Université de Montréal

Design et implémentation sur FPGA d'un algorithme DES

Par

Mohamed AMOUD

Département d'Informatique et de Recherche Opérationnelle

Faculté des études supérieures

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maîtrise ès Sciences
en Informatique

Décembre, 2008

Université de Montréal

Faculté des études supérieures

Ce mémoire intitulé:

Design et implémentation sur FPGA d'un algorithme DES

présenté par :

Mohamed AMOUD

a été évalué par un jury composé des personnes suivantes :

Claude FRASSON
président-rapporteur

Elmostapha ABOULHAMID
directeur de recherche

Philippe LANGLAIS
membre du jury

Résumé

Dans ce mémoire, nous présentons l'optimisation d'une implémentation matérielle de l'algorithme cryptographique DES (*Data Encryption Standard*) sur une plate-forme reconfigurable basée sur FPGA (*Field Programmable Gate Arrays*). Notre conception a été implémentée sur une *Spartan III XC2V2000.* L'architecture du design a été décrite dans le langage VHDL. En vue de réduire le chemin critique du design associé, nous avons utilisé une ***approche pipeline*** avec l'accent sur la non-linéarité des S-boxes de DES, et ce dans l'objectif d'atteindre une performance *haut débit.* En suivant cette voie, nous allons favoriser une architecture optimisée pour la gestion de la rapidité. Notre implémentation réalise une fréquence de 92.3 MHz. Par conséquent, le débit (*throughput*) atteint est de 5907,2 Mbits/s.
Ces résultats sont tout à fait compétitifs par rapport à d'autres implémentations matérielles de DES.

Mots clés: FPGA, DES, Pipeline, Implémentation matérielle, VHDL, Cryptographie.

Abstract

In this paper, we present an efficient and compact optimization of a hardware implementation of the cryptographic algorithm DES (*Data Encryption Standard*) on a reconfigurable platform based on FPGA (*Field Programmable Gate Arrays*). Our design was implemented on a Spartan III XC2V2000 device.

The architecture design has been described in VHDL. As a strategy to reduce the associated design critical path, we used a pipeline structure with emphasis on non-linearity of S-boxes of DES which correspond to a large case in VHDL, and will be implemented in look-up tables in the FPGA.

Our implementation achieved a frequency of **92.3** MHz. Therefore, the speed (*throughput*) is reached **5907.2** Mbits /s.

These results are quite competitive when compared with other reported reconfigurable hardware implementations of DES.

Key Words: FPGA, DES, Pipeline, Hardware Implementation , VHDL, Cryptography.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

REMERCIEMENTS

La seule manière d'être juste envers tous ceux qui m'ont aidé pendant la période de mon mémoire est de les remercier à tous, sans oublier personne.
Or, je risque d'oublier de citer quelques uns.
Donc, la seule manière que j'ai trouvée d'être équitable est de remercier à Dieu de m'avoir permis de croiser avec autant de gens prêtes à m'aider sans demander de retour.
Je remercie *Elmostapha ABOULHAMID*, mon directeur de recherche, professeur à l'Université de Montréal, pour m'avoir fait confiance et pour avoir encadré mes travaux de recherche. Ses conseils et son soutien m'ont été extrêmement précieux.
Je remercie également le président-rappoteur de ce mémoire, *Claude FRASSON* et le membre de jury, *Philippe LANGLAIS*, pour avoir lu et commenté ce manuscrit.
J'adresse mes remerciements aux étudiants de LASSO qui m'ont épaulé pendant cette période. Merci en particulier à *Mazen EZZEDDINE*, *Amine ANANE* et *Mathieu DUBOI*, pour leur soutien et leur gentillesse.
Je souhaiterais remercier mes amis au Canada, *Hamid Ouchrif*, *Elfilali*, *Naoufel* et *Brahim*, pour leur fidélité.
Pour terminer, un gros bisou à *Adam* (mon ptit amazigho_canadien), un grand merci à la famille *YOUSFI*, la famille *ZRAOUTI* ainsi qu'à ma chère petite famille, en particulier à *Badreddine*, *Samira*, *Malika*, *Fatima*, et ma mère *Rkia*, qui m'ont toujours soutenu.
Sans votre soutien je n'aurais pas réussi cette épreuve.
Mon plus grand, sincère et inconditionnel merci à vous tous ! ! !

Chapitre 1

INTRODUCTION

1.1 Motivations

Dans un contexte d'ouverture, de mobilité et d'ubiquité croissante, la sécurité des systèmes de communication et d'information va reposer sur des architectures complexes associant logiciels et matériels. Afin de ne pas devenir un frein au développement des systèmes d'informations, des algorithmes cryptographiques sécurisés et rapides ont été développés pour combattre des menaces de sécurité, ainsi que les composants matériels utilisés ont pris en compte la problématique de la sécurité dès les premières phases de la conception.
En outre, la diversité élevée vue sur des applications de sécurité a posé un défi supplémentaire puisque des algorithmes hautement sécurisés n'étaient pas la seule exigence, mais plutôt, exigence de performance pour certaines applications et pour d'autres, moins d'espace. Dans ce scénario, des designers cryptographiques ont exploré non seulement des réalisations sur des plates-formes logicielles, mais aussi sur le matériel classique ou des plates-formes matérielles reconfigurables.

Ce mémoire traite le design et l'implémentation matérielle de l'algorithme DES (*Data Encryption Standard*) sur une plate-forme reconfigurable basée sur FPGA (*Field Programmable Gate Arrays*) *Spartan III XC2V2000*, utilisant *Xilinx ISE* comme outil de synthèse, *ModelSim* pour la simulation, et *Xilinx ChipScope Pro* comme analyseur logique pour le déboggage. L'architecture du design a été décrite dans le langage VHDL. Dans le but de réduire le chemin critique du design associé, nous avons utilisé une ***approche pipeline*** qui nous permet d'obtenir des performances compétitives. Cette approche sera étudiée avec son efficacité sur le DES en mettant l'accent sur les S-Boxes à haut débit et reconfigurables.

Le choix des paramètres de travail (*implémentation matérielle, FPGA, algorithme DES*) n'était pas aléatoire. En effet, nous pouvons éclaircir les critères principaux de notre choix comme suit:

- **Implémentation matérielle *versus* implémentation logicielle**

Les avantages d'une implémentation logicielle comprennent la facilité d'usage, la facilité de mise à niveau et la portabilité. Toutefois, une implémentation logicielle offre peu de sécurité physique, notamment en matière de stockage des clés [5]. En revanche, des algorithmes cryptographiques (et leurs clés associées) qui sont implémentés dans le matériel sont, par nature, plus sécurisés physiquement comme ils ne peuvent pas facilement être lus ou modifiés par un attaquant extérieur.

En outre, nous pouvons identifier deux scénarios d'application où les implémentations matérielles sont avantageuses sur des implémentations logicielles. Premièrement, ce sont des applications cryptographiques haut-débit où un co-processeur effectue les opérations cryptographiques afin de soulager le reste du système. Deuxièmement, ce sont des applications où une faible puissance et les exigences de ressources sont strictes. Dans les deux scénarios d'application, le stockage sécurisé des clés est important.

D'ailleurs, l'implémentation matérielle est théoriquement plus rapide que l'implémentation logicielle, elle peut tirer profit de l'exécution parallèle. La figure 1.1 démontre cette différence : la routine logicielle a besoin de 12 cycles d'horloge pour calculer le résultat G; Cependant, dans le matériel il prend seulement 2 cycles d'horloge pour calculer le même résultat.

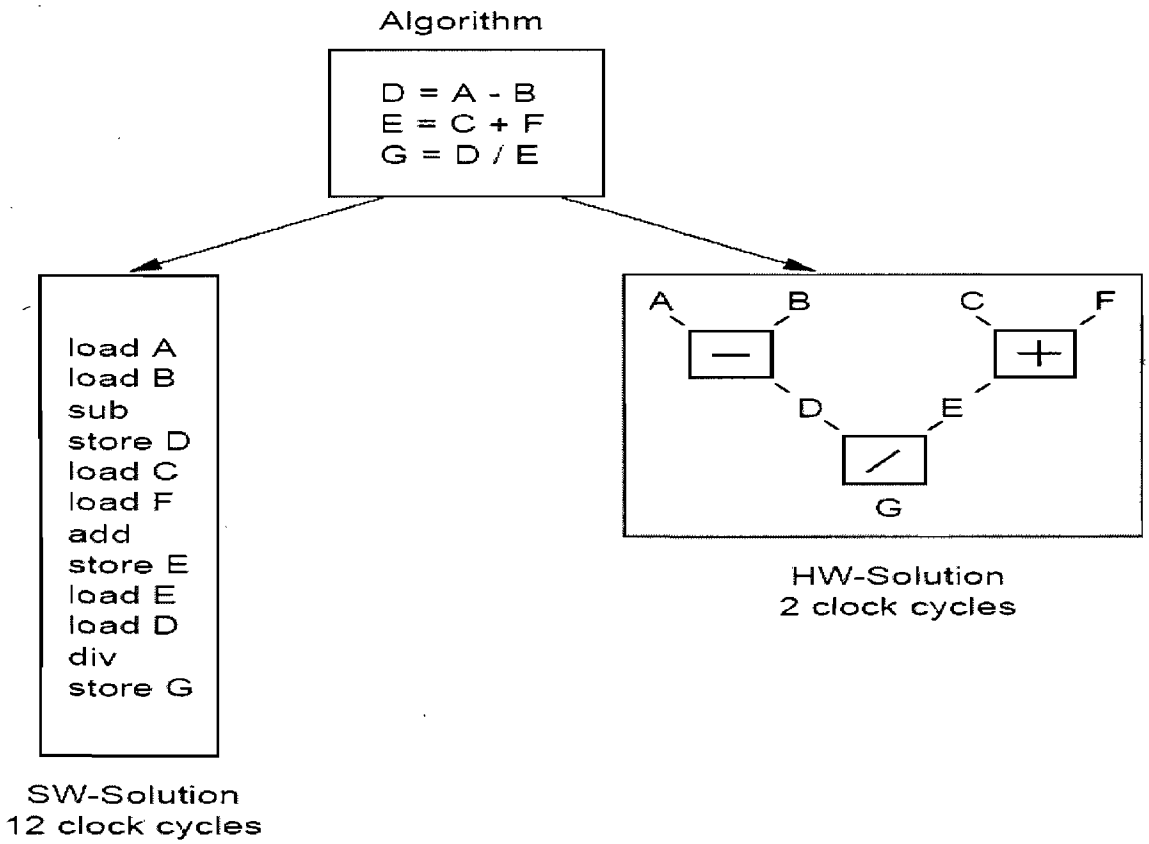

Figure 1.1: logiciel vs materiel [4]

- **FPGA *versus* ASIC?**

Le point faible des implémentations matérielles ASIC (*Application Specific Integrated Circuits*) est le manque de flexibilité en ce qui concerne la commutation des paramètres et de l'algorithme. FPGAs (*Field Programmable Gate Arrays*) sont des alternatives prometteuses pour l'implémentation de procédés de chiffrement à bloc. Elles offrent beaucoup d'avantages, par rapport aux ASICs (*Application Specific Integrated Circuits*): la haute flexibilité incluant la capacité des modifications fréquentes de matériel, le coût de développement bas et le bas prix du produit final. FPGAs sont des dispositifs matériels dont la fonction n'est pas fixée et qui peuvent être programmés en système. Les avantages potentiels des algorithmes de cryptage implémentés sur FPGAS incluent [2, 3, 14]:

Agilité d'Algorithme (*Algorithm Agility*): ce terme se réfère à la commutation d'algorithmes cryptographiques pendant l'opération de l'application ciblée. On peut constater que la majorité des protocoles de sécurité modernes, comme SSL ou IPSEC, sont des algorithmes indépendants et tiennent compte d'algorithmes de cryptage multiples. Tandis que l'agilité d'algorithme peut être très coûteuse avec le matériel traditionnel, FPGA peut être reprogrammé à la volée (*on-the-fly*). Bien que le temps de reconfiguration soit toujours une question ouverte à résoudre, l'agilité d'algorithme par FPGA semble être une possibilité attrayante.

Téléchargement d'algorithme (*Algorithm Upload*): de point de vue cryptographique, le *téléchargement d'algorithme* peut être nécessaire parce que, s'il y a une sorte de connexion à un réseau comme l'Internet, FPGAs peuvent télécharger le nouveau code de configuration. Par contre, la mise à jour des algorithmes implémentés sur l'ASIC est pratiquement infaisable si beaucoup de dispositifs sont affectés ou si les systèmes ne sont pas facilement accessibles, par exemple dans des satellites.

Modification d'Algorithme (*Algorithm Modification*): il y a des applications qui exigent la modification d'un algorithme standardisé, en utilisant des S-boîtes ou des permutations. Ces modifications sont facilement faites avec le matériel reconfigurable. De même un algorithme standardisé peut être échangé avec un propriétaire. Aussi, les modes de fonctionnement peuvent être facilement changés.

L'efficacité d'Architecture (*Architecture Efficiency*) : dans certains cas, une architecture matérielle peut être beaucoup plus efficace si elle est conçue pour un jeu spécifique de paramètres. Avec FPGA, il est possible de concevoir et optimiser une architecture pour un ensemble spécifique de paramètres.

Débit (*Throughput*): les processeurs ne sont pas optimisés pour l'exécution rapide particulièrement dans le cas d'algorithmes à clé-publique. Principalement parce qu'ils manquent

d'instructions pour des opérations arithmétiques modulaires sur de longues opérandes. En général, les implémentations FPGA ont le potentiel d'exécution considérablement plus rapide que des implémentations logicielles.

Efficacité de Coût (*Cost Efficiency*) : les coûts, pour développer une implémentation d'un algorithme donné sur FPGA, sont beaucoup plus bas que pour une implémentation ASIC, parce que l'on est en réalité capable d'utiliser la structure de données du FPGA (LUT : *Look-Up Table*), et l'on peut évaluer et tester la puce reconfigurée pour des temps infinis et sans autres coûts.

- **Pourquoi l'algorithme DES ?**

L'algorithme DES contient les S-Boxes qui utilisent des ressources logiques, sur lesquelles repose la grande sécurité, étant non linéaires très efficaces pour diluer les informations. Par conséquent, nous pourrons appliquer la même approche sur les algorithmes successeurs de DES tels que : AES, 3DES,.., tant qu'ils contiennent les S-Boxes.

Quant à la performance, les algorithmes symétriques, en général, nécessitent d'une capacité de calcul moins intensive que les algorithmes asymétriques [5]. Cela occasionne une rapidité de cryptage et de décryptage atteignant centaines ou milliers de fois supérieure à celle des algorithmes à clé publique.

1.2 Plan du mémoire

Notre mémoire est organisé de la façon suivante :

Le Chapitre 2 amène une discussion au sujet des techniques de cryptographie modernes, tout en focalisant sur l'algorithme de cryptographie DES ; Étant donné que le sujet de ce mémoire concerne l'algorithme DES et son implémentation sur FPGA.

Le chapitre 3 présente une discussion à propos de l'état de l'art des performances des expérimentations antérieures, ainsi que des différentes méthodes d'optimisation matérielles et logicielles pour l'architecture DES.

Le Chapitre 4 discute la méthodologie de travail. Il propose un flot de conception pour le développement d'application de l'algorithme DES implémenté sur FPGA. Il met l'accent sur les différents outils utilisés pour créer le logiciel et le matériel.

Le chapitre 5 montre comment l'architecture du système est implémentée ainsi que les résultats expérimentaux compilés au cours de ce travail de recherche.

Avec le chapitre 6, nous terminons ce rapport par une conclusion et la présentation des perspectives que nous voyons pour la poursuite de ce travail.

Chapitre 2

ALGORITHME CRYPTOGRAPHIQUE *DES*

Le domaine de la cryptographie, longtemps resté un domaine réservé aux services des gouvernements et des services d'espionnage et contre-espionnage des militaires, est devenu depuis les années 70 un domaine de recherche grand public porté par de nombreux laboratoires académiques.

Le but de ce chapitre est de donner un aperçu des techniques de cryptographie modernes, tout en focalisant sur l'algorithme de cryptographie DES, étant donné que le sujet de ce mémoire concerne l'algorithme DES et son implémentation sur FPGA.

2.1. Introduction à la cryptographie

Le développement considérable des réseaux de communication a favorisé l'expansion de nouveaux moyens de paiement à distance avec notamment la carte à puce, le e-commerce, et le e-banking. Ces nouveaux types de communications nécessitent de plus en plus de moyens de transaction dits sûrs, pouvant résister aux diverses attaques cryptanalystiques. De cette lutte effrénée entre d'une part les cryptographes qui mettent en place des systèmes ou algorithmes cryptographiques, et d'autre part les cryptanalystes qui développent des techniques pour briser les systèmes de sécurité.

2.1.1. La cryptographie

La cryptographie est l'étude des différentes méthodes et techniques mathématiques reliées aux aspects de sécurité de l'information, pour assurer le secret et l'authenticité des messages, elle

permet de stocker des informations sensibles ou de les transmettre à travers des réseaux non sûrs (comme Internet) de telle sorte qu'elles ne peuvent être lues par personne à l'exception du destinataire convenu. Elle concerne la transformation d'un message (texte, image, chiffres) intelligible vers un message codé, incompréhensible à tous sauf pour les détenteurs d'un code de déchiffrement.

Le terme "cryptographie" vient du grec "*kriptós*" (caché) et "*gráphein*" (écrite). C'est un ensemble de techniques qui fournit la sécurité de l'information.

Par ailleurs, La cryptographie est une discipline ancienne. Déjà dans l'antiquité, les Grecs avaient inventé des méthodes pour chiffrer les messages. L'une d'entre elles, datant du VIème siècle avant J.C., consistait à enrouler une bande de papier autour d'un cylindre, puis à écrire le message sur la bande. Une fois déroulé, le papier était envoyé au destinataire qui, dès lors qu'il possédait le diamètre du cylindre, pouvait déchiffrer le message.

Pendant de nombreuses années, la cryptographie était exclusivement réservée au domaine militaire et diplomatique. La littérature sur le sujet était donc très peu abondante.

Crypter est un processus de transformation, et décrypter est son inverse. Les fonctions de cryptage sont des algorithmes cryptographiques [11]. Le code, permettant de chiffrer et déchiffrer un message codé, se pose sur une clé cryptographique. Cette clé est un paramètre qui doit être variable et maintenu secret, sauf dans certains algorithmes où une partie de la clé reste exposée. Si auparavant toute la sécurité d'un système cryptographique résidait dans la clé, aujourd'hui il existe d'autres éléments qui composent la robustesse de tels systèmes. Plusieurs algorithmes, dont ceux de hachage, protocoles d'authentification et procédures d'opérations de cryptage et décryptage aident à assurer la fiabilité des systèmes de sécurité.
La cryptographie a pour principales fonctions les quatre points suivants :
1. **Confidentialité** : seulement le destinataire autorisé doit être capable d'extraire le contenu du message de son état crypté. Par ailleurs, l'obtention de l'information à propos du contenu du message (comme par exemple la distribution statistique des caractères) ne doit pas être possible.
2. **Intégrité** : par l'analyse d'intégrité le destinataire peut déterminer si le message a été modifié pendant la transmission.

3. **Authentification** : le destinataire doit avoir la capacité d'identifier l'auteur du message, aussi bien que de savoir si c'est l'auteur présumé qui a en effet envoyé le message.
4. **Non-reniement** : l'expéditeur du message ne doit pas pouvoir nier que c'est lui l'auteur.
On distingue deux grands types d'algorithmes de chiffrement, les algorithmes à clef secrète et les algorithmes à clef publique. Chacune de ces deux classes possède ses propres avantages et inconvénients. Les systèmes à clef secrète nécessitent le partage d'un secret entre les interlocuteurs.

La découverte en 1976 des systèmes à clef publique a permis de s'affranchir de cette contrainte, mais elle n'a pas apporté autant de solution parfaite, dans la mesure où tous les algorithmes de chiffrement à clef publique, par leur lenteur, ne permettent pas le chiffrement en ligne. Dans la plupart des applications actuelles, la meilleure solution consiste à utiliser un système hybride, qui combine les deux types d'algorithmes.

2.1.2. Cryptographie à Clé Symétrique

Les algorithmes à clé symétrique concernent les méthodes de cryptage dans lesquelles l'expéditeur et le récepteur d'un message utilisent la même clé, laquelle a été transmise via un canal sécurisé. Ceci était la seule méthode de cryptage connue jusqu'à 1976.
En effet, la plupart des algorithmes symétriques utilisent des clés identiques, mais certains algorithmes utilisent des clés de cryptage et de décryptage différentes, mais la deuxième peut être calculée de façon simple à partir de la première et vice-versa. Ces systèmes sont aussi connus comme étant à clé secrète, à clé privée ou à clé unique.
Le fonctionnement de ce genre d'algorithme dépend du secret de la clé [9], et par conséquent, présume l'existence d'un canal sécurisé pour le transfert de la clé, à défaut de quoi la sécurité de tout système est compromise. Le cryptage et le décryptage d'un algorithme cryptographique à clé privée est donné par :

$$C = E_k(M)$$
$$M = D_k(C)$$

E est la fonction d'encryption,
D la fonction de décryption,
C le texte chiffré et ***M*** le texte en clair (message),
K la clé.

La figure 2.1 illustre ce procédé.

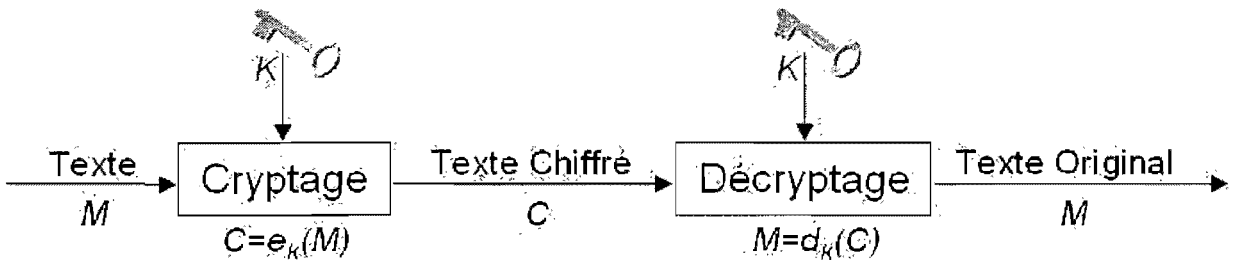

Figure 2.1 : Schéma de processus d'encryption et décryption [6]

Catégories :

Les algorithmes symétriques sont divisés en deux catégories principales. Quelques uns opèrent sur un seul bit (ou octet) à chaque fois. Il s'agit des algorithmes de cryptage par flot. D'autres agissent sur le texte en clair par des ensembles de bits. Ces ensembles sont appelés blocs, d'où la dénomination cryptage par bloc. Ces blocs peuvent varier de 64 à 128bits, voire plus.

Principe de fonctionnement :

La plupart des algorithmes symétriques modernes dérive d'une méthode attribuée à *Horst Feistel*, utilisant des itérations répétées : **rondes** (*rounds*). Une ronde est la permutation des bits du bloc qui vont être cryptés et qui sont mixés avec quelques bits de la clé. Soit B le bloc de bits qui sera crypté. B est divisé en deux parties P_1 et P_2. Pendant que P_1 reste inchangée, P_2 est additionnée (où 'XORé') à une fonction de hachage à sens unique appliquée à P_1. Les deux résultats sont alors échangés à nouveau : $P_1, P_2 \rightarrow P'_2, P_1$; de telle façon que :

$$P'_2 = P_2 + f(P_1, k)$$

Étant donné que la sortie de l'itération accède encore à la valeur P_1, et que l'addition est une opération réversible, la ronde peut être défaite, quelque soit la fonction *f*. Même si une ronde seule consiste en une opération non sécurisée, la répétition des rondes avec des sous-clés différentes incrémente de façon considérable la sécurité du système. Pour décrypter il suffit d'appliquer les itérations dans le sens inverse, avec les sous-clés aussi dans le reverse.

L'exemple classique d'un algorithme à clé privée utilisant cette méthode est le **DES** (*Data Encryption Standard*).

Pourtant même les algorithmes qui n'utilisent pas la structure de Feistel utilisent des rondes afin de promouvoir la diffusion et la confusion, comme c'est le cas pour **AES** (*Advanced Encryption Standard*).

Performance :

Par rapport à la performance, les algorithmes symétriques nécessitent une capacité de calcul moins importante que les algorithmes asymétriques. Cela occasionne une rapidité de cryptage et de décryptage des centaines ou milliers de fois supérieure à celle des algorithmes à clé publique.

Pourtant les algorithmes symétriques ont le désavantage de nécessiter le partage de la clé secrète entre les deux points qui veulent communiquer. Par exemple, dans une population de *n* individus, pour assurer une communication entre chacun des membres, il est nécessaire d'avoir $n * (n-1)/2$ clés.

Actuellement, l'échange de clés secrètes se donne à travers un protocole d'échange basé sur un algorithme de cryptographie asymétrique.

Exemples :

- **DES** (*Data Encryption Standard*)

Le système de chiffrement à clef secrète le plus célèbre et le plus utilisé a été adopté comme standard américain en 1977 (standard FIPS 462) pour les communications commerciales, puis par l'ANSI en 1991. Le DES opère sur des blocs de 64 bits et utilise une clef secrète de 56 bits.

La plupart des applications l'utilisent maintenant sous la forme d'un triple DES à deux clefs, constitué de trois chiffrements DES successifs avec deux clefs secrètes [34].

Cette technique permet de doubler la taille de la clef secrète (112 bits). Plus précisément, pour chiffrer avec le triple DES, on effectue d'abord un chiffrement DES paramétré par une première clef de 56 bits, puis un déchiffrement DES paramétré par une seconde clef, et à nouveau un chiffrement DES avec la première clef. Seules deux clefs sont utilisées dans la mesure où l'emploi de trois clefs secrètes différentes ne permet pas d'accroître la sécurité de l'algorithme.

- **AES** (*Advanced Encryption Standard*)

Il est le nouveau standard de chiffrement à clef secrète. Il a été choisi en octobre 2000 parmi les 15 systèmes proposés en réponse à l'appel d'offre lancé par le NIST (*National Institute of Standards and Technology*). Cet algorithme, initialement appelé RIJNDAEL, a été conçu par deux cryptographes belges, V. Rijmen et J. Daemen. Il opère sur des blocs de message de 128 bits et est disponible pour trois tailles de clef différentes : 128, 192 et 256 bits.

Comme pour la plupart des algorithmes par blocs, le processus de chiffrement de l'AES consiste à itérer une permutation paramétrée par une valeur secrète, appelée sous-clef, qui change à chaque itération. Les différentes sous-clefs sont dérivées de la clef secrète par un algorithme de cadencement de clef. Pour une clef de 128 bits, l'AES effectue 10 itérations de la fonction décrite à la figure 2.2, chacune des sous-clefs comportant également 128 bits. La première itération est précédée d'un XOR bit-à-bit entre le message clair et la sous-clef numéro 0. De même, la dernière itération est légèrement différente des itérations précédentes.

