POLYTECHNIQUE MONTRÉAL
LE GÉNIE EN PREMIÈRE CLASSE

Département de génie informatique et génie logiciel

INF3500

Conception et réalisation de systèmes numériques

Rapport de laboratoire #3

Banc d'essais

Critères	Points
Banc d'essai	/16
Questions	/2
Rapport : Présentation et qualité de la langue	/2
Total	/20

Soumis par :

Alexandre Morinvil, #1897222

Nicolas Valenchon, #2032097

Date :

18 février 2020

Table des matières

Table des matières 2
1 Objectifs 3
2 Vérification par simulation 4
2.1 Stratégie de simulation 4
2.2 Description des résultats obtenus 6
3 Réponses aux questions 13
3.1 Question 1 13
3.2 Question 2 13
3.3 Question 3 13
4 Références ..... **Erreur ! Signet non défini.**

1 Objectifs

Les objectifs de ce laboratoire étaient de confirmer notre compréhension de la conception et l'utilisation de banc d'essai. Pour ce faire, le laboratoire a consisté à programmer un court logiciel générant des vecteurs de test spécifiques aux modules à vérifier. Ils sont ensuite utilisés pour concevoir des bancs d'essais afin de vérifier le fonctionnement des modules en VHDL. Ainsi, concrètement, ce laboratoire vise deux buts spécifiques, soient : apprendre à générer des vecteurs de test et comprendre l'importance des tests.

2 Vérification par simulation

2.1 Stratégie de simulation

2.1.1 Approche préconisée

La stratégie de teste utilisée pour vérifier la conformité du système est la même que celle utilisée lors du laboratoire 2. L'approche utilisée pour les simulations a consisté à imposer 64 entrées de 32 bits pour lesquelles les sorties attendues étaient connues pour chacun des modules.

Parmi les 64 vecteurs d'entrées imposés à chacun des modules, une partie des entrées étaient des cas limites tandis que le reste était des valeurs aléatoires. Les cas limites utilisés sont l'ensemble des combinaisons d'entrées pour lesquelles tous les bits d'une ou de plusieurs des entrées sont des '1' logiques ou des '0' logiques.

La figure suivante affiche un extrait du code déclarant les 8 premières et 8 dernières valeurs imposées à l'entrée `x` (INPUT_1) dans le banc de test du module `ch`, ainsi que les sorties qui lui sont associées (OUTPUT).

```
constant INPUTS_1 : vector32_t(0 to 63) :=
(
       x"3f6cd60e",
       x"2b88aa36",
       x"3bbb6a30",
       x"3da7af08",
       x"2d92de18",
       x"076fab0c",
       x"00f625c1",
       x"38c91383",
       …
       x"00000000",
       x"ffffffff",
       x"00000000",
       x"ffffffff",
       x"00000000",
       x"ffffffff",
       x"00000000",
       x"ffffffff"
);
```

```
constant OUTPUTS : vector16_t(0 to 63) :=(
       x"1808b9f3",
       x"2a8e2cc8",
       x"2309d593",
       x"3bedcc8c",
       x"2e9129ee",
       x"36bfc1ad",
       x"2a5d9aaf",
       x"39342e34",
       x"2205fb54",
       …
       x"00000000",
       x"00000000",
       x"00000000",
       x"ffffffff",
       x"ffffffff",
       x"00000000",
       x"ffffffff",
       x"ffffffff"
  );
```

Figure 2-1 : Extrait de code des huit premières et dernières définitions de l'entrées x et de la sorties pour les vecteurs de test du module ch

Cette stratégie a été choisie puisqu'elle permet de tester le bon fonctionnement du module en évitant de tester l'ensemble des combinaisons possible. Effectivement, puisqu'il s'agit de modules exécutant des fonctions combinatoires (fonctions ayant une sortie fixe pour chaque combinaison d'entrée), en sélectionnant un échantillon d'entrées couvrant les cas limites ainsi qu'un certain nombre de combinaisons totalement aléatoires, il est possible de confirmer le fonctionnement de chaque module, sans nécessiter de teste l'ensemble des 2^{32} à 3×2^{32} combinaisons d'entrées possibles pour chaque module. Ainsi, l'on peut raisonnablement supposer que si le un module donne un résultat exact pour les 64 vecteurs d'entrées lui étant imposé, il donnera un résultat exact pour toutes les entrées que l'on pourrait lui donner.

