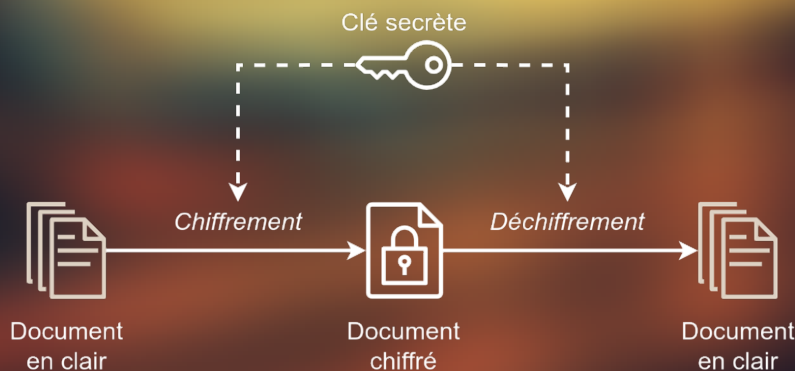


Compte rendu de projet

MODELISATION SYSTEMVERILOG DE L'ALGORITHME DE CHIFFREMENT ASCON



Chloé Larroze

2024-05-02

Table des matières

Introduction	3
Description du projet	4
Fonctionnement global de ASCON128	4
Organisation.....	5
Représentation	5
Permutation	5
Conception et test des transformations pC et pS	5
Addition de constante pC	6
La couche de substitution ps	7
Élaboration de la Sbox	8
Substitution Ps	8
La couche de diffusion linéaire pl	8
Rotation.....	9
Les permutations $p6$ et $p12$.....	10
Permutation 'classique'	11
Permutation XOR.....	11
Machine d'état	14
Test de la FSM	18
Conclusion.....	18
Problèmes ou difficultés rencontrés.....	18

Introduction

Dans un monde où la confidentialité des communications en ligne est essentielle, Bob et Alice cherchent à sécuriser leurs échanges contre les interceptions malveillantes. Pour cela, ils ont choisi l'algorithme ASCON128, une fonction de hachage reconnue pour sa robustesse et son efficacité. Comme l'objectif principal de la cryptographie légère concerne les dispositifs embarqués à ressources limitées, l'une des principales préoccupations concerne les implémentations efficaces et la protection contre les attaques physiques.

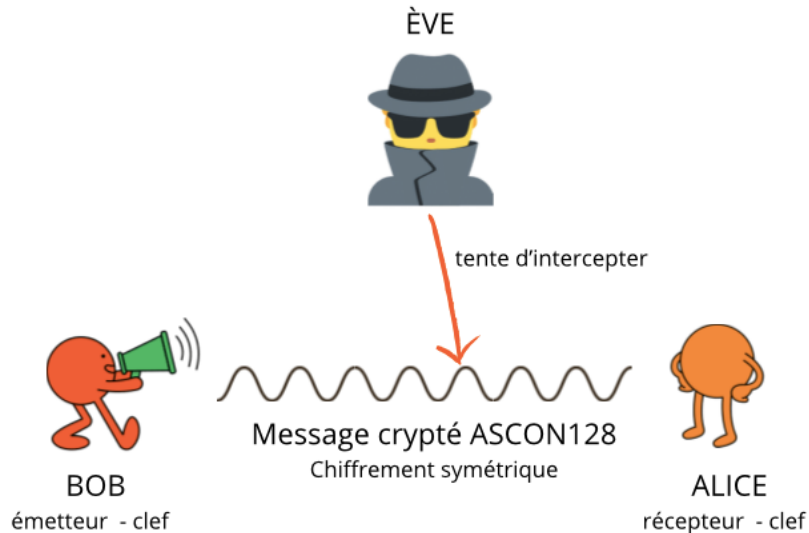


Figure 1 : Communication chiffrée

Dans ce contexte, notre projet consiste à développer une implémentation complète de l'algorithme ASCON128 en utilisant le langage de description matériel VHDL. Notre objectif est de fournir une solution fiable et sécurisée pour garantir la confidentialité et l'authenticité des échanges entre Bob et Alice.

Ce rapport présentera de manière détaillée les étapes de développement de notre projet, en commençant par la conception des transformations fondamentales telles que pC , pS et Σi , suivie du développement des permutations $p6$ et $p12$. Nous explorerons également l'ajout d'opérateurs XOR dans les permutations ainsi que la conception d'une machine à états finis pour l'intégration des compteurs. Enfin, nous aborderons les aspects de test, de finalisation et de documentation du projet.

