



République Algérienne Démocratique et Populaire
الجمهورية الجزائرية الديمقراطية الشعبية
Ministère de l'Enseignement Supérieur et de la Recherche
Scientifique
وزارة التعليم العالي و البحث العلمي



Mémoire de fin d'études

Pour l'obtention du diplôme d'Ingénieur d'État en
Télécommunications

Option : Systèmes de Télécommunications et Réseau

Implémentation de la fonction dehachage SHA3 sur FPGA

Réalisé par :
Mme MECHITOUA Zina

Présenté et soutenu publiquement le :19/09/2024

Composition du Jury :

M. BOUDIAR Toufik	MCA	ENSTA	Président
Mme BOUCHAMA Samira	MCB	ENSTA	Promotrice
Mme BELHACHAT Faiza	MCA	ENSTA	Co-promotrice
M. GUELLAL Amar	MCB	ENSTA	Examineur
M. ABBADI Belkacem	MCB	ENSTA	Examineur

Promotion : 2023/2024

Remerciements

Je tiens à exprimer ma plus profonde gratitude à toutes les personnes qui ont contribué, de près ou de loin, à la réalisation de ce travail.

Tout d'abord, je remercie chaleureusement Mme BOUCHAMA SAMIRA pour ses précieux conseils, son encadrement et sa disponibilité tout au long de ce projet. Son expertise et ses suggestions m'ont été d'une grande aide pour mener à bien cette recherche.

Je tiens également à remercier Mme BELHACHAT FAIZA pour son soutien et ses conseils avisés. Son aide a été inestimable dans l'orientation et la finalisation de ce travail.

Aussi, j'exprime mes sincères remerciements aux membres de jury qui ont accepté d'examiner ce travail.

Un grand merci à mon mari et mes chers parents qui m'ont été d'un grand soutien moral, et mes sœurs pour leur soutien moral et leur encouragement constant. Leur présence et leur écoute m'ont permis de surmonter les moments de doute.

Enfin, je tiens à exprimer ma reconnaissance à toutes les personnes qui, par leur contribution, ont permis la réalisation de ce projet, même si leurs noms ne sont pas mentionnés ici.

Sommaire

Liste des Figures	1
Liste des Tableaux.....	1
Liste des abréviations.....	2
Introduction générale	3
Chapitre I : Cryptographie et Fonction de Hachage SHA3.....	5
I.1. Introduction	6
I.2. Cryptographie	6
I.2.1. Chiffrement symétrique	7
I.2.1.1. Définition.....	7
I.2.1.2. Principe	7
I.2.2. Chiffrement asymétrique	7
I.2.2.1. Définition	7
I.2.2.2. Principe.....	8
I.3. Les fonctions de hachage cryptographiques	8
I.3.1. Définition	8
I.3.2. Propriétés.....	8
I.3.2.1. Les attaques par pré-image	8
I.3.2.2. Les attaques par collision	9
I.3.2.3. Les attaques par deuxième pré-image	9
I.4. Construction des fonctions de hachage cryptographiques.....	10
I.4.1. La construction de Merkle-Damgard	10
I.4.1.1. Définition.....	10
I.4.1.2. Principe	10
I.4.2. La construction de l'éponge (Keccak)	11
I.4.2.1. Absorption de l'information	11
I.4.2.2. L'essorage	11
I.5. Principe de l'algorithme SHA-3.....	11
I.5.1. Technique de Padding (rembourrage)	12
I.5.2. Technique de Mapping	12
I.5.3. Fonction de permutation f de Keccak	14
I.5.3.1. Processus Thêta (θ)	14
I.5.3.2. Processus Rho (ρ).....	15
I.5.3.3. Processus Pi (π).....	16
I.5.3.4. Processus Chi (χ)	17
I.5.3.5. Processus Iota (ι)	18
I.5.4. Technique de troncature.....	20

I.6. Conclusion	20
Chapitre II : Généralités sur les circuits FPGA	21
II.1. Introduction.....	22
II.2. Aperçu sur les circuits logiques programmables	22
II.2.1. PAL (Programmable Array Logic)	23
II.2.2. PLA (Programmable Logic Array).....	23
II.2.3. GAL (Generic Array Logic).....	23
II.2.4. EPLD (Erasable Programmable Logic Device).....	23
II.2.5. CPLD (Complex Programmable Logic Device)	23
II.2.6. FPGA (Field-Programmable Gate Array).....	23
II.3. Définition d'un circuit FPGA	24
II.3.1. Structure d'un circuit FPGA	24
II.3.1.1. Les blocs logiques configurables (CLB - Configurable Logic Blocks)	24
II.3.1.2. Les interfaces d'entrée/sortie (I/O - Input/Output)	25
II.3.1.3. Interconnexions programmables	25
II.3.1.4. Unités de Traitement Numérique du Signal (DSP Blocks)	26
II.3.1.5. Blocs de Mémoire (Block RAM)	27
II.4. Le langage de description utilisé	27
II.5. Étapes de conceptions d'une application FPGA	27
II.5.1. Spécification fonctionnelle.....	28
II.5.2. Langage de description matériel (HDL).....	28
II.5.3. Synthèse	28
II.5.4. Placement et routage	28
II.5.5. Vérification de la synchronisation	29
II.5.6. Configuration	29
II.6. Application des FPGA.....	30
II.7. Les différentes familles des circuits FPGA	30
II.8. Conclusion.....	33
Chapitre III: Implémentation de la fonction de hachage SHA3 sur FPGA.....	34
III.1. Introduction	35
III.2. Implémentation de la fonction SHA3 et génération de l'IP	35
III.2.1. Création du projet	36
III.2.2. Implémentation du programme	37
III.2.3. Vérification du programme C	41
III.2.4. Synthèse haut niveau	41
III.2.4.1. Difficultés rencontrées	42
III.2.5. Vérification de l'implémentation RTL.....	43
III.2.6. Génération de l'IP.....	44
III.3. Implémentation sur la carte FPGA	45
III.4. Conclusion	46

Conclusion générale	47
Annexes.....	48
Références et bibliographie	53
ملخص	55
Résumé.....	55
Abstract	55

Liste des Figures

Figure I. 1 Chiffrement symétrique	7
Figure I. 2. Chiffrement asymétrique.....	8
Figure I. 3 Les attaques par pré-image.....	9
Figure I. 4 Les attaques par collision.....	9
Figure I. 5 Les attaques par deuxième pré-image	9
Figure I. 6 Le principe de la construction de Merkle-Damgard.....	10
Figure I. 7 Construction d'éponge.....	11
Figure I. 8 Etat de SHA-3 [6]	13
Figure I. 9 Représentation visuelle de l'application de la fonction Thêta sur un seul bit [6].	15
Figure I. 10 Le principe fondamental du processus Rho	16
Figure I. 11 L'étiquetage des coordonnées [6].....	16
Figure I. 12 Le principe fondamental du processus Pi.....	17
Figure I. 13 Le principe du processus Chi (une seule ligne).....	18
Figure II. 1. Les composants d'un circuit FPGA	24
Figure II. 2 Les trois types de connexion dans un circuit FPGA	26
Figure II. 3 Flot de conception d'une application FPGA	29
Figure II. 4 Carte de développement ZYBO Z7	31
Figure III. 1 Interface de démarrage de Vivado HLS 2018.3	36
Figure III. 2 Sélection de la carte cible dans Vivado HLS.....	36
Figure III. 3 Addition de fichiers sources dans Vivado HLS	37
Figure III. 4 Les étapes fondamentales du processus de fonctionnement de SHA-3.....	37
Figure III. 5 Vérification du programme en utilisant Code::Blocks	40
Figure III. 6 Vérification du programme en utilisant Vivado HLS	41
Figure III. 7 Résultat obtenu après C simulation	41
Figure III. 8 Synthèse du programme	42
Figure III. 9 Programme proposé pour remplacer la commande memcpy	42
Figure III. 10 C/RTL Cosimulation	43
Figure III. 11 Exportation du RTL	44
Figure III. 12 Génération de l'IP	45
Figure III. 13 Création du projet.....	45
Figure III. 14 L'ajout de l'IP créé à la bibliothèque du VIVADO	46

Liste des Tableaux

Tableau I. 1 Les valeurs de w et l pour chaque b.	13
Tableau I. 2 Le décalage effectué lors du processus Rho.....	15
Tableau I. 3 Les constantes d'itération.....	19

Liste des abréviations

FPGA	F ield- P rogrammable G ate A rray
HLS	H ight L evel S ynthesis
IP	I ntellectual P roperty
MD5	M essage D igest (version 5)
NIST	N ational I nstitute of S tandards and T echnology
RTL	R egister T ransfer L evel
SHA0, SHA1, SHA2, SHA3	S ecure H ash A lgorithm (version 0 , 1 , 2 et 3 respectivement)
SDK	S oftware D evelopment K it

Introduction générale

La cryptographie est un domaine essentiel dans les télécommunications et les réseaux, en particulier grâce à des outils comme les fonctions de hachage. Ces dernières jouent un rôle crucial dans la sécurisation des données, la préservation de la confidentialité et l'authentification des informations échangées. Leur utilisation correcte est vitale pour assurer l'intégrité des communications et la fiabilité des systèmes, ce qui en fait des composants indispensables de l'infrastructure numérique contemporaine [1].

Le SHA3 représente la famille des fonctions de hachage cryptographique, représentant un standard de l'institut américain des standards et des technologies (NIST), et qui a apporté des solutions aux attaques subies par les fonctions MD5[2], SHA0 et SHA1[3][4]. Dans ce contexte, se pose la question de proposer une solution matérielle à travers l'implémentation du SHA3 (Secure Hash Algorithm 3) sur FPGA (Field- Programmable Gate Array). Face aux exigences croissantes en matière de sécurité et de performance dans les domaines des télécommunications et des réseaux, il est pertinent d'évaluer les avantages et les défis de cette approche.

Ce mémoire a pour but d'étudier les étapes de conception et de réalisation d'une implémentation efficace du SHA3 sur une carte FPGA à travers l'outil VIVADO HLS. Cette démarche vise à optimiser les performances de hachage tout en minimisant l'utilisation des ressources matérielles, afin de répondre de manière adéquate aux besoins spécifiques des applications télécoms et réseau.

Dans le premier chapitre introductif, nous explorerons les fondements de la cryptologie, en mettant en lumière l'évolution historique de la cryptographie et de la cryptanalyse. Nous introduirons également les concepts de base des fonctions de hachage cryptographique et leur rôle essentiel dans la sécurisation des données numériques. En outre, nous examinerons en détail la fonction de hachage cryptographique SHA-3, et son algorithme sous-jacent Keccak. Nous aborderons ses mécanismes de fonctionnement, ses propriétés de sécurité et ses applications potentielles dans divers contextes.

Le deuxième chapitre se concentre sur les circuits logiques programmables, en mettant particulièrement l'accent sur les FPGA. Nous explorerons les caractéristiques et les avantages des FPGA par rapport aux circuits logiques traditionnels prédéfinis.

Nous discuterons également les outils et les techniques utilisés pour programmer et implémenter des conceptions matérielles sur des FPGA, avec un accent particulier sur l'outil Vivado HLS de Xilinx.

Dans le dernier chapitre, nous passerons de la théorie à la pratique en explorant l'implémentation de la fonction de hachage SHA-3 sur des circuits FPGA. Nous décrirons le processus de traduction de l'algorithme SHA-3 en une conception matérielle adaptée aux FPGA, en utilisant Vivado HLS comme principal outil de développement. Nous examinerons les étapes clés de cette implémentation, depuis la spécification des entrées et sorties du bloc fonctionnel appelé IP (Intellectual Property) jusqu'à la vérification de l'implémentation RTL (Register Transfer Level) pour garantir la conformité aux exigences de performance et de fonctionnalité.

Chapitre I :

Cryptographie et Fonctionde

Hachage SHA3

I.1. Introduction

Depuis des millénaires, l'humanité a cherché à préserver la confidentialité des échanges, que ce soit dans des contextes de guerre, de négociations diplomatiques ou de transactions commerciales. La cryptologie est née de cette nécessité, offrant un domaine fascinant d'étude et d'application des techniques visant à protéger les données confidentielles. Elle englobe deux aspects essentiels : la cryptographie, qui se concentre sur la création de codes secrets et de mécanismes de chiffrement, et la cryptanalyse, qui cherche à briser ces codes pour accéder aux informations cachées. Au fil du temps, la cryptologie a évolué de méthodes rudimentaires à des algorithmes sophistiqués reposant sur des principes mathématiques complexes [1].

Au cœur de la cryptologie se trouvent les fonctions de hachage cryptographique, jouant un rôle crucial dans la sécurisation des informations confidentielles et des communications numériques. Elles servent de base à de nombreux protocoles de sécurité utilisés quotidiennement [1].

SHA-3, développée par l'Institut National des Standards et de la Technologie (NIST), est une fonction de hachage cryptographique basée sur l'algorithme Keccak. Cette fonction utilise une construction appelée éponge, capable de produire une sortie de taille fixe à partir d'une entrée de taille variable. Conçu pour résister à une variété d'attaques telles que les attaques de collision, de préimage et de seconde préimage, SHA-3 offre une sécurité robuste et efficace adaptée à une multitude d'applications [5].

Dans ce chapitre, nous aborderons le concept général de la cryptographie, ses différentes approches, ainsi que les fonctions de hachage cryptographique et leurs propriétés. Nous examinerons également en profondeur le fonctionnement de SHA-3.

I.2. Cryptographie

La cryptographie est la science qui étudie les techniques, principes et les méthodes pour rendre une information incompréhensible. Elle est utilisée pour sécuriser la communication dans un canal peu sûr, donc elle assure la protection de la confidentialité, l'authenticité, l'intégrité des données et la non-répudiation[1].

Pour crypter une information, il existe deux approches qui sont présentées ci-après[1] [6].

I.2.1. Chiffrement symétrique

I.2.1.1. Définition

Le chiffrement symétrique est basé sur l'utilisation de la même clé pour chiffrer l'information échangée par l'émetteur et la déchiffrer par le récepteur. Cette clé doit être partagée entre l'émetteur et le récepteur seulement.

I.2.1.2. Principe

L'émetteur veut envoyer un message au récepteur. Il souhaite chiffrer ce message auquel soit incompréhensible par quelqu'un d'autre que le récepteur. L'émetteur doit donc chiffrer son message avec la clé et son algorithme de chiffrement et il envoie le message au récepteur. Le récepteur quand il aura reçu le message chiffré, il va utiliser la clé de déchiffrement (qui est identique à la clé de chiffrement) et son algorithme de déchiffrement pour décrypter le message reçu. Le principe général du chiffrement symétrique est représenté sur la Figure I.1.

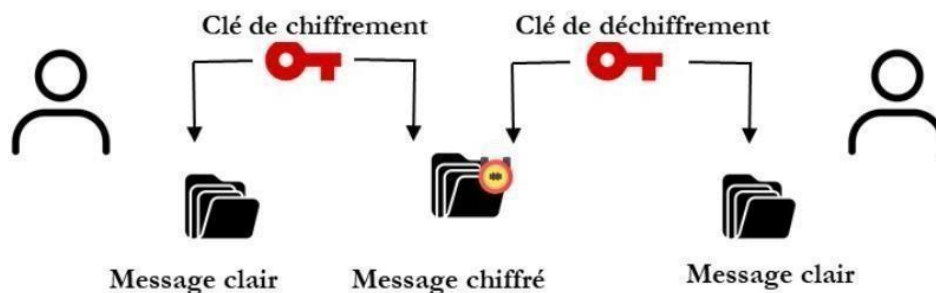


Figure I. 1 Chiffrement symétrique

I.2.2. Chiffrement asymétrique

I.2.2.1. Définition

Le chiffrement asymétrique est basé sur l'utilisation de deux clés différentes ; une pour chiffrer et une autre pour déchiffrer. Chaque participant à la communication a deux clés qui sont la clé publique pour chiffrer et la clé privée pour déchiffrer. La clé publique peut être connue par tout le monde mais la clé privée doit être connue seulement de celui qui la possède.

