Travaux Pratiques n°1

Architecture des ordinateurs.

—IR 1—

Avant la séance de travaux pratiques suivante, vous enverrez une archive contenant les fichiers sources ainsi qu'un rapport au format pdf à votre chargé de td.

Arnaud Carayol : `arnaud.carayol@univ-mlv.fr`
Guillaume Blin : `guillaume.blin@univ-mlv.fr`

Vous donnerez à votre mail le sujet suivant:

```
[Archi IR1][TP1] Nom1--Nom2
```

Le rapport doit répondre aux questions posées dans le sujet et peut éventuellement contenir des portions de code pour illustrer votre propos. Votre code doit être commenté sinon il ne sera pas lu.

1 Rappels

Cette série de travaux pratiques a pour but de vous familiariser avec la programmation en langage machine pour des processeurs 32 bits d'architecture x86 sous linux en mode protégé.

Nous utiliserons l'assembleur NASM (Netwide Assembler) qui est téléchargeable gratuitement à l'adresse suivante:

`http://sourceforge.net/projects/nasm`

Je vous conseille l'excellent livre (traduit en français) de Paul. A. Carter intitulé *PC Assembly Language* qui est disponible à l'adresse suivante:

`http://www.drpaulcarter.com/pcasm/`

Les processeurs *32 bits* d'architecture x86 travaillent avec des registres qui sont en nombre limité et d'une capacité de 32 bits. Parmis ces registres, les registres appelés `eax`, `ebx`, `ecx`, `edx`, `edi`, `esi` sont des registres à usage général. Les registres `esp` et `eip` servent respectivement à conserver l'adresse du haut de la pile et l'adresse de la prochaine instruction à exécuter.

Le premier octet (celui de poids le plus faible) de `eax` est accessible par le registre `al`(de capacité 8 bits), le deuxième octet de poids le plus faible est accessible par le registre `ah`. Les 16 bits de poids faible de `eax` sont accessibles par le registre `ax`. De même pour `ebx`, `ecx`, et `edx`, on dispose des registres `bx`, `bh`, `bl`, `cx`, `ch`, `cl` et `dx`, `dh`, `dl`.

Question 1 Donnez un schéma illustrant les relations entre `eax`, `ax`, `ah` et `al`.

Opérations sur les registres Le processeur peut effectuer des opérations sur les valeurs stockées dans les registres. On notera que **le premier argument est toujours celui qui reçoit le résultat**[(1)].

[(1)] C'est la syntaxe Intel. Attention, il existe d'autres assembleurs qui emploient la convention inverse! On parle de syntaxe ATT.

```
mov eax, 3            ; eax = 3
mov ebx,eax           ; ebx = eax
mov eax, 0x08080808   ; eax=0x08080808

add eax,3             ; eax = eax + 3
add ah,0x1c           ; ah = ah + 28
add ebx,ecx           ; ebx = ebx + ecx
sub eax, 10           ; eax = eax - 10
add ah,al             ; ah = ah + al
```

Question 2 Après l'exécution des instructions suivantes quelle est la valeur de `eax`, `ax`, `ah` et `al`.

```
mov eax, 0xf00000f0
add  ax, 0xf000
add ah,al
```

Lecture et écriture en mémoire Le processeur peut aussi lire et écrire directement dans la mémoire. En mode protégé, on peut s'imaginer la mémoire comme un tableau dont chaque case contient un octet. L'adresse d'une case est donnée par un nombre sur 32 bits. On donne ci-dessous un exemple fictif de d'état de la mémoire.

adresse	valeur
0xffffffff	235
0xfffffffe	254
⋮	⋮
0x00000003	10
0x00000002	9
0x00000001	8
0x00000000	7

Question 3 Quelle est la quantité de mémoire adressable sur 32 bits ?

