

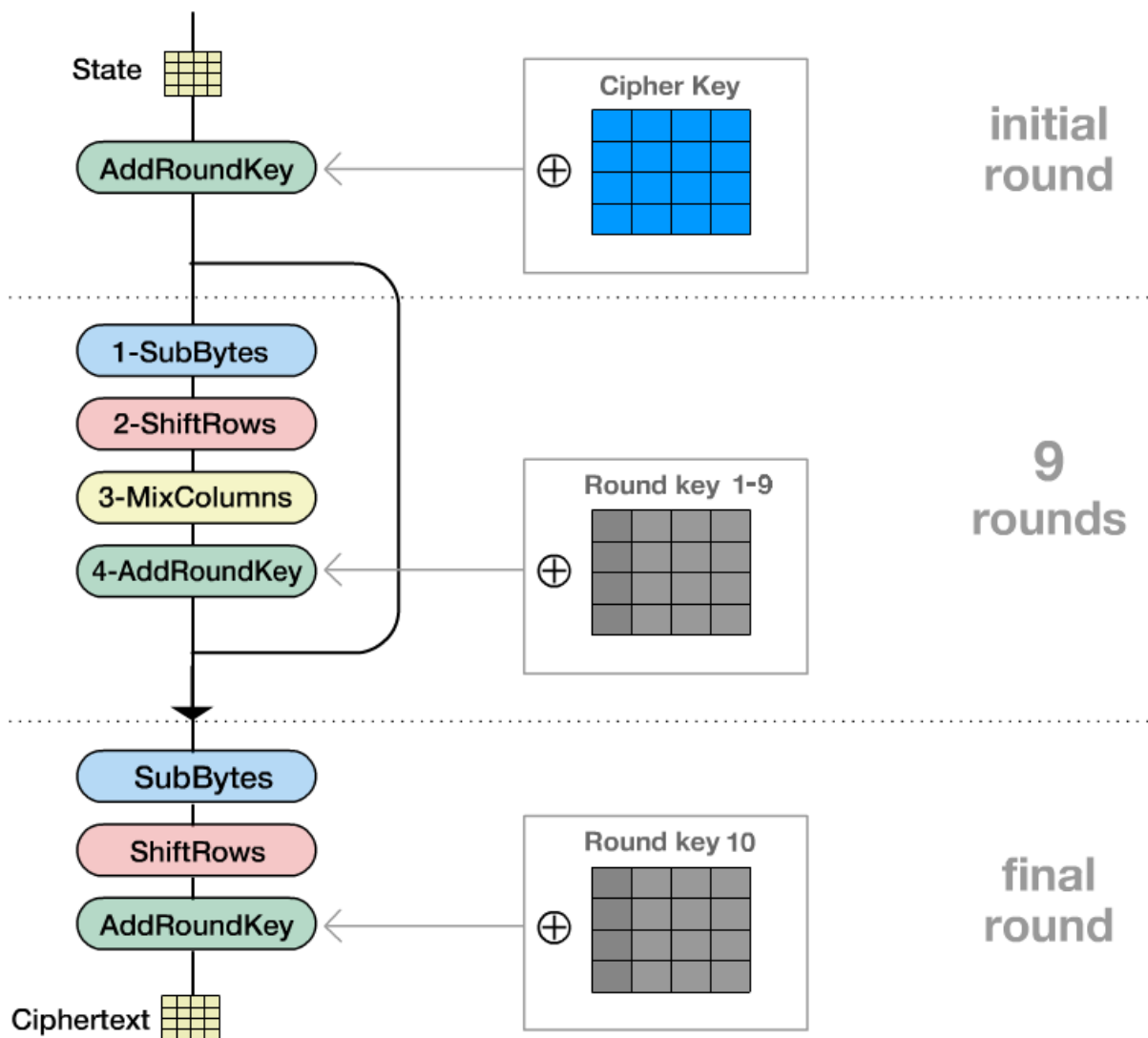
Modélisation VHDL du chiffrement AES

Le but de ce projet est de modéliser par le langage VHDL un module de chiffrement AES. Pour cela nous avons modélisé les différents blocs de traitements. Ils vous seront présentés en détail dans la suite du rapport.

