MINES Saint-Étienne

Une école de l'IMT

ÉCOLE DES MINES DE SAINT-ÉTIENNE – ISMIN

CONCEPTION DE SYSTEME NUMERIQUE

Modélisation VHDL de l'algorithme de déchiffrement ASCON

Benjamin VAUCHEL

2 janvier 2023

Table des matières

1. Introduction

L'algorithme ASCON est un algorithme de chiffrement authentifié avec données associées (AEAD), c'est-à-dire qu'il garantit à la fois la confidentialité d'un message et l'intégrité de celui-ci, ainsi que l'authenticité de son expéditeur, grâce à un tag d'authentification. Il repose sur une permutation cryptographique légère, utilisée à différentes étapes du processus, ce qui le rend efficace et facile à mettre en œuvre en logiciel comme en matériel. Il s'agit d'un algorithme de chiffrement symétrique, où une seule clé secrète est partagée entre les parties communicantes. Dans ASCON-128, la clé, le nonce et le tag font chacun 128 bits, ce qui assure un bon niveau de sécurité.

Dans le cadre d'un projet de conception de système numérique, l'implémentation de l'algorithme ASCON-128 en VHDL permet de s'initier à ce langage de description matériel. Cette implémentation utilise les packages *std_logic_1164* et *numeric_std* de la bibliothèque *IEEE*, notamment pour effectuer des opérations logiques et arithmétiques sur des tableaux de bits. Le circuit sera simulé à l'aide du logiciel *ModelSim*.

2. Description générale

2.1. Synoptique de l'algorithme ASCON128

L'algorithme ASCON128 prend en entrée un texte clair à chiffrer de taille 248 bits, une donnée associée sur 32 bits remise sur 64 bits, une clé de taille 128 bits et un nonce, nombre arbitraire, de taille 128 bits. Il renvoie en sortie le texte chiffré sur 248 bits et le tag d'authentification de taille 128 bits.

Lors du chiffrement, l'algorithme opère sur un état courant S de 320 bits et lui applique une permutation, qui consiste à répéter en boucle 6 ou 12 fois une transformation élémentaire, elle-même composée d'opérations élémentaires. Le chiffrement se fait en 4 phases :

1- L'initialisation de l'état courant S à partir d'un vecteur d'initialisation IV, de la clé et du nonce
2- Le traitement des blocs de données associées en mettant à jour l'état courant
3- Le traitement des blocs de texte clair P_i en mettant à jour l'état courant et en renvoyant les blocs de texte chiffré C_i
4- La finalisation du processus en renvoyant le tag pour l'authentification.

2.2. Modélisation des états S

L'état courant S est manipulé tout au long du processus. Celui-ci est divisé en 5 registres x_i de 64 bits tel que :

$$S = x_0 \,\&\, x_1 \,\&\, x_2 \,\&\, x_3 \,\&\, x_4$$

Ainsi, chaque registre subira des transformations avec les permutations p^6 et p^{12}.

La représentation de l'état courant peut se faire en ligne selon les 5 registres de 64 bits, ou bien en colonne (64 colonnes de 5 bits) (figure 1).

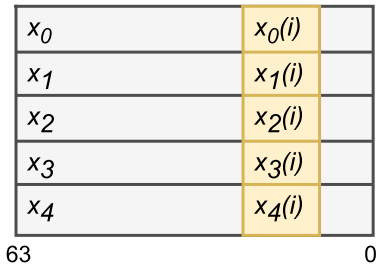


Figure 1 – Représentation double de l'état courant S

3. Transformations élémentaires de ASCON128

On commence par implémenter les transformations élémentaires qui permettent d'appliquer les permutations à l'état S.

3.1. L'addition de constante p_C

3.1.1. Modélisation et implémentation de p_C

Le bloc addition de constante ajoute une constante de ronde au registre 2 de l'état courant S à la ronde indiquée. Le composant prend en entrée **state_i** de type **type_state** et **round_i** de type **bit4** (figure 2).

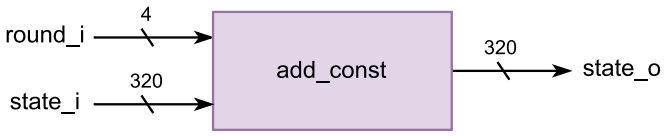


Figure 2 – Entité add_const

L'implémentation de l'addition de constante est faite avec un tableau **round_const** de 12 octets (de type **type_constant**) qui définit les 12 constantes de ronde. Le tableau ainsi que son type sont définis dans **ascon_pack**.

Pour accéder à la constante de la ronde désirée, il faut convertir **round_i** de type **std_logic_vector** en entier. Cela nécessite de passer par le type intermédiaire **unsigned**.

L'addition se fait avec un ou-exclusif (XOR) sur l'octet de poids le plus faible du registre 2 de l'état courant (figure 3).

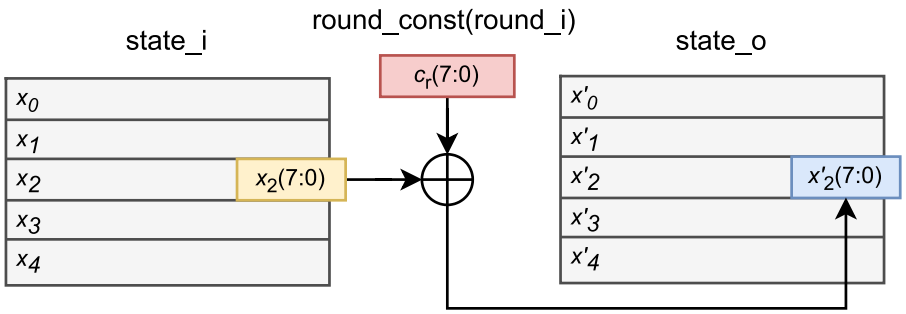


Figure 3 – Fonctionnement de **add_const**

Lors de l'implémentation, on effectue l'opération XOR sur le registre 2 entier plutôt que sur son octet de poids le plus faible. Pour ce faire, on remarque que pour tout **std_logic** a, $a \oplus 0 = a$. Il suffit de faire du **padding** à gauche de la constante c_r, c'est-à-dire qu'on ajoute 56 zéros pour compléter les 8 bits et obtenir 64 bits. L'opérande correspondant à la constante c_r s'écrit alors :

```
x"00000000000000"&round_constant(to_integer(unsigned(round_i)));
```

3.1.2. Simulation de l'addition de constante

Dans chaque testbench, on définit le composant à simuler puis une instance de ce composant qui sera toujours appelée *DUT* (Device Under Test).

On réalise le testbench sur une partie des données utilisées lors du chiffrement final. On vérifie que seul le registre 2 est modifié, et plus particulièrement les deux quartets de poids faible (figure 4).

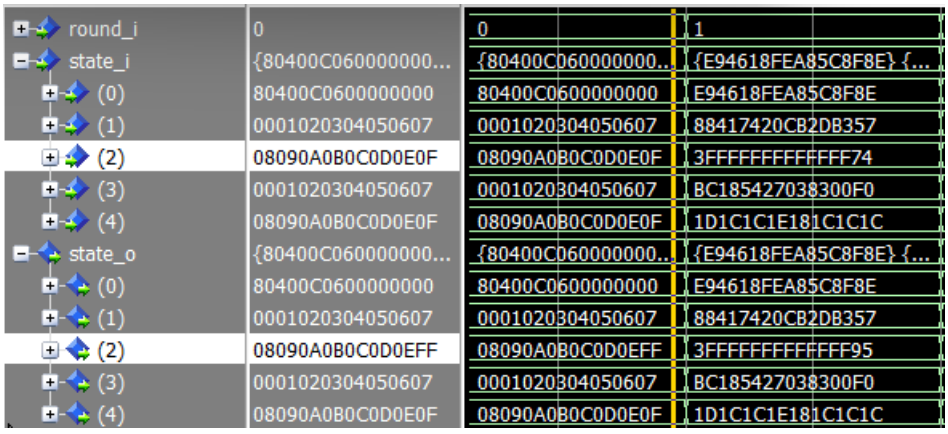


Figure 4 – Simulation de **add_const**

Par exemple, pour la première addition de constante, la ronde 0 est x"F0" et les deux quartets de poids faible du registre 2 de l'état en entrée sont x"0F". On observe bien x"F0" ⊕ x"0F" = x"FF" en sortie.

3.2. La substitution p_s

3.2.1. Substitution d'une donnée de 5 bits : la s-box

3.2.1.1. Modélisation et implémentation de la s-box

On crée d'abord une entité **sbox** qui prend en entrée une donnée de 5 bits (type **bit5**), qui effectue la substitution et qui renvoie en sortie une donnée de 5 bits. On substitue la donnée par les 5 bits correspondant dans la table de substitution S-box présente dans le sujet et rappelée dans la table 1.

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

Table 1 – Table de substitution utilisée

On modélise la table de substitution en VHDL par un tableau de 32 **bit8** initialisé dans *ascon_pack*. En effet, bien que l'on ait besoin uniquement de **bit5**, l'initialisation d'un bit5 par x"00" entraîne une erreur car interprétée comme un **std_logic_vector(7 downto 0)**.

L'entité **sbox** est donnée sur la figure 5.

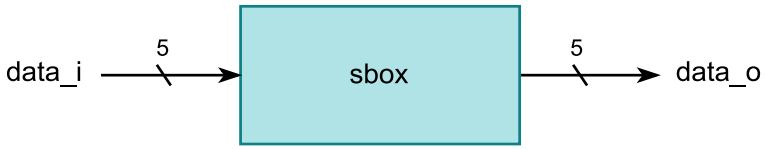


Figure 5 – Entité **sbox**

Encore une fois, pour accéder à l'élément voulu de **table_substitution**, il faut convertir la donnée d'entrée du type **bit5** en **integer**, en passant par le type **unsigned** et la fonction **to_integer** du package *numeric_std*. L'élément du tableau récupéré est du type **bit8** ; pour l'attribuer à la donnée **data_o**, il faut donc sélectionner les 5 bits de poids faible. L'instruction finale dans l'architecture de **sbox** est la suivante.

```
data_o <= table_substitution(to_integer(unsigned(data_i)))(4 downto 0);
```

3.2.1.2. Simulation de la s-box

Le testbench a posé un problème lors de l'attribution d'une valeur au stimulus **data_i_s**, représentant le signal de la donnée d'entrée. En effet, le signal est de type **bit5**, donc un octet est trop large : par exemple, x"11" entraîne une erreur. La solution trouvée consiste à spécifier le nombre de bits :

```
data_i_s <= 5x"08";
```

Cela nécessite de compiler le fichier de testbench avec une version de VHDL ultérieure à 2008 :

```
vcom -2008 -work LIB_BENCH  ./SRC/BENCH/sbox_tb.vhd
```

Ainsi, on teste pour quelques valeurs si la s-box fonctionne : par exemple dans la figure 6, 0x08 devient 0x1B et 0x1D devient 0x0A, comme indiqué dans la table 1.

Figure 6 – Chronogramme de simulation de **sbox**

3.2.2. Substitution complète

3.2.2.1. Modélisation et implémentation

La couche de substitution applique à l'état S une substitution de 5 bits en colonne. En effet, l'état est un tableau de 5 **bit64**, mais il peut être transposé en un tableau de 64 **bit5**. Ainsi, on effectue une substitution à chacune des 64 colonnes. L'entité substitution prend en entrée et renvoie en sortie un **type_state** (figure 7).

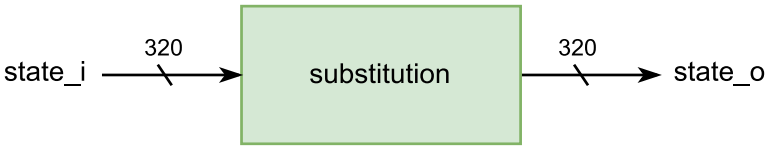


Figure 7 – Entité substitution

Le composant **substitution** instancie donc 64 s-boxes pour effectuer les 64 substitutions. La figure 8 décrit comment une substitution est faite sur la colonne i.

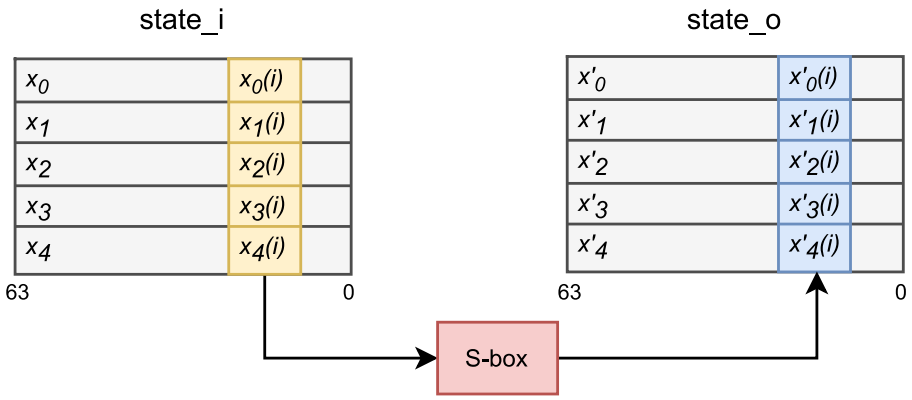


Figure 8 – Fonctionnement de substitution

On crée un type **trans_state** qui correspond à **array (0 to 63) of bit5** et permet de représenter l'état S transposé. On déclare deux signaux intermédiaires de type **trans_state** : **state_trans_in_s** et **state_trans_out_s**. La description de l'architecture de substitution se fait avec 3 **generate** associés à des boucles for afin de répéter des instructions logiques.

