Rapport du projet AES VHDL

TODO: Not all test

TODO: Legend graph

TODO: Graph renvoi dans le texte

Plan

[Mettre Graphe]

SubBytes

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

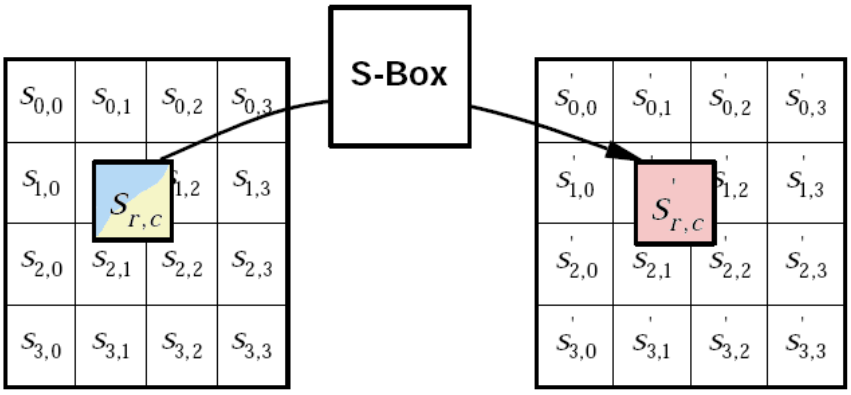


Figure TODO : Principe du SubBytes

SubBytes Entity

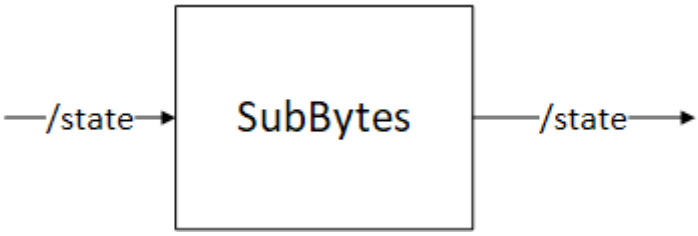


Figure TODO : SubBytes Entity

```
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.

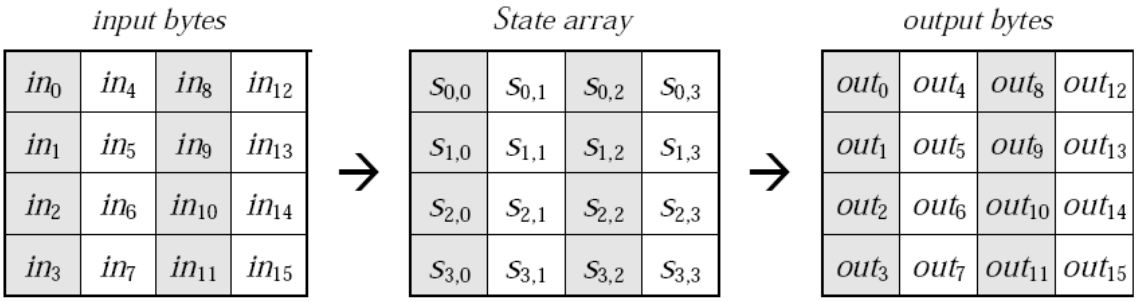


Figure TODO : Représentation d'un State array

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

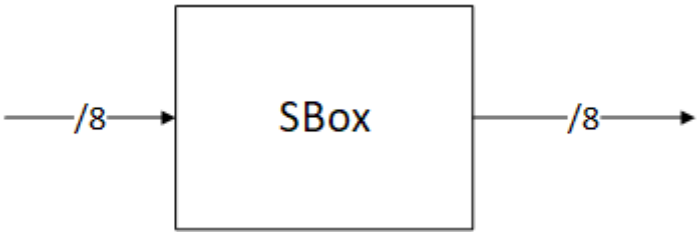


Figure TODO : SBox Entity

```
entity sbox is

  port (
    data_i: in bit8;
    data_o: out bit8
  );

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 TODO : SBox fournie à implémenter

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

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_stc.all` afin de convertir des `std_logic_vector` en `integer`.

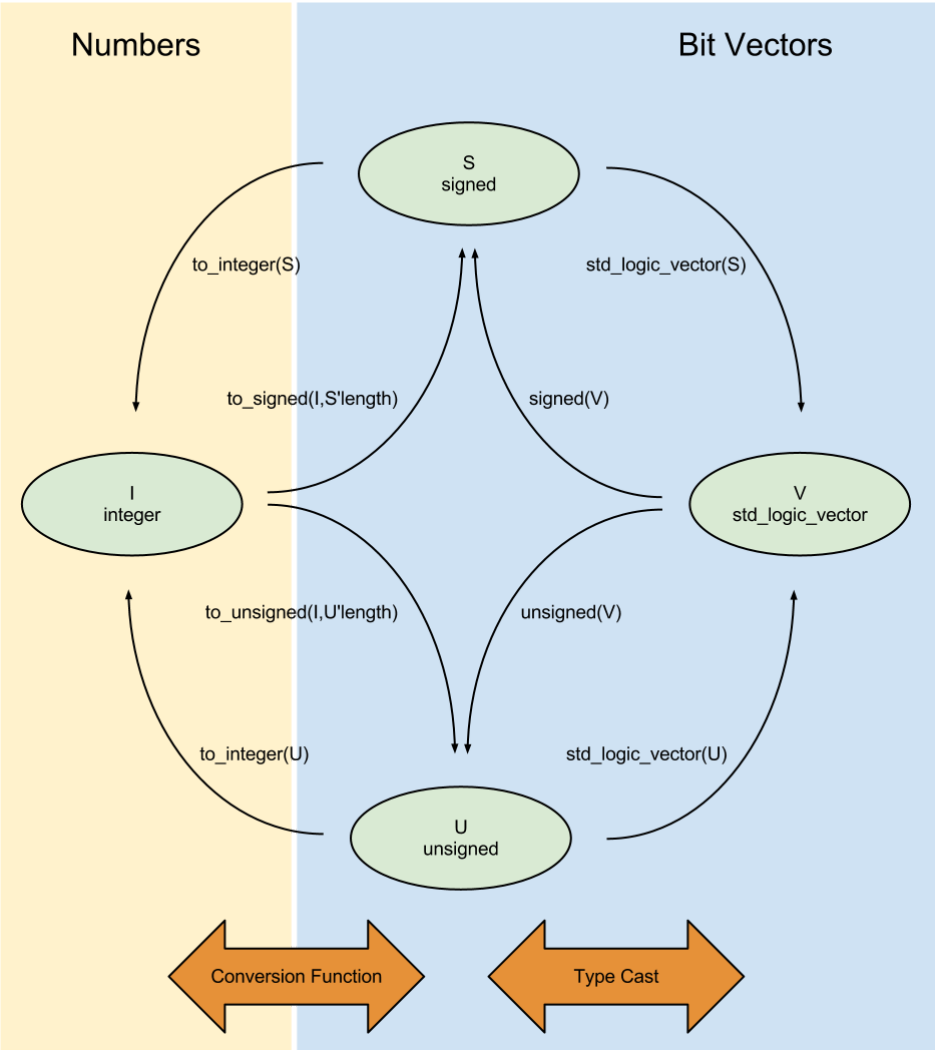


Figure TODO : Conversion des types en VHDL

```vhdl
-- Pour utiliser le type bit8
library lib_aes;
use lib_aes.crypt_pack.bit8;

-- Pour utiliser les types de std_logic_1164
library ieee;
use ieee.std_logic_1164.std_logic;

-- Pour convertir des std_logic_vector en integer
use ieee.numeric_std.unsigned;
use ieee.numeric_std.to_integer;

-- [...]

architecture sbox_arch of sbox is

    -- [...]

begin

  data_o <= sbox_c(to_integer(unsigned(data_i)));

end architecture sbox_arch;
```

Testbench

En entrée : Un variable allant de 0 à 255.

