



Programmation et conception de systèmes numériques

Modélisation VHDL de l'algorithme de déchiffrement AES 128bits

Alexandre LADRIERE

9 Décembre 2018

Table des matières

1	Introduction	4
2	Principe de l'algorithme de déchiffrement	4
3	Implémentation des différents composants	6
3.1	InvSBox	6
3.1.1	Principe	6
3.1.2	Implémentation	7
3.1.3	Tests et résultats	7
3.2	invSubBytes	8
3.2.1	Principe	8
3.2.2	Implémentation	8
3.2.3	Tests et résultats	8
3.3	InvAddRoundKey	9
3.3.1	Principe	9
3.3.2	Implémentation	10
3.3.3	Tests et résultats	10
3.4	InvShiftRows	11
3.4.1	Principe	11
3.4.2	Implémentation	12
3.4.3	Tests et résultats	12
3.5	InvMixColumns	13
3.5.1	Principe	13
3.5.2	Implémentation	13
3.5.3	Tests et résultats	15
3.6	Machine d'état	16
3.6.1	Principe	16
3.6.2	Implémentation	16
3.6.3	Tests et résultats	17
3.7	Décompteur	18
3.7.1	Principe	18
3.7.2	Implémentation	18
3.7.3	Tests et résultats	19
3.8	Registre	19
3.8.1	Principe	19
3.8.2	Implémentation	19

3.8.3	Tests et résultats	20
3.9	InvAESRound	20
3.9.1	Principe	20
3.9.2	Implémentation	20
3.9.3	Tests et résultats	22
4	Implémentation globale de InvAES	23
4.1	Principe	23
4.2	Implémentation	23
4.3	Tests et résultats	23
5	Conclusion	25

1 Introduction

L'Advanced Encryption Standard (ci-après AES) est, comme son nom l'indique, un standard de cryptage symétrique remplaçant le Data Encryption Standard (DES) qui était devenu inefficace notamment face aux attaques par force brute. Il a été instigué par le National Institute of Standards and Technology (NIST) le 2 janvier 1997. L'algorithme Rijindael a été proposé comme norme AES en octobre 2000, puis publié par le NIST dans la norme FIPS PUB 197 le 26 novembre 2001. C'est un algorithme de chiffrement par blocs supportant différentes combinaisons de longueurs de clés et longueurs de blocs.

On distingue trois versions d'AES : AES-128, AES-192 et AES-256. L'algorithme opère sur des blocs de 128 bits qu'il transforme en blocs cryptés de 128 bits à l'aide d'une clé de 128 bits (AES-128), 192 bits (AES-192) ou 256 bits (AES-256). Cela nécessite respectivement 10, 12 ou 14 tours (aussi appelés round). Chaque tour, à l'exception du dernier, comporte les mêmes étapes subdivisées en 4 transformations :

- *SubBytes*
- *AddRoundKey*
- *ShiftRows*
- *MixColumns*

Une fonction supplémentaire est nécessaire pour la génération des clés des rounds. Il s'agit de la fonction *AddRoundKey*.

Le but de ce projet était d'implémenter, en langage VHDL, l'algorithme de déchiffrement AES-128.

2 Principe de l'algorithme de déchiffrement

L'algorithme de déchiffrement est assez semblable à l'algorithme de chiffrement. Le message chiffré va subir plusieurs transformations par blocs au cours de plusieurs round (11 en tout), selon les transformations suivantes :

- *InvSubBytes*
- *InvAddRoundKey*
- *InvShiftRows*
- *InvMixColumns*

Ces blocs font bien entendu, comme leur nom l'indique, les opérations inverses des différents blocs de chiffrement. Ces blocs seront décrit de manière détaillée dans la suite de ce rapport.

De manière générale, on peut représenter le déchiffrement AES 128bits par l'algorithme suivant :

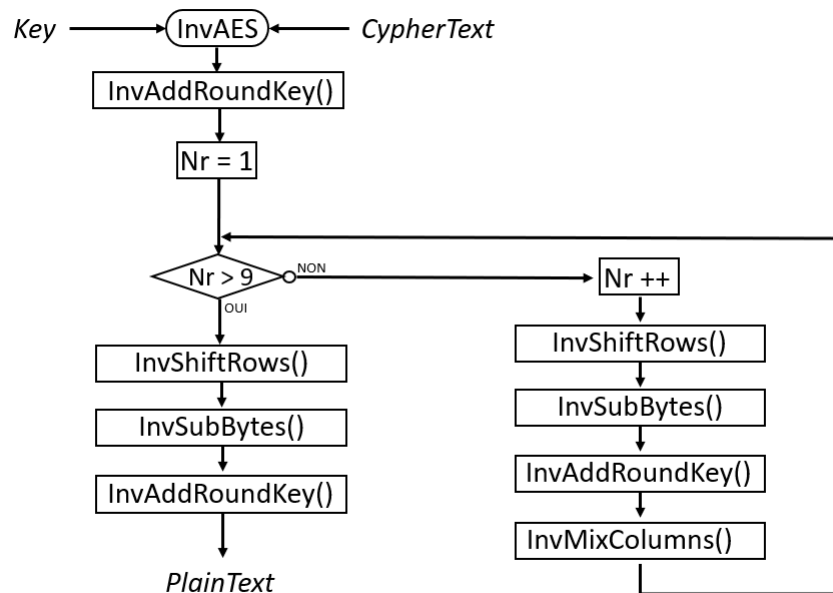


FIGURE 1 – Algorithme de InvAES

Ce schéma peut se traduire par le pseudo-code suivant (fourni dans la FIPS 197) :

```

InvCipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
  byte state[4,Nb]

  state = in

  AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1]) // See Sec. 5.1.4

  for round = Nr-1 step -1 downto 1
    InvShiftRows(state) // See Sec. 5.3.1
    InvSubBytes(state) // See Sec. 5.3.2
    AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
    InvMixColumns(state) // See Sec. 5.3.3
  end for

  InvShiftRows(state)
  InvSubBytes(state)
  AddRoundKey(state, w[0, Nb-1])

  out = state
end
  
```

FIGURE 2 – Pseudo-code de InvAES (FIPS 197)

