L'assembleur en ligne

Avec le langage C et le compilateur GCC

Par Issam Abdallah

Date de publication : 28 juin 2014

Dernière mise à jour : 2 septembre 2014

CONFIRMÉ

Ce tutoriel va vous présenter l'assembleur en ligne avec le langage C et le compilateur *GCC*.

Commentez ♪

I - Introduction

Aujourd'hui, le langage assembleur est assez peu utilisé. La plupart des programmeurs utilisent les langages haut niveau, comme le C ou le C++, pour plusieurs raisons. En premier lieu, parce que ces langages permettent d'écrire des programmes indépendants de l'architecture, et donc portables. En second lieu, parce qu'ils présentent des syntaxes simples et compréhensibles, ce qui permet d'augmenter la productivité, lors de l'écriture et la maintenance du code. Mais, de temps en temps, les programmeurs ont besoin d'utiliser des instructions assembleur dans leurs programmes.

L'assembleur en ligne, *Inline Assembly*, est une extension des langages de programmation haut niveau standard offerte par certains compilateurs. Il permet d'inclure des instructions assembleur dans un programme écrit en langage haut niveau.

Ce tutoriel va vous présenter l'assembleur en ligne avec le langage C et le compilateur *GCC*. Les différents exemples cités sont téléchargeables sur **Developpez.com**.

I-A - Remerciements

Merci à **Cédric Duprez** et **f-leb** pour leur relecture minutieuse.

II - Quand utiliser l'assembleur en ligne ?

L'ajout des instructions assembleur dépendantes de l'architecture à un programme écrit dans un langage haut niveau affecte sa portabilité. Ainsi, vous ne devriez utiliser des instructions assembleur qu'en dernier ressort. Par exemple, lorsque vous constatez que l'utilisation de l'assembleur va optimiser vos codes et vous produira un programme rapide, ou lorsque vous voulez utiliser des instructions spécifiques à l'architecture et non descriptibles par une syntaxe de haut niveau.

En fait, la capacité d'optimisation des compilateurs est limitée à cause de plusieurs facteurs, tels que leur compréhension limitée du comportement du code, et le contexte dans lequel il sera utilisé, et aussi le besoin des programmeurs d'effectuer la compilation rapidement. En effet, la tâche d'optimisation est coûteuse en mémoire et en temps de compilation. Ainsi, la substitution des parties complexes du code et sensibles à la performance du programme par des simples instructions assembleur va simplifier l'optimisation d'une part, et permettra de produire un programme performant et rapide, d'autre part.

En outre, plusieurs fonctionnalités des processeurs sont encore difficiles à décrire par une syntaxe de haut niveau. Ainsi, plusieurs instructions machine doivent être écrites à la main. Parmi celles-ci, on cite les instructions système d'accès aux ports d'entrée/sortie (*x86* : instructions *OUT*, *OUTS*, *IN*, *INS*). On peut aussi citer les instructions de gestion de la mémoire (*x86* : *LLDT*, *LGDT*…) et des tâches (*x86* : *LTR*).

L'utilisation des instructions assembleur nécessite une connaissance avancée de l'architecture et du fonctionnement interne du compilateur. En effet, l'assembleur en ligne est largement utilisé par les développeurs des systèmes d'exploitation et des pilotes de périphériques (*drivers*).

Dans ces domaines de programmation, la portabilité d'un code n'est pas une exigence extrême et n'a parfois pas de sens. En effet, le code source d'un driver ou d'un système d'exploitation ne pourra jamais être portable. La haute priorité est, souvent, donnée à la production d'un programme performant qui s'exécute rapidement et qui permet d'exploiter efficacement toutes les fonctionnalités d'une architecture donnée.

Pour les systèmes d'exploitation, le terme **multiplate-forme** est plus précis. Par exemple, le système Linux est multiplate-forme parce qu'il peut tourner sur une variété d'architectures. En fait, dans le code source de son noyau, chaque architecture possède sa propre description en langage C et en assembleur (*/usr/src/linux/arch/*). Cela vient du fait que les instructions et les caractéristiques diffèrent d'une architecture à une autre. Autrement dit, le code source dépendant d'une architecture ne peut pas être compilé pour tourner sur une autre.

Les instructions assembleur sont souvent encapsulées dans des macros ou des fonctions en ligne définies dans des fichiers d'en-tête. C'est pour faciliter la maintenance du code source et pour faciliter son portage d'une architecture à une autre. Consultez le répertoire */usr/src/linux/arch/x86/include/asm* pour avoir accès à des exemples de code contenant des instructions assembleur *x86*. Par exemple, le fichier *io.h* contient les instructions *x86* d'accès aux ports d'entrée/sortie. En outre, le fichier *string.h* contient l'implémentation des opérations de manipulation des chaînes en utilisant le jeu d'instructions *x86*.

