ÉCOLE POLYTECHNIQUE DE MONTRÉAL

INF3610 : Laboratoire 3

Introduction à SystemC

Frédéric Fortier – Eva Terriault
Avec portions théoriques par Arnaud Desaulty
10/18/2017

Table des matières

I. Introduction 2
1. Objectif 2
2. Mise en contexte 2
II. Les concepts de base de SystemC 3
Une vue d'ensemble des capacités de la bibliothèque 3
Les modules 3
Méthode Vs Thread 3
Communication inter-modules : ports et canaux 4
III. Filtre de Sobel 5
IV. Travail à réaliser (VOIR EXEMPLES DANS L'ANNEXE 1) 7
V. Description des modules 8
1. Description des modules UTF 8
2. Description des modules AT/Cycle Accurate 11
2.1 Description des modules AT/CA de base 11
2.2 Description des modules AT/CA avec structure en ligne à délai 13
VI. Questions 16
VII. Barème et rendu 16
Annexe 1 : Exemples 17
Annexe 2 : Détails de la structure de ligne à délai 22

I. Introduction

1. 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 spécifiques du laboratoire sont :

- s'initier à la librairie SystemC,
- se familiariser au développement de système sur puce (SoC) avec une méthodologie de conception haut niveau,
- connaître les différents niveaux d'abstraction, et
- mettre en pratique les étapes de raffinement.

Vous allez développer et ensuite raffiner une application composée de trois modules à l'aide de SystemC. Autrement dit, vous allez développer chaque module à un haut niveau d'abstraction (*Un-Timed Functional*) et par la suite vous allez modifier le contenu des modules pour avoir un comportement plus détaillé à des niveaux d'abstraction plus bas, soit partiellement *Approximate Timed* (au niveau de l'exécution de l'algorithme) et partiellement *Cycle Accurate* (au niveau des communications). Ces niveaux seront introduits en classe.

2. 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 (ex. mauvaise logique du code applicatif, difficulté de développement à bas niveau, difficulté à changer l'architecture, température élevée du FPGA, 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*) très bas niveau ainsi que, par exemple, l'UTF (Un - Timed *Functional*) 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 (le RTL).

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

II. Les concepts de base de SystemC

Une vue d'ensemble des capacités de la bibliothèque

SystemC permet la création de plusieurs nouveaux éléments. Les **modules** vous permettent de partitionner votre code en plusieurs segments séparés et sont en réalité des objets C++ héritant de la classe *sc_module*. La principale différence entre un module et une classe classique est que le module vous permet de déclarer certaines des méthodes de ces classes en tant que *sc_thread* et *sc_method*. Ces **méthodes** et **threads** sont **sensibles au temps** et peuvent s'exécuter de **manière concurrente** lors de la simulation. Les modules communiquent avec l'extérieur via des **ports**. Ces ports sont reliés entre eux par des **canaux**.

Les modules

Comme cité précédemment, les modules encapsulent une partie du code que vous voulez tester. Pour qu'une de vos classes soit considérée comme un module, il faut qu'elle dérive de la classe *sc_module.* Une fois cela fait, vous pouvez alors déclarer des sc_method et sc_thread à l'intérieur de cette classe. Pour ce faire, vous devez déclarer en privé dans votre classe *SC_HAS_PROCESS(**nom_de_votre_classe**)*. Vous pouvez alors dans le constructeur de votre classe spécifier quelles méthodes de votre classe sont des threads (en appelant *SC_THREAD(**nom_de_votre_méthode**)*) et lesquelles sont des méthodes SystemC (en appelant *SC_METHOD(**nom_de_votre_méthode**)*). Par la suite, vous pouvez déclarer à quels ports vos méthodes et threads seront sensibles. Ces sensibilités indiqueront au simulateur quand démarrer ou redémarrer les threads et méthodes que vous avez déclarés. Pour affilier une sensibilité, après avoir déclaré un thread ou une méthode, il suffit d'écrire *sensitive << **nom_de_votre_port** << **nom_2** ;* . Vous remarquerez l'opérateur de flux permet d'affilier plusieurs ports à une méthode ou thread.

Méthode Vs Thread

Les questions que l'on peut se poser à présent sont : comment choisir de définir une de mes méthodes en tant que *sc_thread* ou *sc_method* ? Quelles sont les différences entre ces deux solutions?

La principale différence entre ces deux conceptions est **leur sensibilité** lors de la simulation. Une méthode, une fois lancée, doit **s'exécuter intégralement** et **ne peut pas être stoppée** en cours de route (imaginez un circuit matériel, il ne peut s'arrêter). Par contre, **la méthode sera appelée à chaque fois qu'un événement se produira sur sa liste de sensibilité** (vue plus haut).

Le thread, en revanche, comme dans µC, **n'est appelé qu'une fois**, en début de simulation, et **tournera indéfiniment si vous ne l'arrêtez à l'aide d'éléments de synchronisation** (attente sur un des évènements de la liste de sensibilité, délai sur l'horloge, etc…). Le thread **doit** posséder une boucle infinie dans son corps de méthode puisqu'il n'est appelé qu'une fois.

