PROJET D' ARCHITECTURE DES ORDINATEURS

Rapport de projet

Sujet:
LA MACHINE À PILE

Elaboré par:
Hania REZK
Sabana SURESH
Groupe de TD6

Encadrant:
Emmanuel Lazard

Université Paris Dauphine PSL – L2 MIDO 2022/2023

Remerciements

Nous tenous à remercier M.Lazard pour vos renseignements, vos conseils et le temps que vous nous avez consacré pour répondre à nos questions.
Nous vous remercions également car c'est grâce à vous et votre enseignement que nous avons les connaissances nécessaires et la capacité de coder en C afin de pouvoir effectuer ce projet.

SOMMAIRE

I- Introduction

Ce rapport est destiné à décrire en détail le processus de création d'un programme qui récupère un fichier texte contenant un programme en assembleur, qui le convertit en langage machine et qui simule son exécution.

Nous avons décidé de nous partager la tâche en deux en suivant l'ordre proposé par le sujet :

- Hania : création du fichier hexa.txt et traduction du fichier texte (en langage machine).
- Sabana : l'exécution du fichier hexa.txt en simulant le fonctionnement de la machine (langage assembleur)

On a décidé de travailler chacun de notre côté sur notre partie respective et ensuite de mettre en commun. Ensemble, nous avons donc essayé de voir si tout fonctionnait, s'il y avait des bugs, comment on pouvait les corriger,...

II- Implémentation

Le programme a été écrit en utilisant le langage de programmation C. Il utilise les fonctions de la bibliothèque standard de C pour ouvrir et lire les fichiers en entrée et en sortie. Il a été répartis en 3 fichiers sources:

-main.c : qui contient la fonction main et la première partie.

-fonctions.c: qui contient toutes les fonctions de gestion de listes chaînées nécessaires pour la première partie.

-partie2.c: qui contient la simulation du fonctionnement de la machine.

Pour la première partie du programme (la conversion), on utilise la fonction fgets pour lire chaque instruction et son paramètre du fichier en entrée, et on convertit cette instruction en langage machine en utilisant un ensemble de règles prédéfinies. Les instructions en langage machine sont ensuite écrites dans le fichier hexa.txt en utilisant la fonction fputs et des variables 'souschaine', 'argument' et 'adresse'.

Pour la deuxième partie du programme (la simulation), on commence par ouvrir le fichier Hexa.txt en mode lecture avec l'instruction 'fopen', après dans une boucle while on utilise la fonction 'fscanf' pour lire les instructions en langage machine depuis le fichier hexa.txt pour ensuite les stocker dans deux variables : 'instruction' et

'param', cette boucle s'arrête après avoir parcourue tout le fichier, ensuite on va comparer la variable 'instruction' avec plusieurs conditions à l'aide de l'instruction 'switch', s'il y a une égalité on exécute le bloc de code correspondant et on sort du switch à l'aide de l'instruction 'break'.
En général ces blocs de code vont soit stocker une valeur dans la mémoire de la structure Machine, soit afficher une valeur de la mémoire, soit incrémenter ou décrémenter le pointeur de pile de la machine, ou bien changer la valeur du pointeur d'instructions.

A la fin on affiche un message qui nous informe de la fin de la lecture du fichier hexa et on le ferme à l'aide de l'instruction 'fclose'.

III- Choix effectués

Le code de la première partie consiste à la récupération de l'instruction, argument et les étiquettes (si elles sont présentes) sur chaque ligne du fichier texte. Pour cela nous avons choisi de parcourir ce fichier texte une première fois pour lire toutes les étiquettes qui sont présentes et les sauvegarder dans des structures de liste chaînée, qui nous a paru la plus convenable puisqu'elle est assez facile à gérer (ajouter des éléments) et à parcourir. Cette structure nous a aussi permis de sauvegarder la ligne sur laquelle figure cette adresse qui serait assez importante pour la suite.

Une fois toutes les étiquettes récupérées, on reparcourt le fichier texte mais cette fois pour lire tous les mots sur chaque ligne. Chaque ligne est récupérée dans une variable 'buff'. Pour cela nous avons créé des chaînes avec des tailles assez grandes, dans lesquelles on va remplir les instructions et arguments lues à chaque itération de la boucle while(on n'oublierait pas le caractère nul à la fin de chaque variable).

