Simulation de Machine à Pile

Par Avihai Halioua et Térence de Saint Léger

1. Introduction

Ce projet a pour objectif de simuler une machine à pile fictive. Il comprend deux modules principaux :

1) **Compiler** : (Compilateur)

   Ce module convertit un programme écrit en langage assembleur en langage machine. Il vérifie la syntaxe des instructions et signale les erreurs avant de produire un fichier compilé.
   Ce module est sous-divisé en 2 parties :

   i) **Read assembly file** : (Lecture du fichier assembleur)
   Ce module permet de lire un fichier assembleur, de détecter la très grande majorité des erreurs de syntaxes contenues dans le fichier assembleur.
   Les informations de chacune des lignes sont extraites et stockées dans des structures pour faciliter leur traitement.

   ii) **Compile assembly file** : (Compilation du fichier assembleur)
   Ce module permet de compiler le fichier assembleur après avoir été lu.
   Les erreurs n'étants pas détectées lors de la lecture sont détecter dans ce module.

2) **Runner** : (Simulateur)

   Ce module lit et exécute le programme compilé, simulant ainsi le comportement d'une machine à pile en traitant les instructions de manière séquentielle.

2. Structure du projet

Le projet suit l'arborescence suivante (image par Chat-GPT) :

```
DeSaintLégerHalioua/
├── src/
│   ├── compiler.c
│   ├── compiler.h
│   ├── runner.c
│   ├── runner.h
│   └── sub/
│       ├── common.c
│       ├── common.h
│       ├── compile_assembly_code.c
│       ├── compile_assembly_code.h
│       ├── read_assembly_file.c
│       └── read_assembly_file.h
├── simulateur
├── docs/
│   ├── user.pdf
│   └── dev.pdf
└── hexa.txt
```

Nous avons choisi d'écrire l'intégralité de notre code en Anglais, en utilisant :

- la ***snake_case*** pour le nom des variables
- la ***camelCase*** pour le nom des fonctions et des structures

Nous avons utilisé ***Github*** pour pouvoir facilement gérer les changements dans le projet.

3. Détails d'implémentation

a) Assembleur et Compilation

L'assembleur est responsable de la conversion du code assembleur en langage machine. L'approche adoptée repose sur une analyse syntaxique rigoureuse pour détecter les erreurs précocement et garantir la cohérence du programme.
Le fichier texte est lu dans la fonction principale ***assemble*** qui fait appel à la fonction ***readAssemblyFile*** qui elle-même appel la fonction ***readAssemblyLine.***

L'idée a été de décomposer chaque ligne en quatre éléments :

- Numéro de la ligne
- Label (= étiquette)
- Instruction
- Paramètre (= donnée)

Ceux-ci ont été placé lors de la lecture dans un tableau à quatre éléments pour nous permettre une meilleure gestion de ces éléments et ceci nous évite de lire plusieurs fois le fichier.
La logique consiste alors à identifier les différents cas selon le nombre de mots lus :

- Si un seul mot est présent, il s'agit soit d'une étiquette, soit d'une instruction, selon que ce mot est suivi d'un caractère « : ».
- Si deux mots sont présents, il peut s'agir soit d'une étiquette et d'une instruction, soit d'une instruction et d'un paramètre, en fonction du fait que le premier mot soit suivi ou non d'un « : ».
- Enfin, si trois mots sont lus, il s'agit forcément d'une étiquette, d'une instruction et d'un paramètre.
- Pour finir, si quatre mots ou plus sont lus, une erreur est annoncée.

Nous avons mis en place la structure ***assemblyLine*** qui initialise ses champs par rapport aux valeurs du tableau précité ; et ce, car manipuler une structure nous semblait être une meilleure stratégie pour effectuer tous les tests d'erreurs de syntaxe, et de pouvoir séquencer le programme en plusieurs fonctions distinctes.
Voici la structure :

```
typedef struct {
    int   ID;           // ID of the line, refers to the line on which it was writen in the assembly file
    int   number;       // Line number of the instruction, may be modified
    char *label;        // Label of the line, may be NULL
    char *instruction;  // Instruction of the line, may be NULL
    char *parameter;    // Parameter of the instruction, may be NULL
} assemblyLine;
```

Le champ du label à été initialisé en testant au préalable sa validité grâce à la fonction ***isValidLabelName*** et en remplaçant le « : » par le caractère nul de fin de chaine.
Pour plus de détails concernant cette dernière fonction, se reporter à la section « Syntaxe accepté pour les fichiers assembleurs ».

