HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Reconfigurable & embedded
Digital Systems

Laboratoire 1

Analyseur de paquets

Départements : TIC
Unité d'enseignement VSE

Auteurs : **Urs Behrmann**
Professeur : **Yann Thoma**
Assistant : **Clément Dieperink**
Classe : **VSE**
Salle de labo : **A07**
Date : **03.11.25**

Contents

1 Introduction

Ce rapport présente les choix effectués lors du développement du testbench SystemVerilog `packet_analyzer_tb.sv`, utilisé pour vérifier le composant VHDL `packet_analyzer`. Le testbench a pour but de générer des paquets de test, de calculer les résultats attendus, et de vérifier que le composant fonctionne correctement en comparant ses sorties avec les valeurs de référence.

2 Architecture du testbench

Le testbench est organisé en trois parties principales qui tournent en parallèle :

2.1 Génération des stimuli

La tâche `generator` crée les paquets de test. On a utilisé une classe `Packet` avec des contraintes aléatoires (randomisation) pour générer automatiquement différents types de paquets. Cette approche est beaucoup plus efficace que de coder manuellement chaque test.

2.2 Modèle de référence

La tâche `compute_reference` calcule ce que le DUV devrait produire comme résultat. Elle analyse le paquet envoyé et détermine :

- Le type du paquet (valide ou non)
- La longueur des données
- Les erreurs détectées (CRC, type, adresses, groupe)

Ce modèle tourne en continu et calcule la référence à chaque cycle d'horloge.

2.3 Vérification

La tâche `verification` compare les sorties du DUV avec les valeurs calculées par le modèle de référence. Elle utilise des assertions SystemVerilog pour détecter les différences et compte le nombre d'erreurs.

3 Choix techniques et justifications

3.1 Classe Packet avec contraintes

La classe `Packet` utilise les fonctionnalités de randomisation de SystemVerilog. Les contraintes définissent :

- Les types valides (1, 2, 5) et invalides (0, 3, 4, 6, 7)
- Les longueurs selon le type de paquet
- Les adresses sources et destinations (valides dans les groupes 0 et 1, ou invalides)
- La possibilité de forcer des erreurs CRC (20% du temps)

Ce système permet de générer automatiquement des milliers de tests différents sans avoir à les écrire manuellement. Le choix d'utiliser la randomisation contrainte plutôt que des boucles manuelles vient du fait qu'on peut explorer beaucoup plus de cas en moins de temps. En définissant des distributions (par exemple 80% d'adresses valides, 20% invalides), on s'assure de tester les cas normaux majoritairement tout en incluant assez de cas d'erreur pour les détecter.

De plus, la randomisation aide à trouver des bugs qu'on n'aurait pas imaginés. Quand on écrit des tests manuellement, on teste ce qu'on pense être important, mais on rate souvent des combinaisons subtiles. La génération aléatoire peut créer des séquences inattendues qui révèlent des problèmes.

3.2 Vérification par assertions

Au lieu d'utiliser des `if` avec des `$display`, on a utilisé des assertions SystemVerilog :

- `assert_error` : vérifie les 5 bits d'erreur [4:0]
- `assert_type` : vérifie le type détecté
- `assert_length` : vérifie la longueur calculée

Les assertions sont plus claires et permettent aux outils de simulation de mieux identifier les problèmes. On compare uniquement les bits `error[4:0]` car les bits supérieurs ne sont pas utilisés dans notre implémentation.

Le choix des assertions plutôt que des `if` simples a plusieurs avantages : premièrement, les assertions ont des noms explicites qui apparaissent directement dans les logs d'erreur

(on sait immédiatement si c'est l'erreur, le type ou la longueur qui pose problème). Deuxièmement, les outils de simulation peuvent générer des rapports d'assertions automatiques, ce qui facilite le débogage. Enfin, c'est la méthode standard dans l'industrie pour la vérification, donc c'est une bonne pratique à adopter.

3.3 Synchronisation posedge/negedge

Le testbench utilise une synchronisation importante : la génération et le calcul de référence se font sur le `posedge` de l'horloge, mais la vérification se fait sur le `negedge`. Pourquoi ?

Parce que le DUV est combinatoire : il faut lui laisser le temps de propager les signaux. Si on vérifie trop tôt (sur le même front montant), on risque de lire des valeurs qui ne sont pas encore stabilisées. Le décalage d'une demi-période évite ce problème.

3.4 Utilisation d'un program

On a mis la logique principale dans un bloc `program` au lieu d'un simple `initial`. Le `program` garantit que notre code de test s'exécute dans la bonne région temporelle (testbench scheduling region), ce qui évite les courses avec le DUV.

