الجمهورية الجزائرية الديمقراطية الشعبية

*Ministère de L'Enseignement Supérieur Et De La Recherche Scientifique*

**Université Yahia Farés de MEDEA**

**Faculté de Technologie**

**Département de génie électrique**

# *Mémoire de fin d'études*

***Présenté pour l'obtention du Diplôme de Master en Télécommunication***

***Option : systèmes intelligents et communicants***

***Thème:***

*Etude et implementation sur un circuit FPGA de l'algorithme de hachage cryptographique SHA-1*

***Réalisé par :***

***Benterki Zineb***
***Rezki Sarrah***

***Encadré par :***

*Mohamed ould zmirli*

**ANNEE UNIVERSITAIRE: 2016 /2017**

# Remerciement

*En premier,*
*Nous tenons à remercier vivement le bon dieu, le clément et Le miséricordieux qui nous a donné le courage, la force et la patience pour achever notre travail.*

*En présentant ce travail, il nous est un agréable devoir celui de remercie ceux, grâce à qui la réalisation de ce mémoire a pu prendre jour.*

*Nous remercions nos parents pour leurs aides et leurs soutient pendant nos études.*

*Nos remerciements les plus sincères et notre reconnaissance vont particulièrement à notre promoteur* ***OULD ZMIRLI MOHAMED*** *qui nous a guidés,*
*Soutenu et éclairé dès le début de notre mémoire.*

*Par la même occasion nous tenons à remercier Également tous ceux qui ont accepté de juger notre Travail.*

*Que toute personne ayant de près ou de loin à l'élaboration de cette étude trouve ici L'expression de notre profonde sympathie.*

*Sarrah*

*Zineb*

# Acronyms

**ARM:** **A**dvanced **R**isc **M**achine

**ASIC:** **A**pplication-**S**pecific **I**ntegrated **C**ircuit

**CLB:** **C**onfigurable **L**ogic **B**loc

**CPU:** **C**entral **P**rocessing **U**nit

**CPLD:** **C**omplex **P**rogrammable **L**ogic **D**evice

**FPGA:** **F**ield **P**rogrammable **G**ate **A**rrays

**EDK*:*** **E**mbedded **D**esign **k**it

**FPU:** **F**loating **P**roduction **U**nit

**HDL:** **H**ardware **D**escription **L**anguage

**HW/SW:** **H**ard**W**are/**S**oft**W**are

**IBM:** **I**nternational **B**usiness **M**achines **C**orporation

**IP*:*** **I**ntellectual **p**roperty

**ISE:** **I**ntegrated **S**ystem **E**nvironnement

**JTAG:** **J**oint **T**est **A**ction **G**roup

**LUT:** **L**ook-**U**p **T**able

**MD4/MD5:** ***Message Digest 4/ Message Digest 5***

**NSA:** **N**ational **S**ecurity **A**gency

**NIST:** **N**ational **I**nstitute of **S**cience and **T**echnology

**O($2^{63}$) :L**a **C**omplixité

**PLD:** **P**rogrammable **L**ogic **D**evice

**SHA:** ***Secure Hash Algorithm***

**SOC:** **S**ystem **O**n **C**hip

**SOPC:** **S**ystem **O**n **P**rogrammable **C**hip

**SSL:** ***Secure Sockets Layer***

**RTL*:*** **R**egister **T**ransfer **L**evel

**UART:** **U**niversal **A**synchronous **R**eceiver **T**ransmitter

**USB:** **U**niversal **S**erial **B**us

**VHDL:** **V**ery High **H**ardware **D**escription **L**anguage

**VGA:** **V**ideo **G**raphics **A**rray

**XPS: Xilinx Platform Studio**

# Bibliographie

**[1] :** Thomas Peyrin. (2008). *Analyse de fonctions de hachage,* Doctorat de l'Université de Versailles Saint-Quentin-en-Yvelines
(Spécialité informatique)

**[2] :**Zakaria Kaddouri. (2014). *Mise en œuvre de nouvelles techniques pour la securité informatique baseé sue les algorithmes evolustionnisteet les fonctions de hachage.* Rabat, Maroc.

**[3]:**Jean-Guillaume Dumas, J.-L. R. (2007). *Théorie Des Codes Compression, cryptage, correction.* Paris.

**[4]:**D. Eastlake, P. J. (2001). *Algorithme 1 de hachage sécurisé (SHA-1) RFC3174.*

**[5]:** Hamzi Imane, Contribution à la sécurité des communications d'un réseau WSN appliqué dans un système de contrôle du trafic routier, Mémoire de Master en Génie électrique, option de systèmes intelligents et communicants, Département du Génie Électrique et de l'informatique, université Dr Yahia Farès de Médéa,2014-2015.

**[6]:** (Digilentt Nexys2 Board Reference Manual, 2011)

**[7]:**FIPS PUB 180-1(1993 May 11). *SECURE HASH STANDARD.*

## I.1. Introduction :

Dans ce chapitre, nous introduisons le principe de hachage avec ses différentes propriétés (la résistance aux attaques recherchant des collisions, pré-images et secondes pré-images). Ensuite, nous présentons les fonctions de la famille SHA, à savoir, SHA0, SHA1, SHA2 et SHA3. Et nous terminons par les domaines d'applications.

## I.2. Définition de la fonction de hachage : [1]

On nomme fonction de hachage, de l'anglais hash function, une fonction particulière qui, à partir d'une donnée fournie en entrée de taille variable, calcule en sortie un élément de taille fixe, servant à identifier rapidement, bien qu'incomplètement, la donnée initiale. Les fonctions de hachage sont utilisées en informatique et en cryptographie. Une fonction de hachage H de taille de sortie n est un algorithme faisant correspondre à un message M de taille arbitraire un élément H(M) de taille n bits appelé haché. En pratique, n est de l'ordre de plusieurs centaines de bits, typiquement 128, 160, 256 ou 512 bits. Ces fonctions sont très utiles notamment dans le domaine des bases de données. Elles permettent de manipuler de très grandes structures tout en gardant une vitesse acceptable en ce qui concerne la recherche d'éléments. Elles permettent notamment de diminuer la quantité d'information à crypter.

Si l'image de x par la fonction de hachage est appelée l'empreinte de x, on peut par exemple ne crypter que l'empreinte. D'autre part, elles permettent de mettre au point des protocoles de signature électronique et d'authentification des messages ainsi que de vérifier l'intégrité d'un document, au même titre que le bit de parité, qui lui-même est une fonction de hachage particulière. Formellement, une fonction de hachage H : $\{0,1\}^* \rightarrow \{0,1\}^n$ est une application qui transforme une chaîne de taille quelconque en une chaîne de taille fixe n, comme illustre la figure **I.1**.

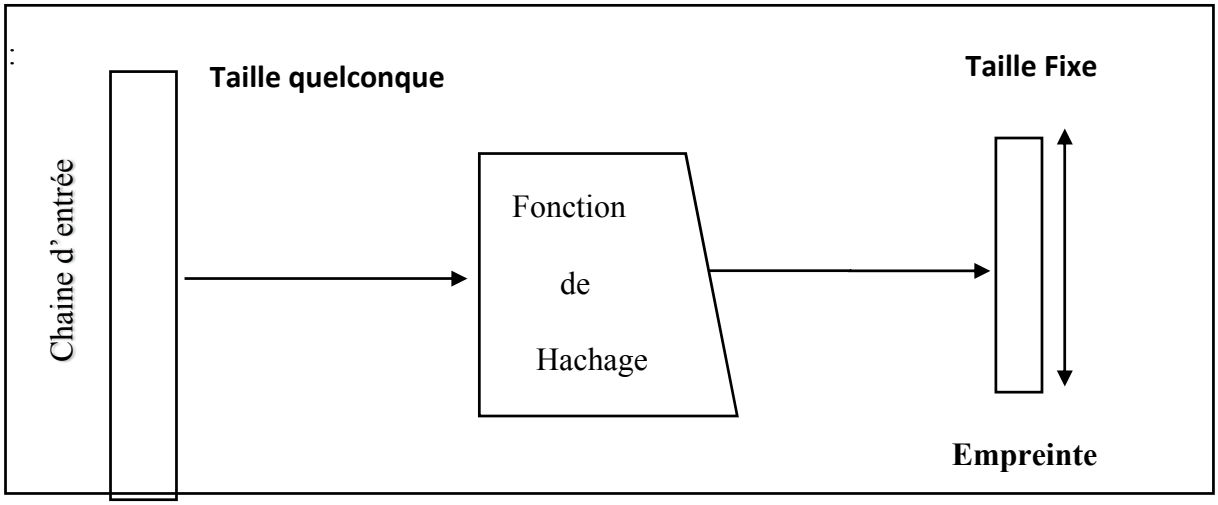



***Figure* I.1** : Principe d'une fonction de hachage

## I.3. Propriétés des fonctions de hachage :

Les fonctions de hachage doivent posséder plusieurs propriétés utiles en cryptographie. Les principales sont la résistance aux attaques recherchant des collisions, des pré-images ou des secondes pré-images.

– **collision** : trouver deux messages distincts $M^1$et $M^2$, tels que $H(M^1) = H(M^2)$.

– **seconde pré-image** : étant donné un message $M^1$ choisi aléatoirement, trouver un message distinct $M^2$ tel que $H(M^1) = H(M^2)$.

– **pré-image** : étant donné un haché H1 choisi aléatoirement, trouver un message M tel que $H(M^1) = H^1$.

Il doit être impossible pour un attaquant de trouver une collision, une pré-image ou une seconde pré-image. On remarque que puisque la taille d'entrée de la fonction de hachage est arbitrairement grande, des collisions existent nécessairement.

Ces notions de sécurité peuvent aussi être appliquées aux fonctions de compression, la seule différence étant la taille fixe des messages et la possibilité pour l'attaquant de choisir ou non la variable de chaînage d'entrée. Le nombre d'opérations pour une attaque générique reste identique aux cas respectifs des fonctions de hachage.

– **collision** : étant donné un état interne $H_{i-1}$ choisi aléatoirement, trouver deux blocs de message distincts $M_i^1$ et $M_i^2$tels que $h(H_{i-1}, M_i^1) = h(H_{i-1}, M_i^2)$.

– **collision libre**: trouver deux blocs de message distincts $M_i^1$ et $M_i^2$ et un état interne $H_{i-1}$ tels que $h(H_{i-1}, M_i^1) = h(H_{i-1}, M_i)$.

– **seconde pré-image** : étant donné un état interne $H_{i-1}$ et un bloc de message $M_i^1$ choisis aléatoirement, trouver un bloc de message distinct $M_i^2$ tel que $h(H_{i-1}, M_i^1) = h(H_{i-1}, M_i^2)$.

– **seconde pré-image libre**: étant donné un bloc de message $M_i^1$ choisi aléatoirement, trouver un bloc de message distinct $M_i^2$ et un état interne $H_{i-1}$ tels que

$H(H_{i-1}, M_i^1) = h(H_{i-1}, M_i^2)$.

– **pré-image** : étant donné deux états internes $H_{i-1}$ et $H_i$ choisis aléatoirement, trouver un bloc de message $M_i^1$ tel que $H_i = h(H_{i-1}, M_i^1)$.

– **Pré-image libre** : étant donné un bloc de message $M_i^1$ choisi aléatoirement, trouver un bloc de message distinct $M_i^2$ et un état interne $H_{i-1}$ tels que $h(H_{i-1}, M_i^1) = h(H_{i-1}, M_i^2)$.

## I.4. Présentation des fonctions de la famille SHA :

### *I.4.1. Généralités sur SHA (Secure Hash Algorithm) :*

SHA est un algorithme mathématique. Il existe plusieurs déclinaisons de SHA: SHA-0, qui est obsolète et ne s'emploie plus, SHA-1, la version encore utilisée pour la majorité des certificats SSL, SHA-2, une version améliorée de l'algorithme, et SHA-3, paru en 2012.

L'algorithme SHA-2 comporte quatre versions, SHA-224, SHA-256, SHA-384 et SHA-512. L'algorithme SHA-256 est celui que les émetteurs de certificats et les navigateurs favorisent, et les termes SHA-2 et SHA-256 se confondent dans l'usage courant.

### *I.4.2 SHA-0 : [2]*

C'est une fonction de hachage cryptographique, conçue par la NSA sur le modèle de la fonction MD4 de Ron Rivest et publiée en 1993 par le NIST (standard FIPS-180). Elle fournit une empreinte de 160 bits. Elle a été rapidement mise de côté pour des raisons de sécurité insuffisante (non explicitées à l'époque), et modifiée pour donner la fonction SHA-1, publiée par le NIST en 1995 (standard FIPS-180-1).

#### *I.4.2.1 Sécurité de la fonction SHA-0 : [2]*

Les faiblesses dans la conception de SHA-0 ont été confirmées en 1998 par la découverte par Antoine Joux et Florent Chabaud d'une attaque par collision contre SHA-0, avec une complexité égale à $2^{61}$ appels à la fonction de compression contre $2^{80}$ (l'attaque générique des anniversaires).

Certes, il n'existe pas actuellement une attaque connue pour la recherche de pré-image ou de seconde pré-image, mais l'utilisation de SHA-0 est tout à fait déconseillée. Elle est pourtant restée un domaine de recherche pour la communauté académique, en tant que prédécesseur de SHA-1.

### *I.4.3 SHA-1 :*

#### *I.4.3.1 Bourrage du message : [4]*

SHA-1 est utilisé pour calculer un résumé de message pour un message ou fichier de données qui est fourni en entrée. Le message ou fichier de données devrait être considéré comme une chaîne binaire. La longueur du message est le nombre de bits dans le message.

Si le nombre de bits dans un message est un multiple de 8, pour le rendre compact, on peut représenter le message en hexadécimal. L'objet du bourrage de message est de rendre la longueur totale d'un message bourré multiple de 512. SHA-1 traite en séquence les blocs de 512 bits lors du calcul du résumé du Message.

Ce qui suit spécifie comment devra être effectué ce bourrage. En résumé, un "1" suivi par des "0" suivis par un entier de 64 bits sont ajoutés à la fin du message pour produire un message bourré de longueur 512 * n. L'entier de 64 bits est la longueur du message d'origine. Le message bourré est alors traité par SHA-1 comme des blocs de 512 bits.

Supposons qu'un message a une longueur $l < 2^{64}$. Avant qu'il soit entré dans SHA-1, le message est bourré à droite comme suit :

**a.** un "1" est ajouté.

**b.** Des "0" sont ajoutés. Le nombre de "0" va dépendre de la longueur d'origine du message. Les derniers 64 bits du dernier bloc de 512 bits sont réservés pour la longueur l du message d'origine.

**c.** Obtenir la représentation de l en deux mots, le nombre de bits dans le message d'origine. Si $l < 2^{32}$ le premier mot est alors tout de zéros. Ajouter ces deux mots au message bourré.

Le message bourré va contenir 16*n mots pour toute n > 0. Le message bourré est considéré comme une séquence des n blocs M(1), M(2),……M(n) des premiers caractères (ou bits) du message.

### *I.4.3.2. Fonctions et constantes utilisées :*

Une séquence des fonctions logiques f(0), f(1),..., f(79) est utilisée dans SHA-1. Chaque f(t), $0 \leq t \leq 79$, opère sur trois mots B, C, D de 32 bits et produit un mot de 32 bits en résultat. f(t;B,C,D) est défini comme suit :

pour les mots B, C, D,

f(t ; B, C, D) = (B ET C) OU ((NON B) ET D) $(0 \leq t \leq 19)$

f(t ; B, C, D) = B OUX C OUX D $(20 \leq t \leq 39)$

f(t ; B ,C ,D) = (B ET C) OU (B ET D) OU (C ET D) $(40 \leq t \leq 59)$

f(t ; B , C, D) = B OUX C OUX D $(60 \leq t \leq 79)$.

Une séquence de mots constants K(0), K(1), ... , K(79) est utilisée dans SHA-1. En hexadécimal, ils donnent :

K(t) = 5A827999 $(0 \leq t \leq 19)$

K(t) = 6ED9EBA1 $(20 \leq t \leq 39)$

K(t) = 8F1BBCDC $(40 \leq t \leq 59)$

K(t) = CA62C1D6 $(60 \leq t \leq 79)$.

### *I.4.3.3 Calcul du résumé de message :*

Le résumé de message est calculé en utilisant le message bourré comme décrit précédemment. Le calcul est décrit avec l'utilisation de deux mémoires tampon, chacune consistant en cinq mots de 32 bits, et une séquence de quatre-vingt mots de 32 bits. Les mots de la première mémoire tampon de cinq mots sont étiquetés A, B, C, D, E. Les mots de la seconde sont étiquetés H0, H1, H2, H3, H4. Les mots de la séquence de 80 mots sont étiquettes W(0), W(1),..., W(79). Une mémoire tampon TEMP d'un seul mot est aussi employée.

Pour générer le résumé de message, les blocs M(1), M(2),..., M(n) de 16 bits sont traités dans cet ordre. Le traitement de chaque M(i) implique 80 étapes.

Avant de traiter un bloc, les H sont initialisés comme suit, en hexadécimal :

H0 = 67452301

H1 = EFCDAB89

H2 = 98BADCFE

H3 = 10325476

H4 = C3D2E1F0

Ensuite M(1), M(2), ..., M(n) sont traités. Pour traiter M(i), on procède comme suit :

**a.** Diviser M(i) en 16 mots W(0), W(1), ..., W(15), ou W(0) est le mot le plus à gauche.

**b.** Pour $16 \leq t \leq 79$ soit W(t) = ( W(t-3) OUX W(t-8) OUX W(t-14) OUX W(t-16) ) <<<1.

**c.** Soit A = H0, B = H1, C = H2, D = H3, E = H4.

**d.** Pour $0 \leq t \leq 79$ faire TEMP = (A) <<<5 + f(t;B, C, D) + E + W(t) + K(t) ; E = D ; D = C; C = (B)<<<30 ; B = A ; A = TEMP;

**e.** Soit H0 = H0 + A, H1 = H1 + B, H2 = H2 + C, H3 = H3 + D, H4 = H4 + E.

Après le traitement de M(n), le résumé de message est la chaîne de 160 bits représentée par les 5 mots H0 H1 H2 H3 H4. Le détail d'un tour SHA-1 est fourni dans la figure **I.2**.

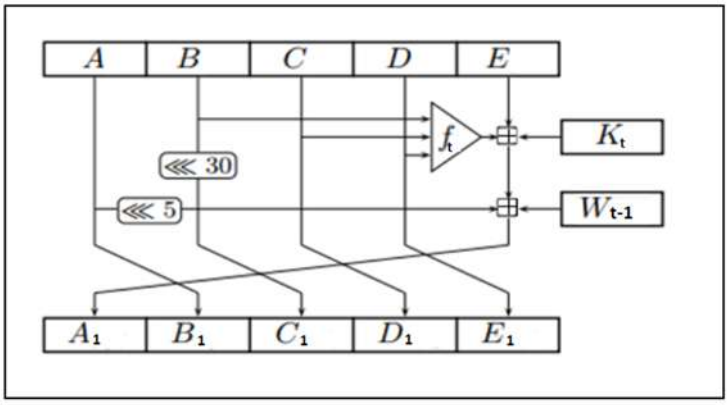



**Figure I.2 :** Fonction de mise à jour des registres

### *I.4.4 Sécurité de SHA-1 : [2]*

La taille d'empreinte dans SHA-1 est de 160 bits. La recherche de collisions par force brute (attaque de Yuval) a donc une complexité en 280 calculs d'empreintes. Récemment, une technique similaire à celle utilisée contre MD5 a permis d'obtenir des collisions en seulement $O(2^{63})$ calculs d'empreinte. On considère donc que SHA-1 est cassé, même si l'attaque reste difficile. De part sa popularité, cette fonction de hachage est encore largement utilisée.

