POLYTECHNIQUE MONTRÉAL
LE GÉNIE EN PREMIÈRE CLASSE
Département de génie informatique et génie logiciel

INF3500

Conception et réalisation de systèmes numériques

Rapport de laboratoire #2

Circuits combinatoires

Critères	Points
Conception du module	/ 7
Synthèse et implémentation	/ 7
Questions	/ 2
Discussion	/ 2
Rapport : Présentation et qualité de la langue	/ 2
Total	/20

Soumis par :

Alexandre Morinvil, #1897222

Nicolas Valenchon, #2032097

Date :

11 mars 2020

Table des matières

1 Objectifs

Les objectifs de ce laboratoire étaient de confirmer notre compréhension des circuits combinatoires en précédant à la simulation, la synthétisation et l'implémentation sur FPGA d'un circuit combinatoire décrit en VHDL. La fonction combinatoire que ce laboratoire a pour objectif de concevoir un module réalisant l'algorithme SHA-256. Concrètement, ce laboratoire vise quatre buts spécifiques, soient : implémenter un circuit séquentiel, comprendre le concept de machine à états, se familiariser avec la notion d'optimisation matérielle et comprendre les dépendances entre les données reliées au cycle d'horloge.

2 Description du système

Tel que mentionné dans le laboratoire No. 2, l'algorithme SHA-256 est une fonction de hachage de la famille des SHA-2 (*Secure Hash Algorithm 2*), générant une « empreinte », ou « hachage », de 256 bits. Ainsi, en appliquant l'algorithme SHA-256 sur un mot encodé sur une longueur ≤ 2^{64} bits, une empreinte de 256 bits est générée. Le calcul de l'empreinte d'un mot est exécuté en plusieurs itérations au travers desquelles des fonctions de compressions sont appliquées sur une fraction du mot initial. (Dion, 2020) Ainsi, dans le cadre de ce laboratoire, en continuation du laboratoire No. 2 où les fonctions de compression du module SHA-256 ont été implémentées, le système conçu dans ce laboratoire est l'ensemble de la fonction de hachage SHA-256.

Dans le cadre de ce laboratoire, l'architecture générale nécessaire au fonctionnement du module a été fournie. Ceci comprend notamment les fichiers `sha256-top.vhd`, `debouncer.vhd` et `pulse.vhd` réalisant un travail d'arrière-plan nécessaire au fonctionnement de l'implémentation du module SHA-256. Le fonctionnement de ces fichiers ne fera pas l'objet de ce laboratoire.

Afin de pouvoir interfacer l'entrer et les sorties du module sur un FPGA, Le système conçu implémente donc une fonction de hachage prenant en entrée 16 bits aléatoires, sans restriction sur la valeur du mot, et renvoyant une sortie sur 256 bits. L'implémentation du module implémentant les calculs réalisés par l'algorithme de la fonction SHA-256 est contenue dans le fichier `sha256.vhd`. Le pseudo code de l'algorithme SHA-256 peut être décrit par la figure suivante.

```
def digest(new_input, input, output, output_valid):

    if new_input:
        a = H[0]
        b = H[1]
        c = H[2]
        d = H[3]
        e = H[4]
        f = H[5]
        g = H[6]
        h = H[7]
        output_valid = False

    for i in range(0, 63):
        if i < 16:
            W[i] = input[i]
        else:
            W[i] = sigma3(W[i-2]) + W[i-7] + sigma2(W[i-15]) + W[i-16]

        T1 = h + sigma1(e) + ch(e, f, g) + K[i] + W[i]
        T2 = sigma0(a) + maj(a, b, c)
        h = g
        g = f
        f = e
        e = d + T1
        d = c
        c = b
        b = a
        a = T1 + T2

    H[0] += a
    H[1] += b
    H[2] += c
    H[3] += d
    H[4] += e
    H[5] += f
    H[6] += g
    H[7] += h

    output = H[0] & H[1] & H[2] & H[3] & H[4] & H[5] & H[6] & H[7]
    output_valid = True
```

Figure 2-1 : Pseudocode de résumant les étapes de l'algorithme SHA-256

Ainsi, l'implémentation de cet algorithme a été complétée en VHDL en réalisant un circuit séquentiel décrivant une machine à états. Le schéma de la machine a états permettant de réaliser la fonction de hachage est montrée sur la figure ci-dessous.

