Florian Pourraz - Mona Ezzadeen
2A - SEI

Générateur d'IP synthétisables
Famille Simon

Mini-projets

Phelma - 2017/2018

Remerciements

Nous tenons beaucoup à remercier nos professeurs encadrants, M. Leveugle et Mme Morin-Allory, pour leur précieuse aide, leur patience et leur suivi tout au long de ce projet.

Introduction

Dans le cadre des mini-projets de deuxième année de la filière SEI, nous avons choisi de réaliser un générateur de crypto-processeurs synthétisables de la famille Simon. En effet, lors de sa publication officielle sur le chiffrement Simon, la NSA (National Security Agency) a proposé 10 versions différentes de par leurs tailles de message et de clé. Notre but fut de concevoir un générateur de crypto-processeurs Simon pouvant au moins générer ces 10 versions de base, et étendre le nombre de versions générées en proposant des options.

Ce rapport traite du développement de ce générateur, ainsi que de la conception d'un crypto-processeur Simon et de son implantation sur FPGA.

Ce rapport est accompagné d'un manuel d'utilisation et de trois dossiers : le dossier `Generateur` contenant le code du générateur, le dossier `cryptoProc-FPGA` contenant la description VDHL d'un crypto-processeur Simon 64/128 avec son wrapper pouvant être implémenté sur FPGA, et le dossier `Dialogue UART (.py)` contenant des fichiers Python permettant l'envoi et la réception de trames (entre l'ordinateur et la carte FPGA) depuis l'ordinateur.

SOMMAIRE

1 La famille de chiffrement Simon

Cette partie a pour objectif de présenter rapidement la famille de chiffrement Simon, proposée par la NSA en 2013.

I. Le chiffrement Simon dans la cryptographie symétrique

La famille de chiffrement Simon appartient à la catégorie du ***chiffrement par bloc*** (“block cipher” en anglais), qui représente l’une des deux grandes classes de ***cryptographie symétrique***, l’autre étant le *chiffrement par flot* (“stream cipher” en anglais).

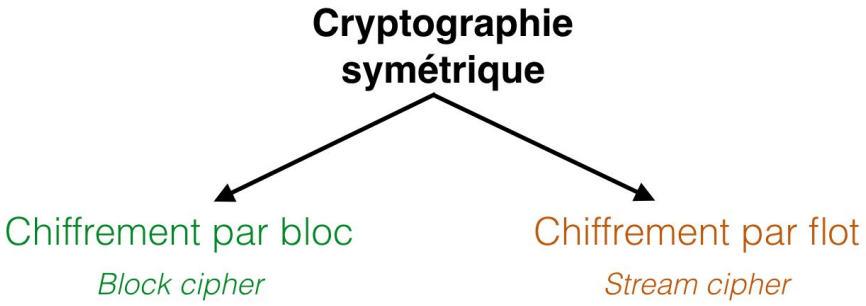


Figure 1 : les grandes catégories de la cryptographie symétrique

La ***cryptographie symétrique*** (ou à clé secrète) est la plus ancienne forme de chiffrement. Elle consiste à disposer d’une clé unique pour chiffrer/déchiffrer des messages. Cette clé est connue de l’émetteur et du récepteur, et sert à la fois à chiffrer et à déchiffrer les messages.
Le schéma ci-dessous illustre ce principe :

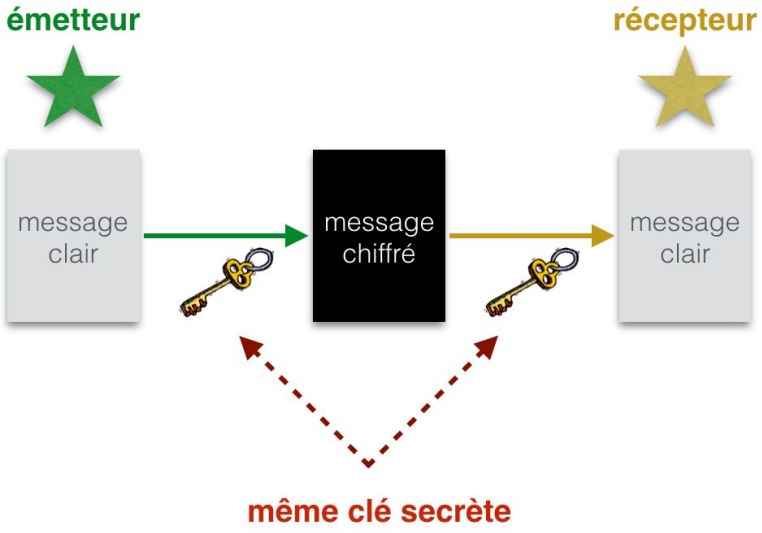


Figure 2 : schéma de principe du chiffrement symétrique

Le ***chiffrement par bloc*** consiste à découper les données en blocs de taille fixe, puis à les chiffrer les uns à la suite des autres. Un bloc après chiffrement aura la même taille qu'avant chiffrement.
Parmis les algorithmes de chiffrement par bloc, on peut citer l'algorithme *DES* (Data Encryption Standard), publié en 1970, et qui fut utilisé pour le système de mots de passe Linux (mais qui est peu utilisé en raison de sa vulnérabilité face à une attaque systématique), l'algorithme *AES* (Advanced Encryption Standard), publié en 2000 et adopté comme standard de chiffrement par les organisations du gouvernement des Etats-Unis, ou encore les algorithmes *Serpent, Blowfish* ou *Twofish*.
La famille Simon utilise le principe du chiffrement par bloc.

Le ***chiffrement par flot*** traite en continu les données bit par bit, et n'a donc pas besoins de les découper. Plus rapide que le chiffrement par bloc, il est très utilisé pour les communications en temps réel comme le WIFI ou le Bluetooth par exemple. De plus, contrairement au chiffrement par bloc, pour un même message, le message chiffré ne sera pas forcément identique à chaque chiffrement, alors qu'il sera le même avec le chiffrement par bloc.
Parmis les algorithmes de chiffrement par flot, on peut citer l'algorithme E0, utilisé dans le protocole Bluetooth, et l'algorithme RC4, conçu en 1987, et qui fut utilisé dans les protocoles WEP, WPA et TLS.

II. Description du chiffrement Simon

La famille de chiffrement Simon (tout comme la famille Speck) fut conçue dans les buts de permettre une implémentation matérielle légère (*lightweight*) tout en permettant une implémentation logicielle sur des supports restrictifs (petits microcontrôleurs, avec peu de mémoire RAM). La famille Simon est particulièrement optimisée pour une implémentation matérielle légère.

Le chiffrement Simon utilise des **messages de taille 2n**, et des **clés de taille mn**, où m et n sont des entiers.
Les différentes versions possibles pour le chiffrements Simon sont données dans le tableau ci-dessous :

Taille du bloc **2n**	Taille de la clé **mn**	**n**	**m**	Constante z_i	Nombre de rondes **T**
32	64	16	4	z_0	32
48	72	24	3	z_0	36
	96		4	z_1	36
64	96	32	3	z_2	42
	128		4	z_3	44
96	96	48	2	z_2	52

	144		3	z_3	54
128	128	64	2	z_2	68
	192		3	z_3	69
	256		4	z_4	72

Les rôles des constantes z_i et du nombre de rondes *T* seront explicités dans les lignes qui suivent.

A. Fonctions de ronde

Basé sur les réseaux de Feistel, le chiffrement Simon consiste à faire passer le message d'origine par plusieurs tours ou étages (*rondes*) identiques de chiffrement.

A chaque *ronde*, la moitié du message (de taille n) est encodée seulement, auquelle on ajoute l'autre moitié via plusieurs *xor* (voir plus loin), avant d'être intervertie avec l'autre moitié pour le ronde suivant. Le nombre de rondes dépend de la version choisie, et est donné par le tableau précédent.

Chaque ronde utilise une clé intermédiaire Ki, appelée *clé de ronde*. Nous aborderons la génération des clés de ronde dans la partie suivante.

Le déchiffrement est identique au chiffrement, à la différence près que les clés de rondes sont utilisés dans l'ordre inverse.

L'étage de chiffrement (*ronde*) pour la famille Simon est le suivant :

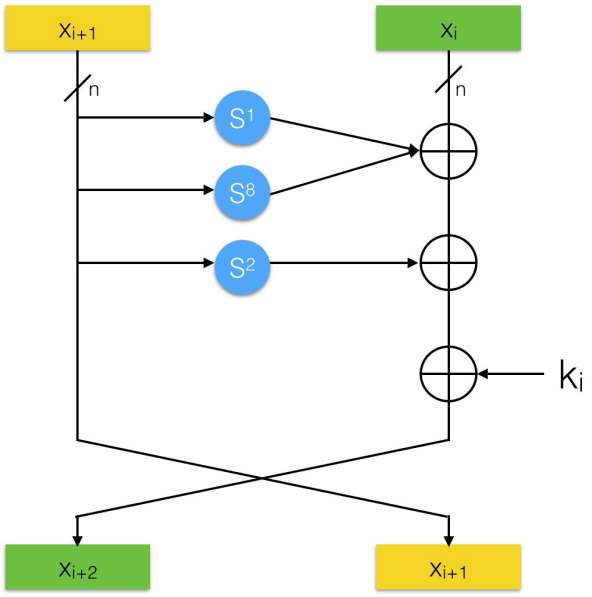


Figure 3 : schéma d'une ronde

Cet étage utilise des permutations circulaires S^j vers la gauche, par paquets de j bits. Par exemple, si l'on a la suite de bit suivante : **011**01, et que l'on réalise une permutation circulaire S^3, alors la suite de bit devient : 01**011**.

Ainsi, le réseau de Feistel complet pour le chiffrement et le déchiffrement Simon a l'allure suivante :

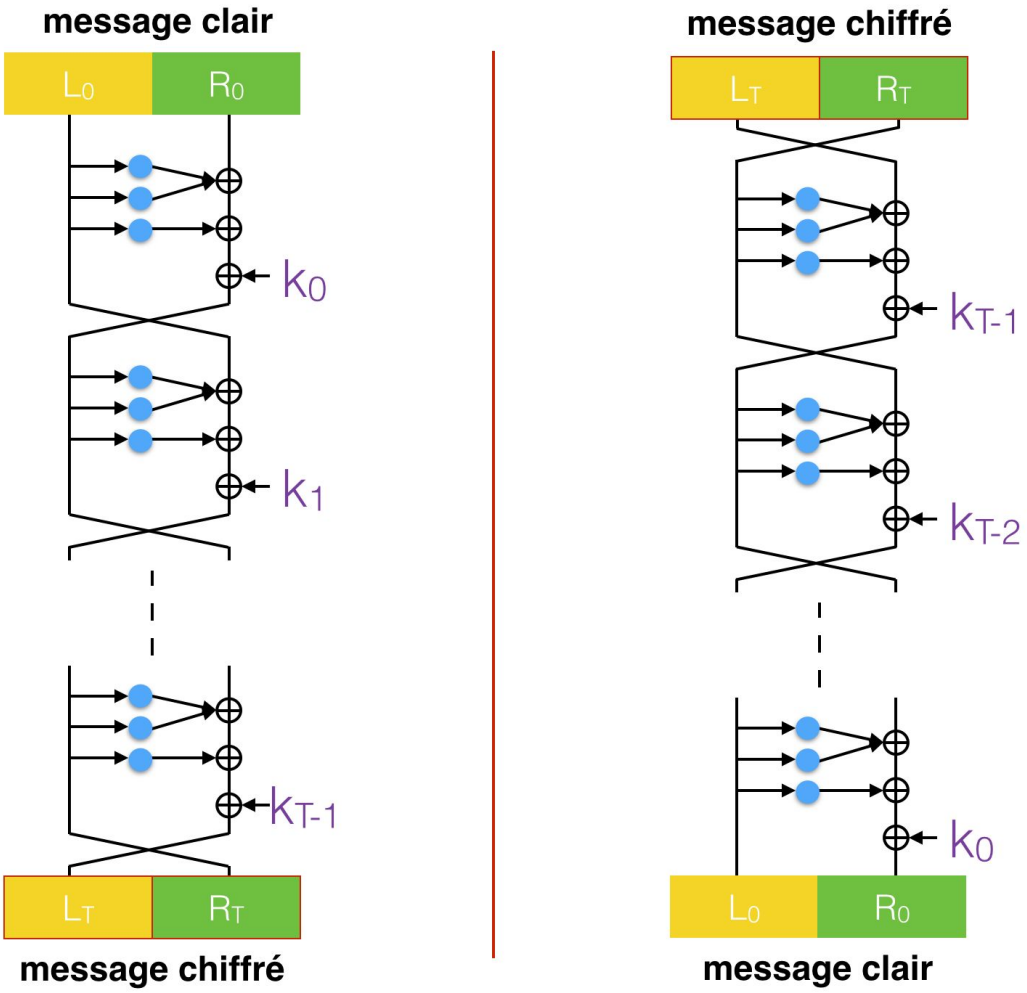


Figure 4 : réseau de Feistel complet pour le chiffrement et le déchiffrement Simon

B. Génération des clés

Deux constantes sont nécessaires pour générer les clés de rondes :

- les z_{ij} sont les bits de la matrice Z suivante :

```
   z =
[11111010001001010110000111001101111101000100101011000011100110

10001110111110010011000010110101000111011111001001100001011010

10101111011100000011010010011000101000100011111100101101100 11

11011011101011000110010111100000010010001010011100110100001111

11010001111001101011011000100000010111000011001010010011101111]
```

- la constante $c = 2^n - 4$

Les m premières clés de rondes k_0 , k_1 ... k_{m-1} sont directement issues du découpage en m paquets de taille n de la clé initiale :

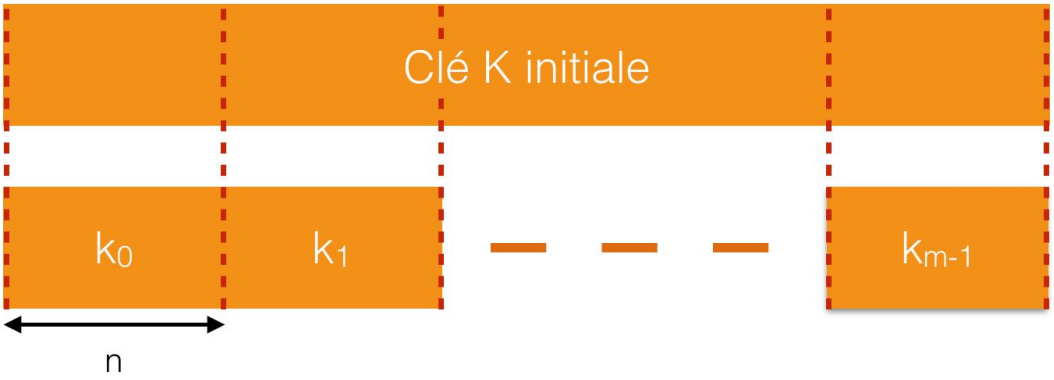


Figure 5 : découpage de la clé initiale pour générer les m premières clé de ronde

La génération de la k_{i+m} ème clé de ronde dépend de la valeur de m et est définie de la manière suivante :

m	k_{i+m}
2	$c \oplus z_{ji} \oplus k_i \oplus (I \oplus S^{-1})S^{-3}k_{i+1}$
3	$c \oplus z_{ji} \oplus k_i \oplus (I \oplus S^{-1})S^{-3}k_{i+2}$
4	$c \oplus z_{ji} \oplus k_i \oplus (I \oplus S^{-1})(S^{-3}k_{i+3} \oplus k_{i+1})$

Schématiquement, la génération des clés, en fonction de la valeur de m, est la suivante :

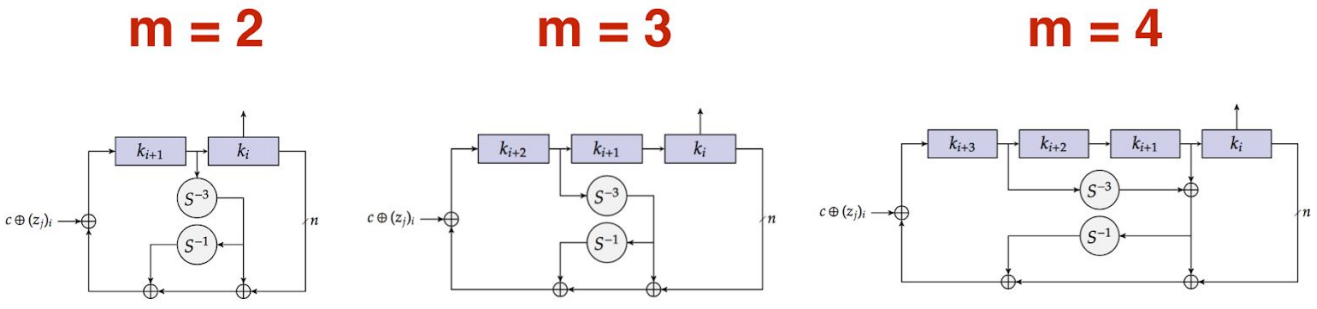


Figure 6 : génération des clés de ronde

(les 3 schémas de cette figure ont été prises du papier officiel de la NSA sur les familles Simon et Speck)

Ces schémas nous seront d'une grande aide pour l'implémentation matérielle de la génération des clés de ronde.

