MICL : TD06 : Tableau et boucles

BEJ – DBO – DHA – HAL – NVS – SRE – YVO *

Année académique 2020 – 2021

Dans ce TD, les tableaux globaux et les techniques pour accéder à leurs contenus sont expliqués. Ensuite, les implémentations en langage d'assemblage des boucles *tant que*, *pour* et *répéter tant que* sont présentées, après l'introduction de deux instructions arithmétiques.

1 Tableau

Définition On s'en tient ici à une définition plutôt ancienne et rigide le la notion de tableau [1] (*array*). Un *tableau* est une structure de données caractérisée par deux propriétés :

— toutes les données du tableau sont du même type [2] ;

— les données du tableau se suivent en mémoire, collées les unes à la suite des autres [3].

Chaque donnée d'un tableau est appelée *élément* ou *cellule*.

En résumé, un tableau est un collection de variables toutes du même type et placées à des emplacements consécutifs en mémoire.

Langage d'assemblage On se limite ici aux tableaux globaux. Les tableaux locaux sont étudiés en même temps que les variables locales lors du TD07.

Au cours du TD04, on a vu qu'en langage d'assemblage, les données ne sont pas typées lors de leur déclaration. C'est lors de leur utilisation, à l'aide d'instructions,

*Et aussi, lors des années passées : ABS – BEJ – DWI – EGR – ELV – FPL – JDS – MBA – MCD – MHI – MWA.

1. https://fr.wikipedia.org/wiki/Tableau_(structure_de_donn%C3%A9es) (consulté le 21 mars 2020).
2. Cette contrainte disparaît en JavaScript ou PHP, par exemple.
3. Cette contrainte peut être assouplie (voir fin de la section 1.4).

qu'une signification leur est donnée. Seule la taille d'une variable est renseignée lors de sa définition. Un tableau est dès lors une collection de variables de même taille.

Il faut bien distinguer deux quantités :

— la taille d'un tableau : c'est le nombre de *bytes* qu'il occupe en mémoire ;

— le nombre d'éléments d'un tableau : c'est le nombre de variables qui le composent.

Comme tous les éléments ont la même taille, la taille du tableau, T_t, est toujours égale au nombre d'éléments du tableau, n, multipliée par la taille d'un élément, T_e :

$$T_t = n \times T_e$$

1.1 Sections `.data` et `.rodata`

Les sections `.data` et `.rodata` permettent de réserver un tableau initialisé avec des valeurs explicites.

On place une étiquette[4] et on utilise une pseudo-instruction de taille (`DB`, `DW`, etc.), comme lors de la définition d'une simple variable. Ensuite, à la place de fournir une seule valeur initiale, on donne une liste de valeurs séparées par une virgule[5]. Chacune des valeurs sert à l'initialisation d'un élément du tableau, dans l'ordre où elles apparaissent dans le code source. Il est aussi possible d'utiliser la pseudo-instruction `times`[6] pour imposer à `nasm` de répéter l'assemblage d'une instruction.

L'étiquette identifie l'adresse du premier *byte* du premier élément du tableau. Pour accéder au contenu du tableau, on utilise cette étiquette et on effectue des calculs d'adresses. Ceux-ci sont détaillés dans la suite du TD.

Par ailleurs, il est possible de faire calculer par `nasm` la taille ou le nombre d'éléments d'un tableau, par le biais de calculs d'adresses réalisés sur des *labels* judicieusement placés dans les sections `.data` et `.rodata`.

Exemples Voici un extrait de code source où des tableaux sont déclarés dans les sections `.data` et `.rodata` :

```
1  section .data
2      ; tableaux
3
4      ; t1 : tableau de 4 éléments de taille 1 byte chacun
5      t1          DB      0, 1, 2, 3
6
7      ; t2 : tableau de 3 (2 + 1) éléments de taille 4 bytes chacun
8      ; 1er élément : 'A' (0x41) codé sur 4 bytes
```

4. Pour rappel, il n'est pas nécessaire de terminer la définition d'un *label* par un deux-points (:) dans les sections `data`, `rodata` et `bss`.

5. On peut aussi, sur une nouvelle ligne, répéter la même pseudo-instruction de taille, suivie d'une valeur initiale, *sans* fournir d'étiquette. Cela revient à définir une variable anonyme initialisée collée à la précédente.

6. https://www.nasm.us/doc/nasmdoc3.html#section-3.2.5 (consulté le 20 mars 2020).

```
9       ;              : en mémoire petit-boutisme hex : 41 00 00 00
10      ; 2e élément : 'B' (0x42) codé sur 4 bytes
11      ;              : en mémoire petit-boutisme hex : 42 00 00 00
12      ; 3e élément : 'C' (0x43) codé sur 4 bytes
13      ;              : en mémoire petit-boutisme hex : 43 00 00 00
14      t2          DD      'A', 'B'
15                  DD      'C'       ; variable anonyme « attachée » à t2
16
17      ; tailles / nombre d'éléments de tableaux
18      ; obtenus pas calculs de différences d'adresses
19
20      ; tailleT1 : taille 8 bytes, contenu égal à 4 (4 × 1)
21      tailleT1    DQ      t2 - t1
22
23      ; tailleT2 : variable 2 bytes, contenu égal à 12 (3 × 4)
24      tailleT2    DW      tailleT1 - t2
25
26      ; nbElemT2 : variable 4 bytes, contenu égal à 3 (12 / 4)
27      nbElemT2    DD      (tailleT1 - t2) / 4
28
29  ; contenu de la section .data (byte par byte, hexadécimal) :
30  ; ---> petites adresses --->
31  ;       t1                    t2
32  ;  ... | 00 | 01 | 02 | 03 | 41 | 00 | 00 | 00 | 42 | 00 | .
33  ;
34  ;                                      tailleT1
35  ;     . 00 | 00 | 43 | 00 | 00 | 00 | 04 | 00 | 00 | 00 | ..
36  ;
37  ;                             tailleT2  nbElemT2
38  ;    .. 00 | 00 | 00 | 00 | 0C | 00 | 03 | 00 |00 | 00 | ...
39  ;
40  ;                                         ---> grande adresses --->
41
42  section .rodata
43
44      ; t3 : tableau de 4 éléments de taille 2 bytes chacun
45      ; 1er élément : 0xAB codé sur 2 bytes
46      ;              : en mémoire petit-boutisme hex : AB 00
47      ; 2e élément : 0xAB codé sur 2 bytes
48      ;              : en mémoire petit-boutisme hex : AB 00
49      ; 3e élément : 0xAB codé sur 2 bytes
50      ;              : en mémoire petit-boutisme hex : AB 00
51      ; 4e élément : 0xAB codé sur 2 bytes
52      ;              : en mémoire petit-boutisme hex : AB 00
```

```
53      t3  times 4     DW  0xAB
54
55      ; tailleT3 : taille 8 bytes, contenu égal à 8 (4 × 2)
56      ; $ : remplacé par l'adresse en début de ligne où il apparait
57      ;   : donc ici $ est égal à tailleT3
58      tailleT3        DQ  $ - t3
59
60  ; contenu de la section .rodata (byte par byte, hexadécimal) :
61  ; ---> petites adresses --->
62  ;       t3                                          tailleT3
63  ;  ... | AB | 00 | AB | 00 | AB | 00 | AB | 00 | 08 | 00 | .
64  ;
65  ;     . 00 | 00 | 00 | 00 | 00 | 00 | ...
66  ;                                       ---> grande adresses --->
```