2.1.2 Génération des bancs d'essai

Lors de ce laboratoire, pour la première fois, les bancs d'essais ont été réalisés dans le cadre du laboratoire plutôt que d'être fournis. Ainsi, les codes des bancs d'essai sont fournis dans les fichiers `CH_tb.vhd`, `MAJ_tb.vhd`, `SIGMA0_tb.vhd`; `SIGMA1_tb.vhd`, SIGMA`2_tb.vhd` et `SIGMA3_tb.vhd` pour les modules `ch`, `maj`, `sigma0`, `sigma1`, `sigma2` et `sigma3` respectivement.

Pour créer les bancs d'essai, il a été décidé de générer des vecteurs d'entrées comprenant quelques cas limites et de les complémenter avec des entrées aléatoires.

Afin de générer les vecteurs de teste, un petit programme C++ a été créé afin d'automatiser le processus de génération de vecteurs de teste. Le code du programme en question est réalisé dans le fichier « generateur_vecteur_de_test.cpp ».

Tel que précisé plus haut, les cas limites utilisés pour un module de N entrées sont les 2^N combinaisons d'entrées pour lesquelles tous les bits d'une ou de plusieurs des entrées sont des '1' logiques ou des '0' logiques. Par exemple, les module `ch` et `maj` ayant chacune 3 entrées de 32 bits, les cas limites sélectionnés sont celles affichées dans le tableau suivant.

Cas limite	Entrée x	Entrée y	Entrée z
1	0x00000000	0x00000000	0x00000000
2	0x00000000	0x00000000	0xFFFFFFFF
3	0x00000000	0xFFFFFFFF	0x00000000
4	0x00000000	0xFFFFFFFF	0xFFFFFFFF
5	0xFFFFFFFF	0x00000000	0x00000000
6	0xFFFFFFFF	0x00000000	0xFFFFFFFF
7	0xFFFFFFFF	0xFFFFFFFF	0x00000000
8 (ou 2^3)	0xFFFFFFFF	0xFFFFFFFF	0xFFFFFFFF

Figure 2-2 : Liste des cas limites considérés pour un module prenant trois entrées de 32 bits

Pour les module sigma0, sigma1, sigma2 et sigma3 prenant chacun une entrée de 32 bits, les cas limites considérés sont les cas valeurs de l'entrée x suivantes : {0x00000000, 0xFFFFFFFF}. Ainsi, pour un module de N entrées ayant donc 2^N cas limites, pour les 64 - 2^N vecteur d'entrées restants des mots aléatoires de 32 bit ont été générés.

Ensuite, les vecteurs de sortie sont générés de manière à ce que chaque valeur(s) imposée(s) en entrée à un module ait sa sortie correspondante. Dans le cas particulier des modules concernés dans ce laboratoire, il n'y avait qu'une seule sortie pour chaque module, par conséquent, les vecteurs de sorties ne contiennent qu'une valeur pour chaque sortie. Les fonctions utilisées pour générer les sorties sont définies dans un fichier séparé nommé « fonctions_TP3.h » qui est inclus dans le fichier « generateur_vecteur_de_test.cpp ».

Finalement, les vecteurs d'entrées et de sortie sont écrits dans les fichiers `ch.txt`, `maj.txt`, `sigma0.txt`, `sigma1.txt`, `sigma2.txt`, `sigma3.txt` où ils constituent le vecteur de test complet à transférer dans le banc de test de leur module respectif.

