

Marc NGUYEN – 12 Janvier 2020

# **Rapport du Projet Conception d'un Système Numérique AES VHDL**

*Modélisation VHDL de l'algorithme de chiffrement AES*

## **Table des Matières**

### **Rapport du Projet Conception d'un Système Numérique AES VHDL**

#### SubBytes

- SubBytes Entity

- SubBytes Architecture

- Component SBox

  - Entity

  - Architecture

  - Testbench

- SubBytes TestBench

#### ShiftRows

- ShiftRows Entity

- ShiftRows Architecture

- ShiftRows TestBench

#### MixColumns

- MixColumns Entity

- MixColumns Architecture

- Component MixColumn

  - Entity

  - Architecture

  - Testbench

- MixColumns TestBench

#### AddRoundKey

- AddRoundKey Entity

- AddRoundKey Architecture

- AddRoundKey TestBench

#### Round

- Round Entity

- Round Architecture

- Component Registre D

  - Entity

  - Architecture

  - TestBench

- Component state\_\_to\_\_bit128 et bit128\_\_to\_\_state

- Round TestBench

#### AES

- AES Entity

- AES Architecture

- Component Machine d'Etat

  - Entity

  - Architecture

  - TestBench

- Component Compteur de Round

  - Entity

  - Architecture

  - TestBench

- AES TestBench

#### Conclusion

# SubBytes

SubBytes effectue une transformation non linéaire appliqué à tous les octets de l'état en utilisant une SBox.

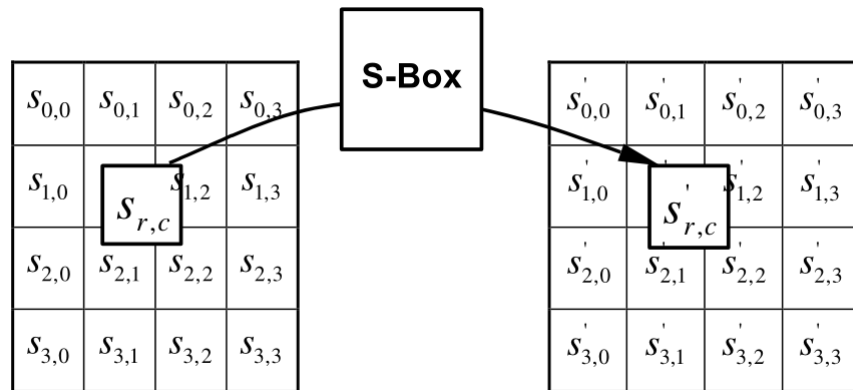


Figure 1 : Principe du SubBytes

## SubBytes Entity

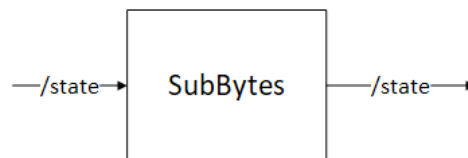


Figure 2 : SubBytes Entity

```

1 entity subbytes is
2
3   port (
4     data_i: in type_state;
5     data_o: out type_state
6   );
7
8 end entity subbytes;

```

Note : Le type type\_state est un array(0 to 3) de row\_state, qui lui-même est un array(0 to 3) de bit8(std\_logic\_vector(7 downto 0)). Il s'agit donc d'un tableau 4 x 4 avec 1 octet par case.

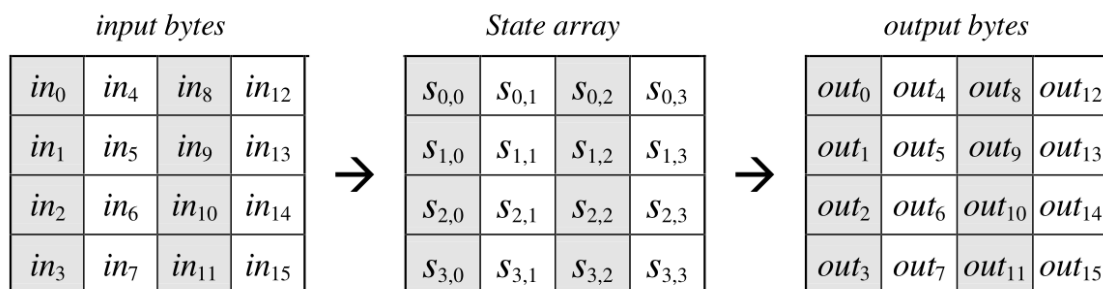


Figure 3 : Représentation d'un State array

## SubBytes Architecture

Ici, on va chercher à appliquer la SBox sur chaque octet de l'état (16 octets) **de manière concurrente**. Par conséquent, on utilise 1 SBox **pour chaque** octet.

**Dans la partie déclarative de l'architecture de SubBytes**, on déclare un component `sbox`, que l'on implémentera plus tard.

```
1 architecture subbytes_arch of subbytes is
2
3     component sbox
4         port (
5             data_i: in bit8;
6             data_o: out bit8
7         );
8     end component;
9
10 begin
```

**Dans la partie descriptive de l'architecture de SubBytes**, on génère 1 `sbox` par case, et on fait entrer la `data_i` et sortir la `data_o` correspondant.

```
1 begin
2
3     S_row: for i in 0 to 3 generate
4         S_case: for j in 0 to 3 generate
5             sbox: sbox port map(
6                 data_i => data_i(i)(j),
7                 data_o => data_o(i)(j)
8             );
9         end generate S_case;
10    end generate S_row;
11
12 end architecture subbytes_arch;
```

Il nous reste plus qu'à implémenter la SBox et **tester**.

## Component SBox

### Entity

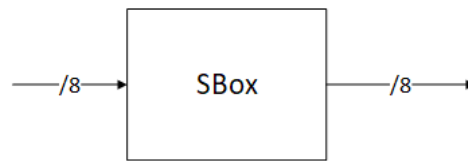


Figure 4 : SBox Entity.

```
1 entity sbox is
2
3   port (
4     data_i: in bit8;
5     data_o: out bit8
6   );
7
8 end entity adder;
```

### Architecture

Dans la partie déclarative de l'architecture **SBox**, on déclare un array de taille 256, **constante**, qui doit représenter la SBox suivante :

		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 5 : SBox fournie à implémenter

```
1 architecture sbox_arch of sbox is
2   -- Déclaration d'un type "sbox"
3   type sbox_t is array (0 to 255) of bit8;
4   -- Déclaration des données de la sbox
5   constant sbox_c: sbox_t := (X"52", X"09", X"6a", [...], X"0c",
6   X"7d");
7 begin
```

Dans la partie descriptive de l'architecture de **SBox**, on envoie l'image de `sbox_c` à `data_o`.

Cependant, il faut noter que `data_i` est en `bit128` (`std_logic_vector(127 downto 0)`). Comme l'opérateur `array()` n'accepte que des `integer` en paramètre, on utilise la librairie `ieee.numeric_std.all` afin de convertir des `std_logic_vector` en `integer`.

```
1  -- Pour utiliser le type bit8
2  library lib_aes;
3  use lib_aes.crypt_pack.bit8;
4
5  -- Pour utiliser les types de std_logic_1164
6  library ieee;
7  use ieee.std_logic_1164.std_logic;
8
9  -- Pour convertir des std_logic_vector en integer
10 use ieee.numeric_std.unsigned;
11 use ieee.numeric_std.to_integer;
12
13 -- [...]
14
15 architecture sbbox_arch of sbbox is
16
17     -- [...]
18
19 begin
20
21     data_o <= sbbox_c(to_integer(unsigned(data_i)));
22
23 end architecture sbbox_arch;
```

## Testbench

Test : "Tout l'ensemble de 0 à 255 doit correspondre à la SBox"

- En entrée : Un variable allant de 0 à 255.
- On s'attend à obtenir la transformation de la variable à partir de la SBox.

Résultat :

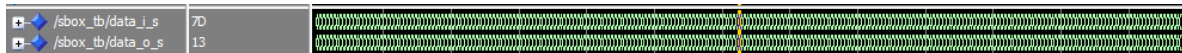


Figure 6 : Résultat obtenu pour le test SBox

En passant par les test par assertions VHDL, on obtient dans la console :

```
1 # ** Warning: NUMERIC_STD.TO_INTEGER: metavalue detected, returning
  0
2 #   Time: 0 ns   Iteration: 0   Instance: /sbox_tb/DUT
3 # ** Failure: Simulation Finished
4 #   Time: 2560 ns   Iteration: 0   Process: /sbox_tb/test File:
  SRC/BENCH/sbox_tb.vhd
5 # Break in Process test at SRC/BENCH/sbox_tb.vhd line 70
```

Donc, toutes les assertions sont passées, donc **sbox est validé**.

Manuellement :  $\text{sbox}(0x7D) = 0x13$ .

