Simulateur NetList – Système digital partie 1

Théophile Bastian

Rendu : 08/11/2015

Résumé

Pour la partie « simulateur » du projet, j'ai choisi d'implémenter un *compilateur* NetList vers C++ pour des raisons de performance ; écrit en OCaml. L'utilisateur est alors libre d'utiliser un quelconque compilateur C++ pour obtenir un exécutable *d'un circuit particulier* ; pouvant recevoir sur l'entrée standard l'état des pins d'entrée et *via* ses arguments un fichier binaire de ROM ; et affichant après chaque cycle d'exécution sur la sortie standard l'état des pins de sortie. Le compilateur gère quelques options permettant de modifier légèrement le code produit.

1 Compilation du compilateur

La compilation est réalisée par un simple appel à make, nécessitant toutefois quelques dépendances :
— ocamlopt
— menhir, pour la compilation du parseur
— bash et les outils classiques d'une console Unix

Ainsi, la compilation est aisée sous Linux et Mac. Cela devrait être vrai aussi sous Windows avec un shell bash installé (tel que CygWin). Le binaire produit est *simcomp*.

Attention ! Il est possible qu'il soit nécessaire, en cas de modification du code, de lancer un make clean && make plutôt qu'un simple make.

2 Utilisation

2.1 Utilisation basique

Pour produire un binaire « circuit.bin » à partir d'un fichier netlist « circuit.net », sans passer par l'intermédiaire C++ et en se plaçant à la racine du projet Simcomp compilé,

```
./compile.sh circuit.net circuit.bin
```

2.2 Utilisation avancée

Pour générer un code C++ (non-indenté mais à peu près lisible une fois indenté et coloré) à partir de « circuit.net », on utilise la commande

```
./simcomp circuit.net
```

(en pratique compile.sh n'est presque qu'un wrapper autour de ./simcomp $1 | g++ -xc++ -O2 -o $2 -).

Simcomp accepte quelques options (./simcomp [options] fichier .net) :
— **-n** : n'affiche pas de retour de chariot (\n) entre les sorties des différents cycles (voir section 2.4) ; n'attend pas de retour de chariot sur l'entrée pour séparer les cycles.
— **--ramSize [k : int]** : définit la taille d'une RAM à k bits (voir section 3.5).
— **-On**, $n \in \{0, 1\}$: définit le niveau d'optimisation. Actuellement seuls les niveaux 0 et 1 sont pris en charge, 1 étant par défaut sélectionné. Voir section 3.4.

Notez que ./compile.sh accepte les mêmes options, placées **après** ses arguments obligatoires.

2.3 Script de test

Le script ./runTests.sh permet d'exécuter des tests automatiquement. Il suffit de lui passer en paramètres une liste de dossiers contenant des tests.

La structure d'un test nommé test01 est la suivante :

— test01.net : la netlist à simuler,
— test01.in : un fichier d'input qui sera l'entrée standard du simulateur,
— test01.out : un fichier contenant la sortie supposée du simulateur,
— test01.rom (*optionnel*) : un fichier binaire chargé en tant que ROM du programme.

Le script comparera alors la sortie espérée à la sortie effective sur chaque test, indiquant les erreurs se produisant.

Tests actuels

Toute une batterie de tests se trouve déjà dans tests/ : on peut lancer ./runTests tests/* comme première vérification.

Le dossier tests/ contient plusieurs sous-dossiers, testant chacun des aspects différents du programme. Les noms des dossiers sont assez explicites sur leur contenu, toutefois détaillons-en deux :

— **singleGate**/ contient un test par équation possible, n'utilisant que cette commande (AND ou CONCAT par exemple), hors des commandes testées en profondeur par un dossier séparé (ram/ et rom/),
— **random**/ contient un ensemble de tests générés aléatoirement, contenant uniquement des TBit, de grande taille. L'output espéré a été généré à partir du simulateur de Nathanaël COURANT, développé indépendamment du mien, ainsi la probabilité que les deux codes donnent le même résultat faux est faible.

