MINES Saint-Étienne | Institut Mines-Télécom

Rapport Conception Système Numérique

Projet ASCON128

Yasser EL KOUHEN
Mai 2024

Table des matières

Introduction

Etant dans un monde en pleine révolution numérique, une proportion de plus en plus importante de processus dans les entreprises sont gérés par des systèmes numériques. Ce qui implique que les cyberattaques seront de plus en plus destructrices pour les entreprises à travers d'importants dommages financiers, une violation de la vie privée de clients ou même la perte de confiance des partenaires. Ceci est d'autant plus vrai pour les systèmes numériques de communication. Nous comprenons donc l'importance de la sécurité des données dans de tels systèmes numériques. Dans ce cadre, le projet consiste à concevoir, modéliser et étudier le système de chiffrement ASCON128.

L'algorithme de cryptage ASCON128 est un algorithme de cryptographie symétrique conçu pour les dispositifs légers et à faible consommation d'énergie. Il a été sélectionné comme l'un des algorithmes gagnants de la compétition NIST Lightweight Cryptography, NIST étant le National Institute of Standards and Technology du département du commerce américain. ASCON128 est un algorithme de chiffrement par bloc qui utilise une clé de 128 bits et un bloc de données de 128 bits. Il est basé sur une structure de permutation-substitution et utilise des opérations arithmétiques simples pour transformer les données en texte chiffré. L'algorithme a été conçu pour offrir une sécurité élevée et une faible consommation d'énergie, et a été évalué comme étant résistant aux attaques connues. Il est flexible et peut être utilisé dans une variété d'applications de cryptographie légère (cryptographie à destination de dispositifs disposant de contraintes de ressources limitées).

L'ambition de ce projet est de développer une compréhension approfondie du fonctionnement du système de cryptage ASCON128 à travers le langage de description matériel SystemVerilog (fichiers .sv), le logiciel de simulation Modelsim et le serveur « tallinn » fourni par l'école, Mines Saint-Etienne.

1. Présentation générale de l'algorithme de chiffrement ASCON 128

1.1 Fonctionnement de ASCON 128

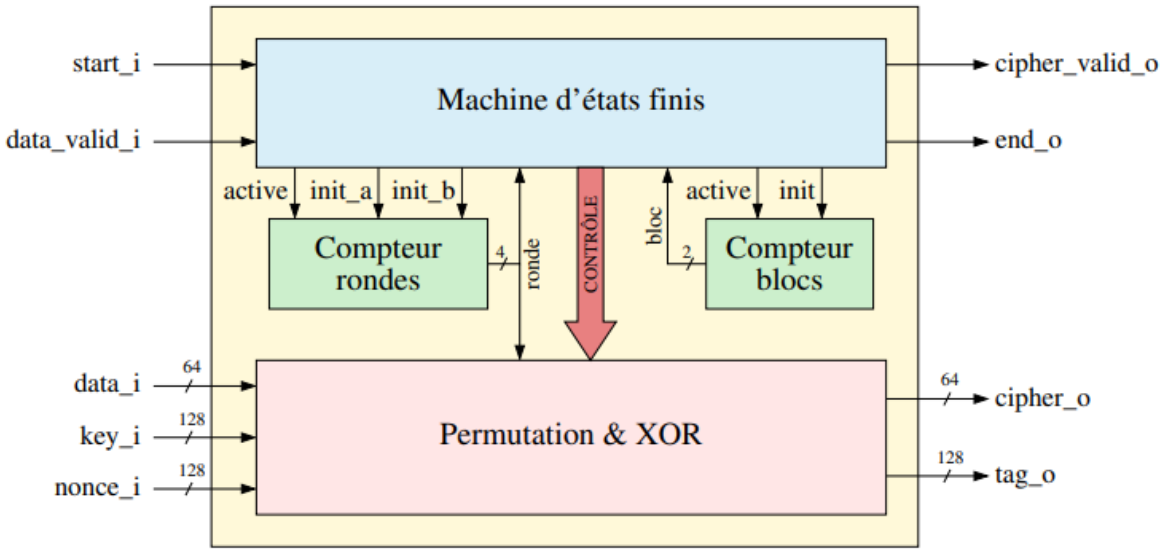


Figure 1 – Architecture globale de ASCON128

Comme on peut le voir ci-dessus, ASCON 128 est divisé en 4 blocs que sont :

- La machine d'états finis
- Le compteur de rondes
- Le compteur de blocs
- Le module de la permutation avec les XORs

Blocs qui seront expliquées dans la suite du rapport.

