TP : architecture des ordinateurs
EISC 1

2018-2019

Introduction à la programmation en assembleur

Ce TP reprend en partie ceux de Samuele Giraudo[*]

Vous avez vu comment fabriquer à partir de transistors et de condensateurs une mémoire avec son système d'adressage et une unité arithmétique et logique (UAL). Vous savez donc réaliser physiquement des instructions élémentaires comme : lire et écrire en mémoire, additionner, comparer, ... Le but de ces TP (5 séances de 2 heures) est de vous montrer comment on peut à partir de ses instructions élémentaires réaliser des programmes informatiques. Ses instructions seront écrites dans le langage assembleur Nasm pour des processeurs 32 bits d'architecture x86.

1 Premier programme

1.1 Structure d'un programme Nasm

Un programme en Nasm a la structure suivante :

```
global main             ; declaration de main en global.

segment .data
    ; declarations des variables  initialisees.
segment .bss
    ; declarations des variables non  initialisees.
segment .text
    ; vos fonctions.

main :                  ; point d'entre du programme.
    ; instructions de votre programme

    mov ebx, 0          ; code de sortie du programme.
    mov eax, 1          ; numero de la commande exit.
    int 0x80            ; interruption Linux : le programme rend la main au systeme.
```

[*]Voir https://igm.univ-mlv.fr/~giraudo/Enseignements

Les trois dernières lignes indiquent la fin du programme, elles devront apparaître dans tous vos programmes. À votre avis **quelle** est l'utilité du point virgule en Nasm ? **Créez** un fichier `Hello.asm`. **Tapez** le programme ci-dessous qui affiche le traditionnel message `Hello world`.

```
global main

segment .data
msg :
    db "Hello World", 10 ; declaration de la chaine de caracteres a afficher,
                         ; 10 est le code ASCII du retour a la ligne.

segment .text

main :
    ; l'affichage necessite 3 arguments
    mov ebx, 1           ; arg1, numero de la sortie pour l'affichage,
                         ; 1 est la sortie standard.
    mov ecx, msg         ; arg2, adresse du message a afficher.
    mov edx, 12          ; arg3, nombre de caracteres a afficher.

    ; demander au systeme d'afficher
    mov eax, 4           ; numero de la commande d'affichage
    int 0x80             ; interruption 0x80, appel au noyau.

    ; fin du programme
    mov ebx, 0           ; code de sortie, 0 est la sortie normale.
    mov eax, 1           ; numero de la commande exit.
    int 0x80             ; interruption 0x80, appel au noyau.
```

1.2 Compilation

Vous devez compiler votre programme avant de pouvoir l'exécuter. **Ouvrez** un terminal, se positionner dans le répertoire contenant le fichier `Hello.asm` et saisir les commandes

```
nasm -f elf32 Hello.asm
ld -o Hello -melf_i386 -e main Hello.o
```

La première commande crée un fichier objet `Hello.o` et la dernière réalise l'édition des liens pour obtenir un exécutable `Hello`. Pour exécuter le programme, il suffit d'utiliser la commande `./Hello`.

Exercice 1. ***Créez*** *et* ***exécutez*** *un programme qui affiche "Bonne journée !" sans retourner à la ligne. Votre programme devra s'appeler* `Bonjour.asm` *et son exécutable* `Bonjour`.

1.3 Registres

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. Parmi ces registres, les registres appelés eax, ebx, ecx, edx, edi et esi sont des registres à usage général. L'instruction mov (déjà utilisée dans le programme `Hello.asm`) permet d'affecter une valeur à un registre :

```
mov eax, 3          ; eax = 3
mov eax, 0b101      ; eax = 0b101 = 5
mov ebx, eax        ; ebx = eax
```

Il existe plusieurs formats pour spécifier une valeur. Par défaut, le nombre est donné en décimal. Une valeur préfixée par `0b` correspond à un nombre donné en binaire.

Exercice 2. *Quelle instruction place la valeur* $(1000111)_2$ *dans le registre* ecx *?*

2 Bibliothèque `asm_io`

Comme vous l'avez constaté avec le programme `Hello.asm`, l'affichage est un peu lourd en Nasm. Afin de se simplifier la vie, nous allons utiliser la bibliothèque `asm_io`. **Téléchargez** les fichiers `asm_io.asm` et `asm_io.inc` à l'adresse

http://igm.univ-mlv.fr/~giraudo/Enseignements/2018-2019/AO

et **ajoutez** les dans votre dossier de travail. Cette bibliothèque fournit plusieurs fonctions comme `print_string`, et `print_int`. Voici quelques explications :

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

2. `print_int` affiche l'entier signé contenu dans eax.

