



Conception de Système Numérique

MODELISATION SYSTEMVERILOG DE
L'ALGORITHME DE CHIFFREMENT
ASCON

Baron Alexandra || EI23

Table des matières

Table des figures	2
Introduction	3
Description de l'algorithme ASCON128	3
1. Processus de chiffrement.....	3
2. Mise en forme, formalisme et structure du projet	4
Implémentation des fonctions élémentaires	5
1. Les permutations p^6 et p^{12}	5
a. L'addition de constante P_C	5
b. La couche de substitution P_S	7
i. La S-Box.....	7
ii. La substitution	8
c. La couche de diffusion linéaire P_1	9
2. Esquisse de permutation et mémorisation de l'état	10
3. Ajout des opérateurs XOR.....	10
a. XOR amont.....	10
b. XOR aval.....	11
4. Permutation finale	12
Architecture globale : « Top Level ».....	13
1. Compteur de rondes.....	13
2. Compteur de blocs.....	13
3. Machine de Moore.....	13
4. Simulation finale	16
Difficultés rencontrées	16
Synthèse	17

Table des figures

Figure 1: Etapes du chiffrement.....	3
Figure 2 : Schéma du chiffrement ASCON 128	4
Figure 3: Organisation du dossier ASCON	5
Figure 4: Fonctionnement de l'addition de constante.....	6
Figure 5: Bloc de l'addition de constante.....	6
Figure 6: Test du bloc de l'addition de constante.....	7
Figure 7: Le module S-Box	7
Figure 8: Test du module S-Box	7
Figure 9 : Fonctionnement de la couche de substitution	8
Figure 10: Le bloc de substitution.....	8
Figure 11: Test du module substitution.....	8
Figure 12 : Fonction de la couche de diffusion.....	9
Figure 13: Bloc de diffusion	9
Figure 14: Test du bloc de diffusion	9
Figure 15 : Permutation sans XOR	10
Figure 16: Test de la première permutation sans XOR	10
Figure 17: Fonctionnement du XOR amont	11
Figure 18 : Fonctionnement du XOR aval.....	11
Figure 19: Permutation finale avec les XOR amont et aval.....	12
Figure 20: Testbench de la permutation finale.....	12
Figure 21 : Permutation finale avec tag et cipher	12
Figure 22: Diagramme des états de la machine de Moore.....	14
Figure 23: Module de la FSM.....	15
Figure 24: ASCON top.....	16
Figure 25: Testbench final.....	16
Figure 26: Erreur après l'état P_P1_End	17

Introduction

Ce projet se concentre sur l'étude et l'implémentation d'un système simplifié de sécurité : l'algorithme de chiffrement ASCON128. Conçu pour répondre aux besoins de confidentialité dans les échanges de données, ASCON128 est une solution efficace pour sécuriser les communications. Dans ce rapport, nous explorerons les principes fondamentaux de cet algorithme, en détail, en utilisant le langage de description matérielle SystemVerilog.

Description de l'algorithme ASCON128

1. Processus de chiffrement

Le processus de chiffrement est réparti en 4 étapes spécifiques. L'algorithme opère sur un état courant de 320 bits, que l'on appelle *state S*. Il est divisé en 5 parties : x_0 , x_1 , x_2 , x_3 et x_4 . La première étape réalisée est l'initialisation. Elle nécessite un vecteur d'initialisation *initialisation vector* IV de 64 bits, une clé secrète *key* K de 128 bits et un nombre arbitraire *nonce* N de 128 bits. La concaténation de ces trois données est bien de 320 bits. Ces données nous sont instaurées par le sujet du projet.

L'étape suivante est celle du traitement des données associées. Elle consiste à injecter dans l'état courant un bloc de données de 64 bits obtenue par padding de A, donnée associée de 48 bits.

Ensuite, se passe le traitement du texte clair, *plaintext*, P, de 184 bits. Par padding on étend cette donnée à 192 bits. On obtient finalement trois vecteurs $\{P_1, P_2, P_3\} = \{P, 10000000\}$. P_1, P_2, P_3 sont chacun de 64 bits. Ces trois vecteurs sont injectés dans l'état courant. Après chaque injection on récupère, séparé de la même manière, les blocs de texte chiffré *ciphertext* C de 184 bits.

Finalement, la dernière étape consiste en la finalisation pour récupérer le tag T de 128 bits. C'est ce bloc de données qui permet de s'assurer la provenance du message chiffré, car il est commun aux deux parties prenantes.

