Ensimag 3A, filière SLE

2016-2017

Modélisation TLM en SystemC
TP n°1 (non noté)

Si ce n'est pas déjà fait, récupérer le dépôt Git du projet (cf la page du wiki `http://ensiwiki.ensimag.fr/index.php/TLM`). Placez-vous dans le répertoire du projet et faites un `git pull` pour vous assurer d'avoir la dernière version.

Pour ce TP et les suivants, les Makefiles fournis supposent que vous avez défini correctement un certain nombre de variables d'environnement, et que vous utilisez GNU Make. Pour avoir tout ceci, sur une machine de l'Ensimag, il suffit de faire

```
source TPs/setup-ensimag.sh
```

Si vous travaillez sur votre machine personnelle, suivez les instructions du wiki pour installer SystemC et TLM-2, puis adaptez le fichier `setup-ensimag.sh` en fonction.

Pour commencer, faites :

```
cd TPs/squelette/tp1/
make
```

Vous devriez obtenir une erreur à l'édition de liens. C'est normal, vous n'avez pas encore écrit la fonction `sc_main` dont SystemC a besoin.

Style de codage

Le TP n'est pas noté donc aucune consigne n'est imposée, mais vous pouvez dès maintenant vous familiariser avec le style de codage assez strict qui sera imposé à partir du TP2 (cf. l'énoncé du TP2).

Objectif

On cherche à créer la plate-forme (modélisation de système sur puce) représentée figure 1. Le composant `Memory` doit modéliser le comportement d'une mémoire réelle. La mémoire est

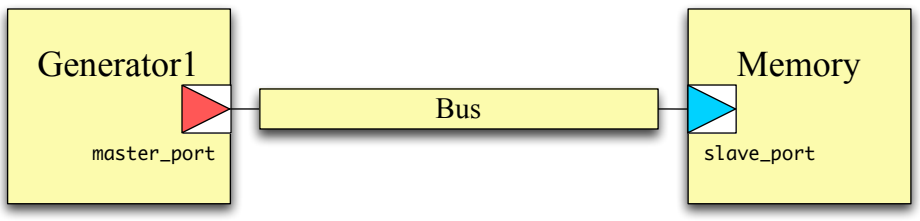

FIGURE 1 – Plateforme à réaliser

une mémoire 32 bits, on ne peut accéder qu'aux mots entiers (pas de byte-enable). Les adresses circulant sur le bus sont cependant en octets. Les accès mémoires non-alignés sont interdits (lèvent une erreur au niveau du bus). Le composant `Generator1` servant à générer des tests permettant de valider le fonctionnement de la mémoire (il s'appelle comme ceci parce qu'il ne fait rien de bien intelligent, mais qu'il *génère* du trafic).

Il est conseillé de compiler son code et de le tester régulièrement, et pas seulement quand cela est explicitement demandé dans l'énoncé, de façon à écluser au fur et à mesure les erreurs de syntaxe éventuelles.

Il est tout à fait possible (et même recommandé !) d'utiliser `git commit` pour sauvegarder périodiquement votre travail. Les prochains `git pull` récupéreront la dernière version du cours et la fusionnera avec votre travail. Vous pouvez même travailler à plusieurs avec ce dépôt Git (des explications sont disponibles sur EnsiWiki).

Préliminaire

Pour vous aider, deux exemples de code sont fournis dans le répertoire `code/` :

— `ensitlm-mini` : exemple minimaliste de connexion d'un initiateur à une cible via un bus, en un seul fichier (pas très propre, mais pratique pour avoir une vue d'ensemble).

— `ensitlm-mini-multi` : le même exemple, avec un découpage 1 classe = 1 fichier `.h` + 1 fichier `.cpp`.

Compiler et exécuter ces programmes. Parcourez rapidement le code : vous pourrez vous en inspirer pour la suite du TP.

Question 1 : création du générateur de transactions

— Créer un module SystemC `"Generator"`, comportant un socket initiateur ENSITLM « `initiator` » et un processus de type `SC_THREAD`.

Remarque : pour des raisons de clarté, ne mettre qu'une classe par fichier .h et .cpp et appeler les fichiers du nom de la classe. Ici, `generator.h` et `generator.cpp` contiendront la classe « `Generator` ».

— Dans un fichier `sc_main.cpp` :

  — Instancier le module sous le nom « `Generator1` ». (cf cours).
  — Instancier un objet de la classe `bus`, sous le nom « `Bus` ».
  — Réaliser la connexion générateur/bus.
  — Utiliser la commande `sc_start();` pour démarrer la simulation après les instanciations et les connexions.

