Ensimag 3A, filière SLE

2016-2017

Modélisation TLM en SystemC
TP n°3 : Intégration du logiciel embarqué

Consignes importantes pour tous les TPs

Rappel : le fichier `TP-commun.pdf` contient un ensemble de **consignes très importantes** pour les 3 TPs. Respectez ces consignes **scrupuleusement** pour ne pas perdre de points bêtement.

1 Objectif

Ce TP s'intéresse à l'intégration du logiciel embarqué. La plate-forme à modéliser a été développée sur FPGA (cf `TP3-zybo.pdf` pour un descriptif du système). C'est un petit système sur puce, avec le strict minimum pour faire tourner du logiciel non-trivial avec affichage graphique (mais pas d'accélérateur matériel ou de bloc IP exotique). Le logiciel embarqué proposé est le jeu de la vie (cf. `http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life` pour les règles du jeu et des exemples rigolos).

Nous allons expérimenter deux approches en simulation :

La simulation native : le logiciel embarqué sera compilé avec le même compilateur que la plate-forme, et liée comme un morceau de code en C quelconque. Les accès mémoires pertinents du point du vue des composants matériels seront routés sur le bus TLM. Le code embarqué est encapsulé dans un composant TLM appelé « wrapper natif ».

La simulation via ISS : L'ISS, ou Instruction Set Simulator, va interpréter directement le code compilé pour le processeur cible (un Microblaze dans notre cas). On utilisera donc la même chaîne de compilation que pour l'intégration du logiciel sur la puce finale (en théorie, le même binaire, au bit près, peut tourner sur ISS et sur la puce). Toutes les lectures/écritures faites par l'ISS seront routées sur le bus.

Il n'est pas garanti qu'un logiciel développé sur ISS continuera à tourner en simulation native. Par contre, un logiciel bien écrit et qui marche en simulation native devrait marcher sans modification sur la puce ou en simulation avec ISS (après recompilation bien entendu).

Dans les deux cas, la plateforme sera « loosely timed », c'est à dire que nous nous servons du temps pour faire une simulation raisonnable (par exemple, les timers sont timés correctement, l'ISS est modélisé avec une période qui correspond à la version FGPA), mais nous ne cherchons pas la précision. Par exemple, en simulation native, nous ignorons totalement la notion de temps pour l'exécution du logiciel, le modèle de bus que nous utilisons n'est pas timé, ...

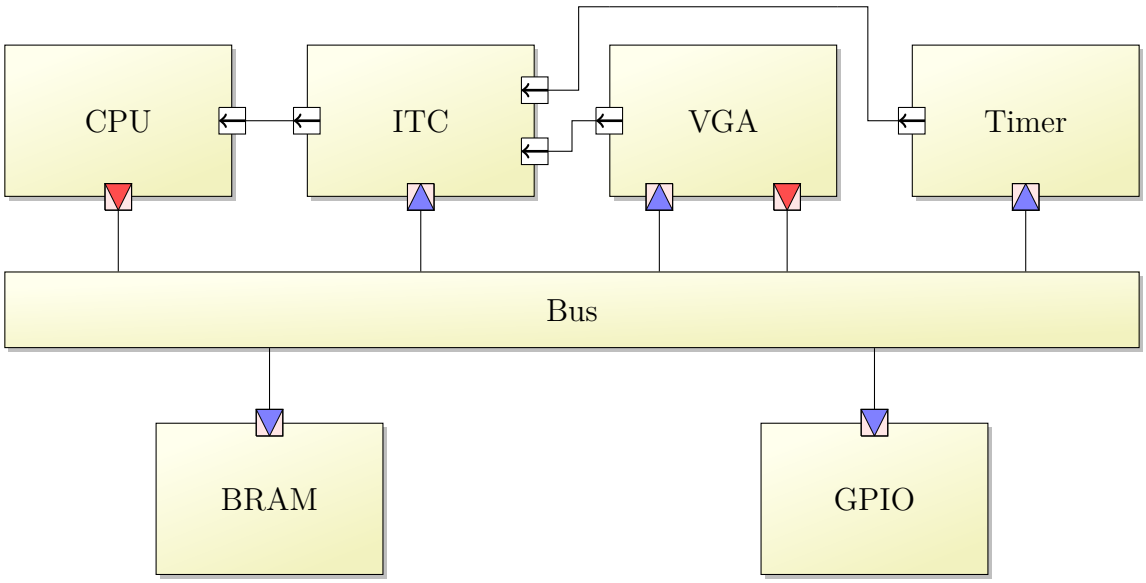

FIGURE 1 – Vue graphique de la plateforme TLM

2 La plateforme

La plateforme modélisée est celle présentée dans le document `TP3-zybo.pdf`. Les détails non-pertinents au niveau TLM sont abstraits. La plate-forme obtenue est décrite dans la figure 1.