Description du projet

Fonctionnement global de ASCON128

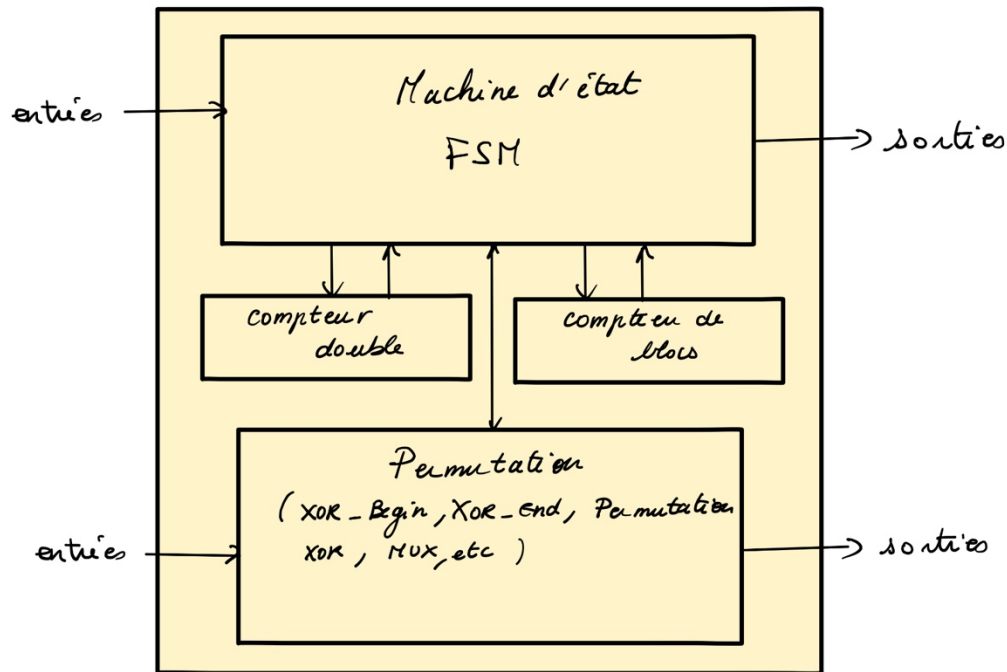


Figure 2 : Fonctionnement complet ASCON

Cet algorithme chiffre et authentifie un message en utilisant l'algorithme ASCON128. Il se compose de quatre blocs principaux :

1. **Machine d'état fini (FSM)** : Gère le flux d'opérations, contrôle les différents états du processus de chiffrement et d'authentification.
2. **Bloc de permutation** : Effectue des permutations sur l'état interne à chaque étape du processus, assurant un mélange sécurisé des données.
3. **Blocs XOR** : Appliquent des opérations XOR pour combiner les données avec l'état interne, contribuant ainsi à l'obscurcissement des données.
4. **Compteurs de permutation** : Maintiennent une trace du nombre de permutations effectuées, régulant ainsi le déroulement du processus de chiffrement et d'authentification.

Ensemble, ces blocs travaillent de manière coordonnée pour sécuriser le message et garantir son intégrité lors de la transmission.

Organisation

Durant tout le projet, nous opterons pour une organisation des fichiers dont l'arborescence est la suivante :

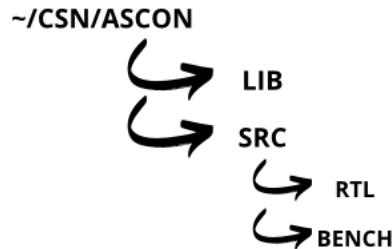


Figure 3 : Arborescence

Les instances seront placées dans /SRC/RTL tandis que les test benches seront positionnés dans /SRC/BENCH. Grâce aux commandes vlog et vsim, les simulations et simulations des test benches seront respectivement placées dans /LIB/LIB_RTL et /LIB/LIB_BENCH.

L'ensemble des fichiers réalisés pendant ce projet seront attachés à ce document.

Par ailleurs, la simulation s'effectuera sur Modelsim installé sur le serveur Tallinn de l'école.

Représentation

L'algorithme agit sur un état actuel de 320 bits, divisé en deux parties : une externe de 64 bits et une interne de 256 bits. Ces parties sont mises à jour par une opération de permutation, soit en 6 itérations (appelées p^6) soit en 12 itérations (appelées p^{12}). L'état S est représenté par cinq registres x_i de 64 bits chacun, à la manière d'une matrice.

Permutation

Conception et test des transformations p_C et p_S

Afin de mieux comprendre l'architecture de cette première partie du système et de visualiser les inputs et outputs, on peut la représenter avec des blocs de la manière suivante, en la décomposant à nouveau :

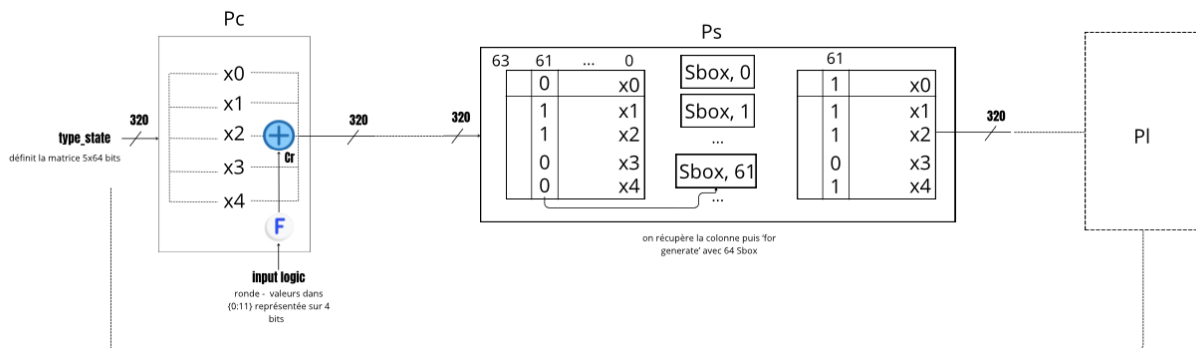


Figure 4 : Architecture des Pc et Ps

Addition de constante p_c

Nous allons tout d'abord créer une instance Pc.sv ainsi que son test bench Pc_tb.sv. Dans ce code, on déclare le module P_c . Il comprend une sortie et deux entrées:

- Pc_in_i, de type type_state qui définit la matrice 5x64 bits.
- round_i, représentée sur 4 bits. Elle permet la sélection de constante Cr_s qui sera envoyée sur le xor.

On crée en effet une variable intermédiaire C_r qui comprends 8 bits mais sera manipulés sur 64 pour respecter la taille des registres (5x64). Elle contient des 0 entre ses bits 63 et 8 puis une valeur particulière sur ses 8 bits de poids faible.

