ENSMSE ISMIN

G3

Projet de Conception d'un Système Numérique
Modélisation du déchiffrement AES en VHDL

Sophie Oms G3

December 16, 2025

Sommaire

Liste des figures

Liste des abréviations employées

FSM	Finite state machine
VHDL	Very High-Speed Integrated Circuits Hardware Description Language
AES	Advanced Encryption Standard

Chapter 1

Introduction

Ce projet avait pour objectif d'implémenter l'algorithme invaes, qui sert à déchiffrer un texte chiffré avec l'algorithme AES (Advanced Encryption Standard).

1.1 Principe du déchiffrement AES

Il s'agit d'une méthode de chiffrement symétrique, par blocs. Le message à déchiffrer en entrée e l'algorithme est un bloc de données sur 16 octets, donc 128 bits. L'algorithme AES utilise une clé secrète lors de la ronde initiale, puis cette clé est dérivée dans les rondes d'après. Lors du déchiffrement, on utilise donc les clés de rondes générées par la fonction KeyExpansion_table.

1.2 Contraintes de développement

Les blocs utilisés faisaient 128 bits en entrée et en sortie; la clé de chiffrement est également codée sur 128 bits, on récupère les sous-clés dans keyExpansion_table. L'algorithme fonctionne par rondes faisant appel aux quatre sous-fonctions suivantes :

1. invshiftrows, une fonction qui mélange les lignes du type_state qu'elle prend en entrée
2. invsubbytes, une fonction qui substitue chaque octet du type_state en entrée selon une table de substitution qui a été fournie par l'énoncé
3. invaddroundkey, une fonction qui ajoute la clé de ronde courante à l'état courant via un xor
4. invmixcolumns, une fonction qui applique une transformation colonne par colonne à un type_state (état). Cette transformation consiste en un produit matriciel.

Cependant, deux rondes sont particulières : la première ronde du déchiffrement (ronde 10 selon la numérotation de l'AES), qui ne comporte que la fonction addroundkey, et la dernière ronde du chiffrement (ronde 0 selon la numérotation de l'AES), qui ne comporte pas la fonction invmixcolumns.

On peut représenter l'algorithme InvAES de cette façon :

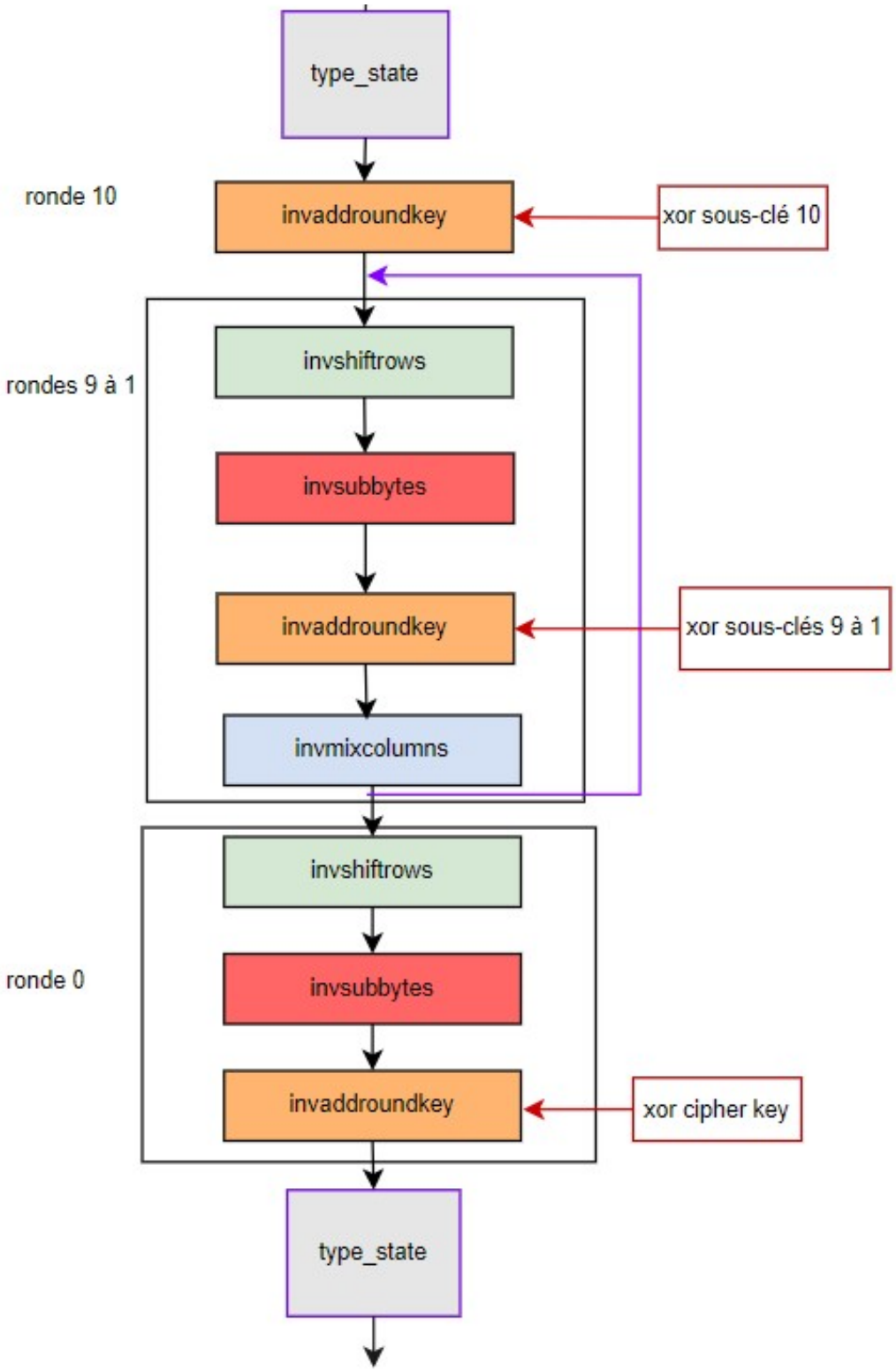


Figure 1.1: Organisation des rondes du déchiffrement invaes

Chapter 2

Implémentation de l'algorithme de déchiffrement

2.1 Décomposition globale

Le projet décompose le chiffrement en plusieurs parties. En effet, on doit appliquer les fonctions inverses des fonctions élémentaires qui composent l'algorithme AES (AddRoundKey, SubBytes, ShiftRows et MixColumns). Ainsi, on va créer des fonctions InvAddRoundKey, InvSubBytes, InvShiftRows et InvMixColumns pour le déchiffrement InvAES. De plus, l'algorithme de chiffrement se déroule en 11 rondes : la ronde initiale (qui exlut MixColumns), puis 9 rondes exécutant les 4 fonctions, et enfin l'ultime ronde 10 qui inclut seulement AddRoundKey.

Le déchiffrement s'exécute dans l'ordre inverse de l'algorithme de chiffrement : la ronde 10 avec uniquement InvAddROundKey, puis les 9 rondes exécutant les 4 fonctions, puis la ronde 0 qui exclut InvMixColumns.

1. Les sous-fonctions Invsubbytes, Invshiftrows, Invaddroundkey et Invmixcolumns
2. L'assemblage d'une ronde intermédiaire
3. La machine d'état qui orchestre l'ensemble de l'invAES
4. Le compteur et la table de sous-clés (déjà fournis en début de projet)
5. L'assemblage final

Ces différentes parties sont dans le dossier SRC du code associé.

2.2 Les sous-fonctions

2.2.1 Invsubbytes

Structure de Invsubbytes

Invsubbytes est une transformation linéaire, qui a pour rôle de faire la substitution inverse appliquée octet par octet lors du chiffrement AES, via la focntion subbytes. Elle fait appel à invsbox, une table de substitution fournie par le sujet. Cette transformation est appliquée à tous les octets de l'entrée de la fonction.

Par exemple, la lecture de la table révèle que l'élément s5,3 (ligne x=5, colonne y=3) du type_state en entrée sera substitué par ed (en hexadécimal) dans le type_state de sortie.

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

Figure 2.1: Table de substitution fournie par l'énoncé
fig:registre

J'ai choisi de représenter la invsbox par un tableau à double entrée, qui contient les valeurs de la table en hexadécimal. En effet, cette approche est intuitive et permet de facilement visualiser la table de substitution. Néanmoins, une implémentation sous la forme de multiplexeur aurait été possible. L'entrée et la sortie de la fonction sont de type bit8.

