

Modélisation Transactionnelle des Systèmes sur Puce avec SystemC

Ensimag 3A — filière SEOC

Grenoble-INP

Modélisation TLM en SystemC

Frédéric Pétrot

frederic.petrot@univ-grenoble-alpes.fr

2021-2022



Planning des séances

- 01/12/21 (FP) CM1 Introduction : systèmes sur puce et modélisation au niveau transactionnel
- 08/12/21 (FP) CM2 Introduction au C++ et présentation de SystemC
- 08/12/21 (FP) CM3 Communications haut-niveau et modélisation TLM en SystemC
- 15/12/21 (FP) CM4 Intervenant extérieur : Jérôme Cornet (STMicroelectronics)
- 15/12/21 (FP) TP1 (1/1) : Plateforme matérielle SystemC/TLM
- 06/01/22 (FP) CM5 Utilisations des plateformes TLM
- 06/01/22 (FP) CM6 Notions Avancées en SystemC/TLM
- 12/01/22 (FP) TP2 (1/2) : Intégration du logiciel embarqué
- 12/01/22 (FP) TP2 (2/2) : Intégration du logiciel embarqué
- 19/01/22 (OM) CM7 Synthèse d'architecture
- 19/01/22 (OM) TP3 (1/2) : Synthèse de haut niveau et génération de circuits numériques
- 21/01/22 (OM) TP4 (2/2) : Synthèse de haut niveau et génération de circuits numériques

Sommaire

- 1 Le but ...
- 2 Dernières notions de SystemC
- 3 Bibliothèque TLM 2.0

Sommaire

- 1 Le but ...
- 2 Dernières notions de SystemC
- 3 Bibliothèque TLM 2.0

Ce qu'on veut pouvoir écrire

Côté initiateur

```
ensitlm::data_t val = 1;
ensitlm::addr_t addr = 2;
while (true) {
    cout << "Entrer x :" << endl;
    cin >> val;;
    socket.write(addr, val);
}
```

Côté cible

```
tlm_response_status
write(const ensitlm::addr_t &a,
      const ensitlm::data_t &d) {
    cout << "j'ai reçu : "
          << d << endl;
    return TLM_OK_RESPONSE;
}
```

Sommaire

- 1 Le but ...
- 2 Dernières notions de SystemC
- 3 Bibliothèque TLM 2.0

Sommaire de cette section

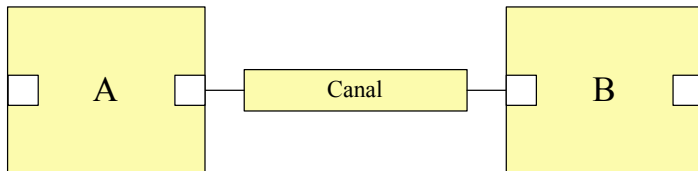
- 2 Dernières notions de SystemC
 - Exports
 - Exports & TLM

Rappel

- **Port** : expose une **interface** à un point de connexion
- **Canal** : **implémente** les différentes interfaces requises pour réaliser la communication
- Utilisation dans les modules : appels de méthodes sur les ports à travers l'opérateur « -> » redéfini
- Appel de méthode par le port dans un module \Rightarrow appel de la même méthode dans le canal auquel est relié le port

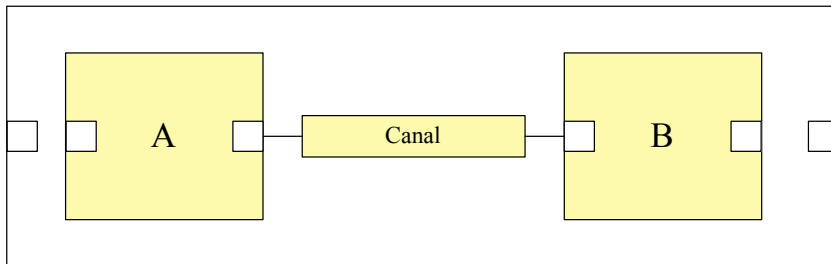
Problème : exposé

- Assemblage d'origine



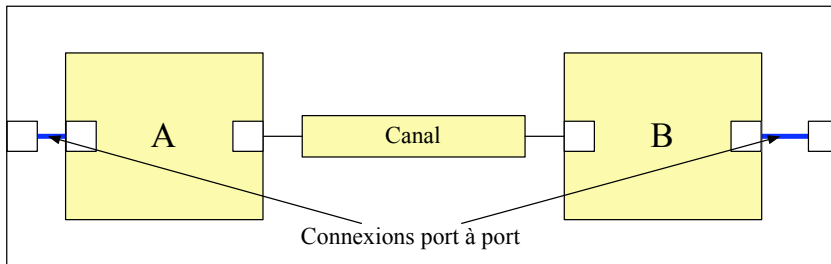
Problème : exposé

- Intégration en un seul composant ?



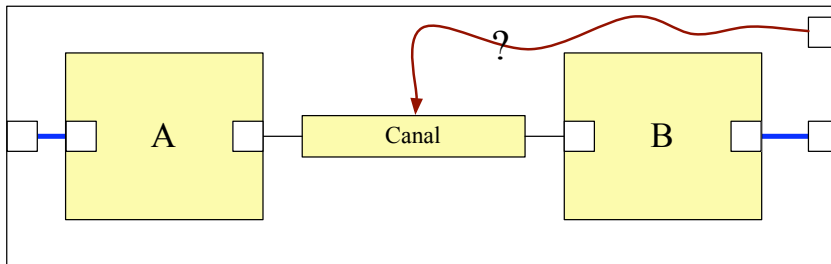
Problème : exposé

- Connexions port à port



Problème : exposé

- Cas problématique :

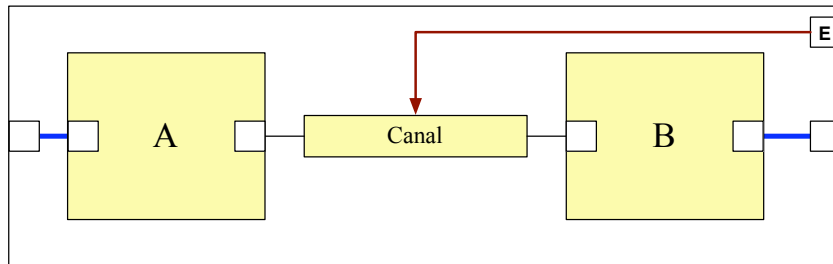


Export : la classe `sc_export`

- Élément (similaire à un port) :
 - ▶ **exposant une interface** à un point de connexion
 - ▶ **connecté à un objet**, auquel il transmet les appels de méthodes
- En pratique :
 - ▶ Objet de la classe `sc_export`
 - ▶ Généricité sur l'interface (comme `sc_port`)
 - ▶ Nécessité de connexion explicite dans le code à l'objet récepteur des appels de méthodes

Export : exemple (1/2)

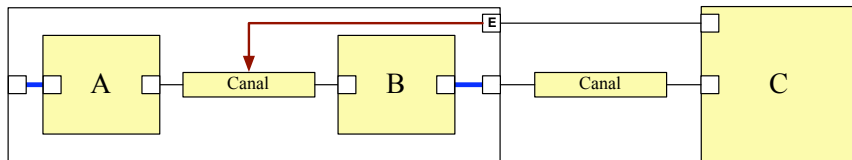
- Sur l'exemple précédent :



- `E->methode()` \Leftrightarrow `canal->methode()`


Export : exemple (2/2)

- Intégration du composant créé :



- C peut appeler directement les méthodes du canal.