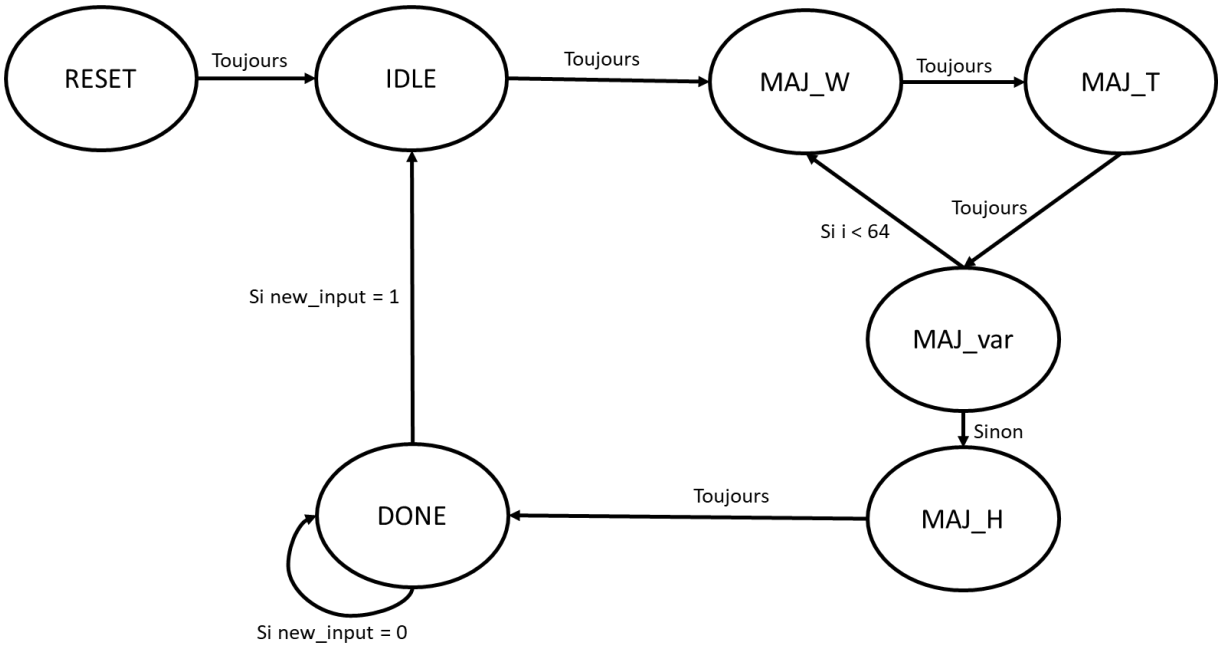

Figure 2-2 : Schéma du diagramme d'états implémenté

Ainsi, en parcourant les états affichés dans le diagramme d'état ci-haut à partir de l'état RESET, le système est en mesure de prendre en entrée un mot de 16 bits et de mettre en sortie un hach de 256 bit. Le comportement de chacun des états est décrit dans le fichier `sha256.vhd` et ceux-ci seront abordés dans les sous-sections suivantes.

2.1 État RESET

L'état RESET, est l'état dans lequel le circuit se place suite lorsque l'on appuie sur le bouton de « reset » du système. Cet état est donc l'état initial du circuit. De plus, le système est conçu de tel manière que pour pouvoir procéder au Hachage d'une nouvelle entrée, il est nécessaire de passer par l'état RESET afin de réinitialiser les valeurs de réinitialiser les digests `H(i)` à leur état initial (le seed). Les valeurs initiales assignées au digest (le seed), sont spécifiées par le standard de l'algorithme du SHA-256 et sont affichées dans la figure ci-dessous.

$$H_0^{(0)} = \texttt{6a09e667}$$
$$H_1^{(0)} = \texttt{bb67ae85}$$
$$H_2^{(0)} = \texttt{3c6ef372}$$
$$H_3^{(0)} = \texttt{a54ff53a}$$
$$H_4^{(0)} = \texttt{510e527f}$$
$$H_5^{(0)} = \texttt{9b05688c}$$
$$H_6^{(0)} = \texttt{1f83d9ab}$$
$$H_7^{(0)} = \texttt{5be0cd19}$$

