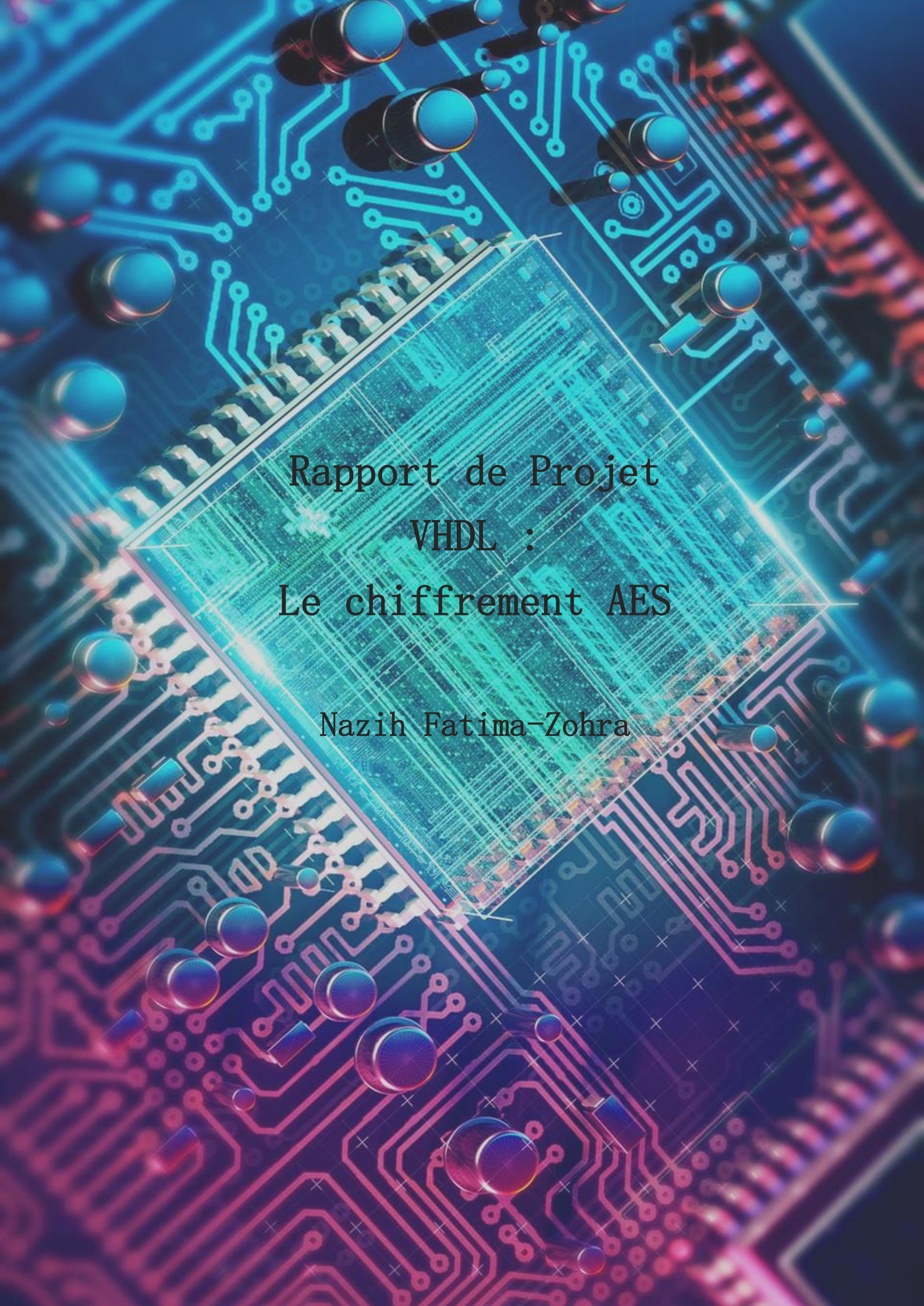


Rapport de Projet VHDL : Le chiffrement AES

Nazih Fatima-Zohra

Introduction

Le rapport qui suit explique la méthode appliquée pour la conception de l'algorithme de chiffrement Advanced Encryption Standard [AES], qui utilise un protocole symétrique (c'est-à-dire qu'il faut appliquer la même clé pour chiffrer et déchiffrer le message). Celui-ci se veut de respecter les propriétés d'authentification et de confidentialité.

Le chiffrement AES consiste en la substitution et la permutation de l'information (`plain_text`) afin de le transformer en un texte chiffré (`cipher_text`) ayant subi des modifications durant 11 rondes, plus ou moins différentes. Même si l'algorithme de chiffrement est disponible en open source sur Internet, son mystère réside dans la clé de chiffrement. En effet, il peut fonctionner avec plusieurs tailles de clés (128 bits, 192 bits et 256 bits) mais, la taille du message durant le projet ayant été définie à 128 bits, le choix d'une clé de 128 bits a été effectué en début de projet.

Le système global du Top Level de l'AES est schématisé sur la figure 22, en dernière partie.

Dû à la limite de temps imposée, le découpage de la clé initiale en onze sous-clés (pour chacune des rondes) a été donné par les professeurs dans le dossier `SRC/THIRDPARTY`.

Le bit de reset a été défini comme étant actif à l'état bas dans ce projet.

Fonctionnement de l'algorithme de chiffrement AES et méthode utilisée pour la description

La donnée entrée de type `bit128` sera tout d'abord convertie en une matrice 4x4, chacune de ses cases contenant un octet du message. Cette matrice, appelée **état**, sera de type `type_state`, type défini par les professeurs pour le projet. En effet, le type `type_state` est composé de quatre `row_states`, qui sont eux-mêmes des vecteurs composés de quatre `bit8`.

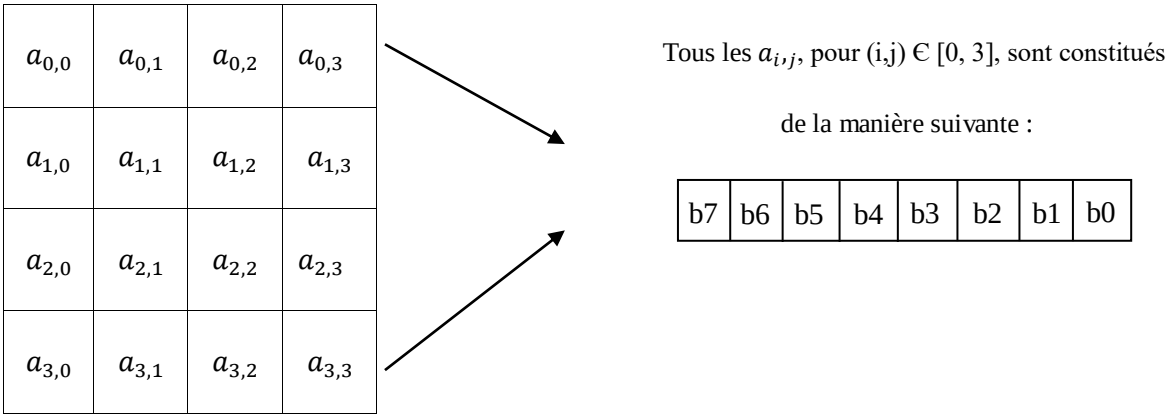


fig. 1 : représentation du type `type_state`

Ainsi, si on veut accéder à l'octet ayant comme indices i et j sur notre état `etat`, il suffit d'écrire : `etat[i][j]`.

