Ensimag 1A Apprentissage — Printemps 2010

Logiciel de base en C pour processeur Pentium et format ELF

Auteurs : des enseignants actuels et antérieurs du projet C.

Table des matières

Chapitre 1

Objectifs du projet

Tout informaticien doit connaître le langage C. C'est une espèce d'espéranto de l'informatique. Les autres langages fournissent en effet souvent une interface avec C (ce qui leur permet en particulier de s'interfacer plus facilement avec le système d'exploitation) ou sont eux-mêmes écrits en C. D'autre part c'est le langage de base pour programmer les couches basses des systèmes informatiques. Par exemple, on écrit rarement un pilote de périphérique en Ada ou Java. Finalement, en compilation, C est souvent choisi comme cible de langages de plus haut niveau.

Toutefois, il est peu probable (ou plus exactement, peu souhaitable) qu'un ingénieur informaticien soit confronté à de gros développements logiciels entièrement en C. L'objectif pédagogique du projet est donc surtout de montrer comment C peut servir d'interface entre les langages de haut niveau et les couches basses de la machine. Plus précisément, les objectifs du stage C de première année sont :

- Apprentissage de C (en soi, et pour la démarche qui consiste à apprendre un nouveau langage).
- Lien du logiciel avec les couches basses de l'informatique, ici logiciel de base et architecture.
- Le premier projet logiciel un peu conséquent, à développer dans les règles de l'art (mise en œuvre de tests, documentation, démonstration du logiciel, partage du travail, ...)
- Lien avec les autres modules de première année (théorie des langages) et de deuxième année (compilation, système)

Dans ce projet, on considère un sous-ensemble du processeur Pentium. L'architecture de la machine Pentium est décrite en chapitre 4 (mode d'adressages) et en chapitre 5 (instructions). Pour programmer sur la machine Pentium, on utilise le langage d'assemblage ASM décrit chapitre 3 qui est lui-même un sous-ensemble du langage d'assemblage *as* de GNU[1]. L'assemblage est le processus qui consiste à produire un fichier binaire exécutable, compréhensible par le microprocesseur. Le langage d'assemblage permet de décrire de manière abstraite la séquence d'instructions et l'allocation statique des données du programme à coder en binaire. Sur un ordinateur utilisant un système d'exploitation, un programme exécutable ne se réduit pas à une suite d'instructions binaires. Il faut en effet que le fichier produit soit utilisable par le chargeur du système d'exploitation et par ses éditeurs de liens. L'édition de lien est nécessaire pour utiliser les fonctions de la librairie standard comme les entrées/sorties. Elle permet aussi de faire de la compilation séparée : on compile ou assemble de manière séparée plusieurs fichiers (en fait des modules qui définissent un jeu de fonctions) que l'édition de liens permet de réunir ensuite

[1]GNU est un projet de la Free Software Foundation. Voir `http ://www.gnu.org/`

en un seul fichier exécutable. Dans ce but, on utilise un format intermédiaire, le format ELF (Executable and Linkable Format) qui a justement été défini afin de réaliser ces opérations. Ce format est utilisé sur de nombreux systèmes d'exploitation. Il est décrit au chapitre 6.

Ainsi, les binaires produits ou lus dans le projet sont pleinement compatibles avec un système d'exploitation Linux sur un processeur Pentium. De plus, ces binaires peuvent être lus ou produits sur tout système d'exploitation pour laquelle une librairie ELF existe (SunOs, MacOsX, etc). Le schéma général, organisé autour du format ELF des fichiers objets, est donné en figure 1.1.

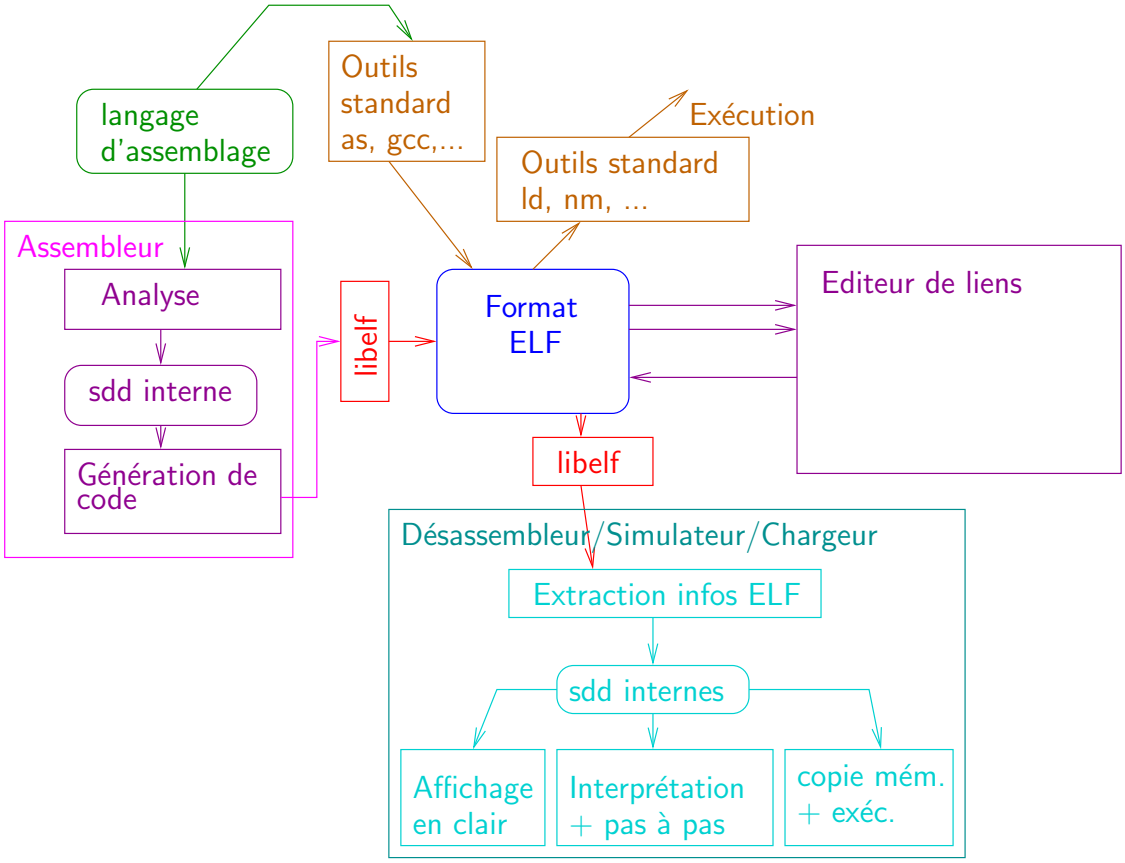

FIG. 1.1 – Schéma général des outils autour de ELF

Assembleur L'assembleur est un programme qui prend un programme écrit dans le langage d'assemblage (fichier "`*.s`") et qui produit un binaire correspond au format Linux/ELF relogeable (fichier "`*.o`").

Edition de liens L'édition de liens permet notamment de recoller ensemble des fichiers binaires qui ont été produits de manière indépendante. Cela procure donc un mécanisme de compilation séparée, ou cela permet à différents langages de communiquer. L'édition de liens permet aussi de partager du code entre différents programmes, et évite ainsi de coûteuses duplications en espace mémoire, que ce soit de la mémoire sur disque ou de la mémoire vive.

[1] sdd : abbréviation de "structure de données"

Pour illustrer ces mécanismes, on va introduire des petits exemples. Il est conseillé de suivre ces explications devant une machine (sur un PC/Linux). Tout d'abord, considérons la fonction `incr` suivante, écrite dans un fichier assembleur `incrfich.s`, et qui incrémente de 1 son argument entier :

```
.globl incr
incr:
        pushl %ebp
        movl %esp,%ebp
        movl 8(%ebp),%eax
        addl $1,%eax
        movl %ebp, %esp
        popl %ebp
        ret
```

Pour produire un fichier binaire relogeable à partir de `incrfich.s`, on peut utiliser l'assembleur GNU `as`. Ce fichier ne peut d'ailleurs pas être directement transformé en fichier exécutable par `as`, car il faudrait pour cela fournir une fonction `main` contenant le code à exécuter. On exécute la ligne suivante et on obtient ainsi le fichier relogeable `incrfich.o` :

```
as incrfich.s -o incrfich.o
```

Pour exécuter la fonction `incr` précédente, nous allons l'appeler depuis un programme C, qui va définir la fonction `main`. Ce fichier `afficheincr.c` est le suivant :

```
#include <stdio.h>
#include <stdlib.h>

/* declaration de la fonction definie
 * dans "incrfich.s"
 */
extern int incr(int x) ;

main () {
  int x ;
  printf("entrez un entier:") ;
  scanf("%d",&x) ;
  printf("incr(%d)=%d\n",x,incr(x)) ;
}
```

On compile ce fichier avec `gcc` et l'option `-c`. Cette option indique que l'on désire obtenir un fichier relogeable appelé "`afficheincr.o`". Remarquons que ce fichier ne peut pas non plus être directement transformé en exécutable puisque, par exemple, la fonction `incr` n'est pas définie. Avec la ligne suivante, on obtient donc un fichier relogeable `afficheincr.o` :

```
gcc -c afficheincr.c
```

Maintenant, on peut lier ensemble ces deux fichiers, via l'éditeur de liens. La commande `ld` avec l'option `-r` permet en effet de produire à partir de plusieurs fichiers relogeables, un seul fichier relogeable. Ainsi la commande suivante lie ensemble `incrfich.o` et `afficheincr.o` dans un seul fichier relogeable `toto.o` :

```
ld -r incrfich.o afficheincr.o -o toto.o
```

Cette forme d'édition de lien, appelée édition de lien statique, recopie le code des 2 fichiers en entrée dans le fichier en sortie. Le fichier `toto.o` est un fichier relogeable qui n'est toujours pas exécutable. En effet, si `incr` et `main` sont définis, les fonctions `printf` et `scanf` ne sont toujours pas définies. Pour créer un exécutable, il faut donc à nouveau réaliser une édition de liens. Cette édition de liens finale doit aussi définir d'autres fonctions utiles pour que l'exécutable puisse être lancé et arrêté proprement par le système d'exploitation. Le plus simple est d'utiliser `gcc`. La ligne suivante crée l'exécutable `toto` :

```
gcc toto.o -o toto
```

Le fichier `toto` est toujours un fichier au format ELF, mais c'est maintenant un programme exécutable : on peut le lancer en l'invoquant depuis la ligne de commande.

Pour comprendre ce que cache cette dernière invocation à `gcc`, on va chercher à obtenir le même comportement en utilisant directement `ld` au lieu de `gcc`. On doit d'abord créer une routine `_start`, appelée au lancement du programme. Cette routine `_start` fait l'appel à la fonction `main`, puis fait un appel système à la routine `sys_exit` du système Linux pour arrêter le processus en cours. On crée cette routine dans un fichier `callmain.s` :

```
.globl _start
_start:
        call  main
        /* on fait ci-dessous un appel systeme a "sys_exit" */
        movl  $1, %eax
        int   $0x80
```

On assemble ce programme avec `as` pour créer le binaire relogeable `callmain.o`. On va maintenant lier `toto.o` avec `callmain.o`. On va de plus indiquer que `printf` et `scanf` sont définies dans les librairies dynamiques du système. Ainsi, le code de `printf` et `scanf` n'est pas dupliqué dans le code de l'exécutable. Lors du chargement du programme en mémoire vive, si le code de ces fonctions n'est pas présent, il sera aussi chargé en mémoire. Si le code de ces fonctions est déjà présent en mémoire, le système utilisera directement celui-ci. C'est ce qu'on appelle l'édition de liens dynamique[2]. La ligne de commande qui réalise tout ça est la suivante :

```
ld toto.o --dynamic-linker /lib/ld-linux.so.2 callmain.o -lc -o toto
```

L'option "`--dynamic-linker/lib/ld-linux.so.2`" indique que l'édition de liens dynamique au moment du chargement doit être réalisée en utilisant "`/lib/ld-linux.so.2`". L'option "`-lc`" indique de se lier (dynamiquement) à la librairie `libc`. Voir la documentation de `ld`.

En fait, `gcc` est capable d'enchaîner tout seul ces différentes opérations directement. Ainsi, ayant seulement créé le binaire `incrfich.o`, on peut obtenir l'exécutable `toto` en tapant simplement :

```
gcc afficheincr.c incrfich.o -o toto
```

Et si on ne veut pas produire le fichier intermédiaire `incrfich.o`, on peut aussi appeler directement `gcc` sur `incrfich.s` :

```
gcc afficheincr.c incrfich.s -o toto
```

[2]On n'étudiera pas ce type d'édition de liens dans le projet. On ne s'intéresse ici qu'au format des fichiers binaires relogeables.

Désassembleur Pour comprendre la structure d'un fichier objet et la nature des informations qu'il contient, rien ne vaut un désassembleur. Désassembler un fichier objet au format ELF revient à : 1) extraire les informations du format ELF (pour cela, on utilise la bibliothèque `libelf` qui fournit un ensemble de procédures et fonctions qui permettent d'extraire les informations sans connaître exactement le format binaire utilisé) ; 2) traduire le codage binaire des instructions machines, des informations sur les symboles (c'est-à-dire les variables et des fonctions) dans une syntaxe lisible.

La phase 1 est indépendante du processeur (ELF est portable sur plusieurs systèmes comme Solaris et Linux, et pour plusieurs processeurs comme le 68000, sparc, pentium, ...). La phase 2 (instructions et symboles) dépend bien sûr du format de codage des instructions du processeur.

Le désassemblage des informations relatives aux symboles ne dépend pas du processeur, mais du système (Linux pour PC dans notre cas). Les informations relatives aux symboles définis dans les fichiers objets sont utiles lors de l'édition de liens, et sont donc définies par des conventions du système, respectées par tous les producteurs de fichiers objets (compilateurs, assembleurs, ...). Un outil unix standard (la commande `nm` pour "names") est disponible pour observer ces informations. Le désassembleur GNU standard s'appelle `objdump` (disponible en particulier sous Linux).

Simulateur Comme le désassembleur, le simulateur commence par le chargement/décodage d'un fichier objet[3], et l'on poursuit par la simulation de la machine en interprétant les instructions décodées. Plus précisément, une fois le fichier chargé dans une mémoire "virtuelle", le simulateur fonctionne en décodant les instructions au fur et à mesure qu'il les interprète.

Un tel logiciel permet par exemple de déboguer des programmes assembleurs pour linux/pentium sans avoir réellement une telle plateforme sous la main (l'assembleur peut lui-aussi s'exécuter sur d'autres machines que des architectures linux/pentium). Un peu comme "gdb" (le débogueur GNU), le simulateur permet d'exécuter des programmes pas-à-pas, de gérer des points d'arrêts, d'observer l'état des registres et de la mémoire, d'afficher le programme assembleur correspondant au binaire, etc...

[3]Dans le cadre du projet, le fichier en entrée devra être un fichier ELF relogeable

Chapitre 2

Consignes et aides pour le projet

2.1 Styles de codage

Indépendemment de la correction des algorithmes, un code de bonne qualité est aussi un code facile, et agréable à lire. Dans un texte en langue naturelle, le choix des mots justes, la longueur des phrases, l'organisation en chapitres et en paragraphes peuvent rendre la lecture fluide, ou bien au contraire très laborieuse.

Pour du code source, c'est la même chose : le choix des noms de variables, l'organisation du code en fonctions, et la disposition (indentation, longueur des lignes, ...) sont très importants pour rendre un code clair. La plupart des projets logiciels se fixent un certain nombre de règles à suivre pour écrire et présenter le code, et s'y tiennent rigoureusement. Ces règles (« Coding Style » en anglais) permettent non seulement de se forcer à écrire du code de bonne qualité, mais aussi d'écrire du code *homogène*. Par exemple, si on décide d'indenter le code avec des tabulations, on le décide une bonne fois pour toutes et on s'y tiens, pour éviter d'écrire du code dans un style incohérent comme :

```
if (a == b) {
  printf("a == b\n");
} else
{
        printf ( "a et b sont différents\n");
}
```

Pour le projet C, les règles que nous vous imposons sont celles utilisées par le noyau Linux. Pour vous donner une idée du résultat, vous pouvez regarder un fichier source de noyau au hasard (a priori, sans comprendre le fond). Vous trouverez un lien vers le document complet sur EnsiWiki, lisez-le. Certains chapitres sont plus ou moins spécifiques au noyau Linux, vous pouvez donc vous contenter des Chapitres 1 à 9. Le chapitre 5 sur les `typedef`s et le chapitre 7 sur les `goto`s sont un peu complexe à assimiler *vraiment*, et sujets à discussion. Vous pouvez ignorer ces deux chapitres pour le projet C.

Nous rappelons ici le document dans les grandes lignes :

- Règles de présentation du code (indentation à 8 caractères, pas de lignes de plus de 80 caractères, placements des espaces et des accolades, ...)
- Règles et conseils pour le nommage des fonctions (trouver des noms courts et expressifs à la fois).
- Règles de découpage du code en fonction : faire des fonctions courtes, qui font une chose et qui le font bien.

– Règles d'utilisations des commentaires : en bref, expliquez *pourquoi* votre code est comme il est, et non *comment*. Si le code a besoin de beaucoup de commentaire pour expliquer comment il fonctionne, c'est qu'il est trop complexe et qu'il devrait être simplifié.

Certaines de ces règles (en particulier l'indentation) peuvent être appliquées plus ou moins automatiquement. Le chapitre 9 vous présente quelques outils pour vous épargner les tâches les plus ingrates : GNU Emacs et la commande `indent` (qui fait en fait un peu plus que ce que son nom semble suggérer).

Pour le projet C, nous vous laissons le choix des outils, mais nous exigeons un code conforme à toutes ces directives.

2.2 Outils

Les outils pour développer, et bien développer en langage C sont nombreux. Nous en présentons ici quelques-uns, mais vous en trouverez plus sur EnsiWiki, et bien sur, un peu partout sur Internet !

