HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD

Prof. Yann Thoma

Laboratoire VSE

semestre d'automne 2023 - 2024

Laboratoire de vérification
Système complet

Introduction

Nous désirons mettre en place un calculateur sur FPGA, accessible depuis un PC. Il s'agira d'un calculateur sur bus avalon, et nous visons à l'embarquer dans un système embarqué [1]. Sur celui-ci, un Linux embarqué tournera sur un processeur ARM, et au final nous désirons avoir une application graphique qui envoie des commandes au système embarqué pour *accélérer* le traitement sur FPGA [2].
Le travail va être décomposé en plusieurs parties :

1. Vérification du composant VHDL à partir de code C/C++
2. Vérification du composant VHDL depuis une application graphique Qt
3. Intégration de différents éléments en évitant la redondance de code

Etape 1

Le calculateur effectue l'opération suivante :

$$result = a^N + b * c$$

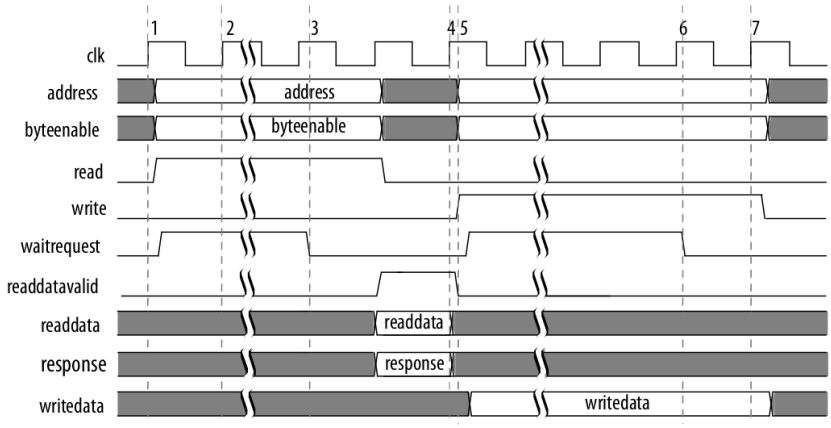
a, b et c seront des entrées qui seront envoyées via des écriture sur le bus alavon, et N un paramètre générique. Un second paramètre générique, `DATASIZE`, permet de spécifier la taille des opérandes et du résultat. Le lancement du calcul se fait via une écriture sur le bus et résultat est récupéré via une lecture sur le bus.

Les adresses des différents registres sont les suivantes :

1. Oui, il y a un peu trop d'embarquement dans cette phrase.
2. Oui, on va avoir de la peine à créer un vrai accélérateur pour ce travail, mais bon, la vie est parfois ainsi faite.

Addresse	bits	Nom	Description
0x0	`DATASIZE-1:0`	`a`	Entrée A
0x1	`DATASIZE-1:0`	`b`	Entrée B
0x2	`DATASIZE-1:0`	`c`	Entrée C
0x3	`DATASIZE-1:0`	`result`	Résultat
0x4	0	`cmd`	Lancement du calcul
0x5	0	`status`	Statut

Une écriture dans le bit 0 du registre `cmd` lance un calcul. Durant le calcul, le bit 0 du registre `status` est à 0, et il passe à 1 lorsque le résultat est disponible dans le registre `result`. Il garde sa valeur de 1 jusqu'au lancement du prochain calcul.
Les lectures/écritures sur le bus avalon correspondent à l'exemple suivant :



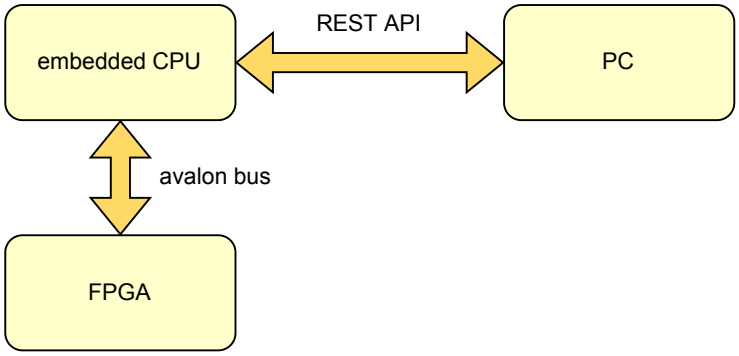
Pour cette première étape, nous allons interfacer un banc de test avec du C, via le DPI de SystemVerilog. Le banc de test est donc en SystemVerilog, mais l'idée est que les scénarios puissent ensuite être écrits en C.
Reprenez le code fourni dans le repo Git et observez les différents composants (SystemVerilog, C).
Le concept est ici de faire que les scénarios C puissent appeler les méthodes bas niveau pour l'accès au composant, à savoir l'écriture et la lecture sur bus avalon.
Dans le testbench, commencez par écrire les fonctions d'écriture et de lecture et vérifiez qu'elles fonctionnent.
Ensuite, attelez-vous à modifier le fichier C pour pouvoir tester différents scénarios.
Quelques pistes intéressantes pour le développement :

1. Créer une fonction pour faire un calcul en C
2. Créer une fonction de référence en C, pour pouvoir facilement tester des cas différents
3. Créer une interface pour le bus avalon en SystemVerilog et l'utiliser pour la connectique
4. Placer les méthodes `avalon_write()` et `avalon_read()` directement dans l'interface
5. Mettez en place une tâche watchdog permettant de terminer la simulation après un temps défini (en cas de blocage)

Pendant la mise au point de votre solution songez au fait que dans la prochaine partie les demandes de calcul seront déportées dans une autre application (en C/C++). Dès lors prévoyez pouvoir si possible récupérer du code (typiquement les scénarios).