### *I.4.5 SHA-2 :[2]*

En 2002 le NIST a normalisé une nouvelle famille de fonctions de hachage, il s'agit de SHA-2 [NIS12]. Quatre versions de cet algorithme permettent de calculer des hachés de taille différente SHA-224, SHA-256, SHA-384 et SHA-512 bits.

SHA-2 utilise l'algorithme d'extension de domaine de Merkle-Damgård et la construction de Davies-Meyer comme ses prédécesseurs. Généralement, SHA-224 et SHA-256 travaillent sur des blocs de message 512 bits et utilisent des variables de chaînage de 256 bits (8 registres de 32 bits), et SHA-384 et SHA-512 travaillent sur des blocs de message 1024 bits et utilisent des variables de chaînage de 512 bits (8 registres de 64 bits). La structure générale des opérations effectuées est analogue entre ces quatre fonctions.

### *I.4.6 SHA-3 : [2]*

En 2007, le NIST a lancé un concours visant à définir un nouveau standard pour les fonctions de hachage, SHA-3. Le besoin d'un nouveau standard fait suite aux nombreuses attaques essuyées par la fonction SHA1, pour laquelle une collision peut être trouvée en $O(2^{63})$. Le standard actuel (SHA-2) étant dérivé de SHA-1 mais avec des tailles de hachés plus grandes, pourrait donc être victime du même type d'attaque.

## I.5. Domaines d'utilisation des fonctions de hachage cryptographiques : [2]

Les applications des fonctions de hachage cryptographiques apparaissent dans de nombreux domaines d'applications. Chacun de ces domaines nécessite des propriétés spécifiques de sécurité, nous citons dans cette partie des exemples de ces domaines.

### *I.5.1 Intégrité des données :*

La vérification de l'intégrité d'une donnée est parmi les principales utilisations d'une fonction de hachage. C'est le fait de s'assurer que les données qui ont été envoyées dans un canal non sûr sont complètes et n'ont pas été changées. La moindre modification dans les données doit aboutir à l'obtention d'empreintes totalement différentes avec une probabilité très grande.

### *I.5.2 Protection de mots de passe :*

Une autre des applications courantes des fonctions de hachage cryptographiques est la protection de mots de passe. Une grande partie des utilisateurs choisissent comme mots de passe des mots courants ou triviaux, n'assurant donc pas une entropie suffisante. Pour cela un mot de passe ne doit pas être stocké en clair sur une machine, seul le résultat du hachage (le haché) du mot de passe est donc stocké. L'ordinateur compare l'empreinte du mot de passe d'origine avec l'empreinte du mot de passe demandé pour identifier un utilisateur.

### *I.5.3 Signature électronique :*

Les schémas de signature électronique sont sans aucun doute l'application la plus importante des fonctions de hachage cryptographiques.

La signature électronique est un mécanisme analogue de la signature manuscrite permettant de garantir l'intégrité d'un document électronique et prouvant au lecteur du document l'identité de son auteur (authentification).

Lorsqu'un utilisateur veut envoyer un message signé elle suit la procédure suivante :

Il calcule le haché du message, le chiffre avec sa clé privée, et le transmet avec le message. Le destinataire décrypte le condensat avec sa clé publique, calcule le haché du message reçu, et les compare. Si les deux hachés sont identiques, il est assuré de l'identité de l'émetteur et l'intégrité du message.

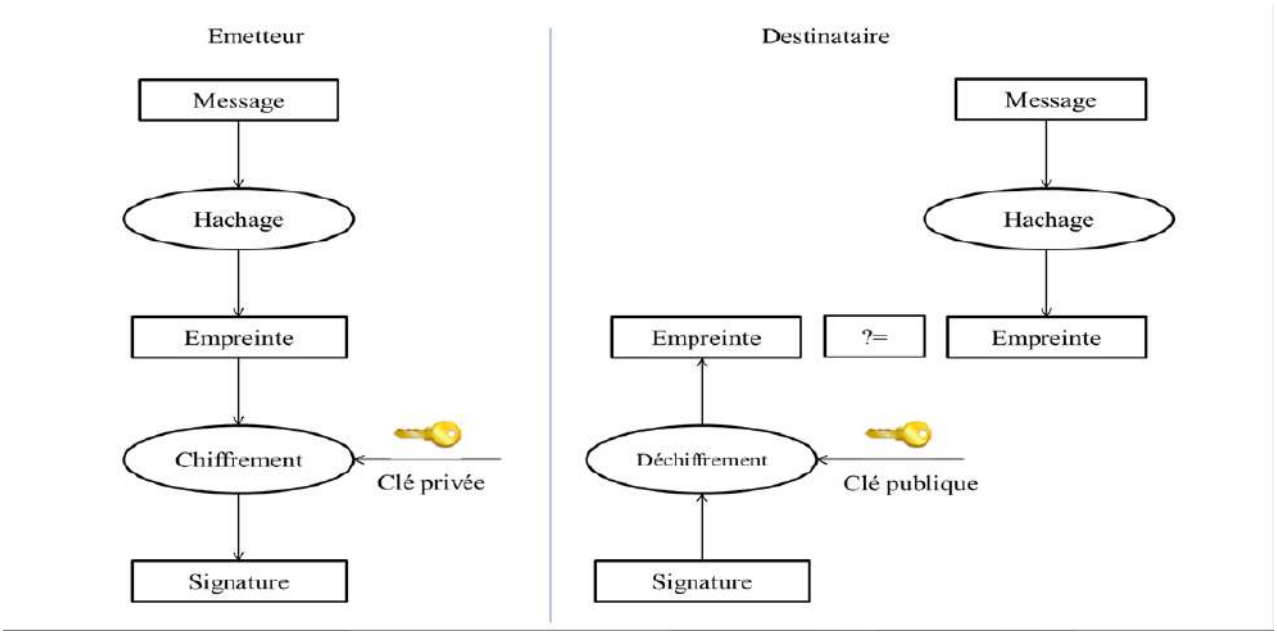



**Figure I.3:** Principe de signatures électroniques.

### *I.5.4 Dérivation de clé :*

Dans le cadre de la cryptographie symétrique, pour assurer une communication entre deux entités, plusieurs clés secrètes sont souvent utilisées. Une fonction de hachage peut générer plusieurs clefs différentes à partir d'une seule information secrète, appelée clef maître. L'utilisation des fonctions de hachage pour cette fonctionnalité a pour objectif d'empêcher un adversaire ayant obtenu une des clés dérivées, de déduire des informations sur la clé maître ou les autres clés dérivées.

### *I.5.5 Génération de nombres pseudo-aléatoires :*

Un générateur pseudo-aléatoire de nombres est un algorithme qui produit des suites de bits présentant certaines propriétés de l'aléatoire. On peut construire un générateur pseudo-aléatoire à partir d'une fonction de hachage.

Par exemple, si H est une fonction de hachage, initialisée avec une graine c, on peut construire une suite pseudo-aléatoire à partir de la suite récurrente xi = H (xi-1 || 0), où x0=c.

Les générateurs pseudo-aléatoires sont nécessaires pour diverses applications cryptographiques, on les retrouve surtout pour la génération des clés pour le chiffrement asymétrique, la confidentialité des échanges sur les réseaux sans fil ou encore dans les signatures électroniques.

## I.6. Conclusion :

Dans ce chapitre, nous avons défini l'algorithme de hachage cryptographique. D'autre part, ses propriétés : la résistance aux attaques recherchant des collisions, les pré-images et les secondes pré-images. Enfin, une description de la famille SHA et en particulier l'algorithme SHA-1.

## II.1. Introduction

Un système embarqué est défini comme un système électronique et informatique autonome, souvent temps réel, spécialisé dans une tâche bien précise. Le terme désigne le matériel informatique ainsi que le logiciel utilisé. Ses ressources sont généralement limitées. Cette limitation est d'ordre spatial (encombrement réduit) et énergétique (consommation restreinte).

## II.2. Les circuits logiques programmables [5]

Un circuit logique programmable, est un circuit logique intégré qui peut être reprogrammé après sa fabrication. Il est composé de nombreuses cellules logiques élémentaires librement assemblables. Un circuit logique programmable permet la réalisation de circuits logiques à grand nombre de variables d'entrée et de sortie, selon la spécification de l'utilisateur et en utilisant un même support matériel. Actuellement, on trouve différentes familles de circuits programmables tels que les PLD, les CPLDs et les FPGAs (figure II-1).

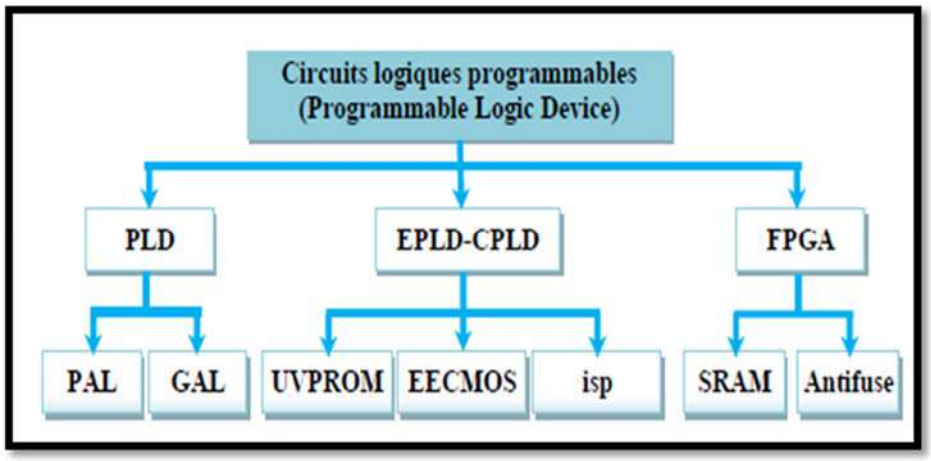



**Figure II.1.** *Les différents circuits programmables*

Les FPGAs, ou bien les réseaux logiques programmables, sont des composants électroniques programmables de la famille des PLDs. Un FPGA est un ensemble de blocs logiques élémentaires que l'utilisateur peut interconnecter pour réaliser les fonctions logiques de son choix. La densité des portes est importante et sans cesse en évolution. L'avantage d'un FPGA est leur grande souplesse dans leur technologie permettant une réutilisation à volonté et en un temps très court dans des algorithmes différents. Le progrès technologique permet de faire des composants toujours plus rapides et à plus haute intégration, autorisant la programmation d'applications importantes. Ils sont constitués d'un réseau de blocs logiques, de blocs mémoires, de blocs dédiés et d'entrées/sorties. L'ensemble est relié par un réseau d'interconnexions programmable. Les blocs logiques permettent de réaliser des opérations avec

quelques variables à travers une LUT (Look Up Table). Le résultat peut être éventuellement stocké dans un registre, les blocs RAM permettent d'implanter des mémoires adressables et des FIFO (First In, First Out), les blocs dédiés permettent de réaliser facilement de nombreuses opérations de traitement (blocs DSP), de gérer l'horloge, ou des interfaces de communication (RocketIO, Ethernet, PCI Express). Les nombreux ports permettent également de connecter des périphériques de la plateforme matérielle à base de FPGA.

Les FPGA se programment grâce à leurs LUT et leur réseau d'interconnexion. La programmation se fait avec un langage de programmation hardware tel que le VHDL ou bien le Verilog. L'outil de développement transforme cette description en un fichier de configuration du FPGA en plusieurs étapes : le HDL (Hardware Design Language) doit d'abord être synthétisé (transformé en éléments logiques de base), puis les éléments doivent être placés sur le composant (placement) et enfin interconnectés (routage). La Figure II.2 montre la structure interne d'un FPGA de type matrice symétrique. L'utilisateur peut programmer la fonction réalisée par chaque cellule (CLB : Configurable Logic Block). On programme aussi les interconnexions entre les cellules. Les FPGA sont utilisés pour un développement rapide et bon marché des ASIC.

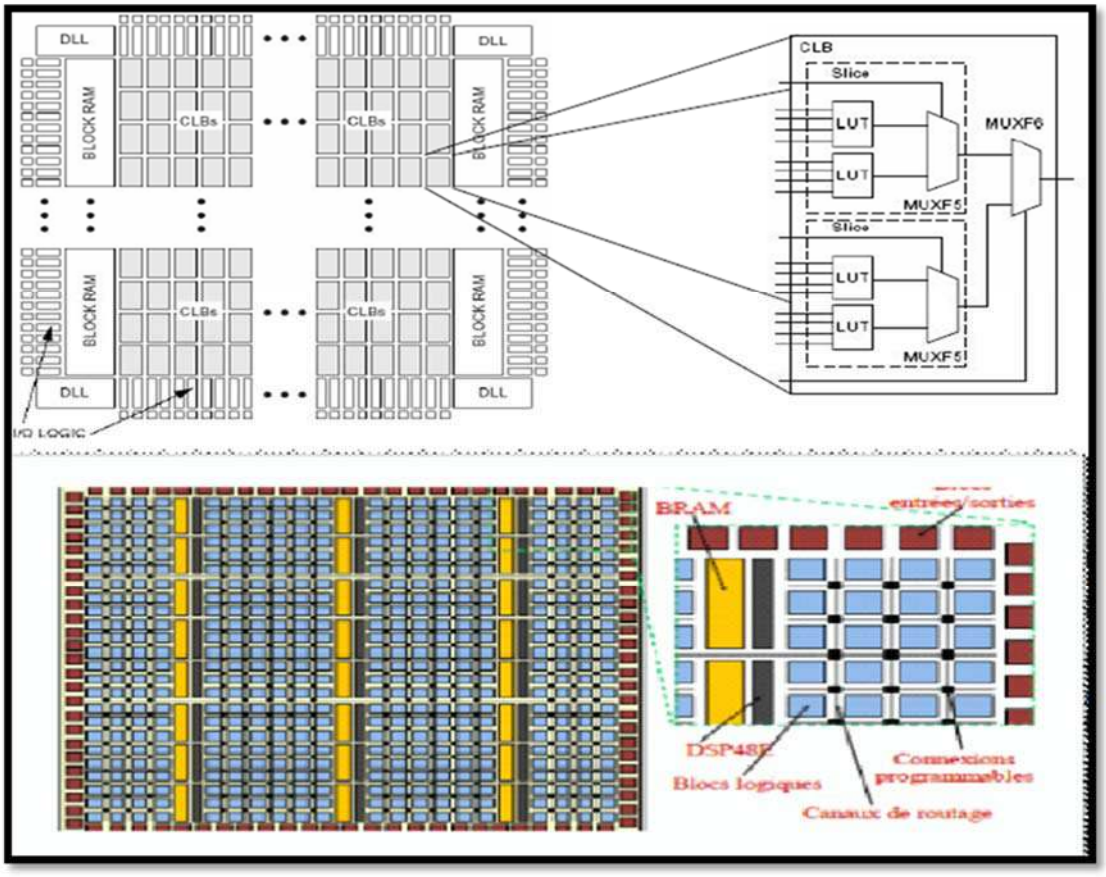



**Figure II.2.** *Architecture conceptuelle d'un FPGA (Architecture, Interconnexions et CLB)*

## *II.3. Les systèmes numériques embarqués [5]*

De nos jours, les systèmes électroniques embarqués ont été introduits dans de nombreux domaines d'applications tels que l'automobile, l'avionique, les systèmes multimédia, les appareils électroménagers, etc. L'architecture générale d'un système sur puce intègre trois principaux types de composants qui sont : les unités de traitements, les mémoires et les systèmes de communication

### *II.3.1. Codesign : Conception conjointe logicielle/matérielle*

Les techniques classiques de conception de systèmes imposent de séparer le hardware du software dans le cycle de conception. En outre, le consensus grandissant pour un modèle de systèmes informatiques «en couches» semble éviter aux développeurs software de telles préoccupations. Ainsi, les concepteurs de hardware spécifient leurs produits sans totalement apprécier les besoins, en ressources matérielles, des logiciels qui s'exécuteront dessus. De leur côté, les concepteurs de logiciels ne tirent pas suffisamment profit des ressources matérielles de la plate-forme hardware sur laquelle seront implantés leurs produits. Finalement, la phase d'intégration des deux parties nécessite souvent des modifications parfois majeures du logiciel et/ou du matériel. Cette approche présente peu de flexibilité dans l'évaluation des différentes options de conception et de répartition du hardware et du software. La tendance actuelle, principalement lorsqu'il s'agit de systèmes temps réel complexes, est de tenir compte dès la phase de spécification des interactions entre les deux parties. Ainsi, la séparation est effectuée le plus tard possible dans le cycle de conception tendant à unifier la spécification pour qu'elle englobe le hardware et le software. Cette approche est nommée "hardware/software codesign", ou "codesign". Le système est conçu à partir d'une spécification unique décrivant son architecture et/ou son comportement. Après la phase de spécification survient une phase de partitionnement du système ayant pour but de décomposer ce dernier en partitions comportant trois parties :

- Une partie matérielle implantée sous forme de circuits hardware. Cette partie peut être un IP récupéré à partir d'une bibliothèque.
- Une partie logicielle implantée sous forme d'un programme exécutable sur un processeur.
- Une interface de communication entre les deux parties.

### *II.3.2. Flot de la conception conjointe logicielle/matérielle*

Le développement de systèmes complexes fait de plus en plus appel à des composants logiciels et spécifiques. La conception conjointe logicielle/matérielle apporte des solutions à ce

type de développement. Un flot de conception standard est généralement constitué de trois étapes principales : la spécification, le partitionnement, et la vérification conjointe HW/SW (Figure II-3).

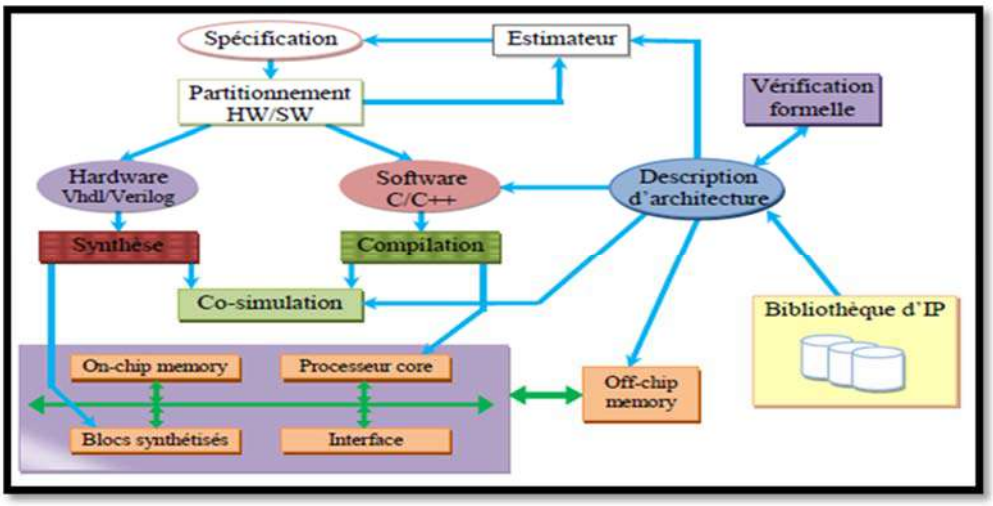



**Figure II.3.** *Flot de conception standard*

Plusieurs outils et compagnies ont adopté cette approche de codesign par le fait de sa simplicité de synthèse et d'intégration de nouveaux composants. Parmi ces outils de développement notons celui proposé par Xilinx (EDK pour Embedded Development Kit).

