Université Côte D'Azur

Polytech Nice Sophia

Département I.S.E
Spécialité Electronique

Labs
SystemC

Fabrice MULLER
Fabrice.Muller@univ-cotedazur.fr

- 2020-2021-

1 - Modélisation d'une Mémoire

Définir une mémoire RAM synchrone dont les caractéristiques sont données ci-dessous :

- Une entrée horloge *clk* synchrone sur front descendant (type booléen)
- une entrée *enable* (type booléen)
- une entrée *rd_we* (lecture :false / écriture :true) (type booléen),
- une entrée adresse *addr* de longueur générique ADDR_SIZE (type sc_uint< ADDR_SIZE>)
- une entrée de donnée (*data_in*) et une sortie de donnée (*data_out*) de longueur générique WORD_SIZE (type sc_lv< WORD_SIZE>)
- une profondeur générique (MEM_SIZE)

Remarque : Les paramètres ADDR_SIZE, WORD_SIZE et MEM_SIZE seront des paramètres « template » de la classe (ou module) RTL_memory.

Travail Demandé

1) Créer un projet « lab1-memory_model » à partir du modèle de projet sur GitHub :

```
git clone https://github.com/fmuller-pns/systemc-vscode-project-template.git
```

2) Définir le schéma du module mémoire « RTL_memory » avec ses E/S, process
3) Décrire dans un fichier « rtl_memory.h » le module complet (pas de .cpp à cause de la généricité)
4) Définir un module de test « test_memory » qui contient le module mémoire « RTL_memory », la génération des signaux pour le tester et la trace des signaux *clk*, *en*, *rw*, *addr*, *data_in* et *data_out* dans un fichier au format VCD. Cela signifie que le module « test_memory » n'a pas d'entrée/sortie. Utiliser *sc_clock* channel pour générer l'horloge.
5) Ecrire le programme principal (fonction *sc_main()*) qui appelle le module de test « test_memory ».
6) Simuler et vérifier le comportement de la mémoire avec GTKWave.

2 - Conception d'une Mini UART

2.1 Cahier des charges

L'objectif est de concevoir une UART simplifiée dont les caractéristiques sont les suivantes :

- Trame : 8 bits de données, pas de parité, 1 bit de stop
- Transmission à 9600 bauds (TxD)
- Réception à 9600 bauds (RxD)
- Une ligne d'interruption en réception (irqRX)
- Une ligne d'interruption en émission (irqTX)

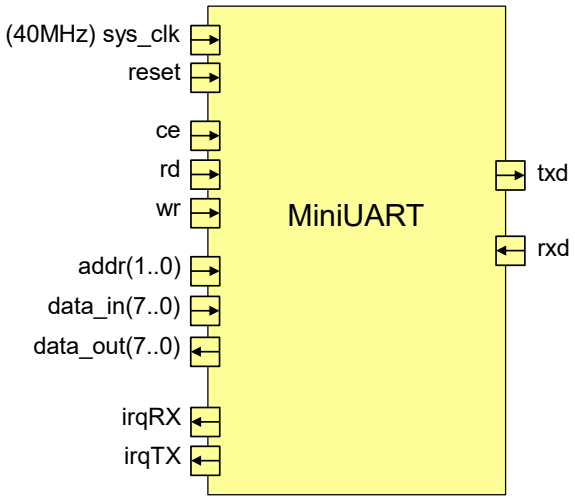

Figure 1 : Le composant miniuart.

La figure ci-dessous illustre le protocole de transmission (txd). La ligne est au repos au niveau 1. Le début d'un transfert commence toujours par un bit de start à 0. Puis, le transfert de chacun des 8 bits s'effectue du bit de poids faible jusqu'au bit de poids fort. La transmission se termine par un bit de stop à 1. La cadence de transmission est de 9600Hz, soit environ 104µs par bit.

