Projet de programmation en langage C

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

Ensimag Apprentissage 1ère année
Printemps 2010

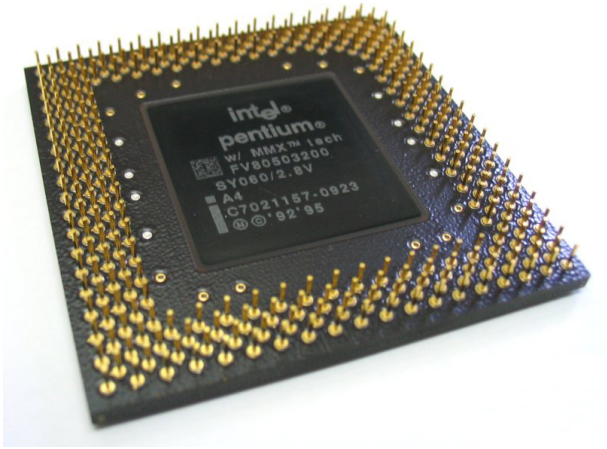


FIG. 1 – Le processeur Pentium

# Table des matières

# Chapitre 1

# Description générale du projet

## 1.1 Objectif du projet

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.

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

## 1.3 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.

## 1.4 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 2.1. 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.

## 1.5 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.

## 1.6 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 2.

## 1.7 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`. [1]

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) ;
}
```

## 1.8 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.

## 1.9 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,
- Le chargement des différentes zones du fichier ELF en mémoire,

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

– 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`.

Les tâches restantes pour vous sont :
– Une bibliothèque pour lire/écrire des entiers en little-endian, de manière portable,
– L'exécution simulée des instructions non gérées (par exemple, `sub`, `and`, ...
– La relocation de symboles au moment du chargement,
– Le mode d'exécution natif.

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 ...)

## 1.10 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[2]. 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).

## 1.11 Toute cette documentation ! ? !

La documentation peut paraître volumineuse, 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 du document général « Logiciel de base en C pour processeur Pentium et format ELF », et le chapitre 1 du présent document.

[2] `http://fr.wikipedia.org/wiki/Test_Driven_Development`

# Chapitre 2

# 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.

## 2.1 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.

## 2.2 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.

## 2.3 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.

## 2.4 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.

## 2.5 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.

## 2.6 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.

## 2.7 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.

## 2.8 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`.

## 2.9 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).

## 2.10 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.

## 2.11 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.

## 2.12 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.

## 2.13 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.

## 2.14 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).

## 2.15 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.

## 2.16 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 3

# 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.

## 3.1 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    $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(.affichere s,fib_rec(x)); */
        pushl   %eax
        pushl   $.affichere s
        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
```

## 3.2 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`.

# Chapitre 4

# Utilisation des fichiers fournis par les enseignants

Dans ce chapitre, nous présentons les fichiers fournis aux étudiants. On appelle ici "*module* `toto`" le couple formé des 2 fichiers `toto.h` et `toto.c`, où `toto.h` est un fichier d'en-tête, qui déclare les types de données et les fonctions exportées par le module `toto`, tandis que `toto.c` implémente ces fonctions. Ainsi, pour utiliser le module `toto` dans un programme C, il suffit de mettre dans le source un `#include "toto.h"`, et de lier le binaire avec `toto.o`. La section 4.7 de ce chapitre donne quelques conseils sur l'ordre dans lequel implémenter les différentes parties du simulateur.

## 4.1 Le module `dictionnaire`

Ce module fournit des fonctions "génériques" pour manipuler des dictionnaires associant des informations à des clefs (voir `dictionnaire.h`). Il fournit en particulier une fonction de recherche, qui permet de retrouver dans un dictionnaire l'information associée à une clef, et une fonction pour ajouter une nouvelle clef avec son information associée dans un dictionnaire. Le fichier `dictionnaire.c` est à compléter par les étudiants. Les types définis dans ce fichier suggèrent d'implémenter les dictionnaires comme des tables de hachage.

Le dictionnaire est utilisé dans le chargeur, pour associer différentes informations aux symboles, et pour associer des noms aux adresses. Il est aussi suggéré de s'en servir pour gérer les points d'arrêt : faire un dictionnaire dont les clefs sont des adresses sur-lesquelles il y a un point d'arrêt (l'information associée à chaque adresse peut être une valeur spéciale, indiquant un point d'arrêt). Le module `breakpoints` à compléter par les étudiants, donne un squelette de solution.

## 4.2 Le module `little_endian`

Ce module décrit comme lire et écrire des entiers de différentes tailles avec la convention little-endian, sur une machine qui ne suit pas forcément cette convention. Il est à compléter par les étudiants. Les fonctions de ce module sont utilisées par le chargeur. Elles sont à utiliser dès qu'on veut lire ou écrire des entiers dans la machine virtuelle.

## 4.3 Le module sim_ES

Ce module déclare les fonctions d'entrée-sortie (et la fonction `arret`) fournies par le simulateur aux programmes simulés. Lors du chargement des fichiers relogeables, les relocations sur les symboles désignant ces fonctions, sont effectuées avec les adresses (de la machine hôte) de ces fonctions. Ainsi, lors de l'exécution en mode natif, ces fonctions pourront être directement appelée par le programme simulé.

Ce module est complètement implémenté, et est utilisé par le chargeur. Il doit être utilisé par la machine virtuelle pour savoir si l'adresse pointée par EIP correspond à une de ces fonctions, et dans ce cas exécuter l'action correspondante.

