Modélisation SystemC-TLM de systèmes à base de processeur

S. Jovanovic et S. Weber,
Institut Jean Lamour (UMR7198) et Pôle CNFM MIGREST – Nancy, Université de Lorraine, Nancy, France
Contact email : slavisa.jovanovic@univ-lorraine.fr

Nous présentons un ensemble de travaux pratiques qui seront dispensés au sein du Master EEA - Électronique Embarquée à l'université de Lorraine dans le cadre du module « Modélisation SystemC ». Ces TP sont destinés à initier les étudiants à la modélisation de systèmes et circuits numériques en SystemC-TLM et sont organisés autour de la suite logicielle open source *Eclipse* et de la chaine de compilation *gcc* pour la simulation, test et vérification.

I. Introduction

Pour initier les étudiants de la formation Master EEA-EE à l'université de Lorraine à la modélisation de systèmes et circuits en SystemC-TLM, nous allons mettre en place une série de travaux pratiques autour des logiciels open source (Eclipse, chaine de compilation gcc) permettant de simuler, tester et vérifier des systèmes complexes comportant des blocs mémoire et des processeurs. La modélisation d'un système en SystemC-TLM s'inscrit dans une unité de 30 heures comprenant:

- Introduction à la modélisation SystemC-TLM (flot de conception, architecture du langage, modules, canaux primitifs et hiérarchiques, interfaces, processus, types de données, scheduler, noyau, temps, modélisation TLM): 10h de cours
- Introduction à la simulation SystemC: module, hiérarchie de modules, vérification et test (4h de TP)
- Modélisation d'une communication comportementale: utilisation de canaux primitifs de type FIFO pour les connexions (4h de TP)
- Modélisation d'un système TLM de base: réalisation d'un routeur permettant de connecter un module initiateur à 2 blocs mémoire en SystemC-TLM (4h de TP)
- Modélisation d'un système à processeur en SystemC-TLM: réalisation d'un système comportant un processeur modelisé par son simulateur de jeu d'instructions (*Instruction Set Simulator* - *ISS*), un bloc mémoire et des périphériques (UART, etc) (8h de TP)

Cette unité s'adresse aux étudiants de niveau Master 2 ayant déjà un bagage de connaissances solide en conception numérique (VHDL, FPGA - au moins 30h), en conception de circuits microélectroniques de base (60h) et en conception de circuits VLSI numériques de complexité moyenne avec la suite logicielle Cadence (30h). De plus, les étudiants ont des connaissances en programmation orientée objet (Java), en programmation structurelle sur microcontrôleur (langage C) et en programmation C sous Linux (compilation croisée) leur permettant d'assimiler les concepts liés à la modélisation SystemC-TLM de manière plus aisée. Par conséquent, on peut considérer que les étudiants ont une expérience suffisante et non négligeable pour mener à bien le projet proposé.

II. Démarche pédagogique

La modélisation de systèmes à processeur en SystemC-TLM passe par les étapes suivantes:

Article publié par EDP Sciences et disponible sur le site https://www.j3ea.org ou https://doi.org/10.1051/j3ea/20191009

- Introduction à la simulation SystemC: module, hiérarchie de modules, vérification et test (4h de TP)
- Modélisation d'une communication comportementale: utilisation de canaux primitifs de type FIFO pour les connexions (4h de TP)
- Modélisation d'un système TLM de base: réalisation d'un routeur permettant de connecter un module initiateur à 2 blocs mémoire en SystemC-TLM (4h de TP)
- Modélisation d'un système à processeur en SystemC-TLM: réalisation d'un système comportant un processeur modelisé par son simulateur de jeu d'instructions (Instruction Set Simulator - ISS), un bloc mémoire et des périphériques (UART, etc) (8h de TP)

