PCSN Limouri Khalil

RAPPORT DE CONCEPTION EN VHDL

PROJET DE CONCEPTION DE SYSTEME NUMERIQUE

21 Février 2022

ENSMSE ISMIN 2A

TABLE DES MATIERES

I. Éléments de contexte

1. Introduction

Dans le cadre de la deuxième année d'études à l'École Nationale Supérieure des Mines de Saint-Étienne, cycle ISMIN, nous avions eu l'occasion d'assister à des séances de Projet Conception de Système Numérique. Ces dernières s'appuient sur les connaissances théoriques acquises lors des cours du langage VHDL. Le présent document constitue un rapport de conception du circuit de déchiffrement du protocole l'AES ayant 11 rondes et une taille de clé de 128 bits.

2. Définition du chiffrement AES

L'Advanced Encryption Standard (noté AES) est un standard de chiffrement symétrique par blocs qui repose sur l'algorithme cryptographique de Rijndael approuvé par le Federal Information Processing Standards (FIPS). L'entrée et la sortie sont des séquences de taille 128 bits, et les clés de chiffrement sont, au choix, de longueur 128, 192 ou 256 bits. Toutes les autres tailles de clé ou des blocs d'entrée et de sortie ne sont pas permises par le standard FIPS.

Même si le standard AES est non-classifié, publique, et sans droits d'utilisation, il est théoriquement très long à résoudre à l'échelle de la vie de l'Homme. Autre que l'attaque du type force brute, aucune autre attaque n'est significativement plus efficace, à l'exception de l'attaque par canaux auxiliaire qui exige l'accessibilité matérielle de l'opérateur autrement dit la proximité physique au circuit pour la réaliser.

3. Utilité de l'AES

L'AES sert essentiellement à la sécurisation des transactions en ligne, la protection des communications sans fil, au stockage des informations non classifiées ou sensibles et plus généralement le cryptage des données électroniques. Comme il est largement exploité lors des échanges où la vitesse du traitement est une ressource clé, Nous estimons que l'intégration d'un circuit dédié au déchiffrement et chiffrement AES est très utile.

4. Fonctionnement du déchiffrement de l'AES

L'implémentation du circuit ne concerne que le déchiffrement de données qui ont été chiffrées en amont. Ainsi la représentation du composant AES inverse correspond relativement à la figure 1.

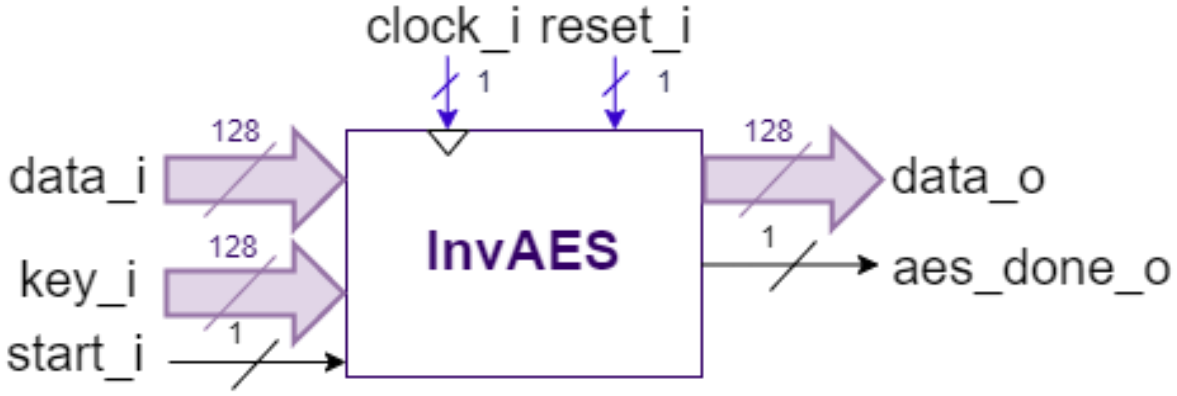


Figure – 1 : Représentation de l'entité de l'AES inverse

La totalité des clés, en incluant celle en entrée[1], est intégrée explicitement au niveau haut[2] de la hiérarchie. L'entrée de la clé n'est plus nécessaire grâce au tableau des clés, dont l'algorithme de génération[3] se trouve en figure 2 ci-dessous.

[1] La clé **key_i** de taille 128 bits.

[2] Intégration réalisée à l'aide du tableau du bloc KeyExpansion_table.

[3] L'extension de la clé initiale en plusieurs valeurs de clé.

Figure – 2 : L'algorithme de la fonction KeyExpansion

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
   word  temp

   i = 0

   while (i < Nk)
      w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
      i = i+1
   end while