Le traitement de la donnée, c'est-à-dire son chiffrement, se fait selon quatre substitutions ou permutations, effectuées de manière plus ou moins régulière durant onze rondes (Rounds) :

- Round 0 : fonction AddRoundKey
- Round 1-9 : fonctions Subbyte, ShiftRows, MixColumns puis AddRoundKey
- Round 10 : fonction Subbyte, ShiftRows, puis AddRoundKey

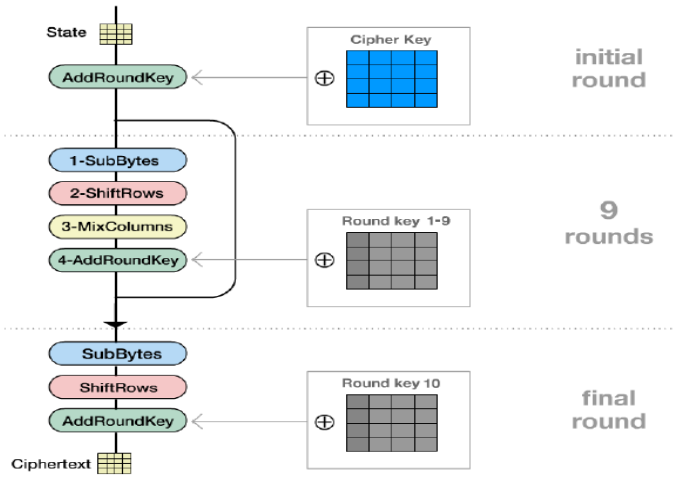


fig. 2 : Fonctionnement général du chiffrement AES – les différents Rounds

SubByte

Le SubByte est une fonction de substitution du message à chiffrer. Son fonctionnement réside sur une table de substitution : la S-Box (Substitution Box). Cette fonction prend un état en entrée, et un état en sortie.

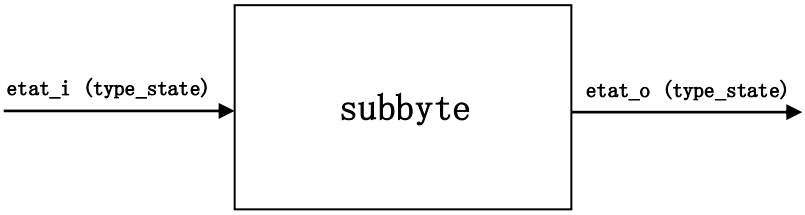


fig. 3 : schéma global de la fonction subbyte

La S-Box est une table de dimension 16x16 (dépendant de l'algorithme de Rijndael choisi) : les seize lignes et colonnes sont numérotées de 0 à F en hexadécimal, et on associe à chaque octet (un octet correspondant à deux caractères en hexadécimal) un nouvel octet, retrouvé grâce aux lignes et colonnes de la S-Box.

En effet, si l'octet que l'on souhaite substituer vaut ED en hexadécimal, alors sont octet correspondant se lit sur la S-Box :

X''ED''

	y															
	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

x

fig. 4 : schéma d'explication du principe de fonctionnement de la Substitution Box

Le SubByte consiste à substituer chacun des seize octets de l'état donné en entrée. Pour cela, l'utilisation d'un `generate` a été choisie, afin de générer une S-Box à chacun des octets du message et effectuer les seize substitutions en parallèle.

Le testbench du SubByte a pour résultats les données suivantes :

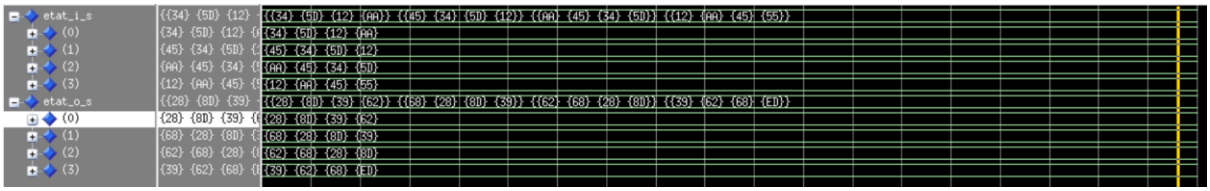


Figure 5 : résultats du testbench du SubByte

On remarque que les résultats du testbench sont bien ceux attendus avec la S-Box considérée, on peut le vérifier facilement à partir de la table donnée en figure 4.

ShiftRows

Le ShiftRows prend en entrée un état, et consiste à décaler chacun des éléments de chaque ligne i de i places vers la gauche (la case la plus à gauche devient celle de droite), i correspondant à l'indice de la ligne et pouvant valoir 0, 1, 2 ou 3, pour donner un nouvel état permuté en sortie. Ainsi, si aucun changement n'est effectué sur la première ligne, chacune des deuxième, troisième et quatrième lignes sont respectivement « décalées » d'une, deux et trois cases vers la gauche.

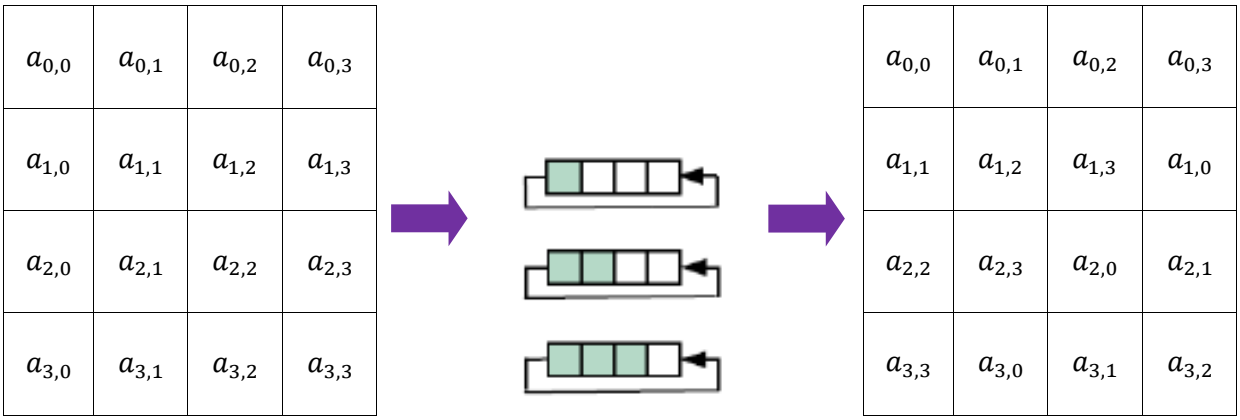


fig. 6 : fonctionnement du ShiftRows

Pour décrire cette permutation, une redéfinition indice par indice a été choisie (car matrice de petite taille). Ainsi, on réécrit simplement l'était de sortie en utilisant les octets de l'état pris en entrée.

