

Architecture des ordinateurs et systèmes d'exploitation

Corrigé du TP 5: Assembleur SPARC

Marcel Bosc

Christophe Dehlinger
Benoît Meister

Arnaud Giersch
Nicolas Passat

Mathieu Haefele

Remarques préliminaires :

- le nom d'un fichier contenant un programme assembleur doit obligatoirement avoir le suffixe `.s` ;
- on passe d'un programme assembleur à un fichier exécutable en utilisant `gcc` de la même manière qu'avec un programme C.

1. Étude d'un programme en assembleur SPARC

- (a) Récupérez le programme `addition.s` qui réalise la somme de 2 entiers. Compilez en utilisant la commande `gcc -o addition addition.s` et exécutez-le. URL :

<http://icps.u-strasbg.fr/~giersch/enseignement/2003-2004/Archi/TP05/addition.s>

Étudiez ce programme et rajoutez des commentaires explicatifs dans ce fichier `addition.s`.

Correction :

```
! Fonction addition
!   -> retourne la somme de ses deux arguments
!-----
.section      ".text"           ! -> code
.align 4     ! aligné sur 4 octets
.global add  ! exporte le symbole « add »

add:
    save %sp,-64,%sp          ! réserve de l'espace sur la pile
    add %i0,%i1,%i0           ! calcule la somme des paramètres,
                                ! le résultat est la valeur de retour
                                ! de la fonction
    ret                       ! retour de la fonction add
    restore

! Programme principal
!-----
.section      ".data"         ! -> données
.align 8     ! alignées sur 8 octets
.PRINTF1:
    .asciz "Entrez a et b : " ! chaîne de caractères avec zéro
                                ! terminal
.SCANF:
    .asciz "%d %d"           ! idem
.PRINTF2:
    .asciz "c : %d\n"        ! idem

.section      ".text"         ! -> code
.align 4     ! aligné sur 4 octets
.global main  ! exporte le symbole « main »

main:
    save %sp,-104,%sp        ! réserve de l'espace sur la pile:
                                ! 96 + 8 octets pour deux variables
                                ! locale aux adresses %fp-4 et %fp-8

    ! printf
    sethi %hi(.PRINTF1),%o0   ! %o0 <- .PRINTF1
    or %o0,%lo(.PRINTF1),%o0 ! %o0 <- %fp-8
    call printf                ! appel de « printf (.PRINTF1) »
    nop                        ! « nop » après un « call »

    ! scanf
    add %fp,-4,%o1            ! %o1 <- %fp-4
    add %fp,-8,%o2            ! %o2 <- %fp-8
    sethi %hi(.SCANF),%o0     ! %o0 <- .SCANF
    or %o0,%lo(.SCANF),%o0    ! %o0 <- .SCANF
    call scanf                ! appel de
                                ! « scanf (.SCANF), %fp-4, %fp-8) »
    nop                        ! « nop » après un « call »

    ! addition
    ld [%fp-4],%o0            ! %o0 <- [%fp-4]
    ld [%fp-8],%o1            ! %o1 <- [%fp-8]
    call add                  ! appel de « add ([%fp-4], [%fp-8]) »
                                ! résultat dans %o0
    nop                        ! « nop » après un « call »

    ! printf
    mov %o0,%o1               ! %o1 <- %o0
    sethi %hi(.PRINTF2),%o0   ! %o0 <- .PRINTF2
    or %o0,%lo(.PRINTF2),%o0  ! %o0 <- [%fp-4], [%fp-8]) »
    call printf                ! appel de « printf (.PRINTF2, %o0) »
    nop                        ! « nop » après un « call »

    ret                       ! retour de la fonction main
    restore
```

- (b) Réalisez un programme C faisant appel à une fonction prenant 8 paramètres, produisez le programme assembleur correspondant (option `-S` de `gcc` : `gcc -S fichiersource`).

Déterminez ensuite quels paramètres sont placés dans les registres, lesquels ne le sont pas et trouvez l'emplacement mémoire de ces derniers.