Nous avons aussi fait le choix d'ajouter des variables 'indicateur(i)' qui nous indiqueront si jamais il y a une étiquette au début de chaque ligne. Cet indicateur nous permettra de traiter chaque étape de 2 manières : un cas où (indicateur==1) et donc avec une étiquette en début de ligne, et un cas où (indicateur==0), sans étiquette.

On commence alors l'écriture dans le fichier 'hexa.txt', il s'agit de comparer à chaque fois l'instruction récupérée dans la variable 'souschaine' à toutes les fonctions possibles grâce aux boucles 'if'. L'octet correspondant à l'instruction va être écrit en premier, ensuite on écrit l'argument qui serait transformé en valeur hexadécimale par l'intermédiaire de la fonction **fprintf.**

La création une structure 'Machine' permet la simplification du codage en facilitant

l'accès et la modification de ses variables, en plus ça permet de clarifier le code et de mieux l'organiser, cette structure a comme données un tableau appelé 'memoire' qui représente la mémoire de notre simulateur, un entier 'sp' qui représente le pointeur de pile ou (stack pointer), et un entier pc qui représente le pointeur d'instruction ou (program counter).

La fonction fscanf a été choisie pour lire les instructions en entrée en raison de sa capacité à lire des données formatées à partir d'un fichier. Cependant, l'utilisation de fscanf peut être sujette à des erreurs si le format des données en entrée n'est pas correct, ce qui peut affecter la qualité des données générées en sortie. Le choix de l'instruction 'switch' et tout simplement en raison de la complexité des branchements de codes et de leur grand nombre.

La fonction 'switch' prend comme paramètre la valeur de l'instruction et la compare avec différentes valeurs comprises entre 0 et 14 chacune correspondant à une instruction de l'assembleur. Pour coder les différentes instructions du langage assembleur nous nous sommes servis principalement des consignes entre crochets; dans la plupart des cas nous les avons traduit en C.

Fonctionnalité : Le programme peut lire un fichier en entrée qui contient un programme en assembleur, il génère un fichier texte appelé hexa.txt qui stocke le programme en langage machine et après simule l'exécution de ce même fichier dans la console.

IV- Problèmes et bugs rencontrés

Une des difficultés de la première partie revient à détecter les erreurs qui seront détectées comme suit:

1- Le manque d'argument va être détecté si jamais la longueur de la chaîne 'argument' est nulle.

2- La présence d'argument en plus serait détectée si jamais la somme des longueur des chaînes 'argument' et 'souschaine' est différente de la longueur de la chaîne 'buff'.

3- Nous avons considéré aussi le cas où (argument>5000 ou argument<0) pour les fonctions sur la mémoire (push, pop, read, write), et le cas où l'argument dépasse les valeurs qui peuvent pas être transformé en hexadécimal. En effet d'après l'énonce, l'espace mémoire de travail a une taille de 5000 adresses.

4-Il ya aussi le cas où l'argument i pour l'instruction push# i, sort de l'intervalle de conversion en hexadécimale qui est $[-((16^8/2)-1);(16^8)/2]$.

5- La présence d'instruction inconnue: qui serait indiquer si jamais la variable 'existe'

reste égale à 0.

6- La répétition de deux étiquettes.

7-La référence, par les instructions sur les adresses, à une étiquette qui n'a pas été définie dans le fichier.

Pour ces cas là, nous avons décider de mettre un message d'erreur.

Nous avons également trouvé un peu difficile la gestion de la mémoire. Au début du projet on rencontrait beaucoup d'erreurs de segmentation du à l'usage incorrect des 'free'. Cependant, nous sommes arrivés à assimiler les 'free' pour les utiliser correctement.

En ce qui concerne la deuxième partie, on avait un souci par rapport aux instructions et leurs paramètres lus par le programme sous forme de chaînes de caractères (car en hexadécimal et l'hexa est écrit en chiffres et caractères), ce qui ne permettait pas de les utiliser en tant que paramètres pour le 'switch' ou en tant qu'indices pour les variables de la structure Machine. Donc la solution était de les convertir en décimal et les stocker dans de nouvelles variables pour pouvoir les utiliser correctement.

