Projet de programmation en langage C

IPSim - Simulateur d'un processeur mini-Pentium, compatible Linux/ELF

Ensimag Apprentissage 1ère année
Printemps 2013

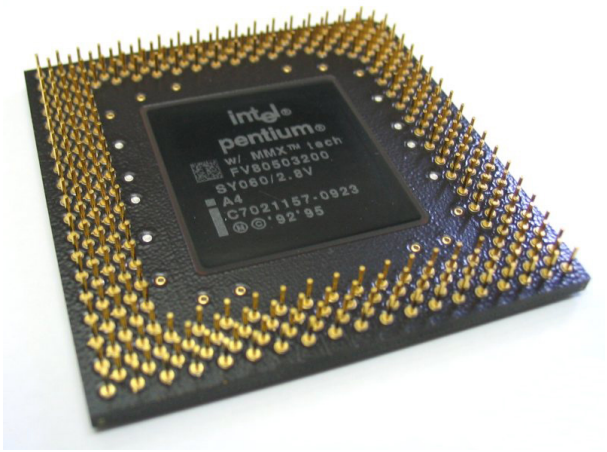

FIGURE 1 – Le processeur Pentium

Table des matières

Chapitre 1

Description générale du projet

1.1 Objectifs

L'objectif de ce projet est de réaliser sous Unix, en langage C, un simulateur d'une machine Pentium permettant d'exécuter et de mettre au point des programmes écrits pour le microprocesseur de la machine Pentium. En réalité, pour ce projet, le microprocesseur en question est un mini-Pentium, c'est-à-dire qu'il n'est capable d'exécuter qu'un sous-ensemble des instructions des Pentium. La configuration de cette machine est décrite dans la documentation générale du projet.

Concrètement, le simulateur sera un logiciel interactif en ligne de commande de type "shell". On le lancera avec la commande `ipsim`. Dans la suite du document, on appelle *interface utilisateur* le langage de commandes interactives qui permet à l'utilisateur de contrôler l'exécution du simulateur.

1.1.1 Le langage C

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 n'est pas forcément probable (ou 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 de ce cours 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)

1.1.2 Initiation au Génie Logiciel

Même si le projet est relativement court, vous serez confrontés à une base de code relativement volumineuse. Ceci implique des méthodes et outils sur la manière d'écrire et tester le code, de travailler à plusieurs (cf. sections 1.1.3, 2.1 et 2.3).

1.1.3 Développement piloté par les tests

Vous allez expérimenter une méthode de développement qui va vous permettre d'avancer rapidement : le développement piloté par les tests [1]. Le squelette de code fourni inclue une batterie de tests automatiques, et les tests sont lancés dans l'ordre dans lequel vous êtes supposés coder les fonctionnalités.

La base de tests se trouve dans le répertoire `TESTS`, et peut être lancée depuis le répertoire `SIMULATEUR` avec la commande `make check`.

En première approximation, le flot de développement peut donc ressembler à ceci :

```
while ! make check
    regarder la prochaine tâche dans err_log
    coder
    while ! make check
        debugger
    end while
    git commit
end while
```

Dans votre cas, vous n'avez pas à écrire les tests. Sur un vrai projet, il faudrait bien sûr ajouter l'écriture des tests (qu'il serait judicieux de placer en tout début de boucle sur le pseudo-code ci-dessus, c'est à dire *avant* de coder).

1.1.4 Toute cette documentation ! ? !

La documentation peut paraître volumineuse (il y a deux polycopiés), mais vous pouvez faire le projet sans l'avoir assimilé en entier : elle est beaucoup plus complète que ce dont vous avez besoin. En particulier, les chapitres sur les modes d'adressages et sur les instructions du Pentium ne devraient pas vous servir.

Il est par contre indispensable que vous ayez lu en détails les consignes, donc les chapitres 1 et 2 de ce document.

1.2 Simulateur de processeur

1.2.1 Principe du simulateur

Comme son nom l'indique, un simulateur est un logiciel capable de reproduire le comportement de l'objet simulé, dans le cas qui nous intéresse la machine Pentium lorsqu'elle exécute un programme P. "Reproduire le comportement" signifie plus précisément que l'on va définir pour notre simulateur une mémoire et des registres de taille semblables à celle de la machine Pentium, puis on doit réaliser l'évolution de l'état de cette mémoire et de ces registres selon les spécifications définies par le programme P. On doit obtenir les mêmes résultats que ceux que l'on obtiendrait avec une exécution sur une machine Pentium réelle mais pas forcément de la même façon : c'est le principe du "faire semblant" (par opposition au "faire comme"). Par exemple, dans notre cas, une instruction du microprocesseur Pentium pourra être simulée/réalisée par un appel de fonction du langage C, elle même compilée en une série d'instructions pour le microprocesseur effectuant la simulation (celui de la machine sur lequel est effectué le travail).