– `readelf`, `objdump`, pour examiner le contenu d'un fichier elf. Par exemple :
  – `readelf -s file.o` : Afficher la table des symboles du fichier `file.o`,
  – `readelf -r file.o` : Afficher la table de relocation du fichier,
  – `objdump -d file.o` : Désassembler (afficher le contenu en langage d'assemblage) du fichier.
– `gdb`, le debugger, et son interface graphique `ddd` permettent de tracer l'exécution. Son utilisation est très intéressante, mais il ne faut pas espérer réussir à converger vers un programme correct par approximations successives à l'aide de cet outil, ...
– `valgrind` sera votre compagnon tout au long de ce projet. Il vous permet de vérifier à l'exécution les accès mémoires faits par vos programmes. Ceci permet de détecter des erreurs qui seraient passées inaperçues autrement, ou bien d'avoir un diagnostique pour comprendre pourquoi un programme ne marche pas. Il peut également servir à identifier les fuites mémoires (i.e. vérifier que les zones mémoires allouées sont bien désallouées). Pour l'utiliser :
  `valgrind [options] <executable> <paramètres de l'exécutable>`
– Deux programmes peuvent vous servir lors de vos tests, ou pour l'optimisation de votre code. `gprof` est un outil de profiling du code, qui permet d'étudier les performances de chaque morceau de votre code. `gcov` permet de tester la couverture de votre code lors de vos tests. L'utilisation des deux programmes en parallèle permet d'optimiser de manière efficace votre code, en ne vous concentrant que sur les points qui apporteront une réelle amélioration à l'ensemble. Pour savoir comment les utiliser, lisez le manuel.
– Finalement, pour tout problème avec les outils logiciels utilisés, ou avec certaines fonctions classiques du C, les outils indispensables restent l'option `--help` des programmes, le manuel (`man <commande>`), et en dernier recours, Google !

2.3 Initiation à Git

Pour la séance machine, choisissez deux terminaux adjacents par équipe (on peut utiliser sa machine à la place d'un terminal). Chaque étudiant travaille sur son compte. Pour les monômes, on peut simplement travailler dans deux fenêtres xterm différentes dans des répertoires différents du même compte, ou travailler avec un TX et un ordinateur portable.

On commence par configurer l'outil Git :

```
emacs ~/.gitconfig
```

Le contenu du fichier `.gitconfig` (à créer s'il n'existe pas) doit ressembler à ceci :

```
[core]
        editor = votre_editeur_prefere
[user]
        name = Votre Nom
        email = Votre.Nom@ensimag.imag.fr
[diff]
        renames = true
[color]
        ui = auto
```

La section `[user]` est obligatoire. Merci d'utiliser votre vrai nom et votre adresse officielle Ensimag ici, et d'utiliser la même configuration sur toutes les machines sur lesquelles vous travaillez.

La ligne `editor` de la section `[core]` définit votre éditeur de texte préféré (par exemple, `emacs`, `vim`, ...). Cette dernière ligne n'est pas obligatoires ; si elle n'est pas présente, la variable d'environnement `VISUAL` sera utilisée ; si cette dernière n'existe pas, ce sera la variable d'environnement `EDITOR`.

la section `[diff]` et la section `[color]` sont là pour rendre l'interface de Git plus jolie.

2.3.1 Création des répertoires du projet

On va récupérer le squelette du projet en utilisant Git. Bien sûr, remplacez ldb00 par votre nom d'équipe.

```
cd
git clone ssh://ldb00@telesun.imag.fr/~/git ipsim
ls
cd ipsim
ls
```

Pour commencer, on va travailler dans le répertoire `sandbox`, qui contient deux fichiers pour s'entraîner à utiliser Git sans casser le reste du projet :

```
cd sandbox
emacs hello.c
```

A partir d'ici, les deux étudiants vont faire des choses légèrement différentes. On les appellera Alice et Bob. Il y a deux problèmes avec `hello.c` (identifiés par des commentaires). Alice résout l'un des problème, et Bob choisis l'autre. Par ailleurs, chacun ajoute son nom en haut du fichier, et sauve le résultat.

2.3.2 Gérer l'archive

Création de nouvelles révision

```
git status                  # comparaison du répertoire de
                            # travail et de l'archive.
```

On voit apparaître :

```
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   hello.c
#
```

Ce qui nous intéresse ici est la ligne « modified : hello.c » (la distinction entre « Changed but not updated » et « Changes to be committed » n'est pas importante pour l'instant), qui signifie que vous avez modifié `hello.c`, et que ces modifications n'ont pas été enregistrées dans l'archive. On peut vérifier plus précisément ce qu'on vient de faire :

```
git diff HEAD
```

Comme *Alice* et *Bob* ont fait des modifications différentes, le diff affiché sera différent, mais ressemblera dans les deux cas à :

```
diff --git a/sandbox/hello.c b/sandbox/hello.c
index a47665a..7f67d33 100644
--- a/sandbox/hello.c
+++ b/sandbox/hello.c
@@ -1,5 +1,5 @@
 /* Chacun ajoute son nom ici */
-/* Auteurs : ... et ... */
+/* Auteurs : Alice et ... */

 #include <stdio.h>
```

Les lignes commençant par '-' correspondent à ce qui a été enlevé, et les lignes commençant par '+' à ce qui a été ajouté par rapport au précédent commit. Si vous avez suivi les consignes ci-dessus à propos du fichier `.gitconfig`, vous devriez avoir les lignes supprimées en rouge et les ajoutées en vert.

Maintenant, *Alice* et *Bob* font :

```
git commit -a      # Enregistrement de l'état courant de
                   # l'arbre de travail dans l'archive locale.
```

L'éditeur est lancé, qui demande d'entrer un message de 'log'. Ajouter des lignes, et d'autres renseignent les modifications apportées à hello.c (on voit en bas la liste des fichiers modifiés). Un bon message de log commence par une ligne décrivant rapidement le changement, suivi d'une ligne vide, suivi d'un court texte expliquant pourquoi la modification est bonne.

On voit ensuite apparaître :

```
[master 2483c22] Ajout de mon nom
 1 files changed, 2 insertions(+), 12 deletions(-)
```

Ceci signifie qu'un nouveau « commit » (qu'on appelle aussi parfois « revision » ou « version ») du projet a été enregistrée dans l'archive. Ce commit est identifié par une chaîne hexadécimale (« 2483c22 » dans notre cas). On va maintenant mettre ce « commit » à disposition des autres utilisateurs.

Fusion de révisions

SEULEMENT *Bob* fait :

```
git push           # Envoyer les commits locaux dans l'archive
```

Pour voir où on en est, les deux équipes peuvent lancer la commande :

```
gitk               # afficher l'historique sous forme graphique
```

ou bien

```
git log            # afficher l'historique sous forme textuelle.
```

A PRESENT, *Alice* peut tenter d'envoyer ses modifications :

```
git push
```

On voit apparaître :

```
To ssh://ldb42@telesun.imag.fr/~/git/
 ! [rejected]        master -> master (non-fast forward)
error: failed to push some refs to 'ssh://ldb42@telesun.imag.fr/~/git/'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes before pushing again.  See the 'non-fast forward'
section of 'git push --help' for details.
```

L'expression « non-fast forward » (qu'on pourrait traduire par « absence d'avance rapide ») veut dire qu'il y a des modifications dans l'archive vers laquelle on veut envoyer nos modifications et que nous n'avons pas encore récupérer. Il faut donc fusionner les modifications avant de continuer.

L'utilisateur *Alice* fait donc :

```
git pull
```

Après quelques messages sur l'avancement de l'opération, on voit apparaître :

```
Auto-merging sandbox/hello.c
CONFLICT (content): Merge conflict in sandbox/hello.c
Automatic merge failed; fix conflicts and then commit the result.
```

Ce qui vient de se passer est que *Bob* et *Alice* ont fait des modifications au même endroit du même fichier dans les commits qu'ils ont fait chacun de leur côté (en ajoutant leurs noms sur la même ligne), et Git ne sait pas quelle version choisir pendant la fusion : c'est un conflit, et nous allons devoir le résoudre manuellement. Allez voir `hello.c`.

La bonne nouvelle, c'est que les modifications faites par Alice et Bob sur des endroits différents du fichier ont été fusionnés. Quand une équipe est bien organisée et évite de modifier les mêmes endroits en même temps, ce cas est le plus courant : les développeurs font les modifications, et le gestionnaire de versions fait les fusions automatiquement.

Un peu plus bas, on trouve :

```
<<<<<<< HEAD
/* Auteurs : Alice et ... */
=======
/* Auteurs : ... et Bob */
>>>>>>> 2483c228b1108e74c8ca4f7ca52575902526d42a
```

Les lignes entre `<<<<<<<` et `=======` contiennent la version de votre commit (qui s'appelle HEAD). les lignes entre `=======` et `>>>>>>>` contiennent la version que nous venons de récupérer par « pull » (nous avions dit qu'il était identifié par la chaîne 2483c22, en fait, l'identifiant complet est plus long, nous le voyons ici).

Il faut alors « choisir » dans `hello.c` la version qui convient (ou même la modifier). Ici, on va fusionner à la main et remplacer l'ensemble par ceci :

```
/* Auteurs : Alice et Bob */
```

Si *Alice* fait à nouveau

```
git status
```

On voit apparaître :

```
# On branch master
# Your branch and 'origin/master' have diverged,
# and have 1 and 1 different commit(s) each, respectively.
#
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#       both modified:      hello.c
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Si on n'est pas sur de soi après la résolution des conflits, on peut lancer la commande :

```
git diff    # git diff sans argument, alors qu'on avait
            # l'habitude d'appeler 'git diff HEAD'
```

Après un conflit, Git affichera quelque chose comme :

```
diff --cc hello.c
index 5513e89,614e4b9..0000000
--- a/hello.c
+++ b/hello.c
@@@ -1,5 -1,5 +1,5 @@@
  /* Chacun ajoute son nom ici */
- /* Auteurs : Alice et ... */
 -/* Auteurs : ... et Bob */
++/* Auteurs : Alice et Bob */

  #include <stdio.h>
```

(les '+' et les '-' sont répartis sur deux colonnes, ce qui correspond aux changements par rapport aux deux « commits » qu'on est en train de fusionner. Si vous ne comprenez pas ceci, ce n'est pas très grave !)

Après avoir résolu manuellement les conflits à l'intérieur du fichier, on marque ces conflits comme résolus, explicitement, avec `git add` :

```
$ git add hello.c
$ git status
# On branch master
# Your branch and 'origin/master' have diverged,
# and have 1 and 1 different commit(s) each, respectively.
#
# Changes to be committed:
#
#       modified:   hello.c
#
```

On note que `hello.c` n'est plus considéré « both modified » (i.e. contient des conflits non-résolus) par Git, mais simplement comme « modified ».

Quand il n'y a plus de fichier en conflit, il faut faire un commit (comme « git pull » nous l'avait demandé) :

```
git commit -a
```

Un éditeur s'ouvre, et propose un message de commit du type « `Merge branch 'master' of ...` », on peut le laisser tel quel, sauver et quitter l'éditeur.

(nb : si il n'y avait pas eu de conflit, ce qui est le cas le plus courant, « git pull » aurait fait tout cela : télécharger le nouveau commit, faire la fusion automatique, et créer si besoin un nouveau commit correspondant à la fusion)

On peut maintenant regarder plus en détails ce qu'il s'est passé :

```
gitk
```

Pour *comm*, on voit apparaître les deux « commit » fait par *Bob* et *Alice* en parallèle, puis le « merge commit » que nous venons de créer avec « git pull ». Pour *Bob*, rien n'a changé.

La fusion étant faite, *Alice* peut mettre à disposition son travail (le premier commit, manuel, et le commit de fusion) avec :

```
git push
```

et *Bob* peut récupérer le tout avec :

```
git pull
```

(cette fois-ci, aucun conflit, tout se passe très rapidement et en une commande)

Les deux utilisateurs peuvent comparer ce qu'ils ont avec :

```
gitk
```

ils ont complètement synchronisé leur répertoires. On peut également faire :

```
git pull
git push
```

Mais ces commandes se contenterons de répondre `Already up-to-date.` et `Everything up-to-date`.

Ajout de fichiers

A PRESENT, *Alice* crée un nouveau fichier, `toto.c`, avec un contenu quelconque.
Alice fait

```
git status
```

On voit apparaître :

```
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       toto.c
nothing added to commit but untracked files present (use "git add" to track)
```

Notre fichier `toto.c` est considéré comme « Untracked » (non suivi par Git). Si on veut que `toto.c` soit ajouté à l'archive, il faut l'enregistrer (`git commit` ne suffit pas) : `git add toto.c`
Alice fait à present :

```
git status
```

On voit apparaître :

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   toto.c
#
```

Alice fait à présent (-m permet de donner directement le message de log) :

```
git commit -m "ajout de toto.c"
```

On voit apparaître :

```
[master b1d56e6] Ajout de toto.c
 1 files changed, 4 insertions(+), 0 deletions(-)
 create mode 100644 toto.c
```

`toto.c` a été enregistré dans l'archive. On peut publier ce changement :

```
git push
```

Bob fait à présent :

```
git pull
```

Après quelques messages informatifs, on voit apparaître :

```
Fast forward
 toto.c |    4 ++++
 1 files changed, 4 insertions(+), 0 deletions(-)
 create mode 100644 toto.c
```

Le fichier `toto.c` est maintenant présent chez *Bob*.

Fichiers ignorés par Git

Bob crée à présent un nouveau fichier `temp-file.txt`, puis fait :

```
git status
```

On voit maintenant apparaître :

```
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       temp-file.txt
nothing added to commit but untracked files present (use "git add" to track)
```

Si *Bob* souhaite que le fichier `temp-file.txt` ne soit pas enregistré dans l'archive (soit « ignoré » par Git), il doit placer son nom dans un fichier `.gitignore` dans le répertoire contenant `temp-file.txt`. Concretement, *Bob* tappe la commande

```
emacs .gitignore   # ou son éditeur préféré !
```

et ajoute une ligne

```
temp-file.txt
```

puis sauve et on quitte. Pour que tous les utilisateurs de l'archivent bénéficient du même fichier `.gitignore`, *Bob* fait :

```
git add .gitignore
```

Bob fait a nouveau

```
git status
```

On voit apparaître :

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   .gitignore
#
```

Quelques remarques :

- Le fichier `temp-file.txt` n'apparaît plus. C'était le but de la manoeuvre. Une bonne pratique est de faire en sorte que « git status » ne montre jamais de « Untracked files » : soit un fichier doit être ajouté dans l'archive, soit il doit être explicitement ignoré. Cela évite d'oublier de faire un « git add ».
- En général, on met dans les `.gitignore` les fichiers générés (*.o, fichiers exécutables, ...), ce qui est en partie fait pour vous pour ce projet.
- Le fichier `.gitignore` vient d'être ajouté (ou bien il est modifié si il était déjà présent). Il faut à nouveau faire un commit pour que cette modification soit disponible pour tout le monde.

A ce stade, vous devriez avoir les bases pour l'utilisation quotidienne de Git. Bien sûr, Git est bien plus que ce que nous venons de voir, et nous encourageons les plus curieux à se plonger dans le manuel utilisateur et les pages de man de Git pour en apprendre plus. Au niveau débutant, voici ce qu'on peut retenir :

Les commandes

git commit -a enregistre l'état courant du répertoire de travail.

git push publie les commits

git pull récupère les commits publiés

git add, git rm et git mv permettent de dire à Git quels fichiers il doit gérer.

Conseils pratiques

- Ne *jamais* s'échanger des fichiers sans passer par Git (email, scp, clé USB), sauf si vous savez *vraiment* ce que vous faites.
- Toujours utiliser `git commit` avec l'option `-a`.
- Faire un `git push` après chaque `git commit -a`, sauf si on veut garder ses modifications privées. Il peut être nécessaire de faire un `git pull` avant un `git push` si des nouvelles révisions sont disponibles dans l'archive partagée.
- Faire des `git pull` régulièrement pour rester synchronisés avec vos collègues. Il faut faire un `git commit -a` avant de pouvoir faire un `git pull` (ce qui permet de ne pas mélanger modifications manuelles et fusions automatiques).
- Ne faites jamais un « `git add` » sur un fichier binaire généré : si vous les faites, attendez-vous à des conflits à chaque modification des sources ! Git est fait pour gérer des fichiers sources, pas des binaires.

(quand vous ne serez plus débutants, vous verrez que la vie n'est pas si simple, et que la puissance de Git vient de `git commit` sans `-a`, des `git commit` sans `git push`, ... mais chaque chose en son temps !)

Chapitre 3

Le langage d'assemblage ASM

Le langage d'assemblage ASM présenté ici est un sous-ensemble de celui de GNU, volontairement moins permissif. On se contente ici de présenter la syntaxe de manière intuitive. Pour le projet simulateur, l'assembleur donné par les enseignants respecte cette syntaxe.

Si l'on ignore les portions de texte en commentaire (voir syntaxe des commentaires ci-dessous), un programme se présente comme une suite de lignes séparées par des caractères de fin de ligne. Chaque ligne peut contenir plusieurs définitions d'étiquette, et éventuellement soit une instruction avec ses paramètres, soit une directive avec son paramètre. Une ligne peut être vide (ce qui permet d'aérer le texte).

3.1 Les commentaires

Il y a deux types de commentaires : un commentaire commence par le caractère '#' et se termine au premier caractère de fin de ligne ou commence par un '/*' et se termine au premier '*/'.

Exemple

```
# Ceci est un commentaire se terminant à la fin de la ligne
/* Ceci est aussi un commentaire, mais sur
   plusieurs lignes */
```

3.2 Les étiquettes

Une étiquette permet de nommer une adresse mémoire. Les étiquettes peuvent servir d'opérande à une instruction ou à une directive. Syntaxiquement, une étiquette est une suite de caractères formée de lettres, de chiffres, des caractères '.' ou '_', et ne commençant pas par un chiffre.

Les définitions d'étiquettes ont lieu en début de ligne. Syntaxiquement, une définition d'étiquette est de la forme le nom de l'étiquette suivi du caractère ':'. L'adresse désignée est celle du prochain octet codé lors de l'assemblage.

Exemple

```
Lab1: Lab2:
Lab3: movl $0,%eax  # les 3 étiquettes repèrent l'adresse de cette instruction
```

Une étiquette ne peut être définie qu'une seule fois par fichier assembleur. Sa valeur lors de l'exécution est égale à son adresse d'implantation dans la mémoire après le chargement. Elle dépend donc de la section dans laquelle elle est définie et de sa position dans cette section (cf. section 3.5.1).

Il y a trois étiquettes prédéfinies : '`.text`' '`.data`' et '`.bss`' qui désignent les adresses respectives de chacune de ces sections (voir section 3.5.1).

3.3 Les nombres littéraux

Un nombre littéral est une suite de chiffres hexadécimaux précédée de '0x', ou une suite de chiffres décimaux.