3 Implémentation des différents composants

Avant toute chose, il est important de noter qu'une librairie regroupant tous les différents types dont nous pourrions avoir besoin nous est fournie dans le fichier

state_definition_package.vhd :

```
package state_definition_package is
    subtype bit4 is std_logic_vector(3 downto 0);
    subtype bit8 is std_logic_vector(7 downto 0);
    subtype bit16 is std_logic_vector(15 downto 0);
    subtype bit32 is std_logic_vector(31 downto 0);
    subtype bit128 is std_logic_vector(127 downto 0);
    type row_state is array(0 to 3) of bit8;
    type column_state is array(0 to 3) of bit8;
    type type_state is array(0 to 3) of row_state;
    type type_shift is array(0 to 3) of integer range 0 to 3;
    type type_temp is array(0 to 3) of bit8;
    type row_key is array(0 to 3) of bit8;
    type type_key is array(0 to 3) of row_key;
    type reg_w is array(0 to 3) of bit32;
    type type_rcon is array(0 to 10) of bit8;
    constant Rcon : type_rcon := (X"01", X"02", X"04", X"08",
        X"10", X"20", X"40", X"80", X"1b", X"36", X"00");
    function "xor" ( L,R: column_state ) return column_state;
end state_definition_package;
```

Par exemple, *bit8* représente un octet, *column_state* une colonne de 4 octets, *type_state* une matrice 4x4 d'octets, etc. Certains de ces types ont été utilisés lors de l'implémentation, et seront utilisés dans la suite de ce rapport.

Tous les tests ont été réalisés à partir des décompositions des étapes de déchiffrement du message original, fournies en fin de sujet du projet. Il faut faire attention au type des données. En effet, dans le sujet les valeurs sont données colonnes par colonnes (4 octets de la première colonne, puis 4 octets de la deuxième colonne, etc), alors que *type_state* est un tableau de ligne (cette confusion au niveau des données à traiter m'a fait perdre beaucoup de temps).

3.1 InvSBox

3.1.1 Principe

L'algorithme AES utilise une table de substitution, appelée S-Box, dans la fonction *InvSubBytes*. Cette table prend en entrée un octet, et fournit en sortie un autre octet. La figure suivante montre la S-Box inversée utilisée pour ce projet :

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	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

FIGURE 3 – Inverse de la S-Box

Par exemple, en entrant l'octet *0xed*, on doit obtenir en sortie *0x53*.

3.1.2 Implémentation

L'inverse de la S-Box a été implémenté sous forme d'un tableau de 256 cases de *bit8* initialisé à la main. On peut représenter l'entité *InvSBox* par le schéma suivant :



FIGURE 4 – Schéma de l'entité InvSBox

L'implémentation de cette entité se trouve dans le fichier *InvSBox.vhd*.

3.1.3 Tests et résultats

Le test bench *InvSBox_tb.vhd* a pour objectif de tester cette S-Box inversée. Plutôt que de tester quelques valeurs au hasard, il a été choisi de créer un processus dans lequel une variable s'incrémente toutes les 200 ns jusqu'à atteindre 255. (Une fois cette valeur atteinte, on remet la variable à 0 et on recommence l'incrémentation). On obtient alors le résultat suivant sur modelsim :

	Msgs						
/invsbox_tb/DUT/invsbox_i	0D	07	08	09	0A	0B	
/invsbox_tb/DUT/invsbox_o	F3	38	BF	40	A3	9E	

FIGURE 5 – Résultat du test bench pour InvSBox

En se référant à la figure 3, page 7, on constate que l'on obtient bien le bon résultat.

3.2 invSubBytes

3.2.1 Principe

La fonction *InvSubBytes* est une transformation non linéaire qui s'applique à tous les octets de l'état en utilisant la S-Box inversée obtenue précédemment.

3.2.2 Implémentation

Cette fonction prend donc en entrée une matrice 4x4 d'octet et retourne également une matrice 4x4 d'octet en ayant utilisée la S-Box inversée. On peut le représenter selon le schéma suivant :

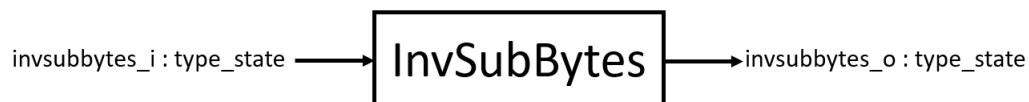


FIGURE 6 – Schéma de l'entité InvSubBytes

Pour implémenter cela, on génère une matrice 4x4 de *InvSBox* (une par octet). Ensuite, il s'agit simplement d'aller chercher la bonne valeur en fonction des coordonnées.

L'implémentation de cette entité se trouve dans le fichier *InvSubBytes.vhd*.

3.2.3 Tests et résultats

Pour tester cette partie, la matrice suivante a été entrée dans le test bench (matrice fournie à la fin du sujet pour le round 9 de la partie déchiffrement) :

0x06	0x85	0xa6	0x61
0x09	0xfb	0x06	0x54
0x99	0xc1	0x5f	0xca
0x5b	0x8e	0x56	0x74

TABLE 1 – Matrice d'entrée du bloc InvSubBytes

La matrice suivante doit être obtenue en sortie :

0xa5	0x67	0xc5	0xd8
0x40	0x63	0xa5	0xfd
0xf9	0xdd	0x84	0x10
0x57	0xe6	0xb9	0xca

TABLE 2 – Matrice de sortie du bloc InvSubBytes

La modélisation sur modelsim donne le résultat suivant :

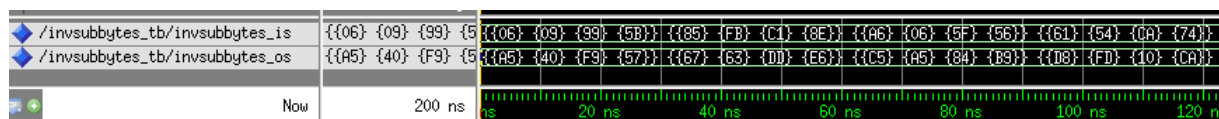


FIGURE 7 – Résultat du test bench pour InvSubBytes

On constate alors que l'on obtient en sortie la matrice souhaitée. On peut donc valider le fonctionnement de cette partie.

3.3 InvAddRoundKey

3.3.1 Principe

La fonction *InvAddRoundKey* ajoute la clé de round courante à l'état (les clés de round sont obtenues à partir de la fonction *KeyExpansion* fournie, hormis pour la clé du round 0). Toutes ses clés sont directement accessibles depuis la librairie fournie avec le sujet.

Elle est tout à fait semblable à la fonction *AddRoundKey*. Cela implique donc qu'il s'agit simplement d'un XOR entre chaque octet de l'état et de la clé en question.

3.3.2 Implémentation

Cette fonction prend en entrée une matrice 4x4 d'octets et une matrice 4x4 correspondant à une clé. Elle fournit en sortie une matrice 4x4 d'octets, résultat du XOR entre l'état et la clé.



FIGURE 8 – Schéma de l'entité InvAddRoundKey

L'implémentation de cette entité se trouve dans le fichier *InvAddRoundKey.vhd*.