### *II.3.3. Les composants virtuels (IPs)*

Les outils de développement et de conception, des circuits spécialisés, sont de plus en plus performants malgré l'augmentation considérable de la capacité des composants matériels. Cependant il est indispensable de réutiliser des conceptions déjà réalisées dans ces outils de développement. En effet, afin de réduire le temps de conception et de prototypage des systèmes sur puce, une approche de réutilisation des composants a été introduite. Cette approche consiste à reprendre des modèles de composants au niveau RTL qui sont considérés comme étant des propriétés intellectuelles (IP). Les IPs peuvent être définis comme des composants virtuels, ce sont des circuits génériques ainsi que des outils associés que l'on peut trouver sur le marché ou bien sous une forme téléchargeable (Open Source).

Un IP peut être une fonction de base ou bien un bloc plus complexe. L'intégration de cœurs de processeurs est également possible, tel que le MicroBlaze ou bien le NIOS. Ces processeurs sont relativement simples, et personnalisables en fonction de l'application. L'assemblage des IPs permet de décrire rapidement des applications complexes. On peut utiliser des IP sans les avoir conçus, en ayant uniquement une description haut-niveau (vue externe et fonctionnelle) de leur comportement. Un composant IP peut apparaître sous différentes formes :

- ***IP Logiciel*** *(softcore)* : Le composant est livré sous sa forme HDL synthétisable. Son principal avantage est sa portabilité. La propriété du fichier source est en soi la meilleure documentation, on peut ainsi maintenir le produit pendant des années et éventuellement modifier à volonté la source et même changer de technologie cible. L'inconvénient majeur est qu'il ne peut être prédictif en termes de superficie, puissance et temps. Le travail d'optimisation du circuit final est à la charge de l'intégrateur du système.
- ***IP Matériel*** *(hardcore)* : Dans ce cas, la description du bloc IP est au niveau porte logique, optimisée pour une technologie particulière (optimisation garantie). Cela englobe la netlist entière, le routage et l'optimisation pour une librairie technologique spécifique et un lay-out personnalisé. L'inconvénient est qu'il est moins flexible car le processus est dépendant de la technologie, par contre il a l'avantage d'être prédictif.
- ***IP Firm*** *(firmcore)* : Le bloc IP firmcore offre un compromis entre les deux autres IPs, plus flexible que le hardcore, plus prédictif en termes de performance et de surface que le softcore (par exemple une interface USB où la vitesse de la norme ne pourra être assurée que dans une technologie bien définie). En général, le travail de synthèse HDL est déjà réalisé pour une technologie cible donnant lieu à une description par netlist (format EDIF par exemple).

## II.4. Les systèmes sur puce (SoC ) [5]

### *II.4.1. Les contraintes liées aux systèmes embarqués*

Les contraintes lors de la construction d'un système embarqué sont relatives au temps, imposés par l'interaction du système avec son environnement et aussi au caractère limité et/ou périssable des ressources dont dispose le système (énergie, mémoire, etc.). A ces contraintes s'ajoutent des contraintes de sûreté de fonctionnement plus ou moins importantes selon le type d'utilisation du système embarqué. En plus, un système embarqué exécute des tâches prédéfinies et a un cahier des charges contraignant à remplir, qui peut être de l'ordre de coût, d'espace compté ou bien les contraintes physiques (le système doit présenter un faible encombrement et un faible poids) ; de puissance de calcul ; de consommation énergétique (la plus faible possible), etc. Des choix doivent donc être réalisés pour privilégier un ou plusieurs de ces critères. Par exemple, pour un ordinateur portable, l'autonomie est souvent négligée par rapport aux contraintes de performances et de taille. En effet, un ordinateur peu performant perd tout son intérêt, d'autant plus qu'il est souvent possible de travailler à proximité d'une prise de courant. Pour un téléphone portable au contraire, l'intérêt étant de pouvoir téléphoner de partout, l'autonomie est un critère primordial.

### *II.4.2. Les SoC et les SoPC*

L'approche SoC a été créée dans un premier temps pour le développement d'ASIC mais a été étendue pour le développement de FPGA. On parle alors de SoPC pour System on Programmable Chip. Le SoPC (acronyme a été inventé par Synopsys) correspond à l'intégration d'un ou plusieurs cours de processeurs et de ses périphériques sur une même puce programmable de type FPGA. Le logiciel est alors situé soit dans une mémoire du circuit FPGA si l'empreinte mémoire le permet, soit dans une mémoire externe le plus souvent. Les deux approches peuvent être résumées à :

- ✓ ***L'approche SoC*** *(généralement en technologie ASIC)* : répond aux besoins de performances et d'intégration mais elle est peu adaptée à l'évolutivité des systèmes et reste réservée aux grands volumes de production. En effet, un SoC peut être retenu pour les applications destinées au grand public. Il permet de meilleures performances en termes de consommation, de vitesse et de surface, mais, la fabrication et le test sont des étapes longues et coûteuses. De plus, un SoC est figé et n'est donc pas réutilisable pour une autre application.

- ✓ ***L'approche SoPC*** *(technologie FPGA) :* Spécialement pour résoudre les problèmes de développement et les prototypages rapides, ainsi que les composants reconfigurables en quelques « ms » et à volonté. Il permet donc un développement et prototypage rapide du système. En contrepartie, la densité d'intégration est généralement moindre (~10 Millions de portes) et la consommation d'énergie est plus grande avec une performance plus faible que celle du SoC.

Les composants FPGA permettent d'implanter deux types de SoPC. Le premier type de SoPC utilise uniquement des IP soft de processeur tels que le MicroBlaze et le NIOS. Le nombre de processeurs intégrés sur un même composant dépend des ressources du composant utilisé. Le deuxième type de composants en plus de la logique reconfigurable, implémente en dur un ou plusieurs cœurs de processeurs (PowerPC pour Xilinx, ARM pour Altera). Ceci n'empêche en rien l'utilisation de processeurs Soft (netlist). Ces deux types de SoPC utilisent les mêmes flots de conception. Chez Xilinx, tout le flot de conception est regroupé sous la forme d'un seul environnement dit Xilinx Platform Studio (XPS).

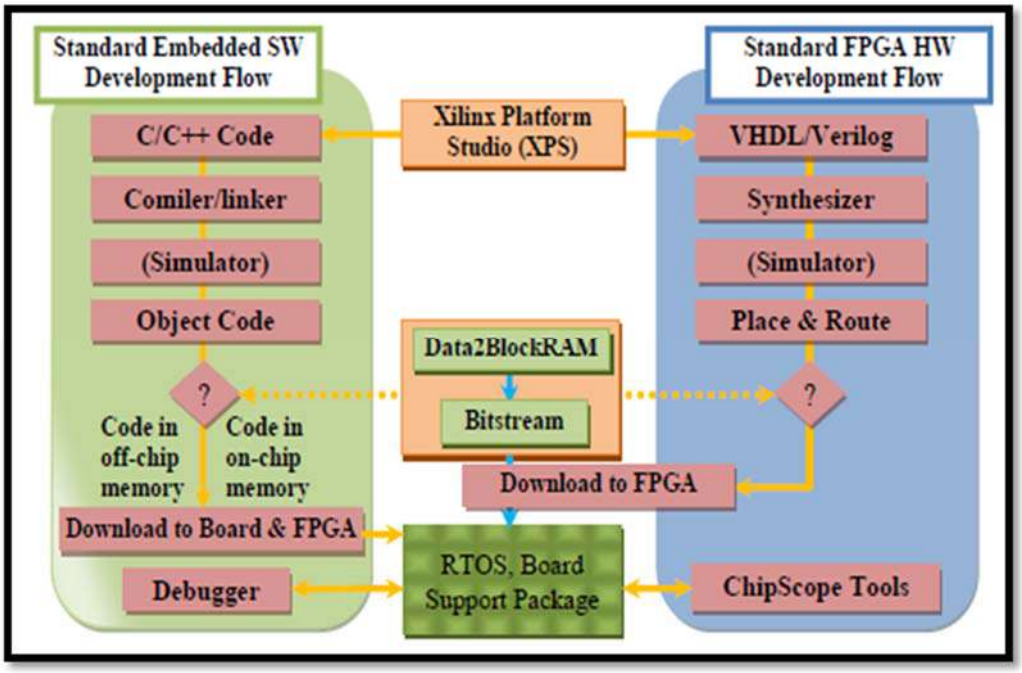



**Figure II.4**. *Flot de conception Xilinx*

## II.5. Communications sur puce [5]

Les systèmes logiques sur puce sont souvent conçus de manière à être dimensionnés pour une seule application particulière. Chaque application a sa propre architecture, de ce fait l'extension ou la réutilisation de telle architecture est difficile et nécessite un effort de réadaptation et pour cette raison plusieurs types de communication sur puce ont été proposés. Un système de communication permet d'établir des connexions entre tous les composants dans un SoC. Il se présente sous forme de bus ou bien de réseau de communication.

Spécialement pour des systèmes sur puce, IBM a proposé l'architecture CoreConnect composée de trois bus qui permettent la connexion : de cœurs de processeur, de logique dédiée, et différents autres accélérateurs matériels. Les trois bus de CoreConnect sont :

- ***Processor Local Bus (PLB) :*** Généralement, le PLB fournit un chemin de données large bande. En effet il est utilisé pour connecter des composants tels que les cœurs de processeurs, les interfaces avec les mémoires externes, et les contrôleurs DMA (Direct Memory Accès).
- ***On-chip Peripheral Bus (OPB) :*** L'OPB est utilisé pour réduire la charge du bus PLB. Il est plus adapté aux composants tels que les ports série, les ports parallèles, UARTs, GPIO et tous les composants à faible largeur de bande. Un Maître sur le PLB peut

accéder aux composants connectés à l'OPB via une macro passerelle (Bridge). Cette passerelle est vue en tant que maître par l'OPB et en tant qu'esclave par le PLB. Les registres d'état et de configuration de faible performance sont généralement lus et écrits via le bus DCR.

- ***Device Control Register (DCR) :*** Le DCR est principalement utilisé pour accéder aux registres de l'état et de contrôle au sein des différents bus PLB et OPB.

Xilinx a repris le comportement de tous les bus CoreConnect d'IBM et les a décrits en VHDL, ce qui permet d'avoir des bus paramétrables. En effet, pour simplifier la connexion d'un module logique utilisateur à un des bus CoreConnect, Xilinx a ajouté une interface de bus préconçue paramétrable appelée IPIF (IP InterFace). L'IPIF gère les signaux du bus, le protocole de communication et plus généralement, l'ensemble des caractéristiques du bus. L'IPIF intègre aussi une interface de gestion, du logique utilisateur, appelée IPIC (IP InterConnect). Lorsque le logique utilisateur est conçu avec un IPIC, elle peut être portable et facilement réutilisable sur d'autres bus en changeant seulement l'IPIF. Des fichiers VHDL qui instancient l'IPIF et fournissent le code nécessaire à l'utilisateur pour ajouter ses modules, simplifient la tâche de connexion du logique utilisateur. Ces fichiers sont les User Core Reference Design. Un système de traitement réalisé par EDK doit être connecté le bus OPB et/ou le bus PLB, le périphérique ajouté au système (accélérateur matériel) doit être connecté à OPB ou bien PLB. La figure II-7 montre la connexion du bus PLB dans un système à base d'un MicroBlaze, d'une DDR2 et d'une mémoire Compact Flash.

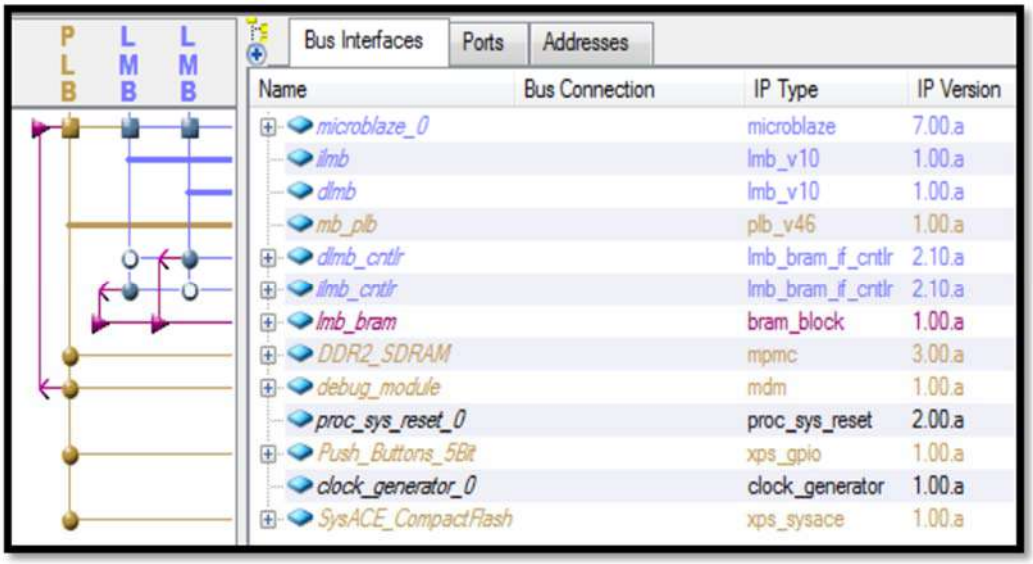



**Figure II.5.** *Le bus PLB dans un système à base de MicroBlaze*

## II.6. Plateformes multi-composants [5]

La maîtrise des Socs nécessite des compétences en électronique et en informatique d'un niveau assez élevé. Pour aider les nouveaux utilisateurs de SoCs à bien maîtriser l'ensemble de la chaîne de mise en place d'applications, des plateformes de prototypage rapide existent aujourd'hui sur le marché. Ces plateformes utilisent un gros FPGA comme cœurs et disposent, en plus, de mémoires SRAM, de convertisseurs A/N et N/A, des interfaces (boutons poussoirs, voyants LED, connecteurs d'usage général, connecteurs pour debug, zone de prototypage, connecteur RS232 ainsi qu'un connecteur JTAG), d'oscillateurs, sortie VGA, etc. Généralement, la configuration des composants se fait par liaison JTAG.

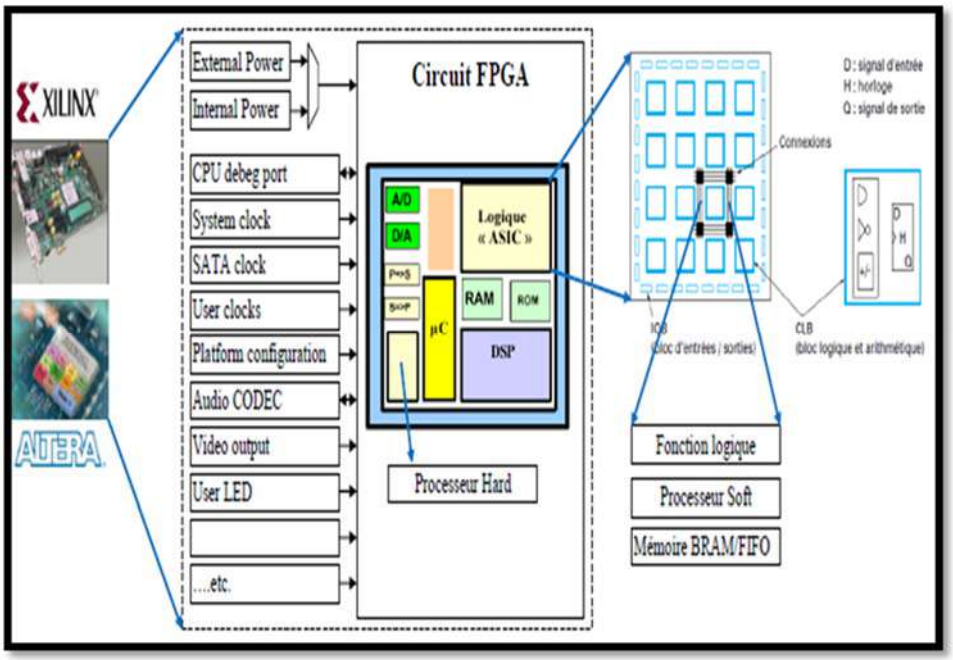



**Figure II.6.** *Détails d'une plateforme multi-composants*

### *II.6.1. Les processeurs embarqués*

Avec l'augmentation exponentielle des performances et de la densité d'intégration des portes logiques, les cœurs de processeurs sont devenus essentiels dans le développement des SoC et SoPC. Un cœur de processeur est la partie matérielle qui traite les données en fonction du décodage des instructions contenues en mémoire. La grande quantité de données à prendre en compte ainsi que les contraintes en temps souhaitées, font du cœur de processeur un élément critique d'un système. Le cœur de processeur idéal doit donc fournir le maximum de performances sur une surface la plus réduite possible. Il doit être le plus flexible possible et ne pas subir de dégradation de performances en fonctions des différents environnements rencontrés (Outils de développement, taille de la cible ou électronique adjacente).

Les cœurs de processeurs peuvent être, soit des macro-cellules spécialisées, dites processeur Hardcore (PowerPC et ARM), soit des modules synthétisables, dits processeur softcore (MicroBlaze, NIOS-II, etc.). Ces soft corés sont configurables à travers plusieurs paramètres (profondeur du pipeline, taille du cache, présence ou non de multiplieurs, etc.). Un processeur softcore est réalisé à partir de la logique programmable présente sur la puce. Ainsi, un concepteur peut mettre autant de processeurs softcore que la puce le permet. A titre illustratif, un processeur MicroBlaze occupe seulement 4% de la surface d'un FPGA de la famille Virtex5-LX50 (composant à 28.800 cellules logiques), et 1,7% de la surface d'un FPGA de la famille Virtex5-LX110T (composant à 69.120 cellules logiques). A l'inverse, le concepteur ne choisit pas le nombre de processeurs hardcore présents dans un FPGA. L'intérêt majeur des FPGA est la possibilité de faire du prototypage rapide, la proximité des cœurs de processeurs avec de la logique programmable rend possible la spécialisation de ces processeurs pour concevoir un système sur puce.

La capacité d'intégration des FPGA permet d'y implémenter des processeurs IPs soft. Ce sont des processeurs décrits à l'aide d'un langage de description matériel (VHDL ou bien Verilog) pouvant être synthétisable. Ces processeurs reprennent exactement le fonctionnement et les performances des autres processeurs implémentés en hard, l'avantage ici réside dans leur capacité d'être paramétrée. En effet, un processeur softcore s'adapte aux besoins de ses développeurs ainsi qu'aux contraintes matérielles (périphériques, performances, ressources, consommation, etc). Le fait de paramétrer un processeur permet d'avoir une flexibilité et une facilité à faire un compromis performance-coût accélérant ainsi le cycle de conception. Cependant, les performances d'un softcore sont, généralement, inférieures à celles d'un processeur hardcore. Parmi les produits existant en ce moment sur le marché on peut citer :

### *a. Le processeur MicroBlaze de Xilinx*

Le MicroBlaze est un microprocesseur softcore RISC, d'architecture Harvard, entièrement 32 bits. En effet, il possède :

- 32 registres internes de 32 bits,
- un bus instructions interne (ILMB),
- un bus donnée interne (DLMB),
- un bus d'instructions externe (IOPB),
- un bus de données externe (DOPB).

