Lyna-Noor Slimani
Aymen Brahimi

Compte-Rendu

Projet

Ce projet consistait à créer un simulateur de processeur (CPU) en langage C. L'idée était de simuler le fonctionnement d'une machine qui comprend et exécute des instructions simples comme MOV, ADD, PUSH, etc. On devait programmer tout ce qui constitue un CPU virtuel : la mémoire, les registres, la pile, les instructions, les systèmes d'adressage, et plus tard, des segments dynamiques comme le segment ES. Le but était d'obtenir un système capable d'exécuter un petit programme à partir d'un fichier texte contenant des lignes d'instructions. Chaque exercice faisait construire une partie bien précise du système, et tous les fichiers .h et .c formaient ensemble le projet.

1. Description des structures principales et du rôle de chaque exercice

Le projet est structuré en plusieurs fichiers .h (headers) et .c (code source). Les fichiers .h servent à déclarer les fonctions et les structures qu'on utilise dans les .c, pour pouvoir réutiliser du code ailleurs. Par exemple, dans Exercice2.h, on inclut Exercice1.h car on a besoin de la HashMap définie dedans. Le #include sert donc à réutiliser ce qui a déjà été fait sans le recopier. On inclut aussi tous les headers dans main.c pour faire les tests. Sans les includes, les fonctions comme hashmap_get ou cpu_init ne seraient pas reconnues par le compilateur.

Dans l'exercice 1, on a créé une table de hachage (HashMap) qui permet de stocker des clés associées à des valeurs. C'est super utile car on peut, par exemple, dire que la clé "AX" est reliée à un int*. On utilise ça pour les registres, les segments, les constantes. Cette structure permet un accès rapide, c'est pour ça qu'on l'utilise tout le long du projet.

L'exercice 2 nous a permis de gérer la mémoire globale du CPU. On a défini une structure Segment pour représenter une portion de mémoire comme "DS" ou "ES", et une structure MemoryHandler qui contient l'ensemble de la mémoire du CPU sous forme d'un tableau de pointeurs. Ce gestionnaire de mémoire nous permet de créer des segments nommés, de les libérer, de stocker et de lire des valeurs dans ces segments.

Dans l'exercice 3, on a créé un parser. Un parser, c'est une fonction qui transforme une ligne de texte comme "MOV AX 5" en une structure de type Instruction avec un champ

pour le nom de l'instruction, un champ pour le premier opérande, et un champ pour le deuxième. Cela permet de rendre le texte exécutable par la suite.

Dans l'exercice 4, on a créé la structure CPU elle-même. Ce fichier contient toutes les données internes du CPU : ses registres (AX, BX...), ses flags (ZF, SF), son pointeur d'instruction (IP), son gestionnaire de mémoire, sa pile, etc. La fonction cpu_init initialise tous ces champs. Par exemple, on crée les registres avec des malloc, on les associe à leur nom dans le HashMap context, on initialise la pile à la fin de la mémoire.

Dans l'exercice 5, on a géré les différents types d'adressage. Une instruction comme MOV AX 5 ou MOV AX [DS:2] n'est pas traitée de la même façon. Il faut que le programme comprenne si l'opérande est une valeur directe, un registre, un accès direct à une case mémoire ou un accès indirect. Chaque fonction de cet exercice teste un type d'adressage. Le but était de tester chacune pour être sûrs que notre resolve_addressing choisit la bonne

L'exercice 6 contient les fonctions qui exécutent vraiment les instructions. Par exemple, handle_instruction prend une instruction comme MOV, récupère les pointeurs vers les deux opérandes (grâce à resolve_addressing) et copie la valeur de l'un dans l'autre. On a aussi géré les comparaisons (CMP), les sauts (JMP, JZ, JNZ), et les instructions de pile (PUSH, POP).

L'exercice 7 est spécialement dédié à la pile. On a codé push_value et int_pop_value pour empiler et dépiler des entiers. Le but est de gérer la pile manuellement avec un pointeur stack_pointer, en simulant le fonctionnement d'une vraie pile machine. On utilise les cases mémoire à la fin du tableau pour stocker les valeurs empilées.

