Architecture, Système, Réseau 2017-2018
{alice.pellet___mary,fabrice.mouhartem}@ens-lyon.fr, florent.de-dinechin@insa-lyon.fr

ASR1, TP7 : Introduction au VHDL

Dans ce TD, on va décrire et simuler des circuits en utilisant VHDL, qui est un langage de description de circuit. C'est un standard industriel, moins facile à prendre en main mais plus puissant que Logisim. On utilisera le simulateur `nvc` pour simuler des circuits décrits en VHDL, et le visualisateur de traces *GTKWave*.

Prise en main de VHDL

Pour cette partie du TP, vous pouvez jeter un œil à l'annexe A du poly. Arrêtez-vous avant A-3, et enchaînez sur le TP.

Faire fonctionner le compilateur *nvc*

Avant tout, décompressez le fichier `tp1-vhdl.tgz`.

Si vous êtes sur votre machine, récupérez `nvc` sur le dépôt git suivant : `https://github.com/nickg/nvc` (`git clone https://github.com/nickg/nvc.git`), et essayez de vous débrouiller. Le service après-vente ne sera assuré que pour les utilisateurs sous *linux*.

Si vous êtes sur les machines des salles libres services, il faudra faire quelques bricoles.

Attention : ceci n'est clairement pas la marche à suivre pour avoir un environnement sain, mais cette méthode a le mérite de fonctionner.

Compilez le fichier `flock.c` de l'archive `tp1-vhdl.tgz` pour en faire une bibliothèque dynamique :

```
mkdir -p "$HOME/.local/lib"
gcc -shared -fPIC flock.c -o "$HOME/.local/lib/flock.so"
```

Et rajoutez dans votre `.bashrc` (ou équivalent suivant la *coque logicielle* que vous utilisez) l'*alias* suivant :

```
alias nvc="LD_PRELOAD=$HOME/.local/lib/flock.so nvc"
```

Finalement, relancez vos terminaux au besoin.

> Pour ceux qui se posent la question de ce qu'on vient de faire, et qui ne souhaitent pas attendre le cours d'*ASR2*, il s'avère que l'appel à la fonction `flock` («`man 2 flock`» pour plus d'informations) foire dans un cas précis. C'est un mécanisme de verrouillage de fichiers, et il s'avère que ça plante sur le fichier… de version de `nvc`. Une chance pour nous, les liens entre les bibliothèques sont gérées dynamiquement ; ce qu'on fait est donc prendre la main sur le *vrai* `flock`, et utiliser le notre qui fait un appel au `flock` véritable mais qui en ignore simplement le retour (donc quand ça marche tant mieux, si ça rate… tant pis).
>
> C'est pour ça qu'il a été dit que ce n'était pas une solution saine, parce que si nous voulions être vicieux, nous aurions pu vous fournir un *.so* compilé sans garantie qu'il ne s'agisse pas d'un programme malveillant (par exemple).

Passer un circuit combinatoire au banc de test

Entités et instances

Ouvrez dans un éditeur de texte `fulladder.vhdl` et `testbench_fulladder.vhdl`.

Le premier contient une *entité* `fulladder` qui est un full-adder (légèrement saboté). L'entité décrit une boîte noire, une interface. L'*architecture* d'une entité décrit une implémentation de cette boîte noire (en Logisim, on avait d'abord décrit l'architecture de `add1bit`, puis on l'avait emballée dans une entité pour pouvoir la réutiliser).

On peut avoir plusieurs architectures par entité, et c'est utile en pratique, d'où cette syntaxe un peu lourde. Mais dans ce TP, on n'aura toujours qu'une architecture par entité.

Le second fichier, `testbench_fulladder.vhdl`, contient une entité `testbench_fulladder`[1] qui est un banc de test. Ce banc de test décrit une *instance* de `fulladder`, et un processus séquentiel `process` qui applique des entrées à ce composant.

Vérifiez que vous comprenez la différence entre une entité et une instance. Indice : c'est la même différence qu'entre une fonction et un appel de fonction, mais pour le matériel (en Logisim, on a utilisé plusieurs instances de l'entité `add1bit`).

Faites un dessin correspondant à l'entité `testbench_fulladder` : un gros rectangle qui représente la boite noire, et des sous-composants dedans, comme en Logisim.

Validez ce dessin avec un enseignant.

1. Ce n'est pas obligatoire que le fichier ait le même nom que l'entité. On peut aussi mettre plusieurs entités par fichier.

Simulation du VHDL

Ajoutez les 6 cas de test qui manquent au banc de test, par copier-coller. Pour simuler, nous utiliserons le simulateur VHDL libre `nvc`.

