Notes du laboratoire de microprocesseur (MIC2-micl)

Nathan Furnal

30 avril 2021

Table des matières

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*).

```
; comment
global main
section .text
main:
    nop ; ne fait rien

    ; met l'immédiat en hexadécimal dans le registre rax
    ; grâce à l'instruction mov
    mov rax, 0x1122334455667788

    ; fin
    mov rax, 60
    mov rdi, 0
    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 mnémonique 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 déboguage

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.

```
al : 10011101b
not al
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

```
; exercice 2 du td 2 de MICL

global main
section .text
main:
;;; 'd' : 0110 0100b
;;; and : 0101 1111b
;;;       0100 0100b
;;; 'D' : 0100 0100b
    mov ah, al
    and ah, 01011111b

    ; fin
    mov rax, 60
    mov rdi, 0
    syscall
```

3.5.3 Exercice 3

```
        ;; exercice 3 du td 02

global main
section .text
main:
        mov bx, 'd' ; charge le caractère 'd' dans le registre al
        btc bx, 5   ; complémentaire du bit en position 5

        ;; fin
        mov rax, 60
        mov rdi, 0
        syscall
```

3.5.4 Exercice 4

```
        ;; exercice 4 du td 02 de MICL

global main
section .text
main:
        mov bl, 00000101b        ; 5 en binaire
        ;; On veut donner la valeur en ascii
        ;; masque `or` pour passer en ascii
        or bl,  00110000b
        ;; fin
        mov rax, 60
        mov rdi, 0
        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.

```
        ;; Montrer l'usage de cmp avec des commandes simples
global main
section .text
main:
        mov rax, 4

        cmp rax, 4                ; ZF : 1 (4 - 4 == 0), SF: 0 (4 - 4 >= 0)
        cmp rax, 5                ; ZF : 0 (4 - 5 != 0), SF: 1 (4 - 5 < 0)
        cmp rax, 2                ; ZF : 0 (4 - 2 != 0), SF: 0 (4 - 2 >= 0)

        ;; fin
        mov rax, 60
        mov rdi, 0
        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 rendre 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 :
  `0b11111111111111111111111110000000100000001000000010000000`
- 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*.

```
    mov rax, 123
    mov rbx, 150

    jmp fin     ; saut inconditionnel vers la fin

    mov rax, 0  ; jamais exécuté
    mov rbx, 0  ; jamais exécuté

```

```
fin:
    mov rax, 60
    mov rdi, 0
    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 à ces outils en main, on peut les utiliser pour sauter vers les instructions qui nous intéresse, 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.

```
    bt rax, 0  ;; Test de parité, est ce que le premier bit vaut zéro
    jc endIf
    mov rbx, 5
endIf:
    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.

```
    cmp rax, -10
    jns endIf
    mov rbx, 5
endIf:
    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.

```
    cmp rax, rdx
    jz endIf
    mov rbx, 5
endIf:
    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.

```
    bt rax, 0
    jc _sinon
    mov rbx, 5
    jmp _fin_si
_sinon:
    mov rbx, 6
_fin_si:
    mov rcx, 12
```

Voici un exemple de test d'infériorité :

```
    cmp rax, -10
    jns _sinon
    mov rbx, 5
    jmp _fin_si
_sinon:
    mov rbx, 6
_fin_si:
    mov rcx, 12
```

Voici un exemple de test d'inégalité :

```
    cmp rax, rdx
    jns _sinon
    mov rbx, 5
    jmp _fin_si
_sinon:
    mov rbx, 6
_fin_si:
    mov rcx, 12
```

4.5 Exercices

4.5.1 Exercice 1

```
;;; TD3 Exo 1

global main
section .text
main:
        mov rax, 42                ; Met 10 dans rax
        cmp rax, 0
        jnz _if_not_null
        jmp _end
_if_not_null:
        mov rbx, 1
_end:
        mov rax, 60
        mov rdi, 0
        syscall
```