Testbench

On a deux signaux intermédiaires, data_i_s et data_o_s. On reprend les valeurs de la figure 11 de l'énoncé pour remplir data_i_s, par exemples la valeur 0x53 prise en exemple, qui correspond à 0xed dans la table.

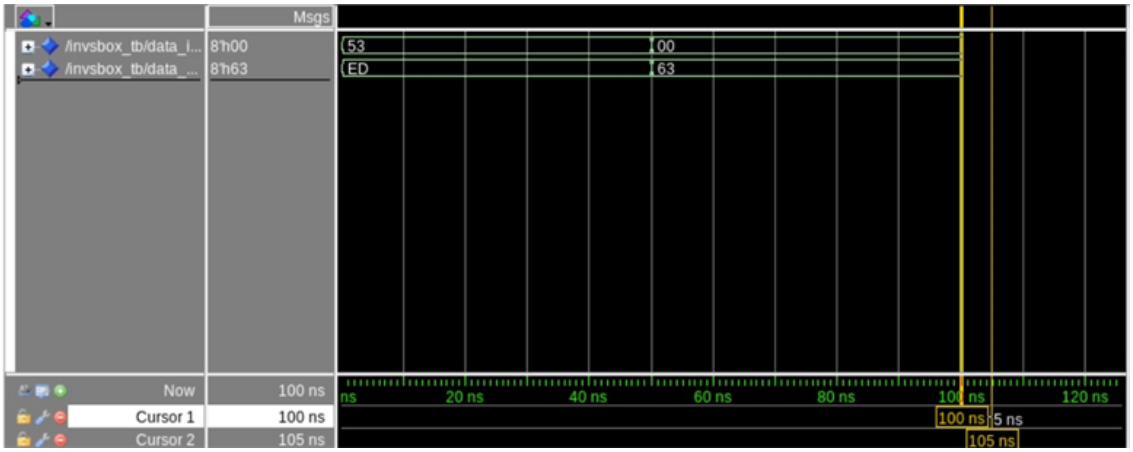


Figure 2.2: Chronogramme de invsbox réalisé sur ModelSim

Ainsi, les substitutions réalisées par invsbox sont bien celles attendues.

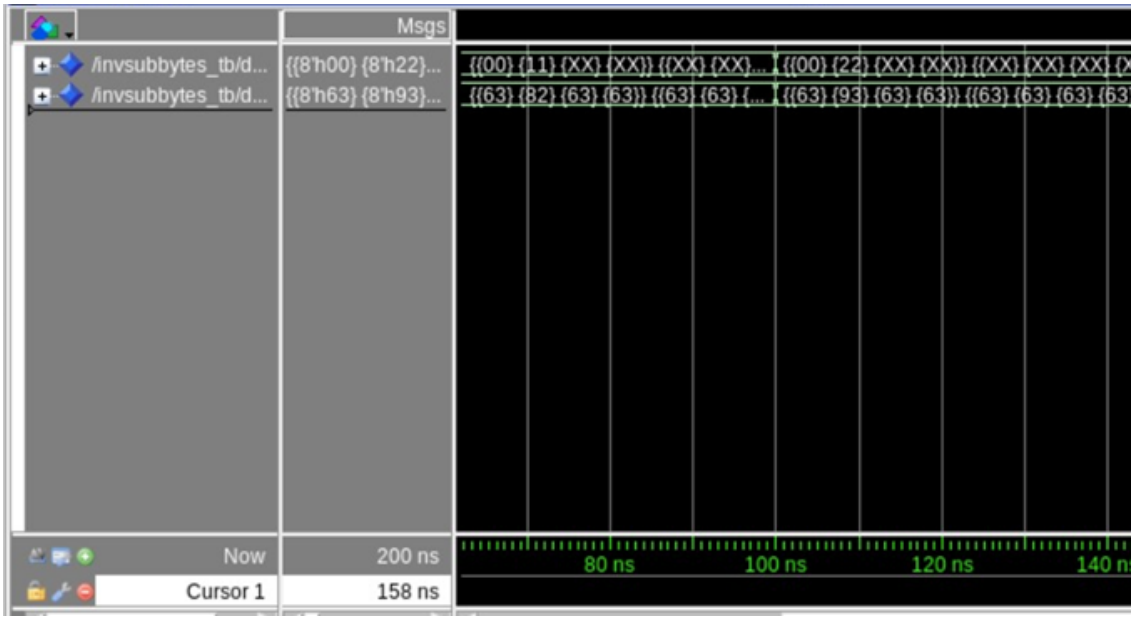


Figure 2.3: Chronogramme de invsubbytes réalisé sur Modelsim

2.2.2 Invshiftrows

Structure

Cette transformation consiste à mélanger les lignes, avec un décalage correspondant à l'indice de la ligne. Par exemple, la première ligne On utilise deux fois generate (boucle for qui fait une substitution ligne par ligne).

J'ai voulu déterminer une formule de généralisation, pour ne pas avoir à faire du ligne par ligne.

On remarque que la première ligne, qui a pour indice r=0, reste inchangée, donc elle subit un décalage de 0.

Pour la deuxième ligne (r=1), tous les octets sont décalés de r=1 colonne vers la droite (s[1][0] prend la place de s[1][0+1], s[1][1] prend la place de s[1][1+1], s[1][3] prend la place de s[1][(3+1)mod 4])

Pour la troisième ligne (r=2), tous les octets sont décalés de r=2 vers la droite (s[2][0] prend la place de s[2][0+2], s[2][1] prend la place de s[2][1+2], s[2][3] prend la place de s[2][(3+r)mod 4])

Pour la quatrième ligne (r=3), tous les octets sont décalés de r=3 vers la droite (s[3][0] prend la place de s[3][0+r] et s[3][3] prend la place de s[3][2]= s[3][(3+r) mod4].

On peut donc généraliser par récurrence immédiate avec la formule : s[r][c] prend la place de s[r][(c+r) mod 4] pour r variant de 0 à 4, avec c l'indice de la colonne.

Testbench

Pour vérifier le bon fonctionnement de la sous-fonction invshiftrows, j'ai pris des valeurs de l'énoncé et j'ai déterminé le type_state en sortie à la main.

Sur vsim, j'ai d'abord obtenu le chronogramme suivant :

Le testbench m'a donc permis de me rendre compte que pour les lignes d'indice impair, il fallait réadapter la formule déterminée précedemment. En effet, seules les octets associés aux lignes paires de data_o correspondaient au résultat attendu.

04	E0	48	28
00	01	02	03
81	19	D3	26
00	01	02	03

donne

04	E0	48	28
03	00	01	02
D3	26	81	19
01	02	03	00

Figure 2.4: Résultat attendu pour la transformation invshiftrows

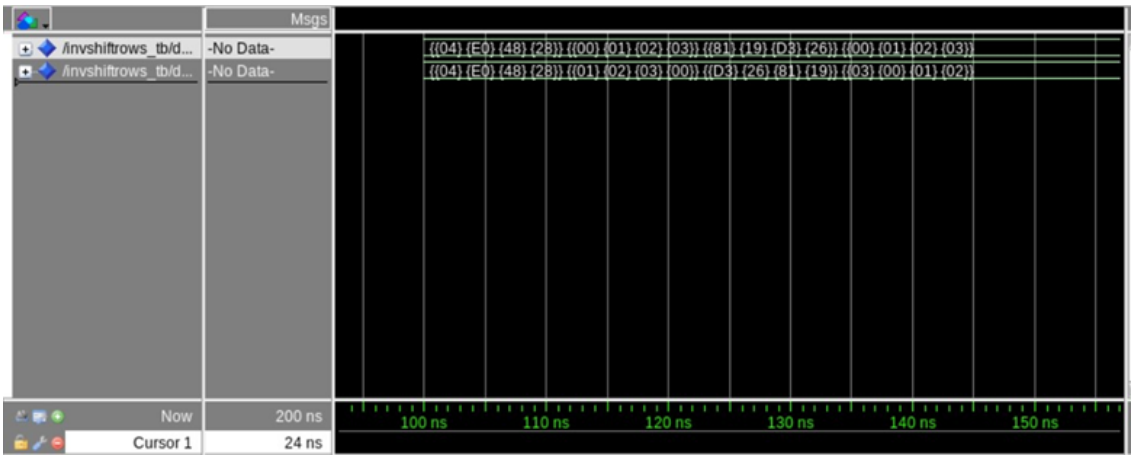


Figure 2.5: Premier chronogramme pour invshiftrows

Donc j'ai interverti data_o(1) et data_o(3), pour obtenir le chronogramme suivant :

Cette fois-ci, on retrouve bien le résultat attendu en sortie : décalage de 0 pour la première ligne, décalage de 1 pour la deuxième, décalage de 2 pour la troisième ligne et décalage de 3 pour la dernière ligne.

