17 Décembre 2023

Rapport de Projet Conception de Système Numérique

Timothée CLOUP-MARTIN
EI 22

1. Introduction

a. Objectif

Ce projet a été initié dans le but de nous familiariser avec l'utilisation du SystemVerilog, une extension du langage de description matérielle Verilog. Pour ce faire, il nous a été demandé d'implémenter un algorithme de chiffrement des données selon le protocole AEAD. La méthode cryptographique AEAD ou Authenticated Encryption with Associated Data combine à la fois le chiffrement pour assurer la confidentialité des données et l'authenticité de ces données. Nous allons implémenter une version simplifiée de l'algorithme ASCON128 pour réaliser un chiffrement authentifié AEAD.

b. Présentation de l'algorithme ASCON128

L'algorithme de chiffrement symétrique ASCON128 opère sur un message en entrée de 256 bits pour produire un message chiffré de sortie de 248 bits. Ce chiffrement va se faire selon 4 étapes :

Étape d'initialisation : Le système prend en entrée un message de 256 bits comprenant : -Le vecteur d'initialisation (IV) de 64 bits pour initier l'état interne de l'algorithme.

-La clé secrète (K) de 128 bits nécessaire pour décrypter le message

-Le nonce qui est un code de 128 bits permettant l'unicité des opérations afin d'éviter les attaques par rejeu.

La seconde étape consiste en l'injection des données associées (de taille 32 bits) dans l'algorithme (Asssociated Data). Ces données font référence aux éléments distinctifs du message, tels que l'adresse IP, la taille du message etc.

Il vient ensuite l'étape de chiffrement du message. Le texte clair P (plaintext) de 248 bits va être inséré dans l'algorithme par tranche de 64 bits. Il sera ensuite chiffré à l'aide d'opérations élémentaires pour devenir un ciphertext C (texte chiffré).

La dernière étape est la finalisation. Cette étape va permettre de générer le tag de 128 bits qui permet au destinataire d'être sûr que le message déchiffré provient bien du bon expéditeur.

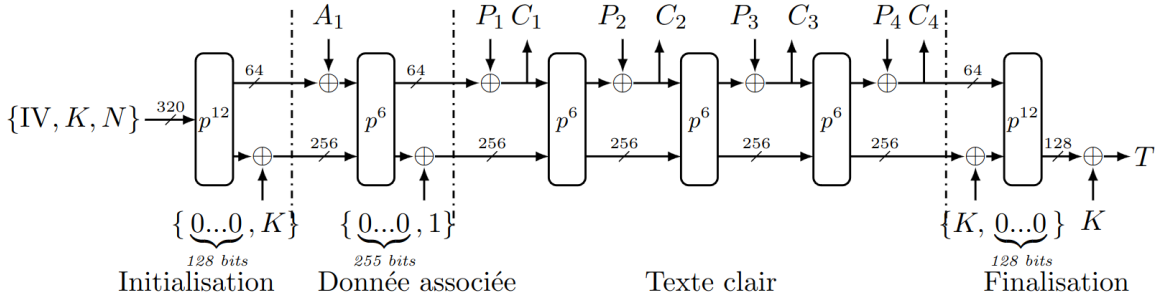


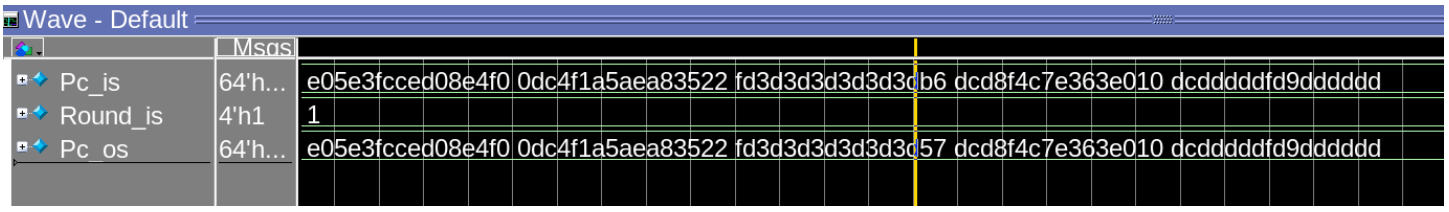
Schéma du chiffrement ASCON128

2. Permutations

ASCON s'appuie sur une fonction cryptographique appelée fonction éponge. L'algorithme va d'abord absorber les données en entrée dans l'état interne. L'état courant est composé de 5 registres de 64 bits ce qui fait au total un état de 320 bits .Ensuite vient la phase de pressage durant laquelle le système génère une sortie continue correspondant au différents blocs du message chiffré. Le chiffrement est un chiffrement par bloc. Nous appelons ce bloc une permutation. La permutation (P) consisté en l'itération successive (6 à 12 fois) de 3 opérations qui sont l'addition de constante (Pc), la couche de substitution (Ps) et la diffusion linéaire (Pl).

