Compilation

TP 4 : Compilateur simple

G. Iooss & C. Alias

Précédemment, on a vu que l'on pouvait lire et comprendre un code source (en utilisant un *lexeur* (`TP1`) et un *parseur* (`TP2`)) et que l'on peut vérifier si un programme avait un sens (via le *typage* (`TP3`)).

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*, càd 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 $\mathbf{r_{dest}}$, 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 :

- (TP1) `lexer.l` : Lexeur (pas de changement)
- (TP2) `parser.ypp` : Parseur (la grammaire a été même simplifiée par rapport au TP3)
- (Nouveau) `Attributes.h/.cc` : Structure de donnée 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. Une table qui associe les variables locales à leur position dans la pile apparaît dans `SymbolTable.h/.cc`. De même, la méthode `allocate` de `Type.h/.cc` (qui permet d'allouer un espace de taille *taille_du_type* dans la mémoire) a été rajouté.
- (Nouveau) `Label.h/.cc` : Gère une multitude de compteur, afin d'avoir toujours des labels de noms différents dans le programme assembleur.
- (Nouveau) `Temporary.h/.cc` : Gestion de l'allocation des variables temporaires (dans les registres ou dans la pile).
- (Nouveau) `CodeDigmips.h/.cc` : Contient les fonctions qui génèrent le code assembleur (càd, affichent l'instruction correspondante sur `cout`).

La grammaire du parseur a été modifiée de la façon suivante, par soucis de simplicité [1] :

- Il ne peut pas avoir d'appel de fonction.
- Il ne peut y avoir qu'une seule fonction dans le programme.

1. Comprendre : vous n'avez pas encore vu le cours correspondant

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

Manip.

- **Jetez rapidement un coup d'œil** sur `Label.h/.cc` et `Temporary.h/.cc`. Essayez de créer et d'allouer 6 `Temporary`. Vérifier que l'on commence bien à allouer sur la pile à partir du cinquième (aux adresses `ARP+shift`).
- *(Génération de code)* **Ouvrez `CodeDigmips.h/.cc`** et complétez les trous dans les fonctions de `Cfg.cc`. Dans le cas d'un `Temporary` qui est dans la pile, pensez à faire le `st` correspondant.
- *(Conditions)* Dans `CodeDigmips.cc` complétez la fonction `cjump`.
- *(Expressions)* **Ouvrez `parser.ypp`**. Dans la partie de gestion des expressions (`1/`, à partir de la ligne 219), complétez les trous en vous inspirant des autres règles pour les expressions. Notamment, essayez de comprendre ce que l'on fait dans les cas déjà complétez (qui peuvent être compliqué pour les tableaux notamment)
- *(Contrôle)* `Dans parser.ypp` allez à la partie `3/` (`Statements`) et implémenter 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()`. Faites tourner cet algorithme manuellement sur une matrice d'entier `int[2][2]` pour voir quel est l'état de la pile après son allocation.

Discussion *De l'utilisation du code compilé*

Manip.

- Faites tourner votre compilateur sur l'exemple fourni `test/test.c` (ou amusez-vous avec pour vérifier la validité du code assembleur produit).
- Le code produit peut-il être directement envoyé au simulateur sous **diglog** ?
- Proposez quelques idées d'améliorations pour rendre ce code exploitable.
- Comment es-ce qu'on pourrait rendre plus rapide le code assembleur produit ?
- Prenez un air pensif et fixez un point à l'horizon en vous disant qu'il y a encore beaucoup à faire...

Exercice bonus : multiplication et division

Essayez d'implémenter la multiplication et la division avec les opérations du processeur. Pour éviter une explosion du code, il est utile d'avoir une sorte de "procédure" `mult` (appel par saut à un label, la procédure étant généré au début du code). Réfléchissez à un moyen pour faire passer les arguments aux fonctions et faites l'implémentation correspondante [2].

2. Encore une fois j'offre gracieusement un Kinder Bueno à toute personne ayant terminé cet exercice dans les 2 heures du TP