HE2B ESI

MICL : TD08 : Fonctions

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

Année académique 2021 – 2022

Dans ce TD, le cœur de l'approche procédurale, à savoir la définition de fonctions munies éventuellement de paramètres et retournant pour certaines une valeur, est étudié.

Dans les TD précédents, diverses mnémoniques de rupture du flot séquentiel d'exécution des instructions d'un programme ont été introduites, telles `jmp` [1], `jz` [2] ou `syscall` [3]. Les deux premières mnémoniques données en exemple correspondent à des instructions de saut, la troisième permet d'invoquer une routine d'interruption. Une des particularités de l'instruction `syscall`, en mode protégé, est de réaliser le passage en *ring* 0 [4] de sorte à permettre l'exécution de bouts de code système.

La matière de ce dernier TD est l'étude des instructions permettant l'invocation (et le retour) de routines *utilisateur*, c'est-à-dire *sans* changement de privilège d'exécution. Par opposition aux *appels système* on pourrait parler d'*appels utilisateur*. En d'autres termes, nous allons voir comment invoquer une routine [5] dans un code écrit en langage d'assemblage et, dans le code de celle-ci, comment s'arranger pour revenir au code qui l'a appelée. Les techniques de transmission de données vers ou depuis une fonction sont également étudiées.

La découpe d'un problème en fonctions permet l'étalement de son code source [6] sur plusieurs fichiers source. Chaque fichier est assemblé [7] séparément. L'éditeur de lien [8] (*linker*)

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

1. https://github.com/HJLebbink/asm-dude/wiki/jmp (consulté le 9 avril 2020).
2. https://github.com/HJLebbink/asm-dude/wiki/jcc (consulté le 9 avril 2020).
3. https://github.com/HJLebbink/asm-dude/wiki/syscall (consulté le 9 avril 2020).
4. https://manybutfinite.com/post/cpu-rings-privilege-and-protection/ (consulté le 9 avril 2020).
5. https://fr.wikipedia.org/wiki/Routine_(informatique) (consulté le 9 avril 2020).
6. https://fr.wikipedia.org/wiki/Code_source (consulté le 9 avril 2020).
7. https://fr.wikipedia.org/wiki/Programme_assembleur (consulté le 9 avril 2020).
8. https://fr.wikipedia.org/wiki/%C3%89dition_de_liens (consulté le 9 avril 2020).

fait le lien entre appel de fonction et fonction compilée. Cela permet, par exemple, de produire des bibliothèques de fonctions [9] réutilisables dans divers programmes.

1. Fonction sans paramètre ni retour

Commençons avec le cas de figure le plus simple, celui d'une fonction sans argument ni valeur de retour.

1.1. Langage C

Pour clarifier le propos, voyons à quoi cela correspond en langage C [10] [11] :

```c
// 01_f_noarg_noret_c.c
//
// gcc -o 01_f_noarg_noret_c 01_f_noarg_noret_c.c

#include <unistd.h>

void show_message(void);

int main(void)
{
    show_message();
    _exit(0);   // http://man7.org/linux/man-pages/man2/_exit.2.html
}

void show_message(void)
{
    write(STDOUT_FILENO,
          "Salut tou·te·s !\n",
          sizeof("Salut tou·te·s !\n") - 1);
    // http://man7.org/linux/man-pages/man2/write.2.html
    // http://man7.org/linux/man-pages/man3/stdout.3.html
    // https://en.cppreference.com/w/c/language/sizeof
}
```

Dans ce programme, la fonction `show_message()` affiche la chaîne de caractères :

```
Salut tou·te·s !
```

9. https://fr.wikipedia.org/wiki/Biblioth%C3%A8que_logicielle (consulté le 9 avril 2020).
10. http://www.open-std.org/jtc1/sc22/wg14/ (consulté le 9 avril 2020).
11. Certes, l'étude du langage C n'apparaît pas dans le bloc 1 du cursus de l'ESI, mais nous avons déjà rencontré ce langage dans le TD consacré aux appels systèmes puis dans celui dédié à la pile et aux variables locales, et la syntaxe de Java s'en inspire grandement.

suivie d'un passage à la ligne. Cette fonction n'a aucun argument et ne retourne aucune valeur.

Pour générer l'exécutable correspondant, utilisez la commande :

```
$ gcc -o 01_f_noarg_noret_c 01_f_noarg_noret_c.c
```

Cela invoque le compilateur C puis l'éditeur de liens.

1.2. Langage d'assemblage

Un équivalent en assembleur est donné par le source [12] suivant :

```
1  ; 01_f_noarg_noret_asm.asm
2  ;
3  ; nasm -f elf64 -F dwarf 01_f_noarg_noret_asm.asm
4  ; ld -o 01_f_noarg_noret_asm 01_f_noarg_noret_asm.o
5
6  global _start
7
8  section .rodata
9      message         DB  `Salut tou·te·s !\n`, 0
10     message_length  DQ  message_length - message - 1
11
12 section .text
13 _start:
14     call    show_message
15
16     mov     rax, 60
17     mov     rdi, 0
18     syscall
19
20 ;
21 ; fonction d'affichage sur la sortie standard
22 ; de la chaîne de caractères "Salut tou·te·s !\n"
23 ;
24 ; argument : aucun
25 ; retour : aucun
26 ;
27 show_message:
28     mov     rax, 1
29     mov     rdi, 1
30     mov     rsi, message
```

12. Pour une version de ce code avec les évolutions des registres `rip` et `rsp` ainsi que celle de l'état de la pile, allez en annexe A.1 (p. 24).

```
31      mov     rdx, [message_length]
32      syscall
33
34      ret
```

La fonction `show_message` du code assembleur est *invoquée* à l'aide de la mnémonique `call` [13]. Celle-ci attend un argument : le nom de la fonction dont l'exécution est demandée, représenté, grâce à l'assembleur `nasm`, par un *label* (étiquette).

La *fin* de la fonction `show_message` y est indiquée par la mnémonique `ret` [14], là où une accolade fermante (`}`) est utilisée dans le source en langage C de la section 1.1. Idéalement, cette instruction modifie le pointeur d'instruction `rip` de sorte à retourner l'exécution du programme à la première instruction qui suit directement la dernière instruction `call` exécutée. Ici, cela correspond à l'instruction `mov rax, 60`.

1.2.1. `call` et `ret`

Comment le retour de la fonction se produit-il *au bon endroit* ?

L'instruction `ret` modifie le contenu de `rip` en y copiant le contenu du *quad word* (8 octets) qui se trouve au sommet de la pile et qu'elle dépile. La pile [15] est donc utilisée par `ret` pour récupérer l'adresse de retour.

L'instruction `call` quant à elle, outre le branchement vers la procédure qu'elle a en argument, sauvegarde l'adresse de retour sur la pile. Cela se fait en empilant le contenu de `rip` lors de l'exécution du `call`, c'est-à-dire l'adresse de l'instruction juste à la suite de l'appel de fonction.

1.2.2. Remarques supplémentaires

Variables globales Remarquez que les variables `message` et `message_length` sont accessibles dans les deux fonctions, `main` et `show_message`. C'est la raison pour laquelle on les qualifie de variable *globales*.

Pile Signalons-le immédiatement : évidemment, rien n'empêche d'utiliser la pile dans une procédure appelée. Il faut alors bien veiller à ce qu'au moment du `ret`, l'adresse de retour soit bien au sommet de la pile. Sinon, après l'exécution de ce `ret`, le programme continue en se branchant au péril d'une erreur.

D'autre part, comme les variables locales vivent également généralement sur la pile, il peut être possible de modifier de manière malveillante l'adresse de retour d'une fonction par le biais d'un accès abusif à une variable locale. On parle alors d'un dépassement de tampon de pile [16].

13. https://github.com/HJLebbink/asm-dude/wiki/call (consulté le 9 avril 2020).
14. https://github.com/HJLebbink/asm-dude/wiki/ret (consulté le 9 avril 2020).
15. https://en.wikipedia.org/wiki/Call_stack (consulté le 9 avril 2020).
16. https://fr.wikipedia.org/wiki/D%C3%A9passement_de_tampon#D.C3.A9tails_techniques_sur_architecture_x86_.28Intel.29 (consulté le 9 avril 2020).

Appels système Les appels systèmes consistent, comme les fonctions utilisateur, en l'exécution d'une routine tierce. L'instruction `syscall`, contrairement à l'instruction `call`, n'utilise pas la pile, mais le registre `rcx` pour sauvegarder l'adresse de retour. Cela a été détaillé dans le TD05.