   i = Nk

   while (i < Nb * (Nr+1)]
      temp = w[i-1]
      if (i mod Nk = 0)
         temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
      else if (Nk > 6 and i mod Nk = 4)
         temp = SubWord(temp)
      end if
      w[i] = w[i-Nk] xor temp
      i = i + 1
   end while
end
```

01	02	04	08	10	20	40	80	1b	36
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00

Rcon

5. Objectif et principe de l'AES inverse

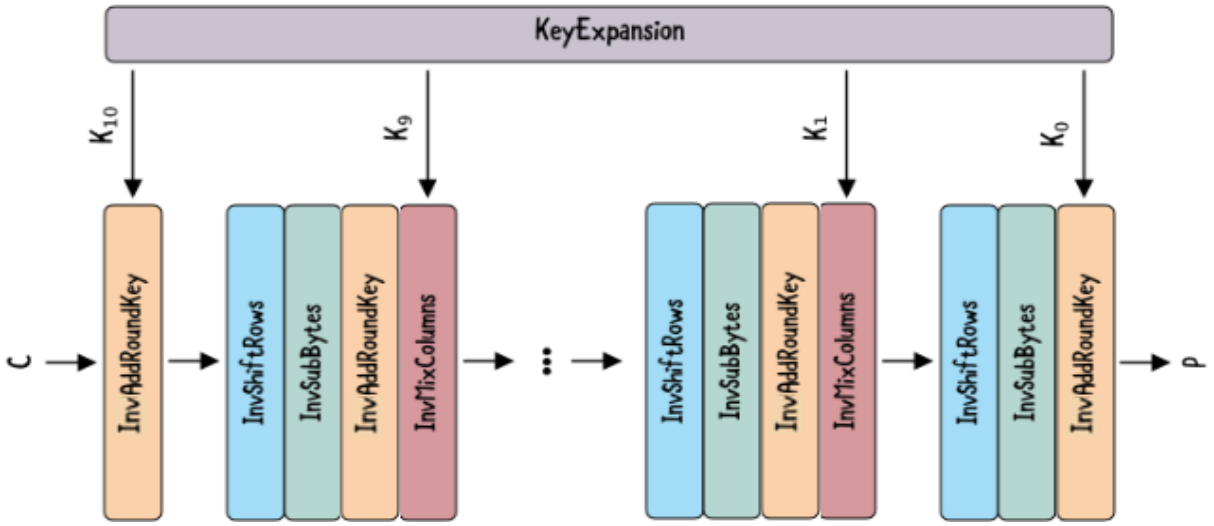
L'intérêt de l'implémentation est avant tout de nature académique et pédagogique, puisque la conception et le code en VHDL du circuit de déchiffrement de l'AES fournit une vue globale et une compréhension solide des langages de description matériel en générale et plus particulièrement le langage VHDL.

Le déchiffrement de l'AES, dont le fonctionnement figure dans l'illustration[4] proposée, se compose de 11 rondes, durant lesquelles les 4 fonctions élémentaires suivantes sont potentiellement[5] utilisées :

1. ***InvShiftRows.***
2. ***InvSubBytes.***
3. ***InvAddRoundKey.***
4. ***InvMixColumns.***

La première ronde n'utilise que la 3ème fonction, par contre la dernière ronde n'utilise pas la 4ème fonction. Et toutes les fonctions sont utilisées par les 9 rondes restantes.

Figure – 3 : Déchiffrement de l'AES à 11 rondes



[4] Avec les conventions suivantes, C : CipherText - P : PlainText - K_i : $i^{ème}$ clé.

[5] Notamment pour les rondes numéro 10 et 0.

II. Implémentation VHDL du Projet

A. Bibliothèques utiles au projet

La bibliothèque IEEE contient toutes les fonctions d'opération entre les bits, conversion de type, de détection de front ainsi que la déclaration des types de base et dérivées pour les signaux et les variables numériques. Le package *numeric_std* qui fait partie de la librairie IEEE est manipulé, explicitement, lors du projet avec des fonctions qui convertissent les bits en des nombres.

La seconde bibliothèque intitulée *state_definition_package.vhd* nous a été fournie au commencement du projet. Elle comporte la définition des sous types dont : bit4, bit8, bit8, bit32 et bit128 qui sont des vecteurs de bits, ainsi que celle de type_state et key_state comme étant une matrice 4x4 de 16 octets. La fonction xor pour les tableaux d'octets, pour l'opération InvMixColomns est aussi définie, mais vu qu'elle était incomplète, nous l'avons utilisé pour ne pas ajouter une source d'erreur dans la conception.

Nous avons importé ces deux bibliothèques, dont la deuxième est placé dans le dossier THIRDPARTY, dans l'entièreté des fichiers register transfert level[6] et test bench[7] qui se trouvent, respectivement dans les dossiers RTL et BENCH du répertoire invaes.

B. Blocs élémentaires du circuit AES inverse

1. La fonction ShiftRows inverse

La permutation des colonnes (shift rows, noté SR) inverse, permet comme l'illustre la figure, de réordonner, par rotation, les octets des lignes de la matrice[8]. Il suffit de changer, en affectant cycliquement, l'ordre des éléments de cette matrice.

L'entité, représentée ci-dessous, a deux ports de données de type type_state, un premier pour l'entrée et un deuxième pour la sortie.

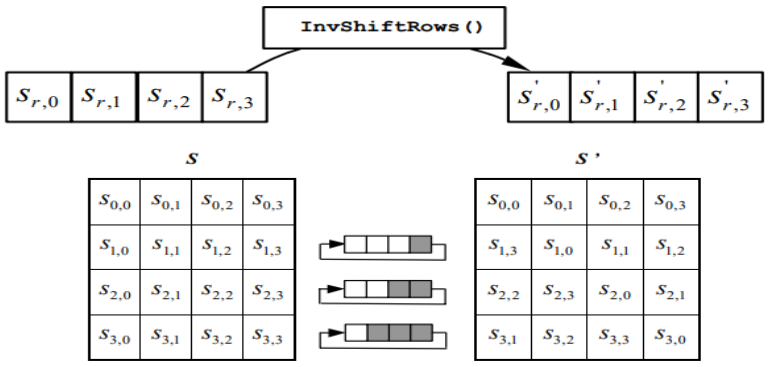


Figure – 4 : Représentation de la fonction SR inverse

L'architecture est de type *dataflow* car elle ne contient pas de l'instanciation ni port mapping[9], et pas de processus[10] explicite séquentielle ou combinatoire. Il est possible de décaler chaque octet ce qui est certes transparent, or c'est redondant pour les 16 octets de la donnée, et peut être raccourcie afin d'avoir uniquement que 5 affectations. Nous avons utilisé le principe de la généricité 4 fois[11] (par le mot clé *generate*) pour réaliser des décalages d'un, de deux, de trois octet vers la droite[12] pour la deuxième, troisième, quatrième ligne, respectivement et pour la première, il n'y a pas de décalage.

Figure – 5 : L'entité de SR inversé

[6] Cela renvoie vers une méthode de description.
[7] Cela signifie un fichier dédié avec des signaux de tests.
[8] La donnée en entrée de la fonction est un tableau de tableau (bidimensionnel).
[9] Dans les composants InvSubBytes et InvMixColmns.
[10] Dans les composants Register, Counter et FSM.
[11] Le nombre de boucles utilisées[2] est 8, dans le chapitre V. Limitations, améliorations et problèmes.
[12] Le sens utilisé en pratique était la gauche, dans le 1er paragraphe de la partie A. Limitations.

Pour le test bench en figure 1 du SR inversé, qui utilise les données en entrée de la ronde 9[13], nous observons que la sortie correspond de l'inverse SR[14] de cette ronde. Ceci permet de conclure au fonctionnement correct de cet élément.

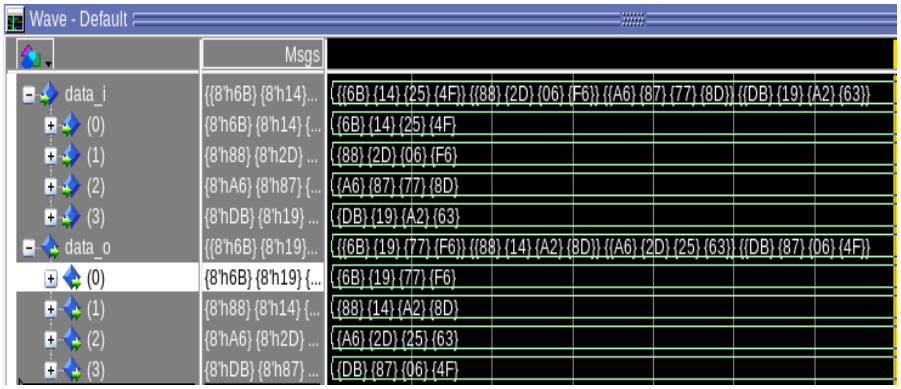


Figure – 6 : Test bench de InvShiftRows

2. La fonction SubBytes inverse

La transformation de soustraction des octets (subbytes noté SB) inverse qui est représentée ci-contre consiste à faire correspondre tout octet en entrée à un octet différent, à l'aide d'une table de Rijndael nommée Inverse S-Box.

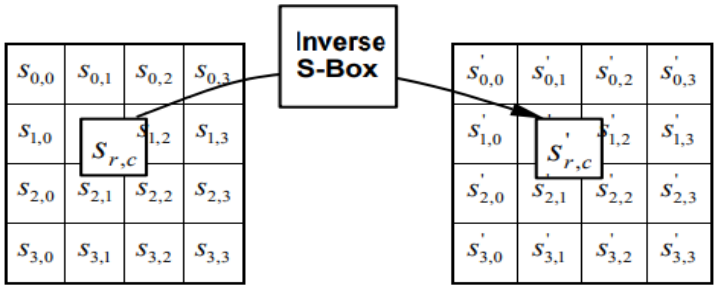


Figure – 7 : Représentation de la fonction SB inverse

i. Inverse S-Box

La table de substitution S-Box inverse est représentée à la figure 8.

Figure – 8 : L'entité du composant S-box inverse

Dans son architecture, nous l'avons modélisée par un tableau constant de 256 octets utilisé le SB inversé. Son contenu est différent de la table S-Box du chiffrement, mais les ports de l'entité, représentée en figure 9, sont identiques : une entrée et une sortie de type bit8. Nous avons associé les ports à l'aide des fonctions de conversion entre les données, en fonction du contenu de la table.

		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 – 9 : Représentation de la table S-box inverse

[13] Le contenu des premiers résultats des blocs sauf le 3ème est en annexe.

[14] Le bloc est nommé en annexe InvShiftRows.

Le test bench ci-dessous, nous permet de vérifier la S-box inversée sur la première ligne de la donnée en ronde 9, avant de l'instancier dans la fonction de substitution principal.

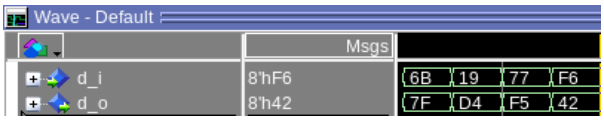


