TP Mips

Pour ces exercices de programmation en assembleur `Mips`, vous utiliserez le simulateur `Mars`, que vous télécharger à l'adresse `http://courses.missouristate.edu/KenVollmar/mars/download.htm` 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. $x(x+1)(x+2)(x+3)$

Correction :

1. *Accumulation du résultat dans* `$t0`*, utilisation de* `$t1` *comme auxiliaire.*

```
        mul     $t0, $a0, $a0   # t0 <- x^2
        li      $t1, 3
        mul     $t1, $t1, $a0   # t1 <- 3x
        add     $t0, $t0, $t1   # t0 <- x^2 + 3x
        addi    $a0, $t0, 5     # a0 <- résultat
        li      $v0, 1
        syscall
```

2. *Accumulation du résultat dans* `$t0`*, utilisation de* `$t1` *comme auxiliaire.*

```
        addi    $t0, $a0, 1     # t0 <- x+1
        mul     $t0, $a0, $t0   # t0 <- x(x+1)
        addi    $t1, $a0, 2     # t1 <- x+2
        mul     $t0, $t0, $t1   # t0 <- x(x+1)(x+2)
        addi    $t1, $a0, 3     # t1 <- x+3
        mul     $a0, $t0, $t1   # a0 <- résultat
        li      $v0, 1
        syscall
```

En transformant un peu, on peut faire encore plus compact (accumulation du résultat dans `$a0`*, utilisation de* `$t0` *comme auxiliaire).*

```
        addi    $t0, $a0, 1     # t0 <- x+1
        mul     $a0, $a0, $t0   # a0 <- x(x+1)
        addi    $t0, $t0, 1     # t0 <- (x+1)+1
        mul     $a0, $a0, $t0   # a0 <- x(x+1)(x+2)
        addi    $t0, $t0, 1     # t0 <- (x+2)+1
        mul     $a0, $a0, $t0   # a0 <- résultat
        li      $v0, 1
        syscall
```

□

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);
```

Correction :

1. *Dans cette version, on branche quand le test est positif. L'instruction suivant un test dans le texte du programme correspond donc à la branche négative.*

```
encadrement:
    blt   $a0, 27, er1
    bgt   $a0, 38, er2
    li    $v0, 3
    b     efin
er1:
    li    $v0, 1
```

```
    b     efin
er2:
    li    $v0, 2
efin:
```

2. *Choix d'allocation : on place* `y` *dans* `$t0` *et* `z` *directement dans* `$v0`*. On réalise la multiplication par deux avec un décalage.*

```
puissance:
    li    $t0, 0
    li    $v0, 1
ploop:
    bge   $t0, $a0, pfin
    sll   $v0, $v0, 1
    addi  $t0, $t0, 1
    b     ploop
pfin:
```

3. *Tant que l'un des deux tests échoue on revient au début, et sinon on poursuit.*

```
saisie:
    li    $v0, 5
    syscall
    blt   $v0, 0, saisie
    bge   $v0, $a0, saisie
    addi  $v0, $v0, 1
```

Note : si on voulait en plus afficher une invite avant chaque demande de saisie, on aurait besoin de rendre `$a0` *disponible pour l'appel système d'affichage. Il faudrait donc au préalable déplacer le paramètre* `x` *dans un autre registre, par exemple* `$t0`*.*

Note : pour tester l'ensemble, on peut faire démarrer le fichier par le code suivant, en fournissant une valeur pour `$a0` *et le nom de la fonction à tester. Aucune de ces fonctions ne faisant elle-même un appel ou ayant besoin d'un tableau d'activation, on n'a rien de particulier à faire pour gérer l'appel lui-même.*

```
main:
    li    $a0, 32
    jal   encadrement
    move  $a0, $v0
    li    $v0, 1
    syscall
    li    $v0, 10
    syscall
```

□

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.

Correction :

1. *Pour itérer efficacement sur la liste, on commence par initialiser notre compteur* `$t0` *avec l'adresse de la liste, puis on incrémente par pas de 4 jusqu'à atteindre* `&s + 4n`*.*

```
    .data
n:  7
s:  1 2 4 8 16 32 64

    .text
somme:
    la    $t0, s              # t0 <- &s (ie. i = 0)
    la    $t1, n
    lw    $t1, 0($t1)
    sll   $t1, $t1, 2         # t1 <- 4n
    add   $t1, $t0, $t1       # t1 <- &s + 4n
    li    $v0, 0              # res = 0
sloop:
    bge   $t0, $t1, sfin      # arrêt si t0 >= &s + 4n
    lw    $t2, 0($t0)
```

```
    add   $v0, $v0, $t2       # res <- res + s[i]
    addi  $t0, $t0, 4         # t0 <- t0+4  (ie. i <- i+1)
    b     sloop
sfin: nop
# résultat dans $v0
```

2. *Même astuce pour l'itération. On interrompt la boucle lorsque l'on rencontre un zéro.*

