Initiation au langage d’assemblage x86

Emmanuel Saracco
Easter-eggs.com

esaracco@easter-eggs.com
esaracco@free.fr

Initiation au langage d’assemblage x86

par Emmanuel Saracco

Historique des versions

Version v1.0a 2004-02-12 Revu par : es
Passage d’un source SGML à un source XML.
Version v1.0 2002-04-09 Revu par : es
Première version.

Table des matières

1. Les premiers pas 1
1.1. Les conventions Intel 1
1.2. Les conventions AT&T 1
1.3. Qu'est-ce que c'est? 2
1.4. Comment ça fonctionne? 4
2. Registres et structures 7
2.1. Les registres de travail 7
2.2. Les registres d'offset 8
2.3. Le registre des drapeaux 9
2.4. Les structures de contrôle 9
2.4.1. Boucle `WHILE` 9
2.4.2. Boucle `DO..WHILE` 10
2.4.3. Boucle `FOR` 11
2.5. Copie de chaîne 12
3. Programmation structurée en assembleur 14
3.1. La pile 14
3.2. Les fonctions 15
3.3. Les macros 18
BIBLIOGRAPHIE 20

Liste des tableaux

1-1. Comparatif des tailles 4
1-2. Exemple d'instructions assembleur 4
2-1. Découpage d'un registre de travail 7
2-2. Rôles des registres de travail 7
2-3. Rôles des registres d'offset 8
2-4. Les bits du registre `eflags` 9
3-1. Etat de la pile après empilement des arguments 16
3-2. Etat de la pile après instruction `call` 17
3-3. Etat de la pile après la première ligne du prologue 17
3-4. Etat de la pile après la seconde ligne du prologue 17

Chapitre 1. Les premiers pas

Ce document consacrés au langage d'assemblage pour processeurs x86 sous linux s'adresse à des débutants ayant quelques notions de programmation et ne se veut en aucun cas exhaustif.

Ceux d'entre vous ayant déjà codé en assembleur[1] sous DOS ou windows seront certainements surpris d'apprendre que la syntaxe qu'ils avaient employée jusqu'à présent n'est pas la seule disponible. En effet, il existe en fait deux styles de notation possibles :

1.1. Les conventions Intel

Ces conventions viennent directement du constructeur de microprocesseur ; c'est pourquoi elles sont suivies par la majeure partie des assembleurs modernes. Le code généré est clair et épuré.

Exemple:

```
push ebp
mov ebp,esp
push ebx
push esi
push edi
```

1.2. Les conventions AT&T

Ce style de codage est né en même temps qu'Unix et les conventions datent donc de cette époque. Il est un peu perturbant si l'on a pris l'habitude de coder en suivant les conventions Intel, mais on se familiarise assez rapidement avec.

Exemple:

```
pushl %ebp
movl %esp,%ebp
pushl %ebx
pushl %esi
pushl %edi
```

Il existe quelques assembleurs qui permettent d'utiliser les conventions Intel sur de l'Unix, le plus abouti étant Nasm. Dans ce document nous utiliserons les conventions de notation Intel et emploierons Nasm pour les raisons suivantes :

- La convention Intel nous semble être plus claire et moins source d'erreurs que la convention AT&T.
- Nasm permet à ceux qui étaient habitués à coder sous DOS de garder leurs habitudes :-)

A vrai dire, deux bonnes raisons qui auraient pu nous faire pencher pour l'utilisation des conventions AT&T sont :

- Le format d'affichage de GNU gdb.
- La syntaxe de l'assembleur inline dans du code C via `__asm__()`.

Néanmoins, nous adressant ici à des débutants en assembleur il ne fallait pas compliquer les choses[2].