1- Un premier generate effectue 64 instructions afin de transposer l'état donné en entrée dans **state_trans_in_s**.
2- Un deuxième instancie les 64 s-boxes en connectant les ports en entrée et en sortie aux colonnes des états transposés avec **port map**.
3- Un dernier transpose à nouveau l'état transposé dans **state_trans_out_s** avec un second generate imbriqué en associant les **std_logic** un par un.

Il est important de remarquer que les valeurs $x_0(i)$ du registre x_0 restent au poids fort, donc il faut les adresser par state_trans$(i)(4-0)$. La numérotation est en fait inversée pour l'ensemble des registres. La figure 9 décrit la numérotation des registres selon la modélisation de l'état courant avec l'exemple du registre 1.

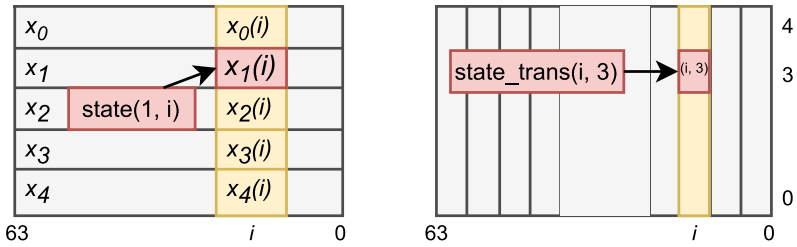


Figure 9 – Représentation de l'état S sous forme de lignes *(gauche)* et sous forme transposée *(droite)*

3.2.2.2. Simulation de la substitution

On teste la substitution avec les valeurs attendues en sortie de l'addition de constante lors de la toute première permutation du test final (figure 10). Ainsi, on obtient les valeurs attendues et l'implémentation de la couche de substitution est validée.

state_i	{80400C06000000...
(0)	80400C0600000000
(1)	0001020304050607
(2)	08090A0B0C0D0EFF
(3)	0001020304050607
(4)	08090A0B0C0D0E0F
state_o	{8849060F0C0D0E...
(0)	8849060F0C0D0EFF
(1)	80410E05040506F7
(2)	FFFFFFFFFFFFFF0F
(3)	80400406000000F0
(4)	0808080A08080808
state_trans_in_s	{0F} {0F} {0F} {05...
state_trans_out_s	{1C} {1C} {1C} {1...

Figure 10 – Simulation de **substitution**

3.3. La diffusion linéaire p_L

3.3.1. Principe et modélisation de la couche de diffusion linéaire

La couche de diffusion linéaire effectue une diffusion sur les 5 registres de l'état *S*. Les opérations sont des sommes représentées par des XOR et des rotations cycliques (vers la droite). L'entité prend en entrée et en sortie les états de type **type_state** (figure 11).

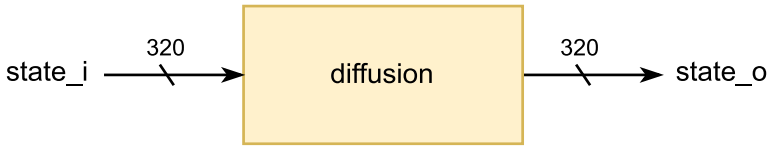


Figure 11 – Entité **diffusion**

Pour implémenter une rotation cyclique, il faut remarquer que :

$$x_i \ggg a = x_i(a-1 \text{ downto } 0) \,\&\, x_i(63 \text{ downto } a)$$

Cette égalité est illustrée dans la figure 12.

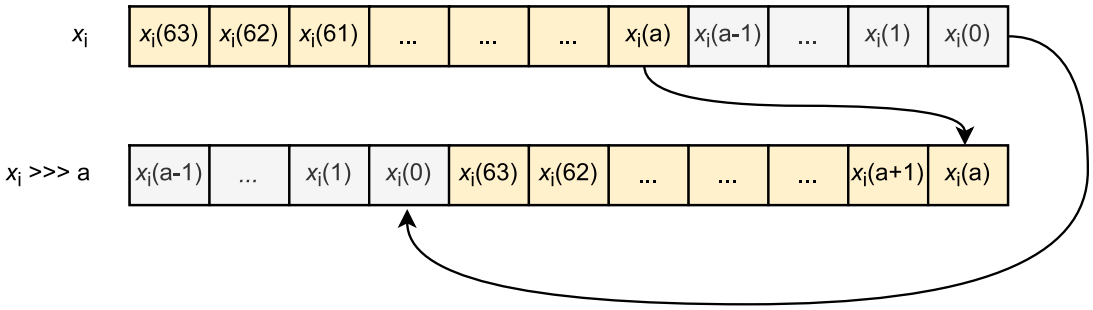


Figure 12 – Illustration d'une rotation cyclique sur un registre

3.3.2. Simulation

Pour le testbench, on attribue à l'état d'entrée les valeurs attendues de l'état de sortie de la substitution de la première permutation du test final. Il suffit de comparer les valeurs attendues en sortie par rapport à celles obtenues sur le chronogramme (figure 13). La couche de diffusion linéaire est alors validée.

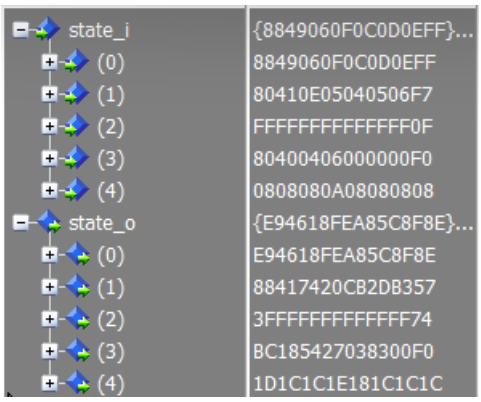


Figure 13 – Simulation de la diffusion linéaire

4. Le bloc permutation

4.1. La permutation de base

La modélisation de la permutation de base est une description structurelle de 5 instances de composants (figure 14) :

- une instance de multiplexeur
- une instance d'addition de constante
- une instance de substitution
- une instance de diffusion linéaire
- une instance de registre de l'état courant sans le signal de write enable **en_i** dans un premier temps.

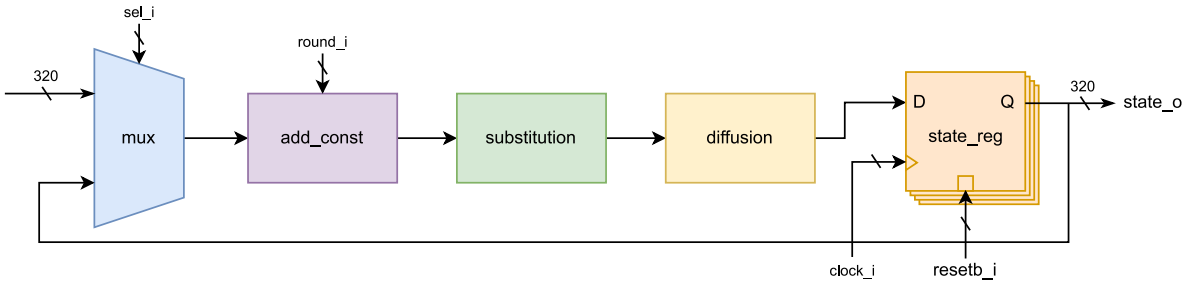


Figure 14 – Structure de la permutation de base **permu_base**

4.1.1. Multiplexeur

Le multiplexeur sert à sélectionner l'état sur lequel on réalise la permutation : ou bien l'état d'entrée au début du chiffrement, ou bien l'état de sortie de la permutation pour faire des permutations itérativement (p^6 ou p^{12}). L'entité du multiplexeur est donnée sur la figure 15.

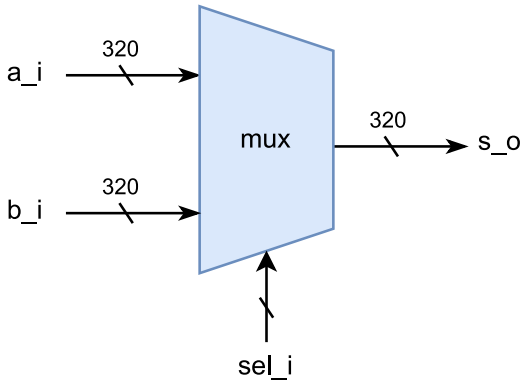


Figure 15 – Diagramme de l'entité du multiplexeur

L'architecture du multiplexeur mise en œuvre prévoit de sélectionner **a_i** quand **sel_i** vaut **1** et **b_i** sinon. C'est un choix arbitraire qu'il ne faudra pas inverser lors de l'utilisation du multiplexeur.

Le code du multiplexeur étant trivial, le testbench a été très rapide à écrire : on observe la sortie lorsque le bit de sélection vaut 1 puis 0, et le composant est validé (figure 16).

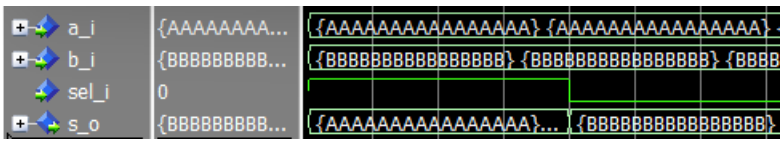


Figure 16 – Chronogramme Simulation du multiplexeur

4.1.2. Registre de l'état courant State

4.1.2.1. Modélisation et implémentation

Afin de mémoriser l'état courant pour chaque ronde, on ajoute un registre après l'instance de la couche de diffusion, avec une horloge et un reset en entrées de la permutation. On ajoute aussi un signal d'entrée **en_i** qui décide ou non de mémoriser (d'écrire) la valeur d'entrée dans le registre. Le registre est synchrone, c'est-à-dire que l'état d'entrée est mémorisé dans le registre lors d'un coup d'horloge sur front montant. En revanche, le reset est asynchrone : les valeurs dans le registre sont mises à 0 aussitôt que **resetb_i** est mis à 0.

L'entité du registre d'état est donnée sur la figure 17.

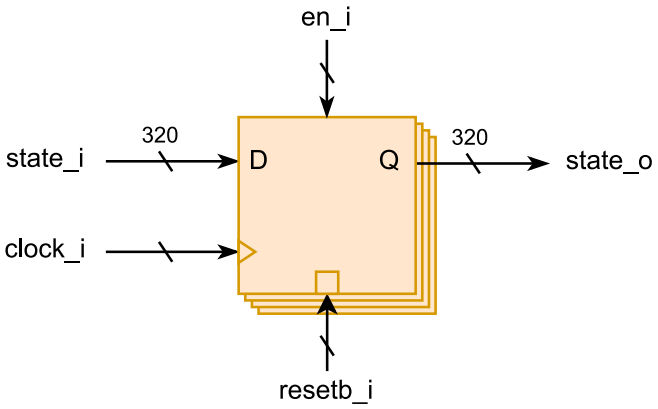


Figure 17 – Entité **state_register_w_en**

Pour l'implémentation en VHDL, on utilise un signal intermédiaire **state_s** auquel on affecte les valeurs dans un processus. Le processus est nécessaire car il permet d'utiliser la structure **if then else** et permet surtout d'exécuter nos instructions sur un changement de signal, en l'occurrence les signaux **clock_i** et **resetb_i**. Ces derniers sont dans une **liste de sensibilité** : on entre dans le **process** uniquement lorsqu'un des deux signaux change.

La syntaxe de remise à 0 du registre utilisée fait appel à **others** :

```
state_s <= (others => (others => '0'));
```

Cette instruction permet de mettre à 0 tous les éléments d'un tableau de dimension 2 : en effet, on rappelle que **state_s** est de type **array (0 to 4) of std_logic_vector(63 downto 0)**.

Pour détecter un front montant d'horloge, il y a deux conditions à vérifier :

— Le signal d'entrée **clock_i** est modifié, ce qui s'écrit **clock_i'event**
— Le signal d'entrée **clock_i** vaut 1 (passage à 1 car front montant), ce qui s'écrit **clock_i = '1'**.

4.1.2.2. Simulation du registre de l'état courant

Le testbench permet de vérifier que le registre mémorise la donnée d'entrée uniquement lorsque le bit enable vaut 1 et sur front montant d'horloge (figure 18, 30 ns). De plus, on vérifie que la remise à zéro est bien asynchrone (45 ns).

