Pile et appels de fonctions (procédures)

Exercice 1 On part du programme suivant *fibo1.ys* (corrigé d'un exercice de la première série) :

```
        irmovl  1, %eax
        irmovl  1, %ebx
        mrmovl  n, %esi         # compteur
        irmovl  n, %edi         # pointeur
boucle: rrmovl  %eax, %ecx
        addl    %ebx, %eax
        rrmovl  %ecx, %ebx
        iaddl   4, %edi
        rmmovl  %eax, (%edi)
        isubl   1, %esi
        jne boucle
        halt
.align 4
n:      .long   16
```

On souhaite transformer ce programme en une procédure avec deux arguments t et n : t est l'adresse du tableau qui contiendra n termes de la suite de Fibonacci. Voici le code du *programme principal* qui place les arguments $t = 0x100$ et $n = 16$ sur la pile, appelle la procédure, nettoie la pile et s'arrête :

```
  0x000: 308400020000 |   irmovl  0x200, %esp
  0x006: 2045         |   rrmovl  %esp, %ebp
  0x008: 308010000000 |   irmovl  16, %eax
  0x00e: a008         |   pushl   %eax  # arg 2
  0x010: 308000010000 |   irmovl  0x100, %eax
  0x016: a008         |   pushl   %eax  # arg 1
  0x018: 8020000000   |   call    fibo
  0x01d: 2054         |   rrmovl  %ebp, %esp
  0x01f: 10           |   halt
```

Adresse	Mot
200	

1. Compléter le tableau ci-dessus pour indiquer le contenu de la pile après exécution de `call fibo`. Quelle est alors la valeur du registre `%esp` ?
2. Identifier les quelques modifications à apporter au programme *fibo1.ys* pour qu'il devienne le code de la procédure `fibo`.
3. TP : mettre bout à bout le code ci-dessus (programme principal) et le code obtenu en réponse à la question précédente dans le fichier *fibo.ys*, compiler — `make fibo.yo`, et exécuter le programme avec le simulateur.

   Pendant une exécution pas à pas examiner avec soin le contenu du registre `%esp` et le contenu de la mémoire : repérer en particulier les arguments et l'adresse de retour sur la pile.

Exercice 2 Le but de l'exercice est d'écrire une procédure `max` qui calcule (dans le registre `%eax`) le maximum de ses arguments, *en nombre variable*.

1. Le nombre d'arguments sera placé sur la pile : avant ou après les arguments "ordinaires" ?
2. Ecrire un exemple de programme appelant qui calcule le plus grand des trois entiers 16, 0x12 et 12.
3. Ecrire la procédure `max`.
4. En TP observer soigneusement le contenu de la pile, et les adresses contenues dans les registres `%esp` et `%ebp`. Observer aussi le résultat du calcul de $a - b$ lorsque $a < b$.

Exercice 3 La notation polonaise (inverse) utilise un principe d'écriture postfixe des opérations, qui permet d'éviter l'usage de parenthèses : $a + b * c$ s'écrit "$a\ b\ c\ *+$" tandis que $(a + b) * c$ s'écrit "$a\ b + c\ *$".

Une expression polonaise peut être évaluée en utilisant une pile comme suit :

- un nombre est empilé ;
- une opération porte toujours sur les deux nombres en haut de la pile, qui sont remplacés par le résultat de l'opération.

Exemples (la pile grandit vers le bas) :

Expression	$a\ b\ c\ *+$				
Action	empiler	empiler	empiler	calculer	calculer
Pile	a	a b	a b c	a $b * c$	$a + b * c$

Expression	$a\ b + c\ *$				
Action	empiler	empiler	calculer	empiler	calculer
Pile	a	a b	$a + b$	$a + b$ c	$(a + b) * c$

Le jeu d'instructions Y86 ne comporte pas d'opérateur de multiplication (à votre avis pourquoi ?), les seules opérations disponibles sont `addl` (addition), `subl` (soustraction), `andl` (conjonction bit à bit), `xorl` (disjonction exclusive bit à bit).

1. Ecrire l'expression $a + ((b - c)\&d)$ en notation polonaise.
2. Ecrire un programme qui calcule cette expression en utilisant la pile selon la méthode décrite ci-dessus ; a, b, c, d sont des adresses d'entiers en mémoire. *Note* : le programme comporte quelques `pushl` immédiatement suivis de `popl`, ne pas essayer de le simplifier, le but de l'exercice est de montrer comment calculer une expression aussi complexe soit-elle à l'aide d'un algorithme simple qui n'utilise que deux registres.
3. Si a, b, c, d sont donnés par :

```
a:      .long 42
b:      .long 15
c:      .long 8
d:      .long 0x3a7b
```

   quel est le résultat du calcul ? Donner la réponse en numération décimale et en numération hexadécimale. Utiliser le simulateur en TP pour suivre l'évolution des calculs et vérifier les résultats.

