HE2B ESI

# MICL : TD07 : Pile et variables locales

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



Année académique 2020 – 2021

Dans ce TD, la pile associée à chaque processus s'exécutant sur un processeur de la famille x86 est d'abord introduite ainsi que les registres et instructions associées. Ensuite, les instructions `add` et `sub` sont brièvement présentées. Enfin, la création et l'utilisation des variables locales, sur la pile, sont expliquées.

## 1 Sections et occupation mémoire

Soit le source :

```
1  ; 00_section.asm
2
3  global _start
4
5  section .data
6      bd0     DB      1
7      bd1     DB      42
8      bd2     DB      -1
9      bd3     DB      23
10
11 section .rodata
12     drd0    DD      314
13     drd1    DD      4
```

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

```
14      drd2    DD      12
15
16  section .bss
17      wb0     RESW    1
18      wb1     RESW    1
19      wb2     RESW    1
20
21  section .text
22  _start:
23
24      mov     rsi, _start     ; 0x4000b0
25
26      mov     rax, drd0       ; 0x40013c
27      mov     rbx, drd1       ; 0x400140
28      mov     rcx, drd2       ; 0x400144
29
30      mov     r8, bd0         ; 0x600148
31      mov     r9, bd1         ; 0x600149
32      mov     r10, bd2        ; 0x60014a
33      mov     r11, bd3        ; 0x60014b
34
35      ; rem. : si bd3 pas déclaré,
36      ;         wb0 quand même en adresse multiple de 4 : alignement (?)
37      ;         https://en.wikipedia.org/wiki/Data_structure_alignment
38
39      mov     r12, wb0        ; 0x60014c
40      mov     r13, wb1        ; 0x60014e
41      mov     r14, wb2        ; 0x600150
42
43      ; rsp : 0x7fffffffe730 : cette valeur peut varier
44
45   .infinity:
46      jmp     .infinity
47
48      mov     rax, 60
49      mov     rdi, 0
50      syscall
```

On l'assemble :

```
$ nasm -f elf64 -F dwarf 00_section.asm
```

On utilise l'éditeur de liens pour produire un exécutable :

```
$ ld -o 00_section 00_section.o
```

Son exécution dans un débogueur :

```
$ kdbg 00_section
```

permet de vérifier que les informations en commentaires dans la `section .text` correspondent plus ou moins aux adresses du code et des variables[1]. Ce qui importe ici ce n'est pas la valeur exacte de chaque adresse, mais plutôt les *plages d'adresses* qu'on peut observer. Les valeurs d'adresses peuvent d'ailleurs différer d'une machine à l'autre.

Par ailleurs, après avoir démarré `00_section` dans le débogueur, il est possible d'obtenir des informations sur l'*occupation mémoire* du programme en cours de débogage. Pour ce faire, dans un nouveau *shell*, on récupère le numéro d'identification du processus `00_section` :

```
$ ps -e | grep 00_section
19497 pts/2    00:00:00 00_section
```

On exécute ensuite :

```
$ cat /proc/19497/maps
00400000-00401000 r-xp 00000000 08:03 1257745           /home/.../00_section
00600000-00601000 rwxp 00000000 08:03 1257745           /home/.../00_section
7ffff7ffe000-7ffff7fff000 r-xp 00000000 00:00 0         [vdso]
7ffffffde000-7fffffffe000 rwxp 00000000 00:00 0         [stack]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

où on a raccourci les lignes `/home/.../00_section`. Pour obtenir davantage d'information sur le fichier `/proc/`*`pid`*`/maps`, consultez la page de manuel de `procfs`[2].

Une première région s'étend sur la plage d'adresses[3] `0x00400000` à `0x00401000`. Si on les compare avec les adresses obtenues dans le débogueur, cela correspond :

— à la `section .text` : l'étiquette `_start` est collée à l'adresse `0x004000b0`, la suite du code suit ;
— à la `section .rodata` : l'étiquette `drd0` est collée à l'adresse `0x40013c`, les variables immuables restantes suivent.

Une deuxième région s'étend sur la plage d'adresses `0x00600000` à `0x00601000`. Si on les compare avec les adresses obtenues dans le débogueur, cela correspond :

— à la `section .data` : l'étiquette `bd0` est collée à l'adresse `0x600148`, les variables restantes de cette section suivent ;
— à la `section .bss` : l'étiquette `wb0` est collée à l'adresse `0x60014c`, juste après les variables de la `section .data` tandis que les autres variables de la `section .bss` suivent.

La région qui nous intéresse dans ce TD est celle notée `[stack]`. Il s'agit de la *pile* du processus.

1. Les valeurs qui apparaissent dans le code ont été obtenues sur la machine `linux1` en mars 2019.
2. http://man7.org/linux/man-pages/man5/proc.5.html (consulté le 6 avril 2020).
3. Rappelons-le, cela peut varier d'une machine à l'autre.

# 2 Pile

La *pile*[4] (*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. Les aspects de l'utilisation de la pile en rapport avec les fonctions sont étudiés lors du TD08. Les variables locales sont abordées plus tard dans ce TD, à la section 5.

## 2.1 Registres associés à la pile

Parmi les registres du processeur, `rsp` et `rbp`, bien que généraux[5], 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.

## 2.2 Instructions d'accès à la pile

La manière la plus simple d'accéder à la pile est d'utiliser les instructions `push`[6] et `pop`[7]. L'instruction `push` permet de stocker un élément dans la pile. On parle d'*empilement* et d'*empiler*. L'instruction `pop` sert à retirer une valeur de la pile. On parle de *dépilement* et de *dépiler*. Le couple `push` / `pop` forme les deux primitives d'accès à la structure de donnée *LIFO*[8] (*Last In, First Out*, « Dernier arrivé, premier sorti ») qu'est la *pile* (*stack*). La Table 1 explique leur fonctionnement.

À l'analyse de ce tableau, on remarque que ni `push`, ni `pop` n'ont d'opérande registre ou variable de 8 ou 32 bits. Notez cependant qu'il est possible de poser / enlever des données dans un registre ou en mémoire de 8 ou 32 bits sur / depuis la pile. Ceci est illustré dans le code source de la section 4.2.

4. https://fr.wikipedia.org/wiki/Pile_%28informatique%29 (consulté le 6 avril 2020).
5. https://stackoverflow.com/q/36529449 (consulté le 6 avril 2020).
6. https://www.felixcloutier.com/x86/push (consulté le 6 avril 2020).
7. https://www.felixcloutier.com/x86/pop (consulté le 6 avril 2020).
8. https://en.wikipedia.org/wiki/Stack_(abstract_data_type) (consulté le 6 avril 2020).

| Instruction | Contrainte | Effet | *Flags* affectés |
|---|---|---|---|
| `push X` | `X` est un registre ou une variable de 16 ou 64 bits | `X` est empilé et `rsp` est décrémenté de la taille en *bytes* de `X` | aucun |
| `push X` | `X` est un immédiat sur 16 bits | `X` est empilé et `rsp` est décrémenté de 2 (la taille de `X` en *bytes*) | aucun |
| `push X` | `X` est un immédiat sur 8 ou 32 bits | `X` est étendu par extension de signe sur 64 bits et empilé tandis que `rsp` est décrémenté de 8 | aucun |
| `push X` | `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 `rsp` est décrémenté de 8 | aucun |
| `pop X` | `X` est un registre ou une variable de 16 ou 64 bits | `X` reçoit le résultat du dépilement d'un nombre de *bytes* égal à sa taille et `rsp` est incrémenté d'autant | aucun |

TABLE 1 – Instructions `push` et `pop`.

D'autres instructions sont dédiées à l'accès à la pile. Il s'agit de `pusha` / `pushad`[9] et `popa` / `popad`[10]. Comme elles sont invalides en 64 bits, nous ne les abordons pas.

Notez également l'existence d'instructions permettant de copier `rflags` sur la pile[11] et de réaliser l'opération réciproque[12]. Nous ne les étudions pas ici.

On verra plus tard dans ce TD, à la section 5.2 plus particulièrement, qu'il est possible d'accéder aux éléments sur la pile en « trichant », c'est-à-dire en outrepassant la règle d'accès *LIFO* pour accéder *directement* à une donnée, quel que soit son emplacement dans la pile.

# 3 Instructions `add` et `sub`

Les instructions `add`[13] et `sub`[14] permettent de réaliser les opérations arithmétiques d'addition et de soustraction.

La TABLE 2 donne un résumé de ces instructions.

Dans la suite de ce TD, on utilise ces instructions pour empiler et dépiler des valeurs sans recourir aux instructions `push` ou `pop`, à la section 4.2, et pour créer et détruire des variables locales, à la section 5.2.

9. https://www.felixcloutier.com/x86/pusha:pushad (consulté le 6 avril 2020).
10. https://www.felixcloutier.com/x86/popa:popad (consulté le 6 avril 2020).
11. https://www.felixcloutier.com/x86/pushf:pushfd:pushfq (consulté le 6 avril 2020).
12. https://www.felixcloutier.com/x86/popf:popfd:popfq (consulté le 6 avril 2020).
13. https://www.felixcloutier.com/x86/add (consulté le 7 avril 2020).
14. https://www.felixcloutier.com/x86/sub (consulté le 7 avril 2020).

| Instruction | Effet | Contraintes | *Flags* affectés |
|---|---|---|---|
| `add X, Y` | `X` ← `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) | `SF` ← bit de rang le plus élevé du résultat<br>`ZF` ← 1 si résultat nul, 0 sinon<br>`CF` et `OF` sont modifiés d'une manière qui dépasse le cadre des MICL |
| `sub X, Y` | `X` ← `X` − `Y` | | |

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

**Remarque** Comme indiqué dans la TABLE 2, les immédiats s'étendent sur maximum 32 bits. Si on désire additionner une valeur immédiate qui s'étend sur 64 bits, il faut passer par un registre intermédiaire [15].

# 4 Exemples d'utilisation de la pile

## 4.1 Accès à la pile avec `push` et `pop`

Voici un premier code source accédant à la pile via les instructions `push` et `pop` :

```
1  ; 01_push_pop_64_fc.asm
2  ;
3  ; nasm -f elf64 -F dwarf 01_push_pop_64_fc.asm
4  ;   il y a des warnings, voir plus bas
5  ; ld -o 01_push_pop_64_fc 01_push_pop_64_fc.o
6
7  global _start
8
9  section .rodata
10     source          DB      1, 2, 3, 4, 5, 6, 7, 8
11
12 section .bss
13     destination     RESQ    1
14
15 section .text
16 _start:
17
18     ; pile vide
19     ;                       petites adresses
20     ;           |       |
21     ; rsp --->  +-------+  grandes adresses 0x7fffffffdab0
22     ; rem. : le contenu initial de rsp peut varier
```

15. https://stackoverflow.com/a/20020648 (consulté le 7 avril 2020).