```
zero:
    la      $t0, s          # t0 <- &s (ie. i = 0)
    la      $t1, n
    lw      $t1, 0($t1)
    sll     $t1, $t1, 2     # t1 <- 4n
    add     $t1, $t0, $t1   # t1 <- &s + 4n
    li      $v0, 0          # res = 0 par défaut
zloop:      bge     $t0, $t1, zfin  # arrêt si t0 >= &s + 4n
    lw      $t2, 0($t0)
    beqz    $t2, zfinpos    # break si zéro
    addi    $t0, $t0, 4     # t0 <- t0+4  (ie. i <- i+1)
    b       zloop
zfinpos:
    li      $v0, 1          # res = 1 si zéro trouvé
zfin:       nop
```

3. *Pas le plus efficace, mais juste pour le plaisir d'utiliser la pile : une première boucle sur s lit les éléments et les empile, puis une deuxième boucle dépile et écrit.*

```
renverse:
    la      $t0, s          # t0 <- &s (ie. i = 0)
    la      $t1, n
    lw      $t1, 0($t1)
    sll     $t1, $t1, 2     # t1 <- 4n
    add     $t1, $t0, $t1   # t1 <- &s + 4n
rloop1:     bge     $t0, $t1, rpart2
    lw      $t2, 0($t0)
    sw      $t2, 0($sp)
    subi    $sp, $sp, 4
    addi    $t0, $t0, 4
    b       rloop1
rpart2:     la      $t0, s          # t0 <- &s (ie. i = 0)
rloop2:     bge     $t0, $t1, rfin
    addi    $sp, $sp, 4
    lw      $t2, 0($sp)
    sw      $t2, 0($t0)
    addi    $t0, $t0, 4
    b       rloop2
rfin:       nop
```

□

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"
```

Correction :

1. *Premier test du programme : branche* `ko` *si* `x-10 < 33`*. Dans le cas contraire, branche* `ok` *si* `2x-10 < 78`*. Et sinon encore, branche* `ko`*. Seule valeur possible pour la branche* `ok` *: 43.*
2. *Le programme compte dans* `$t1` *le nombre de chiffres dans l'écriture décimale de* `$t0` *(approximation : on compte ici zéro chiffre pour 0). Il faut donc donner comme valeur à* `x` *le nombre de chiffres de* `946381` *: 6.*
3. *Le programme calcule dans* `$t0` *la somme des nombres de 0 à* `x`*. Pour obtenir* `4753` *il faut prendre* `x=97`*.*

□

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.

Correction : Version avec échec.

```
let rec tr_expr i e =
  if i >= nb_regs then failwith "not enough registers";
  let ti = regs.(i) in
  match e with
  | Cst(n)  -> li ti n
  | Var(id) -> la ti id @@ lw ti 0(ti)
  | Bop(bop, e1, e2) ->
    let op = match bop with
      | Add -> add
      | Mul -> mul
      | Lt  -> slt
      | And -> and_
    in
    tr_expr i e1
    @@ tr_expr (i+1) e2
    @@ op ti ti regs.(i+1)

let tr_expr e = tr_expr 0 e
```

Version sans échec minimale : utilise le tableau de registres de manière circulaire, et sauvegarde sur la pile un registre qui s'apprêterait à être écrasé.

```
let rec tr_expr i e =
  let ti = regs.(i mod nb_regs) in
  match e with
  | Cst(n)  -> li ti n
  | Var(id) -> la ti id @@ lw ti 0(ti)
  | Bop(bop, e1, e2) ->
    let op = match bop with
      | Add -> add
      | Mul -> mul
      | Lt  -> slt
      | And -> and_
    in
    if i+1 < nb_regs then
      tr_expr i e1
      @@ tr_expr (i+1) e2
      @@ op ti ti regs.(i+1)
    else
      let tj = regs.((i+1) mod nb_regs) in
      tr_expr i e1
      @@ push tj
      @@ tr_expr (i+1) e2
```

```
      @@ op ti ti tj
      @@ pop tj
```

Fonctions de simplification.

```
let rec simp_expr e = match e with
  | Cst _ | Var _ -> e
  | Bop(Add, e1, e2) ->
    begin match simp_expr e1, simp_expr e2 with
      | Cst n1, Cst n2 -> Cst (n1+n2)
      | Cst 0, e' | e', Cst 0 -> e'
      | e1', e2' -> Bop(Add, e1', e2')
    end
  | Bop(Mul, e1, e2) ->
    begin match simp_expr e1, simp_expr e2 with
      | Cst n1, Cst n2 -> Cst (n1*n2)
      | Cst 0, e' | e', Cst 0 -> Cst 0
      | Cst 1, e' | e', Cst 1 -> e'
      | e1', e2' -> Bop(Mul, e1', e2')
    end
  | Bop(op, e1, e2) -> Bop(op, simp_expr e1, simp_expr e2)

let rec simp_instr = function
  | Print(e) -> Print(simp_expr e)
  | Set(x, e) -> Set(x, simp_expr e)
  | While(e, s) ->
     While(simp_expr e, List.map simp_instr s)
  | If(e, s1, s2) ->
     If(simp_expr e, List.map simp_instr s1, List.map simp_instr s2)
```

□