Architecture des ordinateurs

Fiche de TP 1

ESIPE - IR/IG 1 2016-2017

Introduction à la programmation en assembleur

Table des matières

1 Représentation de données **2**
1.1 Représentation des entiers 2
1.2 Opérations en complément à deux 3
1.3 Représentation des flottants 4

2 Le processeur et les registres **4**
2.1 Registres et sous-registres 4
2.2 Opérations sur les registres 5
2.3 Lecture/écriture en mémoire 6

3 Premier programme **9**
3.1 Compilation 9
3.2 Explication du code 9
3.3 Assemblage du programme 12

Cette fiche est à faire en deux séances (soit 4 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, 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 `IR1_Archi_TP1_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

`http://igm.univ-mlv.fr/~giraudo/Enseignements/2016-2017/AO/AO.html`

Le livre de Paul Carter *PC Assembly Language* est disponible gratuitement à l'adresse

http://www.drpaulcarter.com/pcasm/

Il est important de le consulter au fur et à mesure de l'avancement du TP.

Le but de ce TP est d'introduire les bases de la programmation en assembleur sur des processeurs 32 bits d'architecture x86, sous Linux, et en mode protégé. L'assembleur Nasm (Netwide Assembler) sera utilisé en ce but.

La première partie du sujet est une étape préliminaire indispensable pour se familiariser avec la représentation des données en machine. Elle permet un entraînement utile pour tous les futurs TP. Dans la deuxième partie, nous abordons la notion de registre, les opérations sur les registres et les lectures/écritures en mémoire. La troisième partie présente un premier programme écrit en Nasm. Elle décrit comment compiler un programme et comprendre comment fonctionne sa traduction en langage machine.

1 Représentation de données

Ce TP commence en douceur avec des questions d'application directe du cours concernant les représentations de données (entiers non signés, entiers signés, flottants). Contrairement aux parties suivantes, il n'y aura aucun rappel ici. Il est donc utile de mener à bien cette partie en lisant le cours en parallèle.

Exercice 1. (Nombre de données)

1. *Dessiner un tableau à deux lignes dont la ligne du haut est indexée par les entiers allants de* 0 *jusqu'à* 16 *et la ligne du bas contient les valeurs* 2^0 *jusqu'à* 2^{16}*. Apprendre ce tableau par cœur.*
2. *Combien de données différentes peut-on coder sur* n *bits ?*
3. *Combien de données différentes peut-on coder sur* n *octets ?*
4. *Combien de données différentes peut-on coder sur* n Kio *?*
5. *Combien de données différentes peut-on coder sur* n Mio *?*
6. *Combien de données différentes peut-on coder sur* n Gio *?*

1.1 Représentation des entiers

Exercice 2. (Changements de base)

1. *Représenter la valeur* $(999)_{\text{dix}}$ *en base deux, en base quatre et enfin en base seize. L'exécution de l'algorithme de changement de base doit être illustrée pas à pas.*
2. *Représenter la valeur* $(1002)_{\text{trois}}$ *en base dix.*

Exercice 3. (Binaire et hexadécimal)

1. *Représenter la valeur* $(\texttt{101010111111000000000001})_{\text{deux}}$ *en hexadécimal en utilisant la méthode de conversion rapide traduisant chaque suite de quatre bits en une lettre hexadécimale.*
2. *Représenter la valeur* $(\texttt{2BABA101F9})_{\text{seize}}$ *en binaire en utilisant la méthode de conversion rapide traduisant chaque lettre hexadécimale en une suite de quatre bits.*

Exercice 4. (Représentation avec biais)

1. *Représenter sur* $n := 8$ *bits avec un biais de* $k := 1000$ *la valeur* $(-950)_{\text{dix}}$ *en représentation avec biais.*
2. *Représenter sur* $n = 16$ *bits avec un biais de* $k := 256$ *la valeur* $(0)_{\text{dix}}$ *en représentation avec biais.*
3. *Donner la valeur décimale représentée par la suite de bits* $(\texttt{00000000})_{\text{biais}=21}$.
4. *Donner la valeur décimale représentée par la suite de bits* $(\texttt{0000000000010001})_{\text{biais}=17}$.
5. *Donner la valeur décimale représentée par la suite de bits* $(\texttt{00010001})_{\text{biais}=1777}$.

