Chapitre 4

Incorporation de parties en langage d'assemblage dans un programme en langage C

Le langage d'assemblage a des avantages par rapport à un langage « évolué » tel que le langage C mais il est plus difficile pour un être humain de concevoir un programme en langage d'assemblage qu'en langage évolué (c'est la raison d'être des langages évolués). En pratique, on écrit un programme en langage évolué sauf en ce qui concerne les parties qui demandent une attention particulière, par exemple du point de vue de leur efficacité. Il faut alors combiner langage évolué et langage d'assemblage.

Comment inclure une partie de code en langage d'assemblage dans un programme en langage C ?

La façon de le faire dépend du compilateur qu'on utilise. Nous utilisons dans ce chapitre le compilateur `gcc`. Il est inutile de dire qu'il ne faut donc pas appliquer ce qui est écrit ci-dessous pour un autre compilateur.

Le point le plus délicat est le passage des paramètres. Nous commencerons donc par les programmes n'ayant pas besoin de passage de paramètre.

4.1 Programmes sans paramètre

On peut utiliser cette première méthode lorsque le code en langage d'assemblage n'utilise pas les valeurs des variables du programme écrit en langage C.

4.1.1 Principe

Expliquons la méthode sur un exemple. Elle consisite à utiliser la fonction `__asm__()` de `libc`.

Exemple.- Considérons le programme `printint.s` écrit précédemment, et `printint.o` le résultat de l'assemblage de ce programme. Soit le programme `tst-printint.s` en langage d'assemblage :

```
        .section .text
        .type printint,@function
        .globl _start
_start:
        nop
        movq $12345, %rax
        call printint
        movq $60,%rax
        syscall
        nop
```

Remplaçons-le par le programme `c-printint.c` en langage C suivant :

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
__asm__("nop\n\t"
        "movq $12345, %rax\n\t"
        "call printint\n\t"
        "movq $60,%rax\n\t"
        "syscall\n\t"
        "nop\n\t");
}
```

Compilons `c-printint.c` (sans oublier `printint.o` !) :

```
$ gcc c-printint.c printint.o
$ ./a.out
12345
```

Syntaxe.- On voit comment produire l'argument de la fonction `__asm__()` à partir du code en langage d'assemblage `gas` :

— les noms des sections (ainsi que les déclarations) sont enlevés ;
— chaque instruction est écrite sur une ligne séparée ;
— chaque instruction commence par '`"`' et se termine par '`\n\t"`'.

4.1.2 Première application : utilisation du compteur de tops d'horloge

Outre l'utilisation de tous les éléments du langage C dans un programme contenant un appel à la fonction `__asm__()`, l'inclusion de code en langage d'assemblage nous permet d'utiliser les instructions du processeur non disponibles autrement. L'exemple typique est l'utilisation de l'instruction `rdtsc` permettant de lire le registre spécial `tsc` (*Time Stamp Counter*), comptant le nombre de tops d'horloge du processeur.

Question.- Dans quelles situations est-il utile de connaître le nombre de tops d'horloge du processeur ?

Supposons que nous voulions estimer l'efficacité d'un programme. Une mesure possible est le nombre de tops d'horloge (cycles) que le processeur passe à exécuter ce programme. L'instruction `rdtsc` lit les 64 bits du registre `tsc` et place le mot de poids faible du contenu dans `%eax` (une partie du registre `%rax`) et le mot de poids fort dans `%edx`.

Question.- Après avoir effectué l'instruction `rdtsc` quels sont les 32 bits de poids fort des deux registres `%rax` et `%rdx` ? Confirmez expérimentalement votre réponse.

Question.- Comment obtenir la valeur de `tsc` comme valeur du type `long long unsigned int` stockée dans la pile ?

Exercice.- Écrire un programme `tst-rdtsc-01.c` calculant le nombre de cycles que le processeur passe à exécuter :

```
x=0; x++
```

pour un entier `x`.

4.1.3 Deuxième application : détermination de l'adresse d'une variable

Considérons un autre exemple, nous montrant comment déterminer l'adresse d'une variable.

Exemple.- Supposons que l'on veuille afficher la chaîne de caractères « Hello, world ! » sans l'étiqueter ni la mettre dans la section `.data`, c'est-à-dire en plaçant la chaîne de caractères à afficher directement dans le programme. La structure de ce programme (appelons-le `wrstr.s`) est claire :