2.2 Description des résultats obtenus

Dans les simulations réalisées, les signaux `INPUTS` et `OUTPUTS` sont des signaux définis comme étant des tableaux de 64 mots de 32 bits contenant les données des vecteurs de test générés. De plus, les signaux `input` et `output` sont des tous des signaux de 32 bits, changeant de valeur à intervalle de 10 ns, correspondant à l'entrée simulée et la sortie simulée devant correspondre aux valeurs du vecteur de test compris dans leur simulation respective.

L'ensemble des tests réalisés à l'aide des six bancs de test ont fonctionnés. Les sous sections suivantes décrivent les résultats obtenus à l'issu des simulations réalisées.

2.2.1 Simulation du module Ch

La figure suivante affiche les 60 premières nanosecondes du début du chronogramme de la simulation exécutée afin de tester le module `ch`.

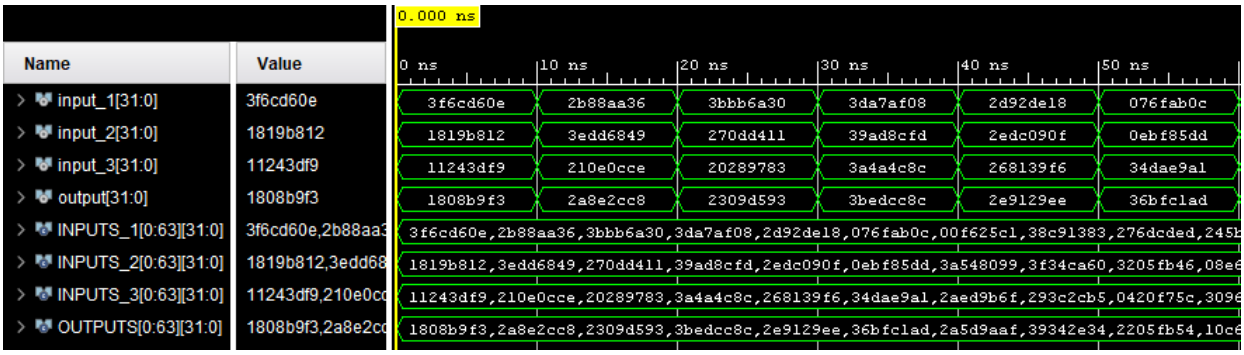

Figure 2-3 : 60 premières nanosecondes du chronogramme de la simulation du module ch

L'on constate effectivement que pour les mêmes premières valeurs d'entrée entre le vecteur de test et le signal simulé, les sorties correspondent, ce qui confirme le bon fonctionnement du module `ch` pour l'intervalle de temps représenté dans le chronogramme.

Enfin, la figure suivante contient la sortie de la console suite à 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).

```
source ch_tb.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window.
#   }
# }
# run 1000ns
Error: Simulation successful
Time: 640 ns  Iteration: 0  Process: /ch_tb/line__301  File: Y:/H2020/INF3500_Personel/Vivado/project_1/projec
INFO: [USF-XSim-96] XSim completed. Design snapshot 'ch_tb_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:02 ; elapsed = 00:00:08 . Memory (MB): peak = 809.320 ; gain = 0.000
```

Figure 2-4 : Sortie de la console suite à la simulation du module Ch

2.2.2 Simulation du module Maj

La figure suivante affiche les 60 premières nanosecondes du début du chronogramme de la simulation exécutée afin de tester le module `maj`.

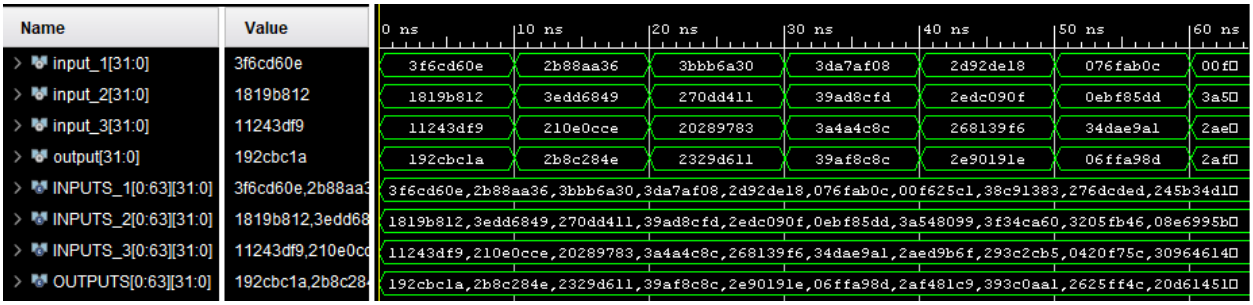

Figure 2-5 : 60 premières nanosecondes du chronogramme de la simulation du module maj

