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

Arnaud Carayol : `arnaud.carayol@univ-mlv.fr`

Vous donnerez à votre mail le sujet suivant:

`[Archi OC1][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.

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). Si vous souhaitez travailler chez vous, vous pouvez le télécharger 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* est disponible à l'adresse suivante:

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

1 Rappels

Les processeurs *32 bits* d'architecture x86 travaillent avec des registres qui sont en nombre limité et d'une capacité de 32 bits (soit 4 octets). Parmis ces registres, les registres appelées `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 l'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`. Le 16 bits de poids faible de `eax` sont accessible par le registre `ax`(qui recouvre `al` et `ah`). Notez que les 2 octets de poids fort ne sont pas directement accessible par un sous-registre.

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 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 Quel est l'effet des instructions suivantes:

```
mov eax,15
mov eax,0xf
mov eax,0x000000f
mov eax,0b1111
```

Question 3 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	4
0xfffffffe	8
⋮	⋮
0x00000003	1
0x00000002	9
0x00000001	3
0x00000000	5

Question 4 Quelle est la quantité de mémoire adressable sur 32 bits ?

Lecture en mémoire 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]
```

Le nombre d'octets lus dépend de la taille de `reg`. Par exemple, 1 octet pour `al` ou `ah`, 2 octets pour `ax` et 4 pour `eax`.

```
mov al, [0x00000003]  ; al recoit l'octet stocké à l'adresse 3
                      ; dans notre example al=1
```

Question 5 Quelle est la différence entre `mov eax,3` et `mov eax,[3]`

Quand on cherche à lire plus d'un octet, il faut adopter une convention pour savoir dans quelle ordre ranger les octets dans le registre. Considerons par exemple, l'instruction suivante avec la mémoire de notre exemple:

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

On cherche à ranger les 4 octets situés aux adresses 0,1,2 et 3 dans `eax`. Ces octets valent respectivement 5,3,9 et 1. Il y a deux résultats possibles:

`eax` = `0x01090305` convention *little endian*
`eax` = `0x05030901` convention *big endian*

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

Au lieu de donner explicitement l'adresse où lire les données, on peut la mettre dans un registre. Ce registre doit nécessairement faire 4 octets.

```
mov eax,0      ; eax = 0
mov al,[eax]   ; al recoit l'octet situé à l'adresse contenue dans eax
```

Question 6 Quelle est la valeur de `eax` à la fin de la suite d'instructions suivante en supposant que la mémoire est dans l'état de notre exemple ?

```
mov eax,0
add eax,1
mov al,[eax]
mov ah,[eax]
```

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

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

Par exemple,

```
mov eax,0x00000102 ;
mov [0],al         ; écrit l'octet contenu dans al (qui vaut 2) à l'adresse 0
mov [2],ax         ; écrit les deux octets de ax à l'adresse 2 et 3
mov [0],eax        ; écrit les quatre octets de eax aux adresses 0, 1, 2 et 3
```

Question 7 Quel est l'état de la mémoire aprés l'exécution de la suite d'instruction ?

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` (1 octet), `word` (2 octets) et `dword` (4 octets).

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

Question 8 Quel est l'état de la mémoire après avoir effectué chacune de ces instructions en partant de la mémoire de l'exemple ci-dessus ?

Question 9 Donnez une suite d'instructions telle qu'après l'avoir exécutée, `al`vaut 1 si le processeur suit la convention *little endian* et 0 si le processeur suit la convention *big endian*.

2 Premier programme

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

`http://www-igm.univ-mlv.fr/ens/IR/IR1/2009-2010/Archi/index.php`

Pour compiler le programme, on exécute:

```
nasm -g -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 10 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 le 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 système 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.

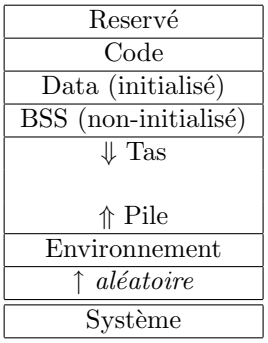
Question 11 Pour les appels systèmes `write` et `exit`, donnez le nombre d'arguments et expliquez leurs rôles.

Question 12 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 caractère terminée par un charactère 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 Environment d'exécution

L'environment 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 `0x08048000`. L'adresse du début de l'`Environnement` est dans le registre. L'adresse à laquelle finie le bloc `Système` est `0xBFFF FFFF`.



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

Question 13 Comment est codée l'instruction `mov ecx,msg` ? Quelle semble être l'adresse de `msg` dans ce codage ? Cela est-il cohérent avec Environment d'exécution décrit précédement ?

Question 14 En utilisant la commande `xxd hello.o`, vous obtenez l'affiche binaire du fichier object `hello.o`. Retrouvez votre code et vos données. Pourquoi le code ne peut-il pas être copié en mémoire directement.

Question 15 En utilisant la commande `xxd hello`, vous obtenez l'affiche binaire du fichier exécutable. Retrouvez votre code et vos données. Quel est le codage final de l'instruction `mov ecx,msg` ?

4 Débugger avec Gdb

Nous allons utiliser un front-end graphique appeler INSIGHT qui permet de suivre pas à pas l'exécution d'un programme et de connaitre les valeurs de la mémoire et des registres .

L'exercice 2 du TP qui se trouve à l'adresse suivante:

`http://www-igm.univ-mlv.fr/~pivoteau/ARCHI/TD/archi_tp1.pdf`

vous donne toutes les informations pour utiliser INSIGHT.

Question 16 Utilisez INSIGHT pour suivre l'exécution de `hello.asm`.

Question 17 Expliquez le comportement du programme `myst.asm`.