Retour sur les appels effectués (2/2)

- Appels sur ports et exports () :

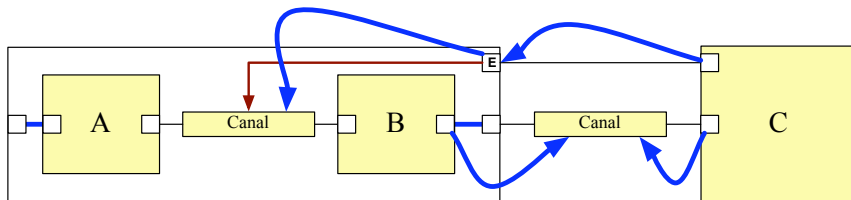
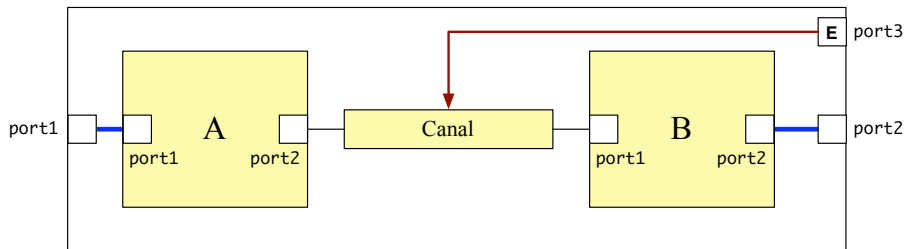


Schéma du module à créer



Exemple (déclaration)

Cf. `code/exports/`

Exemple (déclaration)

```
SC_MODULE (AetB)
{
    // ports et exports
    sc_port<my_interface>    port1, port2;
    sc_export<my_interface>  port3;

    // Constructeur
    SC_CTOR (AetB) ;

    // Objets internes
    A                a;
    B                b;
    Canal            canal;
}
```

Exemple (constructeur)

```
AetB::AetB(sc_module_name name)
    : sc_module(name),
      a(sc_gen_unique_name("A")),
      b(sc_gen_unique_name("B")),
      canal(sc_gen_unique_name("canal"))
{
    // connexions internes
    a.port2(canal);
    b.port1(canal);

    // connexions port a port vers l'exterieur
    a.port1(port1);
    b.port2(port2);

    // connexion de l'export
    port3.bind(canal);
}
```

Exemple (sc_main)

```
int sc_main(int, char**)
{
    AetB          aetb("AetB");
    C             c("C");
    QuickChannel  q1("q1"), q2("q2"), q3("q3");

    aetb.port1.bind(q1);
    c.port1.bind(q1);

    aetb.port2.bind(q2);
    c.port2.bind(q2);

    c.port3.bind(aetb.port3);

    sc_start(); return 0;
}
```

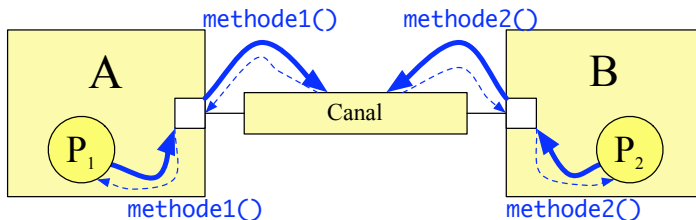
Sommaire de cette section

2 Dernières notions de SystemC

- Exports
- Exports & TLM

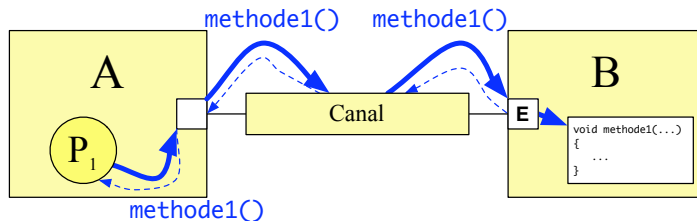
Communications vues jusqu'ici

- Chaque module est « actif »



Communications TLM

- Modules actifs, passifs, actifs/passifs



- A peut appeler directement des méthodes de B

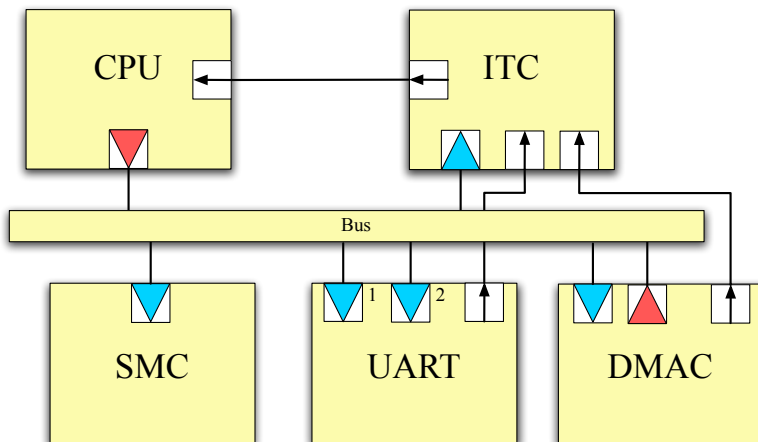
Sommaire

- 1 Le but ...
- 2 Dernières notions de SystemC
- 3 Bibliothèque TLM 2.0

Sommaire de cette section

- 3 Bibliothèque TLM 2.0
 - Présentation
 - Le chemin d'une transaction
 - Couche transport
 - Protocole Ensitlm

Ce que l'on souhaite modéliser



Pourquoi standardiser TLM 2 ?

- Historique :
 - ▶ **SystemC 2.0** : notion de `sc_interface`. Chaque entreprise peut coder ses propres canaux de communications

Pourquoi standardiser TLM 2 ?

- Historique :

- ▶ **SystemC 2.0** : notion de `sc_interface`. Chaque entreprise peut coder ses propres canaux de communications
 - ★ **Problème** : un composant écrit avec le protocole de l'entreprise X ne peut pas se connecter sur le canal de l'entreprise Y !
 - ★ Difficulté à intégrer des composants venant d'entreprises tierces ;
 - ★ Contournements avec des adaptateurs (aussi nommés *wrappers*, lents, pas toujours possibles)

Pourquoi standardiser TLM 2 ?

- Historique :

- ▶ SystemC 2.0 : notion de `sc_interface`. Chaque entreprise peut coder ses propres canaux de communications
 - ★ Problème : un composant écrit avec le protocole de l'entreprise X ne peut pas se connecter sur le canal de l'entreprise Y !
 - ★ Difficulté à intégrer des composants venant d'entreprises tierces ;
 - ★ Contournements avec des adaptateurs (aussi nommés *wrappers*, lents, pas toujours possibles)
- ▶ TLM-1.0 : un pas vers l'interopérabilité
 - ★ Définition d'une interface (template)
 - ★ Mais rien sur le contenu des transactions
 - ★ ⇒ seulement une petite partie d'un vrai protocole standardisé !

