Tristan Marie
Théo Sussat
Corentin Guerard
Julien Chichery

Projet avancé en systèmes embarqués:

Développement d'une solution de minage de Bitcoin sur FPGA

SOMMAIRE

1- Introduction 2

2- Bitcoin / Blockchain 3

3- Principe du minage de Bitcoin 5

4- Principe de fonctionnement du SHA-256 7

5- Architecture du SHA-256 utilisé pour le minage de Bitcoin 10

6- Optimisation sous Vivado HLS 12

7- Module de vérification SHA256 14

8- Architecture du projet 16

9- Optimisation / Travail futur 19

10- Conclusion 20

Bibliographie 21

Annexe: Code SHA-256 23

1- Introduction

Cela fait maintenant plusieurs années que les crypto monnaies ont vu le jour et malgré le crash de 2018 qui a suivi leur croissance exponentielle fin 2017, le milieu est toujours très actif.

Une crypto monnaie est une ressource numérique utilisée comme moyen d'échange de la même manière qu'une devise. Seulement, à la différence d'une devise classique, les crypto monnaies ne sont pas gérées par des banques, leur contrôle est décentralisé et c'est chaque acteur du réseau qui participe à la sécurité des transactions.
Cette participation passe par le minage. Cette tâche consiste en la résolution d'un problème cryptographique visant à trouver une clé qui vérifie une condition sur le hachage des transactions.
Le minage est intéressant car les acteurs participants à la résolution de ce problème sont récompensés. Cela signifie que des machines dédiées au minage constant permettent d'être rémunéré pour son don de puissance de calcul.
L'activité de minage de crypto-monnaie est un travail de calcul intensif et exige beaucoup de puissance de traitement et de temps. La majorité de l'algorithme de calcul mis en jeu est ici le hachage de type SHA-256. Ces algorithmes sont actuellement implantés sur de l'ASIC ou sur des solutions à base de CPU/GPU. Cependant le minage sur FPGA semble être un bon compromis sur le long terme en ce qui concerne l'efficacité énergétique et calculatoire.
L'objectif de ce projet est de concevoir un accélérateur matériel capable de miner du Bitcoin. Cet accélérateur matériel sera donc une carte FPGA et le travail d'implantation se fera avec la synthèse de haut niveau (HLS) .

De ces objectifs nous avons tiré le cahier des charges suivant :

- Importer et optimiser un bloc SHA-256 en SystemC
- Développer une architecture capable de miner
- Vérifier le comportement du SHA-256 et de l'architecture globale
- Implémenter le design sur FPGA
- Mettre en place une architecture de blockchain
- Optimisation des performances et parallélisation

2- Bitcoin / Blockchain

Avant de développer notre solution de minage, nous avons dû nous renseigner sur le fonctionnement du Bitcoin.
Le réseau de Bitcoin se repose sur deux principes: l'architecture de blockchain et le hash. Dans un premier temps, nous allons nous intéresser à l'architecture d'une blockchain. Le hash quant à lui est un algorithme cryptographique qui sera expliqué dans le paragraphe suivant.

Comme son nom l'indique, une blockchain est une chaîne de blocs. Ces blocs sont composées de transactions du réseau Bitcoin qui doivent être validées. La particularité de ces blocs est la façon dont ils sont liés. En effet, afin d'être validé, chaque bloc prend en paramètre le hash du bloc précédent. De ce fait, chaque bloc dépend l'entièreté des blocs qui le précède. Il devient alors impossible pour quelqu'un qui voudrait falsifier une transaction antérieure car il faudrait altérer la totalité des blocs qui dépendent de cette transaction.

On peut observer un aperçu d'un morceau d'une blockchain dans la figure suivante:

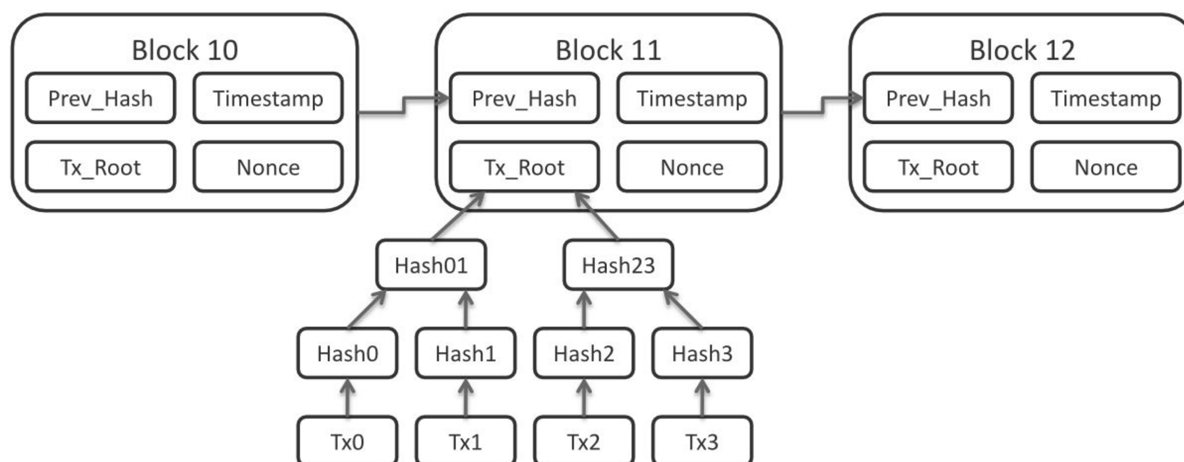

figure 1: Représentation de l'architecture de blockchain

Comme on peut le voir un bloc se compose de plusieurs items:

- Prev_Hash est le hash précédent injecté pour assurer la dépendance et la sécurité du réseau.

- La Tx_Root ou merkle root (racine des transactions) est le résultat du hash successif des transactions comprises dans le bloc.

- La Timestamp est une référence de temps qui indique le début des calculs

- La nonce (Number used ONCE) est le seul élément variable du bloc. C'est ce qui permet d'itérer le minage (en itérant la nonce) jusqu'à l'obtention d'une solution.

- La difficulté représente la complexité du problème à résoudre pour valider le bloc. Elle change en fonction de la puissance de calcul disponible sur le réseau pour que la durée de résolution d'un bloc soit contrôlée et plus ou moins constante.

- Le bloc comprend aussi la version du protocole Bitcoin afin qu'il soit interprété correctement.

Tous ces composants sont donc compris dans le bloc et c'est ces derniers qui servent de message d'entrée au problème de hash à résoudre.

Maintenant que le concept de blockchain est plus clair, il est temps de passer à l'explication de la fonction de hash.