Pour le calcul de la taille d'un tableau, on calcule la différence[7] entre l'adresse du premier *byte* juste au delà du tableau et celle du premier *byte* du tableau. Ces adresses sont obtenues par des étiquettes bien placées. Cette différence d'adresses fournit le nombre d'emplacement adressables existant entre les deux adresses. Comme la taille d'un emplacement adressable est précisément la définition du *byte*[8], cette différence est égale à la taille du tableau exprimée en *bytes*.

Plutôt que de recourir à deux étiquettes, on peut utiliser le symbole dollar[9] (`$`) qui est remplacé lors de l'assemblage par l'adresse du début de la ligne où il apparaît.

Pour le calcul du nombre d'éléments d'un tableau, on divise[10] sa taille par la taille d'un de ses éléments.

1.2 Section `.bss`

La `section .bss` est naturellement adaptée à la déclaration d'un tableau. L'étiquette et la pseudo-instruction pour la déclaration de variables (`RESB`, `RESW`, etc.) y sont en effet suivies non pas d'une valeur initiale pour la variable, mais du *nombre* de variables désirées, c'est-à-dire du nombre d'éléments du tableau. Chacune de ces variables est initialisée à la valeur 0. Si des valeurs différentes de 0 doivent se trouver dans les cellules du tableau, il faut les assigner dynamiquement lors de l'exécution du programme.

Notons de plus qu'un tableau déclaré dans la `section .bss` peut être de très grande taille, sans répercussion sur la taille de l'exécutable, contrairement aux tableaux des sections `.data` et `.rodata`. Pour davantage de détails à ce propos, référez-vous à la partie du TD04 dédiée aux sections pour la déclaration de variables.

Une étiquette dans la `section .bss` identifie l'adresse du premier *byte* du premier élément d'un tableau. L'accès au contenu du tableau, est obtenu suite à des calculs

7. https://www.nasm.us/doc/nasmdoc3.html#section-3.5.5 (consulté le 20 mars 2020).
8. https://fr.wikipedia.org/wiki/Byte (consulté le 20 mars 2020).
9. https://www.nasm.us/doc/nasmdoc3.html#section-3.5 (consulté le 20 mars 2020).
10. https://www.nasm.us/doc/nasmdoc3.html#section-3.5.6 (consulté le 20 mars 2020).

d'adresses, exactement de la même manière que pour les tableaux des sections `.data` ou `.rodata`.

Exemple Voici un bout de source où des tableaux sont déclarés dans la `section .bss` :

```
section .bss
    ; t1 : tableau d'1 élément de taille 1 byte
    t1          RESB    1

    ; t2 : tableau de 6 éléments de taille 2 bytes chacun
    t2          RESW    6

    ; t3 : tableau de 100 éléments de taille 4 bytes chacun
    t3          RESD    100

    ; t4 : tableau de 10 éléments de taille 8 bytes chacun
    t4          RESQ    10

; extrait du contenu de la section .bss (byte par byte, hexadécimal) :
; ---> petites adresses --->
;       t1   t2
;  ... | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | .
;
;                t3
;     . 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | ...
;
;                                        ---> grande adresses --->
```

1.3 Chaîne de caractères

On a affirmé dans le TD05 que les chaînes de caractères littérales sont des tableaux de caractères. Elles sont définis dans la `section .rodata`. Leurs contenus sont fournis entre guillemets, apostrophes ou accents graves. Avec ces derniers, les séquences d'échappement sont interprétées par `nasm`. La pseudo-instruction de spécification de taille qu'il est conseillé d'utiliser est `DB`. N'hésitez pas à retourner voir le TD05.

Pour ce qui concerne les chaînes de caractères modifiables non locales, on utilise des tableaux de *bytes* définis dans les sections `.data` ou `.bss`.