Correction :

- le programme C :

```

#include <stdio.h>

int add8 (int a, int b, int c, int d, int e, int f, int g, int h)
{
    return a + b + c + d + e + f + g + h;
}

```

```

int main (void)
{
    printf ("somme 1..8 = %d\n",
           add8 (1, 2, 3, 4, 5, 6, 7, 8));

    return 0;
}

```

– le code assembleur correspondant :

```

.file "Sarg.c"
.section ".text"
.align 4
.global add8
.type add8,#function
.proc 04
add8:
!#PROLOGUE# 0
save %sp, -112, %sp
!#PROLOGUE# 1
st %i0, [%fp+68]
st %i1, [%fp+72]
st %i2, [%fp+76]
st %i3, [%fp+80]
st %i4, [%fp+84]
st %i5, [%fp+88]
ld [%fp+68], %i1
ld [%fp+72], %i0
add %i1, %i0, %i0
ld [%fp+76], %i1
add %i0, %i1, %i0
ld [%fp+80], %i1
add %i0, %i1, %i0
ld [%fp+84], %i1
add %i0, %i1, %i0
ld [%fp+88], %i1
add %i0, %i1, %i0
ld [%fp+92], %i1
add %i0, %i1, %i0
ld [%fp+96], %i1
add %i0, %i1, %i0
nop
ret
restore
.LLfel:
.size add8,.LLfel-add8
.section ".rodata"

```

```

.align 8
.LLC0:
.asciz "somme 1..8 = %d\n"
.section ".text"
.align 4
.global main
.type main,#function
.proc 04
main:
!#PROLOGUE# 0
save %sp, -120, %sp
!#PROLOGUE# 1
mov 7, %o0
st %o0, [%sp+92]
mov 8, %o0
st %o0, [%sp+96]
mov 1, %o0
mov 2, %o1
mov 3, %o2
mov 4, %o3
mov 5, %o4
mov 6, %o5
call add8, 0
nop
mov %o0, %o1
sethi %hi(.LLC0), %o0
or %o0, %lo(.LLC0), %o0
call printf, 0
nop
mov 0, %o0
mov %o0, %i0
nop
ret
restore
.LLfe2:
.size main,.LLfe2-main
.ident "GCC: (GNU) 3.2"

```

On remarque que les 6 premiers paramètres sont passés par les registres %o0 à %o5 (%i0 à %i5 dans la fonction), les deux derniers sont passés sur la pile aux adresses %sp+92 et %sp+96 (%fp+92 et %fp+96 dans la fonction).

2. Exercices de programmation

- (a) Écrivez un programme assembleur calculant la factorielle d'un entier de manière itérative (une seule fonction principale contenant une boucle).

Correction :

```

!!!
!!! calcul de la factorielle d'un entier, version itérative
!!!

! Programme principal
!-----

.section ".data" ! -> données
.align 8
.PRINTF1:
.asciz "n? "
.SCANF:
.asciz "%u"
.PRINTF2:
.asciz "n! = %u\n"

.section ".text" ! -> code
.align 4
.global main
main:
save %sp, -96, %sp ! réserve de la place sur la pile
! pour un entier [%fp-4] (mais on
! arrondit à un multiple de 8)

sethi %hi(.PRINTF1), %o0
or %o0, %lo(.PRINTF1), %o0
call printf ! printf (.PRINTF1)
nop

```

```

sethi %hi(.SCANF), %o0
or %o0, %lo(.SCANF), %o0
add %fp, -4, %o1
call scanf ! scanf (.SCANF, %fp-4)
nop

! -> calcul de [%fp-4]! dans %i1,
! utilisation de %i0 comme compteur

ld [%fp-4], %i0 ! %i0 <- [%fp-4]
mov 1, %i1 ! %i1 <- 1

loop:
cmp %i0, 1 ! while (%i0 > 1) {
ble end_loop !
!
umul %i0, %i1, %i1 ! %i1 <- %i1 * %i0
dec %i0 ! %i0 --
b loop ! }
nop

end_loop:
sethi %hi(.PRINTF2), %o0
or %o0, %lo(.PRINTF2), %o0
mov %i1, %o1
call printf ! printf (.PRINTF2, %i1)
nop

clr %i0 ! return 0
ret
restore

```

- (b) Écrivez un programme assembleur calculant la factorielle d'un entier de manière *réursive*.

