Temps réel – **TP1**

Initiation à SystemC

Partie A – Analyse d'un exemple

Dans le répertoire *TP1_A* il y a l'exemple simple de producteur/consommateur fourni par SystemC.

Exercices :

A1. Compilez le programme et exécutez-le.

A2. Compréhension du code donné. Identifiez les éléments SystemC utilisés pour modéliser le système (modules, canaux, ports, interfaces, signaux, événements, processus). Expliquez le rôle et le fonctionnement de tous ces éléments.

A3. Dans l'affichage à l'écran du consommateur ajoutez le temps de simulation (fonction *sc_time_stamp()*). Ajoutez des *wait*s dans les fils principaux des deux modules.
Par exemple : un *wait(20, SC_NS)* dans le consommateur. Recompiler et re-exécutez.
Ensuite, un *wait(40, SC_NS)* dans le producteur. Recompiler et re-exécutez.
Qu'est-ce que vous observez ? Expliquez ce comportement.

Partie B – Modélisation d'un système simple

On souhaite modéliser le système présenté dans la figure 1. Un transfert des données par communication sérielle au niveau bit est réalisé entre les modules *BitGen* et *Bit2Byte*. Ensuite, le destinataire (*Bit2Byte*) doit composer des octets des données (à partir des bits reçus) et les envoyer à une entité qui les traite (*Consumer*).

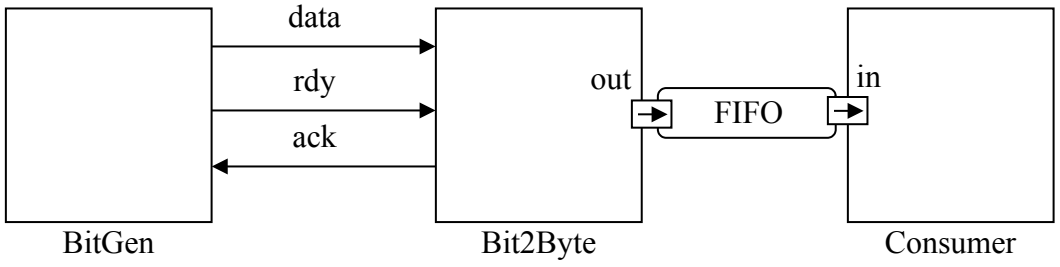

Figure 1. Traitement des données à distance

Le système est composé de 3 modules qui communiquent entre eux :

- *BitGen*, qui décompose en bits les données à envoyer vers le module *Bit2Byte* ;
- *Bit2Byte*, qui recompose les octets à partir des bits reçus et les envoie au module *Consumer* ;
- *Consumer*, qui reçoit les octets de données du module *Bit2Byte* et fait le traitement.

Les deux sous-systèmes de communication entre les modules sont modélisés différemment.

Le premier, qui implémente la communication entre *BitGen* et *Bit2Byte* est réalisé au niveau fils et utilise un protocole poignée de mains (handshaking) – voir la figure 2. Le fil *data* est utilisé pour les données, tandis que *rdy* (ready - prêt) et *ack* (acknowledge – confirmation) sont les signaux de contrôle. *rdy* indique le fait que *BitGen* à mis une donnée et *ack* signalise le fait que *Bit2Byte* a lu la donnée.

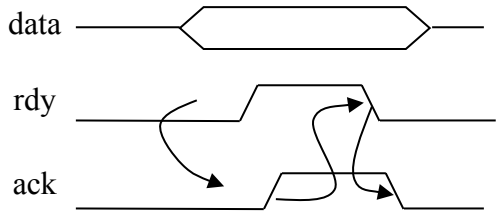

Figure 2. Protocole handshaking

La communication entre *Bit2Byte* et *Consumer* est implémentée à un niveau plus abstrait, en utilisant un canal SystemC. Le canal implémente deux interfaces, une pour l'écriture et l'autre pour la lecture, *write_if* et *read_if*, respectivement.

Exercices :

B1. Compréhension du code donné. Expliquer le rôle et le fonctionnement de tous les éléments.

B2. Ajoutez les ports nécessaires au module *BitGen* et effectuez le lien de ce module avec le reste du système.

