**École Polytechnique de Montréal**

|  |
| --- |
| Polytechnique Montréal - DGIGL |
| Laboratoire #1 : Modélisation avec SystemC |
| INF8500 Conception et vérification de systèmes embarqués |
| Séance 1 (Lab 1, partie 1): modélisation et raffinement d’une application de trie |
|  |

**Département de Génie Informatique**

# Objectif

L’objectif de ce laboratoire est de comprendre la méthodologie de conception haut niveau de systèmes embarqués, en utilisant la librairie de simulation SystemC.

Plus précisément, les objectifs du laboratoire sont :

* S’initier à la librairie SystemC
* Se familiariser au développement de SoC avec une méthodologie de conception haut niveau
* Comprendre le concept d’adaptateur (wrapper)
* Connaître les différents niveaux d’abstraction
* Mettre en pratique les étapes de raffinement

Ce laboratoire consiste à la modélisation et au raffinement d’une application composée de trois modules à l’aide de SystemC. Premièrement, vous allez développer chaque module à un haut niveau d’abstraction *Un-Timed Functional.* Par la suite, vous allez modifier le contenu des modules (raffinement) pour avoir un comportement plus détaillé à un niveau d’abstraction plus bas. Vous ferez 2 raffinements.

# Mise en contexte

Lors de la conception d’un système embarqué, il y a plusieurs étapes à respecter avant d’obtenir un produit final. Certes, il est possible de directement bâtir l’application sur une puce de développement avec un langage RTL, mais cette avenue est souvent problématique, car la source des problèmes peut provenir d’une multitude de facteurs (i.e mauvais logique du code applicatif, difficulté de développement à bas niveau, difficulté à changer l’architecture, difficulté à tester les composants matériels, etc.) ce qui rend le débogage ardu. Dans le but d’accélérer les phases de développement des systèmes embarqués, la modélisation à haut niveau du système à l’aide de la librairie SystemC est une étape importante puisqu’elle permet de valider ou d’infirmer les spécifications, de corriger les bogues applicatifs, de faire une vérification fonctionnelle, entre outres. De plus, il est plus facile de déceler et corriger les problèmes à cette étape qu’aux étapes subséquentes.

SystemC est une librairie de C++ qui permet la modélisation des matériels à plusieurs niveaux d’abstraction, incluant le RTL (*Regiser Transfert Level*) à très bas niveau ainsi comme le UTF (*Un-Timed Functional*) à très haut niveau. Ceci permet aux développeurs de garder un même langage d’un bout à l’autre du flot de conception. La première étape du flot de conception consiste à décrire les modules sans détails concernant l’architecture, sans horloge et sans les détails de communication, à un haut niveau d’abstraction. Le but de cette étape est de faire une vérification fonctionnelle du système, et ainsi valider l’algorithme de calcul de chaque module. En appliquant successivement les étapes de raffinement, il est possible d’obtenir un modèle de très bas niveau, communément appelé RTL. Pour chaque module raffiner on fera le raffinement du calcul (computation) et des communications (interfaces). Par exemple, Bubble AT-CT indiquera un raffinement sur le module Bubble de niveau AT pour le calcul et CT pour les communications. Notez que si un module ne fait que communiquer (i.e. pas de calcul), comme le module Lecteur, on aura Lecteur NA-CT (où NA indique Not Applicable).

La modélisation à haut niveau est une étape qui apporte beaucoup davantage lors du processus d’élaboration des systèmes embarqués.

L’application visée dans ce laboratoire est le tri. Il faut savoir qu’aujourd’hui encore il existe de la recherche pour faire ses implications de tri haute performance. Une étude intéressante est disponible sur arXiv[[1]](#footnote-1). Dans ce laboratoire nous nous simplifierons la vie avec la modélisation d’un simple tri en bulle en sachant que ce dernier peut être remplacer par quelque chose de beaucoup plus complexe.

# Conception du système

La première étape de ce laboratoire consiste à implémenter un modèle SystemC au niveau d’abstraction **UTF** qui exécute l’algorithme de tri à bulle dans un ensemble de valeurs.

