont représentés par des mots de 64 bits, soit des éléments de $\mathbb{F}_2^{64}$.

Dans le module `pc`, seule la composante S_2 de l'état subit une transformation. L'opération appliquée sur ses 8 bits de poids faible est :

$$S_2'[7:0] = S_2[7:0] \oplus c_r = S_2[7:0] + c_r \mod 2$$

où $c_r \in \mathbb{F}_2^8$ est la constante de round. Les 56 bits de poids fort de S_2 restent inchangés. L'état modifié S_2' s'écrit alors comme :

$$S_2' = S_2[63:8] \,\|\, (S_2[7:0] \oplus c_r)$$

Enfin, si l'on représente la constante c_r comme un vecteur de 64 bits étendu par des zéros (noté $\texttt{pad}(c_r) = 0^{56} \| c_r$), on peut réécrire la transformation comme une addition vectorielle dans $\mathbb{F}_2^{64}$:

$$S_2' = S_2 + \texttt{pad}(c_r) \mod 2$$

Ainsi, l'opération `XOR` réalisée dans le module `pc` correspond à une addition dans l'espace vectoriel $\mathbb{F}_2^{64}$, ce qui justifie son usage dans le contexte d'un algorithme cryptographique [1] [3].

3.2 La couche de substitution

Cette étape applique une transformation non linéaire à chaque colonne de l'état S. Elle est essentielle à la sécurité de l'algorithme, car elle introduit la non-linéarité nécessaire pour empêcher toute attaque par analyse différentielle ou linéaire.

Pour chaque colonne $i \in \{0, 1, \ldots, 63\}$ de l'état S :

$$S'[i] = \text{Sbox}(S[i])$$

avec :

- $S[i] = \{S_0[i], S_1[i], S_2[i], S_3[i], S_4[i]\}$: colonne i,
- Sbox : fonction non-linéaire fixe appliquée bit à bit,
- $S'[i]$: colonne substituée dans l'état S'.

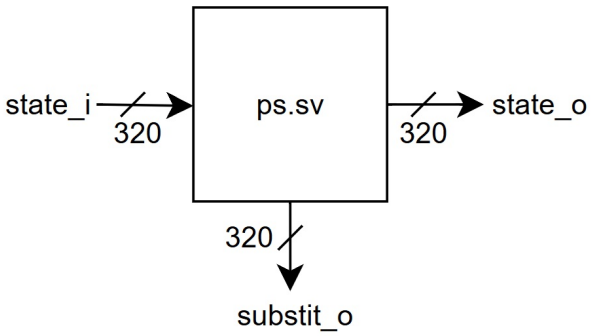

Figure 3.2: Architecture du module `ps.sv` (substitution non-linéaire)

La substitution est implémentée dans une table de correspondance (S-box) appliquée indépendamment à chaque mot. La Table 3.1 montre cette table de substitution.

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

Table 3.1: Table de substitution utilisée dans la couche `ps`

Le module `sbox.sv`, intégré dans `ps.sv`, implémente cette table de correspondance en appliquant la substitution à chaque colonne de l'état.

L'entrée `state_i` représente l'état initial sur 320 bits, découpé en 64 colonnes de 5 bits.

Chaque colonne est alors transformée via la `Sbox`, et le résultat est renvoyé sur la sortie `state_o`, correspondant à l'état après substitution.

En parallèle, une copie de l'état transformé est transmise via la sortie `substit_o`, afin de pouvoir visualiser ou analyser l'impact direct de cette opération sur l'état.

3.3 La couche de diffusion linéaire

La couche de diffusion linéaire, implémentée dans le module `pl.sv`, applique à chaque registre S_i une transformation reposant sur deux rotations circulaires à droite, suivies d'un XOR. Cette étape permet de propager les dépendances entre les bits, ce qui renforce la diffusion de l'information à travers l'état S.

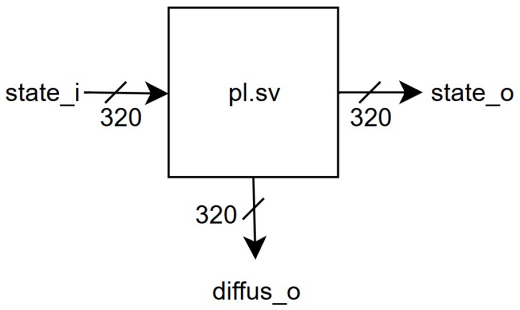

Figure 3.3: Architecture du module `pl.sv` (diffusion linéaire)

La transformation pL est définie par l'expression suivante :

$$S'_i = S_i \oplus (S_i \gg r_1) \oplus (S_i \gg r_2)$$

avec :

- S_i : registre initial (64 bits),
- S'_i : registre après diffusion,
- $\gg r$: rotation circulaire à droite de r bits ($ROTR$),
- (r_1, r_2) : paramètres spécifiques à chaque registre S_i :
  - S_0 : $(19, 28)$,
  - S_1 : $(61, 39)$,
  - S_2 : $(1, 6)$,
  - S_3 : $(10, 17)$,
  - S_4 : $(7, 41)$.

Le module `pl.sv` prend en entrée l'état initial `state_i` sur 320 bits, composé de cinq registres de 64 bits. Il applique à chacun d'eux la transformation de diffusion décrite précédemment. Le résultat de cette opération est transmis en sortie sur `state_o`, représentant l'état modifié après application des rotations et XOR.

3.4 Simulation et validation des résultats

Une simulation a été réalisée à l'aide de ModelSim . L'entrée du module pc correspond à la valeur initiale du tableau ; sa sortie correspond à la ligne Addition de constante, tandis que l'entrée du module ps, qui est l'entrée du module pl correspond à la ligne Substitution S-box.[4]

Ajout de constante avec `pc.sv`

La Figure 3.4 montre que la sortie `add_const_o` obtenue correspond à la valeur d'état attendue après l'injection de la constante dans le registre S_2 :

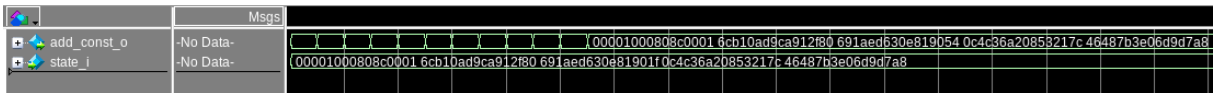

Figure 3.4: Simulation du module `pc.sv` — vérification de `add_const_o`

Substitution non linéaire avec `ps.sv`

La Figure 3.5 compare l'état `state_i` et le résultat de substitution `substit_o`. On observe que la valeur correspond à la sortie attendue par la table S-box. (Voir Figure 8.4 pour le chronogramme complet du module Sbox).

