

Département de génie informatique et génie logiciel

INF8500

Conception et vérification des systèmes embarqués

**Rapport de laboratoire #2**

**Apprentissage de SystemVerilog et de concepts de base**

**pour la vérification fonctionnelle**

Soumis par

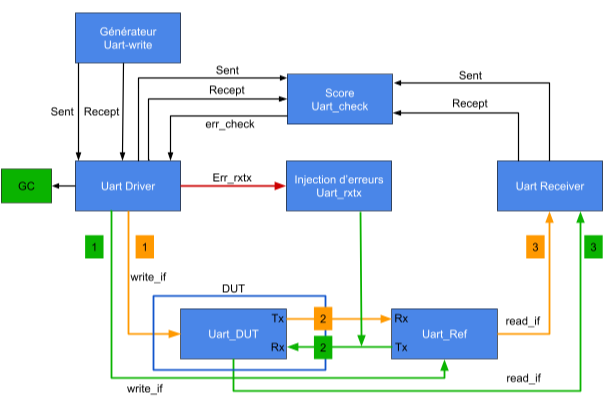
Nicolas Lauzon, 1959682

Timothée Laborde, 1782257

Le 12 décembre 2021

# 3. Réalisations à compléter

## 3.1) Analyse et compréhension du DUT



#### *a) Décrivez en 2-3 lignes le rôle de chacun des bloc (classe) du test bench (Uart\_cover\_group, Config, InjectionErrType, Uart\_config, Uart\_rxtx, Uart\_driver, Uart\_receiver, Uart\_write et Uart\_check) et complétez le schéma de la figure 4 en spécifiant bien les blocs, les fifos (mailbox) et méthodes d’interface qui relient les blocs.*

Uart\_cover\_group est la classe qui contient tous les cover points de nos tests. Ces cover points nous permettent de définir les valeurs des variables que nous voulons tester. De plus, Cette classe fait la configuration et l’injection d’erreurs.

La classe config indique que les variables sont soit random (rand) ou random cyclique (randc). Cette classe est appelée pour choisir au hasard des valeurs définies dans les bins de la classe Uart\_cover\_group.

La classe InjectionErrType permet d’injecter des erreurs dans la communication entre le Uart et Uart\_DUT. Cette classe injecte seulement une erreur si: parité !=None. Si le bit de parité n’est pas à None, La classe va changer un des bit entre 1 et 8.

La classe Uart\_config permet de retourner le baudrate et permet de set les bit du registre de contrôle selon la parité. Cette classe permet aussi d’afficher le baudrate actuel et le bit de parité actuel.

La classe Uart\_rxtx s’occupe de faire la communication entre Uart et Uart\_DUT. La classe s’occupe aussi de faire l’injection d'erreurs dans ses communications et faire la gestion de ses erreurs tant sur la parité que sur les données.

Uart\_driver s’occupe de d’initialiser les Uart et des les configurer. La classe s’occupe d’envoyer les données au Uar et Uart\_DUT à l’aide des méthodes write\_if. Plusieurs malbox sont utilisés pour la communication : send\_data, recept\_data, send\_test, recept\_test, err\_rxtx, err\_check qui permettent de communiquer avec les autres modules tels que: Uart\_write, Uart\_check, Uart\_rxtx, Uart\_receiver.

La classe Uart\_receiver sert à recevoir les résultats des tests de transmission et de réception. Cette classe utilise la méthode read\_if pour communiquer avec le Uart. Ensuite, la classe met les résultats dans le malbox send\_test, recept\_test pour les envoyer au Uart\_check

La classe Uart\_write sert à générer aléatoirement des données à envoyer. Ces données sont mises dans les mailbox envoi, réception pour que Uart\_driver les envoie par la suite aux deux Uart (soit pour test de transmission ou de réception).

La classe Uart\_check sert à vérifier que les tests d’erreurs d’injection de parité de trame ont bien été envoyés et enregistrés (détectés) par le système. La classe se sert de 4 mailbox pour communiquer avec Uart\_receiver et Uart\_driver pour recevoir les données initiales et finales puis une mailbox pour envoyer les erreurs à Uart\_driver pour le calcul d'erreur total.

