

# Notes du laboratoire de microprocesseur (MIC2-micl)

Nathan Furnal

1<sup>er</sup> juin 2021

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Laboratoire 1 : Prise en main</b>	<b>3</b>
2.1	Registres	3
2.2	Langage d'assemblage	3
2.3	Exécution et déboguage	3
<b>3</b>	<b>Laboratoire 2 : Instructions logiques et de manipulation de bits</b>	<b>4</b>
3.1	Valeurs booléennes	4
3.2	Registre rflags	4
3.3	Instructions logiques	4
3.3.1	not	4
3.3.2	and, or, xor	4
3.3.3	Masquage	4
3.4	Instruction de manipulation de bits : bt, bts, btr, btc	5
3.5	Exercices	5
3.5.1	Exercice 1	5
3.5.2	Exercice 2	5
3.5.3	Exercice 3	5
3.5.4	Exercice 4	5
<b>4</b>	<b>Laboratoire 3 : Instructions de saut et alternatives</b>	<b>6</b>
4.1	Comparaison <code>cmp</code>	6
4.2	Label	7
4.3	Branchement	7
4.3.1	Saut inconditionnel <code>jmp</code>	7
4.3.2	Sauts conditionnels <code>jf</code> et <code>jnf</code>	7
4.3.3	Applications	7
4.4	Alternative	8
4.4.1	Alternative si	8
4.4.2	Alternative si-sinon	8
4.5	Exercices	9
4.5.1	Exercice 1	9
4.5.2	Exercice 2	9
4.5.3	Exercice 3	9
4.5.4	Exercice 4	10
4.5.5	Exercice 5	10
<b>5</b>	<b>Laboratoire 4 : Variables globales</b>	<b>11</b>
5.1	Sections dédiées aux variables	11
5.2	Accès à une variable	12
5.2.1	Boutisme	12
5.3	Amnésie de <code>nasm</code>	13
5.3.1	Problème de taille	13
5.3.2	Problème de section	13
5.4	Type de données	14
5.4.1	Entier	14
5.4.2	Flottant	14
5.4.3	Caractère	14
5.5	Exercices	14

5.5.1	Exercice 1 : remplir les pointillés	14
5.5.2	Exercice 2	15
5.5.3	Exercice 3	15
5.5.4	Exercice 4	15
5.5.5	Exercice 5	16
5.5.6	Exercice 6	16
<b>6</b>	<b>Laboratoire 5 : Appels système</b>	<b>17</b>
6.1	Définition	17
6.2	Mise en œuvre	17
6.3	Registres non préservés	17
6.4	Numéro du service	18
6.5	Paramètre et retour	18
6.6	Entrée et sorties standards	19
6.7	Hello, World!	19
6.8	Cinq appels système	19
6.9	Exercices	20
6.9.1	Ouvrir un fichier	20
6.9.2	Ouverture et écriture	20
6.9.3	Convertir rsi en caractères	21
6.9.4	Afficher la parité	21
6.9.5	Écrire des fichiers	22
6.9.6	Stocker une taille de fichier	22
<b>7</b>	<b>Laboratoire 6 : Tableau et boucles</b>	<b>23</b>
7.1	Tableau	23
7.1.1	Section <code>.data</code> et <code>.rodata</code>	23
7.1.2	Section <code>.bss</code>	23
7.1.3	Chaîne de caractères	23
7.1.4	Accès au contenu	24
7.2	Modes d'adressage	24
7.2.1	Immédiat	24
7.2.2	Registre	24
7.2.3	Emplacement mémoire	24
7.3	Instructions <code>inc</code> et <code>dec</code>	24
7.4	Boucles	24
7.5	Déboguage et tableau	24
7.6	Exercices	24
7.6.1	Assigner des valeurs à des tableaux	25
7.6.2	Déclarer et remplir un tableau	25
7.6.3	Remplir un tableau inversé	26
7.6.4	Remplir un tableau par sauts de 2	26
7.6.5	Compter le nombre d'éléments pairs d'un tableau	27
7.6.6	Trouver les extrema d'un tableau	27
7.6.7	Afficher 10 premiers chiffres	28
7.6.8	Afficher les 10 premier chiffres depuis un tableau	29
7.6.9	Lire depuis le clavier et afficher	29
<b>8</b>	<b>Laboratoire 7 : Pile et variables locales</b>	<b>30</b>
8.1	Pile	30
8.1.1	Registres associés à la pile	30
8.1.2	Instructions d'accès à la pile	31
8.2	Instructions <code>ADD</code> et <code>SUB</code>	31
8.3	Exemples d'utilisation de la pile	31
8.3.1	Accès à la pile avec <code>PUSH</code> et <code>POP</code>	31
8.3.2	Accès à la pile sans <code>PUSH</code> ni <code>POP</code>	31
8.4	Variable locale	31
8.5	Exercices	32
8.5.1	Échanger le contenu de registre, via la pile	33
8.5.2	Échanger le contenu de variables via la pile	33
8.5.3	Créer une variable locale sur la pile et sans variable globale	33
8.5.4	Variable locale de dix <i>bytes</i> sur la pile	34
8.5.5	Modifier une variable locale sur la pile	35
8.5.6	Lecture de fichier et variables locales	36

# 1 Introduction

Dans ce laboratoire, on va étudier les instructions des microprocesseurs en 64 bits, plus particulièrement la famille x86. Le but est de pouvoir effectuer des opérations simples en langage assembleur et de les déboguer.

## 2 Laboratoire 1 : Prise en main

### 2.1 Registres

Tout d'abord on doit définir ce qu'est un **registre**. C'est une zone de mémoire au sein du microprocesseur qui est très rapide d'accès. Une autre notion importante est celle de **opcode** ou code opérateur. Les instructions notées en hexadécimal commencent par un **opcode** qui désigne le type d'opération que l'on exécute, suivies par l'endroit en mémoire sur lequel porte l'opération.

De ce fait, le nom du registre est important (`rax`, `rsi`, `r8`,...) car il désigne le type d'opération qu'on va effectuer ou le type d'opération concernée par une autre commande qu'on exécute. En effet, certains registres sont directement modifiés par l'utilisateur et d'autres sont modifiés par d'autres opérations ou programmes sans action directe de l'utilisateur.

Par exemple, le registre `rip` pour «register of instruction pointer», stocke l'adresse de l'instruction à exécuter *après* celle en cours d'exécution. D'autres registres seront détaillés plus tard.

### 2.2 Langage d'assemblage

On trouve les codes sources assembleur dans le [premier TD](#). On y voit surtout des instructions pour stocker des valeurs en mémoire et puis les déplacer de registre en registre.

Un code source en langage d'assemblage est constitué de directives au compilateur, d'instructions, d'identifiants, d'immédiats, de commentaires et d'étiquettes (*labels*).

```

1 ; comment
2 global main
3 section .text
4 main:
5     nop ; ne fait rien
6
7     ; met l'immédiat en hexadécimal dans le registre rax
8     ; grâce à l'instruction mov
9     mov rax, 0x1122334455667788
10
11    ; fin
12    mov rax, 60
13    mov rdi, 0
14    syscall

```

On peut voir le code `global main` qui est une directive, une information pour le compilateur qui n'est pas une instruction. Elle n'exécute pas directement du code mais informe sur *comment* le code être compilé. Dans le cas d'une instruction comme `mov rsi, -1` la destination est le registre `rsi` et la source est la valeur immédiate `-1`. Enfin, `mov` est le mnémotique qui décrit une opération.

Le programme se termine, sous GNU/Linux par l'instruction `syscall`, qui est un *appel système*, expliqué dans le cours théorique.

### 2.3 Exécution et débogage

Les parties suivantes expliquent comment rendre le fichier `.nasm` exécutable et comment le déboguer via un outil comme `kdbg`.

## 3 Laboratoire 2 : Instructions logiques et de manipulation de bits

TD sur les manipulations de bits.

### 3.1 Valeurs booléennes

Il y a deux valeurs booléennes, *Vrai* ou *Faux*. Un seul bit suffit pour déterminer la variable booléenne. On code *Vrai* par 1 et *Faux* par 0.

### 3.2 Registre rflags

Le registre `rflags` est un registre de 64 bits dont certains des indicateurs (drapeaux, *flags*). Son équivalent sur 32 bits est le registre `eflags`, c'est le registre des états des processeurs x86 32bits donc. Les 32 nouveaux bits de poids 32 à 63 dans `rflags` sont tous réservés. Les indicateurs fournissent de l'information sur le déroulement des processus en cours. On parle de *status flag* car ils permettent de savoir comment les opérations arithmétiques se sont déroulées.

**Carry Flag (CF)** indicateur de retenue, bit de rang 0 dans `rflags`.

**Zero flag (ZF)** indicateur de zéro, bit de rang 6 dans `rflags`.

**Sign flag (SF)** indicateur de signe, bit de rang 7 dans `rflags`.

**Overflow flag (OF)** indicateur de débordement, bit de rang 11 de `rflags`.

On ne peut pas accéder directement au contenu de `rflags` mais on peut y accéder indirectement via des instructions qui seront expliquées plus tard. De ce fait, on ne doit pas nécessairement connaître les positions précises des *flags*.

### 3.3 Instructions logiques

#### 3.3.1 not

L'instruction `not` n'a qu'un opérande qui joue le rôle de source et de destination en même temps, pour les registres ou les variables de 8 à 64 bits. Elle **inverse** tous les bits de son opérande.

```
1 al : 10011101b
2 not al
3 al : 01100010b
```

#### 3.3.2 and, or, xor

Les instructions `and`, `or`, `xor` ont deux opérandes : la *destination* à gauche de la virgule et la *source*, à droite. Ces opérandes peuvent être des registres ou des variables de 8 à 64 bits mais pas tous les deux des emplacements mémoire. En outre, la *source* peut être un immédiat (un nombre).

Ces instructions effectuent respectivement un et, ou et un ou exclusif logique bit à bit entre la source et la destination. Ils placent le résultat dans la destination sans modifier la source.

De plus, ces instructions modifient le *sign flag* qui reçoit le bit de rang plus élevé du résultat et le *zero flag* qui indique si le résultat est nul : 1 pour nul et 0 pour non nul. Le *carry flag* et le *overflow flag* sont mis à zéro. Voici quelques exemples. Les tables résumées sont dans le [document du TD](#).

Attention, les immédiats s'étendent sur maximum 32 bits. Si on utilise un immédiat avec une destination dont la taille fait 64 bits, l'immédiat est étendu sur 64 bits par extension de signe : son bit de signe, celui de rang 31, est recopié en bits 32 à 63.