L'on constate effectivement que pour les mêmes premières valeurs d'entrée entre le vecteur de test et le signal simulé, les sorties correspondent, ce qui confirme le bon fonctionnement du module `maj` pour l'intervalle de temps représenté dans le chronogramme.

Enfin, la figure suivante contient la sortie de la console suite à 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).

```
source maj_tb.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window.
#   }
# }
# run 1000ns
Error: Simulation successful
Time: 640 ns  Iteration: 0  Process: /maj_tb/line__301  File: Y:/H2020/INF3500_Personel/Vivado/project_1/proje
INFO: [USF-XSim-96] XSim completed. Design snapshot 'maj_tb_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:02 ; elapsed = 00:00:08 . Memory (MB): peak = 809.320 ; gain = 0.000
```

Figure 2-6 : Sortie de la console suite à la simulation du module Maj

2.2.3 Simulation du module Sigma0

La figure suivante affiche les 60 premières nanosecondes du début du chronogramme de la simulation exécutée afin de tester le module `sigma0`.

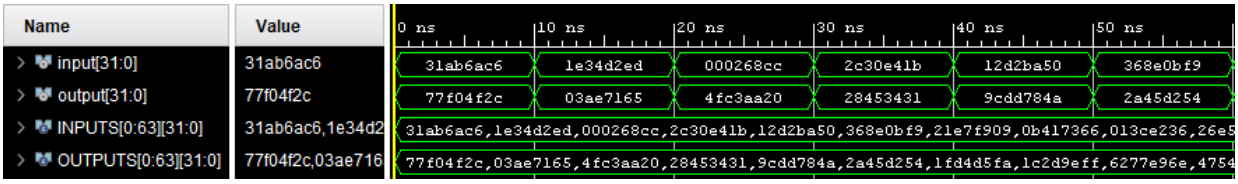

Figure 2-7 : 60 premières nanosecondes du chronogramme de la simulation du module sigma0

L'on constate effectivement que pour les mêmes premières valeurs d'entrée entre le vecteur de test et le signal simulé, les sorties correspondent, ce qui confirme le bon fonctionnement du module `sigma0` pour l'intervalle de temps représenté dans le chronogramme.

Enfin, la figure suivante contient la sortie de la console suite à 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).

```
source sigma0_tb.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window.
#   }
# }
# run 1000ns
Error: Simulation successful
Time: 640 ns  Iteration: 0  Process: /sigma0_tb/line__163  File: Y:/H2020/INF3500_Personel/Vivado/project_1/pr
INFO: [USF-XSim-96] XSim completed. Design snapshot 'sigma0_tb_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:02 ; elapsed = 00:00:09 . Memory (MB): peak = 823.230 ; gain = 0.000
```

Figure 2-8 : Sortie de la console suite à la simulation du module sigma0

2.2.4 Simulation du module Sigma1

La figure suivante affiche les 60 premières nanosecondes du début du chronogramme de la simulation exécutée afin de tester le module `sigma1`.

