ÉCOLE POLYTECHNIQUE DE MONTRÉAL

INF3610 : Laboratoire 3

Introduction à SystemC

Arnaud Desaulty – Frédéric Fortier

10/23/2015

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 SoC avec une méthodologie de conception haut niveau
- Connaître les différents niveaux d'abstraction
- 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 un niveau d'abstraction plus bas (*Approximate Timed*).

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. mauvais 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, le 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 microC, **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 doivent 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  }
```

Figure 1 : Code Snippet 1 : interface et canal

À la Figure 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 class *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 quelque 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 ou vous instanciez vos modules. 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.

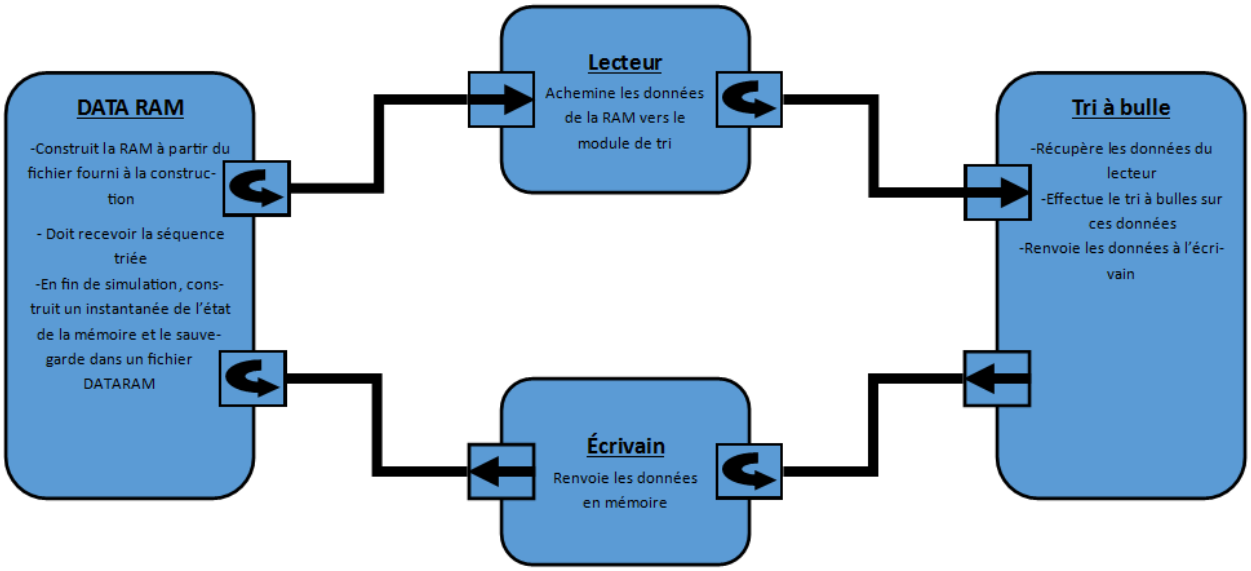
III. Conception du système

L'objectif de ce laboratoire consiste à implémenter un modèle SystemC qui exécute l'algorithme de tri à bulle dans un ensemble de valeurs. Ce modèle haut niveau pourrait ensuite être implémenté en logiciel (sur un processeur embarqué) ou encore sur du matériel (FPGA).

Le circuit est composé de 4 modules :

- Une mémoire de donnée
- Un lecteur
- Module tri à bulle
- Un écrivain

1. Description des modules UTF



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 la mémoire. La communication entre ce module et la mémoire est possible à l'aide du port *dataPortRAM*. Ce port supporte les opérations décrites dans l'interface LMBIF. Finalement, le module lecteur doit implémenter les méthodes de l'interface *interfaceRead*. Ceci permettra au module tri à bulles 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 tri à bulle

Ecrivain

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

Tri à bulle

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

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. La mémoire de donnée est initialisée avec le fichier « chiffre.hex ». Ce fichier contient les données qui vont être chargées dans la mémoire de données. Plus précisément, **le premier nombre indique le nombre d'éléments** à trier tandis que **les nombres subséquents sont les éléments à trier**. Voici un exemple potentiel du contenu du fichier 'chiffre.hex' :

08000000 04000000 06000000 03000000 02000000 01000000 01000000 0A000000 00000000

-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 08 00 00 00 est en réalité le chiffre 8 -> 0b00000008).

-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

- Fonctionnement interne :
  - Le module va tout d'abord lire le nombre de données à trier.
