Rapport du projet ASCON

Ness TCHENIO
Ecole des Mines de Saint-Etienne cycle ISMIN

16 mai 2024

Table des matières

1 Introduction

Dans une ère où nos données circulent à travers un réseau numérique mondial, leur chiffrement est essentiel pour préserver la confidentialité des échanges et informations personnelles.
L'algorithme ASCON est un des algorithmes de chiffrement existant, utilisé notamment dans les dispositifs IoT (Internet of Things) et dans les systèmes embarqués, garantissant la sécurité des données sensibles échangées dans ces environnements.

Chaque algorithme ASCON a une clé de chiffrement : les algorithmes ASCON128 ont des clés de 128 bits assurant une sécurité robuste des données.
La réalisation du projet consiste en la conception et la simulation d'un système de cryptage ASCON128 dans le langage de description matériel SystemVerilog.

2 Présentation de l'algorithme ASCON

2.1 Structure globale

L'algorithme ASCON opère sur un état courant (ou état S) de 320 bits. A cet état S, on associe le type *type_ state* , défini dans le module ascon_pack qui décompose les 320 bits en 5 registres de 64 bits (x_0 , x_1, x_2, x_3 et x_4).

L'algorithme de chiffrement de données ASCON128 se décompose en quatre blocs :

- L'initialisation (en violet)
- La donnée associée (en rose)
- Le texte clair (en bleu)
- La finalisation (en orange)

Ces blocs correspondent à des étapes de chiffrement des données en entrée, état S, jusqu'à l'obtention du Tag de 128 bits, permettant la vérification de l'authenticité du message reçu.

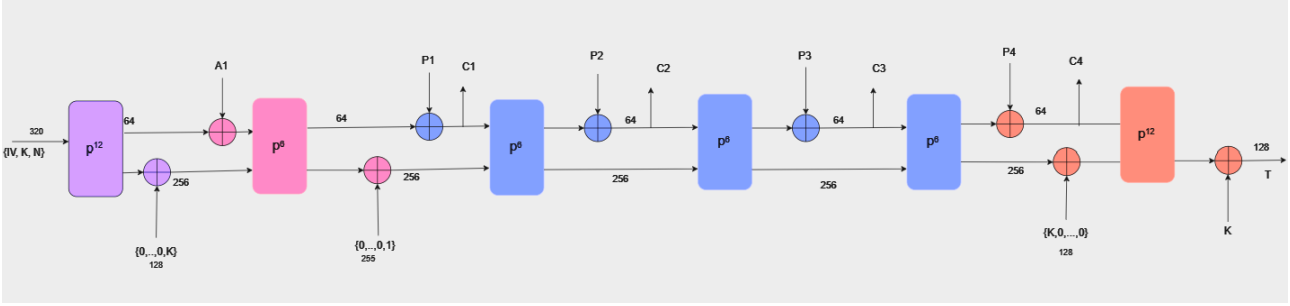

FIGURE 1 – ASCON_128

Chacun de ces blocs est constitué d'une, ou plusieurs, permutation-s et de xor (begin et/ou end). Nous étudierons l'implémentation de ces différents modules dans un second temps.

2.2 Structure du dossier SystemVerilog

La mise en œuvre d'un tel projet nécessite une organisation rigoureuse des différents répertoires. Les codes SystemVerilog sont répartis dans les répertoires SRC et LIB. Les fichiers sources sont placés dans SRC/RTL et les librairies associées dans LIB/LIB_RTL.
Les fichiers de test sont situés dans SRC/ BENCH et les librairies associées dans LIB/LIB_BENCH. Les fichiers de test (*tesbench*) permettent l'éxecution des modules auxquels ils sont associés et ainsi de vérifier leur fonctionnement. Ces *testbench* sont simulés sur l'envrionnement ModelSim.

Le fichier init_modelsim.txt (ASCON / init_modelsim.txt) permet d'éxecuter les commandes afin d'initialiser correctement ModelSim.
Le fichier compile_src.txt (ASCON / compile_src.txt) permet l'éxecution de toutes les commandes de compilation, ainsi que le lancement de la simulation d'un *testbench*, en une seule commande.

3 Les différents modules

3.1 La permutation

La permutation joue un rôle clé dans le chiffrement de données. Chacun de ses composants, notamment par le biais d'opérations non-linéaires, mélange les bits de données assurant l'efficacité de l'algorithme de chiffrement et son indéchiffrabilité.