— Compiler et tester. Pour l'instant, le générateur est connecté à un bus sur lequel aucun composant cible n'est connecté. En d'autres termes, il peut parler, mais personne n'écoute ! Vous devriez donc avoir le message d'erreur approprié.

Question 2 : création de la mémoire

— Créer un module SystemC « `Memory` », comportant un socket cible ENSITLM « `target` » (pour l'instant, il s'agit d'une coquille presque vide).

— Instancier la mémoire dans `sc_main.cpp` sous le nom « `Memory` » et la connecter au bus.

— Enregistrer la plage d'adresses [0x10000000, 0x100000FF] pour le composant Memory (à l'aide de la fonction `map()` du bus).

— Compiler et tester. Si vous n'avez pas encore défini les méthodes `read` et `write`, gcc va refuser de compiler, et il aura raison ! Pour l'instant, on se contentera d'une implémentation vide pour ces fonctions.

— Dans le module générateur, réaliser une suite de 10 écritures d'une valeur quelconque en commençant à l'adresse `0x10000000` en incrémentant à chaque pas l'adresse. Tester le statut en retour de transaction et afficher un message en cas d'erreur.

— Compiler et tester. Réessayer avec des adresses hors de la plage déclarée. Vous devriez obtenir un premier message d'erreur : le bus va refuser de router les transactions !

— Dans le module mémoire, afficher un message lorsque qu'une transaction est effectuée, en précisant le nom du composant, le type de transaction (read/write), l'adresse et la données écrite/renvoyée.
Pour afficher le nom du composant, utiliser la méthode `name()` appelable directement dans le module.

— Compiler et tester.

Question 3 : comportement de la mémoire

— Dans le module mémoire, rajouter un attribut `storage` de type `ensitlm::data_t *` (tableau de `ensitlm::data_t`).

— Modifier le constructeur de façon à ce qu'il prenne en paramètre additionnel la taille de la mémoire à construire.

— Dans le constructeur, allouer *exactement* la taille mémoire passée en paramètre (on fixe comme convention que la taille passée en paramètre est en octets, et on s'autorise à utiliser le fait que `sizeof(ensitlm::data_t) = 4`, i.e. les données sont des entiers sur 32 bits) et stocker le pointeur dans `storage`.

— Rajouter un attribut `size` pour stocker également la taille mémoire dans le module.

— Créer un destructeur pour la classe `Memory` et implémenter la libération de la mémoire avec `delete`.

— Dans l'instanciation de la mémoire (`sc_main.cpp`), fournir en paramètre additionnel la taille de la plage réservée à la mémoire.

— Compiler et tester.

— Modifier maintenant les méthodes correspondant aux accès transactions de façon à réaliser la fonctionnalité attendue : mémorisation des valeurs. Attention : un accès à l'adresse 0 renvoie, sur 32 bits, les 4 octets correspondant aux adresses 0, 1, 2, 3. Un accès aux adresses 1, 2 et 3 est interdit, un accès à l'adresse 4 renvoie les 4 octets suivants, ... Bien entendu, il ne faut pas gaspiller de mémoire. En d'autres termes, un accès à l'adresse 0 doit accéder à la case 0 du tableau `storage`, un accès à l'adresse 4 à la case 1, l'adresse 8 à la case 2, ...

Question 4 : écriture du test de la mémoire

— Modifier le code du générateur de façon à écrire des valeurs différentes à chaque adresse et à balayer l'intégralité de la mémoire.

— À la suite des écritures, réaliser des accès en lecture et vérifier automatiquement si la mémoire fonctionne correctement. Afficher un message d'erreur en cas de problème.

Question 5 : vérification d'adresse

— Diminuer la taille de la mémoire instanciée. Ne pas changer la plage d'adresse donnée au `bus` ni le générateur.

— Compiler et tester. Cette fois-ci, il est probable que la plateforme ne donne pas d'erreur lorsqu'on fait des accès en dehors de la plage allouée. En fait, on retombe sur un problème de C++ classique : un accès en dehors d'un tableau alloué peut faire n'importe quoi, y compris marcher « par chance », ou faire un « segmentation fault ». L'outil `valgrind` peut vous aider à voir ce qu'il se passe (vous aurez un warning `Warning: client switching stacks?`, vous pouvez l'ignorer).

— Modifier la mémoire de façon à inclure un test sur les adresses et à renvoyer un statut erreur en cas d'adresse invalide (trop grande).

— Compiler et tester.