On s'attend à obtenir la sbox.

```
entity sbox_tb is
end entity sbox_tb;

architecture sbox_tb_arch of sbox_tb is

  -- Composant à tester
  component sbox
    port(
      data_i: in bit8;
      data_o: out bit8
    );
  end component;

  -- Signaux pour la simulation
  signal data_i_s: bit8;
  signal data_o_s: bit8;

  -- Signaux attendu (Arrange)
  type sbox_t is array (0 to 255) of bit8;
  constant sbox_c: sbox_t := (
    X"52", X"09", [...], X"0c", X"7d"
  );

begin

  DUT: sbox port map(
    data_i => data_i_s,
    data_o => data_o_s
  );

  test: process
  begin
    for stimuli in 0 to 255 loop
      data_i_s <= std_logic_vector(to_unsigned(stimuli, data_i_s'length));
-- Act
      wait for 5 ns;

      assert data_o_s=sbox_c(stimuli)  -- Assert
        report "Test has failed : data_o_s/=sbox_c(stimuli)"
        severity error;
      wait for 5 ns;
    end loop;
    assert false Report "Simulation Finished" severity failure;
  end process test;

end architecture sbox_tb_arch;
```

Résultat :

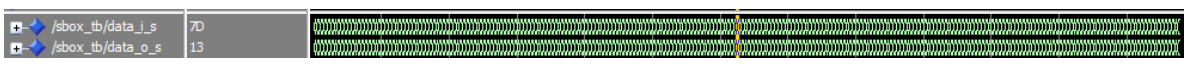


Figure TODO : Résultat obtenu pour le test SBox

Log :

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

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

Manuellement : sbox(0x7d) = 0x13.

SubBytes TestBench

En entrée :

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

Ce que l'on attend :

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

VHDL :

```
architecture subbytes_tb_arch of subbytes_tb is

  -- Composant à tester
  component subbytes
    port(
      data_i: in type_state;
      data_o: out type_state
    );
  end component;

  -- Signaux pour la simulation
  signal data_i_s: type_state;
  signal data_o_s: type_state;

  -- Arrange
  constant state_c: type_state := ((x"af", x"16", x"ce", x"bc"),
                                   (x"44", x"e6", x"91", x"62"),
                                   (x"d3", x"20", x"01", x"06"),
                                   (x"ab", x"b1", x"ab", x"d5"));

begin

  DUT: subbytes port map(
    data_i => data_i_s,
    data_o => data_o_s
  );
```

```
27
28    -- Act
29    data_i_s <= ((x"79", x"47", x"8b", x"65"),
30                 (x"1b", x"8e", x"81", x"aa"),
31                 (x"66", x"b7", x"7c", x"6f"),
32                 (x"62", x"c8", x"e4", x"03"));
33
34    test: process
35    begin
36      -- Assert
37      wait for 5 ns;
38      assert data_o_s = state_c
39        report "data_o_s /= state_c"
40        severity error;
41
42      -- End
43      wait for 5 ns;
44      assert false report "Simulation Finished" severity failure;
45    end process test;
46
47  end architecture subbytes_tb_arch;
```

Résultat :

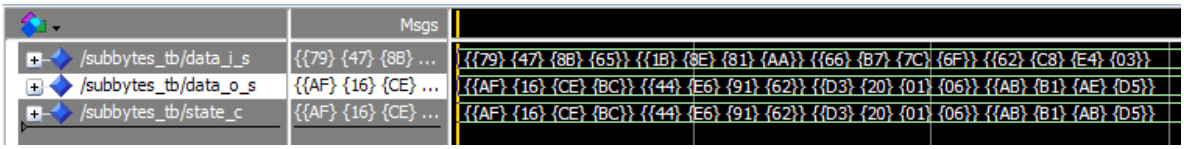


Figure TODO : Résultat obtenu pour le test SubBytes

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 TODO : Extrait de l'énoncé pour la validation SubBytes

ShiftRows

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

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

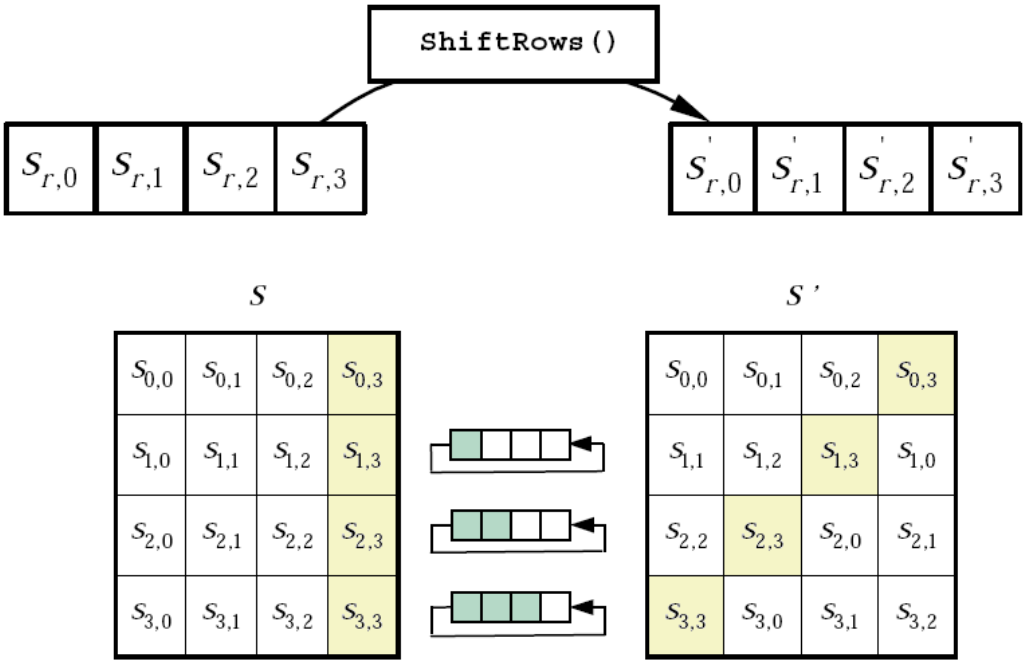


Figure TODO : Fonctionnement de ShiftRows

ShiftRows Entity

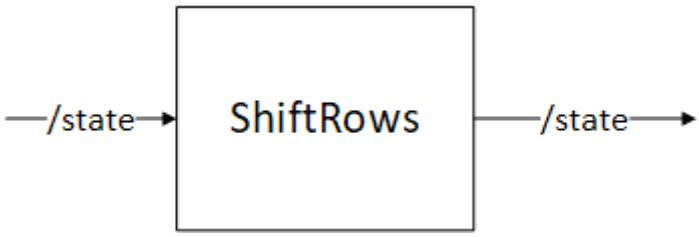


Figure TODO : ShiftRows Entity

```vhdl
entity subbytes is

  port (
    data_i: in type_state;
    data_o: out type_state
  );

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.

