Modélisation Transactionnelle des Systèmes sur Puces

Ensimag 3A, filière SLE

Janvier 2011

Consignes :

- Durée : 2h.
- Tous documents autorisés.
- Le barème est donné à titre indicatif.
- On attend des réponses courtes et pertinentes, inutile de recopier le cours.
- Les schémas brouillons seront pénalisés.

1 Question de cours

Question 1 (2 points) *Citez 3 alternatives à l'approche TLM pour le développement du logiciel embarqué. Pour chacune d'elle, décrivez-là brièvement, et précisez les avantages et/ou inconvénients par rapport à TLM.*

1. RTL : bas niveau, synthétisable, mais lent.
2. FPGA/Emulateurs : hardware programmable qui reproduit la fonctionalité du RTL, cher, debug limité, et nécessite le code RTL
3. Vraie puce : disponible tard dans le cycle, debug limité.

2 Ajout d'une fonctionnalité au programme du TP3

Dans cet exercice, on part du code du TP3 (jeu de la vie sur plate-forme à base de microblaze), et on ajoute une fonctionnalité (qui n'existe pas sur la version RTL de la plate-forme, mais qui pourrait être ajoutée en modifiant le code VHDL). Les modifications sont décrites avec la syntaxe « diff » : les lignes préfixées d'un '-' sont les lignes supprimées, celles préfixés d'un '+' sont ajoutées, et celles commençant par un espace représentent le contexte (l'endroit où la modification est faite).

Le code a été légèrement offusqué : un mot (toujours le même) a été remplacé par `XXXXX`. Dans la suite, on dira qu'on ajoute la fonctionnalité XXXXX.

1. On ajoute un champ publique `bool m_XXXXXing` dans la classe `MicroBlazeIss` (fichier `microblaze.h`). Le constructeur est ensuite modifié de la manière suivante :

```
--- iss/microblaze.cpp
+++ iss/microblaze.cpp
@@ -198,6 +203,7 @@ namespace soclib { namespace common {

        MicroBlazeIss::MicroBlazeIss(uint32_t ident)
-               : Iss(mkname(ident), ident) {
+               : Iss(mkname(ident), ident),
+                 m_XXXXXing(false) {
                }
```

2. On définie `OP_XXXXX` comme suit :

```
--- iss/microblaze.cpp
+++ iss/microblaze.cpp
@@ -63,6 +63,11 @@
 #define OP_BS         0x11
 #define OP_IDIV       0x12
 #define OP_FSL        0x13
+#define OP_XXXXX      0x14
 #define OP_MULI       0x18
 #define OP_BSI        0x19
```

3. L'ajout suivant se trouve dans la fonction `MicroBlazeIss::step`, plus précisément dans un des cas de l'instruction `switch` suivante :

```
void MicroBlazeIss::step(void)
{
        // ...
        switch (ins_opcode) {
                // ...
                // le code est ajouté ici
                // ...
        }
}
```

Le code ajouté est le suivant :

```
--- iss/microblaze.cpp
+++ iss/microblaze.cpp
@@ -732,6 +738,12 @@ namespace soclib { namespace common {
                                r_gpr[ins_rd] = m_ident;
                                next_pc = r_npc + 4;
                                break;
+                       case OP_XXXXX:
+                               m_XXXXXing = true;
+                               break;
                        case OP_IMM:
 #if MBDEBUG
```

4. Dans la fonction `MBWrapper::run_iss(void)` du wrapper SystemC de l'ISS, on fait l'ajout suivant :

```
--- iss/mb_wrapper.cpp
+++ iss/mb_wrapper.cpp
@@ -54,6 +54,10 @@ void MBWrapper::run_iss(void) {
        int inst_count = 0;

        while(true) {
+               if (m_iss.m_XXXXXing) {
+                       wait(irq.posedge_event());
+                       m_iss.m_XXXXXing = false;
+               }
                // cout << "Starting new processor cycle" << endl;

                if (m_iss.isBusy())
```

5. Enfin, la couche d'abstraction du hardware est modifiée comme suit :

```
--- software/cross/hal.h
+++ software/cross/hal.h
@@ -21,7 +21,7 @@
-#define wait_for_irq()  do { irq_received = 0; \
-                        while(irq_received == 0) {}} while (0)
+#define wait_for_irq()  __asm(".byte 0x50, 0x0, 0x0, 0x0")
```

   La directive `__asm()` permet d'utiliser de l'assembleur inline dans du code C, et la directive `.byte` permet d'ajouter les octets donnés en argument dans le code exécutable généré.
   La valeur 0x50 et la valeur 0x14 rencontrée plus haut sont égales, à un décalage de bits près.

