Projet PCSN

Conception d'un Système Numérique

Étudiant : Sami BENNANE

Professeur : Jean-Baptiste RIGAUD

Année : 2024–2025

Table des matières

1 Introduction

1.1 Présentation du projet

Dans une époque où a confidentialité des données est un enjeu clé avec le développement du numérique, la cryptographies'est développée en conséquence. Elle repose sur l'envoi de message chiffrés en respectant trois conditions : **la confidentialité**, **l'intégrité**, c'est-à-dire que les données ne sont pas altérées, et **l'authenticité** qui consiste en la garantie du bon expéditeur. La méthode de chiffrement étudiée dans le cadre de ce rapport est une version simplifiée de **l'algorithme Ascon-AEAD128**. Avec ce procédé, si une personne, Alice, décide d'envoyer un message à une autre personne, Bob, sans qu'un éventuel espion comme Eve puisse avoir accès à son contenu, elle devra lui fournir plusieurs éléments : **la clé K**, **la nonce N**, **le texte chiffré C**. Avec ces trois mots, Bob sera en mesure de calculer un **tag T** pour vérifier l'authenticité du message ; s'il correspond au tag qu'Alice lui a envoyé, elle est vérifiée. Cette introduction est une présentation générale non détaillée de l'algorithme Ascon-AEAD128. Nous nous engagerons dans une étude plus approfondie à travers ce rapport en évoquant plusieurs composants indispensables à cette méthode de chiffrement sur le langage de description **SystemVerilog**, comme **la permutation**, **la machine d'état**, **le compteur de rondes**, etc.

1.2 Etat S

Les données manipulées sont des ensembles de cinq mots de 64 bits. Sur le schéma ci-dessous, ces cinq mots sont représentés par S_0, S_1, S_2, S_3 et S_4. La représentation de cet objet de 320 bits en `SystemVerilog` est établie dans un module appelé `ascon_pack.sv` fourni par monsieur Rigaud.
Sur le schéma ci-dessous, on a par exemple `state_s[0][0]` qui est le bit représenté par le rectangle bleu clair, `state_s[1][1]` en orange, `state_s[2][2]` en vert et `state_s[3][63]` en rouge.

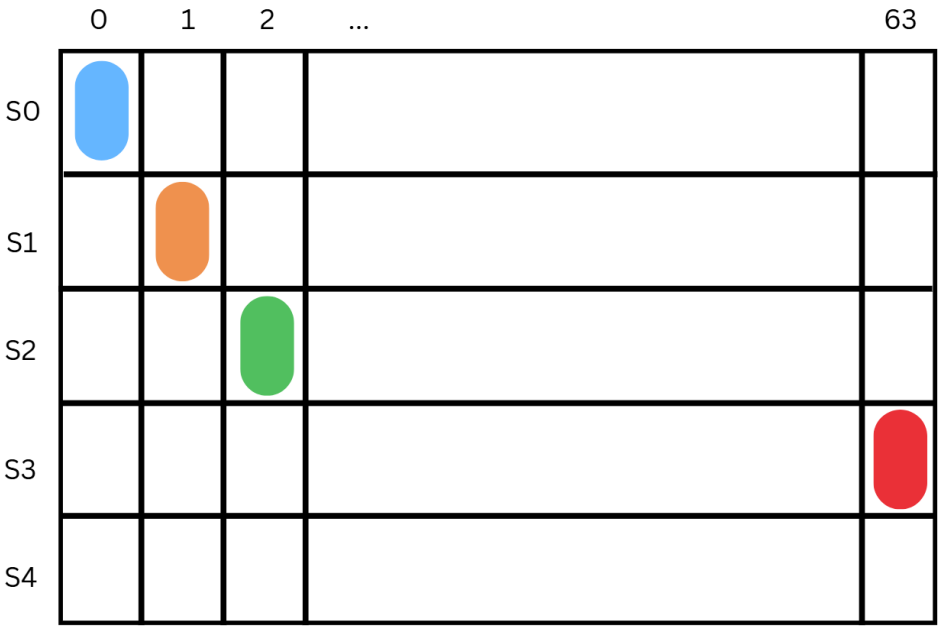

FIGURE 1 – Structure de l'état S

2 Architecture globale du système

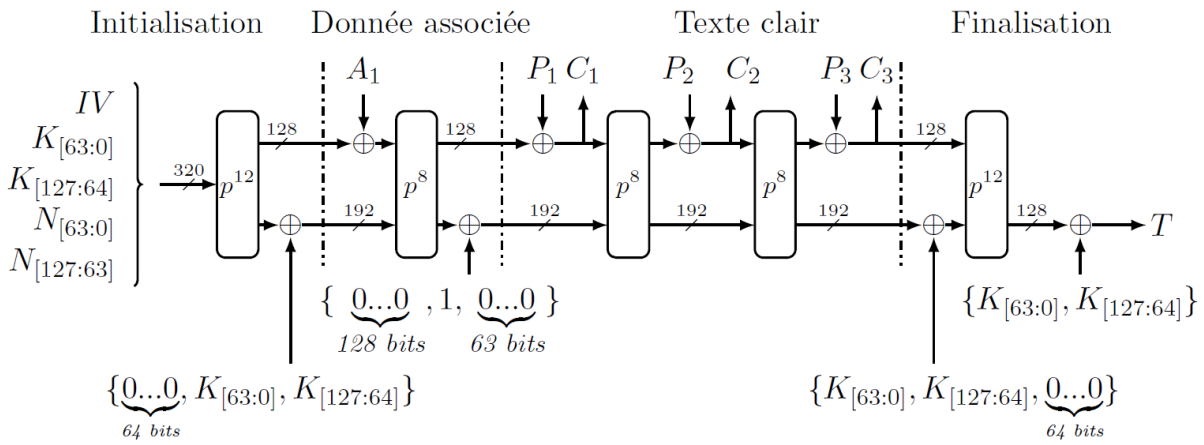

FIGURE 2 – Structure de l'algorithme de chiffrement Ascon-AEAD128.

La nature de l'entrée de l'algorithme étudié est un mot de **320 bits** de type `type_state`, comme celui qui vient d'être présenté. Le premier mot, `IV`, est le vecteur d'initialisation. Les deuxième et troisième mots représentent la clé de 128 bits et les quatrième et cinquième mots sont la `nonce` de 128 bits également. Le processus est composé de deux opérations identifiables : la permutation, répétée huit fois ($\mathbf{p^8}$) ou douze fois ($\mathbf{p^{12}}$) et les opérations `XOR` avant ou après ces dernières. Dans un premier temps, nous nous intéresserons à la structure de la permutation seule, puis nous verrons comment intégrer les opérations `XOR` en SystemVerilog. Enfin, nous nous intéresserons à des éléments non présents sur ce schéma mais qui interagissent avec les opérations qu'il contient, comme la machine d'états ou le compteur de rondes. A chaque fois qu'il y aura un diagramme d'achitecture, les utilités des signaux entrants et sortants seront expliqués lors de l'analyse du testbench.