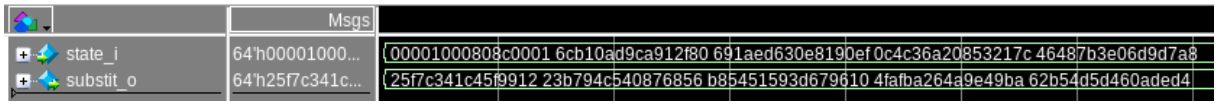

Figure 3.5: Simulation du module `ps.sv` — vérification de `substit_o`

Diffusion linéaire avec `pl.sv`

La Figure 3.6 présente la simulation du module `pl.sv`.

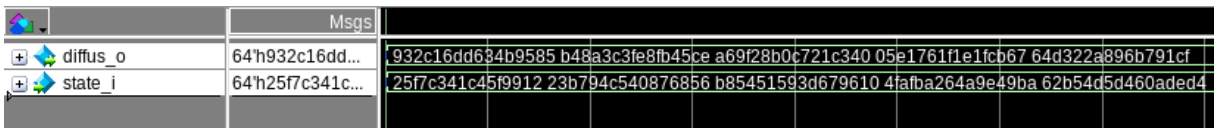

Figure 3.6: Simulation du module `pl.sv` — vérification de `diffus_o`

Ces résultats démontrent que chaque module — `pc.sv`, `ps.sv`, `pl.sv` — respecte fidèlement la spécification de l'algorithme ASCON. On retrouve bien, étape par étape, les états intermédiaires décrits dans la documentation de référence ([4]). La simulation permet ainsi de valider formellement la chaîne de transformation du chiffrement.

4

Architecture des modules de transformation

Ce chapitre détaille l'architecture matérielle des modules de transformation utilisés dans le chemin de chiffrement d'ASCON.

Contrairement aux transformations internes purement algébriques, ces modules intègrent des mécanismes de contrôle et d'interconnexion conditionnelle.

Les modules `xor_begin` et `xor_end` assurent respectivement l'injection des données d'entrée et de la clé dans l'état, en fonction de signaux d'activation.

Les modules de transformation intermédiaire et finale, quant à eux, orchestrent l'enchaînement des permutations internes en combinant les blocs élémentaires de l'algorithme.

Cette structuration modulaire permet d'assurer la flexibilité et la traçabilité de l'état à chaque étape du chiffrement, tout en respectant les contraintes de synchronisation imposées par la machine d'états finis.

4.1 Les modules `xor_begin` et `xor_end`

Les modules `xor_begin` et `xor_end` assurent l'injection conditionnelle des données et de la clé dans l'état interne S, à l'aide de portes `xor` et de multiplexeurs contrôlés par des signaux d'activation. Leur rôle est d'ajouter la non-linéarité nécessaire au chiffrement tout en offrant une certaine flexibilité dans le pipeline de traitement.

Le module `xor_begin` effectue un `xor` entre certains registres de l'état S et les données d'entrée ou la clé. Le tout est contrôlé par des signaux comme `en_xor_data_i` et `en_xor_begin_key_i`.

- Pour $state_i[0]$ et $state_i[1]$, un `xor` est effectué respectivement avec `data_i[63:0]` et `data_i[127:64]` lorsque `en_xor_data_i` est à 1. Le résultat est acheminé vers `state_o[0]` et `state_o[1]`.
- Pour $state_i[2]$ et $state_i[3]$, un `xor` est appliqué avec `key_i[63:0]` et `key_i[127:64]` si `en_xor_begin_key_i` est activé. Les résultats sont dirigés vers `state_o[2]` et `state_o[3]`.

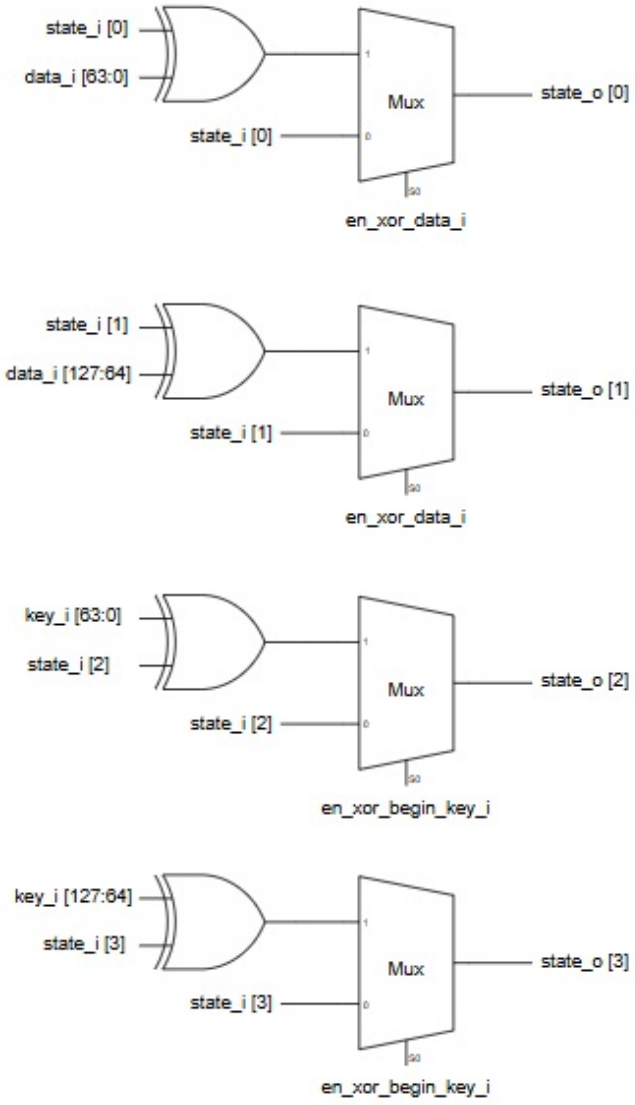

Figure 4.1: Architecture du module `xor_begin`

Le module `xor_end` gère les opérations de fin de permutation. Il réalise un `xor` conditionnel sur les registres S_3 et S_4 à l'aide des signaux `en_xor_end_key_i` et `en_xor_lsb_i`.

- $state_i[3]$ est combiné avec `key_i[63:0]` si `en_xor_end_key_i` est activé, produisant `state_o[3]`.
- $state_i[4]$ est d'abord combiné avec une constante `{en_xor_lsb_i, 0, ..., 0}` pour générer un signal intermédiaire `state_inter4_s`.
- Ensuite, `state_inter4_s` est à nouveau `xoré` avec `key_i[63:0]` sous contrôle de `en_xor_end_key_i` pour produire `state_o[4]`.