I.2.2.2. Principe

L'émetteur veut envoyer un message au récepteur. Il chiffre le message en utilisant la clé publique qui est accessible à tous et que seul le récepteur pourra déchiffrer. Le message obtenu est envoyé au récepteur qui fait le déchiffrement en utilisant la clé privée associée à la clé publique comme illustre la Figure I.2.

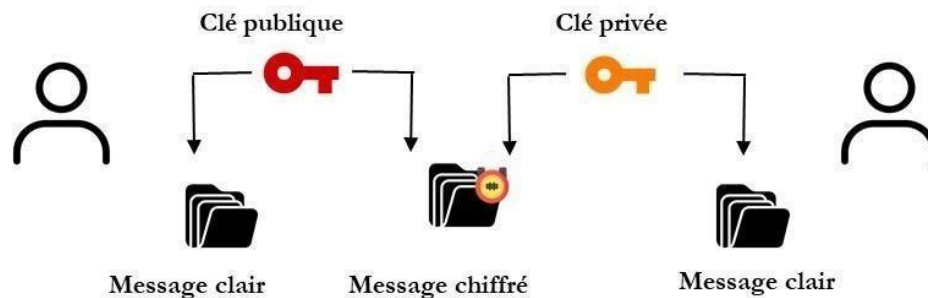


Figure I. 2 . Chiffrement asymétrique

I.3. Les fonctions de hachage cryptographiques

I.3.1. Définition

La fonction de hachage cryptographique est la plus utilisée. Elle est très importante dans le domaine de la sécurité informatique. Son principe est de donner une chaîne binaire de taille fixe qui est l'empreinte ou hash à partir de données (texte, mot, vidéo...etc.) de taille quelconque.

I.3.2. Propriétés

Les propriétés de base d'une fonction de hachage sont les suivantes : le déterminisme, la rapidité du calcul.

La fonction de hachage cryptographique possède quelques propriétés supplémentaires, en effet, elle est résistante à certaines attaques qui sont présentées ci-après [7].

I.3.2.1. Les attaques par pré-image

L'attaque par pré-image (Figure I.3) consiste, pour une valeur d'empreinte spécifiée, à tenter de trouver un message ayant cette empreinte ou haché. Une fonction de hachage est dite résistante aux attaques par pré-image si pour une valeur de hachage donnée h , il est impossible de trouver un message m tel que $h = \text{hachage}(m)$.

*Figure I. 3 Les attaques par pré-image*

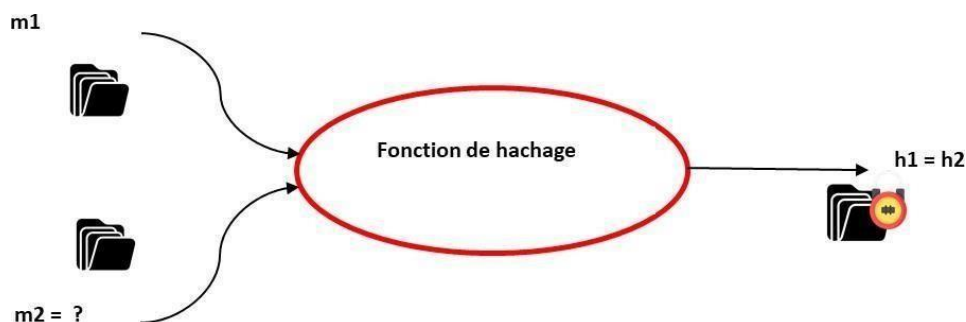
I.3.2.2. Les attaques par collision

L'attaque par collision consiste à trouver deux messages différents, ayant le même haché. Une fonction de hachage est résistante aux collisions s'il est impossible de trouver deux messages m_1 et m_2 tel que le hachage h_1 de m_1 est égal au hachage h_2 de m_2 comme représenté sur la Figure I.4.

*Figure I. 4 Les attaques par collision*

I.3.2.3. Les attaques par deuxième pré-image

La fonction de hachage est dite résistante à la deuxième pré-image (Figure I.5) si pour un message donné m_1 , il est impossible de trouver un message différent m_2 tel que le hachage h_1 de m_1 soit égal au hachage h_2 de m_2 . Cette propriété est parfois appelée faible résistance aux collisions.

*Figure I. 5 Les attaques par deuxième pré-image*

I.4. Construction des fonctions de hachage cryptographiques

Plusieurs fonctions de hachage sont apparues tel que MD5(1991), SHA1(1995), SHA2(2001). Ces dernières sont basées sur le même mécanisme qui est la construction de Merkle-Damgard [3] mais en 2015, la fonction SHA3 a émergée, elle est basée sur un mécanisme complètement différent qui est la construction éponge.

I.4.1. La construction de Merkle-Damgard

I.4.1.1. Définition

La construction de Merkle-Damgard est un algorithme qui permet de résoudre le problème du hachage cryptographique, en acceptant un message de taille quelconque tout en produisant une sortie de taille fixe, en se basant sur l'utilisation d'une fonction de compression.

I.4.1.2. Principe

Le message à hacher M est découpé en blocs de taille fixe, le dernier bloc est plus petit que les autres, il est complété par des bits supplémentaires, c'est le padding.

H_0 est une constante qui est traitée par une fonction de compression f avec le premier bloc pour obtenir H_1 (haché intermédiaire). H_1 et le deuxième bloc sont traités par la fonction de compression pour donner H_2 et ainsi de suite pour tous les blocs afin de produire l'empreinte finale $H(M)$. La figure ci dessous illustre le principe de la construction de Merkle-Damgard.

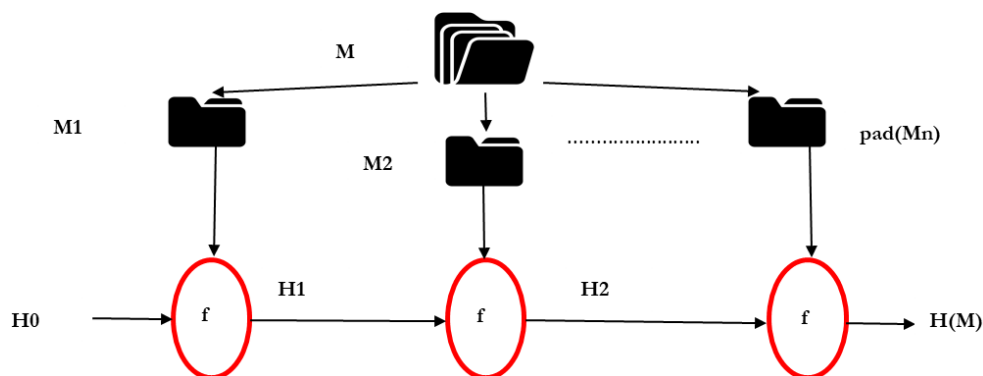


Figure I. 6 Le principe de la construction de Merkle-Damgard

I.4.2. La construction de l'éponge (Keccak)

Cette construction a deux phases : absorption de l'information après l'essorage [8].

I.4.2.1. Absorption de l'information

Deux nombres r et c de bits sont définis, l'entrée est découpée en blocs de taille r bits et un vecteur de taille $(r+c)$ est choisi.

Un XOR entre les r bits initiaux et le premier bloc est fait, le résultat représente une entrée de la fonction f qui est une fonction de permutation pseudo-aléatoire c.à.d. Les bits fournis en entrée sont mélangés entre eux. Cette fonction prend aussi en entrée les c bits de l'état initial et elle produit par la suite deux sorties, une de r bits une de c bits. Le même processus est fait avec tous les blocs.

I.4.2.2. L'essorage

La fonction f est appliquée et son résultat de r bits est ajoutée à une chaîne binaire (qui sera le résultat final) à chaque itération de f jusqu'à où le hach aura au moins la taille désirée.

La figure ci dessous illustre le principe de la construction de l'éponge.

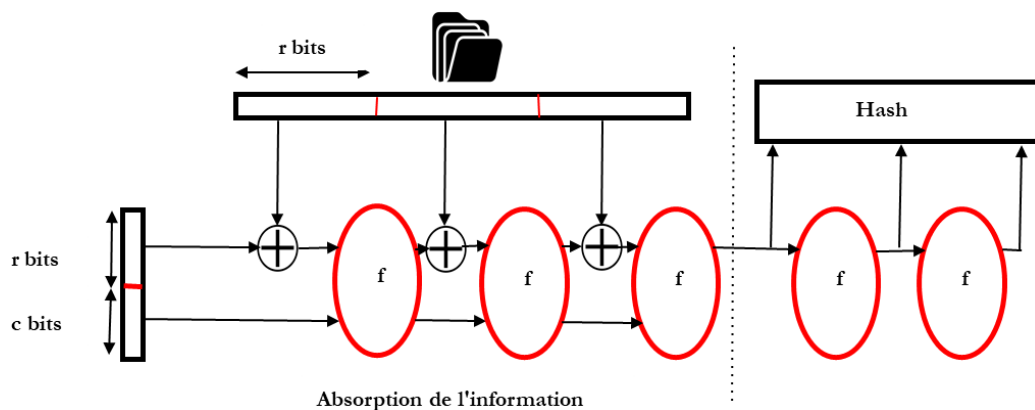


Figure I. 7 Construction d'éponge

I.5. Principe de l'algorithme SHA-3

L'adaptation de la fonction éponge se fait par le choix de ces trois paramètres tels que "b", "c" et "r". "r" correspond à la taille du bloc de message (bitrate). "b" représente la largeur

de l'état. Et "c" représente la capacité qui est la différence entre "b" et "r", soit $c = b - r$.

Pour générer l'haché d'un message à partir de la fonction de hachage cryptographique Keccak, le message doit être prétraité en appliquant les techniques de padding et de mapping. Cette phase de prétraitement prépare une entrée spécifique pour la permutation Keccak en termes de taille et de dimension. Le principe de l'algorithme SHA3 est détaillé dans [9].

I.5.1. Technique de Padding (rembourrage)

Le message à hacher est une donnée de taille arbitraire. Or, Keccak nécessite que cette taille soit fixe. Pour cela, des bits supplémentaires sont ajoutés à la fin du message d'entrée pour atteindre une longueur spécifique requise par l'algorithme de hachage, c'est le padding. Plus précisément, la longueur totale du message doit être un multiple de "r" où "r" dépend de la taille de l'empreinte.

Soit "l" la taille du message à hacher, "x" est le nombre des bits à ajouter qu'est calculé par la relation :

$$x = r - l \text{ mod } r.$$

Après l'ajout des bits de rembourrage, le message résultant est découpé en blocs de taille fixe(r), c'est la segmentation.

Le processus de padding permet que chaque message à hacher, quelle que soit sa taille, soit traité de manière uniforme par l'algorithme de hachage.

I.5.2. Technique de Mapping

Cette technique permet de générer un état (state) qui est utilisé comme entrée de la fonction f en appliquant l'équation :

$$\text{State}(x ; y ; z) = \text{IR}(z + 64 \times (5 \times y + x))$$

$$\text{Avec : IR} = (\text{bloc} \oplus r) \parallel c$$

$$0 \leq x \leq 4, \quad 0 \leq y \leq 4, \quad 0 \leq z \leq 63$$

La Figure I.8 représente un état de SHA3, tel que la valeur de $b = 200$.

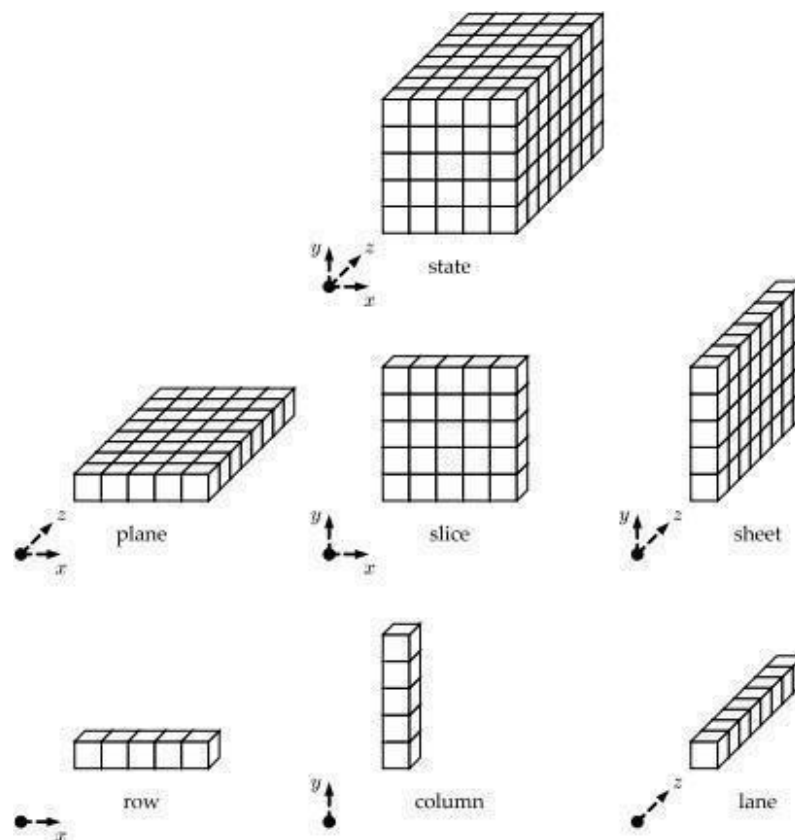


Figure I. 8 Etat de SHA-3 [10].

Cet état est une donnée d'entrée à trois dimensions (matrice tridimensionnelle) de taille $x \times y \times z$ où $|x| + |y| = 5$. Cette matrice est constituée de 25 couloirs (lanes) dont la longueur est déterminée par les paramètres b , w et l illustrés dans le Tableau I.1 :

Tableau I. 1 Les valeurs de w et l pour chaque b .

b	25	50	100	200	400	800	1600
w	1	2	4	8	16	32	64
l	0	1	2	3	4	5	6

Avec : $w = b \setminus 25$, $l = \log_2(b \setminus 25)$

En fonction de b , l'état est composé de $5 \times 5 \times w$ bits.

I.5.3. Fonction de permutation f de Keccak

La permutation f est formée de cinq processus et comporte b bits. Les processus sont les suivants : Thêta (θ), Rho (ρ), Pi (π), Chi (χ) et Iota (ι). Sur la base de ces processus, l'état est mis à jour en répétant la fonction nr fois, où : $n_r = 12 + 2 \times l$.

I.5.3.1. Processus Thêta (θ) :

Ce processus a pour objectif de fournir une diffusion de l'État. L'opération effectuée est une transformation non linéaire qui utilise la fonction XOR pour combiner chaque bit avec les bits adjacents dans les deux colonnes voisines.

Les trois étapes du processus :

Entrée : Un tableau d'état A.

Sortie : Un tableau d'état A' après l'application du processus Thêta.