3- Principe du minage de Bitcoin

Comme annoncé précédemment, le second pilier du réseau Bitcoin est sa fonction de hash (ou de hachage en français).

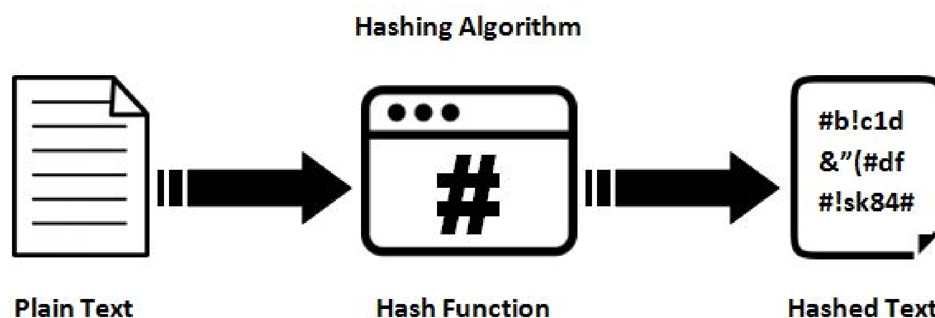

figure 2: Principe des fonctions de hachage

Comme l'illustre la figure ci-dessus, une fonction de hash prend en entrée des données de tailles arbitraire et génère une signature unique à partir du message d'entrée et de taille fixe. L'unicité de cette signature assure que la fonction ne peut pas être inversée et que les données d'origine ne peuvent pas être découvertes. Ce type de fonction est notamment utilisé pour reconnaître des mots de passe sans les avoir besoin de leur passage en clair.

Cet algorithme cryptographique est particulièrement intéressant pour les crypto monnaies et le Bitcoin. En effet, la signature de sortie étant unique, un léger changement mal intentionné provoquerait un changement total de la signature de sortie. Cela rendrait la tâche, déjà extrêmement complexe, de falsifier des transactions encore plus impossible. De plus les données étant cryptées, l'anonymat des utilisateurs est respecté. Cet anonymat n'est en fait qu'un pseudonymat car on fait référence aux utilisateurs par des adresses qui font office de pseudonymes et les transactions peuvent êtres observées en se connectant au réseau Bitcoin.

Dans le cas de Bitcoin, l'algorithme de hash utilisé est le SHA-256 (Simple Hashing Algorithm 256 bits). Mais avant de détailler son fonctionnement et notre travail sur son implémentation, il reste à expliquer comment le minage a lieu.
Le minage est une tâche lourde calculatoirement et donc longue dans le temps. Son exécution peut être illustrée par la figure suivante:

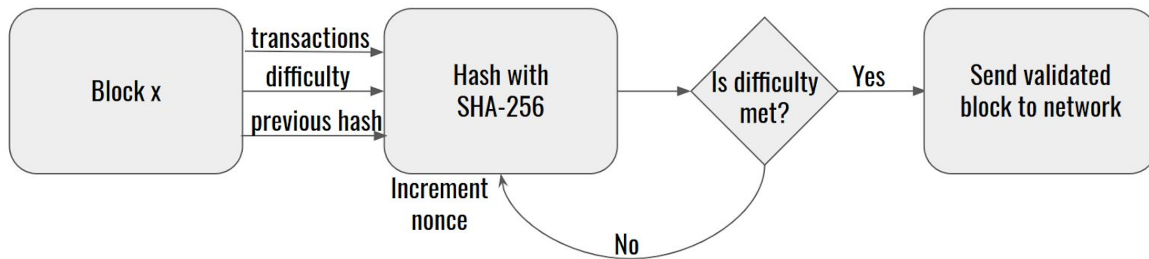

figure 3 : Schéma de principe du minage

Le minage d'un bloc se déroule de la manière suivante :

- Le mineur reçoit un bloc contenant tous les paramètres du bloc
  Il reçoit en particulier les transactions, le hash précédent et la difficulté.
  La difficulté est une valeur maximale que le hash ne doit pas dépasser pour que le bloc soit validé.
- Une fois le bloc reçu, il est haché selon le protocole que le Bitcoin défini. Si le résultat n'est pas inférieur à la difficulté demandée, le mineur recommence la procédure en incrémentant sa nonce jusqu'à qu'une solution soit trouvée ou qu'un nouveau bloc soit reçu.
- Quand une solution est trouvée, elle est envoyée au réseau qui la vérifie facilement comme un seul hash est nécessaire.
- Le bloc est alors validé et les mineurs sont récompensés pour leurs participation au minage

Maintenant que les principes du minage de crypto monnaies sont élucidés, nous pouvons passer au développement de notre solution de minage. Comme la fonction de hachage est le coeur de la complexité d'un mineur, il a fallu s'intéresser tout d'abord à développer un module capable de hacher des données correctement, c'est à dire avec l'algorithme SHA-256 comme le réseau Bitcoin l'a défini.

4- Principe de fonctionnement du SHA-256

Le SHA-256 est un algorithme de hash basé sur la construction de Merkle-Damgård (figure 4). Cette construction permet de traiter un message de taille quelconque en entrée et de fournir en sortie un hash de taille fixe. Elle satisfait aussi les contraintes de sécurité suivante:

- Résistance aux collisions (il est difficile de trouver 2 messages distincts qui donnent le même hash)
- Résistance aux préimages (il est également difficile de trouver le message à l'origine d'un hash et également d'en construire d'autres qui fournissent ce même hash)

La construction de Merkle-Damgård consiste à utiliser une fonction de compression f qui doit être initialisé et capable de traiter des données de taille fixe. Dans le cas du SHA-256 cette fonction prend en entrée 512 bits et donne en sortie 256 bits. Pour traiter un message avec cette fonction, on commence par diviser le message en bloc de 512 bits. On effectue ensuite une opération de padding sur le dernier bloc pour qu'il est la bonne taille. Ce padding consiste à coder la longueur du message à la fin du bloc et à effectuer un bourrage avec des zéros.

- Entrée: taille variable sortie : 256 bits
- Fonction de hachage : entrée 512 bits sortie 256 bits

Une fois cette étape réalisée, la fonction de compression est itérée pour traiter tous les blocs du message. Cette itération se réalise de la façon suivante: la fonction de compression s'initialise en créant un vecteur d'initialisation. Ensuite, un bloc à traiter est transmis à la fonction f qui va ainsi retourner un hash. Cela permet d'initialiser la fonction de compression pour le traitement du prochain bloc.
Une fois tous les messages traités, la sortie contient le hash final.