Exercice 5. (Représentation en complément à deux)

1. *Représenter sur* $n := 8$ *bits la valeur* $(17)_{\text{dix}}$ *selon la représentation en complément à deux.*
2. *Représenter sur* $n := 8$ *bits la valeur* $(-1)_{\text{dix}}$ *selon la représentation en complément à deux.*
3. *Représenter sur* $n := 16$ *bits la valeur* $(-1021)_{\text{dix}}$ *selon la représentation en complément à deux.*
4. *Expliquer s'il est possible de représenter la valeur* $(-1000)_{\text{dix}}$ *en complément à deux sur* $n := 8$.
5. *Donner la valeur décimale représentée par la suite de bits* $(\texttt{00001001})_{\text{c2}}$ *sur* $n := 8$ *bits.*
6. *Donner la valeur décimale représentée par la suite de bits* $(\texttt{10001001})_{\text{c2}}$ *sur* $n := 8$ *bits.*
7. *Donner la valeur décimale représentée par la suite de bits* $(\texttt{11001011})_{\text{c2}}$ *sur* $n := 8$ *bits.*

1.2 Opérations en complément à deux

Exercice 6. (Dépassement de capacité et retenue de sortie)

Considérons les valeurs suivantes sur $n := 8$ *bits :*

— $v_1 := (\texttt{10011100})_{\text{deux}}$;
— $v_2 := (\texttt{00000101})_{\text{deux}}$;
— $v_3 := (\texttt{10001000})_{\text{deux}}$;
— $v_4 := (\texttt{11111001})_{\text{deux}}$;
— $v_5 := (\texttt{00110101})_{\text{deux}}$;
— $v_6 := (\texttt{11100000})_{\text{deux}}$;
— $v_7 := (\texttt{01001111})_{\text{deux}}$;
— $v_8 := (\texttt{01011001})_{\text{deux}}$;
— $v_9 := (\texttt{10000000})_{\text{deux}}$;
— $v_{10} := (\texttt{11000000})_{\text{deux}}$.

Réaliser, en les posant, les additions suivantes et pour chacune d'elle, dire s'il y a une retenue sortante et/ou un dépassement de capacité.

1. $v_1 + v_2$;
2. $v_3 + v_4$;
3. $v_5 + v_6$;
4. $v_7 + v_8$;
5. $v_9 + v_{10}$;
6. $v_1 + v_{10}$.

Les additions posées font partie de la réponse et doivent être utilisées pour justifier les résultats fournis.

1.3 Représentation des flottants

Exercice 7. (Flottants)

1. *Représenter la valeur* $(52.40625)_{\text{dix}}$ *en représentation à virgule fixe où la troncature est représentée sur* $n := 8$ *bits et la partie décimale est représentée sur* $m := 7$ *bits.*
2. *Donner le codage* IEEE 754 SINGLE *de la valeur de la question précédente.*

2 Le processeur et les registres

2.1 Registres et sous-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. La figure 1 répertorie les noms des tailles de données habituelles en Nasm. Parmi ces registres, les registres appelés `eax`, `ebx`, `ecx`, `edx`,

Type	Taille en octets	Taille en bits
`byte`	1	8
`word`	2	16
`dword`	4	32

FIGURE 1 – Taille des types de données usuels en Nasm.

`edi` et `esi` sont des registres à usage général. Les registres `esp` et `eip` servent respectivement à conserver l'adresse du haut de la pile et l'adresse de l'instruction à exécuter. Ces derniers registres seront étudiés et exploités dans un prochain TP. Les registres `eax`, `ebx`, `ecx` et `edx` sont subdivisés en sous-registres. La figure 2 montre la subdivision de `eax` en `ax`, `ah` et `al`. Le premier octet (celui de poids le plus faible) de `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.