Le circuit est composé de 3 modules :

* Une mémoire de données (DataRAM qui elle-même contient RAM)
* Un lecteur
* Module tri à bulle (qu’on pourrait éventuellement voir comme un coprocesseur)

La deuxième étape est de raffiner le module lecteur, le module tri à bulle ainsi que la communication entre ces deux modules au niveau d’abstraction **AT** (*Approximate Time*). Finalement, vous devez raffiner au niveau **CT** (C*ycle Timed)* le module tri à bulle (mais pas les communications). Plus précisément, dans cette troisième étape, le module doit exécuter une machine à états.

## Description complète UTF

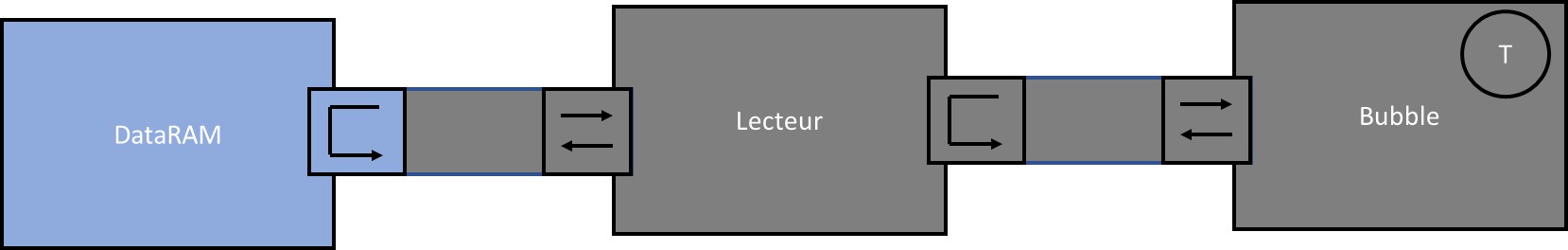


Figure 1 Schéma circuit tri à bulle au niveau *Un-Timed functional.* Le gris indique le code à compléter et le bleu le code fourni.

### Lecteur

|  |  |  |
| --- | --- | --- |
| Type | Nom | Description |
| sc\_port<LMBIF> | dataPortRAM | Port pour la mémoire de donnée |

Le module lecteur sert à interfacer le module tri à bulle à la mémoire de données. Il agit comme un *wrapper* de Bubble. La communication entre le module lecteur et la mémoire est possible à l’aide du port dataPortRAM. Ce port supporte les opérations décrites dans l’interface LMBIF. De plus, ce module hérite de l’interface interfaceRead, cette dernière permettra au module tri à bulle d’envoyer les requêtes de lecture au module lecteur. **(Voir l’exemple de comment implémenter une interface et de comment faire le branchement d’un sc\_port dans l’annexe 1)**

* Fonctionnement interne :
  + Lire la mémoire à l’adresse demandée
  + Envoyer la donnée lue au module tri à bulle

Rappel: au démarrage de la simulation, tous les modules sont appelés 1 fois

### Tri à bulle

|  |  |  |
| --- | --- | --- |
| Type | Nom | Description |
| sc\_port<InterfaceRead> | readPort | Port pour le module lecteur |

Le module tri à bulle doit lire les valeurs qui sont sauvegardées dans la mémoire de données, et ensuite il doit trier les valeurs lues à l’aide d’un algorithme tri à bulle. Vous n’avez pas à coder l’écriture des données dans la mémoire, mais après le tri, les valeurs triées doivent être affichées à l’écran. La mémoire de donnée est initialisée avec le fichier « chiffre.hex » (le fichier contient les données qui vont être chargées dans la mémoire). Plus précisément, le premier nombre indique le nombre d’éléments à trier tandis que les nombres subséquents sont les éléments à tirer.

* Fonctionnement interne :
  + Le premier nombre dans le mémoire est à l’adresse 0d0.
  + Le deuxième nombre se situe à l’adresse 0d4, etc...
  + Après avoir lu tous les nombres, les valeurs doivent être triées.
  + Le résultat est affiché