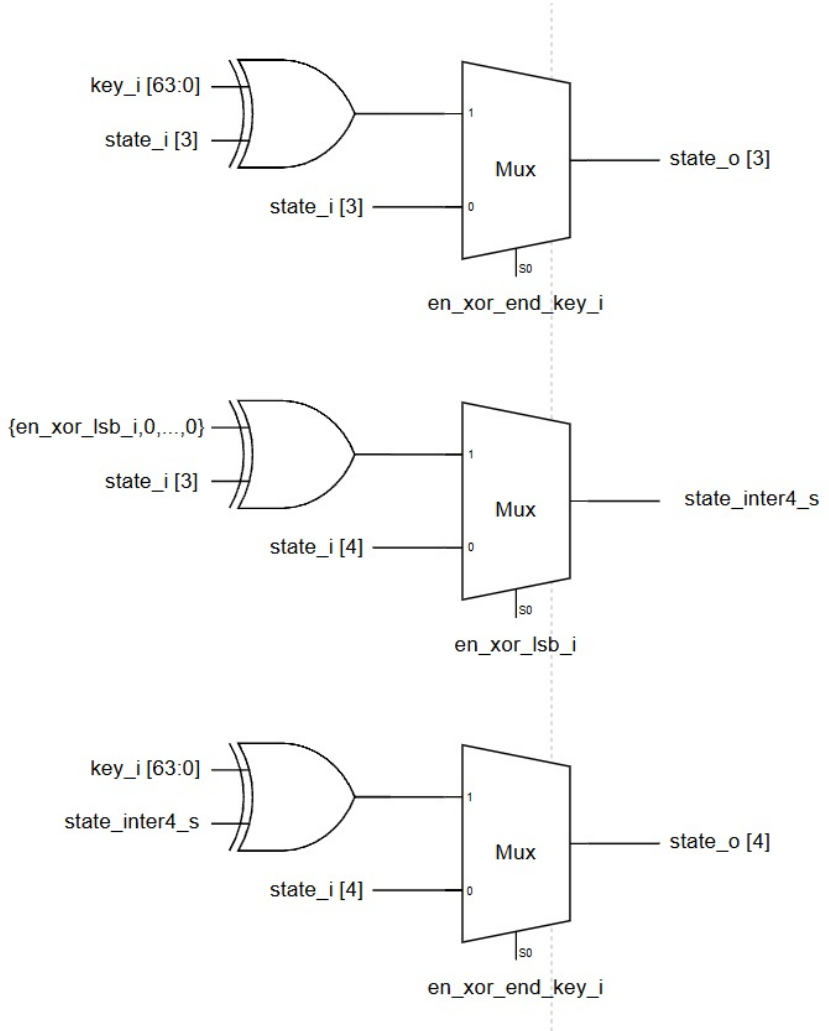

Figure 4.2: Architecture du module `xor_end`

Ces modules assurent l'injection sécurisée et conditionnelle des données dans l'état interne tout en permettant de moduler les opérations de début et de fin de chiffrement.

4.2 Le module `transformation_inter`

Le module `transformation_inter` représente une étape centrale dans le processus de permutation de l'algorithme ASCON-AEAD128. Il applique successivement l'ensemble des transformations élémentaires sur l'état S au sein d'une boucle combinatoire et séquentielle. L'architecture de ce module repose sur un chemin de données bien défini, orchestré par des signaux de contrôle externes.

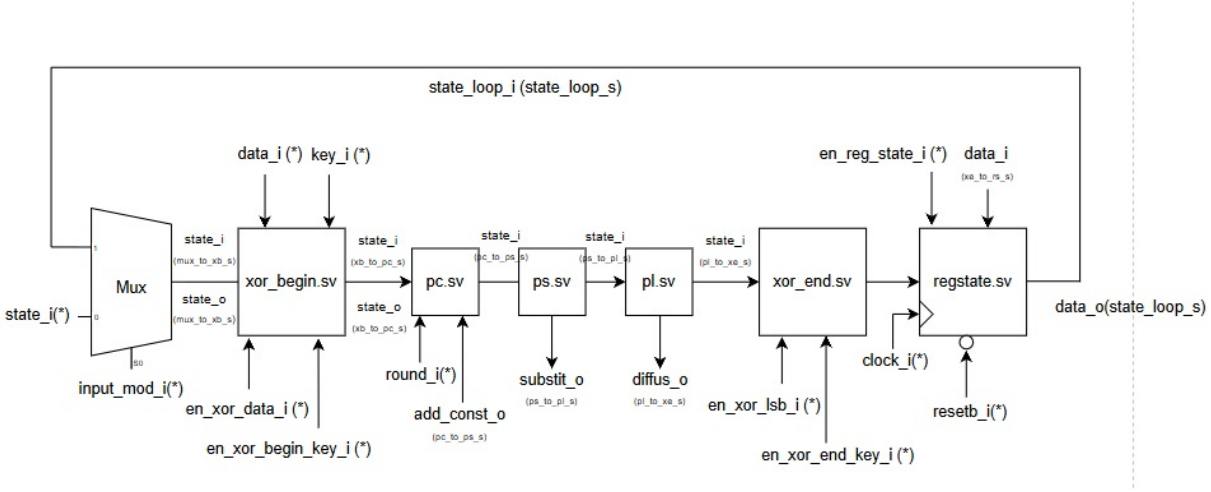

Figure 4.3: Architecture complète du module `transformation_inter`

L'entrée `state_i` est dirigée vers un multiplexeur contrôlé par `input_mod_i`, qui sélectionne soit l'entrée externe soit l'état issu du cycle précédent, nommé `state_loop_s`. Le signal sélectionné (`mux_to_xb_s`) est ensuite traité par le module `xor_begin.sv`, où les données `data_i` et la clé `key_i` peuvent être injectées conditionnellement selon les signaux `en_xor_data_i` et `en_xor_begin_key_i`.

Le résultat de cette opération (`xb_to_pc_s`) entre alors dans le module `pc.sv`, qui applique l'addition de constante de ronde via `round_i` et génère en sortie le fil `add_const_o`. La transformation est ensuite propagée vers `ps.sv`, qui réalise la substitution non linéaire, générant le signal `substit_o`, puis vers `pl.sv`, qui applique la diffusion linéaire en sortie sur le fil `diffus_o`.

L'état est ensuite dirigé vers `xor_end.sv`, qui applique conditionnellement une clé de fin et une constante faible sur `state_i[3]` et `state_i[4]` à l'aide des signaux `en_xor_lsb_i` et `en_xor_end_key_i`. Le résultat est ensuite enregistré par le module `regstate.sv`, sous contrôle du signal `en_reg_state_i`, en fonction de l'horloge `clock_i` et du reset actif bas `resetb_i`.

