TP Mips

Pour ces exercices de programmation en assembleur `Mips`, vous utiliserez le simulateur `Mars`, que vous téléchargez à l'adresse `https://github.com/dpetersanderson/MARS/releases/download/v.4.5.1/Mars4_5.jar` ou depuis la page cours.

Pour le lancer depuis la ligne de commande : `java -jar Mars4_5.jar`

Exercice 1 (Arithmétique) Écrire des programmes `Mips` qui calculent et affichent les résultats des expressions suivantes. On suppose que la valeur de la variable x est initialement stockée dans le registre `$a0`. Vous devez utiliser uniquement des registres (pas la mémoire) et en utiliser aussi peu que possible.

1. $x^2 + 3x + 5$
2. $x(x+1)(x+2)(x+3)$

□

Exercice 2 (Contrôle) Écrire des programmes `Mips` traduisant les programmes suivants. On supposera que la valeur de la variable `x` est stockée dans le registre `$a0`, et qu'une valeur renvoyée par `return` doit être placée dans le registre `$v0` (après quoi il faut aller à la fin du programme).

Programme 1 : encadrement.

```
if (x < 27) {
  return 1;
} else {
  if (x > 38) {
    return 2;
  } else {
    return 3;
  }
}
```

Programme 2 : puissance.

```
int y = 0;
int z = 1;
while (y < x) {
  z = 2*z;
  y = y+1;
}
return z;
```

Programme 3 : saisie.

```
while (true) {
  int y = read_int();
  if (0 <= y && y < x) {
    break;
  }
}
return(y+1);
```

□

Exercice 3 (Séquences de données) Dans le simulateur `Mars`, observer la mémoire après le chargement des données suivantes.

```
    .data
x:  .word 1 2 4 8 16 32 64
```

Comment cette séquence de valeurs est-elle organisée en mémoire ?

On suppose maintenant avoir deux données enregistrées dans la zone des données statiques : un nombre entier n et une séquence s de n nombres entiers. Écrire des programmes `Mips` pour :

1. calculer la somme des éléments de s,
2. déterminer si l'un des éléments de s vaut 0,
3. renverser la séquence s en place.

□

Exercice 4 (Interprétation de code assembleur) Pour chacun des trois programmes `Mips` ci-dessous, trouver la valeur de `x` faisant que le programme affiche `ok`. Vous pouvez utiliser le simulateur `Mars` pour suivre l'évolution des registres (mais il vous faut surtout reconstruire la structure de ces programmes !).

Mystère 1 : la bonne voie.

```
    .text
    la    $t0, x
    lw    $t0, 0($t0)
    subi  $t1, $t0, 10
    blt   $t1, 33, L1
    add   $t2, $t0, $t1
    blt   $t2, 78, L2
L1: la    $a0, ko
    b     L3
L2: la    $a0, ok
L3: li    $v0, 4
    syscall
    li    $v0, 10
    syscall

    .data
x:  .word   ???
ok: .asciiz "ok"
ko: .asciiz "ko"
```

Mystère 2 : la bonne distance.

```
    .text
    li    $t0, 946381
    li    $t1, 0
L1: beqz  $t0, L2
    div   $t0, $t0, 10
    addi  $t1, $t1, 1
    b     L1
L2: la    $t2, x
    lw    $t2, 0($t2)
    beq   $t1, $t2, L3
    la    $a0, ko
    b     L4
L3: la    $a0, ok
L4: li    $v0, 4
    syscall
    li    $v0, 10
    syscall

    .data
x:  .word   ???
ok: .asciiz "ok"
ko: .asciiz "ko"
```

Mystère 3 : le bon départ.

```
    .text
    li    $t0, 0
    la    $t1, x
    lw    $t1, 0($t1)
    b     L2
L1: add   $t0, $t0, $t1
    subi  $t1, $t1, 1
L2: bgtz  $t1, L1
    beq   $t0, 4753, L3
    la    $a0, ko
    b     L4
L3: la    $a0, ok
L4: li    $v0, 4
    syscall
    li    $v0, 10
    syscall

    .data
x:  .word   ???
ok: .asciiz "ok"
ko: .asciiz "ko"
```

