École Nationale Supérieure de Physique, Électronique et Matériaux

---

Compte Rendu Mini-Projet AES128

DEVILLARD Hugo - VOITZWINKLER Tom

5 mai 2024

Table des matières

1 Introduction

Le chiffrement est un élément crucial de la sécurité des données dans de nombreux domaines. L'algorithme de chiffrement avancé AES (Advanced Encryption Standard) est devenu l'un des algorithmes les plus utilisés pour assurer la confidentialité et la sécurité des données. Dans ce rapport, nous présentons notre compte rendu de mini-projet portant sur l'implémentation matérielle de l'algorithme de chiffrement AES128.

L'objectif de ce projet était de concevoir et de simuler une architecture matérielle efficace et optimisée pour le chiffrement et le déchiffrement de données selon la norme AES128. Nous avons choisi de développer cette architecture en utilisant le langage de description matériel SystemVerilog et nous avons réalisé plusieurs étapes allant de la spécification de l'algorithme à la simulation post-layout.

2 Architecture

Pour notre architecture nous avons choisi d'utiliser le standard AES128 défini dans [3]. L'algorithme de chiffrement se compose de deux parties principales :

- La génération des clés successives (RoundKey) à partir de la clé initiale : KeyExpansion
- Les différentes étapes du processus de chiffrement : SubBytes, ShiftRows, MixColumns et AddRoundKey. Le standard prévoit de faire 10 tours de boucle (RoundBlock) dans le cas d'AES128.

Le message et la clé sont représentés sous la forme d'une matrice 4x4 avec 16 coefficients qui représentent chacun 8 bits.

Détail de chaque bloc :

- SubBytes : Ce bloc permet de substituer des groupes de 8 bits selon la matrice définie dans le standard.

  Nous fournissons en annexe la table utilisée pour la substitution.

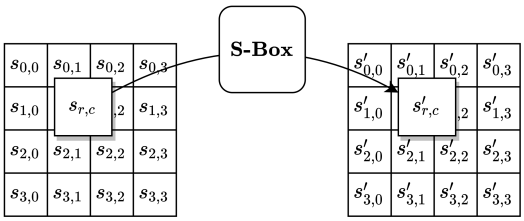

FIGURE 1 – Illustration de SubBytes

- ShiftRows : Permet d'effectuer une permutation circulaire sur une ligne de la matrice.

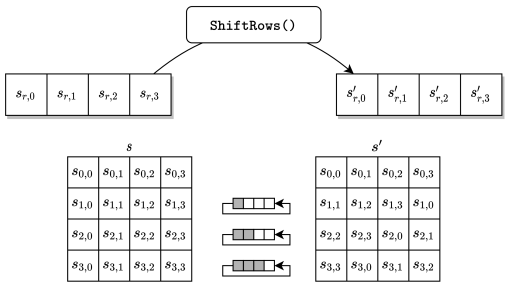

FIGURE 2 – Illustration de ShiftRows

La première ligne ne subit pas de permutation, la deuxième subit une seule permutation, la troisième deux et la dernière trois.

- MixColumns : Chaque colonne de la matrice est multipliée par une autre matrice qui contient les coefficients d'un polynôme dans le domaine de Galois.

  Il est intéressant de noter ici que l'on effectue des multiplications modulaires ainsi lorsque l'on multiplie un coefficient par 2, cela revient à faire l'opération suivante : Soit x un coefficient de la matrice initiale.

  - Si le bit de poids fort est 1 alors : le résultat de la multiplication par 2 est ((x « 1) XOR 00011011)
  - Sinon le résultat est x « 1

  De plus, multiplier par 3 revient à multiplier par 2 et réaliser un XOR avec la valeur initiale.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \leq c < 4,$$

FIGURE 3 – Multiplication d'une colonne par le polynôme de Galois correspondant

- AddRoundKey : consiste en un XOR entre la matrice en cours et la RoundKey (addition dans le domaine de Galois)

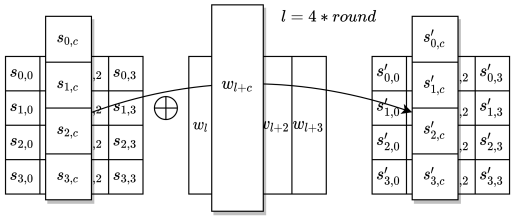

FIGURE 4 – XOR entre la matrice et la RoundKey

- KeyExpansion : permet de générer les 10 clés successives nécessaires à chaque tour dans la boucle de chiffrement. L'algorithme est fourni en annexe. Le terme Rcon dans l'algorithme correspond à une constante fixée pour chaque round.