La boucle est refermée par le signal `state_loop_s`, qui devient l'état d'entrée `state_loop_i` pour l'itération suivante, assurant ainsi le fonctionnement itératif de la permutation.

Ce module combine efficacement logique combinatoire et séquentielle pour implémenter une étape complète du processus de permutation ASCON.

4.3 Le module `transformation_finale`

Le module `transformation_finale` reprend l'enchaînement des opérations élémentaires vues précédemment dans le module `transformation_inter`, mais y ajoute deux blocs supplémentaires destinés à extraire les sorties finales : le texte chiffré et le tag d'authentification.

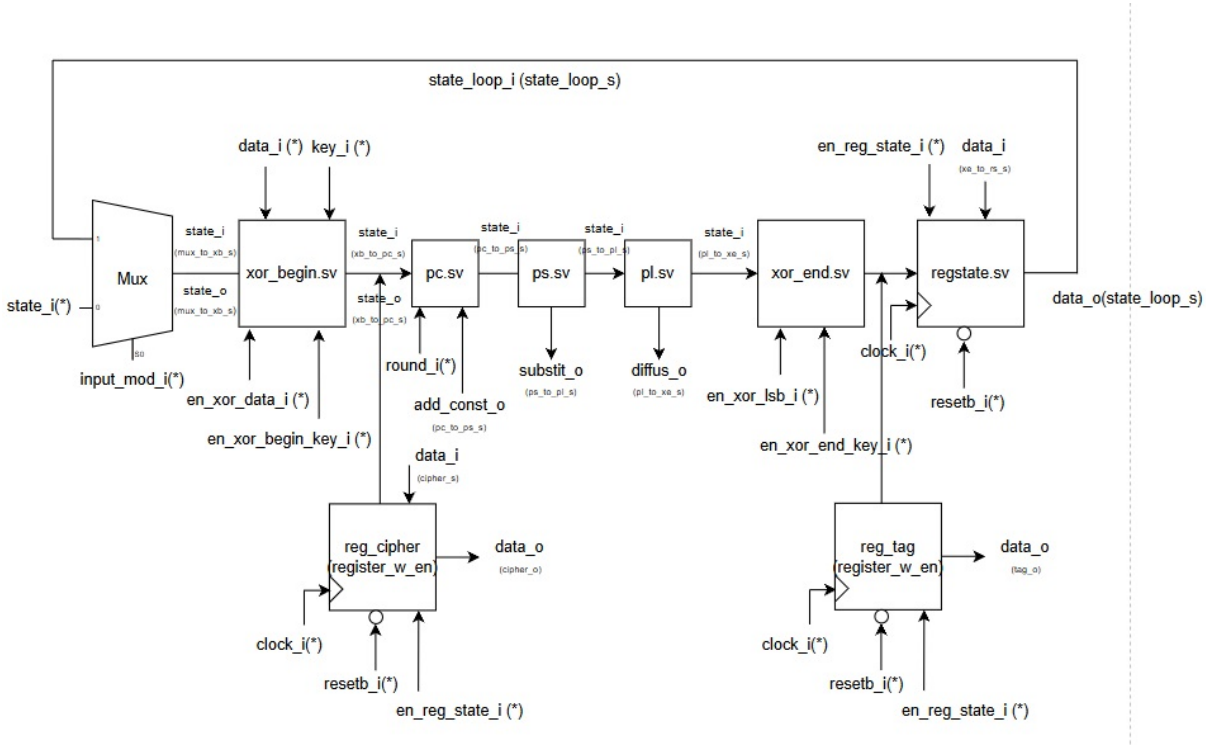

Figure 4.4: Architecture du module `transformation_finale`

Comme pour la transformation intermédiaire, l'état `state_i` est d'abord sélectionné via un multiplexeur selon le signal `input_mod_i`, permettant d'utiliser soit une nouvelle entrée, soit le retour de boucle `state_loop_s` en provenance de `regstate.sv`.

Ensuite, le module `xor_begin.sv` est chargé d'injecter conditionnellement des données via `data_i` ou une clé via `key_i`, en fonction des signaux `en_xor_data_i` et `en_xor_begin_key_i`. L'état ainsi modifié est transmis à `pc.sv` où la constante de ronde `round_i` est ajoutée, et propagée en sortie via `add_const_o`.

La chaîne de transformation continue avec `ps.sv` pour la substitution non linéaire, produisant `substit_o`, puis `pl.sv` pour la diffusion, qui génère le signal `diffus_o`.

Le module `xor_end.sv` termine cette séquence en appliquant les clés finales et éventuellement un bit faible, via les signaux `en_xor_end_key_i` et `en_xor_lsb_i`.

Deux modules de registre sont alors utilisés pour extraire les résultats du chiffrement :

- `reg_cipher` : capture le texte chiffré via le signal `data_o` (alias `cipher_o`) lorsqu'activé par `en_reg_state_i`.
- `reg_tag` : enregistre le tag d'authentification `tag_o` à l'aide du même signal d'activation.

Enfin, le cœur du processus reste la boucle `state_loop_s`, fermée via `regstate.sv` qui assure la mémorisation séquentielle de l'état selon l'horloge `clock_i` et le reset `resetb_i`, avec le contrôle de `en_reg_state_i`.

Ce module garantit l'extraction sécurisée des sorties de l'algorithme tout en assurant la continuité de la permutation selon les règles d'ASCON-AEAD128.

4.4 Simulation et validation des résultats

4.4.1 Modules `xor_begin` et `xor_end`

Les modules `xor_begin` et `xor_end` implémentent les opérations de *XOR conditionnel* respectivement au début et à la fin de l'algorithme ASCON. Afin de vérifier leur bon fonctionnement, une simulation a été réalisée à l'aide d'un testbench dédié.

Chronogramme de simulation du `xor_end` La figure 4.5 présente le chronogramme issu de la simulation du module `xor_end`. Les signaux d'entrée `state_i`, `key_i`, `en_xor_lsb_i`, et `en_xor_end_key_i` ont été initialisés pour déclencher un `XOR` sur les deux derniers mots de l'état.