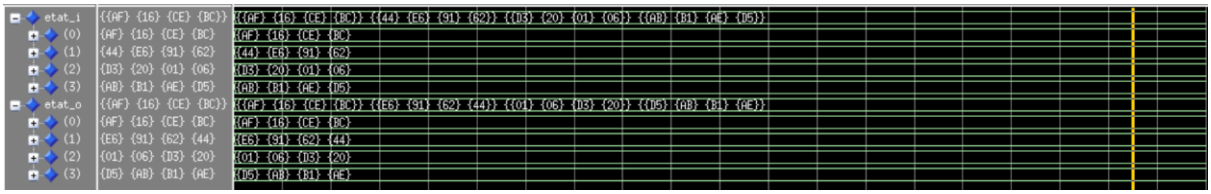


fig. 7 : résultats du testbench du ShiftRows

Ces résultats sont en corrélation avec les résultats attendus sur le schéma explicatif (fig. 6).

MixColumns

La fonction MixColumns utilise la matrice de Rijndael de taille 4x4 et consiste en la multiplication matricielle de cette matrice de Rijndael avec l'état considéré dans le corps de Gallois. Cette fonction prend un état en entrée, et retourne en sortie le résultat de la multiplication de la matrice de Rijndael par notre état pris en entrée. Cette matrice de Rijndael est une matrice dont les coefficients sont exclusivement des 1, 2 et 3. Ainsi, lors de la multiplication matricielle, chacun des octets peut rester intact, être multiplié par deux ou bien par trois.

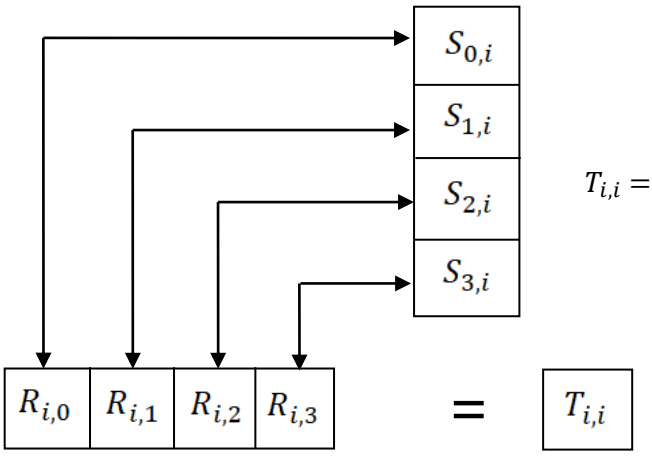
Afin de faciliter le calcul, l'insertion de deux signaux internes de type `type_state` ont été créés : `etat2` et `etat3`. Ces deux matrices correspondent respectivement à la matrice d'entrée dont tous les coefficients ont été « multipliés » par deux ou par trois. La multiplication dans le corps de Gallois se fait de la manière suivante :

Pour multiplier par deux :

- Si le bit de poids fort de l'octet (le MSB) de l'état de départ vaut '0', alors la multiplication par deux revient à décaler tous les bits d'un cran vers la gauche.
- Si le bit de poids fort de l'octet de l'état de départ vaut '1', alors on décale les sept derniers bits de l'octet d'un cran vers la gauche, on met le nouveau bit de poids faible à zéro, puis on effectue un XOR du nouvel octet avec l'octet suivant : '00011011'. Cela permet de faire un "modulo" afin de toujours garder une valeur sur 8 bits.

Pour multiplier par trois, on part du principe que 3 = 2 + 1, sauf que l'addition dans le corps de Gallois revient à effectuer un XOR. Aussi, pour compléter `etat3`, il nous suffit de prendre chacune des cases et de faire un XOR entre l'état de départ (en entrée du MixColumns) et celui créé juste au-dessus (`etat2`).

Par la suite, afin d'obtenir l'état obtenu en sortie, le calcul du produit matriciel a d'abord été effectué sur papier, afin d'attribuer à chacune des cases de l'état de sortie la bonne valeur. Par exemple :



On remarque sur la figure ci-contre le calcul effectué dans la fonction.
Pour chacune des cases, le résultat vaut :

$$T_{i,i} = R_{i,0} \,.\, S_{0,i} \;\otimes\; R_{i,1} \,.\, S_1 \otimes R_{i,2} \,.\, S_2 \otimes R_{i,3} \,.\, S_{0,i}$$

fig. 8 : schéma expliquant le calcul du MixColumns

Ainsi, pour obtenir l'état de sortie, il suffisait d'observer la matrice de Rijndael choisie sur le sujet, et attribuer à chacune des cases de l'état sa valeur en choisissant les indices dans etat_i, etat2 et etat3 et en effectuant un XOR entre chacun des termes.

Les résultats obtenus au testbench sont les suivants :

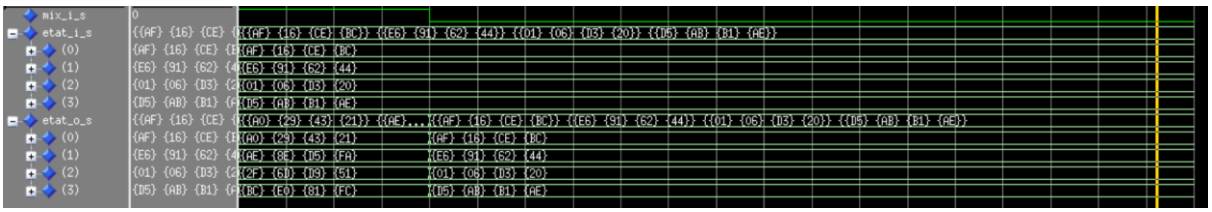


fig. 9 : résultats du testbench du MixColumns

AddRoundKey

Le but de cette étape est d'effectuer un XOR bit à bit entre l'état `etat_i` pris en entrée, et la clé `key_i`.