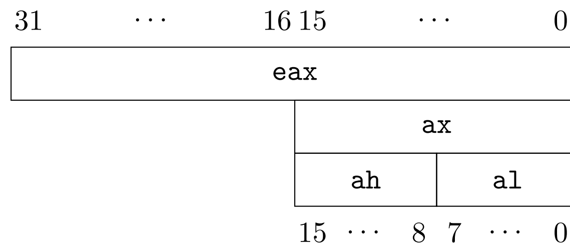

Figure 2 – Subdivision du registre `eax`.

Exercice 8. *Quelles sont les valeurs minimales et maximales qui peuvent être enregistrées respectivement dans les registres* `eax`, `ax`, `ah` *et* `al` *? Les valeurs données doivent être exprimées en décimal non signé, en décimal signé et en hexadécimal (non signé).*

2.2 Opérations sur les registres

Nous allons présenter les opérations permettant d'affecter une valeur à un registre (instruction `mov`) et d'effectuer des calculs (par exemple, instructions `add` et `sub`).

Important. Dans la syntaxe de Nasm, c'est toujours le premier argument qui reçoit la valeur du calcul. C'est la syntaxe Intel (il existe d'autres assembleurs qui emploient la convention inverse, on parle alors de la syntaxe ATT). Voici quelques exemples d'écriture dans des registres :

```
mov eax, 3              ; eax = 3
mov eax, 0x10a          ; eax = 0x10a = 266
mov eax, 0b101          ; eax = 0b101 = 5
mov ebx, eax            ; ebx = eax
mov ax, bx              ; ax = bx
```

Remarque.
1. Il existe plusieurs formats pour spécifier une valeur. Par défaut, le nombre est donné en décimal. Une valeur préfixée par `0x` correspond à un nombre donné en hexadécimal. Ainsi `0x1A` correspond en décimal à 26. Le préfixe `0b` correspond au binaire.
2. On ne peut pas copier la valeur d'un registre dans un registre de taille différente. Ainsi, l'instruction `mov eax, bx` produira une erreur.

Astuce. Google est utile pour faire facilement les conversions. Par exemple, la recherche « `0x1a in decimal` » donnera « `0x1a = 26` ».

Astuce. Connaissant la valeur de `eax` en **hexadécimal**, il est facile de connaître la valeur de `ax`, `ah` et `al`. Par exemple, si `eax = 0x01020304` alors `ax = 0x0304`, `ah = 0x03` et `al = 0x04`. C'est pour cette raison que la plupart du temps, les contenus des registres sont donnés en hexadécimal.

Exercice 9. *Quelles sont les valeurs (en hexadécimal) de* `eax`*,* `ax`*,* `ah` *et* `al` *après l'instruction* `mov eax, 134512768` *? Quelles sont ensuite les valeurs (en hexadécimal) de* `eax`*,* `ax`*,* `ah` *et* `al` *après l'instruction* `mov al, 0` *?*

Exercice 10. *En utilisant uniquement l'instruction* `mov` *et des constantes exprimées en hexadécimal n'occupant pas plus de deux chiffres hexadécimaux, donner une suite d'instructions qui modifie la valeur de* `ebx` *de sorte qu'elle soit égale à* `0x0000ABCD`*.*

Les instructions `add` et `sub` ont le comportement attendu : le résultat de l'addition et de la soustraction est écrit dans le premier argument. En cas de dépassement de la capacité, la retenue est signalée par le *drapeau de retenue* du processeur (ceci sera vu en détail et exploité dans un prochain TP). Nous avons ainsi par exemple,

```
1 add eax, 3           ; eax = eax + 3
2 add ah, 0x1c         ; ah = ah + 28
3 add ebx, ecx         ; ebx = ebx + ecx
4 sub eax, 10          ; eax = eax − 10
5 add ah, al           ; ah = ah + al
```

Exercice 11. *Après l'exécution des instructions suivantes, quelles sont les valeurs des registres* `eax`*,* `ax`*,* `ah` *et* `al` *?*

```
1 mov eax, 0xf00000f0
2 add ax, 0xf000
3 add ah, al
```

2.3 Lecture/écriture en mémoire

En mode protégé, 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 3 illustre un exemple fictif d'état de la mémoire.