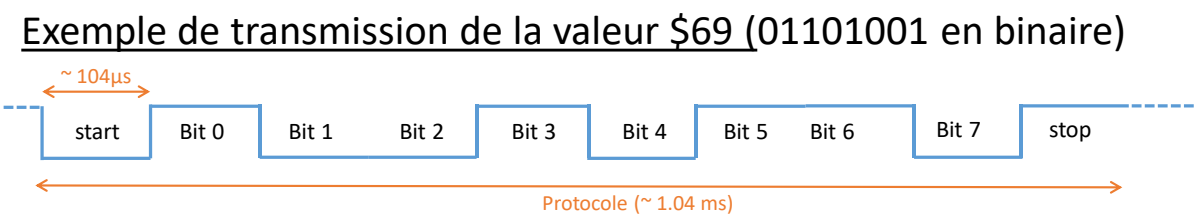

Figure 2 : le protocole RS232.

2.2 Conception de la mini UART

Nous proposons de concevoir la mini UART comme le présente la figure ci-dessous.

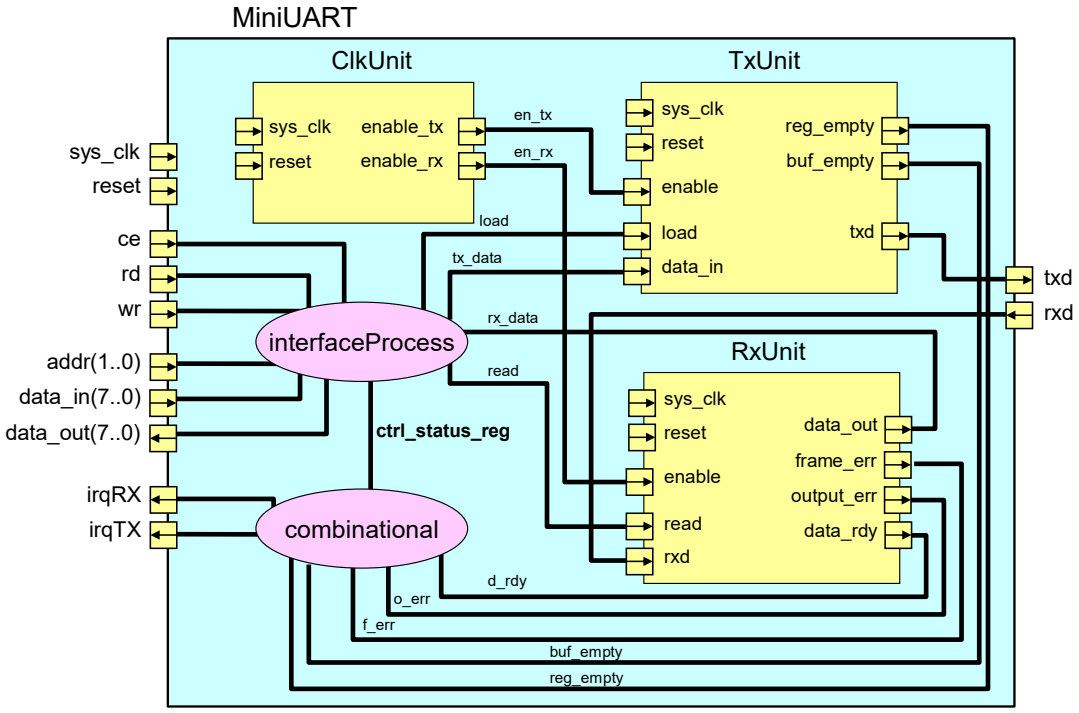

Figure 3 : Raffinement de la MiniUart.

2.2.1 Module ClkUnit

Ce module génère un signal en_tx (enable tx) à une fréquence de 9600 Hz. La précision sera de ±5Hz. Le rapport cyclique sera de 9600Hz/40MHz, soit 1 période de sys_clk (1/40MHz).

Le module génère également un signal en_rx (enable rx) à une fréquence de 16 fois 9600Hz, soit 153600Hz. La précision sera de ± 2KHz. Le rapport cyclique sera de 153600Hz/40MHz, soit 1 période de sys_clk (1/40MHz).

2.2.2 Module TxUnit

Ce module convertit une donnée parallèle en donnée série transmise sur la ligne TxD à 9600 bauds (cf. Figure 2). Le process du module se déclenche sur « sys_clk ». Le module comporte 2 principaux registres :

- Un registre « reg » qui permet de sérialiser la donnée (conversion parallèle/série). Une sortie « reg_empty » donne l'état de ce registre (true/false).
- Un registre tampon noté « buf » qui contient la prochaine donnée à sérialiser. De même, une sortie « buf_empty » donne l'état du registre (true/false).