Pourquoi standardiser TLM 2 ?

● Historique :

- ▶ SystemC 2.0 : notion de `sc_interface`. Chaque entreprise peut coder ses propres canaux de communications
 - ★ Problème : un composant écrit avec le protocole de l'entreprise X ne peut pas se connecter sur le canal de l'entreprise Y !
 - ★ Difficulté à intégrer des composants venant d'entreprises tierces ;
 - ★ Contournements avec des adaptateurs (aussi nommés *wrappers*, lents, pas toujours possibles)
- ▶ TLM-1.0 : un pas vers l'interopérabilité
 - ★ Définition d'une interface (template)
 - ★ Mais rien sur le contenu des transactions
 - ★ \Rightarrow seulement une petite partie d'un vrai protocole standardisé !
- ▶ **TLM-2.0** : l'interopérabilité se rapproche ...
 - ★ Contenu des transactions défini

Architecture de la bibliothèque

- Généricité
- Couche Transport
 - ▶ Mécanismes génériques de transmission des transactions
 - ▶ Permet de modéliser n'importe quel protocole de bus
 - ▶ **Standardisée**
- Couche Protocole
 - ▶ Contenu des transaction standardisé (`tlm::tlm_generic_payload`)
 - ▶ Comportement
 - ▶ “Interfaces de convenances” pour rendre le code plus concis.
 - ▶ Étude d'un exemple : protocole **Ensitlm**
- Couche Utilisateur
 - ▶ Ce que le programmeur doit mettre dans ses modules...

Interfaces de convenances

- Problème : mettre tout le monde d'accord sur l'API utilisateur est
 - ▶ **Difficile** (déjà des années de discussions entre vendeurs pour arriver à TLM-2)

Interfaces de convenances

- Problème : mettre tout le monde d'accord sur l'API utilisateur est
 - ▶ **Difficile** (déjà des années de discussions entre vendeurs pour arriver à TLM-2)
 - ▶ **Pas très utile** : l'important est de pouvoir connecter un composant écrit par X à un canal écrit par Y, pas le code écrit à l'intérieur de Y.

Interfaces de convenances

- Problème : mettre tout le monde d'accord sur l'API utilisateur est
 - ▶ Difficile (déjà des années de discussions entre vendeurs pour arriver à TLM-2)
 - ▶ Pas très utile : l'important est de pouvoir connecter un composant écrit par X à un canal écrit par Y, pas le code écrit à l'intérieur de Y.
- ⇒ TLM-2 définit une API générique mais très verbeuse
- Chaque entreprise peut écrire une API qui lui convient.

Notre interface de convenance : Ensitlm

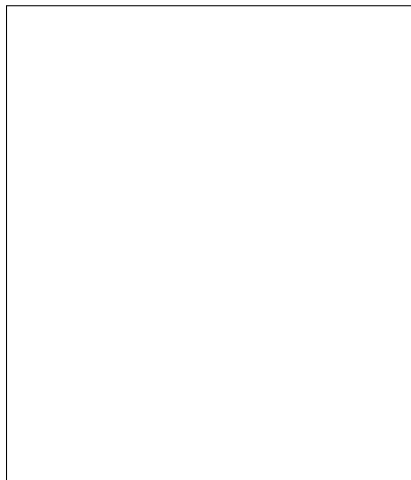
- Faite maison ! (Giovanni Funchal)
- Objectifs :
 - ▶ simplicité du code (\Rightarrow allez voir comment c'est fait !)
 - ▶ simplicité d'utilisation (vous me remercirez bientôt ;-)
- Beaucoup de limitations, mais suffisante pour les TPs.

Sommaire de cette section

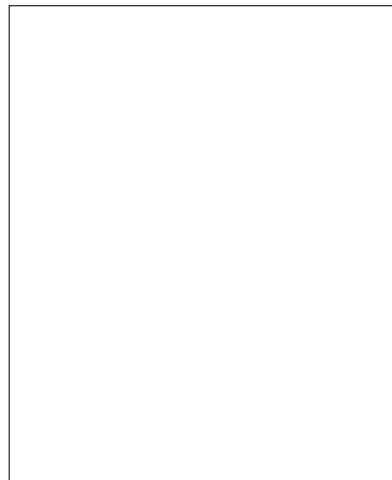
- 3 Bibliothèque TLM 2.0
 - Présentation
 - **Le chemin d'une transaction**
 - Couche transport
 - Protocole Ensitlm

Chemin d'une transaction : l'idée ...