2.1 Première version : chiffrement

Dans cette première version nous nous sommes concentrés sur la partie chiffrement. Nous obtenons alors l'architecture suivante :

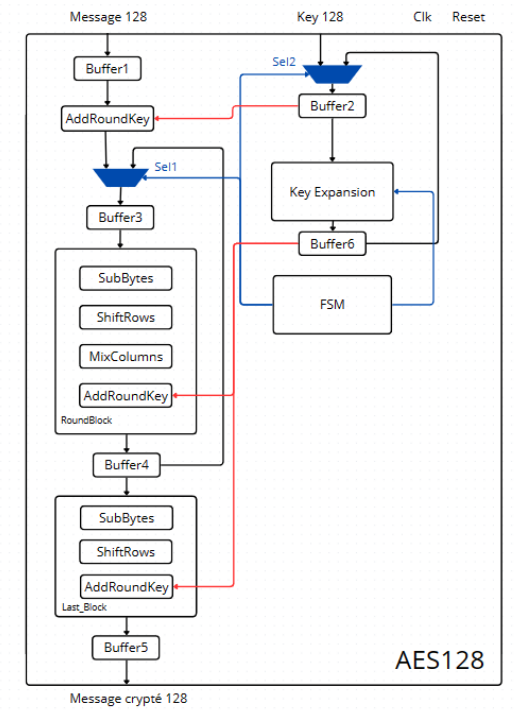

FIGURE 5 – Architecture choisie pour le chiffrement

Dans ce modèle, nous avons réalisé une approche *pipeline* en séparant par des registres les différents grands blocs qui composent notre module de chiffrement. De plus, il est à noter que la génération des *RoundKey* se fait au cours du chiffrement. Ainsi ces clés ne sont pas stockées, ce qui permet de diminuer le nombre de registres.

2.2 Deuxième version : Chiffrement et déchiffrement

Nous avons dans un second temps décidé d'ajouter un module de déchiffrement. Cependant il est n'est plus possible de générer en cours de déchiffrement les *RoundKey* puisque cela signifierait d'effectuer 10! générations de *RoundKey*. En effet, nous avons besoin pour le déchiffrement d'avoir accès à la *RoundKey* numéro 10 en première clé. Cette *RoundKey* ce génère à partir de la 9^{me} *RoundKey* qui elle-même est générée à partir de la 8... La complexité augmente rapidement. C'est pourquoi nous avons choisi de stocker les 11 clés (10 *RoundKey* + 1 *Key*) qui permettent le chiffrement et le déchiffrement du message. Le fait de stocker ces clés augmente toutefois le nombre de registres. Nous obtenons alors l'architecture suivante :

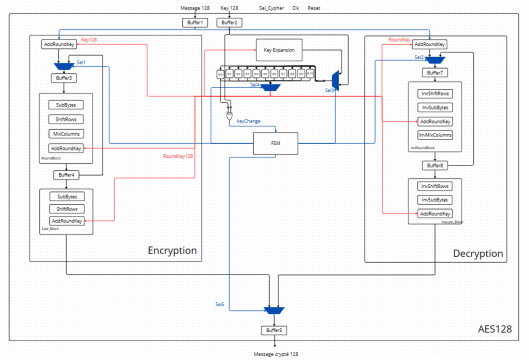

FIGURE 6 – Architecture choisie qui permet le chiffrement et le déchiffrement

Nous avons de plus ajouté une comparaison entre la clé d'entrée et celle stockée dans le registre nommé *B0*. Cette comparaison permet de chiffrer/déchiffrer plusieurs messages de suite avec la même clé sans avoir à régénérer toutes les *RoundKey*.

2.3 Troisième version : Module complet avec interface de communication

Comme notre système comportait un grand nombre de pins (128 pour le message, 128 pour la clé, 128 pour le message chiffré et quelques pins de contrôle), il était nécessaire de créer une interface de communication. Nous nous sommes alors inspirés du bus APB [1]. Nous proposons alors une interface de communication permettant d'envoyer et de recevoir les informations par salves de 32 bits à l'aide de quelques signaux de contrôle :

- Initiate : permet de lancer la communication en passant à l'un des états de lecture/écriture.
- Data : Bus de données.

- R/W : Permet de choisir si on veut lire ou écrire les données sur les registres du module.
- Address : Permet de choisir si on souhaite écrire le message ou la clé (0 : message, 1 : clé)
- SelCypher : Permet de choisir si on veut chiffrer ou déchiffrer le message envoyé. (1 : chiffrement, 0 : déchiffrement)
- Start : Permet de lancer le calcul. Nous n'avons pas lancé le calcul immédiatement après réception des données pour simplifier notre FSM d'interface.
- CS : Un signal permet de déconnecter le registre de sortie (en état haute impédance) tant que la fsm d'interface n'est pas dans l'état d'envoi du résultat sur le bus.

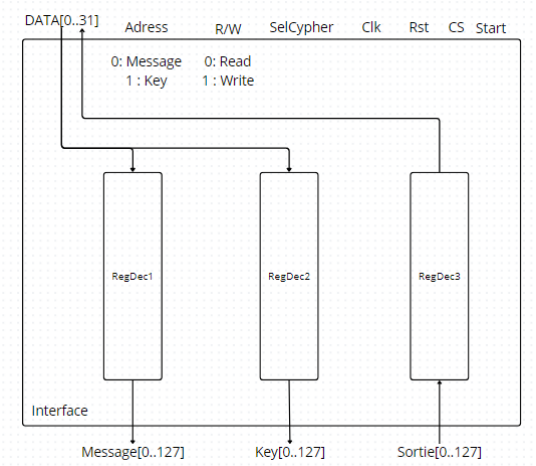

FIGURE 7 – Interface de communication entre le bus et le module AES

En réalité, dans le coeur de notre circuit, nous avons utilisé deux bus de données data_in et data_out : un pour l'entrée et l'autre pour la sortie ainsi qu'un signal CS qui est relié à chaque plot correspondant au bus. En effet, cela nous permet d'utiliser un plot de type BBT16P qui utilise une porte à trois états pour se déconnecter du bus :