3.3.3 Tests et résultats

Pour tester cette partie, la matrice du message chiffré a été utilisée dans le test bench :

0xd6	0x4c	0x47	0xd7
0xef	0xe8	0x6b	0x6a
0xa6	0xef	0x95	0xcd
0xdc	0xd2	0x46	0xf0

TABLE 3 – Matrice d'entrée (message chiffré) du bloc AddRoundKey

Et la clé correspondante suivante a été utilisée :

0xd0	0xc9	0xe1	0xb6
0x14	0xee	0x3f	0x63
0xf9	0x25	0x0c	0x0c
0xa8	0x89	0xc8	0xa6

TABLE 4 – Clé utilisée pour le round 10

La matrice suivante doit alors être obtenue en sortie :

0x06	0x85	0xa6	0x61
0xfb	0x06	0x54	0x09
0x5f	0xca	0x99	0xc1
0x74	0x5b	0x8e	0x56

TABLE 5 – Matrice de sortie du bloc AddRoundKey

La modélisation sur modelsim donne le résultat suivant :

/invaddroundkey_tb/invaddroundkey_is	{D6}	{06}	{4C}	{47}	{D7}	{EF}	{E8}	{6B}	{6A}	{A6}	{EF}	{95}	{CD}	{DC}	{D2}	{46}	{F0}
/invaddroundkey_tb/key_is	{D0}	{D0}	{C9}	{E1}	{B6}	{14}	{EE}	{3F}	{63}	{F9}	{25}	{0C}	{0C}	{A8}	{89}	{C8}	{A6}
/invaddroundkey_tb/invaddroundkey_os	{06}	{06}	{85}	{A6}	{61}	{FB}	{06}	{54}	{09}	{5F}	{CA}	{99}	{C1}	{74}	{5B}	{8E}	{56}

FIGURE 9 – Résultat du test bench pour InvAddRoundKey

On constate alors que l'on retrouve en sortie la matrice souhaitée. On peut donc également valider le fonctionnement de cette partie.

3.4 InvShiftRows

3.4.1 Principe

La fonction *InvShiftRows* effectue une permutation cyclique des octets des lignes de l'état selon le schéma suivant :

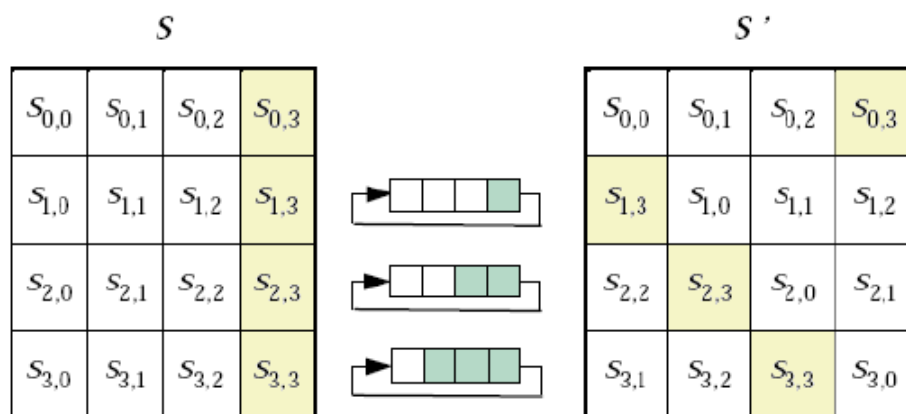


FIGURE 10 – Principe du InvShiftRows

La rotation va dépendre de la ligne. Par exemple, pour la ligne 0 on fait une rotation de 0 octets (donc pas de rotations). Pour la ligne 1 on fera la rotation d'un octet : le dernier

octet de la ligne passe en première position et les autres se retrouvent alors décalés de 1. Pour la deuxième ligne le dernier octet passe en deuxième position et l'avant dernier en première position. Les autres octets se retrouvent alors décalés de 2, et ainsi de suite.

3.4.2 Implémentation

Cette fonction prend en entrée une matrice 4x4 d'octets et fournit en sortie une matrice 4x4.



FIGURE 11 – Schéma de l'entité InvShiftRows

L'implémentation de cette entité se trouve dans le fichier *InvShiftRows.vhd*.

3.4.3 Tests et résultats

Pour tester cette fonction, la matrice suivante a été passée en entrée (résultat de *AddRoundKey* sur le message chiffré) :

0x06	0x85	0xa6	0x61
0xfb	0x06	0x54	0x09
0x5f	0xca	0x99	0xc1
0x74	0x5b	0x8e	0x56

TABLE 6 – Matrice d'entrée du bloc InvShiftRows

Normalement, la matrice suivante doit être obtenue en sortie :

0x06	0x85	0xa6	0x61
0x09	0xfb	0x06	0x54
0x99	0xc1	0x5f	0xca
0x5b	0x8e	0x56	0x74

TABLE 7 – Matrice de sortie du bloc InvShiftRows

La simulation sur modelsim nous donne le résultat suivant :

♦ /invshiftrows_tb/invshiftrows_is	{{06}}	{{06} {85} {46} {61}} {{FB} {06} {54} {09}} {{5F} {CA} {99} {C1}} {{74} {5B} {8E} {56}}
♦ /invshiftrows_tb/invshiftrows_os	{{06}}	{{06} {85} {46} {61}} {{09} {FB} {06} {54}} {{99} {C1} {9F} {CA}} {{5B} {8E} {56} {74}}

FIGURE 12 – Résultat du test bench pour InvShiftRows

3.5 InvMixColumns

3.5.1 Principe

La transformation *InvMixColumns* s'applique colonne par colonne. Chaque octet de chaque colonne est multiplié par 9, 11, 13 et 14. La figure suivante illustre cette transformation :

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

$$\begin{aligned}
 s'_{0,c} &= (\{0e\} \bullet s_{0,c}) \oplus (\{0b\} \bullet s_{1,c}) \oplus (\{0d\} \bullet s_{2,c}) \oplus (\{09\} \bullet s_{3,c}) \\
 s'_{1,c} &= (\{09\} \bullet s_{0,c}) \oplus (\{0e\} \bullet s_{1,c}) \oplus (\{0b\} \bullet s_{2,c}) \oplus (\{0d\} \bullet s_{3,c}) \\
 s'_{2,c} &= (\{0d\} \bullet s_{0,c}) \oplus (\{09\} \bullet s_{1,c}) \oplus (\{0e\} \bullet s_{2,c}) \oplus (\{0b\} \bullet s_{3,c}) \\
 s'_{3,c} &= (\{0b\} \bullet s_{0,c}) \oplus (\{0d\} \bullet s_{1,c}) \oplus (\{09\} \bullet s_{2,c}) \oplus (\{0e\} \bullet s_{3,c})
 \end{aligned}$$

