Rapport projet INFO-F105 - LDP1: phase 3

PROPS Thibaut

1 Questions

1.1 À quoi sert le dernier opérateur de `class Register` ?

L'opérateur `uint16_t()` sert à retourner la valeur de l'élément **privé** `_value` lorsqu'on utilise un objet de la classe `Register` comme un `uint16_t`.

```
Register a; // _value = 0 by default
uint16_t val = a; // Call operator uint16_t()
```

1.2 Essayez de le retirer puis recompilez votre programme. Y a-t-il quelque chose qui a changé ?

Oui, étant donné que `_value` est un élément **privé**, il n'y a aucun moyen d'y accéder directement. Cependant, si `Register` avait été une `struct` au lieu d'une `class`, nous n'aurions pas eu de problème d'accessibilité et il aurait été possible d'accéder directement à `_value`.

```
Register a; // _value = 0 by default
uint16_t val = a._value; // Call directly uint16_t _value
uint16_t val = a; // error: cannot convert 'Register' to 'uint16_t'
```

À noter qu'à présent, il n'est plus possible d'utiliser `a` sans l'attribut `._value`, car plus aucune surcharge ne permet d'obtenir un `uint16_t` depuis ce `Register`.

1.3 Que cela changerait-il si `Instruction` était une `class` plutôt qu'une `struct` ?

Si `Instruction` avait été une `class`, alors l'accès aux éléments `opcode` et `operands` n'aurait pas pu se faire directement via `Instruction->opcode`. Il aurait fallu créer deux `getter` pour pouvoir y accéder.

2 Choix d'implémentation

2.1 Instruction

Lors de la définition de `struct Instruction`, il faut définir deux attributs.

Le premier, `opcode`, est déterminé à l'aide de la fonction `parse_opcode`, qui retourne la bonne valeur de `enum Opcode` via des `if else`, car les `std::string` ne supportent pas les `switch case`.

Le second, `operands`, est un tableau de deux pointeurs vers des `struct Operand`. Pour cela, on vérifie d'abord, pour chaque opérande, s'il existe (c'est-à-dire si la position de l'espace précédent est définie et différente de -1). Si c'est le cas, on alloue dynamiquement un `Operand` (construit à partir de la chaîne correspondante) et on retourne son pointeur. Sinon, on retourne un pointeur nul (`nullptr`).

2.2 Memory

Le défi ici a été d'implémenter la surcharge de l'opérateur `[]` pour pouvoir lire ou modifier deux éléments `uint8_t` comme s'il s'agissait d'un seul `uint16_t`.

Pour remédier à ce problème, la fonction de surcharge `operator[]` retourne un objet de type `class uint16_tMemoryReference`. Cet objet est construit à partir du tableau de la mémoire et de l'adresse avec laquelle on souhaite interagir. Grâce à cela, on construit deux attributs `uint8_t*` qui pointent respectivement sur les cases des 8 bits de poids faible et de poids fort.

C'est ensuite cette classe qui surcharge les opérateurs `=` et `uint16_t()`. - Pour `operator=`, on découpe la valeur assignée en deux `uint8_t` afin de modifier les deux pointeurs dans le bon format. - Pour `uint16_t()`, on fusionne les deux valeurs pointées pour obtenir un `uint16_t`.

```
Memory mem(255);

mem[42] = 1337;
// uint16_tMemoryReference -> *lower8 = &mem[42], *upper8 = &mem[43]
// *lower8 (mem[42]) = 1337 % 256 (and 0xff mask is faster)
// *upper8 (mem[43]) = 1337 // 256 (>> 8 shifting is faster)

uint16_t val = mem[19];
// uint16_tMemoryReference -> *lower8 = &mem[19], *upper8 = &mem[20]
// val = *lower8 (mem[19]) + *upper8 (mem[20]) << 8 (<< 8 shifting is faster)
```

2.3 Register

Ici, étant donné que toutes les fonctions sont définies comme `constexpr`, on peut tout déclarer dans le fichier header.

J'ai défini ma fonction `saturated` (utilisée pour ne pas dépasser les bornes de `uint16_t`) dans un autre fichier, afin d'éviter la répétition, car je l'utilise aussi dans `Operand`.