Exemple Voici un extrait de code source assembleur avec des chaînes de caractères :

```
section .rodata

    s1          DB      "abc"           ; 3 bytes
```

```
4     s1_long     DB      "a", "b", "c"
5     s1_alt      DB      0x61, 0x62, 0x63
6
7     s2          DB      'def', 0        ; 4 bytes, zéro-terminée
8     s2_long     DB      'd', 'e', 'f', 0
9     s2_alt      DB      0x64, 0x65, 0x66, 0x00
10
11    s3          DB      `ghi\n`         ; 4 bytes, \n échappé
12    s3_long     DB      `g`, `h`, `i`, `\n`
13    s3_alt      DB      0x67, 0x68, 0x69, 0x0a  ; \n GNU / Linux
14
15    s4          DB      'jkl\n'         ; 5 bytes
16    s4_long     DB      'j', 'k', 'l', '\', 'n'
17    s4_alt      DB      0x6a, 0x6b, 0x6c, 0x5c,0x6e
18
19    s5          DB      `mnô`   ; 4 bytes si utf-8 comme sur linux1
20    s5_long     DB      `m`, `n`, `ô`
21    s5_alt      DB      0x6d, 0x6e, 0xc3, 0xb4
22
23    s6          DW      "pqr"
24    s6_long     DB      "p", "q", "r", 0
25    s6_alt      DB      0x70, 0x71, 0x72, 0x00
26
27    s7          DQ      `stu`
28    s7_long     DB      `s`, `t`, `u`, 0, 0, 0, 0, 0
29    s7_alt      DB      0x73, 0x74, 0x75, 0x00, 0x00, 0x00, 0x00, 0x00
30
31  section .data
32
33    s10     times   20  DB '.'          ; 20 bytes à '.'
34
35  section .bss
36
37    s20         RESB    20              ; 20 bytes
```

Remarquez que les chaînes ou tableaux `s`*i*, `s`*i*`_long` et `s`*i*`_alt`, où *i* va de 1 à 7, sont strictement identiques.

Les chaînes `s6` et `s7` illustrent ce qui se passe [11] quand la taille des éléments du tableau sous-jacent n'est pas d'un *byte*.

11. https://www.nasm.us/doc/nasmdoc3.html#section-3.4.4 (consulté le 20 mars 2020).

1.4 Accès au contenu

En Java, par exemple, on utilise un indice entier pour accéder à un élément particulier d'un tableau. En langage d'assemblage, comme souvent, les choses sont un peu moins simples. Avec `nasm`, il faut réaliser le calcul explicite d'adresse pour obtenir celle du premier *byte* de l'élément désiré.

Considérons un tableau dont l'adresse du premier *byte* du premier élément est identifiée par le *label* `tab`. Les éléments de ce tableau font T_e *bytes*. On a que :

— l'adresse du premier élément du tableau, celui d'indice 0 dans les langages de plus haut niveau, est égale à : `tab` ;

— l'adresse du deuxième élément du tableau, celui d'indice 1 dans les langages de plus haut niveau, est égale à : `tab` $+ T_e$ puisqu'il faut sauter 1 élément au delà du premier pour atteindre le deuxième élément ;

— l'adresse du troisième élément du tableau, celui d'indice 2 dans les langages de plus haut niveau, est égale à : `tab` $+ 2 \times T_e$ puisqu'il faut sauter 2 éléments au delà du premier pour atteindre le troisième élément.

En toute généralité, l'adresse du énième élément du tableau, celui d'indice $n-1$ dans les langages de plus haut niveau, est égale à :

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

Attention lors d'un calcul de l'adresse d'un élément d'un tableau à bien obtenir l'adresse de son premier *byte*... et à ne pas déborder du tableau ! En cas de débordement, on accède au contenu d'une autre variable que le tableau, ou à un emplacement auquel le programme n'a pas le droit d'accéder, ce qui entraîne son arrêt brutal [12].

Dans la définition de la notion de tableau, on a imposé qu'en mémoire ses éléments soient collés les uns aux autres. Cela n'est pas strictement nécessaire. On peut remplacer cette contrainte par celle qui consiste à imposer qu'un élément d'un tableau doit pouvoir être atteint par un simple calcul d'adresse sur base de l'adresse du début de tableau et de l'indice de l'élément, comme on l'a fait ci-dessus.

Comme il s'agit ici d'utiliser un indice entier pour l'accès à un élément du tableau, les tableaux associatifs [13], où un élément est indexé via une *clé* d'un *type arbitraire*, sont exclus.

Jusqu'ici, nous avons obtenu l'adresse d'un élément d'indice donné d'un tableau. C'est bien, mais nous n'avons toujours pas accédé au *contenu* d'un tableau, contrairement au titre de la section ! Avant de voir des exemples de code sources où on accède effectivement au contenu de tableaux indicés (dans les exemples de la section 2.3), il faut parler des différents modes d'adressage disponibles.

2 Modes d'adressage

Il existe diverses manière de renseigner les opérandes d'une instruction. Certaines ont été vues au fur et à mesure de l'avancée des TD des laboratoire microprocesseur. Nous

12. https://en.wikipedia.org/wiki/Segmentation_fault (consulté le 20 mars 2020).
13. https://fr.wikipedia.org/wiki/Tableau_associatif (consulté le 21 mars 2020).

en faisons ici le bilan et profitons de l'occasion pour introduire de nouvelles techniques d'adressage des opérandes, particulièrement utiles pour l'accès aux données d'un tableau.

On trouve davantage d'information dans la documentation officielle d'Intel ([1], Vol. 1, section 3.7, p. 3-19). Nous n'abordons pas ici *tous* les modes d'adressage des opérandes disponibles pour les processeurs x86, mais uniquement ceux utilisés en MICL.

2.1 Immédiat

Il est possible de renseigner un opérande sous la forme d'une valeur immédiate, en ce compris les valeurs calculées par l'assembleur.

Exemple Voici un extrait de code source où des valeurs immédiates sont utilisées :

```
1  section .rodata
2      i       DQ      -234
3      s       DB      `hello`, 0
4
5  ; contenu de la section .rodata (byte par byte, hexadécimal) :
6  ; ---> petites adresses --->
7  ;       i                                       s
8  ;  ... | 16 | FF | FF | FF | FF | FF | FF | FF | 68 | 65 | 6C | .
9  ;
10 ;     . 6C | 6F | 00 | ...
11 ;                                    ---> grande adresses --->
12
13 section .text
14
15     mov     rbx, 112        ; 112 est évidemment un immédiat
16
17     mov     rcx, 1q | 100q | 2000q
18     ; l'expression « 1q | 100q | 2000q » est calculée par nasm
19     ; à l'exécution, on a la valeur immédiate 2101q (0x441, 1089)
20
21     mov     r8, i
22     ; i est un immédiat « calculé » par nasm, c'est l'adresse
23     ; du 1er byte de la « variable » nommée i (détails voir TD04)
24     ; sur ma machine, j'ai la valeur 0x402000
25
26     mov     r9, s
27     ; s est un immédiat « calculé » par nasm, c'est l'adresse
28     ; du 1er byte de la chaîne de caractères étiquetée s
29     ; sur ma machine, j'ai la valeur 0x402008 (ok : i + 8)
```

2.2 Registre

Il est possible de renseigner un opérande sous la forme d'un registre.