#### 3.3.3 Masquage

Le masquage consiste à effectuer une opération logique afin de conserver certains bits d'un opérande et d'en modifier d'autres. Ils s'effectuent avec `and`, `or` et `xor`. Pour utiliser un masque, on utilise une des instructions ou le second opérande est un masque qu'on désire utiliser. Comme toujours les tables récapitulatives sont le document du TD2.

##### 1. and

L'instruction `and` permet de conserver certains bits d'un opérande et **mettre les autres à zéro**.

##### 2. or

L'instruction `or` permet de conserver certains bits d'un opérande et **mettre les autres à zéro**.

##### 3. xor

L'instruction `xor` permet de conserver certains bits d'un opérande et d'**inverser les autres**.

### 3.4 Instruction de manipulation de bits : bt, bts, btr, btc

L'instruction **bt** signifie *bit test* et test un bit précis d'un motif binaire donné. Les instructions **bts** (*bit test and set*), **btr** (*bit test and reset*) et **btc** (*bit test and complement*), testent également un bit *avant* de le mettre à 1, 0 ou de le complémenter, respectivement.

Le premier opérande de ces instructions est un registre ou une variable de 16, 32 ou 64 bits. Le second opérande est un registre de 16, 32 ou 64 bits ou un immédiat sur 8 bits. Si le deuxième opérande est un registre, il doit être de même taille que le premier.

Ces quatre instructions copient dans le *carry flag* le bit du premier opérande dont le rang est fourni via le second opérande. C'est la partie *test* à laquelle se limite **bt**.

Ensuite l'instruction :

- **bts** met ce bit du premier opérande à 1.
- **btr** met ce bit du premier à 0.
- **btc** complémente ce bit du premier opérande.

Le *zero flag* n'est pas modifié. L'*overflow flag* et le *sign flag* sont indéfinis.

### 3.5 Exercices

#### 3.5.1 Exercice 1

Application directe des masques et manipulation

#### 3.5.2 Exercice 2

```
1  ; exercice 2 du td 2 de M1CL
2
3  global main
4  section .text
5  main:
6      ;; 'd' : 0110 0100b
7      ;; and : 0101 1111b
8      ;;      0100 0100b
9      ;; 'D' : 0100 0100b
10     mov ah, al
11     and ah, 01011111b
12
13     ; fin
14     mov rax, 60
15     mov rdi, 0
16     syscall
```

#### 3.5.3 Exercice 3

```
1      ;; exercice 3 du td 02
2
3  global main
4  section .text
5  main:
6      mov bx, 'd' ; charge le caractère 'd' dans le registre al
7      btc bx, 5   ; complémentaire du bit en position 5
8
9      ;; fin
10     mov rax, 60
11     mov rdi, 0
12     syscall
```

#### 3.5.4 Exercice 4

```

1      ;; exercice 4 du td 02 de MICL
2
3  global main
4  section .text
5  main:
6      mov bl, 00000101b      ; 5 en binaire
7      ;; On veut donner la valeur en ascii
8      ;; masque `or` pour passer en ascii
9      or bl, 00110000b
10     ;; fin
11     mov rax, 60
12     mov rdi, 0
13     syscall

```

## 4 Laboratoire 3 : Instructions de saut et alternatives

Ce TD présente L'instruction `cmp` et la notion de *label* est abordée. Ensuite, on traite de l'instruction `jmp` qui permet de faire un saut sans condition. Finalement, les instructions de saut conditionnel, dépendant de la valeur des *flags* `jf` et `jnf` sont montrées ainsi que l'implémentation des alternatives *si* et *sinon*.

### 4.1 Comparaison `cmp`

L'instruction `cmp` a deux opérandes de même taille. Ils peuvent être des registres ou des variables de 8, 16, 32 ou 64 bits. Il ne peuvent cependant pas tous les deux être des variables. L'opérande de droite peut être un immédiat.

Cette instruction compare l'opérande de gauche et de droite. Elle positionne les *flags* du registre `rflags` comme le ferait une soustraction de ceux-ci mais ne modifie aucun de ses opérandes.

```

1      ;; Montrer l'usage de cmp avec des commandes simples
2  global main
3  section .text
4  main:
5      mov rax, 4
6
7      cmp rax, 4      ; ZF : 1 (4 - 4 == 0), SF: 0 (4 - 4 >= 0)
8      cmp rax, 5      ; ZF : 0 (4 - 5 != 0), SF: 1 (4 - 5 < 0)
9      cmp rax, 2      ; ZF : 0 (4 - 2 != 0), SF: 0 (4 - 2 >= 0)
10
11     ;; fin
12     mov rax, 60
13     mov rdi, 0
14     syscall

```

On peut donc voir que le *zero flag* et le *sign flag* sont affectés par la comparaison. Il faut noter que dans les cas plus extrêmes d'*overflow*, l'*overflow flag* est levé. Aussi, si une opération requiert plus de bits que la taille maximale du registre, le *carry flag* sera levé lui aussi.

Un immédiat peut être codé sur au plus 32 bits et dans le cas où il est comparé à un opérande de 64 bits, l'extension de signe rend en jeu. L'extension de signe se déroule de la manière suivante, avec un immédiat de 32 bits et un registre de 64 bits. Prenons une valeur de `0x80_80_80_80`.

- En binaire : `0x80_80_80_80` → `0b10000000100000001000000010000000`.
- On remarque que le bit de poids le plus fort est 1.
- On étend ce 1 sur les 32 nouveaux bits de poids fort dûs au passage du 32 au 64 bits, ce qui donne en binaire :  
`0b1111111111111111111111111111111110000000100000001000000010000000`
- On reconvertit en hexadécimal, on a bien : `0xFF_FF_FF_FF_80_80_80_80`.

Il faut noter que l'extension de signe n'arrive **pas** avec le mnémonique `mov`.

## 4.2 Label

UN *label* est un repère que le programmeur met dans le code afin de donner un nom à une ligne de programme. Pour signaler le début de programme, on utilise le label `main` par exemple. La définition d'un label se termine par le caractère `:` qui ne fait pas partie du nom du label. Il peut contenir des chiffres, des lettres ou un *underscore* mais doit commencer par une lettre, un *underscore* ou un point.

## 4.3 Branchement

En assembleur, les instructions sont exécutées les unes après les autres, dans l'ordre où elles se trouvent dans la mémoire centrale. On parle d'exécution **séquentielle**. On appelle *saut* ou *branchement* le fait de passer d'une instruction à une autre qui ne la suit pas directement en mémoire. C'est le cas lorsqu'on a des alternatives (*if, else*) ou bien des boucles (*for, while*).

Ce TD aborde les sauts inconditionnels (`jmp`) et les sauts conditionnels : `jf` et `jnf`.

### 4.3.1 Saut inconditionnel `jmp`

L'instruction `jmp` permet d'effectuer un saut vers un *label*. On parle de saut inconditionnel car il a lieu dans tous les cas, sans aucune condition. Cette instruction ne nécessite que le label de la position vers laquelle on veut sauter, et ne modifie aucun *flag*.

```
1  mov rax, 123
2  mov rbx, 150
3
4  jmp fin      ; saut inconditionnel vers la fin
5
6  mov rax, 0   ; jamais exécuté
7  mov rbx, 0   ; jamais exécuté
8
9  fin:
10 mov rax, 60
11 mov rdi, 0
12 syscall
```

Par ce saut vers le label, les instructions après le saut et avant le label ne sont jamais exécutées.

### 4.3.2 Sauts conditionnels `jf` et `jnf`

Les instructions de saut conditionnel permettent d'effectuer un saut si une certaine condition est vraie. Le `f` dans les expressions, représente un *flag*. Par exemple, `jc` représente un saut suivant le *carry flag* et `jnz` représente un saut dépendant du *zero flag*.

Le saut conditionnel `jf` permet de vérifier si un *flag* donné est à 1.

Le saut conditionnel `jnf` permet de vérifier si un *flag* donné est à 0.

### 4.3.3 Applications

Une fois qu'on a ces outils en main, on peut les utiliser pour sauter vers les instructions qui nous intéressent, en fonction du *flag* qu'on a choisi d'analyser.

Voici par exemple un test de parité, si le *carry flag* vaut 1, du au *bit test* alors on va vers la fin ; sinon on exécute l'instruction suivante.

```
1  bt rax, 0    ;; Test de parité, est ce que le premier bit vaut zéro
2  jc endIf
3  mov rbx, 5
4  endIf:
5  mov rcx, 12
```

Ici on teste l'infériorité, on compare `rax` à une valeur. Ensuite, si le *sign flag* est levé (pour les résultats négatifs) alors on va vers le saut, sinon on exécute l'instruction suivante.

```

1      cmp rax, -10
2      jns endIf
3      mov rbx, 5
4  endIf:
5      mov rcx, 12

```

Ici, on teste l'inégalité, on compare rax avec rdx. La comparaison induit une différence temporaire donc s'ils contiennent les mêmes valeurs, le *zero flag* sera levé et on pourra exécuter un saut et sinon, on va vers l'instruction suivante.

```

1      cmp rax, rdx
2      jz endIf
3      mov rbx, 5
4  endIf:
5      mov rcx, 12

```

## 4.4 Alternative

### 4.4.1 Alternative si

En fait, ce qui a été présenté au-dessus représente une alternative "si". Dans le cas où un *bit test* ou bien une comparaison est évalué, ils affectent les *flags* classiques comme le *carry flag*, *sign flag* ou *zero flag* on peut les utiliser pour aller vers le label désigné par `jmp`.

### 4.4.2 Alternative si-sinon

La logique est parfaitement identique quand on veut avoir deux branches, le *si* et le *sinon*. On utilisera deux labels avec chacun une instruction et l'exécution de l'un ou l'autre suivra de la condition.

