# TD4 : Variables globales et instructions mémoire

## Objectif(s)

- ✯ Familiarisation avec la déclaration de variables globales/réservation de mots mémoire.
- ✯ Familiarisation avec les instructions d'accès mémoire MIPS
- ✯ Tableaux et chaines de caractères
- ✯ Appels système de lecture/écriture de chaines de caractères

**Note pédagogique**:
Point important :

— La directive `.space n` n'a pas de contrainte d'alignement, donc il est nécessaire d'utiliser la directive `.align` quand on utilise la directive `.space` pour allouer des mots ou demi-mots qui eux ont des contraintes d'alignement (par exemple pour tableau de 5 mots non initialisés on doit allouer 20 octets → `.align 2` puis `.space 20`).

## Exercice(s)

## Exercice 1 – Déclaration de variables globales et réservation de mots mémoire

### Question 1

1. Rappelez les sections qui composent un programme assembleur et ce que l'on trouve dans ces sections.
2. Que deviennent ces différentes sections à l'exécution ? Où sont elles implantées ?

**Solution**:
Il y a deux types de section dans un programme assembleur.

1. La section `.data` contient les déclarations (allocations mémoire et initialisation) de variables globales du programme. Nous prendrons l'habitude de toujours la mettre tout en haut d'un fichier assembleur. La section `.text` contient les instructions du programme. Il y a au moins un programme principal, il peut aussi y avoir des fonctions.
2. À l'exécution la section `.data` est implantée en mémoire à partir de l'adresse `0x10010000` (segment data) et la section `.text` est implantée à partir de l'adresse `0x00400000` (segment code). Dans le simulateur MARS, cette adresse constitue le point d'entrée par défaut du programme : l'exécution commence par l'instruction qui s'y trouve.

### Question 2

1. Expliquez, à l'aide de votre mémento, les différentes directives de déclaration de variables globales/réservation de zones mémoire.
2. Déclarez une variable globale de type entier et initialisée à 10 en décimal.

3. Déclarez une variable globale de type entier non initialisée.
4. Déclarez une zone mémoire de 10 octets non initialisée.
5. Déclarez une variable globale contenant la chaine de caractères “toto”.
6. Que représentent les étiquettes que l’on utilise pour déclarer les variables globales ?
7. Déclarez 3 entiers consécutifs de telle sorte que l’adresse du premier soit `tab`. Quelle est l’adresse du 2ème entier ?
8. Rappelez comment se calcule l’adresse d’implantation d’une déclaration puis donnez les adresses d’implantation de toutes les déclarations précédentes.

**Solution**:

1. Les différentes directives sont : .space, .word, .half, .byte, .ascii, .asciiz. Elles sont expliquées dans le mémento.

   La directive .space n alloue n octets et les initialise à 0, qui est la valeur par défaut pour les données non initialisées. Cette directive est utilisée pour déclarer des données non initialisées avec des valeurs explicites.

   Cette directive ne prend pas en compte la contrainte d’alignement concernant l’adresse de la donnée déclaréee (la zone allouée commence donc à l’adresse courante), tandis que les directives .word et .half respectent la contrainte d’alignement (l’adresse de la donnée est un multiple de sa taille) : l’adresse retournée par .word est donc un multiple de 4, celle retournée par .half est un multiple de 2.

   Il est donc nécessaire de préciser les contraintes d’alignement des données de type mots ou demi-mots déclarées avec la directive .space. La précision d’une contrainte d’alignement se fait avec la directive .align n qui signifie que l’adresse de la donnée suivante doit être un multiple de $2^n$. Ainsi, il faut utiliser la directive .align 2 juste avant de déclarer, avec la directive .space, des mots non initialisés, et la directive .align 1 avant de déclarer des demi-mots non initialisés.