Exercice 4

1. Ecrire une fonction $f(x)$ qui récupère son argument x sur la pile (comme d'habitude) et calcule $6x$; le résultat sera placé dans le registre `%eax` (comme d'habitude).
2. Ecrire sur le même modèle une fonction $g(x, y)$ qui calcule $4x + 11y$. Utiliser l'instruction de décalage vers la gauche `isall` pour minimiser le nombre d'instructions et le nombre de registres employés.
3. Ecrire un programme `main` qui calcule $f(10) + g(7, 10)$. Calculer la représentation hexadécimale du résultat et vérifier avec le simulateur en TP.
4. Ajouter le calcul de $g(7, f(10))$.
5. (question optionnelle) On souhaite maintenant que le programme appelant trouve les valeurs de $f(x)$ et $g(x, y)$ sur la pile, à la place des arguments (voir exercice 3). Modifier f, g et `main` en conséquence. *Attention* : il y a un piège en ce qui concerne g — lequel ?

Exercice 5 Le résultat de la multiplication de deux registres ne peut pas être stocké (en général) dans le second registre : expliquer pourquoi.

L'instruction `mul` du jeu d'instructions x86 a donc une syntaxe particulière — voir un manuel Intel sur Internet, requête Google : *Intel x86 instruction set manual* — et ne figure pas dans le jeu d'instruction simplifié y86. L'objet de cet exercice est d'effectuer une multiplication à l'aide d'une fonction y86.

L'algorithme employé est l'algorithme ordinaire appris à l'école, adapté à la numération binaire, où les tables de multiplication sont beaucoup plus simples ! On effectue aussi les additions intermédiaires dès que possible, ce qui donne l'algorithme suivant :

```
# calcule le produit m*n dans p
p = 0
tant que n > 0:
    si n est impair:
        p = p + m
    m = 2 * m
    n = n / 2   # division entière
```

Il est facile de vérifier que $p + mn$ est un invariant de la boucle ; au début du programme il vaut mn (puisque $p = 0$), et à la fin il vaut p (puisque $n = 0$).

1. Ecrire en assembleur y86 une fonction `mult` qui réalise cet algorithme ; le résultat sera placé dans le registre `%eax`.
2. Tester en calculant 5*6, puis 6*5.
3. Observer (avec le simulateur) ce qui se passe si on tente de multiplier $0x12345678$ par 16 : quand le code de condition O (Overflow) prend-il la valeur 1 ?
4. Utiliser la fonction `mult` pour écrire un programme *factorielle.ys* qui calcule les valeurs successives de $n!$; développer ce programme progressivement :
   (a) Commencer par une simple boucle infinie, et vérifier avec le simulateur que les premières valeurs calculées (dans le registre `%eax`) sont correctes — pour cela calculer les représentations hexadécimales de 4 !, 5 ! et 6 !
   (b) Utiliser le simulateur pour observer quelle est la première valeur de n pour laquelle le résultat calculé est incorrect (débordement).
   (c) Compléter le programme pour qu'il range en mémoire les valeurs successives de $n!$ tant que c'est possible.
   (d) La multiplication est commutative, donc l'ordre des arguments de la fonction `mult` est normalement sans importance. Ce n'est pas tout à fait vrai, pourquoi ? Par exemple une fois que le programme *factorielle.ys* a calculé $u = 8!$, il appelle la fonction `mult` avec les arguments u et 9 pour calculer 9 ! : dans quel ordre est-il préférable d'empiler ces deux arguments ?