Tout le fichier est ensuite lu ligne par ligne (en sautant les lignes vides et les commentaires qui sont les textes suivant le caractère « ; ») par la fonction ***readAssemblyFile*** , et est alloué pour chaque ligne un espace mémoire pour un élément de la structure ***assemblyLine***.

Nous avons initialisé un tableau de pointeurs d'éléments de la structure ***assemblyLine*** pour stocker chaque ligne du fichier. Celui-ci a une taille de base de 32 lignes au maximum.
Néanmoins, si le fichier texte en comporte plus, la taille du tableau est redimensionnée en réallouant de la mémoire à l'aide de la fonction *realloc*. Le nombre de ligne maximum est doublé. Ceci permet de ne pas allouer une trop grande quantité de mémoire dès le départ auquel cas, celle-ci resterais globalement inutilisé. Cela rend aussi le programme plus rapide.

De la même façon, nous avons initialisé la taille de base d'une ligne à 32 caractères. Dans le cas où une ligne en comporterait plus, ce nombre est doublé et la mémoire est réalloué (avec *realloc*), puis la ligne est relue.

Si une ligne n'est pas lue correctement, la mémoire allouée de cette ligne est alors libérée.

Chaque ligne est ensuite compilée en langage machine à l'aide de la fonction ***compileLine*** qui vérifie à chaque fois la validité de l'étiquette, du registre et/ou d'une constante (selon l'instruction) en appelant les fonctions ***hasValidLabel***, ***hasValidRegistry*** et ***hasValidConstant***.
Ces dernières vérifient entre autres que les instructions qui nécessitent une constante, un registre ou une étiquette en sont bien fourni ; des messages d'erreurs sont affiché le cas échéant.
La fonction ***compile*** prend en paramètre un tableau où chacun de ses éléments comporte une ligne du fichier (initialisé dans la fonction ***readAssemblyFile***), et effectue trois opérations principales :

b) Numérotation des lignes d'instructions

Les lignes qui ne comportent qu'une étiquette ne sont pas comptabilisé. Ceci permet de calculer les adresses correctement par la suite (notamment pour les sauts).
En effet, nous avons accepté les lignes qui ne comporte qu'une étiquette et dont l'instruction se trouve par exemple, à la ligne suivante. En bref, nous n'avons pas garder la numérotation des lignes du fichier texte mais nous avons fait notre propre numérotation.

c) Gestion des labels

Nous commençons par renvoyer une erreur si une étiquette est en double dans le fichier en parcourant pour chacune d'elle le tableau. Celle-ci doit en effet être unique.
Ensuite, chaque étiquette est remplacée par sa valeur (son adresse relative). Ce ne sont seulement les instructions ***jmp***, ***jnz*** et ***call*** qui sont autorisé à utiliser une étiquette.
Voici la formule pour calculer l'adresse relative :

adresseRelative = numéro de la ligne cible – numéro de la ligne actuelle – 1

Ainsi, par exemple, si une étiquette *start :* est défini en ligne 2 et qu'à la ligne 4, on a l'instruction *jmp start*, cette dernière est remplacé par *jmp -3* (retourner 3 lignes en arrière).

d) Compilation des instructions en hexadécimal

Chaque élément du tableau (correspondant à une ligne du fichier) est compilé à l'aide de la fonction ***compileLine*** et est rendu exécutable.

e) Simulateur (runner.c)

Le simulateur exécute le programme compilé en reproduisant le comportement d'une machine à pile. La fonction ***executeProcess*** orchestre la création d'un processus, son exécution via ***runProcess***, et la libération des ressources.
Nous avons initialisé quatre structures dans cette partie. Les voici :

```c
// Process memory
typedef struct {
    short  *registry;  // Registries (0 - <size-1>)
    int  size;      // Maximum number of registries
    int  sp;        // Top of the temporary value pile
} memoryRegistry;


// Process instruction
typedef struct {
    char  instruction;  // Instruction code of the line
    short parameter;    // Parameter of the instruction
} instructionLine;

// Process list of instruction
typedef struct {
    instructionLine **instructions;   // Instruction of the process
    int               nbInstruction;  // Number of instruction
    int               pc;             // Index of the next instruction to run
} programData;

// Process
typedef struct {
    memoryRegistry *memory;   // Memory of the process
    programData    *program;  // Program runned by the process
} process;
```

La première représente la mémoire d'un processus sous forme d'un ensemble de registres.
Le champ size au nombre maximal de registres disponibles et ***SP*** correspond au pointeur de pile, relatif à la position de la pile.
La deuxième représente une instruction individuelle du programme et prend son paramètre (adresse, registre, valeur immédiate…). On gère ainsi chaque instruction sous forme compacte.
La troisième représente le programme du processus, c'est-à-dire, la liste des instructions à exécuter.
Enfin, la quatrième représente un processus en cours d'exécution.