3 Présentation des opérations élémentaires

3.1 Addition de constante

Il s'agit d'effectuer l'opération `XOR` entre le troisième mot, le mot `S2`, et une constante. Cette dernière dépend de la valeur de la ronde dans laquelle elle est impliquée. Le module `pc.sv` est donné dans le répertoire `"modules"`. Voici le diagramme d'architecture du module associé :

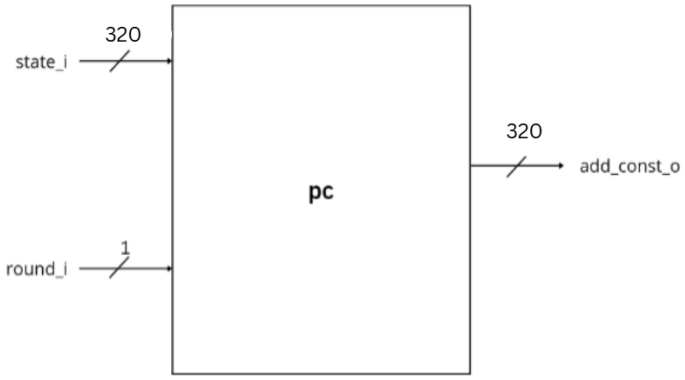

FIGURE 3 – Diagramme d'architecture du module `pc.sv`

Ci-dessous, se trouve le résultat de la simulation Modelsim associé au testbench de cette opération :

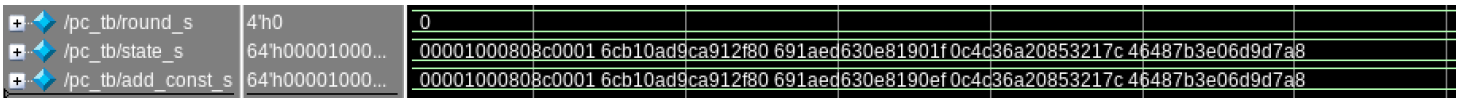

FIGURE 4 – testbench du module `pc.sv`

On y voit figurer les variables suivantes :

— `round_s = 0` : le numéro de la ronde,
— `state_s` : la valeur d'entrée, de type `typestate` (mot de 320 bits),
— `add_const_s` : la valeur de sortie, c'est-à-dire l'état une fois l'opération effectuée.

Le sujet fournit un tableau associant à chaque numéro de ronde une valeur de constante :

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 1 – Constantes c_r utilisées dans les permutations p^8 et p^{12}

Dans ce cas, la constante utilisée est `0000000000000000f0`. On peut alors comparer la valeur attendue, également fournie dans le sujet et encadrée en rouge, avec celle obtenue lors de la simulation :

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

FIGURE 5 – Valeur attendue extraite du sujet pour l'addition de constante à la ronde 0

Elles concordent, le module est fonctionnel

3.2 Couche de substitution

Pour bien comprendre cette opération, il convient de considérer l'état S comme un ensemble de 64 colonnes de cinq bits. Chacune de ces colonnes forme alors un mot de cinq bits : la colonne i correspond au mot $\{S_0[i], S_1[i], S_2[i], S_3[i], S_4[i]\}$.
Le rôle de la couche de substitution est de transformer ce mot de cinq bits en un autre mot de cinq bits, conformément au tableau de substitution fourni dans le sujet :

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 2 – Table de substitution S-box utilisée pour le projet

Le répertoriage des transformations indiquées par ce tableau est implémenté dans le module `sbox.sv`.
Par exemple, pour les valeurs suivantes formant le mot `00110` en binaire, soit 6 :

- $S_0[0] = 0$
- $S_1[0] = 0$
- $S_2[0] = 1$
- $S_3[0] = 1$
- $S_4[0] = 0$

On obtiendra les nouvelles valeurs suivantes qui forment 9 en binaire car Sbox(6)=9 :

- $S_{0\text{new}}[0] = 0$
- $S_{1\text{new}}[0] = 1$
- $S_{2\text{new}}[0] = 0$
- $S_{3\text{new}}[0] = 0$
- $S_{4\text{new}}[0] = 1$

Voici le diagramme d'architecture de la couche de substitution :

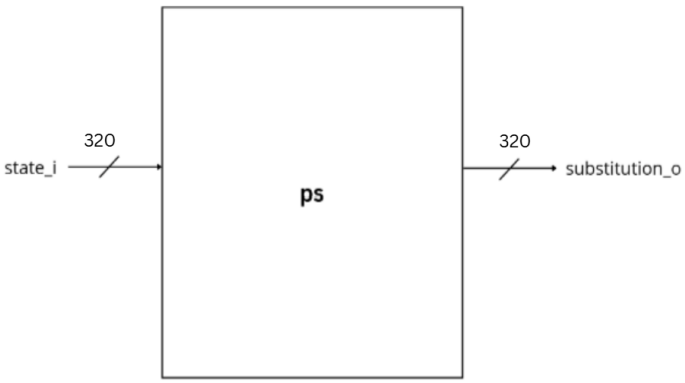

FIGURE 6 – Diagramme d'architecture du module `ps.sv`

On applique la couche de substitution à la valeur obtenue en sortie de l'addition de constante au paragraphe précédent, ci-dessous se trouve le testbench associé :

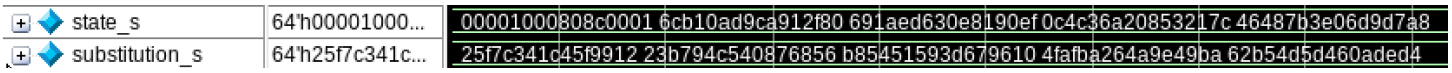

FIGURE 7 – testbench du module `ps.sv`

On y voit figurer les valeurs suivantes :

- `state_s` : la valeur d'entrée, qui correspond à la valeur de sortie du paragraphe précédent,
- `substitution_s` : la valeur de sortie, c'est-à-dire l'état une fois l'opération effectuée.

Voici la valeur attendue, encadrée en rouge :

```
-- Permutation (r=00)
Addition constante : 00001000808c0001 6cb10ad9ca912f80 691aed630e8190ef 0c4c36a20853217c 46487b3e06d9d7a8
Substitution S-box : 25f7c341c45f9912 23b794c540876856 b85451593d679610 4fafba264a9e49ba 62b54d5d460aded4
```

FIGURE 8 – Valeur attendue extraite du sujet pour la couche de substitution à la ronde 0

Les deux valeurs concordent, le module est fonctionnel.

3.3 Diffusion linéaire

La dernière opération consiste à effectuer une opération `XOR` entre un des registres de 64 bits et plusieurs rotations cycliques. Une rotation cyclique de N bits est notée par le symbole $\ggg N$. Les opérations en question sont les suivantes :

$$
\begin{aligned}
S_0 &\leftarrow \Sigma_0(S_0) = S_0 \oplus (S_0 \ggg 19) \oplus (S_0 \ggg 28) \\
S_1 &\leftarrow \Sigma_1(S_1) = S_1 \oplus (S_1 \ggg 61) \oplus (S_1 \ggg 39) \\
S_2 &\leftarrow \Sigma_2(S_2) = S_2 \oplus (S_2 \ggg 1) \oplus (S_2 \ggg 6) \\
S_3 &\leftarrow \Sigma_3(S_3) = S_3 \oplus (S_3 \ggg 10) \oplus (S_3 \ggg 17) \\
S_4 &\leftarrow \Sigma_4(S_4) = S_4 \oplus (S_4 \ggg 7) \oplus (S_4 \ggg 41)
\end{aligned}
$$

Voici le diagramme d'architecture de la diffusion linéaire :