Exemple Voici un extrait de code source où des registres sont utilisées :

```
1 section .text
2
3     mov     rbx, rcx
4     ; le registre rcx est utilisé pour renseigner la source
5     ; le registre rdx est utilisé pour renseigner la destination
```

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[14] qui pointe[15] 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 un 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 :

$$\textit{offset} = \text{base} + \text{indice} \times \text{facteur d'échelle} + \text{déplacement}$$

ou, en anglais :

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

Chacune des quatre composantes d'un *offset* est facultative[16]. On peut ainsi par exemple rencontrer un *offset* constitué d'un déplacement seul[17], ou d'une base et d'un indice[18], sans déplacement ni facteur d'échelle, etc.

14. Pour rappel, il s'agit des registres : `rax`, `rcx`, `rdx`, `rbx`, `rsp`, `rbp`, `rsi`, `rdi`, `r8`, `r9`, `r10`, `r11`, `r12`, `r13`, `r14` et `r15` (voir TD01).
15. Nouveau rappel : « *pointer sur* un emplacement mémoire » signifie « *contenir l'adresse du premier byte* de cet emplacement mémoire ».
16. Étant donné le lien entre indice et facteur d'échelle, on ne rencontre pas de facteur d'échelle sans indice.
17. On parle alors d'*adressage direct* (*absolute* ou *direct address* en anglais).
18. Appelé parfois *adressage indicé* ou *adressage indexé*.

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 [19].

Exemple Voici un extrait de code source où divers *offsets* sont utilisées pour accéder au contenu de variables simples ou de tableaux situés en mémoire :

```
1  section .rodata
2
3      s       DB      `hello`, 0  ; chaîne de caractères zéro-terminée
4
5  ; contenu de la section .rodata (byte par byte, hexadécimal) :
6  ; ---> petites adresses --->
7  ;       s
8  ;  ... | 68 | 65 | 6C | 6C | 6F | 00 | ...
9  ;                                      ---> grande adresses --->
10
11 section .data
12
13     i       DQ      -234
14     t       DQ      -1, 1, -1   ; tableau de 3 quadwords
15
16 ; contenu de la section .data (byte par byte, hexadécimal) :
17 ; ---> petites adresses --->
18 ;         i                                          t
19 ;  .... | 16 | FF | FF | FF | FF | FF | FF | FF | FF | FF | .
20 ;
21 ;       . FF | FF | FF | FF | FF | FF | 01 | 00 | 00 | 00 | ..
22 ;
23 ;      .. 00 | 00 | 00 | 00 | FF | FF | FF | FF | FF | FF | ...
24 ;
25 ;     ... FF | FF | ....
26 ;                                      ---> grande adresses --->
27
28 section .bss
29
30     msg     RESB    5           ; tableau de 5 bytes
31
32 ; contenu de la section .bss (byte par byte, hexadécimal) :
33 ; ---> petites adresses --->
34 ;       msg
35 ;  ... | 00 | 00 | 00 | 00 | 00 | ...
```