```
23
24      ; push registre
25
26      mov     rax, 0x123456789ABCDEF0
27
28      ; push    al          ; ko car al fait 8 bits
29      ; error: invalid combination of opcode and operands
30
31      push    ax
32      ; la valeur 0xDEF0 est placée sur la pile sur 2 bytes
33      ;   rsp <-- rsp - 2
34      ;
35      ;           |      | petites adresses
36      ; rsp --->  | 0xF0 | 0x7fffffffdaae
37      ;           | 0xDE |
38      ;           +------+ grandes adresses 0x7fffffffdab0
39      ; rem. : le contenu initial de rsp peut varier
40
41      ; push    eax         ; ko car eax fait 32 bits
42      ; error: instruction not supported in 64-bit mode
43
44      push    rax
45      ; la valeur 0x123456789ABCDEF0 est placée sur la pile sur 8 bytes
46      ;   rsp <-- rsp - 8
47      ;
48      ;           |      | petites adresses
49      ; rsp --->  | 0xF0 | 0x7fffffffdaa6
50      ;           | 0xDE |
51      ;           | 0xBC |
52      ;           | 0x9A |
53      ;           | 0x78 |
54      ;           | 0x56 |
55      ;           | 0x34 |
56      ;           | 0x12 |
57      ;           | 0xF0 | 0x7fffffffdaae
58      ;           | 0xDE |
59      ;           +------+ grandes adresses 0x7fffffffdab0
60      ; rem. : le contenu initial de rsp peut varier
61
62      ; push mémoire
63
64      ; push    byte [source]     ; ko car source fait 8 bits
65      ; error: invalid combination of opcode and operands
66
```

```
67      push    word [source]
68      ; la valeur 0x0201 est placée sur la pile sur 2 bytes
69      ;   rsp <-- rsp - 2
70      ;
71      ;           |      | petites adresses
72      ; rsp --->  | 0x01 | 0x7fffffffdaa4
73      ;           | 0x02 |
74      ;           | 0xF0 | 0x7fffffffdaa6
75      ;           | 0xDE |
76      ;           | 0xBC |
77      ;           | 0x9A |
78      ;           | 0x78 |
79      ;           | 0x56 |
80      ;           | 0x34 |
81      ;           | 0x12 |
82      ;           | 0xF0 | 0x7fffffffdaae
83      ;           | 0xDE |
84      ;           +------+ grandes adresses 0x7fffffffdab0
85      ; rem. : le contenu initial de rsp peut varier
86
87      ; push    dword [source]      ; ko car source fait 32 bits
88      ; error: instruction not supported in 64-bit mode
89
90      push    qword [source]
91      ; la valeur 0x0807060504030201 est placée sur la pile sur 8 bytes
92      ;   rsp <-- rsp - 8
93      ;
94      ;           |      | petites adresses
95      ; rsp --->  | 0x01 | 0x7fffffffda9c
96      ;           | 0x02 |
97      ;           | 0x03 |
98      ;           | 0x04 |
99      ;           | 0x05 |
100     ;           | 0x06 |
101     ;           | 0x07 |
102     ;           | 0x08 |
103     ;           | 0x01 | 0x7fffffffdaa4
104     ;           | 0x02 |
105     ;           | 0xF0 | 0x7fffffffdaa6
106     ;           | 0xDE |
107     ;           | 0xBC |
108     ;           | 0x9A |
109     ;           | 0x78 |
110     ;           | 0x56 |
```

```
111     ;           | 0x34 |
112     ;           | 0x12 |
113     ;           | 0xF0 | 0x7fffffffdaae
114     ;           | 0xDE |
115     ;           +------+ grandes adresses 0x7fffffffdab0
116     ; rem. : le contenu initial de rsp peut varier
117
118     ; pop registre
119
120     ; pop     r8b             ; ko car r8b fait 8 bits
121     ; error: invalid combination of opcode and operands
122
123     pop     r9w
124     ; la valeur 0x0201 est placée dans r9w et rsp incrémenté de 2
125     ;   r9w <-- 0x0201
126     ;   rsp <-- rsp + 2
127     ;
128     ;           |      | petites adresses
129     ;           | 0x01 | 0x7fffffffda9c
130     ;           | 0x02 |
131     ; rsp --->  | 0x03 | 0x7fffffffda9e
132     ;           | 0x04 |
133     ;           | 0x05 |
134     ;           | 0x06 |
135     ;           | 0x07 |
136     ;           | 0x08 |
137     ;           | 0x01 | 0x7fffffffdaa4
138     ;           | 0x02 |
139     ;           | 0xF0 | 0x7fffffffdaa6
140     ;           | 0xDE |
141     ;           | 0xBC |
142     ;           | 0x9A |
143     ;           | 0x78 |
144     ;           | 0x56 |
145     ;           | 0x34 |
146     ;           | 0x12 |
147     ;           | 0xF0 | 0x7fffffffdaae
148     ;           | 0xDE |
149     ;           +------+ grandes adresses 0x7fffffffdab0
150     ; rem. : le contenu initial de rsp peut varier
151
152     ; pop     r10d            ; ko car r10d fait 32 bits
153     ; error: instruction not supported in 64-bit mode
154
```

```
155     pop     r11
156     ; la valeur 0x0201080706050403 est placée dans r11 et rsp
157     ; incrémenté de 8
158     ;   r11 <-- 0x0201080706050403
159     ;   rsp <-- rsp + 8
160     ;
161     ;           |      | petites adresses
162     ;           | 0x01 | 0x7fffffffda9c
163     ;           | 0x02 |
164     ;           | 0x03 | 0x7fffffffda9e
165     ;           | 0x04 |
166     ;           | 0x05 |
167     ;           | 0x06 |
168     ;           | 0x07 |
169     ;           | 0x08 |
170     ;           | 0x01 | 0x7fffffffdaa4
171     ;           | 0x02 |
172     ; rsp --->  | 0xF0 | 0x7fffffffdaa6
173     ;           | 0xDE |
174     ;           | 0xBC |
175     ;           | 0x9A |
176     ;           | 0x78 |
177     ;           | 0x56 |
178     ;           | 0x34 |
179     ;           | 0x12 |
180     ;           | 0xF0 | 0x7fffffffdaae
181     ;           | 0xDE |
182     ;           +------+ grandes adresses 0x7fffffffdab0
183     ; rem. : le contenu initial de rsp peut varier
184
185     ; pop mémoire
186
187     ; pop     byte [destination]      ; ko car destination fait 8 bits
188     ; error: invalid combination of opcode and operands
189
190     pop     word [destination]
191     ; la valeur 0xDEF0 est placée à l'adresse destination sur 2 bytes
192     ; et rsp incrémenté de 2
193     ;   word [destination] <-- 0xDEF0
194     ;   rsp <-- rsp + 2
195     ;
196     ;           |      | petites adresses
197     ;           | 0x01 | 0x7fffffffda9c
198     ;           | 0x02 |
```

```
199     ;            | 0x03 | 0x7fffffffda9e
200     ;            | 0x04 |
201     ;            | 0x05 |
202     ;            | 0x06 |
203     ;            | 0x07 |
204     ;            | 0x08 |
205     ;            | 0x01 | 0x7fffffffdaa4
206     ;            | 0x02 |
207     ;            | 0xF0 | 0x7fffffffdaa6
208     ;            | 0xDE |
209     ; rsp --->   | 0xBC | 0x7fffffffdaa8
210     ;            | 0x9A |
211     ;            | 0x78 |
212     ;            | 0x56 |
213     ;            | 0x34 |
214     ;            | 0x12 |
215     ;            | 0xF0 | 0x7fffffffdaae
216     ;            | 0xDE |
217     ;            +------+ grandes adresses 0x7fffffffdab0
218     ; rem. : le contenu initial de rsp peut varier
219
220     ; pop     dword [destination]   ; ko car destination fait 32 bits
221     ; error: instruction not supported in 64-bit mode
222
223     pop     qword [destination]
224     ; la valeur 0xDEF0123456789ABC est placée à l'adresse destination
225     ; sur 8 bytes et rsp incrémenté de 8
226     ;   qword [destination] <-- 0xDEF0123456789ABC
227     ;   rsp <-- rsp + 8
228     ;
229     ;            |      | petites adresses
230     ;            | 0x01 | 0x7fffffffda9c
231     ;            | 0x02 |
232     ;            | 0x03 | 0x7fffffffda9e
233     ;            | 0x04 |
234     ;            | 0x05 |
235     ;            | 0x06 |
236     ;            | 0x07 |
237     ;            | 0x08 |
238     ;            | 0x01 | 0x7fffffffdaa4
239     ;            | 0x02 |
240     ;            | 0xF0 | 0x7fffffffdaa6
241     ;            | 0xDE |
242     ;            | 0xBC | 0x7fffffffdaa8
```