La permutation est composée de trois blocs :
- Add_const
- Couche_substitution
- Couche_diffusion

3.1.1 Addition de constante

L'addition de constante (add_const) agit seulement sur le registre x_2 de l'état S. Elle lui ajoute une constante de ronde i , à chaque ronde de la permutation.
Elle se modélise comme un xor entre le registre x_2 et la constante de ronde c_r
$x_2 \leftarrow x_2 \oplus c_r$

Ronde r de p^{12}	Ronde r de p^6	Constante c_r
0		0000000000000000f0
1		0000000000000000e1
2		0000000000000000d2
3		0000000000000000c3
4		0000000000000000b4
5		0000000000000000a5
6	0	0000000000000000096
7	1	0000000000000000087
8	2	0000000000000000078
9	3	0000000000000000069
10	4	000000000000000005a
11	5	000000000000000004b

FIGURE 2 – table de définition de la constante de ronde

Nous pouvons modéliser ce module ainsi :

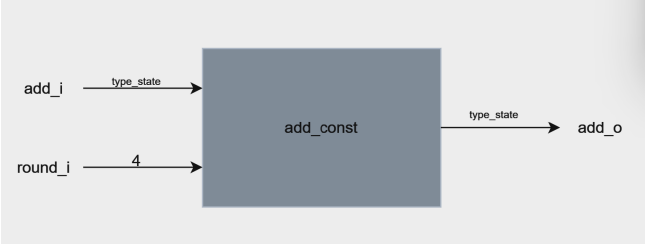

FIGURE 3 – module add_const.sv

L'entrée round_i agit comme un pointeur vers une des constantes c_r qui sera additionnée avec la valeur dans l'entrée add_i. La valeur de round_i est incrémentée de 1 après chaque ronde de permutation.

La sortie add_o correspond au résultat de l'addition entre add_i et une constante.

Le bon fonctionnement du module est assuré par la cohérence des résultats lus en sortie du *testbench*. Ici, l'objectif est de simuler une permutation p^6, et retrouver en sortie le résultat indiqué sur la table de validation, pour chacune des valeurs de round_i.

FIGURE 4 – *testbench* du module add_const.sv

On constate ici que c'est bien le cas.

3.1.2 Couche de substitution

Dans le module de couche de substitution, l'état S est considéré comme un tableau de 5 lignes et 64 colonnes (les lignes contiennent respectivement x_0, x_1, x_2, x_3 et x_4).
Cette transformation va traiter les données par colonnes et appliquer aux 5 bits de chaque colonne, la table de subtistion s_box définie ainsi :

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 5 – table de la sbox

Le premier des 5 bits lus correspond au Most Significant Bit (MSB). On obtient donc pour chaque colonne un nombre binaire à 5 bits qu'on convertit en héxadécimal. On lit alors ce nombre (x) sur la première ligne de la Figure 5 et on le remplace par le nombre S(x), lu sur la seconde ligne.

On représente ainsi ce module :

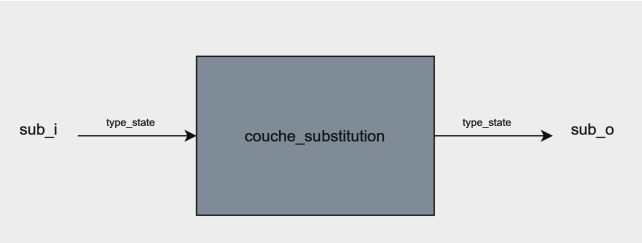

FIGURE 6 – module couche_substitution.sv

On réalise à nouveau un *testbench*, pour vérifier le fonctionnement du module de couche de substitution. On prend en entrée, la valeur sortant du module d'addition de constante de la première ronde de permutation. On compare la valeur en sortie, avec celle annoncée dans le sujet pour la substitution de cette même ronde de permutation.

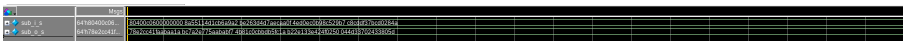

FIGURE 7 – *testbench* du module couche_substitution.sv

Le bon fonctionnement du module permutation_finale, étudié plus loin dans le rapport,nous exempte de tester la couche de substitution (et la couche de diffusion) sur les 6/12 rondes de permutation.