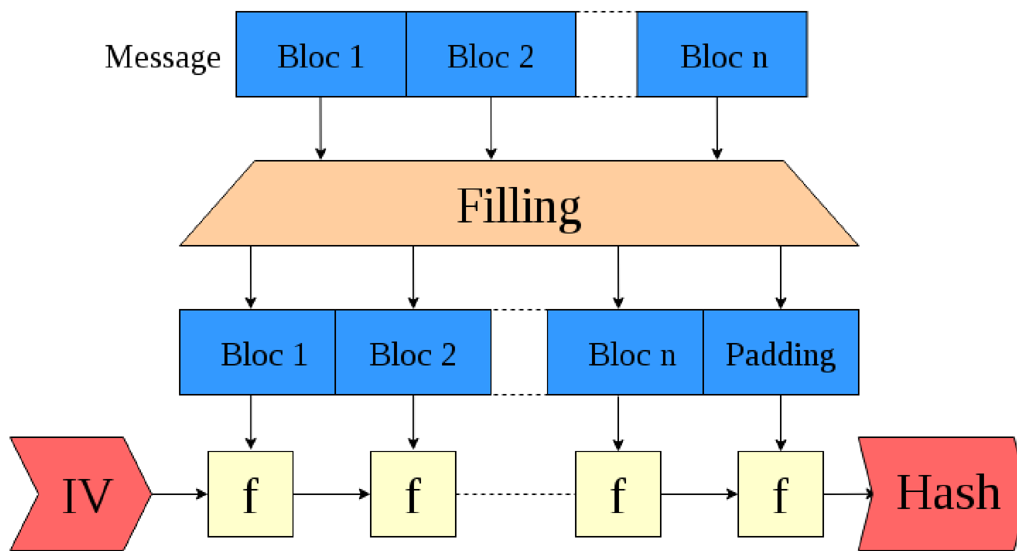

figure 4: construction de Merkle-Damgård

La fonction de compression utilisé par le SHA-256 est basée sur des registres à décalage et des opérations bits à bits. Ces registres nommés de A à H contiennent chacun un mot de 32 bits. Les opérations et l'architecture logique de la fonction sont représentées sur la figure 5 suivante:

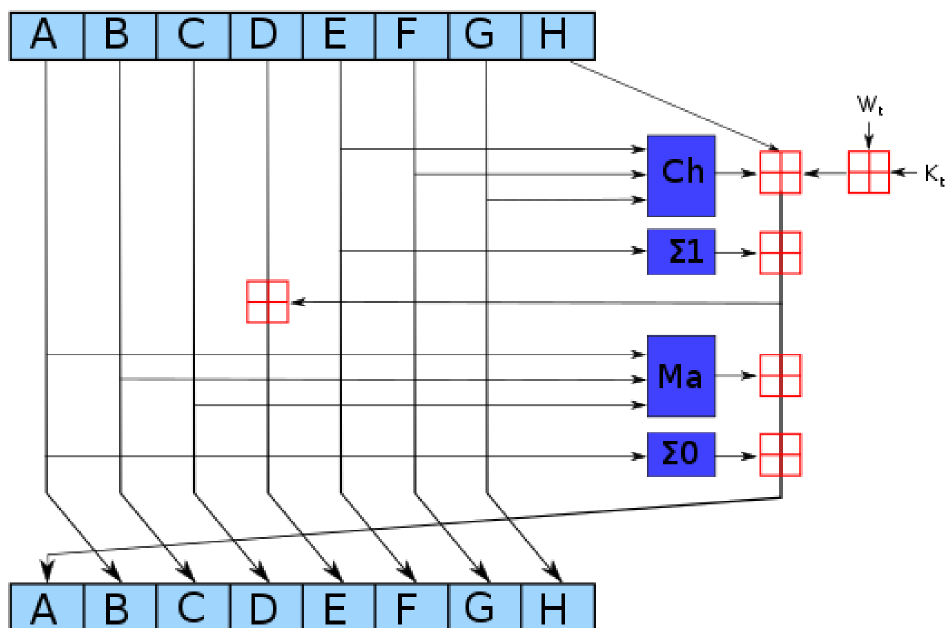

figure 5: Architecture de la fonction de compression du SHA-256

-fonction de hash : registre à décalage + opérations logique sur les bits

➔ Implémentation simple et efficace en logique câblée

Sur la figure ci-dessus, le message à traité, représenté par Kt et Wt, est un message constant. Le vecteur d'initialisation permet d'affecter la valeur initiale aux registres à décalage.

De plus, cette fonction travaille seulement sur des mots de 32 bits et utilise les 6 fonctions logiques suivantes:

- ⊞ est l'addition modulo 2^{32}. D'un point de vue matériel, cela consiste à un additionneur 32 bits qui ne gère pas la retenue.
- CH(x,y,z)
- Ma(x,y,z)
- Σ 0(x)
- Σ 1(x)
- SHRn est le décalage logique de n bits à droite et ROTRn la rotation de 2 bits à droite

$$Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z)$$

$$Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

$$\Sigma_0^{\{256\}}(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$$

$$\Sigma_1^{\{256\}}(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$$

$$\sigma_0^{\{256\}}(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$$

$$\sigma_1^{\{256\}}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$$

Toutes ces opérations logiques s'implémentent facilement et efficacement en logique câblée. C'est pourquoi nous avons choisi de réaliser les calculs de hash sur FPGA plutôt que sur CPU.

Enfin, les opérations de la figure 5 doivent être effectuées 64 fois pour obtenir dans les registres (A à H) la sortie de la fonction de compression.

Nous venons de voir le fonctionnement de la fonction de hachage SHA-256. Nous allons aussi utiliser les particularités du minage de Bitcoin pour optimiser notre architecture de hachage.