a. L'addition constante Pc

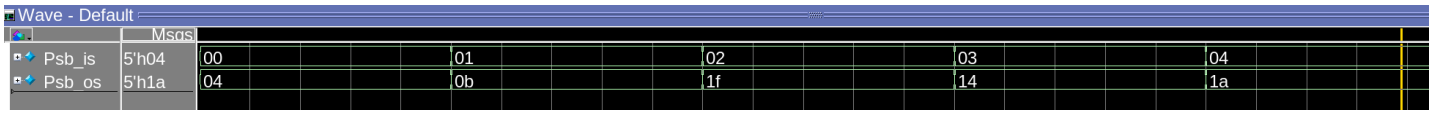
L'addition constante consiste en l'ajout d'une constante appelée constante de ronde au second registre de l'état S. Cette constante dépend du nombre de tours qu'a déjà effectué S.



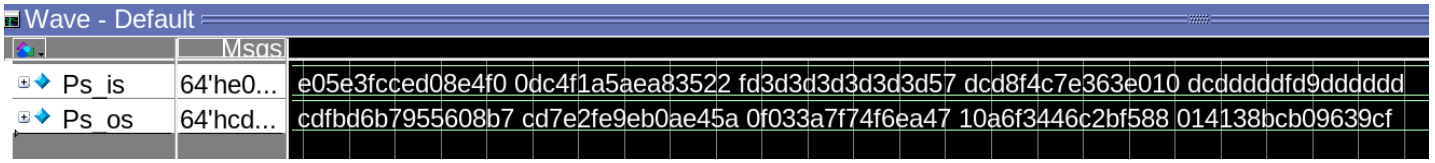
Résultat du test bench de la phase d'addition de constante. On peut voir que les 8 derniers bits (soit les 2 derniers hexadécimaux) du second registre ont bien été modifiés. On a bien h'b6 ⊕ h'e1=h'57.

b. La couche de substitution Ps

L'étape de substitution consiste en la modification de l'ensemble des 320 bits de l'état S. Les ièmes bits des 5 registres de S vont former un nombre de 5 bits qui sera remplacé par la valeur donnée dans le tableau de substitution.



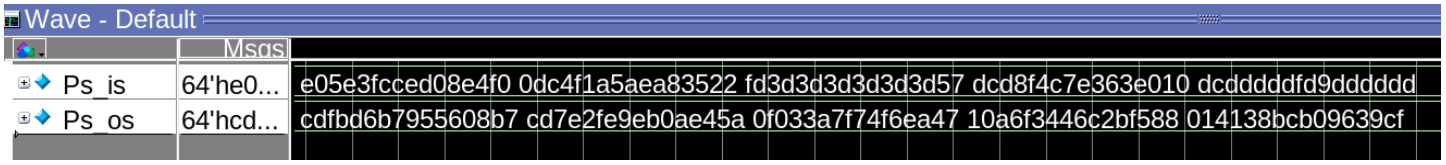
Résultat du test bench de l'implémentation du tableau de substitution.



Résultat du test bench de la phase d'addition de substitution. L'ensemble de l'état S a été modifié. Prenons le premier bit de chaque registre (b'10111=h'17), on est bien en sortie les 5 bits (b'11000=h'18).

c. La diffusion linéaire Pl

La dernière phase d'une permutation est la phase de diffusion linéaire Pl combine à la fois un décalage de bit et une opération xor (ou exclusif) pour chaque registre de l'état S. Lors du décalage, les bits qui ne peuvent pas être davantage décalés vers la droite se sont réinsérés à gauche du registre (bit de poids fort).



Résultat du test bench de la phase de diffusion linéaire. La sortie est en accord avec le résultat attendu.

d. Ajout du Multiplexeur et de la D-flip-flop

Deux autres éléments sont nécessaires pour la permutation : le multiplexeur et la d-flip-flop. Le mux va permettre de choisir entre l'état de sortie du tour précédent comme entrée pour un tour donné ou l'état qui entrera dans le bloc de permutation s'il s'agit du premier tour. La d-flip-flop permet quant à elle de stocker les valeurs de sortie d'un tour de la permutation. C'est aussi en fonction de la d-flip-flop que l'on va synchroniser le bloc permutation avec les autres modules du système.