FIGURE 13 – Illustration de la transformation opérée par InvMixColumn

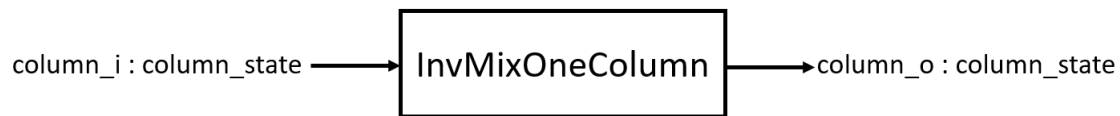
Toutes ces opérations se passent dans le corps de Galois. Les additions peuvent donc se calculer à l'aide d'un XOR. De manière générale, on peut décomposer une multiplication par 13 de la manière suivante :

$$\begin{aligned}
 13 \times x &= x \times (8 \oplus 4 \oplus 1) \\
 &= x \times (2^3 \oplus 2^2 \oplus 2^0) \\
 &= x \times ((2 \oplus 2 \oplus 2 \oplus 2) \oplus (2 \oplus 2) \oplus 1)
 \end{aligned}$$

On remarque donc qu'au final, il ne s'agit que de multiplication et addition par 2 (plus éventuellement une addition avec l'élément lui-même), ce qui est finalement assez simple à mettre en place d'un point de vue informatique.

3.5.2 Implémentation

Etant donné que l'opération *InvMixColumns* opère colonne par colonne, un bloc *InvMixOneColumn* a été mis en place. Le bloc général *InvMixColumns* fera alors appel 4 fois à cette plus petite entité. L'entité *InvMixOneColumn* prend donc en entrée une colonne de 4 octets, et donne le même type en sortie.

FIGURE 14 – Schéma de l'entité *InvMixOneColumn*

Pour implémenter cette transformation, des signaux intermédiaires de type *column_state* ont été créés afin de stocker les résultats des multiplications par 2, 4, 8, 9, 11, 13 et 14. En effet, chacun des résultats de ces multiplications utilise le résultat des multiplications précédentes.

Pour multiplier par 2, il suffit de faire un décalage à gauche des bits et de compléter à droite avec un zéro. Si le bit de poids fort vaut 0, cela ne pose aucun problème, on reste sur 8 bits (le MSB n'a alors aucune importance). Cependant, si le bit de poids fort vaut 1, cela pose problème car on se retrouve alors sur 9 bits. Pour palier à ce problème, il suffit de faire un ou-exclusif avec la valeur b00011011. Ce ou-exclusif ne doit être appliqué que si le MSB initial vaut 1. Cependant, pour éviter de tester à chaque fois cette condition, on effectue systématiquement ce ou-exclusif, mais conditionné par le MSB de l'octet en question. En effet, si ce MSB vaut 0, alors le XOR n'a aucun effet. La ligne suivante illustre ce principe en VHDL :

```

times2_s(i) <= (x_i(i)(6 downto 0)&'0') xor
  ("000"&x_i(i)(7)&x_i(i)(7)&'0'&x_i(i)(7)&x_i(i)(7));
  
```

Une fois les multiplications par 8, 4 et 2 obtenues, il ne reste plus qu'à faire des XOR entre ces différents résultats pour obtenir les multiplications par 9, 11, 13 et 14. Une fois tous ces résultats obtenus, il ne reste plus qu'à appliquer des XOR pour obtenir le résultat final, en suivant les opérations indiqués en figure 13 page 13.

Une fois que cette partie est fonctionnelle, il ne reste plus qu'à faire le bloc général *InvMixColumns*. Celui-ci prend en entrée une matrice (de type *_state*) et retourne également une matrice.

FIGURE 15 – Schéma de l'entité *InvMixColumns*

Cette partie est relativement simple à mettre en œuvre car elle fait appel à 4 *InvMixOneColumn* (une pour chaque colonne). Le seul point particulier est le type de données trai-

tées. En effet, *InvMixOneColumn* travail sur des colonnes tandis que le type *type_state* est un tableau de ligne. Il faut donc créer des signaux intermédiaires pour récupérer des colonnes, puis de faire l'opération inverse une fois chaque colonne traitée.

L'implémentation de ces entités se trouve dans les fichiers *InvMixColumns.vhd* et *InvMixOneColumn*.

3.5.3 Tests et résultats

Un premier test bench a été réalisé pour vérifier le fonctionnement de l'entité *MixOneColumn*. Les résultats de ces tests ne sont pas développés ici, seul les résultats pour l'entité *InvMixColumns* seront présentés (le bon fonctionnement de cette entité témoigne du bon fonctionnement de la sous entité). Pour tester *InvMixColumns*, la matrice suivante a été utilisée en entrée (il s'agit de la matrice de sortie de *AddRoundKey* du round 10) :

0x09	0x7e	0xed	0x8f
0x37	0x99	0x74	0xa1
0x9f	0x01	0xad	0x10
0xa4	0xc7	0xf8	0xa4

TABLE 8 – Matrice d'entrée du bloc *InvMixColumns*

Normalement, la matrice suivante doit être obtenue en sortie :

0x36	0x7e	0x29	0xbf
0x2b	0xe3	0x2d	0xea
0xaa	0x43	0xea	0x0f
0xb2	0xff	0x22	0xc0

TABLE 9 – Matrice de sortie du bloc *InvMixColumns*

Voici les résultats obtenus en simulation :



 /invmixcolumns_tb/invmixcolumns_is	{{09}}	{{09}} {{7E}} {{ED}} {{8F}} {{37}} {{99}} {{74}} {{A1}} {{9F}} {{01}} {{AD}} {{10}} {{A4}} {{C7}} {{F8}} {{A4}}
 /invmixcolumns_tb/invmixcolumns_os	{{36}}	{{36}} {{7E}} {{29}} {{BF}} {{2B}} {{E3}} {{2D}} {{EA}} {{AA}} {{43}} {{EA}} {{0F}} {{B2}} {{FF}} {{22}} {{C0}}

FIGURE 16 – Résultat du test bench pour *InvMixColumns*

On obtient bien le résultat escompté.

3.6 Machine d'état

3.6.1 Principe

Afin de pouvoir faire fonctionner l'algorithme de déchiffrement, une machine d'état (FSM) est nécessaire. Cette machine d'état va contrôler les différents états de l'algorithme et autoriser certaines sorties en fonction de ses états.