5- Architecture du SHA-256 utilisé pour le minage de Bitcoin

Le SHA-256 utilisé pour le minage de Bitcoin nécessite un format d'entrée avec un header de 640 bits. Ces 640 bits se décomposent en 6 parties dans l'ordre suivant:

- 4 octets: Informations sur la version
- 32 octets: Hash du bloc précédent
- 32 octets: Le merkle_root
- 4 octets: Référence de temps lors de la création du bloc
- 4 octets: Difficulté cible
- 4 octets: La nonce

Le header est ensuite suivi du padding comme nous pouvons le voir sur le schéma ci-dessous:

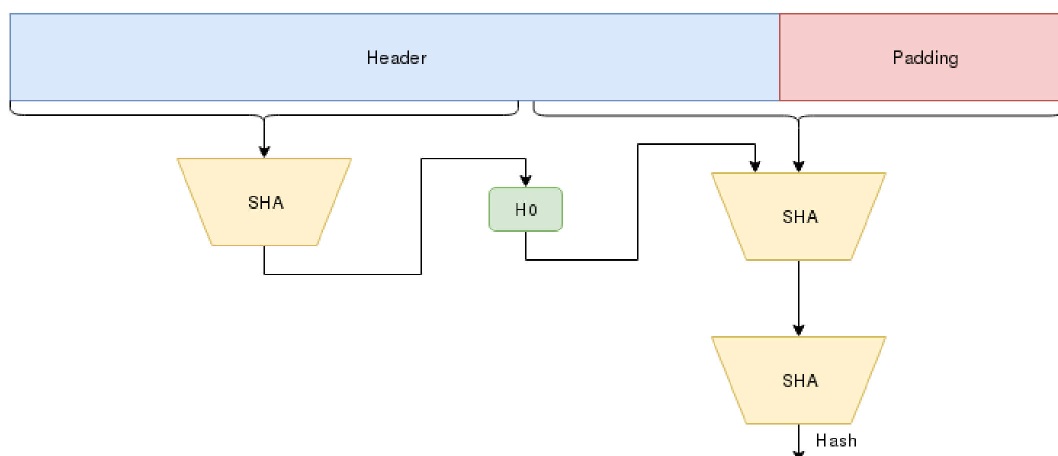

figure 6: Structure du message d'entrée

On peut remarquer sur la figure ci-dessus que le minage de Bitcoin nécessite d'utiliser 3 fois la fonction de compression. Cependant, le temps passé à hacher un même header avec différentes nonce est important. Les 512 premiers bits du header seront donc constant lors de l'itération de la nonce. Nous avons alors choisi d'introduire un élément de mémorisation du hash temporaire (H0), ce qui permet de réduire la complexité du calcul à 2 utilisations de la fonction de compression.

Cette démarche est pertinente car la fonction de compression est la partie de l'architecture la plus complexe et la plus coûteuse en terme de temps de calcul.

A partir des spécification du minage de Bitcoin et les remarques ci -dessus l'architecture du SHA256 suivante (figure 7) a pu être obtenu:

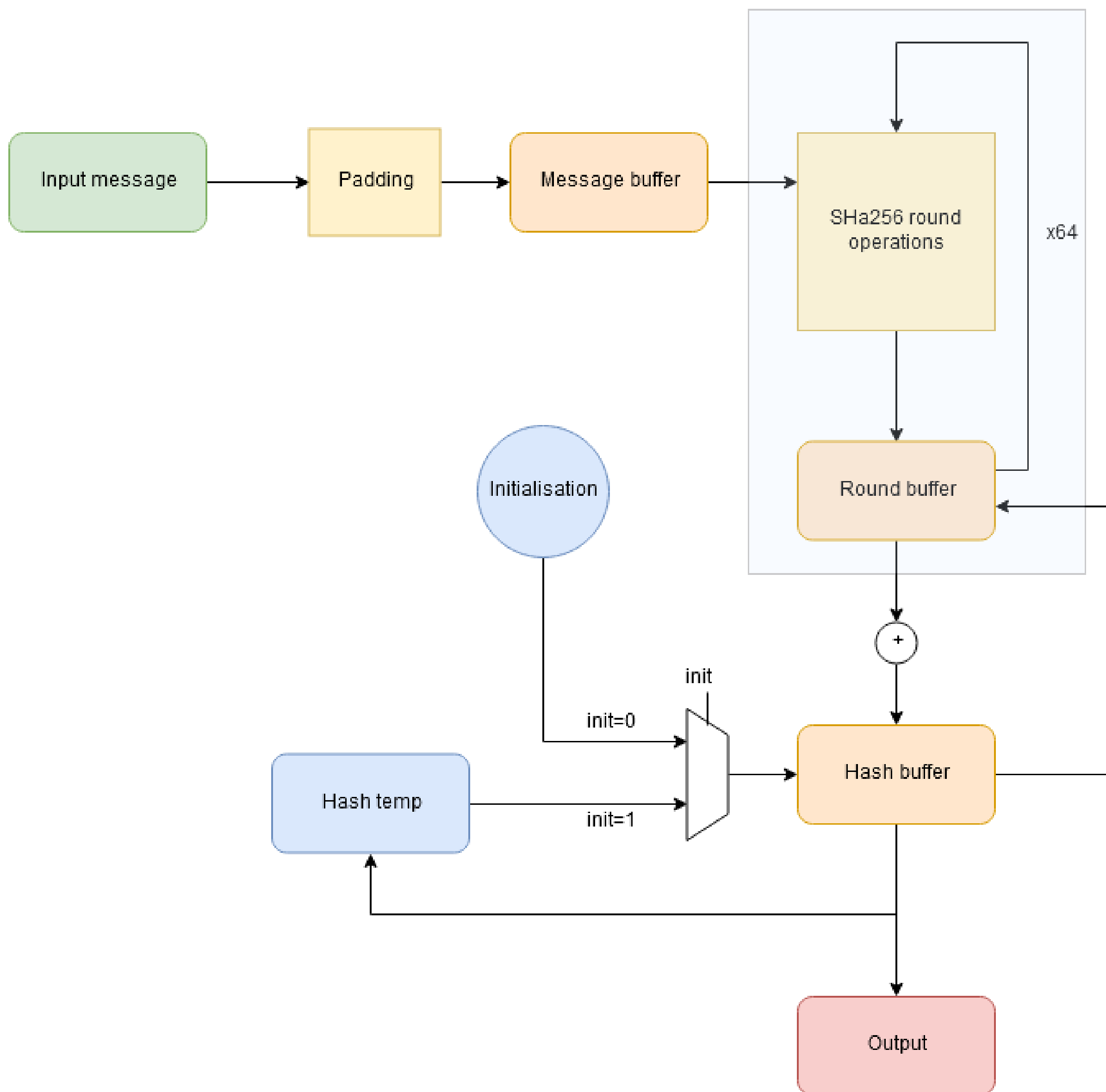

figure 7: Architecture du bloc SHA256

La description de cette architecture a été réalisée en systemC. Dans un premier temps, des fonctions implémentant un SHA-256 réalisées par Brett Nicholas ont été réutilisées afin d'obtenir un premier prototype fonctionnel. Ces fonctions sont génériques et permettent de traiter des messages de taille quelconque. Par la suite, le bloc SHA-256 a été optimisé spécialement pour le minage de Bitcoin. Cette

optimisation s'est traduite par la suppression ou la simplification des fonctions qui gèrent le traitement du message d'entrée. Par la suite, les interfaces ont été redéfinies pour utiliser plutôt des **unsigned int** au lieu d'**unsigned char** car la fonction de compression travail sur des mots de 32 bits.