3. Xor_begin et Xor_end

Des opérateurs xor sont ajoutés au sein d'un bloc permutation afin de pouvoir ajouter dans l'algorithme ASCON le texte clair ou encore les données associés. Ces xors vont aussi avoir pour but de complexifier davantage le chiffrement et générer le tag nécessaire à l'authentification. L'intérêt de l'opérateur xor est que c'est un opérateur non-linéaire. L'introduction d'opérateur non linéaire renforce ainsi l'algorithme.

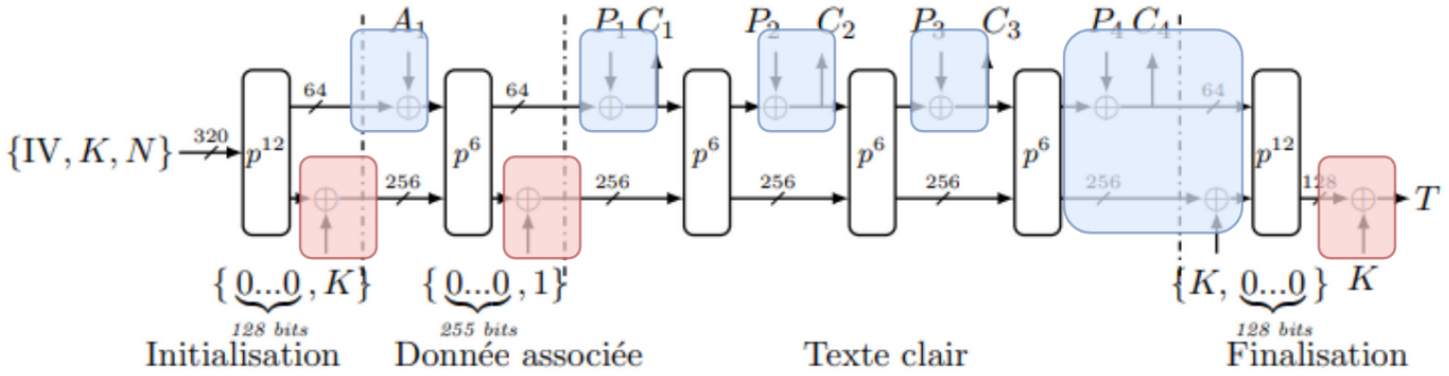
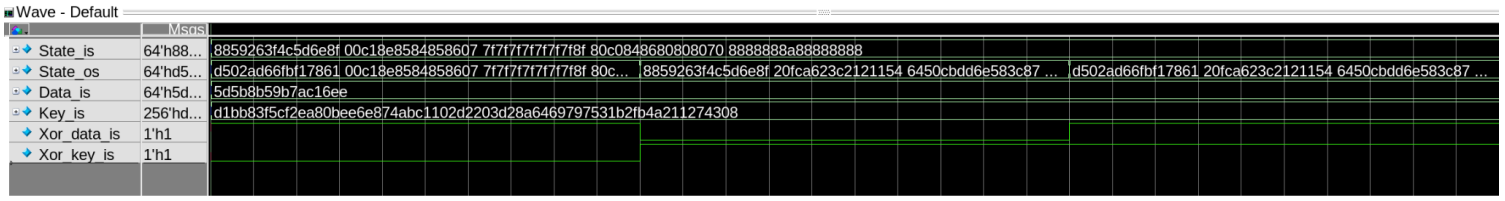


Schéma du chiffrement ASCON. En bleu les opérations xor programmées par Xor_begin, en rouge celles programmées par Xor_end.

a. Xor_begin

L'opération Xor_begin va servir la plupart du temps au système d'absorber le texte clair par tranche de 64 bits(P1,P2,P3,P4) pour l'insérer dans la source interne. Nous allons aussi utiliser l'opération Xor_begin pour récupérer les données associées. Enfin, afin de ne pas complexifier notre programme en ajoutant un nouveau module xor, nous allons combiner le xor qui permet d'injecter le dernier plaintext P4 dans l'état interne et le xor qui injecte la clé K dans la l'état externe lorsqu'on est en amont du