```
    .section .text
    .globl   _start

_start: movq $1, %rax     ## 'write' syscall
        movq $2, %rdi     ## to 'stderr'

        ???????????       ## string address must be in %rsi !!!!!

        movq $14, %rdx    ## string length
        syscall           ## do it!
        movq $60, %rax
        syscall           ## 'exit' syscall
        .string "Hello, world!\n"
        nop
```

Il reste à calculer l'adresse de début de cette chaîne de caractères pour la placer dans `%rsi`. Comment faire ?

Point de syntaxe 1.- En langage d'assemblage `gas` le point '`.`' à la place d'une adresse signifie **l'adresse en cours**.

Point de syntaxe 2.- En langage d'assemblage `gas` l'expression '`.k`', où `k` est un entier **signé**, signifie l'adresse décalée de `k` par rapport à l'adresse en cours (on peut écrire, par exemple, `.+23` ou `.-18`). L'instruction `lea` permet de calculer la valeur de cette adresse.

Première idée.- Nous pouvons mettre à la place de "`?????????`" ci-dessus :

```
leaq .??? , %rsi
```

Il reste alors à calculer la distance en octets entre cette instruction `leaq` et le début de la chaîne de caractères « Hello, world ! ». On peut évidemment la calculer en connaissant la longueur de chaque instruction en langage machine, mais calculons-la d'une façon plus « pratique ». Mettons une valeur « raisonnable » à la place de '`???`' et assemblons le programme.

Question.- Cette « valeur raisonnable » doit-elle être :

— négative ou positive ?
— >100 ou <100 ?

Tâtonnement.- Prenons +8 pour essayer :

```
.section .text
.globl _start

_start:
movq $1, %rax     ## 'write' syscall
        movq $2, %rdi     ## to 'stderr'

        leaq .+8, %rsi    ## string address must be in %rsi !!!!!

        movq $14, %rdx    ## string length
        syscall           ## do it!
        movq $60, %rax
        syscall           ## 'exit' syscall
        .string "Hello, world!\n"
        nop
```

```
$ as -gstabs -o wrstr.o wrstr.s
$ ld -o wrstr wrstr.o
$ gdb wrstr
(gdb) disassem _start
Dump of assembler code for function _start:
   0x0000000000400078 <+0>: mov    $0x1,%rax
   0x000000000040007f <+7>: mov    $0x2,%rdi
   0x0000000000400086 <+14>: lea    0x40008e,%rbx
   0x000000000040008e <+22>: mov    $0xe,%rdx
   0x0000000000400095 <+29>: syscall
   0x0000000000400097 <+31>: mov    $0x3c,%rax
   0x000000000040009e <+38>: syscall
   0x00000000004000a0 <+40>: rex.W
   0x00000000004000a1 <+41>: gs
   0x00000000004000a2 <+42>: insb   (%dx),%es:(%rdi)
   0x00000000004000a3 <+43>: insb   (%dx),%es:(%rdi)
   0x00000000004000a4 <+44>: outsl  %ds:(%rsi),(%dx)
   0x00000000004000a5 <+45>: sub    $0x20,%al
   0x00000000004000a7 <+47>: ja     0x400118
   0x00000000004000a9 <+49>: jb     0x400117
   0x00000000004000ab <+51>: and    %ecx,%fs:(%rdx)
   0x00000000004000ae <+54>: .byte 0x0
   0x00000000004000af <+55>: nop
End of assembler dump.
```

Où est située notre chaîne de caractères ? Entre le deuxième `syscall` et l'instruction `nop`. Pourquoi `gdb` nous y montre-t-il n'importe quoi ? Parce qu'il interprète chaque suite d'octets comme le code d'une suite d'instructions et non comme une suite de codes ASCII. La chaîne de caractères à afficher commence donc à l'adresse `0x00000000004000a0`. Vérifions-le :

```
(gdb) x /15bx 0x00000000004000a0
0x4000a0 <_start+40>: 0x48 0x65 0x6c 0x6c 0x6f 0x2c 0x20 0x77
0x4000a8 <_start+48>: 0x6f 0x72 0x6c 0x64 0x21 0x0a 0x00
```

On voit les codes ASCII ? Lisez les... Ou bien faisons autrement :

