Architecture des ordinateurs

ESIPE - IR1

TP 2

Objectif de cette séance

L'objectif de cette séance est de réaliser des programmes simples en assembleur utilisant des sauts conditionnels. Pour faciliter les entrées/sorties, nous allons utiliser une bibliothèque de fonctions développée par Paul Carter. Cette bibliothèque et son utilisation seront présentées dans la première partie du TP. La seconde partie donne une introduction sur les sauts conditionnels.

Cette fiche est à faire en une séance, et en binôme. Il faudra :

1. Réaliser un – bref – rapport à rendre **au format pdf** contenant les réponses aux questions de cette fiche ;
2. Écrire les – différents – fichiers sources des programmes demandés. Veillez à nommer correctement vos fichiers sources.
3. Réaliser une archive **au format zip** contenant les sources des programmes et le rapport. Le nom de l'archive doit être sous la forme `IR1_Archi_TP1_NOM1_NOM2`.
4. Envoyer le tout à l'adresse de votre chargé de TP, par un courriel dont le sujet est sous la forme

   `IR1 Archi TP2` $NOM1$ `;` $NOM2$

   Les sources des programmes doivent **impérativement** être des fichiers compilables par Nasm.

Tous les fichiers nécessaires pour ce TP se trouvent à l'adresse

`http://www-igm.univ-mlv.fr/~pons/ens/2012-IR1/`.

1 La bibliothèque `asm_io`

Pour utiliser cette bibliothèque, il faut télécharger et copier dans le répertoire de travail les fichiers `asm_io.o` et `asm_io.inc`. Cette bibliothèque fournit plusieurs fonctions : `print_string`, `print_int`, `read_int`, `print_nl`, `print_espace`.

- `print_string` affiche sur la sortie standard la chaîne de caractères (terminée par un octet de valeur 0) dont l'adresse est contenue dans `eax`.

- `print_int` affiche sur la sortie standard l'entier signé contenu dans `eax`.
- `read_int` lit sur l'entrée standard un entier signé et l'enregistre dans `eax`.
- `print_nl` affiche sur la sortie standard un retour à la ligne.
- `print_espace` affiche un espace sur la sortie standard.

Nous allons voir comment les utiliser sur un exemple. Le programme `Add.asm` demande à l'utilisateur de saisir deux nombres et ensuite affiche leur somme.

```
1       %include "asm_io.inc"
2
3       SECTION .data
4       prompt1 : db "Entrer un nombre : ", 0
5       prompt2 : db "Un autre nombre : ", 0
6       outmsg1 : db "La somme est ", 0
7
8       SECTION .bss
9       input1 : resd 1
10      input2 : resd 1
11
12      SECTION .text
13      global  main
14      main :
15          mov eax, prompt1      ; Affichage de prompt1.
16          call print_string
17          call read_int         ; Lecture d'un entier.
18          mov [input1], eax     ; Le stocke à l'adresse input1.
19          mov eax, prompt2      ; Affiche prompt2.
20          call print_string
21          call read_int         ; Lecture d'un entier.
22          mov [input2], eax     ; Le stocke à l'adresse input2.
23          mov eax, [input1]     ; eax = dword @ input1
24          add eax, [input2]     ; eax += dword @ input2
25          mov ebx, eax          ; ebx = eax
26          mov eax, outmsg1
27          call print_string     ; Affichage de outmsg1.
28          mov eax, ebx
29          call print_int        ; Affichage de la somme.
30          call print_nl         ; Affichage d'une nouvelle ligne.
31          mov ebx, 0
32          mov eax, 1
33          int 0x80
```

Pour compiler le programme, taper les lignes suivantes :

```
1       nasm -g -f elf Add.asm -F dwarf
2       ld asm_io.o Add.o -e main -o Add
3       chmod +x Add
```

Question 1. Que fait la ligne 24 de `Add.asm` ?

Remarque 1. L'inclusion de la bibliothèque se fait avec la ligne `%include "asm_io.inc"`. Pour faire appel à ces fonctions, il faut écrire : `call print_int`, `call print_nl`, etc.

Remarque 2. La section `bss` permet de réserver de la mémoire initialisée à 0. Par exemple, `resd 10` réserve 10 `dword` valant 0. L'instruction `resb` permet de réserver des octets plutôt que des `dword`. Écrire `resd 5` au début la section `bss` est équivalent à écrire `dd 0,0,0,0,0` à la fin de la section `data`.

Important 1. La section `bss` peut servir à stocker l'équivalent de variables globales.

Remarque 3. La bibliothèque `asm_io` utilise un tampon de 1000 octets. Si le nombre de caractères entrés dans un programme dépasse 1000 les résultats sont indéfinies. Pour en savoir plus, les sources de la bibliothèque sont consultables et se trouvent dans `asm_io.asm`.

Important 2. Le tampon n'est pas automatiquement vidé quand vous quittez votre programme. Il faut donc penser à faire un appel à `print_nl` avant de sortir du programme.

