Projet NetList

Gabriel Desfrene

Voici mes quelques notes pour la phase 1 du projet “Microprocessor” du cours Systèmes numériques. Le code associé à ce projet est disponible sur un dépôt GitHub à l'adresse suivante :

`https://github.com/desfreng/netlist/tree/gabriel`

1 Langage NetList

Le langage que le simulateur accepte possède quelques extensions par rapport à celui proposé à l'origine. Ces extensions comprennent :

- Toutes les opérations sont étendues à des bus.
- Les constantes peuvent être données en base 10 à l'aide de la syntaxe suivante :

  `0d`⟨valeur en base 10⟩`:`⟨taille⟩

- De la même manière, les constantes peuvent être données en base 16 en préfixant leur valeur par `0x`.
- Par commodité, les constantes données en base 2 peuvent être préfixées par `0b`.
- Il est possible d'ajouter des commentaires dans les fichiers d'entrée. Ceux-ci commencent par `#` et s'étalent sur toute la fin de la ligne.

Une grammaire du langage accepté est donnée en figure 1. Des restrictions ont également été posées afin de garantir une consistance sur la taille des bus manipulés. Ainsi :

- Les opérations logiques binaires **AND**, **NAND**, **OR** et **XOR** doivent avoir des arguments de même taille.
- Les deux derniers arguments de l'opération **MUX** doivent être de même taille.
- Les opérations **SLICE** et **SELECT** doivent avoir des indices de bus cohérents avec la taille de l'argument.

Dans le cas où ces contraintes ne seraient pas respectées, le parseur codé refusera l'entrée. De plus, le parseur vérifiera que chaque équation est “bien formée”, c'est-à-dire que les deux membres ont bien la même taille.

La taille des arguments est déterminée de la manière suivante :

- La taille d'une variable est celle donnée lors de sa déclaration dans la section **VAR**.
- La taille d'une suite de chiffres 0 et 1 est la longueur de cette suite.
- Pour une constante donnée en base 10, la taille est nécessairement donnée après sa valeur suivie de “`:`”.
- Pour les constantes binaires et hexadécimales, la taille peut être indiquée de la même manière que pour la base 10. Si aucune indication n'est fournie, la taille est inférée à partir de la longueur de la suite de chiffres.

Si la taille donnée ne permet pas de représenter la valeur associée, le parseur générera une erreur.

⟨*program*⟩ ::= ‘INPUT’ ⟨*ref-list*⟩ ‘OUTPUT’ ⟨*ref-list*⟩ ‘VAR’ ⟨*decl-list*⟩ ‘IN’ ⟨*eqs*⟩ #

⟨*ref-list*⟩ ::= ⟨*ref-list-non-empty*⟩ | ε

⟨*decl-list*⟩ ::= ⟨*decl-list-non-empty*⟩ | ε

⟨*ref-list-non-empty*⟩ ::= ⟨*ident*⟩ ‘,’ ⟨*ref-list-non-empty*⟩ | ⟨*ident*⟩

⟨*decl-list-non-empty*⟩ ::= ⟨*ident*⟩ ⟨*size-spec*⟩ ‘,’ ⟨*decl-list-non-empty*⟩ | ⟨*ident*⟩ ⟨*size-spec*⟩