2 Cahier des charges

Avant de commencer le développement du générateur de crypto-processeurs Simon, il est nécessaire de déterminer les options que présentera celui-ci. Ce court chapitre a pour but de justifier nos choix en matières d'options du générateur.

Dans le chapitre précédent, nous avons présenté la famille de chiffrement Simon, et notamment les 10 différentes versions possibles proposées par la NSA, que nous rappelons dans le tableau ci-dessous :

Taille du bloc **2n**	Taille de la clé **mn**	**n**	**m**	Constante z_i	Nombre de rondes **T**
32	64	16	4	z_0	32
48	72	24	3	z_0	36
	96		4	z_1	36
64	96	32	3	z_2	42
	128		4	z_3	44
96	96	48	2	z_2	52
	144		3	z_3	54
128	128	64	2	z_2	68
	192		3	z_3	69
	256		4	z_4	72

Notre générateur doit donc au minimum pouvoir générer ces 10 versions, qui ne diffèrent que par les tailles du message et de la clé.

Cependant, il est possible d'introduire des options pour chacune de ces versions. En effet, selon que l'on souhaite optimiser la surface de l'IP ou sa vitesse de traitement, l'IP sera légèrement (ce point sera discuté en détail au chapitre 4) différente.

Nous avons opté pour les options suivantes :

- chiffrement seul
- déchiffrement seul
- chiffrement et déchiffrement

Et pour chacune de ces options, nous offrons deux possibilités :

- garder la même clé pour plusieurs chiffrements/déchiffrements
- recharger la clé à chaque chiffrement/déchiffrement

Nous disposons donc de 6 options au total. Notre générateur doit ainsi pouvoir générer **60 versions différentes**.

Avant de s'attaquer à la conception du générateur, nous avons choisi de développer une version du crypto-processeur Simon qui soit la plus complète possible, c'est-à-dire qui permette à la fois de chiffrer et de déchiffrer, et qui puisse garder la même clé pour plusieurs chiffrements/déchiffrements. Ce crypto-processeur nous servira de base pour générer les 59 autres versions.

Nous allons nous baser sur la famille 64/128 pour le développer, tout en rendant aisée la transition vers les autres tailles de message (2n) et de clé (mn), comme nous allons le présenter en détail dans le chapitre suivant.

3 Conception d'un crypto-processeur Simon

Avant de développer le générateur de crypto-processeurs Simon, il nous faut concevoir un crypto-processeur de cette famille qui soit le plus complet possible en termes d'options, duquel nous pourrons construire toutes les autres versions. Ce chapitre traite de la conception de ce crypto-processeur.

I. Architecture du crypto-processeur

Comme expliqué au chapitre précédent, ce premier crypto-processeur se veut le plus complet possible, c'est-à-dire qu'il doit pouvoir à la fois chiffrer et déchiffrer, et garder la même clé pour plusieurs chiffrements/déchiffrements.
De plus, son implémentation doit permettre de changer de couple (taille message; taille clé) facilement. Ce point sera abordé plus loin dans ce chapitre.

Définir l'architecture du crypto-processeur est la première étape à réaliser.
Pour cela, il nous faut dégager les grands "blocs" fonctionnels qui doivent le composer.
Nous avons traduit les fonctionnalités internes du processeur en blocs matériels, comme le montre le tableau ci-dessous :

Un crypto-processeur Simon doit pouvoir ...	Bloc matériel correspondant
Générer les T clés de ronde	Bloc de génération des clés de ronde
Effectuer les T boucles de chiffrement/déchiffrement (*rondes*)	Bloc de chiffrement/déchiffrement (un ronde) Registre pour stocker les résultats entre deux *rondes*
Stocker le tableau des z_i	Mémoire ROM pour les z_i
Stocker les clés de ronde (au fur et à mesure)	Mémoire RAM pour les clés de ronde
Gérer tous les blocs et les entrées/sorties	Machine à états finis (FSM)

Le crypto-processeur est donc constitué de 6 blocs matériels principaux, que nous allons présenter un par un.

Remarque : nous avons envisagé une architecture de type pipeline. Seulement, si l'on déroule le pipeline complètement, cette architecture prend beaucoup trop de surface pour un crypto-processeur qui se veut *lightweight* (110875 GE pour le pipeline déroulé contre 3419 GE pour l'itératif pour le Simon 128/256, d'après une publication de la NSA - voir bibliographie). Si l'on ne déroule pas le pipeline, il faut gérer le rebouclage, ce qui nous a paru assez compliqué et long à mettre en place au vu des autres objectifs de notre projet (conception du générateur et passage sur FPGA).

A. Bloc de génération des clés de rondes

Ce bloc a pour rôle de générer les T clés utiles pour les rondes de chiffrement/déchiffrement. Si sa structure dépend de la valeur de m (comme montré au chapitre 1), ce bloc est toujours constitué des éléments suivants :

- Un registre à décalage qui reçoit en premier lieu la clé de chiffrement/déchiffrement. Ce registre est ensuite décalé de n bits (lorsque la FSM le demande), pour mettre en sortie la ième clé de ronde (de n bits) k_i, et ajouter une nouvelle clé de ronde k_{i+m} nouvellement calculée à l'autre extrémité du registre à décalage.
- Un bloc combinatoire pour calculer la (i+m)ième clé de ronde

Le bloc a besoin de la clé sur nm bits et des coefficients z_{ji} sur 1 bit en entrée. Sa sortie est sur n bits et correspond à la clé de ronde k_i.

On en déduit le schéma de la structure générale du bloc (indépendamment de la valeur de m) :

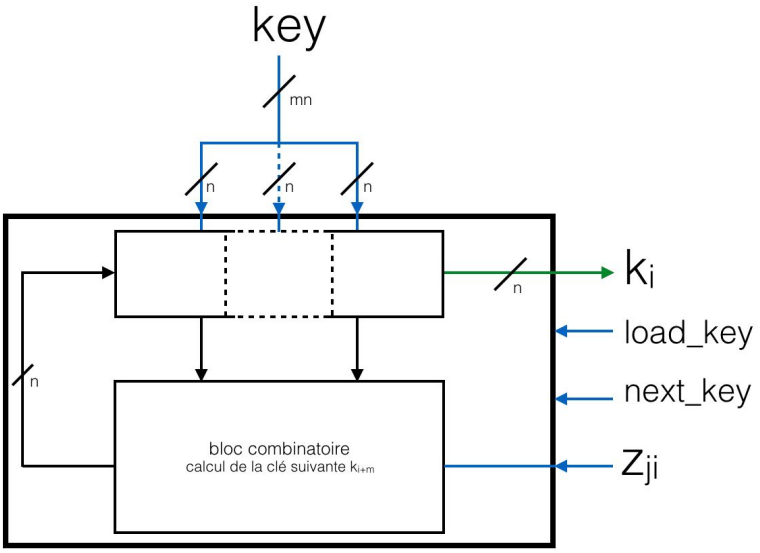


Figure 1 : structure du bloc de génération des clés de ronde

Le signal load_key, envoyé par la FSM, permet de demander le chargement de la clé donnée par l'utilisateur. Le signal *next_key*, également envoyé par la FSM, permet quand à lui de demander le décalage du registre.

B. Bloc de chiffrement/déchiffrement (ronde) et bloc registre

Ce bloc est le coeur du crypto-processeur. Il correspond à l'étage de chiffrement/déchiffrement (ou *ronde*) présenté au chapitre 1.
Rappelons rapidement l'utilité de ce bloc : à chaque *ronde*, la moitié du message (de taille n) est encodée seulement, auquelle on ajoute l'autre moitié via plusieurs *xor* (voir plus loin), avant d'être intervertie avec l'autre moitié pour le ronde suivant. Le nombre de ronde dépend de la version choisie, et est donné par le tableau précédent.

Notons dans la suite **x** la moitié gauche du message (n bits) et **y** sa partie gauche (n bits). Le message total est la concaténation de x et de y que l'on note **x.y** .

Un *ronde* a besoin du message (initial ou en cours de modification, c'est-à-dire qui est passé plusieurs fois par le ronde), et des clés de rondes k_i en entrée. Ces clés de rondes pouvant provenir directement du bloc de génération des clés de rondes ou de la mémoire qui leur est dédiée (le choix de la provenance des clés de rondes sera discuté plus loin dans ce chapitre), nous auront besoin d'un multiplexeur.

On en déduit la structure du bloc d'un ronde :

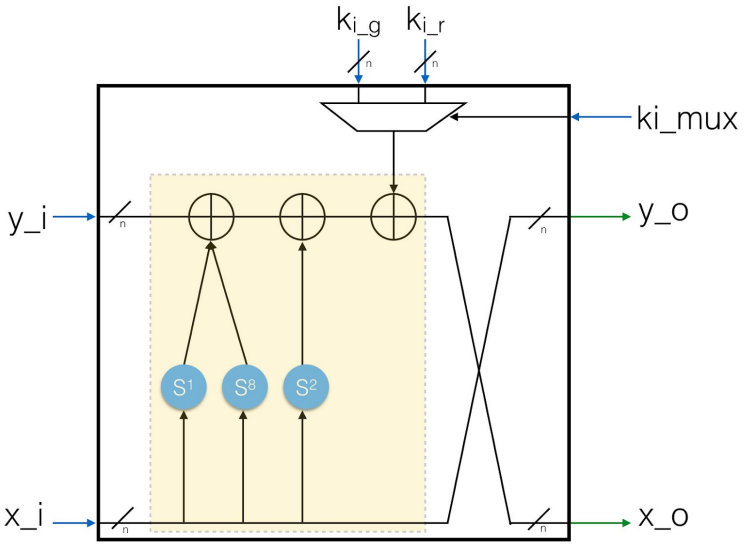


Figure 2 : structure du bloc d'un ronde

Les signaux *ki_g* et *ki_r* sont les clés de ronde qui proviennent respectivement directement du bloc de génération des clés de rondes ou de la mémoire qui leur est dédiée. Le signal *ki_mux* sert à commander le multiplexeur.

On utilise un registre externe pour sauver les résultats intermédiaires entre deux rondes (sauver x_o et y_o). Nous avons choisi d'utiliser un registre indépendant du bloc du ronde pour simplifier la gestion de ce bloc. En effet, les fonctions du bloc du registre sont nombreuses :

- le registre doit bien sûr sauver les résultats intermédiaires entre les rondes;
- le registre doit être accompagné d'un multiplexeur en entrée qui permette de choisir entre le message initial en entrée et la sortie du ronde;
- il existe une petite subtilité qu'il est important de noter : il faut inverser les entrées x_i et y_i avant d'entamer les rondes au début d'un cycle de déchiffrement, et inverser les sorties x_o et y_o à la fin du déchiffrement (car la structure du bloc inclut une inversion à sa sortie, comme le montre le schéma de la figure 2, qui n'est voulue lors du déchiffrement). Ces inversions se voient bien sur le schéma de la structure de Feistel à la figure 4 du chapitre 1. Nous avons choisi de gérer les inversions dans le bloc du registre, pour qu'elles soient transparentes pour le bloc du ronde, qui peut

ainsi rester simple et fonctionner de la même manière qu'il s'agisse d'un chiffrement ou d'un déchiffrement.

Il en découle la structure suivante pour le bloc du registre :

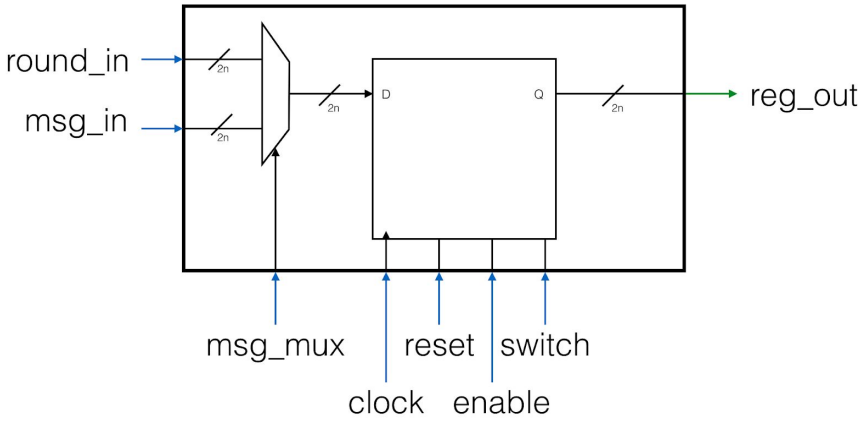


Figure 3 : structure du bloc registre

Le signal *ronde_in* correspond à la concaténation des deux sorties du bloc de ronde, le signal *msg_in* correspond au message initial donné par l'utilisateur. Le multiplexeur est commandé par le signal *msg_mux*.
Enfin, notons la présence du signal *switch*, qui permet d'inverser les parties droite et gauche de sa sortie (utile dans le cas d'un déchiffrement, comme expliqué plus haut).

C. Blocs mémoire

Nous avons besoin de deux blocs de mémoire :

- un bloc de mémoire RAM pour stocker les clés de ronde. Il doit pouvoir :
  - stocker les clés à l'adresse qu'on lui donne en entrée;
  - mettre sur sa sortie la clé qui est à l'adresse qu'on lui donne en entrée;
- un bloc de mémoire ROM pour stocker les constantes z_{ji}. Il doit pouvoir mettre sur sa sortie le coefficient z_{ji} correspondant à l'adresse qu'on lui donne en entrée.

