Ensimag 3A, filière SEOC

2019-2020

Modélisation TLM en SystemC
TP n°1

Si ce n'est pas déjà fait, récupérer le dépôt Git du TP.

```
git clone https://gitlab.ensimag.fr/petrotf/cours-tlm.git
```

Si vous aviez déjà fait cette étape, 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 (c'est généralement le cas sous Linux). 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 (version 2.3.3), puis modifiez le fichier `setup-ensimag.sh` afin de mettre à jour les variables.

Pour commencer, faites :

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

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

Consignes importantes et conseils pour les TPs

Le fichier `TP-commun.pdf` contient un ensemble de **consignes** pour les 2 TPs. Merci donc de les suivre.

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 ajouter un nouveau `remote` dans lequel faire des sauvegardes successives en utilisant `git remote set-url`. Une lecture attentive de `man git remote` pourra vous être utile.

Objectif

Le TP se déroule en deux phases. La première vise à se familiariser avec SystemC en modélisant un tout petit système, et la seconde à modifier ce petit système en y ajoutant des composants existants.

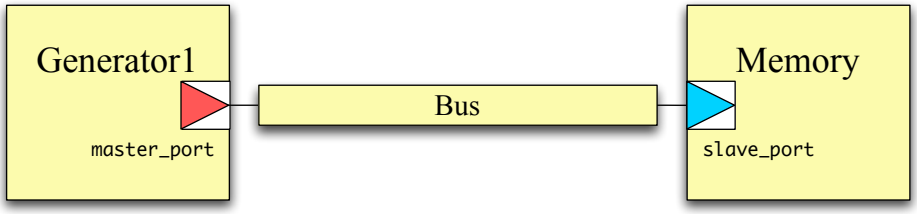


FIGURE 1 – Première plateforme à réaliser

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 (*i.e.* repartir de ce code) pour la suite du TP.

On cherche en premier lieu à créer la plate-forme (modélisation de système sur puce) représentée figure 1. On considèrera dans la suite que la taille des adresses (i.e. `ensitlm::addr_t`) et des données (i.e. `ensitlm::data_t`) est de **32 bits**.

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 implantation 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 l'état 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()` qui peut être appelée directement dans le module.

— Compiler et tester.

Question 3 : comportement de la mémoire

— Dans le module mémoire, ajouter 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`.

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

— Créer un destructeur pour la classe `Memory` et implanter 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 transactions de façon à réaliser la fonctionnalité attendue d'une mémoire : mémoriser 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 par programme que la mémoire fonctionne correctement. Afficher un message d'erreur en cas de problème.

Question 5 : vérification de la légalité d'un accès à la mémoire

— 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 classique du C++ (entre autres) : un accès en dehors d'un tableau alloué peut faire n'importe quoi, y compris marcher « par chance », ou lever une exception de type (généralement) *segmentation fault*[1]. Dans ce cas, `gdb` est votre ami ! L'outil `valgrind` peut vous aussi aider (vous aurez un warning `Warning: client switching stacks?`[2] du à SystemC, vous pouvez l'ignorer).

— Modifier la mémoire de façon à inclure un test sur les adresses et à renvoyer une erreur en cas d'adresse invalide (en dehors de la plage réelle de la mémoire).

— Compiler et tester.

Question 6 : instanciation du contrôleur LCD

Nous passons maintenant à la phase 2, qui exploite les deux composants que vous venez de réaliser, ainsi que d'autres composants préexistants. Vous allez tout d'abord ajouter à la plate-forme un contrôleur (minimaliste) d'écran plat (Liquid Crystal Display Controller ou LCDC). La documentation du composant LDCD est donnée à part dans le fichier `LCDC-doc.pdf`.

— Les fichiers correspondant aux nouveaux composants sont également dans votre archive git, comme vous avez pu le constater (j'espère !).

— Vous utiliserez le `sc_main()`, la mémoire et le composant générateur que vous venez d'écrire comme base de travail. Il faut en revanche effacer les commandes de test de la mémoire écrites dans le générateur de transaction (la méthode associée au **`SC_THREAD`** doit être vide).

— Instancier le module LCDC et le connecter au reste de la plate-forme (il faudra ajouter un port d'interruption au générateur, nous l'utiliserons plus tard). Le constructeur du contrôleur prend un paramètre additionnel de type `sc_time` permettant de fixer la fréquence de rafraichissement de l'écran LCD. Passer comme paramètre : `sc_time(1.0 / 25, SC_SEC)` (rafraichissement 25 fois par seconde).

— Compiler et tester.

Question 2 : dimensionnement des plages d'adresses

L'affichage d'une image sur l'écran LCD par le contrôleur nécessite un espace mémoire dédié pour stocker cette image, appelé *mémoire vidéo* (ou VRAM pour *video ram*). Nous allons dédier une partie de la mémoire déjà présente dans la plate-forme à cet usage.

La mémoire est dimensionnée comme suit :

— 10 Kio sont réservés pour les données du logiciel embarqué (cette partie sera inutilisée dans notre plate-forme qui ne modélise pas le logiciel embarqué fidèlement) ;

— La mémoire vidéo utilise un codage d'1 octet par pixel, donc sa taille sera de $320 \times 240 = 76800$ octets.