La commande `nvc --help` liste ses options. Celles qui nous intéressent sont

- `nvc -a fichier.vhdl` (pour *analyse*) analyse un fichier VHDL (et crée un répertoire `work` dans le répertoire courant et y range des trucs et des machins).
- `nvc -e entity_name` (pour *elaborate*) décrit quelle est l'entité qu'on simule parmi celles qui ont été rangées dans `work`
- `nvc -r` lance la simulation, avec les options suivantes :
  - `--stop-time` sans quoi la simulation tourne indéfiniment (comme votre téléphone portable).
  - `--wave=file.fst` pour produire un fichier de trace (un chronogramme) que nous pourrons observer à l'aide de `gtkwave`.

Essayez donc :

```
nvc -a fulladder.vhdl
nvc -a testbench_fulladder.vhdl
nvc -e testbench_fulladder -r --stop-time=100ns --wave=chronogram.fst
gtkwave chronogram.fst
```

Dans la fenêtre *GTKWave*, la colonne du milieu contient les signaux que vous voulez voir apparaître dans le chronogramme : tirez-y à la souris votre signal `testcin` depuis la colonne de gauche. Il faut aussi cliquer sur "Time → Zoom → Zoom Best Fit" pour mettre à une échelle correcte l'axe des temps (il y a une icône et un raccourci clavier aussi).

Observez les deux sorties du *full adder*. Constatez qu'il ne marche pas du tout. C'est qu'on a saboté ses équations logiques. Réparez-les.

Faites valider par un enseignant une fenêtre GTKWave qui montre un fulladder qui marche.

Construire un additionneur paramétré

Description structurelle de l'additionneur

Dans le fichier `adder.vhdl` vous avez plusieurs nouveautés.

- Le type `std_logic_vector` décrit un vecteur de bits. On décrit sa taille par les indices min et max, inclus tous deux. Si `x` est un `std_logic_vector`, alors `x(3)` est le bit numéro 3.
- Des paramètres génériques de l'architecture peuvent être donnés avec le mot-clé `generic`. On l'utilise ici pour définir la taille n de l'additionneur. Ici c'est un entier, mais on peut avoir des `generic` de n'importe quel type. Une remarque importante : pour simuler une architecture, on aura besoin que tous ses paramètres soient instanciés.
- La boucle `generate for` itère dans l'espace, pas dans le temps.

`testbench_adder.vhdl` fournit un banc de test pour `adder`, en instanciant un additionneur 8 bits. Ajoutez quelques tests d'addition. Vérifiez en simulant que vous calculez bien la somme. L'exercice est surtout de trouver les bonnes lignes de commande.

Déplacez la ligne `c(0) <= cin;` dans le `adder` à la fin, et constatez que cela ne change rien à la simulation.

Expliquez ce miracle à un enseignant.

En attendant qu'il arrive, apprenez à vous servir de GTKWave :

- Dans la colonne du milieu, double-cliquer sur un signal de n bits (par exemple `s[7:0]`) montre tous ses bits séparément.
- La boîte en haut à gauche vous montre la hiérarchie de votre circuit. Ici l'entité racine ne contient qu'une instance, `uut`, et des signaux qui sont listés dans la boîte en bas à gauche. Cliquez sur `uut` : qu'est-ce qui change ? Cliquez sur la petite croix à gauche d'`uut` : qu'est-ce qui change ?
- Continuez à cliquer dans ces deux boîtes de gauche pour comprendre comment vous pouvez observer tous les signaux de votre circuit, même ceux qui sont cachés au fond des sous-circuits.

Un description plus précise des aspects temporels

Le seul endroit dans lequel un calcul est réalisé est le `fulladder`. Pour observer un chronogramme plus réaliste, on va spécifier le temps de ce calcul. VHDL permet d'écrire des choses comme

```
a <= b and c after 5 ps;
```

Ici `ps` désigne la picoseconde (qui vaut combien de secondes ?).

Jeu numéro 1 Supposons que le délai typique d'une porte `XOR` est de 15 ps et le délai typique d'un inverseur, d'un `AND` ou d'un `OR` (à deux entrées) est de 10 ps.

Raffinez votre entité `fulladder`. Observez le chronogramme de l'addition 01111111 + 00000001.

Faites valider par un enseignant.

Jeu numéro 2 (si vous êtes en avance). En CMOS, on n'a droit qu'au `NAND`, `NOR` et inverseur, avec un délai de 5 ps chacun. Réécrivez les équations logique du full adder pour n'utiliser que ces portes. Cherchez des équations logiques qui minimisent le temps de propagation de retenue.

Jeu numéro 3 (si vous êtes très en avance). En fait on contrôle le délai des portes à 20% près. Pour modéliser ceci, mettez (au hasard) certains délais à 4 ps et certains délais à 6 ps. Observez l'apparition de transitoires (*glitches*) dans la simulation.