Ce qui n'a pas été signalé dans ce TD, c'est que l'instruction réciproque de `syscall`, comme `ret` l'est à `call`, est l'instruction `sysret` [17].

Par ailleurs, notez que dans les architectures 16 et 32 bits d'Intel, précédant l'actuelle architecture 64 bits, les appels système étaient réalisés avec la pile comme emplacement de sauvegarde de l'adresse de retour. On utilisait alors le couple d'instructions `int` [18] / `iret` [19] pour invoquer / revenir d'un appel système, et plus particulièrement `int 0x80` [20] sous GNU / Linux. Attention, ces conventions d'appels sont obsolètes en 64 bits.

Correspondance exacte C / assembleur Il est possible de connaître la forme exacte que prend en langage d'assemblage le code initial en langage C de la section 1.1.

Une première manière [21] d'y parvenir est de demander au compilateur `gcc` de compiler le code sans l'assembler et de fournir le résultat dans le dialecte `intel` :

```
$ gcc -S -masm=intel 01_f_noarg_noret_c.c
```

Le résultat de cette commande est le fichier `01_f_noarg_noret_c.s`.

Un second moyen [22], qui permet éventuellement de s'affranchir de l'utilisation du compilateur `gcc`, consiste en l'utilisation du programme `objdump` [23]. Cela donne quelque chose comme :

```
$ gcc -c -g 01_f_noarg_noret_c.c
$ objdump -d -M intel -S 01_f_noarg_noret_c.o
```

Le résultat s'affiche sur la console.

Enfin, grâce au compilateur Compiler Explorer [24], il est possible de réaliser cette correspondance en ligne [25], avec d'autres langages que le langage C comme source possible et un large choix d'architectures cibles via divers compilateurs.

17. https://github.com/HJLebbink/asm-dude/wiki/sysret (consulté le 9 avril 2020).
18. https://www.felixcloutier.com/x86/intn:into:int3:int1 (consulté le 9 avril 2020).
19. https://www.felixcloutier.com/x86/iret:iretd (consulté le 9 avril 2020).
20. https://www.tutorialspoint.com/assembly_programming/assembly_system_calls.htm (consulté le 9 avril 2020).
21. https://stackoverflow.com/a/5638826 (consulté le 9 avril 2020).
22. https://stackoverflow.com/a/1289907 (consulté le 9 avril 2020).
23. https://sourceware.org/binutils/docs-2.32/binutils/objdump.html#objdump (consulté le 9 avril 2020).
24. https://github.com/mattgodbolt/compiler-explorer (consulté le 9 avril 2020).
25. https://godbolt.org/ (consulté le 9 avril 2020).

2. Fonction avec paramètres mais sans retour

2.1. Paramètres

Dès qu'on aborde la problématique des fonctions avec paramètres, une série de questions se posent. Par exemple : *Comment transmettre les arguments ?* ou *Dans quel ordre transmettre les arguments ?* Il n'y a pas de réponse unique à ces questions, mais différentes conventions d'appel de fonction [26] peuvent être adoptées.

Nous nous intéressons ici à certaines d'entre elles. Ce sont celles qui concernent le système d'exploitation GNU / Linux dans sa version 64 bits. Elles sont consignées dans un document intitulé System V Application Binary Interface, AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models) [27]. Dans la suite de ce TD, on utilise l'abréviation « ELF x86-64 psABI » pour faire référence à ce document. Ses sources sont disponibles dans un dépôt GitHub [28].

Deux canaux s'offrent pour la transmission des arguments : les registres et la pile.

Appels système Nous avons déjà vu que les registres seuls sont utilisés par les appels systèmes [29].

GNU / Linux 32 bits Sous GNU / Linux en 32 bits, la pile est (presque) exclusivement utilisée pour passer les arguments de fonctions utilisateur [30].

L'utilisation de la pile présente comme avantages, entre autres, de ne pas limiter le nombre de paramètres de la fonction et de permettre les appels récursifs [31]. Notez cependant que recourir à la pile présente également des inconvénients. L'exécution est plus lente, puisque la pile est en mémoire centrale. De plus, certaines menaces guettent, tels le dépassement de pile [32], si trop d'appels récursifs sont réalisés, et le *stack buffer overflow* [33], évoqué dans la section 1.2.2.

GNU / Linux 64 bits Dans la version 64 bits de GNU / Linux, les registres *et* la pile sont utilisés pour la transmission d'arguments aux fonctions utilisateur. Les conventions relatives au passage d'arguments de fonctions sont assez complexes [34].

---

26. https://en.wikipedia.org/wiki/X86_calling_conventions#List_of_x86_calling_conventions (consulté le 9 avril 2020).
27. https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf (consulté le 9 avril 2020).
28. https://github.com/hjl-tools/x86-psABI (consulté le 9 avril 2020).
29. Ceci est renseigné dans l'appendice A.2.1 du ELF x86-64 psABI.
30. On le trouve décrit dans la section 2.2.3 du System V Application Binary Interface, Intel386 Architecture Processor Supplement, Version 1.1 (https://github.com/hjl-tools/x86-psABI/wiki/intel386-psABI-1.1.pdf, consulté le 9 avril 2020)).
31. https://fr.wikipedia.org/wiki/Algorithme_r%C3%A9cursif (consulté le 9 avril 2020).
32. https://fr.wikipedia.org/wiki/D%C3%A9passement_de_pile (consulté le 9 avril 2020).
33. https://en.wikipedia.org/wiki/Stack_buffer_overflow (consulté le 9 avril 2020).
34. On en trouve la description complète dans la section 3.2.3 du ELF x86-64 psABI.

	Appel système	Fonction utilisateur
Mnémonique d'appel	`syscall`	`call`
Identifiant de l'appelé	numéro : `mov rax, num`	étiquette : `call label`
Stockage adresse retour	`rcx`	pile
Mnémonique de retour	`sysret`	`ret`
Stockage valeur retour	`rax`	entiers / adresses : `rax`, `rdx`
Arguments (dans l'ordre)	`rdi`, `rsi`, `rdx`, `r10`, `r8`, `r9`	entiers / adresses : `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9` puis la pile
Registres de l'appelant (sauvegardés [a] par l'appelé)	`rbp`, `rsp`, `rbx`, `rdx`, `rsi`, `rdi`, `r8`, `r9`, `r10`, `r12`, `r13`, `r14`, `r15`	`rbp`, `rsp` [b], `rbx`, `r12`, `r13`, `r14`, `r15`
Registres de l'appelé (sauvegardés [a] par l'appelant)	`rax`, `rcx`, `r11`	`rax`, `rcx`, `rdx`, `rsi`, `rdi`, `r8`, `r9`, `r10`, `r11`

Prologue (sauvegarde / création contextes de pile, création variables locales)	*hors sujet*	`push rbp`
		`mov rbp, rsp`
		`sub rsp, imm`
		ou
		`enter imm, 0`

Épilogue (destruction variables locales, restauration contexte de pile)	*hors sujet*	`mov rsp, rbp`
		`pop rbp`
		ou
		`leave`

TABLE 1 – Appels systèmes et fonctions utilisateurs sous GNU / Linux 64 bits (voir ELF x86-64 psABI [c]).

a. Si nécessaire.

b. En fait `rsp` ne doit pas être sauvegardé par l'appelant car le mécanisme `call` / `ret` assure la conservation de sa valeur.

c. https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf (consulté le 9 avril 2020).

Dans la suite de ce TD, *on se limite à des arguments entiers (ou adresse) en nombre inférieur ou égal à six*. Dans ce cas, seuls les registres sont employés. Du premier au sixième paramètre [35], les registres `rdi`, `rsi`, `rdx`, `rcx`, `r8` et `r9`, dans cet ordre, sont utilisés.

On retrouve cette information dans la TABLE 1 qui compare les conventions d'appels système et celles des appels de fonctions utilisateurs en GNU / Linux 64 bits.

On y remarque, par exemple, que les registres utilisés pour fournir les arguments d'un appel système ou d'une fonction utilisateur ne sont pas exactement les mêmes. Pour les appels système, le 4^{e} argument est fourni via `r10` tandis que pour les fonctions utilisateur on utilise `rcx`.

35. En langage C (ou en Java), le premier paramètre d'une fonction (ou d'une méthode) est celui qui est le plus proche du nom de cette fonction (ou de cette méthode).

