TD8 : Fonctions récursives et appels imbriqués

Objectif(s)

✯ Ce TD a pour but de consolider les appels de fonctions et d'implémenter des fonctions récursives et imbriquées.

Note pédagogique:
Traiter une fonction récursive ainsi que la fonction `arimean` qui illustre bien aussi la nécessité de sauvegarder la valeur de l'argument avant d'effectuer les appels aux fonctions dans son corps.

Une fonction imbriquée est une fonction qui est appelée par une autre fonction. Les conventions d'appel sont faites pour pouvoir appeler des fonctions écrite soi-même ou par d'autres, sans avoir en avoir se soucier d'autre chose que de les avoir bien suivies.

Une fonction récursive est une fonction qui s'appelle elle-même. C'est donc une fonction qui appelle une fonction, cette fonction a juste le même nom : cela ne change rien au niveau des conventions d'appel ! C'est au programmeur de s'assurer que la fonction récursive termine, c'est un problème d'algorithmique pas de programmation assembleur.

Pour l'écriture de fonctions, vous suivrez consciencieusement les étapes de programmation d'une fonction en assembleur à savoir pour rappel (cf. l'énoncé de la semaine dernière aussi). Vous porterez une attention particulière au contenu des registres non persistants (par exemple `$4`) avant les appels de fonctions dans des fonctions et au besoin de sauvegarder la valeur des arguments dans certains cas (déterminez lesquels).

Exercice 1 – Fonction puissance linéaire

Question 1

On considère le code suivant :

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

void main() {
   int x = 3;
   int p = 2;

   printf("%d", puissance(x, p));
   printf("%d", puissance(2, 6));

   exit();
}
```

Donnez le code de la fonction puissance puis le code du programme principal correspondant au code C ci-dessus. N'oubliez pas qu'il faut allouer de l'espace mémoire sur la pile pour les variables locales, dans les fonctions mais aussi dans le programme principal (le `main`).

Solution:

```
.data

.text


main:
    # x et p non optimises en registre
    addiu $29, $29, -16 # na = 2 + nv = 2 donc 4 mots
    ori   $8, $0, 3     # initialisation de x
    sw    $8, 8($29)
    ori   $8, $0, 2     # initialisation de p
    sw    $8, 12($29)

    # puissance(x,p)
    lw    $4, 8($29)  # 1er arg = x
    lw    $5, 12($29) # 2eme arg = p
    jal   puissance
    ori   $4, $2, 0   # instruction d'addresse @a
    ori   $2, $0, 1   # affichage du resultat
    syscall

    # puissance(2,6)
    ori   $4, $0, 2   # 1er arg = 2
    ori   $5, $0, 6   # 2eme arg = 6
    jal puissance
    ori   $4, $2, 0   # instruction d'adresse @b
    ori   $2, $0, 1
    syscall

    addiu $29, $29, 16
    ori   $2, $0, 10
    syscall

puissance:
    # Prologue
    # On sauvegarde les 2 parametres meme si seul x doit obligatoirement etre sauvegarde
    addiu $29, $29, -16   # nv = 1 + nr = 0 + 1 ($31) + na = 2 = 4 mots
    sw    $31, 12($29)
    sw    $4,  16($29)    # sauvegarde x
    sw    $5,  20($29)    # sauvegarde p

    bne   $5, $0, _else   # si n!= cas recursif
    ori   $2, $0, 1       # sinon val retour = 1
    j     _fin_if
_else:
    # $4 contient deja la bonne valeur
    addiu $5, $5, -1      # p - 1
    jal   puissance
    lw    $8, 16($29)     # relecture x ; instruction d'adresse @c
    mult  $8, $2          # res * x
    mflo  $2
_fin_if:
    lw    $31, 12($29)
```

```
54      addiu $29, $29, +16
55      jr    $31
```

Question 2 – Représentation graphique de l'arbre d'appels

Donnez les appels récursifs engendrés par l'appel `puissance(2,6)`

Solution:

```
puissance(2,6)
     -> puissance(2,5)
          -> puissance(2,4)
               -> puissance(2,3)
                    -> puissance(2,2)
                         -> puissance(2,1)
                              -> puissance(2,0)
```

Question 3 – Représentation graphique de la pile

Numérotez les instructions de votre code assembleur.
Représentez graphiquement l'évolution de la pile au cours de l'exécution du premier appel à la fonction `puissance`. Vous indiquerez par @i les adresses de retour que vous mettrez sur la pile, avec i correspondant au numéro de l'instruction associée à cette adresse de retour.

Solution:

L'évolution de la pile lors de l'appel à `puissance(3, 2)` est représentée dans la Figure 1.

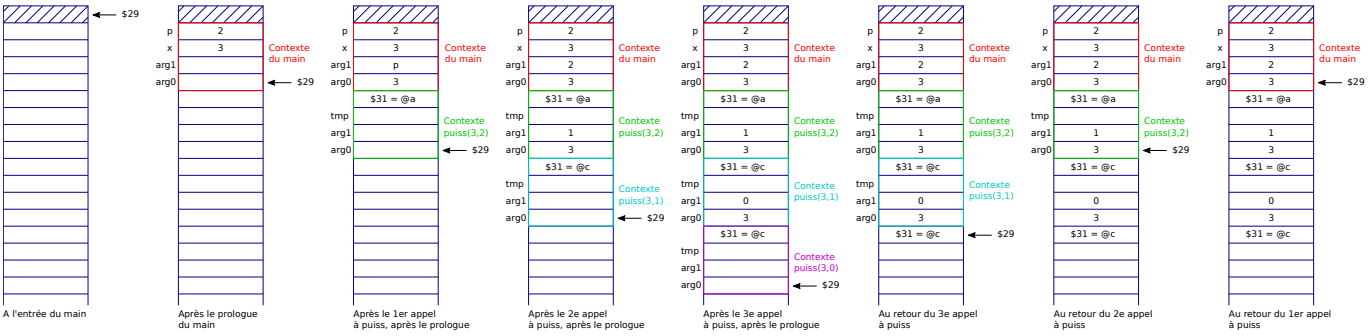

FIGURE 1 – Evolution de la pile lors d'un appel à puiss(3, 2)

Exercice 2 – Fonctions imbriquées

On souhaite écrire en assembleur MIPS32 le programme C qui calcule la valeur moyenne de tous les entiers (non négatifs) stockés dans un tableau de dimension quelconque. Par convention, et pour pouvoir réutiliser la fonction `sumtab()`, on suppose que le dernier élément du tableau possède une valeur négative. Ce dernier élément n'est pas pris en compte dans le calcul de la moyenne.

Le programme principal appelle la fonction `arimean()` qui a pour seul argument un pointeur sur un tableau d'entiers. Elle renvoie la valeur de la moyenne arithmétique des éléments du tableau (tronquée à la valeur entière inférieure). La fonction `arimean()` appelle la fonction `sizetab()` qui prend pour seul argument le pointeur sur le tableau, et renvoie le nombre d'éléments non négatifs contenus dans le tableau. Elle appelle ensuite la fonction `sumtab()`, qui calcule la somme de tous les entiers non négatifs contenus dans le tableau. Elle effectue la division entière et renvoie le quotient au programme appelant.

```c
#include <stdio.h>

/* variables globales initialisées */
int tab[] = {23, 7, 12, 513, -1} ;

/* programme principal */
void main(void) {
    int x = arimean(tab);
    printf("%d", x);
    exit();
}

/* cette fonction renvoie la moyenne arithmétique des éléments d'un tableau */
int arimean(int t[]) {
    int n = sizetab(t);
    int x = sumtab(t);
    return (x / n);
}

/* cette fonction renvoie le nombre d'éléments d'un tableau */
int sizetab(int t[]) {
    int index = 0;
    while (t[index] >= 0) {
        index += 1;
    }
    return index;
}

/* cette fonction renvoie la somme des éléments d'un tableau */
int sumtab(int t[]) {
    int accu = 0;
    int index = 0;
    while (t[index] >= 0) {
        accu = accu + t[index];
        index += 1;
    }
    return accu;
}
```

Question 1

Donnez le code de la fonction `arimean`.

Solution:

```
1  # fonction arimean
2  # nr = 0 + $31 ; nv = 2 ; na = 1
3  arimean:
4      # prologue
5      addiu   $29,    $29,    -16 # decrementation pointeur de pile
6      sw      $31,    12($29)     # sauvegarde de l'adresse de retour
7      sw      $4,     16($29)     # sauvegarde de l'argument de arimean
8      # corps de la fonction
9      jal     sizetab             # branchement a la fonction sizetab
10     sw      $2,     4($29)      # sauvegarde du resultat dans la variable n dans la pile
```