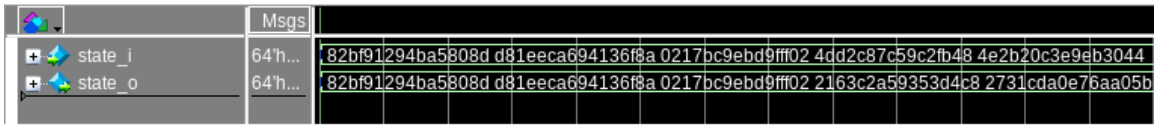

Figure 4.5: Chronogramme de simulation du module `xor_end`

Analyse des résultats L'état initial fourni en entrée est le suivant :

`state_i` = 82BF91294BA5808D D81EECA694136F8A
0217BC9EBD9FFF02 4DD2C87C59C2FB48 4E2B20C3E9EB3044

Après application des signaux de contrôle (`en_xor_end_key_i = 1`), la sortie obtenue est :

`state_o` = 82BF91294BA5808D D81EECA694136F8A
0217BC9EBD9FFF02 2163C2A59353D4C8 2731CDA0E76AA05B

Cette valeur de sortie correspond exactement à la ligne *État XOR donnée A1* décrite dans le sujet ASCON officiel, validant ainsi le comportement attendu du module. [4]

Principe du `xor_begin` Le module `xor_begin` repose sur le même principe de conception. Il effectue des opérations conditionnelles de `XOR` sur les deux premiers mots de l'état avec les données d'entrée, et sur les deux mots suivants avec la clé, selon les signaux de contrôle. La méthode de simulation est similaire, permettant de valider la correspondance avec les lignes *État XOR donnée d'entrée* spécifiées dans le sujet. Voir Figure 8.1 pour le chronogramme complet de ce module.

4.4.2 Modules `transformation_finale`

La transformation finale du système correspond à la dernière étape de permutation ASCON, après le passage par les différents modules (`xor_begin`, `pc`, `ps`, `pl`, etc.).

La figure 4.6 montre la sortie `state_loop_s` obtenue à la fin de la boucle de permutation. (Voir Figure 8.8 pour le chronogramme complet de ce module.)

Figure 4.6: Chronogramme de la sortie finale `state_loop_s`

Analyse des valeurs L'entrée de cette boucle finale correspond à l'état initial fourni dans l'énoncé du sujet ASCON [4]. Après application de toutes les transformations successives, la sortie obtenue est :

$$\texttt{state\_loop\_s} = \text{2d1f984bf2d4fab6 6ae932cc7b93eb54 263a9df175349255}$$
$$\text{4eb3452ce34318db f366f456cb297659}$$

Cette valeur correspond parfaitement à l'état attendu à la fin du processus de permutation, tel qu'il est précisé dans la documentation de référence ASCON [4]. Cela valide l'intégrité et le bon enchaînement des transformations internes de l'architecture.

5

La machine d'états `fsm_moore`

Implémentée sous la forme d'une machine de Moore, la machine d'états `fsm_moore` pilote les signaux de contrôle nécessaires aux différentes phases du processus cryptographique, de l'initialisation jusqu'à la génération finale du `tag` d'authentification.

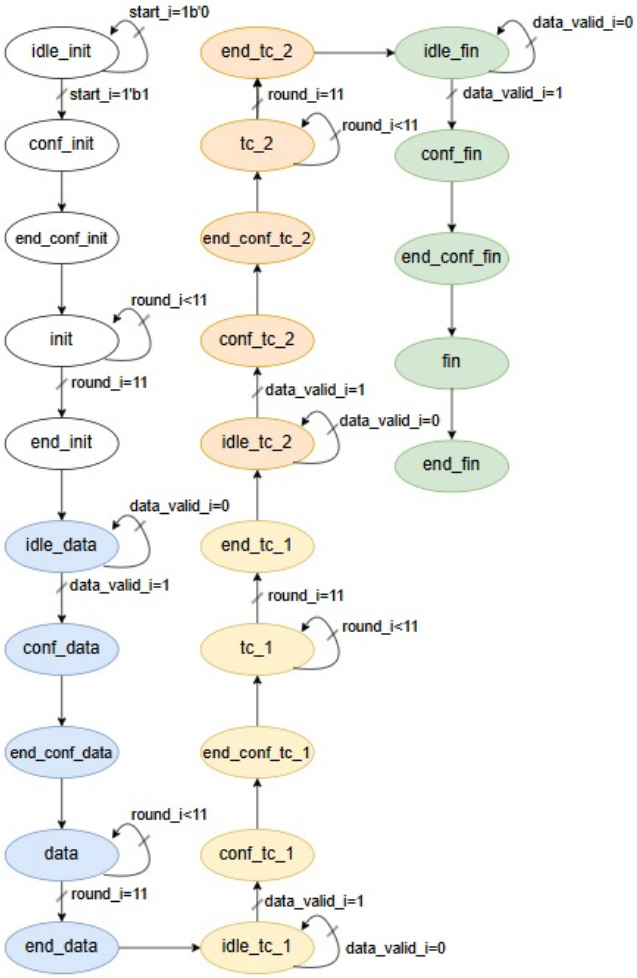

Figure 5.1: Diagramme des états de la machine `fsm_moore`

La machine d'états se divise en plusieurs blocs logiques, correspondant aux grandes étapes du chiffrement :

- **Initialisation** : (`idle_init`, `conf_init`, `end_conf_init`, `init`, `end_init`)
  L'activation de la machine se fait sur le signal `start_i=1`. La constante d'initialisation est injectée dans l'état `S`, suivi d'un enchaînement de permutations contrôlé par le compteur `round_i`.

- **Données associées** : (`idle_data`, `conf_data`, `end_conf_data`, `data`, `end_data`)
  Dès que `data_valid_i=1`, les blocs de données associées sont injectés dans l'état et transformés à l'aide des permutations internes.

- **Texte clair** :

  - Premier bloc : `idle_tc_1` → `conf_tc_1` → `end_conf_tc_1` → `tc_1` → `end_tc_1`
  - Deuxième bloc : `idle_tc_2` → `conf_tc_2` → `end_conf_tc_2` → `tc_2` → `end_tc_2`

  Chaque bloc de texte clair est injecté lorsque `data_valid_i=1`, puis traité en effectuant 12 itérations (jusqu'à `round_i=11`).

- **Finalisation** : (`idle_fin`, `conf_fin`, `end_conf_fin`, `fin`, `end_fin`)
  Cette dernière phase permet la génération du `tag` d'authentification, en reprenant les opérations internes de permutation et de clé.

Les transitions entre états sont déclenchées selon les signaux suivants :

- `start_i` : lancement de la machine.
- `data_valid_i` : présence d'une donnée à injecter.
- `round_i` : progression dans les itérations de permutation.

L'ensemble de ces transitions garantit une progression contrôlée et synchrone du chiffrement, respectant la structure de l'algorithme ASCON-AEAD128.

6

Validation du module finale

Ce chapitre présente l'intégration et la validation du module `ascon_top`, représentant l'architecture de chiffrement complète selon l'algorithme ASCON-AEAD128. Cette structure interconnecte trois blocs essentiels :

- `fsm_moore.sv`, responsable de la gestion des phases de l'algorithme via une machine d'états,
- `compteur_double_init.sv`, qui génère l'indice de ronde,
- `transformation_finale.sv`, qui réalise le traitement cryptographique du message.