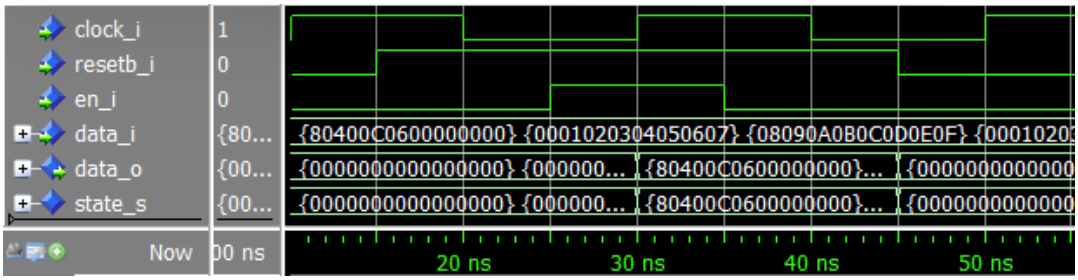


Figure 18 – Simulation de **state_register_w_en**

Pour simuler l'horloge, on l'initialise à 0 dès qu'on déclare le signal de clock avec **:= '0'** puis on alterne sa valeur avec **not**, et la demi-période est définie avec **after**. En définitive, l'instruction qui alterne l'horloge s'écrit :

```
clock_s <= not clock_s after 10 ns;
```

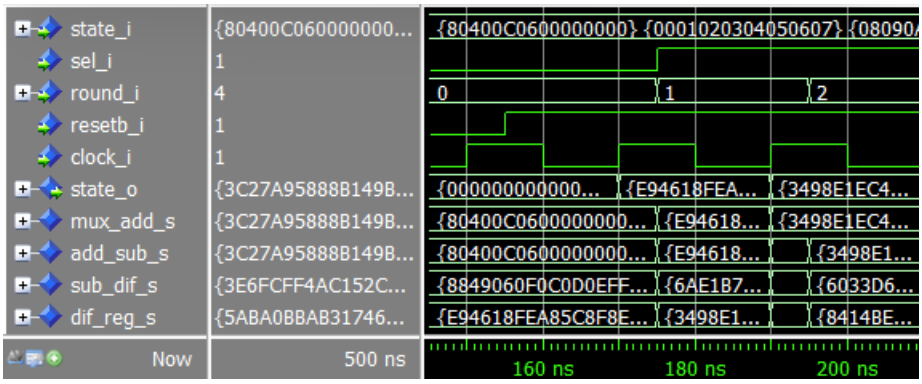
4.1.3. Implémentation et simulation de la permutation de base

Pour l'implémentation en VHDL de la permutation de base, comme on l'a déjà fait avec la couche de substitution, on déclare les composants un par un dans l'architecture avant le **begin**, puis on instancie les composants en reliant les ports d'entrée et de sortie et les signaux intermédiaires avec **port map**. Les signaux intermédiaires permettent en effet de relier la sortie d'un composant à l'entrée d'un autre. Par exemple, **sub_dif_s** est le signal de l'état qui se situe à la sortie du bloc substitution et en entrée du bloc diffusion linéaire. Aussi, dans la configuration il est nécessaire d'importer chacune des architectures des composants instanciés. Cela se fait avec **use entity** en précisant la libraire (en l'occurrence *LIB_RTL* ou *LIB_BENCH*), le nom de l'entité et le nom de l'architecture. Par exemple, pour importer **add_const**, on écrit :

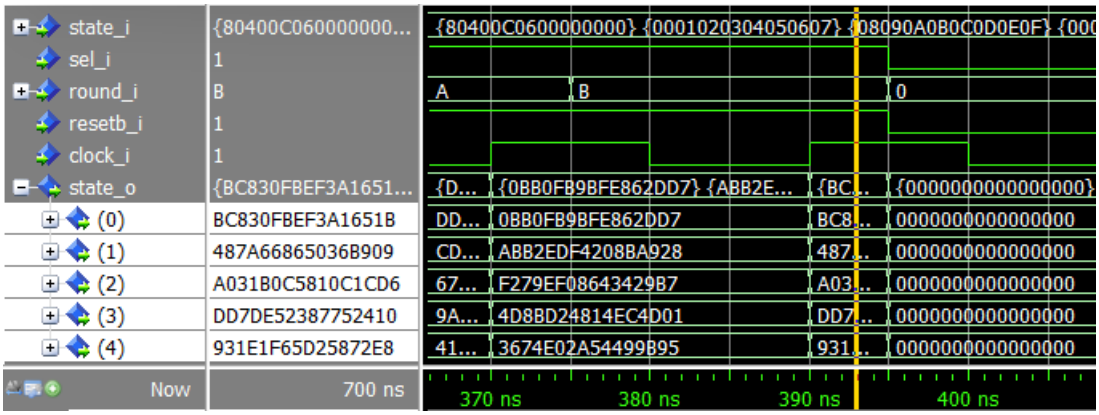
```
for all: add_const
      use entity LIB_RTL.add_const(add_const_arch);
end for;
```

Simulation.

- Dans le testbench de la permutation de base, on commence par sélectionner en entrée du multiplexeur la donnée de départ « valeur initiale ». On lui applique la première permutation de ronde 0 en mettant la valeur de **resetb_i** à 1, ce qui permet de mémoriser la valeur au prochain front montant de l'horloge. Puis on met **sel_i** à 1 pour sélectionner en entrée du multiplexeur l'état de sortie du registre. On augmente le numéro de la ronde entre chaque coup d'horloge (front montant) pour effectuer successivement les 11 permutations restantes (figure 19a).
- La fin de la simulation (figure 19b) permet de vérifier la valeur de l'état **state_o** en sortie de la p^{12} et on obtient bien les valeurs attendues.



(a) Début de la simulation : multiplexeur et première permutation



(b) Fin de la simulation : reset du registre d'état et résultat de la première p^{12}

Figure 19 – Simulation de la permutation de base

Pour faire évoluer les valeurs des signaux, nous avons utilisé un processus de génération de stimuli qui permet d'exécuter les instructions de manière séquentielle. Des instructions d'attente (**wait for**) ponctuent les mises à jour des valeurs des signaux, notamment pour incrémenter la ronde.

4.2. La permutation intermédiaire

La permutation intermédiaire reprend les instances de composant de la permutation de base avec deux nouvelles instances :

- Une instance de **xor_begin** avec deux nouveaux ports d'entrée : **en_xor_key_b_i** et **en_xor_data_b_i**
- Une instance de **xor_end** avec deux nouveaux ports d'entrée : **en_xor_key_e_i** et **en_xor_lsb_e_i**.

La description structurelle de la permutation intermédiaire est donnée en figure 20.

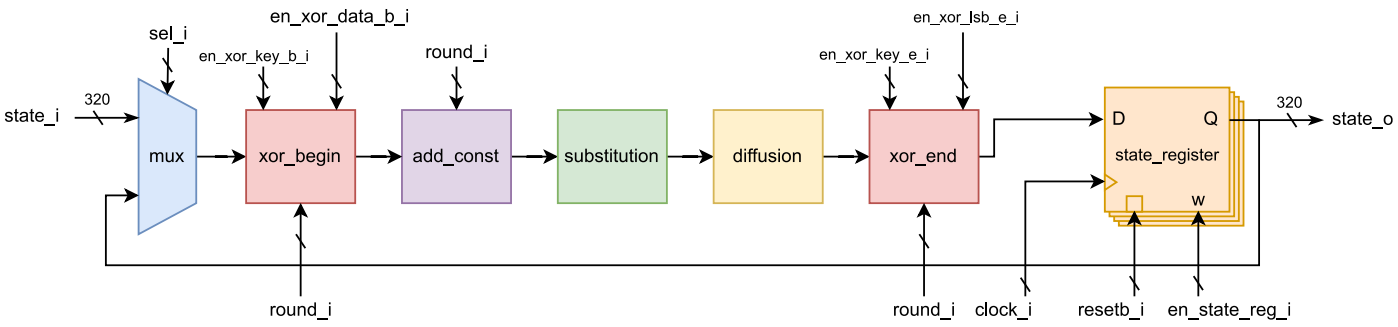


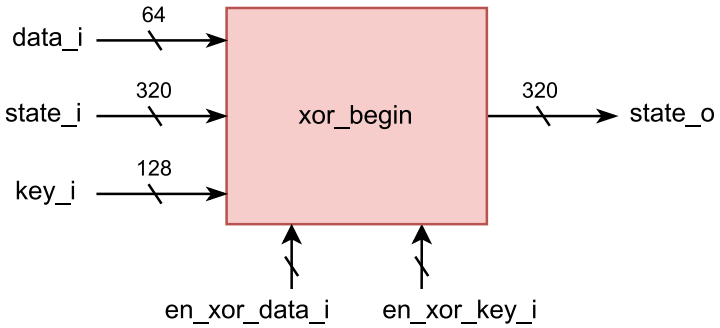
Figure 20 – Diagramme d'architecture de **permu_inter**

La permutation intermédiaire intègre en fait des XOR en début et en fin de permutation pour y mettre la clé, la donnée associée, un bit de poids faible ainsi que le texte clair à chiffrer.

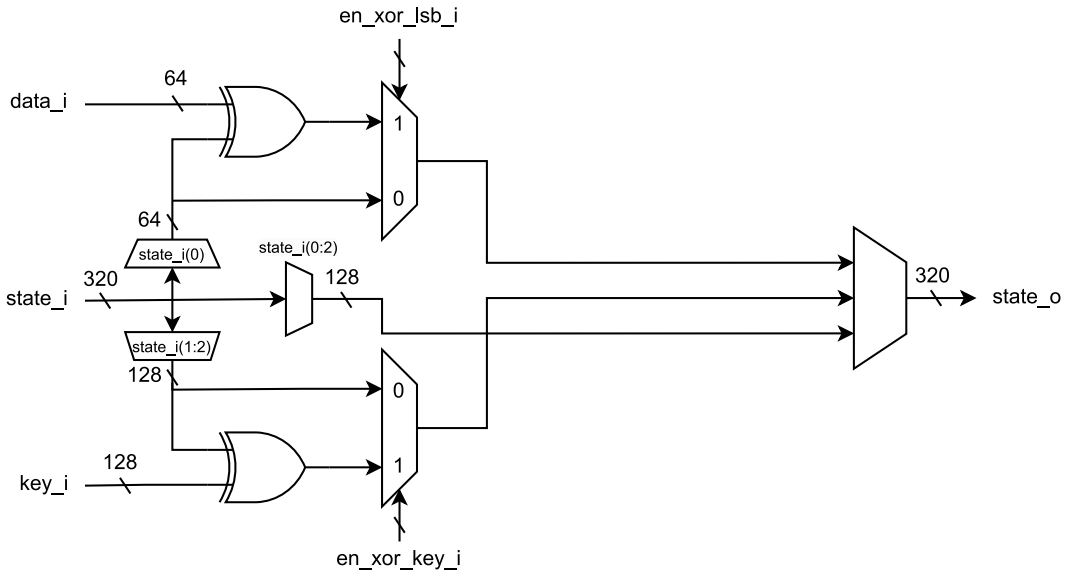
4.2.1. Les opérateurs XOR

4.2.1.1. L'opérateur XOR de début de permutation

L'opérateur XOR de début de permutation peut traiter une donnée de 64 bits ou la clé de 128 bits. Il effectue un ou deux XOR sur l'état en entrée et la donnée ou la clé. L'entité **xor_begin** est donnée en figure 21a, et la description du circuit interne est donnée en figure 21b.



(a) Entité **xor_begin**



(b) Circuit logique interne de **xor_begin**

Figure 21 – Entité et circuit du XOR de début de permutation

Lors de l'implémentation en VHDL, on utilise la syntaxe **when else** pour opérer le XOR si le bit enable vaut 1.

- Le XOR avec la donnée **data_i** se fait sur le registre 0 de l'état courant.
- Pour le XOR avec la clé **key_i**, on réalise un XOR sur chacun des deux registres 1 et 2 avec respectivement les 64 bits de poids forts et les 64 bits de poids faibles de la clé.

Les registres 3 et 4 de l'état en sortie sont égaux aux registres 3 et 4 de l'état en entrée, comme indiqué dans la figure 21b.

Simulation.

Pour tester le composant, on initialise la clé, la donnée et les registres de l'état d'entrée à leur valeur maximale (tous les bits à 1). Pour ce faire, on utilise à nouveau la syntaxe **OTHERS**. La simulation est donnée dans la figure 22.

