Rapport : Simulateur de net-List

Josselin Giet

6 novembre 2016

1 Généralités sur le simulateur

Le simulateur de net-list est écrit en OCaml et s'inspire des codes fournis dans le cadre du tp1.

Pour lancer la simulation d'une net-list il faut se placer dans le répertoire envoyé par mail et construire le fichier `netlist_simul.ml` au moyen de la commande :

```
ocamlbuild -no-hygiene netlist_simul.byte
```

Une fois la construction du fichier `netlist_simul` effectuée, on peut simuler une net-list en exécutant la commande `./netlist_simul.byte`.

Dès lors la programme va demander le nom de la net-list à simuler. Par exemple : `test/fulladder.net`.

Puis le simulateur demande le nombre d'itération à effectuer.

Le simulateur demande aussi la saisie de la ROM et de la RAM sous la forme d'une chaîne de `0` et de `1` en veillant à ce que le début de chaque mémoire corresponde au début de la d'entrée.

A partir de ce moment,le simulateur commence ses calculs.

A chaque début d'étape, le simulateur procède à la saisie des inputs. Il faut ici faire attention car le simulateur ne vérifie pas les inputs (*i.e.* si une input est incorrecte le simulateur va quand même se lancer et échouera donc durant la phase de calculs) Une fois les calculs effectués, le simulateur affiche les outputs.

2 Description du simulateur

Le simulateur suit les étapes suivantes :

1. "Ouverture" et lecture de la net-list à simuler, au moyen de la fonction `read_file`.
2. Saisie de la ROM et de la RAM.
3. Ordonnancement de la net-list au moyen de la fonction `schedule`, puis impression de la net-list ordonnée dans un fichier `filename_sch.net`. (ette phase est un copy-paste d'un fragment du fichier `scheduler_test.ml`.
4. Saisie des inputs (*cf.* supra)
5. Calcul de chaque équation.
6. Gestion des registres
7. Écriture dans la RAM (On écrit donc la valeur de `data` calculée à l'instant, posé comme convention).
8. Affichage des outputs

Remarque : Les étapes 5 à 7 sont effectuées n fois.

2.1 Représentation de l'environnement

Pour représenter l'environnement (*i.e.* les valeurs déjà calculées), on utilise ne table de hashage (module `Hashtbl` de OCaml) dont les clés sont de type `ident` et dont les valeurs stockées sont de type `value`.

Les tables de hashage présent un avantage par rapport au module `Map.Make` : les tables peuvent être modifiées sur place, ce qui facilite l'écriture des fonctions annexes.

2.2 Représentation des registres

La gestion des registre se font au moyen du module `Stack`, pour stocker les changements de variables à effectuer en fin de phase de calcul.

2.3 Représentation de la RAM et de la ROM

La RAM et la ROM sont tous deux représentés par un `bool array ref`. Le type `array` est ici utilisé car il permet un accès en temps constant à chaque case du tableau pour la lecture et l'écriture.

De même que pour les registres, on mémorise les écritures dans la RAM dans un `Stack`.

3 Description des différents dossiers

Nom du fichier	Fonction
`netlist_ast.ml`	Décrit l'arbre de syntaxe abstraite des fichier lus
`netlist_lexer.mll`	Analyseur lexical des fichiers `.net`
`netlist_parser.mly`	Analseur syntaxique des fichiers `.net`
`netlist.ml`	Lit et écrit un fichier `.net`
`netlist_printer.ml`	Sert à l'écriture d'une net-list.
`graph.ml`*	Permet de détecter les cycles combinatoires
`scheduler.ml`*	Ordonne la liste `p.p_eqs` pour effectuer les calculs dans le bon ordre.
`netlist_simul.ml`*	Fichier principal : calculs des équations, gestion des registres,RAM,ROM

FIGURE 1 – INVENTAIRE DES DIFFÉRENTS FICHIERS UTILISÉS

La FIG 1 recense les différents fichiers et précise leurs rôle [1].

On utilise un grand nombre de fichiers donnés dans le cadre du tp1. Je vais donc préciser les fichiers modifiés.

3.1 `graph.ml`

Dans ce fichier ont été rajoutées les fonctions `has_cycle` et `topological` qui correspondent aux fonctions de la question 1 du tp1.

3.2 `scheduler.ml`

Ce fichier a pour but d'ordonner la net-list afin qu'elle puisse être exécutée sans problème d'ordonnancement. On trouve ainsi la fonction `read_eq` quilit les arguments non constants dans une ligne de calcul, et la fonction `schedule` qui ordonne les équations. cette fonction utilise la fonction `has_cycle` du rfichier `graph` pour détecter une boucle combinatoire, mais n'utilise pas la fonction `topological` mais un critère plus fort : on ajoute une équation dans la liste si tous les arguments non constants sont déjà dans la liste. On garde tout de même une compléxité en $O((Card)(p.p\_eqs)^2)$

1. Les fichiers avec un symbole * sont ceux dont le contenu a été modifié.

3.3 `netlist_simul.ml`

Il s'agit du coeur du simulateur. Ce fichier contient deux fonctions principales.

Tout d'abord la fonction `eval_eq` qui étant donnée une equation (*i.e.* de type `eq`) rajoute dans l'envirronement `env` la valeur de la variable de type `ident`.

Et la fonction `main` qui est la fonction principale du simulateur.

Les autres fonctions servent soit à manipuler l'encvirronemen (ajout des inputs), ou bien à faire des transitions entre les types.

4 Commentaire et amélioration possibles

Incertitude sur les conventions :

A de nombreuses reprises j'ai été amené à poser des conventions de manière arbitraire.

Par exemple, dans le cas du `mux` j'ai choisi la convention :

$$mux\, m\, a\, b ::= b \text{ si } m = true$$
$$|c \text{ sinon}$$

qui correspond à celle du polycopié de cours, bien qu'on trouve la convention inverse.

De même, pour la RAM, j'ai choisi de reporter l'écriture à la fin de la phase de calcul. On peut aussi faire l'écriture sur place quitte à poser `0` la valeur d'écriture.

Gestion des mémoires (ROM/RAM) :

La mémoire est ici gérée de façon naïve car on ignore a priori sa taille. Pour cela il faut lire une équation dans laquelle on procède à une lecture/écriture dans la mémoire. Et on ne peut pas déterminer où se trouve une telle équation voir même si une telle équation apparaît dans la net-list.

Ainsi, le fait de saisir *à la main* la net-list semble devoir être modifier car une telle saisie peut être longue et source d'erreur. A l'avenir (et notamment dans le cas de la montre digitale) on pourra spécifier texte dans le code qui correspondra à la mémoire à importer.

Gestion des Inputs :

De manière plus générale le gestion des inputs doit être effectuée de façon plus rigide car si une input est incorrecte le programme ne détecte pas d'erreurs. Il va donc se lancer et échouer plus tard.

Un tel problème semble toutefois secondaire dans la cas d'une application à une montre digitale.