ASR1 – TD10 : Premier contact avec VHDL et ModelSim !

{ Andreea.Chis, Matthieu.Gallet, Bogdan.Pasca } @ens-lyon.fr

4-5, décembre 2008

Pour bien comprendre ce TD, vous devez avoir lu l'introduction au VHDL figurant dans le poly du cours de Florent de Dinechin que vous trouverez à l'adresse :
http://perso.ens-lyon.fr/florent.de.dinechin/enseignement/2007-2008/ASR1/.

1 Simulation avec ModelSim

Le simulateur VHDL que vous allez utiliser pour simuler vos composants s'appelle ModelSim.

1.1 Lancement de ModelSim

Attention, ModelSim n'est pas un logiciel gratuit, il est strictement interdit de le diffuser. Par ailleur, nous ne possédons que 5 licences pour ce logiciel, donc vous ne pouvez pas l'utiliser sur plus de 5 machines en même temps.

Le répertoire contenant le simulateur est `/soft/enseignants/fdedinec/modeltech`. Le répertoire contient (entre autre) un répertoire `docs` contenant un tutoriel de ModelSim auquel vous pouvez vous référer.

Pour utiliser ModelSim, ajouter les trois lignes suivantes à votre `.bashrc` :

```
# pour modelsim
export MODELTECH=/soft/enseignants/fdedinec/modeltech
export PATH=$MODELTECH/linux:$PATH
export LM_LICENSE_FILE="1650@fpga2.lip.ens-lyon.fr"
```

Pour le lancer en mode graphique, il suffit alors de taper `vsim`. Vous trouverez en bas de la fenêtre une fenêtre de commande bien pratique.

Créez le répertoire de travail `work` en tapant `vlib work`.

1.2 Débuggage d'un décodeur pour afficheur 7 segments

1.2.1 Ouverture du décodeur

Récupérez le fichier `demo1_decoder.tar.gz` dans `http://perso.ens-lyon.fr/bogdan.pasca/ASR1/td/`, et décompressez-le.

L'archive est composée de deux fichiers :

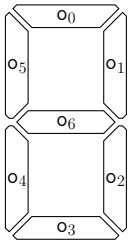
- `decoder.vhd`, qui contient véritablement le décodeur à simuler, et
- `decoder_test.vhd`, qui contient le composant de test qui va nous permettre de vérifier le bon fonctionnement du décodeur.

Tapez `vcom decoder.vhd` pour compiler le code `decoder.vhd` et `vcom decoder_test.vhd` pour compiler le code `decoder_test.vhd`.

Dans l'onglet library à gauche, vous pouvez alors voir dans `work` vos deux composants avec leurs architectures. Un clique droit puis `edit` vous permet de voir le code VHDL.

1.2.2 Le composant principal

Le décodeur a deux ports, le premier `i` en entrée, représentant en binaire le chiffre hexadécimal à afficher (donc sur 4 bits), et le second `o` en sortie, contenant les 7 signaux (numérotés comme indiqué figure suivante) à destination de l'afficheur 7 segments. Quant à l'architecture, pas de surprise, on utilise la construction `when` pour se simplifier la vie. Profitez-en d'ailleurs pour observer l'utilisation des `std_logic_vector`.



1.2.3 Le composant de test

Allez maintenant voir du côté du composant de test `decoder_test`. Ce coup-ci, aucun port d'entrée/sortie : tous les signaux vont être générés par ce composant. Dans l'en-tête de l'architecture sont d'abord définis les trois signaux `done`, `passed` et `ok` de type `boolean` sur lesquels nous reviendrons plus tard. Ensuite se trouve la déclaration du composant `decoder` pour qu'il puisse être instancié depuis `decoder_test`. Enfin, les définitions des signaux `i`, qui va être appliqué en tant que stimulus d'entrée au décodeur, et `o`, qui va recevoir la réponse du décodeur.

Le fonctionnement de ce composant de test est relativement classique : en envoyant un ensemble de stimuli au décodeur, puis en comparant la réponse obtenue avec la réponse attendue, il va pouvoir vérifier s'il fonctionne bien (tout du moins sur cet ensemble de stimuli). D'où le comportement des trois signaux booléens :

- Le signal `ok` nous indiquera pour chacune des réponses si celle-ci coïncide avec la réponse attendue ou pas, nous permettant donc de reprérer sur quels stimuli notre composant a échoué.
- Le signal `passed`, initialisé à `true`, restera vrai tant que le composant n'aura pas fait d'erreur. Ainsi, si à la fin du test, `passed` est toujours vrai, c'est que le composant a réussi le test. Par contre, s'il est faux, cela signifie qu'au moins une erreur a été détectée durant la simulation.
- Enfin, le signal `done` passe à `true` une fois le test achevé (c'est-à-dire que l'ensemble des stimuli a bien été soumis au composant à tester).