B3. Implémentez le thread *thProduce* du module *BitGen* de manière qu'il envoie des bits par ses ports en respectant le protocole de la figure 2.

Les données sont à lire du fichier *a.txt* déjà existant dans le répertoire de travail. Il faut lire les données octet par octet. Chaque octet doit être découpé en bits (en utilisant soit des opérations par bit, soit des méthodes des types SystemC). Chaque bit doit être envoyé par le protocole de la figure 2.

B4. Implémentez la méthode *read* de la FIFO.

B5. Implémentez le thread *thConsume* du module *Consumer*.

B6. Testez le fonctionnement de l'ensemble du système. Visualiser les chronogrammes des signaux et expliquez-les. Tracez d'autres signaux ou variables afin de pouvoir visualiser le moment de l'envoie d'un octet de *Bit2Byte* et la réception d'un octet par *Consumer*. Modifier la fréquence de fonctionnement des différents éléments du système et observez comment le fonctionnement de l'ensemble du système est influencé.

Temps réel – **TP2**

Modélisation et implémentation d'un système multi-tâches et multi-processeur

Le système que nous voulons modéliser en SystemC est composé de plusieurs processeurs (CPUs, Central Processing Units) sur lesquels tournent des processus (ou tâches). Sur chaque CPU il y a un système d'exploitation (operating system, OS) qui gère les tâches. Un exemple d'un tel système est représenté dans la figure 1.

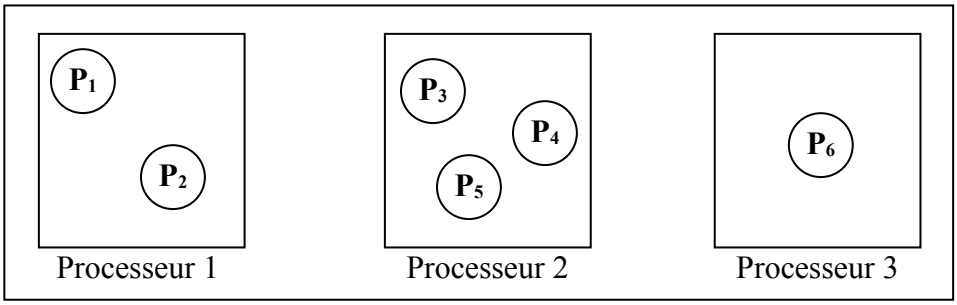

Figure 1. Système multiprocesseur sans communication

Dans cet exemple, il y a les processus P_1 et P_2 qui s'exécutent sur le processeur 1 ; P_3, P_4 et P_5 sur le processeur 2 ; et P_6 sur le processeur 3. Pour l'instant nous considérons qu'il n'y a pas de communication entre les tâches du système.

Dans les TPs suivants nous allons ajouter de la communication entre les tâches :

- à l'intérieur d'un processeur (TP 3) ;
- s'exécutant sur des CPUs différents (TP 4).

Il faut modéliser dans un premier temps (ce TP) le système sans communication, avec la gestion des processus qui s'exécutent sur les processeurs.

Nous modélisons les CPUs par des modules ayant un fil d'exécution (thread SystemC) principal qui implémente le pointeur d'instructions. Ainsi, en utilisant le pseudo-parallélisme implémenté en SystemC par des threads, nous modélisons le fonctionnement parallèle des plusieurs processeurs. Chaque CPU a un lien vers l'OS qui tourne dessus et un identificateur. De plus, pour gérer les processeurs, nous utilisons un vecteur des CPUs, sous la forme d'un attribut statique de la classe.

Le système d'exploitation est implémenté sous la forme d'une classe C++. L'OS doit avoir accès au niveau matériel. Pour cela, il utilise le noyau (ou kernel) que nous avons implémenté. Les principaux attributs de la classe OS sont les listes des tâches qui s'exécutent sur le processeur et qui servent à la gestion des tâches.