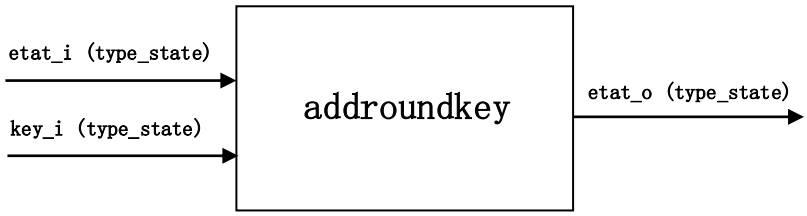


fig. 10 : schéma global de l'AddRoundKey

Lors de chacun des rounds effectués, une nouvelle clé est générée. Celles-ci sont en fait des sous-clés de la clé réelle du chiffrement ; le découpage en sous-clés a déjà été effectué dans le fichier "`KeyExpansion_I_O.vhd`" afin de nous faciliter le projet. Ainsi, la méthode choisie pour décrire cette fonction a été de choisir chacun des bits de l'état d'entrée et de faire un XOR avec le bit de même indice dans la clé, afin d'avoir l'état désiré en sortie.

Les résultats du testbench réalisé sont les suivants :

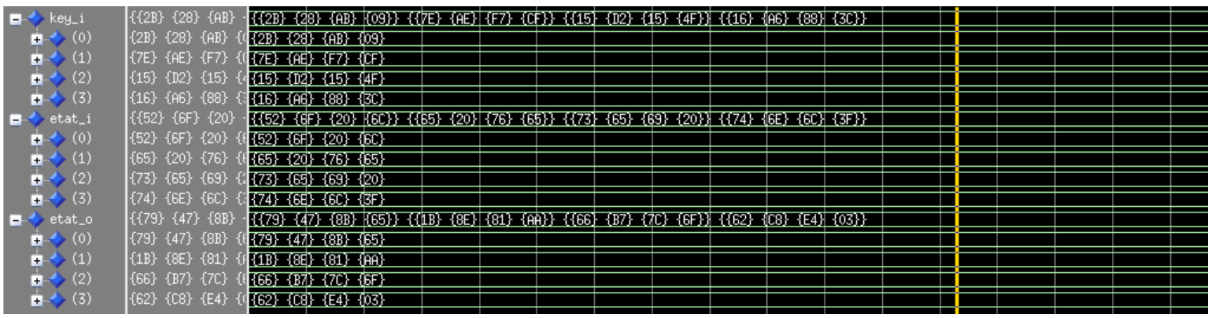


fig. 11 : résultats du testbench de l'AddRoundKey

Cette fonction est simplement constituée d'un double `generate` représentant deux boucles imbriquées qui parcourent les lignes et colonnes des entrées, et qui effectuent un XOR bit à bit entre elles.

Mise en commun des fonctions pour simuler un round : le fichier AESround.vhd

Après avoir rédigé et testé chacune des fonctions de chiffrement à l'aide de testbenches, il faut maintenant toutes les mettre en commun, tout en respectant les règles du protocole concernant chacun des Rounds, expliquées en **1)**. Pour cela, l'ajout de multiplexeurs à deux entrées et une sortie au niveau des fonctions AddRoundKey et MixColumns vont être nécessaires :

En effet, lors du Round 0, le texte en clair que nous souhaitons chiffrer est tout d'abord mélangé avec la première sous-clé, c'est-à-dire que le AddRoundKey prend le texte clair en entrée (après l'avoir préalablement converti, voir partie **8**) alors que lors de tous les Rounds suivants, la fonction AddRoundKey prend en entrée l'état en sortie du MixColumns.

De plus, le Round 10 ne prend pas en compte la fonction MixColumns : l'ajout d'un multiplexeur interne à la fonction lui permet de décider si l'état en sortie prend le résultat du produit matriciel avec la matrice de Rijndael (pour les Rounds 1 à 9) ou bien s'il reste identique à l'état pris en entrée du MixColumns (Round 10).

L'ajout d'un registre (fichier "registre.vhd") permet de mémoriser la donnée en sortie de l'AddRoundKey et de contrôler la gestion de l'AESround. En effet, le bit resetb_i, actif à l'état bas dans ce projet, permet de renvoyer en sortie un état dont tous les octets sont à zéro si on le met à '0'. Au contraire, si ce bit est à '1', alors la donnée est mémorisée et renvoyée en sortie.

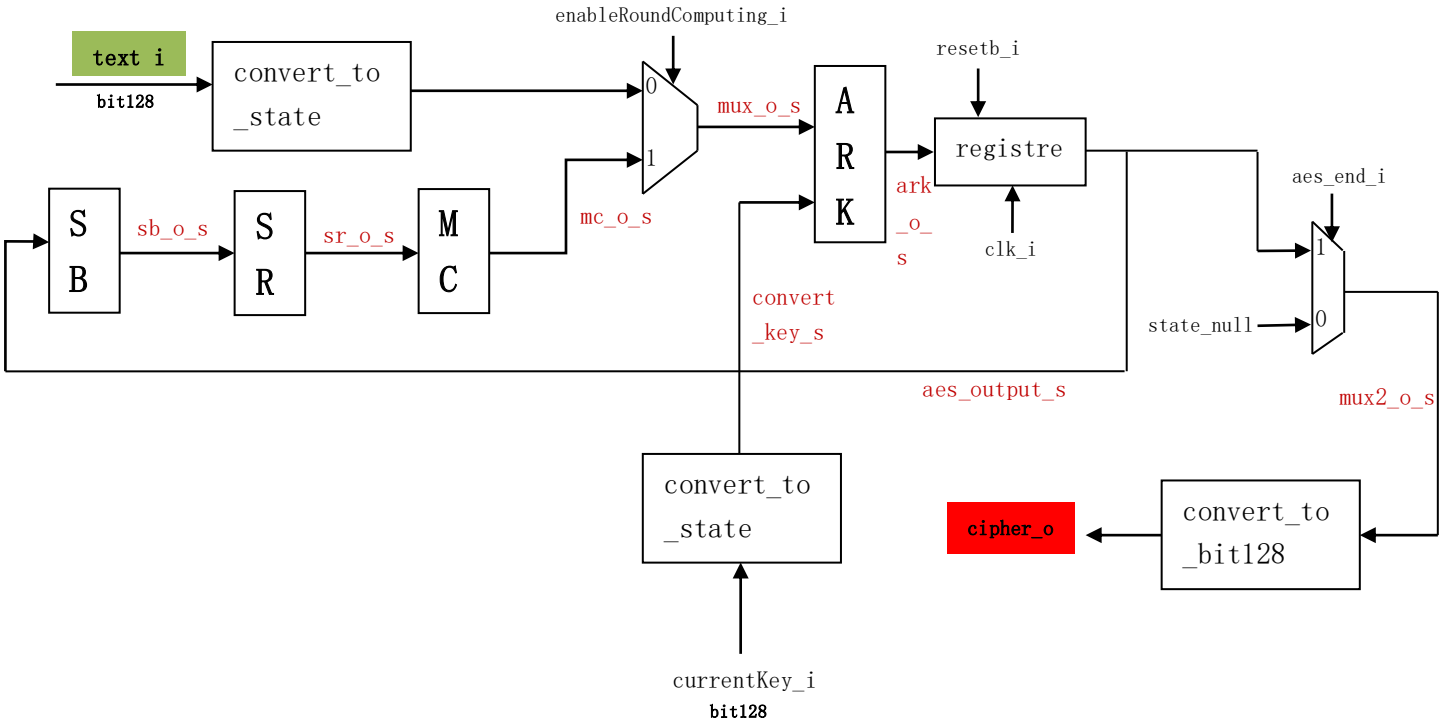


fig. 12 : schéma global représentant chacun des Rounds du chiffrement AES : entité AES_Round

Pour décrire l'entité AES_Round, on insère toutes les entités apparaissant sur le schéma précédent, et on les map les unes avec les autres à l'aide de signaux intermédiaires (représentés en rouge sur le schéma).