Dans ce tutoriel, on va étudier quelques exemples des codes assembleur pris du code source du noyau *Linux 0.01*. Vous pouvez le télécharger sur **https://www.kernel.org/pub/linux/kernel/Historic**.

III - L'assembleur en ligne avec le langage C

En particulier, le compilateur *GCC, GNU Compilers Collection* dispose d'une syntaxe simple et complète permettant d'utiliser efficacement des instructions assembleur dans un programme C.

Dans un programme assembleur classique, chaque instruction possède, typiquement, deux opérandes : un opérande source et un opérande destination. Ces opérandes peuvent être utilisés de façon explicite ou implicite. On peut citer comme exemple l'instruction *x86 MOV*, qui utilise explicitement deux opérandes. En utilisant la syntaxe *AT&T*, on peut écrire cette instruction comme suit :

```
MOV source, destination
```

Les opérandes peuvent être de trois types : registre, mémoire ou immédiat. Dans l'exemple ci-dessous, l'instruction *MOV* va transférer le contenu du registre *EDX* dans *EAX* :

```
MOV EDX, EAX
```

Avec l'assembleur en ligne, on ne va pas réinventer la roue, croyez-moi ! Les instructions seront écrites de la même façon, sauf que les opérandes seront définis séparément et leurs valeurs pourront être des expressions C. Autrement dit, tout ce que vous avez appris sur l'assembleur restera valide et vous en aurez besoin. **Vous allez juste apprendre une syntaxe qui vous permettra d'écrire des instructions assembleur dans un programme C**.

Avec le compilateur *GCC*, les instructions assembleur, ainsi que la définition de leurs opérandes, doivent être encapsulées dans une construction déclarée, typiquement, avec le mot clé ***asm***. Cela va indiquer au compilateur que le code à l'intérieur doit être traité d'une façon différente. Pendant la compilation, le compilateur *GCC* va utiliser les informations incluses dans la construction *asm* pour générer les instructions assembleur et les placer dans le code-cible.

Notez bien que GCC utilise par défaut la syntaxe *AT&T* pour générer le code-cible. En effet, le compilateur utilise, par défaut, le programme *as* (*GNU Assembler*) pour générer le code objet. Ainsi, dans la construction *asm*, les instructions assembleur doivent être écrites en utilisant cette syntaxe, à moins que vous n'utilisiez l'option *-masm* du compilateur. Dans ce qui suit, on va utiliser la syntaxe AT&T. Voici un tutoriel qui va vous aider à l'apprendre rapidement :

http://asm.developpez.com/cours/gas/

IV - Syntaxe de la construction asm

Le mot clé *asm* doit être suivi par une expression entre parenthèses et constituée de sections séparées par deux points. La première section contient des instructions assembleur écrites entre guillemets. Dans la deuxième, on doit spécifier les opérandes de sortie des instructions. La troisième section contient les opérandes d'entrée. La quatrième section sert à déclarer les modifications apportées, sur les registres ou en mémoire, par les instructions.

***Important** : vous devez toujours placer les deux points qui séparent la deuxième section de la troisième, même si la section de sorties est vide.*

Le programme suivant utilise l'instruction assembleur *movl* pour affecter la valeur de la variable *b* à variable *a* (*a = b ;*) :

Listing 1 : movl.c

```
#include <stdio.h>
int main(void)
{
 int a = 10;
 int b = 5;

 __asm__("movl\t%1, %0"

         : "=&r" (a) : "r" (b)
         : /* liste des modifications */
        );

 printf("a = b = %d \n", a);
 return 0;
}
```

Ici, chaque opérande est écrit comme une expression C placée entre parenthèses et précédée d'une chaîne de caractères indiquant sa contrainte. La lettre *r* dans chaque contrainte symbolise le nom d'un registre général du processeur. Ainsi, le compilateur *GCC* va allouer un registre de ce type, pour stocker la valeur (5) de la variable *b*, et un autre pour stocker celle (10) de *a*. Le caractère « = » dans la contrainte « =r » du premier opérande indique que c'est un opérande de sortie. Toutes les contraintes des opérandes de sortie doivent commencer par « = ».

***Note** : le caractère « & » dans la première contrainte va indiquer au compilateur de ne pas allouer le même registre pour les deux opérandes. L'utilisation de ce caractère dans l'exemple ci-dessus n'est pas obligatoire, mais dans certains cas il est très important. On va expliquer, en détail, son utilisation dans la section **IV.E Déclaration des modifications**.*

Pour se référer aux opérandes dans le code assembleur, on peut écrire leurs indices précédés du caractère « % ». Ainsi, l'instruction *movl* va transférer le contenu (5) du deuxième opérande (source ou entrée d'indice 1) dans le registre alloué au premier opérande (destination ou sortie d'indice 0).