Si le thread est plus simple d'utilisation (permet de décrire toute une communication d'un seul tenant, avec des éléments de synchronisation), il ne pourra servir que pour les niveaux d'abstractions les plus hauts du flot de conception de SystemC car il modélise mal le comportement réel d'un circuit électronique, contrairement aux *sc_method* qui forcent cet aspect. Nous verrons des exemples en classe.

Communication inter-modules : ports et canaux

Pour expliquer le fonctionnement les communications entre modules, il est plus commode de décrire en premier le fonctionnement des canaux. Les **canaux** sont des classes dérivées des classes *sc_channel* ou *sc_prim_channel*. De plus, ces classes peuvent implémenter des **interfaces** qui spécifient comment le canal doit communiquer. Pour clarifier les choses, nous allons donc étudier un exemple :

```
52  template <class T>
53  class MyChannel_IF : sc_interface
54  {
55      public:
56          // Notez les "=0" signifiant la virtualité pure des méthodes de cette classe
57          virtual T read() = 0;
58          virtual void write (T value) = 0;
59  };
60
61  template <class T>
62  class MyChannel : sc_channel, MyChannel_IF<T>
63  {
64      public:
65          //Cette classe va devoir implémenter les méthodes read et write
66          virtual T read();
67          virtual void write (T value);
68  }
```

Extrait de code 1 : interface et canal

À l'extrait de code 1, nous créons une classe *MyChannel* dans le but d'en faire un canal. Pour ce faire, nous la faisons hériter de *sc_channel* et d'une classe dérivée de *sc_interface*, *MyChannel_IF*. La virtualité pure des fonctions de notre interface va forcer son implémentation dans notre canal. L'intérêt d'une telle façon de faire est triple :

Tout d'abord, cela permet de définir le fonctionnement dans les grandes lignes de notre canal (ici, notre canal doit pouvoir stocker une valeur lorsque l'on appelle *write()* et la rendre disponible lorsque l'on appelle *read()*).

De plus, cela permet d'occulter l'implémentation à l'utilisateur du canal (il n'a pas besoin de savoir comment le canal fait pour stocker cette donnée). L'utilisateur sait seulement qu'il peut utiliser les fonctions de l'interface.

Enfin, cela permet au programmeur de restreindre l'accès à certaines fonctions selon l'interface utilisée pour communiquer avec le canal (car un canal peut hériter de plusieurs interfaces).

Ce dernier point est beaucoup utilisé dans les canaux primaires fournis par SystemC. Ainsi le canal *sc_signal<T>* permet de faire transiter une donnée via les fonction *read()* et *write()* mais ces deux fonctions

ne font pas partie de la même interface. Ainsi, lorsqu'un module ne va faire que produire des données pour ce canal, il communiquera via l'interface *out* de celui-ci tandis que le module consommateur de ces données communiquera avec ce canal par l'interface *in*.

Ce qui nous amène à la question des ports SystemC. La classe de base des ports est la classe *sc_port<T>*. Le type *T* doit référencer une interface de communication. Le port n'est donc qu'une porte d'entrée vers l'interface en question (en réalité, il s'agit même d'un pointeur sur l'objet canal « casté » dans le type d'une de ses interfaces). Vous pouvez donc par la suite utiliser dans votre code la variable *sc_port* comme un pointeur sur l'interface que vous avez fournie.

Il existe quelques ports de base déjà définis et occultant la notion d'interface, mais le principe sous-jacent est le même. Ainsi le port *sc_in<T>* est en réalité un raccourci pour *sc_port<sc_in_if<T>>*.

La dernière question à régler dans le fonctionnement de ces communications est la connexion entre un port et un canal. Cette connexion s'effectue dans la partie de votre programme où vous instanciez vos modules (e.g. le « main »). Une fois vos modules et vos canaux instanciés, vous pouvez connecter un port d'un module à un canal en écrivant ***mon_module.mon_port(mon_canal) ;***

Un exemple complet (interface, canal, module, ports, connexions) vous est présenté dans l'annexe. Des exemples sont aussi présentés en classe et disponibles sur le site web du cours (chap 4).

III. Filtre de Sobel

Le filtre de Sobel est un algorithme couramment utilisé en traitement d'images pour extraire les contours de celle-ci. Il s'agit donc d'un module (IP) que l'on retrouve dans un très grand nombre de systèmes embarqués de traitement d'images. Il fonctionne en approximant la dérivée de l'image (les contours étant nécessairement aux endroits où cette dérivée est plus grande) à l'aide de deux tables pré-calculées de coefficients données au tableau 1.

-1	-2	-1
0	0	0
1	2	1

-1	0	1
-2	0	2
-1	0	1

Tableau 1 : Masques de coefficients de l'algorithme de Sobel

L'algorithme consiste donc à attribuer une valeur, pour chaque pixel de l'image, égale à la somme de la valeur des pixels avoisinants multipliés par le coefficient correspondant, ce qui permet de passer d'une image comme celle de la figure 11 à une détection de contours comme celle de la figure 12.

Les étudiants intéressés à avoir plus de détails sur l'algorithme ou le traitement d'image en général sont invités à consulter la page Wikipédia dédiée à celui-ci et/ou à suivre le cours INF4725 : Traitement de signaux et d'images, ce laboratoire ayant plutôt pour but d'offrir une introduction à *SystemC* par le biais d'un module exécutant cet algorithme.