2.2. Registres conservés ou non

Parmi les conventions d'appels de fonctions du ELF x86-64 psABI figurent également deux listes de registres : ceux qui appartiennent au code appelant et ceux qui appartiennent au code appelé [36].

Un registre appartenant au code appelant est un registre dont le contenu doit être préservé par la fonction appelée. Si celle-ci utilise un de ces registres, elle doit le remettre en fin d'exécution dans l'état dans lequel elle l'a trouvé en début d'exécution. Ces registres sont : `rbp`, `rbx`, `r12`, `r13`, `r14` et `r15`.

Un registre appartenant au code appelé est un registre qui ne doit pas être préservé par la fonction appelée. Il s'agit des dix registres : `rsp`, `rax`, `rcx`, `rdx`, `rdi`, `rsi`, `r8`, `r9`, `r10` et `r11`. On remarque que les registres qui servent à la transmission des paramètres de fonction appartiennent à la fonction appelée.

On retrouve cette information dans la Table 1.

2.3. Langage C

Nous nous limitons provisoirement à une fonction à un seul argument :

```c
// 02_f_arg_noret_c.c
//
// gcc -o 02_f_arg_noret_c 02_f_arg_noret_c.c

#include <unistd.h>

void show(const char * message);

// variables globales :
// https://fr.wikipedia.org/wiki/Variable_globale
const char * msg_0 = "La vie moderne\n";
const char * msg_1 = "Une qualification olympique\n";

int main(void)
{
    show(msg_0);

    show(msg_1);

    _exit(0);
}

void show(const char * message)
{
```

36. Ces listes sont fournies à la section 3.2.1 du ELF x86-64 psABI.

```
25      // séquence d'échappement en octal :
26      // voir 1er paragraphe de la section « Notes » de :
27      // https://en.cppreference.com/w/cpp/language/escape
28      while (*message != '\0')
29      {
30          write(STDOUT_FILENO, message, 1);
31          ++message;
32          // arithmétique des pointeurs :
33          // http://casteyde.christian.free.fr/cpp/cours/online/x1865.html
34      }
35  }
```

La fonction `show()` affiche sur la console une chaîne de caractères zéro-terminée. Son argument est l'adresse du premier caractère de cette chaîne.

2.4. Langage d'assemblage

Un code[37] en langage d'assemblage produisant un résultat équivalent est :

```
1   ; 02_f_arg_noret_asm.asm
2   ;
3   ; nasm -f elf64 -F dwarf 02_f_arg_noret_asm.asm
4   ; ld -o 02_f_arg_noret_asm 02_f_arg_noret_asm.o
5
6   global _start
7
8   section .rodata
9       msg_0   DB      `La vie moderne\n`, 0
10      msg_1   DB      `Une qualification olympique\n`, 0
11
12  section .text
13  _start:
14      mov     rdi, msg_0
15      call    show
16
17      mov     rdi, msg_1
18      call    show
19
20      mov     rax, 60
21      mov     rdi, 0
22      syscall
23
```

37. Pour une version de ce code avec les évolutions des registres `rip` et `rsp` ainsi que celle de l'état de la pile, allez en annexe A.2 (p. 26).

```
24  ;
25  ; fonction d'affichage sur la sortie standard
26  ; d'une chaîne de caractères constante zéro-terminée
27  ; fournie en paramètre
28  ;
29  ; argument 1 : adresse du premier caractère de la chaîne à afficher
30  ; retour : aucun
31  ;
32  show:
33      push    rbp         ; sauvegarde du contexte de pile de l'appelant
34      mov     rbp, rsp    ; création du contexte de pile de l'appelé
35      ; rem. : les 2 lignes ci-dessus sont ici inutiles car l'appelé
36      ;         (c.-à-d. la fonction show) ne crée pas de variable locale
37      ;         et donc n'a pas besoin d'un contexte de pile propre
38
39      mov     rsi, rdi    ; 1er argument call -> 2e argument syscall
40      mov     rdi, 1      ; STDOUT_FILENO
41      mov     rdx, 1      ; 1 byte à écrire
42
43   .while:
44      cmp     byte [rsi], 0   ; fin de chaîne ?
45      jz      .end
46      mov     rax, 1      ; write
47      syscall
48      inc     rsi         ; passage au caractère suivant
49      jmp     .while
50
51   .end:
52
53      mov     rsp, rbp    ; destruction des variables locales
54      ; rem. : la ligne qui précède n'est pas nécessaire car show
55      ;         n'a pas créé de variable locale
56
57      pop     rbp         ; restauration du contexte de pile de l'appelant
58      ; rem. : la ligne qui précède est indispensable car show s'est
59      ;         créé un contexte de pile propre
60
61      ret
```

Paramètre L'appel de la fonction `show`, par le biais de `call`, est précédé par la copie dans le registre `rdi` de son argument, à savoir l'adresse du premier caractère de la chaîne de caractères à afficher. C'est conforme à la convention de transmission de paramètres à une fonction utilisateur décrite dans la Table 1.

Contexte de pile (stack frame) Au sein d'une fonction, l'accès à ses variables locales (et à ses arguments transmis par la pile), se fait via le registre `rbp` plutôt que `rsp`. Le contenu de ce dernier registre ne cesse en effet de varier au gré, entre autres, des `call`, `push` ou `pop`. Il est plus simple de figer l'accès à la pile en fixant une fois pour toute le point d'accès vers la pile en tout début du code de la fonction. Cela s'appelle la création du contexte de pile [38] de la fonction appelée : l'instruction `mov rbp, rsp` fait ainsi bel et bien pointer `rbp` sur la base [39] de la pile au sein de la fonction. Ainsi le registre `rbp` joue le rôle de pointeur de contexte de pile (*stack frame pointer*) et chaque fonction accède à une portion — un contexte — de pile qui lui est propre.

La création du contexte de pile ne peut cependant pas être la toute première opération [40] réalisée par la fonction. Conformément à l'information présente dans la TABLE 1, le registre `rbp` appartient au code appelant. Comme la création du contexte de pile du code appelé modifie le contenu de ce registre, le code appelé doit préalablement sauvegarder sa valeur et être en mesure de la restaurer en fin d'exécution. Cela explique la présence de l'instruction `push rbp` comme toute première dans la fonction `show` ainsi que celle de l'instruction `pop rbp` juste avant le retour au code appelant (`ret`).

Notez qu'ici précisément, il n'est pas réellement nécessaire de sauvegarder ou créer un contexte de pile car `show` ne reçoit pas d'argument via la pile et ne crée aucune variable locale.

2.4.1. Remarques supplémentaires

Variables locales L'antépénultième [41] instruction de la fonction `show` est l'instruction `mov rsp, rbp`. Elle sert à détruire les variables locales créées sur la pile. Cela a été détaillé dans le TD précédent.

Notez qu'ici précisément ce n'est pas réellement nécessaire car aucune variable locale n'est créée dans `show`.

Prologue et épilogue de fonction utilisateur Dans la section 3.2, on voit qu'il existe des instructions dédiées facilitant la gestion des *stack frames* et des variables locales. C'est déjà signalé dans la TABLE 1.

Étiquette locale Les étiquettes qui commencent par un point (`.`) sont considérées par `nasm` comme *locales* [42]. À la différence des étiquettes *globales*, c'est-à-dire de celles qui ne commencent pas par un point, il est possible de définir plusieurs étiquettes locales de *même nom* dans un source donné. Chaque étiquette locale est associée à la dernière

38. https://en.wikibooks.org/wiki/X86_Disassembly/Functions_and_Stack_Frames (consulté le 9 avril 2020).
39. D'ailleurs `rbp` vaut pour *register* base *pointer*.
40. https://en.wikipedia.org/wiki/Function_prologue#Prologue (consulté le 9 avril 2020).
41. http://www.cnrtl.fr/definition/ant%C3%A9p%C3%A9nulti%C3%A8me Antépénultième : Qui précède immédiatement l'avant-dernière (la pénultième) unité, dans une suite d'éléments qu'on peut dénombrer (consulté le 9 avril 2020).
42. https://www.nasm.us/doc/nasmdoc3.html#section-3.9 (consulté le 9 avril 2020).

étiquette globale définie, en remontant le code source. Ceci permet, dans des fonctions différentes d'un même code, associées donc à des étiquettes globales différentes, de définir des étiquettes locales de même nom, comme `.end`, par exemple.