1.3. Qu'est-ce que c'est?

Vous avez sûrement déjà codé avec un langage de haut niveau comme C ou Pascal. Ces langages permettent d'écrire de manière quasi naturelle ce que nous voulons que la machine fasse : un programme C exécutant : « afficher "coucou" » s'écrira[3]:

Exemple `coucou_c.c`

```
void main()
{
puts("coucou\n");
}
```

La même chose en assembleur s'écrirait[4]:

Exemple `coucou_asm.asm`

```
section .text
global _start

msg        db  'coucou',0x0A
msg_len    equ $ - msg

_start:
mov eax,4
mov ebx,1
mov ecx,msg
```

```
mov edx,msg_len
int 80h

mov eax,1
int 80h
```

Ce dernier source est tout à fait fonctionnel et autonome. Nous aurions pu également nous aider de la libc et éviter la manipulation directe de l'interruption 0x80, et nous aurions alors eu le mélange de C et d'assembleur suivant[5]:

Exemple `coucou_asm_libc.asm`

```
extern puts

section .text
global main

msg db 'coucou',0

main:
push ebp
mov  ebp,esp
push ebx
push esi
push edi

push dword msg
call puts

pop  edi
pop  esi
mov  esp,ebp
pop  ebp
ret
```

Lier son code assembleur avec la libc peut-être effectivement très pratique si l'on ne veut pas réinventer la roue ; mais il faut à ce moment se poser la question de savoir si nous n'aurions pas plus vite fait d'intégrer de l'assembleur inline dans un code C. Dans le cadre d'applications « professionnelles » il est néanmoins conseillé de ne pas passer directement par l'interruption logicielle 0x80 pour faire appel aux routinex kernel. En effet, rien ne certifie que les services proposés par cette interruption ne changeront pas — l'équipe de développement kernel conseillant de toujours passer par la libc s'agissant des appels systèmes et se réservant le droit de modifier quoi que ce soit sans crier gare. Pour en savoir plus sur l'interruption 0x80 et les différents appels systèmes (http://home.snafu.de/phpr/lhpsyscal.html).

Malgré tout il existe quelques avantages à coder en assembleur « pur » (c'est à dire, à ne pas se lier avec la libc) :

- La vitesse d'exécution.
- La taille du code généré.
- Le faible besoin en RAM.

Pour ce qui est de la vitesse d'exécution, il est vrai qu'on arrive rapidement à produire un code moins performant que celui généré par gcc, mais tout dépend de ce qu'on fait de ce pour quoi on utilise l'assembleur. Dans la majeure partie des cas la nécessité d'écrire un programme entier en assembleur ne se posera pas — s'il s'agit d'optimisation, on emploiera plus volontier de l'assembleur inline.

Mais si, par exemple, il est vital d'obtenir un binaire très petit, si la RAM disponible est très faible ou encore si l'on ne veut absolument pas se lier avec une librairie comme la libc, alors on pourra se fier à l'assembleur. Quelques explications supplémentaires pourront être trouvées sur Linuxego (http://linuxego.mine.nu/why.shtml).

Voici en guise d'exemple un tableau récapitulant les différents programmes sourcés ci-dessus et la taille des binaires générés (avant et après un strip [6]) :

Tableau 1-1. Comparatif des tailles

Nom	Taille	Strip
`coucou_asm`	428	428
`coucou_asm_libc`	4751	2956
`coucou_c`	4792	3000

On voit que c'est sans conteste le programme écrit en assembleur qui l'emporte au niveau de la taille finale[7].

1.4. Comment ça fonctionne?

L'assembleur permet d'opérer directement sur le processeur, la mémoire ou les périphériques. Sa syntaxe est simple, pour ne pas dire rudimentaire, et toutes les opérations possibles ne sont faites qu'à l'aide de quelques instructions comme `mov`, `push`, `pop` etc. En règle générale, les instructions sont nommées assez explicitement pour que l'on comprenne plus ou moins intuitivement l'opération effectivement réalisée :

Tableau 1-2. Exemple d'instructions assembleur

Instruction	Opération effectuée

Instruction	Opération effectuée
`mov`	Permet de « bouger » des données d'un emplacement à un autre (espace mémoire ou registre processeur)
`push`	Sert à « pousser » des données sur la pile
`pop`	Permet de récupérer les données empilées via push
`...`	...

Que veut dire exactement « s'adresser directement au processeur » ? Les langages de plus haut niveau ne permettent-ils pas cela également ?

En fait, le compilateur ne peut pas passer directement du C au bytecode. Il est obligé d'implémenter un analyseur syntaxique complexe qui interprètera ce que le programmeur à voulu faire dans un jargon particulier (C, Pascal etc.).