```
0xFdCba987  # nombre hexadécimal
123456789   # nombre décimal
```

En général, les littéraux hexadécimaux sont interprétés comme des nombres non-signés, et les littéraux décimaux comme des nombres signés en complément à 2 (voir chapitre 4). Cette interprétation est en particulier utilisée par l'assembleur pour vérifier le non-débordement des littéraux dans leur contexte d'utilisation. Par exemple, les littéraux signés sur 32 bits doivent avoir des valeurs comprises entre -2147483648 et 2147483647, alors que les littéraux non signés doivent avoir des valeurs entre 0 et 4294967295. La seule exception à cette règle porte sur les littéraux utilisés dans le mode d'adressage avec base et déplacement où le déplacement est toujours considéré comme un signé sur 32 bits, même si le littéral du déplacement est hexadécimal. [1]

Il est possible d'appliquer l'opérateur unaire '-' pour désigner l'opposé d'un littéral signé ou non-signé. Dans le cas non-signé, le nombre alors désigné est toujours considéré comme non-signé et correspond au calcul de l'opposé sur 32 bits.

3.4 Les instructions machines

Une instruction est de la forme un champ opération, suivi éventuellement par un ou deux champs opérandes séparés par le caractère ','.

3.4.1 Le champ opération

Pour les instructions pouvant opérer sur plusieurs tailles d'opérandes, le champ opération est formé du nom de l'instruction en minuscules suivi immédiatement d'une des lettres l, w ou b selon que l'opération porte sur un double mot (32 bits), un mot (16 bits) ou un octet. Pour les autres opérations, le champ opération porte le nom donné par le constructeur sans suffixe. La liste complète des instructions est donnée chapitre 5.

3.4.2 Les champs *opérandes*

Pour les opérations ayant deux opérandes, la spécification de l'opérande source précède toujours celle de l'opérande destination. L'opérande source est le plus souvent un registre ou une valeur immédiate quand l'autre est un registre ou une donnée en mémoire.

[1] Cette interprétation n'est pas celle de l'assembleur GNU qui interprète les littéraux suivant leur représentation en complément à 2 sur 32 bits. L'assembleur GNU ne vérifie donc pas vraiment les débordements de littéraux.

Syntaxe des champs opérandes On présente ici brièvement la syntaxe des opérandes. Les différents modes d'adressages sont décrits au chapitre 4. Pour chaque opération, sa spécification (voir section 5.2) précise les modes d'adressage qui sont permis pour ses opérandes.

- Les noms de registres doivent être écrits en minuscules, et précédé du caractère '%'. Par exemple, '%eax' '%ax' '%ah' '%al' désignent respectivement les quatre registres EAX, AX, AH et AL.
- Un opérande immédiat est de la forme '$' suivi d'un littéral ou d'une étiquette. Par exemple :

```
movl $-0xff,%eax   # eax = 0xffffff01
movb $0xf0,%ah
movl $Lab1, %eax
```

- Pour l'adressage direct, on donne le nom de l'étiquette qui repère la donnée. Par exemple :

```
.section .data
xint: .long 0x12345678
.section .text
        movl xint, %eax # eax = 0x12345678
        movb xint, %bl  # bl = 0x78 en little endian
```

- Pour l'adressage indirect avec déplacement : on écrit D`(%`R`)` où R est le nom de l'un des registres 32 bits permis et D la valeur du déplacement. Le déplacement, s'il est nul, peut être omis. Les déplacements sont des entiers signés donnés sous forme décimale ou hexadécimale.
- Pour l'adressage relatif à une instruction de branchement, on donne le nom d'une étiquette.
- Pour l'adressage indirect dans une instruction de branchement, on écrit $*M$ où M un mode d'adressage direct ou indirect précédant autorisé.

3.5 Les directives

Une directive commence toujours par un point '.'. Il y a trois familles de directives : les directives de sectionnement du programme, les directives de définition de données et les autres.

3.5.1 Directive de sectionnement

Sous Unix, les processus ont accès à 2 zones mémoires : une zone de *programme* partagée par les processus exécutant le même programme, en lecture seule pour ces processus, et une zone *données* propre à chaque processus, modifiable par ces processus. La zone *programme* contient donc en général les instructions et les constantes. La zone *donnée* contient en général les données modifiables par chaque processus. Pour pouvoir profiter de cette possibilité, l'assembleur permet de définir trois sections :

- une section `.text` pour définir la zone *programme*.
- une section `.data` pour définir les données initialisées de la zone *donnée*.
- une section `.bss` pour déclarer les données non initialisées de la zone *donnée*. Ces données ne prennent ainsi pas de place dans le fichier binaire du programme. Elles seront effectivement allouées au moment du chargement du processus. Elles seront initialisées à zéro.

La directive de section `.section` suivie d'un nom de section (`.text`, `.data` ou `.bss`) indique à l'assembleur d'assembler les lignes suivantes dans les sections correspondantes.

A chacune de ces sections correspond une étiquette prédéfinie de même nom, utilisable dans les opérandes des instructions ou les paramètres de directives.

3.5.2 Les directives de définition de données

On distingue les déclarations de données non initialisées qui ont généralement lieu dans la zone `.bss` des déclarations de données initialisées qui ont généralement lieu dans les zones `.text` et `.data`.

Déclaration des données non initialisées : `.skip` *taille*

La directive `.skip` permet de réserver un nombre d'octets égal à *taille* à l'adresse *étiquette*.

```
toto: .skip 13
```

En dehors de la section `.bss`, la zone mémoire correspondant à une directive `.skip` est remplie de zéros.

Déclaration de données initialisées de nombres

Les directives suivantes permettent de réserver 1, 2 ou 4 octets et d'initialiser cette zone mémoire à partir d'un nombre (en little endian).

`.byte` ***valeur*** *valeur* est un littéral signé ou non-signé sur 8 bits. Par exemple, les lignes ci-dessous permettent de réserver deux octets avec les valeurs initiales -4, -1 sous forme hexadécimale. Le premier octet est à l'adresse `Tabb` de la mémoire, l'autre à l'adresse `Tabb+1`.

```
Tabb:   .byte -4
        .byte 0xff
```

`.word` ***valeur*** *valeur* est un littéral signé ou non-signé sur 16 bits. Par exemple, la ligne suivante permet de réserver 2 mots de 16 bits avec la valeur initiale 32767 à l'adresse `Tabw` de la mémoire et 255 à l'adresse `Tabw+2`.

```
Tabw: .word 0x7fff
      .word 255
```

`.long` ***valeur*** *valeur* peut être soit un littéral signé ou non-signé sur 32 bits, soit une étiquette (une adresse dans l'une des sections). Par exemple, la ligne suivante permet de réserver à l'adresse `Tabf` de la mémoire un double mot et de l'initialiser avec l'adresse du début de la zone `.bss`

```
Tabf: .long .bss
```

Déclaration de données initialisées de chaînes de caractères

Syntaxe des chaînes de caractères Une chaîne de caractères est une suite de caractères entre un '`"`' et le premier '`"`' non précédé d'un nombre impair de '\' consécutifs. La suite formée des 2 caractères '\' suivi de 'n' est interprétée comme un caractère de fin de ligne. De même, une suite formée de '\' suivi d'un caractère c, où c est différent de 'n', est interprétée comme le caractère c (en particulier, c peut valoir '`"`' ou '\').

Exemple La chaîne sur la ligne suivante :

```
"/* ceci n'est pas un commentaire */\nle caractere \"\\\" s'appelle backslash\n"
```

est interprétée comme la chaîne :

```
/* ceci n'est pas un commentaire */
le caractere "\" s'appelle backslash
```

`.string` ***str*** L'assembleur recopie en mémoire la suite de codes ascii correspondant à l'interprétation de la chaîne *str*. Le caractère de fin de chaîne (octet de code 0) est lui-aussi recopié.

`.ascii` ***str*** Fait la même chose que `.string`, sans recopier le caractère de fin de chaîne. Par exemple, la réservation suivante :

```
format: .string "Format printf de sortie d'entiers %08d\n"
```

produit le même codage binaire que la suite de réservations :

```
format:  .ascii "Format printf de sortie d'entiers %08"
         .byte 144     # code ascii de 'd'
         .byte 10      # code ascii de '\n'
         .byte 0       # caractère de fin de chaine
```

3.5.3 Directive d'alignement

Sur un PC, on peut adresser directement chaque octet de la mémoire (cf. chapitre 4). Néanmoins, lorsqu'on lit ou qu'on écrit des entiers 32 bits en mémoire, il est plus efficace de travailler avec des adresses qui sont des multiples de 4 octets. La directive `.align` *nombre* indique à l'assembleur de placer l'adresse qui suit sur un multiple de *nombre* octets. Par exemple, la ligne suivante garantit que l'entier long d'étiquette `monentier` sera à une adresse multiple de 4 octets.

```
.align 4
monentier: .long 1
```

3.5.4 Directive d'exportation des noms

Par défaut, les étiquettes définies localement sont considérées comme locales et ne sont pas visibles par d'autres unités de compilation lors de l'édition de liens. La directive `.globl` *etiq* indique que l'étiquette *etiq* doit être exportée. Typiquement, la fonction principale du programme s'appelle `main` et doit être exportée.

```
.globl main
  # ...
main: # ...
```

Les étiquettes non définies localement qui apparaissent dans le programme sont globales.

Chapitre 4

Représentation et adressage des données de Pentium

On donne dans ce chapitre une description succincte des modes d'adressages du sous-ensemble considéré du Pentium : la machine Pentium.

4.1 Définitions et notations

Cette section rappelle quelques définitions et notations utiles.

Octet/mot/long mot Un *octet* est une suite de 8 bits qui constitue la plus petite entité que l'on peut adresser sur la machine (voir figure 4.1). La concaténation de deux octets forme un *mot* de 16 bits, et la concaténation de quatre octets (ou de deux mots) forme un *long mot* de 32 bits. Les bits sont numérotés de la droite (poids faible) vers la gauche (poids fort) de 0 à 7 pour l'octet, de 0 à 15 pour un mot et de 0 à 31 pour un long mot.

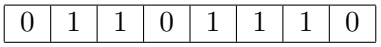

FIG. 4.1 – Octet

Représentation hexadécimale d'un octet/mot/long mot On représente par $0x$ij la valeur d'un octet dont les 4 bits de poids fort valent i et les 4 bits de poids faible j (avec i, j $\in ([0...9, A...F])$. Par exemple, la valeur de l'octet de la figure 4.1 s'écrit $0x6e$ en hexadécimal. Pour un mot ou un long mot, on aura respectivement 4 ou 8 chiffres hexadécimaux, chacun représentant 4 bits.

Codage binaire d'un entier non signé Quand on parle d'un entier non signé codé sur n bits ou plus simplement d'un entier codé sur n bits, il s'agit de sa représentation en base 2 sur n bits, donc d'une valeur entière comprise entre 0 et $2^n - 1$. Un entier codé sur un octet a donc une valeur comprise entre 0 et 255 correspondant aux images binaires $0x00$ à $0xff$, un entier codé sur un mot a une valeur comprise entre 0 et 65535 correspondant aux images binaires $0x0000$ à $0xffff$ et un entier codé sur un long mot a une valeur comprise entre 0 et 4294967295 correspondant aux images binaires $0x00000000$ à $0xffffffff$. Les adresses du processeur de la machine Pentium sont des entiers non signés sur 32 bits.

Codage binaire d'un entier signé Les entiers signés sont représentés en complément à 2 : le codage sur n bits du nombre i est la représentation en base 2 sur n bits de $2^n + i$, si $-2^{n-1} \leq i \leq -1$, et de i, si $0 \leq i \leq 2^{n-1} - 1$. Un entier signé codé sur un octet est compris entre -128 à 127 correspondant aux images binaires $0x80$ à $0x7f$. Un entier signé sur un mot est compris entre -32768 et 32767 correspondant à l'intervalle binaire $0x8000$ à $0x7fff$. Enfin, un entier signé sur un long mot est compris entre -2147483648 et 2147483647 correspondant à l'intervalle binaire $0x80000000$ à $0x7fffffff$. On remarque que le bit de plus fort poids d'un octet/mot/long mot représentant un entier négatif est toujours égal à 1 alors qu'il vaut 0 pour un nombre positif (c'est le bit de signe).

4.2 La mémoire

La mémoire centrale adressable par le processeur Pentium peut être assimilée à un tableau de 4 Goctets[1] (cf. Figure 4.2). La plus petite entité qui peut être désignée (on dit également adressée) dans ce tableau est l'octet (byte en anglais). Le nom d'un octet, aussi appelé adresse effective, est la valeur de l'index représentant son rang dans le tableau. Les adresses effectives varient de 0 à $2^{32} - 1$ et demandent donc 32 bits pour être écrites en base 2 (représentation hexadécimale de $0x0$ à $0xffffffff$). Deux octets adjacents peuvent être considérés comme formant un mot. Son adresse est celle de son octet de poids faible. L'octet de poids fort d'un mot est à l'adresse de son octet de poids faible plus un. Quatre octets consécutifs forment un double mot ou "mot long" ou encore "long" dont l'adresse est celle de son octet de poids faible, les octets aux adresses respectives +1, +2, +3 ayant un poids de plus en plus fort. On parle dans ce cas de machine *little endian*.

<table>
<tr><th>Poids fort</th><th></th><th></th><th>Poids faible</th><th>Adresse</th></tr>
<tr><td></td><td></td><td></td><td></td><td>4 294 967 292</td></tr>
<tr><td></td><td></td><td></td><td></td><td>4 294 967 288</td></tr>
<tr><td colspan="5">...</td></tr>
<tr><td></td><td></td><td></td><td></td><td>12</td></tr>
<tr><td colspan="4">Double mot</td><td>8</td></tr>
<tr><td colspan="2">Mot de poids fort</td><td colspan="2">Mot de poids faible</td><td>4</td></tr>
<tr><td>Octet 3</td><td>Octet 2</td><td>Octet 1</td><td>Octet 0</td><td>0</td></tr>
<tr><td>+3</td><td>+2</td><td>+1</td><td>+0</td><td></td></tr>
</table>

Fig. 4.2 – Mémoire Pentium

4.3 Les registres

Le microprocesseur de la machine Pentium fournit des registres de 8, 16 et 32 bits (Figure 4.3). Les registres de 8 bits sont dénotés respectivement AH, AL, BH, BL, CH, CL, DH, DL ; ceux de 16 bits : AX, BX, CX et DX et enfin ceux de 32 bits EAX, EBX, ECX, EDX, EBP, ESP, EDI, ESI, EIP et EFLAGS. Certains de ces registres : EAX, EBX, ECX, EDX, EBP, EDI et ESI ont des fonctions multiples et les autres peuvent avoir des fonctions spécialisées

[1] 4 Goctets = 4 294 967 296 octets (1 G = 1024 x 1024 x 1024 = 1 073 741 824)

Mnémo.	31 16	15 0		Nom usuel
EAX		AX		Accu
		AH	AL	
EBX		BX		Base Index
		BH	BL	
ECX		CX		Count
		CH	CL	
EDX		DX		Data
		DH	DL	
ESP				Stack Pt.
EBP				Base Pt.
EDI				Dest. Index
ESI				Source Index
EIP				Instruction Pt.
EFLAGS				Flag

FIG. 4.3 – Registres de la machine Pentium

que nous présenterons avec les instructions qui les utilisent de manière spécifique. EAX, EBX, ECX, EDX peuvent être référencés comme des registres de 32 bits, comme des registres de 16 bits (AX, BX, CX, DX) ou comme des registres de 8 bits (AH, AL, BH, BL, CH, CL, DH, DL). Si on utilise les registres de 16 ou 8 bits, seule cette portion du registre de 32 bits associé sera éventuellement modifiée, la partie restante n'étant pas modifiée par l'opération. EDI, ESI, EBP et ESP sont des registres de 32 bits uniquement. EIP et EFLAGS ne peuvent pas être utilisés dans les modes d'adressage. EIP, ESP et EFLAGS sont des registres spécialisés dont les fonctions sont :

- **EIP** (*Extended Instruction Pointer*) : ce registre contient, à la fin d'une instruction, l'adresse de l'instruction suivante à exécuter. Il peut être modifié par les instructions de saut (`jmp`, `jcc`) ou d'appel et de retour (`call`, `ret`) d'un sous-programme.
- **ESP** (*Extended Stack Pointer*) : ce registre est supposé contenir une adresse dans une zone de la mémoire gérée en pile. Par convention, cette adresse est le sommet de la pile. Les instructions call, ret, push et pop modifient sa valeur.
- **EFLAGS** : ce registre est le registre d'état du microprocesseur. Nous ne décrivons que les indicateurs que nous utiliserons (figure 4.4)

31 12	**11**				**7**	**6**						**0**
	O				**S**	**Z**						**C**

FIG. 4.4 – Indicateurs utiles du registre EFLAGS

  - **C** : Carry (bit 0 du registre EFLAGS). Il contient la retenue après une addition ou une soustraction.
  - **Z** : Zéro (bit 6 du registre EFLAGS). Il contient 1 si le résultat d'une opération arithmétique ou logique est nul et 0 sinon.
  - **S** : Signe (bit 7 du registre EFLAGS). Il prend la valeur du bit de signe du résultat après l'exécution d'une opération arithmétique ou logique.
  - **O** : Overflow (bit 11 du registre EFLAGS). Il indique un débordement après une opération dont les opérandes sont considérées comme de valeurs signées. Pour des

opérations sur des valeurs non signées il est ignoré du programmeur.

4.4 Les modes d'adressage généraux

La plupart des instructions fournies par les microprocesseurs auxquels nous nous intéressons opèrent avec deux opérandes, dont l'un est contenu dans un registre et l'autre dans un registre ou dans la mémoire (données ou programme). On appelle mode d'adressage d'un opérande, la méthode qu'utilise le processeur pour calculer l'adresse de l'opérande. S'il s'agit d'un opérande qui est un registre, il ne s'agit pas à proprement parler d'adresse en mémoire mais de nom. S'il s'agit d'un opérande en mémoire, l'adresse sera la valeur de l'index auquel se trouve cet opérande dans le tableau (cf. section 4.2). Nous faisons maintenant la présentation des modes d'adressage disponibles.

Pour illustrer la présentation des modes d'adressage des données en mémoire, nous allons utiliser les trois instructions : `movb`, `movw`, `movl` qui permettent de copier respectivement l'octet, le mot ou le double mot de l'opérande source, le premier spécifié dans l'instruction assembleur, dans respectivement l'octet, le mot ou le double mot de l'opérande destination. Cette instruction ne permet pas aux deux opérandes d'être des données en mémoire, donc l'un des opérandes est soit un registre soit une valeur immédiate. Nous allons nous intéresser à toutes les façons de calculer soit la source soit la destination.

4.4.1 Adressage registre direct

Dans ce mode, la valeur de l'opérande est dans un registre et l'opérande est désigné par le nom du registre en question qui peut être l'un des noms de registres donnés dans le chapitre précédent. Selon que l'opération porte sur un octet, un mot ou un mot long, on doit utiliser une portion du registre de 8, 16 ou 32 bits. Bien entendu, cette taille doit correspondre à celle spécifiée par l'instruction (`b` pour octet, `w` pour un mot et `l` pour un mot long).

En assembleur (cf. le chapitre 3), les registres sont désignés dans les instructions par leur nom précédé du caractère %. D'un point de vue syntaxique, dans l'écriture d'une instruction mettant en jeu une source et une destination, **la source précède toujours la destination**.

Exemples