Figure – 10 : Test bench d'Inverse S-Box

ii. *InvSubBytes*

La finalité du bloc SB inverse est de restaurer la version initiale des données avant le remplacement des octets qui a eu lieu lors du chiffrement. Son entité comporte une entrée, **data_i**, et une sortie, **data_o**, de type type_state. Dans l'architecture, nous déclarons le composant S-Box qui est cité dans la configuration, afin d'en instancier 16 cellules **s_box_cell** dont les ports sont reliés à ceux du composant SB inverse avec la syntaxe *port map*[15]. Il était possible d'exploiter les propriétés mathématiques des fonctions modulo et division naturelle entre entiers pour réduire[16] le nombre d'indices des boucles d'itération.

Figure – 11 : L'entité du composant SB inverse

Le test bench en figure 12, a été réalisée pour les octets en entrée de cette opération à la 9ème ronde. L'entrée a été remplacée suivant la valeur correspondante à l'entrée de la table de la S-Box.

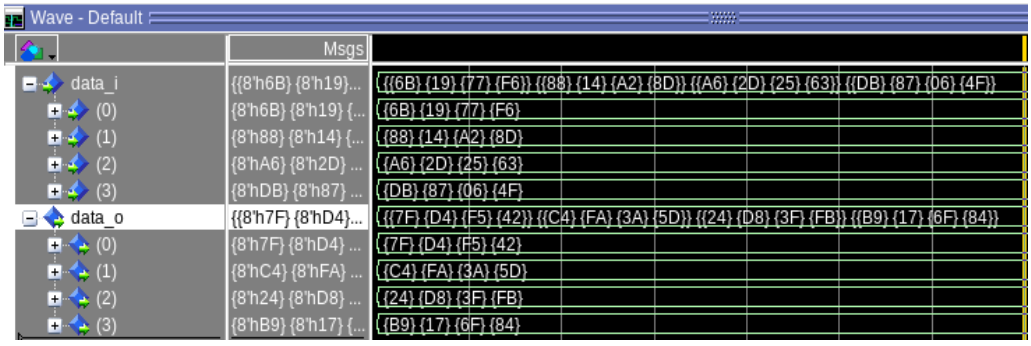


Figure – 12 : Test bench de InvSubBytes

3. La fonction InvAddRoundKey inverse

L'étape de l'ajout de clé de ronde (AddRoundKey noté ARK), illustrée dans la figure 13, est la seule qui occupe toute une ronde[17]. Elle n'utilise que la donnée en entrée et la clé de la KeyExpansion_table, à chaque nouvelle ronde.

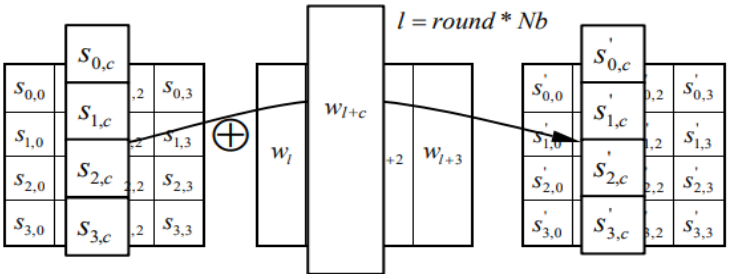


Figure – 13 : Représentation de la fonction ARK

i. *KeyExpansion_table*

L'entité de ce bloc, illustrée ci-dessous dispose d'une entrée pour le numéro de ronde, **round_i**, codé sur 4 bits. L'architecture contient une constante déclarée comme étant tableau de longueur 11 et dont les éléments sont de type bit128. La sortie de ce composant, **expansion_key_o** du même type[18], correspond à la valeur de la clé à utiliser pour la ronde courante. Les fichiers de description et de test utilisés, nous ont été fournis, après avoir modifié légèrement le second fichier.

[15] Avec deux indices I et J entre 0 et 3.
[16] Il est concevable de se limiter à I pour les 16 octets : **data_i**(I/4)(I mod 4).
[17] La première ronde, l'état round10 la machine d'état inverse.
[18] Type d'un élément du tableau qui est bit128.

Figure – 14 : L'entité du composant KeyExpansion_table

Le résultat du test bench, ci-dessous, pour la clé initiale et la dernière clé permet de conclure au fonctionnement de ce bloc. La compatibilité entre les types de l'InvAddRoundKey va être assurée par le composant de conversion[19] bit128 vers type_state.

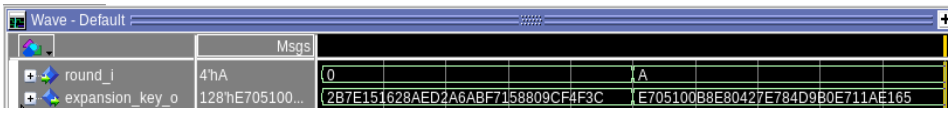


Figure – 15 : Test bench de KeyExpansion_table

ii. *InvAddRoundKey*

La fonction ARK inverse s'appuie sur l'opération xor entre deux tableaux passés en entrées. Le premier correspond à la donnée entrante et le deuxième est la clé issue de la fonction précédemment citée, et qui sont tous les deux du type type_state[20]. L'entité comporte 2 entrées et 1 sortie qui sont toutes du même type. Dans l'architecture, il suffit d'effectuer le xor entre les entrées, en octet par octet[21].

Figure – 16 : L'entité du composant ARK inverse

Nous testons le composant dans le test bench ci-dessous avec la donnée chiffrée initialement en entrée et nous obtenons les résultats pour la première ronde[22], numéro 10, en sortie de l'ARK inversé.

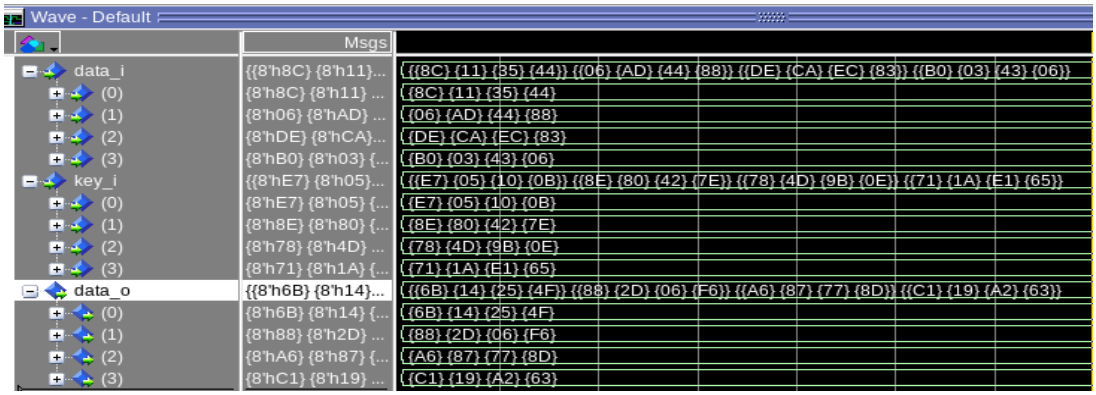


Figure – 17 : Test bench de InvAddRoundKey

4. La fonction MixColumns inverse

L'opération de mixing des colonnes (Mix Columnes noté MC) inverse, représentée dans la figure 18, est la seule à être désactivée lors d'une ronde[23]. Elle repose sur un formalisme mathématique que nous détaillons avant d'établir le circuit, ce qui simplifie davantage l'élaboration du bloc InvMixColumns.

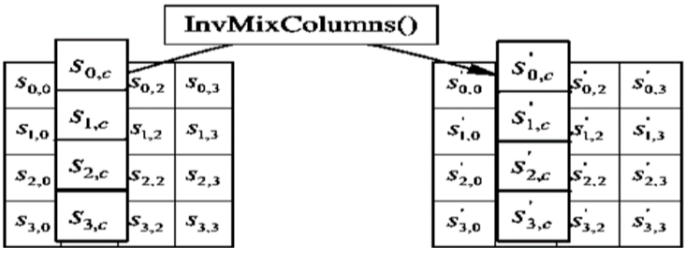


Figure – 18 : Représentation de la fonction MC inverse

[19] Le composant Conv placé au niveau de l'hiérarchie supérieure.

[20] Compte tenu de l'équivalence entre les types type_state et key_state du package.

[21] Avec deux boucles imbriquées de taille 4 avec *end generate* en fin de boucle.

[22] Également, les premiers résultats de ce 3ème composant sont disponibles en annexe.

[23] La dernière ronde, l'état round0 la machine d'état inverse.

i. Étapes calculatoires de InvMixColumns

Pour toutes les colonnes de la donnée il faut réaliser le calcul matriciel colonne par colonne[24] avec la matrice ci-dessus. Il suffit d'avoir les résultats de la multiplication par 2, 4 et 8 pour les 16 octets de la matrice, ce qui nous donne au total 48 octets à enregistrer et affecter dans l'ordre de la multiplication matriciel.

$$\begin{pmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{pmatrix}$$

Figure – 19 : Matrice pour l'opération MC inverse

Tous les calculs à effectuer sont décomposables en des multiplications successives par des puissances de 2. Pour la décomposition des lignes de la matrice, nous obtenons les quartets suivants :

$0b'e_2 = 8_2 \oplus 4_2 \oplus 2_2$ $\quad$ $0b'b_2 = 8_2 \oplus 2_2 \oplus 1_2$

$0b'd_2 = 8_2 \oplus 4_2 \oplus 1_2$ $\quad$ $0b'9_2 = 8_2 \oplus 1_2$

Pour la multiplication des octets par 2[25], nous distinguons 2 cas :

- Si le bit de poids fort est à 0, nous réalisons que le décalage les bits de l'octet vers la gauche d'une unité.
- Sinon, nous ajoutons au décalage, une opération de xor ⊗ avec le nombre 0b'100011011 qui est le tronquée 0x11B et qui vaut sur 9 bits 0b'100011011 correspondant au polynôme $x^8 + x^4 + x^3 + x^1 + x^0$ qui un est polynôme irréductible $GF(2^8)$ et qui module les multiplications.

Les deux cas peuvent être regroupées en une seule opération : le xor de l'octet avec le même nombre 0x1B mais remplacent les bits à 1 par le bit de poids fort de cet l'octet. Dans le cas où ce bit est à 0 le résultat du xor est l'octet en entrée, puisque le 0 est l'élément neutre du xor. Dans l'autre cas, le résultat va être celui du xor entre l'octet et 0x1B. Ainsi, pour un octet d_i tel que d_i_k et le k[ème] bit, $0 \leq k \leq 7$, l'octet d_ix2 du produit par 2 :

$$d\_ix2 = 0b'd\_i_7\ d\_i_6 d\_i_5 d\_i_4 d\_i_3 d\_i_2 d\_i_1 d\_i_0 \otimes 0b'000 d\_i_7 d\_i_7 0 d\_i_7 d\_i_7$$