## Raffinement1 : Lecteur NA-CT et Bubble AT-CT

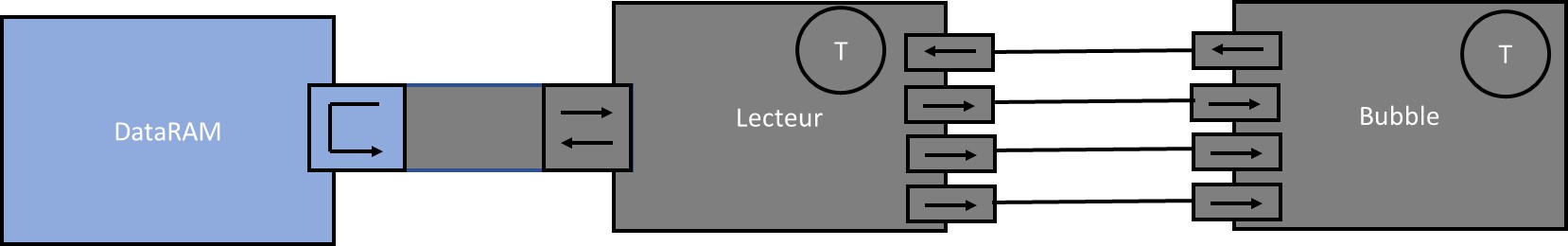


Figure 2 Schéma circuit tri à bulle au niveau Approximate Timed. Le gris indique le code à compléter et le bleu le code fourni.

### Lecteur

|  |  |  |
| --- | --- | --- |
| Type | Nom | Description |
| sc\_in\_clk | clk | Horloge |
| sc\_port<LMBIF> | dataPortRAM | Port pour la mémoire de donnée |
| sc\_out<unsigned int> | data | Donnée |
| sc\_in<unsigned int> | address | Adresse |
| sc\_out<bool> | ack | Accusé de réception |
| sc\_in<bool> | request | Requête |

Le module lecteur sert toujours à interfacer le module tri à bulle à la mémoire de données. Toutefois, maintenant la communication entre ce module et le module tri à bulle est plus raffinée. Une horloge et plusieurs signaux ont été ajoutés dans le module. La synchronisation entre les deux modules se fait par un protocole simple de *handshaking*. **(Voir exemple handshaking dans l’annexe 1)**

* Fonctionnement interne :
  + Attendre une requête
  + Lire la valeur de l’adresse
  + Demander à la mémoire la donnée à l’adresse lue
  + Envoyer un accusé de réception
  + Enlever l’accusé de réception

### Tri à bulle

|  |  |  |
| --- | --- | --- |
| Type | Nom | Description |
| sc\_in\_clk | clk | Horloge |
| sc\_out<unsigned int> | address | Adresse |
| sc\_in<unsigned int> | data | Donnée |
| sc\_out<bool> | request | Accusé de réception |
| sc\_in<bool> | ack | Requête |

Le module tri à bulle doit lire les valeurs qui sont sauvegardées dans la mémoire de données, et ensuite il doit trier les valeurs lues à l’aide d’un algorithme tri à bulle. À la fin, les valeurs triées doivent être affichées. Tel qu’indiqué précédemment, la synchronisation avec le module lecteur se fait par un protocole simple de *handshaking*.

* Fonctionnement interne :
  + Envoyer l’adresse à être lue
  + Envoyer une requête
  + Attendre un accusé de réception
  + Lire la donnée reçue
  + Enlever la requête

Finalement, chaque itération de l’algorithme du tri à bulle doit prendre un cycle d’horloge de la simulation. Pour vérifier que vous respectez cette spécification, on vous demande d’ajouter à votre code au bon endroit les affichages suivants :

*cout << endl << "Bubble Begin " << sc\_time\_stamp() << endl ;*

*cout << "Bubble end " << sc\_time\_stamp() << endl ;*

## Raffinement2 : Lecteur NA-CT et Bubble CA-CA

Graphical user interface

Description automatically generated with medium confidence

Figure 3 Schéma circuit tri à bulle au niveau Cycle Timed. Le gris indique le code à compléter et le bleu le code fourni. Ici le thread de Bubble doit être une machine à états.

### Pour la dernière étape de raffinement, nous allons raffiner seulement le module tri à bulle. Le module Lecteur ne sera pas touché. L’implémentation du module Lecteur de l’étape 2 peut être réutilisée. Le module tri à bulle, quant à lui, doit être ré-implémenté au niveau *Cycle Accurate*, proche du RTL, sous forme d’une machine à état. Plus précisément, la lecture des données, le tri ainsi que l’affichage doivent se faire dans la machine à état. Vous êtes libre d’implémenter la machine à état de la façon que vous voulez.