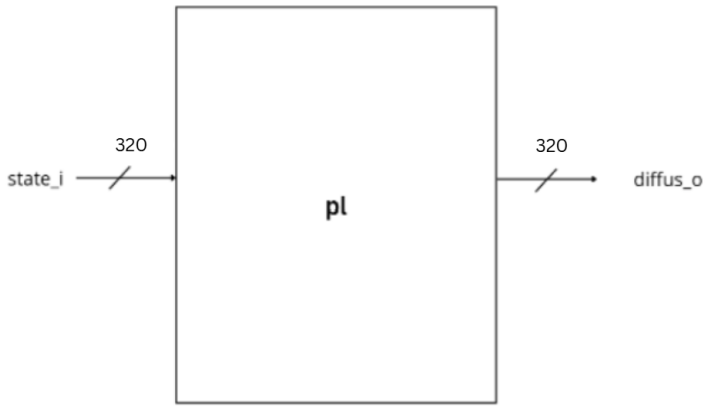

FIGURE 9 – Diagramme d'architecture du module `pl.sv`

De la même manière, on reprend la valeur obtenue avec la couche de substitution et on lui applique la diffusion linéaire afin de vérifier le bon fonctionnement du module à l'aide d'un testbench :

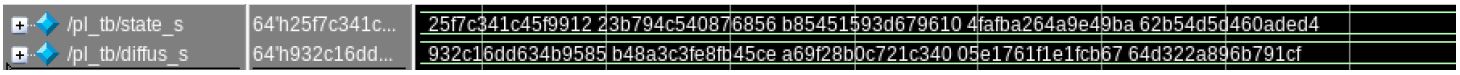

FIGURE 10 – testbench du module `pl.sv`

On y voit figurer les variables suivantes :

- `state_s` : la valeur d'entrée qui correspond à la sortie du paragraphe précédent,
- `diffus_s` : la valeur de sortie, après la diffusion linéaire.

Ci-dessous se trouve la valeur attendue, encadrée en rouge :

```
-- Permutation (r=00)
Addition constante : 00001000808c0001 6cb10ad9ca912f80 691aed630e8190ef 0c4c36a20853217c 46487b3e06d9d7a8
Substitution S-box : 25f7c341c45f9912 23b794c540876856 b85451593d679610 4fafba264a9e49ba 62b54d5d460aded4
Diffusion linéaire : 932c16dd634b9585 b48a3c3fe8fb45ce a69f28b0c721c340 05e1761f1e1fcb67 64d322a896b791cf
```

FIGURE 11 – Valeur attendue extraite du sujet pour la diffusion linéaire à la ronde 0

Les deux valeurs concordent, le module est fonctionnel.

4 Structure de la permutation

La réalisation du chiffrement a été effectuée de manière itérative afin d'obtenir un module permutation capable de réaliser le chiffrement tout en respectant les consignes de l'utilisateur. Ce module a été développé en trois étapes : la permutation simple, la permutation intermédiaire et la permutation finale.

4.1 Permutation simple

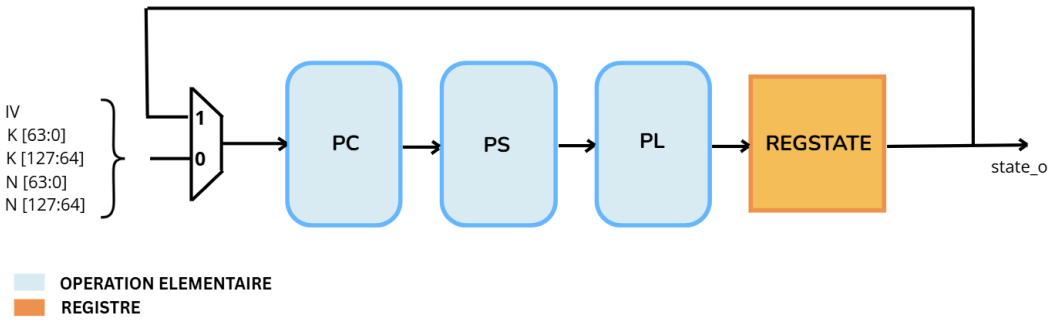

FIGURE 12 – Schéma architecturel du module `permutation_simple.sv`

Sur le schéma ci-dessus, deux possibilités s'offrent pour l'entrée. Celle-ci est sélectionnée en fonction du signal de contrôle du multiplexeur. Le mot d'entrée de 520 bits fourni par l'utilisateur correspond à la première possibilité ; il n'est utilisé qu'une seule fois au cours du processus de chiffrement.

Une fois ce mot injecté, les opérations élémentaires lui sont appliquées. Un registre est ensuite utilisé pour mémoriser la valeur du mot en sortie des trois opérations élémentaires. Cette valeur correspond à la sortie `permutation_simple_o`, qui constitue l'entrée du module lors de la ronde suivante. Il s'agit de la deuxième possibilité d'entrée que le multiplexeur peut sélectionner et qui est celle qui sera quasiment systématiquement utilisé.
Nous pouvons dresser le diagramme d'architecture de la permutation simple en y répertoriant les entrées et les sorties :

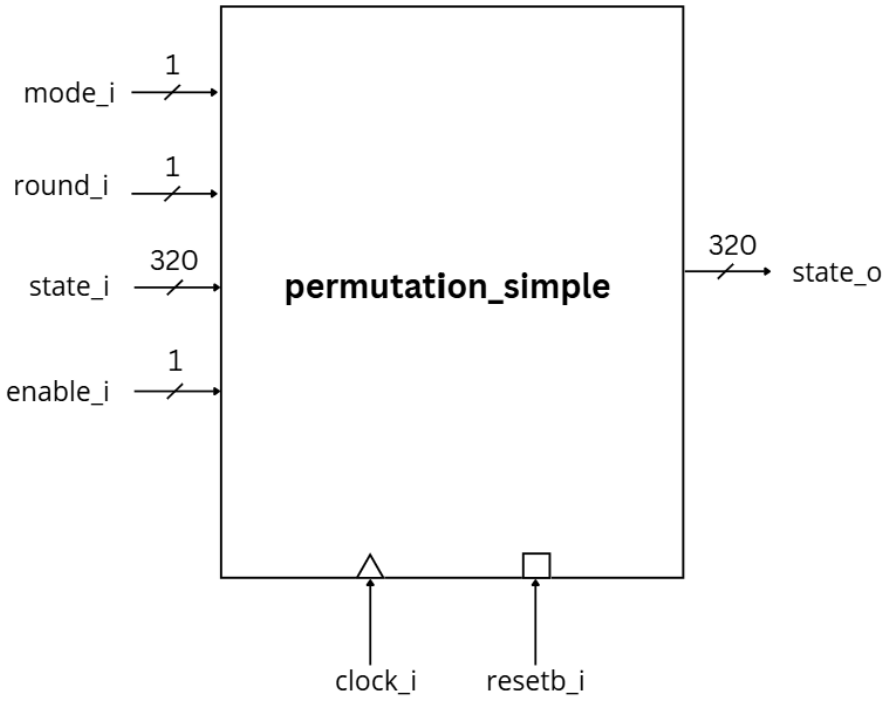

FIGURE 13 – Diagramme d'architecture de la permutation simple

Vérifions la validité du module `permutation_simple` à travers un testbench qui agit sur la phase d'initialisation :

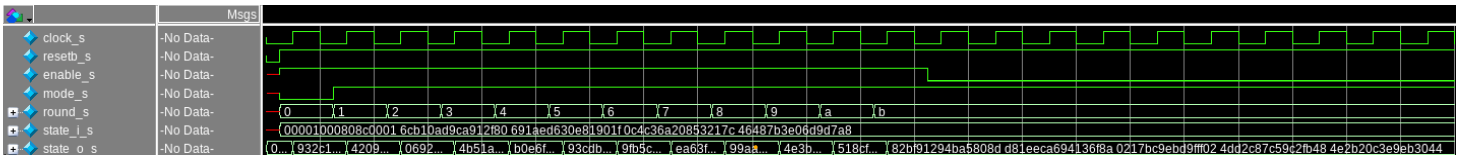

FIGURE 14 – test bench du module `permutation_simple.sv`

On y voit figurer les valeurs suivantes :

- `state_i_s` : la valeur d'entrée
- `resetb_s` : réinitialise tous les signaux quand sa valeur est à 0.
- `state_o_s` : la valeur de sortie
- `enable_s` : active le registre quand sa valeur est à 1.
- `round_s` : la valeur de la ronde en cours
- `mode_s` : dirige l'entrée vers la boucle quand sa valeur est à 1.
- `clock_s` : l'horloge.