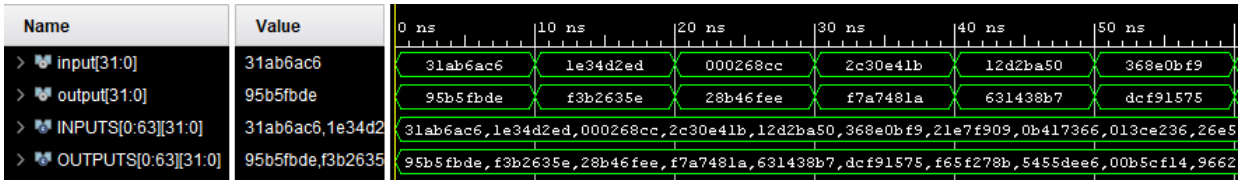

Figure 2-9 : 60 premières nanosecondes du chronogramme de la simulation du module sigma1

L'on constate effectivement que pour les mêmes premières valeurs d'entrée entre le vecteur de test et le signal simulé, les sorties correspondent, ce qui confirme le bon fonctionnement du module `sigma1` pour l'intervalle de temps représenté dans le chronogramme.

Enfin, la figure suivante contient la sortie de la console suite à 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).

```
source sigma1_tb.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window.
#   }
# }
# run 1000ns
Error: Simulation successful
Time: 640 ns  Iteration: 0  Process: /sigma1_tb/line__163  File: Y:/H2020/INF3500_Personel/Vivado/project_1/pr
INFO: [USF-XSim-96] XSim completed. Design snapshot 'sigma1_tb_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:01 ; elapsed = 00:00:09 . Memory (MB): peak = 823.230 ; gain = 0.000
```

Figure 2-10 : Sortie de la console suite à la simulation du module sigma1

2.2.5 Simulation du module Sigma2

La figure suivante affiche les 60 premières nanosecondes du début du chronogramme de la simulation exécutée afin de tester le module `sigma2`.

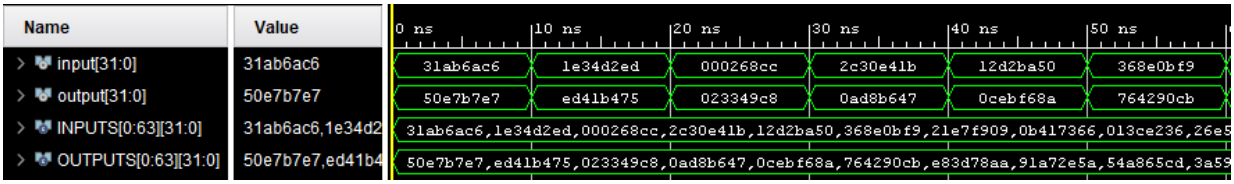

Figure 2-11 : 60 premières nanosecondes du chronogramme de la simulation du module sigma2

L'on constate effectivement que pour les mêmes premières valeurs d'entrée entre le vecteur de test et le signal simulé, les sorties correspondent, ce qui confirme le bon fonctionnement du module `sigma2` pour l'intervalle de temps représenté dans le chronogramme.

Enfin, la figure suivante contient la sortie de la console suite à 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).

```
source sigma2_tb.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window.
#   }
# }
# run 1000ns
Error: Simulation successful
Time: 640 ns  Iteration: 0  Process: /sigma2_tb/line__164  File: Y:/H2020/INF3500_Personel/Vivado/project_1/p
INFO: [USF-XSim-96] XSim completed. Design snapshot 'sigma2_tb_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:03 ; elapsed = 00:00:09 . Memory (MB): peak = 830.012 ; gain = 6.781
```

Figure 2-12 : Sortie de la console suite à la simulation du module sigma2

2.2.6 Simulation du module Sigma3

La figure suivante affiche les 60 premières nanosecondes du début du chronogramme de la simulation exécutée afin de tester le module `sigma3`.

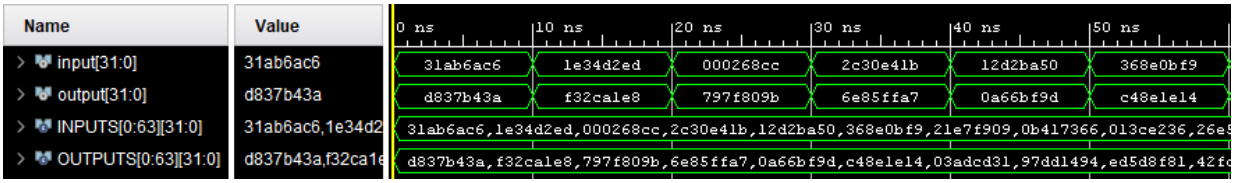

Figure 2-13 : 60 premières nanosecondes du chronogramme de la simulation du module sigma3

L'on constate effectivement que pour les mêmes premières valeurs d'entrée entre le vecteur de test et le signal simulé, les sorties correspondent, ce qui confirme le bon fonctionnement du module `sigma3` pour l'intervalle de temps représenté dans le chronogramme.

