TP numéro 5 : Allocation de registres

Assembleur – Compilation, ENSIIE

Semestre 3, 2021–22

L'objectif de ce TD est d'implémenter l'allocation de registres comme vue en cours, en calculant la durée de vie des variables, en construisant le graphe d'interférence et en le coloriant. Pour simplifier le travail demandé, on ne considèrera que des séquences d'instructions sans branchement.

Vous trouverez à l'adresse `http://www.ensiie.fr/~guillaume.burel/compilation/TP5.tar.gz` une archive contenant les fichiers nécessaires à ce TD. Pour vous faciliter la vie, une partie du code vous est déjà fournie sous forme d'une bibliothèque OCaml. Vous trouverez à l'adresse `http://www.ensiie.fr/~guillaume.burel/ocamlriscv/` la documentation associée à cette bibliothèque, qui se décompose en 6 modules :

Register : définition des variables (registres et pseudo-registres). On y trouvera également des fonctions pour manipuler des ensembles de (pseudo-)registres.

Riscv : définition des instructions RISC-V, en fonction du nombre de registres et de constantes utilisées.

Riscv_lexer et **Riscv_parser :** un analyseur syntaxique pour assembleur RISC-V. Pour récupérer la liste `l` des instructions RISC-V depuis l'entrée standard, on pourra utiliser le code suivant :

```
let lexbuf = Lexing.from_channel stdin in
let l = Riscv_parser.s Riscv_lexer.scan lexbuf in
  ...
```

On rappelle qu'on suppose qu'il n'y a pas de branchement dans le code passé en entrée, les instructions sont donc dans l'ordre de leur exécution.

Coloring : coloriage pour les graphes d'interférence. On associe à chaque registre soit un registre réel soit un spill.

Inter_graph : primitives pour manipuler des graphes d'interférence. On notera la possibilité d'afficher les graphes dans Firefox à l'aide des fonctions `print_graph` et `print_graph_coloring`.

Un `Makefile` est également disponible dans l'archive. Pour l'utiliser, modifier la ligne `OBJS=exemple.cmo` pour y mettre les noms des fichiers `.cmo` à produire pour votre programme.

Suivant votre installation, il sera peut-être nécessaire de recompiler la bibliothèque fournie à l'aide de `make -C ocamlriscv/ clean all`

1 Analyse de durée de vie, graphe d'interférence

Comme le graphe de flot de contrôle est supposé linéaire (on a une liste d'instructions), l'analyse de durée de vie peut se faire en une seule passe : on part de la dernière instruction avec un ensemble de variables vivantes vides, et on remonte les instructions en mettant à jour cet ensemble de variables vivantes.

1. Écrire une fonction `update_live_variables` de type `Riscv.riscv -> Register.reg_set -> Register.reg_set` qui prend une instruction RISC-V, les variables vivantes après cette instruction, et qui retourne les variables vivantes avant. On utilisera l'équation

$$\text{Vivantes}_{\text{avant}}(i) = (\text{Vivantes}_{\text{après}}(i) \setminus \text{Tuées}(i)) \cup \text{Engendrées}(i)$$

2. En déduire une fonction `liveness_analysis` de type `Riscv.riscv list -> Register.reg_set` qui prend un liste d'instructions RISC-V et qui renvoie l'ensemble des variables vivantes en début de programme en supposant qu'aucune variable n'est vivante en fin de programme. On pourra utiliser `List.fold_right`.

On peut en fait construire le graphe d'interférence au moment de l'analyse de durée de vie.

3. Modifier `update_live_variables` pour mettre également à jour le graphe d'interférence. Elle aura donc le type

```
Riscv.riscv -> (Inter_graph.graph * Register.reg_set)
  -> (Inter_graph.graph * Register.reg_set)
```

   On rappelle qu'une variable interfère avec les variables vivantes après les points où elle est définie. On ajoutera des arêtes de préférence en cas de `move`.

4. Modifier `liveness_analysis` pour renvoyer le graphe d'interférence produit par un programme. Elle aura donc pour type `Riscv.riscv list -> Inter_graph.graph`.

2 Coloriage de graphe

5. Écrire une fonction `trivially_colorable` de type `Inter_graph.graph -> Register.reg -> bool` qui dit si un registre est trivialement colorable dans un graphe, c'est-à-dire qu'il a strictement moins de voisins que le nombre de registres disponibles. On utilisera la fonction `Register.nb_regs` pour connaître ce nombre de registres disponibles.
6. Écrire une fonction `george_criterion` de type `Inter_graph.graph -> Register.reg -> Register.reg -> bool` qui implémente le critère de George : deux nœuds vérifient ce critère si tous les voisins non trivialement colorables de l'un sont voisins de l'autre (mais pas forcément réciproquement).

7. Implémenter l'algorithme de coloriage de graphe de George et Appel comme vu en cours : on définira donc cinq fonctions mutuellement récursives `colorier` `simplifier` `fusionner` `geler` et `spiller` de type `Inter_graph.graph -> Coloring.coloring`.
   On affichera l'état du graphe à chaque étape à l'aide des fonctions `print_graph` et `print_graph_coloring`. Il est alors recommandé d'exécuter Firefox avant de lancer votre programme, sinon celui-ci attendra à chaque affichage la fin de Firefox avant de continuer.

3 Allocation des registres

8. Écrire une fonction `apply_coloring` de type `Coloring.coloring -> Riscv.riscv list -> Riscv.riscv list` qui applique un coloriage sur une liste d'instructions RISC-V. Pour les registres spillés, on pourra utiliser les registres `Register.temp0` et `Register.temp1` afin de les charger et sauvegarder sur la pile.
9. Écrire une fonction principale qui lit une liste d'instructions RISC-V sur l'entrée standard, calcule son graphe d'interférence, en déduit un coloriage et affiche sur la sortie standard les instructions auxquelles on a appliqué le coloriage.
   On pourra la tester avec le fichier `exemple.s` fourni dans l'archive.