***Important** : la production de sorties aura lieu, toujours, après l'exécution de la dernière instruction du code assembleur.*

Dans le code ci-dessus, après l'exécution de l'instruction *movl*, la sortie *a* sera produite. C'est en affectant le contenu du registre destination à la variable *a*.

GCC ne peut pas analyser les instructions assembleur et vérifier si elles sont valides ou non. Ainsi, il ne peut pas vérifier si le type des opérandes est raisonnable pour les instructions ou non. En fait, c'est le rôle de l'assembleur *as*. En ce qui concerne la construction *asm*, *GCC* va juste, dans un premier temps, vérifier la syntaxe des expressions C utilisées comme opérandes. Ensuite, il va générer les opérandes des instructions en utilisant les informations contenues dans les sections d'entrées et de sorties. Finalement, il va placer les instructions assembleur avec leurs opérandes dans le code-cible.

Enfin, pour obtenir le code-cible produit par le compilateur *GCC*, faites-lui passer l'option *-S*. Faites passer, également, l'option *-O2*, pour appliquer le deuxième niveau d'optimisation de *GCC* à votre code. Voici une portion du code-cible de l'exemple ci-dessus :

```
        movl    $5, %edx
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        andl    $-16, %esp
        subl    $16, %esp
#APP
# 9 "movl.c" 1
        movl    %edx, %eax
# 0 "" 2
#NO_APP
        movl    $.LC0, (%esp)
        movl    %eax, 4(%esp)
        call    printf
        xorl    %eax, %eax
        leave
```

Dans le code-cible, le bloc *asm* est placé entre les deux lignes de commentaires *#APP* et *#NO_APP*. Le compilateur a alloué deux registres généraux pour stocker les valeurs de *a* et *b* : le registre *EAX* pour *a* et le registre *EDX* pour *b*.

IV-A - Les instructions assembleur

La première section de la construction *asm* contient les instructions assembleur. Dans cette section, vous devez spécifier un modèle d'instructions assembleur, un peu comme ce qui apparaît dans une description machine (un fichier *.s*).

En utilisant la syntaxe *AT&T*, si vous voulez utiliser le nom d'un registre comme opérande, préfixez-le par « *%%* ». Pour se référer à un opérande d'entrée ou de sortie, dans une instruction assembleur, écrivez le caractère « *%* » suivi de son indice (0, 1… 29).

Comme dans un code assembleur classique, les instructions de *asm* doivent être séparées par des séparateurs. En assembleur GNU, on a le choix d'utiliser soit le caractère « *;* », soit le caractère saut de ligne. Si vous choisissez ce dernier, placez « *\n* » après chaque instruction, sauf la dernière.

Notez que vous pouvez mettre plusieurs instructions dans la même chaîne, en les séparant par des « *;* », comme suit :

```
__asm__("\tnop;nop;nop\t"
        "jmp \t1f\n"
        "1:" ::);
```

Le « *\n* » sera traduit en un saut de ligne dans le code-cible. Le caractère « *\t* » est optionnel et il sera traduit en une tabulation.

Pendant la compilation, *GCC* utilise les informations (sur les opérandes) contenues dans les sections d'entrées et de sorties pour générer les opérandes des instructions assembleur.

IV-B - Les opérandes

Chaque opérande peut être écrit comme une expression C placée entre parenthèses et précédée d'une chaîne de caractères indiquant sa contrainte. En général, les instructions assembleur utilisent trois types d'opérandes : les opérandes registre, les opérandes mémoire et les opérandes immédiats. Dans la construction *asm*, les contraintes vont préciser si un opérande doit être stocké dans un registre, et quel type de registre ; si l'opérande doit être une référence dans la mémoire, et quel type d'adresse utilisé ; si l'opérande est une constante immédiate, et quelle valeur elle peut avoir. Une contrainte peut être constituée d'un ou plusieurs modificateurs et lettres.

IV-B-1 - Les lettres

Les lettres spécifient souvent les registres d'une architecture donnée. Le tableau suivant liste les lettres couramment utilisées avec l'architecture *x86* :

Lettre	Registre
R	*EAX*, *EBX*, *ECX*, *EDX*, *ESI*, *EDI*, *ESP* et *EBP*
q	*EAX*, *EBX*, *ECX*, *EDX* (mode 32 bits)
a	Registre *EAX*
b	Registre *EBX*
c	Registre *ECX*
d	Registre *EDX*
S	Registre *ESI*
D	Registre *EDI*
A	Combinaison *EAX:EDX*

Il existe aussi un ensemble de lettres commun à toutes les architectures. Consultez le manuel de *GCC* (à **cette adresse**) pour avoir accès à la liste complète de contraintes.