Figure 1 Image originale

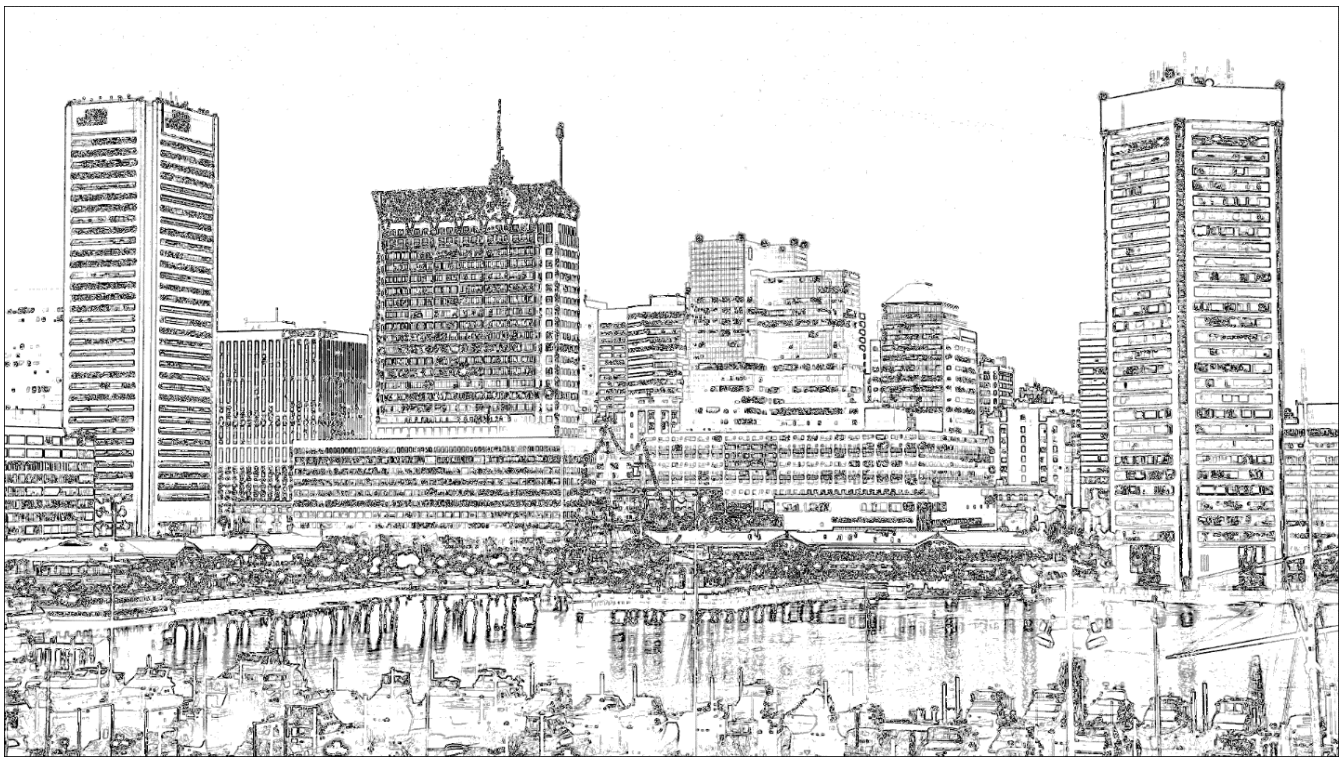


Figure 2 Résultat de l'application du filtre de Sobel sur l'image précédente

IV. Travail à réaliser (VOIR EXEMPLES DANS L'ANNEXE 1)

L'objectif de ce laboratoire consiste à implémenter un modèle SystemC qui exécute un filtre de Sobel sur une image noire et blanc à deux niveaux d'abstraction différents. Ce modèle haut niveau pourrait ensuite être implémenté en logiciel (sur un processeur embarqué) ou encore sur du matériel (FPGA ou ASIC).

Le circuit est composé de 4 modules (détaillés à la section suivante) :

Nom du module	**Fichiers correspondants**
Mémoire de données	DataRAM.h et DataRAM.cpp
Lecteur	Reader.h et Reader.cpp
Module du filtre Sobel	Sobel.h et Sobel.cpp
Écrivain	Writer.h et Writer.cpp (que vous devez créer)
Mémoire « cache » (partie 2.2 seulement)	CacheMem.h et CacheMem.cpp (partie 2.2 seulement)

Le fichier main.cpp s'occupe de faire l'initialisation et de la connexion des modules.

Attention, vous devez utiliser que des sc_thread dans votre code (aussi bien en UTF qu'en AT)

Deux versions du code de départ (pour chaque niveau d'abstraction) sont fournies. Le code dans le répertoire UTF doit être implémenté dans le niveau d'abstraction UTF. Le code dans le répertoire AT-CA doit être implémenté dans le niveau d'abstraction AT (pour le filtre, via un *wait*) et CA (pour les communications, avec le protocole d'*handshaking*).

