Modélisation Transactionnelle des Systèmes sur Puces

Ensimag 3A, filière SLE

Février 2014

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 trop brouillons seront pénalisés.

1 Question de cours sur C++

1.1 Héritage en C++

On considère le programme suivant :

```
#include <iostream>
using namespace std;

struct Base {
        Base() {
                cout << "Base()" << endl;
        }

        Base(int i) {
                cout << "Base() avec i = " << i << endl;
        }
};

struct Derived : Base {
        Derived() : Base(42) {
                cout << "Derived()" << endl;
        }

        Derived(int i) : Base() {
                cout << "Derived() avec i = " << i << endl;
```

```
        }
};

struct Composed {
        Base b1, b2;
};

int main() {
        Derived d1(12), d2;
        Composed c;
}
```

Question 1 (1.5 points) *Qu'affiche ce programme ? Expliquez brièvement chaque ligne de la sortie.*

1.2 Templates en C++

On considère la classe template suivante, dans laquelle certains types ont été remplacés par `TYPE1`, ... `TYPE4` :

```
template<typename T>
struct tableau_de_tableaux {
        vector<vector<T> > elements;

        void ajouter_ligne(TYPE1 ligne) {
                elements.push_back(ligne);
        }

        TYPE2 access_element(TYPE3 x, TYPE4 y) {
                return elements[x][y];
        }
};
```

Question 2 (1.5 points) *Par quoi faut-il remplacer `TYPE1`, `TYPE2`, `TYPE3` et `TYPE4` pour que la déclaration soit proprement typée ?*

2 Questions de cours sur TLM et les SoCs

Question 3 (1 point) *Les besoins en calculs des systèmes sur puces ne cessent d'augmenter. Donnez 3 exemples concrets appuyant cette affirmation.*

Question 4 (1 point) *Citez deux méthodes utilisant SystemC et TLM pour faire de l'évaluation de performance de systèmes sur puces. Quelle est la méthode la plus utilisée aujourd'hui à STMicroelectronics ?*

Question 5 (1 point) *Qu'est-ce qu'IP-XACT ? Quelles sont les utilisations possibles d'IP-XACT liées à SystemC/TLM (citez-en deux) ?*

3 Modélisation du temps et ordonnancement (scheduling) en SystemC

On considère le programme suivant :

```
#include <systemc>
#include <iostream>

using namespace std;
using namespace sc_core;

SC_MODULE(A) {
        void f() {
                cout << "f0" << endl;
                wait(3, SC_SEC);
                cout << "f1" << endl;
                sleep(1);
                cout << "f2" << endl;
        }
        SC_CTOR(A) {
                SC_THREAD(f);
        }
};

SC_MODULE(B) {
        void g() {
                cout << "g0" << endl;
                sleep(2);
                cout << "g1" << endl;
                wait(4, SC_SEC);
                cout << "g2" << endl;
        }
        SC_CTOR(B) {
                SC_THREAD(g);
        }
};

int sc_main(int, char**)
{
        A a1("a1");
        A a2("a2");
        B b("b");

        sc_start();

        return 0;
```

```
}
```

On rappelle que la fonction `sleep(N)` est une fonction POSIX (pas une fonction SystemC) qui provoque une attente de `N` secondes du processus courant.

La norme SystemC autorise plusieurs exécutions possibles de ce programme (i.e. le scheduler a la liberté de choisir entre ces exécutions). Certaines exécutions produisent les mêmes affichages. Il y a trois traces d'exécutions différentes (i.e. avec affichage différent) possibles.

Question 6 (1.5 points) *Expliquez le choix du scheduler SystemC, et donnez les trois traces d'exécution.*

Question 7 (2.5 points) *Représentez deux des exécutions sur un graphique à deux dimensions. On mettra le "wall-clock time" sur l'axe des ordonnées et le temps simulé sur les abscisses. Si on néglige le temps de calcul, combien de temps prendra une exécution en "wall-clock time" ? Combien de temps en temps simulé ?*

4 Modélisation d'une lecture de caractères via l'UART

Dans votre TP3, le composant UART vous permettait d'implémenter un affichage (via une version simplifiée de fonction printf). Comme son nom l'indique, le composant UART (Universal asynchronous receiver/transmitter) permet aussi de recevoir des données.