Remarque : On aurait également pu concaténer directement sans passer par une variable *logic* C_r . Ainsi, seuls les bits de poids faible de round_constant dans ascon_pack sont changés. On aurait donc eu:

```
// assign Pc_out_[2][63:8] = Pc_in_i[2][63:8]; // Bits
                              inchangés
// assign Pc_out_[2][7:0] = Pc_in_i[2][7:0] xor
                              round_constant[round_i];
```

Les tests du test benches sont bien concluants, on retrouve les valeurs indiquées dans la documentation.

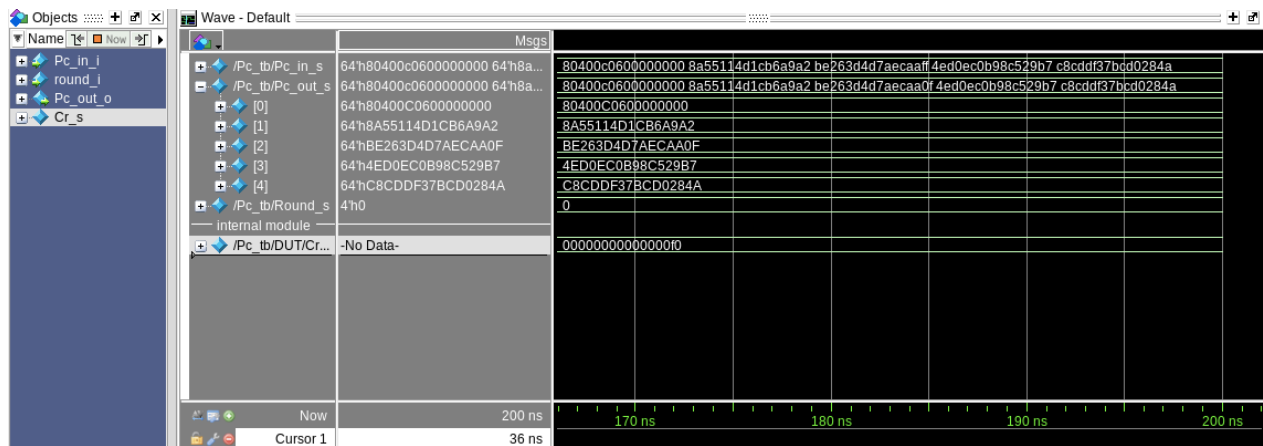


Figure 5 : Résultat du testbench $P_c_tb.sv$

L'addition a bien été effectuée, cette première couche fonctionne donc correctement.

La couche de substitution p_s

Nous diviserons la substitution p_s en deux étapes afin de simplifier son élaboration:

- la sbox
- la couche p_s

Nous nous baserons sur la table de substitution suivante pour l'implémentation :

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
$S(x)$	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 6 : Tableau de valeurs $S(x)$ pour différentes valeurs de x

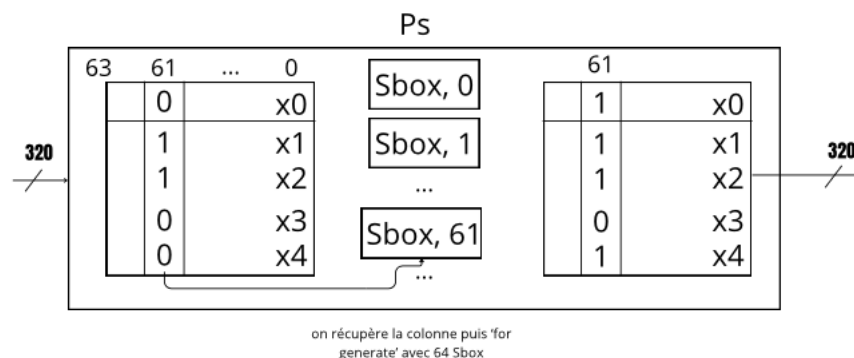


Figure 7: Détail du P_s

Codons premièrement la fonction élémentaire S_{box} . Par la suite, on l'appliquera en parallèle 64 fois de manière bit-sliced (verticalement, à travers les mots, de manière à regarder chaque colonne comme présenté dans la figure ci-dessus).

Élaboration de la Sbox

La Sbox comprend 5 bits en entrée et 5 bits en sortie, correspondant aux valeurs du tableau. Une première option pour la réalisation de la Sbox est l'emploi du « case » listant l'ensemble des possibilités, c'est cette option que nous choisirons. Le « case » disposera ainsi de 2^5 soit 32 valeurs. Il aurait également été possible de l'implémenter sous la forme d'un tableau comme dans la transformation précédente en indexant avec les numéros de round.

En compilant le programme dans un premier temps, une erreur "Illegal reference to net 'Sx'" apparait. Pour résoudre ce problème, on ajoute reg devant output dans la déclaration de Sx, ce qui permet son assignation à l'intérieur d'un bloc « always ».

Testons désormais l'ensemble des cas de notre Sbox à l'aide du fichier de test bench.

	Msgs			
+ /Sbox_tb/x	5'h1f	1f		
+ /Sbox_tb/Sx	5'h1e	1e		

Figure 8 : Résultat du test bench de la Sbox

Substitution Ps

La substitution p_s a désormais pour but de concaténer les valeurs retournées par la Sbox pour l'ensemble des valeurs.