L'entrée « enable » de période 9600Hz permet d'activer la sérialisation toute les 104µs environ.

L'entrée « load » à 1 signifie qu'une donnée est présente sur l'entrée « data_in » et qu'il faut la charger dans le registre tampon « buf », sans oublier de mettre la sortie « buf_empty » à false. Attention, le chargement du registre « reg » ne peut se faire pendant la transaction load. Il faut attendre que l'entrée « load » soit revenu à 0 pour mettre à jour le registre « reg ». Par contre,

l'écriture sur le registre « buf » est effectuée à chaque front montant de sys_clk lorsque load=1. En effet, le signal load peut durer seulement une période d'horloge de sys_clk.

2.2.3 Module RxUnit

Ce module réceptionne une trame série. La ligne série (RxD) est échantillonnée à 16 fois la fréquence de transmission. L'information est prélevée au 8ème échantillon, c'est-à-dire au milieu de chaque bit. La figure suivante illustre le principe de capture.

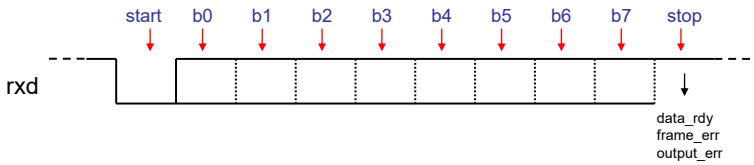

Figure 4 : Principe de capture du bit en réception.

Lors de la réception du bit de stop, différents signaux sont mis à jour :

- Le signal data_rdy (data ready in buffer) est mis à false dès la lecture dans le registre « data_out » (signal *read*).
- Le signal frame_err égale à true si le bit de stop détecté est égale à 0 sinon frame_err = false. Le signal data_rdy reste à faux.
- Le signal output_err égale à true si le signal data_rdy est déjà positionné vrai. Cela signifie que la donnée précédente n'a pas encore été lue et qu'elle a été écrasée.

Notons que

- les signaux frame_err, output_err et data_rdy sont mis à false lors d'un accès en lecture, c'est-à-dire le signal *read* à 1.
- le module comporte au moins un registre « shift_reg » qui convertit la donnée en parallèle. Lors du stop, le registre « shift_reg » est transféré dans le registre « data_out » qui est accessible à partir de la sortie « data_out ».

2.2.4 Les processes du module MiniUART

Ce module renferme un registre « ctrl_status_reg » qui permet de connaître l'état de l'UART. Le format de ce registre est le suivant :

7	6	5	4	3	2	1	0
-	-	-	-	-	empty	frameErr	OutputErr

Process « interfaceProcess »

Le process « interfaceProcess » permet de gérer les signaux load, tx_data, rx_data et read. Les signaux sont spécifiés à l'aide de table de vérité.

Les signaux ce, wr et rd seront du type std_logic alors que addr, data_in et data_out sera du type sc_lv<…>.

La table de vérité des signaux load et tx_data :

ce	wr	addr(1..0)	load	tx_data
0	X	XX	0	0…0
1	0	XX	0	0…0

1	1	00	1	data_in
1	1	X1 ou 1X	0	0…0

La table de vérité des signaux read et data_out :

ce	rd	addr(1..0)	read	data_out
0	X	XX	0	Z…Z
1	0	XX	0	Z…Z
1	1	00	1	rx_data
1	1	01	0	ctrl_status_reg
1	1	1X	0	Z…Z

Process « combinationalProcess »

Le process « combinationalProcess » génère les interruptions en émission/réception ainsi que la mise à jour la concaténation de signaux pour former le « ctrl_status_reg »

L'interruption en émission (irqTX) est true lorsque le signal « buf_empty » est vrai et que le signal « reg_empty » est faux.

L'interruption en réception (irqRX) est true lorsque le signal « d_rdy » est à true, c'est-à-dire qu'une donnée est présente dans le buffer du module TxUnit. Le signal « irqRX » est à false autrement.

Le registre « ctrl_status_reg » est juste une concaténation des signaux buf_empty, f_err et o_err.

