INF8500 – Vérification et conception des systèmes embarqués

Automne 2019

Laboratoire #2

Création d'un environnement de vérification fonctionnelle sous SystemC avec CRAVE et FC4SC

Objectifs

L'objectif de ce deuxième laboratoire est de poursuivre l'apprentissage de SystemC en se familiarisant avec les concepts et les notions de base de bancs d'essai (testbench) pour la vérification fonctionnelle. Plus précisément, les objectifs sont :

1. Comprendre la structure et le rôle des différents blocs d'un testbench (Figure 1)
2. Concevoir un testbench de base et l'intégrer à un DUT existant (laboratoire no 1)
3. Mettre l'emphase sur les concepts suivants du testbench:
    - génération de tests pseudo aléatoires avec CRAVE
    - couverture fonctionnel avec FC4SC

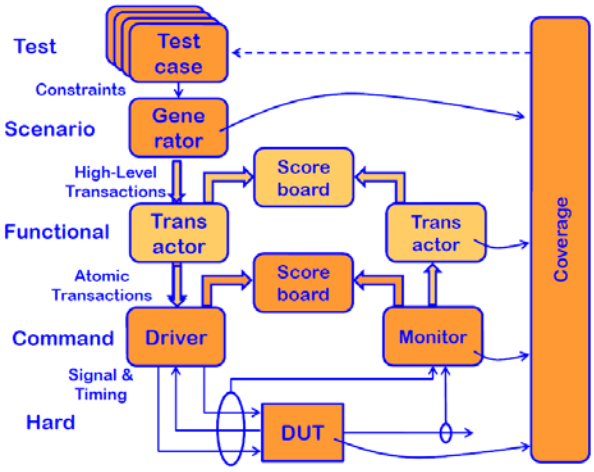

Figure 1 Structure d'un testbench

La figure 2 suivante illustre le système résultant que vous devrez réaliser.

Plan de test (300 tests)

#test (nbre requis)	Copro (type de trie)	Ordre des données	Ordre demandé du tri
1	1	random_desc	up
2	1	random_asc	down
3	1	random_full	up
4	1	continues_asc	down
5	1	continues_desc	up
6	1	random_desc	down
7	1	random_asc	up
8	1	random_full	down
9	1	continues_asc	up
10	1	continues_desc	down
11	2	random_desc	up
12	2	random_asc	down
13	2	random_full	up
14	2	continues_asc	down
15	2	continues_desc	up
16	2	random_desc	down
17	2	random_asc	up
18	2	random_full	down
19	2	continues_asc	up
20	2	continues_desc	down
21	3	random_desc	up
22	3	random_asc	down
23	3	random_full	up
24	3	continues_asc	down
25	3	continues_desc	up
26	3	random_desc	down
27	3	random_asc	up
28	3	random_full	down
29	3	continues_asc	up
30	3	continues_desc	down

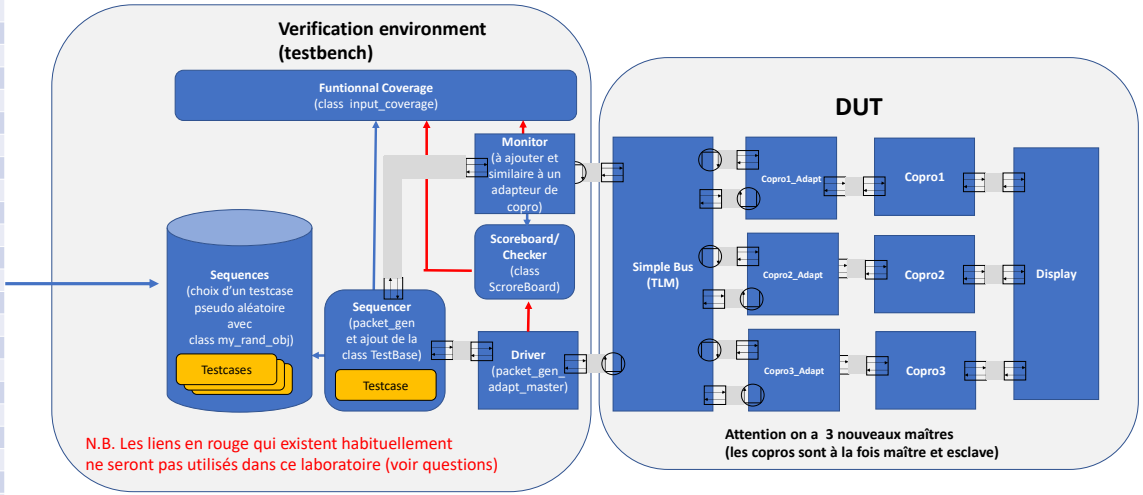

Figure 2 Schéma résultant

Voici d'abord ce qui vous sera fourni :

1) Un coprocesseur qui fait un tri en bulles. On pourrait supposer 3 types de tri, mais pour simplifier le travail on va réutiliser le même tri pour chaque processeur, soit le tri en bulles.