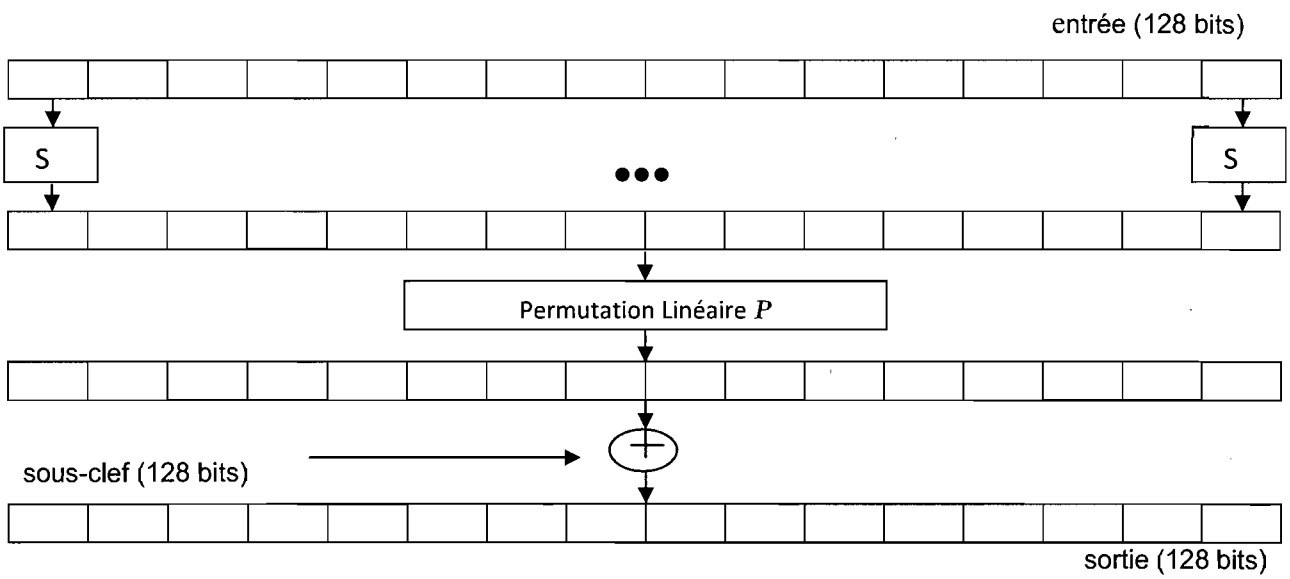

Figure 2.2 : Schéma d'une itération de l'AES

La fonction itérée se décompose elle-même en trois étapes, conformément aux principes fondamentaux de confusion et de diffusion énoncés par *Shannon*. La première étape, dite de confusion, consiste à appliquer à chacun des 16 octets de l'entrée une même permutation *S*.
Cette fonction correspond à la fonction inverse dans le corps fini à 2^8 éléments (dans la pratique, elle est mise en table). Elle assure la résistance de l'algorithme aux attaques classiques (cryptanalyse différentielle, cryptanalyse linéaire ...). Ensuite, lors de la phase de diffusion, on permute les bits du mot obtenu suivant une fonction P qui est également composée d'opérations simples sur le corps à 2^8 éléments, on effectue un XOR bit-à-bit entre le résultat et la sous-clef de l'itération.
Les sous-clefs de 128 bits, numérotées de 0 à 10, sont dérivées de la clef secrète de la manière suivante : la sous-clef numéro 0 correspond à la clef secrète ; ensuite, la sous-clef numéro ***i*** (utilisée à la $i^{ème}$ itération) est obtenue à partir de la sous-clef numéro (i-1) grâce à l'algorithme décrit à la figure 2.3.

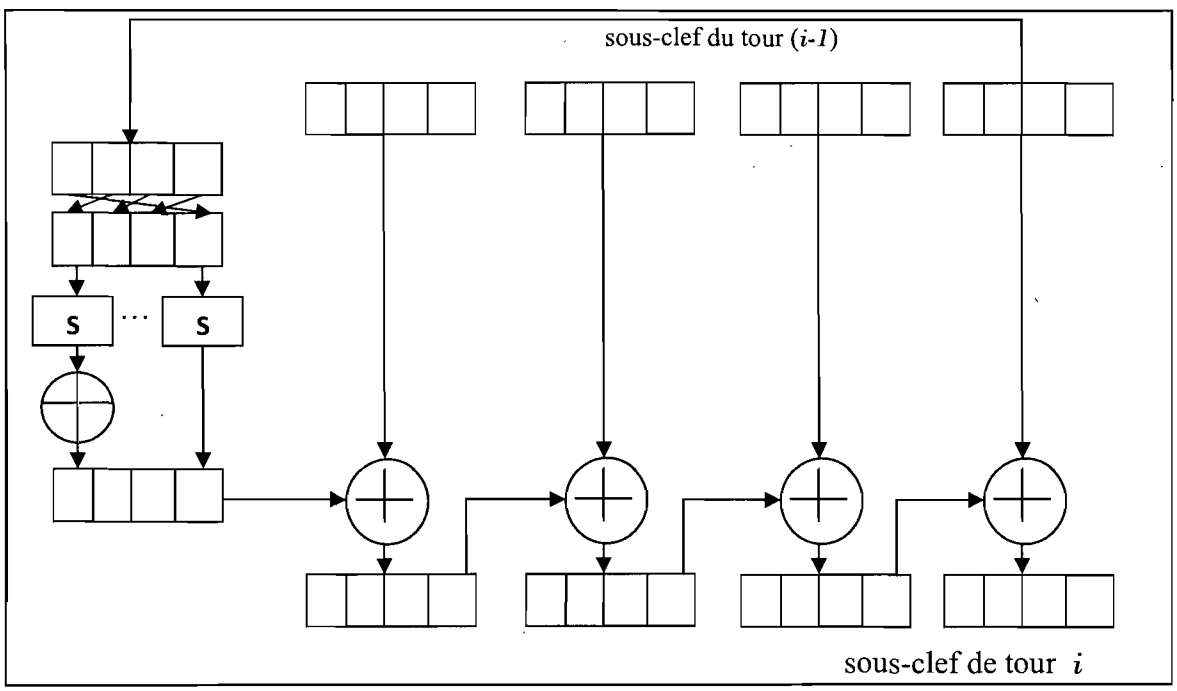

Figure 2.3 : Algorithme de cadencement de clef de l'AES [35]

On permute les quatre derniers octets de la clef numéro (*i - 1*), puis on leur applique la fonction *S*. Après avoir ajouté une constante (dépendant de i) au premier octet, on effectue un **ou exclusif** bit-à-bit entre les quatre octets ainsi obtenus et les quatre premiers octets de la sous-clef précédente. Les trois autres blocs de quatre octets de la clef numéro i sont, simplement, le résultat d'un ou exclusif (XOR) entre le bloc correspondant de la sous-clef (i - 1) et le bloc précédent de la sous-clef i.
Le fait que l'AES soit uniquement composé d'opérations simples sur les octets le rend extrêmement rapide, à la fois pour les processeurs 8 bits utilisés dans les cartes à puce et pour les implémentations tant logicielles que matérielles.

2.1.3. Cryptographie à Clé Asymétrique

Le principal inconvénient des algorithmes privés c'est qu'ils utilisent la même clé pour crypter et pour décrypter. Cela implique la nécessité d'un canal de communication sûr entre l'expéditeur et récepteur du message. L'algorithme est donc vulnérable durant la phase de transport de la clé.
En 1976 a été publié un article proposant un schéma de cryptographie faisant usage de deux clés distinctes (pourtant intrinsèquement liées) : l'une pour crypter et l'autre pour décrypter. Même s'il y a un rapport entre les clés, la possession d'une clé ne permet pas de calculer l'autre.
Aussi appelés algorithmes à clé publique, son nom provient du fait qu'une des clés générées doit être effectivement rendue publique : n'importe qui peut s'emparer de la clé publique et crypter un message, mais seulement une personne spécifique peut le décrypter. Dans ces systèmes, la clé publique est la clé de cryptage, et la clé privée est la clé de décryptage. Le procédé de cryptage est dénoté:

$$C = E\,k_{pu}\,(M)$$
$$M = D\,k_{pr}\,(C)$$

E est la fonction d'encryption,
D la fonction de décryption,
C le texte chiffré et ***M*** le texte en clair,
K_{pu} la clé publique, K_{pr} la clé privée.

La figure 2.4 illustre ce procédé.

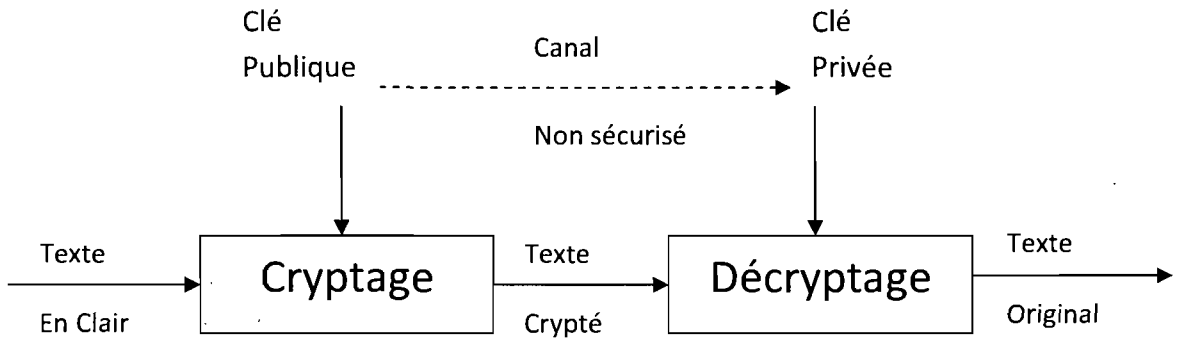

Figure 2.4: Schéma de processus d'encryption et décryption

Fonctionnement :

Du point de vue conceptuel, un algorithme à clé publique peut être imaginé comme une fonction à sens unique avec une trappe. La fonction $E\ k_{pu}$ doit être facile à appliquer. Pourtant le décryptage ne doit être possible que pour la personne ayant la clé privée $\boldsymbol{k_{pr}}$ pour utiliser la fonction $D\ k_{pr}$.

Dans le contexte de la cryptographie, il est désirable que $E\ k_{pu}$ soit une fonction à sens unique injective, afin que le décryptage puisse avoir lieu. Toutefois, il n'existe que des fonctions injectives *supposées* être à sens unique. Par exemple, en supposant que n soit le produit de deux grands nombres premiers p et q, et que b soit un entier naturel supérieur à 2, il est possible de définir $\boldsymbol{f: Zn \rightarrow Zn}$ par :

$$\boldsymbol{f(M) = [M^b]n}$$

Considérant la nécessité d'une trappe pour pouvoir décrypter le message, dans le cas proposé cette fonction est l'inverse f_{-1}, tel que $\boldsymbol{f_{-1}(M) = [M^a]n}$, pour une valeur correcte de a. La trappe ici est une méthode pour retrouver a en connaissant b, qui utilise la factorisation de n.

Sécurité :

Le système cryptographique à clé publique ne peut jamais être considéré comme étant inconditionnellement sûr, car un attaquant qui observe un texte crypté C peut crypter chaque texte clair possible M avec la règle de cryptage $E\ k_{pu}$ jusqu'à ce qu'il trouve l'unique X, tel que $C = E\ k_{pu}\ (M)$. Ce M est le résultat du décryptage de C. Donc la seule sécurité possible dans ce genre d'algorithme est la sécurité calculatoire. Pourtant il est largement utilisé, parce que les algorithmes à clé publique permettent d'accomplir les buts de la cryptographie en ce qui concerne l'authentification et la non-répudiation, à travers des protocoles de signature numérique.

Exemple :

RSA : c'est le système à clef publique le plus utilisé. RSA n'est pas, à proprement dit, un standard mais son utilisation est décrite et recommandée dans un grand nombre de standards officiels, son fonctionnement repose sur des résultats classiques d'arithmétique.

Dans toute la suite, pour deux entiers a et n, la notation ***a mod n*** désigne le reste de la division euclidienne de a par n. Considérons un entier n formé par le produit de deux nombres premiers p et q. D'après le théorème d'Euler [12], si a est un entier tel que ***a mod (p - 1)(q - 1) = 1***, alors, pour tout entier non nul x < n, on a:

$$X^a \bmod n = X$$

Le principe de RSA est alors le suivant : la clef publique d'un utilisateur est formée d'un nombre n produit de deux nombres premiers p et q, et d'un entier e premier avec *(p-1)(q-1)*.

Les valeurs de *n* et *e* sont publiées dans un annuaire.

La clef secrète correspondant est un entier *d* qui vérifie ***ed mod (p - 1)(q - 1) = 1***. Il est très facile de trouver un tel nombre d à partir de e, p et q. En effet, par hypothèse, l'entier *e* est premier avec (p - 1)(q - 1). D'après le théorème de *Bezout*, il existe donc deux entiers A et B non nuls tels que :

$$A\,(p-1)(q-1) + Be = pgcd((p-1)(q-1), e) = 1$$

La clef secrète d est donc l'entier positif correspondant au reste de B modulo (p - 1)(q - 1).

Dans RSA, les blocs de message sont représentés par des entiers compris entre 0 et n-1.

Pour envoyer un message m à Bob, Alice va donc chercher la clé publique de Bob et elle calcule le message chiffré c correspondant par : $c = m^e \bmod n$.

Lorsqu'il reçoit le chiffré c, Bob retrouve le texte clair en calculant $c^d \bmod n = m$.

En effet, par définition, e et d sont tels que ed ≡ 1 mod (p - 1)(q - 1). On a donc :

$$C^d \bmod n = (m^e)^d \bmod n = m^{ed} \bmod n = m$$

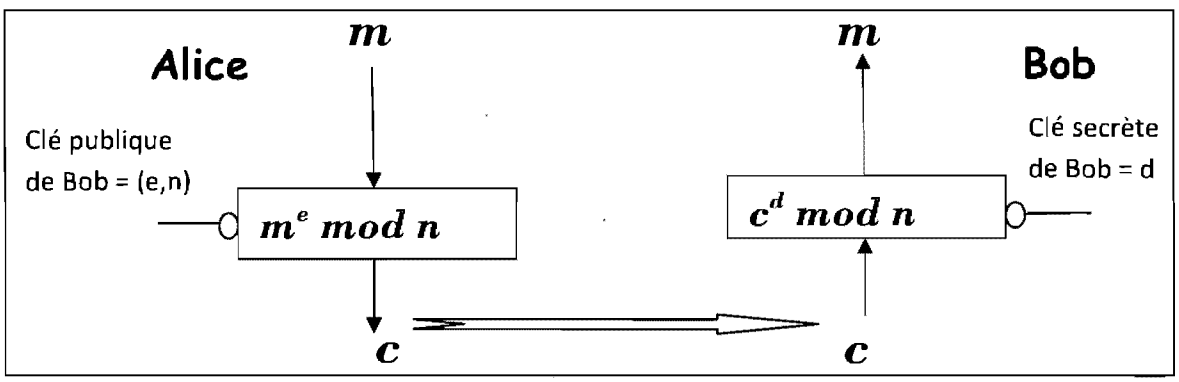

Figure 2.5 : Le chiffrement à clé publique RSA

Sécurité

Une attaque évidente à ce système consiste à tenter de factoriser *n*. Si cela est possible, il est trivial de calculer $(p - 1)(q - 1)$ et alors de calculer l'exposant de décryptage *d* à partir de *e*. Ainsi, pour que le système RSA soit sûr, il est nécessaire que *n* soit suffisamment grand pour que sa factorisation soit calculatoiremment impossible. Aujourd'hui la taille recommandée est au moins 1024-bits.

Donc, attaquer le système RSA consiste à retrouver le texte clair m à partir de la connaissance du chiffré $c = m^e \bmod n$ et de la clef publique *(e,n)*. Aucun algorithme efficace n'est connu à ce jour pour résoudre ce problème. La seule attaque générale connue pour décrypter RSA consiste à retrouver la clef secrète d à partir des valeurs publiques *(e, n)*. On peut démontrer que résoudre ce problème est équivalent à factoriser l'entier n.

L'algorithme RSA reste désormais comme un des algorithmes à clé publique les plus fiables et répandus. Pourtant, une nouvelle catégorie d'attaques, ne reposant plus seulement sur les aspects mathématiques des algorithmes de cryptographie, a vu le jour à la fin des années 90. Ce genre de cryptanalyse est spécialement intéressant dans le domaine de la microélectronique, car elle s'attaque aux implantations matérielles des systèmes cryptographiques.

2.2 Algorithme DES

Le *Data Encryption Standard* (DES) est un algorithme de cryptographie qui a été sélectionné comme un standard pour la *Federal Information Processing Standard* (FIPS) pour les États-Unis en 1976, mais qui a connu un succès international par la suite.

L'algorithme DES est symétrique de bloc, il prend une chaîne de caractères (ou nombres) d'une taille fixée à 64-bits du texte en clair et le transforme à l'aide d'opérations compliquées vers un texte crypté de la même taille. Cette transformation dépend d'une clé, et donc seulement celui qui connaît la clé est, à priori, capable de décrypter le message.

En apparence la clé est constituée de 64-bits ; en fait seulement 56 de ces 64-bits sont vraiment utilisés dans l'algorithme. Huit bits sont employés pour faire la vérification de parité.

La vue d'ensemble de la structure du algorithme DES est montrée dans la Figure 2.6. Il existe 16 étapes de calcul identiques, dites **rondes** (*rounds*). Il y a aussi des permutations, l'une initiale (*PI*) et l'autre finale (*PF*), où *PF* défait l'opération réalisée par *PI* et vice-versa. Il semblerait que *PI* et *PF* ne sont pas de signification cryptographique, mais qui ont été inclus dans l'algorithme pour faciliter le chargement de données dans le matériel des années 70. Avant les itérations principales, le bloc est divisé en deux moitiés de 32-bits chacune, qui sont calculées de façon alternée. Ce croisement est connu comme le schéma de Feistel.

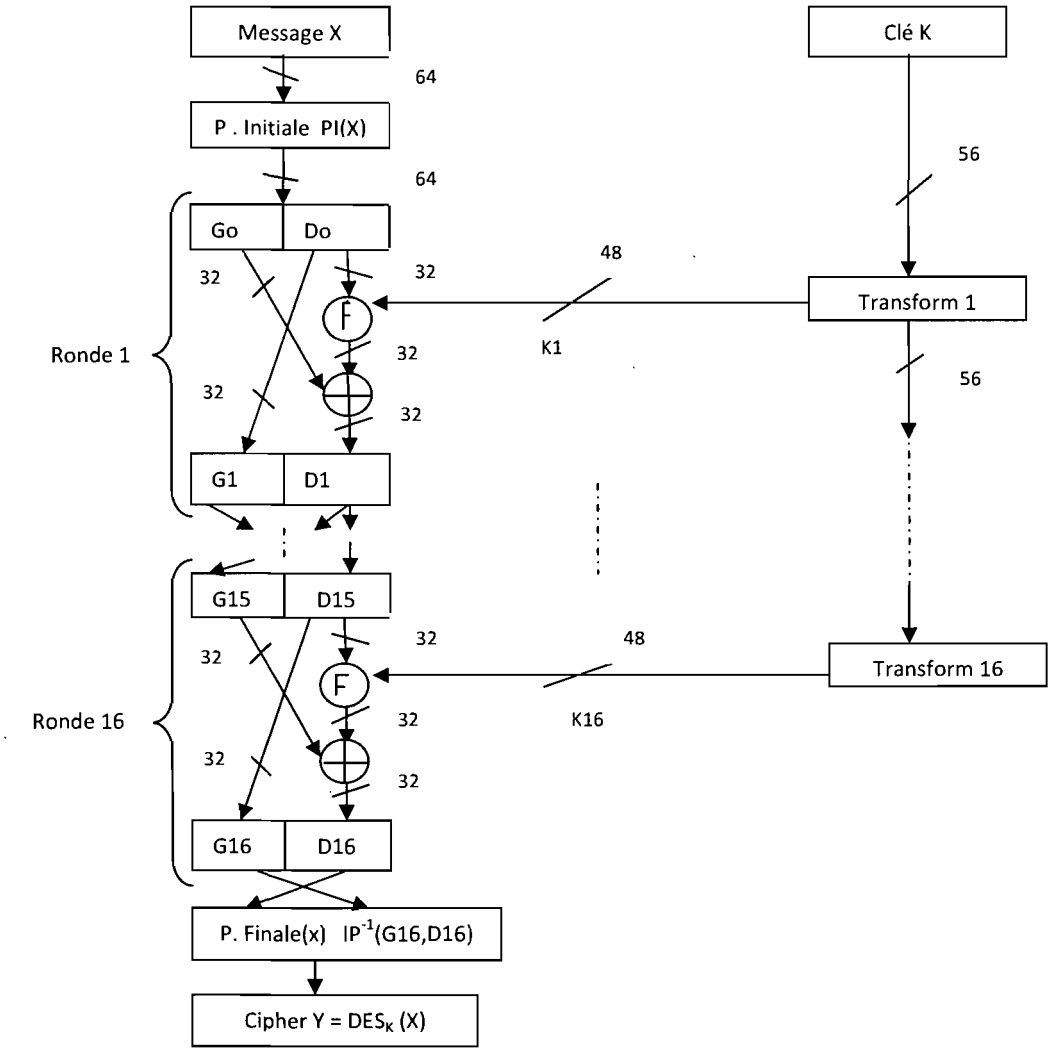

Figure 2.6: Schéma Feistel de l'Algorithme DES [9]

La structure de Feistel garanti que le cryptage et le décryptage sont des procédés similaires. La seule différence c'est que les sous-clés sont appliquées dans l'ordre inverse pendant le décryptage. Cela simplifie l'implantation, notamment en matériel, car il n'est point question de séparer les modules de cryptage et de décryptage.

Les étapes du processus de l'algorithme DES sont comme suit:

- **Permutation Initiale**

Dans un premier temps, chaque bit d'un bloc est soumis à la permutation initiale, pouvant être représentée par la matrice de permutation initiale (notée *PI*) suivante :

Tableau 2.1 : Matrice de permutation initiale de DES

PI	58	50	42	34	26	18	10	2
	60	52	44	36	28	20	12	4
	62	54	46	38	30	22	14	6
	64	56	48	40	32	24	16	8
	57	49	41	33	25	17	9	1
	59	51	43	35	27	19	11	3
	61	53	45	37	29	21	13	5
	63	55	47	39	31	23	15	7

Cette matrice de permutation indique, en parcourant la matrice de gauche à droite puis de haut en bas, que le $58^{ème}$ bit du bloc de texte de 64 bits se retrouve en première position, le $50^{ème}$ en seconde position et ainsi de suite.

- **Scindement en blocs de 32 bits**

Une fois la permutation initiale réalisée, le bloc de 64 bits est scindé en deux blocs de 32 bits, notés respectivement **G** et **D** (pour gauche et droite, la notation anglo-saxonne étant *L* et *R* pour *Left and Right*). On note $\mathbf{G}_0$ et $\mathbf{D}_0$ l'état initial de ces deux blocs :

Tableau 2.2 : Blocs G_0 et D_0 de 32 bits de DES

G_0	58	50	42	34	26	18	10	2
	60	52	44	36	28	20	12	4
	62	54	46	38	30	22	14	6
	64	56	48	40	32	24	16	8
D_0	57	49	41	33	25	17	9	1
	59	51	43	35	27	19	11	3
	61	53	45	37	29	21	13	5
	63	55	47	39	31	23	15	7

Il est intéressant de remarquer que $\mathbf{G_0}$ contient tous les bits possédant une position paire dans le message initial, tandis que $\mathbf{D_0}$ contient les bits de position impaire.

- **Rondes**

Les blocs $\mathbf{G_n}$ et $\mathbf{D_n}$ sont soumis à un ensemble de transformations itératives appelées ***rondes (fonction de Feistel)***, explicitées dans ce schéma (figure 2.7), et dont les détails sont donnés plus bas :

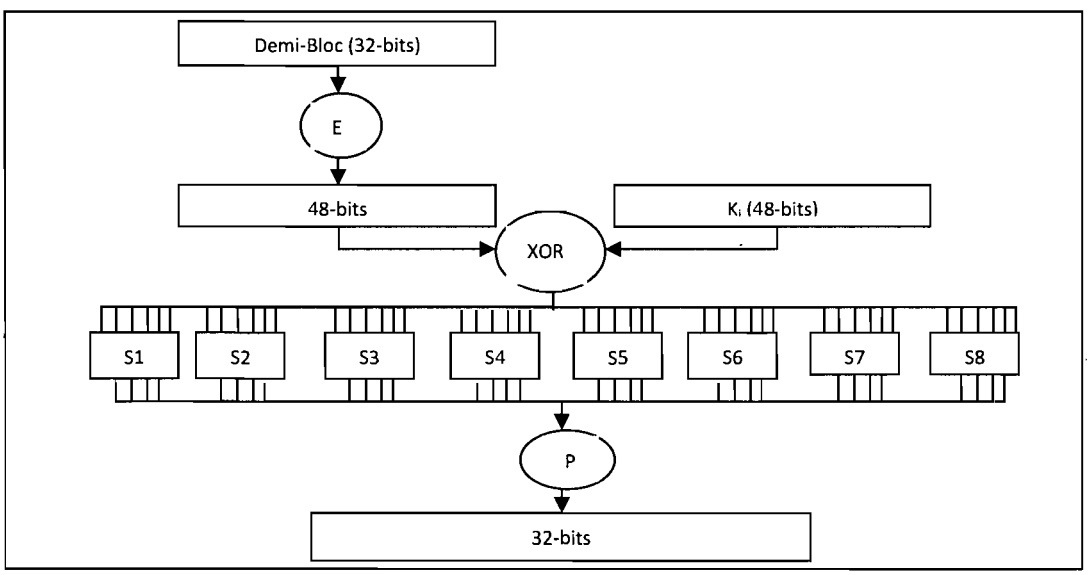

Figure 2.7 : La fonction Feistel -*ronde*- du DES

- **Fonction d'expansion**

Les 32 bits du bloc $\mathbf{D}_0$ sont étendus à 48 bits grâce à une table (matrice) appelé *table d'expansion* (notée **E**), dans laquelle les 48 bits sont mélangés et 16 d'entre eux sont dupliqués :

Tableau 2.3 : la table de la fonction d'expansion

E	32	1	2	3	4	5
	4	5	6	7	8	9
	8	9	10	11	12	13
	12	13	14	15	16	17
	16	17	18	19	20	21
	20	21	22	23	24	25
	24	25	26	27	28	29
	28	29	30	31	32	1

Ainsi, le dernier bit de $\mathbf{D}_0$ (c'est-à-dire le $7^{ème}$ bit du bloc d'origine) devient le premier, le premier devient le second, ...

De plus, les bits 1,4,5,8,9,12,13,16,17,20,21,24,25,28 et 29 de $\mathbf{D}_0$ (respectivement 57, 33, 25, l, 59, 35, 27, 3, 6l, 37, 29, 5, 63, 39, 31 et 7 du bloc d'origine) sont dupliqués et disséminés dans la matrice.

- ***« OU exclusif »* (*XOR*) avec la clé**

La matrice résultante de 48 bits est appelée $\mathbf{D'}_0$ ou bien $\mathbf{E}[\mathbf{D}_0]$. L'algorithme DES procède ensuite à un *OU exclusif* entre la première clé $\mathbf{K}_1$ et $\mathbf{E}[\mathbf{D}_0]$. Le résultat de ce *OU exclusif* est une matrice de 48 bits que nous appellerons $\mathbf{D}_0$ par commodité.

- **Fonction de substitution**

$\mathbf{D_0}$ est ensuite scindée en 8 blocs de 6 bits, notés $\mathbf{D_{0i}}$. Chacun de ces blocs passe par des **fonctions de sélection** (appelées parfois *boîtes de substitution* ou *fonctions de compression : S-Boxes*), notées généralement $\mathbf{S_i}$.

Les premiers et derniers bits de chaque $\mathbf{D_{0i}}$ détermine (en binaire) la ligne de la fonction de sélection, les autres bits (respectivement 2, 3, 4 et 5) déterminent la colonne. La sélection de la ligne se faisant sur deux bits, il y a 4 possibilités (0, 1, 2, 3). La sélection de la colonne se faisant sur 4 bits, il y a 16 possibilités (0 à 15). Grâce à cette information, la fonction de sélection sélectionne une valeur codée sur 4 bits.

Voici la première fonction de substitution, représentée par une matrice de 4 par 16 :

Tableau 2.4 : Matrice de la fonction de substitution

S_i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

Soit $\mathbf{D_{01}}$ égal à *101110*. Les premiers et derniers bits donnent *10*, c'est-à-dire 2 en binaire. Les bits 2, 3, 4 et 5 donnent *0111*, soit 7 en binaire. Le résultat de la fonction de sélection est donc la valeur située à la ligne n°2, dans la colonne n°7. Il s'agit de la valeur *11*, soit en binaire *111*.

Chacun des 8 blocs de 6 bits est passé dans la fonction de sélection correspondante, ce qui donne en sortie 8 valeurs de 4 bits chacune.

Chaque bloc de 6 bits est ainsi substitué en un bloc de 4 bits. Ces bits sont regroupés pour former un bloc de 32 bits.

- **Permutation**

Le bloc de 32 bits obtenu est enfin soumis à une permutation **P** dont voici la table :

Tableau 2.5 : La table de Permutation *P*

P	16	7	20	21	29	12	28	17
	1	15	23	26	5	18	31	10
	2	8	24	14	32	27	3	9
	19	13	30	6	22	11	4	25

- **OU Exclusif**

L'ensemble de ces résultats en sortie de **P** est soumis à un *OU Exclusif* avec le $\mathbf{G}_0$ de départ (comme indiqué sur le premier schéma) pour donner $\mathbf{D}_1$, tandis que le $\mathbf{D}_0$ initial donne $\mathbf{G}_1$.

- **Itération**

L'ensemble des étapes précédentes (*rondes*) est réitéré 16 fois.

- **Permutation initiale inverse**

A la fin des itérations, les deux blocs $\mathbf{G}_{16}$ et $\mathbf{D}_{16}$ sont obtenus, puis soumis à la permutation initiale inverse :

Tableau 2.6 : La table de la permutation initiale inverse

PI^{-1}	40	8	48	16	56	24	64	32
	39	7	47	15	55	23	63	31
	38	6	46	14	54	22	62	30
	37	5	45	13	53	21	61	29
	36	4	44	12	52	20	60	28
	35	3	43	11	51	19	59	27
	34	2	42	10	50	18	58	26
	33	1	41	9	49	17	57	25

Le résultat en sortie est un texte codé de 64 bits.