## SubBytes TestBench

En entrée :

```
1 ( (x"79", x"47", x"8b", x"65"),
2   (x"1b", x"8e", x"81", x"aa"),
3   (x"66", x"b7", x"7c", x"6f"),
4   (x"62", x"c8", x"e4", x"03"))
```

Ce que l'on attend :

```
1 ( (x"af", x"16", x"ce", x"bc"),
2   (x"44", x"e6", x"91", x"62"),
3   (x"d3", x"20", x"01", x"06"),
4   (x"ab", x"b1", x"ab", x"d5"))
```

Résultat :

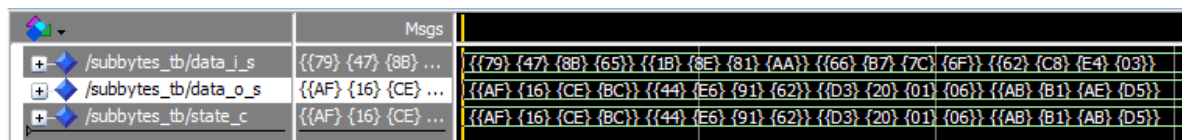


Figure 7 : Résultat obtenu pour le test SubBytes

Toutes les assertions sont passés, donc **SubBytes est validé**.

Manuellement : D'après la figure ci-dessus, nous avons exactement ce que l'on attend :

AddRoundKey : 79 1b 66 62 47 8e b7 c8 8b 81 7c e4 65 aa 6f 03  
Round 0  
SubBytes : af 44 d3 ab 16 e6 20 b1 ce 91 01 ae bc 62 06 d5

Figure 8 : Extrait de l'énoncé pour la validation SubBytes



# ShiftRows

ShiftRows doit permuter les octets de chaque ligne de l'état.

Le décalage dépend de l'indice (0...3) de la ligne.

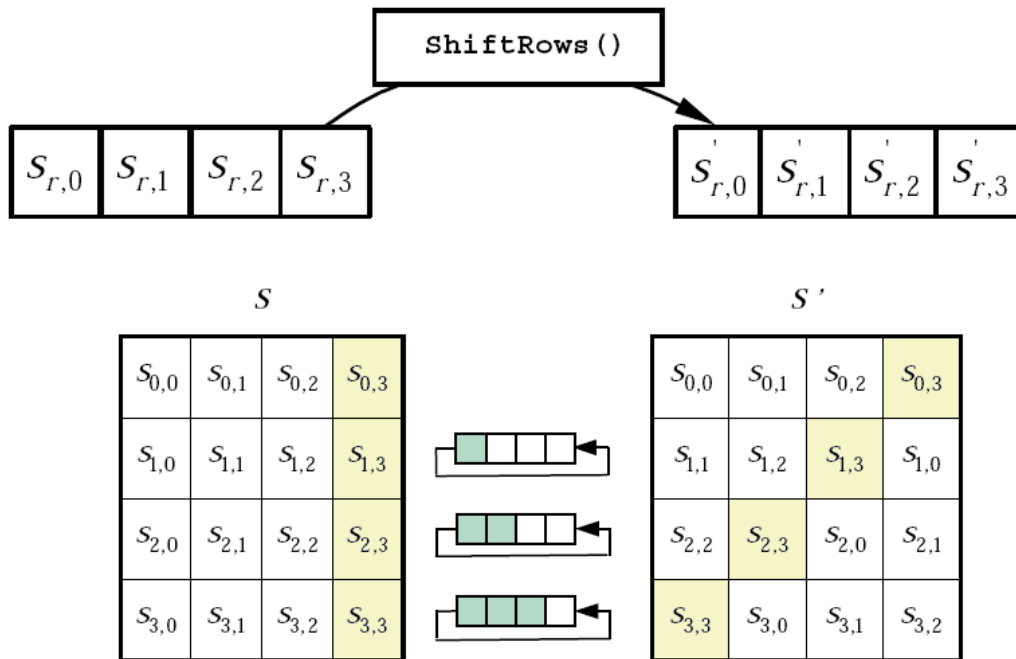


Figure 9 : Fonctionnement de ShiftRows

## ShiftRows Entity



Figure 10 : ShiftRows Entity

```

1 entity subbytes is
2
3   port (
4     data_i: in type_state;
5     data_o: out type_state
6   );
7
8 end entity subbytes;
```

## ShiftRows Architecture

Dans la partie descriptive de l'architecture de ShiftRows, on utilisera des boucles generate afin d'appliquer la permutation de manière concurrente.

```
1 architecture shiftrows_arch of shiftrows is
2   begin
3
4     rows: for i in 0 to 3 generate
5       cases: for j in 0 to 3 generate
6         data_o(i)(j) <= data_i(i)((i + j) mod 4);
7       end generate cases;
8     end generate rows;
9
10  end architecture shiftrows_arch;
```

## ShiftRows TestBench

En entrée :

```
1 ((x"a0", x"29", x"43", x"21"),
2  (x"ae", x"8e", x"d5", x"fa"),
3  (x"2f", x"6d", x"d9", x"21"),
4  (x"bc", x"e0", x"81", x"fc"))
```

Ce que l'on attend :

```
1 ((x"a0", x"29", x"43", x"21"),
2  (x"ae", x"8e", x"d5", x"fa"),
3  (x"2f", x"6d", x"d9", x"21"),
4  (x"bc", x"e0", x"81", x"fc"))
```

## Résultat :

/shiftrows_tb/data_i_s	-No Data-	{{AF} {16} {CE} {BC}} {{44} {E6} {91} {62}} {{D3} {20} {01} {06}} {{AB} {B1} {AE} {D5}}
+ (0)	-No Data-	{AF} {16} {CE} {BC}
+ (1)	-No Data-	{44} {E6} {91} {62}
+ (2)	-No Data-	{D3} {20} {01} {06}
+ (3)	-No Data-	{AB} {B1} {AE} {D5}
/shiftrows_tb/data_o_s	-No Data-	{{AF} {16} {CE} {BC}} {{E6} {91} {62} {44}} {{01} {06} {D3} {20}} {{D5} {AB} {B1} {AE}}
+ (0)	-No Data-	{AF} {16} {CE} {BC}
+ (1)	-No Data-	{E6} {91} {62} {44}
+ (2)	-No Data-	{01} {06} {D3} {20}
+ (3)	-No Data-	{D5} {AB} {B1} {AE}
/shiftrows_tb/state_c	-No Data-	{{AF} {16} {CE} {BC}} {{E6} {91} {62} {44}} {{01} {06} {D3} {20}} {{D5} {AB} {B1} {AE}}
+ (0)	-No Data-	{AF} {16} {CE} {BC}
+ (1)	-No Data-	{E6} {91} {62} {44}
+ (2)	-No Data-	{01} {06} {D3} {20}
+ (3)	-No Data-	{D5} {AB} {B1} {AE}

Figure 11 : Résultat obtenu pour le test ShiftRows

Toutes les assertions sont passés, donc **ShiftRows est validé**.

Manuellement : D'après la figure ci-dessus, nous avons exactement ce que l'on attend :

SubBytes : af 44 d3 ab 16 e6 20 b1 ce 91 01 ae bc 62 06 d5  
ShiftRows : af e6 01 d5 16 91 06 ab ce 62 d3 b1 bc 44 20 ae

Figure 12 : Extrait de l'énoncé pour la validation ShiftRows

# MixColumns

MixColumns applique une transformation linéaire sur chaque colonne de l'état.

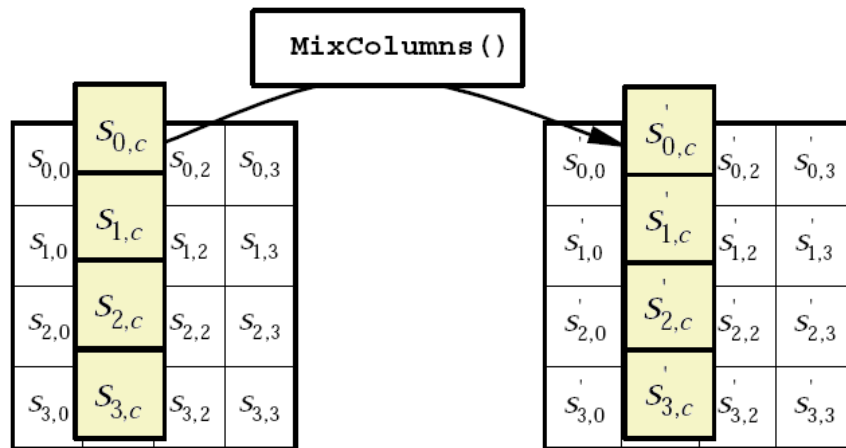


Figure 13 : Fonctionnement de MixColumns

## MixColumns Entity

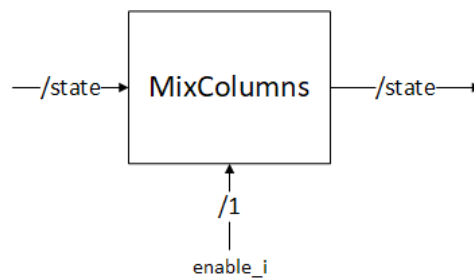


Figure 14 : MixColumns Entity

La MixColumns possède comme **entrée** :