La taille nécessaire pour la mémoire est donc de 87040 (soit 15400_{16}) mots. Modifiez la taille de la mémoire et la plage d'adresse correspondante (`bus.map()`) dans `sc_main()`.

Le LCDC n'expose qu'un (petit) banc de registres. Calculez la taille nécessaire d'après la documentation, et adaptez la plage d'adresse.

1. Rappelez vous l'excellent cours PCSEA de 2A.
2. Pourquoi ? PCSEA again !

Question 3 : modélisation du registre `START_REG` du LCDC

Dans la version récupérée du contrôleur LCD, il manque l'implantation du registre `START_REG` (voir documentation).

- Ajouter le registre dans le fichier `LCDC.cpp`. Que faut-il faire ? Utiliser les constantes définies dans le fichier `LCDC_registermap.h`.
- Compiler et tester.

Question 4 : premiers tests du LCDC

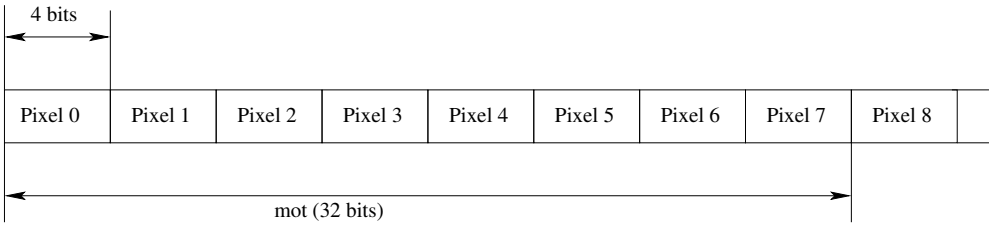
- Définir dans le générateur des constantes pour les différentes adresses que l'on manipulera par la suite : adresse de base de la mémoire, adresse de base de la partie mémoire vidéo, adresse de base du contrôleur LCD (utiliser des directives **`#define`**, ou **`const`** `ensitlm::addr_t ... = ...;`).
- Dans le générateur, réaliser des écritures dans la mémoire vidéo de façon à afficher une image blanche (c'est à dire, donc la mémoire vidéo est une suite de `0xFFFFFFFF`), avec une boucle simple du type

```
for (int i = 0; i < IMG_SIZE; i++) {
        ...;
}
```

- Compiler et tester.
- Réaliser les écritures appropriées dans les registres du LCDC pour observer effectivement l'image à l'écran (penser à utiliser les constantes définies dans le fichier `LCDC_registermap.h`).
- Compiler et tester.

Question 5 : Ajout d'un composant ROM

Nous allons maintenant afficher une image sur l'écran. L'image est stockée dans un composant ROM, dans un format légèrement différent de celui utilisé par le contrôleur LCD : l'image est stockée avec 4 bits par pixels, en niveau de gris. Le LCDC utilisant 8 bits par pixels, on considère que les 4 bits stockés dans la ROM correspondent aux bits de poids fort des 8 bits de la mémoire vidéo. Les pixels de 4 bits sont stockés les uns à la suite des autres, en format *big-endian* :



- Instancier le composant ROM, le connecter au bus dans `sc_main.cpp`.
- Ajouter une plage d'adresse, sachant que l'image fait 38400 octets, et est stockée au début de la ROM.
- Reprendre la boucle d'initialisation de la mémoire à `0xFFFFFFFF`, et remplacer le corps de la boucle par une fonction qui lit un groupe de pixels (un mot de 32 bits contenant 8 pixels de 4 bits chacun) depuis la ROM, et écrit les deux groupes de pixels correspondants dans la mémoire vidéo avec le format attendu par le LCDC. Les opérateurs de décalages de bits (<< et >>) et les masques de bits (`x & 0x00F00000`) devraient vous aider. Cette partie est réalisable en une dizaine de lignes de code.

Question 6 : traitement des interruptions

L'objectif de cette question est de traiter les interruptions du LCDC, de façon à pouvoir réaliser des animations à l'écran. Cela suppose de pouvoir écrire les données en mémoire vidéo au « bon moment », c'est à dire entre deux interruptions.

— Ajouter un port d'entrée d'interruption noté `irq` au générateur et le connecter au signal d'interruption issue du LCDC. Attention, le signal d'interruption va être piloté par deux composants (`LCDC`, et indirectement le générateur de trafic dans son traitement de l'interruption), donc il faudra instancier le signal correspondant comme ceci :

```
sc_signal<bool, SC_MANY_WRITERS> irq_signal("IRQ");
```

— À la suite du code déjà écrit dans le **`SC_THREAD`** du générateur, ajouter une attente sur l'interruption. Vous procéderez en déclarant une **`SC_METHOD`** qui notifiera un événement débloquant le **`SC_THREAD`** principal. Afficher après cette attente un message indiquant la prise en compte de l'interruption.

— Compiler et tester. Comment avez-vous fait ?

— Modifier le code du générateur de façon à avoir une boucle infinie d'attente d'interruptions. Dans cette boucle, ajouter après chaque attente les écritures permettant de générer une image différente à l'écran. On propose de décaler à chaque pas de la boucle l'image de un pixel vers le bas (en déplaçant la partie de l'image tronquée du bas sur le haut de l'image). On peut aussi afficher un fondu vers le noir ou vers le blanc, mais attention aux manipulations de bits un peu fines !