- **Génération des clés**

Etant donné que l'algorithme du DES présenté ci-dessus est public, toute la sécurité repose sur la complexité des clés de chiffrement.

D'abord 56-bits de la clé sont sélectionnés des 64-bits du d épart, à travers le choix permuté 1 (PC1) ; les huit bits restant sont ignorés, ou alors utilisés comme bits de parité. Les 56-bits sont alors divisés en deux parties de 28-bits chacune. A partir de ce moment, chaque moitié est traitée séparément. Dans des itérations (*rondes*) successives, les deux parties de la clé sont rotationées d'un ou de deux bits (spécifique à chaque itération) ; et alors une sous-clé de 48-bits est choisie par le PC2 24-bits de la moitié gauche et 24-bits de la moitié droite. Les rotations (dénotées par "< < <" dans la Figure 2.8) signifient que un différent ensemble de bits est utilisé à chaque sous-clé, chaque bit étant utilisé dans environ 14 des 16 sous-clés.
L'ordonnancement pour le décryptage est similaire, mais il doit générer la clé dans l'ordre inverse. Donc, les rotations sont à droite ("> > >") et non plus à gauche.

L'algorithme ci-dessous montre comment obtenir, à partir d'une clé de 64 bits (composé de 64 caractères alphanumériques quelconques), 8 clés diversifiées de 48 bits chacune servant dans l'algorithme du DES :

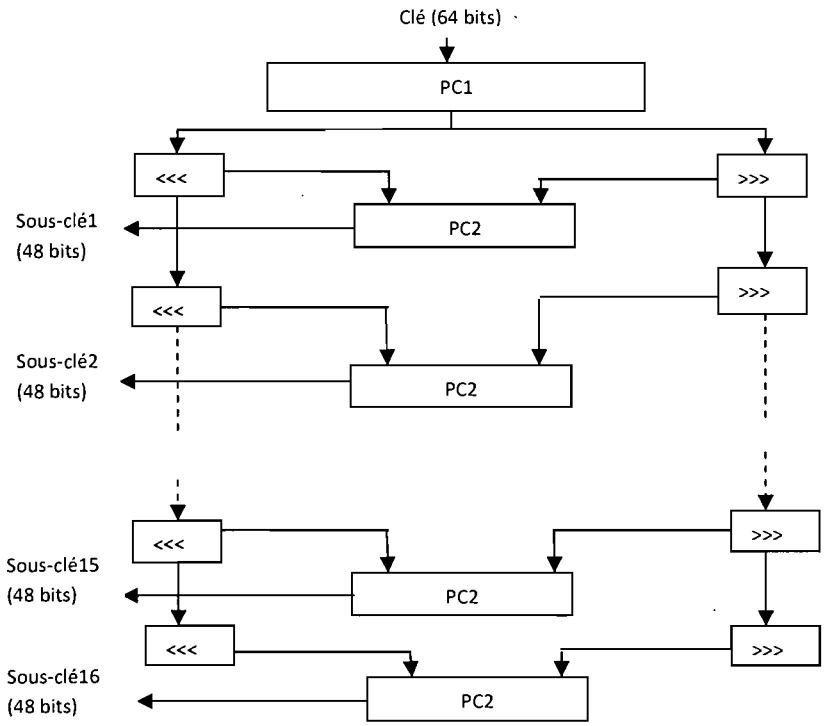

Figure 2.8 : L'ordonnancement de clés

Dans un premier temps les bits de parité de la clé sont éliminés afin d'obtenir une clé d'une longueur utile de 56 bits.

La première étape consiste en une permutation notée **CP1** dont la matrice est présentée ci-dessous :

Tableau 2.7 : La matrice de CP1

CP1														
	57	49	41	33	25	17	9	1	58	50	42	34	26	18
	10	2	59	51	43	35	27	19	11	3	60	52	44	36
	63	55	47	39	31	23	15	7	62	54	46	38	30	22
	14	6	61	53	45	37	29	21	13	5	28	20	12	4

Cette matrice peut en fait s'écrire sous la forme de deux matrices $\mathbf{G_i}$ et $\mathbf{D_i}$ (pour gauche et droite) composées chacune de 28 bits :

Tableau 2.7 : Les deux matrices G_i et D_i

G_i	57	49	41	33	25	17	9
	1	58	50	42	34	26	18
	10	2	59	51	43	35	27
	19	11	3	60	52	44	36
D_i	63	55	47	39	31	23	15
	7	62	54	46	38	30	22
	14	6	61	53	45	37	29
	21	13	5	28	20	12	4

On note $\mathbf{G_0}$ et $\mathbf{D_0}$ le résultat de cette première permutation.

Ces deux blocs subissent ensuite une rotation à gauche, de telles façons que les bits en seconde position prennent la première position, ceux en troisième position prennent la seconde, ... Les bits en première position passent en dernière position.

Les 2 blocs de 28 bits sont ensuite regroupés en un bloc de 56 bits. Celui-ci passe par une permutation, notée **CP2**, fournissant en sortie un bloc de 48 bits, représentant la clé $\mathbf{K_i}$.

Tableau 2.8 : La permutation CP2

CP2	14	17	11	24	1	5	3	28	15	6	21	10
	23	19	12	4	26	8	16	7	27	20	13	2
	41	52	31	37	47	55	30	40	51	45	33	48
	44	49	39	56	34	53	46	42	50	36	29	32

Des itérations de l'algorithme permettent de donner les 16 clés $\mathbf{K_1}$ à $\mathbf{K_{16}}$ utilisées dans l'algorithme du DES.

Chapitre 3

ETAT DE L'ART

L'objectif de notre travail est d'optimiser une implémentation matérielle de l'algorithme cryptographique DES sur une plate-forme reconfigurable basée sur FPGA. Notre conception a été implémentée sur une *Spartan III XC2V2000*. L'architecture du design a été décrite dans le langage VHDL, et dans le but de réduire le chemin critique du design associé, nous avons utilisé une ***approche pipeline*** avec l'accent sur la non-linéarité des S-boxes de DES, et ce pour atteindre une performance *haut débit.*

Dans ce chapitre on va examiner l'état de l'art des performances des expérimentations antérieures, ainsi que des différentes méthodes d'optimisation matérielles et logicielles pour l'architecture DES, ce qui permet de vérifier l'originalité de mon idée et connaitre son exploitabilité, ainsi que détecter des difficultés potentielles, des voies de développement à éviter, et ce pour donner des indications sur des voies nouvelles à explorer et des idées nouvelles de développement à mon projet.

La première section résume brièvement les performances des expérimentations antérieures.
La seconde section explore l'état de l'art des différentes méthodes d'optimisation matérielles et logicielles pour l'architecture DES, l'objectif est essentiellement d'identifier les principales méthodes existantes et leurs caractéristiques [3].

3.1 Revue des travaux antérieurs

Plusieurs recherches ont été faites sur l'implémentation accélérée (à haut débit) de l'algorithme cryptographique DES. La plupart d'entre elles utilise FPGAs comme la technologie cible [21, 22, 23, 24, 25, 26, 27, 28, 29, 32, 33], et seulement un petit nombre de ces recherches adopte ASIC

(processeur,...) comme-plateforme d'implémentation [15, 16, 17, 18, 19, 20], la raison est bien détaillée au premier chapitre (Introduction/Motivations).

L'implémentation Jbits de DES est effectuée sur FPGA Virtex XCV150 [24], parce que l'architecture Virtex implémente efficacement les opérations primitives DES et permet un haut degré de *pipeline*, et JBits fournit une application à base de java pour la création de temps d'exécution et la modification de la configuration bitstream, cela permet la spécialisation de circuit dynamique basée sur une clé spécifique et un mode (d'encryption/ décryption). L'ordonnancement des clés est calculé entièrement dans le logiciel et fait partie du bitstream. En conséquence, toute clé cryptographique et la logique de génération sous-clé sont enlevées de *datapath* entièrement déroulé.

Les travaux de recherche sur le design de l'algorithme DES à la thèse [32] ont conçu et implémenté des options d'architecture diverses avec l'accent fort sur la performance haut débit (à grande vitesse). Les techniques comme le *pipeline* et le *déroulement de boucle* ont été utilisées et leur efficacité pour DES sur FPGAs est examinée. L'optimisation sur un niveau inférieur aussi exécutée. Le résultat le plus intéressant est d'atteindre des débits de données allant jusqu'à 384 Mbit/s au moyen d'un FPGA Xilinx (vitesse-grade-3).

L'implémentation DES sur FPGA XC4028EX [21] utilise les deux techniques *Déroulement de boucle* et *Pipeline* à deux étapes (2-stages) et à quatre étapes (4-stages).

La thèse [29] traite la conception et l'implémentation d'un moteur de cryptage de base sur la base d'un cryptosystème à clé privée symétrique DES, en mettant l'accent sur le haut débit et les reconfigurables S-boxes. Deux architectures ont été mises en place : *pipeline* et *non-pipeline*, elles sont étudiées avec leur efficacité sur le DES. Le système de chiffrement de base est développé sur la plate-forme matérielle reconfigurable basée sur FPGA VirtexE. L'architecture *pipeline* explore des améliorations de la performance et de la vitesse. Un design pipeliné peut traiter simultanément de multiples blocs de données à la fois au lieu d'un seul bloc de données. L'avantage de cette conception par apport à la non-pipelinée est que chaque paire de bloc de données d'entrée et de ses associés principaux peut être chargée et calculée sur chaque cycle d'horloge, parce que dans l'architecture non-pipeline chaque cryptage ou décryptage sur cette conception dure 16 cycles d'horloge complets.

Quant au projet [25], l'implémentation de DES est faite sur la plateforme FPGA Xilinx VirtexII, et pour améliorer la vitesse, elle concentre sur les S-Boxes pour appliquer l'architecture de *pipeline* qui sert à chercher le chemin critique qui est le chemin le plus long entre deux étapes, c'est-à-dire, entre deux registres.

Un autre visage d'implémentation de DES sur FPGA XCV400 [22] est les *free DES cores* (les cœurs DES libres), qui utilise l'approche de *pipeline* sur le mode ECB en réalisant un débit de 3052 Mbits/s.

L'approche *pipeline* est aussi utilisée au projet technique [33], et ce pour implémenter le DES sur la plateforme FPGA Xilinx Spartan II. La première étape est d'analyser les composants utilisés dans la conception. Ainsi, à ce point, tout est prêt à la construction rapide du design en passant par le flot de conception simulation, de synthèse, placement et routage et le teste sur les FPGA.

Dans la nouvelle méthode de l'article [23], l'ordonnancement paramètrisable de la clé est présenté, il peut être utilisé dans des algorithmes de chiffrement à clé privée pipelinable. L'algorithme DES, qui se prête facilement au pipeline, est utilisé pour illustrer cette nouvelle méthode d'ordonnancement de la clé et l'applicabilité plus large de cette méthode à d'autres algorithmes de cryptage. L'implémentation est effectuée sur FPGA Xilinx Virtex XCV1000, En utilisant cette nouvelle méthode, le design de 16 étapes *pipelinées* de DES est réalisé, ce qui peut fonctionner à un taux de cryptage de 3,87 Gbit/s.

sur un FPGA VirtexE XCV400e. Comme une stratégie visant à réduire le chemin critique L'objectif de l'implémentation de DES [26] est de faire face à la cryptanalyse linéaire (attaques de plaintext), ce document présente une implémentation logicielle réalisant une vitesse de 21,3 Gbit/s (333 MHz). Dans cette conception le plaintext, la clé et le mode (encryption/decrytion) peuvent être changés sans cycles morts. Le design obtenu est testé sur huit FPGAs.

Le design à [27] est implémenté du design associé, on a utilisé une *structure parallèle* qui a permis de calculer toutes les huit DES S-boxes simultanément.

Le papier [28] présente des cœurs DES/3DES (*DES/3DES cores*) avec CBC et une architecture réseau qui peut les accélérer en parallèle utilisant efficacement un FPGA Virtex II

XC2V1000FG456 -4. Ce design est implémenté en utilisant ROMS qui sont synthétisées avec LUT. Puisque DES utilise une ronde semblable pour traiter l'implication des S-boxes 16 fois, une seule instance de son implémentation et le output est de retour en boucle dans l'unique étape DES 16 fois pour minimiser les ressources. Ce processus est contrôlé par une machine à états finis. Le design 3DES est tout à fait semblable, c'est le processus de DES répété 3 fois en utilisant des clés différentes, il est aussi contrôlé par une autre machine à états finis.

Toutes les expérimentations qu'on a vues sont implémentées sur la plateforme FPGA, et ce grâce à ses avantages, sa souplesse, et sa puissance. Néanmoins, ça n'empêche pas de trouver des implémentations sur les ASIC.

En effet, à [17] l'implémentation de DES sur DEC3000 et Sun4/280 est *logicielle*, ce qui fait, l'algorithme DES peut être modifié dans le nombre de voies qui n'auront aucune influence sur la fonction calculée selon le programme. Dans chaque cryptage DES il y a 16 paires produit-transformation/bloc-transformation. La transformation de bloc est simplement l'échange de R32 bits avec L32 bits. Pour éviter la transformation de bloc à la fin de chaque transformation de produit, il ya deux transformations de produits différentes: l'une fonctionne de la manière habituelle et l'autre applique les opérations sur L comme si elle était R et verse-versa. De même pour le cas à [16], toujours l'échange est construit dans l'itération, parce que R et L peuvent jouer des rôles réciproques.

L'implémentation à [15] sur SNL DES ASIC utilise l'architecture de *pipeline* pour augmenter de la vitesse de DES, l'opération mène à une fréquence de 105 Mbits/s sur des mots de 64 bits.

L'implémentation VLSI du DES sur la technologie CMOS micron statique 0,6 à [18] est plus performante. En utilisant l'approche de *pipeline*, le chiffrement peut être effectué au taux de plus de 6,7 GBS.

Le travail de [19] s'articule sur l'implémentation de l'algorithme DES sur le processeur TMS320C6000E. Le code source C a été optimisé à l'aide de la version 3.01 du compilateur C (*Optimizing C Compiler*). Le débit sur C6201 (200 MHz) est mesuré à 53 Mbits/s dans le DES et 22 Mbits/s pour le triple-DES. Le rythme sur C6211 (150 MHz) est mesuré à 39 Mbits/s dans le DES et 18 Mbits/s pour le triple-DES.

La thèse [20] fournit une implémentation du DES fortement optimisée pour le processeur Pentium Intel. Cette conception améliore la vitesse de chiffrement DES par le haut débit, cela est réalisé sans augmenter la taille de LUT. Un total de 4 kilo-octets de LUT est utilisé par cette implémentation. Par conséquent, l'algorithme DES s'exécute dans environ 240 cycles sur les AMD Duron (*Spitfire core*), 319 cycles sur l'Intel Pentium III, et 445 cycles sur Pentium 4. Ce sont tous réalisés avec seulement quelques modifications mineures au design, à savoir l'ajout de pré-instructions, et une modification de l'ordonnancement de la fonction ronde.

3.2 Architecture des techniques d'optimisation

Les implémentations matérielles et logicielles des algorithmes cryptographiques ont une longue histoire.

Traditionnellement, les algorithmes ont été implémentés dans le matériel pour atteindre une vitesse plus élevée que l'implémentation dans le software.

Pour accélérer le traitement, il faut tenir compte de l'architecture du circuit cible, à savoir un grand nombre de registres et un traitement pipeline des instructions pour les calculs proprement dit, et de la hiérarchie mémoire c'est à dire de l'accès aux données via les mémoires caches. Les méthodes qu'on va examiner sont pratiques et elles partagent le même objectif qui est l'implémentation à haute vitesse [14, 30, 32].

3.2.1 Architecture DES basique

L'algorithme DES possède une structure itérative. On passe des données 16 fois par le réseau Feistel [20], chaque fois avec une sous-clé différente de la clé de transformation. Cette structure se mène naturellement au bloc diagramme montré dans la figure 3.1. On passe les données entrantes et la clé par la permutation initiale. Alors les données passent 16 fois par le réseau Feistel et aussi 16 sous-clés sont produites simultanément. Tous les deux, l'opération de réseau Feistel et la génération sous-clés sont dénotées dans le bloc diagramme CLU (*Combinatorial Logic Unit*) *i.e.* une seule ronde de réseau Feistel. Afin d'être en mesure de boucler la sortie de retour à l'entrée de CLU nous avons besoin de registres et de multiplexeurs. Les multiplexeurs commutent l'entrée de CLU entre les données de la ronde précédente et les nouvelles données

d'entrée et des clés. Les registres stockent les résultats de chaque boucle et les transmettent au multiplexeur. La sortie du registre de données passe par la permutation finale.

3.2.2 Boucle itérative (*Iterative looping*)

C'est une méthode efficace pour réduire au minimum le matériel nécessaire lors d'implémentation d'une architecture itérative [31]. Si une seule ronde est implémentée, la $n^{ième}$ ronde de chiffrement doit répéter n fois pour effectuer un cryptage. Cette approche a un bas délai registre-à-registre mais nécessite un grand nombre de cycles d'horloge pour effectuer un cryptage.

3.2.3 Déroulement de boucle (*loop-unrolling)*

Le déroulement de boucle est la concaténation des unités combinatoires afin de réduire le nombre d'itérations [14, 31, 32], par exemple, si deux boucles sont déroulées alors deux rondes de DES seront calculées avec un cycle d'horloge. La figure 3.2 montre le bloc diagramme. Ce dernier diffère de la figure 3.1 seulement dans la 2ème CLU. La permutation tant initiale que finale ainsi que les registres et les multiplexeurs sont les mêmes.

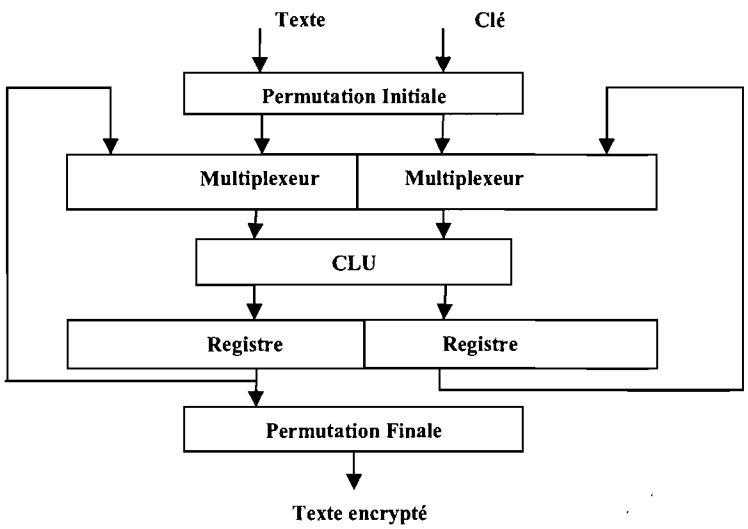

Figure 3.1: Bloc diagramme de DES

Cette technique consiste, dans le cas le plus simple, à dupliquer le corps de la boucle afin d'avoir une meilleure utilisation de l'unité de calcul d'un processeur, par exemple. En fonction de l'algorithme, il est possible de lancer des calculs en «parallèle». Malgré cela le pipeline de l'unité de traitement n'est pas toujours plein. Le software pipelining y remédie en exécutant, dans le corps de boucle, des instructions appartenant à d'autres itérations.
Le déroulement de boucle conduit potentiellement à des améliorations de vitesse. Dans la version non déroulée [29], une itération de DES a le modèle de chronométrage (de choix du temps) simple suivant: $T_{mux}+ T_{cl}+T_{reg}$, où

T_{mux} désigne le temps d'un signal doit passer par l'intermédiaire d'un multiplexeur,

T_{cl} le retard introduit par la logique combinatoire,

T_{reg} le retard présenté par le registre. Les délais de retard (*Wiring delays*) sont assumés pour être inclus dans les temps indiqués pour les éléments. Alors pour l'ensemble de 16 rondes on obtient:

$$16xT_{mux}+16xT_{cl}+16xT_{reg}.$$

L'équation pour une boucle de la structure de la figure 3.2 est ainsi: $T_{mux}+2xT_{cl}+T_{reg}$. Ce doit être exécuté 8 fois, de sorte que l'ensemble de retard (*over-all delay*) est désormais: $8xT_{mux}+16xT_{cl}+8xT_{reg}$. Le même principe peut être appliqué à quatre rondes DES déroulées en résultant: $4xT_{mux}+16xT_{cl}+4xT_{reg}$.

Évidemment, nous ne pouvons pas réduire le retard introduit par CLU, mais nous avons réduit les pistes à travers les multiplexeurs et les buffers. Il existe également une autre motivation pour augmenter la vitesse si les méthodes de conception moderne sont appliquées. Il est possible que les outils de synthèse puissent optimiser une conception mieux déroulée. Le déroulement de boucle fonctionne donc avec un processus de conception moderne permettant de réduire la complexité logique du design.

3.2.4 Pipeline

Le pipeline essaye de réaliser une amélioration de vitesse d'une façon différente [1, 32]. Au lieu de traiter chaque bloc de données après l'autre, une conception de pipeline peut traiter deux blocs

de données ou plus simultanément. La conception avec deux pipelines est montrée à la figure 3.3.

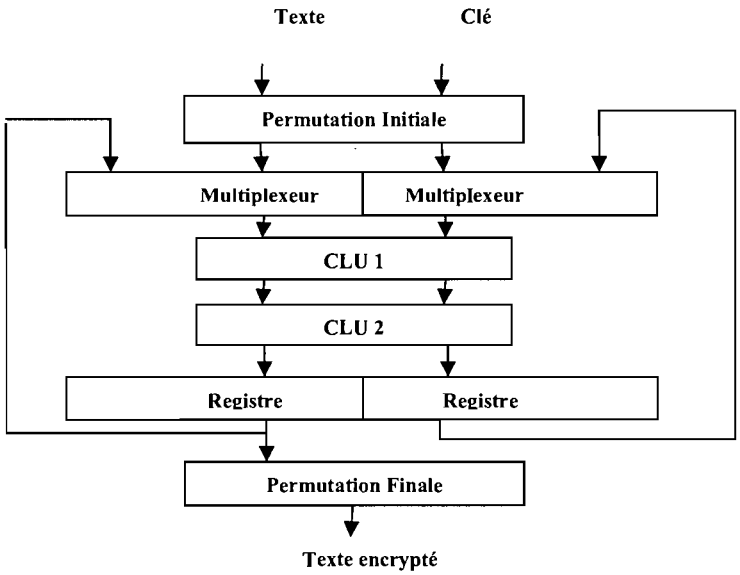

Figure 3.2: Bloc diagramme de DES avec 2 boucles déroulées

Le bloc diagramme dans la figure 3.3 est très semblable à celui avec les deux boucles déroulées (la figure 3.2). La seule différence est le Buffer supplémentaire entre les CLUs.

Le premier bloc de données X_1 et la clé associée K_1 sont chargés et passés par les permutations initiales et le multiplexeur. Premier CLU calcule $X_{1,1}$ et $K_{1,1}$ qui sont stockés dans le premier bloc de registre. Dans le prochain cycle d'horloge $X_{1,1}$ et $K_{1,1}$ quittent les premiers registres et la seconde CLU calcule $X_{1,2}$ et $K_{1,2}$ qui sont mis dans le deuxième bloc de registre. En même temps le deuxième bloc de données X_2 et la clé K_2 sont chargés et passés par les permutations initiales et le multiplexeur. la première unité combinatoire calcule $X_{2,1}$ et $K_{2,1}$ qui sont déplacés dans le premier bloc de registre.

Maintenant le pipeline est rempli à chaque cycle d'horloge. Une autre itération de deux paires de clés et de données est calculée. Les données qui sont entrées au pipeline en premier lieu, le quitteront aussi en premier. À ce moment-là les données suivantes et la paire de clés peuvent être chargées.

L'avantage de ce design est que deux ou plusieurs paires de clés peuvent être travaillées en même temps. Comme il y a seulement un cas des permutations initiales, le multiplexeur et la permutation finale, le coût en termes de ressources sur la plateforme ne sera pas deux fois plus élevé comme si nous avons mis en place deux designs DES non-pipelinés. La vitesse d'horloge maximale devrait être à peu près la même puisque pendant un cycle d'horloge la même quantité de ressources logiques doit être parcourue comme dans le design non-pipeliné. Il est également simple de concevoir des pipelines de plus de deux à quatre étapes.

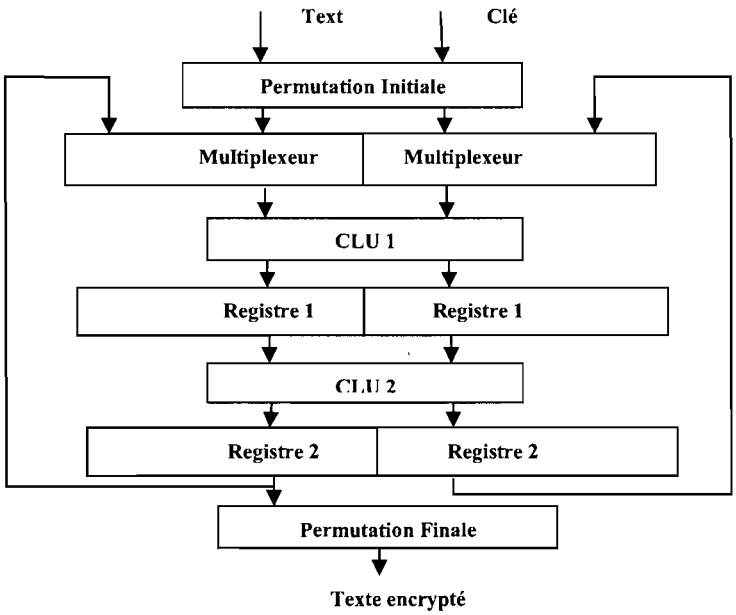

Figure 3.3: Bloc diagramme de DES avec 2 étapes (2-stages) de pipeline

3.2.5 Combinaison du déroulement de boucle et le pipeline

Il est possible de combiner les deux techniques d'optimisation que nous avons décrites ci-dessus [32]. Par exemple, chaque étape de pipeline peut contenir deux boucles déroulées. Le bloc diagramme résultant ressemble à la figure 3.3, sauf que chaque CLU est dupliquée. Pendant un cycle d'horloge deux itérations de deux paires de clés $X_{1,4}$ et $K_{1,4}$ sont calculées à partir de $X_{1,2}$ et $K_{1,2}$, mais $X_{2,2}$ et $K_{2,2}$ sont calculées à partir de X_2 et K_2. Une extension à quatre boucles déroulées par étape de pipeline est aussi possible.

Les besoins contemporains et de futures applications cependant, exigent souvent d'autres propriétés des implémentations matérielles.

3.2.6 Optimisation logicielle de DES

L'algorithme DES peut être modifié dans le nombre de voies qui n'auront aucune influence sur la fonction calculée selon le programme. Dans chaque cryptage DES il y a 16 paires produit-transformation/bloc-transformation. La transformation de bloc est simplement l'échange de R32 bits avec L32 bits. Pour éviter la transformation de bloc à la fin de chaque transformation de produit, il ya deux transformations de produit différentes: l'une fonctionne de la manière habituelle et l'autre applique les opérations sur L comme si elle était R et verse-versa [17]. L'utilisation de ces deux transformations de produit élimine alternativement le besoin d'une transformation de bloc entre des transformations de produit. Le seul problème avec cet arrangement est qu'après la dernière transformation de produit, les blocs de L et R sont échangés (Figure 3.4). Ce renversement est incorporé dans la finale IP^{-1} parce qu'un échange et une permutation sont juste une permutation [16]. Soit ϵ représentant l'expansion E, σ représente la fonction S et π représente la permutation P. K_i étant la $i^{ème}$ sous-clé, Ri étant la $i^{ème}$ valeur de droite, et Li est la $i^{ème}$ valeur de gauche

Par définition :

$$\mathbf{Li \equiv Ri\text{-}1}$$

$$\mathbf{Ri \equiv Li\text{-}1 \oplus \mathit{f}i\ Ri\text{-}1}$$

$$\mathbf{\mathit{f}i\ Ri\text{-}1 \equiv \pi\sigma(Ki \oplus \epsilon Ri\text{-}1)}$$

où f_i est la transformation f qui utilise la sous-clé K_i.