Nous allons voir comment les utiliser sur un exemple. **Testez** le programme ci-dessous que vous nommerez `test_biblio.asm`.

```
%include "asm_io.inc" ; importation de la bibliotheque
global main

section .data
chaine : db "Le registre eax contient : ", 0
retourligne : db 10, 0

section .text
main :
    mov eax, chaine
    call print_string   ; affichage de chaine.

    mov eax, 13
    call print_int      ; affichage de la valeur du registre eax.

    mov eax, retourligne
    call print_string   ; affichage d'une nouvelle ligne.
```

```
; fin du programme
mov ebx, 0
mov eax, 1
int 0x80
```

Pour compiler votre programme (qui utilise la bibliothèque) saisissez les commandes

```
nasm -f elf32 asm_io.asm
nasm -f elf32 test_biblio.asm
ld -o test_biblio -melf_i386 -e main test_biblio.o asm_io.o
```

Il est à noter que la 1re ligne permet l'obtention de `asm_io.o` et que celle-ci n'est à exécuter qu'une seule fois pour tout le TP : en effet, une fois `asm_io.o` obtenu, il est inutile de le générer à nouveau.

Exercice 3. ***Refaites*** *l'exercice 1 en utilisant la bibliothèque* `asm_io`*.*

Exercice 4. ***Faites*** *un programme qui affiche en décimal, avec un retour à la ligne, la valeur* $(10111)_2$*.*

3 Mémoire

3.1 Les sous-registres

Les registres eax, ebx, ecx et edx sont subdivisés en sous-registres. La figure 1 montre la subdivision de eax en ax, ah et al. Le premier octet (celui de poids le plus faible) de

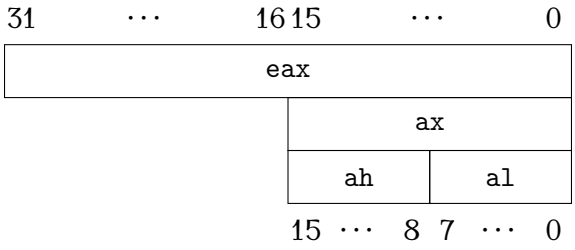


FIGURE 1 – Subdivision du registre eax.

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 (qui recouvre al et ah). Noter que les 2 octets de poids fort ne sont pas directement accessibles par un sous-registre. De même pour ebx, ecx, et edx, on dispose des registres analogues bx, bh, bl, cx, ch, cl et dx, dh, dl.

Important. Lorsque l'on modifie le registre al, les registres ax et eax sont eux aussi modifiés. En effet, `al` est physiquement une partie de `ax` qui lui-même est une partie de `eax`. Cette remarque est évidement valable pour les autres registres et leurs sous-registres.

Exercice 5. *Quelles sont les valeurs de* eax, ax, ah *et* al *après l'instruction* `mov eax, 134512768` *? Quelles sont ensuite les valeurs de* eax, ax, ah *et* al *après l'instruction* `mov al, 0` *?*

3.2 Adresse

La mémoire peut être vue comme un tableau de 2^{32} cases contenant chacune *un octet*. Le numéro (ou l'indice) d'une case est appelé son *adresse*. La mémoire est représentée par un tableau vertical dont les cases indexées de la plus petite adresse à la plus grande. Une adresse est codée sur 32 bits. Le contenu des registres 32 bits (comme eax, ebx, *etc.*) peut représenter un nombre ou adresse en mémoire.

La figure 2 illustre un exemple fictif d'état de la mémoire. Les adresses y sont notées en hexadécimal.

Adresse	Valeur
0x00000000	3
0x00000001	30
0x00000002	90
0x00000003	10
0x00000004	16
0x00000005	9
⋮	⋮
0x00000010	127
⋮	⋮
0xfffffffe	30
0xffffffff	3

FIGURE 2 – Exemple d'état de la mémoire.

Exercice 6. (Mémoire)

1. *Combien de valeurs différentes peut représenter une case de la mémoire ?*
2. *Quelle est la quantité de mémoire adressable sur 32 bits ?*
3. *Combien de cases se situent avant la case d'indice* `0x0000100a` *dans la mémoire ?*

3.3 Lecture en mémoire

La syntaxe générale pour lire la mémoire à l'adresse `adr` et enregistrer la valeur dans le registre `reg` est la suivante (les crochets font partie de la syntaxe) :

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

1

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

```
mov al, [0x00000003] ; al recoit l'octet stocke a l'adresse 3
                     ; dans l'exemple, al = 10 = 0x0a;
mov al, [3]          ; Instruction equivalente a la precedente.
```

Exercice 7. ***Expliquez*** *la différence entre* `mov eax, 3` *et* `mov eax, [3]`.