Initiateur

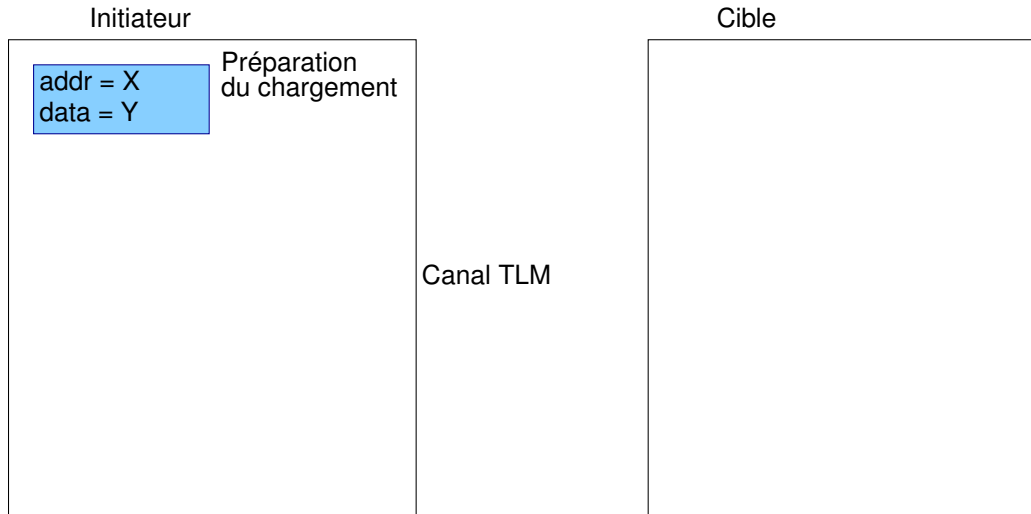


Cible

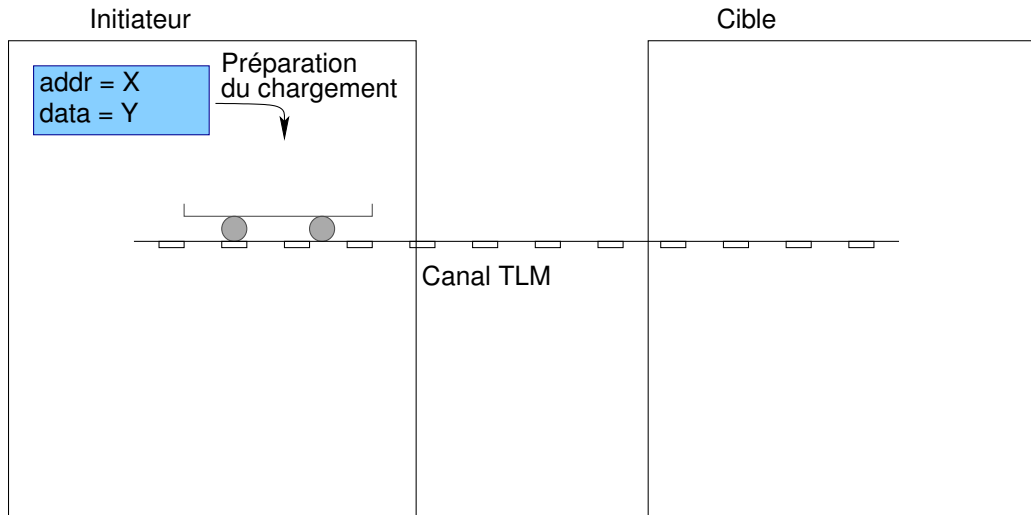


Canal TLM

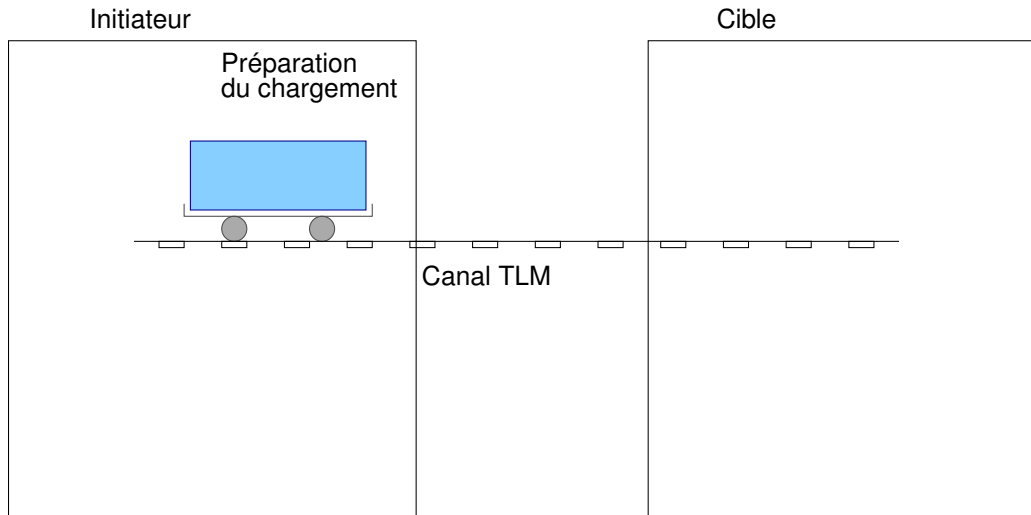
Chemin d'une transaction : l'idée ...



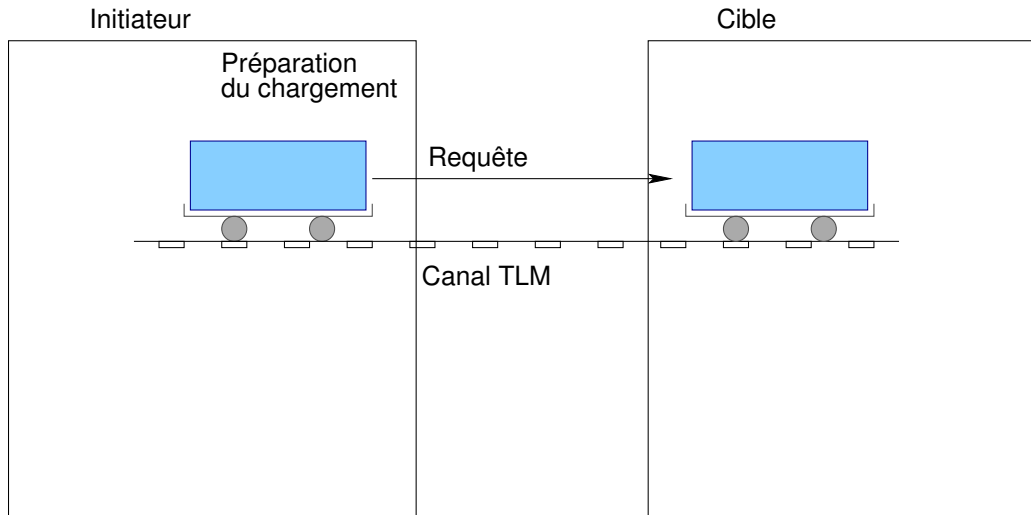
Chemin d'une transaction : l'idée ...



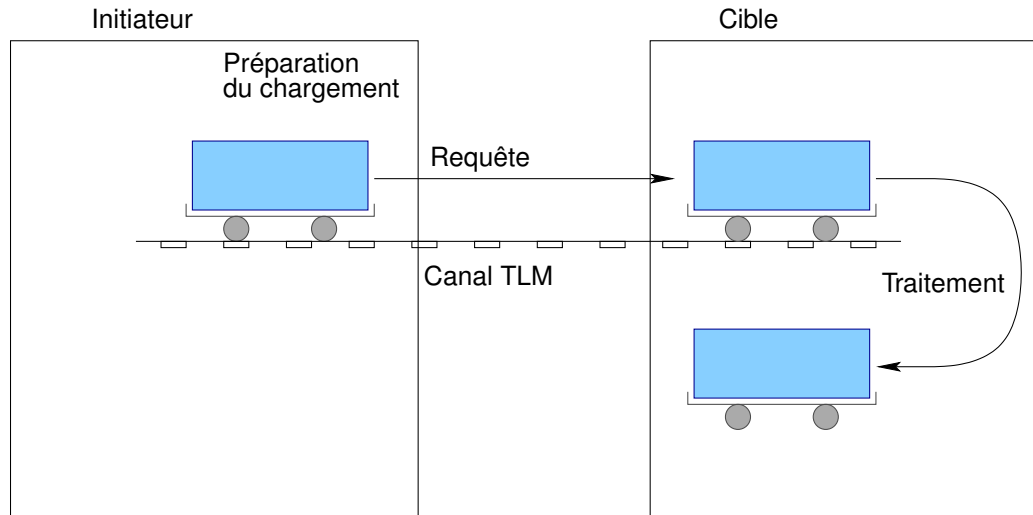
Chemin d'une transaction : l'idée ...



