Projet Prog

Martin CUINGNET

December 2023

1 Comment compiler

L'exécutable `ccomp` qui permet de vérifier le typage et de produire le code est LC3 (stocké dans `nom_fichier.asm`) est produit avec la commande `make`.
J'utilise l'outil `lc3tools` pour exécuter l'assembleur : la commande `make test v=nom_fichier` permet de compiler et d'exécuter le fichier `Test/ok/nom_fichier.c`

2 Fonctionnalités notables

- Gère entièrement les fonctions
- Possibilité d'avoir plus de 32 variables statiques et variables locales en même temps (normalement difficile, car l'offset est dans la plage $[-32, 31]$)
- Possibilité de sauter vers des endroits du code arbitrairement éloignés avec l'instruction temporaire `GLEA`, convertie en code LC3 réel lors d'une seconde passe
- Les opérations de division, multiplication et modulo sont implémentées dans une sous-routine pour être plus efficaces en termes de lignes
- Gère des corps d'instructions `IF` et `WHILE` de plus de 256 lignes en utilisant `GLEA` (qui utilisera après traduction l'instruction `JMP`)
- Utilise une représentation par liste pour la génération de code intermédiaire au lieu d'une simple chaîne de caractères pour éviter la concaténation de chaînes (complexité de $\hat{}$: taille des deux chaînes, complexité de @ : taille de la liste de gauche)
- Gère des fonctions liées aux chaînes telles que `puts`, `putc` et `getc`

3 Choix d'implémentation

- J'utilise une instruction personnalisée : `GLEA` et génère un code intermédiaire sans virgule (pour faciliter le parsing lors de la deuxième passe)

- Pour mettre une valeur immédiate dans un registre, je stocke la valeur immédiate dans un `.FILL` puis je charge ce `.FILL` dans le registre, en utilisant également des labels et `BR` pour sauter par-dessus le `.FILL`. Par exemple :

```
LD R0 CST
BR IGNORE_CST
CST .FILL #42
IGNORE_CST
```

- Je gère les lvalues en ayant un paramètre `get_lvalue` pour `handle_expr` qui indique à la fonction de renvoyer la valeur ou l'adresse d'une lvalue. Par exemple, lorsqu'on rencontre `++x`, on récupère l'adresse de x, ajoute 1 à x et renvoie sa valeur.
- Je stocke chaque chaîne à la fin des instructions et avant la mémoire statique.
- J'accède aux variables globales avec `R4` et un offset, et aux variables locales avec `R6` et un offset.
- Je saute quelque part dans les instructions en plaçant l'adresse de destination dans un registre (cette adresse est calculée lors d'une deuxième passe avec l'instruction `GLEA`) et je saute à ce registre (avec `JSRR` ou `JMP`).
- J'utilise une table des symboles qui est modifiée de manière récursive pour stocker le nom, l'offset et la globalité de chaque variable déclarée rencontrée dans le code.
- Lorsqu'une fonction est appelée, l'appelant empile ses arguments sur la pile (c'est-à-dire, en décrémentant `R6`), saute vers l'appelé, puis dépile les arguments. L'appelé empile `R5`, `R6` et `R7`, réinitialise `R5` à `R6`, et lorsqu'il retourne, il restaure `R5`, `R6` et `R7` avant de les dépiler et de `RET`.