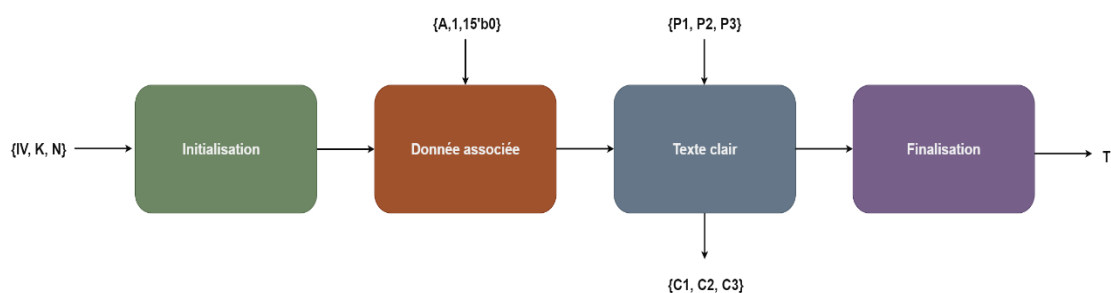


Figure 1: Etapes du chiffrement

Plus concrètement, dans chacune de ces étapes on retrouve des permutations successives, par 6 ou 12. Ces permutations sont tout le temps les mêmes. Elles sont composées des transformations élémentaires suivantes : l'addition de constante, la substitution et la diffusion. Elles seront développées plus tard dans le rapport.

L'insertion des blocs de données évoqués précédemment se fait à travers de bloc de XOR, OU-Exclusif, avec l'état courant *State S*. Il en existe deux types : le XOR amont, *xor_up* et le XOR aval, *xor_down*. Ils se réalisent, soit sur la première permutation d'un bloc de permutation, pour le *xor_up*, soit sur la dernière, pour le *xor_down*.

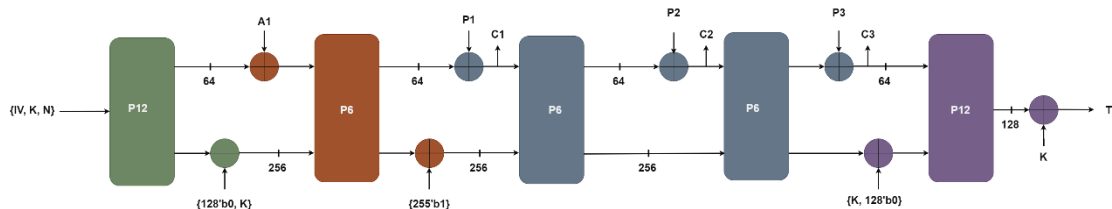


Figure 2 : Schéma du chiffrement ASCON 128

On utilise des registres pour sauvegarder les données nécessaires au destinataire : le texte chiffré *ciphertext* C et le tag T. Un dernier registre est utilisé pour sauvegarder la sortie d'une permutation, au fur et à mesure.

2. Mise en forme, formalisme et structure du projet

Les fichiers de description et de test, en SystemVerilog, ont été séparés de la manière suivante : les codes sources et les fichiers de tests sont placés dans le répertoire SRC, quant aux fichiers résultants de compilations, ils sont dans le répertoire LIB. Ces deux répertoires comprennent deux sous-dossiers chacun.

Le dossier SRC/RTL contient les fichiers sources, dont les bibliothèques associées se trouvent dans le dossier LIB/LIB_RTL. Le dossier SRC/BENCH contient les fichiers de testbench, dont les bibliothèques associées se trouvent dans le dossier LIB/LIB_BENCH. Le fichier *init_models* comprend un script qui initialise le logiciel Modelsim. Le fichier *compile_ascon* comprend un script qui automatise la compilation.

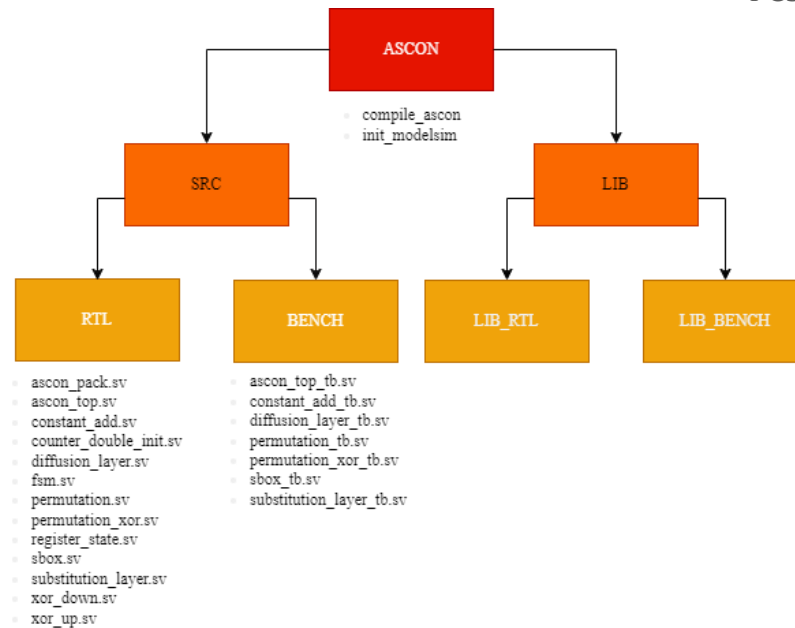


Figure 3: Organisation du dossier ASCON

De plus, nous avons appliqué le formalisme imposé par le sujet. Les entrées d'un module sont suivies par le suffixe « `_i` », les sorties par « `_o` » et les signaux internes par « `_s` ». Pour simplifier la lecture et la programmation, certains signaux peuvent contenir les deux suffixes. C'est un choix personnel.