Lorsque vous codez en assembleur, vous ne dites pas `A = 0`, mais vous demandez directement au processeur de mettre un registre à `0`, par exemple. La variable `A` sera en fait le registre `eax` ou une de ses subdivisions (`al`, `ah` ou `ax` [8]) et sa mise à `0` pourra se faire de plusieurs manières :

- Explicitement `mov eax,0`.
- Via une opération logique : `xor eax,eax`. Cette seconde voie étant bien plus rapide que la première.

Un compilateur C pourra par exemple traduire le code suivant :

```
A = 0
B = 0
```

en :

```
xor eax,eax
mov A,eax
xor eax,eax
mov B,eax
```

On ne peut pas dire que cela soit optimum... Si nous avions fait cela directement en assembleur, nous aurions évité de définir 2 variables en utilisant le plus possible les registres :

```
xor eax,eax
xor ebx,ebx
```

Comme vous l'aurez compris, tout se passe souvent dans le détail, mais au bout du compte, c'est ce qui fera la différence. gcc et les compilateurs modernes génèrent du code de plus en plus optimisé, mais cela restera toujours de la génération automatique et le bytecode se verra toujours pollué de choses inutiles.

On pourrait cependant se demander en quoi une instruction assembleur `xor` est plus proche de la machine qu'une instruction C. En fait, les instructions assembleur sont juste étudiées pour être un peu plus digestes que du code hexadécimal ou binaire ; mais elles y ont leur équivalent exact. Par exemple `xor ax,ax` se traduira directement par le code hexadécimal `31C0` qui lui-même représente le code machine `0011000111000000`. Il faut donc bien comprendre qu'en codant en assembleur, on code en fait quasi directement en binaire, et rien ne vient interférer entre ce que l'on veut faire et la façon dont cela sera effectivement fait.

Notes

1. L'expression exacte serait « langage d'assemblage », l'assembleur étant le programme qui permet de transformer le langage d'assemblage en code binaire ; néanmoins, nous emploierons ici le terme « assembleur » pour nous simplifier la tâche.
2. Nous envisagerons peut-être de dédier un chapitre à la convention AT&T par la suite — tout dépendra de la demande :-)
3. Le code est volontairement dépouillé du superflu. Utilisez **`gcc coucou_c.c -o coucou_c`** pour compiler cet exemple.
4. Utilisez **`nasm -f elf coucou.asm ; ld -s coucou_asm.o -o coucou_asm`** pour compiler cet exemple.
5. Utilisez **`nasm -f elf coucou_asm_libc.asm ; gcc coucou_asm_libc.o -o coucou_libc`** pour compiler cet exemple.
6. **`strip mon_prog`**. Stripper un programme consiste à l'alléger en supprimant les symboles qu'il contient.
7. Pour vous détendre un peu et voir jusqu'ou on peut aller pour faire maigrir un code, jetez un oeil sur A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux (http://www.muppetlabs.com/~breadbox/software/tiny/teensy.html)
8. Nous ne sommes pas encore entré dans les détails des registres, mais anticipons un peu en disant seulement qu'un registre peut contenir des données de différentes tailles : `al` et `ah` contiennent des données sur 8 bits et constituent à eux deux le registre `ax` qui lui contient des données sur 16 bits. Au-dessus nous avons le registre `eax` qui permet de stocker des données sur 32 bits (il n'existe pas de découpage en 2x16 bits comme pour `ax`).

Chapitre 2. Registres et structures

Dans le Chapitre 1 nous avions laissé de côté la description des registres du processeur et leur rôle respectif. Nous allons ici rapidement aborder le sujet — juste assez pour pouvoir nous débrouiller par la suite.

Tout d'abord, précisons que sous GNU/Linux, contrairement à ce qui se passait sous DOS, nous n'avons pas vraiment besoin de nous encombrer avec la notion de « segment ». En effet, bien que les segments soient évidemment présents, leur taille peut aller jusqu'à 4Go... Donc, nous n'aurons plus à jouer avec `cs`, `ds` et `es` constamment :-) Penchons-nous plutôt sur ce qui nous touche de près : les registres du microprocesseur. Les registres qui nous intéressent particulièrement sont :

- Les registres de travail : `eax`, `ebx`, `ecx` et `edx`.
- Les registres d'offset : `esi`, `edi`, `ebp` et `esp`.
- Le registre des flags : `eflags`.

