TD7 : Appels de fonctions

Objectif(s)

- ✯ Maitrise des conventions d'appel en MIPS
- ✯ Écriture de fonctions et d'appels de fonction en assembleur

Exercice(s)

Exercice 1 – Questions préliminaires

Question 1

1. Rappelez les informations à faire passer entre une fonction appelante et une fonction appelée lors d'un appel de fonction et le sens de ces échanges.
2. Quelles sont les informations à sauvegarder lors d'appel de fonction et pourquoi ?
3. Que définissent les conventions d'appel ? A quoi servent-elles ?
4. Rappelez les conventions d'appel en MIPS ?

Solution:

1. Il faut faire passer
   (a) la valeur des paramètres de la fonction appelante à la fonction appelée,
   (b) l'adresse de retour de la fonction appelante à la fonction appelée car une fonction peut être appelée de plusieurs endroits et l'adresse de retour dépend du site d'appel, c'est l'adresse de l'instruction qui suit l'appel (cf. cours).
   (c) la valeur de retour de la fonction appelée à la fonction appelante,
2. Il faut sauvegarder le contenu des registres persistants utilisés dans l'appelant (ou ceux non persistants utilisés dans l'appelé) pour ne pas écraser des valeurs qui seraient utiles ultérieurement (et non recalculables coté appelant). Il faut aussi sauvegarder l'adresse de retour pour éviter d'écraser sa valeur en cas d'appel imbriqué.
3. Les conventions d'appel définissent les régles d'échanges des informations entre appelant et appelé à savoir : comment et dans quel ordre sont passés les paramètres entre une fonction appelante et une fonction appelée, comment est passée la valeur du résultat s'il y en a un. Elles définissent aussi les règles de sauvegarde des registres. Peu importe les conventions, elles permettent d'appeler en suivant ces règles toute fonction, implantée par un autre programmeur ou prédéfinie, qui suit aussi ces règles, dans le programme principal ou dans un sous-programme ou une fonction. Elles permettent donc aussi le bon fonctionnement du code en cas d'appel imbriqués ou récursifs.
4. Conventions d'appel en MIPS : dans le contexte d'exécution d'une fonction, allouée par le prologue de la fonction elle-même, sont réservés autant d'emplacements que le nombre maximal de mots nécessaires pour stocker les paramètres des fonctions appelées par la fonction. Pour chaque appel, on calcule le nombre de mots nécessaire au stockage de ses arguments sur la pile : on compte 1 mot pour chacun des 4 premiers arguments, pour les suivants on calcule comme pour les variables locales le nb d'octets nécessaires en prenant en compte les contraintes d'alignement dictées par l'argument et on arrondit au multiple de 4 octets supérieur (on alloue

toujours un nombre de mots sur la pile). On alloue `na` mots pour les arguments correspondant au nombre max parmi ceux calculés. A noter que dans un contexte de fonction, l'emplacement au sommet de la pile correspond à l'emplacement dédié au premier paramètre d'une fonction appelée.
Lors d'un appel, la valeur des 4 premiers paramètres est passé par registre (`$4-$7`), les suivants sont passés par la pile.
L'adresse de retour est passée via le registre `$31` en effectuant un appel avec l'instruction `jal`, et la valeur de retour est passée via le registre `$2`.

Concernant la sauvegarde des registres : la fonction appelée est chargée de sauvegarder tous les registres persistants qu'elle utilise ainsi que `$31`. Les registres persistants que l'on utilise dans cette partie du cours sont les registres `$16-$23`. La sauvegarde respecte l'ordre suivant : les registres sont sauvegardés par ordre de numéro croissant des plus petites adresses aux plus grandes sur la pile. Autrement dit, le registre persistant utilisé avec le plus petit numéro se trouve le plus près du somment de pile (et `$31` est le plus proche du fond de pile / des emplacement des paramètres).

La figure 1 contient la procédure pour l'implantation de fonction, vous suivrez cette procédure pour TOUTES les fonctions que vous aurez à écrire. Complétez les "trous".