Le processeur comporte 70 options de configuration 2 permettant à l'utilisateur de sélectionner ou de paramétrer les composants internes selon ses besoins :

- un pipeline à 3 ou 5 niveaux,

- utilisation des multiplieurs câblés du FPGA (sauf sur Spartan II)
- opérateur de division
- opérateur de décalage (Barrel Shifter)
- FPU (Floating Point Unit)
- mémoires cache instructions et données
- logique de débug (hardware breakpoints)

En 2006, il occupait entre 900 et 2600 'logic cells' et atteignait une fréquence maximale de 80 à 180 MHz selon la plateforme FPGA et les options sélectionnées.

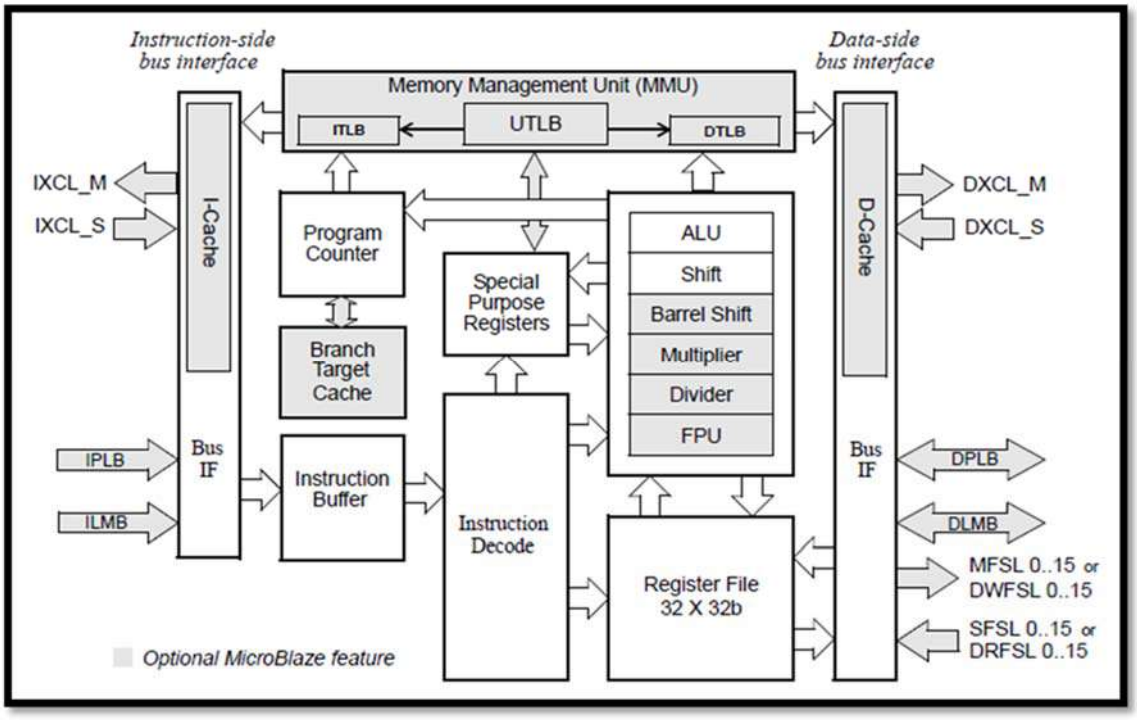



**Figure II.7.** *Schéma-bloc fonctionnel du noyau MicroBlaze*

### *II.6.2. Les Outils de développement de Xilinx*

En plus du matériel, Xilinx fourni des outils de conception des systèmes nous permettant de mettre au point des projets sur leurs différentes plateformes reconfigurables. En effet, Xilinx fournit dans son pack d'outils différents soft permettant la création de systèmes embarqués sur puce, parmi ces softs on dénombre ISE (Integrated Software Environment) et EDK. L'outil ISE est spécialement pour la réalisation des projets d'IPs matériels à partir d'un code HDL. L'outil EDK, englobant SYSTEM GENERATOR FOR PROCESSOR et Xilinx Platform Studio (XPS), va nous permettre d'établir un lien direct entre la partie matérielle et logicielle d'un système. Tous les deux nous offrent la possibilité d'avoir un bitstream pour la programmation des FPGA suivant l'application ciblée.

### *a) Xilinx ISE*

ISE est un environnement intégré de développement de systèmes numériques ayant pour but la synthèse/implémentation matérielle sur FPGA. Les designs peuvent être décrits sous trois formes principales : sous forme de schémas, sous forme de HDL ou bien sous forme de diagrammes d'états. Pour cela, ISE intègre différents outils permettant de passer à travers tout le flot de conception d'un système numérique. Il inclut des éditeurs et outils pour la saisie du système, il dispose aussi d'un ensemble d'outils, de synthèse et d'implémentation, regroupés dans un seul flot de conception. Les détails de ces outils sont donnés comme suit :

1. Saisie de conception (Design Entry) : Éditeur HDL, éditeur de Machine à État Fini (State CAD), éditeur de schéma (Engineering Capture System (ECS)) et générateur d'IP (CORE Generator) ;
2. Synthèse : XST (Xilinx Synthesis Technology), Leonardo Spectrum de Mentor Graphics et Synplify de Cadence ;
3. Simulation : Générateur de TestBench HDL et intégration du simulateur ModelSim de Model Technology ;
4. Implémentation : Translate, MAP, placement et routage (PAR), floor planner, éditeur FPGA, timing Analyzer XPower, Fit (seulement CPLD) et Chipviewer (seulement CPLD) ;
5. Programmation du composant et formatage du fichier de Bitstream : BitGen et iMPACT.

### *b) Xilinx EDK*

L'environnement EDK permet de développer une application complète à base de processeur embarqué et de l'intégrer à un FPGA. EDK donne accès à tous les réglages nécessaires pour l'application embarquée que l'on souhaite créer. Il permet la programmation d'un ou plusieurs MicroBlazes et de leurs périphériques. Il va s'occuper de placer les programmes des MicroBlazes en mémoire lors de la programmation du FPGA.

EDK permet d'une part la création des accélérateurs matériels au niveau RTL à partir d'une description en VHDL (il est possible d'ajouter directement le code synthétisé avec une extension .ngc), et d'autre part de générer l'architecture globale du système en incluant les processeurs logiciels (CPU) ainsi que les contrôleurs d'entrée/sortie nécessaires. L'ensemble de ces composants vont être interconnectés via un bus, qui est lui aussi généré en fonction des besoins architecturaux de l'application. EDK permet aussi :

- Génération du logiciel : permet l'extraction des parties logicielles de l'application et de générer par la suite le code machine correspondant à chaque processeur de la plateforme matérielle.

- Génération des communications : permet de générer le système de communication de l'architecture en fonction du nombre de composants constituants la plateforme matérielle.

## II.7. Conclusion

Dans ce chapitre, nous avons exposé la conception des systèmes numériques embarqués. Ensuite nous avons décrit les circuits logiques programmables, en particulier les circuits FPGAS. En plus, la conception conjointe (logicielle et matérielle) a été détaillé. En fin, les outils de développement de xilinx (ISE et EDK) ont été présentés.

# *Chapitre 1*

Algorithmes de hachage cryptographique

# *Chapitre 2*

*Conception des systèmes numériques embarqués*

## III.1. Introduction :

Dans ce chapitre, nous présentons une implémentation d'une conception logicielle de l'algorithme de hachage cryptographique SHA1 sur une carte de développement basée sur un circuit FPGA de type Xilinx Spartan 3E-500. Enfin, nous exposons les résultats de deux tests pratiques réalisés entre la carte FPGA et un PC interfacés entre eux par le port série (RS232).

## III.2. Présentation de la carte FPGA :

La carte « Digilent Nexys2 » est une plate-forme complète pour le développement des systèmes basés sur les circuits FPGA Xilinx Spartan 3E. Elle est idéale pour les systèmes numériques de toutes sortes, y compris les systèmes avec processeurs embarqués basés sur le MicroBlaze de Xilinx.

Cette carte possède les caractéristiques suivantes **[6] :**

- FPGA de type Xilinx Spartan 3E de 500K portes ;
- Configuration FPGA basée sur l'USB 2 ;
- 16 Mo de SDRAM Micron et 16 Mo de ROM Intel Strataflash ;
- Oscillateur 50 MHz ;
- 8 LEDs, afficheur 7 segments 4 chiffres, 4 boutons, 8 commutateurs à glissière.



***Figure III.1*** *:* carte Digilent Nexys 2

## III.3. Outil Software utilisé :

Nous avons utilisé l'environnement de développement EDK de Xilinx englobant les deux outils XPS et SDK (voir chapitre II). Avec le Base System Builder, un outil d'assistance de XPS, on pourra créer rapidement un nouveau projet. XPS utilise les outils fournis avec ISE pour synthétiser, placer et router la logique dans le matériel. Des outils GNU sont fournis avec l'EDK et utilisés à l'intérieur de XPS pour concevoir le logiciel pour le système embarqué Microblaze. Les modules du système Microblaze, utilisés dans cette partie sont les suivantes :

- Microblaze
- Contrôleur LMB pour BRAM
- BRAM
- BUS PLB
- UART PLB
- MDM PLB

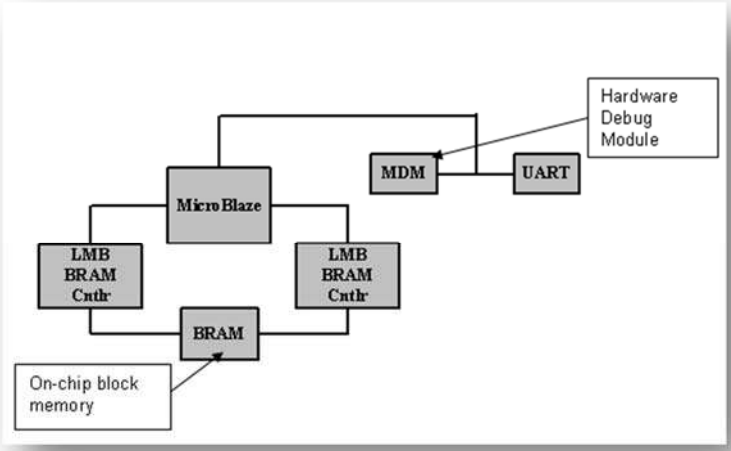



***Figure III.2 :*** Modules du système Microblaze utilisés.

## III.4. Réalisation du projet sous EDK :

### III.4.1. Utilisation du Base System Builder de XPS

On commence par l'ouverture de l'outil Xilinx Platform Studio (XPS). On crée un nouveau projet en utilisant « Base System Builder » (***Figure III.4***). On obtient la ***FigureIII.5***.




***Figure III.3 :*** Démarrage application XPS.

On indique au logiciel le chemin du répertoire du projet (pfesha1 créer auparavant) et on sélectionne le bus d'interconnexion de type PLB.

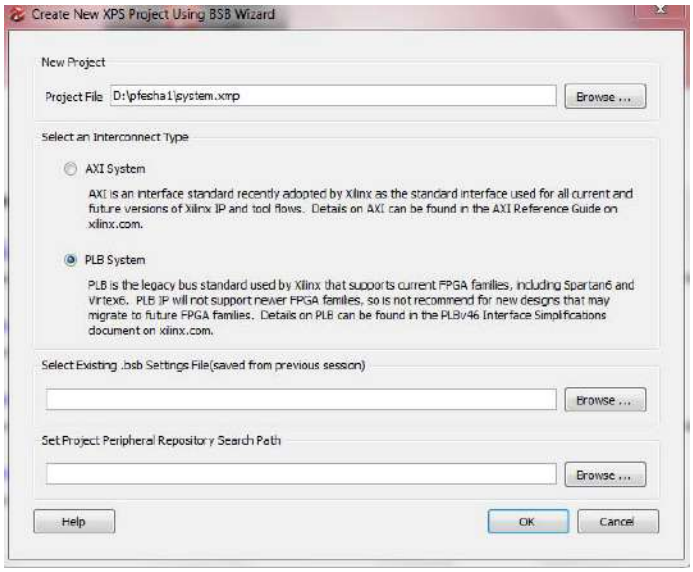



***Figure III.4 :*** création nouveau projet XPS.

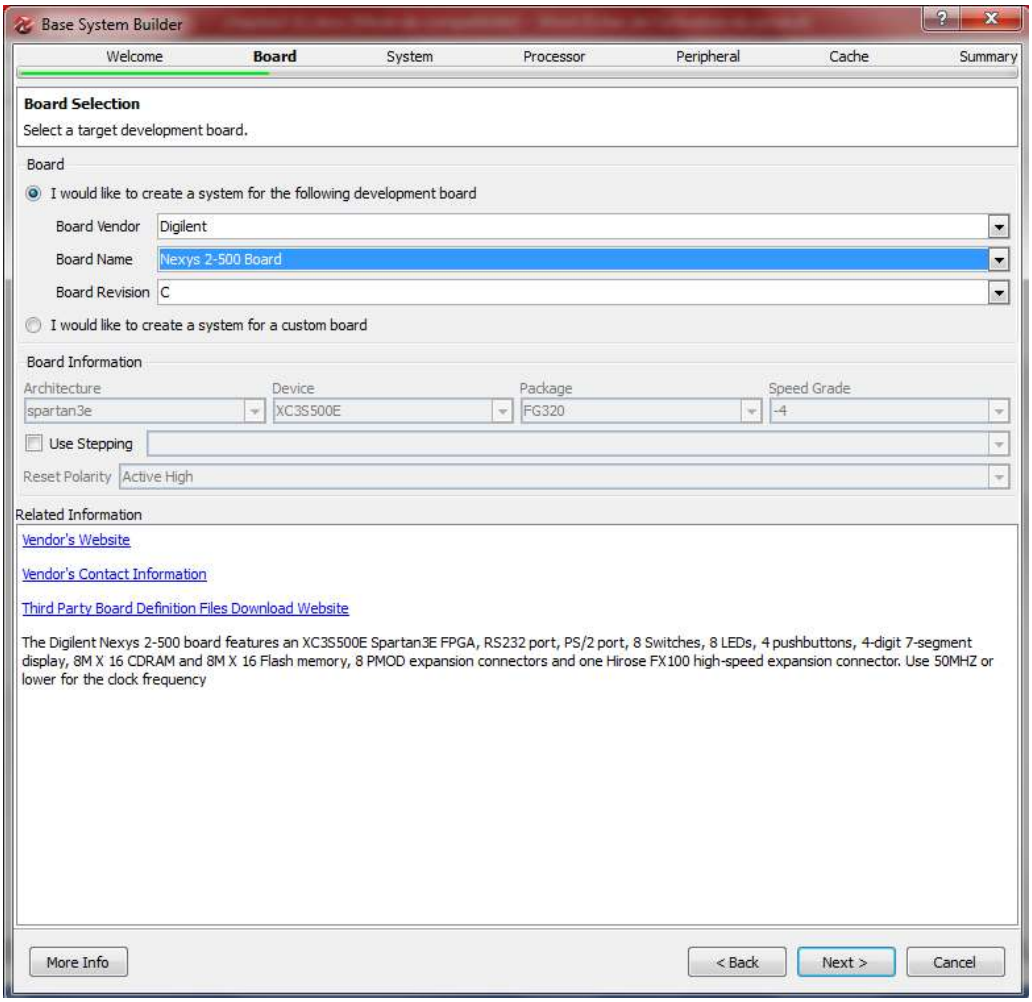



***Figure III.5 :*** choix carte FPGA.

Ensuite, on aura la ***Figure III.5***, où on choisit la carte de développement (nexys2-500 board) et on valide ce choix. On obtient la ***Figure III.6***.

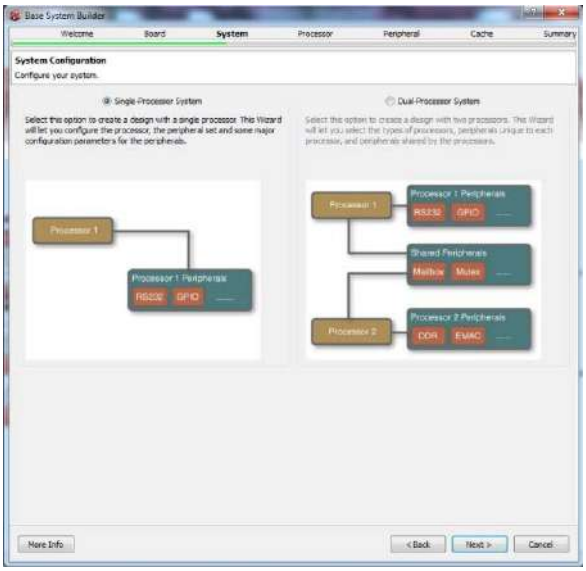



***Figure III.6 :*** Sélection solution mono processeur.

On choisit une architecture mono processeur. Après validation on passe à la ***Figure III.7***.

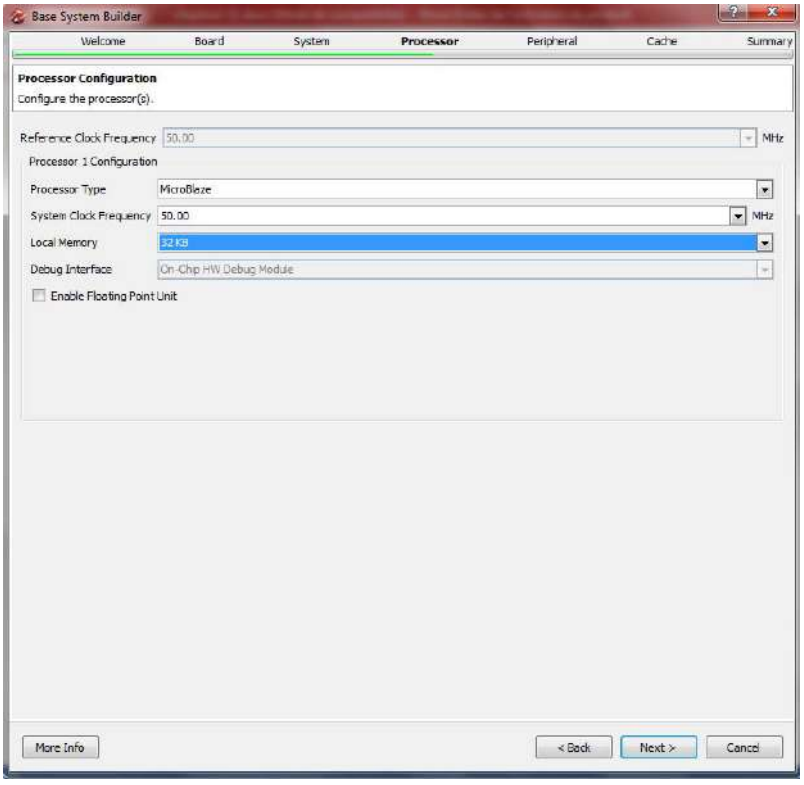



***Figure III.7 :*** sélection taille mémoire et fréquence d'horloge microblaze.

On fixe la taille mémoire interne à 32 koctets avec une fréquence d'horloge du processeur à 50 MHz. Ensuite, on supprime les modules non utilisés comme le montre la ***Figure III.8***.