La fonction ***runProcess*** fonctionne en boucle, lisant et exécutant chaque instruction selon son code opération. A chaque cycle, elle lit l'instruction pointée par ***PC***. Celle-ci peut être de n'importe quel type (opération, appel de fonction, manipulation de la pile…).

Après l'exécution de chaque instruction, ***PC*** est incrémenté de 1 pour passer à l'instruction suivante. Certaines instructions de rupture de séquence (***jmp***, **jnz**, ***call*** *et* ***ret***) permettent de modifier de façon plus complexe ***PC***.

De la même façon, ***SP*** est mis à jour à chaque manipulation de la pile en utilisant notamment les fonctions ***popRegistry*** et ***pushRegistry***. Ces 2 fonctions permettent respectivement de dépiler le premier registre pour en récupérer sa valeur ainsi que d'empiler une valeur.

Le processus créé est ensuite détruit (libération de l'espace mémoire) grâce à la fonction ***killProcess***.

4. Syntaxe accepté pour les fichiers assembleurs

- Il est possible de modifier le nombre de registre en modifiant la macro située dans le répertoire "*./src/sub/common.h*" *:*

  #define MEMORY_REGISTRY_SIZE 5000

  Les registre mémoire vont de 0 au nombre de registre - 1.
  Accéder explicitement à un registre n'existant pas provoque une erreur à la compilation.
  La validité des registres est vérifiée avec la fonction ***isValidRegistry***.

- Il peut y avoir autant d'espaces et ou tabulations ('\t') que voulut avant, après et entre les valeurs d'une ligne d'instruction.

- Une étiquette doit respecter certaines conditions pour être considéré ainsi. Elles sont testées notamment dans la fonction ***isValidLabelName***. Voici ces conditions :

  La première lettre doit être dans les caractères 'A-Z', 'a-z' ou égal à '_'.
  Pour les autres caractères, les chiffres de 0 à 9 sont aussi acceptés.
  Enfin, le caractère « : » doit être placé juste après le nom de l'étiquette, sans mettre d'espaces entre les deux.

- La donnée éventuelle d'une instruction est une suite de chiffres entre 0 et 9 (=un nombre). Elle est vérifiée grâce à la fonction ***isValidNumber***. Les nombres signés sont évidemment accepté.

- Il est possible de mettre des commentaires avec le caractère « ; ». Tout ce qui suis ce caractère sera ignoré par le compilateur. Il est possible de mettre des commentaires après une ligne d'instruction ou sur une ligne vide.

- Les lignes vides, lignes avec seulement des espaces, tabulations et commentaires sont accepté dans le fichier texte. Elles seront simplement ignorées lors de la lecture grâce à la fonction ***isBlankString***.

- Une ligne peut ne comporter qu'une étiquette.

- Certaines erreurs sont détectées lors de l'exécution du code. Parmi celles-ci on retrouve :
  - Arriver à la fin du programme (ou à une ligne n'existant pas) sans avoir rencontré d'instruction ***halt***.
  - Accéder à un registre n'existant pas (généralement en empilant/dépilant une valeur depuis la pile de valeur)

- Ce projet est capable de gérer les fins de lignes encodées non seulement avec le caractère '\n' mais aussi avec les caractères '\r\n'.

5. Fichiers tests

Pour s'assurer du bon fonctionnement de notre machine à pile, nous avons réalisé plusieurs fichiers tests.