19. https://www.nasm.us/doc/nasmdoc2.html#section-2.2.2 (consulté le 21 mars 2020).

```
36  ;                                       ---> grande adresses --->
37
38  section .text
39
40      ; ------------------------------------------------------
41      ; adressage direct : via un immédiat seul : déplacement
42      ; ------------------------------------------------------
43
44      ; le byte à l'adresse 0x402000 est copié dans r10b
45      ; chez moi s = 0x402000 ce qui aboutit à copier `h` dans r10b
46      ; rem. : utiliser mov r10b, [s]
47      mov     r10b, [0x402000]
48
49      ; les 8 bytes qui commencent à l'adresse i sont copiés dans r8
50      ; ce qui aboutit à copier -234 dans r8
51      mov     r8, [i]
52
53      ; les 8 bytes qui commencent à l'adresse t + 2 * 8, c.-à-d. le
54      ; 3e élement du tableau t, sont copiés dans r9, ce qui aboutit
55      ; à copier -1 dans r9
56      ; explication : t est l'adresse du 1er byte d'un tableau
57      ;               2 est l'indice de l'élément désiré (3e élément)
58      ;               8 est la taille d'un élément
59      ; en Java, on aurait simplement r9 = t[2]
60      mov     r9, [t + 2 * 8]
61
62      ; ------------------------------------------------
63      ; adresse via un registre seul : base
64      ; on parle d'adressage indirect (indirect offset)
65      ; ------------------------------------------------
66
67      ; on prépare rbx pour l'instruction suivante
68      ; rbx contient l'adresse de la variable i
69      ; rbx pointe sur la variable i
70      mov     rbx, i
71
72      ; ici adresse via registre (rbx)
73      ; rbx joue le rôle de base dans cet exemple
74      ; ici on récupère dans r12 les 8 bytes qui commencent là
75      ; où pointe rbx, ce qui aboutit à copier -234 dans r12
76      mov     r12, [rbx]
77
78      ; rem. : ici rbx pointe sur une variable globale (i)
79      ;        dont on peut utiliser l'étiquette directement
```

```
80      ;         sans passer par un registre pour accéder à son
81      ;         contenu
82      ;         lorsqu'on travaille avec des variables locales,
83      ;         sans label associé, on n'a pas d'autre choix que
84      ;         d'utiliser l'adressage indirect avec un registre
85      ;         de base
86
87      ; ------------------------------------------------------
88      ; adresse via immédiat + registre : déplacement + indice
89      ; ------------------------------------------------------
90
91      mov     rax, 0
92      ; récupération dans r13b du caractère d'indice rax (ici 0),
93      ; donc le 1er élément, de la chaîne de caractères s, ce qui
94      ; aboutit à copier `h` dans r13b
95      mov     r13b, [s + rax]
96
97      mov     rax, 4
98      ; récupération dans r14b du caractère d'indice rax (ici 4),
99      ; donc le 5e élément, de la chaîne de caractères s, ce qui
100     ; aboutit à copier `o` dans r14b
101     mov     r14b, [s + rax]
102
103     ; ------------------------------------------------------
104     ; adresse via immédiat + registre × immédiat :
105     ;               déplacement + indice × facteur d'échelle
106     ; ------------------------------------------------------
107
108     mov     rax, 0
109     ; récupération dans rsi de l'élément d'indice rax (ici 0),
110     ; donc le 1er élément, du tableau t, ce qui aboutit à copier -1
111     ; dans rsi
112     mov     rsi, [t + rax * 8]
113
114     mov     rax, 1
115     ; récupération dans rdi de l'élément d'indice rax (ici 1),
116     ; donc du 2e élément, du tableau t, ce qui aboutit à copier 1
117     ; dans rdi
118     mov     rdi, [t + rax * 8]
119
120     ; ------------------------------------------------
121     ; adresse via registre + registre (× immédiat) :
122     ;               base + indice (× facteur d'échelle)
123     ; ------------------------------------------------
```

```
124
125     ; rbx pointe sur le 1er élément de la chaîne de caractères s
126     mov     rbx, s
127
128     mov     rax, 1
129     ; récupération dans r14b de l'élément d'indice rax (ici 1),
130     ; donc le 2e élément, de la chaîne de caractères pointée
131     ; par rbx, donc de la chaîne s, ce qui aboutit à copier `e`
132     ; dans r14b
133     mov     r14b, [rbx + rax]
134
135     mov     rax, 2
136     ; récupération dans r14b de l'élément d'indice rax (ici 2),
137     ; donc le 3e élément, de la chaîne de caractères pointée
138     ; par rbx, donc de la chaîne s, ce qui aboutit à copier `l`
139     ; dans r15b
140     mov     r15b, [rbx + rax]
141
142     ; rbx pointe sur le 1er élément du tableau t
143     mov     rbx, t
144
145     mov     rax, 0
146     ; récupération dans rcx de l'élément d'indice rax (ici 0),
147     ; donc le 1er élément, du tableau pointé par rbx, donc
148     ; du tableau t, ce qui aboutit à copier -1 dans rcx
149     mov     rcx, [rbx + rax * 8]
150
151     mov     rax, 1
152     ; récupération dans rdx de l'élément d'indice rax (ici 1),
153     ; donc du 2e élément, du tableau pointé par rbx, donc
154     ; du tableau t, ce qui aboutit à copier 1 dans rdx
155     mov     rdx, [rbx + rax * 8]
156
157     ; rem. : ici rbx pointe sur 2 variables globales (s et t)
158     ;         dont on peut utiliser les étiquettes directement
159     ;         sans passer par un registre pour accéder au
160     ;         contenu
161     ;         lorsqu'on travaille avec des variables locales,
162     ;         sans label associé, on n'a pas d'autre choix que
163     ;         d'utiliser un registre de base pour un adressage
164     ;         indirect
165
166     ; rem. générale : ci-dessus accès mémoire en lecture (source)
167     ;                  pour accès en écriture (destination), la
```

```
168     ;                   syntaxe d'accès (offsets) est identique
169
170     ; --------------------------------
171     ; divers accès mémoire en écriture
172     ; --------------------------------
173
174     ; écriture du caractère `a` dans le 1er byte de msg
175     ; [déplacement]
176     mov     byte [msg], `a`
177
178     ; écriture du caractère `b` dans le 2e byte de msg
179     ; [déplacement] : pas d'indice ici car nasm réalise le calcul
180     ;                  (somme de 2 immédiats) lors de l'assemblage
181     mov     byte [msg + 1], `b`
182
183     ; écriture du caractère `c` dans le 3e byte (indice 2) de msg
184     ; [déplacement + indice] : ici facteur d'échelle vaut 1
185     mov     r8, 2                   ; r8 : indice
186     mov     byte [msg + r8], `c`
187
188     ; écriture du caractère `d` dans le 4e byte (indice 3) de msg
189     ; [base + indice] : ici facteur d'échelle vaut 1
190     mov     r9, msg                 ; r9 : base, pointe sur msg
191     mov     r8, 3                   ; r8 : indice
192     mov     byte [r9 + r8], `d`
```

On ne donne ici pas d'exemple où les quatre parties d'un *offset* sont utilisées. Ce cas, n'est pas rencontré dans les laboratoires microprocesseur. Il se présente, par exemple, lorsqu'on désire accéder à un élément d'un tableau qui est un champ d'une variable globale d'un type structuré [20]. On fait alors correspondre :

- la base avec la position relative du (premier *byte* du premier élément du) tableau au sein du type structuré ;
- l'indice avec l'indice de l'élément du tableau ;
- le facteur de taille avec la taille d'un élément du tableau ;
- le déplacement avec le *label* associé à la variable globale.

3 Instructions `inc` et `dec`

Description Les instructions `inc` [21] et `dec` [22], servent, respectivement, à incrémenter et décrémenter un registre ou une variable. Leur fonctionnement est résumé dans la

20. https://en.wikipedia.org/wiki/Record_(computer_science) (consulté le 21 mars 2020).
21. https://www.felixcloutier.com/x86/inc (consulté le 21 mars 2020).
22. https://www.felixcloutier.com/x86/dec (consulté le 21 mars 2020).

Instruction	Effet	Contraintes	*Flags* affectés
`inc X`	`X` ← `X` + 1	`X` = registre ou variable de 8, 16, 32 ou 64 bits	`CF` n'est pas modifié `SF` ← bit de rang le plus élevé du résultat `ZF` ← 1 si résultat nul, 0 sinon
`dec X`	`X` ← `X` − 1		

TABLE 1 – Instructions `inc` et `dec`.

TABLE 1.