Après deux rondes:

$$\mathbf{Li{+}1 \equiv Li\text{-}1 \oplus \mathit{f}i\ Ri\text{-}1}$$

$$\mathbf{Ri{+}1 \equiv Ri\text{-}1 \oplus \mathit{f}i{+}1(Li\text{-}1 \oplus \mathit{f}i\ Ri\text{-}1)}$$

Puisqu'il y a 16 rondes dans DES, 8 double-rondes de la forme suivante peuvent être utilisées au lieu de cela :

$$\mathbf{Li+1 \equiv Li-1 \oplus \mathit{f}i\ Ri-1}$$

$$\mathbf{Ri+1 \equiv Ri-1 \oplus \mathit{f}i+1\ Li+1}$$

Remarquons qu'il n'existe aucune valeur intermédiaire entre Ri et Li.

Après 1 double-ronde :

$$\mathbf{Li+1 \equiv Li-1 \oplus \mathit{f}i\ Ri-1}$$

$$\mathbf{Ri+1 \equiv Ri-1 \oplus \mathit{f}i+1(Li-1 \oplus \mathit{f}i\ Ri-1)}$$

Qui est le même comme auparavant.

En réalité, l'échange a été construit dans l'itération, parce que R et L peuvent jouer des rôles réciproques.

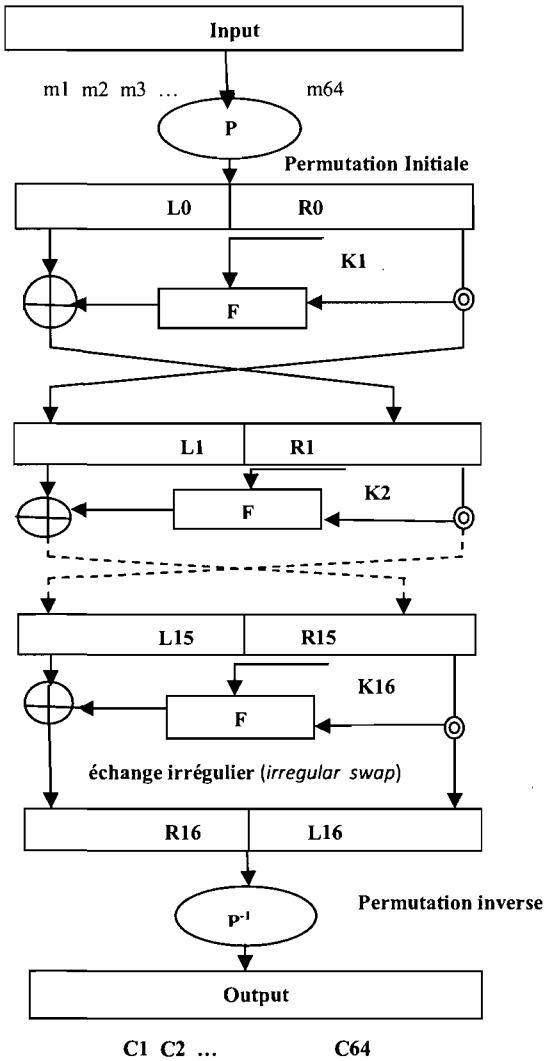

Figure 3.4: DES avec l'échange irrégulier (*irregular swap*)

Conclusion (Positionnement de nos travaux)

Nos travaux de mémoire s'inscrivent dans le développement d'une implémentation matérielle de l'algorithme cryptographique DES. Notre conception a été implémentée sur une plateforme FPGA Spartan III. Comme une stratégie de réduire le chemin critique de conception associé, nous avons utilisé une ***approche pipeline*** qui nous a permis d'obtenir des performances compétitives.

Ces travaux sont innovants sous la réalisation d'un flot de conception qui prend en entrée une application modélisée en algorithme pseudo-code pour générer le code VHDL du design correspondant.

Nous introduisons nos travaux et nos contributions plus en détail dans les chapitres suivants.

Chapitre 4

METHODOLOGIE DE CONCEPTION

Ce chapitre discute la méthodologie de conception. Il propose un flot de conception pour le développement d'application de l'algorithme DES implémenté sur FPGA. Il met l'accent sur les différents outils utilisés pour créer le logiciel et le matériel ainsi que la façon dont ils sont combinés pour réaliser de grands résultats de conception. Cela comprend également le traitement des différentes parties du code VHDL.

4.1 Flot de conception

Pour la conception de matériel complexe, avec DES qui est sans aucun doute, le développement, la simulation, la synthèse et des outils de vérification sont nécessaires. Le flot de conception générale de cette implémentation de DES sur FPGA se base généralement sur les éléments suivants [36]:

- Obtenir des informations sur des expérimentations antérieures (état de l'art), et ce pour avoir une vue d'ensemble des travaux effectués avec succès, ce qui permet de réduire les éventuelles défaillances au cours de la phase de conception ;
- Comparer des différents algorithmes et évaluer les résultats de leurs implémentations matérielles ;
- Choisir une architecture appropriée ;
- Implémenter le matériel avec un langage de description de matériel : VHDL ;
- Vérifier le matériel par la simulation et comparer les résultats avec le modèle de haut niveau pour vérifier la fonctionnalité du matériel décrit. Cette simulation peut être faite sur les différentes couches d'abstraction ;

- Synthétiser le matériel pour générer un tributaire de la technologie sur le *Netlist* niveau que la porte (*gate level*) doit être optimisée dans la plupart des points de vue différents;
- Trouver un fichier de configuration FPGA et comparer les résultats avec le modèle de haut niveau ;
- Optimiser l'implémentation pour la plate-forme cible ;

La figure 4.1 montre le flot de conception pour FPGA :

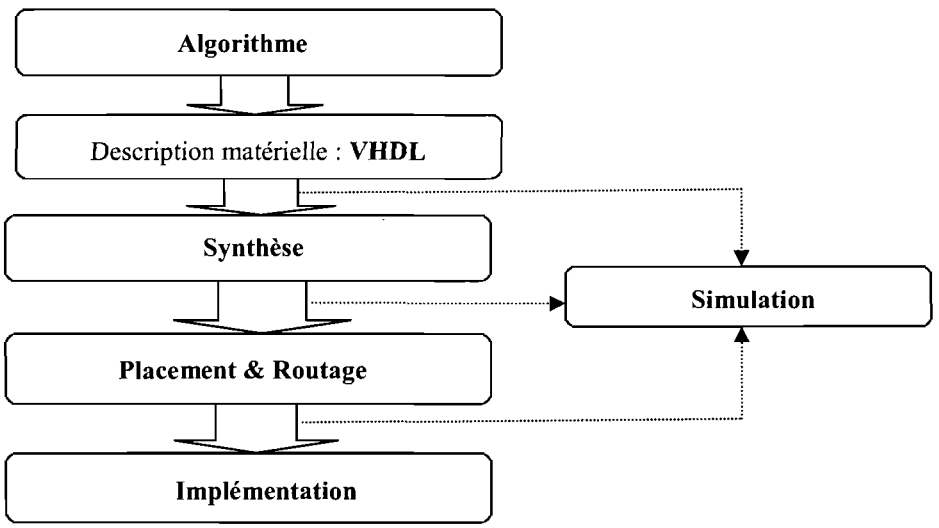

Figure 4.1 : Flot de conception

4.2 Outils

Il est essentiel de détailler les différents outils (et leur version) utilisés dans ce projet parce que chacun a ses caractéristiques (fonctions) spécifiques et d'autres flux.

4.2.1 Logiciel (*Software*)

- Langage VHDL : langage de description de circuits ou de systèmes numériques en vue de modélisation (simulation) standard des circuits ou systèmes, synthèse (génération

automatique) niveau RTL, descriptions de programmes de test (stimuli), et description de type hiérarchique (netlist) :

Il permet la description des aspects les plus importants d'un système matériel (*hardware system*), à savoir son comportement, sa structure et ses caractéristiques temporelles.

- ModelSim (PE Student Edition 6.4e) : il fournit un environnement complet de simulation et déboggage pour les designs complexes en ASIC et en FPGA. Il supporte plusieurs langages de description, dont le VHDL. Il valide l'architecture réalisée, utilisant des *timings* moyens typiques appliqués après placement/routage. Il valide l'instanciation faite sur le FPGA, utilisant les *timings* issus du logiciel de placement/routage du constructeur. Il présente l'intérêt d'une écriture des *stimuli* dans le langage VHDL lui-même.
- Xilinx ISE 7.1i : ISE (*Integrated Software Environment*) est un panel d'outils développé et commercialisé par la société Xilinx [8]. Ce panel permet de réaliser toutes les phases du flot de conception d'applications sur des composants reconfigurables (FPGA) de la société Xilinx. Il est constitué d'une dizaine d'outils, exemples : *Project navigator* : outil de description du projet (spécification du système à concevoir en VDHL, schematic, machines d'états), synthèse, placement, routage. *FloorPlanner* : outil de visualisation et de localisation des éléments utilisés du composant, ainsi que des communications réalisées. *FPGA editor* : outil de visualisation du placement et routage, disposant d'un placeur routeur intégré. *iMPACT* : utilisé pour télécharger directement le fichier *Bitstream* au FPGA.
- Xilinx ChipScope Pro 7.1i : est un logiciel embarqué basé sur un analyseur logique permettant d'effectuer du déboggage. Il permet de visualiser des données internes au FPGA grâce au câble JTAG (*Joint Test Action Group*), et de contrôler des signaux du design en y insérant un ICON et un ILA et en les connectant proprement. Il est constitué de deux parties : CHIPSCOPE CORE GENERATOR et CHIPSCOPE. CHIPSCOPE CORE GENERATOR génère un composant VHDL que l'on insère dans le code. On connecte jusqu'à 255 signaux que l'on désire observer. Une fois le code inséré dans le FPGA, on peut utiliser CHIPSCOPE pour télécharger les données du FPGA. Comme l'ordinateur et le FPGA ne sont pas du tout à la même échelle de temps, on règle un trigger qui va se

déclencher selon certains paramètres des signaux observés. Une fois le trigger déclenché, les données sont stockées dans les blocks RAM du FPGA qui jouent le rôle de buffers. Lorsque le buffer est plein, les données sont téléchargées sur l'ordinateur. On peut alors utiliser l'interface de CHIPSCOPE pour afficher les données sous forme de bus, ou de graphique.

4.2.2 Matériel (*Hardware*)

- **FPGA Spartan III** : FPGA (*field programmable gate arrays*) [2, 4] se traduit en français par circuit prédiffusé programmable. Inventé par la société Xilinx, le FPGA, dans la famille des ASICs (*Application Specific Integer Circuit*), se situe entre les réseaux logiques programmables et les prédiffusés. C'est donc un composant standard combinant la densité et la performance d'un prédiffusé avec la souplesse due à la reprogrammation des PLD (*Programmable Logic Device*). Cette configuration évite le passage chez le fondeur et tous les inconvénients qui en découlent. La plupart des grands FPGA modernes sont basés sur des cellules SRAM aussi bien pour le routage du circuit que pour les blocs logiques à interconnecter. Au niveau le plus bas, les blocs logiques configurables, tels que les slices (tranches) ou les cellules logiques élémentaires, sont constitués de deux éléments essentiels : une table de correspondance (LUT ou *Look-Up-Table*) et d'une bascule (*Flip-Flop* en anglais). Il est important de le souligner car les diverses familles des FPGA se distinguent par la façon dont les bascules et les LUT sont conditionnées ensemble. La LUT sert à implémenter des équations logiques ayant généralement 4 à 6 entrées et une sortie. Elle peut toutefois être considérée comme une petite mémoire, un multiplexeur ou un registre à décalage. Le registre permet de mémoriser un état (machine séquentielle) ou de synchroniser un signal (pipeline). Les blocs logiques sont connectés entre eux par une matrice de routage configurable. Ceci permet la reconfiguration à volonté du composant, mais occupe une place importante sur le silicium et justifie le coût élevé des composants FPGA [6]. Les densités actuelles ne permettent plus un routage manuel, c'est donc un outil de placement-routage automatique qui fait correspondre le schéma logique voulu par le concepteur et les ressources matérielles de la puce. Comme les temps de propagation dépendent de la longueur des liaisons entre cellules logiques, et que les algorithmes d'optimisation des placeurs-

routeurs ne sont pas déterministes, les performances (fréquence max) obtenues dans un FPGA sont variables d'un design à l'autre. L'utilisation des ressources est par contre très bonne, et des taux d'occupation des blocs logiques supérieurs à 90% sont possibles. Comme la configuration (routage et LUTs) est faite par des points mémoire volatiles, il est nécessaire de sauvegarder le design du FPGA dans une mémoire non volatile externe, généralement une mémoire Flash série, compatible JTAG (*Joint Test Action Group*). Certains fabricants se distinguent toutefois par l'utilisation de cellules EEPROM (*Electrically-Erasable Programmable Read-Only Memory*) pour la configuration, éliminant le recours à une mémoire externe, ou par une configuration par anti-fusibles (la programmation par une tension élevée fait "claquer" un diélectrique, créant un contact). Cette dernière technologie n'est toutefois pas reconfigurable [6]. Chaque circuit intégré FPGA, figure 4.2, est constitué d'un nombre limité de ressources prédéfinies avec des interconnexions programmables pour mettre en œuvre un circuit numérique reconfigurable. Dans les spécifications d'un circuit intégré FPGA on trouve la quantité de blocs logiques configurables, le nombre de blocs logiques de fonctions figées, comme les multiplicateurs et la taille des ressources de la mémoire telles que le bloc de RAM embarquée (embedded block RAM). Un circuit intégré FPGA est composé de plusieurs autres éléments, mais ceux-ci sont typiquement les plus importants lors du choix et de la comparaison des FPGA en vue d'une application spécifique.

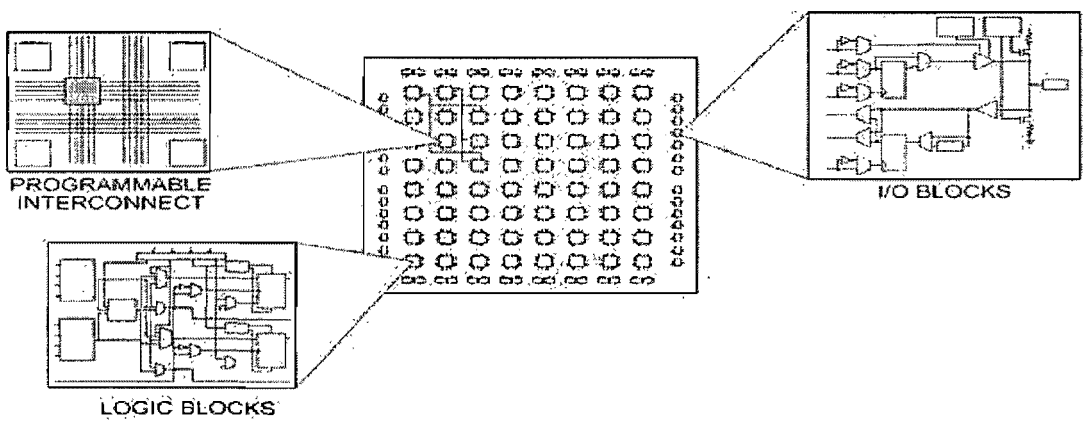

Figure 4.2 Les différents éléments d'un FPGA [7]

Dans le cas de notre projet, j'ai choisi Spartan III XC2V2000 package FF896 speed -4 dont le tableau 4.1 répertorie ses caractéristiques et leurs valeurs appropriées :

Tableau 4.1 Caractéristiques de FPGA Spartan III XC2V2000

Caractéristiques	Valeurs
Slices	10752
Slice Flip Flops	21504
4 input LUTs	21504
bonded IOBs	624
GCLKs	16

- **Câble Parallèle Xilinx IV**:

Les FPGAs sont des dispositifs programmables à base de mémoire volatile, la programmation doit être rechargée chaque fois que l'appareil s'allume. Le processus par lequel le dispositif reçoit son programme est appelé ``la configuration`` dont le FPGA Xilinx peut faire charger par le Câble Parallèle Xilinx IV.

D'habitude, le dispositif est reconfiguré extérieurement en utilisant ce câble parallèle. Il joint le PC au connecteur JTAG du système cible. Son but est de télécharger le *bitstream* au FPGA tant pour la phase de placement et routage que pour celle de déboggage.

Le *bitstream* est d'abord chargé dans l'outil *Xilinx iMPACT*. En suite le fichier sera envoyé, par le câble parallèle, à la carte.

4.3 Description du code

Avant d'entamer la partie d'implémentation, on examine les différents composants de notre code VHDL.

En effet, notre projet se base sur deux fichiers principaux, l'un de pipeline et l'autre pour le banc d'essai (*testbench*) :

- Le fichier *Pipeline* contient l'implémentation de l'algorithme DES, la description utilisée est structurelle, elle comporte plusieurs composants (*components*), ce qui nous permet une architecture hiérarchique et une synthèse logique efficace parce que la synthèse est un processus lent (en terme de temps de calcul). Plus un bloc est gros et complexe, plus sa synthèse

prendra du temps. Il vaut donc mieux travailler sur des blocs plus petits, plus simples à synthétiser, et rassembler le tout à la fin. VHDL permet l'assemblage de ces composants ce qui constitue une description structurelle. Ces composants peuvent être appelés plusieurs fois dans un même circuit. A la différence du mécanisme d'instanciation directe, la déclaration de composant permet de disposer d'une interface entre ce dont le modèle a besoin et ce qui est disponible dans une bibliothèque de conception. Ceci permet de définir les modèles effectifs des composants en-dehors de l'architecture et donc d'offrir plus de souplesse dans les cas où plusieurs architectures de composants sont possibles.

- Le fichier *TestBench* contient une référence qui sera utilisée dans le processus de simulation, il ne contient aucun des signaux « ***out***, ***in***, ***inout*** ou ***buffer*** » qui sont tous convertis en signaux internes. Il ne s'agit plus d'une description matérielle synthétisable, il permet de

- créer un module de vérification qui affiche des messages d'erreur et effectue des tests sur la valeur des signaux dans le temps.

Comme son nom l'indique, le fichier pipeline correspond aux 16 rondes de l'algorithme DES et dont la description est structurelle utilisant 16 rondes.

En lisant et analysant le fichier, nous pouvons écrire un premier schéma, figure 4.3, sur la façon dont le code fonctionne.

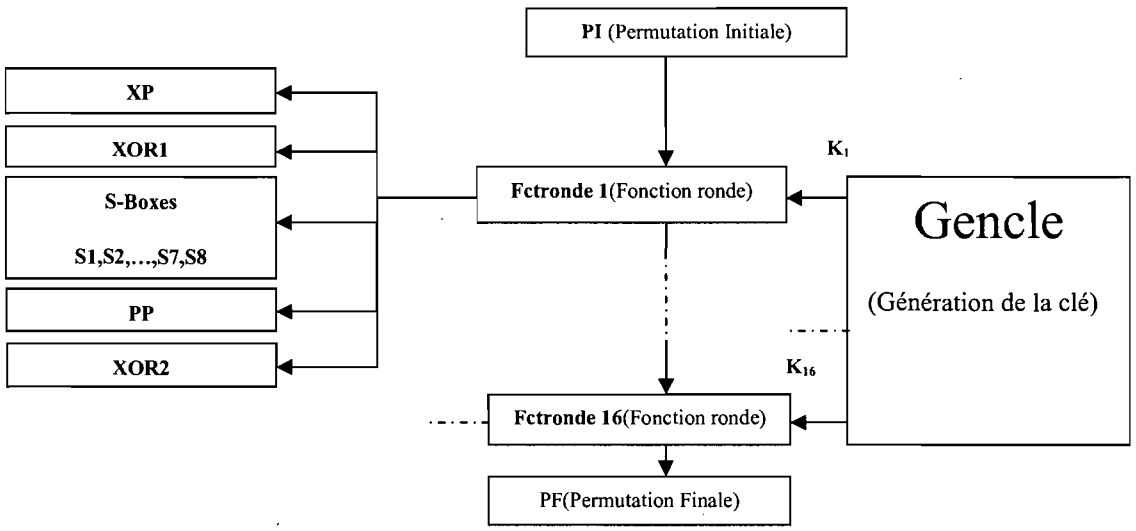

Figure 4.3: La structure du code VHDL

Cette figure correspond à ce que nous avons appris dans la description de l'algorithme DES. Les composants au début et à la fin sont, respectivement, permutation initiale et permutation finale (comme nous l'avons vu dans le deuxième chapitre).

Maintenant, nous pouvons plus facilement décrire la fonction de chaque composant :

- Encdes : c'est le sommet du design, il comprend quatre composants :
  . gencle (génération de la clé)
  . pi (permutation initiale)
  . fctronde (fonction du round)
  . pf (permutation finale

- Gencle : composant de génération de la clé, il contient deux composants :
  . CP1 (choix permuté 1)
  . CP2 (choix permuté 2)
  CP1 et CP2 tous les deux permutent des composants de bits. CP1 renonce à 8 bits de la clé (qui est à l'origine 64 bits de largeur). En pratique, ces 8 bits sont utilisés pour vérifier si la clé n'a pas été changée (avec un contrôle de parité).
  CP2 renonce aussi à quelques bits pour réduire le nombre de bits de 56 à 48.
  Le composant Gencle (et ses subcomposants) utilise seulement des ressources de liaison, parce qu'il ne combine que des permutations et des substitutions seulement. Donc cette partie sera exécutée très vite et aucune optimisation n'aurait aucun effet.

- PI : La permutation initiale est seulement une question d'échange de bits du message à encrypter (texte en clair). Donc nous pouvons déjà affirmer que ce composant n'exigera pas la ressource logique.

- Fctronde : C'est la fonction ronde, se répète 16 fois. Elle est, en fait, une conception structurelle qui interconnecte les composants suivants :
  · xp (fonction d'expansion) : Les 32 bits sont étendus à 48 bits grâce à une table (matrice) appelé *table d'expansion*, dans laquelle les 48 bits sont mélangés et 16 d'entre eux sont dupliqués.· xor1 (La fonction OU exclusif 1 (xor 1)) : L'algorithme DES procède ensuite à un *OU exclusif* entre la première clé et La matrice résultante de 48 bits. Le résultat de ce *OU exclusif* est une matrice de 48 bits.

· s1, s2, s3, s4, s5, s6, s7, s8 (boîtes de substitution (S-boxes)) : qui sont, en fait, des LUT (Look Up Table). Les registres nécessaires au pipeline sont également intégrés dans le S-boxes.

· pp (p-permutation) : le bloc de 32 bits obtenu est enfin soumis à une permutation

· xor2 (La fonction OU exclusif 2(xor 2)) : une autre fonction, OU exclusif, permet de XORer ce qui est permuté avec la partie gauche de la fonction ronde.

- PF : la permutation finale est en faite une permutation initiale inverse.

De là, nous pouvons conclure que beaucoup de composants utilisent seulement des ressources non logiques. Ainsi il n'y aura pas d'effort dans l'optimisation logique pour ces composants.
Par contre, Les seuls composants qui utiliseront des ressources logiques sont xor1, xor2, et les S-Boxes. Néanmoins, les fonctions xor1 et xor2 ne peuvent pas être optimisées logiquement parce qu'elles utilisent les fonctions XOR de base. Quant aux S-BOXES, elles utilisent des ressources logiques. En plus, la grande sécurité repose sur les S-Boxes étant non linéaires, très efficaces pour diluer des informations.

Chapitre 5

IMPLEMENTATION DE L'ARCHITECTURE

Avant toute conception de chiffrement et d'implémentation vient toujours la question des objectifs de performance à atteindre. Déclarer correctement ces objectifs en fonction d'une application cible déterminant les bons paramètres pour l'évaluation des performances est donc une étape importante dans la compréhension des architectures reconfigurables.

Le présent chapitre discute comment l'architecture du système est implémentée ainsi que les résultats expérimentaux compilés au cours de ce travail de recherche.
Les outils d'analyse et de vérification examinent le comportement du circuit. Les outils de synthèse et d'implémentation génèrent et optimisent les schémas de circuit. Les outils de testabilité vérifient la fonctionnalité de la conception. Pour notre design, nous nous servons de *Xilinx foundation tools* pour effectuer l'implémentation et la vérification.

5.1 Conception optimisée: Architecture *Pipeline*

D'après la description de l'algorithme DES, au chapitre IV, nous pouvons conclure que les seuls composants qui utilisent des ressources logiques sont XOR1, XOR2, et les S-Boxes. Néanmoins, les fonctions XOR1 et XOR2 ne peuvent pas être logiquement optimisées, parce qu'elles utilisent les fonctions XORs de base. Quant aux S-Boxes, elles utilisent des ressources logiques, car la grande sécurité repose sur les S-Boxes étant non linéaires et très efficaces pour diluer les informations.
Maintenant, nous allons mettre l'accent sur les S-boxes, qui correspondent à un grand cas en VHDL, et qui seront implémentées dans des LUTs de FPGA [24, 29, 33].
Nous pouvons utiliser la figure 5.1 ci-après pour expliquer, d'une façon générale, comment l'architecture pipeline fonctionne:

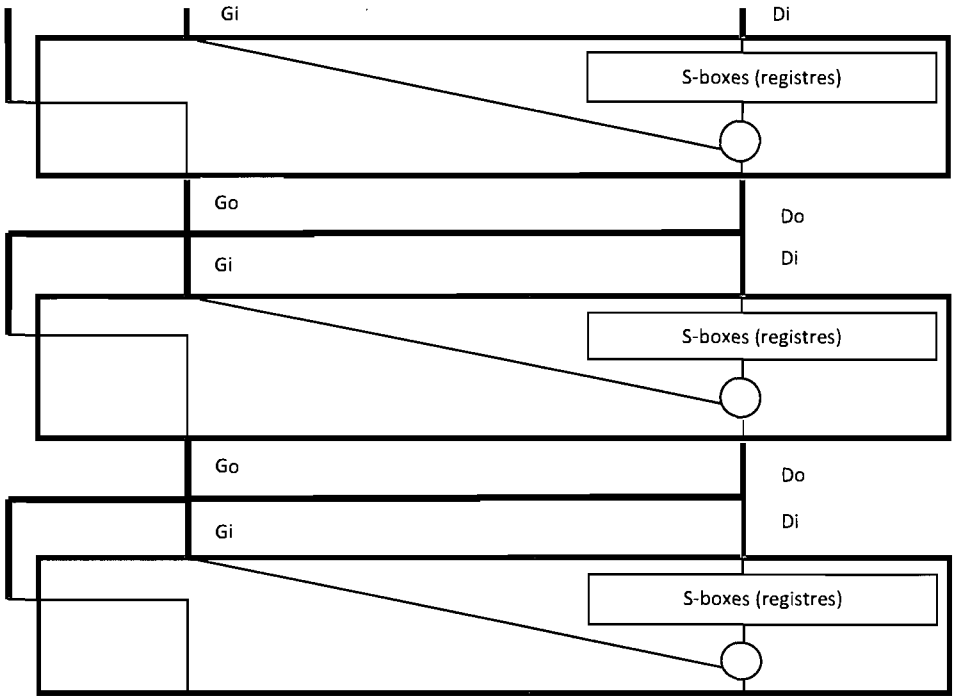

Figure 5.1 : Schéma d'une architecture pipeline

Comme les registres sont de 4 bits de largeur (taille de la production qui a besoin d'être sauvegardée) et comme nous avons 8 S-boxes, nous obtenons 32 bits en mémoire à chaque étape (une étape = une ronde). Il peut sembler un peu étrange parce que nous avons 64 bits à transmettre entre chaque étape. Mais si l'on considère l'algorithme DES, la partie gauche de la prochaine ronde est la partie droite de la précédente. Donc seulement la moitié des informations doit être stockée (32 bits). En faisant cette voie, nous allons favoriser une architecture optimisée pour la gestion de l'espace, comme nous stockons seulement 32 bits au lieu de 64 bits.
Mais cette approche mène à une conception moins efficace en termes de vitesse. Pour prouver cette affirmation, nous devons examiner **le chemin critique**, qui détermine en réalité la fréquence d'horloge [25].

Le chemin critique dans un design pipeliné est le chemin le plus long entre deux étapes (qui est, entre deux registres) [1]. Nous allons caractériser ce chemin à l'aide de la figure 5.1. Alors, nous avons deux chemins à prendre en compte:

1. Commençons à partir du registre, passons par le biais de la fonction (*fctronde*), allons de Do à Di et parvenons au registre suivant.

2. Commençons à partir du registre, passons par le biais de la fonction (*fctronde*), aller de Do à Gi, allons à Go, alors vers Gi depuis la prochaine ronde, passons par la fonction *fctronde*, allons de Do à Di et finalement arrivons au prochain registre.

Nous concluons que le deuxième chemin est le plus long et peut être réduit davantage. En effet, le deuxième chemin a la plus longue distance à faire, mais ça va deux fois par le biais de la fonction *fctronde* (et le XOR qui a été fusionné dans la fonction *fctronde* dans la figure 5.1 par souci de simplicité).