2. ```
   .data
   mot1 :     .word  10   # ou .word 0xA, valeur d'initialisation obligatoire
              .align 2    # obligatoire pour la déclaration suivante
   mot2 :     .space  4   # variable de type entier non initialisée
                          # (donc mise à 0 par défaut),
                          # 1 entier =  4 octets
   espace:    .space 10
   chaine:    .ascii  "toto" # pas de caractère de fin de chaine à la fin
   chaine2:   .asciiz "toto" # caractère de fin de chaine mis à la fin
   ```

6. Les étiquettes représentent les adresses des mots mémoires déclarés (1 variable = 1 contenant = 1 espace mémoire qui a une adresse). Ce sont des valeurs sur 32 bits, manipulables (visibles) par les instructions. L’étiquette de la zone mémoire qui contient la valeur d’une variable portera bien souvent le même nom que la variable elle-même alors qu’elle désigne le contenant (adresse).
7. `tab:  .word 1, 2, 3`
8. la section `.data` est implantée en mémoire à l’adresse 0x1001 0000. En première approximation, on peut dire que l’adresse d’une variable est égale à la somme de l’adresse de la variable précédemment déclarée et de la taille en octet de la variable précédemment déclarée. Ceci n’est vrai que s’il n’y a pas de problème d’alignement (ce qui est majoritairement le cas avec les programmes qu’on demande dans cette ue). Le problème d’alignement vient des contraintes d’alignement évoquées au point 1 de cette question. A noter que pour un octet il n’y a pas de contrainte sur son adresse. Ainsi, si on déclare d’abord un octet puis un mot alors l’adresse du mot sera égale à l’adresse de l’octet + 1 (taille de l’octet) + 3 (octets vides pour réaliser l’alignement du mot).

   Les contraintes d’alignement sont implicites pour les directives .word et .half, et sont explicites lorsqu’il y a l’utilisation d’un .align (qui ne peut réduire l’alignement imposé par une directive mais peut l’augmenter. Par exemple, .align 3 devant un .word 12 impose une adresse multiple de 8 pour le mot, .align 1 devant .word 12 ne sert à rien)

   Rappel des tailles :
   — un mot (word) : 4 octets,
   ```

— un demi_mot (half) : 2 octets,
— un octet (byte) : 1 octet,
— un caractère : 1 octet

Puisque dans ce module on calcule manuellement l'adresse de toutes les variables globales, on pourrait se passer d'étiquettes mais cela rendrait le code moins lisible. Elles sont aussi utiles avec MARS qui donne l'adresse associée à une étiquette, cela permet de vérifier les calculs manuellement faits.

Code C correspondant (en C, une variable globale non initialisée est initialisée à 0 par défaut) :

```
int mot1 = 10;
int mot2;
char espace[10];
char chaine[] = { 't', 'o', 't', 'o' };
char chaine2[] = "toto";
int tab[] = { 1, 2, 3 };

void main() {

    ...
    exit();
}
```

## Exercice 2 – Instructions MIPS de lecture/écriture en mémoire

Les instructions mémoire permettent de lire ou d'écrire des valeurs en mémoire à une adresse donnée, c'est-à-dire de lire ou d'écrire une valeur dans les variables déclarées dans la section `.data`.

### Question 1

Faites le tour des instructions d'accès mémoire en lecture et en écriture de votre mémento. Combien d'opérandes sources et destination ont ces instructions ? De quelle nature sont-ils ? À quoi correspondent les opérandes ?

**Solution**:

Les instructions de lecture sont `lw, lh, lhu, lb, lbu` et sont de la forme $op\ r_{dest},\ imm(r_{addr})$. Il y a deux opérandes source : un immédiat et un registre dont le contenu, ajouté à l'immédiat, donne l'adresse accédée en mémoire. Le registre destination récupère la valeur lue en mémoire.

Les opérations d'écriture sont `sw, sh, sb` et sont de la forme $op\ r_{src},\ imm(r_{addr})$. Il n'y a que des opérandes source : un registre et un immédiat pour calculer l'adresse mémoire où écrire et un registre qui contient la valeur à écrire.

### Question 2

Quel est le format de codage de ces instructions ? Donnez le codage de l'instruction `sw $9, 4($8)`.

**Solution**:

Toutes les instructions d'accès mémoire sont codées au format I. Le codage de l'instruction demandée est `0xAD090004` :

| OPCOD | Rs | Rt | Imm |
|---|---|---|---|
| 1010 11 | 01 000 | 0 1001 | 0000 0000 0000 0100 |

### Question 3

Quelle est la différence entre les instructions `lw`, `lh`, `lb`, `lhu` et `lbu` ?