Les étiquettes locales ont un *nom court*, c'est leur nom commençant par un point. Elles ont aussi un *nom long*, obtenu en concaténant leur étiquette globale avec leur nom court. Sous une étiquette globale, on accède à *ses* étiquettes locales par le biais de leurs noms courts. Pour accéder à une étiquette locale assujettie à une *autre label* global que le courant, on utilise le nom long de cette étiquette.

Le source ci-dessous donne des exemples de définitions et d'utilisations de *labels* locaux, à travers un horrible code spaghetti [43] :

```
; local_label.asm
;
; nasm -f elf64 -F dwarf local_label.asm
; ld -o local_label local_label.o

global _start

section .text
_start:

global1:
    jmp     .end            ; 1re instruction exécutée

    ; .ici ci-dessous associée à global1 :
    ;   nom complet : global1.ici
 .ici:
    jmp     global3.end     ; 3e instruction exécutée

    ; .end ci-dessous associée à global1 :
    ;   nom complet : global1.end
 .end:
    jmp     .ici            ; 2e instruction exécutée

global2:
    ; .end ci-dessous associée à global2 :
    ;   nom complet : global2.end
 .end:
    mov     rax, 60         ; 6e instruction exécutée
    mov     rdi, 0          ; 7e instruction exécutée
    syscall                 ; 8e instruction exécutée

global3:
```

43. https://fr.wikipedia.org/wiki/Programmation_spaghetti (consulté le 13 avril 2020).

```
33      ; .ici ci-dessous associée à global3 :
34      ;   nom complet : global3.ici
35  .ici:
36      jmp     global2.end     ; 5e instruction exécutée
37
38      ; .end ci-dessous associée à global3 :
39      ;   nom complet : global3.end
40  .end:
41      jmp     .ici            ; 4e instruction exécutée
```

Convention d'utilisation label global / label local Dans ce TD, on utilise des étiquettes globales pour identifier des noms de fonctions et des étiquettes locales pour identifier des instructions au sein d'une fonction. On adopte la bonne pratique de ne jamais se brancher sur une étiquette locale d'une fonction à partir d'une autre fonction !

3. Fonction avec paramètres, retour et variables locales

3.1. Retour

Pour ce qui concerne le retour de fonction, divers choix [44] s'offrent : retourner aucune, une ou plusieurs valeurs et ce par le biais d'un ou de registres ou via la pile.

Comme nous utilisons un GNU / Linux 64 bits, nous persistons à respecter les conventions du ELF x86-64 psABI. Elle limite le nombre de valeurs entières (ou adresse) retournées à maximum deux [45]. Les registres `rax` puis `rdx` sont utilisés. Cette information se retrouve dans la Table 1 (p. 7).

3.2. Variables locales

Un TD a été consacré aux variables locales.

On y a vu que les registres pouvaient être utilisés pour les héberger, mais que si ceux-ci étaient insuffisants en nombre, par exemple si la variable locale est un tableau, la pile était mise à contribution.

En cas d'utilisation de la pile, les accès aux variables locales se font via `rbp`. Cela a été rappelé en fin de section 2.4 où la notion de *contexte de pile* est expliquée. Suite à la gestion des contextes de pile de l'appelant — `push rbp` — et de l'appelé — `mov rbp, rsp` —, les variables locales peuvent être créées. Cela peut se faire à l'aide de `push` ou, si on désire créer les variables locales en masse, en produisant un trou de `hole_size` *bytes* sur la pile avec `sub rsp, hole_size`.

Symétriquement, en fin d'exécution de l'appelé, il faut détruire les variables locales avec `mov rsp, rbp` et restaurer le contexte de pile de l'appelant — `pop rbp`.

44. https://en.wikipedia.org/wiki/Calling_convention (consulté le 9 avril 2020).
45. Ceci est renseigné dans la section 3.2.3 du ELF x86-64 psABI.

Prologue / `enter` Le prologue[46] standard d'une fonction :

```
function_name:
    push    rbp
    mov     rbp, rsp
    sub     rsp, hole_size  ; hole_size est un entier
```

peut être remplacé par l'usage de l'instruction `enter`[47] en :

```
function_name:
    enter   hole_size, 0  ; hole_size est un entier
```

Épilogue / `leave` L'épilogue[48] standard d'une fonction :

```
function_name:
    ; ...
    mov     rsp, rbp
    pop     rbp

    ret
```

peut être raccourci par l'usage de l'instruction `leave`[49] en :

```
function_name:
    ; ...
    leave

    ret
```

La TABLE 1 consacre des lignes aux prologue et épilogue des fonctions utilisateurs.

3.3. Langage d'assemblage

3.3.1. `string_asm.asm`

Voici le fichier `string_asm.asm`[50] :

```
; string_asm.asm
;
```

46. https://en.wikipedia.org/wiki/Function_prologue#Prologue (consulté le 9 avril 2020).
47. https://github.com/HJLebbink/asm-dude/wiki/enter (consulté le 9 avril 2020).
48. https://en.wikipedia.org/wiki/Function_prologue#Epilogue (consulté le 9 avril 2020).
49. https://github.com/HJLebbink/asm-dude/wiki/leave (consulté le 9 avril 2020).
50. Le lecteur intéressé par son équivalent en langage C le trouve en annexe B.1.

```
3  ; nasm -f elf64 -F dwarf string_asm.asm
4
5  global print_hex
6
7  section .text
8
9  ;
10 ; print_hex : affiche en hexadecimal sur la console la valeur du
11 ;              1er argument suivi d'un passage à la ligne si
12 ;              le 2e argument vaut 1
13 ;
14 ; argument 1 : entier sur 8 bytes (64 bits, 8 * 2 quartets)
15 ; argument 2 : affichage terminé par `\n` si vaut 1
16 ;
17 ; retour : void
18 ;
19 print_hex:
20     ; push    rbp
21     ; mov     rbp, rsp
22     ;
23     ; sub     rsp, 2 + 8 * 2 + 1 ; `0x??????????????\n`
24
25     enter   2 + 8 * 2 + 1, 0    ; `0x??????????????\n`
26
27     mov     byte [rbp - (2 + 8 * 2 + 1)], `0`
28     mov     byte [rbp - (2 + 8 * 2 + 1) + 1], `x`
29     mov     byte [rbp - 1], `\n`
30
31     ; conversion
32     mov     r8, 0       ; compteur : 0 → 16 (car 8 * 2 quartets)
33     mov     r9, rbp
34     sub     r9, 2       ; pointe sur la chaîne de caractères
35                         ; résultat de la conversion binaire → texte
36  .conversion:
37     cmp     r8, 8 * 2
38     jz      .newline
39
40     mov     al, 0xF
41     and     al, dil     ; conserver le quartet de rang le plus petit
42     cmp     al, 10      ; 0 → 9, a → f ?
43     ; conversion binaire → texte
44     js      .digit10
45     add     al, `a` - 10
46     jmp     .memcopy
```

```
47  .digit10:
48    add     al, `0`
49  .memcopy:
50    mov     [r9], al    ; copie du caractère dans la chaîne
51
52    shr     rdi, 4      ; décalage de rdi de 4 bits à droite
53                        ; de sorte à amener le quartet « suivant »
54                        ; en position de rang le plus petit
55
56    inc     r8          ; compteur de boucle
57    dec     r9          ; pointeur dans la chaîne de caractères
58    jmp     .conversion
59    ; rem. : ci-dessus on peut utiliser r8 et r9 car pas d'appel de
60    ;         fonction dans la boucle => leurs valeurs ne seront
61    ;         pas perdues
62
63  .newline:
64    ; newline ?
65    cmp     rsi, 1
66    jz      .print
67    mov     rsi, 0      ; forcer à 0 si pas 1
68
69  .print:
70    ; affichage
71    mov     rax, 1
72    mov     rdi, 1
73    mov     rdx, 2 + 8 * 2
74    add     rdx, rsi        ; 0 ou 1 (`\n`)
75    mov     rsi, rbp
76    sub     rsi, 2 + 8 * 2 + 1  ; début de la chaîne
77    syscall
78
79    ; mov     rsp, rbp
80    ; pop     rbp
81
82    leave
83
84    ret
```

Seule la fonction `print_hex` y est définie. Celle-ci utilise une variable locale pour stocker une chaîne de caractères. Elle ne retourne rien. La description complète de son utilité est donnée dans le bloc de commentaires qui précède son code.

