Rapport pour le développeur

AIT ELDJOUDI Tamazouzt MOUCHTAKI Mamoune

February 2023

1 Introduction

Nous vous présentons dans les séctions qui suivent les détails de l'implémentation d'un programme écrit en langage C entièrement conçu par nous même dans le cadre d'un projet pédagogique. Il s'agit là d'un simulateur d'une machine fictive appelée machine à pile. Le programme est divisé en deux parties principales, une partie traduction et une partie simulation. Veuillez noter que toutes les fonctions présentes dans le programme ont été commentées avec beaucoup de détails dans le code source. C'est pourquoi, vous trouverez uniquement les informations pertinentes sur la conception du code et non des détails les différentes fonctions qui le composent.

2 Première partie du projet

2.1 Idée globale

Dans cette partie, il a été demandé de récupérer un fichier contenant un code source écrit en assembleur et de traduire ce dernier en langage machine sous réserve que la personne ayant écrit l'assembleur n'ait pas commis d'erreurs syntaxiques ou autre, auquel cas il ne faut pas générer le code machine mais plutôt afficher les erreurs à corriger.
Notre approche a été la plus simple possible. Nous commençons par parcourir le fichier assembleur ligne par ligne (étant donné qu'une ligne représente une instruction) tout en stockant chaque ligne dans un tableau de chaînes de caractère. Cela nous permettra de parcourir ce tableau afin de lire les différentes ligne du fichier. Une fois que nous avons récupéré une ligne, nous la séparons en mots et générerons un tableau contenant les mots de cette ligne. Par mot nous voulons dire des caractères séparés par un ou plusieurs espaces, une tabulation, un retour chariot et le caractère ':' qui signe la fin d'une étiquette.
Maintenant que nous avons construit un tableau contenant les mots de chaque lignes, il devient facile de détecter des possibles erreurs. En effet, ce que nous faisons consiste à regarder le nombre d'éléments du tableau de mots et d'en déduire la présence ou non d'une erreur. Par exemple, si le nombre de mots d'une ligne vaut 4 ou plus, nous savons directement qu'à cette ligne il y a une

erreur étant donné qu'une ligne est composée au maximum de trois éléments: une étiquette, une instruction et éventuellement une donnée. Dans le cas d'un tableau de taille correcte (comprise entre 1 et 3, le 0 n'étant pas inclu car nous traitons les lignes vides autrement), nous commençons à regarder le contenu des éléments du tableau:
Si une ligne contient trois mots, le premier doit nécessairement être une étiquette, si c'est le cas nous poursuivons avec le deuxième élément. Celui-ci doit faire partie des instructions ayant un argument (les tableaux des instructions ayant un argument et de celles qui n'en ont pas ont été préalablement définis). Une fois tous ces tests passés, nous vérifions le dernier élément qui doit respecter les normes du langage et de la machine, à savoir avoir une adresse qui ne dépasse pas le nombre 4999 par exemple.
Si une ligne contient deux mots, de la même manière nous regardons à quoi correspondent ces mots et nous signalons toutes les combinaisons incompatibles avec ce qui a été énoncé dans le projet.
Vous l'aurez compris, une ligne qui contient un unique mot doit nécessairement représenter une instruction ne prenant aucun argument, et rien d'autre.
Les lignes vides sont acceptées par notre programme, par une ligne vide nous voulons dire une ligne contenant que des espaces ou des tabulations ou des retour à la ligne. Lors du parcours, si nous tombons sur une telle ligne, nous décidons simplement de l'ignorer tout en utilisant un compteur qui les compte afin de bien calculer les adresses des étiquettes par la suite.

2.2 Déroulement de cette partie

La première étape consiste à récupérer les lignes d'un fichier et les stocker dans un tableau. Par la suite, nous appliquons la fonction qui découpe cette ligne en son étiquette, son instruction et sa donnée . Ces deux derniers tableaux sont les données principales de notre méthode, à présent il suffit de boucler sur les éléments du tableau de lignes, découper chaque ligne, analyser le contenu comme expliqué précédemment en faisant appel à plusieurs fonctions de vérifications syntaxiques et finir par retourner un booléen qui est mis à zéro dés qu'il rencontre une erreur dans une ligne (il n'est retourné qu'à la fin de la fonction, ce qui permet de continuer à détecter des erreurs jusqu'à la fin du parcours du tableau). Si le booléen renvoyé est à 1, nous générons le fichier héxadécimal en reparcourant le tableau de ligne en traduisant et en appelant des fonctions définies préalablement afin de calculer les adresses des étiquettes. Si il y a des erreurs de compilation, nous ne générons pas le fichier mais des messages d'erreurs s'affichent et l'éxécution du programme s'arrête là.

2.3 Particularités syntaxiques

Vous trouverez dans le document user.pdf l'integralité des règles syntaxiques à respecter en cas d'utilisation du simulateur, en plus des règles imposées par le langage assembleur et la machine à pile.