Correction :

```

!!!
!!! calcul de la factorielle d'un entier, version récursive
!!!

! fonction fact
! -> retourne la factorielle de son argument
!-----
.section      ".text"                ! -> code
.align 4
.global fact

fact:
    save %sp, -96, %sp

    cmp %i0, 1                       ! if (%i0 > 1) {
    ble end_fact1                    !
    nop                               !
    sub %i0, 1, %i0                  !     %i0 <- %i0 - 1
    call fact                         !     %i0 <- fact (%i0)
    nop                               !
    umul %i0, %i0, %i0               !     %i0 <- %i0 * %i0
    b end_fact                        ! } else {
    nop                               !
end_fact1:
    mov 1, %i0                        !     %i0 <- 1
    nop                               !
end_fact:
    ret                               ! return %i0
    restore

! Programme principal
!-----
.section      ".data"                ! -> données
.align 8
.PRINTF1:
.asciz "n?"

```

```

.SCANF:
.asciz "%u"
.PRINTF2:
.asciz "n! = %u\n"

.section      ".text"                ! -> code
.align 4
.global main

main:
    save %sp, -96, %sp                ! réserve de la place sur la pile
                                        ! pour un entier [%fp-4] (mais on
                                        ! arrondit à un multiple de 8)

    sethi %hi(.PRINTF1), %i0
    or %i0, %lo(.PRINTF1), %i0
    call printf                        ! printf (.PRINTF1)
    nop

    sethi %hi(.SCANF), %i0
    or %i0, %lo(.SCANF), %i0
    add %fp, -4, %i0
    call scanf                          ! scanf (.SCANF, %fp-4)
    nop

    ld [%fp-4], %i0
    call fact                          ! %i0 <- fact ((%fp-4))
    nop

    mov %i0, %i1
    sethi %hi(.PRINTF2), %i0
    or %i0, %lo(.PRINTF2), %i0
    call printf                        ! printf (.PRINTF2, %i0)
    nop

    clr %i0                            ! return 0
    ret
    restore

```

(c) Modifiez-le programme précédent pour qu'il affiche à chaque étape de la récursion les valeurs des pointeurs de pile (%sp et %fp).

Correction :

```

!!!
!!! calcul de la factorielle d'un entier, version récursive,
!!! affichage de %fp et %sp
!!!

! fonction fact
! -> retourne la factorielle de son argument
!-----
.section      ".data"                ! -> données
.align 8
.PRINTF_DEBUG:
.asciz "# fact (%u): %fp = %p, %sp = %p\n"

.section      ".text"                ! -> code
.align 4
.global fact

fact:
    save %sp, -96, %sp

    sethi %hi(.PRINTF_DEBUG), %i0
    or %i0, %lo(.PRINTF_DEBUG), %i0
    mov %i0, %i1
    mov %fp, %i2
    mov %sp, %i3
    call printf                        ! printf (.PRINTF_DEBUG,
                                        !     %i0, %fp, %sp)
    nop

    cmp %i0, 1                       ! if (%i0 > 1) {
    ble end_fact1                    !
    nop                               !
    sub %i0, 1, %i0                  !     %i0 <- %i0 - 1
    call fact                         !     %i0 <- fact (%i0)
    nop                               !
    umul %i0, %i0, %i0               !     %i0 <- %i0 * %i0
    b end_fact                        ! } else {
    nop                               !
end_fact1:
    mov 1, %i0                        !     %i0 <- 1
    nop                               !
end_fact:
    ret                               ! return %i0
    restore