2.3 Travail demandé

Tout d'abord, créer un projet « lab2_miniuart » à partir du modèle de projet sur GitHub :

```
git clone https://github.com/fmuller-pns/systemc-vscode-project-template.git
```

2.3.1 Le module ClkUnit

- Ecrire le module ClkUnit (*clk_unit.h* et *clk_unit.cpp*). Le fichier *clk_unit.h* est à compléter.
- Pour tester ce module, compléter le fichier *test_clk_unit.cpp* qui sera appelé depuis la fonction *sc_main()* du fichier *main.c*. Dans ce fichier *main.c*, nous avons
  - Un appel au fichier header contient toutes les méthodes de tests qui seront écrits (*benches.h*).
  - Les constantes sous forme de macro qui vont permettre au fur et à mesure les scénarios.

2.3.2 Le module TxUnit

- Ecrire le module TxUnit (*tx_unit.h* et *tx_unit.cpp*)
- Compléter le fichier *test_tx_unit.cpp*.

2.3.3 Le module RxUnit

- Ecrire le module RxUnit (*rx_unit.h* et *rx_unit.cpp*)
- Copier le fichier *test_tx_unit.cpp* et l'appeler *test_rx_unit.cpp.*
- Compléter le fichier *test_rx_unit.cpp* et en y ajoutant l'instance RxUnit et les connexions nécessaires. Un signal txd_rxd permettra de connecter la sortie txd à l'entrée rxd.

2.3.4 Le module MiniUart

- Ecrire le module qui est composé de 3 instances et 2 processes (SC_METHOD) en complétant le fichier *mini_uart.h* et en créant le fichier *mini_uart.cpp*.
- Pour tester la miniuart, nous allons tout d'abord écrire un module Testbench (fichiers *testbench.h* et *testbench.cpp* qui seront à compléter). Le test proposé est d'envoyer un message sous interruption et de le réceptionner sous interruption. La figure ci-dessous illustre le principe de test.

  Explication des étapes (cf. Figure 5) :

  1. Le process « main » écrit le premier caractère du message.
  2. Le caractère est transmis (L'interruption *irqTX* peut être déclenchée)
  3. Le caractère est reçu.
  4. L'interruption *irqTX* est déclenchée et la routine d'interruption *isr_tx()* écrit le caractère suivant.
  5. L'interruption *irqRX* est aussi déclenchée.
  6. La routine d'interruption *isr_rx()* lit le caractère et l'affiche.

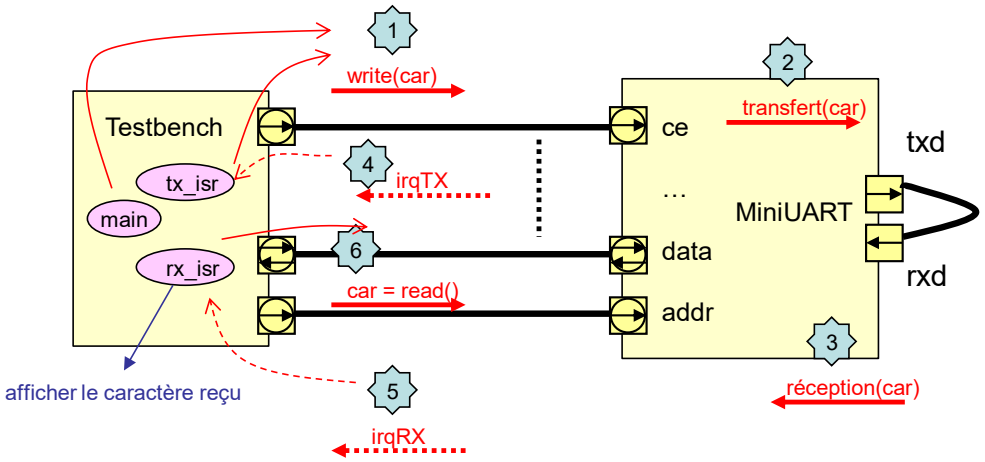

Figure 5 : Scénario complet de test.