3.6.2 Implémentation

Pour implémenter cette FSM, l'architecture retenue est l'architecture de Moore. Ainsi, les sorties dépendent de l'état présent (synchrones, elles changent sur un front d'horloge), et l'état futur dépend de l'état présent et des entrées. Voici le schéma de cette entité :

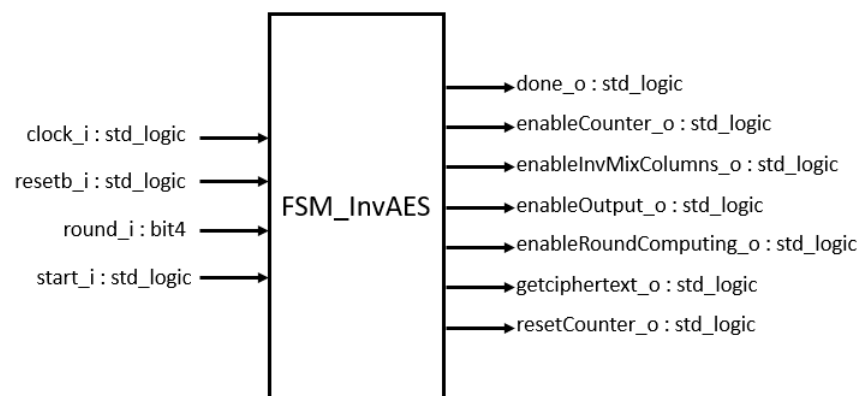


FIGURE 17 – Schéma de l'entité FSM_InvAES

Voici la signification des entrées et sorties :

- *clock_i* représente l'horloge
- *resetb_i* permet d'initialiser le circuit (reset à l'état bas)
- *round_i* désigne le numéro du round actuel
- *start_i* permet de débiter le calcul
- *done_o* indique la fin du calcul
- *enableCounter_o* autorise la décrémentation du compteur
- *enableMixColumns_o* autorise l'étape *InvMixColumns*
- *enableOutput_o* autorise la sortie
- *enableRoundComputing_o* autorise l'étape *InvSubBytes* et *InvShiftRows*
- *getciphertext_o* permet de récupérer le texte chiffré
- *resetCounter_o* permet de réinitialiser le décompteur

Pour implémenter cette partie, le graph d'état suivant a été retenu :

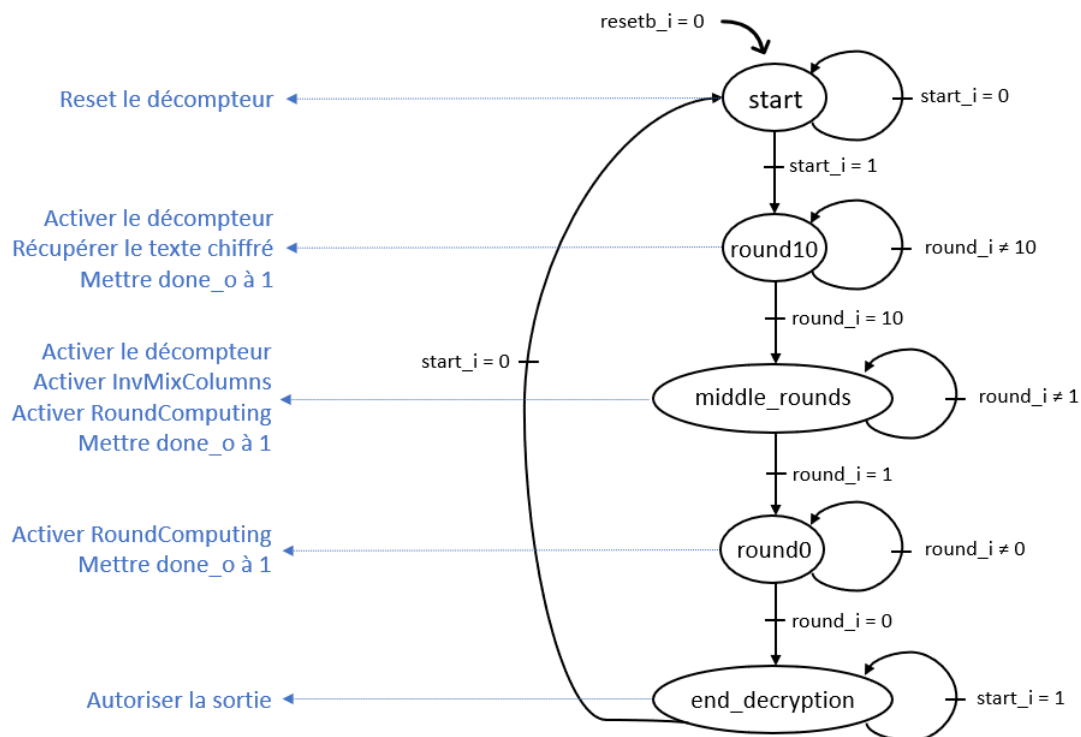


FIGURE 18 – Graph d'état pour Inv_AES

Sur ce graph, la sortie *done_o* est mise à 1 lorsque le calcul est en cours et non lorsqu'il est fini, car sur le schéma général le signal *done_o* correspond au signal *aes_on_o* qui indique que le calcul est en cours. Une autre solution aurait été d'ajouter dans l'entité générale un inverseur sur ce signal, comme cela est fait pour le signal *reset_i*. L'implémentation de cette entité se trouve dans le fichier *FSM_InvAES.vhd*.

3.6.3 Tests et résultats

Pour tester cette partie, les signaux suivants ont été utilisés dans le test bench (le décompteur a été décrémenté à la main) :

```

resetb_is <= '1', '0' after 30 ns, '1' after 40 ns;
start_is <= '0', '1' after 50 ns, '0' after 200 ns;
clock_is <= not clock_is after 25 ns;
round_is <= "1010" after 100 ns, "1001" after 150 ns, "1000" after 200 ns,
           "0111" after 250 ns, "0110" after 300 ns, "0101" after 350 ns,
           "0100" after 400 ns, "0011" after 450 ns, "0010" after 500 ns,
           "0001" after 550 ns, "0000" after 600 ns;

```

On obtient le résultat suivant :

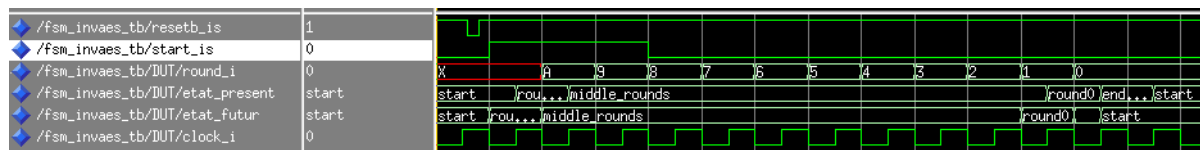


FIGURE 19 – Résultat du test bench pour FSM_InvAES

On remarque que l'on obtient bien le fonctionnement attendu par le graph d'état. On peut donc valider le fonctionnement de cette partie.