Introduction à la simulation SystemC

Dans la première étape programmée sur 4h de TP, l'étudiant est amené à se familiariser avec les mécanismes principaux introduits par la library SystemC nécessaires pour la modélisation de circuits et systèmes. Les notions détaillées et vues en cours, notamment celles de processus (SC_METHOD et SC_THREAD), de constructeur (SC_CTOR), et de hiérarchie entre modules sont principalement abordées dans cette étape, sur l'exemple d'un additionneur N bits à propagation de retenue (CPA). Le circuit CPA est choisi délibérément pour montrer que la modélisation SystemC pourra également être utilisée au niveau RTL (Register Transfer Level), un niveau de modélisation déjà abordé dans les matières relatives à la conception de circuits numérique en VHDL. De plus, le circuit CPA est réalisé de manière structurelle en utilisant les aditionneurs complets FA (Full Adder - FA, voir figure 1), uniquement pour montrer les mécanismes utilisés pour la réalisation d'une hiérarchie plus avancée au sein d'un module (des modules dans les modules, voir figure 2). Dans cette étape, les étudiants sont également amenés à tester les circuits et systèmes modélisés dans l'étape précédente. Pour ce faire, des modules de test sont réalisés dont l'objectif principal est de générer des stimuli permettant de vérifier le bon fonctionnement des circuits modélisés. Un extrait des résultats de simulation d'un additionneur 8 bits est présenté figure 3. Toutes les étapes de cette phase sont effectuées sur un poste Linux équipé de la suite logicielle Eclipse, des outils de développement (chaîne de compilation gcc, édition de liens, ...) et de l'outil GTKWAVE permettant de visualiser les résultats de simulation au niveau signal.

```
// adder_bit1.h
#include "systemc.h"
SC_MODULE(adder_bit1)
{
  sc_in<sc_logic> a;
  sc_in<sc_logic> b;
  sc_in<sc_logic> cin;
  sc_out<sc_logic> s;
  sc_out<sc_logic> cout;
  void addition();
  SC_CTOR(adder_bit1)
  {
    SC_METHOD(addition);
    sensitive << a << b;
    sensitive << cin;
  }
};
```

```
// adder_bit1.cpp
#include "adder_bit1.h"

void adder_bit1::addition()
{
s.write(a.read()^b.read()^cin.read());
cout.write((a.read() & b.read()) |
    (a.read() & cin.read()) |
    (b.read() & cin.read()));
}
```

Fig 1. Full adder modélisé en SystemC

```systemc
#include "systemc.h"
#include "adder_bit1.h"

SC_MODULE(adder_bit4a)
{ // ports d'entrée et de sortie
  sc_in<sc_lv<4> > a;
  sc_in<sc_lv<4> > b;
  sc_in<sc_logic> cin;
  sc_out<sc_lv<4> > s;
  sc_out<sc_logic> co;
  // signaux locaux
  sc_signal<sc_logic> aa[4],bb[4],ss[4],cc[4];

  // déclaration des additionneurs
  adder_bit1* add[4];

  // méthodes
  void addition();
  void output();

  // constructeur
  SC_HAS_PROCESS(adder_bit4a);
  adder_bit4a(sc_module_name _name, unsigned int N);
};
```

Fig 2. Additionneur N bit modélisé en SystemC (uniquement le fichier d'en-tête)

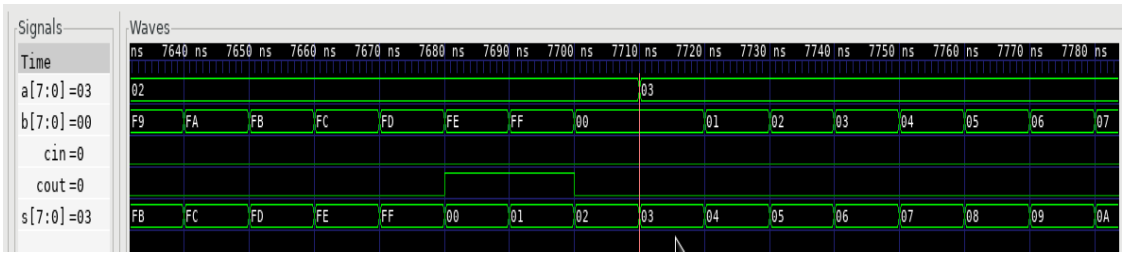

Fig 3. Extrait de résultats de simulation d'un additionneur 8 bits décrit en SystemC

