

PROJET VHDL SECURE HASH ALGORITHM

RÉALISÉ PAR

MOUSLIKI Ismail
HOUSSAM Abderrahmane
KHIZRAN Akram

SUPÉRVISION

Pr.El Mounni



SOMMAIRE

INTRODUCTION

- Objectif
- Description

ETUDE DE L'ALGORITHME SHA-256

- Prétraitement
- Initialisation des variables de hachage
- Transformation des blocs de message
- Mise à jour des variables de hachage
- Production du hash final
- Résumé

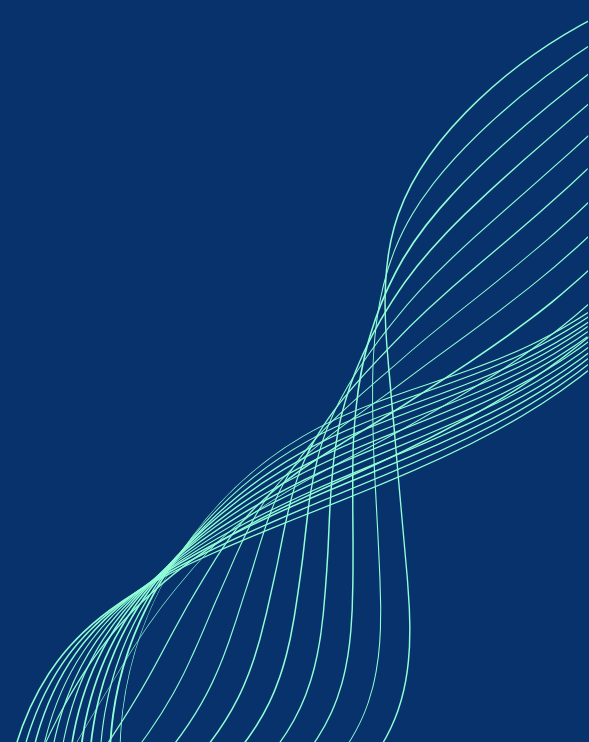
CONCEPTION DE L'ARCHITECTURE

- Décomposition en blocs

IMPORTANCE ET DOMAINE D'APPLICATION

DEFIS

CONCLUSION



INTRODUCTION

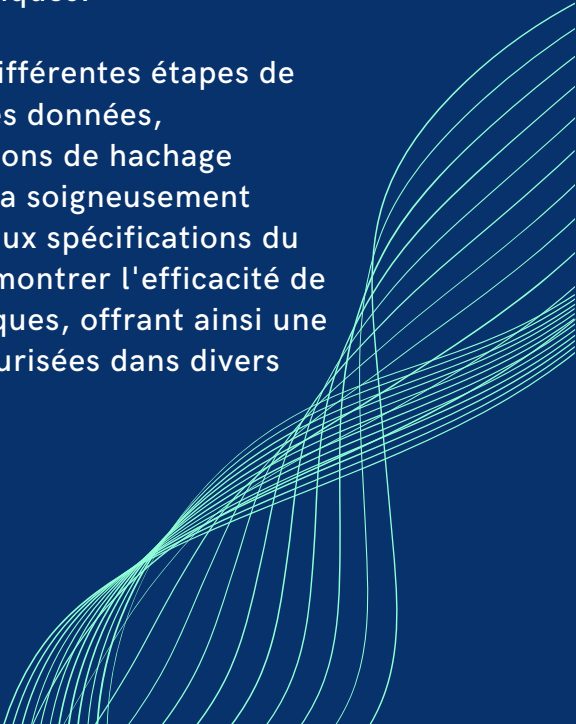
OBJECTIF

L'objectif de ce projet est d'implémenter l'algorithme SHA-256 en VHDL, un langage de description matérielle utilisé pour concevoir et simuler des circuits numériques. SHA-256 (Secure Hash Algorithm 256 bits) est un algorithme de hachage cryptographique qui prend en entrée un message de longueur variable et produit un condensat (hash) de 256 bits. Cet algorithme est largement utilisé pour garantir l'intégrité des données et la sécurité des informations dans divers domaines, notamment dans les signatures numériques, les certificats SSL/TLS et les systèmes de crypto-monnaie comme Bitcoin.

Dans le cadre de ce projet, nous visons à implémenter l'algorithme SHA-256 en utilisant le langage de description de matériel VHDL. SHA-256, appartenant à la famille des algorithmes Secure Hash Algorithm (SHA), est un standard en matière de cryptographie, produisant un condensat de 256 bits à partir d'une entrée donnée. Cet algorithme est largement utilisé pour garantir l'intégrité et la sécurité des données, jouant un rôle crucial dans diverses applications, telles que les signatures numériques, la vérification de l'intégrité des fichiers, et les preuves de travail dans les systèmes de crypto-monnaies.