2.1. Les registres de travail

Comme nous l'indiquions brièvement dans le Chapitre 1, ces registres (sur 32 bits depuis le 80386) peuvent être décomposés en plusieurs sous-registres. Vous serez sûrement surpris de constater qu'on ne peut atteindre directement les 16 bits de poids fort des registres `e*x`, et que les registres `*x` ne représentent que les 16 bits de poids faible[1]. Si cela vous dérange, vous pourrez toujours utiliser l'instruction movzx qui permet d'étendre *x dans e*x.

Tableau 2-1. Découpage d'un registre de travail

Registre(s)		Taille
`eax`		32 bits
`ax`		16 bits
`ah`	`al`	8 bits chacun

On voit que `e*x` (32 bits) est composé de `*x` (16 bits de poids faible) lui-même composé de `*h` (8 bits de poids fort) et `*l` (8 bits de poids faible).

Tableau 2-2. Rôles des registres de travail

Registre	Nom	Description

Registre	Nom	Description
`eax`	**A**ccumulateur	Sert aux opérations mathématiques mais aussi à la transmission des numéros de fonctions à appeler lorsqu'on se sert de l'interruption système `0x80` et au renvoi des résultats dans les fonctions (que nous aborderons au Chapitre 3). Lorsque vous devez utiliser un registre pour une opération quelconque et que vous ne savez pas lequel utiliser, privilégiez celui-ci — c'est le plus optimisé au niveau de la rapidité d'exécution des opérations.
`ebx`	**B**ase	Utilisé pour l'adressage indirect
`ecx`	**C**ompteur	Ce registre est utilisé chaque fois que l'assembleur a besoin d'un « compteur ». Nous devrons l'utiliser lorsque nous mettrons en place des boucles via l'instruction loop ou encore lorsque nous ferons des transferts de données entre `esi` et `edi`.
`edx`	**D**onnées	Sert aussi pour quelques opérations mathématiques.

Même si chacun de ces registres est optimisé ou utilisé pour certaines opérations, ils peuvent être employés à toutes fins utiles. Il suffit juste de savoir, par exemple, que c'est le registre `ecx` et lui seul qui sera utilisé et décrémenté par l'instruction `movsd` dans le cas de la copie d'une chaîne[2]. Cela ne veut pas dire que `ecx` est dédié à cette tâche de « compteur », mais juste qu'il est optimisé par le processeur pour cette tâche et utilisé en tant que tel par d'autres instructions.

2.2. Les registres d'offset

Tableau 2-3. Rôles des registres d'offset

Registre	Nom	Description
`esi`	**S**ource **I**ndex	Utilisé lors des opérations sur les chaînes de caractères
`edi`	**D**estination **I**ndex	Comme `esi`, ce registre sert lors des opérations sur des chaînes de caractères.

Registre	Nom	Description
`ebp`	**B**ase **P**ointer	Référence la base de la pile
`eip`	**I**struction **P**ointer	Ce registre est particulier car il ne peut pas être manipulé directement. Il pointe en permanence sur la *prochaine* opération à exécuter.
`esp`	**S**tack **P**ointer	Pointe vers le dernier élément déposé sur la pile (l'élément courant)

2.3. Le registre des drapeaux

`eflags` est un registre 32 bits qui rend compte de l'état du processeur après chaque instruction. Il est composé du sous-registre `flags` (16 bits de poids faible). On n'accède jamais à ce registre dans son intégralité, mais toujours bit par bit lorsqu'on veut une information bien précise sur, par exemple, le résultat d'une comparaison ou d'une opération arithmétique. Les drapeaux les plus importants pour nous sont les suivants :

Tableau 2-4. Les bits du registre `eflags`

Drapeau	Nom	Position
`cf`	**C**arry **F**lag	`0`
`pf`	**P**arity **F**lag	`2`
`af`	**A**uxiliary carry **F**lag	`4`
`zf`	**Z**ero **F**lag	`6`
`sf`	**S**ign **F**lag	`8`
`if`	**I**nterruption **F**lag	`9`
`df`	**D**irection **F**lag	`10`
`of`	**O**verflow **F**lag	`11`

Il en existe d'autres, mais nous ne les utiliserons pas ici.