6- Optimisation sous Vivado HLS

Une fois le fonctionnement du bloc validé, l'optimisation de ce bloc a pu être envisagée. Le point du programme entraînant le plus de latence a été rapidement identifié et se situe dans la fonction de compression (sha256_transform dans le code ci-dessous). Cette fonction introduit beaucoup de latence car elle comporte plusieurs boucles for comportant 64 itérations. De base Vivado HLS synthétise l'architecture matérielle effectuant les calculs à l'intérieur d'une boucle et ajoute des contrôles autours afin de pouvoir itérer. Dans notre cas, la boucle for introduit donc 64 fois la latence des calculs situés dans la boucle.
Pour réduire cette latence et ainsi améliorer les performances du débit, on peut soit optimiser les calculs effectués dans les boucles, soit essayer de paralléliser ces calculs lorsque cela est possible.
Dans la suite du projet toutes les boucles for dont les itérations sont indépendantes ont été déroulées avec le pragma #pragma HLS UNROLL.

Finalement, il reste 2 boucle for (voir code en annexe) qui possèdent une interdépendance des itérations. L'impact des différentes directives d'optimisations sur ces boucles a ensuite pu être étudié.

Les résultats d'implémentations de Vivado HLS ont été consignés dans le tableau suivant figure 8. Dans un premier temps l'architecture générique a été synthétisée , puis celle optimisée pour le Bitcoin a été implémenté pour comparer les performances. Puis, pour cette dernière les boucles for 1 et 2 ont été déroulées pour réduire la latence du bloc SHA-256. Enfin, la latence maximum correspond au cas où on traite un nouveau header (on réalise alors 3 fois la compression) et la latence minimum correspond au cas où on traite seulement une nouvelle nonce (on effectue la compression seulement 2 fois).

	Generic Architecture	Optimised Architecture	Fully Unroll	Unroll Factor =2	Pipelined
Latence min (cycle)	988	802	462	704	460
Latence max	1472	1204	694	1056	691
Luts	4145	3860	23305	4359	4194
FF	3813	2982	17242	3195	3135
Chemin Critique (ns)	6,473	6,473	8,043	6,473	7,743
Fréquence (MHz)	154,4878727	154,4878727	124,331717	154,4878727	129,1489087
Débit max (ksha/s)	156,3642436	192,6282702	269,1162706	219,443001	280,7584972
Débit min (ksha/s)	104,9510005	128,3121866	179,15233	146,295334	186,9014598

figure 8: Résultats d’implémentations de Vivado HLS

Les pragmas `#pragama HLS UNROLL` et `#pragama HLS UNROLL FACTOR=2` ont été utilisés seulement sur la boucle 1. Le pipeline a lui été appliqué aux 2 boucles. Le fait de dérouler les boucles permet à l’outils de synthèse de paralléliser le plus de calculs possibles. Cependant, cela coûte beaucoup de ressources. En effet, le nombre de LUTs et de FFs utilisé est multiplié par plus de 5.
Le pragma unroll factor=2 permet de dérouler seulement les opérations indépendantes de la boucle 1. On obtient ainsi un gain en latence notable, tout en limitant les ressources utilisées.

Finalement, on obtient de très bonnes performances, pour un coût supplémentaire en ressources réduit avec des pipelines. Cependant, il faut prendre ces chiffres avec précaution car l’outil de synthèse peut avoir mal évalué les dépendances entre les résultats des itérations. Ainsi, il faudra tester l’architecture implémentée sur carte.

7- Module de vérification SHA256

Dans notre architecture de blockchain nous ne vérifions pas le bon fonctionnement ou les résultats du SHA-256 pour ne pas perdre de temps et ainsi gagner en performance.
Il est donc impératif que notre SHA-256 implémenté sur la carte FPGA nous renvoie un résultat correct pour n'importe quelles valeurs qui lui sont envoyées.

Un module en SystemC a donc été développé afin de vérifier le bon fonctionnement du bloc SHA-256. Ce module de test n'est pas utilisé dans l'architecture finale, il est seulement utilisé pour le développement du SHA-256.

Le schéma du fonctionnement de ce module est détaillé ci-dessous figure 9:

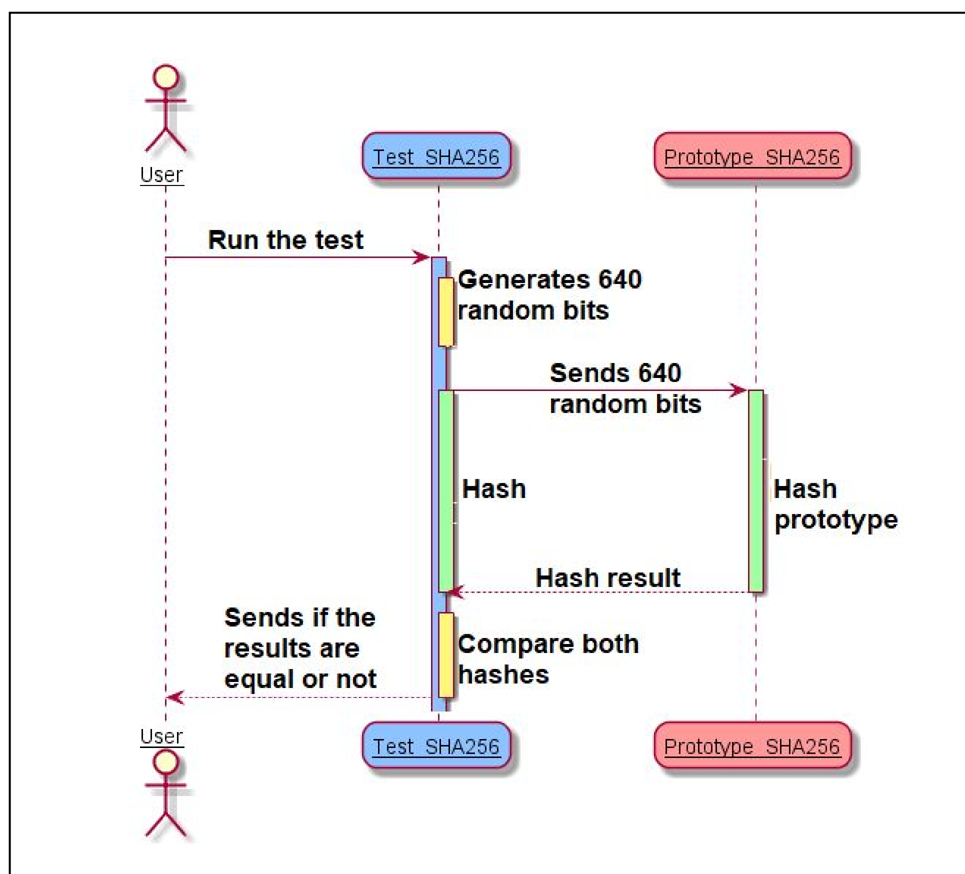

figure 9: Diagramme de séquence du module Test_SHA256

Ce module de test peut se diviser en trois parties distinctes:

- La génération et l'envoi de 640 bits aléatoires au prototype
- Le hachage des bits pour obtenir le bon résultat
- La réception du hachage du prototype pour le comparer au bon résultat

La première partie est une génération de 20 **int**, soit 640 bits correspondant au header. Ces bits sont ensuite envoyés dans une FIFO reliée au prototype de SHA-256 afin qu'il puisse hacher ce header.