Solution:
Il faut remplir les trous dans l'encadré avec le registre `$2`, le registre `$31` et l'instruction `jr $31`.

Exercice 2 – Moyenne de 3 nombres

On souhaite écrire le code du programme C suivant :

```
/* trois variables globales */
int n = 15;
int m = -1;
int l = 124;

/* fonction moyenne3 */
int moyenne3(int p, int q, int r) {
   int sum = p + q + r;
   return sum / 3;
}

/* programme principal */
void main() {
  int tmp;
  tmp = moyenne3(n, m, 5);
  printf("%d", tmp);
  tmp = moyenne3(m, l, m + 5);
  printf("%d", tmp);
  exit();
}
```

Question 1 – Corps de la fonction

Donnez le code assembleur correspondant au corps de la fonction `moyenne3` en optimisant la variable `sum` dans le registre `$16`. Vous utiliserez les registres `$8`, `$9`, `$10` pour contenir des résultats de calculs ou des valeurs intermédiaires.
Solution:

Procédure pour l'implantation d'une fonction

0. Se représenter la pile à l'entrée de la fonction
1. Écrire le code du corps de la fonction en :
   (a) choisissant les registres qu'on veut associer avec les variables locales (si optimisées en registre)
   (b) écrivant les lectures et écriture en pile (de paramètres ou variables locales) en laissant un `??` pour l'immédiat de ces instructions de transfert mémoire (adresse relative au pointeur de pile)
   (c) mettant dans le registre `$_` le résultat de la fonction
2. Déterminer la taille du contexte de la fonction à partir de
   (a) `nr` le nombre de registres persistants utilisés dans le corps de la fonction
   (b) `na` le nombre max de mots nécessairee au stockage des arguments des fonctions appelées par la fonction. Pour determiner ce nombre max de mots, on calcul le nombre de mots pour les arguments de chaque fonction appelée comme suit : on compte 1 mot pour les 4 premiers arguments auquel on ajoute le nombre de mots permettant de stocker les arguments suivants en respectant leur contrainte d'alignement.
   (c) du nombre de mots `nv` nécessaire aux stockages variables locales, déterminable à partir du code source de la fonction
3. Écrire le prologue comportant :
   (a) l'allocation des emplacements sur la pile,
   (b) la sauvegarde des registres persistants et du registre `$__` (= écriture sur la pile) qui doivent être rangés par ordre croissant de leur numéro des adresses les plus petites aux plus grandes.

   NB : si besoin utiliser un dessin de la pile pour déterminer les emplacement des registres, les adresses des variables locales et celles des paramètres.
   (d) Si besoin, sauvegarder les 4 premiers paramètres dans leur emplacement
   (e) Si besoin, initialiser les variables locales
4. Dans le corps de la fonction, adapter les déplacements relatifs aux pointeurs de pile quand nécessaire (accès à aux variables locales, ou aux paramètres de la fonction).
5. Écrire l'épilogue soit dans l'ordre :
   (a) la restauration des registres (= lecture des valeurs sauvegardées sur la pile),
   (b) la désallocation des emplacements sur la pile,
   (c) le retour à l'appelant avec l'instruction `----------`

FIGURE 1 – Étapes préconisées pour l'implantation d'une fonction

```
        # sum = p + q + r;
        addu  $8, $4, $5
        addu  $16, $8, $6    # sum optimisee dans $16

        # return sum / 3;
        ori   $9, $0, 3
        div   $16, $9
        # resultat dans $2
        mflo  $2
```

Question 2 – Prologue

Déterminez combien il faut allouer d'octets sur la pile dans le prologue et écrivez le prologue de la fonction `moyenne3`.

Solution:

NB : la fonction commence par le prologue et précisément par l'étiquette qui porte le nom de la fonction.

Le corps utilise les registres `$8`, `$4`, `$5`, `$6`, `$16`, `$9`, `$0` et `$2`. Seul `$16` est à sauvegarder car c'est le seul qui est persistant, donc `nr` = 1. La fonction a une variable locale.