Le sujet nous fournit les valeurs attendues, qui concordent à chaque ronde de permutation. À titre d'exemple, nous pouvons vérifier la dernière valeur, celle de la ronde `0xB` :

```
82bf91294ba5808d d81eeca694136f8a 0217bc9ebd9fff02 4dd2c87c59c2fb48 4e2b20c3e9eb3044
```

Cette valeur, donnée dans le sujet, est identique à celle obtenue dans le testbench, qui est portée par `state_o_s` .

4.2 Permutation intermédiaire

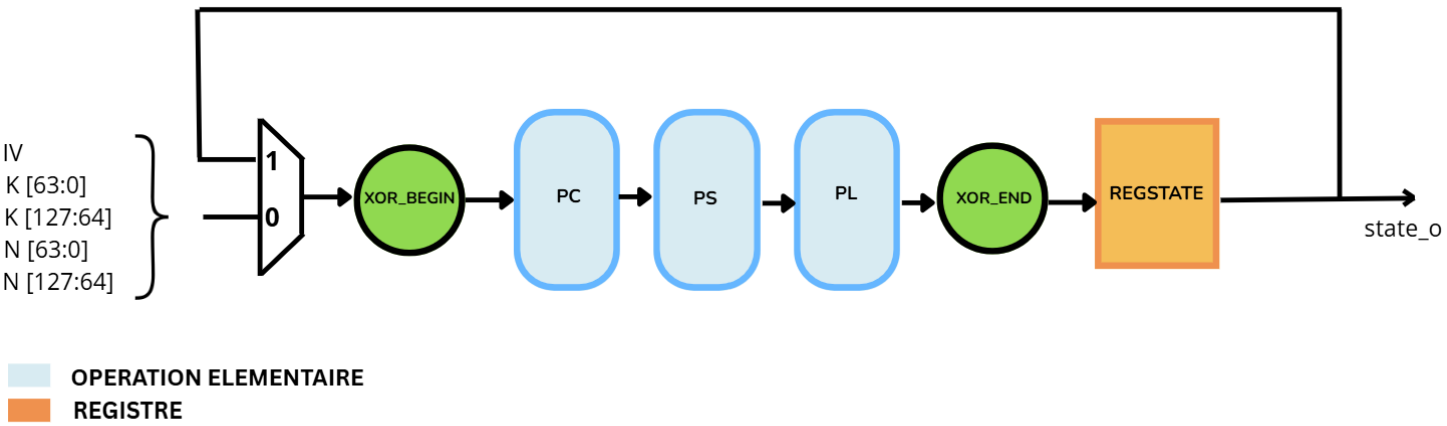

FIGURE 15 – Schéma architecturel du module `permutation_intermediaire.sv`

D'un point de vue interne au module, la **permutation intermédiaire** fonctionne exactement de la même manière que la **permutation simple**, à l'exception près que des opérations de XOR sont ajoutées.

Le XOR_BEGIN du schéma est activé **avant** un enchaînement de huit ou douze permutations, tandis que le XOR_END est activé **à la fin** de celui-ci. Il y a deux types de XOR_BEGIN et deux types de XOR_END, ils diffèrent selon la valeur avec laquelle l'opération est réalisée.

Du point de vue de l'architecture globale, le schéma ci-dessous présente un exemple avec un XOR_BEGIN, ainsi qu'un XOR_END (c'est la phase de données associées) :

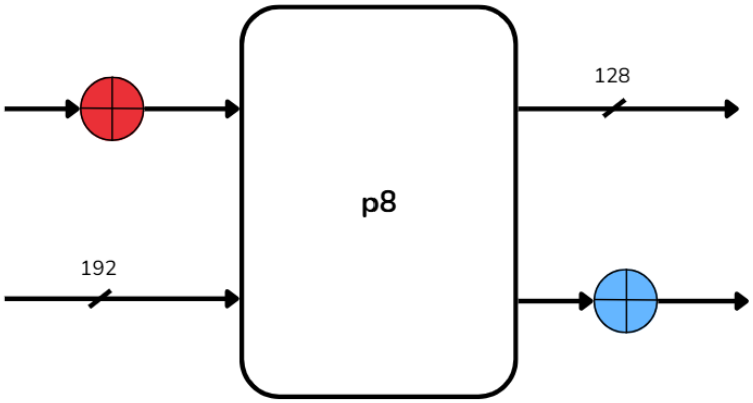

FIGURE 16 – Représentation d'un XOR_BEGIN et d'un XOR_END

Une fois le module permutation_intermediaire.sv créé, c'est-à-dire avec les instanciations des modules xor_begin.sv et xor_end.sv, on obtient le diagramme d'architecture suivant :

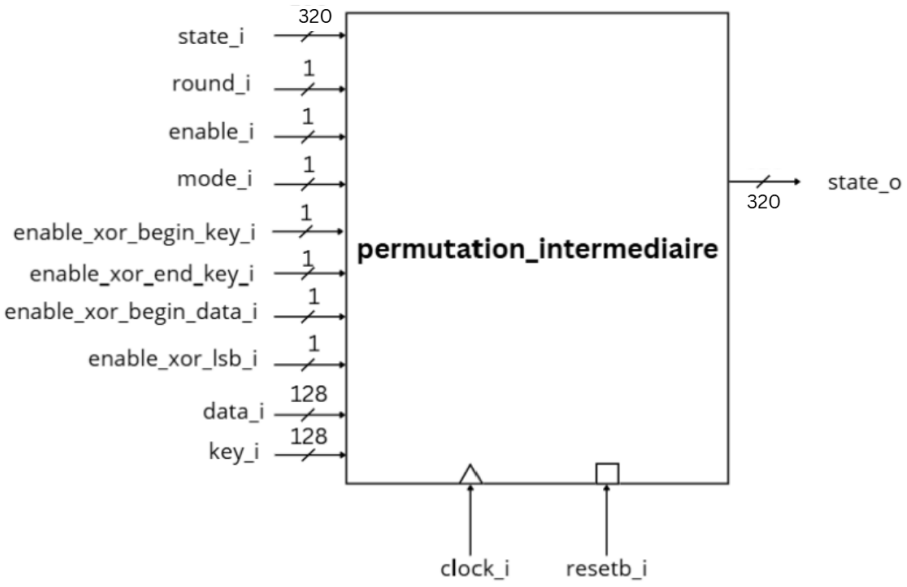

FIGURE 17 – Diagramme d'architecture de la permutation intermediaire

Ci-dessous, nous obtenons les résultats de la simulation du testbench du module de la permutation intermédiaire :