Implémentation des fonctions élémentaires

Comme évoqué précédemment, on retrouve dans l'algorithme ASCON 128, plusieurs transformations élémentaires. Elles ont donc été réalisées une par une pour éviter toute erreur. Chaque fonction a également son propre testbench. Nous allons les aborder dans les parties suivantes.

1. Les permutations p^6 et p^{12}

On commence par mettre en place les permutations. Elles nécessitent donc trois fonctions élémentaires.

a. L'addition de constante P_C

On commence par mettre en place les permutations. La première fonction élémentaire nécessaire pour réaliser une permutation est l'addition de constante. Elle agit uniquement sur le registre x_2 , c'est-à-dire, `state_S[2]`, et consiste à y additionner une

constante définie, appelée *Cr*. La constante dépend du numéro de la permutation en cours. Ce numéro est compris entre 0 et 11. Cette opération est concrètement un XOR entre la constante et le dernier octet de x_2 . Voici un schéma de cette opération. Les opérations sur les autres registres de l'état courant sont équivalentes à des fils.

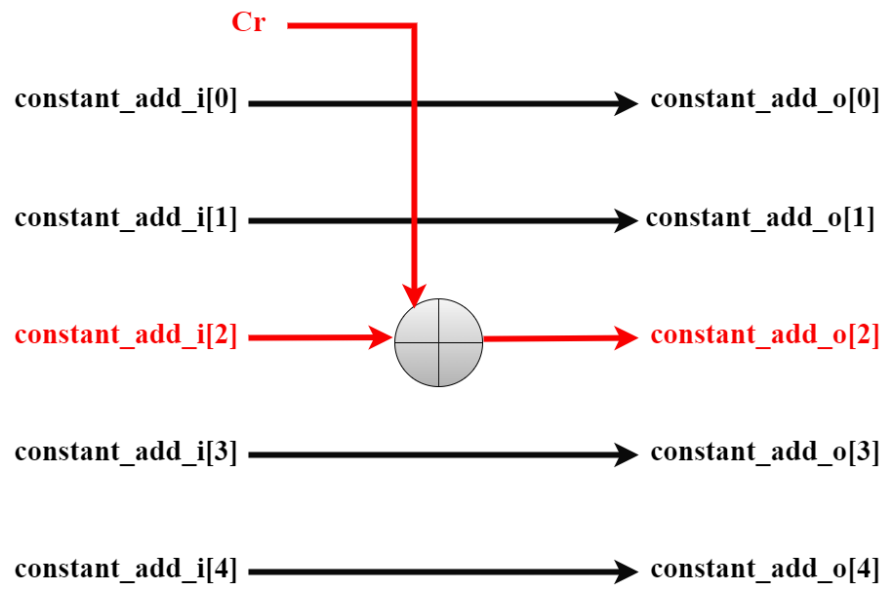


Figure 4: Fonctionnement de l'addition de constante

Voici une modélisation simple de cette fonction élémentaire.

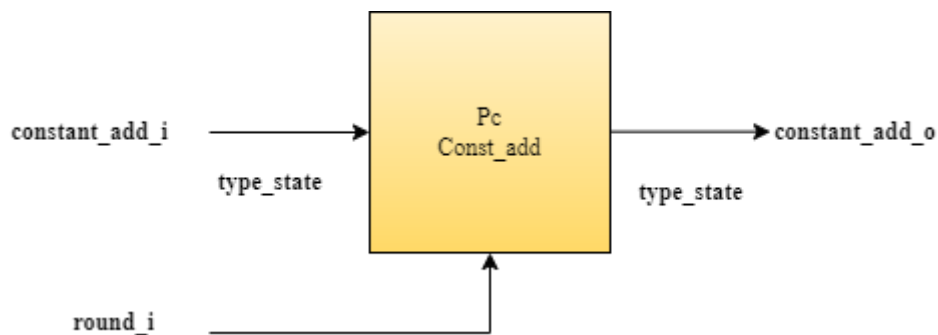




Figure 5: Bloc de l'addition de constante

Nous avons réalisé un testbench pour vérifier le fonctionnement de cette fonction. Nous prenons en entrée un vecteur nul, pour faire apparaître la constante Cr .

Número de ronda		0	1	2	3	4
Constante Cr		f0	e1	d2	c3	b4
 constant_add_i_s  constant_add_o_s[z]  round_s	64h00000000... 64h00000000... 4'hb	0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 00000000000000F0 00000000000000E1 00000000000000D2 00000000000000C3 00000000000000B4 0	1	2	3	4

5	6	7	8	9	10	11
a5	96	87	78	69	5a	4b
0000000000000000A5 000000000000000096 000000000000000087 000000000000000078 000000000000000069 00000000000000005A 00000000000000004B						
5	6	7	8	9	a	b

Figure 6: Test du bloc de l'addition de constante

b. La couche de substitution P_s

La permutation de ASCON est composée ensuite d'une couche de substitution où, 64 S-box de 5 bits, sont appliquées en parallèle. Nous implémentons donc premièrement la S-box.

i. La S-Box

La S-Box prend en entrée 5 bits, et en fonction de la valeur de cette entrée, la substitue en une autre valeur. Les valeurs par lesquelles on substitue sont déterminées par le sujet. Voici une modélisation simple de cette fonction élémentaire.