- la matrice d'état en entrée que l'on nomme `data_i`, de type `type_state`
- `enable_i`, de type `std_logic`, qui permet :
  - Si `enable = 1`, `data_o <= MixColumns(data_i)`
  - Sinon, `data_o <= data_i`, ce qui permet le fonctionnement du round final de l'AES

La MixColumns possède comme **sortie** :

- la matrice d'état future que l'on nomme `data_o` de type `type_state`

```

1  entity mixcolumns is
2
3      port (
4          data_i: in type_state;
5          enable_i: in std_logic;
6          data_o: out type_state
7      );
8
9  end entity mixcolumns;
```

## MixColumns Architecture

Dans la partie déclarative de l'architecture de MixColumns, on déclare le composant `mixcolumn` qui servira à appliquer la fonction MixColumns. On déclare également des signaux qui permet la conversion entre colonne et état.

```
1 architecture mixcolumns_arch of mixcolumns is
2
3   component mixcolumn
4     port (
5       data_i: in column_state;
6       data_o: out column_state
7     );
8   end component;
9
10  type state_col_major is array(0 to 3) of column_state;
11  signal columns_i_s: state_col_major;
12  signal columns_o_s: state_col_major;
13
14  begin
```

Dans la partie descriptive de l'architecture de MixColumns, on convertit l'état en entrée en colonnes, puis on applique `mixcolumn` et on envoie le résultat.

```
1 begin
2
3   -- Slice Data_i in columns
4   rows_order_i: for i in 0 to 3 generate
5     columns_order_i: for j in 0 to 3 generate
6       columns_i_s(j)(i) <= data_i(i)(j);
7     end generate columns_order_i;
8   end generate rows_order_i;
9
10  -- Apply MixColumn for each column
11  columns_order: for j in 0 to 3 generate
12    MC: mixcolumn port map(
13      data_i => columns_i_s(j),
14      data_o => columns_o_s(j)
15    );
16  end generate columns_order;
17
18  -- Restore state from new columns (or old columns depending
19  enable_i)
20  rows_order_o: for i in 0 to 3 generate
21    columns_order_o: for j in 0 to 3 generate
22      data_o(i)(j) <= columns_o_s(j)(i) when enable_i = '1' else
23      data_i(i)(j);
24    end generate columns_order_o;
25  end generate rows_order_o;
26
27  end architecture mixcolumns_arch;
```

## Component MixColumn

Les colonnes doivent être traitées comme des polynômes dans  $GF(2^8)^2$  et multipliées modulo  $x^8 + x^4 + x^3 + x + 1$ .

La multiplication polynômiale par x peut être implémenté à l'aide d'un décalage à gauche suivi d'un ou-exclusif avec la valeur 0b1 0001 1011 conditionné par le bit de poids fort du polynôme.

Exemple avec un octet d'une colonne :

### Cas 1 : Bit de poids fort = 0

$$\begin{aligned}\{02\} \otimes S_{0,c} &= 0x02 \odot 0b0111\ 1111 \\ &= 0b1111\ 1110\end{aligned}$$

Donc, le modulo n'est pas nécessaire.

### Cas 2 : Bit de poids fort = 1

$$\begin{aligned}\{02\} \otimes S_{0,c} &= 0x02 \odot 0b1111\ 0000 \oplus 0b1\ 0001\ 1011 \\ &= 0b1\ 1110\ 0000 \oplus 0b1\ 0001\ 1011 \\ &= 0b1111\ 1011\end{aligned}$$

Ici, le modulo a été utilisé.

Donc, pour l'implémenter, on le conditionne avec le bit de poids fort du polynôme.

La fonction MixColumn doit être appliqué de cette manière :

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

$$\begin{aligned}s'_{0,c} &= (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\ s'_{1,c} &= s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c} \\ s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c}) \\ s'_{3,c} &= (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c})\end{aligned}$$

## Entity



Figure 15 : MixColumn Entity.

```

1 entity mixcolumn is
2
3   port (
4     data_i: in column_state;
5     data_o: out column_state
6   );
7
8 end entity mixcolumn;

```

Note : column\_state est un array(0 to 3) de bit8.

## Architecture

Nous partitionnons notre fonction :

- data2\_s est la colonne en entrée multiplié par 2 dans l'ensemble de Galois

```

1 architecture mixcolumn_arch of mixcolumn is
2   --[...]
3   signal data2_s: column_state;
4
5 begin
6
7   data2_s <= (
8     std_logic_vector((unsigned(data_i(0)(6 downto 0)) & "0") xor
9       ("000" & data_i(0)(7) & data_i(0)(7) & "0" & data_i(0)(7) &
10    data_i(0)(7))),
11     std_logic_vector((unsigned(data_i(1)(6 downto 0)) & "0") xor
12       ("000" & data_i(1)(7) & data_i(1)(7) & "0" & data_i(1)(7) &
13    data_i(1)(7))),
14     std_logic_vector((unsigned(data_i(2)(6 downto 0)) & "0") xor
15       ("000" & data_i(2)(7) & data_i(2)(7) & "0" & data_i(2)(7) &
16    data_i(2)(7))),
17     std_logic_vector((unsigned(data_i(3)(6 downto 0)) & "0") xor
18       ("000" & data_i(3)(7) & data_i(3)(7) & "0" & data_i(3)(7) &
19    data_i(3)(7)))
20   );

```

- data3\_s est la colonne en entrée multiplié par 3 dans l'ensemble de Galois

```

1 architecture mixcolumn_arch of mixcolumn is
2
3     signal data2_s: column_state;
4     signal data3_s: column_state;
5
6 begin
7     --[...]
8     data3_s <= (
9         data2_s(0) xor data_i(0),
10        data2_s(1) xor data_i(1),
11        data2_s(2) xor data_i(2),
12        data2_s(3) xor data_i(3)
13    );

```

- On applique data\_o les opérations décrites ci-dessus :

```

1 architecture mixcolumn_arch of mixcolumn is
2
3     signal data2_s: column_state;
4     signal data3_s: column_state;
5
6 begin
7
8     --[...]
9     data_o(0) <= data2_s(0) xor data3_s(1) xor data_i(2) xor
data_i(3);
10    data_o(1) <= data_i(0) xor data2_s(1) xor data3_s(2) xor
data_i(3);
11    data_o(2) <= data_i(0) xor data_i(1) xor data2_s(2) xor
data3_s(3);
12    data_o(3) <= data3_s(0) xor data_i(1) xor data_i(2) xor
data2_s(3);
13
14 end architecture mixcolumn_arch;

```

## Testbench

En entrée : une colonne

```
1 | ("af", "44", "d3", "ab")
```

Ce que l'on attend :

```
1 | ("a0", "ae", "2f", "bc")
```

Résultat :




 /mixcolumn_tb/data_i_s	{AF} {E6} {01} {D5}	{AF} {E6} {01} {D5}
 /mixcolumn_tb/data_o_s	{A0} {AE} {2F} {BC}	{A0} {AE} {2F} {BC}
 /mixcolumn_tb/column_c	{A0} {AE} {2F} {BC}	{A0} {AE} {2F} {BC}

Figure 16 : Résultat obtenu pour le test MixColumn

Toutes les assertions sont passés, donc **MixColumn** est validé.



Manuellement : D'après la figure ci-dessus, nous avons exactement ce que l'on attend :

ShiftRows : af e6 01 d5  
MixColumns : a0 ae 2f bc

Figure 17 : Extrait de l'énoncé pour la validation MixColumn

## MixColumns TestBench

En entrée : un état et 2 cas d'utilisation (enabled et disabled)

```
1 data_i_s <= ((x"af", x"16", x"ce", x"bc"),
2             (x"e6", x"91", x"62", x"44"),
3             (x"01", x"06", x"d3", x"20"),
4             (x"d5", x"ab", x"b1", x"ae"));
5
6 enable_i_s <= '0',
7             '1' after 50 ns;
```

Ce que l'on attend : Quand enable\_i = 1

```
1 ((x"a0", x"29", x"43", x"21"),
2  (x"ae", x"8e", x"d5", x"fa"),
3  (x"2f", x"6d", x"d9", x"51"),
4  (x"bc", x"e0", x"81", x"fc"));
```

Quand enable\_i = 0

```
1 ((x"af", x"16", x"ce", x"bc"),
2  (x"e6", x"91", x"62", x"44"),
3  (x"01", x"06", x"d3", x"20"),
4  (x"d5", x"ab", x"b1", x"ae"));
```

