Initiation à l'assembleur 8086 sur système GNU/Linux

D'après Rémi HOARAU

Partie II : Test d'exécution et structure d'un programme

I. Test d'exécution

Créons un fichier et écrivons ces lignes :

test.s

```
1  BITS 32
2
3  global _start
4
5  section .data
6      chaine: db "Youpi, ça fonctionne !",0xa,0x0
7
8  section .text
9      _start:
10         mov eax,4
11         mov ebx,1
12         mov ecx,chaine
13         mov edx,24
14         int 0x80
15
16         mov eax,1
17         int 0x80
```

C'est du 32 bits et nous savons assembler du 32 bits. Tapons ces commandes :

`nasm –f elf32 test.s -o test.o`

`ld –m elf_i386 test.o -o programme`

Puis exécutons le programme avec `./programme` :

```
osboxes@osboxes:~$ ./programme
Youpi, ça fonctionne !
```

Notre premier programme en langage d'assembleur ! Trop cool !

Décortiquons le code.

II. Structure d'un programme

Plusieurs sections composent un programme, tels que :

.text : c'est la section qui correspond au code en assembleur où ses instructions seront exécutés.

.data : c'est la section de données initialisées. Ici, nous pourrons placer nos variables (initialisées), nos chaînes de caractères, nos tableaux, nos pointeurs, nos structures, etc...

.bss (block started by symbol) : c'est la section qui regroupe les données non initialisées. C'est ici que nous mettrons les variables non initialisées ou encore les chaînes de caractères que l'utilisateur doit saisir.

Au tout début de notre code, nous avons écrit **BITS 32**, ce qui signifie que nous coderons en 32 bits. Si l'on voulait du 64 bits on aurait écrit **BITS 64**. Cela aurait modifié les appels systèmes, la taille de la pile et la taille des regitres. J'en parlerai un peu plus tard.

La ligne 3 indique **global _start** : Cela permet de dire à **ld** (notre lieur de fichier) que le point d'entrée de notre programme s'appellera _start (si on lie les fichiers avec ld, on devra utiliser _start. Si on les lie avec GCC, on devra utiliser _main)

Les lignes 5 et 6 représentent la section de données initialisées. Ici, nous avons initialisé une chaîne de caractères nommée **chaine**. Il faut absolument savoir que les variables en assembleur représentent DIRECTEMENT des adresses. En indiquant le tableau de caractères **chaine**, on indique directement l'adresse du tableau.

Les types de données existant sont les suivants :

Octet => byte / b => 1 octet => 8 bits

Mot => word / w => 2 octets => 16 bits

Double mot => double word / dword / double / d / dw => 4 octets => 32 bits

Quadruple mot => quad word / qword / quad / q / qw => 8 octets => 64 bits

En fait, il existe plusieurs mots clés pour définir certains types de données dans la section .data :

```
ma_variable: equ 256            ; définir une variable contenant un nombre entier
                                ; égal à 256

mon_tableau: db "abc",0xff,0x2  ; définir un tableau d'octets (db pour
                                ; define byte) contenant les lettres
                                ; 'a','b','c' suivi des valeurs 255 et 2.

mon_tableau: dw 0xabcd,0x1234   ; définir un tableau de mots (dw pour
                                ; define word). Un mot est la
                                ; concaténation de deux octets.

mon_tableau: dd 0xabcd1234      ; définir un tableau de double mots
                                ; (dd pour define double). Un double
                                ; mot est la concaténation de deux
                                ; mots.

mon_tableau: dq 0xabcd,0x1234   ; définir un tableau de quadruple
                                ; mots (dq pour define quad word).
                                ; Un quadruple mot est la
                                ; concaténation de deux double
                                ; mots.
```

Pour la section de données non initialisées .bss, le principe est pratiquement le même mais les mots clés changent :

```
bloc1: resb 5       ; Réserver 5 octets à l'adresse 'bloc1'
                    ; resb pour "reserve byte"
```

bloc2: resw 2 ; Réserver 4 octets à l'adresse 'bloc2'

; **resw** pour "**reserve word**"

; attention, puisque 1 word = 2 bytes on a 2 words = 4 bytes

bloc3: resd 4 ; Réserver 16 octets à l'adresse 'bloc3'

; **resd** pour "**reserve double**"

; attention, puisque 1 dword = 4 bytes on a 4 dword = 16 bytes

bloc4: resq 3 ; Réserver 24 octets à l'adresse 'bloc4'

; **resq** pour "**reserve qword**"

; attention, puisque 1 qword = 8 bytes on a 4 qword = 24 bytes

À la ligne 8, on retrouve la section **.text** avec le point d'entrée du programme juste en dessous, qui est nommé **_start**. Ce point d'entrée représente une étiquette (que l'on nomme aussi "symbol") qui va dire à la machine << Ok, là t'es dans la section **.text** donc tu vas exécuter tout ce que je te mets dedans. Mon code commence juste après le **_start** >>. On pourrait parfaitement écrire d'autres symboles avant le **_start** mais il faut que tous les symboles ammenant à des instructions exécutables soient placés dans la section **.text**.

Notez que les symboles peuvent aussi représenter des noms de fonctions, ou des labels qui peuvent être sautés à la manière d'un **goto** en langage C ou C++.

Quelques points importants :

- Pour écrire un commentaire, il suffit d'utiliser le point-virgule '**;**'
- NASM n'est presque pas case sensitive (sensible aux majuscules/minuscules/espaces) donc libre à nous d'écrire dans des tabulations, ou par exemple d'écrire nos registres ou nos constantes en majuscules

- L'utilisation d'une coloration syntaxique sera extrêmement pratique plutôt qu'un simple bloc notes.
- Il faut parfaitement avoir compris les bases de numération (binaire, hexadécimal, décimal, octal, etc...) et savoir la différence entre chaque type de données (byte / word / dword / qword)
- Tout est adresse et pointeur en assembleur. Une variable est en fait un pointeur, qui pointe vers une adresse. Il faut déréférencer ce pointeur pour récupérer la valeur de la variable (nous le verrons plus tard). Il n'y a que l'opérateur '**equ**' qui considère la variable comme valeur entière brute, et non une adresse.

Dans le prochain chapitre, nous décortiquerons le contenu de la section **.text** et en particulier du symbole **_start**.