La syntaxe générale pour écrire en mémoire à l'adresse `adr` la valeur contenue dans le registre `reg` est la suivante:

```
mov [adr],reg
```

La syntaxe générale pour lire la mémoire à l'adresse `adr` et stocker la valeur dans le registre `reg` est la suivante:

```
mov reg,[adr]
```

La quantité d'information écrite ou lue est déterminée par la taille du registre reg. Par exemple si le registre est `ah` ou `al`, on lit ou écrit un octet.

```
mov al, [0x00000003]  ; al recoit l'octet stocké à l'adresse 3
                      ; dans notre example al=10
mov [0x00000001], ah  ; l'octet contenu dans ah est écrit à l'adresse 1
```

Quand on ne veut plus écrire des octets mais des mots (deux octets) ou des double-mots (quatres octets), il faut adopter une convention sur l'ordre dans lequel on lit les octets en mémoire. Considérons l'instruction ci-dessous:

```
mov eax, [0x00000000]
```

Cette instruction demande de lire 4 octets (car `eax` est un registre 32 bits) en mémoire à partir de l'adresse 0 et de les stocker dans `eax`. Si l'on reprend la mémoire décrite dans notre exemple, deux résultats seraient envisageables:

`eax`	=	`0x0a090807`	convention *little endian*
`eax`	=	`0x0708090a`	convention *big endian*

Les processeurs Intel et AMD utilisent la convention *little endian*.

On peut aussi directement affecter une valeur en mémoire sans la stocker dans un registre. Il faut alors préciser la taille des données avec les mot-clés `byte`, `word` et `dword`.

```
mov byte [0x0000 0000],1
mov word [0x0000 0002],1
mov dword [0x0000 0000],0x0200 0001
```

Question 4 Quel est l'état de la mémoire après avoir effectué chacune de ces instructions en partant de la mémoire de l'exemple ci-dessus ?

2 Premier programme

Le code source `hello.asm` de notre premier programme est téléchargeable à l'adresse suivante:

`http://igm.univ-mlv.fr/ens/IR/IR1/2008-2009/Archi/index.php`

Pour compiler le programme, on exécute:

```
nasm -f elf hello.asm
ld -e debut hello.o -o hello
chmod +x hello
```

La première commande crée un fichier objet `hello.o`. La seconde commande un exécutable `hello`. L'option `-e debut` indique que le programme commence à la ligne marquée par l'étiquette `debut`. La dernière ligne change les droits d'accés pour que le fichier `hello` soit exécutable.

Question 5 Compilez et exécutez le programme `hello`.

La première section du programme intitulée `data` contient les données du programme c'est à dire une succession d'octets. Les caractères sont représentés par leur code ASCII. La primitive *db* permet de déclarer une suite d'octets. Dans la suite l'étiquette `msg` désignera l'adresse du premier octet de la chaîne que l'on veut afficher. De la même manière, l'étiquette `debut` designera l'adresse du premier octet du code.

Le programme `hellobis` présente une version utilisant directement les codes ASCII.

Le programme `helloter` définit une constante `len` grâce à la primitive `equ` pour calculer la taille de la chaîne. Attention `len` n'est pas une adresse mais une valeur.

La deuxième section `text` contient le code. Dans notre cas, le code fait deux appels systèmes via l'instruction `int 80h`. Le registre `eax` contient le numéro de l'appel système : 4 pour `write` et 1 pour `exit`. Les registres `ebx`, `ecx` et `edx` contiennent les paramètres.

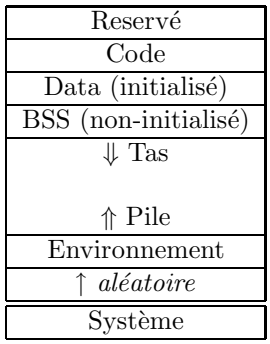
Question 6 Pour les appels systèmes `write` et `exit`, donnez le nombre d'arguments et expliquez leurs rôles.

Question 7 Écrivez un programme qui exécute le programme `/bin/ls`. Pour cela, on utilisera l'appel système `execve` qui a le numero 11.