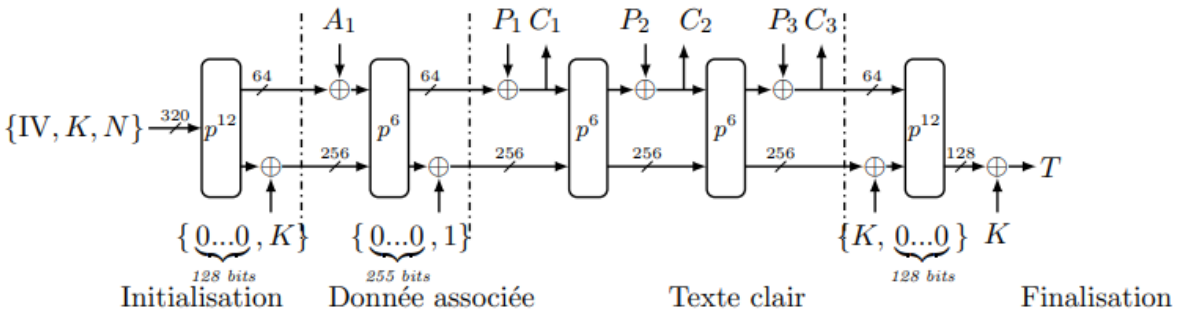


Figure 2 – Schéma du chiffrement ASCON128

C’est à travers les différentes étapes du schéma de chiffrement de l’ASCON 128 de la figure 2 qu’opère l’architecture globale de l’ASCON 128 composé de ses 4 blocs.

Il est important de noter que la figure 2 est à retenir lors de de la conception du bloc de la machine d’états finis nommé FSM (Finite State Machine), de même pour la figure 1 lors de la conception de l’architecture globale de la méthode de chiffrement ASCON 128 nommée ascon_top.

1.2. Notations utilisées

Rappelons les notations du sujet :

• L’algorithme opère sur un état courant (ou state S) de 320 bits qui de manière pratique est un type nommé **type_state** défini dans le module **ascon_pack**.

• Cet état est mis à jour avec une opération appelée permutation. La permutation comprend soit 6 itérations, soit 12 itérations, et sera notée p6 (respectivement p12). Les itérations sont aussi appelées rondes.

L’état se subdivise en deux parties :

• Une partie dite externe de r = 64 bits, notée Sr = S64

• Une partie dite interne de c = 256 bits, notée Sc = S256

1.3. Organisation du code

Dans un souci de clarté d’un code en conception d’un système numérique, les différents modules codées le sont selon une arborescence spécifique :

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

- init_modelsim.txt, le premier texte à compiler avec la commande « source fichier.txt » qui permet de configurer l’environnement Modelsim
- compil_ascon.txt que j’ai renommé compile.txt qui a pour but de compiler et lancer la modélisation de tous les modules nécessaires à l’ASCON 128 (à lancer de la même manière que le premier .txt)

- RTL, dans SRC, où se trouve tous les modules .sv qui constituent l'ASCON 128 avec les versions compilées de ces fichiers dans LIB_RTL, sous-dossier de LIB

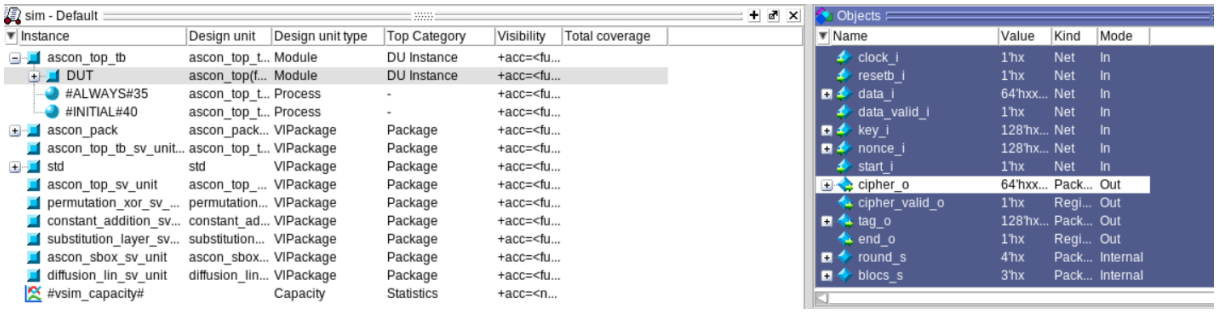
- BENCH, dans SRC, où se trouve tous les testbench des modules .sv qui constituent l'ASCON 128. Avec les versions compilées de ces fichiers qui sont dans LIB_ BENCH, sous-dossier de LIB

Il y a également des fichiers d'importance secondaire dans le dossier ASCON dont il n'est pas nécessaire de connaître l'utilité.

