INF3610 - TP2

Présenté à

Jeff Falcon, chargé de laboratoire

Par

Michael Chidiac, 1466616

Olivier Moussavou Cotte, 1538493

École Polytechnique de Montréal

30 octobre 2013

**Barème correction**

**EXÉCUTION**

 Fonctionnement de l’exécutable  /4

**TOTAL EXÉCUTION** /4

**CODE SOURCE**

 Contenu des modules  /3

 Contenu des adaptateurs  /2

 Contenu du SimpleBusLT  /1

 Commentaire et clarté du code  /1

 Respect de l’énoncé /1

**TOTAL RAPPORT** /8

**RAPPORT**

 Présentation générale, introduction, conclusion /1

 Fonctionnement du système /2

**QUESTIONS**

 Question 1 /2

 Question 2 /2

 Question 3 /1

**TOTAL RAPPORT** /8

**NOTE GLOBALE /20**

**INTRODUCTION**

Afin de nous familiariser avec les systèmes à temps réels, ce quatrième laboratoire vise à nous faire développer un système utilisant la librairie SystemC que nous avons utilisé lors du troisième laboratoire. Nous allons ainsi approfondir nos connaissances de développement de systèmes embarqués à haut niveau en nous attaquant au Transaction-Level-Modeling que nous offre SystemC.

Le système à concevoir est le même qu’au troisième laboratoire, c’est-à-dire une simulation du jeu du bonhomme pendu. Ce système ayant un processeur, nous allons le représenter par un ISS, ou Instruction Set Simulator, nous permettant ainsi de prévoir le fonctionnement sur puce de notre module. Deux autres modules, soit le co-processeur et la console vont aussi devoir implémenter afin de communiquer avec l’utilisateur et de laisser jouer. De plus, un bus générique du standard TLM 2.0 va être utilisé pour gérer les communications entre les différents modules.

**FONCTIONNEMENT DU SYSTÈME**

Nous devons tout d’abord créer des instances des modules ainsi que chacun de leur wrapper dans la fonction main. Une instance SimpleBusLT, de DataRAM ainsi que de InstRAM est aussi créé ici. Une horloge globale est déclaré et connecté aux trois modules.

Afin de pouvoir créer les connections entre les différents modules, des signaux sont déclarés puis utilisés afin de connecter les ports des wrappers aux modules. Par exemple, nous avons le port sc\_in<bool> Wrapper\_Console\_Ready\_InPort du module wrapper de console (qui se trouve dans le fichier wrapper\_console\_TLM.h) est connecté par l’intermédiaire du signal sc\_signal<bool> pont\_Ready\_Console\_Port (déclaré dans le main) au port sc\_out<bool> Console\_Ready\_OutPort de la console (dans le fichier Console.h). Tous les signaux qui nous sont demandés dans la description du laboratoire sont définis dans le fichier header correspondant et sont connectés dans le main.

Pour ce qui est des sockets, nous avons dû connecté un initiator du Wrapper\_Processor au target du bus, et un des deux initiators du bus a chacun des target de Wrapper\_CoProcessor et Wrapper\_Console. La connection se fait à l’aide du code suivant : instance\_wrapper\_Processor.socket.bind(instance\_simplebus.target\_socket). Il faut aussi connecté le dataport du bus à l’instance du dataram.

Les connections entre les modules et leur wrappers se fait de façon très simple en appelant des méthodes déjà implémenté dans la librairie SystemC tel que read() et write(). Nous avons par contre écrit une méthode busLT\_Slave\_write qui nous permet de transmettre l’information reçu du bus dans le wrapper au module. En utilisant une méthode write(), nous signalons que nous nous apprêtons à écrire en mettant le Enable\_OutPort à false, en écrivant l’adresse des données dans le Data\_OutPort puis en signalant que l’écriture est terminé en remettant le signal Enable\_OutPort a True. On attend par la suite l’acquitement de l’écriture grâce à ce bout de code qui fait en sorte que l’on bloque jusqu'à ce qu’il y ai un événement sur le port : wait(Wrapper\_Console\_Ready\_InPort.default\_event()). Une fois l’événement reçu nous pouvons transmettre nos données dans le port Data\_Out et finalement remettre le port Enable\_OutPort a false. Nous attendons ensuite un événement sur le port Ready\_InPort avant de sortir de la fonction. Cette description était spécifique au module wrapper\_Console\_TLM mais s’applique pour toutes les communications entre les wrappers et leur module. Ces autres modules ont aussi une fonction servant à lire qui est implémenté de façon similaire à l’écriture.