Dans son implémentation, on utilise l'instruction `leave` et un masque pour isoler les

quartets[51] (*nibble*) du premier argument. Chacun d'eux est converti en caractère en lui ajoutant la valeur du code ASCII du caractère `'0'` si sa valeur est comprise entre 0 et 9 ou la valeur du code ASCII du caractère `'a'` dont on retire 10 s'il est supérieur ou égal à 10. Ce traitement est réalisé dans une structure de contrôle *si / sinon*. On passe d'un quartet au suivant en décalant le motif binaire du premier argument de 4 positions vers la droite. Pour cela, on utilise l'instruction `shr`[52] (***sh**ift **r**ight*), qui n'a pas été vue dans les TD précédents. Les 16 quartets de l'argument sont traités les uns à la suite des autres dans une boucle *pour* où le registre `r8` fait office de compteur.

Directive `global` La directive `global`[53] demande à l'assembleur `nasm` de rendre publique l'étiquette qui la suit. On a coutume de l'utiliser pour que le point d'entrée du programme, `_start`, `main`, etc., soit accessible à l'éditeur de liens. Cela a été expliqué dès le premier TD. Ici, on demande d'en faire de même pour la fonction `print_hex` pour rendre possible son utilisation depuis l'extérieur du fichier `string_asm.asm`.

3.3.2. `number_asm.asm`

Voici le fichier `number_asm.asm`[54] :

```
1  ; number_asm.asm
2  ;
3  ; nasm -f elf64 -F dwarf number_asm.asm
4
5  global multiply
6  global multiply_print
7
8  extern print_hex
9
10 section .text
11
12 ;
13 ; multiply : retourne le produit des 2 arguments considérés non signés
14 ;
15 ; rem. : la mnémonique mul fait ce boulot !
16 ;
17 ; argument 1 : entier sur 4 bytes : seuls les 4 bytes de poids le
18 ;               plus faible sont pris en compte
19 ; argument 2 : entier sur 4 bytes : seuls les 4 bytes de poids le
20 ;               plus faible sont pris en compte
21 ;
```

51. https://fr.wikipedia.org/wiki/Nibble (consulté le 9 avril 2020).
52. https://github.com/HJLebbink/asm-dude/wiki/sal_sar_shl_shr (consulté le 9 avril 2020).
53. https://www.nasm.us/doc/nasmdoc6.html#section-6.6 (consulté le 9 avril 2020).
54. Le lecteur intéressé par son équivalent en langage C le trouve en annexe B.2.

```
22 ; retour : produit des arguments : argument 1 x argument 2
23 ;          sur 8 bytes
24 ;
25 multiply:
26     enter   0, 0        ; gestion contextes de pile
27                         ;               / pas de variable locale
28
29     ; garder 4 bytes de poids faible
30     mov     rax, 0xffffffff
31     and     rdi, rax        ; car existe pas : and rdi, imm64
32     and     rsi, rax
33
34     ; calcul
35     mov     rax, 0
36
37  .loop:
38     dec     rdi
39     js      .end
40     add     rax, rsi
41     jmp     .loop
42
43     ; rem. : ce serait mieux de boucler min(rdi, rsi) fois
44
45  .end:
46     leave
47
48     ret
49
50 ;
51 ; multiply_print : calcule le produit des 2 arguments + affiche
52 ;                  l'opération et le résultat
53 ;                  attention : opérandes sur 4 bytes
54 ;
55 ; argument 1 : entier sur 4 bytes : seuls les 4 bytes de poids le
56 ;              plus faible sont pris en compte
57 ; argument 2 : entier sur 4 bytes : seuls les 4 bytes de poids le
58 ;              plus faible sont pris en compte
59 ;
60 ; retour : void
61 ;
62 multiply_print:
63     enter   6, 0
64
65     push    r12
```

```
66      push    r13
67
68      mov     byte [rbp - 6], ` `
69      mov     byte [rbp - 5], `*`
70      mov     byte [rbp - 4], ` `
71
72      mov     byte [rbp - 3], ` `
73      mov     byte [rbp - 2], `=`
74      mov     byte [rbp - 1], ` `
75
76      mov     r10, 0xffffffff
77      ; sauvegarde de rdi dans r12 (registre appartenant à appelant)
78      ; car on va appeler des fonctions et on veut conserver la valeur
79      ; si on utilisait r8 : risque de perdre sa valeur car appels de
80      ; fonctions non conservant : il faudrait sauvegarder explicitement
81      ; avant l'appel
82      mov     r12, rdi
83      and     r12, r10
84      ; sauvegarde de rsi dans r13 registre appartenant à appelant
85      ; car on va appeler des fonctions et on veut conserver la valeur
86      mov     r13, rsi
87      and     r13, r10
88
89      mov     rdi, r12
90      mov     rsi, 0
91      call    print_hex
92
93      mov     rax, 1
94      mov     rdi, 1
95      mov     rsi, rbp
96      sub     rsi, 6
97      mov     rdx, 3
98      syscall
99
100     mov     rdi, r13
101     mov     rsi, 0
102     call    print_hex
103
104     mov     rax, 1
105     mov     rdi, 1
106     mov     rsi, rbp
107     sub     rsi, 3
108     mov     rdx, 3
109     syscall
```

```
110
111     mov     rdi, r12
112     mov     rsi, r13
113     call    multiply
114
115     mov     rdi, rax
116     mov     rsi, 1
117     call    print_hex
118
119     pop     r13
120     pop     r12
121
122     leave
123
124     ret
```

Deux fonctions y sont définies : `multiply` et `multiply_print`. Elles sont toutes les deux rendues publiques.

Fonction `multiply` La fonction `multiply` calcule et retourne via le registre `rax` le produit des ses arguments. En ce sens, elle respecte les conventions du ELF x86-64 psABI. Remarquez cependant que le jeu d'instruction du processeur contient la mnémonique `mul` [55] qui réalise bien plus efficacement cette opération arithmétique !

Fonction `multiply_print` La fonction `multiply_print` calcule et affiche sur la sortie standard le produit de ses arguments. Les affichages des valeurs numériques sont réalisés par la fonction `print_hex` du source `string_asm.asm` (section 3.3.1).

La fonction `multiply_print` sauvegarde ses arguments dans les registres `r12` et `r13`. Comme il s'agit de registres appartenant au code appelant (voir TABLE 1, p. 7), les contenus de ces registres doivent être sauvegardés et restaurés en début et fin d'exécution de `multiply_print`, *mais pas* avant d'invoquer `print_hex` ni `multiply`.

Directive `extern` La directive `extern` [56] indique à l'assembleur `nasm` que l'étiquette qui la suit est définie à l'extérieur du source courant.

Comme déjà écrit, la fonction `multiply_print` utilise la fonction `print_hex`. Cette dernière n'est *pas* accessible à `nasm` lorsqu'il assemble le fichier `number_asm.asm` car elle est implémentée dans un autre fichier source : le fichier `string_asm.asm`. Pour que `nasm` n'émette pas d'erreur et dise que la symbole `print_hex` lui est inconnu, on lui indique que celui-ci vient du monde extérieur : `extern print_hex`.

Le lien entre le symbole `print_hex` utilisé dans le fichier `number_asm.asm` et son implémentation dans `string_asm.asm` ne se fait pas entre les fichiers sources. C'est

55. https://github.com/HJLebbink/asm-dude/wiki/mul (consulté le 9 avril 2020).
56. https://www.nasm.us/doc/nasmdoc6.html#section-6.5 (consulté le 9 avril 2020).

l'*éditeur de liens* (*linker*) qui s'en charge lors du traitement des fichiers objets pour produire l'exécutable. C'est d'ailleurs la raison de son nom. Les commandes à utiliser sont données dans la section 4.

3.3.3. `03_test_asm.asm`

Voici le fichier `03_test_asm.asm`[57] :

```
1  ; 03_test_asm.asm
2  ;
3  ; nasm -f elf64 -F dwarf 03_test_asm.asm
4  ; ld -o 03_test_asm 03_test_asm.o number_asm.o string_asm.o
5  ;
6  ; rem. : pour la génération du fichier :
7  ;           + number_asm.o : aller voir number_asm.asm
8  ;           + string_asm.o : aller voir string_asm.asm
9
10 global _start
11
12 extern multiply
13 extern print_hex
14 extern multiply_print
15
16 section .text
17 _start:
18     mov     rdi, 0x2
19     mov     rsi, 0x1000
20     call    multiply
21
22     mov     rdi, rax
23     mov     rsi, 1
24     call    print_hex
25
26     mov     rdi, 0xffffffff
27     mov     rsi, 0x2
28     call    multiply_print
29
30     mov     rax, 60
31     mov     rdi, 0
32     syscall
```

Ce fichier contient le point d'entrée d'un programme qui teste les fonctions `multiply`, `print_hex` et `multiply_print` dont on a parlé dans les sections 3.3.1 et 3.3.2.