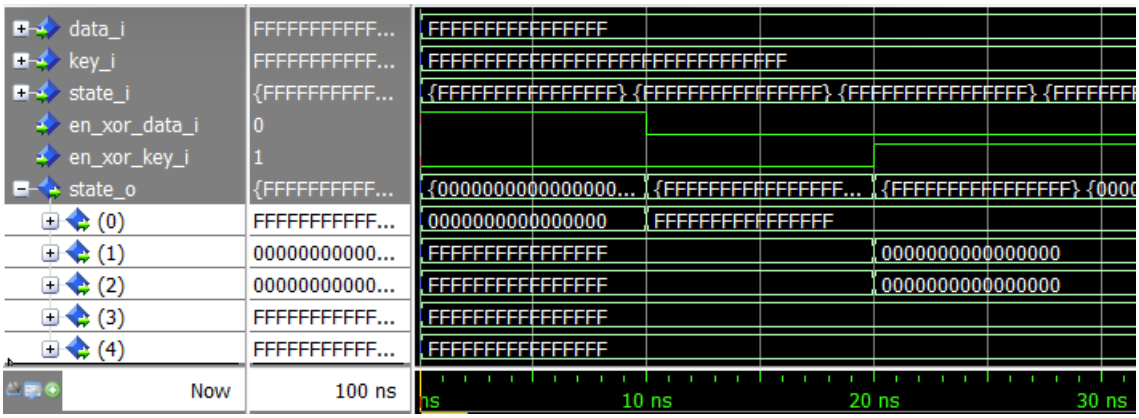
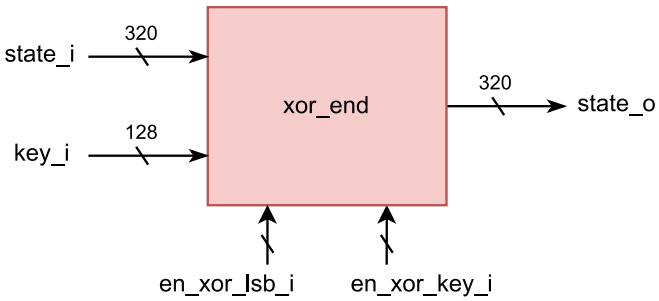


Figure 22 – Entité et circuit du XOR de début de permutation

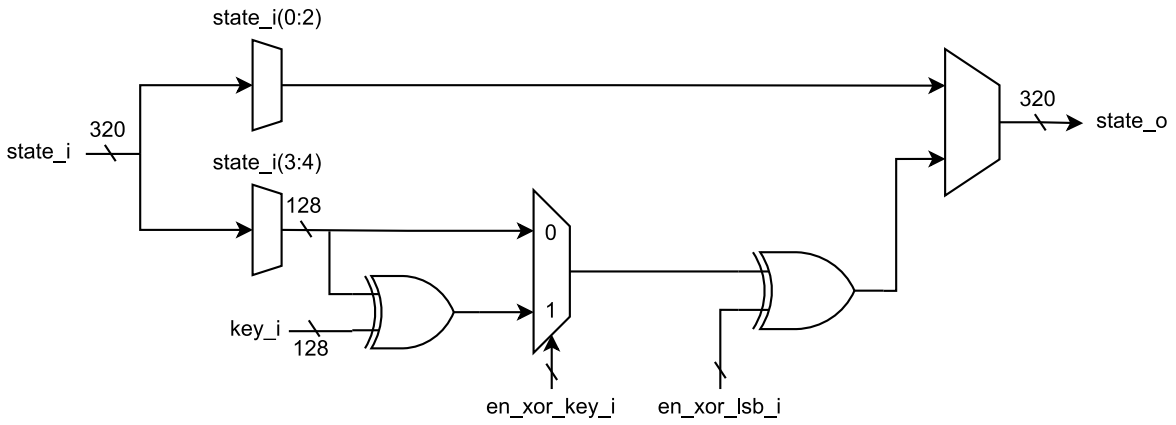
— Jusqu'à 10 ns de simulation, **en_xor_data_i** est à 1, et le XOR entre le registre 0 et la donnée donne bien 0x0000000000000000.
— Entre 10 ns et 20 ns, aucun XOR ne doit être appliqué, et l'état de sortie est égal à l'état d'entrée.
— Après 20 ns, **en_xor_key_i** est à 1, et l'opération XOR avec la clé donne bien 0x000000000000000 aux registres 1 et 2.

4.2.1.2. L'opérateur XOR de fin de permutation

L'opérateur XOR de fin de permutation peut effectuer un XOR avec la clé ou avec un 1 en bit de poids le plus faible. Dans le cadre du chiffrement ASCON128, on n'effectue pas les deux XOR simultanément, mais l'implémentation le prévoit quand-même. L'entité **xor_end** et la description interne du circuit sont données en figure 22.



(a) Entité **xor_end**



(b) Circuit logique interne de **xor_end**

Figure 23 – Entité et circuit du XOR de fin de permutation

Pour l'implémentation en VHDL, les valeurs dans les registres 0 à 2 de l'état en sortie sont les mêmes que ceux de l'état en entrée. Si **en_xor_key_i** est à 1, on effectue le XOR sur le registre 3 avec les 64 bits de poids fort de la clé.

Pour le registre 4, on crée d'abord un signal intermédiaire **x4_s** qui prend la valeur du registre 4 en entrée. On applique d'abord le XOR avec **en_xor_lsb_i** : en effet, lorsque ce dernier vaut 0, la sortie vaut l'entrée, et lorsqu'il vaut 1, la sortie vaut bien **state_i** $\oplus 1$. Enfin, on attribue au registre 4 de sortie :

- la valeur de **x4_s** si **en_xor_key_i** est à 0,
- le résultat de l'opération XOR entre le signal **x4_s** et les 64 bits de poids faible de la clé si **en_xor_key_i** est à 1.

Simulation.

De la même manière qu'avec l'opérateur XOR en début de permutation, on fait le choix d'initialiser la clé et les registres de l'état d'entrée à leur valeur maximale. Le chronogramme est donné dans la figure 24.

— Jusqu'à 10 ns de simulation, **en_xor_lsb_i** est à 1, donc le registre 4 de l'état de sortie doit être le résultat d'un XOR avec un 1 en bit de poids le plus faible. L'opération faite est exactement la suivante :

$$\text{0xFFFFFFFFFFFFFFFF} \oplus \underbrace{0 \dots 01}_{\text{64 bits}} = 1 \dots 1 \oplus 0 \dots 01 = 1 \dots 10 = \text{0xFFFFFFFFFFFFFFFE}$$

— Entre 10 ns et 20 ns de simulation, aucune opération XOR n'est appliquée car les bits enable sont à 0.

— Après 20 ns, **en_xor_key_i** est à 1, et les opérations XOR sont appliquées entre la clé et les deux registres 3 et 4 :

$$0xFFFFFFFFFFFFFFFF \oplus 0xFFFFFFFFFFFFFFFF = 0x0000000000000000$$

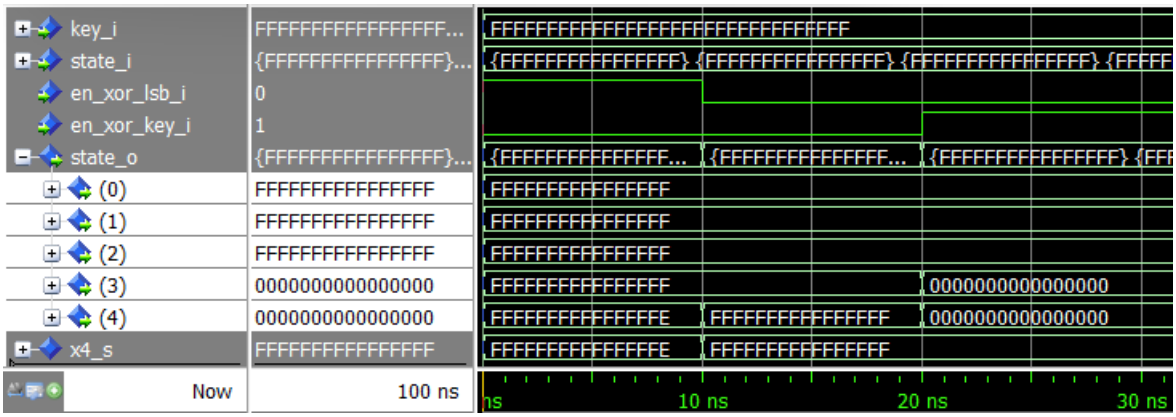


Figure 24 – Simulation de **xor_end**

4.2.2. Modélisation et implémentation de la permutation intermédiaire

L'implémentation de l'entité de la permutation intermédiaire et de son architecture reprend celle de la permutation de base avec les deux nouveaux composants. Cela n'a donc pas posé de difficulté.

Dans le testbench, on teste la première permutation p^{12} avec l'opération de XOR de fin. Ainsi, on reprend le testbench de la permutation de base et on ajoute l'initialisation des bits enable des XOR à 0 et la mise à 1 de **en_xor_key_e_s** à la fin de la p^{12}. La figure 25 se concentre sur la fin du chronogramme, lorsque la dernière permutation est réalisée et que l'opération XOR est appliquée avec la clé. On remarque que les valeurs des registres 3 et 4 de l'état de sortie sont différentes de celles observées dans la simulation de **permu_base**, car le XOR est bien effectué :

DD7DE52387752410 931E1F65D25872E8 $\oplus$ 0001020304050607 08090A0B0C0D0E0F
= DD7CE72083702217 9B17156EDE557CE7

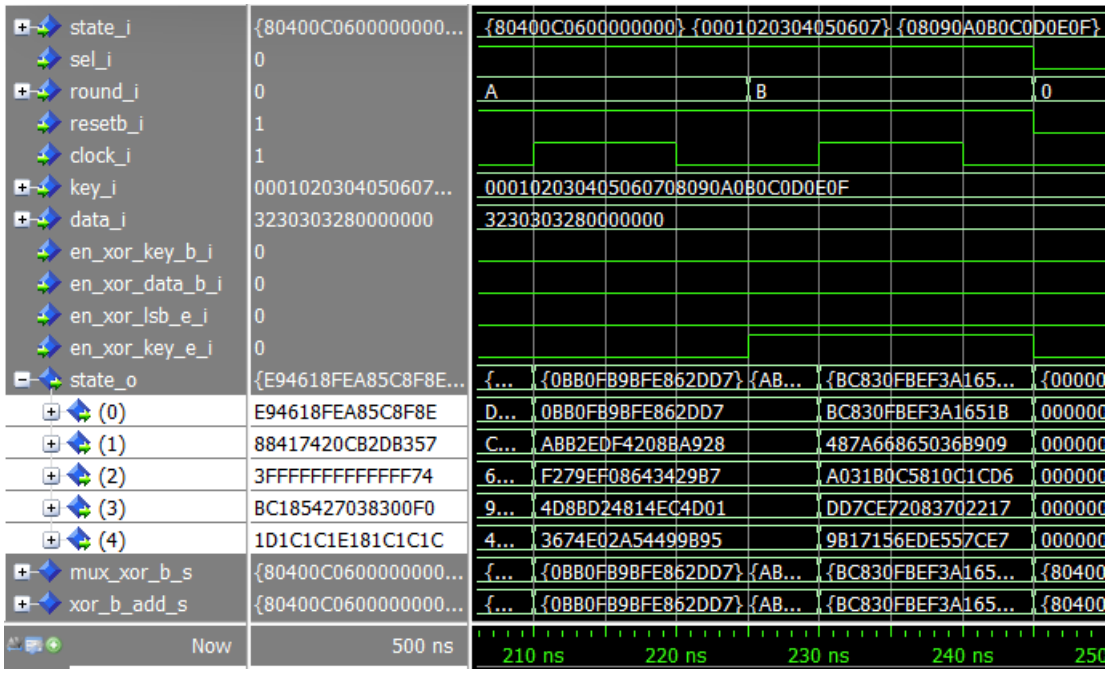


Figure 25 – Chronogramme de la simulation de **permu_inter**

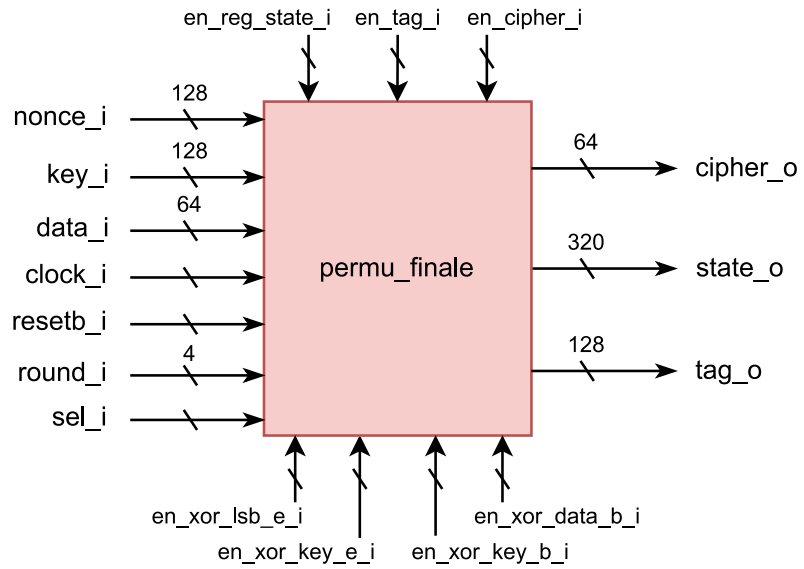
4.3. La permutation finale

Avant chaque permutation p^6, on doit effectuer une opération XOR avec 64 bits du texte clair et mettre en sortie le texte chiffré du bloc **cipher_o** (64 bits). Ce texte chiffré est mémorisé dans un registre jusqu'au prochain calcul des 64 bits chiffrés et écriture dans ce registre. Ce registre est une instance du registre générique **register_w_en** avec 64 bits, présenté en 4.3.1.