D’autre part, la communication entre les wrappers et le bus doit être effectué à l’aide d’un appel à un fonction bloquante b\_transport() qui nous a été fournis dans le code de départ. Ces appels sont fait à partir de l’instance de SimpleBusLT qui nous a aussi été fournis.

La logique utilisée à l’intérieur des modules Console, Processor et CoProcessor sont similaires à celles développés au niveau AT du laboratoire 3. Pour ce qui est de Console, une fonction thread qui attend la réception de données. Une fois reçu, une machine à état va faire apparaitre à l’écran différentes informations tel que le mot actuel, le nombre d’erreur ou la fin du match, tout dépendamment de l’adresse reçu lors de la communication.

Dans le CoProcessor, une fonction TryNewLetter nous est fournis qui permet à l’utilisateur de vérifier si une lettre se trouve réellement dans le mot. La fonction thread s’occupe de gérer la lecture et l’écriture, où en d’autres mots la communication avec le wrapper. On attend de recevoir un true dans le port CoProcessor\_RW\_InPort et, une fois reçu, on vérifie le port CoProcessor\_RW\_InPort pour savoir s’il s’agit d’une lecture ou d’une écriture. Par la suite, selon l’adresse lu dans le CoProcessor\_Ready\_InPort ou le CoProcessor\_Ready\_OutPort, on effectue différente opérations.

**RÉPONSES AUX QUESTIONS**

**QUESTION 1**

Une des différences majeures entre SC\_THREAD et SC\_METHOD est que SC\_METHOD n’a pas son propre fil d’exécution. À chaque fois qu’en événement qui se trouve dans la liste de sensibilité de SC\_METHOD est déclenché, le code est entièrement exécuté. Nous ne pouvons donc pas faire d’appels à wait() ou nous bloquerions la simulation au complet.

Contrairement à SC\_METHOD, SC\_THREAD a sa propre fil d’exécution et est modelé dans une boucle. Ainsi, lorsqu’un appel à wait() est faut, cela ne fait que mettre en arrêt le fil d’exécution dès qu’un événement de la liste de sensibilité est déclenché, l’exécution peut alors continué à la prochaine opération se trouvant après le wait().

Le bus TLM fait des appels bloquants wait() et c’est pourquoi il ne serait pas possible de travailler avec SC\_METHOD.

**QUESTION 2**

L’opération nop est une opération de latence. Elle ne fait aucune autre opération que tout simplement attendre. En attendant ainsi, nous nous assurons que la valeur dans le registre r10 a réellement décrémenté de 1 avant d’effectuer le branchement de la ligne 54.

**QUESTION 3**

Tout d’abord, il est important de mentionner que la classe sc\_buffer est dérivée de sc\_signal. Il supporte donc toutes les méthodes définis dans sc\_signal. Par contre, la façon que sc\_buffer réagit lors d’un wait() est différente de celle d’un sc\_signal. Si nous avons la valeur 0 dans un port et nous attendons de recevoir une valeur différente, un sc\_signal ne continuera pas tant que nous ne recevons pas une valeur différente. Par contre, un sc\_buffer réagit à tout évènement, il se réveillera donc si il reçoit à nouveau la valeur 0 qui est la même valeur.

**CONCLUSION**

Dans ce laboratoire, nous avons pu implémenter un système permettant de jouer au bonhomme pendu grâce à la librairie SystemC. Nous avons effectué tous les branchements nécessaires afin de réaliser la communication entre les composantes ainsi que développer la logique de traitement des données à l’intérieur même des modules.

Finalement, nous comprenons mieux l’utilité de faire des simulations qui font abstractions de multitudes de couches afin de faciliter le développement de système embarqué. L’environnement de développement Visual Studio est aussi beaucoup plus facile à utiliser que la plupart des débuggeurs qui font rouler du code sur des cartes FPGA. La compréhension de la librairie à pris du temps, mais une fois terminé, la développement fût assez rapide. Nous avons eu besoin d’à peu près 14 heures afin de compléter ce laboratoire.