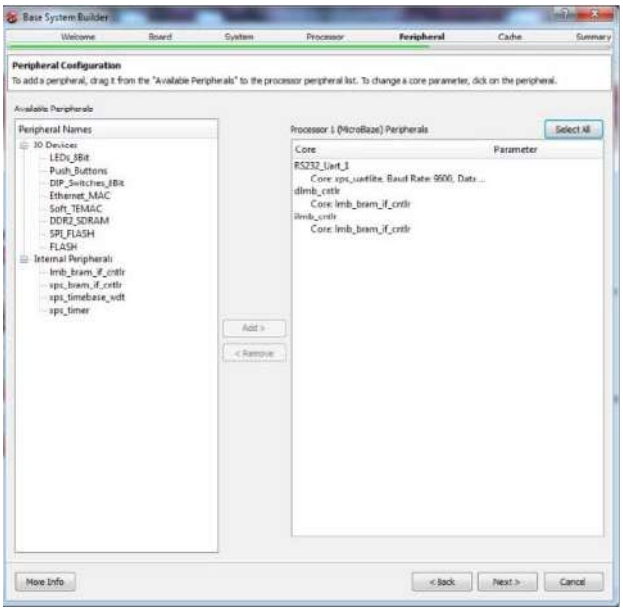



***Figure III.8*** : sélection modules utiles.

Après validation, le navigateur des projets de Xilinx Platform Studio charge notre projet (***Figure III.9***). À ce moment, XPS a généré un fichier de contraintes UCF (***Figure III.13***), un fichier projet (system.xmp), un fichier de spécification de matériel (system.mhs : ***Figure III.12***) et un fichier de configuration logiciel (system.mss) dans votre répertoire.

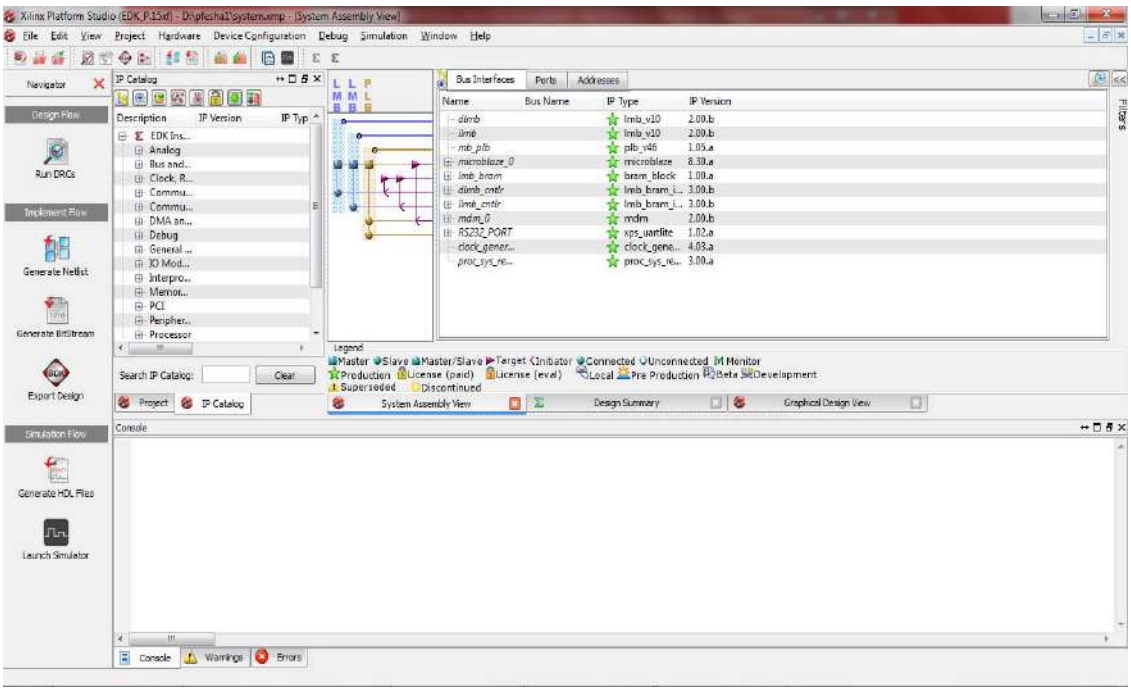



***Figure III.9 :*** Le navigateur des projets de Xilinx Platform Studio.

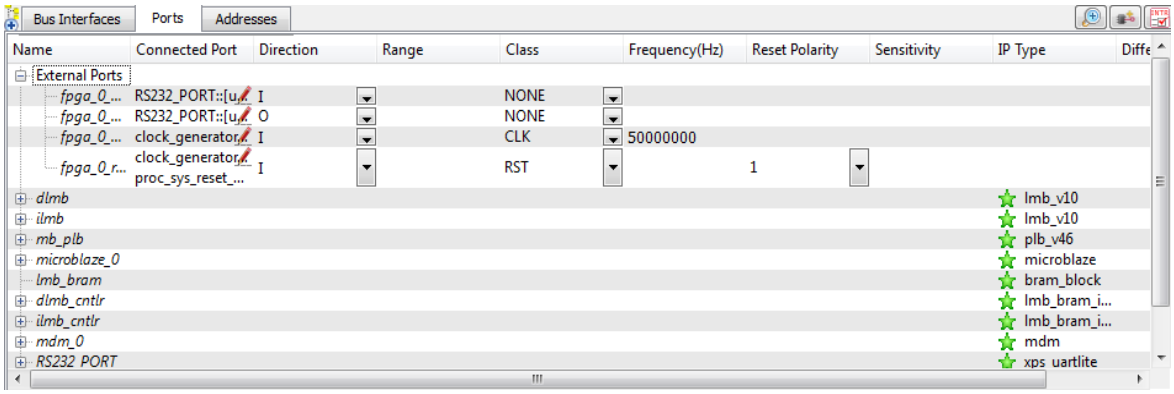



***Figure III.10*** : Capture d'écran des ports utilisés.

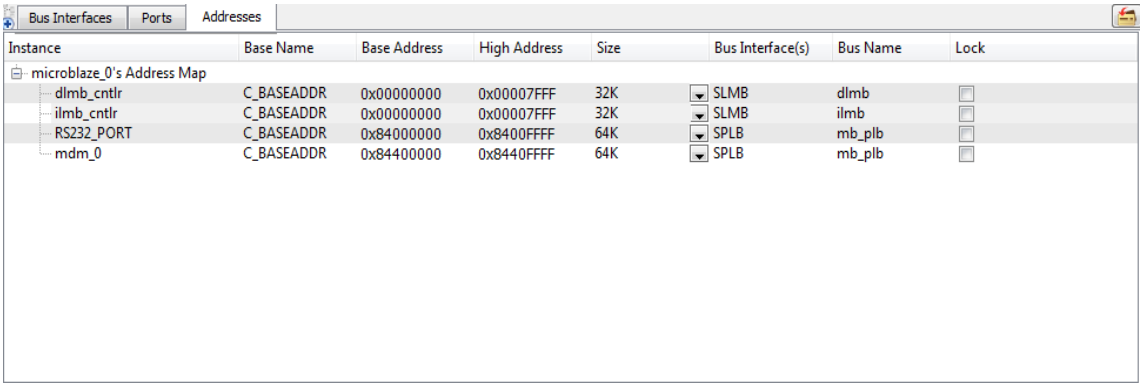



***Figure III.11*** : Capture d'écran des adresses réservées aux différents modules.

```
1
2  # ##############################################################################
3  # Created by Base System Builder Wizard for Xilinx EDK 14.1 Build EDK_P.15xf
4  # Sun Feb 26 10:44:02 2017
5  # Target Board:  Digilent Nexys 2-500 Board Rev C
6  # Family:    spartan3e
7  # Device:    XC3S500E
8  # Package:   FG320
9  # Speed Grade:  -4
10 # Processor number: 1
11 # Processor 1: microblaze_0
12 # System clock frequency: 50.0
13 # Debug Interface: On-Chip HW Debug Module
14 # ##############################################################################
15  PARAMETER VERSION = 2.1.0
16
17
18  PORT fpga_0_RS232_PORT_RX_pin = fpga_0_RS232_PORT_RX_pin, DIR = I
19  PORT fpga_0_RS232_PORT_TX_pin = fpga_0_RS232_PORT_TX_pin, DIR = O
20  PORT fpga_0_clk_1_sys_clk_pin = CLK_S, DIR = I, SIGIS = CLK, CLK_FREQ = 50000000
21  PORT fpga_0_rst_1_sys_rst_pin = sys_rst_s, DIR = I, SIGIS = RST, RST_POLARITY = 1
22
23
24 BEGIN microblaze
25  PARAMETER INSTANCE = microblaze_0
26  PARAMETER C_AREA_OPTIMIZED = 1
27  PARAMETER C_USE_BARREL = 1
28  PARAMETER C_DEBUG_ENABLED = 1
29  PARAMETER HW_VER = 8.30.a
30  BUS_INTERFACE DLMB = dlmb
31  BUS_INTERFACE ILMB = ilmb
32  BUS_INTERFACE DPLB = mb_plb
```

***Figure III.12 :*** *fichier de spécification de matériel MHS.*

```
1  #  Nexys 2-500 Board
2  Net fpga_0_RS232_PORT_RX_pin LOC=U6  |  IOSTANDARD = LVCMOS33  |  PULLUP;
3  Net fpga_0_RS232_PORT_TX_pin LOC=P9  |  IOSTANDARD = LVCMOS33  |  DRIVE = 2;
4  Net fpga_0_clk_1_sys_clk_pin TNM_NET = sys_clk_pin;
5  TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 50000 kHz;
6  Net fpga_0_clk_1_sys_clk_pin LOC=B8  |  IOSTANDARD = LVCMOS33;
7  Net fpga_0_rst_1_sys_rst_pin TIG;
8  Net fpga_0_rst_1_sys_rst_pin LOC=B18  |  IOSTANDARD = LVCMOS33;
9
```

***Figure III.13 :*** fichier de contraintes UCF.

### III.4.2. Implémentation du matériel

On sélectionne l'icône Generate Bitstream pour générer le fichier system.bit contenant l'architecture matérielle de notre système. Cette opération peut prendre une dizaine de minutes.

La ***Figure III.14*** représente le schéma RTL du système à base du microblaze implémenté. Le rapport des résultats concernant l'occupation de surface est comme le montre la ***Table III.1***.

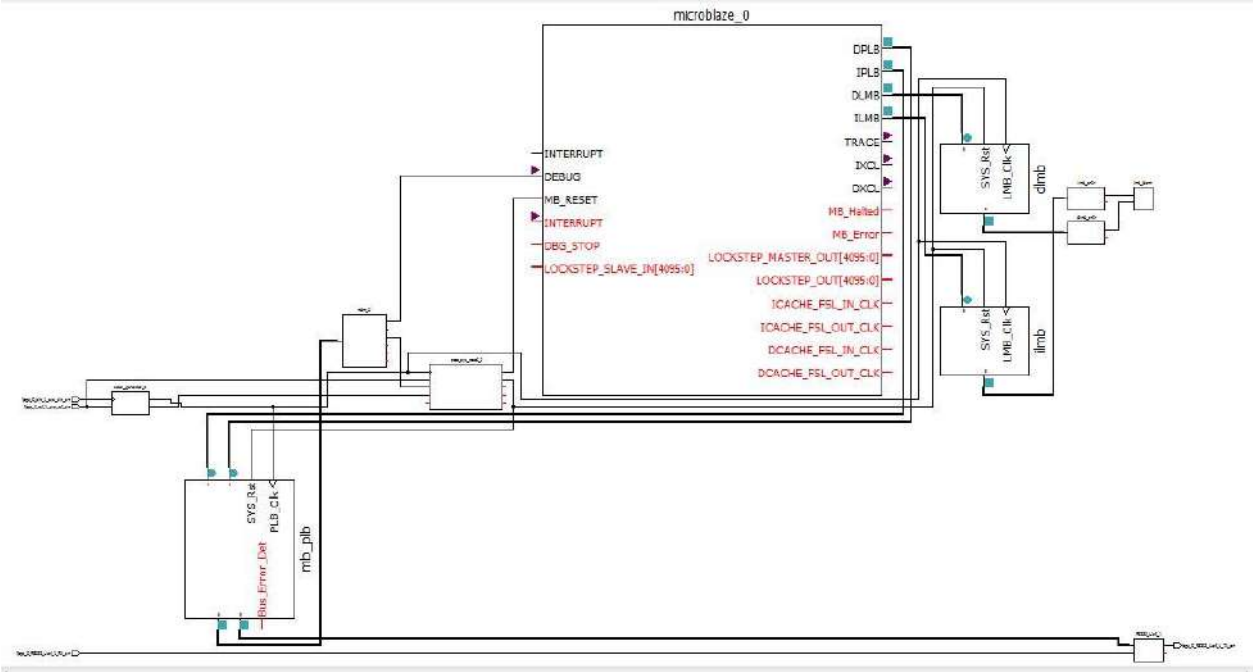



***Figure III.14 :*** Schéma RTL du système à base du MicroBlaze implémenté.

***Table III.1*** **:** Rapport des résultats concernant l'occupation de surface

| **Device Utilization Summary (actual values)** | | | | [-] |
|---|---|---|---|---|
| **Logic Utilization** | **Used** | **Available** | **Utilization** | **Note(s)** |
| Number of Slice Flip Flops | 1,024 | 9,312 | 10% | |
| Number of 4 input LUTs | 2,167 | 9,312 | 23% | |
| Number of occupied Slices | 1,500 | 4,656 | 32% | |
| Number of Slices containing only related logic | 1,500 | 1,500 | 100% | |
| Number of Slices containing unrelated logic | 0 | 1,500 | 0% | |
| Total Number of 4 input LUTs | 2,222 | 9,312 | 23% | |
| Number used as logic | 1,783 | | | |
| Number used as a route-thru | 55 | | | |
| Number used for Dual Port RAMs | 256 | | | |
| Number used as Shift registers | 128 | | | |
| Number of bonded IOBs | 4 | 232 | 1% | |
| IOB Flip Flops | 2 | | | |
| Number of RAMB16s | 16 | 20 | 80% | |
| Number of BUFGMUXs | 2 | 24 | 8% | |
| Number of DCMs | 1 | 4 | 25% | |
| Number of BSCANs | 1 | 1 | 100% | |
| Number of MULT18X18SIOs | 3 | 20 | 15% | |
| Average Fanout of Non-Clock Nets | 3.79 | | | |

### III.4.3. Utilisation de l'outil SDK

Après génération du fichier bitstream de l'architecture matérielle, c'est-à-dire, que notre système à base du microblaze est déjà implémenté. On passe à la conception logicielle de l'algorithme de hachage cryptographique SHA1 en langage C.

On clique sur l'icône Export Design, l'environnement SDK démarre. On crée un nouveau projet (pfesha1), avec le fichier sha1test.c. La ***Figure III.15*** montre l'environnement SDK avec notre projet développé.

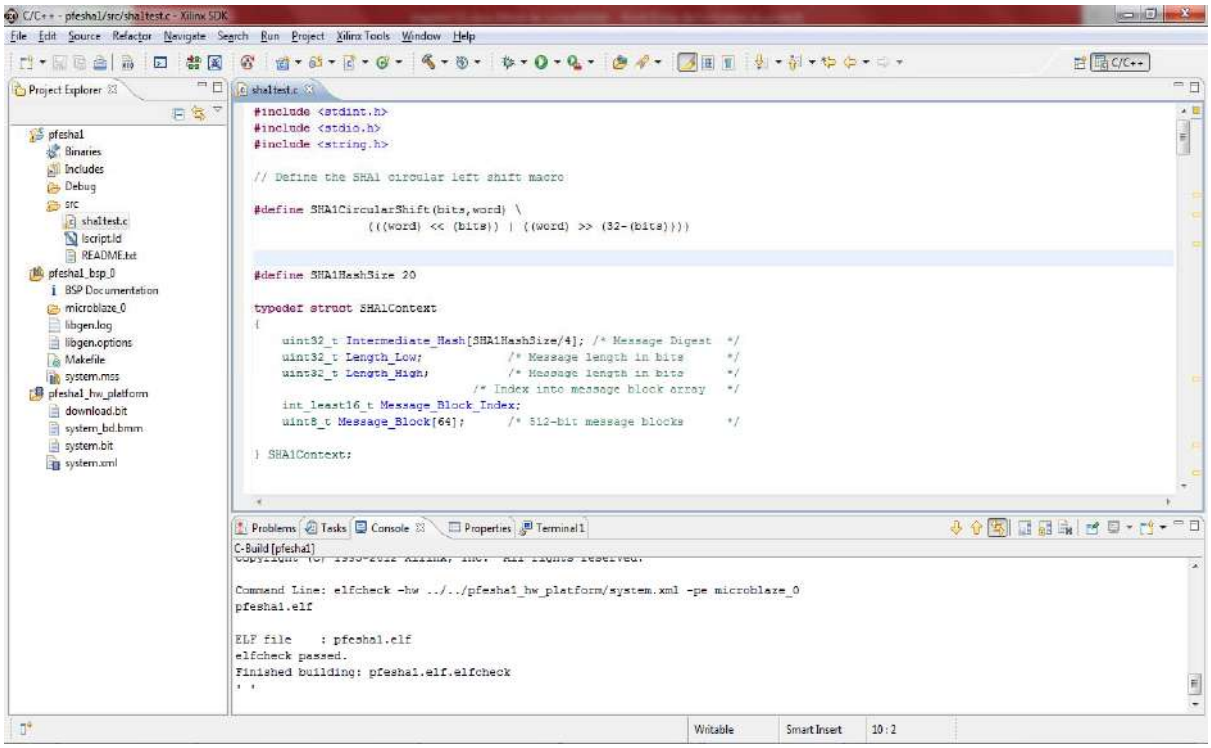



***Figure III.15 :*** Le navigateur des projets de Software Development Kit.

## III.5. Résultats des tests pratiques sur la carte FPGA :

Pour faire le test pratique du système développé sur la carte FPGA, on a utilisé les outils suivants :

- La carte FPGA.
- Un ordinateur portable.
- Un câble adaptateur liaison série RS232 / USB.
- Un hyper Terminal HERCULES 3.2.5 pour l'envoi et la réception des données en hexadécimal via le port USB émulé port série RS232.

Afin d'assurer la synchronisation entre l'ordinateur et la carte, on a utilisé la configuration suivante : le débit de transfert = 9600 bps ; la taille de la donnée = 8 bits ; pas de bits de parité ; 1 bit stop.

Et pour valider le fonctionnement de la fonction de hachage implémentée, nous avons réalisé les tests qui se trouvent dans le document officiel Fips-180-1 **[7].**

### III.5.1. Résultats du 1er test pratique :

Nous avons effectué un test dont le message d'origine est la chaîne de caractères « abc ».

**a) Bourrage du message :**

Le code ASCII du message en binaire est : 01100001 01100010 01100011. Il a une longueur de l=24.

Nous suivons les étapes décrites dans le paragraphe ***I.4.3.1*** Après l'ajout d'un "1", cela donne la chaîne binaire suivante : 01100001 01100010 01100011 1. Le nombre de bits est 25 et 423 "0" sont ajoutés, faisant un total de 448. Cela donne (en hexa) :

61626380 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000.