## 4.4 Les modules sim_chargeur, dico_adr_symbol et symboles

Le module `sim_chargeur` définit en particulier la variable `memory` qui représente la mémoire centrale de la machine virtuelle et la fonction `sim_chargeur` qui permet de charger un fichier binaire relogeable au format ELF en mémoire.

Le module `sim_chargeur` est presque entièrement implémenté, à part la fonction `relocZone` qui doit être complétée par les étudiants. C'est cette fonction qui doit effectuer les relocations de symboles. Pour la relocation de symboles **externes**, on utilisera la fonction `addReloc` du module `symboles` qui se chargera d'effectuer la relocation dès que possible (éventuellement immédiatement), c'est-à-dire dès que l'adresse du symbole sera connue.

Le module `symboles` se charge donc de conserver l'adresse de définition des symboles globaux, les relocations sur les symboles globaux non encore définis, et d'effectuer les relocations enregistrées via `addReloc`. C'est la fonction `changeSymbolAdr` qui permet d'enregistrer l'adresse de définition des symboles globaux. Si on lui passe un symbole local en paramètre, elle se contente d'associer ce symbole à l'adresse correspondante via la fonction `ajouteAddressName` du module `dico_adr_symbol`.

Le module `dico_adr_symbol` permet d'associer un nom à des adresses. La fonction `getAddressName` de ce module permet de retrouver le nom d'une adresse.

## 4.5 L'interpréteur de commandes

Les fichiers correspondant à l'interpréteur de commandes sont `sh_sim.c`, `sh_sim.h`, `sh_com.c` et `help_sim.c`. Cet interpréteur est constitué d'une machine à états finis qui est susceptible d'appeler en fonction de son état courant une fonction (ou une sous-machine à états). Cette machine possède dans sa structure de base 19 états déclarés comme suit (dans le fichier `sh_sim.h`) :

```
/*------------------------------------------------------------------------*/
/* Definition des etats de la machine a etats                             */
typedef enum {
  READ_COM, DISP_ZONES, LOAD_PROG, DISP_MEMORY, DISP_ASM,
  DISP_REGISTER, LOAD_MEMORY, LOAD_REGISTER, NATIVE_RUN, RUN,
  STEP, STEP_INTO, SET_BREAKPOINT, ERASE_BREAKPOINT, DISPLAY_BREAKPOINT,
  HELP, EXIT, NL, ERROR
}t_etat; /* type etat de la machine a etats principale */
```

Cette machine peut recevoir des états supplémentaires si l'on souhaite augmenter le nombre de commandes disponibles pour le simulateur. Les 19 états énumérés ici suffisent à traiter toutes

les commandes décrites du chapitre 2. On associe à chaque état un pointeur de fonction défini comme suit :

```
/*-------------------------------------------------------------------------*/
/* Definition d'un type pointeur de fonctions associe a l'etat courant     */
typedef t_etat (*tm_etat)(); /* type machine a etats */
```

Ainsi à chaque fois que la machine à états entre dans un nouvel état, la fonction associée à cet état est exécutée. Les fonctions exécutant les commandes sont définies dans le fichier `sh_com.c` à l'exception de l'aide en ligne qui est codée dans le fichier `help_sim.c`. Le fichier `sh_com.c` est à compléter par les étudiants.

## 4.6 Autres fichiers fournis

Parmi les fichiers fournis, un fichier `Makefile` simplifié est proposé. L'enrichissement et la complétion de ce fichier reste donc à la charge des étudiants.

Le fichier `i386AsSyntax.h` propose un type de donnée pour représenter la structure abstraite des instructions du mini-Pentium. L'idée est que cette structure peut-être rempli par une fonction chargée de décoder une instruction. On peut alors faire afficher cette structure (désassemblage de l'instruction) ou simuler l'exécution de cette instruction. Grâce à cette structure de donnée, on peut donc bien séparer le décodage des instructions, de l'affichage au désassemblage et de l'interprétation sur la machine virtuelle.

Les fichiers `test_dico.c` et `test_chargeur.c` ont pour but de permettre le test du dictionnaire, et le test du chargeur. N'hésitez-pas à les modifier pour vos besoins.

## 4.7 Etapes de réalisation

Il est conseillé de construire le simulateur de manière incrémentale, et de tester et déboguer au fur et à mesure de la construction. Il est conseillé de commencer par `little_endian.c` qu'il faut tester avant de passer à la suite. Ensuite, il faut coder le dictionnaire générique (fichier `dictionnaire.c`) qui est utilisé par le chargeur. Testez ce programme avec le fichier `test_dico.c`.

Ensuite, le code du chargeur étant entièrement fourni hormis une partie du mécanisme de relocation qui reste à faire, on peut donc continuer, soit en finissant le code du chargeur, soit en commençant le décodage des instructions (prévoir un module à part pour le décodage des instructions). Parallèlement, on peut écrire la fonction qui permet l'affichage des instructions décodées. Cela permet ainsi de déboguer le décodage des instructions. Il est conseillé de ne commencer le module d'exécution des instructions, qu'une fois le module de décodage presque achevé (c'est-à-dire une fois que la majorité des instructions sont **correctement** décodées). Cela permet ainsi de déboguer plus facilement la partie "exécution". L'achèvement du chargeur (cf. section 4.4) peut s'effectuer de manière indépendante (on peut tester des programmes dans le simulateur sans relocation). Il préférable de réserver les "détails" de mise en œuvre de l'interface pour la fin : on peut améliorer l'interface du simulateur au fur et à mesure qu'on en a besoin pour le déboguage du reste du programme.