Chapitre 3

Programmation en langage machine

Nous avons dit que le plus efficace, du point de vue de l'exéution des calculs, est de programmer en langage machine. Pour l'instant, nous avons vu comment programmer dans un langage dit de haut niveau, à savoir le langage C, et dans un langage intermédiaire, un langage d'assemblage. Comment programmer directement en langage machine ? Ce n'est pas toujours facile sur nos systèmes informatique modernes.

On peut programmer sans utiliser de système d'exploitation, en plaçant le programme sur le tout premier secteur d'une disquette (sur celui d'une clé USB de nos jours, ne possèdant pas de secteur à proprement parler, mais cela revient au même car elle est formatée en FAT) et démarrer l'ordinateur à partir de celle-ci. C'est relativement facile pour le microprocesseur `8086` d'*Intel*. C'est un peu plus difficile pour les microprocesseurs suivants d'*Intel*, qu'ils soient 16 bits (comme le `80286`), 32 bits (le `80386`, le `80486` ou le *pentium*) ou 64 bits : un tel microprocesseur démarre comme un `8086` et il faut le préparer pour passer en *mode protégé*, puis éventuellement en mode 64 bits.

Nous allons donc utiliser un système informatique formé d'un PC muni d'un microprocesseur à architecture x86-64 et du système d'exploitation GNU/Linux pour x86-64.

3.1 Adaptation d'un programme exécutable

Un programme exécutable (sous-entendu lancé à partir d'un système d'exploitation donné) n'est que très rarement constitué que par du code en langage machine. C'était cependant le cas des exécutables dit *.com* de CP/M, puis de MS-DOS, *binary* sous Unix.

De nos jours, un exécutable est conçu pour un système d'exploitation donné et est formé d'au moins deux parties : la partie constituée du code en langage machine existe toujours mais elle est précédée d'une première partie, appelée **préfixe** (puisqu'elle précède la partie en code machine), indiquant au système d'exploitation, au minimum, où il faut placer la seconde partie en mémoire vive et par quelle instruction il faut commencer. Il peut y avoir plus de deux parties : la partie en code machine correspond à la section `.text` traditionnelle ; de même, il peut y avoir des parties pour les sections `.data` et `.bss` ou autres, pour la pile, des parties servant au débogage, etc.

Pour savoir comment écrire un programme directement en code machine, par exemple avec un éditeur hexadécimal, et donc le préfixe associé, le mieux est de commencer par étudier le programme exécutable d'un programme simple.

3.1.1 Dissection d'un fichier exécutable simple

Le programme.- Écrivons un programme `add.s` en langage d'assemblage plaçant un entier de 64 bits dans le registre `RAX`, auquel il ajoute un entier de 64 bits :

```
        .section .text
        .global _start
_start: movq $12345678,%rax
        addq $12345678,%rax
        nop
```

sans oublier de terminer par une ligne vide.

L'exécutable obtenu :

```
$ as add.s -o add.o
$ ld ./add.o -o ./add
$ ls -l ./add
-rwxr-xr-x 1 patrick patrick 664 2013-05-07 09:26 ./add
$
```

comporte 664 octets. On peut réduire le code objet en y supprimant les symboles (la *table des symboles* constituant encore une autre partie du code exécutable) :

```
$ strip -s ./add
$ ls -l ./add
-rwxr-xr-x 1 patrick patrick 344 2013-05-07 09:29 ./add
$
```

Il reste quand même 344 octets. Regardons le code grâce à un éditeur hexadécimal, par exemple `hexedit` en mode console ou `GHex` en mode graphique :

```
$ ghex2 ./add &
```

ce qui donne une nouvelle fenêtre (figure 3.1).

