Compilation

TP 5 : Traduction des fonctions

G. Iooss & C. Alias

Le but de ce TD/TP est de rajouter la gestion des fonctions au compilateur simple que l'on a vu au TP précédent.

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** $\mathbf{r_1}, \mathbf{r_2}, lab$: si $r_1 \leq r_2$, saute au label *lab*.
- **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** *lab* : saute au label *lab*.

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.

Partie I *Appel et retour de fonctions*

Considérons les fonctions suivantes :

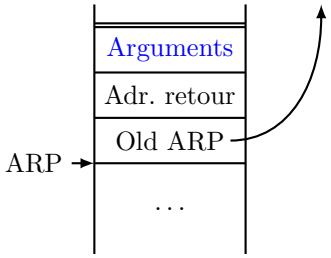
```
int main() {
  int n;
  int i;
  int sum;
  int k;

  n = 10;
  k = 2;
  for (i=0; i<n; i++) {
    sum = sum + mult(i,k);
  }
  return sum;
}

int mult(int a, int b) {
  return (a*b);
}
```

En utilisant le compilateur simple vu précédemment en TP, on suppose que l'on a généré séparément le code assembleur pour chacune de ces fonctions. Ces deux codes font les suppositions suivantes :

- Le code assembleur de chaque fonction commence par un label "`[nom_de_la_fonction] :`" et termine par "`[nom_de_la_fonction]_end :`".
- Au début de l'exécution du code assembleur d'une fonction, l'état de la pile est le suivant :



- Au début de l'exécution du code assembleur d'une fonction, tous les registres/variables temporaires sont libres.

Exo.

- [Gestion des temporaires] Quels sont les variables accessibles dans la fonction `main` ? Dans la fonction `mult` ? Dans la fonction `main` après un appel à `mult`. Déduisez-en quels sont les temporaires à [allouer/désallouer/mettre en réserve/récupérer] au moment de l'appel d'une fonction et de la sortie d'une fonction.
- [Prélude] Quels sont les actions à faire pour pouvoir appeler `mult` au milieu de `main` ?
- [Postlude] Quels sont les actions à faire lorsque l'on rencontre le `return` de `mult` pour pouvoir revenir au `main` ?
- Généralisez ces deux dernières questions au cas d'un appel de fonction quelconque.
- Dans le cas d'appels successifs, comment s'organise la pile ?
- Sachant que la pile a une taille de 256, combien d'appels imbriqués peut-on faire maximum avant de rencontrer un "stack overflow".

Les parties de la pile associé à un appel de fonction est appelé **enregistrement d'activation**.

Partie II *Implementation*

Récupérez les sources du TP `tp5_src.tar.gz` et décompressez-les. Les classes du compilateur sont strictement identiques au compilateur vu durant le dernier TP. Les modifications suivantes ont été faites :

- La grammaire de `parser.ypp` a été étendue au cas des fonctions (et pour gérer les appels de fonctions et les return).
- Des fonctions pour manipuler `ARP` ont été rajouté dans `CodeDigmips.h/.cc`. De plus, une variable `PC` (Program Counter) a été également rajoutée pour connaître le numéro de ligne du programme généré.

De plus, on suppose que la fonction principale (la première a être appelée) s'appelle `main` et n'admet aucun arguments.

Manip.

- Dans `parser.ypp`, trouvez et essayez de comprendre les passages suivants :
  - Appel d'une fonction dans une expression.
  - Appel d'une fonction de type "void" (en tant qu'instruction).
  - Instruction `return`.
  - Gestion des fonctions (notamment au niveau de l'allocation de leur arguments et variables temporaires.
- [Prélude et postlude] Dans `CodeDigmips.ypp`, complétez le prélude et le postlude dans les fonctions `call` et `ret`.
- Testez votre implémentation sur l'exemple vu précédemment, voire sur votre propre exemple.