N'GBALA YAO Azzi Yoann
SIDIBE Niamoye

LA MACHINE À PILE

Table des matières

I. Transformation d'assembleur à hexadécimal

La première partie du sujet consistait à transformer un programme écrit en langage assembleur dans un fichier texte à un programme écrit en hexadécimal assembleur.
Pour cela, nous avons créé un fichier header nommé transformation.h qui contient les prototypes de toutes les fonctions utilisées pour faire cette étape. Puis nous avons aussi défini les macros TAILLE_MAX et TAILLE_ETIQ qui valent respectivement 500 et 25. Elles représentent les tailles maximales d'un programme et d'une étiquette.

Dans le fichier transformation.c nous avons défini les différentes fonctions. Elles sont au nombre de 8 pour être plus exactes.
D'abord, nous avons la fonction ***conversion()*** qui prend en argument une chaîne de caractère qui correspond à une instruction et renvoie un entier qui correspond au code machine de l'instruction entrée en paramètre ou -1 si elle n'existe pas. Ensuite, nous avons la fonction ***detectionEtiquette()*** qui prend en paramètre une chaîne de caractère et renvoie 1 si elle correspond à une étiquette c'est-à-dire si elle contient le caractère ":".
Puis ***suppression()*** qui ne renvoie rien et prend en paramètre une étiquette et une chaine de caractère vide et recopie dans cette nouvelle chaîne, l'étiquette sans le ":".

Dans la fonction ***estDansTabEtiq() est*** pris en argument un tableau contenant les étiquettes et une chaine de caractère. Cette fonction vérifie si la chaîne est dans le tableau. Si oui, la fonction renvoie l'indice auquel il a été trouvé sinon la fonction renvoie -1.
La fonction ***estEntier()*** prend en paramètre une chaîne de caractère et vérifie si elle représente un entier elle renvoie 1 si oui ou 0 sinon. Nous définissons ensuite la fonction ***estDansTab()*** qui prend en paramètre soit le tableau d'entier de toutes les lignes où il y a des étiquettes ou le tableau des lignes où il y a les instructions **ret**, **halt**, **iPush** ou un **iPop**, un entier représentant la taille du tableau et l'entier dont on doit vérifier la contenance dans le tableau. La fonction rend 1 si oui et 0 sinon.
La fonction ***supSautDeLigne()*** ne renvoyant rien et prenant en argument une chaîne de caractère permet d'enlever les marques de sauts de ligne **\n** sous linux ou e**\r\n** sous windows CRLF.

Enfin nous définissons la fonction principale du fichier transformation.c. qui est ***transformationFichier().*** Cette fonction prend en paramètre une chaîne de caractère contenant le nom d'un fichier.
Elle parcourt une première fois le fichier en mode lecture pour détecter les étiquettes et les mettre dans un tableau à l'indice correspondant à la ligne ou l'étiquette se trouve. La fonction met aussi dans un tableau les lignes où se situent les instructions citées plus haut.
Elle parcourt encore une deuxième fois ligne par ligne le fichier. Chaque ligne est lue, mise dans une chaine de caractère où on enlèvera les sauts de ligne à l'aide la fonction ***supSautDeLigne()*** et finalement découpée mot par mot et qui seront mis dans un tableau de chaine de caractère nommé **mots.**

La fonction traduit alors en lisant d'abord le fichier mis en argument puis effectue des tests à l'aide de ***estDansTab()*** pour vérifier si la ligne contient une étiquette et/ou une instruction citée plus haut. Ensuite, elle vérifie si la ligne contient le bon nombre d'arguments en fonction de la taille du tableau **mots** et affiche l'erreur et renvoie -1 en cas d'erreur. S'il il n'y a pas d'erreur d'arguments, l'instruction est mise dans la fonction ***instrus()*** pour obtenir le code machine de l'instruction. Si le code renvoyé est -1, la fonction s'arrête en renvoyant une erreur. Si le code est différent de -1, le programme vérifie la conformité des arguments et mets l'instruction et sa donnée dans le tableau d'Instructions nommé **stockage**.
Lorsque tout s'est passé correctement, la fonction ouvre alors le fichier hexa.txt et traduit les instructions contenues dans le tableau stockage puis les recopie ligne par ligne puis renvoie 1.

II. Exécution du programme

La seconde tâche à faire consistait à récupérer le fichier contenant le programme écrit en langage machine puis à l'exécuter instruction après instruction.
Pour mener à bien cette tâche, nous avons créé un fichier header : instructions.h .
Il y a dans ce fichier, les structures **Memoire** et **Instructions.** Nous y avons aussi défini la macro TAILLE_MEMOIRE pour la taille de la mémoire qui vaut 4000 adresses.