2.4. Les structures de contrôle

On peut bien sûr produire en assembleur le même genre de boucle que dans un langage de haut niveau. Nous verrons à cette occasion l'utilité des registres `eflags` et `ecx`.

2.4.1. Boucle `WHILE`

Le test est fait en *début* de boucle.

En C, nous ferions:

```
    compteur = 0;
    while ( compteur < 10)
    ++compteur;
```

En assembleur, nous faisons:

```
xor ax,ax
debut:
cmp ax,10
jae fin
inc ax
jmp debut
fin:
```

L'instruction jae nous permet ici de tester si le terme gauche de la comparaison faite avec cmp est supérieur ou égal au terme de droite. Dès que ax est *supérieur* ou égale à 10 alors on saute au label `fin`, sinon, on incrémente `ax` et on saute impérativement au label debut. `jae` vérifie la valeur du flag `CF`. Si `CF=0`, alors la condition est remplie.

2.4.2. Boucle `DO..WHILE`

Le test est fait en *fin* de boucle.

En C, nous ferions:

```
compteur = 0;
do
++compteur;
while (compteur < 10);
```

En assembleur, nous faisons:

```
xor ax,ax
debut:
inc ax
cmp ax,10
jb debut
```

L'instruction jb nous permet ici de tester si le terme gauche de la comparaison faite avec `cmp` est inférieur au terme de droite. Tant que `ax` est *inférieur* à 10 alors on saute au label debut. `jb` vérifie la valeur du flag `CF`. Si `CF=1`, alors la condition est remplie.

2.4.3. Boucle `FOR`

Aucun test n'est fait. On effectue un nombre determiné d'itérations.

En C, nous ferions:

```
int compteur;
for (compteur = 0; compteur < 10; compteur++)
;
```

En assembleur, nous faisons:

```
mov ecx,10
debut:
nop
loop debut
```

La boucle `for` n'étant en fin de compte qu'une variante de `while` il existe bien sûr plusieurs façons de faire. Ici, nous nous servons du registre compteur `ecx`. Nous avons un bel exemple d'instruction `loop`

qui s'attend à trouver dans `ecx` le nombre d'itérations à accomplir et qui décrémente automatiquement ce registre.

2.5. Copie de chaîne

Pour vous montrer dans quel cas on utilise les registres d'offset `esi` et `edi`, nous allons aborder sommairement, toujours dans le cadre de l'étude des boucles, le traitement des chaînes de caractères.

Le programme suivant recopie la chaîne se trouvant dans la variable `strsrc` dans la variable `strdst`[3] :

Exemple `strcpy.asm`

```
section        .text
global        _start

_start:
mov esi,strsrc
mov edi,strdst
mov ecx,strsrc_len
debut:
mov eax,[esi]
mov [edi],eax
inc esi
inc edi
loop debut
fin:
mov        eax,1
int        0x80

section .data
strsrc db "premiere chaine",0x0A
strsrc_len equ $ - strsrc
strdst times strsrc_len db 0
```

Nous avons vu au Tableau 2-3 que les registres `esi` et `edi` servaient aux opérations sur les chaînes. Ici, nous faisons pointer `esi` vers la variable contenant la chaîne source, et `edi` vers la variable de destination. Ensuite, nous mettons dans `ecx` le nombre de caractères à recopier, puis nous bouclons jusqu'à ce que l'instruction `loop` ait assez décrémenté `ecx` pour le mettre à zéro.

Au sein de la boucle, nous prenons chacun des caractères contenus par `esi` et nous les mettons dans `edi` . Pour ce faire, nous sommes obligés de passer par un registre intermédiaire car il est impossible de

copier directement le contenu de l'un vers l'autre. Après chaque copie de caractère, on incrémente `esi` et `edi` pour qu'ils pointent sur le caractère suivant.

Ces opérations vous paraissent lourdes ? Effectivement... Ces copies étant quelque chose de courant, il existe des instructions qui les optimisent :-)

En fait, on aurait dû coder la boucle ci-dessus comme suit :

```
mov ecx,strsrc_len
mov esi,strsrc
mov edi,strdst
rep movsb
```

Le processeur exécutera ce code près de trois fois plus vite que le code précédent. L'instruction `rep`, associée à une instruction comme `movsb` répètera autant de fois que `ecx` le lui « demandera » en le décrémentant automatiquement.