3.1.3 Couche de diffusion

Le module de couche de diffusion agit sur chacun des 5 registres de l'état S.

L'écriture $x_i \gg j$ est implémenté ainsi : diff_i[i]←diff_i[i][j-1 :0],diff._i[i][63 :j] avec diff_i[i] le registre i de l'état S en entrée du module couche_diffusion.
La diffusion correspond à une rotation cyclique de j bits vers la droite.

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

FIGURE 8 – opérations de la couche de diffusion

Le module est implémenté ainsi :

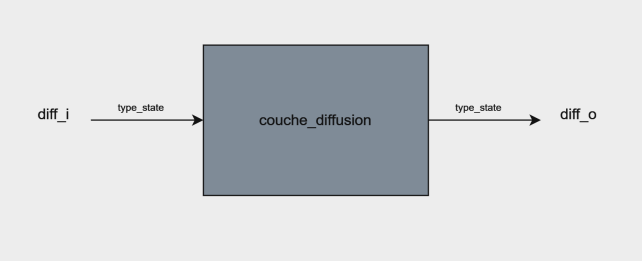

FIGURE 9 – module couche_diffusion

Pour le *testbench*, on applique la sortie du module substitution de la première ronde de permutation, et on obtient bien le résultat attendu.

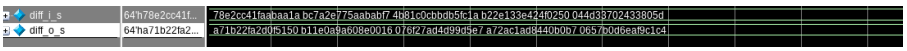

FIGURE 10 – *testbench* du module couche_diffusion

3.2 Les XOR

Au sein du système ASCON, la permutation comprend tantôt 6 itérations (noté p^6), tantôt 12 itérations(noté p^{12}).
A ces trois modules de permutation s'ajoutent des xor, déclenchés en amont (xor_begin) ou en aval (xor_end) des permutations. Les xor sont répartis ainsi :

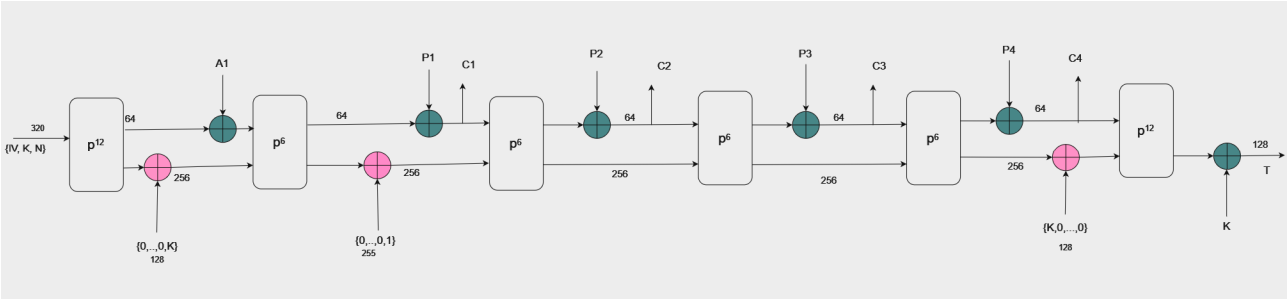

FIGURE 11 – répartition des xor_begin et xor_end.sv

3.2.1 xor_begin

Le xor_begin est ainsi modélisé ainsi :

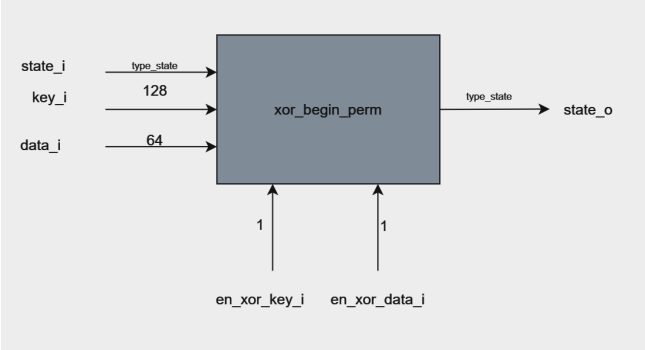

FIGURE 12 – module xor_begin_perm.sv

En entrée on retrouve la donnée à chiffrer (state_i), la clé de chiffrement (key_i), et la valeur appliquée au xor.
L'activation de ce xor (en bleu) est conditionnée par deux entrées différenciant ses utilisations. Les entrées xor_key_beg_i et xor_d_beg_i agissent comme des pointeurs vers la valeur qui sera appliquée au xor.