On souhaite obtenir les valeurs en surbrillance suivantes :

```
-- Permutation (r=00)
Addition constante : 80400c0600000000 8a55114d1cb6a9a2 be263d4d7aecaa0f 4ed0ec0b98c529b7 c8cddf37bcd0284a
Substitution S-box : 78e2cc43faabaa4a bc3a2e755aabab17 4bc1c0c9bd...
Diffusion linéaire : a71b22fa2d0f5150 b11e0a9a608e0016 076f27ad4d99d5e7 a72ac1ad8440b0b7 0657b0d6eaf9c1c4
```

Figure 9 : Résultats attendus du testbench P_s_tb.sv

/Ps_tb/Ps_i_s	64'h80400c06...	80400c0600000000 8a55114d1cb6a9a2 be263d4d7...
[0]	64'h80400C06...	80400C0600000000
[1]	64'h8A55114...	8A55114D1CB6A9A2
[2]	64'hBE263D4...	BE263D4D7AECAAFF
[3]	64'h4ED0EC0...	4ED0EC0B98C529B7
[4]	64'hC8CDDF3...	C8CDDF37BCD0284A
/Ps_tb/Ps_o_s	64'h78e2cc43f...	78e2cc43faabaa4a bc3a2e755aabab17 4bc1c0c9bd...
[0]	64'h78E2CC4...	78E2CC43FAABAA4A
[1]	64'hBC3A2E7...	BC3A2E755AABAB17
[2]	64'h4BC1C0C...	4BC1C0C9BDB5FCEA
[3]	64'hB26E133...	B26E133C424F02A0
[4]	64'h840D3376...	840D33762433805D

Figure 10 : Résultat de la couche de substitution

On retrouve bien le résultat attendu en sortie de la seconde couche, les données de sortie correspondent bien à la concaténation de l'ensemble des vecteurs de bits.

La couche de diffusion linéaire p_l

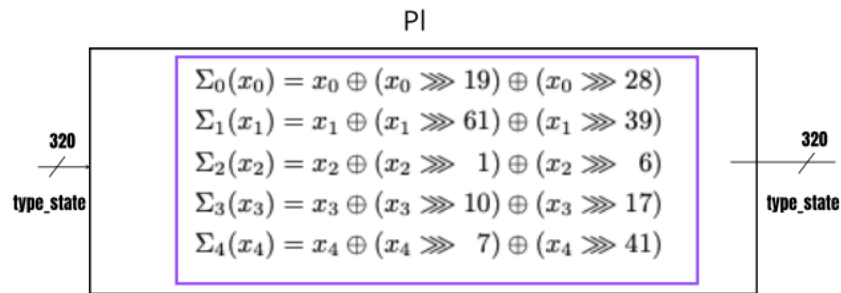


Figure 11 : Couche de diffusion linéaire P_l

Cette instance a en entrée et en sortie un `type_state` de 320 bits nommés respectivement `Pl_in_i` et `Pl_out_o`. La variable x_i de sortie sera une somme de 3 vecteurs. Pour le vecteur x_0 par exemple, on aura

$$x_0 = x_0 \oplus (x_0 \ll 19) \oplus (x_0 \ll 29)$$

Rotation

Expliquons tout d'abord les rotations, exprimées par le symbole \gg dans l'ensemble des schémas. Prenons pour exemple le vecteur suivant : `1001.0000` et appliquons-lui une rotation de 2.

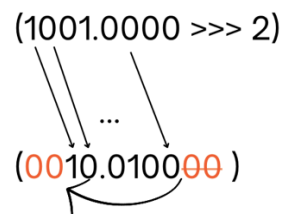


Figure 12 : Rotation

On obtiendra donc en Verilog l'expression d'une rotation de i ($x \gg i$) que l'on pourrait d'ailleurs définir dans un module extérieur afin de l'appeler. On préférera cependant écrire les 5 lignes à la suite pour des soucis de lisibilité :

```
assign x_i = {x[i-1:0], x [63: i]}
```

Le module p_l sera en charge d'effectuer toutes les opérations sur l'ensemble des états correspondant aux « vecteurs » x_i .

On obtient le résultat suivant pour cette dernière couche :













 /Pl_tb/Pl_in_i	64'h80400c06...	80400c0600000000 8a55114d1cb6a9a2 be263d4d7aecaaff 4ed0ec0b98c529b7 c8cddf37bcd0...
 [0]	64'h80400C06...	80400C0600000000
 [1]	64'h8A55114...	8A55114D1CB6A9A2
 [2]	64'hBE263D4...	BE263D4D7AEC A AFF
 [3]	64'h4ED0EC0...	4ED0EC0B98C529B7
 [4]	64'hC8CDDF3...	C8CDDF37BCD0284A
 /Pl_tb/Pl_out_o	64'h80401c06...	80401c0605800060 8455717d1db629be fc073b7e62ecaaff 4ef0bc289f4629b7 cacd5f2fb09008...
 [0]	64'h80401C06...	80401C0605800060
 [1]	64'h8455717...	8455717D1DB629BE
 [2]	64'hFC073B7...	FC073B7E62E C A AFF
 [3]	64'h4EF0BC2...	4EF0BC289F4629B7
 [4]	64'hCACD5F2...	CACD5F2FB090084C

Figure 13 : Résultats de la couche de diffusion linéaire

Les permutations p^6 et p^{12}

Grâce aux données de l'énoncé, nous savons que la permutation globale p ne résulte que de la composition des trois couches : $p = p_L \circ p_S \circ p_C$.

Effectuons un premier test pour lequel le type_state de sortie résulte de la concaténation des trois modules.