Ici, nous introduisons un deuxième registre qui permettra de réduire le chemin critique à une étape (c'est-à-dire une ronde). Nous allons mettre les deux registres à la fin de la ronde. Par conséquent, nous nous attendons à avoir une grande augmentation de la vitesse.
Une modification utile à la conception doit ajouter une broche de remise à zéro *(reset pin)*. C'est vraiment facile d'implémenter et cela n'aura pas de grand impact sur la performance finale.
Un autre point à étudier serait d'implémenter les S-boxes avec des éléments ROM. La FPGA a certaines zones réservées pour ces éléments, comme vu sur la figure 5.2 suivante :

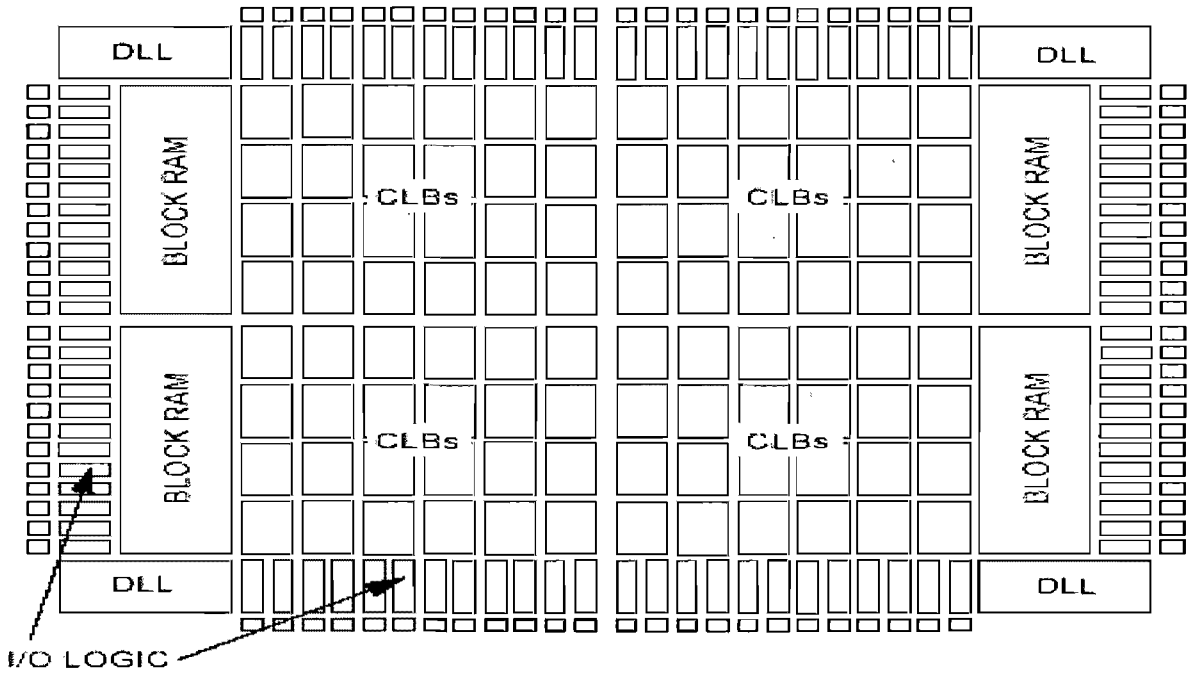

Figure 5.2 : les composants de FPGA Spartan III [4, 8]

ROM serait mieux implémentée dans des blocs de RAM au lieu d'utiliser le CLB, mais dans ce projet, nous n'irons pas aussi loin. Il serait intéressant d'utiliser cette technique si nous voulons avoir une conception entièrement optimisée. Cela réduirait bien sûr le nombre de CLBs utilisés, mais il peut aussi accélérer l'implémentation.
Ainsi pour ce projet, nous aurons confiance en synthétiseur pour avoir la meilleure implémentation des S-boxes.

Nous sommes maintenant prêts à tester notre design optimisé.

5.2 Processus de Simulation

La simulation est aujourd'hui devenue de rigueur et permet un gain de temps élevé par rapport à une mise au point totalement exécutée sur le matériel. Elle s'exécute sur la machine de développement [10].

Dans notre projet, pour faciliter la simulation du design, on a créé un fichier: *Test Bench* (banc d'essai *(testbench.vhd)),* dont le principe est de permettre la création d'un module de vérification qui affiche des messages d'erreur et effectue des tests sur la valeur des signaux dans le temps.

C'est, en quelques sortes, une boîte noire qui utilise le top design, lui donnant quelques vecteurs de simulation prédéterminés (le texte, la clé et l'horloge) et vérifiant si la sortie (texte encrypté) correspond à la validité du processus de cryptage.

Le *Testbench* donne en fait un nouveau texte (*plain text*) et une nouvelle clé tous les 16 cycles d'horloge et compare le résultat à la fin avec le texte encrypté associé.

Notre logiciel de simulation est *MODELSIM* qui fournit un environnement complet de simulation pour les designs complexes en FPGA. Il supporte plusieurs langages de description, dont le VHDL.

Notre architecture pipeline donne un bloc de chiffrement de 64 bits à chaque cycle d'horloge, sauf au début où le pipeline doit être rempli (pendant 16 cycles d'horloge, parce que nous avons 16 rondes), il y a donc une latence de 16 cycles d'horloge.
Nous pourrions donc écrire un autre banc d'essai qui donne un texte (*plain text*) pendant 16 cycle d'horloge et puis un texte chaque cycle d'horloge, mais ce ne serait pas pratique. En effet, à l'aide de *Testbench*, nous pouvons comparer immédiatement la clé et le texte brût qui ont toujours la même valeur calculée avec le procédé de chiffrement, comme nous le verrons dans la figure 5.3 correspondant aux signaux suivants :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity test_pipeline is end test_pipeline;
   architecture testbench of test_pipeline is
      component pipeline port
          (
                        pt          :       in      std_logic_vector(1 TO 64);
                       key          :       in   std_logic_vector(1 TO 64);
                        ct          :       out      std_logic_vector(1 TO 64);
                     reset          :        in       std_logic;
                       clk          :        in   std_logic
);
end component;
type test_vector is record
          key       :          std_logic_vector(1 to 64);
          pt        :          std_logic_vector(1 to 64);
          ct        :          std_logic_vector(1 to 64);
end record;
```

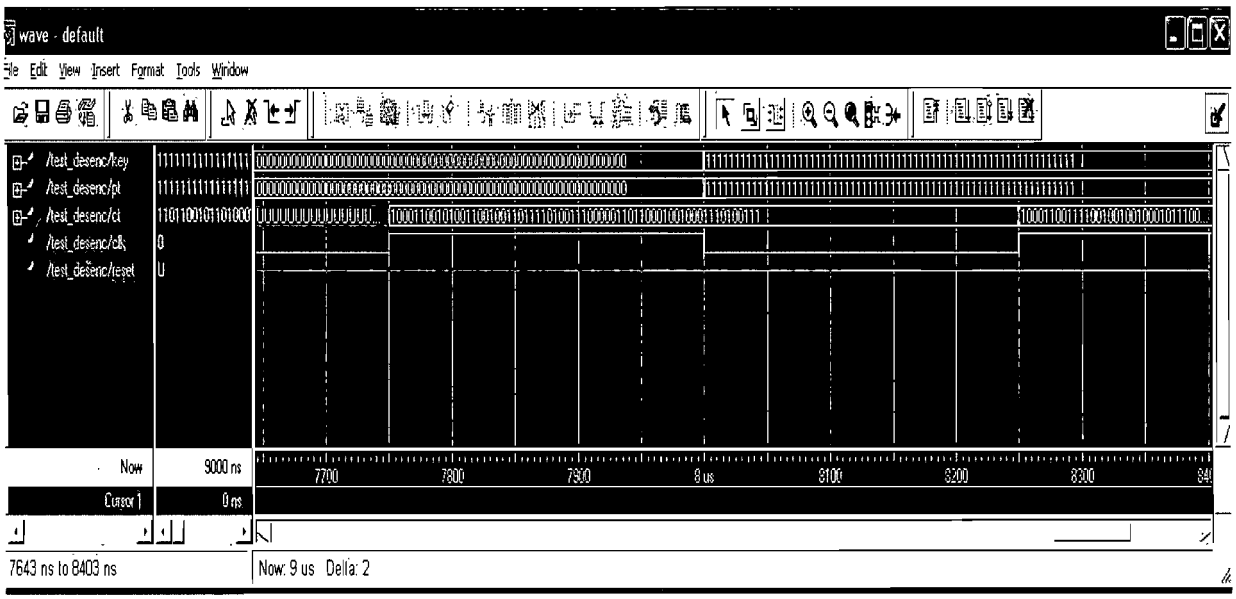

Figure 5.3 : la fenêtre « *wave* » du fonctionnement du design pipeliné

La première ligne dans la figure 5.3 est le texte (plain text), la deuxième est la clé, la troisième est le texte encrypté, la quatrième est l'horloge et la dernière est le reset.

L'algorithme de chiffrement est celui que nous avons prévu. Nous pouvons voir que, au cours des prochains cycles après le premier texte chiffré, d'autres procédés de chiffrement sont présents, mais ils ne signifient rien. Normalement, nous allons donner un procédé de chiffrement à chaque cycle d'horloge, de sorte qu'il ne serait pas ces "chiffrements parasites ".

Jusqu'à présent, le bon fonctionnement de notre design est bel et bien vérifié, nous pouvons maintenant passer au processus de synthèse.

5.3 Processus de synthèse

La synthèse détermine une réalisation du modèle RTL à base de cellules standardes (portes logiques combinatoires et séquentielles). Elle est usuellement gouvernée par un ensemble de contraintes définies séparément (surface, délais, consommation). Elle se base aussi sur une bibliothèque de cellules standardes spécifiques à une technologie donnée qui définit entre autres

pour chaque cellule sa fonction, sa surface, ses délais internes, sa consommation et ses contraintes d'environnement.

Le résultat de la synthèse est une description des instances de portes et de leurs interconnexions (*netlist*) qui peut être générée en VHDL. Le premier format sera utile pour la simulation post-synthèse. Le second format sera utile pour passer à l'étape de placement et routage. Il est aussi possible de générer un fichier SDF (*Standard Delay Format*) contenant les délais des portes.

Notre design est une synthèse visant la FPGA Spartan III XC2V2000 package FF896 speed -4. Les données d'exploitation sont illustrées dans le tableau 5.1

Tableau 5.1 : Résumé de données d'utilisation du système

Number of Slices:	2145	out of	10752	19%
Number of Slice Flip Flops:	024	out of	21504	4%
Number of 4 input LUTs:	4153	out of	21504	19%
Number of bonded IOBs:	194	out of	624	31%
Number of GCLKs:	1	out of	16	6%

Nous allons comparer la vitesse de notre design avec celle du design séquentiel.

Les résultats du design séquentiel sur FPGA Xilinx Spartan III XC2V2000 sont:

· Fréquence Max estimée: 41MHz

· Espace: environ 865 CLB's

Maintenant, nous allons exécuter la synthèse avec notre application et de comparer les résultats dans le tableau 5.2 suivant:

Tableau 5.2 : Résultats de deux implémentations (séquentielle et pipelinée)

	Fréquence Max. estimée (Mhz)	CLB Slices
Design sequential	41	865
Design pipeline	83	2145

Ainsi, le nouveau design a une vitesse beaucoup plus élevée. Nous pouvons expliquer la différence de vitesse avec les deux registres, dont près de multiplier la fréquence d'horloge par 2. Notre implémentation parvient à une fréquence plus élevée, même si elle a utilisé plus du CLBs.

C'est logique que l'implémentation pipelinée réalise un meilleur résultat que la séquentielle, parvenant à une fréquence plus élevée, mais avec plus de CLB. Pour comprendre pourquoi, nous devons comparer les deux implémentations. En effet, le secret est dans les S-Boxes (voir figure 5.4 et 5.5), le design pipeliné utilise plus de CLB. Plus précisément, le pipeliné utilise 25 CLB de FPGA tandis que le séquentiel utilise seulement 15 CLB. Parce que nous avons 8 S-boxes par ronde et 16 rondes, la différence est de **8x(25-15)x16 = 1280** de CLB. Ce qui explique exactement la différence d'espace entre les deux implémentations: **2145-865 = 1280**.

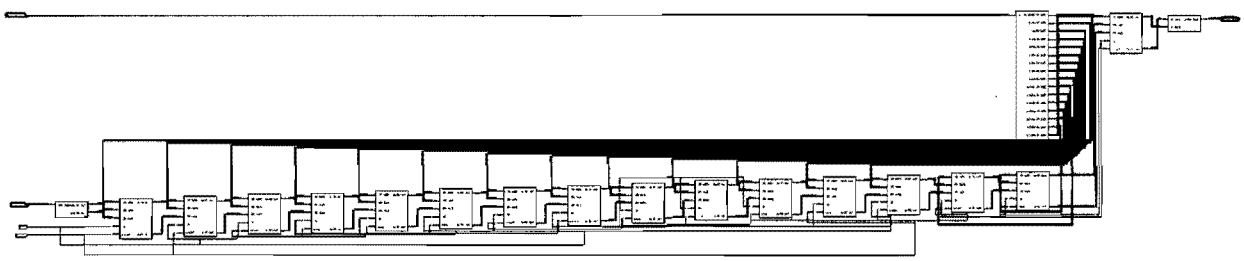

Figure 5.4 : Schéma de niveau supérieur du design pipeline

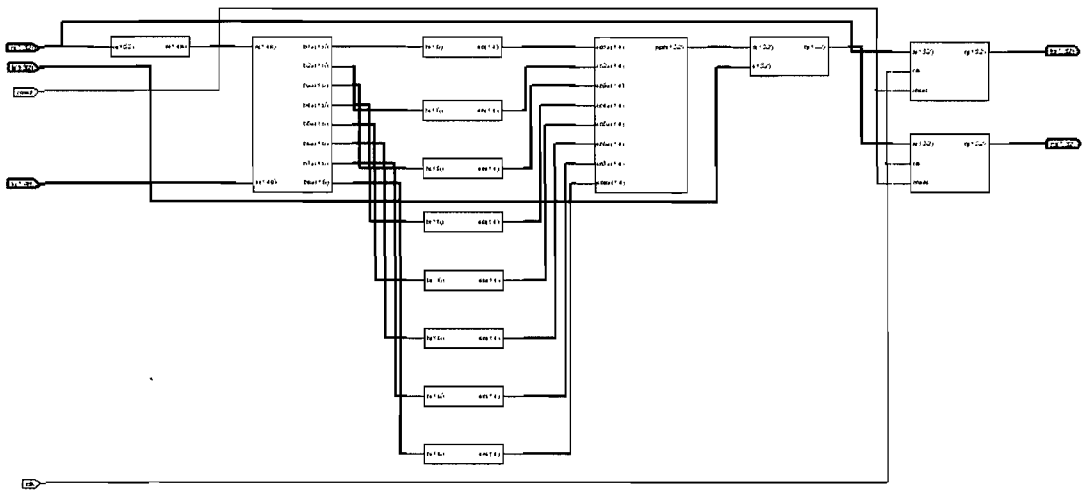

Figure 5.5 : Schéma d'une seule ronde du design pipeline

Nous pouvons ainsi conclure que notre design pipeline obtient une meilleure vitesse parce qu'il a trouvé une meilleure façon d'implémenter les S-boxes. On a utilisé plus de CLB, mais nous ne nous soucions pas puisque nous optimisons pour la vitesse.

Une autre conclusion est que l'implémentation des S-boxes est décisive pour réaliser un haut débit.

En général, le débit (*throughput*) est calculé comme suit:

Débit (*Throughput*) = (la fréquence x 64 bits) / 16

Parce que nous avons un bloc de chiffrement de 64 bits et 16 cycles d'horloge.

Ainsi dans notre cas, à chaque cycle d'horloge, nous avons :

- 2624 Mbits/s pour le design séquentiel.
- 5312 Mbits/s pour le design pipeline.

Donc, nous avons un design optimisé plus rapide de 202,4% du design séquentiel.

Nous avons maintenant un design optimisé, qui fonctionne correctement. Nous sommes donc prêts à passer au processus de placement et routage.

5.4 Placement et routage

L'étape suivante consiste à attribuer les cellules (CLB) du circuit à chaque équation délivrée par la projection et à définir les connexions. L'algorithme de placement place physiquement les différentes cellules et les chemins d'interconnexion dessinés entre les cellules afin de faciliter le routage. Des directives jointes à la *netlist* permettent une bonne répartition des cellules. Ces trois opérations sont réalisées par *FPGA editor* : outil de visualisation du placement et routage, disposant d'un placeur routeur intégré, et par *iMPACT* l'outil de *Xilinx ISE* pour télécharger directement le fichier *Bitstream* au FPGA.

Le résultat du placement et du routage est préparé de façon à pouvoir être directement téléchargé dans le FPGA (phase de mise au point) ou programmé dans une mémoire d'où il sera lu au moment de la configuration.

Dans notre projet, nous commençons par la génération du *fichier PROM* qui peut être écrit sur une plate-forme Flash de Spartan III, de sorte que la configuration du FPGA devient non volatile, c'est-à-dire la configuration est sauvegardée, même lorsque la carte est hors tension. Ensuite, nous allons programmer le design sur la FPGA (figure 5.6), sachant que la carte est liée à l'ordinateur via le câble JTAG.

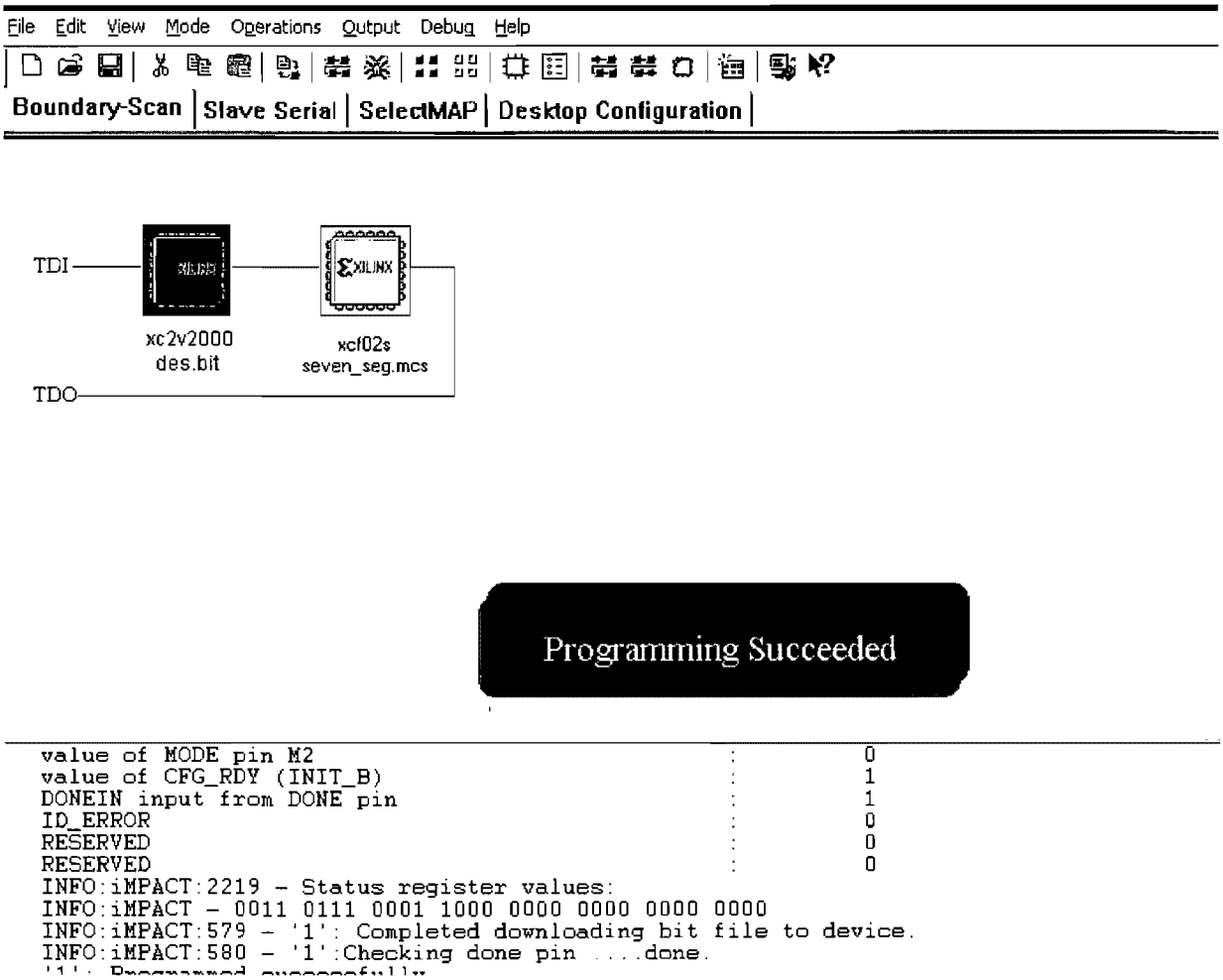

Figure 5.6 : Programmation du design sur FPGA

La fréquence finale de notre design devient très optimiste **92,3 Mhz**. Bien que la différence entre les deux fréquences (avant et après placement et routage) est minuscule.
Donc le débit final (*throughput*) est : **5907,2 Mbits/s.**

5.5 Déboggage et test direct sur FPGA

Déboggage (**ou** analyse logique) permet d'observer le comportement du FPGA dans son contexte et faire apparaître des cas critiques qui ne seraient pas apparus lors des simulations.
En effet, et après avoir téléchargé le fichier généré dans FPGA, nous allons le tester directement sur notre plateforme en utilisant *Chipscope* comme analyseur logique.

Comme l'ordinateur et le FPGA ne sont pas à la même échelle de temps, on règle un trigger qui va se déclencher selon certains paramètres des signaux observés. Une fois le trigger déclenché, les données sont stockées dans les blocks RAM du FPGA qui jouent le rôle de buffers. Lorsque le buffer est plein, les données sont téléchargées sur l'ordinateur.

On peut alors utiliser l'interface de CHIPSCOPE pour afficher les données sous forme de bus, ou de graphique (figure 5.7).

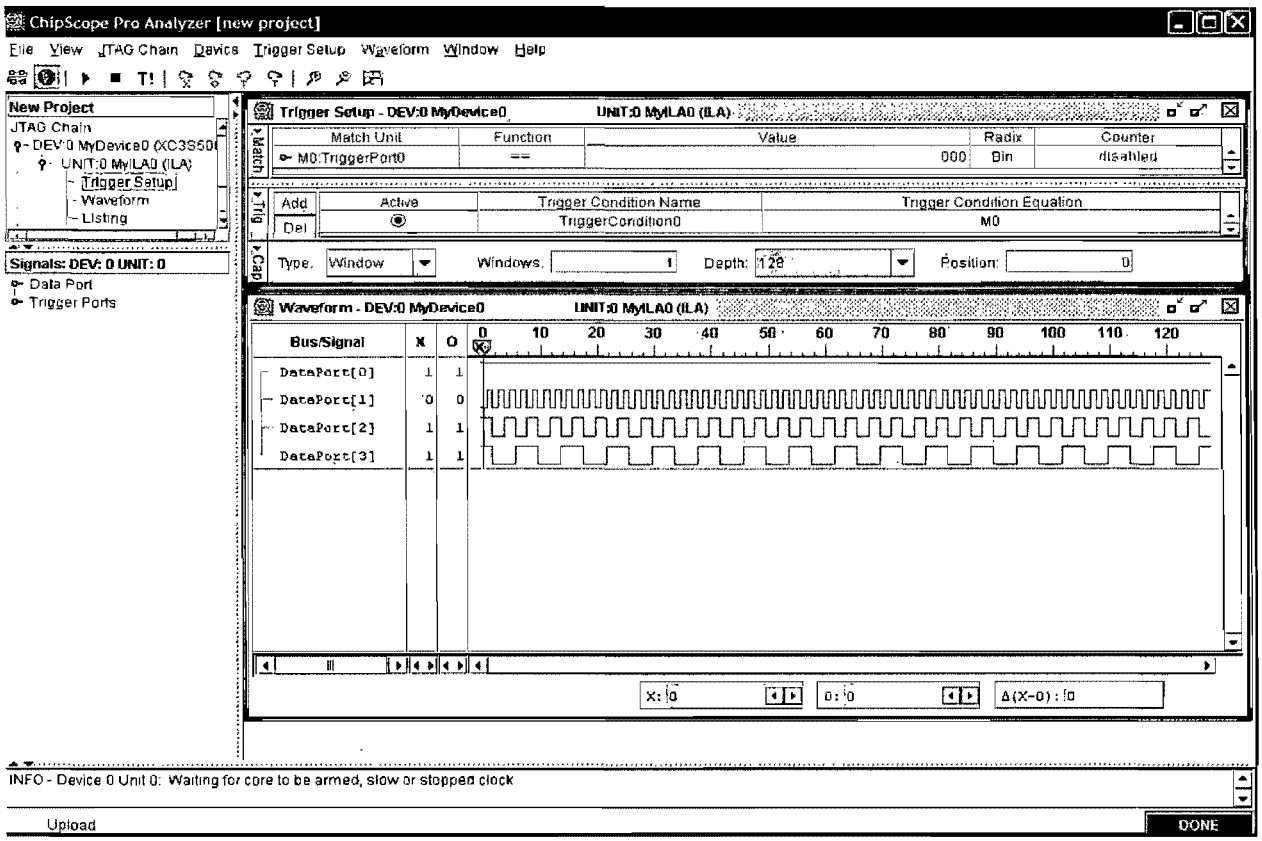

Figure 5.7 : fenêtre « wave » du design

En fin, nous arrivons à observer comment le signal se comporte sur la plateforme qui est très essentiel de répondre à nos questions vis-à-vis de l'optimisation, parce que, c'est le signal qui est directement exploité sur FPGA. Par contre, les signaux Modelsim ne peuvent être observés qu'après l'utilisation du banc d'essai (*Testbench*) fourni avec l'exemple du projet.

5.6 Evaluation de performance

L'implémentation sur FPGA de l'algorithme DES a été accomplie sur un Spartan III XC2V2000 package FF896 speed -4, utilisant Xilinx ISE 7.1i comme outil de synthèse, ModelSim (PE Student Edition 6.4e) pour la simulation, et Xilinx ChipScope Pro 7.1i comme analyseur logique pour le déboggage. Le design a été codé en utilisant VHDL. Il a occupé 2145(19%) CLB slices, 24(4%) de bascules (slice Flip Flops), et 129 (41%) entrée/sortie (I/O).
L'implémentation réalise une fréquence de 92.3 MHz. Par conséquent, le débit (*throughput*) atteint est 5907,2 Mbits/s.

Le tableau 5.3 montre la performance de certaines implémentations matérielles de DES. Notons que les résultats obtenus sont compétitifs avec les implémentations existantes. Néanmoins, la comparaison de ces résultats est strictement ***relative***, parce que ces implémentations matérielles ne sont pas toutes faites sur la même plateforme FPGA (Spartan, par exemple, est moins puissant que Virtex). Par conséquent, iI est difficile de donner une juste comparaison à cet égard, mais, quoi qu'il arrive, nos résultats montrent clairement que les implémentations sur FPGA de l'algorithme DES sont très attrayantes pour de nombreuses applications.

Plusieurs implémentations FPGA de DES ont été annoncées dans la littérature (ETAT DE L'ART) réalisant des débits de 26 à 21811,2 Mbits/s utilisant des stratégies de conception différentes. L'implémentation de DES à [22] utilise l'approche pipeline dans le mode ECB et réalise un débit de 3052 Mbits/s. L'implémentation Jbits de DES est effectuée sur FPGA Virtex XCV150 [24], elle a réalisé une vitesse de cryptage de 10752 Mbits/s. L'implémentation DES sur FPGA XC4028EX à [21] utilise les deux techniques Déroulement de boucle et Pipeline à deux étapes (2-stages) et à quatre étapes (4-stages) et a obtenu un débit de 183,8 Mbits/s et 402,7 Mbits/s successivement. L'approche *pipeline,* utilisée au projet technique [25] pour implémenter

le DES sur la plateforme FPGA Xilinx Virtex II, est achevée réalisant un débit de 21811 Mbits/s avec 8453 de CLB. Le design à [27] est implémenté sur un FPGA VirtexE XCV400e, comme une stratégie visant à réduire le chemin critique du design associé, il a utilisé une *structure parallèle* qui a permis de calculer toutes les huit DES S-boxes simultanément. Le résultat est très

important tant que le débit ne dépasse pas 274 Mbits/s occupant seulement 117 CLB. D'après tout cela, je me suis persuadé que le résultat de notre travail est bel et bien compétitif.

Tableau 5.3 : comparaison des implémentations matérielles de DES sur FPGA

Design	Plateforme	CLB	Fréquence (MHz)	Débit (Mbits/s)
[21]	XC4028EX	741	25.18	402.7
[22]	XCV400	5263	47.7	3052
[23]	XCV1000	6446	59.5	3808
[24]	XCV150	1584	168	10752
[25]	VIRTEX-II	8453	333	21811,2
[27]	XCV400E	117	68.05	274
Notre travail	**XC2V2000**	**2145**	**92.3**	**5907,2**

Conclusion :

Dans le chapitre 3 (ETAT DE L'ART), nous avons constaté que la majorité de travaux s'est basée sur le rapport : Débit/Espace (*throughput/Area*), visant un grand débit mais avec un minimum de ressources.

Quant à notre travail, nous ne nous sommes focalisés que sur un seul facteur qui est la *rapidité*, parce que de nos jours, nous disposons de larges de ressources, mais notre souci est le temps.

C'est à dire: comment exploiter notre existant (ressources) pour gagner plus de temps?, et ce conformément à la devise des applications de notre monde aujourd'hui qui dit : *plus vite, plus haut, plus fort.*

Chapitre 6

CONCLUSION ET PERSPECTIVES

Nous avons présenté une implémentation matérielle optimisée de l'algorithme cryptographique DES (*Data Encryption Standard*) sur une plate-forme reconfigurable basée sur FPGA (*Field Programmable Gate Arrays*). Notre conception a été effectuée sur une *Spartan III XC2V2000*, et ce parce que l'implémentation matérielle est, par nature, plus sécurisée physiquement, plus rapide (haut débit), plus fiable, et surtout les FPGAs sont des périphériques matériels dont la fonction n'est pas fixée, et qui peuvent être programmées dans le système (in-system). C'est donc une alternative prometteuse pour l'implémentation de chiffrement à bloc.

Notre approche pipeline met l'accent sur les S-boxes, qui correspondent à un grand cas en VHDL, et qui seront implémentées dans des LUTs de FPGA, ce qui permet d'atteindre des résultats, tout à fait, compétitifs par rapport à d'autres implémentations matérielles de DES.

En fait, l'architecture pipeline est très efficace pour parvenir à un haut débit, mais elle consomme beaucoup de ressources.

Pour les perspectives, la réalisation du rapport : **débit/ressources** (*throughput/Area*), visant un grand débit mais avec un minimum de ressources, demeure une extension potentielle de nos travaux. En effet, l'architecture pipeline gaspille beaucoup d'espace, car elle utilise 16 fois la même ronde. Le défi est de trouver une façon d'utiliser seulement une ronde et rendre les données en boucle 16 fois. Cette idée peut être implémentée avec une machine à état.

En outre, nous ne devrions pas limiter la portée de ce projet à l'algorithme cryptographique DES, parce que c'était seulement une étude de cas pour une plate-forme reconfigurable FPGA. Le facteur de performance a été prouvé et maintenant nous pouvons implémenter d'autres

algorithmes cryptographiques, et de préférence ceux qui prendront un grand avantage dans une implémentation matérielle, tels que : AES, 3DES, ainsi que les systèmes cryptographiques à base de courbes elliptiques.

Chapitre 7

BIBLIOGRAPHIE

[1] I. E. Bennour, et E. M. Aboulhamid. "*Les problèmes d'ordonnancement cycliques dans la synthèse des systèmes numériques*". Université de Montréal, Montréal, Publication 996, Oct. 1995. http://www.iro.umontreal.ca/~aboulham/pipeline.pdf

[2] Aj. Elbirt, W. Yip, B. Chetwynd, C. Paar. "*An FPGA-Based Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists*" .In: IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Volume 9, pages 545-557, Worcester, USA, 2001.

[3] E. Oswald, M. Feldhofer, K Lemke. "*State of the Art in Hardware Architectures*". Rapport technique, *European Network of Excellence in Cryptology,* Institute for Applied Information Processing and Communications, September 2005.

[4] Xilinx. Spartan 3 Family Overview, février 2006

[5] B_ Schneier, "*Applied Cryptography*", New York, New York, USA : John Wiley & Sons Inc., 2nd ed., 1996.

[6] FPGA : http://fr.wikipedia.org/wiki/PLA#FPGA

[7] National instruments: http://www.ni.com/

[8] XILINX: http://www.xilinx.com/

[9] Fabien Germain. "*Sécurité cryptographique par la conception spécifique de circuits intégrés*". Thèse Ph.D, Ecole Polytechnique, Paris VI, Juin 2006.

[10] Rabie Ben Atitallah. "*Modèles et simulation de systèmes sur puce multiprocesseurs Estimationdes performances et de la consommation d'énergie''*. Thèse de doctorat, Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, France. Décembre 2007.

[11] R. Doud, "*Hardware Crypto Solutions Boost VPN",* Electronic Engineering Times, pp. 57-64, April 1999.

.[12] Daniel Gomes MESQUITA. "*Architectures reconfigurables et cryptographie : une analysede robustesse et contremesures face aux attaques par canaux cachés*". Thèse de doctorat, Laboratoire des Systèmes Automatiques et Microélectroniques, UNIVERSITE MONTPELLIER II, France, Novembre 2006.

[13] François-Raymond Boyer. "*Optimisation lors de la synthèse de circuits à partir de langages de haut niveau*". Thèse de doctorat, DIRO, Faculté des arts et des sciences, Université de Montréal, Avril 2001.

[14] A. Lagger, "*Self-reconfigurable platform for cryptographic application*". *MASTER THESIS,* SCHOOL OF COMPUTER AND COMMUNICATION SCIENCES, SWISS FEDERAL INSTITUTE OF TECHNOLOGY LAUSANNE, Swiss. Février 2006.

[15] D. Craig Wilcox, Lyndon G. Pierson, Perry J. Robertson, Edward L. Witzke et Karl Gass. "*A DES ASIC Suitable for Network Encryption at 10 Gbps and Beyond*". In Cryptographic Hardware and Embedded Systems, Volume 1717/1999, page 725, Utah State University, Albuquerque, New Mexico, Juin 1999.

[16] Susan Landau, "*Standing the Test of Time: The Data Encryption Standard*". NOTICES OF THE AMS, Sun Microsystems Inc, VOLUME 47, NUMBER 3, 2000 Mars.

[17] David C. Feldmeier. "*A high-Speed Software DES Implementation*". In: Computer Communication Research Group, Bellcore, Morristown, NJ. Juin 1989

[18] Wilcox, D., Pierson, L., Robertson, P., Witzke, E.L., Gass, K. "*A DES asi suitable for network encryption at 10 Gbs and beyond*". CHESS 99, LNCS 717 ,pages 37–48, 1999

[19] R. Stephen Preissig." Data Encryption Standard (DES) Implementation on the TMS320C6000". In: Texas Instruments Incorporated, Application Report, SPRA702, November 2000.

[20] Dag Arne Osvik. "*Efficient Implementation of the Data Encryption Standard*". Thesis submitted for the degree of Candidatus Scientiarum. Institutt for Informatikk, Universitas Bergensis, Høyteknologisenteret, Norway. April 2003.

[21] Kaps, J., Paar, C. "*Fast DES implementations for FPGAs and its application to a Universal key-search machine*". In: Proc. 5th Annual Workshop on selected areas in cryptography-Sac' 98, Ontario, Canada, Springer-Verlag, pages 234–247, 1998.

[22] Core(2000), F.D.: (2000) URL: http://www.free-ip.com/DES/.

[23] McLoone, M., McCanny, J.: "*High-performance FPGA implementation of DES using a novel method for implementing the key schedule*". IEE Proc.: Circuits, Devices & Systems 150 (2003) 373–378. 2003.

[24] Patterson, C.: "*High Performance DES Encryption in Virtex FPGAs using Jbits*". In: Field-programmable custom computing machines, FCCM' 00, Napa Valley, CA, USA, IEEE Comput. Soc., CA, USA, (2000) 113–121, 2000.

[25] G. Rouvroy, FX. Standaert, JJ. Quisquater, JD. Legat. "*Efficient Uses of FPGA's for Implementations of DES and its Experimental Linear Cryptanalysis*".In IEEE Transactions on Computers, Special CHES Edition, pages 473-482, April 2003.

[26] G. Rouvroy, F. Standaert, J. Quisquater, J. Legat "*Efficient Uses of FPGAs for Implementations of DES and Its Experimental Linear Cryptanalysis*". In: IEEE Computer Society Washington, DC, USA, Pages: 473 – 482, April 2003.

[27] Wong, K, Wark, M, Dawson, E. "*A single-chip FPGA implementation of the data encryption standard(DES) algorithm*". In : Global Telecommunications Conference, 1998. GLOBECOM 98. The Bridge to Global Integration. IEEE Volume 2, Issue , 1998 Page(s):827 - 832 vol.2. 2003

[28] Chin Mun Wee, Peter R. Sutton and Neil, W. Bergmann. *"AN FPGA NETWORK ARCHITECTURE FOR ACCELERATING 3DES – CBC"*. In: School of Information Technology and Electrical Engineering, University of Queensland, Brisbane, Australia. 2005.

[29] David B. Tang. "*A High Speed DES Implementation with Reconfigurable S-boxes for New Emerging Network Applications Based on FPGAs*". Thesis for ENGINEERING DEGREE(E.E). Faculty of the Stevens Institute of Technology, MI, USA, 2000.

[30] J. Kaps, C. Paar. "*Fast DES Implementations for FPGAs and Its Application to a Universal Key-Search Machine*". In: Selected Areas in Cryptography, Electrical and Computer Engineering Department, Worcester Polytechnic Institute. Page 630, Volume 1556/1999. Worcester, MA, USA. Janvier 1999.

[31] J. Deepakumara, Howard M. Heys and R. Venkatesan. *"FPGA IMPLEMENTATION OF MD5 HASH ALGORITHM"*. Faculty of Engineering and Applied Science, Memorial University of Newfoundland, St.John's, NF, Canada. 2001.

[32] Jens Peter Kaps. *"High speed FPGA Architectures for the Data Enryption Standard"*. Thesis of Master of Science in Electrical Engineering. Faculty of the WORCESTER POLYTECHNIC INSTITUTE, USA, Mai 1998.

[33] A. Lagger, "*Implementation of DES Algorithm Using FPGA Technology*". Microelectronic Systems Laboratory, EPFL, 2002.

[34] Pierre-Yvan LIARDET. "*Ingénierie cryptographique Implantations sécurisées*". Thèse de doctorat, Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier, UNIVERSITE MONTPELLIER II, UNIVERSITE MONTPELLIER II, France, Juillet 2006.

[35] Christophe Giraud. "*Attaques de cryptosystèmes embarqués et contre-mesures associées*". Thèse de doctorat, Université de Versailles Saint-Quentin-en-Yvelines, France, Octobre 2007.

[36] Sébastien LE BEUX. "*Un flot de conception pour applications de traitement du signal systématique implémentées sur FPGA à base d'Ingénierie Dirigée par les Modèles*". Thèse de doctorat, UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE, France, Décembre 2007.

ANNEXE I: Code VHDL pour l'architecture séquentielle