La fonction a une variable locale de type `int`, il faut allouer 4 octets/ 1 mot pour la zone des variables locales dans le contexte de la fonction (nv = 1 ou T_V = 4)

Il faut donc allouer ((nv = 1) + (nr = 1) + 1 pour sauvegarder `$31`) emplacements de 4 octets soit 3 * 4 = 12 octets sur la pile.

```
moyenne3:
        addiu $29, $29, -12
        sw    $31, 8($29)
        sw    $16  4($29)
        # @sum = $29 mais optimisee dans le registre $16

        # sum = p + q + r;
        addu  $8, $4, $5
        addu  $16, $8, $6    # sum optimisee dans $16

        # return sum / 3;
        ori   $9, $0, 3
        div   $16, $9
        # resultat dans $2
        mflo  $2
```

Question 3 – Epilogue

Écrivez l'épilogue de la fonction `moyenne3`.

Solution:

```
moyenne3:
        addiu $29, $29, -12
        sw    $31, 8($29)
        sw    $16  4($29)
        # @sum = $29 mais optimisee dans le registre $16

        # sum = p + q + r;
        addu  $8, $4, $5
        addu  $16, $8, $6    # sum optimisee dans $16
```

```
        # return sum / 3;
        ori   $9, $0, 3
        div   $16, $9
        # resultat dans $2
        mflo  $2

        lw    $16, 4($29)
        lw    $31, 8($29)
        addiu $29, $29, 12
        jr    $31
```

Question 4 – Appel de fonction avec des paramètres

Quelles sont les étapes à suivre pour écrire un appel à une fonction qui a des paramètres ?

Solution:

Lorsque la fonction appelée a des arguments, la fonction appelante est chargée d'écrire les valeurs des 4 premiers arguments dans les registres `$4` à `$7` et les suivants dans la pile. Une fois les paramètres mis dans les registres et sur la pile, l'appel peut avoir lieu (instruction `jal`).

NB : La pile contient toujours des emplacements pour TOUS les paramètres, les 4 premiers sont utilisés par les fonctions appelées pour sauvegarder leur paramètre lorsque nécessaire (en cas de code non optimisé ou pour sauvegarder leur valeur, même si dans ce cas l'utilisation d'un registre persistant pour cela (via une instruction de type `ori` par exemple) est plus performant, cela évite deux accès mémoire). Ces emplacements font partie du contexte d'une fonction et sont alloués une seule fois dans le prologue (allocation du contexte de la fonction) en considérant la taille maximale nécessaire pour tous les appels réalisés dans une fonction. Les emplacements dans la pile pour les paramètres sont désalloués dans l'épilogue, en meme temps que la désallocation du contexte.

Question 5 – Code du programme principal

Quelle est la taille du contexte du programme principal ? Justifiez votre réponse.

Donnez les directives de déclaration des variables globales et le code correspondant au programme principal (`main`).

Solution:

Le programme principal a une variable locale (`tmp` optimisée en registre) et effectue deux appels de fonction en passant 3 arguments à chaque appel. Il faut donc allouer 3 emplacements pour les paramètres (na = 3) et 1 emplacement pour les variables locales (nv = 1). Le programme principal ne sauvegarde pas les registres persistants, ni `$31`. Ce n'est pas ce que ferait *gcc* mais c'est la convention adoptée jusqu'à la semaine 8 incluses. Pour toutes ces raisons, le contexte a une taille de 16 octets.