Exemple Voici un exemple d'utilisation des instructions `inc` et `dec` :

```
1  section .data
2      i8      DQ      29
3
4  section .text
5
6      ; copie du contenu de la variable i8 dans rdi
7      ; rdi prend la valeur 29
8      mov     rdi, [i8]
9
10     ; incrémentation de rdi : rdi passe à 30
11     inc     rdi
12
13     ; décrémentation de la variable i8 : elle passe à 28
14     ; notez l'obligation d'utiliser un spécificateur de taille
15     dec     qword [i8]
```

4 Boucles

4.1 tant que

La TABLE 2(a) illustre le schéma général de comment programmer une boucle *tant que*[23] (*while*) en assembleur.

La TABLE 2(b) illustre l'implémentation d'un *tant que* avec un test de non nullité.

Au fait, si `rax` vaut initialement 1, combien de fois la boucle du code de la TABLE 2(b) est-elle exécutée ?

23. https://fr.wikipedia.org/wiki/Structure_de_contr%C3%B4le#Boucle_%C2%AB_tant_que_%C2%BB_%C3%A0_pr%C3%A9condition (consulté le 21 mars 2020).

code précédant	code précédant
tant que condition	`tant_que:` mettre à jour les *flags*
	sauter vers `fin_tq` si condition *fausse*
instructions diverses	instructions diverses
	`jmp tant_que`
fin tant que	`fin_tq:`
suite du code	suite du code

(a) Schéma de programmation du *tant que*.

tant que `rax` $\neq 0$	`tant_que:` `cmp rax, 0`
	`jz fin_tq`
`rax` $\leftarrow$ `rax` $+ 1$	`inc rax`
	`jmp tant_que`
fin tant que	`fin_tq:`
`rcx` $\leftarrow 12$	`mov rcx, 12`

(b) Exemple de *tant que* avec test de non nullité.

TABLE 2 – tant que / *while*.

4.2 pour

Nous allons voir dans la suite deux versions de la *boucle pour*[24] (*for*) : le *pour ascendant* par pas de 1 et le *pour descendant* par pas de 1[25].

pour ascendant La TABLE 3(a) illustre le schéma général d'un *pour* ascendant[26] en assembleur. La TABLE 3(b) en donne un exemple.

pour descendant La TABLE 4(a) illustre le schéma général d'un *pour* descendant en langage d'assemblage. La TABLE 4(b) en donne un exemple.

4.3 répéter tant que

La TABLE 5(a) illustre le schéma général d'un *répéter tant que*[27] (*do... while*) en langage d'assemblage.

On remarque que contrairement au *tant que* et au *pour* qui nécessitent chacun deux instructions de branchements, un conditionnel et un inconditionnel, la boucle *répéter tant que* ne requiert qu'un seul saut, conditionnel en fin de boucle.

24. https://en.wikipedia.org/wiki/For_loop (consulté le 21 mars 2020).
25. Ou *pour ascendant* par pas de -1.
26. https://fr.wikipedia.org/wiki/Structure_de_contr%C3%B4le#Compteurs (consulté le 21 mars 2020).
27. https://troumad.developpez.com/C/algorigrammes/#L5.2 (consulté le 21 mars 2020).

code précédant	code précédant
	initialiser le compteur à 0 (non pas 1)
pour i de 1 à n faire	`pour:` mettre à jour les *flags*
	sauter vers `fin_pour` si compteur = n
instructions diverses	instructions diverses
	`inc` compteur
fin pour	sauter vers `pour`
	`fin_pour:`
suite du code	suite du code

(a) Schéma de programmation du *pour* ascendant.

	`section .data`
	`output DB 'X'`
	`; ...`
	`section .text`
	`; ...`
	`mov rdi, 1 ; stdout`
	`mov rsi, output ; buffer`
	`mov rdx, 1 ; nb bytes`
	`mov r8b, 0`
pour i de 1 à 10 faire	`pour: cmp r8b, 10 ; r8b non altéré`
	`jz fin_pour`
afficher 'X' à l'écran	`mov rax, 1 ; write`
	`syscall`
	`inc r8b`
fin pour	`jmp pour`
	`fin_pour:`
`rax` ← `rax` ET `rbx`	`and rax, rbx`

(b) Exemple du *pour* ascendant.

TABLE 3 – pour (*for*) ascendant.

code précédant	code précédant
	initialiser le compteur à n
pour i de n à 1 par -1 faire	`pour:` mettre à jour les *flags*
	sauter vers `fin_pour` si compteur = 0
instructions diverses	instructions diverses
	`dec` compteur
fin pour	sauter vers `pour`
	`fin_pour:`
suite du code	suite du code

(a) Schéma de programmation du *pour* descendant.

	`section .data`
	`output DB 'MICL\n'`
	`; ...`
	`section .text`
	`; ...`
	`mov rdi, 1 ; stdout`
	`mov rsi, output ; buffer`
	`mov rdx, 5 ; nb bytes`
	`mov r8b, 10`
pour i de 10 à 1 par -1 faire	`pour: cmp r8b, 0 ; r8b non altéré`
	`jz fin_pour`
afficher 'MICL\n' à l'écran	`mov rax, 1 ; write`
	`syscall`
	`dec r8b`
fin pour	`jmp pour`
	`fin_pour:`
rax ← −1	`mov rax, -1`

(b) Exemple du *pour* descendant.

TABLE 4 – pour (*for*) descendant.

code précédant
répéter
  instructions diverses
tant que condition
suite du code
	code précédant
	`repeter:`
	  instructions diverses
	mettre à jour les *flags*
	sauter vers `repeter` si condition vraie
	suite du code

(a) Schéma de programmation du *répéter tant que*.

tab [ ] = { -8, -2, 7, 12, -6 }
 	
`rax` ← 0
`rcx` ← 0
répéter
  si tab[`rcx`] < 0
  `rax` ← `rax` + 1
  finsi
  `rcx` ← `rcx` + 1
tant que `rcx` ≠ 5
	`section .rodata`
	`tab DD -8, -2, 7, 12, -6`
	`section .text`
	`; ...`
	`mov rax, 0`
	`mov rcx, 0`
	`repeter:`
	  `bt dword [tab + rcx * 4], 31`
	  `jnc positif`
	  `inc rax`
	`positif:`
	  `inc rcx`
	`cmp rcx, 5`
	`jnz repeter`

(b) Exemple de programmation du *répéter tant que*.