Enfin, un multiplexeur est introduit en sortie de l'AESround afin de mettre le texte chiffré en sortie (cipher_o) à zéro durant les 10 premiers Rounds et d'afficher enfin le résultat au dernier

Round. En effet, ce dernier multiplexeur rend le cassage du chiffrement plus complexe, car le fait d'avoir accès aux messages chiffrés intermédiaires peut aider un éventuel hacker à déchiffrer la clé.

Les résultats du testbench sont les suivants :

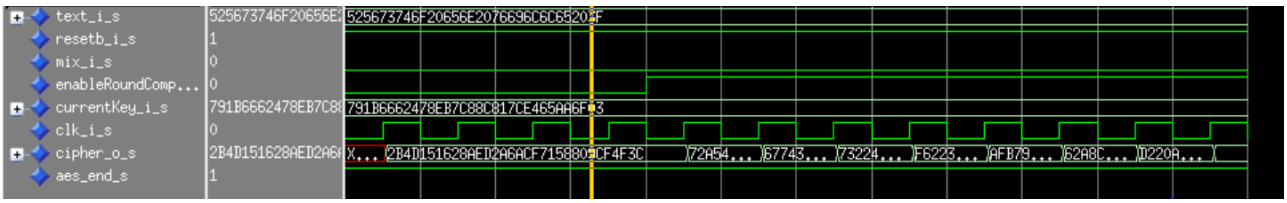


fig. 13 : résultats du testbench de l'AESround

Le `enableRoundComputing` est tout d'abord mis à zéro, afin de simuler le Round0, puis à 1 afin de simuler les Rounds suivants. Les textes chiffrés consécutifs ne correspondent pas à ceux du sujet du projet, étant donné que la clé initiale a été gardée tout au long de cette simulation.

Coversion des variables

Il est à noter que les données en entrée et en sortie de l'entité AES, comme décrit sur la figure 22 sont de type bit128. Cela signifie donc qu'il faut les convertir en type_state afin de pouvoir les exploiter dans les fonctions précédemment décrites, et inversement pour obtenir la sortie de l'entité AES en bit128. Pour ce faire, un calcul manuel a été fait au préalable sur papier, afin de retrouver une formule liant les indices du bit128 (allant de 0 à 127) et du type_state (allant de [0][0] à [3][3]).

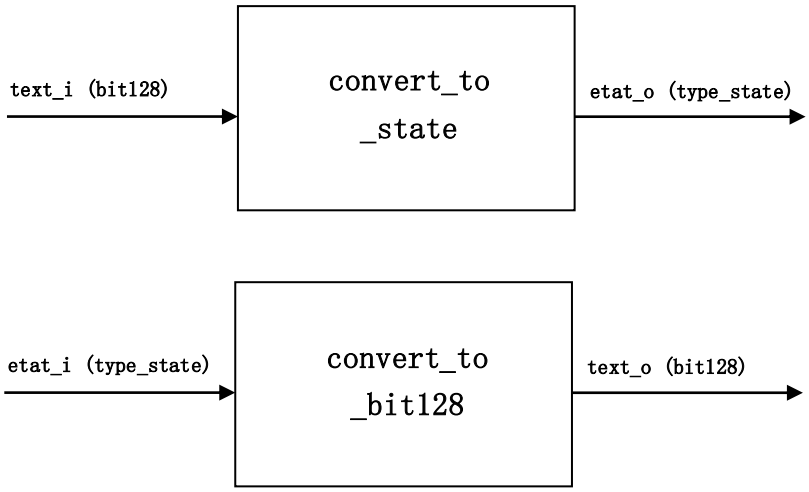


fig. 14 : schémas globaux des convertisseurs

Ces deux fonctions, inverses l'une de l'autre, jouent avec les indices des bits du bit128 (resp. type_state) afin de les transformer en type_state (resp. bit128).

Les résultats des testbenches relatifs à ces fonctions sont les suivants :

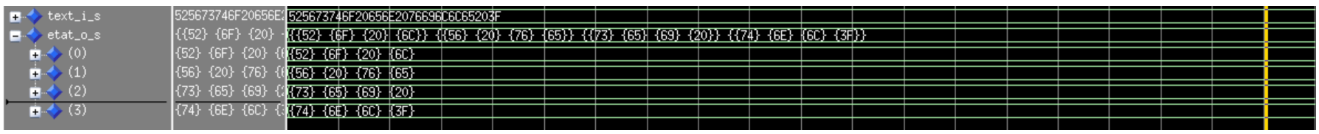


fig. 15 : testbench du convert_to_state

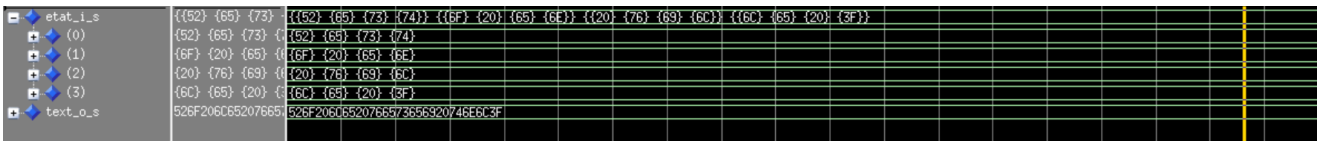


fig. 16 : testbench du convert_to_bit128

Une indexation en colonnes a été choisie pour ces fonctions a été choisie afin de faciliter le calcul.

Gestion des Rounds

i) Machine à états finis de Moore : le fichier FSM_moore.vhd

La machine de Moore permet de gérer et cadencer les différents Rounds du chiffrement AES. En effet, dans une machine de Moore, les sorties dépendent de l'état présent, et l'état futur est calculé à partir des entrées de l'entité ainsi que de l'état présent.

