TD6 : Variable locales et pile d'exécution, tableau et codage ASCII

Objectif(s)

- ✯ Notion de pile et implantation des variables locales
- ✯ Traduction littérale de code C
- ✯ Optimisation des variables locales et des accès mémoire
- ✯ Manipulations de tableau, en variable globale puis locale
- ✯ Codage ASCII et chaîne de caractères
- ✯ Correspondance entre un caractère ASCII représentant un chiffre et la valeur numérique du chiffre ; utilisation de masque
- ✯ Recopie mémoire

Exercice(s)

Exercice 1 – Pile d'exécution, variables locales et traduction littérale de code C

Question 1

Qu'est ce que la pile d'exécution ? À quoi sert-elle ? Comment s'en sert-on ?
Quelles sont les premières instructions d'un programme (ou d'une fonction) ? Quel est leur but ? Que doivent faire les dernières instructions d'un programme ?

Solution:

La pile d'exécution est un segment en mémoire géré comme une pile (LIFO) : on y alloue des mots mémoire quand on en a besoin, on les désalloue lorsqu'on n'en a plus besoin. L'allocation de mémoire en pile est dynamique : elle est réalisée par des instructions du programme.

La pile sert à stocker les contextes d'exécution des fonctions et du programme principal qui contient notamment (et que ça dans ce TD) les emplacements pour les variables locales.

Le sommet de pile correspond au dernier emplacement alloué, le registre $29, appelé "pointeur de pile", contient son adresse. La pile croit vers les adresses décroissantes : on alloue de la place en décrémentant le pointeur de pile, on désalloue de la place en l'incrémentant. Pour stocker ou lire des valeurs en pile, on réalise des accès mémoire avec une adresse relative au pointeur de pile.

Les premières instructions du programme principal (d'une fonction) correspondent au prologue qui alloue la place en pile pour le contexte, pour les variables locales donc. Les premières instructions initialisent les variables lorsque celles-ci ne sont pas optimisées en registre.

Les dernières instructions d'un programme (d'une fonction) doivent désallouer le contexte sur la pile. Elles font partie de ce qu'on appelle l'épilogue.

Question 2

Soit un programme principal qui a trois variables locales `a`, `b` et `c` de type `int`, `char`, `short` déclarées dans cet ordre et initialisées avec les valeurs 12, 3 et 5 respectivement. Quelles seront les premières instructions ainsi que

les dernières instructions du programme principal ? Pendant l'exécution du programme, quelles seront les adresses d'implantation des variables locales ?

Solution:

Les premières instructions allouent les emplacements sur la pile pour les 3 variables. Il faut 4 octets pour `a`, 1 pour `b` et 2 pour `c` qui doit être alignée sur une frontière de demi-mot. La première variable déclarée est celle qui se trouve au sommet, et l'alignement se calcule comme celui des variables globales : il y a donc un alignement de 1 octet (*padding*) entre `b` et `c`.

L'allocation sera donc de 8 octets. La variable `a` se trouve à `$29`, `b` se trouve à `$29 + 4`, et `c` se trouve à `$29 + 6` Ensuite les variables sont initialisées avec des écritures à leur emplacement respectif sur la pile.

```
.text
addiu $29, $29, -8
ori   $8, $0, 12
sw    $8, 0($29)    # a <- 12
ori   $8, $0, 3
sb    $8, 4($29)    # b <- 3
ori   $8, $0, 5
sh    $8, 6($29)    # c <- 5


# epilogue (non demandé)
addiu $29, $29, 8
ori $2, $0, 10
syscall
```

Question 3

Rappelez les règles de traduction littérale en assembleur d'une instruction C de type assignation, par exemple `var1 = var2 + var3`.

Solution:

La traduction littérale est de lire en mémoire la valeur de `var2` et de `var3` (à leur emplacement, dans le segment de données ou la pile selon la nature des variables) , puis de faire l'addition dont le résultat est finalement écrit en mémoire à l'emplacement de `var3`.