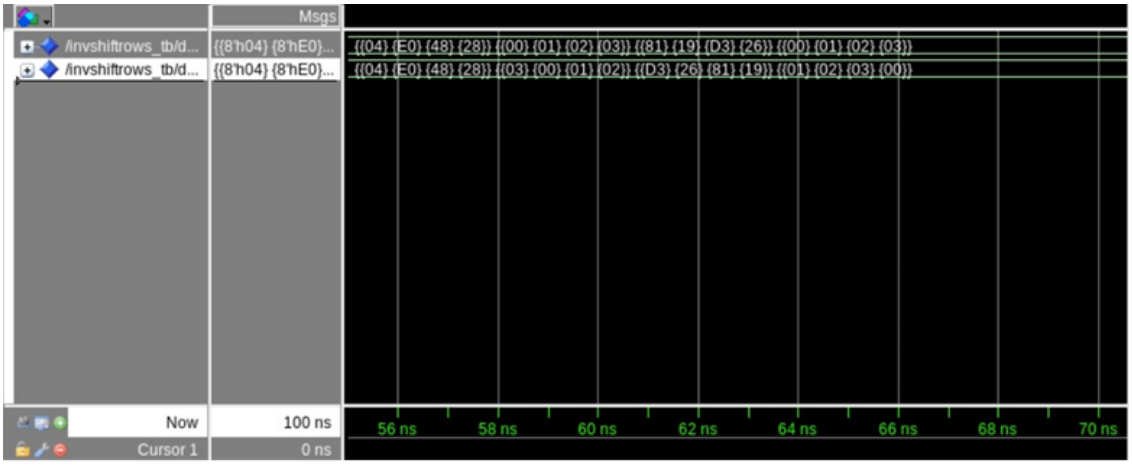


Figure 2.6: Chronogramme résultant pour invshiftrows

2.2.3 Invaddroundkey

Structure

Cette fonction lie le texte chiffré à la sous-clé de déchiffrement, en faisant un XOR (ou exclusif) entre les deux, octet par octet.

Ainsi, on utilise deux boucles for, et on réalise le xor des données en entrée et de la sous-clé associée pour chaque octet de même position.

Testbench

On teste la sous-fonction avec en entrée le bit128 en sortie de la ronde 10, et la sous-clé de la ronde 9 donnés dans l'énoncé. On les place respectivement dans les signaux intermédiaires data_ini_i_s et dans cle_ini_i_s, et on les convertit en type_state dans les signaux internes data_i_s et key_i_s. Pour cette conversion, on utilise la fonction bit128ToType_state, que j'ai définie dans conversion_definition_package. On vérifie qu'en sortie, on a bien à chaque fois le xor entre chaque octet du type_state de l'entrée data_i(première ligne sur le chronogramme) et l'octet correspondant de la sous-clé key_i.

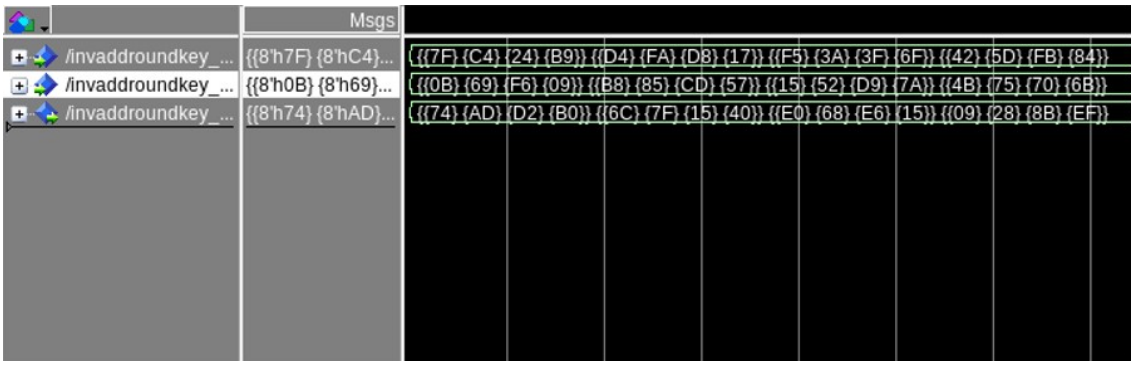


Figure 2.7: Chronogramme résultant pour invaddroundkey

Ainsi, le chronogramme résultant témoigne d'un bon fonctionnement de la sous-fonction invaddroundkey.

2.2.4 Invmixcolumns

Structure

Comme on doit faire des opérations similaires sur chaque colonne, on va instancier quatre fois une sous-fonction invmixcol pour chaque colonne du type_state en entrée. Cette sous-fonction requiert donc d'opérer sur les colonnes, alors que jusqu'ici on opérait sur les lignes. On utilise donc le type column_state, qui est défini dans le state_definition_package de THIRDPARTY.

La transformation invmixcolumns procède colonne par colonne, pour les quatre colonnes du type_state. En entrée d'invmixcolumns, il y a un type_state et un signald'activation en_i, de type std_logic. Ce dernier servira lors de la dernière ronde, qui utilise toutes les sous-fonctions hormis invmixcolumns. Tous les termes de la colonne en cours vont être traités en assemblage. De manière similaire au full_adder du cours, on va utiliser la transformation pour une colonne invmixcol quatre fois, à l'aide d'une boucle for.

La transformation linéaire est définie comme un produit matrciel, qui se fait avec les quatre octets d'une colonne de l'entrée et la matrice de transformation, dont chaque ligne et chaque colonne contient les mêmes éléments: 09, 0b, 0d, 0e. Or les colonnes sont traitées comme

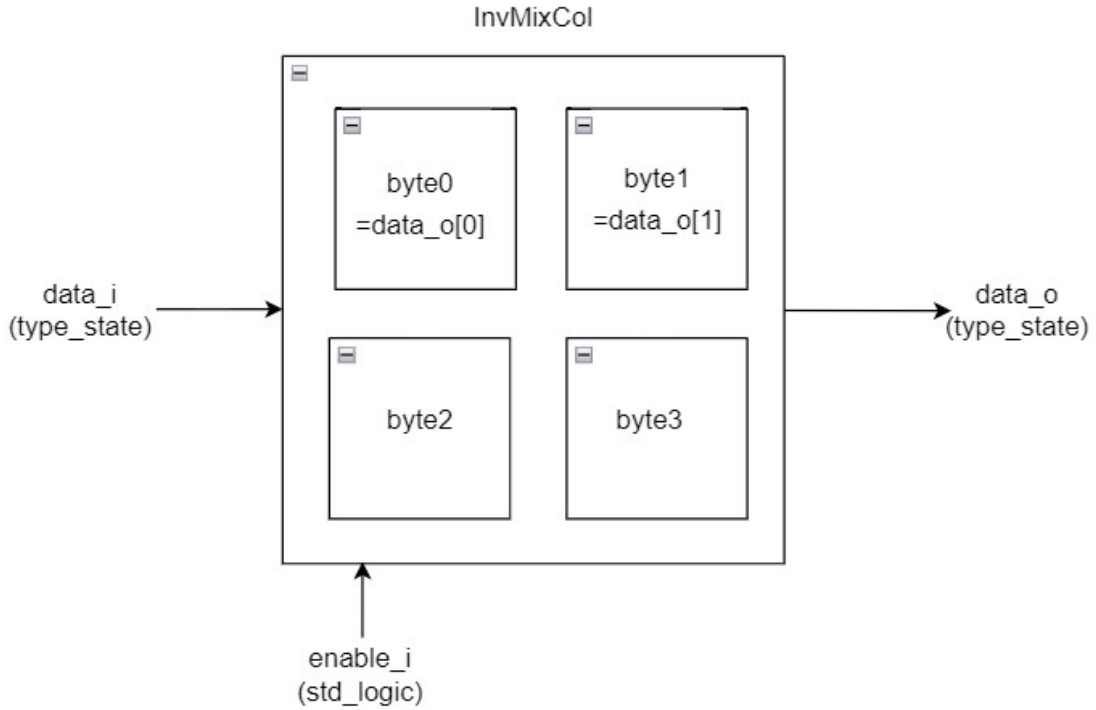


Figure 2.8: Illustration des 4 instanciations dans invmixcol

des polynômes dans le champs de gallois. Ainsi, on traite les produits des coefficients des matrices comme des additions réalisées dans le champs de Gallois (GF). Ces dernières sont calculées par la fonction XOR.

