Travaux Pratiques n°2

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][TP2] 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. Si un fichier que vous rendez n'a pas le comportement attendu, cela doit être dit dans le rapport.

Dans la séance précédente, nous avons vu comment écrire un programme simple entièrement en assembleur. En particulier, nous avons vu comment utiliser les appels système.

À partir de cette scéance, nous n'écrirons plus nos programmes intégralement en assembleur. Nous écrirons uniquement une fonction en assembleur et cette fonction sera appellée par un programme `C`.

Téléchargez les fichiers `add.asm`, `asm_io.inc`, `asm_io.o` et `driver.c`. Le fichier `add.asm` contient le code assembleur d'une fonction nommée `asm_main`. Le fichier `driver.c` est un programme `C` qui a pour seul but d'appeller la fonction `asm_main`.

Regardons le fichier `add.asm`.

1. La première ligne permet d'avoir accés à des fonctions d'entrée-sortie plus faciles à utiliser que les appels système.

   ```
   %include "asm_io.inc"
   ```

2. Le code commence et termine par des opérations ayant pour but de respecter la convention d'appel du `C`. Le sens précis sera vu dans une autre séance.

   ```
   enter   0,0               ; mise en place
   pusha
                .
                .
                .
   popa
   mov     eax, 0            ; retour vers le C
   leave
   ret
   ```

3. Cette fonction fait des appels à des fonctions comme `print_nl`, `print_string` et `print_int`. Le code de ces fonctions se trouve dans `asm_io.o`.

   - `call print_nl` affiche un retour à la ligne

- `call print_int` affiche l'entier (signé) contenu dans `eax`.
- `call print_string` affiche la chaîne terminée par un octet null dont l'adresse est dans `eax`.
- `call read_int` lit un entier signé au clavier et le stocke dans `eax`.

Pour le compiler, on utilisera les commandes suivantes:

```
nasm -f elf add.asm
gcc -o add add.o asm_io.o driver.c
```

Question 1 Compilez le programme et examinez les différents fichiers.

Pour faire vos propres programmes, vous pourrez utiliser le squelette `base.asm` et le script `comp`. Pour compiler `toto.asm`, tapez `./comp toto`.

Pour une description détaillée des différentes instructions nécessaires dans la suite, reportez vous au livre *PC Assembly Language*.

Question 2 Écrire un programme qui lit deux entiers au clavier et affiche le maximum.

Pour vous aider, le tableau ci-dessous (pris dans *PC Assembly Language*) donne les différents branchements conditionnels et leur sens après l'instruction suivante:

cmp vleft,vright

Signé			Non signé		
JE	saute si vleft	= vright	JE	saute si vleft	= vright
JNE	saute si vleft	$\neq$ vright	JNE	saute si vleft	$\neq$ vright
JL, JNGE	saute si vleft	$<$ vright	JB, JNAE	saute si vleft	$<$ vright
JLE, JNG	saute si vleft	$\leq$ vright	JBE, JNA	saute si vleft	$\leq$ vright
JG, JNLE	saute si vleft	$>$ vright	JA, JNBE	saute si vleft	$>$ vright
JGE, JNL	saute si vleft	$\geq$ vright	JAE, JNB	saute si vleft	$\geq$ vright

Question 3 Écrire un programme qui lit deux nombres a et b au clavier et qui affiche `Oui` si b divise a et `Non` sinon. Vous pourrez utiliser `div ebx` qui réalise la division du nombre `edx:eax` par `ebx`. Le quotient va dans `eax` et le reste de `edx`. Quand on parle du nombre `edx:eax`, on désigne le nombre de 64 bits formé en concatenant `edx` et `eax`.

Que ce passe-t-il si votre programme prend $a = -1$ et $b = 17$ en entrée ? Expliquez ce comportement en cherchant des informations sur `DIV`.

Question 4 Écrire un programme qui lit un entier positif au clavier et affiche tous les entiers qui divisent ce nombre. Par exemple si le nombre lu est `60`, le programme affichera:

```
1 2 3 4 5 6 10 12 20 30 60
```

Question 5 Écrire un programme qui lit un entier est affiche sa décomposition binaire sur 32 bits. Par exemple, si le nombre lu est 10, le programme affichera:

```
00000000000000000000000000001010
```

On pourra utiliser la commande `shl reg,1` qui décale les bits de `reg` de 1 vers la gauche et qui stocke le bit qui est sorti dans la retenue (drapeau `CARRY`)

Vous pourrez utilisez les sauts conditionnels `JC` et `JNC` qui sautent si la retenue est à 1 ou 0 respectivement.

Question 6 Écrire un programme qui affiche le nombre de bits de `eax` qui valent 1. Par exemple si `eax` vaut `0x0000F00F`, le programme renverra 8.

L'idée est dans un premier temps d'utiliser l'instruction `shr` vue à la question précédente.

Question 7 On cherche une autre manière de réaliser le programme de la question précédente. En traitant quelque exemple, décrivez ce que l'on obtient si l'on réalise le ET logique entre `eax` et `eax`-1. En déduire un autre programme assembleur calculant le nombre de bits valant 1 dans `eax`.

Quel est l'avantage de cette approche ?

Question 8 Récupérez le programme `jdst.asm`. Décrivez la fonction de ce programme et expliquez son fonctionnement. Ne décrivez pas ligne par ligne mais identifiez des blocs et donnez leur sens.