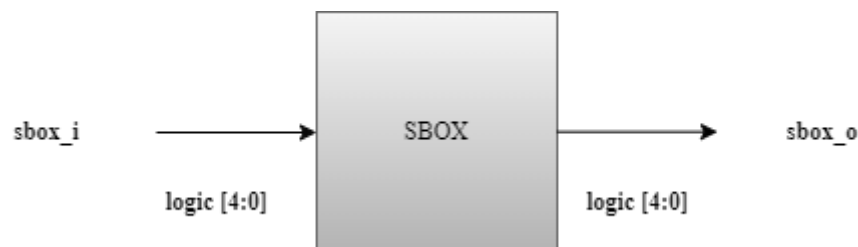


Figure 7: Le module S-Box

Nous avons réalisé un testbench pour vérifier le fonctionnement de la s-box. Nous prenons en entrée un vecteur allant de 0 à 31, pour faire apparaître les valeurs de la S-Box.

sbox_i_s	5h14	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f	10	11	12	13	14
sbox_o_s	5h00	04	0b	1f	14	1a	15	09	02	1b	05	08	12	1d	03	06	1c	1e	13	07	0e	00

sbox_i_s	5h1f	0f	10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f				
sbox_o_s	5h17	1c	1e	13	07	0e	00	0d	11	18	10	0c	01	19	16	0a	0f	17				

Figure 8: Test du module S-Box

ii. La substitution

L'étape de substitution revient à appliquer la S-Box, 5 bits par 5 bits, à l'état courant.

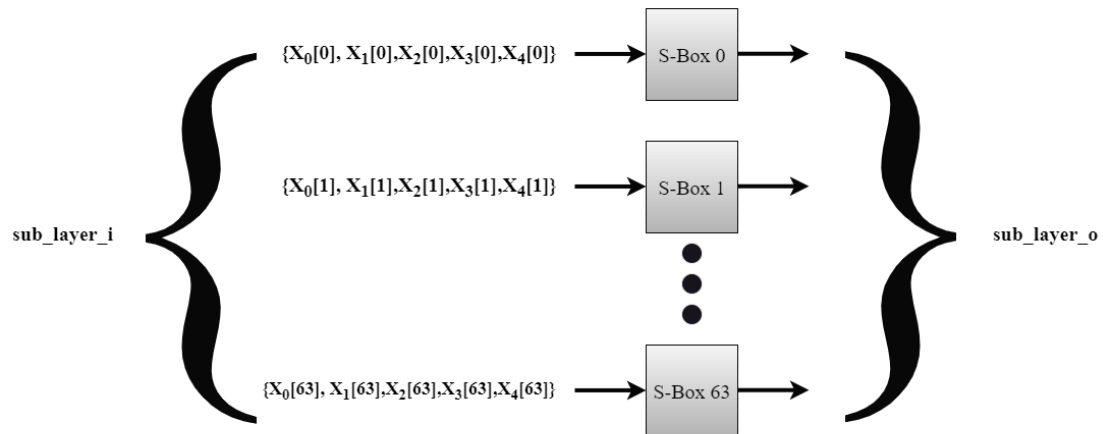


Figure 9 : Fonctionnement de la couche de substitution

Voici une modélisation simple de cette fonction élémentaire.

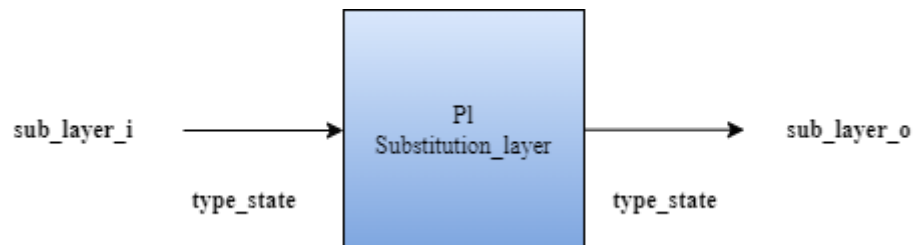


Figure 10: Le bloc de substitution

Nous avons réalisé le testbench pour vérifier le bon fonctionnement de la fonction. Nous avons pris en entrée la sortie du bloc d'addition de constante à la toute première permutation.

sub_layer_i_s	64'h80400c06...	80400c0600000000 8a55114d1cb6a9a2 be263d4d7aecaa0f 4ed0ec0b98c529b7 c8cddf37bcd0284a
sub_layer_o_s	64'h78e2cc41f...	78e2cc41faabaa1a bc7a2e775aababf7 4b81c0cbbdb5fc1a b22e133e424f0250 044d33702433805d

Figure 11: Test du module substitution

c. La couche de diffusion linéaire P_i

La couche de diffusion consiste à appliquer une fonction prédéfinie à chacun des 5 registres de l'état courant. Voici la fonction appliquée, extraite du sujet.