— xor_d_beg_i = 1

.En donnée associée : $x_0 \leftarrow x_0 \oplus A_1$
.Avant chaque permutation dans texte clair. :
$x_0 \leftarrow x_0 \oplus P_1$
$x_0 \leftarrow x_0 \oplus P_2$

$x_0 \leftarrow x_0 \oplus P_3$
$x_0 \leftarrow x_0 \oplus P_4$

— xor_key_beg_i = 1

. En finalisation :x_1,x_2← x_1,x_2 ⊕ Key

3.2.2 xor_end

Le xor_end est utilisé en fin de permutation (à la dernière ronde).
Il est modélisé ainsi :

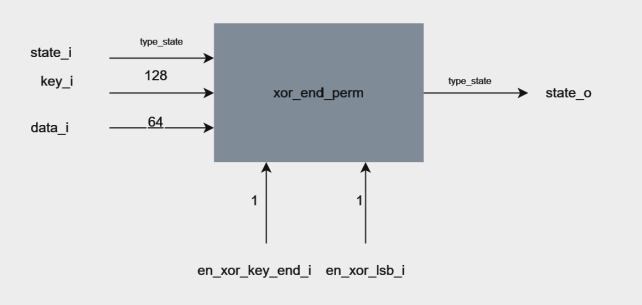

FIGURE 13 – module xor_end_perm.sv

Les entrées xor_lsb_end_i et xor_key_end_i jouent le même rôle que xor_key_beg.i et xor_d_beg_i. L'activation du xor_end (en rose) est conditionnée par ces deux entrées, différenciant ses utilisations.

xor_key_end_i = 1

. En fin d'initialisation : x_3, $x_4 \leftarrow x_3$, $x_4 \oplus$ Key
. En fin de texte clair : x_1, $x_2 \leftarrow x_1$, $x_2 \oplus$ Key

xor_lsb_end_i = 1

. En fin de donnée associée : $x_4[0] \leftarrow x_4[0] \oplus 1$

3.3 Mux et Register_state

3.3.1 state_register_w_en

L'état courant S va subir des ensembles de permutations successives.
Il est donc nécessaire que l'état de fin d'une permutation soit mémorisé. Pour cela, on a implémenté un module state_register_w_en qui, une fois activé, mémorise le nouvel état de la donnée. On modélise ce module ainsi :

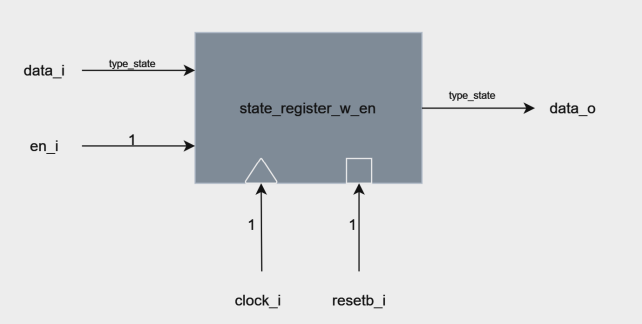

FIGURE 14 – module register_state.sv

3.3.2 mux_state

Le module mux_state permet de sélectionner la donnée qui va subir la permutation.
L'activation, ou non, de l'entrée sel_i permet la sélection de données de l'une ou l'autre des entrées data1_i et data2_i .

Pour la première ronde, le mux sélectionne la donnée entrant dans le système (donnée à chiffrer) puis pour les rondes suivantes, il sélectionne la donnée sortant de la ronde de permutation précédente.
Il est modélisé ainsi :