```
(gdb) x /1bs 0x00000000004000a0
0x4000a0 <_start+40>:  "Hello, world!\n"
```

Vu la ligne :

```
0x0000000000400086 <+14>: lea    0x40008e,%rsi
```

nous pouvons conclure que le décalage qu'on recherche est égal à +26. Plaçons donc celui-ci et vérifions :

```
$ as -o wrstr.o wrstr.s
$ ld -o wrstr wrstr.o
$ ./wrstr
Hello, world!
```

Plaçons le code ci-dessus dans un programme en langage C (le programme ne fera que ça) :

```
    wrstr-in.c:
int main() {
   __asm__("movq $1, %rax\n\t"
           "movq $2, %rdi\n\t"
           "leaq .+26, %rsi\n\t"
           "movq $14, %rdx\n\t"
           "syscall\n\t"
           "movq $60, %rax\n\t"
           "syscall\n\t"
           ".string\"Hello, world!\\n\"\n\t"
           "nop\n\t");
}
```

```
$ gcc wrstr-in.c
$ ./a.out
Hello, world!
```

Exercice.- Pour remplacer la chaîne de caractères '`Hello, world!\n`' par la chaîne de caractères '`/bin/sh\0`', que faut-il modifier de plus dans le fichier source ? Appelez le nouveau programme `wrcmd-in.c` et vérifiez qu'il se comporte comme prévu !

Exercice.- Écrire comme programme `execmd-in.c` la modification du programme `wrcmd-in.c` dans lequel l'appel système `write` est remplacé par l'appel système `execve` appliqué à `/bin/sh`. Que faut-il modifier de plus ? Vérifiez que le nouveau programme se comporte comme prévu.

4.2 Programmes avec paramètres

4.2.1 Intérêt des paramètres

La méthode qu'on vient de présenter dans la section précédente n'est pas toujours commode. L'inconvénient évident est l'absence de liens entre la partie écrite en langage d'assemblage et celle écrite en langage C. Plus précisement, on ne voit pas comment lier les variables du programme et les registres du processeur qui, eux, sont disponibles en langage d'assemblage.

Pourquoi vouloir le faire ? Il y a au moins deux raisons importantes :

— Premièrement, certaines parties du code peuvent être « compilées à la main » par le programmeur plus efficacement que ne le fait le compilateur. Par exemple, il est plus efficace de placer la variable de contrôle d'une boucle `for` dans un registre, lorsque cela est possible, alors que le compilateur la place habituellement en mémoire.

— Deuxièmement, certaines instructions ne sont pas disponibles en langage C de façon simple bien qu'elles puissent être extrêmement utiles dans certaines situations. Un exemple typique est l'instruction `rdtsc`, rencontrée dans la section précédente. Un autre exemple important est l'instruction `cpuid` permettant de déterminer plusieurs caractéristiques du processeur qui l'exécute. Pour en savoir plus, on peut consulter :

`http://www.intel.com/Assets/PDF/appnote/241618.pdf`

Il existe une extension du langage d'assemblage permettant d'établir des liens entre les variables utilisées dans un code en C et les registres utilisés en langage d'assemblage, appelée « *inline extended assembly* » en anglais. Commençons par un exemple simple, qui nous permettra de comprendre la structure et la syntaxe d'un tel mélange (C + Asm). On peut trouver plus d'informations sur les pages « info » en faisant :

```
$ info gcc 'C extensions' 'Extended Asm'
```

pour y accéder.

Exemple.- Considérons le programme `inl-01.c` :

```
void main(void)
  {
  long long int x=1, y=3, z=5, s;
  s=x+y+z;
  printf("s=%d\n",s);
  exit(0);
  }
```

Nous voulons écrire une variante de ce programme dans laquelle la somme `x+y+z` soit calculée en langage d'assemblage. Pour cela, la valeur de chaque variable doit être placée dans un registre.

Notons que, dans cet exemple, les variables `x`, `y` et `z` peuvent être considérées comme des variables d'entrée, en lecture uniquement, et la variable `s` comme une variable de sortie, en écriture uniquement. Nous voulons, de plus, souligner cette distinction en langage d'assemblage.

Notons aussi que, dans le calcul, nous pouvons utiliser des registres « intermédiaires », dans lesquels aucune variable n'a été placée au début. Le compilateur doit être averti que le contenu de ces registres change lors du calcul. On dit qu'ils sont « gâtés » (*clobbered* en anglais), c'est-à-dire qu'on ne doit pas compter retrouver les valeurs de ces registres après l'exécution du bout de code.

Syntaxe.- La structure de la partie en langage d'assemblage est la suivante :