```
243     ;            | 0x9A |
244     ;            | 0x78 |
245     ;            | 0x56 |
246     ;            | 0x34 |
247     ;            | 0x12 |
248     ;            | 0xF0 | 0x7fffffffdaae
249     ;            | 0xDE |
250     ; rsp --->   +------+ grandes adresses 0x7fffffffdab0
251     ; rem. : le contenu initial de rsp peut varier
252
253     ; push immédiat
254
255     push    byte 0xF0
256     ; warning: signed byte value exceeds bounds [-w+number-overflow]
257     ; la valeur 0xF0 est étendue par extension de signe sur 64 bits
258     ; et placée sur la pile, c'est-à-dire que :
259     ; la valeur 0xFFFFFFFFFFFFFFF0 est placée sur la pile sur 8 bytes
260     ;   rsp <-- rsp - 8
261     ;
262     ;            |      | petites adresses
263     ;            | 0x01 | 0x7fffffffda9c
264     ;            | 0x02 |
265     ;            | 0x03 | 0x7fffffffda9e
266     ;            | 0x04 |
267     ;            | 0x05 |
268     ;            | 0x06 |
269     ;            | 0x07 |
270     ;            | 0x08 |
271     ;            | 0x01 | 0x7fffffffdaa4
272     ;            | 0x02 |
273     ;            | 0xF0 | 0x7fffffffdaa6
274     ;            | 0xDE |
275     ; rsp --->   | 0xF0 | 0x7fffffffdaa8
276     ;            | 0xFF |
277     ;            | 0xFF |
278     ;            | 0xFF |
279     ;            | 0xFF |
280     ;            | 0xFF |
281     ;            | 0xFF | 0x7fffffffdaae
282     ;            | 0xFF |
283     ;            +------+ grandes adresses 0x7fffffffdab0
284     ; rem. : le contenu initial de rsp peut varier
285
286     push    word 0xDEF0
```

```
287     ; la valeur 0xDEF0 est placée sur la pile sur 2 bytes
288     ;   rsp <-- rsp - 2
289     ;
290     ;            |      | petites adresses
291     ;            | 0x01 | 0x7fffffffda9c
292     ;            | 0x02 |
293     ;            | 0x03 | 0x7fffffffda9e
294     ;            | 0x04 |
295     ;            | 0x05 |
296     ;            | 0x06 |
297     ;            | 0x07 |
298     ;            | 0x08 |
299     ;            | 0x01 | 0x7fffffffdaa4
300     ;            | 0x02 |
301     ; rsp --->   | 0xF0 | 0x7fffffffdaa6
302     ;            | 0xDE |
303     ;            | 0xF0 | 0x7fffffffdaa8
304     ;            | 0xFF |
305     ;            | 0xFF |
306     ;            | 0xFF |
307     ;            | 0xFF |
308     ;            | 0xFF |
309     ;            | 0xFF | 0x7fffffffdaae
310     ;            | 0xFF |
311     ;            +------+ grandes adresses 0x7fffffffdab0
312     ; rem. : le contenu initial de rsp peut varier
313
314     push    dword 0x9ABCDEF0
315     ; warning: signed dword immediate exceeds bounds [-w+number-overflow]
316     ; warning: dword data exceeds bounds [-w+number-overflow]
317     ; la valeur 0x9ABCDEF0 est étendue par extension de signe sur
318     ; 64 bits et placée sur la pile, c'est-à-dire que :
319     ; la valeur 0xFFFFFFFF9ABCDEF0 est placée sur la pile sur 8 bytes
320     ;   rsp <-- rsp - 8
321     ;
322     ;            |      | petites adresses
323     ;            | 0x01 | 0x7fffffffda9c
324     ;            | 0x02 |
325     ; rsp --->   | 0xF0 | 0x7fffffffda9e
326     ;            | 0xDE |
327     ;            | 0xBC |
328     ;            | 0x9A |
329     ;            | 0xFF |
330     ;            | 0xFF |
```

```
331     ;             | 0xFF | 0x7fffffffdaa4
332     ;             | 0xFF |
333     ;             | 0xF0 | 0x7fffffffdaa6
334     ;             | 0xDE |
335     ;             | 0xF0 | 0x7fffffffdaa8
336     ;             | 0xFF |
337     ;             | 0xFF |
338     ;             | 0xFF |
339     ;             | 0xFF |
340     ;             | 0xFF |
341     ;             | 0xFF | 0x7fffffffdaae
342     ;             | 0xFF |
343     ;             +------+ grandes adresses 0x7fffffffdab0
344     ; rem. : le contenu initial de rsp peut varier
345
346     push    qword 0x123456789ABCDEF0
347     ; warning: signed dword immediate exceeds bounds [-w+number-overflow]
348     ; warning: dword data exceeds bounds [-w+number-overflow]
349     ; les 32 bits de poids faible de la valeur 0x123456789ABCDEF0,
350     ; c'est-à-dire la valeur 0x9ABCDEF0 est étendue par extension
351     ; de signe sur 64 bits et placée sur la pile, c'est-à-dire que :
352     ; la valeur 0xFFFFFFFF9ABCDEF0 est placée sur la pile sur 8 bytes
353     ;   rsp <-- rsp - 8
354     ;
355     ;             |      | petites adresses
356     ; rsp --->    | 0xF0 | 0x7fffffffda96
357     ;             | 0xDE |
358     ;             | 0xBC |
359     ;             | 0x9A |
360     ;             | 0xFF |
361     ;             | 0xFF |
362     ;             | 0xFF | 0x7fffffffda9c
363     ;             | 0xFF |
364     ;             | 0xF0 | 0x7fffffffda9e
365     ;             | 0xDE |
366     ;             | 0xBC |
367     ;             | 0x9A |
368     ;             | 0xFF |
369     ;             | 0xFF |
370     ;             | 0xFF | 0x7fffffffdaa4
371     ;             | 0xFF |
372     ;             | 0xF0 | 0x7fffffffdaa6
373     ;             | 0xDE |
374     ;             | 0xF0 | 0x7fffffffdaa8
```

```
375     ;            | 0xFF |
376     ;            | 0xFF |
377     ;            | 0xFF |
378     ;            | 0xFF |
379     ;            | 0xFF |
380     ;            | 0xFF | 0x7fffffffdaae
381     ;            | 0xFF |
382     ;            +------+ grandes adresses 0x7fffffffdab0
383     ; rem. : le contenu initial de rsp peut varier
384
385     push   0
386     ; les 32 bits de poids faible de la valeur 0x00, c'est-à-dire
387     ; la valeur 0x00 est étendue par extension de signe sur 64 bits
388     ; et placée sur la pile, c'est-à-dire que :
389     ; la valeur 0x00 est placée sur la pile sur 8 bytes
390     ;   rsp <-- rsp - 8
391     ;
392     ;            |      | petites adresses
393     ; rsp --->   | 0x00 | 0x7fffffffda8e
394     ;            | 0x00 |
395     ;            | 0x00 |
396     ;            | 0x00 |
397     ;            | 0x00 |
398     ;            | 0x00 |
399     ;            | 0x00 |
400     ;            | 0x00 |
401     ;            | 0xF0 | 0x7fffffffda96
402     ;            | 0xDE |
403     ;            | 0xBC |
404     ;            | 0x9A |
405     ;            | 0xFF |
406     ;            | 0xFF |
407     ;            | 0xFF | 0x7fffffffda9c
408     ;            | 0xFF |
409     ;            | 0xF0 | 0x7fffffffda9e
410     ;            | 0xDE |
411     ;            | 0xBC |
412     ;            | 0x9A |
413     ;            | 0xFF |
414     ;            | 0xFF |
415     ;            | 0xFF | 0x7fffffffdaa4
416     ;            | 0xFF |
417     ;            | 0xF0 | 0x7fffffffdaa6
418     ;            | 0xDE |
```

```
419     ;            | 0xF0 | 0x7fffffffdaa8
420     ;            | 0xFF |
421     ;            | 0xFF |
422     ;            | 0xFF |
423     ;            | 0xFF |
424     ;            | 0xFF |
425     ;            | 0xFF | 0x7fffffffdaae
426     ;            | 0xFF |
427     ;            +------+ grandes adresses 0x7fffffffdab0
428     ; rem. : le contenu initial de rsp peut varier
429
430     push    0x123456789ABCDEF0
431     ; warning: signed dword immediate exceeds bounds [-w+number-overflow]
432     ; warning: dword data exceeds bounds [-w+number-overflow]
433     ; les 32 bits de poids faible de la valeur 0x123456789ABCDEF0,
434     ; c'est-à-dire la valeur 0x9ABCDEF0 est étendue par extension
435     ; de signe sur 64 bits et placée sur la pile, c'est-à-dire que :
436     ; la valeur 0xFFFFFFFF9ABCDEF0 est placée sur la pile sur 8 bytes
437     ;   rsp <-- rsp - 8
438     ;
439     ;            |      | petites adresses
440     ; rsp --->   | 0xF0 | 0x7fffffffda86
441     ;            | 0xDE |
442     ;            | 0xBC |
443     ;            | 0x9A |
444     ;            | 0xFF |
445     ;            | 0xFF |
446     ;            | 0xFF |
447     ;            | 0xFF |
448     ;            | 0x00 | 0x7fffffffda8e
449     ;            | 0x00 |
450     ;            | 0x00 |
451     ;            | 0x00 |
452     ;            | 0x00 |
453     ;            | 0x00 |
454     ;            | 0x00 |
455     ;            | 0x00 |
456     ;            | 0xF0 | 0x7fffffffda96
457     ;            | 0xDE |
458     ;            | 0xBC |
459     ;            | 0x9A |
460     ;            | 0xFF |
461     ;            | 0xFF |
462     ;            | 0xFF | 0x7fffffffda9c
```