NB : les étiquettes `adr1` et `adr2` présentes ici n'ont aucun rôle dans le programme. Elles permettent simplement de donner les valeurs du registre `$31` dans la représentation graphique de la pile.

```
.data
  n: .word 15
  m: .word -1
  l: .word 124

.text
  # main
        addiu $29, $29, -16 # nv = 1 (variable locale tmp optimisee dans $16)
                            # na = 3 (3 param pour moyenne3)
        # tmp = moyenne3(n, m, 5);
        lui   $3, 0x1001
        lw    $4, 0($3)     # 1er param = valeur de n
        lw    $5, 4($3)     # 2eme param = valeur de m
        ori   $6, $0, 5     # 3eme param = 5
        jal   moyenne3
adr1:
        ori   $16, $2, 0     # tmp = resultat
```

```
        #printf("%d", tmp);
        or    $4, $16, $0    # affichage de tmp
        ori   $2, $0, 1
        syscall

        #tmp = moyenne3(m, l, m+5);
        lui   $3, 0x1001     # $3 non persistant on doit recharger sa valeuer
        lw    $4, 4($3)      # 1er param = valeur de m
        lw    $5, 8($3)      # 2eme param = valeur de l
        addiu $6, $4, 5      # 3eme param = m + 5
        jal   moyenne3
adr2:
        ori   $16, $2, 0     # tmp = resultat

        # printf("%d", tmp);
        or    $4, $16, $0    # affichage du resultat
        ori   $2, $0, 1
        syscall

        # desallocation var locale + exit();
        addiu $29, $29, +16
        ori   $2, $0, 10
        syscall
```

Question 6 – Représentation graphique de la pile

Représentez graphiquement l'évolution de la pile au cours de l'exécution du programme. On fera grandir la pile "vers le bas". On rappelle que, par convention, le pointeur de pile pointe sur le sommet de la pile, c'est-à-dire sur la dernière case occupée de la pile au moment où l'on entre dans une fonction.

Solution:

L'évolution de la pile est représentée dans la Figure 2.

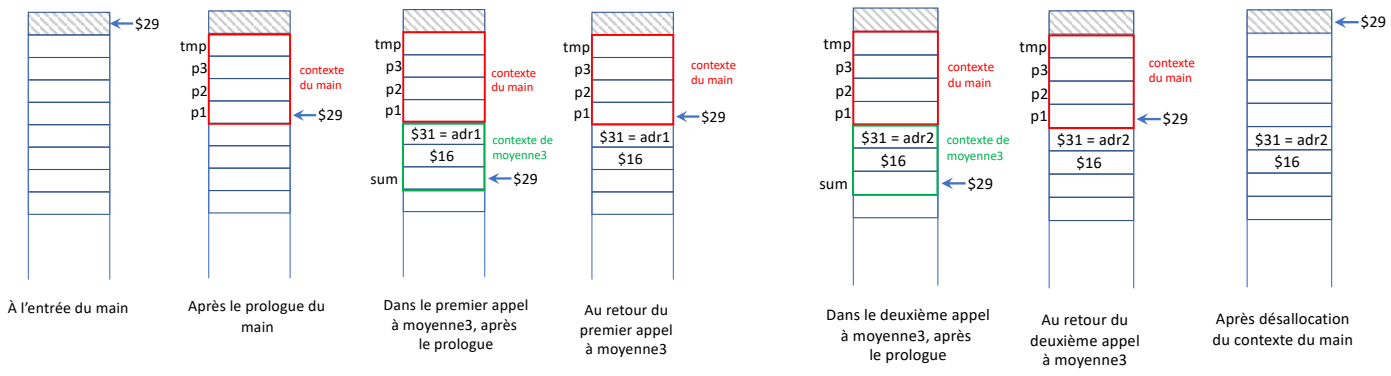

FIGURE 2 – Evolution de la pile

Exercice 3 – Moyenne de 3 et 5 nombres

On souhaite avoir en plus une fonction qui calcule la moyenne de 5 entiers. Voici le code du programme C que l'on considère désormais :

```c
/* trois variables globales */
int n = 15;
int m = -1;
int l = 124;

/* fonction moyenne3 */
int moyenne3(int p, int q, int r) {
   int sum = p + q + r;
   return sum / 3;
}

/* fonction moyenne5 */
int moyenne5(int p, int q, int r, int s, int t) {
   int sum = p + q + r + s + t;
   return sum / 5;
}

/* programme principal */
void main() {
  int tmp;
  tmp = moyenne3(n, m, 5);
  printf("%d", tmp);
  tmp = moyenne5(m, l, m + 5, 12, 35);
  printf("%d", tmp);
  exit();
}
```

Question 1

Quelle est conséquence de la différence du nombre de paramètres entre les fonctions `moyenne5` et `moyenne3` lors de l'écriture du corps de la fonction `moyenne5` ?
En suivant les étapes préconisées, donnez le code du corps la fonction `moyenne5` en utilisant le registre $10 pour contenir la valeur du paramètre `t`. Puis, donnez le prologue et l'épiloque de la fonction `moyenne5`.

