[INF8500] Rapport lab 2

Antonin Godard (2032402), Rayan Neggazi (2038882)

27 octobre 2019

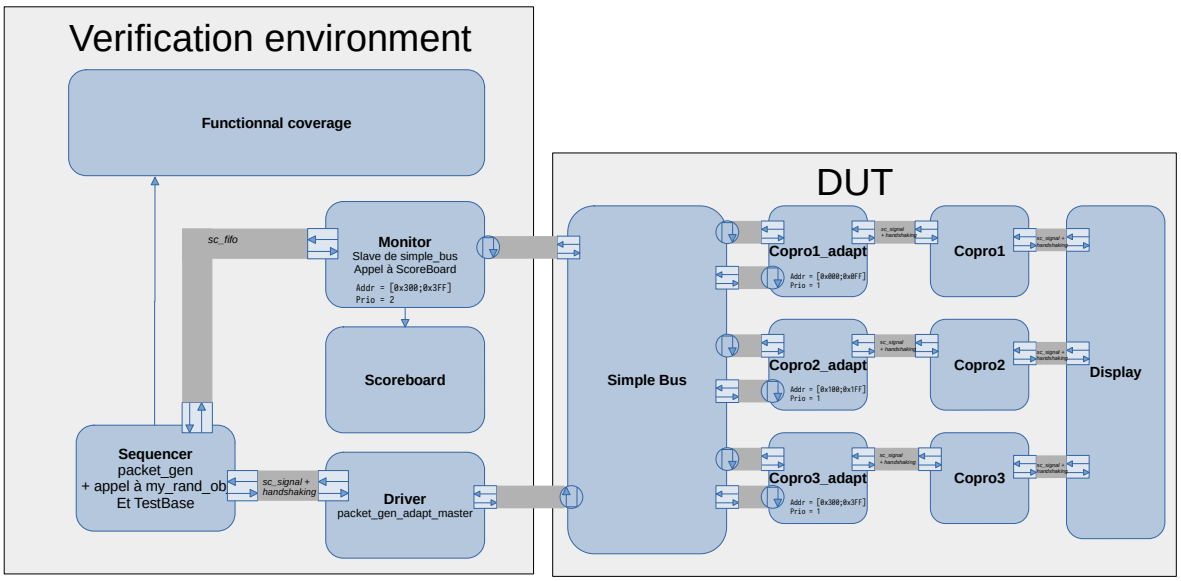

FIGURE 1 – Schema

1. Le schéma figure 1 page 1 représente notre système final (zoom possible pour les détails). On détaille chaque bloc :
   - **Functionnal coverage** est le bloc qui fait la vérification fonctionnelle du système et représente la classe TestBase. On utilise la bibliothèque FC4SC.
     Cette classe est instanciée dans `packet_gen` et on sample chaque paramètre de notre système (les copros, les différents tris et la direction du tri) directement à chaque tour de boucle.
     Nos résultats se trouvent à la figure 2 page 4. Le fichier xml est également présent à la source de notre rendu.
   - **Sequencer** est le bloc représentant `packet_gen`. Ce bloc génère les paquets qui vont être envoyés aux coprocesseurs.
     Il procède en cinq phases :
     (a) Génération d'une nouvelle addresse, d'un type de données et d'une direction de tri avec `my_rand_obj`.
     (b) Sample des valeurs générées par `my_rand_obj` pour la couverture fonctionnelle.
     (c) Génération des données destinées à être triée avec `TestBase` en fonction du type de données généré par `my_rand_obj`.
     (d) Envoi du paquet au moniteur via *un canal FIFO*, avec l'appel `nb_write` pour ne pas bloquer le générateur. [1]
     (e) Création du nouveau paquet avec les données générées précedemment ainsi que l'addresse de `my_rand_obj`, puis envoi au Driver via un canal `sc_signal` et un handshaking.
   - Le **Driver** reçoit les données du générateur (Sequencer) puis les envoient à la bonne addresse via le Simple Bus. Il fait un appel bloquant `burst_write`.