## Résultat :

	/mixcolumns_tb/data_i_s	/mixcolumns_tb/data_o_s	/mixcolumns_tb/enable_i_s	/mixcolumns_tb/state_when_enabled_c	/mixcolumns_tb/state_when_disabled_c
(0)	{(AF) {16} {CE} ...}	{(AF) {16} {CE} {BC} ...}	1	{(A0) {29} {43} {21} ...}	{(AF) {16} {CE} ...}
(1)	{(E6) {91} {62} {44}}	{(E6) {91} {62} {44}}		{(AE) {8E} {D5} {FA}}	{(AF) {16} {CE} {BC} ...}
(2)	{(01) {06} {D3} {20}}	{(01) {06} {D3} {20}}		{(2F) {6D} {D9} {51}}	{(E6) {91} {62} {44}}
(3)	{(D5) {AB} {B1} {AE}}	{(D5) {AB} {B1} {AE}}		{(BC) {E0} {81} {FC}}	{(01) {06} {D3} {20}}
(0)	{(AF) {16} {CE} {BC} ...}	{(AF) {16} {CE} {BC} ...}		{(A0) {29} {43} {21} ...}	{(D5) {AB} {B1} {AE}}
(1)	{(E6) {91} {62} {44}}	{(E6) {91} {62} {44}}		{(AE) {8E} {D5} {FA}}	
(2)	{(01) {06} {D3} {20}}	{(01) {06} {D3} {20}}		{(2F) {6D} {D9} {51}}	
(3)	{(D5) {AB} {B1} {AE}}	{(D5) {AB} {B1} {AE}}		{(BC) {E0} {81} {FC}}	
(0)	{(AF) {16} {CE} {BC} ...}	{(AF) {16} {CE} {BC} ...}		{(A0) {29} {43} {21} ...}	
(1)	{(E6) {91} {62} {44}}	{(E6) {91} {62} {44}}		{(AE) {8E} {D5} {FA}}	
(2)	{(01) {06} {D3} {20}}	{(01) {06} {D3} {20}}		{(2F) {6D} {D9} {51}}	
(3)	{(D5) {AB} {B1} {AE}}	{(D5) {AB} {B1} {AE}}		{(BC) {E0} {81} {FC}}	

Figure 18 : Résultat obtenu pour le test MixColumns

Toutes les assertions sont passés, donc **MixColumns** est validé.

Manuellement : D'après la figure ci-dessus, nous avons exactement ce que l'on attend :

ShiftRows : af e6 01 d5 16 91 06 ab ce 62 d3 b1 bc 44 20 ae  
MixColumns : a0 ae 2f bc 29 8e 6d e0 43 d5 d9 81 21 fa 51 fc

Figure 19 : Extrait de l'énoncé pour la validation MixColumns

# AddRoundKey

AddRoundKey fait simplement un XOR entre l'état et une sous clé (round key).

## AddRoundKey Entity

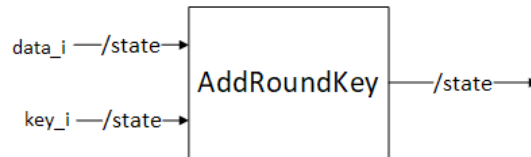


Figure 20 : AddRoundKey Entity.

VHDL :

```
1 entity addroundkey is
2
3   port (
4     data_i: in type_state;
5     key_i: in type_state;
6     data_o: out type_state
7   );
8
9 end entity addroundkey;
```

## AddRoundKey Architecture

Le résultat est immédiat :

```
1 architecture addroundkey_arch of addroundkey is
2   begin
3
4     rows: for i in 0 to 3 generate
5       cases: for j in 0 to 3 generate
6         data_o(i)(j) <= data_i(i)(j) xor key_i(i)(j);
7       end generate rows;
8     end generate cases;
9
10  end architecture;
```

## AddRoundKey TestBench

En entrée : un state et une sous-clé

State :

```
1 | ((x"52", x"6f", x"20", x"6c"),
2 | (x"65", x"20", x"76", x"65"),
3 | (x"73", x"65", x"69", x"20"),
4 | (x"74", x"6e", x"6c", x"3f"))
```

Sous-clé :

```
1 | ((x"2b", x"28", x"ab", x"09"),
2 | (x"7e", x"ae", x"f7", x"cf"),
3 | (x"15", x"d2", x"15", x"4f"),
4 | (x"16", x"a6", x"88", x"3c"))
```

Ce que l'on attend :

```
1 | ((x"79", x"47", x"8b", x"65"),
2 | (x"1b", x"8e", x"81", x"aa"),
3 | (x"66", x"b7", x"7c", x"6f"),
4 | (x"62", x"c8", x"e4", x"03"))
```

Résultat :

/addroundkey_tb/data_i_s		{{{52} {6F} {20} {6C}}} {...	{{{52} {6F} {20} {6C}}}
+ (0)		{52} {6F} {20} {6C}	{52} {6F} {20} {6C}
+ (1)		{65} {20} {76} {65}	{65} {20} {76} {65}
+ (2)		{73} {65} {69} {20}	{73} {65} {69} {20}
+ (3)		{74} {6E} {6C} {3F}	{74} {6E} {6C} {3F}
/addroundkey_tb/key_i_s		{{{2B} {28} {AB} {09}}} {...	{{{2B} {28} {AB} {09}}}
+ (0)		{2B} {28} {AB} {09}	{2B} {28} {AB} {09}
+ (1)		{7E} {AE} {F7} {CF}	{7E} {AE} {F7} {CF}
+ (2)		{15} {D2} {15} {4F}	{15} {D2} {15} {4F}
+ (3)		{16} {A6} {88} {3C}	{16} {A6} {88} {3C}
/addroundkey_tb/data_o_s		{{{79} {47} {8B} {65}}} {...	{{{79} {47} {8B} {65}}}
+ (0)		{79} {47} {8B} {65}	{79} {47} {8B} {65}
+ (1)		{1B} {8E} {81} {AA}	{1B} {8E} {81} {AA}
+ (2)		{66} {B7} {7C} {6F}	{66} {B7} {7C} {6F}
+ (3)		{62} {C8} {E4} {03}	{62} {C8} {E4} {03}
/addroundkey_tb/state_c		{{{79} {47} {8B} {65}}} {...	{{{79} {47} {8B} {65}}}
+ (0)		{79} {47} {8B} {65}	{79} {47} {8B} {65}
+ (1)		{1B} {8E} {81} {AA}	{1B} {8E} {81} {AA}
+ (2)		{66} {B7} {7C} {6F}	{66} {B7} {7C} {6F}
+ (3)		{62} {C8} {E4} {03}	{62} {C8} {E4} {03}

Figure 21 : Résultat obtenu pour le test AddRoundKey

Toutes les assertions sont passés, donc **AddRoundKey** est validé.

Manuellement : D'après la figure ci-dessus, nous avons exactement ce que l'on attend :

```
InitKey : 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c  
SetPlaintext : 52 65 73 74 6f 20 65 6e 20 76 69 6c 6c 65 20 3f  
AddRoundKey : 79 1b 66 62 47 8e b7 c8 8b 81 7c e4 65 aa 6f 03
```

Figure 22 : Extrait de l'énoncé pour la validation AddRoundKey

# Round

Un round doit appliquer toutes les fonctions que nous avons développées jusqu'à là. En fonction du nombre de round, nous devons sélectionner quelle fonction appliquer :