Voici un exemple de test de parité, on insère la valeur du premier bit de rax dans le *carry flag*. Si ce bit vaut zéro (rax est pair) alors on n'exécute **pas** le saut vers `_sinon` et rbx vaut 5. Puis on va vers la fin. S'il s'avère ne pas être pair, on effectue l'instruction au label `_sinon` et rbx vaut 6.

```

1      bt rax, 0
2      jc _sinon
3      mov rbx, 5
4      jmp _fin_si
5  _sinon:
6      mov rbx, 6
7  _fin_si:
8      mov rcx, 12

```

Voici un exemple de test d'infériorité :

```

1      cmp rax, -10
2      jns _sinon
3      mov rbx, 5
4      jmp _fin_si
5  _sinon:
6      mov rbx, 6
7  _fin_si:
8      mov rcx, 12

```

Voici un exemple de test d'inégalité :

```

1      cmp rax, rdx
2      jns _sinon
3      mov rbx, 5
4      jmp _fin_si
5  _sinon:
6      mov rbx, 6
7  _fin_si:
8      mov rcx, 12

```



## 4.5 Exercices

### 4.5.1 Exercice 1

```
1  ;;; TD3 Exo 1
2
3  global main
4  section .text
5  main:
6      mov rax, 42          ; Met 10 dans rax
7      cmp rax, 0
8      jnz _if_not_null
9      jmp _end
10 _if_not_null:
11     mov rbx, 1
12 _end:
13     mov rax, 60
14     mov rdi, 0
15     syscall
```

### 4.5.2 Exercice 2

```
1  ;;; TD3 Exo 2
2
3  global main
4  section .text
5  main:
6      mov rax, 42
7      bt rax, 0
8      jc _if_not_even
9      mov r8, 0
10     jmp _end
11 _if_not_even:
12     mov r8, 1
13 _end:
14     mov rax, 60
15     mov rdi, 0
16     syscall
```

### 4.5.3 Exercice 3

```
1  ;;; TD3 Exo 3
2
3  global main
4  section .text
5
6  main:
7      mov r14, 42
8      mov r15, 43
9      cmp r14, r15
10     jnz _if_not_equal
11     mov r14, 0
12     mov r15, 0
13     jmp _end
14
15 _if_not_equal:
16     xor r14, r15
17     xor r15, r14
18     xor r14, r15
19 _end:
20     mov rax, 60
```

```

21     mov rdi, 0
22     syscall
23
24     ;;; Ici on utilise le xor swap, sachant que le xor est commutatif
25     ;;; 1. xor r14, r15 applique le xor et stocke le résultat dans r14
26     ;;; 2. xor r15, r14 applique le xor et stocke le résultat dans r15
27     ;;; 3. xor r14, r15 applique le xor et stocke le résultat dans r14
28     ;;; En pratique les étapes font pour x valant 1010 et y valant 0011
29     ;;; xor x, y <=> xor 1010, 0011 = 1001 -> x
30     ;;; xor y, x <=> xor 0011, 1001 = 1010 -> y
31     ;;; xor x, y <=> xor 1001, 1010 = 0011 -> x

```

#### 4.5.4 Exercice 4

```

1     ;;; TD3 Exo 4
2
3     global main
4     section .text
5     main:
6         mov rax, 42
7         mov rbx, 11
8         cmp rax, rbx
9         js _rax_is_less
10        mov r8, rax
11        mov r9, rbx
12        jmp _end
13    _rax_is_less:
14        mov r8, rbx
15        mov r9, rax
16    _end:
17        mov rax, 60
18        mov rdi, 0
19        syscall

```

#### 4.5.5 Exercice 5

```

1     ;;; TD3 Exo 5
2
3     global main
4     section .text
5     main:
6         mov rdi, 11
7         bt rdi, 0                ; CF vaut 0 si au moins multiple de 2
8         ;; On envoie vers les multiples de 2 sinon on termine
9         jnc _if_even
10        mov rsi, 0
11        jmp _end
12    _if_even:
13        mov rsi, 1
14        bt rdi, 1                ; CF vaut 0 si au moins multiple de 4
15        ;; On envoie vers les multiples de 4 sinon on termine
16        jnc _if_multiple_of_4
17        jmp _end
18
19    _if_multiple_of_4:
20        mov rsi, 2
21        bt rdi, 2                ; CF vaut 0 si au moins multiple de 8 au plus
22        ;; On envoie vers les multiples de 8 sinon on termine
23        jnc _if_multiple_of_8
24        jmp _end
25

```

```

26 _if_multiple_of_8:
27     mov rsi, 3
28 _end:
29     mov rax, 60
30     mov rdi, 0
31     syscall

```

## 5 Laboratoire 4 : Variables globales

Ce laboratoire va traiter des **variables globales**, elles sont généralement déclarées en début de code et utilisables dans **tout** le fichier. On oppose ce comportement aux variables **locales**.

### 5.1 Sections dédiées aux variables

Un fichier binaire exécutable au format **elf** est divisé en plusieurs sections. La **directive section** permet de les définir dans le code source du programme. D'habitude, on utilise la section **.text** qui contient les instructions exécutables du programme. Les sections **elf** standards pour les variables globales sont au nombre de trois : **.data**, **.rodata**, **.bss**. Les variables créées dans ces sections ont la même durée de vie que le programme, c'est-à-dire qu'elles apparaissent à sa création et disparaissent après.

Les variables en assembleur ne sont pas *typées*, on doit seulement fournir leur taille. Ensuite, elles sont mises l'une après l'autre en mémoire. Pour les sections **.data** et **.rodata**, les variables sont explicitement créées et sont dans le programme.

Nom	Rôle
<b>.text</b>	Instructions exécutables du programme
<b>.data</b>	Variables globales explicitement initialisées
<b>.rodata</b>	Variables globales explicitement initialisées en lecture seule
<b>.bss</b>	Variables globales implicitement initialisées à zéro

Voici les instructions qui permettent de définir la taille attribuée aux variables :

Taille en bytes	Pseudo-instruction	Signification
1	<b>DB</b>	<b>Define Byte</b>
2	<b>DW</b>	<b>Define Word</b>
4	<b>DD</b>	<b>Define Doubleword</b>
8	<b>DQ</b>	<b>Define Quadword</b>

Enfin, voici un exemple d'utilisation en assembleur :

```

1  section .data
2      i1 DB -1                ; 1 byte initialisé
3      i2 DW 23                ; 2 bytes initialisés
4      i4 DD -1                ; 4 bytes initialisés
5      i8 DQ 0x80_00_00_00_00 ; 8 bytes initialisés donc 0x00_00_00_80_00_00_00_00
6
7  section .rodata
8      ci8 DQ 93229
9
10 section .text
11 ;...

```

Attention, les variables **.bss** ne sont littéralement présentes dans le fichier exécutable, elles sont créées et mises à zéro lors du démarrage du programme.

Voici les instructions liées à cette section ainsi qu'un exemple.

Taille en bytes	Pseudo-instruction	Signification
1	<b>RESB</b>	<b>Reserve Byte</b>
2	<b>RESW</b>	<b>Reserve Word</b>
4	<b>RESQ</b>	<b>Reserve Doubleword</b>
8	<b>RESQ</b>	<b>Reserve Quadword</b>

Dans le code source, on met un entier pour indiquer combien de *bytes*, *words*, ou *double words* sont réservés en mémoire au début du programme.

```

1 section .bss
2   ; tout est crée et initialisé à 0
3   x1 RESB 10      ; 10 * 1 bytes réservés
4   x2 RESW 6       ; 6 * 2 bytes réservés
5   x4 RESD 100     ; 100 * 4 bytes réservés
6   x8 RESQ 2       ; 2 * 8 bytes réservés

```

## 5.2 Accès à une variable

Le nom d'une variable est en fait un label, une étiquette. Son utilisation dans le code source est remplacée par l'adresse de l'emplacement où l'étiquette est placée. Cela correspond à l'adresse de la variable lorsqu'on est dans les sections décrites au-dessus. Rappelons que nous travaillons en 64 bits. Les adresses de variables ont donc toujours cette taille, soit 8 *bytes*.

Pour accéder au *contenu* de la variable, il faut réaliser un **déréférencement** : atteindre ce qui se trouve à l'adresse de la variable. Pour indiquer à *nasm* qu'on désire déréférencer un **pointeur**, on place l'adresse entre crochets. Voici un exemple.

```

1 section .data
2   i4 DD 42          ; entier sur 4 bytes
3
4 section .text
5   mov rax, i4        ; rax <-- _adresse_ (8 bytes) de i4.
6
7   mov ebx, [i4]       ; ebx <-- _contenu_ de ce qui se trouve
8                       ; à l'adresse i4 dans ebx.
9                       ; s'étend sur 4 bytes car ebx est sur 4 bytes.
10
11  mov ecx, [rax]       ; rax contient l'adresse de i4 donc [rax] = [i4].
12                       ; donc, ecx <-- _contenu_ de l'adresse rax.
13                       ; s'étend sur 4 bytes car ecx est sur 4 bytes.

```

### 5.2.1 Boutisme

Lorsqu'on stocke des données en mémoire, on peut suivre deux stratégies pour ordonnancer les *bytes*. Soit on met le byte de rend le plus élevé à l'adresse la plus petite, c'est le *big endian*. Soit on met le byte de rend le plus petit à la plus petite adresse, c'est le *little endian*. L'architecture x86 adopte le *little endian*.

Voici un exemple qui en illustre l'usage pour comprendre comment les adresses logiques sont mises l'une à la suite de l'autre et comment accéder aux valeurs stockées en *little endian*.

```

1 section .data
2   vw DW 0x0102      ; Valeur en hexadécimal stockée sur 4 bytes
3
4   ;; à l'adresse vw : 0x02
5   ;; à l'adress vw + 1 : 0x01
6   ;; /.../.../0x02/0x01/.../.../
7   ;; -> petite adresse --> grande adresse -->
8
9   vq DQ 0x1122334455667788
10
11  ;; à l'adresse vq : 0x88
12  ;; à l'adresse vq+1 : 0x77
13  ;; à l'adresse vq+2 : 0x66
14  ;; à l'adresse vq+3 : 0x55
15  ;; à l'adresse vq+4 : 0x44
16  ;; à l'adresse vq+5 : 0x33
17  ;; à l'adresse vq+6 : 0x22
18  ;; à l'adresse vq+7 : 0x11
19
20  ;; /.../0x88/0x77/0x66/0x55/0x44/0x33/0x22/0x11/.../
21  ;; -> petite adresse --> grande adresse -->
22

```

```

23
24 ;; vue complète de la section .data
25 ;; | vw | vq | qu'on décompose en leurs bytes respectifs. Pas d'espaces entre adresses.
26 ;; |...|0x02|0x01|0x88|0x77|0x66|0x55|0x44|0x33|0x22|0x11|..
27 ;; -> petite adresse --> grande adresse -->
28
29 section .text
30 ;...
31 mov r8b, [vw] ; r8b <-- 0x02
32 mov r12b, [vq + 7] ; r12b <-- 0x11
33
34 ;; pas de trous entre variables
35 ;; vw est directement suivie par vq
36
37 mov r13b, [vw+2] ; r13b <-- 0x88 (plus petit byte de vq)
38 mov r14b, [vw+3] ; r14b <-- 0x77
39 mov r15b, [vq - 1] ; r15b <-- 0x01 (dernier byte de vw)

```

## 5.3 Annésie de nasm

Comme il a été mentionné précédent, le nom d'une variable en code source nasm est une étiquette. La seule information à laquelle on accède en fournissant un nom de variable, c'est l'adresse à laquelle l'étiquette est placée. Aucune autre information n'est rendu disponible par nasm, ce qui créera potentiellement des problèmes par la suite.