```
asm(
    <code en langage d'assemblage qui calcule la somme>
   : <liste des registres pour les variables "write-only">
   : <liste des registres pour les autres variables>
   : <liste des registres "gates">
   );
```

dans laquelle :

— Le code en langage d'assemblage est comme précédemment :

```
"<instruction>\n\t"
"<instruction>\n\t"
  ............
"<instruction>"
```

— La liste des registres pour les variables d'entrée est de la forme :

```
"=<registre>" (<variable>), "=<registre>" (<variable), ... , "=<registre>" (<variable)
```

— La liste de registres pour les autres variables est de la forme :

```
"<registre>" (<variable>), "<registre>" (<variable), ... , "<registre>" (<variable)
```

— La liste des registres « gâtés » est de la forme :

```
"<registre>", "<registre>", ... , "<registre>"
```

Qui attribue les registres aux variables ? Cela peut être fait soit par le programmeur, soit automatiquement par le compilateur.

4.2.2 Attribution des registres par le compilateur

Laissons d'abord faire le travail à `gcc`.

Syntaxe.- La partie en langage d'assemblage du programme se présente alors comme suit :

```
asm(
    <code en langage d'assemblage>
   : "=r" (s)
   : "r" (x), "r" (y), "r" (z)
   : <liste des registres "gates">
   );
```

La lettre `r` ci-dessus désigne le nom d'un « registre quelconque » : `gcc` y substitue un nom concret de registre.

Comment écrire le code en langage d'assemblage en utilisant les « registres quelconques » ? Numérotons les variables dans les deux listes de variables ci-dessus par les nombres `0, 1, ...` d'abord dans la liste des variables de sortie de la gauche vers la droite, continuons ensuite par la liste des autres variables dans le même ordre. Le numéro obtenu par une variable est utilisé dans le code comme nom du registre attribué à cette variable. Dans notre cas on obtient :

Variable	Numéro	Registre
s	0	%0
x	1	%1
y	2	%2
z	3	%3

Compilons à la main la ligne `s = x + y + z`, ce qui nous permet d'obtenir, enfin, une version du code :

```
movq %1, %0
addq %2, %0
addq %3, %0
```

Exercice 1.- Réécrire le code ci-dessus en langage C **ligne par ligne.** Qu'obtient-on ?

Notre programme `inl-02.c` devient :

```
void main(void)
  {
  long long int x=1, y=3, z=5, s;
  asm(
      "movq %1, %0\n\t"
      "addq %2, %0\n\t"
      "addq %3, %0"
      : "=r" (s)
      : "r" (x), "r" (y), "r" (z)
     );
  printf("s=%d\n",s);
  exit(0);
  }
```

Remarque sur la syntaxe.- Dans la version du code ci-dessus, il n'y a pas de registres autres que `%0, %1, %2, %3`, c'est-à-dire que la liste des registres « gâtés » est vide. Pour cette raison, on a pu omettre le `':'` à la fin. Mais on est obligé de garder le `':'` si la liste vide n'est pas la dernière.

Exemple 1 (suite).- Ajoutons les commandes `#include` du préprocesseur, compilons et lançons le programme :

```
$ gcc -O0 inl-02.c
$ ./a.out
s=10
```

Ça marche !

Exemple 2.- Pensez-vous avoir tout appris sur la façon de programmer en mélangeant langage C et langage d'assemblage ? Considérons un programme `inl-03.c` un peu plus complexe :

