Architecture des Ordinateurs, corrigé TP 3

Recommandations

— Etudier préalablement, et utiliser le MémoMIPS proposé en ligne.
— Commenter soigneusement *chaque* ligne écrite en assembleur...

Exercice 1. Compiler en MIPS le programme C++ suivant :

```
#include <iostream>

using namespace std;

int power(int x, int n)
{
   int i; int result = 1;

   for (i=1;i<=n;++i) result=result*x;
   return result;
} // power()

int main(void)
{
  int x; int n;

  cin >> x;
  cin >> n;
  cout << power(x,n);
  return 0;
} // main()
```

Correction. *Les variables* `x` *et* `n` *sont des variables globales, donc statiques. Autrement dit, il faut allouer dans le segment de données les deux mots mémoires correspondants (32 bits, puisqu'elles sont* `int`*). Inversement les variables* `i` *et* `result` *de la fonction* `power` *sont locales : leur durée de vie dans la mémoire ne doit pas excéder la durée du temps d'exécution de la fonction, et elles devraient donc être rangées dans la pile. Mais on peut aussi remarquer que ces variables (un incrément de boucle et un résultat) ont une portée très réduite (une boucle d'une instruction) et peuvent être immédiatement et avantageusement associées à deux registres temporaires de façon à minimiser le traffic avec la mémoire. On obtient alors :*

```
          .data
x:        .word 0x00000000         # allocation memoire pour x
n:        .word 0x00000000         # allocation memoire pour n

          .text
main:     la $t0, x                # $t0 contient l'adresse de x
          ori $v0, $zero, 5        # $v0 <- 5
          syscall                  # lecture au clavier
          sw $v0, 0($t0)           # la valeur lue est rangee dans x
          or $a0, $zero, $v0       # $a0 <- $v0

          la $t0, n                # $t0 contient l'adresse de n
          ori $v0, $zero, 5        # $v0 <- 5
          syscall                  # lecture au clavier
          sw $v0, 0($t0)           # la valeur lue est rangee dans n
          or $a1, $zero, $v0       # $a1 <- $v0

          jal power                # appel de power

          or $a0, $zero, $v0       # $a0 <- $v0
          ori $v0, $zero, 1        # $v0 <- 1
          syscall                  # affichage de l'entier $a0

          ori $v0, $zero, 10       # $v0 <- 10 pour exit
          syscall                  # appel systeme de terminaison

power:    ori $t1, $zero, 1        # $t1:result=1
          ori $t0, $zero, 1        # $t0:i=1
for:      slt $t2, $a1, $t0        # $t2=(i>n?1:0)
          bne $t2, $zero, exitfor  # if (i=n) goto exitfor;
          mul $t1, $t1, $a0        # result=result*x;
          addi $t0, $t0, 1         # ++i;
          j for                    # goto for
exitfor:  or $v0, $zero, $t1       # $v0 <- $t1:result
          jr $ra                   # return result
```

Exercice 2. Dans la fonction `power(x,n)` précédente, remplacer en MIPS la boucle :

```
for (i=1;i<=n;++i) result=result*x;
```

par :

```
for (i=0;i<n;++i) result=result*x;
```

Quel est le gain obtenu ?

Correction. *Faire partir l'incrément `i` à `0` au lieu de `1` permet à l'autre extrémité d'inverser le test sur `n` et rend ainsi inutile l'instruction `slt` :*

```
power:    ori $t1, $zero, 1        # $t1:result=1
          or $t0, $zero, $zero     # $t0:i=0
for:      beq $t0, $a1, exitfor    # if (i=n) goto exitfor;
          mul $t1, $t1, $a0        # result=result*x;
          addi $t0, $t0, 1         # ++i;
          j for                    # goto for
exitfor:  or $v0, $zero, $t1       # $v0 <- $t1:result
          jr $ra                   # return result
```

Exercice 3. Remplacer en MIPS la fonction `power(x,n)` précédente par sa version récursive :

```
int power(int x, int n)
{
    if (n==0) return 1;
       else   return x * power(x,n-1);
}
```

Traduire le programme résultant aussi fidèlement et aussi efficacement que possible en assembleur MIPS (en commentant chaque instruction). Quelles optimisations peuvent être réalisées dans la gestion de la pile ?