(Dans l'implémentation réelle de cette fonctionnalité, le tableau `OpcodeTable[]` est aussi modifié pour enregistrer XXXXX comme étant de type A, mais ce n'est pas nécessaire à la compréhension de l'exercice)

2.1 Questions de cours

Question 2 (1 point) *Quelle est la construction C++ utilisée pour la modification 1 au niveau du constructeur ? Dans quels cas, en C++, est-ce différent d'une affectation ?*

Chaînage de constructeur. Ça permet d'initialiser une constante, et de passer des arguments au constructeur des champs.

Question 3 (1 point)
L'ISS utilisé dans ce TP utilise un algorithme d'interprétation du code microblaze (on reconnaît l'instruction `switch` *typique de ce genre de programmes). Citez une autre technique possible pour réaliser un ISS, en donnant au moins un exemple d'outil implémentant cette technique, et en précisant l'avantage ou l'inconvénient de cette autre technique.*

Traduction dynamique de code binaire, qui permet une exécution beaucoup plus rapide. Par exemple, QEmu, OVP, SimSoC, ...

Question 4 (1 point)
Citez une technique alternative à l'ISS pour réaliser la même tâche. Donnez un avantage et un inconvénient de cette autre technique.

La simulation native (via wrapper natif), qui permet une simulation beaucoup plus rapide, mais qui est moins fidèle (besoin de modifier une partie du code embarqué, timing difficile à évaluer, ...)

2.2 Compréhension du code

Question 5 (3 points) *Quelle est la fonctionnalité XXXXX ajoutée par ces modifications ? Expliquez le fonctionnement de l'implémentation pour chaque point modifié (de 1 à 5).*

Ajout d'une instruction SLEEP qui fait une attente explicite d'interruption.

1. On ajoute un champ pour savoir depuis l'extérieur si l'instruction a été appelée.
2. On déclare une constante correspondant au codop de l'instruction.
3. On ajoute un cas au switch pour l'instruction en question, qui positionne `m_XXXXXing` à vrai quand on exécute l'instruction.
4. Dans le wrapper, si l'ISS est en attente d'interruption, on attends effectivement l'interruption (via le `wait` de SystemC) et on redescend le flag `m_XXXXXing`).
5. Comme le microblaze n'a pas réellement d'instruction SLEEP, on code cette instruction directement via un .byte en assembleur. L'instruction correspond directement à la fonction `wait_for_irq()` de notre couche d'abstraction du matériel.

Question 6 (1 point) *Quelle modification serait nécessaire dans les répertoires* `native_wrapper` *(qui contient la fonction* `sc_main` *et le wrapper natif) et* `software/native/` *(qui contient le nécessaire pour compiler le logiciel sur la plate-forme avec wrapper natif, en particulier* `hal.h`*) pour utiliser la fonctionnalité XXXXX ?*

Question 7 (1 point) *En terme de vitesse de simulation, la nouvelle version de* `wait_for_irq` *sera-t-elle plus rapide ou plus lente que l'ancienne ? Pourquoi ?*

3 Écriture d'une plate-forme TLM

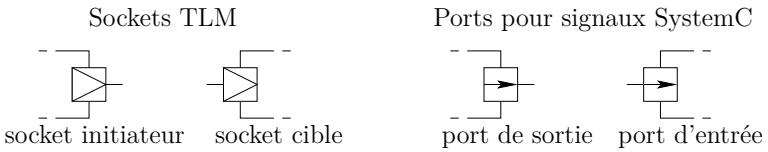
On considère une application de traitement d'image qui utilise notamment deux microprocesseurs ayant accès à une mémoire commune. Les deux processeurs écrivent sur des

zones disjointes de la mémoire. À la fin de son traitement, le processeur 1 met à disposition du processeur 2 l'image qu'il vient de finir de calculer en l'écrivant en mémoire. Le processeur 2 est originellement en attente d'une image disponible en provenance du processeur 1. Lorsqu'une image est disponible, il la recopie dans sa zone mémoire et réalise un traitement sur celle-ci puis se remet en attente d'une autre image (on ne se préoccupe pas de savoir comment le résultat du traitement par le processeur 2 est communiqué vers l'extérieur).

Pour synchroniser les deux processeurs, on souhaite utiliser un composant Mailbox, dont la spécification est fournie en annexe.