Avec l'architecture *x86*, on utilise souvent la contrainte « *r* » ou « *R* » pour spécifier un opérande registre, la contrainte « *m* » pour spécifier un opérande mémoire et la contrainte « *i* » ou « *I* » pour spécifier un opérande immédiat.

En outre, un nombre (0, 1, 2… 9) peut être utilisé dans la contrainte d'un opérande d'entrée, pour dire que cet opérande fait référence à un opérande de sortie. Ainsi, le compilateur va allouer la même *location* pour stocker les valeurs des deux opérandes.

IV-B-2 - Les modificateurs

Pour les modificateurs, on a déjà vu le modificateur « = ». Il en existe cinq autres, parmi lesquels on cite les modificateurs « + » et « *&* ». Le premier indique que l'opérande est en écriture et en lecture à la fois. Le modificateur « *&* » a une utilisation spécifique, qu'on va expliquer dans les sections suivantes.

IV-C - Les sorties

La deuxième section de la construction *asm* contient les opérandes de sortie, séparés par des virgules. Chaque opérande doit être une expression C de type *lvalue*. C'est-à-dire qu'on peut le mettre à gauche d'une affectation. Par exemple, vous ne devez pas utiliser une constante déclarée avec la directive *#define* ou une variable C de type *const* comme opérande de sortie. Le compilateur vérifie cela pour chaque opérande de sortie.

Un opérande de sortie ordinaire est en écriture seulement. Ainsi, les contraintes de sortie contiennent, souvent, le modificateur « = ». La valeur d'un tel opérande reste indéterminée jusqu'à la production des sorties ! Vous pouvez vérifier cela dans l'exemple *movl.c*, en inversant l'ordre des opérandes de l'instruction comme suit :

```
__asm__("movl\t%0, %1"

        : "=&r" (a)
        : "r" (b)
        :/* liste des modifications */
       )
```

Ainsi, si vous décidez d'utiliser un opérande de sortie en lecture et en écriture, vous devez utiliser le modificateur « + » à la place de « = ».

IV-D - Les entrées

Les opérandes d'entrée occupent la troisième section de la construction *asm*. Ils sont traités de manière différente de celles de sortie, bien qu'ils aient la même syntaxe.

Contrairement aux opérandes de sortie, un opérande d'entrée est par défaut en lecture et en écriture (sans utiliser le « + »). Ainsi, vous pouvez spécifier une même *location* (registre ou mémoire) comme opérandes de sortie et d'entrée à la fois. La connexion entre les deux opérandes doit être décrite par une contrainte disant que les deux opérandes doivent occuper la même *location* à l'exécution de l'instruction correspondante. Notez qu'on peut utiliser la même expression C pour les deux opérandes, comme on peut utiliser deux expressions différentes. Pour illustrer, prenons l'exemple suivant :

```
__asm__( "addl %2, %1" : "=r" (a) : "0" (a), "r"(b));
```

La contrainte « 0 » dans le premier opérande d'entrée (d'indice 1) dit que l'expression C doit occuper le même registre général que celui spécifié dans l'opérande de sortie (d'indice 0).

***Important** : un nombre n'est autorisé comme contrainte que dans un opérande d'entrée et il doit se référer à un opérande de sortie.*

Seul un nombre dans une contrainte peut garantir que deux opérandes vont occuper la même *location*. L'utilisation de la même expression dans les deux opérandes ne suffit pas. Ainsi, le résultat du code suivant n'est pas fiable :

```
__asm__( "addl %2, %1" : "=r" (a) : "r" (a), "r"(b));
```

En fait, Le compilateur peut choisir deux registres généraux différents pour stocker les deux opérandes 0 et 1.

IV-E - Déclaration des modifications

Si une instruction modifie (explicitement ou implicitement) les valeurs d'un ou plusieurs registres, spécifiez ces registres dans la quatrième section de la construction *asm*. Les registres modifiés sont décrits dans des chaînes de caractères séparées par des virgules. Dans l'exemple suivant, la valeur du registre *EBX* est modifiée par l'instruction *movl*. Donc, on doit écrire son nom dans la section 4 de la construction *asm* :

Listing 2 : appel sys.c

```
#include<asm/unistd.h>                /* __NR_write */
#include<unistd.h>                    /* STDOUT_FILENO */
#include<string.h>                    /* strlen() */

#define STDOUT STDOUT_FILENO

int main(void)
{
 char msg[] = "hello!\n";
 int res;

 __asm__("movl \t%2, %%ebx\n\t"
         "int \t$0x80"

         :"=a" (res)
         :"0" (__NR_write), "I" (STDOUT),
          "c" (msg), "d" (strlen(msg))
         :"ebx");

return 0;
}
```