```
void main(void)
  {
  long long int x=2,y=3,z=5,s;
  s=(x+y)*x*x+z;
  printf("s=%d\n",s);
  exit(0);
  }
```

Question.- Quelle valeur affiche ce programme ?

Exemple 2 (suite).- Nous voulons toujours calculer la valeur de `s` en langage d'assemblage. Faisons comme dans l'exemple 1 : prenons les mêmes listes de registres et laissons `gcc` faire la repartition précise. Notons que les numéros des variables ainsi que des registres sont exactement les mêmes que dans l'exemple 1 (voir la table ci-dessus). Le résultat de la compilation à la main de l'affectation $s = (x + y) * x * x + z$ est :

```
movq %1, %0
addq %2, %0
mulq %1
mulq %1
addq %3, %0
```

Exercice.- Réécrire le code ci-dessus en langage C **ligne par ligne.** Qu'obtient-on ?

Voici le programme « mixte » `inl-04.c` :

```
void main(void)
  {
  long long int x=2,y=3,z=5,s;
  /*  s=(x+y)*x*x+z; */
  asm(
      "movq %1, %0\n\t"
      "addq %2, %0\n\t"
      "mulq %1\n\t"
      "mulq %1\n\t"
      "addq %3, %0\n\t"
      : "=r" (s)
      : "r" (x), "r" (y), "r" (z)
     );
  printf("s=%d\n",s);
  exit(0);
  }
```

Compilons-le et lançons-le :

```
$ gcc -O0 inl-04.c
$ ./a.out
s=630
```

Oups ! Petite surprise... Où est l'erreur ? Voyons d'abord quelle est la répartition des registres effectuée par `gcc`. L'utilitaire `objdump` va nous y aider (nous montrons seulement la partie qui nous intéresse) :

```
$ objdump -d -j .text a.out
000000000040053c <main>:
.............................
  40054b: 48 c7 45 e0 02 00 00  movq   $0x2,-0x20(%rbp)
  400552: 00
  400553: 48 c7 45 e8 03 00 00  movq   $0x3,-0x18(%rbp)
  40055a: 00
  40055b: 48 c7 45 f0 05 00 00  movq   $0x5,-0x10(%rbp)
  400562: 00
  400563:       48 8b 45 e0             mov    -0x20(%rbp),%rax
  400567:       48 8b 55 e8             mov    -0x18(%rbp),%rdx
  40056b:       48 8b 4d f0             mov    -0x10(%rbp),%rcx
  40056f:       48 89 c0                mov    %rax,%rax
  400572:       48 01 d0                add    %rdx,%rax
  400575:       48 f7 e0                mul    %rax
  400578:       48 f7 e0                mul    %rax
  40057b:       48 01 c8                add    %rcx,%rax
  40057e:       48 89 45 f8             mov    %rax,-0x8(%rbp)
  400582:       b8 9c 06 40 00          mov    $0x40069c,%eax
  400587:       48 8b 55 f8             mov    -0x8(%rbp),%rdx
  40058b:       48 89 d6                mov    %rdx,%rsi
  40058e:       48 89 c7                mov    %rax,%rdi
  400591:       b8 00 00 00 00          mov    $0x0,%eax
  400596:       e8 8d fe ff ff          callq  400428 <printf@plt>
  40059b:       bf 00 00 00 00          mov    $0x0,%edi
  4005a0:       e8 93 fe ff ff          callq  400438 <exit@plt>
.............................
```

Question.- Quels sont les registres attribués aux variables `x`, `y`, `z` et `s` respectivement ?

Question.- Quelle est la suite des opérations effectués par le programme durant l'exécution ?

Le résultat erroné provient du fait que le même registre `%rax` a été attribué aux deux variables `x` et `s`, parce que, pour `gcc`, cela optimise le code final. Dans le cas précédent, `x` était utilisé une seule fois et le calcul était correct, mais ce n'est plus le cas ici parce que `s` est indiquée comme variable de sortie alors qu'elle est également utilisée dans son avatar de `x` en tant que variable d'entrée.

La solution consiste à spécifier explicitement `s` comme variable d'entrée et de sortie, en lecture et écriture, ce qui interdit de l'identifier avec `x`. La syntaxe du langage d'assemblage étendu permet de le faire (`inl-05.c`) :

```
void main(void)
  {
  long long int x=2,y=3,z=5,s;
  /*  s=(x+y)*x*x+z; */
  asm(
      "movq %1, %0\n\t"
      "addq %2, %0\n\t"
      "mulq %1\n\t"
      "mulq %1\n\t"
      "addq %3, %0\n\t"
      : "+r" (s)
      : "r" (x), "r" (y), "r" (z)
     );
  printf("s=%d\n",s);
  exit(0);
  }
```

void main(void) mais exit

Est-ce que maintenant tout va bien ? Regardons...