Dans l'architecture, on y trouve donc d'abord l'instanciation du décodeur, suivi de deux processes :

- le premier est chargé de générer les stimuli, au rythme d'un toutes les 10 nanosecondes,
- et le second, décalé d'une nanoseconde par rapport au premier (pour laisser le temps au décodeur de réagir aux stimuli), vérifie les réponses données par le décodeur, en mettant le signal `ok` à jour selon la réponse.

1.2.4 Compilation et simulation

Pour simuler l'entité `decoder_test`, tapez

```
vsim decoder_test
```

La fenêtre `Objects` vous montre alors les différents signaux et leurs valeurs.

Pour observer le chronogramme associé à un signal, faire un clique droit sur le signal puis `Add to Wave -> Selected Signals`. Evidemment, le chronogramme que vous observez pour le moment est vide. Il faut maintenant faire avancer la simulation.

Dans l'onglet sim dans la colonne de gauche est représentée la hiérarchie des instances de composants.

Vous pouvez aussi si vous le souhaitez afficher les signaux de l'instance `decoder_0` du décodeur, en cliquant sur cette instance dans la colonne de gauche.

Pour lancer la simulation proprement dite, sur une durée de 500ns, tapez :

```
run 500ns
```

qui va simuler le composant sur une durée de 500ns, ou bien

```
run -all
```

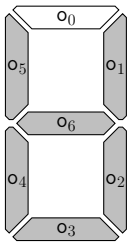
qui va simuler le composant jusqu'à ce que tous les signaux arrêtent de changer de valeur (c'est-à-dire qu'il n'y ait plus un seul événement dans la file d'attente du simulateur).

1.2.5 Un bug

En simulant un peu notre composant de test, vous vous apercevez que la valeur du signal `done` passe à `true` après 161ns de simulation. Et qu'à cet instant-là, la valeur de `passed` est à `false`. Il y a donc un bug dans le circuit.

La première étape est de repérer à quel moment le signal `passed` devient faux. En remontant rapidement le chronogramme, on trouve qu'il s'agit de l'instant 111ns. D'ailleurs, en regardant la valeur de `ok`, on se rend compte qu'il n'y a qu'à cet instant que ce signal est à faux. Nous n'avons donc qu'une erreur dans le test, mais c'en est déjà une de trop.

Regardons donc les valeurs de `i` et de `o` à ce moment-là : `i` vaut `"1011"` (soit le chiffre hexadécimal B) et `o` vaut `"1111110"`, ce qui nous donne l'affichage suivant :



Il y a donc bien un problème ! Le segment correspondant au fil `o(1)` devrait être éteint, pour que l'on puisse y lire un `b`. Heureusement, en allant farfouiller dans l'architecture du décodeur, on trouve bien vite la constante erronée dans la branche correspondante du `when`, qu'il suffit alors de corriger.

En relançant la compilation puis la simulation, ce coup-ci, tout marche bien.

1.2.6 Oui, mais

Et là, normalement, vous êtes censés demander :

"Comment faire s'il y a un bug dans le fichier de simulation ?"

Et vous n'auriez pas tort du tout, car effectivement, c'est une éventualité à ne pas écarter.

En général (même si ce n'est pas le cas du tout dans cet exemple), le fichier de simulation (ou bien le programme C qui aura servi à le générer automatiquement) est beaucoup plus court que le code VHDL à tester, et donc la probabilité d'y faire des erreurs est bien plus faible.

Cependant, soyez toujours vigilants lors de vos simulations, et n'hésitez pas à mettre en doute le composant de test autant que le composant testé.

1.3 Un multiplieur 8×8 bits pipeliné

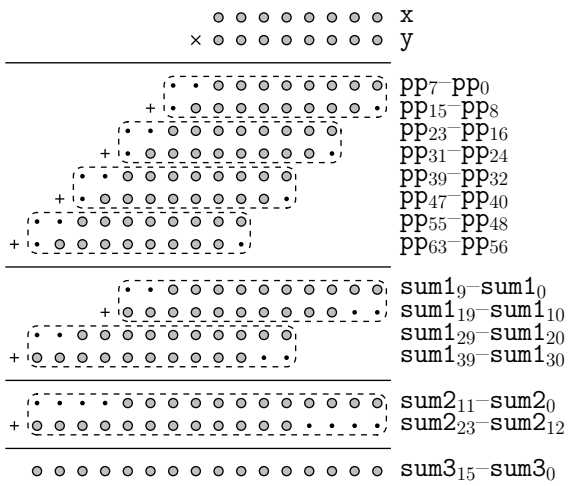
1.3.1 Ouverture du projet

Récupérez maintenant le fichier `demo2_mult.tar.gz` dans `/home/ploiseau` et décompressez-le.