□

Exercice 5 (Compilation améliorée) Cet exercice vise à améliorer la fonction de compilation des expressions IMP donnée dans les notes de cours (fonction `tr_expr` page 55, dont vous pouvez aussi récupérer le code sur la page du cours). Le code assembleur produit par la version actuelle place son résultat dans le registre `$t0`. Il utilise la pile pour stocker tous ses résultats intermédiaires, et n'utilise que deux registres au total (`$t0` et `$t1`). L'expression $(x-1)*(2+y)$ serait ainsi traduite en :

```
la   $t0, x
lw   $t0, 0($t0)
push $t0
li   $t0, 1
pop  $t1
sub  $t0, $t1, $t0
push $t0
li   $t0, 2
push $t0
la   $t0, y
lw   $t0, 0($t0)
pop  $t1
add  $t0, $t1, $t0
pop  $t1
mul  $t0, $t1, $t0
```

Note : les instructions `push` *et* `pop` *n'existent pas en* `Mips`*, il faudrait en réalité les décomposer comme dans le cours en un accès mémoire et une mise à jour du pointeur* `$sp`*.*

On se propose de ne plus utiliser la pile, mais d'utiliser plus de registres à la place. On se donne un tableau caml contenant les registres qu'on s'autorise à utiliser, par exemple :

```
let regs = [| t0; t1; t2; t3; t4; t5 |]
```

On peut alors faire référence à chacun à l'aide d'un indice entier 0, 1, 2, 3, 4 ou 5 dans ce tableau, et on peut écrire une fonction de traduction qui va utiliser ces registres dans l'ordre.

1. Écrire une nouvelle version de `tr_expr`, qui prend pour arguments un indice de registre `i` et l'expression `e` à traduire, et qui produit un code qui calcule et place dans le registre `regs.(i)` la valeur de `e`, sans modifier les valeurs stockées dans les registres d'indice inférieur à `i`. La fonction échouera s'il n'y a pas assez de registres dans le tableau `regs`. Adapter le reste du code pour que cette fonction soit correctement appelée. Pour l'expression $(x-1)*(2+y)$, le code produit pourrait ainsi être :

   ```
   la   $t0, x
   lw   $t0, 0($t0)
   li   $t1, 1
   sub  $t0, $t0, $t1
   li   $t1, 2
   la   $t2, y
   lw   $t2, 0($t2)
   add  $t1, $t1, $t2
   mul  $t0, $t0, t1
   ```

2. *(Défi, pas indispensable pour la suite)* Adapter le code pour que la nouvelle fonction de traduction n'échoue pas lorsqu'il n'y a plus de registres disponibles, mais commence à utiliser la pile à la place (plusieurs stratégies possibles pour cela).

Le langage assembleur `Mips` propose des variantes de certaines instructions arithmétiques, qui prennent une (petite) valeur constante comme deuxième opérande. On pourrait par exemple obtenir pour notre expression $(x-1)*(2+y)$:

```
la   $t0, x
lw   $t0, 0($t0)
addi $t0, $t0, -1
la   $t1, y
lw   $t1, 0($t2)
addi $t1, $t1, 2
mul  $t0, $t0, t1
```

Note : pour pouvoir utiliser `addi` *deux fois, on a également utilisé le fait que* $2+y$ *est égal à* $y+2$*.*

3. Améliorer la fonction `tr_expr` pour tirer parti de ces instructions.

Remarquez enfin que l'on peut améliorer la compilation en travaillant au niveau des expressions elles-mêmes, pour ne pas générer du code assembleur calculant une valeur que l'on saurait déjà prédire. Ainsi, une expression `Bop(Add, Cst 1, Cst 2)` pourrait être simplifiée en `Cst 3` avant même l'appel à `tr_expr`.

4. Écrire des fonctions `simp_expr: expr -> expr` et `simp_instr: instr -> instr` qui prennent en argument l'AST d'une expression ou d'une instruction IMP et renvoient un nouvel AST simplifié, dans lequel les calculs évidents ont été réalisés. Intégrer l'appel à ces fonctions dans le compilateur.

□