### 5.3.1 Problème de taille

L'assembleur nasm ne **retient pas** la taille des variables. Lorsqu'on accède au contenu d'une variable, le nombre de *bytes* déréférencés à partir de l'adresse entre crochet est déduit de la taille du second opérande, s'il existe et n'est pas un immédiat. Dans le cas contraire, il faut renseigner la taille de la donnée avec une spécification de taille, décrite ci-dessous.

Taille en <i>bytes</i>	Spécificateur
1	<code>byte</code>
2	<code>word</code>
4	<code>dword</code>
8	<code>qword</code>

L'exemple repris à la page 8 et 9 du TD4 illustre bien les problèmes qu'on peut rencontrer et comment utiliser les spécificateurs de taille. Il faut particulièrement se méfier des erreurs d'extension de signes ou d'adressage avec des tailles différentes. En ce qui concerne l'extension de signe, seule le `mov` avec un *registre* pour destination et un immédiat accepte le 64bit. Autrement, on aura une extension de signe qui dépend du bit de poids le plus fort.

Il y a aussi un comportement particulier quand on place une variable de 8 bytes dans un registre de 4 bytes qui a aussi un registre plus grand équivalent. Par exemple :

```

1  global main
2  section .data
3  var dq 0x12345678
4
5  section .text
6  main:
7  ;; En plaçant une valeur de 8 bytes dans un registre de 4 bytes
8  ;; qui a un équivalent supérieur de 8 bytes, on va vider le registre de 8 bytes
9  ;; et ensuite seulement le remplir avec la variable définie sur 8 bytes
10 mov eax, [var1] ; rax -> 0x0000...000012345678

```

### 5.3.2 Problème de section

L'assembleur nasm ne retient pas de quelle section vient une variable. De ce fait, il nous laisse créer un programme et compiler un programme qui modifie une variable en lecture seule par exemple. Mais lors de l'exécution on aura une erreur.

## 5.4 Type de données

Dans cette section, nous allons passer en revue quelques types de données classiques et voir comment les implémenter en assembleur. Les chaînes de caractères sont introduites au TD05 et les tableaux au TD06.

### 5.4.1 Entier

Les entiers sont le cas le plus simple, il suffit de choisir une taille sur 1, 2, 4 ou 8 bytes et de réserver l'espace en conséquence.

```
1 section .rodata
2
3 vqd DQ -1          ; valeur décimale
4 vdh DD 0x12345678  ; valeur hexadécimale
5 vwo DW 114q        ; valeur octale
6 vbb DB 10101010b   ; valeur binaire
```

### 5.4.2 Flottant

Il existe des registres spéciaux pour les flottants mais on ne les traite pas au TD.

### 5.4.3 Caractère

Comme en Java, les caractères sont considérés comme un type numérique. La table ASCII associe un nombre à chaque caractère. Pour les caractères au delà de la table ASCII, on utilise généralement le codage UTF-8 sur 2 bytes.

Exemple :

```
1 section .data
2
3 c1 DB 'A'          ; c1 contient le code UTF-8 du caractère 'A'
4 c2 DB 65            ; c2 contient la valeur 65
5 ;; 65 en décimal équivaut à 0x41, qui représente le caractère 'A' en UTF-8
```

## 5.5 Exercices

### 5.5.1 Exercice 1 : remplir les pointillés

```
1 global main
2
3 section .data
4     var1 DB 1
5     var2 DB 2
6     var3 DW 0x0304
7     var4 DQ 0x00_00_00_00_80_00_FF_FF
8     ;; la section des données occupe 12 bytes
9     ;;; Contenu de la plus petite adresse à la plus grande
10    ;;;     v1 v2 v3     v4
11    ;;; ...01020403FFFF008000000000|...
12
13 section .text
14 main:
15     mov rax, var1 ; rax contient l'adresse de var1
16     ;; (on ne connaît pas explicitement cette adresse, c'est l'OS qui la détermine).
17     mov al, [var1] ; al contient 1
18     mov ax, [var1] ; ax contient 0x0201 soit 0000_0010_0000_0001b
19     ;; Car ax est un registre sur 2 bytes, il lit la valeur sur 2 bytes
20     ;; Donc on prend en fait les valeurs de var1 et var2
21     mov al, [var3] ; al contient 0x04 soit 0000_0100b
22     ;; Car al est un registre sur 1 byte, on prend le premier byte de la valeur de var3
23     mov ax, [var3] ; ax contient 0x0304 soit 0000_0011_0000_0100b
24     mov rax, -1 ; rax contient 0xFF_FF_FF_FF_FF_FF_FF_FF par extension de signe
25     mov eax, [var4] ; eax contient : 0x80_00_FF_FF (on ne lit que 4 byte en mémoire)
```

```

26 ;; rax contient 0x00_00_00_00_80_00_FF_FF, les valeurs précédentes de rax sont écrasées
27 ;; c'est le cas quand on utilise un registre 64bits avec une variable 32 bits.
28     mov rax, 60
29     mov rdi, 0
30     syscall

```

### 5.5.2 Exercice 2

```

1  global main
2
3  section .data
4
5      nb dd 11 ; Essayer avec C3_D8_10_00 pour bit de poids fort différent
6
7  section .text
8
9  main:
10     mov eax, nb ; Placer adresse dans eax
11     mov ebx, [nb] ; Placer la valeur dans ebx
12
13     ;; fin
14
15     mov rax, 60
16     mov rdi, 0
17     syscall

```

### 5.5.3 Exercice 3

```

1  ;; td04_ex3.asm
2
3  global main
4
5  section .bss
6
7      nb resq 1 ; On réserve 1 x 8 bytes
8
9  section .text
10
11  main:
12
13      ;; end
14
15      mov byte [nb], 42
16
17      mov rax, 60
18      mov rdi, 0
19      syscall

```

### 5.5.4 Exercice 4

```

1  ;; td04_ex4.asm
2
3  global main
4
5  section .data
6
7      b0 db 0
8      b1 db 0
9      b2 db 0

```

```

10     b3 db 0
11
12 section .rodata
13     nb dd 0x12_34_56_78
14
15 section .text
16
17 main:
18
19     mov al, [nb]
20     mov [b0], ax
21     mov al, [nb+1]
22     mov [b1], ax
23     mov al, [nb+2]
24     mov [b2], al
25     mov al, [nb+3]
26     mov [b3], al
27
28 ;; end
29
30     mov rax, 60
31     mov rdi, 0
32     syscall

```

Attention, vu que les valeurs sont contiguës en mémoire, on peut attribuer nb à b0 en précisant que c'est un dword. Comme ça, on aura bien copié chaque *byte* de nb dans les autres variables.

### 5.5.5 Exercice 5

```

1 ;; td04_ex5.asm
2
3 global main
4
5 section .data
6
7     var1 db 11
8     var2 db 34
9
10 section .text
11
12 main:
13
14     mov r8b, [var1]
15     mov r9b, [var2]
16     mov [var1], r9b
17     mov [var2], r8b
18
19 ;; end
20     mov rax, 60
21     mov rdi, 0
22     syscall

```

### 5.5.6 Exercice 6

```

1 ;; td04_ex6.asm
2
3 global main
4
5 section .data
6
7     var1 db 11

```



```

8      var2 db 34
9
10     section .bss
11
12     var3 resb 1
13
14     section .text
15
16     main:
17         mov r8b, [var1]
18         mov r9b, [var2]
19         cmp r8b, r9b
20         js _var1_is_less
21         mov al, [var2]
22         jmp _else
23     _var1_is_less:
24         mov al, [var1]
25
26     _else:
27         mov [var3], al
28
29     ;; end
30
31     mov rax, 60
32     mov rdi, 0
33     syscall

```

## 6 Laboratoire 5 : Appels système

Ce laboratoire traite des appels systèmes ainsi que leurs arguments, puis de la représentation des chaînes de caractères.

### 6.1 Définition

Un **appel système** est un service offert par le système d'exploitation pour effectuer diverses tâches comme ouvrir ou lire le contenu d'un fichier. Chaque appel système est identifié par un numéro de service. L'instruction `syscall` permet de faire basculer le *CPU* en mode privilégié et passe la main au service système demandé. Une fois le code système exécuté, le code reprend à l'instruction suivant le `syscall`.

### 6.2 Mise en œuvre

En nasm, le un appel système se fait en quatre étapes :

1. Placer le numéro du service désiré dans `rax`
2. Mettre les paramètres, s'il y en a dans `rdi`, `rsi`, `rdx`, `rcx`, `r9`, `r9`.
3. Appeler le système via l'instruction `syscall`.
4. Consulter dans `rax` la valeur de retour, s'il y en a une, ou le statut d'erreur si nécessaire et utile.

Les étapes 1. et 2. peuvent se faire dans l'ordre qu'on veut. Mais tout doit être prêt avant d'exécuter l'instruction `syscall`. Il faut absolument renseigner le numéro de service dans `rax` et utiliser les bons registres dans le bon ordre pour les paramètres.

### 6.3 Registres non préservés

Il est important de savoir que l'instruction `syscall` utilise les registres :

- `rcx` pour la sauvegarde de la valeur du registre `rip` : cela permet, à la fin de l'appel système, le retour au code appelant précisément l'instruction qui suit `syscall` par la restauration de cette valeur sauvegardée.
- `r11` pour la sauvegarde du registre `rflags` et sa restauration lors du retour au code appelant.

Donc attention, si le contenu de `rcx` ou `r11` est important, il faut le sauver avant l'appel système. Le registre `rax` est lui aussi modifié, la valeur de retour de l'appel système y est stockée.

## 6.4 Numéro du service

Pour connaître la liste des numéros de service, il n'y a pas d'autre choix que de [la consulter](#).

## 6.5 Paramètre et retour

Pour connaître les paramètres attendus par un appel système ou savoir s'il retourne une valeur, il faut consulter les man (manuels) de Linux. Plus précisément la [section 2](#).

Là, on peut voir par exemple que le numéro de service de `exit` est 60 et qu'on lui donne l'argument 0, qui signifie que le processus s'est bien terminé. Dans le cas de `exit`, aucune valeur n'est retournée.

Si maintenant on se concentre sur l'instruction `open`, on verra que son premier argument est un *pointeur de caractères constant*, appelé `pathname`. On comprend alors que c'est un chemin vers un fichier, sous forme de chaîne de caractères, qui n'est pas modifiée par l'instruction `open`. On attendra de cette chaîne qu'elle soit zéro-terminée, qu'elle finisse par une byte de zéros pour annoncer que c'est la fin de la chaîne.

Le deuxième argument est une instruction en *octal* qui fournit des indicateurs sur le comportement de l'instruction. On peut combiner plusieurs de ces *flags* avec l'opérateur *pipe* `|`.