```
463     ;               | 0xFF |
464     ;               | 0xF0 | 0x7fffffffda9e
465     ;               | 0xDE |
466     ;               | 0xBC |
467     ;               | 0x9A |
468     ;               | 0xFF |
469     ;               | 0xFF |
470     ;               | 0xFF | 0x7fffffffdaa4
471     ;               | 0xFF |
472     ;               | 0xF0 | 0x7fffffffdaa6
473     ;               | 0xDE |
474     ;               | 0xF0 | 0x7fffffffdaa8
475     ;               | 0xFF |
476     ;               | 0xFF |
477     ;               | 0xFF |
478     ;               | 0xFF |
479     ;               | 0xFF |
480     ;               | 0xFF | 0x7fffffffdaae
481     ;               | 0xFF |
482     ;               +------+ grandes adresses 0x7fffffffdab0
483     ; rem. : le contenu initial de rsp peut varier
484
485
486     mov     rax, 60
487     mov     rdi, 0
488     syscall
```

## 4.2 Accès à la pile sans `push` ni `pop`

Dans ce deuxième code source on empile et dépile des données de taille 1 et 4 *bytes* via les instructions `sub`, `mov` et `add` :

```
1  ; 02_add_sub_stack_fc.asm
2  ;
3  ; nasm -f elf64 -F dwarf 02_add_sub_stack_fc.asm
4  ; ld -o 02_add_sub_stack_fc 02_add_sub_stack_fc.o
5
6  global _start
7
8  section .bss
9      variable    RESB    1
10
11 section .text
12 _start:
```

```
13
14      ; pile vide
15      ;                       petites adresses
16      ;            |      |
17      ; rsp --->   +------+  grandes adresses 0x7fffffffdab0
18      ; rem. : le contenu initial de rsp peut varier
19
20      ; push
21
22      mov     rax, 0x123456789ABCDEF0
23
24      ; push    al            ; ko car al fait 8 bits
25      ; error: invalid combination of opcode and operands
26
27      sub     rsp, 1
28      mov     byte [rsp], al
29      ; la valeur 0xF0 est placée sur la pile sur 1 byte
30      ;   rsp <-- rsp - 1
31      ;
32      ;            |      | petites adresses
33      ; rsp --->   | 0xF0 | 0x7fffffffdaaf
34      ;            +------+ grandes adresses 0x7fffffffdab0
35      ; rem. : le contenu initial de rsp peut varier
36
37      push    ax
38      ; la valeur 0xDEF0 est placée sur la pile sur 2 bytes
39      ;   rsp <-- rsp - 2
40      ;
41      ;            |      | petites adresses
42      ; rsp --->   | 0xF0 | 0x7fffffffdaad
43      ;            | 0xDE |
44      ;            | 0xF0 | 0x7fffffffdaaf
45      ;            +------+ grandes adresses 0x7fffffffdab0
46      ; rem. : le contenu initial de rsp peut varier
47
48      ; push    eax           ; ko car eax fait 32 bits
49      ; error: instruction not supported in 64-bit mode
50
51      sub     rsp, 4
52      mov     dword [rsp], eax
53      ; la valeur 0x9ABCDEF0 est placée sur la pile sur 4 bytes
54      ;   rsp <-- rsp - 4
55      ;
56      ;            |      | petites adresses
```

```
57      ; rsp --->  | 0xF0 | 0x7fffffffdaa9
58      ;           | 0xDE |
59      ;           | 0xBC |
60      ;           | 0x9A |
61      ;           | 0xF0 | 0x7fffffffdaad
62      ;           | 0xDE |
63      ;           | 0xF0 | 0x7fffffffdaaf
64      ;           +------+ grandes adresses 0x7fffffffdab0
65      ; rem. : le contenu initial de rsp peut varier
66
67      push    rax
68      ; la valeur 0x123456789ABCDEF0 est placée sur la pile sur 8 bytes
69      ;   rsp <-- rsp - 8
70      ;
71      ;           |      | petites adresses
72      ; rsp --->  | 0xF0 | 0x7fffffffdaa1
73      ;           | 0xDE |
74      ;           | 0xBC |
75      ;           | 0x9A |
76      ;           | 0x78 |
77      ;           | 0x56 |
78      ;           | 0x34 |
79      ;           | 0x12 |
80      ;           | 0xF0 | 0x7fffffffdaa9
81      ;           | 0xDE |
82      ;           | 0xBC |
83      ;           | 0x9A |
84      ;           | 0xF0 | 0x7fffffffdaad
85      ;           | 0xDE |
86      ;           | 0xF0 | 0x7fffffffdaaf
87      ;           +------+ grandes adresses 0x7fffffffdab0
88      ; rem. : le contenu initial de rsp peut varier
89
90      ; push    byte [variable]     ; ko car 1 byte
91      ; error: invalid combination of opcode and operands
92
93      ; sub     rsp, 1
94      ; mov     byte [rsp], byte [variable] ; ko car 2 mémoires
95      ; error: invalid combination of opcode and operands
96
97      ; pop
98
99      ; pop     r10b            ; ko car r10b fait 8 bits
100     ; error: invalid combination of opcode and operands
```

```
101
102     mov     r10b, byte [rsp]
103     add     rsp, 1
104     ; la valeur 0xF0 est placée dans r10b et rsp incrémenté de 1
105     ;   r10b <-- 0xF0
106     ;   rsp <-- rsp + 1
107     ;
108     ;           |      | petites adresses
109     ;           | 0xF0 | 0x7fffffffdaa1
110     ; rsp --->  | 0xDE | 0x7fffffffdaa2
111     ;           | 0xBC |
112     ;           | 0x9A |
113     ;           | 0x78 |
114     ;           | 0x56 |
115     ;           | 0x34 |
116     ;           | 0x12 |
117     ;           | 0xF0 | 0x7fffffffdaa9
118     ;           | 0xDE |
119     ;           | 0xBC |
120     ;           | 0x9A |
121     ;           | 0xF0 | 0x7fffffffdaad
122     ;           | 0xDE |
123     ;           | 0xF0 | 0x7fffffffdaaf
124     ;           +------+ grandes adresses 0x7fffffffdab0
125     ; rem. : le contenu initial de rsp peut varier
126
127     pop     r11w
128     ; la valeur 0xBCDE est placée dans r11w et rsp incrémenté de 2
129     ;   r11w <-- 0xBCDE
130     ;   rsp <-- rsp + 2
131     ;
132     ;           |      | petites adresses
133     ;           | 0xF0 | 0x7fffffffdaa1
134     ;           | 0xDE | 0x7fffffffdaa2
135     ;           | 0xBC |
136     ; rsp --->  | 0x9A | 0x7fffffffdaa4
137     ;           | 0x78 |
138     ;           | 0x56 |
139     ;           | 0x34 |
140     ;           | 0x12 |
141     ;           | 0xF0 | 0x7fffffffdaa9
142     ;           | 0xDE |
143     ;           | 0xBC |
144     ;           | 0x9A |
```

```
145     ;           | 0xF0 | 0x7fffffffdaad
146     ;           | 0xDE |
147     ;           | 0xF0 | 0x7fffffffdaaf
148     ;           +------+ grandes adresses 0x7fffffffdab0
149     ; rem. : le contenu initial de rsp peut varier
150
151     ; pop     r12d            ; ko car r12d fait 32 bits
152     ; error: instruction not supported in 64-bit mode
153
154     mov     r12d, dword [rsp]
155     add     rsp, 4
156     ; la valeur 0x3456789A est placée dans r12d et rsp incrémenté de 4
157     ;   r12d <-- 0x3456789A
158     ;   rsp <-- rsp + 4
159     ;
160     ;           |      | petites adresses
161     ;           | 0xF0 | 0x7fffffffdaa1
162     ;           | 0xDE | 0x7fffffffdaa2
163     ;           | 0xBC |
164     ;           | 0x9A | 0x7fffffffdaa4
165     ;           | 0x78 |
166     ;           | 0x56 |
167     ;           | 0x34 |
168     ; rsp --->  | 0x12 | 0x7fffffffdaa8
169     ;           | 0xF0 | 0x7fffffffdaa9
170     ;           | 0xDE |
171     ;           | 0xBC |
172     ;           | 0x9A |
173     ;           | 0xF0 | 0x7fffffffdaad
174     ;           | 0xDE |
175     ;           | 0xF0 | 0x7fffffffdaaf
176     ;           +------+ grandes adresses 0x7fffffffdab0
177     ; rem. : le contenu initial de rsp peut varier
178
179     pop     r13
180     ; la valeur 0xF0DEF09ABCDEF012 est placée dans r13 et rsp
181     ; incrémenté de 8
182     ;   r13 <-- 0xF0DEF09ABCDEF012
183     ;   rsp <-- rsp + 8
184     ;
185     ;           |      | petites adresses
186     ;           | 0xF0 | 0x7fffffffdaa1
187     ;           | 0xDE | 0x7fffffffdaa2
188     ;           | 0xBC |
```