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

```
mov eax, 0     ; eax = 0
mov al, [eax]  ; al recoit l'octet situe a l'adresse contenue dans eax
               ; dans l'exemple, al = 0x03.
```

Exercice 8. *Dans l'exemple de mémoire de la figure 2,* ***donnez*** *les valeurs des (sous-)registres demandés, après les instructions suivantes (avant chaque sous-question,* `eax` *est supposé égal à 0) :*

1. ax *après l'instruction* `mov ax, [1]` ;
2. ah *après l'instruction* `mov ah, [0]` ;
3. eax *après l'instruction* `mov eax, [0]` ;
4. eax *après l'instruction* `mov eax, [1]`.

4 Unité Arithmétique et Logique

Voici un tableau récapitulant les principales opérations réalisées par l'UAL :

Instruction	Opérande 1	Opérande 2	Effet
`mov`	*dst*	*src*	recopie *src* dans *dst*
`xchg`	*ds*1	*ds*2	échange *ds*1 et *ds*2
`add`	*dst*	*src*	ajoute *src* à *dst*
`sub`	*dst*	*src*	soustrait *src* à *dst*
`imul`	*src*		multiplie `eax` par *src*
`idiv`	*src*		divise `edx`|`eax` par *src*, quotient : `eax`, reste : `edx`
`inc`	*dst*		place *dst* + 1 dans *dst*
`dec`	*dst*		place *dst* − 1 dans *dst*
`neg`	*dst*		place −*dst* dans *dst*
`not`	*dst*		place (not *dst*) dans *dst*
`and`	*dst*	*src*	place (*src* AND *dst*) dans *dst*
`or`	*dst*	*src*	place (*src* OR *dst*) dans *dst*
`xor`	*dst*	*src*	place (*src* XOR *dst*) dans *dst*
`shl`	*dst*	*nb*	décalage logique à gauche de *nb* bits de *dst*
`shr`	*dst*	*nb*	décalage logique à droite de *nb* bits de *dst*
`rol`	*dst*	*nb*	?
`ror`	*dst*	*nb*	?

Exercice 9. ***Testez*** *chacune de ces opérations. Par exemple, faites un programme qui place x et y dans des registres et affichent leur addition.*

Exercice 10. *Les intructions `not`, `and`, `or` et `xor` sont-elles des opérations bit-à-bit ?*

Exercice 11. *Que font les opérations `rol` et `ror` ?*

5 Les sauts

Jusqu'à présent, vos programmes réalisaient les instructions dans l'ordre ligne à ligne. Les sauts vous permettent de passer d'un endroit à un autre dans le code. Cela est, par exemple, utile pour réaliser des boucles. On distingue deux types de sauts : les sauts inconditionnels et les sauts conditionnels.

5.1 Les sauts inconditionnels

Vous pouvez placer dans votre programme des étiquettes comme ceci :

```
; ...
etiq_1 :
; ...
```

Il vous est ensuite permis d'aller à n'importe laquelle de ces étiquettes grâce à l'instruction

```
jmp etiquette
```

où `etiquette` est le nom de votre étiquette.

Exercice 12. ***Testez*** *le programme suivant*

```
    section .data
    bo : db "Bonjour", 0
    me : db "Merci", 0

    section .text
    main :
    mov eax, bo
    jmp etq

    mov eax, me
    etq :
    call print_string   ; Affichage de chaine.
```

Expliquez pourquoi il n'affiche pas `Merci`.

Exercice 13. ***Réalisez*** *un programme qui affiche avec retour à la ligne les nombres de 0 à l'infini. Comment interrompre le programme ?*

5.2 les sauts conditionnels

Les sauts conditionnels sont des sauts qui ne sont réalisés que sous certaines conditions. Une façon simple d'utiliser les sauts conditionnels est en conjonction avec l'instruction `cmp`. Par exemple, dans

```
cmp eax, 0
je etq
mov ebx, ecx
```

si eax est égal à `0`, le programme saute jusqu'à l'étiquette `etq` et sinon il continue à l'instruction suivante, c'est à dire `mov ebx, ecx` dans cet exemple. Il existe d'autres instructions de saut. Par exemple,

```
cmp eax, 0
jge etq
```

saute si eax est supérieur ou égal à 0.