La structure Memoire contenant un tableau d'entier de taille TAILLE_MEMOIRE et deux variables entières SP le pointeur de pile et PC le registre désignant la prochaine instruction à exécuter.
La structure Instructions quant à elle contient deux variables entières de type short nommé **instru** qui contient le code machine d'une instruction et **donnes** qui correspond à la donnée associée. Cette structure permet d'associer chaque instruction avec son argument donné dans le programme.

Ensuite, dans ce fichier sont mis les prototypes des fonctions qui vont servir à l'exécution du programme. Ces fonctions sont définies dans le fichier instructions.c.
D'abord nous avons la fonction ***queFaire()*** qui prend en argument un pointeur sur la mémoire et une Instructions nommé **ligne**. Elle vérifie l'égalité entre le champ instru de **ligne** et les instructions existantes. En cas d'égalité la fonction ***queFaire()*** fait appel à la fonction correspondante. Elle renvoie -1 en cas d'erreur et 1 dans le cas contraire.
Ensuite nous avons traduit les différentes instructions autorisées sont forme de fonctions ne renvoyant rien ou renvoyant -1 en cas d'erreur ou 1 dans le cas contraire. Chaque fonction (pop, read, push, op etc…) prend en argument un pointeur sur la mémoire et/ou une donnée. Elle applique aux registres SP et PC les modifications nécessaires (incrémentation, décrémentation etc.…). Nous avons aussi défini deux fonctions ***estVide()*** et ***estRempli()*** prenant en argument un pointeur sur la mémoire et vérifiant si la pile en mémoire est vide ou pleine dans le cas de l'utilisation des fonctions pop et push et push# nommé ici push1. Ces deux fonctions renvoient 1 si c'est vrai et 0 dans le cas contraire.

Dans le fichier main.c, la fonction main appelle la fonction ***transformationFichier()*** avec le fichier mis en argument dans le terminal. Si aucun fichier n'est mis en argument dans le terminal, la fonction s'arrête et renvoie 0. Elle stocke dans la variable entière test l'entier renvoyé par ***transformationFichier()*** s'il est égal à -1 la fonction renvoie 0 et s'arrête.
Lorsque la fonction ***transformationFichier()*** renvoie 1, on crée un tableau d'**Instructions** de taille TAILLE_MAX nommé **stockage** et une **Memoire** nommée **memoire** avec ses champs SP et PC initialisé à 0.
Ensuite, le fichier hexa.txt contenant le programme à exécuter écrit en hexadécimal est parcouru ligne par ligne. Les instructions et leurs arguments sont en même temps stockés dans le tableau stockage.

A partir de là on commence l'exécution du programme à l'aide d'une boucle while qui ne s'arrête que lorsque le programme est terminé dans ce cas on affiche « ---sortie du programme--- » ou lorsqu'il y a eu une erreur lors de l'exécution d'une instruction et affiche « Erreur !».

III. Difficultés rencontrées

Pour la réalisation de ce projet nous avons fait face à de nombreuses difficultés et/ou des impasses.
En effet, au début nous avions fait un programme très complexe nécessitant l'utilisation d'une structure liste mais nous nous sommes rendues compte au fur et à mesure qu'on pouvait proposer une solution plus simple pour répondre au sujet. De plus, notre choix d'utiliser fgets pour lire les lignes du programme a demandé qu'on fasse une fonction pour supprimer les marques de sauts ligne en distinguant les cas en fonction qu'on soit sur Linux ou Windows.
Une des plus grandes difficultés a été de synthétiser tous les cas d'erreurs possibles. Nous n'avions pas pensé à beaucoup de cas d'erreurs et c'est à travers les tests que nous avions pu voir certains oublis. Cependant la création de fichier à tester n'était pas si simple vu que nous n'étions pas habitués à ce type de langage assembleur. Heureusement qu'il existait un système de partage de fichiers test entre étudiants.

Réaliser ce projet a été une très bonne expérience pour nous et nous a appris beaucoup de choses qui nous serons utiles dans nos études. Il nous a permis de mieux comprendre le fonctionnement de la mémoire.
Nous avons appris l'importance de la compréhension et de l'analyse du sujet. En effet, il nous a fallu modifier plusieurs fois notre programme car nous n'avions pas bien lu ou compris certaines parties du sujet. Nous avons aussi appris l'importance de savoir se remettre en question et prendre quelques distances par rapport à ce qu'on est en train de faire lorsqu'on est bloqué. Et la leçon la plus importante est de ne pas avoir peur de poser des questions lorsqu'on ne comprend pas une notion malgré nos appréhensions. De plus, nous avons réalisé qu'il ne suffit pas que le programme fonctionne, il faut qu'il soit facile à comprendre et à faire évoluer. Pour finir nous voulions aussi dire que c'était assez gratifiant de voir qu'on a fait un code pour un but précis et le programme s'exécutait correctement.