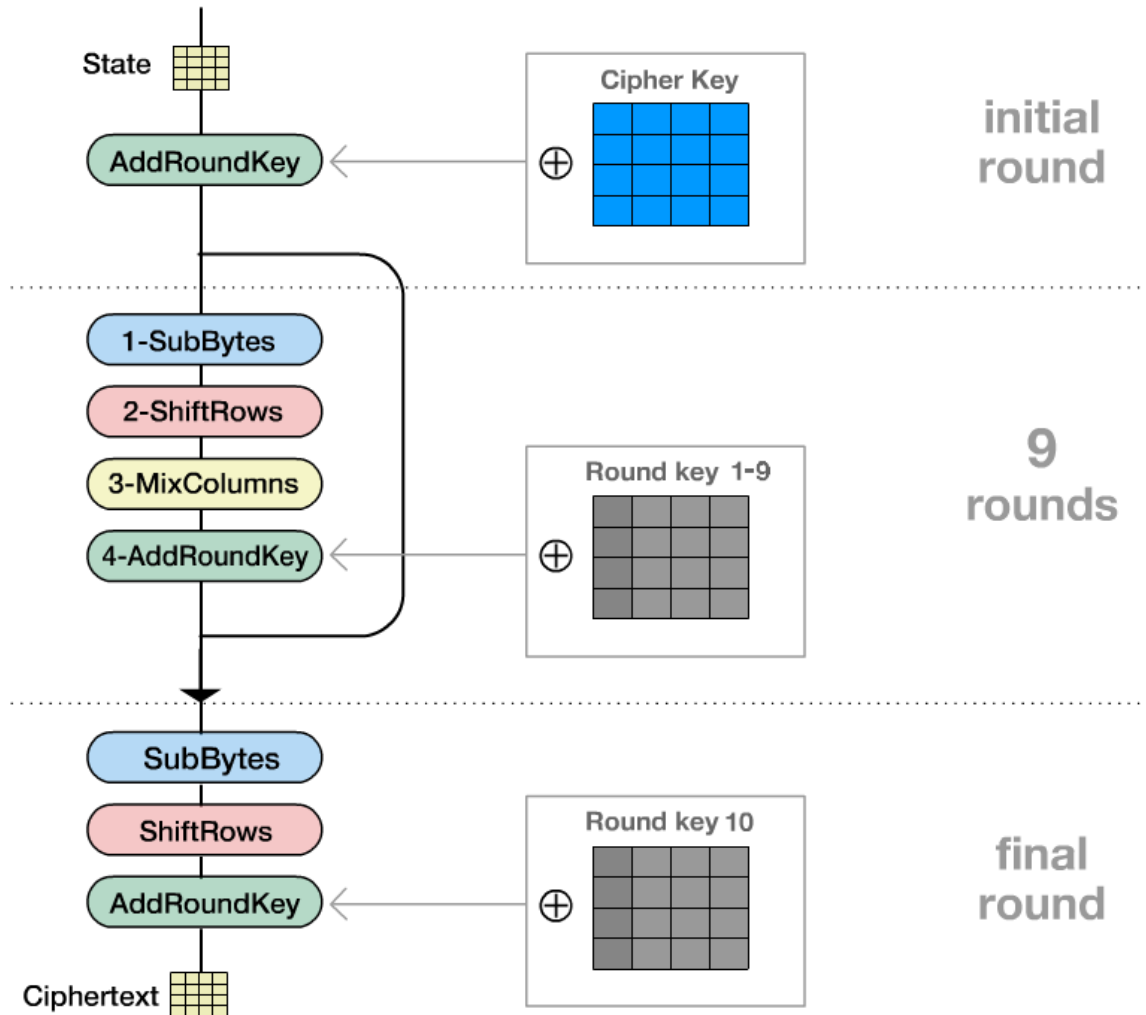


Figure 23 : Composition des rounds

Il nous faudra donc cadencer notre architecture avec un registre D.

## Round Entity

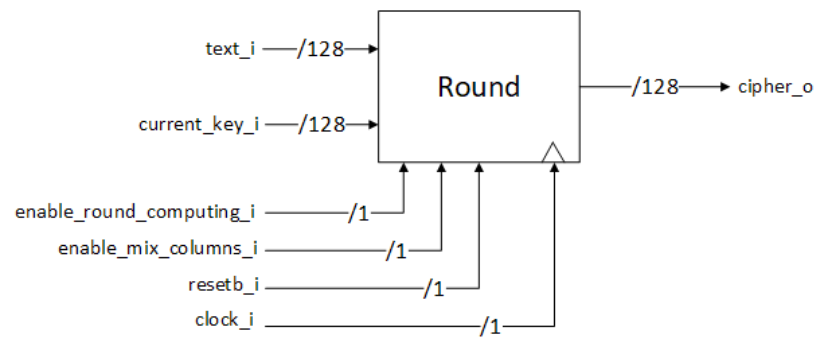


Figure 24 : Round Entity.

Le Round possède comme **entrée** :

- Le texte clair que l'on nomme `text_i`, de type `bit128`
- La sous-clé en entrée que l'on nomme `current_key_i`, de type `bit128`
- L'horloge `clock_i` en `std_logic` et le reset `resetb_i` (reset si le niveau est bas)
- `enable_round_computing_i` en `std_logic` qui servira de choisir l'entrée entre le texte clair pour le round 0, ou les résultats des précédant round.
- `enable_mix_columns_i` qui servira de d'activer/désactiver MixColumns pour le Round 0 et Round 10

Le round possède comme **sortie** :

- Le texte chiffré du round que l'on nomme `cipher_o` de type `bit128`

```
1  entity round is
2
3      port (
4          text_i: in bit128;
5          current_key_i: in bit128;
6          clock_i: in std_logic;
7          resetb_i: in std_logic;
8          enable_round_computing_i: in std_logic;
9          enable_mix_columns_i: in std_logic;
10         cipher_o: out bit128
11     );
12
13 end entity round;
```

## Round Architecture

Comme nos entrées sont des bit128 et que notre architecture se base sur des type\_state, on convertira les bits128 en entrée en state, et les state en sortie en bit128

On prévoit donc notre architecture VHDL :

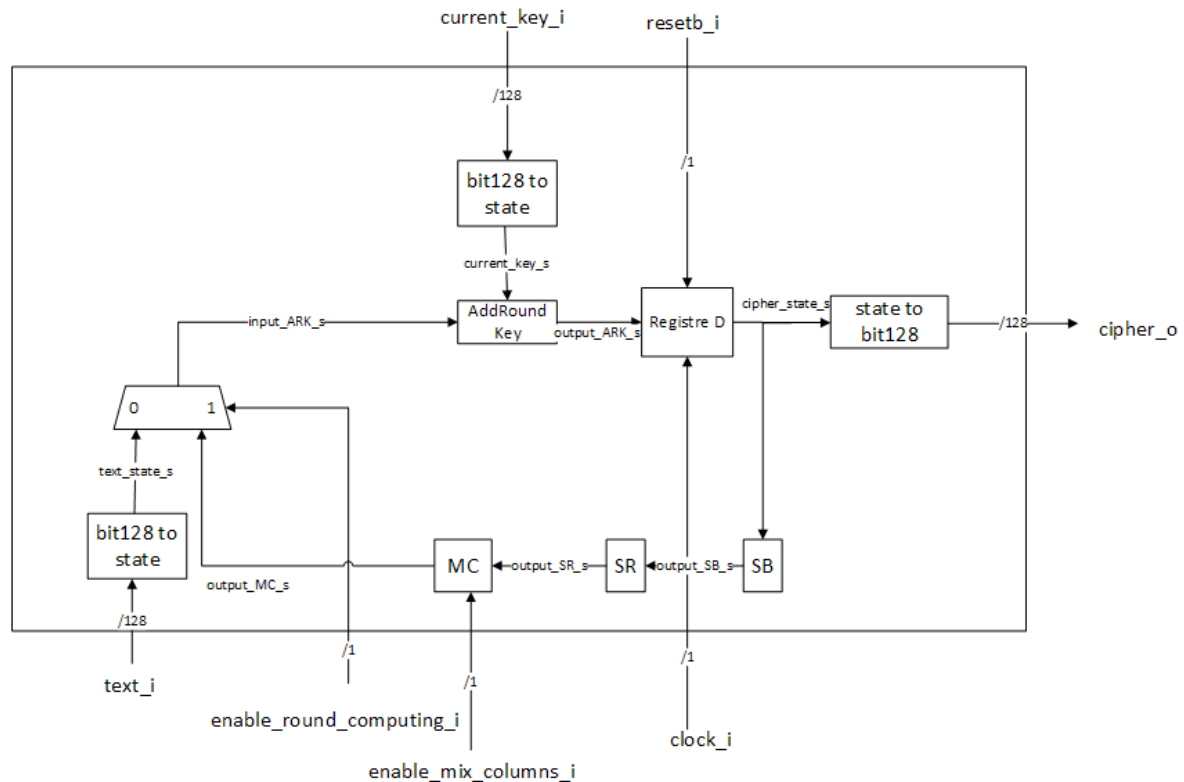


Figure 25 : Round Architecture

On connecte donc nos différents composants en VHDL :

```

1  architecture round_arch of round is
2
3      ... -- Déclaration des composants et signaux affichés dans le
         schéma
4
5  begin
6
7      text_bit128_to_state: bit128_to_state
8          port map(
9              data_i => text_i,
10             data_o => text_state_s
11         );
12
13
14     -- demux
15     input_ARC_s <= output_MC_s when enable_round_computing_i = '1'
16     else text_state_s;
17
18     current_key_bit128_to_state: bit128_to_state
19         port map(

```



```

19     data_i => current_key_i,
20     data_o => current_key_s
21 );
22
23 addroundkey_instance: addroundkey
24     port map(
25         data_i => input_ARK_s,
26         key_i => current_key_s,
27         data_o => output_ARK_s
28     );
29
30 register_d_instance: register_d
31     port map(
32         resetb_i => resetb_i,
33         clock_i => clock_i,
34         state_i => output_ARK_s,
35         state_o => cipher_state_s
36     );
37
38 cipher_to_bit128: state_to_bit128
39     port map(
40         data_i => cipher_state_s,
41         data_o => cipher_o
42     );
43
44 subbytes_instance: subbytes
45     port map(
46         data_i => cipher_state_s,
47         data_o => output_SB_s
48     );
49
50 shiftrows_instance: shiftrows
51     port map(
52         data_i => output_SB_s,
53         data_o => output_SR_s
54     );
55
56 mixcolumns_instance: mixcolumns
57     port map(
58         data_i => output_SR_s,
59         data_o => output_MC_s,
60         enable_i => enable_mix_columns_i
61     );
62
63
64 end architecture round_arch;

```

## Component Registre D

Le registre D permettra de cadencer notre round et de synchroniser avec un compteur de round et une machine d'état.