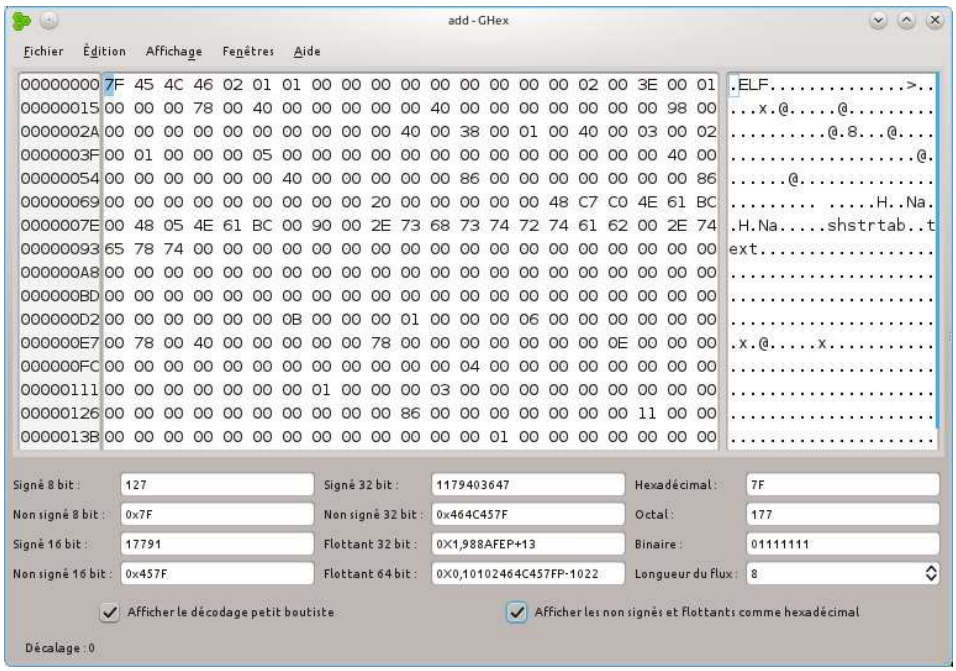

FIGURE 3.1 – Le contenu du fichier `add`

Les fichiers ELF.- UNIX a commencé par utiliser des exécutables au format dit `a.out` (rappelé par le nom par défaut d'un exécutable), puis COFF (pour *Common Object File Format*) et enfin ELF (pour *Executable and Linking Format* puis *Executable and Linkable Format*, c'est-à-dire *format exécutable et liable*), développé par USL (*Unix System Laboratories*).

L'exécutable que nous avons obtenu est codé dans ce format ELF :

```
$ file ./add
./add: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, stripped
$
```

En nous aidant de la documentation [ELF, ELF-64], décrivons le contenu du fichier `add`, détaillé à la figure 3.1. Le fichier commence par l'**en-tête ELF**, qui n'est pas d'une taille fixe :

En-tête ELF

— Cet en-tête commence par un **identificateur**, champ `e_ident`, de taille 16 octets, dont 9 seulement sont utilisés :

  — Les quatre premiers octets, à savoir `0x7F`, `0x45`, `0x4C` et `0x46`, les trois derniers donnant 'ELF' en ASCII, constituent la **signature** permettant de reconnaître un fichier ELF parmi tous les formats possibles (les formats exécutables, certes, mais aussi les formats d'images, de sons, etc). C'est comme cela que l'utilitaire `file` a reconnu qu'il s'agit d'un fichier ELF.

  Existe-t-il un organisme de normalisation assurant l'unicité des identificateurs des types de fichiers ? Non, malheureusement.

  Windows utilise une technique différente de celle de *Linux* pour déterminer l'application avec lequel ouvrir une fichier, reposant sur le suffixe du nom du fichier.

  — Le cinquième octet, ici 2, spécifie la **classe**.

Il n'existe que deux classes pour l'instant : 1 pour l'architecture 32 bits et 2 pour l'architecture 64 bits. On retrouve bien que nous sommes sur une architecture 64 bits.

— Le sixième octet, ici 1, spécifie la façon de coder.
Là aussi, il n'y en a que deux : 1 pour complément à deux et petit boutien et 2 pour complément à deux et grand boutien.
Nous utilisons un microprocesseur *Intel*, il est donc normal que le codage soit petit boutien.

— Le septième octet, ici 1, spécifie la **version**.
Il n'y a qu'une seule valeur possible : 1 pour la version actuelle. Si on avait 0, la version ne serait pas valide et l'exécution ne devrait pas continuer.

— Le huitième octet, appelé **OS/ABI**, identifie le système d'exploitation (**OS**) et l'interface binaire des applications (**ABI** pour *Application Binary Interface*).
Il existe évidemment un certain nombre de possibilités (consulter le fichier en-tête `elf.h` pour les détails). On a ici 0 pour 'UNIX - System V' là où on aurait pu s'attendre à 3 pour 'Linux', mais cela n'a pas d'importance puisque les deux sont compatibles.