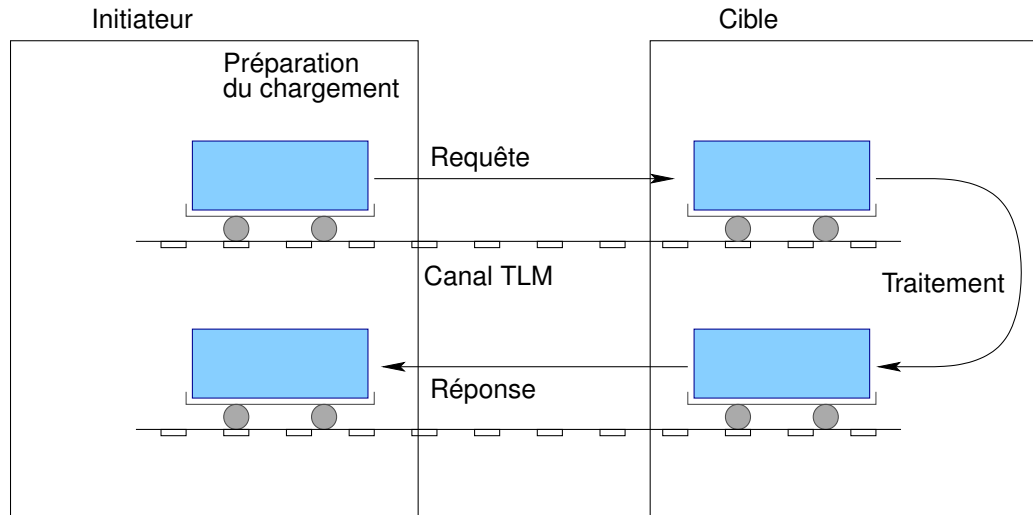
Chemin d'une transaction : l'idée ...



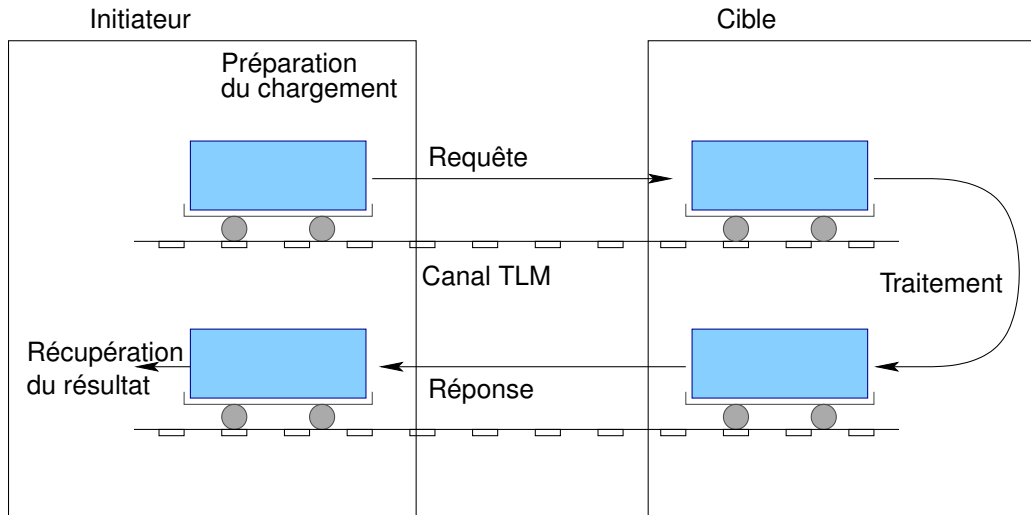
Chemin d'une transaction : l'idée ...



Chemin d'une transaction : l'idée ...



Chemin d'une transaction : l'idée ...



Des tonnes de variantes

- 1 appel de fonction, ou plusieurs phases successives
- Connexion point à point, ou via un canal
- Communication de valeurs ou d'un bloc de valeurs
- Possibilité de rendre la main ou pas
- ...

Sommaire de cette section

- 3 Bibliothèque TLM 2.0
 - Présentation
 - Le chemin d'une transaction
 - **Couche transport**
 - Protocole Ensitlm

Couche transport (1/4)

- Interface pour transactions bloquantes

- ▶ Toute la transaction doit se faire en un appel de fonction
- ▶ Interface `tlm_blocking_transport_if<TRANS>`

```
template <typename TRANS = tlm_generic_payload>
struct tlm_blocking_transport_if :
    virtual sc_core::sc_interface {
    virtual void b_transport(TRANS& trans,
                            sc_core::sc_time& t) = 0;
};
```

Couche transport (1/4)

- Interface pour transactions bloquantes

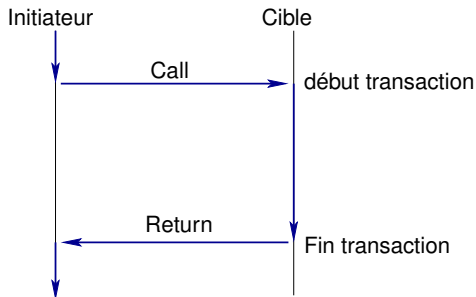
- ▶ Toute la transaction doit se faire en un appel de fonction
- ▶ Interface `tlm_blocking_transport_if<TRANS>`

```
template <typename TRANS = tlm_generic_payload>
struct tlm_blocking_transport_if :
    virtual sc_core::sc_interface {
    virtual void b_transport(TRANS& trans,
                            sc_core::sc_time& t) = 0;
};
```

- ▶ Communication initiateur/cible :
 - ★ initiateur → cible : transaction passée en argument
→ **Call path**
 - ★ cible → initiateur : même transaction (passée par référence)
→ **Return path**
- ▶ (Pour l'instant, on ignore le deuxième argument de `b_transport`)

Couche transport (2/4)

- Message Sequence Chart pour transport **bloquant** :



Couche transport (3/4)

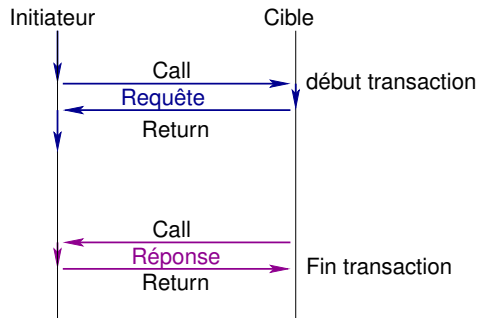
- Interface pour transactions non-bloquantes
 - ▶ L'initiateur fait un appel de fonction : requête
 - ▶ La cible fait un appel de fonction dans l'autre sens : réponse

```
// ForWard path
template <typename TRANS = tlm_generic_payload,
          typename PHASE = tlm_phase>
struct tlm_fw_nonblocking_transport_if :
    virtual sc_core::sc_interface {
    virtual tlm_sync_enum nb_transport_fw
        (TRANS& trans, PHASE& phase, sc_time& t) = 0;
};

// BackWard path
template <typename TRANS = tlm_generic_payload,
          typename PHASE = tlm_phase>
struct tlm_bw_nonblocking_transport_if :
    virtual sc_core::sc_interface {
    virtual tlm_sync_enum nb_transport_bw
        (TRANS& trans, PHASE& phase, sc_time& t) = 0;
};
```

Couche transport (4/4)

- Message Sequence Chart pour transport **non-bloquant** :



Interface des composants

- Un composant TLM **initiateur** peut :
 - ▶ Lancer une transaction bloquante
 - ▶ Lancer une transaction non-bloquante
 - ▶ Recevoir une réponse de transaction non-bloquante