Exercice 12. (Mémoire)

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

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

FIGURE 3 – Exemple d'état de la mémoire.

2.3.1 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]
```

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 13. *Expliquer la différence entre* `mov eax, 3` *et* `mov eax, [3]`*.*

Quand on lit plus d'un octet, il faut adopter une convention sur l'ordre dans lequel les octets sont rangés dans le registre. Il existe deux conventions *little-endian* (*petit boutisme*) et *big-endian* (*grand boutisme*). La convention employée dépend du processeur. Pour nous se sera toujours *little-endian*.

Considérons par exemple l'instruction suivante, avec la mémoire dans l'état représenté par la figure 3 :

```
mov eax, [0x00000000]
```

Le nombre d'octets lus dépend de la taille du registre. Ici on va lire les 4 octets situés aux adresses 0, 1, 2 et 3. Dans l'exemple de mémoire, ces octets valent respectivement 3 (= `0x03`), 30 (= `0x1e`), 90 (= `0x5a`) et 10 (= `0x0a`). Les deux choix possibles pour les ranger dans `eax` sont

- `eax` = `0x0a5a1e03` en convention *little-endian* ;
- `eax` = `0x031e5a0a` en convention *big-endian*.

Important. Les processeurs Intel et AMD utilisent la convention *little-endian*. Dans cette convention, les octets situés aux adresses basses deviennent les octets de poids faible du registre.

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,

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

Exercice 14. *Dans l'exemple de mémoire de la figure 3, donner les valeurs des (sous-)registres demandés, après les instructions suivantes (avant chaque sous-question,* `eax` *est supposé égal à* `0x0`*) :*

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]`*.*

Exercice 15. *Quelle est la valeur de* `eax` *à la fin de la suite d'instructions suivante, en supposant que la mémoire est dans l'état de la figure 3 ?*

```
1 mov eax, 5
2 sub eax, 1
3 mov al, [eax]
4 mov ah, [eax]
```

2.3.2 Écriture en mémoire

La syntaxe générale pour écrire en mémoire à l'adresse `adr` la valeur du registre `reg` est la suivante (les crochets font partie de la syntaxe) :

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

L'écriture suit la même convention que la lecture (*little-endian* en ce qui nous concerne).

Exercice 16. *Quel est l'état de la mémoire après l'exécution de la suite d'instructions qui suit ?*

```
1 mov eax, 0x04030201
2 mov [0], eax
3 mov [2], ax
4 mov [3], al
```

On peut aussi directement affecter une valeur en mémoire sans la stocker dans un registre. Il faut alors préciser la taille des données avec les mots-clés `byte` (1 octet), `word` (2 octets) et `dword` (4 octets).

Exercice 17. *Quel est l'état de la mémoire après avoir effectué la suite d'instructions suivante ?*

```
1 mov dword [0], 0x020001
2 mov byte [1], 0x21
3 mov word [2], 0x1
```

Exercice 18. *Quelle est la valeur de* `eax` *à la fin de la suite d'instructions suivante dans la convention* little-endian *et* big-endian *?*

```
1 mov ax, 0x0001
2 mov [0], ax
3 mov eax, 0
4 mov al, [0]
```

3 Premier programme

3.1 Compilation

Notre premier programme `Hello.asm` affiche le traditionnel message `Hello world`. Pour le compiler, lancer un terminal, se positionner dans le répertoire contenant le fichier `Hello.asm` et saisir les commandes

```
1 nasm -f elf32 Hello.asm
2 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`. Par ailleurs,

— `-f elf32` est une option d'assemblage. Elle permet à Nasm de fabriquer du code objet 32 bit pour Linux.

— `-melf_i386` permet de faire en sorte que l'édition des liens puisse se faire sur un système 64 bits (à supprimer sur un système 32 bits).