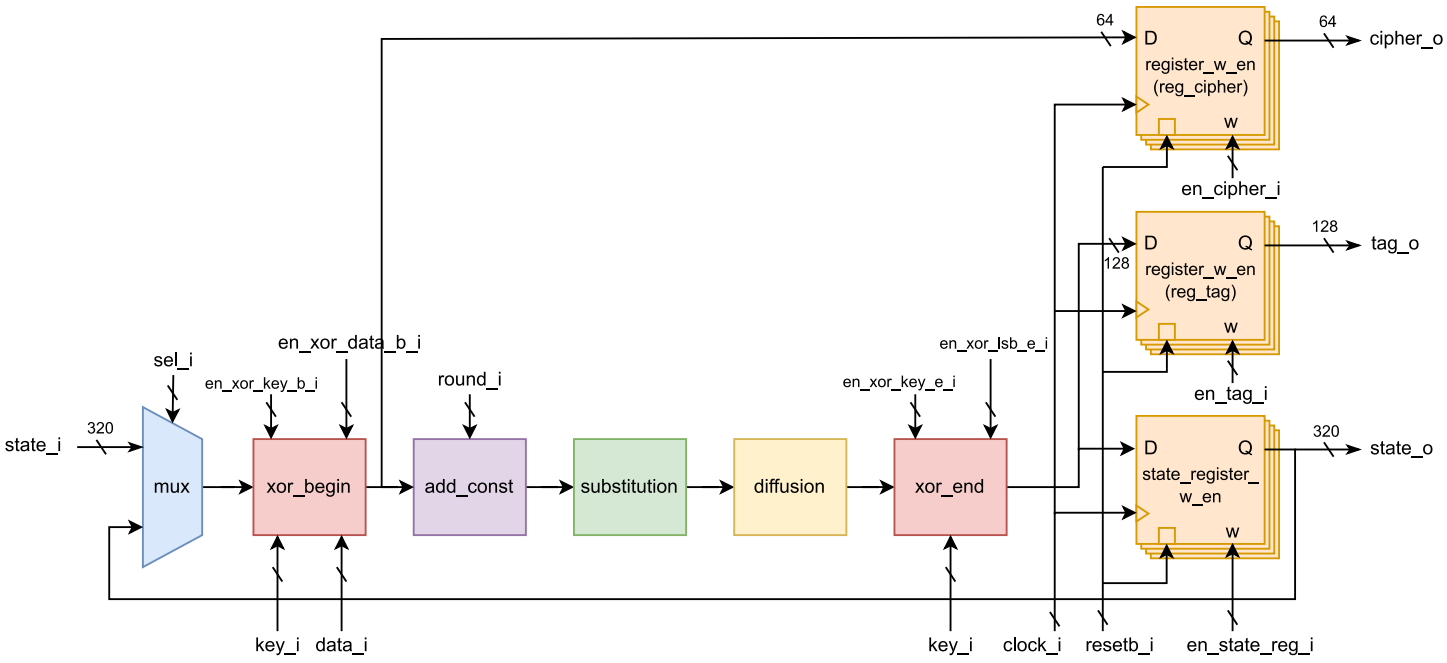
De plus, à la fin du chiffrement ASCON128, après la dernière opération XOR avec la clé, on renvoie le tag en sortie **tag_o** (128 bits). Ce tag est aussi mémorisé dans une instance de registre générique de taille 128 bits.

Enfin, contrairement aux entités **permu_base** et **permu_inter**, on utilise les instances des registres avec le bit d'écriture (*write enable*) qui autorise l'écriture dans le registre lorsqu'il vaut 1.

La figure 26 présente l'entité de la permutation finale ainsi que sa structure. On remarque qu'il faut sélectionner les 64 bits de poids forts (registre 0 de l'état courant S) en entrée du registre du texte chiffré et les 128 bits de poids faibles (registres 3 et 4 de S) en entrée du registre du tag. Pour ce dernier, il faut concaténer les registres 3 et 4 du signal intermédiaire entre le XOR de fin de permutation et le registre de tag.



(a) Entité **permu_finale**



(b) Structure de **permu_finale**

Figure 26 – Entité et structure de la permutation finale

4.3.1. Registre générique

Le registre générique est un composant très similaire au registre de l'état courant. Il permet de mémoriser une donnée de taille indiquée grâce à une variable générique **nb_bits_g** de type **natural** définie dans une partie de la déclaration de l'entité appelée **generic** puis utilisée dans la partie **port**. **nb_bits_g** définit le type des données d'entrée et de sortie **data_i** et **data_o** et le signal intermédiaire **state_s** : **std_logic_vector(nb_bits_g-1 downto 0)**. L'entité du registre générique est donnée sur la figure 27.

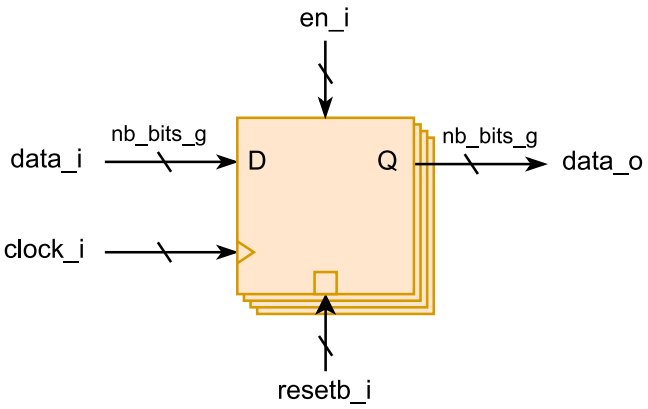


Figure 27 – Entité **register_w_en**

Pour le testbench, l'entité **register_w_en_tb** n'est pas complètement vide car elle doit contenir la partie **generic** avec **nb_bits_g**. On déclare aussi le composant **register_w_en** avec la partie **generic** et le signal d'entrée **data_in_s** avec le type dépendant de **nb_bits_g**. La définition du composant Device Under Test (**DUT**) doit comporter un **generic map** qui définit la valeur de **nb_bits_g**, par exemple 32. Enfin, il ne faut pas oublier d'ajouter aussi un **generic map** dans la configuration du testbench pour le composant DUT :

```
for DUT : register_w_en
      use entity lib_rtl.register_w_en(register_w_en_arch)
            generic map (
                  nb_bits_g => 32);
end for;
```

Comme pour le registre de l'état courant, on teste la réinitialisation du registre asynchrone (45 ns) et la mémorisation synchrone (30 ns) (figure 28).

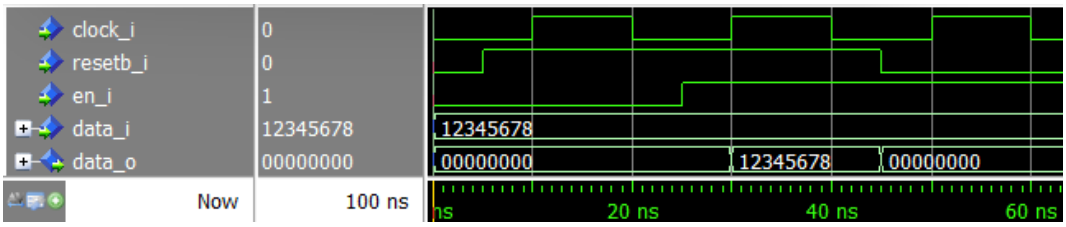


Figure 28 – Simulation de **register_w_en**

4.3.2. Simulation de la permutation finale

Pour tester le composant de la permutation finale, on reprend le testbench de la permutation intermédiaire, mais on teste la phase de finalisation avec le XOR en début et le XOR en fin de la p^{12}. Cela permet de vérifier le bon fonctionnement du registre avec le bit de write enable. La simulation comporte 8 signaux de contrôle qui seront pilotés par la finite-state machine (FSM) à l'avenir.

On initialise donc l'état aux valeurs attendues du texte clair P, on effectue une permutation avec le XOR de début avec la clé, puis 10 permutations simples, et une permutation avec le XOR de fin avec la clé (figure 29, 225 ns). De plus, on écrit dans le registre destiné au tag à la fin en mettant **en_tag_i** à 1. Après avoir vérifié que les valeurs de l'état de sortie et du tag sont les mêmes que celles attendues, on valide le composant.

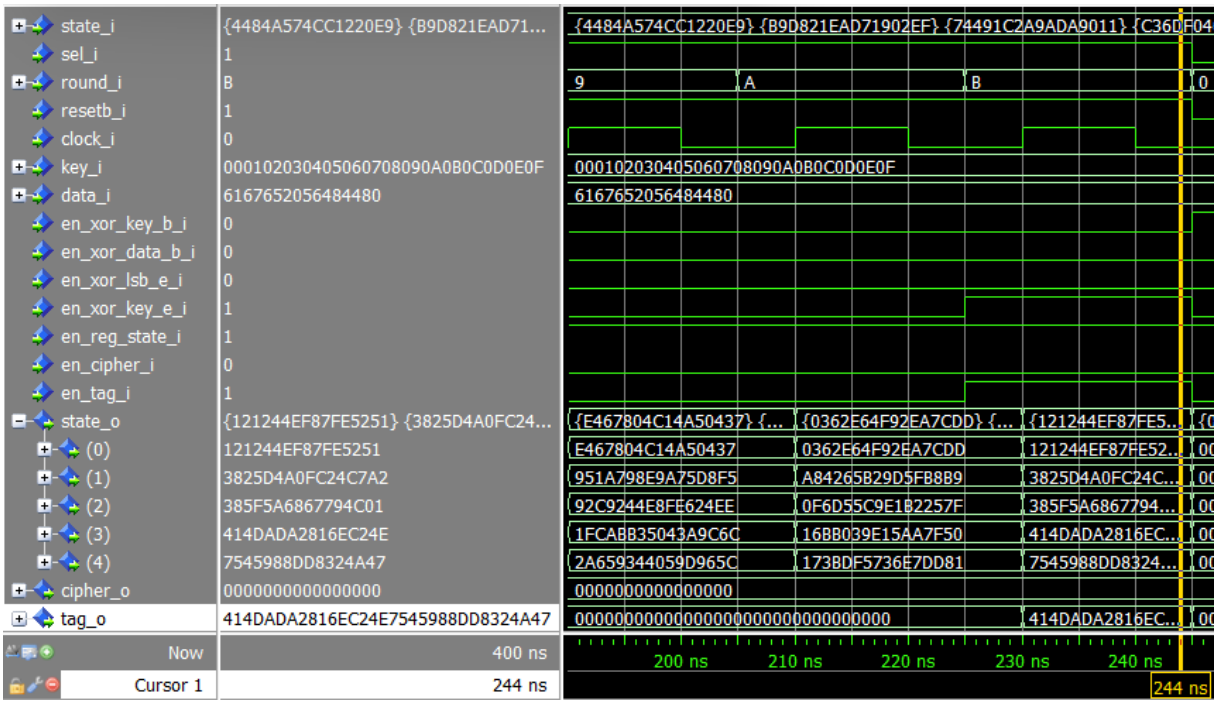


Figure 29 – Chronogramme de **permu_finale**

5. Modélisation des composants top level de ASCON128

5.1. Compteurs de rondes et de blocs

On ajoute des compteurs afin d'automatiser l'incrémentation des rondes et des blocs de chiffrement. Les compteurs sont synchrones et peuvent être réinitialisés de manière asynchrone comme les registres.

5.1.1. Compteur de rondes

Le compteur de rondes est un composant qui permet d'incrémenter les rondes automatiquement. Celui-ci a deux entrées selon le nombre de rondes que l'on veut

effectuer (p^6 ou p^{12}). La sortie **cpt_o** est sur 4 bits car la ronde ne peut excéder la valeur 0xB.

L'entité du compteur de rondes à double initialisation est donnée dans la figure 30.

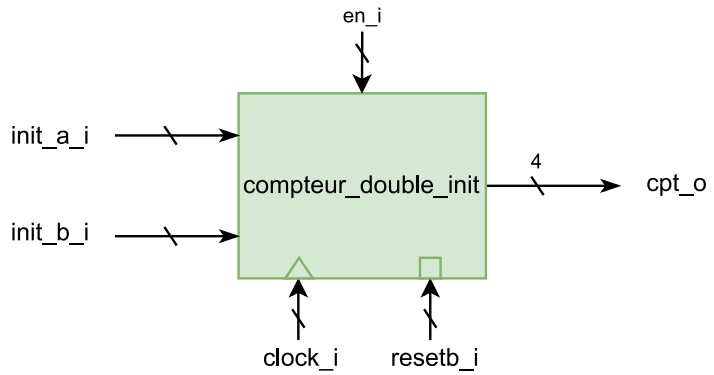


Figure 30 – Entité du compteur de rondes à double initialisation

On peut résumer l'implémentation du compteur par un **processus séquentiel** :

- Si le compteur est réinitialisé avec **resetb_i** = 0, alors le compteur est mis à 0
- Sinon
  - Si le compteur est activé avec **en_i** = 1
    - Si **init_a_i** = 1, alors le compteur est initialisé à 0 (p^{12})
    - Si **init_b_i** = 1, alors le compteur est initialisé à 6 (p^6)
    - Sinon, le compteur est incrémenté
  - Sinon, le compteur ne change pas.