Dans la suite du document, on désigne sous le nom de *machine virtuelle*, la machine mini-Pentium simulée par le programme. On désigne sous le nom de *machine hôte*, la machine exécutant

1. `http://fr.wikipedia.org/wiki/Test_Driven_Development`

le programme du simulateur. Dans le cadre du projet, cette machine hôte peut être un Sun-Sparc ou elle-même un Pentium sous Linux.

1.2.2 Les programmes (au format ELF)

Un programme en langage machine mini-Pentium se présente sous la forme d'un ou plusieurs fichiers binaires relogeables au format ELF. Un tel fichier est issu de la traduction par un assembleur d'un fichier écrit dans le langage d'assemblage ASM décrit dans la documentation générale du projet. Les enseignants vous donnent un exécutable `ASSEMBLEUR/asm` qui réalise cet assemblage. Alternativement, vous pouvez essayer d'utiliser `gcc`, mais `gcc` peut coder certaines instructions avec un codage non considéré dans le projet (une même instruction peut se coder de plusieurs façon sur le processeur Pentium). Le mini-assembleur, quant à lui, reste sur le sous-ensemble du Pentium considéré dans le cadre du projet.

1.2.3 Chargement des programmes en mémoire

Pour être "exécuté" par le simulateur, un fichier binaire doit d'abord être recopié, on dit "chargé", dans la mémoire de la machine virtuelle en respectant les conventions qui sont décrites en section 3.1. Au cours de ce chargement, il y a une étape d'édition de lien à effectuer (il faut par exemple reloger les symboles de la zone DATA dans la zone TEXT). En fait, il est possible de charger un programme sous forme de plusieurs fichiers ELF relogeables : dans ce cas, le simulateur fait une édition de liens sur ces fichiers au moment du chargement.

Concrètement, on peut passer en argument les fichiers à charger sur la ligne de commandes de `ipsim`, et/ou charger les fichiers un à un de manière interactive (commande `lp`) après avoir lancé `ipsim`. Lors du chargement d'un fichier, le code de ce fichier est placé à la suite des fichiers déjà chargés (avec une phase d'édition de liens).

Il n'est pas possible de "décharger" un fichier de la mémoire ou de réinitialiser complètement la mémoire : le mieux est de sortir de `ipsim` et de recommencer une nouvelle session.

1.2.4 Exécution du programme

La simulation proprement dite d'une instruction du programme, c'est-à-dire l'interprétation du ou des octets consécutifs représentant le code instruction éventuellement suivi d'extensions et d'opérandes, consiste à décoder l'instruction et à réaliser à l'intérieur de la mémoire et des registres de la machine virtuelle les modifications qu'elle spécifie.

L'exécution du programme complet consiste évidemment à exécuter une à une l'ensemble des instructions codées dans l'ordre spécifié par le programme. Par défaut, cet ordre est séquentiel : on exécute les instructions dans l'ordre où elles arrivent dans le programme. Toutefois, certaines instructions (Jcc, JMP, CALL) induisent des ruptures de séquences à des endroits variés du programme. Le positionnement à l'intérieur du programme est assuré par un registre spécial appelé Extend Instruction Pointer (EIP). La gestion du retour des branchements (dans le cas de CALL/RET) est assurée par le registre "spécialisé" Extend Stack Pointer (ESP) qui adresse une zone de la mémoire dans laquelle l'instruction CALL empile implicitement les adresses programme de retour.

1.2.5 Interface utilisateur

En réalité, pour que l'utilisateur puisse mettre au point des programmes à l'aide du simulateur, il est nécessaire qu'il garde le contrôle de la simulation. C'est le rôle de l'interface utilisateur. Ainsi l'utilisateur pourra par exemple avoir le choix de taper la commande d'exécution des

programmes en lui adjoignant l'adresse de début du programme ou encore d'utiliser la commande de modification des registres pour affecter au compteur ordinal l'adresse de début du programme avant de lancer la commande d'exécution. Si le programme comporte une erreur qui peut être soit un code opération invalide, soit un opérande invalide, l'utilisateur doit reprendre la main afin de modifier son programme. Pour cela il doit disposer de commandes permettant de visualiser ou modifier le contenu de la mémoire et des registres. Il doit pouvoir exécuter le programme pas-à-pas (une instruction à la fois) ou de gérer des points d'arrêt dans le programme. La liste complète et le détail des commandes de l'interface sont donnés dans le chapitre 3.