```
--------------------------------------------------------------------------------
--      Autheur:        Mohamed AMOUD, [information retirée / information withdrawn]   --
--                                                                              --
--      Projet:         Design et implémentation sur FPGA d'un algorithme DES   --
--                                                                              --
--      Version :       Architecture séquentielle (une seule ronde)             --
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity desenc is port
(
        pt      :       in      std_logic_vector(1 TO 64);
        key     :       in      std_logic_vector(1 TO 64);
        ct      :       out     std_logic_vector(1 TO 64);
        reset   :       in      std_logic;
        clk     :       in      std_logic
);
end desenc;

architecture behavior of desenc is
        signal  k1x :   std_logic_vector(1 to 48);
        signal e1 :     std_logic_vector(1 to 48);
        signal  l0x,r0x, ppo, l1x,r1x       :       std_logic_vector(1 to 32);
        signal  XX      :       std_logic_vector(1 to 56);
  signal  YY      :       std_logic_vector(1 to 56);
  signal  c1x, d1x, c0x, d0x:     std_logic_vector(1 to 28);
  signal b1x,b2x,b3x,b4x,b5x,b6x,b7x,b8x :  std_logic_vector (1 TO 6);

  signal so1x,so2x,so3x,so4x,so5x,so6x,so7x,so8x :   std_logic_vector(1 to 4);




begin

        process

        begin
```

```
            if (clk'event and clk='1') then
--génération de la clé
--permutation pc1

XX(1)<=key(57);  XX(2)<=key(49);  XX(3)<=key(41);  XX(4)<=key(33);  XX(5)<=key(25);  XX(6)<=key(17);
        XX(7)<=key(9);
XX(8)<=key(1);XX(9)<=key(58);XX(10)<=key(50);XX(11)<=key(42);XX(12)<=key(34);XX(13)<=key(26);
        XX(14)<=key(18);
XX(15)<=key(10); XX(16)<=key(2);  XX(17)<=key(59); XX(18)<=key(51); XX(19)<=key(43); XX(20)<=key(35);
        XX(21)<=key(27);
XX(22)<=key(19); XX(23)<=key(11); XX(24)<=key(3);          XX(25)<=key(60); XX(26)<=key(52);
        XX(27)<=key(44);        XX(28)<=key(36);
XX(29)<=key(63); XX(30)<=key(55); XX(31)<=key(47); XX(32)<=key(39); XX(33)<=key(31); XX(34)<=key(23);
        XX(35)<=key(15);
XX(36)<=key(7);          XX(37)<=key(62); XX(38)<=key(54); XX(39)<=key(46); XX(40)<=key(38); XX(41)<=key(30);
        XX(42)<=key(22);
XX(43)<=key(14); XX(44)<=key(6);          XX(45)<=key(61); XX(46)<=key(53); XX(47)<=key(45); XX(48)<=key(37);
        XX(49)<=key(29);
XX(50)<=key(21); XX(51)<=key(13); XX(52)<=key(5);          XX(53)<=key(28); XX(54)<=key(20); XX(55)<=key(12);
        XX(56)<=key(4);

--deux blocs

c0x<=XX(1 to 28); d0x<=XX(29 to 56);

--les deux blocs subissent une rotation à gauche

c1x<=To_StdLogicVector(to_bitvector(c0x) rol 1);    d1x<=To_StdLogicVector(to_bitvector(d0x) rol 1);
        YY(1 to 28)<=c1x;          YY(29 to 56)<=d1x;

-- une permutation CP-2 fournissant en sortie un bloc de 48 bits représentant la clé Ki


k1x(1)<=YY(14);  k1x(2)<=YY(17);  k1x(3)<=YY(11);  k1x(4)<=YY(24);  k1x(5)<=YY(1);   k1x(6)<=YY(5);
k1x(7)<=YY(3);   k1x(8)<=YY(28);  k1x(9)<=YY(15);  k1x(10)<=YY(6);  k1x(11)<=YY(21); k1x(12)<=YY(10);
k1x(13)<=YY(23); k1x(14)<=YY(19); k1x(15)<=YY(12); k1x(16)<=YY(4);  k1x(17)<=YY(26); k1x(18)<=YY(8);
k1x(19)<=YY(16); k1x(20)<=YY(7);  k1x(21)<=YY(27); k1x(22)<=YY(20); k1x(23)<=YY(13); k1x(24)<=YY(2);
k1x(25)<=YY(41); k1x(26)<=YY(52); k1x(27)<=YY(31); k1x(28)<=YY(37); k1x(29)<=YY(47); k1x(30)<=YY(55);
k1x(31)<=YY(30);        k1x(32)<=YY(40);        k1x(33)<=YY(51);        k1x(34)<=YY(45);
        k1x(35)<=YY(33);        k1x(36)<=YY(48);

k1x(37)<=YY(44); k1x(38)<=YY(49); k1x(39)<=YY(39); k1x(40)<=YY(56); k1x(41)<=YY(34); k1x(42)<=YY(53);
k1x(43)<=YY(46); k1x(44)<=YY(42); k1x(45)<=YY(50); k1x(46)<=YY(36); k1x(47)<=YY(29); k1x(48)<=YY(32);

--permutation initiale

l0x(1)<=pt(58);          l0x(2)<=pt(50);          l0x(3)<=pt(42);          l0x(4)<=pt(34);
l0x(5)<=pt(26);          l0x(6)<=pt(18);          l0x(7)<=pt(10);          l0x(8)<=pt(2);
l0x(9)<=pt(60);  l0x(10)<=pt(52); l0x(11)<=pt(44); l0x(12)<=pt(36);
l0x(13)<=pt(28); l0x(14)<=pt(20); l0x(15)<=pt(12); l0x(16)<=pt(4);
l0x(17)<=pt(62); l0x(18)<=pt(54); l0x(19)<=pt(46); l0x(20)<=pt(38);
l0x(21)<=pt(30); l0x(22)<=pt(22); l0x(23)<=pt(14); l0x(24)<=pt(6);
```

```
l0x(25)<=pt(64); l0x(26)<=pt(56); l0x(27)<=pt(48); l0x(28)<=pt(40);
l0x(29)<=pt(32); l0x(30)<=pt(24); l0x(31)<=pt(16); l0x(32)<=pt(8);

r0x(1)<=pt(57);  r0x(2)<=pt(49);  r0x(3)<=pt(41);  r0x(4)<=pt(33);
r0x(5)<=pt(25);  r0x(6)<=pt(17);  r0x(7)<=pt(9);   r0x(8)<=pt(1);
r0x(9)<=pt(59);  r0x(10)<=pt(51); r0x(11)<=pt(43); r0x(12)<=pt(35);
r0x(13)<=pt(27); r0x(14)<=pt(19); r0x(15)<=pt(11); r0x(16)<=pt(3);
r0x(17)<=pt(61); r0x(18)<=pt(53); r0x(19)<=pt(45); r0x(20)<=pt(37);
r0x(21)<=pt(29); r0x(22)<=pt(21); r0x(23)<=pt(13); r0x(24)<=pt(5);
r0x(25)<=pt(63); r0x(26)<=pt(55); r0x(27)<=pt(47); r0x(28)<=pt(39);
r0x(29)<=pt(31); r0x(30)<=pt(23); r0x(31)<=pt(15); r0x(32)<=pt(7);

--fct expansion

e1(1)<=r0x(32); e1(2)<=r0x(1);  e1(3)<=r0x(2);  e1(4)<=r0x(3);  e1(5)<=r0x(4);  e1(6)<=r0x(5);
        e1(7)<=r0x(4); e1(8)<=r0x(5);
e1(9)<=r0x(6);  e1(10)<=r0x(7); e1(11)<=r0x(8); e1(12)<=r0x(9); e1(13)<=r0x(8); e1(14)<=r0x(9);
        e1(15)<=r0x(10); e1(16)<=r0x(11);
e1(17)<=r0x(12); e1(18)<=r0x(13); e1(19)<=r0x(12); e1(20)<=r0x(13); e1(21)<=r0x(14); e1(22)<=r0x(15);
        e1(23)<=r0x(16); e1(24)<=r0x(17);
e1(25)<=r0x(16); e1(26)<=r0x(17); e1(27)<=r0x(18); e1(28)<=r0x(19); e1(29)<=r0x(20); e1(30)<=r0x(21);
        e1(31)<=r0x(20); e1(32)<=r0x(21);
e1(33)<=r0x(22); e1(34)<=r0x(23); e1(35)<=r0x(24); e1(36)<=r0x(25); e1(37)<=r0x(24); e1(38)<=r0x(25);
        e1(39)<=r0x(26); e1(40)<=r0x(27);
e1(41)<=r0x(28); e1(42)<=r0x(29); e1(43)<=r0x(28); e1(44)<=r0x(29); e1(45)<=r0x(30); e1(46)<=r0x(31);
        e1(47)<=r0x(32); e1(48)<=r0x(1);

--fct xor1

        XX<=k1x xor e1;
b1x<=XX(1 to 6);
b2x<=XX(7 to 12);
b3x<=XX(13 to 18);
b4x<=XX(19 to 24);
b5x<=XX(25 to 30);
b6x<=XX(31 to 36);
b7x<=XX(37 to 42);
b8x<=XX(43 to 48);

--fct substitution
--s1
case b1x is
when "000000"=>     so1x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "000010"=>     so1x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "000100"=>     so1x<=To_StdLogicVector(Bit_Vector'(x"d"));


when "000110"=>     so1x<=To_StdLogicVector(Bit_Vector'(x"1"));

when "001000"=>     so1x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "001010"=>     so1x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "001100"=>     so1x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "001110"=>     so1x<=To_StdLogicVector(Bit_Vector'(x"8"));
```

```
when "010000"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "010010"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "010100"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "010110"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "011000"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "011010"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "011100"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "011110"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "000001"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "000011"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "000101"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "000111"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "001001"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "001011"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "001101"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "001111"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "010001"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "010011"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "010101"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "010111"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "011001"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "011011"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "011101"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "011111"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "100000"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "100010"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "100100"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "100110"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "101000"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "101010"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "101100"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "101110"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "110000"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "110010"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "110100"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "110110"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "111000"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "111010"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "111100"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "111110"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "100001"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "100011"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "100101"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "100111"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "101001"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "101011"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "101101"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "101111"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "110001"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "110011"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"b"));

when "110101"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "110111"=>      so1x<=To_StdLogicVector(Bit_Vector'(x"e"));
```

```
when "111001"=>         so1x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "111011"=>         so1x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "111101"=>         so1x<=To_StdLogicVector(Bit_Vector'(x"6"));
when others=>           so1x<=To_StdLogicVector(Bit_Vector'(x"d"));
end case;

--s2
case b2x is
when "000000"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "000010"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "000100"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "000110"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "001000"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "001010"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "001100"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "001110"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "010000"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "010010"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "010100"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "010110"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "011000"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "011010"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "011100"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "011110"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "000001"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "000011"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "000101"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "000111"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "001001"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "001011"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "001101"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "001111"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "010001"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "010011"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "010101"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "010111"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "011001"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "011011"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "011101"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "011111"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "100000"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "100010"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "100100"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "100110"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "101000"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "101010"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "101100"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "101110"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "110000"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "110010"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "110100"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "110110"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "111000"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"9"));
```

```
when "111010"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "111100"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "111110"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "100001"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "100011"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "100101"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "100111"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "101001"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "101011"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "101101"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "101111"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "110001"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "110011"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "110101"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "110111"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "111001"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "111011"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "111101"=>         so2x<=To_StdLogicVector(Bit_Vector'(x"e"));
when others=>           so2x<=To_StdLogicVector(Bit_Vector'(x"9"));
end case;

--s3

case b3x is

when "000000"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "000010"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "000100"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "000110"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "001000"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "001010"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "001100"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "001110"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "010000"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "010010"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "010100"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "010110"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "011000"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "011010"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "011100"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "011110"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "000001"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "000011"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "000101"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "000111"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "001001"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "001011"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "001101"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "001111"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "010001"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "010011"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "010101"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "010111"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"e"));
```

```
when "011001"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "011011"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "011101"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "011111"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "100000"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "100010"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "100100"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "100110"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "101000"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "101010"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "101100"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "101110"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "110000"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "110010"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "110100"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "110110"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "111000"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "111010"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "111100"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "111110"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "100001"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "100011"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "100101"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "100111"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "101001"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "101011"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "101101"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "101111"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "110001"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "110011"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "110101"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "110111"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "111001"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "111011"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "111101"=>         so3x<=To_StdLogicVector(Bit_Vector'(x"2"));
when others=>           so3x<=To_StdLogicVector(Bit_Vector'(x"c"));
end case;

--s4
case b4x is
when "000000"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "000010"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "000100"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "000110"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "001000"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "001010"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "001100"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "001110"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "010000"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "010010"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "010100"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "010110"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "011000"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"b"));
```

```
when "011010"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "011100"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "011110"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "000001"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "000011"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "000101"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "000111"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "001001"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "001011"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "001101"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "001111"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "010001"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "010011"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "010101"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "010111"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "011001"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "011011"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "011101"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "011111"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "100000"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "100010"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "100100"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "100110"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "101000"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "101010"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "101100"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "101110"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "110000"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "110010"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "110100"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "110110"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "111000"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "111010"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "111100"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "111110"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "100001"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "100011"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "100101"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "100111"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "101001"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "101011"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "101101"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "101111"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "110001"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "110011"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "110101"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "110111"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "111001"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "111011"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "111101"=>         so4x<=To_StdLogicVector(Bit_Vector'(x"2"));
when others=>           so4x<=To_StdLogicVector(Bit_Vector'(x"e"));
end case;
```

```
--s5
case b5x is
when "000000"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "000010"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "000100"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "000110"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "001000"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "001010"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "001100"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "001110"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "010000"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "010010"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "010100"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "010110"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "011000"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "011010"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "011100"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "011110"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "000001"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "000011"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "000101"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "000111"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "001001"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "001011"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "001101"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "001111"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "010001"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "010011"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "010101"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "010111"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "011001"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "011011"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "011101"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "011111"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "100000"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "100010"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "100100"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "100110"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "101000"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"a"));

when "101010"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "101100"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "101110"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "110000"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "110010"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "110100"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "110110"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "111000"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "111010"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "111100"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "111110"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "100001"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "100011"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"8"));
```

```
when "100101"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "100111"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "101001"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "101011"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "101101"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "101111"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "110001"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "110011"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "110101"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "110111"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "111001"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "111011"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "111101"=>         so5x<=To_StdLogicVector(Bit_Vector'(x"5"));
when others=>           so5x<=To_StdLogicVector(Bit_Vector'(x"3"));
end case;

--s6
case b6x is
when "000000"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "000010"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "000100"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "000110"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "001000"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "001010"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "001100"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "001110"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "010000"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "010010"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "010100"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "010110"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "011000"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "011010"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "011100"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "011110"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "000001"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "000011"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "000101"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "000111"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "001001"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "001011"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "001101"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "001111"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "010001"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "010011"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "010101"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "010111"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "011001"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "011011"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "011101"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "011111"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "100000"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "100010"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "100100"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"f"));
```

```
when "100110"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "101000"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "101010"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "101100"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "101110"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "110000"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "110010"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "110100"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "110110"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "111000"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "111010"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "111100"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "111110"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "100001"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "100011"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "100101"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "100111"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "101001"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "101011"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "101101"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "101111"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "110001"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "110011"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "110101"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "110111"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "111001"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "111011"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "111101"=>         so6x<=To_StdLogicVector(Bit_Vector'(x"8"));
when others=>           so6x<=To_StdLogicVector(Bit_Vector'(x"d"));
end case;
--s7
case b7x is
when "000000"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "000010"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "000100"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "000110"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "001000"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "001010"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "001100"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "001110"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "010000"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "010010"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "010100"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "010110"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "011000"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "011010"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "011100"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "011110"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"1"));when "000001"=>
        so7x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "000011"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "000101"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "000111"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "001001"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"4"));
```

```
when "001011"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "001101"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "001111"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "010001"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "010011"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "010101"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "010111"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "011001"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "011011"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "011101"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "011111"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "100000"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "100010"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "100100"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "100110"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "101000"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "101010"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "101100"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "101110"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "110000"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "110010"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "110100"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "110110"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "111000"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "111010"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "111100"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "111110"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "100001"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "100011"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "100101"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "100111"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "101001"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "101011"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "101101"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "101111"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "110001"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "110011"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "110101"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "110111"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "111001"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "111011"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "111101"=>         so7x<=To_StdLogicVector(Bit_Vector'(x"3"));
when others=>           so7x<=To_StdLogicVector(Bit_Vector'(x"c"));
end case;
--s8
case b8x is
when "000000"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "000010"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "000100"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "000110"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "001000"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "001010"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "001100"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"b"));
```

```
when "001110"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "010000"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "010010"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "010100"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "010110"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "011000"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "011010"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "011100"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "011110"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "000001"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "000011"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "000101"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "000111"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "001001"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "001011"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "001101"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "001111"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "010001"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "010011"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "010101"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "010111"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "011001"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "011011"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "011101"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "011111"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "100000"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "100010"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"b"));
when "100100"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "100110"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "101000"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"9"));
when "101010"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "101100"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "101110"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "110000"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "110010"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"6"));
when "110100"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "110110"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "111000"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "111010"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "111100"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "111110"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "100001"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"2"));
when "100011"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"1"));
when "100101"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"e"));
when "100111"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"7"));
when "101001"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"4"));
when "101011"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"a"));
when "101101"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"8"));
when "101111"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"d"));
when "110001"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"f"));
when "110011"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"c"));
when "110101"=>         so8x<=To_StdLogicVector(Bit_Vector'(x"9"));
```

```
when "110111"=>          so8x<=To_StdLogicVector(Bit_Vector'(x"0"));
when "111001"=>          so8x<=To_StdLogicVector(Bit_Vector'(x"3"));
when "111011"=>          so8x<=To_StdLogicVector(Bit_Vector'(x"5"));
when "111101"=>          so8x<=To_StdLogicVector(Bit_Vector'(x"6"));
when others=>            so8x<=To_StdLogicVector(Bit_Vector'(x"b"));
end case;