4.5.2 Exercice 2

```
;;; TD3 Exo 2

global main
```

```
section .text
main:
        mov rax, 42
        bt rax, 0
        jc _if_not_even
        mov r8, 0
        jmp _end
_if_not_even:
        mov r8, 1
_end:
        mov rax, 60
        mov rdi, 0
        syscall
```

4.5.3 Exercice 3

```
;;; TD3 Exo 3

global main
section .text

main:
        mov r14, 42
        mov r15, 43
        cmp r14, r15
        jnz _if_not_equal
        mov r14, 0
        mov r15, 0
        jmp _end

_if_not_equal:
        xor r14, r15
        xor r15, r14
        xor r14, r15
_end:
        mov rax, 60
        mov rdi, 0
        syscall

;;; Ici on utilise le xor swap, sachant que le xor est commutatif
;;; 1. xor r14, r15 applique le xor et stocke le résultat dans r14
;;; 2. xor r15, r14 applique le xor et stocke le résultat dans r15
;;; 3. xor r14, r15 applique le xor et stocke le résultat dans r14
;;; En pratique les étapes font pour x valant 1010 et y valant 0011
;;; xor x, y <=> xor 1010, 0011 = 1001 -> x
;;; xor y, x <=> xor 0011, 1001 = 1010 -> y
;;; xor x, y <=> xor 1001, 1010 = 0011 -> x
```

4.5.4 Exercice 4

```
;;; TD3 Exo 4

global main
section .text
main:
        mov rax, 42
        mov rbx, 11
        cmp rax, rbx
        js _rax_is_less
        mov r8, rax
        mov r9, rbx
```

```
        jmp _end
_rax_is_less:
        mov r8, rbx
        mov r9, rax
_end:
        mov rax, 60
        mov rdi, 0
        syscall
```

4.5.5 Exercice 5

```
;;; TD3 Exo 5

global main
section .text
main:
        mov rdi, 11
        bt rdi, 0               ; CF vaut 0 si au moins multiple de 2
        ;; On envoie vers les multiples de 2 sinon on termine
        jnc _if_even
        mov rsi, 0
        jmp _end
_if_even:
        mov rsi, 1
        bt rdi, 1                ; CF vaut 0 si au moins multiple de 4
        ;; On envoie vers les multiples de 4 sinon on termine
        jnc _if_multiple_of_4
        jmp _end

_if_multiple_of_4:
        mov rsi, 2
        bt rdi, 2               ; CF vaut 0 si au moins multiple de 8 au plus
        ;; On envoie vers les multiples de 8 sinon on termine
        jnc _if_multiple_of_8
        jmp _end

_if_multiple_of_8:
        mov rsi, 3
_end:
        mov rax, 60
        mov rdi, 0
        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 crées 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
`.text`	Instructions exécutables du programme
`.data`	Variables globales explicitement initialisées
`.rodata`	Variables globales explicitement initialisées en lecture seule
`.bss`	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	`DB`	**D***efine* **B***yte*
2	`DW`	**D***efine* **W***ord*
4	`DD`	**D***efine* **D***oubleword*
8	`DQ`	**D***efine* **Q***uadword*

Enfin, voici un exemple d'utilisation en assembleur :

```
section .data
    i1 DB -1                ; 1 byte initialisé
    i2 DW 23                ; 2 bytes initialisés
    i4 DD -1                ; 4 bytes initalisés
    i8 DQ 0x80_00_00_00_00  ; 8 bytes initialisés donc 0x00_00_00_80_00_00_00_00

section .rodata
    ci8 DQ 93229

section .text
;...
```

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	`RESB`	**Res***erve* **B***yte*
2	`RESW`	**Res***erve* **W***ord*
4	`RESD`	**Res***erve* **D***oubleword*
8	`RESQ`	**Res***erve* **Q***uadword*

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.

```
section .bss
    ; tout est crée et initalisé à 0
    x1 RESB 10      ; 10 * 1 bytes réservés
    x2 RESW 6       ; 6 * 2 bytes réservés
    x4 RESD 100     ; 100 * 4 bytes réservés
    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.

```
section .data
    i4 DD 42              ; entier sur 4 bytes

section .text
    mov rax, i4           ; rax <-- _adresse_ (8 bytes) de i4.