Système complet

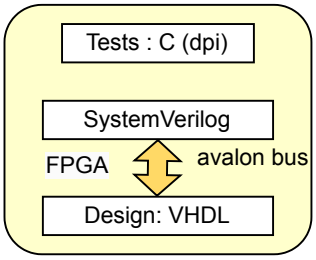
Nous sommes maintenant intéressés à tester le système complet FPGA-CPU-PC, et plus spécifiquement le cas où un calcul est déporté dans la FPGA depuis le PC, selon le schéma général suivant :



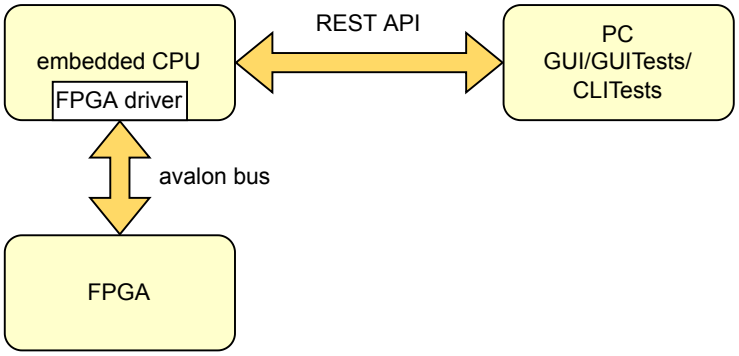
Ce calcul sera celui déjà exploité lors de l'étape précédente. Nous aurons juste une nouveauté :

— A l'adresse 6 se trouve un compteur de nombre de calculs. Il s'incrémente de 1 à chaque fois qu'un résultat est fourni en sortie du calculateur. Il est possible de le remettre à 0 simplement en écrivant n'importe quelle valeur à cette même adresse.

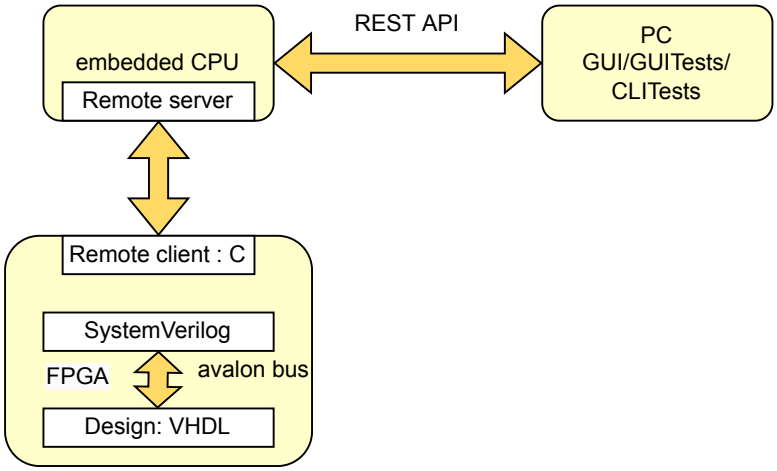
Lors de la première étape du labo vous avez mis en place une vérification du calculateur grâce à des tests écrits en C, via le DPI, selon le schéma suivant :



Nous désirons maintenant tester le système entier, en mêlant différents éléments allant de l'interface graphique au FPGA, avec ou sans tests automatiques :



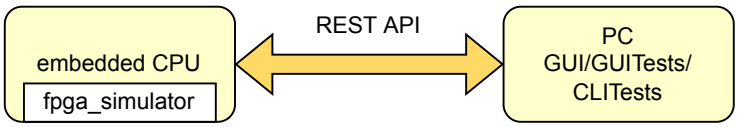
Pour ce laboratoire nous n'allons pas exploiter réellement une FPGA, mais resterons en simulation, en exploitant SystemVerilog et le DPI, de la manière suivante :



La communication entre le PC et le système embarqué a été implémentée via une API Rest. Le client exploite les possibilités offertes par Qt, alors que le serveur (embarqué) utilise la librairie pistache, qui est moins lourde que l'ensemble de Qt.
Pour simplifier le code, nous resterons avec une valeur de `N` fixée à 3 et une taille de données de 16 bits.