Exercice 6 (très optionnel) Voici un programme un peu curieux (mais court) qui permet de calculer n'importe quelle expression écrite en notation polonaise (voir exercice 3) :

```
0x000: 308400020000 |          irmovl  0x200,%esp
0x006: 308600010000 |          irmovl  expr,%esi
0x00c: 500600000000 | boucle:  mrmovl  (%esi),%eax
0x012: 503604000000 |          mrmovl  4(%esi),%ebx
0x018: 308208000000 |          irmovl  8,%edx
0x01e: 6026         |          addl    %edx,%esi
0x020: 6200         |          andl    %eax,%eax
0x022: 742e000000   |          jne     calcul
0x027: a038         |          pushl   %ebx
0x029: 700c000000   |          jmp     boucle
0x02e: 6232         | calcul:  andl    %ebx,%edx
0x030: 7453000000   |          jne     stop
0x035: b018         |          popl    %ecx
0x037: b008         |          popl    %eax
0x039: 308260100000 |          irmovl  0x1060,%edx     # addl %ecx,%eax
0x03f: 6032         |          addl    %ebx,%edx
0x041: 402848000000 |          rmmovl  %edx,op
0x048:              | .align   4
0x048: 00000000     | op:      .long   0
0x04c: a008         |          pushl   %eax
0x04e: 700c000000   |          jmp     boucle
0x053: 10           | stop:    halt
0x100:              | .pos     0x100
0x100: 00000000     | expr:     .long   0
0x104: 2a000000     |          .long   42       # a
0x108: 00000000     |          .long   0
0x10c: 0f000000     |          .long   15       # b
0x110: 00000000     |          .long   0
0x114: 08000000     |          .long   8        # c
0x118: 01000000     |          .long   1
0x11c: 01000000     |          .long   1        # sub
0x120: 00000000     |          .long   0
0x124: 7b3a0000     |          .long   0x3a7b   # d
0x128: 01000000     |          .long   1
0x12c: 02000000     |          .long   2        # and
0x130: 01000000     |          .long   1
0x134: 00000000     |          .long   0        # add
0x138: 01000000     |          .long   1
0x13c: 08000000     |          .long   8        # signal de fin
```

L'expression qui termine le programme correspond à l'exemple étudié dans l'exercice 3.

1. Comment l'expression est-elle codée ?
2. Analyser ce programme en expliquant le rôle de chaque instruction (ou de chaque paire d'instructions qui réalise une action élémentaire).
3. Les instructions d'adresses 0x39 à 0x4b sont a priori incompréhensibles ; essayer pourtant de les décrypter, en utilisant le simulateur et en sachant que l'instruction codée `00` (un seul octet) est l'instruction `nop` (ne rien faire).

Exercice 2, corrigé

Ce programme calcule le plus grand des trois entiers 16, 0x12 et 12.

```
0x000: 308400020000 | main:   irmovl  0x200,%esp
0x006: 2045         |         rrmovl  %esp,%ebp
0x008: 30800c000000 |         irmovl  12,%eax
0x00e: a008         |         pushl   %eax
0x010: 308012000000 |         irmovl  0x12,%eax
0x016: a008         |         pushl   %eax
0x018: 308010000000 |         irmovl  16,%eax
0x01e: a008         |         pushl   %eax
0x020: 308003000000 |         irmovl  3,%eax
0x026: a008         |         pushl   %eax
0x028: 8030000000   |         call    max
0x02d: 2054         |         rrmovl  %ebp,%esp       # tout depiler
0x02f: 10           |         halt
                    |
0x030: a058         | max:    pushl   %ebp
0x032: 2045         |         rrmovl  %esp, %ebp
0x034: 502508000000 |         mrmovl  8(%ebp),%edx    # compteur
0x03a: a068         |         pushl   %esi
0x03c: 2056         |         rrmovl  %ebp,%esi       # pointeur
0x03e: c0860c000000 |         iaddl   12,%esi
0x044: 500600000000 |         mrmovl  (%esi),%eax     # m = max temporaire
0x04a: c18201000000 | boucle: isubl   1, %edx
0x050: 7373000000   |         je      fin
0x055: c08604000000 |         iaddl   4,%esi
0x05b: 501600000000 |         mrmovl  (%esi),%ecx     # argument a
0x061: a018         |         pushl   %ecx
0x063: 6101         |         subl    %eax,%ecx       # a - m
0x065: b018         |         popl    %ecx
0x067: 714a000000   |         jle     boucle          # a <= m
0x06c: 2010         |         rrmovl  %ecx,%eax
0x06e: 704a000000   |         jmp     boucle
0x073: b068         | fin:    popl    %esi
0x075: b058         |         popl    %ebp
0x077: 90           |         ret
```