1. Pour toutes les paires (x, z) telles que $0 \leq x < 5$ et $0 \leq z < w$, C [x, z] est défini comme étant égal à l'opération XOR entre A [x, 0, z], A [x, 1, z], A [x, 2, z], A [x, 3, z] et A [x, 4, z].

$$C [x, z] = A [x, 0, z] \oplus A [x, 1, z] \oplus A [x, 2, z] \oplus A [x, 3, z] \oplus A [x, 4, z] ;$$

2. Pour toutes les paires (x, z) telles que $0 \leq x < 5$ et $0 \leq z < w$, D [x, z] est défini comme étant égal à l'opération XOR entre C[(x-1) mod 5, z] et C[(x+1) mod 5, (z - 1) mod w].

$$D [x, z] = C [(x - 1) \bmod 5, z] \oplus C [(x + 1) \bmod 5, (z - 1) \bmod w] ;$$

3. Pour tous les triplets (x, y, z) tels que $0 \leq x < 5$, $0 \leq y < 5$ et $0 \leq z < w$, A' [x, y, z] est défini comme étant égal à l'opération XOR entre A [x, y, z] et D [x, z].

$$A' [x, y, z] = A [x, y, z] \oplus D [x, z] ;$$

4. Retourner A'

Dans la Figure I.9 qui représente l'étape du processus θ , le symbole de sommation (Σ) est utilisé pour indiquer la parité, qui correspond à la somme XOR de tous les bits présents dans la colonne.

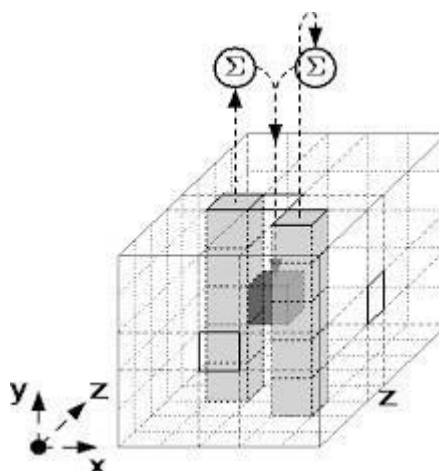


Figure I. 9 Représentation visuelle de l'application de la fonction Thêta sur un seul bit [10]

I.5.3.2. Processus Rho (ρ) :

Ce processus a pour objectif de perturber l'ordre des bits. Cette étape effectue une rotation de chaque voie (lane) d'un certain nombre de positions, qui est le décalage qui dépend des coordonnées fixes x et y de Lane. De manière équivalente, pour chaque bit dans la voie, la coordonnée z est modifiée en ajoutant le décalage, modulo la taille de la voie. Le tableau I. 2 présente le décalage des bits (x,y) effectués lors d'un processus Rho.

Tableau I. 2 Le décalage effectué lors du processus Rho.

	$x = 3$	$x = 4$	$x = 0$	$x = 1$	$x = 2$
$y = 2$	153	231	3	10	171
$y = 1$	55	276	36	300	6
$y = 0$	28	91	0	1	190
$y = 4$	120	78	210	66	253
$y = 3$	21	136	105	45	15

Les étapes du processus :

Entrée : Un tableau d'état A.

Sortie : Un tableau d'état A' après l'application du processus Rho.

1. Pour chaque z tel que $0 \leq z < w$, $A'[0, 0, z]$ est initialisé à la valeur de $A[0, 0, z]$. (x, y) est initialisé à la valeur $(1, 0)$.
2. Pour chaque t allant de 0 à 23 :
 - a. Pour chaque z tel que $0 \leq z < w$, $A'[x, y, z]$ est mis à jour en lui assignant la valeur de $A[x, y, (z - (t + 1)(t + 2) / 2) \bmod w]$.
 - b. (x, y) est mis à jour en lui assignant la valeur $(y, (2x + 3y) \bmod 5)$.
3. Retourner A'.

La Figure I.10 illustre le principe du processus Rho :

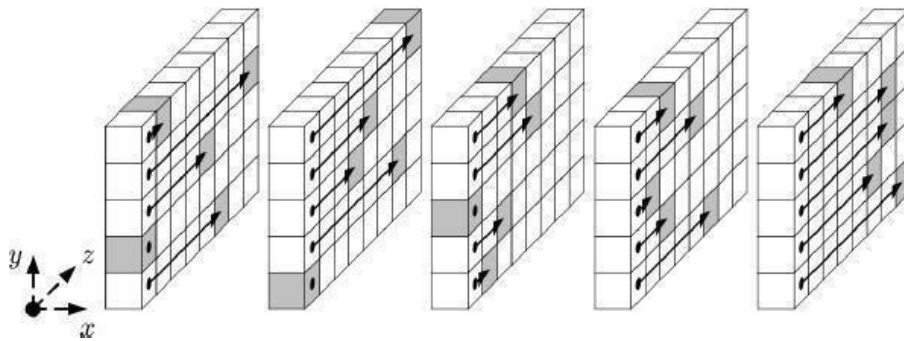


Figure I. 10 Le principe fondamental du processus Rho

I.5.3.3. Processus Pi (π) :

Ce processus a pour objectif de réorganiser les positions des voies. La Figure I.11 montre la convention utilisée pour étiqueter les coordonnées ; ainsi, le bit ayant les coordonnées $x = y = 0$ est représenté au centre de la section.

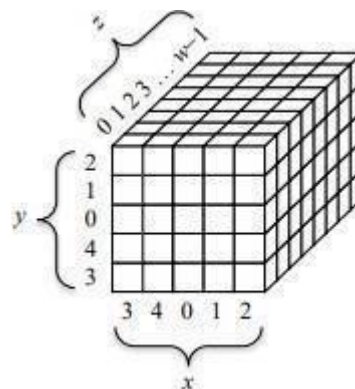


Figure I. 11 L'étiquetage des coordonnées [10].

Les étapes du processus :

Entrée : un tableau d'état A.

Sortie : Un tableau d'état A' après l'application du processus Pi.

1. Pour tous les triplets (x, y, z) tels que x est compris entre 0 et 4, y est compris entre 0 et 4, et z est compris entre 0 et w, A' [x, y, z] est calculé en utilisant la formule suivante : A [(x + 3y) modulo 5, x, z].

$$A' [x, y, z] = A [(x + 3y) \bmod 5, x, z]$$

2. Retourner A'.

La Figure I.12 illustre le principe du processus Pi :

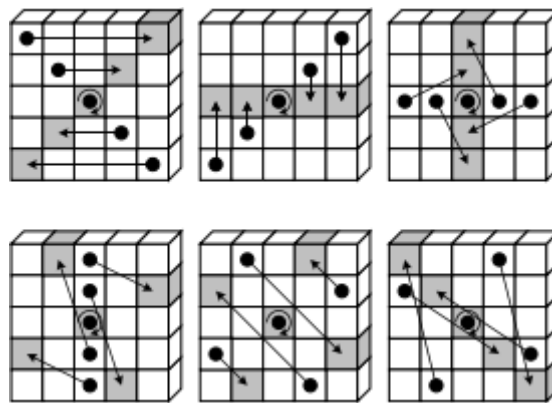


Figure I. 12 Le principe fondamental du processus Pi.

I.5.3.4. Processus Chi (χ) :

Ce processus a pour objectif de traiter chaque ligne du tableau d'état intermédiaire en appliquant une fonction booléenne non linéaire. Ce traitement mélange les bits de chaque ligne de manière à dissiper toute corrélation entre les bits, ce qui renforce la résistance de la fonction de hachage contre les attaques différentielles qui cherchent à exploiter des différences entre les entrées et les sorties de la fonction de hachage pour déduire des informations sur la fonction de hachage elle-même.

Les étapes du processus :

Entrée : un tableau d'état A.

Sortie : Un tableau d'état A' après l'application du processus Chi.

1. Pour tous les triplets (x, y, z) tels que x est compris entre 0 et 4, y est compris entre 0 et 4, et z est compris entre 0 et w, A'[x, y, z] est calculé en appliquant une formule booléenne à partir des éléments de A :

$$A'[x, y, z] = A[x, y, z] \oplus ((A[(x+1) \bmod 5, y, z] \oplus 1) \cdot A[(x+2) \bmod 5, y, z]).$$

2. Retourner A'.

La Figure I.13 illustre le principe du processus Chi :

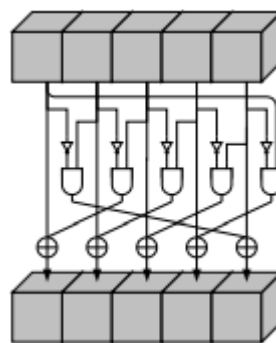


Figure I. 13 Le principe du processus Chi (une seule ligne).

I.5.3.5. Processus Iota (ι) :

Ce processus a pour objectif de générer une constante d'itération RC unique à chaque itération de l'algorithme Keccak. Cette constante est utilisée pour XORer un Lane à chaque itération de l'algorithme, ce qui renforce la sécurité de la fonction de hachage en ajoutant de l'aléatoire. Soit les 24 constantes d'itération RC dans ce tableau :

Tableau I. 3 Les constantes d'itération.

RC[0]	0000000000000000 1	RC[8]	0000000000000008 A	RC[16]	8000000000000802
RC[1]	0000000000000808 2	RC[9]	0000000000000008 8	RC[17]	8000000000000080
RC[2]	8000000000000808 A	RC[10]	0000000080000800 9	RC[18]	0000000000000800 A
RC[3]	8000000080000800 0	RC[11]	0000000080000000 A	RC[19]	8000000080000000 A
RC[4]	0000000000000808 B	RC[12]	0000000080000808 B	RC[20]	8000000080000801
RC[5]	0000000080000000 1	RC[13]	8000000000000008 B	RC[21]	8000000000000808
RC[6]	8000000080000808 1	RC[14]	8000000000000808 9	RC[22]	0000000080000001
RC[7]	8000000000000800 9	RC[15]	8000000000000800 3	RC[23]	8000000080000808

Les étapes du processus :

RC(t) :

Entrée : un entier t.

Sortie : un bit RC(t).

1. Si t est divisible par 255, renvoyer 1.
2. Initialiser R à 10000000.
3. Pour chaque entier i de 1 à t modulo 255, effectuer les opérations suivantes sur R :
 - a. $R = 0 \parallel R$;
 - b. $R[0] = R[0] \text{ XOR } R[8]$;
 - c. $R[4] = R[4] \text{ XOR } R[8]$;
 - d. $R[5] = R[5] \text{ XOR } R[8]$;
 - e. $R[6] = R[6] \text{ XOR } R[8]$;
 - f. $R = \text{Trunc8}[R]$.
4. Retourner R[0].

I(A, ir):

Entrées : un tableau d'état A et l'indice de tour ir.

Sortie : Un tableau d'état A' après l'application du processus Iota.

1. Pour chaque triplet (x, y, z) tel que $0 \leq x < 5$, $0 \leq y < 5$ et $0 \leq z < w$, $A[x, y, z]$ est copie dans $A'[x, y, z]$.
2. Initialiser RC à $0w$.
3. Pour chaque entier j compris entre 0 et l , la valeur de $RC[2j-1]$ est calculé en appliquant la fonction $RC(j+7ir)$.
4. Pour chaque entier z compris entre 0 et $w-1$, l'opération XOR est appliqué entre la valeur de $A'[0, 0, z]$ et la valeur correspondante de RC, c'est-à-dire $RC[z]$.
5. Retourner A' .

I.5.4. Technique de troncature

L'objectif de cette phase est d'effectuer une opération inverse de celle du Mapping, permettant de générer un mot binaire de 1600 bits à partir d'un état sous forme d'un tableau à trois dimensions ($5 \times 5 \times 64$ bits). Une segmentation est ensuite effectuée sur ce mot binaire pour produire une sortie (haché) de la taille souhaitée.

Il est important de noter que la troncature de la sortie de hachage peut affecter la sécurité de la fonction de hachage en réduisant l'espace des sorties possibles et en rendant la fonction de hachage plus vulnérable aux attaques de collision et de préimage. Par conséquent, la longueur de sortie doit être choisie avec soin en fonction des exigences de sécurité de l'application.

I.6. Conclusion

Ce premier chapitre nous a permis de comprendre pleinement l'importance cruciale de la cryptologie dans la préservation de la confidentialité des informations échangées. Nous avons également mis en lumière l'importance des fonctions de hachage cryptographique, soulignant leur rôle central dans la sécurisation des informations confidentielles et des communications numériques. Enfin, nous avons examiné en détail les deux principales constructions de fonctions de hachage, en mettant particulièrement l'accent sur la fonction de hachage cryptographique Keccak. Cette fonction, basée sur la construction éponge et composée de deux phases distinctes, illustre l'ingéniosité et la complexité nécessaires pour renforcer la résistance des fonctions de hachage contre les attaques.

Dans le cadre de notre travail, nous proposons de faire une implémentation matérielle de la fonction de hachage SHA3 sur circuit FPGA. Dans le prochain chapitre nous présentons le principe des circuits FPGA ainsi que les outils utilisés dans le cadre de ce travail.

Chapitre II :

Généralités sur les circuits FPGA

II.1. Introduction

Les dispositifs électroniques programmables, notamment les circuits logiques programmables, jouent un rôle essentiel dans le domaine de l'électronique numérique. Ils offrent une solution indispensable, car la capacité à programmer un composant en fonction des besoins du concepteur s'avère précieuse pour concevoir de manière efficace des circuits complexes avec des délais courts et un coût réduit. Contrairement aux circuits logiques prédéfinis par les fabricants, tels que les portes logiques ou les fonctions logiques, qui ont des entrées et des sorties fixes, une architecture figée, une empreinte spatiale étendue (comprenant quelques portes logiques dans un boîtier volumineux) et une consommation d'énergie élevée, ces composants sont moins adaptés aux petites applications simples et aux circuits de densité moyenne [11].

Dans le cadre de notre travail, nous proposons une implémentation matérielle de la fonction de Hachage SHA3 en exploitant l'FPGA, qui est un dispositif logique programmable très flexible.

Dans ce chapitre, nous présentons les circuits logiques programmables et particulièrement le circuit FPGA ainsi que les outils que nous allons utiliser pour notre implémentation, à savoir : Vivado HLS de Xilinx.

II.2. Aperçu sur les circuits logiques programmables

Les circuits logiques programmables sont des éléments électroniques qui offrent la possibilité aux ingénieurs et aux concepteurs de circuits de définir des fonctions logiques sur mesure en ajustant leur comportement par le biais d'un processus de programmation.

Les circuits logiques programmables peuvent être adaptés pour réaliser diverses opérations logiques, notamment des opérations booléennes, des décodages, des multiplexages, des comptages, et bien d'autres. Ils trouvent fréquemment leur utilité dans un large éventail d'applications, notamment les systèmes embarqués, le traitement du signal numérique, l'automatisation industrielle, ainsi que dans la phase de prototypage électronique.

Diverses catégories de circuits logiques programmables sont disponibles, notamment les PAL (Programmable Array Logic), les PLA (Programmable Logic Array), les GAL (Generic Array Logic), les EPLD (Erasable Programmable Logic Device), les CPLD (Complex Programmable Logic Device) et les FPGA (Field-Programmable Gate Array). Chacun de ces types de composants propose une gamme variée de complexité, de souplesse et de possibilités de

programmation, adaptées pour répondre aux exigences spécifiques des applications électroniques.

Voici une brève description de ces circuits logiques programmables [12].

II.2.1. PAL (Programmable Array Logic)

Il s'agit d'un dispositif logique programmable qui exploite une matrice de portes logiques personnalisables afin d'exécuter des fonctions logiques sur mesure, principalement conçu pour des applications de faible complexité.