6.1 Le module `ascon_top`

La Figure 6.1 illustre la structure globale du module `ascon_top`. On y distingue les différents signaux de contrôle issus de la machine d'états, qui pilotent les sous-modules au cours du processus de chiffrement et d'authentification.

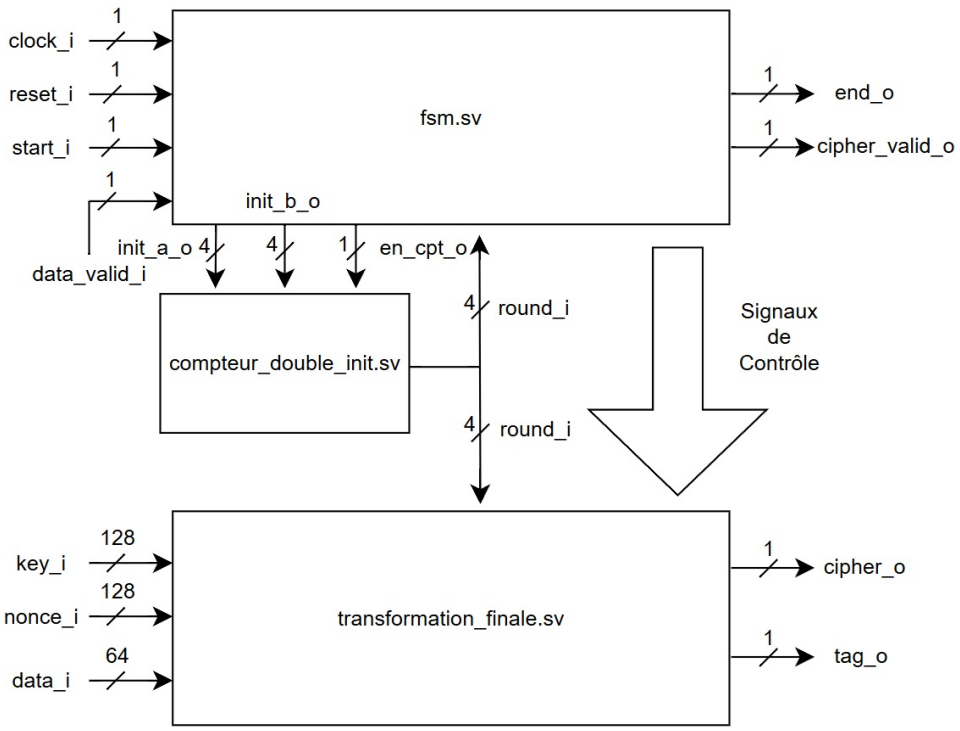

Figure 6.1: Architecture complète du module `ascon_top`

Signaux de contrôle générés par la machine d'états

Le tableau suivant regroupe les principaux signaux de contrôle émis par la `fsm_moore` pour piloter le chiffrement dans le module `transformation_finale` :

Nom du signal	Taille (bits)
en_cpt_o	1
init_a_o	1
init_b_o	1
en_xor_begin_key_o	1
en_xor_data_o	1
en_xor_end_key_o	1
en_xor_lsb_o	1
input_mod_o	1
en_tag_o	1
en_cipher_o	1
en_reg_state_o	1

Table 6.1: Signaux de contrôle entre la `fsm_moore` et le module `transformation_finale`

Ces signaux permettent de gérer dynamiquement les différentes étapes internes du chiffrement : initialisation, injection de la clé, injection des données, génération du tag d'authenticité et stockage des résultats. Le module a été simulé et testé afin de garantir la bonne séquence des opérations au cours de chaque phase de l'algorithme.

6.2 Résultats de simulation du module `ascon_top.sv`

Afin de vérifier le bon fonctionnement du module `ascon_top.sv`, une simulation complète a été réalisée à l'aide de ModelSim. Le test consiste à injecter les entrées prévues dans le sujet (clé, nonce, texte clair, données associées), puis à observer les sorties `cipher_o`, `cipher_valid_o` et `tag_o`.

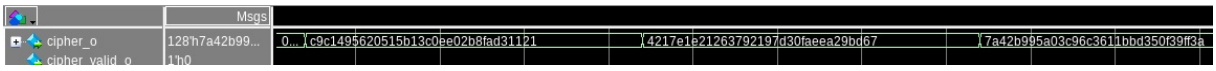

Figure 6.2: Sortie `cipher_o` et signal de validité `cipher_valid_o`

La Figure 6.2 présente les trois blocs de texte chiffré successifs sur le signal `cipher_o`, validés respectivement par le front haut du signal `cipher_valid_o`. Les valeurs observées sont :

- C_1 = `c9c1495620515b13c0ee02b8fad31121`,
- C_2 = `4217e1e21263792197d30faeea29bd67`,
- C_3 = `7a42b995a03c96c3611bbd350f39ff3a`.

Figure 6.3: Sortie du tag d'authentification `tag_o`

La Figure 6.3 montre la sortie du tag d'authentification obtenu à la fin du chiffrement. Sa valeur est :

$$T = \texttt{f366f456cb2976594eb3452ec34318db}$$

Ces résultats sont strictement conformes aux valeurs attendues dans l'énoncé du sujet, ce qui valide l'exactitude fonctionnelle du module `ascon_top.sv` [4]. (Voir Figure 8.8 pour le chronogramme complet de ce module.)

7

Problèmes rencontrés et limites

Problèmes rencontrés

Bien que le projet ait abouti à une implémentation fonctionnelle et validée de l'algorithme ASCON-AEAD128, plusieurs difficultés ont été rencontrées au cours du développement, notamment lors de la modélisation matérielle des modules et de l'intégration globale.

Une première difficulté notable concernait la synchronisation entre les différents modules (`xor_begin`, `pc`, `ps`, `pl`, `xor_end`, etc.). La gestion des signaux d'activation et des transitions d'état dans la machine d'états finis s'est avérée sensible, en particulier pour assurer une coordination fluide des étapes (initialisation, absorption, permutation, finalisation). Plusieurs tests ont été nécessaires pour ajuster précisément les moments d'activation et de mise à jour des registres internes.

Des problèmes spécifiques ont également émergé lors de la construction des modules de permutation intermédiaire et finale. Par exemple, l'intégration séquentielle des blocs dans une boucle de transformation a initialement entraîné des conflits de synchronisation et des incohérences dans les sorties. Une revue du flot de contrôle a permis de corriger ces défauts.

Enfin, la validation des sorties avec les valeurs de référence a nécessité une vigilance particulière sur le format des données, le traitement du padding et l'alignement des blocs dans l'état. Une attention spécifique a été portée à la conformité avec les exemples fournis dans le sujet officiel [4].