La représentation en deux mots de 24 est l'hexadécimal 00000000 00000018. Donc, le message bourré final est en hexadécimal :

61626380 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000018.

**b) Résumé du message :**

Le message contient un seul bloc (n=1). Il est composé des mots comme il est représenté dans la ***Figure III.16***.

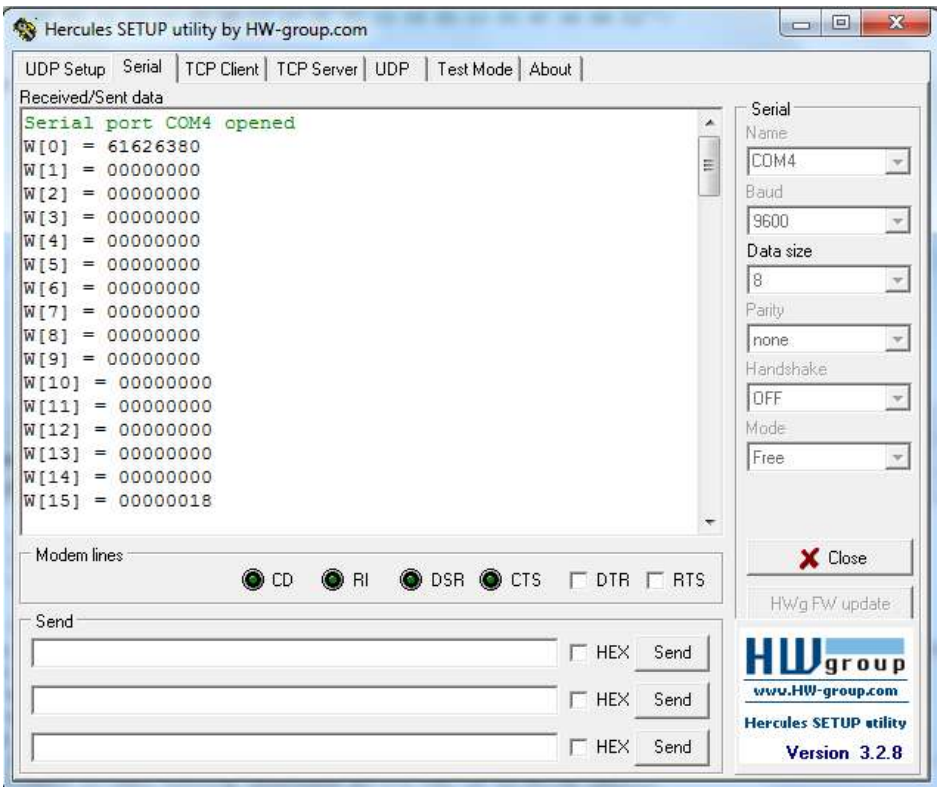



***Figure III.16 :*** Capture écran du message bourré.

Les valeurs obtenus en hexadécimal des registres A, B, C, D, et E pendant les 80 tours sont :

t = 0: 0116FC33 67452301 7BF36AE2 98BADCFE 10325476
t = 1: 8990536D 0116FC33 59D148C0 7BF36AE2 98BADCFE
t = 2: A1390F08 8990536D C045BF0C 59D148C0 7BF36AE2
t = 3: CDD8E11B A1390F08 626414DB C045BF0C 59D148C0
t = 4: CFD499DE CDD8E11B 284E43C2 626414DB C045BF0C
t = 5: 3FC7CA40 CFD499DE F3763846 284E43C2 626414DB
t = 6: 993E30C1 3FC7CA40 B3F52677 F3763846 284E43C2
t = 7: 9E8C07D4 993E30C1 0FF1F290 B3F52677 F3763846
t = 8: 4B6AE328 9E8C07D4 664F8C30 0FF1F290 B3F52677
t = 9: 8351F929 4B6AE328 27A301F5 664F8C30 0FF1F290
t = 10: FBDA9E89 8351F929 12DAB8CA 27A301F5 664F8C30
t = 11: 63188FE4 FBDA9E89 60D47E4A 12DAB8CA 27A301F5
t = 12: 4607B664 63188FE4 7EF6A7A2 60D47E4A 12DAB8CA
t = 13: 9128F695 4607B664 18C623F9 7EF6A7A2 60D47E4A
t = 14: 196BEE77 9128F695 1181ED99 18C623F9 7EF6A7A2
t = 15: 20BDD62F 196BEE77 644A3DA5 1181ED99 18C623F9
t = 16: 4E925823 20BDD62F C65AFB9D 644A3DA5 1181ED99
t = 17: 82AA6728 4E925823 C82F758B C65AFB9D 644A3DA5
t = 18: DC64901D 82AA6728 D3A49608 C82F758B C65AFB9D
t = 19: FD9E1D7D DC64901D 20AA99CA D3A49608 C82F758B
t = 20: 1A37B0CA FD9E1D7D 77192407 20AA99CA D3A49608
t = 21: 33A23BFC 1A37B0CA 7F67875F 77192407 20AA99CA
t = 22: 21283486 33A23BFC 868DEC32 7F67875F 77192407
t = 23: D541F12D 21283486 0CE88EFF 868DEC32 7F67875F
t = 24: C7567DC6 D541F12D 884A0D21 0CE88EFF 868DEC32
t = 25: 48413BA4 C7567DC6 75507C4B 884A0D21 0CE88EFF
t = 26: BE35FBD5 48413BA4 B1D59F71 75507C4B 884A0D21
t = 27: 4AA84D97 BE35FBD5 12104EE9 B1D59F71 75507C4B
t = 28: 8370B52E 4AA84D97 6F8D7EF5 12104EE9 B1D59F71
t = 29: C5FBAF5D 8370B52E D2AA1365 6F8D7EF5 12104EE9
t = 30: 1267B407 C5FBAF5D A0DC2D4B D2AA1365 6F8D7EF5
t = 31: 3B845D33 1267B407 717EEBD7 A0DC2D4B D2AA1365
t = 32: 046FAA0A 3B845D33 C499ED01 717EEBD7 A0DC2D4B
t = 33: 2C0EBC11 046FAA0A CEE1174C C499ED01 717EEBD7
t = 34: 21796AD4 2C0EBC11 811BEA82 CEE1174C C499ED01
t = 35: DCBBB0CB 21796AD4 4B03AF04 811BEA82 CEE1174C
t = 36: 0F511FD8 DCBBB0CB 085E5AB5 4B03AF04 811BEA82
t = 37: DC63973F 0F511FD8 F72EEC32 085E5AB5 4B03AF04

t = 38: 4C986405 DC63973F 03D447F6 F72EEC32 085E5AB5
t = 39: 32DE1CBA 4C986405 F718E5CF 03D447F6 F72EEC32
t = 40: FC87DEDF 32DE1CBA 53261901 F718E5CF 03D447F6
t = 41: 970A0D5C FC87DEDF 8CB7872E 53261901 F718E5CF
t = 42: 7F193DC5 970A0D5C FF21F7B7 8CB7872E 53261901
t = 43: EE1B1AAF 7F193DC5 25C28357 FF21F7B7 8CB7872E
t = 44: 40F28E09 EE1B1AAF 5FC64F71 25C28357 FF21F7B7
t = 45: 1C51E1F2 40F28E09 FB86C6AB 5FC64F71 25C28357
t = 46: A01B846C 1C51E1F2 503CA382 FB86C6AB 5FC64F71
t = 47: BEAD02CA A01B846C 8714787C 503CA382 FB86C6AB
t = 48: BAF39337 BEAD02CA 2806E11B 8714787C 503CA382
t = 49: 120731C5 BAF39337 AFAB40B2 2806E11B 8714787C
t = 50: 641DB2CE 120731C5 EEBCE4CD AFAB40B2 2806E11B
t = 51: 3847AD66 641DB2CE 4481CC71 EEBCE4CD AFAB40B2
t = 52: E490436D 3847AD66 99076CB3 4481CC71 EEBCE4CD
t = 53: 27E9F1D8 E490436D 8E11EB59 99076CB3 4481CC71
t = 54: 7B71F76D 27E9F1D8 792410DB 8E11EB59 99076CB3
t = 55: 5E6456AF 7B71F76D 09FA7C76 792410DB 8E11EB59
t = 56: C846093F 5E6456AF 5EDC7DDB 09FA7C76 792410DB
t = 57: D262FF50 C846093F D79915AB 5EDC7DDB 09FA7C76
t = 58: 09D785FD D262FF50 F211824F D79915AB 5EDC7DDB
t = 59: 3F52DE5A 09D785FD 3498BFD4 F211824F D79915AB
t = 60: D756C147 3F52DE5A 4275E17F 3498BFD4 F211824F
t = 61: 548C9CB2 D756C147 8FD4B796 4275E17F 3498BFD4
t = 62: B66C020B 548C9CB2 F5D5B051 8FD4B796 4275E17F
t = 63: 6B61C9E1 B66C020B 9523272C F5D5B051 8FD4B796
t = 64: 19DFA7AC 6B61C9E1 ED9B0082 9523272C F5D5B051
t = 65: 101655F9 19DFA7AC 5AD87278 ED9B0082 9523272C
t = 66: 0C3DF2B4 101655F9 0677E9EB 5AD87278 ED9B0082
t = 67: 78DD4D2B 0C3DF2B4 4405957E 0677E9EB 5AD87278
t = 68: 497093C0 78DD4D2B 030F7CAD 4405957E 0677E9EB
t = 69: 3F2588C2 497093C0 DE37534A 030F7CAD 4405957E
t = 70: C199F8C7 3F2588C2 125C24F0 DE37534A 030F7CAD
t = 71: 39859DE7 C199F8C7 8FC96230 125C24F0 DE37534A
t = 72: EDB42DE4 39859DE7 F0667E31 8FC96230 125C24F0
t = 73: 11793F6F EDB42DE4 CE616779 F0667E31 8FC96230
t = 74: 5EE76897 11793F6F 3B6D0B79 CE616779 F0667E31
t = 75: 63F7DAB7 5EE76897 C45E4FDB 3B6D0B79 CE616779
t = 76: A079B7D9 63F7DAB7 D7B9DA25 C45E4FDB 3B6D0B79
t = 77: 860D21CC A079B7D9 D8FDF6AD D7B9DA25 C45E4FDB

t = 78: 5738D5E1 860D21CC 681E6DF6 D8FDF6AD D7B9DA25

t = 79: 42541B35 5738D5E1 21834873 681E6DF6 D8FDF6AD

Après le traitement du bloc, les valeurs des $H_i$ et le message haché obtenu sont représentés dans la ***Figure III.17***.

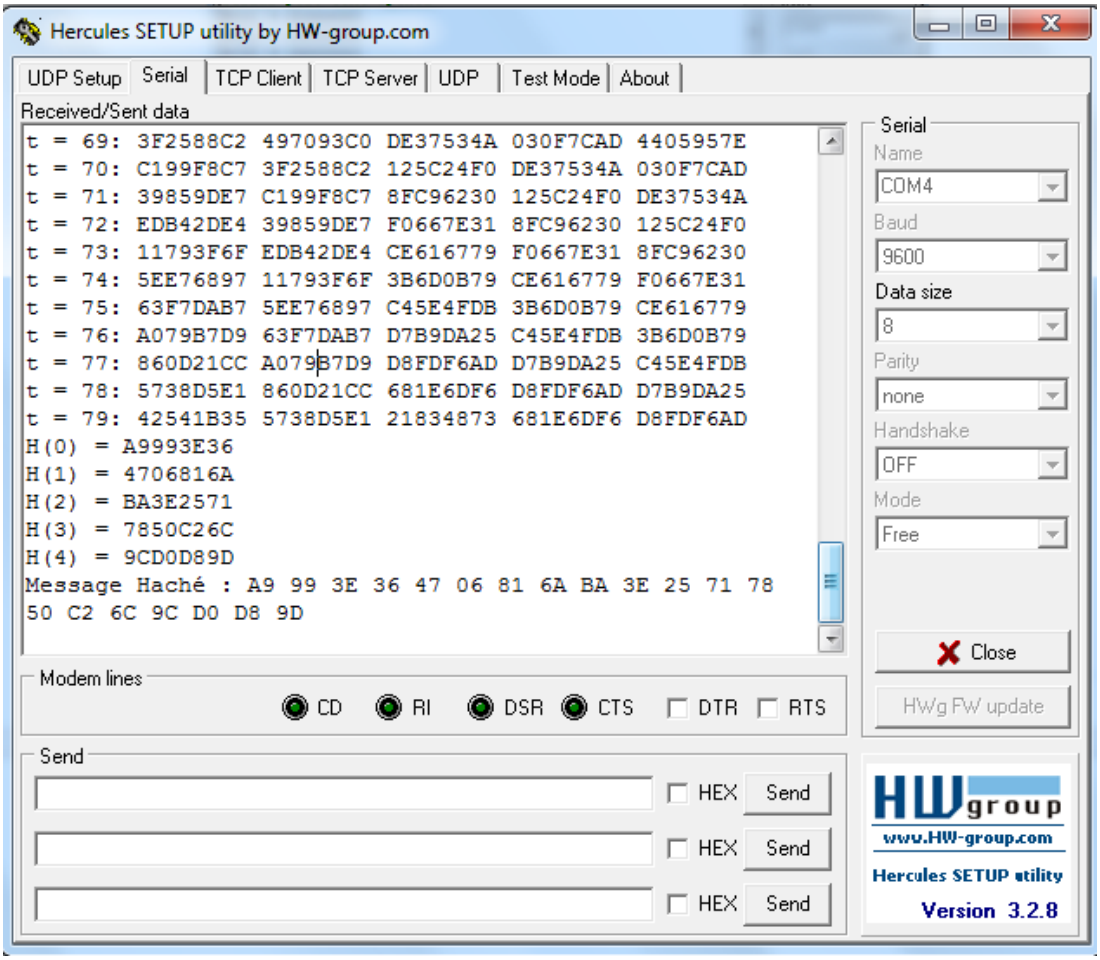



***Figure III.17 :*** Résultat du 1er test pratique sur la carte.

### III.5.2. Résultats du 2ème test pratique :

Nous avons effectué un deuxième test sur un message d'origine composé de 56 caractères « abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq ».

**a) Bourrage du message :**

Comme chaque caractère est codé sur 8 bits, la longueur du message est l=448. Après l'ajout d'un "1", 511 "0" sont ajoutés, faisant un total de 960 bits. La représentation en deux mots de la valeur 448 est l'hexadécimal 00000000 000001C0.

Donc, le message bourré final est composé de 1024 bits, c'est-à-dire deux blocs de 512 bits (n=2).

**b) Résumé du message :**

Les deux blocs du message d'entré sont représentés respectivement dans les ***Figures III.18*** et ***III.19***.

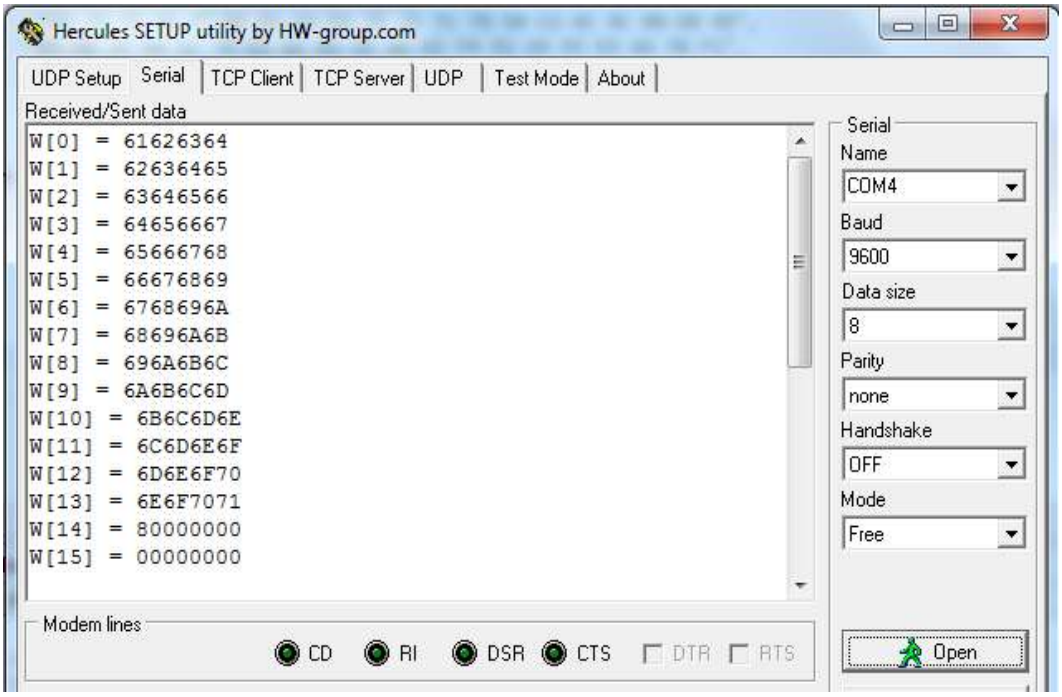



***Figure III.18 :*** 1er bloc du message bourré.

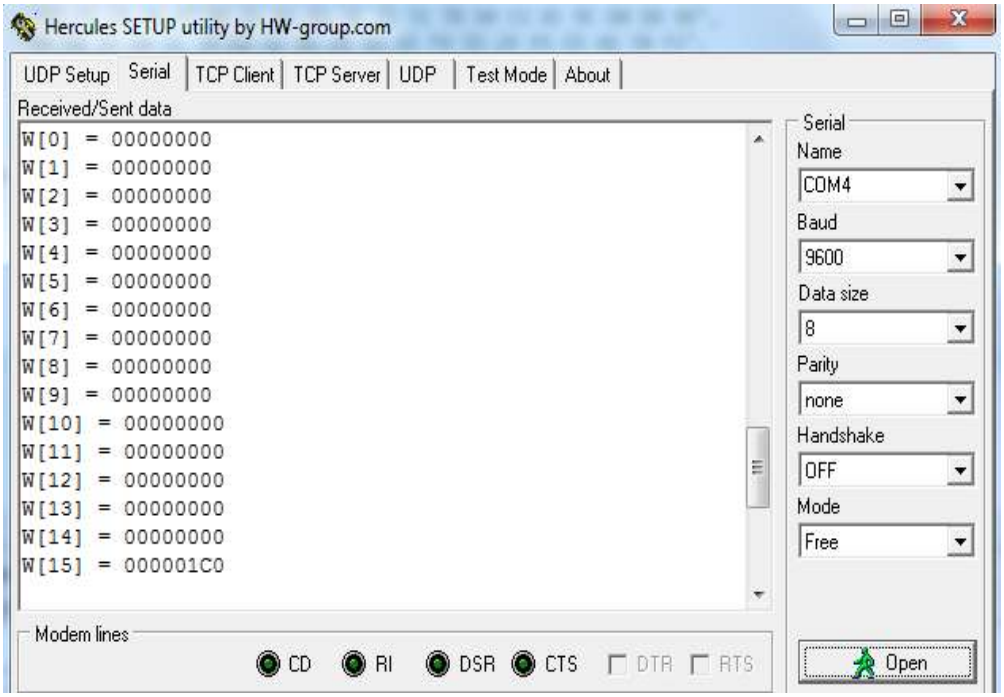



***Figure III.19*** : 2ème bloc du message bourré.

Après le traitement du premier bloc, les valeurs des $H_i$ sont représentés dans la ***Figure III.20***.

