Université de Bordeaux
Licence STS

ARCHITECTURE DES ORDINATEURS

TP : **02**

Mémoire & Tableau

On utilise des *étiquettes* pour repérer l'emplacement des données en mémoire ; ces données peuvent être initialisées comme suit :

```
t:      .long 5
        .long 22
```

Dans cet exemple t désigne un tableau dont les deux premiers éléments sont les entiers 5 et 22, codés sur 32 bits. Ne pas confondre la *directive* `.long` avec une *instruction* exécutable par le processeur.

Pour transférer des données entre la mémoire et les registres, on utilise deux formes nouvelles de l'instruction `movl` :

```
irmovl  t, %ebx
mrmovl  (%ebx), %eax   # load
rmmovl  %eax, (%ebx)   # store
```

Décryptage : `m` = memory, `r` = register. Soit x l'entier qui se trouve à l'adresse t (dans cet exemple $x = 5$) ; la première instruction place cette adresse dans le registre `ebx` ; la seconde instruction (lecture) copie x dans le registre `eax`, et la troisième (écriture) remplace x par le contenu du registre `eax`. Le registre `ebx` est donc utilisé ici comme pointeur ; pour lire ou écrire ailleurs en mémoire, il suffit de modifier la valeur du registre, sans oublier que les adresses de deux entiers consécutifs diffèrent de 4, car un entier occupe 4 octets (on dit que chaque octet est adressable).

Pour déclarer de grands tableaux, il serait fastidieux d'utiliser une longue série de `.long`. Il est plus simple d'utiliser la directive `.pos` pour forcer l'adresse de la variable suivant le tableau. Par exemple :

```
        .pos 0x100
t:
        .pos 0x180
t2:
        .pos 0x200
```

déclare deux tableaux de 128 octets (0x80 en hexadécimal, et 0x180+0x80 = 0x200).

Exercice 1 : Décalage dans un tableau

On veut réaliser l'équivalent du code C suivant qui décale un tableau :

```
long t[4] = { 1, 2, 3, 4 };
long n = 4;

for (i = 0; i < n - 1; i ++)
   t[i] = t[i+1];
```

Notez que la boucle for est équivalente à celle ci-dessous, ce qui simplifiera l'écriture en assembleur :

```
for (p = &t[0]; n > 1; p ++, n --)
  *p = *(p+4);
```

Pour aller plus loin...

Exercice 2 : Fibonacci

La suite de Fibonacci est définie par récurrence :

$$\begin{cases} u_1 = u_2 = 1 \ , \\ u_{n+2} = u_{n+1} + u_n \ . \end{cases}$$

Les premiers termes de la suite sont donc 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, etc.

On donne ci-dessous le code C permettant de calculer les 16 premiers termes de la suite de Fibonacci. Ce code opère sur deux variables `u` et `v`, représentant respectivement les dernière et avant-dernière valeurs de la suite ayant été calculées.

```
long u = 1, v = 1, n = 16, f;
for (n = n - 2; n > 0; n --) {  /* Ou, de façon plus compacte : n -= 2 */
  register long tmp;

  tmp = u;
  u   = u + v;                  /* Ou, de façon plus compacte : u += v */
  v   = tmp;
}
f = u;
```

Question 1

Écrivez le programme y86 correspondant qui calcule les termes successifs de la suite de Fibonacci. On commençera par écrire une boucle infinie simple — équivalente à une boucle `while (1)` en langage C. Sauver ce programme dans le fichier `fibo.ys` .

Après avoir compilé et chargé ce programme dans le simulateur, procédez à une exécution pas à pas, puis à vitesse lente. L'instruction `halt` stoppe la simulation en cours.

Question 2

Modifiez le programme fibo.ys pour qu'il s'arrête après avoir calculé 16 termes de la suite.

Question 3

Modifiez encore le programme fibo.ys pour que les 16 termes calculés soient stockés en mémoire dans un tableau.