Exercice 2 – Chaine de caractères et codage ASCII

Question 1

- Donnez le codage ASCII de la chaine `"AaBb"`
- Quel est le codage ASCII correspondant aux chiffres 0, 1, ..., 9 ?
- Donnez le codage ASCII de la chaine de caractères `"1024"`
- Quelle est l'adresse mémoire du ième caractère d'une chaine ?
- Comment retrouver à partir du codage ASCII d'un chiffre (par exemple le codage du caractère '3') la valeur de l'entier correspondant (3 dans l'exemple) ?

Solution:

La chaine "AaBb" se code en un tableau d'octets : 0x41 0x61 0x42 0x62 0x00 (l'octet à zéro est ajouté si on a utilisé la directive `.asciiz` pour la déclaration).

La chaine "1024" se code en 0x31 0x30 0x32 0x34 0x00

L'adresse du premier caractère d'une chaine est **ch**, celle du deuxième est **ch+1**, du ième est **ch+i-1**.

Les chiffres 0 .. 9 ont respectivement pour code ASCII 0x30 .. 0x39

Pour retrouver la valeur du chiffre, il faut **masquer** (= mettre à 0) les 4 bits de poids fort de l'octet représentant le codage ASCII (8 bits = 1 octet). La valeur ASCII étant dans un registre de 32 bits, on masque en fait les 28 bits de poids forts (mais les 24 premiers sont déjà à 0 si on a fait un `lbu`). La notion de masque (= mise à zéro de certains bits) est mise en œuvre avec les opérateurs `and` ou `andi` (pour une opération logique, il n'y a pas d'extension de signe de l'immédiat sur 16 bits. `andi` masque donc systématiquement les 16 bits de poids fort. Si on veut les conserver, il faut utiliser `and`).

```
andi r,r,0x000F   # met à zéro les bits 4 jusqu'à 31 dans le registre r
```

Exercice 3 – Calcul du nombre associé à une chaîne de caractères représentant un nombre et affichage de la valeur correspondante

Question 1

On considère le programme C ci-dessous qui comporte une variable globale `ch` de type chaine de caractères représentant un nombre entier positif. Le programme principal calcule la valeur numérique correspondante puis affiche cette valeur. Écrivez un programme assembleur qui correspond à la traduction littérale du code ci-dessous.

```
char ch[] = "1234";

void main() {
   int i = 0;
   int val = 0;
   char c;

   while (ch[i] != 0) {
      c = ch[i];
      c = c & 0x0F; /* récupération de la valeur du chiffre */
      val = val * 10 + c;
      i += 1;
   }
   printf("%d", val);
   exit();
}
```

Remarque:
Ce programme utilise le fait que :

- 3456 = (((3 * 10 + 4) * 10 + 5) * 10 + 6)
- une chaine de caractères déclarée par la directive `.asciiz` se termine toujours par le caractère nul (qui vaut 0) qui correspond au caractère de fin de chaine.