Multiplier par 2 dans GF revient à décaler les bits à gauche, puis xorer avec X"1B" si le bit de poids fort vaut 1.

Décomposition : 09=08+01=02x02x02 + 01 puis 0b= 09 + 02 puis 0d= 09 + 04 et 0e= 0d + 01 = 08 + 04 + 02

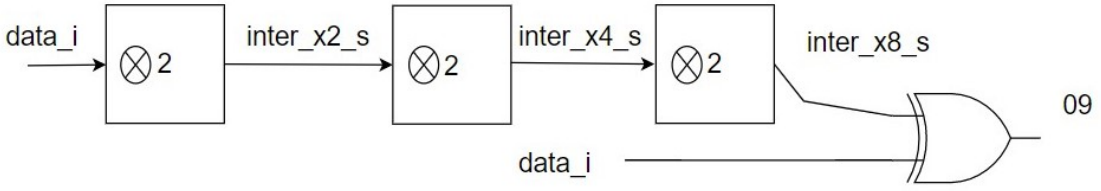


Figure 2.9: Utilisation de signaux intermédiaires pour la multiplication par 09

Ainsi, j'ai créé des signaux intermédiaires, pour faire une multiplication par 2, par 4 et par 8 dans GF. Pour Oe, on prend l'entrée data_i, qu'on xore avec inter_x4, résultat du produit de data_i par 2 dans GF.

Testbench

Pour vérifier, on utilise l'annexe de l'énoncé.

On décide d'instancier un composant Invmixcol ans l'architecture de InvMixColumns, avant le begin. Pour cela, on déclare ses variables d'entrée et de sortie Pour prendre une colonne du tableau en entrée et pas la ligne, on utilise les lignes de la matrice transposée. Puis on

fait le produit matriciel de chaque colonne de l'entrée data_i par la matrice de transformation. Puis dans le begin, pour les quatre colonnes (columns_state) en affectant les valeurs d'une Invmixcol à une colonne de InvMixColumn.

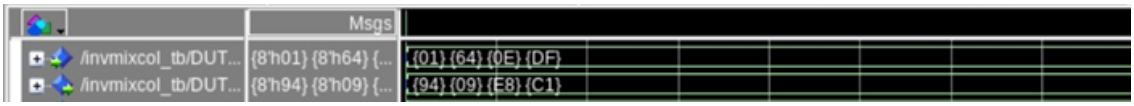
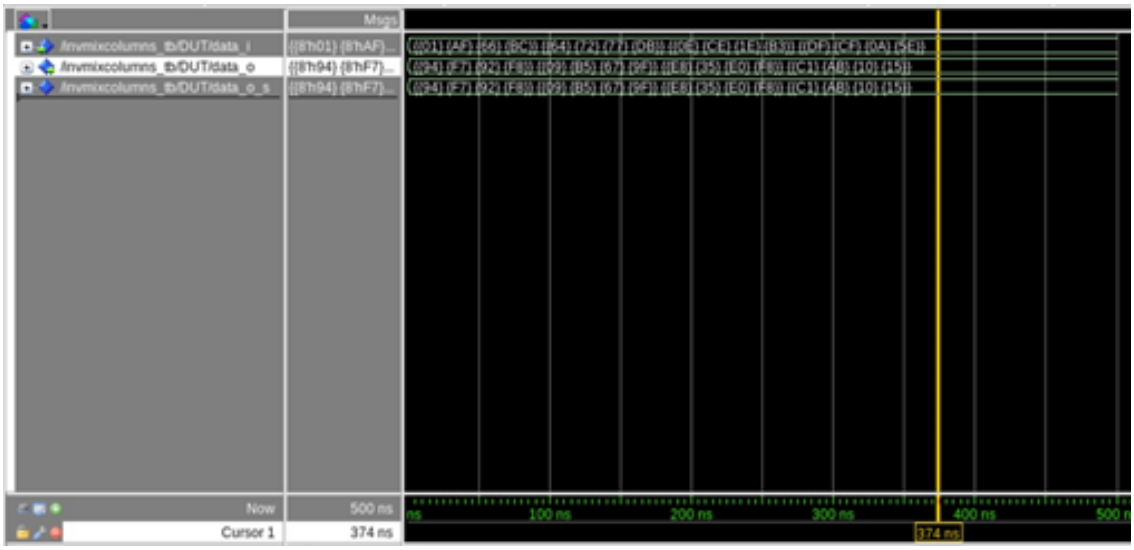


Figure 2.10: Chronogramme de invmixcol, obtenu sur ModelSim

Ainsi, la transformation sur une colonne se réalise bien. On en déduit que les opérations entre les signaux intermédiaires sont correctes.

Dans invmixcolumns_tb.vhd, on utilise invmixcol quatre fois. On obtient le chronogramme suivant :



Avec les valeurs de l'énoncé correspondant à la ronde 4, on constate qu'on obtient bien ce qui a été prévu en sortie de InvMixColumns.

Figure 2.11: Chronogramme de invmixcolumns, obtenu sur ModelSim

Chapter 3

Assemblage des composants

Il s'agit à présent de rassembler les sous-fonctions, qui ont été testés au préalable, dans une ronde.

3.1 Assemblage d'une ronde intermédiaire

3.1.1 Principe

On veut stabiliser le registre dans la ronde le temps de la ronde suivante. On casse la boucle combinatoire et on stocke le calcul dans le registre à chaque coup d'horloge.

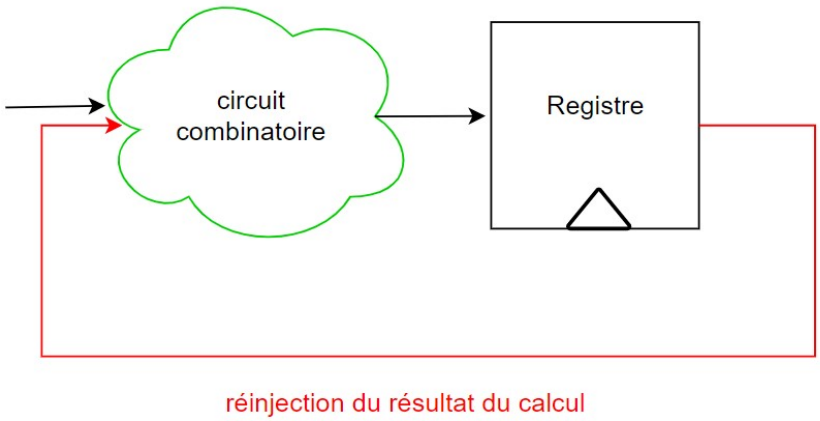


Figure 3.1: Circuit synchrone d'une ronde de invaes

3.1.2 Structure

Dans l'architecture, on fait appel aux quatre sous-fonctions, ainsi qu'au registre et au mux. On utilise deux multiplexeurs, MUX1 et MUX2. Le premier sert pour Invaddroundkey, et le second pour Invmixcolumns, afin de prendre en compte les rondes 0 et 10. Leurs signaux de sélection respectifs sont enable_Round_Computing et enable_Invmixcolumns. On sauvegarde dans le registre le résultat en sortie de InvMixColumns.

3.1.3 Testbench

Voici le chronogramme obtenu : la ronde est donc fonctionnelle.

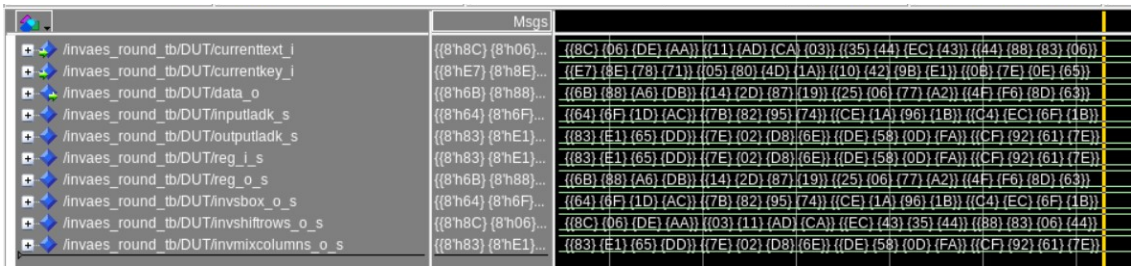


Figure 3.2: Chronogramme issu de invaes_round_tb.vhd via ModelSim

3.2 Machine d'état

La machine d'état a le rôle d'orchestrer l'organisation du déchiffrement InvAES

Tout d'abord, j'ai réalisé le graphe d'état suivant : Il s'agissait d'identifier les conditions de transition, ainsi que les valeurs que doivent prendre les sorties.