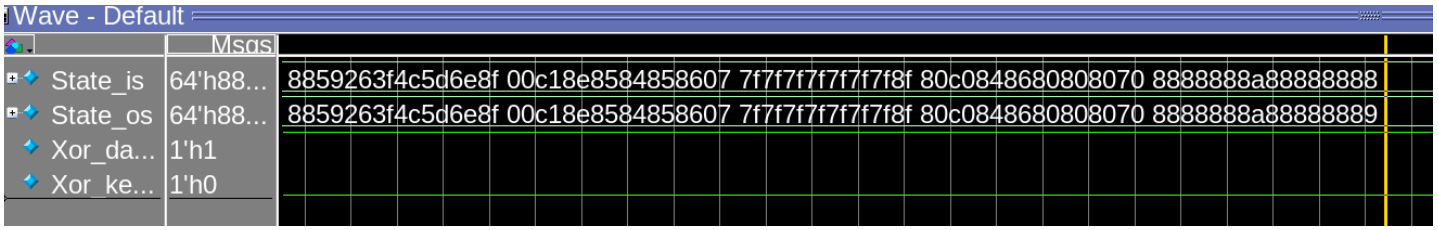
bloc de permutation lors de l'étape de finalisation. Nous distinguons donc 3 utilisations possibles du Xor_begin. Ces trois options seront codées sur deux bits à l'aide de deux variables binaires logiques : xor_key_i et xor_data_i.



Résultat du test bench de l'opérateur Xor_begin. On peut voir sur la sortie que les deux variables binaires xor_key_i et xor_data_i permettent bien d'implémenter les 3 différentes solutions du programme.

b. Xor_end

Le programme de l'opération Xor_end est très similaire à celui du Xor_begin. Cependant celui-ci n'opère que sur la source interne Sc. On l'utilise ce xor pour 3 opérations différentes interprétées par les mêmes variables que pour Xor_begin. Xor_end permet d'injecter la clé K sur l'état externe après le bloc de permutation lors de la phase d'initialisation. Il est aussi utilisé pour dans la phase de donnée associée. Enfin, il constitue la dernière étape de chiffrement avant la publication du tag lors de la phase de finalisation.



Résultat du test bench de l'opérateur Xor_end. Nous avons simulé le cas correspondant au xor de l'étape 2. On peut remarquer sur le dernier hexadécimal que l'opération a bien eu lieu.

4. Bloc Permutation

a. Cipher et Tag

Deux nouvelles D-flip-flop ont été ajouté au module Permutation pour pouvoir sauvegarder la valeur du cipher (DFFC) et la valeur du tag (DFFT). La valeur du cipher va être récupérer entre 2 blocs de permutation tranche par tranche (C1 à C4) dans la sortie continue du système. Le tag quant à lui sera récupéré à la fin du parcours de l'algorithme.

b. Simulation du bloc de permutation

Les modules Pc, Ps, Pl, Xor_begin, Xor_end ainsi que la D-flip-flop et le multiplexeur vont permettre d'implémenter un bloc permutation.

