Modélisation Transactionnelle des Systèmes sur Puces

Ensimag 3A, filière SLE

Matthieu Moy — 8 Février 2017

Consignes :

- Durée : 2h,
- Document autorisé : une feuille A4 manuscrite recto-verso seulement,
- Le barème est donné à titre indicatif.

1 C++

1.1 Héritage

On considère le programme C++ ci-dessous :

```
#include <iostream>

using namespace std;

struct A {
        int a;
        int b;
        A() : a(42), b(12) {}
        A(int x) : a(x), b(123) {}
};

struct B : A {
        int b;
        B() : A(666), b(456) {}
};

struct C : A {
        int c;
        C() : c(789) {}
};

struct D : B, C {

};
```

```
int main() {
        D d;
        cout << "d.a␣=␣" << d.a << endl;        // 1
        cout << "d.B::a␣=␣" << d.B::a << endl; // 2
        cout << "d.C::a␣=␣" << d.C::a << endl; // 3

        cout << "d.b␣=␣" << d.b << endl;        // 4
        cout << "d.B::b␣=␣" << d.B::b << endl; // 5
        cout << "d.C::b␣=␣" << d.C::b << endl; // 6

        cout << "d.c␣=␣" << d.c << endl;        // 7
        cout << "d.B::c␣=␣" << d.B::c << endl; // 8
        cout << "d.C::c␣=␣" << d.C::c << endl; // 9
}
```

Les définitions de classes sont toutes correctes, mais la fonction `main` n'est pas compilable.

Question 1 (2 points) *Pour chaque ligne numérotée de 1 à 9 : la ligne est-elle compilable ? Si oui, qu'affiche-t-elle ? Dans tous les cas, justifiez très brièvement votre réponse.*

- 0,25 par réponse fausse.

1. non-compilable, a est ambiguë.
2. 666 (passé via chaînage de constructeur)
3. 42 (constructeur par défaut de A)
4. non-compilable, b est ambiguë
5. 456 (via B)
6. 12 (via A)
7. 789 (initialisé dans C)
8. non-compilable, c n'existe pas dans B
9. 789 (identique à d.c)

1.2 Macros

On considère le programme C++ ci-dessous (attention, il y a des pièges ...) :

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

#define FOO() \
        cout << "A" << endl; \
        cout << "B" << endl;
```

```
#define BAR(x) x * x

int return_10() {
        cout << "return_10" << endl;
        return 10;
}

int main() {
        int a = 4;
        cout << BAR(return_10() + 2) << endl;
        if (a == 42)
                FOO();
}
```

Question 2 (1.5 points) *Qu'affiche ce programme ? Justifiez brièvement chaque ligne de l'affichage.*