```

```

! Programme principal
!-----
.section      ".data"                ! -> données
.align 8
.PRINTF1:
.asciz "n?"
.SCANF:
.asciz "%u"
.PRINTF2:
.asciz "n! = %u\n"

.section      ".text"                ! -> code
.align 4
.global main

main:
    save %sp, -96, %sp                ! réserve de la place sur la pile
                                        ! pour un entier [%fp-4] (mais on
                                        ! arrondit à un multiple de 8)

    sethi %hi(.PRINTF1), %i0
    or %i0, %lo(.PRINTF1), %i0
    call printf                        ! printf (.PRINTF1)
    nop

    sethi %hi(.SCANF), %i0
    or %i0, %lo(.SCANF), %i0
    add %fp, -4, %i0
    call scanf                          ! scanf (.SCANF, %fp-4)
    nop

    ld [%fp-4], %i0
    call fact                          ! %i0 <- fact ((%fp-4))
    nop

    mov %i0, %i1
    sethi %hi(.PRINTF2), %i0
    or %i0, %lo(.PRINTF2), %i0
    call printf                        ! printf (.PRINTF2, %i0)
    nop

    clr %i0                            ! return 0
    ret
    restore

```

Exemple d'exécution :

```

$ ./fact_r_print
n? 6
# fact (6): %fp = ffbef800, %sp = ffbef7a0

```

```

# fact (5): %fp = ffbef7a0, %sp = ffbef740
# fact (4): %fp = ffbef740, %sp = ffbef6e0
# fact (3): %fp = ffbef6e0, %sp = ffbef680
# fact (2): %fp = ffbef680, %sp = ffbef620
# fact (1): %fp = ffbef620, %sp = ffbef5c0
n! = 720

```

- (d) Dans un processeur où la multiplication n'est pas implémentée par un circuit, celle-ci peut être réalisée efficacement en se basant sur l'algorithme suivant :

$$a \times b := \text{soit } b' = b/2;$$

si $b = 0$ **alors** 0
sinon si $b = 2 \times b'$ **alors** $(a \times 2) \times b'$
sinon $((a \times 2) \times b') + a$

En vous appuyant sur cette méthode, proposez une fonction assembleur réalisant la multiplication entière en utilisant uniquement des additions et des décalages.

Correction :

Deux versions,
– une version récursive :

```

!!!
!!! fonction multiplication: version récursive
!!!
.section ".text"
.align 4
.global mult

mult:
    save %sp, -96, %sp

    cmp 0, %i1          ! if (%i1 != 0) {
    be zero            !
    nop                !
    sll %i0, 1, %o0    ! %o0 <- %i0 << 1
    srl %i1, 1, %o1    ! %o1 <- %i1 >> 1

```

```

    call mult          ! %o0 <- mult (%o0, %o1)
    nop                !
    andcc %i1, 1, %g0 ! if (%i1 & 1 != 0) {
    bz even            ! // -> équivalent à
    nop                ! // if (%i1 % 2 == 1) {
    add %o0, %i0, %o0 ! %o0 <- %o0 + %i0
    even:              !
    mov %o0, %i0       ! %i0 <- %o0
    b return           !
    nop                !
    zero:              ! } else {
    mov 0, %i0         ! %i0 <- 0
    nop                !
    return:            !
    ret                ! return %i0
    restore

```

– une version itérative :

```

!!!
!!! fonction multiplication: version itérative
!!!
.section ".text"
.align 4
.global mult

mult:
    save %sp, -96, %sp

    clr %i0            ! %i0 <- 0

loop:
    cmp 0, %i1        ! while (%i1 != 0) {
    be end_loop       !

```

```

    nop                !
    andcc %i1, 1, %g0 ! if (%i1 & 1 != 0) {
    bz even            ! // -> équivalent à
    nop                ! // if (%i1 % 2 == 1) {
    add %i0, %i0, %i0 ! %i0 <- %i0 + %i0
    even:              !
    sll %i0, 1, %i0    ! %i0 <- %i0 << 1
    srl %i1, 1, %i1    ! %i1 <- %i1 >> 1
    b loop             !
    nop                !
end_loop:
    mov %i0, %i0       ! return %i0
    ret
    restore

```

- (e) Utilisez la fonction précédente dans un programme C. L'exécutable s'obtient en donnant simplement le fichier C et le fichier assembleur au programme gcc, ce dernier s'occupe de faire les liens.