Les fichiers .sln vous permettent d'ouvrir les projets Visual Studio correspondants. Il faudra rajouter dans les paramètres du projet l'include de votre src SystemC ainsi que la librairie systemC.lib dans les options du linker (**suivre la procédure donnée dans le fichier Création d'un projet SystemC.pdf**). Notez que le code fourni a des problèmes gênants d'erreur de segmentation lorsque compilé en 64 bits : assurez-vous donc de le compiler en 32 bits. Une description plus détaillée de l'architecture à réaliser pour chaque niveau d'abstraction est présentée à la section suivante.

Dans les deux projets, le contenu de la mémoire est enregistré sur le disque à la fin de la simulation, dans un fichier binaire de même nom que votre objet *DataRAM*. Si vous voulez voir l'image résultante, celle-ci peut être convertie en bitmap à l'aide de l'utilitaire MemToBMP fourni[1]. De plus, si vous voulez être certain d'avoir le bon résultat, vous devriez vous attendre à une somme sha1 du fichier contenant la mémoire de

[1] Notez que ce programme s'attend à un fichier nommé « DataRAM » en entrée. Si vous voulez lui passer un autre nom, vous être libres de modifier la source de ce programme.

2d381a9475a1... Si vous voulez tester votre système avec d'autres images, vous pouvez modifier le fichier « image.mem », qui est généré depuis un bitmap via l'utilitaire BMPToMem fourni.

Une base de code est fournie (sauf pour l'écrivain). Vous ne devrez ajouter du code qu'aux endroits mentionnant : « /* à compléter */ » (il se peut qu'il n'y ait rien à ajouter dans certains de ces endroits).

Finalement, **assurez-vous d'appeler la fonction sc_stop()** à la fin de l'exécution de votre thread Sobel pour arrêter la simulation après un filtre. Comme sc_stop arrête la simulation à la fin du cycle courant, et qu'il n'y a qu'un seul cycle en UTF, vous voudrez aussi ajouter un wait() après en UTF.

V. Description des modules

1. Description des modules UTF

Ce premier projet fera le traitement d'une image avec le filtre de Sobel comme expliqué ci-dessous. Le fichier main.cpp vous est fourni et effectue la connexion entre les différents modules demandés. **Il ne vous est nécessaire que d'ajouter les connexions pour le module de l'écrivain que vous devez créer.**

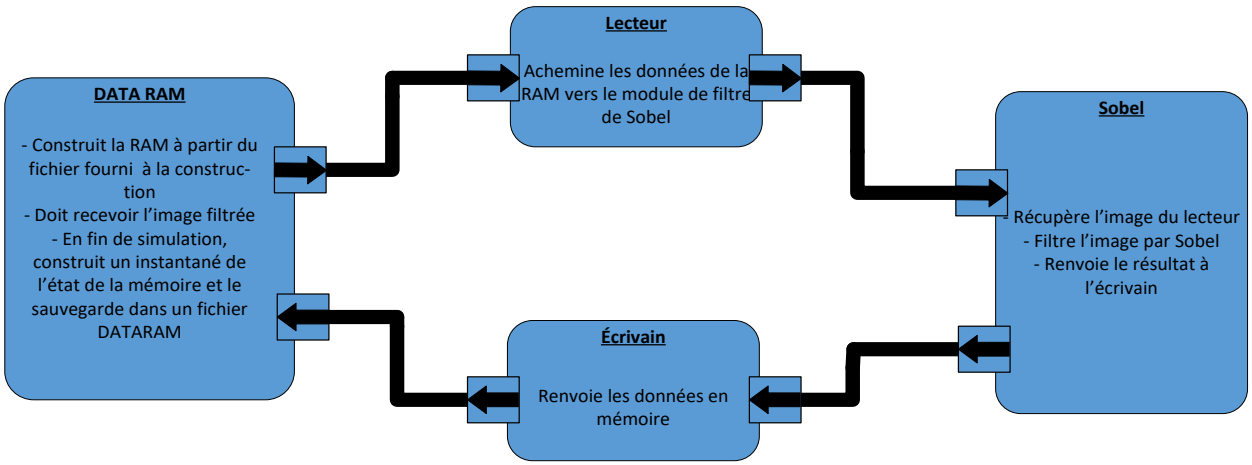


Figure 3 Schéma filtre de Sobel au niveau *Untimed-functionnal*

Lecteur (À compléter)

Type	Nom	Description
sc_port<LMBIF>	dataPortRAM	Port pour la mémoire de donnée

Le module lecteur sert à interfacer le module de Sobel à la mémoire de données. Il agit comme un *wrapper* (adaptateur[2]) de la mémoire. La communication entre ce module et la mémoire est possible à l'aide du

[2] Il permet de convertir l'interface d'un module (classe) en une autre interface que l'autre module attend.

port *dataPortRAM*. Ce port supporte les opérations décrites dans l'interface LMBIF. Finalement, le module lecteur implémente les méthodes de l'interface *InterfaceRead*. Ceci permettra au module Sobel d'envoyer au module lecteur les requêtes de lecture de la mémoire.

- Fonctionnement interne :
  - Lire la mémoire à l'adresse demandée
  - Renvoyer la donnée lue au module de Sobel

Écrivain (À créer)

Type	Nom	Description

Ce module doit être entièrement construit par votre groupe. Vous devez vous inspirer du fonctionnement de son homologue Lecteur. Il doit hériter de l'interface *InterfaceWrite* (que vous devez également créer).