3.2.1 Registre

Le registre (registre.vhd) sert à stocker en mémoire un résultat intermédiaire.

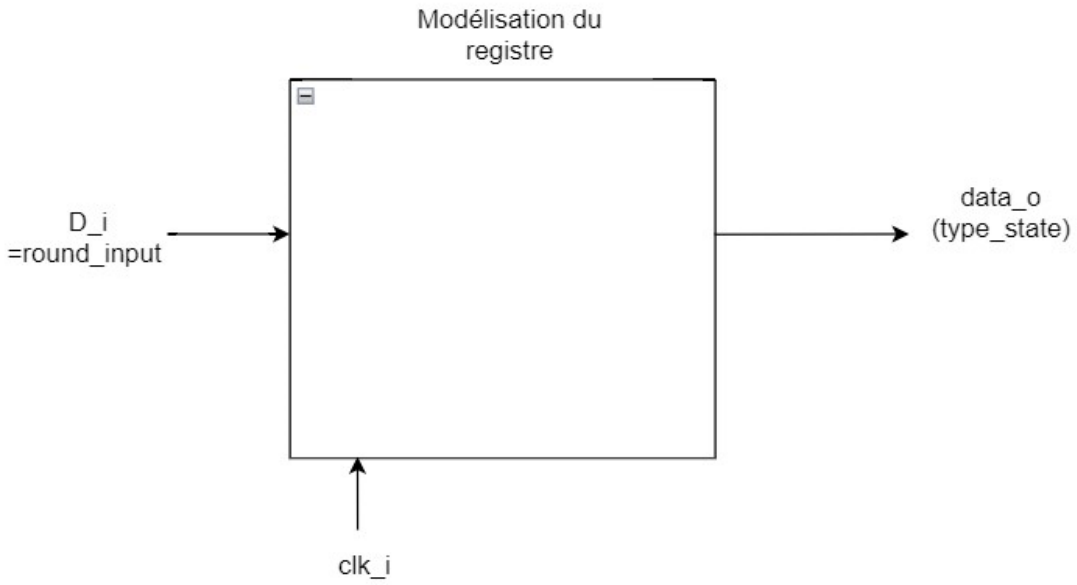


Figure 3.3: Modélisation d'un registre

3.2.2 Compteur

Principe

La machine d'état se sert du compteur (counter.vhd) pour savoir à quelle ronde elle se trouve. En effet, en VHDL, on ne peut modéliser du temps qu'avec un compteur. Ainsi, le compteur est initialisé à 10, puis lors du passage d'un état à un autre, il y a un coup d'horloge et le compteur décrémente de 1. Le signal reset_i sert à réinitialiser le compteur à 10 quand ce signal est à 1 (état haut).

Entre en_i et init_i, il faudrait avoir une priorité de ces signaux de contrôle. Le premier test sera fait sur le signal prioritaire.

Le compteur présente une subtilité à gérer: s'il n'a pas de condition d'arrêt, il atteint son maximum et repasse à zéro. De plus, il faut prendre en compte le type de la sortie. Par exemple, 16 est codé sur 5 bits et pas sur 4 bits. Ici, comme on veut un compteur allant de 10 à 0 (pour aller avec les rondes), on compte sur 4 bits. On définit néanmoins de 0 à 15 pour avoir une marge : cette précaution permet d'éviter certaines erreurs pendant la simulation. La condition d'arrêt peut être efficacement gérée par la machine d'état.

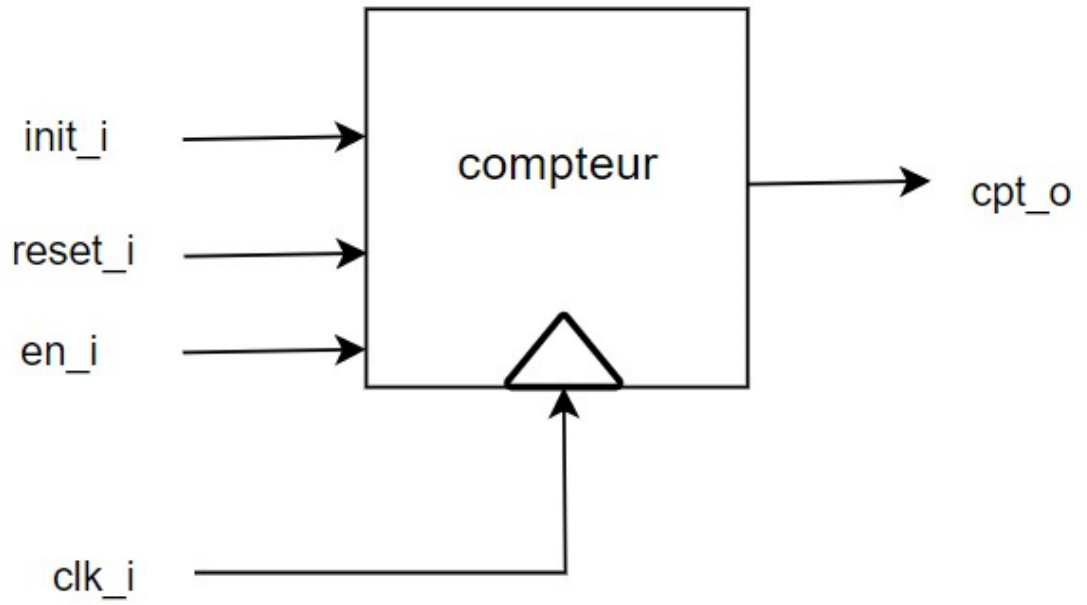


Figure 3.4: Illustration du compteur

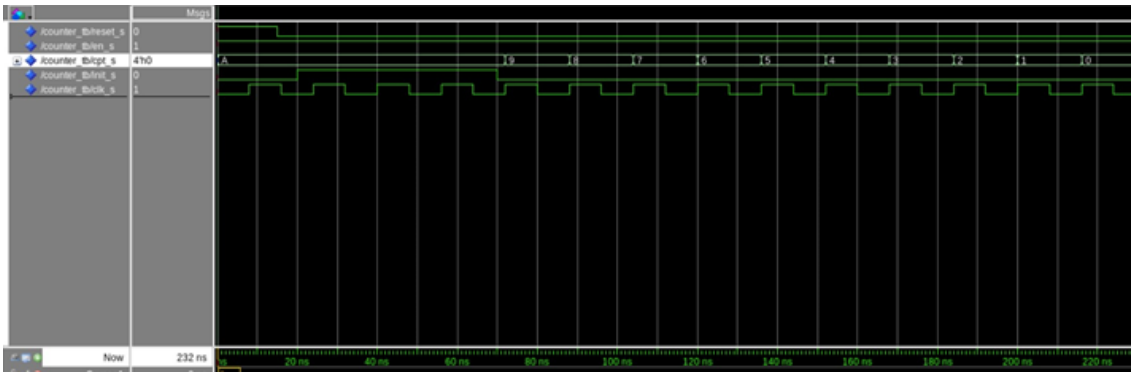


Figure 3.5: Chronogramme du compteur, obtenu via ModelSim

Testbench

On observe que le compteur va bien de 10 (A en hexadécimal) à 0 comme attendu. Ainsi, son fonctionnement est validé.

3.2.3 Modélisation de la machine d'état

3.2.4 Structure

D'abord, la structure choisie est un multiplexeur pour le reboutage. Cette structure permet l'affectation, grâce à des case...when... En VHDL, on a besoin de conditions lorsque le graphe contient au moins 2 transitions.

Le signal start_i sert à déterminer si on passe de l'état initial à l'état d'après ou pas.