Les fonctions principales du système d'exploitation sont :

RegisterTask – enregistre une tâche

TaskYield – cède le processeur

TaskExit – termine une tâche

Schedule – planifie les tâches

Consume – modélise l'exécution des tâches en temps

Les threads SystemC sont concurrents. Comme nous l'avons dit, les processeurs sont modelés par des threads SystemC. Dans notre système, sur chaque processeur s'exécutent plusieurs tâches. Il faut modéliser l'exécution pseudo-parallèle des tâches tournant sur le même processeur. Etant donné que l'exécution des processeurs est modelée par des threads SystemC, pour l'implémentation de changement des tâches à l'intérieur d'un processeur il faut utiliser un mécanisme différent. Sinon, tous les processus qui s'exécutent sur tous les processeurs seraient concurrents entre eux.

Alors, pour implémenter le pseudo-parallélisme pour les tâches d'un processeur, nous utilisons des contextes utilisateur.

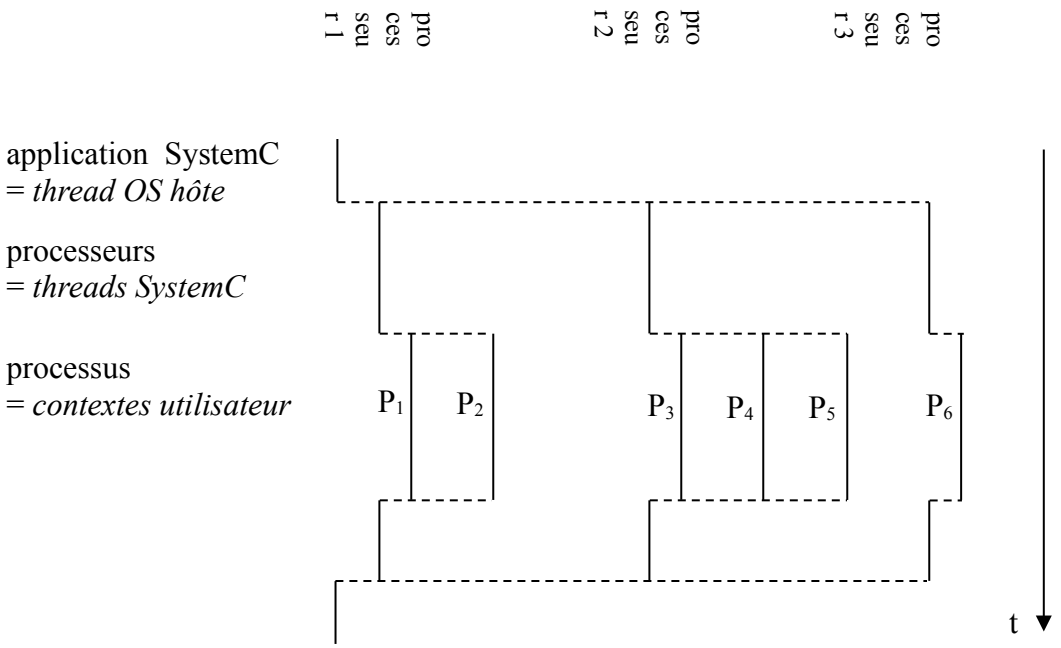

Figure 2. Modélisation du parallélisme pour le système multiprocesseur

Dans la figure 2, nous avons représenté la modélisation en SystemC du parallélisme des processeurs et du pseudo-parallélisme des tâches. Ainsi, les processeurs sont modélisés par des threads SystemC, alors que les processus sont modélisés utilisant des contextes utilisateur.

L'exécution pseudo-parallèle des deux tâches, P_1 et P_2 est représenté dans la figure 3. Pour le changement de contexte est utilisée la fonction SwitchContext de Kernel.

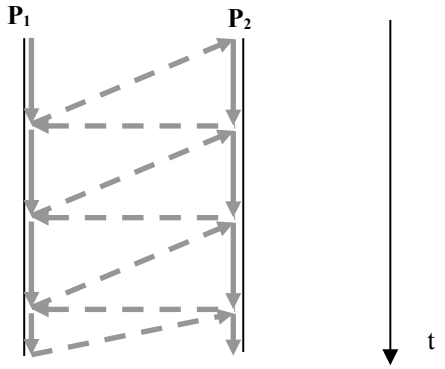

Figure 3. Changement de tâches

Exercices :

1) Analyse et compréhension du code donné. Expliquer le rôle et le fonctionnement de tous les éléments du système (processeur, système d'exploitation, tâche etc.).