#### *b) Dans la figure 2, le Tx et Rx du UART DUT sont reliés via un loopback. Dans la figure 4, nous utilisons un UART de référence. Quel est le rôle de l’UART de référence ?*

Le Uart de référence permet d’avoir un modèle que nous savons qui est fonctionnel. De plus, l’utilisation d’un modèle de référence permet de tester la communication entre deux Uart en faisant des tests de transmission et de réception sur un Uart\_DUT.

#### *c) Si cela n’a pas été fait en a), décrivez et justifiez le rôle des variables aléatoires (rand et randc) et leur contrainte (constraint). N’oubliez pas que le testbench est basé sur de la génération pseudoaléatoire.*

Les variables parite et baudrate sont définies respectivement par rand (random) et randc (random cyclique). Parite permet de déterminer si lors d’une transmission un seul bit a été changé. Baudrate permet de déterminer la fréquence ou la vitesse de transmission. Leur contrainte est que s’il n’y a pas de bit de parité, on ne peut pas déterminer d’erreur, sinon le bit a changer se trouve entre 1 et 8.

#### *d) Justifiez l’utilisation des sémaphores sem\_dut et sem\_ref. Notez qu’aucun sémaphore n’est utilisé dans le code de la Figure 2.*

Les sémaphores sem\_dut et sem\_ref permettent d’isoler les transmissions du driver et les réceptions du receiver pour pas qu’elles se déroulent en simultané.

#### *e) Vous remarquerez que l’injection de fautes n’est pratiquée que pour la réception du dut (et évidemment la transmission du UART de référence). Y aurait-il eu un intérêt à le faire également pour la réception du UART de référence (et la transmission du dut)? Justifiez*.

Il n’y a pas d'intérêt à injecter des erreurs dans la réception du UART de référence car celui-ci nous sert comme indiqué de référence afin de déterminer si l’UART DUT a détecté les erreurs de parité et de data.

#### *f) Quelle(s) classe(s) a-t-elle (ont-t-elles) accès à la classe Uart\_cover\_group? Et comment?*

Uart\_config utiliser la fonction set\_config pour définir la vitesse de transmission et le bit de parité.

Uart\_driver appelle la fonction do\_cover qui permet d’utiliser les différents bins définits dans la classe pour faire les tests. Cette classe utilise aussi la fonction set\_err\_injection opur l’injection des erreurs pour les tests de réception.

Uart\_receiver Utilise les fonctions do\_cover pour vérifier les valeurs qui viennent des bins.

## 3.2) Raffinement du groupe de couverture et automatisation pour la terminaison du programme test\_uart

#### *2) Lorsque vous faites le croisement, avez-vous rencontré d’autres situations que Tx (DUT) vs Rx (UART de référence) ou vice versa? Plus précisément avez-vous rencontré Rx (DUT) vs Rx (UART de référence)? Si oui, comment est-ce possible?*

#### Non, il ne doit pas y avoir d’autres croisements. Les Uart ne doivent pas envoyer des données au même moment ni recevoir des données au même moment. Les tests bin56 et bin57 ont été fait, mais ne fonctionnent pas. Ce qui n’est pas important puisque ces tests sont inutiles puisqu’ils test des situation impossibles. Au total, nous avons 55 tests.

#### *3) Plutôt que d’avoir à décider du nombre de test avec le repeat (ligne 65 du fichier test\_uart.sv), automatisez la fin du test : lorsque la couverture est atteinte, le testbench d’arrête tout seul (e.g. avec utilisation de fork). Combien de transmissions furent nécessaires pour y arriver, c’est-à-dire pour avoir 100% de couverture? Bonne pratique : Assurez-vous toutefois de mettre un nombre maximal d’itérations possibles afin que votre simulation ne soit pas prise dans une boucle infinie.*

Les tests ont nécessité 66 itérations de la boucle pour avoir une couverture de 100%. Pour ce faire, nous avons ajouté un int qui sert de compteur pour le nombre d’itération. Nous avons aussi mis le code dans une boucle while qui dépend de deux conditions: le code coverage est inférieur à 100% et le nombre d’itération est inférieur ou égal à 1000. La première condition s’assure que les tests s’arrêtent lorsque la couverture est complète. La deuxième condition sert à ne pas rester dans une boucle infinie si la couverture de code n’atteint jamais 100%.

.