  - Il va ensuite lire et stocker ces valeurs
  - Puis, le module va appeler la fonction de tri à bulle sur ces valeurs (ceci vous est fourni)
  - Après avoir lu tous les nombres, les valeurs doivent être triées.
  - Le résultat est affiché
  - Le résultat est ensuite envoyé à l'écrivain

2. Description des modules AT

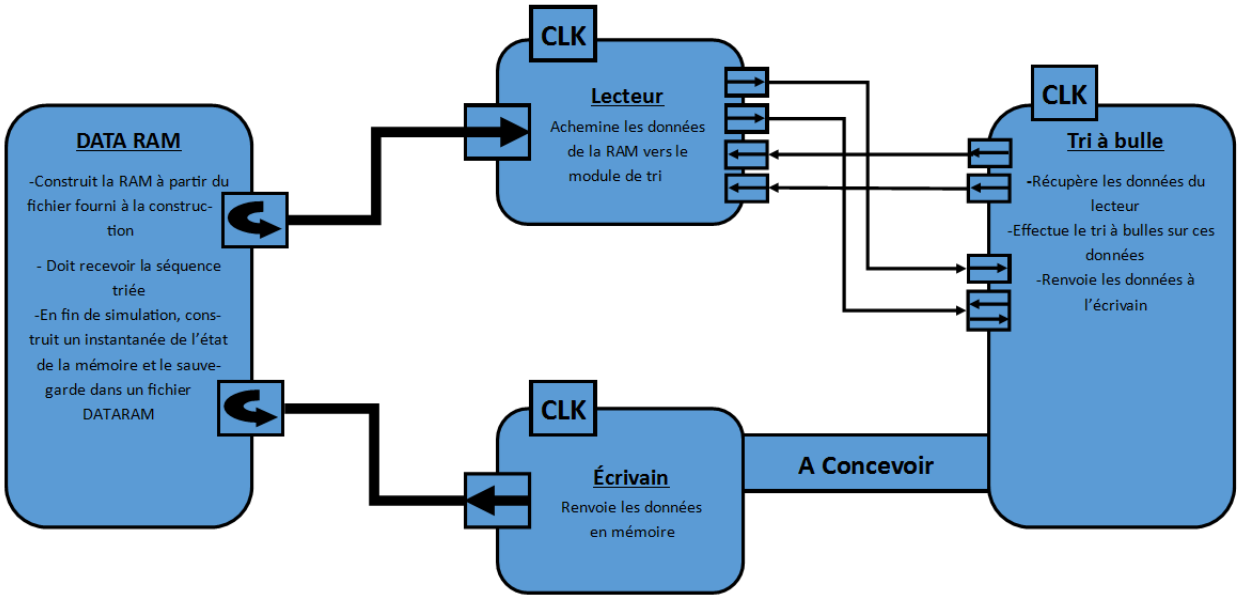


Figure 1 Schéma circuit tri à bulle au niveau Approximate Timed

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 type *handshaking*. (Voir exemple 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

Écrivain

Type	Nom	Description

Comme pour l'UTF, il vous est demandé d'implémenter ce module par vous-même. Aidez-vous de **la description du fonctionnement interne, des exemples et du lecteur.**

➢ 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

Tri à bulle

Type	Nom	Description
sc_in_clk	clk	Horloge
sc_out<unsigned int>	address	Adresse
sc_inout<unsigned int>	data	Donnée
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 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. Toutefois, la synchronisation avec le module lecteur se fait par un protocole simple de *handshaking*.

Une fois le tri effectué le module tri doit renvoyer les valeurs 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
  - Trier
  - Protocole avec écrivain

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

Vous devez compléter les fichiers :

- Reader.h et Reader.cpp
- Bubble.h et Bubble.cpp
- Main.cpp
- Vous devez aussi créer les fichiers Writer.h et Writer.cpp
- **Attention, vous devez utiliser que des sc_thread dans votre code (aussi bien en UTF qu'en AT)**

Le code dans le répertoire code/UT doit être implémenté dans le niveau d'abstraction UTF. Le code dans le répertoire code/AT doit être implémenté dans le niveau d'abstraction AT. Si vous voulez tester votre système avec d'autres valeurs, vous pouvez modifier le fichier « chiffre.hex »

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**)

Questions

1- Quel est l'intérêt d'utiliser les modules writer et reader au lieu d'interfacer directement le module bubble au module dataRam ?

2- Aurait-il été possible d'implémenter la communication handshake du channel Reader dans le modèle AT à l'aide d'une *sc_method* plutôt que d'un sc_thread ? Expliquez votre réponse.

V. 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 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 la veille du prochain laboratoire à minuit (soit 2 semaines après le premier laboratoire)

Barème	
Exécution du code	
UTF	/6
AT	/8
Réponse aux questions	
Question 1	/3
Question 2	/3
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 2 : 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 3 : 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 4 : 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*)