- Fonctionnement interne :
  - Écrire la valeur en mémoire à l'adresse demandée

Filtre de Sobel (À compléter)

Type	Nom	Description
sc_port<InterfaceRead>	readPort	Port pour le module lecteur
sc_port<InterfaceWrite>	writePort	Port pour le module écrivain

Le module de filtre de Sobel doit lire les valeurs d'une image de taille multiple de 4 qui sont sauvegardées dans la mémoire de données et doit ensuite filtrer celle-ci, puis réécrire l'image filtrée à la même adresse. La mémoire de données est initialisée avec le fichier « image.mem ». Ce fichier contient les données qui vont être chargées dans la mémoire de données. Plus précisément, **le premier entier (32 bits) indique la largeur (en pixels 8 bits) de l'image et le deuxième sa hauteur. Les données subséquentes sont les pixels de l'image**. Voici un exemple potentiel du contenu du fichier 'image.mem' pour une image de 4x4 pixels uniformément gris.

04000000 04000000 7F7F7F7F 7F7F7F7F 7F7F7F7F 7F7F7F7F

Notes sur le stockage des valeurs en mémoire :

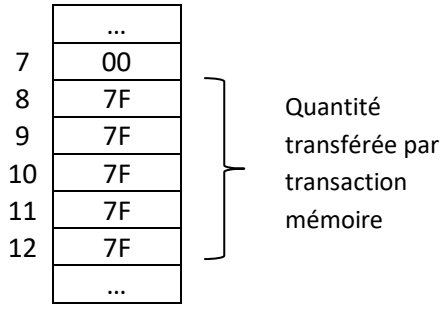
-Vous noterez que les valeurs sont stockées de manière little-endian (le premier octet en mémoire est l'octet de poids le plus faible. Ainsi la première valeur ci-dessus 04 00 00 00 est en réalité le chiffre 4 -> 0x00000004).

-Le système d'adressage utilisé tout au long de ce laboratoire numérote les octets en mémoire. Ainsi le premier mot de 32 bits se situe à l'adresse 0 et le second mot se trouve à l'adresse 4

-Bien que **les pixels soient sur 8 bits**, les transferts de données depuis et vers la mémoire doivent se faire sur des **longueurs de 32 bits** (taille d'un unsigned int). Vous transférez donc 4 pixels par transaction

mémoire. Pour les détails d'implémentation, voir fichier DataRAM.cpp et les arguments de la fonction memcpy().

Ainsi, une représentation en mémoire des valeurs présentées ci-haut serait :



- Fonctionnement interne :
  - Le module va tout d'abord lire la taille de l'image
  - Il va ensuite lire et stocker l'image, et s'allouer de l'espace pour le résultat
  - Puis, le module va **assigner la valeur 0 aux contours de l'image résultante résultat** (première et dernière rangées et colonnes), ce contour n'étant pas passé au filtre.
  - Pour chaque autre pixel de l'image, le module appellera la fonction *sobel_operator* fournie et stockera le résultat
  - Le résultat est ensuite envoyé à l'écrivain de manière à remplacer les données l'image originale par l'image filtrée (c.-à-d. qu'on écrit à la même adresse que la première image (soit après sa taille), et non à la suite de celle-ci).

2. Description des modules AT/Cycle Accurate

Il vous est demandé d'implémenter deux versions du projet à niveau AT/CA. Tout d'abord, le même principe de base que celui du niveau UTF sera utilisé pour la première version, mais en introduisant une granularité aux niveaux des communications par le handshaking et du traitement du temps de traitement par l'utilisation d'un wait(). Puis, une deuxième version utilisant une structure en ligne à délai (raster scan line) vous sera demandé.

Pour les deux versions, une partie du main sera fournie. Ce main doit pouvoir passer d'une version à l'autre en changeant seulement un booléen (la variable utiliseCacheMem).