Figure 2-3 : Valeurs initiales des digests (Seed) pour l'algorithme SHA-256

2.2 État IDLE

L'état IDLE, ou « immobile », est l'état dans lequel le système demeure en attente d'une nouvelle entrée. Ainsi, le système vérifie si l'entrée qui lui est donnée est une nouvelle entrée, ce qui lui est indiqué par une variable nommée « new_inpput ». Si le système a reçu une nouvelle entrée, il met à jour la valeur des variables intermédiaires `a` à `h`, réinitialise le compteur d'itération `i` à 0 et passe à l'état MAJ_W. Le pseudocode correspondant au traitement effectué dans cette état est affiché dans la figure suivante.

```
if new_input:
    a = H[0]
    b = H[1]
    c = H[2]
    d = H[3]
    e = H[4]
    f = H[5]
    g = H[6]
    h = H[7]
    output_valid = False
```

Figure 2-4 : Pseudocode du traitement effectué dans l'état IDLE

2.3 État MAJ_W (Mise à jour de W)

L'état MAJ_W, pour mise à jour de la variable W, est l'état dans lequel la variable intermédiaire W est mise à jour. Selon que la valeur du compteur d'itérations est inférieur à 16 ou non cet état assignera une valeur à l'espace approprié de la variable W. Le pseudocode associé au traitement effectué dans cet état est affiché dans la figure suivante.

```
if i < 16:
    W[i] = input[i]
else:
    W[i] = sigma3(W[i-2]) + W[i-7] + sigma2(W[i-15]) + W[i-16]
```

Figure 2-5 : Pseudocode du traitement effectué dans l'état MAJ_W

Il est à noter que les fonctions `sigma2(x)` sont les fonctions qui ont été implémentées dans le laboratoire 2.

Cet état mène systématiquement à l'état MAJ_T.

2.4 État MAJ_T (Mise à jour de T)

L'état MAJ_T, pour mise à jour de la variable T, est l'état dans lequel la variable intermédiaire T est mise à jour et le compteur d'itération est concurremment incrémenté. Ainsi, cet état incrément la valeur de le variable `i` et le pseudocode associé au traitement effectué sur les variables `T1` et `T2` est affiché dans la figure suivante.

```
T1 = h + sigma1(e) + ch(e, f, g) + K[i] + W[i]
T2 = sigma0(a) + maj(a, b, c)
```

Figure 2-6 : Pseudocode du traitement effectué sur les variables T1 et T2 dans l'état MAJ_W

Il est à noter que les fonctions `ch(x, y, z)`, `sigma0(x)` et `sigma1(x)` sont les fonctions qui ont été implémentées dans le laboratoire 2. De plus, la valeur `K[i]` correspond à une valeur de la table de constantes définies dans les spécifications de l'algorithme SHA-256. Cet état passe systématiquement à l'état MAJ_var.

2.5 État MAJ_var (Mise à jour des variables intermédiaires)

L'état MAJ_var, pour mise à jour des variables intermédiaires, est l'état dans lequel les variables intermédiaire `a`, `b`, `c`, `d`, `e`, `f`, `g` et `h` sont mises à jour. Le pseudocode associé au traitement effectué dans cet état est affiché dans la figure suivante.

```
h = g
g = f
f = e
e = d + T1
d = c
c = b
b = a
a = T1 + T2
```

Figure 2-7 : Pseudocode du traitement effectué dans l'état MAJ_var

Si le compteur d'itération a une valeur inférieure à 64, l'état suivant sera l'état MAJ_W sinon l'état suivant sera l'état MAJ_H.

2.6 État MAJ_H (Mise à jour des variables intermédiaires)

L'état MAJ_H, pour mise à jour des variables `H(i)`, est l'état dans lequel l'on met à jour la variable `H(i)`. Le pseudocode associé au traitement effectué dans cet état est affiché dans la figure suivante.

```
H[0] += a
H[1] += b
H[2] += c
H[3] += d
H[4] += e
H[5] += f
H[6] += g
H[7] += h
```

Figure 2-8 : Pseudocode du traitement effectué dans l'état MAJ_var

Cet état mène systématiquement à l'état DONE.

2.7 État DONE

L'état DONE, ou « terminé », est l'état signalant la fin du hachage d'un blob ou du message entier. Le pseudocode associé au traitement effectué dans cet état est affiché dans la figure suivante.

```
output = H[0] & H[1] & H[2] & H[3] & H[4] & H[5] & H[6] & H[7]
output_valid = True
```

Figure 2-9 : Pseudocode du traitement effectué dans l'état MAJ_var

Si la variable `new_input` a une valeur de 0, cet état restera sur lui-même, sinon si la variable `new_input` a une valeur de 1, une transition se fera vers l'état IDLE.