Le but de cette partie est d'implémenter la fonction suivante dans le logiciel :

```
uint32_t uart_getchar (void);
```

Cette fonction lit un caractère depuis la FIFO d'entrée du composant UART (le caractère est consommé), et le renvoie sous forme d'un entier 32 bits. Si la FIFO de l'UART est vide, alors la fonction attend qu'un caractère soit reçu par l'UART.

Pour la vraie plateforme, l'UART serait connectée à un câble série RS232. Dans notre modèle, l'entrée de l'UART sera simplement l'entrée clavier du processus (stdin, ou cin en C++).

Une version simplifiée de la documentation du composant UART Xilinx est disponible en annexe.

Le fichier `uart.h` contient la déclaration suivante :

```
class UART : public sc_core::sc_module {
public:
        ensitlm::target_socket<UART> target;

        tlm::tlm_response_status
                read(ensitlm::addr_t a, ensitlm::data_t& d);

        tlm::tlm_response_status
                write(ensitlm::addr_t a, ensitlm::data_t d);
```

```
        SC_HAS_PROCESS(UART);
        UART(sc_core::sc_module_name name);
private:
        sc_core::sc_event m_fifo_not_empty_event;
};
```

Un squelette est donné pour les méthodes `read` et `write` :

```
tlm::tlm_response_status
UART::write(ensitlm::addr_t a, ensitlm::data_t d)
{
        char c = (char)d;
        switch (a) {
        case UART_FIFO_READ:
                // ...
        case UART_FIFO_WRITE:
                if (c == '\n') {
                        cout << endl;
                } else {
                        cout << c;
                }
                break;
        default:
                SC_REPORT_ERROR(sc_core::sc_module::name(),
                                "...");
                return tlm::TLM_ADDRESS_ERROR_RESPONSE;
        }
        return tlm::TLM_OK_RESPONSE;
}

tlm::tlm_response_status
UART::read(ensitlm::addr_t a, ensitlm::data_t& d)
{
        switch (a) {
        case UART_FIFO_READ:
                // ...
        case UART_STATUS:
                // ...
        default:
                SC_REPORT_ERROR(sc_core::sc_module::name(),
                                "...");
                return tlm::TLM_ADDRESS_ERROR_RESPONSE;
        }
        return tlm::TLM_OK_RESPONSE;
}
```

On suppose qu'on dispose d'une fonction `char_on_stdin()` qui renvoie `true` si un caractère est disponible sur l'entrée standard, et `false` sinon.

4.1 Version simplifiée sans interruptions

Dans un premier temps, on ne s'intéressera pas aux interruptions : le logiciel peut utiliser le registre `UART_STATUS` pour savoir si des caractères sont disponibles dans la FIFO. Le port d'interruption n'est même pas modélisé.

Question 8 (1 point) *Proposez une implémentation pour* `write` *dans le cas* `UART_FIFO_READ`*.*

Question 9 (1 point) *Proposez une implémentation pour* `read` *dans le cas* `UART_FIFO_READ`*.*

Question 10 (1 point) *Proposez une implémentation pour* `read` *dans le cas* `UART_STATUS`*.*

Question 11 (1 point) *En comparaison avec votre TP3, a-t-on besoin de modifier la fonction* `sc_main` *? Si oui, quelles sont les modifications à faire ?*

On s'intéresse maintenant à l'implémentation de fonctions logicielles permettant d'accéder à l'UART.

Question 12 (1 point) *A-t-on besoin de modifier la couche d'abstraction du matériel (*`hal.h`*) ? Si oui, quelles sont les modifications à faire ?*

Question 13 (0.5 point) *Proposez une implémentation de la fonction (logicielle)* `int uart_char_available(void)`*, qui renvoie une valeur non-nulle si et seulement si un caractère est disponible sur l'entrée de l'UART.*

Question 14 (1 point) *Proposez une implémentation de la fonction* `uart_getchar()` *(toujours dans le logiciel).*

4.2 Version avec gestion propre des interruptions

Un problème avec la solution ci-dessus est que si la fonction `uart_getchar()` est appelée alors que la FIFO est vide, alors cette dernière est obligée de faire de l'attente active.