— Le neuvième octet spécifie la version de l'ABI.
Pour l'instant, seul 0 existe.

— Les octets 10 à 16 sont réservés pour l'avenir.
Le remplissage (*padding* en anglais) est constitué de 0 mais, de toute façon, il n'est pas lu.

— L'identificateur est suivi d'un **entier court**, c'est-à-dire codé sur deux octets, champ `e_type`, spécifiant le **type** de fichier, ici `02 00`.
Puisque le codage est petit-boutien, on a 2 dans notre cas, spécifiant un fichier exécutable, seul type nous intéressant pour l'instant.

— L'entier court suivant, champ `e_machine`, ici `3E 00`, spécifie l'architecture.
On vérifie dans le fichier `elf.h` que 62 correspond à 'AMD x86-64'.
Rappelons que cette architecture a été originellement définie par AMD et non par *Intel*.

— L'**entier long** suivant, codé sur quatre octets, champ `e_version`, ici `01 00 00 00`, indique la version du fichier.
Une seule valeur est possible pour l'instant : 1 pour version actuelle.

— Le champ suivant, `e_entry`, spécifie l'adresse virtuelle à laquelle le système doit transférer initialement le contrôle, pour démarrer le processus.
Puisqu'il s'agit d'une architecture 64 bits, cette adresse occupe huit octets : `78 00 40 00 00 00 00 00`, soit `0x400078` en codage petit boutien.

— Le champ suivant, `e_phoff` pour *Program Header OFFset*, contient le décalage, en octets, depuis le début du fichier de la table contenant les en-têtes de programme.
Puisqu'il s'agit d'une architecture 64 bits, ce décalage occupe huit octets : `40 00 00 00 00 00 00 00`, soit 64 en codage petit boutien.

— Le champ suivant, `e_shoff` pour *Section Header OFFset*, contient le décalage, en octets, de la table des en-têtes de sections : `98 00 00 00 00 00 00 00`, soit 152.
Dans le vocabulaire ELF, ce qui s'appelle 'section' pour les programmes en langage d'assemblage (à savoir *txt*, *data*, *bss*) s'appelle **segment**. Les **sections** ELF ne servent pas à l'exécution d'un programme, mais sont utiles pour le débogage par exemple.

— Le champ suivant, `e_flags`, est un entier long contenant les drapeaux nécessaires à l'exécution du programme : dans notre cas, on a `00 00 00 00`, donc aucun drapeau n'est positionné.

— Le champ suivant, `e_hsize` pour *Header SIZE*, est un entier court contenant la taille de l'en-tête ELF, en octets. Nous avons ici `40 00`, soit 64.

L'en-tête ELF sera donc suivi de la table contenant les en-têtes de programme.

— Le champ suivant, `e_phentsize` pour *Program Header ENTry SIZE*, est un entier court contenant la taille, en octets, d'une entrée de la table des en-têtes de programme. Toutes les entrées ont la même taille. Nous avons ici `38 00`, soit 56.

— Le champ suivant, `e_phnum` pour *Program Header NUMber*, est un entier court contenant le nombre d'entrées dans la table des en-têtes de programme. Nous avons ici `01 00`, soit 1.

Ceci se justifie puisque nous avons une seule section dans notre programme : la section de code.

— Le champ suivant, `e_shentsize` pour *Section Header Entry SIZE*, est un entier court contenant la taille, en octets, d'un en-tête de section (toutes les entrées de la table des en-têtes de section ayant la même taille). Nous avons ici `40 00`, soit 64.

— Le champ suivant, `e_shnum` pour *Section Header NUMber*, est un entier court contenant le nombre d'entrées dans la table des en-têtes de section. Nous avons ici `03 00`, soit 3.

— Le champ suivant, `e_shstmdx` pour *Section Header STring M inDeX*, est un entier court contenant l'indice, dans la table des entrées des en-têtes de section, de l'entrée associée à la table des chaînes de caractères des noms des sections. Nous avons ici `02 00`, soit 2.

Il s'agit du dernier champ puisque la taille de l'en-tête, spécifiée par le champ `e_hsize`, est de 64 octets.

