Simulateur de machine virtuelle - Développement

1 Introduction

Ce document a pour but de retracer la création du projet du simulateur de machine virtuelle, exposer les difficultés rencontrées lors de son développement,

2 Compilateur

Une documentation en ligne a été créée. Pour y accéder, suivez le lien ci-dessous :

https://exatio.github.io/Architecture-des-ordinateurs/ProjetDevCompiler

3 Simulation du microprocesseur

3.1 Structures utilisées

Pour la simulation du microprocesseur, nous avons opté sur l'utilisation des objets suivant :

1. **struct CPU :** Cette structure contient toutes les informations demandées pour simuler le microprocesseur. On y retrouve notament :
   - int PC : indice permettant de connaître la prochaîne instruction à récupérer en mémoire. Il est incrémenté dans la fonction *readMemory()*.
   - short register[32] : permet de simuler les 32 registres numéroté de 0 à 32. Chaque registre a une capacité de 2 octets.
   - int bitC, bitZ, bitN : comme indiqué dans le sujet, on a décidé de manipuler directement ces bits un par un au lieu de les mettre dans un registre spécifique.
2. **struct Instruction :** Cette structure contient toutes informations concernant une instruction en tant qu'attribut. L'utilisation de cette structure permet de facilité les appels de fonction et apporte une simplicité à la lecture du code.
3. **unsigned char Memory[65536] :** Ce tableau de char nous permet de simuler la mémoire. Chaque emplacement mémoire occupe 1 octet comme spécifié dans le sujet. Les nombres stockés sont non signé, les fonctions réinterprètent les nombres suivant le bit de poids fort du nombre stocké.
4. **Fonctions d'opérations :** Chaque opération a été représentée par une fonction portant son nom. On y passe en argument une structure CPU, une structure Instruction et si necesssaire un pointeur vers un tableau Memory.

3.2 Processus

Dans cette partie, il sera expliqué de manière détaillée le parcours d'un programme.

- **Etape 1 :** On récupère le fichier en contenant le programme en héxadécimale pour pouvoir le stocker dans la mémoire, c'est le rôle de la fonction *loadMemory()*.

  loadMemory() prend en argument un nom de fichier *char *file* et un pointeur vers un tableau de unsigned *unsigned char Memory[]*. La boucle suit la logique suivante :

- Récupération d'une ligne du fichier.
- Par paquet de 2 caractères, on réécrit le nombre hexadecimal en decimal (ex : $D8_{16} = 216_{10}$).
- On stocke le nombre décimal en mémoire. On procède de cette manière car ce nombre en hexadimal s'écrit sur 8 bits, soit la taille d'un emplacement mémoire.

exemple : On récupère l'instruction 'D8400000' puis on le sépare en paquet de 2 que l'on réécrit en décimal pour ensuite le stocker dans la mémoire : $D8_{16} = 216_{10}$ puis Memory[i] reçoit 216 et ainsi de suite.

- **Etape 2 :** Une fois que le programme a été chargé en mémoire, on commence l'exécution de ce dernier en récupérant dans la mémoire instruction par instruction, c'est le rôle de readMemory(), cette fonction permettant de reconstruire une instruction en lisant le contenu de la mémoire.

  readMemory() prend en argument *CPU *cpu* et *char* Memory* et renvoie un pointeur de type *Instruction*. Une instruction correspondant à 4 emplacements mémoire, on récupère les nombres précédemment stocker pour reconstruire l'instruction sous sa forme binaire. En effectuant les bon calculs, on retombe bien sur l'instruction voulue.

- **Etape 3 :** Les informations corcenant l'instruction ayant été chargé dans l'objet renvoyé par la fonction readMemory(), c'est par le biais de la fonction selectorInstruction() que le bon appel est effectué.

  selectorInstruction() prend en argument un pointeur vers une structure CPU, un autre vers une structure Instruction et un pointeur vers un tableau de char. On effectue un switch permettant de rediriger l'instruction vers la bonne fonction pour pouvoir correctement exécuter l'instruction. Le switch est effectué sur l'attribut *codeOp* de la structure Instruction.

Le programme boucle sur l'étape 2 et 3 pour l'exécution complète du code assembleur ! :)