```
$ gcc -O0 inl-05.c
$ ./a.out
s=5
```

Oups ! Encore un problème... Vérifions, comme ci-dessus, la répartition des registres :

```
$ objdump -d -j .text a.out
000000000040053c <main>:
.............................
  40054c: 48 c7 45 d0 02 00 00  movq   $0x2,-0x30(%rbp)
  400553: 00
  400554: 48 c7 45 d8 03 00 00  movq   $0x3,-0x28(%rbp)
  40055b: 00
  40055c: 48 c7 45 e0 05 00 00  movq   $0x5,-0x20(%rbp)
  400563: 00
  400564: 48 8b 55 d0           mov    -0x30(%rbp),%rdx
  400568: 48 8b 4d d8           mov    -0x28(%rbp),%rcx
  40056c: 48 8b 5d e0           mov    -0x20(%rbp),%rbx
  400570: 48 8b 45 e8           mov    -0x18(%rbp),%rax
  400574: 48 89 d0              mov    %rdx,%rax
  400577: 48 01 c8              add    %rcx,%rax
  40057a: 48 f7 e2              mul    %rdx
  40057d: 48 f7 e2              mul    %rdx
  400580: 48 01 d8              add    %rbx,%rax
  400583: 48 89 45 e8           mov    %rax,-0x18(%rbp)
  400587: b8 9c 06 40 00        mov    $0x40069c,%eax
  40058c: 48 8b 55 e8           mov    -0x18(%rbp),%rdx
  400590: 48 89 d6              mov    %rdx,%rsi
  400593: 48 89 c7              mov    %rax,%rdi
  400596: b8 00 00 00 00        mov    $0x0,%eax
  40059b: e8 88 fe ff ff        callq  400428 <printf@plt>
  4005a0: bf 00 00 00 00        mov    $0x0,%edi
  4005a5: e8 8e fe ff ff        callq  400438 <exit@plt>
  4005aa: 90                    nop
.....................................
```

Question.- Quels sont les registres attribués aux variables `x`, `y`, `z` et `s` respectivement ?

Question.- Quelle est la suite des opérations effectuées par le programme durant l'exécution ?

Au premier coup d'œil le code ci-dessus ne contient rien d'anormal mais rappelons-nous la sémantique de l'instruction `mulq` : elle place le mot de poids fort dans le registre `%rdx`, or il a été attribué à la variable `x` par `gcc`. Lors de la première multiplication (ligne `40057a`), `%rdx` prend la valeur `0` et la deuxième multiplication (ligne `40057d`) met `s` à zéro. Cela explique le résultat obtenu ci-dessus.

Comment réparer le programme ? La solution est évidente : il faut placer `%rdx` dans la liste des registres « gâtés », ce qui empêche de l'attribuer à une variable « visible » du programme en question. Nous obtenons `inl-06.c` :

```
void main(void)
  {
  long long int x=2,y=3,z=5,s;
  /*  s=(x+y)*x*x+z; */
  asm(
      "movq %1, %0\n\t"
      "addq %2, %0\n\t"
      "mulq %1\n\t"
      "mulq %1\n\t"
      "addq %3, %0\n\t"
      : "+r" (s)
      : "r" (x), "r" (y), "r" (z)
      : "%rdx"
     );
  printf("s=%d\n",s);
  exit(0);
  }
```