```
189     ;             | 0x9A | 0x7fffffffdaa4
190     ;             | 0x78 |
191     ;             | 0x56 |
192     ;             | 0x34 |
193     ;             | 0x12 | 0x7fffffffdaa8
194     ;             | 0xF0 | 0x7fffffffdaa9
195     ;             | 0xDE |
196     ;             | 0xBC |
197     ;             | 0x9A |
198     ;             | 0xF0 | 0x7fffffffdaad
199     ;             | 0xDE |
200     ;             | 0xF0 | 0x7fffffffdaaf
201     ; rsp --->  +-------+ grandes adresses 0x7fffffffdab0
202     ; rem. : le contenu initial de rsp peut varier
203
204     mov     rax, 60
205     mov     rdi, 0
206     syscall
```

# 5 Variable locale

Dans les langages de plus haut niveau que le langage d'assemblage, une variable locale [16] est une variable dont la portée est limitée au bloc ou à la fonction où elle est définie. Communément, une variable locale est automatique [17] : lors de l'exécution de son programme, elle nait au début du bloc où se trouve l'instruction de sa définition [18] et meurt à la fin de ce bloc.

## 5.1 Mise en œuvre en langage d'assemblage

En langage d'assemblage, la réalisation d'une variable locale est obtenue soit à l'aide d'un registre, mais il faut en avoir un libre de bonne taille, soit de la pile, mais c'est coûteux en performance car la pile est en mémoire. Corollairement, l'aspect automatique doit entièrement être pris en charge par le programmeur lors de l'utilisation de variables locales sur la pile !

### 5.1.1 Registre

Utiliser un registre comme variable locale ne nécessite aucune explication supplémentaire. Évidemment, une telle variable n'a pas d'adresse.

16. https://en.wikipedia.org/wiki/Local_variable (consulté le 7 avril 2020).
17. https://en.wikipedia.org/wiki/Automatic_variable (consulté le 7 avril 2020).
18. Mais elle n'est accessible qu'après sa définition.

Comme les accès aux registres sont beaucoup plus rapides [19] que ceux en mémoire, il est encouragé de privilégier l'utilisation de registres comme variables locales. Ce n'est cependant pas toujours possible. Le nombre ou la taille des variables locales [20] ou encore l'obligation de passer par la mémoire [21], par exemple, sont des circonstances rédhibitoires à l'usage de registres en guise de variables locales.

### 5.1.2 Pile

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` [22] / `inc` [23] 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 à la section 3 et dans la Table 2.

Cependant, pour la gestion des variables locales des fonctions et procédures, l'utilisation du couple d'instructions dédiées `enter` [24] et `leave` [25] est encore plus simple. On n'en dit pas plus à leur sujet ici. On les utilise dans le TD08, consacré aux fonctions.

## 5.2 Mise en pratique

### 5.2.1 Premiers exemples basiques

**Langage C** Voici un code écrit dans le langage de programmation de haut niveau [26] qu'est le langage C [27], où, dans un bloc, deux variables sont définies et initialisées avant d'être utilisées :

19. https://colin-scott.github.io/personal_website/research/interactive_latency.html (consulté le 7 avril 2020).
20. On pense ici par exemple aux tableaux.
21. On pense par exemple ici à certains appels système tels `read` ou `write` (voir la section 5.2.2).
22. https://github.com/HJLebbink/asm-dude/wiki/dec (consulté le 7 avril 2020).
23. https://github.com/HJLebbink/asm-dude/wiki/inc (consulté le 7 avril 2020).
24. https://www.felixcloutier.com/x86/enter (consulté le 7 avril 2020).
25. https://www.felixcloutier.com/x86/leave (consulté le 7 avril 2020).
26. https://fr.wikipedia.org/wiki/Langage_de_programmation_de_haut_niveau (consulté le 7 avril 2020).
27. https://fr.wikipedia.org/wiki/C_(langage) (consulté le 7 avril 2020).

```c
// 03_a_loc_var.c
//
// gcc -o 03_a_loc_var_c -std=c11 -Wall -pedantic-errors -g 03_a_loc_var.c
// kdbg 03_a_loc_var_c

int main()  // point d'entrée du programme
{

// ici il peut y avoir du code : i et j n'existent pas

    {   // début du bloc où vivent i et j
        long int i = 4;
        long int j = -8;

        i = 23;
        i -= 4;
        ++j;
    }   // fin du bloc où vivent i et j

// ici il peut y avoir du code : i et j n'existent plus

    return 0;       // équivalent à exit(0);
}   // fin de main()
```

**Langage d'assemblage** Voici un code en langage d'assemblage réalisant les mêmes traitements que ceux du code en langage C fourni juste avant :

```asm
; 03_a_loc_var_fc.asm
;
; nasm -f elf64 -F dwarf 03_a_loc_var_fc.asm
; ld -o 03_a_loc_var_asm 03_a_loc_var_fc.o
; kdbg 03_a_loc_var_asm

global _start

section .text
_start:         ; point d'entrée du programme

; ici il peut y avoir du code : i et j n'existent pas

    ; pile vide
    ;                       petites adresses
    ;           |       |
```

```
17      ; rsp --->  +-------+  grandes adresses 0x7fffffffd3c0
18      ; rem. : le contenu initial de rsp peut varier
19
20      ; sauvegarde du pointeur de contexte de pile (stack frame pointer)
21      ; https://en.wikipedia.org/wiki/Call_stack#Structure
22      ; rem. : c'est inutile ici puisqu'on est dans _start
23      ;
24      ; début du bloc où vivent i et j
25      ; les 2 instructions qui suivent sont équivalentes à l'accolade
26      ; ouvrante ({) du code en langage C
27      push    rbp
28      ; le contenu de rbp est sauvegardé sur la pile
29      ; la valeur 0x00 est placée sur la pile sur 8 bytes
30      ;   rsp <-- rsp - 8
31      ;
32      ;           |      | petites adresses
33      ; rsp --->  | 0x00 | 0x7fffffffd3b8
34      ;           | 0x00 |
35      ;           | 0x00 |
36      ;           | 0x00 |
37      ;           | 0x00 |
38      ;           | 0x00 |
39      ;           | 0x00 |
40      ;           | 0x00 |
41      ;           +------+ grandes adresses 0x7fffffffd3c0
42      ; rem. : le contenu initial de rsp peut varier
43
44      ; mise à jour du pointeur de contexte de pile (stack frame pointer)
45      mov     rbp, rsp
46      ;
47      ;           |      | petites adresses
48      ; rsp --->  | 0x00 | 0x7fffffffd3b8  <--- rbp
49      ;           | 0x00 |
50      ;           | 0x00 |
51      ;           | 0x00 |
52      ;           | 0x00 |
53      ;           | 0x00 |
54      ;           | 0x00 |
55      ;           | 0x00 |
56      ;           +------+ grandes adresses 0x7fffffffd3c0
57      ; rem. : le contenu initial de rsp peut varier
58
59      push    qword 4     ; équivalent définition + initialisation de i
60      ;
```

```
61      ;             |      | petites adresses
62      ; rsp --->    | 0x04 | 0x7fffffffd3b0 (i)
63      ;             | 0x00 |
64      ;             | 0x00 |
65      ;             | 0x00 |
66      ;             | 0x00 |
67      ;             | 0x00 |
68      ;             | 0x00 |
69      ;             | 0x00 |
70      ; rbp --->    | 0x00 | 0x7fffffffd3b8
71      ;             | 0x00 |
72      ;             | 0x00 |
73      ;             | 0x00 |
74      ;             | 0x00 |
75      ;             | 0x00 |
76      ;             | 0x00 |
77      ;             | 0x00 |
78      ;             +------+ grandes adresses 0x7fffffffd3c0
79      ; rem. : le contenu initial de rsp peut varier
80
81      push    qword -8    ; équivalent définition + initialisation de j
82      ;
83      ;             |      | petites adresses
84      ; rsp --->    | 0xF8 | 0x7fffffffd3a8 (j)
85      ;             | 0xFF |
86      ;             | 0xFF |
87      ;             | 0xFF |
88      ;             | 0xFF |
89      ;             | 0xFF |
90      ;             | 0xFF |
91      ;             | 0xFF |
92      ;             | 0x04 | 0x7fffffffd3b0 (i)
93      ;             | 0x00 |
94      ;             | 0x00 |
95      ;             | 0x00 |
96      ;             | 0x00 |
97      ;             | 0x00 |
98      ;             | 0x00 |
99      ;             | 0x00 |
100     ; rbp --->    | 0x00 | 0x7fffffffd3b8
101     ;             | 0x00 |
102     ;             | 0x00 |
103     ;             | 0x00 |
104     ;             | 0x00 |
```

```
105     ;            | 0x00 |
106     ;            | 0x00 |
107     ;            | 0x00 |
108     ;            +------+ grandes adresses 0x7fffffffd3c0
109     ; rem. : le contenu initial de rsp peut varier
110
111     mov     qword [rbp - 8], 23     ; i = 23
112     ;
113     ;            |      | petites adresses
114     ; rsp --->   | 0xF8 | 0x7fffffffd3a8 (j)
115     ;            | 0xFF |
116     ;            | 0xFF |
117     ;            | 0xFF |
118     ;            | 0xFF |
119     ;            | 0xFF |
120     ;            | 0xFF |
121     ;            | 0xFF |
122     ;            | 0x17 | 0x7fffffffd3b0 (i)
123     ;            | 0x00 |
124     ;            | 0x00 |
125     ;            | 0x00 |
126     ;            | 0x00 |
127     ;            | 0x00 |
128     ;            | 0x00 |
129     ;            | 0x00 |
130     ; rbp --->   | 0x00 | 0x7fffffffd3b8
131     ;            | 0x00 |
132     ;            | 0x00 |
133     ;            | 0x00 |
134     ;            | 0x00 |
135     ;            | 0x00 |
136     ;            | 0x00 |
137     ;            | 0x00 |
138     ;            +------+ grandes adresses 0x7fffffffd3c0
139     ; rem. : le contenu initial de rsp peut varier
140
141     sub     qword [rbp - 8], 4      ; i -= 4;
142     ;
143     ;            |      | petites adresses
144     ; rsp --->   | 0xF8 | 0x7fffffffd3a8 (j)
145     ;            | 0xFF |
146     ;            | 0xFF |
147     ;            | 0xFF |
148     ;            | 0xFF |
```

