

ARCHITECTURE DES ORDINATEURS

TP : 03

## PILE ET APPEL DE FONCTION

---

### Exercice 1 : Fonction simple

Écrivez le code assembleur Y86 correspondant au programme C suivant. Faites tourner dans le simulateur et observez bien l'évolution de la pile, les adresses utilisées, etc.

```
long sub (long i, long j)
{
    return (i - j);
}

long res;

void main ()
{
    res = sub (5, 7);
    halt();
}
```

### Exercice 2 : Mise en œuvre d'une variable locale

Maintenant, on a besoin de stocker le résultat calculé dans une variable locale car entre le calcul et le `return` on effectue d'autres choses :

```
long sub (long i, long j)
{
    long x = i - j;
    ...
    return (x);
}

long res;

void main ()
{
    res = sub (5, 7);
    halt();
}
```

### Exercice 3 : Appeler plusieurs fonctions

```
long g (long i)
```

```

{
    return (sub (i, 1) + sub (i, 2));
}

long res;

void main()
{
    res = g (2);
    halt();
}

```

## Exercice 4 : Appel récursif

Faites vraiment l'appel à f, sans optimiser pour l'instant.

```

long f (long n)
{
    if (n == 0)
        return 0;
    else
        return (f (n - 1) + n);
}

long res;
void main ()
{
    res = f (10);
    halt();
}

```

## Exercice 5 : Appel récursif terminal

On ajoute ici un accumulateur :

```

long g (long n, long acc)
{
    if (n == 0)
        return acc;
    else
        return (g (n - 1, acc + n));
}

long f (long n)
{
    return g (n, 0);
}

long res;
void main ()
{
    res = f (10);
    halt();
}

```

Convainquez-vous que le calcul est le même. `g` utilise un appel récursif *terminal* : après l'appel récursif elle n'a plus rien à faire d'autre que de retourner simplement le même résultat

Dans un premier temps, n'optimisez pas l'appel récursif dans `g`. Empilez comme d'habitude les paramètres. Ensuite, remplacez les empilements par des `rmmovl`, le `call g` par un `jmp g2`, où `g2` pointe au début de la fonction, et enlevez les `popl`. Vous obtenez une fonction qui se comporte comme une fonction récursive, mais elle n'utilise plus la pile.

Compilez le code C ci-dessus en assembleur à l'aide de `gcc` :

```
gcc -m32 test.c -o test.s -S
```

Lisez la sortie `test.s` (note : pour le rendre plus lisible, supprimez les directives du genre `.cfi_foo`; `leave` est un raccourci pour `movl %ebp,%esp; pop %ebp`; `leal (a,b),c` calcule simplement `c = a + b`). Essayez d'ajouter `-O2`, constatez que `gcc` a su exploiter la récursivité terminale.

## Exercice 6 : Nombre d'arguments variables

Écrivez une fonction `max` qui calcule le maximum de ses paramètres. La fin des paramètres sera indiqué par un paramètre 0.

## Exercice 7 : Passage d'argument par valeur ou par référence

```
void f (long x, long y, long *sum, long *diff) {
    *sum = x + y;
    *diff = x - y;
}

long a, b;
void main (void)
{
    f (7, 10, &a, &b);
    halt();
}
```

Ici, 7 et 10 sont passés par valeur, et `a` et `b` sont passés par référence. C'est une pratique courante en C pour qu'une fonction puisse retourner plusieurs résultats : elle écrit dans des variables dont on passe l'adresse.

## Pour aller plus loin...

## Exercice 8 : Appel de fonction, pointeurs, tableaux

```
long g (long n)
{
    return (n & 1);
}

void f (long *t)
{
    long *p;
    for (p = t; *p != 0; p++)
        *p = g (*p);
}
```