1.2.6 Arrêt et entrée-sortie des programmes simulés

L'exécution d'un programme simulé doit s'arrêter lorsque le compteur ordinal passe par une adresse qui correspond à un point d'arrêt enregistré ou lorsqu'une erreur survient (instruction non reconnue, lecture ou écriture à une adresse invalide).

Pour permettre au programme de s'arrêter proprement, le simulateur fournit une routine `arret` que le programme simulé peut appeler. Pour appeler cette routine depuis un programme, il suffit de mettre un "`jmp arret`" dans le source assembleur du programme à simuler. L'édition de lien lors du chargement se chargera de mettre l'adresse adéquate de `arret`.[2]

Avec la même idée que la routine `arret`, le simulateur fournit des fonctions d'entrée-sortie de haut-niveau, appelable par le programme à simuler. Ces fonctions `affiche`, `afficheint` et `lisint` ont pour but de permettre respectivement l'affichage d'une chaîne, l'affichage d'un entier et la lecture d'un entier au clavier. Le code C correspondant à ces fonctions est donné ci-dessous.

```
void affiche(char *s) {
        printf("%s",s) ;
}

void afficheint(char *s, long d) {
        printf(s,d) ;
}

void lisint(long *res) {
        scanf("%ld",res) ;
}
```

1.2.7 Exécution "native" (sur la machine hôte)

Lorsque la machine hôte est un Linux-PC, le simulateur peut faire exécuter le programme à simuler directement par la machine hôte. A ce moment-là, le comportement n'est pas garantit en cas d'erreur à l'exécution (l'interface utilisateur ne reprendra pas la main), et l'exécution du programme ne s'arrête pas sur les points d'arrêt. Par contre, lorsque le programme appelle la routine `arret`, l'interface utilisateur reprend la main.

1.2.8 Tâches à réaliser

Vu l'ampleur du travail pour écrire un simulateur complet et le temps imparti, le projet démarre avec un squelette qui fait déjà la majorité du travail. En particulier, le squelette gère déjà :

– Le mini interpréteur de commande,

2. Il n'est donc pas possible pour le programme simulé de définir un symbole global `arret`.

– Le chargement des différentes zones du fichier ELF en mémoire,
– Les fonctions de bases nécessaires à l'exécution simulée (modes d'adressages, gestion des registres, ...),
– Un désassembleur, qui lit les instructions en mémoire, construit une structure de données plus facile à manipuler que les instructions codées, et permet de les afficher,
– Un certain nombre d'instructions, par exemple, `add` et `cmp`.

L'ordre de codage est imposé par l'ordre dans lequel passe la base de tests (cf. section sur le développement piloté par les tests dans l'autre polycopié). Dans les grandes lignes, le déroulement du projet sera le suivant :

– Pour commencer, un petit exercice : `little_endian.c`. Notre simulateur doit pouvoir tourner sur machines little-endian ou big-endian, et la représentation de la mémoire est celle de la machine simulée, donc little-endian. Pour permettre ceci, on a besoin des primitives pour lire et écrire un entier en codage little-endian, de manière portable.
– Ensuite, il faudra implémenter le code permettant la simulation des instructions de base qui ne sont pas déjà gérées par le squelette (par exemple, `sub`, `and`, ...). À ce stade, on ne peut pas utiliser de symbole en opérande.
– On implémentera alors la relocation dans le chargeur. C'est ce qui permettra d'utiliser correctement les symboles en opérande.
– Le chargeur étant complété, on reviendra sur des instructions qui ont besoin de la relocation pour fonctionner correctement (jmp, call, ...)
– Pour finir, il faudra implémenter la commande `nrun`, qui permet de lancer une exécution native (y compris au milieu d'une exécution simulée).

Les portions manquantes sont marquées explicitement dans le code par une macro `TODO` qui arrête l'exécution du programme avec le message d'erreur approprié. En général, la présence de cette macro est accompagnée d'un commentaire pour vous aider à implémenter la fonctionnalité. Si vous codez proprement, vous devriez avoir un peu moins de 200 lignes de code à écrire (mais chaque ligne de code devrait vous demander une certaine reflexion ...).

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

2.3.1 Configuration de 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 = Prénom Nom
        email = Prenom.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`, `gvim -f`,... mais évitez `gedit` qui vous posera problème ici). 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.