En seconde partie, le module de test effectue le hachage du header en parallèle du hachage par le prototype. Le SHA-256 utilisé dans ce module provient de la bibliothèque OpenSSL dont nous somme sûr du bon fonctionnement et du résultat. Ce résultat est ainsi utilisé comme valeur de référence par la suite. Bien que ce SHA-256 soit fonctionnel, il n'est pas utilisé dans l'architecture globale car il ne serait pas optimisé pour une implémentation sur carte FPGA.

Dans la dernière partie, les deux résultats obtenues sont comparés. Si ces deux hachages sont égaux, le prototype de SHA-256 est validé. Cependant, si le prototype ne renvoie pas de résultats ou renvoie un résultat différent de celui obtenu grâce à la bibliothèque OpenSSL, cela indique qu'il y a alors une erreur dans le code du prototype.

8- Architecture du projet

Le développement du bloc SHA-256 terminé, ce bloc pouvait être ajouter à une architecture de minage de Bitcoin. L'architecture de minage choisie est présentée figure 10:

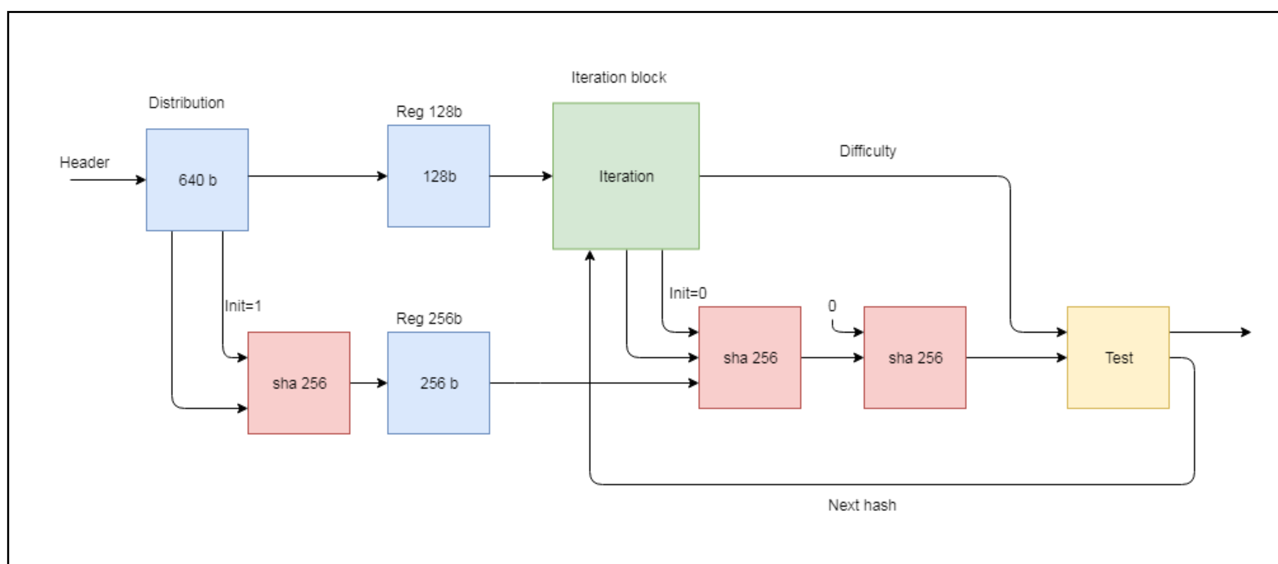

figure 10: Architecture du bloc de minage

Celle-ci reçoit le header de 640 bits contenant les différentes informations requises pour réaliser le minage du Bitcoin. Ce header est divisé par le bloc distribution qui transmets les 512 premiers bits au bloc SHA-256 et les 128 bits restants au registre associé. Un signal d'initialisation est envoyé au bloc SHA-256 pour lui signaler que ce premier message sert à fabriquer le vecteur d'initialisation. Le SHA256 va lui aussi stocké ce vecteur d'initialisation dans un registre. En pratique ces blocs de registres sont compris dans le bloc SHA256 et d'itération.

Après l'initialisation réalisée, le bloc d'itération permet de contrôler le minage effectif. Celui-ci reçoit les 128 bits précédemment évoqués qu'il va compléter avec la nonce ainsi qu'un padding de 0 pour créer un message de 256 bits. Ce message est envoyé au bloc SHA-256 avec le signal de contrôle d'initialisation à 0 cette fois car celle -ci a déjà été effectuée. Le bloc SHA-256 va alors effectuer deux SHA-256 consécutifs comme le stipule le protocole du Bitcoin.

Enfin, le bloc de test va alors récupérer le résultat issu du SHA256 ainsi que la difficulté précédemment extraite par le bloc Itération à partir du Header. Le bloc test

va alors comparer si la difficulté, c'est à dire le nombre de 0 au début du hash obtenu est suffisant pour valider ce bloc. Si c'est le cas, le hash est envoyé sur la sorti, sinon la un signal Next_hash est transmis pour incrémenter la nonce et réitérer les étapes détaillées précédemment.