3.7 Décompteur

3.7.1 Principe

Afin de pouvoir savoir dans quel round on se trouve, il est nécessaire d'implémenter un décompteur. Ce décompteur doit pouvoir être initialisé à une certaine valeur, et se décrémenter à chaque coup d'horloge. A noter également qu'il faudra gérer un éventuel débordement et empêcher au décompteur de se décrémenter lorsqu'il atteint 0.

3.7.2 Implémentation

Ainsi, le décompteur dispose de 3 entrées (une horloge, un signal d'initialisation et un signal d'autorisation à se décrémenter) et une sortie correspondant aux différentes valeurs prises par cette entité.



FIGURE 20 – Schéma de l'entité Counter

On peut remarquer que le reset du décompteur est un reset à l'état haut. En effet, dans la FSM la sortie *resetCounter_o* est mise à 1 lorsque nous voulons reset le counter (il est plus logique, de premier abord, de mettre à 1 une sortie que l'on veut activer). On pourrait également plutôt mettre un reset à l'état bas en modifiant seulement la valeur de *resetCounter_o* dans la FSM en fonction des états présents. Ici, le reset à l'état haut a été gardé. L'implémentation de cette entité se trouve dans le fichier *Counter.vhd*.

3.7.3 Tests et résultats

Le test bench du décompteur nous permet d'obtenir le résultat suivant :

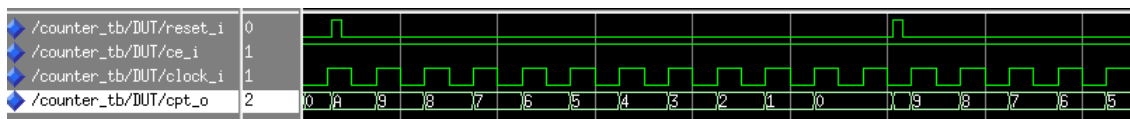


FIGURE 21 – Résultat du test bench pour l'entité Counter

Le décompteur prend donc toutes les valeurs de 10 à 0, ce qui permet bien de pouvoir accéder par la suite à tous les états de la FSM. On remarque aussi que le décompteur conserve bien la valeur 0 jusqu'à un nouveau reset. On peut donc valider le fonctionnement de cette entité.

3.8 Registre

3.8.1 Principe

Un registre est nécessaire dans l'entité générale de InvAES afin de pouvoir sauvegarder en mémoire pendant un certains temps le résultat. Pour faire cela, on utilise un registre. Il s'agit en réalité d'une simple bascule D. Cette entité sera nommée *RTL_REG*.

3.8.2 Implémentation

Etant donné qu'il s'agit d'une bascule D, on retrouve en entrée une horloge, un reset (ici à l'état bas), un signal d'autorisation et l'entrée des données. Il y a une seule sortie, celle des données. On a donc l'entité suivante :

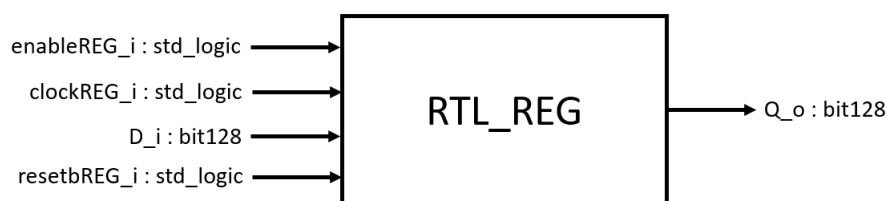


FIGURE 22 – Schéma de l'entité RTL_REG

Ce registre est constitué de deux processus séquentiels. Le premier est utilisé pour le reset du registre et le second pour garder en mémoire ou non la donnée. L'implémentation de cette entité se trouve dans le fichier *RTL_REG.vhd*.

3.8.3 Tests et résultats

Je n'ai pas réalisé de test bench pour cette entité car il s'agit d'une simple bascule D et une partie du cours a été donnée lors d'une séance de cours.

3.9 InvAESRound

3.9.1 Principe

Cette partie permet de réaliser toutes les étapes d'un round en fonction du round considéré. Ainsi, cette entité fait appel à *InvSubBytes*, *InvShiftRows*, *InvMixColumns* et *InvAddRoundKey*.

3.9.2 Implémentation

Cette entité prend donc en entrée le signal *enableInvMixColumns_i*, le signal *enableRoundComptuing_i*, le signal *reseth_i*, le texte courant (*currenttext_i*), la clé du round correspondant (*currentkey_i*) et une horloge (*clock_i*). En sortie, on retrouve un bit128.

Comme on ne fait pas toutes les étapes dans tous les rounds, il faut sélectionner quelles sous-entités sont utilisées en fonction du round. Pour cela, deux multiplexeurs sont utilisés. Pour cela, un composant à part a été crée (*Multiplexer_type_state.vhd*). Ce multiplexer travail avec le type *type_state*.

De même, afin de pouvoir garder un minimum la valeur de sortie en mémoire, il faut un registre. Celui-ci a été implémenté directement à l'intérieur de l'entité *InvAESRound* (les essais pour réutiliser le registre *RTL_REG* n'ont pas été très concluants) sous forme d'un processus combinatoire.

Enfin, les données d'entrée et de sortie sont du type *bit128*. Cependant, les sous-entités utilisent le type *type_state*. Il faut donc faire une conversion de type (de bit128 à *type_state* et inversement). Pour faire ces conversions, il faut appliquer le pseudo-code suivant :

```
- - Conversion bit128 en type_state
  k entier
  l entier
  Pour i allant de 0 à 3 faire
    Pour j allant de 0 à 3 faire
      k := 127-8*(i+j*4)
      l := 127-8*(i+j*4+1)
      MatriceTypeState[i][j] := TypeBit128[k à l]
    fin pour
```

```

fin pour

-- Conversion type_state en bit128
k entier
l entier
Pour i allant de 0 à 3 faire
    Pour j allant de 0 à 3 faire
        k := 127-8*(i+j*4)
        l := 127-8*(i+j*4+1)
        TypeBit128[k à l] := MatriceTypeState[i][j]
    fin pour
fin pour

```

Je n'ai pas trouvé la formule de conversion seul. Un camarade (Cédric GERNIGON) me l'a donnée.