```
149     ;             | 0xFF |
150     ;             | 0xFF |
151     ;             | 0xFF |
152     ;             | 0x13 | 0x7fffffffd3b0 (i)
153     ;             | 0x00 |
154     ;             | 0x00 |
155     ;             | 0x00 |
156     ;             | 0x00 |
157     ;             | 0x00 |
158     ;             | 0x00 |
159     ;             | 0x00 |
160     ; rbp --->     | 0x00 | 0x7fffffffd3b8
161     ;             | 0x00 |
162     ;             | 0x00 |
163     ;             | 0x00 |
164     ;             | 0x00 |
165     ;             | 0x00 |
166     ;             | 0x00 |
167     ;             | 0x00 |
168     ;             +------+ grandes adresses 0x7fffffffd3c0
169     ; rem. : le contenu initial de rsp peut varier
170
171     inc     qword [rbp - 16]        ; ++j;
172     ;
173     ;             |      | petites adresses
174     ; rsp --->     | 0xF9 | 0x7fffffffd3a8 (j)
175     ;             | 0xFF |
176     ;             | 0xFF |
177     ;             | 0xFF |
178     ;             | 0xFF |
179     ;             | 0xFF |
180     ;             | 0xFF |
181     ;             | 0xFF |
182     ;             | 0x13 | 0x7fffffffd3b0 (i)
183     ;             | 0x00 |
184     ;             | 0x00 |
185     ;             | 0x00 |
186     ;             | 0x00 |
187     ;             | 0x00 |
188     ;             | 0x00 |
189     ;             | 0x00 |
190     ; rbp --->     | 0x00 | 0x7fffffffd3b8
191     ;             | 0x00 |
192     ;             | 0x00 |
```

```
193     ;           | 0x00 |
194     ;           | 0x00 |
195     ;           | 0x00 |
196     ;           | 0x00 |
197     ;           | 0x00 |
198     ;           +------+ grandes adresses 0x7fffffffd3c0
199     ; rem. : le contenu initial de rsp peut varier
200
201     ; fin du bloc où vivent i et j
202     ; les 2 lignes qui suivent sont équivalentes à l'accolade
203     ; fermante (}) du code en langage C
204     ;
205     ; destruction des variables locales
206     mov     rsp, rbp
207     ;
208     ;           |      | petites adresses
209     ;           | 0xF9 | 0x7fffffffd3a8 (j)
210     ;           | 0xFF |
211     ;           | 0xFF |
212     ;           | 0xFF |
213     ;           | 0xFF |
214     ;           | 0xFF |
215     ;           | 0xFF |
216     ;           | 0xFF |
217     ;           | 0x13 | 0x7fffffffd3b0 (i)
218     ;           | 0x00 |
219     ;           | 0x00 |
220     ;           | 0x00 |
221     ;           | 0x00 |
222     ;           | 0x00 |
223     ;           | 0x00 |
224     ;           | 0x00 |
225     ; rsp --->  | 0x00 | 0x7fffffffd3b8  <--- rbp
226     ;           | 0x00 |
227     ;           | 0x00 |
228     ;           | 0x00 |
229     ;           | 0x00 |
230     ;           | 0x00 |
231     ;           | 0x00 |
232     ;           | 0x00 |
233     ;           +------+ grandes adresses 0x7fffffffd3c0
234     ; rem. : le contenu initial de rsp peut varier
235
236     ; restauration de la valeur initiale de rbp
```

```
237     ; rem. : c'est inutile ici car on est dans _start
238     pop     rbp
239     ;
240     ;           |      | petites adresses
241     ;           | 0xF9 | 0x7fffffffd3a8 (j)
242     ;           | 0xFF |
243     ;           | 0xFF |
244     ;           | 0xFF |
245     ;           | 0xFF |
246     ;           | 0xFF |
247     ;           | 0xFF |
248     ;           | 0xFF |
249     ;           | 0x13 | 0x7fffffffd3b0 (i)
250     ;           | 0x00 |
251     ;           | 0x00 |
252     ;           | 0x00 |
253     ;           | 0x00 |
254     ;           | 0x00 |
255     ;           | 0x00 |
256     ;           | 0x00 |
257     ;           | 0x00 | 0x7fffffffd3b8
258     ;           | 0x00 |
259     ;           | 0x00 |
260     ;           | 0x00 |
261     ;           | 0x00 |
262     ;           | 0x00 |
263     ;           | 0x00 |
264     ;           | 0x00 |
265     ; rsp --->  +------+ grandes adresses 0x7fffffffd3c0
266     ; rem. : le contenu initial de rsp peut varier
267
268 ; ici il peut y avoir du code : i et j n'existent plus
269
270     ; les 3 lignes qui suivent sont équivalentes à return 0;
271     mov     rax, 60
272     mov     rdi, 0
273     syscall
274
275 ; on n'arrive jamais ici
```

Les parties relatives à la sauvegarde / restauration de `rbp` qui remplit le rôle de pointeur de contexte de pile (*stack frame pointer*[28]) sont développées et expliquées lors

28. https://en.wikipedia.org/wiki/Call_stack#Structure (consulté le 7 avril 2020).

du TD08.

**Langage C bis** Voici un second code écrit en langage C, où, dans un bloc, deux variables sont définies sans être initialisées puis sont utilisées :

```c
// 03_b_loc_var.c
//
// gcc -o 03_b_loc_var_c -std=c11 -Wall -pedantic-errors -g 03_b_loc_var.c
// kdbg 03_b_loc_var_c

int main()  // point d'entrée du programme
{

// ici il peut y avoir du code : i et j n'existent pas

    {   // début du bloc où vivent i et j
        long int i, j;
        // ici les contenus de i et j sont indéterminés
        i = 23;
        j = i;
    }   // fin du bloc où vivent i et j

// ici il peut y avoir du code : i et j n'existent plus

    return 0;       // équivalent à exit(0);
}   // fin de main()
```

**Langage d'assemblage bis** Et voici à nouveau un code en langage d'assemblage qui réalise les mêmes opérations que celles du code en langage C du paragraphe précédent :

```asm
; 03_b_loc_var_fc.asm
;
; nasm -f elf64 -F dwarf 03_b_loc_var_fc.asm
; ld -o 03_b_loc_var_asm 03_b_loc_var_fc.o
; kdbg 03_b_loc_var_asm

global _start

section .text
_start:         ; point d'entrée du programme

; ici il peut y avoir du code : i et j n'existent pas

```

```
14      ; pile vide
15      ;                       petites adresses
16      ;             |       |
17      ; rsp --->  +-------+  grandes adresses 0x7fffffffd3c0
18      ; rem. : le contenu initial de rsp peut varier
19
20      ; sauvegarde du pointeur de contexte de pile (stack frame pointer)
21      ; https://en.wikipedia.org/wiki/Call_stack#Structure
22      ; rem. : c'est inutile ici puisqu'on est dans _start
23      ;
24      ; début du bloc où vivent i et j
25      ; les 2 instructions qui suivent sont équivalentes à l'accolade
26      ; ouvrante ({) du code en langage C
27      push    rbp
28      ; le contenu de rbp est sauvegardé sur la pile
29      ; la valeur 0x00 est placée sur la pile sur 8 bytes
30      ;   rsp <-- rsp - 8
31      ;
32      ;             |       | petites adresses
33      ; rsp --->   | 0x00 | 0x7fffffffd3b8
34      ;             | 0x00 |
35      ;             | 0x00 |
36      ;             | 0x00 |
37      ;             | 0x00 |
38      ;             | 0x00 |
39      ;             | 0x00 |
40      ;             | 0x00 |
41      ;             +-------+ grandes adresses 0x7fffffffd3c0
42      ; rem. : le contenu initial de rsp peut varier
43
44      ; mise à jour du pointeur de contexte de pile (stack frame pointer)
45      mov     rbp, rsp
46      ;
47      ;             |       | petites adresses
48      ; rsp --->   | 0x00 | 0x7fffffffd3b8  <--- rbp
49      ;             | 0x00 |
50      ;             | 0x00 |
51      ;             | 0x00 |
52      ;             | 0x00 |
53      ;             | 0x00 |
54      ;             | 0x00 |
55      ;             | 0x00 |
56      ;             +-------+ grandes adresses 0x7fffffffd3c0
57      ; rem. : le contenu initial de rsp peut varier
```

```
58
59      ; définition de 2 variables de 8 bytes chacune
60      sub     rsp, 2 * 8      ; équivalent définition de i et j
61      ;
62      ;           |      | petites adresses
63      ; rsp --->  | 0x?? | 0x7fffffffd3a8 (j)
64      ;           | 0x?? |
65      ;           | 0x?? |
66      ;           | 0x?? |
67      ;           | 0x?? |
68      ;           | 0x?? |
69      ;           | 0x?? |
70      ;           | 0x?? |
71      ;           | 0x?? | 0x7fffffffd3b0 (i)
72      ;           | 0x?? |
73      ;           | 0x?? |
74      ;           | 0x?? |
75      ;           | 0x?? |
76      ;           | 0x?? |
77      ;           | 0x?? |
78      ;           | 0x?? |
79      ; rbp --->  | 0x00 | 0x7fffffffd3b8
80      ;           | 0x00 |
81      ;           | 0x00 |
82      ;           | 0x00 |
83      ;           | 0x00 |
84      ;           | 0x00 |
85      ;           | 0x00 |
86      ;           | 0x00 |
87      ;           +------+ grandes adresses 0x7fffffffd3c0
88      ; rem. : le contenu initial de rsp peut varier
89
90      ; ici les contenus de i et j sont indéterminés...
91      ; ça dépend de ce qui traîne sur la pile
92
93      mov     qword [rbp - 8], 23     ; i = 23
94      ;
95      ;           |      | petites adresses
96      ; rsp --->  | 0x?? | 0x7fffffffd3a8 (j)
97      ;           | 0x?? |
98      ;           | 0x?? |
99      ;           | 0x?? |
100     ;           | 0x?? |
101     ;           | 0x?? |
```