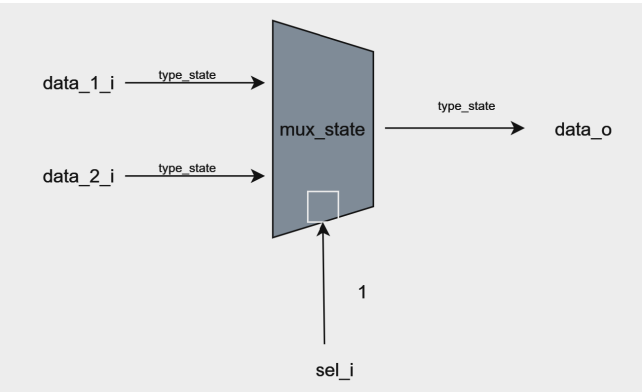

FIGURE 15 – module mux_state.sv

3.4 La permutation finale

La permutation finale utilise tous les modules précédemment expliqués et les interconnecte afin d'effectuer la permutation complète. Elle est modélisée ainsi :

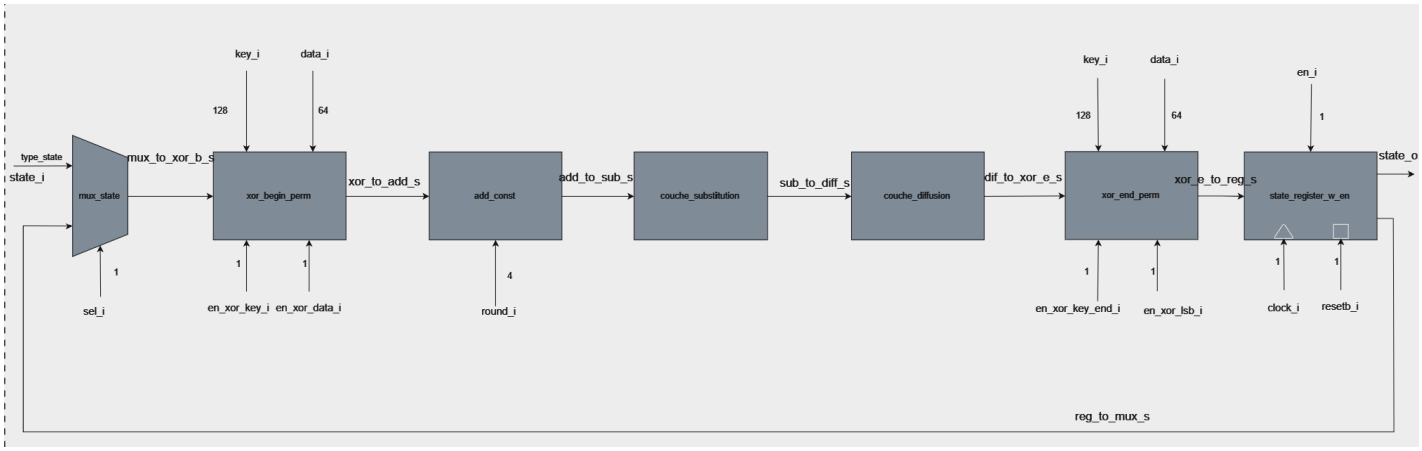

FIGURE 16 – module permutation_finale.sv

Pour vérifier le bon fonctionnement de cette permutation, nous vérifions le résultat en sortie de la phase "Initialisation".
Nous appliquons en entrée, la valeur initiale indiquée dans le sujet. Dans le *testbench* de la permutation finale, à l'issue de la permutation p^{12} , on active l'entrée xor_key_end_i pour activer le xor entre la donnée en sortie et 0,...,0,K où K est la clé de chiffrement sur 128 bits.

Nous lisons le résultat souhaité sur la ligne « reg_to_mux» pour counter_i = b.