57. Le lecteur intéressé par son équivalent en langage C le trouve en annexe B.3.

La production de l'exécutable correspondant est expliquée dans la section 4.

4. Production d'un exécutable

Pour obtenir un exécutable à partir d'un fichier source en langage d'assemblage, on a vu dans le TD01 qu'on commence par assembler le source puis que l'éditeur de lien produit l'exécutable.

C'est une bonne pratique de rassembler des fonctions de même genre de traitement dans un même fichier source. On peut ainsi avoir un fichier source avec des fonctions arithmétiques, un autre avec des fonctions de traitement de chaînes de caractères, etc. On doit alors utiliser des fonctions réparties dans plusieurs fichiers sources. Chaque fichier source est assemblé *séparément*. C'est ensuite l'éditeur de liens qui rassemble au sein de l'exécutable les morceaux éparpillés dans les fichiers objets.

Voici par exemple la chaîne de production[58] de l'exécutable `03_test_asm` obtenu à partir des sources de la section 3.3 :

```
$ nasm -f elf64 string_asm.asm
$ nasm -f elf64 number_asm.asm
$ nasm -f elf64 03_test_asm.asm
$ ld -o 03_test_asm -e _start string_asm.o number_asm.o 03_test_asm.o
```

Voici les commandes à utiliser si on désire un exécutable avec informations de déboguage :

```
$ nasm -f elf64 -F dwarf string_asm.asm
$ nasm -f elf64 -F dwarf number_asm.asm
$ nasm -f elf64 -F dwarf 03_test_asm.asm
$ ld -o 03_test_asm -e _start string_asm.o number_asm.o 03_test_asm.o
```

Dans les deux cas, l'ordre des fichiers objets fournis au *linker* n'a pas d'importance.

Erreurs Comme il a été signalé dans le TD01, des erreurs peuvent se produire lors de l'assemblage, lors de l'édition des liens ou lors de l'exécution.

Pour ce qui concerne les erreurs de *linker*, en particulier quand plusieurs fichiers objet interviennent pour la production de l'exécutable, signalons :

- l'utilisation dans un fichier objet d'un symbole qui n'a été défini dans aucun des fichiers objet renseignés, c'est-à-dire l'oubli d'un fichier objet dans la liste de ceux renseignés à l'éditeur de liens ;
- l'utilisation dans un fichier objet d'un symbole externe, mais qu'on a oublié de rendre `global` dans son source ;
- l'utilisation dans un fichier objet d'un symbole externe, mais qu'on a oublié de qualifier d'`extern` là où on tente de l'utiliser ;
- la définition multiple d'un même symbole `global` dans plusieurs fichiers objet.

58. L'indication du point d'entrée est facultative puisqu'il s'agit de `_start`.

5. Exercices

Pour réaliser les exercices qui suivent, vous ne pouvez utiliser que les instructions étudiées au long des TD précédents ainsi que de celui-ci. Aucune des fonctions ne peut utiliser de variable globale, sauf éventuellement pour les chaînes de caractères.

Vous devez tester chacune de vos fonctions à partir d'une fonction principale, `_start` ou `main`, écrite dans un fichier source *différent* de ceux où elles sont implémentées. N'oubliez donc pas d'utiliser les directives `global` et `extern`. Compilez chaque source séparément. Éditez les liens en spécifiant la liste des fichiers objets nécessaires à la production de l'exécutable désiré. Suivez, en l'adaptant, la chaîne de production d'un exécutable donnée en section 4.

Ex. 1 Écrivez une fonction, `affStr`, qui reçoit comme unique argument l'adresse du premier caractère d'une chaîne zéro-terminée, qui affiche cette chaîne sur la sortie standard et qui ne retourne rien. La chaîne ne peut être modifiée par `affStr`.

Ex. 2 Écrivez une fonction, `estPair`, qui reçoit comme unique argument un entier codé sur 8 octets et qui retourne 1 si l'entier est pair, 0 sinon.

Ex. 3 Écrivez une fonction, `lgrStr`, qui reçoit comme unique argument l'adresse du premier caractère d'une chaîne zéro-terminée et qui retourne la taille en octet de la chaîne, sans compter le zéro final. La chaîne ne peut être modifiée par `lgrStr`.

Ex. 4 Écrivez une fonction, `swap`, qui reçoit deux arguments : les *adresses* de deux variables (locales) de la fonction appelant. Chacune de ces deux variables fait 8 octets. La fonction permute le contenu des ces deux variables. Elle ne retourne rien.

Ex. 5 Écrivez une fonction, `nbPair`, qui reçoit deux arguments : deux entiers codés sur 8 octets. La fonction retourne 0 si aucune des valeurs n'est paire, 1 si une seule est paire, 2 si elles sont toutes les deux paires.

Aide : Utiliser la fonction `estPair` de l'Ex. 2 devrait vous faciliter le travail.

Ex. 6 Écrivez une fonction, `affBin`, qui reçoit comme unique argument un entier codé sur 8 octets et qui affiche la valeur de cet entier en binaire. Elle ne retourne rien.

Notions à retenir

Définition et invocation de fonctions avec ou sans paramètre, avec ou sans retour, avec ou sans variables locales ; chaîne de production d'un exécutable.

A. Codes assembleur fortement commentés

A.1. Fonction sans paramètre ni retour

Voici le code de la section 1.2 (p. 3) enrichi de commentaires où les évolutions des registres `rip` et `rsp` ainsi que celle de l'état de la pile sont renseignées :

```
1  ; 01_f_noarg_noret_asm_fc.asm
2  ;
3  ; nasm -f elf64 -F dwarf 01_f_noarg_noret_asm_fc.asm
4  ; ld -o 01_f_noarg_noret_asm_fc 01_f_noarg_noret_asm_fc.o
5
6  global _start
7
8  section .rodata
9      message         DB  `Salut tou·te·s !\n`, 0
10     message_length  DQ  message_length - message - 1
11
12 section .text
13 ;
14 ; point d'entrée
15 ;
16 _start:
17
18     ; rip : 0x401000
19     ; rsp : 0x7fffffffd3a0
20     ;
21     ; pile vide
22     ;                       petites adresses
```

```
23      ;           |      |
24      ; rsp --->  +------+  grandes adresses 0x7fffffffd3a0
25      ; rem. : les contenus initiaux de rip et rsp peuvent varier
26
27      call    show_message
28
29      ; rip : 0x401005
30      ; rsp : 0x7fffffffd3a0
31      ;
32      ; pile vide
33      ;                     petites adresses
34      ;           |      |
35      ;           | 0x05 | 0x7fffffffd398
36      ;           | 0x10 |
37      ;           | 0x40 |
38      ;           | 0x00 |
39      ;           | 0x00 |
40      ;           | 0x00 |
41      ;           | 0x00 |
42      ;           | 0x00 |
43      ; rsp --->  +------+  grandes adresses 0x7fffffffd3a0
44      ; rem. : les contenus initiaux de rip et rsp peuvent varier
45
46      mov     rax, 60
47      mov     rdi, 0
48      syscall
49
50  ;
51  ; fonction d'affichage sur la sortie standard
52  ; de la chaîne de caractères "Salut tou·te·s !\n"
53  ;
54  ; argument : aucun
55  ; retour : aucun
56  ;
57  show_message:
58
59      ; rip : 0x401011
60      ; rsp : 0x7fffffffd3a0
61      ;
62      ; adresse de retour au sommet de la pile
63      ;
64      ;                     petites adresses
65      ;           |      |
66      ; rsp --->  | 0x05 | 0x7fffffffd398
```

```
67      ;               | 0x10 |
68      ;               | 0x40 |
69      ;               | 0x00 |
70      ;               | 0x00 |
71      ;               | 0x00 |
72      ;               | 0x00 |
73      ;               | 0x00 |
74      ;               +------+  grandes adresses 0x7fffffffd3a0
75      ; rem. : les contenus initiaux de rip et rsp peuvent varier
76
77      mov     rax, 1
78      mov     rdi, 1
79      mov     rsi, message
80      mov     rdx, [message_length]
81      syscall
82
83      ; rip : 0x40102f
84      ; rsp : 0x7fffffffd3a0
85      ;
86      ; adresse de retour au sommet de la pile
87      ;
88      ;                       petites adresses
89      ;               |       |
90      ; rsp --->     | 0x05 | 0x7fffffffd398
91      ;               | 0x10 |
92      ;               | 0x40 |
93      ;               | 0x00 |
94      ;               | 0x00 |
95      ;               | 0x00 |
96      ;               | 0x00 |
97      ;               | 0x00 |
98      ;               +------+  grandes adresses 0x7fffffffd3a0
99      ; rem. : les contenus initiaux de rip et rsp peuvent varier
100
101     ret
```