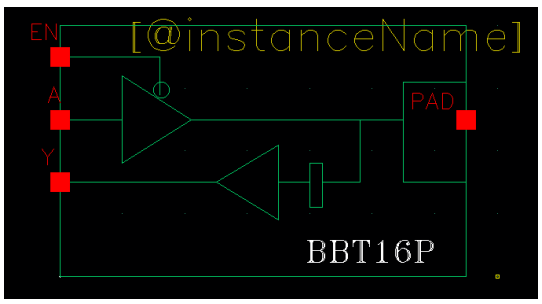

FIGURE 8 – Vue schematic du plot BBT16P

Le schéma complet de l'architecture réalisée est présent en annexe.

Avec cette architecture, nous nous attendons donc à retrouver :

- Les registres de stockage des *RoundKey* : $11 \times 128 = 1408$ registres
- Les registres de l'interface : $3 \times 128 = 384$ registres
- Les registres présents entre les étapes de l'algorithme : $7 \times 128 = 896$ registres
- Les registres d'états des deux fsm : 7 registres.

Au total, notre architecture compte donc : 2695 registres.

3 RTL

Nous avons codé notre module de chiffrement en *System Verilog*. L'ensemble de notre code ainsi que les *testbench* est disponible sur le répertoire Github Cliquer ICI. Nous fournissons également les fichiers .do permettant d'avoir un affichage plus lisible des signaux sur ModelSim. Il est à noter que nous avons vérifié toutes les erreurs de syntaxe ainsi que les potentiels latch inference à l'aide du module LINT de *Spyglass*. Le répertoire git présente plusieurs branches avec le code pour la simulation fonctionnelle, le code après synthèse et le code après placement-routage.

4 Simulation fonctionnelle

On réalise ensuite, par l'intermédiaire de *ModelSim*, des simulations de chaque sous-bloc de notre module ainsi que de notre module entier. Là encore les testbenchs sont présents sur le répertoire Github Cliquer ICI.

La complexité de l'algorithme fait que si l'un des sous blocs crée un bit erroné le message crypté final aura presque 50% de bits erroné. Un test validé signifie qu'il est peu probable que notre module fonctionne par chance. De plus de nombreux logiciels qui permettent de chiffrer/déchiffrer un message sont disponibles en ligne, ce qui nous a permis de tester aisément le bon fonctionnement de notre module.

Prenons par exemple la simulation de notre bloc au complet et testons de chiffrer puis de déchiffrer le message :

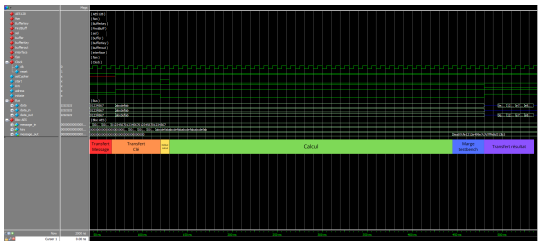

FIGURE 9 – Chiffrement d'un message

Le résultat est bien celui qui est attendu, comme le montre le calculateur utilisant l'algorithme que l'on cherche à implémenter :

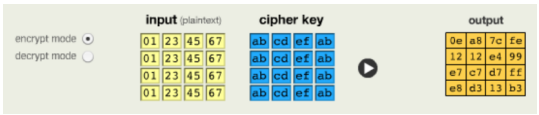

FIGURE 10 – Résultat du chiffrement

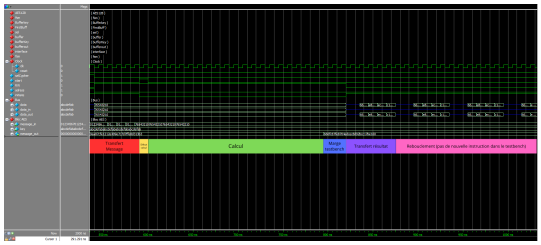

FIGURE 11 – Déchiffrement d'un message

FIGURE 12 – Résultat du déchiffrement

On note qu'il n'a pas été nécessaire de retransmettre la clé de chiffrement entre les deux étapes. Cette fonctionnalité nous a paru pertinente du fait que AES128 est un chiffrement symétrique donc la clé est la même pour le chiffrement et le déchiffrement. On peut ainsi éviter le chemin critique imposé par la génération des *RoundKey* et ainsi gagner du temps sur la phase de calcul.

5 Synthèse logique

On cherche à présent à traduire la description fonctionnelle du circuit en une représentation logique qui sera implémentée sous forme de portes logiques et de bascules génériques. Pour cela, nous utilisons *DesignVision*. L'objectif de cette partie est dans un premier temps, la vérification de la cohérence de notre module *i.e.* s'assurer que la conception répond bien aux spécifications fonctionnelles du circuit. Puis nous nous intéresserons à l'étude du chemin critique, la détermination d'une fréquence d'horloge valide, l'estimation de la consommation et de la surface. De plus cette synthèse nous permet de vérifier une seconde fois la bonne réalisation de notre module en étudiant le nombre de registres alloués. Comme il n'y avait pas de contraintes sur les performances du circuit final, nous choisissons une approche qui n'optimise aucun paramètre en particulier.