Solution:

La fonction a 5 arguments, le 5ème argument est passé via la pile, alors que la valeur des 4 premiers se trouve dans les registres $4 à $7. Il faut donc lire la valeur du 5ème argument avec une lecture mémoire. L'emplacement du 5ème paramètre est fixe par rapport au pointeur de pile en entrant dans la fonction ($29 + 16). Il est plus aisé de calculer son adresse relative au pointeur de pile si on fait un schéma comme préconisé dans les recommandations pour écrire le code d'une fonction. Lors de l'écriture du code de la fonction, on lit le paramètre sur la pile sans indiquer le déplacement relatif au pointeur de pile (16 + taille du contexte de la fonction) qu'on ajuste une fois que le prologue est écrit.

Le code de la fonction `moyenne5` est donné ci-dessous :

```
moyenne5:
        addiu $29, $29, -12
        sw    $31, 8($29)
        sw    $16  4($29)
        # @sum = $29 mais sum est optimisee dans le registre $16
        # lecture du 5eme parametre s dans $10
        lw    $10, 28($29)
        # sum = p + q + r + s + t;
        addu  $8, $4, $5    # P + q
        addu  $8, $8, $6    # p + q + r
        addu  $8, $8, $7    #  p + q + r + s + t
        addu  $16, $8, $10  # sum optimisee dans $16 = (p + q + r + s) + t
```

```
        # return sum / 5;
        ori   $9, $0, 5
        div   $16, $9
        # resultat dans $2
        mflo  $2

        lw    $16, 4($29)
        lw    $31, 8($29)
        addiu $29, $29, 12
        jr    $31
```

Question 2
Quelle est la taille du contexte de ce nouveau programme principal ?
Donnez le code correspondant au programme principal.
Solution:

```
.text
  # main
        addiu $29, $29, -24 # nv = 1 (variable locale tmp optimisee dans $16)
  # na = 5 (4 premiers arguments = 4 mots + 1 mot pour 5eme arg)

        # tmp = moyenne3(n, m, 5);
        lui   $3, 0x1001
        lw    $4, 0($3)      # 1er param = valeur de n
        lw    $5, 4($3)      # 2eme param = valeur de m
        ori   $6, $0, 5      # 3eme param = 5
        jal   moyenne3
adr1:
        ori   $16, $2, 0     # tmp = resultat

        #printf("%d", tmp);
        or    $4, $16, $0    # affichage de tmp
        ori   $2, $0, 1
        syscall              # tmp est potenetiellement ecrase

        #tmp = moyenne5(m, l, m+5, 12, 35);
        lui   $3, 0x1001     # $3 non persistant on doit recharger sa valeuer
        lw    $4, 4($3)      # 1er param = valeur de m
        lw    $5, 8($3)      # 2eme param = valeur de l
        addiu $6, $4, 5      # 3eme param = m + 5
        ori   $7, $0, 12    # 4eme param = 12
        ori   $8, $0, 35     # 5eme param vaut 35
        sw    $8, 16($29)    # mise en pile 5eme param
        jal   moyenne5
adr2:
        ori   $16, $2, 0     # tmp = resultat

        # printf("%d", tmp);
        or    $4, $16, $0    # affichage du resultat
        ori   $2, $0, 1
        syscall

        # desallocation var sur la pile + exit();
        addiu $29, $29, +24
        ori   $2, $0, 10
        syscall
```

Question 3
Représentez la pile avant à l'entrée de la fonction `moyenne5` et après son prologue. **Solution**:
Schéma à faire...