Fonction linéaire Σ_i

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

Figure 12 : Fonction de la couche de diffusion

Cette fonction réalise en fait des rotations cycliques vers la droite. Plus simplement, pour $x_i \ggg j$, nous décalons les bits du registre i , j fois vers la droite. Voici le module de cette fonction élémentaire.

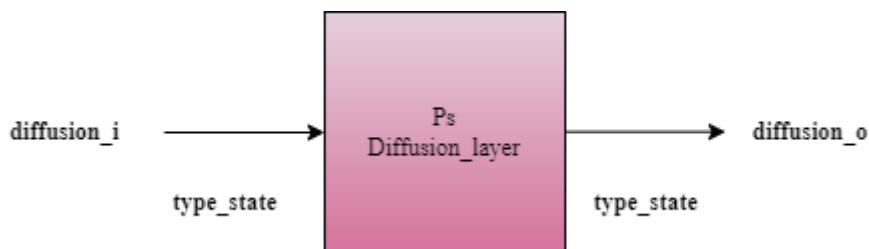


Figure 13: Bloc de diffusion

Nous avons réalisé le testbench pour vérifier le bon fonctionnement de la fonction. Nous avons pris en entrée la sortie du bloc de diffusion à la toute première permutation.

diffusion_i_s	64'h78e2cc41f...	78e2cc41faabaa1a bc7a2e775aababf7 4b81c0cbbdb5fc1a b22e133e424f0250 044d33702433805d
diffusion_o_s	64'ha71b22fa2...	a71b22fa2d0f5150 b11e0a9a608e0016 076f27ad4d99d5e7 a72ac1ad8440b0b7 0657b0d6eaf9c1c4

Figure 14: Test du bloc de diffusion

Nous avons désormais tous les modules permettant de réaliser une première esquisse de permutation, qui ne comprend pas les XOR amont et aval. Nous utilisons un registre, qui nous a été fourni, pour enregistrer les valeurs de sortie de permutation. On rajoute également un multiplexeur commandé pour, soit prendre comme état courant *State S* l'état en entrée *state_i*, soit la sortie du registre, c'est-à-dire, la fin de la permutation précédente. Voici le schéma de cette esquisse de permutation.

[illegible]

Figure 16: Test de la première permutation sans XOR

Comme on le voit sur la figure 2, il y a des opérateurs XOR à ajouter dans la permutation, soit en début, soit en fin d’une succession de permutations.

Le *xor_up*, il prend en paramètre le registre x_0 de l'état courant et un bloc de données sur 64 bits. Voici un schéma qui explique le fonctionnement de ce bloc.

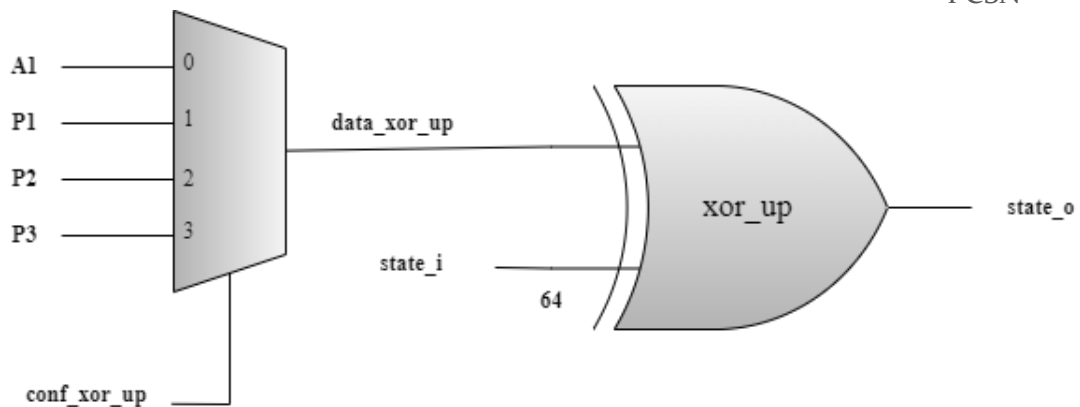


Figure 17: Fonctionnement du XOR amont

En effet, nous avons mis en place un signal *conf_xor_up*, qui permet de déterminer quelle donnée sera choisie pour réaliser l'opération. Cela équivaut à un multiplexeur.

b. XOR aval

Concernant le *xor_down*, il prend en paramètre les registres x_1 , x_2 , x_3 et x_4 de l'état courant et un bloc de données sur 256 bits. Voici un schéma qui explique le fonctionnement de ce bloc.