```
return_10
return_10 // deux instances de x dans BAR => deux appels de return_10
32 // 10 + 2 * 10 + 2
B // En dehors du if à cause du manque de do {...} while (0).
```

2 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) {
        sc_event e;
        void f() {
                cout << "f1" << endl;
                wait(e);
                cout << "f2" << endl;
                wait(3, SC_SEC);
                cout << "f3" << endl;
                e.notify();
        }
```

```
        void g() {
                cout << "g1" << endl;
                sleep(2);
                cout << "g2" << endl;
                e.notify();
                cout << "g3" << endl;
                wait(e);
                sleep(1);
                cout << "g4" << endl;
        }
        SC_CTOR(A) {
                SC_THREAD(f);
                SC_THREAD(g);
        }
};

int sc_main(int, char**) {
        A a("a");
        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).

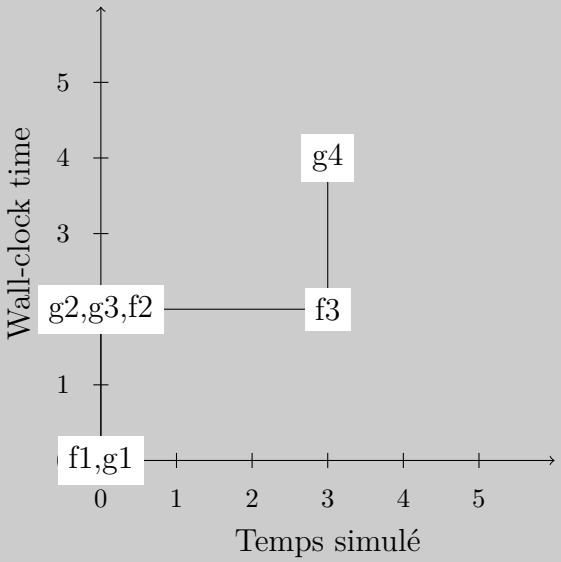
Question 3 (1.5 points) *Combien y a-t-il d'exécution différentes (i.e., produisant des affichages différents) autorisées par SystemC ? Donnez ces traces d'exécution.*

Initialement, les deux processus sont éligibles, et peuvent donc s'exécuter dans n'importe quel ordre. Si g démarre d'abord, la notification de e est perdue et f bloquera sur wait(e). Sinon, f se bloque sur wait(e) et est débloqué par le notify(). Le wait(e) de g n'a pas le problème : le wait() est fait à t=0 dans g et le notify() à t=3, donc le notify réveille forcément le wait. On a donc 2 solutions :

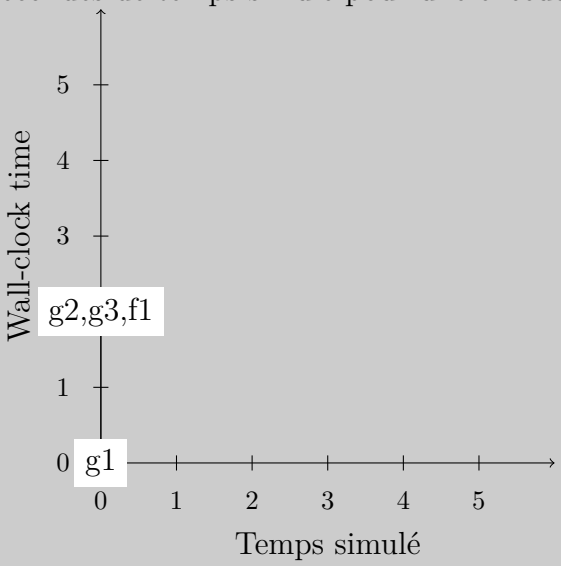
```
f1
g1
g2
g3
f2
f3
g4

g1
g2
g3
f1
```

Question 4 (2 points) *Représentez toutes les 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. Le schéma fera apparaître* `f1`*,* `f2`*,* `f3`*,* `g1`*, ...,* `g4`*. Si on néglige le temps de calcul, combien de temps prendra chaque exécution en "wall-clock time" ? Combien de temps en temps simulé ?*



3 secondes de temps simulé pour une exécution de 3 secondes en wall-clock time.

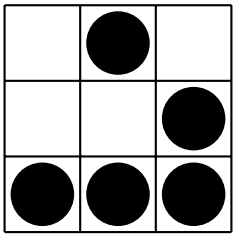


0 seconde de temps simulé pour une exécution de 2 secondes en wall-clock time.

3 Implémentation d'un accélérateur matériel

Dans cette partie, nous allons ajouter un accélérateur matériel à la plate-forme du TP3 (jeu de la vie sur processeur MicroBlaze). Une opération réalisée par le logiciel est de dessiner un « glider » (motif qui se déplace naturellement à l'écran avec le jeu de la

vie) :



Une manière de faire ceci en logiciel est la suivante :

```
#define draw_glider draw_glider_soft

void white_pixel(uint32_t base_addr, int x, int y) {
        uint32_t addr = compute_addr(base_addr, x, y);
        int bit = 31 - x % (sizeof(uint32_t) * CHAR_BIT);

        uint32_t data = read_mem(addr);
        data |= (1 << bit);
        write_mem(addr, data);
}

void draw_glider_soft(uint32_t img_addr, int x, int y) {
        white_pixel(img_addr, x + 1, y + 0);
        white_pixel(img_addr, x + 2, y + 1);
        white_pixel(img_addr, x + 0, y + 2);
        white_pixel(img_addr, x + 1, y + 2);
        white_pixel(img_addr, x + 2, y + 2);
}
```

La fonction `compute_addr` calcule l'adresse du mot mémoire contenant le pixel de coordonnées (x, y) (le corps n'est pas donné mais vous pouvez l'utiliser dans vos réponses).

Dans la fonction `main`, cette fonction est appelée comme ceci :

```
/* glider */
draw_glider(old_img_addr, 10, 10);