II.2.2. PLA (Programmable Logic Array)

Ressemblant au PAL, le PLA se distingue par une flexibilité accrue grâce à la programmation possible de la matrice d'entrées et de sorties, le rendant adapté aux fonctions logiques de plus grande complexité.

II.2.3. GAL (Generic Array Logic)

Les GAL sont des dispositifs logiques programmables qui fournissent une flexibilité comparable à celle des PAL et des PLA, tout en ayant la particularité d'être réutilisables, ce qui les rend précieux pour la création rapide de circuits.

II.2.4. EPLD (Erasable Programmable Logic Device)

Les EPLD sont des dispositifs programmables qui peuvent être effacés et reprogrammés. Ils sont plus polyvalents que les GAL et peuvent être utilisés pour des applications diverses.

II.2.5. CPLD (Complex Programmable Logic Device)

Les CPLD sont des composants logiques programmables de complexité supérieure, fusionnant plusieurs EPLD en une seule unité, ce qui les rend appropriés pour des applications de dimensions allant de moyennes à grandes.

II.2.6. FPGA (Field-Programmable Gate Array)

Les FPGA sont des dispositifs logiques programmables très flexibles permettant de créer des circuits numériques personnalisés à l'aide de blocs logiques configurables.

II.3. Définition d'un circuit FPGA

Les circuits FPGA (Field-Programmable Gate Array) sont des composants électroniques intégrés qui peuvent être programmés pour accomplir des tâches particulières. Ce sont des dispositifs hautement configurables et reprogrammables [12][13].

II.3.1. Structure d'un circuit FPGA

Un circuit FPGA (Field-Programmable Gate Array) est composé de plusieurs éléments clés qui travaillent ensemble pour créer un environnement programmable permettant la conception de circuits logiques personnalisés.

La Figure II.1 représente les éléments qui composent un circuit FPGA.

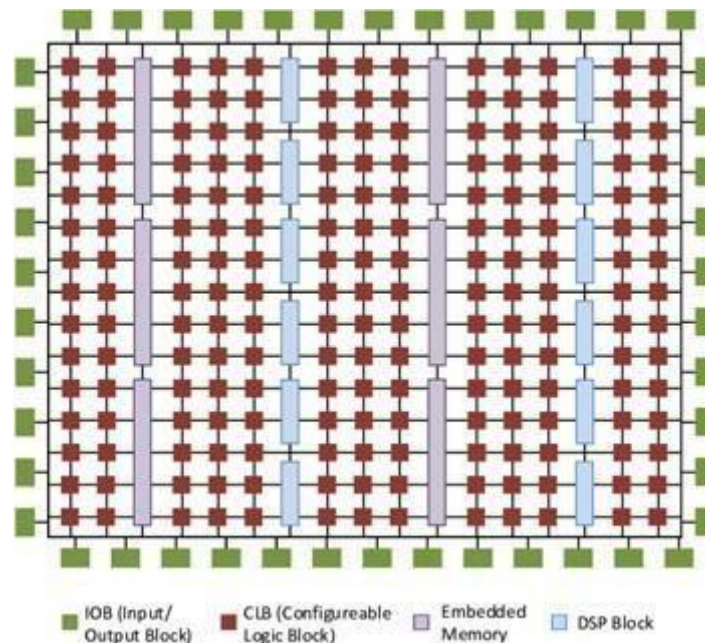


Figure II. 1 . Les composants d'un circuit FPGA

Les principaux composants d'un circuit FPGA sont présenté ci-après.

II.3.1.1. Les blocs logiques configurables (CLB - Configurable Logic Blocks)

Les blocs logiques configurables constituent l'élément fondamental au cœur d'un FPGA. Chacun de ces blocs est une unité modulaire composée de diverses portes logiques, telles que des portes AND, OR, ou NOT, ainsi que de bascules (Flip-Flops).

Ces CLB sont hautement flexibles car ils peuvent être configurés par le concepteur en fonction des besoins de l'application. En utilisant la description matérielle (HDL) fournie par le concepteur, chaque CLB peut être programmé pour exécuter des fonctions logiques spécifiques. Par exemple, on peut configurer un CLB pour réaliser une addition, une soustraction ou une opération plus complexe, en déterminant les connexions internes des portes logiques et des bascules. Cette capacité de personnalisation permet aux concepteurs de créer des circuits numériques sur mesure pour une grande variété d'applications.

II.3.1.2. Les interfaces d'entrée/sortie (I/O - Input/Output)

Les interfaces d'entrée/sortie revêtent une importance cruciale en simplifiant la connexion du FPGA à d'autres composants électroniques ou à des systèmes plus étendus. Ces interfaces englobent une gamme de broches et de configurations spécifiques, offrant ainsi une flexibilité dans la manière dont les données sont échangées avec le monde extérieur.

Les FPGA offrent une variété de configurations d'entrées/sorties, afin de satisfaire les besoins diversifiés des applications. Parmi ces configurations, on peut mentionner les entrées différentielles, qui permettent de mesurer les variations de tension entre deux broches, améliorant ainsi la résistance au bruit électromagnétique. De plus, les sorties haute performance sont disponibles pour assurer une réactivité rapide aux signaux émis. En complément, les FPGA sont capables de prendre en charge des normes de communication spécifiques, telles que les protocoles de communication série ou parallèle, ce qui leur confère une compatibilité avec une vaste gamme de périphériques et de systèmes.

II.3.1.3. Interconnexions programmables

Les interconnexions au sein d'un FPGA jouent un rôle crucial en établissant des connexions entre les blocs logiques configurables (CLB) ainsi qu'entre ces blocs et les points d'entrée et de sortie du FPGA. Ce qui distingue ces interconnexions est leur nature configurable, ce qui implique que le concepteur a le pouvoir de définir avec précision les liaisons spécifiques entre les différentes composantes logiques. Cette capacité à personnaliser les connexions permet de créer des voies de transfert de données sur mesure, adaptées aux exigences de la conception. En d'autres termes, les interconnexions programmables offrent une flexibilité considérable pour acheminer les signaux et les données au sein du FPGA, facilitant ainsi la mise en œuvre d'une large gamme de fonctionnalités logiques complexes et spécifiques à chaque application.

Il y a trois types d'interconnexions entre les divers blocs au sein des circuits FPGA :

- **Interconnexion directe :** Elle relie directement les différents blocs logiques entre eux.
- **Interconnexions par le biais d'une matrice :** Ces interconnexions passent par une matrice pour relier les blocs.
- **Interconnexion par les grandes lignes :** Ces lignes relient tous les CLB aux extrémités des circuits FPGA.

La Figure II.2 donne un récapitulatif des trois méthodes de connexion.

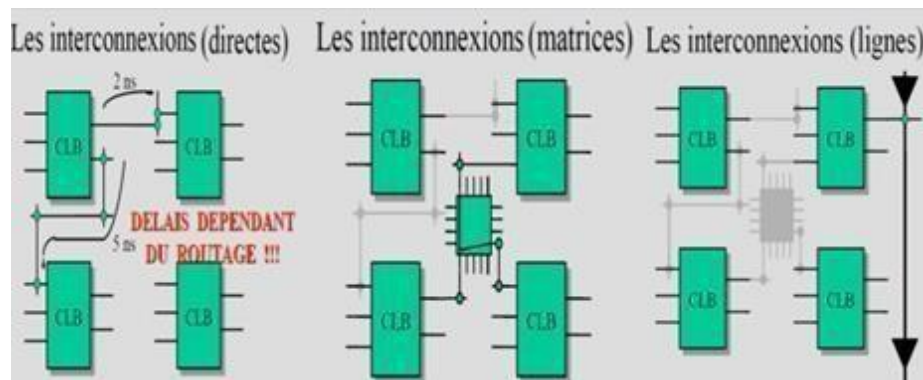


Figure II. 2 Les trois types de connexion dans un circuit FPGA.

II.3.1.4. Unités de Traitement Numérique du Signal (DSP Blocks)

Au sein de certains FPGA, il existe des unités dédiées au traitement numérique du signal (DSP) qui sont spécialement conçues pour accélérer les opérations de traitement numérique. Cette fonctionnalité les rend particulièrement appropriées pour des applications exigeant des calculs intensifs et complexes, généralement dans les domaines du traitement des données analogiques, du filtrage, de la transformation de Fourier, de la modulation/démodulation, ainsi que pour d'autres algorithmes mathématiques nécessaires à la manipulation de signaux numériques. Les blocs DSP (DSP Blocks) fournissent une puissance de calcul optimisée, permettant une exécution rapide et efficace de ces opérations, ce qui se révèle essentiel pour le traitement de signaux en temps réel dans des secteurs tels que les communications sans fil, le traitement audio et vidéo, la vision par ordinateur, et bien d'autres.

II.3.1.5. Blocs de Mémoire (Block RAM)

Dans les FPGA contemporains, il est courant de trouver des blocs de mémoire, parfois appelés blocs RAM, qui servent à la conservation de données temporaires ou de tables de recherche. Ces blocs de mémoire se distinguent par leur configurabilité, ce qui signifie qu'ils peuvent être adaptés en fonction des exigences propres à chaque application.

Ces blocs RAM constituent une ressource précieuse pour le stockage et l'accès rapide à des données, qu'il s'agisse de tampons pour le traitement de flux de données, de mémoires pour des algorithmes de traitement d'images, ou de tables de recherche pour des applications de filtrage. Leur capacité à être configurés selon les besoins spécifiques de l'application en fait un outil polyvalent pour le stockage de données temporaires, contribuant ainsi à optimiser les performances et l'efficacité des FPGA dans une variété de contextes.

Donc un FPGA se compose d'une multitude de blocs logiques configurables (CLB), de blocs mémoire, de blocs dédiés et de blocs d'entrées / sorties. L'ensemble est relié par des interconnexions programmables, qui peuvent être configurées d'une manière flexible pour créer des circuits numériques complexes.

II.4. Le langage de description utilisé

La programmation du circuit FPGA se fait généralement à l'aide d'un langage matérielle (HDL) tels que VHDL ou Verilog.

Le VHDL (VHSIC Hardware Description Language) est un langage de programmation dédié à la description du matériel électronique. Il permet de spécifier le comportement et la structure des circuits numériques utilisés dans les systèmes embarqués, les FPGA et les ASIC. En utilisant VHDL, les concepteurs peuvent définir les fonctionnalités, les connexions et les contraintes temporelles des circuits, facilitant ainsi leur conception, leur simulation et leur vérification [14].

II.5. Étapes de conceptions d'une application FPGA

Les conceptions basées sur FPGA suivent généralement un processus de conception relativement standard, comprenant les étapes suivantes.

II.5.1. Spécification fonctionnelle

La spécification fonctionnelle d'une application FPGA décrit les fonctionnalités et le comportement souhaités du circuit FPGA à concevoir. Elle fournit une description détaillée des entrées, des sorties et des opérations que le circuit doit effectuer.

II.5.2. Langage de description matériel (HDL)

Une fois la spécification fonctionnelle établie, on utilise un langage de description matériel (HDL) tel que VHDL ou Verilog pour décrire le circuit de manière abstraite. Ces langages permettent de définir les composants, les connexions et le comportement du circuit.

Le VHDL permet de spécifier le comportement et la structure des circuits numériques en utilisant des constructions de haut niveau. Il offre la possibilité de définir les fonctionnalités, les contraintes de temporisation et la structure interne d'un circuit. Le langage permet également de créer des entités, des architectures, des signaux, des processus et des composants.

Dans le cadre de notre travail, nous programmons notre algorithme avec le langage C/C++. Nous utilisons l'outil Vivado HLS pour traduire le code en VHDL et générer un IP ensuite l'exporter en RTL afin d'obtenir notre fonction de hachage sous forme d'un IP.

II.5.3. Synthèse

Après l'écriture du code on utilise un outil de synthèse pour le traduire en une représentation logique du circuit. La synthèse le convertit en une structure de portes logiques et de bascules, en utilisant des bibliothèques de composants prédéfinis.

II.5.4. Placement et routage

Le placement et le routage sont deux étapes clés dans la conception d'un circuit FPGA après la synthèse. Ces étapes consistent à déterminer l'emplacement optimal des différents éléments logiques sur le FPGA et à établir les chemins de connexion entre ces éléments.

Le placement se réfère au processus de décision sur l'emplacement physique des différents éléments logiques (portes logiques, bascules, multiplexeurs, etc.) sur les ressources matérielles disponibles du FPGA. L'objectif est de trouver un placement qui minimise les distances entre les éléments logiques liés par des connexions afin de réduire les retards de propagation et d'optimiser les performances du circuit.

Le routage est le processus de création des chemins de connexion (routes) entre les différents éléments logiques placés sur le FPGA. Ces chemins de connexion permettent le passage des signaux entre les éléments logiques et assurent la communication correcte et efficace du circuit. Le routage doit prendre en compte les contraintes spécifiées, telles que les contraintes de temporisation, les contraintes de routage physique, les règles d'interconnexion, etc.

Le placement et le routage sont des étapes complexes et intensives en calcul, car elles nécessitent de prendre en compte plusieurs facteurs, tels que la densité du circuit, les contraintes de temporisation, les interférences entre les signaux, les ressources disponibles, etc. Un placement et un routage réussis contribuent à l'obtention d'un circuit FPGA fonctionnel, performant et fiable.

II.5.5. Vérification de la synchronisation

Après le placement et le routage, une vérification de synchronisation est effectuée pour s'assurer que toutes les contraintes de synchronisation, telles que les retards des signaux, sont respectées. Cela garantit que le circuit fonctionnera correctement à des fréquences données.

II.5.6. Configuration

Une fois que le placement, le routage et la vérification de synchronisation sont terminés, le code généré est téléchargé sur le FPGA. Le FPGA est alors configuré pour exécuter le circuit numérique. On procède ensuite à une vérification pour s'assurer que le circuit se comporte comme prévu et respecte les spécifications fonctionnelles établies à l'étape initiale.

La Figure II.3 représente le flot de conception d'une application FPGA.

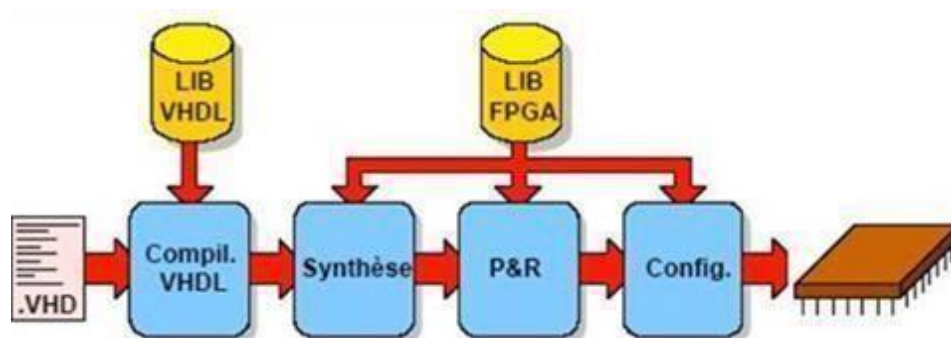


Figure II. 3 Flot de conception d'une application FPGA.