  - Comprendre et compléter le code dans le module Testbench en utilisant les 3 méthodes d'aides (Helper Functions).
    - resetTest() : active le signal reset pendant 20 cycles horloges
    - write() : écriture d'un octet sur le bus
    - read() : lecture d'un octet sur le bus
- Ecrire les processes (au choix SC_THREAD ou SC_METHOD ?) du module Testbench.
  – Utilisez sc_signal_resolved (1 bit) et sc_signal_rv<N> (N bits) pour relier le Testbench et la MiniUart.
- Ecrire la méthode *test_miniuart()* (fichier *test_mini_uart.cpp* à compléter) qui sera appelée à partir du *sc_main()*.
- Tester en simulation. Problème ?

  Au démarrage, le process *main()* écrit une valeur dans le buffer. Cependant, une interruption est déclenchée en même temps que l'écriture du *main* et une seconde écriture réalisée par le process *isr_tx()* est faite en parallèle. Cela engendre un conflit !

Comme solution, utiliser un mutex (sc_mutex) qui permet d'obtenir une seule exécution de lecture/écriture à la fois.

3 - Canal Primitif (Primitive Channel)

3.1 Présentation

Nous voulons réaliser un bus de niveau transactionnel entre un maitre (Master) et un esclave (Slave). Pour cela, nous allons définir un canal primitif. Ce type de canal permet de connecter des modules entre eux et de fournir un ensemble de services pour communiquer.

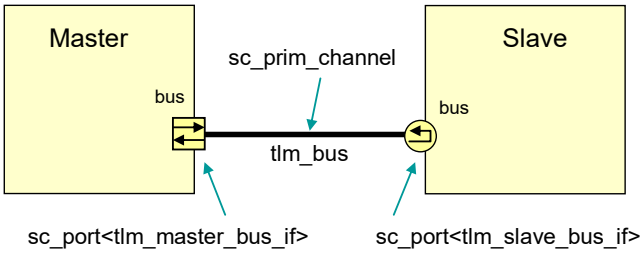

Figure 6 : Présentation du bus TLM.

3.2 Les interfaces

L'interface est la partie visible du canal qui fournit des fonctions ou méthodes qui peuvent être différentes du coté maitre et du coté esclave. Les méthodes, coté maitre, (*tlm_master_bus_if*) sont :

- Service 1 : Lecture d'une donnée à une adresse,
```
int readToSlave(unsigned int addr_)
```
- Service 2 : Ecriture d'une donnée à une adresse,
```
void writeToSlave(unsigned int addr_, int data_)
```
- Service 3 : 2 lignes d'interruptions (int0 et int1)
```
const sc_event& irq0Event() const
const sc_event& irq1Event() const
```

Du coté esclave (tlm_slave_bus_if), nous avons besoin des services suivants :

- Service 1 & 2 : Un événement qui permet d'être prévenu d'une lecture ou d'une écriture
```
const sc_event& rwEvent() const
```
- Service 1 & 2 : Des fonctions qui permettent de récupérer le type d'accès (lecture ou écriture), l'adresse
```
const bool isRead() const
const unsigned int getAddress() const
const int getData() const
```
- Service 1 : Une fonction qui permet de s'acquitter et d'envoyer la donnée au maitre dans le cas d'une lecture.
```
void sendDataToMaster(int data_)
```
- Service 2 : Une fonction qui permet de s'acquitter au maitre dans le cas d'une écriture.
```
void sendAckToMaster()
```
- Service 3 : Récupérer la donnée dans le cas d'une écriture par le maitre
```
const int getData() const
```
- Service 4 : Deux fonctions qui envoient respectivement l'interruption int0 et int1
```
void irq0Notify()
void irq1Notify()
```

Travail demandé

Les services sont décrits dans le fichier *services.txt*.

1) Ecrire l'interface tlm_master_bus_if (*fichier tlm_master_bus_if.h*)
2) Ecrire l'interface tlm_slave_bus_if (*fichier tlm_slave_bus_if.h*)

3.3 L'implémentation de la communication primitive tlm_bus

La lecture et l'écriture utilise un événement *rwToSlaveEv* (renvoyé par la méthode *rwEvent()*) pour prévenir l'esclave qu'une demande d'accès en lecture ou écriture est en cours. Le bus ne peut effectuer qu'un seul accès à la fois.