Le projet est encore une fois composé de deux fichiers et de deux composants, `mult` qui définit le multiplieur pipeliné, et `mult_test` qui définit le composant de test pour ce multiplieur. Compilez les deux.

1.3.2 Le composant principal

Ce multiplieur est basé sur une génération en parallèle de tous les produits partiels, suivie d'un arbre d'additions. Des registres sont insérés après la génération des produits partiels, puis après chaque étage d'additions. Remarquez comme les suffixes `_`i des noms des signaux correspondent à l'étage du pipeline dans lequel interviennent ces signaux.



Les produits partiels sont donc générés par une double boucle `for ... generate` de sorte que le signal `pp_0`$_{8j+i}$ reçoive le produit partiel `x`$_i \cdot$ `y`$_j$. Les 64 bits du signal `pp_0` sont ensuite stockés dans un registre. Le signal en sortie du registre est noté `pp_1`.

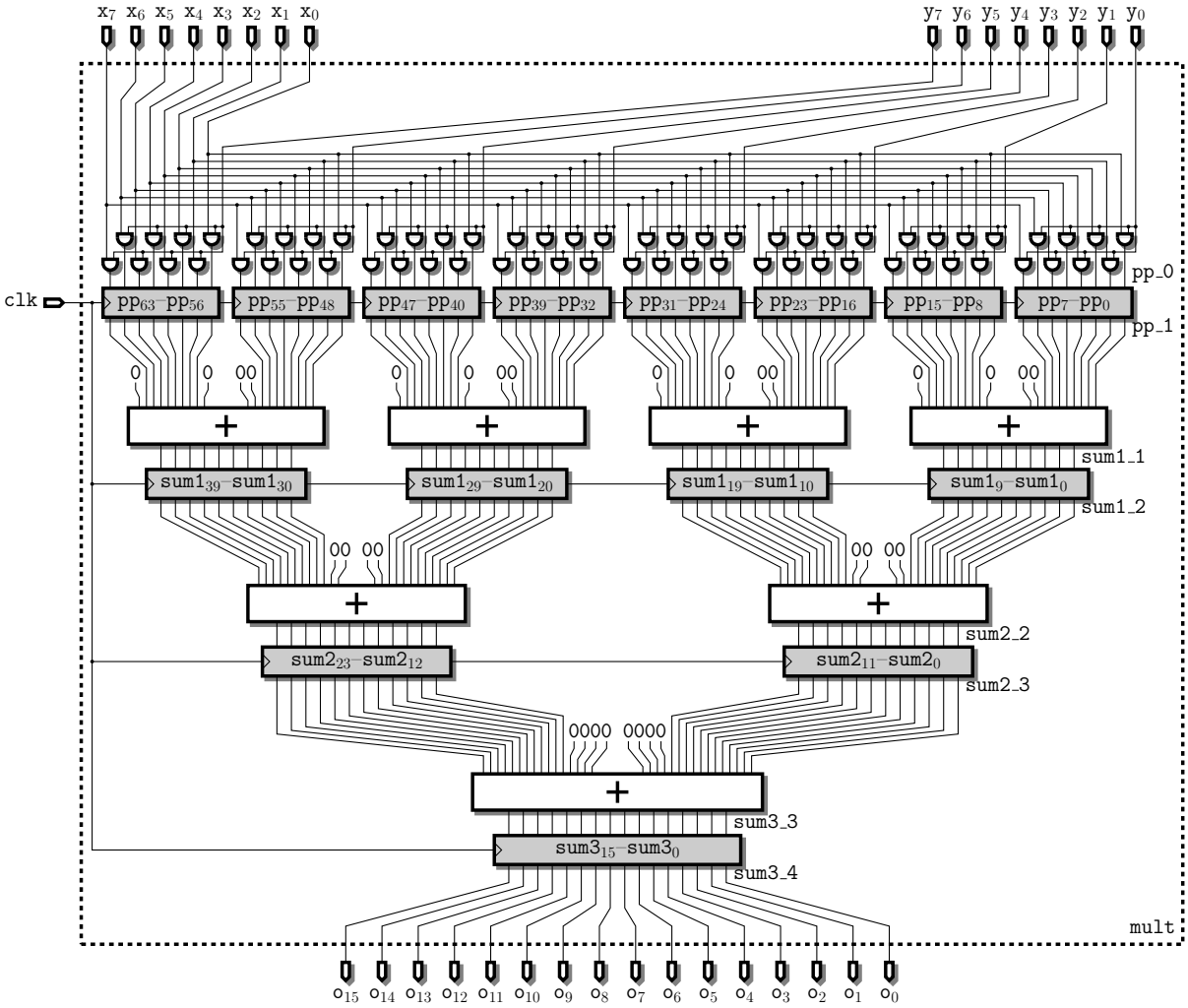
Les produits partiels sont ensuite répartis en huit groupes de 8 bits chacun : `pp_1`$_{8j+7}$–`pp_1`$_{8j}$ correspondant au produit `x`$\cdot$`y`$_j$. Les premières additions s'effectuent donc sur les quatre paires consécutives de ces groupes, avec les décalages adéquats. Remarquez au passage que pour ces premières additions, une retenue est succeptible de se propager, il faut donc l'absorber en rajoutant artificiellement des `'0'` en poids forts. Ces additions nous donnent donc quatre nombres de 10 bits, sous forme du signal `sum1_1`, qui est à son tour stocké dans un registre, pour donner `sum1_2` à l'étage suivant du pipeline.