2.3.2 Récupération de la clé SSH pour l'accès au dépôt Git

Vous devez dans un premier temps récupérer la clé privée qui vous permettra d'accéder au dépôt Git sur telesun :

```
rsync --chmod=go-rwx \
   monlogin@telesun.imag.fr:/home/perms/moy/ldb-apprentissage/keys/'$LOGNAME' \
   ~/.ssh/id_projet_gl
```

Quelques remarques :

- Remplacez bien entendu `monlogin` par votre login sur telesun.
- La commande peut être entrée sur une seule ligne (sans les \), elle est coupée pour une meilleure lisibilité.
- Il faut bien des guillemets simples autour de `'$LOGNAME'`.
- Si vous travaillez sur `telesun`, vous pouvez faire une simple copie de fichier suivie d'un chmod, mais la commande ci-dessus marche aussi depuis vos machines personnelles.

Il faut ensuite dire à SSH d'utiliser cette clé. Cela peut se faire en créant un fichier `~/.ssh/config` sur la machine depuis laquelle vous vous connectez, avec le contenu suivant :

```
Host telesun.imag.fr
IdentityFile ~/.ssh/id_projet_gl
```

Si vous travaillez une machine autre que telesun, et que vous utilisiez déjà une clé pour vous connecter à votre propre compte telesun, vous pouvez au choix :

- utiliser `ssh-agent` et `ssh-add` sur votre clé existante. SSH va essayer toutes les clés actives dans l'agent avant d'essayer `id_projet_gl`.
- Ajouter plusieurs lignes `IdentityFile` derrière `Host telesun.imag.fr`, ssh va les essayer en séquence. Par exemple, si votre clé habituelle est `id_rsa` :

```
Host telesun.imag.fr
IdentityFile ~/.ssh/id_projet_gl
IdentityFile ~/.ssh/id_rsa
```

Pour plus d'information, lire l'article d'EnsiWiki sur les Clés SSH.

2.3.3 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 choisit l'autre. Par ailleurs, chacun ajoute son nom en haut du fichier, et enregistre le résultat.

2.3.4 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 peut visualiser ce qui s'est passé avec les commandes

```
gitk               # Visualiser l'historique graphiquement
```

et

```
git gui blame hello.c     # voir l'historique de chaque
                          # ligne du fichier hello.c
```

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 (e.g. 'git pull') before pushing again.  See the
'Note about fast-forwards' 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 récupérées. 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 (i.e. avec un éditeur de texte) 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 contenteront 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'archive 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.

git status pour voir où on en est

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.
- Ne faites pas de modifications autres que celles nécessaires (e.g. ajouter/supprimer des espaces, des lignes blanches, modifier l'indentation) sauf si vous êtes certains que vos collègues ne sont pas en train de modifier les mêmes fichiers. En particulier en utilisant un IDE : ne le laissez pas reformater votre code à tout va (mais formatez-le correctement du premier coup !).

(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 !)

Quand rien ne va plus ...

En cas de problème avec l'utilisation de Git

- Consulter la page `http://ensiwiki.ensimag.fr/index.php/FAQ_Git` sur EnsiWiki. cette page a été écrite pour le projet GL, mais la plupart des explications s'appliquent directement pour vous.
- Si vous êtes bloqués, demandez de l'aide à votre enseignant. En général, les problèmes sont faciles à résoudre si on s'en occupe tôt, et bien plus dur si on a laissé la situation s'empirer en s'échangeant des fichiers en dehors de Git.

Dans tous les cas, lire la documentation est également une bonne idée : `http://git-scm.com/documentation` !

Chapitre 3

Description des commandes du simulateur

Dans cette section, on donne la liste et les spécifications des commandes de l'interface utilisateur du simulateur. On s'attachera à ce que chaque commande contrôle le bon format des paramètres. En particulier, et à titre d'exemple, les adresses mémoires passées en paramètre doivent être des valeurs hexadécimales sur 4 octets. Ainsi, pour toutes les commandes faisant intervenir un accès en mémoire, tout dépassement doit être signalé par un message d'erreur et la main doit être rendue à l'utilisateur. De même, on contrôlera le bon format des paramètres relatifs aux registres. De plus, si le nombre de chiffres hexadécimaux des valeurs passées en paramètre est inférieur à celui spécifié, on complète ces valeurs par des zéros à gauche (sur les bits de poids forts). S'il est supérieur, on doit à nouveau renvoyer un message d'erreur et rendre la main à l'utilisateur. Vous pouvez fournir d'autres fonctionnalités que celles présentées ici. Mais, dans ce cas, il faut au moins fournir celles-là.