TABLE 5 – répéter tant que / *do... while*

La Table 5(b) montre un exemple d'utilisation et d'implémentation du *répéter tant que*. Pouvez-vous expliquer ce que fait cet exemple, sans utiliser de termes techniques propres au langage d'assemblage, mais uniquement avec le vocabulaire d'algorithmique ?

Code source Voici le code source complet de l'exemple de la table Table 5(b) :

```
; 09_repeter_exemple_complet.asm

global _start

section .rodata
    tab     DD      -8, -2, 7, 12, -6    ; DD : double word : 4 bytes

; contenu de la section .rodata (byte par byte, hexadécimal) :
; ---> petites adresses --->
;         tab
;  .... | F8 | FF | FF | FF | FE | FF | FF | FF | 07 | 00 | .
;
;       . 00 | 00 | 0C | 00 | 00 | 00 | FA | FF | FF | FF | ...
;
;                                          ---> grande adresses --->

section .text
_start:

    mov     rax, 0  ; compteur
    mov     rcx, 0  ; indice

 repeter:
    bt      dword [tab + rcx * 4], 31   ; test du bit de signe
    jnc     positif
    inc     rax     ; ici CF == 1 : strictement < 0
  positif:
    inc     rcx
    cmp     rcx, 5  ; car indices : 0 → 4 car 5 éléments
    jnz     repeter

    ; ici : rax == # éléments de tab < 0

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

5 Débogage et tableau

Dans le TD04, on a vu que `KDbg` considère par défaut que les variables s'étendent sur 4 *bytes*. Pour voir le contenu d'une variable de taille différente dans la vue *Expressions surveillées* (*Watched Expressions*), il faut utiliser une syntaxe où on utilise les opérateurs de transtypage (*cast*) du langage C.

Pour inspecter le contenu d'un élément d'un tableau il faut faire pareil. Nous en donnons des exemples, mais sans explications supplémentaires. Elles vous seront données en Langage C / C++ [28].

Soient :

— `t1` une étiquette identifiant le premier élément d'un tableau de *bytes* ;
— `t2` une étiquette identifiant le premier élément d'un tableau de *words* ;
— `t4` une étiquette identifiant le premier élément d'un tableau de *double words* ;
— `t8` une étiquette identifiant le premier élément d'un tableau de *quad words* ;

pour voir le 5e élément de ces tableaux, c'est-à-dire celui d'indice 4, il faut encoder :

— `*(((char *) &t1) + 4)` ou `((char *) &t1)[4]` ;
— `*(((short *) &t2) + 4)` ou `((short *) &t2)[4]` ;
— `*(((int *) &t4) + 4)` ou `((int *) &t4)[4]` ;
— `*(((long long *) &t8) + 4)` ou `((long long *) &t8)[4]`.

dans la vue *Expressions surveillées* de `KDbg`.

Pour changer le format de l'affichage des valeurs, il faut utiliser les options de formatage en sortie [29] de `gdb`.

Une autre possibilité pour voir le contenu d'une variable ou d'un tableau en mémoire est d'ouvrir la vue *Mémoire* de `KDbg` et de renseigner dans sa zone d'édition l'étiquette de la variable ou du tableau précédée par une esperluette (&). Avec le tableau `t2`, par exemple, il faut donner l'expression `&t2`. Le contenu de la mémoire à partir de l'adresse qui correspond à cette étiquette est alors affiché. Via un clic droit dans la zone d'affichage de la vue *Mémoire*, on peut demander que cet affichage soit fait *byte* par *byte*.

6 Exercices

Pour réaliser les exercices qui suivent, vous ne pouvez utiliser que les instructions étudiées au long de ce TD et des précédents.

Ex. 1 Écrivez un programme qui :
— déclare un tableau de 10 entiers de 2 *bytes* chacun initialisés à 0 ;
— assigne la valeur 3 à son 3e élément et 8 au 8e.

28. https://heb-esi.github.io/he2besi-web/online/cours/ac1920_dev3_cpp.html (consulté le 21 mars 2020).
29. http://sourceware.org/gdb/current/onlinedocs/gdb/Output-Formats.html#Output-Formats (consulté le 21 mars 2020).

Ex. 2 Écrivez un programme qui :
— déclare un tableau de 100 entiers de 8 *bytes* chacun initialisés à 0 ;
— assigne à son élément d'indice i la valeur i.

Par exemple, le premier élément du tableau contient la valeur 0 et son dernier la valeur 99, car l'indice de son dernier élément vaut 99.

Aide : pour le parcours du tableau, n'hésitez pas à vous inspirer du code fourni en page 20 pour illustrer la boucle *do... while*.

Ex. 3 Écrivez un programme qui :
— déclare un tableau de 100 entiers de 8 *bytes* chacun initialisés à 0 ;
— assigne à son élément d'indice i la valeur $99 - i$.

Par exemple, le premier élément du tableau contient la valeur 99 et son dernier la valeur 0, car l'indice de son dernier élément vaut 99 et $99 - 99 = 0$.

Ex. 4 Écrivez un programme qui :
— déclare un tableau de 100 entiers de 8 *bytes* chacun initialisés à 0 ;
— assigne à son élément d'indice i la valeur $2i$.

Par exemple, le premier élément du tableau contient la valeur 0 et son dernier la valeur 198, car l'indice de son dernier élément vaut 99 et $2 \times 99 = 198$.

Aide : comme on n'a pas encore vu d'instruction pour calculer la somme ou le produit de deux entiers quelconques, il faut ici se débrouiller en utilisant plusieurs fois d'affilée les instructions `inc` ou `dec`.

Ex. 5 Écrivez un programme qui :
— déclare un tableau de 10 entiers constants de 8 *bytes* chacun initialisés aux valeurs de votre choix ;
— détermine combien parmi les éléments de ce tableau sont pairs et stocke ce nombre dans `r15`.

USASCII code chart

b_7 b_6 b_5 → / Bits ↓	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4 b_3 b_2 b_1 Row ↓ / Column →	0	1	2	3	4	5	6	7
0 0 0 0	0	NUL	DLE	SP	0	@	P	`	p
0 0 0 1	1	SOH	DC1	!	1	A	Q	a	q
0 0 1 0	2	STX	DC2	"	2	B	R	b	r
0 0 1 1	3	ETX	DC3	#	3	C	S	c	s
0 1 0 0	4	EOT	DC4	$	4	D	T	d	t
0 1 0 1	5	ENQ	NAK	%	5	E	U	e	u
0 1 1 0	6	ACK	SYN	&	6	F	V	f	v
0 1 1 1	7	BEL	ETB	'	7	G	W	g	w
1 0 0 0	8	BS	CAN	(	8	H	X	h	x
1 0 0 1	9	HT	EM	)	9	I	Y	i	y
1 0 1 0	10	LF	SUB	*	:	J	Z	j	z
1 0 1 1	11	VT	ESC	+	;	K	[	k	{
1 1 0 0	12	FF	FS	,	<	L	\	l	|
1 1 0 1	13	CR	GS	-	=	M	]	m	}
1 1 1 0	14	SO	RS	.	>	N	^	n	~
1 1 1 1	15	SI	US	/	?	O	_	o	DEL

FIG. 1 – Table `ASCII` (Illustration Wikipedia [a]).

a. https://commons.wikimedia.org/wiki/File:USASCII_code_chart.png (consulté le 21 mars 2020).

Ex. 6 Écrivez un programme qui :
- déclare un tableau de 10 entiers constants de 8 *bytes* chacun initialisés aux valeurs de votre choix ;
- stocke dans `r8` la valeur minimale parmi les éléments du tableau ;
- stocke dans `r10` la valeur maximale parmi les éléments du tableau.

Ex. 7 Écrivez un programme qui affiche à l'écran tous les chiffres de 0 à 9, chacun sur une ligne différente.

Les seules variables que vous pouvez utiliser sont déclarées dans la `section .bss`.

N'hésitez pas à consulter la table ASCII fournie à la FIG. 1.

Ex. 8 Écrivez un programme qui affiche à l'écran tous les chiffres de 0 à 9, chacun sur une ligne différente.

La seule variable que vous pouvez utiliser est la suivante :