L'implémentation de SHA-256 en VHDL offre une solution matérielle efficace et sécurisée pour le calcul des valeurs de hachage, essentielle pour les systèmes embarqués et les applications à hautes performances. Ce projet permettra non seulement de concevoir une architecture robuste et fiable pour le calcul de hachages cryptographiques, mais également de renforcer notre compréhension des concepts fondamentaux de la cryptographie et de la conception de circuits numériques.

En réalisant ce projet, nous allons explorer les différentes étapes de l'algorithme SHA256, y compris le prétraitement des données, l'initialisation des variables de travail, et les opérations de hachage complexes. Chaque composante de l'algorithme sera soigneusement implémentée et testée pour garantir la conformité aux spécifications du standard SHA-256. Cette implémentation vise à démontrer l'efficacité de l'approche matérielle pour les calculs cryptographiques, offrant ainsi une base solide pour le développement de solutions sécurisées dans divers domaines technologiques

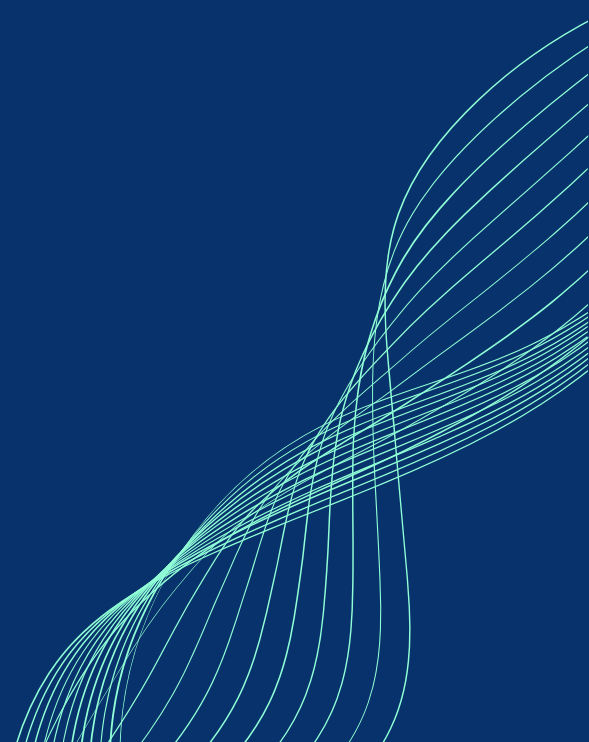


DESCRIPTION

L'algorithme SHA-256 fait partie de la famille des algorithmes de hachage SHA-2, définie par le National Institute of Standards and Technology (NIST). Il fonctionne en plusieurs étapes clés :

1. Prétraitement: Le message d'entrée est complété (padding) et découpé en blocs de 512 bits.
2. Initialisation: Les variables initiales de hachage sont définies par des constantes spécifiques.
3. Traitement des blocs de message: Chaque bloc de 512 bits est traité via une série d'opérations logiques et arithmétiques, produisant des valeurs intermédiaires de hachage.
4. Finalisation: Les valeurs intermédiaires sont combinées pour produire le condensat final de 256 bits.

L'implémentation de SHA-256 en VHDL implique la conception de modules pour chaque étape de l'algorithme, ainsi que la simulation et la vérification de ces modules pour assurer leur conformité aux spécifications.



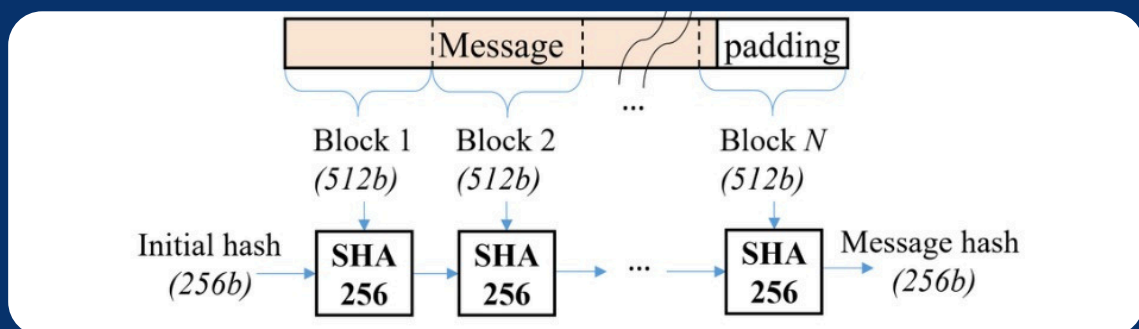
ÉTUDE DE L'ALGORITHME

SHA-256

PRÉTRAITEMENT

Objectif : Préparer le message pour qu'il soit compatible avec les opérations de hachage.