Registres et mémoires

Jusqu'à présent on s'est contenté de décrire des circuits combinatoires : la valeur des sorties de votre additionneur à un instant donné t ne dépend pas du passé, mais uniquement de la valeur de ses entrées à cet instant précis.

Voyons à présent comment on décrit en VHDL des circuits séquentiels.

Description comportementale des registres en VHDL

Description comportementale ou description structurelle

Jusque là, on a dessiné nos architectures comme des boites qu'on assemble par des fils. Il est donc naturel que les outils comme Logisim offrent une interface à base de boites et de fils. C'est aussi le cas de VHDL, qui appelle une boîte une *entity* et un fil un *signal*.

Il faut aussi décrire l'intérieur de la boîte noire. VHDL appelle cela une architecture pour une entité. On peut donc décrire un gros circuit par l'assemblage de boîtes plus petites. Comme en LogiSim, quoi. Ceci s'appelle une description structurelle.

Mais VHDL permet aussi de décrire un composant par son comportement. Tant qu'on reste sur des comportements simples (par exemple le comportement d'un registre), on saura les compiler automatiquement en un assemblage de porte qui implémente ce comportement. Cela s'appelle une description comportementale.

Les deux ne sont pas antinomiques ; il s'agit de trouver le bon équilibre. En gros,

— on aime bien décrire les gros circuits de manière structurée : des boîtes composées de boîtes plus petites. Sinon on ne s'y retrouve pas. Donc pour les grandes lignes, description structurelle.

— Par contre on veut laisser à des outils automatiques l'optimisation de l'intérieur des boîtes les plus petites : les TD sur l'optimisation logique étaient un peu gonflants, n'est-ce pas ? Donc pour ces plus petites boîtes, on se contentera volontiers d'une description comportementale.

Rétrospectivement, quand vous avez écrit en VHDL

```
z <= a and b and c;
```

c'était déjà en fait une description comportementale : cette ligne sera compilée sous forme de portes `NAND` et `NOR` en `CMOS`, sous forme de LUTs en FPGA, etc. Toute implémentation respectant sont comportement logique fera aussi bien l'affaire.

Pour décrire un comportement temporel, c'est plus compliqué. VHDL vous fournit un langage de programmation presque classique, avec variables, conditionnelles, etc. Ce langage séquentiel est emballé dans un "`process`". Il est destiné à s'exécuter dans le simulateur. Il faut bien comprendre qu'il ne sera pas transposé directement en un circuit.

Mais d'abord, un générateur d'horloge

Pour commencer nous allons simuler le plus simple des circuits séquentiels : un générateur d'horloge. Un exemple de code VHDL est donné dans le fichier `clock.vhdl`.

Allez le lire et comprenez-le bien. Si vous avez l'impression qu'il y a des fautes de frappe dedans, c'est normal. Réparez-le jusqu'à pouvoir le simuler.

Description du comportement du registre

Un registre est défini en VHDL, dans une `architecture`, par un `process`. Ce process doit réagir à un évènement sur l'horloge, ce qu'on écrit `process(clk)`. Voici une manière d'écrire un registre à un bit :

```
process(clk)
   begin
      if rising_edge(clk) then
         q <= d;
      end if;
   end process;
```

Voici un registre avec clock enable :

```
process(clk)
   begin
      if rising_edge(clk) then
         if enable = '1' then
            q <= d;
         end if;
      end if;
   end process;
```

Remarquez que l'on n'a pas écrit `process(clk,enable)`. Ce serait syntaxiquement correct : la parenthèse après le `process` décrit la *liste de sensibilité* du processus, c'est à dire la liste des signaux sur lesquels une transition réveille le processus. Mais on veut construire un registre avec `enable` *synchrone* : il doit prendre en compte `enable` uniquement au front montant de l'horloge. Inutile donc de réagir à un évènement sur `enable`.

Enfin voici un registre avec reset :

```
process(clk)
   begin
      if rising_edge(clk) then
         if reset = '1' then
            q <= 0;
         else
            q <= d;
         end if;
      end if;
   end process;
```

Question 1 Dessinez un chronogramme du fonctionnement de ce registre avec reset, quand reset passe de 0 à 1 puis à 0 à deux fronts descendants successifs de l'horloge.

Faites valider par un enseignant ! Si vous avez un doute, faites un testbench et montrez à l'enseignant un chronogramme dans GTKwave.

Question 2 Écrivez une entité `Reg_N_Reset` pour "registre n bits avec reset".
Remarque : On pourrait emballer le code ci-dessus dans un registre 1 bit, puis construire une architecture n bits avec un `for generate`. Mais on peut aussi faire moins structurel : les morceaux de code ci-dessus marchent (presque) si `q` et `d` sont des `std_logic_vector`.
Par contre pour le reset, il faut assigner à `q` un vecteur de bits composé de n zéros : cela s'écrit

```
q <= (n-1 downto 0 => '0');
```

et cette flèche `=>` n'a rien à voir avec celle que vous avez vu dans le `port map`, qui n'avait rien à voir avec la flèche `<=`... Désolé.

