Architecture des ordinateurs

Fiche de TP 2

ESIPE – INFO 1 2020–2021

Entrées/sorties et sauts

Table des matières

Cette fiche est à faire en une séance (soit 2 h, sans compter le temps de travail personnel), et en binôme. Il faudra

1. réaliser un rapport soigné à rendre **au format pdf** contenant les réponses aux questions de cette fiche (exercices marqués par ■), une introduction et une conclusion ;
2. écrire les différents fichiers sources des programmes demandés. Veiller à nommer correctement les fichiers sources. Ceux-ci doivent **impérativement** être des fichiers compilables par Nasm ;
3. réaliser une archive **au format zip** contenant les fichiers des programmes et le rapport. Le nom de l'archive doit être sous la forme `AO-TP2_NOM1_NOM2.zip` où `NOM_1` et `NOM_2` sont respectivement les noms de famille des deux binômes dans l'ordre alphabétique ;
4. déposer l'archive sur la plate-forme de rendu.

Tous les fichiers complémentaires nécessaires à ce TP se trouvent sur le site

https://igm.univ-mlv.fr/~giraudo/Teaching/AO/AO.html

L'objectif de ce TP 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.

1 La bibliothèque d'entrée/sortie `asm_io`

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

1. `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 ;
2. `print_int` affiche sur la sortie standard l'entier signé contenu dans eax ;
3. `read_int` lit sur l'entrée standard un entier signé et l'enregistre dans eax ;
4. `print_nl` affiche sur la sortie standard un retour à la ligne ;
5. `print_espace` affiche une espace sur la sortie standard.

Nous allons voir comment les utiliser sur un exemple. Le programme `Add.asm` de la figure 1 demande à l'utilisateur de saisir deux nombres et affiche ensuite leur somme.

Pour compiler le programme, nous employons les trois commandes

```
1nasm -f elf32 asm_io.asm
2nasm -f elf32 Add.asm
3ld -o Add -melf_i386 -e main Add.o asm_io.o
```

Il est à noter que la 1re ligne permet l'obtention de `asm_io.o` et que celle n'est à exécuter qu'une unique fois dans toute la suite : en effet, une fois `asm_io.o` obtenu, il est inutile de le générer à nouveau pour simplement l'utiliser.

Exercice 1. ■ En observant le programme `Add.asm`, expliquer

1. ce que fait la ligne 17 ;
2. ce que fait la ligne 24 ;
3. ce que fait la ligne 29 ;
4. ce que font les lignes 31, 32 et 33.

Remarque.

1. L'inclusion de la bibliothèque se fait avec la ligne `%include "asm_io.inc"`. Pour faire appel aux fonctions qu'elle fournit, il faut écrire : `call print_int`, `call print_nl`, *etc.* Le mot réservé `call` est très utile en assembleur et sera étudié dans un prochain TP.
2. La bibliothèque `asm_io` utilise un tampon de 1000 octets. Si le nombre de caractères entrés dans un programme dépasse 1000, les résultats sont indéfinis. Pour en savoir plus, les sources de la bibliothèque sont consultables et se trouvent dans `asm_io.asm`.
3. Le tampon n'est pas automatiquement vidé quand l'exécution du programme s'arrête. Ainsi, il se peut que des instructions de sortie réalisant des affichages ne soient pas effectivement visibles sur la sortie. Il faut donc penser à faire un appel à `print_nl` avant de sortir du programme, instruction qui a pour effet de vider le tampon.

```
%include "asm_io.inc"

SECTION .data
prompt1 : db "Entrer un nombre : ", 0
prompt2 : db "Un autre nombre : ", 0
outmsg1 : db "La somme est ", 0

SECTION .bss
input1 : resd 1
input2 : resd 1

SECTION .text
global main
main :
    mov eax, prompt1
    call print_string   ; Affichage de prompt1.
    call read_int       ; Lecture d'un entier.
    mov [input1], eax
    mov eax, prompt2
    call print_string   ; Affichage de  prompt2.
    call read_int       ; Lecture d'un entier.
    mov [input2], eax
    mov eax, [input1]
    add eax, [input2]
    mov ebx, eax
    mov eax, outmsg1
    call print_string   ; Affichage de outmsg1.
    mov eax, ebx
    call print_int      ; Affichage de ?
    call print_nl       ; Affichage d'une nouvelle ligne.
    mov ebx, 0
    mov eax, 1
    int 0x80
```

FIGURE 1 – Le programme `Add.asm`.