Padding



1. Ajouter un bit '1' :

- Ajoutez un bit '1' à la fin du message. Cela garantit que le message est différent de tout message qui n'aurait pas été complété.

2. Ajouter des bits '0'* :

- Ajouter des bits '0' jusqu'à ce que la longueur du message soit congruente à 448 modulo 512. Autrement dit, la longueur du message doit être 64 bits de moins qu'un multiple de 512 bits.

3. Ajouter la longueur du message* :

- Ajouter 64 bits représentant la longueur originale du message en bits avant le padding. Ces 64 bits sont ajoutés à la fin du message complété.

Exemple :

Supposons que le message original soit "abc" (en ASCII) :

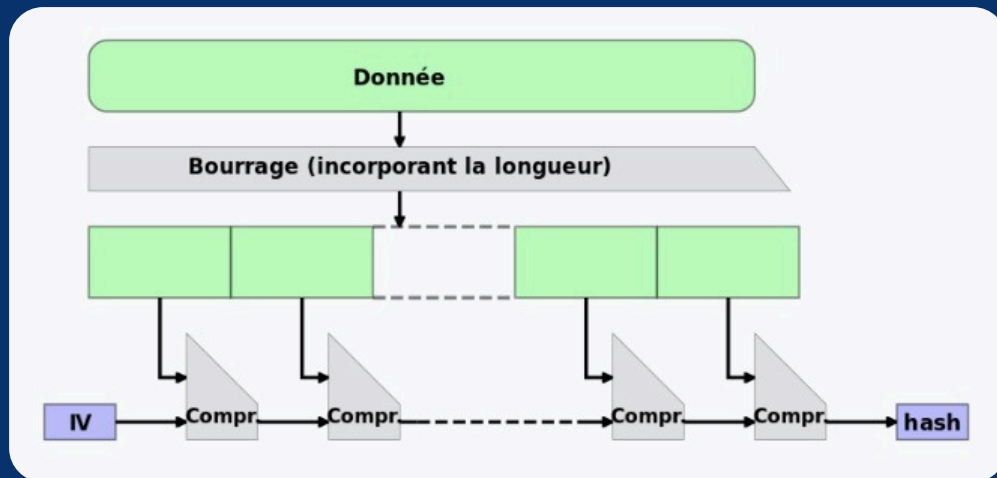
- "abc" en binaire (ASCII) : 01100001 01100010 01100011 (24 bits)

- Ajouter '1' : 01100001 01100010 01100011 1 (25 bits)

- Ajouter des '0' jusqu'à 448 bits : 01100001 01100010 01100011
10000000 00000000... (448 bits au total)

- Ajouter la longueur originale (24 bits en binaire) : 00000000...00011000
(64 bits)

Découpage



- Diviser le message complété en blocs de 512 bits. Chaque bloc sera traité séparément par l'algorithme.

Exemple :

Si le message complété est de 1024 bits, il sera divisé en 2 blocs de 512 bits chacun.

INITIALISATION DES VARIABLES DE HACHAGE

Objectif : Définir des valeurs initiales fixes pour commencer le processus de hachage.

Valeurs Initiales :

Les variables de hachage sont initialisées avec des valeurs spécifiques, définies par les spécifications SHA-256 :