Enfin, la figure suivante contient la sortie de la console suite à 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).

```
source sigma3_tb.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window.
#   }
# }
# run 1000ns
Error: Simulation successful
Time: 640 ns  Iteration: 0  Process: /sigma3_tb/line__163  File: Y:/H2020/INF3500_Personel/Vivado/project_1/pr
INFO: [USF-XSim-96] XSim completed. Design snapshot 'sigma3_tb_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:02 ; elapsed = 00:00:09 . Memory (MB): peak = 849.500 ; gain = 0.000
```

Figure 2-14 : Sortie de la console suite à la simulation du module sigma3

3 Réponses aux questions

3.1 Question 1

De quelle façon pourriez-vous déterminer que votre programme générateur de tables fonctionne? En d'autres mots, pourquoi faites-vous confiance en votre programme pour générer les bonnes valeurs?

On pourrait calculer à la main certaines valeurs précise et vérifier si le résultat est bon. On peut aussi analyser des valeurs générées aléatoirement, et voir le résultat qui y est associé, et refaire les calculs associés. Ceci étant dit, nous n'avons pas validé le fonctionnement de notre générateur de tables manuellement.

La confiance portée au programme générateur de tables est basée sur deux éléments. D'une part, le programme générateur de table est programmé dans un langage de programmation qui est davantage maitrisé que le VHDL et plus simplement manipulable, soit le C++. Par conséquent, le résultat d'une fonction écrite en C++ est plus prévisible que le résultat de la même fonction pour laquelle une implémentation en VHDL aurait été écrite. Ceci est donc la première raison pour laquelle on fait confiance au générateur de table écrit en C++.

L'autre élément qui permet de faire confiance au générateur de table est, paradoxalement, le fait que la sortie du générateur de table ainsi que celle du module que l'on souhaite tester correspondent. Ainsi, puisque les deux correspondent, cela a pour effet de confirmer le fonctionnement du module et également d'augmenter le niveau de confiance porté dans le générateur de table puisqu'il est peu probable que les deux éléments correspondent si l'un d'eux possédait une erreur.

3.2 Question 2

Copier/coller les valeurs générées est une tâche fatigante. De quelle(s) façon(s) pourrait-on faire un banc d'essai, sans avoir à copier/coller de nouvelles tables?

Pour automatiser cette tâche, il pourrait être intéressant de modifier directement depuis le logiciel développé les fichiers du banc d'essai. Il est aussi possible de faire cela au travers d'un script, mais le logiciel peut sans trop de difficulté réaliser cette tâche.

Autrement, il serait également possible de directement générer le banc de test en entier à partir du générateur de vecteur de tests. Ainsi, le résultat serait un banc de test directement utilisable, sans nécessité de copier/coller des tables.

3.3 Question 3

Quel(s) avantage(s) (autre que celui mentionné plus haut) offre la génération de valeurs de tests aléatoires?

La génération de valeur aléatoire permet d'augmenter l'intervalle de confiance dans le module testé : si un grand ensemble de valeur, n'ayant aucun lien entre elles, fonctionne, la probabilité qu'il y ait un bug dans le module est faible.

De plus, l'approche par valeurs aléatoires permet d'éviter une approche par test exhaustif de l'ensemble des valeurs possibles. Notamment, dans le cas d'un module de 3 entrées à 32 bits, cela représenterait un ensemble de $3 \times 2^{32} = 12\ 884\ 901\ 888$ valeurs possibles !