Compilation

TP 5 : Traduction dirigée par la syntaxe

A. Isoard & C. Alias

Il est maintenant temps de traduire des programmes vers le code assembleur que l'on a vu au `TP0`. Plus précisément, on va tenter de compiler en *une seule passe*, le code assembleur sera produit directement par le parseur (sans passer par une *représentation intermédiaire*).

Rappel des instructions du code assembleur visé

- **add $\mathbf{r_{dest}}, \mathbf{r_1}, \mathbf{r_2}$** : additionne le contenu des registres r_1 et r_2, et place le résultat dans le registre r_{dest}.
- **sub $\mathbf{r_{dest}}, \mathbf{r_1}, \mathbf{r_2}$** : calcule $r_1 - r_2$ et place le résultat dans le registre r_{dest}.
- **ld $\mathbf{r_{dest}}, [\mathbf{r_{base}} + \mathbf{imm7}]$** : charge dans le registre r_{dest} la donnée en mémoire à l'adresse $r_{base} + imm7$, où imm7 est un entier 7 bits. On parle aussi de *valeur immédiate*, puisque l'entier est immédiatement disponible dans l'instruction.
- **st $\mathbf{r_1}, [\mathbf{r_{base}} + \mathbf{imm7}]$** : stocke la valeur du registre r_1 dans la mémoire à l'adresse $r_{base} + imm7$.
- **ble $\mathbf{r_1}, \mathbf{r_2}, \mathbf{imm7}$** : si $r_1 \leq r_2$, saute à l'instruction située à l'adresse $pc + imm7 + 1$. (sinon, on passe à l'instruction suivante située à l'adresse courante plus 1). imm7 peut être négatif (en utilisant le complément à 2), ce qui permet les sauts en arrière. Cette instruction permet d'implémenter la boucle `for`, la boucle `while` et le `if`.
- **ldi $\mathbf{r_{dest}}, \mathbf{imm8}$** : écrit l'entier 8 bits imm8 dans le registre r_{dest}.
- **ja $\mathbf{r_1}, \mathbf{r_2}$** : saute à l'adresse 13 bits définie par r_2 pour les 8 bits de poids faible et par r_1 pour les 5 bits restants (de poids fort).
- **j imm13** : saute à l'adresse 13 bits imm13, où imm13 est un entier 13 bits. Cette instruction, avec `ja`, permet d'implémenter les appels de fonctions.

Il n'y a que 8 registres pouvant contenir des entiers signés sur 8 bits. Les adresses mémoires sont également sur 8 bits (maximum : 255) et il ne peut avoir plus de 8192 instructions dans le programme.

Description des classes du compilateur

Comme d'habitude, on reprend le compilateur du TP précédent. Vous devriez reconnaître pas mal de choses. Voilà un bilan des fichiers présents :

- **lexer.l**. Analyseur lexical (pas de changement)
- **parser.ypp** : Grammaire de notre sous-ensemble du langage C.
- **(Nouveau) Attributes.h/.cc** : Structure de données pour stocker les informations relatives aux expressions gauches (`lhs`) et droites (`rhs`). Essayez de deviner à quoi correspondent les attributs de ces classes ?
- **(TP3) Type.h/.cc** and `SymbolTable.h/.cc` : Classes utilisées pour (notamment) le type-checking. La table des symboles implémente l'environnement ρ, qui associe à chaque variable (argument ou variable locale) le temporaire dans lequel elle est stockée.

– **(Nouveau) Label.h/.cc** : Gère une multitude de compteur, afin d'avoir toujours des labels de noms différents dans le programme assembleur.
– **(Nouveau) Register.h/.cc** : Produit des variables temporaires "fraîches" (improprement appelées registres).
– **(Nouveau) CodeDigmips.h/.cc** : Contient les fonctions qui émettent le code assembleur sur la sortie standart.

Exercice 1. *À vos marques... prêt... Générez !*

Manip.

- **Jetez rapidement un coup d'œil** sur `Label.h/.cc` et `Temporary.h/.cc`. Essayez de créer des labels et des variables temporaires.
- *(Idiomes)* **Ouvrez `CodeDigmips.h/.cc`** et complétez les trous dans la macro `cjump`.
- *(Expressions/Conditions)* **Ouvrez `parser.ypp`**. **Complétez les trous** dans les parties `1/ Expressions 2/ Conditions`, en vous inspirant des autres règles déjà complétées.
- *(Contrôle)* **Allez à la partie** `3/ Statements` et implémentez les parties manquantes des traduction des contrôles `while` et `for`. En cas de panne d'inspiration, regardez du côté du `if/then/else`
- *(Allocation mémoire)* `Ouvrez Type.cc` et étudiez la fonction `allocate()`.
- *(Fonctions)* Trouvez où sont traduites les fonctions. Comment ρ est il construit ? Comment est-il utilisé dans les expressions ? Comment le résultat est-il passé ?

Exercice 2. *De l'utilisation du code compilé*

Manip.

- Faites tourner votre compilateur sur les exemples fournis dans le répertoire test.
- Le code produit peut-il être directement envoyé au simulateur sous **diglog** ? Que manque-t-il ?