Nous avons utilisé l'opérateur de concaténation & pour lier les bits de l'octet d_i et ceux à zéro du nombre 0b'00011011. Puisque la même multiplication matricielle est effectuée pour chaque colonne de la matrice d'origine, nous avons créé un couple/entité architecture qui réalise les calculs de l'étape MC inverse décrite ci-dessus.

ii. InvMixColumn

L'entité de ce composant, représentée dans la figure 20, possède une entrée et une sortie de type row_state[26], qui est équivalent au type column_state. Les multiplications des octets en entrée par des puissances croissantes de 2 sont stockées dans des signaux, déclarés dans l'architecture, qui sont de type column_state[27]. Ils sont par la suite combinés avec le xor en fonction[28] de la ligne de la matrice ci-dessus, et affectés aux octets de la colonne en sortie. Les calculs préliminaires sur l'entrée, qui sont les trois multiplications, peuvent être effectués sur tous les octets de la colonne en utilisant une boucle de taille 4. Cette dernière permet aussi de rediriger les combinaisons des signaux de multiplication vers la sortie. Il suffit de décaler l'ordre de la première ligne de la matrice en modifiant l'octet des signaux avec l'opération de l'addition modulo 4. Pour le port de sortie, nous nous sommes contentées d'affecter un par un les éléments de la colonne sortante. Ce bloc limite les sources d'erreur puisque les calculs arithmétiques sont conduits par colonne séparément de la donnée[29].

Figure – 20 : L'entité du composant MC[30] inverse

[24] Préférentiellement en vue de simplifier l'opération.

[25] Ce qui équivaut au décalage à gauche d'un bit.

[26] La vue des ports comme lignes car ce bloc traite la donnée ligne par ligne.

[27] L'opération mathématique agit, en interne du composant, sur des colonnes.

[28] Cela dépend aussi de l'indice de l'élément de la colonne.

[29] C'est une matrice bidimensionnelle.

[30] Le mixing inverse est pour une seule colonne.

Le test bench ci-dessous nous permet de vérifier le calcul sur une colonne avant de le généraliser dans la fonction qui agit sur toute la donnée entrante. La colonne en entrée est la ligne de la ronde 9, et les signaux du produit avec 2, 4 et 8 combinés fournissent la sortie attendue.

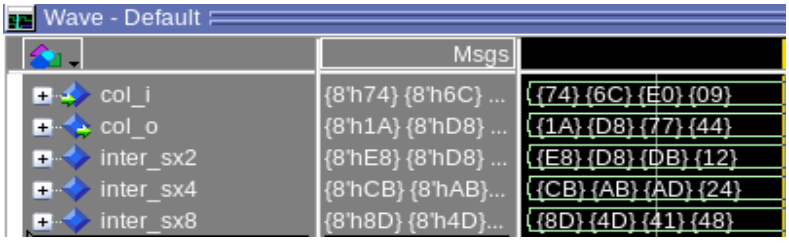


Figure - 21 : Test bench de InvMixColumn

iii. *InvMixColumns*

La fonction de MC inverse est réalisée assurée par l'instanciation 4 fois le composant précédent, cité dans la configuration, à l'aide de la généricité. L'entité, dans la figure 22, comporte les deux ports d'entrée et sortie des données ainsi qu'une entrée d'activation, de type std_logic. Dans l'architecture, les mots clés *when* et *else* permettent de modéliser[31] le multiplexeur nommé Mux non instancié illustré ci-dessous.

Figure – 22 : L'entité du composant MC inverse

Nous utilisons deux signaux, dont un est temporaire[32] et un autre permanent utile calculs et sert d'entrée pour le multiplexeur. Le premier permet de lier l'entrée des 4 instances[33] au colonnes de la donnée en entrée, et le deuxième, qui récupère le résultat de chaque instance, est affecté à la sortie si le bit **enable_i** est à 1.

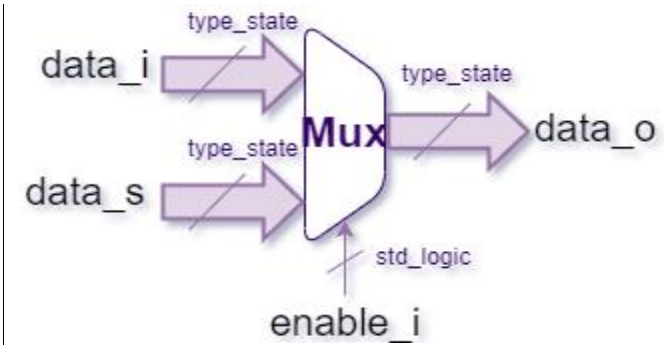
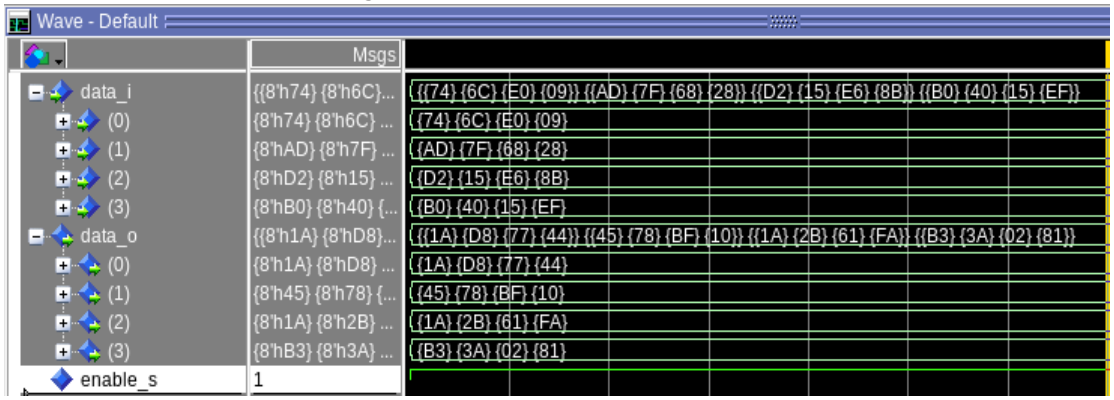


Figure – 23 : La représentation des ports du multiplexeur

Nous vérifions le fonctionnement du composant MC inverse, grâce au test bench ci-dessous qui utilise les octets de la matrice d'entrée à la ronde 9.

Figure – 24 : Test bench de InvMixColumn



[31] Ceci permet une meilleure synthèse du Mux comparée à la description par un processus.

[32] Afin de manipuler les données sous forme de colonnes d'octets par transposition.

[33] Elles sont instanciées à partir du InvMixColumn.

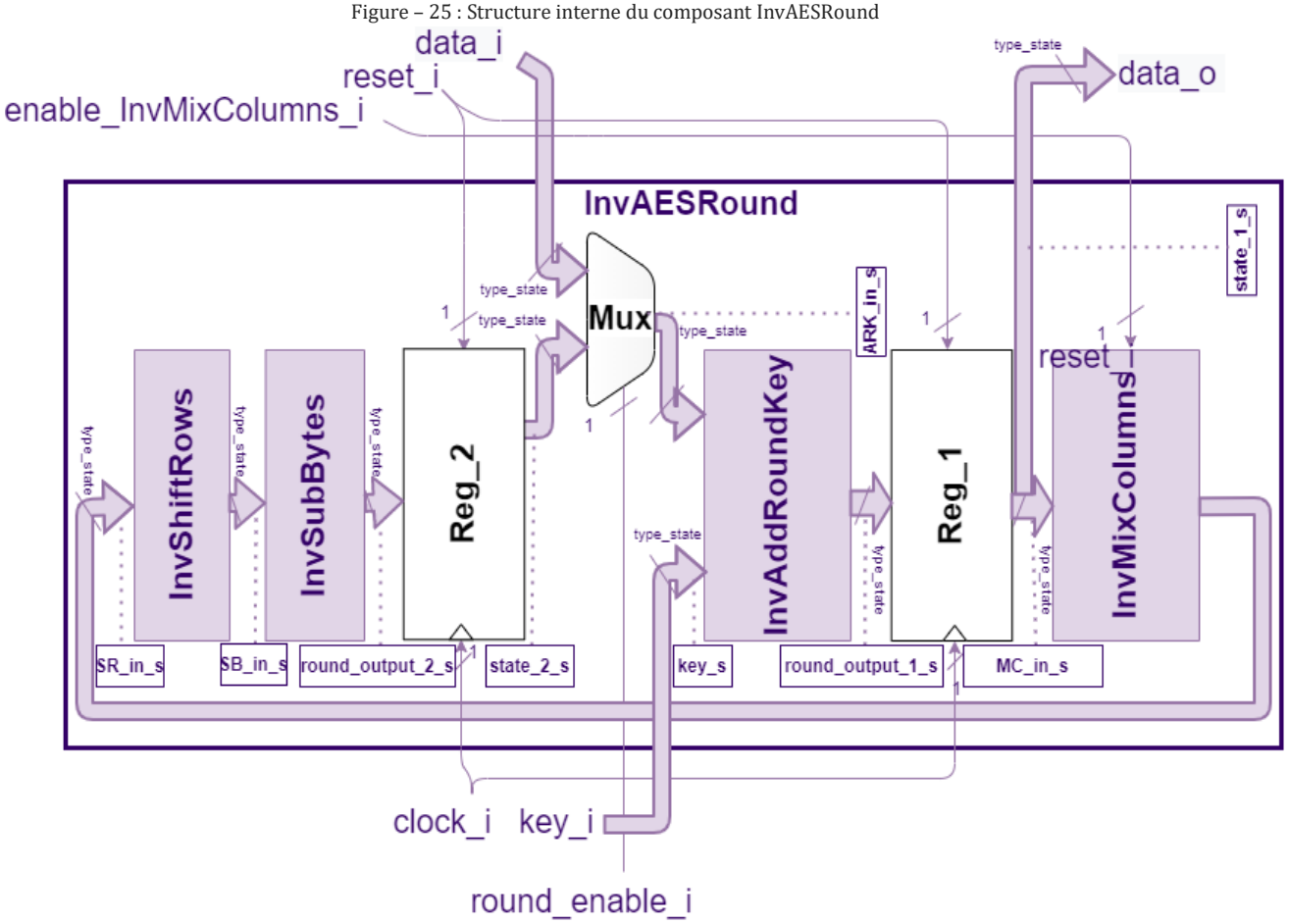
C. Ronde de l'AES inversé

1. Implémentation de la ronde inversée

Le composant InvAESRound modélise une ronde de calcul, en intégrant les 4 fonctions précitées. En s'appuyant sur la représentation[34] de sa structure ci-dessous, nous notons que ce composant est conçu pour traiter toutes les types de rondes : initiale n°10, intermédiaire n°9 au n°1, finale n°0. Le registre intitulé Reg_1 intercalé entre les blocs InvAddRoundKey et InvMixColumns permet de mettre à jour et de sauvegarder le résultat du premier bloc à chaque front montant d'horloge. Le deuxième[35] registre Reg_2 qui est placé juste après le bloc InvSubBytes sert d'élément synchrone dont le rôle est de restituer la sortie de ce bloc un coup d'horloge plus tard. Ceci nous a facilité l'obtention des résultats à chaque état de la FSM étudiée ci-après, et nous notons qu'il rajoute un cycle d'horloge au temps de calcul total. L'entrée de InvAddRoundKey, est issue de la sortie d'un multiplexeur, en fonction du signal **round_enable_s**, et dont les entrées sont le texte à déchiffrer ou bien la sortie du Reg_2. Alors que l'entrée du bloc, InvMixColumns, reçoit la sortie de Reg_1, et qui nous permet de prélever le résultat de calcul d'une ronde.

Nous avons récupéré des documents du projet, un exemple de registre sous la forme de processus auquel l'étape de la mémorisation et les commentaires pertinents ont été ajoutés. Comme le registre est utilisé à 2 reprises, nous avons déclaré et instancié 2 fois le composant Reg_i, pour les données type_state[36], pour mieux adhérer à la structure dans la figure 25. Les deux registres sont modélisés par deux processus explicites séquentiels dont la liste de sensibilité contient les entrées de l'horloge et du reset active à l'état haut. L'entrée de réinitialisation asynchrone remet chacun des octets des sorties des registres à la valeur 0 grâce au mot clé *others* imbriqué 3 fois.

Figure – 25 : Structure interne du composant InvAESRound



[34] Seul le bloc Mux à contour arrondi et dont le contenu est sans couleur, n'est pas déclaré ni instancié.

[35] Ce registre n'est pas présent dans les documents explicatifs du sujet du projet.

[36] Un registre 128 bit, RegSel, a été conçu pour le composant final InvAES.