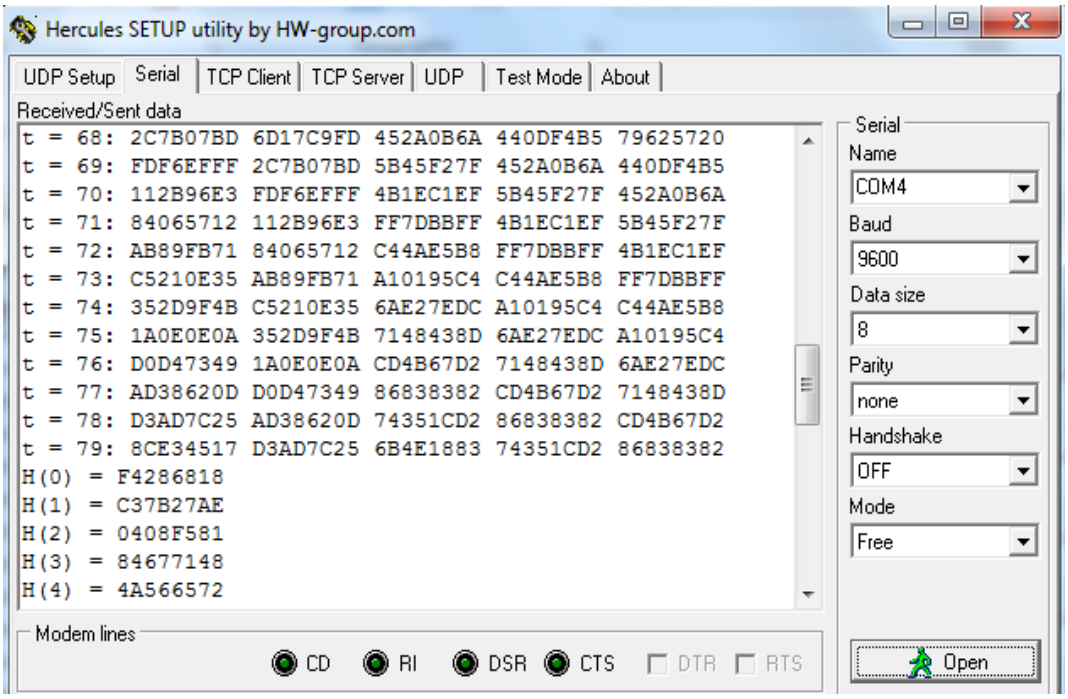



***Figure III.20*** : Résultat du 1er bloc.

Après le traitement du 2ème bloc, les valeurs des $H_i$ et le message haché obtenu sont représentés dans la ***Figure III.21***.

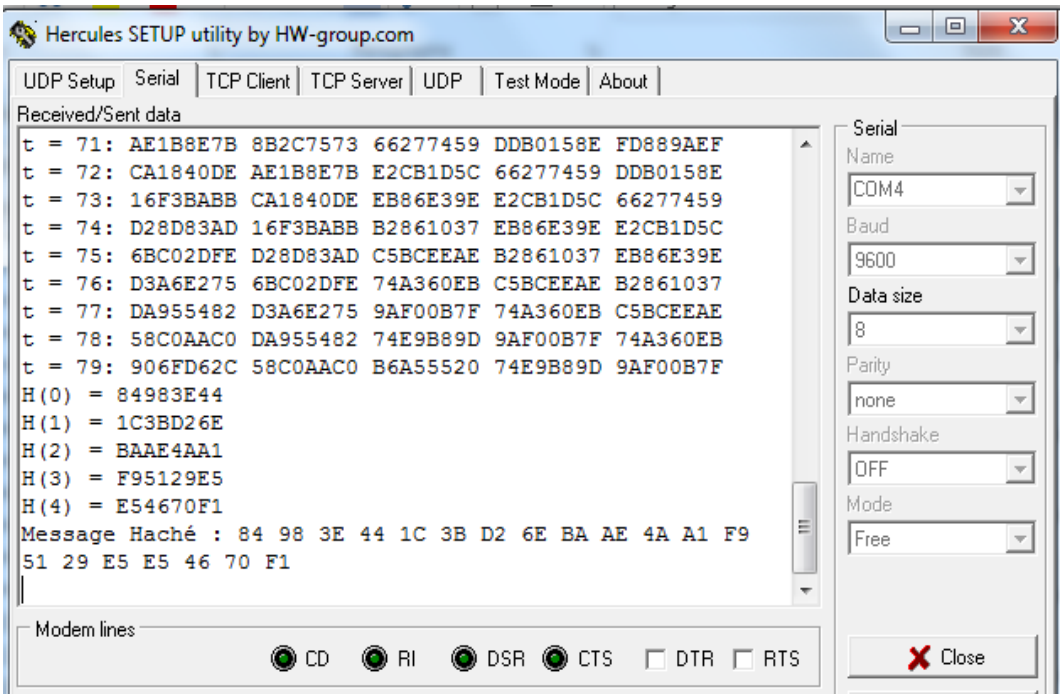



***Figure III.21*** : Résultat du 2ème test pratique sur la carte.

## III.6. Conclusion :

Notre conception pour le hachage cryptographique de l'algorithme SHA1 a été implémentée sur un circuit FPGA avec une carte Digilent Nexys2.

En plus, le circuit implémenté a été testé pratiquement avec 2 exemples. Les résultats des tests sont entièrement validés avec les résultats qui se trouvent sur le document officiel.

# *Chapitre 3*

Conception et implémentation de l'algorithme SHA1

# Conclusion générale

Une fonction de hachage est une méthode permettant de caractériser une information, une donnée. En faisant subir une suite de traitements reproductibles à une entrée, elle génère une empreinte servant à identifier la donnée initiale. De telles fonctions datent de la fin des années 1980 mais l'idée est plus ancienne, et a né à partir de l'apparition des codes correcteurs d'erreurs.

Une fonction de hachage prend donc en entrée un message de taille quelconque, applique une série de transformations et réduit ces données. On obtient à la sortie une chaîne de caractères, le condensé ou le message haché, qui résume en quelque sorte le fichier. Cette sortie a une taille fixe qui varie selon les algorithmes (160 bits pour SHA-1).

Dans ce mémoire, nous avons présenté une conception logicielle de l'algorithme de hachage cryptographique SHA-1. L'utilisation des systèmes reconfigurables est une solution qui permet la validation de façon matérielle de nouveaux principes de sécurité.

Notre conception a été validée, les différents modules sont entièrement fonctionnels. Le circuit développé est implémenté par les outils logiciels de Xilinx sur un circuit FPGA de type Spartan 3E 500. Les résultats des tests pratiques ont été validés avec deux exemples du document officiel.

# Dédicace

*Je dédie ce modeste travail à*

*Mes très chers parents qui veillent sur moi pour me voir réussir, et qui n'ont jamais cessé de m'encourager, de me conseiller…ils m'ont tous donné.*

*A ma Grand-mère qui n'a cessé de prier pour moi tout au long de ma carrière scolaire que dieu la protège.*

*A mes très chères sœurs : Imene, Katia, Assia, Linda*

*A mon binôme et amie Zineb Benterki*

*A mes très chers amies : Naima, Houria*

*A tous les collègues de promos 2016/2017*

*SARRAH REZKI*

# Dédicace

*Je dédie ce modeste travail à*

*Mes très chers parents qui veillent sur moi pour me voir réussir, et qui n'ont jamais cessé de m'encourager, de me conseiller…ils m'ont tous donné.*

*A ma Grand-mère qui n'a cessé de prier pour moi tout au long de ma carrière scolaire que dieu la protège.*

*A mes très chers frères et sœurs:*

*Sid Ahmed, Yassine, Nassima, Faiza,*

*A mon binôme et amie Sarrah Rezki*

*A mes très chers amies : Naima, Houria, Ahlem*

*A tous les collègues de promos 2016/2017*

*Zineb Benterki*

## I.1. Introduction :

Dans ce chapitre, nous introduisons le principe de hachage avec ses différentes propriétés (la résistance aux attaques recherchant des collisions, pré-images et secondes pré-images). Ensuite, nous présentons les fonctions de la famille SHA, à savoir, SHA0, SHA1, SHA2 et SHA3. Et nous terminons par les domaines d'applications.

## I.2. Définition de la fonction de hachage : [1]

On nomme fonction de hachage, de l'anglais hash function, une fonction particulière qui, à partir d'une donnée fournie en entrée de taille variable, calcule en sortie un élément de taille fixe, servant à identifier rapidement, bien qu'incomplètement, la donnée initiale. Les fonctions de hachage sont utilisées en informatique et en cryptographie. Une fonction de hachage H de taille de sortie n est un algorithme faisant correspondre à un message M de taille arbitraire un élément H(M) de taille n bits appelé haché. En pratique, n est de l'ordre de plusieurs centaines de bits, typiquement 128, 160, 256 ou 512 bits. Ces fonctions sont très utiles notamment dans le domaine des bases de données. Elles permettent de manipuler de très grandes structures tout en gardant une vitesse acceptable en ce qui concerne la recherche d'éléments. Elles permettent notamment de diminuer la quantité d'information à crypter.

Si l'image de x par la fonction de hachage est appelée l'empreinte de x, on peut par exemple ne crypter que l'empreinte. D'autre part, elles permettent de mettre au point des protocoles de signature électronique et d'authentification des messages ainsi que de vérifier l'intégrité d'un document, au même titre que le bit de parité, qui lui-même est une fonction de hachage particulière. Formellement, une fonction de hachage H : $\{0,1\}^* \rightarrow \{0,1\}^n$ est une application qui transforme une chaîne de taille quelconque en une chaîne de taille fixe n, comme illustre la figure **I.1**.

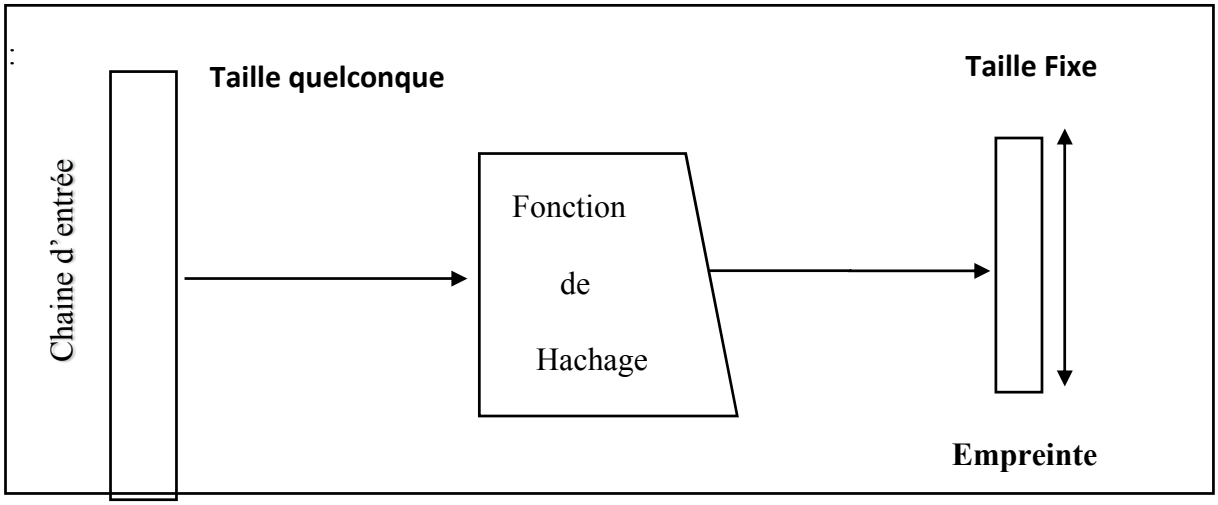



***Figure* I.1** : Principe d'une fonction de hachage

## I.3. Propriétés des fonctions de hachage :

Les fonctions de hachage doivent posséder plusieurs propriétés utiles en cryptographie. Les principales sont la résistance aux attaques recherchant des collisions, des pré-images ou des secondes pré-images.

– **collision** : trouver deux messages distincts $M^1$et $M^2$, tels que $H(M^1) = H(M^2)$.

– **seconde pré-image** : étant donné un message $M^1$ choisi aléatoirement, trouver un message distinct $M^2$ tel que $H(M^1) = H(M^2)$.

– **pré-image** : étant donné un haché H1 choisi aléatoirement, trouver un message M tel que $H(M^1) = H^1$.

Il doit être impossible pour un attaquant de trouver une collision, une pré-image ou une seconde pré-image. On remarque que puisque la taille d'entrée de la fonction de hachage est arbitrairement grande, des collisions existent nécessairement.

Ces notions de sécurité peuvent aussi être appliquées aux fonctions de compression, la seule différence étant la taille fixe des messages et la possibilité pour l'attaquant de choisir ou non la variable de chaînage d'entrée. Le nombre d'opérations pour une attaque générique reste identique aux cas respectifs des fonctions de hachage.

– **collision** : étant donné un état interne $H_{i-1}$ choisi aléatoirement, trouver deux blocs de message distincts $M_i^1$ et $M_i^2$tels que $h(H_{i-1}, M_i^1) = h(H_{i-1}, M_i^2)$.

– **collision libre**: trouver deux blocs de message distincts $M_i^1$ et $M_i^2$ et un état interne $H_{i-1}$ tels que $h(H_{i-1}, M_i^1) = h(H_{i-1}, M_i)$.

– **seconde pré-image** : étant donné un état interne $H_{i-1}$ et un bloc de message $M_i^1$ choisis aléatoirement, trouver un bloc de message distinct $M_i^2$ tel que $h(H_{i-1}, M_i^1) = h(H_{i-1}, M_i^2)$.

– **seconde pré-image libre**: étant donné un bloc de message $M_i^1$ choisi aléatoirement, trouver un bloc de message distinct $M_i^2$ et un état interne $H_{i-1}$ tels que

$H(H_{i-1}, M_i^1) = h(H_{i-1}, M_i^2)$.

– **pré-image** : étant donné deux états internes $H_{i-1}$ et $H_i$ choisis aléatoirement, trouver un bloc de message $M_i^1$ tel que $H_i = h(H_{i-1}, M_i^1)$.

– **Pré-image libre** : étant donné un bloc de message $M_i^1$ choisi aléatoirement, trouver un bloc de message distinct $M_i^2$ et un état interne $H_{i-1}$ tels que $h(H_{i-1}, M_i^1) = h(H_{i-1}, M_i^2)$.

## I.4. Présentation des fonctions de la famille SHA :

### *I.4.1. Généralités sur SHA (Secure Hash Algorithm) :*

SHA est un algorithme mathématique. Il existe plusieurs déclinaisons de SHA: SHA-0, qui est obsolète et ne s'emploie plus, SHA-1, la version encore utilisée pour la majorité des certificats SSL, SHA-2, une version améliorée de l'algorithme, et SHA-3, paru en 2012.

L'algorithme SHA-2 comporte quatre versions, SHA-224, SHA-256, SHA-384 et SHA-512. L'algorithme SHA-256 est celui que les émetteurs de certificats et les navigateurs favorisent, et les termes SHA-2 et SHA-256 se confondent dans l'usage courant.

### *I.4.2 SHA-0 : [2]*

C'est une fonction de hachage cryptographique, conçue par la NSA sur le modèle de la fonction MD4 de Ron Rivest et publiée en 1993 par le NIST (standard FIPS-180). Elle fournit une empreinte de 160 bits. Elle a été rapidement mise de côté pour des raisons de sécurité insuffisante (non explicitées à l'époque), et modifiée pour donner la fonction SHA-1, publiée par le NIST en 1995 (standard FIPS-180-1).

#### *I.4.2.1 Sécurité de la fonction SHA-0 : [2]*

Les faiblesses dans la conception de SHA-0 ont été confirmées en 1998 par la découverte par Antoine Joux et Florent Chabaud d'une attaque par collision contre SHA-0, avec une complexité égale à $2^{61}$ appels à la fonction de compression contre $2^{80}$ (l'attaque générique des anniversaires).

Certes, il n'existe pas actuellement une attaque connue pour la recherche de pré-image ou de seconde pré-image, mais l'utilisation de SHA-0 est tout à fait déconseillée. Elle est pourtant restée un domaine de recherche pour la communauté académique, en tant que prédécesseur de SHA-1.

### *I.4.3 SHA-1 :*

#### *I.4.3.1 Bourrage du message : [4]*

SHA-1 est utilisé pour calculer un résumé de message pour un message ou fichier de données qui est fourni en entrée. Le message ou fichier de données devrait être considéré comme une chaîne binaire. La longueur du message est le nombre de bits dans le message.

Si le nombre de bits dans un message est un multiple de 8, pour le rendre compact, on peut représenter le message en hexadécimal. L'objet du bourrage de message est de rendre la longueur totale d'un message bourré multiple de 512. SHA-1 traite en séquence les blocs de 512 bits lors du calcul du résumé du Message.

Ce qui suit spécifie comment devra être effectué ce bourrage. En résumé, un "1" suivi par des "0" suivis par un entier de 64 bits sont ajoutés à la fin du message pour produire un message bourré de longueur 512 * n. L'entier de 64 bits est la longueur du message d'origine. Le message bourré est alors traité par SHA-1 comme des blocs de 512 bits.

Supposons qu'un message a une longueur $l < 2^{64}$. Avant qu'il soit entré dans SHA-1, le message est bourré à droite comme suit :

**a.** un "1" est ajouté.

**b.** Des "0" sont ajoutés. Le nombre de "0" va dépendre de la longueur d'origine du message. Les derniers 64 bits du dernier bloc de 512 bits sont réservés pour la longueur l du message d'origine.

**c.** Obtenir la représentation de l en deux mots, le nombre de bits dans le message d'origine. Si $l < 2^{32}$ le premier mot est alors tout de zéros. Ajouter ces deux mots au message bourré.

Le message bourré va contenir 16*n mots pour toute n > 0. Le message bourré est considéré comme une séquence des n blocs M(1), M(2),……M(n) des premiers caractères (ou bits) du message.

### *I.4.3.2. Fonctions et constantes utilisées :*

Une séquence des fonctions logiques f(0), f(1),..., f(79) est utilisée dans SHA-1. Chaque f(t), $0 \leq t \leq 79$, opère sur trois mots B, C, D de 32 bits et produit un mot de 32 bits en résultat. f(t;B,C,D) est défini comme suit :

pour les mots B, C, D,

f(t ; B, C, D) = (B ET C) OU ((NON B) ET D) $(0 \leq t \leq 19)$

f(t ; B, C, D) = B OUX C OUX D $(20 \leq t \leq 39)$

f(t ; B ,C ,D) = (B ET C) OU (B ET D) OU (C ET D) $(40 \leq t \leq 59)$

f(t ; B , C, D) = B OUX C OUX D $(60 \leq t \leq 79)$.

Une séquence de mots constants K(0), K(1), ... , K(79) est utilisée dans SHA-1. En hexadécimal, ils donnent :

K(t) = 5A827999 $(0 \leq t \leq 19)$

K(t) = 6ED9EBA1 $(20 \leq t \leq 39)$

K(t) = 8F1BBCDC $(40 \leq t \leq 59)$

K(t) = CA62C1D6 $(60 \leq t \leq 79)$.