```
    lw      $4,     16($29)     # recuperation de l'argument (sauve dans le prologue)
    jal     sumtab              # branchement a la fonction sumtab
    sw      $2,     8($29)      # ecriture x
    lw      $8,     4($29)      # recuperation de n
    div     $2,     $8          # division entiere
    mflo    $2                  # stockage resultat dans $2
    # epilogue
    lw      $31,    12($29)     # restauration adresse de retour
    addiu   $29,    $29,    16  # incrementation pointeur de pile
    jr      $31                 # retour au programme appelant
```

Question 2

Donnez le code des fonction `sumtab` et `sizetab`.

Solution:

```
# fonction sizetab
# nr = 1 ($31) ; nv = 1 ; na = 0
# index : $2 parce que c'est une fonction terminale donc $2 ne sera pas modifie par des appels
# on utilisera $4 directement pour le parcours des cases du tableau
sizetab:
    # prologue
    addiu   $29,    $29,    -8 # decrementation pointeur de pile
    sw      $31,    4($29)     # sauvegarde de l'adresse de retour
    # corps de la fonction
    ori     $2,     $0,      0 # initialisation index

loop0:
    sll     $8,     $2,      2 # index * 4
    addu    $8,     $4,     $8 # &t[index]
    lw      $8,     0($8)      # chargement tab[i] dans $9
    bltz    $8,     fin_loop0  # test de sortie de boucle
    addiu   $2,     $2,    1   # incrementation index
    j loop0                    # retour test sortie de boucle
fin_loop0:
    # epilogue
    lw      $31,    4($29)     # restauration de l'adresse de retour
    addiu   $29,    $29,    8  # incrementation pointeur de pile
    jr      $31                # retour au programme appelant

# fonction sumtab
# on choisit $2 pour accumulateur
# on choisit $8 pour index
# nr = 0 + $31 + nv = 2 + na = 0
# prologue
sumtab:
    addiu   $29,   $29,  -12
    sw      $31,   8($29)
    ori     $2,    $0,     0   # initialisation registre accumulateur
    addiu   $8,    $0,     0   # initialisation index
loop1:
    sll     $9,    $8,     2   # index * 4
    addu    $9,    $9,    $4   # &t[index]
    lw      $9,    0($9)       # chargement tab[i] dans $9
    bltz    $9,    fin_loop1   # test de sortie de boucle
    addu    $2,    $2,    $9   # accumulation
    addiu   $8,    $8,    1    # incrementation index
    j       loop1
```

```
43 fin_loop1:                       # retour test sortie boucle
44     lw      $31,    8($29)
45     addiu   $29,    $29,    12
46     jr      $31                 # retour au programme appelant
```

Question 3

Donnez le code du programme principal ainsi que les déclarations des variables globales du programme.
Solution:

```
1  # variables globales initialisees
2  .data
3      tab: .word 23, 7, 12, 513, -1
4
5
6  .text
7  # programme principal
8  # nv = 1 na = 1
9  main:
10     # prologue
11     # na = 1, nr = 0, nv = 1
12     addiu   $29,    $29,    -8  # decrementation pointeur de pile $29
13     # corps de la fonction
14     lui     $4,     0x1001      # chargement adresse de tab dans $4
15     jal     arimean             # branchement a la fonction mean
16     sw      $2,     4($29)      # ecriture x
17     lw      $4,     4($29)      # argument de l'appel systeme affichage
18     ori     $2,     $0,     1   # code de l'appel systeme affichage d'entierr
19     syscall
20
21     # epilogue
22     addiu   $29,    $29,    8   # decrementation pointeur de pile $29
23     ori     $2, $0, 10          # code de l'appel systeme 'exit'
24     syscall
```

Exercice 3 – Élément de la suite de Fibonnaci

Question 1

Écrivez le code assembleur correspondant au code C ci-dessous. Ce code contient une fonction calculant le nombre de Fibonnaci à l'ordre n et un programme principal affichant le résultat pour une valeur stockée dans une variable globale.

```
int n = 4;

void main() {
    printf("%d", fib(n));
    exit();
}

int fib(int n) {
    if (n == 0 || n == 1) {
        return 1;
```

```
    }
    else {
        return fib(n - 1) + fib(n - 2);
    }
}
```

Solution:

```
.data
n: .word 4

.text
main:
    addiu $29, $29, -4    # na = 1, nv = 0
    lui   $8, 0x1001
    lw    $4, 0($8)       # 1er arg = n
    jal   fib
    ori   $4, $2, 0       # instruction d'adresse @a
    ori   $2, $0, 1       # affichage du resultat
    syscall

    addiu $29, $29, 4
    ori   $2, $0, 10
    syscall


fib:
    addiu $29, $29, -12  # 3 mots : na = 1 + nv = 0 + nr = 1 ($16) + $31
    sw    $31, 8($29)
    sw    $16, 4($29)
    sw    $4, 12($29)    # on conserve n/1er param dans son emplacement

    beq   $4, $0, _then  # cas n = 0
    ori   $8, $0, 1
    beq   $4, $8, _then  # cas n = 1
    j     _else
_then:
    ori   $2, $0, 1
    j     _fin_if
_else:
    addiu $4, $4, -1     # n - 1
    jal   fib
    ori   $16, $2, 0     # fib(n-1) dans reg persistant, inst. @b
    lw    $4, 12($29)    # relecture n
    addiu $4, $4, -2     # n - 2
    jal   fib
    addu $2, $16, $2     #  fib(n-1) + fib(n-2), inst. @c
_fin_if:
    lw    $31  8($29)
    lw    $16, 4($29)
    addiu $29, $29, 12
    jr    $31
```

Question 2 – Représentation graphique de l'arbre d'appels

Représentez graphiquement l'arbre des appels engendrés par l'appel `fib(n)` avec `n` qui vaut 4.

Solution:

```
fib(4) -> fib(3) -> fib(2) -> fib(1)
                           -> fib(0)
                  -> fib(1)
        -> fib(2) -> fib(1)
                  -> fib(0)
```

Question 3 – Représentation graphique de la pile

Numérotez les instructions de votre code assembleur.

Représentez graphiquement l'évolution de la pile au cours de l'exécution. Vous indiquerez par @i les adresses de retour que vous mettrez sur la pile, avec i correspondant au numéro de l'instruction associée à cette adresse de retour.

Solution:

L'évolution de la pile est représentée dans la Figure 2. On se limite ici à `fib(3)`.

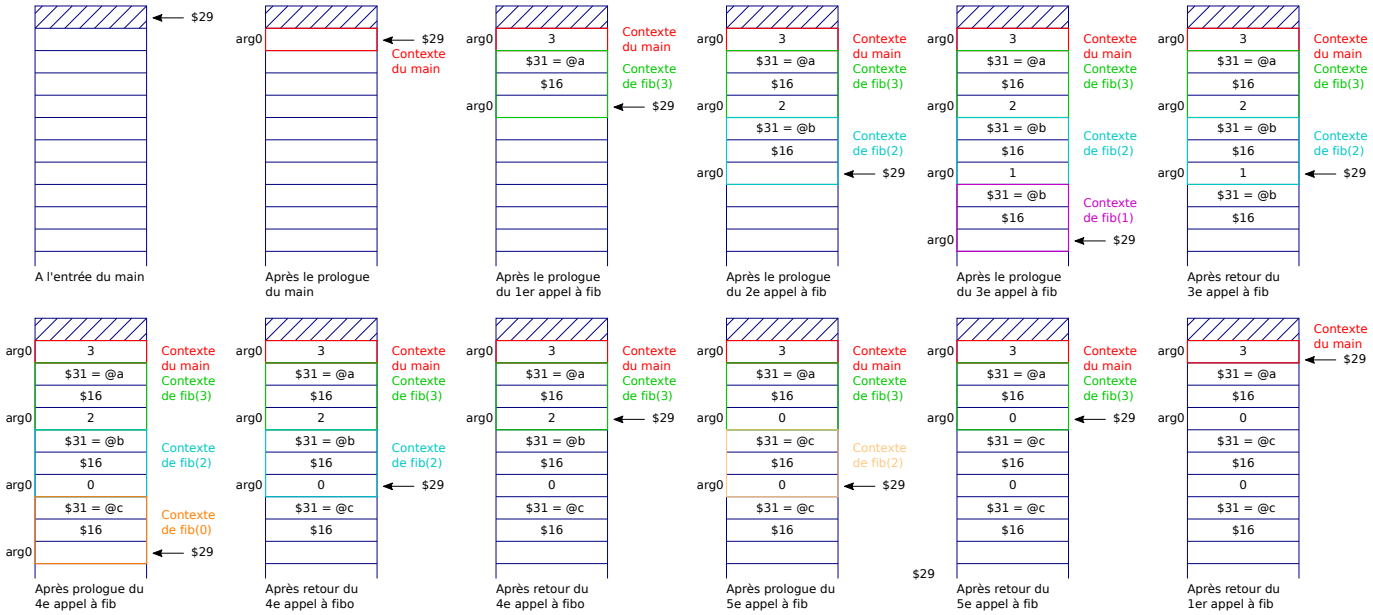

FIGURE 2 – Evolution de la pile lors d'un appel à fib(3)