plaintext

H0 = 6a09e667

H1 = bb67ae85

H2 = 3c6ef372

H3 = a54ff53a

H4 = 510e527f

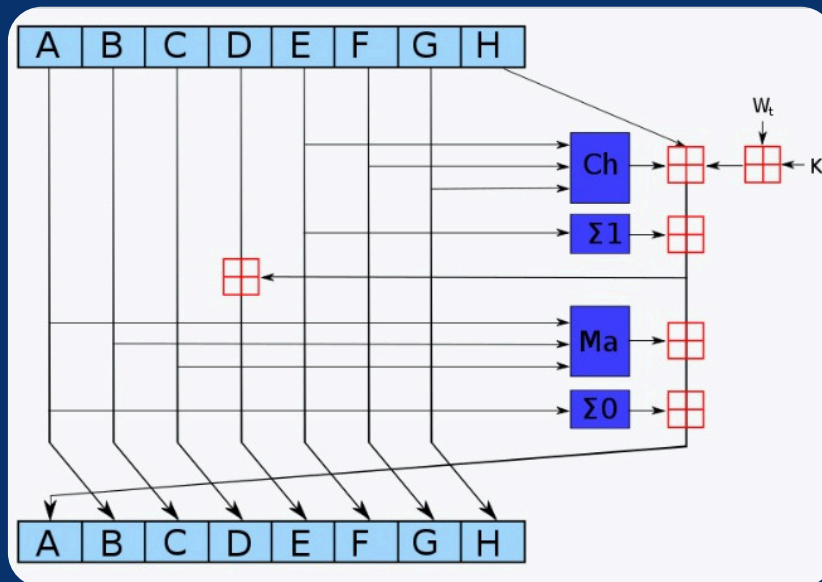
H5 = 9b05688c

H6 = 1f83d9ab

H7 = 5be0cd19

TRANSFORMATION DES BLOCS DE MESSAGE

Objectif : Appliquer une série de transformations à chaque bloc de 512 bits pour mélanger les bits du message.



Préparation des Mots de Message

1. Diviser chaque bloc de 512 bits* en 16 mots de 32 bits.
2. Étendre ces 16 mots en 64 mots* de 32 bits en utilisant les formules suivantes :

plaintext

$w[i] = \text{bloc}[i]$ pour $0 \leq i \leq 15$

$w[i] = \sigma_1(w[i-2]) + w[i-7] + \sigma_0(w[i-15]) + w[i-16]$ pour $16 \leq i \leq 63$

Fonctions :

$\sigma_0(x) = (x \gg 7) \oplus (x \gg 18) \oplus (x \gg 3)$

$\sigma_1(x) = (x \gg 17) \oplus (x \gg 19) \oplus (x \gg 10)$

Exemple :

Pour un bloc donné :

plaintext

w0 = mot initial 0

w1 = mot initial 1

...

w15 = mot initial 15

w16 = $\sigma_1(w_{14}) + w_9 + \sigma_0(w_1) + w_0$

...

Initialisation des Variables de Travail

1. Copier les valeurs des variables de hachage dans des variables temporaires :

plaintext

a = H0

b = H1

c = H2

d = H3

e = H4

f = H5

g = H6

h = H7

Boucle Principale

1. Pour chaque i de 0 à 63* :

plaintext

T1 = $h + \Sigma_1(e) + \text{Ch}(e, f, g) + K[i] + w[i]$

T2 = $\Sigma_0(a) + \text{Maj}(a, b, c)$

h = g

g = f

f = e

e = d + T1

d = c

c = b

b = a

a = T1 + T2

Fonctions Utilisées :

plaintext

$$\Sigma 0(x) = (x \gg 2) \oplus (x \gg 13) \oplus (x \gg 22)$$

$$\Sigma 1(x) = (x \gg 6) \oplus (x \gg 11) \oplus (x \gg 25)$$

$$\text{Ch}(x, y, z) = (x \& y) \oplus (\sim x \& z)$$

$$\text{Maj}(x, y, z) = (x \& y) \oplus (x \& z) \oplus (y \& z)$$

Constantes K :

$K[i]$ sont des constantes spécifiques pour chaque opération, définies par les spécifications SHA-256.

Exemple de Boucle Simplifiée:

pseudo

Pour chaque bloc de message :

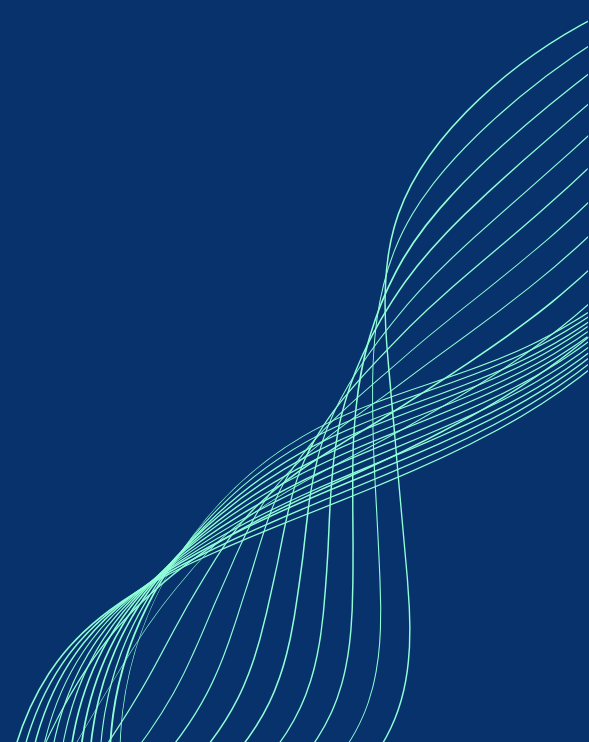
Initialiser a, b, c, d, e, f, g, h avec H0, H1, H2, H3, H4, H5, H6, H7

Pour i de 0 à 63 :

Calculer T1 et T2

Mettre à jour a, b, c, d, e, f, g, h

Mettre à jour H0, H1, H2, H3, H4, H5, H6, H7 avec a, b, c, d, e, f, g, h



MISE À JOUR DES VARIABLES DE HACHAGE

Objectif : Combiner les résultats de chaque bloc pour les variables de hachage.