5.1 Étude du chemin critique

En testant les limites de notre module on remarque que pour une fréquence d'horloge idéale de 8 ns on obtient un bon fonctionnement du circuit si la réponse de chaque bloc est inférieur à 7.82 ns, ce qui est tout juste le cas :

```
clock clk_8 (rise edge)                                  8.00       8.00
clock network delay (ideal)                              0.00       8.00
AES128_1/B0/buff_out_reg[2]/C (DFEC1)                    0.00       8.00 r
library setup time                                      -0.18       7.82
data required time                                                  7.82
--------------------------------------------------------------------------
data required time                                                  7.82
data arrival time                                                  -7.82
--------------------------------------------------------------------------
slack (MET)                                                         0.00
```

FIGURE 13 – Rapport du chemin critique par Design Vision pour $T_{CLK} = 8ns$

Le logiciel de synthèse permet également d'obtenir une répartition des chemins prenant le plus de temps dans le circuit. Pour cela, il additionne le temps propre à chaque porte logique pour chaque chemin possible et en déduit ainsi le chemin critique.

Puisque nous n'avions pas de cahier des charges nous avons choisi une période d'horloge de 10 ns pour les prochaines études ainsi que pour la génération de la netlist. Dans ce cas nous observons la répartition suivante des chemins critiques :

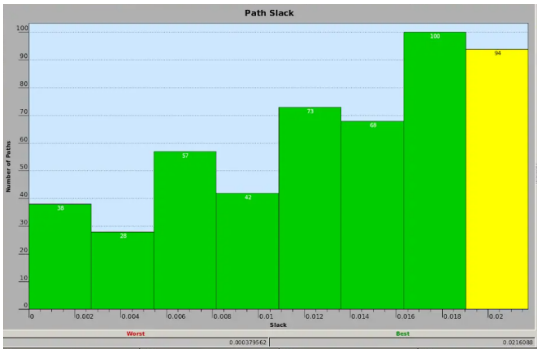

FIGURE 14 – Répartition des différents temps de réponse

Nous remarquons alors que le chemin critique se trouve lors de la génération des *RoundKey*.

Il pourrait être intéressant de comparer le temps lié au passage dans RoundKey et de le comparer avec l'écart obtenu lors de la simulation avec réutilisation de la clé pour voir si notre architecture permet effectivement de gagner du temps en stockant toutes les clés successives.

5.2 Étude de la surface

Pour ce qui est de l'étude de la surface occupée par notre circuit, nous avons utilisé les fonctionnalités de synthèse de Design Vision pour estimer la taille occupée par notre conception sur la puce. Cette surface n'est pas représentative du circuit final puisque la surface estimée par DesignVision ne tient pas compte des contraintes de placement routage et de géométrie mais se contente d'additionner les surfaces de toutes les portes logiques utilisées. À une fréquence d'horloge de 10 ns, la synthèse nous indique que la surface totale occupée par notre circuit est d'environ $4.8mm^2$.

```
Number of ports:                          72
Number of nets:                          456
Number of cells:                           2
Number of combinational cells:             0
Number of sequential cells:                0
Number of macros/black boxes:              0
Number of buf/inv:                         0
Number of references:                      2

Combinational area:            2803710.018639
Buf/Inv area:                   419874.007179
Noncombinational area:          930220.168549
Macro/Black Box area:                0.000000
Net Interconnect area:         1035802.921627

Total cell area:               3733930.187187
Total area:                    4769733.108814
```

FIGURE 15 – Rapport de la surface totale de la synthèse

```
Cell                    Reference       Library            Area  Attributes
--------------------------------------------------------------------------------
AES128_1                AES128                          3573224.191162
                                                                  h, n
interfaceAES_1          interfaceAES                    160705.996025
                                                                  h, n
--------------------------------------------------------------------------------
Total 2 cells                                           3733930.187187
```

FIGURE 16 – Détail de la surface du module complet

On remarque que la surface est avant tout limitée par la présence d'un grand nombre de registres, à l'aide des des outils de DesignVision, nous avons pu vérifier que la synthèse avait bien le même nombre de registres que notre architecture.

5.3 Étude de la consommation

Finalement on s'intéresse à la consommation de notre circuit. *Design Vision* nous permet d'obtenir une estimation de la puissance de notre module. Le logiciel additionne ainsi les consommations statique et dynamique de chaque porte logique mais ne prend pas en compte les éléments ajoutés lors du placement routage comme la consommation de l'arbre d'horloge, des interconnexions, des plots et des fillers entre les cellules. Un extrait du rapport est disponible ci-dessous.

```
Global Operating Voltage = 3.3
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1pW


  Cell Internal Power  = 121.9441 mW   (81%)
  Net Switching Power  =  28.9658 mW   (19%)
                         ---------
Total Dynamic Power    = 150.9099 mW  (100%)

Cell Leakage Power     =  27.6094 nW


                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000              0.0000             0.0000  (   0.00%)
memory             0.0000           0.0000              0.0000             0.0000  (   0.00%)
black_box          0.0000           0.0000              0.0000             0.0000  (   0.00%)
clock_network      0.0000           0.0000              0.0000             0.0000  (   0.00%)
reglster          90.7826           0.9614          6.7229e+03            91.7441  (  60.79%)
sequential        15.4468           0.1643          1.1196e+03            15.6111  (  10.34%)
combinational     15.7146          27.8400          1.9767e+04            43.5547  (  28.86%)
--------------------------------------------------------------------------------------------------
Total            121.9440 mW       28.9658 mW       2.7609e+04 pW        150.9099 mW
```