```
architecture shiftrows_arch of shiftrows is
begin

  rows: for i in 0 to 3 generate
    cases: for j in 0 to 3 generate
      data_o(i)(j) <= data_i(i)((i + j) mod 4);
    end generate cases;
  end generate rows;

end architecture shiftrows_arch;
```

ShiftRows TestBench

En entrée :

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

Ce que l'on attend :

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

VHDL :

```
architecture shiftrows_tb_arch of shiftrows_tb is

  -- Composant à tester
  component shiftrows
    port(
      data_i: in type_state;
      data_o: out type_state
    );
  end component;

  signal data_i_s: type_state;
  signal data_o_s: type_state;

  -- Arrange
  constant state_c: type_state := ((x"a0", x"29", x"43", x"21"),
                                   (x"ae", x"8e", x"d5", x"fa"),
                                   (x"2f", x"6d", x"d9", x"21"),
                                   (x"bc", x"e0", x"81", x"fc"));

begin

  DUT: shiftrows port map(
    data_i => data_i_s,
    data_o => data_o_s
  );

```

```
27    -- Act
28    data_i_s <= ((x"af", x"16", x"ce", x"bc"),
29                 (x"44", x"e6", x"91", x"62"),
30                 (x"d3", x"20", x"01", x"06"),
31                 (x"ab", x"b1", x"ae", x"d5"));
32
33    test: process
34    begin
35      -- Assert
36      wait for 5 ns;
37      assert data_o_s = state_c
38        report "data_o_s /= state_c"
39        severity error;
40
41      -- End
42      wait for 5 ns;
43      assert false report "Simulation Finished" severity failure;
44    end process test;
45
46  end architecture shiftrows_tb_arch;
```

Résultat :

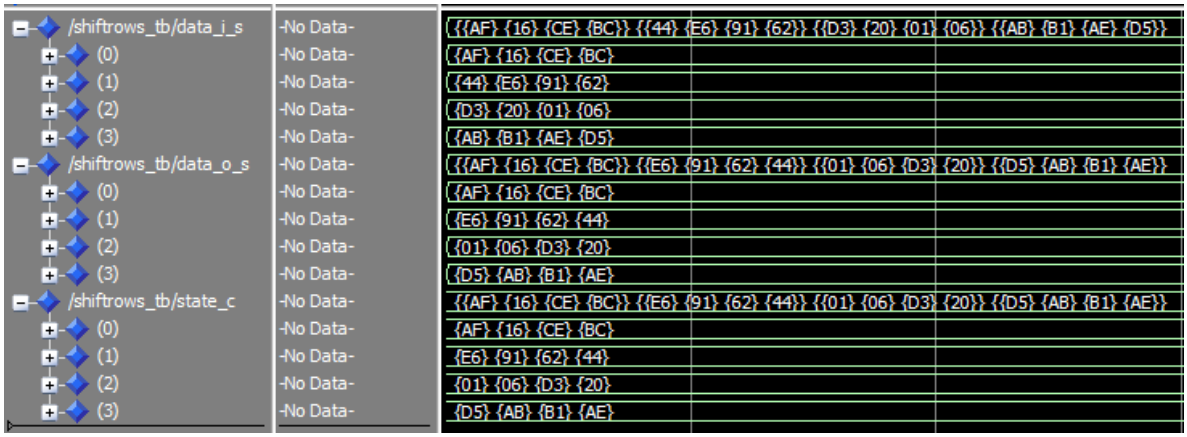


Figure TODO : Résultat obtenu pour le test ShiftRows

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 TODO : Extrait de l'énoncé pour la validation ShiftRows

MixColumns

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

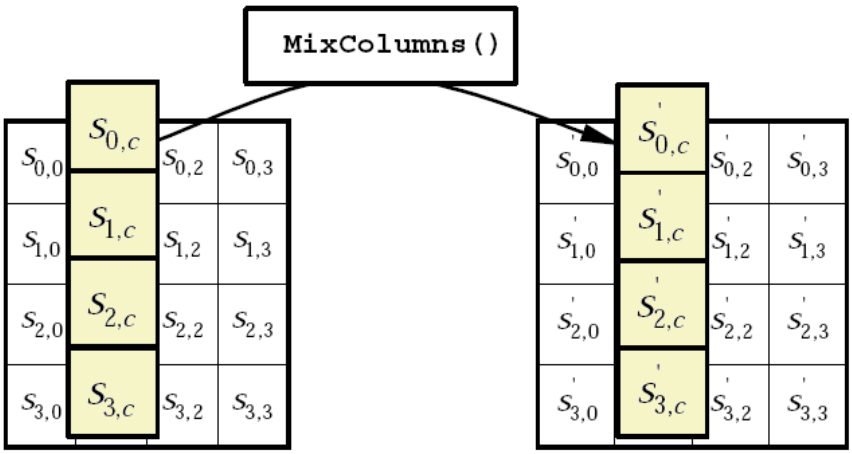


Figure TODO : Fonctionnement de MixColumns

MixColumns Entity

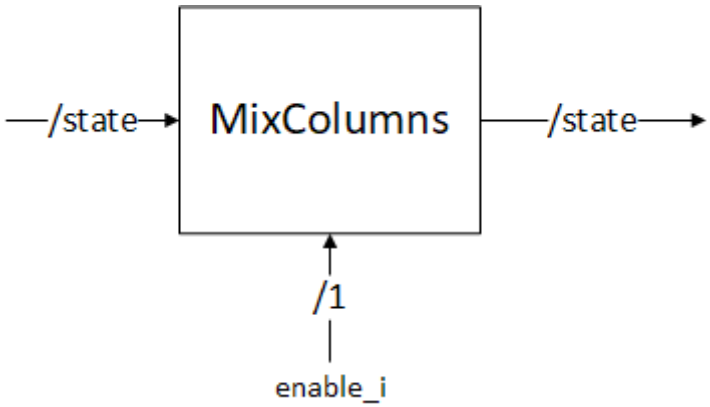


Figure TODO : MixColumns Entity

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 <= MixcColumns(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`

```
entity mixcolumns is

  port (
    data_i: in type_state;
    enable_i: in std_logic;
    data_o: out type_state
  );

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.

```
architecture mixcolumns_arch of mixcolumns is

  component mixcolumn
    port (
      data_i: in column_state;
      data_o: out column_state
    );
  end component;

  type state_col_major is array(0 to 3) of column_state;
  signal columns_i_s: state_col_major;
  signal columns_o_s: state_col_major;

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.

```
begin

  -- Slice Data_i in columns
  rows_order_i: for i in 0 to 3 generate
    columns_order_i: for j in 0 to 3 generate
      columns_i_s(j)(i) <= data_i(i)(j);
    end generate columns_order_i;
  end generate rows_order_i;

  -- Apply MixColumn for each column
  columns_order: for j in 0 to 3 generate
    MC: mixcolumn port map(
      data_i => columns_i_s(j),
      data_o => columns_o_s(j)
    );
  end generate columns_order;

  -- Restore state from new columns (or old columns depending enable_i)
  rows_order_o: for i in 0 to 3 generate
    columns_order_o: for j in 0 to 3 generate
      data_o(i)(j) <= columns_o_s(j)(i) when enable_i = '1' else data_i(i)
(j);
    end generate columns_order_o;
  end generate rows_order_o;

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 :

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

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

Figure TODO : Produit matriciel de la fonction MixColumns

Entity

Figure TODO : MixColumn Entity

```
entity mixcolumn is

  port (
    data_i: in column_state;
    data_o: out column_state
  );

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