--ppermutation

XX(1 to 4)<=so1x;XX(5 to 8)<=so2x;XX(9 to 12)<=so3x;        XX(13 to 16)<=so4x;
XX(17 to 20)<=so5x;      XX(21 to 24)<=so6x;      XX(25 to 28)<=so7x;      XX(29 to 32)<=so8x;

ppo(1)<=XX(16);          ppo(2)<=XX(7);           ppo(3)<=XX(20);          ppo(4)<=XX(21);
ppo(5)<=XX(29);          ppo(6)<=XX(12);          ppo(7)<=XX(28);          ppo(8)<=XX(17);
ppo(9)<=XX(1);           ppo(10)<=XX(15);ppo(11)<=XX(23);ppo(12)<=XX(26);
ppo(13)<=XX(5);          ppo(14)<=XX(18);ppo(15)<=XX(31);ppo(16)<=XX(10);
ppo(17)<=XX(2);          ppo(18)<=XX(8);          ppo(19)<=XX(24);ppo(20)<=XX(14);
ppo(21)<=XX(32);ppo(22)<=XX(27);ppo(23)<=XX(3);          ppo(24)<=XX(9);
ppo(25)<=XX(19);ppo(26)<=XX(13);ppo(27)<=XX(30);ppo(28)<=XX(6);
ppo(29)<=XX(22);ppo(30)<=XX(11);ppo(31)<=XX(4);          ppo(32)<=XX(25);

--xor2
r1x<=ppo xor l0x;
l1x<=r0x;
--permutation finale
ct(1)<=r1x(8);   ct(2)<=l1x(8);   ct(3)<=r1x(16);  ct(4)<=l1x(16);  ct(5)<=r1x(24);  ct(6)<=l1x(24);
        ct(7)<=r1x(32);  ct(8)<=l1x(32);
ct(9)<=r1x(7);   ct(10)<=l1x(7);  ct(11)<=r1x(15); ct(12)<=l1x(15); ct(13)<=r1x(23); ct(14)<=l1x(23);
        ct(15)<=r1x(31); ct(16)<=l1x(31);
ct(17)<=r1x(6);  ct(18)<=l1x(6);  ct(19)<=r1x(14); ct(20)<=l1x(14); ct(21)<=r1x(22); ct(22)<=l1x(22);
        ct(23)<=r1x(30); ct(24)<=l1x(30);
ct(25)<=r1x(5);  ct(26)<=l1x(5);  ct(27)<=r1x(13); ct(28)<=l1x(13); ct(29)<=r1x(21); ct(30)<=l1x(21);
        ct(31)<=r1x(29); ct(32)<=l1x(29);
ct(33)<=r1x(4);  ct(34)<=l1x(4);  ct(35)<=r1x(12); ct(36)<=l1x(12); ct(37)<=r1x(20); ct(38)<=l1x(20);
        ct(39)<=r1x(28); ct(40)<=l1x(28);
ct(41)<=r1x(3);  ct(42)<=l1x(3);  ct(43)<=r1x(11); ct(44)<=l1x(11); ct(45)<=r1x(19); ct(46)<=l1x(19);
        ct(47)<=r1x(27); ct(48)<=l1x(27);
ct(49)<=r1x(2);  ct(50)<=l1x(2);  ct(51)<=r1x(10); ct(52)<=l1x(10); ct(53)<=r1x(18); ct(54)<=l1x(18);
        ct(55)<=r1x(26); ct(56)<=l1x(26);
ct(57)<=r1x(1);  ct(58)<=l1x(1);  ct(59)<=r1x(9);  ct(60)<=l1x(9);  ct(61)<=r1x(17); ct(62)<=l1x(17);
        ct(63)<=r1x(25); ct(64)<=l1x(25);
end if;
wait;
end process ;
end behavior;
```

ANNEXE II: Code VHDL pour l'architecture pipeline

```
--------------------------------------------------------------------------------
--      Auteur:         Mohamed AMOUD, [information retirée / information withdrawn]   --
--                                                                              --
--      Projet:         Design et implémentation sur FPGA d'un algorithme DES   --
--                                                                              --
--      Version:        Architecture pipeline                                   --
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
entity pipeline is port
(
        pt      :       in      std_logic_vector(1 TO 64);
        key     :       in      std_logic_vector(1 TO 64);
        ct      :       out     std_logic_vector(1 TO 64);
        reset   :       in      std_logic;
        clk     :       in      std_logic
);
end pipeline;
architecture behavior of pipeline is
        signal  k1x,k2x,k3x,k4x,k5x,k6x,k7x,k8x,k9x,k10x,k11x,k12x,k13x,k14x,k15x,k16x           :
        std_logic_vector(1 to 48);
        signal  l0xa,l1x,l2x,l3x,l4x,l5x,l6x,l7x,l8x,l9x,l10x,l11x,l12x,l13x,l14x,l15x,l16x   :
        std_logic_vector(1 to 32);
        signal  r0xa,r1x,r2x,r3x,r4x,r5x,r6x,r7x,r8x,r9x,r10x,r11x,r12x,r13x,r14x,r15x,r16x      :
        std_logic_vector(1 to 32);

        component gencle
        port (
                key     :       in      std_logic_vector(1 to 64);
                k1x,k2x,k3x,k4x,k5x,k6x,k7x,k8x,k9x,k10x,k11x,k12x,k13x,k14x,k15x,k16x
                        :       out     std_logic_vector(1 to 48)
        );
        end component;

        component pi
        port (
                pt      :       in std_logic_vector(1 TO 64);
                l0x     :       out std_logic_vector(1 TO 32);
                r0x     :       out std_logic_vector(1 TO 32)
        );
```

```
end component;

component fctronde
port (
        clk             :       in      std_logic;
        reset           :       in      std_logic;
        li,ri   :       in      std_logic_vector(1 to 32);
        k               :       in      std_logic_vector(1 to 48);
        lo,ro   :       out     std_logic_vector(1 to 32)
);
end component;

component pf
port (
        l,r     :       in      std_logic_vector(1 to 32);
        ct      :       out     std_logic_vector(1 to 64)
);
end component;
begin

keyscheduling:  gencle  port map        (       key=>key,       k1x=>k1x,       k2x=>k2x,
k3x=>k3x,       k4x=>k4x,       k5x=>k5x,       k6x=>k6x,       k7x=>k7x,       k8x=>k8x,
k9x=>k9x,       k10x=>k10x,     k11x=>k11x,     k12x=>k12x,     k13x=>k13x,     k14x=>k14x,
k15x=>k15x,     k16x=>k16x      );

iperm:                  pi                      port map        (       pt=>pt,  l0x=>l0xa,
r0x=>r0xa       );

round1: fctronde port map       (       clk=>clk, reset=>reset,     li=>l0xa, ri=>r0xa,k=>k1x,  lo=>l1x,
ro=>r1x  );
round2: fctronde port map       (       clk=>clk, reset=>reset,     li=>l1x,  ri=>r1x,  k=>k2x,  lo=>l2x,
ro=>r2x  );
round3: fctronde port map       (       clk=>clk, reset=>reset,     li=>l2x,  ri=>r2x,  k=>k3x,  lo=>l3x,
ro=>r3x  );
round4: fctronde port map       (       clk=>clk, reset=>reset,     li=>l3x,  ri=>r3x,  k=>k4x,  lo=>l4x,
ro=>r4x  );
round5: fctronde port map       (       clk=>clk, reset=>reset,     li=>l4x,  ri=>r4x,  k=>k5x,  lo=>l5x,
ro=>r5x  );
round6: fctronde port map       (       clk=>clk, reset=>reset,     li=>l5x,  ri=>r5x,  k=>k6x,  lo=>l6x,
ro=>r6x  );
round7: fctronde port map       (       clk=>clk, reset=>reset,     li=>l6x,  ri=>r6x,  k=>k7x,  lo=>l7x,
ro=>r7x  );
round8: fctronde port map       (       clk=>clk, reset=>reset,     li=>l7x,  ri=>r7x,  k=>k8x,  lo=>l8x,
ro=>r8x  );
round9:         fctronde  port map      (       clk=>clk, reset=>reset,     li=>l8x,  ri=>r8x,  k=>k9x,
lo=>l9x,  ro=>r9x  );
round10:        fctronde  port map      (       clk=>clk, reset=>reset,     li=>l9x,  ri=>r9x,
k=>k10x,        lo=>l10x,       ro=>r10x        );
round11:        fctronde port map       (       clk=>clk, reset=>reset,     li=>l10x, ri=>r10x,
k=>k11x,        lo=>l11x,       ro=>r11x        );
round12:        fctronde port map       (       clk=>clk, reset=>reset,     li=>l11x, ri=>r11x,
k=>k12x,        lo=>l12x,       ro=>r12x        );
```

```
        round13:    fctronde port map    (    clk=>clk, reset=>reset,    li=>l12x, ri=>r12x,
        k=>k13x,    lo=>l13x,    ro=>r13x    );
        round14:    fctronde port map    (    clk=>clk, reset=>reset,    li=>l13x, ri=>r13x,
        k=>k14x,    lo=>l14x,    ro=>r14x    );
        round15:    fctronde port map    (    clk=>clk, reset=>reset,    li=>l14x, ri=>r14x,
        k=>k15x,    lo=>l15x,    ro=>r15x    );
        round16:    fctronde port map    (    clk=>clk, reset=>reset,    li=>l15x, ri=>r15x,
        k=>k16x,    lo=>l16x,    ro=>r16x    );
        fperm:  fp    port map    ( i=>r16x,    r=>l16x, ct=>ct    );

end behavior;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity pi is port
(
        pt      :       in std_logic_vector(1 TO 64);
        l0x     :       out std_logic_vector(1 TO 32);
        r0x     :       out std_logic_vector(1 TO 32)
);
end pi;

architecture behavior of pi is
begin
l0x(1)<=pt(58);          l0x(2)<=pt(50);          l0x(3)<=pt(42);          l0x(4)<=pt(34);
l0x(5)<=pt(26);          l0x(6)<=pt(18);          l0x(7)<=pt(10);          l0x(8)<=pt(2);
l0x(9)<=pt(60);          l0x(10)<=pt(52); l0x(11)<=pt(44); l0x(12)<=pt(36);
l0x(13)<=pt(28); l0x(14)<=pt(20); l0x(15)<=pt(12); l0x(16)<=pt(4);
l0x(17)<=pt(62); l0x(18)<=pt(54); l0x(19)<=pt(46); l0x(20)<=pt(38);
l0x(21)<=pt(30); l0x(22)<=pt(22); l0x(23)<=pt(14); l0x(24)<=pt(6);
l0x(25)<=pt(64); l0x(26)<=pt(56); l0x(27)<=pt(48); l0x(28)<=pt(40);
l0x(29)<=pt(32); l0x(30)<=pt(24); l0x(31)<=pt(16); l0x(32)<=pt(8);

r0x(1)<=pt(57);          r0x(2)<=pt(49);          r0x(3)<=pt(41);          r0x(4)<=pt(33);
r0x(5)<=pt(25);          r0x(6)<=pt(17);          r0x(7)<=pt(9);           r0x(8)<=pt(1);
r0x(9)<=pt(59);          r0x(10)<=pt(51); r0x(11)<=pt(43); r0x(12)<=pt(35);
r0x(13)<=pt(27); r0x(14)<=pt(19); r0x(15)<=pt(11); r0x(16)<=pt(3);
r0x(17)<=pt(61); r0x(18)<=pt(53); r0x(19)<=pt(45); r0x(20)<=pt(37);
r0x(21)<=pt(29); r0x(22)<=pt(21); r0x(23)<=pt(13); r0x(24)<=pt(5);
r0x(25)<=pt(63); r0x(26)<=pt(55); r0x(27)<=pt(47); r0x(28)<=pt(39);
r0x(29)<=pt(31); r0x(30)<=pt(23); r0x(31)<=pt(15); r0x(32)<=pt(7);

end behavior;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity gencle is port
(
        key     :       in      std_logic_vector(1 to 64);
        k1x,k2x,k3x,k4x,k5x,k6x,k7x,k8x,k9x,k10x,k11x,k12x,k13x,k14x,k15x,k16x
                :       out     std_logic_vector(1 to 48)
);
end gencle;
```

```
architecture behaviour of gencle is
        signal c0x,c1x,c2x,c3x,c4x,c5x,c6x,c7x,c8x,c9x,c10x,c11x,c12x,c13x,c14x,c15x,c16x        :
        std_logic_vector(1 to 28);
        signal d0x,d1x,d2x,d3x,d4x,d5x,d6x,d7x,d8x,d9x,d10x,d11x,d12x,d13x,d14x,d15x,d16x   :
        std_logic_vector(1 to 28);

        component pc1
        port (
                key             :       in std_logic_vector(1 TO 64);
                c0x,d0x  :      out std_logic_vector(1 TO 28)
        );
        end component;

        component pc2
        port (
                c,d     : in std_logic_vector(1 TO 28);
                k       : out std_logic_vector(1 TO 48)
        );
        end component;

begin
        pc_1: pc1 port map ( key=>key, c0x=>c0x, d0x=>d0x );
        c1x<=To_StdLogicVector(to_bitvector(c0x) rol 1);    d1x<=To_StdLogicVector(to_bitvector(d0x) rol 1);
        c2x<=To_StdLogicVector(to_bitvector(c1x) rol 1);    d2x<=To_StdLogicVector(to_bitvector(d1x) rol 1);
        c3x<=To_StdLogicVector(to_bitvector(c2x) rol 2);    d3x<=To_StdLogicVector(to_bitvector(d2x) rol 2);
        c4x<=To_StdLogicVector(to_bitvector(c3x) rol 2);    d4x<=To_StdLogicVector(to_bitvector(d3x) rol 2);
        c5x<=To_StdLogicVector(to_bitvector(c4x) rol 2);    d5x<=To_StdLogicVector(to_bitvector(d4x) rol 2);
        c6x<=To_StdLogicVector(to_bitvector(c5x) rol 2);    d6x<=To_StdLogicVector(to_bitvector(d5x) rol 2);
        c7x<=To_StdLogicVector(to_bitvector(c6x) rol 2);    d7x<=To_StdLogicVector(to_bitvector(d6x) rol 2);
        c8x<=To_StdLogicVector(to_bitvector(c7x) rol 2);    d8x<=To_StdLogicVector(to_bitvector(d7x) rol 2);
        c9x<=To_StdLogicVector(to_bitvector(c8x) rol 1);    d9x<=To_StdLogicVector(to_bitvector(d8x) rol 1);
        c10x<=To_StdLogicVector(to_bitvector(c9x) rol 2);   d10x<=To_StdLogicVector(to_bitvector(d9x) rol 2);
        c11x<=To_StdLogicVector(to_bitvector(c10x) rol 2);  d11x<=To_StdLogicVector(to_bitvector(d10x) rol 2);
        c12x<=To_StdLogicVector(to_bitvector(c11x) rol 2);  d12x<=To_StdLogicVector(to_bitvector(d11x) rol 2);
        c13x<=To_StdLogicVector(to_bitvector(c12x) rol 2);  d13x<=To_StdLogicVector(to_bitvector(d12x) rol 2);
        c14x<=To_StdLogicVector(to_bitvector(c13x) rol 2);  d14x<=To_StdLogicVector(to_bitvector(d13x) rol 2);
        c15x<=To_StdLogicVector(to_bitvector(c14x) rol 2);  d15x<=To_StdLogicVector(to_bitvector(d14x) rol 2);
        c16x<=To_StdLogicVector(to_bitvector(c15x) rol 1);  d16x<=To_StdLogicVector(to_bitvector(d15x) rol 1);

        pc2x1:  pc2 port map    (       c=>c1x,         d=>d1x,         k=>k1x  );
        pc2x2:  pc2 port map    (       c=>c2x,         d=>d2x,         k=>k2x  );
        pc2x3:  pc2 port map    (       c=>c3x,         d=>d3x,         k=>k3x  );
        pc2x4:  pc2 port map    (       c=>c4x,         d=>d4x,         k=>k4x  );
        pc2x5:  pc2 port map    (       c=>c5x,         d=>d5x,         k=>k5x  );
        pc2x6:  pc2 port map    (       c=>c6x,         d=>d6x,         k=>k6x  );
        pc2x7:  pc2 port map    (       c=>c7x,         d=>d7x,         k=>k7x  );
        pc2x8:  pc2 port map    (       c=>c8x,         d=>d8x,         k=>k8x  );
        pc2x9:  pc2 port map    (       c=>c9x,         d=>d9x,         k=>k9x  );
        pc2x10: pc2 port map    (       c=>c10x,d=>d10x,        k=>k10x );
        pc2x11: pc2 port map    (       c=>c11x,d=>d11x,        k=>k11x );
        pc2x12: pc2 port map    (       c=>c12x,d=>d12x,        k=>k12x );
        pc2x13: pc2 port map    (       c=>c13x,d=>d13x,        k=>k13x );
        pc2x14: pc2 port map    (       c=>c14x,d=>d14x,        k=>k14x );
```

```
        pc2x15: pc2 port map    (       c=>c15x,d=>d15x,        k=>k15x );
        pc2x16: pc2 port map    (       c=>c16x,d=>d16x,        k=>k16x );

end behaviour;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity pc1 is port
(
        key             :       in std_logic_vector(1 TO 64);
        c0x,d0x  :      out std_logic_vector(1 TO 28)
);
end pc1;

architecture behavior of pc1 is
        signal XX:      std_logic_vector(1 to 56);
begin
        XX(1)<=key(57);         XX(2)<=key(49);         XX(3)<=key(41);         XX(4)<=key(33);
        XX(5)<=key(25);         XX(6)<=key(17);         XX(7)<=key(9);
        XX(8)<=key(1);          XX(9)<=key(58);         XX(10)<=key(50); XX(11)<=key(42); XX(12)<=key(34);
        XX(13)<=key(26); XX(14)<=key(18);
        XX(15)<=key(10); XX(16)<=key(2);        XX(17)<=key(59); XX(18)<=key(51); XX(19)<=key(43);
        XX(20)<=key(35); XX(21)<=key(27);
        XX(22)<=key(19); XX(23)<=key(11); XX(24)<=key(3);       XX(25)<=key(60); XX(26)<=key(52);
        XX(27)<=key(44); XX(28)<=key(36);
        XX(29)<=key(63); XX(30)<=key(55); XX(31)<=key(47); XX(32)<=key(39); XX(33)<=key(31); XX(34)<=key(23);
        XX(35)<=key(15);
        XX(36)<=key(7);         XX(37)<=key(62); XX(38)<=key(54); XX(39)<=key(46); XX(40)<=key(38);
        XX(41)<=key(30); XX(42)<=key(22);
        XX(43)<=key(14); XX(44)<=key(6);        XX(45)<=key(61); XX(46)<=key(53); XX(47)<=key(45);
        XX(48)<=key(37); XX(49)<=key(29);
        XX(50)<=key(21); XX(51)<=key(13); XX(52)<=key(5);       XX(53)<=key(28); XX(54)<=key(20);
        XX(55)<=key(12); XX(56)<=key(4);

        c0x<=XX(1 to 28); d0x<=XX(29 to 56);
end behavior;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity pc2 is port
(
        c,d     : in std_logic_vector(1 TO 28);
        k       : out std_logic_vector(1 TO 48)
);
end pc2;

architecture behavior of pc2 is
        signal YY:      std_logic_vector(1 to 56);
begin
        YY(1 to 28)<=c;         YY(29 to 56)<=d;

        k(1)<=YY(14);   k(2)<=YY(17);   k(3)<=YY(11);   k(4)<=YY(24);   k(5)<=YY(1);    k(6)<=YY(5);
        k(7)<=YY(3);    k(8)<=YY(28);   k(9)<=YY(15);   k(10)<=YY(6);   k(11)<=YY(21);  k(12)<=YY(10);
        k(13)<=YY(23);  k(14)<=YY(19);  k(15)<=YY(12);  k(16)<=YY(4);   k(17)<=YY(26);  k(18)<=YY(8);
```

```
        k(19)<=YY(16);   k(20)<=YY(7);    k(21)<=YY(27);   k(22)<=YY(20);   k(23)<=YY(13);   k(24)<=YY(2);
        k(25)<=YY(41);   k(26)<=YY(52);   k(27)<=YY(31);   k(28)<=YY(37);   k(29)<=YY(47);   k(30)<=YY(55);
        k(31)<=YY(30);   k(32)<=YY(40);   k(33)<=YY(51);   k(34)<=YY(45);   k(35)<=YY(33);   k(36)<=YY(48);
        k(37)<=YY(44);   k(38)<=YY(49);   k(39)<=YY(39);   k(40)<=YY(56);   k(41)<=YY(34);   k(42)<=YY(53);
        k(43)<=YY(46);   k(44)<=YY(42);   k(45)<=YY(50);   k(46)<=YY(36);   k(47)<=YY(29);   k(48)<=YY(32);
end behavior;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity fctronde is port
(
        clk             :       in      std_logic;
        reset           :       in      std_logic;
        li,ri   :       in      std_logic_vector(1 to 32);
        k               :       in      std_logic_vector(1 to 48);
        lo,ro   :       out     std_logic_vector(1 to 32)
);
end fctronde;

architecture behaviour of fctronde is
        signal  xp_to_xor       :       std_logic_vector(1 to 48);
        signal  b1x,b2x,b3x,b4x,b5x,b6x,b7x,b8x         :   std_logic_vector(1 to 6);
        signal so1x,so2x,so3x,so4x,so5x,so6x,so7x,so8x      :       std_logic_vector(1 to 4);
        signal ppo,r_toreg32,l_toreg32                      : std_logic_vector(1 to 32);

        component xp
        port (
                ri      : in std_logic_vector(1 TO 32);
                e       : out std_logic_vector(1 TO 48)
        );
        end component;

        component  xor1
        port (
                                                e   :       in std_logic_vector(1 TO 48);
                b1x,b2x,b3x,b4x,b5x,b6x,b7x,b8x :       out std_logic_vector (1 TO 6);
                                                k   :       in std_logic_vector (1 TO 48)
        );
        end component;

        component s1
        port (
                b       :       in      std_logic_vector(1 to 6);
                so      :       out     std_logic_vector(1 to 4)
        );
        end component;

        component s2
        port (
                b       :       in      std_logic_vector(1 to 6);
                so      :       out     std_logic_vector(1 to 4)
        );
        end component;
```

```
component s3
port (
        b       :       in      std_logic_vector(1 to 6);
        so      :       out     std_logic_vector(1 to 4)
);
end component;

component s4
port (
        b       :       in      std_logic_vector(1 to 6);
        so      :       out     std_logic_vector(1 to 4)
);
end component;

component s5
port (
        b       :       in      std_logic_vector(1 to 6);
        so      :       out     std_logic_vector(1 to 4)
);
end component;

component s6
port (
        b       :       in      std_logic_vector(1 to 6);
        so      :       out     std_logic_vector(1 to 4)
);
end component;

component s7
port (
        b       :       in      std_logic_vector(1 to 6);
        so      :       out     std_logic_vector(1 to 4)
);
end component;

component s8
port (
        b       :       in      std_logic_vector(1 to 6);
        so      :       out     std_logic_vector(1 to 4)
);
end component;

component pp
port (
        so1x,so2x,so3x,so4x,so5x,so6x,so7x,so8x  : in      std_logic_vector(1 to 4);
                                                   ppo     :       out     std_logic_vector(1 to 32)
);
end component;

component  xor2
port (
        d,l     :       in      std_logic_vector(1 to 32);
```

```
        q       :       out     std_logic_vector(1 to 32)
    );
    end component;

    component reg32
    port (
        a  : in   std_logic_vector (1 to 32);
        q  : out  std_logic_vector (1 to 32);
        reset :       in      std_logic;
        clk : in   std_logic
    );
    end component;
begin

    xpension:           xp          port map    (   ri=>ri,  e=>xp_to_xor   );
    des_xor1:           xor1        port map    (   e=>xp_to_xor,  k=>k,
    b1x=>b1x,   b2x=>b2x,   b3x=>b3x,   b4x=>b4x,   b5x=>b5x,   b6x=>b6x,
    b7x=>b7x,   b8x=>b8x    );

    s1a:                s1          port map    (   b=>b1x, so=>so1x        );
    s2a:                s2          port map    (   b=>b2x, so=>so2x        );
    s3a:                s3          port map    (   b=>b3x, so=>so3x        );
    s4a:                s4          port map    (   b=>b4x, so=>so4x        );
    s5a:                s5          port map    (   b=>b5x, so=>so5x        );
    s6a:                s6          port map    (   b=>b6x, so=>so6x        );
    s7a:                s7          port map    (   b=>b7x, so=>so7x        );
    s8a:                s8          port map    (   b=>b8x, so=>so8x        );
    pperm:              pp          port map    (   so1x=>so1x,    so2x=>so2x,
    so3x=>so3x,   so4x=>so4x,   so5x=>so5x,   so6x=>so6x,   so7x=>so7x,   so8x=>so8x,
    ppo=>ppo    );
    des_xor2:           xor2        port map    (   d=>ppo, l=>li,  q=>r_toreg32
    );
    l_toreg32<=ri;
    register32_left:  reg32     port map    ( a=>l_toreg32, q=>lo, reset=>reset, clk=>clk );
    register32_right: reg32     port map    ( a=>r_toreg32, q=>ro, reset=>reset, clk=>clk );

end;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity xp is port
(
    ri      : in std_logic_vector(1 TO 32);
    e       : out std_logic_vector(1 TO 48));
end xp;

architecture behavior of xp is
begin
    e(1)<=ri(32);   e(2)<=ri(1);    e(3)<=ri(2);    e(4)<=ri(3);    e(5)<=ri(4);    e(6)<=ri(5);
    e(7)<=ri(4);    e(8)<=ri(5);    e(9)<=ri(6);    e(10)<=ri(7);   e(11)<=ri(8);   e(12)<=ri(9);
    e(13)<=ri(8);   e(14)<=ri(9);   e(15)<=ri(10);  e(16)<=ri(11);
```

```
        e(17)<=ri(12);   e(18)<=ri(13);   e(19)<=ri(12);   e(20)<=ri(13);   e(21)<=ri(14);   e(22)<=ri(15);
        e(23)<=ri(16);   e(24)<=ri(17);
        e(25)<=ri(16);   e(26)<=ri(17);   e(27)<=ri(18);   e(28)<=ri(19);   e(29)<=ri(20);   e(30)<=ri(21);
        e(31)<=ri(20);   e(32)<=ri(21);
        e(33)<=ri(22);   e(34)<=ri(23);   e(35)<=ri(24);   e(36)<=ri(25);   e(37)<=ri(24);   e(38)<=ri(25);
        e(39)<=ri(26);   e(40)<=ri(27);
        e(41)<=ri(28);   e(42)<=ri(29);   e(43)<=ri(28);   e(44)<=ri(29);   e(45)<=ri(30);   e(46)<=ri(31);
        e(47)<=ri(32);   e(48)<=ri(1);
end behavior;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity desxor1 is port
(
                                        e   :       in std_logic_vector(1 TO 48);
        b1x,b2x,b3x,b4x,b5x,b6x,b7x,b8x   :       out std_logic_vector (1 TO 6);
                                        k   :       in std_logic_vector (1 TO 48)
);
end  xor1;

architecture behavior of  xor1 is
        signal XX:          std_logic_vector( 1 to 48);
begin
        XX<=k xor e;
        b1x<=XX(1 to 6);
        b2x<=XX(7 to 12);
        b3x<=XX(13 to 18);
        b4x<=XX(19 to 24);
        b5x<=XX(25 to 30);
        b6x<=XX(31 to 36);
        b7x<=XX(37 to 42);
        b8x<=XX(43 to 48);
end behavior;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity s1 is port
(
        b       :       in      std_logic_vector(1 to 6);
        so      :       out     std_logic_vector(1 to 4)
);
end s1;

architecture behaviour of s1 is
begin
process(b)
begin
case b is
        when "000000"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
        when "000010"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
        when "000100"=>         so<=To_StdLogicVector(Bit_Vector'(x"d"));
        when "000110"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
        when "001000"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
        when "001010"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
```

```
when "001100"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
when "001110"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
when "010000"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
when "010010"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
when "010100"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
when "010110"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
when "011000"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
when "011010"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
when "011100"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
when "011110"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
when "000001"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
when "000011"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
when "000101"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
when "000111"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
when "001001"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
when "001011"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
when "001101"=>         so<=To_StdLogicVector(Bit_Vector'(x"d"));
when "001111"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
when "010001"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
when "010011"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
when "010101"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
when "010111"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
when "011001"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
when "011011"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
when "011101"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
when "011111"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
when "100000"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
when "100010"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
when "100100"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
when "100110"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
when "101000"=>         so<=To_StdLogicVector(Bit_Vector'(x"d"));
when "101010"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
when "101100"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
when "101110"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
when "110000"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
when "110010"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
when "110100"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
when "110110"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
when "111000"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
when "111010"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
when "111100"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
when "111110"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
when "100001"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
when "100011"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
when "100101"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
when "100111"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
when "101001"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
when "101011"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
when "101101"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
when "101111"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
when "110001"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
when "110011"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
when "110101"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
```