L'entité de InvAESRound comporte 4 entrées qui sont de type std_logic, ainsi que 2 entrées et 1 sortie qui sont respectivement la clé et les données, en type_state, à l'entrée et à la sortie de la ronde. Les entrées **clock_i** et **reset_i**, utiles pour les registres, vont découler du composant global, et les entrées **round_enable_i** et **enable_InvMixColumns_i** vont commander, respectivement, le multiplexeur et InvMixColumns. Dans la partie déclarative de l'architecture, nous incluons les 4 blocs élémentaires et le bloc du registre, qui sont ajoutés dans la configuration, avec les mots clés *use entity*. Dans la description, nous câblons les 6 instances en utilisant 1 signal par couple[37] où les registres Reg_1 et Reg_2 sont deux instances du même composant Reg_i. Nous obtenons au total 9 signaux, représentés par des rectangles liés avec des pointillés en tenant compte des signaux **state_k_s** et **round_output_k_s** aux ports d'entrée et de sortie de Reg_k[38].

2. Tests bench

Nous avons vérifié le fonctionnement du composant en envisageant les trois cas de rondes dans les tests bench en figure 26 et 27. Le premier test où seul l'entrée **enable_InvMixColumns_i** est à 0 pour stimuler la ronde 0, avec la valeur de la clé correspondante à cette ronde. Et le second test, avec les entrées **round_enable_i** et **enable_InvMixColumns_i** à 0 en cas de la ronde 10, puis ces dernières sont mises à 1 pour tester les deux premières rondes intermédiaires[39] où tous les blocs sont en utilisations.

Figure – 26 : Premier test bench de l'InvAESRound

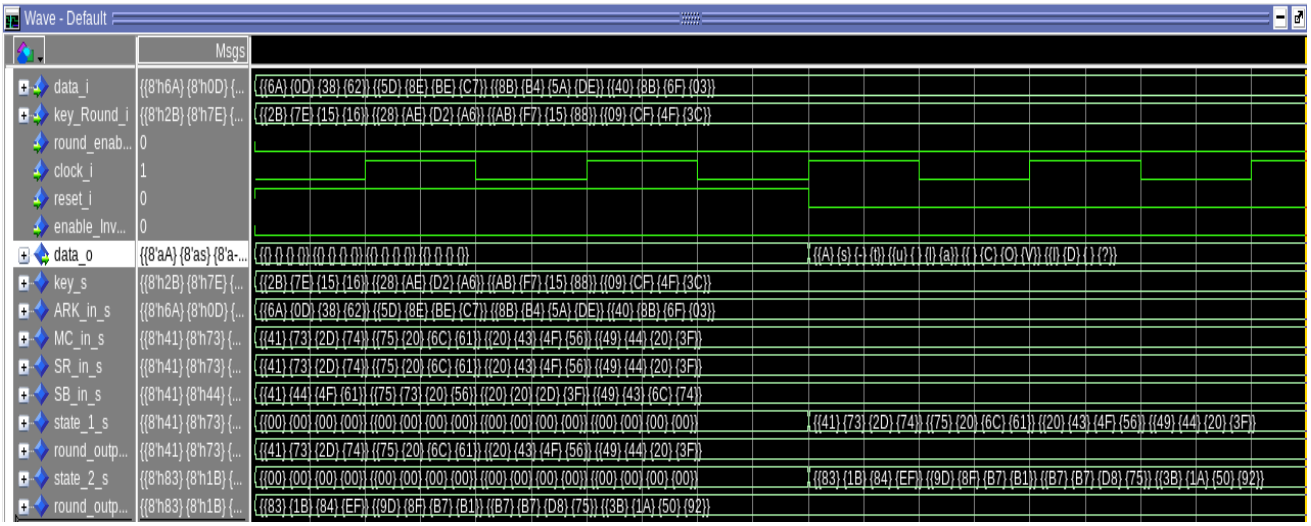
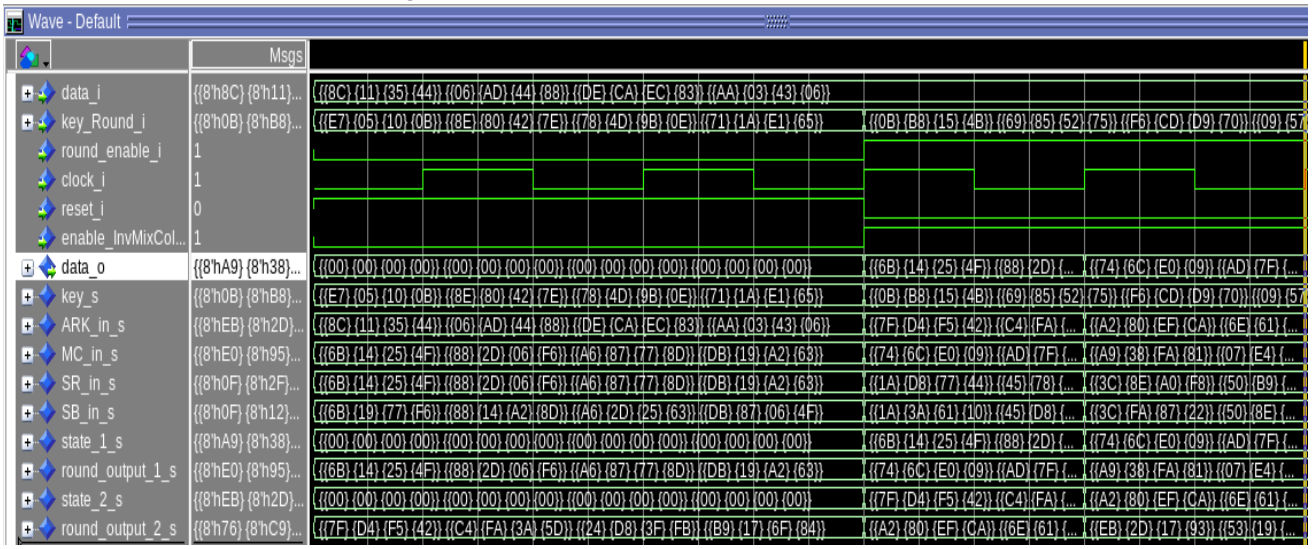


Figure – 27 : Deuxième test bench de l'InvAESRound



[37] Couple vaut (l'entrée de l'instance d'un bloc/la sortie d'une d'instance du bloc suivant).

[38] Avec k dans {1, 2}.

[39] 9ème et 8ème round dont les résultats sont en annexe.

D. Machine d'état inverse

1. Implémentation de la machine d'état

La machine FSM (Finite State Machine) inverse pilote les composants InvAESRound, RegSel[40] et Counter[41], et permet de les relier ensemble. Grâce à ce bloc, Nous pouvons maintenir un bon cadencement et une temporisation adéquate des opérations de déchiffrement. L'entité de la FSM dans la figure 28 ci-dessous contient, 4 entrées et 6 sorties, les entrées sont **round_i**, égale au numéro de la ronde de type bit4 qui est issu du Counter, et **start_i** de type std_logic qui permet de sortir de l'état initial[42] et de passer aux autres états, ainsi que deux entrées, **clock_i** et **reset_i** du même type. Les sorties de la FSM sont toutes du type std_logic, dont deux qui sont **init_cpt_o** et **enable_cpt_o** permettent de commander le Counter. Les deux sorties qui servent d'entrée pour l'InvAESRound sont **round_enable_o** et **enable_InvMixColumns_o**. Les deux autres sorties qui sont **aes_done_o** et **enable_output_o**, ne sont activés qu'après la dernière ronde pour respectivement signaler la fin du calcul et fournir le résultat à la sortie.

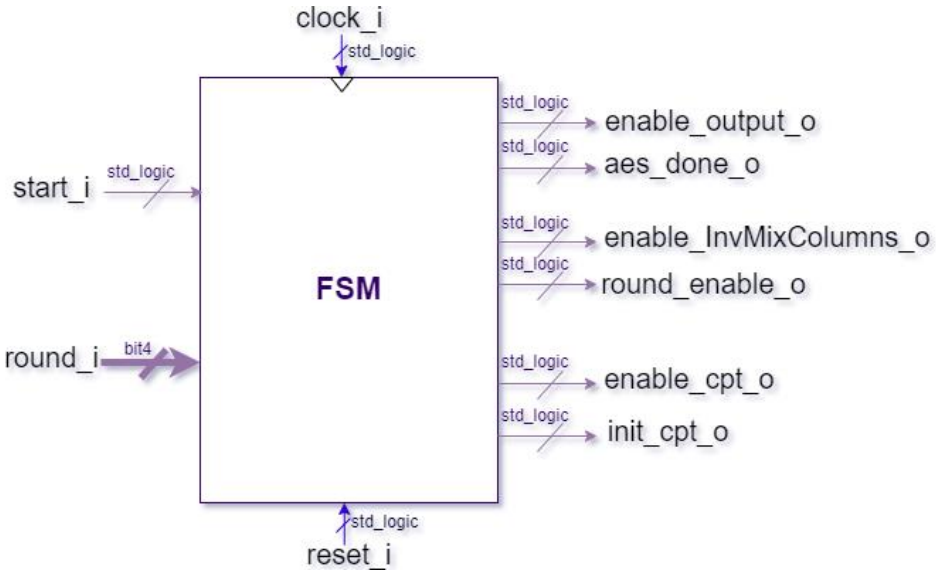


Figure – 28 : L'entité du composant FSM inverse

Nous avons opté pour une FSM de type machine de Moore avec des sorties synchrones et 3 processus séparés[43], qui contient 5 états : init, round10, round9_0, round0, et fin. L'architecture contient la définition du type des états qui est énuméré en ces 5 états, ainsi que la déclaration des deux signaux, l'état[44] présent et futur qui sont du type prédéfini[45]. La partie descriptive de l'architecture se compose, pour des raisons de lisibilité, de 3 process explicites, 2 process combinatoires et 1 séquentiel. Ce dernier permet, si l'entrée de **reset_i** n'est pas à 1, d'actualiser l'état présent à chaque nouveau front montants d'horloge, avec une liste de sensibilité qui contient les entrées de **clock_i** et **reset_i**. Le deuxième process effectue le calcul de l'état futur en fonction de l'état présent et des entrées, qui figurent naturellement dans sa liste de sensibilité. Le dernier process qui est aussi combinatoire s'occupe de l'affectation de la sortie à partir uniquement de l'état présent. L'absence des entrées dans la liste de sensibilité de ce process découle de notre choix, non imposé par le sujet[46], du type de FSM. En tenant compte du nombre d'état, les deux derniers process doivent contenir 5 cas dans les conditions décrits par la syntaxe *when*…⇒

2. Description des états de la FSM

Tout d'abord, avant que les sorties de la FSM ne soient calculées, elles sont toutes initialisées à 0.

[40] Le composant registre de sélection.
[41] Le composant compteur.
[42] L'état qui est nommé init.
[43] La description est plus étendue sauf qu'elle plus compréhensible.
[44] Ou de façon plus ou moins équivalente la ronde.
[45] Le type énuméré et intitulé state.
[46] FSM de Moore était conseillée lors des cours et du projet.

Les 5 états[47] de la FSM, qui sont décrits par la figure 29[48] sont dans l'ordre :

a. *Etat init :*

C'est l'état par défaut de la FSM si le **reset_i** est à 1 et dans lequel seules les sorties dont le suffixe est **cpt_o** qui correspondent au Counter sont activées. La condition sur l'entrée du passage à l'état suivant est que **start_i** soit égale à 1. Notons que la sortie **enable_cpt_o** doit toujours maintenues à 1, pour la décrémentation dans tous les états excepté le dernier.

b. *Etat round10 :*

Les sorties de **round_enable_o** et **enable_InvMixColumns_o** étant à 0, le calcul de la première ronde est réalisé en utilisant la donnée à déchiffrer en entrée du composant InvAddRoundKey et durant laquelle le bloc InvMixColumns est désactivé. La transition à l'état futur est conditionnée par la décrémentation du Counter.

c. *Etat round9_1 :*

Durant les rondes intermédiaires, l'InvMixColumns est actif et la sortie prise en compte est celle de l'InvSubBytes. Si **round_i**, le numéro la ronde, est égale à 0b'0000 sur 4 bits, l'état futur est la ronde finale.

d. *Etat round0 :*

La sortie **enable_InvMixColumns_o** doit être remise à 0 puisque lors de la dernière ronde la transformation du InvMixColumns n'est pas appliquée, tandis que les tous autres blocs doivent fonctionner. L'état suivant est atteint sans conditions sur les entrées.