```
architecture mixcolumn_arch of mixcolumn is
  --[...]
  signal data2_s: column_state;

begin

  data2_s <= (
    std_logic_vector((unsigned(data_i(0)(6 downto 0)) & "0") xor
      ("000" & data_i(0)(7) & data_i(0)(7) & "0" & data_i(0)(7) & data_i(0)(7))),
    std_logic_vector((unsigned(data_i(1)(6 downto 0)) & "0") xor
      ("000" & data_i(1)(7) & data_i(1)(7) & "0" & data_i(1)(7) & data_i(1)(7))),
    std_logic_vector((unsigned(data_i(2)(6 downto 0)) & "0") xor
      ("000" & data_i(2)(7) & data_i(2)(7) & "0" & data_i(2)(7) & data_i(2)(7))),
    std_logic_vector((unsigned(data_i(3)(6 downto 0)) & "0") xor
      ("000" & data_i(3)(7) & data_i(3)(7) & "0" & data_i(3)(7) & data_i(3)(7)))
  );
```

- `data3_s` est la colonne en entrée multiplié par 3 dans l'ensemble de Galois

```
architecture mixcolumn_arch of mixcolumn is

  signal data2_s: column_state;
  signal data3_s: column_state;

begin
  --[...]
  data3_s <= (
    data2_s(0) xor data_i(0),
    data2_s(1) xor data_i(1),
    data2_s(2) xor data_i(2),
    data2_s(3) xor data_i(3)
  );
```

- On applique `data_o` les opérations décrites ci-dessus :

```
architecture mixcolumn_arch of mixcolumn is

  signal data2_s: column_state;
  signal data3_s: column_state;

begin

  --[...]
  data_o(0) <= data2_s(0) xor data3_s(1) xor data_i(2) xor data_i(3);
  data_o(1) <= data_i(0) xor data2_s(1) xor data3_s(2) xor data_i(3);
  data_o(2) <= data_i(0) xor data_i(1) xor data2_s(2) xor data3_s(3);
  data_o(3) <= data3_s(0) xor data_i(1) xor data_i(2) xor data2_s(3);

end architecture mixcolumn_arch;
```

Testbench

En entrée : une colonne

```
(x"af", x"44", x"d3", x"ab")
```

Ce que l'on attend :

```
(x"a0", x"ae", x"2f", x"bc")
```

VHDL :

```
architecture mixcolumn_tb_arch of mixcolumn_tb is

  -- Composant à tester
  component mixcolumn
    port(
      data_i: in column_state;
      data_o: out column_state
    );
  end component;

  -- Signaux pour la simulation
  signal data_i_s: column_state;
  signal data_o_s: column_state;

  -- Arrange
  constant column_c: column_state := (x"a0", x"ae", x"2f", x"bc");

begin

  DUT: mixcolumn port map(
    data_i => data_i_s,
    data_o => data_o_s
  );

  -- Act
  data_i_s <= (x"af", x"44", x"d3", x"ab");

  test: process
  begin
    -- Assert
    wait for 5 ns;
    assert data_o_s = column_c
      report "data_o_s /= column_c"
      severity error;

    -- End
    wait for 5 ns;
    assert false report "Simulation Finished" severity failure;
  end process test;

end architecture mixcolumn_tb_arch;
```

Résultat :

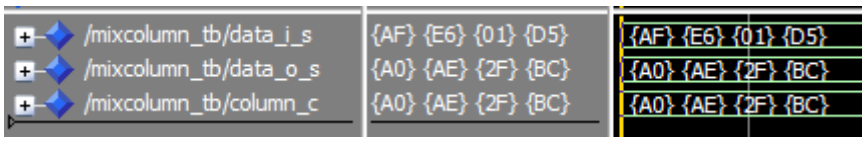


Figure TODO : Résultat obtenu pour le test MixColumn

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 TODO : Extrait de l'énoncé pour la validation MixColumn

MixColumns TestBench

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

```
  data_i_s <= ((x"af", x"16", x"ce", x"bc"),
               (x"e6", x"91", x"62", x"44"),
               (x"01", x"06", x"d3", x"20"),
               (x"d5", x"ab", x"b1", x"ae"));

  enable_i_s <= '0',
                '1' after 50 ns;
```

Ce que l'on attend : Quand `enable_i = 1`

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

Quand `enable_i = 0`

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

VHDL :

```
architecture mixcolumns_tb_arch of mixcolumns_tb is

  -- Composant à tester
  component mixcolumns
    port(
      data_i: in type_state;
      enable_i: in std_logic;
      data_o: out type_state
    );
  end component;

```

```
  -- Signaux pour la simulation
  signal data_i_s: type_state;
  signal data_o_s: type_state;
  signal enable_i_s: std_logic;

  -- Arrange
  constant state_when_enabled_c: type_state := ((x"a0", x"29", x"43", x"21"),
                                                (x"ae", x"8e", x"d5", x"fa"),
                                                (x"2f", x"6d", x"d9", x"51"),
                                                (x"bc", x"e0", x"81", x"fc"));
  constant state_when_disabled_c: type_state := ((x"af", x"16", x"ce", x"bc"),
                                                 (x"e6", x"91", x"62", x"44"),
                                                 (x"01", x"06", x"d3", x"20"),
                                                 (x"d5", x"ab", x"b1", x"ae"));
begin

  DUT: mixcolumns port map(
    data_i => data_i_s,
    enable_i => enable_i_s,
    data_o => data_o_s
  );

  -- Stimuli
  data_i_s <= ((x"af", x"16", x"ce", x"bc"),
               (x"e6", x"91", x"62", x"44"),
               (x"01", x"06", x"d3", x"20"),
               (x"d5", x"ab", x"b1", x"ae"));

  enable_i_s <= '0',
                '1' after 50 ns;

  test: process
  begin
    -- Assert
    wait for 5 ns;
    assert data_o_s = state_when_disabled_c
      report "data_o_s /= state_when_disabled_c"
      severity error;

    wait for 50 ns;
    assert data_o_s = state_when_enabled_c
      report "data_o_s /= state_when_enabled_c"
      severity error;

    -- End
    wait for 5 ns;
    assert false report "Simulation Finished" severity failure;
  end process test;