**Solution**:

Les valeurs lues/écrites peuvent être de différentes tailles (1, 2 ou 4 octets), il existe donc des instructions correspondant à ces différentes tailles.

Un mot (w pour word) représente une valeur sur 4 octets / 32 bits, un demi-mot (h pour half-word) une valeur sur deux octets ou b (pour byte) signifie un seul octet. La présence du "u" dans le code de l'opération indique que les nombres manipulés sont des entiers naturels. L'absence de "u" dans le code de l'opération indique que les nombres manipulés sont des entiers relatifs. Le nom de l'opération correspond donc à load-tailledumot-u_si_valeur_non_signée. Les lectures sans "u" étendent donc sur les bits de poids fort du registre le bit de signe de l'octet ou demi-mot lu. Pour les lectures avec un "u", les bits de poids fort sont mis à 0.

## Question 4

L'adresse du mot à lire/écrire est la somme d'une valeur contenue dans un registre (adresse de base) et d'un déplacement (valeur immédiate). Par exemple $lw\ r_{dest},\ imm(r_{addr})$. Pour pouvoir exécuter cette instruction, il faut que le registre $r_{addr}$ contienne une valeur telle que sa somme avec l'immédiat corresponde à une adresse mémoire valide. Avant tout accès à la mémoire, il faut donc charger l'adresse de base dans le registre $r_{addr}$.

1. Quelle est la taille d'une adresse mémoire ? Donnez la suite d'instructions permettant de mettre dans le registre $8 l'adresse d'un mot mémoire implanté à l'adresse 0x1001 0004 correspondant à la variable `var1`.
2. Écrivez un programme assembleur avec la déclaration d'un mot mémoire d'adresse `var1` initialisé à 0xFF et d'un mot d'adresse `var2` non initialisé. Le programme principal charge dans $9 la valeur contenue en mémoire à l'adresse `var1`, ajoute 5 à cette valeur et stocke le résultat en mémoire à l'adresse `var2`.
3. Donnez le code C correspondant au programme que vous avez écrit.

**Solution**:

1. Une adresse est sur 32 bits, on doit donc utiliser 2 instructions pour la charger.

```
lui $8, 0x1001     # pour charger les 16 bits de poids fort
ori $8, $8, 0x0004 # pour charger les 16 bits de poids faible
```

2. La directive .word sert à l'allocation de mots de 4 octets initialisés, il faut donc utiliser .space pour la réservation d'espace mémoire non initialisé (qui sera mis à 0 par défaut)

```
.data
   var1:    .word 0xFF    # Adresse 0x10010000
            .align 2      # pour respecter la contrainte d'alignement des mots
   var2:    .space 4      # Adresse 0x10010004

.text
   lui   $8, 0x1001
   lw    $9, 0($8)       # Le 0 est nécessaire lorsqu'on interdit
                         # Les pseudo-instructions dans Mars
   addiu $9, $9, 5
   lui   $8, 0x1001      # Facultatif
   ori   $8, $8, 0x0004
   sw    $9, 0($8)       # ou directement sw $9, 4($8)

   ori   $2, $0, 10      # Fin du programme principal
   syscall
```

3. Voici le programme C équivalent :

```
int var1 = 0xFF;
int var2;            /* globale non initialisée <=> initialisée à 0 */

void main() {
```

```
    var2 = var1 + 5  /* lit le contenu de la variable globale var1 */
                     /* stocke le résultat dans la variable globale var2 */
    exit();
}
```

## Exercice 3 – Contenu mémoire et rangement mémoire

Soit le programme suivant :

```
.data
   var1: .word 0xCCDDEEFF
   var2: .byte 0x11
   var3: .byte 0x22
   var4: .byte 0x33
   var5: .byte 0x44
   var6: .asciiz "123"

.text
   lui   $8, 0x1001
   lw    $9, 0($8)
   lw    $10, 4($8)
   lb    $11, 0($8)
   lbu   $12, 0($8)
   addiu $12, $12, 1
   addiu $11, $11, 1
   sw    $11, 0($8)
   sb    $9, 7($8)

   ori   $2, $0, 10
   syscall
```

