TP1: Un interprète de net-list

Timothy Bourke <Timothy.Bourke@ens.fr>

1er octobre 2016

Les fichiers nécessaires au TP peuvent être récupérés à l'adresse : `http://perso.telecom-paristech.fr/~guilley/ENS/program_2016_2017.html` :

- tp1.zip : sources, binaire du simulateur du net-list et quelques test pour le TP.
- minijazz.pdf : une description des languages MiniJazz et net-list.
- minijazz.zip : sources, binaire, et quelques exemples pour le compilateur MiniJazz (sans le code source du simulateur qui constitue la première partie du projet). Vous pouvez modifier ces sources pour la suite de votre projet.

Ordonnancement

Le compilateur MiniJazz génère des net-list dans un ordre quelconque. La première étape pour les simuler est d'ordonner les équations afin de pouvoir les exécuter séquentiellement. Cela se fera par un tri topologique du graphe de dépendences du circuit.

Question 1. *Le fichier* `graph.ml` *contient une implémentation simple de graphes orientés impératifs. Complétez le fichier pour ajouter la détection de cycles et le tri topologique.*

```
(* Detection de cycles *)
val has_cycle : 'a graph -> bool
(* Renvoie les noeuds dans un ordre topologique *)
val topological : 'a graph -> 'a list
```

Pour tester votre module, compilez le programme de test avec :

```
> ocamlbuild graph_test.byte
```

et vérifiez que le résultat est bien celui attendu.

On va maintenant construire le graphe de dépendences d'un circuit. Les noeuds de ce graphe sont les variables du circuit et il existe un chemin de `x` vers `y` si on a besoin de la valeur de `y` pour calculer `x`.

Question 2. *Ecrivez une fonction* `read_exp` *(dans le fichier* `scheduler.ml`*) qui renvoie la liste des variables lues par une équation, de signature :*

```
val read_exp : Netlist_ast.equation -> Netlist_ast.ident list
```

(On ignorera pour l'instant le cas des registres et des mémoires qui seront traités dans la question 4)

Question 3. *Ecrivez une fonction* `schedule` *(dans* `scheduler.ml`*) qui prend en entrée une net-list et renvoie la net-list ordonnée afin de permettre une exécution séquentielle :*

```
val schedule : Netlist_ast.program -> Netlist_ast.program
```

Votre fonction devra lever l'exception `Combinational_cycle` *si jamais la net-list contient une boucle combinatoire (On ignorera pour l'instant le cas des registres et des mémoires qui seront traités dans la question 4).*

Pour tester votre ordonnanceur, compilez le programme avec :

```
> ocamlbuild scheduler_test.byte
```

puis exécutez les programmes de test. Par exemple :

```
> ./scheduler_test.byte test/fulladder.net
```

L'exécution du programme génère une nouvelle *net-list* `fulladder_sch.net`, puis exécute le simulateur sur cette net-list (sauf si l'option `-print` est donnée). On peut choisir le nombre de pas effectués par le simulateur avec l'option `-n <nombre>`.

Les registres permettent de casser les cycles de causalité, puisque l'on peut lire la valeur d'un registre avant d'avoir écrit la nouvelle valeur.

Question 4. *Ajoutez la prise en compte des registres et des mémoires à votre ordonnanceur.*

Un interprète de net-list

Un *interprète* est un programme qui évalue une net-list en utilisant un *environnement* pour stocker les valeurs calculées précédemment. Cet environnment associe à un identifiant (`Netlist_ast.ident`) une valeur (`Netlist_ast.value`).

Question 5. *Quelle est la structure générale de l'interprète ne gérant que les circuits purement combinatoires ? Ecrivez le pseudo-code correspondant.*

Question 6. *Comment gérer les registres et les mémoires ?*