/* another glider */
draw_glider(old_img_addr, 50, 10);
```

Question 5 (1 point) *Pourquoi la fonction `white_pixel` fait-elle une lecture puis une écriture ? Expliquez brièvement son fonctionnement.*

La représentation de l'image du TP3 utilise un bit par pixel. Les accès au bus sont sur 32 bits donc on doit lire 32 bits, masquer le bit correspondant au bon pixel, puis écrire le résultat, ce que fait `write_pixel`.

Nous allons maintenant modéliser un composant matériel permettant une implémentation matérielle de `draw_glider`. Nous appellerons ce composant `GLIDER`.

Le gain sur le TP3 sera bien entendu très faible, puisqu'on n'utilise cette fonction que deux fois, au démarrage du programme et que la fonction n'est pas très gourmande. Mais le principe serait le même pour faire un calcul plus lourd qu'un affichage de glider.

La fonction `draw_glider` du logiciel utilisant le composant matériel est définie comme suit. Elle aura exactement le même comportement que `soft_draw_glider`, mais utilisera le composant matériel. Pour plus de clarté, on la nomme `hard_draw_glider` (mais c'est une fonction du logiciel embarqué) :

```
#define draw_glider draw_glider_hard

void draw_glider_hard(uint32_t ptr, int x, int y);
```

Une spécification partielle du composant `GLIDER` est fournie en annexe. On suppose qu'une macro `GLIDER_BASEADDR` est définie (dans tous les fichiers C ou C++) et représente l'adresse de base du composant sur le bus.

Question 6 (0.5 point) *Donnez une implémentation possible de la macro `GLIDER_SIZE` qui représente la taille de la plage d'adresse utilisée par le composant GLIDER sur le bus.*

Réaliste du point de vue HW (décodeur qui ne regarde que les bits de poids fort) :

```
#define GLIDER_SIZE     0x00010000
```

Mais toute valeur supérieure ou égale à 16 est correcte.

3.1 Version simplifiée sans interruptions

Dans cette partie, on fait l'hypothèse simplificatrice que le dessin du glider en mémoire est fait en temps nul et atomiquement par le composant `GLIDER` (le dessin est terminée quand l'écriture sur le registre `START` termine). Cette hypothèse peut être utilisée dans le modèle TLM du composant et dans le logiciel embarqué.

Question 7 (1 point) *Expliquez les rôles respectifs des sockets initiateur et cible du composant `GLIDER`.*

Initiateur : pour pouvoir accéder à la RAM. Cible : pour recevoir les ordres depuis le logiciel embarqué.

Le modèle TLM du composant `GLIDER` est donné par la classe ci-dessous :

```
class GLIDER : public sc_core::sc_module {
public:
        ensitlm::target_socket<GLIDER> target;
        ensitlm::initiator_socket<GLIDER> initiator;

        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_CTOR(GLIDER) {/* */};
private:
        void white_pixel(uint32_t base_addr, int x, int y);

        uint32_t m_addr, m_x, m_y;

        /* Dessine un glider aux coordonnées x, y dans l'image ptr */
        void run(uint32_t ptr, int x, int y);
};
```

Dans un premier temps, on suppose que la méthode `run` est déjà implémentée. Vous pouvez donc l'appeler dans vos réponses.

On donne un squelette d'implémentation des fonctions `read` et `write` du modèle de composant `GLIDER` :

```
tlm::tlm_response_status
GLIDER::write(ensitlm::addr_t a, ensitlm::data_t d)
{
        switch (a) {
        case GLIDER_ADDR:
                // ...
                break;
        case GLIDER_X:
                // ...
                break;
        case GLIDER_Y:
                // ...
                break;
        case GLIDER_START:
                // ...
                break;
        default:
                SC_REPORT_ERROR(name(), "No register at this address");
                return tlm::TLM_ADDRESS_ERROR_RESPONSE;
        }
        return tlm::TLM_OK_RESPONSE;
```

```
}

tlm::tlm_response_status
GLIDER::read(ensitlm::addr_t a, ensitlm::data_t& d)
{
        switch (a) {
        case GLIDER_ADDR:
                d = m_addr;
                break;
        case GLIDER_X:
                d = m_x;
                break;
        case GLIDER_Y:
                d = m_y;
                break;
        case GLIDER_START:
                // ...
                break;
        default:
                SC_REPORT_ERROR(name(), "No register at this address");
                return tlm::TLM_ADDRESS_ERROR_RESPONSE;
        }
        return tlm::TLM_OK_RESPONSE;
}
```

Question 8 (1 point) *Proposez une implémentation pour les cas `GLIDER_ADDR`, `GLIDER_X` et `GLIDER_Y` dans la méthode `write`.*