3 Point de départ

Pour ce TP, on ne réutilisera pas les composants des TPs précédents. Récupérer le squelette de plate-forme dans `TPs/squelette/tp3` dans votre archive Git.

Le répertoire est organisé en plusieurs sous-répertoires :

zybo/ la « vraie » plateforme, prévue pour tourner sur carte FPGA Zybo, connectée à un écran VGA.

hardware/ les composants HW de la plateforme.

native-wrapper/ le toplevel (i.e. le fichier `sc_main_XXX.cpp` contenant la fonction `sc_main`) et les composants spécifiques au wrapper natif.

iss/ toplevel et composants spécifiques à l'ISS microblaze.

elf-loader/ le chargeur de fichier elf et ses dépendances. Utilisé par Memory.cpp pour charger le logiciel embarqué au format ELF lorsqu'on fait une simulation avec ISS.

software/ Le logiciel embarqué. Pour nous, il sera très simple : c'est une implémentation du jeu de la vie, qui tient en un seul fichier (`main.c`).

software/cross/ le nécessaire pour la compilation croisée (cross-compilation) du logiciel embarqué.

software/native/ le nécessaire pour la compilation en wrapper natif.

Pour commencer, on peut essayer :

```
cd iss/
make -k
cd ../native-wrapper
make -k
```

(l'option `-k` de `make` permet de ne pas arrêter la compilation à la première erreur rencontrée)

Le premier `make` va échouer sur la compilation du logiciel embarqué (votre première tâche sera de compléter le Makefile en question), mais construit tout de même l'exécutable `run.x` pour la plateforme. Le second va échouer pour la même raison (et l'édition de lien ayant besoin du logiciel embarqué, on ne peut même pas construire `run.x`). Une fois le TP terminé, les mêmes commandes créeront `iss/run.x` et `native-wrapper/run.x`, correspondant aux deux versions de la plate-forme.

La suite du TP sera donc de reconstituer les différents points qui manquent pour la compilation et l'exécution de ces deux plate-formes.

4 Compilation croisée du logiciel embarqué

```
cd software/cross/
ls
make
```

Pour l'instant, le `Makefile` fourni n'est pas complet : il manque les règles pour compiler, assembler et lier le logiciel embarqué (qui sera le fichier `a.out`). On compilera le logiciel avec un compilateur C, des macros sont fournies en tête du `Makefile` pour les noms des commandes. Pour l'édition de liens, on utilise un « linker script », qui donne à l'éditeur de liens les adresses finales des différentes sections et de certains symboles. Le script est dans le fichier `software/cross/ldscript`, et l'option `-T` de `ld` sera nécessaire. Une fois ces règles implémentées dans le `Makefile`, les règles `make dump.txt` et `make sections.txt` fournissent un résumé « lisible » du fichier compilé.

5 Simulation avec ISS

```
cd ../../
cd iss
make
./run.x
```

La plateforme devrait maintenant compiler et être capable de charger le logiciel embarqué. Parmi les choses qui ont été faites pour vous :

- Le composant `Memory` expose directement son tableau de stockage (champ `storage`).
- La fonction `sc_main()` charge directement le logiciel embarqué dans ce tableau, en utilisant le chargeur ELF.
- Un ISS microblaze fait partie de la plateforme, il est écrit en C++ (sans SystemC, ni TLM)
- Un wrapper pour cet ISS (un composant SystemC, avec interface ENSITLM in un port `sc_in` pour les IRQ, qui fait appel aux méthodes C++ de l'ISS) est partiellement implémenté.

Il manque cependant plusieurs choses, que nous allons implémenter dans les sections suivantes.

5.1 Gestion de la mémoire

- La couche d'abstraction (`software/cross/hal.h`) n'est pas implémentée. Tous les accès mémoires via cette API stopperont la simulation brutalement sur un `abort()` (dans un premier temps, vous pouvez ignorer la fonction `printf()`).

— L'ISS est totalement implémenté, mais le wrapper pour l'ISS ne l'est pas. Lorsque l'ISS demande une lecture ou une écriture, c'est au wrapper de faire l'accès effectif au bus. Il y a 3 types d'accès à implémenter : le fetch, les « load » et les « store ». Ils sont identifiés comme tels dans `mb_wrapper.cpp`. Attention, l'ISS gère en interne les données en big endian, et la plateforme d'exécution est une machine Intel en little-endian, donc le reste de la plateforme s'attend à recevoir des entiers en little-endian. Les macros `uint32_machine_to_be` et `uint32_be_to_machine` peuvent aider.
— La pile est positionnée, mais pas à une adresse raisonnable. Il faut positionner correctement la valeur de `_stack_top` dans `software/cross/ldscript`.