Présentation de l'AES :

L'algorithme de cryptage AES peut être décomposé en trois phases. La phase d'initialisation ou de première itération, la (les) phase(s) d'itérations, puis l'itération finale.

Ce qui donne l'organigramme suivant :



En plus des composants présentés dans l'organigramme, il faudra penser à modéliser le composant générant les clés de cryptage associé à chacun des rounds. Nous appellerons ce composant KeyExpander_IO.

Composant SubBytes :

Ce composant permet d'appliquer une transformation non linéaire à un tableau via une table de substitution communément appelée Sbox.

Ce composant prend donc en entrée un l'état courant et renvoie en sortie l'état courant auquel on aura appliqué la transformation.

L'entité se résume donc à :

```
entity subBytes is
  port(
    data_i : in type_state;
    data_o : out type_state
  );
end entity subBytes;
```

Pour maximiser la clarté du code, nous avons décidé de créer un composant nommé Sbox qui effectue la transformation sur 1 octet uniquement.

```
entity sbox is
  port(
    SBOX_I : in bit8;
    SBOX_O : out bit8
  );
end entity sbox;
```

On y déclare dedans la table de substitution, et on renvoie l'octet modifié

```
----- Sbox.vhd -----
SBOX_O <= sbox_c(to_integer(unsigned(SBOX_I)));
```

Le composant Sbox permet de simplifier l'écriture du composant SubBytes. En effet, il suffit "d'appliquer" le composant Sbox à chacun des octets de l'entrée de SubBytes.

Pour cela deux méthodes sont possibles.

- **Méthode 1 :** On instancie un unique composant Sbox dans SubBytes et on change les valeurs d'entrées de Sbox par un process dans SubBytes
- **Méthode 2 :** On instancie une Sbox par octet. Cela peut s'automatiser grâce à l'instruction for ... generate.

Nous avons choisi la méthode 2. Le code donne donc :

```

----SubBytes.vhd----
11 : for i in 0 to 3 generate
    12 : for j in 0 to 3 generate
        U0 : Sbox
        port map (
            SBOX_I => data_i(i)(j),
            SBOX_O => data_o(i)(j)
        );
    end generate 12;
end generate 11;

```

Test Unitaire SubBytes :

Wave - Default		Msgs
/subbytes_tb/data_i_s		{{8'h00} {8'h01} {8'h02} {8'h03}} ...
(0)		{8'h00} {8'h01} {8'h02} {8'h03}
(1)		{8'h10} {8'h11} {8'h12} {8'h13}
(2)		{8'h20} {8'h21} {8'h22} {8'h23}
(3)		{8'h30} {8'h31} {8'h32} {8'h33}
/subbytes_tb/data_o_s		{{8'h63} {8'h7C} {8'h77} {8'h7B}} ...
(0)		{8'h63} {8'h7C} {8'h77} {8'h7B}
(1)		{8'hCA} {8'h82} {8'hC9} {8'h7D}
(2)		{8'hB7} {8'hFD} {8'h93} {8'h26}
(3)		{8'h04} {8'hC7} {8'h23} {8'hC3}

On a mis en entrée de SubBytes data_i_s. La sortie correspond bien à ce qu'on voulait.

Composant ShiftRow :

Ce composant applique une transformation linéaire aux lignes de la matrice de notre état. Celle-ci se résume à une permutation des octets d'une ligne en fonction de son numéro de ligne.