Modélisation d'une communication comportementale: utilisation de canaux primitifs de type FIFO

Dans la phase précédente, les étudiants ont l'occasion de se familiariser avec les mécanismes de base de la modélisation SystemC au niveau RTL. L'objectif n'est pas d'utiliser le langage SystemC à ce niveau RTL, ni de remplacer les langages de description de matériel (VHDL ou Verilog) très utilisés à ce niveau d'abstraction, mais de l'utiliser à un niveau d'abstraction plus élevé. C'est la raison principale pour laquelle dans la deuxième étape de ce projet, les étudiants doivent modéliser une communication comportementale, en se basant sur les mécanismes de transfert de données n'existant pas dans les langages de description de matériel. Dans cette étape, programmée sur une séance de TP de 4 heures, l'étudiant est amené à décrire un système comportant un module nommé Producer, dont l'objectif principal est de générer les données périodiquement (voir fig 4), et un module nommé Consumer, dont le rôle est de consommer les données produites par le module Producer, également de manière périodique comme illustré à la figure 4 (les périodes de génération et de consommation de données sont différentes). La communication entre les deux modules est effectuée en utilisant un canal primitif de type sc_fifo permettant d'émuler un échange de données

à vitesse variable via une mémoire tampon, comme illustré à la figure 5. Comme dans la phase précédente, toute la modélisation est effectuée avec l'outil Eclipse et les outils de développement (chaîne de compilation, édition de liens, ...). Les phases de débogage et de test sont vérifiées en affichant les messages d'exécution dans la console Linux. Un extrait de tracé d'exécution est présenté à la figure 5.

```
#include "systemc.h"

SC_MODULE(producer)
{
   sc_fifo_out<int> fo;
   sc_in<bool> enable;

   void out_function();
   SC_CTOR(producer);

   int counter;

};
```

```
#include "systemc.h"

SC_MODULE(consumer)
{
  sc_fifo_in<int> fi;
  sc_out<bool> done;

  void in_function();
  SC_CTOR(consumer);

  sc_fifo<double> fifo;
  int tab[10];
  double mean;
};
```

Fig 4. Les fichiers d'en-tête des modules producer et consumer décrits en SystemC

```
#include "producer.h"
#include "consumer.h"
#include "tb_pc.h"

SC_MODULE(top_pc)
{
 producer p;
 consumer c;
 tb_pc tb;

 sc_fifo<int> fifo_loc;
 sc_signal<bool> en;
 sc_signal<bool> done;
 SC_CTOR(top_pc)
 {
  p.fo(fifo_loc);
  p.enable(en);
  c.fi(fifo_loc);
  c.done(done);
  tb.enable(en);
 }
};
```

```
Time: 320 ns Consumer computes mean value 7.5
Time: 320 ns Consumer's fifo has 87 free locations
Time: 320 ns Producer generates 15
Time: 340 ns Producer generates 16
Time: 345 ns Consumer receives 13
Time: 345 ns Consumer computes mean value 8.5
Time: 345 ns Consumer's fifo has 86 free locations
Time: 360 ns Producer generates 17
Time: 370 ns Consumer receives 14
Time: 370 ns Consumer computes mean value 9.5
Time: 370 ns Consumer's fifo has 85 free locations
Time: 380 ns Producer generates 18
Time: 395 ns Consumer receives 15
Time: 395 ns Consumer computes mean value 10.5
Time: 395 ns Consumer's fifo has 84 free locations
Time: 400 ns Producer generates 19
Time: 420 ns Consumer receives 16
Time: 420 ns Consumer computes mean value 11.5
Time: 420 ns Consumer's fifo has 83 free locations
Time: 420 ns Producer generates 20
Time: 440 ns Producer generates 21
Time: 445 ns Consumer receives 17
Time: 445 ns Consumer computes mean value 12.5
Time: 445 ns Consumer's fifo has 82 free locations
Time: 460 ns Producer generates 22
Time: 470 ns Consumer receives 18
```

Fig 5. Le fichier d'en-tête du module reliant les modules producer et consumer via un canal primitif de type sc_FIFO (gauche) et le tracé d'exécution montrant la communication entre les deux modules.