```
... write(...) { ...
        case GLIDER_ADDR:
                m_addr = d;
                break;
        case GLIDER_X:
                m_x = d;
                break;
        case GLIDER_Y:
                m_y = d;
                break;
```

Question 9 (1 point) *Proposez une implémentation pour les cas `GLIDER_START` des méthodes `read` et `write`.*

```
... write(...) { ...
        case GLIDER_START:
                run(m_addr, m_x, m_y);
                break;

... read(...) { ...
        case GLIDER_START:
                d = 0;
                break;
```

Nous allons maintenant implémenter les méthodes `white_pixel` et `run` du composant.

Question 10 (1 point) *La méthode* ***white_pixel*** *du composant* ***GLIDER*** *fait la même chose que la fonction du même nom du logiciel embarqué (ci-dessus). Donnez-en une implémentation. Si vous le souhaitez, vous pouvez répondre en donnant uniquement les différences avec l'implémentation du logiciel.*

```
void GLIDER::white_pixel(uint32_t base_addr, int x, int y) {
        tlm::tlm_response_status status;
        uint32_t addr = compute_addr(base_addr, x, y);
        int bit = 31 - x % (sizeof(uint32_t) * CHAR_BIT);

        uint32_t data;
        status = initiator.read(addr, data);
        assert(status == tlm::TLM_OK_RESPONSE);

        data |= (1 << bit);
        status = initiator.write(addr, data);
        assert(status == tlm::TLM_OK_RESPONSE);
}
```

Question 11 (1 point) *Donnez une implémentation de la méthode* ***run****. Là aussi, vous pouvez répondre en ne donnant que les différences avec les fonctions existantes ailleurs.*

Identique à la méthode `draw_glider_soft` du soft.

Question 12 (1 point) *Que faut-il ajouter à la fonction* ***sc_main*** *?*

```
        Glider glider("glider");
        bus.initiator(glider.target);
        bus.target(glider.initiator);
        bus.map(glider.target, GLIDER_BASEADDR, GLIDER_SIZE);
```

Question 13 (0.5 point) *Est-il nécessaire de modifier la couche d'abstraction du matériel (`hal.h`), et pourquoi ? Si oui, quelles modifications faut-il faire ?*

Non, rien à changer. Les accès au GLIDER se font via `read_mem` et `write_mem`.

Question 14 (1 point) *Donnez une implémentation de la fonction `draw_glider_hard` du logiciel, utilisant le composant matériel `GLIDER`.*

```
void draw_glider_hard(uint32_t ptr, int x, int y) {
        write_mem(GLIDER_BASEADDR + GLIDER_ADDR, ptr);
        write_mem(GLIDER_BASEADDR + GLIDER_X, x);
        write_mem(GLIDER_BASEADDR + GLIDER_Y, y);
        write_mem(GLIDER_BASEADDR + GLIDER_START, 1);
}
```

3.2 Version avec gestion propre des interruptions

La version proposée ci-dessus a l'avantage d'être simple à modéliser, mais n'est pas réaliste du point de vue du vrai système : le calcul fait au moment de l'accès au registre `START` est supposé instantané et atomique.

Question 15 (1 point) *Expliquez la différence entre un calcul instantané et un calcul atomique en SystemC. L'un implique-t-il l'autre ?*

Instantané : ne laisse pas le temps simulé s'exécuter. Atomique : ne rend pas la main. L'atomicité implique l'instantanéité mais pas l'inverse (`wait(SC_ZERO_TIME)` permet de rendre la main instantanément par exemple).

Pour pouvoir nous passer de cette hypothèse simplificatrice, le composant doit fournir un moyen de savoir quand le calcul est terminé, et le logiciel doit utiliser cette information pour attendre le résultat du calcul. La solution proposée est que le composant `GLIDER` va notifier le processeur via une interruption quand le calcul est terminé.

Question 16 (1 point) *Que faut-il ajouter à l'interface du composant `GLIDER` pour permettre ceci ?*

Un port d'IRQ en sortie, et un registre `FINISHED` qui permet de savoir si l'opération est terminée.

Question 17 (1 point) *Pour adapter le comportement du composant, faut modifier la manière de gérer les transactions sur le registre* ***`START`****. Quelles modifications faudrait-il faire dans le composant* ***`GLIDER`*** *? Répondez au choix en écrivant du code ou en décrivant précisément les constructions SystemC à utiliser (quel processus fait quoi, quelle méthode appelle qui, ... ?).*

Il faut ajouter un processus SystemC qui exécute la méthode C++ `run` quand elle reçoit un évènement `start`. Le traitant de transaction de `write` pour `START` notifiera cet évènement pour réveiller le processus. Un `wait` suivra l'appel à `run` puis le signal d'IRQ est levé.

Question 18 (1 point) *Proposez une modification du logiciel embarqué qui permette une gestion correcte des interruptions. Attention, le composant n'est pas le seul à pouvoir envoyer des interruptions !*

Il faut ajouter une attente d'interruption à la fin de la fonction `draw_glider` :