Comment ? Ajouter les valeurs des variables temporaires aux variables de hachage :

plaintext

$H0 = H0 + a$

$H1 = H1 + b$

$H2 = H2 + c$

$H3 = H3 + d$

$H4 = H4 + e$

$H5 = H5 + f$

$H6 = H6 + g$

$H7 = H7 + h$

PRODUCTION DU HASH FINAL

Objectif : Combiner toutes les variables de hachage pour obtenir le condensat final de 256 bits.

Comment ? Concaténer les valeurs finales des variables de hachage :

plaintext

Explication

Lorsque vous utilisez un algorithme de hachage comme SHA-256 pour transformer un message en une valeur hachée (ou digest), l'algorithme passe par plusieurs étapes complexes. Mais une fois toutes ces étapes terminées, ce qu'il reste, ce sont les valeurs finales de plusieurs variables internes de l'algorithme.

Dans le cas de SHA-256, il y a huit variables de hachage internes qui sont mises à jour tout au long du processus. À la fin, ces huit variables contiennent chacune une partie du résultat final.

Pour obtenir le digest final (ou somme de contrôle), vous faites simplement ceci :

1. Obtenez les Valeurs Finales : Après que l'algorithme a terminé de traiter le message, chaque variable contient une valeur finale. Par exemple, imaginons que les valeurs finales des huit variables sont :

- H0 = 3d2e1f5a
- H1 = c0de23ab
- H2 = a4b5c6d7
- H3 = 89e1f2a3
- H4 = 456789ab
- H5 = 12c3d4e5
- H6 = f5e6d7c8
- H7 = 789abc12

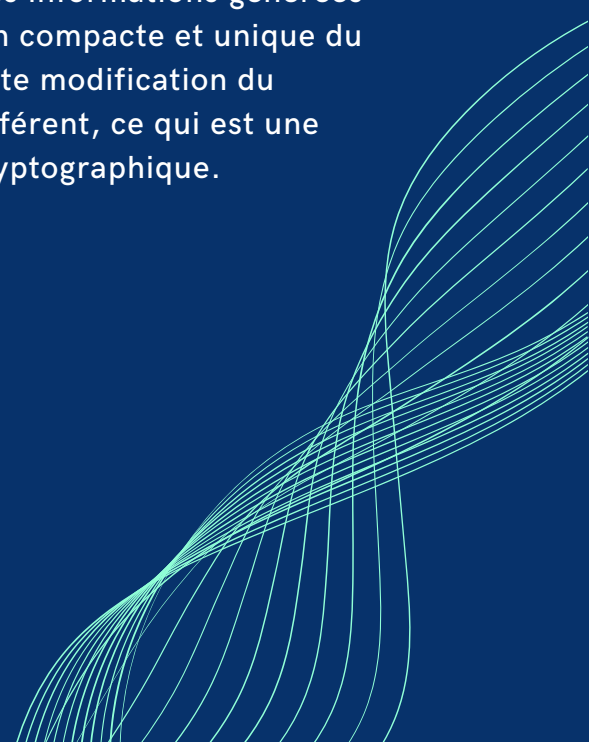
hash = H0 || H1 || H2 || H3 || H4 || H5 || H6 || H7

2. Concaténez les Valeurs: Vous prenez ces huit valeurs et les mettez bout à bout, dans l'ordre. Cela signifie que vous écrivez la valeur de H0, puis directement après celle de H1, puis celle de H2, et ainsi de suite jusqu'à H7. Dans notre exemple, cela donnerait :

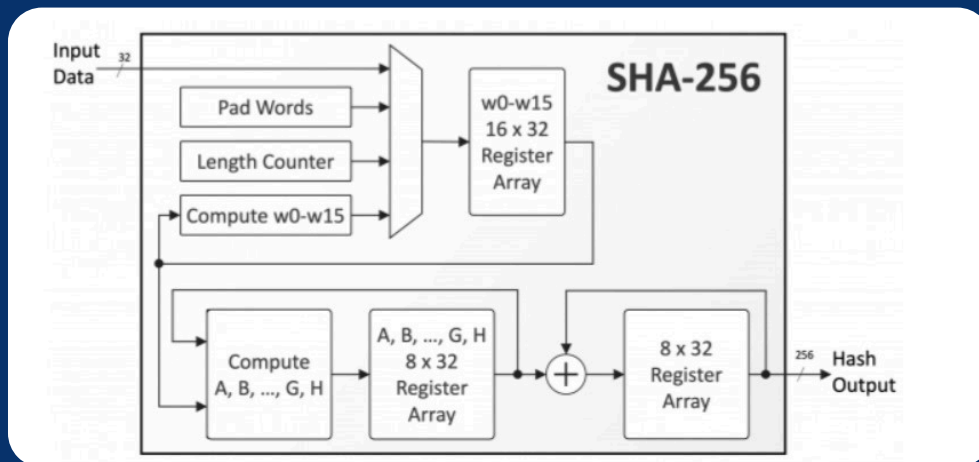
3. Résultat Final : La chaîne de caractères résultante, après la concaténation, est la valeur hachée finale (le digest) du message initial.