3 Deuxième partie du projet

3.1 Idée globale

Dans cette partie, totalement indépendante de la première, il s'agit de récupérer un fichier héxadécimal représentant un code machine et de l'éxécuter ligne par ligne. L'idée est encore une fois de récupérer les lignes de ce fichier héxa, de les stocker dans un tableau et ensuite de les éxecuter l'une après l'autre (ou pas...) en faisant appel aux fonctions qui correspondent aux instruction de la machine à simuler (jmp, call, ret...). La partie délicate de la simulation est de faire correctement les sauts (jmp, jpz, call) qui perturbent la succession des instructions et qui peuvent faire pointer le PC un peu partout dans le code. Il aura donc fallu faire attention à la considération des lignes vides dans les calculs vu que nous les acceptons à la compilation.

3.2 Déroulement de cette partie

Nous commençons par écrire les fonctions qui représentent chaque instruction, en recopiant exactement leurs définitions. Par la suite, nous définissons une fonction "executeLine" qui, comme son nom l'indique, éxécute une ligne. Cette fonction fait appel à la fameuse fonction de la première partie, qui la découpera en deux blocs, un qui représente sa donnée et l'autre le code de son instruction. Le fichier Héxa ayant une forme fixe, nous faisons beaucoup moins de vérifications comparé à la première partie. Une fois que nous savons éxécuter une ligne, nous bouclant sur la même fonction en lui mettant en argument une ligne du fichier, les lignes se trouvant dans un tableau. Notre éxécution s'arrête à deux conditions, lorsqu'elle croise un halt, auquel cas on affecte une valeur négative au PC et on l'utilise comme condition d'arrêt, ou lorsqu'elle arrive à la fin du fichier.

4 Appréciations et améliorations/corrections

Dans cette section, nous allons présenter les défauts du programme ainsi que des améliorations que nous n'avons pas pû nous même implémenter.
Notre approche répond parfaitement aux demandes du projet, en effet, c'est une approche qui ne présente aucune difficulté à être implémentée et reprise par un autre développeur, elle est claire, concise et donne naissance à un programme correct et robuste. Cependant, voici une liste d'améliorations et de constats:

4.1 Les failles du code et les améliorations à envisager

- Notre programme compile parfaitement sans aucun warning et s'execute aussi parfaitement. Cependant, le programme Valgrind nous signale quelques fuites de mémoire dû à une baisse de garde de notre part et que nous n'avons malheureusement pas eu le temps de nettoyer entièrement. Ce genre d'erreurs peuvent, dans des cas rares, nuire à l'éxécution d'un programme. Il faut donc

impérativement retrouver la source de chaque fuite et règler le problème. Mise à part ces quelques fuites mémoires, le programme ne contient aucune autre erreur en terme d'accès mémoire ou autre.

- On ne peut nier le fait que notre solution ne soit pas optimisée mémoriellement parlant. En effet, nous avons priviligé la simplicité et la clarté du programme, étant donné le type et la taille des fichiers que nous comptons parcourir. Comme amélioration, nous proposons de réfléchir à un moyen de ne pas perdre l'accès aux tableaux des mots d'une ligne afin de ne pas redécouper à chaque fois une ligne déjà traitée. Il est aussi possible d'envisager d'utiliser des structures ayant comme champs le tableau de mots ainsi que sa taille pour une ligne, et le tableau de lignes ainsi que sa taille pour chaque fichier. On peut aussi penser à fusionner les différentes fonctions de la première partie en faisant en sorte que le traitement soit fait en une fonction et un parcours de lignes, et donc en générant le fichier exa mais en le supprimant dès lors qu'une erreur est croisée. A vrai dire, nous avions peur qu'une telle proposition ne soit pas en accord avec l'énoncé qui demande explicitement de ne pas générer le fichier avant compilation.

- Nous ne faisons pas de suppositions concernant le nombre de lignes d'un fichier, mais nous en faisons concernant le nombre d'étiquettes, de mots d'une ligne et de caractères d'une étiquette. Le programme peut donc être généralisé à ce niveau là.

- On pourrait diviser le code source en plusieurs fichiers incluant des fichier headers afin de le rendre plus sécurisé et plus lisible, malgrés le fait qu'il ne présente déjà aucun problème de lisibilité, étant très bien indenté et très bien commenté.

4.2 Difficultés rencontrées

Nous n'avons pas rencontré de difficultés significatives lors de la répartition des tâches, étant donné que le projet était divisé en deux parties distinctes. Cependant, nous avons plusieurs fois dû effectué des recherches et consulté différents forums et sites, tels que StackOverflow, ce qui a permis d'améliorer nos compétences en matière de recherche et de filtrage des informations. Ce processus nous a également permis d'acquérir une plus grande maturité en matière de projets informatiques et projets de groupe.