L'interface de la mémoire RAM pour stocker les clés de ronde est la suivante :

Figure 4 : interface de la mémoire des clés de ronde

Le signal *mem_key* permet d'indiquer que l'on souhaite stocker une clé de ronde *ki_in*.
L'adresse de la clé que l'on souhaite lire ou écrire est donnée par *key_address.*
Note : [T] est le nombre de bits que compose le nombre de rondes T.

L'interface de la mémoire ROM, très simple, pour stocker les constantes z_{ji} est la suivante :

Figure 5 : interface de la mémoire des coefficients z_{ji}

D. Machine à états finis (FSM)

La machine à états finis a pour rôle de commander l'ensemble des blocs du crypto-processeur, et de gérer les signaux de commandes en entrée et en sortie.
Les signaux de commandes internes ont déjà étés présentés dans les structures des blocs précédents.

Dans la mesure où le crypto-processeur doit pouvoir à la fois chiffrer et déchiffrer, et garder la même clé pour plusieurs chiffrements/déchiffrements, les signaux de commandes externes comportent :

- en entrée :
  - un signal *start_stop* pour indiquer le début de la requête puis la bonne réception du message chiffré/déchiffré par le processeur maître.
  - un signal *cipher_sig* indiquant si l'on souhaite chiffrer ou déchiffrer
  - un signal new_key indiquant si l'on souhaite changer de clé
- en sortie, le signal *eoc* pour indiquer la fin du chiffrement/déchiffrement

Il en découle l'interface de la FSM suivante :

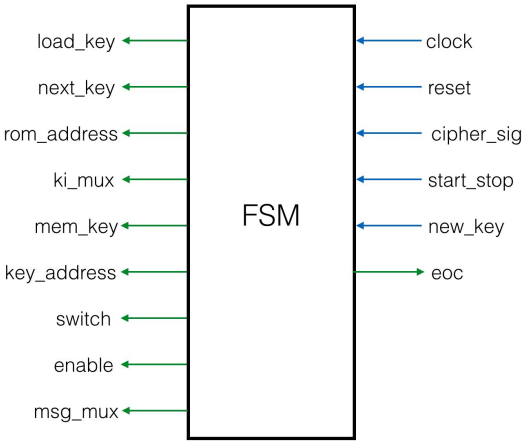


Figure 6 : interface de la FSM

La construction de la FSM est détaillée dans la partie suivante (II).

E. Architecture globale

Il ne reste plus qu'à assembler tous les blocs pour former le crypto-processeur ! L'interface de l'IP est donnée par la figure suivante (les signaux ont été présentés dans les parties précédentes):

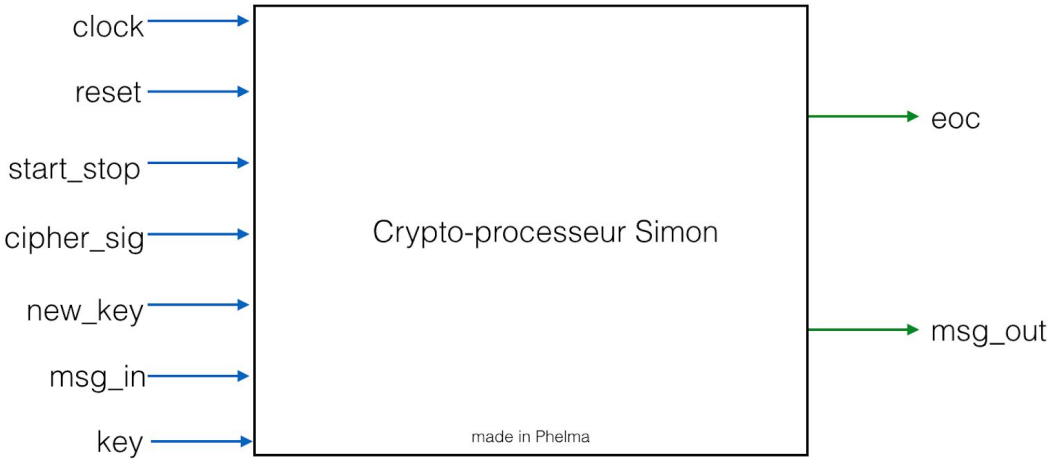


Figure 7 : interface de l'IP

Le schéma global du crypto-processeur est le suivant (ici le bloc de génération des clés dans le cas où m = 3) :

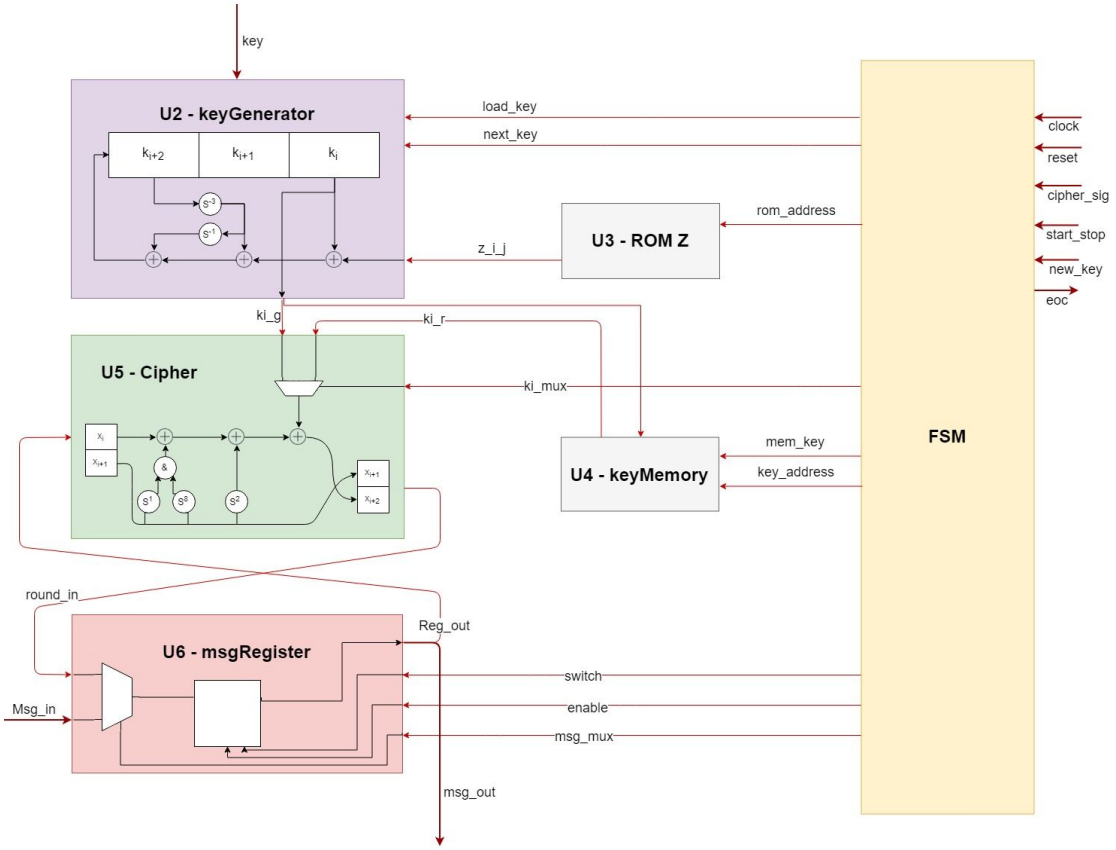


Figure 8 : architecture de l'IP

II. Machine à états

La machine à états finis doit gérer les cas suivants :

- un chiffrement avec une nouvelle clé
- un déchiffrement avec une nouvelle clé
- un chiffrement avec la même clé
- un déchiffrement avec la même clé

Pour construire la FSM, nous avons tracé les chronogrammes correspondants aux quatre cas précédents.

A. Chiffrement avec une nouvelle clé

Dans le cas d'un chiffrement avec une nouvelle clé, nous avons tracé le chronogramme ci-dessous :

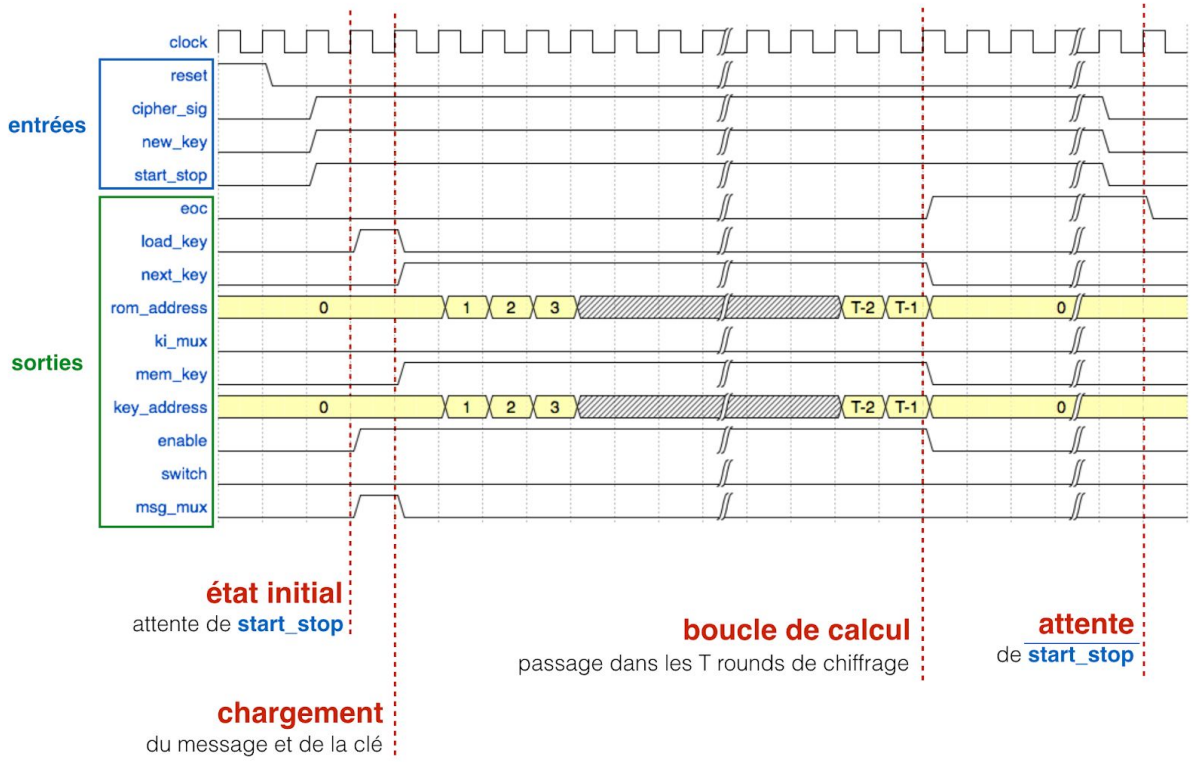


Figure 9 : chronogramme pour un chiffrement avec une nouvelle clé

Description du chronogramme

Etat initial (nommé **init**): c'est l'état dans lequel se trouve la FSM lorsqu'elle est en attente d'un chiffrement (donc en attente du signal *start_stop*), ou après un reset.

Chargement (nommé **load_cipher**): une fois le signal *start_stop* reçu, indiquant une nouvelle requête, il faut charger le message (*load_key*) et la clé (*msg_mux.enable*) dans les registres correspondants.

Boucle de calcul (nommé **cipher**): une fois le message et la clé chargés, il faut générer les clés de ronde (*next_key*) et enchaîner les rondes de chiffrement (*enable*, qui permet au registre de transmettre au bloc de ronde les valeurs intermédiaires à chaque front montant d'horloge). Ces deux tâches peuvent être effectuées en parallèle : chaque décalage

du registre des clés de ronde permet de générer la clé de ronde qui sera utilisée par le bloc de ronde au front montant suivant. Les clés de ronde sont également mémorisées (*mem_key*) pour servir lors d'un éventuel chiffrement/déchiffrement avec la même clé. Notons que, puisque les clés de ronde proviennent directement du générateur de clés, *ki_mux* est à zéro.

Attente (nommé **waitState**): une fois le message chiffré, le signal eoc est mis à 1 jusqu'à ce que le signal d'entrée *start_stop* redescende au niveau zéro. On repasse ensuite à l'état initial.

On peut à présent construire une première partie de la FSM :

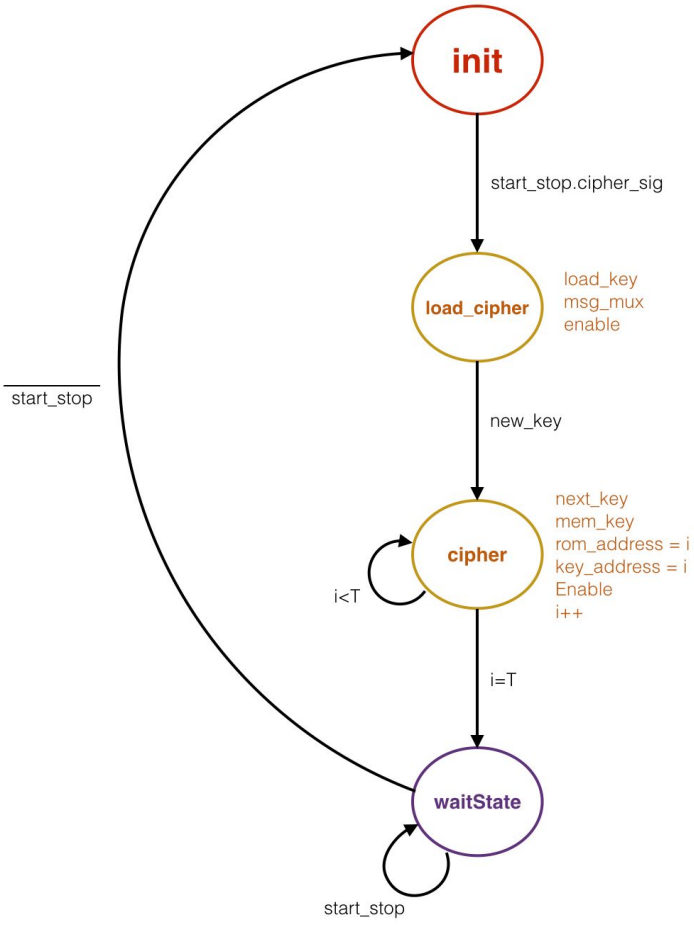


Figure 10 : FSM pour un chiffrement avec une nouvelle clé

Note : pour alléger l'écriture de la FSM, quand un signal est omis, c'est qu'il est à l'état zéro. Ainsi, à l'état init, tous les signaux (de sortie) sont à l'état bas.