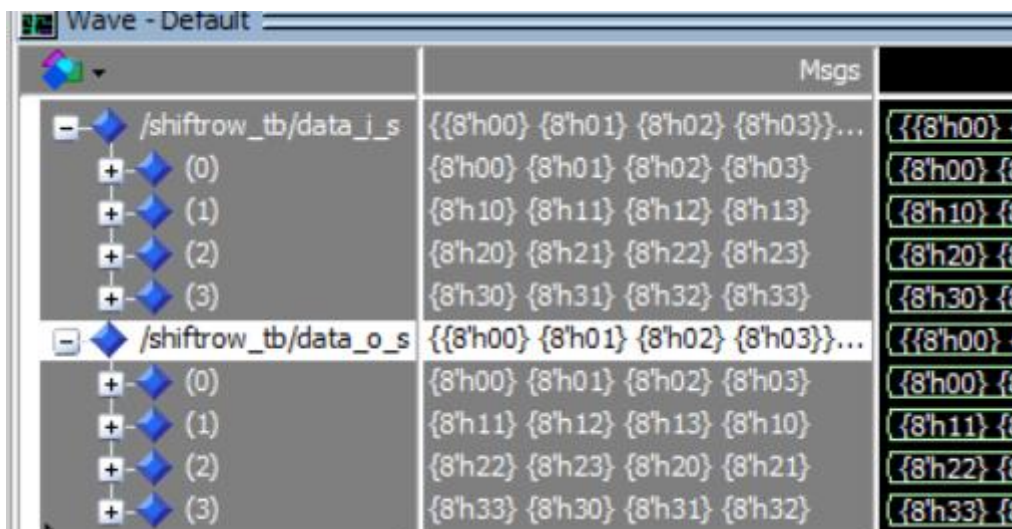
L'entité de ce composant se résume à :

```
entity shiftRow is
  port(
    data_i : in type_state;
    data_o : out type_state
  );
end entity shiftRow;
```

L'architecture de ce composant se résume à l'échange de deux données dans un tableau. Pour cela nous avons utilisé un process afin de séquentialiser l'opération.

```
architecture shiftRow_arch of shiftRow is
begin
  l1 : for i in 0 to 3 generate
    l2 : for j in 0 to 3 generate
      data_o(i)(j) <= data_i(i)((i + j ) mod 4);
    end generate;
  end generate;
end;
```

Test Unitaire ShiftRows :



	Msgs
/shiftrow_tb/data_i_s	{{8'h00} {8'h01} {8'h02} {8'h03}}...
(0)	{8'h00} {8'h01} {8'h02} {8'h03}
(1)	{8'h10} {8'h11} {8'h12} {8'h13}
(2)	{8'h20} {8'h21} {8'h22} {8'h23}
(3)	{8'h30} {8'h31} {8'h32} {8'h33}
/shiftrow_tb/data_o_s	{{8'h00} {8'h01} {8'h02} {8'h03}}...
(0)	{8'h00} {8'h01} {8'h02} {8'h03}
(1)	{8'h11} {8'h12} {8'h13} {8'h10}
(2)	{8'h22} {8'h23} {8'h20} {8'h21}
(3)	{8'h33} {8'h30} {8'h31} {8'h32}

On a mis en entrée data_i_s, on remarque que la sortie est bien l'entrée dont les octets ont subi une permutation circulaire d'ordre égale à son numéro de ligne.

Le composant ShiftRow fonctionne bien.

Composant MixColumns :

Comme les deux autres composants, celui-ci permet d'appliquer une transformation à la matrice de l'état. Pour cela il agit colonne après colonne, et effectue un produit matriciel dans l'algèbre de Galois avec une matrice donnée.

Ce composant prend donc en entrée l'état courant, et l'état courant modifié en sortie. Une entrée est rajoutée afin de confirmer la mise en fonctionnement du composant ou non.

L'entité de ce composant prend la forme suivante :

```
entity mixColumn is
  port(
    data_i : in type_state;
    enableMixColumns_i : in std_logic;
    data_o : out type_state
  );
end entity mixColumn;
```

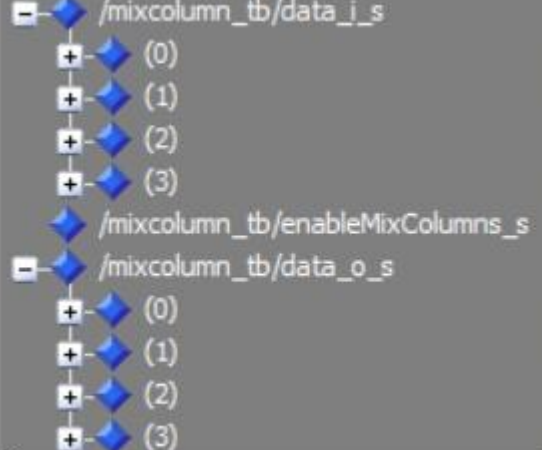
Pour ce qui est de l'architecture, nous avons pu faire en sorte que les produits de chacune des colonnes soient faits concurrenciellement.