Interface des composants

- Un composant TLM **initiateur** peut :
 - ▶ Lancer une transaction bloquante
i.e. **appeler** `b_transport`
 - ▶ Lancer une transaction non-bloquante
i.e. **appeler** `nb_transport_fw`
 - ▶ Recevoir une réponse de transaction non-bloquante
i.e. **exposer une fonction** `nb_transport_bw`

Interface des composants

- Un composant TLM initiateur peut :
 - ▶ Lancer une transaction bloquante
i.e. appeler `b_transport`
 - ▶ Lancer une transaction non-bloquante
i.e. appeler `nb_transport_fw`
 - ▶ Recevoir une réponse de transaction non-bloquante
i.e. exposer une fonction `nb_transport_bw`
- Un composant TLM **cible** peut :
 - ▶ Recevoir une transaction bloquante
 - ▶ Recevoir une requête de transaction non-bloquante
 - ▶ Envoyer une réponse à une transaction non-bloquante

Interface des composants

- Un composant TLM initiateur peut :
 - ▶ Lancer une transaction bloquante
i.e. appeler `b_transport`
 - ▶ Lancer une transaction non-bloquante
i.e. appeler `nb_transport_fw`
 - ▶ Recevoir une réponse de transaction non-bloquante
i.e. exposer une fonction `nb_transport_bw`
- Un composant TLM **cible** peut :
 - ▶ Recevoir une transaction bloquante
i.e. **exposer une fonction** `b_transport`
 - ▶ Recevoir une requête de transaction non-bloquante
i.e. **exposer une fonction** `nb_transport_fw`
 - ▶ Envoyer une réponse à une transaction non-bloquante
i.e. **appeler** `nb_transport_bw`

Exporter/appeler une fonction (1/2)

Question



Comment un module expose-t-il une fonction aux autres objets ?

Exporter/appeler une fonction (1/2)

Question



Comment un module expose-t-il une fonction aux autres objets ?

- `sc_export` !

Exporter/appeler une fonction (1/2)

Question



Comment un module expose-t-il une fonction aux autres objets ?

- `sc_export` !

Question



Comment un module appelle-t-il une fonction d'un autre objet ?

Exporter/appeler une fonction (1/2)

Question



Comment un module expose-t-il une fonction aux autres objets ?

- `sc_export` !

Question



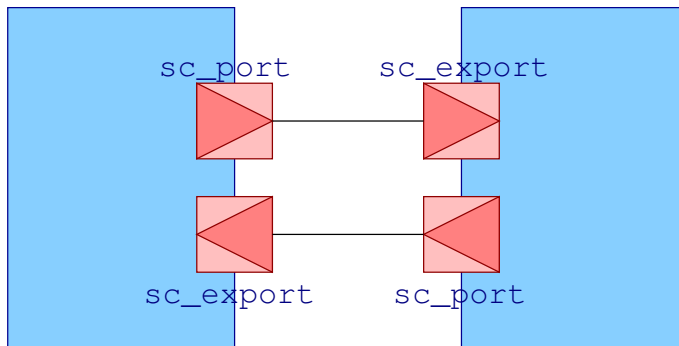
Comment un module appelle-t-il une fonction d'un autre objet ?

- `sc_port` !

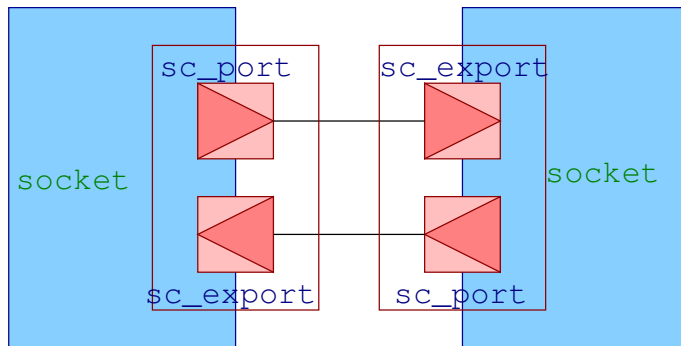
Exporter/appeler une fonction (1/2)



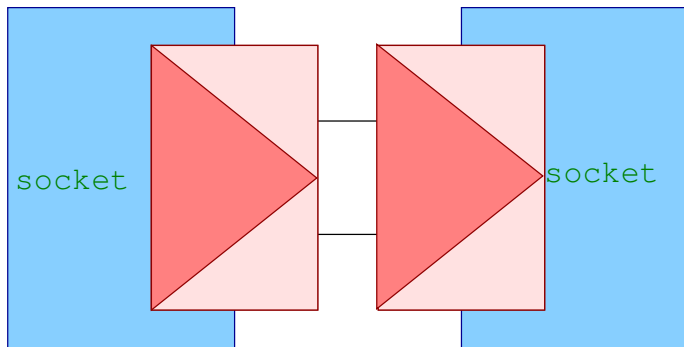
Exporter/appeler une fonction (1/2)



Exporter/appeler une fonction (1/2)



Exporter/appeler une fonction (1/2)



⇒ en TLM-2, on n'utilise plus que des sockets
(mais il y a quand même des ports et exports sous le capot)

Sockets et TLM-2

- Beaucoup de types de sockets différents.
- On va utiliser `tlm::tlm_initiator_socket /`
`tlm::tlm_target_socket` et en dériver `ensitlm::initiator_socket`
`/ensitlm::target_socket`.

Communication entre N composants

- Jusqu'ici, on n'a fait que du point à point ...

Question



Que manque-t-il ?

Communication entre N composants

- Jusqu'ici, on n'a fait que du point à point ...

Question



Que manque-t-il ?

- Connexion N initiateurs vers M cible.
- Routage (choisir à quelle cible on parle) → addressmap.

Communication entre N composants

- Jusqu'ici, on n'a fait que du point à point ...

Question



Que manque-t-il ?

- Connexion N initiateurs vers M cible.
- Routage (choisir à quelle cible on parle) → addressmap.

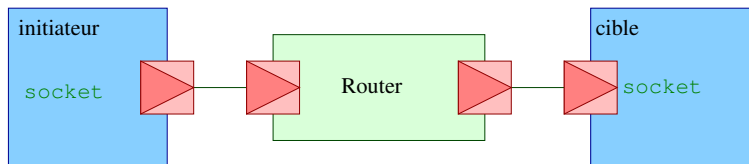
Question



Comment faire ?