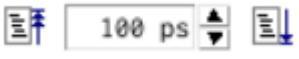
1.4. Méthode de compilation et simulation

Pour compiler, simuler et analyser un certain module en particulier, il faut effectuer les commandes suivantes dans Putty :

0) Se placer dans le dossier ascon (et non pas un de ces sous-dossiers)
1) « source init_modelsim.txt »
2) « source compile.txt » , il faut bien noter au préalable qu'il faut retirer des commentaires seulement une des commandes de simulation qui lance Modelsim. Cette commande à retirer des commentaires est celle associée au testbench du module que l'on souhaite simuler et analyser.
3) Une fois dans Modelsim, il faut sélectionner les signaux pertinents à l'analyse voulue par « ctrl + clic gauche » dans les différents modules présentés en haut à gauche. Une fois ces signaux sélectionnés, pour accéder au chronogramme associé, il faut faire « add wave » via le raccourci « ctrl + W » :



Puis sélectionner la durée à générer sur le chronogramme à chaque clic sur le logo droit :



Et enfin si on s'est trompé sur notre simulation, on peut la réinitialiser en cliquant sur le logo gauche puis en confirmant cette décision.

MINES Saint-Étienne

2. Description et test de chaque élément

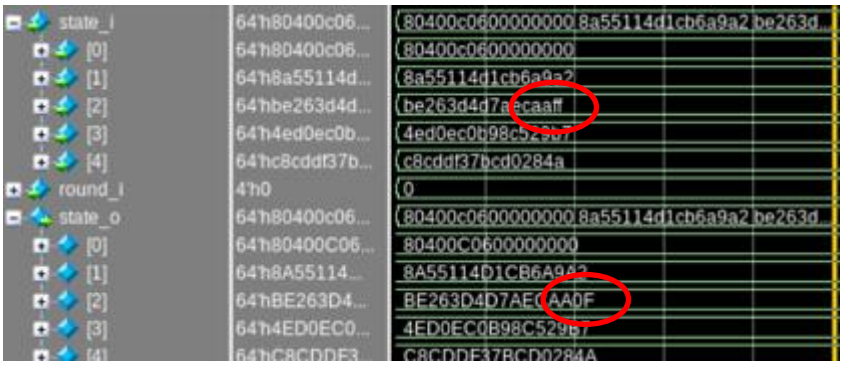
2.1. Les opérations pour la permutation

2.1.1. L'addition de constante p_c

Ce module a pour but de xorer l'entrée de 320 bits (de type type_state) S={x0,x1,x2,x3,x4} avec un constante de ronde définie dans le module ascon_pack de cette manière :

```
        // Round constant for constant addition
        localparam logic [7:0]   round_constant [0:11] = {8'hF0, 8'hE1, 8'hD2,
8'hC3, 8'hB4, 8'hA5, 8'h96, 8'h87, 8'h78, 8'h69, 8'h5A, 8'h4B};
```

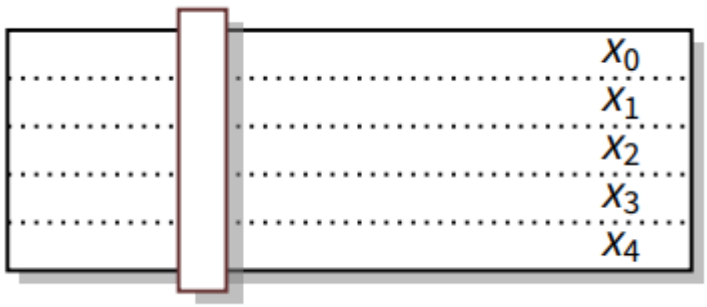
Ce round_constant est xoré avec le x2 de l'entrée de cette manière : $x_2 \leftarrow x_2 \oplus c_r$



On remarque que l'addition de constante s'opère bien sur les valeurs initiales de S={IV, K, N}, de la même manière on vérifie les rondes suivantes, ce qui conforme le bon fonctionnement de l'addition de constante du module constant_addition.

2.1.2. La couche de substitution p_s

L'objectif de ce module est d'effectuer la substitution de 5 bits en colonne de notre entrée S de type type_state de cette manière :



MINES Saint-Étienne

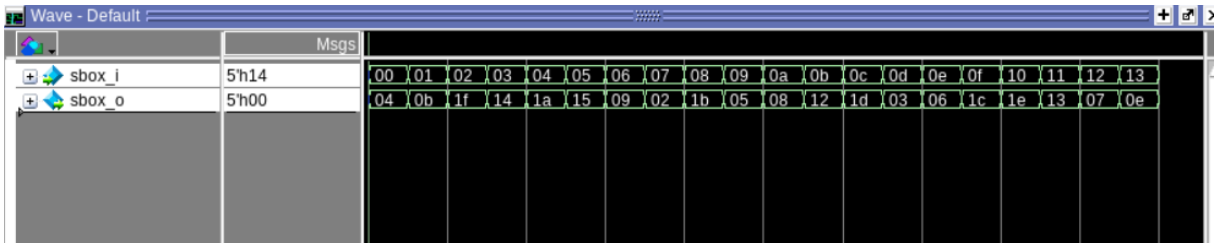
Pour ce faire, on décompose la couche de substitution en 2 modules :

- ascon_sbox qui est la fonction qui à 5 bits d'entrée vas nous associer les 5 bits de sortie voulue
- substitution_layer qui s'occupe d'appliquer ascon_sbox aux 64 colonnes de 5 bits que constituent l'entrée de 320 bits.