Encore une fois, chaque itération de l’algorithme du tri à bulle devra prendre un cycle d’horloge de la simulation. Pour vérifier que vous respectez cette spécification, on vous demande d’ajouter à votre code au bon endroit les affichages suivants :

*cout << endl << "Bubble Begin " << sc\_time\_stamp() << endl ;*

*cout << "Bubble end " << sc\_time\_stamp() << endl ;*

# Travail à réaliser

Vous devez compléter les fichiers :

* Reader.h et Reader.cpp
* Bubble.h et Bubble.cpp
* Main.cpp

Le code dans le répertoire code/UTF doit implémenter le niveau d’abstraction UTF, etc... Si vous voulez tester votre système avec d’autres valeurs, vous pouvez modifier le fichier « chiffre.hex ».

Dans le but de vous simplifier la tâche, le code de base fourni contient un *makefile* qui permet de compiler votre projet sur Linux. Afin de compiler votre projet, lancer simplement la commande *make* dans le répertoire du laboratoire.

Vous êtes libre d’utiliser l’IDE de votre choix (vscode est installé sur les ordinateurs de Poly).

Malgré tous les choix que vous ferez, assurez-vous que votre laboratoire compile et fonctionne dans les laboratoires du cours puisque c’est à cet endroit que le chargé de laboratoire validera votre travail.

N.B. Ajoutez les cout nécessaires pour que la situation la séquence d’évènement soit visible.

# Évaluation

Ce laboratoire s’inscrit comme la partie 1 du laboratoire no 1. Vous disposez de 2 semaines pour effectuer ce travail. Une partie 2 viendra s’ajouter avec 2 semaines pour la faire (4 semaines en tout). La remise et le rapport se feront directement dans Microsoft Teams via la fonction devoirs. Nous vous communiquerons les détails de la remise dans les jours à venir.

# Questions

**Question # 1** Quel est le lien entre DataRAM et RAM?

**Question # 2** Justifier le (ou les) rôles de Lecteur? Pourquoi ne pas avoir connecté directement Bubble à DataRAM?

**Question # 3** Sachant que l’implémentation finale de Bubble est en logiciel, quel raffinement (entre 1 et 2) est le plus justifié?

**Question # 4** Justifiez le choix du type de thread utilisé pour Bubble dans le raffinement 2 (avantages/désavantages).

# Barème correction

|  |  |
| --- | --- |
| * Partie UTF (code + exécution) * Partie AT (code + exécution) * Partie CT(code + exécution) * Respect spécification système * Question #1 * Question #2 * Question # 3 * Question # 4 | /1  /2  /2  /1  /1  /1  /1  /1 |
| **TOTAL** | /10 |

Le laboratoire compte pour 7.5 points de la note finale.

**Bon travail!**

# Annexe 1

|  |
| --- |
| *// Variable*  *sc\_signal<bool> sEnable;*  *// On effectue le branchement*  *Instance\_cd.enable(sEnable);*  *Instance\_auto.enable(sEnable);* |

Exemple de branchement

|  |  |
| --- | --- |
| Interface\_audio.h | cd.h |
| …  class Interface\_audio : public virtual sc\_interface  {  …  private:  virtual void play() = 0;  virtual void stop() = 0;  } | **…**  **class cd : public sc\_module, public Interface\_audio**  **{**  **…**  **private:**  **virtual void play() ;**  **virtual void stop() ;**  **…**  **};** |
|  |  |

Exemple comme implémenter une interface

|  |  |
| --- | --- |
| auto.h | Main.cpp |
| …  class auto : public sc\_module  {  public:  sc\_port<Interface\_audio> cdPort;  …  } | **…**  **main**  **{**  **…**  **Auto instance\_auto();**  **Cd instance\_cd();**  **Instance\_auto.cdPort (instance\_cd);**  **…**  **};** |
|  |  |

Exemple branchement sc\_port

|  |  |
| --- | --- |
| auto.cpp | cd.cpp |
| …  // Envoie de l’adresse  address.write(addr);  enable.write(true);  // Syncronisation  do{  wait(clk->posedge\_event()  }while(!ack.read() );  // Poursuite du traitement  enable.write(false);  … | **…**  **do{**  **wait(clk->posedge\_event()**  **}while(!enable.read());**  **// On lit l’address**  **addr = address.read();**    **// Syncronisation**  **ack.write( true );**  **…** |

Exemple de synchronisation (*handshaking*)

1. R. Kastner and al., Parallel Programming for FPGAs, arXiv:1805.03648v1 [cs.AR] 9 May 2018 [↑](#footnote-ref-1)