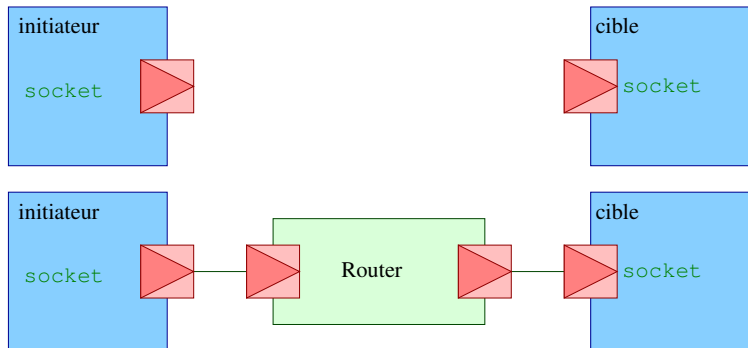
Modéliser l'interconnexion

- On ajoute un composant pour modéliser le bus.
- Une solution (pas exactement celle de Ensitlm) :



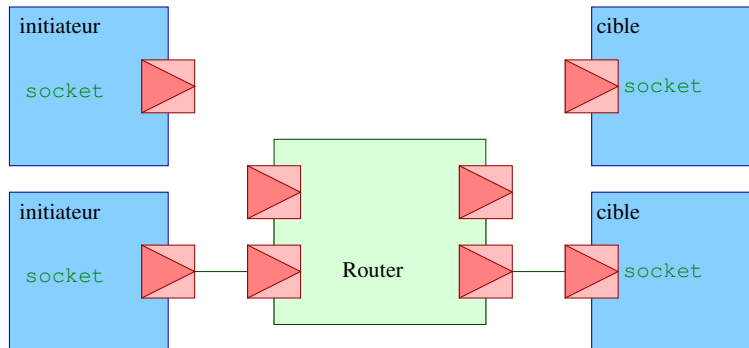
Modéliser l'interconnexion

- On ajoute un composant pour modéliser le bus.
- Une solution (pas exactement celle de Ensitlm) :



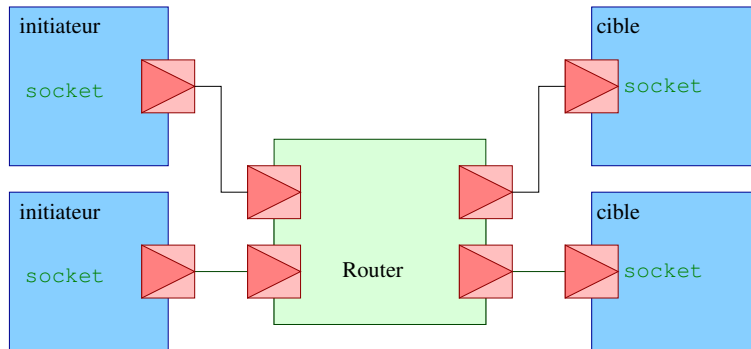
Modéliser l'interconnexion

- On ajoute un composant pour modéliser le bus.
- Une solution (pas exactement celle de Ensitlm) :



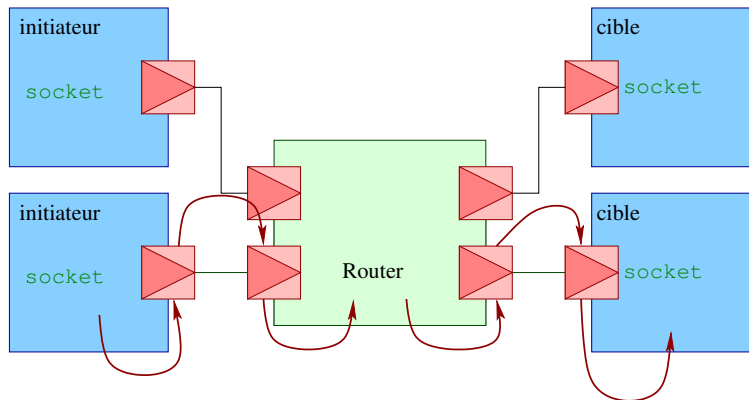
Modéliser l'interconnexion

- On ajoute un composant pour modéliser le bus.
- Une solution (pas exactement celle de Ensitlm) :



Modéliser l'interconnexion

- On ajoute un composant pour modéliser le bus.
- Une solution (pas exactement celle de Ensitlm) :



Sommaire de cette section

- 3 Bibliothèque TLM 2.0
 - Présentation
 - Le chemin d'une transaction
 - Couche transport
 - **Protocole Ensitlm**

Ce qu'on veut pouvoir écrire

Côté initiateur

```
ensitlm::data_t val = 1;
ensitlm::addr_t addr = 2;
while (true) {
    cout << "Entrer x :" << endl;
    cin >> val;;
    socket.write(addr, val);
}
```

Côté cible

```
tlm_response_status
write(const ensitlm::addr_t &a,
      const ensitlm::data_t &d) {
    cout << "j'ai reçu : "
         << d << endl;
    return TLM_OK_RESPONSE;
}
```

Ensitlm : limitations

- Protocole bloquant seulement
(On ne s'embête pas avec le "backward path")
- Pas de généricité :
 - ▶ adresses : `typedef uint32_t addr_t;`
 - ▶ données : `typedef uint32_t data_t;`
- Pas de byte-enable,
- Pas de transaction par bloc,
- Seulement deux commandes : read/write,
- Peu d'optimisations de performances possibles.

Ensitlm : principe

- `ensitlm/initiator_socket.h` : pour ne pas avoir à construire explicitement une `tlm_generic_payload`.
- `ensitlm/target_socket.h` : pour ne pas avoir à écrire une méthode `b_transport`, mais juste `read` et `write`.
- `bus.h` : une classe `Bus`.

Pour utiliser Ensitlm

```
/* pour utiliser les sockets */  
#include "ensitlm.h"
```

Exemples de code

- `code/ensitlm-mini` : exemple minimaliste, un seul fichier (pas très propre, mais pratique pour avoir une vue d'ensemble).
- `code/ensitlm-mini-multi` : le même exemple, avec un découpage 1 classe = 1 fichier `.h` + 1 fichier `.cpp`.

ensitlm_initiator_socket.h (1/4)

- Le code : déclaration

```
namespace ensitlm {  
  
    template <typename MODULE>  
    class initiator_socket :  
        public tlm::tlm_initiator_socket  
    { ... }
```

ensitlm_initiator_socket.h (1/4)

- Le code : déclaration

```
namespace ensitlm {  
  
    template <typename MODULE,  
              bool MULTIPORT = false>  
    class initiator_socket :  
        public tlm::tlm_initiator_socket  
    { ... }
```

ensitlm_initiator_socket.h (1/4)

- Le code : déclaration

```
namespace ensitlm {  
  
    template <typename MODULE,  
              bool MULTIPORT = false>  
    class initiator_socket :  
    public tlm::tlm_initiator_socket  
        <CHAR_BIT * sizeof(data_t),  
        tlm::tlm_base_protocol_types,  
        MULTIPORT?0:1>  
    { ... }
```

ensitlm_initiator_socket.h (1/4)

- Le code : déclaration

```
namespace ensitlm {  
  
    template <typename MODULE,  
              bool MULTIPORT = false>  
    class initiator_socket :  
    public tlm::tlm_initiator_socket  
        <CHAR_BIT * sizeof(data_t),  
         tlm::tlm_base_protocol_types,  
         MULTIPORT?0:1>,  
    private tlm::tlm_bw_transport_if  
        <tlm::tlm_base_protocol_types>  
    { ... }
```

ensitlm_initiator_socket.h (2/4)

- Le code : API

```
[...]  
class initiator_socket : [...] {  
    initiator_socket();  
    explicit initiator_socket(const char* name);  
  
    tlm::tlm_response_status  
    read(const addr_t& addr, data_t& data,  
         int port = 0);  
  
    tlm::tlm_response_status  
    write(const addr_t& addr, data_t data,  
          int port = 0);  
};
```

initiator_socket.h (3/4)