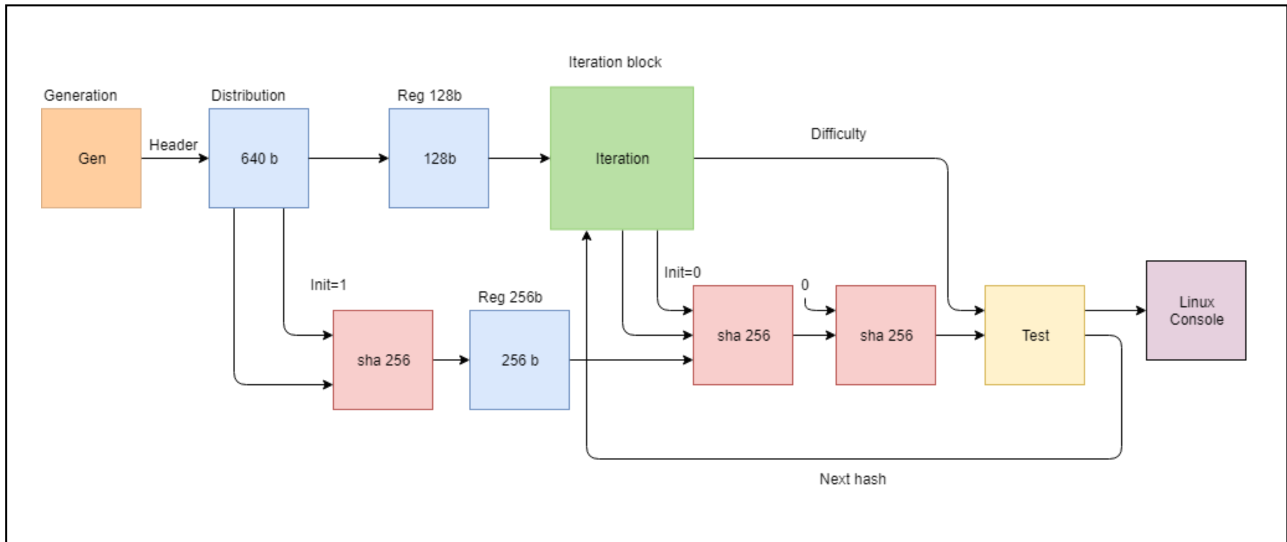

figure 11: Architecture du test du bloc de minage

Cette architecture a pu être tester en ajoutant un génération créant un header et l'envoyant au bloc distribution. La sortie du bloc de minage, c'est à dire la solution trouvée et vérifiée par le bloc test et quant à elle affichée sur la console Linux.

Une fois le comportement de l'architecture validée cette dernière devait être portée sur FPGA. L'architecture globale du projet requiert une communication entre le bloc de minage sur FPGA et l'ordinateur. En effet, une fois implémenté sur le FPGA le bloc de minage ne peut plus communiquer directement sur la console de notre ordinateur. L'architecture de la figure 12 ci-dessous permet de remédier à ce problème.

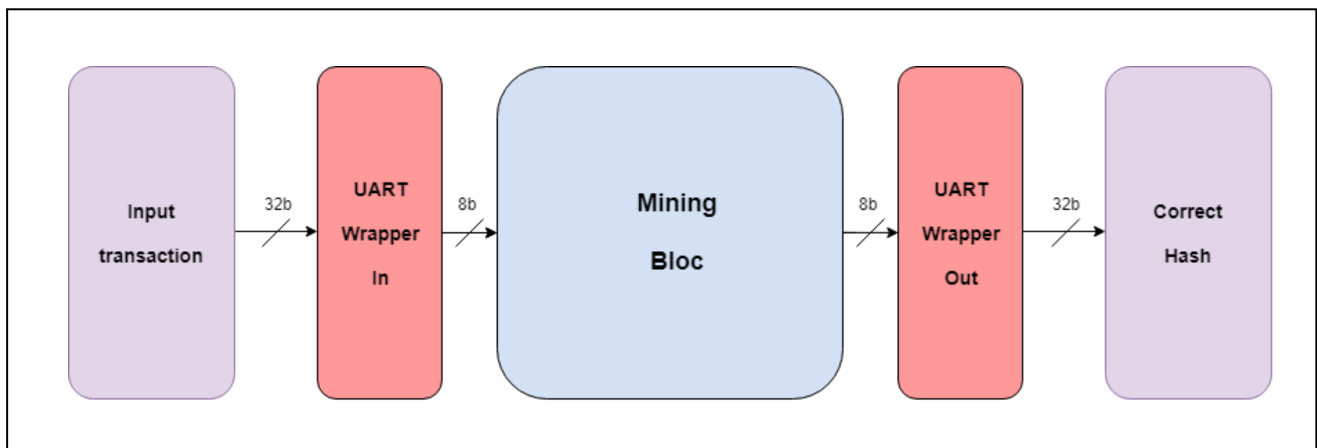

figure 12: Architecture globale

L'utilisation du protocole UART a été choisi pour assurer cette communication. Notre bloc prenant en entré des messages sur 8 bits, il fallait mettre en place des wrappers d'entrée et de sortie permettant de respectivement déconcaténer et concaténer les messages reçus et envoyés.

Cependant, après développement de cette architecture, la communication entre la carte et le PC s'est révélée non fonctionnelle.

Dans son état actuel, l'architecture est capable de miner un bloc seulement. Afin que son fonctionnement puisse être rendu automatique, il a fallu rajouter un bloc pouvant gérer la réinitialisation du processus de minage. Il a aussi fallu modifier les autres blocs pour qu'ils puissent reprendre leur fonctionement du début.
Cette architecture est donc commandée par l'envoi d'un bloc ainsi qu'un signal qui indique à l'architecture de reprendre son travail du début.

Pour commander cette architecture nous avons simplement eu le temps de décrire l'idée générale de son fonctionnement comme ci-dessous :

