Annexe Assembleur

Un programme assembleur est un fichier texte d'extension .asm.
Compilation :
nasm -f elf64 votreprogramme.asm
ld votreprogramme.o -o exec
./exec

Structure d'un programme NASM

```
1  section .data
2      ; declaration des variables initialisées
3  section .bss
4      ; declaration des varaibles
5  section .text
6      global _start           ;declaration de _start en global
7
8  _start:     ; point d'entree du programme
9
10     ; instructions de votre programme
11
12 mov ebx, 0      ;code de sortie du programme
13 mov eax, 1      ;numero de la commande exit
14 int 0x80        ;interruption Linux: le programme rend la main au systeme.
```

Un programme NASM est composé de trois sections :

La section .data

- Déclaration de constantes (leur valeur ne changera pas durant l'exécution)
- Elle est constituée de lignes de la forme `étiquette pseudo-instruction valeur`
- Les pseudo instructions sont les suivantes :

db	define byte	Déclare un octet
dw	define word	Déclare deux octets
dd	define doubleword	Déclare quatre octets
dq	define quadword	Déclare huit octets
dt	Define tenbytes	Déclare dix octets

- **Exemples**
  - const1 db 1
  - const2 dw 123

La section .bss

- Déclaration de variables non initialisées
- Elle est constituée de lignes de la forme `étiquette pseudo-instruction nb`
- Les pseudo instructions sont les suivantes :

resb	Reserve byte	Déclare un octet
resw	Reserve word	Déclare deux octets
resd	Reserve doubleword	Déclare quatre octets
resq	Reserve quadword	Déclare huit octets
rest	Reserve tenbytes	Déclare dix octets

- **nb** represente le nombre d'octets (pour resb) de mots (pour resw) . . . à réserver

- **Exemples**
  - input1 resb 100 ; reserve 100 octets
  - wordvar resw 1 ; reserve un mot

La section .text

- Instructions qui composent le programme.
- Cette section doit commencer par déclarer global l'étiquette de début de programme (main) pour qu'elle soit visible :

```
SECTION .text
        global _start

_start :

….
```

- fin du fichier :

```
mov ebx, 0      ; code de sortie, 0= normal
mov eax, 1      ; numéro de la commande exit
int 0x80        ; appel au noyau
```

- Cette section est constituée de lignes de code de la forme

  Etiquette : instruction opérandes ; commentaire

Les commentaires commencent par « ; »

****

Un **syscall** (abréviation de system call), est, comme son nom l'indique, un appel au système, afin de lui faire effectuer une tâche définie. Cette tâche peut être de lire du texte, d'en écrire, de lire un fichier, d'exécuter un programme, etc...
Un syscall est une fonction, banale dans son utilisation, un peu différente dans sa structure, qui utilise différents arguments.

- Le numéro de la tâche (fonction) à effectuer se met dans le registre eax (voir tableau en bas).
- Le premier argument dans ebx, le second dans ecx, le troisième dans edx, etc...
- Puis on appelle le kernel, via le « kernel interrupt », int 80h

Prenons un exemple simple : écrire du texte.

- Le numéro du syscall write est 4.
- Le numéro de la sortie standard (votre écran) est 1, et il doit être le premier argument.
- La chaîne à écrire doit être le second argument.
- La taille de la chaîne est donc la troisième.

Eax	Name	Ebx	Ecx	edx
1	Sys_exit	Code de sortie		
3	Sys_read	Descripteur+	Char*	Nb octets à lire
4	Sys_write	Descripteur+	Char*	Nb octets à lire

Descripteur+ : 0=stdin , 1=stdout, 2=stderr
Avec ce que nous avons déjà vu, vous devriez être capable d'esquisser le code de ce syscall :

```
mov eax, 4
mov ebx, 1
mov ecx, chaine
mov edx, TailleChaine
int 80h
```