Ci-dessous un diagramme d'état représentant tous les états possibles du chiffrement AES :

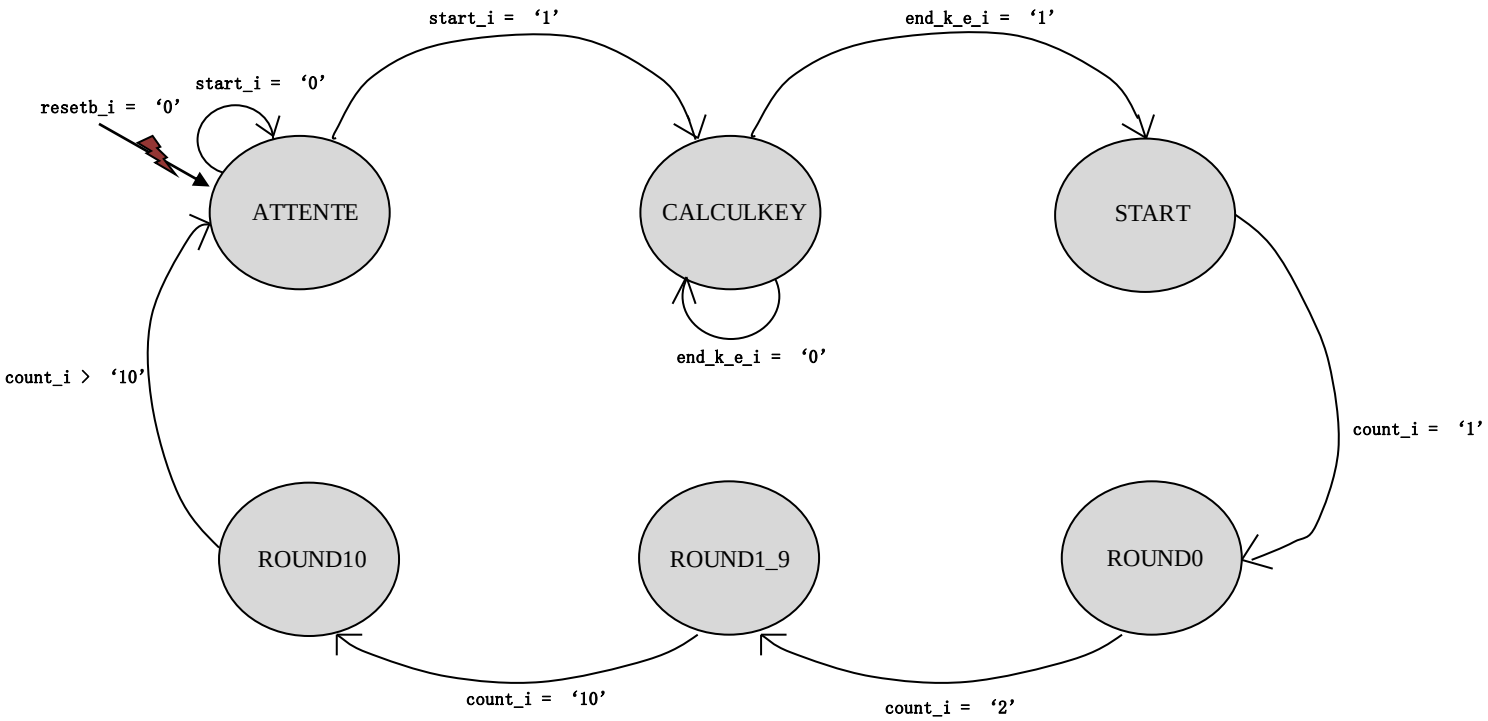


fig. 17 : diagramme d'états de la Machine de Moore du chiffrement AES

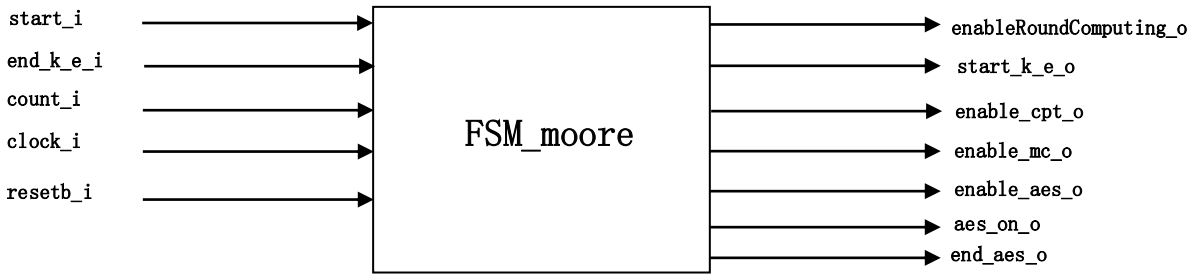
Légende :

: le `resetb_i` étant un signal asynchrone, on revient à l'état d'attente dès qu'il est activé, sans attente le coup d'horloge suivant.

: tant que `start_i` est nul, on reste à l'état d'attente.

On remarque bien la distinction faite entre le premier Round, les Rounds intermédiaires et le dernier. En effet, les Rounds 1 à 9 sont représentés par un seul et même état, puisqu'ils sont identiques. Le passage de chaque état i à l'état i+1 dépend des bits pris en compte sur le diagramme précédent.

L'entité représentant la machine de Moore a pour schéma global le schéma suivant :



La description de l'architecture de la machine de Moore est constituée de trois processus, deux combinatoires, et un séquentiel. Ces trois processus permettent de déterminer, à chaque front d'horloge, et à partir de l'état des entrées de l'entité, la valeur des entrées/sorties ainsi que les états futur et présent.

Les résultats du testbench de la machine de Moore sont les suivants :

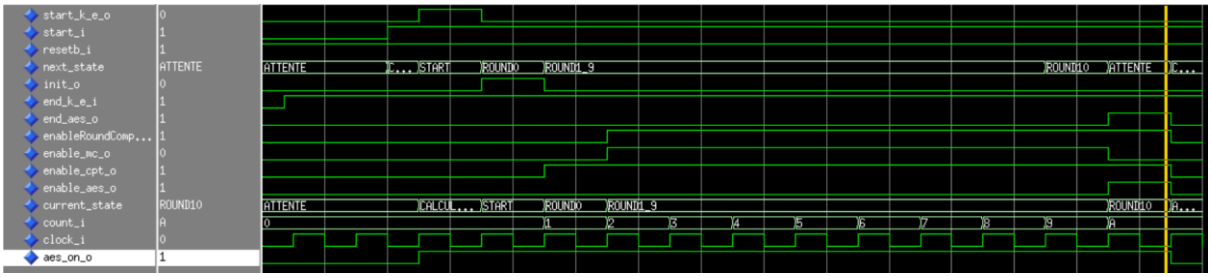


fig. 18 : simulation du Round0 (`enableRoundComputing = '0'`) avec `resetb_i` inactif