1. En fait dans notre modèle si on faisait simplement un `write` le système bloquerait car le moniteur n'effectue un `read` sur la FIFO seulement lorsqu'il reçoit un paquet des copros

— Chaque **Copro*_Adapt** agit ici comme un esclave et comme un maître. C'est un esclave car il reçoit les paquets du Driver, comme au lab 1. C'est également un maître car il envoie les données triées au Moniteur avec un appel bloquant `burst_write`.
Il envoie également les données triées au display via un `sc_signal` et un handshaking.
Il sont définis sur les plages `[0x000; 0x0FF]`, `[0x100; 0x1FF]`, et `[0x200; 0x3FF]`, et ont chacun une priorité de 1, supérieure au moniteur.
— Le **Moniteur** est un esclave qui reçoit un paquet de la part du Séquenceur ainsi que des coprocesseurs. Il a pour rôle de comparer les deux paquets en instanciant la classe `ScoreBoard` et en appelant la méthode `check_int` de cette dernière.
Il est de priorité inférieure aux adaptateurs de coprocesseurs (=2) et est défini sur la plage d'addresse `[0x300; 0x3FF]`.
— Le **ScoreBoard** compare les pointeurs de chaque paquets. Sa fonctionnalité pourrait être améliorée en comparant réellement les deux contenus de chaque payload de paquets, en triant le contenu du paquet en provenance du Séquenceur au préalable en utilisant une fonction de tri fiable.

2. Nous avons réalisé notre couverture fonctionnelle à l'aide de la librairie FC4SC. Pour le groupe de couverture nous avons créé la classe `input_cvg` comprenant les variables à couvrir : `copro`, `data_order` et `sort_dir`.

```
class input_cvg : public covergroup
{

public:
  int copro;
  int data_order;
  int sort_dir;
  int nb_de_cov = 0;
```

Le sampling est alors effectué sur l'ensemble de ces variables, on mesure le nombre de couvertures grâce à la variable `nb_de_cov` qui s'incrémente à chaque appel de « `sample` ».

```
void sample(int copro, int data_order, int sort_dir)
{
        this->copro = copro;
        this->data_order = data_order;
        this->sort_dir = sort_dir;
        covergroup::sample();
        nb_de_cov += 1;
}
```

On crée ensuite une sonde pour chaque variable grâce au macro `COVERPOINT`. Pour chacune des sondes on crée une boîte par valeur que peut prendre la variable, ainsi nous pourrons observer les occurrences de chaque valeur une par une et avoir une couverture complète.

```
COVERPOINT(int, copro_cvp, copro){
        bin<int>("copro 1", 0),
        bin<int>("copro 2", 1),
        bin<int>("copro 3", 2),
};

COVERPOINT(int, sort_dir_cvp, sort_dir){
        bin<int>("up", 1),
        bin<int>("down", 0)};
```

```
COVERPOINT(int, data_order_cvp, data_order)
    {
        bin<int>("random_desc", 0),
        bin<int>("random_asc", 1),
        bin<int>("random_full", 2),
        bin<int>("continues_asc", 3),
        bin<int>("continues_desc", 4),
};
```

Afin de couvrir l'ensemble des combinaisons possibles entre nos variables, nous ferons des mesures croisées.

```
cross<int, int, int> reset_valid_cross = cross<int, int, int>(this,
                "Croisement des 3 parametres",
                &copro_cvp,
                &data_order_cvp,
                &sort_dir_cvp);
```

Les résultats de la simulation sont présentés dans le tableau joint figure 2 page 4, nous avons obtenu 100% ce qui signifie que notre couverture a parcouru chacune des combinaisons possibles entre les trois variables.

3. Voici les améliorations que nous avons / aurions pu apporter :
— Un canal FIFO entre le Séquenceur et le Moniteur. Comme mentionné précedemment, nous effectuons un `nb_write` depuis le Séquenceur, ce qui évite de bloquer le Séquenceur (`packet_gen`). Cependant, nous aurions également pu définir *une taille* à la FIFO et également bloquer le Séquenceur lorsque celle-ci est pleine. En faisait par exemple :

```
if (packet_monitor.num_free() > 0) packet_monitor.nb_write();
```

— Le Scoreboard aurait pu être amélioré en implantant une fonction de tri à bulles préfaite (ou bien une fonction de tri classique, éventuellement plus performante pour accélerer la simulation) et en triant les données reçues du Séquenceur. Une fois ces données triées nous les comparons avec les données reçues par les coprocesseurs afin de vérifier le bon fonctionnement du tri.