Le troisième argument est attendu dans certains cas suivant le deuxième argument.

Si tout se passe bien, la valeur de retour est un petit entier positif appelé *descripteur de fichier*. C'est par son biais qu'on peut ensuite accéder, écrire, lire ou fermer un fichier ouvert. Cette valeur est retournée dans `rax`.

Voici un exemple :

```
1 ; 01_open_extraite.asm
2
3 global main
4
5 section .rodata
6     nomFichier db 'brol', 0 ; ne pas oublier le zéro en fin
7
8 section .text
9
10 main:
11     ; ouverture de brol en écriture seule avec placement
12     ; de la tête d'écriture en fin de fichier
13
14     mov rax, 2          ; open
15     mov rdi, nomFichier ; /adresse/ du 1er caractère du nom
16     mov rsi, 1q | 2000q ; WRONLY + O_APPEND
17     syscall             ; appel système
```

On voit que le nom de fichier utilise des apostrophes simples `'`, on peut aussi utiliser les doubles `"` et même les accents `‘` quand on a besoin de séquences d'échappement. Ici, le nom de fichier est un immédiat de type caractère. C'est un type de variable immuable déclaré dans `.rodata`, on utilise le *byte* avec l'instruction `DB` pour le déclarer.

Par exemple,

```
1 section .rodata
2
3     str1 db "abc"          ; 3 bytes initialisés
4     str2 db 'abc', 0       ; 4 (3+1) bytes initialisés, str2 zéro-déterminé
5     str3 db "abc\n", 0     ; 6 bytes initialisés, zéro-déterminé
6     str4 db `abc\n`, 0     ; 5 bytes initialisés, zéro-déterminé
```

Pour mesurer la taille d'une chaîne de caractères, on peut faire calculer par `nasm` le nombre d'espaces adressables entre deux étiquettes. Ces deux étiquettes sont remplacées par les adresses où elles ont été collées, au moment de l'assemblage. On peut utiliser le symbole `$` qui réfère à l'adresse courante.

```
1 section .rodata
2     str0 DB "abc"
3     lenStr0 DQ lenStr0 - str0 ; taille en byte de str0 : 3
4     str1 DB `abc\n`, 0
5     lenStr1 DQ $ - str1      ; taille en bytes de str1 : 5
```

Le troisième exemple repris dans le TD concerne l'instruction `write`

## 6.6 Entrée et sorties standards

Au démarrage d'un programme, trois **flux** sont disponibles sans devoir être ouverts : le flux d'entrée standard `stdin`, associé par défaut au clavier. Il peut être accédé comme un fichier avec 0 comme valeur de descripteur de fichier.

Le flux de sortie standard, `stdout` associé par défaut à l'écran, il peut être accédé comme un fichier avec 1 comme valeur de descripteur de fichier.

Le flux de sortie d'erreur standard, `stderr`, également associé par défaut à l'écran, il peut être accédé comme un fichier avec 2 comme valeur de descripteur de fichier.

## 6.7 Hello, World!

Voici ce que donne un Hello, World en assembleur.

```
1 ; 02_hello_world.asm
2
3 global main
4
5 section .rodata
6     msg      DB  `Hello, Wolrd!\n`
7     lgrMsg   DQ  lgrMsg - msg
8
9 section .text
10 main:
11     ; affichage
12     mov rax, 1          ; write
13     mov rdi, 1          ; stdout
14     mov rsi, msg        ; adresse du 1er caractère
15     mov rdx, [lgrMsg]   ; nombre de caractères
16     syscall
17 ; end
18
19 mov rax, 60             ; exit
20 mov rdi, 0              ; ok
21 syscall
```

## 6.8 Cinq appels système

La table suivante reprend les appels systèmes les plus fréquents ainsi que quelques indications.

Service	Numéro ( <code>rax</code> )	But	Paramètres ( <code>rdi</code> , <code>rsi</code> , <code>rdx</code> , <code>rcx</code> , <code>r8</code> , <code>r9</code> )	Retour ( <code>rax</code> )	Notes
<code>exit</code>	60	quitter un programme	entier à retourner au processus parent, 0 si tout ok	aucun	à mettre en fin de tout programme
<code>open</code>	2	ouvrir ou créer un fichier	1 <sup>er</sup> : chemin vers le fichier : chaîne zéro-terminée ; 2 <sup>e</sup> : options d'ouverture : indicateurs à combiner avec <code>O</code> ; 3 <sup>e</sup> : mode : si création d'un fichier	descripteur de fichier ou entier négatif en cas d'erreur	options d'ouverture : <code>/usr/include/bits/fcntl-linux.h</code> (voir p. 6)
<code>close</code>	3	fermer un fichier	descripteur du fichier	0 si ok, -1 si erreur	le descripteur de fichier est celui retourné par <code>open</code>
<code>read</code>	0	lire depuis un fichier	1 <sup>er</sup> : descripteur du fichier ; 2 <sup>e</sup> : adresse où stocker le résultat de la lecture ; 3 <sup>e</sup> : nombre de <i>bytes</i> à lire	nombre d'octets effectivement lus, -1 en cas d'erreur	— le descripteur de fichier est celui retourné par <code>open</code> ou 0 pour lire au clavier ; — la tête de lecture est avancée du nombre de <i>bytes</i> lus
<code>write</code>	1	écrire dans un fichier	1 <sup>er</sup> : descripteur du fichier ; 2 <sup>e</sup> : adresse de ce qui doit être écrit ; 3 <sup>e</sup> : nombre de <i>bytes</i> à écrire	nombre d'octets effectivement écrits, -1 en cas d'erreur	— le descripteur de fichier est celui retourné par <code>open</code> ou 1 pour écrire à l'écran ; — la tête d'écriture est avancée du nombre de <i>bytes</i> écrits

## 6.9 Exercices

### 6.9.1 Ouvrir un fichier

Quand le fichier `bro1` existe avec droit d'écriture, on le code 3 qui indique le descripteur de fichier, tout se déroule correctement.

Quand on a pas de droit d'écriture, on a une erreur et `rax` contient -13.

Quand le fichier `bro1` n'existe pas, on a aussi une erreur et `rax` contient -2.

Une fois qu'on change les arguments donnés à l'ouverture de fichier :

Quand le fichier `bro1` existe avec droit d'écriture, on a le code 3 qui indique le descripteur de fichier, tout se déroule correctement.

Quand le fichier n'a pas de droit d'écriture, on a une erreur et `rax` contient -13.

Par contre, quand `bro1` n'existe pas, on réussi à l'écrire dans ce cas-ci et `rax` contient 3.

### 6.9.2 Ouverture et écriture

```
1  ;; td05_ex2.asm
2
3  global main
4
5  section .rodata
6      nomFichier db `bro1`, 0          ; nécessaire pour le nom du fichier
7      msgOK      db `fichier ouvert avec succès\n`, 0
8      lenOk      dq $ - msgOK
9      msgFail    db `échec lors de l'ouverture du fichier\n`, 0
10     lenFail     dq $ - msgFail
11
12  section .text
13  main:
14
15     mov rax, 2                        ; open
16     mov rdi, nomFichier              ; fichier
17     mov rsi, 0q                      ; RONLY
18     syscall
19
20     cmp rax, 0
21     js _fail
22  ;; affichage
23     mov rbx, rax                    ; sauver le descripteur du fichier
24     mov rax, 1                      ; write
25     mov rdi, 1                      ; stdout
26     mov rsi, msgOK                  ; adresse du premier char
27     mov rdx, [lenOk]                ; nombre de caractères
28     syscall
29
30     mov rax, 3                      ; close
31     mov rdi, rbx                    ; descripteur du fichier
32     syscall
33
34     mov rax, 60                     ; end
35     mov rdi, 0                      ; zero, pas d'erreur
36     syscall
37
38  _fail:
39     mov rax, 1                      ; write
40     mov rdi, 1                      ; stdout
41     mov rsi, msgFail                ; message d'erreur, adresse du premier char
42     mov rdx, [lenFail]              ; nombre de caractères
43     syscall
44
45     mov rax, 60                     ; end
```

```

46     mov rdi, 1                ; 1, erreur
47     syscall

```

### 6.9.3 Convertir rsi en caractères

```

1  ;; td05_ex3.asm
2
3  global main
4
5  section .bss
6      charToShow resb 1
7
8  section .text
9  main:
10
11     mov rsi, 7
12     or rsi, 00110000b        ; masque pour convertir en ASCII
13     mov [charToShow], rsi    ; Place la valeur de rsi dans la variable
14
15  ;; affichage
16
17     mov rax, 1                ; write
18     mov rdi, 1                ; stdout
19     mov rsi, charToShow      ; Adress du premier caractère
20     mov rdx, 1                ; On lit exactement 1 byte
21     syscall
22
23  ;; end
24
25     mov rax, 60
26     mov rdi, 0
27     syscall

```

Le code fonctionne mais quand on met une valeur plus grande que 9, le masque booléen converti la valeur une valeur ASCII différente, par exemple la valeur 15 devient le caractère point d'interrogation.

### 6.9.4 Afficher la parité

```

1  ;; td05_ex4.asm
2
3  global main
4  section .rodata
5      pairMessage    db `Le contenu est pair\n`
6      lenPair        dq $ - pairMessage
7      impairMessage  db `Le contenu est impair\n`
8      lenImpair      dq $ - impairMessage
9  section .text
10 main:
11     mov rcx, 11
12
13     bt rcx, 0            ; Donne la valeur du 1er bit donc 1 si impair
14     jc _impair
15
16     mov rax, 1            ; write
17     mov rdi, 1            ; stdout
18     mov rsi, pairMessage  ; adresse du 1er caractère
19     mov rdx, [lenPair]    ; longueur du message
20     syscall
21     jmp _end
22
23 _impair:

```

```

24     mov     rax, 1                ; write
25     mov rdi, 1                    ; stdout
26     mov rsi, impairMessage        ; adresse du 1er caractère
27     mov rdx, [lenImpair]          ; longueur du message
28     syscall
29
30 _end:
31
32     mov rax, 60
33     mov rdi, 0
34     syscall

```

### 6.9.5 Écrire des fichiers

```

1  ;; td05_ex5.asm
2
3  global main
4  section .bss
5      var resq 1                    ; Réserve 1x8 bytes
6  section .rodata
7      nomPair db `pair`, 0
8      nomImpair db `impair`, 0
9
10 section .text
11
12 main:
13
14     mov rcx, 77
15     mov qword [var], rcx          ; sauver le contenu de rcx
16
17     bt rcx, 0                     ; CF prend la valeur du 0eme bit de rcx
18     jc _impair
19
20     mov rax, 2                     ; open
21     mov rdi, nomPair              ; adresse du 1er caractère du nom
22     mov rsi, 1q                   ; WRONLY
23     syscall
24     jmp _write
25
26 _impair:
27     mov rax, 2                     ; open
28     mov rdi, nomImpair            ; adresse du 1er caractère du nom
29     mov rsi, 1q                   ; WRONLY
30     syscall
31
32 _write:
33     mov rbx, rax                  ; sauver le descripteur
34     mov rax, 1                    ; write
35     mov rdi, rbx                  ; descripteur du fichier
36     mov rsi, var                  ; adresse de ce qui doit être écrit
37     mov rdx, 8                    ; Nombre de bytes à écrire
38     syscall
39
40 ;; end
41     mov rax, 60
42     mov rdi, 0
43     syscall

```

### 6.9.6 Stocker une taille de fichier

À faire.