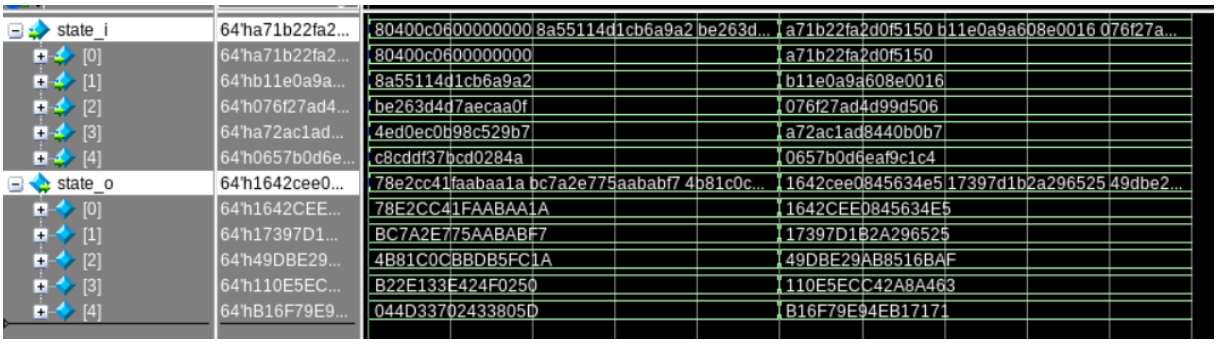
Sans oublier que ascon_sbox se repose sur la table de substitution S-box définie dans ascon_pack :

```
        localparam logic [4:0] sbox_t[0:31]={5'h04,5'h0B,5'h1F,5'h14,5'h1A,
5'h15,5'h09,5'h02,5'h1B,5'h05,5'h08,5'h12,5'h1D,5'h03,5'h06,5'h1C,5'h1E,
5'h13,5'h07,5'h0E,5'h00,5'h0D,5'h11,5'h18,5'h10,5'h0C,5'h01,5'h19,5'h16,5'h0A,
5'h0F,5'h17};
```

Après avoir testé que ascon_sbox renvoyait bien les bonnes valeurs comme on remarque ci-dessous.



On vérifie la simulation de substitution layer avec les résultats de constant_addition, ce qui nous permet d'obtenir les mêmes résultats que ceux notés dans le sujet comme on peut le voir ci-dessous.



MINES Saint-Étienne

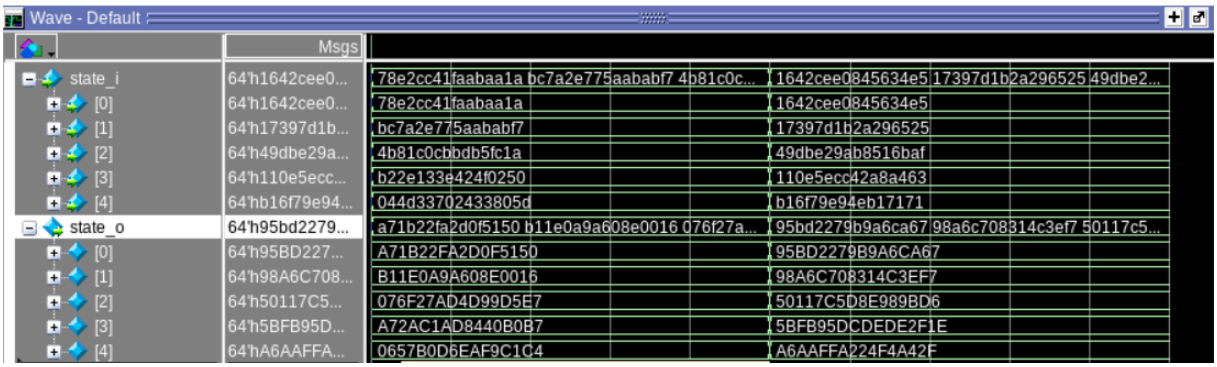
2.1.3. La couche de diffusion linéaire p_l

Le module associé à p_l diffusion_lin effectue l'opération de diffusion linéaire de cette manière :

$$x_i \leftarrow \Sigma_i(x_i)$$

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

De manière assez basique, le fichier diffusion_lin.sv n'est que la retranscription en SystemVerilog des opérations mathématiques écrites ci-dessus.



