Rapport Projet Phase 1 (LDP)

1 La fonction `saturated(new_value)`

Cette fonction retourne un `int` selon la formule de la documentation :

$$\text{saturated}(n) = \min(\max(n, 0), 2^{16})$$

Pour cela, j'utilise un système de `if` à 3 cas :

- Si le nombre est plus grand que la valeur maximale qu'un `uint16_t` peut avoir (65.536), alors retourner le maximum que peut contenir un `uint16_t`.
- S'il est plus petit que 0, alors retourner le minimum que peut contenir un `uint16_t`.
- Sinon, cela signifie que le nombre est bien dans les bornes d'un `uint16_t`, et on peut donc le retourner sans modification.

À noter que j'utilise la méthode `std::numeric_limits` qui peut me donner les valeurs `max` et `min` d'un type.

2 La fonction `parse_opcode(instr)`

Les opcodes (mots-clés) sont séparés des opérandes (arguments) par un espace. Je vais alors utiliser la fonction `find(' ')` pour trouver l'index de l'espace dans la chaîne de caractères de l'instruction, puis utiliser `substr(0, index du ' ')` pour retourner le sous-texte du début jusqu'à l'espace.

À noter que lorsqu'il n'y a pas d'espace dans la ligne (pour les commandes `PRINT` et `IFNZ`), la fonction `find` renvoie la valeur `-1`, considérée comme l'indice universel pour le dernier élément d'une chaîne de caractères.

3 La fonction `parse_operand(instr)`

Je commence par décomposer l'instruction avec la même méthode que dans `parse_opcode()`, sauf qu'ici je ne mets que l'index de l'espace en paramètre, ce qui me retourne uniquement l'opérande au format `string`. Il ne reste plus qu'à le passer dans la fonction `stoi` qui transforme la chaîne en `int`.

Enfin, le résultat est passé dans la fonction `saturated()` pour garantir une valeur contenue dans un `uint16_t`, comme imposé dans l'énoncé.

4 La fonction `exec(program_path)`

On définit `single_register` à 0 et on réserve une page mémoire pour `instruction` (la ligne qui va être lue) et `opcode` (le nom de l'instruction à exécuter). (Pas besoin de réserver une page pour les opérandes, car ils sont directement envoyés dans une fonction et stockés dans `single_register` à la fin.)

La méthode `std::ifstream instructions_file(program_path);` permet, avec `getline(instructions_file, instructio` de stocker les lignes une par une dans `instruction`. Étant donné que cette fonction renvoie `false` une fois que toutes les lignes ont été lues, on peut l'utiliser dans la condition d'une boucle `while` pour continuer à lire jusqu'à la fin. Cela permet d'exécuter les instructions **à la volée**, comme demandé.

Dans la boucle :

- On extrait `opcode` avec `parse_opcode(instruction)`.
- On l'évalue avec une série de `if-else` pour exécuter la bonne fonction.

Remarque : Un `switch-case` aurait été plus optimisé, mais il ne fonctionne qu'avec les types natifs (`int`, `char`, ...) et non avec les `string`.

5 La fonction `main(int argc, char* argv[])`

Le paramètre `argc` contient le nombre d'arguments passés au programme (normalement 2), et `argv[]` stocke ces arguments sous forme de tableau.

L'argument d'index 1 (le second) est passé à `exec(program_path)` pour que `ifstream` puisse accéder au bon fichier.

Remarque : J'ai ajouté une gestion d'erreur même si elle n'était pas demandée, afin d'éviter l'avertissement de variable `argc` non utilisée.