— Nous avons essayé d'implanter une amélioration pour les tri Ascendant Random et Descendant Random.
Malheureusement nous obtenions une erreur que nous ne comprenons pas. Voici ce que ça aurait donné sinon :

```
struct item_Asc : public crv_sequence_item {
  crv_variable<unsigned> x;

  crv_constraint constr{ x() > x(prev), x() <= x(prev) * 2 };

  item_Asc(crv_object_name) {}
};

int * Test_Random_Asc_data_gen::runphase(int a){

  item_Asc obj("obj");
  static unsigned int n = 0;

  for(int i = 0;i<MAX;i++)
  {
    CHECK(obj.randomize());
    n += obj.x;
    iarray[i] = n;
  }
  return iarray;
}
```

L'idée est de générer un nouveau `x` en fonction de sa valeur précédente. `x` doit être compris entre lui-même et lui-même $\times$ 2, tout en ayant une valeur aléatoire. C'est seulement une façon de faire, et elle pourrait être amélioré en :

— définissant une valeur de départ assez petite (ou grande selon le tri)

— contrôlant mieux les bornes à chaque itérations

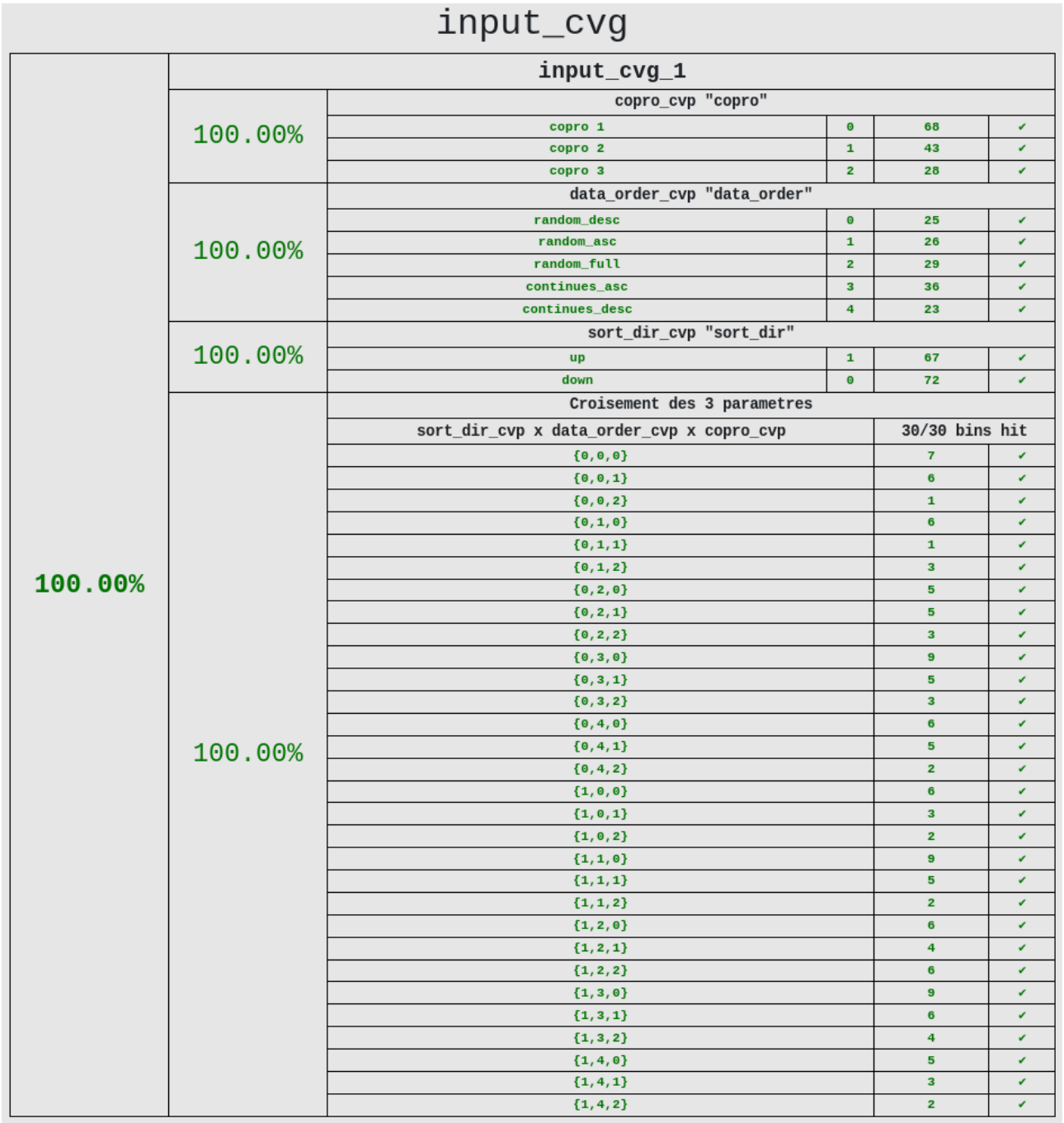

FIGURE 2 – Résultats de FC4SC