2) Etudiez l'ordonnancement des tâches à l'aide de la trace SystemC. Que se passe t-il lorsque la consommation des tâches augmente ? Est-ce un ordonnancement préemptif ?

3) a) Expliquez la création et la destruction d'une tâche, en précisant les étapes et les acteurs impliqués (CPU, OS, Kernel).

   b) Ajoutez une nouvelle tâche appelée *proc4*, s'exécutant sur le processeur *cpu2*, qui reçoit comme arguments 3 paramètres : deux de type *int* et le troisième de type chaîne de caractères. La tâche affiche en boucle infinie la somme des deux paramètres et le troisième paramètre.

   c) Modifier la tâche 1 de telle manière qu'au pas 5 elle crée dynamiquement une nouvelle tâche, *proc5*. La nouvelle tâche est similaire à la *proc4* crée précédemment, mais elle fait l'affichage une seule fois et ensuite se termine.

4) a) Etudiez le rôle du port *interrupt* des modules CPU. Quel est l'intérêt d'un tel port ?

   b) Modélisez et implémentez une interruption de temps de fréquence 1 micro-seconde dans les CPUs de notre système. Pour cela, vous pourrez créer des modules timer SystemC connectés aux CPUs.

   c) Quel est l'impact de cette interruption sur l'ordonnancement des tâches, en particulier dans un contexte temps-réel ?

Temps réel – **TP3**

La communication intra-processeur dans un système multi-tâches et multi-processeurs

Pour ce TP nous utilisons le système multi-tâches et multi-processeurs implémenté pour le TP 2. Il faut implémenter pour ce système la communication entre les tâches qui tournent sur un même processeur (la communication intra-processeur). Le système avec cette nouvelle fonctionnalité est représenté dans la figure 1.

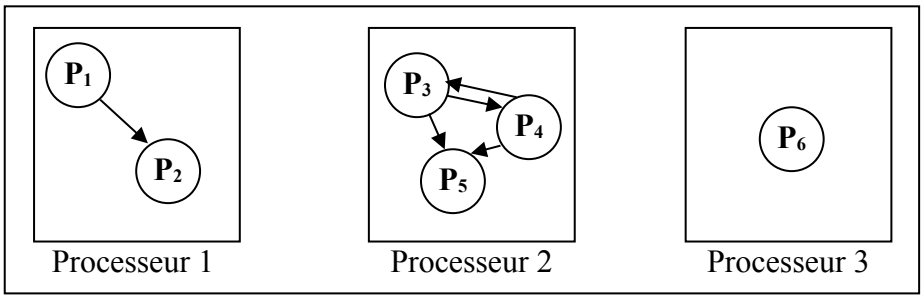

Figure 1. Système multiprocesseur avec communication intra-processeur

Le modèle de coopération entre processus utilise :
- la notion du CANAL pour l'interaction entre processus concurrents
- le passage de MESSAGE comme procédé de communication
- le RENDEZ-VOUS comme mécanisme de synchronisation

Pour utiliser ce modèle, il faut implémenter dans le système : le canal de communication, les messages et les éléments de synchronisation. Comme éléments de synchronisation nous utiliserons les événements fournis par SystemC.

Dans notre système, il faut mettre à la disposition des tâches deux types de communication intra-processeur :
- entre un seul émetteur et un seul récepteur (SESR)
- entre plusieurs émetteurs et un seul récepteur (PESR)

Pour les réaliser, nous implémentons trois primitives dans le système d'exploitation :
- *ChanOut*, qui doit être appelé par un émetteur (aussi bien pour SESR que pour PESR) ;
- *ChanIn*, qui doit être appelé par un récepteur lorsqu'il fait partie d'une communication SESR ;
- *AltIn*, qui doit être appelé par un récepteur lorsqu'il fait partie d'une communication PESR.

Dans la figure 2 sont présentées les différentes étapes pour la communication SESR :
- le processus arrivé le premier au rendez-vous (le canal est libre) – voir figure 2.a – occupe le canal et se bloque (figure 2.b) en attendant l'autre processus qui participe à la communication (il attend l'activation de l'événement associé au canal).
- le processus qui arrive le deuxième au rendez-vous, P_2 dans cet exemple, effectue la communication proprement dite (le passage du message) – voir figure 2.c – et libère

ensuite le canal (figure 2.d). Le processus P_1 sera débloqué par l'activation de l'événement associe au canal (l'événement est activé par P_2).