2.4 Utilisation du programme compilé

Supposons que circuit.bin soit un circuit compilé par Simcomp.

Entrée standard (stdin)

Le programme attend sur son entrée standard :

— Sur la première ligne, un *entier* n : le nombre de cycles à simuler,
— Sur les n lignes suivantes, séparées par un retour chariot (\n) : k caractères 0 ou 1, où k est le nombre de bits attendus par cycle. Si l'option -n a été passée, aucun retour de chariot n'est attendu.

Les variables attendues lors d'un cycle sont celles mentionnées dans la section INPUT du NetList, dans cet ordre ; une variable TBit attendant un bit, une variable de type TBitArray(p) attendant p bits.

Dans le cas d'un TBitArray(p), le premier bit entré est le bit d'indice 0 de la nappe de fils (*big endian*).

Dans le cas où *aucune* variable n'est variable d'entrée, le programme n'attend *pas* de retours de chariot, permettant de le lancer simplement avec echo "42" | ./circuit.bin pour 42 cycles.

Sortie standard (stdout)

Le programme écrit *à la fin de chaque cycle* sur la sortie standard un caractère (0 ou 1) par bit de sortie, traitant les TBitArray(p) comme pour l'entrée, c'est-à-dire en écrivant chaque bit un à un, en commençant par le bit de poids faible.

Chaque sortie de cycle est séparée par un retour de chariot (\n) *si l'option* -n n'a *pas* été passée au compilateur, sinon les lignes ne sont pas séparées du tout (plus rapide).

Paramètres du programme (argv)

Le programme peut éventuellement prendre en premier argument le chemin vers un fichier binaire qui sera chargé comme ROM. Si par exemple on souhaite charger dans la ROM un seul octet, 0b00101010, le fichier devra contenir le caractère ASCII « * ».

Si cette ROM n'est pas fournie et que le programme tente de faire des accès ROM, il échouera en lançant une exception non rattrapée.

Si cette ROM est fournie mais que le programme ne fait aucun accès ROM, elle sera simplement ignorée.

3 Fonctionnement et choix d'implémentation

3.1 Généralités

Le choix du C++ par rapport au C (qui semblait *a priori* meilleur puisque plus bas niveau comme langage de destination d'un compilateur) est principalement motivé par la bibliothèque standard comportant des std::bitset et des std::vector, à la fois bien optimisés et théoriquement sans bugs (limitant donc les risques d'erreurs dans le code final).

Dans sa généralité, Simcomp fonctionne comme conseillé. Le code est transformé par le lexer et parser (légèrement transformés) en un AST, puis un graphe ayant pour sommets les équations et arrêtes les dépendances est créé. Les dépendances des registres sont retournées ; seule la read_addr est considérée comme dépendance pour RAM. Puis, les équations sont topologiquement ordonnées, et sont une à une converties en C++ à l'intérieur d'un squelette de code. Le code de lecture de RAM est placé à l'endroit décidé par le tri topologique ; l'écriture en RAM à la fin de la boucle, juste avant l'affichage des valeurs de sortie.