### Entity

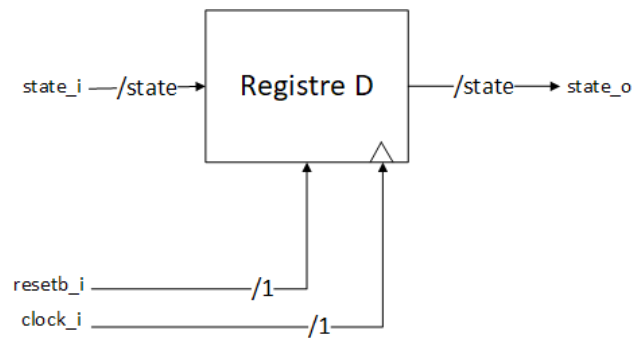


Figure 26 : Registre D Entity.

```
1 entity register_d is
2
3   port (
4       resetb_i : in std_logic;
5       clock_i : in std_logic;
6       state_i : in type_state;
7       state_o : out type_state
8   );
9
10 end entity register_d;
```

### Architecture

On adapte notre registre D au type state.

```
1 architecture register_d_arch of register_d is
2
3   signal state_s : type_state;
4
5   begin
6
7       seq_0 : process (clock_i, resetb_i) is
8
9       begin
10
11         -- Reset clears state
12         if resetb_i = '0' then
13             for i in 0 to 3 loop
14                 for j in 0 to 3 loop
15                     state_s(i)(j) <= (others => '0');
16                 end loop;
17             end loop;
18
19         -- New data at RISING
20         elsif clock_i'event and clock_i='1' then
```



## Component state\_to\_bit128 et bit128\_to\_state

La relation entrée/sortie avec le composant est assez explicite.

Input bit sequence	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	...
Byte number	0								1								2								...
Bit numbers in byte	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	...

$$\begin{aligned}
 a0 &= \{input_0, input_1, \dots, input_7\}; \\
 a1 &= \{input_8, input_9, \dots, input_{15}\}; \\
 &\vdots \\
 a15 &= \{input_{120}, input_{121}, \dots, input_{127}\}.
 \end{aligned}$$

Figure 28 : Concordance des bits avec les octets d'un état

Source : NIST, [“Fips-197, announcing the ADVANCED ENCRYPTION STANDARD \(AES\)”](#)

Avec bit128 to state :

```

1  entity bit128_to_state is
2
3      port (
4          data_i: in bit128;
5          data_o: out type_state
6      );
7
8  end entity bit128_to_state;
9
10 architecture bit128_to_state_arch of bit128_to_state is
11 begin
12
13     rows: for i in 0 to 3 generate
14         cases: for j in 0 to 3 generate
15             data_o(3 - i)(3 - j) <= data_i((8 * (i+1) - 1) + (32 * j)
16             downto (i * 8 + j * 32));
17         end generate cases;
18     end generate rows;
19 end architecture bit128_to_state_arch;

```

## Round TestBench

On teste le Round 0 et le Round 1 inscrit dans l'énoncé.

Test 1 : "Le round n'est pas initialisé avant le coup d'horloge"

Test 2 : "Le resultat du round 1 (791b6662478eb7c88b817ce465aa6f03) est obtenu au premier coup d'horloge."

- En entrée :
  - enable\_round\_computing\_s=0
  - enable\_mix\_columns\_s=0
  - text\_i\_s="526573746f20656e2076696c6c65203f"
  - current\_key\_i\_s="2b7e151628aed2a6abf7158809cf4f3c"

Test 3 : "Le resultat du round 2 (d54257ea74ccc710b56066f9de80a1b8) est obtenu au 2e coup d'horloge."

- En entrée :
  - La suite du test 2 (791b6662478eb7c88b817ce465aa6f03)
  - enable\_round\_computing\_s=1
  - enable\_mix\_columns\_s=1
  - text\_i\_s="526573746f20656e2076696c6c65203f"
  - current\_key\_i\_s="75ec78565d42aaf0f6b5bf78ff7af044"

### Résultat :

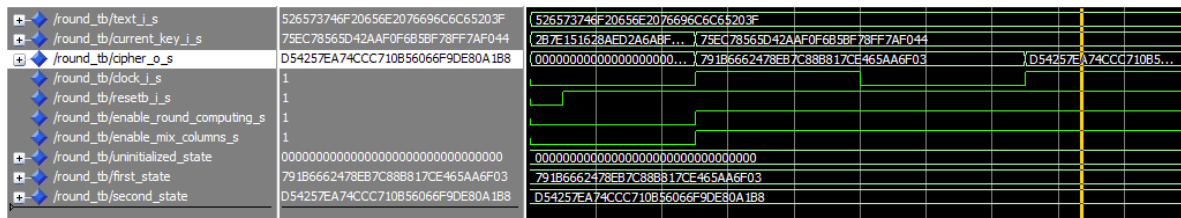


Figure 29 : Résultat obtenu pour le test Round

Toutes les assertions sont passés, donc **Round est validé**.

Manuellement : D'après la figure ci-dessus, nous avons exactement ce que l'on attend :

```
InitKey : 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c
SetPlaintext : 52 65 73 74 6f 20 65 6e 20 76 69 6c 6c 65 20 3f
AddRoundKey : 79 1b 66 62 47 8e b7 c8 8b 81 7c e4 65 aa 6f 03
Round 0
SubBytes : af 44 d3 ab 16 e6 20 b1 ce 91 01 ae bc 62 06 d5
ShiftRows : af e6 01 d5 16 91 06 ab ce 62 d3 b1 bc 44 20 ae
MixColumns : a0 ae 2f bc 29 8e 6d e0 43 d5 d9 81 21 fa 51 fc
ComputeKey : 75 ec 78 56 5d 42 aa f0 f6 b5 bf 78 ff 7a f0 44
AddRoundKey : d5 42 57 ea 74 cc c7 10 b5 60 66 f9 de 80 a1 b8
```

Figure 30 : Extrait de l'énoncé pour la validation Round

# AES

Pour gérer les états de l'AES, nous utiliserons une machine d'état couplé avec un compteur de round. En fonction de l'état, nous fournissons la sous-clé correspondante au round et exécutons le round.

## AES Entity

On utilisera un signal `start_i` pour démarrer l'AES et un signal `aes_on_o` affichant si l'AES est en cours d'exécution.

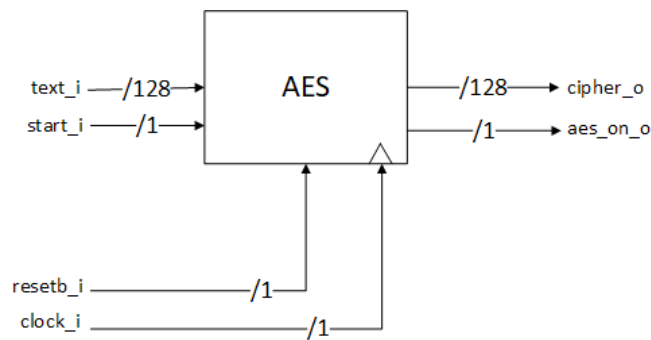


Figure 31 : AES Entity.

```
1  entity aes is
2
3  port (
4      clock_i: in std_logic;
5      resetb_i: in std_logic;
6      start_i: in std_logic;
7      text_i: in bit128;
8      aes_on_o: out std_logic;
9      cipher_o: out bit128
10 );
11
12 end entity aes;
```

## AES Architecture

On prévoit cette architecture :