L'instruction assembleur *int* utilise, implicitement, les registres *EAX*, *EBX*, *ECX* et *EDX*. Le registre *ECX* contient l'adresse du premier caractère de la chaîne *msg*, et *EDX* contient sa taille. Le registre *EAX* est utilisé par l'instruction *int* comme opérandes d'entrée et de sortie à la fois. En tant qu'opérande d'entrée, il doit être initialisé avec la valeur numérique *__NR_write*. Ainsi, l'instruction *int* va générer l'appel système 4 (la fonction *write()* du fichier */usr/include/unistd.h*), pour afficher le message « hello ! » sur la console. La valeur de retour de l'appel système sera stockée dans le registre *EAX*, en tant qu'opérande de sortie.

***Important** : vous ne devez jamais écrire le nom d'un registre dans la section de déclaration des modifications, si ce registre fait partie d'un opérande d'entrée ou de sortie.*

En fait, un tel registre est, à l'avance, déclaré modifié. Dans l'exemple ci-dessus, les registres *ECX ("c")*, *EDX ("d")* et *EAX ("=a")* sont utilisés pour stocker les valeurs des opérandes d'entrée et de sortie. Donc, on n'a pas listé leurs noms dans la section 4 de la construction *asm*.

En utilisant les informations sur la modification des registres, le compilateur détermine les valeurs qui doivent être sauvegardées dans la pile et restaurées après l'exécution du bloc *asm*.

IV-E-1 - Modification de la mémoire : « memory »

Si vos instructions assembleur accèdent à la mémoire d'une manière imprévisible et arbitraire, ajoutez « memory » à la liste des modifications. Cela va renseigner à *GCC* de ne pas maintenir les valeurs (destinées à être chargées dans la mémoire) stockées dans des registres et de ne pas optimiser l'accès à la mémoire, durant l'exécution des instructions. D'autre part, si vous connaissez la taille de la mémoire accédée, ajoutez-la comme entrée, sinon utilisez « memory ».

IV-E-2 - Modification du registre code condition : « cc »

Si vos instructions assembleur peuvent modifier le registre de code condition *cc* d'une manière inhabituelle, ajoutez « cc »à la liste de modifications. Avec l'architecture *x86*, la notation *cc* est utilisée par *GCC* pour représenter le registre des indicateurs *EFLAGS*.

Certaines instructions de l'architecture *x86*, telles que celles de décalage et de rotation logique ou arithmétique, peuvent altérer quelques bits du registre *EFLAGS* pendant leur exécution. Pour illustrer, soit l'exemple suivant :

Listing 3 : shll.c
```
__asm__( "shll $2, %1" : "=r" (res) : "r" (a) : "cc");
```

Dans l'exemple ci-dessus, l'instruction *shll* (*Logical Left Shift*) décale les bits du registre deux positions vers la gauche. Le dernier bit (de gauche) décalé est transféré dans le bit *CF* (*Carry Flag*). Ainsi, le registre code condition (*EFLAGS*) est toujours modifié avant qu'on puisse le tester ! Donc, « cc » doit être placé dans la section 4 pour renseigner le compilateur. Par contre, le bit *OF* (*Overflow Flag*) sera modifié automatiquement par le processeur comme résultat de l'exécution de *shll*, ce qui n'est pas pris en compte par le compilateur comme modification du registre de code condition *cc*.

Les instructions de test *TEST*, de comparaison *CMP*, ainsi que les instructions de contrôle des indicateurs comme *CLD* et *STD* n'ont pas d'opérandes de sortie (comme *shll*). Ainsi, ils ne sont pas considérés par le compilateur comme modificateurs de registre *cc*.

IV-E-3 - Le modificateur « & »

À moins qu'un opérande de sortie contienne « & » dans sa contrainte, *GCC* peut le stocker dans le même registre alloué à un opérande d'entrée qui ne lui fait pas référence. C'est parce que le compilateur *GCC* suppose toujours que

les instructions assembleur finissent l'utilisation des opérandes d'entrée avant que les sorties soient produites. Or, dans certains cas, cette supposition est fausse, ce qui peut provoquer des problèmes. Ainsi, le programme suivant n'est pas fiable :

Listing 4 : addition2.c

```
__asm__("movl %1, %%eax\n\t"
        "addl %2, %%eax\n\t"
        "subl %1, %1"
        :"=a" (res)
        :"r" (a), "b" (b));
```

Si le compilateur alloue le registre *EAX* au premier opérande d'entrée, le résultat de l'addition (contenu du registre *EAX*) sera modifié avant son chargement dans *res*. C'est parce que la production de sortie *res* aura lieu juste après l'exécution de l'instruction *subl* ! Voici une portion du code-cible du programme ci-dessus :

```
        movl        $3, %edx
        movl        $10, %eax
#APP
# 9 "addition2.c" 1
        movl         %eax, %eax
        addl         %edx, %eax
        subl         %eax, %eax
# 0 "" 2
#NO_APP
```

Seule l'utilisation du modificateur « & », dans la contrainte de l'opérande de sortie, va prévenir l'allocation du registre *EAX* à l'opérande d'entrée.