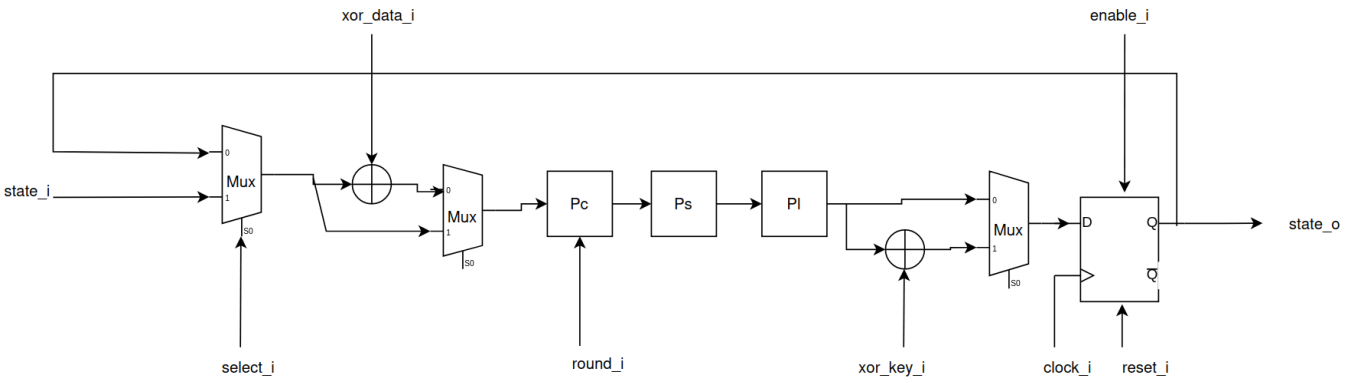
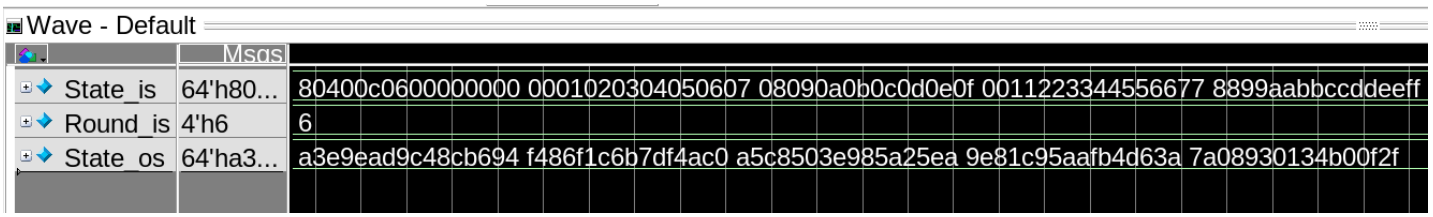
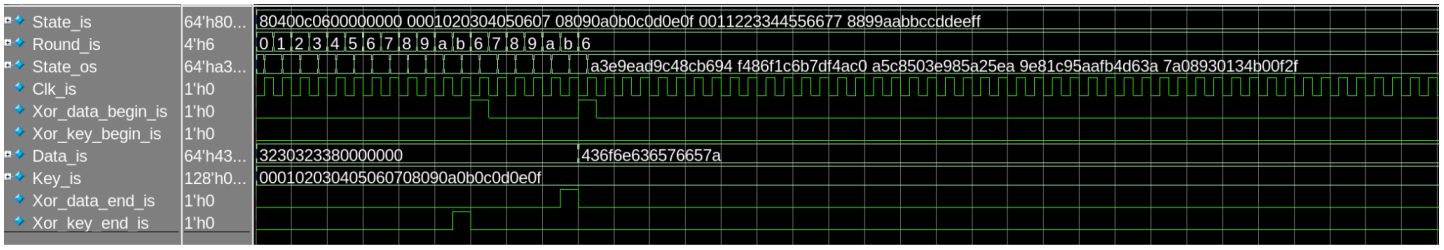


Schéma d'un bloc de permutation



Résultat du test_bench du module permutation4. Ce module vise à tester l'efficacité d'un bloc de permutation ainsi que des opérateurs xor qui lui sont intégrés. Le résultat attendu correspond bien au premier round du bloc permutation de la phase de donnée associée.



Chronogramme du déroulement complet du programme permutation4.sv . On peut voir le changement de state_o après chaque itération de boucle ainsi que l'activation des différents modes du xor_begin et du xor_end.

5. Machine d'états finis

a. Compteur de tours et compteur de blocs de permutation

Deux compteurs sont intégrés dans la conception de la machine d'états finis : le compteur de tours et le compteur de blocs. Le compteur de tours est utilisé pour déterminer la position actuelle dans le bloc de permutation. La valeur de la constante d'addition est déterminée par ce compteur. En fonction du type de bloc de permutation (p6 ou p12), deux variables d'initialisation sont déclarées pour faciliter la réinitialisation du compteur. Il peut y avoir au maximum 12 tours donc le compteur est codé sur 4 bits. Ce compteur sera une condition pour synchroniser les différents modules de l'algorithme dans la fsm.

Le compteur de blocs va quant à lui permettre de se situer dans la phase de chiffrement. Il y a 3 blocs de permutation de type p6 nécessaires pour injecter les 4 plaintext P1 à P4 de 64 bits. C'est pourquoi le compteur est programmé sur 2 bits. Ce compteur sera lui aussi une condition pour synchroniser les différents modules de l'algorithme dans la fsm.

b. Machine de Moore

La machine d'états finis (FSM) doit être en mesure de sélectionner de configurer les variables de contrôle pour un état donné. La FSM est principalement dirigée par la variable la variable data_valid_i, qui indique la validité des données et la variable start_i, qui déclenche le début du chiffrement, permettant ainsi à la machine de progresser. Dans le cas contraire, si data_valid_i n'est pas activé, l'ensemble de l'algorithme est stoppé. De plus, la machine d'état est responsable de la gestion des compteurs et de tous les signaux qui contrôlent les opérations de permutation. Enfin, la machine d'états finis aura pour rôle d'incrémenter les compteurs et de superviser tous les signaux liés au contrôle des opérations de permutation.