```

```
    mov ebx, [i4]          ; ebx <-- _contenu_ de ce qui se trouve
                           ; à l'adresse i4 dans ebx.
                           ; s'étend sur 4 bytes car ebx est sur 4 bytes.

    mov ecx, [rax]         ; rax contient l'adresse de i4 donc [rax] = [i4].
                           ; donc, ecx <-- _contenu_ de l'adresse rax.
                           ; 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*.

```
section .data
    vw DW 0x0102  ; Valeur en hexadécimal stockée sur 4 bytes

;; à l'adresse vw : 0x02
;; à l'adress vw + 1 : 0x01
;; |...|...|0x02|0x01|...|...|
;; -> petite adresse --> grande adresse -->

    vq DQ 0x1122334455667788

;; à l'adresse vq : 0x88
;; à l'adresse vq+1 : 0x77
;; à l'adresse vq+2 : 0x66
;; à l'adresse vq+3 : 0x55
;; à l'adresse vq+4 : 0x44
;; à l'adresse vq+5 : 0x33
;; à l'adresse vq+6 : 0x22
;; à l'adresse vq+7 : 0x11

;; |...|0x88|0x77|0x66|0x55|0x44|0x33|0x22|0x11|...|
;; -> petite adresse --> grande adresse -->


;; vue complète de la section .data
;; | vw | vq | qu'on décompose en leurs bytes respectifs. Pas d'espaces entre adresses.
;; |...|0x02|0x01|0x88|0x77|0x66|0x55|0x44|0x33|0x22|0x11|..
;; -> petite adresse --> grande adresse -->

section .text
;...
mov r8b, [vw]      ; r8b <-- 0x02
mov r12b, [vq + 7] ; r12b <-- 0x11

;; pas de trous entre variables
;; vw est directement suivie par vq

mov r13b, [vw+2]   ; r13b <-- 0x88 (plus petit byte de vq)
mov r14b, [vw+3]   ; r14b <-- 0x77
mov r15b, [vq - 1] ; r15b <-- 0x01 (dernier byte de vw)
```

5.3 Amnésie 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 *bytes*	**Spécificateur**
1	`byte`
2	`word`
4	`dword`
8	`qword`

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 `ASCI` 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 :

```
section .data

c1 DB 'A'       ; c1 contient le code UTF-8 du caractère 'A'
c2 DB 65        ; c2 contient la valeur 65
;; 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

```
global main

section .data
    var1 DB 1
    var2 DB 2
    var3 DW 0x0304
    var4 DQ 0x00_00_00_00_80_00_FF_FF
;; la section des données occupe 12 bytes
;;; Contenu de la plus petite adresse à la plus grande
;;;    v1 v2 v3    v4
;;; ...01020403FFFF008000000000|...

section .text
main:
    mov rax, var1 ;  rax contient l'adresse de var1
;; (on ne connaît pas explicitement cette adresse, c'est l'OS qui la détermine).
    mov al, [var1] ; al contient 1
    mov ax, [var1] ; ax contient 0x0201 soit 0000_0010_0000_0001b
;; Car ax est un registre sur  2 bytes, il lit la valeur sur 2 bytes
;; Donc on prend en fait les valeurs de var1 et var2
    mov al, [var3] ; al contient 0x04 soit 0000_0100b
;; Car al est un registre sur 1 byte, on prend le premier byte de la valeur de var3
    mov ax, [var3] ; ax contient 0x0304 soit 0000_0011_0000_0100b
    mov rax, -1 ; rax contient 0xFF_FF_FF_FF_FF_FF_FF_FF par extension de signe
    mov eax, [var4] ; eax contient : 0x80_00_FF_FF (on ne lit que 4 byte en mémoire)
;; rax contient 0x00_00_00_00_80_00_FF_FF, les valeurs précédentes de rax sont écrasées
;; c'est le cas quand on utilise un registre 64bits avec une variable 32 bits.
    mov rax, 60
    mov rdi, 0
    syscall
```