Exercice 14. ***Réalisez*** *un programme qui affiche avec retour à la ligne les nombres de* 0 *à* 50*.*

Exercice 15. *Que font les instructions* `jl` *et* `jle` *?*

Exercice 16. *On rappelle que factorielle n est égale à* $n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$*.* ***Réalisez*** *un programme qui calcule factorielle* 10*.*

Exercice 17. *La suite de Fibonacci est définie par :* $f_0 = 0$*,* $f_1 = 1$ *et* $f_n = f_{n-1} + f_{n-2}$ *pour* $n \geq 2$*. Ses premiers termes sont* $0, 1, 1, 2, 3, 5, 8, 13, \ldots$ ***Réalisez*** *un programme qui affiche les* 30 *premiers termes de la suite de Fibonacci.*

Exercice 18. *On rappelle qu'un nombre est dit premier si ses seuls diviseurs sont* 1 *et lui-même.* ***Faites*** *un programme qui affiche avec retour à la ligne "*`x` *est premier" si le nombre* x *placé dans* ecx *est premier et "*`x` *n'est pas premier" sinon.*

6 Tableau

Vous pouvez déclarer un tableau dans la section `.bss` (voir la section 1) par la procédure suivante

```
section .bss
tab: resd 50
```

Cette instruction déclare un tableau `tab` de 50×4 octets. Attention : le tableau n'est pas initialisé pour autant. Pour affecter la valeur de eax à la case i du tableau, on procède ainsi

```
mov [tab + ((i-1)*4)], eax
mov dword [tab + ((i-1)*4)], 13
```

Exercice 19. ***Faites*** *un programme qui calcule et affiche la somme et la moyenne des éléments d'un tableau.*

Exercice 20. * *On suppose maintenant que les éléments du tableau sont des notes comprises entre* 0 *et* 20. ***Faites*** *un programme qui affiche les notes dans l'ordre croissant.*

7 La pile

La pile est une zone de la mémoire dans laquelle on peut empiler et dépiler des données. Pour empiler une valeur sur la pile, on peut procéder ainsi

```
push 10 ; empile la valeur 10
push eax ; empile la valeur contenue dans eax
push [eax] ; empile la valeur contenue a l'adresse indiquee par eax
```

Pour dépiler la dernière valeur ajoutée à la pile, on peut procéder ainsi

```
pop eax
```

Cette instruction extrait la dernière valeur ajoutée à la pile et la place dans le registre eax.

Exercice 21. ***Faites*** *un programme qui empile les valeurs* 1, 2 *et* 3 *puis les dépile. On affichera la valeur dépilée après chaque dépilement. Que constatez-vous ?*

Le registre esp contient l'adresse de la tête de la pile. Vous n'avez pas à le mettre à jour, le système le fait déjà pour vous ! La dernière valeur ajoutée à la pile est donc [esp].

Exercice 22. ***Affichez*** *la valeur de* esp *avant et après chaque instruction du programme suivant*

```
push 3
push 5
push 8
pop eax
pop eax
pop eax
```

Que constatez-vous ?

Exercice 23. ***Donnez*** *des suites d'instructions utilisant uniquement les instructions* `mov`, `sub` *et* `add` *afin de simuler l'instruction.*

```
push eax
```

Faites la même chose pour l'instruction.

```
pop ebx
```

Exercice 24. ***Faites*** *un programme qui :*

1. *empile* 9 *notes entre* 0 *et* 20,
2. *calcule et affiche la somme totale et la moyenne des notes,*
3. *dépile les notes.*

8 Fonction

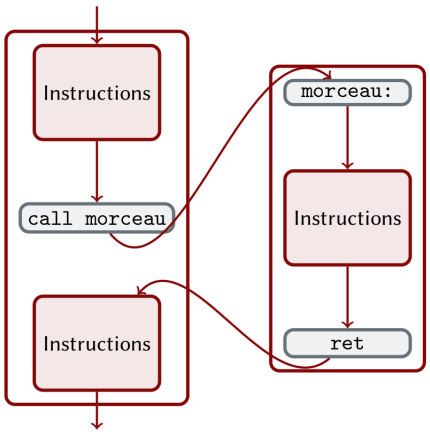
On peut déclarer une fonction `fonc` dans la zone `segment .text` (voir la section 1.1) ainsi

```
fonc :
; code de la fonction
ret
```

Pour appeler la fonction `fonc` on procède ainsi :

```
call fonc
```

Vous l'avez déjà fait avec les fonctions `print_string` et `print_int`. Voici un schéma montrant le comportement d'un programme qui appelle une fonction nommée `morceau` :



Exercice 25. ***Faites*** *une fonction qui réalise l'addition des registres* eax *et* ebx*. Votre programme devra appeler votre fonction et afficher le résultat.*

Vous avez passé à votre fonction les paramètres (les nombres à additionner) par les registres eax et ebx. Cependant, votre fonction est susceptible d'appeler une autre fonction (y compris elle-même) ou d'avoir plus que deux paramètres, **expliquez** quels problèmes cela peut engendrer ?