Modélisation d'un routeur en SystemC-TLM

Dans cette étape de ces TP, programmée sur une durée de 4h, les étudiants réalisent leur première modélisation SystemC au niveau transaction (TLM) [1-6]. Ils partent d'une description SystemC-TLM décrivant une communication entre un élément de

calcul de base nommé Initiator (voir la figure 6) et un bloc mémoire nommé Memory (voir la figure 7). La communication entre ces deux modules est entièrement réalisée en utilisant les sockets, le mécanisme introduit dans la library TLM. L'objectif principal de cette étape est de rajouter au système initial un autre bloc mémoire et de réaliser sa connexion à l'élément de calcul Initiator sans modifier les modules mis à disposition. Pour ce faire, la seule solution est de décrire un module intermédiaire Router dont le rôle sera d'accepter toutes les demandes d'accès aux blocs mémoire de la part de Initiator et de les transmettre au bloc mémoire correspondant. Le module Router doit comporter un socket de type simple_target_socket pour communiquer avec Initiator et deux sockets de type simple_initiator_socket pour transmettre les données aux 2 blocs mémoire. Le fichier d'en-tête du module Router est présenté à la figure 8. Comme dans la phase précédente, toute la modélisation est effectuée avec l'outil Eclipse et les outils de développement (chaîne de compilation, édition de liens, ...), tandis que le débogage et les tests sont vérifiés en affichant les messages d'exécution dans la console Linux. Un exemple de tracé d'exécution est présenté à la figure 9.

```
#include "systemc.h"
#include "tlm.h"
#include "tlm_utils/simple_target_socket.h"
SC_MODULE(Memory){
  tlm_utils::simple_target_socket<Memory> socket;
  enum {SIZE = 256};
  void b_transport(tlm::tlm_generic_payload& trans, sc_time& time);
  SC_HAS_PROCESS(Memory);
  Memory(sc_module_name _sc_name, unsigned int _id):
  sc_module(_sc_name),id(_id),socket("socket")
  {
    socket.register_b_transport(this, &Memory::b_transport);
    for(int i=0;i<SIZE;i++) mem[i]=0xAA000000 | (rand()%256);
  }
  int mem[SIZE];
  unsigned int id;
};
```

Fig 6. Le fichier d'en-tête du module Initiator

```
#include "systemc.h"
#include "tlm.h"
#include "tlm_utils/simple_initiator_socket.h"
#include "tlm_utils/simple_target_socket.h"
SC_MODULE(Router)
{
  tlm_utils::simple_target_socket<Router> target;
  tlm_utils::simple_initiator_socket<Router> initiator0;
  tlm_utils::simple_initiator_socket<Router> initiator1;
  SC_CTOR(Router):target("target"), initiator0("initiator0"),
  initiator1("initiator1")
  {
    target.register_b_transport(this,&Router::b_transport);
  }
  void b_transport(tlm::tlm_generic_payload& trans,
                   sc_core::sc_time& delay)
  {
    sc_dt::uint64 address = trans.get_address();
    sc_dt::uint64 masked_address;
    unsigned int target_mem =
            static_cast<unsigned int>( (address >> 8) & 0x3 );
    masked_address = address & 0xFF;
    trans.set_address(masked_address);
    if(target_mem==0) initiator0->b_transport(trans,delay);
    else if (target_mem==1)initiator1->b_transport(trans,delay);
   }
};
```

```
#include "systemc.h"
#include "tlm.h"
#include "tlm_utils/simple_initiator_socket.h"
#include "tlm_utils/simple_target_socket.h"
SC_MODULE(Router)
{
  tlm_utils::simple_target_socket<Router> target;
  tlm_utils::simple_initiator_socket<Router> initiator0;
  tlm_utils::simple_initiator_socket<Router> initiator1;
  SC_CTOR(Router):target("target"), initiator0("initiator0"),
  initiator1("initiator1")
  {
    target.register_b_transport(this,&Router::b_transport);
  }
  void b_transport(tlm::tlm_generic_payload& trans,
                   sc_core::sc_time& delay)
  {
    sc_dt::uint64 address = trans.get_address();
    sc_dt::uint64 masked_address;
    unsigned int target_mem =
            static_cast<unsigned int>( (address >> 8) & 0x3 );
    masked_address = address & 0xFF;
    trans.set_address(masked_address);
    if(target_mem==0) initiator0->b_transport(trans,delay);
    else if (target_mem==1)initiator1->b_transport(trans,delay);
   }
```