5.2 Affichage avec `printf`

Notre plate-forme a une capacité de debug limitée (on peut activer des traces via des macros au début des fichiers `mb_wrapper.cpp` et `microblaze.cpp`, mais elles sont en général soit trop soit pas assez verbeuses ...). Pour travailler plus confortablement, il est souhaitable de pouvoir utiliser la fonction `printf` pour afficher du texte sur la sortie standard du programme SystemC. Ce n'est pas aussi simple qu'on aurait pu le croire, vu que le code exécutable est interprété par l'ISS, on ne peut pas appeler directement la fonction `printf` de notre libc (hôte) depuis le code embarqué. La solution retenue est d'avoir un composant `UART`, qui va recevoir des caractères depuis le bus Ensitlm, et les afficher via `cout` en C++ (le composant physique aurait envoyé les caractères sur un lien série). Le composant UART vous est fourni, il vous reste :

— À instancier et connecter correctement le composant au bus dans `sc_main`.
— À écrire le corps de la fonction `printf` dans `hal.h`. On ne s'intéresse qu'au cas de `printf` à un seul argument, et sans caractères spéciaux (i.e. on affiche la chaîne passée en argument sans traitement particulier). Il suffit d'afficher les caractères un par un jusqu'au caractère `'\0'`.
— L'exécution de la fonction ci-dessus va probablement faire un accès en lecture sur un seul caractère sur le bus (`READ_BYTE` dans `mb_wrapper.cpp`). Vous allez devoir implémenter ce cas (ou éventuellement trouver une implémentation de `printf` qui ne l'utilise pas, mais ce n'est probablement pas le plus simple) pour que `printf` marche correctement. Par exemple, pour lire un caractère à l'adresse 5, il faut faire un accès à l'adresse 4 (i.e. le multiple de 4 immédiatement inférieur), qui va donner 4 octets, puis extraire l'octet numéro 1 (via un décalage et un masque de bit par exemple).

Vérifier que les instructions `printf` présente dans `main.c` sont bien prises en compte (l'exécution peut être lente, mais un message doit être affiché au début de la fonction `main`).

5.3 Gestion des interruptions

La gestion des interruptions est totalement absente du wrapper. Il faudra implémenter un processus SystemC sensible aux fronts sur le port `irq`, et qui utilise la fonction `m_iss.setIrq(true)` pour signaler à l'ISS qu'une interruption a été reçue. Une fois l'interruption traitée par l'ISS, il faut appeler `m_iss.setIrq(false)`. Pour que l'ISS ait vu l'interruption, il faut que la fonction `m_iss.step()` ait été appelée plusieurs (par exemple, 5) fois. Il faudra donc ajouter un compteur qui remet l'interruption à faux après 5 cycles.

6 Compilation pour simulation native

```
cd ../
cd software/native/
cat hal.h
```

```
make
```

Peu de choses à faire dans ce répertoire : il manque simplement la règle pour compiler le logiciel embarqué en mode natif. On compilera avec un compilateur C (`gcc`) et non C++ (`g++`).

Le fichier `hal.h` est complet, mais il se contente de rediriger les appels sur des fonctions qui seront implémentées dans le wrapper natif (`native-wrapper/native_wrapper.cpp`).

7 Execution en simulation native

```
cd ../../
cd native-wrapper/
make
```

Pour la simulation native, il reste à implémenter le wrapper natif. Il se trouve dans le fichier `native_wrapper.cpp`. Le squelette est fourni, mais le corps de la plupart des fonctions n'est pas implémenté.

— Les fonctions `hal_read32`, `hal_write32`, `hal_cpu_relax` et `hal_wait_for_irq` doivent rediriger sur les méthodes correspondantes de `NativeWrapper`.

— Les méthodes de `NativeWrapper` doivent être implémentées.

Par ailleurs, en exécution native, on peut avoir des problèmes liés au fait que par défaut, SystemC ne laisse pas le temps s'écouler. Une boucle d'attente active (polling) dans le logiciel embarqué va donc figer la simulation (et il y en a une dans `main.c` !). L'astuce classique consiste à « casser » les boucles d'attente avec un appel à `hal_cpu_relax()`, qui laisse le temps s'écouler [1].

8 Comparaison de performances

Comparer les performances obtenues avec l'ISS et avec le wrapper natif.

9 Origine des composants

Certains composants viennent du projet SocLib (`https://www.soclib.fr/`) :

— Le chargeur ELF (`elf-loader/`),

— L'ISS MicroBlaze (`microblaze.*`, `arithmetics.h`, `iss.h`, `register.h`),

— Le fichier `soclib_endian.h`.

La chaîne de compilation microblaze (gcc, ld, ...) est celle de Xilinx (`http://xilinx.wikidot.com/mb-gnu-tools`).

1. sur la plupart des plate-formes, cette fonction `hal_cpu_relax()` aurait des choses intéressantes à faire en dehors du contexte de la simulation native, comme diminuer la priorité du processus courant, vider des caches, ... mais ce n'est pas le cas sur notre microblaze sans OS, sans cache, ...