Pour l'implémentation en VHDL, on crée un signal intermédiaire de type **integer** dans la plage $[\![0, 15]\!]$. On utilise ce signal pour l'incrémentation du compteur qui se voit facilitée. En dehors du processus, on affecte alors la valeur du signal à la sortie du compteur. Cela nécessite une conversion de **integer** à **std_logic_vector** en utilisant le type intermédiaire **unsigned**.

Dans la simulation (figure 31), on teste tous les cas possibles décrits par le processus séquentiel. On ne tente pas les cas limites qui ne sont pas gérés, notamment lorsque **init_a_i** et **init_b_i** sont tous les deux à 1, et lorsque la sortie du compteur dépasse 0xF, la valeur maximale possible définie par le **range**.

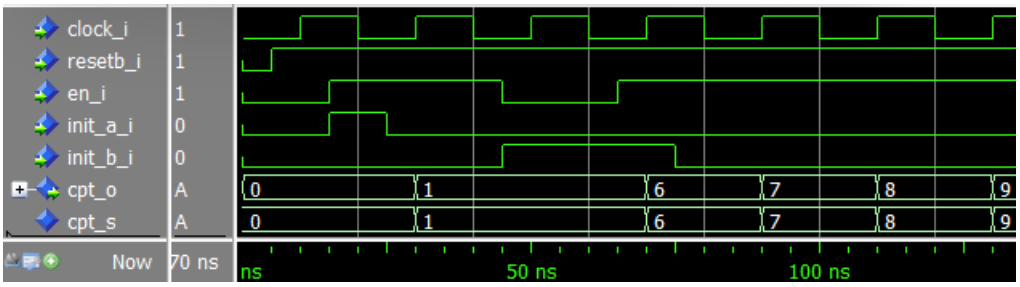


Figure 31 – Chronogramme de la simulation du compteur de rondes

5.1.2. Compteur de blocs

Le compteur de blocs permet de gérer et d'incrémenter le numéro du bloc de chiffrement en cours. En effet, la partie de chiffrement du texte clair se fait avec 3 permutations p^6. Pour avoir un code plus lisible et automatique, il convient de ne pas répéter trois fois les mêmes instructions dans la machine d'états mais d'utiliser un compteur de blocs.

Le compteur de blocs est initialisé à 0 par le port d'entrée **init_i** et renvoie une valeur **cpt_o** comprise entre 0 et 3.

L'entité du compteur de blocs est donnée dans la figure 32.

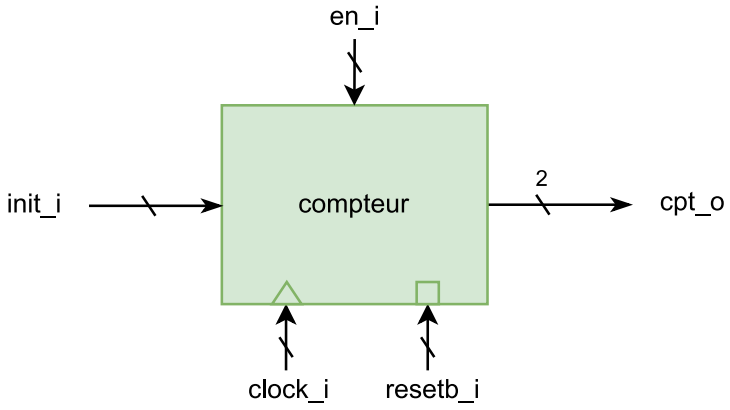


Figure 32 – Entité du compteur de blocs

La simulation étant très similaire à celle du compteur de rondes, aucune remarque supplémentaire n'est nécessaire.

5.2. La machine à états finis (FSM)

Afin de piloter l'ensemble du circuit, on utilise une machine à états finis (en anglais *finite-state machine* ou *FSM*) qui contrôle les compteurs et active et désactive les signaux de contrôle.

La FSM est modélisée par une machine de Moore, c'est-à-dire que les sorties dépendent de l'état présent et leur évolution est synchrone, et l'état futur est calculé à partir des

entrées et de l'état présent. Les conditions se situent donc sur les transitions et non sur les états.

5.2.1. Implémentation partielle de la FSM

La FSM est implémentée en VHDL avec trois processus.

On déclare d'abord dans l'architecture un type énuméré **state_t** qui contient l'ensemble des états qui constituent le graphe d'états. De plus, on déclare deux signaux de type **state_t** : **etat_present** et **etat_futur**.

- Le premier processus est un processus séquentiel qui initialise l'état présent au premier état et qui met à jour l'état présent par l'état futur de manière synchrone. La liste de sensibilité de ce processus comprend donc le signal d'horloge **clock_i** et le signal de réinitialisation **resetb_i**. La structure utilisée est **if then else**.
- Le deuxième processus est un processus combinatoire qui calcule l'état futur à partir de l'état présent et des entrées. La liste de sensibilité contient donc l'état présent et certaines entrées de la FSM. La structure utilisée est **case when** car elle permet d'énumérer exhaustivement tous les états. Au total, la machine pilote 8 signaux de la permutation, 5 signaux pour les compteurs et 2 sorties primaires, soit 15 signaux de contrôle.
- Le dernier processus est aussi combinatoire et permet de calculer les sorties à partir de l'état présent. La liste de sensibilité contient uniquement l'état présent et la structure utilisée est aussi une structure **case when**. Dans ce processus, on commence par initialiser toutes les sorties à une valeur par défaut, ce qui permet de n'écrire que les changements de signaux dans les **when**. On notera que le bit de sélection du multiplexeur **data_sel_o** est mis à 1 par défaut, car celui-ci restera à 1 après la toute première permutation.

On remarquera que l'on peut fusionner les deux processus combinatoires car leurs listes de sensibilité sont compatibles, mais pour des raisons de clarté et de compréhension, nous ne les regroupons pas.

L'entité de la machine d'états de Moore est donnée dans la figure 33.

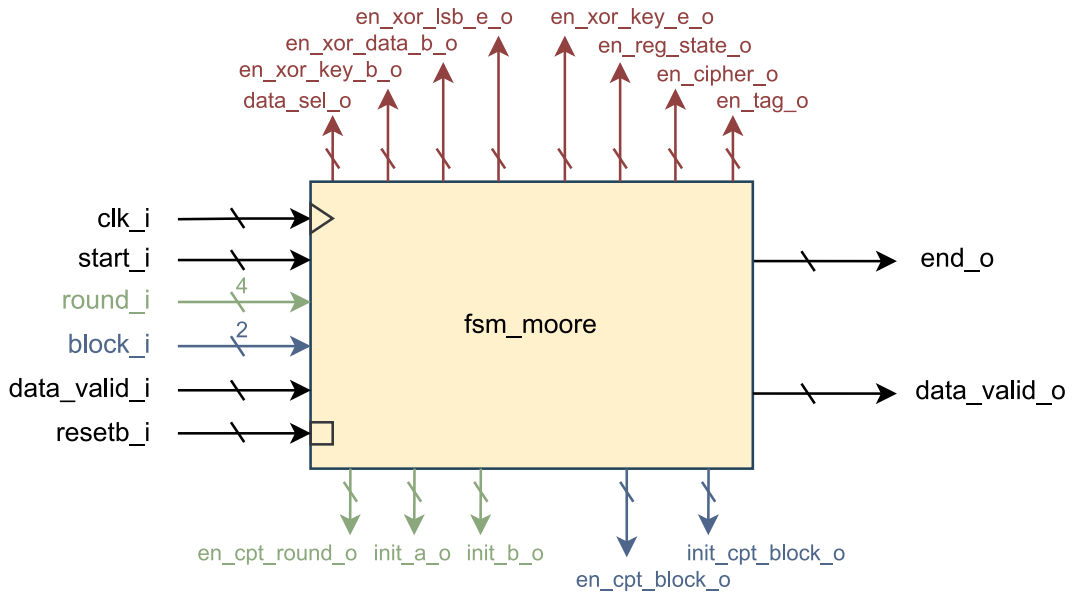


Figure 33 – Entité de la machine d'états finis de Moore

(En noir les entrées et sorties primaires, en rouge les sorties vers le composant permutation, en vert les entrées et sorties vers le compteur de rondes, en bleu les entrées et sorties vers le compteur de blocs)

Dans un premier temps, nous implémentons uniquement la phase d'initialisation. Nous implémentons le reste des états une fois que l'architecture globale est faite, afin de faciliter la vérification lors des simulations.

Le graphe d'états de la FSM partielle est donné en figure 34. Celui-ci permet d'écrire le processus combinatoire qui détermine l'état futur en fonction des entrées et de l'état présent. Le programme de chiffrement est lancé uniquement lorsque **start_i** est à 1. Le compteur de rondes est utilisé pour la première fois à l'état **init** : tant que le compteur ne vaut pas A, c'est-à-dire que la p^{12} n'est pas terminée, on boucle sur l'état **init**. La condition d'arrêt de la boucle est **round_i = x"A"** car le compteur commence à 0 et l'état **end_init** achève la p^{12} en réalisant la 12e permutation. Enfin, on reboucle sur l'état **idle_da** lorsqu'on y est car nous n'avons pas encore implémenté la suite.

La table 2 décrit la valeur des sorties que la machine d'états doit piloter pour chaque état.

- L'état **idle** ne sert que de point de départ du programme.
- A l'état **conf_init**, on initialise le compteur des rondes à 0 pour préparer la p^{12}.
- A l'état **end_conf_init**, on applique la première permutation avec le multiplexeur sélectionnant l'état d'entrée **state_i** (d'où **data_sel_o** à 0).
- A l'état **init**, on réalise les 10 permutations suivantes en prenant l'état de sortie en entrée.
- A l'état **end_init**, on réalise la dernière permutation avec le XOR de fin avec la clé.

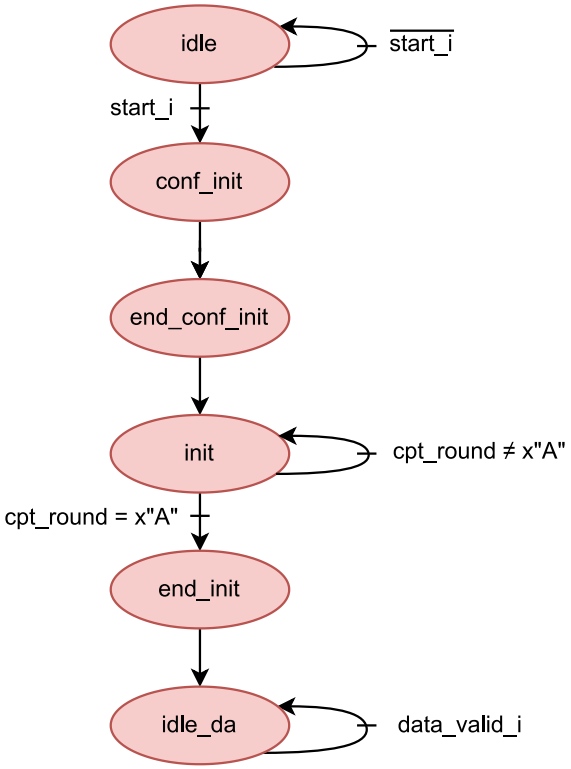


Figure 34 – Graphe d'états de la phase d'initialisation

	MUX	XOR				REG			CPT				
sorties	data_sel_o	en_xor_key_b_o	en_xor_data_b_o	en_xor_key_e_o	en_xor_lsb_e_o	en_reg_state_o	en_cipher_o	en_tag_o	en_cpt_round_o	Init_a_o	Init_b_o	en_cpt_block_o	Init_cpt_block_o
idle	0	0	0	0	0	0	0	0	0	0	0	0	0
conf_init	0	0	0	0	0	0	0	0	**1**	**1**	0	0	0
end_conf_init	0	0	0	0	0	**1**	0	0	1	**0**	0	0	0
init	**1**	0	0	0	0	1	0	0	1	0	0	0	0
end_init	1	0	0	**1**	0	1	0	0	**0**	0	0	0	0
idle_da	1	0	0	0	0	0	0	0	0	0	0	0	0

Table 2 – Valeurs des sorties de la FSM pour chaque état jusqu'à **idle_da**

5.2.2. Simulation de la FSM partielle

Pour simuler la FSM partielle, on simule le compteur de rondes manuellement comme s'il était relié à la machine d'états. On vérifie que pour chaque état les signaux de contrôle sont mis à jour au bon moment, et cela valide partiellement le composant (figure 35).