B. Déchiffrement avec une nouvelle clé

Dans le cas d'un déchiffrement avec une nouvelle clé, nous avons tracé le chronogramme ci-dessous :

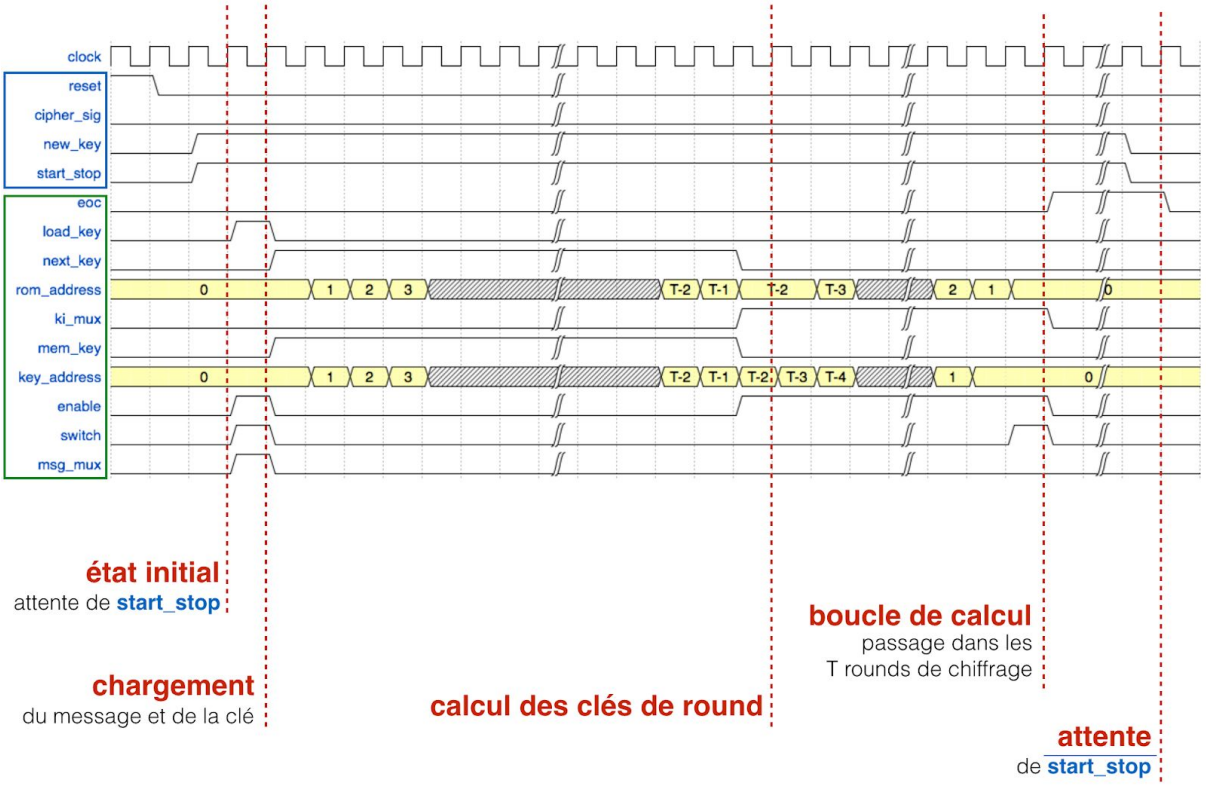


Figure 11 : chronogramme pour un déchiffrement avec une nouvelle clé

Description du chronogramme

Etat initial (nommé **init**): identique au cas précédent.

Chargement (nommé **load_decipher**): identique au cas précédent, à la différence près que le signal *switch* est à l'état haut pour inverser les deux moitié du message initial.

Calcul des clés de ronde (nommé **mem_key**): comme les réseau de Feistel utilise les clés de ronde dans l'ordre inverse pour le déchiffrement, on doit calculer(*next_key*) et mémoriser (*mem_key*) toutes les clés de ronde avant de déchiffrer le message.

Boucle de calcul (nommé **decipher**): une fois les clés de ronde calculées, il faut enchaîner les rondes de chiffrement (*enable*, qui permet au registre de transmettre au bloc de ronde les valeurs intermédiaires à chaque front montant d'horloge). Notons que, puisque les clés de ronde proviennent de la mémoire RAM des clés de ronde, *ki_mux* est à l'état haut. A la fin de la boucle de calcul, signal *switch* passe à l'état haut pour une période d'horloge pour inverser les deux moitié du message déchiffré.

Attente (nommé **waitState**): une fois le message déchiffré, le signal eoc est mis à 1 jusqu'à ce que le signal d'entrée *start_stop* redescende au niveau zéro. On repasse ensuite à l'état initial.

On peut à présent construire une deuxième partie de la FSM :

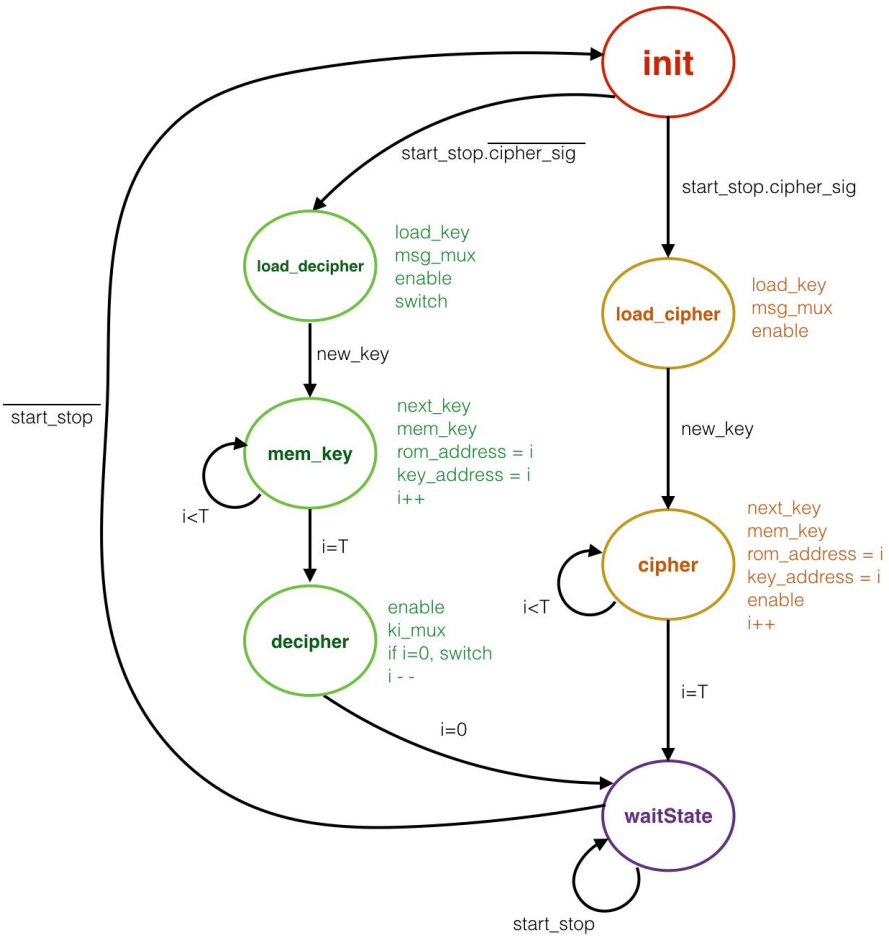


Figure 12 : FSM pour chiffrement/déchiffrement avec nouvelle clé

C. Chiffrement/Déchiffrement avec la même clé

Si l'on souhaite chiffrer ou déchiffrer en gardant la même clé que pour le chiffrement/déchiffrement précédent, il suffit de sauter un état dans la machine à états finis :

- l'état **cipher** si l'on souhaite chiffrer : on le remplace par un état (**recipher**) où on enchaîne les ronde sans calculer en parallèle les clés de ronde
- l'état **mem_key** si l'on souhaite déchiffrer

On peut en déduire la FSM complète :

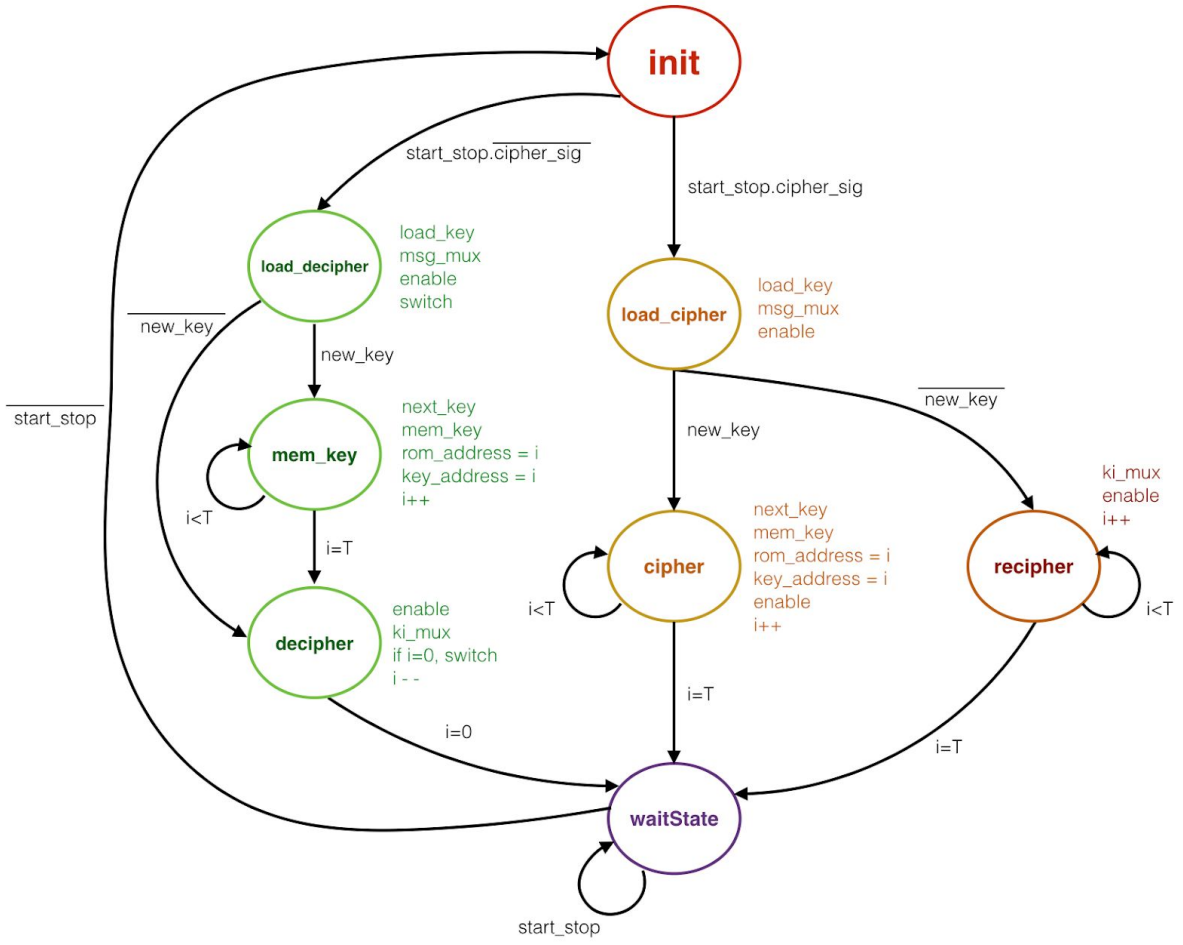


Figure 13 : FSM complète

III. Description VHDL

A présent que tous les blocs sont définis, il ne reste plus qu'à implémenter le circuit en VHDL. Nous avons créé un fichier .vhd pour chaque entité ("bloc" dans les parties précédentes), auxquels nous avons ajouté un fichier *cryptoProc.vhd* qui rassemble les blocs et qui décrit complètement le crypto-processeur.

Dans l'optique d'utiliser cette description VHDL comme base de départ pour notre générateur (voir chapitre suivant), nous avons ajouté un fichier *constants.vhd* (package) qui regroupe les grandeurs les plus utilisés dans la plupart des blocs du circuit :

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.numeric_std.all;
package constants is

constant N : integer := 64; -- 2*N est la longueur du message
constant Tbit : integer := 6; -- Tbit est le nombre de bits
```

```
constituant T (le nombre de rondes)
constant T : integer := 72; -- T est le nombre de rondes
constant NM : integer := 256; -- NM est la longueur de la clé
end constants;

package body constants is
end constants;
```

Tous les blocs VHDL utilisent ces constantes dans leur description. Cela permet en outre de faciliter la maintenance du code.

*Nous vous invitons à consulter le dossier **cryptoProc_FPGA** qui contient la description VHDL de la version 64/128 (avec chiffrement/déchiffrement et possibilité de sauvegarde de la clé) commentée.*

IV. Simulation et synthèse

A. Banc de test

Afin de simuler notre circuit, nous avons écrit un banc de test qui lance :

- un chiffrement avec une nouvelle clé suivi d'un chiffrement avec la même clé
- un chiffrement avec une nouvelle clé suivi d'un déchiffrement avec la même clé
- un déchiffrement avec une nouvelle clé suivi d'un déchiffrement avec la même clé
- un déchiffrement avec une nouvelle clé suivi d'un chiffrement avec la même clé

Nous avons testé le circuit avec les vecteurs de tests fournis par la NSA (voir bibliographie).

Afin de s'assurer du bon résultat de la simulation, nous avons insérés des "asserts" à des instants bien précis. Pour un chiffrement (avec ou sans nouvelle clé) ou un déchiffrement (avec la même clé), le résultat du calcul doit être obtenu après $T \times T_{clock} + 3 \times T_{clock}$ secondes, soit après T boucles de chiffrement et 3 périodes d'horloge qui correspondent :

- à une période dans l'état initial (qui attendait le signal *start_stop*, et qui passera à l'état suivant au front montant d'horloge)
- à une période pour le chargement de la clé (et éventuellement du message)
- à une période (au minimum) après la fin du chiffrement/déchiffrement pour remettre le signal *eoc* à zéro - une fois que le signal start_stop passe à zéro (dans le banc de test, ce signal passe à zéro dès que le signal *eoc* passe à un)

Pour un déchiffrement avec une nouvelle clé le résultat du calcul doit être obtenu après $2T \times T_{clock} + 3 \times T_{clock}$, car on doit ajouter T périodes pour le calcul des T clés de rondes, qui ne se fait pas en parallèle du déchiffrement dans ce cas (comme le montre le chronogramme du déchiffrement de la partie II).

Voici un extrait du banc de test pour la version 64/128 où l'on teste le chiffrement avec une nouvelle clé suivit d'un chiffrement avec la même clé :

```
report "test chiffrement/chiffrement ...";
-- TEST chiffrement/chiffrement AVEC LA MEME CLE

-- chiffrement avec nouvelle clé
msg_in <= x"656b696c20646e75"; -- le message d'entrée
key <= x"1b1a1918131211100b0a090803020100"; -- la clé
start_stop <= '1';
new_key <= '1';
cipher_sig <= '1';
reset <= '0';

wait until (eoc = '1') for time1; -- time1 est défini plus haut et
correspond à 470 ns (avec une période d'horloge de 10 ns)
assert (msg_out= x"44c8fc20b9dfa07a") report "erreur chiffrement C/C";
start_stop <= '0';

wait for 20 ns;

-- puis chiffrement avec la même clé
msg_in <= x"656b696c20646e75";
start_stop <= '1';
new_key <= '0';
cipher_sig <= '1';

wait until (eoc = '1') for time1;
assert (msg_out= x"44c8fc20b9dfa07a") report "erreur chiffrement avec même
clé C/C";
start_stop <= '0';