Observation. Ce protocole est symétrique, quel que soit le processus arrivé en premier, soit l'émetteur, soit le récepteur.

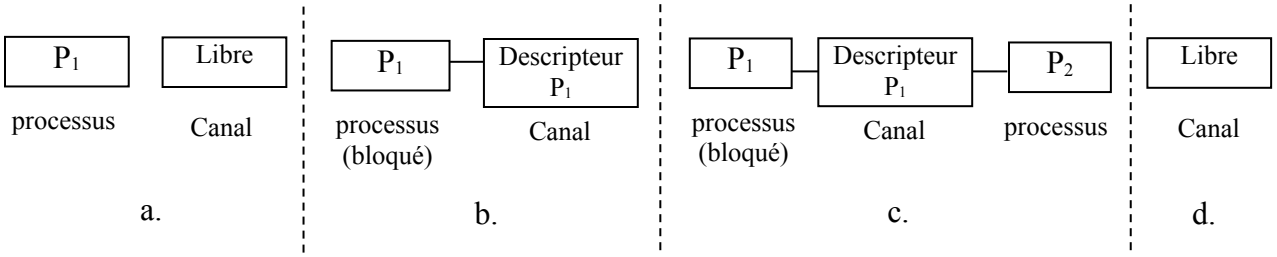

Figure 2. Communication entre un émetteur et un récepteur

Dans la figure 3 est présentée la communication PESR lorsque le récepteur arrive le premier au rendez-vous :

- le processus récepteur, P_R, arrivé le premier au rendez-vous (tous les canaux sont libres) – figure 3.a – occupe tous les canaux (figure 3.b) et reste bloqué en attendant l'activation d'un d'entre les événements associés aux canaux (il y a un événement par canal).
- le premier processus émetteur qui arrive au rendez-vous, P_E, effectue la communication (le passage du message) par son canal. P_E active aussi l'événement associé à son canal, qui déterminera le déblocage du processus récepteur. Tous les canaux occupés par le récepteur seront ensuite libérés (figure 3.d).

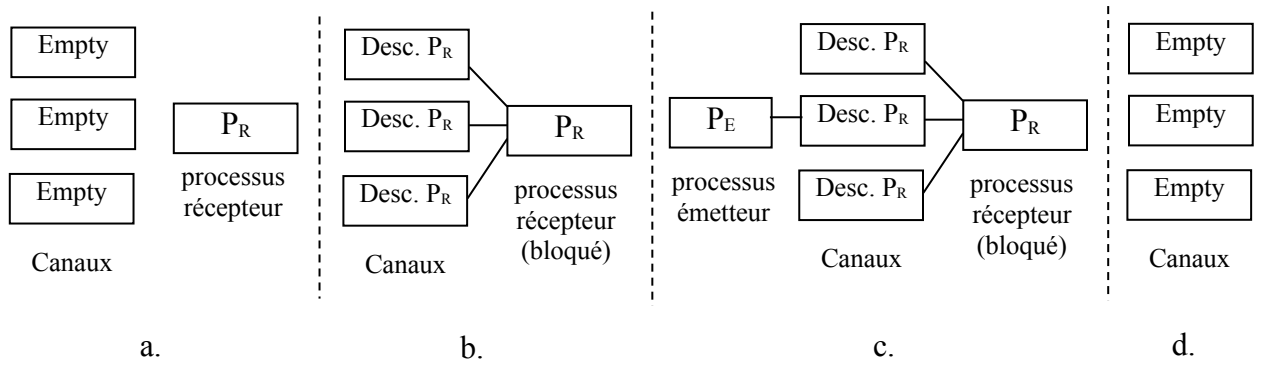

Figure 3. Communication entre plusieurs émetteurs et un récepteur (récepteur premier au RDV)

Dans le cas d'un émetteur arrivé le premier au rendez-vous pour une communication PESR, les étapes sont similaires avec celles du cas de la communication SESR.