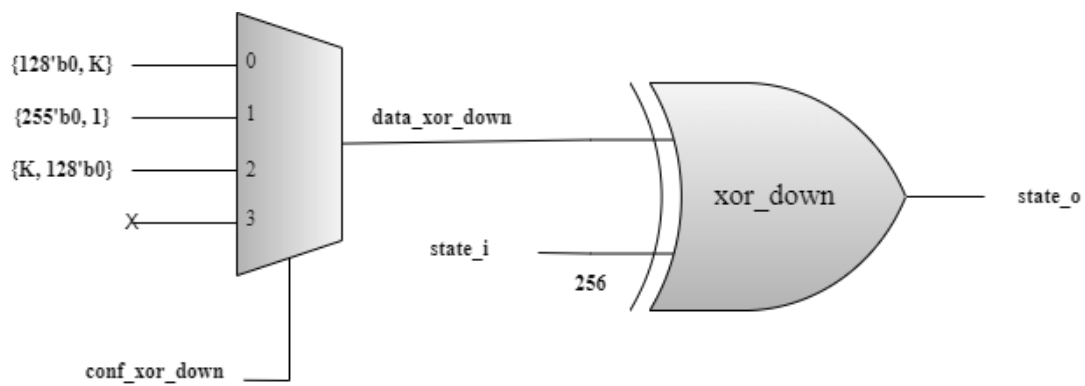


Figure 18 : Fonctionnement du XOR aval

On commence par ajouter à la permutation précédente les deux XORs. Voici le schéma correspondant au fonctionnement de la permutation.

[illegible]

Figure 20: Testbench de la permutation finale

Finalement, il est nécessaire d'y ajouter deux registres permettant de sauvegarder le *tag* T et le *ciphertext* C. Voici le schéma final de la permutation.



Architecture globale : « Top Level »

L'architecture globale comprend le flot de contrôle et le flot de données. Il est nécessaire de gérer l'architecture de contrôle des permutations. Pour cela on met en place un compteur de permutation, compteur de rondes. On pourrait également mettre en place un compteur de blocs. Finalement il est nécessaire de construire une machine de Moore pour le contrôle.

1. Compteur de rondes

Comme évoqué précédemment pour pouvoir répéter un certain nombre de fois les permutations, nous avons utilisé un compteur de rondes. Il gère la variable *round*. Pour cela, nous avons utilisé le module *counter_double_init*, qui permet d'initialiser le compteur soit à 0 pour réaliser 12 permutations, soit à 6 pour en réaliser 6. Ce module incrémente également le compteur à l'aide du signal *ena_cpt_o*.

2. Compteur de blocs

Nous aurions également pu mettre en place un compteur de blocs pendant la réalisation de l'étape de traitement du texte clair. Je n'ai pas réalisé ce compteur. Ce point bloquant est abordé plus bas.

3. Machine de Moore

Il est donc nécessaire maintenant de réaliser la machine d'états finis. Elle contrôle les compteurs et la permutation avec les XOR. Pour simplifier sa compréhension, j'ai réalisé un diagramme d'états. Nous y avons laissé le même code couleur que sur la figure 2, ce qui permet de suivre facilement l'avancé des états. Elle comprend 22 états. Elle fonctionne avec un processus séquentiel qui permet de mémoriser la valeur de l'état courant, un processus combinatoire pour faire les transitions entre les états et un dernier processus combinatoire pour changer les valeurs des sorties.