II.6. Application des FPGA

Les FPGA sont utilisés dans de nombreuses applications, notamment [15] :

- **Réseaux de communication** : Les FPGA peuvent être utilisés pour implémenter des protocoles de communication, des passerelles réseau, des pare-feux, des routeurs et d'autres fonctions réseau.
- **Stockage de données** : Les FPGA peuvent être utilisés pour accélérer les opérations de stockage de données, telles que la compression, la décompression et le cryptage.
- **Traitement de signal** : Les FPGA peuvent être utilisés pour réaliser des opérations de traitement de signal en temps réel, telles que la compression audio et vidéo, la reconnaissance de voix, la vision par ordinateur, etc.
- **Industrie** : Les FPGA sont utilisés dans les systèmes de contrôle industriel, tels que les commandes de moteurs, les automates programmables, les robots, les capteurs et les actionneurs.
- **Autres domaines** : Les FPGA sont également utilisés dans de nombreuses autres applications telles que la recherche scientifique, les jeux vidéo, la finance et la cryptomonnaie.

En résumé, les FPGA sont utilisés dans une grande variété d'applications pour améliorer les performances, la flexibilité et la fiabilité des systèmes électroniques.

II.7. Les différentes familles des circuits FPGA

Au début des années 2000, les FPGA ont commencé à être utilisés pour accélérer des algorithmes spécifiques tels que la cryptographie et la compression de données. Les deux principaux producteurs de circuits logiques programmables : Xilinx et Altera. Xilinx est l'un des principaux fournisseurs de FPGA et propose une large gamme de produits pour répondre aux besoins de différents types d'applications.

La famille de FPGA de Xilinx comprend plusieurs séries de produits, notamment :

- **La série Artix** : Cette série de FPGA est destinée aux applications à faible coût et à faible consommation d'énergie, telles que les systèmes embarqués et les équipements de communication.
- **La série Kintex** : Cette série de FPGA est conçue pour offrir des performances élevées avec une faible consommation d'énergie, ce qui la rend adaptée aux applications de traitement de signal, de vision par ordinateur et d'apprentissage en profondeur.
- **La série Virtex** : Cette série de FPGA offre les performances les plus élevées de la

gamme de produits Xilinx, avec des fonctionnalités avancées telles que des blocs de traitement de signal numérique et des blocs de mémoire embarquée.

- **La série Zynq** : la série Zynq de FPGA est une famille de dispositifs FPGA développée par Xilinx. Ces FPGA combinent des blocs de traitement programmables (FPGA) avec des cœurs de processeur ARM pour offrir des fonctionnalités de traitement à la fois programmable et de type système sur puce (SoC).

Dans le cadre de notre travail, nous allons utiliser la carte de développement ZYBO Z7-10 basée sur la gamme Xilinx Zynq 7000. Pour plus de détails, l'auteur peut consulter la référence [16].

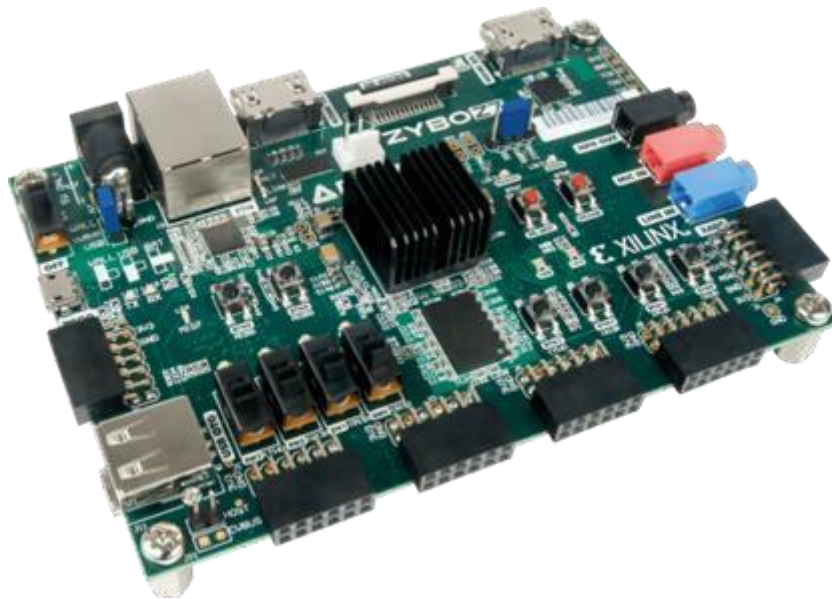


Figure II. 4 Carte de développement ZYBO Z7.

La série Zynq est caractérisée par les éléments suivants [17] :

- **Architecture** : Les FPGA de la série Zynq intègrent un ou plusieurs cœurs de processeur Arm Cortex-A9 ou Cortex-A53. Ces cœurs de processeur fonctionnent en parallèle avec les blocs de traitement FPGA, permettant ainsi d'exécuter à la fois du code logiciel et du code matériel sur la même puce.
- **Ressources FPGA** : Les FPGA de la série Zynq offrent une grande capacité de logique programmable (PL) pour la conception de circuits FPGA personnalisés. Ils comprennent également des blocs de mémoire intégrés, des DSP (Digital Signal Processing) et d'autres fonctionnalités spécifiques pour accélérer le traitement de signaux et d'autres tâches.
- **Système sur puce (SoC)** : La combinaison des cœurs de processeur ARM et des ressources FPGA crée un environnement de système sur puce complet. Cela permet d'exécuter des systèmes d'exploitation complets, tels que Linux, et de développer des applications combinant à la fois du traitement logiciel et matériel.

- **Interfaces** : Les FPGA de la série Zynq sont dotés de diverses interfaces pour la communication avec d'autres composants et périphériques. Cela inclut des interfaces Ethernet, USB, UART, SPI, I2C, HDMI, PCIe, et bien d'autres encore, offrant une connectivité polyvalente.
- **Outils de développement** : Xilinx propose une suite complète d'outils de développement pour la série Zynq, tels que le Vivado Design Suite. Ces outils permettent de concevoir, de simuler et de programmer les FPGA, ainsi que de développer et de déboguer les logiciels exécutés sur les cœurs de processeur ARM.
- **Applications** : La série Zynq est largement utilisée dans de nombreux domaines, notamment l'électronique embarquée, les communications sans fil, l'imagerie médicale, l'aérospatiale, l'automatisation industrielle, l'Internet des objets (IoT) et bien d'autres encore. Les capacités combinées de traitement logiciel et matériel offrent une flexibilité et une performance élevées pour ces applications.
- Chaque série de FPGA de Xilinx est disponible dans différentes tailles et configurations pour répondre aux besoins spécifiques de chaque application. Xilinx propose également une large gamme d'outils de développement et de support technique pour aider les concepteurs à développer des systèmes basés sur FPGA.

II.8. Conclusion

En conclusion, ce chapitre nous a fourni une vue d'ensemble approfondie des circuits logiques programmables, en mettant particulièrement en lumière les FPGA.

Les FPGA se distinguent par leur grande souplesse, leur aptitude à être reconfigurés et leur adaptabilité à une vaste gamme d'applications. L'architecture d'un FPGA englobe des blocs logiques configurables (CLB), des interfaces d'entrée/sortie (I/O), des interconnexions programmables, des blocs de traitement numérique du signal (DSP Blocks) et des blocs de mémoire (Block RAM).

En général, la programmation des FPGA s'effectue à travers l'utilisation de langages de description matérielle (HDL) tels que VHDL ou Verilog. Le processus de conception d'une application FPGA comporte plusieurs étapes, de la spécification fonctionnelle à la vérification finale après le placement et le routage.

Dans le cadre de ce travail, nous effectuons l'implémentation de la fonction SHA3 en langage C. Nous exploitons Vivado HLS (High-Level Synthesis) qui est un outil développé par Xilinx et qui permet de concevoir des circuits numériques à partir de descriptions en haut niveau. Les étapes de travail sont présentées dans le chapitre suivant.

Chapitre III:

Implémentation de la fonction de hachage SHA3 sur FPGA

III.1. Introduction

L'implémentation de la fonction de hachage SHA3 sur FPGA explore la transition de l'algorithme SHA3, décrit précédemment, vers une mise en œuvre matérielle sur des circuits FPGA. Cette étape nécessite une adaptation de l'algorithme pour qu'il soit compatible avec les ressources matérielles spécifiques des FPGA. Nous examinerons en détail le processus de traduction de l'algorithme SHA3 en une conception matérielle utilisable sur les FPGA, en utilisant des outils tels que Vivado HLS. Cette implémentation matérielle permettra une exécution plus rapide et plus efficace de l'algorithme de hachage, adaptée aux exigences de performances des applications embarquées et des systèmes sur puce.

Après avoir exposé en détail l'algorithme SHA3 dans le premier chapitre, nous passons à sa concrétisation pratique en utilisant Vivado HLS. Ce processus englobe plusieurs étapes essentielles : la spécification des entrées et sorties de l'IP, la transformation du code C correspondant à l'algorithme en une conception matérielle FPGA, et enfin la création de l'IP final prêt à être intégré dans des projets FPGA. Ces étapes sont expliquées dans ce chapitre.

III.2. Implémentation de la fonction SHA3 et génération de l'IP

La mise en œuvre de la fonction SHA3 et la création de l'IP impliquent la transformation de l'algorithme SHA3 en une conception matérielle pour simplifier ainsi le processus de conception pour les FPGA.

Il convient de noter que l'algorithme SHA3 a été décrit en détail dans le premier chapitre. Nous allons désormais passer à sa mise en œuvre pratique en utilisant Vivado HLS qui permet de traduire un code C ou C++ en un langage matériel VHDL. Ce processus implique la spécification des entrées et des sorties de l'IP, la conversion du code C correspondant à l'algorithme en une conception matérielle FPGA et la génération de l'IP final prêt à être intégré dans des projets FPGA.

III.2.1. Création du projet

La première étape pour créer un nouveau design HLS consiste à démarrer un nouveau projet. Nous utiliserons l'assistant Nouveau Projet de Vivado HLS pour cette tâche.

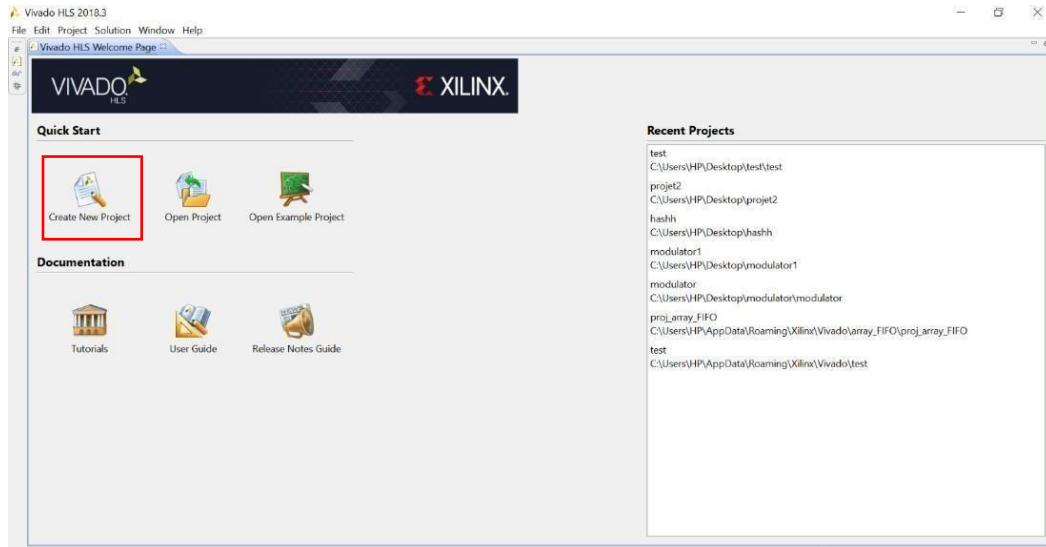


Figure III. 1 Interface de démarrage de Vivado HLS 2018.3

Dans les fenêtres apparues, nous spécifions les détails du projet comme le nom, l'emplacement de sauvegarde et la carte désirée.

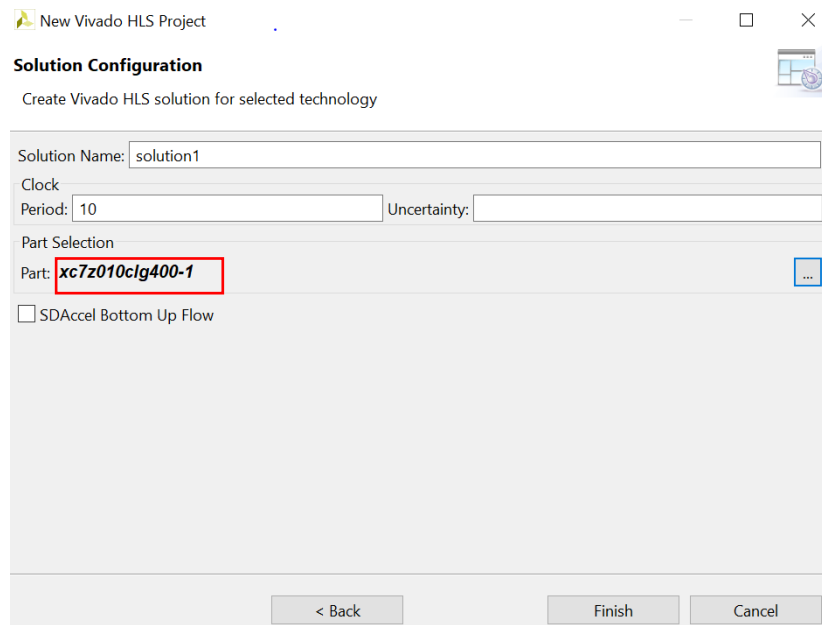


Figure III. 2 Sélection de la carte cible dans Vivado HLS.

Nous ajoutons aussi les fichiers sources.

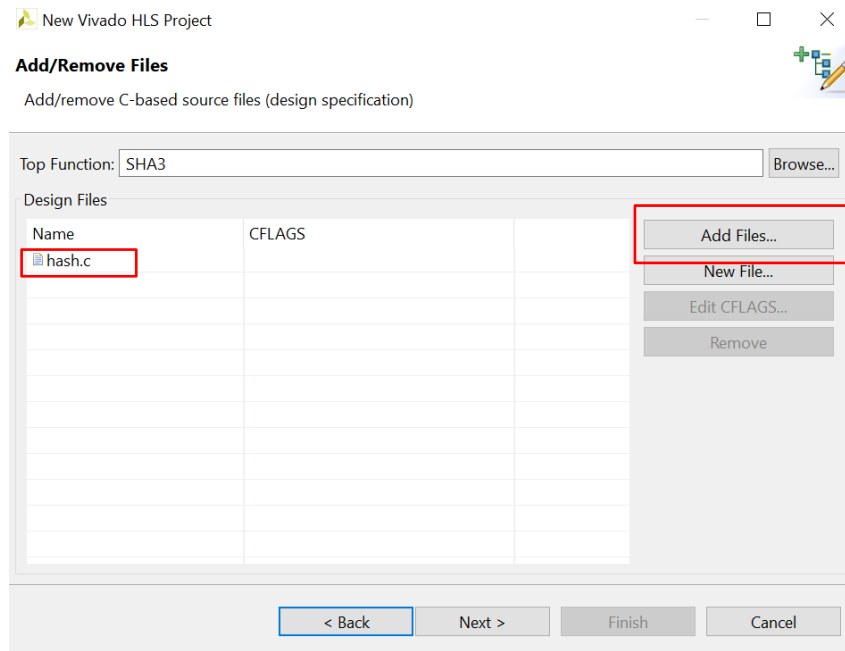


Figure III. 3 Addition de fichiers sources dans Vivado HLS.