Note : chaque processeur dispose d'une unique entrée interruption nommée `irq`, active sur front montant. Les autres entrées/sorties (processeur et mémoire) sont les mêmes que pour les composants vus en TP.

Question 8 (3 points) *En utilisant les conventions vues en cours et rappelées partiellement ci-dessous, donnez une vue graphique de la plate-forme.*



Expliquer comment intégrer le composant Mailbox et comment l'utiliser, de façon concise mais précise.

Question 9 (5 points) *Écrire en SystemC (fichier .h et fichier .cpp) la classe Mailbox implémentant la spécification donnée en annexe.*

Vous utiliserez les mêmes éléments de la bibliothèque ENSITLM que ceux vus en cours et en TPs. On supposera l'existence de deux types `ensitlm::addr_t` *et* `ensitlm::data_t`*, chacun codés sur 32 bits. Enfin, vous considérerez que les sorties d'interruptions sont à* `false` *en début de simulation (pas d'initialisation nécessaire).*

cf. `../2005-2006/mailbox/`

On souhaite maintenant écrire en langage C le logiciel embarqué sur le processeur **1**. On suppose l'existence des fonctions suivantes :

- `void production_image()` : calcule une nouvelle image et la met à disposition dans la mémoire commune (a utiliser sur le processeur 1),
- `void recopie_image()` : depuis le processeur 2, lit une image dans la mémoire partagée, et la recopie dans une zone mémoire utilisée seulement par le processeur 2,
- `void consommation_image()` : fait un second traitement sur l'image contenue dans la zone mémoire utilisée par le processeur 2.
- `void write_mem(a, d)` : écrit la donnée `d` à l'adresse `a` sur le bus auquel est relié le processeur,
- `ensitlm::data_t read_mem(a)` : lit une donnée à l'adresse `a` sur le bus auquel est relié le processeur, et la retourne,

– `void wait_for_irq()` : attend que le port d'interruption passe à 1. Si le signal d'interruption est déjà à 1, cette fonction retourne immédiatement.

Le port esclave (cible) du composant Mailbox est à l'adresse `0x1000` sur le bus principal.

Question 10 (2 points) *Écrire le logiciel embarqué du processeur 1 (une fonction `main_proc1`), et du processeur 2 (une fonction `main_proc2`).*

```
int main_proc1() {
    while(true) {
        production_image();
        write_mem(0x1000);
        wait_for_irq();
    }
}

int main_proc2 () {
    while (true) {
        wait_for_irq();
        recopie_image();
        read_mem(0x1000);
        consommation_image();
    }
}


20
   20
```

Annexe

Mailbox Documentation

Le composant Mailbox est un module esclave *Advanced Microcontroller Bus Architecture* (AMBA) destiné à être connecté à un bus haute-performance *Advanced High-performance Bus* (AHB).

Fonctionnalités

Le composant Mailbox fournit la logique et les entrées/sorties nécessaires à la synchronisation de deux modules maîtres pour un échange unidirectionnel maître 1 vers maître 2 d'une donnée sur 32 bits.

Entrées/Sorties

Le composant Mailbox possède les entrées/sorties suivantes :

- Interface esclave compatible AMBA
- Sortie d'interruption `int_sender` pour le composant maître 1 (émetteur)
- Sortie d'interruption `int_receiver` pour le composant maître 2 (récepteur)

Fonctionnement interne

Initialement les deux sorties d'interruption du composant Mailbox sont à 0 (ou `false`). Lors d'une écriture de valeur sur le registre `DATA_REG`, une interruption sur `int_receiver` est émise (passage à 1 ou `true`), la sortie d'interruption `int_sender` est réinitialisée (à 0 ou `false`) et la valeur est stockée dans le registre. Lors d'une lecture de valeur sur le registre `DATA_REG`, une interruption sur `int_sender` est émise (passage à 1), la sortie d'interruption `int_receiver` est réinitialisée (à 0) et la valeur du registre renvoyée.

Récapitulatif des registres

Adresse relative	Type	Taille	Valeur initiale	Nom	Description
0x00	Lecture/Écriture	32 bits	0x00000000	DATA_REG	Registre de données

Utilisation du composant

Le registre de données contient la valeur à échanger entre les deux composants maîtres. L'écriture d'une valeur dans ce registre déclenche une interruption sur `int_receiver` et remet à zéro l'interruption sur `int_sender`. De façon symétrique, une lecture du registre provoque une interruption sur `int_sender` et une remise à zéro de l'interruption de `int_receiver`.