Rapport sur le simulateur de netlist

Théophile Wallez

06/11/2016

1 Spécifications

1.1 Déclaration des variables

Un nom de variable doit être reconnue par l'expression régulière suivante : `[_a-z][_a-z0-9]*`. Un nom de variable peut être suivi de "`:` n" où n indique que la variable est une nappe de fils à n fils. Un fil est assimilé à une nappe de un fil.

Après les mots-clés `INPUT`, `OUTPUT`, `VAR` il y a une liste de variables séparées par des virgules. Si une taille de nappe est donnée deux fois, la première valeur donnée est prioritaire.

Les variables données après `INPUT` sont des variables dont la valeur sera demandée à chaque cycle.

Les variables données après `OUTPUT` sont des variables dont la valeur sera affichée à chaque cycle.

Par la suite, si `a` est une variable, on va noter $|\texttt{a}|$ la taille de sa nappe.

1.2 Déclaration des portes logiques

La liste des portes logiques est donnée après le mot-clé `IN`, où chaque ligne contient une porte logique.

- `a = OR b c` : assigne à `a` la valeur du "ou" bit à bit de `b` et `c`. Il est requis que $|\texttt{a}| = |\texttt{b}| = |\texttt{c}|$
- `a = XOR b c` : pareil, avec une porte "ou exclusif"
- `a = AND b c` : pareil, avec une porte "et"
- `a = NAND b c` : pareil, avec une porte "non-et"
- `a = NOT b` : assigne à `a` la valeur du "non" bit à bit de `b`. Il est requis que $|\texttt{a}| = |\texttt{b}|$
- `a = REG b` : assigne à `a` la valeur de `b` au cycle précédent. Au premier cycle, `a = 0`. Il est requis que $|\texttt{a}| = |\texttt{b}|$
- `a = RAM` ad w `ra we wa d` : assigne à `a` les bits de la RAM de l'adresse `ra` à `ra`$+w-1$ (inclus). Si `we`$= 1$, les bits de la RAM de l'adresse `wa` à `wa`$+w-1$ sont assignés à `d`, ce changement étant visible seulement au cycle suivant. Il est requis que $ad = |\texttt{ra}| = |\texttt{wa}|$, $w = |\texttt{a}| = |\texttt{d}|$ et $|\texttt{we}| = 1$, et que 8 divise `ra` et `wa`.
- `a = ROM` ad w `ra` : assigne à `a` les bits de la ROM de l'adresse `ra` à `ra`$+w-1$. Il est requis que $ad = |\texttt{ra}|$, $w = |\texttt{a}|$, et que 8 divise `ra`
- `a = MUX b c d` : si `c`$= 1$, assigne à `a` la valeur de `b`, sinon assigne à `a` la valeur de `d`. Il est requis que $|\texttt{a}| = |\texttt{c}| = |\texttt{d}|$ et $|\texttt{b}| = 1$
- `a = SELECT` i `b` : assigne à `a` la valeur du ième bit de `b` (on compte à partir de 0). Il est requis que $|\texttt{a}| = 1$ et $i < |\texttt{b}|$.
- `a = SLICE` i j `b` : assigne à `a` les valeurs du ième bit au jème bit de `b`. Il est requis que $i \leq j < |\texttt{b}|$, $|\texttt{a}| = j - i + 1$

— `a = CONCAT b c` : assigne à `a` la concaténation des nappes de `b` et `c`. Les bits de poids faible de `a` sont ceux de `c`. Il est requis que $|a| = |b| + |c|$

2 Détails sur l'implémentation & difficultées rencontrées

2.1 Implémentation de l'interpréteur

J'ai programmé en C++. Le lexer utilise la bibliothèque `lexertl`, et j'ai écrit le parseur à la main. Le programme contient en fait deux simulateurs : un interpréteur basique, et un interpréteur faisant de la compilation à la volée en utilisant la bibliothèque `asmjit`, pour plus de performances.