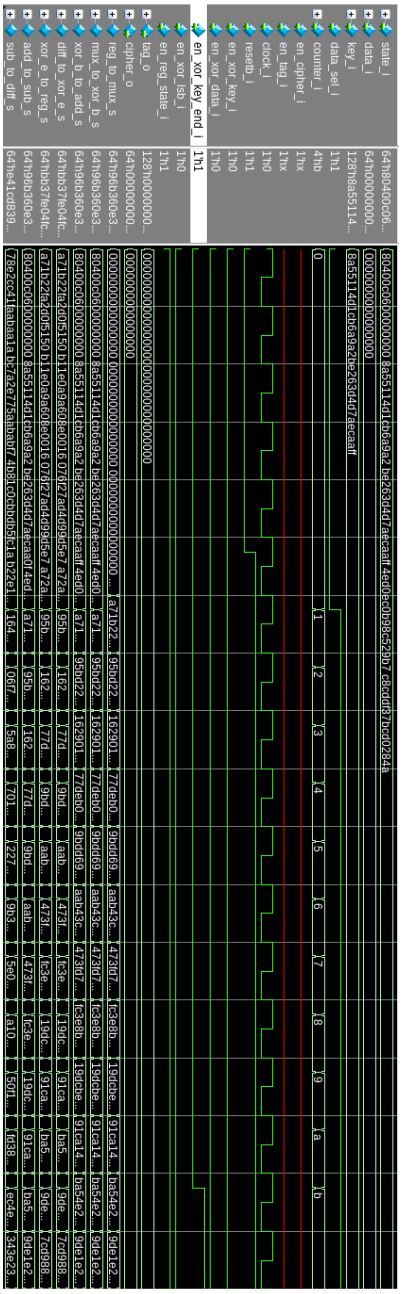

FIGURE 17 – testbench module permutation_finale.sv

En sortie du *testbench*, on constate que la sortie reg_to_mux est décalée d'une ronde par rapport au résultat attendu. Cela s'explique par l'activation du reg_state qui n'enregistre les résultats qu'à l'issue de la première ronde de permutation.

4 ASCON TOP

4.1 Finale state machine(FSM)

La machine d'état, doit gérer les différentes variables de contôles utilisées lors de la permutation.
Elle va donc activer ou désactiver les variables de contrôle, en fonction de l'étape en cours.

Les variables d'entrée du module sont les suivantes :
— start_i
— clock_i
— resetb_i
— data_valid_i
— round_i

Les variables de contrôle sont les variables en sortie du module :
— data_select_o, variable de sélection du mux
— en_xor_key_beg_o
— en_xor_d_beg_o
— en_xor_key_end_o
— en_xor_lsb_end_o
— en_reg_state_o
— cipher_valid_o
— end_o
— en_cpt_round_o
— init_a_o, variable qu'on active pour une permutation à 12 rondes
— init_b_o,variable qu'on active pour une permutation à 6 rondes

Dans mon code, j'ai initialisé toutes les variables à une valeur par défaut. Ainsi pour chaque état, certaines valeurs sont modifiées, puis reprennent leur valeur initiale à la fin de l'état.
On peut modéliser le diagramme d'état suivant où les variables à modifier sont indiquées à côté de chaque état.

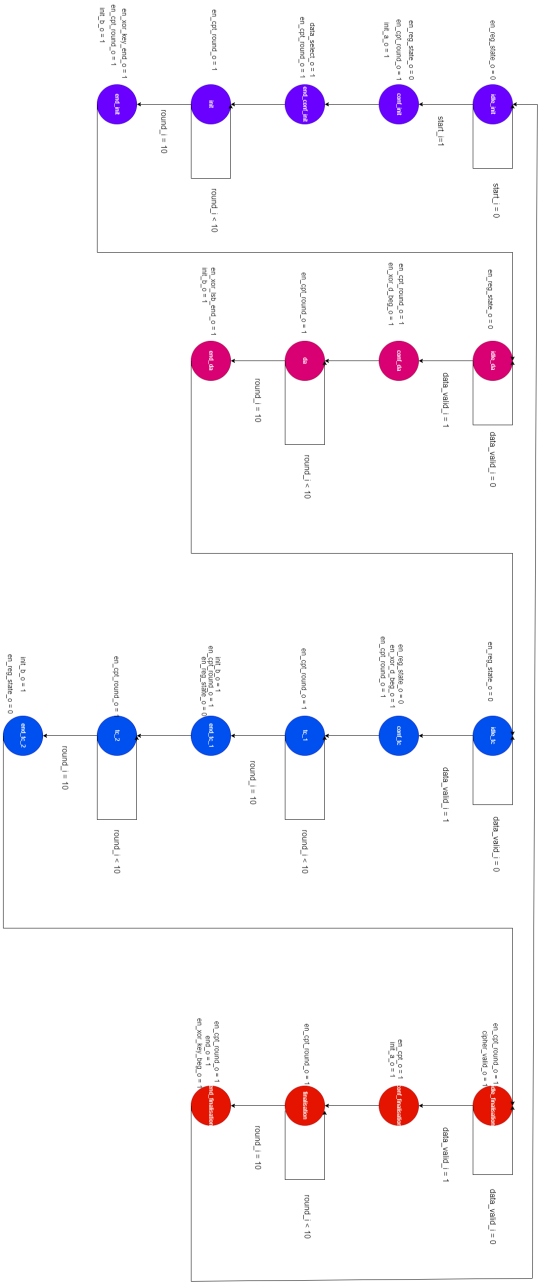

FIGURE 18 – Diagramme d'états

4.2 Le module ASCON_top

Le module ASCON_top permet de lier les modules
permutation_finale
FSM_moore
compteur_double_init