L'esclave est prévenu qu'un accès en lecture/écriture a lieu grâce à un événement (*rwToSlaveEv*). Lors de la réception de cet événement, 2 cas peuvent se produire :

- C'est une écriture. La méthode *isRead()* renvoi false. Des méthodes permettent de récupérer l'adresse (*getAddress()*) et la donnée (*getData()*). L'esclave retourne un accusé de réception pour s'aquitter de l'écriture en utilisant la méthode *sendAckToMaster()*.
- C'est une lecture. La méthode *isRead()* renvoi true. Seule la méthode de récupération de l'adresse est utile. L'esclave ira chercher la donnée correspondant à l'adresse et enverra la donnée en utilisant le méthode *sendDataToMaster()*.

De plus, l'esclave peut envoyer à travers le bus 2 interruptions (*irq0Notify()* et *irq1Notify()*) qui correspondent à 2 événements. En résumé, il faut :

- Un événement *rwToSlaveEv* (le maitre effectue un accès en lecture/écriture),
- Un événement *ackToMasterEv* (l'esclave envoi la donnée vers le maitre pour une lecture ou s'acquitte pour d'une écriture),
- Deux événements pour les interruptions *irq0*, *irq1*,
- Des champs pour stocker la donnée, l'adresse et le type d'accès,
- Un mutex noté *mutex* qui permet de contraindre un seul accès au bus à la fois.

Travail demandé

1) Ecrire la communication primitive (*tlm_bus.h* et *tlm_bus.cpp*) qui hérite de *sc_prim_channel* et des deux interfaces *tlm_master_bus_if* et *tlm_slave_bus_if*. Le fichier *tlm_bus.h* est à compléter.

3.4 Test

Pour tester le bus, nous allons écrire le maitre (Master) et un module de test (Slave) comme le montre la Figure 6.

Tout d'abord, créer un projet « lab3_primitive_channel » à partir du modèle de projet sur GitHub :

```
git clone https://github.com/fmuller-pns/systemc-vscode-project-template.git
```

Travail demandé

1) Ecrire le maitre (*master_module.h* à compléter) qui comprend 3 processus :
   - Un processus qui effectue des lectures/écritures,
   - Un processus qui réagit sur l'interruption *irq0* et affiche un message,
   - Un processus qui réagit sur l'interruption *irq1* et affiche un message.

2) Ecrire l'esclave (slave_*module.h* à compléter) qui modélise une mémoire. Cet esclave comprend 2 processus :
   - Un processus qui réagit aux lectures/écritures.
   - Un processus qui envoie des interruptions sur les lignes *irq0* et *irq1*.

4 - Transactor (Channel)

4.1 Présentation

Nous voulons utiliser la MiniUart dont le bus de communication est de niveau cycle à partir d'un bus de niveau TLM (Transaction Level Modeling), « tlm_bus » développé dans le Lab3. Ceci est possible si une transformation est réalisée pour passer du niveau TLM au niveau cycle et vice-versa. Ce module s'appelle un « Transactor ».

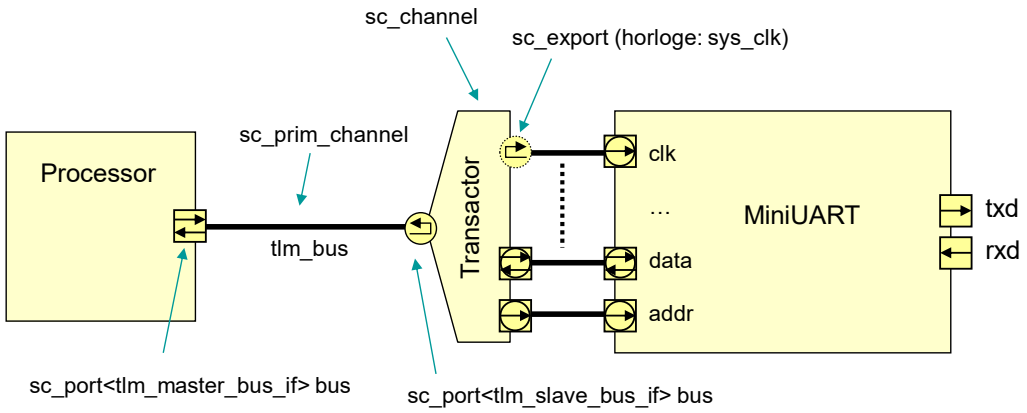

Figure 7 : Architecture.