2.2 Stockage des variables

Je stocke les variables sur des entiers 64-bit : cela a pour conséquence que les nappes ne peuvent pas contenir plus de 64 fils. J'estime qu'il y aura suffisemment peu de variables pour que ça consomme quand même peu de mémoire, et que ça tienne dans le cache. Cela permet d'utiliser directement les opérations bit à bit du `C++`.

Un problème qu'il peut avoir est que certains bits de poids fort qui ne représentent pas la valeur de la nappe soient à 1 (comme après une porte `NOT` ou `NAND`). Cela fait qu'il faut faire un masque sur les adresses données à la RAM et la ROM, et au bit de sélection de l'instruction `MUX`. Concernant `MUX`, ça ne coûte rien de faire le masque (il suffit de faire `testq b, 1` au lieu de `testq b, b`), mais cela coûte une instruction de plus pour la RAM et la ROM.

2.3 Calcul des masques

Pour faire un masque pour les n premiers bits, je faisais $(1 << n) - 1$. Cela fonctionne correctement pour des petites valeurs de n, mais c'est une opération à comportement indéterminé lorsque $n \geq 64$, et c'est un problème lorsque j'ai des nappes de 64 fils. En effet, `g++` compile le décalage avec `sal` au lieu de `shl` ce qui fait que le 1 retourne au début et alors $1 << 64 = 1$. J'ai donc codé une fonction qui prends ce cas particulier en compte.

2.4 Adresses de la RAM et la ROM

J'ai décidé de n'accepter que les addresses multiples de 8 dans la RAM et la ROM. Cela a pour conséquence que je n'ai pas besoin de faire de calculs avec des bitshift et que le code est plus performant. De plus, j'estime que ce n'est pas utile de supporter autre chose que des adresses multiples de 8.

J'aurais pu demander à ce que l'unité des addresses soit en octet et pas en bit, mais je ne l'ai pas fait afin de rester cohérent avec les simulateurs des personnes avec qui je vais faire le processeur.

La taille du mot n'est pas très importante car je lis 64 bits dans tous les cas : les bits de poids fort seront de toute façon ignorés si la taille de la nappe est petite.

L'interpréteur basique vérifie que l'adresse est bien un multiple de 8 (avec un `assert`), tandis que l'interpréteur faisant du JIT ne vérifie pas, il effectue seulement une division par 8.

2.5 Endianess

L'endianess des instructions `RAM` est `ROM` est la même que celle de l'ordinateur hôte. Dans la majorité des cas, c'est little-endian (processeurs x86 ou x86-64).

2.6 Gestion des REG

J'ai cherché un moyen efficace de gérer les `REG`. Une première idée naïve est de se dire qu'il suffit de les simuler au tout début, mais ça ne fonctionne pas car il peut y avoir des cycles : `x = REG y` et `y = REG x`.

J'ai trouvé plusieurs méthodes pour implémenter ça de façon correcte :

— copier le tableau contenant les variables à chaque cycle

— détecter les cycles d'assignation et les "couper" avec une variable temporaire (comme quand on échange la valeur de deux variables)

— Avoir tout le temps deux tableaux : un qui contient les valeurs des variables au cycle courant, et l'autre les valeurs au cycle précédent, et on alterne leurs rôles à chaque cycle.

J'ai décidé de partir avec la dernière option, en partant du principe qu'il n'y aurait pas beaucoup de variables et que ça ne poserait pas de problème du point de vue du cache. C'est aussi une solution optimale en terme d'instructions exécutées sur le processeur hôte.

3 Utilisation pratique

On lance la commande `./bin/NetlistSim` avec les arguments suivants :

— `-i` ou `--input` suivi du nom du fichier contenant la netlist (requis)

— `-r` ou `--rom` suivi du nom du fichier contenant la ROM (optionel : par défaut la ROM est vide)

— `-s` ou `--ramsize` suivi d'un nombre désignant la taille de la RAM souhaitée (optionel : par défaut 0)

— `-n` ou `--nbiter` suivi d'un nombre désignant le nombre d'itération souhaité (optionel : par défaut fait un nombre infini d'itération)

— `-j` ou `--jit` active l'utilisation de l'interpréteur JIT