A.2. Fonction avec paramètres mais sans retour

Voici le code de la section 2.4 (p. 9) enrichi de commentaires où les évolutions des registres `rip` et `rsp` ainsi que celle de l'état de la pile sont renseignées :

```
1   ; 02_f_arg_noret_asm_fc.asm
2   ;
3   ; nasm -f elf64 -F dwarf 02_f_arg_noret_asm_fc.asm
```

```
4  ; ld -o 02_f_arg_noret_asm_fc 02_f_arg_noret_asm_fc.o
5
6  global _start
7
8  section .rodata
9      msg_0   DB      `La vie moderne\n`, 0
10     msg_1   DB      `Une qualification olympique\n`, 0
11
12 section .text
13 _start:
14
15     ; rip : 0x401000
16     ; rsp : 0x7fffffffd3b0
17     ;
18     ; pile vide
19     ;                       petites adresses
20     ;             |       |
21     ; rsp --->  +-------+  grandes adresses 0x7fffffffd3b0
22     ; rem. : les contenus initiaux de rip et rsp peuvent varier
23
24     mov     rdi, msg_0
25     call    show
26
27     ; rip : 0x40100f
28     ; rsp : 0x7fffffffd3b0
29     ;
30     ; pile vide
31     ;                       petites adresses
32     ;             |       |
33     ;             | 0x00 | 0x7fffffffd3a0
34     ;             | 0x00 |
35     ;             | 0x00 |
36     ;             | 0x00 |
37     ;             | 0x00 |
38     ;             | 0x00 |
39     ;             | 0x00 |
40     ;             | 0x00 |
41     ;             | 0x0f | 0x7fffffffd3a8
42     ;             | 0x10 |
43     ;             | 0x40 |
44     ;             | 0x00 |
45     ;             | 0x00 |
46     ;             | 0x00 |
47     ;             | 0x00 |
```

```
48      ;           | 0x00 |
49      ; rsp --->  +------+  grandes adresses 0x7fffffffd3b0
50      ; rem. : les contenus initiaux de rip et rsp peuvent varier
51
52      mov     rdi, msg_1
53      call    show
54
55      ; rip : 0x40101e
56      ; rsp : 0x7fffffffd3b0
57      ;
58      ; pile vide
59      ;                       petites adresses
60      ;           |      |
61      ;           | 0x00 | 0x7fffffffd3a0
62      ;           | 0x00 |
63      ;           | 0x00 |
64      ;           | 0x00 |
65      ;           | 0x00 |
66      ;           | 0x00 |
67      ;           | 0x00 |
68      ;           | 0x00 |
69      ;           | 0x1e | 0x7fffffffd3a8
70      ;           | 0x10 |
71      ;           | 0x40 |
72      ;           | 0x00 |
73      ;           | 0x00 |
74      ;           | 0x00 |
75      ;           | 0x00 |
76      ;           | 0x00 |
77      ; rsp --->  +------+  grandes adresses 0x7fffffffd3b0
78      ; rem. : les contenus initiaux de rip et rsp peuvent varier
79
80      mov     rax, 60
81      mov     rdi, 0
82      syscall
83
84  ;
85  ; fonction d'affichage sur la sortie standard
86  ; d'une chaîne de caractères constante zéro-terminée
87  ; fournie en paramètre
88  ;
89  ; argument 1 : adresse du premier caractère de la chaîne à afficher
90  ; retour : aucun
91  ;
```

```
92  show:
93
94      ; 1er appel de show
95      ;
96      ; rip : 0x40102a
97      ; rsp : 0x7fffffffd3a8
98      ;
99      ; adresse de retour au sommet de la pile
100     ;
101     ;                       petites adresses
102     ;             |      |
103     ; rsp --->   | 0x0f | 0x7fffffffd3a8
104     ;             | 0x10 |
105     ;             | 0x40 |
106     ;             | 0x00 |
107     ;             | 0x00 |
108     ;             | 0x00 |
109     ;             | 0x00 |
110     ;             | 0x00 |
111     ;             +------+  grandes adresses 0x7fffffffd3b0
112     ; rem. : les contenus initiaux de rip et rsp peuvent varier
113
114     ; 2e appel de show
115     ;
116     ; rip : 0x40102a (identique 1er appel)
117     ; rsp : 0x7fffffffd3a8 (identique 1er appel)
118     ;
119     ; adresse de retour au sommet de la pile (différent 1er appel)
120     ;
121     ;                       petites adresses
122     ;             |      |
123     ;             | 0x00 | 0x7fffffffd3a0
124     ;             | 0x00 |
125     ;             | 0x00 |
126     ;             | 0x00 |
127     ;             | 0x00 |
128     ;             | 0x00 |
129     ;             | 0x00 |
130     ;             | 0x00 |
131     ; rsp --->   | 0x1e | 0x7fffffffd3a8
132     ;             | 0x10 |
133     ;             | 0x40 |
134     ;             | 0x00 |
135     ;             | 0x00 |
```

```
136     ;           | 0x00 |
137     ;           | 0x00 |
138     ;           | 0x00 |
139     ;           +------+  grandes adresses 0x7fffffffd3b0
140     ; rem. : les contenus initiaux de rip et rsp peuvent varier
141
142     push    rbp         ; sauvegarde du contexte de pile de l'appelant
143     mov     rbp, rsp    ; création du contexte de pile de l'appelé
144     ; rem. : les 2 lignes ci-dessus sont ici inutiles car l'appelé
145     ;         (c.-à-d. la fonction show) ne crée pas de variable locale
146     ;         et donc n'a pas besoin d'un contexte de pile propre
147
148     ; 1er appel de show
149     ;
150     ; rip : 0x40102e
151     ; rsp : 0x7fffffffd3a0
152     ;
153     ; sauvegarde de rbp au sommet de la pile
154     ;
155     ;                     petites adresses
156     ;           |      |
157     ; rsp --->  | 0x00 | 0x7fffffffd3a0
158     ;           | 0x00 |
159     ;           | 0x00 |
160     ;           | 0x00 |
161     ;           | 0x00 |
162     ;           | 0x00 |
163     ;           | 0x00 |
164     ;           | 0x00 |
165     ;           | 0x0f | 0x7fffffffd3a8
166     ;           | 0x10 |
167     ;           | 0x40 |
168     ;           | 0x00 |
169     ;           | 0x00 |
170     ;           | 0x00 |
171     ;           | 0x00 |
172     ;           | 0x00 |
173     ;           +------+  grandes adresses 0x7fffffffd3b0
174     ; rem. : les contenus initiaux de rip et rsp peuvent varier
175
176     ; 2e appel de show
177     ;
178     ; rip : 0x40102e (identique 1er appel)
179     ; rsp : 0x7fffffffd3a0 (identique 1er appel)
```

```
180     ;
181     ; sauvegarde de rbp au sommet de la pile (identique 1er appel)
182     ;
183     ;                       petites adresses
184     ;             |       |
185     ; rsp --->   | 0x00 | 0x7fffffffd3a0
186     ;             | 0x00 |
187     ;             | 0x00 |
188     ;             | 0x00 |
189     ;             | 0x00 |
190     ;             | 0x00 |
191     ;             | 0x00 |
192     ;             | 0x00 |
193     ;             | 0x1e | 0x7fffffffd3a8
194     ;             | 0x10 |
195     ;             | 0x40 |
196     ;             | 0x00 |
197     ;             | 0x00 |
198     ;             | 0x00 |
199     ;             | 0x00 |
200     ;             | 0x00 |
201     ;             +------+  grandes adresses 0x7fffffffd3b0
202     ; rem. : les contenus initiaux de rip et rsp peuvent varier
203
204     mov     r8, rdi     ; car rdi utilisé par syscall
205
206     mov     rdi, 1      ; STDOUT_FILENO
207     mov     rdx, 1      ; 1 byte à écrire
208
209 .while:
210     cmp     byte [r8], 0    ; fin de chaîne ?
211     jz      .end
212     mov     rax, 1      ; write
213     mov     rsi, r8     ; adresse du byte à écrire
214     syscall
215     inc     r8          ; passage au caractère suivant
216     jmp     .while
217
218 .end:
219
220     mov     rsp, rbp    ; destruction des variables locales
221     ; rem. : la ligne qui précède n'est pas nécessaire car show
222     ;        n'a pas créé de variable locale
223
```

```
224     pop     rbp         ; restauration du contexte de pile de l'appelant
225     ; rem. : la ligne qui précède est indispensable car show s'est
226     ;        créé un contexte de pile propre
227
228     ; 1er appel de show
229     ;
230     ; rip : 0x401054
231     ; rsp : 0x7fffffffd3a8
232     ;
233     ; adresse de retour au sommet de la pile
234     ;
235     ;                       petites adresses
236     ;           |      |
237     ;           | 0x00 | 0x7fffffffd3a0
238     ;           | 0x00 |
239     ;           | 0x00 |
240     ;           | 0x00 |
241     ;           | 0x00 |
242     ;           | 0x00 |
243     ;           | 0x00 |
244     ;           | 0x00 |
245     ; rsp --->  | 0x0f | 0x7fffffffd3a8
246     ;           | 0x10 |
247     ;           | 0x40 |
248     ;           | 0x00 |
249     ;           | 0x00 |
250     ;           | 0x00 |
251     ;           | 0x00 |
252     ;           | 0x00 |
253     ;           +------+  grandes adresses 0x7fffffffd3b0
254     ; rem. : les contenus initiaux de rip et rsp peuvent varier
255
256     ; 2e appel de show
257     ;
258     ; rip : 0x401054 (identique 1er appel)
259     ; rsp : 0x7fffffffd3a8 (identique 1er appel)
260     ;
261     ; adresse de retour au sommet de la pile (différent 1er appel)
262     ;
263     ;                       petites adresses
264     ;           |      |
265     ;           | 0x00 | 0x7fffffffd3a0
266     ;           | 0x00 |
267     ;           | 0x00 |
```

```
268     ;             | 0x00 |
269     ;             | 0x00 |
270     ;             | 0x00 |
271     ;             | 0x00 |
272     ;             | 0x00 |
273     ; rsp --->    | 0x1e | 0x7fffffffd3a8
274     ;             | 0x10 |
275     ;             | 0x40 |
276     ;             | 0x00 |
277     ;             | 0x00 |
278     ;             | 0x00 |
279     ;             | 0x00 |
280     ;             | 0x00 |
281     ;             +------+  grandes adresses 0x7fffffffd3b0
282     ; rem. : les contenus initiaux de rip et rsp peuvent varier
283
284     ret
```