Question 15 (1 point) *Citez deux inconvénients de l'attente active par rapport à l'attente passive.*

En réalité, l'UART que nous utilisons permet d'utiliser les interruptions pour mieux gérer ce cas. Nous allons maintenant raffiner notre modèle et corriger le logiciel pour refléter la réalité.

Le contrôleur d'interruption est modifié pour gérer 3 ports d'entrée (i.e. un nouveau port d'entrée `in2` est ajouté en plus de `in0` et `in1`).

Question 16 (0.5 point) *Que faut-il ajouter à la déclaration de la classe* `UART` *(*`uart.h`*) pour refléter l'interface avec interruption de l'UART ?*

Question 17 (1 point) *A-t-on besoin de modifier la fonction* `sc_main` *? Si oui, quelles sont les modifications à faire ?*

On suppose qu'un mécanisme a déjà été implémenté pour que l'évènement SystemC `m_fifo_not_empty_event` soit notifié quand l'utilisateur entre des caractères sur l'entrée standard [1].

Question 18 (1.5 points) *Que faut-il ajouter dans le module SystemC UART pour implémenter le paragraphe « Interruptions » de la spécification du composant ?*

Question 19 (1 point) *Proposez une nouvelle implémentation de* `uart_getchar()` *utilisant la fonction appropriée de* `hal.h` *pour attendre une interruption, si nécessaire. Attention, des interruptions peuvent venir de composants autres que l'UART, et ce cas doit être traité correctement.*

1. Dans mon corrigé, ce mécanisme lance un thread qui en boucle appelle `cin.peek()` pour attendre le prochain caractère, puis `async_request_update()` pour exécuter la notification d'évènement côté SystemC. Mais vous n'avez pas besoin de tout ça pour répondre !

Annexe

UART Documentation

Le composant UART est un module cible destiné à être connecté à un bus Xilinx On-chip Peripheral Bus (OPB). Il comporte un port de sortie d'interruption.

Fonctionnalités

Le composant UART permet de recevoir et d'envoyer des données caractère par caractère sur un lien série (RS232).

Les données reçues depuis le lien série sont accumulées dans une FIFO d'entrée et accessibles via le registre `UART_FIFO_READ`.

Les données envoyées par le système sur le registre `UART_FIFO_WRITE` sont entrées dans une FIFO de sortie et sont envoyées dès que possible sur le lien série.

Récapitulatif des registres

Adresse relative	Type	Nom	Description
`0x00`	Lecture seule	`UART_FIFO_READ`	Lecture depuis la FIFO d'entrée
`0x04`	Écriture seule	`UART_FIFO_WRITE`	Écriture sur la FIFO de sortie
`0x08`	Lecture seule	`UART_STATUS`	Informations sur l'état de l'UART

Registre `UART_FIFO_READ`

Si un caractère est présent dans la FIFO d'entrée, alors ce caractère est lu, supprimé de la FIFO, et renvoyé à l'initiateur de la transaction (en ajoutant 24 bits « 0 » en tête pour étendre le caractère vers un entier non-signé sur 32 bits). Sinon, un caractère quelconque est renvoyé.

Registre `UART_FIFO_WRITE`

La donnée reçue par l'UART est tronquée à 8 bits, et envoyée sur la FIFO de sortie pour envoi sur le lien série.

Registre `UART_STATUS`

Seul le bit `RX_FIFO_VALID_DATA` (numéro 31) est utilisé dans cette version de l'UART. Quand un caractère est présent dans la FIFO d'entrée, alors la valeur renvoyée à l'initiateur positionne `RX_FIFO_VALID_DATA` à 1. Si la FIFO d'entrée est vide, alors `RX_FIFO_VALID_DATA` est positionné à 0.

La valeur des autres bits est non-spécifiée et peut varier d'une implémentation à l'autre.

Gestion des interruptions

Quand un caractère est reçu sur la FIFO d'entrée, le signal d'interruption est levé à 1. Le signal d'interruption est réinitialisé à 0 quand une lecture a lieu sur le registre `UART_FIFO_READ`.