2) On va également vous fournir une nouvelle version de la classe paquet :

   Paquet ---> ID (32 bits) | adresse (32 bits) | direction du tri (32 bits) | payload (512 bits) |

   La classe paquet contiendra aussi de nouvelles fonctions *putPacket* (utilisé pour insérer le résultat du tri avant son envoi au moniteur) et *getDir()* pour extraire la direction du tri (notez que *getPacket* existe déjà). Finalement on va aussi vous fournir le code du coprocesseur qui après être reçu de l'adaptateur (*ack* = true) va extraire du paquet le payload à trier, va appeler le trie en bulles et va remettre le résultat dans le payload (il pourra donc être imprimé par *display* avant d'être renvoyé à *monitor*).

3) Le scoreboard/checker[1] qui doit vérifier si le tri est correctement complété. Ici on va simplement vérifier les adresses du tableau avant le tri et après le tri. Si ceux-ci sont les mêmes, on va vérifier si le tableau est en ordre croissant ou décroissant (selon la direction demandée). Évidemment, ce n'est pas l'unique façon de faire cette vérification.

Voici ce que vous devez compléter/intégrer/améliorer :

4) Deux classes qui vont s'ajouter à *packet gen* (qui lui devient un séquenceur dans le contexte du testbench) :

   - La classe *my_rand_obj* qui génère pseudo aléatoirement les testcase du plan de test (on souhaite autant que possible éviter les tests dirigés). Ici on demande :
     - une plage de mémoire de 0 à 255 octets (8 bits) : 0 à 255 pour copro1, 256 à 511 pour copro2 et 512 à 767 pour copro3. Vous devez donc ajouter des contraintes pour éviter de déborder. Référez-vous à l'exemple 7 (crave2_api/ex7_rand_enum).
     - On souhaite aussi distribuer les tests sur les 3 coprocesseurs de la manière suivante : copro1 50% du temps, copro2 25% du temps et copro3 25% du temps. Référez-vous à l'exemple 1 (crave2_api/ex1_seed_dist).

[1] Compare les transactions envoyées aux coprocesseurs et celles récupérées par le moniteur. On verra en classe leurs fonctionnalités plus en détails.

- La classe *TestBase* (fichier *testcase.h*) qui à partir du résultat de 4) (i.e., l'ordre des données à trier) va générer la bonne séquence de données. Ici je vous donne une première solution et je vous demande de me suggérer des améliorations, surtout pour l'ordre ascendant random et l'ordre descendant random. Référez-vous à l'exemple 10 (experimental_api/ex10_variable_prev).

Toujours concernant *packet_gen*, vous devrez aussi considérer une connexion avec le module monitor afin de lui transférer l'information sur le *testcase* (pointeur sur le tableau et direction du tri). Afin de faciliter le déverminage, commencez par une connexion comme celle de *copro1* et son *adapteur*. Ainsi, une seule transaction sera acheminée à la fois. Par la suite on pourrait penser à un *sc_fifo* (on pourrait aussi utiliser le ID du paquet si plusieurs paquets transitent en même temps).

5) *Packet_gen_adapt_master* va devenir un driver[2] dans le contexte de testbench. Il faudra simplement l'adapter à la nouvelle longueur des paquets.

6) Les 3 adaptateurs de coprocesseurs devront hérités d'une interface maître afin d'envoyer à au module moniteur le résultat d'un tri (eh oui! un module peut être à la fois maître et esclave). Aussi, pour éviter que *packet_gen* parte en boucle, je suggère de donner une plus grande priorité aux adaptateurs de coprocesseurs.

Voici ce que vous devrez compléter :

1) L'ajout d'un moniteur[3]. Ce dernier va recevoir un paquet qui est le résultat d'un tri (payload). Il devra extraire le payload (pointeur prt_lgintr) tout comme dans *copro1.cpp*, puis récupérer de *packet_gen* le payload initial (*ptr_lgint_r*) et faire une comparaison avec la fonction *scb.check_lgint(ptr_lgint_r,ptr_lgint,direction);*

2) Vous devrez finalement créer une classe *input_coverage* avec FC4SC afin de récupérer différent *bins* sur les entrées. Il sera ainsi possible de s'assurer que notre couverture est complète (voir plan de test figure 2). Possiblement en faisant des croisements de sondes.

   Il faudrait aussi vérifier qu'elles sont les cas limites d'adressage (p.e. 0 ou encore 255-(12+16*4) pour copro1, etc.) qui ont été testés. Si aucun d'eux n'est couvert, on pourrait alors avoir recours à la distribution (on en reparle en classe).

[2] Il transforme le *testcase* en appel(s) de méthode(s) (e.g. read & write) avec ou sans rafale, bloquant ou non bloquant, etc.
[3] Le moniteur récupère les transactions exécutées par le module sous test (DUT) ici un des coprocesseurs.

Pour cette partie, référez-vous à l'exemple FIR vu en classe. Je vous donnerai d'ici peu, ce que je souhaite voir exactement (surtout que cette partie est en lien avec le rapport).

Rapport, questions et date de remise:

Je communiquerai d'ici peu cette information. J'ajusterai en tenant compte du délai du lab 1.

Guy Bois, prof. GIGL
Responsable du INF8500