L'utilitaire `readelf` permet de décoder automatiquement les en-êtes ELF plutôt que de le faire à la main :

```
$ readelf -h ./add
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400078
  Start of program headers:          64 (bytes into file)
  Start of section headers:          152 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         1
  Size of section headers:           64 (bytes)
  Number of section headers:         3
  Section header string table index: 2
$
```

On retrouve ce que nous venons de dire (heureusement !).

Table des en-têtes de programme

Comme nous l'avons vu lors de l'étude de l'en-tête ELF, celui-ci est suivi de la table des en-têtes de programme, contenant ici une seule entrée de 56 octets. Détaillons la structure des en-têtes de programme pour une architecture 64 bits à propos de cet en-tête :

— Le premier champ, `p_type`, est un entier long spécifiant le type du segment. Nous avons ici `01 00 00 00`, soit 1 en petit boutien. Le type 1 spécifie un segment chargeable.

— Le second champ, `p_flags`, est un entier long qui est un masque de bits des attributs relatifs à ce segment : 1 pour un segment exécutable, 2 pour un segment lisible et 4 pour un segment sur lequel on peut écrire. On a ici `05 00 00 00`, c'est-à-dire un segment qu'on peut lire et exécuter, caractéristique de la section `.text`.

— Le troisième champ, `p_offset`, contient le décalage du premier octet du segment par rapport au début du fichier. On a ici `00 00 00 00 00 00 00 00`, soit un décalage de zéro.

— Le quatrième champ, `p_vaddr` pour *Virtual ADDRess*, contient l'adresse virtuelle en mémoire du premier octet du segment. On a ici `00 00 40 00 00 00 00 00`, soit `0x400000`.

— Le cinquième champ, `p_paddr` pour *Physical ADDRess*, contient l'adresse physique en mémoire du premier octet du segment, si cela est pertinent pour l'architecture, 0 si ce ne l'est pas. On a ici `00 00 40 00 00 00 00 00`, donc la même valeur que pour l'adresse virtuelle.

— Le sixième champ, `p_filesz` pour *FILE SiZe*, est un **entier très long**, c'est-à-dire codé sur huit octets, contenant la taille, en octets, de l'image fichier de ce segment. Ici on a `86 00 00 00 00 00 00 00`, soit 134.

— Le septième champ, `p_memsz` pour *MEMory SiZe*, est un entier très long contenant la taille, en octets, de l'image mémoire de ce segment. Ici on retrouve la même valeur 134.

— Le dernier champ, `p_align`, est un entier très long contenant la valeur selon laquelle le segment est aligné en mémoire et dans le fichier. On a ici `00 00 20 00 00 00 00 00`, soit `0x200000`.

On peut retrouver ces informations avec l'utilitaire `readelf`, avec une option différente :

```
$ readelf -a ./add
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400078
  Start of program headers:          64 (bytes into file)
  Start of section headers:          152 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         1
  Size of section headers:           64 (bytes)
  Number of section headers:         3
```

```
  Section header string table index: 2

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         0000000000400078  00000078
       000000000000000e  0000000000000000  AX       0     0     4
  [ 2] .shstrtab         STRTAB           0000000000000000  00000086
       0000000000000011  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)

There are no section groups in this file.

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000086 0x0000000000000086  R E    200000

 Section to Segment mapping:
  Segment Sections...
   00     .text

There is no dynamic section in this file.

There are no relocations in this file.

There are no unwind sections in this file.

No version information found in this file.
$
```

Le segment de code

Nous avons vu que le point d'entrée dans le segment de code se situe à l'adresse `0x400078` alors que le début du fichier se situe à l'adresse `0x400000`. On en déduit que le code commence au décalage `0x78` du fichier ELF. On a vu de plus que la taille de ce segment est `0xE`, soit 15 octets. Ces renseignements nous permettent d'en extraire le code :

```
          0  1  2  3  4  5  6  7    8  9  A  B  C  D  E  F
00000070 00 00 00 00 00 00 00 00   48 C7 C0 4E 61 BC 00 48
00000080 04 05 4E 61 BC 00 90 00   00 00 00 00 00 00 00 00
```

On aurait pu se servir d'un autre utilitaire pour récupérer le code machine de notre fichier exécutable :

```
$ objdump -D ./add

./add:     file format elf64-x86-64


Disassembly of section .text:

0000000000400078 <.text>:
  400078:       48 c7 c0 4e 61 bc 00    mov    $0xbc614e,%rax
  40007f:       48 05 4e 61 bc 00       add    $0xbc614e,%rax
  400085:       90                      nop
$
```

On retrouve bien le même code machine.

