

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

[Table des matières 2](#_Toc31476711)

[1 Objectifs 3](#_Toc31476712)

[2 Description du système 4](#_Toc31476713)

[2.1 Module Choose 5](#_Toc31476714)

[2.2 Module Majority 6](#_Toc31476715)

[2.3 Module Sigma 0 6](#_Toc31476716)

[2.4 Module Sigma 1 6](#_Toc31476717)

[2.5 Module Sigma 2 6](#_Toc31476718)

[2.6 Module Sigma 3 6](#_Toc31476719)

[3 Vérification par simulation 7](#_Toc31476720)

[3.1 Stratégie de simulation 7](#_Toc31476721)

[3.2 Description des résultats obtenus 8](#_Toc31476722)

[4 Ressources utilisées et performance 9](#_Toc31476723)

[4.1 Statistiques d’utilisation 9](#_Toc31476724)

[4.2 Performance 9](#_Toc31476725)

[5 Réponses aux questions 9](#_Toc31476726)

[5.1 Question 1 9](#_Toc31476727)

[5.2 Question 2 10](#_Toc31476728)

[6 Discussion 10](#_Toc31476729)

[7 Références 11](#_Toc31476730)

# 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.

# Description du système

*Décrire le système conçu dans le laboratoire.*

*Le cas échéant, donner un diagramme montrant la structure du système en ses modules principaux. Bien décrire les interfaces entre les modules.*

*Décrire l’organisation du code. Donner les noms des fichiers et quelle fonction chaque module réalise.*

*Décrire comment le système fonctionne. Ne pas répéter ce qui est donné en commentaire. La documentation donnée ici doit compléter le code. Donner des extraits de code si nécessaire.*

Le multiprocesseur construit pour ce laboratoire comportait 5 blocs principaux, montrés à la Figure 1. Le premier bloc … le deuxième bloc …

Le système est réparti sur 6 fichiers. Dans le fichier top.vhdl on retrouve …

Pour la fonction exponentielle, nous avons utilisé le pseudocode suivant … Son équivalent en VHDL est montré dans le fichier exp.vhdl.

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 ≤ 264 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.

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.

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.

L’implémentation du module opérant la fonction SHA-256 est réalisée dans le sha256.vhd. Le pseudo code de l’algorithme de di

L’implémentation du système est réalisée en mettant en cascade 6 modules de compression. Le tableau suivant affiche l’ensemble des modules composant le système, l’équation décrivant le traitement effectué par chacun des modules sur son signal d’entrée ainsi que le nom du fichier dans lequel est décrit chacune des fonctions.

**Tableau 2‑1:** Fonction de compression utilisée dans l'algorithme SHA-256 et organisation des fichiers implémentant leur module respectif