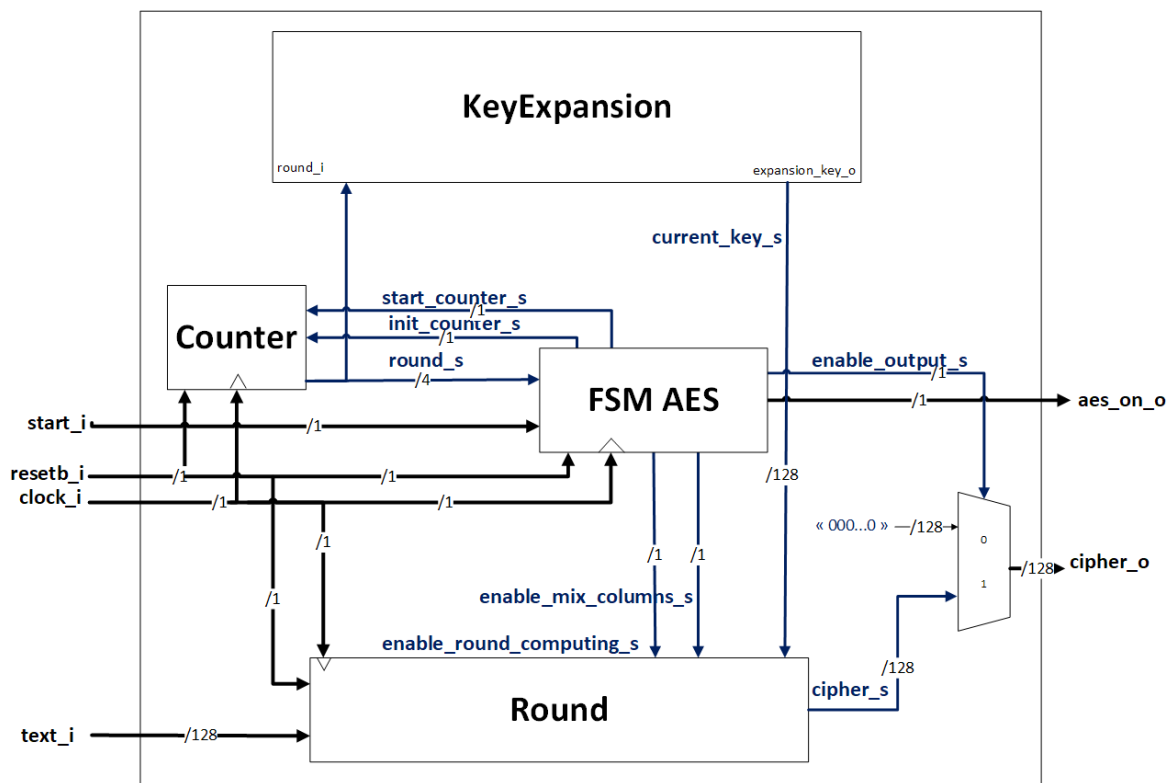


Figure 32 : AES Architecture

Cela se fait sans problème en VHDL :

```

1  architecture aes_arch of aes is
2
3      component KeyExpansion_I_0_table is
4          -- [...] I/O décrit sur le schéma
5      end component KeyExpansion_I_0_table;
6
7      component counter is
8          -- [...] I/O décrit sur le schéma
9      end component counter;
10
11     component fsm_aes is
12         -- [...] I/O décrit sur le schéma
13     end component fsm_aes;
14
15     component round is
16         -- [...] I/O décrit sur le schéma
17     end component round;
18
19     -- [...] Signaux décrit sur le schéma
20
21 begin
22
23     KeyExpansion_I_0_instance: KeyExpansion_I_0_table
24     port map(
25         round_i => round_s,
```

```

26     expansion_key_o => current_key_s
27 );
28
29 counter_instance: counter
30     port map(
31         clock_i => clock_i,
32         resetb_i => resetb_i,
33         init_counter_i => init_counter_s,
34         start_counter_i => start_counter_s,
35         round_o => round_s
36     );
37
38 fsm_aes_instance: fsm_aes
39     port map(
40         round_i => round_s,
41         clock_i => clock_i,
42         resetb_i => resetb_i,
43         start_i => start_i,
44         init_counter_o => init_counter_s,
45         start_counter_o => start_counter_s,
46         enable_output_o => enable_output_s,
47         aes_on_o => aes_on_o,
48         enable_round_computing_o => enable_round_computing_s,
49         enable_mix_columns_o => enable_mix_columns_s
50     );
51
52 round_instance: round
53     port map(
54         text_i => text_i,
55         current_key_i => current_key_s,
56         clock_i => clock_i,
57         resetb_i => resetb_i,
58         enable_round_computing_i => enable_round_computing_s,
59         enable_mix_columns_i => enable_mix_columns_s,
60         cipher_o => cipher_s
61     );
62
63 -- Mux
64 cipher_o <= cipher_s when enable_output_s='1' else
65     X"00000000000000000000000000000000";
66 end architecture aes_arch;

```



## Component Machine d'Etat

La machine d'état contrôle le comportement du round en fonction du compteur de round.

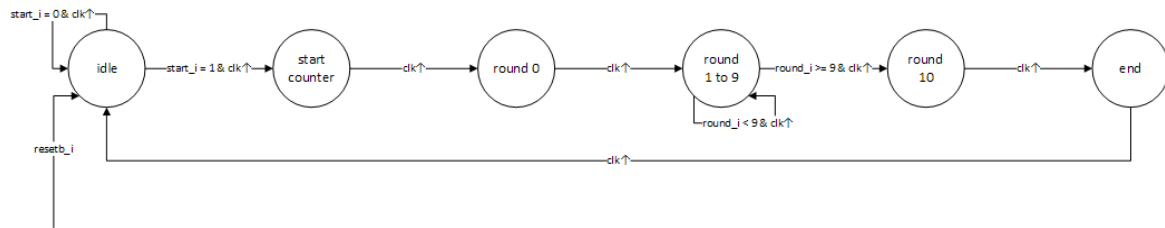


Figure 33 : Machine d'état

Voici donc la configuration des données en fonction des états :

	idle	start_counter	round_0	round_1to9	round10	end_fsm
init_counter_o	<b>1</b>	<b>1</b>	0	0	0	0
start_counter_o	0	<b>1</b>	<b>1</b>	<b>1</b>	0	0
enable_output_o	0	0	0	0	0	<b>1</b>
aes_on_o	0	0	<b>1</b>	<b>1</b>	<b>1</b>	0
enable_RC_o	0	0	0	<b>1</b>	<b>1</b>	0
enable_MC_o	0	0	0	<b>1</b>	0	0

## Entity

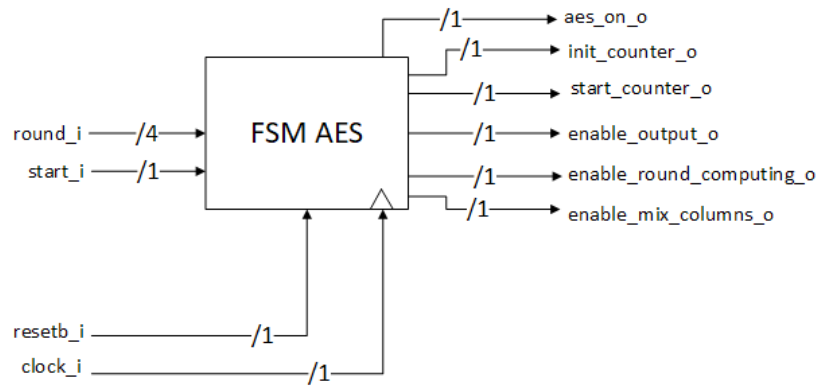


Figure 34.: Machine d'état Entity.

D'après le diagramme d'état, on définit rapidement les entrées et les sorties :

```
1  entity fsm_aes is
2
3  port (
4      round_i: in bit4;  -- Utilise: 10, Max: 16
5      clock_i: in std_logic;
6      resetb_i: in std_logic;
7      start_i: in std_logic;
8      init_counter_o: out std_logic;
9      start_counter_o: out std_logic;
10     enable_output_o: out std_logic;
11     aes_on_o: out std_logic;
12     enable_round_computing_o: out std_logic;
13     enable_mix_columns_o: out std_logic;
14 );
15
16 end entity fsm_aes;
```

## Architecture

Dans la partie déclarative, on définit nos états :

```
1 architecture fsm_aes_arch of fsm_aes is
2
3     type state_fsm is (idle, start_counter, round_0, round_1to9,
4       round10, end_fsm);
5     signal etat_present, etat_futur: state_fsm;
6 begin
```

Dans la partie descriptive, on définit 3 process.

Pour changer d'état en fonction de l'horloge et du reset, nous utiliserons un process dédié, sensible à l'horloge et au reset :

```
1 -- architecture fsm_aes_arch
2   event_dispatcher: process (clock_i, resetb_i)
3   begin
4     if resetb_i = '0' then
5       etat_present <= idle;
6     elsif clock_i'event and clock_i = '1' then
7       etat_present <= etat_futur;
8     end if;
9   end process event_dispatcher;
```

Pour changer d'état en fonction des entrées, nous utiliserons un autre process dédié, sensible à l'état présent, le start et le round :

```
1 -- architecture fsm_aes_arch
2   event_map_to_state: process (etat_present, start_i, round_i)
3   begin
4     case etat_present is
5     when idle =>
6       if start_i = '0' then
7         etat_futur <= idle; -- loop until start
8       else
9         etat_futur <= start_counter;
10      end if;
11    when start_counter =>
12      etat_futur <= round_0;
13    when round_0 =>
14      etat_futur <= round_1to9;
15    when round_1to9 =>
16      if to_integer(unsigned(round_i)) < 9 then
17        etat_futur <= round_1to9; -- loop while round_i < 9
18      else
19        etat_futur <= round10;
20      end if;
21    when round10 =>
22      etat_futur <= end_fsm;
23    when end_fsm =>
```

```

24     etat_futur <= idle;
25     end case;
26     end process event_map_to_state;

```

Pour changer les données en fonction de l'état, nous utiliserons également un process dédié, sensible à l'état présent :

```

1  -- architecture fsm_aes_arch
2  state_model: process (etat_present)
3  begin
4      case etat_present is
5          when idle =>
6              init_counter_o <= '1';
7              start_counter_o <= '0';
8              enable_output_o <= '0';
9              aes_on_o <= '0';
10             enable_round_computing_o <= '0';
11             enable_mix_columns_o <= '0';
12             ... -- Le comportement est défini dans le tableau ci-dessus.
13         when end_fsm =>
14             init_counter_o <= '0';
15             start_counter_o <= '0';
16             enable_output_o <= '1';
17             aes_on_o <= '0';
18             enable_round_computing_o <= '0';
19             enable_mix_columns_o <= '0';
20         end case;
21     end process state_model;