Pourquoi Concaténer ?

La concaténation des valeurs finales de ces variables de hachage internes est simplement une manière de rassembler toutes les informations générées par l'algorithme pour créer une seule représentation compacte et unique du message d'origine. Cela garantit que même une petite modification du message initial produira un digest complètement différent, ce qui est une propriété essentielle des algorithmes de hachage cryptographique.



RÉSUMÉ DES ETAPES



Prétraitement :

- Ajouter '1', compléter avec '0', ajouter la longueur du message.
- Diviser en blocs de 512 bits.

Initialisation

- Définir les variables de hachage initiales.

Transformation des Blocs

- Préparer les mots de message.
- Initialiser les variables de travail.
- Effectuer la boucle principale de transformation pour chaque mot.
- Mettre à jour les variables de hachage après chaque bloc.

Mise à Jour

- Combiner les valeurs des variables temporaires avec les variables de hachage.

Hash Finale

- Concaténer les variables de hachage pour obtenir le résultat final.

CONCEPTION DE L'ARCHITECTURE

DÉCOMPOSITION EN BLOCS

Blocs Fonctionnels Extraits

1. Module de Pré-Traitement

Le module de pré-traitement est responsable de la préparation du message avant qu'il ne soit transformé. Il effectue le padding du message pour s'assurer qu'il est aligné à une taille correcte (512 bits) pour l'algorithme SHA-256.

- Entity : pre_processing
- Ports :

 message_in : Le message d'entrée sous forme de vecteur de 512 bits.
 padded_message : Le message de sortie après padding, également un vecteur de 512 bits.

- Architecture :

Ce module ajoute des bits de padding au message d'entrée afin qu'il atteigne la taille nécessaire pour le traitement SHA-256. Le padding inclut l'ajout d'un bit '1' suivi de suffisamment de bits '0' et de la longueur du message original encodée sur 64 bits.

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

entity pre_processing is
    port(
        message_in : in std_logic_vector(0 to (16 * WORD_SIZE)-1); -- 512 bits
        padded_message : out std_logic_vector(0 to (16 * WORD_SIZE)-1) -- 512 bits
    );
end entity;

architecture behavior of pre_processing is
begin
    process(message_in)
    begin
        -- Padding the message (simplified example)
        padded_message <= message_in & "1" & (others => '0') &
            std_logic_vector(to_unsigned(len(message_in), 64));
    end process;
end architecture;
```

2. Module de Transformation

Le module de transformation réalise les calculs de hachage intermédiaires nécessaires pour SHA-256.

-
- Entity : transformation
- Ports :
 - clk : Signal d'horloge.
 - reset : Signal de réinitialisation.
 - message_block : Le bloc de message de 512 bits qui sera transformé.
 - hash_out : La valeur de hachage intermédiaire résultante de 256 bits.

- Architecture :

Ce module calcule les valeurs de hachage intermédiaires à partir du bloc de message. Les valeurs initiales des registres a, b, c, d, e, f, g, h sont mises à jour à chaque cycle d'horloge en fonction des opérations SHA-256 définies

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.sha_256_pkg.all;

entity transformation is
  port(
    clk : in std_logic;
    reset : in std_logic;
    message_block : in std_logic_vector(0 to (16 * WORD_SIZE)-1); -- 512 bits
    hash_out : out std_logic_vector((WORD_SIZE * 8)-1 downto 0) -- 256 bits
  );
end entity;

architecture behavior of transformation is
  -- Signal and constant declarations
  signal a, b, c, d, e, f, g, h : std_logic_vector(WORD_SIZE-1 downto 0);
  signal temp1, temp2 : std_logic_vector(WORD_SIZE-1 downto 0);
  constant k : array(0 to 63) of std_logic_vector(WORD_SIZE-1 downto 0) := (
    -- Initialization of constants
  );

  signal w : array(0 to 63) of std_logic_vector(WORD_SIZE-1 downto 0);
begin
  process(clk, reset)
  begin
    if reset = '1' then
      -- Initialization of hash values
    elsif rising_edge(clk) then
      -- Computation of intermediate hash values
    end if;
  end process;
end architecture;
```

3. Module de Post-Traitement

Le module de post-traitement combine les valeurs de hachage intermédiaires pour produire la valeur de hachage finale.