### Question 1

Donnez le contenu de la mémoire, octet par octet et mot par mot, après le chargement du programme et avant son exécution :

| Adresse | octet n | octet (n+1) | octet (n+2) | octet (n+3) | mot |
|---|---|---|---|---|---|
| 0x10010000 | | | | | |
| 0x10010004 | | | | | |
| 0x10010008 | | | | | |
| 0x1001000c | | | | | |

**Solution**:

Cet exercice met en évidence le caractère petit boutien du processeur. Les octets de poids faible d'un mot sont rangés aux adresses les plus petites. L'affichage par mot inverse l'ordre des 4 octets car dans un mot l'octet de poids fort est tout à gauche.

| Adresse | octet n | octet (n+1) | octet (n+2) | octet (n+3) | mot |
|---|---|---|---|---|---|
| 0x10010000 | 0xFF | 0xEE | 0xDD | 0xCC | 0xCCDDEEFF |
| 0x10010004 | 0x11 | 0x22 | 0x33 | 0x44 | 0x44332211 |
| 0x10010008 | 0x31 | 0x32 | 0x33 | 0x00 | 0x00333231 |
| 0x1001000c | 0x00 | 0x00 | 0x00 | 0x00 | 0x00000000 |

### Question 2

Déroulez à la main l'exécution du programme et donnez le contenu des registres à la fin du programme ainsi que le contenu de la mémoire (faites évoluer le contenu de la mémoire instruction par instruction).

| registre | $8 | $9 | $10 | $11 | $12 |
|---|---|---|---|---|---|
| contenu | | | | | |

| Adresse | octet n | octet (n+1) | octet (n+2) | octet (n+3) | mot |
|---|---|---|---|---|---|
| 0x10010000 | | | | | |
| 0x10010004 | | | | | |
| 0x10010008 | | | | | |
| 0x1001000c | | | | | |

**Solution**:

```
.data
   var1: .word 0xCCDDEEFF
   var2: .byte 0x11
   var3: .byte 0x22
   var4: .byte 0x33
   var5: .byte 0x44
   var6: .asciiz "123"

.text
   lui   $8, 0x1001
   lw    $9, 0($8)      # $9 = 0xCCDDEEFF
   lw    $10, 4($8)     # $10 = 0x44332211
   lb    $11, 0($8)     # $11 = 0xFFFFFFFF => octet signé donc le bit de signe
                        # est étendu sur les bits de poids fort du registre
   lbu   $12, 0($8)     # $12 = 0xFF => octet non signé donc les bits de poids
                        # fort du registre sont mis a 0
   addiu $12, $12, 1    # $12 = 256 ou 0x100
   addiu $11, $11, 1    # $11 = 0
   sw    $11, 0($8)     # le premier mot mémoire est mis à 0
   sb    $9, 7($8)      # le 8eme octet est mis à FF

   ori   $2, $0, 10
   syscall
```

| registre | $8 | $9 | $10 | $11 | $12 |
|---|---|---|---|---|---|
| contenu | 0x10010000 | 0xCCDDEEFF | 0x44332211 | 0x00000000 | 0x00000100 |

| Adresse | octet n | octet (n+1) | octet (n+2) | octet (n+3) | mot |
|---|---|---|---|---|---|
| 0x10010000 | 0x00 | 0x00 | 0x00 | 0x00 | 0x00000000 |
| 0x10010004 | 0x11 | 0x22 | 0x33 | 0xFF | 0xFF332211 |
| 0x10010008 | 0x31 | 0x32 | 0x33 | 0x00 | 0x00333231 |
| 0x1001000c | 0x00 | 0x00 | 0x00 | 0x00 | 0x00000000 |

## Exercice 4 – Accès à un tableau, codage de caractère

Remarque :
- On fait l'hypothèse que le type int en C est codé sur 4 octets (c'est généralement le cas sur les architectures 32 bits)
- Un tableau `tab` est un ensemble de valeurs rangées consécutivement en mémoire et `tab[0]` désigne le premier élément du tableau.
- Une chaîne de caractères `ch` est un tableau de caractères, déclarée comme un tableau, par exemple `char ch[] = "exemple";`.