3.1 Visualiser les zones de la mémoire chargées

- Nom de la commande : dz (display zones)
- Syntaxe : *dz*
- Paramètres : néant
- Description :
  La mémoire de la machine virtuelle est segmentée en trois zones distinctes `.data`, `.text` et `.bss` (zone contenant les données non initialisées). La zone `.data` et la zone `.text` font 16KO, et la zone `.bss` fait 32KO. Lorsqu'on charge un fichier ELF en mémoire, les zones du fichier ELF sont placées à la première adresse disponible (modulo contraintes d'alignement) de chacune des zones correspondantes de la mémoire virtuelle.
  La commande `dz` affiche pour chacune des zones, l'adresse du début de la zone, et l'adresse de fin de chargement de la zone (c'est-à-dire, la première adresse disponible).
  La commande affiche aussi pour chaque zone, la liste des adresses correspondant à un nom dans les fichiers chargés.
  Les adresses mémoires utilisées sur la machine virtuelle sont celles de la machine hôte : une adresse de la machine virtuelle peut être utilisée sur la machine hôte pour désigner le même emplacement en mémoire. Cette convention permet de réaliser simplement l'exécution en mode natif.

3.2 Charger un programme

- Nom de la commande : lp (load program)
- Syntaxe : $lp \ < nom\_du\_fichier >$
- Paramètres :
  nom_du_fichier : nom de fichier Unix
- Description :
  Le fichier dont le nom est passé en paramètre doit être un fichier ELF relogeable pouvant éventuellement contenir des symboles externes. Les zones du fichier sont placées à la suite des zones correspondantes déjà chargées dans la mémoire virtuelle. Les relocations internes du fichier sont effectuées. Il en va de même pour les relocations sur les symboles externes du fichier déjà définis comme symboles globaux par les fichiers précédemment chargés, et pour les relocations sur les symboles externes des fichiers précédemment chargés, que ce fichier définit comme symboles globaux. La double définition de symbole global est interdite.

3.3 Visualiser la mémoire

- Nom de la commande : dm (display memory)
- Syntaxe : $dm \ < adresse > < nb\_octets >$
- Paramètres :
  adresse : valeur hexadécimale
  nb_octets : nombre entier d'octets à afficher
- Description :
  Cette primitive visualise sur la console le contenu de la mémoire dont les adresses sont spécifiées en paramètres. L'affichage se fera à raison de 4 octets par ligne séparés par des espaces (un octet sera visualisé par deux chiffres hexadécimaux.). Chaque ligne commencera par l'adresse hexadécimale de l'octet le plus à gauche de la ligne.

3.4 Visualiser le code assembleur

- Nom de la commande : dasm (display assembler)
- Syntaxe : $dasm \ < adresse > < nb\_instr >$
- Paramètres :
  adresse : valeur hexadécimale
  nb_instr : nombre entier d'instructions à afficher
- Description :
  Cette primitive visualise sur la console le contenu de la mémoire qui a été désassemblée. L'affichage indiquera l'adresse correspondant à chaque instruction.

3.5 Visualiser les registres

- Nom de la commande : dr (display register)
- Syntaxe : $dr \ [< nom\_reg >]*$
- Paramètres :
  nom_reg : nom(s) du (des) registre(s) à afficher. En l'absence de paramètre, on affiche tous les registres.
- Description :
  Cette primitive visualise sur la console de visualisation le contenu des registres de la machine

Pentium. Chaque registre sera visualisé sous la forme nom : valeur, valeur étant définie par 8 chiffres hexadécimaux.

3.6 Modifier une valeur en mémoire

- Nom de la commande : lm (load memory)
- Syntaxe : $lm \ < adresse > < valeur >$
- Paramètres :
  adresse : valeur hexadécimale
  valeur : valeur hexadécimale d'octet (1 ou 2 chiffres hexadécimaux)
- Description : Cette primitive écrit dans la mémoire, à l'adresse fournie en paramètre, la valeur fournie en paramètre. Si l'utilisateur ne fournit qu'un chiffre hexadécimal, on force les quatre bits de poids fort de l'octet écrit à 0.

3.7 Modifier une valeur dans un registre

- Nom de la commande : lr (load register)
- Syntaxe : $lr \ < nom\_registre > < valeur >$
- Paramètres :
  nom_registre : l'un des noms de registres 32 bits valides
  valeur : valeur hexadécimale sur 32 bits (1 à 8 chiffres hexadécimaux)
- Description : De façon analogue à la primitive précédente, cette primitive écrit la valeur donnée en paramètre dans le registre dont le nom est passé en paramètre. Si on passe un nombre de chiffres hexadécimaux insuffisant, la primitive complète par des 0 à gauche.