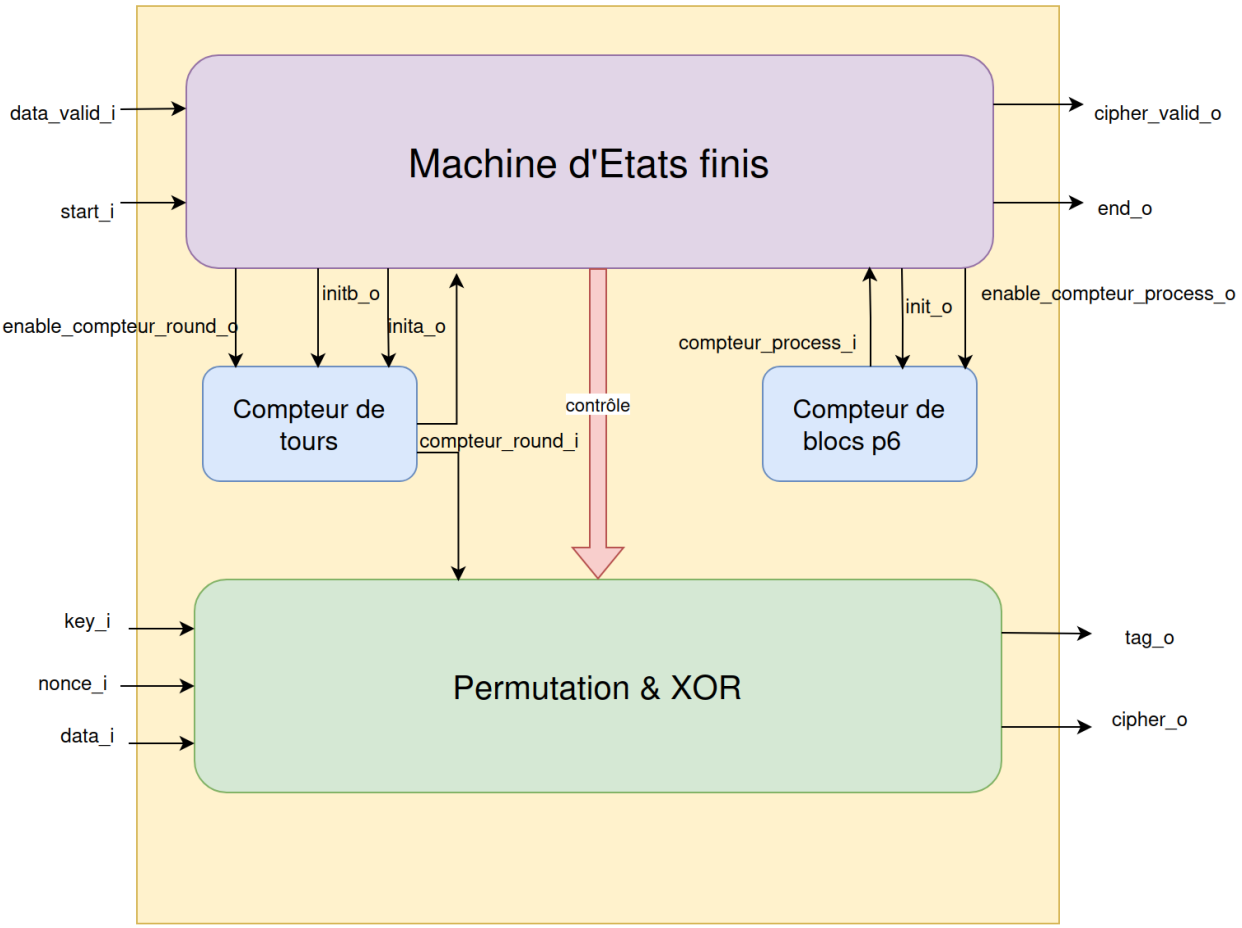
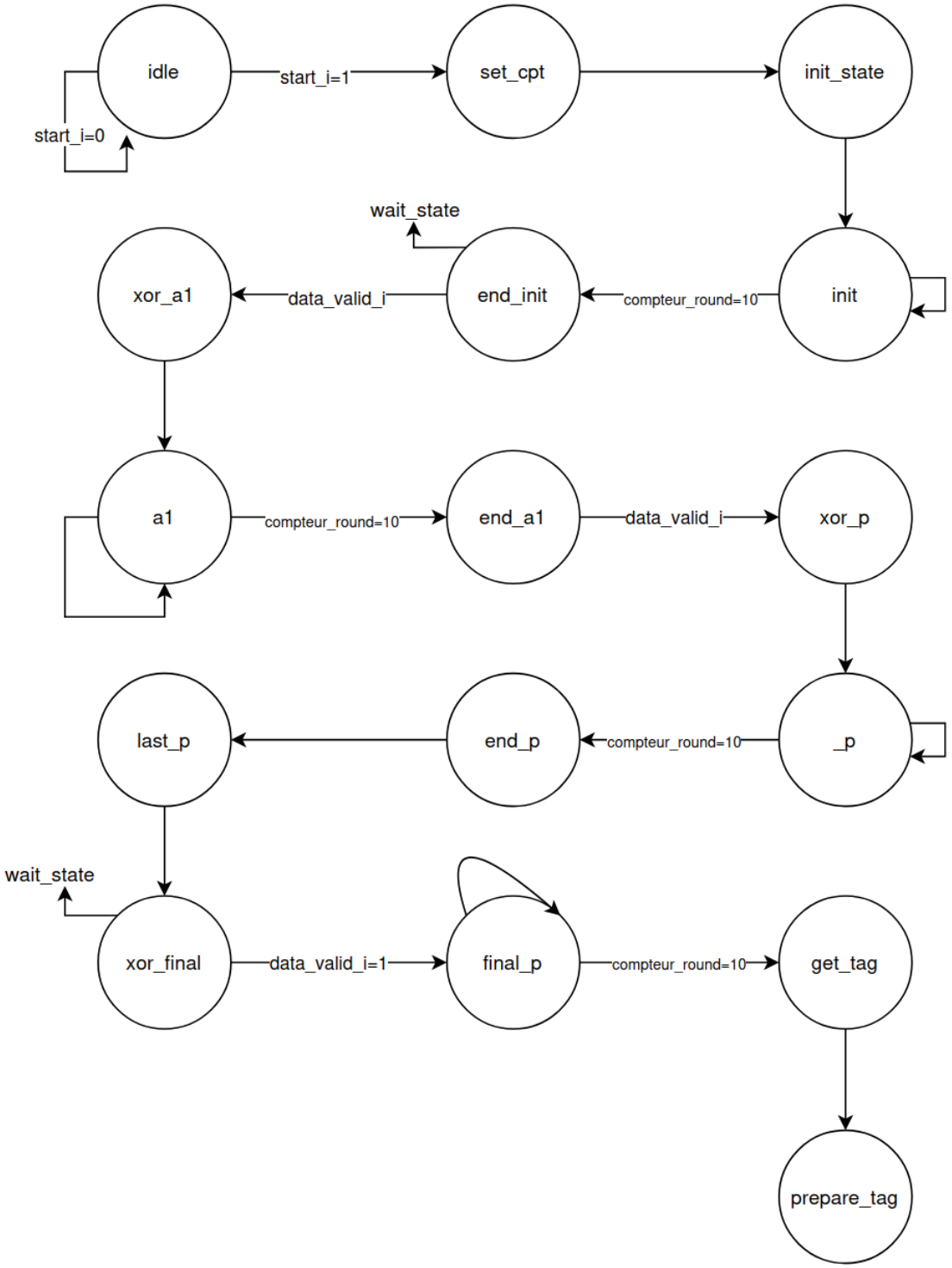