Un compteur

Question 3 Construisez une entité compteur. Son architecture assemblera votre additionneur n bit et votre registre n bit, selon la première figure du ch.5 du poly.

Si vous ne savez pas démarrer, dessinez sur un papier un grand rectangle : l'entité compteur. Dessinez dedans la figure précédente. Nommez les fils, et reliez ceux qu'il faut aux entrées/sorties de votre entité.

Question 4 Construisez un banc de test qui alimente votre compteur par un générateur d'horloge (après avoir mis reset à 1 assez longtemps), et regardez-le compter dans GTKWave. Affichez aussi dans GTKWave la sortie de l'additionneur (en allant la chercher dans la hiérarchie du circuit, panneau en haut à gauche). Qu'observez vous ?

Question 5 Un peu d'*overclocking* ! Augmentez la fréquence d'horloge jusqu'à ce que votre compteur ne marche plus.

Pour cette question on pourra modifier le générateur d'horloge pour qu'il prenne un paramètre générique qui donne sa fréquence.

Faites valider par un enseignant.

Si vous êtes en avance ou si vous voulez faire des heures sup, vous pouvez écrire en VHDL une architecture structurelle (à base de MUX) pour votre entité registre avec reset. VHDL permet de remplacer une architecture par une autre de manière transparente. Vous devrez chercher comment on choisit quelle architecture est associée à quelle instance de quelle entité. C'est relativement bien expliqué ici : `http://www.pldworld.com/_hdl/1/www.ireste.fr/fdl/vcl/lesd/les_6.htm`

Construire une ALU en VHDL

Deux constructions syntaxiques utiles pour les vecteurs de bits :

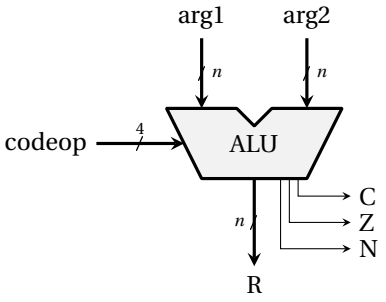
— `(n-1 downto 0 => '0')` crée un vecteur constant `"000000"` de taille n.

— l'opérateur `&` fait la concaténation de deux vecteurs de bits.

Par exemple la constante 1 sur n bits s'écrit `one <= (n-1 downto 1 => '0') & '1';`

Par ailleurs le fichier `mux.vhdl` contient un exemple de syntaxe pour un multiplexeur à 4 entrées. Il vous faudra ci-dessous un multiplexeur à 16 entrées. Vous pouvez en faire une entité séparée, mais vous pouvez aussi utiliser la syntaxe à base de `select` directement dans votre `architecture`.

On vous demande de réaliser une ALU *(arithmetic and logic unit)* dont la boîte noire est la suivante :



Elle ne calcule qu'en complément à 2 sur 8 bits. Les sorties c, z, n sont des *drapeaux* qui indiquent respectivement s'il y a un *carry out*, si le résultat est nul *(zero)*, si le résultat est négatif. Le codage de l'opération est donné par la table suivante.

TABLE 1 – Encodage des différentes opérations possibles

codeop	mnémonique	MàJ drapeaux	remarques
0000	`arg1 + arg2 -> dest`	oui	addition
0001	`arg1 - arg2 -> dest`	oui	soustraction
0010	`arg1 and arg2 -> dest`	oui	`et` logique bit à bit
0011	`arg1 or arg2 -> dest`	oui	`ou` logique bit à bit
0100	`arg1 xor arg2 -> dest`	oui	`ou exclusif` bit à bit
0101	`LSR arg1 -> dest`	oui	logical shift right ; bit sorti dans C ; arg2 inutilisé
1000	`(not) arg1 -> dest`	oui	arg2 inutilisé
1001	`arg2 -> dest`	non	arg1 inutilisé

Remarque : les codeop non mentionnés dans cette table sont inutilisés pour le moment (réservés pour une extension future...). En attendant, votre ALU peut bien sortir ce qu'elle veut dans ces cas-là : *don't care !*

Écrivez un banc de test qui teste différentes opérations.

Et s'il reste du temps

On s'use un peu les yeux sur ces chronogrammes. VHDL vous permet donc d'écrire un banc de test qui compare la sortie de votre ALU avec la sortie attendue, et produit un message d'erreur en cas de désaccord. Vous trouverez tout sur le Ternet, par exemple en l'interrogeant sur les mots-clé *assert* et *report*.