- `ebx` doit contenir l'adresse de la chaine de caractères terminée par un caractère de code ASCII 0 correspondant à la commande à exécuter.
- Dans notre cas, `ecx` et `edx` peuvent être égaux à 0. En fait, `ecx` et `edx` servent à passer les arguments et les variables d'environnement respectivement.

3 Environnement d'exécution

L'environnement d'exécution d'un programme utilisateur est donné par le figure suivante. Le haut de la figure correspond à l'adresse `0x0000 0000` et le bas à l'adressse `0xFFFF FFFF`. L'adresse à laquelle commence le `Code` est toujours `0x08048080`. L'adresse du début de l'`Environnement` est dans le registre `esp`(voir plus loin). L'adresse à laquelle débute le bloc `Système` est `0xBFFF FFFF`.



Avec la commande `nasm -f elf -l hello.lst hello.asm`, vous obtenez dans `hello.lst` un listing donnant le code machine des différentes instructions.

Question 8 En utilisant la commande `xxd hello.o`, vous obtenez l'affichage binaire du fichier objet `hello.o`. Retrouvez votre code et vos données. Pourquoi le code binaire ne peut-il pas être copié en mémoire directement.

Question 9 Quels changements observez-vous dans l'exécutable `hello` ?

4 Utilisation de GDB

Nous allons maintenant voir comment utiliser le programme GDB pour suivre l'exécution en machine de notre premier programme. Pour plus de détails, reportez-vous à la documentation de GDB.

Commencez par lancer GDB sur le programme `hello`.

```
gdb hello
```

Commencez par désassembler le programme pour avoir les adresses des différentes instructions.

```
set disassembly-flavor intel
disassemble debut
```

Si vous tapez `run`, le programme s'exécute. Pour pouvoir l'arrêter, il faut placer un point d'arrêt sur l'une des commandes. Ainsi la commande `break *debut+5` place un point d'arrêt sur la deuxième commande du programme.

```
run
info registers
```

Puis `next` pour passer à l'instruction suivante.

Question 10 Créer un fichier exécutant le code de la question 2 et utilisez GDB pour vérifier votre résultat.

Une fonction très utile de GDB est qu'il peut afficher l'état de la mémoire. Les commandes suivantes affichent la mémoire à partir de l'adresse `adr` (ou `&label`):

`x/3wx adr`	3 double-mots (`w`) (double-mot = quatre octets) en hexadécimal (`x`)
`x/1bd adr`	un octet (`b`) en décidemal (`d`)
`x/1s adr`	une chaîne terminée par un caractère 0

Question 11 Utilisez cette fonction de GDB pour exploirer le bloc environnement dont la structure est décrite ci-dessous.

Toutes les valeurs sont sur 4 octets. `Argc` est le nombre d'arguments du programme plus 1. `Arg[0]` est l'adresse du premier caractère d'une chaîne terminée par un 0 qui correspond au nom du programme. `Arg[1]` est l'adresse d'une chaîne correspondant au premier argument.

esp	Argc
esp+ 4	Arg[0]
esp+ 8	Arg[1]
...	...
esp+ 4n +4	Arg[n]
esp+ 4n +8	0
esp+ 4n + 12	Env[0]
...	...

Question 12 A quoi correspond `Env[0]` ?

5 Améliorations

Question 13 Récupérez le fichier `hello2.asm` et déterminez la fonction du premier bloc d'instruction.

Les instructions ci-dessous ont le comportement suivant: tant que l'octet pointé par `edi` n'est pas égal à `al` ou que `ecx` n'est pas égal à 0 alors `ecx` est décrémenté de 1 et `edi` est incrémenté de 1.

```
cld
repnz   scasb
```

Question 14 Ecrivez un programme qui affiche son nom d'appel.

Question 15 Ecrivez un programme qui affiche son nombre d'arguments si ce nombre est inférieur à 10.