Schéma de l'architecture globale du protocole ASCON128

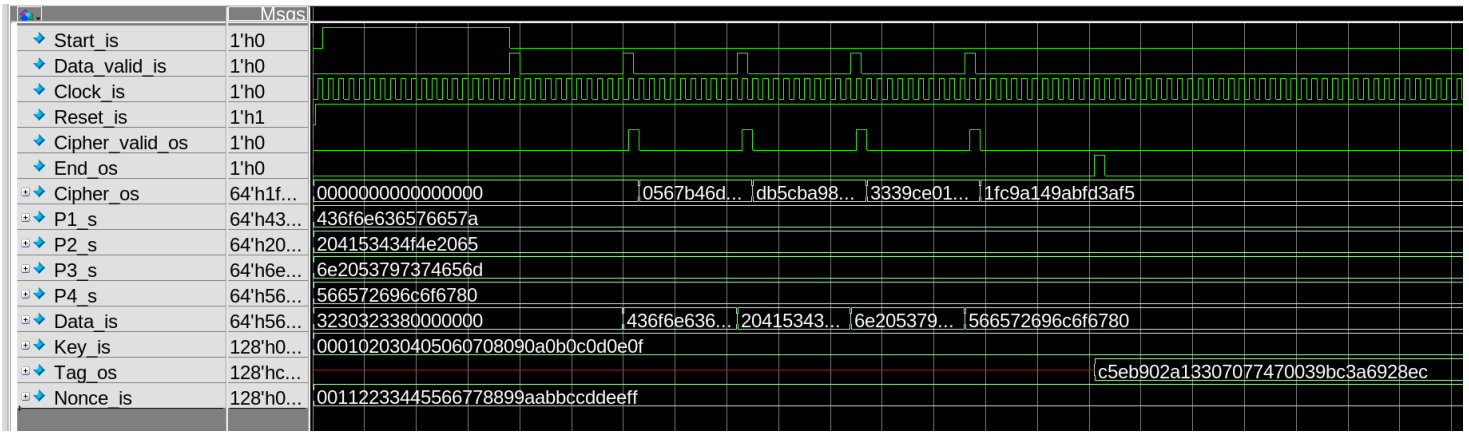
La FSM coordonne les différentes phases (initialisation, donnée associée, chiffrement et finalisation) en fonction du contexte d'utilisation de l'algorithme. Chaque état envoie des instructions au bloc de permutation et xor. Par la suite, l'état peut passer à l'état suivant ou rester en attente, selon le contexte.