```
102     ;             | 0x?? |
103     ;             | 0x?? |
104     ;             | 0x17 | 0x7fffffffd3b0 (i)
105     ;             | 0x00 |
106     ;             | 0x00 |
107     ;             | 0x00 |
108     ;             | 0x00 |
109     ;             | 0x00 |
110     ;             | 0x00 |
111     ;             | 0x00 |
112     ; rbp --->     | 0x00 | 0x7fffffffd3b8
113     ;             | 0x00 |
114     ;             | 0x00 |
115     ;             | 0x00 |
116     ;             | 0x00 |
117     ;             | 0x00 |
118     ;             | 0x00 |
119     ;             | 0x00 |
120     ;             +------+ grandes adresses 0x7fffffffd3c0
121     ; rem. : le contenu initial de rsp peut varier
122
123     ; mov mem, mem interdit
124     ;  => utilisation d'un registre intermédiaire
125     mov     r8, [rbp - 8]           ; r8 registre intermédiaire
126     mov     [rbp - 16], r8          ; j = i;
127     ;
128     ;             |      | petites adresses
129     ; rsp --->     | 0x17 | 0x7fffffffd3a8 (j)
130     ;             | 0x00 |
131     ;             | 0x00 |
132     ;             | 0x00 |
133     ;             | 0x00 |
134     ;             | 0x00 |
135     ;             | 0x00 |
136     ;             | 0x00 |
137     ;             | 0x17 | 0x7fffffffd3b0 (i)
138     ;             | 0x00 |
139     ;             | 0x00 |
140     ;             | 0x00 |
141     ;             | 0x00 |
142     ;             | 0x00 |
143     ;             | 0x00 |
144     ;             | 0x00 |
145     ; rbp --->     | 0x00 | 0x7fffffffd3b8
```

```
146     ;           | 0x00 |
147     ;           | 0x00 |
148     ;           | 0x00 |
149     ;           | 0x00 |
150     ;           | 0x00 |
151     ;           | 0x00 |
152     ;           | 0x00 |
153     ;           +------+ grandes adresses 0x7fffffffd3c0
154     ; rem. : le contenu initial de rsp peut varier
155
156     ; fin du bloc où vivent i et j
157     ; les 2 lignes qui suivent sont équivalentes à l'accolade
158     ; fermante (}) du code en langage C
159     ;
160     ; destruction des variables locales
161     mov     rsp, rbp
162     ;
163     ;           |      | petites adresses
164     ;           | 0x17 | 0x7fffffffd3a8 (j)
165     ;           | 0x00 |
166     ;           | 0x00 |
167     ;           | 0x00 |
168     ;           | 0x00 |
169     ;           | 0x00 |
170     ;           | 0x00 |
171     ;           | 0x00 |
172     ;           | 0x17 | 0x7fffffffd3b0 (i)
173     ;           | 0x00 |
174     ;           | 0x00 |
175     ;           | 0x00 |
176     ;           | 0x00 |
177     ;           | 0x00 |
178     ;           | 0x00 |
179     ;           | 0x00 |
180     ; rsp --->  | 0x00 | 0x7fffffffd3b8  <--- rbp
181     ;           | 0x00 |
182     ;           | 0x00 |
183     ;           | 0x00 |
184     ;           | 0x00 |
185     ;           | 0x00 |
186     ;           | 0x00 |
187     ;           | 0x00 |
188     ;           +------+ grandes adresses 0x7fffffffd3c0
189     ; rem. : le contenu initial de rsp peut varier
```

```
190
191     ; restauration de la valeur initiale de rbp
192     ; rem. : c'est inutile ici car on est dans _start
193     pop     rbp
194     ;
195     ;           |      | petites adresses
196     ;           | 0x17 | 0x7fffffffd3a8 (j)
197     ;           | 0x00 |
198     ;           | 0x00 |
199     ;           | 0x00 |
200     ;           | 0x00 |
201     ;           | 0x00 |
202     ;           | 0x00 |
203     ;           | 0x00 |
204     ;           | 0x17 | 0x7fffffffd3b0 (i)
205     ;           | 0x00 |
206     ;           | 0x00 |
207     ;           | 0x00 |
208     ;           | 0x00 |
209     ;           | 0x00 |
210     ;           | 0x00 |
211     ;           | 0x00 |
212     ;           | 0x00 | 0x7fffffffd3b8
213     ;           | 0x00 |
214     ;           | 0x00 |
215     ;           | 0x00 |
216     ;           | 0x00 |
217     ;           | 0x00 |
218     ;           | 0x00 |
219     ;           | 0x00 |
220     ; rsp --->  +------+ grandes adresses 0x7fffffffd3c0
221     ; rem. : le contenu initial de rsp peut varier
222
223 ; ici il peut y avoir du code : i et j n'existent plus
224
225     ; les 3 lignes qui suivent sont équivalentes à return 0;
226     mov     rax, 60
227     mov     rdi, 0
228     syscall
229
230 ; on n'arrive jamais ici
```

### 5.2.2 `cat` sans argument

Les appels système `read` et `write`, vus au cours du TD05, attendent chacun en deuxième argument l'adresse d'un tampon (*buffer*) en lecture / écriture. Comme il s'agit d'une adresse en mémoire, on ne peut s'en tirer uniquement avec des registres.

Un des exercices du TD06 consiste en la réalisation en langage d'assemblage de la commande filtre `cat` sans argument. Pour y arriver, à la lumière des connaissances acquises jusqu'alors, on définit une variable d'un *byte* dans la `section .bss` pour faire office de tampon. Voyons maintenant une version alternative, où le *buffer* est une variable locale :

```
1  ; 04_cat_local_variable_fc.asm
2  ;
3  ; nasm -f elf64 -F dwarf 04_cat_local_variable_fc.asm
4  ; ld -o 04_cat 04_cat_local_variable_fc.o
5
6  global _start
7
8  section .text
9  _start:
10
11     ; pile vide
12     ;                       petites adresses
13     ;            |      |
14     ; rsp --->  +------+  grandes adresses 0x7fffffffd3c0
15     ; rem. : le contenu initial de rsp peut varier
16
17     ; sauvegarde du pointeur de contexte de pile (stack frame pointer)
18     ; https://en.wikipedia.org/wiki/Call_stack#Structure
19     ; rem. : c'est inutile ici puisqu'on est dans _start
20     push    rbp          ; sauvegarde du contenu original de rbp
21     ; le contenu de rbp est sauvegardé sur la pile
22     ; la valeur 0x00 est placée sur la pile sur 8 bytes
23     ;   rsp <-- rsp - 8
24     ;
25     ;            |      | petites adresses
26     ; rsp --->  | 0x00 | 0x7fffffffd3b8
27     ;            | 0x00 |
28     ;            | 0x00 |
29     ;            | 0x00 |
30     ;            | 0x00 |
31     ;            | 0x00 |
32     ;            | 0x00 |
33     ;            | 0x00 |
34     ;            +------+ grandes adresses 0x7fffffffd3c0
```

```
35      ; rem. : le contenu initial de rsp peut varier
36
37      ; mise à jour du pointeur de contexte de pile (stack frame pointer)
38      mov     rbp, rsp    ; rbp pointe sur sa sauvegarde sur la pile
39      ;
40      ;           |      | petites adresses
41      ; rsp --->  | 0x00 | 0x7fffffffd3b8  <--- rbp
42      ;           | 0x00 |
43      ;           | 0x00 |
44      ;           | 0x00 |
45      ;           | 0x00 |
46      ;           | 0x00 |
47      ;           | 0x00 |
48      ;           | 0x00 |
49      ;           +------+ grandes adresses 0x7fffffffd3c0
50      ; rem. : le contenu initial de rsp peut varier
51
52      ; création / initialisation des variables locales
53      sub     rsp, 1      ; trou dans la pile : ici 1! byte
54      ; on n'a pas besoin d'initialiser cette variable
55      ;
56      ;           |      | petites adresses
57      ; rsp --->  | 0x?? | 0x7fffffffd3b7 : adresse de la variable locale
58      ; rbp --->  | 0x00 | 0x7fffffffd3b8
59      ;           | 0x00 |
60      ;           | 0x00 |
61      ;           | 0x00 |
62      ;           | 0x00 |
63      ;           | 0x00 |
64      ;           | 0x00 |
65      ;           | 0x00 |
66      ;           +------+ grandes adresses 0x7fffffffd3c0
67      ; rem. : le contenu initial de rsp peut varier
68
69      ; à partir d'ici : utilisation possible des variables locales
70
71      ; préparation de rsi :
72      ; 2e argument de read / write :
73      ; pointe sur le buffer où read écrit le résultat de la lecture
74      ; pointe sur le buffer où write récupère la donnée à écrire
75      mov     rsi, rbp        ; adresse du byte où stocker / récupérer
76      dec     rsi             ;  le résultat de la lecture
77      ; lea     rsi, [rbp - 1]  ; alternative aux 2 lignes avant
78      ; rem. : on utilise rbp car il pourrait y avoir eu des push / pop
```

```
79      ;         => utiliser rsp est en général plus compliqué que rbp qui
80      ;            _ne change pas de valeur_ tout au long de la fonction
81      ;
82      ;             |      | petites adresses
83      ; rsp --->   | 0x?? | 0x7fffffffd3b7  <--- rsi (pointeur)
84      ; rbp --->   | 0x00 | 0x7fffffffd3b8
85      ;             | 0x00 |
86      ;             | 0x00 |
87      ;             | 0x00 |
88      ;             | 0x00 |
89      ;             | 0x00 |
90      ;             | 0x00 |
91      ;             | 0x00 |
92      ;             +------+ grandes adresses 0x7fffffffd3c0
93      ; rem. : le contenu initial de rsp peut varier
94
95      ; préparation de rdx :
96      ; 3e argument de read / write :
97      ; nombre de bytes à lire (read) / à écrire (write)
98      mov     rdx, 1          ; nombre de bytes à lire / écrire
99
100 boucle:
101     ; lecture d'un byte sur stdin
102     mov     rax, 0          ; numéro de l'appel système : read
103     mov     rdi, 0          ; 1er argument : file descriptor : stdin
104     ; rsi et rdx sont prêts
105     syscall
106
107     cmp     rax, 1
108     jnz     fin_boucle      ; rax == 1 => arrêt si pas 1 byte lu
109
110     ; écriture d'un byte sur stdout
111     mov     rax, 1          ; numéro de l'appel système : write
112     mov     rdi, 1          ; 1er argument : file descriptor : stdout
113     ; rsi et rdx sont prêts
114     syscall
115
116     jmp     boucle
117
118 fin_boucle:
119
120     ; à partir d'ici : fin de l'utilisation des variables locales
121
122     ; destruction des variables locales
```