end architecture mixcolumns_tb_arch;
```

Résultat :

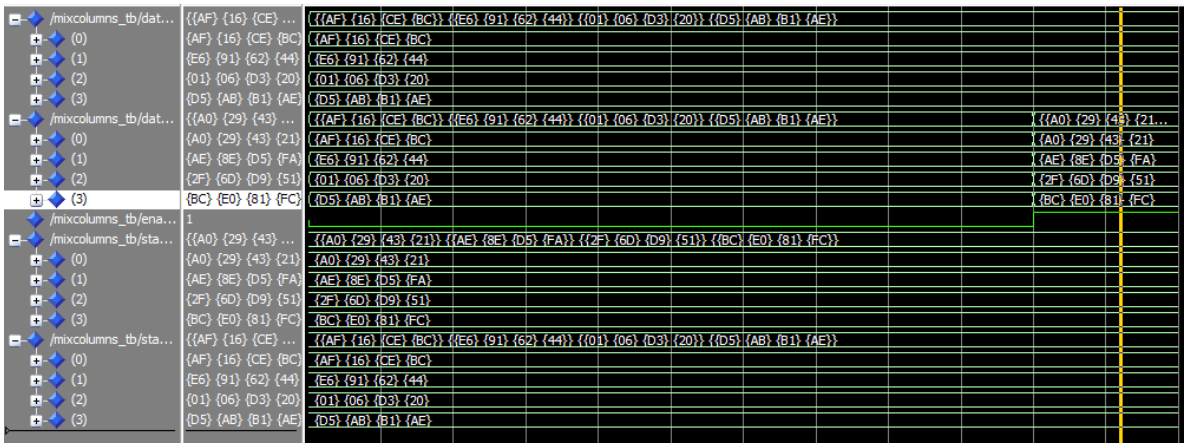


Figure TODO : Résultat obtenu pour le test MixColumns

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

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

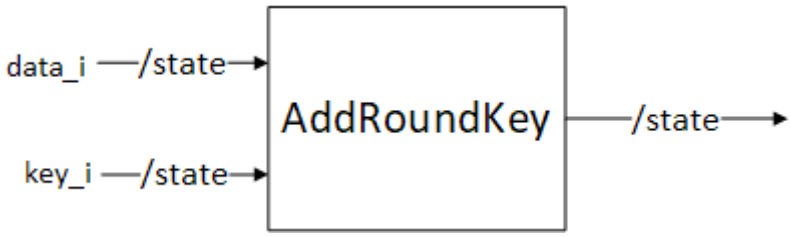
```
ShiftRows : af e6 01 d5 16 91 06 ab ce 62 d3 b1 bc 44 20 ae
MixColumns : a0 ae 2f bc 29 8e 6d e0 43 d5 d9 81 21 fa 51 fc
```

Figure TODO : 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



VHDL :

```vhdl
entity addroundkey is

  port (
    data_i: in type_state;
    key_i: in type_state;
    data_o: out type_state
  );

end entity addroundkey;
```

AddRoundKey Architecture

Le résultat est immédiat :

```
architecture addroundkey_arch of addroundkey is
begin

  rows: for i in 0 to 3 generate
    cases: for j in 0 to 3 generate
      data_o(i)(j) <= data_i(i)(j) xor key_i(i)(j);
    end generate rows;
  end generate cases;

end architecture;
```

AddRoundKey TestBench

En entrée : un state et une sous-clé

State :

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

Sous-clé :

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

Ce que l'on attend :

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

VHDL :

```
architecture addroundkey_tb_arch of addroundkey_tb is

  -- Composant à tester
  component addroundkey
    port(
      data_i: in type_state;
      key_i: in type_state;
      data_o: out type_state
    );
  end component;

  -- Signaux pour la simulation
  signal data_i_s: type_state;
  signal key_i_s: type_state;
  signal data_o_s: type_state;

  -- Arrange
```

```
  constant state_c: type_state := ((x"79", x"47", x"8b", x"65"),
                                   (x"1b", x"8e", x"81", x"aa"),
                                   (x"66", x"b7", x"7c", x"6f"),
                                   (x"62", x"c8", x"e4", x"03"));

begin

  DUT: addroundkey port map(
    data_i => data_i_s,
    key_i => key_i_s,
    data_o => data_o_s
  );

  -- Act
  data_i_s <= ((x"52", x"6f", x"20", x"6c"),
               (x"65", x"20", x"76", x"65"),
               (x"73", x"65", x"69", x"20"),
               (x"74", x"6e", x"6c", x"3f"));

  key_i_s <= ((x"2b", x"28", x"ab", x"09"),
              (x"7e", x"ae", x"f7", x"cf"),
              (x"15", x"d2", x"15", x"4f"),
              (x"16", x"a6", x"88", x"3c"));

  test: process
  begin
    -- Assert
    wait for 5 ns;
    assert data_o_s = state_c
      report "data_o_s /= state_c"
      severity error;

    -- End
    wait for 5 ns;
    assert false report "Simulation Finished" severity failure;
  end process test;