⟨*eqs*⟩ ::= ⟨*ident*⟩ ‘=’ ⟨*expr*⟩ ⟨*eqs*⟩ | ε

⟨*expr*⟩ ::= ⟨*arg*⟩
| ‘NOT’ ⟨*arg*⟩
| ‘AND’ ⟨*arg*⟩ ⟨*arg*⟩
| ‘NAND’ ⟨*arg*⟩ ⟨*arg*⟩
| ‘OR’ ⟨*arg*⟩ ⟨*arg*⟩
| ‘XOR’ ⟨*arg*⟩ ⟨*arg*⟩
| ‘MUX’ ⟨*arg*⟩ ⟨*arg*⟩ ⟨*arg*⟩
| ‘REG’ ⟨*ident*⟩
| ‘CONCAT’ ⟨*arg*⟩ ⟨*arg*⟩
| ‘SELECT’ ⟨*bus-size*⟩ ⟨*arg*⟩
| ‘SLICE’ ⟨*bus-size*⟩ ⟨*bus-size*⟩ ⟨*arg*⟩
| ‘ROM’ ⟨*bus-size*⟩ ⟨*bus-size*⟩ ⟨*arg*⟩
| ‘RAM’ ⟨*bus-size*⟩ ⟨*bus-size*⟩ ⟨*arg*⟩ ⟨*arg*⟩ ⟨*arg*⟩ ⟨*arg*⟩

⟨*arg*⟩ ::= ⟨*ident*⟩ | ⟨*bin-digits*⟩ | ⟨*bin-const*⟩ | ⟨*dec-const*⟩ | ⟨*hex-const*⟩

⟨*ident*⟩ ::= [‘a’-‘z’‘A’-‘Z’‘_’][‘a’-‘z’‘A’-‘Z’‘0’-‘9’‘_’]*

⟨*bin-digits*⟩ ::= [‘0’‘1’][‘0’‘1’]*

⟨*bin-const*⟩ ::= ‘0b’[‘0’‘1’][‘0’‘1’]* ⟨*size-spec*⟩

⟨*dec-const*⟩ ::= ‘0d’[‘0’-‘9’][‘0’-‘9’]* ‘:’ ⟨*bus-size*⟩

⟨*hex-const*⟩ ::= ‘0x’[‘0’-‘9’‘a’-‘f’][‘0’-‘9’‘a’-‘f’]* ⟨*size-spec*⟩

⟨*size-spec*⟩ ::= ‘:’ ⟨*bus-size*⟩ | ε

⟨*bus-size*⟩ ::= [‘1’-‘9’][‘0’-‘9’]*

FIGURE 1 – Grammaire acceptée par le simulateur pour les fichiers NetList. Axiome : *program*.

2 Fonctionnalités

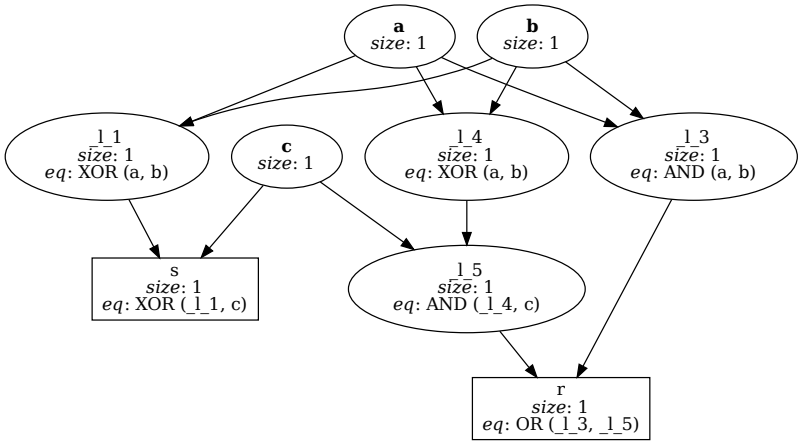
Le simulateur est doté de plusieurs fonctionnalités permettant la manipulation des NetList. Celles-ci sont sélectionnées lors de l'exécution via les arguments donnés au programme. Une description des combinaisons acceptées est disponible dans l'aide du programme. Pour l'obtenir, ajoutez l'argument `--help` en premier. Voici les fonctionnalités disponibles :

— L'affichage de la NetList parsée sur la sortie standard.
— L'export de la NetList sous la forme d'un graphe.
— L'affichage du résultat de l'ordonnanceur.
— La simulation de la NetList.

