Manuel de MiniJazz

Cédric Pasteur <cedric.pasteur@ens.fr>

5 décembre 2012

1 Le langage MiniJazz

1.1 Description du langage

Le langage MiniJazz est un langage de description de circuits digitaux synchrones. Il permet de décrire des circuits de façon récursive et hiérarchique et d'utiliser des nappes de fils. Voici par exemple la description d'un *full-adder* :

```
fulladder(a,b,c) = (s, r) where
  s = (a ^ b) ^ c;
  r = (a & b) + ((a ^ b) & c);
end where
```

Bloc

Un circuit est décrit par un ensemble de blocs. Un bloc est défini par son nom `f`, ses entrées `a_1, .., a_p`, ses sorties `o_1, ..., o_q` et un ensemble `D` d'équations définissant les sorties en fonction des entrées. Il est possible de définir des variables locales sans les déclarer.

```
f(a_1, ..., a_p) = (o_1, ..., o_q) where
  D
end where
```

Les équations sont soit de la forme `x = e` où e est une expression, soit l'instantiation d'un autre bloc de la forme `(y_1, ..., y_q)=g(x_1, .., x_p)`, soit un bloc conditionnel **`if`** `c` **`then`** `D1` **`else`** `D2` **`end if`** où `c` est une expression statique booléenne.

Les opérateurs prédefinis sont l'inverseur (`not`), le *et* logique (`&` ou `and`), le *ou* logique (`or` ou +), le *ou exclusif* (`xor` ou `^`), `nand` et le multiplexeur 1 bit (**`mux`**). Les constantes `true`(ou `1`) et `false` (ou `0`) sont également disponibles.

Nappes de fils

On peut déclarer une variable `x` (entrée ou sortie) comme étant une nappe de `n` fils avec la syntaxe `x : [n]`. On peut alors accéder au iéme fil avec `x[i]` (les indices commencent à 0) ou extraire une sous nappe avec `x[i1 .. i2]`. On peut aussi écrire `x[i1..]` (resp. `x[..i2]`) comme raccourci de `x[i1..n-1]` (resp. `x[0..i2]`) si le tableau est de taille `n`. Il est également utile de concaténer des nappes avec l'opérateur `..`

Voici par exemple un bloc qui décale sa sortie de 1 bit vers la gauche :

```
shift_left(a:[4]) = (o:[4]) where
  o = a[1..3] . 0;
end where
```

Paramétricité

La taille d'un tableau est donné par une *expression statique*, qui est soit une constant globale, soit un paramètre d'un bloc, soit un opérateur statique (`+`, `-`, `*`, `/`, `^`, `=`, `<=`) appliqué à deux expression statiques. Les constantes globales sont déclarées par :

```
const n = 32
```

Les paramètres statiques d'un bloc sont donnés entre crochets après son nom. Combinés avec les blocs conditionnels et la récursion, ils permettent de créer des circuits décrits de façon récursive. Un autre élément utile est la valeur `[]`, qui est une nappe de taille 0. On peut par exemple définir un additionneur n bits à partir d'un `fulladder` et d'un additionneur $n-1$ bits :

```
adder<n>(a:[n], b:[n], c_in) = (o:[n], c_out) where
  if n = 0 then
    o = [];
    c_out = 0
  else
    (s_n1, c_n1) = adder<n-1>(a[1..], b[1..], c_in);
    (s_n, c_out) = fulladder(a[0], b[0], c_n1);
    o = s_n . s_n1
  end if
end where
```

Primitives mémoire

Il est possible de déclarer des blocs mémoire, ROM ou RAM, avec la syntaxe suivante :

```
o1 = ram<addr_size, word_size>(read_addr, write_enable,
    write_addr, write_data);
o2 = rom<addr_size, word_size>(read_addr);
```

`word_size` est la taille des mots mémoires, c'est-à-dire le nombre de bits lus à chaque cycle, qui est aussi la taille de la sortie. `addr_size` est la taille des adresses en mémoire, i.e. la mémoire contient $2^{\texttt{addr\_size}}$ mots. Une ROM prend en entrée une adresse et renvoie le mot stocké à cette adresse. Une RAM fait le même chose mais prend aussi en entrée un bit indiquant s'il faut écrire (`write_enable`), suivi de l'adresse et du mot à écrire. La RAM fonctionne comme un registre : on peut lire et écrire au même emplacement de la mémoire (i.e. `read_addr =write_addr` et `write_enable =1`) sans qu'il y ait de boucle combinatoire. La valeur lue sera celle dans la mémoire à l'instant précédent.

1.2 Compilateur

Le compilateur MiniJazz génère à partir d'un programme une net-list non ordonnée, ne contenant plus de blocs et dont chaque équation ne contient plus qu'une seule opération. Il faut spécifier le bloc principal du circuit avec l'option `-m` (par défaut, il s'appelle `main`).

```
> ./mjc.byte -m mon_bloc mon_fichier.mj
```

Le fichier généré est un simple fichier texte décrivant une *net-list*, avec l'extension `.net`, dont le contenu est décrit dans la partie suivante.

2 Le langage de *net-list*

2.1 Syntaxe concrète

Une *net-list* a la forme suivante :

```
INPUT inputs
OUTPUT outputs
VAR vars IN
  D
```

`inputs` et `outputs` sont les entrées et sorties du circuit, sous forme d'une liste d'identifiants. `vars` donne le type des entrées, sorties et variables locales (avec la même syntaxe que pour MiniJazz). Enfin, *D* représente la liste des portes du circuit. Chaque équation ne contient plus qu'une seule opération élémentaire, soit une opération booléenne (`OR`, `XOR`, `AND`, `NAND`), soit définir un registre (`REG`) ou une mémoire (`RAM`, `ROM`), soit une opération sur des nappes de fils (`SELECT`, `SLICE`, `CONCAT`). Voici par exemple la *net-list* générée par le compilateur pour le circuit `Fulladder` :

```
INPUT a, b, c
OUTPUT s, r
VAR
  _l_1, _l_3, _l_4, _l_5, a, b, c, r, s
IN
r = OR _l_3 _l_5
s = XOR _l_1 c
_l_1 = XOR a b
_l_3 = AND a b
_l_4 = XOR a b
_l_5 = AND _l_4 c
```

2.2 Syntaxe abstraite

La syntaxe abstraite du langage de *net-list*, c'est-à-dire l'ensemble des types représentant un circuit, est donnée dans le fichier `netlist_ast.ml`. Un analyseur lexical (`netlist_lexer.mll`) et un analyseur syntaxique (`netlist_parser.mly`) permettent de traduire un fichier texte dans la syntaxe concrète en la structure de données correspondante. On utilisera pour cela directement la fonction suivante définie dans le fichier `Netlist.ml`, qui attend un nom de fichier et renvoie le circuit correspondant :

```
val read_file : string -> Netlist_ast.program
```

Pour faire l'opération inverse, on utilisera la fonction suivante :

```
val print_program : out_channel -> Netlist_ast.program -> unit
```

Par exemple, pour écrire un programme dans un fichier `filename`, on pourra faire :

```
let oc = open_out filename in
print_program oc p
```