- Entity :post_processing
- Ports :
 - hash_in : La valeur de hachage intermédiaire de 256 bits en entrée.
 - final_hash : La valeur de hachage finale de 256 bits en sortie.
- Architecture :

Ce module reçoit les valeurs de hachage intermédiaires calculées par le module de transformation et les combine pour produire la valeur de hachage finale SHA-256.

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

entity post_processing is
    port(
        hash_in : in std_logic_vector((WORD_SIZE * 8)-1 downto 0); -- 256 bits
        final_hash : out std_logic_vector((WORD_SIZE * 8)-1 downto 0) -- 256 bits
    );
end entity;

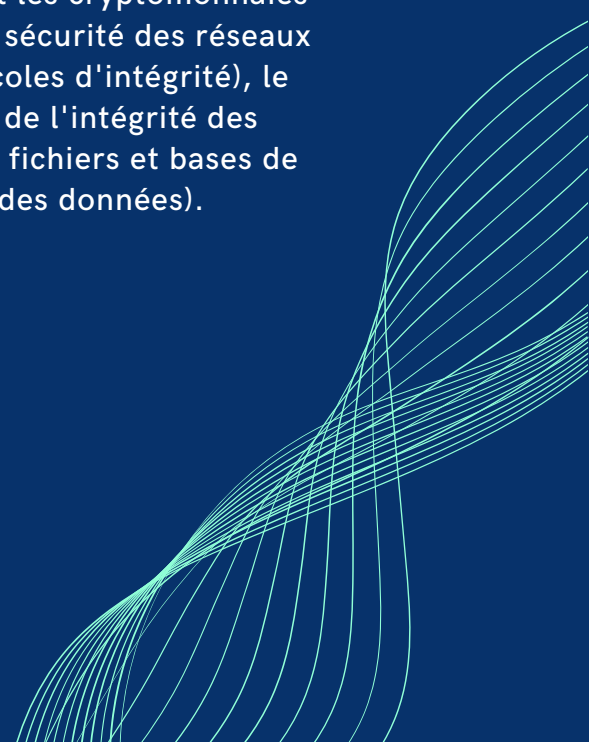
architecture behavior of post_processing is
begin
    process(hash_in)
    begin
        -- Combine intermediate hash values to produce the final hash
        final_hash <= hash_in;
    end process;
end architecture;
```


IMPORTANCE DE SHA ET DOMAINES D'APPLICATION

SHA (Secure Hash Algorithm) est une famille de fonctions de hachage cryptographiques largement utilisées pour garantir l'intégrité et la sécurité des données. Les fonctions de hachage SHA produisent un condensé unique à partir de données d'entrée. Toute modification des données d'origine, même minime, se traduit par un changement significatif dans le hash, permettant ainsi de vérifier l'intégrité des données. Dans les protocoles de sécurité, les hachages sont utilisés pour authentifier les messages. Par exemple, les HMAC (Hash-based Message Authentication Codes) utilisent des hachages pour assurer l'authenticité et l'intégrité des messages échangés.

Les fonctions de hachage cryptographique sont couramment utilisées pour stocker les mots de passe de manière sécurisée. Les mots de passe sont hachés avant d'être stockés, et la vérification se fait en comparant les hachages plutôt que les mots de passe en clair. De plus, les signatures numériques utilisent des hachages pour garantir que le contenu d'un message ou d'un document n'a pas été altéré. Le hash du message est chiffré avec une clé privée pour créer une signature qui peut être vérifiée par d'autres. Enfin, les algorithmes SHA sont fondamentaux pour la sécurité et l'intégrité des blockchains. Chaque bloc contient un hash du bloc précédent, assurant une chaîne immuable et sécurisée.

Les fonctions SHA sont utilisées dans divers domaines, notamment la cryptographie (protocoles de sécurité SSL/TLS, signatures numériques, et infrastructure à clés publiques PKI), la blockchain et les cryptomonnaies (pour le minage et la validation des transactions), la sécurité des réseaux (HMAC pour l'authentification des messages, protocoles d'intégrité), le stockage et la transmission de données (vérification de l'intégrité des fichiers téléchargés ou transmis), et les systèmes de fichiers et bases de données (déduplication et vérification de l'intégrité des données).



DÉFIS

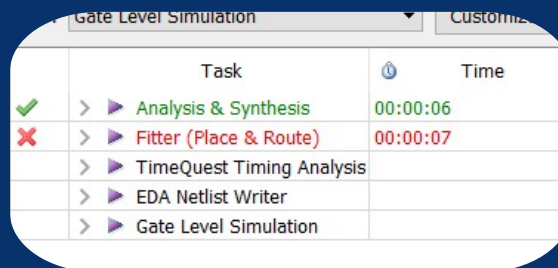
Implémenter l'algorithme SHA-256 en VHDL présente plusieurs défis. Premièrement, SHA-256 est un algorithme complexe avec de nombreuses étapes de transformation et de mélange, impliquant des opérations logiques, des additions, des rotations, et des permutations non linéaires. La gestion des états constitue un autre défi. L'algorithme doit gérer plusieurs états internes, incluant la lecture des blocs de message, les transformations intermédiaires, et l'assemblage du hash final. Cela nécessite une machine à états bien définie pour contrôler le flux de données.

Pour être utile en pratique, l'implémentation doit être efficace en termes de temps et de ressources matérielles. Cela implique des optimisations au niveau du pipeline, de la latence et de l'utilisation des ressources FPGA/ASIC. La précision et la fiabilité sont également cruciales. Les erreurs dans l'implémentation peuvent avoir des conséquences graves sur la sécurité. Chaque étape doit être correctement vérifiée et testée pour assurer la conformité avec la spécification SHA-256.

Exploiter le parallélisme matériel pour accélérer les calculs sans compromettre l'intégrité des données est un défi supplémentaire. Enfin, la simulation et le débogage des circuits numériques peuvent être complexes, surtout pour un algorithme comme SHA-256 qui manipule de grandes quantités de données et passe par de nombreuses transformations internes.

Difficulté de simulation

Lors de la mise en œuvre de l'algorithme SHA-256 en VHDL, une des principales difficultés rencontrées a été la simulation du code dans le logiciel Quartus.



	Task	Time
✓	> Analysis & Synthesis	00:00:06
✗	> Fitter (Place & Route)	00:00:07
	> TimeQuest Timing Analysis	
	> EDA Netlist Writer	
	> Gate Level Simulation	