Parmi eux on retrouve les fichiers suivants (en vert sont les lignes de commentaire, ces programmes sont entièrement compatibles avec notre machine à pile) :

Fibo.txt

```
                read 1000           ; Lecture d'une valeur n (supposée >= 0)
                push 1000           ; On empile la valeur n pour la passer en argument de fibo
                call fibo_function  ; Calcul de fibo(n)
                pop 1000
                write 1000          ; Affichage de fibo(n)
                halt


; Fonction permettant d'inverser l'ordre des 2 valeurs en haut de la pile

swap_function:  pop 4999
                pop 4998
                pop 4997
                push 4998
                push 4997
                push 4999
                ret


; Fonction calculant le n-ième terme de la séquence de Fibonacci de manière récursive naïve
; L'argument n est passé en l'empilant juste avant d'appeler la fonction
; À la fin de la fonction, 1 seule valeur a été ajouté à la pile : fibo(n)

fibo_function:  call swap_function  ; Récupère la valeur n (située en dessous de l'adresse du call)
                dup
                push# 1
                op 3
                jnz fibo_return     ; Si n <= 0, alors renvoie n
                push# 1
                op 11               ; n--
                dup
                call fibo_function  ; Calcul de fibo(n-1)
                call swap_function
                push# 1
                op 11               ; n--
                call fibo_function  ; Calcul de fibo(n-2)
                op 10               ; fibo(n) = fibo(n-1) + fibo(n-2)
fibo_return:    call swap_function  ; Récupère l'adresse du call
                ret                 ; Retourne la valeur fibo(n) qui est placée en haut de la pile
```

Square.txt

```
; Lit 2 nombres x et y en entrée
; Affiche le carré des nombres z de [[ x ; y ]]
; Si x < y, affiche x^2, (x+1)^2, ... , (y-1)^2, y^2
; Si x > y, affiche x^2, (x-1)^2, ... , (y+1)^2, y^2
; Sinon, affiche juste x^2


        read 1000  ; Lit x (entier quelconque)
        read 1001  ; Lit y (entier quelconque)
        push 1000
        push 1001
        op 3       ; Compare x et y
        push# 2
        op 12
        push# 1
        op 11
        pop 1002   ; Définit le pas p à +1 si x <= y, -1 sinon
```

```
        push 1000  ; Initialise z à x

loop:   dup
        dup
        op 12      ; Calcule et ajoute à la pile z^2
        write 1    ; Affiche z^2
        pop 1      ; Retire de la pile z^2
        dup
        push 1001
        op 0
        jnz end    ; Si z == y, arrête la boucle
        push 1002
        op 10      ; Incrémente z du pas p
        jmp loop   ; Boucle

end:    halt
```

Dice.txt

```
; Lance un dé à n faces une infinité de fois jusqu'à obtenir la face n
; Affiche le résultat de chaque lancer
; Affiche le nombre total de lancer



        read 1000   ; Lit un nombre n (supposé > 0)
        push 1000
        push# 0     ; Initialise le nombre de lancer total à 0

loop:   push# 1
        op 10       ; Incrémente le nombre de lancer total de 1
        rnd 32767   ; Génère un nombre "aléatoire" x
        push 1000
        op 14       ; x = x mod n
        push# 1
        op 10       ; x++
        dup         ; On obtient un nombre x dans [[ 1 ; n ]]
        pop 1001
        write 1001  ; Affichage du résultat du lancer x
        push 1000
        op 1
        jnz loop    ; Si x != n, alors recommence

        pop 1001
        write 1001  ; Affiche le nombre total de lancer
        halt
```

6. Erreurs et informations de développements

Nous n'avons pas rencontré beaucoup de problèmes lors du développement de ce projet.

La majorité des problèmes rencontrés étaient dû à des erreur bête de programmation (du style un + au lieu d'un - ou l'oubli d'un incrément) ainsi qu'à quelques erreurs de segmentations.

Le seul élément nous ayant vraiment posé un problème est le débogage du projet.
En effet, nous avions seulement 1 fichier test à disposition (celui du sujet), donc beaucoup d'instructions n'étaient pas du tout testées.
Il nous a donc fallut créer nous même des fichiers de test afin de s'assurer du bon fonctionnement de notre machine à pile.
Nous avons vérifié nos résultats à la main (en simulant l'exécution du code sur papier). Cela nous a pris beaucoup de temps mais nous n'avons pas trouvé de solutions plus efficaces.

De plus, nous avons dû attendre d’avoir fini des gros morceaux de projet avant de pouvoir les essayer. Par exemple pour la compilation, nous avons eu besoin de développer tout le système avant de vérifier que le résultat donné était le bon, de même pour la simulation du programme.