Partie 2

Nous allons commencer par nous intéresser à la communication entre le PC et le système embarqué :



Pour ce faire, nous n'allons pas directement simuler le VHDL, mais offrir une simulation C++ de ce que fait la FPGA. Pour ce faire, la commande qmake doit être faite avec l'argument `CONFIG+=config_fpgasimulator`. Ceci offre la possibilité de travailler sans la lourdeur d'une simulation HDL.
Le système est déjà partiellement fonctionnel, et lance deux calculs. Tout est en place pour que ceci se passe bien. A vous de mettre en place ce qui est nécessaire pour récupérer la valeur du compteur de nombre de calcul, son reset, et de mettre à jour les tests pour le valide.
Le code de la partie embarquée et celui de la partie graphique possède des `TODO`, qui vous guideront dans votre écriture. Votre professeur est également là pour vous guider.
Avant de vous lancer dans les modifications, effectuez les étapes de la mise route (ci-dessous), et passez un peu de temps à observer le code.

Partie 3

Maintenant que nous avons une communication entre le PC et la partie embarquée, il n'y a plus qu'à rajouter la communication CPU embarqué - FPGA. Pour ce faire, les fichiers `fpgaaccessremote.h` et `fpgaaccessremote.cpp` du logiciel embarqué devront être modifiés. La commande qmake doit être faite avec l'argument `CONFIG+=config_fpgaremote` pour exploiter ces fichiers.
Dans la partie simulation, il faudra modifier le fichier `amiq_top.sv`, de manière à implémenter les méthodes `avalon_write()` et `avalon_read()`, ainsi que la lecture des commandes ainsi que leur exécution.
Il vous faudra donc à priori ajouter deux commandes et implémenter la communication entre les deux parties.
Etant donné votre travail sur la partie 2, une fois ces deux commandes implémentées vous devriez pouvoir valider l'entier du système.

Mise en route

Copiez le répertoire `installation` en local de la VM (nécessaire pour des liens). Placez-vous ensuite dans ce répertoire, et dans un terminal, exécutez :

```
./install.sh
./install_pistache.sh
```

Pour la première installation, vous devriez avoir deux fois à répondre y et Y.
Après ces étapes vous disposez de pistache, Spix, anyRpc, une mise à jour des googletest, et tout ce qu'il faut pour le labo.

Exécution

L'exécution du tout implique plusieurs exécutables, qui doivent être lancés dans l'ordre :

1. Soft embarqué
2. Simulation de la partie FPGA
3. Interface graphique ou tests unitaires simulant l'interface

Tout peut se faire en ligne de commande, mais le soft embarqué et la partie PC peuvent être lancés depuis QtCreator, en ouvrant, compilant et lançant les projets correspondants.

Soft embarqué

En ligne de commande, allez dans le répertoire `embedded_soft`, puis tapez les commandes suivantes :

```
mkdir build
cd build
qmake ..
make -j4
./embedded
```

Simulation FPGA

En ligne de commande, allez dans le répertoire `fpga_sim_remote`, puis tapez les commandes suivantes :

```
export PROJ_HOME=$(pwd)
./arun.sh -tool questa
```

Logiciel PC

Il y a plusieurs moyens de le lancer :

1. GUI en version utilisateur
2. GUI en version tests automatiques
3. Tests automatiques en ligne de commande

Pour ces trois options, il est suggéré d'utiliser QtCreator, les sources se trouvant dans le répertoire `gui_soft`. L'interface graphique s'ouvre via le projet `gui/mathcomputergui.pro`.
Pour lancer les tests automatiques, il faut ajouter `CONFIG+=config_guitest` à la commande qmake.
La version ligne de commande pour tests automatiques correspond au projet `noguitests/noguitest.pro`
Pour info, si vous préférez la version sans QtCreator, il suffit d'effectuer

```
make
./<nom_executable>
```

Quelques informations

Le logiciel PC exploite la librairie Qt, et une interface écrite en QML. La librairie Spix est utilisée pour créer des tests automatiques de l'interface. La communication avec la partie embarquée exploite les possibilités de Qt concernant le réseau. Le système mis en place permet de facilement ajouter de nouvelles requêtes et créant de nouvelles sous-classes de `RESTCall`.
Le logiciel embarqué exploite la librairie pistache pour mettre en place un serveur REDS. Elle permet de facilement créer de nouveaux noeuds. La communication avec la simulation VHDL distante se fait via des socket Linux et la librairie pthread. Il est à noter que le même code peut être compilé avec connexion distante, simulation interne de la partie FPGA, ou interface réelle à la FPGA (non implémenté ici).
La simulation du VHDL exploite un banc de test SystemVerilog et le DPI pour s'interfaçer à du C. Dans la partie C un client TCP est mis en place pour se connecter au soft embarqué. Il permet ensuite de récupérer les requêtes du soft embarqué et d'interagir avec le VHDL.

Travail à rendre

Vous rendrez l'entier de vos sources, ainsi qu'un petit rapport présentant vos choix ainsi que vos commentaires sur le déroulement de ce laboratoire.
Etant donné qu'il y a beaucoup à rendre nous vous laissons le soin de préparer une archive avec tous les fichiers nécessaires.