En testant le module diffusion_lin via son testbench et des résultats de la couche de substitution écrits dans le sujet. A la lumière des valeurs à obtenir pour la diffusion linéaire, on note que l'opération de diffusion linéaire fonctionne correctement.

2.2. De la permutation élémentaire à la permutation avec XOR

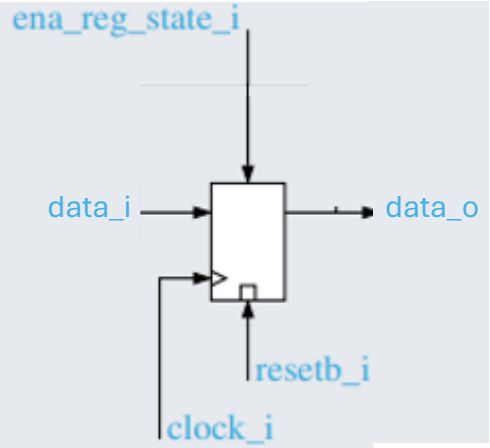
2.2.1. Le Multiplexeur

Le module associé au multiplexeur du nom de mux_state est un simple module fonction de l'entrée logique sel_i qui renvoie :

- L'entrée, de type type_state, data1_i lorsque sel_i=0
- Et l'entrée, de type type_state, data2_i lorsque sel_i=1

2.2.2. Fonctionnement des registres

Le registre d'états représenté par le module state_register_w_en permet d'enregistrer l'entrée data_i lorsque en_reg_state_i = 1, data_i et data_o étant de type type-state. La nuance à retenir est que ce registre enregistre la valeur qui lui arrive au coup de clock suivant. C'est-à-dire que si on est à la permutation numéro r au coup de clock n, au coup de clock n+1, la valeur en sortie du reg_state est celle associé à la permutation numéro r et non pas r+1.



De la même manière qu'opère le registre d'états via le module state_register_w_en, un autre type de registre associé au module du nom de register_w_en est utilisé. La seule différence entre ces 2 types de registres est les types donnés en sortie et en entrée. Le premier, state_register_w_en, a data_i et data_o de type type-state. Tandis que le 2ème type de registre présenté, register_w_en , a data_i et data_o de type binaire avec leur taille modifiable comme ci-dessous :

```
logic [nb_bits_g-1 : 0]
```

Ce deuxième type de registre « variable » permet de pouvoir enregistrer les ciphers et le tag de par l'implémentation de 2 registres de ce type dans le diagramme de la permutation avec XOR.

2.2.3. Fonctionnement des XORs

Le bloc XOR begin est un module utilisé dans les phases de Données Associées, de Texte Clair et de Finalisation. Son rôle consiste à effectuer une opération XOR entre le signal d'entrée en sortie du multiplexeur et la clé K ou la donnée associée (A, P1, P2 ou P3 selon l'étape de l'algorithme). Si le signal en_xor_data_b_o est activé, l'opération est effectuée avec la donnée, tandis que si en_xor_key_b_o est activé, l'opération est effectuée avec la clé K.

De plus, les phases d'Initialisation de Données Associées et de Finalisation utilisent également le bloc XOR end. Ce module fonctionne de manière similaire au XOR Begin, mais effectue l'opération entre le signal en sortie du multiplexeur et la clé ou le dernier bit de x4 avec un 1. Les signaux d'activation correspondants sont en_xor_key_e_o et en_xor_lsb_e_o.

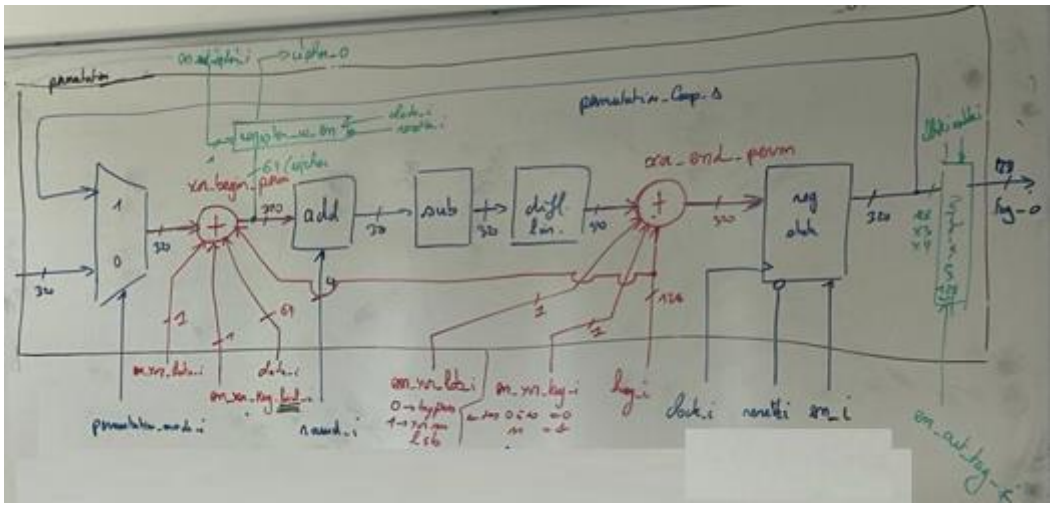
2.2.4. La permutation

Dans un premier temps pour modéliser la permutation p qui est telle que : $p = p_L \circ p_S \circ p_C$