Pour cela nous avons explicitement écrit le produit matriciel pour une colonne puis nous l'avons appliqué à chacune des colonnes avec un for ... generate. Nous aurions pu (du ?) créer un composant du même style que la Sbox pour SubBytes.

Pour des raisons de lisibilité le code l'architecture n'est pas dans le rapport. Il est consultable dans le fichier MixColumn.vhd

Remarque : Comme vous pourrez le voir, le code est assez lourd et peu lisible. On peut drastiquement réduire la taille du code et augmenter la lisibilité de celui-ci en travaillant dans un processus (**MixColumn_nonImplementable.vhd**). Le problème est que certains codes marchant en simulation ne sont tous simplement pas synthétisable. Il m'a été dit que ce code pourrait être un exemple.

Test unitaire MixColumns :

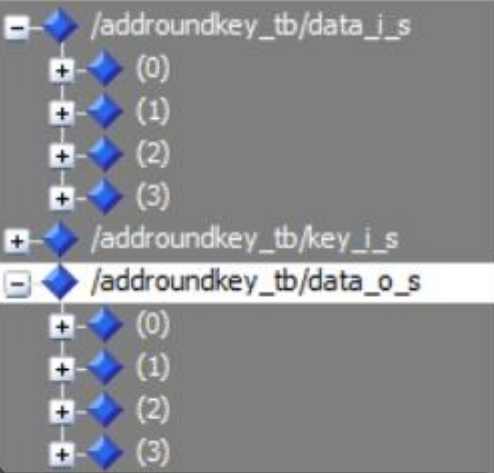
	<pre>{8'hDB} {8'hF2} {8'h01} {8'hC6... {8'hDB} {8'hF2} {8'h01} {8'hC6} {8'h13} {8'h0A} {8'h01} {8'hC6} {8'h53} {8'h22} {8'h01} {8'hC6} {8'h45} {8'h5C} {8'h01} {8'hC6} 1 {8'h8E} {8'h9F} {8'h01} {8'hC6... {8'h8E} {8'h9F} {8'h01} {8'hC6} {8'h4D} {8'hDC} {8'h01} {8'hC6} {8'hA1} {8'h58} {8'h01} {8'hC6} {8'hBC} {8'h9D} {8'h01} {8'hC6}</pre>	<pre>{8'hDB} {8'hF2} {8'h01} {8'hC6... {8'hDB} {8'hF2} {8'h01} {8'hC6} {8'h13} {8'h0A} {8'h01} {8'hC6} {8'h53} {8'h22} {8'h01} {8'hC6} {8'h45} {8'h5C} {8'h01} {8'hC6} 1 {8'h8E} {8'h9F} {8'h01} {8'hC6... {8'h8E} {8'h9F} {8'h01} {8'hC6} {8'h4D} {8'hDC} {8'h01} {8'hC6} {8'hA1} {8'h58} {8'h01} {8'hC6} {8'hBC} {8'h9D} {8'h01} {8'hC6}</pre>
---	--	--

Ce résultat montre que le composant MixColumns est fonctionnel

Composant AddRoundKey :

Ce composant permet d'ajouter la clé de ronde générée par le KeyExpander à la matrice de l'état courant. Composant identique aux composants ShiftRows et SubBytes.

Test Unitaire AddRoundKey :

	<pre>{8'h04} {8'hE0} {8'h48} {8'h28}} {8'h... {8'h04} {8'hE0} {8'h48} {8'h28} {8'h66} {8'hCB} {8'hF8} {8'h06} {8'h81} {8'h19} {8'hD3} {8'h26} {8'hE5} {8'h9A} {8'h7A} {8'h4C} {{8'hA0} {8'h88} {8'h23} {8'h2A}} {{8'h... {{8'hA4} {8'h68} {8'h6B} {8'h02}} {{8'h... {8'hA4} {8'h68} {8'h6B} {8'h02} {8'h9C} {8'h9F} {8'h5B} {8'h6A} {8'h7F} {8'h35} {8'hEA} {8'h50} {8'hF2} {8'h2B} {8'h43} {8'h49}</pre>
---	---

On remarque que le composant AddRoundKey fonctionne bien.

