Université de Bordeaux
Licence STS

université de BORDEAUX

ARCHITECTURE DES ORDINATEURS

TP : **02**

Adressage mémoire & Tableaux

Rappels

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 à partir d'une adresse variable, on utilise le mode d'adressage *indirect par registre*. Le mode d'adressage *indexé* en est une amélioration, qui permet d'ajouter à la volée une constante à la valeur d'adresse qui sera utilisée pour accéder à la mémoire :

```
irmovl  t,%esi
mrmovl  (%esi),%eax   # Lecture, en adressage indirect
rmmovl  %eax,4(%esi)  # Écriture, en adressage indexé
```

Décryptage : Soit x l'entier qui se trouve à l'adresse d'étiquette `t`, autrement dit : `t[0]` ou `*t` (dans cet exemple, $x = 5$). La première instruction place cette adresse dans le registre `esi`. La deuxième instruction (lecture) copie $x =$ `t[0]` dans le registre `eax`. La troisième instruction (écriture) écrit x dans `t[1]`. Le registre `esi` est donc utilisé ici comme pointeur. Pour lire ou écrire ailleurs en mémoire, il suffit de modifier la valeur du registre, ou bien d'ajouter une constante dans le cas de l'adressage indexé.
N.B. : les adresses de deux entiers consécutifs diffèrent de 4, car un entier occupe `sizeof (long)` $= 4$ octets.

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
t1:
        .pos 0x180
t2:
        .pos 0x200
```

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

Exercice 1 : Suite de 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.

Question 1

Commencez par réserver une zone mémoire, commençant à une adresse d'étiquette `t`, et comprenant les deux premiers termes de la suite, de valeurs 1 et 1.

Traduisez ensuite en Y86 le code suivant, permettant de calculer en mémoire u_{16} et tous ses prédécesseurs :

```
long i, *p, n = 16;
for (p = t + 2, i = n - 2; i >= 0; p ++, i --) {
  *p = p[-1] + p[-2];
}
```

Question 2

Afin de calculer la valeur u_n sans utiliser un un tableau qui croît continûment, il faut ne garder que les valeurs dont on a besoin, en les faisant évoluer au cours du temps. Ici, pour calculer la prochaine valeur u_n, on n'a besoin que de u_{n-1} et u_{n-2}, soit une « fenêtre glissante » de deux valeurs.

On peut donc réécrire l'algorithme de calcul d'un certain terme `f` $= u_n$ de façon à s'appuyer 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, f, n = 16;
for (n = n - 2; n > 0; n --) { /* Autrement dit : n -= 2 */
  long tmp;

  tmp = u;
  u   = u + v;                 /* Autrement dit : u += v */
  v   = tmp;
}
f = u;
```

Écrivez le programme Y86 correspondant, qui calcule les termes successifs de la suite de Fibonacci.

Pour progresser par étapes, vous pouvez commencer par écrire une boucle infinie simple, équivalente à une boucle `while (1)` en langage C, afin de vérifier que votre corps de boucle calcule bien les valeurs attendues. Vous pourrez ensuite instrumenter cette première version afin de gérer la boucle et vérifier qu'elle s'arrête bien au bon moment, quel que soit n, pour calculer `f` $= u_n$.

Exercice 2 : Décalage dans un tableau

Question 1

Écrivez en Y86 l'équivalent du code C suivant, qui décale d'une case un tableau vers la gauche :

```
long n = 4;
long t[4] = { 1, 2, 3, 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; n > 1; p ++, n --)
  *p = *(p + 1);    /* (p + 1) en termes de cases, pas d'adresses ! */
```

Question 2

Écrivez une version qui décale les indices de tableau dans l'autre sens. Réfléchissez bien au sens de votre boucle, pour ne pas écraser les valeurs dans le mauvais sens !

Pour aller plus loin...

Exercice 3 : Fusion de tableaux

À la fin de l'exercice précédent, on dispose d'un tableau d'entiers *de taille variable*, codé comme suit : le premier entier du tableau contient sa taille, et les éléments suivants constituent le tableau proprement dit. Dans cet exercice, tous les tableaux sont codés ainsi.

Question 1

Écrivez un programme qui *fusionne* deux tableaux `u` et `v` (de tailles respectives `m` et `n`) constitués d'entiers croissants. Le résultat sera un nouveau tableau ordonné de taille `m + n`, placé à la suite dans la mémoire.

Pour tester ce programme, utilisez un tableau `u` constitué des 10 premiers nombres de Fibonacci (1, 1, 2, 3, 5, 8, 13, 21, 34, 55) et un tableau `v` constitué des 7 premières puissances de 2 (1, 2, 4, 8, 16, 32, 64). Le résultat de la fusion doit être le tableau suivant, constitué de 17 nombres : 1, 1, 1, 2, 2, 3, 4, 5, 8, 8, 13, 16, 21, 32, 34, 55, 64.

Conseil : développez ce programme par étapes :

- Commencez par calculer la taille du nouveau tableau, et sauvegardez-la au bon endroit ;
- Écrivez une boucle *sans fin* qui effectue le travail principal (fusion) ;
- Ajoutez enfin les instructions qui gèrent le cas où l'un des deux tableaux `u` ou `v` a été entièrement parcouru.

Testez soigneusement chaque étape avec le simulateur avant de passer à l'étape suivante !