```
        when "110111"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
        when "111001"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
        when "111011"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
        when "111101"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
        when others=>           so<=To_StdLogicVector(Bit_Vector'(x"d"));
end case;
end process;
end;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity s2 is port
(
        b       :       in      std_logic_vector(1 to 6);
        so      :       out     std_logic_vector(1 to 4)
);
end s2;

architecture behaviour of s2 is
begin
process(b)
begin
case b is
        when "000000"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
        when "000010"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
        when "000100"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
        when "000110"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
        when "001000"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
        when "001010"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
        when "001100"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
        when "001110"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
        when "010000"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
        when "010010"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
        when "010100"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
        when "010110"=>         so<=To_StdLogicVector(Bit_Vector'(x"d"));
        when "011000"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
        when "011010"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
        when "011100"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
        when "011110"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
        when "000001"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
        when "000011"=>         so<=To_StdLogicVector(Bit_Vector'(x"d"));
        when "000101"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
        when "000111"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
        when "001001"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
        when "001011"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
        when "001101"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
        when "001111"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
        when "010001"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
        when "010011"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
        when "010101"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
        when "010111"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
        when "011001"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
        when "011011"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
```

```
        when "011101"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
        when "011111"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
        when "100000"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
        when "100010"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
        when "100100"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
        when "100110"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
        when "101000"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
        when "101010"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
        when "101100"=>         so<=To_StdLogicVector(Bit_Vector'(x"d"));
        when "101110"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
        when "110000"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
        when "110010"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
        when "110100"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
        when "110110"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
        when "111000"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
        when "111010"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
        when "111100"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
        when "111110"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
        when "100001"=>         so<=To_StdLogicVector(Bit_Vector'(x"d"));
        when "100011"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
        when "100101"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
        when "100111"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
        when "101001"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
        when "101011"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
        when "101101"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
        when "101111"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
        when "110001"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
        when "110011"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
        when "110101"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
        when "110111"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
        when "111001"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
        when "111011"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
        when "111101"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
        when others=>           so<=To_StdLogicVector(Bit_Vector'(x"9"));
end case;
end process;
end;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity s3 is port
(
        b       :       in      std_logic_vector(1 to 6);
        so      :       out     std_logic_vector(1 to 4)
);
end s3;

architecture behaviour of s3 is
begin
process(b)
begin
case b is
        when "000000"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
```

```
when "000010"=>     so<=To_StdLogicVector(Bit_Vector'(x"0"));
when "000100"=>     so<=To_StdLogicVector(Bit_Vector'(x"9"));
when "000110"=>     so<=To_StdLogicVector(Bit_Vector'(x"e"));
when "001000"=>     so<=To_StdLogicVector(Bit_Vector'(x"6"));
when "001010"=>     so<=To_StdLogicVector(Bit_Vector'(x"3"));
when "001100"=>     so<=To_StdLogicVector(Bit_Vector'(x"f"));
when "001110"=>     so<=To_StdLogicVector(Bit_Vector'(x"5"));
when "010000"=>     so<=To_StdLogicVector(Bit_Vector'(x"1"));
when "010010"=>     so<=To_StdLogicVector(Bit_Vector'(x"d"));
when "010100"=>     so<=To_StdLogicVector(Bit_Vector'(x"c"));
when "010110"=>     so<=To_StdLogicVector(Bit_Vector'(x"7"));
when "011000"=>     so<=To_StdLogicVector(Bit_Vector'(x"b"));
when "011010"=>     so<=To_StdLogicVector(Bit_Vector'(x"4"));
when "011100"=>     so<=To_StdLogicVector(Bit_Vector'(x"2"));
when "011110"=>     so<=To_StdLogicVector(Bit_Vector'(x"8"));
when "000001"=>     so<=To_StdLogicVector(Bit_Vector'(x"d"));
when "000011"=>     so<=To_StdLogicVector(Bit_Vector'(x"7"));
when "000101"=>     so<=To_StdLogicVector(Bit_Vector'(x"0"));
when "000111"=>     so<=To_StdLogicVector(Bit_Vector'(x"9"));
when "001001"=>     so<=To_StdLogicVector(Bit_Vector'(x"3"));
when "001011"=>     so<=To_StdLogicVector(Bit_Vector'(x"4"));
when "001101"=>     so<=To_StdLogicVector(Bit_Vector'(x"6"));
when "001111"=>     so<=To_StdLogicVector(Bit_Vector'(x"a"));
when "010001"=>     so<=To_StdLogicVector(Bit_Vector'(x"2"));
when "010011"=>     so<=To_StdLogicVector(Bit_Vector'(x"8"));
when "010101"=>     so<=To_StdLogicVector(Bit_Vector'(x"5"));
when "010111"=>     so<=To_StdLogicVector(Bit_Vector'(x"e"));
when "011001"=>     so<=To_StdLogicVector(Bit_Vector'(x"c"));
when "011011"=>     so<=To_StdLogicVector(Bit_Vector'(x"b"));
when "011101"=>     so<=To_StdLogicVector(Bit_Vector'(x"f"));
when "011111"=>     so<=To_StdLogicVector(Bit_Vector'(x"1"));
when "100000"=>     so<=To_StdLogicVector(Bit_Vector'(x"d"));
when "100010"=>     so<=To_StdLogicVector(Bit_Vector'(x"6"));
when "100100"=>     so<=To_StdLogicVector(Bit_Vector'(x"4"));
when "100110"=>     so<=To_StdLogicVector(Bit_Vector'(x"9"));
when "101000"=>     so<=To_StdLogicVector(Bit_Vector'(x"8"));
when "101010"=>     so<=To_StdLogicVector(Bit_Vector'(x"f"));
when "101100"=>     so<=To_StdLogicVector(Bit_Vector'(x"3"));
when "101110"=>     so<=To_StdLogicVector(Bit_Vector'(x"0"));
when "110000"=>     so<=To_StdLogicVector(Bit_Vector'(x"b"));
when "110010"=>     so<=To_StdLogicVector(Bit_Vector'(x"1"));
when "110100"=>     so<=To_StdLogicVector(Bit_Vector'(x"2"));
when "110110"=>     so<=To_StdLogicVector(Bit_Vector'(x"c"));
when "111000"=>     so<=To_StdLogicVector(Bit_Vector'(x"5"));
when "111010"=>     so<=To_StdLogicVector(Bit_Vector'(x"a"));
when "111100"=>     so<=To_StdLogicVector(Bit_Vector'(x"e"));
when "111110"=>     so<=To_StdLogicVector(Bit_Vector'(x"7"));
when "100001"=>     so<=To_StdLogicVector(Bit_Vector'(x"1"));
when "100011"=>     so<=To_StdLogicVector(Bit_Vector'(x"a"));
when "100101"=>     so<=To_StdLogicVector(Bit_Vector'(x"d"));
when "100111"=>     so<=To_StdLogicVector(Bit_Vector'(x"0"));
when "101001"=>     so<=To_StdLogicVector(Bit_Vector'(x"6"));
when "101011"=>     so<=To_StdLogicVector(Bit_Vector'(x"9"));
```

```
        when "101101"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
        when "101111"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
        when "110001"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
        when "110011"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
        when "110101"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
        when "110111"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
        when "111001"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
        when "111011"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
        when "111101"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
        when others=>           so<=To_StdLogicVector(Bit_Vector'(x"c"));
end case;
end process;
end;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity s4 is port
(
        b       :       in      std_logic_vector(1 to 6);
        so      :       out     std_logic_vector(1 to 4)
);
end s4;

architecture behaviour of s4 is
begin
process(b)
begin
case b is
        when "000000"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
        when "000010"=>         so<=To_StdLogicVector(Bit_Vector'(x"d"));
        when "000100"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
        when "000110"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
        when "001000"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
        when "001010"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
        when "001100"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
        when "001110"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
        when "010000"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
        when "010010"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
        when "010100"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
        when "010110"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
        when "011000"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
        when "011010"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
        when "011100"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
        when "011110"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
        when "000001"=>         so<=To_StdLogicVector(Bit_Vector'(x"d"));
        when "000011"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
        when "000101"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
        when "000111"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
        when "001001"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
        when "001011"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
        when "001101"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
        when "001111"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
        when "010001"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
```

```
        when "010011"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
        when "010101"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
        when "010111"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
        when "011001"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
        when "011011"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
        when "011101"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
        when "011111"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
        when "100000"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
        when "100010"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
        when "100100"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
        when "100110"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
        when "101000"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
        when "101010"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
        when "101100"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
        when "101110"=>         so<=To_StdLogicVector(Bit_Vector'(x"d"));
        when "110000"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
        when "110010"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
        when "110100"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
        when "110110"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
        when "111000"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
        when "111010"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
        when "111100"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
        when "111110"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
        when "100001"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
        when "100011"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
        when "100101"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
        when "100111"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
        when "101001"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
        when "101011"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
        when "101101"=>         so<=To_StdLogicVector(Bit_Vector'(x"d"));
        when "101111"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
        when "110001"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
        when "110011"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
        when "110101"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
        when "110111"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
        when "111001"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
        when "111011"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
        when "111101"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
        when others=>           so<=To_StdLogicVector(Bit_Vector'(x"e"));
end case;
end process;
end;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity s5 is port
(
        b       :       in      std_logic_vector(1 to 6);
        so      :       out     std_logic_vector(1 to 4)
);
end s5;

architecture behaviour of s5 is
```

```
begin
process(b)
begin
case b is
        when "000000"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
        when "000010"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
        when "000100"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
        when "000110"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
        when "001000"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
        when "001010"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
        when "001100"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
        when "001110"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
        when "010000"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
        when "010010"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
        when "010100"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
        when "010110"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
        when "011000"=>         so<=To_StdLogicVector(Bit_Vector'(x"d"));
        when "011010"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
        when "011100"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
        when "011110"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
        when "000001"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
        when "000011"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
        when "000101"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
        when "000111"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
        when "001001"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
        when "001011"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
        when "001101"=>         so<=To_StdLogicVector(Bit_Vector'(x"d"));
        when "001111"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
        when "010001"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
        when "010011"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
        when "010101"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
        when "010111"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
        when "011001"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
        when "011011"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
        when "011101"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
        when "011111"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
        when "100000"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
        when "100010"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
        when "100100"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
        when "100110"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
        when "101000"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
        when "101010"=>         so<=To_StdLogicVector(Bit_Vector'(x"d"));
        when "101100"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
        when "101110"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
        when "110000"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
        when "110010"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
        when "110100"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
        when "110110"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
        when "111000"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
        when "111010"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
        when "111100"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
        when "111110"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
        when "100001"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
```

```
        when "100011"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
        when "100101"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
        when "100111"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
        when "101001"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
        when "101011"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
        when "101101"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
        when "101111"=>         so<=To_StdLogicVector(Bit_Vector'(x"d"));
        when "110001"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
        when "110011"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
        when "110101"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
        when "110111"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
        when "111001"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
        when "111011"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
        when "111101"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
        when others=>           so<=To_StdLogicVector(Bit_Vector'(x"3"));
end case;
end process;
end;
```

```
library ieee;
use ieee.std_logic_1164.all;

entity s6 is port
(
        b       :       in      std_logic_vector(1 to 6);
        so      :       out     std_logic_vector(1 to 4)
);
end s6;

architecture behaviour of s6 is
begin
process(b)
begin
case b is
        when "000000"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
        when "000010"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
        when "000100"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
        when "000110"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
        when "001000"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
        when "001010"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
        when "001100"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
        when "001110"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
        when "010000"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
        when "010010"=>         so<=To_StdLogicVector(Bit_Vector'(x"d"));
        when "010100"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
        when "010110"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
        when "011000"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
        when "011010"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
        when "011100"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
        when "011110"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
        when "000001"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
        when "000011"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
        when "000101"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
```

```
        when "000111"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
        when "001001"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
        when "001011"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
        when "001101"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
        when "001111"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
        when "010001"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
        when "010011"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
        when "010101"=>         so<=To_StdLogicVector(Bit_Vector'(x"d"));
        when "010111"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
        when "011001"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
        when "011011"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
        when "011101"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
        when "011111"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
        when "100000"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
        when "100010"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
        when "100100"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
        when "100110"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
        when "101000"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
        when "101010"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
        when "101100"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
        when "101110"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
        when "110000"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
        when "110010"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
        when "110100"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
        when "110110"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
        when "111000"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
        when "111010"=>         so<=To_StdLogicVector(Bit_Vector'(x"d"));
        when "111100"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
        when "111110"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
        when "100001"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
        when "100011"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
        when "100101"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
        when "100111"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
        when "101001"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
        when "101011"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
        when "101101"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
        when "101111"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
        when "110001"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
        when "110011"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
        when "110101"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
        when "110111"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
        when "111001"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
        when "111011"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
        when "111101"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
        when others=>           so<=To_StdLogicVector(Bit_Vector'(x"d"));
end case;
end process;
end;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity s7 is port
(
```

```
        b       :       in      std_logic_vector(1 to 6);
        so      :       out     std_logic_vector(1 to 4)
);
end s7;

architecture behaviour of s7 is
begin
process(b)
begin
case b is
        when "000000"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
        when "000010"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
        when "000100"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
        when "000110"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
        when "001000"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
        when "001010"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
        when "001100"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
        when "001110"=>         so<=To_StdLogicVector(Bit_Vector'(x"d"));
        when "010000"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
        when "010010"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
        when "010100"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
        when "010110"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
        when "011000"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
        when "011010"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
        when "011100"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
        when "011110"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
        when "000001"=>         so<=To_StdLogicVector(Bit_Vector'(x"d"));
        when "000011"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
        when "000101"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
        when "000111"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
        when "001001"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
        when "001011"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
        when "001101"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
        when "001111"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
        when "010001"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
        when "010011"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
        when "010101"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
        when "010111"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
        when "011001"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
        when "011011"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
        when "011101"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
        when "011111"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
        when "100000"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
        when "100010"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
        when "100100"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
        when "100110"=>         so<=To_StdLogicVector(Bit_Vector'(x"d"));
        when "101000"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
        when "101010"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
        when "101100"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
        when "101110"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
        when "110000"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
        when "110010"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
        when "110100"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
```

```
        when "110110"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
        when "111000"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
        when "111010"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
        when "111100"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
        when "111110"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
        when "100001"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
        when "100011"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
        when "100101"=>         so<=To_StdLogicVector(Bit_Vector'(x"d"));
        when "100111"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
        when "101001"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
        when "101011"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
        when "101101"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
        when "101111"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
        when "110001"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
        when "110011"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
        when "110101"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
        when "110111"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
        when "111001"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
        when "111011"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
        when "111101"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
        when others=>           so<=To_StdLogicVector(Bit_Vector'(x"c"));
end case;
end process;
end;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity s8 is port
(
        b       :       in      std_logic_vector(1 to 6);
        so      :       out     std_logic_vector(1 to 4)
);
end s8;

architecture behaviour of s8 is
begin
process(b)
begin
case b is
        when "000000"=>         so<=To_StdLogicVector(Bit_Vector'(x"d"));
        when "000010"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
        when "000100"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
        when "000110"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
        when "001000"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
        when "001010"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
        when "001100"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
        when "001110"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
        when "010000"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
        when "010010"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
        when "010100"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
        when "010110"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
        when "011000"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
        when "011010"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
```

```
when "011100"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
when "011110"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
when "000001"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
when "000011"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
when "000101"=>         so<=To_StdLogicVector(Bit_Vector'(x"d"));
when "000111"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
when "001001"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
when "001011"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
when "001101"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
when "001111"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
when "010001"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
when "010011"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
when "010101"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
when "010111"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
when "011001"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
when "011011"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
when "011101"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
when "011111"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
when "100000"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
when "100010"=>         so<=To_StdLogicVector(Bit_Vector'(x"b"));
when "100100"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
when "100110"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
when "101000"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
when "101010"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
when "101100"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
when "101110"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
when "110000"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
when "110010"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
when "110100"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
when "110110"=>         so<=To_StdLogicVector(Bit_Vector'(x"d"));
when "111000"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
when "111010"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
when "111100"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
when "111110"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
when "100001"=>         so<=To_StdLogicVector(Bit_Vector'(x"2"));
when "100011"=>         so<=To_StdLogicVector(Bit_Vector'(x"1"));
when "100101"=>         so<=To_StdLogicVector(Bit_Vector'(x"e"));
when "100111"=>         so<=To_StdLogicVector(Bit_Vector'(x"7"));
when "101001"=>         so<=To_StdLogicVector(Bit_Vector'(x"4"));
when "101011"=>         so<=To_StdLogicVector(Bit_Vector'(x"a"));
when "101101"=>         so<=To_StdLogicVector(Bit_Vector'(x"8"));
when "101111"=>         so<=To_StdLogicVector(Bit_Vector'(x"d"));
when "110001"=>         so<=To_StdLogicVector(Bit_Vector'(x"f"));
when "110011"=>         so<=To_StdLogicVector(Bit_Vector'(x"c"));
when "110101"=>         so<=To_StdLogicVector(Bit_Vector'(x"9"));
when "110111"=>         so<=To_StdLogicVector(Bit_Vector'(x"0"));
when "111001"=>         so<=To_StdLogicVector(Bit_Vector'(x"3"));
when "111011"=>         so<=To_StdLogicVector(Bit_Vector'(x"5"));
when "111101"=>         so<=To_StdLogicVector(Bit_Vector'(x"6"));
when others=>           so<=To_StdLogicVector(Bit_Vector'(x"b"));
end case;
end process;
```

```
end;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity pp is port
(
        so1x,so2x,so3x,so4x,so5x,so6x,so7x,so8x   :       in      std_logic_vector(1 to 4);
        ppo                                       :       out     std_logic_vector(1 to 32)
);
end pp;

architecture behaviour of pp is
        signal XX:          std_logic_vector(1 to 32);
begin
        XX(1 to 4)<=so1x; XX(5 to 8)<=so2x; XX(9 to 12)<=so3x;          XX(13 to 16)<=so4x;
        XX(17 to 20)<=so5x;         XX(21 to 24)<=so6x;         XX(25 to 28)<=so7x;         XX(29 to 32)<=so8x;
        ppo(1)<=XX(16);             ppo(2)<=XX(7);              ppo(3)<=XX(20);             ppo(4)<=XX(21);
        ppo(5)<=XX(29);             ppo(6)<=XX(12);             ppo(7)<=XX(28);             ppo(8)<=XX(17);
        ppo(9)<=XX(1);              ppo(10)<=XX(15); ppo(11)<=XX(23); ppo(12)<=XX(26);
        ppo(13)<=XX(5);             ppo(14)<=XX(18); ppo(15)<=XX(31); ppo(16)<=XX(10);
        ppo(17)<=XX(2);             ppo(18)<=XX(8);             ppo(19)<=XX(24); ppo(20)<=XX(14);
        ppo(21)<=XX(32); ppo(22)<=XX(27); ppo(23)<=XX(3);           ppo(24)<=XX(9);
        ppo(25)<=XX(19); ppo(26)<=XX(13); ppo(27)<=XX(30); ppo(28)<=XX(6);
        ppo(29)<=XX(22); ppo(30)<=XX(11); ppo(31)<=XX(4);           ppo(32)<=XX(25);
end;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity  xor2 is port
(
        d,l     :       in      std_logic_vector(1 to 32);
        q       :       out     std_logic_vector(1 to 32)
);
end  xor2;

architecture behaviour of xor2 is
begin
        q<=d xor l;
end;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity pf  is port
(
        l,r     :       in      std_logic_vector(1 to 32);
        ct      :       out     std_logic_vector(1 to 64)
);
end  pf;

architecture behaviour of  pf is
begin
ct(1)<=r(8);    ct(2)<=l(8);    ct(3)<=r(16);   ct(4)<=l(16);   ct(5)<=r(24);   ct(6)<=l(24);
        ct(7)<=r(32);   ct(8)<=l(32);
ct(9)<=r(7);    ct(10)<=l(7);   ct(11)<=r(15);  ct(12)<=l(15);  ct(13)<=r(23);  ct(14)<=l(23);
        ct(15)<=r(31);  ct(16)<=l(31);
```

```
ct(17)<=r(6);   ct(18)<=l(6);   ct(19)<=r(14);  ct(20)<=l(14);  ct(21)<=r(22);  ct(22)<=l(22);
        ct(23)<=r(30);  ct(24)<=l(30);
ct(25)<=r(5);   ct(26)<=l(5);   ct(27)<=r(13);  ct(28)<=l(13);  ct(29)<=r(21);  ct(30)<=l(21);
        ct(31)<=r(29);  ct(32)<=l(29);
ct(33)<=r(4);   ct(34)<=l(4);   ct(35)<=r(12);  ct(36)<=l(12);  ct(37)<=r(20);  ct(38)<=l(20);
        ct(39)<=r(28);  ct(40)<=l(28);
ct(41)<=r(3);   ct(42)<=l(3);   ct(43)<=r(11);  ct(44)<=l(11);  ct(45)<=r(19);  ct(46)<=l(19);
        ct(47)<=r(27);  ct(48)<=l(27);
ct(49)<=r(2);   ct(50)<=l(2);   ct(51)<=r(10);  ct(52)<=l(10);  ct(53)<=r(18);  ct(54)<=l(18);
        ct(55)<=r(26);  ct(56)<=l(26);
ct(57)<=r(1);   ct(58)<=l(1);   ct(59)<=r(9);   ct(60)<=l(9);   ct(61)<=r(17);  ct(62)<=l(17);
        ct(63)<=r(25);  ct(64)<=l(25);

end;
```

```
library ieee ;
use ieee.std_logic_1164.all;
entity reg32 is
  port(
    a          : in    std_logic_vector (1 to 32);
    q          : out   std_logic_vector (1 to 32);
    reset : in      std_logic;
    clk    : in    std_logic
  );
end reg32;

architecture synth of reg32 is

signal memory : std_logic_vector (1 to 32) ;

begin
process(clk,reset)
begin
                if(clk = '1' and clk'event) then

                        -- on affecte la mémoire interne au coup d'horloge
                        memory <= a;

                end if;

                if(reset = '1') then
                        memory <= (others => '0');
                end if;
end process;
q <= memory;
end synth;
```

Fichier testbench

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity test_pipeline is end test_desenc;
architecture testbench of test_pipeline is
component pipeline port
(
        pt      :       in      std_logic_vector(1 TO 64);
        key     :       in      std_logic_vector(1 TO 64);
        ct      :       out     std_logic_vector(1 TO 64);
        reset   :       in      std_logic;
        clk     :       in      std_logic
);
end component;
type test_vector is record
        key     :       std_logic_vector(1 to 64);
        pt      :       std_logic_vector(1 to 64);
        ct      :       std_logic_vector(1 to 64);
end record;

type test_vector_array is array(natural range <>) of test_vector;
constant test_vectors: test_vector_array :=(
        (       key=>x"0000000000000000",       pt=>x"0000000000000000",
        ct=>x"8ca64de9c1b123a7"         ),
        (       key=>x"ffffffffffffffff",       pt=>x"ffffffffffffffff",       ct=>x"7359b2163e4edc58"         ),
        (       key=>x"3000000000000000",       pt=>x"1000000000000001",
        ct=>x"958e6e627a05557b"         ),
        (       key=>x"1111111111111111",       pt=>x"1111111111111111",       ct=>x"f40379ab9e0ec533"
        ),
        (       key=>x"0123456789abcdef",       pt=>x"1111111111111111",
        ct=>x"17668dfc7292532d"         ),
        (       key=>x"1111111111111111",       pt=>x"0123456789abcdef",       ct=>x"8a5ae1f81ab8f2dd"
        ),
        (       key=>x"0000000000000000",       pt=>x"0000000000000000",
        ct=>x"8ca64de9c1b123a7"         ),
        (       key=>x"fedcba9876543210",       pt=>x"0123456789abcdef",       ct=>x"ed39d950fa74bcc4"
        ),
        (       key=>x"7ca110454a1a6e57",       pt=>x"01a1d6d039776742",
        ct=>x"690f5b0d9a26939b"         ),
        (       key=>x"0131d9619dc1376e",       pt=>x"5cd54ca83def57da",
        ct=>x"7a389d10354bd271"         ),
        (       key=>x"07a1133e4a0b2686",       pt=>x"0248d43806f67172",
        ct=>x"868ebb51cab4599a"         ),
        (       key=>x"3849674c2602319e",       pt=>x"51454b582ddf440a",
        ct=>x"7178876e01f19b2a"         ),
```

```
        (       key=>x"04b915ba43feb5b6",       pt=>x"42fd443059577fa2",        ct=>x"af37fb421f8c4095"
        ),
        (       key=>x"0113b970fd34f2ce",       pt=>x"059b5e0851cf143a",
        ct=>x"86a560f10ec6d85b"         ),
        (       key=>x"0170f175468fb5e6",       pt=>x"0756d8e0774761d2",
        ct=>x"0cd3da020021dc09"         ),
        (       key=>x"43297fad38e373fe",       pt=>x"762514b829bf486a",
        ct=>x"ea676b2cb7db2b7a"         ),
        (       key=>x"07a7137045da2a16",       pt=>x"3bdd119049372802",        ct=>x"dfd64a815caf1a0f"
        ),
        (       key=>x"04689104c2fd3b2f",       pt=>x"26955f6835af609a",        ct=>x"5c513c9c4886c088"
        ),
        (       key=>x"37d06bb516cb7546",       pt=>x"164d5e404f275232",        ct=>x"0a2aeeae3ff4ab77"
        ),
        (       key=>x"1f08260d1ac2465e",       pt=>x"6b056e18759f5cca",        ct=>x"ef1bf03e5dfa575a"
        ),
        (       key=>x"584023641aba6176",       pt=>x"004bd6ef09176062",
        ct=>x"88bf0db6d70dee56"         ),
        (       key=>x"025816164629b007",       pt=>x"480d39006ee762f2",
        ct=>x"a1f9915541020b56"         ),
        (       key=>x"49793ebc79b3258f",       pt=>x"437540c8698f3cfa",        ct=>x"6fbf1cafcffd0556"
        ),
        (       key=>x"4fb05e1515ab73a7",       pt=>x"072d43a077075292",        ct=>x"2f22e49bab7ca1ac"
        ),
        (       key=>x"49e95d6d4ca229bf",       pt=>x"02fe55778117f12a",        ct=>x"5a6b612cc26cce4a"
        ),
        (       key=>x"018310dc409b26d6",       pt=>x"1d9d5c5018f728c2",
        ct=>x"5f4c038ed12b2e41"         ),
        (       key=>x"1c587f1c13924fef",       pt=>x"305532286d6f295a",        ct=>x"63fac0d034d9f793"
        ),
        (       key=>x"0101010101010101",       pt=>x"0123456789abcdef",        ct=>x"617b3a0ce8f07100"
        ),
        (       key=>x"1f1f1f1f0e0e0e0e",       pt=>x"0123456789abcdef",        ct=>x"db958605f8c8c606"
        ),
        (       key=>x"e0fee0fef1fef1fe",pt=>x"0123456789abcdef",       ct=>x"edbfd1c66c29ccc7" ),
        (       key=>x"0000000000000000",       pt=>x"ffffffffffffffff",        ct=>x"355550b2150e2451"
        ),
        (       key=>x"ffffffffffffffff",       pt=>x"0000000000000000",        ct=>x"caaaaf4deaf1dbae" ),
        (       key=>x"0123456789abcdef",       pt=>x"0000000000000000",        ct=>x"d5d44ff720683d0d"
        ),
        (       key=>x"fedcba9876543210",       pt=>x"ffffffffffffffff",        ct=>x"2a2bb008df97c2f2" )
);

        signal  key : std_logic_vector(1 to 64);
        signal  pt : std_logic_vector(1 to 64);
        signal  ct : std_logic_vector(1 to 64);
        signal  clk : std_logic;
        signal  reset : std_logic;

begin

dut:    pipeline  port map      (  key=>key,  pt=>pt,  ct=>ct, reset=>reset,  clk=>clk    );
```

```
	process
		variable vector   :	test_vector;
		variable errors   :	boolean:=false;
	begin
		for i in test_vectors'range loop
			vector:=test_vectors(i);
			key<=vector.key; pt<=vector.pt;

			for j in 0 to 15 loop	clk<='0'; wait for 250 ns;  clk<='1'; wait for 250 ns;  end
loop;

		end loop;
		wait;
	end process;
end testbench;
```