Exercice 4, corrigé

Ce programme calcule $f(10) + g(7, 10)$, avec $f(x) = 6x$ et $g(x, y) = 4x + 11y$:

```
  0x000: 308400020000 | main:   irmovl  0x200,%esp
  0x006: 2045         |         rrmovl  %esp,%ebp
  0x008: 30800a000000 |         irmovl  10,%eax
  0x00e: a008         |         pushl   %eax
  0x010: 8033000000   |         call    f
  0x015: 2003         |         rrmovl  %eax,%ebx   # registre "safe"
  0x017: b008         |         popl    %eax        # depiler argument
  0x019: 30800a000000 |         irmovl  10,%eax
  0x01f: a008         |         pushl   %eax        # empiler argument 2
  0x021: 308007000000 |         irmovl  7,%eax
  0x027: a008         |         pushl   %eax        # empiler argument 1
  0x029: 8048000000   |         call    g
  0x02e: 6030         |         addl    %ebx,%eax
  0x030: 2054         |         rrmovl  %ebp,%esp   # depiler arguments
  0x032: 10           |         halt
                      |
  0x033: a058         | f:      pushl   %ebp
  0x035: 2045         |         rrmovl  %esp,%ebp
  0x037: 500508000000 |         mrmovl  8(%ebp),%eax    # x
  0x03d: 6000         |         addl    %eax,%eax       # 2x
  0x03f: 2001         |         rrmovl  %eax,%ecx
  0x041: 6000         |         addl    %eax,%eax       # 4x
  0x043: 6010         |         addl    %ecx,%eax       # 6x
  0x045: b058         |         popl    %ebp
  0x047: 90           |         ret
                      |
  0x048: a058         | g:      pushl   %ebp
  0x04a: 2045         |         rrmovl  %esp,%ebp
  0x04c: 500508000000 |         mrmovl  8(%ebp),%eax    # x
  0x052: c48002000000 |         isall   2,%eax          # 4x
  0x058: 50150c000000 |         mrmovl  12(%ebp),%ecx   # y
  0x05e: 6010         |         addl    %ecx,%eax       # 4x + y
  0x060: 6011         |         addl    %ecx,%ecx       # 2y
  0x062: 6010         |         addl    %ecx,%eax       # 4x + 3y
  0x064: c48102000000 |         isall   2,%ecx          # 8y
  0x06a: 6010         |         addl    %ecx,%eax       # 4x + 11y
  0x06c: b058         |         popl    %ebp
  0x06e: 90           |         ret
```

Les trois instructions d'adresses 0x017, 0x019 et 0x01f sont inutiles si f ne modifie pas son argument ; c'est le cas ici, et c'est très souvent le cas, mais ce n'est pas garanti. Pour calculer $g(7, f(10))$, il faut remplacer le début du programme par :

```
main:   irmovl  0x200,%esp
        rrmovl  %esp,%ebp
        irmovl  10,%eax
        pushl   %eax        # empiler argument
        call    f
        iaddl   4,%esp      # depiler argument
        pushl   %eax        # empiler f(10)
        irmovl  7,%eax
        pushl   %eax        # empiler argument 1
        call    g
        rrmovl  %ebp,%esp   # depiler arguments
        halt
```

Exercice 5, corrigé

Ce programme calcule $n!$ jusqu'à $n = 12$ (au delà $n!$ n'est pas représentable sur 32 bits) :

```
0x000: 308400020000 |          irmovl  0x200, %esp
0x006: 308001000000 |          irmovl  1, %eax     # 0!
0x00c: 6333         |          xorl    %ebx, %ebx  # n = 0
                    |
0x00e: c08301000000 | fact:    iaddl   1, %ebx     # n++
0x014: a038         |          pushl   %ebx
0x016: a008         |          pushl   %eax        # resultat precedent
0x018: 8043000000   |          call    mult
0x01d: c08408000000 |          iaddl   8, %esp     # depiler arguments
0x023: c48302000000 |          isall   2, %ebx     # 4n
0x029: 400300010000 |          rmmovl  %eax, t(%ebx)
0x02f: c58302000000 |          isarl   2, %ebx     # restaurer n
0x035: 2031         |          rrmovl  %ebx, %ecx
0x037: c1810c000000 |          isubl   12, %ecx
0x03d: 740e000000   |          jne     fact
0x042: 10           |          halt
                    |
0x043: a058         | mult:    pushl   %ebp
0x045: 2045         |          rrmovl  %esp, %ebp
0x047: 501508000000 |          mrmovl  8(%ebp), %ecx   # m
0x04d: 50250c000000 |          mrmovl  12(%ebp), %edx  # n
0x053: 6300         |          xorl    %eax, %eax      # p = 0
                    |          # sauver %ebx avant utilisation
0x055: a038         |          pushl   %ebx
0x057: 2023         | mloop:   rrmovl  %edx, %ebx
0x059: c28301000000 |          iandl   1, %ebx     # n pair ?
0x05f: 7366000000   |          je pair
0x064: 6010         |          addl    %ecx, %eax  # p = p+m
0x066: 6011         | pair:    addl    %ecx, %ecx  # m = 2*m
0x068: c58201000000 |          isarl   1, %edx     # n = n/2
0x06e: 7457000000   |          jne     mloop
0x073: b038         |          popl    %ebx
0x075: b058         |          popl    %ebp
0x077: 90           |          ret
                    |
0x100:              | .pos     0x100
0x100: 01000000     | t:       .long   1   # 0!
0x104: 01000000     |          .long   1   # 1!
```