```
1 section .rodata
2     digits      DB      `0123456789\n`
```

Ex. 9 Écrivez un programme qui se comporte comme la commande `cat`[30] sans argument.

Pour rappel, le descripteur de fichier correspondant à l'entrée standard, qui, par défaut, est associée au clavier, vaut 0. Celui de la sortie standard, associée à l'écran, vaut 1.

Aide : lisez les *bytes* de l'entrée standard *un à un* dans *une seule* variable de taille *un byte*.

Profitez de la mise en tampon (*buffer*) des données issues du clavier ou écrites à l'écran. Les tampons sont vidés :

— lorsqu'il est plein ;
— lorsque le caractère de contrôle `` `\n` `` y est injecté ;
— lorsqu'on demande sa vidange à l'aide de l'appel système `fflush`[31].

Donc, lorsque votre programme s'exécute et attend des données de l'entrée standard, il ne les reçoit que lorsqu'une de ces trois circonstances intervient. Il peut alors, en boucle, lire *un byte* sur l'entrée standard et l'écrire sur la sortie standard. Cette boucle s'arrête lorsqu'il n'y a plus de donnée disponible sur l'entrée standard.

Au clavier, il faut enfoncer :

— la touche Enter pour injecter le caractère `` `\n` `` dans le flux en entrée ;
— les touches Ctrl-D pour forcer le *flush* du tampon.

Pour indiquer au programme la fin de flux en entrée, il faut forcer une lecture alors qu'*aucun* caractère n'est disponible[32] dans ce flux. Il faut donc enfoncer les touches Ctrl-D alors qu'aucun caractère n'est dans le flux, c'est-à-dire juste après une vidange précédente. Dans le cas de la commande `cat` avec entrée sur `stdin`, cela revient à enfoncer Ctrl-D en tout début d'une nouvelle ligne.

Ex. 10 Écrivez un programme qui affiche à l'écran le contenu d'un fichier. Le nom du fichier est contenu dans une variable globale. Si l'ouverture du fichier échoue, affichez à l'écran : `impossible d'ouvrir le fichier`.

30. https://man.cx/cat (consulté le 21 mars 2020).
31. http://man7.org/linux/man-pages/man3/fflush.3.html (consulté le 21 mars 2020).
32. Une des manières de détecter la fin d'un fichier (*end-of-file*, *EOF*) de n *bytes*, alors qu'on ignore n, est de lire son contenu *byte* par *byte* depuis son début. Il faut alors $n+1$ lectures, soit une de plus que la taille du fichier. Les n premières lectures se déroulent sans problème tandis que la dernière échoue : elle ne lit *aucun byte*.

Aide : recyclez votre solution de l'Ex. 9. À la place de lire au clavier, lisez dans un fichier.

Ex. 11 Écrivez un programme qui concatène deux fichiers, c'est-à-dire qui copie le contenu du deuxième fichier à la fin du premier. Les noms des deux fichiers sont contenus dans des variables globales. On suppose que les deux fichiers existent déjà.

Aide : recyclez votre solution de l'Ex. 10. À la place d'écrire sur la sortie standard, écrivez dans un fichier.

Notions à retenir

Définition d'un tableau global et accès à son contenu, modes d'adressage des opérandes, instructions `inc` et `dec`, implémentations des boucles *tant que*, *pour* et *répéter tant que*.

Références

[1] *Intel© 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*, octobre 2017. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.

[2] Igor Zhirkov. *Low-Level Programming*. Apress, 2017. https://www.apress.com/gp/book/9781484224021.