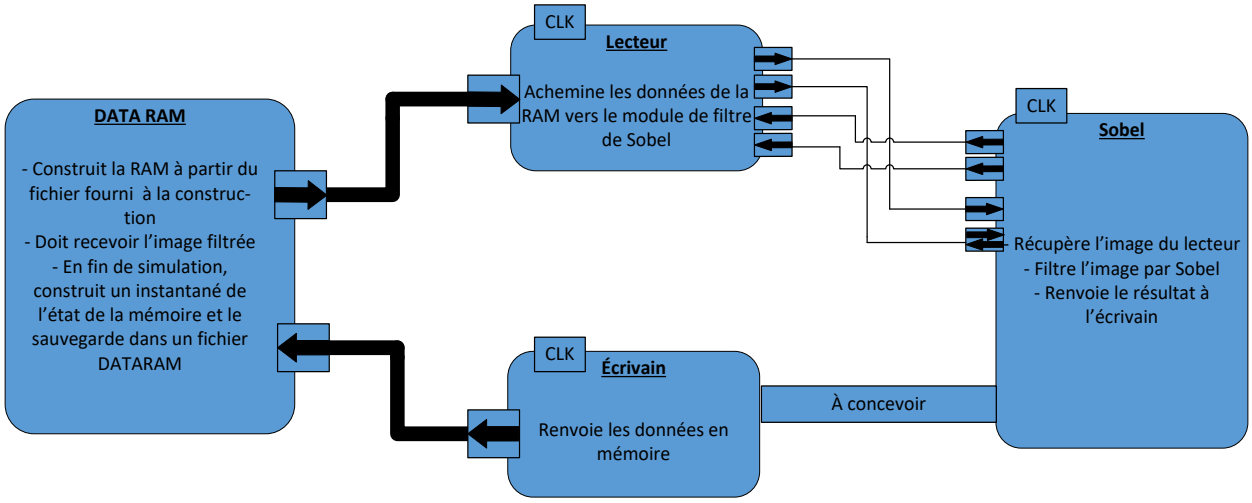


Figure 4 Schéma filtre de Sobel à un niveau Approximate Timed/Cycle Accurate

2.1 Description des modules AT/CA de base

Lecteur (Fourni)

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 de la donnée à lire
sc_out<bool>	ack	Accusé de réception
sc_in<bool>	request	Requête

Le module lecteur sert toujours à interfacer le module de Sobel à la mémoire de données. Toutefois, maintenant la communication entre ce module et le module de communication 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 type *handshaking* (Voir exemple annexe 1). Ce module vous est fourni, mais vous devez comprendre son fonctionnement afin de pouvoir l'utiliser correctement depuis le module Sobel.

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

Écrivain (À créer)

Type	Nom	Description

Comme pour l'UTF, et similairement au lecteur, le module de l'écrivain sert toujours à interfacer le module de Sobel à la mémoire de données, mais de façon plus raffinée par une communication de type handshaking. Inspirez-vous du module du lecteur et de l'exemple en annexe pour l'implémentation.

- Fonctionnement interne :
  - Attendre une requête
  - Lire la valeur de l'adresse
  - Lire la donnée
  - Écrire la donnée en mémoire à l'adresse reçue
  - Envoyer un accusé de réception
  - Enlever l'accusé de réception

Filtre de Sobel (À compléter)

Type	Nom	Description
sc_in_clk	clk	Horloge
sc_out<unsigned int>	address	Adresse à envoyer au lecteur ou à l'écrivain
sc_inout<unsigned int>	data	Donnée à envoyer au lecteur ou à recevoir de l'écrivain (inout)
sc_out<bool>	requestRead	Requête au lecteur
sc_out<bool>	requestWrite	Requête à l'écrivain
sc_in<bool>	ack	Accusé de réception

Encore une fois le module de filtre de Sobel doit lire l'image sauvegardée dans la mémoire de données et doit ensuite la filtrer, puis renvoyer le résultat dans la mémoire. Toutefois, la synchronisation avec le module lecteur se fait par un protocole simple de *handshaking*.
Pour ce qui est du filtre, vous devez modéliser approximativement[3] le temps pris par l'algorithme à l'aide d'**un** appel à la fonction *wait()* de SystemC contenant les bons paramètres (assurez-vous que cette attente est fonction d'un nombre de cycles de l'horloge (clk) et non une unité de temps arbitraire). Ceci pourrait nécessiter un ajustement à la liste de sensitivité du module.

Une fois le filtre complété, le module doit renvoyer l'image résultante en mémoire en les envoyant au module Ecrivain. Vous devez compléter ports, signaux et protocole afin de rendre cette communication possible. Notez que les ports de données et d'*acknowledge* du module sont accédés en écriture par plusieurs modules (le lecteur et l'écrivain), ce qui n'est pas supporté par SystemC par défaut : vous devez spécifier que vous voulez bien ce comportement et que ce n'est pas une erreur en ajoutant *SC_MANY_WRITERS* à la création du signal les connectant (voir l'annexe pour plus de détails). Bien sûr, un tel mode à plusieurs écrivains ne fonctionnera que si le signal n'est accédé en écriture que par un module (au maximum) à chaque cycle de votre simulation.

- Fonctionnement interne :
  - Envoyer l'adresse à être lue
  - Envoyer une requête
  - Attendre un accusé de réception
  - Lire la donnée reçue
  - Enlever la requête
  - Filtrer
  - Protocole avec écrivain

2.2 Description des modules AT/CA avec structure en ligne à délai

Les modules du lecteur et de l'écrivain son identiques à la partie 2.1. Vous pouvez utiliser la même solution qu'à la section précédente, en faisant les modifications suivantes :

- Modifier le booléen utiliseCacheMem dans le main
- Ajouter les fichiers CacheMem.h/.cpp

[3] Cette approximation devrait être fonction de la taille des données à traiter, de la complexité de l'algorithme et de comment vous visualisez l'implémentation de cet algorithme en matériel. Il serait pertinent de justifier le délai que vous avez choisi en commentaire dans votre code ou dans votre rapport.