Notes

1. On désigne par « bits de poids faible » les bits situés le plus à droite, et « bits de poid fort » ceux situés le plus a gauche. Par exemple, si nous travaillons avec un registre de 16 bits et que nous y mettons la valeur hexadécimale `0x4d50` : `mov ax,4d50h`, `al` contiendra `0x50` et `ah` contiendra `0x4d`.
2. nous aborderons cet exemple plus loin lorsque nous parlerons des boucles.
3. Utilisez **`nasm -f elf strcpy.asm ; ld -s strcpy.o -o strcpy`** pour compiler cet exemple.

Chapitre 3. Programmation structurée en assembleur

3.1. La pile

On utilise la pile pour stocker des données ou des adresses de manière temporaire. Elle peut être vue comme un lieu d'échange dans lequel on dépose et on reprend des éléments. Elle fonctionne sur le mode LIFO[1]. C'est pour cela que tout ce qu'on déposera sur la pile via l'instruction `push` devra obligatoirement en être retiré dès que possible dans l'ordre inverse à l'ordre de dépôt via l'instruction `pop`.

```
push eax
push ebx
push ecx

[...]

pop ecx
pop ebx
pop eax
```

Le code ci-dessus vaut si vous avez besoin de retrouver dans les mêmes registres les valeurs sauvegardées sur la pile. Nous aurions très bien pu récupérer la valeur poussée en premier (celle de `eax` dans `ebx` en modifiant l'ordre des `pop`).

Il y a également des fois ou l'on se moque complètement de récupérer les valeurs empilées[2]. Dans ce cas l'utilisation de `pop` est inutile, et même mauvaise pour les performances. Il faudra donc pouvoir accéder directement à la pile en nous servant de la valeur contenue dans le registre `esp`.

En fait, lorsque le programme se lance, `esp` pointe sur la fin du segment principal[3] : le début de la pile pour notre programme. Les opérations faites par `push` feront descendre le pointeur, et celles effectuées par `pop` le feront remonter. Il faudra donc toujours penser qu'ajouter quelque chose sur la pile (en fait, *en-dessous*), décrémentera le pointeur courant `esp`. Nous venons de voir que l'on se sert habituellement de `pop` pour rétablir l'état de la pile, mais qu'il existe d'autres moyens. On peut très bien, puisqu'il ne s'agit en fait que de déplacer le pointeur dans la pile, augmenter la valeur de ce pointeur directement.

Lorsque, par exemple, j'empile `eax`, je diminue la valeur du pointeur de 4 octets (32 bits), et lorsque je fais un `pop`, je récupère la valeur et le pointeur est incrémenté de 4 octets pour retrouver son emplacement d'origine en haut de la pile. Cette opération étant lourde au niveau des performances[4] on pourra dès que possible vouloir juste faire en sorte que le pointeur retrouve son emplacement :

```
push dword arg2
push dword arg1

call ma_fonction
```

```
  add sp,8
```

La fonction `ma_fonction` nécessitant deux arguments et les attendant sur la pile[5], nous les empilons avans l'appel. Au retour de la fonction nous faisons le ménage en incrémentant le pointeur de pile de 8 octets[6].

Cela ne vaut évidemment que lorsque nous appelons une fonction qui respecte les conventions du C au niveau du passage d'arguments[7].

En ce qui concerne la pile, nous en savons bien assez pour aborder les fonctions.

3.2. Les fonctions

Notre découverte de l'assembleur nous montre bien qu'au bout d'un moment les programmes doivent devenir plutôt indigestes... C'est pourquoi, comme avec tout autre langage, il est possible de structurer son code en mettant en place des fonctions[8].

Une fonction utile, par exemple, pourrait prendre en charge l'affichage d'une chaîne. Voici donc notre `coucou_asm.asm` de la Section 1.3 un peu plus structuré[9]:

Exemple `coucou_func_asm.asm`

```
section .text
        global _start

_start:
        push dword msg_len
        push dword msg
        call write_screen
        add esp,8

        mov eax,1
        int 80h

write_screen:
        push ebp
        mov ebp,esp

        mov eax,4
        mov ebx,1
        mov ecx,[ebp + 8]
        mov edx,[ebp + 12]
        int 80h
```

```
        mov esp,ebp
        pop ebp
        ret

section .data
        msg db "coucou",0x0A
        msg_len equ $ - msg
```

Ca n'allonge notre code que de 68 petits octets, et c'est tout de même plus propre (si l'on doit faire un programme plus utile évidemment qui aura recours plus d'une fois à cette fonction).