III.2.2. Implémentation du programme

L'organigramme ci-dessous illustre les étapes fondamentales du processus de fonctionnement de SHA-3. Il englobe l'initialisation des paramètres, le padding du message, le mapping, la permutation de l'état avec la fonction f de Keccak, la troncature de la sortie, et enfin la génération de l'haché final.

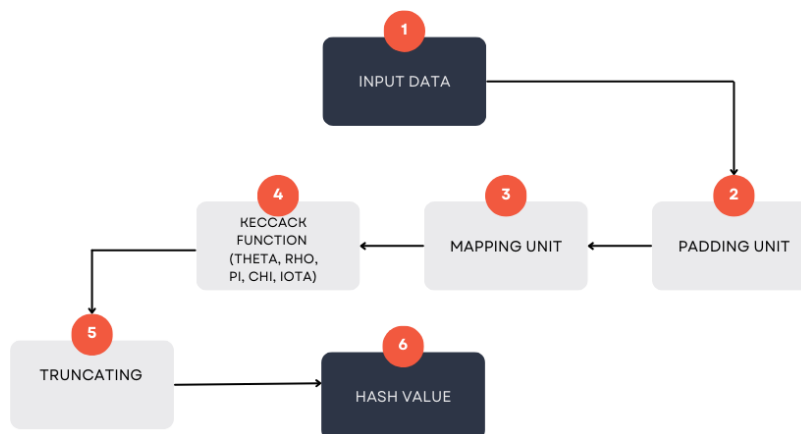


Figure III. 4 Les étapes fondamentales du processus de fonctionnement de SHA-3

La mise en œuvre de la fonction de hachage SHA-3 dans un programme C repose sur l'intégration de cinq processus fondamentaux : Thêta (θ), Rho (ρ), Pi (π), Chi (χ) et Iota (ι) qui sont détaillées dans le premier chapitre. Voici ces processus en langage C :

```
void sha3_keccak_f(unsigned long long A[5][5])
```

```

{
    for (int i = 0; i < SHA3_NR; i++)
    {
        // THETA =====
        unsigned long long C[5];
        unsigned long long D[5];
        // calculate C
        for (int x = 0; x < 5; x++)
        {
            C[x] = 0L;
            for (int y = 0; y < 5; y++)
            {
                C[x] ^= A[y][x];
            }
        }
        // calculate D
        for (int x1 = 0; x1 < 5; x1++)
        {
            D[x1] = C[(x1 + 4) % 5] ^ leftRotateLL(C[(x1 + 1) % 5], 1);
        }
        // put result into A
        for (int x2 = 0; x2 < 5; x2++)
        {
            for (int y = 0; y < 5; y++)
            {
                A[y][x2] ^= D[x2];
            }
        }
        // RHO, PI =====
        unsigned long long B[5][5];
        memset(B, 0, 5 * 5 * sizeof(unsigned long long));
        for (int x3 = 0; x3 < 5; x3++)
        {
            for (int y = 0; y < 5; y++)

```

```

{
    B[(2 * x3 + 3 * y) % 5][y] = leftRotateLL(A[y][x3], sha3_rotConst[y][x3]);
}
}
// CHI =====
for (int x4 = 0; x4 < 5; x4++)
{
    for (int y = 0; y < 5; y++)
    {
        A[y][x4] = B[y][x4] ^ ((~B[y][(x4 + 1) % 5]) & B[y][(x4 + 2) % 5]);
    }
}
// IOTA =====
A[0][0] ^= sha3_roundConsts[i];
}
}

```

Nous avons d'abord utilisé Code::Blocks pour vérifier la validité et la fonctionnalité du programme. Cette étape préliminaire nous a permis de nous assurer que le code était correct et fonctionnel dans un environnement de développement standard avant de le transférer dans Vivado HLS. En procédant ainsi, nous avons pu identifier et corriger d'éventuelles erreurs logiques ou syntaxiques, garantissant ainsi que le code était prêt pour la synthèse et l'optimisation dans Vivado HLS.

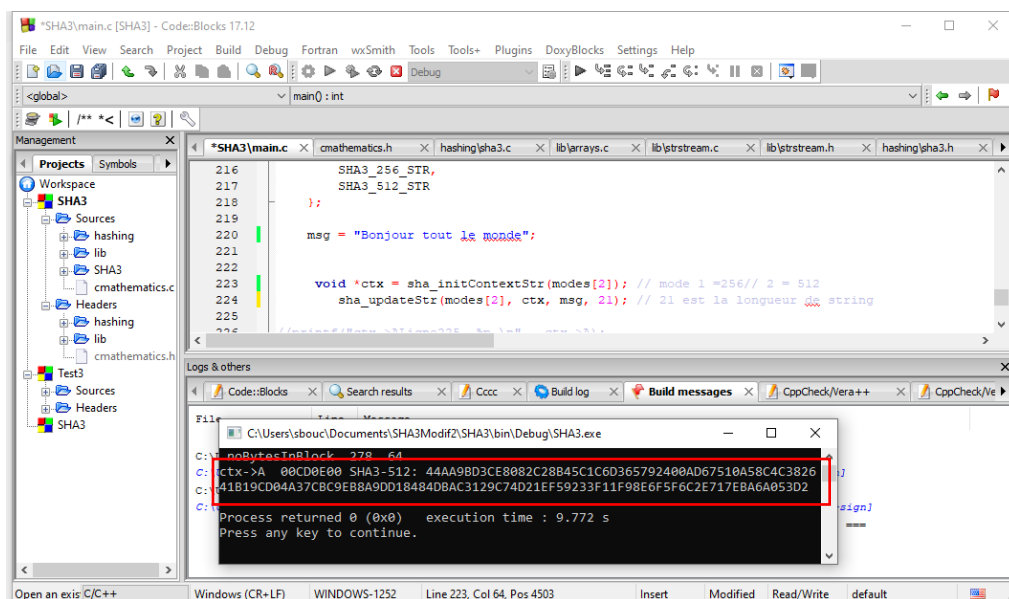


Figure III. 5 Vérification du programme en utilisant Code::Blocks.

Les résultats ont été vérifiés avec des sites de hachage en ligne tel que :

https://emn178.github.io/online-tools/sha3_256.html

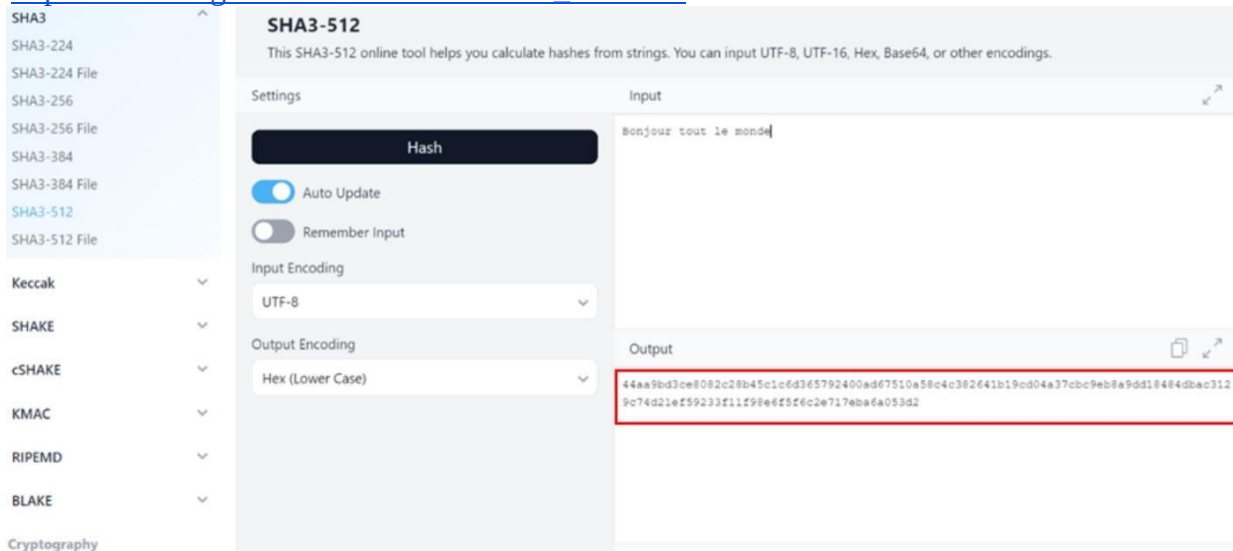


Figure III.6 Résultat obtenu en utilisant le site.

Plusieurs tests sont effectués (Annexe A).

Le code est composé d'un fichier main.c qui orchestre l'exécution du programme. Il gère la saisie du message par l'utilisateur, appelle les fonctions nécessaires pour calculer le hachage, et affiche le résultat. Ce dernier contient plusieurs fonctions tels que :

- sha3_keccak_f : Implémente la fonction de permutation principale de SHA-3 (Keccak-f). Elle effectue des transformations sur un tableau de 5x5 mots de 64 bits en appliquant les étapes Theta, Rho, Pi, Chi et Iota.
- sha_updateStr : Elle commence par déterminer le nombre de blocs complets en fonction des données d'entrée. Ensuite, elle met à jour l'état en effectuant une opération XOR entre les données du message et l'état interne. Après chaque bloc complet, la fonction applique la transformation KECCAK-f pour assurer l'intégrité du calcul.

Plusieurs fichiers headers sont inclus pour fournir des options précises. Parmi ces fichiers :

- sha3.h : définit les éléments nécessaires pour l'implémentation de l'algorithme SHA-3. Il inclut des déclarations pour les constantes de rotation et les constantes de ronde utilisées par SHA-3. De plus, il expose la fonction sha3_keccak_f nécessaire pour la transformation principale de l'état interne. Il sert de lien entre le code de calcul et les fonctions spécifiques à SHA-3, facilitant ainsi l'intégration et l'organisation du code.
- math.h : fournit des déclarations pour les fonctions et constantes mathématiques en C.
- arrays.h : définit des fonctions pour diverses opérations sur des tableaux en C. Il propose des outils pour vérifier la présence d'un élément dans un tableau (arrContains), générer des données aléatoires (newRandomBytes et randomBytes), et manipuler les tableaux en les inversant (reverseArray) ou en les faisant pivoter à gauche et à droite (leftRotate,

rightRotate). De plus, il fournit des fonctions pour effectuer des rotations de bits sur différents types de données (leftRotateLL, rightRotateI, etc.), facilitant ainsi la gestion et la transformation des données dans les tableaux.

III.2.3. Vérification du programme C

Avant de synthétiser un programme C dans Vivado HLS, il est essentiel de le vérifier pour s'assurer de sa validité et de son optimisation. Ce processus est appelé C simulation.

Vivado HLS nous permet d'effectuer cette tâche en utilisant 'Run C simulation'.



Figure III. 6 Vérification du programme en utilisant Vivado HLS

Voici le résultat obtenu, et donc le design est prêt pour la synthèse.

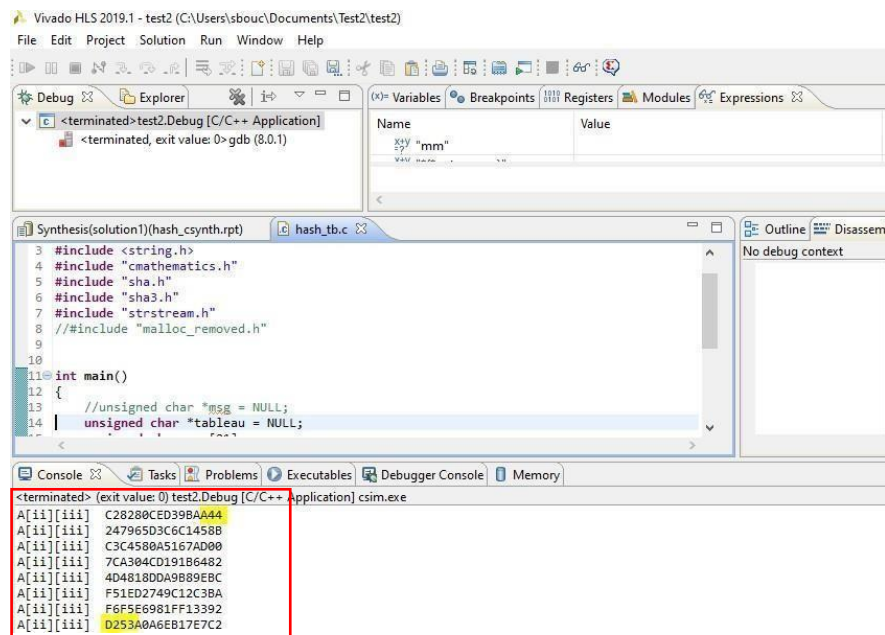


Figure III. 7 Résultat obtenu après C simulation.

III.2.4. Synthèse haut niveau

La synthèse dans Vivado HLS est une étape cruciale qui transforme le code C en une représentation matérielle adaptée au FPGA. Ce processus implique une conversion minutieuse du code source C en une description matérielle optimisée, spécifiquement conçue pour le matériel FPGA cible. Au cours de cette transformation, diverses techniques d'optimisation sont appliquées pour garantir que la conception résultante soit efficace en termes de performance, d'utilisation des ressources matérielles et de consommation d'énergie.

Vivado HLS nous permet d'effectuer cette tâche en utilisant 'Run synthesis'.

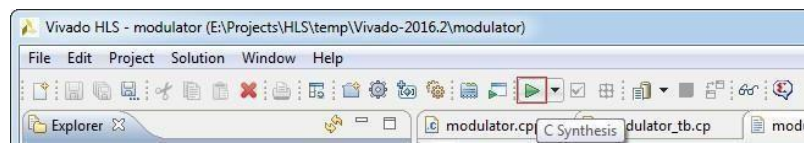


Figure III. 8 Synthèse du programme.

III.2.4.1. Difficultés rencontrées

Vivado HLS (High-Level Synthesis) permet de convertir du code C, C++ ou System C en description matérielle (HDL) pour des FPGA. Cependant, il est nécessaire d'apporter des modifications dans le programme : Dans un premier lieu il est recommandé de rassembler le programme dans un seul fichier C et réduire le nombre des fichiers Headers. Aussi, il existe des contraintes qui sont liées à certaines fonctions qui ne sont pas synthétisables. Dans notre cas, nous avons rencontré des problèmes par rapport à :

- Le pointeurs et arithmétique des pointeurs : Bien que certaines opérations sur les pointeurs puissent être synthétisables, des constructions complexes peuvent poser problème.

Par exemple, dans notre programme, le message d'entrée défini initialement comme un pointeur (`unsigned char * msg`) est converti en tableau (`unsigned char msg[N]`, avec `N` la taille du message d'entrée)

- La gestion dynamique de la mémoire : Les fonctions qui utilisent ``malloc()``, ``free()``, `memcpy` ou d'autres formes d'allocation dynamique de mémoire ne sont pas synthétisables.

Dans notre programme, voici un exemple d'une commande avec `memcpy` :

```
memcpy(&tmp, in + cursor + blockCursor - stateCursor, noBytesInWord);
```

Cette commande copie, dans l'adresse `tmp` (la destination), un nombre d'octet ``noBytesInWord`` à partir d'une position (source) calculée dans le tableau ``in`` (en utilisant les valeurs de ``cursor``, ``blockCursor`` et ``stateCursor``). Cette commande est remplacée par le programme suivant :

```
int j=1;
tmp1=in + cursor + blockCursor - stateCursor; //mettre le contenu de la source dans tmp1
for(i=(noBytesInWord-1); i>0; i--)
{
    tmp=tmp1[i]*pow(256,(noBytesInWord-j)); // Transférer les octets vers tmp en faisant
                                           //des decalage pour pouvoir concatener les different octs
    tmp = tmp + tmp2;                       // faire une concaténation
    tmp2=tmp;
    j=j+1;                                  // incrementer pour passer a l'octet suivant
}
tmp=tmp+tmp1[0];                          // l'octet initiale [0] est positionner a la fin car il ne subit pas le
                                           // decalage (pow(256,(noBytesInWord-j)))
```

Figure III. 9 Programme proposé pour remplacer la commande `memcpy`.