Voici le graphe d'état :

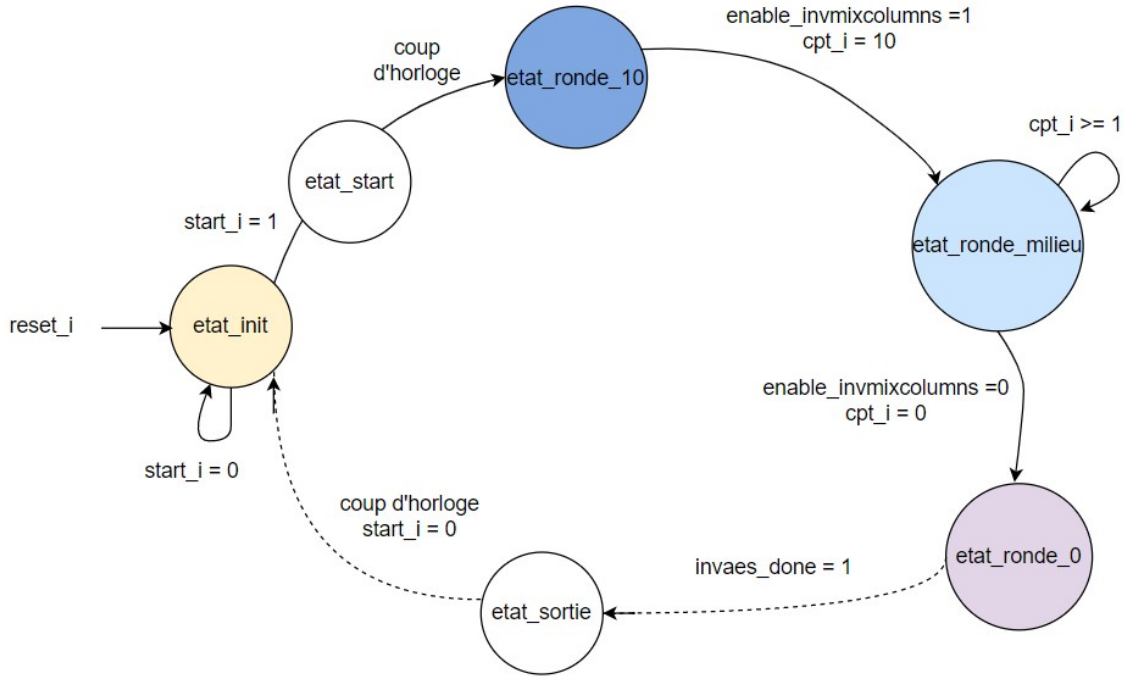


Figure 3.6: Graphe d'état

L'état initial sert à initialiser les signaux : ils sont tous à 0 sauf ceux qui reset et initialisent le compteur.

La ronde_10 est la première ronde du déchiffrement InvAES; elle présente la particularité de ne pas utiliser toutes les sous-fonctions de inv_aes. C'est pourquoi cet état met la valeur des signaux enable_round_computing_s et enable_invmixcolumns_s à 0, pour désactiver InvMixColumns et les fonctions avant InvAddRoundKey.

La ronde du milieu fait toutes les sous-fonctions, donc cet état passe enable_round_computing_s à 1 (pour activer Invsubbytes et Invshiftrows).

La ronde 0 passe enable_invmixcolumns_s à 0 et enable_round_computing_s à 1, pour faire toutes les sous-fonctions sauf InvMixColumns.

L'état de sortie met invaes_done à 1 pour dire que le déchiffrement a été fait, et passe enableCounter_s à 0 pour arrêter le compteur.

Enfin, l'état start permet de remettre invaes_done à 0 pour pouvoir relancer un déchiffrement.

J'ai ensuite organisé cette machine d'état de Moore (Moore Finite State Machine), en trois processus:

1. Le premier, seq0, est un processus séquentiel. Son rôle est de mettre à jour l'état présent en le remplaçant par l'état futur sur les fronts montants d'horloge. Ce processus sert aussi à faire l'initialisation, il s'occupe du reset de manière asynchrone.

2. Le deuxième, comb0, est un processus combinatoire. Il calcule l'état futur à partir des entrées et de l'état présent, nt, liste des entrées Structure à base de case...when

3. Et le troisième, comb1, est un processus combinatoire, qui calcule les sorties à partir des entrées et de l'état présent.

3.2.5 Testbench de la FSM

Après avoir testé le compteur décrémental séparément, je mets clk_i (signal d'horloge) et start_i à 1 pour un fornt montant d'horloge. Le compteur devrait aller de 10 (ronde 10) à 0 (ronde 0). Les états présents devraient être l'état initial, puis l'état de la ronde 10, puis l'état de ronde intermédiaire pour 9 rondes, puis l'état de ronde 0.

Cependant, le premier essai n'était pas concluant. En effet, l'état présent restait bloqué à l'état de ronde_milieu.

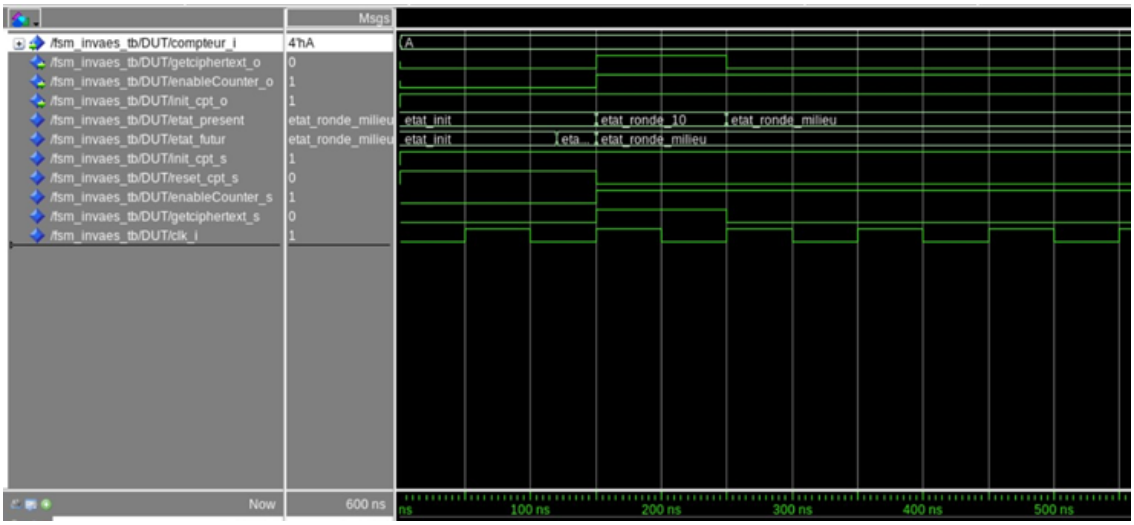


Figure 3.7: Premier chronogramme de la machine d'état

Le problème venait des conditions de passage d'état présent à état futur, le compteur étant bloqué à 10.

Après correction, l'enchaînement des états était correct, et j'ai obtenu le chronogramme suivant :

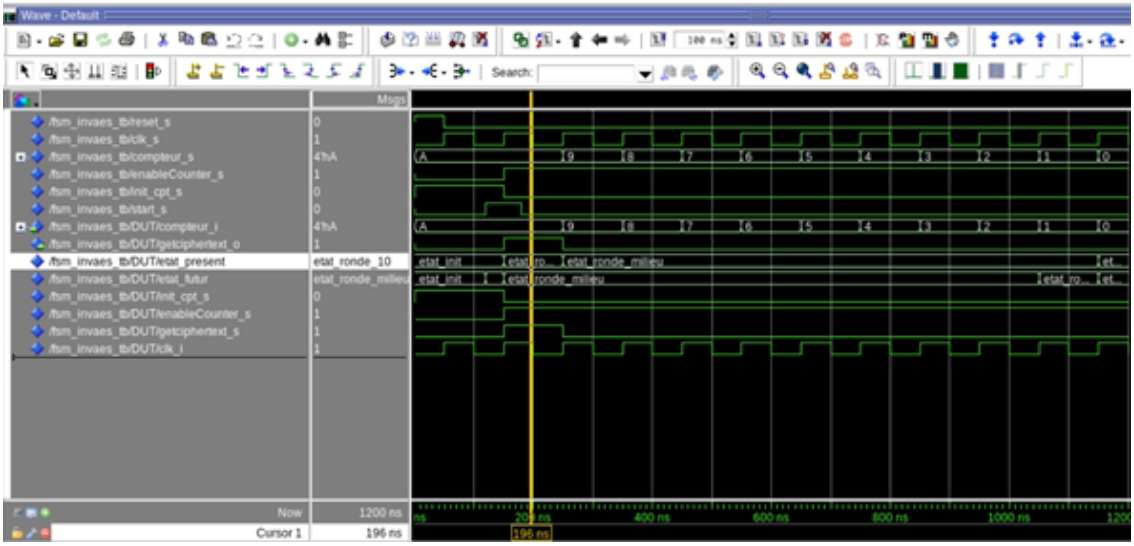


Figure 3.8: Chronogramme final de la machine d'état

3.3 La fonction inv_aes

3.3.1 Conversion des entrées et sorties

Pour convertir l'entrée de bit128 vers type_state et la sortie de type_state vers bit128, on passe par deux fonctions, qui ont aussi été appelées dans les testbenchs de invaddroundkey et de invmixcolumns. L'idée de modifier ou de reprendre state_definition_package a été donnée en cours, pour donner davantage de clarté au code que si les conversions avaient été faites directement dans les fichiers des composants. Ainsi, j'ai défini les deux fonctions dans le fichier conversion_definition_package. Ce dernier reprend la syntaxe de state_definition_package, et je l'ai également placé dans THIRDPARTY.