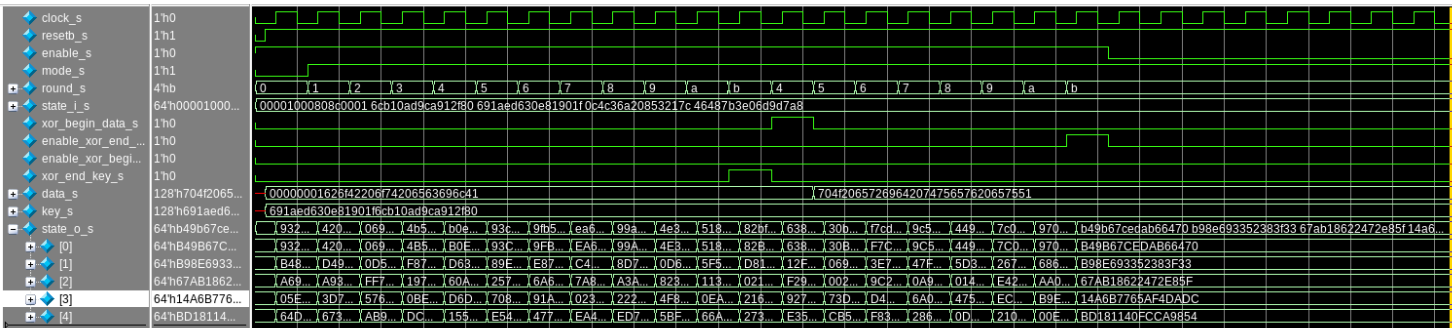

FIGURE 18 – test bench du module `permutation_intermédiaire.sv`

Ce testbench a été effectué pour l'initialisation jusqu'à la fin de la phase de données associées. Les valeurs de sortie à chaque permutation (`state_o_s` ici) concordent une à une avec les valeurs de l'énoncé.

On y voit figurer les valeurs suivantes :

- `state_i_s` : la valeur d'entrée
- `resetb_s` : réinitialise tous les signaux quand sa valeur est à 0.
- `state_o_s` : la valeur de sortie
- `enable_s` : active le registre quand sa valeur est à 1.
- `xor_begin_data_s` : Active le premier type de `XOR_BEGIN`.
  (Il aurait été préférable d'ajouter un `enable_` devant le nom ; inattention de ma part.)
- `enable_xor_begin_key_s` : Active le deuxième type de `XOR_BEGIN`.
- `xor_end_key_s` : Active le premier type de `XOR_END`.
  (Il aurait été préférable d'ajouter un `enable_` devant le nom ; inattention de ma part.)
- `enable_xor_end_lsb_s` : Active le deuxième type de `XOR_END`.
- `round_s` : Contient la valeur de la ronde en cours.
- `mode_s` : dirige l'entrée vers la boucle quand sa valeur est à 1.
- `clock_s` : l'horloge.
- `data_s` : recueille la valeur de A1, P1, P2 ou P3 selon la situation, cf Figure 2.
- `key_s` : la clé.

À titre d'exemple, nous pouvons vérifier la dernière valeur, obtenue juste après que le `XOR_END` associé au signal de contrôle `enable_xor_end_lsb_s` a été activé. La valeur du sujet, ci-dessous encadrée en rouge, coïncide bien avec la valeur obtenue dans le testbench, portée par `state_o_s` :

```
Permutation (r=11) : b49b67cedab66470 b98e693352383f33 67ab18622472e85f 14a6b7765af4dadc 3d181140fcca9854
État ^ (0...0 & 1) : b49b67cedab66470 b98e693352383f33 67ab18622472e85f 14a6b7765af4dadc bd181140fcca9854
```

FIGURE 19 – Valeur attendue extraite du sujet de la donnée associée

4.3 Permutation finale

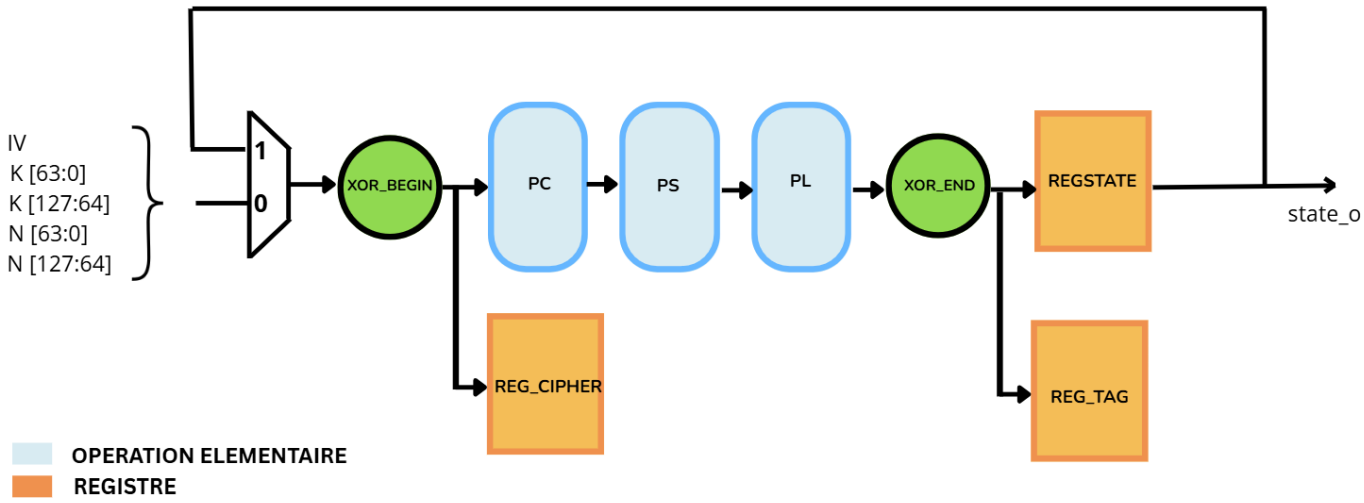

FIGURE 20 – Schéma architecturel du module `permutation_finale.sv`

La permutation intermédiaire et la permutation finale ne sont pas très différentes, à la différence près qu'il y a deux registres : un registre pour le *cipher* et un pour le *tag*.

Le *cipher* recueille les deux premiers bits en sortie du `XOR_BEGIN` lorsque son signal de contrôle est activé, tandis que le *tag* recueille les deux derniers bits du `XOR_END` lorsque son propre signal de contrôle est activé.

On a le diagramme d'architecture suivant :