Assembleur

- **Copie : mov**

  mov destination source

- ❏ copie source vers destination
- ❏ source: adresse, constante ou register
- ❏ destination: un registre ou une adresse
- ❏ Les copies registre-registre sont possibles mais pas les copies mémoire-mémoire
- ❏ Lorsqu'on copie vers un registre ou depuis un registre, c'est la taille du registre qui indique le nombre d'octets copiés
- ❏ Lorsqu'on copie une constante en mémoire, il faut préciser le nombre d'octets à copier, à l'aide des mots clefs :
  - ❏ byte pour un octet
  - ❏ word pour deux octets
  - ❏ dword pour quatre octets
- ❏ **Exemple** : mov word [var], 1

- **Empiler : push**

Syntaxe push source

- ❏ copie le contenu de source au sommet de la pile
- ❏ commence par décrémenter esp de 4 puis effectue la copie
- ❏ source: adresse, constante ou registre

- **Dépiler: pop**

Syntaxe pop destination

- ❏ copie les 4 octets qui se trouvent au sommet de la pile dans destination
- ❏ commence par effectuer la copie puis incrémente esp de 4
- ❏ destination: un registre ou une adresse

NB : Si on travaille avec des registres de 32 bits

- **Addition : add**

Syntaxe add destination, source

- ❏ Effectue destination = destination + source
- ❏ destination: registre ou adresse
- ❏ source: constante, registre ou adresse
- ❏ modifie eventuellement les flags overflow (OF) et carry (CF)
- ❏ Les operations registre - registre sont possibles, mais pas les opérations mémoire – mémoire

- **Soustraction : Sub**

Syntaxe sub destination, source

- ❏ Effectue destination = destination - source
- ❏ destination: registre ou adresse
- ❏ source: constante, registre ou adresse
- ❏ modifie eventuellement les flags overflow (OF) et carry (CF)
- ❏ Les operations registre - registre sont possibles, mais pas les opérations mémoire - mémoire

- **Multiplication : mul**

Syntaxe mul source

- ❏ Effectue eax = eax * source
- ❏ La multiplication de deux entiers codes sur 32 bits peut nécessiter 64 bits.
- ❏ les quatre octets de poids de plus faible sont mis dans eax et les quatre octets de poids le plus fort dans edx (edx :eax).
- ❏ source: constante, registre ou adresse

- **Division : div**

Syntaxe div source

- ❏ Effectue la division edx: eax / source
- ❏ Le quotient est mis dans eax
- ❏ Le reste est mis dans edx
- ❏ source: constante, registre ou adresse

- **Opérations logiques**
  - and destination source
  - or destination source
  - xor destination source

- not destination

- ❏ Effectue les operations logiques correspondantes bit à bit
- ❏ Le resultat se trouve dans destination
- ❏ operandes :
  - ❏ source peut etre : une adresse, un registre ou une constante
  - ❏ destination peut etre : une adresse ou un registre

Comparaison: cmp

Syntaxe cmp destination, source

- ❏ Effectue l'opération destination - source
- ❏ Le résultat n'est pas stocké
- ❏ destination: registre ou adresse
- ❏ source: constante, registre ou adresse
- ❏ Les valeurs des flags ZF , SF et CF sont éventuellement modifiées.

Instruction de saut inconditionnel : JMP

Syntaxe

jmp adr

- ❏ Va à l'adresse/ étiquette adr

Instructions de saut conditionnel

	Instruction	Description	Indicateurs
Valeurs non signées	JB	Jump below	CF = 1
	JBE	Jump below or equal	CF = 1 et ZF=1
	JA	Jump above	CF =0 et ZF=0
	JAE	Jump above or equal	CF=0
	JE/JZ	Jump if equal /Jump if zero	ZF=1
	JNE/JNZ	Jump if not equal/ Jump if not zero	ZF=0
Valeurs signées	JL	Jump less	SF=NOT OF
	JLE	Jump less or equal	ZF=1 ou SF=NOT OF
	JG	Jump greater than	ZF=0 et SF=OF
	JGE	Jump greater or equal	SF=OF

Rappel: ZF (zero flag), SF (sign flag), CF (carry flag), OF (overflow flag)