Code produit par gcc

Cette fonction C calcule n termes de la suite de Fibonacci, et les range dans un tableau t :

```
void fibo1 (int *t, int n){
    int tmp, u = 1, v = 1;
    for (int i = 0; i < n; i++){
        tmp = v;
        v = u + v;
        u = tmp;
        t[i] = v;
    }
}
```

Si le fichier est appelé `tp6.c` la commande pour produire le code assembleur dans `tp6.s` est la suivante :

```
gcc -S -std=c99 tp6.c
```

Voici le résultat (certains détails peuvent dépendre de la version du compilateur) :

```
fibo1:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    $1, -8(%ebp)
    movl    $1, -12(%ebp)
    movl    $0, -16(%ebp)
    jmp .L2
.L3:
    movl    -12(%ebp), %eax
    movl    %eax, -4(%ebp)
    movl    -8(%ebp), %eax
    addl    %eax, -12(%ebp)
    movl    -4(%ebp), %eax
    movl    %eax, -8(%ebp)
    movl    -16(%ebp), %eax
    sall    $2, %eax
    addl    8(%ebp), %eax
    movl    -12(%ebp), %edx
    movl    %edx, (%eax)
    addl    $1, -16(%ebp)
.L2:
    movl    -16(%ebp), %eax
    cmpl    12(%ebp), %eax
    jl  .L3
    leave
    ret
```

1. Commenter ce code, en expliquant en particulier ce que contiennent les mots d'adresses $b - 16$ à $b + 12$, où b désigne le contenu du registre `%ebp`.
2. A quoi sert l'instruction `subl $16,%esp` ?
3. Que fait l'instruction `sall $2,%eax` ? Pourquoi ?
4. Rechercher sur Internet quel est le rôle de l'instruction `leave`.

Pour produire le code assembleur optimisé dans `tp6b.s` il suffit d'ajouter l'option `-O2` et de spécifier le nom du fichier résultatde la compilation (pour ne pas écraser `tp6.s`) :

```
gcc -S -std=c99 -O2 -o tp6b.s tp6.c
```

```
fibo1:
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %edi
    movl    8(%ebp), %edi
    pushl   %esi
    movl    12(%ebp), %esi
    pushl   %ebx
    testl   %esi, %esi
    jle .L4
    xorl    %eax, %eax
    movl    $1, %edx
    movl    $1, %ebx
    jmp .L3
.L6:
    movl    %ecx, %edx
.L3:
    leal    (%edx,%ebx), %ecx
    movl    %edx, %ebx
    movl    %ecx, (%edi,%eax,4)
    addl    $1, %eax
    cmpl    %esi, %eax
    jne .L6
.L4:
    popl    %ebx
    popl    %esi
    popl    %edi
    popl    %ebp
    ret
```

5. Pourquoi les références comme `-8(%ebp)` ont-elles disparu du code optimisé ?
6. Construire une table d'utilisation des registres.
7. Pourquoi les registres `%edi`, `%esi` et `%ebx` sont-ils sauvegardés avant utilisation ?
8. Rechercher sur Internet quel est le rôle de l'instruction `leal`, et décrypter son usage dans ce programme.
9. Dans le code C, dupliquer la fonction `fibo1` en une fonction `fibo2`, et modifier (très légèrement) le code de `fibo2` pour utiliser un pointeur au lieu de $t[i]$; analyser les modifications dans le code optimisé produit.
10. Ajouter au code C une version `fibo3` qui n'utilise plus de variable locale autre que i, en codant directement la récurrence sous la forme $t_{i+2} = t_i + t_{i+1}$; analyser les modifications dans le code optimisé produit.