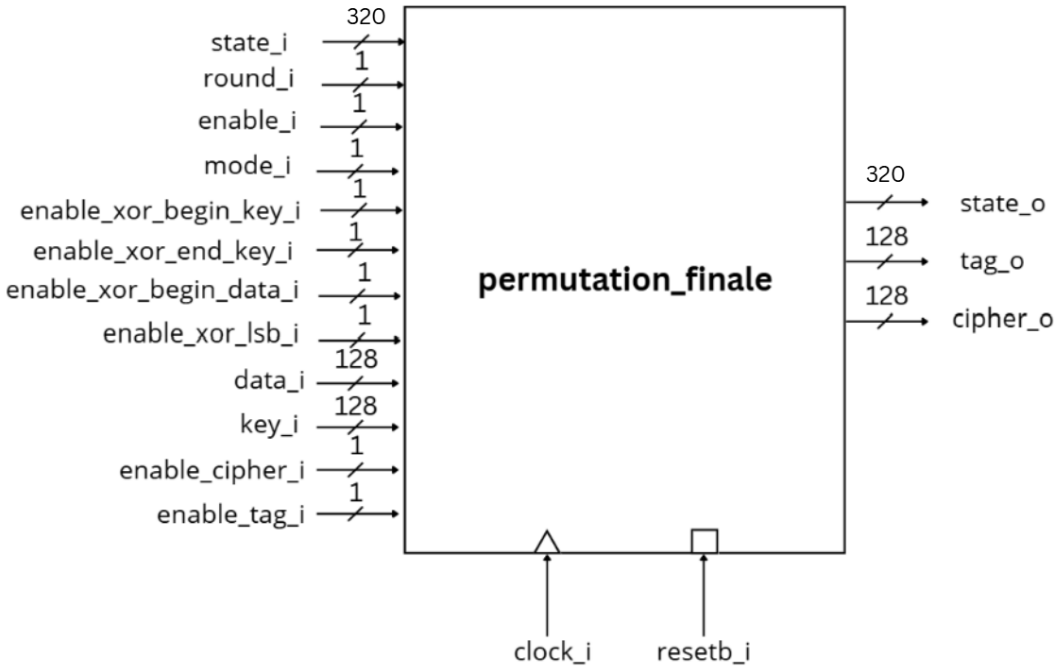

FIGURE 21 – Diagramme d'architecture de la permutation finale

Le testbench de la permutation finale doit être en mesure de tester les éléments intégrés, à savoir le registre du *cipher* et le registre du *tag*.

On choisit alors de le faire agir à partir de la deuxième permutation de la phase du texte clair, ce qui permet de vérifier la valeur du troisième *cipher*, jusqu'au *tag*.

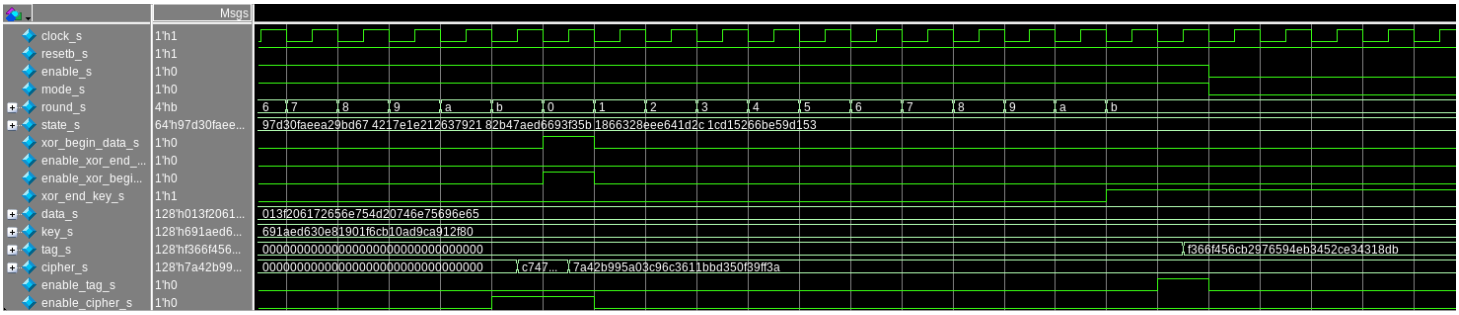

FIGURE 22 – test bench du module `permutation_finale.sv`

On y voit figurer les valeurs suivantes :

- `state_s` : la valeur d'entrée
- `resetb_s` : réinitialise tous les signaux quand sa valeur est à 0.
- `enable_s` : active le registre quand sa valeur est à 1.
- `xor_begin_data_s` : active le premier type de `XOR_BEGIN`.
- `enable_xor_begin_key_s` : active le deuxième type de `XOR_BEGIN`.
- `xor_end_key_s` : active le premier type de `XOR_END`.
- `enable_xor_end_lsb_s` : active le deuxième type de `XOR_END`.
- `round_s` : la valeur de la ronde en cours
- `mode_s` : dirige l'entrée vers la boucle quand sa valeur est à 1.
- `clock_s` : l'horloge.
- `data_s` : recueille la valeur de A1, P1, P2 ou P3 selon la situation, cf Figure 2.
- `key_s` : la clé.
- `enable_cipher_s` : active le registre du cipher quand sa valeur est à 1.
- `enable_tag_s` : active le registre du tag quand sa valeur est à 1.
- `tag_s` : tag.
- `cipher_s` : cipher.

Cette fois, il n'y a pas la valeur de sortie à chaque ronde dans le testbench, ce n'est pas indispensable, nous pouvons vérifier si le testbench est correct en regardant les valeurs du tag et du cipher.
Le troisième cipher est bon (`cipher_s`), il correspond à la valeur du sujet :

```
-- Texte chiffré C3 = 0x 42B995A03C96C3611BBD350F39FF3A
```

Il en est de même pour le tag (`tag_s`), le testbench fonctionne correctement.