2 Les sauts inconditionnels/conditionnels

La forme la plus simple de saut est le saut inconditionnel. La syntaxe est :

```
1       jmp label
```

Cette instruction saute à l'adresse `label`.

Les sauts conditionnels ne sont réalisés que sous certaines conditions. Ces conditions dépendent de la valeur des drapeaux du processeur. Par exemple, `jc` saute si le drapeau `Carry` est à 1 et passe à la ligne suivante sinon.

Une façon simple d'utiliser les sauts conditionnels est en conjonction avec l'instruction `cmp`. Par exemple,

```
1       cmp eax, 0
2       je fin
3       mov ebx, ecx
```

Si `eax` est égal à 0, le programme saute au label `fin` et sinon il continue à l'instruction suivante, c'est à dire `mov ebx, ecx` dans cet exemple.

Voici une liste des sauts conditionnels les plus courants et leurs effets :

	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

Table 1: Sauts conditionnels usuels sur l'instruction `cmp` vleft, vright.

Question 2. Quel est l'affichage produit par le programme ci-dessous ?

```
1               mov eax, 0xFFFFFFFF
2               cmp eax, 0
3               jg aff_1
4               mov eax, 0
5               call print_int
6       aff_1 : mov eax, 1
7               call print_int
```

Question 3. Quel est l'affichage produit si l'on remplace `jg` par `ja` ?

3 Un peu de pratique

Astuce 1. Il est possible d'utiliser le fichier `Base.asm` comme base pour vos propres programmes et le script `Comp` pour les assembler. Taper simplement `./Comp Base` pour assembler le programme `Base.asm`.

Remarque 4. Il est possible que vous ayez besoin de rendre le script `Comp` executable. Pour cela tapez `chmod +x Comp`.

Question 4. Écrire un programme `Q4.asm` qui lit deux entiers au clavier et affiche le maximum.

L'instruction `div` sert à diviser deux nombres. Elle prend un unique argument : le diviseur. La quantité à diviser est toujours le nombre de 64 bits obtenu en prenant les octets de `edx` (poids fort) puis ceux de `eax` (poids faible). Le quotient est stocké dans `eax` et le reste est stocké dans `edx`.

Question 5. Donner les valeurs de `eax` et de `edx` après les instructions suivantes :

```
1           mov edx, 0x00000000
2           mov eax, 0x000005DE
3           mov ebx, 15
4           div ebx
5
```

Question 6. Écrire un programme `Q6.asm` qui lit deux nombres a et b au clavier et qui affiche `Oui` si b divise a et `Non` sinon. Si la réponse est négative, le reste de la division sera affiché.

Question 7. Que se passe-t-il si ce programme prend $a := -1$ et $b := 17$ en entrée ?

Question 8. Réaliser un programme `Q8.asm` qui marche avec les entiers signés en utilisant `idiv`.

Question 9. Écrire un programme `Q9.asm` 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 15 20 30 60
```

Question 10. Écrire un programme `Q10.asm` qui lit un entier et 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 du registre *reg* de 1 vers la gauche et qui stocke le bit qui est sorti dans la retenue (drapeau `Carry`). Il est possible d'utiliser les sauts conditionnels `JC` et `JNC` qui sautent si la retenue est à 1 ou 0 respectivement.

Remarque 5. Le second opérande de l'instruction `shl` peut être le registre `cl` et uniquement ce registre. Dans ce cas on écrit `shl reg,cl`.

Question 11. Écrire un programme `Q11.asm` qui demande un entier et affiche le nombre de bits de cet entier qui valent 1. Par exemple si l'entier vaut `0x0000F00F`, le programme renverra 8.

Astuce 2. L'idée est dans un premier temps d'utiliser l'instruction `shr`, analogue à l'instruction `shl`, mais qui décale cette fois-ci les bits du registre de 1 vers la droite.

Question 12. On cherche une autre manière de réaliser le programme de la question précédente. En traitant manuellement quelque exemples, décrire ce que l'on obtient si l'on réalise le *ET* logique (instruction `and`) entre `eax` et `eax` -1. En déduire un autre programme `Q12.asm` calculant le nombre de bits valant 1 dans `eax`. Quel est l'avantage de cette approche ?

Question 13. Écrire un programme `Q13.asm` qui lit en entrée une suite d'entiers compris entre 0 et 50 et terminée par -1 et affiche ensuite la liste dans l'ordre croissant des entiers entre 0 et 50 qui ne sont pas apparus dans la liste d'entrée. Par exemple, si la liste tapée en entrée est

```
4 1 15 1 2 -1,
```

Le programme affichera

```
0 3 5 6 7 8 9 10 11 12 13 14 16 17 18 ... .
```

Astuce 3. Il est possible de réserver dans la section `bss` 51 octets qui serviront à se souvenir des nombres qui ont été vus.

Bonus 1. Réaliser ce programme sans faire de lecture/écriture en mémoire. Vous L'apppelerez `B1.asm`.