--  FIN chiffrement/chiffrement
```

*Nous vous invitons à consulter le dossier **cryptoProc_FPGA** qui contient le banc de test commenté.*

B. Résultats de la simulation

Parmis les nombreuses versions que nous pouvons tester, nous avons choisi de présenter ici les résultats de la simulation d'une version médiane : 64/128 (avec chiffrement/déchiffrement et la possibilité de garder la clé).

Les vecteurs de tests de la NSA associés à cette famille sont :
Clé : 1b1a1918 13121110 0b0a0908 03020100
Message clair : 656b696c 20646e75
Message chiffré : 44c8fc20 b9dfa07a

Voici le résultat de partie de la simulation où l'on teste le chiffrement avec une nouvelle clé suivit d'un chiffrement avec la même clé :

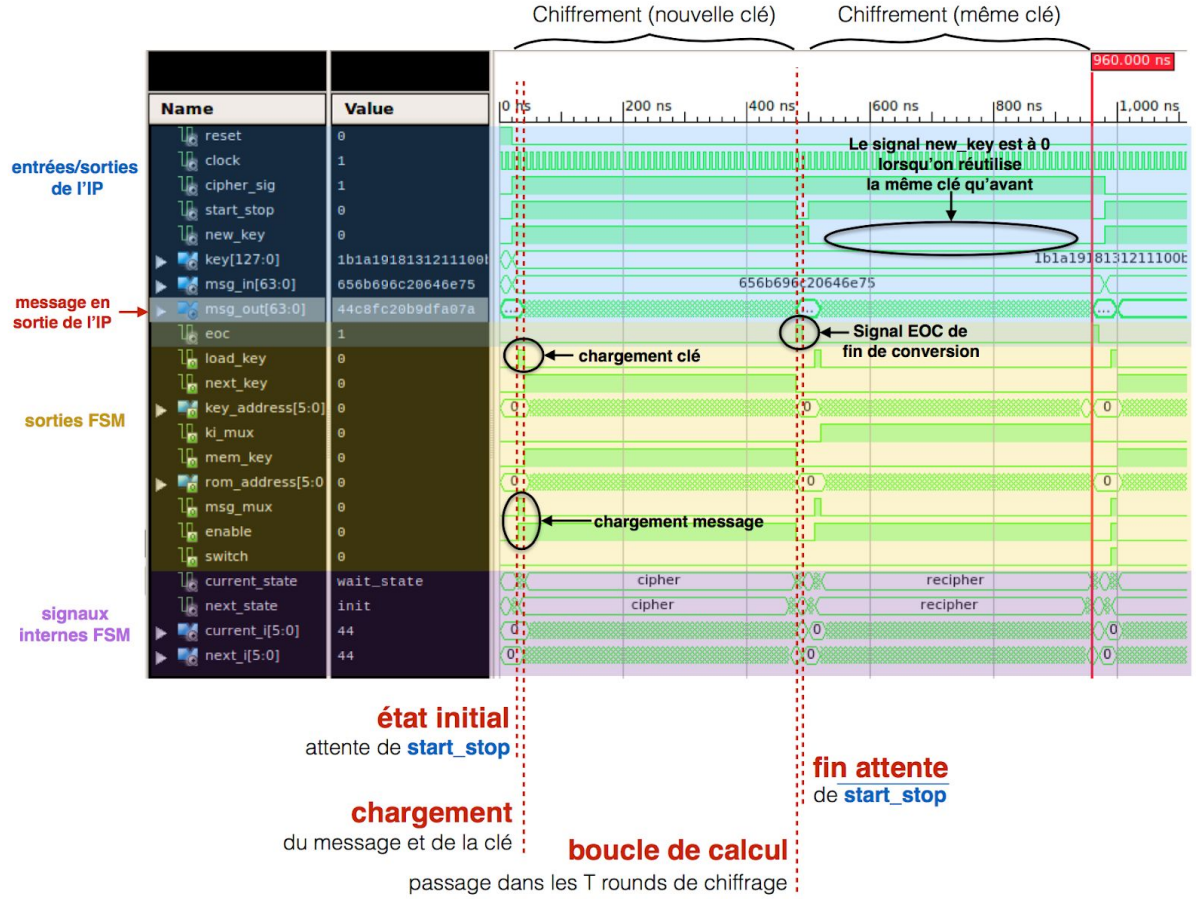


Figure 14 : chronogramme de simulation

La simulation n'a pas relevé d'erreurs (asserts), et on peut apercevoir sur le chronogramme de la figure 12 que le message chiffré issu du chiffrement avec la même clé est le bon (*value* pour *msg_out* : 44c8fc20b9dfa07a qui correspond à l'endroit où se trouve le curseur rouge). On aperçoit également le message d'entrée (*value* pour *msg_in :* 656b696c20646e75), mais la clé (*key*), trop longue, n'apparaît pas entièrement sur ce chronogramme.

Nous vous invitons à prendre connaissance du manuel d'utilisateur si vous souhaitez lancer la simulation et avoir accès à l'ensemble du chronogramme.

C. Synthèse

Nous avons ensuite synthétisé notre circuit sur une cible FPGA. Nous avons opté pour la carte FPGA **ZedBoard Zynq-7000** (pour des raisons de disponibilité de la carte - notre circuit étant simple et *lightweight*, il n'est pas très contraignant sur le choix de la carte FPGA). Les résultats de la synthèse sont présentés dans le chapitre suivant, avec ceux des autres versions que nous avons généré.

4 Générateur et scripts

Le but de cette partie est de présenter l'architecture du générateur Python ainsi que les scripts utilisés pour automatiser la compilation, la simulation, la synthèse et la simulation après synthèse de l'ensemble des versions des crypto-processeurs Simon créés par le générateur.

I. Objectif du générateur

Comme expliqué au chapitre 2, le rôle du générateur est de permettre la génération des 10 versions de crypto-processeurs de la famille Simon, avec leurs 6 options (ce qui fait un total de 60 versions). Le générateur doit produire un code VHDL synthétisable de la version choisie.

Rappelons rapidement les options que nous souhaitons intégrer dans le générateur :

- chiffrement seul
- déchiffrement seul
- chiffrement et déchiffrement

Et pour chacune de ces options, nous offrons deux possibilités :

- garder la même clé pour plusieurs chiffrements/déchiffrements
- recharger la clé à chaque chiffrement/déchiffrement

Nous allons à présent présenter la conception du générateur.

II. Conception du générateur

A. Structure du générateur

Le générateur, codé en Python, doit prendre en paramètres les tailles du message et de la clé, l'option de chiffrement (chiffrement, déchiffrement ou chiffrement/déchiffrement) et la politique d'utilisation de la clé, pour générer la description VHDL du crypto-processeur demandé.

Les descriptions VHDL des 60 versions à générer possèdent beaucoup de parties en commun. Dans le but d'optimiser le travail du générateur, et d'éviter à avoir à produire ce contenu commun directement via le code python avec de nombreux *print* longs et difficiles à écrire (retours à la ligne etc), nous avons opté pour la structure suivante :

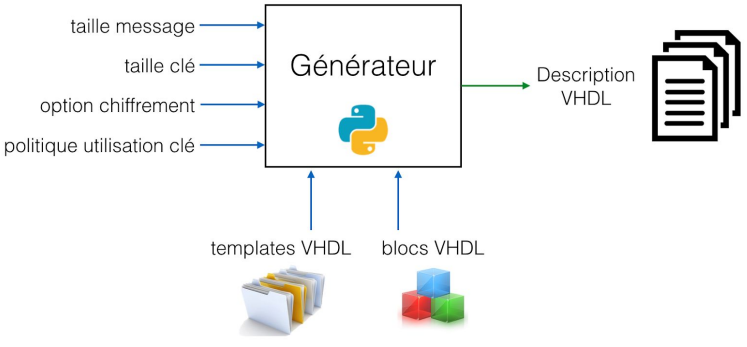


Figure 1 : présentation du générateur

Pour chacun des 9 fichiers .vhd de base qui constituent le crypto-processeur Simon 64/128, nous avons construit un fichier .vhd “à trous” qui ne conserve que les parties communes à toutes les versions du générateur. Nous appellerons ces fichiers des *templates*. Le rôle du générateur sera alors de remplir ces “trous” avec des blocs VHDL (plus ou moins grands) en fonction des paramètres qui lui sont donnés en entrée.
Notre générateur dispose donc de deux dossiers, un dossier ***vhd_templates*** qui regroupe les fichiers “à trous” et un dossier ***blocs*** qui regroupe les blocs VHDL à insérer au sein des fichiers templates.

La version 64/128 (avec chiffrement/déchiffrement et possibilité de garder la clé) présente les fichiers .vhd suivants :

fichier	description
cryptoProc.vhd	crypto-processeur complet, c’est dans ce fichier que l’on fait les mapping de tous les blocs du crypto-processeur
constants.vhd	fichier des constantes utilisées par le crypto-processeur
FSM.vhd	machine à état finis
romZ.vhd	mémoire ROM dans laquelle sont stockés le tableau des z_i
keyGenerator.vhd	bloc de génération des clés de chiffrement
keyMemory.vhd	mémoire ROM dans laquelle sont stockés les clés calculées par le bloc de génération des clés (keyGenerator.vhd)
cipher.vhd	bloc de chiffrement
msgRegister.vhd	registre (un peu élaboré) dans lequel sont stockés les chiffrements à chaque ronde
cryptoProc_tb.vhd	testbench

Etendre cette version aux 60 autres versions nécessite alors d’adapter tous ces fichiers en fonction de la famille et des options choisies.

B. Fichiers VHDL à compléter et blocs de remplacement

1. Présentation des fichiers à compléter et des blocs

Comme expliqué dans la partie précédente, le principe de ce générateur est de disposer de l’ensemble de fichiers VHDL de base pour décrire le crypto-processeur qui sont des fichiers à “*trous*”, c’est-à-dire à compléter en fonction de la version souhaitée par l’utilisateur.
Par exemple, le fichier *constants.vhd* dépend fortement de la version souhaitée. On procède alors a un petit traitement sur ce fichier : toutes les valeurs à compléter sont signalées par un symbole du type #symbole#. Nous verrons par la suite l’utilité de ce formalisme. On place aussi le fichier dans le dossier des fichiers VHDL à compléter.

Le fichier constants.vhd se présente alors ainsi :

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.numeric_std.all;
package constants is

constant N : integer := #N#;
constant Tbit : integer := #Tbit#;
constant T : integer := #T#;
constant NM : integer := #NM#;

end constants;

package body constants is
end constants;
```

Parfois, le bloc à compléter est complexe et varie beaucoup d'une version à l'autre. Dans ce cas, on crée un fichier texte par version du bloc, et le programme Python placera directement l'une de ces versions dans la zone à compléter.
C'est le cas par exemple du bloc de génération des clés de ronde *keyGenerator.vhd*, dont la structure varie beaucoup selon la valeur de m (2, 3 ou 4). On dispose alors de 3 fichiers texte qui décrivent ce bloc, un pour chaque valeur de m.

Le fichier *keyGenerator.vhd* devient alors :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.constants.all;

entity keyGenerator is
     Port ( load_key : in  std_logic;
          next_key : in  std_logic;
          key : in  std_logic_vector(NM-1 downto 0);
          z_j_i : in  std_logic_vector(0 downto 0);
          ki : out  std_logic_vector(N-1 downto 0);
     clock : in std_logic;
     reset : in std_logic);
end keyGenerator;

architecture Behavioral of keyGenerator is

#keyGenerator_architecture#

end Behavioral;
```

Le générateur remplacera `#keyGenerator_architecture#` par l'un des trois blocs décrits par les fichiers keyGenerator_m2.txt, keyGenerator_m3.txt, et keyGenerator_m4.txt.

Par exemple, le bloc keyGenerator_m2.txt est le suivant :

```
-- déclaration des signaux internes
signal k_ip1 : std_logic_vector(N-1 downto 0);
signal k_i : std_logic_vector(N-1 downto 0);
signal calc : std_logic_vector(N-1 downto 0);
signal shift_3 : std_logic_vector(2 downto 0);
signal shift_4 : std_logic_vector(3 downto 0);
signal zero : std_logic_vector(N-2 downto 0) := (others => '0');
signal un : std_logic_vector(N-3 downto 0) := (others => '1');

begin

-- calcul de la clé suivante
ki <= k_i;
shift_3 <= k_ip1 (2 downto 0);
shift_4 <= k_ip1(3 downto 0);
calc <= k_i xor (shift_3 & k_ip1(N-1 downto 3)) xor (shift_4 &
k_ip1(N-1 downto 4)) xor ((zero & z_j_i) xor (un & "00"));