Solution:

```
# passage d'une chaine de caracteres (variable globale) representant un nombre
# entier positif au nombre entier correspondant

.data
    ch: .asciiz "1234"      # resultat en hexa 0x4d2

.text
    addiu $29, $29, -12     # 3 var. locales : i, val et c

    sw    $0, 0($29)        # i <- 0
    sw    $0, 4($29)        # val <- 0

loop:
```

```
    # ch[i] != 0
    lw    $9, 0($29)        # $9 <- lecture de i
    lui   $8, 0x1001
    ori   $8, $8, 0x0000    # $8 <- @ch
    addu  $9, $8, $9        # $9 <- @ch[i]
    lbu   $9, 0($9)         # $9 <- ch[i] chargement du caractere i
    beq   $9, $0, finloop   # si vaut 0, fin de la chaine c'est fini

    # c = ch[i]
    lw    $9, 0($29)        # $9 <- lecture de i
    lui   $8, 0x1001
    ori   $8, $8, 0x0000    # $8 <- @ch
    addu  $9, $8, $9        # $9 <- @ch[i]
    lbu   $9, 0($9)         # $9 <- ch[i] chargement du caractere i
    sb    $9, 8($29)        # c <- ch[i]

    # c = c & 0x0F
    lb    $9, 8($29)        # lecture de c
    andi  $9,$9,0x0F
    sb    $9, 8($29)        # ecriture de c

    # val = val * 10 + c
    ori   $10, $0, 10       # $10 <- 10
    lw    $11, 4($29)       # $11  <- val
    multu $11, $10          # multiplication du val par 10
    mflo  $11               # $11 <- val * 10
    lb    $12, 8($29)       # $12 <- c
    addu  $12, $12, $11     # $12 <- val * 10 + c
    sw    $12, 4($29)       # val <- val * 10 + c

    # i += 1
    lw    $9, 0($29)        # $9 <- lecture de i
    addiu $9, $9, 1         # $9 <- i + 1
    sw    $9, 0($29)        # ecriture de i

    j     loop
finloop:
    lw    $4, 4($29)        # val a afficher doit etre mis dans $4
    ori   $2, $0, 1         # code de l'appel systeme affichage d'un entier
    syscall

    addiu $29, $29, 12
    ori   $2, $0, 10
    syscall
```

Question 2

Donnez une version dans laquelle toutes les variables locales sont optimisées en registre.

Solution:

Version avec les variables locales dont la valeur réside dans un registre dédié pour chacune. Cela nécessite de changer un peu les registres utilisés dans la fonction.

```
# passage d'une chaine de caracteres (variable globale) representant un nombre
# entier positif au nombre entier correspondant

.data
    ch: .asciiz "1234"       # resultat en hexa 0x4d2

.text
```

```
    addiu $29, $29, -12      # 3 var. locales : i, val et c

    ori   $9,  $0, 0         # i optimisee dans $9 <- 0
    ori   $10, $0, 0         # val optimisee dans $10 <- 0

loop:
    # ch[i] != 0
    lui    $8, 0x1001
    ori    $8, $8, 0x0000    # $8 <- @ch
    addu   $11, $8, $9       # $11 <- @ch + i
    lbu    $11, 0($11)       # $11<- ch[i]
    beq    $11, $0, finloop  # si vaut 0, fin de la chaine c'est fini

    # c = ch[i]
    lui   $8, 0x1001
    ori   $8, $8, 0x0000     # $8 <- @ch
    addu  $11, $8, $9        # $11 <- @ch + i
    lbu   $11, 0($11)        # $11 <- ch[i]
    ori   $12, $11, 0        # c optimisee dans $12 <- ch[i]

    # c = c & 0x0F
    andi  $12, $12, 0x0F     # $12 (c) <- c & 0x0F

    # val = val * 10 + c
    ori   $11, $0, 10        # $11 <- 10
    multu $11, $10           # multiplication du val par 10
    mflo  $11                # $11 <- val * 10
    addu  $10, $12, $11      # S10 (val) <- val * 10 + c

    # i += 1
    addiu $9, $9, 1          # $9 (i) <- i + 1
    j     loop
finloop:
    ori   $4, $10, 0         # val a afficher doit etre mis dans $4
    ori   $2, $0, 1          # code de l'appel systeme affichage d'un entier
    syscall

    addiu $29, $29, 12
    ori   $2, $0, 10
    syscall
```

Question 3

Donnez une version dans laquelle les calculs ou chargements mémoire redondants sont éliminés. Quels autres types d'optimisation peut-on faire ?

Solution:

On peut éviter de lire une deuxième fois `ch[i]` en conservant sa valeur dans un registre (`$11` dans la solution fournie), on économise de recalculer son adresse en même temps.