3.2 Contenu des fichiers

- **checkNetlist.ml** : teste quelques propriétés sur la NetList fournie avant la compilation. Actuellement, teste qu'aucune variable n'est affectée deux fois dans un cycle (comportement indéterminé) et que les word_size et addr_size des accès ROM sont bien cohérents (puisqu'une seule ROM est gérée).
- **cpp.ml** : généré automatiquement par *genCpp.sh* à partir des fichiers dans *cpp/**, contient du code C++ sous forme de string.
- **cpp/skeleton/*** : le squelette pur de code allant autour du code généré (n'a aucun sens pris morceau par morceau).
- **cpp/[0-9]+_*** : des morceaux de code ayant un sens lorsque isolés, comme des fonctions, etc., insérables dans le code final.
- **depGraph.ml** : implémentation du graphe de dépendances entre les équations et du tri topologique.
- **genCode.ml** : génère les différentes parties de code (entrée/sortie, déclarations de variables, équations, ...).
- **genCpp.sh** : génère *cpp.ml* à partir du contenu de *cpp/**.
- **main.ml** : point d'entrée dans le programme, traite les paramètres et appelle le reste du code.
- **parameters.ml** : contient des constantes et références modifiables en tout point du code, utile pour gérer les paramètres du programme par exemple.
- **skeleton.ml** : assemble le code généré par *genCode.ml* et celui trouvé dans *cpp.ml* pour produire le code final.
- **transformNetlist.ml** : applique des transformations à la netlist pour gérer des cas particuliers et certaines optimisations (voir section 3.4). Actuellement, le cas du registre dont la sortie est une sortie de la netlist est géré par ce fichier en ajoutant un « fil » (une variable intermédiaire inutile) permettant à l'astuce de la dépendance inverse de fonctionner.

3.3 Modification apportées à Netlist

— Les constantes TBitArray étant mal gérées dans le cas où celle-ci commence par un zéro (*eg.* $00011 = [1;1] \neq [0;0;0;1;1]$), une modification permet d'entrer les nappes constantes sous la forme 0b00101010, interprétées correctement.
— Faute de documentation à ce sujet, j'ai choisi de considérer que pour une porte **mux**, le bit sélecteur était le *troisième* paramètre.
— Contrairement à ce que MiniJazz semble indiquer en compilant certains codes, j'ai choisi de supporter les opérations binaires sur les nappes de fil, qui sont en pratique effectuées sur des std :: bitset optimisés dans ce but. Il en va de même pour les not et les mux. Dans le cas de ce dernier, on impose tout de même que le sélecteur soit un TBit.

3.4 Optimisations apportées

— Regroupement de variables : j'ai remarqué qu'un code MiniJazz tel que x = (a xor b) and c ; y = (a xor b) and d produit *deux* variables intermédiaires de valeur a xor b ; ainsi cette optimisation repère les variables ayant la même équation et les regroupe en une seule variable. On itère le procédé jusqu'à ne plus rien simplifier (car un renommage de variables peut rendre deux équations identiques alors qu'elles ne l'étaient pas avant).

3.5 Représentation mémoire

Chaque variable de type TBit est un bool ; chaque TBitArray(p) est un std :: bitset<p> (permettant des opérations bitwise efficaces). Les mémoires (RAM et ROM) de word_size *ws* sont quant à elles des std :: vector<std:: bitset <ws> >.

La mémoire ROM est initialisée à false et à la taille du fichier ROM fourni, 0 sinon. Les mémoires RAM sont également initialisées à false, de taille 256 bits par défaut, modifiable en passant −−ramSize au compilateur.

Toutes les commandes ROM accèdent à la même mémoire ; par contre, chaque chaque commande RAM accède à une mémoire séparée. Une netlist de la forme

```
1 a = RAM (...)
2 [...]
3 a = RAM (...)
```

menant à un comportement indéterminé (dans quel ordre mettre les lignes ?), le compilateur renvoie une erreur.

3.6 Défauts actuellement constatés

— Les erreurs de compilation sont difficilement corrigées car Simcomp est particulièrement inexpressif (aucune localisation de l'erreur) : l'AST gagnerait à être décoré pour retenir la portion de code fautive.
— L'optimisation regroupant des équations identiques étant nécessairement quadratique (pour chaque couple d'équations, ...), la compilation peut être longue pour de grosses netlists : j'atteins les 6min18 sur une netlist aléatoire de 70000 équations. Ainsi, il est conseillé pour de trop grosses netlists de passer l'option −O0 au compilateur pour désactiver les optimisations.
Toutefois, regrouper les équations par type (Earg, Ereg, ...) avant d'appliquer cette optimisation a significativement diminué ce temps de compilation.