Les quatre sommes partielles `sum1_2`$_{10j+9}$–`sum1_2`$_{10j}$ sont ensuite aussi additionnées deux par deux, en tenant compte bien-sûr des décalages. Ce coup-ci, il est facile de prouver qu'aucune retenue supplémentaire ne peut se propager hors de l'additionneur, donc pas besoin de ruser. On obtient donc le signal `sum2_2`, composé des deux sommes de 12 bits, qui devient `sum2_3` après un étage de registres.

Enfin, la dernière somme s'effectue de la même manière, et ne provoque pas non plus de retenue sortante. On obtient donc le nombre de 16 bits `sum3_3`, qui passe par un dernier étage

de pipeline avant d'être placé sur le port de sortie du composant.

On obtient donc le circuit suivant, effectivement pipeliné sur quatre étages :



1.3.3 Le composant de test

Ce coup-ci, le composant de test est bien plus simple que pour le décodeur 7 segments. On y retrouve bien les signaux usuels `done`, `passed` et `ok`, ainsi que les signaux de stimulus et de réponse pour le composant testé.

Le signal d'horloge à 100MHz est donc généré par le premier process, comme vu précédemment.

Le principal stimulus soumis au composant est généré par le second process, qui énumère tout simplement toutes les valeurs possibles pour les entrées `x` et `y` (on parle alors de test exhaustif[1]).

Enfin, le dernier process, après avoir attendu les quatre cycles d'horloge nécessaires à la propagation des valeurs au travers du pipeline du multiplieur, vérifie que le résultat donné par le multiplieur correspond bien au produit de ses deux opérandes.

[1]Question : en général, est-ce que ce genre de test exhaustif est suffisant ?

1.3.4 Un bug, bis

Compilez et simulez le tout sur quelques cycles. Ça semble marcher correctement. Poursuivez la simulation sur une petite dizaine de microsecondes. Et là, constatez que le signal `passed` est à `false`.

Allons donc voir ce qu'il se passe. En sélectionnant le signal `passed` dans le chronogramme, puis en cliquant sur le bouton Find next transition, vous pouvez accéder directement à l'instant où `passed` est passé à faux. Effectivement, le résultat obtenu est 1 alors que les opérandes, qui sont donc décalés de quatre cycles en arrière, sont respectivement 1 et 4.

Malheureusement, ici, la correction est moins simple que pour le décodeur, car le circuit est bien plus complexe. Il faut donc effectuer la simulation avec le détail des signaux internes de l'opérateur.

Pour cela, commencez par supprimer tous les signaux affichés dans le chronogramme, puis sélectionnez l'option Restart... dans le menu Run du menu Simulate (vous pouvez aussi tapez `restart -f`). Ajoutez au chronogramme les signaux `ok` et `clk` du composant `mult_test`, ainsi que `x`, `y`, `pp_1`, `sum1_2`, `sum2_3` et `sum3_4` du composant `mult`. Vous pourrez ainsi voir les données contenues dans chacun des registres de l'opérateur, ce qui suffit généralement à localiser le bug.

Simulez donc le circuit jusqu'à l'instant fatidique. Si vous regardez plus précisément ce qu'il se passe, vous pouvez constater que les produits partiels (`pp_1`) sont corrects, ainsi que les premières sommes (`sum1_2`). Par contre, la somme $\texttt{sum2\_3}_{11}$–$\texttt{sum2\_3}_0$ (celle des poids faibles) est fausse.

Et effectivement, l'erreur se trouve bien sur la ligne 66 de `mult.vhd`. Vous pouvez la corriger puis relancer la simulation exhaustive pour vérifier que tout est bien correct.