Exercices :

1) Quelle sont les avantages qui justifient l'utilisation d'un tel mécanisme de communication ?
2) a) Implémenter la primitive de communication SESR *ChanIn, en vous basant sur ChanOut*, en vous appuyant sur la structure *Channel.*

   b) Proposer un exemple de taches permettant de valider la communication SESR
3) a) Implémenter la primitive *AltIn,* afin de permettre la communication PESR.

   b) Proposer un ensemble de taches permettant de valider la communication PESR.

   c) Comparer vos résultats avec vos voisins. Expliquer les différences éventuelles.
4) Pour aller plus loin…

a) Considérez l'éventualité d'un buffer de réception trop petit, et proposez une solution pour gérer cette situation dans les communications SESR et PESR.

b) Mettez en évidence les problèmes de famine (un ou plusieurs émetteurs ne sont jamais lus) dans la communication PESR, et proposez une solution.

Eléments de base de SystemC

Structure du module

```
SC_MODULE(nom_module)
//ou    class nom_module : public sc_module [, public classA …]
{
public :
       SC_CTOR(nom_module)
       /*ou
       SC_HAS_PROCESS(nom_module);
       nom_module(sc_module_name nom, arg1 [, arg2, …]) : sc_module(nom)
       */
       {
              //enregistrer les processus
                     SC_METHOD(nom_processus);
                     SC_THREAD(nom_processus);
                     SC_CTHREAD(nom_processus, horloge);
              //liste de sensibilité statique
                     //syntaxe fonctionnelle :
                     sensitive(evenement1 [, evenement2, …] );
                     sensitive_pos(evenement1 [, evenement2, …] );
                     sensitive_neg(evenement1 [, evenement2, …] );
                     //syntaxe de stream
                     sensitive << evenement1 << evenement2 …;
                     sensitive << evenement1.pos() ;
                     sensitive << evenement2.neg() ;
                     sensitive_pos << evenement1 << evenement2 …;
                     sensitive_neg << evenement1 << evenement2 …;
              //initialisation des attributs
              //relier les modules et les canaux
                     submoduleA->subport(port);
                     submoduleA->subport(canal);
                     //or par position
                     submoduleA(port, canal);
       }
       //déclaration des ports
              sc_port<nom_interface> [, nombre_de_canaux] nom_port, …
              sc_in<type> [, nom_port, nom_port …];
              sc_out<type> [, nom_port, nom_port …];
              sc_inout<type> [, nom_port, nom_port …];
       //déclaration des canaux locaux
              type_canal nom_canal [, nom_canal, nom_canal …];
       //déclaration des attributs
              type_attribut nom_attribut [, nom_attribut, nom_attribut …];
       //déclaration des processus
              void nom_processus();
       //déclaration d'autres méthodes
       //autres modules
};
```

Structure de la fonction principale

```
#include "systemc.h"
//include autres fichiers de declarations

int sc_main(int argc, char *argv[])
{
        //déclarations de canaux
        //déclarations de variables
        //déclarations des modules
        //liaison des ports des modules
        //unité de temps / établir la résolution
        //initialiser le traçage
        //début de la simulation
        return 0;
}
```

Fonctionnalité

`dont_initialize();`
empêche l'exécution automatique de SC_METHOD ou SC_THREAD au début de la simulation

`next_trigger();`
temporairement outrepasse la liste statique de sensibilité d'une méthode
les arguments spécifient le prochain lancement

`sc_time_stamp();`
renvoie le temps courant de simulation comme `sc_time`

`sc_simulation_time();`
renvoie le temps courrant de simulation comme `double`

`sc_start(value, sc_time_unit);` *ou* **`sc_start();`**
initialise la simulation et avance le temps

`sc_stop();`
arrête la simulation

`sc_set_default_time_unit(value, sc_time_unit);`
`sc_get_default_time_unit();`
fixe et obtient l'unité de temps par défaut

`sc_set_default_time_resolution(value, sc_time_unit);`
`sc_get_default_time_resolution();`
fixe et obtient la résolution minimale

`wait();`
suspend l'exécution d'un thread
les arguments spécifient quand l'exécution sera reprise