Fig 7. Le fichier d'en-tête du module Memory

```
#include "systemc.h"
#include "tlm.h"
#include "tlm_utils/simple_initiator_socket.h"
#include "tlm_utils/simple_target_socket.h"
SC_MODULE(Router)
{
  tlm_utils::simple_target_socket<Router> target;
  tlm_utils::simple_initiator_socket<Router> initiator0;
  tlm_utils::simple_initiator_socket<Router> initiator1;
  SC_CTOR(Router):target("target"), initiator0("initiator0"),
  initiator1("initiator1")
  {
    target.register_b_transport(this,&Router::b_transport);
  }
  void b_transport(tlm::tlm_generic_payload& trans,
                   sc_core::sc_time& delay)
  {
    sc_dt::uint64 address = trans.get_address();
    sc_dt::uint64 masked_address;
    unsigned int target_mem =
            static_cast<unsigned int>( (address >> 8) & 0x3 );
    masked_address = address & 0xFF;
    trans.set_address(masked_address);
    if(target_mem==0) initiator0->b_transport(trans,delay);
    else if (target_mem==1)initiator1->b_transport(trans,delay);
   }
};
```

Fig 8. Le fichier d'en-tête du module Routeur

```
mem 0 - reading from mem[addr]: aa000041
trans = { R, ec } , data =aa000041 at time 110 ns delay = 10 ns
mem 0 - reading from mem[addr]: aa000021
trans = { R, f0 } , data =aa000021 at time 120 ns delay = 10 ns
mem 0 - reading from mem[addr]: aa00003d
trans = { R, f4 } , data =aa00003d at time 130 ns delay = 10 ns
mem 0 - reading from mem[addr]: aa0000dc
trans = { R, f8 } , data =aa0000dc at time 140 ns delay = 10 ns
mem 0 - reading from mem[addr]: aa000087
trans = { R, fc } , data =aa000087 at time 150 ns delay = 10 ns
mem 1 - reading from mem[addr]: aa000008
trans = { R, 100 } , data =aa000008 at time 160 ns delay = 10 ns
mem 1 - reading from mem[addr]: aa000070
trans = { R, 104 } , data =aa000070 at time 170 ns delay = 10 ns
mem 1 - reading from mem[addr]: aa0000d4
trans = { R, 108 } , data =aa0000d4 at time 180 ns delay = 10 ns
mem 1 - writing to mem[addr]: ff00010c
trans = { W, 10c } , data =ff00010c at time 190 ns delay = 10 ns
```

Fig 9. Le tracé d'exécution montrant la communication entre le bloc Initiator et les deux blocs de type Memory utilisant des sockets