-- description du registre à décalage
process(load_key, next_key, clock)
begin
    if (clock'event and clock = '1') then
      if (reset = '1') then
           k_i <= (others => '0');
           k_ip1 <= (others => '0');
      elsif (load_key = '1') then
           k_ip1 <= key(NM-1 downto N);
           k_i <= key(N-1 downto 0);
      elsif (next_key = '1') then
           k_i <= k_ip1;
           k_ip1 <= calc;
      end if;
    end if;
end process;
```

2. Les modifications à effectuer d'une version à l'autre

a) Changements à effectuer en fonction des tailles de message et de clé

Avant de présenter les modifications à apporter aux fichiers *blocs* en fonction des options (chiffrement et gestion de la clé), nous devons connaître ceux à apporter en fonction des tailles de message et de clé. Cela revient à savoir ce qui différencie les 10 versions de bases présentées par la NSA.

La principale modification à effectuer pour passer d'une de ces 10 version à l'autre est la **structure du bloc de génération des clés de rondes**. En effet, la structure de ce bloc dépend de la constante m (nm étant la taille de la clé, 2n étant la taille du message), qui peut valoir 2, 3 ou 4.

Le **fichier des constantes** doit évidemment être modifié car il contient les tailles de message, de clé, le nombre de rondes etc.
La **mémoire ROM des constantes z_{ji}** doit être modifiée selon la valeur de j (donnée par le tableau des 10 versions présentées par la NSA)
Le **banc de test** doit également être modifié pour inclure les bons vecteurs de test.

Le tableau ci-dessous regroupe les modifications à apporter en fonction des tailles de message et de clé (pour le banc de test, les vecteurs de tests sont dans la publication de la NSA citée dans la bibliographie) :

Taille du bloc **2n**	Taille de la clé **mn**	**constants.vhd**	**romZ.vhd**	***keyGenerator.vhd*** **(bloc utilisé)**
32	64	n =16 m = 4 T = 32	z_0	keyGenerator_m4
48	72	n =24 m = 3 T = 36	z_0	keyGenerator_m3
	96	n =24 m = 4 T = 36	z_1	keyGenerator_m4
64	96	n =32 m = 3 T = 42	z_2	keyGenerator_m3
	128	n =32 m = 4 T = 42	z_3	keyGenerator_m4
96	96	n =48 m = 2 T = 52	z_2	keyGenerator_m2
	144	n =48 m = 3 T = 54	z_3	keyGenerator_m3
128	128	n =64 m = 2 T = 68	z_2	keyGenerator_m2
	192	n =64 m = 3 T = 68	z_3	keyGenerator_m3
	256	n =64 m = 4 T = 68	z_4	keyGenerator_m4

b) Changements à effectuer en fonction des options

Les options introduisent de nombreux changements supplémentaires par rapport à ceux engendrés par les tailles de clés et de message.
Les principales modifications sont les suivantes :

- De nombreux états peuvent être supprimés de la FSM en fonction de l'option (par exemple, si l'on ne fait que du chiffrement, on supprime les états *load_decipher*, *mem_key* et *decipher*)
- Dans le cas où l'on chiffre seulement en rechargeant la clé à chaque reprise, la mémoire des clés est inutile et est supprimée
- De nombreux signaux peuvent être supprimé : par exemple, si l'on chiffre seulement, le signal *switch* qui permet d'inverser les parties droite et gauche du message est inutile. Où encore, le multiplexeur à l'entrée du bloc de ronde est inutile si l'on déchiffre (les clés viendront toujours de la mémoire des clés de ronde).
- … la liste est très longue!

Le tableau ci-dessous met en évidence les changements que l'on doit effectuer en fonction des options choisies, par rapport à une version avec chiffrement/déchiffrement avec possibilité de garder la clé.

Code couleur utilisé :

- les signaux à supprimer sont marqués en rouge
- les signaux internes à supprimer sont marqués en vert-bleu
- les états à supprimer sont marqués en vert
- les lignes particulières de code à supprimer sont marqués en violet
- les structures à supprimer (comme les multiplexeurs) sont marqués en bleu
- les cases à fond violet présentent les blocs VHDL à utiliser

Entité	Chiffrement seul		Déchiffrement seul		Chiffrement/Déchiffrement	
	complet	recharge clé	complet	recharge clé	complet	recharge clé
FSM	Cipher_sig switch load_decipher mem_key decipher	cipher_sig mem_key key_address New_key Switch ki_mux load_decipher mem_key decipher recipher	Cipher_sig ki_mux load_cipher cipher recipher	cipher_sig New_key ki_mux load_cipher cipher recipher Load_decipher => decipher		new_key recipher Load_decipher => decipher
	FSM_C_complet FSM_cipher_complet FSM_init_C FSM_load_cipher_complet FSM_mem_key_state FSM_recipher FSM_state_C_complet	FSM_C_recharge FSM_cipher_recharge FSM_init_C FSM_load_cipher_recharge FSM_state_C_recharge	FSM_D_complet FSM_decipher FSM_init_D FSM_load_decipher_complet FSM_mem_key_state FSM_state_D	FSM_D_recharge FSM_decipher FSM_init_D FSM_load_decipher_recharge FSM_mem_key_state FSM_state_D	FSM_state_CD_complet FSM_CD_complet FSM_init_CD FSM_load_decipher_complet FSM_decipher FSM_load_cipher_complet FSM_cipher_complet FSM_recipher FSM_mem_key_state	FSM_CD_recharge FSM_cipher_recharge FSM_decipher FSM_init_CD FSM_load_cipher_recharge FSM_load_decipher_recharge FSM_mem_key_state FSM_state_CD_recharge
keyMemory		supprimé				
msgRegister	switch	switch				
	msgRegister_port_C msgRegister_regout_C		msgRegister_port_DCD msgRegister_regout_DCD			
cipher		mux ki_r ki_mux	mux Ki_g ki_mux	mux Ki_g ki_mux		
	Cipher_mux cipher_process_mux	cipher_C_recharge cipher_process_C_recharge	cipher_D cipher_process_D		cipher_mux cipher_process_mux	
cryptoProc	cipher_sig switch	cipher_sig new_key keyMemory switch ki_reg ki_mux Key_address mem_key	cipher_sig ki_mux	cipher_sig new_key keyMemory ki_mux		new_key keyMemory
	cryptoproc_CxorD_complet FSM_C_complet msgRegister_port_C Cipher_mux	cryptoproc_CxorD_recharge FSM_C_recharge msgRegister_port_C cipher_C_recharge	cryptoproc_CxorD_complet FSM_D_complet msgRegister_port_DCD cipher_D	cryptoproc_CxorD_recharge FSM_D_recharge msgRegister_port_DCD cipher_D	cryptoproc_CD_complet FSM_CD_complet msgRegister_port_DCD Cipher_mux	cryptoproc_CD_recharge FSM_CD_recharge msgRegister_port_DCD Cipher_mux
cryptoProc_tb	cryptoproc_CxorD_complet bench _CC	cryptoproc_CxorD_recharge bench_C	cryptoproc_CxorD_complet bench_DD	cryptoproc_CxorD_recharge bench_D	cryptoproc_CD_complet bench_CC bench_DD bench_CD bench_DC	cryptoproc_CD_recharge bench_C bench_D

C. Programme Python

Les fichiers de remplacement

Le but du programme Python est d'effectuer les remplacements décrits dans la partie précédente. Pour assurer la modularité de notre programme, avons crée un fichier python par fichier vhdl à modifier:

- *constants.py* modifie *constants.vhd*
- *romZ.py* modifie *romZ.vhd*
- *keyGenerator.py* modifie *keyGenerator.vhd*
- *testbench.py* modifie *cryptoProc_tb.vhd*
- *etc.*

Dans la suite, nous appellerons ces fichiers Python des fichiers de remplacement.
Les fonctions de ces fichiers Python ont la même structure : leur but est de parcourir le fichier vhdl auquel ils sont associés à la recherche des symboles du type #symbole# présentés auparavant pour les remplacer par des blocs ou des valeurs.

Prenons l'exemple du fichier *keyGenerator.py*:

```
# keyGenerator
import fileinput
import sys

# on stocke le nom du fichier vhdl auquel est rattaché ce fichier
python
file_name = "../cryptoProc/keyGenerator.vhd"

def fill_keyGenerator(m):

    # on stocke dans la variable "architecture" le bloc de
génération des clés correspondant à la bonne valeur de m
    if m == 2:
      file = open('./patterns/keyGenerator_m2.txt', 'r')
      architecture = file.read()
      file.close()
    elif m == 3:
      file = open('./patterns/keyGenerator_m3.txt', 'r')
      architecture = file.read()
      file.close()
    elif m == 4:
      file = open('./patterns/keyGenerator_m4.txt', 'r')
      architecture = file.read()
      file.close()
    else :
      raise ValueError('m out of range (must be between 2 and
4)')

    # on parcourt le fichier keyGenerator.vhd à la recherche du
symbole #keyGenerator_architecture#, une fois trouvé on le
remplace par le bloc stocké dans la variable "architecture"
```

```
    for line in fileinput.FileInput(file_name, inplace =1):
      if '#keyGenerator_architecture#' in line:
           line =
line.replace('#keyGenerator_architecture#',architecture)
      print line,

if __name__ == '__main__':
    fill_keyGenerator(int(sys.argv[1]))
```

Fichier principal : generator.py
Il faut ajouter à ces fichiers de remplacement un fichier python principal qui sera appelé pour lancer le générateur. Nous avons nommé ce fichier *generator.py*.

Le rôle de ce fichier est de :

- Récupérer les choix de l'utilisateur en terme de tailles de message et de clé : nous avons opté pour récupérer ces choix via les options de ligne de commande. Cela permettra d'intégrer facilement l'exécution du générateur dans des scripts shell (voir plus bas)
- copier le dossier des fichiers vhdl à compléter **`vhd_template`** à la racine du projet, et d'écraser une éventuelle version qui lui est antérieure. Tous les fichiers Python du générateur vont modifier les fichiers VHDL contenus dans cette copie, et non pas dans le dossier d'origine, qui doit rester propre et non modifié, pour permettre de le réutiliser indéfiniment. La copie est appelée `cryptoProc`.
- Faire appel aux fonctions python des fichiers de remplacement pour compléter les fichiers VHDL

La ligne de commande qui permet d'exécuter le générateur est la suivante :
`python generator.py -k "taille clé" -b "taille message" -p "option de chiffrement" -opt "politique d'utilisation de la clé"`

Les options de chiffrement sont les suivantes :

- **cd** : chiffrement/déchiffrement
- **c** : chiffrement seul
- **d** : déchiffrement seul

Les politiques d'utilisation de la clé sont les suivantes :

- **c** : possibilité de garder la clé
- **r** : la clé doit être rechargé à chaque chiffrement/déchiffrement

Ainsi, par exemple, si l'on souhaite générer la version Simon 64/128 pour le déchiffrement seulement, et sans possibilité de garder la clé, on écrira dans le terminal :
`python generator.py -k 128 -b 64 -p d -opt r`

Voici le code du fichier *generator.py* :

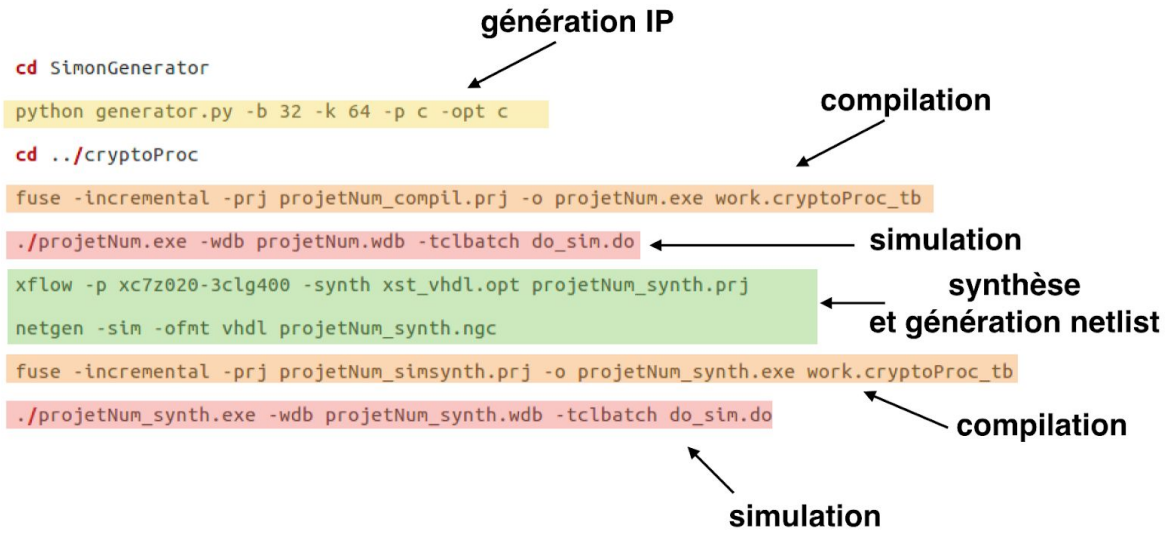
```
# generator of Simon's cryptoprocessors

#import modules
# et on importe les fonctions des fichiers de remplacement
from keyGenerator import *
```

```
from romZ import *
from constants import *
from testbench import *
import shutil
import os

from argparse import ArgumentParser

def main():

# on supprime le dossier du précédent du processeur généré s'il
existe pour en créer un nouveau à la racine du projet
    if os.path.exists('../cryptoProc') and
os.path.isdir('../cryptoProc'):
            shutil.rmtree('../cryptoProc')
    shutil.copytree('./vhd_template','../cryptoProc')

# on récupère les données de l'utilisateur via les options de
lignes de commande
    parser = ArgumentParser()
    parser.add_argument("-k", "--keysize", choices = ["64", "72",
"96", "128", "144", "192", "256"], help="Key size",
required=True)
    parser.add_argument("-b", "--blocksize",
choices=["32","48","64","96","128"], help="Block size",
required=True)
    parser.add_argument("-p", "--purpose",
choices=["c","d","cd"], help="Purpose", required=True)
    parser.add_argument("-opt", "--options", choices=["c","r"],
help="Options", required=True)
    args = parser.parse_args()

# on fait appel aux fonctions de remplacement
    (z_index, m, T) =fill_constants(int(args.blocksize),
int(args.keysize))
    fill_FSM(args.purpose, args.options)
    fill_msgRegister(args.purpose, args.options)
    fill_cipher(args.purpose, args.options)
    fill_cryptoProc(args.purpose, args.options)
    fill_fichiersPRJ(args.purpose, args.options)
    fill_keyGenerator(m)
    fill_romZ(z_index)
    fill_testbench(int(args.blocksize), int(args.keysize), T,
args.purpose, args.options)

if __name__ == '__main__':
     main()
```

III. Scripts

Le grand nombre de version que nous devons générer nous a poussé à utiliser des scripts pour automatiser la génération, la compilation, la simulation, la synthèse et la simulation après synthèse.

Voici le script de base pour une version particulière :



Ce script nécessite la présence de quatre fichiers dont nous n'avons pas parlé précédemment :

- ***do_sim.do***, qui contient les commandes pour le simulateur Isim, et permet de simuler sans lancer l'interface de Isim :