```
✗ 169281 There are 546 IO input pads in the design, but only 492 IO input pad locations available on the device.  
171121 Fitter preparation operations ending: elapsed time is 00:00:04  
✗ 171000 Can't fit design in device
```

En effet, les dispositifs disponibles dans Quartus ne pouvaient pas accueillir la complexité et la taille de l'algorithme SHA-256, empêchant ainsi une simulation réussie. Cette limitation met en évidence les exigences élevées en termes de ressources matérielles nécessaires pour exécuter des algorithmes cryptographiques complexes et souligne l'importance de choisir des dispositifs adaptés pour des implémentations matérielles de cette envergure.

CONCLUSION

L'implémentation de l'algorithme SHA-256 en VHDL représente un défi considérable tant en termes de calculs que d'architecture matérielle. SHA-256, en tant qu'algorithme de hachage cryptographique, joue un rôle crucial dans la sécurité des données, garantissant l'intégrité et l'authenticité des informations dans diverses applications allant de la cryptographie et des signatures numériques à la blockchain et à la sécurité des réseaux.

L'algorithme SHA-256 comporte plusieurs étapes de transformation et de mélange impliquant des opérations logiques, des additions, des rotations et des permutations non linéaires. La mise en œuvre de ces étapes en VHDL doit être soigneusement planifiée pour garantir l'efficacité et la fiabilité du processus de hachage. Les défis de l'implémentation incluent la gestion des états internes de l'algorithme, la synchronisation des opérations à l'aide de signaux d'horloge et l'optimisation de l'utilisation des ressources FPGA ou ASIC.

Chaque module, du pré-traitement au post-traitement, doit être conçu de manière modulaire pour permettre des tests et des vérifications indépendants, assurant ainsi la conformité avec les spécifications du SHA-256. Les calculs complexes et la gestion précise des états ajoutent à la difficulté de l'implémentation, exigeant une profonde compréhension de l'algorithme et une attention minutieuse aux détails.

Malgré la complexité inhérente à l'algorithme et les défis de son implémentation matérielle, réussir à le traduire en VHDL offre des avantages significatifs. Cela permet de créer des systèmes de hachage efficaces, sécurisés et capables de fonctionner à des vitesses élevées, essentielles pour les applications modernes. En conclusion, la combinaison de l'algorithme SHA-256 et de VHDL démontre la puissance de l'ingénierie numérique dans la construction de systèmes de sécurité avancés, tout en soulignant la nécessité de maîtriser à la fois les aspects théoriques et pratiques de la conception matérielle.