Il suffit de simplement connecter les différents modules présentés au 3.1 dans le bon ordre comme cela a été fait avec le module permutation_elementaire.

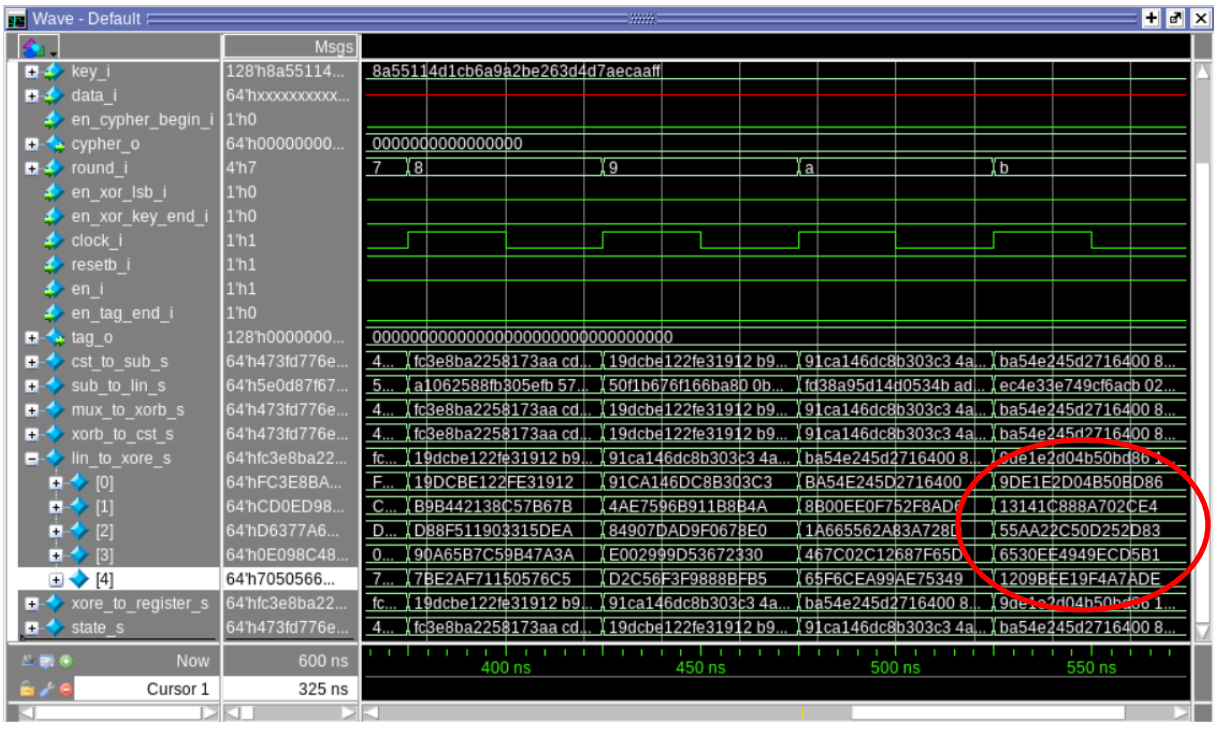
Etant donné que l'on effectue p tel que p^{12} et p^6 (respectivement 12 et 6 fois successivement). L'étape suivante est de créer un unique module qui permet d'opérer p le nombre de fois que l'on veut. Afin d'arriver à ce résultat, il est nécessaire d'ajouter à permutation_elementaire, le multiplexeur (mux_state.sv) pour différencier la première ronde (provenant d'une entrée) et les suivantes (crées et enregistrées au fur et à mesure), ainsi que le registre d'états (state_register.sv) pour enregistrer les valeurs intermédiaires obtenues à chaque ronde et ainsi pouvoir les utiliser pour la ronde suivante. Le module obtenu porte ainsi le nom permutation.sv.