`sc_clock``("id", periode, pourcent_vrai, offset, premier_front_positif);`

`sc_event`

- **`notify()`**
  notification
- **`cancel()`**
  annule la notification d'un événement

Types de données

sc_time_unit
SC_FS, SC_PS, SC_NS, SC_US, SC_MS, SC_SEC
sc_int<N>, sc_uint<N>
N: nombre de bits, max = 64
sc_bigint<N>, sc_biguint<N>
N: nombre de bits, max = 512
sc_logic
valeur d'un bit: '0', '1', 'X', 'Z'
sc_lv<N>
vecteur de sc_logic, N est la dimension du vecteur
sc_bit
valeur d'un bit: '0', '1'
sc_bv<N>
vecteur de sc_bit, N est le nombre de bits
sc_fixed<longueur, longueur_partie_entière, quantification, dépassement, nombre_bits_saturés>
sc_ufixed<longueur, longueur_partie_entière, quantification, dépassement, nombre_bits_saturés >

Canaux élémentaires

sc_buffer
communication multi-point, un écrivain, plusieurs lecteurs
implémente l'interface sc_signal_inout_if
sc_fifo<T>
communication point-à-point, un écrivain, un lecteur.
implémente les interfaces sc_fifo_in_if, sc_fifo_out_if
sc_mutex
communication multi-point, utilisé pour accéder une ressource partagée
implémente l'interface sc_mutex_if
sc_semaphore
communication multi-point, accès concurrent limité
implémente l'interface sc_semaphore_if
sc_signal<T>
communication multi-point, un écrivain, plusieurs lecteurs
implémente l'interface sc_signal_inout_if
sc_signal_resolved
plusieurs écrivains
implémente l'interface sc_signal_inout_if
table de décision :

	'0'	'1'	'X'	'Z'
'0'	'0'	'X'	'X'	'0'
'1'	'X'	'1'	'X'	'1'
'X'	'X'	'X'	'X'	'X'
'Z'	'0'	'1'	'X'	'Z'

sc_signal_rv<N>
vecteur de signaux résolus, N est le nombre de signaux

Temps réel – Informations générales

1) Ouverture d'une session

A partir de l'environnement graphique, vous pouvez ouvrir une fenêtre de commande (terminal, **xterm**), qui permet d'entrer de commandes. Pour gérer votre travail, vous pouvez utiliser les commandes Linux et/ou l'interface graphique.

2) Emplacement du code donné

Le répertoire contenant le code donné pour les TPs est :

`/tp/xpe2i3/xpe2i3cr/TP_TR`

Pour le TP1, le code donné se trouve dans les sous-répertoires TP1_A et TP1_B.
Pour le TP numéro *i*, le code donné se trouve dans le sous-répertoire TP*i*, *i* = 2, 3 ou 4.
Copiez le code nécessaire pour chaque TP dans votre compte.
La commande pour copier est : **cp** *source destination*. Pour copier un répertoire entier, utiliser l'option **r**. (cp –r *source destination*).
Exemple :

```
cp -r /tp/xpe2i3/xpe2i3cr/TP_TR/TP1_A /tp/xpe2i3/xpe2i321
```

3) Compilation

A l'aide d'un éditeur de texte (*nedit*, par exemple), écrire votre code. Pour la compilation, on utilise les fichiers *Makefile* et *Makefile.defs* (donnés).
Pour compiler, utilisez la commande **`make`**.

4) Visualisation des chronogrammes

Pour visualiser les chronogrammes, lancez gtkwave (à lancer après la création du fichier de trace, c'est-à-dire après l'exécution du projet !):

`gtkwave nom_fichier.vcd &`

Menu `Search -> Signal Search Hierarchy` pour ajouter les signaux à visualiser.

Pour lancer gtkwave, connectez vous d'abord sur une des stations {cimepc*ii*, *ii* = 66, 67, 68, 71, 78}.
Exemple de connexion : `ssh -XC cimepc66`

5) Documentation

```
/usr/local/systemc2_1/docs/
/softs/utilitaires_sol10/systemc2_2/docs/
```

http://www.systemc.org/

Pour ouvrir un fichier avec Acrobat Reader, tapez: `acroread nom_fichier &`
Pour lancer mozilla, tapez : `/softs/mozilla/mozilla &`