INF8500 – Conception et vérification des systèmes embarqués

Automne 2021

Laboratoire #1

Modélisation SystemC

Objectifs

L'objectif de ce laboratoire est de comprendre la méthodologie de conception pour systèmes embarqués basée sur la librairie SystemC.

Plus précisément, les objectifs du laboratoire sont :

- S'initier à la librairie SystemC
- Connaître les différents niveaux d'abstraction offerts par SystemC
- Mettre en pratique la modélisation les différents composants d'un système (processeur, bus, mémoire, etc.)
- Se familiariser avec le modèle transactionnel TLM, plus particulièrement simple bus.

Partie 1 : Modélisation à haut niveau

La première partie du laboratoire consiste à développer un système embarqué à haut niveau d'abstraction. Ce système est composé d'un processeur (*processor*) et de trois coprocesseurs (*coproX*) interconnectés via le module *interconnexion.* Le processeur génère un paquet qui est acheminé à un coprocesseur pour être traité. Une fois le traitement terminé, le coprocesseur met le résultat dans un paquet et le renvoie au processeur.

Pour vous aider et tel qu'illustré à la figure 1, l'architecture du projet vous est fournie.

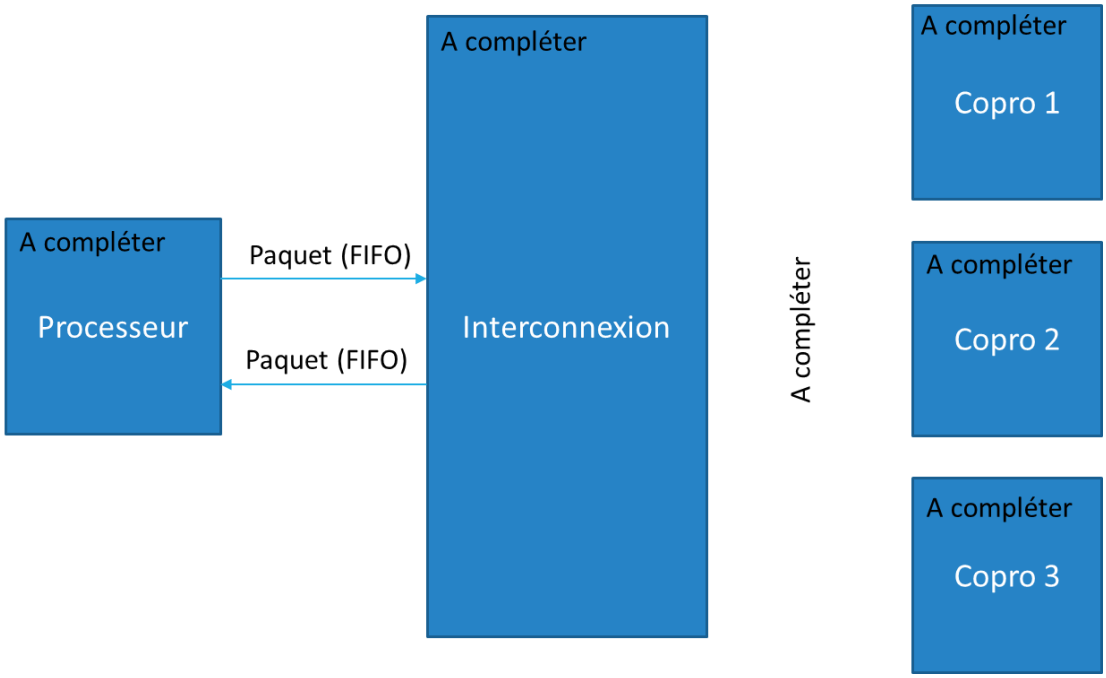

Figure 1

Définitions

Le paquet :

Le paquet est un objet constitué de trois attributs :

- Un identifiant (numéro du paquet créé) : de type unsigned int.
- Une adresse de destination (1 à 3 selon le coprocesseur ciblé) : de type unsigned int.
- Une charge utile (contenu à livrer aux coprocesseurs pour traitement) : tableau de 6 unsigned int.