Dans l'exemple ci-dessus, l'utilisation de « 0 » comme contrainte du premier opérande d'entrée n'est pas correcte. C'est parce qu'il est utilisé par l'instruction *subl*, en tant qu'opérande d'entrée, avant la production de la sortie *res*. Par contre, dans l'exemple suivant, l'utilisation de la contrainte « 0 » est correcte :

Listing 5 : multiplication.c

```
#include <stdio.h>

int main(void)
{
 int x = 10;              /* EAX */
 int mult = 3;
 int res;                 /* EAX */

 __asm__("movl %2, %%ebx\n\t"
         "imul %%ebx"
         :"=&a" (res)
         :"0" (x), "r" (mult)
         :"ebx");

 printf("%d * %d = %d \n", x, mult, res);
 return 0;
}
```

L'instruction *imul* utilise implicitement le registre *EAX* pour stocker le nombre à multiplier (comme entrée) et pour stocker le résultat de la multiplication. Ainsi, le premier opérande d'entrée est utilisé seulement avant la production de la sortie *res*. Voici un autre exemple montrant l'importance du modificateur « & » dans quelques cas :

Listing 6 : division.c

```
#include <stdio.h>

int main(void)
{
 int x = 10;            /* EAX */
 int divs = 3;
 int quot;              /* EAX */
 int reste;             /* EDX */
```

Listing 6 : division.c

```
__asm__("subl %%edx, %%edx\n\t"
        "movl %3, %%ebx\n\t"
        "idivl %%ebx"

        :"=a" (quot), "=&d" (reste)
        :"0" (x), "r" (divs)
        :"ebx");

printf("%d / %d = %d \n", x, divs, quot);
printf("%d %% %d = %d \n", x, divs, reste);
return 0;
}
```

L'instruction *idivl* utilise implicitement le registre *EAX* pour stocker le dividende (*x*) et le quotient (*quot*). Le registre *EDX* sera utilisé implicitement pour stocker le reste (*reste*) de la division. Ainsi, le registre *EDX* doit être indiqué dans la deuxième contrainte de sortie.

Supposons qu'on n'ait pas utilisé l'identificateur « & » dans la deuxième contrainte. Dans ce cas, le compilateur peut allouer le registre *EDX* pour stocker le diviseur (*divs*). Et par conséquent, l'exécution du programme peut provoquer une exception de type division par 0 !

Ainsi, lorsque le nom d'un registre est indiqué comme contrainte de sortie, il vaut mieux d'utiliser le modificateur « & ».

V - Problèmes d'optimisation

Durant l'optimisation, *GCC* tente de réordonner et de réécrire le code du programme, même en présence de la construction *asm*. Si les opérandes de sortie ne sont pas utilisés (la section 2 de *asm* est vide), ou si leurs valeurs ne sont pas modifiées par les instructions, l'optimiseur considère que les instructions n'ont pas un effet de bord dans le programme. Ainsi, la construction *asm* peut être supprimée. Pour bien comprendre le problème, on va étudier la portion de code suivante, prise du fichier *include/string.h* du code source du noyau *Linux-0.01*) :

```
extern inline char *strcpy(char *dest,const char *src)
{

__asm__("cld\n"
        "1: lodsb\n\t"
        "stosb\n\t"
        "testb %%al,%%al\n\t"
        "jne 1b"
        ::"S" (src), "D" (dest)
        :"ax", "memory");
return dest;
}
```

La construction *asm* du code ci-dessus ne possède pas d'opérande de sortie. Pour le moment, on n'a pas besoin d'utiliser de sortie puisque les instructions *lodsb* et *stosb* utilisent directement la mémoire. Ainsi, on n'a pas de résultat à récupérer depuis un registre dans la mémoire à la fin de l'exécution du bloc *asm*. Mais l'optimiseur est parfois fou ! L'optimisation risquera de supprimer la construction *asm*. Et, par conséquent, la fonction *strcpy()* devient inutilisable.