Limites et pistes d'amélioration

Plusieurs pistes d'amélioration ont été identifiées pour affiner la robustesse, la lisibilité et l'extensibilité de l'implémentation matérielle :

- La machine d'états actuels pourrait être optimisée en réduisant le nombre d'états nécessaires, notamment en fusionnant certaines phases redondantes ou en exploitant des conditions combinatoires plus fines.
- Le compteur de rondes `compteur_double_init` pourrait être rendu plus générique et paramétrable, pour gérer d'autres configurations (nombre variable de rondes, extensions futures).
- Une version complète du système avec un module de padding générique et un traitement direct du texte ASCII (et non uniquement hexadécimal) pourrait offrir une meilleure lisibilité des résultats.
- Enfin, l'ajout d'un banc de test automatique, comparant chaque sortie à des valeurs de référence issues de la spécification, renforcerait la fiabilité et faciliterait la maintenance du projet.

Ces évolutions permettraient d'enrichir le projet tout en le rendant plus proche d'une implémentation matérielle exploitable dans un contexte contraint.

8

Conclusion

Ce projet a constitué une immersion concrète dans le domaine de la cryptographie matérielle, et plus particulièrement dans l'implémentation bas niveau de l'algorithme ASCON, récemment standardisé pour le chiffrement léger.

Il a permis de franchir une étape significative dans la compréhension de l'architecture d'un système cryptographique embarqué, tout en consolidant la maîtrise du langage `SystemVerilog` ainsi que des outils de simulation et de validation associés.

Au-delà de l'acquisition de compétences techniques, ce travail a mis en lumière les défis spécifiques liés à la transposition matérielle d'un algorithme conçu à l'origine pour une exécution logicielle. Concevoir des modules fonctionnels, gérer les contraintes de synchronisation et raisonner en termes de cycles, registres et chemins critiques ont été des étapes clés de l'apprentissage.

Si l'architecture globale développée répond aux spécifications du protocole ASCON, certaines optimisations restent envisageables. En particulier, la machine d'états pourrait être simplifiée par l'intégration d'un compteur de blocs, et ses performances améliorées par le remplacement de la machine de Moore actuelle par une machine de Mealy.

En définitive, ce projet m'a permis d'acquérir une vision claire et structurée du processus de conception matérielle en cryptographie. Il ouvre des perspectives intéressantes pour des travaux futurs, notamment dans l'optimisation d'architectures sécurisées ou l'analyse de leur robustesse face aux attaques physiques.

Bibliography

[1] L. Dutertre, *Mathématiques pour la cryptographie*, École des Mines de Saint-Étienne. https://www.emse.fr/~dutertre/documents/math_crypto.pdf

[2] ASCON, *Official website of the ASCON authenticated encryption algorithm*. https://ascon.isec.tugraz.at/

[3] D. Matei et al., *Secure Lightweight Authenticated Encryption: Final Report on the Development of the ASCON Family of Algorithms*, 2022. https://hal.science/hal-03596732/document

[4] ASCON, *Sujet ASCON - Projet de cryptographie*, École des Mines de Saint-Étienne.https://ecampus.emse.fr/pluginfile.php/83942/mod_resource/content/6/sujet_ascon.pdf

[5] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, *Ascon v1.2. Submission to the NIST Lightweight Cryptography Competition*, Technical Report, TU Graz, 2019.

[6] National Institute of Standards and Technology, *Ascon-based Lightweight Cryptography Standards for Constrained Devices*, FIPS PUBS 232, U.S. Department of Commerce, Washington, D.C., 2024.

Annexes : Chronogrammes complets de simulation

Cette annexe regroupe les chronogrammes complets issus de la simulation des différents modules du projet ASCON. Ces résultats ont été obtenus à l'aide de l'outil ModelSim et permettent de visualiser précisément les transformations effectuées sur les signaux internes du système.

Chronogramme du module `xor_begin.sv`

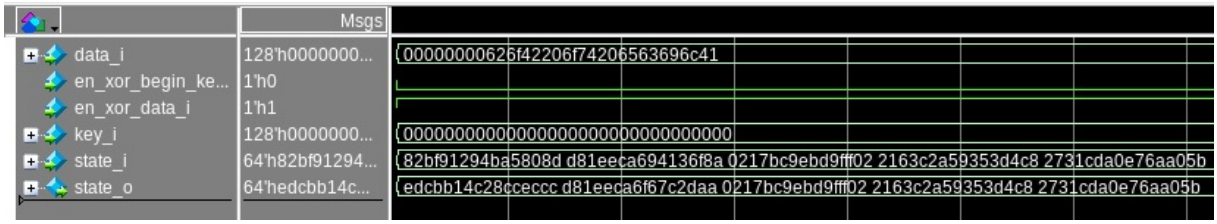

Figure 8.1: Chronogramme du module `xor_begin` : application conditionnelle de XOR avec les données et la clé

Chronogramme du module `xor_end.sv`

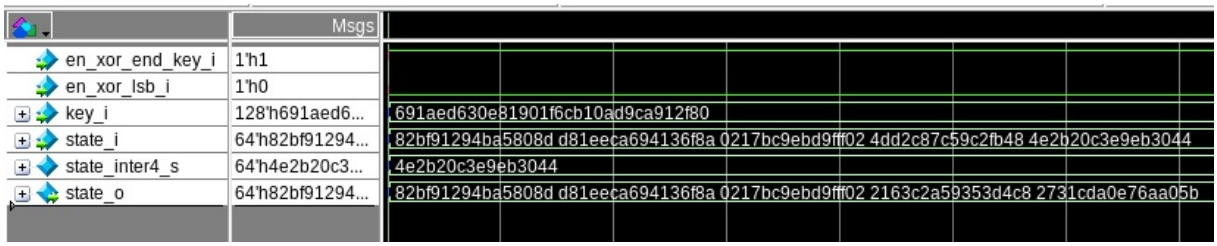

Figure 8.2: Chronogramme du module `xor_end` : fin du chiffrement avec padding et XOR

Chronogramme du module `pc.sv`

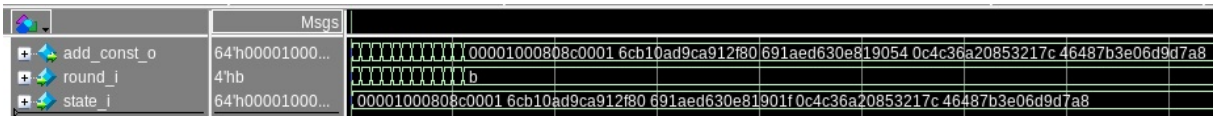

Figure 8.3: Chronogramme du module `pc` : injection de la constante de round

Chronogramme du module `ps.sv`

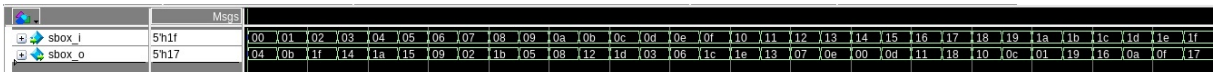

Figure 8.4: Chronogramme de simulation de la S-box — vérification des 32 valeurs de substitution