Au final, on dispose de l'entité suivante :

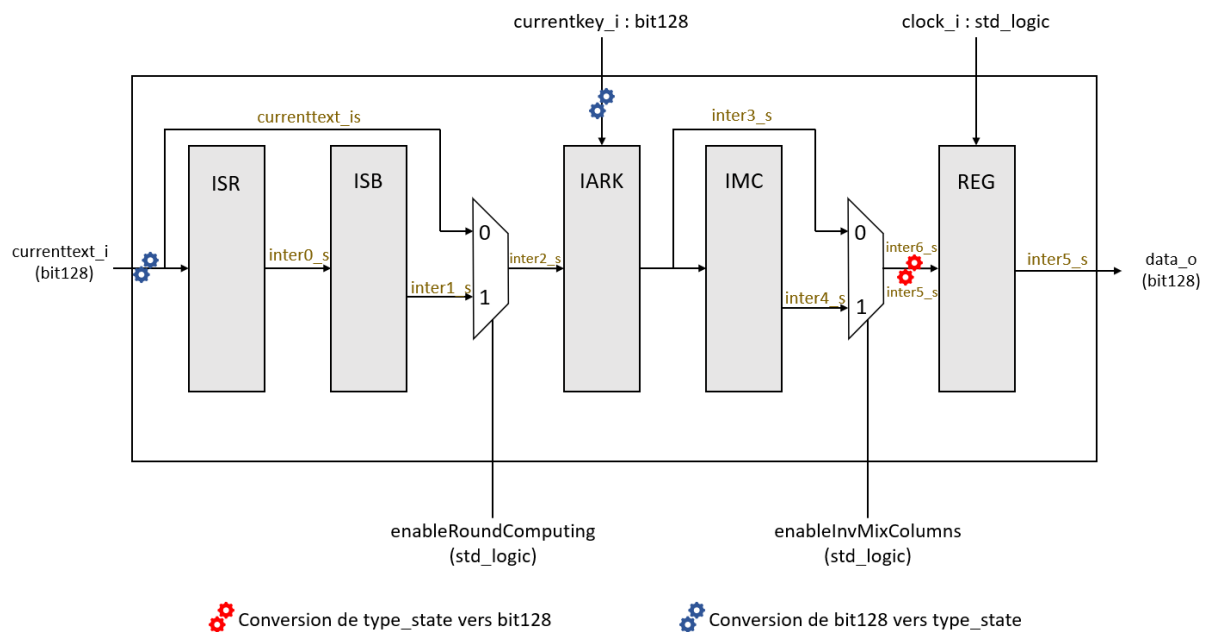


FIGURE 23 – Schéma de l'entité *InvAESRound*

La conversion de type aurait également pu se faire dans une entité à part entière. L'implémentation de l'entité *InvAESRound* se trouve dans le fichier *InvAESRound.vhd*.

3.9.3 Tests et résultats

Cette entité à été testée pour le round 10 et 9. Seuls les résultats pour le round 10 sont détaillés ici. Ainsi, les matrices d'entrée sont les suivantes :

0xd6	0x4c	0x47	0xd7
0xef	0xe8	0x6b	0x6a
0xa6	0xef	0x95	0xcd
0xdc	0xd2	0x46	0xf0

TABLE 10 – *currenttext_i* pour le round 10

0xd0	0xc9	0xe1	0xb6
0x14	0xee	0x3f	0x63
0xf9	0x25	0x0c	0x0c
0xa8	0x89	0xc8	0xa6

TABLE 11 – *currentkey_i* pour le round 10

On doit normalement obtenir la matrice suivante en sortie :

0x06	0x85	0xa6	0x61
0xfb	0x06	0x54	0x09
0x5f	0xca	0x99	0xc1
0x74	0x5b	0x8e	0x56

TABLE 12 – Résultat de InvAESRound pour le round 10

Le test bench nous donne le résultat suivant :



FIGURE 24 – Résultat du test bench pour l'entité InvAESRound

On obtient bien le résultat attendu. On remarque sur ce test bench le résultat sorti pour le round 9 (qui correspond lui aussi à ce que l'on doit obtenir) qui dure bien un cycle d'horloge.

4 Implémentation globale de InvAES

4.1 Principe

Une fois tous ces composants créés, il ne reste plus qu'à les rassembler au sein de l'entité générale *InvAES*. Il faut noter que le composant *KeyExpansion_table* est déjà fourni, c'est pourquoi il n'a pas été détaillé comme les autres composants. Il a seulement pour but de fournir la bonne clé intermédiaire en fonction du round.

4.2 Implémentation

Pour implémenter cette entité générale, on utilise le schéma suivant :

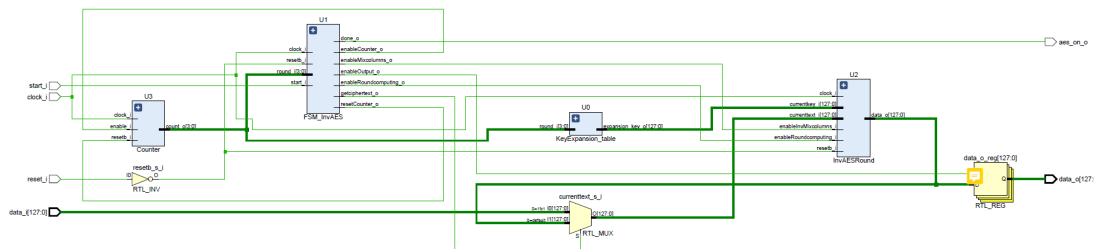


FIGURE 25 – Schéma de l'entité générale InvAES

La première chose que l'on remarque est l'inverseur. En effet, le signal de reset passé en entrée est actif haut. Cependant, de nombreux composants internes utilisent un signal de reset actif bas. Cet inverseur a été simplement implémenté directement dans l'entité générale.

De plus, on peut également apercevoir un multiplexeur. Celui-ci a été implémenté directement, sous forme d'un processus séquentiel, dans l'entité générale plutôt que d'en faire une entité à part entière. L'implémentation de cette entité se trouve dans le fichier *InvAES.vhd*.