```
movl %esp, %ebp /* Copie le contenu de esp dans ebp */
movw %ax, %bx   /* Copie le contenu de ax dans bx */
movb %al, %ah   /* Copie le contenu de al dans ah */
```

4.4.2 Adressage immédiat

Dans ce mode, c'est la valeur de l'opérande qui est directement fournie dans l'instruction[2]. Une valeur immédiate est toujours précédée en assembleur GNU du caractère '$'. La valeur immédiate peut être fournie sous forme symbolique, sous forme d'un nombre décimal signé, ou sous forme d'une suite de chiffres hexadécimaux précédée des deux caractères `0x`

Exemples

[2]L'adresse de l'opérande est donc ce cas l'adresse de la mémoire qui suit immédiatement celle du code de l'instruction

```
movb $0xff, %al   /* al := -1 */
movw $0xffff, %ax /* ax := -1 */
movl $-1, %eax    /* eax := -1 */
movl $toto, %eax  /* eax := valeur du symbole toto */
```

Ce dernier exemple est intéressant car toto peut être l'adresse d'une donnée ou d'une instruction. Dans ce cas, elle n'est connue qu'au chargement du programme en mémoire.

Attention N'oubliez pas le caractère $ devant l'opérande immédiat sous peine de le voir mal interprété, comme un opérande adressé directement.

4.4.3 Adressage direct

Dans ce mode, c'est l'adresse de l'opérande dans la mémoire qui est fournie dans le champ opérande de l'instruction. La valeur effective de cette adresse n'est en général[3] connue qu'au moment du chargement du programme. Notre assembleur n'autorisera donc que les noms d'étiquette comme opérande de ce mode d'adressage.

Illustrons ceci par un exemple. On suppose qu'à partir de l'adresse d'étiquette *etiq* de la mémoire, on a rangé les quatre octets : $0xaa$, $0xbb$, $0xcc$ et $0xdd$. Donc, comme nous l'avons déjà vu plus haut, l'octet à l'adresse *etiq* est $0xaa$, le mot à l'adresse *etiq* est $0xbbaa$ et le double mot à l'adresse *etiq* est $0xddccbbaa$. On peut le vérifier en exécutant les instructions ci-dessous et en visualisant le résultat avec un metteur au point.

```
movb etiq,%al  /* affecte 0xaa au registre octet al */
movw etiq,%ax  /* affecte 0xbbaa au registre mot ax */
movl etiq,%eax /* affecte 0xddccbbaa au registre eax */
```

4.4.4 Adressage indirect registre

Dans ce mode l'adresse de l'opérande en mémoire est contenue dans le registre. La syntaxe Gnu de ce mode d'adressage est `(%`*`REGISTER`*`)` où *REGISTER* est l'un quelconque des registres EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP.

Par exemple, après l'instruction "`movl $`*etiq*`, %ebx`" qui place l'adresse d'étiquette *etiq* dans le registre EBX, les instructions "`movb (%ebx),%al`", "`movw (%ebx),%ax`" et "`movl (%ebx),%eax`" donnent les mêmes résultats que les trois instructions de l'exemple donné en 4.4.3.

4.4.5 Adressage indirect avec base et déplacement

Dans ce mode, interviennent un registre appelé *registre de base*, et une constante signée appelée *déplacement*. Le registre peut être, comme précédemment, l'un des registres EAX, EBX, ECX, EDX, EDI, ESI, EBP et ESP. La syntaxe associée par l'assembleur à ce mode est *`Dep`*`(%`*`REGISTER`*`)`.

Pour calculer l'adresse de l'opérande, le processeur ajoute au contenu du registre de base *REGISTER* la valeur signée sur 4 octets du déplacement. Par exemple, après l'exécution de l'instruction "`movl $`*etiq*`,%ebx`", 6(%ebx) repérera l'opérande à l'adresse d'étiquette *etiq* plus 6. Par conséquent, les instructions "`movb 6(%ebx),%al`", "`movw 6(%ebx),%ax`" et "`movl 6(%ebx),%eax`" ont un effet équivalent aux trois instructions de l'exemple de 4.4.4.

[3]Les exceptions concernent par exemple les programmes "système" pilotant un périphérique, et qui communiquent avec lui via des adresses fixes connues par construction.

4.5 Modes d'adressage réservés aux instructions de saut et d'appel de fonction

Les modes d'adressage suivants ne sont utilisables que dans les instructions `jcc`, `jmp` et `call`.

4.5.1 Adressage de branchement relatif

Le champ opérande contient sur 8, ou 32 bits une valeur signée représentant le déplacement, positif ou négatif, qu'il faut ajouter au registre EIP pour atteindre l'instruction que l'on veut exécuter après l'exécution de l'instruction de saut. Attention quand le déplacement est ajouté au registre EIP, ce dernier pointe sur l'instruction qui suit l'instruction `jmp`. En assembleur, le champ opérande est fourni en général sous forme symbolique et est le nom de l'étiquette qui repère dans le code l'instruction que l'on veut exécuter après l'instruction de saut.

Exemple

```
Ici :   jmp Ici /* Attention : déplacement généré sur 8 bits : il vaut -2 */
                /* il vaudrait -5 si on le générait sur 32 bits */
```

4.5.2 Adressage de branchement indirect (seulement pour `jmp` et `call`)

Le champ opérande désigne une adresse ou un registre qui contient l'adresse absolue du branchement sur 32 bits. En assembleur, le champ opérande s'écrit en faisant précéder d'un `*` les formes d'adressage direct et indirect précédentes. Par exemple, si *REG* est l'un des registres EAX, EBX, ECX, EDX, ESP, EBP, EDI, ESI, et si l'opérande est `*%`*REG*, *REG* doit contenir l'adresse de l'instruction à exécuter ensuite.

Exemple

```
.section .data   /* déclaration d'une zone de données */
AdIci: .long Ici
.section .text   /* déclaration d'une zone de programme */
Ici:
        pushl $Ici      /* empile l'adresse d'Ici */
        jmp *(%esp)     /* remonte à Ici ; boucle infinie */
        jmp *AdIci      /* remonte à Ici ; boucle infinie */
```

4.6 Conventions pour les appels de fonction

Il est nécessaire de suivre les conventions usuelles pour les appels de fonctions (instructions `call` et `ret`) pour pouvoir utiliser des fonctions écrites par d'autres (notamment les fonctions de la librairie C) ou rendre accessibles des fonctions à d'autres (notamment depuis des programmes écrits en C). Mais si vous écrivez une fonction que vous n'appellerez que depuis un de vos programmes assembleurs, vous pouvez adopter une convention ad-hoc. Ces conventions sont décrites dans le document donné en cours de logiciel de base [1].

Chapitre 5

Les instructions de Pentium

5.1 Format général des instructions du Pentium

Dans cette section, on décrit le format binaire des instructions du microprocesseur Pentium, c'est-à-dire le format dans lequel elles doivent être traduites par l'assembleur. Dans ce format, les instructions contiennent trois types d'information : l'identification de la fonction à exécuter, la taille du ou des opérandes de la fonction, et la valeur ou l'emplacement de ces opérandes.

Le format général d'une instruction est donné Figure 5.1. Il sera détaillé dans les sections suivantes. D'une façon générale, une instruction peut comporter : un octet de préfixe optionnel, un ou deux octets associés au code opération de l'instruction, suivi si c'est nécessaire de la spécification du mode d'adressage des opérandes consistant en un octet appelé *ModR/M* suivi éventuellement d'un octet d'index (*SIB* = *Scalable Index Byte*). Ces champs peuvent être suivis d'une valeur de déplacement codée sur 1 ou 4 octets et/ou d'une valeur immédiate codée sur 1, 2 ou 4 octets.

Préfixe	Code OP	*ModR/M*	Index *SIB*	Déplacement	Valeur immédiate
1 octet optionnel	1 ou 2 octets	1 octet si nécessaire	1 octet si nécessaire	déplacement sur 1 ou 4 octets ou rien	donnée immédiate sur 1, 2 ou 4 octets ou rien

FIG. 5.1 – Format général d'une instruction

5.1.1 Préfixe des instructions

Ce champ n'est utilisé que lorsqu'on veut utiliser l'un des registres de 16 bits (AX, BX, CX etc..) dans une opération ou plus généralement quand l'opération opère sur un mot de 16 bits plutôt que sur un double mot. Dans ce cas le préfixe est toujours présent et prend la valeur hexadécimale $0x66$.

5.1.2 Code opération

Le champ code opération peut être codé sur 8, 16, 12 ou 19 bits. Dans ces 2 derniers cas, les premiers (un ou deux) octets constituent la *partie primaire* du code opération et les trois bits restant du code opération, codés dans les bits 3, 4 et 5 de l'octet *ModR/M* (cf. figure 5.2), constituent son *extension*. Pour spécifier que l'opération codée porte sur un mot (dans ce cas l'instruction est précédée du préfixe $0x66$) ou un double mot, le codage du code opération comprend un bit appelé bit w qui vaut 1. Si ce bit vaut 0, l'opération porte sur un octet. Le bit

w est le plus souvent le bit 0 du champ code opération quand celui-ci est codé sur un seul octet (voir figure 7.4).

Si l'instruction utilise une donnée immédiate sur 8 bits qui doit être étendue à 16 ou 32 bits avant que l'opération soit effectuée, l'extension peut être effectuée avec ou sans recopie du bit de signe. Le mode choisi est sélectionné par le bit appelé s du code opération qui vaut 0 si on opère sans extension du bit de signe et 1 sinon (voir figure 7.4).

Pour les instructions à deux opérandes, l'avant-dernier bit du code opération indique s'il s'agit du registre de l'opérande source ou de l'opérande destination (voir figure 7.4).

Pour les instructions conditionnelles (branchement conditionnel ou positionnement si condition) le champ condition (*tttn*) est codé pour spécifier la condition que l'on teste. Les bits *ttt* spécifient la condition et le bit `n` spécifie si on s'intéresse à la condition (n = 0) ou à sa négation ($n = 1$). Pour un code opération dont la partie primaire est codée sur un seul octet , le champ *tttn* est situé dans les bits 3, 2, 1 et 0 du code opération. Pour un code opération dont la partie primaire du code opération est codée sur deux octets, le champ *tttn* est dans les bits 3, 2, 1 et 0 du second octet du code opération. La figure 7.1 page 57 contient les valeurs de *tttn* et leur correspondance en terme de la condition testée.

Pour certaines instructions qui n'ont qu'une opérande, si celle-ci est un registre, on additionne le code du registre au code opération : on économise ainsi l'octet ModR/M. C'est le cas de l'instruction `pushl`. Par exemple,

```
pushl %ecx
```

peut être codée sur un seul octet : `0x51` obtenu en additionnant le code-op `0x50` de `pushl` avec `1` le code de ECX. (voir figure 7.5).

5.1.3 Octet *ModR/M*

Les instructions qui utilisent un opérande en mémoire ont un octet *ModR/M* qui suit la partie primaire du code opération et qui sert à spécifier le mode d'adressage de l'opérande. Cette partie *ModR/M* est constituée de trois champs, comme le montre la figure 5.2 :

- le champ *Mod* qui, combiné avec le champ *R/M*, permet de coder les modes d'adressage permis ;
- le champ *Reg/Op* qui peut spécifier soit un nom de registre, soit la partie secondaire ou extension du code opération ;
- le champ *R/M* qui spécifie généralement un nom de registre utilisé dans un mode d'adressage spécifié par *Mod*.

7 6	5 3	2 0
Mod	Reg/Op	R/M

FIG. 5.2 – Format de l'octet *ModR/M*

Lorsque *Mod* vaut 3 (en décimal), *R/M* désigne toujours un registre utilisé en adressage direct (ou indirect de branchement pour les instructions JMP et CALL). Dans les autres cas (*Mod* entre 0 et 2), *R/M* ne désigne pas toujours un registre. Mais, en général, si *Mod* vaut 0 (respectivement 1 et resp. 2), *R/M* désigne un registre utilisé en adressage indirect sans déplacement (respectivement avec déplacement codé sur 8 bits et resp. déplacement codé sur 32 bits). Les exceptions à cette règle concernent les valeurs 4 et 5 (en décimal) de *R/M* qui correspondent aux registres ESP et EBP. Le détail de ces cas est donné en annexe dans la figure 7.3.

5.1.4 Octet d'index variable (*Scalable Index Byte*)

Cet octet est utilisé pour un mode d'adressage plus général que l'adressage avec base : l'adressage avec base et index, qui n'est pas spécifié dans ce projet. Il n'est donc pas considéré, sauf dans le cas particulier suivant qui sera respecté de façon à ce que le logiciel fonctionne correctement pour tous les programmes n'utilisant pas le mode d'adressage avec base et index. Ce cas particulier se présente quand les trois bits de poids faible (champ *R/M*) de l'octet *ModR/M* prennent la valeur 4 : cette valeur signifie normalement pour la machine que cet octet est suivi par un octet *SIB* et donc que le mode d'adressage est le mode avec base et index. Cela interdirait donc le choix du registre ESP comme registre indirect ou indirect avec déplacement puisque 4 est son codage. En fait, on a le droit aux trois modes d'adressage *(%esp)*, *d8(%esp)* et *d32(%esp)*. Dans ce cas, l'octet *ModR/M* est suivi du registre *SIB* qui prend la valeur $0x24$.

5.1.5 Déplacements et données immédiates

Certains modes d'adressage utilisent un déplacement qui est codé immédiatement à la suite soit du champ *ModR/M*, soit du champ *SIB* s'il est présent. Le déplacement, s'il est requis, peut être codé sur 1 ou 4 octets.
Si l'instruction spécifie un opérande immédiat, son codage suit le champ déplacement éventuel. Le codage d'une donnée immédiate peut comporter 1, 2 ou 4 octets.

5.1.6 Exemple de codage d'une instruction

Pour coder les instructions et leurs opérandes, il faut s'appuyer sur les tables de la section 5.2, et des figures 7.3, 7.4 et 7.5. Vous pouvez aussi vous aider des outils gnu : `as` (assembleur) et `objdump` (désassembleur). Par exemple, on peut copier la ligne suivante dans un fichier `foo.s` :

```
cmpl $64, 0x7034(%esp)
```

En ligne de commande, on tape alors :

```
as foo.s -o foo.o
objdump -d foo.o
```

On obtient que le codage (en hexadécimal) de la ligne précédente est :

```
83 bc 24 34 70 00 00 40
```

Ces valeurs correspondent au calcul suivant :

- Code de l'instruction = 83 : opération de comparaison d'une valeur source immédiate sur un octet, et d'une destination qui est un registre ou une valeur en mémoire (voir section 5.2.4).
- Octet *ModR/M* = BC : l'opérande en mémoire est adressé avec un déplacement sur 32 bits et un registre de base (cf. figure 7.3 page 58), l'extension du code opération indiquée par la description de l'instruction vaut 7.
- Octet *SIB* = 24.
- Champ déplacement = 34 70 00 00 : déplacement sur 32 bits, poids faible d'abord.
- Valeur immédiate = 40 : codage de 64 en hexadécimal sur un octet.

Une même ligne assembleur peut être codée de plusieurs façon. C'est à l'assembleur de choisir le codage le plus court. Par exemple, on peut coder la ligne ci-dessus en prenant le codage :