Comme on peut le remarquer à travers la Figure 2, le schéma du chiffrement ASCON 128 consiste également en l'utilisation de XORs (XOR begin et XOR end). Ainsi il est nécessaire d'ajouter ces modules XOR à notre module permutation.sv pour avoir un diagramme de la permutation avec XOR qui sera la base pour le schéma de chiffrement de ASCON 128. Il faut également pouvoir enregistrer et exploiter les valeurs xorés, ce qui se fait avec l'utilisation de 2 registres « variable ». Un de taille de données 64 bits pour les ciphers (car les ciphers sont chacun de taille 64 bits) et un autre de taille de données 128 bits (car le tag est de taille 128 bits). Ainsi, nous avons un fichier final pour la permutation, prenant en compte les XOR et les registres précédemment décrits, du nom de permutation_xor.sv associé au diagramme ci-dessous :



Ayant construit notre permutation_xor.sv par « constructions additives », en effectuant le test_bench de la permutation_xor, on vérifie également nos 2 modules intermédiaires de permutations. Après avoir effectué un testbench de permutation_xor pour vérifier que les 12 rondes opéraient bien sans pour autant vérifier les XOR, on observe qu'on obtient bien la valeur attendue dans le sujet :

```
Diffusion linéaire : 9de1e2d04b50bd86 13141c888a702ce4 55aa22c50d252d83 6530ee4949ecd5b1 1209bee19f4a7ade
```



MINES Saint-Étienne

2.3 Les différents compteurs

Le compteur de blocs, compteur qui permet de diminuer le nombre d'états de notre machine d'états est un compteur simple de module compteur_simple_init qui se réinitialise à 0 lorsque init_a_i =1 et en_i=1 et s'incrémente lorsque seulement en_i=1.

Le compteur de rondes quant à lui, qui permet de déterminer le nombre de rondes à effectuer pour une étape de permutation est un compteur double de module compteur_double_init qui se réinitialise à 0 lorsque init_a_i =1 et en_i=1, se réinitialise à 6 lorsque init_b_i =1 et en_i=1 et s'incrémente lorsque seulement en_i=1.

2.4 Machine d'état

N'ayant malheureusement pas le temps de faire un diagramme d'états numérique au propre, il est quand même avantageux de partager les états de la FSM dans le bon ordre :

```
typedef enum {attente, conf_init, init_rd0, init_rd0_to_rd10, init_rd11,
conf_da, da_rd0, da_rd0_to_rd5, da_rd6, conf_tc, tc_rd0, tc_rd0_to_rd6,
conf_finalisation, finalisation_rd0, finalisation_rd0_to_rd10,
finalisation_rd11, fin} state_t;
```