```
while (!read_mem(GLIDER_BASEADDR + GLIDER_FINISHED)) {
        wait_for_irq();
}
```

On suppose maintenant que le processeur de notre plateforme fait tourner un OS multi-tâche, et que notre logiciel embarqué est multi-thread.

Question 19 (1 point) *Que se passe-t-il sur la vraie plateforme deux threads appellent* ***`draw_glider_soft()`*** *en parallèle ? Et* ***`draw_glider_hard()`*** *? Dans quelle situation aura-t-on des problèmes précisément ?*

`soft_draw_glider` est thread-safe si tous les appels concurrents agissent sur des zones de l'écran différentes (sur des mots de 32 pixels différents), mais pas `draw_glider_hard` qui ne correspond qu'à une ressource matérielle partagée. Il faudrait ajouter un mutex autour de `draw_glider` par exemple.

Question 20 (1.5 points) *Verra-t-on les problèmes mentionnés à la question précédente en simulation (version simplifiée et version avec gestion des interrupts) ? Verra-t-on des problèmes différents en simulation native ou en intégrant le logiciel avec un ISS ? Pourquoi ?*

Si `soft_draw_glider` est appelé sur deux zones de l'écran qui partagent des mots de 32 bits, on ne verra pas le problème en simu native (pas de wait entre `read` et `write` dans `white_pixel`). On le verra en simu avec ISS si l'ISS rend la main entre `read` et `write`.

On a les problèmes avec ou sans gestion des IT : la concurrence entre les accès aux registres avant `START` se passe de la même manière. Avec gestion d'IT, on ajoute un problème : un thread pourrait se réveiller sur l'IT déclenchée par l'autre.

Pour le problème de concurrence sur `hard_draw_glider`, idem mais c'est si l'ISS rend la main entre les accès en écriture aux registres X, Y et ADDR.

22.5
22.5

4 Annexe : composant GLIDER

Le composant GLIDER est destiné à être connecté à un bus Xilinx AXI. Il possède les entrées/sorties suivantes :

— Interface initiateur compatible AXI
— Interface cible compatible AXI

Fonctionnalités

Le port cible du GLIDER permet de dessiner le motif « glider » du jeu de la vie dans une image 1 bit par pixel (1 = blanc, 0 = noir).

Fonctionnement interne

Lorsqu'une écriture est réalisée dans le registre `START`, le composant lance le dessin du motif aux coordonnées `X`, `Y` dans l'image située à l'addresse `ADDR`.

Récapitulatif des registres

Adresse relative	**Type**	**Nom**	**Description**
`0x00`	Lecture/écriture	`GLIDER_ADDR`	Adresse de départ
`0x04`	Lecture/écriture	`GLIDER_X`	Abscisse du motif
`0x08`	Lecture/écriture	`GLIDER_Y`	Ordonnée du motif
`0x0c`	Lecture/écriture	`GLIDER_START`	Démarrer l'initialisation

Registres `GLIDER_ADDR`, `GLIDER_X`, `GLIDER_Y`

Permettent de stocker les paramètres de l'initialisation.

Registre `GLIDER_START`

Écrire dans ce registre lance le dessin du motif. Lire dans le registre renvoie toujours 0.