Afin de palier à ces problèmes, nous allons utiliser la pile selon la convention suivante (même convention que le langage C). Vos fonctions seront déclarées ainsi

```
fonc :
push ebp
mov ebp, esp
; code de la fonction
pop ebp
ret
```

et appelées ainsi

```
push ARG_N
...
push ARG_1
```

```
    call fonc

    add esp, 4*N
```

Exercice 26. *Que fait la fonction ci-dessous ?*

```
fonc :
    push ebp
    mov ebp, esp

    mov ebx, [ebp + 8]
    cmp ebx, 0
    je fin

    imul ebx

    dec ebx
    push ebx
    call fonc
    pop ebx

    fin :
    pop ebp
    ret
```

Exercice 27. ***Expliquez*** *les rôles de* ebp *et de l'instruction* `add esp, 4*N`*. Quelle est la différence entre* `jmp` *et* `call` *?*

Exercice 28. ***Réalisez*** *une fonction qui calcule la somme des n premiers termes $(n + (n-1) + \cdots + 2 + 1)$. Votre programme devra respecter la convention, appeler votre fonction et afficher le résultat.*

Exercice 29. *La fonction d'Ackermann à deux paramètres est définie par :*

$$A(m,n) = \begin{cases} n+1, & \text{si } m = 0 \\ A(m-1,1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m-1, A(m,n-1)) & \text{si } m > 0 \text{ et } n > 0 \end{cases}$$

Programmez *en respectant la convention la fonction d'Ackermann.* ***Calculez*** $A(2,2)$, $A(2,3)$, $A(4,1)$ *et* $A(4,3)$.

9 Problèmes

La bibliothèque `asm_io` (voir section 2) fournit également une fonction nommée `read_int`. En l'appelant vous pouvez récupérer dans le registre eax l'entier que l'utilisateur aura tapé au clavier. **Essayez !**

Problème 1. *(Jeu de la vie en une dimension) L'utilisateur entre une configuration de départ, c'est à dire un tableau binaire (ne contenant que des* 0 *et des* 1*), et un nombre d'itération. À chaque itération, vous transformerez et afficherez le tableau ainsi :*
si la case est dans l'une de ces deux conditions :

— *la case à la même valeurs que ses voisines de gauche et de droite,*

— *la case vaut* 0*, sa voisine de gauche vaut* 1 *et sa voisine de droite vaut* 0

alors elle est transformée (ou conservée) en 0*. Dans tous les autres cas, elle est transformée (ou conservée) en* 1*. On suppose que la case à gauche de la case d'indice* 0 *et la case à droite de la case d'indice "taille du tableau" valent* 0*. Soignez l'affichage !*

Problème 2. *(Jeu du plus ou moins)* ***Programmez*** *le jeu suivant : votre programme contient un nombre secret entre* 0 *et* 10000*. À chaque tour, l'utilisateur propose un nombre. Vous lui indiquerez si son nombre est plus grand ou plus petit que le nombre secret. L'utilisateur gagne si et seulement s'il trouve le nombre secret en au plus* 10 *tours.*

Problème 3. *Étant donné un entier N, on définit sa suite de Syracuse de la manière suivante :*

$$U_n = \begin{cases} N & \text{si } n = 0 \\ \frac{U_{n-1}}{2} & \text{si } n > 0 \text{ et } U_{n-1} \text{ est pair} \\ 3U_{n-1} + 1 & \text{sinon} \end{cases}$$

On conjecture que pour tout nombre, il existe un indice n tel que $U_n = 1$*.* ***Testez*** *la conjecture. Si vous trouvez un contre-exemple, signalez le discrètement à votre professeur...*

Problème 4. ***Faites*** *un programme qui propose des grilles de Sudoku, permette d'y jouer et propose une solution quand cela est possible.*

Instructions pour rendre le TP

Il faudra

1. réaliser un rapport soigné à rendre **au format pdf** contenant les réponses aux questions de cette fiche ;
2. écrire les 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 Nasmen `.asm` (ne pas inclure les fichiers compilés) et le rapport. Le nom de l'archive doit être sous la forme `Nom_prenom.zip` ;
4. envoyer l'archive à l'adresse `christophe.cordero@u-pem.fr` avant le 10 juin 23 h. L'objet de votre mail devra commencer par "[ESIPE]".