 /p_tb/pcin_s	64'h598da474...	598da474303d9164 7559456e06c73ad3 94beaba9335e44cd 8866d2abc492c960 c11bf1...
 [0]	64'h598DA474...	598DA474303D9164
 [1]	64'h7559456E...	7559456E06C73AD3
 [2]	64'h94BEABA...	94BEABA9335E44CD
 [3]	64'h8866D2A...	8866D2ABC492C960
 [4]	64'hC11BF1D...	C11BF1D12E77B520
 /p_tb/pcout_s	64'h46c3cb56...	46c3cb562e460b55 d4cd4b6f37c1eeb0 4196cf4d9197a1b5 d4bf9e19035c34a0 fa10855f...
 [0]	64'h46C3CB5...	46C3CB562E460B55
 [1]	64'hD4CD4B6...	D4CD4B6F37C1EEB0
 [2]	64'h4196CF4...	4196CF4D9197A1B5
 [3]	64'hD4BF9E1...	D4BF9E19035C34A0
 [4]	64'hFA10855F...	FA10855FA6744234

Figure 14 : Résultat de la permutation simple $p = p_L \circ p_S \circ p_C$

Pour la suite, nous aurons besoin de réaliser la permutation plusieurs fois. Par exemple, la sortie reboucle vers l'entrée afin de faire p^2 , on réitérera l'opération autant de fois que l'on souhaite (6 ou 12 fois selon la transformation choisie). Nous y ajouterons donc un compteur : le compteur de round.

D'autre part, il risque d'y avoir un conflit lorsque l'on "boucle" le module. Il est donc nécessaire d'y ajouter un multiplexeur afin de décider s'il faut renvoyer la sortie de la permutation ou bien retourner à l'entrée. On retrouve le schéma du montage dans la figure suivante.

Permutation 'classique' avec multiplexeur et registre

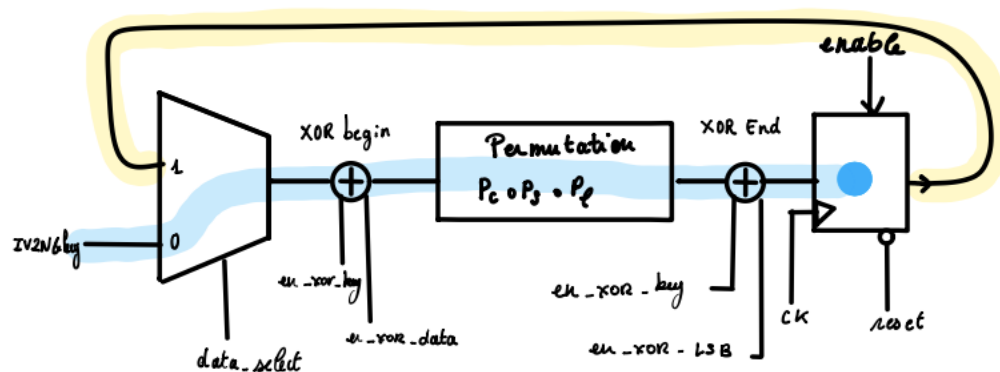


Figure 15 : Permutation p

Le chemin bleu correspond donc à la première itération des permutations. En effet, on rentre une donnée de 320 bits, la concaténation des données à chiffrer puis, en fonction des valeurs, on décide de sauvegarder dans le registre ou de sélectionner la donnée en sortie afin de reboucler, correspondant au chemin jaune. Cette première permutation sera complétée par l'ajout de XOR.

◆ /Permutation_tb/C...	32'h0000000a	0000000a
◆ /Permutation_tb/cl...	1'h0	
[-] ◆ /Permutation_tb/d...	64'h80400c06...	80400c0600000000 8a55114d1cb6a9a2 be263d4d7aecaff 4ed0ec0b98c529b7 c8cddf37bcd...
+ ◆ [0]	64'h80400C06...	80400C0600000000
+ ◆ [1]	64'h8A55114...	8A55114D1CB6A9A2
+ ◆ [2]	64'hBE263D4...	BE263D4D7AEC A A F F
+ ◆ [3]	64'h4ED0EC0...	4ED0EC0B98C529B7
+ ◆ [4]	64'hC8CDDF3...	C8CDDF37BCD0284A
[-] ◆ /Permutation_tb/d...	64'h78e2cc43f...	78e2cc43faabaa1a bc3a2e755aababf7 4bc1c0c9bdb5fc1a b26e133c424f0250 840d33762433...
+ ◆ [0]	64'h78E2CC4...	78E2CC43FAABAA1A
+ ◆ [1]	64'hBC3A2E7...	BC3A2E755AABABF7
+ ◆ [2]	64'h4BC1C0C...	4BC1C0C9BDB5FC1A
+ ◆ [3]	64'hB26E133...	B26E133C424F0250
+ ◆ [4]	64'h840D3376...	840D33762433805D
◆ /Permutation_tb/e...	1'h1	
◆ /Permutation_tb/r...	1'h1	
[+] ◆ /Permutation_tb/r...	4'h0	0
◆ /Permutation_tb/s...	1'h0	

Figure 16 : Résultat de la permutation

Permutation XOR

Cependant, on a trois types de permutations distinctes à réaliser dans le processus. Créons donc une permutation 'générale' permettant d'exécuter l'ensemble des permutations suivantes à l'aide de XOR.

- Permutation classique : On ne prend en compte aucun XOR, seulement les données sur le chemin 'principal' seront prises en compte.
- Permutation XOR Begin : Si **en_xor_key** vaut 1, on prend **data_i** et on applique une transformation XOR sur x_0 , visible dans le module **en_xor_begin.sv**. Ils seront donc placés devant les permutations seulement lors de leur première itération.
- Permutation XOR End : Si un des deux signaux **en_xor_key** ou **en_xor_LSB** vaut 1, on effectue des opérations supplémentaires sur x_3 et x_4 . On saura par ailleurs que les calculs sont terminés grâce à la FSM. Ils seront donc quant à eux logiquement placés à la fin des permutations lors de leur dernière itération.