Exécution du programme.- Le programme ne peut pas être exécuté tel quel sous Linux, puisqu'il ne se termine pas correctement par un retour au système d'exploitation et qu'il n'y a rien de visible. On peut, cependant, utiliser `gdb` pour voir ce qui se passe pas à pas :

```
$ as -gstabs add.s -o ./add.o
$ ld -O0 add.o -o ./add
$ gdb ./add
[...]
(gdb) b 1
Breakpoint 1 at 0x400078: file add.s, line 1.
(gdb) r
Starting program: /home/add

Breakpoint 1, _start () at add.s:3
3       _start: movq $12345678,%rax
(gdb) info registers
rax            0x0      0
[...]
rsp            0x7fffffffdfb0   0x7fffffffdfb0
[...]
rip            0x400078 0x400078 <_start>
eflags         0x202    [ IF ]
cs             0x33     51
ss             0x2b     43
[...]
(gdb) s
4               addq $12345678,%rax
```

```
(gdb) info registers
rax            0xbc614e 12345678
[...]
rsp            0x7fffffffdfb0   0x7fffffffdfb0
[...]
rip            0x40007f 0x40007f <_start+7>
eflags         0x202    [ IF ]
cs             0x33     51
ss             0x2b     43
[...]
(gdb) s
5               nop
(gdb) info registers
rax            0x178c29c        24691356
[...]
rsp            0x7fffffffdfb0   0x7fffffffdfb0
[...]
rip            0x400085 0x400085 <_start+13>
eflags         0x216    [ PF AF IF ]
cs             0x33     51
ss             0x2b     43
[...]
```

qui nous montre que l'addition a été effectuée correctement. Remarquez le positionnement de l'indicateur AF dû au débordement sur le premier demi-octet.

3.1.2 Un autre débogueur : `fdbg`

Le débogueur GNU, traditionnellement utilisé sous *Linux*, présente de gros défauts, en particulier celui d'exiger l'utilisation de l'option `-gstabs` lors de l'assemblage, ce qui a pour conséquence de grossir (artificiellement) le fichier exécutable.

Il existe un débogueur beaucoup plus léger, `fdbg` (pour *Fasm DeBuGger*, `FASM` étant un assembleur concurrent de `gas`).

Installation.- Le débogueur `fdbg`, écrit en langage d'assemblage et assemblé avec FASM, n'utilise aucune bibliothèque. On le récupère sur le site :

`http://fdbg.x86asm.net/`

et on le place soit dans le répertoire dans lequel on veut l'utiliser, soit dans le répertoire `usr/bin` (si on dispose des droits d'administrateur).

Utilisation.- Pour lancer `fdbg`, il suffit d'écrire son nom en ligne de commande, suivi du nom de l'exécutable que l'on veut analyser :

```
$ fdbg ./add
0000000000400078  > _start: mov rax,00BC614E
-
```

La première ligne du programme est affichée. Pour visualiser l'état des registres, on utilise la commande `r` (pour *Registers*) :

```
r
rax=0000000000000000 rbx=0000000000000000 rcx=0000000000000000
rdx=0000000000000000 rsp=00007FFFBD064DC0 rbp=0000000000000000
rsi=0000000000000000 rdi=0000000000000000 r8=0000000000000000
r9=0000000000000000  r10=0000000000000000 r11=0000000000000200
r12=0000000000000000 r13=0000000000000000 r14=0000000000000000
r15=0000000000000000 rip=0000000000400078 rflags=0000000000000200
cf=0 pf=0 af=0 zf=0 sf=0 tf=0 if=1 df=0 of=0
-
```

