Architecture des ordinateurs et systèmes d'exploitation

Corrigé du projet assembleur SPARC

Arnaud Giersch Benoît Meister Nicolas Passat

1 Triangle de Pascal

1.1 Formules de calcul de C_n^p

Le nombre C_n^p (généralement utilisé en dénombrement) peut être défini de deux manières distinctes (ces deux manières aboutissant à des valeurs identiques, bien évidemment !) :

$$C_n^p = \begin{cases} \frac{n\cdot(n-1)\cdot\cdots\cdot(p+2)\cdot(p+1)}{(n-p)!} & \text{si } n > p \\ 1 & \text{si } n = p \end{cases}$$

ou bien :

$$C_n^p = \begin{cases} C_{n-1}^p + C_{n-1}^{p-1} & \text{si } 0 < p < n \\ 1 & \text{si } p = n \\ 1 & \text{si } p = 0 \end{cases}$$

On considérera par ailleurs que C_n^p est nul pour $n < 0$, $p < 0$ ou $n < p$

1.2 Forme du triangle de Pascal

Le triangle de Pascal est composé des valeurs successives de C_n^p, où n et p prennent toutes les valeurs dans les entiers naturels. Ce triangle a la forme suivante :

$$\begin{array}{ccccc} C_0^0 & & & & \\ C_1^0 & C_1^1 & & & \\ C_2^0 & C_2^1 & C_2^2 & & \\ C_3^0 & C_3^1 & C_3^2 & C_3^3 & \\ C_4^0 & C_4^1 & C_4^2 & C_4^3 & C_4^4 \\ \multicolumn{5}{c}{\cdots\cdots\cdots\cdots\cdots\cdots} \end{array}$$

Ainsi, chaque ligne représente une valeur fixée de n et chaque colonne, une valeur fixée de p.

2 Calcul du triangle de Pascal

2.1 Version itérative

En utilisant la première formule présentée, proposez un programme en assembleur Sparc qui affiche les 13 premières lignes du triangle de Pascal. Votre programme devra *impérativement* définir et utiliser les fonctions suivantes :

– une fonction de calcul de factorielle, récursive ou bien itérative ;

Correction :

```
        ! Fonction de calcul de factorielle
        !       version récursive:       factor_rec
        !       version itérative:       factor_iter
        !       version indéfinie:       factor

.section ".text"
        .align 4

        .global factor_rec
        .global factor_iter
        .global factor


factor_rec:                     ! factor_rec (n): calcul de n! de manière
                                !                 récursive
                                !       { 0             si n = 0
                                ! n! := {
                                !       { n * (n-1)!    sinon
        save %sp, -64, %sp
        cmp %i0, %g0            ! %i0 est le paramètre
        bne fr_ret1             ! if (%i0 != 0) {
        nop
        sub %i0, 1, %o0
        call factor_rec         !       %o0 <- (%i0 - 1)!
        nop
        umul %i0, %o0, %i0      !       return %i0 * %o0
        ret
        restore                 ! } else {
```

```
fr_ret1:
        mov 1, %i0              !       return 1
        ret
        restore                 ! }


factor:
factor_iter:                    ! factor_iter (n): calcul de n! de manière
                                !                  itérative
                                ! 0! := 1
                                ! n! := n * (n-1) * ... * 2 * 1
        save %sp, -64, %sp
        mov 1, %l0              ! %i0 est le paramètre
                                ! %l0 est le résultat
fi_loop:
        cmp %i0, 1              ! while (%i0 > 1) {
        ble fi_end
        nop
        umul %l0, %i0, %l0      !       %l0 <- %i0 * %l0
        sub %i0, 1, %i0         !       %i0 --
        ba fi_loop              ! }
        nop
fi_end:
        mov %l0, %i0            ! return %l0
        ret
        restore
```

– une fonction de calcul de produit partiel ($f(a,b) = a \cdot (a+1) \cdot \cdots \cdot (b-1) \cdot b$), récursive ou bien itérative ;

Correction :

```
        ! Fonctions de calcul de produit partiel:
        !       version recursive:       ppartiel_rec
        !       version iterative:       ppartiel_iter
        !       version indefinie:       ppartiel

        /* On appelle produit partiel de a à b le produit
         *      a * (a+1) * ... * (b-1) * b     si a <= b
         *
         * NB: si a > b, on retournera a.
         */

.section ".text"
        .align 4

        .global ppartiel_rec
        .global ppartiel_iter
        .global ppartiel


ppartiel_rec:                   ! ppartiel_rec (a, b): calcul du produit
                                !                      partiel de a à b de
                                !                      manière récursive
                                ! ppartiel_rec (a, b) :=
                                !       { a             si a >= b
                                !       { a * ppartiel_rec (a+1, b)   sinon

        save %sp, -64, %sp
        cmp %i0, %i1            ! les paramètres sont %i0 et %i1
        bge ppr_ret             ! if (%i0 < %i1) {
        nop
        add %i0, 1, %o0
        mov %i1, %o1
        call ppartiel_rec       !       %o0 <- ppartiel_rec (%i0+1, %i1)
        nop
        umul %i0, %o0, %i0      !       %i0 <- %i0 * %o0
                                ! }
ppr_ret:
        ret                     ! return %i0
        restore


ppartiel:
ppartiel_iter:                  ! ppartiel_iter (a, b): calcul du produit
                                !                       partiel de a à b de
                                !                       manière itérative
        save %sp, -64, %sp
                                ! les paramètres sont %i0 et %i1
                                ! le résultat dans %i0
        mov %i0, %l0            ! %l0 <- %i0  (compteur)
ppi_loop:
        cmp %l0, %i1            ! while (%l0 < %i1) {
        bge ppi_end
        nop
```

```
        add %l0, 1, %l0         !       %l0 ++
        umul %i0, %l0, %i0      !       %i0 <- %i0 * %l0
        ba ppi_loop             ! }
        nop
ppi_end:
        ret                     ! return %i0
        restore

        /*
         * NB:  on aurait pu definir factorielle en utilisant le
         *      produit partiel définit plus haut
         */
        ! Fonction de calcul de factorielle
        !       version recursive:       factor_rec
        !       version iterative:       factor_iter
        !       version indefinie:       factor


/*-------------------------

        .global factor_rec
        .global factor_iter
        .global factor


factor_rec:                     ! factor_rec (n): calcul de n! de manière
                                !                 récursive
                                ! n! := ppartiel_rec (1, n)
        save %sp, -64, %sp
        mov 1, %o0
        mov %i0, %o1
        call ppartiel_rec
        nop
        mov %o0, %i0
        ret
        restore


factor:
factor_iter:                    ! factor_iter (n): calcul de n! de manière
                                !                  itérative
                                ! n! := ppartiel_iter (1, n)
        save %sp, -64, %sp
        mov 1, %o0
        mov %i0, %o1
        call ppartiel_iter
        nop
        mov %o0, %i0
        ret
        restore

-------------------------*/
```