En outre, on doit pouvoir commencer ou arrêter la mémorisation quand on le décide. Il faut donc pouvoir temporiser les opérations en sortie du module Pl. Comme dans une architecture pipeline, on intercale un registre entre la sortie du module Pl et du re-bouclage vers le multiplexeur de l'entrée du module permutation. Cette permutation complexe est donc sauvegardée puis synchronisée avec la clock. Ces modifications sont visibles sur la figure suivante:

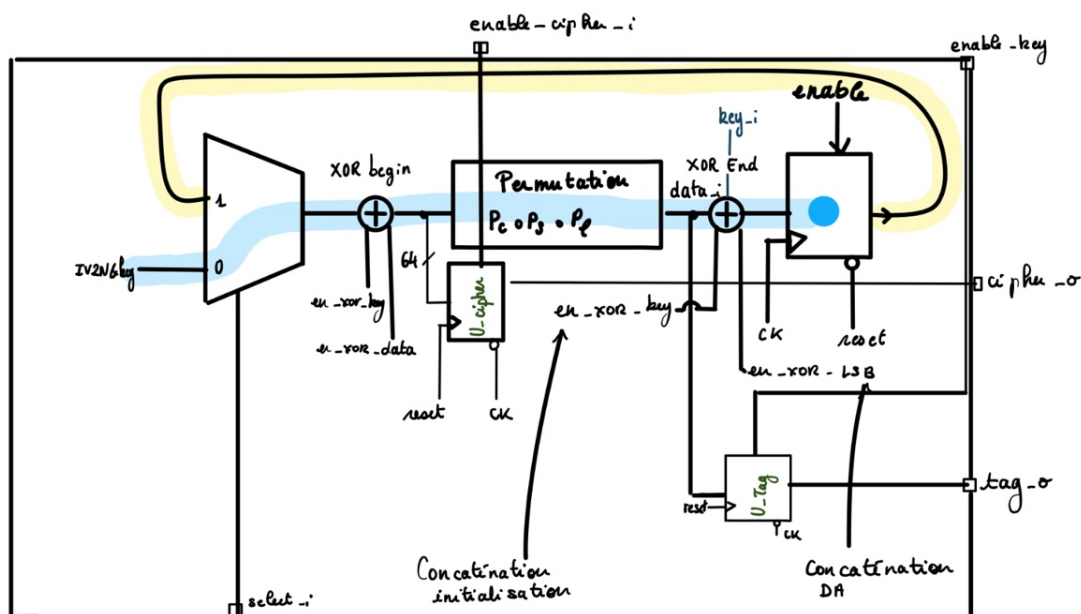


Figure 17 : Permutation XOR

Nous aurons donc quatre types de XOR selon l'algorithme décrit dans le sujet :

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

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

Ci-dessous, une première version de la permutation XOR sans la gestion des cipher (PermutationXOR_1 .sv et son test bench associé) correspondants ou encore le tag, qui seront ajouté dans la version suivante.

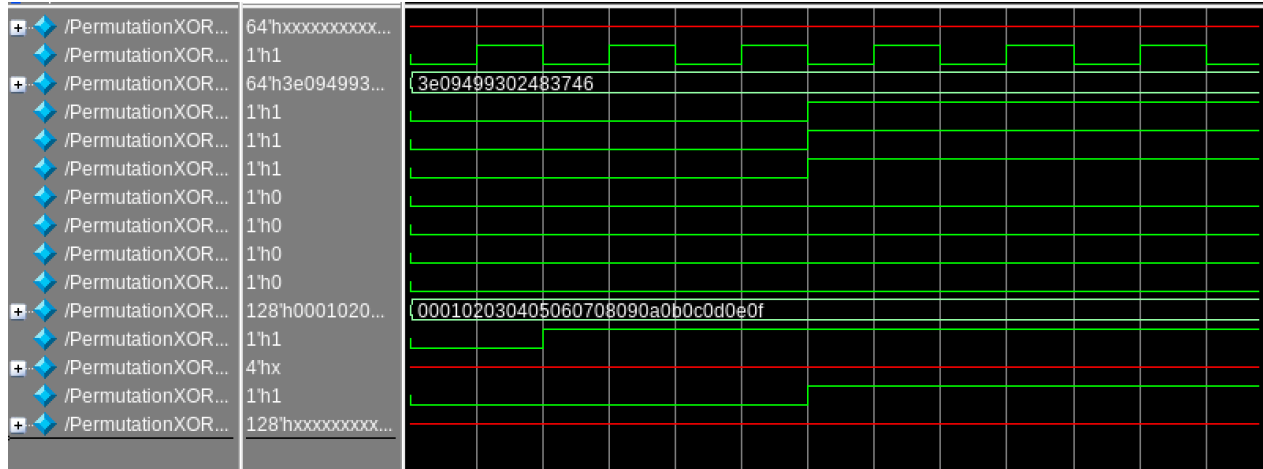


Figure 18 : Résultat intermédiaire permutation XOR

Enfin, on réalise la permutation finale par l'ajout de la gestion des ciphers.

Machine d'état

Implémentons à présent notre machine d'état : la FSM. Elle sera le cerveau de notre système, pilotant l'ensemble des signaux. On peut premièrement se la représenter de manière assez grossière sous forme de bloc puis de la détailler dans un second temps afin de n'omettre aucun signal d'entrée ou de sortie.