3.8 Exécuter à partir d'une adresse

- Nom de la commande : run
- Syntaxe : $run \ [< adresse >]$
- Paramètres :
  adresse : valeur hexadécimale (facultatif)
- Description :
  Cette primitive charge EIP avec l'adresse fournie en paramètre et lance le microprocesseur. Si le paramètre est omis, on se contente de lancer le microprocesseur, qui commencera son exécution à la valeur courante de EIP.
  Cette primitive doit rendre la main à l'utilisateur, lorsque EIP passe par un point d'arrêt enregistré, qu'une erreur survient (instruction invalide, lecture ou écriture à une adresse invalide) ou que le programme appelle `arret`.

3.9 Exécution pas à pas (ligne à ligne)

- Nom de la commande : s (step)
- Syntaxe : s
- Paramètres : néant
- Description :
  Cette primitive provoque l'exécution de l'instruction dont l'adresse est contenue dans le registre EIP puis rend la main à l'utilisateur. Si l'on rencontre un appel à une procédure, cette dernière s'exécute complètement jusqu'à l'instruction RET incluse. La main est alors

rendu à l'utilisateur sur l'instruction suivant l'appel.
Cette primitive doit rendre la main à l'utilisateur, lorsque EIP passe par un point d'arrêt enregistré ou qu'une erreur survient (instruction invalide, lecture ou écriture à une adresse invalide).

3.10 Exécution pas à pas (exactement)

- Nom de la commande : si (step into)
- Syntaxe : *si*
- Paramètres : néant
- Description :
  Cette primitive provoque l'exécution de l'instruction dont l'adresse est contenue dans le registre EIP puis rend la main à l'utilisateur. Si l'on rencontre un appel à une procédure, cette dernière s'exécute alors pas à pas.

3.11 Mettre un point d'arrêt sur une adresse

- Nom de la commande : sb (set breakpoint)
- Syntaxe : *sb* $< adresse >$
- Paramètre :
  adresse : valeur hexadécimale
- Description :
  Cette primitive met un point d'arrêt à l'adresse fournie en paramètre. Lorsque le compteur ordinal sera égal à cette valeur, l'exécution sera interrompue et l'utilisateur reprendra la main.

3.12 Supprimer un point d'arrêt

- Nom de la commande : eb (erase breakpoint)
- Syntaxe : *eb* $[< adresse >]$
- Paramètres :
- Description :
  Cette primitive ôte le point d'arrêt à l'adresse fournie en paramètre. Si le paramètre est omis, la primitive efface tous les points d'arrêt.

3.13 Visualiser les points d'arrêt

- Nom de la commande : db (display breakpoint)
- Syntaxe : *db*
- Paramètres : néant
- Description :
  Cette primitive affiche sur la console de visualisation l'adresse des points d'arrêt positionnés dans la mémoire.

3.14 Exécution sur la machine hôte

- Nom de la commande : nrun (native run)

- Syntaxe : *nrun* [< *adresse* >]
- Paramètres :
  adresse : valeur hexadécimale (facultatif)
- Description :
  Cette commande n'a un effet que si la machine hôte est un PC sous Linux. Si c'est le cas, elle fait exécuter le programme par la machine hôte, au lieu de l'exécuter dans la machine virtuelle.
  La commande commence par sauvegarder la valeur de tous les registres de la machine virtuelle sauf EIP dans ceux correspondant de la machine hôte. [1] Ensuite, si aucun paramètre n'a été fourni à `nrun`, on fait exécuter à la machine hôte l'instruction pointée par le EIP de la machine virtuelle. Si une adresse a été fournie en paramètre de `nrun`, on lance le programme sur la machine hôte à partir de cette adresse.
  L'interface utilisateur ne reprend la main que si le programme exécuté fait appel à la routine `arret` du simulateur. Dans ce cas, la valeur des registres de la machine virtuelle est indéterminée (on pourra par exemple les réinitialiser).

3.15 Aide

- Nom de la commande : `?`
- Syntaxe : ? [< *commande* >]
- Paramètres :
  commande : nom d'une commande ipsim.
- Description :
  Affiche l'aide concernant cette commande. Si le paramètre est omis, affiche la liste des commandes.

3.16 Quitter le programme

- Nom de la commande : ex (exit)
- Syntaxe : *ex*
- Description : Cette primitive provoque la fin du programme simulateur.

1. Indication pour l'implémentation : utilisez une routine en langage machine pour effectuer cette opération. Pour sauvegarder le registre EFLAG, on pourra utiliser l'instruction Pentium `popf`.