III.2.5. Vérification de l'implémentation RTL

La vérification de l'implémentation RTL dans Vivado HLS est cruciale pour garantir que la conception matérielle résultante du code C répond aux exigences de fonctionnalité et de performance prévues. Cette étape implique l'utilisation de simulations RTL pour évaluer le fonctionnement du circuit synthétisé au niveau du transfert de registres. Des bancs d'essai RTL sont développés afin de tester divers scénarios et configurations, permettant ainsi de détecter et de résoudre tout problème potentiel dans la conception avant la synthèse finale. Cette vérification assure que la conception matérielle générée est précise et conforme aux spécifications définies.

La simulation C/RTL utilise le banc d'essai en C pour vérifier automatiquement la conception RTL. Ce processus se déroule en trois phases :

- La simulation en C est exécutée pour enregistrer les entrées de la fonction de niveau supérieur, ou du Dispositif Sous Test (DUT), sous forme de "vecteurs d'entrée".
- Les "vecteurs d'entrée" sont utilisés dans une simulation RTL à l'aide du RTL généré par Vivado HLS. Les sorties du RTL sont enregistrées sous forme de "vecteurs de sortie".
- Les "vecteurs de sortie" de la simulation RTL sont appliqués au banc d'essai en C, après la synthèse de la fonction, pour vérifier que les résultats sont corrects. Le banc d'essai en C effectue la vérification des résultats.

Vivado HLS nous permet d'effectuer cette tâche en utilisant 'Run C/RTL Cosimulation'.

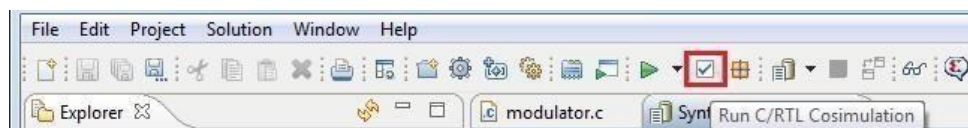


Figure III. 10 C/RTL Cosimulation.

Voici le résultat obtenu :

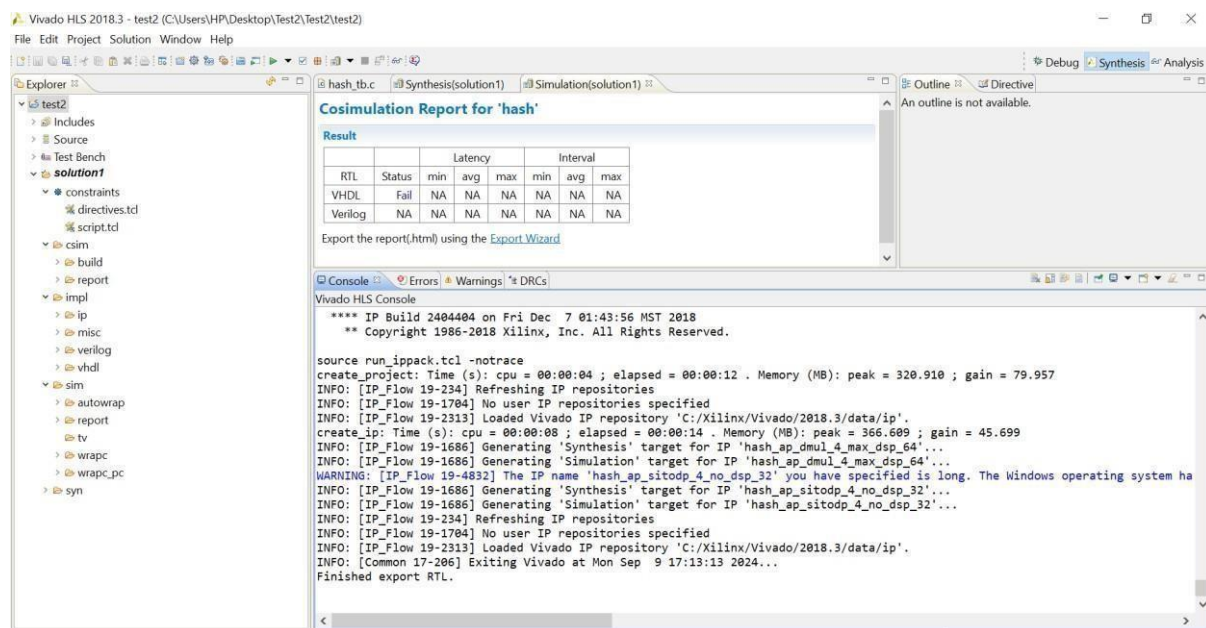


Figure III. 11 Exportation du RTL.

III.2.6. Génération de l'IP

Les IP (Intellectual Property) sont des blocs de conception préfabriqués et réutilisables qui réalisent des fonctions spécifiques, facilitant leur intégration dans des circuits intégrés et accélérant ainsi le développement de systèmes numériques complexes.

L'étape de génération de l'IP sur Vivado HLS est essentielle pour convertir un code C, C++ ou SystemC en une description matérielle en langage HDL comme VHDL ou Verilog. Après la phase de synthèse, où le code est analysé et optimisé pour le matériel, Vivado HLS crée un noyau IP qui encapsule la logique matérielle correspondante. Ce noyau IP peut ensuite être intégré dans un design plus complexe avec Vivado Design Suite.

La Figure III.11 montre la génération de l'IP.

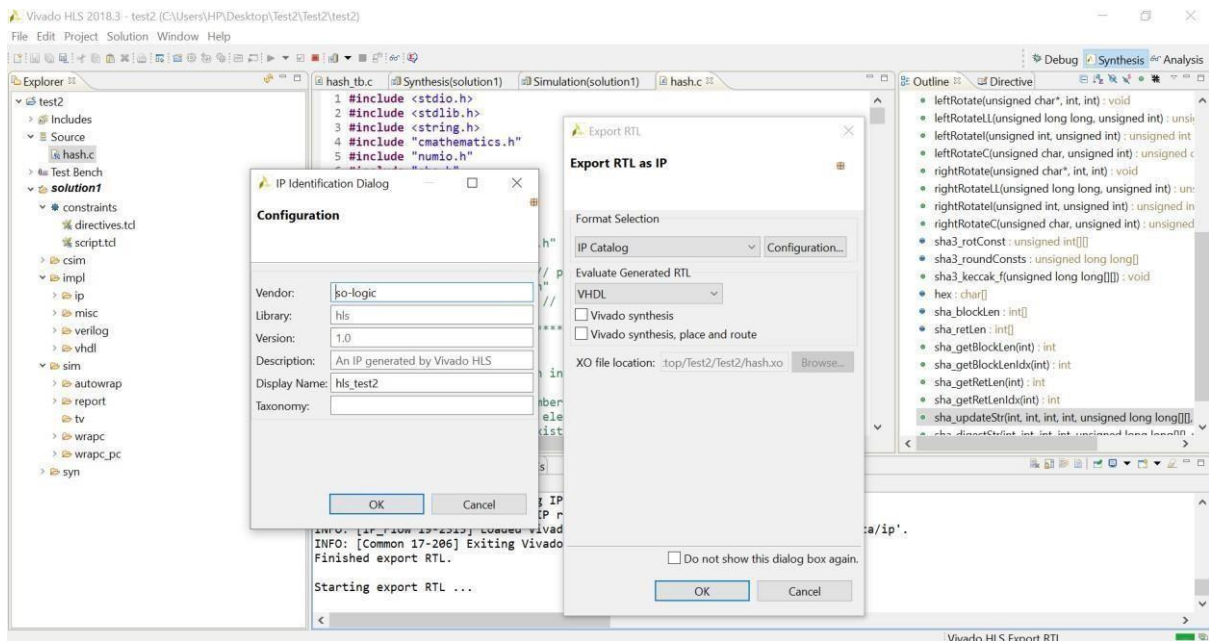


Figure III. 12 Génération de l'IP.

L'IP est prêt à intégrer dans une architecture matérielle en utilisant Vivado.

III.3. Implémentation sur la carte FPGA

Au cours de cette phase, nous utiliserons l'environnement logiciel VIVADO pour intégrer notre IP dans une carte FPGA.

La figure III.13 représente l'interface du logiciel VIVADO, on commence par la création d'un nouveau projet en spécifiant son nom et son emplacement.

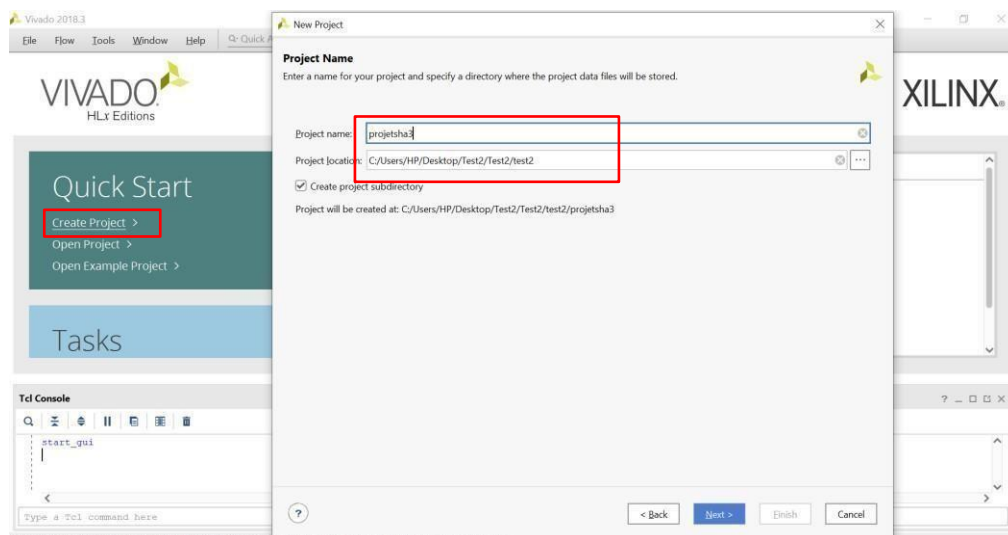


Figure III. 13 Création du projet.

On ajoute notre IP dans la bibliothèque des IPs, puis on crée un block design et y intègre notre IP.

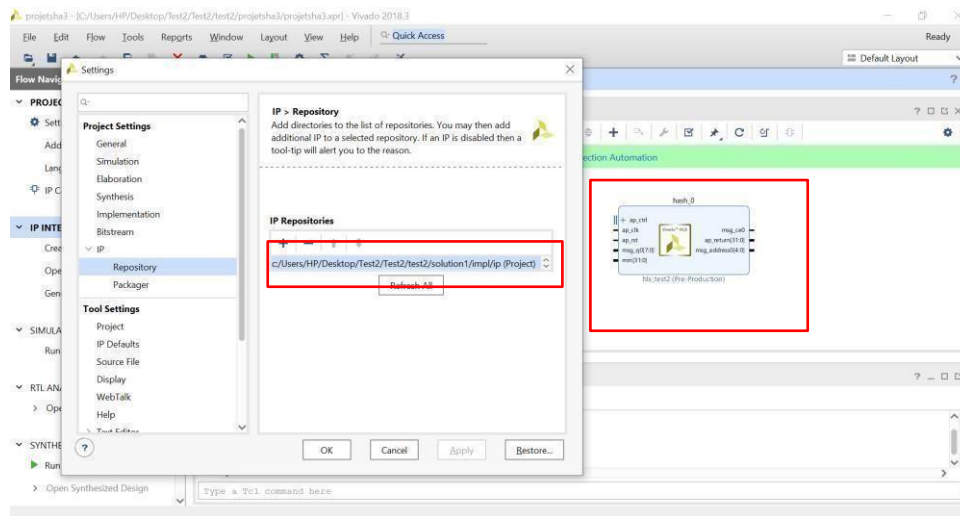


Figure III. 14 L'ajout de l'IP créé à la bibliothèque du VIVADO.

Après avoir développé et testé le bloc IP dans Vivado HLS, il faut l'exporter vers Vivado. Dans Vivado, on crée un projet et on ajoute le bloc IP au design. On l'intègre dans le « block design », ajuste ses connexions et configure ses paramètres spécifiques. Avant la synthèse, une simulation fonctionnelle est essentielle pour vérifier le comportement du bloc IP. Une fois la simulation validée, on procède à la synthèse, transformant le bloc IP en une représentation matérielle. Ensuite, l'implémentation optimise et place le design sur le FPGA. Enfin, on génère le bitstream, un fichier binaire qui configure le FPGA selon le design du bloc IP.

III.4. Conclusion

Dans ce chapitre, nous avons exploré en détail le processus d'implémentation de la fonction de hachage SHA3 sur FPGA. L'utilisation de l'outil Vivado HLS a été cruciale pour sa mise en œuvre pratique, permettant la traduction efficace du code C en matériel VHDL à travers des étapes essentielles.

En conclusion, ce chapitre jette les bases nécessaires pour une mise en œuvre de la fonction SHA3 sur FPGA, ouvrant la voie à de nombreuses applications dans le domaine de la sécurité informatique et de la cybersécurité.

Conclusion générale

Les fonctions de hachage sont très importantes pour la fiabilité des systèmes de communication, elles permettent d'authentifier les données, vérifier leur intégrité et renforcer la sécurité des communications. La fonction de hachage SHA3 se distingue par ses performances et sa flexibilité en termes de taille de sortie puisqu'elle offre des tailles de sortie différentes selon le besoin de l'application en termes de sécurité.

L'implémentation hardware d'une fonction de hachage est très importante afin d'améliorer ses performances, son efficacité énergétique et sa résilience à différents types d'attaques.

Dans le cadre de ce travail, nous avons exploré l'implémentation de la fonction de hachage SHA-3 sur FPGA à travers l'utilisation de VIVADO HLS, en abordant successivement les aspects théoriques.

Nous avons d'abord expliqué les mécanismes internes de l'algorithme SHA3, ainsi que son importance dans le domaine de la cryptographie moderne. Nous avons également rappelé certains concepts des FPGA.

Dans la partie pratique nous avons utilisé un outil très important dans l'implémentation Hardware sur FPGA, c'est l'outil HLS (High-Level Synthesis) qui permet la conception de systèmes en utilisant des langages de haut niveau (C, C++). En effet, ayant implémenté la fonction SHA3 initialement en C en utilisant l'environnement de Code Blocks, HLS devient très utile pour la transition de la conception logicielle vers une mise en œuvre matérielle efficace. Toutefois nous avons rencontré à ce niveau des difficultés étant donné que certaines instructions en C ne sont pas synthétisables nécessitant d'être remplacées par un autre code équivalent en termes de fonction. Enfin, nous avons pu générer le bloc IP (Intellectual Property) qui contient la fonction et qui peut être intégré dans un circuit intégrable programmable.

Comme perspectives de travail, les étapes de simulation et de synthèse doivent être effectuées après l'intégration dans un design FPGA. La validation pourra être effectuée sur un environnement réel avec un test matériel permettant ainsi de réfléchir à des pistes d'optimisation.