end architecture addroundkey_tb_arch;
```

Résultat :

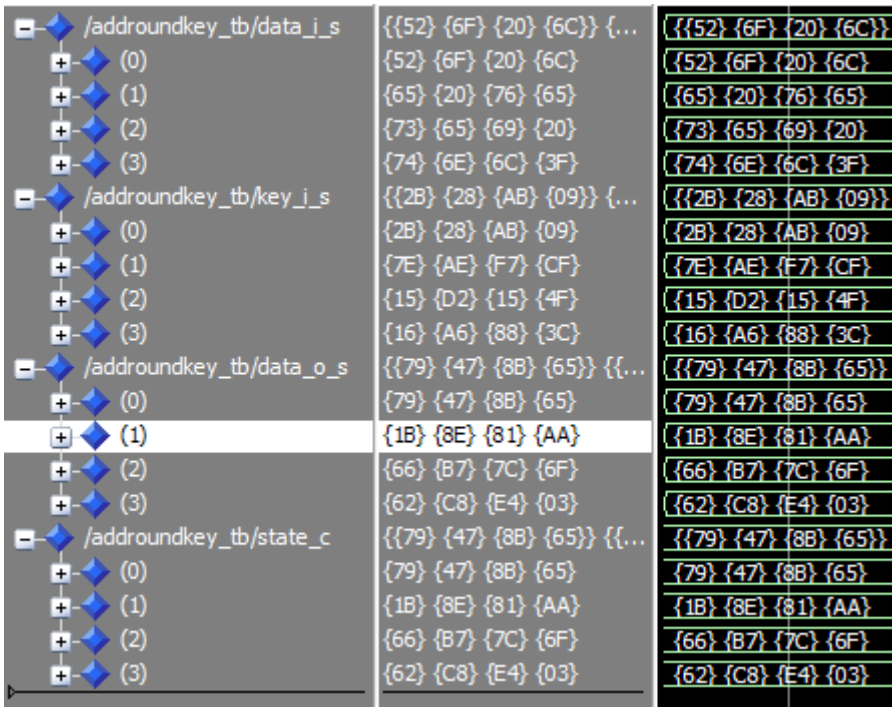


Figure TODO : Résultat obtenu pour le test AddRoundKey

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 TODO : Extrait de l'énoncé pour la validation AddRoundKey

Round

Un round doit appliquer toute les fonctions que nous avons développé jusque là. En fonction du nombre de round, nous devrons sélectionner quel fonction appliquer :

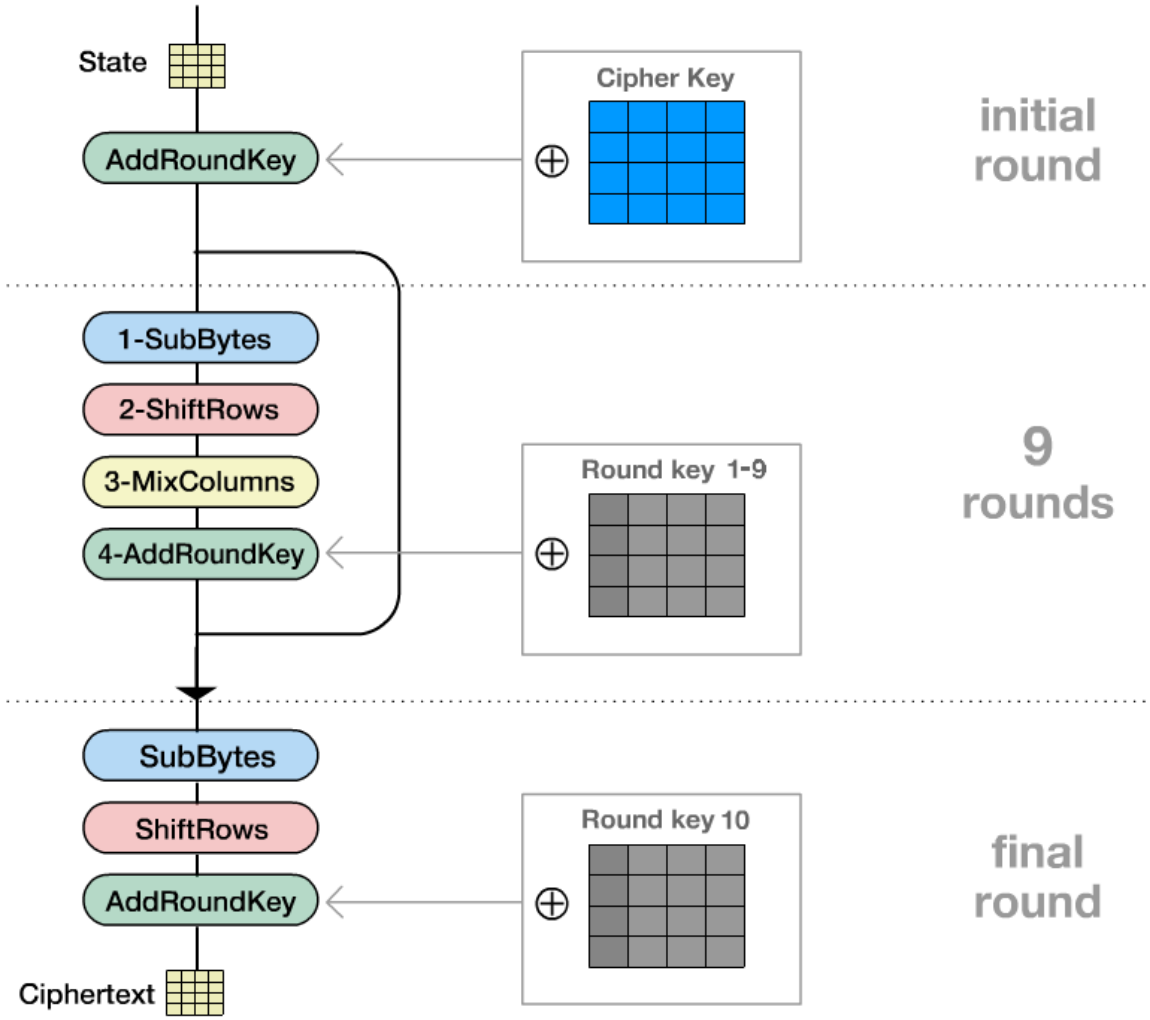


Figure TODO : Composition des rounds

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

Round Entity

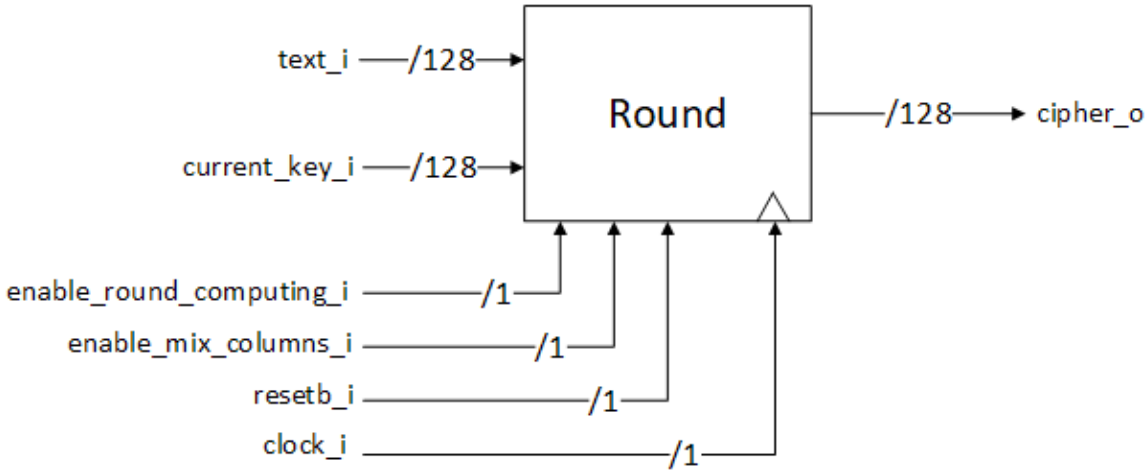


Figure TODO : Round Entity

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`

```
entity round is

  port (
    text_i: in bit128;
    current_key_i: in bit128;
    clock_i: in std_logic;
    resetb_i: in std_logic;
    enable_round_computing_i: in std_logic;
    enable_mix_columns_i: in std_logic;
    cipher_o: out bit128
  );

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 :