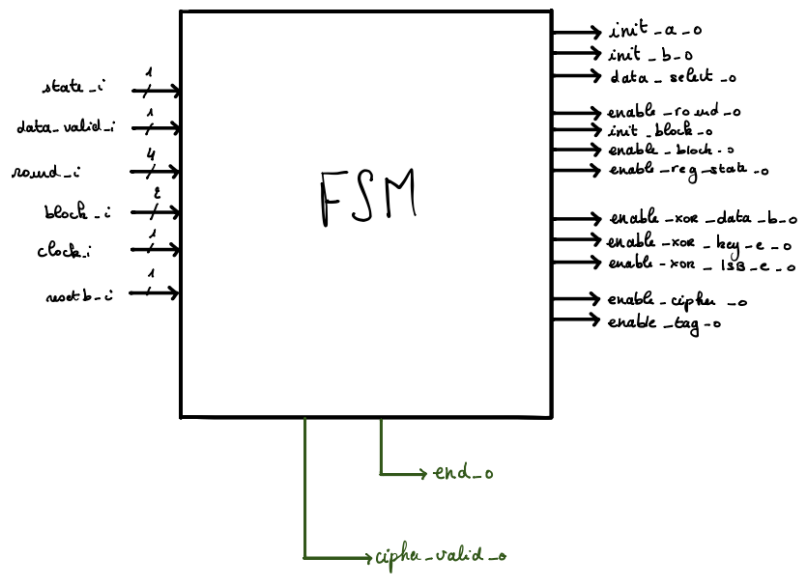


Figure 19 : Machine d'état, inputs et outputs

Plus précisément, cette dernière se décompose en plusieurs états, visibles dans la figure ci-dessous:

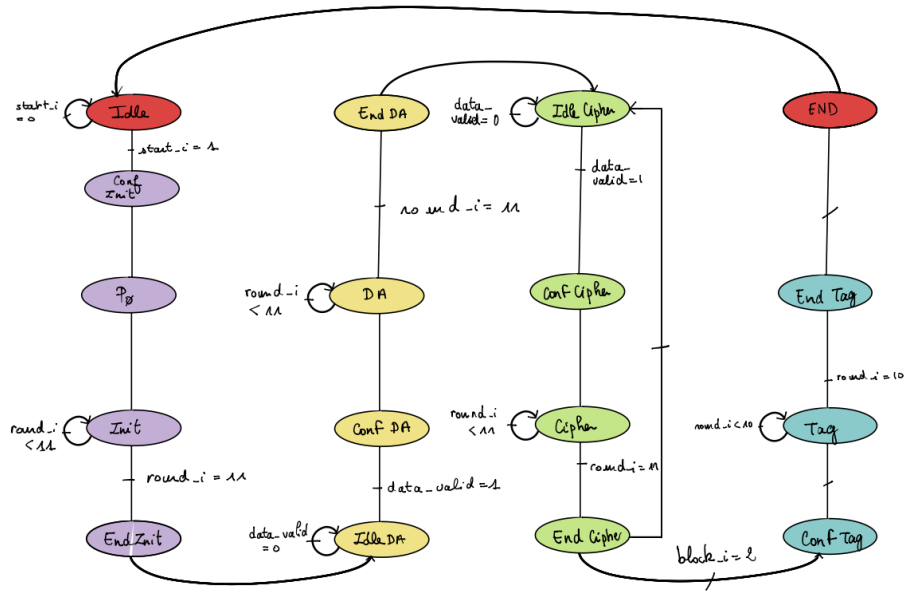


Figure 20 : Diagramme d'état de la FSM

Pour plus de simplicité dans le développement et dans les explications, détaillons un peu plus ce diagramme. Dans un premier temps, on traitera la partie allant de l'Idle au DA.

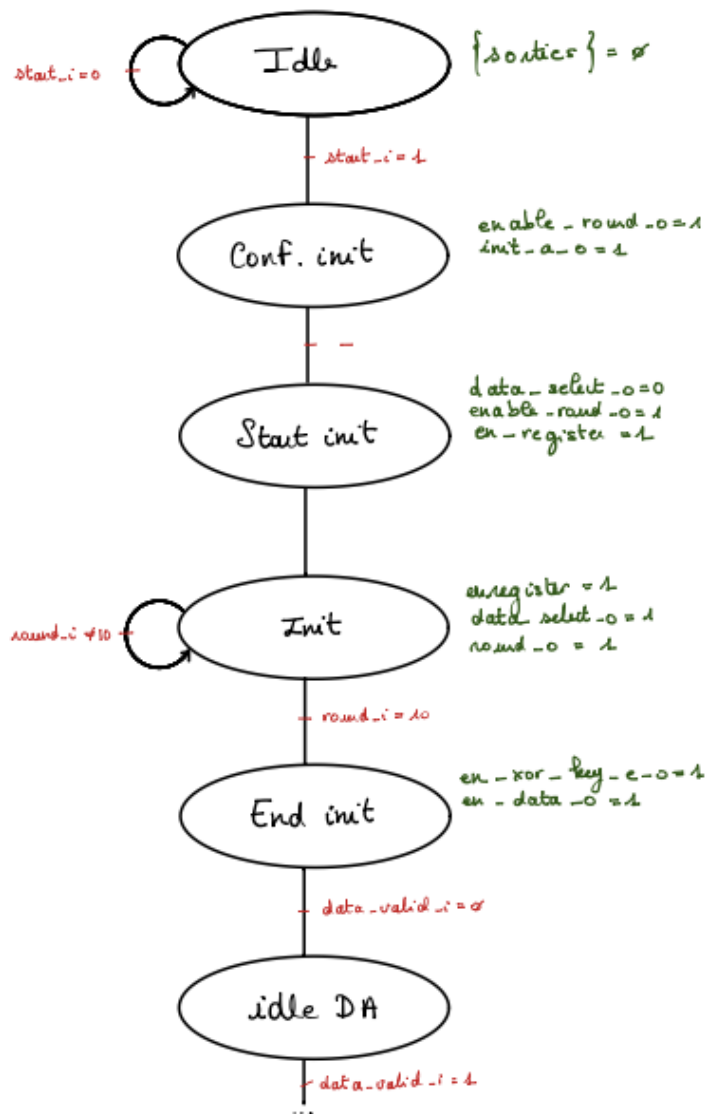


Figure 21 : Diagramme d'état détaillé - 1ère étape

Les états présents sur ce schéma sont les suivants :

- **Idle** : Maintient toutes les variables à zéro, à l'exception des fonctions XOR, en attendant le signal de démarrage pour progresser.
- **Conf Init** : Prépare les permutations pour le prochain cycle d'horloge et met le compteur de rondes à zéro (pour compter jusqu'à 11, soit 12 permutations).
- **Init** : Effectue les permutations jusqu'à ce que le nombre de rondes soit égal à 11.
- **End Init** : Effectue la dernière permutation en désactivant le contournement de la fin pour activer le XOR de sortie de l'étape d'initialisation.