KeyExpander IO :

Ce composant nous servira à générer la clé courante de chaque ronde.

L'algorithme de génération d'une clé d'après la clé primaire est expliqué dans le sujet, et ne sera donc pas repris ici.

Pour faciliter l'écriture du KeyExpander_IO, nous avons décidé de créer quelques composants :

Compteur : Ce composant permet de compter. Il renvoie un entier sur 4 bits s'incrémentant tous les fronts montant de l'horloge.

```
entity Compteur is
    port(
        clock_i : in std_logic;
        reset_i : in std_logic;
        enable_i : in std_logic;
        init_i : in std_logic;
        counter_o : out bit4
    );
end entity;
architecture Compteur_arch of Compteur is
    signal count_s : integer range 0 to 15 := 0;
    begin
        process(clock_i, reset_i)
            begin
                if reset_i = '1' then count_s <= 0;
                else
                    if clock_i'event and clock_i = '1' then --test d'un front montant

                        if enable_i = '1' then
                            if init_i = '1' then count_s <= 0;
                            else count_s <= count_s + 1;
                            end if;
                        else
                            count_s <= count_s;
                        end if;
                    end if;
                end if; --pas besoin de else dans un process de test d'un front
montant
            end process;
            counter_o <= std_logic_vector(to_unsigned(count_s, 4));
        end architecture;
```

Test unitaire du composant Compteur :

/compteur_tb/dock	1				
/compteur_tb/reset	0				
/compteur_tb/counter	4'b0011	4b0000	4b0001	4b0010	
/compteur_tb/init	0				

KeyExpander : Ce composant prend en paramètre une clé et octet et renvoie la clé générée selon l'algorithme proposé dans le sujet.

Pour des raisons de lisibilité le code l'architecture n'est pas dans le rapport. Il est consultable dans le fichier KeyExpansion.vhd

Test Unitaire KeyExpander :

/keyexpansion_tb/key_i_s	128'h2B7E151628AED2A6ABF7158809CF4F3C
/keyexpansion_tb/recon_i_s	8'h01
/keyexpansion_tb/expansion_key_o_s	128'hA0FAFE1788542CB123A339392A6C7605

KeyExpander_IO :

L'entité de ce composant est la suivante :

```
entity KeyExpansion_IO is
  port(
    key_i : in bit128;
    expansion_key_o : out bit128;
    clock_i : in std_logic;
    reset_i : in std_logic;
    start_i : in std_logic;
    round_i : in bit4;
    end_o : out std_logic
  );
end entity;
```

Ce composant va générer les 10 clés de rondes, les garder en mémoire et renvoyer la clé relative à la ronde demandée via l'entrée round_i.

Le Tout en 1 coup d'horloge :

Il est possible de demander la génération des 10 clés en un unique coup d'horloge. Il suffit de brancher 10 composants KeyExpander les uns à la suite des autres et garder en mémoire la sortie de chacun de ces composants. On relie ensuite ces sorties à un multiplexeur sélectionné par round_i. Ce qui donne le code suivant :