3 Vérification par simulation

3.1 Stratégie de simulation

Le banc d'essai utilisé est implémenté dans le fichier `sha256_tb.vhd`. La stratégie utilisée pour vérifier la conformité du système a consisté à imposer au système un vecteur de test composé de 8 entrées aléatoire prédéfinies de 16 bits pour lesquelles la sortie attendue est connue. Le vecteur de test utilisé est affiché dans la figure suivante.

```
constant TESTS_VECTOR : test_vector(0 to 7) := (
  (X"6162", X"0603"),
  (X"BEEF", X"9C7C"),
  (X"DEAD", X"931C"),
  (X"BADD", X"1C6F"),
  (X"BABE", X"3E84"),
  (X"2BAD", X"E19E"),
  (X"FEED", X"09BC"),
  (X"0000", X"CFC7")
  );
```

Figure 3-1 : Vecteur de test utilisé dans le banc de test en VHDL

Ainsi, le banc d'essai a été programmé pour vérifier que pour les 8 entrées qui sont imposées au système, la sortie obtenue correspond à la sortie attendue. Pour chacun mot bits imposé en entrée, le banc de test attend que le signal `output_valid` ait la valeur de '1' afin de savoir qu'il peut procéder au traitement de l'entrée suivante.

Lors de la simulation, en cas de non correspondance entre la sortie attendue et la sortie obtenue pour une entrée donnée, une erreur de gravité « warning » est lancée. Dans le cas d'une simulation réussie, une erreur de gravité « failure » est lancée indiquant que la simulation a terminé sans erreur.

Cette stratégie n'a pas été choisie puisqu'elle était imposée dans les spécifications du travail à accomplir. Ceci étant dit, l'on peut considérer que cette approche de simulation est adéquate en testant uniquement un échantillon puisque l'on peut raisonnablement supposer que si le système donne un résultat exact pour 8 valeurs, il donnera un résultat exact pour toutes les entrées que l'on pourrait lui donner. Ceci évite de procéder à un test exhaustif des 2^{16} entrée possibles.

Ceci étant dit, il aurait été pertinent de tester davantage de cas limites, par exemple la valeur `X"1111"`.

3.2 Description des résultats obtenus

La figure suivante affiche les 20 premières microsecondes du chronogramme de la simulation exécutée.

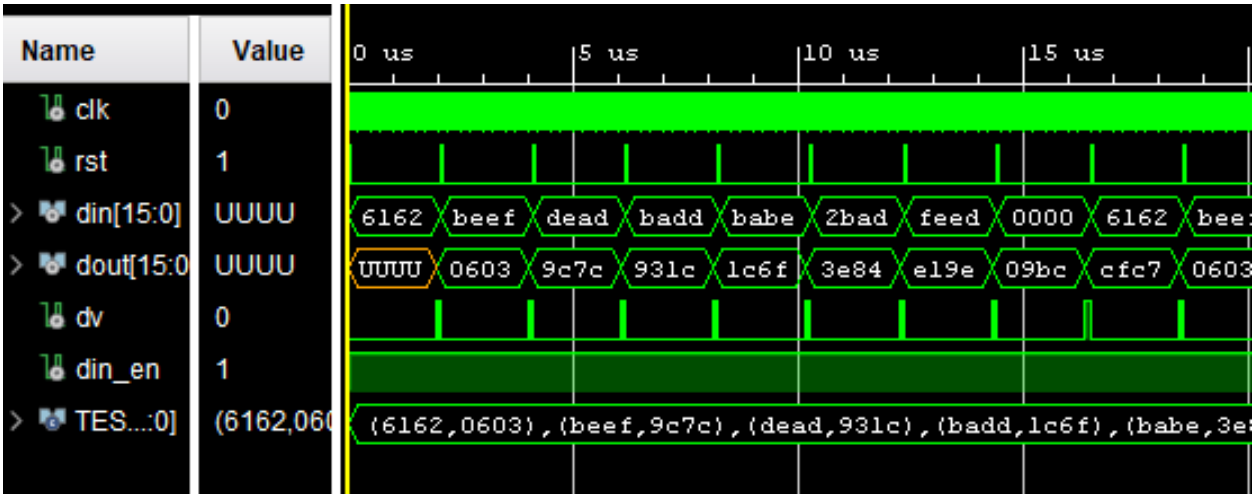

Figure 3-2 : 20 premières microsecondes du chronogramme de la simulation du système de compression