Modélisation d'un système à processeur en SystemC-TLM

Dans la dernière étape de ces TP, programmée sur deux séances de 4h, les étudiants sont amenés à modéliser un système à base de processeur comportant également un bloc mémoire et un périphérique de communication série (UART). Le processeur choisi pour cette modélisation est le processeur OpenRISC1000 à 32 bits à jeu d'instruction réduit. L'approche utilisée pour la modélisation de ce processeur est l'approche basée sur le simulateur de jeu d'instruction ISS (Instruction Set Simulator). Ce simulateur écrit en langage C doit être adapté (wrappé) avant d'être utilisé dans une simulation SystemC-TLM. Cette adaptation consiste à l'appeler depuis un module SystemC. De plus, pour faire un échange de données entre un module SystemC et un simulateur ISS écrit en C, le mécanisme de fonctions statiques est utilisé. Dans cette étape, les étudiants sont amenés à se familiariser avec le module permettant d'émuler le comportement du processeur OpenRISC1000. De plus, le compilateur croisé pour le processeur en question est mis à leur disposition. Il permet de générer les fichiers binaires qui seront donnés en entrée du module embarquant le simulateur ISS. Ainsi, ils peuvent voir l'exécution de ces programmes compilés sur un modèle de processeur. Dans la seconde étape de cette phase, les étudiants s'intéressent au module UART. Il s'agit d'un module basé sur un composant existant (National Semiconductor 16450). Dans cette étape, un squelette de code SystemC-TLM modélisant une grande partie du fonctionnement de l'UART en question est fourni. Les étudiants doivent décrire l'implémentation d'un nombre limité de fonctions de l'UART (les fonctions de réception et d'envoi). De surcroît, le module processeur doit également être adapté pour permettre cette communication entre le module de communication série UART et le processeur. Il s'agit d'une communication par socket, comme présenté précédemment. Dans la phase finale, un programme simple permettant d'envoyer depuis le processeur un certain nombre de messages vers l'UART est réalisé. Ceci permettra de valider le bon fonctionnement du système complet. Le schéma bloc du système complet est présenté figure 10.

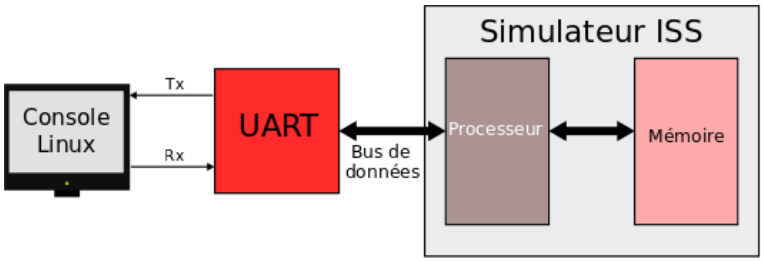

Fig. 10 Système à processeur à modéliser en SystemC-TLM [4]

III. Conclusion

Nous avons présenté une suite de travaux pratiques destinés à familiariser les étudiants du master EEA-Electronique Embarquée de l'université de Lorraine à la modélisation SystemC-TLM de systèmes à processeur. Les travaux pratiques décrits dans cet article ont partiellement eu lieu en 2018, notamment les 3 premières étapes de ces travaux (sans partie sur la modélisation d'un processeur). Le bilan de ces premières expériences (sur la première partie) est plutôt positif, malgré des difficultés rencontrées pour aller jusqu'au bout des étapes dans le temps imparti. L'étape qui pose le plus de problèmes aux étudiants est l'étape de modélisation d'un routeur selon les spécifications énoncées. Malgré une relativement bonne expérience en programmation C/C++ acquise dans le cadre d'autres matières, les étudiants perdent beaucoup de temps dans les étapes de débogage et de compilation. Les erreurs dans leur code, souvent les erreurs

de syntaxe, et le manque d'habitude de coder et compiler de manière incrémentale leur font perdre beaucoup plus de temps que prévu et ne leur permettent pas de se focaliser sur le sujet et les objectifs principaux de chaque étape. Ces travaux pratiques seront conduits dans leur globalité à la rentrée 2019/20. Un bilan plus précis pourra alors être présenté.

Références

1. John Aynsley. *Getting started with TLM-2.0: Tutorial 1 - sockets, generic payload, blocking transport*. Doulos, Developing and Delivering Know-how (2008)
2. John Aynsley. *OSCI TLM-2.0 - The Transaction Level Modeling standard of the Open SystemC Initiative (OSCI)*, Doulos (2009)
3. Brian Bailey. *ESL design and verification a prescription for electronic system-level methodology*. Amsterdam; Boston : Morgan Kaufmann (2007)
4. Jeremy Bennett. *Building a Loosely Timed SoCModel with OSCI TLM 2.0*, EMBECOSM Application Note 1. Issue 2 (2010)
5. Frank Ghenassia. *Transaction Level Modeling with SystemC*, Springer, F. Ghenassia (ed.), 1-22 (2005)
6. Matthieu Moy. *Modélisation TLM*. Technique et Science Informatiques, 33(3):285--293, 2014 (2014)