Pour exécuter le programme, il suffit de lancer la commande `./Hello`.

3.2 Explication du code

Nous allons maintenant expliquer en détail la programme dont le contenu est donné ci-dessous.

```
1 ; Premier programme en assembleur
2
3 SECTION .data ; Section des donnees.
4 msg :
5     db "Hello World", 10 ; La chaine de caracteres a afficher ,
6                          ; 10 est le code ASCII du retour a la ligne.
7
8 SECTION .text            ; Section du code.
```

```
9  global main            ; Rend l' etiquette  visible  de l' exterieur.
10 main :                 ; Etiquette  pointant au debut du programme.
11     mov edx, 0xc       ; arg3, nombre de caracteres a  afficher
12                        ; (equivalent  a mov edx, 12).
13     mov ecx, msg       ; arg2, adresse du premier caractere
14                        ; a  afficher.
15     mov ebx, 1         ; arg1, numero de la  sortie  pour l' affichage ,
16                        ; 1 est  la  sortie  standard.
17     mov eax, 4         ; Numero de la commande write pour
18                        ; l' interruption 0x80.
19     int 0x80           ; Interruption 0x80, appel au noyau.
20     mov ebx, 0         ; Code de sortie , 0 est la  sortie  normale.
21     mov eax, 1         ; Numero de la commande exit.
22     int 0x80           ; Interruption 0x80, appel au noyau.
```

3.2.1 Structure du programme

Le programme est composé de deux sections :

Section .data (lignes 3 – 6) contient les données du programme, dans cet exemple, la chaîne de caractères à afficher.

Section .text (lignes 8 – 23) contient le code du programme.

Lors de son exécution, le programme est chargé en mémoire comme décrit dans la figure 4. L'adresse à laquelle débute le code est toujours la même : `0x08048080`. En revanche, l'adresse

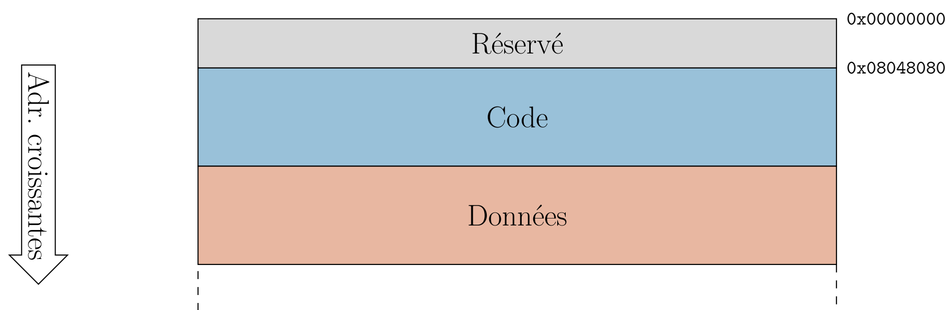

FIGURE 4 – Mémoire occupée par le programme lors de son exécution.

à laquelle commence les données dépend de la taille du code (plus le code est long, plus l'adresse de la case à laquelle commence les données est élevée).

Important. Lors de l'écriture d'un programme, l'adresse à laquelle est stockée la chaîne de caractères n'est pas connue. Il est difficile de la connaître car c'est le système qui l'attribue. C'est pour remédier à ce problème que l'on met l'étiquette `msg` au début de la ligne 2. Dans le programme, `msg` représentera l'adresse du premier octet la chaîne de caractères.

3.2.2 Section .data

La primitive `db` permet de déclarer une suite d'octets. La chaîne de caractères donnée entre guillemets correspond simplement à la suite des octets ayant pour valeur le code ASCII des caractères. Par exemple, le code ASCII de `'H'` est 72 et celui de `'e'` est 101. On aurait pu écrire, de manière équivalente,

```
1 msg : db 72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100, 10
```

ou bien encore

```
1 msg : db 0x48,0x65,0x6c,0x6c,0x6f,0x20,0x57,0x6f,0x72,0x6c,0x64,0x0a
```

3.2.3 Section .text

La section `.text` contient le code. L'instruction de la ligne 9 permet de rendre visible l'étiquette `main`, et en particulier, dans `Hello.o`. On peut ainsi y faire référence dans la commande `ld -o Hello -melf_i386 -e main Hello.o` pour signaler que le point d'entrée du programme est cette étiquette.