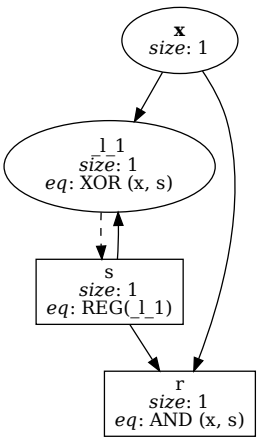
2.1 Export en graphe

Lors de l'exportation de la NetList sous forme de graphe, le fichier NetList donné en argument est converti en un graphe au format *Graphviz* sur la sortie standard. Vous pouvez utiliser un outil tel que `dot` pour exporter le graphe au format PDF. La figure 2 donne des exemples de graphes que retourne le programme. Les conventions suivantes sont utilisées lors de l'export en graphe :

— Un nœud ayant un nom en **gras** représente une variable d'entrée.
— Un nœud de forme rectangulaire représente une sortie.
— Les arcs pleins représentent les dépendances entre les variables de la NetList. Ces dépendances sont prises en compte par l'ordonnanceur.
— Les arcs en pointillés représentent les dépendances non prises en compte par l'ordonnanceur, telles que les dépendances induites par une opération **REG** ou **RAM** lors de l'écriture de la valeur en mémoire.



(a) `fulladder.net`



(b) `cm2.net`

FIGURE 2 – Quelques graphes produits par le programme.

2.2 La simulation

Pour simuler les NetList, certaines contraintes ont été définies :

— Les blocs **ROM**, identifiés par la variable dont ils déterminent la valeur, doivent nécessairement être initialisés avant la simulation. Le programme reportera une erreur dans le cas contraire.
— Les blocs **RAM**, identifiés par la variable dont ils déterminent la valeur, peuvent être initialisés avant la simulation. Dans le cas où aucune valeur n'est donnée, le programme affichera un avertissement et la mémoire est initialisé à 0.
— La première valeur des registres est fixée à 0.

La figure 3 donne le format requis pour les fichiers de l'entrée qui initialisent les blocs de mémoire.

⟨*input-file*⟩ ::= ⟨*block-list*⟩ #

⟨*block-list*⟩ ::= ⟨*block-list-non-empty*⟩ | ϵ

⟨*block-list-non-empty*⟩ ::= ⟨*block-def*⟩ ⟨*block-list-non-empty*⟩ | ⟨*block-def*⟩

⟨*block-def*⟩ ::= ⟨*ident*⟩ '`:`' ⟨*bus-size*⟩ '`=`' '`[`' ⟨*value-list*⟩ '`]`'

⟨*value-list*⟩ ::= ⟨*value*⟩ '`,`' ⟨*value-list*⟩ | ⟨*value*⟩

⟨*value*⟩ ::= ⟨*bin-digits*⟩ | ⟨*bin-const*⟩ | ⟨*dec-const*⟩ | ⟨*hex-const*⟩

FIGURE 3 – Grammaire acceptée par le simulateur pour les fichiers d'entrée. Axiome : *input-file*.

Ainsi, pour le fichier `ram.net`, la structure suivante convient afin d'initialiser la mémoire de l'opération **RAM** :

```
o : 4 = [ 0xf, 0b1001, 0d3:4, 0110 ]
```

Si aucune limite de cycles n'est spécifiée, le programme s'arrête lorsqu'il reçoit un *SIGTERM* ou un *SIGINT*.

3 Difficultés

Lors de la réalisation de ce projet, quelques difficultés ont été rencontrées :

- L'apprentissage des versions de C++ récentes ne s'est pas fait en douceur. Ce projet a permis d'apprendre et de manipuler des constructions en C++ plus récentes telles que les *templates*[1].
- La mise au propre de la grammaire du langage NetList parsée a également posé quelques soucis. Puisque le parseur a été réimplémenté en faisant attention à la cohérence du fichier donné en entrée, des problèmes sont apparus pour certaines opérations telles que **CONCAT** ou **SLICE** lorsqu'elles prennent des constantes comme argument.
- Il a été difficile[2] de faire en sorte que MSVC accepte de compiler ce projet. La mise en place de builds automatiques a beaucoup aidé à vérifier que ce projet compile dans de bonnes conditions.

1. Je n'avais pas fait de C++ depuis **très** longtemps...
2. Voir la liste des commits.