Projet système digital

I. Utilisation

Le simulateur fonctionne de la manière suivante : une netlist est compilée vers un fichier C, qui est ensuite compilé vers un fichier exécutable. Celui-ci peut ensuite être éxécuté pour simuler la netlist.

Pour compiler le compilateur, il suffit de faire `ocamlbuild compiler.native` ; celui-ci dépend de ocamllex et menhir comme le simulateur du TP 1.

Pour l'utiliser, il suffit de faire `./compiler.native [options] fichier.net` ; le compilateur produira ensuite un fichier C `fichier.c` et un fichier exécutable `fichier`. Les options acceptées sont les suivantes :

- `-print` : N'écrit que le code C du programme, sans faire appel à `gcc -O2` pour le compiler ensuite
- `-iomode` n : Choisit le mode d'entrée-sortie de la netlist compilée ; le mode 0 (par défaut) correspond à des entrées-sorties interactives, où l'exécutable demande la valeur de chaque variable à chaque cycle (en décimal pour les nappes de fils), et effectue également l'affichage en décimal à chaque fin de cycle. Le mode 1 correspond à des entrées-sorties consituées d'un caractère par bit d'entrée-sortie, qui doit être '0' ou '1'. Les sorties sont affichées dans le même format que l'entrée.
- `-n` n : Nombre de cycles à simuler ; si omis, supposé égal à $+\infty$.

Note : le compilateur peut prendre un temps assez long, principalement dû à l'appel de `gcc` ; ainsi, sur une netlist contenant environ 10^4 équations, le compilateur prend un temps de 0.2s pour produire un fichier C, tandis que la compilation de ce fichier par `gcc` prend de l'ordre de 1.5s, et 50s avec `gcc -O2`.

Le fichier exécutable prend lui autant d'arguments que le nombre de ROMs du circuit ; chacun étant un fichier binaire contenant le contenu initial des ROMs.

II. Fonctionnement

Le compilateur commence par ordonner les équations afin que celles-ci s'exécutent dans le bon ordre (on n'utilise pas une variable avant de la mettre à jour, avec un traitement spécial des registres : ceux-ci sont en effet modifiés à la fin). Ensuite, la traduction de chaque équation est ajoutée dans le corps de la boucle principale du programme.

Les nappes de fils sont stockées dans des entiers 64 bits ; en particulier, cela implique que les nappes de fils de plus de 64 bits ne sont pas supportées facilement. Par souci de simplicité, j'ai choisi de ne pas supporter ces nappes, qui ne sont pas très utiles en pratique.

III. Optimisations

On utilise bien sûr des opérations bit-à-bit lors d'opérations sur des nappes de fils. De plus, on garde une table de permettant de savoir où trouver chaque nappe de fils ou fil (i.e. dans quelle variable, et sa position interne à cette variable), ce qui permet de drastiquement diminuer le nombre d'opérations SLICE, SELECT et CONCAT. Le compilateur optimise finalement l'appel d'opérateurs sur chaque fil d'une nappe (ou de deux nappes) en un seul opérateur bit-à-bit, afin de simplifier encore plus le code produit. Ainsi, un morceau de code MINIJAZZ de ce type :

```
or_n<n>(a:[n], b:[n]) = (o:[n]) where
  if n = 0 then
    o = []
  else
    o = (a[0] or b[0]).(or_n<n-1>(a[1..], b[1..]))
  end if
end where

main(a:[10], b:[10]) = (o:[word]) where
  o = or_n<10>(a, b)
end where
```

sera traduit en :

```
o = a | b;
```

On effectue donc une simplification assez importante de la netlist de cette manière.