FIGURE 17 – Estimation de la consommation

A nouveau nous n'avions pas de cahier des charges, cette étude nous permet donc uniquement d'obtenir une idée de la consommation. Nous n'avons pas cherché à optimiser ce paramètre. On peut toutefois noter que la majeur partie de la consommation est liée aux registres utilisés : si on veut réduire la puissance consommée, il faut ainsi réduire le nombre de registres.

A la fin de la synthèse logique nous disposons d'une netlist verilog contenant les portes logiques génériques utilisées pour notre circuit ainsi qu'une estimation du délai dans chaque chemin réalisé par STA (static timing analysis) dans un fichier sdf.

6 Simulation post-synthèse

Après avoir effectué la synthèse logique, qui consiste à traduire la description fonctionnelle du circuit en une représentation logique, la simulation post-synthèse permet de vérifier le comportement fonctionnel et chronologique du circuit synthétisé.

Le *testbench* nommé : **Puce_sans_plot_tb.sv** permet la vérification du bon fonctionnement de notre module de chiffrement après la synthèse logique.

Nous illustrons ici deux exemples prenant en compte le fichier SDF (si on zoomait sur la figure, on pourrait observer les décalages temporels induits par les retards dans les portes logiques) :

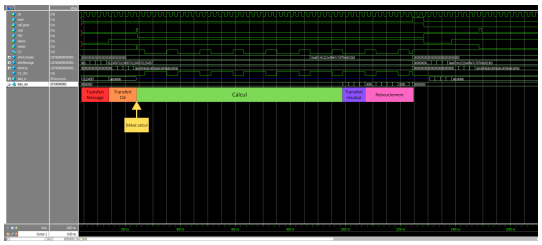

FIGURE 18 – Simulation d'un chiffrement post-Synthèse avec prise en compte du fichier SDF

FIGURE 19 – Déchiffrement post-Synthèse avec fichier SDF

Nous avons vérifié dans les deux cas la valeur retournée par notre module à l'aide du calculateur cité en bibliographie. La phase de rebouclement est simplement due à notre testbench qui permet de renvoyer le plus tôt possible la valeur calculée.

7 Placement-routage

Nous avons ensuite procédé à l'étape de Placement-Routage de notre circuit en utilisant l'outil Innovus. Cette étape consiste à mapper la conception logique du circuit sur la surface physique de la puce, en assignant des positions physiques à chaque élément du circuit. C'est ici que l'on définit les contraintes physiques de notre circuit :

- Position des plots
- Ring et Stripes pour l'alimentation
- Définition de l'arbre d'horloge
- Choix de la taille du coeur

Pour simplifier la conception, nous avons réalisé les étapes à l'aide de l'interface graphique puis nous avons copié les commandes qui nous convenaient dans le script init.tcl d'Innovus.

Comme indiqué dans le TP Filtre, il faut prendre des pistes de l'ordre de 1 μm par mA pour les rings d'alimentation : or d'après le rapport DesignVision, notre circuit consomme presque 50 mA, c'est pourquoi nous avons choisi des pistes de 50 μm de large.

Nous avons choisi de mettre 5 stripes pour éviter les zones de chutes de tension dues aux pertes linéiques.

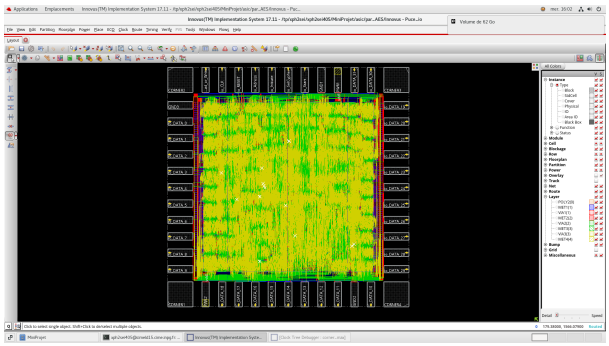

FIGURE 20 – Layout final après placement-routage

Il reste quelques erreurs liées aux contraintes du fondeur mais qui pourraient être résolues.