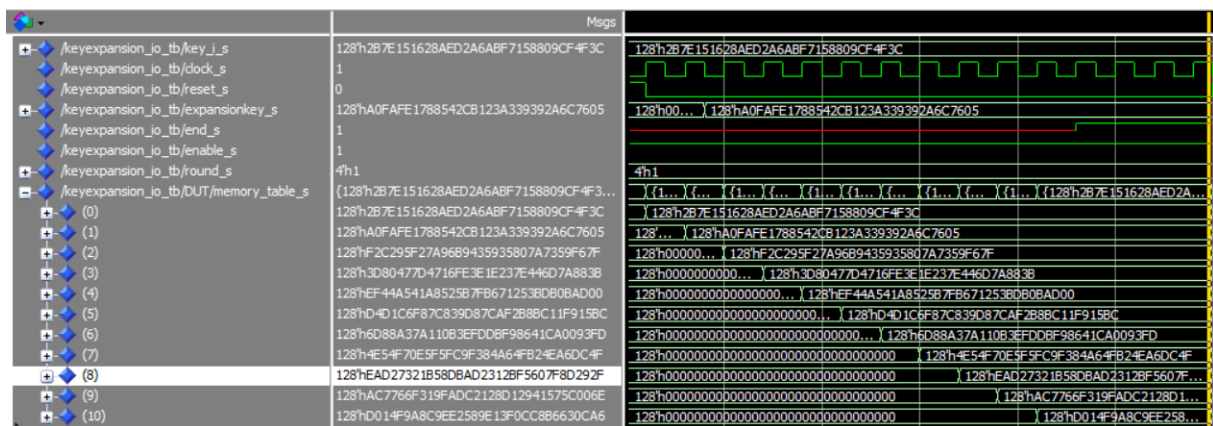
```
architecture KeyExpansion_IO_arch of KeyExpansion_IO is
    component KeyExpansion
        port(
            key_i : in bit128;
            recon_i : in bit8;
            expansion_key_o : out bit128
        );
    end component;
    signal memory : key_memory := (others => X"0000000000000000000000000000");
    begin
        memory(0) <= key;
        g1 : for i in 1 to 10 generate
            S0 : KeyExpansion
                port map(
                    key_i => memory(i-1),
                    recon_i => rcon(i-1),
                    expansion_key_o => memory(i)
                );
        end generate g1;
        expansionkey_o <= memory(conv_integer(unsigned(round_i)));
    end architecture;
```

Le problème c'est qu'on a besoin de 10 KeyExpander. La surface de silicium à utiliser est donc assez grande.

Une autre idée serait de garder qu'un KeyExpander et juste modifier ses entrées à chaque coup d'horloge afin qu'il génère une clé par coup d'horloge.

Pour des raisons de lisibilité le code l'architecture relative à cette méthode n'est pas dans le rapport. Il est consultable dans le fichier KeyExpansion_IO.vhd

Test unitaire KeyExpansion_IO :



On remarque que notre KeyExpander_IO prend 10 coups d'horloge pour générer les 10 clés. Ces clés sont bien générées à chaque coup d'horloge. La diffusion de la clé e ronde se fait en un coup d'horloge.

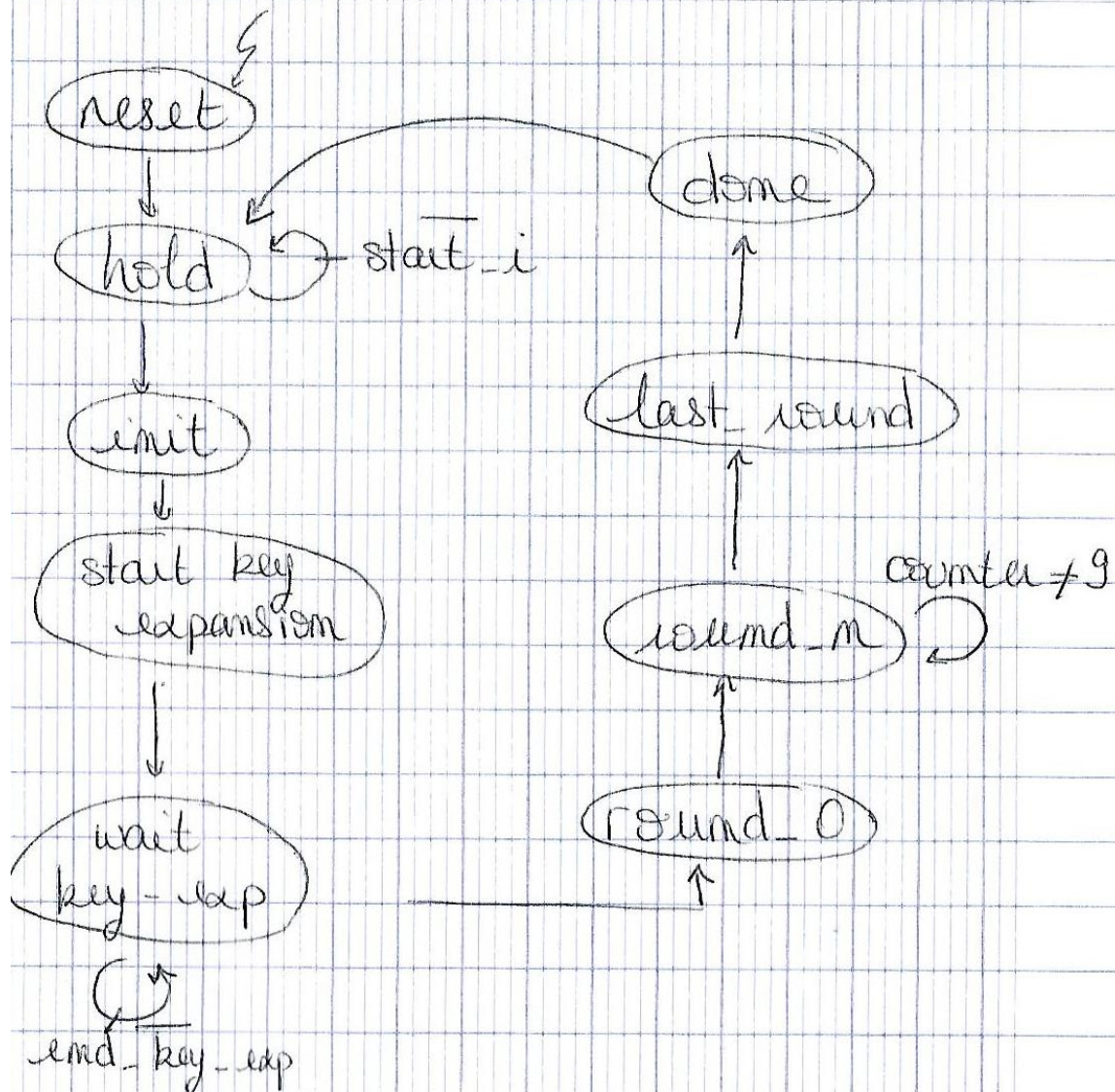
FSM AES :