Un transactor est un « channel » (classe *sc_channel*) qui est juste une redéfinition de la classe *sc_module*. Ainsi, les caractéristiques d'un « channel » sont les mêmes qu'un « module »: déclaration des ports d'entrées/sorties, communications hiérarchiques, sous modules, processus …

4.2 Conception du Transactor

Nous rappelons que l'objectif du Channel « Transactor » à concevoir est de transformer ou de traduire le passage d'une action (lecture, écriture, interruption) au niveau TLM vers le niveau cycle et vice-versa.

4.2.1 Introduction

4.2.1.a Les entrées/sorties

Le challen « Transactor » a un port composé de deux ensembles :

- Le premier ensemble d'entrées/sorties qui sont connectés au bus niveau cycle (RTL : Register Transfert Level).

  Attention : On remarque que la sortie horloge *sys_clk* est un port de type « sc_export ».
- Le second ensemble est un port appelé *bus* de type « tlm_slave_bus_if ».

4.2.1.b Les processus

Identifions les transformations pour déterminer le nombre de processus à écrire :

- Le bus de niveau cycle gère 2 interruptions *irq0* et *irq1* de type « signal ». Il suffit de créer un processus *isr0()* et *isr1()* (thread ou method ?) pour chaque interruption déclenchable sur fronts montants.

- Du côté du bus TLM, le maitre (processeur) peut envoyer un ordre d'écriture ou de lecture. La traduction d'un cycle de lecture/écriture a déjà été présentée dans le Lab2 MiniUart. Un processus (thread ou method ?) en attente sur l'événement fourni par la méthode *rwEvent()* semble être une solution.

4.2.2 Travail demandé

Tout d'abord, créer un projet « lab2_miniuart » à partir du modèle de projet sur GitHub :

- Dupliquer le lab « lab2_miniuart » en le nommant « lab4_transactor »
- Copier les fichiers utiles (tlm_xxx.*) du lab « lab3_primitive_channel » dans le nouveau lab « lab4_transactor ».
- Compléter le fichier fourni « transactor.h ».
- Ecrire le fichier « transactor.cpp ».

4.3 Test du Transactor

La figure ci-dessous présente le principe pour tester le Transactor vu dans le Lab2 MiniUart. En effet, ce test a déjà été partiellement écrit lors du Testbench de la MiniUart.

Nous allons écrire un module « transactor_bench » qui englobe le processeur, le transactor et la MiniUART. Nous appellerons le module « transactor_bench » à partir du fichier *test_transactor.cpp* fourni.

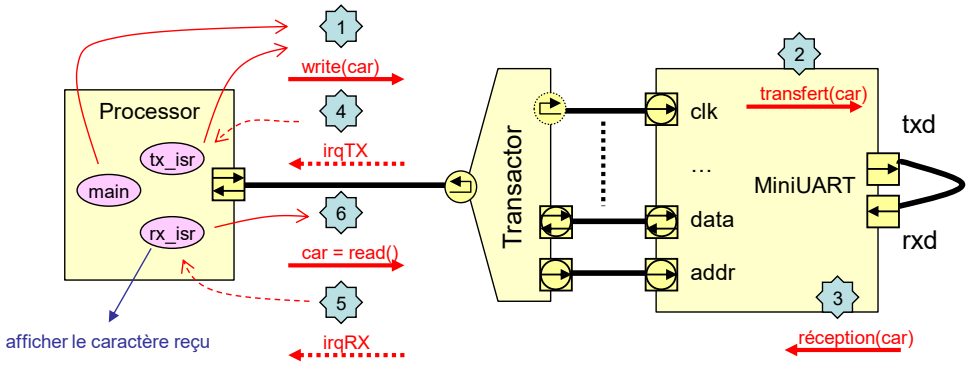

Figure 8 : Principe de test du Transactor.

- Compléter le fichier fourni « transactor_bench.h ».
- Ecrire la fonction *trace()* dans un nouveau fichier « test_transactor.cpp ».
- Copier le fichier fourni « test_transactor.cpp »
- Compléter le fichier « main.cpp » en ajoutant une constante #define TEST_TRANSACTOR 4 ainsi que l'appel à la fonction *test_transactor()* dans le *sc_main()*.
- Tester le transactor