4.3 Tests et résultats

Pour tester le bon fonctionnement général, on envoie en entrée le texte chiffré et la clé correspondante. On doit retrouver en sortie le texte *"Resto en ville ?"*. Voici le résultat obtenu sur modelsim :

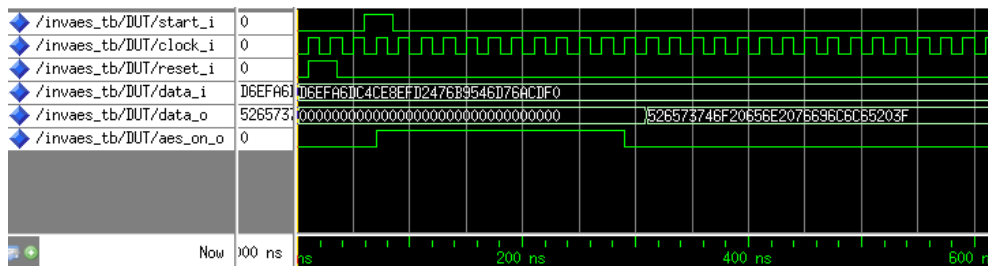


FIGURE 26 – Résultat du test bench pour l'entité InvAES

Une fois le résultat affiché en ASCII on trouve :

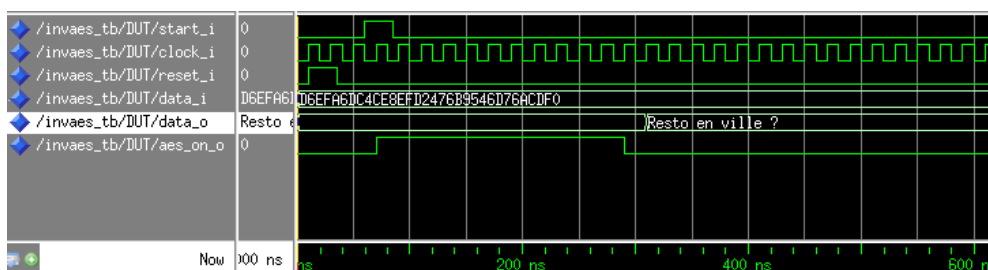


FIGURE 27 – Résultat du test bench pour l'entité InvAES affiché en ASCII

On obtient le résultat attendu. On remarque bien que *aes_on_o* est à 1 pendant 11 coups d'horloge.

On peut également noter que la simulation sur modelsim nous fait apparaître des warnings. Aucune solution n'a été trouvée pour résoudre ce problème, qui ne semble pas avoir d'importance pour le résultat final. Néanmoins, notre déchiffrement AES fonctionne.

En regardant le résultat attendu dans le sujet, on remarque qu'il y a également un autre message que l'on peut passer au déchiffrement. Le sujet indique que ce message est la réponse d'Alice. ce message chiffré est :

`x"d7ca070cc0d3ce1e3943287756404506"`

On obtient alors le résultat suivant sur modelsim :

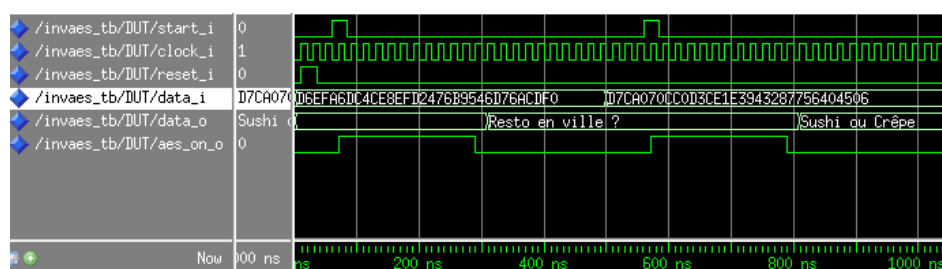


FIGURE 28 – Résultat final du déchiffrement

On obtient donc ce qui semble être la réponse de Alice à Bob.

5 Conclusion

Ce projet m'a permis de comprendre l'intérêt et le principe du langage de description qu'est le VHDL. Les plus grosses difficultés se sont situées au début du projet et à la fin. En effet, au début je ne maîtrisais pas assez bien le cours et ce langage de description pour pouvoir avancer assez rapidement. A ce moment, il m'a fallu beaucoup de temps pour bien assimiler les principes d'entités, d'architectures et de configurations. De plus, la plupart des erreurs étaient des erreurs de syntaxes. Une fois ces problèmes surmontés, la réalisation des différents composants a été assez simple une fois le principe de fonctionnement du composant compris. Enfin, j'ai du passer beaucoup de temps lors de la réalisation finale (pour *InvAESRound* et *InvAES*) pour que tout fonctionne. Il y avait de nombreux problèmes entre des signaux actifs à l'état bas ou haut, avec des *reset* et *init* activés au mauvais moment ou bien encore avec des *start* dans les test bench qui ne duraient pas assez longtemps pour être actifs sur un front montant d'horloge (c'est ce type d'erreur qui m'a fait perdre le plus de temps à la fin).

Mais au final le déchiffrement fonctionne correctement.

Table des figures

1	Algorigramme de InvAES	5
2	Pseudo-code de InvAES (FIPS 197)	5
3	Inverse de la S-Box	7
4	Schéma de l'entité InvSBox	7
5	Résultat du test bench pour InvSBox	8
6	Schéma de l'entité InvSubBytes	8
7	Résultat du test bench pour InvSubBytes	9
8	Schéma de l'entité InvAddRoundKey	10
9	Résultat du test bench pour InvAddRoundKey	11
10	Principe du InvShiftRows	11
11	Schéma de l'entité InvShiftRows	12
12	Résultat du test bench pour InvShiftRows	13
13	Illustration de la transformation opérée par InvMixColumn	13
14	Schéma de l'entité InvMixOneColumn	14
15	Schéma de l'entité InvMixColumns	14
16	Résultat du test bench pour InvMixColumns	15
17	Schéma de l'entité FSM_InvAES	16
18	Graph d'état pour Inv_AES	17
19	Résultat du test bench pour FSM_InvAES	18
20	Schéma de l'entité Counter	18
21	Résultat du test bench pour l'entité Counter	19
22	Schéma de l'entité RTL_REG	19
23	Schéma de l'entité InvAESRound	21
24	Résultat du test bench pour l'entité InvAESRound	22
25	Schéma de l'entité générale InvAES	23
26	Résultat du test bench pour l'entité InvAES	24
27	Résultat du test bench pour l'entité InvAES affiché en ASCII	24
28	Résultat final du déchiffrement	24

Liste des tableaux

1	Matrice d'entrée du bloc InvSubBytes	9
2	Matrice de sortie du bloc InvSubBytes	9
3	Matrice d'entrée (message chiffré) du bloc AddRoundKey	10
4	Clé utilisée pour le round 10	10

5	Matrice de sortie du bloc AddRoundKey	11
6	Matrice d'entrée du bloc InvShiftRows	12
7	Matrice de sortie du bloc InvShiftRows	12
8	Matrice d'entrée du bloc InvMixColumns	15
9	Matrice de sortie du bloc InvMixColumns	15
10	<i>currenttext_i</i> pour le round 10	22
11	<i>currentkey_i</i> pour le round 10	22
12	Résultat de InvAESRound pour le round 10	22