4 Stratégie de tests

On a combiné deux approches :

4.1 Tests dirigés (directed tests)

Six tests spécifiques qui ciblent des scénarios particuliers :

1. `test_invalid_types` : teste tous les types invalides
2. `test_crc_errors` : force des erreurs CRC sur des paquets valides
3. `test_source_errors` : teste des adresses sources hors des groupes
4. `test_destination_errors` : teste des adresses destination invalides
5. `test_group_mismatch` : teste source et destination dans des groupes différents
6. `test_multiple_errors` : combine plusieurs types d'erreurs

Chaque test génère 100 paquets avec des contraintes spécifiques. Ces tests garantissent qu'on couvre bien tous les cas de base.

Le choix de faire des tests dirigés d'abord est important : on veut s'assurer que chaque type d'erreur est bien détecté individuellement avant de lancer des tests aléatoires. Si on détecte un problème dans un test dirigé, on sait exactement quelle fonctionnalité est cassée. De plus, 100 paquets par test est un bon compromis : assez pour avoir confiance que ça marche, mais pas trop pour garder des temps de simulation raisonnables.

4.2 Tests aléatoires guidés par la couverture

Après les tests dirigés, `test_random_coverage` génère des paquets complètement aléatoires jusqu'à atteindre 99% de couverture fonctionnelle (ou 10000 paquets maximum). Cette approche permet de découvrir des combinaisons qu'on n'aurait pas pensé à tester manuellement.

L'objectif a été fixé à 99% plutôt que 100% car, bien qu'il soit possible d'atteindre 100%, cela ne semble pas réaliste dans un contexte réel de vérification. Il reste toujours quelques bins difficiles à atteindre aléatoirement et 99% représente déjà une excellente couverture.

5 Couverture fonctionnelle

On a défini un `covergroup` qui mesure :

5.1 Coverpoints de base

- `cp_type` : types valides vs invalides
- `cp_length` : longueurs petites/moyennes/grandes
- `cp_error_crc`, `cp_error_type`, etc. : chaque bit d'erreur
- `cp_source_addr`, `cp_dest_addr` : adresses aux frontières des groupes

5.2 Crosses (croisements)

- `cr_type_error` : croisement type × erreur de type
- `cr_errors` : croisement de tous les bits d'erreur
- `cr_address_groups` : croisement des groupes source × destination

5.3 Ignore bins

On a utilisé `ignore_bins` pour exclure les combinaisons impossibles :

- Type valide avec erreur de type (impossible par définition)
- Erreur CRC sur un type invalide (le CRC n'est pas vérifié)
- Erreur de groupe avec source ou destination invalide

Cela donne un pourcentage de couverture réaliste qui ne compte que les bins atteignables.

Le choix d'utiliser `ignore_bins` est crucial pour avoir une métrique de couverture honnête. Sans ces exclusions, on aurait un pourcentage de couverture qui inclut des bins qu'on ne peut jamais atteindre, ce qui donnerait une fausse impression de qualité insuffisante. Par exemple, le DUV ne vérifie pas le CRC quand le type est invalide, donc il est logiquement impossible d'avoir à la fois "type invalide" et "erreur CRC". En excluant ces combinaisons, on mesure vraiment ce qui est testable, pas ce qui est théoriquement possible.

5.4 Tests de frontières

On teste spécifiquement les adresses aux limites des groupes (début, fin, juste avant, juste après) car c'est là qu'on trouve souvent des bugs de comparaison (erreurs de type "off-by-one").

Cette décision vient de l'expérience : les bugs de comparaison (utiliser `<` au lieu de `<=`, ou se tromper d'un dans une limite) sont très fréquents. En testant explicitement les valeurs aux frontières, on augmente beaucoup les chances de les détecter. Par exemple, si le groupe 0 va de 100 à 199, on teste spécifiquement 100, 101, 198, 199, et aussi 99 et 200 pour vérifier qu'ils sont bien rejetés.

6 Décisions d'implémentation

6.1 Paramètre TESTCASE

Le testbench accepte un paramètre `TESTCASE` qui permet de choisir quels tests exécuter :

- `TESTCASE = 0` : exécute tous les tests (1 à 7) en séquence
- `TESTCASE = 1-7` : exécute uniquement le test spécifié

Cette flexibilité est utile pour déboguer un test spécifique sans avoir à attendre l'exécution complète de tous les autres. Pendant le développement, on peut isoler un test qui pose problème, et pour la validation finale, on lance tous les tests ensemble.