```
81 bc 24 34 70 00 00 40 00 00 00
```

Le code opération `81` indique en effet que la donnée immédiate est sur 4 octets. Alors que l'instruction `83` indique que la donnée immédiate est sur 1 octet, qui sera interprétée sur 4 octets après extension de signe.

5.2 Description des instructions

Nous décrivons maintenant les instructions interprétées par le processeur de notre machine. Comme nous l'avons déjà signalé, le compteur ordinal EIP contient à la fin de l'exécution d'une instruction, l'adresse effective de la prochaine instruction à exécuter. Certaines instructions, comme les sauts conditionnels, agissent directement sur EIP puisqu'elles forcent sa valeur à une valeur calculée pour provoquer une rupture de séquence dans l'exécution des instructions. Pour les instructions ne provoquant pas de ruptures de séquence, la progression du compteur ordinal dépend uniquement de la taille de l'instruction (du nombre d'octets nécessaires à son codage).

Certaines instructions modifient les indicateurs du registre EFLAGS. Ces modifications sont précisées dans la description des instructions et le calcul des différents indicateurs sont rassemblés dans la figure 7.2 en annexe du document. La signification des tableaux décrivant les instructions est la suivante :

- La première colonne est divisée en trois sous-colonnes dont la première donne la valeur hexadécimale du premier octet du code opération, la seconde la valeur du second octet s'il existe et la troisième le type du code opération. Dans cette colonne :
  - un chiffre d (entre 0 et 7) indique que le champ *Reg/Op* de l'octet *ModR/M* est utilisé comme extension du code opération. Dans ce cas la valeur hexadécimal de cette extension est d.
  - le symbole "/r" indique que le champ *Reg/Op* de l'octet *ModR/M* est utilisé pour coder un registre. La figure 7.3 indique le code de chaque registre.
  - le symbole "+r" indique qu'il n'y a pas d'octet *ModR/M* et que le code d'un registre est ajouté au code opération.
  - le symbole "-" indique qu'il n'y a pas d'octet *ModR/M* (pas d'opérande ou opérande immédiate).
- La seconde colonne décrit la syntaxe autorisée pour cette instruction sous la forme suivante : d'abord le nom de l'instruction en assembleur, puis les types autorisés pour chacune des opérandes. Ceux-ci sont décrits par :
  - **imm**n : pour une donnée immédiate qui peut être codée sur au plus n bits.
  - **r**n : pour un registre sur n bits.
  - **m**n : pour une adresse mémoire (qui n'est pas un registre) qui contient une valeur de n bits.
  - **r/m**n : pour une adresse mémoire ou un registre qui contient une valeur de n bits.
  - **rel**n : pour un entier relatif codable sur n bits à ajouter au registre EIP (dans un branchement).
  - ***r/m** : pour le mode d'adressage indirect dans les branchements.
- La dernière colonne décrit sommairement la signification de l'instruction.

5.2.1 Somme : ADD

Code opération			Instruction	Commentaires
O_0	**O_1**	**type**	**ADD source, dest**	
80	-	0	addb imm8, r/m8	Add donnée immédiate sur 8 bits à l'octet r/m8
81	-	0	addw imm16, r/m16	Add donnée immédiate sur 16 bits au mot r/m16
81	-	0	addl imm32, r/m32	Add donnée immédiate sur 32 bits au double r/m32
83	-	0	addw imm8, r/m16	Le signe de la donnée immédiate est étendu
83	-	0	addl imm8, r/m32	Le signe de la donnée immédiate est étendu
00	-	/r	addb r8, r/m8	Add registre 8 bits source à l'octet r/m8
01	-	/r	addw r16, r/m16	Add registre 16 bits source au mot r/m16
01	-	/r	addl r32, r/m32	Add registre 32 bits source au double r/m32
02	-	/r	addb r/m8, r8	Add l'octet r/m8 au registre 8 bits r8
03	-	/r	addw r/m16, r16	Add le mot r/m16 au registre 16 bits r16
03	-	/r	addl r/m32, r32	Add le double r/m32 au registre 32 bits r32

Description L'instruction ajoute le premier opérande au second et range le résultat dans le second. La destination peut être un registre ou une donnée en mémoire, le premier opérande peut être une donnée immédiate, un contenu de registre ou une donnée en mémoire. Cependant, les deux opérandes ne peuvent pas être simultanément en mémoire. Quand une donnée immédiate est utilisée sa taille est étendue par propagation de son signe à la taille de l'opérande destination. L'addition ne fait pas de distinction entre les valeurs signées ou non signées. Le processeur calcule le résultat et positionne les bits O de débordement et C de retenue de façon à indiquer une retenue dans les cas respectivement d'un résultat signé et d'un résultat non signé. Le bit S indique le signe du résultat. Le bit Z est également mis à jour par l'opération.

5.2.2 Et bit à bit : AND

Code opération			Instruction	Commentaires
O_0	**O_1**	**type**	**AND source, dest**	
80	-	4	andb imm8, r/m8	And donnée immédiate sur 8 bits à l'octet r/m8
81	-	4	andw imm16, r/m16	And donnée immédiate sur 16 bits au mot r/m16
81	-	4	andl imm32, r/m32	And donnée immédiate sur 32 bits au double r/m32
83	-	4	andw imm8, r/m16	Le signe de la donnée immédiate est étendu
83	-	4	andl imm8, r/m32	Le signe de la donnée immédiate est étendu
20	-	/r	andb r8, r/m8	And registre 8 bits source à l'octet r/m8
21	-	/r	andw r16, r/m16	And registre 16 bits source au mot r/m16
21	-	/r	andl r32, r/m32	And registre 32 bits source au double r/m32
22	-	/r	andb r/m8,r8	And l'octet r/m8 au registre 8 bits r8
23	-	/r	andw r/m16,r16	And le mot r/m16 au registre 16 bits r16
23	-	/r	andl r/m32,r32	And le double r/m32 au registre 32 bits r32

Description L'instruction effectue une opération 'ET' bit à bit entre le premier opérande et le second et range le résultat dans le second. Chaque bit du résultat est à 1 si les bits correspondants des deux opérandes sont à 1 et est à 0 sinon. La destination peut être un registre ou une donnée en mémoire. Le premier opérande peut être une donnée immédiate, un contenu de registre ou une donnée en mémoire. Cependant, les deux opérandes ne peuvent pas être simultanément en mémoire. Les indicateurs O de débordement et C de retenu sont mis à 0, les bits S et Z sont positionnés en fonction du résultat.

5.2.3 Appel de sous programme : CALL

Code opération			Instruction	Commentaires
O_0	**O_1**	**type**	**CALL dest**	
E8	-	-	call rel32	rel32 est le déplacement relatif sur 32 bits de la fonction
FF	-	2	call *r/m	L'adresse r/m32 contient l'adresse absolue de la fonction

Description L'instruction empile (cf. 5.2.12) l'adresse de l'instruction qui suit l'instruction CALL sur la pile pointée par le registre ESP et effectue le branchement à l'adresse calculée à partir de l'opérande fourni. Cet opérande peut être soit une valeur immédiate signée (sur 32 bits) qui est un déplacement à ajouter au contenu du registre EIP, soit une adresse de type r/m qui peut être un nom de registre ou une adresse en mémoire. Si c'est un nom de registre, le contenu de ce registre est directement affecté à EIP. Si c'est une adresse en mémoire, le double mot à cette adresse est affecté à EIP. Aucun indicateur du registre EFLAGS n'est modifié.

5.2.4 Comparaison : CMP

Code opération			Instruction	Commentaires
O_0	**O_1**	**type**	**CMP source, dest**	
80	-	7	cmpb imm8, r/m8	Cmp donnée immédiate sur 8 bits à l'octet r/m8
81	-	7	cmpw imm16, r/m16	Cmp donnée immédiate sur 16 bits au mot r/m16
81	-	7	cmpl imm32, r/m32	Cmp donnée immédiate sur 32 bits au double r/m32
83	-	7	cmpw imm8, r/m16	Le signe de la donnée immédiate est étendu
83	-	7	cmpl imm8, r/m32	Le signe de la donnée immédiate est étendu
38	-	/r	cmpb r8, r/m8	Cmp registre 8 bits source à l'octet r/m8
39	-	/r	cmpw r16, r/m16	Cmp registre 16 bits source au mot r/m16
39	-	/r	cmpl r32, r/m32	Cmp registre 32 bits source au double r/m32
3A	-	/r	cmpb r/m8,r8	Cmp l'octet r/m8 au registre 8 bits r8
3B	-	/r	cmpw r/m16,r16	Cmp le mot r/m16 au registre 16 bits r16
3B	-	/r	cmpl r/m32,r32	Cmp le double r/m32 au registre 32 bits r32

Description L'instruction compare le premier opérande et le second et positionne les indicateurs du registre EFLAGS en fonction du résultat.. La comparaison est effectuée par une soustraction entre l'opérande destination (second opérande) et l'opérande source et les codes condition sont positionnés comme pour la soustraction. Les valeurs immédiates représentées sur 8 bits sont étendues à 16 ou 32 bits par recopie de leur bit de signe. La correspondance entre la valeur des codes et l'instruction Jcc qui suit en général l'instruction de comparaison est donnée à la figure 7.1. Les indicateurs du registre EFLAGS modifiés sont : O, C, S, Z

5.2.5 Saut conditionnel : Jcc

Code opération			Instruction	Commentaires
O_0	**O_1**	**type**	**Jcc dest**	
77	-	-	ja rel8	Jump court si plus grand (C = 0 et Z = 0)
73	-	-	jae rel8	Jump court si plus grand ou égal (C = 0)
72	-	-	jb rel8	Jump court si plus petit (C = 1)
76	-	-	jbe rel8	Jump court si plus petit ou égal (C = 1 ou Z = 1)
74	-	-	je rel8	Jump court si égal (Z = 1)
7F	-	-	jg rel8	Jump court si plus grand (Z = 0 et S = 0)
7D	-	-	jge rel8	Jump court si plus grand ou égal (S = 0)
7C	-	-	jl rel8	Jump court si plus petit (S != 0)
7E	-	-	jle rel8	Jump court si plus petit ou égal (Z = 1 ou S != 0)
75	-	-	jne rel8	Jump court si pas égal (Z = 0)
0F	87	-	ja rel32	Jump court si plus grand (C = 0 et Z = 0)
0F	83	-	jae re32	Jump court si plus grand ou égal (C = 0)
0F	82	-	jb rel32	Jump court si plus petit (C = 1)
0F	86	-	jbe rel32	Jump court si plus petit ou égal (C = 1 ou Z = 1)
0F	84	-	je rel32	Jump court si égal (Z = 1)
0F	8F	-	jg rel32	Jump court si plus grand (Z = 0 et S = 0)
0F	8D	-	jge rel32	Jump court si plus grand ou égal (S = 0)
0F	8C	-	jl rel32	Jump court si plus petit (S != 0)
0F	8E	-	jle rel32	Jump court si plus petit ou égal (Z = 1 ou S != 0)
0F	85	-	jne rel32	Jump court si pas égal (Z = 0)

Description L'instruction teste l'état de un ou plusieurs indicateurs du registre EFLAGS, et si le ou les indicateurs sont dans l'état spécifié, le saut à l'instruction spécifié par l'opérande est réalisé. Sinon, le processeur interprète l'instruction suivante. L'instruction est spécifiée avec un déplacement relatif au contenu courant du registre d'instruction EIP. Ce dernier contient lors de l'interprétation de l'instruction Jcc l'adresse de l'instruction suivante (c'est-à-dire l'adresse de l'instruction Jcc + 2 si le déplacement est sur 1 octet ou +6 si le déplacement est sur 4 octets). La valeur du déplacement est ajoutée au contenu du registre EIP pour donner l'adresse de la destination du saut s'il doit être exécuté. Cette instruction ne modifie aucun des indicateurs de EFLAGS.

5.2.6 Saut inconditionnel : JMP

Code opération			Instruction	Commentaires
O_0	**O_1**	**type**	**JMP dest**	
EB	-	-	jmp rel8	Jump relatif, le déplacement est relatif à l'instruction suivante
E9	-	-	jmp rel32	Jump relatif, le déplacement est relatif à l'instruction suivante
FF	-	4	jmp *r/m	L'adresse r/m32 contient l'adresse absolue de destination du saut

Description L'instruction effectue un saut inconditionnel à l'instruction dont l'adresse est spécifiée soit par son déplacement donné sur 8 ou 32 bits, soit par une adresse en mémoire. Le déplacement supposé calculé relativement à l'instruction qui suit l'instruction JMP (adresse de JMP + 2 ou +5) est ajouté au contenu de EIP pour donner sa nouvelle valeur. L'adresse de type r/m est soit un nom de registre auquel cas le contenu du registre est affecté à EIP, soit une adresse absolue, soit enfin une adresse indirecte avec déplacement ou non. Dans ces deux

derniers cas, le double mot à l'adresse calculée ou donnée contient la nouvelle valeur de EIP. Cette instruction ne modifie aucun des indicateurs de EFLAGS.

5.2.7 Calcul de l'adresse d'un opérande en mémoire : LEA

Code opération			Instruction	Commentaires
O_0	**O_1**	**type**	**LEA source, dest**	
8D	-	/r	leal m32, r32	Stocke l'adresse effective de m32 dans le registre r32

Description L'instruction calcule l'adresse de l'opérande source et la range dans le registre r32. L'opérande m est une adresse mémoire spécifiée avec l'un des modes permis. Le registre r32 est l'un quelconque des registres généraux 32 bits. Cette instruction ne modifie aucun des indicateurs de EFLAGS.

5.2.8 Copier des données : MOV

Code opération			Instruction	Commentaires
O_0	**O_1**	**type**	**MOV source, dest**	
88	-	/r	movb r8, r/m8	Copie r8 dans r/m8
89	-	/r	movw r16, r/m16	Copie r16 dans r/m16
89	-	/r	movl r32, r/m32	Copie r32 dans r/m32
8A	-	/r	movb r/m8, r8	Copie r/m8 dans r8
8B	-	/r	movw r/m16, r16	Copie r/m16 dans r16
8B	-	/r	movl r/m32, r32	Copie r/m32 dans r32
C6	-	0	movb imm8, r/m8	Copie la donnée immédiate imm8 dans la mémoire r/m8
C7		0	movw imm16, r/m16	Copie la donnée immédiate imm16 dans la mémoire r/m16
C7	-	0	movl imm32, r/m32	Copie la donnée immédiate imm32 dans la mémoire r/m32

Description L'instruction copie le premier opérande dans le second. Le premier opérande peut être un registre, une donnée en mémoire ou une valeur immédiate. Le second opérande peut être un registre ou une valeur en mémoire. Cependant, les deux opérandes ne peuvent pas être simultanément en mémoire. Cette instruction ne modifie pas les indicateurs de EFLAGS.

5.2.9 NOP

Code opération			Instruction	Commentaires
O_0	**O_1**	**type**	**NOP**	
90	-	-	nop	aucune opération

Description L'instruction ne fait pas d'opération. Elle à une taille de 1 octet. Elle ne modifie pas le contexte du processeur sauf le registre EIP. Aucun des indicateurs du registre EFLAGS n'est modifié.

5.2.10 OU bit à bit : OR

Code opération			Instruction	Commentaires
O_0	**O_1**	**type**	**OR source, dest**	
80	-	1	orb imm8, r/m8	Or donnée immédiate sur 8 bits à l'octet r/m8
81	-	1	orw imm16, r/m16	Or donnée immédiate sur 16 bits au mot r/m16
81	-	1	orl imm32, r/m32	Or donnée immédiate sur 32 bits au double r/m32
83	-	1	orw imm8, r/m16	Le signe de la donnée immédiate est étendu
83	-	1	orl imm8, r/m32	Le signe de la donnée immédiate est étendu
08	-	/r	orb r8, r/m8	Or registre 8 bits source à l'octet r/m8
09	-	/r	orw r16, r/m16	Or registre 16 bits source au mot r/m16
09	-	/r	orl r32, r/m32	Or registre 32 bits source au double r/m32
0A	-	/r	orb r/m8,r8	Or l'octet r/m8 au registre 8 bits r8
0B	-	/r	orw r/m16,r16	Or le mot r/m16 au registre 16 bits r16
0B	-	/r	orl r/m32,r32	Or le double r/m32 au registre 32 bits r32

Description L'instruction effectue une opération 'OU' bit à bit entre le premier opérande (source) et le second et range le résultat dans le second (destination). Chaque bit du résultat est à 0 si les bits correspondants des deux opérandes étaient à 0, et il est à 1 sinon. La destination peut être un registre ou une donnée en mémoire, le premier opérande peut être une donnée immédiate, un contenu de registre ou une donnée en mémoire. Cependant les deux opérandes ne peuvent pas être simultanément en mémoire. Les indicateurs O de débordement et C de retenu sont mis à 0, les bits S et Z sont positionnés en fonction du résultat.

5.2.11 Dépiler : POP

Code opération			Instruction	Commentaires
O_0	**O_1**	**type**	**POP dest**	
8F	-	0	popw r/m16	Dépile le mot au sommet de pile dans r/m16
8F	-	0	popl r/m32	Dépile le double mot au sommet de pile dans r/m32
58	-	+r	popw r16	Dépile le mot au sommet de pile dans r16
58	-	+r	popl r32	Dépile le double mot au sommet de pile dans r32

Description L'instruction charge la valeur en sommet de pile (un mot ou un double mot selon qu'on exécute popw ou popl) dans l'emplacement spécifié par l'opérande destination puis augmente la valeur du pointeur de pile de 2 ou de 4 selon la taille de l'élément dépilé. L'opérande destination peut être une donnée en mémoire ou un registre. Aucun des indicateurs du registre EFLAGS n'est modifié.

5.2.12 Empiler : PUSH

Code opération			Instruction	Commentaires
O_0	**O_1**	**type**	**PUSH source**	
FF	-	6	pushw r/m16	Empile le mot désigné par r/m16
FF	-	6	pushl r/m32	Empile le double mot désigné par r/m32
50	-	+r	pushw r16	Empile le mot désigné par r16
50	-	+r	pushl r32	Empile le double mot désigné par r32
6A	-	-	pushl/w imm8	Empile la donnée après extension à 32 ou 16 du signe
68	-	-	pushw imm16	Empile la donnée immédiate imm16 (1 mot)
68	-	-	pushl imm32	Empile la donnée immédiate imm32

Description L'instruction diminue le pointeur de pile de 2 (pushw) ou 4 (pushl) puis stocke l'opérande au sommet de la pile (l'opérande est un opérande source puisque la destination est implicitement la pile). Le pushl %esp empile la valeur du pointeur de pile telle qu'elle était avant l'exécution de cette instruction. Aucun des indicateurs du registre EFLAGS n'est modifié.

5.2.13 Retour de procédure ou de fonction : RET

Code opération			Instruction	Commentaires
O_0	**O_1**	**type**	**RET**	
C3	-	-	ret	Retour à la fonction appelante.

Description Cette instruction dépile (cf. 5.2.11) le double mot au sommet courant de la pile dans le compteur d'instruction EIP. Si le sommet de pile contient une adresse de retour stockée au préalable par l'instruction call ayant fait appel au sous-programme où se trouve l'instruction ret, cette dernière provoquera le retour à l'appelant à la fin d'une fonction ou d'un sous programme. Aucun des indicateurs du registre EFLAGS n'est modifié.

5.2.14 Soustraction : SUB

Code opération			Instruction	Commentaires
O_0	**O_1**	**type**	**SUB source, dest**	
80	-	5	subb imm8, r/m8	Sub donnée immédiate sur 8 bits à l'octet r/m8
81	-	5	subw imm16, r/m16	Sub donnée immédiate sur 16 bits au mot r/m16
81	-	5	subl imm32, r/m32	Sub donnée immédiate sur 32 bits au double r/m32
83	-	5	subw imm8, r/m16	Le signe de la donnée immédiate est étendu
83	-	5	subl imm8, r/m32	Le signe de la donnée immédiate est étendu
28	-	/r	subb r8, r/m8	Sub registre 8 bits source à l'octet r/m8
29	-	/r	subw r16, r/m16	Sub registre 16 bits source au mot r/m16
29	-	/r	subl r32, r/m32	Sub registre 32 bits source au double r/m32
2A	-	/r	subb r/m8,r8	Sub l'octet r/m8 au registre 8 bits r8
2B	-	/r	subw r/m16,r16	Sub le mot r/m16 au registre 16 bits r16
2B	-	/r	subl r/m32,r32	Sub le double r/m32 au registre 32 bits r32

Description L'instruction retranche le premier opérande au second et range le résultat dans le second. La destination peut être un registre ou une donnée en mémoire, le premier opérande peut être une donnée immédiate, un contenu de registre ou une donnée en mémoire. Cependant les deux opérandes ne peuvent pas être simultanément en mémoire. Quand une donnée immédiate est utilisée, sa taille est étendue par propagation de son signe à la taille de l'opérande destination. La soustraction ne fait pas de distinction entre les valeurs signées ou non signées. Le processeur calcule le résultat et positionne les bits O de débordement et C de retenue de façon à indiquer une retenue dans les cas respectivement d'un résultat signé et d'un résultat non signé. Le bit S indique le signe du résultat. Le bit Z est également mis à jour par l'opération.

5.2.15 OU Exclusif bit à bit : XOR

Code opération			Instruction	Commentaires
O_0	**O_1**	**type**	**XOR source, dest**	
80	-	6	xorb imm8, r/m8	Xor donnée immédiate sur 8 bits à l'octet r/m8
81	-	6	xorw imm16, r/m16	Xor donnée immédiate sur 16 bits au mot r/m16
81	-	6	xorl imm32, r/m32	Xor donnée immédiate sur 32 bits au double r/m32
83	-	6	xorw imm8, r/m16	Le signe de la donnée immédiate est étendu
83	-	6	xorl imm8, r/m32	Le signe de la donnée immédiate est étendu
30	-	/r	xorb r8, r/m8	Xor registre 8 bits source à l'octet r/m8
31	-	/r	xorw r16, r/m16	Xor registre 16 bits source au mot r/m16
31	-	/r	xorl r32, r/m32	Xor registre 32 bits source au double r/m32
32	-	/r	xorb r/m8,r8	Xor l'octet r/m8 au registre 8 bits r8
33	-	/r	xorw r/m16,r16	Xor le mot r/m16 au registre 16 bits r16
33	-	/r	xorl r/m32,r32	Xor le double r/m32 au registre 32 bits r32

Description L'instruction effectue une opération 'OU Exclusif' bit à bit entre le premier opérande (source) et le second et range le résultat dans le second (destination). Chaque bit du résultat est à 1 si les bits correspondants des deux opérandes sont différents, et est à 0 sinon. La destination peut être un registre ou une donnée en mémoire, le premier opérande peut être une donnée immédiate, un contenu de registre ou une donnée en mémoire. Cependant les deux opérandes ne peuvent pas être simultanément en mémoire. Les indicateurs O de débordement et C de retenu sont mis à 0, les bits S et Z sont positionnés en fonction du résultat.

Chapitre 6

ELF : Executable and Linkable Format

Ce chapitre[1] décrit brièvement le format ELF et montre comment en extraire des informations. La lecture de fichiers au format ELF est nécessaire pour : le désassemblage, la simulation, l'édition de liens. L'écriture de fichiers au format ELF est nécessaire pour l'assemblage et l'édition de liens.

Le format ELF est le format des fichiers objets dans plusieurs systèmes d'exploitation, dont Linux et Solaris. Il est conçu pour assurer une certaine portabilité entre différentes plates-formes. Il s'agit d'un format standard pouvant supporter l'évolution des architectures et des systèmes d'exploitation. Le format ELF est manipulable, en lecture et écriture, par utilisation d'une bibliothèque de fonctions d'accès `libelf`. Cette librairie suffit à lire et écrire des fichiers ELF, même quand ELF n'est pas le format utilisé par le système d'exploitation (par exemple, on peut l'utiliser sous MacOs X).

Dans ce chapitre, nous nous limitons aux fichiers dits *à lier*, fichiers contenant du *code binaire relogeable* (ou *translatable*). Ce chapitre décrit tout d'abord la structure d'un fichier objet, puis donne quelques indications sur les fonctions d'accès à travers la bibliothèque. Voir `man elf` pour plus d'informations sur ces fonctions.

6.1 Structure générale d'un fichier objet au format ELF

Un fichier objet à lier est formé d'un en-tête donnant des informations générales sur la version, la machine, etc., puis d'un certain nombre de pointeurs et de valeurs décrits ci-dessous. Ce que nous appelons ici pointeur est en fait une valeur représentant un déplacement en nombre d'octets par rapport au début du fichier. Les tailles, elles aussi, sont exprimées en nombre d'octets. La figure 6.1 donne une idée de la structure d'un fichier au format ELF. Le fichier est constitué d'un *en-tête* donnant les caractéristiques générales du fichier, puis d'un certains nombre de *sections* contenant différentes formes de données (par exemple, section "`.text`", section "`.data`", section des "relocations en zone text", section de la table des symboles, etc).

Entête d'un fichier ELF L'en-tête est décrit par le type C `struct ELF32_Ehdr` dont voici la description des principaux champs :

- `e_ident` : identification du format et des données indépendantes de la machine permettant d'interpréter le contenu du fichier (Cf. exemple dans le paragraphe suivant).
- `e_type` : un fichier relogeable a le type `ET_REL` (constante égale à 1).

[1] Ce chapitre est en partie emprunté à Fabienne Lagnier, UFR IMA

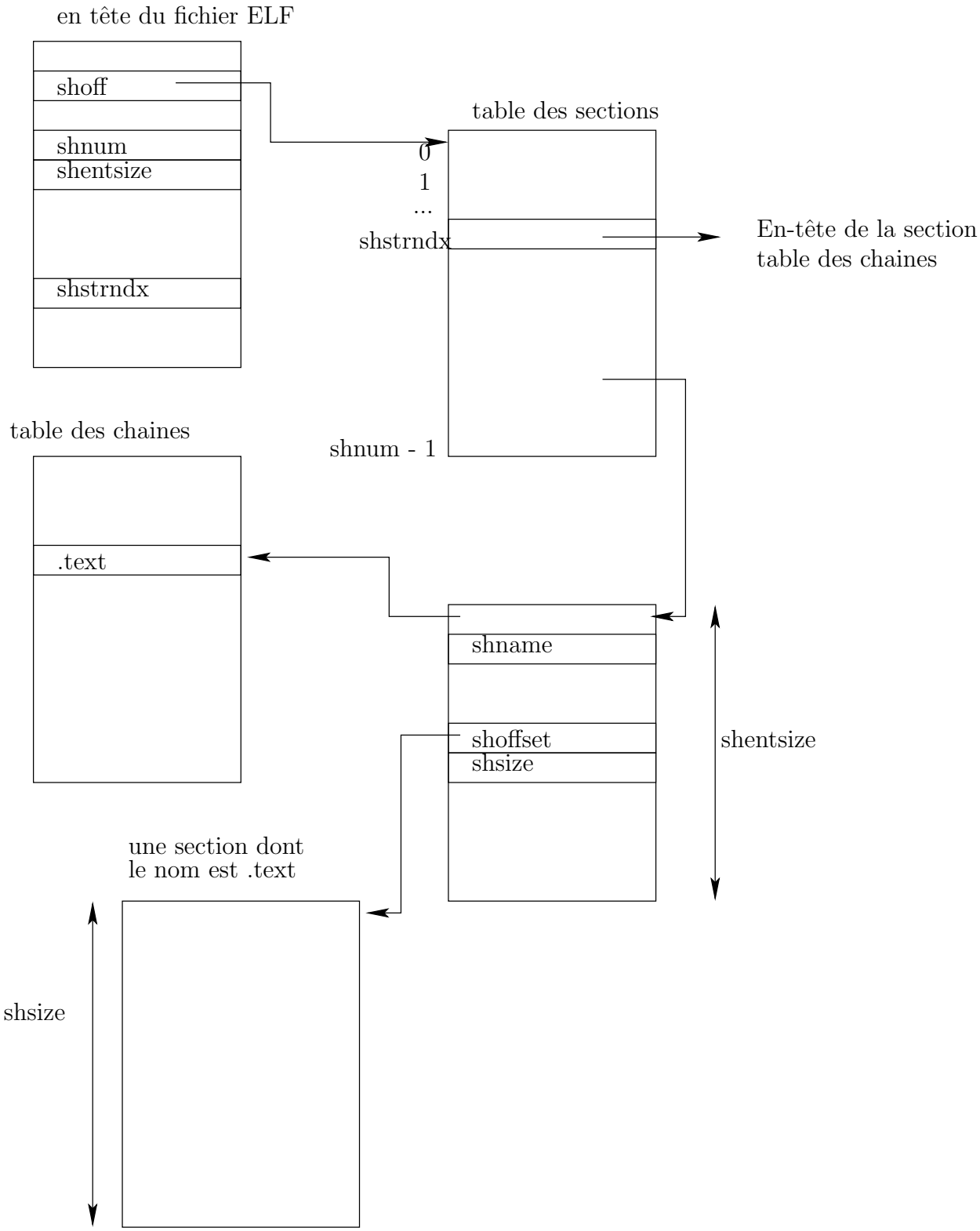

FIG. 6.1 – Structure d'un fichier relogeable au format ELF

- `e_machine` : le processeur intel qui nous intéresse est identifié par la valeur `EM_386` (constante égale à 3).
- `e_version` : la version courante est 1.
- `e_ehsize` : taille de l'en-tête en nombre d'octets.
- `e_shoff` : pointeur sur la *table des en-têtes de sections* (ou plus simplement "*table des sections*"). Chaque entrée de cette table est l'en-tête d'une section qui décrit la nature de cette section, et donne sa localisation dans le fichier (voir ci-dessous). La table des sections est appelée *shdr* dans la documentation ELF. Dans le reste du format ELF, les sections sont généralement désignées par l'index de leur en-tête dans cette table. Le premier index (qui est 0) est une sentinelle qui désigne la section "non définie".
- `e_shnum` : nombre d'entrées dans la table des sections.
- `e_shentsize` : taille d'une entrée de la table des sections.
- `e_shstrndx` : index de la table des noms de sections (dans la table des en-têtes de sections).
- Les autres champs de l'en-tête ne sont pas utilisés dans le cadre du projet. On les mets à 0.

En-têtes des sections Un en-tête de section est décrit par le type C `struct Elf32_shdr`. Il définit les champs suivants :

- `sh_name` : index dans la table des noms de sections (section "`.shstrtab`").
- `sh_type` : type de la section. Les types qu'on utilise dans ce projet sont les suivants :
  - `SHT_PROGBITS` (constante 1) : type des sections "`.text`" et "`.data`". Ce type indique que la section contient une suite d'octets correspondant aux données ou aux instructions du programme.
  - `SHT_NOBITS` (constante 8) : type de la section "`.bss`" (données non initialisées). La section ne contient aucune donnée. Elle sert essentiellement à déclarer la taille occupée par les données non initialisées (voir ci-dessous).
  - `SHT_SYMBTAB` (constante 2) : type des tables des symboles. Dans le projet, on en utilise une seule, appelée "`.symtab`".
  - `SHT_STRTAB` (constante 3) : type des tables de chaînes. Dans le projet, deux sections ont ce type : la table des noms de sections, appelée "`.shstrtab`", et la tables des noms de symboles, appelée "`.strtab`" (elle est souvent désignée par "table des chaînes").
  - `SHT_REL` (constant 9) : type des tables de relocations. Dans le projet, on en considérera deux : "`.rel.text`" pour les relocations en zone `.text` et "`.rel.data`" pour les relocations en zone `.data`.
- `sh_offset` : pointeur sur le début de la section dans le fichier.
- `sh_size` : taille de la section (en octets). Si le type de la section n'est pas `SHT_NOBITS`, la section doit contenir effectivement `sh_size` octets. Si le type de la section est `SHT_NOBITS`, ce champ sert à déclarer la taille de la zone non initialisée (mais la section contient en réalité 0 octet).
- `sh_addralign` : contrainte d'alignement sur l'adresse finale de la zone en mémoire. L'adresse finale doit être un multiple de ce nombre.
- `sh_entsize` : certaines sections ont des entrées de taille fixe. Cet entier donne donc cette taille. Dans le projet, seules les sections de type `SHT_SYMBTAB` et `SHT_REL` sont concernées par ce champ.
- `sh_link` et `sh_info` : ont des interprétations qui dépendent du type de la section. Dans tous les cas, le champs `sh_link` est l'index d'un en-tête de section (dans la table des en-têtes de sections). Dans le cadre du projet, on a :

sh_type	sh_link	sh_info
SHT_REL	l'index de la table de symbole associée	l'index (dans la table des entêtes de sections) de la section à reloger
SHT_SYMTAB	l'index de la table des noms de symboles	l'index (dans la table des symboles) du premier symbole global [2]

Remarque le contenu d'un fichier objet *exécutable* ressemble à celui d'un fichier objet relogeable. Il est formé de segments au lieu de sections et on y trouve ainsi une table des segments (au lieu d'une table des sections). Dans l'en-tête, les informations décrivant la table des segments sont données par les champs dont les noms commencent par `ph` au lieu de `sh`.

6.2 Exemple de fichier relogeable

Avant de continuer l'étude du format ELF en détaillant le format de chacune des sections considérées dans le projet, introduisons un petit exemple, qui permettra de rendre cette étude plus concrète. Considérons ainsi le programme en langage d'assemblage `exempleElf.s`, dont le code suit :

