Ensimag 3A, filière SEOC

2020-2021

Modélisation TLM en SystemC
TP n°2 : Intégration du logiciel embarqué

Consignes importantes pour tous les TPs

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

1 Objectifs

Ce TP s'intéresse à l'intégration du logiciel embarqué au sein d'un SoC. 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 un bête affichage d'une image noir et blanc parcourue par une balle bondissante changeant pseudo-aléatoirement de direction et vitesse.

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 : un ISS (*Instruction Set Simulator*), va interpréter directement le code compilé pour le processeur cible (un risc-v 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 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, ...

2 Travail attendu et organisation

Pour mémoire, ce TP est, à la différence du précédent, noté, et doit être fait en *binôme*, et comme vous êtes 26, tout baigne. Il compte pour $\frac{2}{3}$ des 50% de la note finale, les 50% autres venant d'un

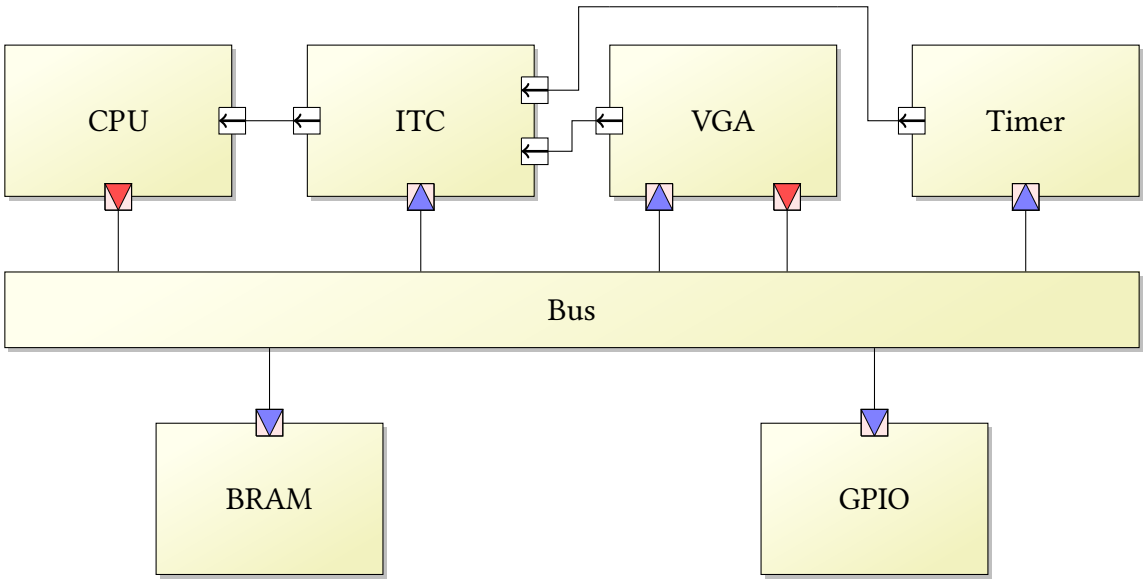

FIGURE 1 – Vue graphique de la plateforme TLM

examen papier, calqué sur les précédents. Le rendu attendu est un fichier texte indiquant les membres du binôme et l'url d'un git accessible par moi contenant votre rendu. Attention, le dernier commit pris en compte sera celui de la deadline attendue, le lundi 18 janvier 2021 à 8h00, heure de Grenoble.

Le déroulement des séances que nous vous suggérons est le suivant :

— 1er TP : simulation native fonctionnelle ;
— 2nd TP : simulation à l'aide de l'ISS sans les interruptions ;
— 3ème TP : simulation à l'aide de l'ISS avec les interruptions.

Si vous dérivez fortement, nous vous conseillons de travailler entre les séances.

3 La plateforme

Les détails non-pertinents au niveau TLM sont abstraits, en conséquence la plate-forme TLM équivalente est décrite dans la figure 1.

4 Point de départ

Pour ce TP, on ne réutilisera pas les composants du TP précédent. Récupérer le squelette de plate-forme dans `TPs/squelette/tp2` dans votre archive Git.