Correction :

```

#include <stdio.h>
extern unsigned mult (unsigned, unsigned);

int main (void)
{
    unsigned a, b;

```

```

    printf ("a, b? ");
    scanf ("%u %u", &a, &b);
    printf (" %u x %u = %u\n", a, b, mult (a, b));

    return 0;
}

```

- (f) Écrivez un programme assembleur de recherche de l'élément minimum d'un tableau. Le tableau est une variable locale à la routine principale. Cette dernière fait appel à une routine pour la recherche de l'élément minimum d'un tableau.

Correction :

```

!!!
!!! recherche du minimum dans un tableau
!!!
! Fonction min: recherche du minimum dans un tableau d'entiers
! 2 arguments: adresse du tableau et nombre d'éléments
! retourne l'indice du minimum dans le tableau
! précondition: la tableau contient au moins 1 élément
!-----

.section      ".text"                ! -> code
.align 4
.global min

min:
    save %sp, -64, %sp

                                ! registres locaux utilisés:
                                ! %10: index du min. courant
                                ! %11: minimum courant
                                ! %12: index courant
                                ! %13: valeur courante

    clr %10                      ! %10 <- 0
    ld [%10], %11                ! %11 <- tab[0]
    clr %12                      ! %12 <- 0
    orcc %11, %g0, %g0

min_loop:
    bz end_min                   ! while (%11 != 0) {
    nop                          !
    inc %12                      ! %12 ++
    add %10, 4, %10              ! %10 <- %10
                                ! + sizeof(int)
    ld [%10], %13                ! %13 <- tab[%12]
    cmp %11, %13                 ! if (%11 > %13) {
    ble min_ok                   !
    nop                          !
    mov %12, %10                 ! %10 <- %12
    mov %13, %11                 ! %11 <- %13
min_ok:
    deccc %11                    ! %11 --
    b min_loop                   ! }
    nop

end_min:
    mov %10, %10                 ! return %10
    ret
    restore

! Programme principal
! -> lit 10 entiers et trouve le minimum
!-----

.section      ".data"                ! -> données
.align 8
.PRINTF1:
.asciz "Entrez 10 entiers:\n"
.SCANF:

                                .asciz "%d"
.PRINTF2:
.asciz "minimum: [%d] = %d\n"

.section      ".text"                ! -> code
.align 4
.global main

main:
    save %sp, -136, %sp          ! réserve de la place pour un
                                ! tableau de 10 entiers à
                                ! l'adresse %fp-40

    set .PRINTF1, %o0            ! printf (.PRINTF1)
    call printf
    nop

                                ! lecture des entiers
                                ! %10 <- 10 (nombre d'entiers
                                ! à lire)
                                ! %11 <- %fp-40 (adresse du
                                ! tableau)
read_loop:
    do {
        set .SCANF, %o0
        mov %11, %o1
        call scanf
        nop
        add %11, 4, %11
        deccc %10
        bnz read_loop
        nop
    } while (%10 != 0)

                                ! calcul du minimum
                                ! %o0 <- %fp-40 (adresse du
                                ! tableau)
                                ! %o1 <- 10 (taille du
                                ! tableau)
                                ! %o0 <- min (%o0, %o1)
    sub %fp, 40, %o0
    mov %10, %o1
    call min
    nop

                                ! affichage du résultat
                                ! %o1 <- %o0 (indice du min.)
                                ! %o0 <- %o0 * 4
                                ! (4 == sizeof (int))
                                ! %o0 <- %o0 - 40
                                ! %o2 <- [%fp+%o0] (tab[%o1],
                                ! valeur du min.)
    mov %o0, %o1
    sll %o0, 2, %o0
    sub %o0, 40, %o0
    ld [%fp+%o0], %o2
    set .PRINTF2, %o0
    call printf
    nop

                                ! printf (.PRINTF2,
                                ! indice_du_min,
                                ! valeur_du_min)

    clr %10                      ! return 0
    ret
    restore

```