Le premier bloc, allant de la ligne 11 à la ligne 19, permet d'afficher la chaîne de caractères et le deuxième bloc, allant de la ligne 20 à la ligne 22 permet de quitter le programme.

Il n'y a pas dans le jeu d'instructions du processeur d'instruction spécifique pour afficher un message. Pour afficher un message à l'écran, il faut faire appel au système d'exploitation (dans notre cas, le noyau Linux). C'est le rôle de l'instruction `int 0x80`. Cette opération va passer la main au noyau pour réaliser une tâche. Une fois cette tâche réalisée, le noyau reprend l'exécution du programme à la ligne suivant l'interruption. Ce procédé est appelé un *appel système*. L'instruction `int 0x80` peut servir à faire toutes sortes de tâches : quitter le programme, supprimer un fichier, agrandir la taille du tas, exécuter un autre programme. Le noyau détermine la tâche à accomplir en regardant la valeur des registres `eax`, `ebx`, *etc.*, qui jouent alors le rôle de paramètres. En particulier, la valeur du registre `eax` correspond au numéro de la tâche à accomplir. Par exemple, `1` correspond à `exit` et `4` correspond à `write`. Pour connaître le numéro de chacun des appels systèmes, il faut regarder dans les sources du noyau. Par exemple pour le noyau 2.6, on le trouve dans le fichier `unistd_32.h` qui commence par les lignes suivantes :

```
1 #define __NR_restart_syscall 0
2 #define __NR_exit 1
3 #define __NR_fork 2
4 #define __NR_read 3
5 #define __NR_write 4
6 #define __NR_open 5
7 #define __NR_close 6
8 #define __NR_waitpid 7
9 #define __NR_creat 8
10 #define __NR_link 9
11 #define __NR_unlink 10
12 #define __NR_execve 11
13 #define __NR_chdir 12
14 #define __NR_time 13
15 #define __NR_mknod 14
16 #define __NR_chmod 15
17 #define __NR_lchown 16
18 #define __NR_break 17
```

Pour l'appel système `write`, les registres `ebx`, `ecx`, `edx` sont utilisés comme suit :

— `ebx` contient le numéro de la sortie sur laquelle écrire. La valeur `1` correspond à la sortie standard.

— `ecx` contient l'adresse mémoire du premier caractère à afficher

— `edx` contient le nombre total de caractères à afficher.

Si l'on revient sur le premier bloc du code (lignes 11 – 19), on demande au système d'afficher (car `eax = 4`) sur la sortie standard (car `ebx = 1`) les 12 caractères (car `ecx = 12`) commençant à l'adresse `msg`. Le deuxième bloc (ligne 20 – 22) quitte le programme avec l'appel système `exit` qui a pour code `1`. Le code de retour est donné dans `ebx`. La valeur `0` signifiant qu'il n'y a pas eu d'erreur.

Exercice 19. *Modifier* `Hello.asm` *pour afficher* « `L'ordinateur va s'eteindre !` ». *Le nouveau programme doit s'appeler* `E19.asm`.

Pour connaître les paramètres attendus par un appel système, il est possible d'utiliser la section 2 des pages de `man`. Par exemple, il faut taper `man 2 write` pour l'appel système `write`. La documentation donne le prototype de l'appel système pour le langage C. On peut l'adapter à l'assembleur en sachant que le premier argument va dans `ebx`, le second `ecx`, et ainsi de suite.