C'est la façon la plus propre de coder une fonction en assembleur. Le *prologue* :

```
  push ebp
  mov ebp,esp
```

et l'*épilogue*:

```
  mov esp,ebp
  pop ebp
```

permettent à notre fonction de créer son propre espace sur la pile pour le cas ou nous aurions à nous en servir. C'est assez simple à comprendre. Nous avons vu dans le Tableau 2-3 qu'il existe un registre qui contient en permanence l'offset du sommet de la pile : `ebp` . Donc notre programme, pour savoir ou commence la pile, se fie toujours à ce registre. Ce que nous faisons à l'entrée de notre fonction permet de déplacer la base du nouvel espace de notre pile à la fin de la pile effectivement utilisée par le programme appelant, en nous fiant à `esp` qui pointe la dernière valeur empilée par l'appelant.

Pour récupérer chacun des arguments passés à notre fonction, on utilise l'adressage indirect, relatif à la base de notre nouvelle pile, en remontant pour aller les chercher sur le sommet de la pile du programme appelant. On voit donc que le dernier argument empilé par l'appelant sera accessible via le plus petit déplacement (ici, 8).

La question qui se pose est : pourquoi 8 ? En fait, l'instruction `call` à demandé un petit travail supplémentaire au processeur : l'empilement discret du contenu du registre `eip`, qui contient l'offset d'exécution de l'appelant au moment de son appel. `call` va donc empiler la valeur d'`eip` et `ret` s'en servira pour sauter vers cet offset une fois notre fonction terminée.

Après avoir poussé nos deux arguments, notre pile ressemble donc à :

Tableau 3-1. Etat de la pile après empilement des arguments

Registres	Pile	Offset factice (en décimal)
ebp->	msg	255
esp->	msg_len	251

Lorsque nous faisons appel à `call`, cette instruction modifie la pile comme suit :

Tableau 3-2. Etat de la pile après instruction `call`

Registres	Pile	Offset factice (en décimal)
ebp->	msg	255
	msg_len	251
esp->	eip	247

Ensuite, après la première ligne du prologue (`push ebp`) :

Tableau 3-3. Etat de la pile après la première ligne du prologue

Registres	Pile	Offset factice (en décimal)
ebp->	msg	255
	msg_len	251
	eip	247
esp->	ebp	243

et après la seconde ligne du prologue (`mov ebp,esp`) :

Tableau 3-4. Etat de la pile après la seconde ligne du prologue

Registres	Pile	Offset factice (en décimal)
	msg	255
	msg_len	251
	eip	247
ebp = esp->	ebp	243

On sauvegarde donc `ebp` pour pouvoir retrouver la base de notre ancienne pile dans le programme appelant, et on l'initialise avec `esp` après lequel nous pouvons empiler tout ce que l'on voudra :-)

C'est une bonne habitude à prendre, et puis cela permet également de faire des fonctions récursives qui ne crachent pas dès le premier retour d'imbrication...

On comprend à présent pourquoi il est nécessaire d'ajouter 8 octets à `ebp` pour accéder à nos arguments : c'est qu'entre le début de notre nouvelle pile et la fin de celle de l'appelant, le processeur à stocké l'adresse de retour.

Il faudra bien évidemment penser à sauvegarder tous les registres que nous utiliserons au sein de notre fonction et à les restaurer avant l'épilogue. Il faut toujours avoir à l'esprit la restitution en l'état de la pile et des registres au programme appelant.

Pour les fonctions qui retournent une valeur, la convention veut que la valeur retournée le soit dans `eax`[10].