Ce composant modélise la machine d'état gérant l'AES. Il permet d'activer ou non certains composants suivant un ordre spécifique. Le diagramme de cette machine est le suivant.

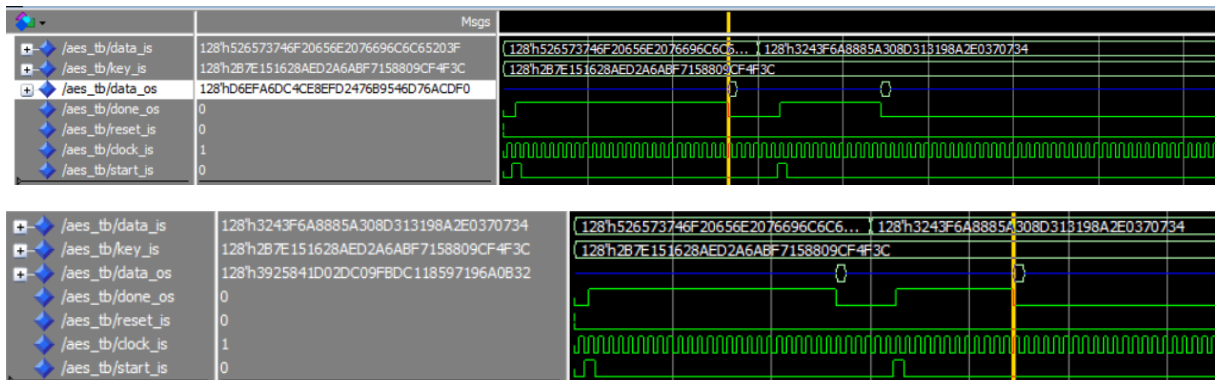
La gestion de passage d'un état à l'autre est gérée par le processus C0.

Pour ce qui est des opérations à appliquer à chacun des états, elles sont gérées par le processus C1.

C'est avec cette FSM qu'on se rend compte de l'utilité des bits enable_i, start_i etc... intégré à certains des composants.



Test global de l'AES :



On remarque que le chronographe est quasi identique à celui du sujet. Il en est de même pour les valeurs du message crypté. Certains changements de comportement ont été implantés dans le fichier FSM_AES afin de modifier le comportement de certains signaux tels que les reset ou les enable.

Notre AES est néanmoins fonctionnel. Il ne nous reste plus qu'à le synthétiser en porte logique.