- Utilisation :

```
#include "ensitlm.h"

struct Foo : sc_core::sc_module
{
    ensitlm::initiator_socket<Foo> socket;
    SC_CTOR(Foo);
    void compute() {
        // ...
        status = socket.write(i, data);
        if (status != tlm::TLM_OK_RESPONSE) ...;
        // ...
    }
};
```

ensitlm_initiator_socket.h (4/4)

- Ce que vous économisez à chaque read/write :

```
tlm::tlm_response_status read(const addr_t& addr,
                              data_t& data, int port = 0) {
    tlm::tlm_generic_payload* trans;
    if (!container.empty()) {
        trans = container.back();
        container.pop_back();
    } else {
        trans = new tlm::tlm_generic_payload();
    }
    trans->set_command(tlm::TLM_READ_COMMAND);
    trans->set_address(addr);
    trans->set_data_ptr
        (reinterpret_cast<unsigned char*>(&data));
    trans->set_data_length(sizeof(data_t));
    trans->set_streaming_width(sizeof(data_t));
    (*this)[port]->b_transport(*trans, time);
    container.push_back(trans);
    return trans->get_response_status();
}
```

ensitlm_target_socket.h (1/4)

- Le code : déclaration

```
namespace ensitlm {  
  
    template <typename MODULE,  
              bool MULTIPORT = false>  
    class target_socket :  
    public tlm::tlm_target_socket  
        <CHAR_BIT * sizeof(data_t),  
         tlm::tlm_base_protocol_types,  
         MULTIPORT:0?1>,  
    public tlm::tlm_fw_transport_if  
        <tlm::tlm_base_protocol_types>  
    { ... };  
};
```


ensitlm_target_socket.h (2/5)

- La fonction `b_transport` :
 - ▶ C'est la fonction appelée par l'initiateur (le bus)
 - ▶ Appelle des fonctions `read` et `write` sur le module englobant
 - ▶ \Rightarrow l'utilisateur devra définir les fonctions `read` et `write`.

ensitlm_target_socket.h (3/5)

- Le code : la fonction `b_transport`

```
void b_transport(tlm::tlm_generic_payload& trans,
                 sc_core::sc_time& t) {
    addr_t addr = static_cast<addr_t>(trans.get_address());
    data_t& data = *(reinterpret_cast<data_t*>
                     (trans.get_data_ptr()));
    switch(trans.get_command()) {
        case tlm::TLM_READ_COMMAND:
            trans.set_response_status(m_mod->read(addr, data));
            break;
        case tlm::TLM_WRITE_COMMAND:
            trans.set_response_status(m_mod->write(addr, data));
            break;
        case tlm::TLM_IGNORE_COMMAND:
            break;
        default:
            trans.set_response_status
                (tlm::TLM_COMMAND_ERROR_RESPONSE);
    }
}
```

ensitlm_target_socket.h (4/5)

- Utilisation : implémenter `read/write`

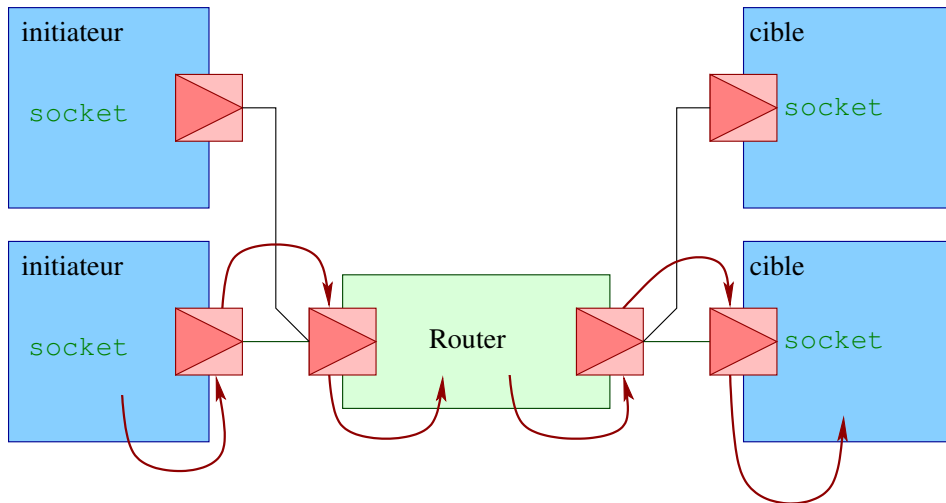
```
#include "ensitlm_target_socket.h"
struct target : sc_module {
    ensitlm::target_socket<target> socket;
    tlm::tlm_response_status write(const ensitlm::addr_t &a,
                                   const ensitlm::data_t &d) {
        cout << "j'ai reçu : " << d << endl;
        return tlm::TLM_OK_RESPONSE;
    }
    tlm::tlm_response_status read (const ensitlm::addr_t &a,
                                   ensitlm::data_t &d) {
        // [...]
    }
};
```

ensitlm_target_socket.h (5/5)

- Pour implémenter `read/write` :
 - ▶ Doivent avoir **exactement** le même type que `read/write` de la classe de base (copier/coller ...)
 - ▶ Reçoivent des adresses relatives au début de bloc (i.e. une écriture à l'adresse 142 sur un module « mappé » sur l'intervalle [100, 200[donne une adresse 42 côté cible)
 - ▶ `read` peut modifier la donnée, `write` ne peut pas.

ensitlm_router.h (1/3)

- Utilise des sockets “multiport” :



bus.h (2/3)

- Le code :

```
SC_MODULE(Bus) {  
    // Parametre 'true' pour connection multiport  
    // (Specificite du bus)  
    ensitlm::initiator_socket<Bus, true> initiator;  
    ensitlm::target_socket<Bus, true> target;  
  
    Bus(sc_core::sc_module_name name);  
  
    tlm::tlm_response_status  
        read(ensitlm::addr_t a, ensitlm::data_t& d);  
    tlm::tlm_response_status  
        write(ensitlm::addr_t a, ensitlm::data_t d);  
    void map(ensitlm::compatible_socket& port,  
            ensitlm::addr_t start_addr, ensitlm::addr_t size);  
    // ...  
};
```

- Un module (presque) comme les autres.

bus.h (3/3)

- Utilisation :

```
int sc_main(int, char**)
{
    Generator    generator1("Generator1");
    Memory       memory("Memory", 100);
    Bus          router("router");

    generator1.socket.bind(router.target);
    router.initiator.bind(memory.target);

    // address map
    //          target port | address | size
    router.map(memory.target, 0      , 100);

    sc_core::sc_start(); return 0;
}
```

Conclusion

- TLM-2 :
 - ▶ Interfaces standardisées,
 - ▶ Contenu de transaction standardisée,
 - ▶ Comportement des bus laissés à l'utilisateur.
- Protocole Ensitlm : ce que l'on va utiliser en TP
 - ▶ Plus concis que TLM-2 « brut »
 - ▶ Router avec addressmap
- Et les interruptions ?
 - ▶ Plusieurs solutions...
 - ▶ Utilisation de `sc_in`, `sc_out`, etc. pas parfaite mais raisonnable.