Le type_state se représente comme une matrice de 4 lignes et 4 colonnes, tandis que le bit128 se représente comme un tableau d'une seule ligne. Les valeurs adjacentes correspondent à celles d'une même colonne du type_state. C'est pourquoi on implémente le curseur "indice" pour la conversion : cette variable parcourt l'entrée de la fonction comme un curseur, qui prend les valeurs entières correspondantes. Par exemple, pour la conversion de type_state vers bit128, on utilise une double boucle for puis on incrémente la variable indice de 8 bits.

3.3.2 RTL_mux

Un multiplexeur sert à sélectionner un signal en entrée, grâce au signal de sélection selection_i. Ici, ce multiplexeur choisit les bonnes données en entrées pour inv_aes.

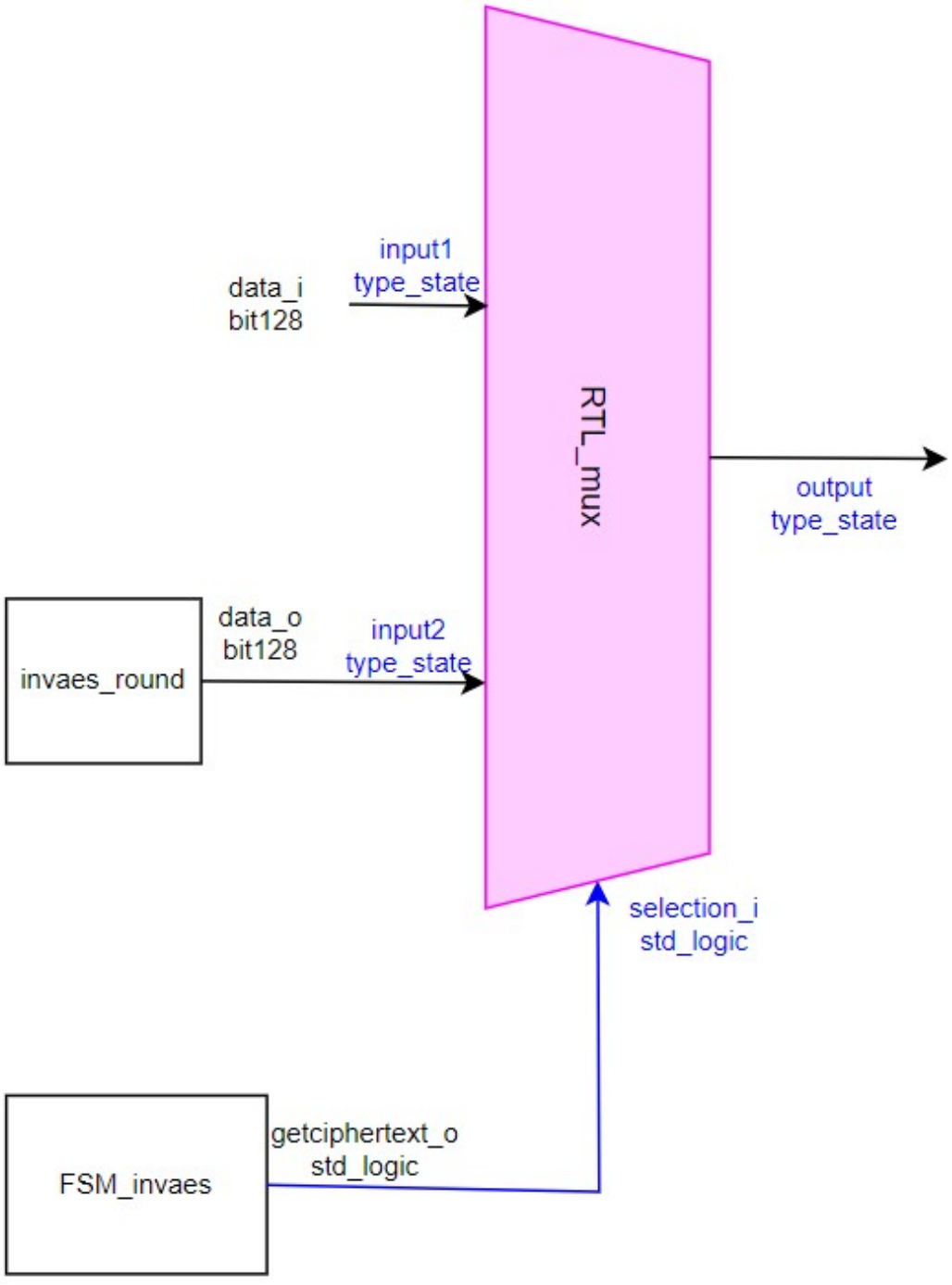


Figure 3.9: Multiplexeur pour choisir entre le message chiffré du début de l'algorithme et le résultat intermédiaire issu d'une ronde

3.3.3 Assemblage final

Deux multiplexeurs servent à gérer les spécificités de la première et de la dernière ronde. Pour la première ronde, on se sert du signal enable_round_computing du type std_logic, afin que l'entrée ne passe pas par invshiftrows et invsubbytes.

Pour la dernière ronde, on se sert du signal enable_invmixcolumns du type std_logic, afin que l'entrée ne passe pas par invmixcolumns.

C'est la machine d'état qui gère la valeur de ces signaux afin d'orchestrer les rondes.

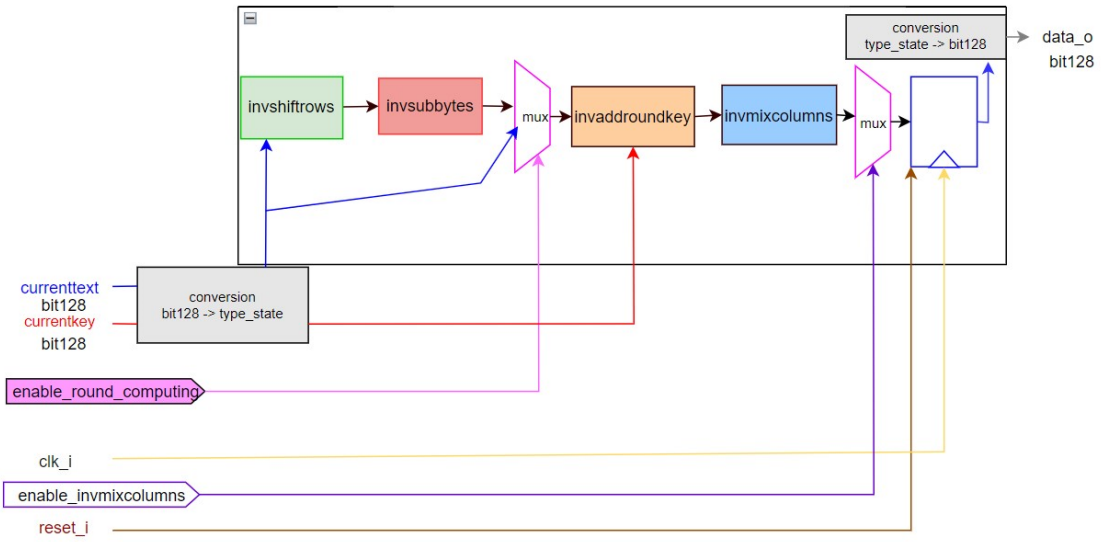


Figure 3.10: Assemblage des rondes spécifiques et des rondes intermédiaires

D'abord on définit l'entité inv_aes. Puis on importe les composants dont on a besoin : invaes_round pour une ronde, la machine d'état de Moore FSM_invaes, la table des clés de déchiffrement KeyExpansion_table.

