

TP 1 : Architecture des ordinateurs

Prise en main du simulateur MARS, Comprendre l'organisation et l'exécution du code d'assemblage MIPS, Se familiariser avec l'architecture externe du MIPS via un environnement virtuel.

Remarque : le simulateur MARS4.5 est à télécharger de la plateforme e-learning de l'ubma.

Exercice 1

Créer un nouveau fichier avec l'éditeur : **File → New**

Saisir le code d'assemblage suivant :

```

1 ##### Segment data #####
2 .data
3 theString:
4     .space 64          # reservation de 64 octets dans la mémoire pour la chaîne theString
5
6 ##### Segment text #####
7 .text
8 main:
9     li    $v0, 8        # chargement immédiat de la valeur 8 dans $v0
10                    # pour la lecture d'une chaîne de caractères
11     la    $a0, theString # chargement de l'adresse de la chaîne de caractères dans $a0
12     li    $a1, 64       # $a1 contient la longueur de la chaîne de caractères
13     syscall
14 ##### Fin du programme #####
15     li    $v0, 10
16     syscall
  
```

1. Faire l'assemblage du programme : **Run → Assemble** puis exécuter le programme par le bouton « **Run the current program** ».
2. Dans la fenêtre « **Run I/O** » en bas de l'écran saisir la chaîne de caractères « hello ».
3. Vérifier le contenu du registre \$v0. Traduire la valeur obtenue en décimal.
4. Vérifier le contenu du registre \$a1. Traduire la valeur obtenue en décimal.
5. Vérifier le contenu du registre \$a0. Que signifie la valeur obtenue ?
6. Dans la fenêtre « **Data segment** » lire les deux colonnes Value(+0) et Value(+4) qui correspondent à la ligne d'adresse 0x10010000.
7. Traduire ces deux valeurs en code ASCII. Quel est le résultat obtenu ?

Exercice 2

En utilisant l'éditeur de texte MARS, saisir le programme P. Le commentaire est précédé par (#). Le mot suivi par (:) est un étiquette. Le mot qui commence par un (.) est une directive de l'assembleur.

Programme P :

```

.data
    msg1: .asciiz "Enter un nombre entier: "
.text
main:
    li $v0, 4          # Appel système pour affichage d'un message
    la $a0, msg1
    syscall
# Lecture du clavier du nombre entier saisi par l'utilisateur
    li $v0, 5          # appel système pour lecture d'un entier
  
```

syscall	# L'entier lu sera placé dans le registre \$v0
# Réalisation de certains calculs avec le nombre entier	
addu \$t0, \$v0, \$0	# transfert du nombre saisi dans le registre \$t0
sll \$t0, \$t0, 2	
# Affichage du résultat	
li \$v0, 1	# Appel système pour l'affichage d'un entier
addu \$a0, \$t0, \$0	# transfert du nombre à afficher dans \$a0
syscall	
li \$v0, 10	# Appel système pour quitter l'exécution du programme
syscall	

1. Sauvegarder le fichier sous le nom « programme1.asm ».
2. En quoi servent les deux directives « .data » et « .text »
3. Chercher dans le « help » la signification de la directive « .ascii »
4. Trouver la valeur initiale du registre PC. Que signifie cette valeur ?
5. Donner le contenu du registre \$a0. Que représente cette valeur ?

Exercice 3

Supposez que vous voulez multiplier deux variables **a** et **b**, stockées dans les positions de mémoire M[adr] et M[adr+4], respectivement, pour affecter cette valeur à la variable Val, stockée à la position de mémoire M[adr+8]. C'est-à-dire, vous voulez effectuer l'opération:

Val = a*b ou: M[adr+8] = M[adr]*M[adr+4]. Avec **adr = 0x10010000**.

Le processeur possède 8 registres (R0, R1, ..., R7). Le registre R0 contient toujours la valeur zéro. Les instructions du langage machine du processeur sont:

```

LOAD R1, M[adr]           # R1 ← a
LOAD R2, M[adr+4]         # R2 ← b
ADD R3, R0, R0            # R3 ← 0
BZ R1, write              # IF a == 0, ranger le résultat dans la mémoire
etiq: BZ R2, write         # IF b == 0, ranger le résultat dans la mémoire
ADD R3, R3, R1            # R3 ← R3 + a
DEC R2                   # b ← b - 1
JMP etiq
write: STORE R3, M[adr+8]  # M[adr+8] ← R3

```

Avant d'écrire le programme, nous devons trouver un algorithme réalisant la tâche voulue.

- Représenter l'algorithme de ce programme.
- Donnez le code correspondant à cet algorithme en langage d'assemblage MIPS. Les registres (R0, R1, ..., R7) correspondent respectivement aux registres généraux du µp MIPS (\$zero, \$t1, ..., \$t7).
- Exécutez le programme avec les valeurs suivantes **a = 673** et **b = 3**.