Avertissement

Gardez bien à l'esprit tout de même que, s'il est vrai que les fonctions permettent une meilleure lisibilité et une meilleure maintenance du code, elle le ralentissent également[11]. Si c'est donc la plus grande rapidité possible que vous recherchez et que votre code n'est après tout pas si long que ca, n'hésitez pas à écrire le même code trois ou quatre fois au lieu de l'encapsuler dans une fonction. Le programme généré sera bien sûr plus long, mais aussi plus rapide. A vous de voir.

3.3. Les macros

Si vous préférez éviter les fonctions, vous pouvez toujours préserver une certaine lisibilité de votre code à l'aide des macros.

Contrairement aux fonctions, qui sont effectivement traduites comme telles dans le code binaire généré par le compilateur, les macros ne sont là que pour aider le programmeur à y voir plus clair, mais dans le code final on ne trouvera aucun appel. Leur code sera juste inséré à l'endroit des différentes références qui y auront été faites.

Notre programme `coucou_func_asm.asm` se transforme donc en :

Exemple `coucou_func_asm.asm`

```
%macro write_screen 2
        mov eax,4
        mov ebx,1
        mov ecx,%1
        mov edx,%2
        int 80h
%endmacro

section .text
        global _start

_start:
        write_screen msg,msg_len

        mov eax,1
        int 80h

section .data
```

```
msg db "coucou",0x0A
msg_len equ $ - msg
```

On voit que la définition d'une macro est toute simple. Il suffit juste de spécifier derrière son nom le nombre d'arguments qu'elle attend. Pour ce qui est de l'appel, on dirait presque un langage évolué :-)

Si vous regardez la taille du code généré, elle est presque équivalente à la taille de notre `coucou_asm.asm` de la Section 1.3, mais avec l'appel en plus.

Notes

1. « Last In First Out ». Il faut voir ça comme un empilement d'assiettes un peu spécial, par exemple, auquel on ajouterait et retirerait des assiettes *par en-dessous*. L'exemple des assiettes est un peu trompeur tout de même ; en effet, on peut très bien accéder à n'importe quel élément de la pile en y faisant référence de manière indirecte via `esp`, comme nous le verrons.
2. Dans le cas des appels de fonctions avec passage d'arguments, par exemple.
3. Si nous n'étions pas en mode protégé, le segment relatif à la pile serait `ss`.
4. Toute opération concernant la pile est plus lourde que les opérations avec les registres, c'est pourquoi il est préférable lorsque cela est possible de sauvegarder les valeurs dans des registres au lieu de les empiler/dépiler.
5. Nous aurions tout aussi bien pu lui passer via deux registres, ce qui aurait d'ailleurs été plus rapide.
6. Chaque valeur poussée sur la pile occupant ici 4 octets (`dword`).
7. Il existe deux conventions pour l'empilement/dépilement des arguments passés à une fonction : la convention C (c'est à l'appelant de nettoyer la pile — cela permet une gestion plus aisée des arguments variables), et la convention Pascal (c'est à la fonction appelée de nettoyer la pile).
8. Que les pascaliens nous excusent, nous ne ferons pas la différence entre fonction et procédure ici, et suivrons plutôt le vocabulaire C.
9. Compilez avec **`nasm -f elf coucou_asm.asm ; ld -s coucou_asm.o -o coucou_asm`**
10. C'est en tout cas ainsi que procèdent les fonctions système Linux.
11. Les instructions `call`, `ret`, ainsi que toutes les opérations impliquant la pile sont très coûteuses en cycles processeur.

BIBLIOGRAPHIE

Livres

A.B. Fontaine, 1983, 1984, Publié par Masson, 2-225-80313-7, Masson, Paris, *Le microprocesseur 16 bits 8086/8088, matériel - logiciel - système d'exploitation.*

M. Margenstern, 1983, 1991, Publié par Masson, 2-225-82500-9, Masson, Paris, *ASSEMBLAGE, modélisation, programmation (80x86)*.

Jeff Duntemann, 1992, 2000, Publié par WILEY, 2-225-80313-7, John Wiley & Sons, Inc, *Assembly Language Step-by-Step, Second Edition, Programming with DOS and Linux*.

Holger Schakel, 1992, 1993, Publié par Micro Application, 2-86899-796-1, Micro Application, *Programmer en Assembleur sur PC*.

Philippe Mercier, 1989, Publié par Marabout, 2-501-01176-1, Micro Application, *Programmer en Assembleur sur PC*.