On peut le modéliser ainsi

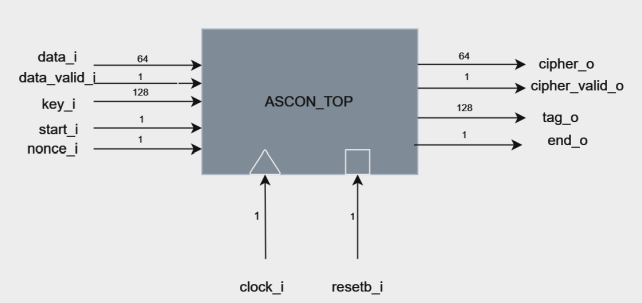

FIGURE 19 – module ascon_top.sv

On réalise un *testbench* permettant de vérifier la validité de l'implémentation complète réalisée. Nous mettons en entrée les valeurs de clé K, la valeur de A, celle de N et l'état S donnés dans le sujet. On divise aussi la valeur P donnée, en 4 blocs de 64 bits (en appliquant un padding manuellement si nécessaire)

start_i est activé (passé à l'état haut) pour lancer le chiffrement.

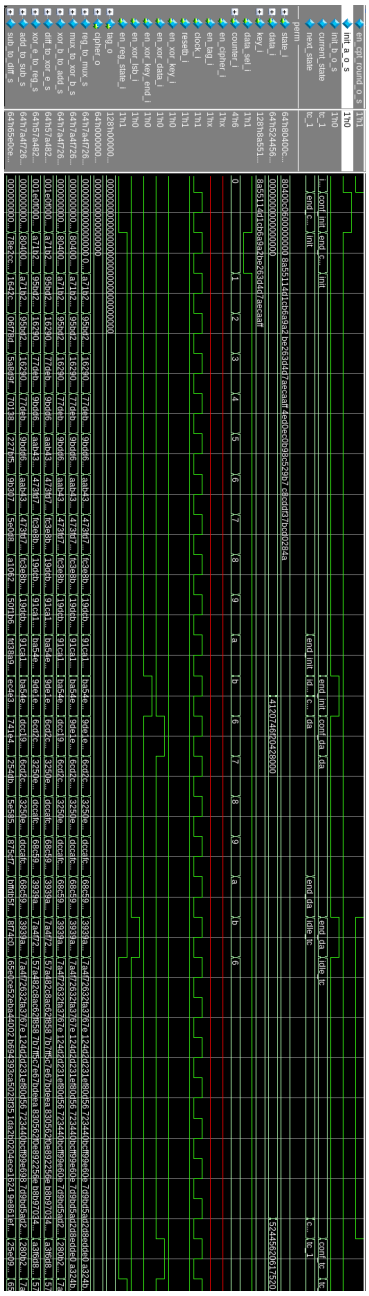

FIGURE 20 – testbench du module ascon_top.sv

On constate que les résultats sont conformes à ceux attendus, jusqu'à l'état end_da.
On retrouve la valeur correcte pour l'état idle_tc, décalée d'une ronde, puis les valeurs ne sont plus cohérentes.

5 Conclusion

5.1 Les points à corriger

L'algorithme dans son état ne permet pas le chiffrement complet de la donnée en entrée. Cela est dû à une activation incorrecte des variables d'état pour les états des blocs texte clair et finalisation. Corriger cela dans le module FSM_moore aurait probablement réglé ce défaut.

5.2 Retour sur le projet

Finalement ce projet a été très instructif. Il m'a d'une part familiarisée avec de nouveaux logiciels (lateX,ModelSim), et avec le langage de programmation SystemVerilog mais m'a aussi appris à être rigoureuse dans mon travail.
Cela a été essentiel, par exemple, pour la mise en place de la Machine à États Finis (FSM), où chaque détail compte et où une compréhension approfondie des variables manipulées est cruciale pour l'obtention de résultats probants.