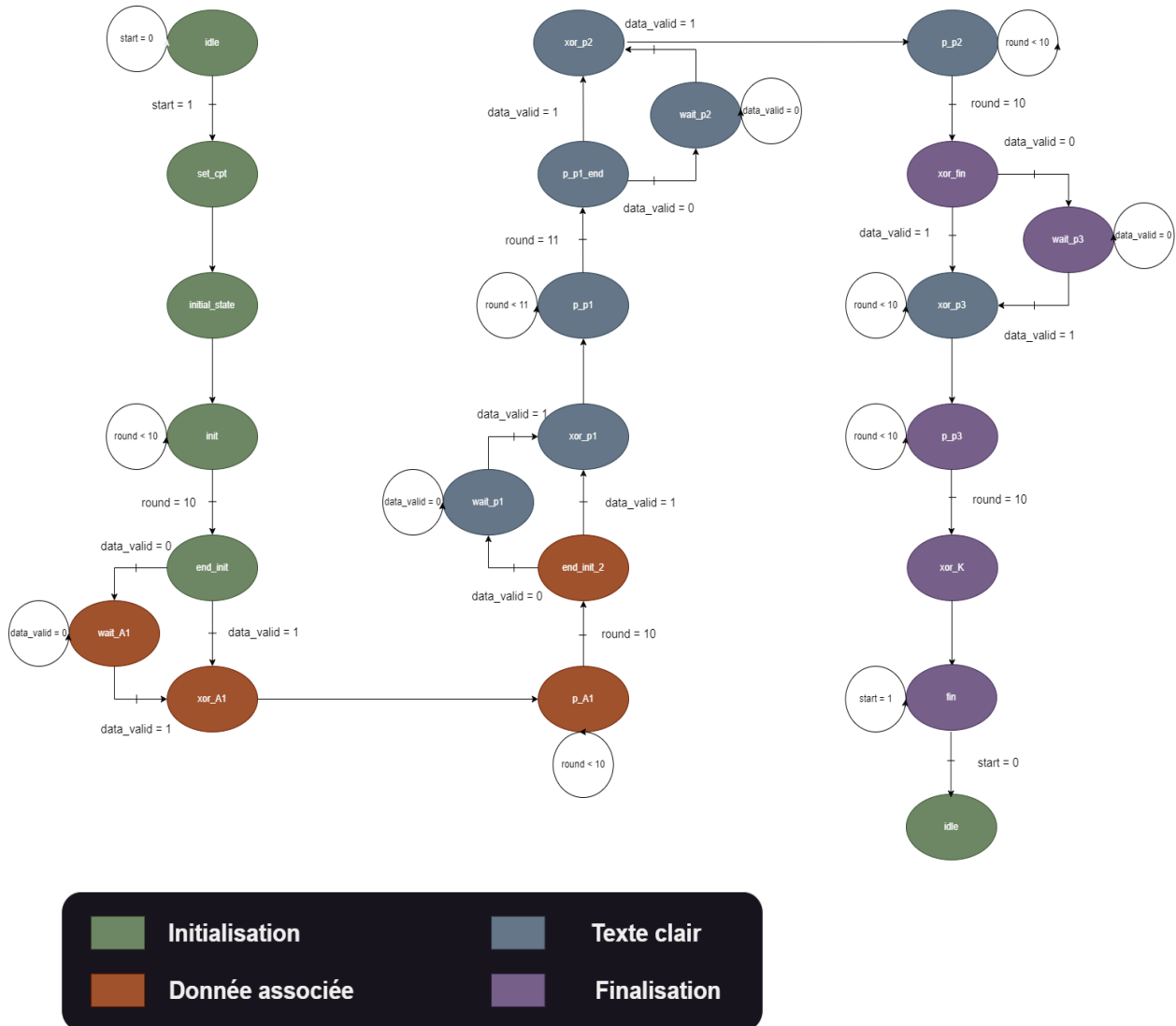


Figure 22: Diagramme des états de la machine de Moore

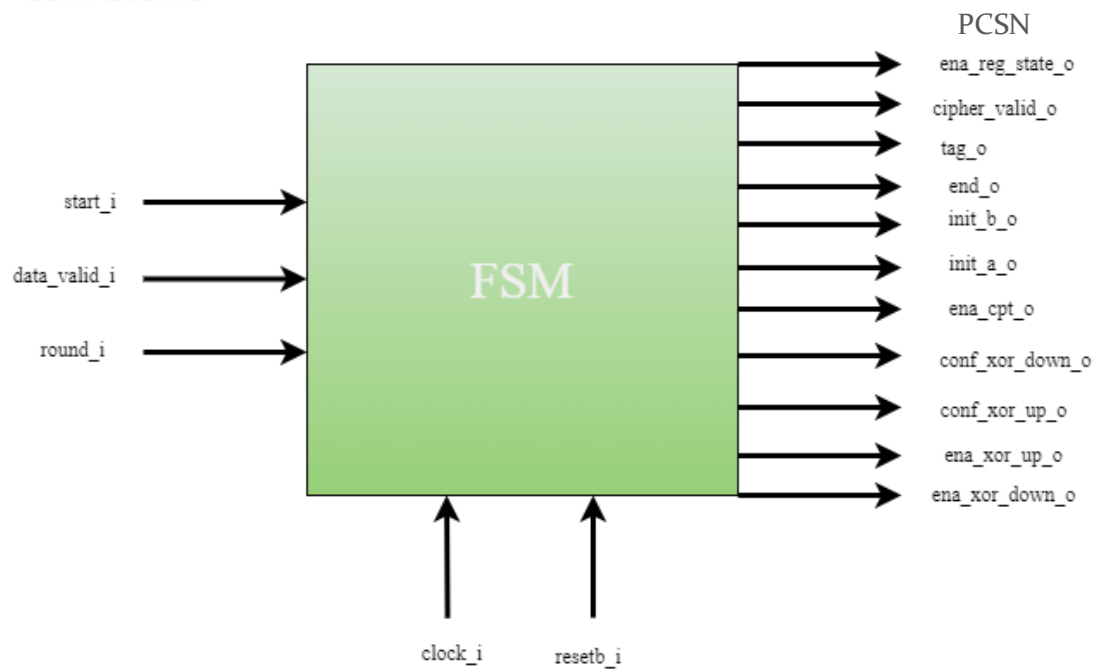


Figure 23: Module de la FSM

4. Simulation finale

Pour réaliser la simulation finale, on implémente un module nommé *ascon_top*.