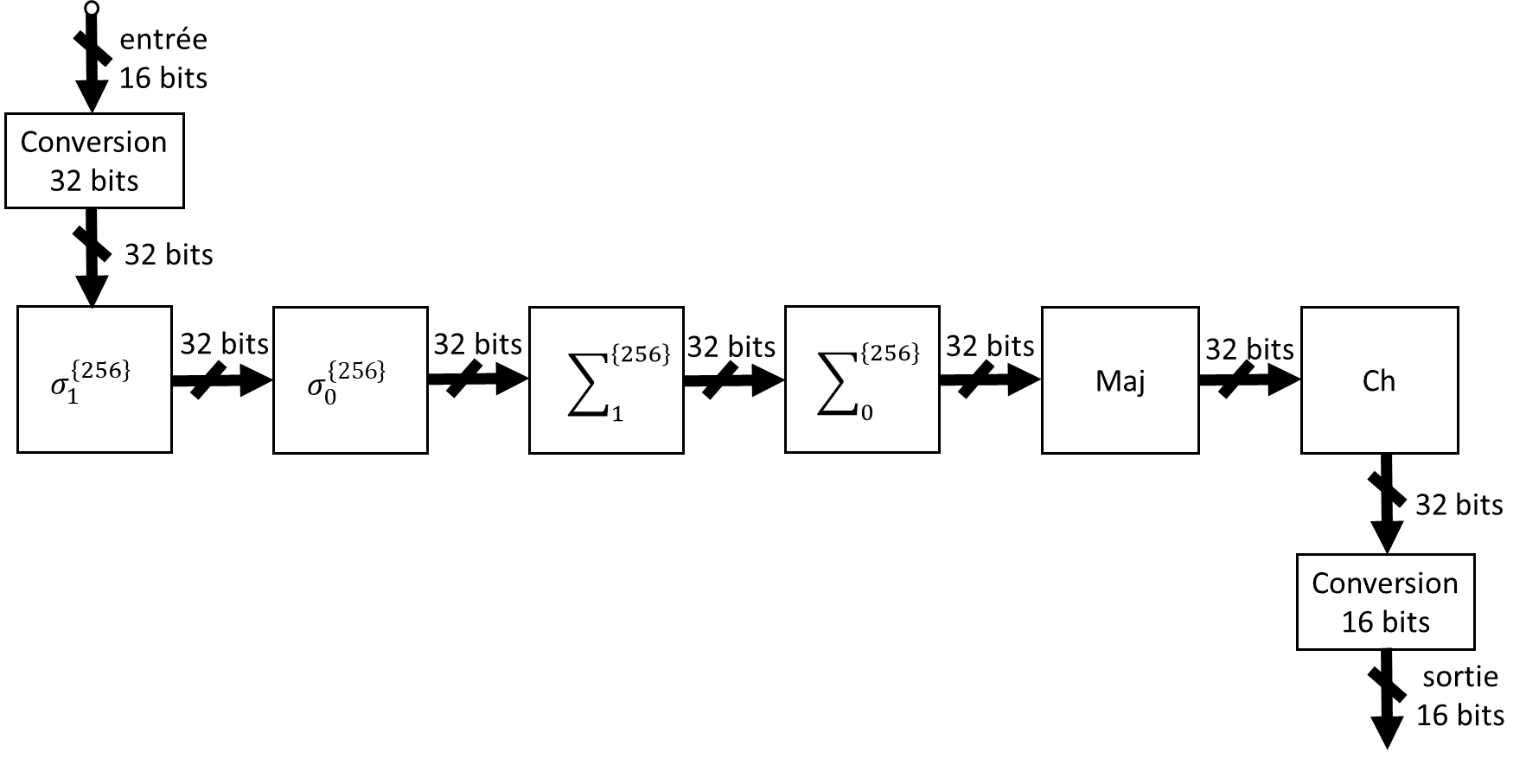
|  |  |  |
| --- | --- | --- |
| **Nom** | **Fichier** | **Équation** |
| Choose | ch.vhd |  |
| Majority | maj.vhd |  |
| Sigma0 | sigma0.vhd |  |
| Sigma1 | sigma1.vhd |  |
| Sigma2 | sigma2.vhd |  |
| Sigma3 | sigma3.vhd |  |

La description VHDL de chaque module est réalisée dans le fichier .vhdl correspondant tel qu’affiché dans le tableau ci-haut.

Ainsi, le système est réalisé en mettant en cascade les six modules dans une fichier nommée top.vhdl. L’ordre de la séquence des modules est affiché dans la figure suivante où le nom de chaque module est abrévié par son nom dans l’équation qui le décrit.

***Figure 2‑1 :*** *Cascade de compression des modules du système*

Ainsi, la réalisation du système peut donc être schématisé par le diagramme qui montre l’ensemble des modules connectés ainsi que le nombre de bits qu’ils prennent en entrée et en sortie.



***Figure 2‑2 :*** *Structure du système en ses modules principaux*

Les spécifications de chaque module composant la figure seront abordées dans les sous-sections suivantes.

Il est à noter que la conversion des mots de 16 bits en mots de 32 bits et la conversion du sens inverse est réalisée directement dans le fichier top.vhd en utilisant la fonction resize() de la librairie IEEE.NUMERIC\_STD.

## Module Choose

Le module choose est implémentée dans le fichier ch.vhd. Ce module prend en entrée un mot de 32 bits provenant du module majority et il donne une sortie sur 32 bits. Le traitement effectué sur son entrée est décrit dans le tableau 2-1, soit la fonction suivante :

La description détaillée de cette fonction a été traitée lors du rapport no. 1. Les valeurs des signaux , et sont toutes trois égales au mot de 32 bits provenant du module Majority.