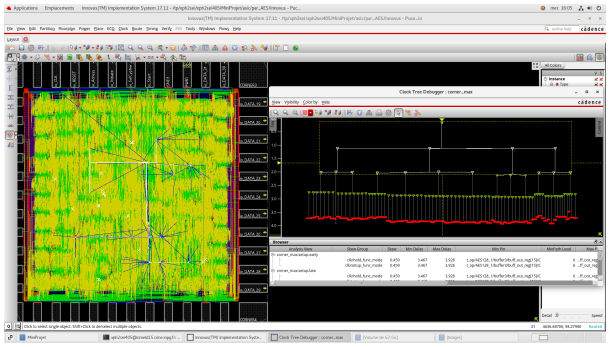

FIGURE 21 – Emplacement de l'arbre d'horloge

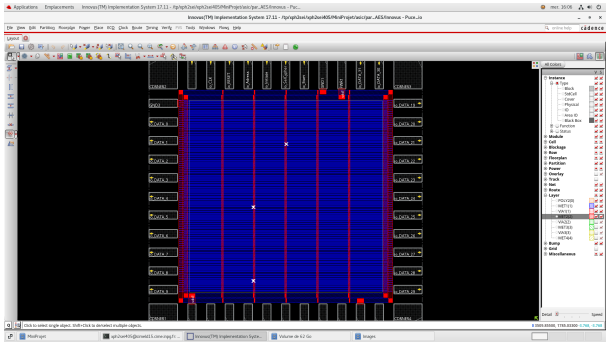

FIGURE 22 – Rings et Stripes du design

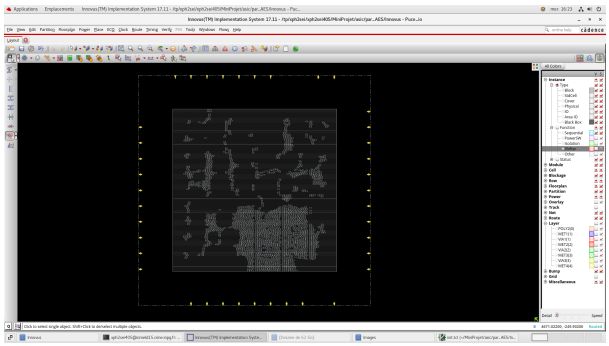

FIGURE 23 – Espace occupé par les registres dans le coeur

Une grande partie de la surface est occupée par les registres comme ce qui avait été prévu par DesignVision. On peut notre que le logiciel de placement-routage a placé tous les registres qui concernent le traitement des données dans la même région même si ceux-ci sont dans des unités différentes dans le code Verilog initial.

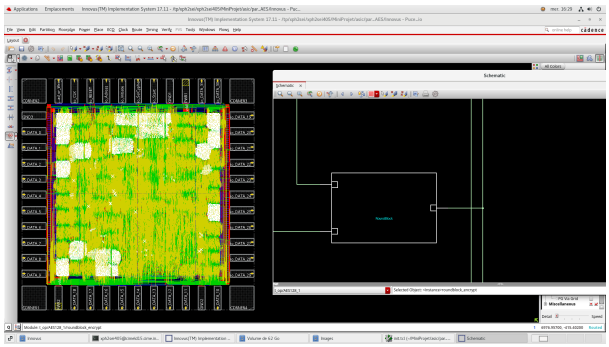

FIGURE 24 – Espace occupé par le bloc RoundBlock

La logique combinatoire représente au final une partie très limitée du design.

Floorplan/Placement Information

Total area of Standard cells	5334638.400 um^2
Total area of Standard cells(Subtracting Physical Cells)	3566344.600 um^2
Total area of Macros	0.000 um^2
Total area of Blockages	0.000 um^2
Total area of Pad cells	1961248.640 um^2
Total area of Core	5334638.400 um^2
Total area of Chip	10703289.120 um^2

FIGURE 25 – Surface design après placement-routage

On obtient une surface occupée par les cellules beaucoup plus grande que lors de la synthèse, en effet, les contraintes de placement routage ajoutent une surface supplémentaire liée aux fillers et à l'arbre d'horloge par exemple.

Area of Power Net Distribution

Layer Name	Area of Power Net	Routable Area	Percentage
MET1	1025151.4000	5334638.4000	19.2169%
MET2	384296.0500	5334638.4000	7.2038%
MET3	0.0000	5334638.4000	0.0000%
MET4	15895.3650	5334638.4000	0.2980%

FIGURE 26 – Répartition des liaisons sur les niveaux de métallisation

La majorité des liaisons se fait sur le niveau MET1, ce qui signifie que le circuit aura moins de pertes liées aux interconnexions. En effet, plus on utilise des niveaux de métallisation élevés, plus les pertes linéiques augmentent.

```
SPACING: Regular Wire of Net t_op/AES128_1/roundblock_decrypt/InvMixColumns1/n91 & Special Wire of Net vdd!  ( MET2 )
Bounds : ( 1629.900, 2285.100 ) ( 1630.400, 2285.700 )
Actual:    0.5    Min:    0.8

SHORT: Regular Via of Net t_op/AES128_1/roundblock_decrypt/InvSubBytes1/genblk1[112].InvS_Box1/n469 & Regular Wire of Net t_op/AES128_1/roundblock_decrypt/InvSubBytes1/genblk1[112].InvS_Box1/n631  ( MET3 )
Bounds : ( 1988.400, 2233.100 ) ( 1989.000, 2233.700 )


Begin Summary ...
  Cells     : 0
  SameNet   : 0
  Wiring    : 1
  Antenna   : 0
  Short     : 1
  Overlap   : 0
End Summary

 Total Violations : 2 Viols.
```