Le registre `RAX` contient 0, ce qui est normal puisque la première instruction de copie n'a pas été exécutée. Faisons-la exécuter, grâce à la commande `s` (pour *Step*) :

```
s
000000000040007F  > add rax,00BC614E
r
rax=00000000BC614E rbx=0000000000000000 rcx=0000000000000000
rdx=0000000000000000 rsp=00007FFFBD064DC0 rbp=0000000000000000
rsi=0000000000000000 rdi=0000000000000000 r8=0000000000000000
r9=0000000000000000  r10=0000000000000000 r11=0000000000000200
r12=0000000000000000 r13=0000000000000000 r14=0000000000000000
r15=0000000000000000 rip=000000000040007F rflags=0000000000000202
cf=0 pf=0 af=0 zf=0 sf=0 tf=0 if=1 df=0 of=0
-
```

Le registre `RAX` a pour valeur celle que nous y avons placée. Continuons pour exécuter l'addition :

```
s
0000000000400085  > nop
r
rax=000000000178C29C rbx=0000000000000000 rcx=0000000000000000
rdx=0000000000000000 rsp=00007FFFBD064DC0 rbp=0000000000000000
rsi=0000000000000000 rdi=0000000000000000 r8=0000000000000000
r9=0000000000000000  r10=0000000000000000 r11=0000000000000200
r12=0000000000000000 r13=0000000000000000 r14=0000000000000000
r15=0000000000000000 rip=0000000000400085 rflags=0000000000000216
cf=0 pf=1 af=1 zf=0 sf=0 tf=0 if=1 df=0 of=0
-
```

Pour quitter `fdbg`, on utilise la commande `k` pour *Kill* :

```
k
$
```

3.1.3 Adaptation du programme

La dissection de l'exécutable du programme précédent nous permet d'écrire directement des programmes en langage machine, en adaptant le programme précédent.

Notre but est d'ajouter une instruction supplémentaire, à savoir :

```
subq $32,%rax
```

avant l'instruction `nop`.

Traduction en code machine.- Le premier problème est de savoir comment traduire cette instruction en code machine. Quel est le code opération de `sub ,%ax` ? Une petite recherche nous montre qu'il s'agit de `0x2d`. On aura donc ici, en utilisant le préfixe `REX` pour affecter le registre `RAX` :

```
48 2D 20 00 00 00 00 00 00 00
```

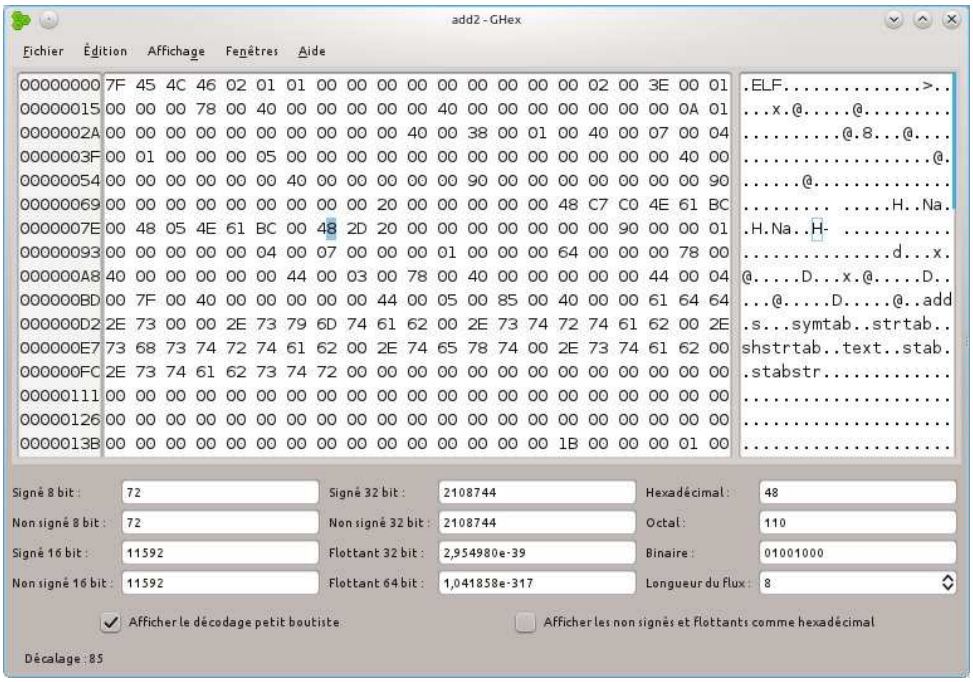

FIGURE 3.2 – Changement du segment de code de `add` en `add2`

Changement dans le segment de code.- Renommons `add` en `add2` :

```
$ cp add ./add2
```

Effectuons le changement voulu avec `ghex2`, en utilisant le mode d'insertion : plaçons le curseur sur le 9 de l'octet de décalage `0x85`, c'est-à-dire à l'emplacement de l'instruction `nop` ; passons en mode d'insertion dans le menu déroulant de *Édition* ; insérons les dix octets `48 2D 20 00 00 00 00 00 00 00` ; sauvegardons le nouveau fichier.