L'exercice 8 ajoute un segment dynamique : ES. C'est un segment qu'on peut créer et détruire pendant le programme. Il est utile pour des opérations plus avancées. L'instruction ALLOC crée un segment de taille AX, avec une stratégie BX. Le FREE supprime le segment. Pour l'instant, on n'a pas encore réussi à faire fonctionner l'écriture dans ES avec l'adressage [ES:AX], même si l'ALLOC marche. Le segment est bien créé, l'adresse est correcte, le registre AX est bon, mais le programme ne reconnaît pas correctement cette opération.

2. Détail des jeux d'essais

Pour chaque exercice, on a conçu un ou plusieurs jeux d'essais dans le fichier main.c pour valider que le code fonctionne.

Pour l'exercice 1, on a testé qu'on pouvait insérer une valeur dans la table de hachage et la retrouver. Par exemple on a mis 42 dans une clé "cle" et on a vérifié que le get donnait bien 42. C'est important car tout le reste repose sur les accès aux registres ou segments par nom.

Dans l'exercice 2, on a testé qu'on pouvais créer un segment "DS" de 3 cases, et stocker une valeur dedans, puis la relire. J'ai aussi testé que si je dépasse la taille, ça ne marche pas, ce qui est normal.

Pour l'exercice 3, on a testé que quand je passe une ligne comme "MOV AX 15", le parser me renvoie une instruction avec les bons champs. C'est essentiel pour la suite car sinon aucune instruction n'est comprise.

Dans l'exercice 4, on a vérifié que cpu_init me donne bien un CPU complet, avec ses registres AX, BX, etc. On a testé que'on pouvais accéder à ces registres dans le contexte.

Dans l'exercice 5, on a testé chaque mode d'adressage : immédiat, registre, direct et indirect. Par exemple pour DS[2] j'ai stocké une valeur dans cette case et vérifié que je la retrouvais. J'ai aussi vérifié que *DS[2] accédait à la valeur pointée. On a aussi testé une version comme [ES:AX] pour l'exercice 8.

Pour l'exercice 6, on a simulé des instructions comme MOV AX 15, MOV BX AX, ADD BX 5 et j'ai vérifié que BX valait bien 20 à la fin. Cela montre que les opérandes sont bien résolues et que les valeurs sont bien copiées.

Pour l'exercice 7, on a 10, 20, 30, puis dépilé trois fois et vérifié que j'avais bien 30, 20, 10, ce qui confirme que la pile fonctionne en LIFO.

Pour l'exercice 8, on a mis AX = 1, BX = 0, puis on a appelé ALLOC et on a vérifié que ES valait l'adresse du début. Ensuite j'ai fait MOV [ES:AX] 10 pour tester l'écriture dans le segment. Mais ça ne fonctionne pas encore car segment_override_addressing retourne NULL. On a ajouté des printf pour comprendre mais on y arrive toujours pas

3. Analyse commentée des performances

La HashMap fonctionne très bien, elle permet un accès rapide aux registres et aux segments. La gestion de la mémoire avec les segments est claire et propre, on peut voir ce qu'on fait. Le traitement des instructions est fluide, on peut exécuter un programme ligne par ligne. Le seul point faible est dans l'exercice 8, où on a un problème dans la gestion du segment ES. Le segment est alloué, mais sa taille est incorrecte, du coup l'accès avec [ES:AX] est rejeté. Je suis en train de corriger ça.

4. Conclusion

Ce projet nous a permis de comprendre en profondeur le fonctionnement d'un processeur. On a vu comment chaque instruction est interprétée, comment la mémoire est gérée, comment fonctionne une pile, et surtout comment on assemble toutes ces structures ensemble. On a aussi appris à bien tester chaque partie de façon autonome, avec des jeux d'essais concrets. Il y a encore quelques détails à corriger, surtout dans le segment ES, mais globalement tout le reste fonctionne.