On peut éviter de recalculer `ch` et sortir ce calcul de la boucle car il ne change pas d'une itération à une autre, idem on peut charger 10 dans un registre en dehors de la boucle (pour information, cette optimisation qui sort des calculs d'une boucle s'appelle du *code hoisting*).

```
# passage d'une chaine de caracteres (variable globale) representant un nombre
# entier positif au nombre entier correspondant

.data
    ch: .asciiz "1234"       # resultat en hexa 0x4d2
```

```
.text
    addiu $29, $29, -12      # 3 var. locales : i, val et c

    ori   $9,  $0, 0         # i <- 0
    ori   $10, $0, 0         # val <- 0

    lui   $8, 0x1001
    ori   $8, $8, 0x0000     # $8 <- @ch   # independant de la boucle, hoisting
    ori   $14, $0, 10        # $14 <- 10  # independant de la boucle, hoisting

loop:
    # ch[i] != 0
    addu  $11, $8, $9        # $9 <- @ch + i
    lbu   $11, 0($11)        # $9 <- ch[i] chargement du caractere i
    beq   $11, $0, finloop   # si vaut 0, fin de la chaine c'est fini

    # c = ch[i]
    ori   $12, $11, 0        # c <- ch[i]

    # c = c & 0x0F
    andi  $12,$12,0x0F

    # val = val * 10 + c
    multu $14, $10           # multiplication de val par 10
    mflo  $13                # $13 <- val * 10
    addu  $10, $12, $13      # val <- val * 10 + c

    # i += 1
    addiu $9, $9, 1          # $9 <- i + 1
    j     loop
finloop:
    ori   $4, $10, 0         # val a afficher doit etre mise dans $4
    ori   $2, $0, 1          # code de l'appel systeme affichage d'un entier
    syscall

    addiu $29, $29, 12
    ori   $2, $0, 10
    syscall
```

Question 4

On considère désormais la variante ci-dessous dans laquelle la chaine de caractère est locale au programme principal. Que faut-il changer au programme écrit à la question précédente pour obtenir un code optimisé correspondant à cette variante ?

```
void main() {
   int i = 0;
   int val = 0;
   char c;
   char ch[] = "1234";

   while (ch[i] != 0) {
      c = ch[i];
      c = c & 0x0F; /* récupération de la valeur du chiffre */
      val = val * 10 + c;
      i += 1;
   }
   printf("%d", val);
```

```
   exit();
}
```

Solution:

Les modifications à apporter correspondent uniquement à l'allocation de la chaine sur la pile et son initialisation (écriture des caractères un à un), puis la sauvegarde de l'adresse de la chaine (adresse relative au pointeur de pile) dans le même registre que celui utilisé lorsqu'elle était globale. Bien sûr, la section de données est désormais vide.