```
$ gcc -O0 inl-06.c
$ ./a.out
s=25
```

Maintenant le résultat est enfin correct...

Question.- Que se passe-t-il si, dans le programme ci-dessus, les variables `x`, `y`, `z` et `s` sont déclarées comme `int` à la place de `long long int` ? Vérifiez et expliquez ce que vous observez !

4.2.3 Attribution des registres par l'utilisateur

Syntaxe.- L'utilisateur peut indiquer lui-même la correspondance désirée entre les variables du programme en langage C et les registres du processeur dans les deux listes : celle des variables de sortie et celle des autres variables. Pour cela, il faut :

- remplacer dans chaque élément `"..r" (var)` la lettre 'r' par le nom symbolique désiré du registre pour la variable `var` ;
- utiliser dans le code en langage d'assemblage les « vrais » noms de registres, mais préfixés par '`%`', par exemple `%%rax` pour `%rax`.

Les noms symboliques des registres sont :

Registre	nom qui remplace `r`
`%rax, %eax, %ax, %al`	`a`
`%rbx, %ebx, %bx, %bl`	`b`
`%rcx, %ecx, %cx, %cl`	`c`
`%rdx, %edx, %dx, %dl`	`d`
`%rsi, %esi, %si`	`S`
`%rdi, %edi, %di`	`D`

Remarque.- Les noms symboliques pour les registres `%r8 - %r15` n'existent pas, tout au moins on ne les retrouve pas dans la documentation de `gcc`.

Question.- Est-il possible que les registres `%r8 - %r15` soient attribués à certaines variables ? Dans quelles conditions peuvent-ils quand même apparaître dans le code obtenu selon une des méthodes ci-dessus ?

Exemple.- Réécrivons le programme `inl-06.c` avec la deuxième méthode. Notons que le registre `%rdx` doit obligatoirement appartenir à la liste des registres « gâtés » parce que le code contient l'instruction `mulq`. Supposons que le programmeur décide d'attribuer :

- le registre `%rax` à la variable `s` ;
- le registre `%rbx` à la variable `x` ;
- le registre `%rcx` à la variable `y` ;
- le registre `%rsi` à la variable `z`.

Le programme qu'on obtient est alors (`inl-07.c`) :

```
void main(void)
  {
  long long int x=2,y=3,z=5,s;
  /*  s=(x+y)*x*x+z; */
  asm(
      "movq %%rbx, %%rax\n\t"
      "addq %%rcx, %%rax\n\t"
      "mulq %%rbx\n\t"
      "mulq %%rbx\n\t"
      "addq %%rsi, %%rax\n\t"
      : "+a" (s)
      : "b" (x), "c" (y), "S" (z)
      : "%rdx"
     );
  printf("s=%d\n",s);
  exit(0);
  }
$ gcc -O0 inl-06.c
$ ./a.out
s=25
```

4.3 Fonction encapsulant du code en langage d'assemblage

La pratique raisonnable de l'utilisation de code en langage d'assemblage consiste à encapsuler ce code dans une fonction et d'appeler ensuite cette fonction dans tel ou tel programme en langage C ou bien dans un autre langage de haut niveau. Montrons la mise en œuvre de cette pratique dans l'exemple suivant.

Rappel.- Le registre `tsc` est un registre de 64 bits réinitialisé au moment du démarrage de la machine, et incrémenté à chaque top d'horloge.

Exemple.- Écrivons une fonction `tsc_value()` qui renvoie la valeur de `tsc` lorsqu'elle est appelée.

Soient deux variables `lo` et `hi` qui contiendront les contenus des registres `%eax` et `%edx` respectivement. Alors, on obtient :

```
long long unsigned int tsc_value(void)
    {
    static unsigned int hi=0,lo=0;
       asm (
            "xorl %%eax,%%eax\n\t"
            "xorl %%edx,%%edx\n\t"
            "rdtsc"
            : "=a" (lo), "=d" (hi)
           );
       return((long long unsigned int) (hi<<32)|lo);
    }
```

Exercice.- Écrire un programme en langage C permettant d'estimer le nombre de tops d'horloge nécessaires pour exécuter le code de la fonction `tsc_value()`.

Exercice.- Soit une fonction `fndbyte(unsigned char *ptr,unsigned char patt)` qui, étant donnée l'adresse de début d'une suite d'octets quelconques (terminée par l'octet '\0') et un octet donné, compte le nombre d'occurrences de ce dernier octet dans la suite. Écrire deux versions de cette fonction : l'une en langage C pur et l'autre en utilisant le langage d'assemblage pour la partie de cette fonction paraissant convenable. Utiliser la fonction `tsc_value()` ci-dessus et comparer l'efficacité des deux versions pour des suites d'octets dont la longueur est ∼100 MiO. Qui gagne ?