## Module Majority

Le module majority est implémenté dans le fichier maj.vhd. Ce module prend en entrée un mot de 32 bits provenant du module sigma0 et il donne une sortie sur 32 bits. Le traitement effectué sur son entrée est décrit dans le tableau 2-1, soit la fonction suivante :

Les valeurs des signaux , et sont toutes trois égales au mot de 32 bits provenant du module Sigma0.

## Module Sigma 0

Le module sigma0 est implémenté dans le fichier sigma0.vhd. Ce module prend en entrée un mot de 32 bits provenant du module sigma1 et il donne une sortie sur 32 bits. Le traitement effectué sur son entrée est décrit dans le tableau 2-1, soit la fonction suivante :

## Module Sigma 1

Le module sigma1 est implémenté dans le fichier sigma1.vhd. Ce module prend en entrée un mot de 32 bits provenant du module sigma2 et il donne une sortie sur 32 bits. Le traitement effectué sur son entrée est décrit dans le tableau 2-1, soit la fonction suivante :

## Module Sigma 2

Le module sigma2 est implémenté dans le fichier sigma2.vhd. Ce module prend en entrée un mot de 32 bits provenant du module Sigma3 et il donne une sortie sur 32 bits. Le traitement effectué sur son entrée est décrit dans le tableau 2-1, soit la fonction suivante :

## Module Sigma 3

Le module sigma3 est implémenté dans le fichier sigma3.vhd. Ce module prend en entrée un mot de 32 bits provenant de l’entrée du système ayant été redimensionnée de 16 à 32 bits et il donne une sortie sur 32 bits. Le traitement effectué sur son entrée est décrit dans le tableau 2-1, soit la fonction suivante :

# Vérification par simulation

## Stratégie de simulation

Le banc d’essai utilisé est implémenté dans le fichier top\_tb.vhd. La stratégie utilisée pour vérifier la conformité du système a consisté à imposer au système 64 entrées aléatoires de 16 bits prédéfinis pour lesquelles la sortie attendue est connue. Par exemple, il était connu que pour une entrée de 0x7E51 la sortie attendue est de 0xC118. La figure suivante affiche un extrait du code déclarant les 5 premières valeurs imposées en entrée (INPUT) ainsi que les 5 premières valeurs attendues en sortie (OUTPUT).

|  |  |
| --- | --- |
| constant INPUTS : vector16\_t(0 to 63) :=(  x"7e51",  x"8740",  x"9040",  x"137f",  x"3d0d",  … | constant OUTPUTS : vector16\_t(0 to 63) :=(  x"c118",  x"28d4",  x"b75a",  x"1ac1",  x"1f9f",  … |

**Figure 3‑1 :** Extrait de code des premières définitions d’entrées et de sorties

Ainsi, le banc d’essai a été programmé pour vérifier que pour les 64 entrées qui son imposées au système, la sortie obtenue correspond à la sortie attendue. Pour chacun mot de 16 bits imposé en entrée, un intervalle de temps de 10 ns a été imposé avant la transition vers le mot suivant. Par conséquent, la simulation s’étale sur un intervalle de temps de 640 ns si aucune erreur ne survient. 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 « failure » est lancée indiquant que la simulation a échoué. Dans le cas d’une simulation réussie, une erreur de gravité « error » est lancée indiquant l’absence d’erreur.

L’extrait de code montrant l’implémentation de la vérification des couples entrée-sortie est affiché dans la figure suivante :

|  |
| --- |
| for i in 0 to INPUTS'length - 1 loop  switches <= INPUTS(i);  wait for 10 ns;  assert(leds = OUTPUTS(i))  report "Simulation failed" severity failure;  end loop;  assert (false)  report "Simulation successful" severity error; |

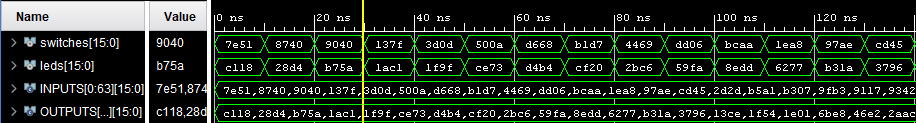
**Figure 3‑2 :** Extrait de code de la vérification des couples entrée-sortie

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 échantillant uniquement un certain nombre d’entrées possibles puisque cette fonction est déterministe et invariable, donc pour une entrée donnée il n’y a qu’une seule sortie possible et elle sera toujours la même. Donc, l’on peut raisonnablement supposer que si le système donne un résultat exact pour 64 valeurs, il donnera un résultat exact pour toutes les entrées que l’on pourrait lui donner.