5 Machine d'états

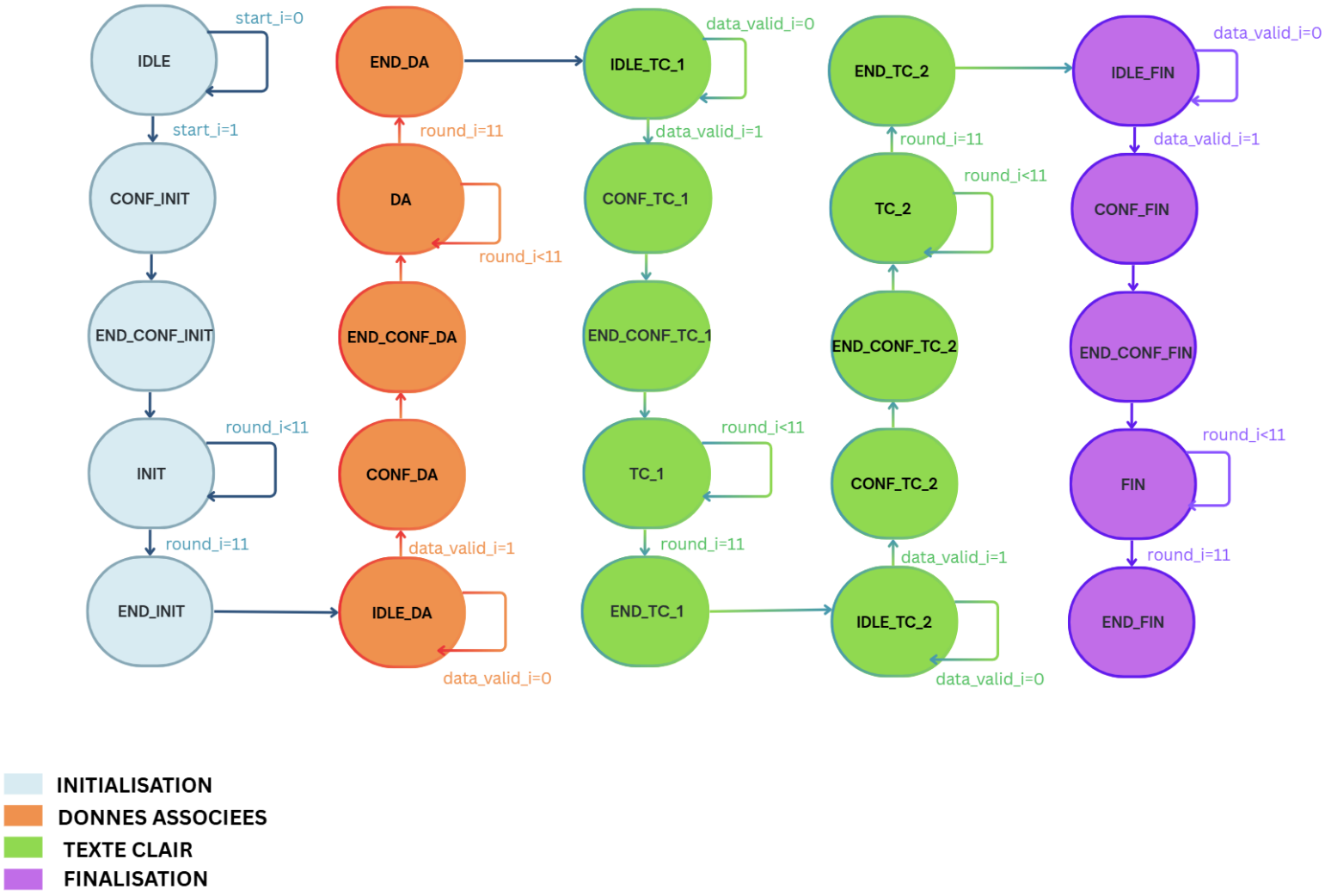

FIGURE 23 – Diagramme d'états

La machine d'états présentée est de type **Moore**. En effet, les transitions entre états dépendent bien de certaines entrées (`start_i`, `round_i`, `data_valid_i`), mais les **sorties sont uniquement déterminées par l'état courant**, et non par les entrées directement. En d'autres mots, chaque état produit toujours les mêmes sorties, ce qui justifie la nature de cette machine de Moore.

Cette machine d'états permet de contrôler le comportement de la permutation finale, en lui indiquant notamment si elle doit entamer une permutation de huit ou de douze rondes, grâce au module `compteur_de_rondes.sv`.

Elle pilote également l'activation des différents blocs fonctionnels, tels que les modules de `XOR`, le registre *cipher* ou encore le registre *tag*, selon l'état courant du système.
Le schéma suivant résume, de manière générale, les dépendances entre les modules :

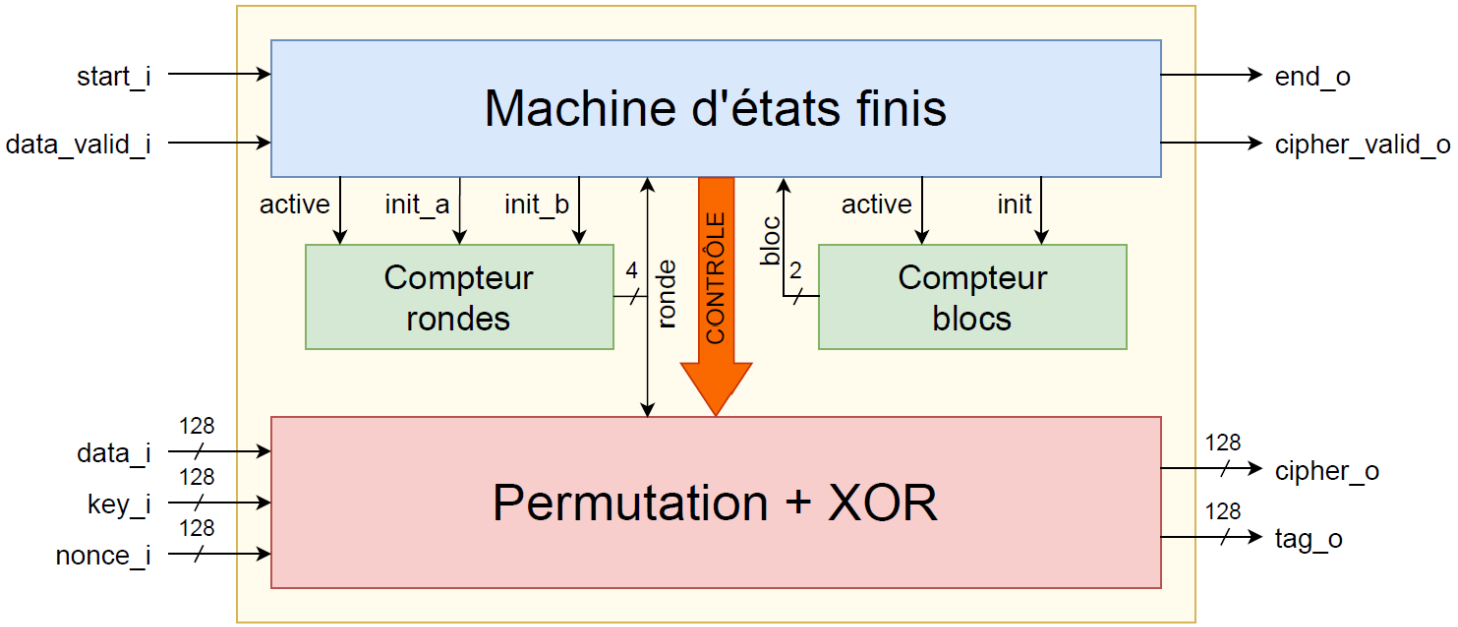

FIGURE 24 – Diagramme d'architecture du module `ascon_top.sv`

Pour avoir une idée plus précise des signaux d'entrée et de sortie de la machine d'états, il est nécessaire d'établir un diagramme d'architecture :

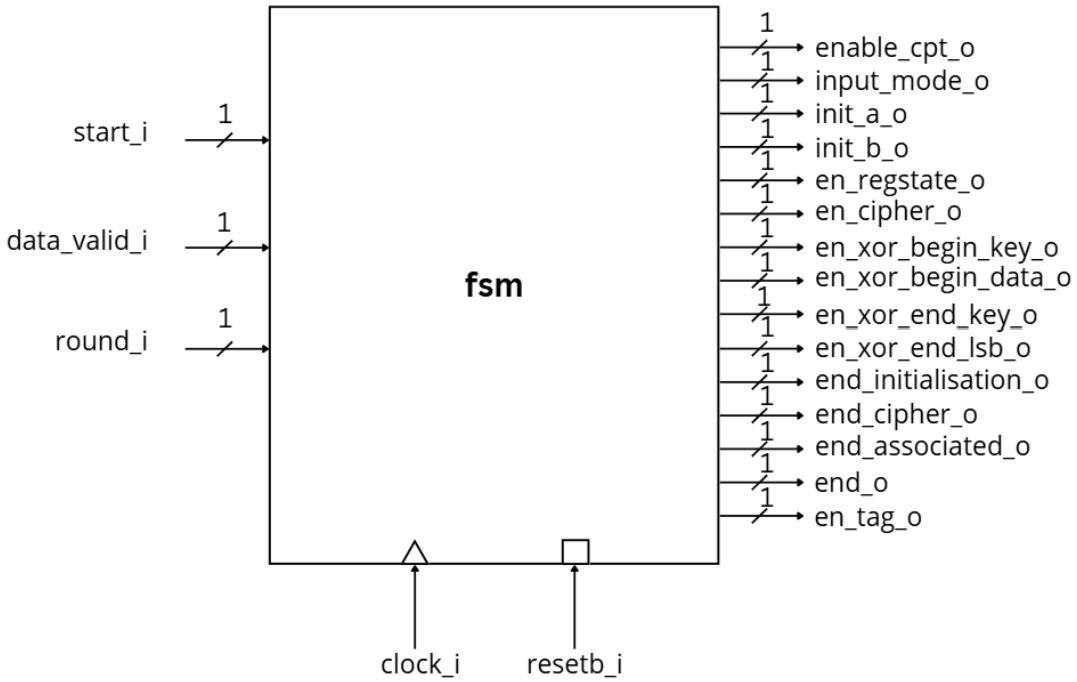

FIGURE 25 – Diagramme d'architecture de la machine d'états

Les utilités de ces signaux d'entrée et de sortie sera expliqué lors de l'analyse du testbenchh du module `ascon_top.sv`. Cette machine d'états interagit avec le compteur de rondes en

lui indiquant s'il s'agit d'effectuer huit ou douze rondes, respectivement à travers les signaux `init_b_i` et `init_a_i`, à condition que `en_i` soit activé.