Cependant, bien que mon code compilait, la simulation a été dysfonctionnelle.

Voici donc le coeur de mon code de inv_aes.vhd :

```
1
2     signal compteur_s : bit4;
3     signal expansion_key_s : type_state;
4     signal expansion_key_bit_s : bit128;
5     signal enable_round_computing_s, enable_invmixcolumns_s, start_s,
     init_cpt_s, invaes_done_s, enable_output_s, getciphertext_s : std_logic
     ;
6     signal enableCounter_s : std_logic;
7     signal invaes_stateoutput_s, data_o_s, currenttext_s : type_state;
8     signal invaes_o : type_state;
9
10 begin
11
12    Key_E : KeyExpansion_table
13    port map(
14        round_i => compteur_s,
15        expansion_key_o => expansion_key_bit_s
16    );
17
18    fsm : FSM_invaes
19    port map(
```

```
20        clk_i => clk_i,
21      --  start_i => start_i,
22        reset_i => reset_i,
23        compteur_i => compteur_s,
24        enable_round_computing_o => enable_round_computing_s,
25        enable_invmixcolumns_o => enable_invmixcolumns_s,
26        start_i => start_s,
27   enableCounter_o => enableCounter_s,
28        init_cpt_o => init_cpt_s,
29   getciphertext_o => getciphertext_s,
30   enable_output_o => enable_output_s,
31        invaes_done_o => invaes_done_s
32     );
33
34     ronde : invaes_round port map(
35     clk_i => clk_i,
36     currentkey_i => expansion_key_s,
37     enable_Invmixcolumns_i => enable_invmixcolumns_s,
38     enable_Round_Computing_i => enable_round_computing_s,
39     reset_i => reset_i,
40     currenttext_i => ciphertext_i,
41     data_o => invaes_o
42       );
43
44     --pour choisir entre le message a dechiffrer au debut de l'algo et le
       resultat intermediaire d'une ronde
45     mux_res : mux port map(
46           input1_i => data_o_s, --resultat intermediaire d'une ronde
47     input2_i => currenttext_s, --message a dechiffrer au debut
48     selection_i => getciphertext_s,
49     output_o => data_o_s
50     );
51
52     cpt : counter port map(
53     clk_i => clk_i,
54     reset_i => reset_i,
55     en_i => start_s,
56     init_i => init_cpt_s,
57     cpt_o => compteur_s
58       );
59
60     expansion_key_s <= bit128ToType_state(expansion_key_bit_s);
61     invaes_done_o <= invaes_done_s;
62     text_o <= data_o_s;
```

Listing 3.1: Code snippet of inv_aes.vhd

Chapter 4

Resultats

4.1 Testbench de inv_aes

J'ai modifié le fichier compile_src_model.txt mis à disposition sur campus, afin de compiler tous les fichiers sources vhdl de THIRDPARTY, RTL et BENCH. En tapant dans le terminal "./source compile_src.txt", on compile les fichiers voulus. On peut également lancer une simulation, en décommentant la ligne voulue qui commence par "vsim".

Le résultat de la simulation de inv_aes n'est pas concluant. J'ai d'abord obtenu le chronogramme suivant :

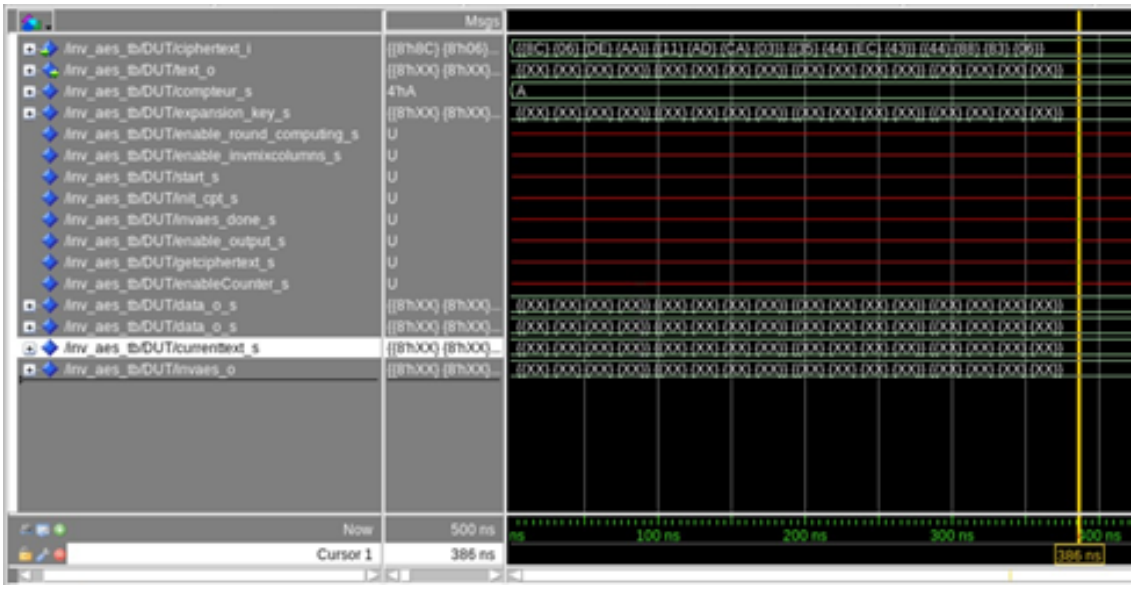


Figure 4.1: Chronogramme issu de inv_aes

On observe des 'X', qui correspondent à un état indéterminé. Cela signifie que la fonction affecte un '0' et un '1' en même temps aux signaux de l'assemblage final. La console de ModelSim a indiqué que registre.vhd et KeyExpansion_table étaient "not bound". J'ai donc modifié les lignes en lien avec ces composants, et j'ai enlevé la configuration de inv_aes_tb.vhd.

J'ai alors rencontré des problèmes de type entre type_state et bit128 pour les signaux de KeyExpansion_table.

Une fois les erreurs corrigées, le fichier compilait, mais la simaulation ModelSim rencontrait l'erreur suivante :

Ainsi, j'en ai déduit que le dysfonctionnement venait de la gestion de KeyExpansion_table.

```
VSIM 4> run
# Cannot continue because of fatal error.
# HDL call sequence:
# Stopped at ./SRC/RTL/KeyExpansion_table.vhd 28 Block Key_E
#
```

Figure 4.2: Message d'erreur de la console de ModelSim

4.2 Difficultés et voies d'améliorations

Voici les points bloquants majeurs de mon projet :

- Conversions : il m'a semblé ardu de gérer l'insertion des fonctions de conversion dans inv_aes, et j'ai fait face à des problèmes de type pendant un long moment.
- Utilisation de multiplexeurs : je n'étais pas habituée à en utiliser, et j'ai eu des difficultés à gérer les signaux pour l'assemblage final.
- Utilisation des modules fournis avec l'énoncé : je n'ai pas bien réussi à utiliser KeyExpansion_table.

Ainsi, j'ai encore des progrès à faire, mais les problèmes auquelles j'ai été confrontée m'ont permis une réflexion poussée sur le langage VHDL et sur le sujet du projet.

Chapter 5

Conclusion

Ce projet permet une vision globale de la conception et de l'implémentation d'un algorithme de chiffrement. Malgré les difficultés rencontrées lors de la réalisation, j'ai appris de nombreuses subtilités du langage VHDL, ainsi que la mnaière de configurer un environnement de travail. Finalement, le projet m'a enseigné la syntaxe et les conventions de ce langage de manière concrète. En outre, ce projet m'a permis de maîtriser de nouveaux outils : le serveur distant tallinn, pour la conception d'un système numérique, et ModelSim de mentorgraphics, pourles simulations.

En second lieu, l'algorithme de déchiffrement AES avait été présenté en cours de cryptographie; ainsi, le projet était d'autant plus intéressant qu'il permettait d'adopter une approche transversale et pluridisciplinaire de notre cursus en cycle ISMIN.

Enfin, le projet m'a entraînée à être méthodique dans la conception d'un système numérique, notamment grâce au découpage des séances, à la vérification systématique des composants via les testbenchs, et à la schématisation des éléments essentiels.