Dans la simulation ci-haut, l'on peut observer que les premières entrées et sorties (`din` et `dout`) correspondent aux couples d'entrée et de sortie attendus, soient {`(X"6162", X"0603"), (X"BEEF", X"9C7C"), (X"DEAD", X"931C"), (X"BADD", X"1C6F"), (X"BABE", X"3E84"), (X"2BAD", X"E19E"), (X"FEED", X"09BC"), (X"0000", X"CFC7")`}. On observe également que la sortie est indéfinie au début de la simulation. Ceci est le résultat attendu puisque lors du calcul de la première valeur de hach, la sortie ne doit pas être connue puisqu'elle n'est pas encore calculée.

Enfin, la figure suivante contient la sortie de la console suite a la simulation. Celle-ci confirme que la simulation a réussi puisqu'elle contient le message d'erreur attendu en cas de simulation réussie, soit « Error : Simulation ended ».

```
run all
Failure: Simulation ended
Time: 33055 ns  Iteration: 0  Process: /sha256_tb/line__61  File: C:/TEMP/INF3500/TP4/
$finish called at time : 33055 ns : File "C:/TEMP/INF3500/TP4/project_tp4/project_tp4.
```

Figure 3-3 : Sortie de la console suite à la simulation

4 Ressources utilisées et performance

4.1 Statistiques d'utilisation

Les figures suivantes résument les statistiques d'utilisation du FPGA Nexys4DDR.

Resource	Utilization	Available	Utilization...
LUT	1031	63400	1.63
LUTRAM	172	19000	0.91
FF	628	126800	0.50
IO	36	210	17.14
BUFG	1	32	3.13

Figure 4-1 : Tableau des ressources utilisées lors de l'implémentation du système sur un FPGA Nexys4DDR

Notre implémentation utilise relativement peu de ressources du FPGA, avec une utilisation de 1,63% des LUT. Le nombre de bascule utilisé est lui aussi relativement faible par rapport à la quantité disponible, soit de 0.50%. De plus, la simulation indique également que 36 des 210 entrée/sorties accessibles sur le FPGA sont utilisées, ce qui représente une utilisation de 17,14% des entrées/sorties accessibles.

Du plus, des statistiques consernant les LUTRAM (LookUp Table RAM) ainsi que les BUFG (Global Clock Buffer) sont fournies dans la figure ci-haut et il est possible de constater que l'utilisation de ces deux ressources est relativement basse également, à 0.91% et 3.13% respectivement.

4.2 Performance

La figure suivante affiche un résumé de l'utilisation des ressources et des performances temporelles du système réalisé.

Tableau 4-1 : Résumé de l'utilisation des ressources et des performances du système

LUT	Bascules	Fréquence (MHz)	Latence (ns)	Débit (opérations/s)
1031	628	121,728	3920	255102

Ainsi, pour ce qui concerne l'utilisation des LUT et des Bascules, ces valeurs proviennent de la description de l'utilisation des ressources réalisée dans la section ci-haut. Pour ce qui est des informations concernant les performances du système, les calculs et les données permettant d'obtenir la valeur de la fréquence, de la latence et du débit sont affichées ici-bas.

La figure suivante affiche les informations des délais nécessaires au calcul de la fréquence.

Figure 4-2 : pire marge de temps avant le prochain front d'horloge et pire marge de temps après un front d'horloge

Ainsi, à l'aide de ces deux valeurs, la fréquence peur être déterminée par le calcul suivant :

$$Fréquence = \frac{1}{(8.054 + 0.161) * 10^{-9}} = 121{,}728\ MHz$$

Pour ce qui est de la fréquence et du débit, il est d'abord nécessaire de déterminer le nombre de cycle d'horloge entre deux passage à l'état IDLE. En connaissant le diagramme d'états du système et en sachant qu'un changement d'état ne ce produit qu'une fois par cycle de 20 ns, il est possible de déterminer ces deux valeurs à l'aide des calculs suivants :

$$\text{Nombre de cycles} = 1 + 3 \times 64 + 1 + 1 + 1 = 196 \text{ cycles d'horloges par opération}$$

$$Latence = 196 \times 20\ ns = 3920\ ns$$

$$Débit = \frac{1}{3920 * 10^{-9}} = 255102\ opérations/s$$

Le temps de latence est relativement faible : il est possible de traiter un grand nombre de mot par seconde. Ce qui représente 512 × 255102 = 130612224 bits/seconde = 124,6 mio /seconde.