Plusieurs constructeurs et méthodes sont à votre disposition. Pour plus de renseignements, reportez-vous au code source du fichier packet.cpp et packet.h.

Le processeur :

Le module *processor* répète 8 fois cette boucle :

1. Génère un paquet qui contient une charge utile aléatoire. Chaque élément est compris entre 0 et 999. Cela se fait via l'appel au constructeur (voir processor.cpp). L'adresse du coprocesseur destinataire est aussi générée de manière aléatoire.
2. Envoie le paquet au copro destinataire pour traitement.
3. Effectue le même traitement que le coprocesseur pour comparaison ultérieure.
4. Réceptionne le paquet traité par le coprocesseur (attente bloquante).

5. Compare le paquet réceptionné au traitement effectué en étape 3 et affiche un message dans la console.

Pour plus de renseignements, reportez-vous au code source du fichier processor.cpp et processor.h.

L'interconnexion :
Le module *interconnexion* fait le lien entre le processeur et les 3 coprocesseurs. En boucle, il :

1. Réceptionne un paquet envoyé par le processeur (attente bloquante).
2. Lis l'adresse du destinataire du paquet (coprocesseur 1 à 3).
3. Envoie le paquet à ce destinataire.
4. Réceptionne le paquet traité par le coprocesseur (attente bloquante).
5. Envoie ce paquet traité au processeur.

Pour plus de renseignements, reportez-vous au code source du fichier interconnexion.cpp et interconnexion.h

Les coprocesseurs :
Les 3 coprocesseurs fonctionnent de manière identique :

1. Réception d'un paquet venant de l'interconnexion (attente bloquante).
2. Traitement du paquet. Le traitement est évoqué plus en détail dans la section suivante.
3. Renvoi du paquet traité au processeur.

Pour plus de renseignements, reportez-vous au code source du fichier copro1.cpp et copro1.h

Travail à réaliser pour la partie 1

Le processeur :
Le code du processeur est partiellement complété. Il vous reste à :

- Effectuer le traitement identique au coprocesseur (voir coprocesseur).
- Afficher un message qui indique si le traitement s'est bien passé.

Le processeur est le seul module qui peut afficher un message à la console dans le code que vous rendrez.

L'interconnexion :
La structure de base de l'interconnexion vous est fournie. Il vous reste à coder les interfaces avec le processeur et les 3 coprocesseurs, et la lecture de l'adresse contenue dans le paquet.

Le coprocesseur 1:
Le code du coprocesseur 1 est intégralement à compléter. Le traitement du coprocesseur 1 est un tri à bulles du plus petit au plus grand. Le coprocesseur 1 doit uniquement utiliser des *sc_signal* pour communiquer avec l'interconnexion (pas de *sc_buffer* ou *sc_fifo*).

Le coprocesseur 2 :
Le code du coprocesseur 2 est intégralement à compléter. Le traitement du coprocesseur 2 est un tri à bulles du plus grand au plus petit. Le coprocesseur 2 doit uniquement utiliser des *sc_signal et sc_buffer* pour communiquer avec l'interconnexion (pas de *sc_fifo*).

Le coprocesseur 3 :
Le code du coprocesseur 3 est intégralement à compléter. Le traitement du coprocesseur 3 est un tri à bulles du plus petit au plus grand. Le coprocesseur 3 doit uniquement utiliser des *sc_fifo* pour communiquer avec l'interconnexion (pas de *sc_signal et sc_buffer*).

Partie 2 Raffinement de l'interconnexion en utilisant simple bus

L'objectif de cette partie est de remplacer le module *interconnexion* de la partie 1 par Simple Bus et un ensemble d'adaptateurs. La figure 2 présente le résultat.