Corrigé : code non optimisé commenté

```
fibo1:
    pushl   %ebp                    # sauvegarde ebp
    movl    %esp, %ebp
    # allocation memoire pour 4 variables locales
    subl    $16, %esp
    movl    $1, -8(%ebp)            # u = 1
    movl    $1, -12(%ebp)           # v = 1
    movl    $0, -16(%ebp)           # i = 0
    jmp .L2
.L3:
    movl    -12(%ebp), %eax
    movl    %eax, -4(%ebp)          # tmp = v
    movl    -8(%ebp), %eax          # u
    # jeu d'instructions CISC: addition directement en memoire
    addl    %eax, -12(%ebp)         # v = u + v
    movl    -4(%ebp), %eax
    movl    %eax, -8(%ebp)          # u = tmp
    movl    -16(%ebp), %eax         # compteur i
    # decalage (shift) de 2 bits vers la gauche
    sall    $2, %eax                # 4*i
    addl    8(%ebp), %eax           # t + 4*i
    movl    -12(%ebp), %edx         # v
    movl    %edx, (%eax)            # t[i] = v
    addl    $1, -16(%ebp)           # i++
.L2:
    movl    -16(%ebp), %eax         # i
    cmpl    12(%ebp), %eax          # calcule i-n (n = second argument)
    jl  .L3                         # cas i-n < 0 soit i < n
    # leave combine deux instructions (CISC):
    #   movl    %ebp, %esp
    #   popl    %ebp
    leave
    ret
```

Corrigé : code optimisé commenté

Les variables locales sont conservées dans des registres (il y en a juste assez) :

```
fibo1:
    pushl   %ebp            # sauvegarde ebp
    movl    %esp, %ebp
    pushl   %edi            # sauvegarde edi avant utilisation
    movl    8(%ebp), %edi   # t
    pushl   %esi            # sauvegarde esi avant utilisation
    movl    12(%ebp), %esi  # n
    pushl   %ebx            # sauvegarde ebx avant utilisation
    testl   %esi, %esi
    jle .L4                 # cas n <= 0
    xorl    %eax, %eax      # i = 0
    movl    $1, %edx        # u ou v ? la suite montre que c'est v
    movl    $1, %ebx        # u ou v ? la suite montre que c'est u
    jmp .L3
.L6:
    movl    %ecx, %edx          # v = ecx
.L3:
    leal    (%edx,%ebx), %ecx   # ecx = edx + ebx = u + v
    movl    %edx, %ebx          # u = v
    movl    %ecx, (%edi,%eax,4) # t[i] = ecx
    addl    $1, %eax            # i++
    cmpl    %esi, %eax          # calcule i-n
    jne .L6                     # cas i <> n
.L4:
    # restaure les trois registres "callee save", puis ebp
    popl    %ebx
    popl    %esi
    popl    %edi
    popl    %ebp
    ret
```

Si dans le code C on remplace `t[i]=v` par `*t++=v`, le code optimisé est le même avec cette version du compilateur !

Corrigé : version 3

```
void fibo3 (int *t, int n){
    t[0]=1;
    t[1]=1;
    for (int i = 0; i < n; i++){
        t[i+2] = t[i]+t[i+1];
    }
}
```

```
fibo3:
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %ebx
    movl    12(%ebp), %ebx  # n
    movl    8(%ebp), %edx   # t
    testl   %ebx, %ebx
    movl    $1, (%edx)      # t[0] = 1
    movl    $1, 4(%edx)     # t[1] = 1
    jle .L15                # cas n <= 0
    movl    $1, %ecx
    xorl    %eax, %eax      # i = 0
    jmp .L14
.L16:
    movl    4(%edx,%eax,4), %ecx    # t[i+1]
.L14:
    addl    (%edx,%eax,4), %ecx     # ecx = t[i] + t[i+1]
    movl    %ecx, 8(%edx,%eax,4)    # t[i+2] = ecx
    addl    $1, %eax                # i++
    cmpl    %ebx, %eax
    jne .L16                # cas i <> n
.L15:
    popl    %ebx
    popl    %ebp
    ret
```

On a gagné deux registres, au prix d'un adressage complexe spécifique du jeu d'instructions x86.