Mémoire Cache (fourni)

Type	Nom	Description
sc_in_clk	clk	Horloge
sc_in<unsigned int*>	addressData	Adresse de début du tableau où les données doivent être placées
sc_in<unsigned int>	length	Longueur des données à lire
sc_in<bool>	requestFromCPU	Requête provenant du processeur
sc_out<bool>	ackToCPU	Accusé de réception au processeur
sc_out<unsigned int>	address	Adresse de la donnée à lire
sc_in<unsigned int>	dataReader	Donnée lue provenant du lecteur
sc_out<bool>	requestFromCPU	Requête au lecteur
sc_in<bool>	ackFromReader	Accusé de réception du lecteur

Le module de la mémoire cache s'occupe de faire les requêtes au lecteur afin de lire et stocker plusieurs données. L'espace mémoire allouée pour stocker ces données est allouée par le module de Sobel (voir ci-dessous). La mémoire cache a donc comme tâche de lire une à une les valeurs de la mémoire via le lecteur, et les stocker au bon endroit en mémoire.

- Fonctionnement interne :
  - Attendre une requête
  - Lire l'adresse à lire
  - Lire la longueur des données à lire
  - Lire l'adresse où stocker les données
  - Faire les requêtes nécessaires au lecteur pour lire une à une les données, et les stocker au bon endroit en mémoire, avec une communication de type handshaking
  - Envoyer un accusé de réception
  - Enlever l'accusé de réception

Sobel (à modifier à partir de 2.1)

Les ports clk, dataRW, address, requestRead, requestWrite et ackReaderWriter correspondent aux ports décrits dans la partie 2.1. Les ports additionnels sont les suivants :

Type	Nom	Description
sc_in<unsigned int*>	addressRes	Adresse de début du tableau où les données doivent être placées
sc_in<unsigned int>	length	Longueur des données à lire de la cache
sc_in<bool>	requestCache	Requête à la mémoire cache
sc_out<bool>	ackCache	Accusé de réception provenant de la mémoire cache

Le fonctionnement du module Sobel est très similaire à la partie 2.1, à une exception près. Il vous est donc fortement conseillé de reprendre votre code de la partie 2.1.

Ici, il vous est demandé de n'utiliser qu'un tampon d'une taille correspondant à 4 lignes de votre image (par exemple, 4 * 1920 pixels pour une image de 1080 x 1920 pixels) pour stocker les données de l'image à traiter. Le processeur va donc alterner entre la lecture de quelques lignes, et le traitement par filtre de Sobel sur ces lignes. Pour simplifier les choses, nous garderons l'image traitée en entier en mémoire, avant de l'envoyer une fois tout le traitement terminé. Notons qu'en pratique il ne suffirait que de connecter le Reader et le Writer sur des ports différents pour pouvoir renvoyer les pixels traités immédiatement.

Puisque le filtre de Sobel nécessite les 8 pixels envoisinant le pixel à traiter afin d'effectuer le traitement, nous avons en réalité besoin que de 3 lignes de données à la fois. L'annexe 2 explique en détails le fonctionnement de la structure à délai illustré sur une image de 8 pixels de largeur.

- Fonctionnement :

1. Lire les informations de l'image (largeur et hauteur) directement du lecteur (idem à 2.1)
2. Assignation de 0 aux contours
3. Lecture des trois premières lignes de l'image (latence) via la mémoire cache avec une communication de type handshaking
4. Traitement de la ligne reçue avec Sobel (sans les contours)
5. Pendant ce temps, la mémoire cache va lire la prochaine ligne en mémoire RAM
6. Répétition de 4 et 5 pour toutes les lignes
7. Écriture dans la mémoire RAM de l'image traitée avec l'écrivain directement

VI. Questions

1- Quel est le principal avantage de faire une version UTF du système avant d'aller plus loin dans le processus de raffinement (c'est-à-dire avant d'aller vers du AT/CA ou même RTL).
2- Sachant que dans la suite du laboratoire 3 (c.-à-d., le laboratoire 4) vous allez raffiner davantage le système pour l'amener vers une implémentation RTL qui pourra être placée et routée sur le FPGA, quel est l'avantage du modèle AT/CA que vous avez conçu (section V.2) par rapport au modèle UTF (section V.1) ?
3- Aurait-on pu transférer 3 lignes (ou moins) à la fois plutôt que 4 lignes dans le modèle AT/CA ? Expliquez (hint : ceci pourrait être un argument de plus pour votre réponse no 2…).
4- Expliquez comment SystemC gère la concurrence de processus, par rapport à µC. Pour vous aider, étudiez le comportement de la fonction *wait*().

VII. Barème et rendu

A l'issue de ce laboratoire vous devrez remettre sur Moodle, une fois par groupe de 2, une archive respectant la convention INF3610Lab3_matricule1_matricule2.zip contenant :

- Dans un dossier src, le code de vos fichiers modifiés en UTF et en AT-CA dans deux dossiers séparés
- A la racine, un bref rapport contenant les réponses aux questions du laboratoire

Vous devez rendre ce laboratoire au plus tard le 2 (B1) ou le 9 (B2) novembre à minuit (soit 2 semaines après le laboratoire).