Vous pouvez vous baser sur les exemples de SystemC dans le dossier suivant :
systemc-2.3.3\examples\sysc\simple_bus
Deux fichiers sont intéressants :
Simple_bus_master_blocking et simple_bus_slow_mem.

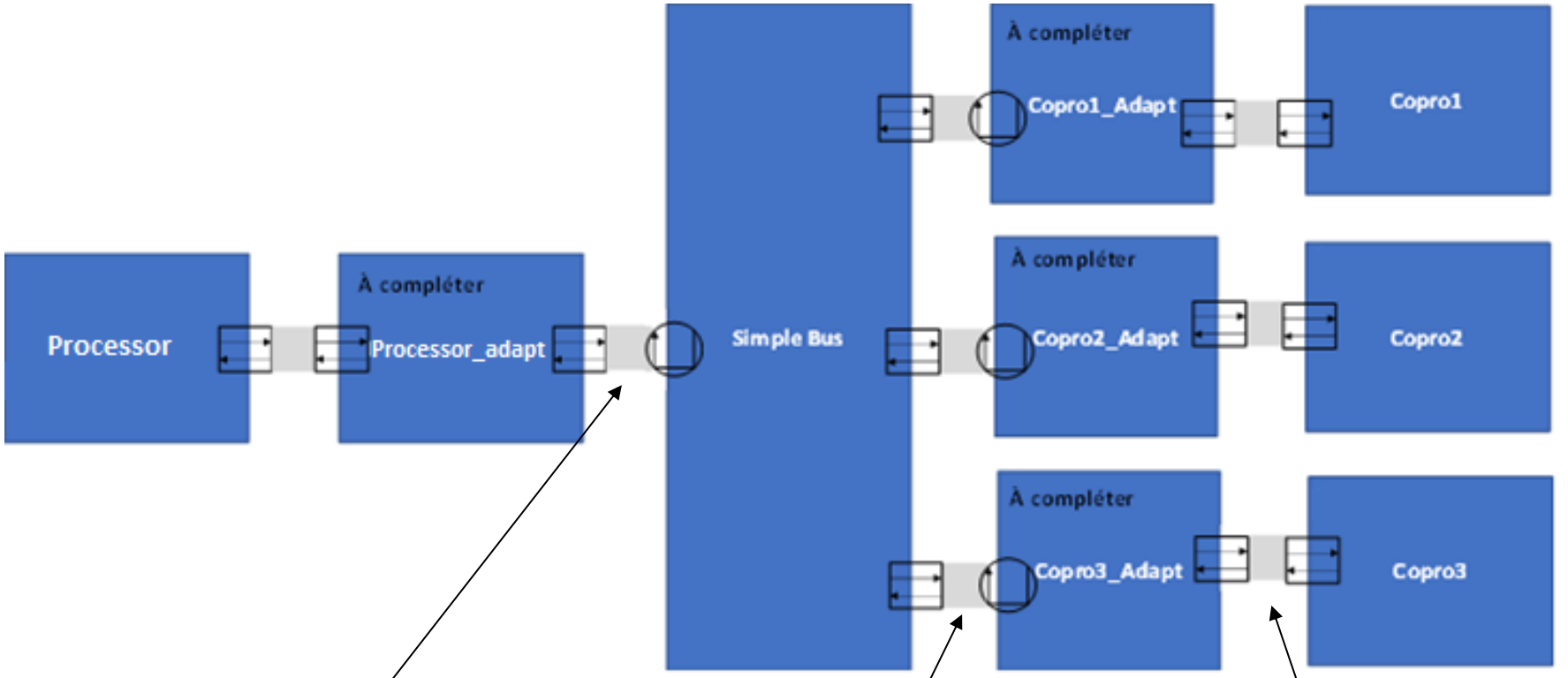

Figure 2

Méthodes read/write définies dans l'interface de Simple Bus.

PacketGen_Adapt a juste à appeler bus_port->burst_write(...) et bus_port->burst_read(...)

Méthodes read/write à définir dans coproX_adapt

Protocole maison.