## 7 Laboratoire 6 : Tableau et boucles

Dans ce TD, on traite des tableaux globaux et des différentes techniques pour accéder à leur contenu sont expliquées. Ensuite, les implémentations de différents types de boucle sont présentés.

### 7.1 Tableau

Pour ce laboratoire, nous nous tenons à une définition simple du tableau (*array*). On considère qu'un tableau est une structure de données homogène et dont les éléments sont contigus en mémoire. Il ne faut cependant pas oublier qu'en assembleur, les données ne sont pas typées. Donc, même si un tableau est composé de données de même type, c'est lors de l'utilisation qu'une signification et qu'un type sont donnés. En effet, lorsqu'elle est définie, seule la taille de la variable est renseignée.

Comme un tableau possède des éléments de même taille, le nombre de *bytes* qu'il occupe en mémoire dépend du nombre d'éléments et de la taille d'un élément :

$$T_t = n \times T_e \quad (1)$$

Ce qui signifie que la taille totale est égale au nombre d'éléments multipliés par la taille d'un élément.

#### 7.1.1 Section `.data` et `.rodata`

Les sections `.data` et `.rodata` permettent de réserver un tableau initialisé avec des valeurs explicites. On place une étiquette et une pseudo-instruction de taille (`db`, `dw`) comme lors de la définition d'une simple variable. Pour définir le tableau en tant que tel, on va séparer les valeurs par des virgules ou utiliser l'instruction `times` pour répéter une valeur. L'étiquette identifie l'adresse du premier *byte* du premier élément du tableau, on peut donc l'utiliser pour trouver la taille d'un tableau ou accéder à ses éléments. Par exemple :

```
1 section .data:
2
3     tab db 0, 1, 2, 3
4     tab2 dd, 'A', 'B'
```

#### 7.1.2 Section `.bss`

La section `.bss` permet de réserver des emplacements mémoire et initialisent un tableau vide à 0, on doit préciser si on veut utiliser d'autres valeurs. Comme pour les variables, là où les données dans `.data` et `.rodata` indiquent ce qu'on y met, `.bss` donne le nombre de bytes qu'il faut réserver. On voit que la déclaration est la même que pour les variables, mais on en fera un usage différent :

```
1 section .bss:
2
3     t1 resb 1
4     t2 resw 6
```

#### 7.1.3 Chaîne de caractères

En pratique, les chaînes de caractères littérales sont des tableaux de caractères. Elles sont définies dans `.rodata`. Leurs contenus sont fournis entre guillemets, apostrophes ou accents graves. Avec ces derniers, les séquences d'échappement sont interprétées par `nasm`. Il est conseillé d'utiliser `db`. Pour les chaînes de caractères modifiables non locales, on utilise des tableaux de *bytes* définis dans les sections `.data` et `.rodata`.

Par exemple :

```
1 section .rodata
2
3     s1      db "abc"           ; 3 bytes
4     s1_long db "a", "b", "c"   ; 3 bytes
5     s1_alt  db 0x61, 0x62, 0x63 ; 3 bytes
```

### 7.1.4 Accès au contenu

Pour accéder au contenu d'un tableau, on doit calculer l'adresse de chaque élément désiré et de manière générale, pour l'adresse du  $n$ -ième élément est donné par :

$$tab + (n - 1) \times T_e \quad (2)$$

Aussi, il n'est pas strictement nécessaire que les éléments d'un tableau soient contigus, on peut vouloir simplement accéder à chaque élément par un index comme montré dans (2).

Avant de pouvoir accéder à un élément, nous devons voir comment le faire via les modes d'adressage présentés au point suivant.

## 7.2 Modes d'adressage

### 7.2.1 Immédiat

On peut renseigner un opérande sous la forme d'une valeur immédiate, en ce compris les valeurs calculées par l'assembleur, comme l'adresse d'un immédiat.

### 7.2.2 Registre

Il est possible de renseigner un opérande sous la forme d'un registre, c'est le cas classique où un registre est la source, et l'autre la destination d'une instruction `mov`.

### 7.2.3 Emplacement mémoire

Il est possible de renseigner un opérande sous la forme d'une expression qui fait référence à un emplacement mémoire. C'est ainsi qu'on accède aux variables qui vivent en mémoire centrale. Une telle expression, appelée *offset* en anglais, est constituée de quatre parties :

**une base (*base*)** il s'agit obligatoirement d'un des 16 registres généraux qui pointe sur une variable ou sur le premier élément d'un tableau, qu'ils soient globaux ou locaux.

**un indice (*index*)** il s'agit obligatoirement d'un des 16 registres généraux dont on se sert typiquement pour l'accès à un élément spécifique d'un tableau.

**un facteur d'échelle (*scale*)** il s'agit obligatoirement d'un des 4 immédiats suivants : 1, 2, 4 ou 8, dont on se sert comme facteur multiplicatif à la partie indice pour tenir compte de la taille des éléments d'un tableau lors de l'accès à une de ses éléments.

**un déplacement (*displacement*)** il s'agit obligatoirement d'un immédiat, le plus souvent une étiquette pour l'accès à une variable ou à un tableau global.

La forme générale d'un *offset* est :

$$\text{offset} = \text{base} + \text{index} \times \text{scale} + \text{déplacement} \quad (3)$$

Chacune des quatre composantes d'un *offset* est facultative. On peut ainsi par exemple rencontrer un *offset* constitué d'un déplacement seul, ou d'une base et d'un indice sans déplacement ni facteur d'échelle, etc.

Un *offset* ne peut servir que dans le cadre d'un accès, en lecture ou en écriture, à la mémoire. Avec `nasm`, les *offsets* se trouvent donc toujours entre les crochets de l'opérateur de déréférencement.

## 7.3 Instructions `inc` et `dec`

Les instructions `inc` et `dec` servent à incrémenter et décrémenter un registre ou une variable.

## 7.4 Boucles

Voir les tableaux récapitulatifs du TD06.

## 7.5 Déboguage et tableau

Expliqué clairement dans le TD06.

## 7.6 Exercices

Dans `kdbg`, on peut utiliser `/d(char[101])tab` pour afficher les 101 premiers éléments du tableau `tab` pour des `char`. On peut aussi utiliser `int`, `long`, etc.



### 7.6.1 Assigner des valeurs à des tableaux

```
1  ;; td06_ex1.asm
2
3  global main
4
5  ;; Pour observer les 10 cases du tableau avec 2 bytes par cellule,
6  ;; On fait /d(short[10])tab dans "Watched expressions" pour KDBG.
7
8  section .bss
9      tab resw 10                ; 10 * 2 bytes initialisés à zéro par défaut
10
11 section .text
12
13 main:
14     mov rax, 2                  ; 3ème valeur donc index 2
15     mov word [tab + rax*2], 3 ; multiplier par 2 car RESW donc 2 bytes par case
16
17     mov rax, 7                  ; 8ème valeur donc index 7
18     mov word [tab + rax*2], 8 ; multiplier par 2 car RESW donc 2 bytes par case
19
20 ;; end
21
22     mov rax, 60
23     mov rdi, 0
24     syscall
```

### 7.6.2 Déclarer et remplir un tableau

```
1  ;; td06_ex2.asm
2
3  global main
4
5  ;; Pour observer les 100 cases du tableau avec 100 bytes par cellule,
6  ;; On fait /d(long long[100])tab dans "Watched expressions" pour KDBG.
7
8  section .bss
9
10     tab resq 100                ; Tableau de 100 entiers de 8 bytes chacun
11
12 section .text
13
14 main:
15
16     mov rax, 0                  ; compteur ET indice
17
18 _for_less_than_100:
19     cmp rax, 100                ; Pour RAX de 0 à 99
20     jz _end                     ; Sauter à la fin si RAX == 100
21     mov qword [tab + rax * 8], rax ; On assigne au tableau par sauts de 8 bytes
22     inc rax                     ; incrémentation de RAX
23     jmp _for_less_than_100      ; Retourner dans le loop
24
25 _end:
26
27     mov rax, 60
28     mov rdi, 0
29     syscall
```

### 7.6.3 Remplir un tableau inversé

```
1  ;; td06_ex3.asm
2
3  global main
4
5  section .bss
6
7      tab resq 100                ; tableau de 100 entiers initialisés à 0
8
9  section .text
10
11  main:
12
13      mov rax, 0                  ; indice
14      mov rcx, 99                ; indice inversé
15
16  _while_less_than_100:
17      cmp rax, 100               ; tant que RAX < 100
18      jz _end                   ; Sauter à la fin
19      mov qword [tab + rax * 8], rcx ; Mettre la valeur de RCX dans le tableau
20      inc rax                   ; incrémenter RAX de 1
21      dec rcx                   ; décrémenter RCX de 1
22      jmp _while_less_than_100   ; loop
23
24  _end:
25
26  ;; end
27
28      mov rax, 60
29      mov rdi, 0
30      syscall
```

### 7.6.4 Remplir un tableau par sauts de 2

```
1  ;; td06_ex4.asm
2
3  global main
4
5  section .bss
6
7      tab resq 100                ; tableau de 100 entiers initialisés à 0
8
9  section .text
10
11  main:
12
13      mov rax, 0                  ; indice
14      mov rcx, 0                  ; valeur
15
16  _while_less_than_100:
17      cmp rax, 100               ; tant que RAX < 100
18      jz _end                   ; saut vers la fin
19      mov qword [tab + rax * 8], rcx ; On met la valeur de RCX au bon indice dans le tableau
20      inc rax                   ; incrémentation de RAX de 1
21      inc rcx                   ; incrémentation de RCX de 2
22      inc rcx
23      jmp _while_less_than_100   ; loop
24
25  _end:
26  ;; end
27
```

```

28     mov rax, 60
29     mov rdi, 0
30     syscall