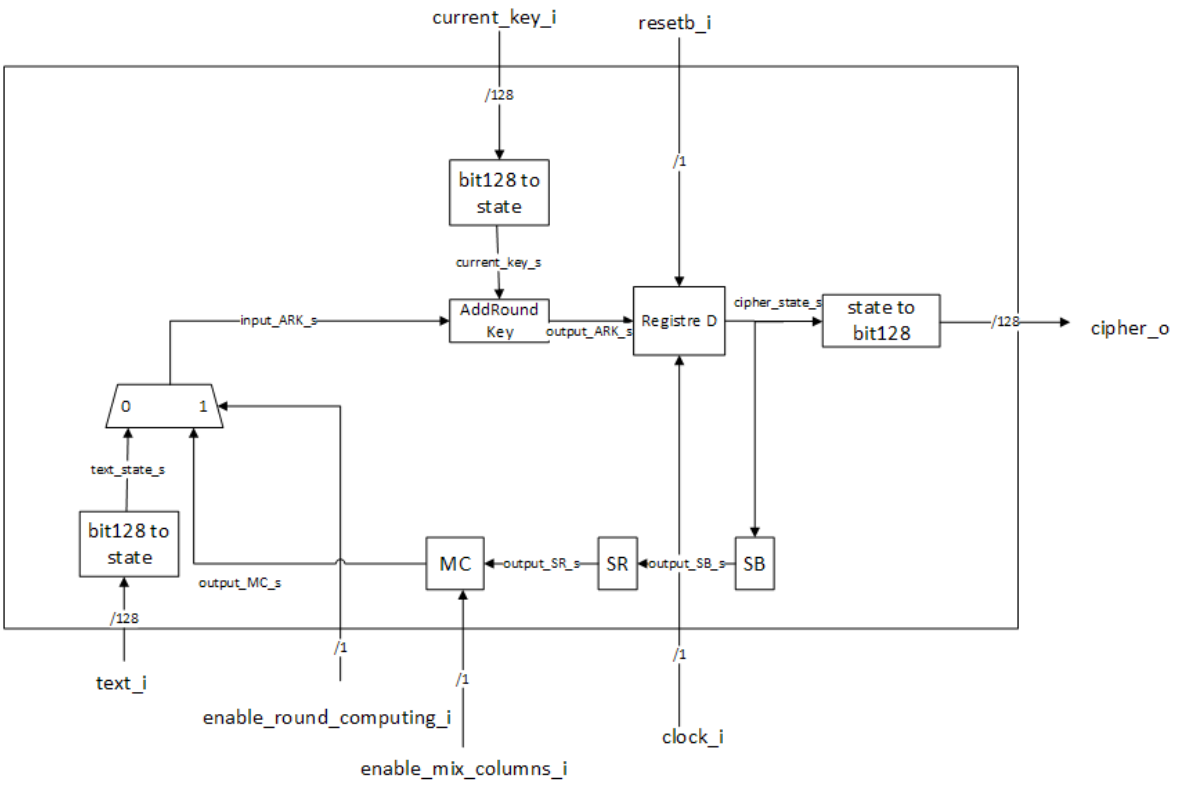


Figure TODO : Round Architecture

On connecte donc nos différents composants en VHDL :

```
architecture round_arch of round is

  ...  -- Déclaration des composants et signaux affichés dans le schéma

begin

  text_bit128_to_state: bit128_to_state
    port map(
      data_i => text_i,
      data_o => text_state_s
    );


  -- demux
  input_ARK_s <= output_MC_s when enable_round_computing_i = '1' else
text_state_s;

  current_key_bit128_to_state: bit128_to_state
    port map(
      data_i => current_key_i,
      data_o => current_key_s
    );

  addroundkey_instance: addroundkey
    port map(
      data_i => input_ARK_s,
      key_i => current_key_s,
      data_o => output_ARK_s
    );

  register_d_instance: register_d
    port map(
      resetb_i => resetb_i,
      clock_i => clock_i,
      state_i => output_ARK_s,
      state_o => cipher_state_s
    );

  cipher_to_bit128: state_to_bit128
    port map(
      data_i => cipher_state_s,
      data_o => cipher_o
    );

  subbytes_instance: subbytes
    port map(
      data_i => cipher_state_s,
      data_o => output_SB_s
    );

  shiftrows_instance: shiftrows
    port map(
      data_i => output_SB_s,
      data_o => output_SR_s
    );

  mixcolumns_instance: mixcolumns
    port map(
```

```
      data_i => output_SR_s,
      data_o => output_MC_s,
      enable_i => enable_mix_columns_i
    );


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

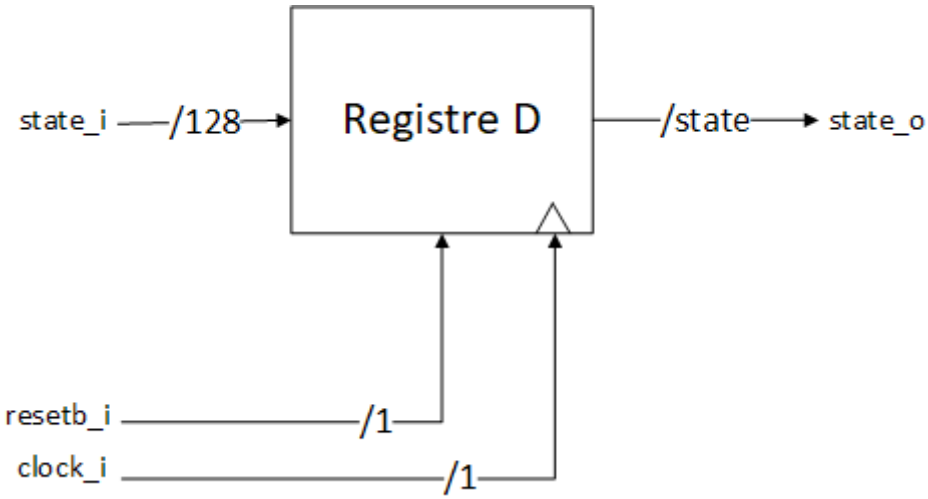


Figure TODO : Registre D Entity

```
entity register_d is

  port (
    resetb_i : in std_logic;
    clock_i : in std_logic;
    state_i : in type_state;
    state_o : out type_state
  );

end entity register_d;
```

Architecture

On adapte notre registre D au type `state`.

```
architecture register_d_arch of register_d is

  signal state_s : type_state;

begin

  seq_0 : process (clock_i, resetb_i) is

  begin
```

```
    -- Reset clears state
    if resetb_i = '0' then
      for i in 0 to 3 loop
        for j in 0 to 3 loop
          state_s(i)(j) <= (others => '0');
        end loop;
      end loop;

    -- New data at RISING
    elsif clock_i'event and clock_i='1' then
      state_s <= state_i;
    end if;

  end process seq_0;

  -- Output
  state_o <= state_s;

end architecture register_d_arch;
```

TestBench

```
architecture register_d_tb_arch of register_d_tb is

  component register_d
    port (
      resetb_i : in std_logic;
      clock_i : in std_logic;
      state_i : in type_state;
      state_o : out type_state
    );
  end component register_d;

  signal resetb_i_s : std_logic;
  signal clock_i_s : std_logic;
  signal state_i_s : type_state;
  signal state_o_s : type_state;

  -- Arrange
  constant unintialized_state: type_state := ((x"00", x"00", x"00", x"00"),
                                              (x"00", x"00", x"00", x"00"),
                                              (x"00", x"00", x"00", x"00"),
                                              (x"00", x"00", x"00",
x"00"));
  constant first_state: type_state := ((x"00", x"04", x"08", x"0C"),
                                       (x"01", x"05", x"09", x"0D"),
                                       (x"02", x"06", x"0A", x"0E"),
                                       (x"03", x"07", x"0B", x"0F"));
  constant second_state: type_state := ((x"DE", x"AD", x"BE", x"EF"),
                                        (x"BA", x"DD", x"CA", x"FE"),
                                        (x"DE", x"AD", x"C0", x"DE"),
                                        (x"CA", x"DE", x"D0", x"0D"));

begin

  DUT: register_d
```

```
    port map(
      resetb_i => resetb_i_s,
      clock_i => clock_i_s,
      state_i => state_i_s,
      state_o => state_o_s
    );

  resetb_i_s <= '0', '1' after 5 ns, '0' after 290 ns;

  -- CLK T = 100 ns
  clock_i_s <= '0', '1' after 50 ns,
               '0' after 100 ns, '1' after 150 ns,
               '0' after 200 ns, '1' after 250 ns,
               '0' after 300 ns, '1' after 350 ns,
               '0' after 400 ns, '1' after 450 ns,
               '0' after 500 ns, '1' after 550 ns;

  state_i_s <= ((x"00", x"04", x"08", x"0C"),
                (x"01", x"05", x"09", x"0D"),
                (x"02", x"06", x"0A", x"0E"),
                (x"03", x"07", x"0B", x"0F")),
               ((x"DE", x"AD", x"BE", x"EF"),
                (x"BA", x"DD", x"CA", x"FE"),
                (x"DE", x"AD", x"C0", x"DE"),
                (x"CA", x"DE", x"D0", x"0D")) after 100 ns;

  test: process
  begin
    -- Assert :
    -- Test : Register D is not initialized before RISING
    wait for 5 ns;  -- t = 5 ns
    assert state_o_s = unintialized_state
      report "state_o_s /= unintialized_state"
      severity error;

    -- Test : Register D is now initialized  after RISING
    wait for 50 ns;  -- t = 55 ns
    assert state_o_s = first_state
      report "state_o_s /= first_state"
      severity error;

    -- Test : Register D change state after RISING
    wait for 100 ns;  -- t = 155 ns
    assert state_o_s = second_state
      report "state_o_s /= second_state"
      severity error;

    -- Test : Register D reset suddenly without caring about clock_i
    wait for 140 ns;  -- t = 295 ns
    assert state_o_s = unintialized_state
      report "state_o_s /= unintialized_state"
      severity error;