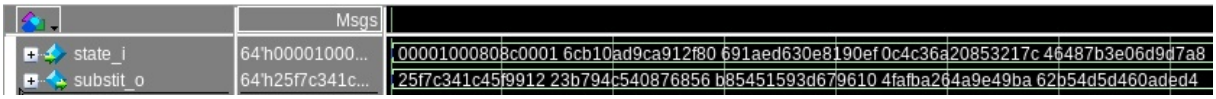

Figure 8.5: Chronogramme du module `ps` : substitution non-linéaire via S-box

Chronogramme du module `pl.sv`

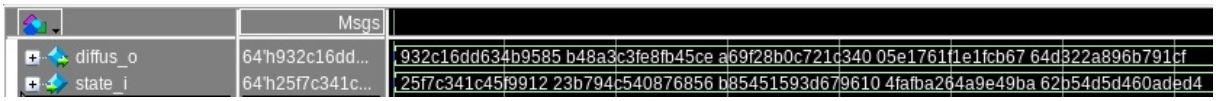

Figure 8.6: Chronogramme du module `pl` : diffusion linéaire par rotations et XOR

Chronogramme du module `transformation_simple.sv`

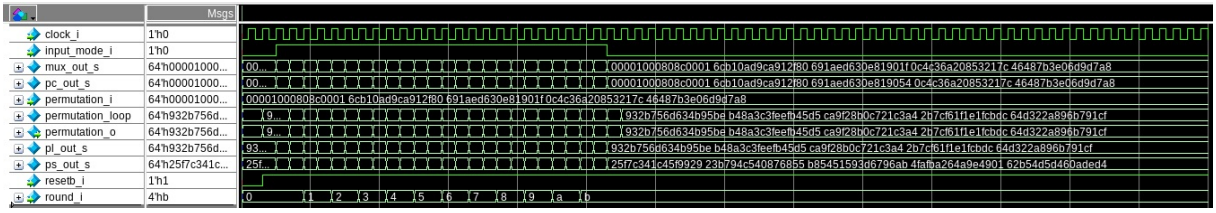

Figure 8.7: Chronogramme du module `transformation_simple` : boucle PC → PS → PL

Chronogramme du module `transformation_finale.sv`

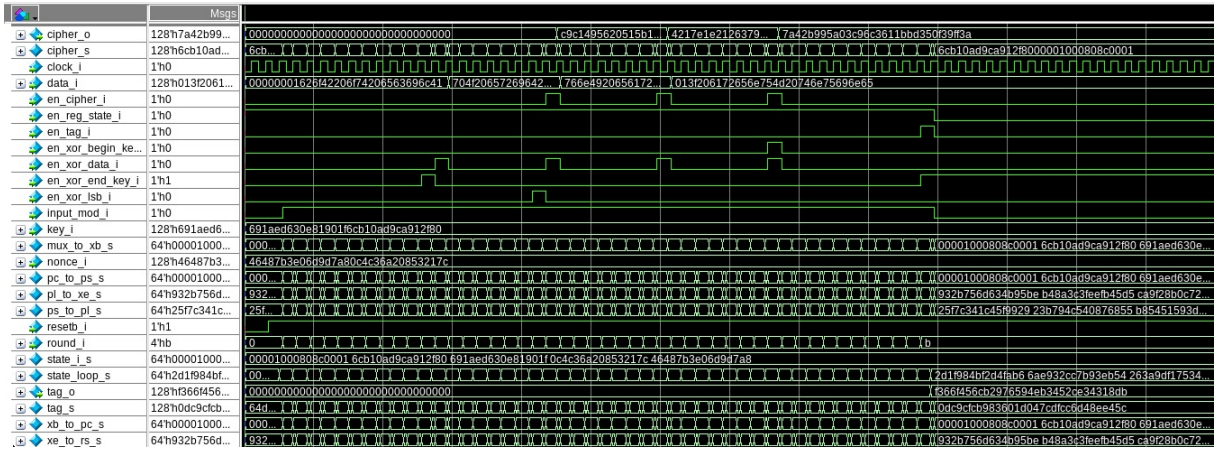

Figure 8.8: Chronogramme du module `transformation_finale` : assemblage des modules de transformation

Chronogramme du module principal `ascon_top.sv`

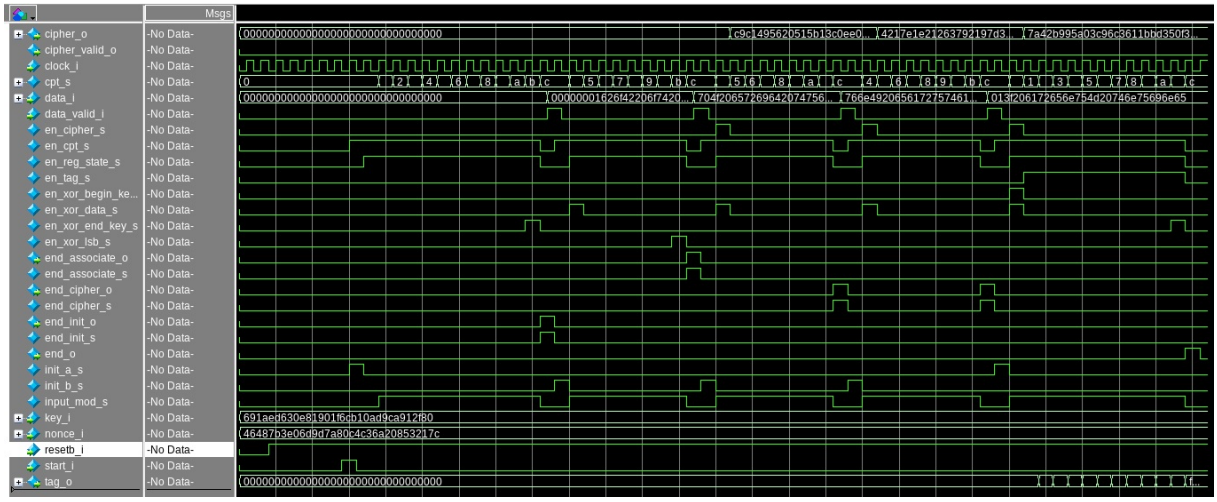

Figure 8.9: Chronogramme du module `ascon_top` : vue globale du fonctionnement complet de l'algorithme