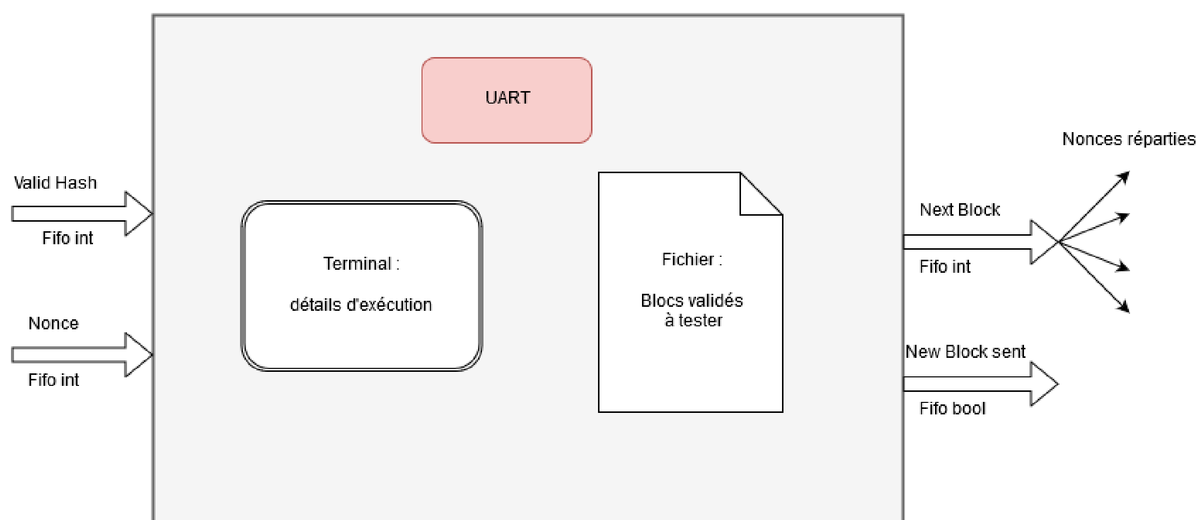

figure 13: Croquis d'architecture de contrôleur

Ce contrôleur a pour but d'envoyer des blocs déjà validés par le réseau Bitcoin à notre architecture et de les répartir sur les différents mineurs qui lui sont reliés. Quand un bloc est envoyé, est aussi envoyé le signal de réinitialisation.
Dès qu'un mineur a trouvé la solution, elle est envoyée au PC et elle est affiché sur le terminal avant d'être vérifiée et comparée au fichier contenant les blocs validés.
Si le résultat est correct le contrôleur passe au bloc suivant et le traite de la même manière.

Au lieu d'avoir recours à un fichier de blocs validés, nous avons aussi pensé à utiliser un LFSR (Linear Feedback Shift Register ou registre à décalage à rétroaction linéaire) pour générer nos blocs à valider de manière pseudo aléatoire.

9- Optimisation / Travail futur

Avec une communication fonctionnelle entre le PC et la carte, les mesures de performances auraient pu être effectuées. Grâce à ces résultats nous pouvions envisager la parallélisation d'unité de minage sur une ou plusieurs cartes FPGA.
Cette parallélisation aurait étée possible grâce au contrôleur dont nous avons esquissé l'architecture. Le travail serait ainsi réparti entre les différentes unités de calcul en leur affectant initialement une plage de nonces. L'architecture des mineurs resterait intacte et une simple parallélisation des blocs de minage serait nécessaire.

De plus, l'architecture de minage utilisée n'est pas optimisée. Ses performances peuvent être améliorée en réduisant les temps de transfert entre les blocs.
Pour ce faire les fonctions n'ayant pas besoin d'être effectuée dans des blocs externes au bloc SHA-256 sont incluses dans ce dernier.

10- Conclusion

Grâce à ce projet nous avons pu découvrir l'environnement des crypto monnaies et en particulier celui du Bitcoin. De plus, nous avons pu mettre en application les connaissances acquises sur la synthèse HLS au cours de ce semestre. Le développement du module de SHA-256 fut complexe car l'algorithme sous jacent exige la plus grande rigueur. Aussi, le développement de ce module nous a fait comprendre l'importance du code libre qui nous a permis de grandement nous faciliter la tâche et d'avoir un point de référence tout au long du projet.

Bien que le comportement final de l'architecture ait été validé, la communication avec la carte demeure quant à elle non fonctionnelle. Nos connaissances des protocoles de communication se sont révélées insuffisantes, en particulier au sujet du protocole UART.

Ce projet nous a permis de comprendre l'importance du travail en équipe et de la coordination en son sein. La répartition du travail a aussi occupé une place importante dans le déroulement du projet. En effet, le travail a été divisé en deux parties : le développement d'un SHA-256 fonctionnel et la conception de l'architecture du bloc de minage.

Bibliographie

Explication de la blockchain/Bitcoin:
https://chrispacia.wordpress.com/2013/09/02/bitcoin-mining-explained-like-youre-five-part-2-mechanics/

https://fr.wikipedia.org/wiki/Blockchain#Fonctionnement

https://medium.com/futurs-io/pour-une-poign%C3%A9e-de-bitcoins-la-validation-des-transactions-sur-une-blockchain-1-2-d740497d8108

https://www.youtube.com/watch?v=pHEw4Z3UgUc

https://crypto-monnaie.pro/minage-crypto-monnaie/

https://bitcoin.stackexchange.com/questions/12427/can-someone-explain-how-the-bitcoin-blockchain-works

https://en.bitcoin.it/wiki/Protocol_documentation

http://www.doc.ic.ac.uk/~ma7614/topics_website/tech.html

https://www.khanacademy.org/economics-finance-domain/core-finance/money-and-banking/bitcoin/v/bitcoin-transaction-block-chains

https://www.khanacademy.org/economics-finance-domain/core-finance/money-and-banking/bitcoin/v/bitcoin-transaction-block-chains

Historique blockchain bitcoin:
https://www.blockchain.com/explorer

Demo blockchain
https://anders.com/blockchain/

Code BlockChain:
https://github.com/openblockchains/awesome-blockchains

Calculateur en ligne SHA string ou hex:
https://www.pelock.com/products/hash-calculator

Explication du SHA-256:
https://en.wikipedia.org/wiki/Cryptographic_hash_function#message_digest

Code SHA-256:
http://www.zedwood.com/article/cpp-sha256-function

https://github.com/stiggy87/ZynqBTC

https://github.com/ikwzm/SECURE_HASH

https://github.com/stiggy87/ZynqBTC/wiki/HLS-C-Code-Explanation

SHA-256 réalisé par Brett Nicholas:
https://github.com/bigbrett/wssha256_hls

Bibliothèque OpenSSL:
https://github.com/openssl/openssl

Annexe: Code SHA-256

```
void sha256_transform(unsigned int data[16]){

  unsigned int a,b,c,d,e,f,g,h,i,t1,t2,m[64];
  for (i=0; i < 16; ++i){
#pragma HLS UNROLL
     m[i]=data[i];
  }

//boucle 1 de mise en forme du message:
  for (int i = 0; i < 64; ++i){
#pragma HLS ...
     m[i] = SIG1(m[i-2]) + m[i-7] + SIG0(m[i-15]) + m[i-16];
  }
  a = state[0];
  b = state[1];
  c = state[2];
  d = state[3];
  e = state[4];
  f = state[5];
  g = state[6];
  h = state[7];

//boucle 2 de traitement du message
  for (i = 0; i < 64; ++i) {
#pragama HLS ...
     t1 = h + EP1(e) + CH(e,f,g) + k[i] + m[i];
     t2 = EP0(a) + MAJ(a,b,c);
     h = g;
     g = f;
     f = e;
     e = d + t1;
     d = c;
     c = b;
     b = a;
     a = t1 + t2;
  }

  state[0] += a;
```

```
  state[1] += b;
  state[2] += c;
  state[3] += d;
  state[4] += e;
  state[5] += f;
  state[6] += g;
  state[7] += h;

}
```