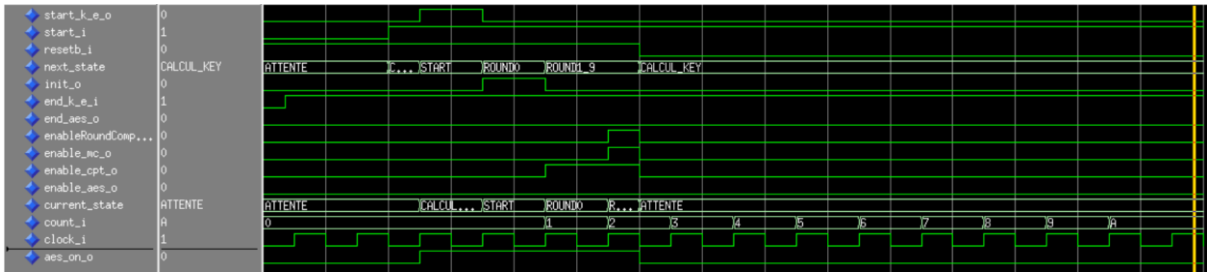


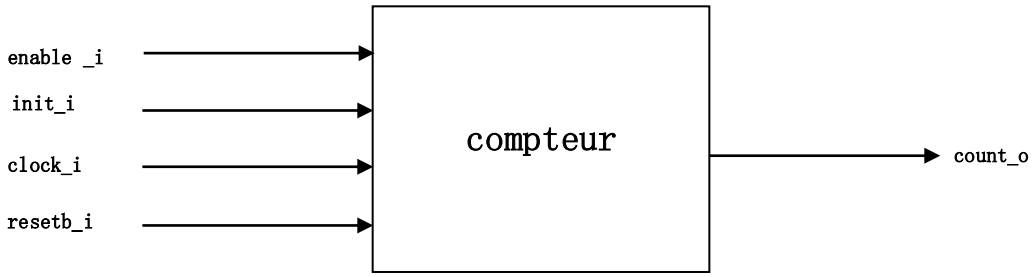
fig. 19 : simulation du Round0 (`enableRoundComputing = '0'`) avec `resetb_i` actif après qq ns

Les états correspondent bien aux attentes du diagramme. En effet, le port `start_k_e_o` est bien mis à '1' lors de l'état `CALCULKEY`, `init_o` est à '1', et l'état actuel entre en `ATTENTE` dès que le `resetb_i` est actif.

ii) Le compteur

Le compteur est une entité clé dans la description du chiffrement, car celui-ci permet de gérer la machine de Moore et toute la structure. En effet, celui-ci s'incrémente à la fin de chaque étape si son entrée `enable_i` est active et que l'entrée de reset `resetb_i` est inactive, et se remet à 0 si l'une des deux conditions n'est plus respectée, ou si `init_i` est active.

Le schéma général du compteur est le suivant :



La fonction est construite à l'aide d'un `process` sensible à `clock_i` et `resetb_i` qui incrémente le compteur si `enable_i` est actif lors d'un front montant d'horloge, ou bien le remet à zéro sinon.

Le choix d'un compteur de type `integer` a été effectué, afin de faciliter l'incrémentation au lieu d'avoir à passer par les fonctions de conversions de types (`to_integer()`, `unsigned()`…). Ensuite, la valeur du compteur est transmise à la machine d'états, et celle-ci transmet l'information à `KeyExpansion_I_O_table` sous forme de `bit4` par la suite.

Les résultats du testbench prouvent ce fonctionnement :

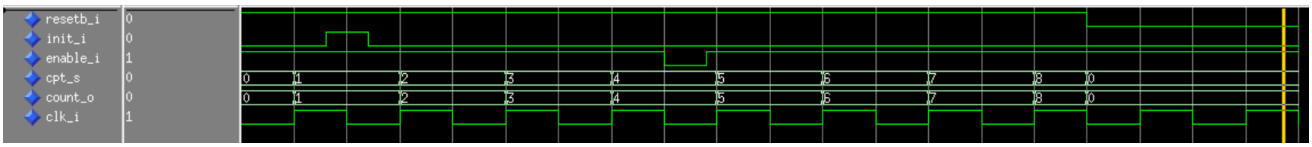


fig. 20 : résultats du testbench issu du compteur

Ces résultats sont effectivement ceux attendus. En effet, lorsque le bit de reset est inactif et que `enable_i` est actif, le compteur s'incrémente correctement, mais est remis à zéro une fois `resetb_i` mis à zéro.

De plus, le compteur est bien synchrone car même si enable_i est mis à zéro à un moment et remis à 1 juste avant le prochain coup d'horloge, le compteur continue de s'incrémenter.

Regroupement des entités dans le fichier « AES.vhd »

La dernière étape du chiffrement AES consiste en le regroupement de tous les composants globaux, à savoir la machine à états finis, le compteur, le générateur de clés ainsi que le bloc AESround décrit précédemment. Cette fonction est très simple, dans la mesure où il suffit d'y mapper les composants entre eux à l'aide de signaux en s'inspirant du schéma de la figure 22.

Le schéma général du bloc AES est le suivant :

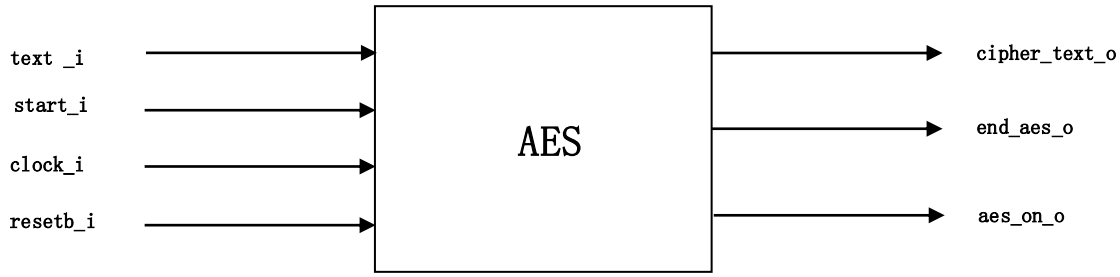


Fig. 21 : schéma global de l'AES Top Level

L'architecture choisie pour représenter le Top Level est schématisée dans la figure suivante :