Correction. *On obtient pour la fonction `power(x,n)` récursive :*

```
power:  addi $sp, $sp, -12      # alloue l'espace dans la pile
        sw $ra, 8($sp)          # empile l'adresse de retour
        sw $a1, 4($sp)          # empile l'argument n
        sw $a0, 0($sp)          # emile l'argument x
        bne $a1, $zero, recurs  # if (n!=0) goto recurs
        ori $v0, $zero, 1       # return x; avec x^0 = 1
        addi $sp, $sp, 12       # pop des trois valeurs sauvees
        jr $ra                  # retour a l'appelant
recurs: addi $a1, $a1, -1       # n>=1: l'argument recoit n-1
        jal power               # appel recursif
        lw $a0, 0($sp)          # depile l'argument x
        lw $a1, 4($sp)          # depile l'argument n
        lw $ra, 8($sp)          # depile l'adresse de retour
        addi $sp, $sp, 12       # rend l'espace dans la pile
        mul $v0, $a0, $v0       # return x * power(x, n-1)
        jr $ra                  # retour a la fonction appelante
```

Dans cette version, les arguments `x` *et* `n` *ne sont pas dépilés dans la partie terminale non récursive de la fonction car ils ne sont pas modifiés entre le moment où ils sont empilés et le moment où il pourraient être dépilés. Il suffit donc de se contenter de rendre l'espace mémoire alloué. Mais plusieurs optimisations peuvent encore tre réalisées dans ce programme. En premier lieu, il est aisé de constater qu'en réalité* `x` *n'est jamais modifié, et que la valeur de* `n` *ne sert jamais au retour de l'appel récursif : il n'est donc tout simplement pas nécessaire d'empiler les arguments correspondants.*

```
power:    addi $sp, $sp, -4        # alloue 1 mot dans la pile
          sw $ra, 0($sp)           # empile l'adresse de retour
          bne $a1, $zero, recurs   # if (n!=0) goto recurs
          ori $v0, $zero, 1        # return x; avec x^0 = 1
          lw $ra, 0($sp)           # depile l'adresse de retour
          addi $sp, $sp 4          # rend 1 mot dans la pile
          jr $ra                   # retour a l'appelant
recurs:   addi $a1, $a1, -1        # n>=1: l'argument recoit n-1
          jal power                # appel recursif
          lw $ra, 0($sp)           # depile l'adresse de retour
          addi $sp, $sp 4          # rend 1 mot dans la pile
          mul $v0, $a0, $v0        # return x * power(x,n-1)
          jr $ra                   # retour a la fonction appelante
```

Deuxième optimisation : Dans la partie de la fonction qui ne génère pas d'appel récursif (c'est-à-dire quand `$a1` *contient* 0 *dans le branchement conditionnel* `bne`*), on constate que* `$ra` *n'est pas modifié (puiqu'on se trouve alors dans le cas d'une occurence terminale de la fonction). Il est donc inutile de le dépiler, et il suffit de rendre l'espace alloué dans la pile.*

```
power:    addi $sp, $sp, -4        # alloue 1 mot dans la pile
          sw $ra, 0($sp)           # empile l'adresse de retour
          bne $a1, $zero, recurs   # if (n!=0) goto recurs
          ori $v0, $zero, 1        # return x; avec x^0 = 1
          addi $sp, $sp 4          # rend 1 mot dans la pile
          jr $ra                   # retour a l'appelant
recurs:   addi $a1, $a1, -1        # n>=1: l'argument recoit n-1
          jal power                # appel recursif
          lw $ra, 0($sp)           # depile l'adresse de retour
          addi $sp, $sp 4          # rend 1 mot dans la pile
          mul $v0, $a0, $v0        # return x * power(x,n-1)
          jr $ra                   # retour a la fonction appelante
```

Troisième optimisation : Précisément dans le cas d'une occurence terminale de la fonction, il devient inutile d'empiler l'adresse de retour, pourvu que l'empilement soit réalisé par ailleurs dans le cas des occurences non terminales ; il suffit donc de réaliser cette empilement dans le cas d'un nouvel appel récursif.

```
power:    bne $a1, $zero, recurs   # if (n!=0) goto recurs
          ori $v0, $zero, 1        # return x; avec x^0 = 1
          jr $ra                   # retour a l'appelant
recurs:   addi $sp, $sp, -4        # alloue 1 mot dans la pile
          sw $ra, 0($sp)           # empile l'adresse de retour
          addi $a1, $a1, -1        # n>=1: l'argument recoit n-1
          jal power                # appel recursif
          lw $ra, 0($sp)           # depile l'adresse de retour
          addi $sp, $sp 4          # rend 1 mot dans la pile
          mul $v0, $a0, $v0        # return x * power(x,n-1)
          jr $ra                   # retour a la fonction appelante
```