Chapitre 4

Exemple d'utilisation de `ipsim`

On va illustrer l'utilisation de `ipsim` sur un exemple. Ci-dessous, on commence par décrire le programme qu'on va simuler, puis on montre la simulation dans un deuxième temps. Cet exemple se déroule ici sur une machine PC-Linux.

4.1 Présentation du programme à simuler

Le programme à simuler calcule en fait "Fibonacci(20)". Il se compose de trois "modules". Le premier `lancement.s` est un module qui sert à initialiser la pile convenablement, et à appeler la fonction `main`. Le second `affichefib.s` contient la fonction `main`. Le troisième `fib_rec.s` contient la fonction récursive `fib_rec` qui calcule Fibonacci par un calcul récursif naïf.

Module `lancement.s` Essentiellement, ce module réserve un tableau de 8192 octets dans la zone `.bss`. Ce tableau sert à coder la pile. La routine `_start` initialise donc le registre pointant sur le sommet de pile `%esp` et le registre pointant sur le bloc local `%ebp`. Puis on appelle la fonction `main`, et enfin, on appelle la routine `arret` pour quitter proprement le programme.

```
.section .text
.globl _start
_start:
        movl    $debutpile, %esp
        movl    %esp, %ebp
        call    main
        jmp     arret
.section .bss
finpile:       .skip 8192
debutpile:
```

Module `affichefib.s` Cette fonction fait des entrées-sorties (appels aux fonctions `affiche` et `afficheint`) et appelle la fonction `fib_rec`, puis affiche son résultat.

```
.section .text
.affichenb:
        .string "fib(%d) = "
.afficheres:
        .string "%d\n"
.globl main
main:
/* PRELUDE FONCTION */
        pushl   %ebp
```

```
        movl    %esp, %ebp
        subl    $4, %esp         /* RESERV. VAR. LOCALE "x" SUR LA PILE */
/* DEBUT DU PROGRAMME */
        /* movl    $30, -4(%ebp) */     /* x <- 30, pour voir la lenteur de la simu */
        movl    $20, -4(%ebp)   /* x <- 20 */
/* INST afficheint(.affichenb,x); */
        pushl   -4(%ebp)
        pushl   $.affichenb
        call    afficheint
        addl    $4, %esp    /* x reste dans la pile */
/* CALCUL %eax <- fib_rec(x) */
        call    fib_rec
        addl    $4, %esp
/* INST afficheint(.affichenb,fib_rec(x)); */
        pushl   %eax
        pushl   $.affichenb
        call    afficheint
/* FIN FONCTION */
        movl    %ebp, %esp
        popl    %ebp
        ret
```

Module `fib_rec.s` Cette fonction calcule "Fibonacci(n)" où n est un paramètre entier lu sur la pile (à l'adresse "`8(%ebp)`" dans le corps de la fonction). Comme d'habitude le résultat est stocké dans `%eax`.

```
.section .text
/* On note "n" le par. de "fib_rec" */
fib_rec:
        pushl   %ebp
        movl    %esp,%ebp
        pushl   %ebx             /* sauvegarde de %ebx */
        cmpl    $1, 8(%ebp)
        jle     .cas1            /* si ("n" <= 1) alors jmp .cas1 */
        pushl   8(%ebp)          /* on met "n" en sommet de pile */
        subl    $1, (%esp)       /* on decr. le sommet de pile */
        call    fib_rec          /* calcul de fib_rec(n-1) */
        movl    %eax, %ebx       /* on sauvegarde le resultat dans "%ebx" */
        subl    $1, (%esp)       /* on decr. le sommet de pile */
        call    fib_rec
        addl    $4, %esp         /* on dépile "n-2" */
        addl    %ebx, %eax       /* %eax <- fib_rec(n-1)+fib_rec(n-2) */
        jmp     .finfib_rec
.cas1:
        movl    $1,%eax
.finfib_rec:
        popl    %ebx             /* restauration de %ebx */
        movl    %ebp, %esp
        popl    %ebp
        ret
.globl fib_rec
```

Assemblage des modules et test du programme On commence par assembler chacun des modules avec le programme `mini-as` fourni. On obtient ainsi `lancement.o`, `affichefib.o` et `fib_rec.o`.