    -- End
    wait for 5 ns;
    assert false report "Simulation Finished" severity failure;
  end process test;
end architecture register_d_tb_arch;
```

Résultat :

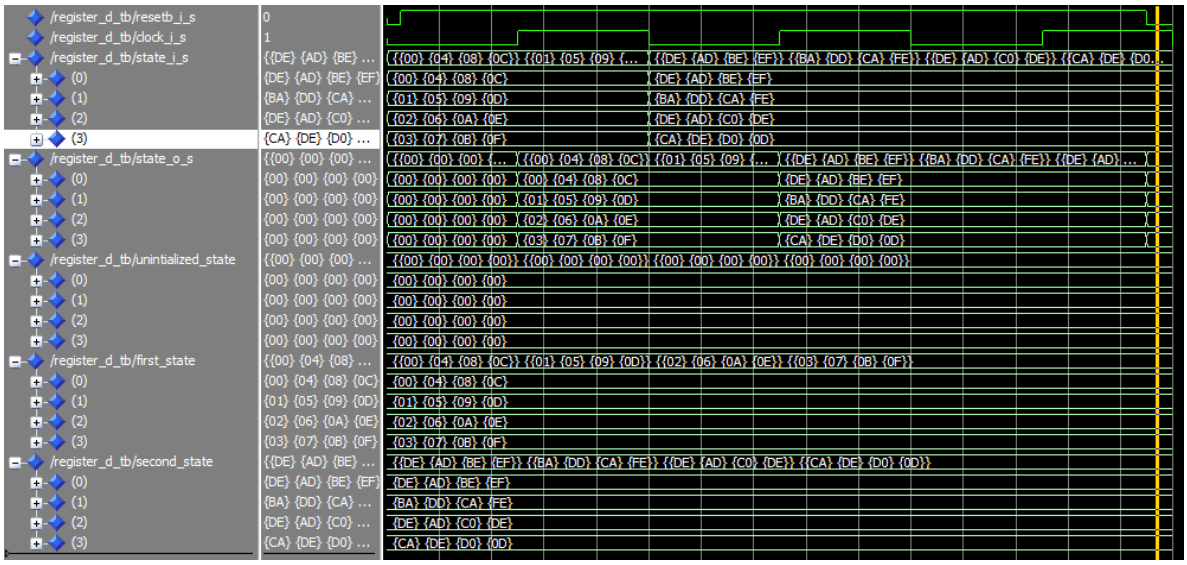


Figure TODO : Résultat obtenu pour le test Registre D

Toutes les assertions sont passés, donc **Registre D est validé**.

Component state_to_bit128 et bit128_to_state

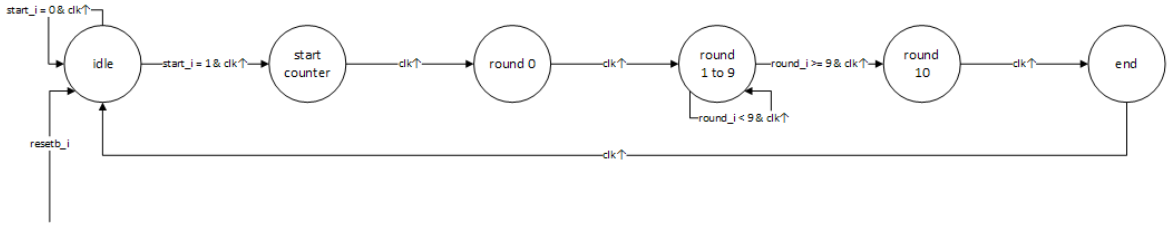
La relation entrée/sortie avec le composant est assez explicite.

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	…

Round TestBench

Machine d'Etat

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



Voici donc la configuration des données en fonction des états :

	idle	start_counter	round_0	round_1to9	round10	end_fsm
init_counter_o	**1**	**1**	0	0	0	0
start_counter_o	0	**1**	**1**	**1**	0	0
enable_output_o	0	0	0	0	0	**1**
aes_on_o	0	0	**1**	**1**	**1**	0
enable_RC_o	0	0	0	**1**	**1**	0
enable_MC_o	0	0	0	**1**	0	0

Machine d'Etat Entity

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

```
entity fsm_aes is

  port (
    round_i: in bit4;  -- Utilise: 10, Max: 16
    clock_i: in std_logic;
    resetb_i: in std_logic;
    start_i: in std_logic;
    init_counter_o: out std_logic;
    start_counter_o: out std_logic;
    enable_output_o: out std_logic;
    aes_on_o: out std_logic;
    enable_round_computing_o: out std_logic;
    enable_mix_columns_o: out std_logic
  );

end entity fsm_aes;
```

Machine d'Etat Architecture

Dans la partie déclarative, on définit nos états :

```
architecture fsm_aes_arch of fsm_aes is

  type state_fsm is (idle, start_counter, round_0, round_1to9, round10, end_fsm);
  signal etat_present, etat_futur: state_fsm;

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 :

```
-- architecture fsm_aes_arch
  event_dispatcher: process (clock_i, resetb_i)
  begin
    if resetb_i = '0' then
      etat_present <= idle;
    elsif clock_i'event and clock_i = '1' then
      etat_present <= etat_futur;
    end if;
  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 :

```
-- architecture fsm_aes_arch
  event_map_to_state: process (etat_present, start_i, round_i)
  begin
    case etat_present is
      when idle =>
        if start_i = '0' then
          etat_futur <= idle;  -- loop until start
        else
          etat_futur <= start_counter;
        end if;
      when start_counter =>
        etat_futur <= round_0;
      when round_0 =>
        etat_futur <= round_1to9;
      when round_1to9 =>
        if to_integer(unsigned(round_i)) < 9 then
          etat_futur <= round_1to9;  -- loop while round_i < 9
        else
          etat_futur <= round10;
        end if;
      when round10 =>
        etat_futur <= end_fsm;
      when end_fsm =>
        etat_futur <= idle;
    end case;
  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 :

```
-- architecture fsm_aes_arch
  state_model: process (etat_present)
  begin
    case etat_present is
      when idle =>
        init_counter_o <= '1';
        start_counter_o <= '0';
        enable_output_o <= '0';
        aes_on_o <= '0';
        enable_round_computing_o <= '0';
        enable_mix_columns_o <= '0';
      ... -- Le comportement est défini dans le tableau ci-dessus.
      when end_fsm =>
```

```
        init_counter_o <= '0';
        start_counter_o <= '0';
        enable_output_o <= '1';
        aes_on_o <= '0';
        enable_round_computing_o <= '0';
        enable_mix_columns_o <= '0';
    end case;
  end process state_model;
```

Machine d'Etat TestBench

Compteur de Round

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

Compteur de Round Entity

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

```
entity counter is

  port (
    clock_i: in std_logic;
    resetb_i: in std_logic;
    init_counter_i: in std_logic;
    start_counter_i: in std_logic;
    round_o: out bit4
  );

end entity counter;
```

Compteur de Round Architecture

```
architecture counter_arch of counter is

  signal round_s : bit4;

begin

  seq_0 : process (clock_i, resetb_i) is
  begin
    -- Reset clears state
    if resetb_i = '0' then
      round_s <= "0000";

    -- New data at RISING
    elsif clock_i'event and clock_i = '1' then
      -- Arm
      if init_counter_i = '1' then
        round_s <= "0000";

      -- Start counting
      elsif start_counter_i = '1' then
        if round_s = "1010" then  -- Limit
```

```
          round_s <= "0000";
        else
          round_s <= std_logic_vector(unsigned(round_s) + 1);
        end if;
      end if;
    end if;
  end process seq_0;

  round_o <= round_s;

end architecture counter_arch;
```

Compteur de Round TestBench

AES

Bibliographie

Annexe