e. *Etat fin :*

Les sorties de la FSM propres au compteur et au bloc InvAESRound repassent à 0, alors que celles de la terminaison du calcul sont mises à 1. Il s'agit des sorties **aes_done_o** et **round_output_o** sollicitées que si la donnée est déchiffrée[49] afin de l'orienter vers la sortie[50] du RegSel. La condition pour passer à l'état initial est que l'entrée **start_i** soit à 1 ce qui permet de traiter éventuellement une autre donnée chiffrée en entrée.

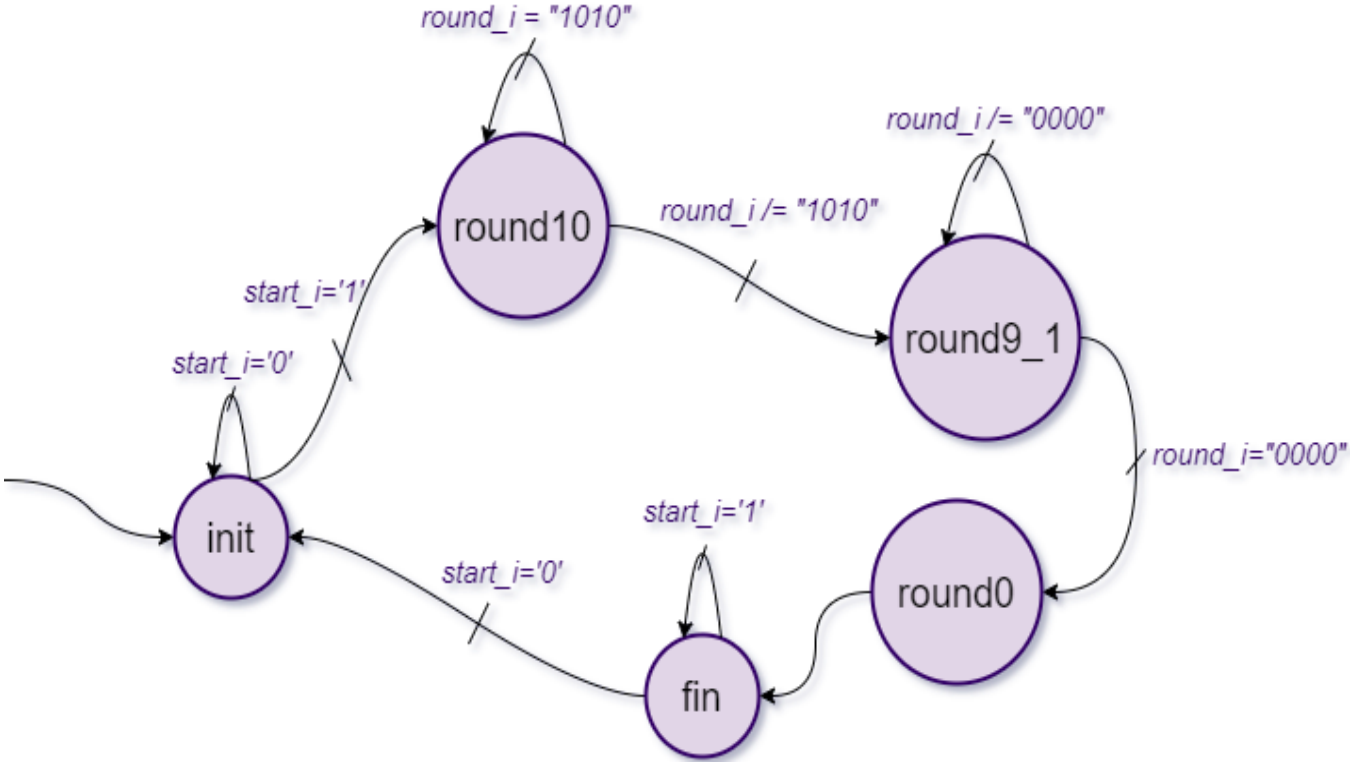


Figure – 29 : Représentation des états de la FSM[51] inversé

[47] Ils sont représentés à l'intérieur les disques colorés.
[48] Les transitions sont conditionnées par les clauses en italique.
[49] Peu importe la pertinence du résultat.
[50] En pratique, le résultat prend un coup d'horloge de plus.
[51] Son type impose, visiblement, plus d'états qu'une FSM de Mealy.

Nous avons effectué deux tests bench du composant dans deux environnements assez similaires, qui se situent dans deux fichiers *_tb.vhd* distincts. Lors du premier environnement en figure 30 dans lequel nous testons seulement la FSM, l'entrée **round_i** est générée par un processus qui simule la présence du composant Counter. Pour le deuxième test en figure 31, le Counter est ajouté à la configuration et une instance de ce dernier est déclarée dans l'architecture, en plus des signaux de liaisons : **init_cpt_s**, **enable_cpt_s**, et **round_s**. Les deux tests bench, où les sorties s'activent aux états adéquats, sont conduits dans les mêmes conditions, autrement dit l'entrée d'horloge, de reset, et de start sont identiques.

Figure – 30 : Test bench de FSM_InvAES

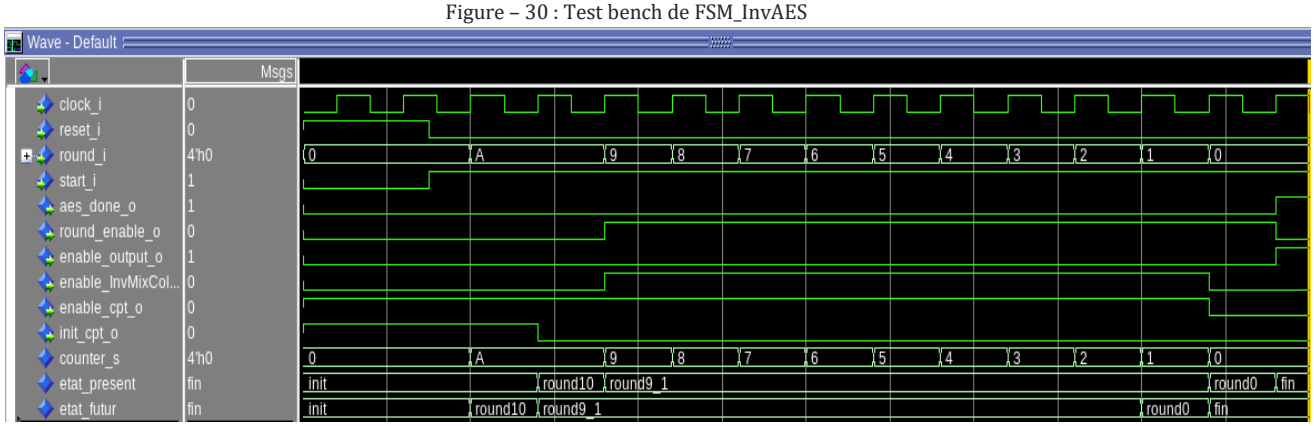
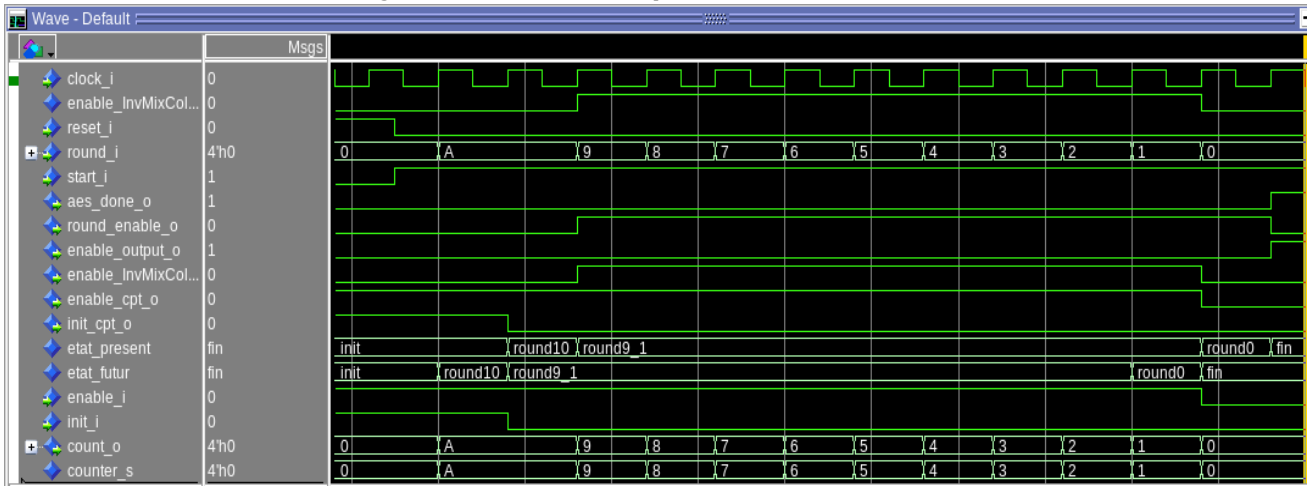


Figure – 31 : Test bench du Compteur et de la FSM_InvAES



E. Composants en Top Level de l'AES

Nous allons décrire dans ce qui suit les blocs qui sont primordiales au fonctionnement du déchiffrement de l'AES, et qui sont instanciés dans le circuit global c'est-à-dire dans le même niveau hiérarchique que ce dernier.

1. Counter

Ce composant nous a été fourni avec les fichiers de base pour le projet. Nous avons modifié l'entité, représentée ci-dessous, ainsi que le processus qui modélise le compteur. Nous avons ajouté l'entrée **ini_t** qui initialise la valeur de la sortie à 10 si elle est à 1 ou bien pour mettre à jour sa valeur en le décrémentant à[52] tout front montant d'horloge. L'entrée de reset permet de réinitialiser le compteur à la valeur 0 de façon asynchrone, indépendamment des entrées précédentes. Le type de la sortie étant bit4, nous avons converti le signal, de type natural, déclaré dans l'architecture grâce aux fonctions déjà utilisées dans les blocs élémentaires[53]. Sa sortie, qui est la valeur du numéro ronde codée sur 4 bits, constitue l'entrée **round_i** des composants KeyExpansion_table, et FSM. Cette dernière permet de fournir les deux entrées **init_i** et **enable_i** qui sont issues respectivement des sorties **init_cpt_o** et **enable_cpt_o** de la FSM.

[52] À condition que le compteur soit activé avec **enable_i** à 1.

[53] La S-Box à titre d'exemple.

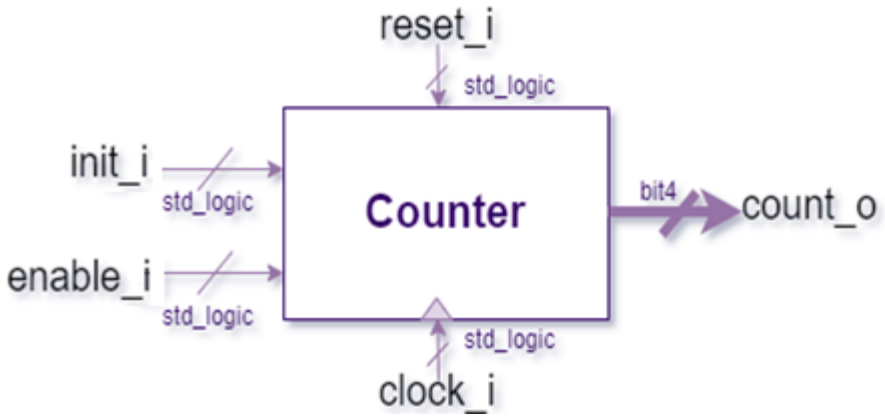


Figure – 32 : L'entité du composant Counter

Le test bench en figure a été adapté aux changements effectués, nous remarquons que la sortie du compteur continue à se décrémenter, tant que la sortie est non nulle. Une fois le signal **count_s** représentatif de la sortie atteint 0, le process, étant explicite, s'achève avec le mot clé *end process*.

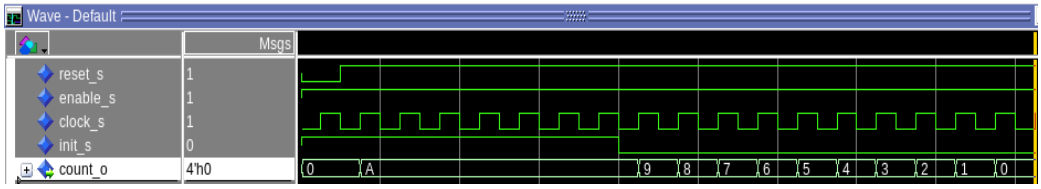


Figure – 33 : Test bench du Counter

2. RegSel

Il s'agit d'un composant qui modélise un registre qui sert à enregistrer le résultat final du déchiffrement et permet d'en disposer à la sortie du circuit final. Son entité, qui est illustrée ci-dessous, comporte 4 entrées, dont 3 sont de type std_logic et la 4ème, du type bit128, est la donnée à l'issuee de chaque ronde. La sortie déchiffrée a le même type que l'entrée précédente. Si l'entrée **enable_i** est à 1, la sortie correspond à la valeur de l'entrée à chaque front montant d'horloge, sinon la sortie garde la même valeur qui est mémorisée. L'entrée **reset_i** asynchrone permet de réinitialiser la valeur du registre à une donnée qui ne contiennent que des octets nuls d'une manière analogue à celle des registres[54] Reg_1 ou Reg_2. Les entrées **reset_i** et **clock_i** proviennent du bloc InvAES, l'entrée d'activation, **enable_i**, est fournie par la FSM, et la donnée en entrée est le résultat du calcul des rondes.

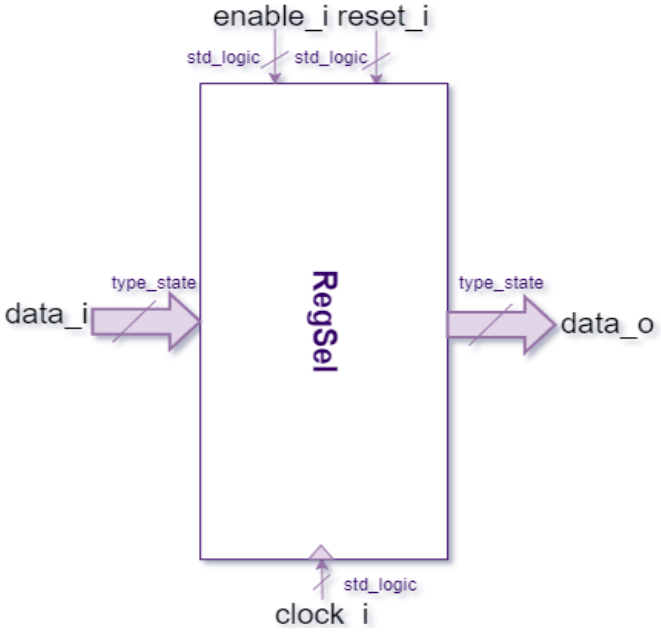


Figure – 34 : L'entité du composant RegSel

[54] En tenant compte du type de données.

Un test bench est proposée ci-dessous, avec deux valeurs différentes en entrée, notons que la donnée en sortie du registre n'est mise à jour à chaque nouveau front d'horloge, à condition que **reset_i** soit à 0 et que le registre soit actif par **enable _i**.

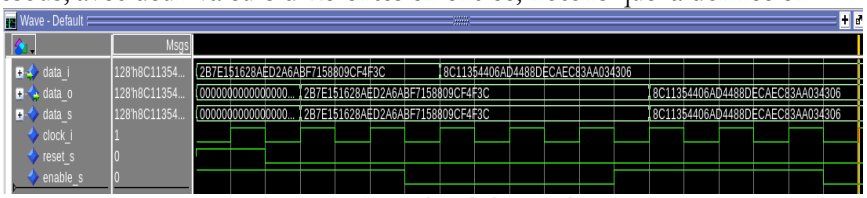


Figure – 35 : Test bench de RegSel

3. Conv & InvConv

Les fonctions de conversion sont assurées par ces composants, dont l'un peut être déduit de l'autre, d'une façon symétrique. L'entité proposée qui figure ci-dessous à gauche est celle de la conversion dans le sens direct, du type bit128 au type_state, noté Conv. La deuxième est relative à la conversion inverse dans le sens, type_state en entrée et bit128 en sortie, abrégée InvConv. L'entrée du Conv est la donnée ou[55] la clé de type bit128 correspondant respectivement à la donnée à déchiffrer et à la sortie de la table des clés. L'entrée du InvConv ne peut être que la donnée en sortie du calcul de chaque ronde de l'InvAESRound.

Figure – 36 : L'entité du composant Conv

Figure – 37 : L'entité du composant InvConv

Les test bench en figure 38 et figure 39 sont ceux des bloc Conv et InvConv respectivement. Les 128 bits des entrées des deux blocs sont identiques, mais ils sont de type différent. Nous vérifions leurs fonctionnements exacts avant de les intégrer dans le dernier bloc.