Barème	
Exécution du code	
UTF	/6
AT-CA	/8
Réponse aux questions	
Question 1	/1
Question 2	/2
Question 3	/1
Question 4	/2
Avis sur le laboratoire	/0
Respect des consignes	
Entraine des points négatifs (peut aussi invalider les points d'un exercice)	
TOTAL	/20

Annexe 1 : Exemples

```
template <class T>
class MyChannel_IF : sc_interface
{
    public:
        // Notez les "=0" signifiant la virtualité pure des méthodes de cette classe
        virtual T read() = 0;
        virtual void write (T value) = 0;
};

template <class T>
class MyChannel : sc_channel, MyChannel_IF<T>
{
    public:
        //Cette classe va devoir implémenter les méthodes read et write
        virtual T read();
        virtual void write (T value);
}
```

Code Snippet 1 : interface et canal

```
class Station : sc_module
{
    /* ********************************************************************
    // MODULE PORTS
    ******************************************************************** */
    sc_in<int> int_port_1;
    sc_inout<int> int_port_2;
    sc_out<bool> bool_port;
    // Notez que le sc_port demande une sc_interface en paramètre de template
    sc_port<MyChannel_IF<int>> channel_port;


    /* ********************************************************************
    // LOCAL VARIABLES
    ******************************************************************** */
    int var;

    /* ********************************************************************
    // MODULE METHODS
    ******************************************************************** */
    void thread(); //THREAD
    void method_1(); // METHOD
    int calculate_CRC(); // Normal class method

    /* ********************************************************************
    // MODULE CONSTRUCTOR
    ******************************************************************** */
    Station()
    {
        //On définit thread() comme étant un sc_thread sensible au port port3
        SC_THREAD(thread);
        sensitive << port3;
        //On définit method_1() comme étant une sc_method sensible au port2
        SC_METHOD(method_1);
        sensitive << port2;
    }

    private:
        SC_HAS_PROCESS(station); // Permet la création de threads et de méthodes
};
```

Code Snippet 2 : Module header

```
int main()
{
    // MODULE INSTANCIATION
    Station station1();

    // CHANNEL INSTANTIATION
    MyChannel<int> channel_1;
    sc_signal<int> signal_1;
    sc_buffer<bool> buffer_1;

    //CONNECTIONS
    station1.int_port_1(signal_1);
    station1.int_port_2(signal_2);
    station1.bool_port(buffer_1);
    station1.channel_port(channel_1);

    //Reste à connecter ces canaux sur un autre module pour permettre la
    //communication entre station1 et un autre module

    // Puis démarrer la simulation
}
```

Code Snippet 3 : Main et connexions

```
// Variable
sc_signal<bool> sEnable;

// On effectue le branchement
Instance_cd.enable(sEnable);
Instance_auto.enable(sEnable);
```

Exemple de branchement

```
// Variable
sc_signal<unsigned int, SC_MANY_WRITERS> sData;

// On effectue le branchement
Instance_cd.data(sData);
Instance_auto.data(sData);
Instance_radio.data(sData);
```

Exemple de branchement à plusieurs écrivains

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 : comment 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

…
// Envoi de l'adresse
address.write(addr);
enable.write(true);
 	
// Synchronisation
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'addresse
	addr = address.read();
	
	// Syncronisation
	ack.write(true);
	
	…

Exemple de synchronisation (*handshaking*)

Annexe 2 : Détails de la structure de ligne à délai

P1,1	P1,2	P1,3	P1,4	P1,5	P1,6	P1,7	P1,8
P2,1	**P2,2**	P2,3	P2,4	P2,5	P2,6	P2,7	P2,8
P3,1	P3,2	P3,3	P3,4	P3,5	P3,6	P3,7	P3,8

On débute à P2,2 après transfert 3 lignes (latence). Pendant le traitement sur coprocesseur on transfert la 4e ligne.

P2,1	P2,2	P2,3	P2,4	P2,5	P2,6	**P2,7**	P2,8
P3,1	P3,2	P3,3	P3,4	P3,5	P3,6	P3,7	P3,8
P4,1	P4,2	P4,3	P4,4	P4,5	P4,6	P4,7	P4,8

Après P2,7 on peut flusher la ligne 1

P2,1	P2,2	P2,3	P2,4	P2,5	P2,6	P2,7	P2,8
P3,1	**P3,2**	P3,3	P3,4	P3,5	P3,6	P3,7	P3,8
P4,1	P4,2	P4,3	P4,4	P4,5	P4,6	P4,7	P4,8

On poursuit à P3,2. Pendant le traitement sur coprocesseur on transfert la 5e ligne.

P5,1	P5,2	P5,3	P5,4	P5,5	P5,6	P5,7	P5,8
P3,1	P3,2	P3,3	P3,4	P3,5	P3,6	**P3,7**	P3,8
P4,1	P4,2	P4,3	P4,4	P4,5	P4,6	P4,7	P4,8

Après P3,7 on peut flusher la ligne 2

P5,1	P5,2	P5,3	P5,4	P5,5	P5,6	P5,7	P5,8
P3,1	P3,2	P3,3	P3,4	P3,5	P3,6	P3,7	P3,8
P4,1	**P4,2**	P4,3	P4,4	P4,5	P4,6	P4,7	P4,8

On poursuit à P4,2. Pendant le traitement sur coprocesseur on transfert la 6e ligne.

P5,1	P5,2	P5,3	P5,4	P5,5	P5,6	P5,7	P5,8
P6,1	P6,2	P6,3	P6,4	P6,5	P6,6	P6,7	P6,8
P4,1	P4,2	P4,3	P4,4	P4,5	P4,6	**P4,7**	P4,8

Après P4,7 on peut flusher la ligne 3

Etc.