```
.section .data
        X:  .long 42
        Y:  .long X

.section .text
        addl $X, %esp

.section .bss
        .skip 1200
```

La commande `gcc -c exempleElf.s` produit un fichier objet `exempleElf.o` dont on peut visualiser le contenu, par la commande Unix `od -tx1 exempleElf.o`. L'option `-tx1` indique que l'affichage attendu est de type `x1` c'est-à-dire que le contenu du fichier est affiché octet par octet en hexadécimal. Le résultat de la commande est donné en figure 6.2, page 56.

C'est assez difficile à lire. On va utiliser un *désassembleur* construit par les enseignants pour observer les fichiers objets (alternativement, on peut utiliser `objdump` sous Linux). Il s'agit d'un outil capable d'analyser la séquence de bits ci-dessus pour y retrouver la structure d'un fichier objet, et d'afficher en clair les informations intéressantes. Le désassembleur, pour le même fichier objet, produit :

```
 ====================== HEADER ====================
Ident bits : 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Type = 1
Machine = 3
Version = 1
Entry point  = 0
phdr file offset = 0
shdr file offset = 0x78
file flags : 0
sizeof ehdr = 52
```

[2] On verra en effet en sous-section 6.3.6 que tous les symboles globaux doivent être rassemblés à la fin de la table des symboles.