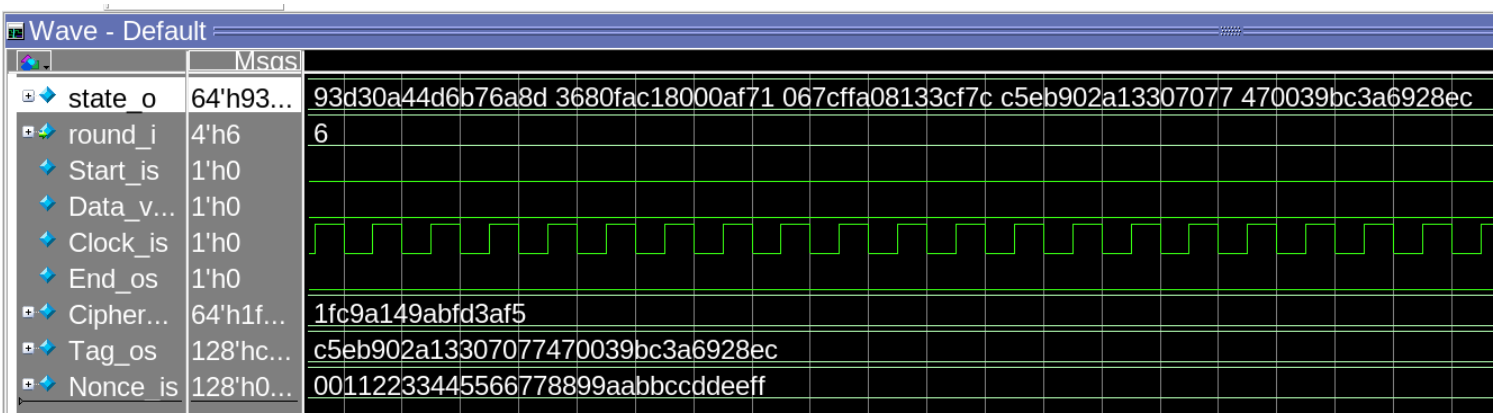
Graphe d'état du protocole ASCON128

c. Module top

Le module top level rassemble tous les modules nécessaires pour la permutation, XOR, et la machine à états finis d'ASCON-128. Son test bench permet de vérifier le bon fonctionnement global de l'algorithme.



Les résultats du test bench du module top_tb révèlent le chronogramme complet de l'algorithme ASCON128 jusqu'à sa conclusion. L'observation du chronogramme indique que le tag est généré à la fin, et que les données chiffrées (cipher) sont publiées progressivement entre les blocs de permutations.



Zoom sur la phase finale du test bench. On retrouve bien la valeur de state_o demandé indiquée dans l'énoncé. L'algorithme est opérationelle et le message est bien chiffré.

6. Conclusion

Ce projet d'implémentation de l'algorithme ASCON128 en SystemVerilog a constitué une opportunité enrichissante pour se familiariser les langages de description du matérielle. En mettant en œuvre l'algorithme ASCON128, nous avons combiné les aspects de confidentialité et d'authenticité des données, offrant ainsi une compréhension approfondie des protocoles de sécurité. Ce projet s'est révélé extrêmement instructif, et j'apprécie particulièrement le fait que nous ayons mis en œuvre, même de manière simplifiée, un protocole de chiffrement qui est toujours d'actualité. Bien que le projet semblait très abstrait au départ, l'implémentation des derniers modules, en particulier le top level, m'a permis de visualiser concrètement la finalité du protocole de chiffrement.