Vérifions que le nouveau fichier `add2` obtenu comporte bien dix octets de plus que `add` :

```
$ ls -l ./add
-rwxr-xr-x 1 patrick patrick 928 2013-05-10 09:31 ./add
$ ls -l ./add2
-rwxr-xr-x 1 patrick patrick 938 2013-05-10 10:04 ./add2
```

Sortons du mode d'insertion. Nous avons deux actions de plus à effectuer :

— Il faut changer le décalage de la table des en-têtes de section : remplaçons le `00` de l'avant-dernier octet de la deuxième ligne par `0A` (de façon à ajouter dix).
— Il faut changer la taille du segment de code en passant de `86` à `90` aux octets de décalage `0x60` et `0x68`.

Exécution du programme.- Si on essaie d'utiliser `gdb`, on obtient la réaction suivante :

```
$ gdb ./add2
[...]
Reading symbols from /home/add2...(no debugging symbols found)...done.
(gdb)
```

Utilisons donc `fdbg` :

```
$ fdbg ./add2
0000000000400078  > mov rax,00BC614E
s
000000000040007F  > add rax,00BC614E
r
rax=0000000000BC614E rbx=0000000000000000 rcx=0000000000000000
rdx=0000000000000000 rsp=00007FFF720EF790 rbp=0000000000000000
rsi=0000000000000000 rdi=0000000000000000 r8=0000000000000000
r9=0000000000000000  r10=0000000000000000 r11=0000000000000200
r12=0000000000000000 r13=0000000000000000 r14=0000000000000000
r15=0000000000000000 rip=000000000040007F rflags=0000000000000202
cf=0 pf=0 af=0 zf=0 sf=0 tf=0 if=1 df=0 of=0
s
0000000000400085  > sub rax,00000020
r
rax=000000000178C29C rbx=0000000000000000 rcx=0000000000000000
rdx=0000000000000000 rsp=00007FFF720EF790 rbp=0000000000000000
rsi=0000000000000000 rdi=0000000000000000 r8=0000000000000000
r9=0000000000000000  r10=0000000000000000 r11=0000000000000200
r12=0000000000000000 r13=0000000000000000 r14=0000000000000000
r15=0000000000000000 rip=0000000000400085 rflags=0000000000000216
cf=0 pf=1 af=1 zf=0 sf=0 tf=0 if=1 df=0 of=0
s
000000000040008B  > add [rax],al ; [000000000178C27C]=?
r
rax=000000000178C27C rbx=0000000000000000 rcx=0000000000000000
rdx=0000000000000000 rsp=00007FFF720EF790 rbp=0000000000000000
rsi=0000000000000000 rdi=0000000000000000 r8=0000000000000000
r9=0000000000000000  r10=0000000000000000 r11=0000000000000200
r12=0000000000000000 r13=0000000000000000 r14=0000000000000000
r15=0000000000000000 rip=000000000040008B rflags=0000000000000202
cf=0 pf=0 af=0 zf=0 sf=0 tf=0 if=1 df=0 of=0
k
$
```

Le début est le même que pour `add` puis l'instruction de soustraction est reconnue et effectuée, comme le montre le contenu du registre `RAX`.

Remarquons que l'instruction suivante n'a pas de sens, ce qui est normal puisque `fgdb` essaie de désassembler ce qui n'est pas du code mais le début de la section suivante.

3.1.4 Retour sur gdb

Nous avons dit qu'il n'était pas facile de visualiser avec gdb un fichier ELF modifié. Ce n'est pas facile mais pas impossible. Une première façon de faire est d'analyser un peu plus le format ELF64 et d'effectuer tous les changements nécessaires. Mais ceci n'est pas un cours sur *Linux* et sur le format phare des exécutables de ce système d'exploitation. Une autre façon est d'ajouter un grand nombre d'instructions nop au code de celui dont on partira, qui seront pris en considération par gdb, puis de les surcharger.