```
run 10 us
exit
```

- ***projetNum_compil.prj***, qui contient la liste des fichiers .vhd à compiler
- ***projetNum_synth.prj***, qui contient la liste des fichiers .vhd à synthétiser. Son écriture diffère légèrement de ***projetNum_compil.prj***
- ***projetNum_simsynth.prj*** qui contient le nom du fichier .vhd généré après la synthèse, et qui permettra de lancer la simulation après synthèse

Nous avons inclu ces quatre fichiers dans le dossier des *template*, et nous avons crée un fichier Python fichiersPRJ.py qui modifie les fichiers ***projetNum_compil.prj*** et ***projetNum_synth.prj*** (supprime ou ajoute des fichiers en fonction des paramètres du générateur).

La synthèse se fait sur cible FPGA (c'est le but même du projet). Comme présenté dans le chapitre précédent, nous avons opté pour la carte FPGA **ZedBoard Zynq-7000** (pour des raisons de disponibilité de la carte - notre circuit étant simple et *lightweight*, il n'est pas très contraignant sur le choix de la carte FPGA).
La ligne de commande de la synthèse précise la cible avec le bon package (3clg400) :
`xflow -p **xc7z020-3clg400** -synth xst_vhdl.opt projetNum_synth.prj`

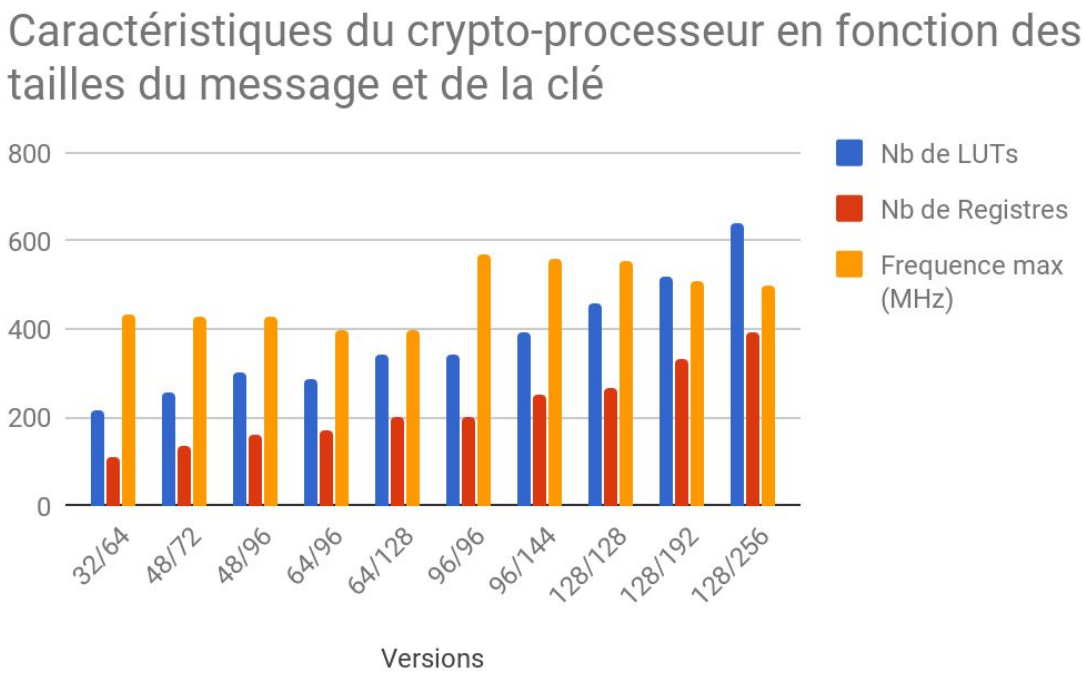
IV. Résultats

Une fois munis des scripts, nous avons pu générer, simuler, synthétiser (et simuler après synthèse) les 60 versions proposées par notre générateur.

Nous avons relevé ce que nous avons jugé être les trois principales caractéristiques des circuits synthétisés :

- le nombre de LUTs utilisés
- le nombre de registres utilisés
- la fréquence maximale d'utilisation

Le graphique ci-dessous présente l'évolution de ces caractéristiques en fonction des tailles de message et de clé (avec chiffrement/déchiffrement et possibilité de garder la clé) :



On remarque que l'évolution du nombre de registre et du nombre de LUTs semble cohérente : plus les tailles du message et de la clé sont importantes, plus le nombre de registres et de LUT utilisés est important.

L'évolution de la fréquence maximale est quand à elle un peu déconcertante. En effet, pour les 5 premières versions, la fréquence maximale a l'évolution attendue (elle baisse plus la complexité du circuit augmente), mais elle augmente soudainement pour la version 96/96, avant de baisser à nouveau.

Le critère par défaut d'optimisation avec *xflow* est la vitesse. L'augmentation soudaine de la fréquence maximale est peut-être dûe à un changement d'algorithme de synthèse, peut-être déclenché par le changement de la structure de génération des clés (première utilisation de la structure avec m = 2).

En théorie, le nombre de registres utilisés doit être la somme de:

- la taille du message (2n)
- la taille de la clé (nm)
- la taille du compteur i de la FSM (qui est égale à la taille de T, le nombre de rondes)
- la taille du codage des états (codage binaire)

Cependant, pour les trois premières versions (32/64, 48/72 et 48/96), l'outil de synthèse juge que la taille de la mémoire des clés de ronde est trop faible et décide de synthétiser cette mémoire avec des bascules (flip-flop). Il en découle que la variable interne de l'adresse de lecture read_address est sauvée dans des registres (le nombre de registres étant égal à la taille de read_address, qui est égale à la taille de T).

Ainsi, pour la version 96/96 par exemple, on a :

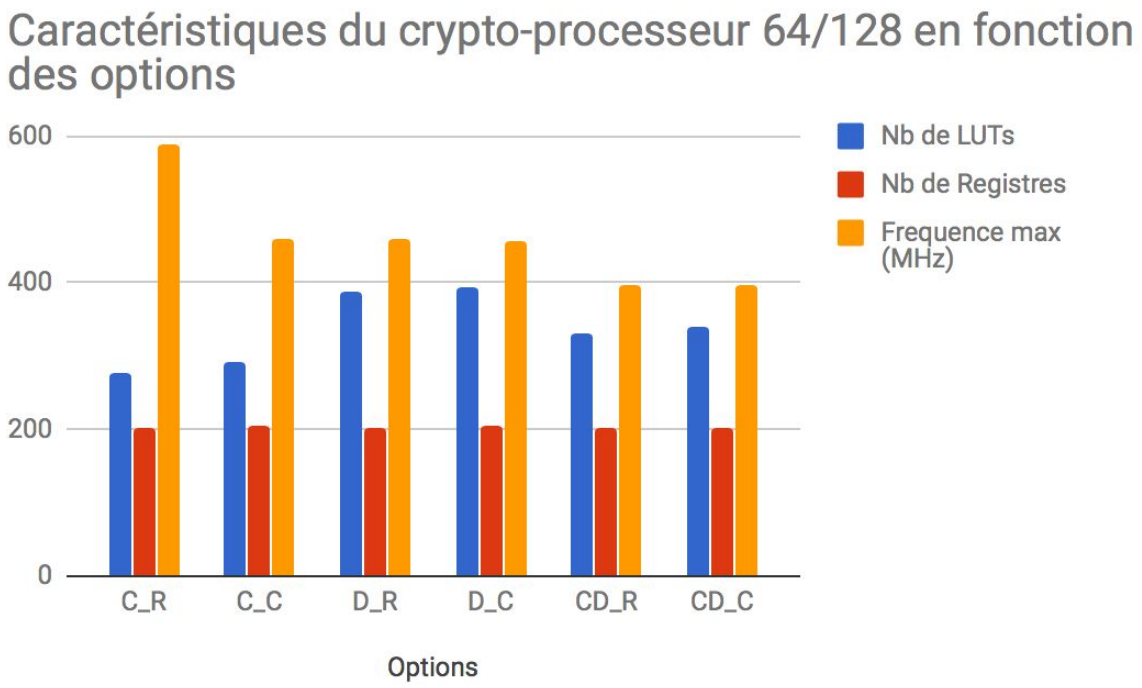
- la taille du message : 96
- la taille de la clé : 96
- la taille du compteur i : 6
- la taille du codage des états : 3

On doit donc obtenir 96 + 96 + 6 + 3 = 201 registres.
Le résultat après synthèse est de 203 registres. En effet, pour des raisons de timing, l'outil de synthèse duplique certains signaux, ici il a dupliqué deux fois une composante du signal *current_state*. On obtient alors 201+2 = 203 registres!

On vérifie de même que nous obtenons les bonnes valeurs pour toutes les versions :

Taille message	Taille clé	Taille i	Taille codage états	Nb registres attendu	Taille read_address	Réplicats de signaux	Nb registres obtenu
32	64	6	3	105	5		110
48	72	6	3	129	6		135
48	96	6	3	153	6		159
64	96	6	3	169			169
64	128	6	3	201			201
96	96	6	3	201		2	203
96	144	6	3	249		2	251
128	128	7	3	266		1	267
128	192	7	3	330			330
128	256	7	3	394			394

On souhaite ensuite observer l'évolution des caractéristiques du crypto-processeur en fonction des options. On choisi alors de faire varier les options du Simon 64/128 :



On trouve en abscisse les options du crypto-processeur sous la forme A_B :

- A est l'option de chiffrement : C pour le chiffrement seul, D pour le déchiffrement seul, CD pour le chiffrement et le déchiffrement.
- B est la politique d'utilisation de la clé : C lorsqu'on souhaite pouvoir réutiliser la clé, R lorsqu'on la charge à chaque chiffrement/déchiffrement.

Le nombre de registre est constant, ce qui est naturel car il ne dépend que de la taille de la clé et du message, du nombre de rondes et de la taille du codage des états (on ne supprime jamais assez d'états pour diminuer cette taille).

La fréquence maximale d'utilisation diminue, et semble assez indépendante de la politique d'utilisation de la clé, sauf pour le cas du chiffrement avec réutilisation de la clé, pour lequel la fréquence maximale est plus faible lorsqu'on réutilise la même clé. Cela est peut-être dû au temps d'accès à la mémoire des clés de rondes en écriture et en lecture, et à la gestion des nombreux signaux supplémentaires dont se passe la version avec rechargement de la clé. Comme le montre le tableau des changements à effectuer en fonction des options, la version C_R est la plus légère (beaucoup de signaux et d'états sont supprimés, et la mémoire des clés de rondes est également supprimée).

5 Implémentation sur FPGA

Le but de cette partie est de présenter notre démarche et nos résultats pour l'implémentation du crypto-processeur Simon 64/128 avec chiffrement/déchiffrement et possibilité de garder la clé.

I. Cible FPGA

Comme présenté dans les parties précédentes, nous avons choisi de travailler sur la carte FPGA **ZedBoard Zynq-7000**.

Nos besoins en terme de matériel ne sont pas très importants (ce circuit se veut *lightweight*) : nous avons besoins d'au moins 267 registres (pour la version 64/128, voir fin du chapitre précédent), de deux mémoires RAM de 64 bits pour les constantes z_{ji} et de 52*48 = 2496 bits pour la mémoire des clés de rondes. Nous avons également besoins de 6 entrées (reset, start_stop, cipher_sig, new_key, msg_in (bit par bit en série, voir partie suivante), key (également bit par bit en série)) et 2 sorties.

La carte **ZedBoard Zynq-7000** présente 106400 registres, 125 entrées/sorties et de 512 Mbits de mémoire RAM. Cette carte couvre donc largement nos besoins au niveau matériel.

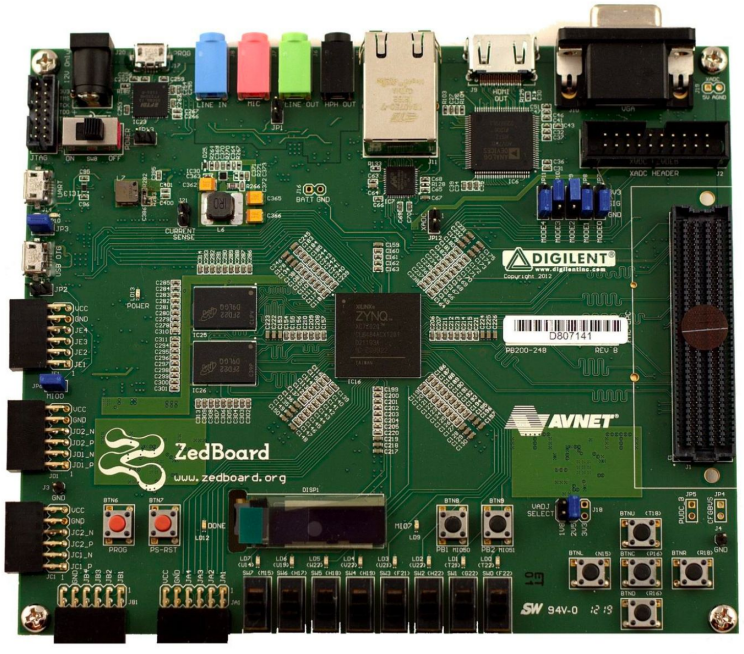


Figure 1 : carte ZedBoard Zynq-7000
Image copiée du manuel d'utilisation de cette carte FPGA

II. Conception du Wrapper

Au vu des très grandes tailles des entrées/sorties de notre IP (64 bits * 2 pour le message en entrée et en sortie et 128 bits pour la clé), la carte FPGA choisie ne dispose pas d'assez d'entrées/sorties pour répondre à nos besoins. Il nous faut donc transmettre les bits du message et de la clé en série.

Pour communiquer en série avec l'ordinateur, nous nous sommes basé sur le protocole de communication série RS-232, dont le principe est très simple : les bits sont envoyés en série par paquets (de 8 bits maximum), précédés d'un bit de start à 0 et suivis d'un bit de stop à 1 (on peut également ajouter un bit de parité, mais nous nous sommes contentés des bits de start et de stop pour simplifier le projet).
Le schéma ci-dessous montre une trame respectant le protocole RS-232 :

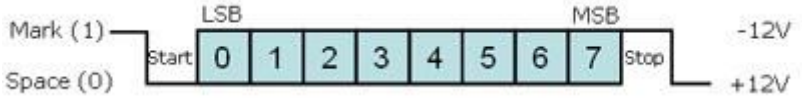


Figure 2 : trame de 8 bits avec le protocole RS-232
Schéma copié du site www.commfront.com

Pour recevoir le message, la clé, les signaux de contrôle et envoyer le message chiffré/déchiffré, nous avons besoins d'une UART. Le rôle de l'UART est :

- en réception : de recevoir les informations par paquet de 8 bits conformément au protocole RS-232, les ajouter à un registre à décalage après avoir retiré les bits de start et de stop, et, une fois l'ensemble des informations reçues, donner à l'IP l'accès au registre à décalage;
- en émission : découper le message chiffré/déchiffré en paquets de 8 bits, les mettres en formes conformément au protocole RS-232, et les envoyer.

L'émission et la réception se feront à une vitesse de 115200 bauds.

A. Développement de l'UART

Au vu du travail assez important et varié que nous devions fournir pour ce projet, nous n'avons pas développé notre UART par nous même. Nous nous somme basés sur l'UART proposée par le site www.nandland.com. Celui-ci met gratuitement à disposition des internautes une UART qui permet de recevoir et d'envoyer des bits en séries conformément au protocole RS-232.

Cependant, cette UART ne pouvait gérer qu'un seul paquet de 8 bit. Nous avons donc dû la modifier pour ajouter un registre à décalage pour pouvoir stocker les paquets de 8 bits jusqu'à la réception de l'ensemble des informations (message, clé, signaux de contrôle). Nous avons également apporté d'autres modifications, comme l'ajout d'un reset.

L'interface de l'UART de réception est la suivante :

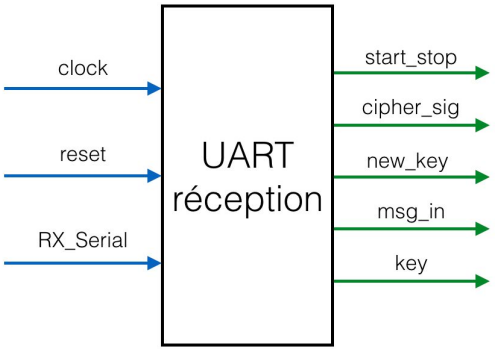


Figure 3 : interface de l'UART de réception

L'UART de réception reçoit les données par paquets de 8 bits via l'entrée ***RX_Serial***. Elle les stocke dans un registre à décalage, jusqu'à réception des 25 paquets nécessaires à l'IP : en effet, la taille totale des informations que l'on doit transmettre à l'UART est la somme de :

- la taille du message : 64
- la taille de la clé : 128
- le nombre de signaux de contrôle : 3

Soit un total de 195 bits. C'est-à-dire 195/8 = 24,3 soit 25 paquets de 8 bits.

Une fois les 25 paquets reçus, l'UART met les informations contenues dans le registre à la sortie de celui-ci, afin que l'IP de chiffrement/déchiffrement puisse les utiliser.

L'interface de l'UART d'émission est la suivante :

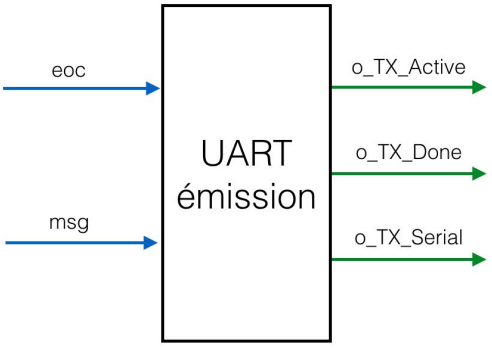


Figure 4 : interface de l'UART d'émission

L'UART d'émission a pour mission d'envoyer le message de son entrée ***msg*** dès réception du signal eoc. Elle envoi le message ***msg*** par paquets de 8 bits conformément au protocole RS-232 sur sa sortie ***o_TX_Serial***. Pendant l'envoi, le signal ***o_TX_Active*** est à l'état haut, et une fois l'envoi de l'ensemble des paquets terminé, le signal ***o_TX_Done*** passe à l'état haut pendant une période d'horloge.

B. Mappage de l'IP avec les deux UART

Maintenant que nous disposons d'une UART, nous pouvons connecter l'UART à l'IP de chiffrement/déchiffrement.

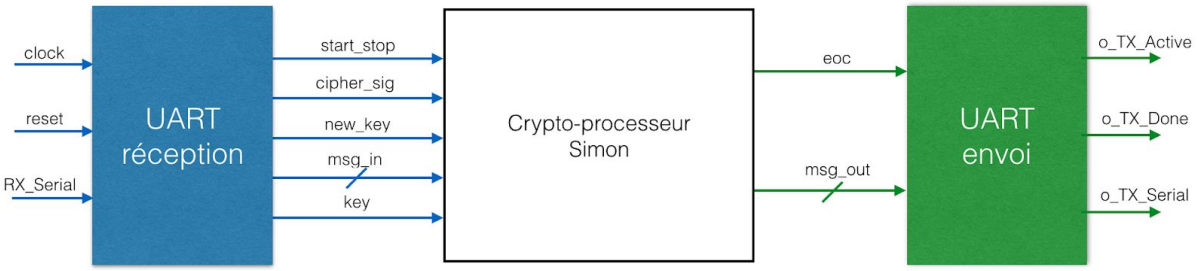


Figure 5 : les UART avec le crypto-processeur

Il est temps à présent de définir la constitution de la trame complète que l'on enverra à l'UART de réception, car elle permettra d'effectuer le bon mappage entre l'UART de réception et le crypto-processeur.

Nous avons défini la trame suivante :

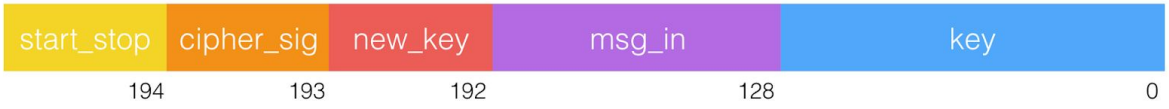


Figure 6 : trame d'envoi

Ainsi, on envoie tout d'abord la clé, puis le message, puis les signaux de contrôle (start_stop, cipher_sig et new_key).

Note : une fois la trame envoyé, et le message chiffré/déchiffré reçu, il faut absolument envoyer une nouvelle trame constituée de zéros pour remettre le signal start_stop à zéro et permettre à l'IP de repasser à l'état initial.

Ainsi, la description complète du crypto-processeur avec son wrapper est donnée par les fichiers suivant :

- **top.vhd** (qui regroupe tous l'UART et le crypto-processeur)
  - **UART_RX.vhd** : l'UART de réception
  - **cryptoProc.vhd** : le fichier top du crypto-processeur
    - **FSM.vhd**
    - **keyGenerator.vhd**
    - **keyMemory.vhd**
    - **romZ.vhd**
    - **cipher.vhd**
    - **msgRegister.vhd**
  - **UART_TX.vhd** : l'UART d'émission

III. Test

Nous pouvons à présent passer au test de notre circuit sur la carte FPGA! Mais avant cela, nous devons définir le brochage des entrées/sorties et trouver un moyen pour envoyer et recevoir des trames depuis l'ordinateur.

A. Définition du brochage

Nous devons d'abord définir le brochage de nos entrées/sorties sur la carte FPGA. Nous nous sommes inspirés pour cela du fichier .ucf fourni par M. Rolland, et nous tenons à l'en remercier.

Notre fichier pins.ucf est le suivant :