4. 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`. La section `bss` peut ainsi servir à stocker l'équivalent de variables globales.

2 Les sauts inconditionnels/conditionnels

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

```
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 `CF` (« Carry Flag ») 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, dans

```
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.

La table 1 recense les sauts conditionnels 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	$\leqslant$	vright	`jbe, jna`	saute si vleft	$\leqslant$	vright
`jg, jnle`	saute si vleft	$>$	vright	`ja, jnbe`	saute si vleft	$>$	vright
`jge, jnl`	saute si vleft	$\geqslant$	vright	`jae, jnb`	saute si vleft	$\geqslant$	vright

TABLE 1 – Sauts conditionnels usuels sur l'instruction `cmp` vleft, vright.

Exercice 2. ■ Considérons la suite d'instructions

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

1. Expliquer ce qu'elles affichent en précisant ce que fait chaque étape de l'exécution.
2. Reprendre la question précédente en en considérant la suite d'instructions obtenue en remplaçant `jg` par `ja` en ligne 3.

Remarque. Pour l'exercice précédent, mais aussi pour les prochains dans ce TP et dans les suivants, prendre l'habitude de programmer et d'exécuter les suites d'instructions dont le comportement est à expliquer. Les fonctions d'entrée/sortie permettent d'afficher et d'exploiter les valeurs calculées.

3 Mise en pratique

Il est possible d'utiliser le fichier `Base.asm` comme base pour vos propres programmes et le script `Comp` pour les assembler. Il suffit maintenant de saisir la commande

```
./Comp Base
```

pour assembler le programme `Base.asm`. La compilation fournie par ce script nécessite la présence de `asm_io.o` dans le répertoire courant. Il faut de plus que le script soit exécutable ; si ce n'est pas le cas, le rendre exécutable par

```
chmod +x Comp
```

Exercice 3. Écrire un programme `E3.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 (en bits de poids forts) puis ceux de eax (en bits de poids faibles). Le quotient est écrit dans eax et le reste est écrit dans edx.

Exercice 4. Donner les valeurs de eax et de edx après les instructions

```
mov edx, 0x00000000
mov eax, 0x000005DE
mov ebx, 15
div ebx
```

Exercice 5.

1. Écrire un programme `E5.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.
2. Expliquer ce qu'il se passe lorsque ce programme prend $a := -1$ et $b := 17$ en entrée.
3. Améliorer le programme précédent en un programme `E5+.asm` qui fonctionne correctement avec les entiers signés en utilisant `idiv` à la place de `div`. Pour cela, on peut utiliser l'instruction `cdq` (« Convert Double word to Quad word ») pour étendre le signe de `eax` dans `edx:eax`.

Exercice 6. ■ Écrire un programme `E6.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
```

Exercice 7. ■ Écrire un programme `E7.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* d'un pas vers la gauche et qui récupère le bit qui est sorti dans le drapeau de retenue `CF`. Il est possible d'utiliser les sauts conditionnels `jc` et `jnc` qui sautent si la retenue est à `1` ou `0` respectivement.

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

Exercice 8. Écrire un programme `E8.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. L'idée est dans un premier temps d'utiliser l'instruction `shr`, analogue à l'instruction `shl` *reg*, `1`, mais qui décale cette fois-ci les bits du registre d'un pas vers la droite.

Exercice 9. On cherche une autre manière de réaliser le programme de l'exercice précédent. 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 `E9.asm` calculant le nombre de bits valant 1 dans eax. Expliquer l'avantage de cette approche par rapport à celle de l'exercice précédent.

Exercice 10. ■ Écrire un programme `E10.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  ...  49 50.
```

Astuce. Il est possible de réserver 51 octets dans la section `data` qui serviront à se souvenir des nombres qui ont été vus. Chaque octet en position `i` va contenir `1` ou `0` suivant si l'entier `i` a été vu ou non.

Exercice 11. Écrire une nouvelle version `E11.asm` du programme de l'exercice précédent dans lequel on s'interdit toute lecture/écriture en mémoire. On n'utilise ainsi que les registres.

Astuce. Un registre est une suite de 32 bits qui peut être utilisée pour représenter l'absence ou la présence de tout entier compris entre `0` et `31`. Par exemple, le fait que le bit d'indice 3 soit à `1` et le bit d'indice 9 soit à `0` dans `eax` signifie que `eax` représente un ensemble d'entiers contenant 3 mais par 9.