La chaine a une taille de 5 octets. Mais elle est déclarée juste après `c` qui est un caractère. Son adresse est donc celle de `c` + 1 (soit `$29` + 9). Comportant 5 octets, le dernier se trouve à l'adresse de `c` + 5, soit `$29` + 14. Pour respecter l'alignement du pointeur de pile, il faut allouer un mot en plus sur la pile pour la chaine, soit au total 16 octets.

```
# passage d'une chaine de caracteres (variable globale) representant un nombre
# entier positif au nombre entier correspondant

.data


.text
    addiu $29, $29, -16      # 4 var. locales : i, val, c et ch

    ori   $9,  $0, 0         # i <- 0
    ori   $10, $0, 0         # val <- 0
    ori   $8, $0, 0x31       # '1'
    sb    $8, 9($29)         # ch[0] <- '1'
    ori   $8, $0, 0x32       # '2'
    sb    $8, 10($29)        # ch[1] <- '2'
    ori   $8, $0, 0x33       # '3'
    sb    $8, 11($29)        # ch[2] <- '3'
    ori   $8, $0, 0x34       # '4'
    sb    $8, 12($29)        # ch[3] <- '4'
    sb    $0, 13($29)        # ch[4] <- 0

    addiu $8, $29, 9         # $8 <- @ch   # adresse de la chaine


    ori   $14, $0, 10        # $14 <- 10  # independant de la boucle, hoisting

loop:
    # ch[i] != 0
    addu  $11, $8, $9        # $9 <- @ch + i
    lbu   $11, 0($11)        # $9 <- ch[i] chargement du caractere i
    beq   $11, $0, finloop   # si vaut 0, fin de la chaine c'est fini

    # c = ch[i]
    ori   $12, $11, 0        # c <- ch[i]

    # c = c & 0x0F
    andi  $12,$12,0x0F

    # val = val * 10 + c
    multu $14, $10           # multiplication de val par 10
    mflo  $13                # $13 <- val * 10
    addu  $10, $12, $13      # val <- val * 10 + c

    # i += 1
    addiu $9, $9, 1          # $9 <- i + 1
    j     loop
finloop:
    ori   $4, $10, 0         # val a afficher doit etre mise dans $4
```

```
    ori   $2, $0, 1          # code de l'appel systeme affichage d'un entier
    syscall

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

Exercice 4 – Recopie mémoire d'une chaine de caractère

Question 1

On considère le programme C qui réalise la recopie de `N` caractères de la chaine de caractères `ch1`, globale et initialisée, dans une autre chaine de caractères `ch2`, globale mais non initialisée. Le caractère de fin de chaine est ajouté après les `N` caractères. Le programme affiche la chaine recopiée à la fin du programme.

On supposera que le nombre de caractères à copier sera toujours inférieur à la taille de la chaine `ch1`, qui sera elle toujours strictement inférieure à 20.

```
char ch2[20];
int  N = 2;           /* N <= strlen(ch1) < 20, faire varier pour vos tests */
char ch1[] = "Hello";

void main() {
   int i = 0;
   for (; i < N; i += 1) {
      ch2[i] = ch1[i];
   }
   ch2[i] = '\0';
   printf("%s", ch2);
   exit();
}
```

Donnez 2 versions du code assembleur de ce programme : une version sans optimisation et une version optimisée.
Solution:

La traduction un peu longuette et littérale

```
.data
chbis: .space 20           # espace alloue pour la recopie
N:     .word 2
ch:    .asciiz "Hello"

.text
   addiu $29, $29, -4      # var. locale i
   sw    $0, 0($29)        # i <- 0

for:
   # test i < N
   lw    $8, 0($29)        # lecture de i
   lui   $9, 0x1001        # adresse de N
   lw    $9, 20($9)        # lecture de N
   slt   $10, $8, $9       # test i < N
   beq   $10, $0, finfor

   # ch2[i] = ch1[i]
   lui   $9, 0x1001        # @ch1
   ori   $9, $9, 24
   lw    $8, 0($29)        # lecture de i
```

```
   addu  $10, $9, $8       # @ch1[i]
   lb    $10, 0($10)       # lecture ch1[i]
   lui   $9, 0x1001
   lw    $8, 0($29)        # lecture de i
   addu  $9, $8, $9        # @ch2[i]
   sb    $10, 0($9)        # recopie du caractere ch1[i] dans ch2[i]
   # i = i + 1
   lw    $8, 0($29)        # lecture de i
   addiu $8, $8, 1         # i + 1
   sw    $8, 0($29)        # ecriture de i
   # retour condition de boucle
   j     for

finfor:
   # ch2[i] = 0
   lui   $9, 0x1001
   lw    $8, 0($29)        # lecture de i
   addu  $9, $8, $9        # @ch2[i]
   sb    $0, 0($9)         # caractere de fin de chaine !