Annexes

Annexe A :

Le tableau suivant illustre quelques exemples pour tester la validité de notre programme C.

Message à hacher	Fonction de hachage
Hachage mode 256	efdda31f60f0875cfd08ee208835a8c8ef3e5a54bd7125177facbb4b4992b600
Message à hacher	PFE
Hachage mode 256	5e4c05cad01c90a240b4be02d944e22ca8fb9cd06c3b3e1b2784a6472e5ffb2f
Message à hacher	Programme C
Hachage mode 512	9c60b73560fdc6dd7106a269a6a00afaa1bf7ab4c726089d466d61be4e8ca30c e330311af50eff3ff030b66d5c935e0338bd5da58b55ffb2b099cd91c82b8bb3

Les résultats obtenus en utilisant le site, CodBlocks et VIVADO HLS respectivement:

Exemple 1 : Fonction de hachage

The image displays two screenshots related to the SHA3-256 hashing function.

The top screenshot shows a web browser window with the URL `emn178.github.io/online-tools/sh...`. The page title is **SHA3-256**. The description states: "This SHA3-256 online tool helps you calculate hashes from strings. You can input UTF-16, Hex, Base64, or other encodings." The **Input** field contains the text "Fonction de hachage". The **Output** field displays the resulting hash: `7b58e9b77988d03df997fbcf9c58948ffed4b680ce8a7a23405547448846001f`.

The bottom screenshot shows the Code::Blocks IDE interface. The project is named **SHA3**. The source file `sha3.c` is open, showing the following code:

```
215     SHA3_128_STR,  
216     SHA3_256_STR,  
217     SHA3_512_STR  
218 };  
219  
220     msg = "Fonction de hachage";  
221  
222
```

The IDE's console window shows the execution output:

```
noBytesInBlock 278 32  
ctx->A 00A40E00 SHA3-256: 7B58E9B77988D03DF997FBCF9C58948FFED4B680CE8A7A23405547448846001F  
Process returned 0 (0x0)   execution time : 6.106 s  
Press any key to continue.
```

Vivado HLS 2019.1 - test2 (C:\Users\sbouc\Documents\Test2\test2)

File Edit Project Solution Run Window Help

Debug Explorer i-> Variables Breakpoints

<terminated>test2.Debug [C/C++ Application]
 <terminated, exit value: 0>gdb (8.0.1)

hash.c hash_tb.c hash_A.vhd hash_ap_sitodp_

```

12 {
13     //unsigned char *msg = NULL;
14     unsigned char *tableau = NULL;
15     //unsigned char msg[21];
16     unsigned char msg[19];
17     int mm;
18     //msg = "Bonjour tout le monde";
19     //tableau = "Bonjour tout le monde";
  
```

Console Tasks Problems Executables Debugger Console

<terminated> (exit value: 0) test2.Debug [C/C++ Application] csim.exe

```

A[ii][iii] 3DD08879B7E9587B
A[ii][iii] 8F94589CCFFB97F9
A[ii][iii] 237A8ACE80B6D4FE
A[ii][iii] 1F00468844475540
  
```

Exemple 2 : PFE

← → ↺ emn178.github.io/online-tools/sh...

☰

SHA3-256

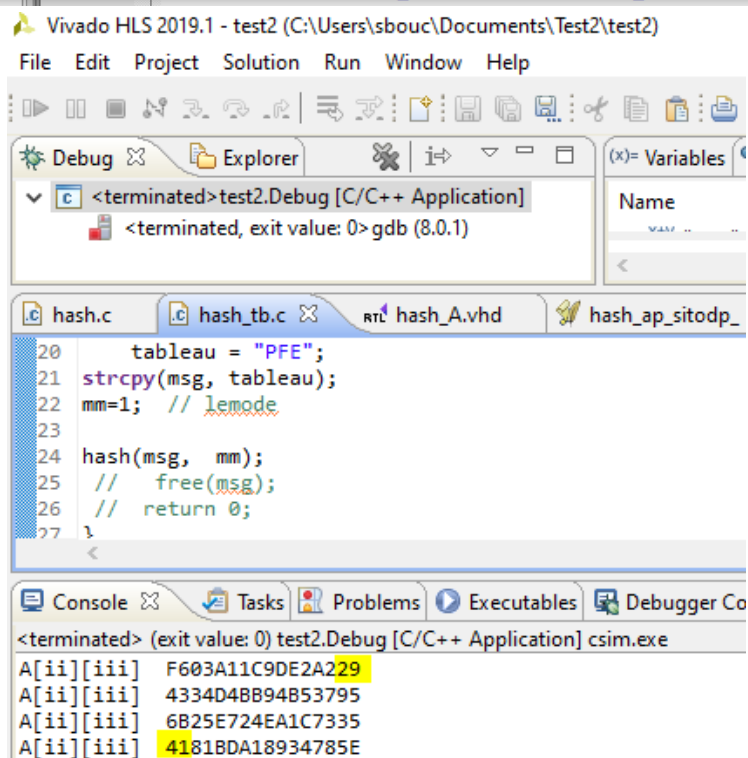
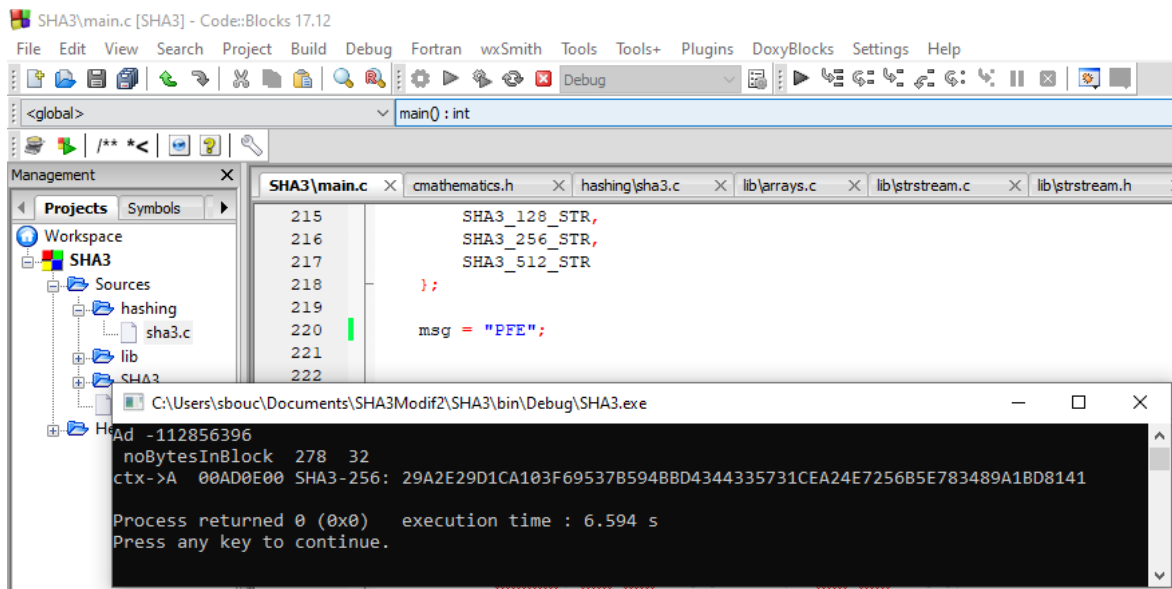
This SHA3-256 online tool helps you calculate hashes from strings. You can input UTF-16, Hex, Base64, or other encodings.

Input

PFE

Output

29a2e29d1ca103f69537b594bbd4344335731cea24e7256b5e783489a1bd8141



Exemple 3 : Programme C

SHA3-512

This SHA3-512 online tool helps you calculate hashes from strings. You can input UTF-8, UTF-16, Hex, Base64, or other encodings.

Input

Programme C

Output

585e2db512f4e00ab54ad0de359102cc7ecb1092fff4c473eff122e59f9e326a53ff295e1cc752d6a75be0a5f8418016048000a3a73ad86c0b7ca1a809195fdb

*SHA3\main.c [SHA3] - Code::Blocks 17.12

File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help

<global> main() : int

219
220 msg = "Programme C";
221
222
223 void *ctx = sha_initContextStr(modes[2]); // mode 1 =256// 2 = 512
224 sha_updateStr(modes[2], ctx, msg, 11); // 11 est la longueur de
225

C:\Users\s bouc\Documents\SHA3Modif2\SHA3\bin\Debug\SHA3.exe

noBytesInBlock 278 64
ctx->A 00700E00 SHA3-512: 585E2DB512F4E00AB54AD0DE359102CC7ECB1092FFF4C473EFF122E59F9E326A53FF295E1CC752D6A75BE0A5F8418016048000A3A73AD86C0B7CA1A809195FDB

Process returned 0 (0x0) execution time : 0.121 s
Press any key to continue.

Vivado HLS 2019.1 - test2 (C:\Users\sbouc\Documents\Test2\test2)

File Edit Project Solution Run Window Help

Debug Explorer

<terminated>test2.Debug [C/C++ Application]
<terminated, exit value: 0> gdb (8.0.1)

hash.c hash_tb.c hash_A.vhd hash_a

```
19     tableau = "Programme C";
20     strcpy(msg, tableau);
21     mm=2; // lemode
22
23     hash(msg, mm);
24     // free(msg);
25
```

Console Tasks Problems Executables Del

<terminated> (exit value: 0) test2.Debug [C/C++ Application] csim.ex

```
A[ii][iii] 0AE0F412B52D5E58
A[ii][iii] CC029135DED04AB5
A[ii][iii] 73C4F4FF9210CB7E
A[ii][iii] 6A329E9FE522F1EF
A[ii][iii] D652C71C5E29FF53
A[ii][iii] 168041F8A5E05BA7
A[ii][iii] 6CD83AA7A3008004
A[ii][iii] DB5F1909A8A17C0B
```

Références et bibliographie

- [1] S. El Moumni. Implémentation à haute performance de la fonction de hachage cryptographique SHA3 sur des circuits FPGA, Thèse de Doctorat. Département de Physique. Université Hassan II de Casablanca, Faculté des Sciences ben M'sick.Maroc.2020. Disponible sur : https://www.researchgate.net/publication/352329857_Implementation_a_haute_performance_de_la_fonction_de_hachage_cryptographique_SHA3_sur_des_circuits_FPGA.
- [2] R Rivest and S Dusse. The MD5 Message Digest Algorithm, Network Working Group Internet Draft. 1991.
- [3] National Institute of Standards and Technology, FIPS PUB 180.disponible : <http://security.isu.edu/pdf/fips180.pdf>, 1993).
- [4] NIST (National Institute of Standards and Technology). "SHA-3 Standard : Permutation-Based Hash and Extendable-Output Functions." FIPS PUB 202.
- [5] NIST (National Institute of Standards and Technology). "SHA-3 Standard : Permutation-Based Hash and Extendable-Output Functions." FIPS PUB 202 (2015).
- [6] W. Stallings, Cryptography and Network Security: Principles and Practice, 7eme édition, Upper Saddle River: NJ: Pearson Education, 2020.
- [7] A. J. Menezes, P. C. van Oorschot, et S. A. Vanstone, Handbook of Applied Cryptography. Boca Raton, CRC Press, 1996.
- [8] A. Zellagui. Synthèse et implémentation des fonctions de hachage appliquées à la signature électronique, Thèse de Doctorat. Département électronique. Université d'Oran Mouhamed Boudiaf, Algerie.2022. Disponible : http://www.univusto.dz/theses_en_ligne/doc_num.php?explnum_id=3571
- [9] G. Bertoni, J. Daemen , M. Peeters et G. Van Assche. The Keccak Reference. 2011. Disponible sur : <https://keccak.team/files/Keccak-reference-3.0.pdf>.
- [10] National Institute of Standards and Technology NIST. SHA-3 Standard : Permutation-Based Hash and Extendable-Output Functions. NIST Special Publication 800-185. 2015. Disponible sur : <https://csrc.nist.gov/publications/detail/fips/202/final>

- [11] J. A. Johnston, "Programmable Logic Devices: A Key to Efficient Digital Design," *IEEE Transactions on Circuits and Systems*, 2003.
- [12] S. Brown et Z. Vranesic. Fundamentals of Digital Logic with Verilog Design. McGraw-Hill Education, 2017. Disponible :
<https://notesavior.files.wordpress.com/2018/02/stephen-brown-and-zvonko-vranesic-fundamental-of-digital-logic-with-verilog-design.pdf>
- [13] C. Maxfield. The Design Warrior's Guide to FPGAs: Devices, Tools, and Flows. Newnes. 2004.
- [14] T. Cuvelier, Introduction au langage VHDL. Développez.com. 2004. Disponible sur :
<https://tcuvelier.developpez.com/tutoriels/vhdl/introduction-langage/>.
- [16] M. Ziegler. Applications of FPGAs in Industrial Automation. IEEE Industrial Electronics Magazine.2018.
- [17] Xilinx Inc. Xilinx FPGA Families Overview. Xilinx Inc. Website. Disponible sur :
<https://www.xilinx.com/products/silicon-devices/fpga.html>
- [18] Xilinx Inc. Xilinx Zynq-7000 All Programmable SoC Overview. Xilinx Inc. Website. Disponible sur :
<https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html>

ملخص

التشفير هو عنصر أساسي في تأمين الاتصالات والشبكات، خاصة من خلال استخدام دوال التجزئة. وتعتبر SHA-3، التي تم اعتمادها كمعيار من قبل NIST، من بين أقوى دوال التجزئة التشفيرية، حيث تتميز بمرونتها وقدرتها على توليد مخرجات بأحجام مختلفة لتلبية احتياجات التطبيقات المتنوعة.

في هذا العمل، نقوم بدراسة مبدأ عمل دالة SHA-3 واستعراض خطوات تنفيذها على مستوى جهاز FPGA. قمنا ببرمجة الدالة SHA-3 باستخدام لغة البرمجة C، ثم استخدمنا أداة HLS Vivado لتحويل النسخة البرمجية إلى تصميم مادي، وإنشاء IP يمكن دمجه في دارة متكاملة قابلة للبرمجة.

الكلمات المفتاحية: التشفير، دالة التجزئة، IP، HLS Vivado، FPGA، SHA-3، أمن الاتصالات.

Résumé

La cryptographie est essentielle pour la sécurisation des communications et des réseaux, notamment par l'utilisation des fonctions de hachage. Le SHA-3, standardisé par le NIST, se distingue parmi les fonctions de hachage cryptographiques pour sa robustesse et sa flexibilité, offrant des tailles de sortie variables adaptées aux exigences spécifiques des applications.

Dans ce travail nous étudions le principe de la fonction SHA3 ainsi que les étapes d'une implémentation Hardware sur FPGA. Nous implémentons la fonction SHA3 et nous exploitons l'outil VIVADO HLS pour transformer la version logicielle et générer le bloc IP qui pourra être intégré dans un circuit programmable.

Mots-clés : cryptographie, fonction de hachage, SHA3, FPGA, VIVADO HLS, IP, sécurité des communications.

Abstract

Cryptography is essential for securing communications and networks, particularly through the use of hash functions. SHA-3, standardized by NIST, stands out among cryptographic hash functions for its robustness and flexibility, offering variable output sizes tailored to specific application requirements.

In this work, we study the principles of the SHA-3 function as well as the steps involved in a hardware implementation on FPGA. We implement the SHA-3 function and utilize the VIVADO HLS tool to transform the software version and generate the IP block that can be integrated into a programmable logic device.

Keywords: cryptography, hash function, SHA-3, FPGA, VIVADO HLS, IP, communication security.