```
123     mov     rsp, rbp
124     ;
125     ;           |      | petites adresses
126     ;           | 0x?? | 0x7fffffffd3b7 : variable locale détruite
127     ; rsp --->  | 0x00 | 0x7fffffffd3b8  <--- rbp
128     ;           | 0x00 |
129     ;           | 0x00 |
130     ;           | 0x00 |
131     ;           | 0x00 |
132     ;           | 0x00 |
133     ;           | 0x00 |
134     ;           | 0x00 |
135     ;           +------+ grandes adresses 0x7fffffffd3c0
136     ; rem. : le contenu initial de rsp peut varier
137
138     ; restauration de la valeur initiale de rbp
139     ; rem. : c'est inutile ici car on est dans _start
140     pop     rbp     ; récupération de la valeur sauvegardée de rbp
141     ; le contenu initial de rbp est restauré depuis la pile
142     ;   rsp <-- rsp + 8
143     ;
144     ;           |      | petites adresses
145     ;           | 0x?? | 0x7fffffffd3b7 : variable locale détruite
146     ;           | 0x00 | 0x7fffffffd3b8
147     ;           | 0x00 |
148     ;           | 0x00 |
149     ;           | 0x00 |
150     ;           | 0x00 |
151     ;           | 0x00 |
152     ;           | 0x00 |
153     ;           | 0x00 |
154     ; rsp --->  +------+ grandes adresses 0x7fffffffd3c0
155     ; rem. : le contenu initial de rsp peut varier
156
157 fin:
158     mov     rax, 60
159     mov     rdi, 0
160     syscall
```

Rappelons-le, les parties relatives à la sauvegarde et à la restauration du pointeur de contexte de pile sont développées et expliquées lors du TD08.

L'instruction lea[29] apparaît en commentaire. On peut s'en passer. Cependant, son

29. https://www.felixcloutier.com/x86/lea (consulté le 7 avril 2020).

usage simplifie la vie. Que ceux d'entre vous qui le désirent se renseignent à son propos !

# 6 Exercices

Pour réaliser les exercices qui suivent, vous ne pouvez utiliser que les instructions étudiées au long des TD précédents ainsi que celui-ci.

**Ex. 1** Écrivez un code qui échange le contenu des registres `rax` et `rbx` en utilisant *exclusivement* la pile. Par « *exclusivement* », on entend que *seuls* les deux registres, `rax` et `rbx`, et la pile interviennent lors de l'échange des contenus.

Il est possible de répondre en utilisant *une seule* variable locale temporaire automatique sur la pile ou *deux*. Produisez les deux solutions. Dans chaque cas, fournissez l'équivalent en langage C [30] de la partie du code spécifique à l'échange des contenus des registres, en associant au registre `rax` une variable nommée `a` et au registre `rbx` une variable nommée `b`.

*Rem.* : utiliser la pile alors que des registres sont disponibles n'est pas une bonne idée.

**Ex. 2** Écrivez un code qui échange, en utilisant *exclusivement* la pile, le contenu de deux variables de taille 8 *bytes* déclarées dans la `section .data`. Par « *exclusivement* », on entend que *seules* les deux variables globales et la pile interviennent lors de l'échange des contenus.

Il est possible de répondre en utilisant *une seule* variable locale temporaire automatique sur la pile ou *deux*. Produisez les deux solutions. Dans chaque cas, fournissez l'équivalent dans votre langage de programmation de haut niveau préféré de la partie du code spécifique à l'échange des contenus des variables.

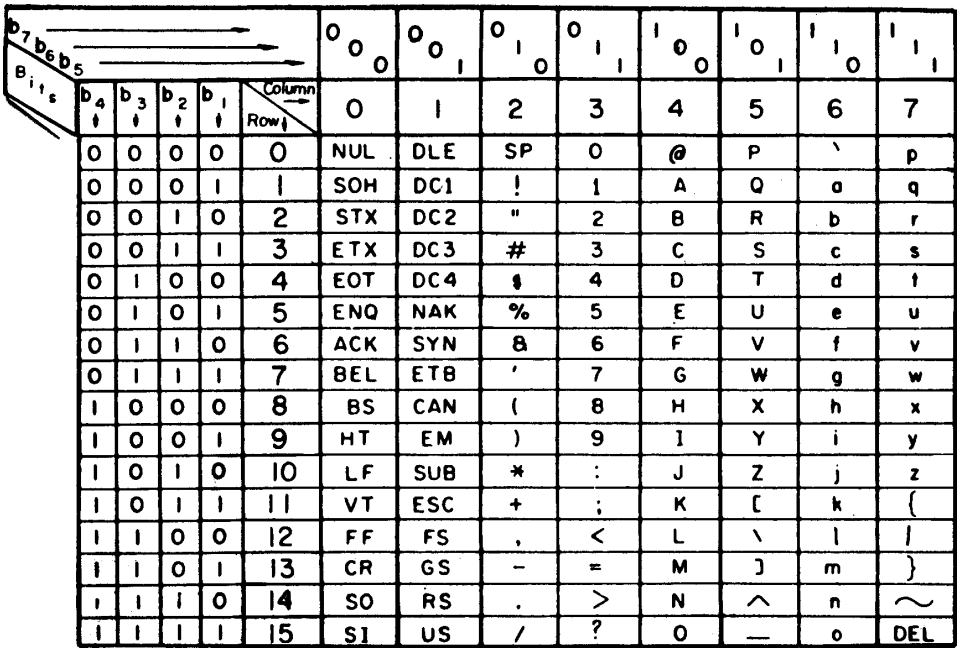
*Rem.* : utiliser la pile alors que des registres sont disponibles n'est pas une bonne idée.

**Ex. 3** Écrivez un code qui crée une variable locale sur la pile de taille un *byte*, initialisée à une valeur comprise entre 0 et 9, ces valeurs incluses. Le programme transforme le chiffre binaire en caractère textuel puis l'affiche à l'écran. N'hésitez pas à consulter la table ASCII de la Fig. 1.

Aucune variable globale n'est autorisée.

30. Ou en Java ou dans le langage de programmation de haut niveau de votre choix.

**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 |
|---|---|---|---|---|---|---|---|---|
| $b_4$ $b_3$ $b_2$ $b_1$ Column → / Row ↓ | 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 8 avril 2020).

**Ex. 4** Même exercice que l'Ex. 3, mais il faut maintenant créer dix *bytes* sur la pile. Le premier est initialisé à la valeur 0, le deuxième à 1, etc. Le dixième et dernier est donc mis à l'entier 9. Le programme transforme tous ces chiffres en caractères puis les affiche un par un à l'écran en les séparant par un espace puis termine l'affichage par un passage à la ligne. Pour l'espace et le passage à la ligne, il faut également recourir à des variables locales.

Aucune variable globale n'est autorisée.

**Ex. 5** Écrivez un code qui crée sur la pile une variable locale de taille 8 *bytes* et met :

- son bit de rang 0 :
  - à 0 si le contenu de `rdi` est impair ;
  - à 1 si le contenu de `rdi` pair ;
- son bit de rang 63 :
  - à 0 si `rdi` contient un nombre impair de bits à 1 ;
  - à 1 si `rdi` contient un nombre pair de bits à 1 ;

— ses bits de rang 1 à 62 à 0.

Le contenu de `rdi` doit être préservé.

**Ex. 6** Écrivez un code qui :

1. tente d'ouvrir en lecture seule le fichier `loremipsum.txt` [31] [32] :
   — si l'ouverture échoue un message d'erreur est affiché et le programme s'arrête ;
   — si l'ouverture réussit, le programme affiche un message indiquant cette réussite et passe à la suite ;
2. lit le contenu du fichier *byte* par *byte* et compte le nombre de majuscules, le nombre de minuscules et celui de caractères qui ne sont ni l'une ni l'autre, présents au sein du fichier ;
3. ferme le fichier et termine proprement.

Le fichier `loremipsum.txt` ne contient que des caractères de la table table ASCII (voir la Fig. 1) et donc en particulier *aucun* caractère accentué. On qualifie de :

— *majuscule* : tout caractère de code compris entre `'A'` et `'Z'`, ces valeurs incluses ;

— *minuscule* : tout caractère de code compris entre `'a'` et `'z'`, ces valeurs incluses.

Pour les décomptes, vous devez utiliser trois variables locales résidant sur la pile.

Les seules variables globales autorisées sont les trois chaînes de caractères non modifiables qui servent à stocker le nom du fichier et les deux messages à afficher.

*Rem.* : on voit au TD08 comment afficher sur la sortie standard des données numériques.

## Notions à retenir

Pile associée à un processus ; registres `rsp` et `rbp` ; instructions `push`, `pop`, `add` et `sub` ; création, manipulation et destruction de variables locales sur la pile.

## 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.

31. https://poesi.esi-bru.be/pluginfile.php/5768/mod_folder/content/0/code/loremipsum.txt (consulté le 13 avril 2020).
32. Généré depuis ici : https://fr.lipsum.com/ (consulté le 13 avril 2020).

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