```
sizeof phdr = 0
number phdrs = 0
sizeof shdr = 40
number shdrs  = 9
shdr string index = 0x6
On parcourt les sections en cherchant la table des chaines
        [1].text (type=SHT_PROGBITS)
        [2].rel.text (type=SHT_REL)
        [3].data (type=SHT_PROGBITS)
        [4].rel.data (type=SHT_REL)
        [5].bss (type=SHT_NOBITS)
        [6].shstrtab (type=SHT_STRTAB)
        [7].symtab (type=SHT_SYMTAB)
        [8].strtab (type=SHT_STRTAB)

================================================================================
[1].text (type=SHT_PROGBITS)

----------
Name = 0x1f
Type = 1
Flags = 6
Virtual address = 0
File offset = 0x34
Size = 6
Link = 0
Info = 0
Addralign = 4
Entry size = 0
----------
        Taille = 6
        Offset = 0
        Entry Size  = 0
 TEXT
 0000:  addl    $0x0,%esp
        [ 81 c4 00 00 00 00 ]
 -- fin de section --

================================================================================
[2].rel.text (type=SHT_REL)
 Table de translation sans champ addend (TEXT ou DATA)

----------
Name = 0x1b
Type = 9
Flags = 0
Virtual address = 0
File offset = 0x248
Size = 8
Link = 7
Info = 1
Addralign = 4
Entry size = 8
----------
        Taille = 8
```

```
        Entry Size  = 8
        Number of elements : 1
-------------------------------------------------
r_offset| r_info = <sym,     |   type>             |
--------|--------------------|---------------------|
     2  |             2      |                  32
 -- fin de section --

================================================================================
[3].data (type=SHT_PROGBITS)

----------
Name = 0x29
Type = 1
Flags = 3
Virtual address = 0
File offset = 0x3c
Size = 8
Link = 0
Info = 0
Addralign = 4
Entry size = 0
----------
        Taille = 8
        Offset = 0
        Entry Size  = 0
 DATA
               0 :  |2a|00|00|00| --- *
               4 :  |00|00|00|00| ---
 -- fin de section --

================================================================================
[4].rel.data (type=SHT_REL)
 Table de translation sans champ addend (TEXT ou DATA)

----------
Name = 0x25
Type = 9
Flags = 0
Virtual address = 0
File offset = 0x250
Size = 8
Link = 7
Info = 3
Addralign = 4
Entry size = 8
----------
        Taille = 8
        Entry Size  = 8
        Number of elements : 1
-------------------------------------------------
r_offset| r_info = <sym,     |   type>             |
--------|--------------------|---------------------|
     4  |             2      |                  32
 -- fin de section --
```

```
================================================================================
[5].bss (type=SHT_NOBITS)

----------
Name = 0x2f
Type = 8
Flags = 3
Virtual address = 0
File offset = 0x44
Size = 1200
Link = 0
Info = 0
Addralign = 4
Entry size = 0
----------
        Taille = 1200
 -- fin de section --

================================================================================
[6].shstrtab (type=SHT_STRTAB)
 Table de chaines

----------
Name = 0x11
Type = 3
Flags = 0
Virtual address = 0
File offset = 0x44
Size = 52
Link = 0
Info = 0
Addralign = 1
Entry size = 0
----------
        Table des chaines, Taille = 52
  0 :
0x1 : .symtab
0x9 : .strtab
0x11 : .shstrtab
0x1b : .rel.text
0x25 : .rel.data
0x2f : .bss
0x34 :
 -- fin de section --

================================================================================
[7].symtab (type=SHT_SYMTAB)
 Table des symboles

----------
Name = 0x1
Type = 2
Flags = 0
Virtual address = 0
File offset = 0x1e0
```

```
Size = 96
Link = 8
Info = 6
Addralign = 4
Entry size = 16
----------
        Taille = 96
        Entry Size  = 16
------------------------------------------------------------------------------------
No  |  name   | value   | size |  Bind  |   Type   | Other | S. idx | name
    |         |         |      |  STB_  |   STT_   | Other | S. idx |
____|_________|_________|______|________|__________|_______|________|_______________
  0 |    0    |       0 |   0  |  LOCAL |  NOTYPE  |    0  |      0 | (num. inutilise)
  1 |    0    |       0 |   0  |  LOCAL | SECTION  |    0  |      1 |
  2 |    0    |       0 |   0  |  LOCAL | SECTION  |    0  |      3 |
  3 |    0    |       0 |   0  |  LOCAL | SECTION  |    0  |      5 |
  4 |    1    |       0 |   0  |  LOCAL |  NOTYPE  |    0  |      3 | X
____|_________|_________|______|________|__________|_______|________|_______________
  5 |    3    |       4 |   0  |  LOCAL |  NOTYPE  |    0  |      3 | Y
 -- fin de section --

================================================================================
[8].strtab (type=SHT_STRTAB)
 Table de chaines

----------
Name = 0x9
Type = 3
Flags = 0
Virtual address = 0
File offset = 0x240
Size = 5
Link = 0
Info = 0
Addralign = 1
Entry size = 0
----------
        Table des chaines, Taille = 5
  0 :
0x1 : X
0x3 : Y
0x5 :
 -- fin de section --
```

6.3 Détail des sections

6.3.1 L'en-tête

```
0000000 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
0000020 01 00 03 00 01 00 00 00 00 00 00 00 00 00 00 00
0000040 78 00 00 00 00 00 00 00 34 00 00 00 00 00 28 00
0000060 09 00 06 00
```

Les 16 premiers octets constituent le champ `e_ident` (taille définie par la constante `EI_IDENT`). Les quatre premiers identifient le format : notons que $0x45$ $0x4c$ $0x46$ sont les codes ASCII des caractères `'E', 'L', 'F'`. Le cinquième `01` donne la classe du fichier, ici `ELFCLASS32`, ce qui signifie que les adresses sont exprimées sur 32 bits. Le sixième donne le type de codage des données, ici `1`, ce qui signifie que les données sont codées en complément à deux, avec les bits les plus significatifs occupant les adresses les plus hautes (little endian).

Par ailleurs on repère : le déplacement par rapport au début du fichier donnant accès à la table des sections ($0x78$ octets) ; la taille de l'en-tête ($52 = 0x34$ octets) ; la taille d'une entrée de la table des sections ($40 = 0x28$ octets), le nombre d'entrées dans la table des sections (9), l'entrée numéro 6 dans la table des sections est celle du descripteur de la table des noms de sections (celle-ci est indispensable pour savoir quelle section décrit un autre descripteur).

6.3.2 La table des noms de sections

Observons ce que produit le désassembleur. La table des noms de sections est du type table de chaînes de caractères. Elle contient les noms suivants :

```
[6].shstrtab (type=SHT_STRTAB)

 Table de chaines

----------
Name = 0x11
Type = 3
Flags = 0
Virtual address = 0
File offset = 0x44
Size = 52
Link = 0
Info = 0
Addralign = 1
Entry size = 0
----------
        Table des chaines, Taille = 52
  0 :
0x1 : .symtab
0x9 : .strtab
0x11 : .shstrtab
0x1b : .rel.text
0x25 : .rel.data
0x2f : .bss
0x34 :
 -- fin de section --
```

Le format ELF impose certaines règles à respecter. La première chaîne doit être la chaîne vide. Chaque chaîne doit se terminer par le caractère de code ascii 0 (comme en C). Remarquons que dans cette table, la chaine "`.text`" est accessible à partir de l'adresse $0x1f$ et que la chaine "`.data`" est accessible à partir de l'adresse $0x29$.

6.3.3 La section table des chaînes

Cette section (qui porte l'index 8) rassemble tous les noms des symboles. Les règles à respecter pour construire cette section sont les mêmes que pour `.shstrtab` ci-dessus.

```
[8].strtab (type=SHT_STRTAB)
 Table de chaines

----------
Name = 0x9
Type = 3
Flags = 0
Virtual address = 0
File offset = 0x240
Size = 5
Link = 0
Info = 0
Addralign = 1
Entry size = 0
----------
        Table des chaines, Taille = 5
  0 :
0x1 : X
0x3 : Y
0x5 :
```

6.3.4 La section `TEXT`

La portion de fichier objet correspondant à la zone TEXT (les instructions) est la première portion encadrée dans le résultat de la commande `od` figure 6.2, c'est-à-dire :

```
81 c4 00 00 00 00
```

Ce sont 6 octets correspondant au codage de l'instruction `addl $X, %esp`. Les deux premiers codent l'instruction elle-même, le mode d'adressage utilisé, et le numéro de registre concerné. Les 4 derniers octets devraient être remplis par la valeur immédiate d'adresse correspondant à l'étiquette `X`. Comme cette adresse ne sera en fait connue que lors du chargement final en mémoire, les 4 octets codent la valeur connue lors de l'assemblage, qui est un déplacement par rapport au début de la zone de définition du symbole concerné (ici 0 en zone DATA), et une donnée de relocation TEXT est associée à cette instruction (cf. plus loin).

6.3.5 La section `DATA`

La portion de fichier objet correspondant à la zone DATA (les données initialisées) est la seconde et la troisième portion encadrée dans le résultat de la commande `od` figure 6.2, c'est-à-dire :

```
2a 00 00 00 00 00 00 00
```

Les 4 premiers octets codent en little-endian la valeur 42 (2a en hexa). Les 4 octets suivants devraient contenir la valeur immédiate d'adresse correspondant à l'étiquette `X`. Comme cette adresse ne sera en fait connue que lors du chargement final en mémoire, les 4 octets codent la valeur 0, et une donnée de translation DATA est associée à cette instruction (cf. plus loin).

6.3.6 La section table des symboles

Observons ce que donne le désassembleur.

```
[7].symtab (type=SHT_SYMTAB)
 Table des symboles
```

```
----------
Name = 0x1
Type = 2
Flags = 0
Virtual address = 0
File offset = 0x1e0
Size = 96
Link = 8
Info = 6
Addralign = 4
Entry size = 16
----------
        Taille = 96
        Entry Size  = 16
```

No	name	value	size	Bind STB_	Type STT_	Other Other	S. idx S. idx	name
0	0	0	0	LOCAL	NOTYPE	0	0	(num. inutilise)
1	0	0	0	LOCAL	SECTION	0	1	
2	0	0	0	LOCAL	SECTION	0	3	
3	0	0	0	LOCAL	SECTION	0	5	
4	1	0	0	LOCAL	NOTYPE	0	3	X
5	3	4	0	LOCAL	NOTYPE	0	3	Y

Le type décrivant une entrée de la table des symboles est `struct Elf32Sym`. Une entrée occupe 16 octets, il y a ici 6 entrées. La première entrée est à zéro (elle sert de sentinelle). Les entrées de numéros 1, 2 et 3 représentent sont des symboles spéciaux qui représentent en fait respectivement les sections `.text`, `.data` et `.bss`. On remarque en effet qu'elles ont le type `STT_SECTION` (constante 3). Le champ `idx` de ces 3 entrées indique l'index de la section considérée (dans la table des en-têtes de section). Les entrées numéros 4 et 5 sont de vrais symboles de l'utilisateur, correspondant aux étiquettes `X` et `Y`.
Les différents champ de la structure `Elf32Sym` sont les suivants :

- `st_name` : index du symbole dans la table des noms de symboles. Lorsque le symbole est de type `STT_SECTION`, aucun nom ne lui est associé. La valeur de l'index est alors 0 (qui désigne donc la chaîne vide, d'après les contraintes sur `.strtab`).
- `st_value` : il vaut 0 pour un symbole non défini ; pour un symbole défini localement, `st_value` est le déplacement (en octets) par rapport au début de la zone de définition.
- `st_shndx` : indique l'index (dans la table des en-têtes de .sections) de la section où le symbole est défini. Si le symbole n'est défini dans aucune section (symbole externe), cet index vaut 0. Ainsi **l'index 0 de la table des en-têtes de section est une sentinelle qui sert à marquer les symboles externes**. Si le symbole est de type `STT_SECTION`, cet index est directement l'index de la section.
- `st_size` et `st_other` : non utilisés dans le projet (`st_size` sert à associer une taille de donnée au symbole).
- `st_info` : ce champ codé sur un octet sert en fait à coder 2 champs : le champ "bind" et le champ "type". Les macros suivantes (définie dans les fichiers d'en-tête du format ELF) permettent d'encoder ou de décoder le champ `st_info` :

  ```
  #define ELF32_ST_BIND(i)   ((i)>>4)              /* de info vers bind */
  #define ELF32_ST_TYPE(i)   ((i)&0xf)             /* de info vers type */
  #define ELF32_ST_INFO(b,t) (((b)<<4)+((t)&0xf))  /* de (bind,type) vers info */
  ```

  Le champ "bind" indique la portée du symbole. Dans le cadre du projet, on considère les

2 cas suivants :
- `STB_LOCAL` (constante 0) indique que le symbole est défini localement et non exporté.
- `STB_GLOBAL` (constante 1) indique que le symbole est soit défini et exporté, soit non défini et importé. Il faut noter qu'à l'édition de lien, il est interdit à 2 fichiers distincts de définir deux symboles de même nom ayant la portée `STB_GLOBAL`.

Dans le cadre du projet, on ne considère que les 2 cas suivants pour le champ "type" :
- `STT_SECTION` (constante 3) indiquant que le symbole désigne en fait une section.
- `STT_NOTYPE` (constante 0), dans les autres cas.

Le format ELF impose certaines contraintes sur l'ordre des symboles dans la table : le premier symbole n'est pas utilisé. Tous les symboles globaux doivent être regroupés à la fin de la table.

6.3.7 La section de relocation zone text

Les données de relocation TEXT décrivent les "trous" dans le codage des instructions de la zone TEXT. Ces trous correspondent à des valeurs immédiates que l'on ne peut pas connaître au moment de la fabrication du fichier objet, c'est-à-dire des valeurs d'adresse associées aux symboles (étiquettes). L'instruction est codée en réservant les 4 octets nécessaires au stockage de la valeur d'adresse. La valeur indiquée dans ce "trou" est provisoire, mais elle est très importante, comme nous allons le voir. Sur l'exemple, la table des relocations en zone texte est :

```
[2].rel.text (type=SHT_REL)
 Table de translation sans champ addend (TEXT ou DATA)

----------
Name = 0x1b
Type = 9
Flags = 0
Virtual address = 0
File offset = 0x248
Size = 8
Link = 7
Info = 1
Addralign = 4
Entry size = 8
----------
        Taille = 8
        Entry Size  = 8
        Number of elements : 1
--------------------------------------------------
r_offset| r_info = <sym,     |   type>             |
--------|--------------------|---------------------|
     2  |             2      |                  32
```

Le type `struct Elf32_Rel` décrit une entrée des tables de relocations.
- `r_offset` : indique la position du "trou" dans la zone à reloger. C'est un déplacement en octets par rapport au début de la zone.
- `r_info` : est un champ codé sur 4 octets qui contient en fait 2 champs : "sym" et "type". Les macros de codage/décodage du champ `r_info` défini dans les fichiers d'en-têtes de la librairie ELF sont :

```
#define ELF32_R_SYM(i)    ((i)>>8)                      /* info vers sym */
#define ELF32_R_TYPE(i)   ((unsigned char)(i))          /* info vers type */
#define ELF32_R_INFO(s,t) (((s)<<8)+(unsigned char)(t)) /* (sym,type) vers info */
```

Le champ "sym" est l'index du symbole dans la table des symboles. Dans le cadre du projet, le champ "type" vaut soit `R_386_32` (constante 1), soit `R_386_PC32` (constante 2). Il indique le mode de calcul de la relocation.

Détaillons, maintenant le mode de calcul (c-a-d la valeur sur 4 octets que l'éditeur de lien met finalement dans le "trou") en fonction du type de la relocation. Dans la suite S désigne l'adresse "finale" du symbole à reloger, P désigne l'adresse "finale" du "trou" et A désigne la valeur provisoire (sur 4 octets) se trouvant dans le "trou" (c-a-d la valeur à l'adresse P).

- Pour le type `R_386_32` : la valeur mise à l'adresse P vaut $S + A$. On se sert de ce mode pour les adressages absolus.
  Dans le cas où le symbole est défini localement et non exporté, A sert à mettre la position du symbole dans sa zone de définition. Le champ "sym" de la relocation ne correspond pas alors à l'index du symbole mais à celui de sa zone ! C'est en fait le cas de l'exemple ci-dessus. L'index de numéro 2 est en effet dans la table des symboles, celui du symbole correspondant à la zone DATA. Dans ce cas S est en fait l'adresse "finale" de la zone.
  Sinon, A a en principe (dans le cadre du projet) la valeur 0 et l'éditeur de lien calcule la valeur de S en ajoutant le champ `st_value` du symbole à l'adresse finale de la zone de définition du symbole.
  Cette différence de traitement entre les symboles locaux et les symboles globaux permet en fait d'omettre tous les symboles locaux (excepté les symboles de section) de la table des symboles et de la table des chaines : on réduit ainsi la taille du fichier généré.[3] Toutefois, il peut être commode de garder les symboles locaux lorsqu'on utilise un débogueur.
- Pour le type `R_386_PC32` : la valeur mise à l'adresse P vaut $S+A-P$. On se sert de ce mode pour les adressages relatifs. Ce type de relocation n'est utilisé que pour les symboles non définis localement ou extérieurs à la zone TEXT. Dans le cas contraire, on peut en effet calculer dès l'assemblage la valeur du saut relatif : il n'est donc pas nécessaire d'utiliser une relocation. En général, A a ici la valeur -4, car sur les pentium, un saut relatif s'exprime par rapport à l'adresse de l'instruction qui suit le saut (et donc l'adresse qui suit le trou).
  Pour un saut relatif sur un symbole défini localement mais dans une autre zone (zone DATA ou BSS), on peut appliquer le même principe que précédement : mettre dans A la position du symbole dans sa zone moins 4, et mettre l'index du symbole de zone dans le champ "sym" de la relocation.

Expérience à faire : reprendre le programme d'exemple, remplacer la référence à `X` par une référence à `Y` dans la zone TEXT, assembler grâce à `gcc -c`, observer le fichier objet produit grâce au désassembleur, et repérer la valeur relative de `Y` dans le codage de l'instruction. Même chose ensuite avec une référence à un symbole externe.
Observer également l'exemple suivant :

```
    .section .data

A: .long func
B: .long 13

    .section .text
boucle :

    addl %eax, %eax
    jne  boucle             # codage complet, pas de translation
    addl %eax, %eax
```

[3] Pour omettre les symboles locaux avec l'assembleur du projet `asm`, il faut l'invoquer avec l'option `-n`.