FIGURE 27 – Violations des règles du fondeur

Il reste quelques violations des règles du fondeur qu'il serait possible de régler en relançant le placement-routage avec de nouveaux paramètres ou en modifiant manuellement le layout.

```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

+--------------------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):| -71.026 | -71.026 | 36.740  |
|           TNS (ns):|-42094.2 |-42094.2 |  0.000  |
|    Violating Paths:|  2296   |  2296   |    0    |
|          All Paths:|  7965   |  5199   |  2769   |
+--------------------+---------+---------+---------+

+----------------+-------------------------------+------------------+
|                |             Real              |       Total      |
|    DRVs        +------------------+------------+------------------|
|                |  Nr nets(terms)  | Worst Vio  |  Nr nets(terms)  |
+----------------+------------------+------------+------------------+
|   max_cap      |    750 (750)     |   -4.001   |    750 (750)     |
|   max_tran     |   4692 (36336)   |  -184.780  |   5089 (36733)   |
|   max_fanout   |      0 (0)       |     0      |      0 (0)       |
|   max_length   |      0 (0)       |     0      |      0 (0)       |
+----------------+------------------+------------+------------------+

Density: 67.015%
      (100.000% with Fillers)
Total number of glitch violations: 0
------------------------------------------------------------
```

FIGURE 28 – Rapport de timing après placement-routage

L'analyse temporelle ne trouve pas de violations de timing qui pourraient causer des glitchs dans le circuit.

Notre circuit est core-limited autrement dit, l'espace occupé par le coeur (avec le paramètre de densité de 0.7 choisi) ne permet pas de tenir au sein de la couronne de plots si ceux-ci ne sont pas écartés.

Après cette étape, nous disposons du fichier GDS qui contient le layout du design, d'une netlist Verilog qui contient les portes utilisées dans le layout final ainsi que

d'un nouveau fichier SDF qui prend en compte les délais dans les interconnexions après analyse des parasites RC.

8 Simulation post-layout

Après avoir effectué le placement et le routage nous effectuons une simulation post-layout. Cette dernière permet de vérifier le comportement fonctionnel et chronologique du circuit après qu'il a été physiquement implémenté sur la puce. Nous avons notamment récupérer le fichier Standard Delay Format (sdf) qui contient des informations sur les délais de propagation, les délais d'arrivée, les contraintes de timing, les informations sur les horloges, les charges capacitives des lignes de signal etc...

Le *testbench* nommé : **Puce_io_after_rout_tb** permet la vérification du bon fonctionnement de notre module de chiffrement après le placement routage.

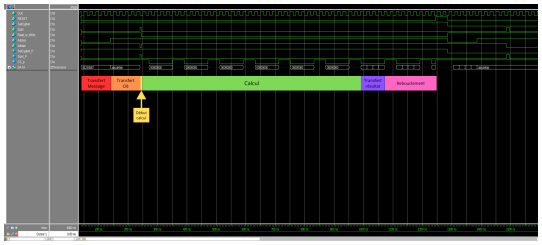

FIGURE 29 – Chiffrement post-Layout avec SDF

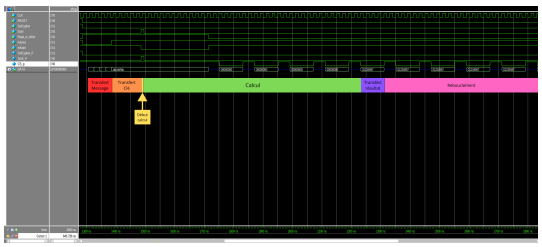

FIGURE 30 – Dechiffrement post-Layout avec SDF

Nous obtenons encore une fois le bon résultat en vérifiant avec le calculteur fourni en bibliographie.

9 Améliorations possibles

Étant limités par le temps, notre projet n'est pas optimisé et encore moins prêt pour une fabrication éventuelle. C'est pourquoi nous analysons dans cette partie une liste de points que nous aurions aimé améliorer.

- **Diminution du nombre de SBox** : nous avons remarqué lors de la synthèse logique que nous utilisions à deux reprises le module *SubBytes*, une fois dans RoundBlock et une fois dans LastBlock. SubBytes génère 16 S_Box. Nous aurions pu optimiser d'avantage notre puce en réutilisant les 16 S_Box de Roundblock dans le module LastRound. Comme notre puce contient le cryptage et le decryptage cela aurait permis d'économiser 32 S_Box. Ce qui représente $2 \times 16 \times 16 \times 8 = 1024$ registres.
- **Diminution du nombre de registres** : Nous pouvons encore réduire le nombre de registres utilisés, par exemple en réunissant les blocs de calcul et d'interface, ce qui permettrait d'économiser les registres d'interface. On pourrait par ailleurs penser à une architecture qui ne stocke pas les roundKey.
- **Sécurité** : Notre projet n'avait pas pour objectif la conception d'une puce sécurisée, ce qui est pourtant un attendu d'une puce cryptographique. En effet, il est très certainement possible de retrouver la clé secrète utilisée pour les calculs que ce soit à l'aide d'une attaque physique ou par canaux auxiliaires.
- **Testbenchs** : Les testbenchs ont été réalisés dans des conditions favorables même si une erreur dans le chiffrement provoque 50 % d'erreur en sortie de l'algorithme selon le standard, il pourrait être intéressant de faire des testbenchs à validation automatique (scripts et non vérification visuelle) pour s'assurer du fonctionnement de notre système.
- **Vérification** : partie du flot que nous n'avons pas réalisé mais qui constituerait une partie importante du temps alloué au projet si on devait la réaliser.
- **Chaîne de test** : Insertion de scan chains et d'autres éléments pour tester notre circuit après fabrication.
- **Modification de la fsm d'interface** : Nous avons fait une fsm d'interface très simplifiée qui mériterait d'être adaptée à un bus de communication standard type APB pour que notre projet soit utilisable en condition réelle. De plus, il serait intéressant de réaliser un meilleur système de lancement des calculs puisque celui-ci se réalise à l'aide d'un reset de la fsm du bloc de calcul provoqué par l'utilisateur.