– une fonction de calcul de C_n^p utilisant les deux fonctions précédentes ;

Correction :

```
.section ".text"
        .align 4

        .global c_n_p
```

```
c_n_p:                          ! c_n_p (n, p):  calcul de C_n^p de manière
                                !                itérative
                                !          { 1          si n = p
                                ! C_n^p := {
                                !          { ppartiel (p+1, n) / (n-p)!
                                !          {            sinon (n > p)
```

```
        save %sp, -64, %sp
                                ! les paramètres sont %i0 (n) et %i1 (p)
        cmp %i0, %i1            ! if (%i0 <= %i1) {
        bg cnp_great
        nop
        mov 1, %i0              !       return 1
        ret
        restore
cnp_great:                      ! } else {
        add %i1, 1, %o0
        mov %i0, %o1
        call ppartiel           !       %o0 <- ppartiel (%i1+1, %i0)
        nop
        mov %o0, %l0            !       %l0 <- %o0
        sub %i0, %i1, %o0
        call factor             !       %o0 <- (%i0 - %i1)!
        nop
        mov %g0, %y             !       le premier opérande de udiv est
                                !       sur 64 bits
        udiv %l0, %o0, %i0      !       %i0 <- %l0 / %o0
        ret                     !       return %i0
        restore                 ! }
```

– une fonction principale (`main`) qui affiche les 13 premières lignes du triangle.

Correction :

```
.section ".rodata"
        .align 1

.PF_UNSIGNED:
        .asciz " %3u"
.PF_NEWLINE:
        .asciz "\n"


.section ".text"
        .align 4

        .global main


main:                           ! fonction principale: main
        save %sp, -96, %sp

        set 13, %l3             ! nombre de lignes à afficher

        mov 0, %l0
loop_n:                         ! for (%l0 = 0; %l0 < %l3; %l0++) {
        cmp %l0, %l3
        bge endloop_n
        nop

        mov 0, %l1
loop_p:                         !       for (%l1 = 0; %l1 <= %l0; %l1++) {
        cmp %l1, %l0
        bg endloop_p
        nop

        mov %l0, %o0
        mov %l1, %o1
        call c_n_p              !               %o0 <- C_%l0^%l1
        nop

        mov %o0, %o1            !               %o1 <- %o0
        set .PF_UNSIGNED, %o0
        call printf             !               printf (" %3u", %o1)
        nop

        add %l1, 1, %l1         !               incrément du for(%l1...)
        ba loop_p
        nop
endloop_p:                      !       }

        set .PF_NEWLINE, %o0
        call printf             !       printf ("\n")
        nop

        add %l0, 1, %l0         !       incrément du for(%l0...)
        ba loop_n
        nop
endloop_n:                      ! }
        clr %i0
        ret                     ! return 0
        restore
```

2.2 Version récursive

En utilisant la seconde formule présentée, proposez un programme en assembleur Sparc qui affiche les 13 premières lignes du triangle de Pascal. Votre programme devra *impérativement* définir et utiliser les fonctions suivantes :

– une fonction de calcul de C_n^p *récursive* ;

Correction :

```
.section ".text"
        .align 4

        .global c_n_p


c_n_p:                          ! c_n_p (n, p):  calcul de C_n^p de manière
                                !                récursive
                                !           { 1  si p = 0 ou p = n
                                ! C_n^P := {
                                !           { C_(n-1)^p + C_(n-1)^(p-1)  sinon
        save %sp, -64, %sp
                                ! les paramètres sont %i0 (n) et %i1 (p)
        cmp %i0, %i1
        be cnp_ret1             ! if (%i0 != %i1
        nop
        cmp %i1, %g0            !     && %i1 != 0) {
        be cnp_ret1
        nop
        sub %i0, 1, %o0
        mov %i1, %o1
        call c_n_p              !       %o0 <- C_(n-1)^p
        nop
        mov %o0, %l0            !       %l0 <- %o0
        sub %i0, 1, %o0
        sub %i1, 1, %o1
        call c_n_p              !       %o0 <- C_(n-1)^(p-1)
        nop
        add %l0, %o0, %i0       !       %i0 <- %l0 + %o0
        ret                     !       return %i0
        restore
cnp_ret1:                       ! } else {
        mov 1, %i0              !       return 1
        ret                     ! }
        restore
```

– une fonction principale (`main`) qui affiche les 13 premières lignes du triangle.

Correction : *La fonction `main` est la même que pour l'exercice précédent.*