De plus, avec des mots de 16 bits il existe 216 (65 536) couples d’entrée-sortie possible, il ne serait donc pas pratique de tester l’ensemble des valeurs d’entrée possible.

## Description des résultats obtenus

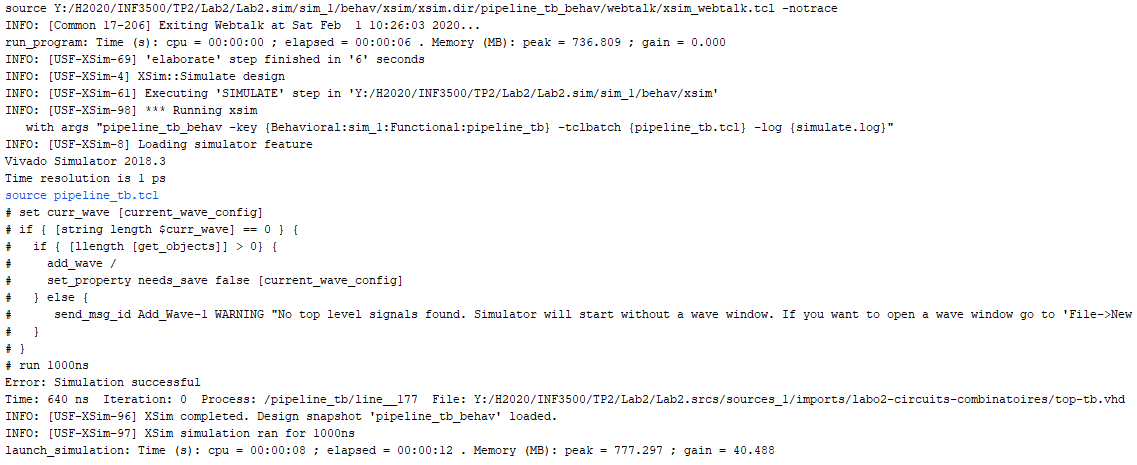
La figure suivante affiche les 100 premières nanosecondes du début du chronogramme de la simulation exécutée.



**Figure 3‑3 :** 100 premières nanosecondes du chronogramme de la simulation du système de compression

Dans la simulation ci-haut, l’on peut observer que les signaux INPUTS et OUTPUTS sont des signaux définis comme étant des tableaux de 64 mots de 16 bits chacun dont les premières valeurs sont respectivement {0x7E51, 0x8740, 0x9040, 0x3D0D, 0x500A} et {0xC118, 0x28D4, 0x1AC1, 0x1AC1, 0x1F9F}. De plus, l’on observe que les signaux switches et leds sont des tous les deux des signaux de 16 bits, changeant de valeur à intervalle de 10 ns, correspondant respectivement à l’entrée simulée et la sortie simulée. Or, l’on constate effectivement que les premières valeurs que prenne la sortie simulée correspond aux premières valeurs que possède la sortie attendue pour leurs entrées homologues.

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 successful » (à la cinquième ligne avant la fin).

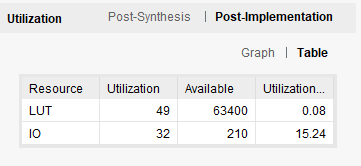


**Figure 3‑4 :** Sortie de la console suite a la simulation

# Ressources utilisées et performance

## Statistiques d’utilisation

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



**Figure 4‑1 :** Tableau des ressources utilisées lors de l’implémentation du système sur un FPGA Nexys4DDR

Ainsi, la figure indique que 49 des 63 400 LUT utilisables du FPGA ont été utilisées, ce qui représente une utilisation de moins de 0,08% des LUT disponibles. De plus, la simulation indique également que 32 des 210 entrée/sorties accessibles sur le FPGA sont utilisées, ce qui représente une utilisation de 15,24% des entrées/sorties accessibles.

## Performance

Le circuit système réalisé est un circuit combinatoire sans horloge, par conséquence il n’y a pas de contrainte de temps permettant d’évaluer les performances du système.