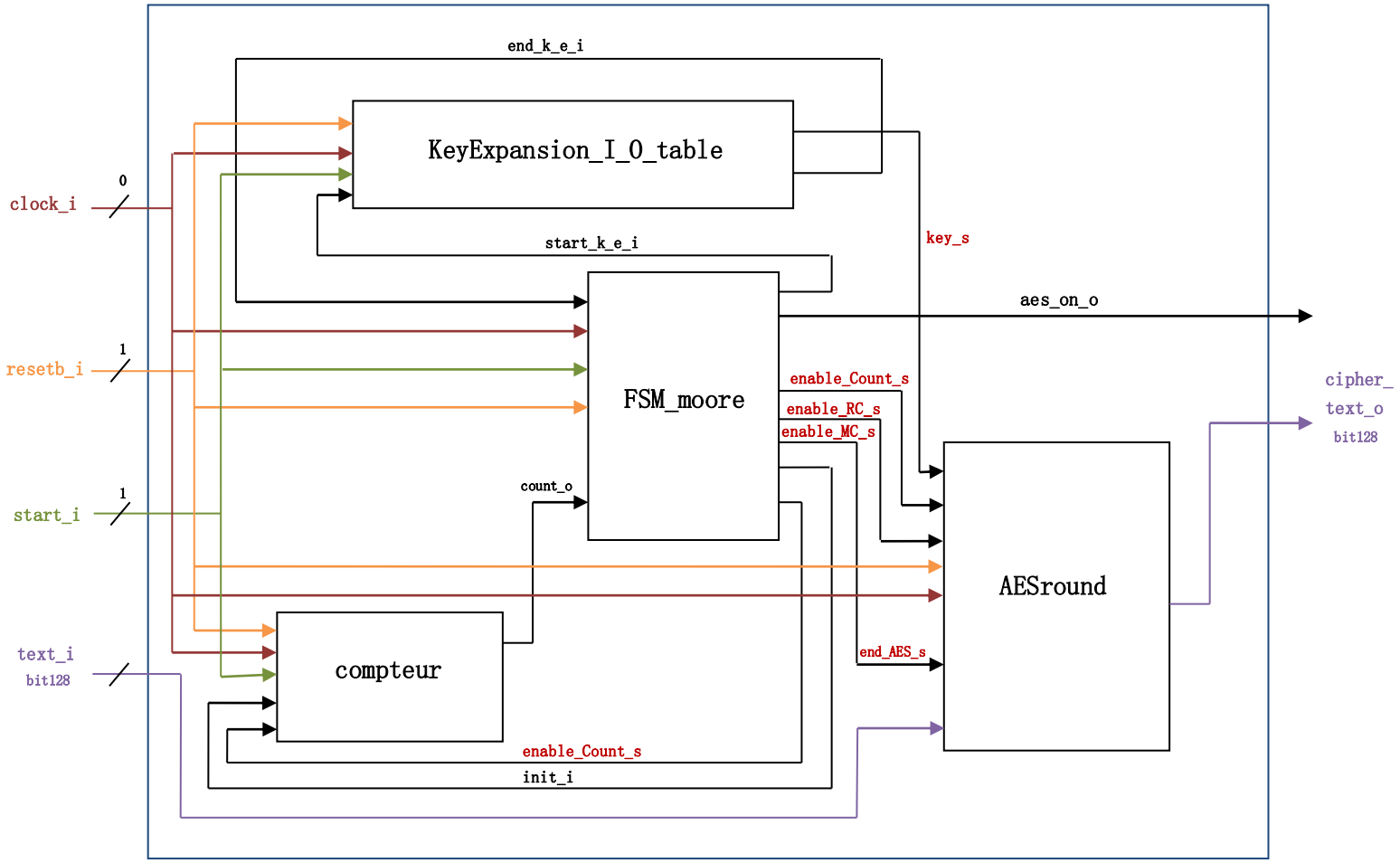


fig. 22 : schéma général de l'AES Top Level

On remarque ici que l'architecture du Top Level permet de synchroniser tous les composants du bloc afin de permettre à la machine de Moore de les contrôler. Un code couleur a été mis en place pour représenter les signaux entrants et contrôlant le dispositif (`clock_i`, `start_i`, `resetb_i`, `text_i` et `cipher_text_o`) afin d'obtenir un schéma plus épuré et faciliter la lecture.

Les résultats du testbench du Top Level sont les suivants :

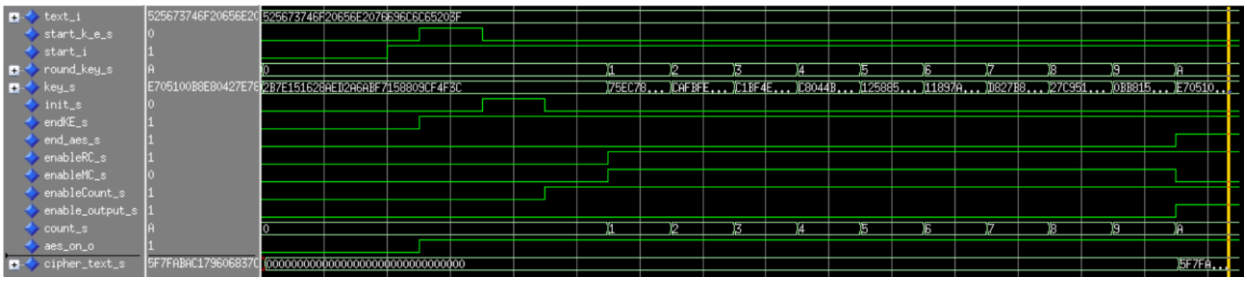


fig. 23 : résultats de la simulation du Top Level AES

On remarque sur la simulation que même si les clés sélectionnées sont correctes, le texte chiffré obtenu en sortie du dernier Round ne correspond pas à celui demandé sur le sujet. Cependant, une période de debug a mis en évidence que le problème ne vient pas des Rounds précédents (0 à 9). En effet, après avoir mis mon signal `end_aes_s` à '1' dans tous les états sur la machine de Moore afin de pouvoir observer les messages chiffrés intermédiaires, j'ai remarqué qu'ils étaient tous corrects, à part le dernier. L'hypothèse effectuée est que peut-être que le `mix_i` du MixColumns n'a pas été désactivé lors du dernier Round, malgré que cela soit le cas sur le programme, ou bien un problème de désynchronisation du compteur.

D'autre part, le fonctionnement global du Top Level fonctionne bien : les numéros de Rounds, les clés, les signaux d'initialisation et de fin…

Conclusion

La réalisation de ce projet a été très constructive pour moi, tout d'abord car elle constitue une première mise en contact avec les programmes de description matérielle, mais également car elle m'a permis de développer un projet du début à la fin, selon ma conception du sujet, contrairement aux autres projets enseignés à l'ISMIN pour lesquels nous avons dû constituer des binômes. Cela m'a non seulement permis d'être autonome dans mes choix d'architecture et de programmation, mais également de travailler avec régularité.

J'ai ensuite pu développer des compétences techniques, et particulièrement en programmation de description matérielle, le VHDL. J'ai également pu mieux comprendre un aspect de la cryptographie qui est la constitution d'un algorithme de chiffrement, et en suis ravie car c'est ce qui m'intéresse pour mon projet professionnel.

La découverte de l'outil ModelSim a également été intéressante, il a d'ailleurs été très utile pour le debug. En effet, la dernière semaine a été entièrement consacrée au debug de l'AESround et du Top Level, et l'observation de l'activité de chacun des signaux m'a beaucoup aidée à trouver mes erreurs.

De plus, malgré le fait que je n'ai pas pu afficher le bon message chiffré, je suis très satisfaite des compétences que j'ai appris à développer durant ce mois de projet et reste convaincue que du temps supplémentaire m'aurait permis de trouver l'erreur, grâce aux simulations de ModelSim.