Université de Bordeaux
Licence STS

ARCHITECTURE DES ORDINATEURS

TP : **03**

Pile et appel de fonction

Exercice 1 : Calcul en notation polonaise inverse

La notation polonaise inverse utilise un principe d'écriture postfixe des opérations :
— l'expression $5 + 10$ s'écrit `5 10 +`
— l'expression $2 - (10 + 4)$ s'écrit `2 10 4 + -`
— l'expression $(10 + 4) - 2$ s'écrit `10 4 + 2 -`
— etc.

L'idée est d'utiliser une pile pour stocker les opérandes, et d'implémenter les opérations de façon à utiliser les valeurs en sommet de pile (c.-à-d. le sommet et le sous-sommet pour une opération binaire).

Par exemple, pour calculer $(10 + 4) - 2$, noté `10 4 + 2 -`, on effectue les actions suivantes :
— `10` : on empile `10`
— `4` : on empile `4`
— `+` : on dépile deux valeurs, que l'on additionne, et on remet le résultat dans la pile.
— `2` : on empile `2`
— `-` : on dépile deux valeurs, que l'on soustrait, et on remet le résultat dans la pile.

Soit l'expression suivante, utilisant quatre variables globales `a`, `b`, `c` et `d` :

$$(a - b) + ((b - c)\&(c - d)) \ .$$

Question 1

Écrivez cette expression en notation polonaise inverse.

Question 2

Écrivez un programme utilisant la pile pour la calculer, en suivant donc la notation polonaise et en n'utilisant que deux registres de travail (en plus du pointeur de pile).

Lors de chaque opération de calcul, dépilez systématiquement les deux opérandes avant de faire le calcul, puis empilez le résultat.

Dans un second temps, optimisez les empilements/dépilements vraiment triviaux.

On arrive ainsi à produire sans réfléchir un code qui fait tout le calcul en utilisant seulement deux registres de travail.

Exercice 2 : Fonction simple

Écrivez le code assembleur Y86 correspondant au programme C suivant. Faites-le 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 3 : Mise en œuvre d'une variable locale

Supposons maintenant que, dans la fonction `sub`, on ait 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);
}
```

Modifiez le code de `sub` du programme précédent, afin de stocker le résultat de la soustraction dans une variable temporaire.

Exercice 4 : Utilisation de registres *callee save*

Modifiez le programme précédent pour utiliser, sans danger, le registre `ebx` au sein de la fonction `sub`.

Exercice 5 : Utilisation de registres *caller save*

Supposez que, dans la fonction principale, `eax` doive garder sa valeur pendant l'appel de la fonction. Modifiez le programme principal en conséquence.

Exercice 6 : Appeler plusieurs fonctions

Écrivez le code assembleur Y86 correspondant au programme C suivant.

```
long g (long i)
{
  return (sub (i, 7) + sub (i, 1));
}

long res;

void main ()
{
  res = g (3);
  halt ();
}
```

Exercice 7 : Appel récursif

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

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

long res;

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

Pour aller plus loin...

Exercice 8 : Appel de fonction, pointeurs, tableaux

Écrivez le code assembleur Y86 correspondant au programme C suivant.

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

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

Exercice 9 : 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 10 : Passage d'argument par valeur ou par référence

Écrivez le code assembleur Y86 correspondant au programme C suivant.

```
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 a passé les références.