Un peu d'assembleur dans notre processeur

Jean-Marie Madiot
Hugo Feree

1 Compilateur

1.1 Modification du fichier asm.cc

Nous avons complété les instructions à l'aide des fonctions `read_reg` et `read_imm8_or_label` fournies. Nous avons ajouté les instructions `ldlline` et `ldhline` qui chargent dans l'accumulateur respectivement les bits de poids faible et de poids fort de l'adresse mémoire de la ligne courante.

Pour cela, on passe la ligne courante (`current_address`) en argument à `read_op`.

1.2 Data

(Ou comment écrire dans la mémoire directement)

Au lieux d'écrire $3n$ lignes de code pour stocker n entiers en mémoire à l'aide de multiples registres, nous avons décidé de pouvoir utiliser le programme comme donnée. c'est à dire qu'à certains endroits du programme, un mot de 16 bits n'est plus considéré comme une instruction, mais comme une donnée. Ainsi, en écrivant :

```
data 10 15 25 5 21 24 37 14 15 5 12 0
```

le compilateur écrit les instructions codées par les entiers 10, 15, 25, etc. Ainsi, on peut considérer ces instructions comme des données qu'on pourra lire en connaissant leur adresse dans la mémoire grâce à `ldlline` et `ldhline`.

Les données ci-dessus représentent la chaîne de caractères "JOYEUX NOEL" suivie du caractère nul pour marquer la fin de la chaîne comme en C.

2 Simulateur

Hormis la suppression de quelques variables et `includes` ainsi que la modification de `rom.data`, nous voudrions insister sur les points suivants.

2.1 Flags

Nous avons renommé les flags, ce qui permettait de gérer les prédicats plus clairement, notamment en gérant les overflows différemment. Le comportement est peut-être différent du simulateur initial, mais nous paraît plus cohérent.

Nous avons également réécrit la gestion des flags dans les instructions `add`, `sub`, `cmp` et `mul`.

2.2 Fréquence d'actualisation

Le simulateur de base étant beaucoup trop lent, nous avons augmenté la période de rafraîchissement. Celle-ci est modifiable lors de l'appel du simulateur à l'aide de l'option -a suivie du nombre d'instructions entre deux rafraîchissements.

2.3 `rts`

Nous avons permuté les deux lignes de `rts`, ce qui nous semblait plus logique : `mem[--r127]` au lieu de `mem[r127--]`, ce qui reste cohérent avec les sauts à modification de pile.

2.4 `mul`

Nous avons reprogrammé la multiplication sur 16 bits au lieu de 8, et avec une gestion correcte des overflows.

2.5 `cmp`

En réutilisant le code de `sub`, nous avons fait en sorte que `cmp` gère la pleine comparaison au lieu de la simple égalité.

3 jmrisp, jmri, jmi

Il est important de noter qu'on a modifié le jeu d'instructions pour faire fonctionner les fichiers `plot.asm` et `plottext.asm` de façon cohérente.

Modifications :

- jmi : ne modifie pas la pile d'appels, jugé inutile, et inutilisé. Nous l'avons enlevé.
- jmri : ne modifie pas la pile d'appels, très utile pour les sauts courts conditionnels, les boucles, etc.
- jmrisp : similaire à jmri mais modifie la pile d'appels, très utiles pour les fonctions avec rts.

Evidemment ces changements n'étaient pas indispensables, mais ils facilitent l'écriture et la compilation des programmes. Pour éviter ces changements, on aurait pu systématiser mieux les `ldlline` et `ldhline` pour pouvoir utiliser jmp sans avoir à changer le jeu d'instructions.

A noter que les autres fichiers `.asm` fonctionnent sans ce changement. (Modifier simplement le fonctionnement de `jmri` en le faisant modifier la pile d'appels fait exploser la pile.)

A posteriori il aurait été préférable de passer plus de temps sur le compilateur `asm.cc` sans changer le jeu d'instructions, car la multiplication des appels à `jmri` faisait sauvegarder la ligne en cours dans un registre manuellement (laborieux).

4 Un peu de maths

Le fichier `polynôme.asm` se compose de l'affichage d'un entier, de l'écriture du polynôme dans la mémoire, de l'affichage du polynôme et de son évaluation en un point.

Pour plus de détails, voir les commentaires dans le fichier même.

5 Joyeux Noël !

Le fichier `neige.asm` affiche ce que l'on veut. Pour cela, le fichier `canon.ml` nous a permis de générer automatiquement le code qui gère l'affichage de chaque flocon.