Comme la fonction `show` est invoquée deux fois, les commentaires indiquant l'état des registres `rip` et `rsp` et de la pile y apparaissent chaque fois en deux versions.

B. Codes C

B.1. string_c

Le fichier d'en-têtes et le fichier source qui suivent constituent les équivalents en langage C du source `string_asm.asm` de la section 3.3.1 en page 14.

B.1.1. `string_c.h`

```
1  /*!
2   * \file string_c.h
3   */
4  #include <stdint.h>
5
6  /*!
7   * \brief Affichage hexadécimal.
8   *
9   * Affiche en hexadecimal sur la console la valeur du 1er argument
10  * suivi d'un passage à la ligne si le 2e argument vaut 1.
11  *
12  * \param value l'entier à afficher.
13  * \param newline s'il vaut 1, l'affichage se termine par un passage
```

```
 *                à la ligne.
 */
void print_hex(uint64_t value, uint64_t newline);
```

B.1.2. string_c.c

```
/*!
 * \file string_c.c
 */
#include <stdint.h>

#include <unistd.h>

void print_hex(uint64_t value, uint64_t newline)
{
    char hex_str [] = "0x??????????????\n";

    // conversion
    char * hex_ptr = hex_str + (sizeof(hex_str) - 1) - 2;
    for (unsigned u = 0; u != 16; ++u, --hex_ptr)
    {
        char tmp = 0xF;
        tmp &= value;

        if (tmp >= 10)
        {
            tmp += 'a' - 10;
        }
        else
        {
            tmp += '0';
        }

        *hex_ptr = tmp;

        value >>= 4;
    }

    if (newline != 1)
    {
        newline = 0;
    }

```

```
38      write(STDOUT_FILENO, hex_str, (sizeof(hex_str) - 1) - 1 + newline);
39
40      return;
41  }
```

B.1.3. Commentaires

Le premier argument de la fonction `print_hex()` est `value`.

B.2. number_c

Le fichier d'en-têtes et le fichier source qui suivent constituent les équivalents en langage C du source `number_asm.asm` de la section 3.3.2 en page 17.

B.2.1. number_c.h

```
1   /*!
2    * \file number_c.h
3    */
4   #include <stdint.h>
5
6   /*!
7    * \brief Retourne le produit des 2 arguments considérés non signés.
8    *
9    * Rem. : l'opérateur * fait ce boulot !
10   *
11   * \param lhs seuls les 4 bytes de poids le plus faible sont pris
12   *            en compte.
13   * \param rhs seuls les 4 bytes de poids le plus faible sont pris
14   *            en compte.
15   *
16   * \return le produit de lhs par rhs.
17   */
18  uint64_t multiply(uint64_t lhs, uint64_t rhs);
19
20  /*!
21   * \brief Calcule le produit des 2 arguments avec affichage de
22   *        l'opération et du résultat.
23   *
24   * Attention : opérandes sur 4 bytes.
25   *
26   * \param lhs seuls les 4 bytes de poids le plus faible sont pris
27   *            en compte.
28   * \param rhs seuls les 4 bytes de poids le plus faible sont pris
```

```
 *             en compte.
 */
void multiply_print(uint64_t lhs, uint64_t rhs);
```

B.2.2. number_c.c

```
/*!
 * \file number_c.h
 */
#include <stdint.h>

#include <unistd.h>

#include "string_c.h"

uint64_t multiply(uint64_t lhs, uint64_t rhs)
{
    // garder 4 bytes de poids faible
    lhs &= 0xffffffff;
    rhs &= 0xffffffff;

    // calcul
    uint64_t product = 0;
    while ((int64_t) --lhs >= 0)
    {
        product += rhs;
    }
    // rem. : ce serait mieux de boucler min(rdi, rsi) fois

    return product;
}

void multiply_print(uint64_t lhs, uint64_t rhs)
{
    char mul_str [] = " * ";
    char equ_str [] = " = ";

    // garder 4 bytes de poids faible
    lhs &= 0xffffffff;
    rhs &= 0xffffffff;

    print_hex(lhs, 0);

```

```
38      write(STDOUT_FILENO, mul_str, sizeof(mul_str) - 1);
39
40      print_hex(rhs, 0);
41
42      write(STDOUT_FILENO, equ_str, sizeof(equ_str) - 1);
43
44      print_hex(multiply(lhs, rhs), 1);
45
46      return;
47  }
```

B.3. 03_test_c

Le fichier qui suit constitue l'équivalent en langage C du fichier `03_test_asm.asm` de la section 3.3.3 en page 21.

B.3.1. `03_test_c.c`

```
1   /*!
2    * \file 03_test_c.c
3    */
4   #include <unistd.h>
5
6   #include "number_c.h"
7   #include "string_c.h"
8
9   int main()
10  {
11      print_hex(multiply(0x2, 0x1000), 1);
12
13      multiply_print(0xffffffff, 0x2);
14
15      _exit(0);
16  }
```

B.4. Exécutable

Voici la commande pour produire l'exécutable `03_test_c` obtenu à partir des sources des sections B.1, B.2 et B.3 :

```
$ gcc -o 03_test_c string_c.c number_c.c 03_test_c.c
```

Si on désire un exécutable avec des informations de déboguage :

```
$ gcc -g -o 03_test_c string_c.c number_c.c 03_test_c.c
```

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] *System V Application Binary Interface, AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models) Version 1.0*, janvier 2018. https://github.com/hjl-tools/x86-psABI/wiki/X86-psABI.

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