```
    jmp  toto               # translation, symbole externe
    addl %eax, %eax
    jmp  *A                 # absolu, translation sur symbole def. localement

func:
    addl %eax, %eax         # ce pourrait etre une fonction
```

les données de translation produites :

```
r_offset| r_info = <sym,     |   type>             |
--------|--------------------|---------------------|
     f  |             2      |                  32 |
     7  |             8      |                PC32 |
```

Et la table des symboles :

```
No  |  name   | value   | size |  Bind  |   Type   | Other | S. idx | name
    |         |         |      |  STB_  |   STT_   | Other | S. idx |
----|---------|---------|------|--------|----------|-------|--------|---------------
  0 |    0    |       0 |   0  |  LOCAL |  NOTYPE  |    0  |      0 | (num. inutilise)
  1 |    0    |       0 |   0  |  LOCAL | SECTION  |    0  |      1 |
  2 |    0    |       0 |   0  |  LOCAL | SECTION  |    0  |      3 |
  3 |    0    |       0 |   0  |  LOCAL | SECTION  |    0  |      5 |
  4 |    1    |       0 |   0  |  LOCAL |  NOTYPE  |    0  |      3 | A
----|---------|---------|------|--------|----------|-------|--------|---------------
  5 |    3    |      13 |   0  |  LOCAL |  NOTYPE  |    0  |      1 | func
  6 |    8    |       4 |   0  |  LOCAL |  NOTYPE  |    0  |      3 | B
  7 |    a    |       0 |   0  |  LOCAL |  NOTYPE  |    0  |      1 | boucle
  8 |   11    |       0 |   0  | GLOBAL |  NOTYPE  |    0  |      0 | toto
```

La deuxième donnée de translation (adresse 7 en zone TEXT) correspond à l'instruction `jmp toto`, et la table des symboles comporte bien une entrée numéro 8 correspondant au symbole `toto` qui est GLOBAL et n'a pas de numéro de section de définition. Le mode d'adressage utilisé indique un branchement relatif, donc le type est `PC32`.

La première donnée de translation (adresse f en zone TEXT) correspond à l'instruction `jmp *A`. Il s'agit d'un branchement absolu, et le type de translation est donc `32`. Le trou de l'instruction doit être rempli avec la valeur d'adresse représentée par l'étiquette `A`, qui est connue localement. La donnée de translation indique sa zone de définition (2 = numéro du symbole de section DATA).

6.3.8 La section de translation zone data

Le principe est complètement similaire à celui de la zone TEXT. Les symboles utilisés comme valeurs d'initialisation dans la directive "`.long`" sont des relocations de type `R_386_32`.

6.4 La bibliothèque *libelf*

Le format ELF est manipulable, en lecture et écriture, par utilisation d'une bibliothèque de fonctions d'accès `libelf`. La description des fonctions est accessible par `man` (sur `telesun`). Nous ne donnons ici que quelques éléments comme point d'entrée dans la bibliothèque ELF.

6.4.1 Lecture d'un fichier au format ELF

La lecture d'un fichier au format ELF commence par l'acquisition du descripteur ELF grâce à la fonction `elf_begin` :

```
Elf *elf;
elf = elf_begin(fd, ELF_C_READ, NULL)
```

L'accès à l'en-tête peut alors être réalisé avec la fonction `elf32_getehdr` :.

```
Elf32_Ehdr *ehdr;
ehdr = elf32_getehdr(elf)
```

On parcourt la table des sections, section par section en utilisant la fonction `elf_getscn`. Le champ `e_shstrndx` de l'en-tête permet de retrouver le nom de la section en spécifiant l'emplacement de la table des noms de sections.

```
Elf_Scn *scn;
scn = elf_getscn(elf, ehdr->e_shstrndx)
```

On peut alors lire le contenu de la section avec la fonction `elf_getdata` :

```
Elf_data *data;
data = elf_getdata(scn, NULL)
```

Les fonctions `elf_getehdr`, `elf_getscn` et `elf_getdata` allouent de la mémoire (`malloc`) pour lire l'information demandée et retournent un pointeur sur la zone mémoire allouée.

Les fichiers source fournis pour le désassembleur montrent comment enchaîner des appels aux primitives fournies par `libelf` pour lire un fichier objet.

6.4.2 Ecriture d'un fichier au format ELF

L'écriture dans un fichier ELF commence par :

```
elf = elf_begin(fd_object, ELF_C_WRITE, NULL)
```

Puis, il faut produire un en-tête : `ehdr = elf32_newehdr (elf)` et en définir chaque champ. On peut ensuite créer les différentes sections, en commençant par la table des sections (`elf_newscn`), les données pour une section étant créées en utilisant la fonction : `elf_newdata`. La prise en compte de ce travail et l'écriture effective dans le fichier est réalisée lors de la commande `elf_update`.

Adresse	Octets du fichier															
0000000	7f	45	4c	46	01	01	01	00	00	00	00	00	00	00	00	00
0000020	01	00	03	00	01	00	00	00	00	00	00	00	00	00	00	00
0000040	78	00	00	00	00	00	00	00	34	00	00	00	00	00	28	00
0000060	09	00	06	00	81	c4	00	00	00	00	00	00	2a	00	00	00
0000100	00	00	00	00	00	2e	73	79	6d	74	61	62	00	2e	73	74
0000120	72	74	61	62	00	2e	73	68	73	74	72	74	61	62	00	2e
0000140	72	65	6c	2e	74	65	78	74	00	2e	72	65	6c	2e	64	61
0000160	74	61	00	2e	62	73	73	00	00	00	00	00	00	00	00	00
0000200	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0000240	1f	00	00	00	01	00	00	00	06	00	00	00	00	00	00	00
0000260	34	00	00	00	06	00	00	00	00	00	00	00	00	00	00	00
0000300	04	00	00	00	00	00	00	00	1b	00	00	00	09	00	00	00
0000320	00	00	00	00	00	00	00	00	48	02	00	00	08	00	00	00
0000340	07	00	00	00	01	00	00	00	04	00	00	00	08	00	00	00
0000360	29	00	00	00	01	00	00	00	03	00	00	00	00	00	00	00
0000400	3c	00	00	00	08	00	00	00	00	00	00	00	00	00	00	00
0000420	04	00	00	00	00	00	00	00	25	00	00	00	09	00	00	00
0000440	00	00	00	00	00	00	00	00	50	02	00	00	08	00	00	00
0000460	07	00	00	00	03	00	00	00	04	00	00	00	08	00	00	00
0000500	2f	00	00	00	08	00	00	00	03	00	00	00	00	00	00	00
0000520	44	00	00	00	b0	04	00	00	00	00	00	00	00	00	00	00
0000540	04	00	00	00	00	00	00	00	11	00	00	00	03	00	00	00
0000560	00	00	00	00	00	00	00	00	44	00	00	00	34	00	00	00
0000600	00	00	00	00	00	00	00	00	01	00	00	00	00	00	00	00
0000620	01	00	00	00	02	00	00	00	00	00	00	00	00	00	00	00
0000640	e0	01	00	00	60	00	00	00	08	00	00	00	06	00	00	00
0000660	04	00	00	00	10	00	00	00	09	00	00	00	03	00	00	00
0000700	00	00	00	00	00	00	00	00	40	02	00	00	05	00	00	00
0000720	00	00	00	00	00	00	00	00	01	00	00	00	00	00	00	00
0000740	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0000760	00	00	00	00	00	00	00	00	00	00	00	00	03	00	01	00
0001000	00	00	00	00	00	00	00	00	00	00	00	00	03	00	03	00
0001020	00	00	00	00	00	00	00	00	00	00	00	00	03	00	05	00
0001040	01	00	00	00	00	00	00	00	00	00	00	00	00	00	03	00
0001060	03	00	00	00	04	00	00	00	00	00	00	00	00	00	03	00
0001100	00	58	00	59	00	00	00	00	02	00	00	00	01	02	00	00
0001120	04	00	00	00	01	02	00	00								
0001130																

FIG. 6.2 – Résultat de `od -tx1 exempleElf.o`

Chapitre 7

Figures Annexes

tttn	Mnémo	Condition	
0010	B,	Below	`(Eflag.C == 1)`
0011	AE	Above or equal	`(Eflag.C == 0)`
0100	E	Equal	`(Eflag.Z == 1)`
0101	NE,	Not equal	`(Eflag.Z == 0)`
0110	BE	Below or equal	`(Eflag.Z == 1) || (Eflag.C == 1)`
0111	A	Above	`(Eflag.Z == 0) && (Eflag.C == 0)`
1100	L,	Less than	`(Eflag.S != Eflag.O)`
1101	GE	Greater than or equal to	`(Eflag.S == Eflag.O)`
1110	LE,	Less than or equal	`(Eflag.S != Eflag.O) || (Eflag.Z == 1)`
1111	G	Greater than	`(Eflag.S == Eflag.O) && (Eflag.Z == 0)`

FIG. 7.1 – Définition du champ tttn des instructions conditionnelles

Opérations	S	Z	O et C
ADD	S = Rm	Z = !Rm... !R0	O = Sm.Dm. !Rm + !Sm. !Dm.Rm C = Sm.Dm + !Rm.Dm + Sm. !Rm
AND, XOR, OR	S = Rm	Z = !Rm... !R0	O = 0 C = 0
SUB	S = Rm	Z = !Rm... !R0	O = !Sm.Dm. !Rm + Sm. !Dm.Rm C = Sm. !Dm + Rm. !Dm + Sm.Rm
CMP	S = Rm	Z = !Rm... !R0	O = !Sm.Dm. !Rm + Sm. !Dm.Rm C = Sm. !Dm + Rm. !Dm + Sm.Rm

Légende

- Sm = bit de poids fort de l'opérande source
- Dm = bit de poids fort de l'opérande destination
- Rm = bit de poids fort du résultat, !P = 0 si P vaut 1, !P = 1 si P vaut 0.

FIG. 7.2 – Calcul des codes conditions du registre EFLAGS.

R8			AL	CL	DL	BL	AH	CH	DH	BH
R16			AX	CX	DX	BX				
R32			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
Code opération			0	1	2	3	4	5	6	7
Reg/op			000	001	010	011	100	101	110	111
Adresse effective	**Mod**	**R/M**	**Valeur de l'octet *ModR/M* en hexadécimal**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[–][–][1]		100	04	0C	14	1C	24	2C	34	3C
Dir32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
Depl8[EAX][3]	01	000	40	48	50	58	60	68	70	78
Depl8[ECX]		001	41	49	51	59	61	69	71	79
Depl8[EDX]		010	42	4A	52	5A	62	6A	72	7A
Depl8[EBX]		011	43	4B	53	5B	63	6B	73	7B
Depl8[–][–]		100	44	4C	54	5C	64	6C	74	7C
Depl8[EBP]		101	45	4D	55	5D	65	6D	75	7D
Depl8[ESI]		110	46	4E	56	5E	66	6E	76	7E
Depl8[EDI]		111	47	4F	57	5F	67	6F	77	7F
Depl32[EAX][3]	10	000	80	88	90	98	A0	A8	B0	B8
Depl32[ECX]		001	81	89	91	99	A1	A9	B1	B9
Depl32[EDX]		010	82	8A	92	9A	A2	AA	B2	BA
Depl32[EBX]		011	83	8B	93	9B	A3	AB	B3	BB
Depl32[–][–]		100	84	8C	94	9C	A4	AC	B4	BC
Depl32[EBP]		101	85	8D	95	9D	A5	AD	B5	BD
Depl32[ESI]		110	86	8E	96	9E	A6	AE	B6	BE
Depl32[EDI]		111	87	8F	97	9F	A7	AF	B7	BF

1. [–][–] indique qu'un octet *SIB* suit l'octet *ModR/M* (cf. section 5.1.4).
2. Dir32 indique le mode d'adressage direct mémoire : une adresse sur 32 bits suit.
3. Depl8 (resp. Depl32) indique qu'un déplacement signé sur 8 (resp. 32) bits suit l'octet *SIB* éventuel.

Légende Le corps du tableau donne en hexadécimal les différentes valeurs permises pour l'octet *ModR/M* quand *Mod* est différent de 11 en binaire. Les premières lignes de la table donnent la signification et la valeur du champ *Reg/Op*. On rappelle que les trois bits de ce champ servent à désigner le second opérande qui doit être un registre ou l'extension du code opération de l'instruction si l'instruction n'a pas de second opérande. Ils déterminent le choix de la colonne dans le corps du tableau.
La première colonne de la table intitulée "**Adresse effective**" donne la liste des 24 modes d'adressage de la mémoire. La seconde et la troisième colonne donnent la valeur des champs *Mod* et *R/M* : ils déterminent le choix de la ligne dans le corps du tableau. Chacune des valeurs du champ *R/M* permet de coder trois registres (par exemple, la valeur binaire 000 de ce champ permet de coder AL, AX, EAX). Lequel de ces trois registres est effectivement utilisé dépend du préfixe pour choisir entre EAX et AX, du code opération, ainsi que de l'attribut de taille pour choisir entre EAX et AL (voir figures 7.4).

FIG. 7.3 – Définition de l'octet *ModR/M* quand $Mod \neq 3$ (en décimal)

Format des instructions	Codage
ADD - Add	
registre1 vers registre 2	0000 000w : 11 reg1 reg2
registre2 vers registre 1	0000 001w : 11 reg1 reg2
mémoire vers registre	0000 001w : mod reg r/m
registre vers mémoire	0000 000w : mod reg r/m
immédiat vers registre	1000 00sw : 11 000 reg : donnée immédiate
immédiate vers mémoire	1000 00sw : mod 000 r/m : donnée immédiate
AND - Logical AND	
registre1 vers registre 2	0010 000w : 11 reg1 reg2
registre2 vers registre 1	0010 001w : 11 reg1 reg2
mémoire vers registre	0010 001w : mod reg r/m
registre vers mémoire	0010 000w : mod reg r/m
immédiat vers registre	1000 00sw : 11 100 reg : donnée immédiate
immédiate vers mémoire	1000 00sw : mod 100 r/m : donnée immédiate
CMP - Compare Two Operands	
registre1 avec registre2	0011 100w : 11 reg1 reg2
registre2 avec registre1	0011 101w : 11 reg1 reg2
mémoire avec registre	0011 101w : mod reg r/m
registre avec mémoire	0011 100w : mod reg r/m
immédiat avec registre	1000 00sw : 11 111 reg : donnée immédiate
immédiate avec mémoire	1000 00sw : mod 111 r/m : donnée immédiate
LEA - Load Effective Address	
adresse mémoire vers registre	1000 1101 : mod reg r/m
MOV - Move Data	
registre1 vers registre2	1000 100w : 11 reg1 reg2
registre2 vers registre1	1000 101w : 11 reg1 reg2
mémoire vers registre	1000 101w : mod reg r/m
registre vers mémoire	1000 100w : mod reg r/m
donnée immédiate vers registre	1100 011w : 11 000 reg : donnée immédiate
donnée immédiate vers mémoire	1100 011w : mod 000 r/m : donnée immédiate
OR - Logical Inclusive OR	
registre1 vers registre2	0000 100w : 11 reg1 reg2
registre2 vers registre1	0000 101w : 11 reg1 reg2
mémoire vers registre	0000 101w : mod reg r/m
registre vers mémoire	0000 100w : mod reg r/m
donnée immédiate vers registre	1000 00sw : 11 001 reg : donnée immédiate
donnée immédiate vers mémoire	1000 00sw : mod 001 r/m : donnée immédiate
SUB - Integer Substraction	
registre1 avec registre2	0010 100w : 11 reg1 reg2
registre2 avec registre1	0010 101w : 11 reg1 reg2
mémoire avec registre	0010 101w : mod reg r/m
registre avec mémoire	0010 100w : mod reg r/m
donnée immédiate avec registre	1000 00sw : 11 101 reg : donnée immédiate
donnée immédiate avec mémoire	1000 00sw : mod 101 r/m : donnée immédiate
XOR - Logical Exclusive OR	
registre1 vers registre2	0011 000w : 11 reg1 reg2
registre2 vers registre1	0011 001w : 11 reg1 reg2
mémoire vers registre	0011 001w : mod reg r/m
registre vers mémoire	0011 000w : mod reg r/m
donnée immédiate vers registre	1000 00sw : 11 110 reg : donnée immédiate
donnée immédiate vers mémoire	1000 00sw : mod 110 r/m : donnée immédiate

FIG. 7.4 – Codage des instructions à 2 opérandes

Format des instructions	Codage
CALL - Call Procedure	
Direct	1110 1000 : déplacement sur 32 bits
indirect registre	1111 1111 : 11 010 reg
indirect mémoire	1111 1111 : mod 010 r/m
Jcc - Jump if Condition is Met	
déplacement sur 8 bits	0111 tttn : déplacement sur 8 bits
déplacement sur 32 bits	0000 1111 : 1000 tttn : déplacement sur 32 bits
JMP - Unconditional Jump	
direct court	1110 1011 : déplacement sur 8 bits
direct long	1110 1001 : déplacement sur 32 bits
indirect via un registre	1111 1111 : 11 100 reg
indirect via un relais en mémoire	1111 1111 : mod 100 r/m
NOP - No Operation	1001 0000
POP - Pop a Word from the Stack	
Registre	0101 1 reg
Registre	1000 1111 : 11 000 reg
Mémoire	1000 1111 : mod 000 r/m
PUSH- Push Word from the Stack	
Registre	0101 0 reg
Registre	1111 1111 : 11 110 reg
Mémoire	1111 1111 : mod 110 r/m
donné immédiate	0110 10s0 : donnée immédiate
RET - return from function	1100 0011

FIG. 7.5 – Codage des instructions à 0 ou 1 opérande

Bibliographie

[1] X. Rousset de Pina. *Programmation en assembleur GNU sur des microprocesseurs de la gamme Intel* Ensimag/Telecom 1A. 12 janvier 2004.

[2] Amblard P., Fernandez J.C., Lagnier F., Maraninchi F., Sicard P. et Waille P. *Architectures Logicielles et Matérielles.* Dunod, collection Siences Sup., 2000. ISBN 2 10 004893 7.

[3] *Intel Architecture Software Developer's Manual*, volume 1 (*Basic Architecture*) and volume 2 (*Instruction Set Reference*) Intel, 1999. `http://www.intel.com/design/pentium/manuals/`

[4] *Executable and Linkable Format (ELF).* Tools Interface Standard (TIS). Portable Formats Specification, Ver 1.1. `http://www.skyfree.org/linux/references/references.html`

[5] Dean Elsner, Jay Fenlason and friends. *Using as, the* GNU *Assembler*. Free Software Foundation, January 1994. `http://www.gnu.org/software/binutils/manual/gas-2.9.1//`

[6] FSF. *GCC online documentation.* Free Software Foundation, January 1994. `http://gcc.gnu.org/onlinedocs/`

[7] Steve Chamberlain, Cygnus Support. *Using ld, the* GNU *Linker*. Free Software Foundation, January 1994. `http://www.gnu.org/software/binutils/manual/ld-2.9.1/`

[8] Linux Assembly `http://linuxassembly.org/`