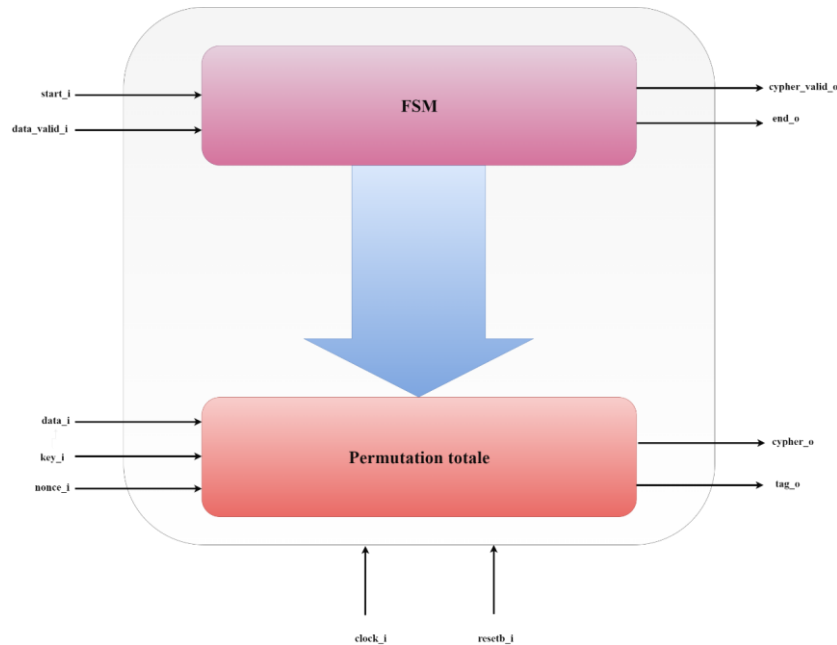


Figure 24: ASCON top

On réalise le testbench, pour vérifier le résultat final. On se rend compte qu'il y a une erreur dans l'étape de traitement du texte clair. J'aborde ce point dans les difficultés rencontrées.

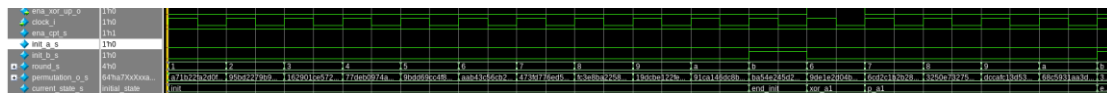


Figure 25: Testbench final

Difficultés rencontrées

Ce projet a été particulièrement chronophage. J'ai rencontré beaucoup de points bloquants, dont la majorité est sans importance.

Cependant, voici les points qui m'ont particulièrement gênée durant ce projet.

Premièrement, je me suis rendue compte assez tard qu'il me manquait un état dans la FSM pour pouvoir réaliser correctement l'enchaînement entre la permutation p6 et le lancement du XOR avec le texte clair P2. Je ne pouvais pas initialiser correctement le compteur de ronde. J'ai donc ajouté un état « intermédiaire », que j'ai appelé *p_p1_end*. Cette erreur m'a forcée à reprendre mes codes. Comme montré précédemment, cet état

supplémentaire a chamboulé ma simulation. Après cet état mes résultats ne concordent plus avec ceux attendus. Je me rends donc à l'évidence que le problème provient forcément de cet état *p_pl_end*. Je n'ai pas réussi à régler ce problème.

round_s	4th	a	0	c	6
permutation_o_s	64ha7axxxa...	6052ab98b	0b07c38c42988c646c8e20d4c33b38c644b7b91349b2b52b0846f7b0b96c70516988b11	5400ee585f74814b972b048b3b005c7018c484b750331c4b2b45811b0bc664b240b255e	5f27f8a26b2053109486637e445844b67b4b3
current_state_s	initial_state	p_p1	p_p1_end	xor_p2	
conf_xor_up_s	2th	0		12	

Figure 26: Erreur après l'état *P_P1_End*

Ensuite, pour la configuration du XOR amont, *xor_up*, j'ai décidé d'utiliser la même manière de configuration que celle du XOR aval, *xor_down*. J'avais donc deux signaux sur 2 bits chacun : *conf_xor_up* et *conf_xor_down*. J'ai pris beaucoup de temps à comprendre comment les configurer dans le module final. Je n'étais pas sûre de s'il fallait uniquement les faire apparaître dans le testbench final, ce qui me paraissait bizarre, ou de les faire apparaître dans le module finale, *ascon_top*. C'est au bout d'un certain temps que j'ai pensé à réaliser un processus combinatoire pour chaque signal.

Finalement, ce qui m'a beaucoup fait réfléchir, c'est l'utilité et le fonctionnement du compteur de bloc. En effet, je n'ai pas saisie l'utilité de réaliser une boucle sur ces blocs. Il n'y a pas grand-chose qui se répète en dehors de l'enchaînement de 6 permutations « xorées » avec un extrait du texte clair. Cependant, sur la dernière permutation, il faut ajouter le XOR avec la clé complétée de 0. Ce qui nous obligerait d'ajouter une condition dans la boucle.

Synthèse

Pour conclure, j'ai particulièrement apprécié ce projet, malgré le fait qu'il a été particulièrement chronophage. Il m'a permis de mettre en pratique un cours théorique à un sujet assez intéressant. Je remercie tout particulièrement Mr Guillaume Reymond pour son aide sur ce projet.