Vous pouvez empêcher la construction *asm* d'être supprimée, par l'utilisation du mot-clé *volatile*, qui va indiquer au compilateur que les instructions ont un effet de bord important. *GCC* ne supprime jamais un *asm* volatile. Mais il peut le déplacer dans le code. Pour éviter ça, vous devez toujours spécifier tout opérande modifié dans le code assembleur, comme opérande de sortie. Ainsi :

```
extern inline char *strcpy(char *dest,const char *src)
{
int S, D, A;

__asm__ __volatile__(
        "cld \n"
```

```
          "1:lodsb \n\t"
          "stosb \n\t"
          "testb %%al, %%al \n\t"
          "jne 1b"

          :"=&S (S)", "=&D (D)", "=&a (A)"
          :"0" (src),"1" (dest), "2" (0));
 return dest;
}
```

Dans l'exemple ci-dessus, le contenu des registres *ESI*, *EDI* et *EAX* a été modifié. Ainsi, on les a spécifiés comme opérandes de sortie. Les variables *S*, *D* et *A* déclarées dans le code C sont utilisées juste pour créer une dépendance avec le bloc *asm*. Ainsi, l'optimiseur ne déplacera pas ce dernier.

VI - Utilisation des macros

Si vous décidez d'utiliser des instructions assembleur dépendantes de l'architecture, il vaut mieux pour vous les encapsuler dans des macros et les placer dans un fichier d'en-tête. Cela peut vous aider à la maintenance de vos programmes. Ainsi, si vous décidez de porter un programme vers une autre architecture, vous n'aurez besoin de réécrire qu'un seul fichier. Dans l'exemple *max.c*, on a encapsulé la construction *asm* dans la macro *max(a,b)* définie dans le fichier d'en-tête *max.h* :

Listing 7 : max.h

```
#ifndef MAX_H
#define MAX_H

#define max(a,b)                        \
({                                      \
 int __res, __x = (a), __y = (b);       \
 __asm__ __volatile__(                  \
        "cmpl %1, %2\n\t"               \
        "jge 1f\n\t"                    \
        "movl %1, %0\n\t"               \
        "jmp 2f\n\t"                    \
        "1: movl %2, %0\n\t"            \
        "2:"                            \
        :"=r" (__res)                   \
        :"r" (__x), "r" (__y));         \
 __res;                                 \
})

#endif
```

Notez que la dernière instruction d'une instruction composée, écrite entre « {} », doit être une expression suivie par « ; ». Elle sert à donner une valeur à l'instruction entière. Dans notre exemple, on veut récupérer la valeur de l'opérande de sortie dans le code C, pour l'afficher par exemple. Ainsi, on a écrit *__res ;* à la fin de l'instruction composée.

Dans la définition de la macro *max(a,b)*, les variables *__x* et *__y* sont utilisées pour s'assurer que la construction *asm* opère sur des valeurs entières. Une autre méthode pour faire en sorte que la construction *asm* opère sur le type de données correct est l'utilisation de forçage de type (*casting*) dans les entrées. Dans l'exemple ci-dessus, on peut forcer l'utilisation du type entier dans les opérandes d'entrée de la construction *asm* comme suit :

```
 (::"r" ((int)a), "r" ((int)b);
```

Pour le même objectif, les constructions *asm* peuvent être encapsulées dans des fonctions en ligne. La section suivante en donne un exemple.

VII - Un exemple détaillé

VII-A - Opérations sur les chaînes de caractères avec l'architecture x86

En bref, les instructions *x86* de manipulation des chaînes des caractères, comme *lodsb* et *stosb*, utilisent implicitement les registres *ESI* et *EDI*. *ESI* doit contenir l'adresse, dans le segment *DS*, de la chaîne source, et *EDI* doit contenir celle, dans le segment *ES*, de la chaîne de destination. Le registre *EAX* est utilisé implicitement par ces instructions pour stocker temporairement les données traitées. L'instruction *cld* va mettre à 0 le bit *DF* (*Direction Flag*) du registre *EFLAGS*. Ainsi, *ESI* et *EDI* seront incrémentés par le processeur durant l'exécution des instructions. *A contrario*, l'instruction *std* est utilisée pour mettre à 1 le bit *DF*, et ainsi *ESI* et *EDI* seront décrémentés.

Le programme *string.c* utilise les deux fonctions en ligne *my_strcpy* et *my_strcmp* définies dans le fichier *string.h*. La première fonction va initialiser une chaîne de caractères ; la deuxième va comparer deux chaînes initialisées et retourner 0 si elles sont égales, sinon elle retourne -1 ou 1.