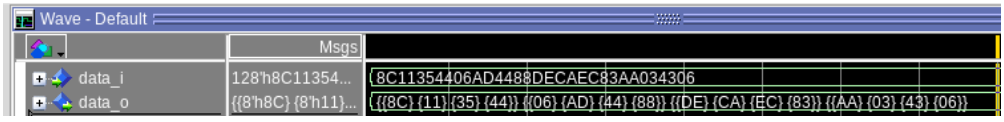


Figure – 38 : Test bench de Conv

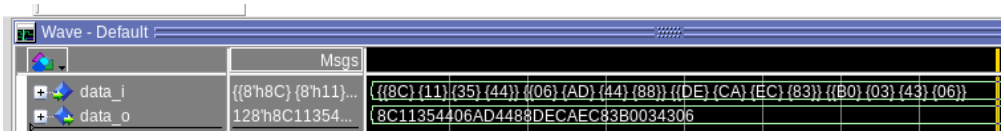


Figure – 39 : Test bench de InvConv

F. Implémentation du bloc AES inverse

1. Couple entité/architecture

Le composant final InvAES réalise la fonction de base du circuit qui est le déchiffrement d'une donnée, texte claire, qui a été chiffrée, après 11 rondes et avec une clé initiale de 128 bits, par le protocole AES. L'entité de l'InvAES est dotée de 4 entrées et de 2 sorties. La première entrée est la donnée à déchiffrer du type bit128, les entrées restantes : **clock_i**, **reset_i** et **start_i** sont tous du type std_logic et servent respectivement d'horloge, d'entrée de réinitialisation et de démarrage du calcul de l'InvAES. La sortie principale est celle de la donnée déchiffrée du type bit128, la deuxième sortie qui est du type std_logic, passe à 1 pour indiquer que les opérations calculatoires du déchiffrement sont abouties et que la donnée est disponible à la sortie du registre RegSel. Nous citons les blocs InvAESRound, FSM, Counter, KeyExpansion_table, Conv, InvConv, et RegSel dans la partie déclarative de l'architecture. Notons qu'il faut les mentionner dans la configuration du composant final autant fois que nécessaire[56]. Dans la description de l'architecture, nous instancions les composants déclarés, et nous réalisons le câblage grâce à 11 signaux. Les signaux d'horloge et de reset sont communs aux blocs : InvAESRound, FSM, Counter, et RegSel qui les utilisent.

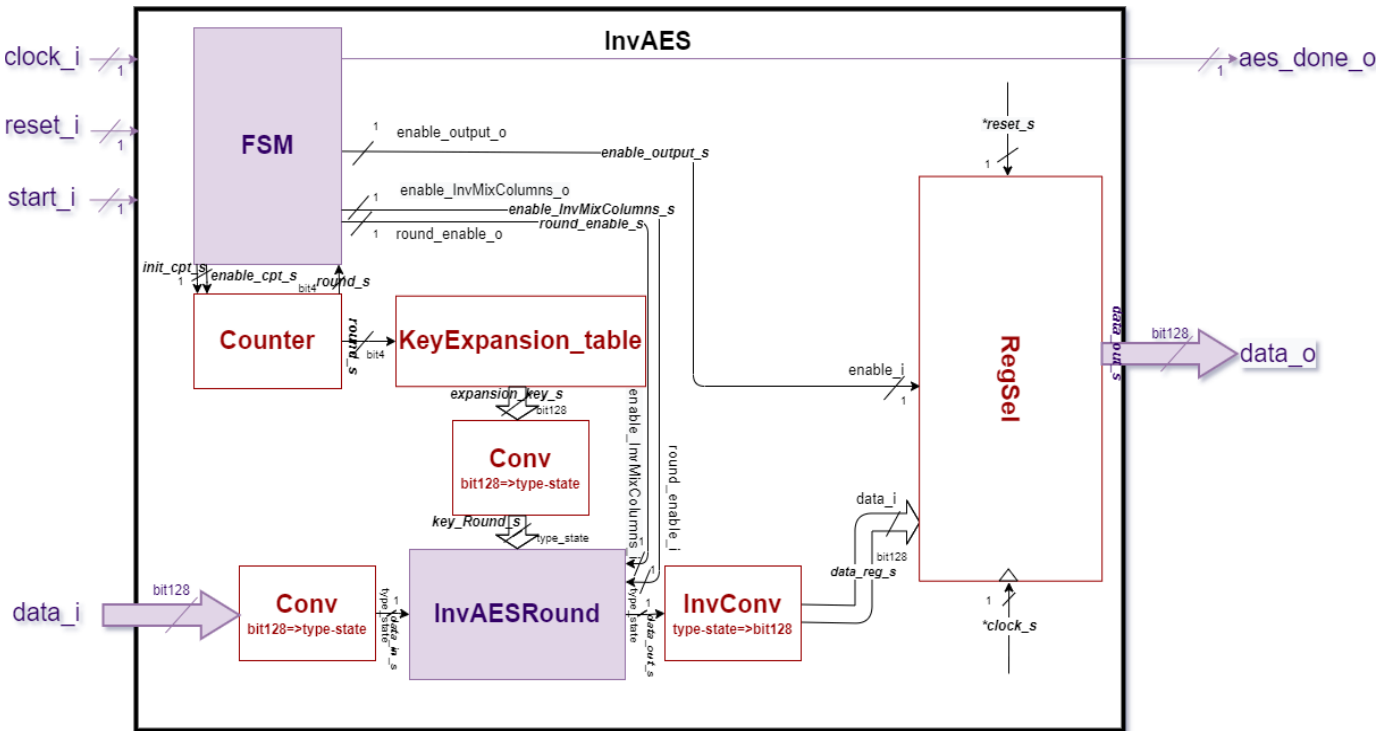
[55] La conversion directe est utilisée deux fois dans le circuit final.

[56] 8 fois pour les 8 instances utilisées.

2. Circuit final

Nous représentons, en figure 40, la totalité de l'entité et de l'architecture de l'InvAES, en correspondance avec le paragraphe précédent. Nous prenons en compte l'entièreté des composants[57] évoqués antérieurement qui figurent avec leurs ports d'entrée et de sortie. Les liaisons[58] entre les ports de ces composants, qui sont réalisées par des signaux, sont représentées en gras et en italique sur les flèches d'entrée et de sortie des composants.

Figure – 40 : Circuit du couple de l'entité/l'architecture du composant InvAES



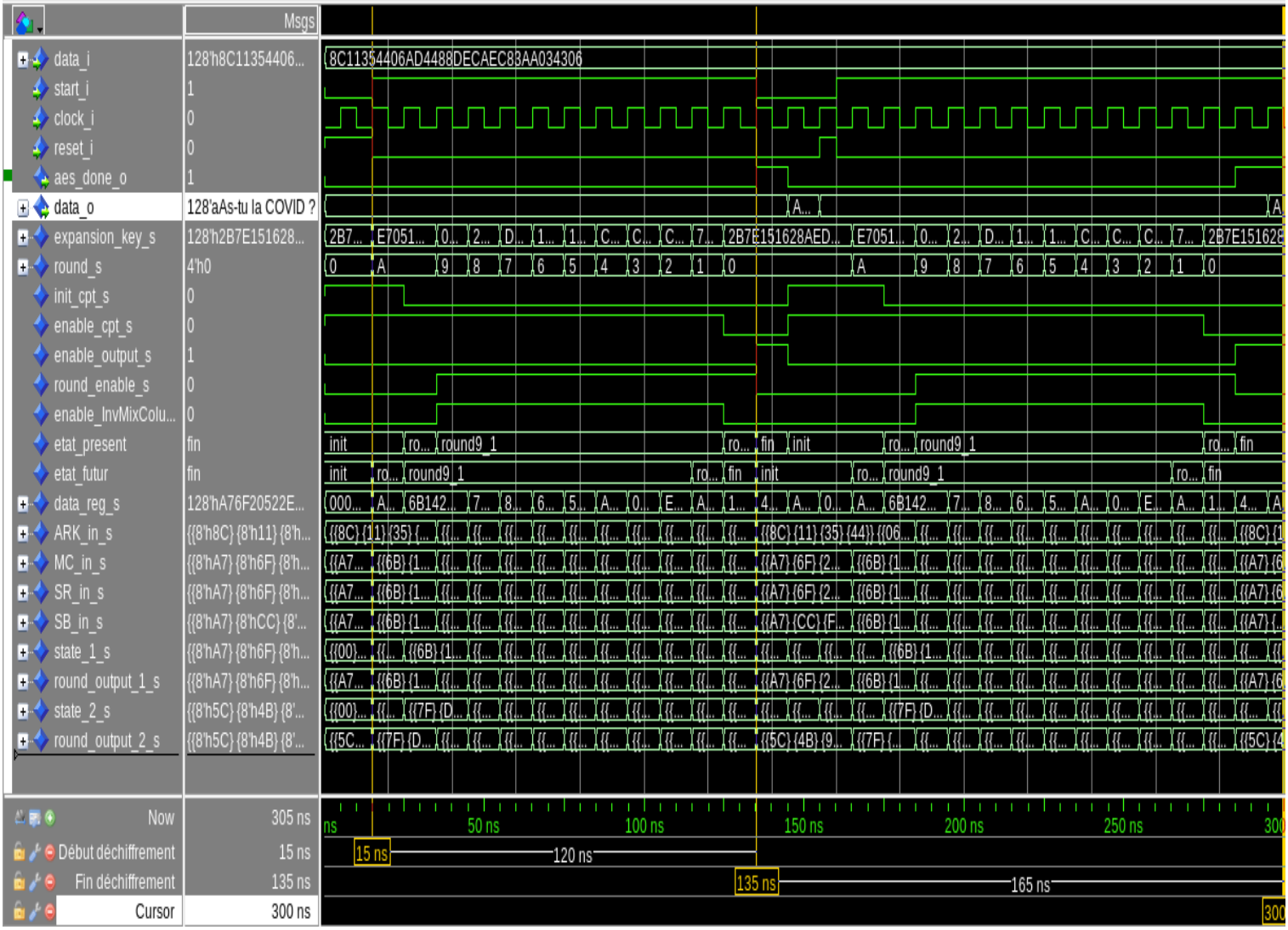
3. Test bench

Nous avons réalisé plusieurs tests bench, avant d'aboutir au résultat représenté dans la figure 41, Les valeurs des données statiques telles que la donnée[59] à déchiffrer et le reset[60] ont été fixés par les documents supports du projet. Le calcul est effectué deux fois à la suite avec la même valeur de **data_i**, ce qui nous facilite la simulation des opérations de déchiffrement successifs. Chaque calcul s'achève au bout de 13 coups d'horloge pour l'exécution des 11 rondes de calcul, la sauvegarde des résultats dans les registres Reg_1 et Reg_2 et la mise à jour de la sortie du registre RegSel, ce qui correspond, avec une période de l'horloge de 5 ns, à 130 ns à peu près. Le port d'entrée du DUT **start_i** est considéré à 1 une fois que le **reset_i** n'est plus actif[61], et les ports de sortie, **aes_done_o** et **round_output_o** sont à 0 tant que le déchiffrement n'est pas abouti. La réussite de l'InvAES est conditionnée par l'exactitude et la justesse de tous les composants détaillés dans les parties précédentes.

Nous avons ajouté au chronogramme du test ci-dessous, les signaux utiles provenant de tous les composants instanciés et en particulier la FSM et l'InvAESRound. Nous remarquons que pour quelques rondes, les sorties des fonctions élémentaires sont correctes : pour la première ronde, valeur hexadécimal A du compteur, dans laquelle seul InvAddRoundKey est utilisé, et pour la dernière ronde, valeur 0 du compteur, où la sortie de RegSel est mise à jour grâce au signal. En analysant les résultats, formatées en ASCII avec le champ qui est libellé Radix, en sortie de l'InvAES, nous pouvons conclure à la conception juste et au dimensionnement correct du circuit.

[57] Ceux qui sont coloriés sont les blocs principaux, les autres en couleur rouge sont les blocs secondaires.
[58] Dont 3 parmi 14, indiquées par un astérisque, servent uniquement pour la compréhension du schéma.
[59] Le texte `0x8c11354406ad4488decaec83aa034306` de 16 octets de 128 bits.
[60] Reset asynchrone actif à l'état haut.
[61] Le **start_i** est effectif au front montant d'horloge dans lequel le **reset_i** est à 0.

Figure – 41 : Test bench final de l'InvAES



III. Limitations, problèmes et améliorations

1. Limitations

Une limitation rencontrée lors de la conception du bloc SR a été découverte et corrigée quelques semaines après l'achèvement des 4 blocs de fonctions élémentaires. Il s'agissait d'une faute logicielle dans le sens de rotation, contraire au sens attendue[62], ce qui était partiellement dû aux recherches personnelles avant les séances encadrées. L'erreur était réglée à l'aide de l'utilisation de deux signaux temporaires. Le premier prenait l'entrée en inversant les indices des lignes et colonnes, et le deuxième stockait les rotations dans le sens alternatif[63] et qui était, une fois inverse, affecté à la sortie.

La seconde limitation était liée à l'ordre des transformations lors du déchiffrement. Une fois l'InvAESRound assemblé, nous avons mené plusieurs tentatives pour ranger les blocs élémentaires dans le même sens que lors du chiffrement. Ces essais étaient voués à l'échec sans l'implémentation de deux opérations additionnelles. La première lors de la fonction KeyExpansion non réalisée, et la deuxième qui était MC inversé à la fin de la dernière ronde. Cette approche est connue sous le nom[64] de « Equivalent Inverse Ciphering ».

[62] De la gauche vers la droite.

[63] De la droite vers la gauche.

[64] Il est abrégé en « EqInvAes ».

Ces deux limitations, quoique simplistes, prouvent que l'algorithme de déchiffrement et son implémentation peuvent être en partie et même entièrement conçus différemment.