Vous pouvez reprendre le même qu'entre l'interconnexion et le copro de la partie 1

Définitions :

Le bus :
Comme son nom l'indique, *simple bus* modélise un bus c'est-à-dire un canal de communication permettant de relier ensemble différents modules. Ces différents modules se divisent en deux catégories, les modules maîtres (initiateurs) et les modules esclaves (cibles). Les modules maîtres peuvent envoyer des requêtes de lecture et d'écriture tandis que les modules esclaves ne font que répondre à ces requêtes.

Adaptateurs (Wrappers) *processor_adapt*:
Ici le seul maître *processor* (mais on pourrait en avoir plus d'un puisqu'on a un arbitre), possède un adaptateur (*processor_adapt*) qui fait l'interface entre le processeur et Simple bus. Afin de pouvoir créer des requêtes sur le bus, le maître doit posséder un port *sc_port<simple_bus_blocking_if>* par lequel il pourra accéder aux méthodes de l'interface bloquante du bus. Ces méthodes sont définies dans le fichier *simple_bus.cpp*.

Les modules maîtres (module) : `simple_bus_blocking_if.h`	
Interface	Définition

`Virtual simple_bus_status burst_read(`
`  unsigned int unique_priority`
`, int *data`
`, unsigned int start_address`
`, unsigned int length = 1`
`, bool lock = false) = 0;`
	Cette méthode vous permet d'effectuer une lecture bloquante de taille `length`*4 octets vers `*data`. Les données sont lues depuis la mémoire d'un esclave à l'adresse de départ `start_address`. `unique_priority` est la priorité de ce maître et la variable `lock` permet d'empêcher des requêtes de plus hautes priorités de prendre la main

`virtual simple_bus_status burst_write(`
`  unsigned int unique_priority`
`, int *data`
`, unsigned int start_address`
`, unsigned int length = 1`
`, bool lock = false) = 0;`
	Cette méthode permet l'écriture bloquante de la donnée `*data` de taille `length`*4 octets à l'adresse `start_address` d'un esclave.

Les adaptateurs Copro, *Copro1_Adapt, Copro2_Adapt et Copro3_Adapt :*

Les esclaves *copro1*, *copro2* et *copro3* possèdent chacun un adaptateur. Ces derniers héritent de l'interface *simple_bus_slave_if* esclave lors de la déclaration de classe.

Les modules esclaves : simple_bus_slave_if.h	
Interface	Définition

`Virtual simple_bus_status read(`
`int *data`
`, unsigned int address) = 0;`
	Cette méthode permet de spécifier la méthode de lecture sur un périphérique connecté sur le bus. Cette méthode est à implémenter.

`virtual simple_bus_status write(`
`int *data`
`, unsigned int address) = 0;`
	Cette méthode permet de spécifier la méthode d'écriture sur un périphérique connecté sur le bus. Cette méthode est à implémenter. Elle est dépendante du périphérique utilisé.

`virtual unsigned int start_address()`	Renvoie l'adresse de départ de la mémoire du

	module (à implémenter)
`virtual unsigned int end_address()`	Renvoie l'adresse de fin e la mémoire du module (à implémenter)

Adressage et Simple bus
Simple Bus fonctionne avec un mécanisme d'adressage pour envoyer le paquet au bon destinataire. Chaque adaptateur de coprocesseur obtient une plage d'adresse lors de l'instanciation. Chaque adaptateur doit avoir une plage d'adresse assez grande pour contenir un la charge utile d'un paquet (24 octets). Il faut également faire attention à ne pas faire déborder la mémoire, car elle n'est pas circulaire ; ni écrire un paquet sur une plage d'adresse qui est à cheval entre deux coprocesseurs. Enfin, les plages d'adresse et l'adresse à laquelle on écrit à un coprocesseur doivent être des multiples de 4, car simple bus utilise des mots de 32 bits (4*8 bits).