- **Idle DA** : Attend que le signal *data_valid* en entrée passe à 1. Passe à l'état de configuration des données associées lors du prochain cycle d'horloge si la condition est vérifiée. Initialise également le compteur de rondes à 6 (pour avoir seulement 6 permutations lorsque démarré).
- **Conf DA** : Démarre les permutations en désactivant le contournement du XOR au début pour effectuer le XOR en mode initialisation (donc *mode_init_data* = 0).
- **DA** : Effectue les permutations tant que le nombre de rondes n'est pas égal à 11.
- **End DA** : Configure la dernière permutation avec le XOR de fin. Ainsi, définissez le contournement de fin à 0 et le *mode_init_data* à 1 (puisque nous sommes toujours dans la phase d'initialisation).

On peut par ailleurs donner un tableau de l'état des sorties du système. Il est important de rappeler que ces dernières sont actives sur les états présents et non pas sur les transitions comme cela aurait été le cas avec une machine de Mealy.

On démarre le compteur de bloc. Tant que celui-ci ne vaut pas 3, on reste dans la même boucle correspondant à la phase cypher

- **Idle Cipher** : La même condition que précédemment est observée, avec comme prochaine étape la configuration du chiffrement.
- **Conf Cipher** : La première opération de permutation débute avec l'activation de XOR_begin et le démarrage du compteur de tours.
- **Cypher** : Les tours sont comptés jusqu'à ce que le compteur atteigne 11.
- **End Cypher** : La validation du chiffrement est effectuée.

Une fois que le compteur de bloc vaut 3, on peut lancer la dernière étape :

- **Conf Tag** : La première opération de permutation inclut XOR_begin et active *mode_int_end* pour le XOR avec la clé.
- **Tag** : Les permutations se poursuivent jusqu'à 12.
- **End Tag** : La dernière permutation intègre le calcul du tag lors du XOR_end avec la clé.
- **END** : L'état final, où la variable de fin est renvoyée pour indiquer que le calcul des textes chiffrés est terminé, avant de revenir à l'état d'attente idle.

On devra ainsi relier les blocs de la manière suivante :

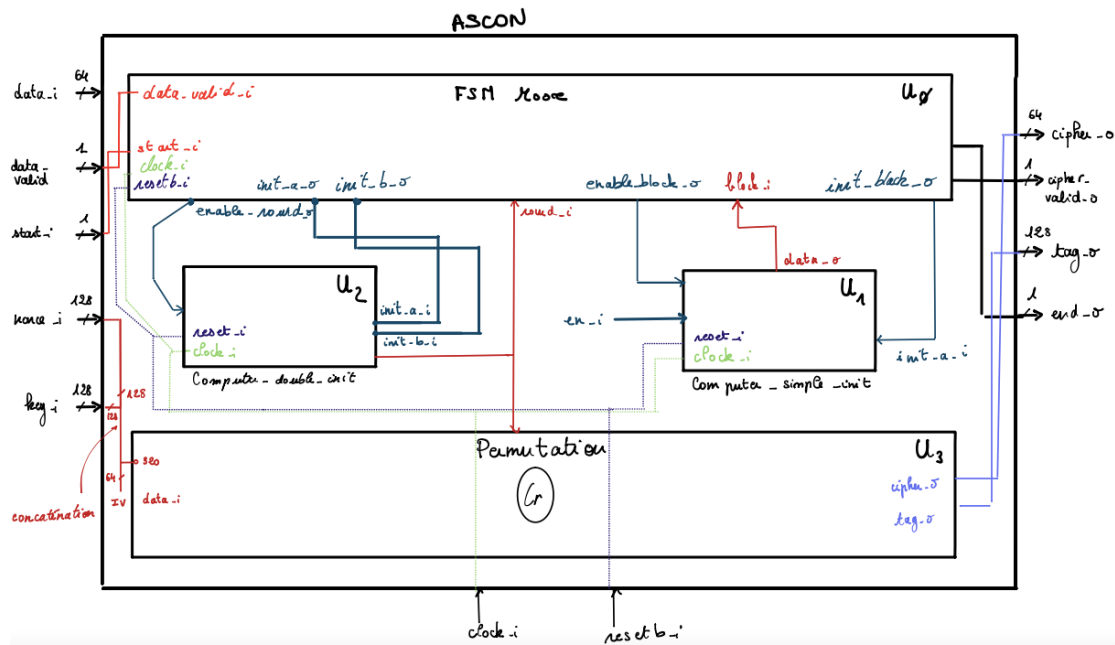


Figure 22 : Machine d'état

Test de la FSM

Conclusion

Problèmes ou difficultés rencontrés

Lors de ce projet, j'ai pu rencontrer des difficultés lors de la simulation des fichiers. Ces derniers compilaient sans erreur ou avertissement mais *ModelSim* ne parvenait pas à exécuter les test benches pour la simulation à cause de paramètres d'optimisation de 'opt2' et 'opt3' impossibles à changer.

Pour résoudre ce problème et obtenir des résultats de simulation, j'ai installé *Icarus Verilog*, un logiciel open-source de simulation matérielle, ainsi que *Modelsim*, avec lequel j'ai dû aussi composer, sur mon ordinateur pour mener à bien les tests.