2.5. Architecture globale de l'ASCON 128 :

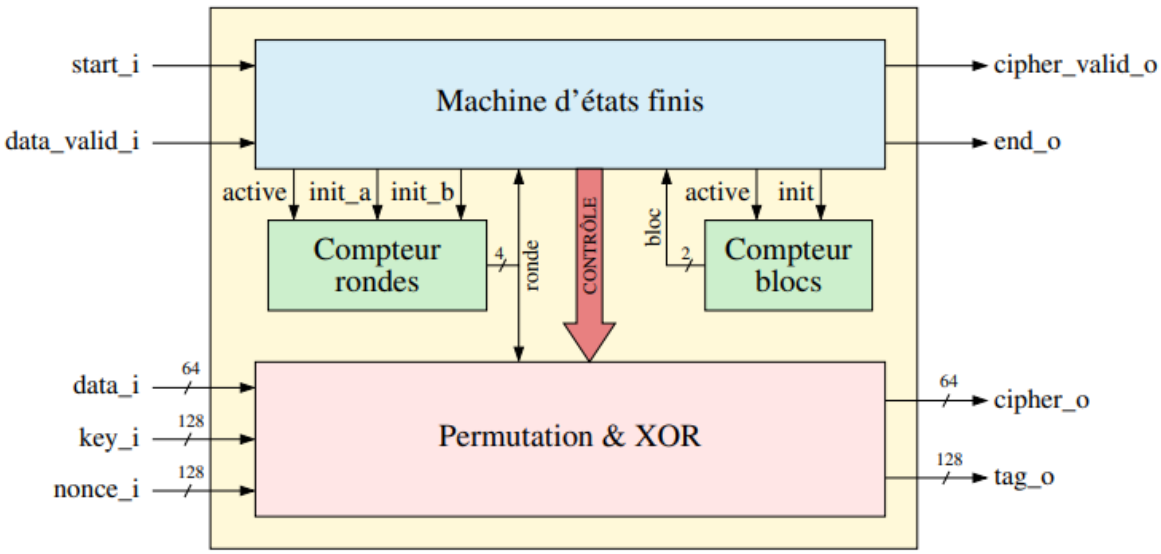


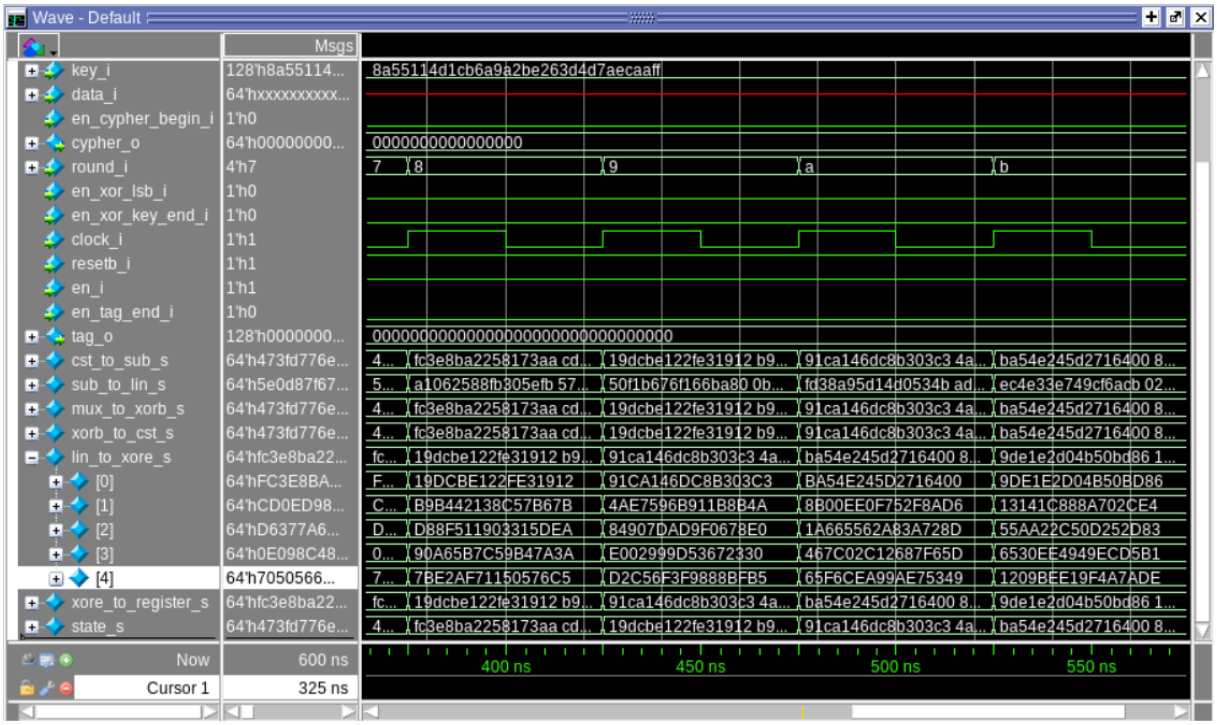
Figure 1 – Architecture globale de ASCON128

MINES Saint-Étienne

Comme on peut le voir dans la figure 1, l'architecture globale de l'ASCON 128 consiste en la connexion de 4 blocs que sont la machine d'états, le bloc permutation & XOR, le compteur de blocs et le compteur de rondes. Ainsi cette architecture globale est représenté par un module ascon_top qui consiste en la connexion de ces 4 blocs via les modules FSM, pemutation_xor, compteur_rondes et compteurs_blocs.

C'est d'ailleurs le testbench de l'ascon_top qui permet de vérifier si la FSM est correcte, comme on peut le simuler, l'aspect « combinational process for transitions » est correct au détail près qu'il faudrait rajouter un état « data_valid_x » après chaque état où l'on extrait une valeur (et également l'insertion de {IV,K,N}).

3. Difficultés rencontrées



Pour des raisons inconnues, bien que le premier p12 est effectué ainsi que le 1er XOR, il m'est impossible de comprendre pourquoi cette valeur n'est pas retenue à l'état suivant l'opération du XOR

4. Conclusion

Pour synthétiser, le projet ASCON128 a été une expérience bénéfique pour moi. J'ai réussi à implémenter la plupart des composants nécessaires, tels que la machine d'états et les permutations, et les résultats de la simulation ont montré un fonctionnement correct du chiffrement.

Toutefois, il reste des améliorations à apporter. Par exemple, l'utilisation d'une machine d'état de Mealy pourrait améliorer les performances du système électronique en temps réel, mais cela pourrait également complexifier la gestion des transitions d'état et des sorties. Car nous avons utilisé une machine de Moore, en effet, les états futurs dépendent uniquement des états précédents

En somme, ce projet m'a permis de renforcer mes compétences en conception de systèmes numériques et en électronique numérique, et m'a préparé pour notre cours de cryptographie à venir. Des optimisations de performances, des explorations de variantes de sécurité renforcée et des adaptations pour d'autres environnements spécifiques sont autant de pistes intéressantes pour des développements futurs.