Question 4
Que faudrait-il changer au code précédemment écrit si les paramètres de la fonction `moyenne5`, étaient déclarés comme des **`char`** (**donc des entiers signés codés sur un octet**) ?

Solution:
Il faut toujours allouer 4 octets / 1 mot pour les 4 premiers arguments. Pour les suivants, on a besoin d'autant d'octets que nécessaire pour les paramètres + le respect des contraintes d'alignement dictées par le type de l'argument. On alloue aussi toujours un nombre de mots pour respecter l'alignement du pointeur de pile. Si le 5ème argument était de type **`char`**, on allouerait donc aussi 4 octets pour lui (mais s'il y en avait un 6ème, alors on aurait besoin de 2 octets pour les 2, et donc on allouerait aussi toujours 1 mot sur le pile pour les 2).

Il est utile de faire ici un schéma de pile pour bien comprendre : même si on n'a besoin que d'un octet sur un emplacement de 4 octets, il est nécessaire de garder le pointeur de pile à une adresse multiple de 4. Afin d'éviter tout problème de compabilité, il est important lors de la lecture d'un paramètre en pile de lire en mémoire exactement le bon nombre d'octets (et donc d'utiliser la bonne instrucction, ici cela serait donc un `lb` pour récupérer le 5ème argument sur la pile).

Exercice 4 – Tableau en paramètre
On souhaite écrire en assembleur le programme suivant, qui calcule le nombre d'éléments d'un tableau d'entiers positifs et se terminant par -1.

```
int tab1[] = {23, 4, 5, -1};
int tab2[] = {2, 345, 56, 23, 45, -1};

int nb_elem(int tab[]) {
  int nb_elem = 0;
  int i = 0;
  while (tab[i] != -1) {
    nb_elem += 1;
    i += 1;
  }
  return nb_elem;
}

/* programme principal */
void main() {
  printf("%d", nb_elem(tab1));
  printf("\n"); /* affichage du caractère retour à la ligne */
  printf("%d", nb_elem(tab2));
  exit();
}
```

Question 1
Donner le code correspondant à la fonction `nb_elem`.
Solution:
Le point à souligner ici est le passage du paramètre : celui-ci est de type adresse et sa valeur correspond à l'adresse du premier élément du tableau.

```
nb_elem:
    addiu $29, $29, -12     # nv = 2 + nr = 0 + $31
    sw    $31, 8($29)

    # tab est dans $4
    # variable nb_elem optimisee dans $2
    # la variable i n'est pas utilisee

    xor   $2, $2, $2        # $2 / nb_elem = 0

bwhile:
    lw    $6, 0($4)         # lecture tab(i)
    bltz  $6, fin_for       # tab(i) > 0 ?
    addiu $2, $2, 1         # $2 / nb_elem++
    addiu $4, $4, 4         # tab++
    j     bwhile

fin_for :
    lw    $31, 8($29)
    addiu $29, $29, 12      # nv = 2 + nr = 2 + $31
    jr    $31
```

Question 2

Donner le code correspondant au `main` et aux déclarations des tableaux.

Solution:

À nouveau, le point à souligner ici est le passage du paramètre de type adresse.

```
.data
    tab1: .word 23, 4, 5, -1
    tab2: .word 2, 345, 56, 23, 2,  -1

.text
#main
    addiu $29, $29, -4      # na = 1, nv = 0
    lui   $4, 0x1001        # $4 = adresse tab1
    jal   nb_elem

    ori   $4, $2, 0
    ori   $2, $0, 1
    syscall

    ori   $4, $0, 0xA       # affichage retour a la ligne
    ori   $2, $0, 11
    syscall
    lui   $4, 0x1001        # adresse tab1
    ori   $4, $4, 16        # $4 = adresse tab2
    jal   nb_elem

    ori   $4, $2, 0
    ori   $2, $0, 1
    syscall

    addiu $29, $29, 4       # desallocation contexte
    ori   $2, $0, 10
    syscall
```