```
net "clock"  LOC = "Y9" | IOSTANDARD = LVCMOS33;     # LD0
net "reset" LOC = "P16" | IOSTANDARD = LVCMOS33;     # BTNS
net "RX_Serial" LOC = "W8" | IOSTANDARD = LVCMOS33;     # JB4
net "o_TX_Serial" LOC = "V10" | IOSTANDARD = LVCMOS33;     # JB3
net "o_TX_Done"  LOC = "W11" | IOSTANDARD = LVCMOS33;     # JB2
net "o_TX_Active"  LOC = "W12" | IOSTANDARD = LVCMOS33;     # JB1
```

Concrètement, cela correspond à connecter les pins d'entrée/sorties de cette manière sur la carte FPGA :

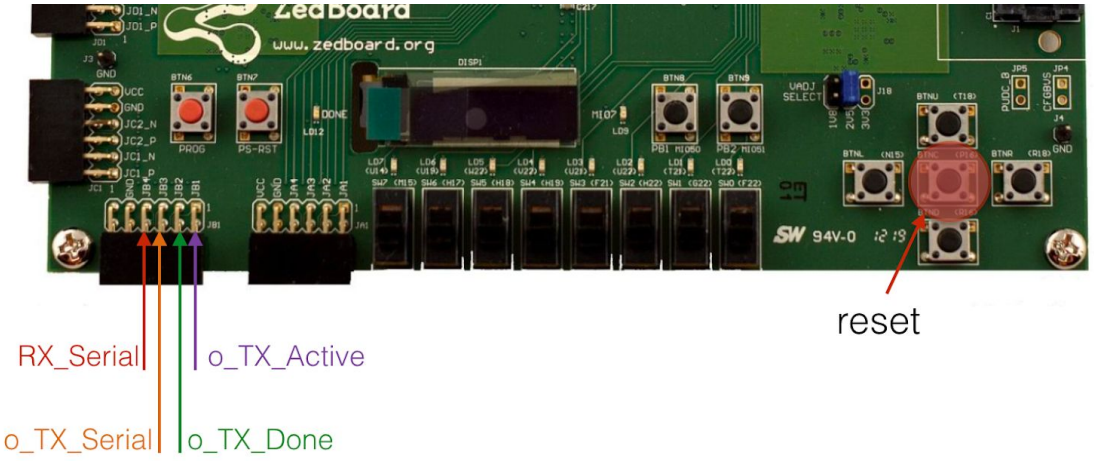


Figure 7 : brochage sur la carte FPGA

B. Envoi et réception d'une trame depuis l'ordinateur

Pour envoyer une trame à la carte FPGA, nous avons utilisé un fichier Python (basé sur la librairie Serial) qui permet d'envoyer les données en binaire selon le protocole RS-232. Ce fichier nous a été fourni par M. Rolland.

Nous avons adapté ce fichier à nos besoins, en ajoutant notamment des lignes de codes permettant de recevoir directement le résultat du chiffrement/déchiffrement sur l'ordinateur et de l'afficher sur le terminal.

Voici le fichier Python que nous avons utilisé (après modifications):

```
#importation des librairies utiles
import time
import serial

# configuration de la connection série
ser = serial.Serial(
    port='/dev/ttyUSB0',
    baudrate=115200,
    parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_ONE,
    bytesize=serial.EIGHTBITS,
    timeout=1
)

#ouverture de la liaison série
ser.isOpen()

idn=(0x02)
length= (0x04)
mode= (0x03) # write
data= (0x19)

##envoi de la trame
# envoi de la clé
ser.write(chr(0x00))
ser.write(chr(0x01))
ser.write(chr(0x02))
ser.write(chr(0x03))
ser.write(chr(0x08))
ser.write(chr(0x09))
ser.write(chr(0x0a))
ser.write(chr(0x0b))
ser.write(chr(0x10))
ser.write(chr(0x11))
ser.write(chr(0x12))
ser.write(chr(0x13))
ser.write(chr(0x18))
ser.write(chr(0x19))
ser.write(chr(0x1a))
ser.write(chr(0x03))
# envoi du message
ser.write(chr(0x13))
ser.write(chr(0xd6))
ser.write(chr(0x31))
ser.write(chr(0x50))
ser.write(chr(0xfa))
ser.write(chr(0x1a))
ser.write(chr(0x0b))
ser.write(chr(0x21))
# envoi des signaux de contrôle
ser.write(chr(0x05))

# réception du résultat
```

```
serial_readline = ser.readline()
time.sleep(0.01)
print(serial_readline.encode("hex")) # affichage du résultat sur le
terminal
#fermeture de la liaison série
ser.close()

print("fin de la routine")
exit
```

C. Test

Il ne reste plus qu'à envoyer la trame ! Nous allons tester le circuit avec les vecteurs de tests de la NSA pour la version 64/128 :

Clé : 1b1a1918 13121110 0b0a0908 03020100
Message clair : 656b696c 20646e75
Message chiffré : 44c8fc20 b9dfa07a

Nous formons plusieurs trames pour pouvoir tester le circuit :

```
Chiffrage, nouvelle clé :
07 656b696c20646e75 1b1a1918131211100b0a090803020100
Déchiffrage, même clé :
04 44c8fc20b9dfa07a 00000000000000000000000000000000
Chiffrage, même clé :
06 656b696c20646e75 00000000000000000000000000000000
Déchiffrage, nouvelle clé :
05 44c8fc20b9dfa07a 1b1a1918131211100b0a090803020100
```

On crée un fichier python par trame pour faciliter les manoeuvres sur le terminal, et on lance le programme. Il ne faut pas oublier d'appuyer sur le bouton reset avant !

Par exemple, pour le test du déchiffrement avec une nouvelle clé :

```
xph2sei403@ocaepc24:~$ python2.7 dechif_new_cle.py
Plaintext : 756e64206c696b65
fin de la routine
```

Le message déchiffré est alors, dans l'ordre de l'envoi des bits (donc dans l'ordre inverse) `756e64206c696b65`. Il s'agit bien du bon résultat !

Une fois la trame reçue, il faut envoyer une trame avec des zéros pour mettre le signal start_stop à zéro et permettre à l'IP de repasser à l'état initial :

```
xph2sei403@ocaepc24:~$ python2.7 zero.py
fin de la routine
```

On peut ensuite lancer d'autres tests.
On peut observer les trames envoyées et reçues par l'ordinateur sur un oscilloscope (nous avons utilisé l'oscilloscope **Tektronix MSO4014B**) :

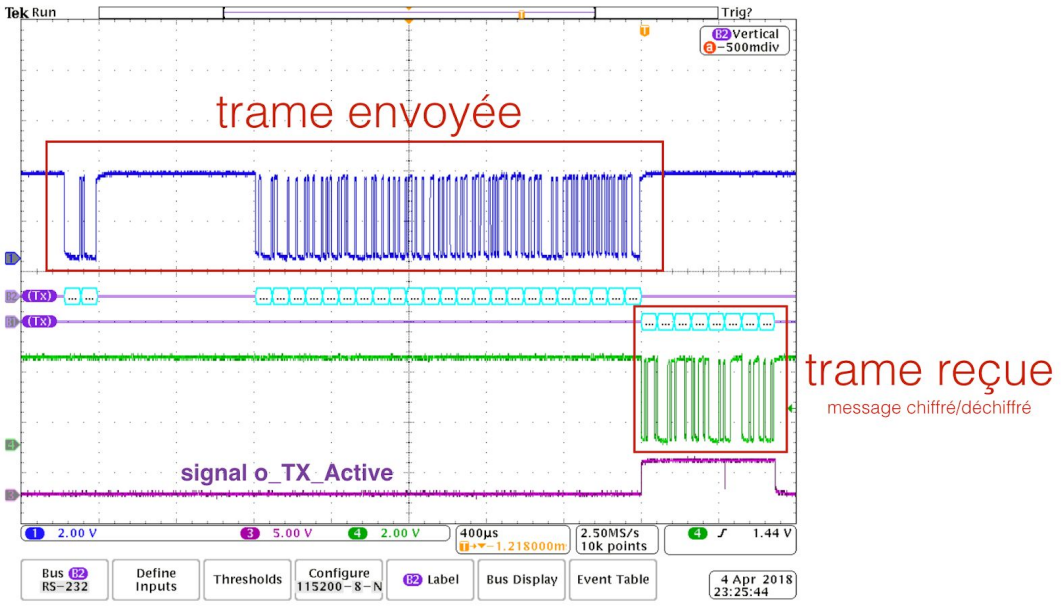


Figure 8 : observation des trames sur oscilloscope

On remarque que la trame envoyée comporte un temps mort après les deux premier paquets. Nous n'avons pas réussi à déterminer la cause de ce temps mort. Il n'occasionne pas de dysfonctionnements car l'UART attend toujours le signal de start pour recevoir les informations.
On note bien que le signal o_TX_Active reste à l'état haut pendant le chiffrement/déchiffrement.

On peut lire les valeurs des octets sur les bus dédiés que l'on a préalablement rattaché aux signaux d'entrée ***RX_Serial*** et de sortie ***o_TX_Serial***, ici dans le cas d'un déchiffrement avec la même clé :

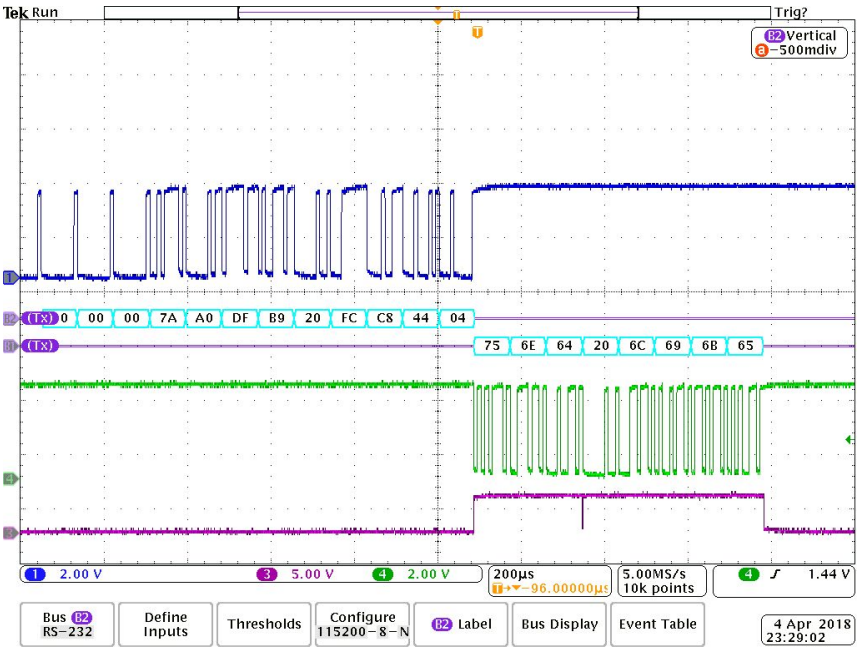


Figure 9 : observation des trames sur oscilloscope
Ici dans le cas d'un déchiffrement avec la même clé

On note bien que le résultat est juste ! Le message déchiffré est, dans l'ordre d'envoi : `756e64206c696b65.` Bien sûr, cela ne remplace pas une démonstration!

Nous avons également testé des vecteurs aléatoires, en vérifiant qu'après un chiffrement et un déchiffrement, on retrouve bien le message de départ!

Conclusion

A l'issue de ce projet, nous avons réalisé un générateur de crypto-processeurs Simon pouvant générer jusqu'à soixante versions différents par leurs tailles de message et de clé, ainsi que par leurs options de conception. Nous avons également pu implémenter l'un de ces crypto-processeurs générés sur une carte FPGA et s'assurer du bon fonctionnement du circuit.

Ce projet fut l'occasion d'approfondir nos connaissances du flot de conception microélectronique numérique. Multidisciplinaire, alliant la conception numérique et l'informatique, il fut agréable à mener, même si pas exempt de difficultés !

Bibliographie

Chiffrement Simon :

Papiers officiel de la NSA:
https://eprint.iacr.org/2013/404.pdf *-- contient notamment les vecteurs de test*
http://csrc.nist.gov/groups/ST/lwc-workshop2015/papers/session1-shors-paper.pdf *-- contient notamment les caractéristiques du crypto-processeurs pour différentes implémentations (pipline, itératif) et différentes tailles de clé et de messages*

www.wikipedia.fr

Description du chiffrement symétrique :
https://blog.devensys.com/chiffrement-symetrique/

Site de l'UART : www.nandland.com