5.5.2 Exercice 2

```
global main

section .data

    nb dd 11  ; Essayer avec C3_D8_10_00 pour bit de poids fort différent

section .text

main:
    mov eax, nb                 ; Placer adresse dans eax
    mov ebx, [nb]                ; Placer la valeur dans ebx

;; fin

    mov rax, 60
    mov rdi, 0
    syscall
```

5.5.3 Exercice 3

```asm
;; td04_ex3.asm

global main

section .bss

    nb resq 1 ; On réserve 1 x 8 bytes

section .text

main:

;; end

    mov byte [nb], 42

    mov rax, 60
    mov rdi, 0
    syscall
```

5.5.4 Exercice 4

```asm
;; td04_ex4.asm

global main

section .data

    b0 db 0
    b1 db 0
    b2 db 0
    b3 db 0

section .rodata
    nb dd 0x12_34_56_78

section .text

main:

    mov al, [nb]
    mov [b0], ax
    mov al, [nb+1]
    mov [b1], ax
    mov al, [nb+2]
    mov [b2], al
    mov al, [nb+3]
    mov [b3], al

;; end

    mov rax, 60
    mov rdi, 0
    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

```
;; td04_ex5.asm

global main

section .data

    var1 db 11
    var2 db 34

section .text

main:

    mov r8b, [var1]
    mov r9b, [var2]
    mov [var1], r9b
    mov [var2], r8b

;; end
    mov rax, 60
    mov rdi, 0
    syscall
```

5.5.6 Exercice 6

```
;; td04_ex6.asm

global main

section .data

    var1 db 11
    var2 db 34

section .bss

    var3 resb 1

section .text

main:
    mov r8b, [var1]
    mov r9b, [var2]
    cmp r8b, r9b
    js _var1_is_less
    mov al, [var2]
    jmp _else
_var1_is_less:
    mov al, [var1]

_else:
    mov [var3], al

;; end

    mov rax, 60
    mov rdi, 0
    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 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. Dan le cas de `exit`, aucune valeur n'est retournée.

Si maintenant on se concentre sur l'instruction `open`, on verra que son premier argument 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-terminé, 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_extrait.asm
2 global main
3
4 section .rodata
```

```
    nomFichier db 'brol', 0 ; ne pas oublier le zéro en fin

section .text

main:
    ; ouverture de brol en écriture seule avec placement
    ; de la tête d'écriture en fin de fichier

    mov rax, 2          ; open
    mov rdi, nomFichier ; /adresse/ du 1er caractère du nom
    mov rsi, 1q | 2000q ; WRONLY + O_APPEND
    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,

```
section .rodata

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

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

Le troisième exemple repris dan 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.

```
; 02_hello_world.asm

global main

section .rodata
    msg      DB  `Hello, Wolrd!\n`
    lgrMsg   DQ  lgrMsg - msg

section .text
```

```
main:
    ; affichage
    mov rax, 1          ; write
    mov rdi, 1          ; stdout
    mov rsi, msg        ; adresse du 1er caractère
    mov rdx, [lgrMsg]   ; nombre de caractères
    syscall
; end

mov rax, 60             ; exit
mov rdi, 0              ; ok
syscall
```

6.8 Cinq appels système

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

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

6.9 Exercices

6.9.1 Ouvrir un fichier

Quand le fichier `brol` 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 `brol` 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 `brol` 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 `brol` n'existe pas, on réussi à l'écrire dans ce cas-ci et `rax` contient 3.

6.9.2 Ouverture et écriture

```
;; td05_ex2.asm

global main

section .rodata
    nomFichier db `brol`, 0         ; nécessaire pour le nom du fichier
```

```
    msgOK       db `fichier ouvert avec succès\n`, 0
    lenOk       dq $ - msgOK
    msgFail     db `échec lors de l'ouverture du fichier\n`, 0
    lenFail     dq $ - msgFail

section .text
main:

    mov rax, 2                          ;open
    mov rdi, nomFichier                  ; fichier
    mov rsi, 0q                          ; RONLY
    syscall

    cmp rax, 0
    js _fail
;; affichage
    mov rbx, rax                ; sauver le descripteur du fichier
    mov rax, 1                          ; write
    mov rdi, 1                          ; stdout
    mov rsi, msgOK                ; adresse du premier char
    mov rdx, [lenOk]                ; nombre de caractères
    syscall

    mov rax, 3                          ; close
    mov rdi, rbx                ; descripteur du fichier
    syscall

    mov rax, 60                          ; end
    mov rdi, 0                          ; zero, pas d'erreur
    syscall