3.3 Assemblage du programme

Dans ce paragraphe, nous allons voir en détail comment le code de l'exécutable est assemblé en partant du code source du programme. En utilisant la commande

```
1 nasm  Hello.asm -l Hello.lst -f elf32
```

on obtient dans `Hello.lst` un listing donnant le code machine des différentes instructions. Le voici (sans ses commentaires) :

```
1   1
2   2
3   3                                   SECTION .data
4   4                                   msg :
5   5 00000000 48656C6C6F20576F72-          db "Hello World", 10
6   6 00000009 6C640A
7   7
8   8
9   9                                   SECTION .text
10 10                                   global main
11 11                                   main :
12 12 00000000 BA0C000000                   mov edx, 0xc
13 13
14 14 00000005 B9[00000000]                 mov ecx, msg
15 15
16 16 0000000A BB01000000                   mov ebx, 1
17 17
18 18 0000000F B804000000                   mov eax, 4
19 19
20 20 00000014 CD80                         int 0x80
21 21 00000016 BB00000000                   mov ebx, 0
22 22 0000001B B801000000                   mov eax, 1
23 23 00000020 CD80                         int 0x80
```

À cette étape, on connaît les octets correspondant à la section `.data`. En hexadécimal, cela donne : 48656C6C6F20576F726C640A.

Les octets correspondant aux différentes instructions sont connus. Seules les adresses des étiquettes ne sont pas connues ; pour l'instant, l'adresse `msg` ne l'est pas. En particulier, l'instruction `mov ecx, msg` est codée par `B9[00000000]`. Les crochets signifient que l'adresse n'est pas encore finalisée.

On remarque aussi que chaque instruction est précédée de son adresse. On peut par ailleurs déduire la taille en octets d'une instruction en regardant la 3e colonne du `Hello.lst` puisque celle-ci contient son codage en langage machine. Par exemple, en ligne 18, l'instruction `mov eax, 4` commence à l'adresse `0x0F` mémoire et se traduit en `0xB804000000`. Elle tient donc sur cinq octets.

Ceci étant dit, mesurons la taille de notre code. La dernière instruction est à l'adresse `0x20` et tient sur deux octets. La dernière adresse utilisée par le code est donc `0x21`, ce qui représente 33 en décimal. Notre code s'étend donc sur $34 = 33 - 0 + 1$ octets[1]. Cependant, pour des raisons d'alignement, tout code doit toujours occuper un nombre d'octets multiple de 4. Notre code occupera 36 octets (les deux octets de remplissage valent `0`).

Le calcul de l'adresse des étiquettes est réalisé par `ld`. Reprenons la figure précédente. Le bloc de données va commencer à l'adresse : `0x08048080` + 36 = `0x080480a4`. L'adresse `msg` est donc `0x080480a4`. L'instruction `mov ecx, msg` sera codée par `B9a4800408`.

Tout ceci mis bout à bout, on obtient que la traduction en langage machine de `Hello.asm` donne

```
BA 0C 00 00 00 B9 a4 80 04 08
BB 01 00 00 00 B8 04 00 00 00
CD 80 BB 00 00 00 00 B8 01 00
00 00 CD 80 00 00  48 65 6C 6C
6F 20 57 6F 72 6C 64 0A
```

Exercice 20. *Commenter les différences (éventuelles) entre les fichiers* `Hello.lst` *et* `E19.lst` *en les expliquant.*

Exercice 21. *Écrire un programme* `E21.asm` *respectant les consignes suivantes. Dans la section de données du programme, il doit être défini une étiquette* `debut` *sur la chaîne de caractères «* `Le premier TP` *» ainsi qu'une étiquette* `fin` *sur la chaîne de caractères «* `est presque termine !` *». Par deux appels système, le programme doit afficher la chaîne adressée par* `debut` *puis celle adressée par* `fin`*. La terminaison du programme doit être gérée de manière propre.*

Exercice 22. *Fournir le listing* `E21.lst` *du programme* `E21.asm` *et l'étudier pour calculer la taille du code machine du programme. Le raisonnement doit apparaître dans la réponse.*

1. Ceci provient du fait que pour compter le nombre de cases situées entre deux cases d'indices i et j, on utilise la formule $j - i + 1$.