Une troisième limitation était dû de l'impossibilité du test du l'AES inversé sur des d'autres données en entrée, comme des textes clairs avec caractères ASCII accentués ou pointés. Cette limitation apparente[65] se justifie par le nombre d'exemples fournis pour tester le circuit final InvAES. Nous avons opéré deux déchiffrements à la suite en utilisant les mêmes[66] données chiffrées.

2. Problèmes

Dans l'ordre chronologique, le premier problème est une confusion lors de la conception de la S-Box par rapport à l'ordre dans lequel il fallait imbriquer les fonctions de conversion vers le type bit8 pour chaque octet du SB inversé. Ceci a été rapidement rectifié une fois que les notes et les du projet et du cours ont été relues et comprises. Un deuxième problème inhérent au non-respect des langages VHDL est celui rencontré lors de l'instanciation d'un composant dans une architecture structurelle. Il s'agissait de la règle, valable aussi pour les fichiers test d'un composant, selon laquelle l'instance du composant doit avoir le même nom que l'entité de ce composant. Ce qui causait l'absence des instances dans le simulateur, une fois les fichiers d'un composant sont compilées et mappées dans les librairies. L'erreur affichée « component is not bound » permettait de se rendre compte de l'erreur et de la corriger assez rapidement.

Le dernier problème n'a eu lieu qu'après la conception du composant de ronde inverse et celle de la machine d'états qui étaient fonctionnels. Une boucle infinie apparaissait lors de l'exécution du test bench associé au InvAESRound, qui a aussi impacté celui de l'InvAES. L'erreur, qui a été découverte tardivement à cause de de l'étape de test bench négligée, était dû au fait que l'entrée et sortie de la ronde étaient liées de la mauvaise façon. Il a fallu donc reprendre la décomposition d'une ronde et câbler de nouveau soigneusement les entrées et les sorties des blocs élémentaires, des registres et du multiplexeur en utilisant des signaux intermédiaires adaptés.

3. Améliorations

Tout d'abord, en premier temps, l'implémentation de la fonction de la KeyExpansion est possible d'après l'algorithme[67] qui génère la table des clés pour l'opération du ARK inverse. Ainsi il faut ajouter une nouvelle entrée au circuit final[68] et qui sera la clé de base[69] de la dernière ronde 11 par exemple, et à partir de laquelle nous pourrons essayer d'en déduire tous les sous clés nécessaires pour les autres rondes. Par ailleurs, l'agrégation des trois composants FSM et Counter était aussi concevable.

En deuxième temps, Le déchiffrement de l'AES[70] peut être implémenté pour une longueur de la clé plus grande, à savoir 24 ou 32 octets avec des modules de conversion et de diversification des clés plus larges. De plus, nous optons le nombre de rondes qui doit aussi être agrandi, en atteignant 12 ou 14 rondes, ce qui garantit un déchiffrement plus compliqué[71].

En troisième temps, une amélioration possible est d'utiliser 4 états au lieu de 5 dans la FSM, ce qui nous permettra de diminuer le temps de traitement et de calcul et d'accélérer davantage le processus de déchiffrement. Afin de se limiter à 4 états, il suffit de réunir deux états en un seul, il s'agit plus précisément de l'état de la ronde finale et l'état fin. Les états de la FSM ronde initiale et intermédiaire sont invariables. En ce qui concerne l'état init, les sorties de la FSM qui contrôlent le Counter doivent repasser à 1, afin de l'initialiser à la valeur 0xA après l'avoir activée, et les sorties qui commandent le InvAESRound doivent être utilisées par conséquent.

[65] Il n'est pas du tout nécessaire de tester le circuit sur une autre donnée.

[66] La donnée a été repassée en entrée à la fin de son premier déchiffrement.

[67] Dans le chapitre I. Éléments de contexte, la partie 4. Fonctionnement du déchiffrement de l'AES.

[68] Dans le 1er paragraphe de la même partie et du même chapitre.

[69] A partir de laquelle les clés des rondes suivantes sont dérivées.

[70] Naturellement, le chiffrement aussi.

[71] Et aussi un chiffrement plus sûr.

IV. Conclusion

Au tout commencement du projet, les difficultés apparentes face au projet étaient assez nombreuses. En particulier, l'usage logiciel Modelsim de Mentor Graphics et la maîtrise du langage VHDL n'ont été pas aisés après les premières séances de cours. Toutefois la segmentation des fonctionnalités du projet a contribué en l'allègement du travail.

La conduite du projet était très intéressante vu que celui-ci représente un système concret et qui a une réelle utilité. Le projet reprenait en détail tous les éléments abordés lors du cours de Projet Conception de Système Numérique, et permettait de l'appliquer pour résoudre des problématiques logiques en modélisant des circuits numériques. Le projet est en adéquation avec le contenu des cours Architecture des Processeurs et Introduction à la Cryptographie. Dans le premier, nous avons simulé le fonctionnement de quelques éléments clés définis dans le cours électronique numérique dispensé en première année, comme registre, multiplexeur, encodeur, bascule asynchrone ainsi que les fonctions logiques principales. Dans le deuxième cours, nous avons appréhendé théoriquement la méthode de chiffrement AES et de façon plus pratique son déchiffrement.

V. Annexe

Texte clair : (Hex) d4 f1 25 f0 97 f7 ce e7 47 66 9b 78 30 56 ca a7

Texte chiffré : 8c 11 35 44 06 ad 44 88 de ca ec 83 aa 03 43 06

KeyExpansion (round 0) : 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c

KeyExpansion (round 9) : 0b b8 15 4b 69 85 52 75 f6 cd d9 70 09 57 7a 6b

KeyExpansion (round 10) : e7 05 10 0b 8e 80 42 7e 78 4d 9b 0e 71 1a e1 65

Round 10

InvAddRoundKey : 6b 14 25 4f 88 2d 06 f6 a6 87 77 8d db 19 a2 63

Round 9

InvShiftRows : 6b 19 77 f6 88 14 a2 8d a6 2d 25 63 db 87 06 4f

InvSubBytes : 7f d4 f5 42 c4 fa 3a 5d 24 d8 3f fb b9 17 6f 84

InvAddRoundKey : 74 6c e0 09 ad 7f 68 28 d2 15 e6 8b b0 40 15 ef

InvMixColumns : 1a d8 77 44 45 78 bf 10 1a 2b 61 fa b3 3a 02 81

Round 8

InvShiftRows : 1a 3a 61 10 45 d8 02 fa 1a 78 77 81 b3 2b bf 44

InvSubBytes : a2 80 ef ca 6e 61 77 2d a2 bc f5 0c 6d f1 08 1b