5 Réponses aux questions

5.1 Question 1

Lors de la synthèse, les boucles sont déroulées par le compilateur, si le nombre d'itérations est connu à la compilation, rendant ainsi le code synthétisable. Or, on vous a dit que vous ne pouvez pas faire de boucle pour votre module, car ça ne serait pas synthétisable. Expliquez pour quelle raison majeure cela est le cas.

Notre module n'aurait pas pu contenir de boucle car il faut plusieurs cycles d'horloges pour exécuter une instruction. Cela est dû à l'interdépendance des valeurs : Pour calculer a, il faut au préalable avoir mis à jour T1 qui a besoin de la dernière valeur de W qui vient d'être calculé au début de l'instruction, or cette valeur sera mise à jour au prochain front d'horloge. On se doit donc de diviser notre instruction en 3 états et de dérouler nous-même la boucle. Ainsi, l'interdépendance des valeur empêche leur calcul de manière concurrente dans une boucle for.

5.2 Question 2

Les modules debouncer et pulse (`debouncer.vhd`, `pulse.vhd`) sont deux modules séquentiels. Pour chaque module, énumérer le(s) bascule(s) et expliquer ce que le module fait.

Dans le module debouncer une première bascule met à jour la valeur de `output` et une autre celle de `counter`. Les deux dépendent de `input` et de `counter`. Le reset remet les valeurs à 0. Le module va attendre une entrée `input`. Tant qu'une entrée est présente, à chaque front montant d'horloge la variable `counter` va s'incrémenter et la valeur `output` va rester à 0, jusqu'à ce que `counter` atteigne sa valeur maximale à savoir 2**W – 1. Le `counter` va arrêter de s'incrémenter et le `output` va passer à 1. S'il n'y a pas ou plus d'entrée, `counter` et `output` sont remis à 0.

Le module pulse est composée de 2 bascules : une pour mettre la valeur de `state` à jour et une pour la valeur de `output`. Le reset met les deux valeurs à 0. `state` prend la valeur de `input` au front montant de l'horloge. `output` prendre la valeur de l'opération logique `input and not state`.

6 Discussion

La fonction de hachage SHA-256 a été décrite en VHDL, simulée, synthétisée et implémentée sur un FPGA Nexys4DDR. Lors de l'utilisation du banc de teste fourni avec les spécifications du laboratoire, il a été possible de confirmer le bon fonctionnement du système conçu puisque pour un ensemble de 8 entrée aléatoires la sortie obtenue correspond à la sortie attendue. Ainsi, les résultats obtenus expérimentalement correspondent aux résultats attendus théoriquement, il n'y a donc pas eux de phénomènes inattendus lors de la réalisation du système.

Pour ce qui est de l'optimalité du système conçu, deux aspects sont généralement à prendre en considération : l'utilisation des ressources et la performance. En ce qui concerne l'utilisation des ressources, l'on peut de manière générale affirmer que l'utilisation des ressources est minimale puisque que moins de 4% des ressources du FPGA de toujours genre sont utilisées, à l'exception des entrées et sortie dont une utilisation de 17,14% est faite. Pour ce qui est des performances, avec une fréquence de 121,728 MHz, on peut considérer que le système conçu et implémenté est relativement efficace.

Lors du laboratoire, une des difficultés rencontrées est par rapport au sens de la lecture des vecteurs de données. De plus, puisque nous ne connaissions pas les résultats intermédiaires attendus dans le chemin des données, il a été plutôt ardu de débugger cette erreur.

De plus, une autre difficulté rencontrée est en ce qui a trait à la modélisation de la machine a été permettant de réaliser l'algorithme en matériel. Ceci étant dit, ce premier défi technique a été formateur et à améliorer notre compréhension des circuits séquentiels en VHDL.

Enfin, il aurait pu être intéressant de tester l'algorithme en réalisant plusieurs digests sur les blobs.

7 Références

Dion, O. (2020). *Labo 4 - Circuits séquentiels.* Montréal.

National Institute of Standards and Technology (2015). Secure Hash Standard (SHS). Gaithersburg,