_fail:
    mov rax, 1                          ; write
    mov rdi, 1                           ; stdout
    mov rsi, msgFail                ; message d'erreur, adresse du premier char
    mov rdx, [lenFail]                  ; nombre de caractères
    syscall

    mov rax, 60                 ; end
    mov rdi, 1                          ; 1, erreur
    syscall
```

6.9.3 Convertir `rsi` en caractères

```
;; td05_ex3.asm

global main

section .bss
    charToShow resb 1

section .text
main:

    mov rsi, 7
    or rsi, 00110000b             ; masque pour convertir en ASCII
    mov [charToShow], rsi    ; Place la valeur de rsi dans la variable

;; affichage

    mov rax, 1                          ; write
    mov rdi, 1                          ; stdout
```

```
    mov rsi, charToShow                 ; Adress du premier caractère
    mov rdx, 1                          ; On lit exactement 1 byte
    syscall

;; end

    mov rax, 60
    mov rdi, 0
    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

```
;; td05_ex4.asm

global main
section .rodata
    pairMessage    db `Le contenu est pair\n`
    lenPair        dq $ - pairMessage
    impairMessage  db `Le contenu est impair\n`
    lenImpair      dq $ - impairMessage
section .text
main:
    mov rcx, 11

    bt rcx, 0                       ; Donne la valeur du 1er bit donc 1 si impair
    jc _impair

    mov rax, 1                          ; write
    mov rdi, 1                          ; stdout
    mov rsi, pairMessage        ; adresse du 1er caractère
    mov rdx, [lenPair]                  ; longueur du message
    syscall
    jmp _end

_impair:
    mov         rax, 1                          ; write
    mov rdi, 1                          ; stdout
    mov rsi, impairMessage        ; adresse du 1er caractère
    mov rdx, [lenImpair]        ; longueur du message
    syscall

_end:

    mov rax, 60
    mov rdi, 0
    syscall
```

6.9.5 Écrire des fichiers

```
;; td05_ex5.asm

global main
section .bss
    var resq 1                          ; Réserve 1x8 bytes
section .rodata
    nomPair   db   `pair`, 0
    nomImpair db  `impair`, 0

```

```
section .text

main:

    mov rcx, 77
    mov qword [var], rcx        ; sauver le contenu de rcx

    bt rcx, 0                       ; CF prend la valeur du 0eme bit de rcx
    jc _impair

    mov rax, 2                      ; open
    mov rdi, nomPair               ; adresse du 1er caractère du nom
    mov rsi, 1q                      ; WRONLY
    syscall
    jmp _write

_impair:
    mov rax, 2                      ; open
    mov rdi, nomImpair              ; adresse du 1er caractère du nom
    mov rsi, 1q                      ; WRONLY
    syscall

_write:
    mov rbx, rax                ; sauver le descripteur
    mov rax, 1                      ; write
    mov rdi, rbx                 ; descripteur du fichier
    mov rsi, var                ; adresse de ce qui doit être écrit
    mov rdx, 8                      ; Nombre de bytes à écrire
    syscall

;; end
    mov rax, 60
    mov rdi, 0
    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 :

$$T_t = nxT_e \tag{1}$$

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 :

```
section .data:

  tab db 0, 1, 2, 3
  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 :

```
section .bss:

  t1 resb 1
  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 :

```
section .rodata

  s1      db "abc"              ; 3 bytes
  s1_long db "a", "b", "c"      ; 3 bytes
  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 :

$$tab + (n-1) \times T_e \tag{2}$$

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 :

$$\text{offset} = \text{base} + \text{index} \times \text{scale} + \text{displacement} \tag{3}$$

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.