Une valeur indiquant le numéro de la ronde, `cpt_o` est alors envoyée à la machine d'états, ce qui lui permet notamment de passer d'un état à un autre, comme par exemple de `TC_1` à `END_TC_1` lorsque cette valeur vaut 11. Elle est également transmise au module de la permutation afin que celui-ci effectue le bon nombre de rondes.

Cette description correspond au diagramme d'architecture suivant :

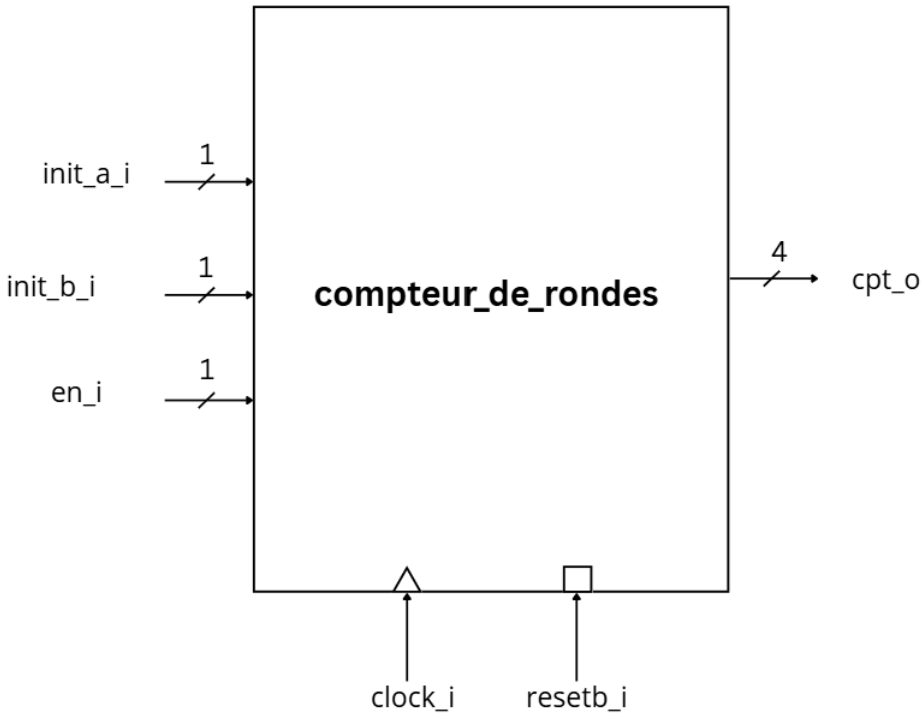

FIGURE 26 – Diagramme d'architecture du compteur de rondes

Il est nécessaire d'établir un module général qui englobe tous les modules présentés ci-dessus (à l'exception du compteur de blocs, qui n'est pas indispensable). Il s'agit du module `ascon_top.sv`.
Après avoir implémenté le module de la machine d'états, fsm.sv, nous sommes en mesures de mettre en place un testbench. Voici le résultat :

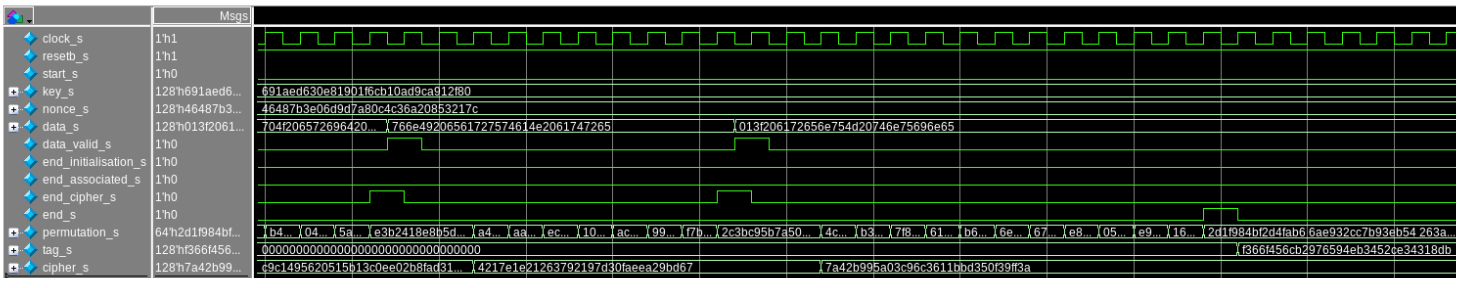

FIGURE 27 – test bench du module `ascon_top.sv`

On y voit figurer les valeurs suivantes :

- `resetb_s` : réinitialise tous les signaux quand sa valeur est à 0.
- `permutation_s` : valeur de sortie à chaque ronde (pas indispensable de l'afficher).
- `mode_s` : dirige l'entrée vers la boucle quand sa valeur est à 1.
- `clock_s` : l'horloge.
- `end_s` : indique que l'algorithme est fini quand sa valeur est à 1.
- `end_initialisation_s` : indique que l'initialisation est finie quand sa valeur est à 1.
- `end_cipher_s` : indique que la phase de chiffrement est finie quand sa valeur est à 1.
- `end_associated_s` : indique que la phase de données associées est finie quand sa valeur est à 1.
- `data_s` : recueille la valeur de A1, P1, P2 ou P3 selon la situation, cf Figure 2.
- `data_valid_s` : indique la présence d'une donnée valide quand sa valeur est à 1.
- `key_s` : la clé.
- `nonce_s` : nonce
- `tag_s` : le tag.
- `cipher_s` : le cipher.

Les trois valeurs du cipher (`cipher_s`) ainsi que la valeur du tag (`tag_s`) correspondent à celles attendues dans le sujet. Le module `ascon_top.sv` fonctionne donc correctement.

6 Conclusion

La réalisation de ce projet a été majoritairement bénéfique pour moi sur deux points : il m'a permis de découvrir le fonctionnement d'un algorithme de chiffrement en détails et m'a permis de développer des compétences dans un langage de programmaation que je ne connaissais pas : le Systemverilog. De plus, c'était une occasion de mettre en pratique les cours théoriques dispensés au premier semestre du cursus ISMIN, notamment en établissant une machine d'états plus fournie et plus concrète que celles étudiées jusqu'à présent.
Ce projet n'a pas été sans difficultés pour autant, il m'a fallu du temps pour comprendre et apprivoiser un nouvel environnement : s'adapter et maîtriser le langage Verilog, être à l'aise sur Modelsim, bien comprendre comment tous les modules interagissaient entre eux. Un deuxième écueil a été le débogage et le long temps que ce dernier a requis, je peinais à réaliser cette tâche, mais avec l'aide de monsieur Rigaud, j'ai pu bien comprendre la démarche : trouver l'erreur en analysant de manière verticale la transmission du signal du testbench et la corriger en conséquence.