Au début, nous avons oublié de considérer le cas de la division (avec 'op') par un nombre nul, on avait également oublié le cas où la valeur entrée par l'utilisateur lors de l'instruction read sortirais de l'intervalle de conversion en hexadécimale qui est $[-((16^8/2)-1);(16^8)/2]$. Et pour ces cas là nous avons décidé de mettre un message d'erreur.

=> Ainsi on peut déduire qu'une grande partie des difficultés étaient alors dans la considerations de tous les cas qui peuvent générer des erreurs.

Un autre problème de cette deuxième partie était le fait qu'on avait d'abord écrit un programme qui récupère chaque ligne dans "hexa.txt" et exécute l'instruction sur cette ligne. Sauf que dans ce cas, nous n'utilisons pas le paramètre PC et donc les jmp et jpz ne fonctionnait pas correctement.
Nous avons alors décidé de créer une liste chainée dans laquelle on sauvegarde l'instruction, le paramètre et la ligne sur laquelle figure cette instruction. Pour simuler notre machine, on compare le paramètre PC avec toutes les lignes de la liste jusqu'à trouver l'instruction correspondante. À la fin de chaque itération on incrémente PC, et dans le cas des jmps et jpz on modifie la valeur de PC pour revenir à l'instruction souhaitée. Ceci nous a également permis de comprendre plus clairement comment fonctionne un processeur lors des sauts.

Pour l'instruction read, nous avons décidé de récupérer la valeur entrée par

l’utilisateur dans une variable du type long long, ensuite nous vérifions si jamais elle sort de l’intervalle défini pour les entiers non signés sur 4 octets: si c’est le cas la fonction renvoie, sinon cette valeur sera converti en type long int.

V- Améliorations et piste d'ouverture

Voici quelques pistes d'améliorations envisageables pour perfectionner la machine:

• Considérer d'autres manières pour récupérer plus simplement les instructions et adresses sur chaque ligne du fichier texte , au lieu de remplir caractère par caractère des chaînes.

• Ajouter des vérification supplémentaire pour s'assurer que le fichier texte de l'utilisateur se termine toujours par l'instruction 'halt': donc renvoyer une erreur dans la phase de traduction (ceci a été fait lors de la simulation de la machine).

• Ajouter la possibilité de lire des fichiers en entrée et en sortie à partir de l'interface utilisateur.

• Éliminer d'éventuels variables qui peuvent ne pas être utilisées pour certains programmes en langage assembleur. (par exemple dans notre code la variable 'arg' ne serait pas utilisée s'il s'agit d'un fichier texte avec que des 'jmp, jpz , halt, ret…' comme instructions. Cela ne génère pas d'erreurs ▪

Le programme peut servir de base pour des projets plus importants en matière de simulation de machines virtuelles, tels que la simulation d'un mini-ordinateur personnel. Il peut également être utilisé pour le développement d'un système embarqué.

VI- Conclusion

Concernant notre ressenti, nous sommes satisfaits de notre programme car nos objectifs ont été atteints. Le projet a été agréable à réaliser et nous avons su faire preuve d'efficacité, d'organisation et de vigilance pour mener à bien notre travail.

Ce travail était l'occasion d'appliquer à la fois nos connaissances d'architecture des ordinateurs et de programmation C. En effet, il mêlait ensemble plusieurs disciplines. Ce projet nous également a permit d'aller plus loin dans les possibilités du langage et d'acquérir de nouvelles connaissances.

VII- Annexes

Veuillez trouver ci-dessous les références des sites web et documentations utilisés pour l'élaboration du projet:

[1] Slide de cours "Programmation C" de Emmanuel Lazard – Dauphine 2022-2023
[2] Polycopié de C (H. Garreta)
[3] Slide de cours "Programmation C" de Emmanuel Lazard – Dauphine 2020-2021
[4] Polycopié de Architecture des ordinateurs (E. Lazard)
[5] Créez et initialisez des pointeurs (https://openclassrooms.com/fr/courses/19980-apprenez-a-programmer-en-c/7672176-creez-et-initialisez-des-pointeurs)
[6] Gestion de la mémoire (https://fr.wikibooks.org/wiki/Programmation_C/Gestion_de_la_m%C3%A9moire)