10 Conclusion

Ce projet nous a permis de mettre en application tous les concepts appris cette année notamment ceux du flot de conception. Le fait d'être en autonomie permet de progresser rapidement sur la prise en main des outils et d'acquérir des connaissances durables dans notre domaine d'activité.

Bibliographie

[1] *AMBA APB Protocol Specification*. ARM. 2010. URL : https://developer.arm.com/documentation/ihi0024/c/.

[2] Lubos GASPAR Viktor FISCHER Florent BERNARD Lilian BOSSUET Pascal COTRET. *A Novel Concept of Crypto-processor with Secured Key Management*. International Conference on Reconfigurable Computing. 2010. URL : https://hal.science/hal-00750348/document.

[3] National Institute of STANDARDS et TECHNOLOGY. *Advanced Encryption Standard*. Federal Information Processing Standards Publication (FIPS). 2023. DOI : 10.6028/NIST.FIPS.197-upd1.

[4] FORMAESTSTUDIO. *Rijndael (AES) Animation*. URL : https://formaestudio.com/portfolio/aes-animation/.

Annexes

Table de conversion pour le bloc SubBytes

		y														
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

FIGURE 31 – Table utilisée pour la substitution dans le bloc SubBytes où xy est l'octet noté en hexadécimal

Tableau des constantes de Round

j	$Rcon[j]$	j	$Rcon[j]$
1	[01,00,00,00]	6	[20,00,00,00]
2	[02,00,00,00]	7	[40,00,00,00]
3	[04,00,00,00]	8	[80,00,00,00]
4	[08,00,00,00]	9	[1b,00,00,00]
5	[10,00,00,00]	10	[36,00,00,00]

FIGURE 32 – Tableau des constantes Rcon

Schéma complet de l'ASIC

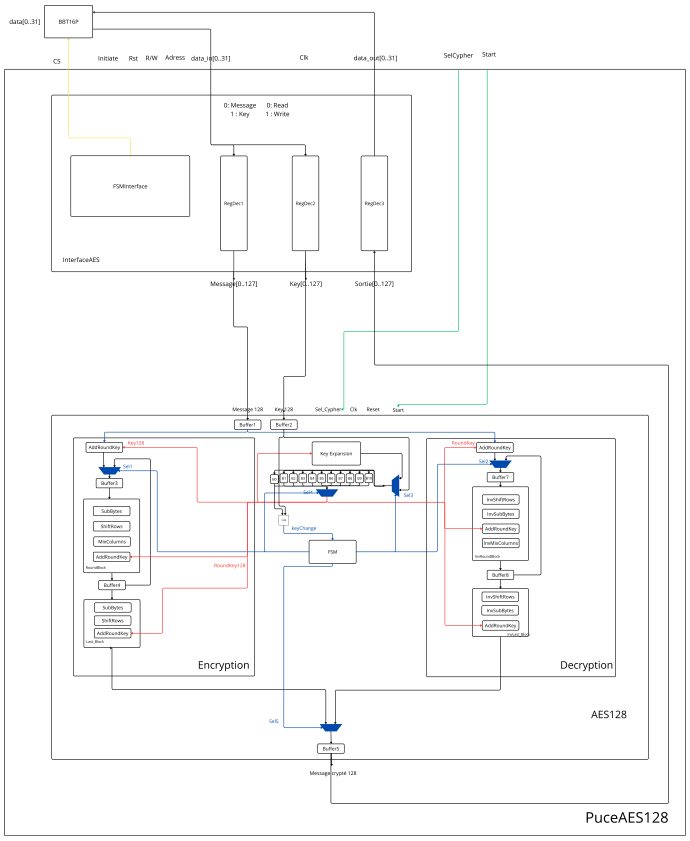

FIGURE 33 – Schéma complet du circuit réalisé

Algorithme de KeyExpansion

Algorithm 2 Pseudocode for KEYEXPANSION()

```
1:  procedure KEYEXPANSION(key)
2:      i ← 0
3:      while i ≤ Nk − 1 do
4:          w[i] ← key[4 ∗ i..4 ∗ i + 3]
5:          i ← i + 1
6:      end while                          ▷ When the loop concludes, i = Nk.
7:      while i ≤ 4 ∗ Nr + 3 do
8:          temp ← w[i − 1]
9:          if i mod Nk = 0 then
10:             temp ← SUBWORD(ROTWORD(temp)) ⊕ Rcon[i/Nk]
11:         else if Nk > 6 and i mod Nk = 4 then
12:             temp ← SUBWORD(temp)
13:         end if
14:         w[i] ← w[i − Nk] ⊕ temp
15:         i ← i + 1
16:     end while
17:     return w
18: end procedure
```

FIGURE 34 – Algorithme de génération des roundKey