6.2 Logs minimaux

On a désactivé l'affichage détaillé de chaque paquet testé car ça génère des milliers de lignes inutiles. On garde seulement :

- Les messages de début/fin de chaque test
- Les erreurs détectées
- La progression tous les 1000 paquets dans la phase aléatoire
- Le résumé final (nombre de tests, erreurs, couverture)

Cette décision vient d'une contrainte pratique : avec des logs détaillés pour chaque paquet, un test de 10000 paquets génère un fichier de plusieurs mégaoctets, ce qui ralentit la simulation et rend le transcript illisible. En gardant uniquement les informations importantes (début/fin de test, erreurs, progression), on obtient un transcript clair où on voit immédiatement s'il y a des problèmes, sans être noyé dans les détails. Les logs de progression tous les 1000 paquets permettent de suivre l'avancement sans surcharger l'affichage.

7 Conclusion

Les principaux choix du testbench sont :

- **Randomisation contrainte** : génération automatique de milliers de tests variés
- **Assertions** : vérification claire et automatique des résultats
- **Couverture fonctionnelle** : mesure objective de la qualité des tests
- **Mix dirigé/aléatoire** : garantit la couverture des cas importants tout en explorant largement
- **Synchronisation soignée** : évite les races conditions sur un design combinatoire

Cette approche permet de tester efficacement le composant avec un bon niveau de confiance dans les résultats.

8 Note sur l'utilisation d'outils IA

ChatGPT a été utilisé pour corriger l'orthographe et la grammaire de ce document.

A Exemple d'exécution

Voici un exemple de sortie de simulation lors de l'exécution complète des tests :

```
# packet_analyzer_tb: INFO: (@ 0) ======================================
# packet_analyzer_tb: INFO: (@ 0) Starting Packet Analyzer Verification
# packet_analyzer_tb: INFO: (@ 0) ======================================
# packet_analyzer_tb: INFO: (@ 0) TESTCASE: 0, ERRNO: 1
# packet_analyzer_tb: INFO: (@ 0) ======================================
# packet_analyzer_tb: INFO: (@ 5) ======================================
# packet_analyzer_tb: INFO: (@ 5) Running ALL test cases
# packet_analyzer_tb: INFO: (@ 5) ======================================
# packet_analyzer_tb: INFO: (@ 5) === Test Case 1: Invalid Type Packets ===
# packet_analyzer_tb: INFO: (@ 1005) === Test Case 1 Complete ===
# packet_analyzer_tb: INFO: (@ 1005) === Test Case 2: CRC Error Packets ===
# packet_analyzer_tb: INFO: (@ 2005) === Test Case 2 Complete ===
# packet_analyzer_tb: INFO: (@ 2005) === Test Case 3: Invalid Source Address ===
# packet_analyzer_tb: INFO: (@ 3005) === Test Case 3 Complete ===
# packet_analyzer_tb: INFO: (@ 3005) === Test Case 4: Invalid Destination Address ===
# packet_analyzer_tb: INFO: (@ 4005) === Test Case 4 Complete ===
# packet_analyzer_tb: INFO: (@ 4005) === Test Case 5: Group Mismatch (different groups
# packet_analyzer_tb: INFO: (@ 5005) === Test Case 5 Complete ===
# packet_analyzer_tb: INFO: (@ 5005) === Test Case 6: Multiple Simultaneous Errors ===
# packet_analyzer_tb: INFO: (@ 6005) === Test Case 6 Complete ===
# packet_analyzer_tb: INFO: (@ 6005) === Test Case 7: Random Coverage-Driven Testing =
# packet_analyzer_tb: INFO: (@ 8415) === Test Case 7 Complete ===
# packet_analyzer_tb: INFO: (@ 8525) ======================================
# packet_analyzer_tb: INFO: (@ 8525) Simulation Complete
# packet_analyzer_tb: INFO: (@ 8525) ======================================
# packet_analyzer_tb: INFO: (@ 8525) Total tests executed: 852
# packet_analyzer_tb: INFO: (@ 8525) Total errors detected: 0
# packet_analyzer_tb: INFO: (@ 8525) Functional coverage: 99.52%
# packet_analyzer_tb: INFO: (@ 8525) ======================================
# packet_analyzer_tb: INFO: (@ 8525) *** TEST PASSED ***
# packet_analyzer_tb: INFO: (@ 8525) ======================================
```

On voit clairement :

- L'exécution séquentielle des 7 tests
- Le nombre total de tests : 852 (600 dirigés + 241 aléatoires pour atteindre 99%)
- Aucune erreur détectée (0 erreurs)

- Couverture finale de 99.52%
- Le test est passé avec succès