Le répertoire est organisé en plusieurs sous-répertoires :

hardware/ les modèles des composants matériels 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 risc-v.

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 implantation d'une balle rebondissante 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 native-wrapper
make -k
cd ../iss/
make -k
```

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

Le premier `make` va échouer car vous devez définir les règles de compilation pour la version native du logiciel embarqué. Le fichier à modifier est `squelette/tp2/software/native/Makefile`. Le second `make` va échouer sur la compilation du logiciel embarqué, mais construit tout de même l'exécutable `run.x` pour la plateforme. Le fichier à modifier se trouve dans `squelette/tp2/software/cross/Makefile`. Une fois le TP terminé, les mêmes commandes créeront `native-wrapper/run.x` et `iss/run.x`, correspondant aux deux versions de la plate-forme.

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

5 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 implantées dans le wrapper natif (`native-wrapper/native_wrapper.cpp`).

6 Execution en simulation native

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

Pour la simulation native, il reste à implanter le wrapper natif. Il se trouve dans le fichier `native_wrapper.cpp`. Le squelette est fourni, mais le corps de la plupart des fonctions est vide, à vous d'y mettre ce qu'il faut !

- 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 écrites.

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].

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 risc-v sans OS, sans cache, ...

7 Compilation croisée du logiciel embarqué

Le cross-compilateur est disponibles sur les machines de l'Ensimag (si vous avez sourcé le fichier `setup-ensimag.sh`. Si vous voulez l'installer sur votre propre machine, il faut recompiler `gcc`. Les promoteurs du risc-v maintiennent une branche configurée proprement dans un dépôt qui peut être récupéré avec la commande `git clone https://github.com/riscv/riscv-gnu-toolchain`. Vous y trouverez également la procédure détaillée d'installation.

```
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 ajoutées dans le `Makefile`, les règles `make dump.dis` et `make sections.txt` fournissent un résumé « lisible » du fichier compilé.

8 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 risc-v 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 disponible.

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

8.1 Gestion de la mémoire

- la couche d'abstraction (`software/cross/hal.h`) n'est pas écrite. 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 interprètre toutes les instructions, mais le wrapper permettant d'en faire usage doit être écrit ; 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 implanter : la lecture des instruction (« fetch »), les « load » et les « store ». Ils sont identifiés comme tels dans `rv32_wrapper.cpp`. L'ISS gère en interne les données en little endian, et la plateforme d'exécution est, classiquement, une machine Intel en little-endian, donc le reste de la plateforme s'attend à recevoir des entiers en little-endian. En revanche, si la machine sur laquelle s'effectue la simulation est big-endian

(genre power d'IBM), la simulation se passera mal. On va donc systématiquement utiliser des macros, au cas où, pour adapter la machine simulée (dont on sait qu'elle est little) à la machine de simulation (dont on ne connaît *a priori* pas l'endianness) : `uint32_machine_to_le` et `uint32_le_to_machine`;
— la pile est positionnée, mais pas à une adresse raisonnable. Il faut positionner correctement la valeur de `_stack_top` dans `software/cross/ldscript`.

8.2 Affichage avec `printf`

Notre plate-forme a une capacité de debug limitée (on peut activer des traces via des macros au début des fichiers `rv32_wrapper.cpp` et `rv32.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 par l'intermédiaire de `READ_BYTE` dans `rv32_wrapper.cpp`. Pour mémoire, le bus `ensitlm` ne supporte que des transferts de mots, il faut donc trouver la « bonne » adresse mot correspondant à l'adresse octet, puis extraire le bon octet du mot.
Par exemple, pour lire un caractère à l'adresse `0x016af`, il faut faire un accès à l'adresse `0x016ac` (i.e. le multiple de 4 immédiatement inférieur), qui va donner 4 octets, puis extraire l'octet correspondant via un décalage de 3 octets et un masque de bit :
`((*(uint32_t *)0x016ac) >> (3 * 8)) & 0xF`.

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`).

8.3 Gestion des interruptions

La gestion des interruptions est totalement absente du wrapper. Il faudra ajouter 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.