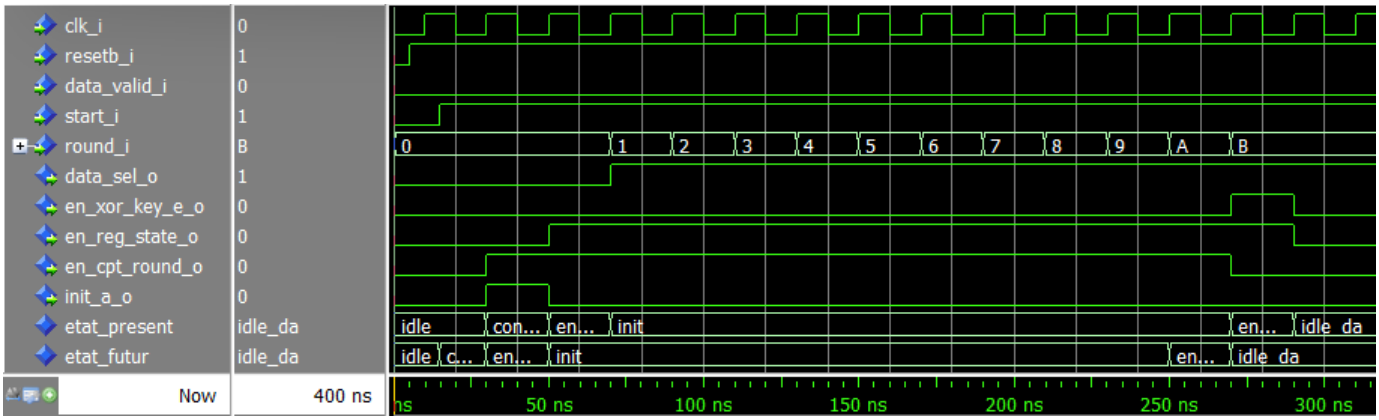


Figure 35 – Chronogramme du testbench de la FSM

5.3. Architecture globale ascon_top

L'architecture globale du processus de chiffrement ASCON128 est composée de 4 composants principaux :

- La machine d'états finis de Moore (FSM) qui contrôle les entrées des autres composants
- Le bloc de permutation qui réalise les rondes des permutations
- Le compteur de rondes qui gère le numéro de la ronde en cours
- Le compteur de blocs qui gère le numéro du bloc de chiffrement P_i courant.

L'architecture du composant global **ascon_top** est donnée dans la figure 36.

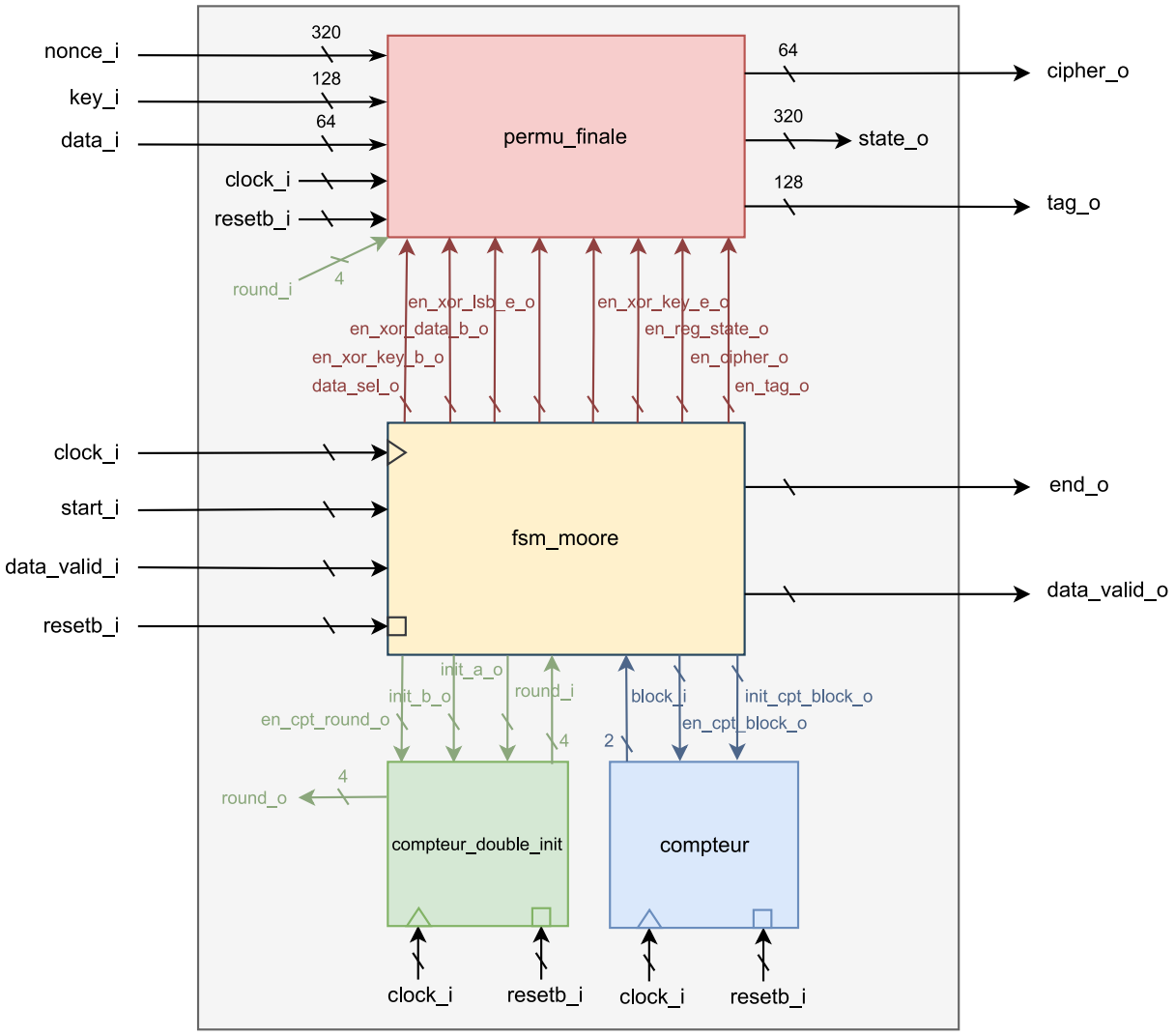


Figure 36 – Architecture globale **ascon_top** de ASCON128

5.3.1. Implémentation de ascon_top

On déclare les 4 composants dans la partie déclarative de l'architecture ainsi que tous les signaux intermédiaires qui permettent de relier les entrées et sorties des composants. Grâce à **port map**, on instancie les composants et on relie donc les signaux. Dans la description de l'architecture, on affecte aux registres du signal d'état en entrée **state_i_s** les données d'entrée : **data_i**, **key_i** et **nonce_i**.

5.3.2. Simulation complète de ascon_top

5.3.2.1. Implémentation complète de la FSM

Afin de simuler entièrement le processus de chiffrement ASCON128, nous finissons l'implémentation de la FSM.

Le graphe d'états complet de la FSM est donné dans la figure 37.

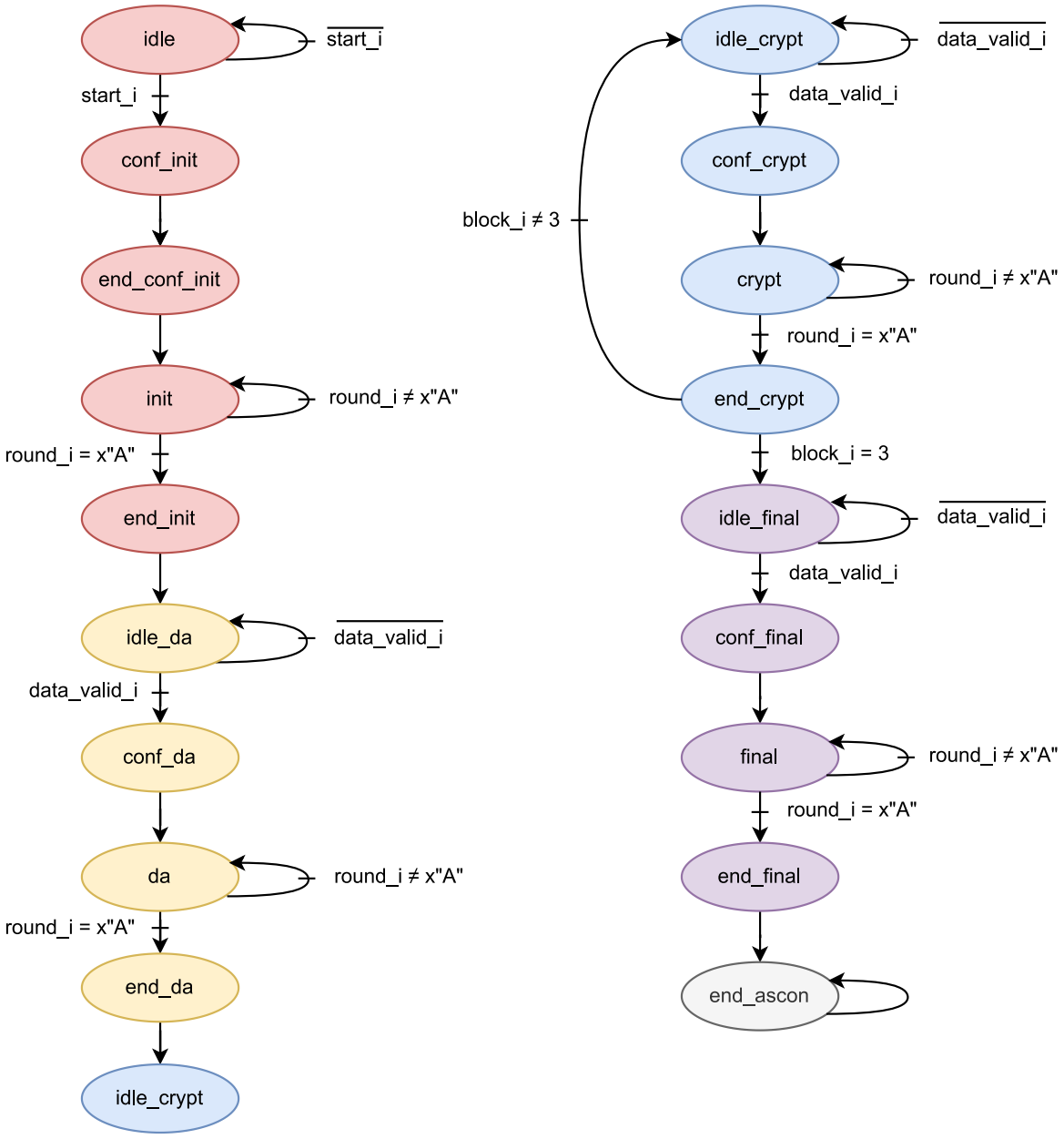


Figure 37 – Graphe d'états de la FSM

Pour chaque état commençant par « idle », on attend que la donnée soit valide avant de procéder aux permutations. On remarque l'implémentation du compteur de blocs dans la phase de chiffrement du texte clair (de **idle_crypt** à **end_crypt**).

A chaque état, la FSM pilote des signaux différents, ce qui est regroupé dans la table 3. L'explication des signaux de contrôle des états **idle** à **end_init** sont en 5.2.1.

sorties	MUX			XOR			REG			CPT				PRIMAIRE	
	data_sel_o	en_xor_key_b_o	en_xor_data_b_o	en_xor_key_e_o	en_xor_lsb_e_o	en_reg_state_o	en_cipher_o	en_tag_o	en_cpt_round_o	Init_a_o	Init_b_o	en_cpt_block_o	Init_cpt_block_o	cipher_valid_o	end_o
idle	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
conf_init	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
end_conf_init	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0
init	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0
end_init	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0
idle_da	1	0	0	0	0	0	0	0	1	0	1	0	0	0	0
conf_da	1	0	1	0	0	1	0	0	1	0	0	0	0	0	0
da	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0
end_da	1	0	0	0	1	1	0	0	0	0	0	1	1	0	0
idle_crypt	1	0	0	0	0	0	0	0	1	0	1	0	0	0	0
conf_crypt	1	0	1	0	0	1	1	0	1	0	0	1	0	1	0
crypt	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0
end_crypt	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0
idle_final	1	0	0	0	0	0	0	0	1	1	0	0	0	0	0
conf_final	1	1	1	0	0	1	1	0	1	0	0	0	0	1	0
final	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0
end_final	1	0	0	1	0	1	0	1	0	0	0	0	0	0	0
end_ascon	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Table 3 – Valeurs des sorties de la FSM pour chaque état du graphe

— A l'état **idle_da**, on initialise le compteur de rondes à 6 pour commencer la p^6.
— A l'état **conf_da**, on applique la première permutation avec le XOR d'entrée avec la donnée associée. On incrémente donc le compteur de rondes et on écrit dans le registre de l'état courant.
— A l'état **da**, on réalise les permutations suivantes toujours en incrémentant le compteur de rondes et en écrivant dans le registre de l'état courant.
— A l'état **end_da**, on réalise la dernière permutation avec le XOR de fin avec le 1 de bit de poids le plus faible et on initialise déjà le compteur de blocs à 0.
— A l'état **idle_crypt**, on initialise le compteur de rondes à 0, et c'est à cet état que l'on rebouclera pour les 2 prochains blocs de chiffrement.
— Aux états **conf_crypt**, **crypt** et **end_crypt**, on applique les permutations avec le XOR de début qui prend le texte clair. On incrémente le compteur de blocs à l'état **conf_crypt** et c'est là où on valide le texte chiffré.
— A l'état **idle_final**, on initialise le compteur de rondes à 0.