Nous vous conseillons d'adresser les coprocesseurs de 0 à 287, soit 96 octets par coprocesseur. Ainsi, pour écrire au coprocesseur 1, il est possible décrire aux adresses: 0, 4, 8, 12, ..., 72 (inclus).

Pour générer de tels paquets, nous vous conseillons de tirer deux nombres aléatoires dans le processeur : le premier nombre détermine le numéro du coprocesseur, et le second indique l'offset en mémoire.

Travail à réaliser pour la partie 2 :

Le processeur
Reprenez le processeur de la partie 1. Il faudra changer les adresses de destination des paquets.

Adaptateur (Wrapper) *processor_adapt*:
L'adaptateur récupère un paquet envoyé par le processeur, et envoie la payload au coprocesseur via Simple bus. L'adapteur doit ensuite effectuer une lecture à la même adresse que l'écriture pour récupérer la payload traitée. Pour ce laboratoire, vous travaillerez avec *burst_write et burst_read.*

Adaptateurs *Copro1_Adapt, Copro2_Adapt et Copro3_Adapt :*

Méthode write (processeur vers coprocesseur)
Comportement attendu : lorsque l'adaptateur du processeur écrit une payload avec burst_write bloquant, la fonction write de l'interface slave (définie dans coprox_adapt_slave.cpp) doit être appelée 7 fois. Une première fois pour le wait state (délai du slave à s'activer[1]) et 6 autre fois pour l'écriture de chaque mot (7 cycles pour le burst complet). Vous pouvez jouer avec la wait_loop et le paramètre nr_wait_states des adaptateurs pour arriver à ce comportement.

[1] Le wait state est paramétrable via *m_nr_wait_states*

Vous devez utiliser un compteur pour savoir quel mot de la payload est réceptionné. Sauvegardez l'adresse lors de la réception du premier mot. Lorsque le 6^{e} mot est réceptionné, le paquet peut être reconstruit (excepté l'identifiant, mais ce n'est pas grave), et être envoyé au coprocesseur. Puis attendre que le coprocesseur renvoie le paquet traité. Enfin, vous pouvez écrire payload traitée dans la mémoire à l'adresse sauvegardée précédemment. C'est à cette adresse que le maitre va venir récupérer la payload traitée.

Méthode read (coprocesseur vers processeur)
Similaire à l'exemple de SystemC.

Ces deux méthodes doivent s'exécuter en un seul delta-cycle (exception pour la communication avec le coprocesseur). À l'issue de cette exécution, elles doivent renvoyer une de ces trois valeurs :

• SIMPLE_BUS_OK : La requête s'est terminée avec succès. Le maître peut continuer son exécution.

• SIMPLE_BUS_WAIT : La requête se poursuit, la fonction *write* sera réexécutée au cycle suivant. Le maître reste bloqué.

• SIMPLE_BUS_ERROR : La requête s'est soldée par un échec. Le maître peut continuer son exécution s'il le désire.

Chaque adaptateur devra posséder une mémoire permettant de contenir une payload complète (soit un *unsigned int*[6]). Voir la section adressage ci-dessus.

Les coprocesseurs
Identiques à la partie 1.

Rapport, questions et date de remise:

Le rapport et le code sont à rendre sur moodle avant le dimanche 10 octobre 23h55.
Nous vous remercions d'avance de remettre sous moodle un fichier .zip avec la structure suivante (cela nous facilite grandement la tâche):
Lab1_INF8500_A20_matricule1_matricule2.zip

- Votre rapport Pdf
- Partie 1 (dossier)
    - Src (dossier)
        - Tous les fichiers sources de la partie 1 (.cpp et .h)
    - Makefile (disponible sur Moodle)
- Partie 2 (dossier)
    - Src (dossier)
        - Tous les fichiers sources de la partie 2 (.cpp et .h)
    - Makefile (disponible sur Moodle)

Guy Bois, prof. GIGL
Responsable du INF8500

Julien Posso
Chargé de laboratoire INF8500