Reprenons notre programme add.s et ajoutons-lui des nop, qu'importe le nombre mais un grand nombre (au moins dix dans notre cas), pour obtenir add3.s :

```
        .section .text
        .global _start
_start: movq $12345678,%rax
        addq $12345678,%rax
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
```

Assemblons-le, lions-le et effectuons-en une copie add4 :

```
$ as -gstabs ./add3.s -o ./add3.o
$ ld -O0 ./add3.o -o ./add3
$ cp add3 ./add4
```

Effectuons le changement voulu avec ghex2, sans utiliser le mode d'insertion (ce qui aura pour effet de remplacer du code précédent par le nouveau code) : plaçons le curseur sur le 9 de l'octet de décalage 0x85, c'est-à-dire à l'emplacement de la première instruction nop ; insérons les dix octets 48 2D 20 00 00 00 00 00 00 00 ; sauvegardons le nouveau fichier.

Visualisons avec gdb :

```
$ gdb ./add4
[...]
Reading symbols from /add4...done.
(gdb) b 1
Breakpoint 1 at 0x400078: file ./add3.s, line 1.
(gdb) r
Starting program: /home/add4

Breakpoint 1, _start () at ./add3.s:3
3       _start: movq $12345678,%rax
(gdb) info registers
rax            0x0      0
[...]
rip            0x400078 0x400078 <_start>
eflags         0x202    [ IF ]
```

```
cs             0x33     51
ss             0x2b     43
[...]
(gdb) s
4               addq $12345678,%rax
(gdb) info registers
rax            0xbc614e 12345678
[...]
rip            0x40007f 0x40007f <_start+7>
eflags         0x202    [ IF ]
cs             0x33     51
ss             0x2b     43
[...]
(gdb) s
5               nop
(gdb) info registers
rax            0x178c29c        24691356
[...]
rip            0x400085 0x400085 <_start+13>
eflags         0x216    [ PF AF IF ]
cs             0x33     51
ss             0x2b     43
[...]
(gdb) s
11              nop
(gdb) info registers
rax            0x178c27c        24691324
[...]
rip            0x40008b 0x40008b <_start+19>
eflags         0x202    [ IF ]
cs             0x33     51
ss             0x2b     43
(gdb)
```

Remarquez que `gdb` croit encore travailler avec `add3` (ligne 5) : il dit que la dernière instruction qu'il va effectuer est `nop` mais, cependant, il exécute bien la soustraction, comme le montre le contenu du registre `RAX`.

On peut donc utiliser `gdb` pour de petits changements dans le code. Ce serait beaucoup moins convivial pour des changements plus importants.

3.2 Création d'un fichier exécutable à partir de rien

Utilisons `ghex` pour créer un fichier exécutable de type *binary*.

Curieusement, `ghex` n'est pas capable de créer un fichier. Créons donc un fichier `add5` (vide ou avec 'a') avec un éditeur de texte. Ouvrons-le ensuite avec `ghex` : il contient un seul octet, de valeur `0x61`.

Utilisons le mode *insertion* (dans le menu déroulant *Édition*). On peut supprimer le premier octet grâce à la touche *Suppr*. Insérons ensuite le code voulu, par exemple celui pour :

```
movq $12345678,%rax
addq $12345678,%rax
nop
```

à savoir :

```
48 C7 C0 4E 61 BC 00
48 04 05 4E 61 BC 00
90
```

```
$ objcopy -I binary -O elf64-x86-64 ./add5 ./add6
$
```

3.3 Bibliographie

[ELF] **ELF**, page manuel de l'administrateur Linux, 28 décembre 2007. Version anglaise ou traduite en français le 8 janvier 2008.

[Il s'agit en fait d'un commentaire du fichier en-tête `elf.h`, qu'il faut absolument lire simultanément pour obtenir les valeurs des constantes. La façon de décrire les champs à la C et non octet par octet rend la lecture de la description assez difficile.]

[ELF-64] Hewlett-Packard, **Elf-64 Object File Format**, version 1.5 Draft 2, May 27, 1998, 18 p.

[Hen] HENSZEY, **Smallest x86 ELF Hello World**. Téléchargeable à l'adresse :

`http://timelessname.com/elfbin/`