— A l'état **conf_final**, on réalise la permutation avec les deux XOR de début de permutation et on valide le dernier bloc chiffré C_4 en mettant **cipher_valid_o** à 1 et on écrit dans le registre du texte chiffré.
— Aux états **final** et **end_final**, on finit la p^{12}. A l'état **end_final**, on renvoie le tag en mettant **en_tag_o** à 1.
— Enfin, à l'état **end_ascon**, on déclare la fin du calcul en passant **end_o** à 1.

5.3.2.2. Simulation de ascon_top avec la FSM complète

Dans le testbench du composant global, on pilote encore manuellement certains signaux, en particulier **data_s** et **data_valid_s**. En effet, on met à jour les valeurs de **data_s** juste avant les états commençant par « idle », c'est-à-dire que l'on doit s'adapter en fonction du temps pris par le processus de chiffrement. A chaque fois, nous activons pendant 20 ns le signal **data_valid_s** 5 ns après avoir mis à jour la donnée **data_s**.

Le dernier bloc du texte à chiffrer P_4 est sur 56 bits. Il faut donc faire du **bit stuffing**, c'est-à-dire qu'on remplit les 8 bits restants à droite avec un 1 suivi de sept 0 :

$$P_4 \leftarrow 0\text{x}61676520564844 \,\&\, 0\text{b}10000000$$

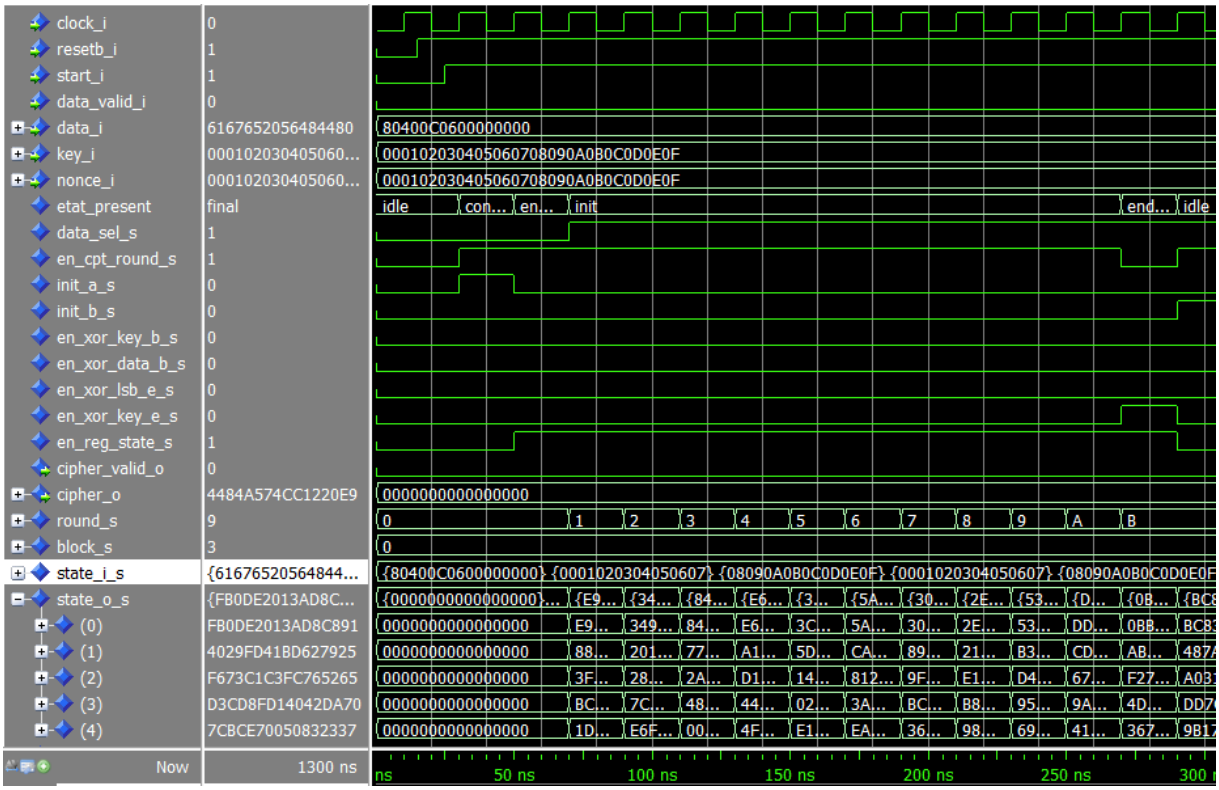
Cela revient à écrire :

$$\begin{aligned} & P_4 \leftarrow 0\text{x}61676520564844 \,\&\, 0\text{x}80 \\ \equiv \quad & P_4 \leftarrow 0\text{x}6167652056484480 \end{aligned}$$

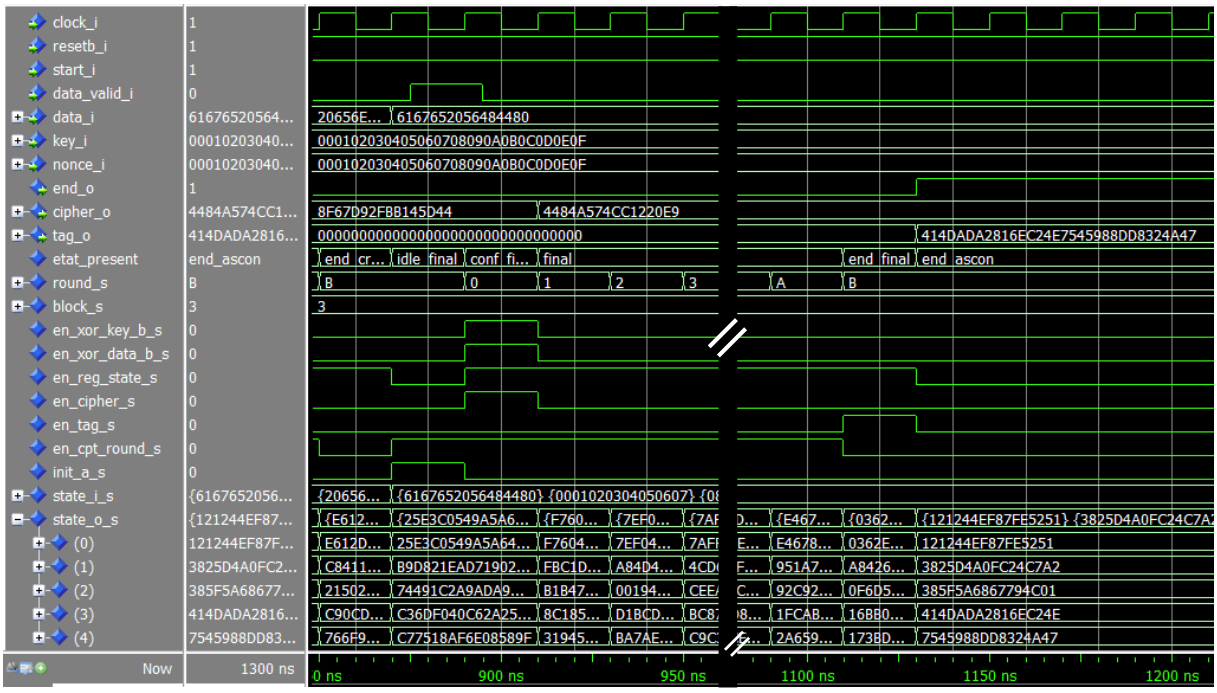
La figure 38a correspond au début du chronogramme de la simulation avec les signaux qui changent de valeur sur cette portion de temps. On vérifie à chaque étape que les valeurs des sorties **state_o_s**, **cipher_o** et **tag_o** sont correctes. Sinon, on vérifie à chaque état la valeur des signaux de contrôle pour trouver ce qui pose problème. On remarque que la dernière valeur du bloc chiffré **cipher_o** renvoyée est sur 64 bits, alors qu'on souhaite récupérer uniquement les 56 bits de poids fort. En effet, le texte chiffré complet est de taille $3 \times 8 + 7$ octets. Il faut donc ignorer les deux quartets de poids faible en sortie.

La figure 38b correspond à la fin de la simulation et permet de vérifier finalement le tag utilisé pour l'authentification et l'état de sortie des permutations.

Enfin, la figure 39 présente le chronogramme complet de la simulation de l'architecture globale.



(a) Chronogramme de début de simulation de **ascon_top**



(b) Chronogramme de fin de simulation de **ascon_top**

Figure 38 – Chronogrammes de simulation du composant global de ASCON128

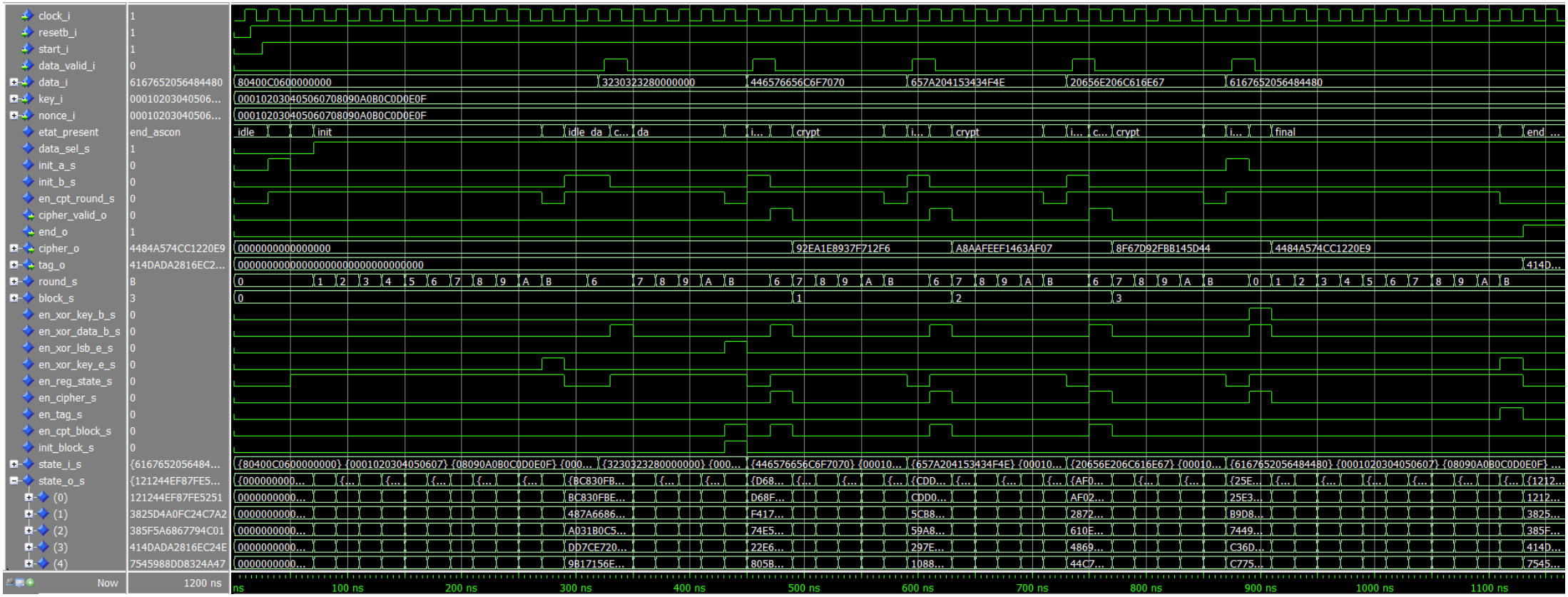


Figure 39 – Chronogramme complet de la simulation de l'architecture globale de ASCON128

6. Conclusion

Ce projet de conception de système numérique a été l'occasion d'approfondir considérablement mes connaissances en VHDL, en plus de comprendre précisément le fonctionnement d'un algorithme de chiffrement authentifié avec données associées. Nous avons en effet découvert pas à pas l'architecture de l'algorithme et compris au fur et à mesure le fonctionnement du langage et ses particularités.

La solution mise en œuvre est sans doute perfectible afin d'optimiser le circuit et accélérer le procédé de chiffrement. Des optimisations sont possibles, par exemple en simplifiant la machine d'états ou en réduisant le nombre de cycles nécessaires au traitement.

Comme tout algorithme de chiffrement symétrique, ASCON-128 nécessite le partage sécurisé d'une clé entre les interlocuteurs, ce qui peut être un défi dans certains contextes. Il serait également intéressant d'explorer d'autres variantes d'ASCON, adaptées à des tailles de clé ou à des exigences spécifiques en matière de performance ou de sécurité.

7. Références

- Jean-Max Dutertre, Olivier Potin, Guillaume Reymond, Jean-Baptiste Rigaud, « Modélisation VHDL de l'algorithme de déchiffrement ASCON, » Projet PCSN EI2021, 2022