```

### 7.6.5 Compter le nombre d'éléments pairs d'un tableau

```

1  ;; td06_ex5.asm
2
3  global main
4
5  ;; Pour observer :
6  ;; le tableau tab, on fait /d(long long[10])tab
7  ;; la longueur tab_len, on fait /d(long long)tab_len
8  ;; le nombre d'éléments nb_elem, on fait /d(long long)nb_elem
9  ;; Le registre R15 en décimal, /d$r15
10
11 section .rodata
12
13     tab      dq 1, 2, 3, 4, 5, 6, 7, 8, 10, 12 ; 10 éléments arbitraires
14     tab_len  dq $ - tab                      ; longueur du tableau : 10 * 8 bytes
15     nb_elem  dq (tab_len - tab) / 8 ; nbr éléments : (lg tab - tab) / taille case
16
17 section .text
18
19 main:
20
21     mov rax, 0                ; indice
22     mov r15, 0                ; compteur de valeurs paires
23
24 _while_less_than_nb_elem:
25     cmp rax, [nb_elem] ; tant que RAX < nb_elem, on compare la valeur donc []
26     jz _end
27     bt qword [tab + rax * 8], 0 ; on teste le (0ème) bit, CF prend sa valeur
28     jc _odd                  ; Si CF = 1, on va vers _odd
29     inc r15                  ; Si c'est pair, on incrémente R15
30 _odd:
31     inc rax                  ; on incrémente RAX dans tous les cas
32     jmp _while_less_than_nb_elem ; loop
33
34
35 _end:
36
37     mov rax, 60
38     mov rdi, 0
39     syscall

```

### 7.6.6 Trouver les extrema d'un tableau

```

1  ;; td06_ex6.asm
2
3  global main
4
5  section .rodata
6
7     tab      dq 4, 3, 2, 1, 8, 9, 11, 0, 100, 7 ; tableau de 10 entiers arbitraires
8     tab_len  dq $ - tab                      ; longueur du tableau
9     nb_elem  dq (tab_len - tab) / 8          ; nbr d'éléments du tableau
10
11 section .text
12
13 main:

```

```

14
15     mov rax, 0                ; indice
16     mov r8, [tab]            ; on initialise le min avec la 1ère valeur de tab
17     mov r10, [tab]           ; on initialise le max avec la 1ère valeur de tab
18
19 _while_less_than_nb_elem:
20     cmp rax, [nb_elem]        ; tant que RAX < nbr d'éléments
21     jz _end                   ; saut vers la fin
22     mov rcx, [tab + rax * 8]  ; on met la valeur de la case dans RCX
23     cmp rcx, r8               ; Si RCX < R8 => SF = 1
24     jns _greater_than_min     ; si SF = 0 on va vers _greater_than_min
25     mov r8, rcx               ; si SF = 1, R8 <- RCX
26     jmp _else                 ; saut vers la fin
27 _greater_than_min:
28     cmp rcx, r10              ; Si RCX >= R10 => SF = 0
29     js _else                  ; saut vers la fin
30     mov r10, rcx              ; Si SF = 0, R10 <- RCX
31 _else:
32     inc rax                   ; On incrémente RAX dans tous les cas
33     jmp _while_less_than_nb_elem ; loop
34
35 _end:
36 ;; end
37
38     mov rax, 60
39     mov rdi, 0
40     syscall

```

### 7.6.7 Afficher 10 premiers chiffres

```

1  ;; td06_ex7.asm
2
3  global main
4
5  section .bss
6  ;; On initialise 2*4 bytes
7  ;; le premier pour le caractère et le second pour l'échappement
8
9      charToShow resd 2
10
11 section .text
12 main:
13
14     mov r8, 0                ; compteur
15
16 _while_less_than_10:
17     cmp r8, 10
18     jz _end
19     mov r9, r8
20     or r9, 00110000b          ; masque pour convertir en ASCII / UTF-8
21     mov [charToShow], r9
22     mov dword [charToShow + 4], '\n' ; On met l'échappement
23
24     mov rax, 1                ; write
25     mov rdi, 1                ; stdout
26     mov rsi, charToShow       ; adresse du 1er caractère
27     mov rdx, 8                ; on lit exactement 8 bytes
28     syscall
29
30     inc r8
31     jmp _while_less_than_10
32

```

```

33 _end:
34 ;; end
35
36     mov rax, 60
37     mov rdi, 0
38     syscall

```

### 7.6.8 Afficher les 10 premier chiffres depuis un tableau

```

1  ;; td06_ex8.asm
2
3  global main
4
5  section .rodata
6
7      digits db `0123456789\n`
8
9  section .text
10
11 main:
12
13     mov r8, 0
14     mov r9, digits
15     mov r10, digits
16     add r10, 10
17
18 _while_less_than_10:
19     cmp r8, 10
20     jz _end
21
22     mov rax, 1          ; write
23     mov rdi, 1          ; stdout
24     mov rsi, r9         ; adresse du 1er caractère
25     mov rdx, 1          ; on lit exactement 1 byte
26     syscall
27
28     mov     rax, 1      ; write
29     mov rdi, 1          ; stdout
30     mov rsi, r10        ; adresse du 1er caractère
31     mov rdx, 1          ; on lit exactement 1 byte
32     syscall
33
34     inc r8
35     inc r9
36     jmp _while_less_than_10
37
38 _end:
39
40     mov rax, 60
41     mov rdi, 0
42     syscall

```

### 7.6.9 Lire depuis le clavier et afficher

```

1  ;; td06_09.asm
2
3  global main
4
5  section .bss
6
7      tmp resb 1

```

```

8
9 section .text
10 main:
11
12 _start_loop:
13     ; *tentative de* lecture au clavier
14     mov     rax, 0          ; read
15     mov     rdi, 0          ; clavier
16     mov     rsi, tmp        ; adresse à lire
17     mov     rdx, 1          ; nbr de bytes à lire
18     syscall
19
20     cmp     rax, 0          ; Si RAX == 0 => fin parce que 0 bytes lus
21     jz      _end_loop
22
23     ; affichage à l'écran
24     mov     rax, 1          ; write
25     mov     rdi, 1          ; écran
26     mov     rsi, tmp        ; adresse à lire
27     mov     rdx, 1          ; nbr de bytes à lire
28     syscall
29
30     jmp     _start_loop
31
32 _end_loop:
33     mov     rax, 60
34     mov     rdi, 0
35     syscall

```

## 8 Laboratoire 7 : Pile et variables locales

Ce TD traite de la pile ou *stack*. On parlera de son fonctionnement ainsi que de son utilisation dans le cadre des variables locales.

Quand on exécute un programme compilé, on se rend compte qu'une partie que certaines adresses portent le nom de `[stack]`, c'est la fameuse pile.

### 8.1 Pile

La pile (*stack*) d'un processus est un espace mémoire dans lequel il lui est possible de lire et d'écrire. Cet espace est utilisé à diverses fins. Par exemple, les arguments du programme y sont placés. Aussi, lors de l'appel d'une fonction, l'adresse de retour au code appelant y est stockée. Par ailleurs, les arguments de fonctions peuvent, dans certains cas, y être placés. Encore, les variables locales peuvent y vivre.

#### 8.1.1 Registres associés à la pile

Parmi les registres du processeur, RSP et RBP, bien que généraux, sont automatiquement destinés à la gestion de la pile. Le registre RSP (*register stack pointer*) contient l'adresse du (premier octet) du dernier élément empilé. Le registre RBP (*register base pointer*) est, quant à lui, un registre qui permet de se balader sur la pile afin de récupérer ou modifier le contenu d'un élément sans le dépiler. Remarquez que le remplissage de la pile se fait en remontant dans la mémoire : le deuxième élément de la pile est placé en mémoire juste avant le premier, c'est-à-dire à une plus petite adresse que le premier et sans trou entre lui et le premier ; le troisième juste avant, en mémoire, le deuxième, etc. Un élément mis sur la pile précédemment un autre est placé à une adresse plus grande que cet autre. Dit encore autrement, les éléments empilés successivement à d'autres se trouvent à des adresses mémoire plus petites que ces autres. À chaque empilement, la valeur du registre rsp est décrémentée d'une quantité égale à la taille de l'élément empilé. Inversement, à chaque dépilement, la valeur de RSP augmente de la taille de la donnée dépilée.

### 8.1.2 Instructions d'accès à la pile

Instruction	Contrainte	Effet	Flags affectés
<code>push X</code>	X est un registre ou une variable de 16 ou 64 bits	X est empilé et <code>rsp</code> est décrémenté de la taille en <i>bytes</i> de X	aucun
<code>push X</code>	X est un immédiat sur 16 bits	X est empilé et <code>rsp</code> est décrémenté de 2 (la taille de X en <i>bytes</i> )	aucun
<code>push X</code>	X est un immédiat sur 8 ou 32 bits	X est étendu par extension de signe sur 64 bits et empilé tandis que <code>rsp</code> est décrémenté de 8	aucun
<code>push X</code>	X est un immédiat sur 64 bits	Les 32 bits de poids faible de X sont étendus par extension de signe sur 64 bits et empilés tandis que <code>rsp</code> est décrémenté de 8	aucun
<code>pop X</code>	X est un registre ou une variable de 16 ou 64 bits	X reçoit le résultat du dépilement d'un nombre de <i>bytes</i> égal à sa taille et <code>rsp</code> est incrémenté d'autant	aucun

## 8.2 Instructions `ADD` et `SUB`

Instruction	Effet	Contraintes	Flags affectés
<code>add X, Y</code>	$X \leftarrow X + Y$	Registres ou variables de 8, 16, 32 ou 64 bits, pas deux variables, Y peut être un immédiat (8, 16 ou 32 bits)	<code>SF</code> ← bit de rang le plus élevé du résultat <code>ZF</code> ← 1 si résultat nul, 0 sinon <code>CF</code> et <code>OF</code> sont modifiés d'une manière qui dépasse le cadre des MICL
<code>sub X, Y</code>	$X \leftarrow X - Y$		

TABLE 2 – Instructions `add` et `sub`.

## 8.3 Exemples d'utilisation de la pile

### 8.3.1 Accès à la pile avec `PUSH` et `POP`

Voir code assembleur du TD. En gros, on place des valeurs sur la pile avec `push` ce qui **décrémente** `RSP`. On enlève des valeurs de la pile avec `pop`, ce qui **incrémente** `RSP`. Il y a aussi des limitations dues aux tailles des registres et des immédiats qu'on utilise. Attention donc de faire des opérations légales avec les registres et de se méfier des extensions de signe avec les immédiats. On n'empile pas des *r/m* 8 ou 32 bits sur la pile !

On peut mettre des immédiats de 8 ou 32 bits sur la pile mais attention à l'extension dans ce cas.