```

## TestBench

Les tests unitaires sont :

- Test 1 : "Tester que la FSM est en état initial (Idle)"
- Test 2 : "Pendant un signal start, la FSM reste en état initial mais son état futur est start\_counter"
- Test 3 : "Après un signal start, la FSM est en état start\_counter"
- Test 4 : "Après l'état start, la FSM est en état R0"
- Test 5 : "Après l'état R0, la FSM est en état R1toR9"
- Test 6 : "Après l'état R1toR9, la FSM est en état R10"
- Test 7 : "Après l'état R10, la FSM est en état End"
- Test 8 : "Après l'état End, la FSM est en état Idle"

Résultat :

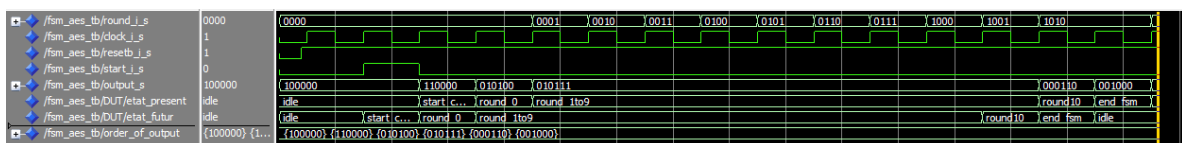


Figure 35 : Résultat obtenu pour le test Machine d'Etat

Toutes les assertions sont passées, donc **la machine d'état est validé**.

Manuellement : On peut voir que l'ordre des états décrit la même évolution prévu que sur la figure 42.

## Component Compteur de Round

Notre compteur doit aller de 0 à 10, on utilise donc un compteur 4 bits.

### Entity

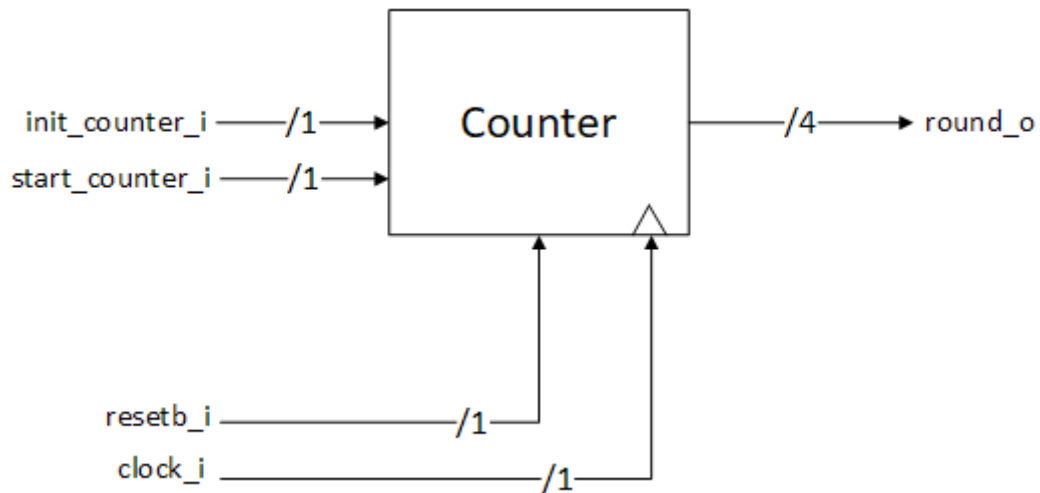


Figure 36 : Compteur Entity

Notre compteur devra être armé avec `init_counter_i`, et devra s'incrémenter dès que le `start_counter_i` passe à 1.

```
1  entity counter is
2
3      port (
4          clock_i: in std_logic;
5          resetb_i: in std_logic;
6          init_counter_i: in std_logic;
7          start_counter_i: in std_logic;
8          round_o: out bit4
9      );
10
11 end entity counter;
```

## Architecture

L'architecture est simplement basé sur un registre D.

```
1 architecture counter_arch of counter is
2
3     signal round_s : bit4;
4
5 begin
6     seq_0 : process (clock_i, resetb_i) is
7     begin
8         -- Reset clears state
9         if resetb_i = '0' then
10             round_s <= "0000";
11
12         -- New data at RISING
13         elsif clock_i'event and clock_i = '1' then
14             -- Arm
15             if init_counter_i = '1' then
16                 round_s <= "0000";
17
18             -- Start counting
19             elsif start_counter_i = '1' then
20                 round_s <= std_logic_vector(unsigned(round_s) + 1);
21                 if round_s = "1011" then -- if round > 10
22                     round_s <= "0000";
23                 end if;
24             end if;
25         end if;
26     end process seq_0;
27
28     round_o <= round_s;
29 end architecture counter_arch;
```

## TestBench

Les tests unitaires sont :

- Test 1 : "Le compteur s'incrémente et suit le timing prévu"
- Test 2 : "Le compteur s'arrête quand start = 0"
- Test 3 : "Le compteur se réinitialise quand init = 1"

Résultat :

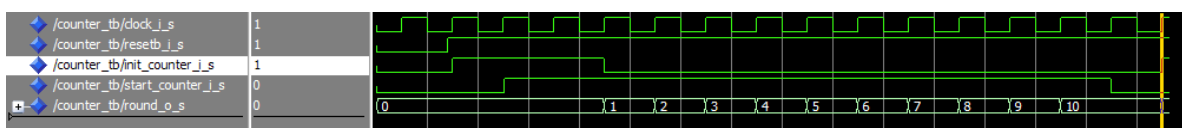


Figure 37 : Résultat obtenu pour le test Compteur

Toutes les assertions sont passés, donc **la machine d'état est validé**.

Manuellement : On peut voir que nous incrémentons jusqu'à 10, puis s'arrête quand start = 0, et se réinitialise quand init = 1.

## AES TestBench

Nous faisons 2 starts.

Les tests unitaires sont :

- Test 1 : "aes\_on\_o\_s = 1 après un start"
- Test 2 : "aes\_on\_o\_s = 0 à la fin et obtient le bon résultat"

**Résultat final :**

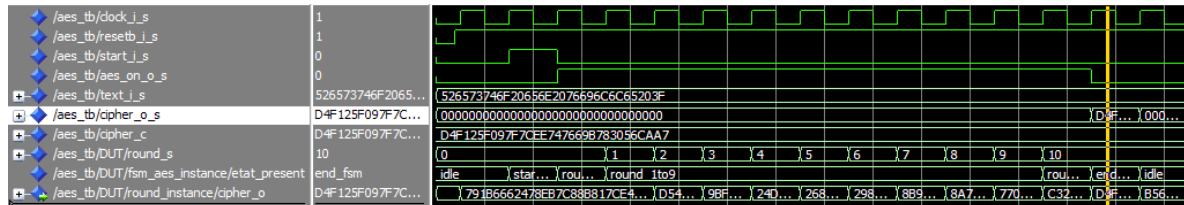


Figure 38 : Résultat obtenu pour le test AES

Toutes les assertions sont passées, donc **'AES' est validé**.

Manuellement : On peut vérifier chaque cipher\_o correspond aux fin des rounds :

Round 0

AddRoundKey : 79 1b 66 62 47 8e b7 c8 8b 81 7c e4 65 aa 6f 03

Round 1

AddRoundKey : d5 42 57 ea 74 cc c7 10 b5 60 66 f9 de 80 a1 b8

Round 2

AddRoundKey : 9b f4 64 34 f9 fb 21 92 36 a3 28 d6 e4 27 a8 4d

Round 3

AddRoundKey : 24 dd ad 90 50 14 a4 da co af 77 b9 oa 7a d6 d7

Round 4

AddRoundKey : 26 84 65 31 b7 10 13 be 29 24 bc 90 db a2 6c oc

Round 5

AddRoundKey : 29 8a b2 69 ab do 4b 5f 75 af af 5b c5 2c 50 56

Round 6

AddRoundKey : 8b 91 bo 15 24 bb 54 18 ba fc 4c 1d 42 3d 56 81

Round 7

AddRoundKey : 8a 78 9a 75 7b o7 fd 4b 28 93 38 7f 5b a5 ea eg

Round 8

AddRoundKey : 77 o7 6c ff 86 93 4a dc d8 61 6d b1 43 5c ca d4

Round 9

AddRoundKey : c3 29 63 ee d4 bf f1 38 75 f5 96 25 83 f1 64 of

Round 10

AddRoundKey : d4 f1 25 f0 97 f7 ce e7 47 66 9b 78 30 56 ca a7



# Conclusion

La modélisation VHDL du chiffrement AES est maintenant validé. Il ne reste plus qu'à intégrer un mode d'opération (tel que CBC ou Cipher Block Chaining) pour pouvoir chiffrer plusieurs blocs et intégrer au niveau matériel.

Ce projet n'a posé aucune difficulté, ni ralentissement et a été très formateur.