# Réponses aux questions

## Question 1

**La fonction maj est le préfix de *majority*. Expliquez de façcon concrète et concise pour quelle raison cette fonction est appelée ainsi dans le contexte où les entrées sont des vecteurs de bits (std\_logic\_vector).**

La fonction maj renvoi vrai si la majorité de ses entrées est vraies, c’est-à-dire quand au moins deux des trois entrées sont vraies et elle renvois faux si la majorité de ses entrées est fausse. La figure suivante illustre cette idée à l’aide d’une table de vérité :

|  |  |  |  |
| --- | --- | --- | --- |
| x | y | z | 𝑚𝑎𝑗(𝑥, 𝑦, 𝑧) |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

**Figure 6‑1 :** Table de vérité de la fonction

## Question 2

**Donnez un avantage et un désavantage de tester les 6 fonctions de compression du SHA-256 en utilisant un pipeline comme celui du module top.**

L’avantage de tester les 6 fonctions de compression du SHA-s56 en utilisant un pipeline tel que le top est la simplicité de la procédure. Effectivement, puisque l’on ne considère que l’entrée et la sortie du pipeline, le test est très simple à programmer et exécuter. De plus, si les tests fonctionnent, cela signifie que chacun des 6 module fonctionne puisque le système dans son ensemble fonctionne. Ceci est par opposition à tester les 6 fonctions de compressions individuellement, sans utiliser de pipeline.

Un désavantage de tester les 6 fonctions de compression du SHA-256 en utilisant un pipeline tel que le top est que cela rende le débogage plus difficile puisque l’on confirme la validité du système à partir de l’entrée et la sortie du pipeline entier plutôt que de vérifier le fonctionnement de chacun des modules individuellement. Donc, si une erreur était survenue, le test n’aurait pas offert d’information sur quel module est concerné.

# Discussion

La fonction de compression désirée a été conçue, 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 64 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 subjectivement affirmer qu’elles ont minimales puisque 0,08% des LUT sont utilisées lors de l’implémentation sur FPGA. Pour ce qui est de l’utilisation des entrée et sorties du FPGA, bien que l’analyse effectuée affirme que 15,24% de ces ressources sont utilisées, cette métrique n’a pas d’importance puisque les entrées et sorties, c’est-à-dire les DEL et les commutateurs ne sont utilisés que dans le but d’interfacer ce système afin de pouvoir plus facilement le manipuler. Ceci étant dit, lors de sont intégration éventuelle dans un système effectuant l’algorithme de hachage SHA-256, les entrées et sorties du systèmes de compression conçu dans ce laboratoire ne seront pas reliées aux entrées et sorties du FPGA. Enfin, pour ce qui est de la performance, puisque le système réalisé est un circuit combinatoire sans horloge, il n’y a pas de contrainte de temps permettant d’évaluer ses performances.

La procédure de test par simulation utilisée a permis de confirmer que le système effectue la compression désirée telle qu’attendu. De plus, la simulation de 64 entrées aléatoires différentes plutôt que de l’ensemble des 65 536 valeurs d’entrée possible permet de simplifier la simulation et ne compromet pas la validité du test étant donné que le système conçu est déterministe, invariable et relativement simple.

Ceci étant dit, l’approche de test utilisée a pour désavantage qu’elle ne vérifie que le fonctionnement de l’ensemble du système plutôt que pour chaque module individuellement, ce qui est une approche de teste en boîte noire. Par conséquent, s’il y avait eu des erreurs lors des tests effectués sur le système, il aurait été difficile de déterminer quel est le module concerné.

Par conséquent, une amélioration suggérée pour ce laboratoire serait de concevoir un banc de teste pour chacun des modules individuellement. Ceci demanderait davantage de travail. Ceci étant dit, cela simplifierait le travail de débogage en cas d’erreur et cela rendrait le système plus facile à modifier puisqu’il serait alors possible de cibler un module spécifique, le modifier et le tester de manière isolée.

Lors de ce laboratoire, il n’y a pas eu de difficultés particulières rencontrées, hormis le fait que le logiciel Vivado est encore relativement un nouvel outil auquel il a fallu se familiariser.

Pour la suite, il serait intéressant de continuer la conception d’une fonction de hachage SHA-256 complète en y intégrant le module de compression conçu et décrit dans ce rapport.

# Références

Dion, O. (2020). *Labo 2 - Circuits combinatoires.* Montréal.