### Question 1

Écrivez un programme assembleur correspondant au code C suivant :

```
int tab[] = { 1, 2, 34, 256, -1 }; /* tableau d'entiers */
char chaine[] = "toto";            /* chaîne de caractères */

void main() {
   printf("%d", tab[3]);
   printf("%d", chaine[2]);
   exit();
}
```

**Solution**:

```
.data
   tab:     .word 1, 2, 34, 256, -1   # 0x10010000 -> 20 octets
   chaine:  .asciiz "toto"            # 0x10010014

.text
   lui $8, 0x1001          # $8 = tab
   lw  $4, 12($8)          # tab[3] = *(tab + 3 x taille)
   ori $2, $0, 1
   syscall

   lui $8, 0x1001
   ori $8, $8, 0x0014      # $8 = chaine
   lb  $4, 2($8)
   ori $2, $0, 1
   syscall

   ori $2, $0, 10
   syscall
```

### Question 2

Quelles sont les valeurs affichées lors de l'exécution ?

**Solution**:

La valeur 256 est affichée (en décimal) et la valeur 116 qui correspond à 0x74 c'est-à-dire le codage ASCII du caractère 't'. Vous pouvez voir cela sur votre mémento. Vous aurez besoin de ce dernier pour certains exercices de TD/TME et le jour de l'examen.

## Exercice 5 – Appels système autour des chaines de caractères

### Question 1

1. Écrivez un programme avec une variable globale de type chaine de caractères contenant la valeur "test d'affichage". Écrivez le programme principal qui affiche la chaîne de caractères à l'aide de l'appel système correspondant (cherchez-le dans le mémento).
2. Modifiez votre programme en déclarant une zone mémoire non allouée de 32 octets. Le programme lit au clavier une chaîne de caractères qu'il stocke dans cette zone mémoire (cherchez l'appel système correspondant dans votre mémento). Ensuite, il affiche le résultat de la lecture (la chaine de caractères lue) à l'aide de l'appel système correspondant (voir le mémento).

**Solution**:

L'affichage d'une chaîne de caractères se fait avec l'appel système numéro 4 et l'adresse du premier caractère de la chaine doit être mise au préalable dans le registre `$4`.

L'affichage d'une chaîne de caractères affiche les caractères correspondant aux valeurs de tous les octets consécutifs stockés à partir de l'adresse contenue dans `$4`, et jusqu'à rencontrer l'octet nul : il est donc important de déclarer la chaine avec la directive `.asciiz` et non la directive `.ascii`. D'une manière générale, il est très rare que l'on veuille utiliser la directive `.ascii`.

La lecture d'une chaine de caractères au clavier se fait avec l'appel système numéro 8. Le registre `$4` doit contenir l'adresse de la zone mémoire allouée pour stocker la chaine lue et le registre `$5` doit contenir la taille de cette zone mémoire. Cette valeur fixe le nombre maximum de caractères que l'appel système pourra lire (les caractères supplémentaires seront ignorés), sachant qu'un octet est automatiquement réservé pour stocker la valeur `0` en fin de chaine. Une taille de `8` permet donc la saisie de `7` caractères. Si la chaîne saisie en comporte moins, le retour chariot (code ASCII `0A`) est stocké avec la chaine.

```
.data
   ma_chaine:     .asciiz  "test d'affichage"
   chaine_a_lire: .space   32

.text
   # alinea 1
   lui $8, 0x1001        # $8 = ma_chaine

   ori $4, $8, 0         # $4 = parametre de l'appel systeme
   ori $2, $0, 4         # appel systeme pour afficher une chaine de caracteres
   syscall

   # alinea 2
   lui $8, 0x1001
   ori $8, $8, 0x11      # $8 = chaine_a_lire

   ori $4, $8, 0         # transfert de $8 dans $4
   ori $5, $0, 32        # espace alloue a l'adresse chaine_a_lire
   ori $2, $0, 8         # appel systeme pour lire une chaine de caracteres
   syscall               # lecture de la chaine

   lui $4, 0x1001
   ori $4, $4, 0x11      # $4 = @chaine_a_lire
   ori $2, $0, 4         # appel systeme pour afficher une chaine de caracteres
   syscall               # affichage de la chaine lue

   ori $2, $0, 10        # fin du programme
   syscall
```