

Compilation — TD 1

J. Boender & G. Henry

2009–2010

1. Dans un fichier `td1-1.s`, écrivez le programme suivant :

```
.data
ala:    .ascii "Ala ma kota.\n"
.text
        .globl main
main:
        la $a0, ala
        li $v0, 4
        syscall
        li $v0, 10
        syscall
```

Lancez l'émulateur `mars`, ouvrez le fichier `td1-1.s`, puis exécutez-le

1. d'abord dans sa totalité ;
2. ensuite pas-à-pas ;
3. enfin en deux étapes en plaçant un *breakpoint* au premier appel à l'instruction `syscall`.

Modifiez le programme ci-dessus pour placer la chaîne `Ala ma kota` dans la section `.text`. Fonctionne-t-il toujours ? Quelle est la différence entre les deux sections ?

2. On considère le programme « C » suivant :

```
int main()
{
    int j;
    j = 3 * 5 + 2;
    printf("%d\n", j * j);
    exit(0);
}
```

1. Convertissez ce programme en code à trois valeurs, i.e. en code où toute instruction est soit sous la forme `x = y op z`, soit sous la forme de l'appel d'une fonction appliquée à des variables. (Il faudra probablement introduire de nouvelles variables.)

2. Affectez un registre à chacune des variables du programme obtenu ci-dessus. (On prendra soin de choisir des registres temporaires afin de ne pas avoir à en sauvegarder les valeurs.)
3. Traduisez le programme obtenu en assembleur MIPS, et exécutez-le à l'aide de **mars**.

3. On considère le programme « C » suivant :

```
int main()
{
    int i, j;
    for(i = 0; i < 10; i++) {
        j = 3 * i + 2;
        printf("%d\n", j * j);          /* (1) */
    }
    exit(0);
}
```

On appelle *branchement* une conditionnelle de la forme

```
if(e) goto l;
```

c'est à dire une conditionnelle dont la conséquence est une instruction de saut et dont l'alternative est vide.

1. Éliminez les boucles de ce programme en convertissant les **for** en **while**, puis les **while** en branchements.
2. Convertissez le programme ainsi obtenu en code à trois valeurs, puis affectez des registres temporaires à toutes les variables.
3. Traduisez le programme résultant en assembleur MIPS et exécutez-le avec **mars**.
4. Même exercice après avoir remplacé la ligne (1) par

```
if(i % 2 != 0) printf("%d\n", j * j);
```

4. Allocation de donnees

On considère le programme « C » suivant :

```
int main()
{
    int a[10];
    int i, j;
    a[0] = 1;
    a[1] = 1;
    for(i = 2; i < 10; i++)
        a[i] = a[i - 2] + a[i - 1];
    for(j = 9; j >= 0; j--)
        printf("%d\n", a[j]);
    exit(0);
}
```

4.1 Allocation statique

On va d'abord allouer le tableau `a` statiquement¹, comme ce serait le cas si on le définissait à l'extérieur de la fonction `main` ou si on utilisait le mot clef `static` en « C ».

1. Écrivez le code assembleur qui alloue 40 octets de mémoire dans la section `.data` et lui donne l'étiquette `a`. (On pourra utiliser soit la pseudo-instruction `.word` suivie de 10 zéros, soit la pseudo-instruction `.space 40`. Il faudra en outre penser à utiliser `.align`.)
2. On rappelle que la donnée `a[i]` réside à l'adresse `a + 4i`. Convertissez le programme précédent en code à trois valeurs et traduisez-le en assembleur MIPS.

4.2 Allocation sur la pile

1. Modifiez le programme précédent pour qu'il alloue 40 octets en décrémentant `sp` de 40, et qu'il accède à cette mémoire à travers `sp` au lieu de la zone statique `a`. Vous n'omettez pas de restaurer la valeur de `sp` à la fin.
2. Il est tentant d'optimiser le programme précédent en laissant `sp` inchangé et en accédant aux données en utilisant des déplacements négatifs à partir de `sp`. Pourquoi une telle approche ne peut-elle pas marcher sur un vrai processeur (indication : pensez aux interruptions asynchrones) ?

¹*Statique* caractérise tout ce qui se fait lors de l'écriture du programme ou lors de sa compilation, contrairement à *dynamique*, qui se réfère à ce qui se fait lors de l'exécution.