   # printf("%s", ch2)
   lui   $4, 0x1001
   ori   $2, $0, 4
   syscall                 # affichage de ch2

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

Avec optimisation des variables locales en registre (`i`), et avant la boucle for le chargement des adresses des chaines `ch1` et `ch2` ainsi que la lecture mémoire de la valeur de `N` :

```
.data
 chbis: .space 20        # espace alloue pour la recopie
 N:     .word 2
 ch:    .asciiz "Hello"

.text
   addiu $29, $29, -4    # var. locale i
   ori   $8, $0, 0       # S8 (i) <- 0

   lui   $9, 0x1001      # adresse de N
   lw    $9, 20($9)      # lecture de N

   lui   $11, 0x1001     # @ch1
   ori   $11, $11, 24
   lui   $12, 0x1001     # @ch2

for:
   # test i < N
   slt   $10, $8, $9     # test i < N
   beq   $10, $0, finfor

   # ch2[i] = ch1[i]
   addu  $10, $11, $8    # @ch1[i]
   lb    $10, 0($10)     # lecture ch1[i]
   addu  $13, $12, $8    # @ch2[i]
   sb    $10, 0($13)     # recopie du caractere ch1[i] dans ch2[i]

   # i = i + 1
```

```
   addiu $8, $8, 1        # i + 1
   j     for

finfor:
   addu  $13, $12, $8     # @ch2[i]
   sb    $0, 0($13)       # caractere de fin de chaine !

   # printf("%s", ch2)
   or    $4, $0, $12
   ori   $2, $0, 4
   syscall                # affichage de ch2

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

Question 2

On considère désormais que les deux chaines sont des variables locales du programme principal :

```
int  N = 2;       /* N <= strlen(ch1) < 20, faire varier pour les tests */

void main() {
   int i = 0;
   char ch2[20];
   char ch1[] = "Hello";

   for (; i < N; i += 1) {
      ch2[i] = ch1[i];
   }
   printf("%s", ch2);
   exit();
}
```

Que faut-il changer dans le programme précédent pour produire le code assembleur correspondant à ce programme, sans optimisation et avec optimisation ?

Solution:

Comme pour la question similaire à l'exercice précédent : il faut allouer plus de place sur la pile pour les 2 chaines, et initialiser la chaine `ch1` caractère par caractère. Il faut changer les adresses de `N`, et des chaines pour que le reste du programme fonctionne, et bien sûr adapter l'épilogue pour désallouer le bon nombre d'octets.

```
.data
 N:     .word 5


.text
   addiu $29, $29, -32      # var. locale i, ch2, ch1

   ori   $8, $0, 0x48       # 'H'
   sb    $8, 24($29)        # ch1[0] <- 'H'
   ori   $8, $0,  0x65      # 'e'
   sb    $8, 25($29)        # ch1[1] <- 'e'
   ori   $8, $0,  0x6c      # 'l'
   sb    $8, 26($29)        # ch1[2] <- 'l'
   ori   $8, $0,  0x6c      # 'l'
   sb    $8, 27($29)        # ch1[3] <- 'l'
   ori   $8, $0,  0x6f      # 'o'
   sb    $8, 28($29)        # ch1[4] <- 'o'
   sb    $0, 29($29)        # ch1[5] <- caractÃ¨re nul
```

```
   ori   $8, $0, 0          # S8 (i) <- 0

   lui   $9, 0x1001         # adresse de N
   lw    $9, 0($9)          # lecture de N

   addiu $11, $29, 24       # @ch1
   addiu $12, $29, 4        # @ch2

for:
   # test i < N
   slt   $10, $8, $9        # test i < N
   beq   $10, $0, finfor

   # ch2[i] = ch1[i]
   addu  $10, $11, $8       # @ch1[i]
   lb    $10, 0($10)        # lecture ch1[i]
   addu  $13, $12, $8       # @ch2[i]
   sb    $10, 0($13)        # recopie du caractere ch1[i] dans ch2[i]

   # i = i + 1
   addiu $8, $8, 1          # i + 1

   j     for

finfor:
   addu  $13, $12, $8       # @ch2[i]
   sb    $0, 0($13)         # caractere de fin de chaine !

   # printf("%s", ch2)
   or    $4, $0, $12
   ori   $2, $0, 4
   syscall                  # affichage de ch2

   addiu $29, $29, 32       # desallocation
   ori   $2, $0, 10
   syscall                  # exit
```