Aussi, on met d'abord les bits de poids fort dans la pile. Cela signifie que lorsqu'on prend des valeurs de la pile, on enlève par la fin des valeurs qui avaient été empilées.

### 8.3.2 Accès à la pile sans `PUSH` ni `POP`

On peut aussi se passer des instructions `PUSH` et `POP` on incrémentant et décrémentant le registre `RSP` nous-mêmes et en chargeant les mémoires avec `MOV`.

## 8.4 Variable locale

En assembleur, on peut créer des variables locales en passant par la pile. Normalement, on utilise les registres qui sont plus rapides car les accès sont plus rapide que ceux en mémoire. Cependant, ce n'est pas toujours possible d'utiliser des registres et on sera parfois amené à utiliser la mémoire et donc la pile pour stocker des données.

Pour créer une variable locale sur la pile, on peut créer les variables une à une en utilisant successivement l'instruction `push` ou toutes les créer en une fois. Dans ce dernier cas, on creuse un trou sur la pile en décrémentant le registre `RSP` de la taille en bytes de l'ensemble des variables locales voulues. L'espace ainsi créé sur la pile est celui pour les variables locales. On accède à cette zone mémoire par le biais du registre `RBP` plutôt que `RSP`. Celui-la

doit être correctement initialisé : sa valeur est sauvegardée et fixée en début de bloc. On préfère utiliser RBP à RSP car ce dernier est automatiquement mis à jour par `push` et `pop`, ce qui complique le calcul des accès aux variables locales sur la pile. Pour détruire les variables locales, il suffit de combler le trou de pile à l'aide d'une série de `pop`, ou en augmentant RSP de la valeur dont il a été diminué pour leur création. Pour décrémenter / incrémenter le registre RSP et créer / combler un trou de plusieurs bytes sur la pile, nous pourrions utiliser les instructions `dec` `inc` du TD06. S'il s'agit simplement de modifier RSP, il est plus simple d'utiliser les instructions `sub` et `add` qui ont été présentées précédemment.

Voici le premier exemple sans les commentaires :

```

1  global main
2  section .text
3
4  main:
5
6      push rbp      ; (1)
7      mov rbp, rsp
8
9      push qword 4  ; (2)
10
11     push qword -8 ; (3)
12
13     mov qword [rbp - 8], 23 ; (4)
14
15     sub qword [rbp - 8], 4 ; (5)
16
17     inc qword [rbp - 16] ; (6)
18
19     mov rsp, rbp ; (7)
20
21     pop rbp ; (8)
22
23 ;; end
24     mov rax, 60
25     mov rdi, 0
26     syscall

```

Maintenant on peut commenter ce qu'il se passe.

1. Le contenu de RBP est sauvé sur la pile. Comme il n'y a pas de contenu, on sauve en pratique 0x00 sur 8 *bytes*. On décrémente RSP du nombre de *bytes* qu'on vient de placer, donc  $RSP \leftarrow RSP - 8$ .  
Puis, on met à jour le pointeur de contexte de pile. À ce stade, RBP et RSP pointent vers le même endroit sur la pile.
2. On dépose une variable de 8 *bytes* sur la pile, elle vaut 4. RSP s'incrémente automatiquement de 8 mais RBP reste à sa place et pointe donc 8 *bytes* plus bas sur la pile.
3. On dépose une autre variable de 8 *bytes* sur la pile, elle est déposée 8 *bytes* plus haut que la précédente car RSP s'incrémente automatiquement. RBP reste à sa place et à maintenant un décalage de 16 *bytes* avec RSP. Elle vaut -8.
4. RBP est toujours 16 *bytes* plus loin que le haut de la pile (RSP), si on remonte de 8 *bytes* (en décrémentant !) on arrive à la valeur de la variable définie à cet endroit. De ce fait, comme elle n'a pas de valeur ailleurs, elle passe bien de 4 à 23.
5. On soustrait 4 à la valeur présente 8 *bytes* plus haut que RBP et donc on passe de 23 à 19.
6. On incrémente la valeur qui est 16 *bytes* plus haut dans la pile que RBP, on est en haut de pile et on y trouve le valeur -8, elle vaut maintenant -7.
7. On déplace RSP pour retrouver le niveau initial dans la pile, au même endroit que RBP. On détruit les variables locales.
8. Restauration de la valeur de RBP.

## 8.5 Exercices



### 8.5.1 Échanger le contenu de registre, via la pile

```
1 ; b131_ex01.asm
2
3 global _start
4
5 section .text
6 _start:
7     mov     rax, 27
8     mov     rbx, -9
9
10    push    rax        ; { int tmp = rax;
11    mov     rax, rbx    ;   rax = rbx;
12    pop     rbx        ;   rbx = tmp; }
13
14    push    rax        ; { int i = rax;
15    push    rbx        ;   { int j = rbx;
16    pop     rax        ;     rax = j; }
17    pop     rbx        ;   rbx = i; }
18
19    mov     rax, 60
20    mov     rdi, 0
21    syscall
```

### 8.5.2 Échanger le contenu de variables via la pile

```
1 ;; td07_ex2.asm
2
3 global main
4
5 ;; On observe avec /d(long)var
6
7 section .data
8     var1    dq    123
9     var2    dq    11
10
11 section .text
12
13 main:
14
15     push qword [var1]        ; On met la valeur de var1 sur la pile
16     push qword [var2]        ; On met la valeur de var2 sur la pile
17     pop  qword [var1]        ; var1 prend la valeur de haut de pile
18     pop  qword [var2]        ; var2 prend la valeur après dans la pile
19
20
21 ;; end
22     mov rax, 60
23     mov rdi, 0
24     syscall
```

### 8.5.3 Créer une variable locale sur la pile et sans variable globale

```
1 ; b132_ex03.asm
2
3 global _start
4
5 section .text
6 _start:
7     ; sauvegarde du contexte de pile
8     ; (inutile ici mais c'est une bonne habitude)
```

```

9      push    rbp
10     ; création du nouveau contexte de pile
11     mov     rbp, rsp
12
13     ; création variable locale
14     sub     rsp, 1
15     ; initialisation
16     mov     byte [rbp - 1], 8
17
18     ; corps du bloc
19
20     ; transformation binaire -> caractère
21     ; il faut setter les bits de rangs 4 et 5
22     ; bts     byte [rbp - 1], 4
23     ; bts     byte [rbp - 1], 5
24     ; les 2 instructions ci-dessus ko car dest r/m 16, 32 ou 64
25
26     ; utiliser masque
27     ; or      byte [rbp - 1], .....b
28     or      byte [rbp - 1], 00110000b
29
30     ; affichage à l'écran
31     mov     rax, 1      ; write
32     mov     rdi, 1      ; stdout
33     mov     rsi, rbp
34     sub     rsi, 1      ; contenu de rsi : adresse de la variable locale
35     mov     rdx, 1      ; nombre de bytes à écrire
36     syscall
37
38     ; destruction variable locale
39     mov     rsp, rbp
40
41     ; restauration de l'ancien contexte de pile
42     pop     rbp
43
44     mov     rax, 60
45     mov     rdi, 0
46     syscall

```

#### 8.5.4 Variable locale de dix *bytes* sur la pile

```

1      ;; td07_ex4.asm
2
3      global main
4
5      section .text
6
7      main:
8
9      ;; Sauver le contexte de pile
10     push rbp
11     ;; Création du nouveau contexte de pile
12     mov rbp, rsp
13     ;; Création d'une variable locale sur 10 bytes
14     ;; On crée un trou en soustrayant 10 à RSP
15     ;; RSP est 10 bytes au-dessus de RBP
16     sub rsp, 10
17
18     ;; Boucle pour initialiser et lire
19     mov bl, 0
20     mov r8, -10
21     for_ten_bytes:

```

```

22     cmp r8, 0
23     jz end
24     mov byte [rbp + r8], bl
25     or byte [rbp + r8], 0x30
26
27     inc bl
28     inc r8
29     jmp for_ten_bytes
30 end:
31     mov     rax, 1      ; write
32     mov     rdi, 1      ; stdout/écran
33     mov     rsi, rbp
34     sub     rsi, 10     ; contenu de rsi : adresse de la variable locale
35     mov     rdx, 10     ; nombre de bytes à écrire
36     syscall
37 ;; stop
38     mov rax, 60
39     mov rdi, 0
40     syscall

```

### 8.5.5 Modifier une variable locale sur la pile

```

1 ; b131_ex05.asm
2
3 global _start
4
5 section .text
6 _start:
7
8     ; mettre du déchet sur la pile
9     push    qword -20124587
10 ;     pop     rax
11     push    qword -212121212
12     pop     rax
13     pop     rax
14
15     mov     rdi, 230
16
17     ; sauvegarde du contexte de pile
18     ; (inutile ici, mais bonne habitude)
19     push    rbp
20
21     ; création du nouveau contexte de pile
22     mov     rbp, rsp
23
24     ; création de la variable locale
25     ; on pourrait faire : push qword 0
26     ; mais on fait ici plus compliqué pour voir
27     ; comment faire avec sub etc.
28     sub     rsp, 8      ; trou de 8 bytes sur la pile
29
30     ; mettre les bits de la variable locale de Orangs 1 -> 62 à 0
31     ; and     [rbp - 8], 0x80_00_00_00_00_00_00_01
32     ; 1000_0000_0000_..._0000_0001b
33     ; problème : and mem64, imm64 n'existe pas
34     ; mais on a imm32 étendu par extension de signe
35     ; => il faut passer par and mem64, reg64
36     mov     rax, 0x80_00_00_00_00_00_00_01
37     and     [rbp - 8], rax
38
39     ; si rdi impair : bit de rang 0 à 0
40     ; si rdi pair : bit de rang 0 à 1

```

```

41     bt     rdi, 0      ; cf <- bit de rang 0 de rdi
42     jnc    rdi_pair
43 rdi_impair:
44     btr    qword [rbp - 8], 0
45     jmp     suite
46 rdi_pair:
47     bts    qword [rbp - 8], 0
48
49 suite:
50
51     ; ici il faut compter le nombre de bits de rdi qui sont à 1
52     ; dans une boucle de compteur 0 -> 63 avec bt rdi, compteur
53     ; puis modifier en conséquence le bit de rang 63 de la variable locale
54     ; TODO
55
56     ; destruction de la variable locale
57     mov     rsp, rbp
58
59     ; restauration du contexte de pile
60     pop     rbp
61
62     ; fin
63     mov     rax, 60
64     mov     rdi, 0
65     syscall

```

#### 8.5.6 Lecture de fichier et variables locales

#### 8.5.7 TODO

## 9 Crédits

Cours de MIC2 et laboratoire MIC2 à l'ESI, 2020-21.