### *I.4.3.3 Calcul du résumé de message :*

Le résumé de message est calculé en utilisant le message bourré comme décrit précédemment. Le calcul est décrit avec l'utilisation de deux mémoires tampon, chacune consistant en cinq mots de 32 bits, et une séquence de quatre-vingt mots de 32 bits. Les mots de la première mémoire tampon de cinq mots sont étiquetés A, B, C, D, E. Les mots de la seconde sont étiquetés H0, H1, H2, H3, H4. Les mots de la séquence de 80 mots sont étiquettes W(0), W(1),..., W(79). Une mémoire tampon TEMP d'un seul mot est aussi employée.

Pour générer le résumé de message, les blocs M(1), M(2),..., M(n) de 16 bits sont traités dans cet ordre. Le traitement de chaque M(i) implique 80 étapes.

Avant de traiter un bloc, les H sont initialisés comme suit, en hexadécimal :

H0 = 67452301

H1 = EFCDAB89

H2 = 98BADCFE

H3 = 10325476

H4 = C3D2E1F0

Ensuite M(1), M(2), ..., M(n) sont traités. Pour traiter M(i), on procède comme suit :

**a.** Diviser M(i) en 16 mots W(0), W(1), ..., W(15), ou W(0) est le mot le plus à gauche.

**b.** Pour $16 \leq t \leq 79$ soit W(t) = ( W(t-3) OUX W(t-8) OUX W(t-14) OUX W(t-16) ) <<<1.

**c.** Soit A = H0, B = H1, C = H2, D = H3, E = H4.

**d.** Pour $0 \leq t \leq 79$ faire TEMP = (A) <<<5 + f(t;B, C, D) + E + W(t) + K(t) ; E = D ; D = C; C = (B)<<<30 ; B = A ; A = TEMP;

**e.** Soit H0 = H0 + A, H1 = H1 + B, H2 = H2 + C, H3 = H3 + D, H4 = H4 + E.

Après le traitement de M(n), le résumé de message est la chaîne de 160 bits représentée par les 5 mots H0 H1 H2 H3 H4. Le détail d'un tour SHA-1 est fourni dans la figure **I.2**.

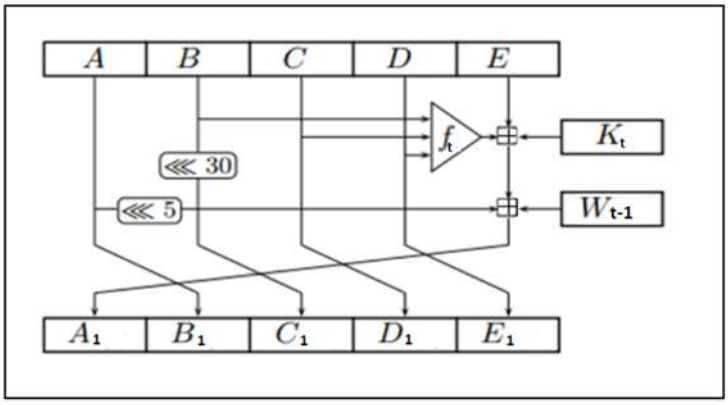



**Figure I.2 :** Fonction de mise à jour des registres

### *I.4.4 Sécurité de SHA-1 : [2]*

La taille d'empreinte dans SHA-1 est de 160 bits. La recherche de collisions par force brute (attaque de Yuval) a donc une complexité en 280 calculs d'empreintes. Récemment, une technique similaire à celle utilisée contre MD5 a permis d'obtenir des collisions en seulement $O(2^{63})$ calculs d'empreinte. On considère donc que SHA-1 est cassé, même si l'attaque reste difficile. De part sa popularité, cette fonction de hachage est encore largement utilisée.

### *I.4.5 SHA-2 :[2]*

En 2002 le NIST a normalisé une nouvelle famille de fonctions de hachage, il s'agit de SHA-2 [NIS12]. Quatre versions de cet algorithme permettent de calculer des hachés de taille différente SHA-224, SHA-256, SHA-384 et SHA-512 bits.

SHA-2 utilise l'algorithme d'extension de domaine de Merkle-Damgård et la construction de Davies-Meyer comme ses prédécesseurs. Généralement, SHA-224 et SHA-256 travaillent sur des blocs de message 512 bits et utilisent des variables de chaînage de 256 bits (8 registres de 32 bits), et SHA-384 et SHA-512 travaillent sur des blocs de message 1024 bits et utilisent des variables de chaînage de 512 bits (8 registres de 64 bits). La structure générale des opérations effectuées est analogue entre ces quatre fonctions.

### *I.4.6 SHA-3 : [2]*

En 2007, le NIST a lancé un concours visant à définir un nouveau standard pour les fonctions de hachage, SHA-3. Le besoin d'un nouveau standard fait suite aux nombreuses attaques essuyées par la fonction SHA1, pour laquelle une collision peut être trouvée en $O(2^{63})$. Le standard actuel (SHA-2) étant dérivé de SHA-1 mais avec des tailles de hachés plus grandes, pourrait donc être victime du même type d'attaque.

## I.5. Domaines d'utilisation des fonctions de hachage cryptographiques : [2]

Les applications des fonctions de hachage cryptographiques apparaissent dans de nombreux domaines d'applications. Chacun de ces domaines nécessite des propriétés spécifiques de sécurité, nous citons dans cette partie des exemples de ces domaines.

### *I.5.1 Intégrité des données :*

La vérification de l'intégrité d'une donnée est parmi les principales utilisations d'une fonction de hachage. C'est le fait de s'assurer que les données qui ont été envoyées dans un canal non sûr sont complètes et n'ont pas été changées. La moindre modification dans les données doit aboutir à l'obtention d'empreintes totalement différentes avec une probabilité très grande.

### *I.5.2 Protection de mots de passe :*

Une autre des applications courantes des fonctions de hachage cryptographiques est la protection de mots de passe. Une grande partie des utilisateurs choisissent comme mots de passe des mots courants ou triviaux, n'assurant donc pas une entropie suffisante. Pour cela un mot de passe ne doit pas être stocké en clair sur une machine, seul le résultat du hachage (le haché) du mot de passe est donc stocké. L'ordinateur compare l'empreinte du mot de passe d'origine avec l'empreinte du mot de passe demandé pour identifier un utilisateur.

### *I.5.3 Signature électronique :*

Les schémas de signature électronique sont sans aucun doute l'application la plus importante des fonctions de hachage cryptographiques.

La signature électronique est un mécanisme analogue de la signature manuscrite permettant de garantir l'intégrité d'un document électronique et prouvant au lecteur du document l'identité de son auteur (authentification).

Lorsqu'un utilisateur veut envoyer un message signé elle suit la procédure suivante :

Il calcule le haché du message, le chiffre avec sa clé privée, et le transmet avec le message. Le destinataire décrypte le condensat avec sa clé publique, calcule le haché du message reçu, et les compare. Si les deux hachés sont identiques, il est assuré de l'identité de l'émetteur et l'intégrité du message.

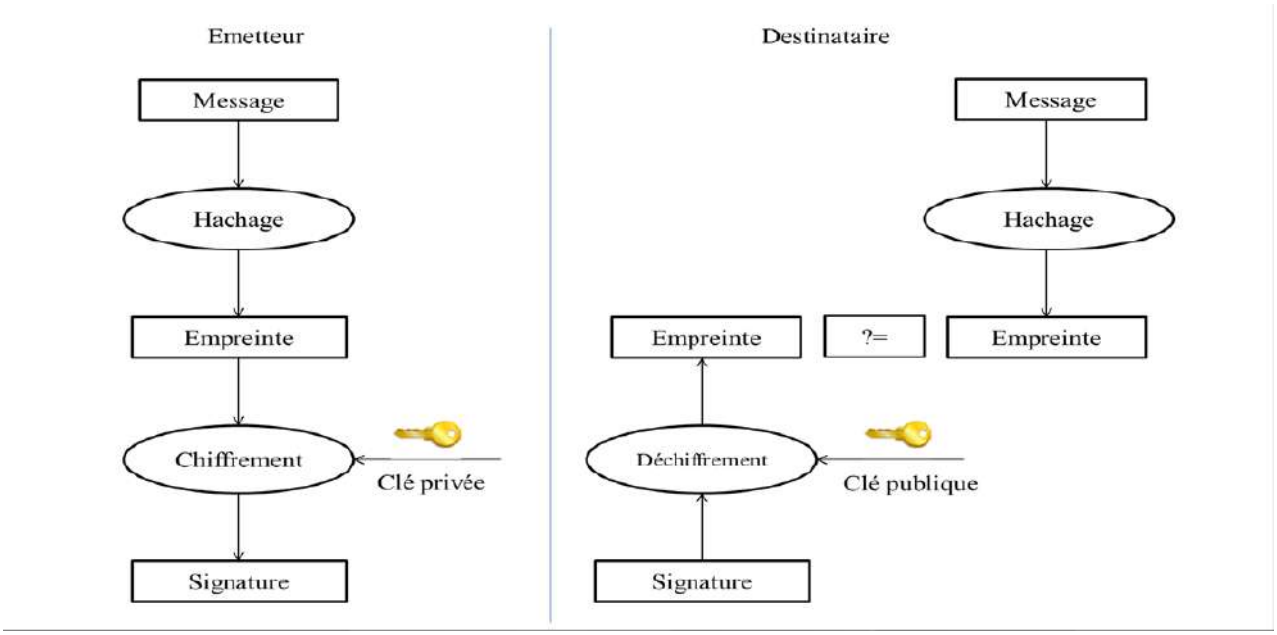



**Figure I.3:** Principe de signatures électroniques.

### *I.5.4 Dérivation de clé :*

Dans le cadre de la cryptographie symétrique, pour assurer une communication entre deux entités, plusieurs clés secrètes sont souvent utilisées. Une fonction de hachage peut générer plusieurs clefs différentes à partir d'une seule information secrète, appelée clef maître. L'utilisation des fonctions de hachage pour cette fonctionnalité a pour objectif d'empêcher un adversaire ayant obtenu une des clés dérivées, de déduire des informations sur la clé maître ou les autres clés dérivées.

### *I.5.5 Génération de nombres pseudo-aléatoires :*

Un générateur pseudo-aléatoire de nombres est un algorithme qui produit des suites de bits présentant certaines propriétés de l'aléatoire. On peut construire un générateur pseudo-aléatoire à partir d'une fonction de hachage.

Par exemple, si H est une fonction de hachage, initialisée avec une graine c, on peut construire une suite pseudo-aléatoire à partir de la suite récurrente xi = H (xi-1 || 0), où x0=c.

Les générateurs pseudo-aléatoires sont nécessaires pour diverses applications cryptographiques, on les retrouve surtout pour la génération des clés pour le chiffrement asymétrique, la confidentialité des échanges sur les réseaux sans fil ou encore dans les signatures électroniques.

## I.6. Conclusion :

Dans ce chapitre, nous avons défini l'algorithme de hachage cryptographique. D'autre part, ses propriétés : la résistance aux attaques recherchant des collisions, les pré-images et les secondes pré-images. Enfin, une description de la famille SHA et en particulier l'algorithme SHA-1.

# Introduction générale

Dans ce mémoire, nous nous intéressons aux fonctions de hachage cryptographiques particulièrement utiles en cryptographie. Elles font correspondre à un message de taille arbitraire, une sortie appelée haché, de taille fixée, 160 bits en général actuellement. Pour des raisons de sécurité on tend à augmenter la taille de l'empreinte.

L'idée est que toute modification, du message d'entrée, doit induire des modifications importantes et imprévisibles du haché en sortie. Par exemple, l'une des propriétés recherchées est qu'il doit être très difficile pour un attaquant de trouver une collision, c'est-à-dire deux messages distincts ayant un haché identique.

Le présent projet consiste à la conception et l'implémentation sur un circuit FPGA de l'algorithme de hachage cryptographique SHA-1.

Le mémoire est organisé comme suit :

Le premier chapitre fournit un bref aperçu sur les fonctions de hachage cryptographiques de la famille SHA, et une description détaillée des différentes étapes de l'algorithme SHA-1. Le deuxième chapitre présente la conception des systèmes numériques embarqués, et en particulier l'environnement de la description de l'architecture matérielle/logicielle qui cible les circuits FPGAs. Dans le dernier chapitre, nous utiliserons l'outil de développement XPS de Xilinx pour créer l'architecture matérielle à base du Microblaze, ensuite nous avons utilisé l'outil SDK pour concevoir le programme de l'algorithme de hachage SHA-1 qui sera exécuté sur le système embarqué à base du Microblaze. Enfin, nous présentons les résultats des tests pratiques du circuit développé sur la carte de type Diglilent Nexys 2 Spartan 3E.

# Sommaire

Résumé……..…….I

Sommaire ……..……..……..……ii

Liste des figures……..……...……iv

liste des tableaux……...…vi

liste des acronymes…….…..vii

Introduction général……....1

## *Chapitre 1 : Algorithmes de hachage cryptographique*

I.1. Introduction……..……...……..2

I.2. Définition de la fonction de hachage ……..……...……..2

I.3. Propriétés des fonctions de hachage ……..……....……3

I.4. Présentation des fonctions de la famille SHA ...……..…….4

*I.4.1. Généralités sur SHA (Secure Hash Algorithm) ....……..4*

*I.4.2 SHA-0 ….……..4*

*I.4.2.1 Sécurité de la fonction SHA-0 :……...……..4*

*I.4.3 SHA-1 :……..4*

*I.4.3.1 Bourrage du message ……...……4*

*I.4.3.2. Fonctions et constantes utilisées ……..5*

*I.4.3.3 Calcul du résumé de message ……..6*

*I.4.4 Sécurité de SHA-1 ……...7*

*I.4.5 SHA-2 ……....…….7*

*I.4.6 SHA-3 ……...……...7*

I.5. Domaines d'utilisation des fonctions de hachage cryptographiques ……..…….7

*I.5.1 Intégrité des données ……..7*

*I.5.2 Protection de mots de passe ……..8*

*I.5.3 Signature électronique ……...8*

*I.5.4 Dérivation de clé ……..9*

*I.5.5 Génération de nombres pseudo-aléatoires ……..9*

I.6. Conclusion ……9

## *Chapitre2 : Conception des systèmes numériques embarqués*


II.1 Introduction ..... 10
II.2 Les circuits logiques programmables [5] ..... 10
II.3 Les systèmes numériques embarqués [5] ..... 12
II.3.1 Codesign : Conception conjointe logicielle/matérielle ..... 12
II.3.2 Flot de la conception conjointe logicielle/matérielle ..... 13
II.3.3 Les composants virtuels (IPs) ..... 13
II.4 Les systèmes sur puce (SoC ) [5] ..... 14
II.4.1 Les contraintes liées aux systèmes embarqués ..... 14
II.4.2 Les SoC et les SoPC ..... 15
II.5 Communications sur puce [5] ..... 16
II.6 Plateformes multi-composants [5] ..... 18
II.6.1 Les processeurs embarqués ..... 19
II.6.2 Les Outils de développement de Xilinx ..... 21
II.7 Conclusion ..... 22


## *Chapitre 3 : Conception et implémentation de l'algorithme SHA1*


III.1. Introduction ..... 23
III.2. Présentation de la carte FPGA ..... 23
III.3. Outil Software utilisé ..... 23
III.4. Réalisation du projet sous EDK : ..... 24
III.4.1. Utilisation du Base System Builder de XPS ..... 24
III.4.2. Implémentation du matérie ..... 29
III.4.3. Utilisation de l'outil SDK ..... 30
III.5. Résultats des tests pratiques sur la carte FPGA ..... 31
III.5.1. Résultats du 1er test pratique ..... 31
III.5.2. Résultats du 2ème test pratique ..... 35
III.6. Conclusion ..... 38
Conclusion générale ..... 39
Bibliographie ..... 40

# *Liste des tables*

***Table III.1 :*** Rapport des résultats concernant l'occupation de surface……..……....30

# Liste des figures

## Chapitre I

**Figure I.1** : Principe d'une fonction de hachage……..2

**Figure I.2 :** Fonction de mise à jour des registres……..6

**Figure I.3:** Principe de signatures électroniques ……..8

## Chapitre II

**Figure II.1.** Les différents circuits programmables……..10

**Figure II.2.** Architecture conceptuelle d'un FPGA (Architecture, Interconnexions et CLB)..11

**Figure II.3.** Flot de conception standard……...13

**Figure II.4**. Flot de conception Xilinx……....16

**Figure II.5.** Le bus PLB dans un système à base de MicroBlaze……..17

**Figure II.6.** Détails d'une plateforme multi-composants……...18

**Figure II.7.** Schéma-bloc fonctionnel du noyau MicroBlaze……20

## Chapitre III

**Figure III.1** : carte Digilent Nexys 2……23

**Figure III.2** : Modules du système Microblaze utilisés……....24

**Figure III.3** : Démarrage application XPS…… …24

**Figure III.4** : création nouveau projet XPS……25

**Figure III.5** : choix carte FPGA……..25

**Figure III.6** : Sélection solution mono processeur……...26

**Figure III.7** : sélection taille mémoire et fréquence d'horloge microblaze……26

**Figure III.8** : sélection modules utiles……27

**Figure III.9 :** Le navigateur des projets de Xilinx Platform Studio……27

**Figure III.10** : Capture d'écran des ports utilisés…… 28

**Figure III.11** : Capture d'écran des adresses réservées aux différents modules……...28

**Figure III.12 :** fichier de spécification de matériel MHS……28

**Figure III.13 :** fichier de contraintes UCF……29

***Figure III.14 :*** *Schéma RTL du système à base du MicroBlaze implémenté……..29*

***Figure III.15 :*** *Le navigateur des projets de Software Development Kit……..31*

***Figure III.16*** *: Capture écran du message bourré……..32*

***Figure III.17*** *: Résultat du 1$^{er}$ test pratique sur la carte……35*

***Figure III.18*** *: 1$^{er}$ bloc du message bourré……36*

***Figure III.19*** *: 2$^{ème}$ bloc du message bourré……36*

***Figure III.20*** *: Résultat du traitement du 1$^{er}$ bloc……37*

***Figure III.21*** *: Résultat du 2$^{ème}$ test pratique sur la carte……37*

**ملخص :**

في هذا المشروع، نستخدم أداة التطوير XILINX XPS لخلق بنية الأجهزة على أساس MicroBlaze. ثم نستخدم أداة SDK لتصميم برنامج تشغيل خوارزمية التجزئة SHA-1 على المنصة الأساسية MicroBlaze. وأخيرا، نقوم بإجراء الاختبارات العملية على البطاقة Diglilent Nexys 2 Spartan 3E.

**كلمات مفتاحية :** تجزئة التشفير، نظام مضمن ، FPGA ، SHA-1

## *Résumé :*

Dans ce projet, nous utilisons l'outil de développement XPS de Xilinx pour créer une architecture matérielle à base du Microblaze. Ensuite nous utilisons l'outil SDK pour concevoir un programme exécutant l'algorithme de hachage SHA-1 sur la plateforme à base du Microblaze. Enfin, nous réalisons des tests pratiques sur une carte de type Diglilent Nexys 2 Spartan 3E.

**Mot clé** : FPGA, hachage cryptographique, SHA-1, système embarqué

## *Abstract :*

In this project, we use Xilinx's XPS development tool to create a Microblaze based hardware architecture. Then we use the SDK tool to design a program executing the SHA-1 hashing algorithm on the Microblaze-based platform. Finally, we carry out practical tests on a Digilent Nexys 2 Spartan 3E card.

**Key words:** cryptographic hach, embedded system, FPGA, SHA-1