Exercice 4

Considérons le fragment de code d'assemblage MIPS suivant:

```

..... à compléter
sw $t0, 0($a0)
sw $t1, 4($a0)
etiq1:


```

```

lw $t2, 0($a0)
lw $t3, 4($a0)
beq $t2, $t3, fin
addi $t2, $t2, 1
sw $t2, 0($a0)
j etiq1
toto:
.....compléter (Affichage)
syscall
fin :
li $v0, 10
syscall

```

Compléter le programme d'assemblage MIPS en tenant compte des données suivantes :

1. Les registres \$a0, \$t0, \$t1 contiennent respectivement l'adresse 0x10000000, les nombres entiers 30 et 35.
2. On veut afficher le contenu de \$t2
3. Exécuter le programme pas à pas, utiliser le bouton : .
4. Observer le contenu de la ligne mémoire à l'adresse 0x10000000, extraire du programme l'instruction induisant la variation de son contenu.
- 5.

Exercice 5 (Alignement Mémoire)

Soit la déclaration de variables C suivante :

```

char X[9] = { 0x10, 0x32, 0x54, 0x76, 0x98, 0xBA, 0xDC, 0xEF, 0x01 };
short Y[2] = { 0x1234, 0x5678 };
int Z = 0xABCDEFAC;
int C = { 10, 20, 30 };
char D[] = "hello world";
char E[] = "fin de l'exercice";

```

Cette déclaration doit correspondre à la zone .data du programme de votre programme:

```

.data
X : .byte 0x10, 0x32, 0x54, 0x76, 0x98, 0xBA, 0xDC, 0xEF, 0x01
Y : .half 0x1234, 0x5678
Z : .word 0xABCDEFAC
C : .word 10 20 30
D : .asciiz "hello world"
E : .asciiz "fin de l'exercice"

```

1. Avec le simulateur **Mars**, après chargement de votre programme, observer le contenu mémoire dans la zone .data (user data segment) en utilisant l'exécution pas à pas.
2. Pourquoi, les données doivent être rangées avec des adresses mémoires alignées ?
3. En observant l'implémentation mémoire de votre programme, peut-on en déduire la nature *big-endian* ou *little-endian* pour le microprocesseur MIPS 32 ?

Exercice 6

Supposons que le registre « \$a0=0x1234 » et « \$a1=0x3 ». Compléter le fragment de code en langage d'assemblage MIPS ci-dessous (le programme devra afficher le contenu de \$t0). Remplacer l'instruction « add \$t0, \$0, \$0 » par une instruction d'addition immédiate. Donner

la boucle équivalente en langage C. Quelle est l'opération arithmétique équivalente à l'instruction « srl \$t0, \$t0, 4 ».

```
add $t0,$0,$0
eti: add $t0,$t0,$a0
addi $a1,$a1,-1
bne $a1,$zero,eti
srl $t0, $t0, 4
```

Exercice 7

Soit le programme Java ci-dessous. Traduire ce programme en langage d'assemblage MIPS.

```
int a = 2;    // use $t6 to keep track of a's value
int b = 10;   // use $t7 to keep track of b's value
int m = 0;    // use $t0 to keep track of m's value
while (a > 0) {
    m += b;
    a -= 1;
}
```

Exécutez votre programme pour plusieurs valeurs initiales (nombres entiers positifs) de **a** et **b** afin de s'assurer de son exactitude.

Exercice 8

Réécrire le code suivant :

```
.data
msg1: .asciiz "Entrer un nombre entier: "
.text
.globl main

# A l'intérieur de la fonction « main », il existe certains appels (syscall) qui affectent
# le contenu du registre $31 ($ra) qui doit sauver l'adresse de retour à la fonction main.
    jal proc
    li $v0, 10
    syscall

proc: li $v0, 4 # Appel système pour affichage d'un message
    la $a0, msg1 # chargement de l'adresse de la chaîne de caractères pour affichage
    syscall

    # Maintenant demander à l'utilisateur d'introduire un nombre entier
    li $v0, 5 # Appel système pour lecture d'un entier
    syscall # L'entier introduit est rangé dans $v0

    # Réalisation de quelques opérations arithmétiques avec le nombre entier
    addu $t0, $v0, $0 # transfert du nombre dans $t0
```

```
sll $t0, $t0, 3
```

```
# Affichage du résultat
```

```
li $v0, 1 # Appel système pour affichage d'un entier
```

```
addu $a0, $t0, $0 # Transfert du nombre à afficher dans $a0
```

```
syscall
```

```
jr $ra
```

1. Quel est le rôle de l'instruction « jal » ? Expliquez.
2. Donnez le contenu du registre \$31 (\$ra). D'où vient cette valeur ?
3. Pourquoi nous avons utilisé l'instruction « jr \$ra » à la fin de la fonction « proc » ?