Listing 8 : string.c

```
#include <stdio.h>
#include <stdlib.h>
#include "string.h"

#define  TAILLE 10

int main (void)
{
 char * str1 = (char*)
        malloc(TAILLE*sizeof(char));
 char * str2 = (char*)
        malloc(TAILLE*sizeof(char));

 if(!str1 || !str2)
        return EXIT_FAILURE;

 my_strcpy((char*)"foo", str1);
 my_strcpy((char*)"bar", str2);

 if(!my_strcmp(str1, str2))
        printf("Les deux chaines \"%s\" et \"%s\" "\
          "sont egales.\n", str1, str2);
 else
        printf("Les deux chaines \"%s\" et \"%s\" "\
          "ne sont pas egales.\n", str1, str2);

 free (str1);
 free (str2);

 return EXIT_SUCCESS;
}
```

VII-B - La fonction my_strcpy

```
static inline void
my_strcpy(char * src, char * dest)
{
 int S, D, A;

 __asm__ __volatile__ (
        "cld\n\t"                   /* ESI++, EDI++ */
        "1:\tlodsb\n\t"             /* MOVB DS:ESI, AL */
        "stosb\n\t"                 /* MOVB AL, ES:EDI */
        "testb\t%%al,%%al\n\t"      /* ZF=1 si AL == 0 */
        "jne\t1b"                   /* JMP si ZF == 0 */

        :"=&S" (S),"=&D" (D), "=&a" (A)
        :"0" (src),"1" (dest), "2" (0)
```

```
        :"memory");
}
```

Dans le code assembleur, on a utilisé l'instruction *cld*. Donc, *ESI* et *EDI* doivent contenir les adresses *src* et *dest* du premier caractère de chaque chaîne.

En outre, les instructions *lodsb* et *stosb* accèdent à la mémoire (*src* et *dest*) d'une manière imprévisible. En effet, on ne connaît pas à l'avance le nombre d'octets (caractères) à copier. Donc, on a utilisé « memory ».

D'autre part, les trois registres *ESI*, *EDI* et *EAX* seront modifiés par les instructions assembleur, donc on les a spécifiés comme opérandes de sortie. Ainsi, on doit utiliser le modificateur « & » pour prévenir l'allocation de ces registres à une entrée non correcte. La connexion entre les entrées et les sorties aura lieu avec l'utilisation des contraintes *« 0 »*, *« 1 »* et *« 2 »*. Cela est autorisé parce que les sorties *S*, *D* et *A* seront produites après la consommation des entrées par les instructions. Elles sont utilisées juste pour créer une dépendance entre le code C et le bloc *asm* !

L'utilisation du mot clé *volatile* et des opérandes de sortie va prévenir la suppression ou le déplacement de la construction *asm* pendant l'optimisation.

VII-C - La fonction my_strcmp

La fonction *my_strcmp* est aussi extraite du fichier *include/string.h* du code source du noyau *Linux-0.01*. Son fonctionnement est similaire à celui de la fonction *strcmp* de la bibliothèque C standard, définie dans le fichier */usr/include/string.h*.

```
static inline int
my_strcmp(const char * str1,const char * str2)
{
 int S, D, __res;

 __asm__ __volatile__(
        "cld\n"                 /* ESI++, EDI++ */
        "1:\tlodsb\n\t"         /* MOV DS:ESI, AL */
        "scasb\n\t"             /* SUB ES:ESI, AL */
        "jne 2f\n\t"            /* JMP si ZF == 0 (ES:EDI != AL) */
        "testb %%al,%%al\n\t"   /* ZF=1 si AL == 0 */
        "jne 1b\n\t"            /* JMP si ZF == 0 => il y a encore des caracteres */
                                /* pour comparer */
        "xorl %%eax,%%eax\n\t" /* (str1 == str2) => __res = 0 */
        "jmp 3f\n"              /* on a termine ! */
        "2:\tmovl $1,%%eax\n\t"
        "jl 3f\n\t"             /* (str1 != str2) et (str1[i] > str2[i]) => __res = 1 */
        "negl %%eax\n"          /* (str1 != str2) et (str1[i] < str2[i]) => __res = -1 */
        "3:"

        :"=&D" (S), "=&S" (S),"=&a" (__res)
        :"0" (str1),"1" (str2), "2" (0)
        :"memory");

 return __res;
}
```

VII-D - La compilation avec GCC

Voici le *makefile* utilisé pour compiler le programme *string.c* :

```
CFLAGS = -O2 -fomit-frame-pointer -W -Wall

string: string.o
        cc string.o -o string

string.o: string.c string.h
        cc -c $(CFLAGS) string.c -o string.o
```

```
clean:
	rm -rfv string.o
```

O2 est le niveau d'optimisation recommandé. Le compilateur va essayer d'augmenter les performances sans compromettre la taille et sans prendre trop de temps en compilation. Ce niveau d'optimisation permet de produire un code rapide. L'option *-fomit-frame-pointer* permet aussi de produire un code rapide et de taille réduite. Consultez le manuel de *GCC* pour avoir accès à encore plus d'informations.