Avant d'exécuter ce programme sous `ipsim`, il est préférable de vérifier que ce programme fonctionne normalement. On va donc d'abord le transformer un exécutable Linux. Pour cela, il faut fournir le code correspondant aux fonctions d'E/S du simulateur. On utilise donc le module suivant `librairie_linux.s`

```
/* Librairie Linux des fonctions d'E/S de "ipsim"
 *
 * affiche de type C:       void affiche(char *s)
 * afficheint de type C:    void afficheint(char *s, long d)
 * lisint de type C:        void lisint(int *res)
 */

.globl arret
.globl affiche
.globl afficheint
.globl lisint
arret:
        pushl   $0
        call    exit
afficheint:
        jmp printf
affiche:
        pushl 4(%esp)
        pushl $chaine
        call printf
        addl $8, %esp
        ret
lisint:
        pushl 4(%esp)
        pushl $lisintd
        call scanf
        addl $8, %esp
        ret
lisintd: .string "%d"
chaine: .string "%s"
```

On assemble ce dernier module et on obtient le fichier `librairie_linux.o`. On réalise alors l'édition de lien de tous ces modules pour en faire l'exécutable `fib`.

```
ld lancement.o affichefib.o fib_rec.o librairie_linux.o -o fib \
   --dynamic-linker /lib/ld-linux.so.2 -lc
```

On peut alors lancer `./fib` et observer le résultat :

```
fib(20) = 10946
```

4.2 Simulation sous ipsim

On lance la simulation avec la commande `ipsim lancement.o affichefib.o fib_rec.o`. On entre alors dans la session interactive de `ipsim`. On obtient :

```
IPSIM, le simulateur du microprocesseur Intel Pentium

Entrez une commande :
```

Ci-dessus, la phrase "`Entrez une commande :`" est l'invite de `ipsim` qui demande à l'utilisateur de saisir une commande. En tapant `dz` (affichage des zones), on obtient :

```
zone .data -- debut 8050d60 -- fin actuelle 8050d60
zone .text -- debut 8054d60 -- fin actuelle 8054df3
_start: 8054d60
.affichenb: 8054d74
.afficheres: 8054d7f
main: 8054d83
fib_rec: 8054dbc
.cas1: 8054de6
.finfib_rec: 8054dec
zone .bss -- debut 8058d60 -- fin actuelle 8059d60
finpile: 8058d60
```

On sait ainsi que `main` se trouve à l'adresse `8054d83`. On fait donc afficher les 10 premières instructions de `main`.

```
Entrez une commande : dasm 8054d83 10

main<0x8054d83>:
 8054d83:       pushl   %ebp
        [ ff f5 ]
 8054d85:       movl    %esp,%ebp
        [ 89 e5 ]
 8054d87:       subl    $0x4,%esp
        [ 83 ec 04 ]
 8054d8a:       movl    $0x14,0xfffffffc(%ebp)
        [ c7 45 fc 14 00 00 00 ]
 8054d91:       pushl   0xfffffffc(%ebp)
        [ ff 75 fc ]
 8054d94:       pushl   $0x8054d74
        [ 68 74 4d 05 08 ]
 8054d99:       call    afficheint<0x804bff8>
        [ e8 5a 72 ff ff ]
 8054d9e:       addl    $0x4,%esp
        [ 83 c4 04 ]
 8054da1:       call    fib_rec<0x8054dbc>
        [ e8 16 00 00 00 ]
 8054da6:       addl    $0x4,%esp
        [ 83 c4 04 ]
```

On pose ensuite un point d'arrêt en `8054da1` juste avant l'appel à `fib_rec`.

```
Entrez une commande : sb 8054da1
Ajout du point d'arret <8054da1>
```

On positionne le compteur ordinal à l'adresse de `_start`.

```
Entrez une commande : lr %eip 8054d60
```

On lance alors la simulation du programme.

```
Entrez une commande : run
Run program
fib(20) = point d'arret <8054da1>
```

Ici, le programme à affiché "`fib(20)=`" et l'exécution s'est arrêtée sur le point d'arrêt `8054da1`. En principe l'argument qui va être donné à `fib_rec` est en sommet de pile. On va modifier cette valeur. Pour cela, on commence par regarder l'adresse du sommet de pile

```
Entrez une commande : dr %esp
reg %esp = 8059d50
```

On vérifie la valeur en sommet de pile : 20 en décimal, c'est-à-dire 14 en hexadécimal, codé sur 4 octets.

```
Entrez une commande : dm 8059d50 4
 8059d50        14 00 00 00
```

On modifie cette valeur, en mettant 13 à la place (19 en décimal).

```
Entrez une commande : lm 8059d50 13
```

On relance ensuite le programme en mode natif.

```
Entrez une commande : nrun
Native Run
6765
```

Le programme affiche `6765` qui est bien la valeur de Fibonacci(19). On aurait eu ici exactement le même comportement (avec un calcul un peu plus lent) si on avait relancé l'exécution sur la machine virtuelle, avec la commande `run` au lieu de `nrun`.